RealView® 编译工具

4.0 版

链接器用户指南

ARM®

ARM DUI 0206IC

RealView 编译工具
链接器用户指南

版本信息

本手册进行了以下更改。

更改历史记录

日期	发行号	保密性	更改
2002 年 8 月	A	非保密	1.2 版
2003 年 1 月	B	非保密	2.0 版
2003 年 9 月	C	非保密	ARM® RealView® Developer Suite 2.0.1 版
2004 年 1 月	D	非保密	RealView Developer Suite 2.1 版
2004 年 12 月	E	非保密	RealView Developer Suite 2.2 版
2005 年 5 月	F	非保密	RealView Development Suite 2.2 SP1 版
2006 年 3 月	G	非保密	RealView Development Suite 3.0 版
2007 年 3 月	H	非保密	RealView Development Suite 3.1 版
2008 年 9 月	I	非保密	RealView Development Suite 4.0 版

所有权声明

保密状态

本文档的内容是非保密的。根据 ARM 与 ARM 将本文档交予的参与方的协议条款，使用、复制和公开本文档内容的权利可能会受到许可限制的制约。

受限访问是一种 ARM 内部分类。

产品状态

本文档的信息是开发的产品的最新信息。

网址

http://www.arm.com

目录
RealView Compilation Tools
链接器用户指南

前言
关于本手册 viii
反馈 xi

第 1 章 简介
1.1 关于 ARM 链接器 1-2
1.2 与旧对象和库的兼容性 1-5

第 2 章 ARM 链接器使用入门
2.1 链接模型 2-2
2.2 使用命令行选项 2-6

第 3 章 使用基本链接器功能
3.1 指定映像结构 3-2
3.2 节放置 3-7
3.3 节删除 3-10
3.4 反馈 3-13
3.5 RW 数据压缩 3-15
3.6 中间代码 3-17
3.7 内联 3-20

3.8 使用命令行选项创建简单映像 3-22
3.9 使用命令行选项处理 C++ 异常 3-28
3.10 获得有关映像的信息 3-29
3.11 库搜索、选择和扫描 3-32

第 4 章 访问映像符号
4.1 ARM/Thumb 同义词 4-2
4.2 访问链接器定义的符号 4-3
4.3 访问其他映像中的符号 4-9
4.4 隐藏和重命名全局符号 4-12
4.5 使用 $Super$$ 和 $Sub$$ 覆盖符号定义 4-14
4.6 符号版本控制 4-15

第 5 章 使用分散加载描述文件
5.1 关于分散加载 5-2
5.2 指定区和节地址的示例 5-9
5.3 简单映像的等效分散加载描述 5-31

前言

本前言介绍 《RealView 编译工具链接器用户指南》。本章分为以下几节：

- 第viii 页的*关于本手册*
- 第xi 页的*反馈*

关于本手册

本手册提供有关随 ARM RealView® 编译工具提供的 ARM® 链接器 armlink 的用户信息。文中还概述了命令行选项。有关命令行选项、控制文件命令、分散加载描述文件、基础平台 ABI (BPABI) 和 *System V release 4* (SysV) 共享库和可执行文件的详细信息，请参阅 《链接器参考指南》。

适用对象

本手册是为所有使用 RealView 编译工具生成应用程序的开发人员编写的。本手册假定您是一位有经验的软件开发人员，并且熟悉 《RealView 编译工具要点指南》中所介绍的 ARM 开发工具。

使用本手册

本手册由以下章节组成：

第 1 章 ***简介***

本章简要介绍了链接器及其关联的输入和输出文件。

第 2 章 ***ARM 链接器使用入门***

本章概述了链接器所支持的链接模型、文件命名约定和命令行选项。

第 3 章 ***使用基本链接器功能***

本章概述了映像结构、内存映射、节放置、链接器优化和库文件的使用。

第 4 章 ***访问映像符号***

本章概述了如何访问链接器定义的符号、如何使用控制文件管理输出文件中的符号名称以及符号版本控制。

第 5 章 ***使用分散加载描述文件***

本章概述了分散加载描述文件以及如何使用这些文件指定简单或复杂映像的内存映射。

本手册假定 ARM 软件安装在缺省位置。例如，在 Windows 上，这可能是 *volume*:\Program Files\ARM。引用路径名时，假定安装位置为 *install_directory*，如 *install_directory*\Documentation\...。如果将 ARM 软件安装在其他位置，则可能需要更改此位置。

印刷约定

本手册使用以下印刷约定：

`monospace` 表示可以从键盘输入的文本，如命令、文件和程序名以及源代码。

`mono`space 表示允许的命令或选项缩写。可只输入下划线标记的文本，无需输入命令或选项的全名。

`monospace italic`

表示此处的命令和函数的变量可用特定值代替。

等宽粗体 表示在示例代码以外使用的语言关键字。

斜体 突出显示重要注释、介绍特殊术语以及表示内部交叉引用和引文。

粗体 突出显示界面元素，如菜单名称。有时候也用在描述性列表中以示强调，以及表示 ARM 处理器信号名称。

更多参考出版物

本部分列出了 ARM 公司和第三方发布的、可提供有关 ARM 系列处理器开发代码的附加信息的出版物。

ARM 公司将定期对其文档进行更新和更正。有关最新勘误表、附录和 *ARM 常见问题* (FAQ)，请访问 `http://infocenter.arm.com/help/index.jsp`。

ARM 公司出版物

本手册包含有关如何使用随 RealView 编译工具提供的 ARM 链接器的信息。该套件中包含的其他出版物有：

- 《RealView 编译工具要点指南》 (ARM DUI 0202)
- 《RealView 编译工具编译器用户指南》 (ARM DUI 0205)
- 《RealView 编译工具编译器参考指南》 (ARM DUI 0348)
- 《RealView 编译工具库和浮点支持指南》 (ARM DUI 0349)
- 《RealView 编译工具链接器参考指南》 (ARM DUI 0381)
- 《RealView 编译工具实用程序指南》 (ARM DUI 0382)
- 《RealView 编译工具汇编器指南》 (ARM DUI 0204)

- 《RealView 编译工具开发指南》 (ARM DUI 0203)

有关基本标准、软件接口和 ARM 支持的标准的完整信息，请参阅 *install_directory*\Documentation\Specifications\...。

此外，有关与 ARM 产品相关的特定信息，请参阅下列文档：

- 《ARM 体系结构参考手册》， ARMv7-A™ 和 ARMv7-R™ 版 (ARM DDI 0406)
- 《ARM7-M™ 体系结构参考手册》 (ARM DDI 0403)
- 《ARM6-M™ 体系结构参考手册》 (ARM DDI 0419)
- 《ARM 体系结构参考手册》 (ARM DDI 0100)
- 您的硬件设备的 ARM 数据手册或技术参考手册

反馈

ARM Limited 欢迎提供有关 RealView 编译工具与文档的反馈。

对 RealView 编译工具的反馈

如果您对 RealView 编译工具有任何问题，请与供应商联系。为便于供应商快速提供有用的答复，请提供：

- 您的姓名和公司
- 产品序列号
- 您所用版本的详细信息
- 您运行的平台的详细信息，如硬件平台、操作系统类型和版本
- 能重现问题的一小段独立的程序
- 您预期发生和实际发生的情况的详细说明
- 您使用的命令，包括所有命令行选项
- 能说明问题的示例输出
- 工具的版本字符串，包括版本号和内部版本号

关于本手册的反馈

如果您发现本手册有任何错误或遗漏之处，请发送电子邮件到 `errata@arm.com`，并提供：

- 文档标题
- 文档编号
- 您有疑问的页码
- 问题的简要说明

我们还欢迎您对需要增加和改进之处提出建议。

第 1 章
简介

本章介绍随 RealView® 编译工具提供的 ARM® 链接器 armlink。本章分为以下几节：

- 第1-2 页的*关于ARM 链接器*
- 第1-5 页的*与旧对象和库的兼容性*

1.1 关于 ARM 链接器

链接器可以链接 ARM 代码、Thumb® 代码和 Thumb-2 代码，并自动生成交互操作中间代码，以便在需要时切换处理器状态。链接器还可以在需要时自动生成内联中间代码或长跳转中间代码，以扩展跳转指令的范围。

链接器根据链接的对象的生成属性，自动选择要链接的相应标准 C 或 C++ 库变体。

链接器支持的一些命令行选项可用于指定代码和数据在系统内存映射中的位置。另外，您也可以使用分散加载描述文件，在加载时和执行时指定输出映像中各个代码和数据节在内存中的位置。这样可以创建跨越多个内存的复杂映像。

链接器支持读/写数据压缩，以最小化 ROM 大小。

链接器可以执行未使用节删除，以减少输出映像的大小。另外，链接器还可以：

- 生成关于链接文件的调试和引用信息
- 生成静态调用图并列出堆栈的使用情况
- 控制输出映像中符号表的内容
- 显示输出中代码和数据的大小

链接器针对下一次文件编译提供反馈信息，提示编译器有关未使用函数的情况。然后在后续编译中将未使用的函数放置在各自的节中，以便链接器将来删除这些函数。

有关详细信息，请参阅第 2 章 *ARM 链接器使用入门*。

1.1.1 ARM 链接器的输入

`armlink` 的输入包括：

- 采用 ARM *可执行和链接格式* (ELF) 的一个或多个对象文件。《ARM ELF 规范》中描述了这种格式。
- 用 `armar` 创建的一个或多个库，如 《实用程序指南》的第 3 章 *使用 armar* 中所述。
- 一个符号定义文件。
- 分散加载描述文件。

1.1.2 ARM 链接器的输出

成功调用 armlink 后的输出为下列项之一：

- ELF 可执行格式的可执行映像
- ELF 共享对象格式的共享对象
- ELF 对象格式的部分链接对象
- ELF 对象格式的可重定位对象

对于简单映像， ELF 可执行文件包含相当于映像中的 RO 和 RW 输出节的段。ELF 可执行文件还具有包含映像输出节的 ELF 节。

可以使用 fromelf 将 ELF 可执行格式的可执行映像转换为其他文件格式。有关详细信息，请参阅 《实用程序指南》中的第 2 章 *使用 fromelf*。

构建可执行映像

使用链接器构建可执行映像时，链接器将：

- 解析输入对象文件之间的符号引用
- 从库中提取对象模块来满足还未满足的符号引用的需要
- 根据属性和名称排序输入节，并将属性和名称相似的节合并为相邻块
- 删除未使用节
- 删除重复的公共组和公共代码、数据及调试节
- 根据提供的分组和布局信息将对象片段组织为内存区
- 给可重定位值分配地址
- 生成可执行映像

有关详细信息，请参阅第3-2 页的*指定映像结构*。

构建部分链接对象

使用链接器构建部分链接对象时，链接器将：

- 删除重复的调试节副本
- 将符号表合并成一个表
- 将未解析的引用保留为未解析状态
- 合并 Comdat 组
- 生成一个对象，将其用作后续链接步骤的输入

注意

如果使用部分链接，则不能在分散加载描述文件中使用名称来引用组件对象。

1.2 与旧对象和库的兼容性

如果从早期版本升级到 RealView 编译工具，请务必阅读 《RealView 编译工具要点指南》以了解最新信息。

RealView 编译工具 v4.0 中使用的应用程序二进制接口 (ABI) 与 v2.0 之前的早期版本工具中使用的接口不同。因此，旧对象和库在新版本中不兼容，必须重新生成它们。

有关详细信息，请参阅 《库指南》中第1-3 页的*ARM 体系结构 ABI 遵从性*，以及 《RealView 编译工具要点指南》中有关与旧对象和库的兼容性的章节。

第 2 章
ARM 链接器使用入门

本章概括了 ARM® 链接器 armlink 接受的命令行选项。具体介绍了命令行选项的排序、如何调用链接器、必须如何配置环境变量以及 ARM RealView® 编译工具文件命名约定。

该链接器将一个或多个对象文件的内容与一个或多个对象库的选定部分合并起来，生成一个映像或对象文件。本章分为以下几节：

- 第2-2 页的*链接模型*
- 第2-6 页的*使用命令行选项*

2.1 链接模型

本节介绍 ARM 链接器支持的链接模型，并概括了具体的链接行为以及适用的限制。本节未介绍所有链接模型通用的选项。

一个链接模型就是一组控制链接器行为的命令行选项、内部选项和内存映射。

裸机 此模型不针对任何特定平台。使用此模型可以创建一个映像，其中包含您自己自定义的操作系统、内存映射和应用程序代码（如果需要）。可提供一些有限的动态链接支持。根据是否使用分散加载文件，还可应用其他选项。

