链接程序和库指南

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

文件号码 819–7050–10
2006 年 10 月

版权所有 2005 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. 保留所有权利。

对于本文档中介绍的产品，Sun Microsystems, Inc. 对其所涉及的技术拥有相关的知识产权。需特别指出的是（但不局限于此），这些知识产权可能包含一项或多项美国专利，或在美国和其他国家/地区申请的待批专利。

美国政府权利－商业软件。政府用户应遵循 Sun Microsystems, Inc. 的标准许可协议，以及 FAR（Federal Acquisition Regulations，即"联邦政府采购法规"）的适用条款及其补充条款。

本发行版可能包含由第三方开发的内容。

本产品的某些部分可能是从 Berkeley BSD 系统衍生出来的，并获得了加利福尼亚大学的许可。UNIX 是 X/Open Company, Ltd. 在美国和其他国家/地区独家许可的注册商标。

Sun、Sun Microsystems、Sun 徽标、Solaris 徽标、Java 咖啡杯徽标、docs.sun.com、 Java 和 Solaris 是 Sun Microsystems, Inc. 在美国和其他国家/地区的商标或注册商标。所有 SPARC 商标的使用均已获得许可，它们是 SPARC International, Inc. 在美国和其他国家/地区的商标或注册商标。标有 SPARC 商标的产品均基于由 Sun Microsystems, Inc. 开发的体系结构。

OPEN LOOK 和 Sun™ 图形用户界面是 Sun Microsystems, Inc. 为其用户和许可证持有者开发的。Sun 感谢 Xerox 在研究和开发可视或图形用户界面的概念方面为计算机行业所做的开拓性贡献。Sun 已从 Xerox 获得了对 Xerox 图形用户界面的非独占性许可证，该许可证还适用于实现 OPEN LOOK GUI 和在其他方面遵守 Sun 书面许可协议的 Sun 许可证持有者。

本出版物所介绍的产品以及所包含的信息受美国出口控制法制约，并应遵守其他国家/地区的进出口法律。严禁将本产品直接或间接地用于核设施、导弹、生化武器或海上核设施，也不能直接或间接地出口给核设施、导弹、生化武器或海上核设施的最终用户。严禁出口或转口到美国禁运的国家/地区以及美国禁止出口清单中所包含的实体，包括但不限于被禁止的个人以及特别指定的国家/地区的公民。

本文档按"原样"提供，对于所有明示或默示的条件、陈述和担保，包括对适销性、适用性或非侵权性的默示保证，均不承担任何责任，除非此免责声明的适用范围在法律上无效。

061230@15821

目录

前言 17

1 Solaris 链接程序介绍 21

链接编辑 22

  静态可执行文件 22

运行时链接 23

相关主题 23

  动态链接 23

  应用程序二进制接口 24

  32 位环境和 64 位环境 24

  环境变量 24

  支持工具 25

2 链接编辑器 27

调用链接编辑器 28

  直接调用 28

  使用编译器驱动程序 28

指定链接编辑器选项 29

输入文件处理 30

  归档处理 30

  共享库处理 31

  与其他库链接 32

  初始化和终止节 36

符号处理 38

  符号解析 39

  未定义符号 44

  输出文件中暂定符号的顺序 48

定义其他符号 49
缩减符号范围 57
外部绑定 62
字符串表压缩 63
生成输出文件 63
标识硬件和软件功能 64
重定位处理 68
位移重定位 68
调试帮助 69

3 运行时链接程序 75
共享库依赖项 76
查找共享库依赖项 76
运行时链接程序搜索的目录 76
配置缺省搜索路径 79
动态字符串标记 79
重定位处理 80
重定位符号查找 81
执行重定位的时间 84
重定位错误 85
装入其他目标文件 86
延迟装入动态依赖项 87
提供 `dlopen()` 的替代项 89
初始化和终止例程 91
初始化和终止顺序 92
安全性 95
运行时链接编程接口 96
装入其他目标文件 97
重定位处理 99
获取新符号 104
调试帮助 109
调试库 109
调试器模块 114

4 共享库 119
命名约定 119
记录共享库名称 120
具有依赖项的共享库 123
依赖项排序 124
作为过滤器的共享库 125
生成标准过滤器 126
生成辅助过滤器 129
过滤组合 133
filtee 处理 133
性能注意事项 134
分析文件 134
基础系统 136
延迟装入动态依赖项 137
与位置无关的代码 137
删除未使用的材料 140
最大化可共享性 140
最小化换页活动 142
重定位 143
使用 `-B symbolic` 148
配置共享库 148

5 应用程序二进制接口与版本控制 153
接口兼容性 154
内部版本控制 154
创建版本定义 155
绑定到版本定义 163
指定版本绑定 168
版本稳定性 174
可重定位目标文件 174
外部版本控制 174
协调版本化文件名 175

6 支持接口 179
链接编辑器支持接口 179

调用支持接口 179
支持接口函数 180
支持接口示例 183
运行时链接程序审计接口 186
建立名称空间 187
创建审计库 187
调用审计接口 188
记录局部审计程序 188
审计接口函数 189
审计接口示例 194
审计接口演示 195
审计接口限制 196
运行时链接程序调试器接口 196
控制进程和目标进程之间的交互 197
调试器接口代理 198
调试器导出接口 198
调试器导入接口 207

7 目标文件格式 211
文件格式 211
数据表示形式 212
ELF 头 214
ELF 标识 218
数据编码 220
节 221
特殊节 234
COMDAT 节 238
组节 239
硬件和软件功能节 240
散列表节 241
移动节 243
注释节 246
重定位节 247
字符串表节 259
符号表节 260

Syminfo 表节 267
版本控制节 269
动态链接 274
程序头 275
程序装入（特定于处理器） 280
运行时链接程序 286
动态节 287
全局偏移表（特定于处理器） 299
过程链接表（特定于处理器） 299

8 线程局部存储 313
C/C++ 编程接口 313
线程局部存储节 314
线程局部存储的运行时分配 315
程序启动 315
创建线程 316
启动后动态装入 317
延迟分配线程局部存储块 317
线程局部存储的访问模型 318
SPARC: 访问线程局部变量 319
SPARC: 线程局部存储的重定位类型 326
32 位 x86: 访问线程局部变量 328
32 位 x86: 线程局部存储的重定位类型 332
x64: 访问线程局部变量 334
x64: 线程局部存储的重定位类型 338

9 Mapfile 选项 341
Mapfile 结构和语法 341
段声明 342
映射指令 345
段内节的排序 347
大小符号声明 347
文件控制指令 347
映射示例 347
Mapfile 缺省选项 349

内部映射结构 350

A 链接编辑器快速参考 353
静态模式 353
创建可重定位目标文件 353
创建静态可执行文件 354
动态模式 354
创建共享库 354
创建动态可执行文件 356

B 版本控制快速参考 357
命名约定 357
定义共享库的接口 358
共享库的版本控制 359
现有（非版本化）共享库的版本控制 360
更新版本化共享库 360
添加新符号 360
内部实现更改 361
新符号和内部实现更改 362
将符号迁移到标准接口 363

C 使用动态字符串标记建立依赖性 367
特定于硬件功能的共享库 367
减少 filtee 搜索 369
特定于指令集的共享库 370
减少 filtee 搜索 371
特定于系统的共享库 372
查找关联的依赖项 373
非绑定产品之间的依赖性 374
安全 377

D 链接程序和库的更新及新增功能 379
Solaris 10 1/06 发行版 379
Solaris 10 发行版 379

Solaris 9 9/04 发行版 380
Solaris 9 4/04 发行版 380
Solaris 9 12/03 发行版 380
Solaris 9 8/03 发行版 380
Solaris 9 12/02 发行版 381
Solaris 9 发行版 381
Solaris 8 07/01 发行版 381
Solaris 8 01/01 发行版 382
Solaris 8 10/00 发行版 382
Solaris 8 发行版 383
Solaris 7 发行版 383
Solaris 2.6 发行版 384

索引 385

表

表 2–1 CA_SUNW_SF_1 标志组合状态表 66
表 5–1 接口兼容性示例 154
表 7–1 ELF 32 位数据类型 213
表 7–2 ELF 64 位数据类型 213
表 7–3 ELF 标识索引 218
表 7–4 ELF 特殊节索引 221
表 7–5 ELF 节类型 *sh_type* 225
表 7–6 ELF 节头表项：索引 0 229
表 7–7 ELF 扩展的节头表项：索引 0 230
表 7–8 ELF 节属性标志 230
表 7–9 ELF sh_link 和 sh_info 解释 233
表 7–10 ELF 特殊节 234
表 7–11 ELF 组节标志 239
表 7–12 ELF 功能数组标记 241
表 7–13 SPARC: ELF 重定位类型 252
表 7–14 64 位 SPARC: ELF 重定位类型 256
表 7–15 32 位 x86: ELF 重定位类型 256
表 7–16 x64: ELF 重定位类型 258
表 7–17 ELF 字符串表索引 260
表 7–18 ELF 符号绑定：ELF32_ST_BIND 和 ELF64_ST_BIND 262
表 7–19 ELF 符号类型：ELF32_ST_TYPE 和 ELF64_ST_TYPE 263
表 7–20 ELF 符号可见性 265
表 7–21 ELF 符号表项：索引 0 266
表 7–22 SPARC: ELF 符号表项：寄存器符号 267
表 7–23 SPARC: ELF 寄存器编号 267
表 7–24 ELF 版本依赖性索引 272
表 7–25 ELF 段类型 277
表 7–26 ELF 段标志 279

表 7–27 ELF 段权限 280
表 7–28 SPARC: ELF 程序头段（64 K 对齐） 282
表 7–29 32 位 x86: ELF 程序头段（64 K 对齐） 283
表 7–30 32 位 SPARC: ELF 共享库段地址示例 286
表 7–31 32 位 x86: ELF 共享库段地址示例 286
表 7–32 ELF 动态数组标记 288
表 7–33 ELF 动态标志 DT_FLAGS 295
表 7–34 ELF 动态标志 DT_FLAGS_1 296
表 7–35 ELF 动态位置标志 DT_POSFLAG_1 298
表 7–36 ELF 动态功能标志 DT_FEATURE_1 298
表 7–37 32 位 SPARC: 过程链接表示例 300
表 7–38 64 位 SPARC: 过程链接表示例 304
表 7–39 32 位 x86: 绝对过程链接表示例 308
表 7–40 32 位 x86: 与位置无关的过程链接表示例 309
表 7–41 x64: 过程链接表示例 311
表 8–1 ELF PT_TLS 程序头项 315
表 8–2 SPARC: 常规动态线程局部变量的访问代码 319
表 8–3 SPARC: 局部动态线程局部变量的访问代码 321
表 8–4 32 位 SPARC: 初始可执行的线程局部变量的访问代码 323
表 8–5 64 位 SPARC: 初始可执行的线程局部变量的访问代码 324
表 8–6 SPARC: 局部可执行的线程局部变量的访问代码 325
表 8–7 SPARC: 线程局部存储的重定位类型 326
表 8–8 32 位 x86: 常规动态线程局部变量的访问代码 328
表 8–9 32 位 x86: 局部动态线程局部变量的访问代码 329
表 8–10 32 位 x86: 初始可执行的、位置无关线程局部变量的访问代码 330
表 8–11 32 位 x86: 初始可执行的、位置相关线程局部变量的访问代码 330
表 8–12 32 位 x86: 初始可执行的、位置无关动态线程局部变量的访问代码 331
表 8–13 32 位 x86: 初始可执行的、位置无关线程局部变量的访问代码 331
表 8–14 32 位 x86: 局部可执行的线程局部变量的访问代码 331
表 8–15 32 位 x86: 局部可执行的线程局部变量的访问代码 332
表 8–16 32 位 x86: 局部可执行的线程局部变量的访问代码 332
表 8–17 32 位 x86: 线程局部存储的重定位类型 333
表 8–18 x64: 常规动态线程局部变量的访问代码 334
表 8–19 x64: 局部动态线程局部变量的访问代码 335
表 8–20 x64: 初始可执行的线程局部变量的访问代码 336
表 8–21 x64: 初始可执行的线程局部变量的访问代码 II 336

表 8–22 x64: 局部可执行的线程局部变量的访问代码 337
表 8–23 x64: 局部可执行的线程局部变量的访问代码 II 337
表 8–24 x64: 局部可执行的线程局部变量的访问代码 III 338
表 8–25 x64: 线程局部存储的重定位类型 338
表 9–1 Mapfile 段属性 342
表 9–2 节属性 345

图 1–1	静态或动态链接编辑	22
图 3–1	单个 `dlopen()` 请求	101
图 3–2	多个 `dlopen()` 请求	102
图 3–3	具有公共依赖项的多个 `dlopen()` 请求	103
图 6–1	rtld-debugger 信息流程	197
图 7–1	目标文件格式	212
图 7–2	数据编码 ELFDATA2LSB	220
图 7–3	数据编码 ELFDATA2MSB	221
图 7–4	符号散列表	242
图 7–5	注释信息	246
图 7–6	注释段示例	247
图 7–7	ELF 字符串表	260
图 7–8	SPARC: 可执行文件（64 K 对齐）	281
图 7–9	32 位 x86: 可执行文件（64 K 对齐）	282
图 7–10	32 位 SPARC: 进程映像段	284
图 7–11	x86: 进程映像段	285
图 8–1	线程局部存储的运行时存储布局	315
图 8–2	线程局部存储的访问模型和转换	319
图 9–1	简单的映射结构	351
图 C–1	非绑定依赖项	373
图 C–2	非绑定共同依赖性	375

前言

在 Solaris™ 操作系统 (Solaris Operating System, 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，并且可以借助于运行时链接程序 `ld.so.1`(1) 来执行这些目标文件。本手册适用于需要更加全面地了解使用 Solaris 链接程序过程中所涉及的概念的工程师。

注 – 此 Solaris 发行版支持使用以下 SPARC® 和 x86 系列处理器体系结构的系统：UltraSPARC®、SPARC64、AMD64、Pentium 和 Xeon EM64T。支持的系统可以在 http://www.sun.com/bigadmin/hcl 上的《Solaris 10 Hardware Compatibility List》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” 指出了有关 AMD64 或 EM64T 系统的特定 64 位信息。
- “32 位 x86” 指出了有关基于 x86 的系统的特定 32 位信息。

若想了解本发行版支持哪些系统，请参见《Solaris 10 Hardware Compatibility List》。

关于本手册

本手册介绍了 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库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。

目标读者

本手册适用于对 Solaris 链接程序感兴趣的程序员（从好学的初级用户到高级用户）。

- 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、使用以及有效地自定义库。
- 高级程序员（例如语言工具开发者）可以了解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。

本书的结构

第 1 章概述 Solaris OS 下的链接进程。本章适用于所有程序员。

第 2 章介绍链接编辑器的功能。本章适用于所有程序员。

第 3 章介绍代码和数据的执行环境和程序控制的运行时绑定。本章适用于所有程序员。

第 4 章提供共享库的定义，介绍它们的机制，并说明如何创建和使用它们。本章适用于所有程序员。

第 5 章介绍如何管理动态库提供的接口的演变。本章适用于所有程序员。

第 6 章介绍用于监视、在某些情况下用于修改、用于链接编辑器和运行时链接程序处理的接口。本章适用于高级程序员。

第 7 章是有关 ELF 文件的参考章节。本章适用于高级程序员。

第 8 章介绍线程局部存储。本章适用于高级程序员。

第 9 章介绍链接编辑器的 `mapfile` 指令，这些指令指定输出文件的布局。本章适用于高级程序员。

附录 A 概述最常用的链接编辑器选项，本附录适用于所有程序员。

附录 B 提供共享库版本控制的命名约定和原则，本附录适用于所有程序员。

附录 C 提供如何使用保留的动态字符串标记定义动态依赖性的示例，本附录适用于所有程序员。

附录 D 概述已添加到链接编辑器的新增功能和更新，并指示已添加这些新增功能和更新的发行版。

在本文档中，所有命令行示例均使用 `sh`(1) 语法。所有编程示例均采用 C 语言编写。

文档、支持和培训

Sun Web 站点提供有关以下附加资源的信息：

- 文档 (`http://www.sun.com/documentation/`)
- 支持 (`http://www.sun.com/support/`)
- 培训 (`http://www.sun.com/training/`)

印刷约定

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体	含义	示例
AaBbCc123	命令、文件和目录的名称；计算机屏幕输出	编辑 .login 文件。 使用 ls -a 列出所有文件。 machine_name% you have mail.
AaBbCc123	用户键入的内容，与计算机屏幕输出的显示不同	machine_name% **su** Password:
aabbcc123	要使用实名或值替换的命令行占位符	删除文件的命令为 rm *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *Class* 选项。 **注意**：有些强调的项目在联机时以粗体显示。
新词术语强调	新词或术语以及要强调的词	**高速缓存**是存储在本地的副本。 请**勿**保存文件。
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表列出了 C shell、Bourne shell 和 Korn shell 的缺省 UNIX® 系统提示符和超级用户提示符。

表 P–2 Shell 提示符

Shell	提示符
C shell	machine_name%
C shell 超级用户	machine_name#
Bourne shell 和 Korn shell	$
Bourne shell 和 Korn shell 超级用户	#

第 1 章

Solaris 链接程序介绍

本手册介绍 Solaris 链接编辑器和运行时链接程序的操作，以及链接编辑器操作的目标文件。Solaris 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) 串联并解释一个或多个输入文件中的数据。这些文件可以是可重定位目标文件、共享库或归档文件库。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、可执行应用程序或共享库。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) 在运行时处理动态的可执行文件和共享库，从而将可执行文件和共享库绑定在一起来创建可运行进程。

共享库

共享库是链接编辑阶段的一种输出形式。共享库有时被称为**共享库**。共享库在创建强大灵活的运行时环境方面非常重要。

目标文件

Solaris 链接程序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

链接编辑可处理各种输入文件，这些文件通常由编译器、汇编程序或者 `ld(1)` 生成。链接编辑器会串联并解释这些输入文件内的数据以形成单个输出文件。虽然链接编辑器提供许多选项，但是生成的输出文件为以下四种基本类型之一：

- **可重定位目标文件**－可在随后的链接编辑阶段中使用的可重定位输入目标文件的串联。
- **静态可执行文件**－已解析所有符号引用的可重定位输入目标文件的串联。此可执行文件代表运行就绪进程。请参见第 22 页中的 "静态可执行文件" 。
- **动态可执行文件**－要求运行时链接程序进行干预以生成可运行进程的可重定位输入目标文件的串联。动态可执行文件可能仍需要在运行时绑定的符号引用。动态可执行文件通常具有一个或多个以共享库形式表示的依赖项。
- **共享库**－提供各种服务的可重定位输入目标文件的串联，运行时可能绑定到动态可执行文件。一个共享库可能依赖于其他共享库。

图 1–1 中显示了这些输出文件及其创建过程中使用的主要链接编辑器选项。

动态可执行文件和**共享库**通常共同称为**动态库**。本文档重点介绍动态库。

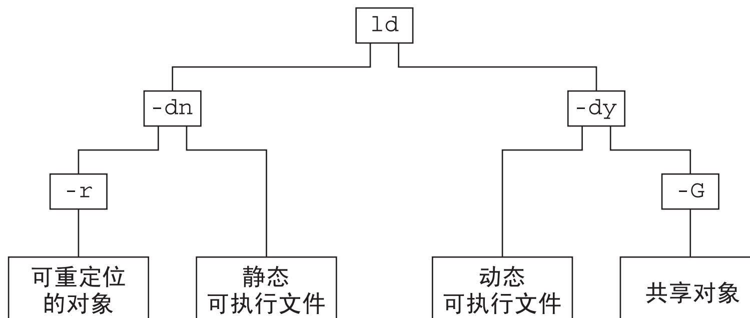


图 1–1 静态或动态链接编辑

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 位系统归档文件库。因为静态可执行文件是基于系统归档文件库生成的，所以这种可执行文件包含关于系统实现的详细信息。这种自包含特性有许多缺点：

- 静态可执行文件无法利用以共享库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。
- 可能会影响这种可执行文件是否能够运行在未来的发行版上。
- 系统实现详细信息的重复会对系统性能造成负面影响。

从 Solaris 10 发行版开始，不再提供 32 位系统归档文件库。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文件库的处理方式保持不变。

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库，这些功能可以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**共享库**。

动态可执行文件是在运行时链接程序控制下执行的应用程序。这些应用程序通常具有共享库（由运行时链接程序定位并绑定来创建可运行进程）形式的依赖性。动态可执行文件是链接编辑器生成的缺省输出文件。

共享库向动态链接系统提供主要组成单元。共享库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库。通常，必须将一个或多个共享库绑定到动态可执行文件以生成可运行进程。因为共享库可被许多应用程序使用，所以其构造的各个方面直接影响共享性、版本控制以及性能。

共享库是由链接编辑器处理还是由运行时链接程序处理，可以使用共享库的**环境**来区分：

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

运行时环境
: 共享库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进程中那些生成动态可执行文件和共享库的部分。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库来使用此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，动态链接还增加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 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。Solaris 链接程序依靠这些接口来装配要执行的应用程序。虽然 Solaris 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称为 *Solaris ABI*。

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* **兼容性定义** (SPARC Compliance Definition, SCD)。

32 位环境和 64 位环境

链接编辑器有 32 位应用程序和 64 位应用程序两种。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。但是，在一个链接编辑过程中不能同时处理 32 位目标文件和 64 位目标文件。在运行 64 位环境的系统上，链接编辑器的两个版本都可以运行。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。虽然 32 位链接编辑器可以生成 64 位目标文件，但是生成目标文件的大小（不包括 `.bss`）限制为 2 GB。

不需要命令行选项来区分 32 位链接编辑或 64 位链接编辑。链接编辑器使用命令行上第一个可重定位目标文件的 ELF 类来管理操作的模式。专用链接编辑（如，只来自 `mapfile` 或归档文件库的链接）不受命令行目标文件的影响。这些链接编辑缺省为 32 位模式。在这些情况下，可以使用链接编辑器的 `-64` 选项强制执行 64 位链接编辑。

链接编辑器对 32 位目标文件和 64 位目标文件的操作相同。本文档通常使用 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。

有关 64 位应用程序的更多信息，请参阅《Solaris（64 位）开发者指南》。

环境变量

链接编辑器支持许多以字符 `LD_` 开头的环境变量，如 `LD_LIBRARY_PATH`。每个环境变量都可以其通用形式存在，也可以使用 `_32` 或 `_64` 后缀指定，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。此后缀还覆盖任何可能有效的通用无后缀环境变量版本。

注 – 在 Solaris 10 发行版之前，链接编辑器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 Solaris 10 发行版开始，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都在 ld(1) 和 ld.so.1(1) 中定义。

支持工具

Solaris 操作环境还提供了许多支持工具和库。可以使用这些工具分析和检查这些目标文件和链接过程。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，许多过程都含有这些工具的示例。

第 2 章

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。

所有文件都使用**可执行链接格式** (executable and linking format, ELF) 表示。有关 ELF 格式的完整说明，请参见第 7 章。本章介绍两种 ELF 结构：**节**和**段**。

节是 ELF 文件中可以处理的最小不可分割单位。段是节的集合，节表示可由 `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位。

虽然存在许多类型的 ELF 节，但就链接编辑阶段而言可将所有节都归为两种类别：

- 包含**程序数据**的节，其解释仅对应用程序有意义，如程序指令 `.text` 以及关联的数据 `.data` 和 `.bss`。
- 包含**链接编辑信息**（如 `.symtab` 和 `.strtab` 中的符号表信息以及诸如 `.rela.text` 的重定位信息）的节。

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讨论的主题：

- 对提供的所有选项进行验证和一致性检查。
- 串联多个输入可重定位目标文件中具有相同特征的节，以便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库中的符号表信息，以便验证并把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，并通过更新其他输入节将此信息应用于输出文件。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，如共享库依赖项和符号绑定。

把相似的**节**串联起来以及关联**节**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。 请参见第 9 章。

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）。

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使用的目标文件模块或库不会作出任何假设。例如，当您发布以下命令时：

```
$ ld test.o
```

链接编辑器仅使用输入文件 `test.o` 创建一个名为 `a.out` 的动态可执行文件。要使 `a.out` 成为有用的可执行文件，应该包括用于启动和退出处理的代码。此代码可以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库）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) 和 `CC`(1) 等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其他文件，例如：

```
$ cc -# -o prog main.o

/usr/ccs/bin/ld -dy /opt/COMPILER/crti.o /opt/COMPILER/crt1.o \
```

```
/usr/ccs/lib/values-Xt.o -o prog main.o \

-YP,/opt/COMPILER/lib:/usr/ccs/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。

指定链接编辑器选项

大多数选项可以通过编译器驱动程序的命令行传递到链接编辑器。通常，编译器选项和链接编辑器选项不会产生冲突。如果产生冲突，编译器驱动程序通常提供一种命令行语法，您可以使用该语法将特定选项传递到链接编辑器。也可以通过设置 LD_OPTIONS 环境变量为链接编辑器提供选项。

```
$ LD_OPTIONS="-R /home/me/libs -L /home/me/libs" cc -o prog main.c -lfoo
```

链接编辑器解释 -R 和 -L 选项。这些选项位于从编译器驱动程序接收的任何命令行选项的前面。

链接编辑器解析整个选项列表，以查找任何无效选项或具有无效关联参数的任何选项。如果发现其中任何一种情况，则生成一条适当的错误消息。如果该错误被认为是致命错误，则链接编辑将终止。在以下示例中，链接编辑器通过检查捕获到非法选项 -X 和 -z 选项的非法参数。

```
$ ld -X -z sillydefs main.o

ld: illegal option -- X

ld: fatal: option -z has illegal argument ‘sillydefs’
```

如果指定了某个需要关联参数的选项两次，则链接编辑器将产生适当的警告并继续进行链接编辑。

```
$ ld -e foo ...... -e bar main.o

ld: warning: option -e appears more than once, first setting taken
```

链接编辑器还会检查选项列表以查看是否存在任何致命的不一致性错误。

```
$ ld -dy -a main.o

ld: fatal: option -dy and -a are incompatible
```

处理完所有选项之后，如果未检测到任何致命错误状态，则链接编辑器将继续处理输入文件。

请参见附录 A 以了解最常用的链接编辑器选项，并参见 ld(1) 以了解所有链接编辑器选项的完整说明。

输入文件处理

链接编辑器按输入文件在命令行中的出现顺序读取这些文件。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库。

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库。 请参见第 32 页中的 “与其他库链接” 。在链接编辑期间，归档库和共享库的解释完全不同。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关系。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所需的目标文件以完成绑定过程。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，它导致提取定义相同 DATA 符号的可重定位目标文件。
- 链接编辑器 -z allextract 有效。此选项暂挂选择性归档提取，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，请参见第 40 页中的 “简单解析” 。

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，链接编辑器会检查整个归档多遍。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。请参见第 33 页中的 “命令行中归档的位置”。

注 – 虽然链接编辑器检查整个归档多遍以解析符号，但此机制的开销很大。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，应使用诸如 `lorder`(1) 和 `tsort`(1) 的工具对归档中的可重定位目标文件排序。该排序操作可减少链接编辑器必须检查归档的遍数。

共享库处理

共享库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库时，共享库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库中定义的所有符号项。在执行进程期间实际上会复制共享库。

链接编辑器不使用共享库的程序数据节和大多数链接编辑信息节。绑定共享库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，会记录出现的共享库。信息存储在输出文件映像中，以便表明此目标文件是在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库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库提供的符号，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。或者，链接编辑器的 `-z ignore` 选项可以抑制记录未使用的共享库的依赖项。

如果某个共享库依赖于其他共享库，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，不会将共享库名称作为依赖项进行记录。

虽然命令行中共享库的位置没有归档处理那么重要，但该位置具有全局效果。可重定位库与共享库之间以及多个共享库之间允许存在多个名称相同的符号。请参见第 39 页中的 “符号解析”。

链接编辑器处理共享库的顺序由存储在输出文件映像中的依赖性信息维护。运行时链接程序读取此信息，并按相同的顺序装入指定的共享库。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确保对链接编辑器指定适当的库，但您经常必须提供自己的库。通过显式指定链接编辑器需要的输入文件可以指定共享库和归档。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，通常指定共享库具有前缀 lib 和后缀 .so。指定归档具有前缀 lib 和后缀 .a。例如，libc.so 是可用于编译环境的标准 C 库的共享版本。libc.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库都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。请参见第 119 页中的 “命名约定” 和第 175 页中的 “协调版本化文件名” 。

当链接编辑处于动态模式时，可以选择同时链接共享库和归档。当链接编辑处于静态模式时，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库。如果未找到任何匹配项，则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。

同时链接共享库和归档

动态模式下的库搜索机制搜索给定目录以查找共享库，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，可以分别在共享库或归档之间切换库搜索。例如，要将应用程序与归档 libfoo.a 和共享库 libbar.so 链接，可发布以下命令：

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

-B static 和 -B dynamic 关键字并不完全对称。指定 -B static 时，链接编辑器要等到下一次出现 -B dynamic 时才接受共享库作为输入。但是，指定 -B dynamic 时，链接编辑器首先在任何给定的目录中查找共享库，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，然后搜索 libbar.so，如果此搜索失败，则搜索 libbar.a。最后，链接编辑器搜索 libc.so，如果此搜索失败，则搜索 libc.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的未定义或暂定外部引用。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搜索 libfoo.a，仅仅是为了解析从 file1.c 中获取的符号引用。不能使用 libfoo.a 归档解析 file2.c 或 file3.c 中的符号引用。

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

注 – 应该在命令行末尾指定任何归档，除非多重定义冲突要求采取其他方式。

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。例如：

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

确定和维护重复指定的归档是一个繁琐的任务。使用 -z rescan 选项可以简化此过程。处理完所有输入文件后，此选项将导致重新处理整个归档列表。此处理过程尝试查到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任何新成员为止。因此，上一个示例可以简化为：

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

链接编辑器搜索的目录

上面所有示例都假定链接编辑器了解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三个标准目录中查找库：先搜索 /usr/ccs/lib，然后搜索 /lib，最后搜索 /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，然后搜索 /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来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，将改变搜索路径。例如，以下命令搜索 path1，然后搜索 /usr/ccs/lib 和 /lib，最后搜索 /usr/lib 来查找 libfoo。此命令搜索 path1，然后搜索 path2，接着搜索 /usr/ccs/lib、/lib 和 /usr/lib 来查找 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 – 如果要链接编辑器在当前目录中搜索库，则必须指定 -L。可以使用句点 (.) 来表示当前目录。

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，以下命令仅在目录 /opt/COMPILER/lib 和 /home/me/lib 中搜索 libfoo。

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

可以使用 -L 选项补充使用 -Y 选项指定的目录。

使用环境变量

还可以使用环境变量 LD_LIBRARY_PATH（采用冒号分隔的目录列表形式）将要搜索的目录添加到链接编辑器的库搜索路径中。LD_LIBRARY_PATH 最常见的形式是以分号分隔的两个目录列表。系统按照命令行中提供的列表依次进行搜索。

以下示例说明在设置 LD_LIBRARY_PATH 且调用链接编辑器时使用多个 -L 的情况下所得结果：

```
$ 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:/usr/ccs/lib:/lib:/usr/lib。

如果在 LD_LIBRARY_PATH 定义中未指定分号，则将在解释所有 -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:/usr/ccs/lib:/lib:/usr/lib。

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

注 – 还可以使用此环境变量扩充运行时链接程序的搜索路径。 请参见第 76 页中的 “运行时链接程序搜索的目录” 。为了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`。必须显式地将要搜索的所有其他目录添加到运行时链接程序的搜索路径中。

当动态可执行文件或共享库与其他共享库链接时，这些共享库将作为依赖性进行记录。运行时链接程序执行进程期间必须找到这些依赖性。链接动态库时，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来查找该目标文件的依赖性。

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。该选项的用法表示使用目标文件的运行路径可以查找该目标文件的所有依赖性。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，该搜索路径中的最后一个元素始终是缺省位置。

注 – 可以使用运行时配置文件管理缺省搜索路径。 请参见第 79 页中的 “配置缺省搜索路径” 。但是，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它的依赖性。

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库中。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）来获取任何共享库的依赖性。在本示例中，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，上一个示例还可以按如下所示表示。

```
$ 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。请参见第 373 页中的 “查找关联的依赖项”。

注 – 以前指定 -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。

初始化和终止节

动态库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库时，都会执行一次动态库的初始化代码。每次从进程中卸载动态库或进程终止时，都会执行一次动态库的终止代码。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。

.preinit_array、.init_array 和 .fini_array 节分别提供运行时预初始化数组、初始化数组和终止函数数组。创建动态库时，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]）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。

.init 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提供 .init 和 .fini 节以及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。这些函数分别用保留符号名称 _init 和 _fini 标识。创建动态库时，链接编辑器相应地使用 .dynamic 标记（DT_INIT 和 DT_FINI）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，请参见第 91 页中的 “初始化和终止例程”。

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 .initarray 元素中，并将 bar() 的地址放置在 .finiarray 元素中。

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

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

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

main()
{
        (void) printf("main()\n");
        return (0);
}

$ 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元语来简化其声明。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库中时，这些代码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。在运行时装入依赖性时，针对共享库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。限制初始化代码和终止代码的内容可简化此分析过程。简化的初始化代码和终止代码提供可预测的运行时行为。有关更多详细信息，请参见第 92 页中的 “初始化和终止顺序”。

如果初始化代码中包含可以使用 dldump(3C) 转储其内存的动态库，则数据初始化应该是一个独立的过程。

符号处理

在输入文件处理期间，输入可重定位目标文件中的所有**局部**符号都将传递到输出文件映像。所有全局符号都在链接编辑器内部累积。在此内部符号表中搜索可重定位目标文件提供的每个**全局**符号。如果遇到了名称与上一个输入文件中某个符号的名称相同的符号，则调用符号解析过程。符号解析过程决定这两项中哪一项会被保留。

完成输入文件处理并且没有发生致命符号解析错误时，链接编辑器将确定是否存在任何未解析的符号引用。未解析的符号引用可能导致链接编辑终止。

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。

以下各小节详细说明了符号解析和未定义符号处理过程。

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，而其他重定位可能导致致命错误状态。

这两种符号的解析取决于符号的属性、提供符号的文件的类型以及要生成的文件的类型。有关符号属性的完整说明，请参见第 260 页中的 “符号表节” 。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**－文件中已创建但尚未确定大小或尚未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 符号或 FORTRAN COMMON 块。
- **已定义**－文件中已创建并且已指定存储地址和空间的符号。

符号解析最简单的形式涉及使用优先级关系。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;



d_foo()

{

        return (u_foo(u_bar, t_bar, d_bar));

}

$ cc -o main.o -c main.c
```

```
$ nm -x main.o

[Index]   Value      Size      Type  Bind  Other Shndx   Name
...............
[8]     |0x00000000|0x00000000|NOTY |GLOB |0x0  |UNDEF  |u_foo
[9]     |0x00000000|0x00000040|FUNC |GLOB |0x0  |2      |d_foo
[10]    |0x00000004|0x00000004|OBJT |GLOB |0x0  |COMMON |t_bar
[11]    |0x00000000|0x00000000|NOTY |GLOB |0x0  |UNDEF  |u_bar
[12]    |0x00000000|0x00000004|OBJT |GLOB |0x0  |3      |d_bar
```

简单解析

到目前为止，简单符号解析是最常见的一种解析形式。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。

要解析的符号可以具有全局绑定或弱绑定。弱绑定的优先级低于全局绑定，因此，将根据略有改变的基本规则解析具有不同绑定的符号。

通常可以通过编译器单独定义弱符号或将它们定义为全局符号的别名。一种机制使用 #pragma 定义：

```
$ cat main.c
#pragma weak    bar
#pragma weak    foo = _foo


int             bar = 1;


_foo()
{
```

```
        return (bar);
}
$ cc -o main.o -c main.c
$ nm -x main.o
[Index]   Value      Size      Type  Bind  Other Shndx   Name
...............
[7]     |0x00000000|0x00000004|OBJT |WEAK |0x0  |3      |bar
[8]     |0x00000000|0x00000028|FUNC |WEAK |0x0  |2      |foo
[9]     |0x00000000|0x00000028|FUNC |GLOB |0x0  |2      |_foo
```

请注意，对弱别名 foo 指定了与全局符号 _foo 相同的属性。此关系由链接编辑器维护，并将导致在输出映像中对符号指定相同的值。在符号解析过程中，定义的弱符号会被名称相同的任何全局定义覆盖，且没有任何提示。

在可重定位目标文件与共享库之间或者多个共享库之间进行的另一种形式的简单符号解析是插入。在这些情况下，如果多重定义了某个符号，则链接编辑器会采用可重定位目标文件或多个共享库之间的第一个定义，且不作任何提示。可重定位目标文件的定义或第一个共享库的定义**插入**进来取代所有其他定义。这种插入可用于覆盖其他共享库提供的功能。

弱符号和符号插入的组合可以提供有用的编程方法。例如，标准 C 库提供了多个允许您重新定义的服务。但是，ANSI C 定义了一组必须出现在系统中的标准服务。在严格遵循规则的程序中不能替换这些服务。

例如，函数 fread(3C) 是一个 ANSI C 库函数，而系统函数 read(2) 不是。遵循 ANSI 的C 程序必须可以重新定义 read(2)，并且仍以可预测的方法使用 fread(3C)。

此处的问题是，在标准 C 库中 read(2) 是 fread(3C) 实现的基础。因此，重新定义 read(2) 的程序可能会混淆 fread(3C) 实现。为了避免出现这种情况，ANSI C 声明实现只能使用为该实现保留的名称。使用以下 #pragma 指令可定义这种保留名称。使用此名称可生成函数 read(2) 的别名。

```
#pragma weak read = _read
```

因此，您可以非常自由地定义自己的 read() 函数，而不会破坏 fread(3C) 实现，而 fread(3C) 是使用 _read() 函数实现的。

链接编辑器在链接共享库或标准 C 库的归档版本时，可以轻松重新定义 read()。在前一种情况下，可以执行插入操作。在后一种情况下，由于 C 库中 read(2) 的定义较弱，所以允许默认覆盖该定义。

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

复杂解析

如果发现两个符号的名称相同，但属性不同，则可以进行复杂解析。在这些情况下，链接编辑器将选择最适合的符号并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 };