部分链接 此模型生成与平台无关的对象，可适用于在后续链接步骤中作为链接器的输入。此模型还可用作开发过程中的中间步骤，并对输入对象执行有限的处理以生成单个输出对象。

BPABI 此模型支持类似于 DLL 的 BPABI。它用于生成在复杂性各不相同的平台 OS 上运行的应用程序和 DLL。内存模型根据 BPABI 规范而受到限制。

SysV 此模型支持在 ARM Linux 所使用的 ELF 中指定的 SysV 模型。内存模型根据 ELF 规范而受到限制。

在每个模型中，可以组合相关选项以加强对输出的控制。以下各节更详细地介绍了这些组合：

有关详细信息，请参阅《链接器参考指南》中第 4 章 *BPABI 和 SysV 共享库和可执行文件*。

2.1.1 裸机

裸机模型重点针对常规嵌入式领域，其中整个程序（可能包括 RTOS）在一轮链接中进行链接。对于裸机系统的内存映射，链接器只能进行极少的假设，因此如果需要更精确的控制，必须使用分散加载机制。

缺省情况下，链接器会尝试以静态方式解析所有重定位。但是，也可以创建与位置无关或可重定位的映像，可以从不同的地址执行该映像，并在加载或运行时解析该映像的重定位。可以使用动态模型实现此目的。

此类型的模型分三部分：

- 使用分散加载描述文件或命令行选项标识可以进行重定位或与位置无关的区。
- 使用控制文件标识可以导入或导出的符号。
- 使用控制文件标识 ELF 文件所需的共享库。

另请参阅

可在裸机系统中使用以下选项：

- 《链接器参考指南》中第2-18 页的*--edit=file_list*
- 《链接器参考指南》中第2-51 页的*--scatter=file*
- 《链接器参考指南》中第2-63 页的*控制文件命令*

在未使用分散加载时还可以使用以下选项：

- 《链接器参考指南》中第2-47 页的*--reloc*
- 《链接器参考指南》中第2-48 页的*--ro_base=address*
- 《链接器参考指南》中第2-48 页的*--ropi*
- 《链接器参考指南》中第2-49 页的*--rosplit*
- 《链接器参考指南》中第2-49 页的*--rw_base=address*
- 《链接器参考指南》中第2-50 页的*--rwpi*
- 《链接器参考指南》中第2-54 页的*--split*

2.1.2 部分链接

部分链接会删除所有输入文件都有的重复信息，并将符号表合并为一个表。会生成单个输出文件，该文件可用作后续链接步骤的输入。如果链接器在输入文件中找到多个入口点，则会生成错误，因为输出文件只能有一个入口点。

命令行选项

在进行部分链接时可以使用以下选项：

- 《链接器参考指南》中第2-40 页的*--partial*
- 《链接器参考指南》中第2-18 页的*--edit=file_list*
- 《链接器参考指南》中第2-20 页的*--[no_]exceptions_tables=action*

2.1.3 对 BPABI 和 SysV 通用的概念

对于 BPABI 和 SysV，映像和共享对象通常在现有操作平台上运行。

BPABI 与 SysV 模型之间有许多相似之处。例如，两者都会生成一个映射异常表的程序头文件。主要差异在于内存模型以及*过程链接表* (PLT) 和*全局偏移表* (GOT) 结构。有许多选项对两个模型是通用的。

限制

两个模型都有以下限制：

- 对于共享库和 DLL，关闭了未使用节删除功能
- 关闭了虚函数删除功能
- 不允许进行读写数据压缩
- 不允许进行分散加载
- 不允许使用 __AT 节

另请参阅

以下选项可以用于 --bpabi 和 --sysv：

- 《链接器参考指南》中第2-17 页的*--dynamic_debug*
- 《链接器参考指南》中第2-24 页的*--[no_]force_so_throw*
- 《链接器参考指南》中第2-49 页的*--runpath=pathlist*
- 《链接器参考指南》中第2-53 页的*--soname=name*
- 《链接器参考指南》中第2-57 页的*--symver_script=file*
- 《链接器参考指南》中第2-57 页的*--symver_soname*

2.1.4 BPABI

BPABI 是一个元标准，供第三方参考以生成自己的特定于平台的映像格式。这意味着 BPABI 模型会生成尽可能多的信息，而不会集中于任何特定平台。

异常

假定共享对象无法引发异常，请参阅 《链接器参考指南》中第2-24 页的*--[no_]force_so_throw*。

符号版本控制

必须使用符号版本控制来确保所需全部符号在加载时均可用。

另请参阅

以下选项可以用于 --bpabi：

- 《链接器参考指南》中第2-17 页的*--dll*
- 《链接器参考指南》中第2-41 页的*--pltgot=type*
- 《链接器参考指南》中第2-48 页的*--ro_base=address*

- 《链接器参考指南》中第2-49 页的*--rosplit*
- 《链接器参考指南》中第2-49 页的*--rw_base=address*
- 《链接器参考指南》中第2-50 页的*--rwpi*

2.1.5 SysV

SysV 模型生成 SysV 共享对象和可执行文件。该模型还可用于生成与 ARM Linux 兼容的共享对象和可执行文件。

线程局部存储

此模型中支持线程局部存储。

异常

假定共享对象可以引发异常，请参阅 《链接器参考指南》中第2-24 页的*--[no_]force_so_throw*。

另请参阅

以下选项特定于 --sysv：

- 《链接器参考指南》中第2-24 页的*--fpic*
- 《链接器参考指南》中第2-35 页的*--linux_abitag=version_id*
- 《链接器参考指南》中第2-52 页的*--shared*

2.2 使用命令行选项

根据选项类型而定，以下规则可能适用：

单字母选项

在所有单字母选项或带参数的单字母选项之前，都有一个单短划线 -。选项与参数之间可以有空格，参数也可以紧跟在选项之后。

关键字选项

在所有关键字选项或带参数的关键字选项前面，都有一个双短划线 --。选项和参数之间需要用 = 或空格字符分隔。例如：

```
--scatter=file
--scatter file
```

包含非前置 - 或 _ 的选项可以使用这两个字符中的任何一个，例如 `--force_so_throw` 与 `--force-so-throw` 相同。

对于以短划线开头的文件名，请使用 POSIX 选项 -- 指定将所有的后续参数均视为文件名，而不是命令开关。例如，要链接名为 `-ifile_1` 的文件，请使用下面的命令：`armlink -- -ifile_1`

2.2.1 调用 ARM 链接器

调用 ARM 链接器的命令是：

```
armlink [help-options] [license-option] [project-template-options]
[library-options] [linker-control-options] [output-options]
[memory-map-options] [debug-options] [image-content-options]
[veneer-generation-options] [byte-addressing-mode] [image-info-options]
[diagnostic-options] [via-file-options] [miscellaneous-options]
[arm-linux-options] [input-file]
```

有关以下每个选项的详细信息，请参阅《链接器参考指南》中第 2 章 *链接器命令行选项*：

help-options

显示主要命令行选项、编译器的版本号，以及编译器处理命令行的方式：

- 第2-25 页的 *--help*
- 第2-52 页的 *--show_cmdline*
- 第2-61 页的 *--vsn*

license-option

可使用下面的选项多尝试几次获取浮动许可证：

- 第2-35 页的*--licretry*

project-template-options

项目模板是包含项目信息的文件，如用于特定配置的命令行选项。这些文件存储在项目模板工作目录中。

使用以下选项可控制项目模板的使用：

- 第2-44 页的*--[no_]project=filename*
- 第2-47 页的*--reinitialize_workdir*
- 第2-61 页的*--workdir=directory*

library-options

使用以下选项可控制库文件和路径：

- 第2-33 页的*--libpath=pathlist*
- 第2-34 页的*--library_type=lib*
- 第2-45 页的*--[no_]reduce_paths*
- 第2-50 页的*--[no_]scanlib*
- 第2-59 页的*--userlibpath=pathlist*

linker-control-options

使用以下选项可控制对象的链接方式：

- 第2-37 页的*--match=crossmangled*
- 第2-55 页的*--strict*
- 第2-55 页的*--[no_]strict_relocations*
- 第2-58 页的*--unresolved=symbol*

output-options

以下选项指定输出文件的格式和名称：

- 第2-5 页的*--bpabi*
- 第2-11 页的*--[no_]combreloc*
- 第2-17 页的*--dll*
- 第2-39 页的*--output=file*
- 第2-40 页的*--partial*
- 第2-43 页的*--[no_]prelink_support*
- 第2-47 页的*--reloc*

- 第2-52 页的*--shared*
- 第2-57 页的*--sysv*

memory-map-options

使用以下选项可指定内存映射：

- 第2-4 页的*--[no_]autoat*
- 第2-24 页的*--fpic*
- 第2-42 页的*--predefine="string"*
- 第2-48 页的*--ro_base=address*
- 第2-48 页的*--ropi*
- 第2-49 页的*--rosplit*
- 第2-49 页的*--rw_base=address*
- 第2-50 页的*--rwpi*
- 第2-51 页的*--scatter=file*
- 第2-54 页的*--split*

debug-options

使用以下选项可控制映像中的调试信息：

- 第2-5 页的*--[no_]bestdebug*
- 第2-11 页的*--[no_]compress_debug*
- 第2-13 页的*--[no_]debug*
- 第2-17 页的*--dynamic_debug*

image-content-options

使用以下选项可控制影响映像内容的其他因素：

- 第2-6 页的*--[no_]branchnop*
- 第2-12 页的*--[no_]cppinit*
- 第2-12 页的*--cpu=list*
- 第2-12 页的*--cpu=name*
- 第2-13 页的*--datacompressor=opt*
- 第2-18 页的*--edit=file_list*
- 第2-18 页的*--entry=location*
- 第2-19 页的*--[no_]exceptions*
- 第2-20 页的*--[no_]exceptions_tables=action*
- 第2-20 页的*--[no_]export_all*
- 第2-23 页的*--[no_]filtercomment*

- 第2-23 页的*--fini=symbol*
- 第2-23 页的*--first=section_id*
- 第2-24 页的*--[no_]force_so_throw*
- 第2-25 页的*--fpu=list*
- 第2-25 页的*--fpu=name*
- 第2-28 页的*--init=symbol*
- 第2-28 页的*--[no_]inline*
- 第2-30 页的*--keep=section_id*
- 第2-32 页的*--last=section_id*
- 第2-36 页的*--[no_]locals*
- 第2-36 页的*--ltcg*
- 第2-38 页的*--max_visibility=type*
- 第2-38 页的*--[no_]merge*
- 第2-39 页的*--[no_]muldefweak*
- 第2-39 页的*--override_visibility*
- 第2-40 页的*--pad=num*
- 第2-41 页的*--pltgot=type*
- 第2-41 页的*--pltgot_opts=mode*
- 第2-44 页的*--profile=filename*
- 第2-46 页的*--[no_]ref_cpp_init*
- 第2-47 页的*--[no_]remove*
- 第2-53 页的*--soname=name*
- 第2-53 页的*--sort=algorithm*
- 第2-55 页的*--[no_]startup=symbol*
- 第2-57 页的*--symver_script=file*
- 第2-57 页的*--symver_soname*
- 第2-58 页的*--[no_]tailreorder*
- 第2-60 页的*--vfemode=mode*

veneer-generation-options

使用以下选项可控制中间代码的生成：

- 第2-29 页的*--[no_]inlineveneer*
- 第2-40 页的*--[no_]piveneer*
- 第2-59 页的*--[no_]veneershare*

byte-addressing-mode

使用以下选项可控制字节寻址模式：

- 第2-4 页的*--be8*
- 第2-5 页的*--be32*

image-info-options

除 --callgraph 之外，缺省情况下链接器会在标准输出流 stdout 中输出请求的信息。您可以使用 --list 命令行选项将信息重定向到文本文件。

使用以下选项可控制如何提取并呈现映像的相关信息：

- 第2-7 页的*--[no_]callgraph*
- 第2-8 页的*--callgraph_file=filename*
- 第2-9 页的*--callgraph_output=fmt*
- 第2-9 页的*--cgfile=type*
- 第2-10 页的*--cgsymbol=type*
- 第2-10 页的*--cgundefined=type*
- 第2-21 页的*--feedback=file*
- 第2-21 页的*--feedback_image=option*
- 第2-25 页的*--info=topic[,topic,...]*
- 第2-27 页的*--info_lib_prefix=opt*
- 第2-36 页的*--[no_]list_mapping_symbols*
- 第2-36 页的*--ltcg*
- 第2-36 页的*--[un]mangled*
- 第2-37 页的*--[no_]map*
- 第2-52 页的*--section_index_display=type*
- 第2-56 页的*--[no_]symbols*
- 第2-56 页的*--symdefs=file*
- 第2-62 页的*--[no_]xref*
- 第2-62 页的*--[no_]xrefdbg*
- 第2-62 页的*--xref{from|to}=object(section)*

diagnostic-options

使用以下选项可控制链接器发出诊断消息的方式：

- 第2-15 页的*--diag_style=arm|ide|gnu*
- 第2-16 页的*--diag_suppress=tag[,tag,...]*

- 第2-16 页的*--diag_warning=tag[,tag,...]*
- 第2-19 页的*--errors=file*
- 第2-35 页的*--list=file*
- 第2-59 页的*--verbose*

via-file-options

使用以下选项可以指定包含附加链接器命令行参数的 via 文件：

- 第2-61 页的*--via=file*

miscellaneous-options

- 第2-32 页的*--[no_]legacyalign*

arm-linux-options

可将以下选项用于 ARM Linux 应用程序：

- 第2-2 页的*--[no_]add_needed*
- 第2-3 页的*--arm_linux*
- 第2-17 页的*--dynamiclinker=name*
- 第2-33 页的*--library=name*
- 第2-35 页的*--linux_abitag=version_id*
- 第2-49 页的*--runpath=pathlist*
- 第2-51 页的*--[no_]search_dynamic_libraries*

input-file 这是以空格分隔的对象、库或*符号定义* (symdefs) 文件的列表。有关详细信息，请参阅第2-29 页的*input_file_list*。