$ cc -dn -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，则会生成一个类似的诊断。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

bar()

{

        return (0);

}

$ cc -o libfoo.so -G -K pic foo.c
```

```
$ cat main.c
int     bar = 1;

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 表示的类别。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库之间进行解析时将采用可重定位目标文件定义。或者，在两个共享库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。此消息指出符号名称和提供这些符号的文件的名称。不生成输出文件。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的处理。在此方式下，可以标识所有致命解析错误。

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
bar()
{
        return (0);
}

$ cc -dn -r -o temp.o foo.c bar.c
ld: fatal: symbol ‘bar’ is multiply-defined:
        (file foo.o and file bar.o);
ld: fatal: File processing errors. No output written to int.o
```

对于符号 `bar` 来说，`foo.c` 和 `bar.c` 具有相冲突的定义。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

在读取所有输入文件并完成所有符号解析后，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。这些未定义符号在链接编辑过程中的效果根据要生成的输出文件类型而不同，也可能根据符号类型而不同。

生成可执行的输出文件

生成可执行的输出文件时，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生成相应的错误消息。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，则表示此符号未定义：

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

main()
```

```
{
        return (foo());
}
$ cc -o prog main.c
Undefined           first referenced
 symbol                   in file
foo                       main.o
ld: fatal: Symbol referencing errors. No output written to prog
```

同样，如果使用共享库创建动态可执行文件，并且该库中的符号引用始终未解析，则会产生未定义符号错误。

```
$ cat foo.c
extern int bar;
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. No output written to prog
```

要允许未定义符号（与上一个示例一样），可使用链接编辑器的 `-z nodefs` 选项抑制出现缺省错误状态。

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，则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库中的符号定义时，符号也可以保持未定义。例如，继续使用上一个示例中使用的文件 `main.c` 和 `foo.c`：

```
$ 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. No output written to prog
```

`prog` 是使用对 `libbar.so` 的**显式**引用生成的。`libbar.so` 依赖于 `libfoo.so`。因此，将从 `prog` 建立对 `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` 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库的依赖项。因此，如果开发一种不再依赖于 `libfoo.so` 的新版本 `libbar.so`，则 `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。

生成共享库输出文件

链接编辑器生成共享库输出文件时，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库从将其定义为依赖性的动态可执行文件导入符号。

可以使用链接编辑器的 -z defs 选项强制在存在任何未定义符号的情况下生成致命错误。建议在创建任何共享库时使用此选项。引用应用程序中的符号的共享库可以使用 -z defs 选项，并可以使用 extern mapfile 指令定义符号。请参见第 49 页中的 “定义其他符号”。

自包含的共享库（通过指定的依赖性来满足对外部符号的所有引用）可提供最大的灵活性。此共享库可由许多用户使用，并且这些用户无需确定和建立依赖性来满足共享库的要求。

弱符号

无论要生成哪种类型的输出文件，未解析的弱符号引用不会产生致命错误状态。

如果要生成静态可执行文件，则可将此符号转换为绝对符号，并指定值零。

如果要生成动态可执行文件或共享库，则可将此符号保留为未定义弱引用，并指定值零。在进程执行期间，运行时链接程序将搜索此符号。如果运行时链接程序未找到匹配项，则将此引用绑定到地址零，而不是生成致命重定位错误。

以前，这些未定义弱引用符号被用作一种机制，用于测试功能是否存在。例如，在共享库 libfoo.so.1 中可能使用了以下 C 代码段：

```
#pragma weak    foo



extern  void    foo(char *);



void bar(char * path)

{

        void (* fptr)(char *);



        if ((fptr = foo) != 0)

                (* fptr)(path);

}
```

生成引用 libfoo.so.1 的应用程序时，无论是否找到符号 foo 的定义，链接编辑都将成功完成。如果在执行此应用程序时函数地址测试为非零，则将调用此函数。但是，如果未找到符号定义，则函数地址测试将为零，因此不调用此函数。

编译系统将此地址比较方法视为未定义语义，这将导致在优化时删除测试语句。此外，运行时符号绑定机制会对使用此方法设定其他限制。这些限制防止所有动态库使用一致的模型。

注 – 建议不要按照此方式使用未定义弱引用。相反，应将 dlsym(3C) 与 RTLD_DEFAULT 或 RTLD_PROBE 句柄配合使用，以测试符号是否存在。 请参见第 106 页中的 "测试功能"。

输出文件中暂定符号的顺序

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。处理暂定符号及其关联的存储空间时，情况却有所不同。完成这些符号的解析后才会完全定义这些符号。如果解析可重定位目标文件中的**已定义**符号，则此符号出现在定义后面。

如果需要控制一组符号的顺序，则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在输出文件中重新排序数据项：

```
$ cat foo.c
char A_array[0x10];
char B_array[0x20];
char C_array[0x30];


$ cc -o prog main.c foo.c
$ nm -vx prog | grep array
[32]    |0x00020754|0x00000010|OBJT |GLOB |0x0  |15  |A_array
[34]    |0x00020764|0x00000030|OBJT |GLOB |0x0  |15  |C_array
[42]    |0x00020794|0x00000020|OBJT |GLOB |0x0  |15  |B_array
```

通过将这些符号定义为已初始化的数据项，这些符号在输入文件中的相对顺序将被传递到输出文件：

```
$ cat foo.c

char A_array[0x10] = { 0 };

char B_array[0x20] = { 0 };

char C_array[0x30] = { 0 };



$ cc -o prog main.c foo.c

$ nm -vx prog | grep array

[32]    |0x000206bc|0x00000010|OBJT |GLOB |0x0  |12  |A_array

[42]    |0x000206cc|0x00000020|OBJT |GLOB |0x0  |12  |B_array

[34]    |0x000206ec|0x00000030|OBJT |GLOB |0x0  |12  |C_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符号引用或定义。使用链接编辑器的 `-u` 选项可以生成符号引用的最简单形式。链接编辑器的 `-M` 选项和关联的 `mapfile` 可以提供更大的灵活性。使用此 mapfile，可以定义符号引用和各种符号定义。

`-u` 选项提供一种在链接编辑命令行中生成符号引用的机制。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，请参见第 30 页中的 “归档处理”一节。

例如，您可能要从可重定位目标文件 `main.o` 生成动态可执行文件，此目标文件引用符号 `foo` 和 `bar`。您要从 `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` 还包含定义符号 `bar` 的可重定位目标文件。对于 `lib2.a` 中提供的可重定位目标文件，此可重定位目标文件的功能可能不同。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` 中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` 既没有直接引用也没有间接引用符号 `bar`。如果 `lib1.a` 引用 `bar`，则在处理 `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论述，请参见第 30 页中的 "归档处理"。

可以使用链接编辑器的 `-M` 选项和关联的 `mapfile` 提供一组更丰富的符号定义。符号定义 `mapfile` 的项使用以下语法。

```
[ name ] {
        scope:
                symbol [ = [ type ] [ value ] [ size ] [ attribute ] ];
} [ dependency ];
```

name
: 此组符号定义的标签（如果存在）标识映像中的**版本定义**。 请参见第 5 章。

scope
: 指示要生成的输出文件中符号绑定的可见性。在链接编辑过程中，使用 `mapfile` 定义的所有符号都将视为全局符号。将针对任何其他具有相同名称的符号（从所有输入文件中获取）解析这些符号。以下定义和别名定义在要创建的目标文件中符号的可见性：

  `default` / `global`
  : 此范围的符号对所有外部目标文件都可见。在运行时绑定从目标文件内部对这种符号的引用，从而允许进行插入。

  `protected` / `symbolic`
  : 此范围的符号对所有外部目标文件都可见。在链接编辑时绑定从目标文件内部对这种符号的引用，从而防止在运行时插入。此范围定义与具有 `STV_PROTECTED` 可见性的符号产生相同的效果。 请参见表 7–20。

  `hidden` / `local`
  : 此范围的符号缩减为具有本地绑定的符号。此范围的符号对其他外部目标文件不可见。此范围定义与具有 `STV_HIDDEN` 可见性的符号产生相同的效果。 请参见表 7–20。

  `eliminate`
  : 此范围的符号是 `hidden`。删除这些符号的符号表项。

symbol
: 所需符号的名称。如果该名称未后跟符号属性 *type*、*value*、*size* 或 `extern` 之一，则将创建符号引用。此引用与本节前面所述的使用 `-u` 选项生成的引用完全相同。如果符号名称后跟任何符号属性，则将使用关联的属性生成符号定义。

当符号属于 `local` 范围时，可以将此符号名称定义为特殊**自动缩减**指令 "*"。此指令将任何 `mapfile` 中未显式定义为 `global` 的所有全局符号降级为要生成的动态库中的本地绑定。

type
指示符号类型属性。此属性可以为 `data`、`function` 或 `COMMON`。前两种类型的属性会产生绝对符号定义。 请参见第 260 页中的 "符号表节" 。后一种类型的属性会产生暂定符号定义。

value
指示值属性。此属性的格式为 `V`*number*。

size
指示大小属性。此属性的格式为 `S`*number*。

attribute
此关键字提供符号的其他属性：

`EXTERN`
指示在外部对要创建的目标文件定义符号。 此属性可以与 `DIRECT` 或 `NODIRECT` 属性配合使用，以建立单独的直接或非直接引用。还可以使用此选项抑制将使用 `-z defs` 选项标记的未定义符号。

`DIRECT`
指示应直接绑定到此符号。此属性可以与 `EXTERN` 属性配合使用以控制绑定到外部符号。 请参见第 82 页中的 "直接绑定" 。

`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。此属性可以与 `EXTERN` 属性配合使用以控制绑定到外部符号。 请参见第 82 页中的 "直接绑定" 。

`FILTER` *name*
指示此符号在共享库 *name* 中是一个过滤器。 请参见第 126 页中的 "生成标准过滤器" 。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。

`AUXILIARY` *name*
指示此符号在共享库 *name* 中是一个辅助过滤器。 请参见第 129 页中的 "生成辅助过滤器" 。

dependency
表示此定义继承的**版本定义**。 请参见第 5 章。

如果指定了版本定义或自动缩减指令，则将在创建的映像中记录版本控制信息。如果此映像是可执行文件或共享库，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库时，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 5 章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。

以下各小节提供了多个使用 mapfile 语法的示例。

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范围缩减到**局部**。

```
$ cat foo.c

foo()

{

        (void) printf("foo: called from lib.a\n");

}

$ cat bar.c

bar()

{

        (void) printf("bar: called from lib.a\n");

}

$ cat main.c

extern  void    foo(), bar();



main()

{

        foo();
```

```
        bar();
}
$ ar -rc lib.a foo.o bar.o main.o
$ cat mapfile
{
        local:
                foo;
                bar;
        global:
                main;
};
$ cc -o prog -M mapfile lib.a
$ prog
foo: called from lib.a
bar: called from lib.a
$ nm -x prog | egrep "main$|foo$|bar$"
[28]    |0x00010604|0x00000024|FUNC |LOCL |0x0  |7      |foo
[30]    |0x00010628|0x00000024|FUNC |LOCL |0x0  |7      |bar
[49]    |0x0001064c|0x00000024|FUNC |GLOB |0x0  |7      |main
```

在第 57 页中的 "缩减符号范围"一节中更详细地说明了将符号范围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
extern  int     foo();
extern  int     bar;

main()
{
        (void) printf("&foo = %x\n", &foo);
        (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
        global:
                foo = FUNCTION V0x400;
                bar = DATA V0x800;
};
$ cc -o prog -M mapfile main.c
$ prog
&foo = 400 &bar = 800
$ nm -x prog | egrep "foo$|bar$"
[37]    |0x00000800|0x00000000|OBJT |GLOB |0x0  |ABS    |bar
[42]    |0x00000400|0x00000000|FUNC |GLOB |0x0  |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。如果在共享库中定义符号，则应避免绝对定义。 请参见第 56 页中的 “扩充符号定义” 。

定义暂定符号

还可以使用 mapfile 定义 COMMON 或暂定符号。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。

暂定符号与其他类型的符号的一个不同特征在于，暂定符号的 *value* 属性指示其对齐要求。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
extern  int     foo;
int             bar[0x10];


main()
{
        (void) printf("&foo = %x\n", &foo);
        (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
        global:
                foo = COMMON V0x4 S0x200;
                bar = COMMON V0x100 S0x40;
};
$ cc -o prog -M mapfile main.c
ld: warning: symbol ‘bar’ has differing alignments:
        (file mapfile value=0x100; file main.o value=0x4);
```

```
        largest value applied
$ prog
&foo = 20940
&bar = 20900
$ nm -x prog | egrep "foo$|bar$"
[37]    |0x00020900|0x00000040|OBJT |GLOB |0x0  |16     |bar
[42]    |0x00020940|0x00000200|OBJT |GLOB |0x0  |16     |foo
```

注 – 使用链接编辑器的 `-t` 选项可以不诊断此符号解析。

扩充符号定义

应避免在共享库中创建绝对数据符号。从动态可执行文件对共享库中数据项的外部引用通常需要创建复制重定位。请参见第 144 页中的 “复制重定位”。要提供此重定位，应该在目标文件中定义数据项，以便将符号定义与数据存储关联。

可以过滤数据符号。请参见第 125 页中的 “作为过滤器的共享库”。要提供此过滤，可以使用 `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。虽然可以在 `mapfile` 中显式创建函数定义，但数据定义将扩充输入可重定位目标文件提供的定义。

```
$ cat bar.c
int bar = 0;
$ cat mapfile
{
        global:
                foo = FUNCTION FILTER filtee.so.1;
                bar = FILTER filtee.so.1;
        local:
                *;
};
```

```
$ cc -o filter.so.1 -G -Kpic -h filter.so.1 -M mapfile -R. bar.c
$ nm -x filter.so.1 | egrep "foo|bar"
[39]    |0x000102b0|0x00000004|OBJT |GLOB |0    |12     |bar
[45]    |0x00000000|0x00000000|FUNC |GLOB |0    |ABS    |foo
$ elfdump -y filter.so.1 | egrep "foo|bar"
       [1]  F        [0] filtee.so.1        bar
       [7]  F        [0] filtee.so.1        foo
```

运行时，从外部目标文件对任一符号的引用将解析为 filtee 中的定义。

缩减符号范围

可以使用在 mapfile 中定义为具有局部范围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，从而限制其他目标文件可以使用的功能。

例如，假设要从文件 foo.c 和 bar.c 生成一个简单的共享库。文件 foo.c 包含全局符号 foo，此符号提供其他目标文件可以使用的服务。文件 bar.c 包含符号 bar 和 str，这两个符号提供共享库的基础实现。使用这些文件创建的共享库通常导致创建三个具有全局范围的符号。

```
$ 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 lib.so.1 -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[29]    |0x000104d0|0x00000004|OBJT |GLOB |0x0  |12     |str
[32]    |0x00000418|0x00000028|FUNC |GLOB |0x0  |6      |bar
[33]    |0x000003f0|0x00000028|FUNC |GLOB |0x0  |6      |foo
```

现在，可以在另一个应用程序的链接编辑过程中使用 lib.so.1 提供的功能。将对符号 foo 的引用绑定到共享库提供的实现。

由于符号 bar 和 str 具有全局绑定，因此还可以直接引用这些符号。此可见性会产生严重后果，因为您可能会在以后更改作为函数 foo 的基础的实现。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。

全局绑定符号 bar 和 str 的另一个后果是，可以在这些符号中插入相同名称的符号。第 40 页中的 “简单解析”一节中说明了在共享库中插入符号。此插入可能是有意的，可以用来禁用共享库提供的预期功能。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库的结果。

在开发共享库时，可以通过将符号 bar 和 str 的范围缩减为本地绑定来防止出现这种情况。在以下示例中，不能再将符号 bar 和 str 用作共享库接口。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库的接口。可以在隐藏基础实现详细信息的同时管理此接口。

```
$ cat mapfile
{
        local:
                bar;
                str;
};
```

```
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[27]    |0x000003dc|0x00000028|FUNC |LOCL |0x0  |6      |bar
[28]    |0x00010494|0x00000004|OBJT |LOCL |0x0  |12     |str
[33]    |0x000003b4|0x00000028|FUNC |GLOB |0x0  |6      |foo
```

缩减符号范围还具有其他性能方面的优点。现在，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，请参见第 143 页中的 “何时执行重定位” 。

随着链接编辑期间处理的符号数的增加，在 mapfile 中定义局部范围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定义共享库的接口。全局符号定义允许链接编辑器将所有其他符号缩减为本地绑定。可使用特殊的**自动缩减**指令 "*" 实现此机制。例如，可以重新编写上面的 mapfile 定义，以便将 foo 定义为生成的输出文件中需要的唯一全局符号：

```
$ cat mapfile
lib.so.1.1
{
        global:
                foo;
        local:
                *;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[30]    |0x00000370|0x00000028|FUNC |LOCL |0x0  |6      |bar
[31]    |0x00010428|0x00000004|OBJT |LOCL |0x0  |12     |str
[35]    |0x00000348|0x00000028|FUNC |GLOB |0x0  |6      |foo
```

此示例还将版本名称 lib.so.1.1 定义为 mapfile 指令的一部分。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版本控制机制的基础。 请参见第 5 章。

注 – 如果未提供版本名称，则将使用输出文件名标记版本定义。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控制信息。

每次指定版本名称时，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，则链接编辑器将生成致命错误状态：

```
$ cat mapfile
lib.so.1.1 {
        global:
                foo;
};
$ cc -o lib.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. No output written to lib.so.1
```

可以使用 -B local 选项在命令行中声明**自动缩减**指令 "*"。使用以下指令可以成功编译上一个示例：

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

生成可执行文件或共享库时，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，会创建一个中间可重定位目标文件，但不缩减任何符号。

```
$ cat mapfile
lib.so.1.1 {
```

```
        global:
                foo;
        local:
                *;
};
$ ld -o lib.o -M mapfile -r foo.o bar.o
$ nm -x lib.o | egrep "foo$|bar$|str$"
[17]    |0x00000000|0x00000004|OBJT |GLOB |0x0  |3      |str
[19]    |0x00000028|0x00000028|FUNC |GLOB |0x0  |1      |bar
[20]    |0x00000000|0x00000028|FUNC |GLOB |0x0  |1      |foo
```

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库时，将会缩减符号。也就是说，链接编辑器将按照与在 `mapfile` 中处理版本控制数据相同的方式，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库：

```
$ ld -o lib.so.1 -G lib.o
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[22]    |0x000104a4|0x00000004|OBJT |LOCL |0x0  |14     |str
[24]    |0x000003dc|0x00000028|FUNC |LOCL |0x0  |8      |bar
[36]    |0x000003b4|0x00000028|FUNC |GLOB |0x0  |8      |foo
```

创建可执行文件或共享库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 lib.o -M mapfile -B reduce -r foo.o bar.o
$ nm -x lib.o | egrep "foo$|bar$|str$"
[15]    |0x00000000|0x00000004|OBJT |LOCL |0x0  |3      |str
[16]    |0x00000028|0x00000028|FUNC |LOCL |0x0  |1      |bar
[20]    |0x00000000|0x00000028|FUNC |GLOB |0x0  |1      |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。可以使用链接编辑器的 -s 选项或 strip(1) 从目标文件中删除整个表。有时，您可能需要保留 .symtab 符号表，但删除选择的局部符号定义。

可以使用 mapfile 指令 eliminate 删除符号。与 local 指令一样，可以单独定义符号。或者，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

lib.so.1.1

{

        global:

                foo;

        local:

                str;

        eliminate:

                *;

};

$ cc -o lib.so.1 -M mapfile -G foo.c bar.c

$ nm -x lib.so.1 | egrep "foo$|bar$|str$"

[31]    |0x00010428|0x00000004|OBJT |LOCL |0x0  |12     |str

[35]    |0x00000348|0x00000028|FUNC |GLOB |0x0  |6      |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

共享库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。提供定义的共享库通常具有依赖性。

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性。

还可以创建目标文件以使用直接绑定。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 82 页中的"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。压缩的 .dynstr 表的文本段较小，因此可以减少运行时换页活动。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成本，应使用链接编辑器的 -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库时，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 mmap(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 -s 选项或在链接编辑后使用 strip(1) 甚至可以从映像中删除这些节。

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
: 文本段后面的第一个位置。

_edata
: 已初始化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 动态信息节的地址。

END
　与 _end 相同。此符号具有局部范围，它与 _START_ 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）的位置无关的引用。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。 请参见第 137 页中的 “与位置无关的代码” 。

_PROCEDURE_LINKAGE_TABLE_
　对链接编辑器提供了地址表（即 .plt 节）的与位置无关的引用。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使用 -K pic 选项编译的目标文件中。 请参见第 137 页中的 “与位置无关的代码” 。

START
　文本段中的第一个位置。此符号具有局部范围，它与 _END_ 一起提供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，则将使用该符号。否则，链接编辑器将查找保留符号名称 _start，然后查找 main。如果这些符号都不存在，则将使用文本段的第一个地址。

标识硬件和软件功能

通常在编译时记录可重定位目标文件的硬件和软件功能。链接编辑器合并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。 请参见第 240 页中的 “硬件和软件功能节” 。

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输入可重定位目标文件提供的功能。

以下各小节说明如何使用 mapfile 定义功能。

标识硬件功能

目标文件的硬件功能标识目标文件正常执行要满足的平台硬件要求。例如，要求可能是，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：

```
hwcap_1 = TOKEN | Vval [ OVERRIDE ];
```

使用一个或多个标记限定 hwcap_1 声明，这些标记是硬件功能的符号表示。此外，通过在值前面加上 V 作为前缀可以提供表示多个功能中的某个功能的数值。对于 SPARC 平台，硬件功能定义为 sys/auxv_SPARC.h 中的 AV_ 值。对于 x86 平台，硬件功能定义为 sys/auxv_386.h 中的 AV_ 值。

以下 x86 示例说明如何将 MMX 和 SSE 声明为目标文件 foo.so.1 需要的硬件功能。

```
$ egrep "MMX|SSE" /usr/include/sys/auxv_386.h
#define AV_386_MMX    0x0040
#define AV_386_SSE    0x0800
$ cat mapfile
hwcap_1 = SSE MMX;
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Hardware/Software Capabilities Section:  .SUNW_cap
     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。链接编辑器可以合并多个输入可重定位目标文件中的任何硬件功能值。生成的 CA_SUNW_HW_1 值是对关联的输入值进行按位或运算的结果。缺省情况下，这些值与 mapfile 指定的硬件功能合并。

可以使用 OVERRIDE 关键字在 mapfile 中显式控制输出文件的硬件功能要求。OVERRIDE 关键字和硬件功能值 0 可有效地从要生成的目标文件中删除任何硬件功能要求。

```
$ elfdump -H foo.o

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x840  [ SSE  MMX ]
$ cat mapfile
hwcap_1 = V0x0 OVERRIDE;
$ cc -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
```

```
$
```

运行时链接程序会针对进程可用的硬件功能验证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，则不会在运行时装入该目标文件。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 ]
        libc.so.1 =>    /lib/libc.so.1
```

利用不同硬件功能的动态库可以使用过滤器提供灵活的运行时环境。请参见第 367 页中的 “特定于硬件功能的共享库” 。

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。目前，可识别的唯一软件功能与目标文件使用的帧指针有关。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。

sys/elf.h 中定义的两个标志表示帧指针状态：

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

可以使用以下 mapfile 语法标识这些软件功能要求：

```
sfcap_1 = TOKEN | Vval [ OVERRIDE ];
```

可以使用 FPKNWN 和 FPUSED 标志限定 sfcap_1 声明。或者，可以使用表示这些状态的数值进行限定。

可重定位目标文件可以包含软件功能值。链接编辑器合并多个输入可重定位目标文件中的软件功能值。可按如下方法根据两个输入值计算 CA_SUNW_SF_1 的值。

表 2–1 CA_SUNW_SF_1 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>

表 2–1 CA_SUNW_SF_1 标志组合状态表　（续）

输入文件 1	输入文件 2		
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，或者此节不包含 CA_SUNW_SF_1 值，或者未设置 SF1_SUNW_FPKNW 和 SF1_SUNW_FPUSED 标志，则目标文件的软件功能未知。

缺省情况下，使用相同的状态模型处理 mapfile 指定的任何软件功能。

可以使用 OVERRIDE 关键字在 mapfile 中显式控制输出文件的软件功能要求。OVERRIDE 关键字和软件功能值 0 可有效地从要生成的目标文件中删除任何软件功能要求。

```
$ elfdump -H foo.o



Hardware/Software Capabilities Section:  .SUNW_cap

     index  tag               value

       [0]  CA_SUNW_SF_1     0x3  [ SF1_SUNW_FPKNWN  SF1_SUNW_FPUSED ]

$ cat mapfile

sfcap_1 = V0x0 OVERRIDE;

$ cc -o bar.o -r -Mmapfile foo.o

$ elfdump -H bar.o

$
```

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的映像。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的映像。

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。有两种状态需要进行详细地论述。第一种状态涉及位置相关代码产生的文本重定位。第 137 页中的“与位置无关的代码”中对此状态进行了更详细的说明。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说明。

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，则可能会出现错误状态。第 144 页中的“复制重定位”中对复制重定位进行了详细介绍。

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库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，则位移重定位将变为无效。

尝试捕获这些类型的错误时要解决的问题包括：

- 生成共享库时，如果复制的数据涉及位移重定位，则标记可能存在问题的任何潜在复制可重定位数据项。构造共享库期间，链接编辑器无法确定将对数据项执行哪种外部引用。因此，只能标记**潜在**问题。
- 生成可执行文件时，标记其数据涉及位移重定位的复制重定位的创建过程。

  然而，在链接编辑时创建复制重定位期间，可能会完成应用于共享库的位移重定位。因此，对引用此共享库的应用程序进行的链接编辑无法确定任何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 标志指示动态库使用的位移重定位，如表 7–34 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[] 的共享库，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，则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，如果创建引用数据项 bar[] 的应用程序，则将创建复制重定位。此复制将导致位移重定位无效。因为链接编辑器可以很清楚地发现此情况，所以无论是否使用 -z verbose 选项，都将生成一条错误消息。

```
$ 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。请使用静态定义或链接编辑器的作用域设置方法。请参见第 57 页中的 “缩减符号范围”。通过使用功能接口访问共享库中的数据可以避免此类型的重定位问题。

调试帮助

Solaris 链接程序附带有调试库。使用此库，可以更详细地跟踪链接编辑过程。此库有助于您了解或调试应用程序和库的链接编辑。使用此库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。

如果您不太了解 ELF 格式，则可能会不熟悉某些调试输出。不过，也许您希望大概了解其中许多方面。

使用 -D 选项可以启用调试。生成的所有输出将被定向到标准错误。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 可以显示可用的标记。

```
$ ld -Dhelp
debug:
debug:          For debugging the link-editing of an application:
debug:                LD_OPTIONS=-Dtoken1,token2 cc -o prog ...
debug:          or,
debug:                ld -Dtoken1,token2 -o prog ...
```

```
debug:            where placement of -D on the command line is significant
debug:            and options can be switched off by prepending with '!'.
debug:
debug:
debug: args       display input argument processing
debug: basic      provide basic trace information/warnings
debug: cap        display hardware/software capability processing
debug: detail     provide more information in conjunction with other options
debug: entry      display entrance criteria descriptors
debug: files      display input file processing (files and libraries)
debug: got        display GOT symbol information
debug: help       display this help message
debug: libs       display library search paths; detail flag shows actual
debug:              library lookup (-l) processing
debug: map        display map file processing
debug: move       display move section processing
debug: reloc      display relocation processing
debug: sections   display input section processing
debug: segments   display available output segments and address/offset
debug:              processing; detail flag shows associated sections
debug: statistics display processing statistics
debug: strtab     display information about string table compression; detail
debug:              shows layout of string tables
debug: support    display support library processing
```

```
debug: symbols    display symbol table processing; detail flag shows
debug:              internal symbol table addition and resolution
debug: tls        display TLS processing info
debug: unused     display unused/unreferenced files; detail flag shows
debug:              unused sections
debug: versions   display version processing
```

注 – 此列表是一个示例，用于显示对链接编辑器有意义的选项。确切选项可能随发行版的不同而有所变化。

大多数编译器驱动程序在预处理阶段解释 -D 选项。因此，LD_OPTIONS 环境变量是一种适合将此选项传递到链接编辑器的机制。

以下示例说明如何跟踪输入文件。确定已找到的库或已从归档中提取的可重定位目标文件时，此语法特别有用。

```
$ 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)
............
```

此例中，从归档库 libfoo.a 中提取了成员 foo.o，以满足对 prog 的链接编辑要求。请注意，对归档搜索了两次，以验证提取 foo.o 是否没有提取其他可重定位目标文件。多个 "again" 诊断指示该归档使用 lorder(1) 和 tsort(1) 进行排序的候选归档。

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，并且被添加到链接编辑器的内部符号表中。此符号引用导致从归档 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。

将 detail 标记和 symbols 标记一起使用，可以观察输入文件处理期间的符号解析详细信息。

```
$ 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

............
```

已使用提取的归档成员 `foo.o` 中的符号定义覆盖 `main.o` 中未定义原始符号 `foo`。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，可直接在链接编辑命令行中放置 `-D` 选项。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。请参见第 28 页中的 “使用编译器驱动程序” 。

第 3 章

运行时链接程序

在初始化和执行**动态可执行文件**的执行过程中，会调用一个**解释程序**将应用程序与其依赖项绑定。在 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` 是共享库的特例。此处使用的版本号为 1。但是，以后的 Solaris 发行版可能会提供更高的版本号。

在执行动态库的过程中，内核将装入该文件并读取程序头信息。请参见第 275 页中的 “程序头”。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。

运行时链接程序的作用：

- 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。

共享库依赖项

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库。通过重复连接引用的共享库及其依赖项，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库，运行时链接程序也只会将该目标文件连接到进程一次。

查找共享库依赖项

链接动态可执行文件时，会显式引用一个或多个共享库。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库的依赖项。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**文件 `libc.so.1` 和 `libm.so.2`。

可以使用 `dump`(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。

```
$ dump -Lvp /usr/bin/cat



/usr/bin/cat:

[INDEX] Tag      Value

[1]     NEEDED   libc.so.1

.........
```

在前面的 dump(1) 示例中，依赖项表示为简单文件名。换言之，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。请参见第 119 页中的 "命名约定" 和第 120 页中的 "记录共享库名称"。

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库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，请参见第 35 页中的 "运行时链接程序搜索的目录"。

使用 dump(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 具有依赖项 libfoo.so.1。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。

```
$ dump -Lvp prog



prog:

[INDEX] Tag      Value

[1]     NEEDED   libfoo.so.1

[2]     NEEDED   libc.so.1

[3]     RUNPATH  /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，根据第 34 页中的 “使用环境变量”和第 35 页中的 “运行时链接程序搜索的目录”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。请参见第 109 页中的 “调试库”。

```
$ 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 ]

...
```

如果无法找到依赖项，则 ldd(1) 会指示找不到该目标文件。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。

- $HWCAP－指示一个可在其中查找提供不同硬件功能的目标文件的目录。 请参见第 367 页中的 “特定于硬件功能的共享库” 。
- $ISALIST－扩展为可在此平台上执行的本机指令集。 请参见第 370 页中的 “特定于指令集的共享库” 。
- $ORIGIN－提供当前目标文件的目录位置。 请参见第 373 页中的 “查找关联的依赖项” 。
- $OSNAME－扩展为操作系统的名称。 请参见第 372 页中的 “特定于系统的共享库” 。
- $OSREL－扩展为操作系统发行版级别。 请参见第 372 页中的 “特定于系统的共享库” 。
- $PLATFORM－扩展为当前计算机的处理器类型。 请参见第 372 页中的 “特定于系统的共享库” 。

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，随可重定位输入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库时，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，请参见第 249 页中的 “重定位类型（特定于处理器）” 。重定位存在两个基本类型。

- 非符号重定位
- 符号重定位

使用 dump(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。

```
$ dump -rvp libbar.so.1



libbar.so.1:



.rela.got:

Offset      Symndx                Type              Addend



0x10438     0                     R_SPARC_RELATIVE  0

0x1043c     foo                   R_SPARC_GLOB_DAT  0
```

第一个重定位是一个简单的相对重定位，这可通过其重定位类型以及值为零的符号索引 (Symndx) 字节看出。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。此符号搜索基于请求目标文件的符号**搜索范围**，以及进程中每个目标文件所提供的符号**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。

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。前者将在下一节第 81 页中的 “缺省符号查找”中概述，后者（也可使用各种符号查找属性）将在第 99 页中的 “符号查找”中介绍。

动态库使用直接绑定时，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 82 页中的 “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* 搜索范围和 *global* 符号可见性。 请参见第 99 页中的 “符号查找”。针对动态可执行文件或随其装入的任何依赖项的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。

如以上各节中所述，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，共享库 libbar.so.1 需要符号 foo 的地址来完成重定位。动态可执行文件 prog 将 libbar.so.1 指定为其依赖项之一。

```
$ ldd prog

        libfoo.so.1 =>   /home/me/lib/libfoo.so.1

        libbar.so.1 =>   /home/me/lib/libbar.so.1
```

运行时链接程序首先会在动态可执行文件 prog 中查找 foo，然后在共享库 /home/me/lib/libfoo.so.1 中查找，最后在共享库 /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 134 页中的 “性能注意事项”中详细介绍。有关替代查找模型，请参见第 82 页中的 “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。

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。

第 40 页中的 “简单解析”中概述了插入如何影响符号解析。第 57 页中的 “缩减符号范围”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。

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。

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此信息直接搜索关联目标文件中的符号，而不执行缺省符号搜索模型。只能对使用链接编辑指定的依赖项建立直接绑定信息。因此，建议使用 -z defs 选项。

可使用下列机制之一建立符号引用与符号定义的直接绑定。

- 使用 -B direct 选项。此选项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。此选项还会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 还允许延迟装入。允许进行延迟装入等效于在链接编辑命令行的前端添加选项 -z lazyload。 请参见第 87 页中的 "延迟装入动态依赖项"。
- 使用 -z direct 选项。此选项在要生成的目标文件与命令行上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。此选项不会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 属性。此属性用于直接绑定各个符号。 请参见第 49 页中的 "定义其他符号"。

直接绑定可以大大降低由于包含许多符号重定位与依赖项的动态进程而导致的符号查找开销。此模型还允许在已直接绑定到的不同目标文件中查找同名的多个符号。

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。

缺省符号搜索模型允许将符号的所有引用绑定到某个定义。由于直接绑定忽略缺省搜索模型，因此直接绑定禁用隐式插入符号。但是，在搜索提供符号定义的目标文件之前，会先搜索任何显式标识为插入项的目标文件。显式插入项包括使用环境变量 LD_PRELOAD 装入的目标文件，或使用链接编辑器的 -z interpose 选项创建的目标文件。 请参见第 82 页中的 "插入"。

某些接口可为缺省技术提供替代实现。这些接口期望其实现成为该技术在进程内的唯一实例。例如 malloc(3C) 系列。malloc() 系列实现有多种，并且每个系列都期望成为进程中使用的唯一实现。应该避免直接绑定到此类系列中的接口，否则同一进程可能会引用该技术的多个实例。例如，进程中的一个依赖项可针对 libc.so.1 直接绑定，而另一个依赖项可针对 libmapmalloc.so.1 直接绑定。对 malloc() 和 free() 的两种不同实现的不一致使用很可能会产生错误。

提供期望成为进程中的单一实例的接口的目标文件应该避免直接绑定到其接口。为防止任何调用方直接绑定到某接口，可以使用下列机制之一标记该接口。

- 使用 -B nodirect 选项。此选项禁止直接绑定到目标文件提供的所有接口。
- 使用 NODIRECT mapfile 属性。此属性用于禁止直接绑定到各个符号。 请参见第 49 页中的 "定义其他符号"。

非直接标记禁止任何符号引用直接绑定到实现。用于满足引用要求的符号搜索将使用缺省符号搜索模型。在生成随 Solaris 提供的各种 malloc() 系列实现时，使用了非直接标记。

注 – NODIRECT mapfile 指令可与命令行选项 -B direct 或 -z direct 组合使用。未显式定义 NODIRECT 的符号位于该命令行指令之后。同样，DIRECT mapfile 指令也可与命令行选项 -B nodirect 组合使用。未显式定义 DIRECT 的符号位于该命令行指令之后。

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库进行的函数调用。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。

延迟引用指的是在目标文件执行时可确定的重定位。这些引用通常是通过与位置无关的共享库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 .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时，控制权会移交给运行时链接程序。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，可以推迟此类型的重定位，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，请参见第 44 页中的“生成可执行的输出文件”。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 共享库的版本生成了 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库的重定位要求。

如果在使用 ldd(1) 时指定 -d 选项，将列显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

运行时链接程序允许您使用环境变量 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库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* 搜索范围和 *global* 符号可见性。

在以下示例中，首先装入动态可执行文件 prog，然后装入共享库 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。

```
$ LD_PRELOAD=./newstuff.so.1 ldd 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库。然后，会按照前面示例中预装入共享库 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：

```
$ 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库中的函数的新实现。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库中的函数。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 dlopen(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

在内存中装入动态库时，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。请参见第 84 页中的 “执行重定位的时间”。通常，建议不要在动态库之间进行即时引用。

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。对于更复杂的与已知依赖项的交互，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上与位置相关。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 没有这样操作。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，动态库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

延迟装入可以提供替代 dlopen(3C) 和 dlsym(3C) 的方法。 请参见第 96 页中的 “运行时链接编程接口” 。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);
```

```
        ....

}
```

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。

使用延迟装入，可以实现同样的 libbar.so.1 推迟装入。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。此外，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. -zlazyload -zdefs -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。

带有 RTLD_PROBE 句柄的 dlsym(3C) 可用于验证依赖项是否存在以及是否将其装入。例如，对 bar1() 的引用可以验证在链接编辑时建立的延迟依赖项是否可用。此测试可用于控制对依赖项以使用 dlopen(3C) 的方式提供的函数的引用。

```
void foo()

{

    if (dlsym(RTLD_PROBE, "bar1")) {

        bar1(arg1);

        bar2(arg2);

        ....

}
```

此方法允许安全推迟装入已记录的依赖项以及标准函数调用。

注 – 特殊句柄 RTLD_DEFAULT 提供的机制与使用 RTLD_PROBE 类似。但是，使用 RTLD_DEFAULT 可能会导致在尝试查找的符号不存在时处理所有暂挂延迟装入目标文件。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。

建议使用 -z defs 选项来生成利用延迟装入的所有目标文件。

初始化和终止例程

动态库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库时，都会执行一次动态库的初始化代码。每次在进程中卸载动态库或进程终止时，都会执行一次动态库的终止代码。

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库，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库时创建。

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库中执行。

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。

动态库还可提供终止节。终止节 .fini_array 和 .fini 由链接编辑器在生成动态库时创建。

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，请参见第 36 页中的 “初始化和终止节” 。

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。装入其他目标文件或更改已装入目标文件的重定位模式的初始化节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。

对于 Solaris 2.6 之前的发行版，依赖项初始化例程是以**相反**装入顺序（即，使用 `ldd(1)` 显示的依赖项的相反顺序）调用的。同样，依赖项终止例程是以装入顺序调用的。但是，由于依赖项分层结构较为复杂，因此这种简单排序方法可能不适合。

对于 Solaris 2.6 发行版，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。

注意 – 在 Solaris 8 10/00 发行版之前，环境变量 `LD_BREADTH` 可设置为非空值。此设置强制运行时链接程序按 Solaris 2.6 发行版之前的顺序执行初始化节和终止节。该功能自此发行版以后已被禁用，因为许多应用程序的初始化依赖项变得很复杂并且要求进行拓扑排序。 现已忽略 `LD_BREADTH` 设置而无任何提示。

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：

```
$ dump -Lv B.so.1 | grep NEEDED

[1]     NEEDED      C.so.1

$ dump -Lv C.so.1 | grep NEEDED

[1]     NEEDED      B.so.1

$ dump -Lv main | grep NEEDED

[1]     NEEDED      A.so.1

[2]     NEEDED      B.so.1

[3]     NEEDED      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。 请参见第 109 页中的 “调试库” 。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。定义所有动态库的依赖项要求。请参见第 47 页中的 “生成共享库输出文件”。不要表示不需要的依赖项。请参见第 31 页中的 “共享库处理”。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库和应用程序开发的影响。请参见第 124 页中的 “依赖项排序”。

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。

如果 issetugid(2) 系统调用对某进程返回 true，则运行时链接程序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。 请参见第 76 页中的 “运行时链接程序搜索的目录” 。

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，运行路径必须是全路径名，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。 请参见第 377 页中的 “安全” 。

在安全进程中，LD_CONFIG 会被忽略。如果存在缺省配置文件，则安全进程将使用该配置文件。 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全路径名或简单文件名。全路径名仅限于已知的可信目录。简单文件名（名称中没有 '/'）的查找受前面描述的搜索路径限制的约束。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。以全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库。

- 使用 dlopen(3C) 来查找共享库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库的所有依赖项。

- 重定位添加的共享库及其依赖项。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，并且通过共享库 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 Solaris 以前的发行版中，共享库 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。请参见第 76 页中的 “运行时链接程序搜索的目录”。例如，文件 main.c 包含以下代码片段。

```
#include        <stdio.h>

#include        <dlfcn.h>



main(int argc, char ** argv)

{

        void *  handle;

        .....
```

```
        if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
                (void) printf("dlopen: %s\n", dlerror());
                exit (1);
        }
        .....
```

要查找共享库 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) 指定的任何共享库。有关版本化的更多信息，请参见第 175 页中的 “协调版本化文件名” 。

如果找不到所需目标文件，则 dlopen(3C) 会返回 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，如果前面的 main.c 示例包含以下 dlopen() 调用：

```
if ((handle = dlopen((const char *)0, RTLD_LAZY)) == NULL) {
```

则通过指定了 RTLD_GLOBAL 标志的 dlopen(3C)，可使用从 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

如第 3 章中所述，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。

可根据重定位的时间对重定位进行分类。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）。

该机制也适用于模式定义为 RTLD_LAZY 时使用 dlopen(3C) 添加的所有目标文件。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 82 页中的 “直接绑定” 。

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**搜索范围**，第二个属性是进程中每个目标文件提供的符号**可见性**。目标文件的搜索范围可以是：

world
　目标文件可以查找进程中的任何其他全局目标文件。

group
　目标文件只能查找同一**组**中的目标文件。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。

目标文件中符号的可见性可以是：

global
  可在具有 *world* 搜索范围的任何目标文件中引用该目标文件的符号。

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符号。

缺省情况下，使用 dlopen(3C) 获取的目标文件将被指定 *world* 符号搜索范围和 *local* 符号可见性。第 100 页中的 "缺省符号查找模型"一节将使用此缺省模型说明典型的目标文件组交互。第 103 页中的 "定义全局目标文件"、第 103 页中的 "隔离组"和第 104 页中的 "目标文件分层结构"等节将介绍使用 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 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，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 和共享库 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) 获取共享库 B.so.1，则会首先在 prog 中查找重定位共享库 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) 获取的共享库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。

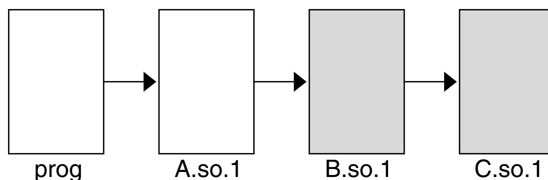


图 3–1 单个 `dlopen()` 请求

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库 `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范围。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目标文件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范围。此范围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，假定共享库 `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) 装入此共享库及共享库 `B.so.1`。下图说明了目标文件之间的符号查找关系。

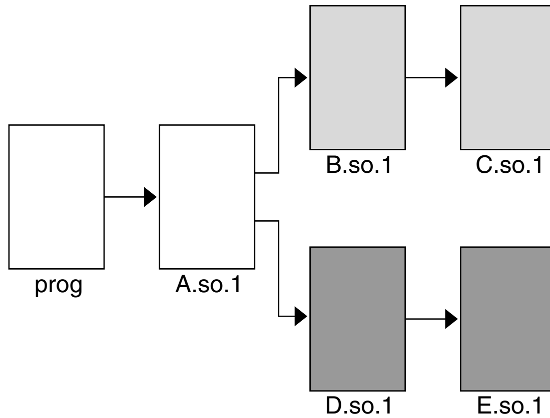


图 3–2 多个 dlopen() 请求

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，共享库 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库执行 dlopen(3C)。由于共享库 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。

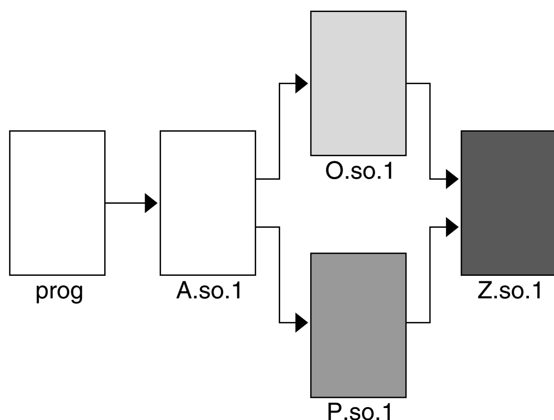


图 3–3 具有公共依赖项的多个 `dlopen()` 请求

`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库间不同，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，具有全局符号搜索范围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、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 – 如果某个组成员具有局部符号可见性，并且被另一个需要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范围缩小为组。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范围。

注 – 如果某个组成员具有组搜索功能，并且被另一个需要全局搜索功能的组引用，则该目标文件的搜索功能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) 依赖项树的一部分，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方。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) 获取且与调用方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 允许从调用方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>



main()

{

        void *  handle;

        int *   dptr, (* fptr)();



        if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {

                (void) printf("dlopen: %s\n", dlerror());

                exit (1);

        }



        if (((fptr = (int (*)())dlsym(handle, "foo")) == NULL) ||

            ((dptr = (int *)dlsym(handle, "bar")) == NULL)) {
```

```
                (void) printf("dlsym: %s\n", dlerror());
                exit (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。接下来，在 return() 语句中使用单个参数 bar 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第 81 页中的 “重定位符号查找” 中所述的模型。

如果找不到所需符号，则 dlsym(3C) 会返回 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 找不到符号 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。符号搜索采用重定位调用目标文件时所用的同一模型。请参见第 100 页中的 “缺省符号查找模型” 。例如，如果应用程序 prog 包含以下代码片段：

```
        if ((fptr = (int (*)())dlsym(RTLD_DEFAULT, "foo")) != NULL)
                (*fptr)();
```

则会首先在 prog 中搜索 foo，然后在 /lib/libc.so.1 中搜索。如果此代码片段包含在图 3–1 中显示的示例的文件 B.so.1 中，则会继续在 B.so.1 中搜索 foo，然后在 C.so.1 中搜索。

此机制为未定义的弱引用的使用提供了强大而灵活的替代方案，如第 47 页中的 “弱符号”中所述。

使用插入

使用特殊句柄 RTLD_NEXT，应用程序可在符号范围内查找下一个符号。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());

                exit (1);

        }



        return ((*fptr)());
```

则会在与 prog 关联的共享库（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 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，在共享库 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 (0);
                }
        }

        (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，可使用以下代码实现相同插入：

```
$ 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

Solaris 链接程序附带有调试库和 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。

调试库

此调试库有助于了解或调试应用程序和依赖项的执行。使用此库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，也许您希望大概了解其中许多方面。

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，并且在缺省情况下会指示为标准错误。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。当进程在显示信息后终止时，可以使用任何动态可执行文件请求此信息。

```
$ LD_DEBUG=help prog

11693:

11693:          For debugging the runtime linking of an application:

11693:                  LD_DEBUG=token1,token2  prog

11693:          enables diagnostics to the stderr.  The additional
```

```
11693:           option:
11693:                  LD_DEBUG_OUTPUT=file
11693:           redirects the diagnostics to an output file created
11593:           using the specified name and the process id as a
11693:           suffix.  All diagnostics are prepended with the
11693:           process id.
11693:
11693:
11693: audit     display runtime link-audit processing
11693: basic     provide basic trace information/warnings
11693: bindings  display symbol binding; detail flag shows
11693:             absolute:relative addresses
11693: cap       display hardware/software capability processing
11693: detail    provide more information in conjunction with other
11693:             options
11693: files     display input file processing (files and libraries)
11693: help      display this help message
11693: init      display init and fini processing
11693: libs      display library search paths
11693: move      display move section processing
11693: reloc     display relocation processing
11693: symbols   display symbol table processing
11693: tls       display TLS processing info
11693: unused    display unused/unreferenced files
11693: versions  display version processing
```

此示例显示对运行时链接程序有意义的选项。确切选项可能随发行版的不同而有所变化。

环境变量 LD_DEBUG_OUTPUT 可用来指定使用输出文件来代替标准错误。进程标识符将作为后缀添加到输出文件。

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库。

```
$ cat bar.c

int bar = 10;

$ cc -o bar.so.1 -K pic -G bar.c



$ cat foo.c

foo(int data)

{

        return (data);

}

$ cc -o foo.so.1 -K pic -G foo.c



$ cat main.c

extern  int     foo();

extern  int     bar;



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: .......
```

符号 bar 是即时重定位所需的，将在应用程序获取控制权**之前**绑定。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。

当运行时链接程序执行函数重定位时，将重写与函数 .plt 关联的数据。通过 .plt 进行的后续调用将直接转至该函数。环境变量 LD_BIND_NOT 可设置为任何值以免更新此数据。通过将此变量与对详细绑定的调试请求一起使用，可以全盘了解所有函数绑定的运行时情况。此组合可能会产生大量输出，从而导致应用程序性能下降。

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:.  (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:.  (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，请参见第 144 页中的“复制重定位”。

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) 与运行时链接程序调试器模块一起使用的简单情况。

```
$ 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
```

进程中的每个动态库都表示为链接映射 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) 在设置推迟断点时也很有用。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。推迟断点会指示调试器在装入动态库时设置实际断点。

```
> ::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，请参阅用法和帮助信息。

第 4 章

共享库

共享库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，共享库 libfoo.so.1 根据输入文件 foo.c 生成。

```
$ cc -o libfoo.so.1 -G -K pic foo.c
```

共享库是一个**不可分割的**单元，通过一个或多个可重定位的目标文件生成。共享库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，共享库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库的共享库，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库，就像已将这些共享库添加到要生成的输出文件的逻辑地址空间。共享库的所有功能均可用于输出文件。

所有输入共享库都变成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库的处理。

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库。这些环境在第 23 页中的“运行时链接”中介绍。

命名约定

链接编辑器和运行时链接程序都不根据文件名解释文件。将检查所有文件以确定其 ELF 类型（请参见第 214 页中的“ELF 头”）。链接编辑器使用此信息来推导文件的处理要求。但是，共享库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。

当共享库用作编译环境的一部分时，链接编辑器将读取和处理这些共享库。虽然可以在传递到链接编辑器的命令中根据显式文件名指定这些共享库，但是通常使用 -l 选项来利用链接编辑器的库搜索功能。请参见第 31 页中的“共享库处理”。

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库。例如，`/lib/libc.so` 便是可用于编译环境的标准 C 库的共享库。根据约定，64 位共享库位于 `lib` 目录名为 `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`。

当共享库用作运行时环境的一部分时，运行时链接程序将读取和处理这些共享库。要允许在一系列软件发行版中对共享库的导出接口进行更改，请将共享库作为**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 库的共享库。

如果从不打算在编译环境中使用共享库，则可能会从共享库名称中删除常规的 `lib` 前缀。仅用于 `dlopen`(3C) 的共享库便是此类共享库。仍然建议使用后缀 `.so` 来表明实际文件类型。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库绑定。第 5 章更详细地介绍了版本控制。

注 – `dlopen`(3C) 中使用的共享库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，请参见第 86 页中的 "装入其他目标文件" 。

记录共享库名称

缺省情况下，动态可执行文件或共享库中的依赖项记录将是链接编辑器所引用的关联共享库的文件名。例如，根据同一共享库 `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

$ dump -Lv prog | grep NEEDED

[1]     NEEDED   libfoo.so



$ cc -o prog main.o ../tmp/libfoo.so

$ dump -Lv prog | grep NEEDED

[1]     NEEDED   ../tmp/libfoo.so
```

```
$ cc -o prog main.o /usr/tmp/libfoo.so

$ dump -Lv prog | grep NEEDED

[1]     NEEDED   /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库的位置也可能会与已安装系统上的共享库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，共享库可以在自身中记录运行时引用共享库应依据的文件名。

在链接编辑共享库过程中，可以使用 `-h` 选项在共享库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库的运行时名称 `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 `dump(1)` 并引用具有 `SONAME` 标记的项来显示 soname 记录。

```
$ dump -Lvp ../tmp/libfoo.so



../tmp/libfoo.so:

[INDEX] Tag      Value

[1]     SONAME   libfoo.so.1

.........
```

当链接编辑器处理包含 soname 的共享库时，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

$ dump -Lv prog | grep NEEDED

[1]     NEEDED   libfoo.so.1



$ cc -o prog main.o ../tmp/libfoo.so

$ dump -Lv prog | grep NEEDED
```

```
[1]     NEEDED   libfoo.so.1
```

```
$ cc -o prog main.o /usr/tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   libfoo.so.1
```

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，请参见第 76 页中的 “查找共享库依赖项” 。

在归档文件中包含共享库

如果共享库始终通过归档库进行处理，则在共享库中记录 soname 是基本机制。

归档文件可以根据一个或多个共享库生成，并可用于生成动态可执行文件或共享库。可以从归档文件中提取共享库来满足链接编辑的要求。与处理可重定位的目标文件（串联成要创建的输出文件）不同，从归档文件中提取的任何共享库都将记录为依赖项。有关归档文件提取条件的更多详细信息，请参见第 30 页中的 “归档处理” 。

归档成员的名称由链接编辑器构造，并且由归档文件名称和归档文件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
$ dump -Lv main | grep NEEDED
[1]     NEEDED   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， 在共享库中提供 soname 是生成依赖项有意义运行时文件名的唯一方法。

注 – 运行时链接程序不会从归档文件中提取目标文件。因此， 在上一示例中， 必须从归档文件中提取所需的共享库依赖项并使其可用于运行时环境。

已记录名称冲突

使用共享库创建动态可执行文件或其他共享库时，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。

如果用作链接编辑的输入文件的两个共享库包含相同的soname，则会出现依赖项名称冲突。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
ld: fatal: File processing errors. No output written to prog
```

如果某个没有已记录soname的共享库的文件名与同一链接编辑过程使用的其他共享库的soname匹配，则也会出现类似的错误情况。

如果要生成的共享库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
ld: fatal: File processing errors. No output written to libbar.so
```

具有依赖项的共享库

共享库可以具有自己的依赖项。运行时链接程序查找共享库依赖项使用的搜索规则在第76页中的"运行时链接程序搜索的目录"中介绍。如果共享库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。可以使用链接编辑器的`-R`选项来记录运行路径。

在以下示例中，共享库`libfoo.so`依赖于`libbar.so`，而在运行时预期后者位于目录`/home/me/lib`中，如果在该目录中未找到该项，则该项位于缺省位置中。

```
$ cc -o libbar.so -G -K pic bar.c
$ cc -o libfoo.so -G -K pic foo.c -R/home/me/lib -L. -lbar
$ dump -Lv libfoo.so
```

```
libfoo.so:

  **** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag      Value
[1]     NEEDED   libbar.so
[2]     RUNPATH  /home/me/lib
.........
```

共享库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库的任何依赖项。

环境变量 LD_LIBRARY_PATH 的范围更具全局性。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库依赖项。虽然此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此变量。有关更全面的介绍，请参见第 76 页中的 "运行时链接程序搜索的目录"。

依赖项排序

当动态可执行文件和共享库都依赖于相同的常用共享库时，这些目标文件的处理顺序可能会变得有些难以预测。

例如，假设共享库开发者生成的 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
```

如果使用此共享库创建动态可执行文件 prog，并定义显式依赖 libC.so.1，则生成的共享库顺序如下：

```
$ 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库 `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库处理顺序中存在这种潜在的更改。

作为过滤器的共享库

可以定义共享库以将其用作**过滤器**。此技术涉及将过滤器提供的接口与备用共享库进行关联。在运行时，此备用共享库可提供**过滤器**所提供的一个或多个接口。此备用共享库称为 *filtee*。filtee 的生成方式与任意共享库的生成方式相同。

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库。

使用 `mapfile` 指令 `FILTER` 或 `AUXILIARY`，可以将在共享库中定义的单个接口定义为过滤器。或者，共享库可以使用链接编辑器的 `-F` 或 `-f` 标志将共享库提供的所有接口定义为过滤器。通常单独使用这些技术。请参见第 126 页中的 “生成标准过滤器”和第 129 页中的 “生成辅助过滤器” 。还可以在同一共享库中组合使用这些技术。 请参见第 133 页中的 “过滤组合” 。

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` 指令 `FILTER` 或链接编辑器的 `-F` 标志，可以定义接口以将其用作标准过滤器。此 `mapfile` 指令或标志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。

将跳过在运行时无法处理的 filtee。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 指令 AUXILIARY 或链接编辑器的 -f 标志，可以定义接口以将其用作辅助过滤器。此 mapfile 指令或标志使用在运行时提供符号定义的一个或多个 filtee 的名称进行限定。

将跳过在运行时无法处理的 filtee。如果在 filtee 中找不到辅助过滤器符号，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。

生成标准过滤器

要生成标准过滤器，应先定义要应用过滤的 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。要将共享库提供的所有接口都声明为过滤器，请使用链接编辑器的 -F 标志。要将共享库的单个接口声明为过滤器，请使用链接编辑器 mapfile 和 FILTER 指令。

在以下示例中，将共享库 filter.so.1 定义为过滤器。filter.so.1 提供符号 foo 和 bar，并且是 filtee filtee.so.1 的过滤器。在本示例中，使用环境变量 LD_OPTIONS 禁止编译器驱动程序解释 -F 选项。

```
$ cat filter.c

char * bar = 0;
```

```
char * foo()

{

    return (0);

}

$ 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库时，链接编辑器可将标准过滤器 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，对该过滤器符号的任何引用都会导致对此 filtee filtee.so.1 的额外装入。运行时链接程序使用此 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();



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，共享库 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，使用 mapfile 指令 FUNCTION 以确保创建 foo 的符号表项。

```
$ cat filter.c
char * bar = "defined in filter";
$ cat mapfile
{
    global:
        foo = 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对 filtee filtee.so.1 的额外装入。运行时链接程序使用此 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。不能使用此 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库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库创建接口组。标准过滤器还提供一种将接口重定向到实现的方法。在 Solaris OS 中，可以使用多个标准过滤器。

/usr/lib/libsys.so.1 过滤器提供标准 C 库 /usr/lib/libc.so.1 的子集。此子集表示位于必须由兼容应用程序导入的 C 库中兼容 ABI 的函数和数据项。

/lib/libxnet.so.1 过滤器使用多个 filtee。 此库提供来自 /lib/libsocket.so.1、/lib/libnsl.so.1 和 /lib/libc.so.1 的套接字接口和 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 strerror(3C)。以前，libnsl.so.1 和 libc.so.1 针对此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。当 libnsl.so.1 继续导出 gethostname() 时，此库接口将一直兼容早期发行版。

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。请参见第 49 页中的 “定义其他符号” 。

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。请使用链接编辑器的 -64 选项以仅通过 mapfile 创建 64 位过滤器。

在过滤器中生成数据符号时，始终会初始化数据项。生成的数据定义可确保从动态可执行文件正确地建立引用。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。请参见第 39 页中的 “符号解析” 。

生成辅助过滤器

要生成辅助过滤器，应先定义要应用过滤的 filtee。以下示例将生成 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。要将共享库提供的所有接口都声明为辅助过滤器，请使用链接编辑器的 `-f` 标志。要将共享库的单个接口声明为辅助过滤器，请使用链接编辑器 `mapfile` 和 `AUXILIARY` 指令。

在以下示例中，会将共享库 `filter.so.1` 定义为辅助过滤器。`filter.so.1` 提供符号 `foo` 和 `bar`，并且是 filtee `filtee.so.1` 的辅助过滤器。在本示例中，使用环境变量 `LD_OPTIONS` 禁止编译器驱动程序解释 `-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库时，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`。如果找到此 filtee，则运行时链接程序会使用此 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，则会使用过滤器中的初始符号。

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**从过滤器 filter.so.1 中获取。但是，从过滤器 filter.so.1 中获取 bar，因为此符号在 filtee filtee.so.1 中没有备选定义。

```
$ cat main.c

extern char * bar, * foo();



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，共享库 filter.so.2 将它的一个接口 foo 定义为 filtee filtee.so.1 的辅助过滤器。

```
$ cat filter.c

char * bar = "defined in filter";



char * foo()

{

        return ("defined in filter");

}

$ cat mapfile

{

    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。如果找到此 filtee，则会装入此 filtee。然后，使用此 filtee 来解析 filter.so.2 定义的符号 foo。如果未找到此 filtee，则使用 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。如果 filtee filtee.so.1 存在，则执行 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
```

如果 filtee filtee.so.1 不存在，则执行 prog 会导致从过滤器 filter.so.2 中获取 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。不能使用此 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库的备用接口的机制。在 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库。有关示例，请参见第 367 页中的“特定于硬件功能的共享库”、第 370 页中的“特定于指令集的共享库”和第 372 页中的“特定于系统的共享库”。

注 – 可以设置环境变量 LD_NOAUXFLTR 以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

可以在同一共享库中同时定义用于定义标准过滤器的单个接口以及用于定义辅助过滤器的单个接口。通过使用 mapfile 指令 FILTER 和 AUXILIARY 指定所需的 filtee，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库可以是标准过滤器，也可以是辅助过滤器。

共享库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，请考虑过滤器 filter.so.1。此过滤器使用链接编辑器的 -f 标志，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 指令 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 指令 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 中未找到 foo，则不会对过滤器执行进一步处理。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 中未找到 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 中未找到 bar，则最终会使用过滤器 filter.so.1 中的 bar 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。 这种实现类似于过滤器在需要 filtee 时，使用模式 RTLD_LOCAL 对每个 filtee 执行 dlopen(3C)。这种实现考虑了由诸如 ldd(1) 的工具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 选项以便在运行时立即处理 filtee。此外，通过将 LD_LOADFLTR 环境变量设置为任意值，可触发在进程中立即处理所有的 filtee。

性能注意事项

一个共享库可供同一系统中的多个应用程序使用。共享库的性能会影响使用此共享库的应用程序，并且会影响整个系统。

虽然共享库中的代码会直接影响运行进程的性能，但此处讨论的性能问题则涉及共享库的运行时处理。本节通过考虑各个方面（如文本大小和纯度）以及重定位开销，更详细地介绍了这种处理。

分析文件

有多种工具可用来分析 ELF 文件的内容。要显示文件的大小，请使用 size(1) 命令。

```
$ size -x libfoo.so.1

59c + 10c + 20 = 0x6c8



$ size -xf libfoo.so.1

..... + 1c(.init) + ac(.text) + c(.fini) + 4(.rodata) + \

..... + 18(.data) + 20(.bss) .....
```

第一个示例指明共享库**文本**、**数据**以及 *bss*（SunOS 操作系统早期发行版中使用的一种分类）的大小。

ELF 格式通过将数据组织到多个**节**中，为表示文件中的数据提供了更佳的粒度。第二个示例显示了文件的每个可装入节的大小。

分配给单元的各节称为**段**，某些段描述如何将文件的各部分映射到内存。请参见 mmap(2)。可以使用 dump(1) 命令并检查 LOAD 项来显示这些可装入段。

```
$ dump -ov libfoo.so.1



libfoo.so.1:

 ***** PROGRAM EXECUTION HEADER *****

Type        Offset      Vaddr       Paddr

Filesz      Memsz       Flags       Align
```

```
LOAD        0x94        0x94        0x0

0x59c       0x59c       r-x         0x10000


LOAD        0x630       0x10630     0x0

0x10c       0x12c       rwx         0x10000
```

在共享库 `libfoo.so.1` 中存在两种可装入段，通常称为**文本**段和**数据**段。将映射文本段以允许读取和执行其内容 (`r-x`)，同时将映射数据段以允许修改其内容 (`rwx`)。数据段的内存大小 (`Memsz`) 不同于文件大小 (`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。可以使用 `nm`(1) 显示这些符号。 例如：

```
$ nm -x libfoo.so.1


[Index]   Value      Size      Type  Bind  Other Shndx   Name

.........

[39]    |0x00000538|0x00000000|FUNC |GLOB |0x0  |7      |_init

[40]    |0x00000588|0x00000034|FUNC |GLOB |0x0  |8      |foo

[41]    |0x00000600|0x00000000|FUNC |GLOB |0x0  |9      |_fini

[42]    |0x00010688|0x00000010|OBJT |GLOB |0x0  |13     |data

[43]    |0x0001073c|0x00000020|OBJT |GLOB |0x0  |16     |bss

.........
```

可以通过引用符号表中的节索引 (`Shndx`) 字段并使用 `dump`(1) 显示文件各节来确定包含符号的节。 例如：

```
$ dump -hv libfoo.so.1


libfoo.so.1:
```

```
          **** SECTION HEADER TABLE ****
[No]    Type    Flags   Addr      Offset    Size      Name
.........
[7]     PBIT    -AI     0x538     0x538     0x1c      .init


[8]     PBIT    -AI     0x554     0x554     0xac      .text


[9]     PBIT    -AI     0x600     0x600     0xc       .fini
.........
[13]    PBIT    WA-     0x10688   0x688     0x18      .data


[16]    NOBI    WA-     0x1073c   0x73c     0x20      .bss
.........
```

上述 nm(1) 和 dump(1) 示例的输出显示函数 _init、foo 和 _fini 与节 .init、.text 和 .fini 关联。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。

注 – 先前的 dump(1) 显示已针对本示例进行了简化。

基础系统

使用共享库生成应用程序时，会在运行时将此目标文件的全部可装入内容映射到此进程的虚拟地址空间。每个使用共享库的进程通过引用内存中此共享库的单个副本来启动。

处理共享库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共享库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。

基于动态共享库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库的引用时写入一个共享库页，则会生成此页的专用副本。此专用副本不会影响此共享库的其他用户。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。

映射到内存的共享库段分为两种基本类别，分别是只读的**文本**段和可读写的**数据**段。有关如何从 ELF 文件中获取此信息，请参见第 134 页中的 “分析文件” 。开发共享库时的最重要目标是最大化文本段以及最小化数据段。这样可优化代码共享量，同时减少初始化和使用共享库所需的处理量。本节介绍有助于实现此目标的机制。

延迟装入动态依赖项

通过将共享库建立为延迟可装入目标文件，可以延迟装入该目标文件依赖项，直到首次引用依赖项。请参见第 87 页中的 “延迟装入动态依赖项” 。

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，会延迟依赖项处理，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与位置**相关**，并且与内存中的固定地址关联。相反，共享库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。建议在创建共享库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。

如果共享库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。文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库时，会通过共享库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，请参见第 299 页中的 “全局偏移表（特定于处理器）” 和第 299 页中的 “过程链接表（特定于处理器）” 。

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。

通常，x64 位置相关代码序列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。由于共享库通常装入内存高位，因此需要地址的高 32 位。这样，x64 共享库中位置相关代码便无法满足重定位要求。在共享库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 共享库的要求。

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 共享库中。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，共享库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库。以下示例使用 dump(1) 确定是否存在 TEXTREL 项动态项。

```
$ cc -o libfoo.so.1 -G -R. foo.c
$ dump -Lv libfoo.so.1 | grep TEXTREL
[9]     TEXTREL  0
```

注 – TEXTREL 项的值无关紧要。共享库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库，请使用链接编辑器的 -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库。

```
$ 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
```

将根据文本段生成两个重定位，因为通过文件 foo.o 生成了位置相关代码。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，则在共享库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。

SPARC: -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。请参见第 299 页中的 "全局偏移表（特定于处理器）"。

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。以下代码序列使用 -K pic 引用项：

```
        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以下致命错误：

```
$ 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，请使用 -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了 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
```

可以使用带有 -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的可重定位目标文件。

删除未使用的材料

包含要生成的目标文件未使用的函数和数据是一种浪费。此类材料使得目标文件变得过大，从而导致不必要的重定位开销以及关联的换页活动。对未使用的依赖项的引用也是一种浪费。这些引用会导致不必要地装入和处理其他共享库。

使用链接编辑器调试标记 -D unused 时，会在链接编辑过程中显示未使用的节。应该从链接编辑中删除标识为未使用的节。可以使用链接编辑器 -z ignore 选项删除未使用的节。

在以下情况下，链接编辑器会将可重定位目标文件中的节标识为未使用：

- 此节可分配
- 没有其他节绑定（重定位）到此节
- 此节不提供任何全局符号

通过定义共享库的外部接口可以改进链接编辑器删除节的功能。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。现在，可以将未从其他目标文件引用的降级后符号明确标识为删除目标文件。

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删除这些项。可以使用诸如 -xF 的编译器选项完善此节。较早的编译器仅可用于将函数指定给其自己的节。较新的编译器已扩展了 -xF 语法，可以将数据变量指定给其自己的节。较早的编译器要求在使用 -xF 时禁用 C++ 异常处理。较新的编译器中已删除了此限制。

如果可以删除可重定位目标文件中的所有可分配节，则在链接编辑时会放弃整个文件。

除了删除输入文件之外，链接编辑器还可标识未使用的依赖项。如果要生成的目标文件未绑定某个依赖项，则会将此依赖项视为未使用。可以使用 -z ignore 选项生成目标文件，以避免记录未使用的依赖项。

-z ignore 选项仅应用于链接编辑命令行中该选项后的文件。可以使用 -z record 取消 -z ignore 选项。

最大化可共享性

如第 136 页中的 “基础系统”中所述，只有共享库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，每个使用共享库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" };
```

更佳定义可能如下：

```
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) 查看共享库中的字符串用法。以下示例在文件 libfoo.so.1 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。

最小化换页活动

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。由于共享库可供许多进程使用，因此，减少由于访问共享库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，如果每次调用函数 foo() 都会导致调用函数 bar()，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) 的工具来确定代码适用范围和配置。

应将相关功能与其共享库隔离开来。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，刚开始设计共享库时，只在此共享库中维护相关函数。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

在第 80 页中的 “重定位处理” 中，介绍了运行时链接程序重定位动态可执行文件和共享库以创建可运行进程所依据的机制。第 81 页中的 “重定位符号查找” 和第 84 页中的 “执行重定位的时间” 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，缺省情况下它会通过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，然后按照共享库的装入顺序搜索每个共享库。在多数情况下，会找出需要符号重定位的共享库以提供符号定义。

在这种情况下，如果不需要此重定位所用的符号成为共享库接口的一部分，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库接口中删除符号。 有关更多详细信息，请参见第 57 页中的 “缩减符号范围” 。通过进行上述转换，链接编辑器在创建共享库过程中，会产生针对这些符号处理符号重定位的开销。

应在共享库中可见的全局数据项只是那些属于此共享库用户界面的数据项。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。请参见第 57 页中的 “缩减符号范围” 。减少从共享库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 82 页中的 “直接绑定” 。

何时执行重定位

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。

第 144 页中的 “复制重定位” 一节中介绍了使用功能接口的优点。该节介绍了一种在动态可执行文件与共享库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

缺省情况下，按照应用重定位的节对重定位进行分组。但是，当使用 `-z combreloc` 选项生成目标文件时，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。请参见第 299 页中的 “过程链接表（特定于处理器）” 。

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

共享库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，请参见第 137 页中的 “与位置无关的代码” 。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库中。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库中的数据项的大小。还为此空间指定在共享库中定义的符号名称。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库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库引用。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此副本。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用于应用程序中：

```
$ 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]);
}
```

应用程序使用函数 error 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行文件进行检查，可以更详细地显示复制重定位的实现：

```
$ cc -o prog main.c foo.c
$ nm -x prog | grep sys_
[36]  |0x00020910|0x00000260|OBJT |WEAK |0x0  |16 |sys_errlist
[37]  |0x0002090c|0x00000004|OBJT |WEAK |0x0  |16 |sys_nerr
$ dump -hv prog | grep bss
[16]    NOBI    WA-    0x20908   0x908    0x268   .bss
$ dump -rv prog

    **** RELOCATION INFORMATION ****

.rela.bss:
Offset      Symndx              Type            Addend

0x2090c     sys_nerr            R_SPARC_COPY      0
0x20910     sys_errlist         R_SPARC_COPY      0
..........
```

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 库中复制的。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库而言是本地数据，因此数据不再是其接口的一部分。因此，共享库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，从共享库接口中删除数据项会改善性能，同时使得共享库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 最初根据共享库 libfoo.so.1 生成，并且已记录以下两个复制重定位：

```
$ nm -x prog | grep _size_

[36]   |0x000207d8|0x40|OBJT |GLOB |15  |_size_gets_smaller

[39]   |0x00020818|0x40|OBJT |GLOB |15  |_size_gets_larger

$ dump -rv size | grep _size_

0x207d8     _size_gets_smaller    R_SPARC_COPY      0

0x20818     _size_gets_larger     R_SPARC_COPY      0
```

以下是此共享库的新版本，其中包含这些符号的不同数据大小：

```
$ nm -x libfoo.so.1 | grep _size_

[26]   |0x00010378|0x10|OBJT |GLOB |8   |_size_gets_smaller

[28]   |0x00010388|0x80|OBJT |GLOB |8   |_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

    ...........

    copy relocation sizes differ: _size_gets_smaller

       (file prog size=40; file ./libfoo.so.1 size=10);

       ./libfoo.so.1 size used; possible insufficient data copied

    copy relocation sizes differ: _size_gets_larger

       (file prog size=40; file ./libfoo.so.1 size=80);
```

```
./prog size used; possible data truncation
```

ldd(1) 显示此动态可执行文件将复制此共享库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。请参见第 137 页中的 "与位置无关的代码"。

使用 -B symbolic

使用链接编辑器的 -B symbolic 选项，可以将符号引用绑定到共享库中的相应全局定义。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。请参见第 57 页中的 "缩减符号范围"。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。请参见第 144 页中的 "复制重定位"。

注 – 以符号形式绑定的共享库由 .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已在链接编辑阶段创建。

配置共享库

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库生成配置信息。运行时链接程序负责将共享库绑定到应用程序，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，请参见第 84 页中的 "执行重定位的时间"。

LD_PROFILE 环境变量将指定配置文件的共享库名称。可以使用此环境变量分析单个共享库。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库的方式。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。此设置会使用应用程序的 libc 用法累积分析信息：

```
# 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，以便使用 `LD_PROFILE_OUTPUT` 环境变量存储配置数据。

此配置数据文件用于存储 `profil`(2) 数据，并调用与使用指定共享库相关的计数信息。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。其相关共享库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) 的顶部生成。

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库。通常，使用一个或两个应用程序执行配置分析。但是，共享库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库的方式，可以了解应在何处投入更多精力以改善共享库的整体性能。

以下示例给出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库的地址范围之外进行引用。因此，在上一示例中，从可执行文件，或者从其他共享库（正在进行配置分析时绑定到 `libc`）对 `libc` 中的 open(2) 函数进行了 1634 次调用。

注 – 共享库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。可以使用 fork(2) 删除此限制。

第 5 章

应用程序二进制接口与版本控制

由链接编辑器处理的 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的更改的技术。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。

本章介绍如何定义目标文件的 ABI，并对此接口的更改影响向下兼容性的方式进行分类。此外，还介绍了用于将接口与实现的更改引入目标文件新发行版的模型。

本章重点介绍动态可执行文件与共享库的运行时接口。对说明和管理这些动态库内更改的方法只作一般性介绍。附录 B 中提供了应用于共享库的一组通用命名约定和版本控制方案。

动态库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。

由任何动态库实现可用的全局符号都代表目标文件的公共接口。通常，在链接编辑结束时，目标文件中剩余的全局符号数多于希望公开的符号数。这些全局符号是从用于创建该目标文件的可重定位目标文件之间所需的关系中产生的。它们代表目标文件自身的专用接口。

定义目标文件的二进制接口之前，应首先确定要创建的目标文件中那些希望使其公开可用的全局符号。可以使用链接编辑器的 `-M` 选项以及作为最终链接编辑一部分的关联 `mapfile` 来建立这些公共符号。第 57 页中的 “缩减符号范围” 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，以便对其进行适当管理。

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。这些更新中新增了接口，所有先前可用的接口仍保持不变。
- **不兼容**更新。这些更新更改了现有的接口，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 5–1 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
向非 varargs(3EXT) 函数中添加参数	不兼容
从函数中删除参数	不兼容
函数数据项或外部定义数据项的大小或内容更改	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

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，因此实际上这种情况非常少见。

可以通过维护要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。以下各节中更详细地介绍了这两种技术。

内部版本控制

一个动态库可以具有一个或多个与之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。符号名称则只能与**一个**版本定义关联。但是，一个版本定义可以继承其他版本定义的符号。这样，便存在一个结构，用于定义正在创建的目标文件内部的一个或多个独立或者相关的版本定义。对目标文件进行新的更改后，可以添加新版本定义来反映这些更改。

提供共享库内的版本定义有两种结果：

- 根据版本化共享库生成的动态库可以记录它们与所绑定的版本定义之间的依赖性。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。
- 动态库可以在其链接编辑过程中选择要绑定的版本定义。通过此机制，开发者可以针对接口或功能控制其与共享库的依赖性，以提供最大的灵活性。

创建版本定义

版本定义通常由符号名称与唯一版本名称关联组成。这些关联在 `mapfile` 内建立，并且在使用链接编辑器的 `-M` 选项最终链接编辑目标文件时应用它们。第 57 页中的 “缩减符号范围”一节中介绍了此技术。

版本定义是在将版本名称指定为 `mapfile` 指令的一部分时定义的。在以下示例中，将两个源文件与 `mapfile` 指令组合在一起，以生成具有已定义公共接口的目标文件：