2.2.2 命令行选项排序

通常，命令行选项可以按任意顺序出现。不过，一些选项的效果取决于它们如何与其他相关选项组合。

如果选项在相同的命令行上覆盖先前的选项，则最后的选项将优先执行。如果某个选项不遵循该规则，则在说明中予以指明。使用 `--show_cmdline` 可查看如何从命令行处理选项。命令以标准化方式显示，并展开所有 via 文件的内容。

有关输入文件的优先级排序的详细信息，请参阅 《链接器参考指南》中第2-29 页的*input_file_list*。

2.2.3 使用环境变量指定命令行选项

通过设置 RVCT*ver*_LINKOPT 环境变量的值，可以指定命令行选项。其语法与命令行语法相同。链接器读取 RVCT*ver*_LINKOPT 的值，然后将其插入到命令字符串之前。这意味着在 RVCT*ver*_LINKOPT 中指定的选项可由命令行中的参数覆盖。

2.2.4 从文件读取命令行选项

如果操作系统限制命令行的长度，则可以使用以下编译器选项在文件中提供附加的命令行选项：

`--via` *filename*

编译器将打开指定文件，并从中读取附加命令行选项。

有关详细信息，请参阅 《编译器参考指南》中附录 A *via 文件语法*。

第 3 章
使用基本链接器功能

本章介绍了随 ARM RealView® 编译工具一起提供的 ARM® 链接器 armlink 中的基本功能。本章分为以下几节：

- 第3-2 页的*指定映像结构*
- 第3-7 页的 *节放置*
- 第3-10 页的 *节删除*
- 第3-13 页的*反馈*
- 第3-15 页的*RW 数据压缩*
- 第3-17 页的 *中间代码*
- 第3-20 页的*内联*
- 第3-22 页的*使用命令行选项创建简单映像*
- 第3-28 页的*使用命令行选项处理 C++ 异常*
- 第3-29 页的*获得有关映像的信息*
- 第3-32 页的*库搜索、选择和扫描*

3.1 指定映像结构

映像的结构由以下各项定义：

- 映像的组成区和输出节的数量
- 加载映像时这些区和节在内存中的位置
- 执行映像时这些区和节在内存中的位置

描述内存映射时：

- 术语 *根区域*用于描述加载地址和执行地址相同的区。
- 载入区相当于 ELF 段。

3.1.1 对象和映像的构建块

可执行文件由映像、区、输出节和输入节的层次结构构成：

- 映像由一个或多个区组成。每个区由一个或多个输出节组成。
- 每个输出节包含一个或多个输入节。
- 输入节是对象文件中的代码和数据信息。

图 3-1 显示了区、输出节和输入节之间的关系。

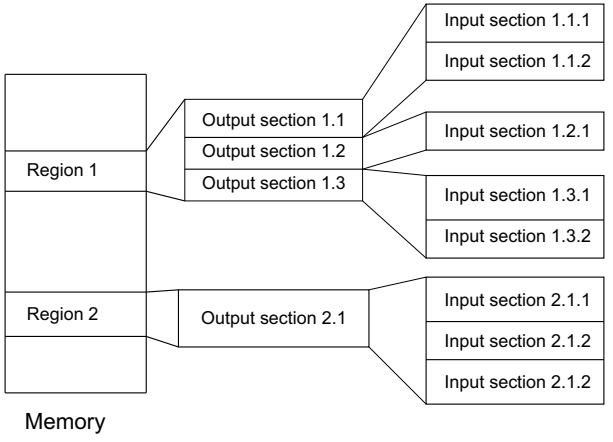

图 3-1 构建映像块

第3-2 页的图 3-1 显示了构成映像的构建块：

输入节 输入节包含代码、初始化数据，或描述未初始化的或在映像执行前必须设为 0 的内存片断。这些特性通过 RO、RW 和 ZI 这样的属性来表示。有关详细信息，请参阅第3-8 页的*按属性排序输入节*。armlink 使用这些属性，将输入节组织到称为输出节和区的更大的构建块中。

输出节 一个输出节由若干个具有相同 RO、RW 或 ZI 属性的相邻输入节组成。输出节的属性与组成它的输入节的属性相同。在输出节中，输入节根据第3-7 页的*节放置*中描述的规则进行排序。

区 一个区由一个、两个或三个相邻的输出节组成。区中的输出节根据其属性排序。首先是 RO 输出节，然后是 RW 输出节，最后是 ZI 输出节。区通常映射到物理内存设备，如 ROM、 RAM 或外围设备。

3.1.2 映像的加载视图和执行视图

映像区在加载时放入系统内存映射。在执行映像之前，您可能要将它的一些区移到执行地址并创建 ZI 输出节。例如，必须将已初始化的 RW 数据从 ROM 中的加载地址复制到 RAM 中的执行地址。

映像的内存映射具有以下不同的视图 （如图 3-2 中所示）。

加载视图 根据映像加载到内存时所在的地址 （即映像开始执行*之前*的位置）来描述每个映像的区和节。

执行视图 根据映像执行*时*所在的地址来描述每个映像的区和节。

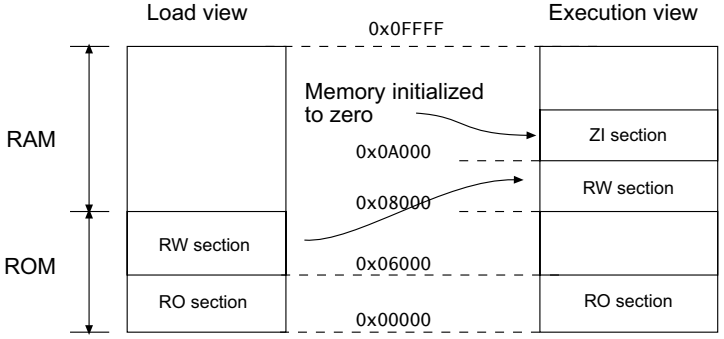

图 3-2 加载和执行内存映射

表 3-1 比较了加载视图和执行视图。

表 3-1 比较加载视图和执行视图

加载	说明	执行	说明
加载地址	在包含节或区的映像开始执行之前，该节或区加载到内存中的地址。节或非根区的加载地址可以与其执行地址不同。	执行地址	当包含节或区的映像执行时，节或区所在的地址。
载入区	加载地址空间中的区	执行区	执行地址空间中的区

3.1.3 指定映像的内存映射

映像可以由任意个区和输出节组成。所有这些区可以有不同的加载地址和执行地址。要构建映像的内存映射，armlink 必须有以下各项信息：

分组 如何将输入节分组为输出节和区。

布局 在内存映射中如何安排映像区的位置。

根据映像的内存映射的复杂程度，有两种方法可将此信息传递到 armlink：

使用命令行选项

对于映像只有一个或两个加载区和最多三个执行区的简单情况，可以使用以下选项：

- `--ro-base`
- `--rw-base`
- `--ropi`
- `--rwpi`
- `--first`
- `--last`
- `--split`
- `--rosplit`

以上列出的选项是一种简化的表示法，它提供的设置与简单映像的分散加载描述相同。有关详细信息，请参阅第3-22 页的*使用命令行选项创建简单映像*。

使用分散加载描述文件

对于需要对映像组件的分组和布局进行全面控制的较复杂的情况，可以使用分散加载描述文件。要使用分散加载描述文件，请在命令行中指定 `--scatter=filename`。这些内容在第 5 章 *使用分散加载描述文件*中有详细描述。

3.1.4 映像入口点

映像中的入口点是程序开始执行的位置。有两种不同类型的入口点：

初始入口点

映像的*初始*入口点是存储在 ELF 头文件中的单个值。对于由操作系统或引导加载程序将程序加载到 RAM 的情况，加载程序通过将控制权传送到映像中的初始入口点来启动映像的执行。

一个映像只能有一个初始入口点。初始入口点可以是 （但不必一定是）由 ENTRY 指令设置的入口点之一。

由 ENTRY 指令设置的入口点

这些是在汇编语言源代码中使用 ENTRY 指令设置的入口点。在嵌入式系统中，该指令通常用于标记通过处理器异常向量输入的代码，如 RESET、 IRQ 和 FIQ。

可以用 ENTRY 指令在映像中指定多个入口点。该指令用 ENTRY 关键字标记输出代码节，指示链接器在执行未使用节删除时不能删除该节。

对于 C 和 C++ 程序， C 库中的 `__main()` 函数也是入口点。

有关 ENTRY 指令的详细信息，请参阅 《汇编器指南》。

如果嵌入式映像用于加载程序，则它必须在头文件中指定单个初始入口点。有关详细信息，请参阅*指定初始入口点*。

指定初始入口点

必须为程序指定至少一个初始入口点，否则链接器会生成警告。并非每个源文件都必须具有入口点。单个源文件中不允许有多个入口点。

对于 ROM 起始地址为 0 的嵌入式应用程序，可使用 `--entry 0x0` （或对于有高位向量的 CPU，可以使用 `0xFFFF0000`）。

初始入口点必须满足以下条件：

- 映像入口点必须始终在执行区内

- 执行区必须是非重叠的，而且必须是根执行区 （加载地址与执行地址相同）。

如果不使用 --entry 选项指定初始入口点，则：

- 如果输入对象仅包含一个由 ENTRY 指令设置的入口点，则链接器将使用该入口点作为映像的初始入口点
- 在以下情况时，链接器生成不包含初始入口点的映像：
  - 已使用 ENTRY 指令指定多个入口点
  - 未使用 ENTRY 指令指定任何入口点。

有关详细信息，请参阅：

- 《链接器参考指南》中第2-18 页的*--entry=location*
- 《汇编器指南》中第7-69 页的*ENTRY*

3.2 节放置

链接器根据输入节的属性在一个区内对它们进行排序。具有相同属性的输入节在区内形成相邻块。

每个输入节的基址由链接器定义的排序顺序确定，并且在包含它的输出节中正确对齐。

通常，生成映像时链接器按以下顺序对输入节进行排序：

1. 按属性。
2. 按输入节名称。
3. 按其在输入列表中的位置，除非由 FIRST 或 LAST 覆盖。请参阅第3-8 页的*使用 FIRST 和 LAST 放置节*。

注意

此排序顺序不受分散加载描述文件或对象文件名中的排序的影响。

如果执行区包含 4MB 的 Thumb 代码、 16MB 的 Thumb-2 代码或超过 32MB 的 ARM 代码，则链接器可能会更改排序顺序，从而将长跳转中间代码的数量减至最小。有关详细信息，请参阅第3-17 页的*中间代码*。