```
$ cat foo.c

extern  const char * _foo1;



void foo1()

{

        (void) printf(_foo1);

}



$ cat data.c

const char * _foo1 = "string used by foo1()\n";



$ cat mapfile

SUNW_1.1 {                  # Release X

        global:

                foo1;

        local:
```

```
                *;

};

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o

$ nm -x libfoo.so.1 | grep "foo.$"

[33]    |0x0001058c|0x00000004|OBJT |LOCL |0x0  |17   |_foo1

[35]    |0x00000454|0x00000034|FUNC |GLOB |0x0  |9    |foo1
```

符号 foo1 是定义用来提供共享库公共接口的唯一全局符号。特殊的自动缩减指令 "*" 可使所有其他全局符号缩减，以在要生成的目标文件中具有本地绑定。第 49 页中的 “定义其他符号” 中介绍了此指令。关联的版本名称 SUNW_1.1 将生成版本定义。因此，共享库的公共接口包含与全局符号 foo1 关联的内部版本定义 SUNW_1.1。

每次使用版本定义或自动缩减指令生成目标文件时，还会创建基本版本定义。此基本版本是使用文件本身的名称定义的，用于将链接编辑器生成的所有保留符号关联在一起。有关这些保留符号的列表，请参见第 63 页中的 “生成输出文件” 。

可以使用带有 -d 选项的 pvs(1) 来显示目标文件内包含的版本定义：

```
$ pvs -d libfoo.so.1

        libfoo.so.1;

        SUNW_1.1;
```

目标文件 libfoo.so.1 具有名为 SUNW_1.1 的内部版本定义以及基本版本定义 libfoo.so.1。

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。

从此初始版本定义开始，可以通过添加新接口和已更新功能来演变目标文件。例如，可以通过更新源文件 foo.c 和 data.c，将新函数 foo2 及其支持数据结构添加到目标文件中：

```
$ cat foo.c

extern  const char * _foo1;

extern  const char * _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";
```

可以通过创建新版本定义 SUNW_1.2 来定义表示符号 foo2 的新接口。此外，还可以将此新接口定义为继承原始版本定义 SUNW_1.1。

新接口的创建非常重要，因为它标识目标文件的演变，并且使用户可以检验和选择要绑定到的接口。第 163 页中的 “绑定到版本定义” 和第 168 页中的 “指定版本绑定”中更详细地介绍了这些概念。

以下示例说明了创建这两个接口的 mapfile 指令。

```
$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
```

```
};

SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ nm -x libfoo.so.1 | grep "foo.$"
[33]    |0x00010644|0x00000004|OBJT |LOCL |0x0  |17   |_foo1
[34]    |0x00010648|0x00000004|OBJT |LOCL |0x0  |17   |_foo2
[36]    |0x000004bc|0x00000034|FUNC |GLOB |0x0  |9    |foo1
[37]    |0x000004f0|0x00000034|FUNC |GLOB |0x0  |9    |foo2
```

符号 foo1 和 foo2 都定义为共享库公共接口的一部分。但是，其中每个符号都指定给不同的版本定义；foo1 指定给 SUNW_1.1，foo2 指定给 SUNW_1.2。

可以使用同时带有 -d、-v 和 -s 选项的 pvs(1) 来显示这些版本定义、版本继承和符号关联。

```
$ 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 依赖于版本定义 SUNW_1.1。

不同版本定义之间的继承是一项很有用的技术，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。第 163 页中的 “绑定到版本定义”一节中更加详细地介绍了版本继承。

任何内部版本定义都会创建一个关联的**版本定义符号**。如上述 pvs(1) 示例中所示，使用 -v 选项时将显示这些符号。

创建弱版本定义 (weak version definition)

可以通过创建**弱**版本定义来定义目标文件的不需要引入新接口定义的内部更改。此类更改的示例有错误修复或性能改善。

此类版本定义为空，因为它没有与之关联的全局接口符号。

例如，假设对上述示例中使用的数据文件 data.c 进行更新，以提供更详细的字符串定义：

```
$ cat data.c
const char * _foo1 = "string used by function foo1()\n";
const char * _foo2 = "string used by function foo2()\n";
```

可以引入弱版本定义 (weak version definition) 来标识此更改：

```
$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
```

```
        local:
                *;
};

SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;

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};
```

空版本定义通过弱标志指示出来。通过这些弱版本定义 (weak version definition)，应用程序可以通过绑定到与此功能关联的版本定义来检验是否存在特定的实现。第 163 页中的 “绑定到版本定义”一节更详细地说明了如何使用这些定义。

定义不相关接口

上述示例说明了添加到目标文件中的新版本定义如何继承任何现有的版本定义。还可以创建唯一且独立的版本定义。以下示例将两个新文件 `bar1.c` 和 `bar2.c` 添加到目标文件 `libfoo.so.1` 中。这两个文件分别提供新符号 `bar1` 和 `bar2`：

```
$ cat bar1.c
extern  void foo1();
```

```
void bar1()
{
        foo1();
}
$ cat bar2.c
extern  void foo2();

void bar2()
{
        foo2();
}
```

这两个符号旨在定义两个新的公共接口。这些新接口互不相关。但是，每个接口都依赖于原始的 `SUNW_1.2` 接口。

以下 `mapfile` 定义将创建这种所需的关联：

```
$ cat mapfile
SUNW_1.1 {                   # Release X
        global:
                foo1;
        local:
                *;
};

SUNW_1.2 {                   # Release X+1
        global:
```

```
                foo2;
} SUNW_1.1;

SUNW_1.2.1 { } SUNW_1.2;     # Release X+2

SUNW_1.3a {                  # Release X+3
        global:
                bar1;
} SUNW_1.2;

SUNW_1.3b {                  # Release X+3
        global:
                bar2;
} SUNW_1.2;
```

同样，可以使用 pvs(1) 检查使用此 mapfile 时在 libfoo.so.1 中创建的版本定义及其依赖项：

```
$ 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};
```

以下各节介绍如何使用这些版本定义记录来检验运行时绑定要求以及在创建目标文件过程中控制其绑定。

绑定到版本定义

根据其他共享库生成动态可执行文件或共享库时，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，请参见第 31 页中的 “共享库处理”和第 120 页中的 “记录共享库名称” 。如果这些共享库依赖项还包含版本定义，则会在正在生成的目标文件中记录关联的版本依赖项。

以下示例采用上一节中的数据文件并生成适用于编译时环境的共享库。在随后的绑定示例中将使用此共享库 `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
```

实际上，此共享库提供六个公共接口。其中 SUNW_1.1、SUNW_1.2、SUNW_1.3a 以及 SUNW_1.3b 这四个接口定义导出的符号名称。还有一个接口 SUNW_1.2.1 介绍共享库的内部实现更改，另一个接口 libfoo.so.1 定义一些保留标号。使用此共享库作为依赖项而创建的动态库将记录其绑定到的接口的版本名称。

以下示例将创建引用符号 foo1 和 foo2 的应用程序。可以使用带有 -r 选项的 pvs(1) 来检查此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。

由于在 libfoo.so.1 内定义的版本定义 SUNW_1.1 由版本定义 SUNW_1.2 继承，因此还需要记录 SUNW_1.2 的版本依赖项。通过对版本定义依赖项进行这种标准化，可以减少目标文件内维护的版本信息量，并减少运行时需要进行的处理。

由于应用程序 prog 是根据包含弱版本定义 (weak version definition) SUNW_1.2.1 的共享库的实现而生成的，因此还会记录此依赖项。尽管此版本定义被定义为继承版本定义 SUNW_1.2，但此版本的弱性质不会包括通过 SUNW_1.1 进行的标准化，并且会生成单独的依赖项记录。

如果存在多个相互继承的弱版本定义 (weak version definition)，则会按照标准化非弱版本定义 (non-weak version definition) 的方式来标准化这些定义。

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。

记录了这些版本定义依赖项之后，运行时链接程序将验证执行应用程序时绑定到的目标文件中是否存在所需的版本定义。可以使用带有 -v 选项的 ldd(1) 来显示此验证。例如，通过对应用程序 prog 运行 ldd(1)，便会显示在共享库 libfoo.so.1 中正确地找到了版本定义依赖项：

```
$ 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 – 带有 -v 选项的 ldd(1) 意味着**详细**输出，将生成所有依赖项以及所有版本控制需求的递归列表。

如果找不到非弱版本定义 (non-weak version definition) 依赖项，则应用程序初始化过程中将发生致命错误。所有找不到的弱版本定义 (weak version definition) 依赖项将被忽略而无任何提示。例如，如果应用程序 prog 在 libfoo.so.1 只包含版本定义 SUNW_1.1 的环境中运行，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)
```

在应用程序 prog 未记录任何版本定义依赖项的情况下，如果必需的接口符号 foo2 不存在，则可能表明本身在执行应用程序期间的某个时刻出现致命的重定位错误。此重定位错误可能会在进程初始化时或进程执行过程中发生，也可能根本不会发生（如果应用程序的执行路径未调用函数 foo2）。请参见第 85 页中的“重定位错误”。

记录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。

如果应用程序 prog 在 libfoo.so.1 只包含版本定义 SUNW_1.1 和 SUNW_1.2 的环境中运行，则会满足所有非弱版本定义 (non-weak version definition) 需求。缺少弱版本定义 (weak version definition) SUNW_1.2.1 被认为不具有致命性，因此不生成任何运行时错误状态。但是，可以使用 ldd(1) 来显示所有找不到的版本定义：

```
$ 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()

$ ldd prog

        libfoo.so.1 =>   ./libfoo.so.1

        libfoo.so.1 (SUNW_1.2.1) =>          (version not found)

        ...........
```

注 – 如果目标文件需要给定依赖项中的版本定义，并且在运行时找到此依赖项的实现，但此依赖项不包含版本定义信息，则会忽略此依赖项的版本验证而无任何提示。此策略提供从非版本化共享库转换到版本化共享库时的向下兼容性级别。但是，仍可使用 `ldd`(1) 来显示任何版本需求差异。可以使用环境变量 `LD_NOVERSION` 来抑制所有运行时版本验证。

检验附加目标文件中的版本

版本定义符号还提供了一种机制，可检验通过 `dlopen`(3C) 获取的目标文件的版本需求。使用此函数添加到进程地址空间的任何目标文件都不会使运行时链接程序执行自动版本依赖项验证。这样，此函数的调用程序将负责检验是否满足所有版本控制需求。

可以通过使用 `dlsym`(3C) 查找关联的版本定义符号来检验是否存在所需版本定义。在以下示例中，使用 `dlopen`(3C) 将共享库 `libfoo.so.1` 添加到进程中，并检验接口 `SUNW_1.2` 是否可用。

```
#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());

                exit (1);

        }
```

```
if (dlsym(handle, vers) == NULL) {
        (void) printf("fatal: %s: version ‘%s’ not found\n",
            file, vers);
        exit (1);
}
....
```

指定版本绑定

根据包含版本定义的共享库创建动态库时，可以指示链接编辑器将绑定仅限于特定版本定义。实际上，利用链接编辑器可以控制目标文件到特定接口的绑定。

可以使用**文件控制指令**来控制目标文件的绑定需求。此指令通过链接编辑器的 `-M` 选项以及关联的 `mapfile` 提供。可以使用以下文件控制指令语法：

name - *version* [*version* ...] [$ADDVERS=*version*];

- *name*－表示共享库依赖项的名称。此名称应该与链接编辑器所使用的共享库的编译环境名称相匹配。请参见第 32 页中的 “库命名约定”。
- *version*－表示应该可用于绑定的共享库内的版本定义名称。可以指定多个版本定义。
- `$ADDVERS`－允许记录其他版本定义。

此绑定控制在下面的情况下很有用：

- 当共享库定义独立的、唯一的版本时。定义不同的标准接口时可以进行此版本控制。可以通过绑定控制生成目标文件，以确保该目标文件只绑定到特定接口。
- 已经在多个软件发行版中对共享库进行版本化时。可以通过绑定控制生成目标文件，从而将该目标文件限制为绑定到上一个软件发行版中提供的接口。这样，使用共享库的最新发行版生成的目标文件仍可以与共享库依赖项的旧发行版一起运行。

以下示例说明了版本控制机制的用法。此示例使用包含以下版本接口定义的共享库 `libfoo.so.1`：

```
$ 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 表示 libfoo.so.1 内的接口，这些接口分别在软件 Release X 和 Release X+1 中提供。

可以使用以下版本控制 mapfile 指令来生成应用程序，使其只绑定到 Release X 中提供的接口：

```
$ cat mapfile
libfoo.so - SUNW_1.1;
```

例如，假设您开发一个应用程序 prog，并且要确保此应用程序可以在 Release X 中运行。这样此应用程序只能使用此发行版中提供的接口。如果应用程序错误地引用了符号 bar，便会与所需接口不兼容。链接编辑器会将此情形报告为未定义的符号错误：

```
$ 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. No output written to prog
```

为了与 SUNW_1.1 接口兼容，必须删除对 bar 的引用。可以对应用程序重新进行处理以删除对 bar 的需求，也可以在创建应用程序时添加 bar 的实现。

注 – 缺省情况下，还会根据任意文件控制指令来检验链接编辑过程中遇到的共享库依赖项。使用环境变量 LD_NOVERSION 可抑制任何共享库依赖项的版本验证。

到其他版本定义的绑定

要使记录的版本依赖项多于从目标文件的正常符号绑定中生成的版本依赖项，请使用 $ADDVERS 文件控制指令。本节介绍此附加绑定可能有用的情况。

在一个 libfoo.so.1 示例中，假设在 Release X+2, 中，版本定义 SUNW_1.1 分为两个标准发行版：STAND_A 和 STAND_B。要保持兼容性，必须维护 SUNW_1.1 版本定义。在本示例中，此版本定义表示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 的唯一需求是接口符号 foo1，则此应用程序将仅依赖于版本定义 STAND_A。这将阻止在 libfoo.so.1 小于 Release X+2 的系统上运行 prog。尽管早期的发行版具有接口 foo1，但没有版本定义 STAND_A。

生成应用程序 prog 时，可以通过创建对 SUNW_1.1 的依赖性来使其要求与早期发行版一致：

```
$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=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);
```

此显式依赖性足以涵盖实际的依赖性要求。此依赖性可满足与旧发行版的兼容性。

第 159 页中的 "创建弱版本定义 (weak version definition)"介绍了如何使用弱版本定义 (weak version definition) 标记内部实现更改。这些版本定义非常适用于指示针对目标文件所做的错误修复以及性能改善。如果需要弱版本，可以生成对此版本定义的显式依赖性。当错误修复或性能改善对于目标文件的正常工作至关重要时，创建此类依赖性非常重要。

在上一个 libfoo.so.1 示例中，假设在软件 Release X+3 中以弱版本定义 (weak version definition) SUNW_1.2.1 引入了错误修复：

```
$ 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，如果根据此共享库生成应用程序，则生成的应用程序将记录与版本定义 SUNW_1.2.1 的弱依赖性。此依赖性仅用于提供信息。如果运行时使用的 libfoo.so.1 中不存在此版本定义，则此依赖性不会导致终止应用程序。

文件控制指令 $ADDVERS 可用于生成版本定义的显式依赖性。如果此定义为弱定义，则此显式引用还会导致版本定义提升为强依赖性。

可以使用以下文件控制指令生成应用程序 prog，以强制满足 SUNW_1.2.1 接口在运行时可用的需求：

```
$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=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);
```

生成 prog 时创建了与接口 STAND_A 的显式依赖性。由于版本定义 SUNW_1.2.1 提升为强版本，因此它也会通过依赖性 STAND_A 进行标准化。在运行时，如果找不到版本定义 SUNW_1.2.1，则会产生致命错误。

注 – 如果处理少量的依赖项，可以使用链接编辑器的 -u 选项显式绑定到某个版本定义。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。如果是处理多个依赖项（包含多个类似命名的版本定义）时，则此方法可能不足以创建显式绑定。

版本稳定性

只有当单个版本定义在目标文件的生命周期内保持不变时，绑定到该目标文件内版本的各种模型才会保持不变。

创建目标文件的版本定义并将其公开后，此定义就必须在该目标文件的后续发行版中保持未更改状态。版本名称以及关联的符号必须保持不变。因此，不支持对版本定义内定义的符号名称进行通配符扩展。在目标文件演变过程中，与通配符匹配的符号数可能会有所不同。

可重定位目标文件

可以在动态库内记录并使用版本信息。可重定位目标文件可用类似的方式维护版本控制信息。但是，在如何使用此信息方面存在一些细微差异。

记录提供给可重定位目标文件链接编辑的任何版本定义时采用的格式与生成动态可执行文件或共享库时采用的格式相同。但是，缺省情况下，不会对正在创建的目标文件执行符号缩减。相反，当最终使用可重定位目标文件作为生成动态库的输入时，将使用版本记录来确定要应用的符号缩减。

此外，在可重定位目标文件中找到的所有版本定义都会传播到该动态库。有关可重定位目标文件中版本处理的示例，请参见第 57 页中的 “缩减符号范围” 。

外部版本控制

对共享库的运行时引用应该始终引用文件的版本文件名。版本文件名通常表示为带有版本号后缀的文件名。由于共享库的接口以不兼容的方式变化，这种变化会造成无法继续使用旧应用程序，因此，应该使用新的版本化文件名来分发新的共享库。此外，仍必须使用原来的版本化文件名分发旧的共享库，以提供旧应用程序所需的接口。

针对多个软件发行版生成应用程序时，应当在运行时环境中通过单独的版本化文件名来提供共享库。这样可以保证生成应用程序所依据的接口可用，以使应用程序在其执行过程中可以绑定到此接口。

下面一节介绍如何协调编译环境和运行时环境之间的接口绑定。

协调版本化文件名

在链接编辑过程中，输入共享库的最常见方法是使用 `-l` 选项。此选项使用链接编辑器的库搜索机制来查找带有 `lib` 前缀以及 `.so` 后缀的共享库。

但是，在运行时，所有共享库依赖项都应该以其**版本化**名称形式存在。可以创建两个文件名之间的文件系统链接，而不是维护遵循这些命名约定的两个不同的共享库。

要使运行时共享库 `libfoo.so.1` 可用于编译环境，请提供从编译文件名到运行时文件名的符号链接。例如：

```
$ 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
```

符号链接和硬链接都可使用。但是，符号链接在文档和诊断辅助方面更有用。

已经针对运行时环境生成共享库 `libfoo.so.1`。通过生成符号链接 `libfoo.so`，可以在编译环境中使用此文件。例如：

```
$ cc -o prog main.o -L. -lfoo
```

链接编辑器处理具有共享库 `libfoo.so.1`（通过符号链接 `libfoo.so` 即可找到）所描述的接口的可重定位目标文件 `main.o`。

针对多个软件发行版，可以分发具有已更改接口的此共享库的新版本。可以将编译环境构建为通过更改符号链接即可使用适用的接口。例如：

```
$ 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
```

此共享库提供三个主要的版本。其中两个共享库 `libfoo.so.1` 和 `libfoo.so.2` 为现有应用程序提供依赖项。`libfoo.so.3` 则为创建和运行新应用程序提供最新主发行版。

使用此符号链接机制本身并不足以保证在编译环境中使用了正确的共享库绑定，就能满足运行时环境对此绑定的需求。如示例目前所示，链接编辑器将在动态可执行文件 prog 中记录其已处理的共享库的文件名。在这种情况下，此文件名即是编译环境文件名。

```
$ dump -Lv prog


prog:

 **** DYNAMIC SECTION INFORMATION ****

.dynamic:

[INDEX] Tag      Value

[1]     NEEDED   libfoo.so

.........
```

执行应用程序 prog 后，运行时链接程序将搜索依赖项 libfoo.so。prog 将绑定到此符号链接所指向的文件。

要提供记录为依赖项的正确运行时名称，应该生成共享库 libfoo.so.1（通过 soname 定义生成）。此定义标识共享库的运行时名称。链接此共享库的任何目标文件将此名称用作依赖项的名称。在共享库本身的链接编辑过程中，可以使用 -h 选项提供此定义。 例如：

```
$ 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

$ dump -Lv prog


prog:

 **** DYNAMIC SECTION INFORMATION ****

.dynamic:

[INDEX] Tag      Value

[1]     NEEDED   libfoo.so.1
```

....

此符号链接和 soname 机制已经在编译和运行时环境的共享库命名约定之间建立了很强的协调。将在生成的输出文件中准确记录链接编辑过程中处理的接口。此记录可确保在运行时提供目标接口。

注意 – 创建新的外部版本化共享库是一项主要的更改。请务必了解使用此共享库的所有进程的全部依赖项。

例如，应用程序可能依赖于 `libfoo.so.1` 以及从外部传送的目标文件 `libISV.so.1`。后者也可能依赖于 `libfoo.so.1`。如果重新设计应用程序，使其使用 `libfoo.so.2` 中的新接口，而不对外部目标文件 `libISV.so.1` 的使用做任何更改，则 `libfoo.so` 的两个主要版本都会引入正在运行的进程。由于更改 `libfoo.so` 版本的唯一原因是标记不兼容的更改，因此在一个进程中具有该目标文件的两个版本可能会导致错误的符号绑定并由此导致不需要的交互。

第 6 章

支持接口

链接编辑器提供了许多支持接口，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* – 第 179 页中的 "链接编辑器支持接口"
- *rtld-audit* – 第 186 页中的 "运行时链接程序审计接口"
- *rtld-debugger* – 第 196 页中的 "运行时链接程序调试器接口"

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本身（使用此接口处理可重定位目标文件内的调试信息）以及 `make`(1S) 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:libldstab.so.1 cc ...
```

`-S` 选项用于指定单个支持库。可以指定多个 `-S` 选项：

```
$ LD_OPTIONS="-S./support.so.1 -Slibldstab.so.1" cc ...
```

支持库是共享库。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，请参见第 180 页中的 “32 位环境和 64 位环境” 。

注 – 缺省情况下，链接编辑器使用 Solaris 支持库 libldstab.so.1 来处理和压缩输入可重定位目标文件内提供的编译器生成的调试信息。如果对使用 -S 选项指定的任何支持库调用链接编辑器，则抑制此缺省处理。如果除支持库服务之外还要求 libldstab.so.1 的缺省处理，请将 libldstab.so.1 显式添加到提供给链接编辑器的支持库列表中。

---

32 位环境和 64 位环境

如第 24 页中的 “32 位环境和 64 位环境”中所述，64 位链接编辑器 ld(1) 可以生成 32 位目标文件，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 位类的单个共享库 libldstab.so.1 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零，或者返回的版本值高于链接编辑器所支持的 ld-support 接口的版本，则无法使用支持库。

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，如 sys/elf.h 中所定义。*caller* 是调用接口的应用程序，通常为 /usr/ccs/bin/ld。

ld_file()
执行任何文件数据处理之前，会针对每个输入文件调用此函数。

```
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，如 libelf.h 中所定义。*flags* 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库的依赖项获取。

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* 是指向文件的 ELF 描述符的指针。

`ld_input_section()`

此函数会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* 是指向关联节标题的指针。*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。

节标题的修改是通过重新分配节标题并为新标题重新指定 **shdr* 来完成的。链接编辑器使用从 `ld_input_section()` 返回时 **shdr* 所指向的节标题信息来处理节。

通过重新分配数据并重新指定 `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。

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()`

此函数针对传播给输出文件的输入文件的每一节调用，并且在执行任何节数据处理之前即会调用此函数。

```
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* 是指向关联节标题的指针。*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定 `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 – 使用链接编辑器的 `-s` 选项删除的各节，或者由于 `SHT_SUNW_COMDAT` 处理或`SHF_EXCLUDE` 标识而废弃的各节不会向 `ld_section()` 进行报告。请参见第 238 页中的 “COMDAT 节”和表 7–8。

`ld_input_done()`
此函数在完成输入文件处理之后但在对输出文件进行布局之前调用。

```
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`，如 `stdlib.h` 中所定义。

支持接口示例

以下示例创建了一个支持库，其中列显了在 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。

```
$ 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 -Slibldstab.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

进程可以使用 rtld-audit 接口访问与其自身相关的运行时链接信息。第 148 页中的 “配置共享库”中介绍对共享库的运行时配置即是使用此机制的一个示例。

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。可以通过审计库修改函数参数及其返回值。请参见第 299 页中的 “过程链接表（特定于处理器）”。

通过预装入专用的共享库可以实现其中的部分功能。但是，预装入的目标文件与进程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的共享库的实现或者使实现变得更为复杂。 rtld-audit 接口会为用户提供唯一的名称空间，用于在其中执行其审计库。此名称空间可确保在进程内进行正常绑定时审计库不会侵入。

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，用于对此进程进行说明。/usr/include/sys/link.h 中定义的链接映射结构说明了进程内的每个目标文件。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进程符号解析的**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，从而可防止以任何方式将应用程序直接绑定到运行时链接程序内的服务。应用程序只能通过过滤器 libc.so.1 或libdl.so.1 调用运行时链接程序的公共服务。

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，在应用程序的的符号绑定要求中，不涉及审计库。通过 dlmopen(3C) 可检查应用程序链接映射列表。将 RTLD_NOLOAD 标志用于 dlmopen(3C) 时，审计库可以在不装入目标文件的情况下查询此目标文件是否存在。

/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 */
```

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。

创建审计库

审计库的生成方式与其他任何共享库的生成方式相同。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。

如果审计库调用 printf(3C)，则审计库必须定义与 libc 之间的依赖性。请参见第 47 页中的 “生成共享库输出文件”。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范围。

- **全局**审计，通过使用环境变量 LD_AUDIT 来启用。 通过此方法可用的审计库附带有与进程所使用的所有动态库相关的信息。
- **局部**审计，通过生成目标文件时在目标文件内记录的动态项来启用。通过此方法可用的审计库附带有与标识用于审计的那些动态库相关的信息。

任一调用方法都包含一个字符串，其中包含通过 dlopen(3C) 装入的以冒号分隔的共享库列表。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。

通过 rtld-audit 接口可以提供多个审计库。希望以此方式使用的审计库不应更改通常由运行时链接程序返回的绑定。更改这些绑定会在后面的审计库中产生意外结果。

安全应用程序只能从受信任的目录中获取审计库。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受信任的目录仅有 /lib/secure 和 /usr/lib/secure。对于 64 位目标文件，受信任的目录是 /lib/secure/64 和 /usr/lib/secure/64。

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
$ dump -Lv libfoo.so.1 | fgrep AUDIT
[3]    AUDIT      audit.so.1
```

在运行时，如果存在此审计标识符，则会装入审计库并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
$ dump -Lv main | fgrep AUDIT
[5]    DEPAUDIT   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
$ dump -Lv main | fgrep AUDIT
[5]    DEPAUDIT   audit.so.1
```

注 – 通过将环境变量 `LD_NOAUDIT` 设置为非空值，可以在运行时禁用审计。

审计接口函数

rtld-audit 接口提供了以下函数。这些函数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()`

此函数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 *version* 最高的 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。

`la_activity()`
此函数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。

`la_objsearch()`
此函数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_objopen()`
此函数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*。审计库可以修改此标识符，以便更好地标识其他 rtld-audit 接口例程的目标文件。

`la_objopen()` 函数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。

la_objfilter()
此函数在过滤器装入新的 filtee 时调用。请参见第 125 页中的“作为过滤器的共享库”。

```
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 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()
为应用程序装入所有目标文件之后但在将控制权转交给应用程序之前，会调用一次此函数。

```
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。

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，会调用此函数。

```
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() 函数的标识符相同，此函数会返回 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() 函数存在，则对于过程链接表的各项，这些函数在 la_symbind() 之后调用。每次引用符号时都会调用这些函数。 另请参见第 196 页中的 “审计接口限制” 。

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() 函数。
- LA_SYMB_NOPLTEXIT－**请勿**针对此符号调用 la_pltexit() 函数。

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()

这些函数是系统特定的。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，会调用这些函数。

```
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，会调用此函数。此函数在调用方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。请参见第 196 页中的 “审计接口限制” 。

`la_objclose()`
此函数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。

审计接口示例

以下简单示例创建了一个审计库，其中列显了动态可执行文件 `date(1)` 装入的每个共享库依赖项的名称。

```
$ 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 2000
```

审计接口演示

/usr/demo/link_audit 下的 SUNWosdem 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库之间的过程调用。

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 SUNWtoo 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

使用 `la_pltexit()` 系列存在一些限制。这些限制是由于需要在调用方和被调用方之间插入额外栈帧，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。这些调试程序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，并使用该库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库。
- 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。这些例程将在第 207 页中的 “调试器导入接口” 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。

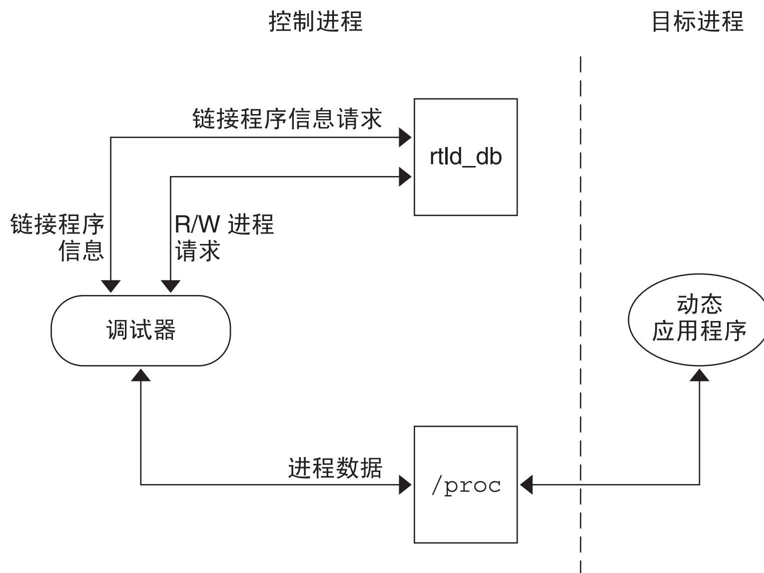


图 6–1 rtld-debugger 信息流程

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。

/usr/demo/librtld_db 下的 SUNWosdem 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
: Is an opaque structure created by thertld-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。请参见第 200 页中的 “扫描可装入目标文件”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。请参见第 205 页中的 “跳过过程链接表”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）可以返回以下错误状态：

```
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;

} 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
: 指向包含动态库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
: 动态库的基本地址。

`rl_data_base`
动态库数据段的基本地址。

`rl_lmident`
链接映射标识符（请参见第 187 页中的 “建立名称空间”）。

`rl_refnameaddr`
如果动态库是标准过滤器，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（包括其节标题）的结尾。

`rl_padstart`
动态库之前填充的基本地址（请参阅第 207 页中的 “动态库填充”）。

`rl_padend`
动态库之后填充的基本地址（请参阅第 207 页中的 “动态库填充”）。

`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 7–32）。

`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()`
会对当前在目标进程中装入的所有动态库重复执行此函数。

```
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范围内发生的特定事件。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库，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库。

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
: 正在装入动态库，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
: 正在删除动态库，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 (step into) 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。

动态库填充

运行时链接程序的缺省行为取决于要装入动态库的操作系统（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）。使用具有 PROT_NONE 权限和 MAP_NORESERVE 标志的 mmap(2) 可将填充保留为内存映射。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 mmap(2) 从具有 MAP_NORESERVE 标志的 /dev/zero 中获取的预留空间。

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* 的名为 *obj* 的目标文件内搜索名为 *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* 的名为 *obj* 的目标文件内搜索名为 *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。

第 7 章

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)。其中存在三种重要类型的目标文件。

- **可重定位目标**文件包含代码部分和数据部分。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、共享库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库文件以创建进程映像。
- **共享库**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库文件一起处理，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库文件合并，以创建进程映像。

本章的第一节，第 211 页中的 “文件格式”，重点介绍目标文件的格式以及格式如何与创建程序相关。第二节，第 274 页中的 “动态链接”，重点介绍格式如何与装入程序相关。

程序可以使用 ELF 访问库 libelf 提供的函数来处理目标文件。有关 libelf 内容的说明，请参阅 elf(3ELF)。/usr/demo/ELF 目录下的 SUNWosdem 软件包中提供了使用 libelf 的源代码样例。

文件格式

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

图 7–1 目标文件格式

ELF 头位于目标文件的起始位置，其中包含用于说明文件结构的**指南**。

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

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

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

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

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

数据表示形式

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

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

表 7–1 ELF 32 位数据类型

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

表 7–2 ELF 64 位数据类型

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

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

注 – 为便于移植，ELF 不使用位字段。

ELF 头

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

ELF 头具有以下结构。 请参见 sys/elf.h。

```
#define EI_NIDENT       16

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

typedef struct {
```

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

`e_ident`
将文件标记为目标文件的初始字节。这些字节可提供与计算机无关的数据，用于解码和解释文件的内容。第 218 页中的 “ELF 标识” 中提供了完整说明。

`e_type`
标识目标文件类型，如下表中所列。

名称	值	含义
ET_NONE	0	无文件类型
ET_REL	1	可重定位文件
ET_EXEC	2	可执行文件
ET_DYN	3	共享库文件

名称	值	含义
ET_CORE	4	核心转储文件
ET_LOPROC	0xff00	特定于处理器
ET_HIPROC	0xffff	特定于处理器

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

e_machine
指定独立文件所需的体系结构。下表中列出了相关体系结构。

名称	值	含义
EM_NONE	0	无计算机
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_SPARC32PLUS	18	Sun SPARC 32+
EM_SPARCV9	43	SPARC V9
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 定义的标志会使用前缀 EF_。EM_XYZ 计算机的名为 WIDGET 的标志可称为 EF_XYZ_WIDGET。

e_version
标识目标文件版本，如下表中所列。

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

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

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

e_phoff
程序头表的文件偏移（以字节为单位）。如果文件没有程序头表，则此成员值为零。

e_shoff
节头表的文件偏移（以字节为单位）。如果文件没有节头表，则此成员值为零。

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

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

e_ehsize
ELF 头的大小（以字节为单位）。

e_phentsize
文件的程序头表中某一项的大小（以字节为单位）。所有项的大小都相同。

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

e_shentsize
节头的大小（以字节为单位）。节头是节头表中的一项。所有项的大小都相同。

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

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

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

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

ELF 标识

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

ELF 头和目标文件的初始字节对应于 e_ident 成员。

表 7–3 ELF 标识索引

名称	值	目的
EI_MAG0	0	文件标识
EI_MAG1	1	文件标识
EI_MAG2	2	文件标识
EI_MAG3	3	文件标识
EI_CLASS	4	文件类
EI_DATA	5	数据编码
EI_VERSION	6	文件版本
EI_OSABI	7	操作系统/ABI 标识
EI_ABIVERSION	8	ABI 版本
EI_PAD	9	填充字节的开头
EI_NIDENT	16	e_ident[] 的大小

这些索引可访问值为以下各项的字节。

EI_MAG0 – EI_MAG3
4 字节**魔数**，用于将文件标识为 ELF 目标文件，如下表中所列。

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

EI_CLASS

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

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

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

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

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

EI_DATA

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

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

第 220 页中的 “数据编码” 一节中提供了有关这些编码的更多信息。其他值保留供将来使用。

EI_VERSION

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

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

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

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

数据编码

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

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

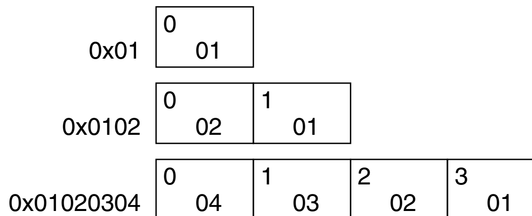


图 7–2 数据编码 ELFDATA2LSB

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

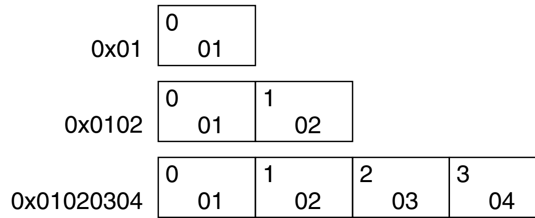


图 7–3 数据编码 ELFDATA2MSB

节

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

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

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

表 7–4 ELF 特殊节索引

名称	值
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_BEFORE	0xff00
SHN_AFTER	0xff01
SHN_AMD64_LCOMMON	0xff02
SHN_HIPROC	0xff1f
SHN_LOOS	0xff20
SHN_LOSUNW	0xff3f

表 7–4 ELF 特殊节索引 （续）

名称	值
SHN_SUNW_IGNORE	0xff3f
SHN_HISUNW	0xff3f
SHN_HIOS	0xff3f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_XINDEX	0xffff
SHN_HIRESERVE	0xffff

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

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

SHN_LORESERVE
　所保留索引的范围的下边界。

SHN_LOPROC - SHN_HIPROC
　此范围内包含的值保留用于特定于处理器的语义。

SHN_LOOS - SHN_HIOS
　此范围内包含的值保留用于特定于操作系统的语义。

SHN_LOSUNW - SHN_HISUNW
　此范围内包含的值保留用于特定于 Sun 的语义。

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

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

SHN_AMD64_LCOMMON
　特定于 x64 的通用块标签。此标签与 SHN_COMMON 类似，但用于标识较大的通用块。

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

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

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

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

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

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

节头具有以下结构。 请参见 sys/elf.h。

```
typedef struct {

        elf32_Word      sh_name;

        Elf32_Word      sh_type;

        Elf32_Word      sh_flags;

        Elf32_Addr      sh_addr;

        Elf32_Off       sh_offset;

        Elf32_Word      sh_size;

        Elf32_Word      sh_link;

        Elf32_Word      sh_info;
```

```
        Elf32_Word      sh_addralign;

        Elf32_Word      sh_entsize;

} Elf32_Shdr;


typedef struct {

        Elf64_Word      sh_name;

        Elf64_Word      sh_type;

        Elf64_Xword     sh_flags;

        Elf64_Addr      sh_addr;

        Elf64_Off       sh_offset;

        Elf64_Xword     sh_size;

        Elf64_Word      sh_link;

        Elf64_Word      sh_info;

        Elf64_Xword     sh_addralign;

        Elf64_Xword     sh_entsize;

} Elf64_Shdr;
```

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

sh_type
: 用于将节的内容和语义分类。表 7–5 中列出了节类型及其说明。

sh_flags
: 节可支持用于说明杂项属性的 1 位标志。表 7–8 中列出了标志定义。

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

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

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

`sh_link`
节头表索引链接，其解释依赖于节类型。表 7–9 说明了相应的值。

`sh_info`
额外信息，其解释依赖于节类型。表 7–9 说明了相应的值。

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

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

节头的 `sh_type` 成员用于指定节的语义，如下表中所示。

表 7–5 ELF 节类型 *sh_type*

名称	值
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_INIT_ARRAY	14

表 7–5 ELF 节类型 *sh_type* （续）

名称	值
SHT_FINI_ARRAY	15
SHT_PREINIT_ARRAY	16
SHT_GROUP	17
SHT_SYMTAB_SHNDX	18
SHT_LOOS	0x60000000
SHT_LOSUNW	0x6ffffff4
SHT_SUNW_dof	0x6ffffff4
SHT_SUNW_cap	0x6ffffff5
SHT_SUNW_SIGNATURE	0x6ffffff6
SHT_SUNW_ANNOTATE	0x6ffffff7
SHT_SUNW_DEBUGSTR	0x6ffffff8
SHT_SUNW_DEBUG	0x6ffffff9
SHT_SUNW_move	0x6ffffffa
SHT_SUNW_COMDAT	0x6ffffffb
SHT_SUNW_syminfo	0x6ffffffc
SHT_SUNW_verdef	0x6ffffffd
SHT_SUNW_verneed	0x6ffffffe
SHT_SUNW_versym	0x6fffffff
SHT_HISUNW	0x6fffffff
SHT_HIOS	0x6fffffff
SHT_LOPROC	0x70000000
SHT_SPARC_GOTDATA	0x70000000
SHT_AMD64_UNWIND	0x70000001
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

SHT_NULL
将节头标识为非活动。此节头没有关联的节。节头的其他成员具有未定义的值。

SHT_PROGBITS
标识由程序定义的信息，这些信息的格式和含义仅由程序确定。

SHT_SYMTAB、SHT_DYNSYM
标识符号表。通常，SHT_SYMTAB 节会提供用于链接编辑的符号。作为完整的符号表，该表可以包含许多对于动态链接不需要的符号。因此，目标文件还可以包含 SHT_DYNSYM 节，其中包含一组尽可能少的动态链接符号，从而可节省空间。有关详细信息，请参见第 260 页中的“符号表节”。

SHT_STRTAB、SHT_DYNSTR
标识字符串表。目标文件可以有多个字符串表节。有关详细信息，请参见第 259 页中的“字符串表节”。

SHT_RELA
标识包含显式加数的重定位项，如 32 位类的目标文件的 Elf32_Rela 类型。目标文件可以有多个重定位节。有关详细信息，请参见第 247 页中的“重定位节”。

SHT_HASH
标识符号散列表。动态链接的目标文件必须包含符号散列表。当前，目标文件只能有一个散列表，但此限制在将来可能会放宽。有关详细信息，请参见第 241 页中的“散列表节”。

SHT_DYNAMIC
标识动态链接的信息。当前，目标文件只能有一个动态节。有关详细信息，请参见第 287 页中的“动态节”。

SHT_NOTE
标识以某种方法标记文件的信息。有关详细信息，请参见第 246 页中的“注释节”。

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

SHT_REL
标识不包含显式加数的重定位项，如 32 位类的目标文件的 Elf32_Rel 类型。目标文件可以有多个重定位节。有关详细信息，请参见第 247 页中的“重定位节”。

SHT_SHLIB
标识具有未指定的语义的保留节。包含此类型的节的程序不符合 ABI。

SHT_INIT_ARRAY
标识包含指针数组的节，这些指针指向初始化函数。数组中的每个指针都视为不返回任何值的无参数过程。有关详细信息，请参见第 36 页中的“初始化和终止节”。

SHT_FINI_ARRAY
标识包含指针数组的节，这些指针指向终止函数。数组中的每个指针都视为不返回任何值的无参数过程。有关详细信息，请参见第 36 页中的“初始化和终止节”。

SHT_PREINIT_ARRAY
: 标识包含指针数组的节，这些指针指向在其他所有初始化函数之前调用的函数。数组中的每个指针都视为不返回任何值的无参数过程。 有关详细信息，请参见第 36 页中的 “初始化和终止节” 。

SHT_GROUP
: 标识节组。节组可标识一组相关的节，这些节必须作为一个单位由链接编辑器进行处理。SHT_GROUP 类型的节只能出现在可重定位目标文件中。 有关详细信息，请参见第 239 页中的 “组节” 。

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

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

SHT_LOOS－SHT_HIOS
: 此范围内包含的值保留用于特定于操作系统的语义。

SHT_LOSUNW－SHT_HISUNW
: 此范围内包含的值保留用于 Solaris 语义。

SHT_SUNW_cap
: 指定硬件和软件的功能要求。 有关详细信息，请参见第 240 页中的 “硬件和软件功能节” 。

SHT_SUNW_SIGNATURE
: 标识模块验证签名。

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

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

SHT_SUNW_move
: 标识用于处理部分初始化的符号的数据。 有关详细信息，请参见第 243 页中的 “移动节” 。

SHT_SUNW_COMDAT
: 标识允许将相同数据的多个副本减少为单个副本的节。 有关详细信息，请参见第 238 页中的 “COMDAT 节” 。

SHT_SUNW_syminfo
标识其他符号信息。有关详细信息，请参见第 267 页中的 “Syminfo 表节”。

SHT_SUNW_verdef
标识此文件定义的细分版本。有关详细信息，请参见第 269 页中的 “版本定义节”。

SHT_SUNW_verneed
标识此文件所需的细分依赖性。有关详细信息，请参见第 272 页中的 “版本依赖性节”。

SHT_SUNW_versym
标识用于说明符号与文件提供的版本定义之间关系的表。有关详细信息，请参见第 271 页中的 “版本符号节”。

SHT_LOPROC－SHT_HIPROC
此范围内包含的值保留用于特定于处理器的语义。

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

SHT_AMD64_UNWIND
标识特定于 x64 的数据，其中包含对应于栈展开的展开函数表的各项。

SHT_LOUSER
指定保留用于应用程序的索引范围的下边界。

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

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

表 7–6 ELF 节头表项：索引 0

名称	值	说明
sh_name	0	无名称
sh_type	SHT_NULL	非活动
sh_flags	0	无标志
sh_addr	0	无地址
sh_offset	0	无文件偏移
sh_size	0	无大小

表 7–6 ELF 节头表项：索引 0　　（续）

名称	值	说明
sh_link	SHN_UNDEF	无链接信息
sh_info	0	无辅助信息
sh_addralign	0	无对齐
sh_entsize	0	无项

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

表 7–7 ELF 扩展的节头表项：索引 0

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

节头的 sh_flags 成员包含用于说明节属性的 1 位标志：

表 7–8 ELF 节属性标志

名称	值
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MERGE	0x10

表 7–8 ELF 节属性标志 （续）

名称	值
SHF_STRINGS	0x20
SHF_INFO_LINK	0x40
SHF_LINK_ORDER	0x80
SHF_OS_NONCONFORMING	0x100
SHF_GROUP	0x200
SHF_TLS	0x400
SHF_MASKOS	0x0ff00000
SHF_AMD64_LARGE	0x10000000
SHF_ORDERED	0x40000000
SHF_EXCLUDE	0x80000000
SHF_MASKPROC	0xf0000000

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

SHF_WRITE
标识在进程执行过程中应可写的节。

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

SHF_EXECINSTR
标识包含可执行计算机指令的节。

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

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

SHF_INFO_LINK
此节头的 sh_info 字段包含节头表索引。

SHF_LINK_ORDER

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

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

此标志的一个典型用法是生成按地址顺序引用文本或数据节的表。

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

SHF_OS_NONCONFORMING

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

SHF_GROUP

此节是节组的一个成员，可能是唯一的成员。此节必须由 SHT_GROUP 类型的节引用。只能对可重定位目标文件中包含的节设置 SHF_GROUP 标志。 有关详细信息，请参见第 239 页中的 “组节”。

SHF_TLS

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

SHF_MASKOS

此掩码中包括的所有位都保留用于特定于操作系统的语义。

SHF_AMD64_LARGE

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

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

SHF_ORDERED

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

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

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

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

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

SHF_MASKPROC
此掩码中包括的所有位都保留用于特定于处理器的语义。

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

表 7–9 ELF sh_link 和 sh_info 解释

sh_type	sh_link	sh_info
SHT_DYNAMIC	关联的字符串表的节头索引。	0
SHT_HASH	关联的符号表的节头索引。	0
SHT_REL SHT_RELA	关联的符号表的节头索引。	应用重定位的节的节头索引。另请参见表 7–10 和第 247 页中的 “重定位节” 。
SHT_SYMTAB SHT_DYNSYM	关联的字符串表的节头索引。	比上一个局部符号 STB_LOCAL 的符号表索引大一。
SHT_GROUP	关联的符号表的节头索引。	关联的符号表中项的符号表索引。指定的符号表项的名称用于提供节组的签名。
SHT_SYMTAB_SHNDX	关联的符号表的节头索引。	0
SHT_SUNW_move	关联的符号表的节头索引。	0
SHT_SUNW_COMDAT	0	0
SHT_SUNW_syminfo	关联的符号表的节头索引。	关联的 .dynamic 节的节头索引。
SHT_SUNW_verdef	关联的字符串表的节头索引。	节中版本定义的编号。
SHT_SUNW_verneed	关联的字符串表的节头索引。	节中版本依赖性的编号。

表 7–9 ELF sh_link 和 sh_info 解释 （续）

sh_type	sh_link	sh_info
SHT_SUNW_versym	关联的符号表的节头索引。	0

特殊节

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

表 7–10 ELF 特殊节

名称	类型	属性
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	无
.data、.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dynamic	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.eh_frame_hdr	SHT_AMD64_UNWIND	SHF_ALLOC
.eh_frame	SHT_AMD64_UNWIND	SHF_ALLOC + SHF_WRITE
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.fini_array	SHT_FINI_ARRAY	SHF_ALLOC + SHF_WRITE
.got	SHT_PROGBITS	请参见第 299 页中的 “全局偏移表（特定于处理器）”
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.init_array	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
.interp	SHT_PROGBITS	请参见第 286 页中的 “程序的解释程序”
.note	SHT_NOTE	无
.lbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE
.ldata、.ldata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE

表 7–10 ELF 特殊节 （续）

名称	类型	属性
.lrodata、.lrodata1	SHT_PROGBITS	SHF_ALLOC + SHF_AMD64_LARGE
.plt	SHT_PROGBITS	请参见第 299 页中的 “过程链接表（特定于处理器）”
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
.rela	SHT_RELA	无
.rel*name*	SHT_REL	请参见第 247 页中的 “重定位节”
.rela*name*	SHT_RELA	请参见第 247 页中的 “重定位节”
.rodata、.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	无
.strtab	SHT_STRTAB	请参阅此表后面的说明。
.symtab	SHT_SYMTAB	请参见第 260 页中的 “符号表节”
.symtab_shndx	SHT_SYMTAB_SHNDX	请参见第 260 页中的 “符号表节”
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata、.tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.SUNW_bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_cap	SHT_SUNW_cap	SHF_ALLOC
.SUNW_move	SHT_SUNW_move	SHF_ALLOC
.SUNW_reloc	SHT_REL	SHF_ALLOC
	SHT_RELA	
.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，请参见第 287 页中的 “动态节”。

.dynstr
进行动态链接所需的字符串，通常是表示与符号表各项关联的名称的字符串。

.dynsym
动态链接符号表。有关详细信息，请参见第 260 页中的 “符号表节”。

.eh_frame_hdr、.eh_frame
用于展开栈的调用帧信息。

.fini
可执行指令，用于构成包含此节的可执行文件或共享库的单个终止函数。有关详细信息，请参见第 91 页中的 “初始化和终止例程”。

.fini_array
函数指针数组，用于构成包含此节的可执行文件或共享库的单个终止数组。有关详细信息，请参见第 91 页中的 “初始化和终止例程”。

.got
全局偏移表。有关详细信息，请参见第 299 页中的 “全局偏移表（特定于处理器）”。

.hash
符号散列表。有关详细信息，请参见第 241 页中的 “散列表节”。

.init
可执行指令，用于构成包含此节的可执行文件或共享库的单个初始化函数。有关详细信息，请参见第 91 页中的 “初始化和终止例程”。

.init_array
函数指针数组，用于构成包含此节的可执行文件或共享库的单个初始化数组。有关详细信息，请参见第 91 页中的 “初始化和终止例程”。

.interp
程序的解释程序的路径名。有关详细信息，请参见第 286 页中的 “程序的解释程序”。

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

.ldata 和 .ldata1
特定于 x64 的已初始化数据。此数据与 .data 类似，但用于大小超过 2 GB 的节。

.lrodata 和 .lrodata1
特定于 x64 的只读数据。此数据与 .rodata 类似，但用于大小超过 2 GB 的节。

.note
第 246 页中的 “注释节” 中说明了该格式的信息。

.plt
过程链接表。 有关详细信息，请参见第 299 页中的 “过程链接表（特定于处理器）” 。

.preinit_array
函数指针数组，用于构成包含此节的可执行文件或共享库的单个 预初始化 数组。 有关详细信息，请参见第 91 页中的 “初始化和终止例程” 。

.rela
不适用于特定节的重定位。此节的用途之一是用于寄存器重定位。 有关详细信息，请参见第 267 页中的 “寄存器符号” 。

.rel*name*、 .rela*name*
重定位信息，如第 247 页中的 “重定位节” 中所述。如果文件具有包括重定位的可装入段，则此节的属性将包括 SHF_ALLOC 位。否则，该位会处于禁用状态。通常，*name* 由应用重定位的节提供。因此，.text 的重定位节的名称通常为 .rel.text 或 .rela.text。

.rodata、 .rodata1
通常构成进程映像中的非可写段的只读数据。 有关详细信息，请参见第 275 页中的 “程序头” 。

.shstrtab
节名。

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

.symtab
符号表，如第 260 页中的 “符号表节” 中所述。如果文件具有包括符号表的可装入段，则此节的属性将包括 SHF_ALLOC 位。否则，该位会处于禁用状态。

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

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

.tdata、.tdata1
　这些节包含已初始化的线程局部数据，这些数据构成程序的内存映像。对于每个新执行流，系统会对数据内容的副本进行实例化。 有关详细信息，请参见第 8 章。

.text
　程序的**文本**或可执行指令。

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

.SUNW_heap
　从 dldump(3C) 中创建的动态可执行文件的**堆**。

.SUNW_cap
　硬件和软件的功能要求。 有关详细信息，请参见第 240 页中的 “硬件和软件功能节” 。

.SUNW_move
　部分初始化数据的附加信息。 有关详细信息，请参见第 243 页中的 “移动节” 。

.SUNW_reloc
　重定位信息，如第 247 页中的 “重定位节” 中所述。此节是多个重定位节的串联，用于为引用各个重定位记录提供更好的临近性。由于仅有重定位记录的偏移有意义，因此节的 sh_info 值为零。

.SUNW_syminfo
　其他符号表信息。 有关详细信息，请参见第 267 页中的 “Syminfo 表节” 。

.SUNW_version
　版本控制信息。 有关详细信息，请参见第 269 页中的 “版本控制节” 。

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

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

现有扩展使用其历史名称。

COMDAT 节

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

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

组节

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

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

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

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

当前定义了以下标志：

表 7–11 ELF 组节标志

名称	值
GRP_COMDAT	0x1

GRP_COMDAT

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

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

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

- 必须使用符号表各项中包含的 STB_GLOBAL 或 STB_WEAK 绑定和节索引 SHN_UNDEF，才能从组外的节中引用包含该组的节。包含引用的目标文件中定义的相同符号必须具有与独立于该引用的符号表项。组外的各节不能引用对组节中包含的地址具有 STB_LOCAL 绑定的符号，包括 STT_SECTION 类型的符号。

- 不允许从组外对包含该组的节进行非符号引用。例如，不能在 sh_link 或 sh_info 成员中使用组成员的节头索引。
- 如果废弃了某个组的成员，则可以删除所定义的与该组某一节相关的符号表项。如果此符号表项包含在不属于该组的符号表节中，则会进行此删除。

硬件和软件功能节

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

```
typedef struct {

        Elf32_Word      c_tag;

        union {

                Elf32_Word      c_val;

                Elf32_Addr      c_ptr;

        } c_un;

} Elf32_Cap;



typedef struct {

        Elf64_Xword     c_tag;

        union {

                Elf64_Xword     c_val;

                Elf64_Addr      c_ptr;

        } c_un;

} Elf64_Cap;
```

对于此类型的每一个目标文件，c_tag 会控制 c_un 的解释。

c_val
: 这些目标文件表示具有各种解释的整数值。

c_ptr
: 这些目标文件表示程序虚拟地址。

存在以下功能标记。

表 7–12 ELF 功能数组标记

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

CA_SUNW_NULL
标记功能数组的结尾。

CA_SUNW_HW_1
表示硬件功能值。c_val 元素包含用于表示关联硬件功能的值。在 SPARC 平台上，硬件功能在 sys/auxv_SPARC.h 中定义。在 x86 平台上，硬件功能在 sys/auxv_386.h 中定义。

CA_SUNW_SF_1
表示软件功能值。c_val 元素包含用于表示 sys/elf.h 中定义的关联软件功能的值。

可重定位目标文件可以包含功能节。链接编辑器会将多个可重定位输入目标文件中的所有功能节合并到一个单独的功能节中。使用链接编辑器，还可在生成目标文件时定义功能。请参见第 64 页中的 “标识硬件和软件功能” 。

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

利用不同硬件功能的动态库可以提供使用过滤器的灵活运行时环境。请参见第 367 页中的 “特定于硬件功能的共享库” 。

散列表节

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

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

图 7–4 符号散列表

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

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

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

散列函数如下所示：

```
unsigned long

elf_Hash(const unsigned char *name)

{

    unsigned long h = 0, g;



        while (*name)

        {

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

             if (g = h & 0xf0000000)

                  h ^= g >> 24;
```

```
                h &= ~g;

        }

        return h;

}
```

移动节

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

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

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

ELF32_Move 和 Elf64_Move 项的定义如下：

```
typedef struct {

        Elf32_Lword       m_value;

        Elf32_Word        m_info;

        Elf32_Word        m_poffset;

        Elf32_Half        m_repeat;

        Elf32_Half        m_stride;

} Elf32_Move;



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

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

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

```
typedef struct {
        Elf64_Lword       m_value;
        Elf64_Xword       m_info;
        Elf64_Xword       m_poffset;
        Elf64_Half        m_repeat;
        Elf64_Half        m_stride;
} Elf64_Move;

#define ELF64_M_SYM(info)       ((info)>>8)
#define ELF64_M_SIZE(info)      ((unsigned char)(info))
#define ELF64_M_INFO(sym, size) (((sym)<<8)+(unsigned char)(size))
```

这些结构的元素如下：

m_value
: 初始化值，即移到内存映像中的值。

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

m_poffset
: 与应用初始化的关联符号相关的偏移。

m_repeat
: 重复计数。

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

以下数据定义以前在目标文件中会占用 0x8000 个字节：

```
typedef struct {
        int     one;
```

```
        char    two;
} Data

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

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

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

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

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

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

注释节

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

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

namesz
descsz
type
name ...
desc ...

图 7–5 注释信息

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

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

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

下图中所示的注释段包含两项。

	+0	+1	+2	+3	
namesz	7				
descsz	0				无描述符
type	1				
name	X	Y	Z		
	C	o	\0	pad	
namesz	7				
descsz	8				
type	3				
name	X	Y	Z		
	C	o	\0	pad	
desc	word0				
	word1				

图 7–6 注释段示例

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

重定位节

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

可重定位项可具有以下结构。 请参见`sys/elf.h` 。

```
typedef struct {

        Elf32_Addr      r_offset;

        Elf32_Word      r_info;

} Elf32_Rel;
```

```
typedef struct {

        Elf32_Addr      r_offset;

        Elf32_Word      r_info;

        Elf32_Sword     r_addend;

} Elf32_Rela;



typedef struct {

        Elf64_Addr      r_offset;

        Elf64_Xword     r_info;

} Elf64_Rel;



typedef struct {

        Elf64_Addr      r_offset;

        Elf64_Xword     r_info;

        Elf64_Sxword    r_addend;

} Elf64_Rela;
```

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

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

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

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

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

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

```
#define ELF32_R_SYM(info)             ((info)>>8)

#define ELF32_R_TYPE(info)            ((unsigned char)(info))

#define ELF32_R_INFO(sym, type)       (((sym)<<8)+(unsigned char)(type))



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

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

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

                                        (Elf64_Xword)(type))
```

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

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

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

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

                                        (Elf64_Xword)(type))
```

r_addend
此成员指定常量加数，用于计算将存储在可重定位字段中的值。

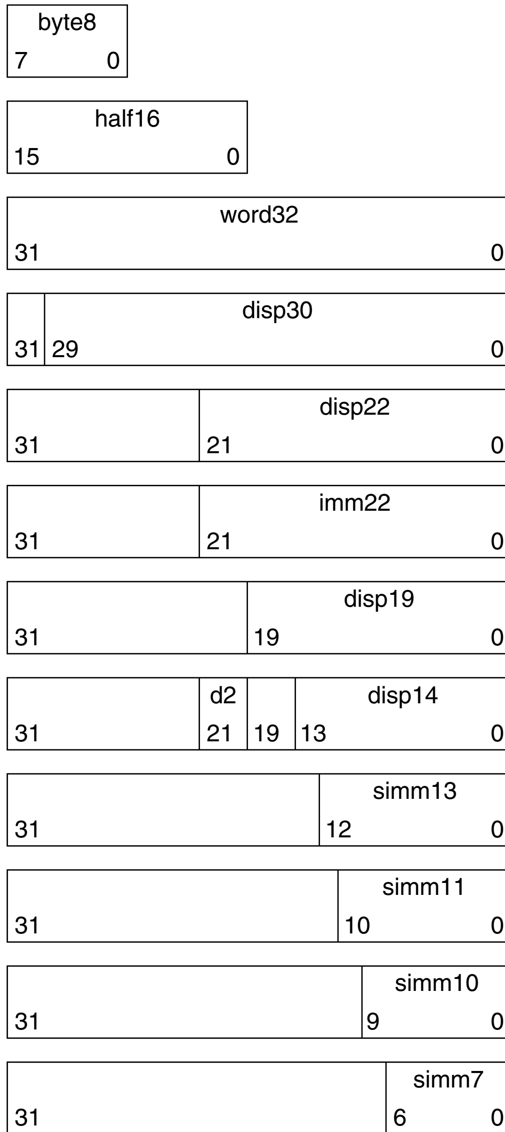
Rela 项包含显式加数。Rel 类型的项会在要修改的位置中存储一个隐式加数。32 位 SPARC 仅使用 Elf32_Rela 重定位项。 64 位 SPARC 和 64 位 x86 仅使用 Elf64_Rela 重定位项。因此， r_addend 成员可用作重定位加数。x86 仅使用 Elf32_Rel 重定位项。要重定位的字段包含该加数。在所有情况下，加数和计算所得的结果使用相同的字节顺序。

重定位节可以引用其他两个节：符号表（由 sh_link 节头项标识）和要修改的节（由 sh_info 节头项标识）。第 221 页中的 “节” 中指定了这些关系。如果可重定位目标文件中存在重定位节，则需要 sh_info 项，但对于可执行文件和共享库，该项是可选的。重定位偏移满足执行重定位的要求。

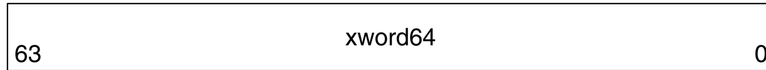
重定位类型（特定于处理器）

重定位项说明如何修改下图中的指令和数据字段。位数显示在框的下角。

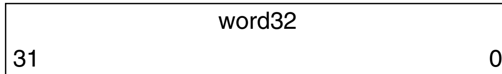
在 SPARC 平台上，重定位项应用于字节 (byte8)、半字 (half16) 或字。



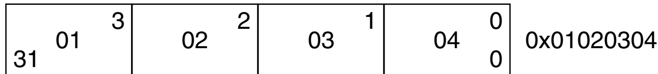
在 64 位 SPARC 和 x64 上，重定位项还会应用于扩展字 (`xword64`)：



在 x86 上，重定位项应用于字 (word32)：



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



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

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

A　　用于计算可重定位字段的值的加数。

B　　执行过程中将共享库装入内存的基本地址。通常，生成的共享库文件的基本虚拟地址为 0。但是，共享库的执行地址不相同。 请参见第 275 页中的 “程序头” 。

G　　执行过程中，重定位项的符号地址所在的全局偏移表中的偏移。 请参见第 299 页中的 “全局偏移表（特定于处理器）” 。

GOT　　全局偏移表的地址。 请参见第 299 页中的 “全局偏移表（特定于处理器）” 。

L　　符号的过程链接表项的节偏移或地址。 请参见第 299 页中的 “过程链接表（特定于处理器）” 。

P　　使用 r_offset 计算出的重定位的存储单元的节偏移或地址。

S　　索引位于重定位项中的符号的值。

SPARC: 重定位类型

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

表 7–13 SPARC: ELF 重定位类型

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

表 7–13 SPARC: ELF 重定位类型 （续）

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

表 7–13 SPARC: ELF 重定位类型 （续）

名称	值	字段	计算
R_SPARC_GOTDATA_HIX22	80	T-imm22	((S + A - GOT) >> 10) ^ ((S + A - GOT) >> 31)
R_SPARC_GOTDATA_LOX22	81	T-imm13	((S + A - GOT) & 0x3ff) | (((S + A - GOT) >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP_HIX22	82	T-imm22	(G >> 10) ^ (G >> 31)
R_SPARC_GOTDATA_OP_LOX22	83	T-imm13	(G & 0x3ff) | ((G >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP	84	Word32	请参阅此表后面的说明。

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

一些重定位类型的语义不只是简单的计算：

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

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

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

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

R_SPARC_COPY
: 由链接编辑器为动态可执行文件创建，用于保留只读文本段。此重定位偏移成员指向可写段中的位置。符号表索引指定应在当前目标文件和共享库中同时存在的符号。执行过程中，运行时链接程序将与共享库的符号关联的数据复制到偏移所指定的位置。 请参见第 144 页中的 “复制重定位” 。

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

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

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

R_SPARC_UA32
: 与 R_SPARC_32 类似，不同的是此重定位指向未对齐的字。必须将要重定位的字作为任意对齐的四个独立字节进行处理，而不是作为根据体系结构要求对齐的字进行处理。

R_SPARC_LM22
: 与 R_SPARC_HI22 类似，不同的是此重定位会进行截断而不是验证。

R_SPARC_PC_LM22
: 与 R_SPARC_PC22 类似，不同的是此重定位会进行截断而不是验证。

R_SPARC_HIX22
: 与 R_SPARC_LOX10 一起用于可执行文件，这些可执行文件在 64 位地址空间中的上限为 4 GB。与 R_SPARC_HI22 类似，但会提供链接值的补码。

R_SPARC_LOX10
: 与 R_SPARC_HIX22 一起使用。与 R_SPARC_LO10 类似，但始终设置链接值的位 10 到 12。

R_SPARC_L44
: 与 R_SPARC_H44 和 R_SPARC_M44 重定位类型一起使用，以生成 44 位的绝对寻址模型。

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

R_SPARC_GOTDATA_OP_HIX22、R_SPARC_GOTDATA_OP_LOX22 和 R_SPARC_GOTDATA_OP
: 这些重定位类型用于代码转换。

64 位 SPARC: 重定位类型

重定位计算中使用的以下表示法是特定于 64 位 SPARC 的。

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

下表中列出的重定位类型是扩展或修改针对 32 位 SPARC 定义的重定位类型所得的。请参见第 252 页中的 "SPARC: 重定位类型"。

表 7–14 64 位 SPARC: ELF 重定位类型

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

以下重定位类型的语义不只是简单的计算：

R_SPARC_OLO10
: 与 R_SPARC_LO10 类似，不同的是会添加额外的偏移，以充分利用 13 位带符号的直接字段。

32 位 x86: 重定位类型

下表中列出的重定位类型是针对 32 位 x86 定义的。

表 7–15 32 位 x86: ELF 重定位类型

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

表 7–15 32 位 x86: ELF 重定位类型 （续）

名称	值	字段	计算
R_386_RELATIVE	8	word32	B + A
R_386_GOTOFF	9	word32	S + A - GOT
R_386_GOTPC	10	word32	GOT + A - P
R_386_32PLT	11	word32	L + A
R_386_16	20	word16	L + A
R_386_PC16	21	word16	L + A - P
R_386_8	22	word8	L + A
R_386_PC8	23	word8	L + A - P

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

一些重定位类型的语义不只是简单的计算：

R_386_GOT32
: 计算 GOT 的基本地址与符号的 GOT 项之间的距离。此重定位还指示链接编辑器创建全局偏移表。

R_386_PLT32
: 计算符号的过程链接表项的地址，并指示链接编辑器创建一个过程链接表。

R_386_COPY
: 由链接编辑器为动态可执行文件创建，用于保留只读文本段。此重定位偏移成员指向可写段中的位置。符号表索引指定应在当前目标文件和共享库中同时存在的符号。执行过程中，运行时链接程序将与共享库的符号关联的数据复制到偏移所指定的位置。 请参见第 144 页中的 “复制重定位” 。

R_386_GLOB_DAT
: 用于将 GOT 项设置为所指定符号的地址。使用特殊重定位类型，可以确定符号和 GOT 项之间的对应关系。

R_386_JMP_SLOT
: 由链接编辑器为动态库创建，用于提供延迟绑定。此重定位偏移成员可指定过程链接表项的位置。运行时链接程序会修改过程链接表项，以将控制权转移到指定的符号地址。

R_386_RELATIVE
: 由链接编辑器为动态库创建。此重定位偏移成员可指定共享库中包含表示相对地址的值的位置。运行时链接程序通过将装入共享库的虚拟地址与相对地址相加，计算对应的虚拟地址。此类型的重定位项必须为符号表索引指定值零。

R_386_GOTOFF
计算符号的值与 GOT 的地址之间的差值。此重定位还指示链接编辑器创建全局偏移表。

R_386_GOTPC
与 R_386_PC32 类似，不同的是它在其计算中会使用 GOT 的地址。此重定位中引用的符号通常是 _GLOBAL_OFFSET_TABLE_，该符号还指示链接编辑器创建全局偏移表。

x64: 重定位类型

下表中列出的重定位是针对 x64 定义的。

表 7–16 x64: ELF 重定位类型

名称	值	字段	计算
R_AMD64_NONE	0	无	无
R_AMD64_64	1	word64	S + A
R_AMD64_PC32	2	word32	S + A - P
R_AMD64_GOT32	3	word32	G + A
R_AMD64_PLT32	4	word32	L + A - P
R_AMD64_COPY	5	无	请参阅此表后面的说明。
R_AMD64_GLOB_DAT	6	word64	S
R_AMD64_JUMP_SLOT	7	word64	S
R_AMD64_RELATIVE	8	word64	B + A
R_AMD64_GOTPCREL	9	word32	G + GOT + A - P
R_AMD64_32	10	word32	S + A
R_AMD64_32S	11	word32	S + A
R_AMD64_16	12	word16	S + A
R_AMD64_PC16	13	word16	S + A - P
R_AMD64_8	14	word8	S + A
R_AMD64_PC8	15	word8	S + A - P
R_AMD64_PC64	24	word64	S + A - P
R_AMD64_GOTOFF64	25	word64	S + A - GOT
R_AMD64_GOTPC32	26	work32	GOT + A + P

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

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。

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

图 7–7 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 7–17 ELF 字符串表索引

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

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

符号表节

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

符号表项具有以下格式。 请参见 `sys/elf.h`。

```
typedef struct {

        Elf32_Word      st_name;

        Elf32_Addr      st_value;

        Elf32_Word      st_size;

        unsigned char   st_info;
```

```
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;


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

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

st_value
: 关联符号的值。根据上下文，该值可以是绝对值或地址。 请参见第 266 页中的 “符号值” 。

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

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

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


#define ELF64_ST_BIND(info)          ((info) >> 4)
```

```
#define ELF64_ST_TYPE(info)          ((info) & 0xf)

#define ELF64_ST_INFO(bind, type)    (((bind)<<4)+((type)&0xf))
```

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

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

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

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

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

根据符号的 st_info 字段确定的符号绑定可确定链接可见性和行为。

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

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

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

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

STB_WEAK
弱符号。这些符号与全局符号类似，但其定义具有较低的优先级。

STB_LOOS－STB_HIOS
此范围内包含的值保留用于特定于操作系统的语义。

STB_LOPROC－STB_HIPROC
此范围内包含的值保留用于特定于处理器的语义。

全局符号和弱符号在以下两个主要方面不同：

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

  同样，如果存在通用符号，则出现相同名称的弱符号也不会导致错误。链接编辑器将使用通用定义，并忽略弱定义。通用符号具有包含 SHN_COMMON 的 st_shndx 字段。请参见第 39 页中的 “符号解析” 。
- 链接编辑器搜索归档库时，将会提取包含未定义全局符号或暂定全局符号的定义的归档成员。此成员的定义可以是全局符号或弱符号。

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

注 – 弱符号主要适用于系统软件。建议不要在应用程序中使用弱符号。

在每个符号表中，具有 STB_LOCAL 绑定的所有符号都优先于弱符号和全局符号。如第 221 页中的 “节” 中所述，符号表节的 sh_info 节头成员包含第一个非局部符号的符号表索引。

根据符号的 st_info 字段确定的符号类型用于对关联实体进行一般分类。

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

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

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

名称	值
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。此符号的处理与对 STT_OBJECT 的处理完全相同。

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

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

STT_LOOS － STT_HIOS
此范围内包含的值保留用于特定于操作系统的语义。

STT_LOPROC － STT_HIPROC
此范围内包含的值保留用于特定于处理器的语义。

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

表 7–20 ELF 符号可见性

名称	值
STV_DEFAULT	0
STV_INTERNAL	1
STV_HIDDEN	2
STV_PROTECTED	3

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

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

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

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

STV_INTERNAL
此可见性属性当前被保留。

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

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

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

SHN_ABS
: 此符号具有不会由于重定位而发生更改的绝对值。

SHN_COMMON 和 SHN_AMD64_LCOMMON
: 此符号标记尚未分配的通用块。与节的 sh_addralign 成员类似，符号的值也会指定对齐约束。链接编辑器在值为 st_value 的倍数的地址位置为符号分配存储空间。符号的大小会指明所需的字节数。

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

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

表 7–21 ELF 符号表项：索引 0

名称	值	说明
st_name	0	无名称
st_value	0	零值
st_size	0	无大小
st_info	0	无类型，本地绑定
st_other	0	
st_shndx	SHN_UNDEF	无节

符号值

不同目标文件类型的符号表的各项对于 st_value 成员的解释稍有不同。

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

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

寄存器符号

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

表 7–22 SPARC: ELF 符号表项：寄存器符号

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

下表中列出了为 SPARC 定义的寄存器值。

表 7–23 SPARC: ELF 寄存器编号

名称	值	含义
STO_SPARC_REGISTER_G2	0x2	%g2
STO_SPARC_REGISTER_G3	0x3	%g3

如果缺少特定全局寄存器的项，则意味着目标文件根本没有使用特定全局寄存器。

Syminfo 表节

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

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

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

Syminfo 项具有以下格式。 请参见 sys/link.h。

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


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

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

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

si_flags
此位字段可以设置标志，如下表所示。

名称	值	含义
SYMINFO_FLG_DIRECT	0x01	符号引用与包含定义的目标文件直接关联。
SYMINFO_FLG_COPY	0x04	符号定义通过副本重定位生成。
SYMINFO_FLG_LAZYLOAD	0x08	符号引用应延迟装入的目标文件。
SYMINFO_FLG_DIRECTBIND	0x10	符号引用应与定义直接绑定。
SYMINFO_FLG_NOEXTDIRECT	0x20	不允许将外部引用与此符号定义直接绑定。

版本控制节

链接编辑器创建的目标文件可以包含以下两种类型的版本控制信息：

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

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

版本定义节

此节由 SHT_SUNW_verdef 类型定义。如果此节存在，则必须同时存在 SHT_SUNW_versym 节。这两种结构在文件中提供符号与版本定义之间的关联。请参见第 155 页中的 “创建版本定义”。此节中的元素具有以下结构：