在缺省情况下，链接器创建由 RO、 RW 和可选的 ZI 输出节组成的映像。RO 输出节在具有内存管理硬件的系统上运行时可以受到保护。RO 节也可以放在目标的 ROM 中。

3.2.1 按属性排序输入节

部分映像集合在一起，形成最小数量的相邻区。armlink 按如下属性对输入节排序：

1. 只读代码
2. 只读数据
3. 读写代码
4. 读写数据
5. 零初始化数据

具有相同属性的输入节按名称排序。名称是区分大小写的，并且使用 ASCII 字符排列顺序，对名称按字母顺序进行比较。

属性和名称相同的输入节根据它们在输入文件中的相对位置进行排序。

这些规则意味着库中所含属性和名称相同的输入节的位置不可预判。如果需要更精确的定位，可以显式指定各个模块并将这些模块包含在输入列表中。

3.2.2 使用 FIRST 和 LAST 放置节

如果未使用分散加载，请使用 --first 和 --last 链接器选项来放置输入节。

如果使用分散加载而且放置顺序很重要，则可在文件中使用 FIRST 和 LAST 属性以标记执行区中的第一个和最后一个输入节。

但是，FIRST 和 LAST 不能破坏在*按属性排序输入节*中描述的基本的属性排列顺序。例如，FIRST RW 放在任何只读代码或只读数据之后。

3.2.3 对齐节

排序输入节之后和修复基址之前，armlink 根据需要插入填充，以强制每个输入节的开始地址是输入节对齐的倍数。

ARM 链接器允许 ELF 程序头文件和输出节在四字节边界对齐，而不考虑输入节的最大对齐要求。这样使 armlink 可以将插入映像的填充数量最小化。

如果需要严格遵循 ELF 规范，则使用 --no_legacyalign 选项。对于未对齐的区的基址，链接器会报错，因此可以插入填充以确保遵从性。使用 --no_legacy_align 时，区对齐是区所包含的任何输入节的最大对齐。

可以使用 ALIGN 来扩展区的对齐，例如，将通常按 4 字节的对齐改为按 8 字节对齐。您不能减少自然对齐，例如，强制将通常按 4 字节的对齐改为按 2 字节对齐。有关详细信息，请参阅第5-29 页的*在页边界上创建区*。

您也可以增加输入节的对齐。请参阅 《汇编器指南》的指令参考中对 ALIGN 的说明。

3.2.4 对包含 Thumb 代码的执行区进行排序

Thumb 的跳转范围为 4MB。如果执行区包含的 Thumb 代码超过 4MB， armlink 尝试将平均调用深度相近的节排序，并将最常调用的节放置在中间。这有助于最小化生成的中间代码数量。有关详细信息，请参阅第3-17 页的*中间代码*。

Thumb-2 的跳转范围为 16MB。只有在超出限制时，才需要重新排序节。

3.3 节删除

本节介绍链接器对重复节、未用节和调试节的优化。

3.3.1 公共调试节删除

在 DWARF 2 中，编译器和汇编器为构成编译单元的每个源文件生成一组调试节。armlink 可以检测到特定源文件的调试节的多个副本，并在最终的映像中只保留一个副本，而丢弃其他所有副本。这可以显著地减小映像调试的大小。

在 DWARF 3 中，公共调试节放在公共组中。armlink 只保留每组的具有相同签名的一个副本，而丢弃其他所有副本。

3.3.2 公共组或节删除

如果在 C++ 源代码中使用了内联函数或模板，则 ARM 编译器生成用于链接的完整对象，因此每个对象都包含其需要的内联函数和模板函数的外部副本。当在公共头文件中声明这些函数时，函数可能在随后链接在一起的各个对象中被定义多次。为了删除重复副本，编译器会将这些函数编译到公共代码节或组的单独实例中。

公共代码节或组的单独实例可能不同。例如，可能在库中找到某些副本，它们已经用不同的 （但兼容的）编译选项、不同的优化选项或不同的调试选项进行编译。

如果副本不同，则 armlink 根据输入对象的属性保留每个公共代码节或组的最常用的变体。armlink 将丢弃其他变体。

如果副本相同，则 armlink 保留找到的第一个节或组。

此优化可由以下链接器选项控制：

- 选择 --bestdebug 选项可使用最大的 Comdat 组 （可能提供最佳的调试视图）。
- 选择 --no_bestdebug 选项可使用最小的 Comdat 组 （可能提供最小的代码大小）。这是缺省设置。

  由于 --no_bestdebug 为缺省值，因此不管您在使用 --debug 编译时是否生成调试表，最终映像都是一样的。

  有关详细信息，请参阅：
  — 《编译器用户指南》中第5-16 页的*函数内联*
  — 《编译器参考指南》中第2-34 页的*--[no_]debug*

3.3.3 未使用节删除

未使用节删除是指从最终映像中删除不会用到的代码和数据。可以通过 --remove、--no_remove、--first、--last 和 --keep 链接器选项以及间接通过 --entry 来控制此优化。使用 --info unused 链接器选项可以指示链接器生成已删除的未使用节的列表。

在可能导致删除所有节的情况下，禁止删除未使用节。

在以下条件下，将在最终映像中保留输入节：

- 输入节包含入口点
- 通过非弱引用从包含入口点的输入节直接或间接引用输入节
- 输入节由 --first 或 --last 选项 （或等效的分散加载）指定为第一个或最后一个输入节
- 输入节由 --keep 选项标记为不可删除

注意

编译器通常会将函数和数据收集到一起，并为每个类别发出一个节。链接器只能删除完全不用的节。

3.3.4 未使用虚函数删除

虚函数删除 (VFE) 是对未使用节删除的细化，旨在减少由 C++ 代码生成的映像中的 ROM 大小。此优化可用于从代码中删除未使用的虚函数和 RTTI 对象。

在输入节包含多个函数的情况下，只有当*所有*函数都未使用时，才能删除该输入节。链接器无法从节*中*删除未使用的函数。在后面的部分中，可以假设函数在各自的节中进行编译。

未使用节删除可有效删除 C 代码中未使用的函数。在 C++ 应用程序中，虚函数和 RTTI 对象由指针表 （称为 vtable）引用。如果没有额外信息，则链接器无法确定在运行时要访问哪些 vtable 条目。这意味着链接器采用的标准未使用节删除算法无法保证删除未使用虚函数和 RTTI 对象。

ARM 编译器和链接器利用 VFE 相互协作，编译器提供有关未使用虚函数的附加信息，而链接器将使用这些信息。依此分析，链接器能够删除未使用的虚函数和 RTTI 对象。

注意

只要汇编器源文件未引用 C++ 库，便无需 VFE 注释。这是因为链接器假设未引用 C++ 库的对象文件没有进行任何虚函数调用。类似地，只要使用旧版本 armcc 编译的 C 源文件未引用 C++ 库，便可以参与 VFE。

VFE 有四种运行模式：

- On
- Off
- Force
- Force no RTTI

有关详细信息，请参阅 《链接器参考指南》中第2-60 页的*--vfemode=mode*。

编译器将附加信息放置在名称以 .arm_vfe 开头的节中。不支持 VFE 的链接器会忽略这些节。

3.4 反馈

本节介绍了如何使用链接器反馈进行重复编译。

armlink 针对下一次文件编译提供反馈信息，提示编译器有关未使用函数的情况。这些函数将放置在各自的节中，以便链接器将来删除它们。

当启用链接器的 --inline 优化时 （请参阅第3-20 页的*内联*），链接器内联的函数也发出到反馈文件中。这些函数也放置在各自的节中。

--feedback *file* 选项生成一个反馈文件，其中以备注形式包含每个输出文件名，并包含在文件中找到的未使用符号，例如：

```
;#<FEEDBACK># ARM Linker, RVCT ver [Build num]: Last Updated: Date
;VERSION 0.2
;FILE foo.o
unused_func1 <= USED 0
inlined_func <= LINKER_INLINED
;FILE bar.o
unused_func2 <= USED 0
```

下次编译源时，使用编译器选项 --feedback *file* 可指定要使用的链接器生成的反馈文件。如果不存在反馈文件，则编译器将发出一则警告消息。

3.4.1 示例

要查看链接器反馈的工作方式，请执行以下操作：

1. 创建一个 fb.c 文件，其中包含示例 3-1 中显示的代码。

示例 3-1 反馈示例

```
#include <stdio.h>

void legacy()
{
    printf("This is a legacy function, that is no longer used.\n");
}

int cubed(int i)
{
    return i*i*i;
}

void main()
{
```

```
    int n = 3;
    printf("%d cubed = %d\n",n,cubed(n));
}
```

2. 编译器 （忽略指示反馈文件不存在的警告）：

   ```
   armcc --asm -c --feedback fb.txt fb.c
   ```

   缺省情况下，这将内联 cubed() 函数并创建汇编器文件 fb.s 和对象文件 fb.o。在汇编器文件中， legacy() 和 cubed() 的代码仍存在。由于此内联操作， main 中没有 cubed() 的调用。

   由于 cubed() 未声明为 **static**，因此将保留其外联副本。

3. 使用以下命令行链接对象文件以创建链接器反馈文件：

   ```
   armlink --info sizes --list fbout1.txt --feedback fb.txt fb.o -o fb.axf
   ```

   链接器诊断将输出到文件 fbout1.txt 中。

   链接器反馈文件在注释中标识包含未使用函数的源文件 （未被编译器使用），并且包括 legacy() 和 cubed() 函数的条目：

   ```
   ;#<FEEDBACK># ARM Linker, RVCT ver [Build num]: Last Updated: Date
   ;VERSION 0.2
   ;FILE fb.o
   cubed <= USED 0
   legacy <= USED 0
   ```

   它显示了未使用的函数。

4. 重新编译并将各个阶段与不同的诊断文件链接：

   ```
   armcc --asm -c --feedback fb.txt fb.c
   armlink --info sizes --list fbout2.txt fb.o -o fb.axf
   ```

5. 比较两个诊断文件 fbout1.txt 和 fbout2.txt，查看映像组件 （例如，代码、 RO 数据、 RW 数据和 ZI 数据）的大小。代码组件较小。

   在汇编器文件 fb.s 中， legacy() 和 cubed() 函数不再位于主 .text 区域中。这些函数将编译到各自的 ELF 节中。因此， armlink 可以从最终映像中删除 legacy() 和 cubed() 函数。

注意

要从链接器反馈获得最大益处，必须执行至少两次完全编译和链接。不过，通常使用上次编译中的反馈执行单次编译和链接就足够了。

3.5 RW 数据压缩

本节介绍链接器对于数据压缩的优化。

RW 数据区通常包含大量的重复值 （例如 0），使其适合于压缩。缺省情况下，将会启用 RW 数据压缩以便最大程度地减小 ROM 大小。

ARM 库包含一些解压缩算法，链接器会从中选择最佳的一个添加到映像，在执行映像时用于解压缩数据区域。不过，您可以覆盖链接器选择的算法。

3.5.1 选择压缩器

在选择最适当的压缩算法以生成最小映像之前，armlink 会收集有关数据节内容的信息。如果适合于压缩，链接器将只使用一个数据压缩器来处理映像中的所有可压缩数据节，并对这些节尝试用不同的压缩方法以便生成最佳的整体规模。如果满足以下条件，可以自动进行压缩：

```
Compressed data size + Size of decompressor < Uncompressed data size
```

选定压缩器后，armlink 会将解压缩器添加到映像的代码区域。如果最终映像不包含任何压缩的数据，则不添加解压缩器。

您可以通过以下方式覆盖链接器使用的压缩方法：

- 使用 --datacompressor off 选项以关闭压缩功能
- 指定您选择的压缩器。

使用命令行选项 --datacompressor list 获取链接器中可用压缩器的列表，例如：

```
Num     Compression algorithm
=========================================================
0       Run-length encoding
1       Run-length encoding, with LZ77 on small-repeats
2       Complex LZ77 compression
```

3.5.2 如何应用压缩？

行程长度压缩可将数据编码为非重复字节和重复的零字节。非重复字节按原样输出，后面跟随零字节的个数。Limpel-Ziv 1977 (LZ77) 压缩可跟踪已看到数据的最后 n 个字节，遇到重复短语时，便会输出一对值，分别对应于该短语先前在数据缓冲区中出现的位置以及短语的长度。