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


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

```
typedef struct {
        Elf64_Half      vd_version;
        Elf64_Half      vd_flags;
        Elf64_Half      vd_ndx;
        Elf64_Half      vd_cnt;
        Elf64_Word      vd_hash;
        Elf64_Word      vd_aux;
        Elf64_Word      vd_next;
} Elf64_Verdef;


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

vd_version
此成员标识该结构的版本，如下表中所列。

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

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

vd_flags
此成员包含特定于版本定义的信息，如下表中所列。

名称	值	含义
VER_FLG_BASE	0x1	文件的版本定义。

名称	值	含义
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。请参见第 159 页中的 “创建弱版本定义 (weak version definition)” 。

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

vd_cnt
Elf32_Verdaux 数组中的元素数目。

vd_hash
版本定义名称的散列值。该值是通过使用第 241 页中的 “散列表节” 中介绍的同一散列函数生成的。

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

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

vda_name
以空字符结尾的字符串的字符串表偏移，用于提供版本定义的名称。

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

版本符号节

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

```
typedef Elf32_Half      Elf32_Versym;

typedef Elf64_Half      Elf64_Versym;
```

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

表 7–24 ELF 版本依赖性索引

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

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

版本依赖性节

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

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


typedef struct {
        Elf32_Word      vna_hash;
        Elf32_Half      vna_flags;
        Elf32_Half      vna_other;
        Elf32_Word      vna_name;
        Elf32_Word      vna_next;
```

```
} Elf32_Vernaux;


typedef struct {
        Elf64_Half      vn_version;
        Elf64_Half      vn_cnt;
        Elf64_Word      vn_file;
        Elf64_Word      vn_aux;
        Elf64_Word      vn_next;
} Elf64_Verneed;


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

vn_version
此成员标识该结构的版本，如下表中所列。

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

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

vn_cnt
Elf32_Vernaux 数组中的元素数目。

vn_file
以空字符结尾的字符串的字符串表偏移，用于提供版本依赖性的文件名。此名称与文件中找到的 .dynamic 依赖项之一匹配。 请参见第 287 页中的 “动态节” 。

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

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

vna_hash
版本依赖性名称的散列值。该值是通过使用第 241 页中的 “散列表节” 中介绍的同一散列函数生成的。

vna_flags
版本依赖性特定信息，如下表中所列。

名称	值	含义
VER_FLG_WEAK	0x2	弱版本标识符。

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

vna_other
目前未使用。

vna_name
以空字符结尾的字符串的字符串表偏移，用于提供版本依赖性的名称。

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

动态链接

本节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。

可执行文件和共享库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。本节包括以下主要小节。

- 第 275 页中的 “程序头”，其中介绍了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。
- 第 280 页中的 “程序装入（特定于处理器）”，其中介绍了用于将程序装入内存的信息。
- 第 286 页中的 “运行时链接程序”，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

可执行文件或共享库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，如第 280 页中的 “段内容”中所述。

程序头仅对可执行文件和共享库文件有意义。文件使用 ELF 头的 e_phentsize 和 e_phnum 成员来指定各自的程序头大小。

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 7–25 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库具有未指定的内容。

`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 7–26 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）。请参见第 280 页中的 “程序装入（特定于处理器）”。

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 7–25 ELF 段类型

名称	值
PT_NULL	0
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_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
指定可装入段，通过 p_filesz 和 p_memsz 进行描述。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。请参见第 287 页中的 “动态节”。

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库中。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，请参见第 286 页中的 “程序的解释程序” 。

PT_NOTE
: 指定辅助信息的位置和大小。 有关详细信息，请参见第 246 页中的 “注释节” 。

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，请参见第 286 页中的 “程序的解释程序” 。

PT_TLS
: 指定线程局部存储模板。 有关详细信息，请参见第 314 页中的 “线程局部存储节 ” 。

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_LOSUNW - PT_HISUNW
: 此范围内包含的值保留用于特定于 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
: 指定硬件功能要求。 有关详细信息，请参见第 240 页中的 “硬件和软件功能节” 。

PT_LOPROC - PT_HIPROC
: 此范围内包含的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

可执行文件和共享库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。

可执行文件或共享库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。 请参见第 280 页中的 “程序装入（特定于处理器）” 。

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 7–26 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 7–27 ELF 段权限

标志	值	确切解释	允许解释
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
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 7–10。

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，请参见第 299 页中的 “全局偏移表（特定于处理器）” 和第 299 页中的 “过程链接表（特定于处理器）” 。

类型为 SHT_NOBITS 的节不会占用文件空间，但却可构成段的内存映像。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，可执行文件和共享库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，文件都适合进行换页。

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 上的地址空间内，包括其文本、数据、堆、栈和共享库依赖性。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 上的地址空间内进行链接，但仍可以使用 `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。

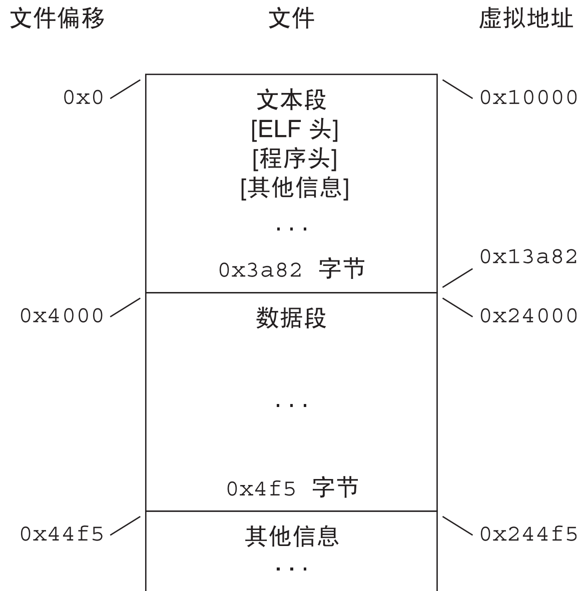


图 7–8 SPARC: 可执行文件（64 K 对齐）

下表定义了上图中可装入段的各元素。

表 7–28 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。

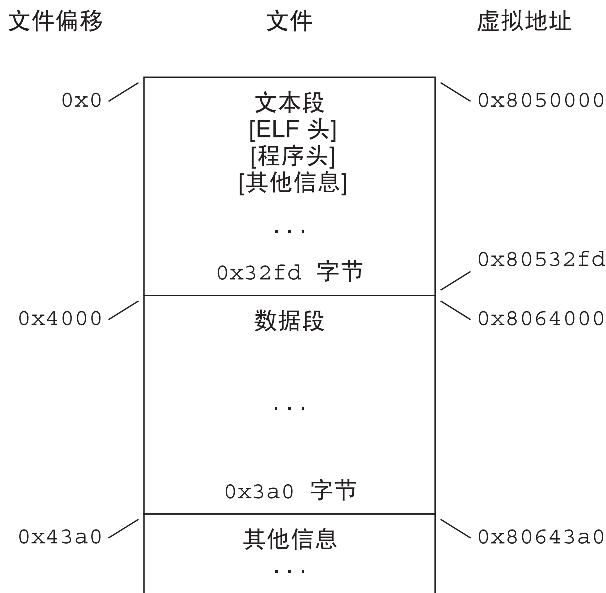


图 7–9 32 位 x86: 可执行文件（64 K 对齐）

下表定义了上图中可装入段的各元素。

表 7–29 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
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其文本段取整的典型的 Solaris 系统二进制文件。

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。

其他三个页面中的混合内容在逻辑上不属于进程映像。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。

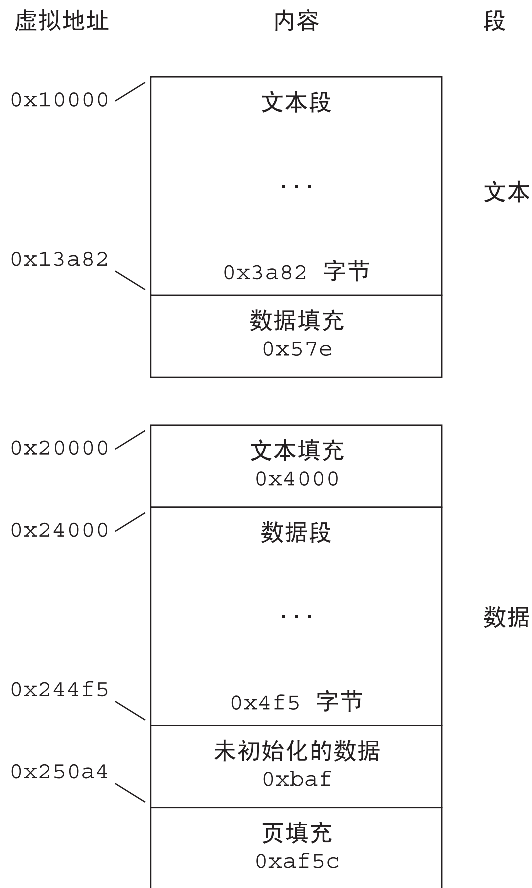


图 7–10 32 位 SPARC: 进程映像段

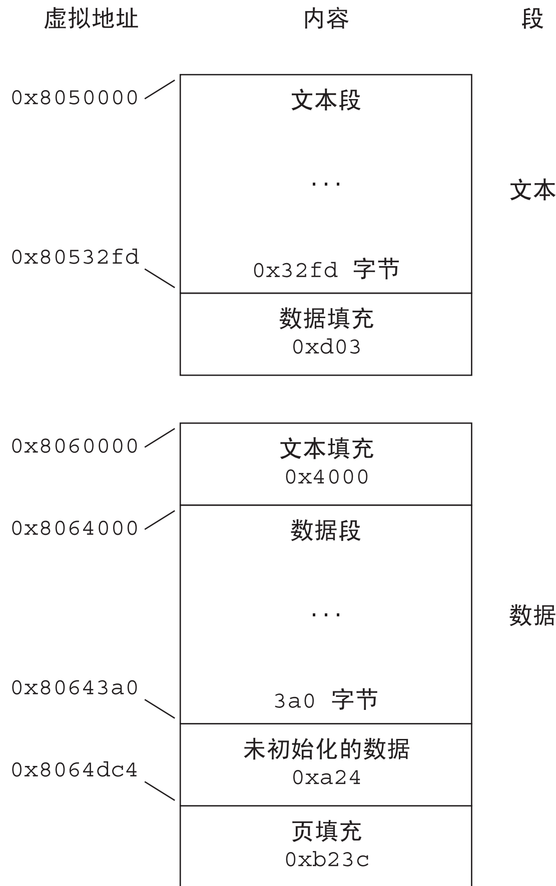


图 7–11 x86: 进程映像段

可执行文件和共享库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，共享库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库虚拟地址，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 7–30 32 位 SPARC: ELF 共享库段地址示例

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 7–31 32 位 x86: ELF 共享库段地址示例

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库可以包含一个 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 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库时，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) 和运行时链接程序可进行协作，以便为程序创建进程映像。

链接编辑器可为可执行文件和共享库文件构造协助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。

共享库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。

除两个特殊兼容性范围中的那些标记外，每个动态标记的值都可确定 d_un 联合的解释。借助此约定，外部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，也即是该标记既不使用 d_ptr，也不使用 d_val。值小于特定值 DT_ENCODING 的标记以及值位于 DT_HIOS 和 DT_LOPROC 之间的标记不遵循这些规则。

下表概述了可执行文件和共享库文件的标记要求。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 7–32 ELF 动态数组标记

名称	值	d_un	可执行文件	共享库文件
DT_NULL	0	忽略	强制	强制
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	强制	强制
DT_STRTAB	5	d_ptr	强制	强制
DT_SYMTAB	6	d_ptr	强制	强制
DT_RELA	7	d_ptr	强制	可选
DT_RELASZ	8	d_val	强制	可选
DT_RELAENT	9	d_val	强制	可选
DT_STRSZ	10	d_val	强制	强制
DT_SYMENT	11	d_val	强制	强制
DT_INIT	12	d_ptr	可选	可选
DT_FINI	13	d_ptr	可选	可选
DT_SONAME	14	d_val	忽略	可选
DT_RPATH	15	d_val	可选	可选
DT_SYMBOLIC	16	忽略	忽略	可选

表 7–32 ELF 动态数组标记　（续）

名称	值	d_un	可执行文件	共享库文件
DT_REL	17	d_ptr	强制	可选
DT_RELSZ	18	d_val	强制	可选
DT_RELENT	19	d_val	强制	可选
DT_PLTREL	20	d_val	可选	可选
DT_DEBUG	21	d_ptr	可选	忽略
DT_TEXTREL	22	忽略	可选	可选
DT_JMPREL	23	d_ptr	可选	可选
DT_BIND_NOW	24	忽略	可选	可选
DT_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	忽略
DT_PREINIT_ARRAYSZ	33	d_val	可选	忽略
DT_LOOS	0x6000000d	未指定	未指定	未指定
DT_SUNW_RTLDINF	0x6000000e	d_ptr	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_FEATURE_1	0x6ffffdfc	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

表 7–32 ELF 动态数组标记 （续）

名称	值	d_un	可执行文件	共享库文件
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。 请参见第 76 页中的 “共享库依赖项” 。

DT_PLTRELSZ
: 与过程链接表关联的重定位项的总大小（以字节为单位）。请参见第 299 页中的 “过程链接表（特定于处理器）”。

DT_PLTGOT
: 与过程链接表或全局偏移表关联的地址。请参见第 299 页中的 “过程链接表（特定于处理器）”和第 299 页中的 “全局偏移表（特定于处理器）”。

DT_HASH
: 符号散列表的地址。该表引用 DT_SYMTAB 元素指示的符号表。请参见第 241 页中的 “散列表节”。

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。请参见第 259 页中的 “字符串表节”。

DT_SYMTAB
: 符号表的地址。请参见第 260 页中的 “符号表节”。

DT_RELA
: 重定位表的地址。请参见第 247 页中的 “重定位节”。

目标文件可以有多个重定位节。为可执行文件或共享库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库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。如果文件必须重定位，则可以存在 DT_RELA 或 DT_REL。

DT_RELASZ
: DT_RELA 重定位表的总大小（以字节为单位）。

DT_RELAENT
: DT_RELA 重定位项的大小（以字节为单位）。

DT_STRSZ
: DT_STRTAB 字符串表的总大小（以字节为单位）。

DT_SYMENT
: DT_SYMTAB 符号项的大小（以字节为单位）。

DT_INIT
: 初始化函数的地址。请参见第 36 页中的 “初始化和终止节”。

DT_FINI
: 终止函数的地址。请参见第 36 页中的 “初始化和终止节”。

DT_SONAME
: 以空字符结尾的字符串的 DT_STRTAB 字符串表偏移，用于标识共享库的名称。请参见第 120 页中的 “记录共享库名称”。

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。请参见第 76 页中的 "运行时链接程序搜索的目录"。

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。此元素的用途已被 DF_SYMBOLIC 标志取代。请参见第 148 页中的 "使用 -B symbolic"。

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（以字节为单位）。

DT_RELENT
DT_REL 重定位项的大小（以字节为单位）。

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。请参见第 299 页中的 "过程链接表（特定于处理器）"。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可以相应地进行准备。此元素的用途已被 DF_TEXTREL 标志取代。请参见第 137 页中的 "与位置无关的代码"。

DT_JMPREL
与过程链接表单独关联的重定位项的地址。请参见第 299 页中的 "过程链接表（特定于处理器）"。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 7–35。

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。请参见第 84 页中的 "执行重定位的时间"。

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。请参见第 36 页中的 "初始化和终止节"。

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。请参见第 36 页中的 "初始化和终止节"。

DT_INIT_ARRAYSZ
DT_INIT_ARRAY 数组的总大小（以字节为单位）。

DT_FINI_ARRAYSZ
DT_FINI_ARRAY 数组的总大小（以字节为单位）。

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。 请参见第 76 页中的 “运行时链接程序搜索的目录”。

DT_FLAGS
特定于此目标文件的标志值。 请参见表 7–33。

DT_ENCODING
大于或等于 DT_ENCODING 且小于或等于 DT_HIOS 的动态标记值，遵循 d_un 联合的解释规则。

DT_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库中，则会被忽略。 请参见第 36 页中的 “初始化和终止节”。

DT_PREINIT_ARRAYSZ
DT_PREINIT_ARRAY 数组的总大小（以字节为单位）。

DT_LOOS - DT_HIOS
此范围内包含的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_RTLDINF
保留供运行时链接程序内部使用。

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。 请参见第 267 页中的 “Syminfo 表节”。

DT_SYMINENT
DT_SYMINFO 信息项的大小（以字节为单位）。

DT_SYMINSZ
DT_SYMINFO 表的总大小（以字节为单位）。

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。 请参见第 269 页中的 “版本定义节”。

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。 请参见第 272 页中的 “版本依赖性节”。

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。 请参见第 143 页中的 “组合重定位节” 。

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。 请参见第 143 页中的 “组合重定位节” 。

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。 请参见第 129 页中的 “生成辅助过滤器” 。

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。 请参见第 126 页中的 “生成标准过滤器” 。

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（以字节为单位）。

DT_MOVESZ
DT_MOVETAB 表的总大小（以字节为单位）。

DT_MOVETAB
移动表的地址。此元素要求同时存在 DT_MOVEENT 和 DT_MOVESZ 元素。请参见第 243 页中的 “移动节” 。

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。请参见第 79 页中的 “配置缺省搜索路径” 。

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。请参见第 186 页中的 “运行时链接程序审计接口” 。

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。请参见第 186 页中的 “运行时链接程序审计接口” 。

DT_FLAGS_1
特定于此目标文件的标志值。请参见表 7–34。

DT_FEATURE_1
特定于此目标文件的功能值。请参见表 7–36。

DT_VALRNGLO - DT_VALRNGHI
　此范围内包含的值使用动态结构的 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。请参见第 267 页中的 “寄存器符号”。

DT_LOPROC - DT_HIPROC
　此范围内包含的值保留用于特定于处理器的语义。

除动态数组结尾的 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 7–33 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。请参见第 373 页中的 “查找关联的依赖项”。

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。请参见第 148 页中的 “使用 -B symbolic”。

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。请参见第 137 页中的 “与位置无关的代码”。

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。请参见第 84 页中的 “执行重定位的时间”。

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) 或延迟装入动态装入的目标文件中，不能使用静态线程局部存储。由于此限制，因此链接编辑器不支持创建要求静态线程局部存储的共享库。

表 7–34 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。
DF_1_NOKSYMS	0x80000	内部使用。
DF_1_NORELOC	0x400000	内部使用。

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。请参见第 84 页中的“执行重定位的时间”。

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。请参见第 104 页中的 “目标文件分层结构” 。

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。请参见第 133 页中的 “filtee 处理” 。

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。请参见第 373 页中的 “查找关联的依赖项” 。

DF_1_DIRECT
: 表示目标文件应使用直接绑定信息。请参见第 82 页中的 “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。此标志通过使用链接编辑器的 -z interpose 选项进行记录。请参见第 82 页中的 “直接绑定” 。

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。请参见第 35 页中的 “运行时链接程序搜索的目录” 。

DF_1_NODUMP
: 表示此目标文件不通过 dldump(3C) 进行转储。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。请参见第 371 页中的 “减少 filtee 搜索” 。

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已被废弃，因此此目标文件中将不再存在这些记录。请参见第 68 页中的 “位移重定位” 。

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。请参见第 68 页中的 “位移重定位” 。

DF_1_NODIRECT
表示此目标文件包含无法直接绑定的符号。请参见第 49 页中的 “定义其他符号” 。

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。

表 7–35 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。请参见第 87 页中的 “延迟装入动态依赖项” 。

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。请参见第 103 页中的 “隔离组” 。

表 7–36 ELF 动态功能标志 DT_FEATURE_1

名称	值	含义
DTF_1_PARINIT	0x1	需要部分初始化。
DTF_1_CONFEXP	0x2	需要配置文件。

DTF_1_PARINIT
表示目标文件需要部分初始化。请参见第 243 页中的 “移动节” 。

DTF_1_CONFEXP
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*。此标志的效果与 DF_1_CONFALT 相同。

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库具有不同的 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 重定位项。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库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。符号 _GLOBAL_OFFSET_TABLE_ 可用于访问该表。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库之间的执行传输（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。可执行文件和共享库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 动态库，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 7–37 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 7–37 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 7–37 32 位 SPARC: 过程链接表示例

目标文件	内存段
.PLT0:	.PLT0:
unimp	save %sp, -64, %sp
unimp	call runtime_linker
unimp	nop
.PLT1:	.PLT1:
unimp	.word identification
unimp	unimp
unimp	unimp

表 7–37 32 位 SPARC: 过程链接表示例　（续）

目标文件	内存段
`.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方。
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，但可以安全废弃以前的内容。
- 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。

64 位 SPARC: 过程链接表

对于 64 位 SPARC 动态库，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 7–38 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 7–38 显示了若干项。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 7–38 64 位 SPARC: 过程链接表示例

目标文件	内存段
.PLT0:	.PLT0:
unimp	save %sp, -176, %sp
unimp	sethi %hh(runtime_linker_0), %l0
unimp	sethi %lm(runtime_linker_0), %l1
unimp	or %l0, %hm(runtime_linker_0), %l0
unimp	sllx %l0, 32, %l0
unimp	or %l0, %l1, %l0
unimp	jmpl %l0+%lo(runtime_linker_0), %o1
unimp	mov %g1, %o0
.PLT1:	.PLT1:
unimp	save %sp, -176, %sp
unimp	sethi %hh(runtime_linker_1), %l0
unimp	sethi %lm(runtime_linker_1), %l1
unimp	or %l0, %hm(runtime_linker_1), %l0
unimp	sllx %l0, 32, %l0
unimp	or %l0, %l1, %l0
unimp	jmpl %l0+%lo(runtime_linker_0), %o1
unimp	mov %g1, %o0
.PLT2:	.PLT2:
unimp	.xword identification

表 7–38 64 位 SPARC: 过程链接表示例　（续）

目标文件	内存段
`.PLT101:`	`.PLT101:`
`    sethi   (.-.PLT0), %g1`	`    nop`
`    ba,a    %xcc, .PLT1`	`    mov     %o7,  %g1`
`    nop`	`    call    name101`
`    nop`	`    mov     %g1, %o7`
`    nop;    nop`	`    nop;    nop`
`    nop;    nop`	`    nop;    nop`
`.PLT102:`	`.PLT102:`
`    sethi   (.-.PLT0), %g1`	`    nop`
`    ba,a    %xcc, .PLT1`	`    sethi   %hi(name102), %g1`
`    nop`	`    jmpl    %g1+%lo(name102), %g0`
`    nop`	`    nop`
`    nop;    nop`	`    nop;    nop`
`    nop;    nop`	`    nop;    nop`
`.PLT103:`	`.PLT103:`
`    sethi   (.-.PLT0),  %g1`	`    nop`
`    ba,a    %xcc, .PLT1`	`    sethi   %hh(name103), %g1`
`    nop`	`    sethi   %lm(name103), %g5`
`    nop`	`    or      %hm(name103), %g1`
`    nop`	`    sllx    %g1, 32, %g1`
`    nop`	`    or      %g1, %g5, %g5`
`    nop`	`    jmpl    %g5+%lo(name103), %g0`
`    nop`	`    nop`

表 7–38 64 位 SPARC: 过程链接表示例 （续）

目标文件	内存段
.PLT32768:	.PLT32768:
mov %o7, %g5	<unchanged>
call .+8	<unchanged>
nop	<unchanged>
ldx [%o7+.PLTP32768 -	<unchanged>
(.PLT32768+4)], %g1	
jmpl %o7+%g1, %g1	<unchanged>
mov %g5, %o7	<unchanged>
...	...
.PLT32927:	.PLT32927:
mov %o7, %g5	<unchanged>
call .+8	<unchanged>
nop	<unchanged>
ldx [%o7+.PLTP32927 -	<unchanged>
(.PLT32927+4)], %g1	
jmpl %o7+%g1, %g1	<unchanged>
mov %g5, %o7	<unchanged>

表 7–38 64 位 SPARC: 过程链接表示例 （续）

目标文件	内存段

`.PLTP32768`
`    .xword  .PLT0 -`
`            (.PLT32768+4)`
`    ...`
 	
`.PLTP32927`
`    .xword  .PLT0 -`
`            (.PLT32927+4)`
	`.PLTP32768`
	`    .xword  name32768 -`
	`            (.PLT32768+4)`
	`    ...`
	
	`.PLTP32927`
	`    .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方。
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 动态库，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。可执行文件和共享库文件包含不同的过程链接表。

表 7–39 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 7–40 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库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，全局偏移表包含以下 `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 动态库，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。可执行文件和共享库文件包含不同的过程链接表。

表 7–41 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库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，全局偏移表包含以下 `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 8 章

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，但更多时候用首字母缩略词 TLS 表示。通过将变量声明为线程局部变量，编译器会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、文件范围的静态变量或函数范围的静态变量。它对始终为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，则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初始化为线程局部变量的地址。

绑定

可以在外部声明和引用线程局部变量。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)` 动态装入共享库。如果使用动态 TLS 模型编译每个包含对线程局部变量的引用的转换单元，则可以在启动后装入包含该引用的共享库。

静态 TLS 模型生成执行速度更快的代码。但是，编译为使用此模型的代码不能在启动后动态装入的库中引用线程局部变量。动态 TLS 模型可以引用所有 TLS。这些模型将在第 318 页中的“线程局部存储的访问模型”中介绍。

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由此运算符获取的地址，只要计算该地址的线程仍然存在。当线程终止时，指向该线程中的线程局部变量的所有指针将变为无效。

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

编译时已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，可使用 TLS 节指定大小和初始内容。

编译环境在使用 `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在 `.tdata` 或 `.tdata1` 节中分配。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，可使用此模板分配 TLS。

此模板的已初始化部分称为 TLS 初始化映像。将由于已初始化的线程局部变量而产生的所有重定位应用于此模板。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`。这些符号被指定相对于 TLS 模板的开头的偏移。与这些符号关联的实际虚拟地址无关紧要。地址仅指向模板，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库中，对于已定义的符号，`STT_TLS` 符号的 `st_value` 字段包含指定的偏移，或者对于未定义的符号，此字段包含零。

定义了多个重定位以支持访问 TLS。请参阅第 326 页中的 “SPARC: 线程局部存储的重定位类型”、第 332 页中的 “32 位 x86: 线程局部存储的重定位类型”和第 338 页中的 “x64: 线程局部存储的重定位类型”。TLS 重定位仅引用 STT_TLS 类型的符号。

在动态可执行文件和共享库中，PT_TLS 程序项描述 TLS 模板。此模板包含以下成员。

表 8–1 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，将会在三个时间创建 TLS。

- 程序启动时。
- 创建新线程时。
- 程序启动后装入共享库之后，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 8–1 中所示。

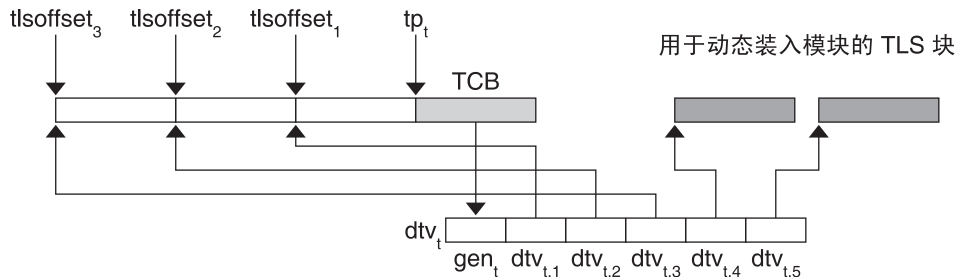


图 8–1 线程局部存储的运行时存储布局

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库（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库的 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 $align_{m+1}$ 分别是动态库 m 的分配模板的大小和对齐方式。其中，$1 <= m <= M$，M 是已装入动态库的总数。round(offset, align) 函数返回向上舍入为最接近 *align* 倍数的偏移。TLS 模板正好放置在线程指针 tp_t 的前面。根据从 tp_t 中减去的数字访问 TLS 数据。

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 $tlsoffset_M$。

然后，运行时链接程序构造初始化记录的链接表。此列表中的每条记录都描述一个已装入动态库的 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。将初始化此存储空间，并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动态库分配一个新的 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编号 gen_t，该生成编号用于确定需要扩展向量的时间。请参见第 317 页中的 “延迟分配线程局部存储块”。

向量 $dtv_{t,m}$ 中剩余的每个元素都是一个指针（指向为属于动态库 m 的 TLS 保留的块）。

对于启动后动态装入的目标文件，线程库延迟分配 TLS 块。将在第一次引用已装入目标文件中的 TLS 变量时进行分配。必须使用动态 TLS 模型引用在启动后动态装入的目标文件中定义的 TLS。对于延迟分配的块，将指针 $dtv_{t,m}$ 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 的值从值 1 开始。

然后，线程库将初始化映像复制到新存储块中的相应位置。

启动后动态装入

可以在进程启动后装入包含 TLS 的共享库。因此，运行时链接程序必须扩展初始化记录列表，以包含新目标文件的初始化模板。为新目标文件指定索引 m = M + 1，并以 1 为增量累加计数器 M。但是，直到实际引用块时才会分配新的 TLS 块。

卸载包含 TLS 的库时，将释放该库使用的 TLS 块。

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。为了表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() 都检查每线程生成计数器 gen_t，以便确定是否需要更新向量。如果向量 dtv_t 已过时，则例程将更新此向量，可能会重新分配此向量以便为更

多项留出空间。然后，例程将检查是否已分配与 `dtv`$_{t,m}$ 对应的 TLS 块。如果尚未分配此向量，则例程将使用运行时链接程序提供的初始化记录列表中的信息来分配并初始化块。指针 `dtv` $_{t,m}$ 被设置为指向已分配的块。例程返回一个指向块中的给定偏移的指针。

线程局部存储的访问模型

每个 TLS 引用都遵循下列其中一个访问模型。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
: 此模型允许从共享库或动态可执行文件引用所有 TLS 变量。如果第一次从特定线程引用 TLS 块，则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 *GD* 模型的优化模型。如果编译器确定变量在要生成的动态库中将局部绑定或受到保护，则编译器将指示链接编辑器静态绑定动态 `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`$_{0,m}$ 地址。将在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 模板中包含的 TLS 变量。此模板由进程启动时可用的所有 TLS 块组成。在此模型中，给定变量 *x* 的相对于线程指针的偏移存储在 *x* 的 `GOT` 项中。此模型不能从在初始进程启动后通过延迟装入、过滤器或 `dlopen`(3C) 装入的共享库引用 TLS 变量。此模型不能访问使用延迟分配的 TLS 块。

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`。不能使用此模型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）。使用独特的 TLS 重定位可实现此转换。这些重定位不仅请求执行更新，还标识要使用的 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可以执行转换。例如，如果要将使用 *GD* 访问模型的可重定位目标文件链接到动态可执行文件，则链接编辑器可以根据需要转换使用 *IE* 或 *LE* 访问模型的引用。然后执行模型所需的重定位。

下图说明了不同的访问模型，以及一个模型可以转换为另一个模型的时间。

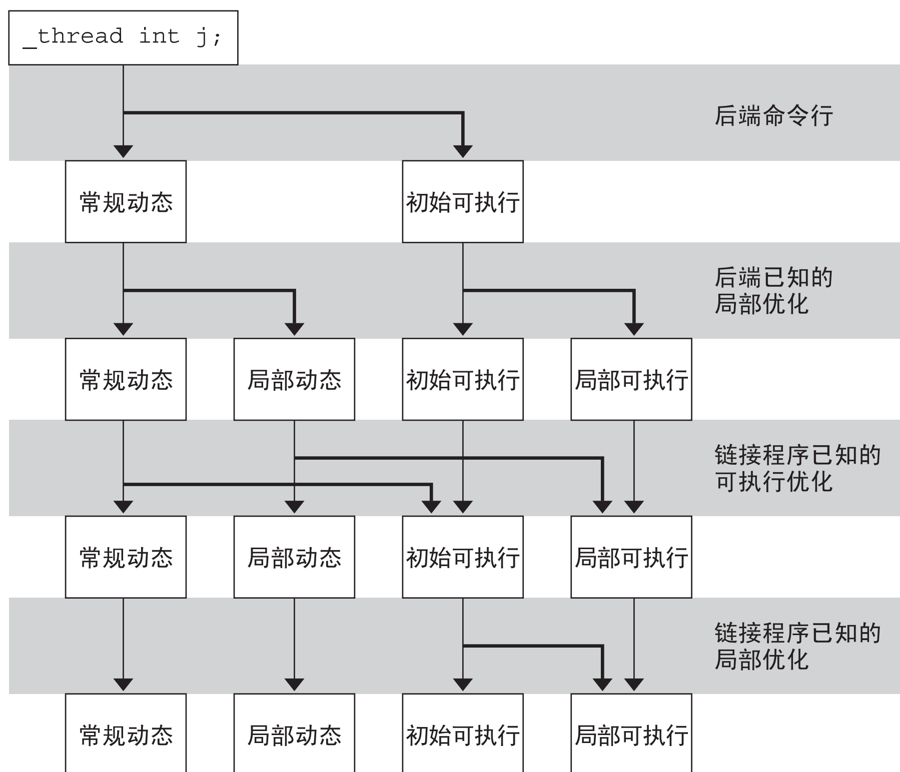


图 8–2 线程局部存储的访问模型和转换

SPARC: 访问线程局部变量

在 SPARC 上，可使用以下代码序列模型来访问线程局部变量。

SPARC: 常规动态 (General Dynamic, GD)

此代码序列实现第 318 页中的 “线程局部存储的访问模型”中介绍的 GD 模型。

表 8–2 SPARC: 常规动态线程局部变量的访问代码

代码序列	初始重定位	符号

表 8–2 SPARC: 常规动态线程局部变量的访问代码 （续）

# %l7 - initialized to GOT pointer		
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	x
# %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。链接编辑器通过将相对于 GOT 的偏移替换为新的 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，链接编辑器根据 GOT 放置 R_SPARC_TLS_DTPMOD32 和 R_SPARC_TLS_DPTOFF32 重定位，以便运行时链接程序处理。

第二个 add 指令导致生成 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，才使用此重定位。

call 指令使用特殊语法 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。不能将 add 指令放在调用的延迟槽中。由于后面进行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)

此代码序列实现第 318 页中的 “线程局部存储的访问模型”中介绍的 LD 模型。

表 8–3 SPARC: 局部动态线程局部变量的访问代码

代码序列	初始重定位	符号

表 8–3 SPARC: 局部动态线程局部变量的访问代码 （续）

# %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		
0x10 sethi %hi(@dtpoff(x1)), %l1	R_SPARC_TLS_LDO_HIX22	x1
0x14 xor %l1, %lo(@dtpoff(x1)), %l1	R_SPARC_TLS_LDO_LOX10	x1
0x18 add %o0, %l1, %l1	R_SPARC_TLS_LDO_ADD	x1
# %l1 - contains address of local TLS variable x1		
0x20 sethi %hi(@dtpoff(x2)), %l2	R_SPARC_TLS_LDO_HIX22	x2
0x24 xor %l2, %lo(@dtpoff(x2)), %l2	R_SPARC_TLS_LDO_LOX10	x2
0x28 add %o0, %l2, %l2	R_SPARC_TLS_LDO_ADD	x2
# %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**

表 8–3 SPARC: 局部动态线程局部变量的访问代码　　（续）

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。链接编辑器通过将相对于 GOT 的偏移替换为新的 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 指令分别由 R_SPARC_TLS_LDM_ADD 和 R_SPARC_TLS_LDM_CALL 重定位标记。

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。在链接编辑时将确定每个局部符号的 TLS 偏移，因此将直接填充这些值。add 指令使用 R_SPARC_TLS_LDO_ADD 重定位标记。

当一个过程引用多个局部符号时，编译器将生成一次获取 TLS 块的基本地址的代码。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)

此代码序列实现第 318 页中的 “线程局部存储的访问模型”中介绍的 IE 模型。

表 8–4 32 位 SPARC: 初始可执行的线程局部变量的访问代码

代码序列	初始重定位	符号

表 8–4 32 位 SPARC: 初始可执行的线程局部变量的访问代码 （续）

# %l7 - initialized to GOT pointer, %g7 - thread pointer		
0x00 sethi %hi(@tpoff(x)), %o0	R_SPARC_TLS_IE_HI22	x
0x04 or %o0, %lo(@tpoff(x)), %o0	R_SPARC_TLS_IE_LO10	x
0x08 ld [%l7 + %o0], %o0	R_SPARC_TLS_IE_LD	x
0x0c add %g7, %o0, %o0	R_SPARC_TLS_IE_ADD	x
# %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 重定位未完成，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)

此代码序列实现第 318 页中的 “线程局部存储的访问模型”中介绍的 IE 模型。

表 8–5 64 位 SPARC: 初始可执行的线程局部变量的访问代码

代码序列	初始重定位	符号

表 8–5 64 位 SPARC: 初始可执行的线程局部变量的访问代码 （续）

# %l7 - initialized to GOT pointer, %g7 - thread pointer		
0x00 sethi %hi(@tpoff(x)), %o0	R_SPARC_TLS_IE_HI22	x
0x04 or %o0, %lo(@tpoff(x)), %o0	R_SPARC_TLS_IE_LO10	x
0x08 ldx [%l7 + %o0], %o0	R_SPARC_TLS_IE_LD	x
0x0c add %g7, %o0, %o0	R_SPARC_TLS_IE_ADD	x
# %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)

此代码序列实现第 318 页中的 “线程局部存储的访问模型”中介绍的 LE 模型。

表 8–6 SPARC: 局部可执行的线程局部变量的访问代码

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。此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 项中的第一项的地址。

`@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()`。将此结构的 `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`。将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 8–7 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。

表 8–7 SPARC: 线程局部存储的重定位类型 （续）

名称	值	字段	计算
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 指针的寄存器是此序列中的第一个寄存器。此重定位标记的指令出现在 R_SPARC_TLS_GD_CALL 重定位标记的 call 指令前面。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
: 可以按处理引用 __tls_get_addr() 函数的 R_SPARC_WPLT30 重定位的方式处理此重定位。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 指针的寄存器是此序列中的第一个寄存器。此重定位标记的指令出现在 R_SPARC_TLS_GD_CALL 重定位标记的 call 指令前面。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
: 可以按处理引用 __tls_get_addr() 函数的 R_SPARC_WPLT30 重定位的方式处理此重定位。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: 访问线程局部变量

在 x86 上，可使用以下代码序列模型来访问 TLS。

32 位 x86: 常规动态 (General Dynamic, GD)

此代码序列实现第 318 页中的 “线程局部存储的访问模型”中介绍的 GD 模型。

表 8–8 32 位 x86: 常规动态线程局部变量的访问代码

代码序列	初始重定位	符号
0x00 leal x@tlsgd(,%ebx,1), %eax	R_386_TLS_GD	x
0x07 call x@tlsgdplt	R_386_TLS_GD_PLT	x
# %eax - contains address of TLS variable		
	未完成的重定位	**符号**
GOT[n]	R_386_TLS_DTPMOD32	x
GOT[n + 1]	R_386_TLS_DTPOFF32	

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。链接编辑器通过将相对于 GOT 的偏移替换为新的 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 放置 R_386_TLS_DTPMOD32 和 R_386_TLS_DTPOFF32 重定位，以便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)

此代码序列实现第 318 页中的 "线程局部存储的访问模型"中介绍的 LD 模型。

表 8–9 32 位 x86: 局部动态线程局部变量的访问代码

代码序列	初始重定位	符号
`0x00 leal  x1@tlsldm(%ebx), %eax`	`R_386_TLS_LDM`	`x1`
`0x06 call  x1@tlsldmplt`	`R_386_TLS_LDM_PLT`	`x1`
`# %eax - contains address of TLS block of current object`		
`0x10 leal  x1@dtpoff(%eax), %edx`	`R_386_TLS_LDO_32`	`x1`
`# %edx - contains address of local TLS variable x1`		
`0x20 leal  x2@dtpoff(%eax), %edx`	`R_386_TLS_LDO_32`	`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。链接编辑器通过将相对于 `GOT` 的偏移替换为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，并在此结构的 `ti_tlsoffset` 字段中填充零。`call` 指令使用 `R_386_TLS_LDM_PLT` 重定位标记。

在链接编辑时将确定每个局部符号的 TLS 偏移，因此链接编辑器将直接填充这些值。

当一个过程引用多个局部符号时，编译器将生成一次获取 TLS 块的基本地址的代码。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)

此代码序列实现第 318 页中的 "线程局部存储的访问模型"中介绍的 IE 模型。

IE 模型存在两个代码序列。一个序列用于使用 GOT 指针的位置无关代码。另一个序列用于不使用 GOT 指针的位置相关代码。

表 8–10 32 位 x86: 初始可执行的、位置无关线程局部变量的访问代码

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 8–11 32 位 x86: 初始可执行的、位置相关线程局部变量的访问代码

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，而不是使用 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。

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），如下面两个序列中所示。

表 8–12 32 位 x86: 初始可执行的、位置无关动态线程局部变量的访问代码

代码序列	初始重定位	符号
`0x00 movl  x@gotntpoff(%ebx), %eax`	R_386_TLS_GOTIE	x
`0x06 movl  %gs:(%eax), %eax`	<none>	
`# %eax - contains address of TLS variable`		
	未完成的重定位	**符号**
GOT[n]	R_386_TLS_TPOFF	x

表 8–13 32 位 x86: 初始可执行的、位置无关线程局部变量的访问代码

代码序列	初始重定位	符号
`0x00 movl  x@indntpoff, %ecx`	R_386_TLS_IE	x
`0x06 movl  %gs:(%ecx), %eax`	<none>	
`# %eax - contains address of TLS variable`		
	未完成的重定位	**符号**
GOT[n]	R_386_TLS_TPOFF	x

在最后一个序列中，如果使用的是 `%eax` 寄存器而不是 `%ecx` 寄存器，则第一个指令的长度可为 5 或 6 字节。

32 位 x86: 局部可执行 (Local Executable, LE)

此代码序列实现第 318 页中的 “线程局部存储的访问模型” 中介绍的 LE 模型。

表 8–14 32 位 x86: 局部可执行的线程局部变量的访问代码

代码序列	初始重定位	符号

表 8–14 32 位 x86: 局部可执行的线程局部变量的访问代码 （续）

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，借助相同重定位可以访问变量 x 的内容而不是地址。

表 8–15 32 位 x86: 局部可执行的线程局部变量的访问代码

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		

可以使用以下序列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 8–16 32 位 x86: 局部可执行的线程局部变量的访问代码

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。此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)
: 可以按处理引用 ___tls_get_addr() 函数的 R_386_PLT32 重定位的方式处理此重定位。

@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()。将 TLS_index 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。

@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 8–17 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)

表 8–17 32 位 x86: 线程局部存储的重定位类型 （续）

名称	值	字段	计算
R_386_TLS_DTPOFF32	36	Word32	@dtpoff(S)

x64: 访问线程局部变量

在 x64 上，可使用以下代码序列模型来访问 TLS。

x64: 常规动态 (General Dynamic, GD)

此代码序列实现第 318 页中的 “线程局部存储的访问模型”中介绍的 GD 模型。

表 8–18 x64: 常规动态线程局部变量的访问代码

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 0x666	<none>	
0x0a rex64	<none>	
0x0b call __tls_get_addr@plt	R_AMD64_PLT32	__tls_get_addr
# %iax - 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（GOT[n] 和 GOT[n+1]）中。这些 GOT 项与 R_AMD64_DTPMOD64 和 R_AMD64_DTPOFF64 重定位关联。

地址 0x00 处的指令计算第一个 GOT 项的地址。此计算将 GOT 起始位置的 PC 相对地址（在链接编辑时确定）与当前指令指针相加。使用 %rdi 寄存器将结果传递给 __tls_get_addr() 函数。

注 – leaq 指令计算第一个 GOT 项的地址。将 GOT 的 PC 相对地址（在链接编辑时确定）与当前指令指针相加来执行此计算。.byte、.word 和 .rex64 前缀确保整个指令序列占用 16 字节。由于这些前缀不会对代码造成负面影响，因此可以使用这些前缀。

---

x64: 局部动态 (Local Dynamic, LD)

此代码序列实现第 318 页中的 "线程局部存储的访问模型"中介绍的 LD 模型。

表 8–19 x64: 局部动态线程局部变量的访问代码

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
`# %rcx - 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。此索引是指包含偏移为零的 x1 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。

因为各个偏移会单独装入，所以不需要 R_AMD64_DTOFF32 重定位。x1@dtpoff 表达式用于访问符号 x1 的偏移。将指令用作地址 0x10 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 指令创建相同的重定位。