要指定压缩器，请在链接器命令行上使用所需的 ID，例如：

```
armlink --datacompressor 2 ...
```

选择压缩器时，您必须注意：

- 压缩器 0 适合压缩包含大量零字节、而非零字节很少的数据。
- 压缩器 1 适合压缩有重复的非零字节的数据。
- 压缩器 2 适合压缩包含重复值的数据。

链接器会优先选择压缩器 0 或 1 来压缩包含大量零字节 (>75%) 的数据。如果数据中包含的零字节非常少 (<10%)，则选择压缩器 2。如果映像仅由 ARM 代码组成，则会自动使用 ARM 解压缩器。如果映像包含任何 Thumb 代码，则使用 Thumb 解压缩器。如果无法明确选择，则会试用所有压缩器，以便生成最佳的整体规模。（请参阅第3-15 页的*选择压缩器*）。

注意

您无法将自己的压缩器添加至链接器。可用算法及链接器选择使用这些算法的方式在将来可能会有所变动。

3.5.3 使用 RW 数据压缩

使用 RW 数据压缩时：

- 使用链接器选项 `--map` 可查看压缩已应用于代码中的哪些区。
- 如果使用 `Load$$`*`region_name`*`$$Base` 符号，其中 *`region_name`* 跟在包含同一加载区中的压缩数据的任何执行区后面，则链接器不应用压缩。
- 如果使用了带片上高速缓存的 ARM 处理器，则在解压缩后启用高速缓存，以避免出现代码一致性问题。

  有关详细信息，请参阅 《开发指南》中第 3 章 *嵌入式软件开发*。

RW 数据压缩使用库代码，例如必须放置在根区中的 `__dc*.o`。最好通过在分散加载描述文件中使用 `InRoot$$Sections` 来实现此目的。

如果使用分散加载描述文件，则指定不得通过添加 `NOCOMPRESS` 属性来压缩加载区或执行区。有关详细信息，请参阅第 3 章 *分散加载描述文件的形式语法*。

3.6 中间代码

中间代码是由链接器生成并插入到您的程序中的小段代码。当跳转涉及超过当前状态的跳转范围的目标时，armlink 必须生成中间代码。

对于 ARM，BL 指令的范围为 32MB，对于 Thumb-2 为 16MB，对于 Thumb 为 4MB。因此，中间代码可以通过变成指令的中间目标来扩展跳转范围，然后将 PC 设置为目标地址。如果混合使用 ARM 和 Thumb，则中间代码也会更改处理器状态。

armlink 支持以下中间代码类型：

- ARM 跳转到 ARM
- ARM 跳转到 Thumb （交互操作中间代码）
- Thumb 跳转到 ARM （交互操作中间代码）
- Thumb 跳转到 Thumb

armlink 为每个中间代码创建一个名为 Veneer$$Code 的输出节。仅当其他现有中间代码都不能满足需求时才会生成中间代码。如果两个输入节包含目标相同的长跳转，则只生成一个中间代码。仅当两个节都可以到达该中间代码时才以这种方式共享中间代码。

如果使用 ARMv4T，则当跳转涉及 ARM 与 Thumb 之间的状态更改时，armlink 会生成中间代码。在 ARMv5 及更高版本中，使用 BLX 指令。

3.6.1 中间代码共享

您可以使用命令行选项 --no_veneershare 来指定不共享中间代码。这会将已创建的中间代码节的所有权分配给创建该中间代码的对象，从而使您能够从分散加载描述文件中的特定对象选择中间代码，例如：

```
LR 0x8000
{
    ER_ROOT +0
    {
          object1.o(Veneer$$Code)
    }
}
```

使用中间代码共享时，不能指定所属对象。因此，使用 --no_veneershare 选项可以提供更加一致的映像布局。这是以显著增加代码大小为代价的。

3.6.2 中间代码变体

根据您需要的跳转范围的不同，中间代码会有不同的变体：

- 内联中间代码：
  - 必须正好在目标节的范围之前插入中间代码
  - ARM-Thumb 交互操作的中间代码具有 256 字节的范围，因此函数代码必须出现在紧跟中间代码后面的第一个 256 字节中
  - Thumb-ARM 交互操作的中间代码具有 0 字节的范围，因此函数代码必须紧跟在中间代码后面
  - 内联中间代码始终与位置无关

  这些限制意味着不能使用分散加载描述文件将内联中间代码移出执行区。使用命令行选项 `--no_inlineveneer` 可防止生成内联中间代码。
- 短跳转中间代码：
  - ARM-Thumb 短跳转中间代码具有 4MB 的范围
  - 短跳转中间代码始终与位置无关。
- 长跳转中间代码：
  - ARM-Thumb 和 Thumb-ARM 交互操作中间代码均具有 2^{32} 字节的范围
  - `armlink` 将长跳转功能与状态更改功能相结合，因此所有长跳转中间代码同时也是交互操作中间代码
  - 长跳转中间代码是绝对的或与位置无关

使用中间代码时，请注意以下几点：

- 不能将所有中间代码集合到一个输入节中，因为生成的中间代码输入节可能不在其他输入节的范围之内。如果节不在寻址范围内，则不可能使用长跳转功能。
- 链接器自动生成与位置无关的中间代码变体。不过，因为此类中间代码大于非位置无关变体，所以链接器仅在必要时 （即当源和目标执行区均与位置无关且两者间密切相关的情况下）才生成此类中间代码。

生成中间代码是为了优化代码大小。因此，`armlink` 按以下优先顺序选择变体：

1. 内联中间代码。
2. 短跳转中间代码。
3. 长中间代码。

3.6.3 “PI 到绝对”中间代码

正常的调用指令将目标地址编码为调用地址的偏移量。执行从 PI 代码到绝对代码的调用时，无法在链接时计算偏移量，因此链接器必须插入长跳转中间代码。

使用 --piveneer 选项可以控制“PI 到绝对”中间代码的生成，缺省情况下将设置该选项。使用 --no_piveneer 关闭该选项时，如果检测到从 PI 代码到绝对代码的调用，链接器会生成一个错误。

3.6.4 在重叠执行区重复使用中间代码

通过分散加载描述文件可以创建重叠区 （也就是共享相同 RAM 区域的区）。链接器会尽可能重复使用中间代码，但在重叠的区中重复使用中间代码有一些限制。但是，重复使用时必须满足以下条件：

- 重叠执行区不能重复使用放在任何其他重叠执行区中的中间代码。
- 其他任何执行区都不能重复使用放在重叠执行区中的中间代码。

如果不满足这些条件，则创建新的中间代码，而不重复使用现有中间代码。除非已使用分散加载指示链接器将中间代码放在特定的位置，否则，中间代码始终放在包含需要使用该中间代码的调用的执行区中。这意味着：

- 对于重叠执行区，它的所有中间代码都包含在执行区中。
- 重叠执行区从不需要其他执行区中的中间代码。

3.7 内联

链接器可以内联小函数以替换指向该函数的跳转指令。若使链接器实现此功能，该函数（不带返回指令）必须能放入跳转指令的四个字节中。

使用以下命令行选项控制跳转内联：

- 第2-6 页的*--[no_]branchnop*
- 第2-28 页的*--[no_]inline*
- 第2-58 页的*--[no_]tailreorder*

如果启用跳转内联优化，则链接器会扫描映像中的每个函数调用，然后根据需要进行内联。当链接器找到适合进行内联的函数时，会使用被调用方函数中的指令替换函数调用。

链接器在删除任何未使用节之前应用跳转内联优化，以便同样可以删除不再调用的内联节。有关详细信息，请参阅第3-11 页的*未使用节删除*。

使用 `--info=inline` 命令行选项可列出所有内联的函数。有关详细信息，请参阅《链接器参考指南》中第2-25 页的*--info=topic[,topic,...]*。

3.7.1 影响内联的因素

以下因素会影响函数的内联方式：

- 链接器只处理最简单的情况，并且不内联读取或写入 PC 的任何指令，因为这样的指令依赖于函数的位置。
- 如果映像同时包含 ARM 和 Thumb 代码，则必须构建从另一种状态调用的函数以便进行交互操作。链接器可以内联最多包含两个 16 位 Thumb 指令的函数。但是，ARM 调用方只能内联包含单个 16 位 Thumb 指令或 32 位 Thumb-2 指令的函数。
- 链接器执行的操作取决于被调用方函数的大小以及调用方和被调用方的状态，如表 3-2 中所示。

表 3-2 内联小函数

调用方状态	被调用方状态	被调用方函数大小
ARM	ARM	4 到 8 字节
ARM	Thumb	2 到 6 字节
Thumb	Thumb	2 到 6 字节

- 为了执行内联，函数的最后一条指令必须是以下两者之一：

  ```
  MOV pc, lr
  ```

  或

  ```
  BX lr
  ```

  只由一个返回序列组成的函数可以作为 NOP 内联。

- 条件 ARM 指令只能在满足以下某条件时进行内联：
  - BL 上的条件与要内联的指令上的条件匹配。例如，BLEQ 只能内联类似于 ADDEQ 的具有匹配条件的指令。
  - BL 指令或要内联的指令是无条件执行的。无条件的 ARM BL 可以内联满足所有其他标准的任何有条件的或无条件的指令。如果 BL 指令是有条件的，则无法条件执行的指令将不能进行内联。
- IT 块的最后一条指令 BL 不能内联 16 位 Thumb 指令或 32 位 MRS、MSR 或 CPS 指令。这是因为 IT 块会更改其范围内的指令的行为，因此内联指令会更改程序的行为。

3.7.2 处理尾调用节

如第3-20 页的*影响内联的因素*中所述，如果有任何跳转包含的重定位解析为具有 NOP 的下一条指令，则链接器会替换该跳转。这意味着可以优化尾调用节（即以跳转指令结尾的节），使跳转目标紧跟尾调用节出现在执行区中，且跳转更改为 NOP。

通过使用命令行选项 --tailreorder 将尾调用节恰好移到其目标之前，可以利用这种行为。有关详细信息，请参阅 《链接器参考指南》中第2-58 页的*--[no_]tailreorder*。

使用 --info=tailreorder 命令行选项可显示有关尾调用优化的信息。有关详细信息，请参阅 《链接器参考指南》中第2-25 页的*--info=topic[,topic,...]*。

3.8 使用命令行选项创建简单映像

数个 RO、RW 和 ZI 类型的输入节可以组成一个简单的映像。这些输入节整合成 RO、RW 和 ZI 输出节。根据载入区和执行区中输出节的排列方式，有三种基础类型的简单映像：

类型 1 加载视图中有一个区，执行视图中有三个相邻区。使用 `--ro-base` 选项可创建此类型的映像。

有关详细信息，请参阅*类型 1，一个加载区和几个连续执行区*。

类型 2 加载视图中有一个区，执行视图中有三个不连续的区。使用 `--ro-base` 和 `--rw-base` 选项可创建此类型的映像。

有关详细信息，请参阅第3-23 页的*类型 2，一个加载区和几个不连续的执行区*。

类型 3 加载视图中有两个区，执行视图中有三个不连续的区。使用 `--ro-base`、`--rw-base` 和 `--split` 选项可创建此类型的映像。您也可以使用 `--rosplit` 选项将缺省载入区分为两个 RO 输出节，一个用于代码，另一个用于数据。

有关详细信息，请参阅第3-26 页的*类型 3，两个加载区和几个不连续的执行区*。

在所有这三种类型的简单映像中，最多允许有三个执行区：

- 第一个执行区包含 RO 输出节
- 第二个执行区包含 RW 输出节 （如果有）
- 第三个执行区包含 ZI 输出节 （如果有）。

这些执行区称为 RO、 RW 和 ZI 执行区。

也可以用分散加载描述文件创建简单映像。有关如何执行此操作的详细信息，请参阅第5-31 页的*简单映像的等效分散加载描述*。

3.8.1 类型 1，一个加载区和几个连续执行区

此类型的映像由加载视图中的单个载入区和内存映射中相邻放置的三个执行区组成。此方法适用于将程序加载到 RAM 中的系统，例如 OS 引导加载程序或桌面系统 （请参阅第3-23 页的图 3-3）。