```
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。访问其他每个线程局部变量只需要三个新指令。此外，不需要其他 GOT 项或运行时重定位。

x64: 初始可执行 (Initial Executable, IE)

此代码序列实现第 318 页中的 “线程局部存储的访问模型”中介绍的 IE 模型。

表 8–20 x64: 初始可执行的线程局部变量的访问代码

代码序列	初始重定位	符号
`0x00 movq  %fs:0, %rax`	`<none>`	
`0x09 addq  x@gottpoff(%rip), %rax`	R_AMD64_GOTTPOFF	x
`# %rax - contains address of TLS variable`		
	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 重定位使用根据目前所装入模块确定的符号 x 的值。偏移将写入到 GOT 项中，然后由 addq 指令装入。

要装入 x 的内容（而不是 x 的地址），可使用以下序列。

表 8–21 x64: 初始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

表 8–21 x64: 初始可执行的线程局部变量的访问代码 II （续）

0x00 movq x@gottpoff(%rip), %rax	R_AMD64_GOTTPOFF	x
0x06 movq %fs:(%rax), %rax	<none>	
# %rax - contains contents of TLS variable		
	未完成的重定位	**符号**
GOT[n]	R_AMD64_TPOFF64	x

x64: 局部可执行 (Local Executable, LE)

此代码序列实现第 318 页中的 “线程局部存储的访问模型”中介绍的 LE 模型。

表 8–22 x64: 局部可执行的线程局部变量的访问代码

代码序列	初始重定位	符号
0x00 movq %fs:0, %rax	<none>	x
0x06 leaq x@tpoff(%rax), %rax	R_AMD64_TPOFF32	
# %rax - contains address of TLS variable		

要装入 TLS 变量的内容（而不是 TLS 变量的地址）， 可使用以下序列。

表 8–23 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	<none>	x
0x06 movq x@tpoff(%rax), %rax	R_AMD64_TPOFF32	
# %rax - contains contents of TLS variable		

以下序列更短。

表 8–24 x64: 局部可执行的线程局部变量的访问代码 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。此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()`。只能在确切的常规动态代码序列中使用此指令。

`@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，将目标文件的 ti_offset 偏移字段设置为零，并初始化 ti_module 偏移。对 `__tls_get_addr()` 函数的调用将返回动态 TLS 块的起始偏移。只能在确切的代码序列中使用此指令。

`@dtpoff`
计算变量相对于包含它的 TLS 块起始位置的偏移。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。不创建任何 GOT 项。

表 8–25 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)

表 8–25 x64: 线程局部存储的重定位类型 （续）

名称	值	字段	计算
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)

第 9 章

Mapfile 选项

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。通过结合使用 -M 选项和关联的 mapfile，可以更改链接编辑器提供的缺省映射。此外，还可以使用 mapfile 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，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 mapfiles 样例驻留在 /usr/lib/ld 目录中。

Mapfile 结构和语法

可以将以下基本类型的指令输入 mapfile：

- 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，然后定义节成为段的一部分所依据的标准。如果未先声明要映射到的段（内置段除外），便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，以下约定都适用：

- 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。
- 以**斜体**表示的所有项都是可替换的。
- { ... }* 表示“零个或多个”。
- { ... }+ 表示“一个或多个”。
- [...] 表示“可选”。
- `section_names` 和 `segment_names` 与 C 标识符遵守相同的规则，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可以出现空格的位置。

段声明

段声明可在输出文件中创建新段，或者更改现有段的属性值。现有段可以是指先前定义的段，也可以是随后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`，都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 9–1 Mapfile 段属性

属性	值
`segment_type`	`LOAD | NOTE | 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 7–5。最简单的 bss 声明：

```
bss =;
```

便足以创建 bss 段。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。请参见第 349 页中的 “Mapfile 缺省选项”。

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 或 STACK。如果未指定值，则缺省为 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 (?) 和构成 segment_flags 值的各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 段指定任何 segment_type 以外的段属性值。
- 允许有一个值为 STACK 的 segment_type。只能指定从 segment_flags 选择的段访问要求。
- 隐式声明的段缺省为：segment_type 值为 LOAD，segment_flags 值为 RWX，缺省的 virtual_address、physical_address 和 alignment 值，并且没有 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、不可写和不可执行。

- alignment 值用于计算段开头的虚拟地址。此对齐仅影响指定了对齐的段。其他段仍使用缺省对齐值，除非更改了它们的 alignment。
- 如果未设置任何 virtual_address、physical_address 或 length 属性值，链接编辑器便会在创建输出文件时计算这些值。
- 如果没有为段指定 alignment 值，则将此段设置为内置缺省段。此缺省段随 CPU 的不同而不同，甚至也可能随软件修订版的不同而不同。
- 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 字段包含缺省对齐值。
- 如果为段设置了 rounding 值，则将此段的虚拟地址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，则不执行任何舍入操作。

注 – 如果指定了 virtual_address 值，则将段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。

可以使用 ?E 标志创建空段。此空段没有关联的节。此段只能为可执行文件指定，并且其类型必须为具有指定大小和对齐值的 LOAD。允许具有多个此类型的段定义。

可以使用 ?N 标志控制是否将 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。可以使用 ?N 选项从第一个段的第一个节开始计算映像的虚拟地址。

可以使用 ?O 标志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，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() 位于文件 a.o 和 b.o 中，并且要将文件 a.o 中的函数 bar() 放置在文件 b.o 中的函数 bar() 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项：

```
text: .text%bar: a.o b.o;
```

但此项不能保证将文件 a.o 中的函数 bar() 放置在 b.o 中的函数 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。某个节为映射到特定段而必须具备的 section_attribute_values 集称为此段的**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，其中每个值都由空格进行分隔。每个节属性最多允许具有一个节属性值。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 9–2 节属性

节属性	值
section_name	任何有效的节名

表 9–2 节属性 （续）

节属性	值
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_type 中选择一个 section_type。上面列出的 section_type 是内置类型。有关 section_type 的更多信息，请参见第 221 页中的 “节”。
- 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。问号、叹号和构成 section_flags 值的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，则链接编辑器会模拟从命令行对文件执行 basename(1)，并将结果与指定的 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进行匹配。请参见第 347 页中的 “映射示例”。
- 如果在链接编辑期间使用 -l 选项，并且 -l 选项之后的库位于当前目录中，则必须在 mapfile 中将 ./ 或整个路径名作为库的前缀，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;
  ```

  为一个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，使用在带有入口条件的 mapfile 中遇到的第一个段。例如，如果 mapfile 显示为：

  ```
  S1 : $PROGBITS;

  S2 : $PROGBITS;
  ```

则将 `$PROGBITS` 节映射到 `S1` 段。

段内节的排序

使用以下表示法可以指定段中放置节的顺序：

```
segment_name | section_name1;

segment_name | section_name2;

segment_name | section_name3;
```

将以上述形式命名的节按照它们在 `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，以表示指定段的大小（以字节为单位）。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，可以指定共享库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

以下示例是用户定义的 `mapfile`。示例中左边的号码用于教学演示。实际上只有号码右边的信息会出现在 `mapfile` 中。

示例 9–1 用户定义的 Mapfile

```
1.  elephant : .data : peanuts.o *popcorn.o;

2.  monkey : $PROGBITS ?AX;
```

示例 9–1 用户定义的 Mapfile （续）

```
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。隐式声明的段 elephant（第 1 行）从文件 peanuts.o 和 popcorn.o 中接收所有 .data 节。请注意，*popcorn.o 与可以提供给链接编辑的任何 popcorn.o 文件相匹配。该文件无需位于当前目录中。另一方面，如果将 /var/tmp/peanuts.o 提供给链接编辑，则不会与 peanuts.o 相匹配，因为它没有前缀 *。

隐式声明的段 monkey（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段 elephant 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，相同行中的属性之间存在“与”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”关系。

monkey 段在第 2 行中隐式声明为：segment_type 值为 LOAD，segment_flags 值为 RWX，未指定 virtual_address、physical_address、length 或 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 并且最大 length 值设置为 0x4000。

第 5 行隐式声明 donkey 段。入口条件指定为将所有 .data 节路由到此段。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 并且 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 data 段并指定其虚拟地址必须从 0x1000 字节边界开始。构成 data 段的第一个节还驻留在文件映像内的 0x1000 字节边界上。

Mapfile 缺省选项

链接编辑器使用缺省的 segment_attribute_values 和相应的缺省映射指令定义四个内置段（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的行为方式就好像已经读入了 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 中尚未存在此段，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 中没有一个段与刚读入的段的 segment_type 值相同，则按 segment_type 值添加段以保持以下顺序：

   INTERP

   LOAD

   DYNAMIC

   NOTE
3. 如果段的 segment_type 值为 LOAD，并且已经为此可装入 (LOAD) 的段定义了 virtual_address 值，则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，但要放置在任何 virtual_address 值较小的段之后。

读入 mapfile 中的每个映射指令时，将该指令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 9–1 展示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。循环显示节本身。

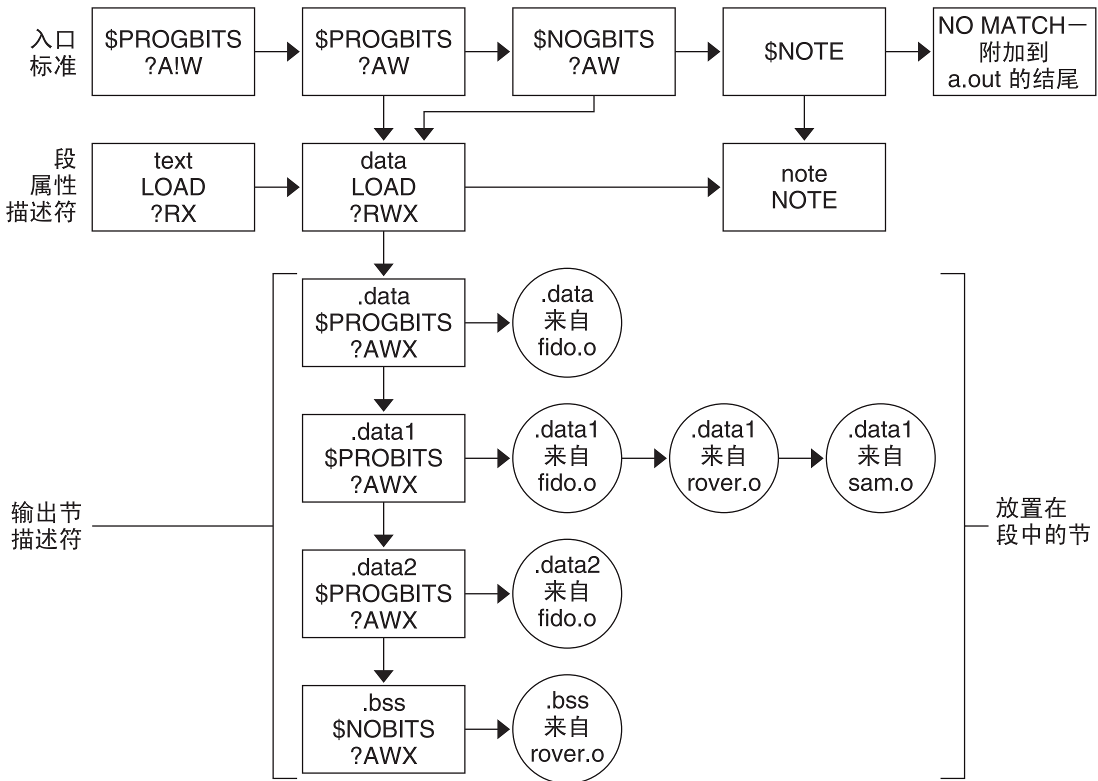


图 9–1 简单的映射结构

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匹配。必须与所有指定的条件相匹配。

   在图 9–1 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_flags` 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 $PROGBITS、section_flags 值为 ?AWX 的节将进入图 9–1 中第二个“入口条件”框，可以将其放置在 data 段中。此节与第二个“输出节描述符”框（.data1、$PROGBITS、?AWX）完全匹配，并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相同节类型的最后一个输出节描述符之后。图 9–1 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，则创建新的输出节描述符，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。在 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范（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，则不会为此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。

附录 A

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，请参见第 22 页中的 “链接编辑” 。

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，将会使用 cc(1)。 请参见第 28 页中的 “使用编译器驱动程序” 。

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。 请参见第 30 页中的 “输入文件处理” 。

可以使用 `-l` 选项输入遵循命名约定 lib*x*.so 的共享库，以及遵循命名约定 lib*x*.a 的归档库。 请参见第 32 页中的 “库命名约定” 。这为允许使用 `-L` 选项来指定搜索路径提供了更大的灵活性。 请参见第 33 页中的 “链接编辑器搜索的目录” 。

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` 选项时会选定静态模式，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，可将 `-d n` 和 `-r` 选项一使用：

```
$ cc -dn -r -o temp.o file1.o file2.o file3.o .....
```

创建静态可执行文件

静态可执行文件的使用将受到限制。请参见第 22 页中的 “静态可执行文件” 。静态可执行文件通常包含特定于平台实现的细节，这会限制可执行文件在备用平台上运行的能力。许多 Solaris 库的实现都取决于动态链接功能，如 `dlopen(3C)` 和 `dlsym(3C)`。请参见第 86 页中的 “装入其他目标文件” 。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操作的缺省模式。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，则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、共享库和归档库。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库。如果未找到任何共享库，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。请参见第 32 页中的 “同时链接共享库和归档” 。

创建共享库

- 要创建共享库，请使用 `-G` 选项。由于缺省情况下会执行 `-d y` 选项，故此选项是可选的。
- 输入可重定位目标文件应该通过与位置无关的代码生成。例如，C 编译器使用 `-K pic` 选项生成与位置无关的代码。请参见第 137 页中的 “与位置无关的代码” 。使用 `-z text` 选项可以强制实施此要求。
- 请避免包括未使用的可重定位目标文件。或者，请使用 `-z ignore` 选项，此选项可指示链接编辑器删除未引用的 ELF 节。请参见第 140 页中的 “删除未使用的材料” 。
- 如果共享库旨在供外部使用，请确保其不使用任何应用程序寄存器。如果不使用应用程序寄存器，则外部用户可自由使用这些寄存器，而不必担心会危及共享库的实现。例如，SPARC C 编译器在使用 `-xregs=no%appl` 选项时不使用应用程序寄存器。
- 请通过定义应从共享库可见的全局符号并将其他任何全局符号限制到局部范围来建立共享库公共接口。该定义由 `-M` 选项与关联的 `mapfile` 共同提供。请参见附录 B。
- 请针对共享库使用版本化名称以便将来可以升级。请参见第 175 页中的 “协调版本化文件名” 。

- 独立的共享库可提供最大的灵活性。目标文件表示所有依赖性需要时会生成独立的共享库。使用 -z defs 可强制实现这种独立。 请参见第 47 页中的 “生成共享库输出文件” 。
- 请避免使用不需要的依赖项。请使用带有 -u 选项的 ldd 来检测并删除不需要的依赖项。 请参见第 31 页中的 “共享库处理” 。或者，请使用 -z ignore 选项，此选项可指示链接编辑器将依赖项仅记录到所引用的目标文件中。
- 如果要生成的共享库依赖于其他共享库，则表明应该使用 -z lazyload 选项以延迟方式装入这些依赖项。 请参见第 87 页中的 “延迟装入动态依赖项” 。
- 如果要生成的共享库依赖于其他共享库，并且这些依赖项不是位于缺省的搜索位置中，请使用 -R 选项将其路径名记录在输出文件中。 请参见第 123 页中的 “具有依赖项的共享库” 。
- 请通过将可重定位的各节合并成单独一个 .SUNW_reloc 节来优化可重定位处理。请使用 -z combreloc 选项。
- 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 82 页中的 “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 combreloc -z ignore -R /home/lib foo.o -L. -lbar -lc
```

- 如果要生成的共享库用作其他链接编辑的输入，请使用 -h 选项在其中记录共享库的运行时名称。 请参见第 120 页中的 “记录共享库名称” 。
- 请通过创建指向非版本化共享库名称的文件系统链接，使共享库可用于编译环境中。 请参见第 175 页中的 “协调版本化文件名” 。

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 combreloc -z ignore -R /home/lib -h libfoo.so.1 foo.o -L. -lbar -lc

$ ln -s libfoo.so.1 libfoo.so
```

- 请考虑共享库的性能含义，即最大化共享性，如第 140 页中的 “最大化可共享性”中所述：最小化换页活动，如第 142 页中的 “最小化换页活动”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，如第 57 页中的 “缩减符号范围”中所述：允许通过功能接口访问数据，如第 144 页中的 “复制重定位”中所述。

创建动态可执行文件

- 要创建动态可执行文件，请不要使用 -G 或 -d n 选项。
- 表明应使用 -z lazyload 选项以延迟方式装入动态可执行文件的依赖项。 请参见第 87 页中的 “延迟装入动态依赖项” 。
- 请避免使用不需要的依赖项。请使用带有 -u 选项的 ldd 来检测并删除不需要的依赖项。 请参见第 31 页中的 “共享库处理” 。或者，请使用 -z ignore 选项，此选项可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是位于缺省搜索位置中，请使用 -R 选项将其路径名记录在输出文件中。 请参见第 35 页中的 “运行时链接程序搜索的目录” 。
- 请使用 -B direct 来建立直接绑定信息。 请参见第 82 页中的 “直接绑定” 。

以下示例结合了以上几点：

```
$ cc -o prog -R /home/lib -z ignore -z lazyload -B direct -L. \

-lfoo file1.o file2.o file3.o .....
```

附录 B

版本控制快速参考

ELF 目标文件可使得全局符号可用，这样，其他目标文件可绑定到这些全局符号。可以将其中某些全局符号标识为提供目标文件的**公共接口**。其他符号是目标文件内部实现的一部分，不能在外部使用。目标文件接口可以随着软件发行版的发展而升级，最好是具有标识此发展的功能。

此外，还可能需要标识目标文件随软件发行版的发展而发生的**内部实现**更改。

可以通过建立内部**版本定义**在目标文件内记录接口和实现标识。有关内部版本控制概念的更完整介绍，请参见第 5 章。

共享库是内部版本控制将使用的主要目标文件。此技术定义目标文件的发展过程，在运行时处理过程中提供接口验证（请参见第 163 页中的 “绑定到版本定义”），同时还提供可选用的应用程序绑定（请参见第 168 页中的 “指定版本绑定”）。本附录中多处使用共享库作为示例。

以下各节简要概述了链接编辑器提供的应用于共享库的内部版本控制机制，可以将这些概述称为**备忘单**。其中的示例为共享库的版本控制提供了一些建议的约定和机制（从初始构造到多个常见的更新方案）。

命名约定

共享库遵循的命名约定包括一个**主**编号文件后缀。请参见第 119 页中的 “命名约定”。在这个共享库中，可以创建一个或多个**版本定义**。每个版本定义都对应于以下类别之一：

- 定义行业标准接口（例如，*System V* **应用程序二进制接口**）。
- 定义特定于供应商的公共接口。
- 定义特定于供应商的专用接口。
- 定义特定于供应商的目标文件内部实现的更改。

以下版本定义命名约定有助于指明定义代表上述哪个类别。

这些类别中的前三个属于接口定义。这些定义由构成接口的全局符号名称与版本定义名称关联组成。请参见第 155 页中的 “创建版本定义” 。共享库内的接口更改通常称为次修订。因此，此类版本定义带有一个次要版本号后缀，次版本号基于文件名的主版本号后缀。

最后一个类别指明目标文件内发生了更改。此定义由充当标号的版本定义组成，没有与之关联的符号名称。因此将其称为弱版本定义 (weak version definition)。请参见第 159 页中的 “创建弱版本定义 (weak version definition)” 。共享库内的实现更改通常称为微修订。因此，此类版本定义带有一个微版本号后缀，微版本号基于应用内部更改的上一个次版本号。

任何行业标准的接口都应使用能够反映此标准的版本定义名称。任何供应商接口都应使用供应商独有的版本定义名称。通常使用公司的股票代码号。

专用版本定义指明限制使用或不供使用的符号，而且 “专用” 一词应清晰可见。

所有版本定义都会创建关联的版本符号名称。使用唯一名称和次/微后缀约定可减少在正在生成的目标文件中出现符号冲突的几率。

以下版本定义示例说明了这些命名约定的可能用途：

`SVABI.1`
: 定义 *System V* **应用程序二进制接口**标准接口。

`SUNW_1.1`
: 定义 Solaris 公共接口。

`SUNWprivate_1.1`
: 定义 Solaris 专用接口。

`SUNW_1.1.1`
: 定义 Solaris 内部实现更改。

定义共享库的接口

建立共享库的接口时，应首先确定此共享库提供的哪些全局符号能够与三个接口版本定义类别之一关联：

- 行业标准接口符号通常在供应商提供的公用头文件和关联的手册页中进行定义，同时记录在已发布的标准文献中。
- 供应商公共接口符号通常在供应商提供的公用头文件和关联的手册页中进行定义。
- 供应商专用接口符号可以包含少量公共定义或不包含公共定义。

通过定义这些接口，供应商可指明此共享库的每个接口的承诺级别。行业标准接口和供应商公共接口在各个发行版中几乎保持不变。如果确定应用程序在不同的发行版中都能继续正常运作，则可安全地随意绑定到这些接口。

其他供应商提供的系统上也可能存在行业标准接口。通过将应用程序限制为使用这些接口，可以实现更高级别的二进制兼容性。

其他供应商提供的系统上也可能不存在供应商公共接口。但是，当这些接口所在的系统发展时，它们会保持不变。

供应商专用接口非常不稳定，并且在不同发行版中会不同甚至会删除。这些接口用于实现未确定用途的功能或实验功能，或者是仅用于访问特定于供应商的应用程序。如果要实现某个级别的二进制兼容性，则应避免使用这些接口。

任何不属于上述任一类别的全局符号都应该缩减到本地范围，以使其不再可见，以免进行绑定。请参见第 57 页中的 “缩减符号范围” 。

共享库的版本控制

确定共享库的可用接口后，可以使用 mapfile 和链接编辑器的 -M 选项创建关联的版本定义。有关此 mapfile 语法的介绍，请参见第 49 页中的 “定义其他符号” 。

以下示例定义了共享库 libfoo.so.1 中的供应商公共接口：

```
$ cat mapfile
SUNW_1.1 {                  # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
$ cc -G -o libfoo.so.1 -h libfoo.so.1 -z text -M mapfile foo.c
```

全局符号 foo1 和 foo2 指定给共享库的公共接口 SUNW_1.1。输入文件中提供的任何其他全局符号均通过自动缩减指令 “*” 缩减到本地。请参见第 57 页中的 “缩减符号范围” 。

注 – 每个版本定义的 `mapfile` 项均应带有一个注释，用于反映更新的发行版或日期。可利用此信息将多个共享库更新整合（可能由不同的开发者进行）到一个版本定义中，从而使共享库的发布作为软件发行的一部分进行。

现有（非版本化）共享库的版本控制

对现有的非版本化共享库进行版本控制需要格外小心。上一个软件发行版中提供的共享库已使其所有全局符号可供其他目标文件绑定。尽管可以确定共享库的目标接口，但其他目标文件可能已发现其他符号并绑定到这些符号。因此，删除任何符号都可能会导致应用程序无法传送新的版本化共享库。

如果可以确定并应用接口，而不中断任何现有应用程序，则可以对现有的非版本化共享库进行内部版本控制。运行时链接程序的调试功能对验证各种应用程序的绑定要求非常有用。 请参见第 109 页中的 “调试库” 。但是，确定现有绑定要求会假定共享库的所有用户都是已知的。

如果无法确定现有的非版本化共享库的绑定要求，则应使用新的版本化名称创建一个新的共享库文件。 请参见第 175 页中的 “协调版本化文件名” 。除了这个新的共享库外，还必须传送原始共享库，以满足任何现有应用程序的依赖性。

如果要冻结原始共享库的实现，则只需维护和传送共享库二进制文件。但是，如果原始共享库需要更新，则更适合使用从中生成共享库的备用源树。修补程序升级可能需要更新，只有升级共享库的实现才能与新平台保持兼容时也可能要更新。

更新版本化共享库

能够对可执行内部版本控制的共享库进行的唯一更改就是兼容更改。 请参见第 154 页中的 “接口兼容性” 。任何不兼容的更改都要求使用新的外部版本化名称生成一个新的共享库。 请参见第 175 页中的 “协调版本化文件名” 。

通过内部版本控制便可进行的兼容更新分为以下三种基本类别 :

- 添加新符号
- 从现有符号创建新接口
- 内部实现更改

前两个类别可以通过将接口版本定义与相应符号关联来实现。后一个类别可以通过创建不具有任何关联的弱版本定义 (weak version definition) 来实现。

添加新符号

任何包含新全局符号的新的共享库兼容发行版都应将这些符号指定给新的版本定义。此新版本定义应继承上一个版本定义。

以下 mapfile 示例将新符号 foo3 指定给新接口版本定义 SUNW_1.2。此新接口继承原始接口 SUNW_1.1。

```
$ cat mapfile

SUNW_1.2 {                  # Release X+1.

        global:

                foo3;

} SUNW_1.1;



SUNW_1.1 {                  # Release X.

        global:

                foo2;

                foo1;

        local:

                *;

};
```

版本定义的继承可减少所有共享库用户必须记录的版本信息量。

内部实现更改

任何包含目标文件实现更新（例如错误修复或性能改进）的新的共享库兼容发行版都应带有一个**弱**版本定义。此新版本定义应继承发生更新时所用的最新版本定义。

以下 mapfile 示例生成弱版本定义 (weak version definition) SUNW_1.1.1。此新接口指明对上一个接口 SUNW_1.1 提供的实现进行了内部更改。

```
$ cat mapfile

SUNW_1.1.1 { } SUNW_1.1;     # Release X+1.



SUNW_1.1 {                  # Release X.
```

```
        global:
                foo2;
                foo1;
        local:
                *;
};
```

新符号和内部实现更改

如果在同一个发行版中发生了内部更改并添加了新接口，则应创建弱版本定义 (weak version definition) 和接口版本定义。以下示例显示了添加版本定义 SUNW_1.2 和接口更改 SUNW_1.1.1 的情况，它们是在同一个发行周期内添加的。两个接口都继承原始接口 SUNW_1.1。

```
$ cat mapfile
SUNW_1.2 {                 # Release X+1.
        global:
                foo3;
} SUNW_1.1;


SUNW_1.1.1 { } SUNW_1.1;     # Release X+1.


SUNW_1.1 {                 # Release X.
        global:
                foo2;
                foo1;
        local:
```

```
                *;
};
```

注 – SUNW_1.1 和 SUNW_1.1.1 版本定义的注释指明它们应用于同一个发行版

将符号迁移到标准接口

有时，新的行业标准中会采用供应商接口提供的符号。创建新标准接口时，请务必维护共享库提供的原始接口定义。创建一些中间版本定义，以便根据它们生成新标准定义和原始接口定义。

以下 mapfile 示例显示了添加新的行业标准接口 STAND.1。此接口包含新符号 foo4 以及现有符号 foo3 和 foo1，它们最初分别通过接口 SUNW_1.2 和 SUNW_1.1 提供。

```
$ cat mapfile
STAND.1 {                    # Release X+2.
        global:
                foo4;
} STAND.0.1 STAND.0.2;


SUNW_1.2 {                   # Release X+1.
        global:
                SUNW_1.2;
} STAND.0.1 SUNW_1.1;


SUNW_1.1.1 { } SUNW_1.1;     # Release X+1.


SUNW_1.1 {                   # Release X.
        global:
```

```
                foo2;
        local:
                *;
} STAND.0.2;
                             # Subversion - providing for
STAND.0.1 {                  # SUNW_1.2 and STAND.1 interfaces.
        global:
                foo3;
};
                             # Subversion - providing for
STAND.0.2 {                  # SUNW_1.1 and STAND.1 interfaces.
        global:
                foo1;
};
```

符号 foo3 和 foo1 被引入各自的中间接口定义（用于创建原始接口定义和新接口定义）中。

SUNW_1.2 接口的新定义引用了自身的版本定义符号。如果没有此引用，SUNW_1.2 接口将不包含任何即时符号引用，从而将归类为弱版本定义 (weak version definition)。

将符号定义迁移到标准接口中时，任何原始接口定义都必须继续表示相同符号列表。可以使用 pvs(1) 验证此要求。以下示例显示了软件发行版 X+1 中存在的 SUNW_1.2 接口的符号列表。

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1
        SUNW_1.2:
                foo3;
        SUNW_1.1:
                foo2;
                foo1;
```

尽管新标准接口在软件发行版 X+2 中的引入更改了可用的接口版本定义，但每个原始接口提供的符号列表均保持不变。以下示例显示了接口 SUNW_1.2 仍然提供符号 foo1、foo2 和 foo3。

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1

        SUNW_1.2:

        STAND.0.1:

                foo3;

        SUNW_1.1:

                foo2;

        STAND.0.2:

                foo1;
```

应用程序可能只引用新的子版本中的一个。在这种情况下，尝试在上一个发行版中运行此应用程序将导致运行时版本控制错误。 请参见第 163 页中的 “绑定到版本定义” 。

通过直接引用现有的版本名称，可以提升应用程序的版本绑定。 请参见第 170 页中的 “到其他版本定义的绑定” 。例如，如果一个应用程序只引用共享库 libfoo.so.1 的符号 foo1，则其版本引用为 STAND.0.2。要使此应用程序能够在以前的发行版上运行，可以使用版本控制指令 mapfile 将版本绑定提升到 SUNW_1.1。

```
$ cat prog.c

extern void foo1();



main()

{

        foo1();

}

$ cc -o prog prog.c -L. -R. -lfoo

$ pvs -r prog

        libfoo.so.1 (STAND.0.2);
```

```
$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.1;
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.1);
```

实际上，很少需要按这种方式提升版本绑定。因为很少会引入新的标准二进制接口，而且大多数应用程序都引用一个接口系列中的许多符号。

附录 C

使用动态字符串标记建立依赖性

动态库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性。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：

- $HWCAP
- $ISALIST
- $OSNAME、$OSREL 和 $PLATFORM
- $ORIGIN

以下各节提供了如何使用这些标记的示例。

特定于硬件功能的共享库

动态标记 $HWCAP 可用于指定特定于硬件功能的共享库所在的目录。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。

路径名称指定必须包含以 $HWCAP 标记结束的全路径名。由 $HWCAP 标记指定的目录中的共享库将在运行时受到检查。这些目标文件应指明其硬件功能要求。 请参见第 64 页中的“标识硬件和软件功能”。将根据可用于此进程的硬件功能验证每个目标文件。适用于进程的那些目标文件按其硬件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。

硬件功能目录中的 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/hwcap/$HWCAP' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
```

```
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"
  [1]    SONAME    libfoo.so.1
  [2]    AUXILIARY /opt/ISV/lib/hwcap/$HWCAP
$ elfdump -H /opt/ISV/lib/hwcap/*

/opt/ISV/lib/hwcap/filtee.so.3:

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x1000  [ SSE2 ]

/opt/ISV/lib/hwcap/filtee.so.1:

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x40  [ MMX ]

/opt/ISV/lib/hwcap/filtee.so.2:

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x800  [ SSE ]
```

如果在具有 MMX 和 SSE 功能的平台上处理过滤器 `libfoo.so.1`，则出现以下 filtee 搜索顺序。

```
$ cc -o prog prog.c -R. -lfoo

$ LD_DEBUG=symbols prog

.....

debug: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]

debug: symbol=foo;  lookup in file=hwcap/filtee.so.2  [ ELF ]

debug: symbol=foo;  lookup in file=hwcap/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 $HWCAP，可使一个或多个 filtee 实现过滤器内定义的接口。

指定的 $HWCAP 目录中的所有共享库都会被检查，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

.....

debug: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]

debug: symbol=foo;  lookup in file=hwcap/filtee.so.2  [ ELF ]

.....
```

特定于指令集的共享库

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。

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

$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"

  [1]    SONAME    libfoo.so.1

  [2]    AUXILIARY /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

$ dump -Lv libfoo.so.1 | egrep "RUNPATH|AUXILIARY"

  [1]    RUNPATH   /opt/ISV/lib/$ISALIST

  [2]    AUXILIARY libbar.so.1
```

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
```

特定于系统的共享库

将在运行时扩展动态标记 $OSNAME、$OSREL 和 $PLATFORM，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 以反映基础硬件实现，如 uname -i 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
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"
  [1]    SONAME    libfoo.so.1
  [2]    AUXILIARY /platform/$PLATFORM/lib/libbar.so.1
```

此机制在 Solaris 操作环境中用于提供特定于平台的共享库 /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、共享库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。

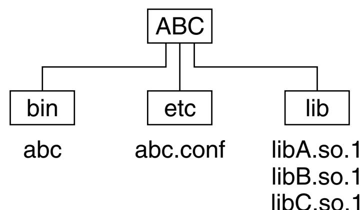


图 C–1 非绑定依赖项

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

% dump -Lv abc

  [1]    NEEDED  libA.so.1

  [2]    RUNPATH /opt/ABC/lib
```

类似地，对于依赖项 libA.so.1，它将显示为：

```
% cc -o libA.so.1 -G -Kpic A.c -R/opt/ABC/lib -L/opt/ABC/lib -lB

% dump -Lv libA.so.1

  [1]    NEEDED  libB.so.1

  [2]    RUNPATH /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

% dump -Lv abc
```

```
  [1]    NEEDED  libA.so.1
  [2]    RUNPATH $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
% dump -Lv libA.so.1
  [1]    NEEDED  libB.so.1
  [2]    RUNPATH $ORIGIN
```

如果此产品目前安装在 `/usr/local/ABC` 下，并在用户的 `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  (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  (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
```

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：

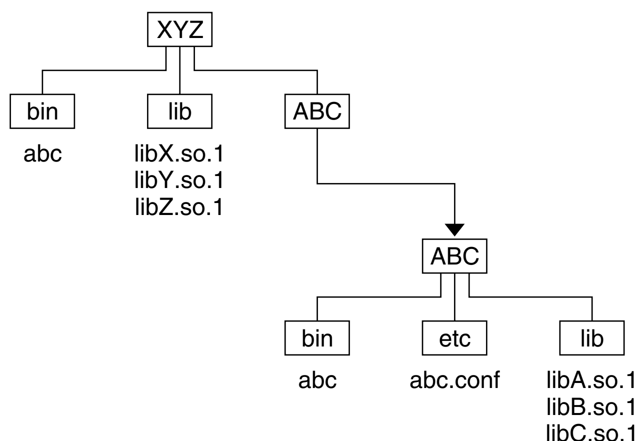


图 C–2 非绑定共同依赖性

XYZ 产品的二进制代码和共享库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
% dump -Lv xyz
  [1]    NEEDED  libX.so.1
  [2]    NEEDED  libA.so.1
  [3]    RUNPATH $ORIGIN/../lib:$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
% dump -Lv libX.so.1
  [1]    NEEDED  libY.so.1
  [2]    NEEDED  libC.so.1
  [3]    RUNPATH $ORIGIN:$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
```

如果在用户的 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  (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  (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 \
                (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 \

              (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  (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
```

安全

在安全的进程中，只有将 $ORIGIN 字符串扩展到受信任的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，可能会产生这种安全风险。

```
% 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。

附录 D

链接程序和库的更新及新增功能

本附录概述了已添加到 Solaris OS 各个发行版中的更新和新增功能。

Solaris 10 1/06 发行版

- 提供了对 x64 介质代码模型的支持。 请参见表 7–4、表 7–8 和表 7–10。
- 可以使用 dlinfo(3C) 标志 RTLD_DI_ARGSINFO 获取命令行参数、环境变量以及进程的辅助向量阵列。
- 通过链接编辑器 -B nodirect 选项，可以更加灵活地禁止对直接绑定进行外部引用。 请参见第 82 页中的 "直接绑定" 。

Solaris 10 发行版

- 目前支持 x64。请参见表 7–5、第 234 页中的 "特殊节" 、第 258 页中的 "x64: 重定位类型" 、第 334 页中的 "x64: 访问线程局部变量" 和第 338 页中的 "x64: 线程局部存储的重定位类型" 。
- 重新构造文件系统会将许多组件从 /usr/lib 下移动到 /lib 下。对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。 请参见第 33 页中的 "链接编辑器搜索的目录" 、第 76 页中的 "运行时链接程序搜索的目录" 和第 95 页中的 "安全性" 。
- 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。 请参见第 22 页中的 "静态可执行文件" 。
- 通过 crle(1) 的 -A 选项，可以更加灵活地定义替代依赖项。
- 现在，链接编辑器可处理未指定值的环境变量。 请参见第 24 页中的 "环境变量" 。
- 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。 请参见附录 C。通过新的实用程序 moe(1) 可提供对使用保留标记的路径名的评估。

- 通过 dlsym(3C) 和新句柄 RTLD_PROBE 可提供用于测试接口是否存在的最佳方法。 请参见第 89 页中的 “提供 dlopen() 的替代项” 。

Solaris 9 9/04 发行版

- 通过链接编辑器，可以更加灵活地定义 ELF 目标文件的硬件和软件要求。 请参见第 240 页中的 “硬件和软件功能节” 。
- 已添加了运行时链接审计接口 la_objfilter()。 请参见第 189 页中的 “审计接口函数” 。
- 扩展了共享库过滤功能，可逐个符号进行过滤。 请参见第 125 页中的 “作为过滤器的共享库” 。

Solaris 9 4/04 发行版

- 支持新的节类型 SHT_SUNW_ANNOTATE、SHT_SUNW_DEBUGSTR、SHT_SUNW_DEBUG 以及 SHT_SPARC_GOTDATA。 请参见表 7–5。
- 通过新的实用程序 lari(1) 简化了运行时接口的分析。
- 通过链接编辑器选项 -z direct 和 -z nodirect，以及 DIRECT 和 NODIRECT mapfile 指令，可以进一步控制直接绑定。请参见第 49 页中的 “定义其他符号”和第 82 页中的 “直接绑定” 。

Solaris 9 12/03 发行版

- ld(1) 内的性能改进可显著缩短大型应用程序的链接编辑时间。

Solaris 9 8/03 发行版

- 可以使用通过 RTLD_FIRST 标志创建的 dlopen(3C) 句柄来减少 dlsym(3C) 符号处理。 请参见第 104 页中的 “获取新符号” 。
- 可以使用 dlinfo(3C) 标志 RTLD_DI_GETSIGNAL 和 RTLD_DI_SETSIGNAL 来管理运行时链接程序用来终止错误进程的信号。

Solaris 9 12/02 发行版

- 链接编辑器提供字符串表压缩，从而可以生成简化的 .dynstr 节和 .strtab 节。可以使用链接编辑器的 -z nocompstrtab 选项禁用此缺省处理。 请参见第 63 页中的 “字符串表压缩” 。
- 扩展了 -z ignore 选项，可在链接编辑过程中删除未引用部分。 请参见第 140 页中的 “删除未使用的材料” 。
- 可以使用 ldd(1) 确定未引用的依赖项。请参见 -U 选项。
- 链接编辑器支持扩展的 ELF 节。请参见第 214 页中的 “ELF 头” 、表 7–5、第 221 页中的 “节” 、表 7–10 和第 260 页中的 “符号表节” 。
- 通过 protected mapfile 指令，可以更加灵活地定义符号可见性。 请参见第 49 页中的 “定义其他符号” 。

Solaris 9 发行版

- 提供了线程局部存储 (Thread-Local Storage, TLS) 支持。 请参见第 8 章。
- -z rescan 选项在指定归档库进行链接编辑方面提供了更大的灵活性。 请参见第 33 页中的 “命令行中归档的位置” 。
- -z ld32 和 -z ld64 选项在使用链接编辑器支持接口方面提供了更大的灵活性。 请参见第 180 页中的 “32 位环境和 64 位环境” 。
- 添加了其他链接编辑器支持接口 ld_input_done()、ld_input_section()、ld_input_section64() 和 ld_version()。 请参见第 180 页中的 “支持接口函数” 。
- 现在，可以通过在配置文件中指定由运行时链接程序解释的环境变量，为多个进程建立这些变量。请参见 crle(1) 的 -e 和 -E 选项。
- 添加了对 64 位 SPARC 目标文件中超过 32,768 个过程链接表项的支持。 请参见第 302 页中的 “64 位 SPARC: 过程链接表” 。
- 通过 mdb(1) 调试器模块，可以在进程调试操作过程中检查运行时链接程序数据结构。 请参见第 114 页中的 “调试器模块” 。
- 通过 bss 段的声明指令，可以更轻松地创建 bss 段。 请参见第 342 页中的 “段声明” 。

Solaris 8 07/01 发行版

- 可以使用 ldd(1) 确定未使用的依赖项。请参见 -u 选项。
- 添加了各种 ELF ABI 扩展格式。请参见第 36 页中的 “初始化和终止节” 、第 91 页中的 “初始化和终止例程” 、表 7–3、表 7–8、表 7–9、第 239 页中的 “组节” 、表 7–10、表 7–20、表 7–32、表 7–33 以及第 280 页中的 “程序装入（特定于处理器）” 。

- 通过添加 _32 和 _64 变体，可以更加灵活地使用链接编辑器环境变量。请参见第 24 页中的 “环境变量” 。

Solaris 8 01/01 发行版

- 通过引入 dladdr1()，增强了通过 dladdr(3C) 可用的符号信息。
- 可以从 dlinfo(3C) 获取动态库的 $ORIGIN。
- 简化了通过 crle(1) 创建的运行时配置文件的维护。配置文件的检查会显示用于创建此文件的命令行选项。提供了通过 -u 选项进行更新的功能。
- 扩展了运行时链接程序及其调试器接口，可检测过程链接表项的解析。此更新通过新版本号进行标识。请参见第 198 页中的 “代理处理接口” 下的 rd_init()。此更新会扩展 rd_plt_info_t 结构。请参见第 205 页中的 “跳过过程链接表” 下的 rd_plt_resolution()。
- 可以使用新的 mapfile 段描述符 STACK 将应用程序的栈定义为不可执行。请参见第 342 页中的 “段声明” 。

Solaris 8 10/00 发行版

- 运行时链接程序会忽略环境变量 LD_BREADTH。请参见第 91 页中的 “初始化和终止例程” 。
- 扩展了运行时链接程序及其调试器接口，从而可更好地进行运行时和核心转储文件分析。此更新通过新版本号进行标识。请参见第 198 页中的 “代理处理接口” 下的 rd_init()。此更新会扩展 rd_loadobj_t 结构。请参见第 200 页中的 “扫描可装入目标文件” 。
- 现在，可以通过副本重定位来验证关于位移重定位数据的使用或可能的使用。请参见第 68 页中的 “位移重定位” 。
- 通过使用链接编辑器的 -64 选项可以仅从 mapfile 生成 64 位过滤器。请参见第 126 页中的 “生成标准过滤器” 。
- 可以使用 dlinfo(3C) 检查用于定位动态库的依赖项的搜索路径。
- 通过新句柄 RTLD_SELF 扩展了 dlsym(3C) 和 dlinfo(3C) 查找语义。
- 可以通过在每个动态库中建立直接绑定信息，显著减少用于重定位动态库的运行时符号查找机制。请参见第 82 页中的 “直接绑定” 。

Solaris 8 发行版

- 现在，可以预装入文件的安全目录为 /usr/lib/secure（针对 32 位目标文件）和 /usr/lib/secure/64（针对 64 位目标文件）。请参见第 95 页中的 "安全性"。
- 通过链接编辑器的 -z nodefaultlib 选项以及新实用程序 crle(1) 创建的运行时配置文件，可以更加灵活地修改运行时链接程序的搜索路径。请参见第 35 页中的 "运行时链接程序搜索的目录" 和第 79 页中的 "配置缺省搜索路径"。
- 通过新的 EXTERN mapfile 指令，可以将 -z defs 用于外部定义的符号。请参见第 49 页中的 "定义其他符号"。
- 新的 $ISALIST、$OSNAME 以及 $OSREL 动态字符串标记在建立特定于指令集和系统的依赖项方面提供了更大的灵活性。请参见第 79 页中的 "动态字符串标记"。
- 链接编辑器选项 -p 和 -P 提供了调用运行时链接审计库的其他方法。请参见第 188 页中的 "记录局部审计程序"。添加了运行时链接审计接口 la_activity() 和 la_objsearch()。请参见第 189 页中的 "审计接口函数"。
- 通过新的动态节标记 DT_CHECKSUM，可以协调 ELF 文件与核心映像。请参见表 7–32。

Solaris 7 发行版

- 目前支持 64 位的 ELF 目标文件格式。有关详细信息，请参见第 211 页中的 "文件格式"。链接编辑器针对 64 位处理的扩展和差异包括使用 /usr/lib/64（请参见第 33 页中的 "链接编辑器搜索的目录"、第 35 页中的 "运行时链接程序搜索的目录" 以及第 119 页中的 "命名约定"）、环境变量 LD_LIBRARY_PATH_64（请参见第 34 页中的 "使用环境变量" 和第 76 页中的 "运行时链接程序搜索的目录"）以及运行时链接程序 /usr/lib/64/ld.so.1（请参见第 3 章）。
- 可以使用链接编辑器的 -z combreloc 选项生成包含优化重定位节的共享库。请参见第 143 页中的 "组合重定位节"。
- 新的 $ORIGIN 动态字符串标记在非绑定软件中建立依赖项方面提供了更大的灵活性。请参见第 79 页中的 "动态字符串标记"。
- 现在，可以延迟共享库的装入，直到运行的程序实际引用了此目标文件为止。请参见第 137 页中的 "延迟装入动态依赖项"。
- 通过新的 SHT_SUNW_COMDAT 节类型，可删除多重定义的符号。请参见第 238 页中的 "COMDAT 节"。
- 新的 SHT_SUNW_move 节类型会启用部分初始化的符号。请参见第 243 页中的 "移动节"。
- 添加了运行时链接审计接口 la_symbind64()、la_sparcv9_pltenter()、la_pltexit64() 以及新的链接审计标志 LA_SYMB_ALTVALUE。请参见第 189 页中的 "审计接口函数"。

Solaris 2.6 发行版

- 弱符号引用可以使用链接编辑器的 -z weakextract 选项触发归档成员提取。使用 -z allextract 选项可以提取所有归档成员。 请参见第 30 页中的 “归档处理” 。
- 可以使用链接编辑器的 -z ignore 选项，忽略所生成的目标文件未引用的在链接编辑过程中指定的共享库。 请参见第 31 页中的 “共享库处理” 。
- 链接编辑器可生成保留符号 _START_ 和 _END_ 来提供建立目标文件地址范围的方法。 请参见第 63 页中的 “生成输出文件” 。
- 对初始化和结束代码的运行时排序进行了更改，以更好地适应依赖性要求。 请参见第 91 页中的 “初始化和终止例程” 。
- 针对 dlopen(3C) 扩展了符号解析语义。请参见第 99 页中的 “符号查找” 、第 103 页中的 “隔离组” 中的 RTLD_GROUP 以及第 104 页中的 “目标文件分层结构” 中的 RTLD_PARENT。
- 通过新的 dlsym(3C) 句柄 RTLD_DEFAULT 扩展了符号查找语义。 请参见第 100 页中的 “缺省符号查找模型” 。
- 对过滤器处理进行了扩展，从而可定义多个 filtee， 并可强制装入这些 filtee。 请参见第 125 页中的 “作为过滤器的共享库” 。
- 可以使用 mapfile 文件控制指令 $ADDVERS 记录其他版本的依赖项。 请参见第 170 页中的 “到其他版本定义的绑定” 。
- 运行时链接程序审计接口支持从进程内监视和修改动态链接的应用程序。 请参见第 186 页中的 “运行时链接程序审计接口” 。
- 运行时链接程序调试器接口支持从外部进程监视和修改动态链接的应用程序。 请参见第 196 页中的 “运行时链接程序调试器接口” 。
- 支持附加的节信息。对于 SHN_BEFORE 和 SHN_AFTER，请参见表 7–4。对于 SHF_ORDERED 和 SHF_EXCLUDE，请参见表 7–8。
- 支持新的动态节标记 DT_1_FLAGS。有关各种标志值的信息，请参见表 7–34。
- 提供了演示 ELF 程序的软件包。 请参见第 7 章。
- 目前，链接编辑器支持国际化消息。所有系统错误均使用 strerror(3C) 来进行报告。
- 通过新的 eliminate mapfile 指令或 -B eliminate 选项，可以删除局部符号表的各项。 请参见第 62 页中的 “删除符号” 。

索引

数字和符号

$ADDVERS, **请参见**版本控制
$HWCAP, **请参见**搜索路径
$ISALIST, **请参见**搜索路径
$ORIGIN, **请参见**搜索路径
$OSNAME, **请参见**搜索路径
$OSREL, **请参见**搜索路径
$PLATFORM, **请参见**搜索路径
32 位/64 位，24
  ld-support，180
  rtld-audit，189
32 位/64 位, 环境变量，24
32 位/64 位
  搜索路径
    安全性，95
    链接编辑器，33-35
    配置，79
    运行时链接程序，35-36, 76-79, 98
  运行时链接程序，75

A

ABI, **请参见**应用程序二进制接口
ar(1)，30
as(1)，22
atexit(3C)，91

C

CC(1)，28
cc(1)，22, 28
COMDAT，183, 239
COMMON，39, 51, 223
crle(1)，383
  安全，377
  安全性，95, 96
  交互，297, 298
  审计，190
  选项
    -A，379
    -E，381
    -e，148, 381
    -l，79
    -s，95
    -u，382

D

dladdr(3C)，382
dladdr1(3C)，382
dlclose(3C)，91, 97
dldump(3C)，38
dlerror(3C)，97
dlfcn.h，97
dlinfo(3C)，379, 380, 382
dlopen(3C)，75, 96, 97, 103
  版本验证，167
  动态可执行文件，98, 103
  共享库命名约定，120
  模式
    RTLD_FIRST，104, 367, 370, 380

dlopen(3C), 模式（续）
RTLD_GLOBAL，99, 103, 105
RTLD_GROUP，103
RTLD_LAZY，99
RTLD_NOLOAD，187
RTLD_NOW，84, 94, 99
RTLD_PARENT，104
排序影响，102
组，98, 99
dlsym(3C)，75, 97, 104
版本验证，167
特殊句柄
RTLD_DEFAULT，48, 105
RTLD_NEXT，105
RTLD_PROBE，48, 90, 105, 380
RTLD_SELF，382

E
ELF，21, 27
另请参见目标文件
elf(3E)，179
exec(2)，27, 75, 212

F
filtee，125

G
GOT, **请参见**全局偏移表
.got, **请参见**全局偏移表

L
lari(1)，380
LCOMMON，222
ld(1), **请参见**链接编辑器
LD_AUDIT，96, 188
LD_BIND_NOT，112
LD_BIND_NOW，84, 94, 112
LD_BIND_NOW（续）
IA 重定位，310, 312
SPARC 32 位重定位，302
SPARC 64 位重定位，308
LD_BREADTH，92
LD_CONFIG，96
LD_DEBUG，109
LD_DEBUG_OUTPUT，111
LD_LIBRARY_PATH，78, 124
安全性，96
审计，190
LD_LOADFLTR，133
LD_NOAUDIT，189
LD_NOAUXFLTR，133
LD_NODIRECT，83
LD_NOLAZYLOAD，89
LD_NOVERSION，167, 170
LD_OPTIONS，29, 71
LD_PRELOAD，82, 86, 96
LD_PROFILE，148
LD_PROFILE_OUTPUT，149
LD_RUN_PATH，36
LD_SIGNAL，96
ld.so.1(1), **请参见**运行时链接程序
ldd(1)，76
ldd(1) 选项
-d，69, 86, 147
-i，93
-r，69, 86, 147
-u，31
-v，165
/lib，33, 35, 76, 98
/lib/64，33, 35, 76, 98
/lib/secure，95
/lib/secure/64，95
libelf.so.1，180, 211
libldstab.so.1，180
lorder(1)，31, 71

M
mapfile，341
大小符号声明，347
段声明，342

mapfile（续）
结构，341
缺省值，349
示例，347
映射结构，350
映射指令，345
语法，341
mdb(1)，381
mmap(2)，27, 63, 75, 134
moe(1)，379

N

NEEDED，76, 120
nm(1)，135

P

PIC, **请参见**与位置无关的代码
.plt, **请参见**过程链接表
profil(2)，149
pvs(1)，156, 158, 162, 164

R

RTLD_DEFAULT，48
另请参见依赖项排序
RTLD_FIRST，104, 367, 370, 380
RTLD_GLOBAL，99, 103, 105
RTLD_GROUP，103
RTLD_LAZY，99
RTLD_NEXT，105
RTLD_NOLOAD，187
RTLD_NOW，84, 94, 99
RTLD_PARENT，104
RTLD_PROBE，48
另请参见依赖项排序
RUNPATH, **请参见**运行路径

S

SCD, **请参见**应用程序二进制接口
SGS_SUPPORT，179
size(1)，134
Solaris ABI, **请参见**应用程序二进制接口
Solaris 应用程序二进制接口, **请参见**应用程序二进制接口
SONAME，121
SPARC 兼容性定义, **请参见**应用程序二进制接口
strings(1)，142
strip(1)，62, 63
SUNWosdem，195, 198, 211
SUNWtoo，195
SYMBOLIC，148
System V 应用程序二进制接口，357
请参见应用程序二进制接口

T

TEXTREL，138
__thread，313
TLS, **请参见**线程局部存储
tsort(1)，31, 71

U

/usr/ccs/bin/ld, **请参见**链接编辑器
/usr/ccs/lib，33
/usr/lib，33, 35, 76, 98
/usr/lib/64，33, 35, 76, 98
/usr/lib/64/ld.so.1，75, 196
/usr/lib/ld.so.1，75, 196
/usr/lib/secure，95, 188
/usr/lib/secure/64，95, 188

安

安全，377
安全性，95

版
版本控制，153
  标准化，165
  到定义的绑定，163, 168
    $ADDVERS，168
  定义，153, 154, 163
  定义公共接口，60, 155
  概述，153-177
  基本版本定义，156
  文件控制指令，168
  文件名，154, 360
  运行时验证，165, 167
  在映像中生成定义，50, 60, 154-174

绑
绑定
  到版本定义，163
  到共享库依赖项，120, 163
  到弱版本定义 (weak version definition)，172
  延迟，84, 99, 112
  依赖项排序，124
  直接，81, 82, 143

编
编译环境，23, 32, 119
  另请参见链接编辑和链接编辑器
编译器驱动程序，28-29
编译器选项
  -K PIC，139
  -K pic，137, 354
  -xF，140, 239, 344
  -xpg，149
  -xregs=*no%appl*，354

标
标准过滤器，125, 126-129

插
插入，41, 82, 87, 107
  检查，42
  接口稳定性，154

程
程序的解释程序，286
  另请参见运行时链接程序

初
初始化和终止，28, 36-38, 91-95

错
错误消息
  链接编辑器
    soname 冲突，123
    不兼容的选项，29
    不可用的版本，169
    多重定义符号，44
    非法选项，29
    符号警告，42
    根据不可写节重定位，138
    共享库名称冲突，122-123
    未定义符号，44, 45
    未指定给版本的符号，60
    选项的多个实例，29
    选项的非法参数，29
    隐式引用中未定义符号，46
  运行时链接程序
    复制重定位大小差异，69, 147
    找不到版本定义，165
    找不到符号，106
    找不到共享库，78, 98
    重定位错误，85, 166

调

调试帮助
  链接编辑，69-73
  运行时链接，109-118

动

动态可执行文件，22
动态链接，23-24
  实现，247-259, 283
动态信息标记
  NEEDED，76, 120
  RUNPATH，77
  SONAME，121
  SYMBOLIC，148
  TEXTREL，138

段

段，27, 134
  数据，135, 137
  文本，135, 137

多

多重定义符号，31, 41, 238
多重定义数据，141-142, 238

符

符号
  COMMON，39, 51, 223
  LCOMMON，222
  存在测试，47
  定义，30
  多重定义，31, 41, 238
  范围，99, 103
  符号可见性，100
  公共接口，153
  归档提取，30
  寄存器，255, 267
符号（续）
  局部，262
  绝对，51, 222, 223
  可见性，262, 264
  类型，263
  全局，153, 262
  弱，47-48, 262, 263
  删除，62
  未定义，30, 39, 44-48, 222
  已定义，39
  已排序，222
  引用，30
  运行时查找，99, 108
    推迟，84, 99, 112
  暂定，39, 51
    COMMON，223
    LCOMMON，222
    输出文件中的顺序，48-49
    重新对齐，55
  专用接口，153
  自动删除，62
  自动缩减，51, 156, 359
符号保留名称，63
  _DYNAMIC，63
  _edata，63
  _end，63
  END，64
  _etext，63
  _fini，36
  _GLOBAL_OFFSET_TABLE_，64, 139, 299
  _init，36
  main，64
  _PROCEDURE_LINKAGE_TABLE_，64
  _start，64
  START，64
符号解析，38-63, 39-44, 63-67
  插入，82
  多重定义，31
  符号可见性，262, 264
    本地，100
    全局，100
  复杂，42-43
  简单，40-42

符号解析（续）
  搜索范围
    组，99
  致命，43-44

辅

辅助过滤器，126, 129-133

共

共享库，21, 22, 76, 119-151
  请参见共享库
  记录运行时名称，120-123
  具有依赖项，123
  链接编辑器处理，31-32
  命名约定，32, 119
  实现，247-259, 283
  显式定义，46
  依赖项排序，124
  隐式定义，46
  作为过滤器，125-133

归

归档，32
  多次检查整个，31
  链接编辑器处理，30
  命名约定，32
归档文件, 包含共享库，122

过

过程链接表，237, 286
  _PROCEDURE_LINKAGE_TABLE_，64
  动态引用，291, 292
  延迟引用，84
  与位置无关的代码，137
  重定位，251, 299-312
    64–位 SPARC，302-308
    SPARC，252-255, 300-302

过程链接表, 重定位（续）
    x64，258-259, 310-312
    x86，256-258, 308-310
过滤器，125-133
  标准，125, 126-129
  辅助，126, 129-133
  减少 filtee 搜索，369, 371-372
  特定于系统，372
  特定于指令集，370-372
  支持硬件，367-369

环

环境变量
  32 位/64 位，24
  LD_AUDIT，96, 188
  LD_BIND_NOT，112
  LD_BIND_NOW，84, 94, 112
  LD_BREADTH，92
  LD_CONFIG，96
  LD_DEBUG，109
  LD_DEBUG_OUTPUT，111
  LD_LIBRARY_PATH，34, 78, 124
    安全性，96
    审计，190
  LD_LOADFLTR，133
  LD_NOAUDIT，189
  LD_NOAUXFLTR，133
  LD_NODIRECT，83
  LD_NOLAZYLOAD，89
  LD_NOVERSION，167, 170
  LD_OPTIONS，29, 71
  LD_PRELOAD，82, 86, 96
  LD_PROFILE，148
  LD_PROFILE_OUTPUT，149
  LD_RUN_PATH，36
  LD_SIGNAL，96
  SGS_SUPPORT，179

换

换页，280-286

基

基本地址，279

接

接口
- 公共，153, 357
- 专用，153

节

节，27, 134
- **另请参见**节标志, 节名, 节数和节类型

节标志
- SHF_ALLOC，231, 237
- SHF_EXCLUDE，183, 233
- SHF_EXECINSTR，231
- SHF_GROUP，232, 239
- SHF_INFO_LINK，231
- SHF_LINK_ORDER，222, 232
- SHF_MASKOS，232
- SHF_MASKPROC，233
- SHF_MERGE，231
- SHF_ORDERED，233
- SHF_OS_NONCONFORMING，232
- SHF_STRINGS，231
- SHF_WRITE，231
- SHT_TLS，232

节类型
- SHF_TLS，314
- SHT_DYNAMIC，227, 286
- SHT_DYNSTR，227
- SHT_DYNSYM，227
- SHT_FINI_ARRAY，227
- SHT_GROUP，228, 232, 239
- SHT_HASH，227, 241, 287
- SHT_HIOS，228
- SHT_HIPROC，229
- SHT_HISUNW，228
- SHT_HIUSER，229, 352
- SHT_INIT_ARRAY，227
- SHT_LOOS，228
- SHT_LOPROC，229

节类型（续）
- SHT_LOSUNW，228
- SHT_LOUSER，229, 352
- SHT_NOBITS，227
  - .bss，235
  - .lbss，236
  - p_memsz 计算，280
  - sh_offset，224
  - sh_size，225
  - .SUNW_bss，238
  - .tbss，237
- SHT_NOTE，227, 246
- SHT_NULL，226
- SHT_PREINIT_ARRAY，228
- SHT_PROGBITS，227, 286
- SHT_REL，227
- SHT_RELA，227
- SHT_SHLIB，227
- SHT_SPARC_GOTDATA，229
- SHT_STRTAB，227
- SHT_SUNW_ANNOTATE，228
- SHT_SUNW_cap，228
- SHT_SUNW_COMDAT，183, 228, 239
- SHT_SUNW_DEBUG，228
- SHT_SUNW_DEBUGSTR，228
- SHT_SUNW_move，228, 243
- SHT_SUNW_SIGNATURE，228
- SHT_SUNW_syminfo，229
- SHT_SUNW_verdef，229, 269, 272
- SHT_SUNW_verneed，229, 269, 272
- SHT_SUNW_versym，229, 269, 271
- SHT_SYMTAB，227, 264
- SHT_SYMTAB_SHNDX，228

节名
- .bss，27, 144
- .data，27, 141
- .dynamic，63, 75, 148
- .dynstr，63
- .dynsym，63
- .fini，36, 91
- .fini_array，36, 91
- .got，64, 80
- .init，36, 91
- .init_array，36, 91

节名（续）
.interp，75
.picdata，141
.plt，64, 84, 148
.preinit_array，36, 91
.rela.text，27
.rodata，141
.strtab，27, 63
.SUNW_reloc，144, 355
.SUNW_version，269
.symtab，27, 62, 63
.tbss，314
.tdata，314
.tdata1，314
.text，27
节数
SHN_ABS，223, 264, 266
SHN_AFTER，222, 232, 233
SHN_AMD64_LCOMMON，222, 266
SHN_BEFORE，222, 232, 233
SHN_COMMON，223, 263, 266
SHN_HIOS，222
SHN_HIPROC，222
SHN_HIRESERVE，223
SHN_LOOS，222
SHN_LOPROC，222
SHN_LORESERVE，222
SHN_SUNW_IGNORE，222
SHN_UNDEF，222, 266
SHN_XINDEX，223

解

解释程序, **请参见**运行时链接程序

静

静态可执行文件，22

局

局部符号，262

可

可执行链接格式, **请参见**ELF
可重定位目标文件，22

库

库
共享，247-259, 283
归档，32
命名约定，32

链

链接编辑，22-23, 260, 283
到版本定义的绑定，163, 168
动态，247-259, 283
共享库处理，31-32
归档处理，30-31
混合共享库和归档，32-33
库链接选项，30
库输入处理，30
命令行中文件的位置，33
输入文件处理，30-38
搜索路径，33-35
添加其他库，32-36
链接编辑器，21, 27-73
错误消息
请参见错误消息
调试帮助，69-73
段，27
概述，27-73
节，27
使用编译器驱动程序调用，28-29
外部绑定，63
直接调用，28-29
指定选项，29-30
链接编辑器输出
动态可执行文件，22
共享库，22
静态可执行文件，22
可重定位目标文件，22
链接编辑器选项
-64，24, 129

链接编辑器选项（续）
- -a，354
- -B direct，355, 356
- -B dynamic，32
- -B eliminate，62
- -B group，99, 103, 297
- -B local，60
- -B reduce，51, 61
- -B static，32, 354
- -D，69
- -d n，353, 356
- -d y，354
- -e，64
- -F，125
- -f，126
- -G，119, 354, 356
- -h，77, 121, 176, 355
- -i，35
- -L，34, 353
- -l，30, 32-36, 119, 353
- -M，341
- -m，32, 42
- -M
  - 定义版本，153, 155, 359
  - 定义段，28
  - 定义符号，49, 50
  - 定义接口，153, 354
  - 控制绑定需求，168
- -P，188
- -p，188
- -R，35, 123, 355, 356
- -r，28, 353
- -S，179
- -s，62, 63
- -t，42, 43
- -u，49
- -Y，34
- -z allextract，30
- -z combreloc，355
- -z defaultextract，31
- -z defs，47, 51, 188, 355
- -z direct，380
- -z endfiltee，297
- -z finiarray，36

链接编辑器选项（续）
- -z groupperm，298
- -z ignore
  - 节删除，140, 354
  - 依赖项删除，31, 355, 356
- -z initarray，36
- -z initfirst，297
- -z interpose，82, 297
- -z lazyload，88, 298, 355, 356
- -z ld32，180
- -z ld64，180
- -z loadfltr，133, 297
- -z muldefs，44
- -z nocompstrtab，63, 381
- -z nodefaultlib，35, 297
- -z nodefs，45, 85
- -z nodelete，297
- -z nodirect，380
- -z nodlopen，297
- -z nodump，297
- -z nolazyload，88
- -z nopartial，245
- -z noversion，60, 156, 165
- -z now，84, 94, 99
- -z record
  - 依赖项删除，140
- -z rescan，33
- -z text，138, 354
- -z verbose，68
- -z weakextract，30, 263

链接编辑器支持接口 (ld-support)，179
- ld_atexit()，183
- ld_atexit64()，183
- ld_file()，181
- ld_file64()，181
- ld_input_done()，183
- ld_input_section()，182
- ld_input_section64()，182
- ld_section()，182
- ld_section64()，182
- ld_start()，181
- ld_start64()，181
- ld_version()，180

链接程序编辑器, 直接绑定，82

名

名称空间，187

命

命名约定
　共享库，32, 119
　归档，32
　库，32

目

目标文件，21
　程序的解释程序，286
　程序头，275-280
　程序装入，280-286
　段类型，276, 279
　段内容，280
　段权限，279, 280
　符号表，260, 266
　过程链接表
　　请参见过程链接表
　基本地址，279
　节对齐，225
　节类型，225, 238
　节名，238
　节属性，230, 238
　节头，221, 238
　节组标志，239
　全局偏移表
　　请参见全局偏移表
　数据表示形式，212
　在运行时预装入，86
　重定位，247-259
　注释节，246-247, 247
　字符串表，259-260, 260

全

全局符号，153, 262
全局偏移表，286, 299
　_GLOBAL_OFFSET_TABLE_，64

全局偏移表（续）

　.got，236
　动态引用，291
　检查，80
　与位置无关的代码，137
　重定位，251
　　SPARC，252-255
　　x64，258-259
　　x86，256-258
　　与 procedure linkage table 组合，308-310, 310-312

软

软件包
　SUNWosdem，195, 198, 211
　SUNWtoo，195

弱

弱符号
　未定义，30, 47-48
弱符号。，262, 263

生

生成共享库，47
生成可执行，44-46
生成输出文件映像，63-67

输

输入文件处理，30-38

数

数据表示形式，212

搜

搜索路径
  链接编辑，33-35
  运行时链接程序，35-36, 76-79
    `$HWCAP` 标记，367-369
    `$ISALIST` 标记，370-372
    `$ORIGIN` 标记，373-377
    `$OSNAME` 标记，372
    `$OSREL` 标记，372
    `$PLATFORM` 标记，372

未

未定义符号，44-48

线

线程局部存储，313, 381
  节定义，314

性

性能
  动态分配缓冲区，142
  改善引用的邻近性，143-148, 148-151
  基础系统，136-137
  使用自动变量，142
  与位置无关的代码
    请参见位置相关代码
  折叠多个定义，141-142
  重定位，143-148, 148-151
  最大化可共享性，140-142
  最小化数据段，141

虚

虚拟寻址，280-286

延

延迟绑定，84, 99, 112, 186

演

演示
  `prefcnt`，195
  `sotruss`，195
  `symbindrep`，195
  `whocalls`，195

依

依赖项
  组，98, 99
依赖项排序，124

应

应用程序二进制接口，24, 129, 153

与

与位置无关的代码，137-140, 292
  全局偏移表，299

预

预装入目标文件, **请参见**`LD_PRELOAD`

运

运行路径，35, 77, 98, 123
  安全性，96
运行时环境，23, 32, 119
运行时链接，23
运行时链接程序，23, 75, 286-287
  安全性，95
  版本定义验证，165

运行时链接程序（续）
  编程接口
    另请参见dlclose(3C)、dldump(3C)、dlerror(3C)、dlmopen(3C) 和 dlopen(3C)
  初始化和终止例程，91-95
  共享库处理，76
  链接映射，187
  名称空间，187
  搜索路径，35-36, 76-79
  延迟绑定，84, 99, 112
  直接绑定，81, 82, 143
  重定位处理，80-86
  装入其他目标文件，86-87
运行时链接程序支持接口 (rtld-audit)，179, 186-196
  la_activity()，190
  la_amd64_pltenter()，192
  la_i86_pltenter()，192
  la_objclose()，194
  la_objfilter()，191
  la_objopen()，190
  la_objseach()，190
  la_pltexit()，193
  la_preinit()，191
  la_sparcv8_pltenter()，192
  la_sparcv9_pltenter()，192
  la_symbind32()，191
  la_symbind64()，191
  la_version()，189
运行时链接程序支持接口 (rtld-debugger)，179, 196-209
  ps_global_sym()，207
  ps_pglobal_sym()，208
  ps_plog()，208
  ps_pread()，208
  ps_pwrite()，208
  rd_delete()，199
  rd_errstr()，199
  rd_event_addr()，203
  rd_event_enable()，203
  rd_event_getmsg()，204
  rd_init()，198
  rd_loadobj_iter()，201
  rd_log()，199
  rd_new()，199
运行时链接程序支持接口 (rtld-debugger)（续）
  rd_objpad_enable()，207
  rd_plt_resolution()，205
  rd_reset()，199

暂
暂定符号，39

支
支持接口
  链接编辑器 (ld-support)，179
  运行时链接程序 (rtld-audit)，179, 186-196
  运行时链接程序 (rtld-debugger)，179, 196-209

直
直接绑定，81, 82, 143

重
重定位，80-86, 143, 148, 247-259
  非符号，80, 143
  符号，80, 143
  复制，68, 144
  即时，84
  位移，68
  延迟，84
  运行时链接程序
    符号查找，81, 84, 99, 112