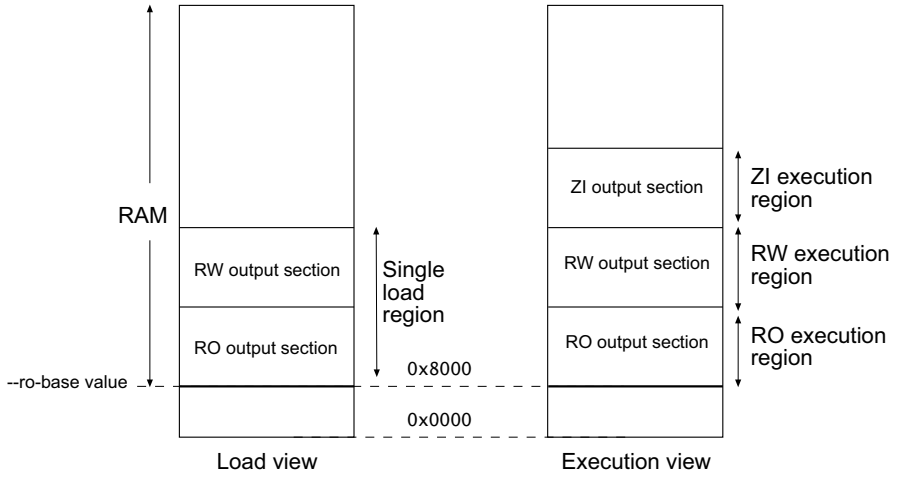

图 3-3 简单类型 1 映像

使用以下命令创建此类映像：

```
armlink --ro-base 0x8000
```

加载视图

单个载入区由连续放置的 RO 和 RW 输出节组成。RO 和 RW 执行区都是根区。加载时不存在 ZI 输出节。该节是在执行前使用映像文件中的输出节描述创建的。

执行视图

包含 RO、RW 和 ZI 输出节的三个执行区相邻排列。RO 和 RW 执行区的执行地址与其加载地址相同，因此不需要从加载地址向执行地址移动任何内容。不过，包含 ZI 输出节的 ZI 执行区是在运行时创建的。

使用 `armlink` 选项 `--ro-base` *address* 可指定包含 RO 输出的区的加载和执行地址。缺省地址是 `0x8000`，如图 3-3 中所示。

3.8.2 类型 2，一个加载区和几个不连续的执行区

此类型的映像由单个加载区和执行视图中的三个执行区组成。RW 执行区与 RO 执行区是不相邻的。例如，此方法适用于基于 ROM 的嵌入式系统 （请参阅第3-24 页的图 3-4），其中 RW 数据在启动时从 ROM 复制到 RAM。

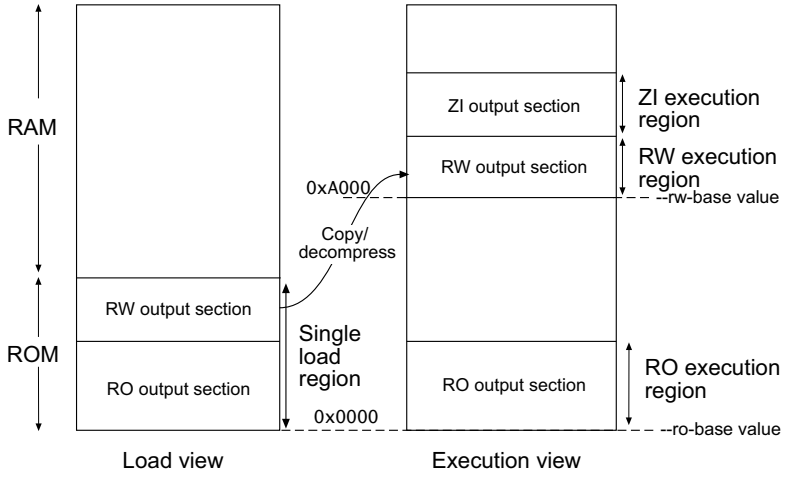

图 3-4 简单类型 2 映像

使用以下命令创建此类映像：

```
armlink --ro-base 0x0 --rw-base 0xA000
```

加载视图

在加载视图中，单个加载区由连续放置的 RO 和 RW 输出节 （例如在 ROM 中）组成。此处的 RO 区是根区， RW 区是非根区。加载时不存在 ZI 输出节。该节在运行时创建。

执行视图

在执行视图中，第一个执行区包含 RO 输出节，第二个执行区包含 RW 和 ZI 输出节。

包含 RO 输出节的区的执行地址与加载地址相同，因此不必移动 RO 输出节。即，它是根区。

包含 RW 输出节的区的执行地址与加载地址不同，因此 RW 输出节从 （单载入区）加载地址移到其执行地址 （第二个执行区中）。ZI 执行区及其输出节与 RW 执行区相邻放置。

使用 `armlink` 选项 `--ro-base` *address* 可指定 RO 输出节的加载地址和执行地址，使用 `--rw-base` *exec_address* 可指定 RW 输出节的执行地址。如果未使用 `--ro-base` 选项指定地址，则 `armlink` 将使用缺省值 `0x8000`。对于嵌入式系

统，--ro-base 值通常为 0x0。如果未使用 --rw-base 选项指定地址，则缺省情况下将 RW 放在紧靠 RO 的上方。（如第3-22 页的*类型 1，一个加载区和几个连续执行区*中所述）。

注意

RW 和 ZI 输出节的执行区不能与任何加载区重叠。

3.8.3 类型 3，两个加载区和几个不连续的执行区

此类型的映像与类型 2 的映像相似，只是单个加载区现在分成了两个根加载区。（请参阅图 3-5）。

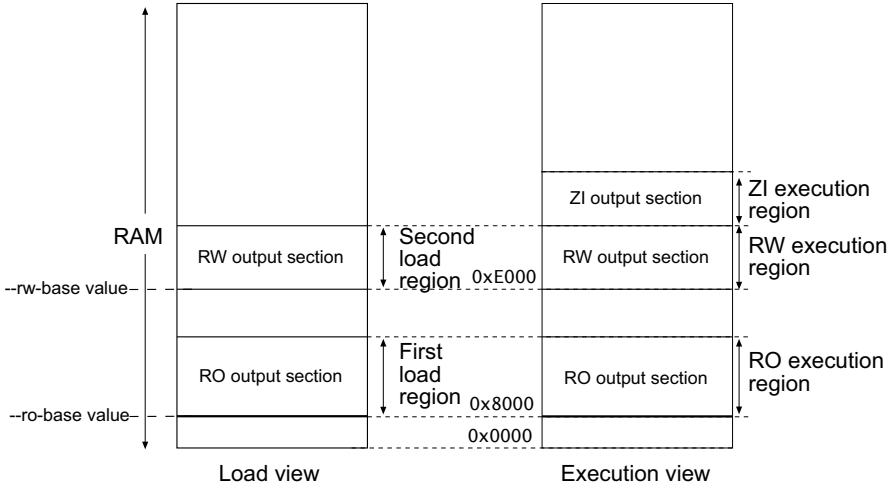

图 3-5 简单类型 3 映像

使用以下命令创建此类映像：

```
armlink --split --ro-base 0x8000 --rw-base 0xE000
```

加载视图

在加载视图中，第一个载入区由 RO 输出节组成，第二个载入区由 RW 输出区组成。加载时不存在 ZI 输出节。它是在执行前，使用映像文件中包含的输出节描述创建的。

执行视图

在执行视图中，第一个执行区包含 RO 输出节，第二个执行区包含 RW 和 ZI 输出节。

RO 区的执行地址与其加载地址相同，因此 RO 输出节的内容不必从加载地址移动或复制到执行地址。RO 和 RW 都是根区。

RW 区的执行地址也与其加载地址相同，因此 RW 输出节的内容不必从加载地址移到执行地址。不过， ZI 输出节在运行时创建，并且紧邻 RW 区放置。

使用以下链接器选项指定加载和执行地址：

--split　　将缺省的单加载区 （包含 RO 和 RW 输出节）分成两个根加载区（一个包含 RO 输出节，另一个包含 RW 输出节），以便可以使用 --ro-base 和 --rw-base 分别放置这两个加载区。

--ro-base *address*

指示 armlink 将包含 RO 节的区的加载和执行地址设置在 4 字节对齐的 *address* 处（例如，ROM 中第一个位置的地址）。如果未使用 --ro-base 选项指定地址，则 armlink 将使用缺省值 0x8000。

--rw-base *address*

指示 armlink 将包含 RW 输出节的区的执行地址设置在 4 字节对齐的 *address* 处。如果此选项与 --split 结合使用，则会同时指定 RW 区 （例如根区）的加载地址和执行地址。

3.9 使用命令行选项处理 C++ 异常

缺省情况下，或如果指定选项 --exceptions，则映像可以包含异常表。如果无代码抛出异常，则自动丢弃异常表。但是，如果已指定选项 --no_exceptions，并且在删除未使用节之后出现任何异常节，则链接器会生成一则错误消息。

如果需要确保您的代码不出现异常，请使用 --no_exceptions 选项。链接器将生成一则错误消息以突出显示已找到的异常，并且不生成最终映像。

不过，您可以将 --no_exceptions 选项和 --diag_warning 选项结合使用，将错误消息降级为警告。链接器会生成最终映像，但也会生成一则消息，警告您发现了异常。

链接器可以为包含调试帧信息的传统对象创建异常表。链接器可以为 C 和汇编语言对象安全地进行此操作。缺省情况下，链接器不创建异常表。这与使用链接器选项 --exceptions_tables=nocreate 的结果相同。

使用链接器选项 --exceptions_tables=unwind，链接器可以通过 .debug_frame 信息为不带异常表的映像中的每个节创建一个寄存器恢复展开表。如果无法创建，链接器将改为创建非展开表。

使用链接器选项 --exceptions_tables=cantunwind 可为不带异常表的映像中的每个节创建一个非展开表。

注意

有以下几点需要注意：

- 使用缺省设置 （即 --exceptions --exception_tables=nocreate）时，通过 C 代码或汇编代码抛出异常是不安全的 （除非使用 --exceptions 选项编译 C 代码）。
- 对于不带异常支持编译的 C++ 代码 （例如，使用 v2.1 之前的 RealView 编译工具编译的任何代码，或使用 --no_exceptions 选项编译的代码），链接器不能为其中的自动变量生成清除代码。

3.10 获得有关映像的信息

您可以使用 --info 选项来获得有关链接器如何生成映像的信息，例如：

```
armlink --info sizes ...
```

此处 sizes 提供映像中每个输入对象和库成员的代码和数据大小的列表。使用此选项相当于 --info sizes,totals。

有关详细信息，请参阅 《链接器参考指南》中第2-25 页的*--info=topic[,topic,...]*。

示例 3-2 显示了表格形式的输出，其中各个总和相互分开以方便阅读。

示例 3-2 映像信息

```
Code (inc. data)   RO Data   RW Data    ZI Data      Debug

3712        1580        19        44      10200       7436   Object Totals
0              0        16         0          0          0   (incl. Generated)
0              0         3         0          0          0   (incl. Padding)
21376        648       805         4        300      10216   Library Totals
0              0         6         0          0          0   (incl. Padding)

==============================================================================

Code (inc. data)   RO Data    RW Data    ZI Data      Debug
25088       2228       824         48      10500      17652   Grand Totals
25088       2228       824         48      10500      17652   ELF Image Totals
25088       2228       824         48          0          0   ROM Totals

==============================================================================

Total RO  Size (Code + RO Data)             25912 (  25.30kB)
Total RW  Size (RW Data + ZI Data)          10548 (  10.30kB)
Total ROM Size (Code + RO Data + RW Data)   25960 (  25.35kB)
```

Code (inc. Data)

显示代码占用了多少字节。在此映像中，有 3712 字节的代码。其中包括 1580 字节的内联数据 (inc. data)，例如，文字池和短字符串。

RO Data 显示只读数据占用了多少字节。这是除 Code (inc. data) 列中包括的内联数据之外的数据。

RW Data 显示读写数据占用了多少字节。

ZI Data 显示零初始化的数据占用了多少字节。

Debug 显示调试数据占用了多少字节，例如，调试输入节以及符号和字符串表。

Object Totals

显示链接到一起以生成映像的对象占用了多少字节。

(incl. Generated)

armlink 会生成映像内容，例如，交互操作中间代码和输入节 （如区表）。如果 Object Totals 行包含此类型的数据，则会显示在该行中。在第3-29 页的示例 3-2 中，共有 19 字节的 RO 数据，其中 16 字节是链接器生成的 RO 数据。

Library Totals

显示已提取并作为单个对象添加到映像中的库成员占用了多少字节。

(incl. Padding)

armlink 将根据需要插入填充，以强制节对齐 （请参阅第3-8 页的 *对齐节*）。如果 Object Totals 行包括此类型的数据，则会显示在相关联的 (incl. Padding) 行中。类似地，如果 Library Totals 行包括此类型的数据，则会显示在相关联的行中。在第3-29 页的示例 3-2 中，对象总计中有 19 字节的 RO 数据，其中 3 字节是链接器生成的填充；而库总计中有 805 字节的 RO 数据，其中带有 6 字节的填充。

Grand Totals

显示映像的真实大小。在第3-29 页的示例 3-2 中，有 10200 字节的 ZI 数据 （在 Object Totals 中）和 300 字节的 ZI 数据 （在 Library Totals) 中），总共为 10500 字节。

ELF Image Totals

如果您使用 RW 数据压缩 （缺省值）优化 ROM 大小，则最终映像的大小会更改，并且这会在 --info 生成的输出中反映出来。比较 Grand Totals 和 ELF Image Totals 下的字节数，查看压缩效果。

在第3-29 页的示例 3-2 中，未启用 RW 数据压缩。如果对数据进行压缩，则 RW 值会更改。有关详细信息，请参阅第3-15 页的*RW 数据压缩*。

ROM Totals

显示包含映像所需的 ROM 的最小大小。这不包括 ZI 数据和存储在 ROM 中的调试信息。

3.10.1 使用与映像相关的信息

您可以使用 `--map` 选项创建映像映射。这包括映像中每个加载区、执行区和输入节的地址和大小，并显示应用 RW 数据压缩的方式。有关详细信息，请参阅《链接器参考指南》中第2-37 页的*--[no_]map*。

您可以使用 `--info inputs` 标识某些链接错误的起源。例如，您可以搜索输出，以便查找库对象中未定义的引用，或者查找由于重定向某些库函数而不重定向其他函数而导致的多重定义的符号。从该输出的结尾开始反向搜索，查找并解决链接错误。还可以使用 `--verbose` 选项输出包含有关链接器操作的附加信息的类似文本。

3.11 库搜索、选择和扫描

链接器向映像添加对象文件的方式与向映像添加库的方式的区别在于：

- 不论是否被引用，输入列表中的每个对象文件都无条件地添加到输出映像。必须至少指定一个对象。
- 仅当对象文件或已包含的库成员对某库成员进行非弱引用时，或者明确指示链接器添加该库成员时，该库成员才包含在输出中。

  注意

  如果在输入文件列表中明确请求库成员，则即使它不解析任何当前引用，也会载入它。在这种情况下，明确请求的成员被视为普通对象。

除非使用了 `--no_remove` 或 `--keep`，否则随后会删除未使用的节。

未解析的弱符号引用不会导致加载库成员。

注意

如果指定 `--no_scanlib` 命令行选项，则链接器不搜索缺省的 ARM 库，而仅使用输入文件列表中指定的库来解析引用。

因此，链接器会如下创建内部库列表：

1. 将在输入文件列表中明确指定的所有库都添加到列表中。
2. 检查用户指定的搜索路径来标识 ARM 标准库，以便满足输入对象中嵌入的请求。

   从搜索目录及其子目录中选择最适合的库变体。ARM 提供的库有多个变体，它们根据成员的属性进行命名。

3.11.1 搜索 ARM 库

您可以指定用于查找 ARM 标准库的搜索路径，方法是：

- 使用环境变量 RVCT40LIB。这是缺省设置。
- 将 --libpath 选项添加到 armlink 命令行，并在命令行中包含用逗号分隔的父目录列表。

  此列表必须以 ARM 库目录 armlib 和 cpplib 的父目录结束。RVCT40LIB 环境变量中保存此路径。

  注意

  链接器命令行选项 --libpath 将覆盖由 RVCT40LIB 变量指定的路径。

链接器将 --libpath 或 RVCT40LIB 变量给定的每个父目录与输入对象中的每个子目录请求组合在一起，并标识搜索 ARM 库的位置。将使用格式为 Lib$$Request$$*sub_dir_name* 的符号，将父目录中 ARM 子目录的名称放在每个编译的对象中。

如果两个或更多个库定义了同一个符号，则搜索的有序性可确保链接器选择列表中较早出现的库。

选择 ARM 库变体

根据其成员对象的属性，有不同的 ARM 库变体。ARM 库的变体被编码为库名称。链接器必须从库搜索时识别的每个目录中选择最适合的变体。

链接器收集每个输入对象的属性，然后选择最适合那些属性的库变体。如果多个选定的库同样适合，则链接器保留第一个选定的库而拒绝其他所有的库。

最终的列表中包含链接器为了解析引用而扫描的所有库。

有关库变体的详细信息，请参阅 《库指南》中的第 2 章 *C 和 C++ 库*。

3.11.2 搜索用户库

您可以按下述方法指定用户库：

- 将其明确包含在输入文件列表中
- 将 --userlibpath 选项添加到 armlink 命令行，并在命令行中包含以逗号分隔的目录列表和作为输入文件的库的名称。

如果未在命令行中指定库的完整路径名，链接器会尝试在 --userlibpath 选项指定的目录中查找库。例如，如果目录 /mylib 中包含 my_lib.a 和 other_lib.a，则使用以下命令将 /mylib/my_lib.a 添加到输入文件列表：

```
armlink --userlibpath /mylib my_lib.a *.o
```

如果从库中添加一个特定成员，则不会将库添加到链接器使用的可搜索库的列表中。要加载特定成员*并*将库添加到可搜索库的列表中，请包括库 *filename* 本身，同时指定 *library(member)*。例如，要加载 strcmp.o 并将 mystring.lib 放在可搜索库列表中，请将以下库添加到输入文件列表中：

```
mystring.lib(strcmp.o) mystring.lib
```

注意

由 RVCT40LIB 环境变量或链接器命令行选项 --libpath 指定的、用于搜索 ARM 标准库的搜索路径不会用于搜索用户库 （请参阅第3-33 页的*搜索 ARM 库*）。

3.11.3 扫描库

当链接器构建库列表后，它将重复扫描列表中的每个库以解析引用。

当搜索所有目录，选择最适合的库变体并将它们添加到库列表之后，将扫描每个库以加载所需的成员：

1. 对于每个当前未满足的非弱引用，链接器将顺序搜索库列表，以查找匹配的定义。找到的第一个定义将标记为用于步骤 2。
   如果两个或更多个库定义了同一个符号，则搜索的有序性可确保链接器选择列表中较早出现的库。这样可以覆盖其他库中的函数定义，例如，通过将库添加到输入文件列表以覆盖 ARM C 库。但是，您必须谨慎，在覆盖库成员中的所有符号时应保持一致，否则行为将不可预测。
2. 加载步骤 1 中标记的库成员。加载每个成员时，它可能满足某些未解析的引用 （也许包括弱引用）。加载库成员也可能会创建新的未解析的弱引用和非弱引用。
3. 继续步骤 1 和 2 中的过程，直到所有非弱引用都得到解析，或者不能被任何库解析。

如果在扫描操作结束时仍有任何非弱引用未满足，则链接器将生成错误消息。

第 4 章
访问映像符号

本章介绍了如何使用 ARM® 链接器 armlink 来引用符号。本章分为以下几节：

- 第4-2 页的*ARM/Thumb 同义词*
- 第4-3 页的*访问链接器定义的符号*
- 第4-9 页的*访问其他映像中的符号*
- 第4-12 页的*隐藏和重命名全局符号*
- 第4-14 页的*使用 $Super$$ 和 $Sub$$ 覆盖符号定义*
- 第4-15 页的*符号版本控制*

4.1 ARM/Thumb 同义词

链接器允许符号的多个定义在映像中共存，但前提是每个定义与不同处理器状态相关联。当使用 ARM/Thumb 同义词引用符号时，armlink 将应用以下规则：

- 针对处于 ARM 状态的符号执行的 B、BL 或 BLX 指令将解析为 ARM 定义。
- Thumb 状态下对符号执行的 B、BL 或 BLX 指令将解析为 Thumb 定义。

任何其他符号引用将解析为链接器遇到的第一个定义。在这种情况下，armlink 将显示警告，说明指定了所选的符号。

4.2 访问链接器定义的符号

链接器定义了一些包含 $$ 字符序列的符号。这些符号和所有其他包含 $$ 序列的外部名称都是 ARM 的保留名称。

汇编语言程序可以导入这些符号地址并将其用作可重定位的地址，或者从 C 或 C++ 源代码中将其作为 **extern** 符号进行引用。有关详细信息，请参阅第4-8 页的*导入链接器定义的符号*。

注意

仅当代码引用链接器定义的符号时，才会生成这些符号。

4.2.1 与区相关的符号

当链接器创建映像时，会生成与区相关的符号。

Image$$ 执行区符号

表 4-1 显示了链接器为映像中存在的每个执行区生成的符号。表 4-1 中的所有符号都在初始化 C 库后引用执行地址。

表 4-1 Image$$ 执行区符号

符号	说明
Image$$*region_name*$$Base	区的执行地址。
Image$$*region_name*$$Length	执行区长度 （以字节为单位），不包括 ZI 长度。
Image$$*region_name*$$Limit	执行区中非 ZI 部分末尾后面的字节的地址。
Image$$*region_name*$$RO$$Base	此区中的 RO 输出节的执行地址。
Image$$*region_name*$$RO$$Length	RO 输出节的长度 （以字节为单位）。
Image$$*region_name*$$RO$$Limit	执行区中 RO 输出节末尾后面的字节的地址。
Image$$*region_name*$$RW$$Base	此区中的 RW 输出节的执行地址。
Image$$*region_name*$$RW$$Length	RW 输出节的长度 （以字节为单位）。
Image$$*region_name*$$RW$$Limit	执行区中 RW 输出节末尾后面的字节的地址。

表 4-1 Image$$ 执行区符号 （续）

符号	说明
Image$$*region_name*$$ZI$$Base	此区中的 ZI 输出节的执行地址。
Image$$*region_name*$$ZI$$Length	ZI 输出节的长度 （以字节为单位）。
Image$$*region_name*$$ZI$$Limit	执行区中 ZI 输出节末尾后面的字节的地址。

Load$$ 执行区符号

链接器为在 RW 压缩后引用加载地址的重定位执行一轮额外的地址分配和重定位。通过这种延迟的重定位，在链接器定义的符号中可以使用有关加载地址的更多信息。

表 4-2 显示了链接器为映像中存在的每个 Load$$ 执行区生成的符号。此表中的所有符号都在初始化 C 库之前引用加载地址。有以下几点需要注意：

- 这些符号是绝对的，因为与节相关的符号只能具有执行地址。
- 这些符号会考虑 RW 压缩。
- 这些符号不包括 ZI 输出节，因为该节在初始化 C 库之前不存在。
- 经过 RW 压缩的执行区中的所有重定位都必须在压缩之前执行，因为链接器无法对压缩数据解析延迟的重定位。
- 如果链接器检测到从经过 RW 压缩的区到依赖于 RW 压缩的链接器定义符号的重定位，则链接器对该区禁用压缩。
- 写入到文件的任何零字节都可见。因此，限制 (Limit) 和长度 (Length) 值必须考虑写入到文件中的零字节。

表 4-2 Load$$ 执行区符号

符号	说明
Load$$*region_name*$$Base	区的加载地址。
Load$$*region_name*$$Length	加载区长度 （以字节为单位）。
Load$$*region_name*$$Limit	执行区末尾后面的字节的地址。
Load$$*region_name*$$RO$$Base	此执行区中的 RO 输出节的地址。
Load$$*region_name*$$RO$$Length	RO 输出节的长度 （以字节为单位）。

表 4-2 Load$$ 执行区符号 （续）

符号	说明
Load$$*region_name*$$RO$$Limit	执行区中 RO 输出节末尾后面的字节的地址。
Load$$*region_name*$$RW$$Base	此执行区中的 RW 输出节的地址。
Load$$*region_name*$$RW$$Length	RW 输出节的长度 （以字节为单位）。
Load$$*region_name*$$RW$$Limit	执行区中 RW 输出节末尾后面的字节的地址。

Load$$LR$$ 加载区符号

表 4-3 显示了链接器为映像中存在的每个 Load$$LR$$ 加载区生成的符号。一个 Load$$LR$$ 加载区可以包含许多执行区，因此没有单独的 $$RO 和 $$RW 组件。

表 4-3 Load$$LR$$ 加载区符号

符号	说明
Load$$LR$$*load_region_name*$$Base	加载区的地址。
Load$$LR$$*load_region_name*$$Length	加载区的长度。
Load$$LR$$*load_region_name*$$Limit	加载区末尾后面的字节的地址。

非分散加载时的区名称值

如果未使用分散加载，链接器将使用以下项的 *region_name* 值：

- ER_RO，适用于只读执行区
- ER_RW，适用于读写执行区
- ER_ZI，适用于零初始化的执行区。

注意

- 映像的 ZI 输出节不是静态创建的，而是在运行时自动动态创建的。因此，ZI 输出节没有加载地址符号。
- 建议优先使用与区相关的符号，而不是与节相关的符号。

使用分散加载描述文件

如果使用分散加载，描述文件将命名映像中的所有执行区，并提供其加载和执行地址。

如果描述文件定义了堆栈和堆，链接器还会生成特殊堆栈和堆符号。

有关详细信息，请参阅第 5 章 *使用分散加载描述文件*。

将堆栈和堆放在 ZI 区上面

与区相关的符号的一个常见用途是，将堆直接放在 ZI 区上面。示例 4-1 说明了如何使用汇编语言创建改变目标的 __user_initial_stackheap() 版本。该示例假设使用 ARM C 库中的缺省单区内存模型。

请参阅 《库指南》中第2-69 页的*__user_initial_stackheap()*。

示例 4-1 将堆栈和堆放在 ZI 区上面

```
    EXPORT __user_initial_stackheap
    IMPORT ||Image$$region_name$$ZI$$Limit||
__user_initial_stackheap
    LDR r0, =||Image$$region_name$$ZI$$Limit||
    MOV pc, lr
```

4.2.2 与节相关的符号

如果使用命令行选项创建简单映像，则会生成表 4-4 中所示的输出节符号。简单映像有三个输出节 （RO、 RW 和 ZI），它们生成三个执行区。对于映像中存在的每个输入节，链接器将生成第4-8 页的表 4-5 中所示的输入符号。

链接器先按 RO、 RW 或 ZI 属性对执行区内的节进行排序，然后按名称进行排序。例如，将所有 .text 节放在一个连续块中。包含相同属性和名称的节的连续块称为*合并节*。

映像符号

如果使用分散加载描述文件，则不会定义表 4-4 中的输出节符号。如果代码访问这些符号，则必须将其视为弱引用。

__user_initial_stackheap() 的标准实现使用 Image$$ZI$$Limit 中的值。因此，如果使用分散加载描述文件，则必须手动放置堆栈和堆。可以在分散描述文件中实现此目的，或是通过重新实现 __user_initial_stackheap() 设置堆和堆栈边界来实现。有关详细信息，请参阅第 5 章 *使用分散加载描述文件*。

表 4-4 与映像相关的符号

符号	节类型	说明
Image$$RO$$Base	输出	RO 输出节的起始地址。
Image$$RO$$Limit	输出	RO 输出节末尾后面的第一个字节的地址。
Image$$RW$$Base	输出	RW 输出节的开始地址。
Image$$RW$$Limit	输出	ZI 输出节末尾后面的字节的地址。 （选择 ZI 区末尾而不是 RW 区末尾是为了与遗留代码之间保持兼容。）
Image$$ZI$$Base	输出	ZI 输出节的开始地址。
Image$$ZI$$Limit	输出	ZI 输出节末尾后面的字节的地址。

输入节符号

如果代码引用输入节符号，则假设您希望将映像中所有具有相同名称的输入节连续放在映像内存映射中。如果分散加载描述以非连续方式放置这些输入节，则链接器会诊断出错误，因为在不连续的内存上使用 base 和 limit 符号通常会产生不可预见和不可取的结果。

表 4-5 与节相关的符号

符号	节类型	说明
SectionName$$Base	输入	名为 *SectionName* 的合并节的开始地址。
SectionName$$Length	输入	名为 *SectionName* 的合并节的长度 （以字节为单位）。
SectionName$$Limit	输入	名为 *SectionName* 的合并节末尾后面的字节的地址。

4.2.3 导入链接器定义的符号

可以使用两种方法，将链接器定义的符号导入到 C 或 C++ 源代码中。请使用以下方法之一：

```
extern unsigned int symbol_name;
```

或

```
extern char symbol_name[];
```

如果将符号声明为 int，则必须使用取址运算符获取正确的值，如示例 4-2 中所示。

示例 4-2 导入链接器定义的符号

```
extern unsigned int Image$$ZI$$Length;
extern char Image$$ZI$$Base[];
memset(Image$$ZI$$Base,0,(unsigned int)&Image$$Length);
```

4.3 访问其他映像中的符号

如果要使一个映像能够识别另一个映像的全局符号值，可以使用*符号定义* (symdefs) 文件。

例如，如果一个映像始终位于 ROM 中并且将多个映像加载到 RAM 中，则可以使用这种方法。加载到 RAM 中的映像可以访问位于 ROM 中的映像的全局函数和数据。

4.3.1 创建 symdefs 文件

使用 `armlink` 选项 `--symdefs=filename` 可生成 symdefs 文件。

链接器将在成功完成的最后链接阶段生成 symdefs 文件。部分链接或最后失败的链接不会生成此文件。

注意

如果 *filename* 不存在，创建的文件将包含所有全局符号。如果 *filename* 存在，则使用 *filename* 的现有内容来选择链接器重写文件时输出的符号。这意味着只更新 filename 中的现有符号，不添加任何新符号 （如果有）。如果不希望出现此行为，请确保在链接步骤之前删除所有现有的 symdefs 文件。

输出全局符号的子集

缺省情况下，所有全局符号都写入到 symdefs 文件中。

如果 *filename* 存在，链接器将使用其内容将输出限制为全局符号的一个子集。要限制输出符号，请执行以下操作：

1. 在执行 *image1* 将近最后的链接时，指定 `--symdefs=filename`。链接器将创建 symdefs 文件 *filename*。
2. 在文本编辑器中打开 *filename*，删除不希望出现在最终列表中的所有符号条目，然后保存文件。
3. 在执行 *image1* 的最终链接时，指定 `--symdefs=filename`。

   可以随时编辑 *filename* 以添加注释，然后重新链接 *image1*，例如，在用于创建 *image1* 的一个或多个对象发生变化后更新符号定义。

4.3.2 读取 symdefs 文件

可以将 symdefs 文件视为一个对象文件，其中包含符号信息，但不包含代码或数据。就像任何对象文件一样，要读取 symdefs 文件，请将其添加到文件列表中。链接器将读取该文件，并将符号及其值添加到输出符号表中。添加的符号具有 ABSOLUTE 和 GLOBAL 属性。

如果执行部分链接，则会将符号添加到输出对象符号表中。如果执行完整链接，则会将符号添加到映像符号表中。

对于文件中的无效行，链接器将生成错误消息。无效行是指：

- 缺少任何列的行；
- 任何列具有无效值的行。

从 symdefs 文件中提取的符号的处理方式与从对象符号表中提取的符号完全相同。有关多个符号定义和 ARM/Thumb 同义词的限制同样适用。

4.3.3 symdefs 文件格式

symdefs 文件包含符号及其值。

该文件由标识行、可选注释和符号信息组成，如示例 4-3 中所示。

示例 4-3 symdefs 文件格式

```
#<SYMDEFS># ARM Linker, RVCTver [Build num]: Last Updated: Date
;value type name, this is an added comment
0x00008000 A __main
0x00008004 A __scatterload
0x000080E0 T main
0x0000814D T _main_arg
0x0000814D T __argv_alloc
0x00008199 T __rt_get_argv
...
   # This is also a comment, blank lines are ignored
...
0x0000A4FC D __stdin
0x0000A540 D __stdout
0x0000A584 D __stderr
0xFFFFFFFD N __SIG_IGN
```

标识字符串

如果文本文件的前 11 个字符是 #<SYMDEFS>#，则链接器将该文件识别为 symdefs 文件。

标识字符串后面是链接器版本信息以及最近更新 symdefs 文件的日期和时间。版本和更新信息不是标识字符串的一部分。

注释

可以使用文本编辑器手动插入注释。注释具有以下属性：

- 第一行的开头必须为特殊标识注释 #<SYMDEFS>#。此注释是链接器在生成该文件时插入的，不能手动将其删除。
- 如果任何行的第一个非空白字符是分号 (;) 或井号 (#)，则该行为注释。
- 位于第一个非空白字符之后的分号 (;) 或井字 (#) 不作为注释开始字符。
- 空行将被忽略，因此可以插入空行以提高可读性。

符号信息

符号信息是通过放在一行中的符号地址、类型和名称提供的：

符号值 链接器以固定十六进制格式写入符号的绝对地址，例如，0x00008000。如果编辑该文件，您可以使用十六进制或十进制格式的地址值。

类型标记 用于说明符号类型的单个字母：

A ARM 代码

T Thumb 代码

D 数据

N 数字。

符号名 符号名称。

4.4 隐藏和重命名全局符号

本节介绍了如何使用控制文件来管理输出文件中的符号名称。例如，可以使用控制文件保护知识产权，或者避免发生命名空间冲突。控制文件是一个文本文件，其中包含一组用于编辑输出对象符号表的命令。

使用 armlink 命令行选项 --edit *file-list* 可指定控制文件。如果指定多个控制文件，可以使用以下语法之一：

```
armlink --edit file1 --edit file2 --edit file3
```

```
armlink --edit file1,file2,file3
```

不要在逗号和文件名之间留空格。有关详细信息，请参阅第2-18 页的*--edit=file_list*。

4.4.1 控制文件格式

控制文件是采用以下格式的纯文本文件：

- 第一个非空白字符是分号 (;) 或井号 (#) 的行会解释为注释。注释将作为空行处理。
- 空行将被忽略。
- 每个非空的非注释行要么是命令，要么是命令 （拆分为连续的非空行）的一部分。
- 以逗号 (,) 作为最后一个非空白字符结束的命令行会在下一个非空行继续。

每个命令行由一个命令以及后面的一组或多组以逗号分隔的操作数组成。每组操作数由一个或两个操作数组成，具体取决于命令。命令适用于其中的每组操作数。以下规则适用：

- 命令不区分大小写，但按惯例以大写字母显示。
- 操作数区分大小写，因为它们必须与区分大小写的符号名相匹配。您可以在操作数中使用通配符。

命令仅适用于全局符号。其他符号 （如局部符号）不受影响。

有关控制文件命令的详细信息，请参阅 《链接器参考指南》中的以下各节：

- 第2-66 页的*IMPORT*
- 第2-64 页的*EXPORT*
- 第2-67 页的*RENAME*

- 第2-69 页的*RESOLVE*
- 第2-68 页的*REQUIRE*
- 第2-65 页的*HIDE*
- 第2-71 页的*SHOW*

4.5 使用 $Super$$ 和 $Sub$$ 覆盖符号定义

在某些情况下，无法修改现有符号，例如，由于符号位于外部库或 ROM 代码中。

可以使用 $Super$$ 和 $Sub$$ 模式来修补现有符号。

例如，要修补函数 foo() 的定义，请按如下方式使用 $Super$$foo() 和 $Sub$$foo()：

$Super$$foo　标识未修补的原始函数 foo()。使用它可以直接调用原函数。

$Sub$$foo　标识调用的新函数，而不是原始函数 foo()。可以使用此模式在原始函数之前或之后添加处理。

注意

$Sub 和 $Super 机制只在静态链接时起作用， $Super$$ 引用无法导入或导出到动态符号表中。

示例 4-4 说明修改遗留函数 foo() 而导致调用 ExtraFunc() 和 foo()。有关详细信息，请参阅 《ARM 体系结构的 ELF》。

示例 4-4 使用 $Super$$ 和 $Sub$$

```
extern void ExtraFunc(void);
extern void $Super$$foo(void):

/* this function is called instead of the original foo() */
void $Sub$$foo(void)
{
  ExtraFunc();    /* does some extra setup work */
  $Super$$foo();  /* calls the original foo() function */
}
```

4.6 符号版本控制

符号版本控制记录有关从动态共享对象中导入或由其导出的符号的额外信息。动态加载程序使用此额外信息来确保，可以在加载时使用映像所需的所有符号。

通过进行符号版本控制，共享对象创建者可以生成新的符号版本以供所有新客户机使用，同时与链接到旧版本的共享对象上的客户机保持兼容。

4.6.1 版本

符号版本控制将*版本*概念添加到动态符号表中。版本是与符号关联的名称。当动态加载程序尝试解析与版本名称关联的符号引用时，它只能与具有相同版本名称的符号定义相匹配。

注意

版本可能与以前的版本名称相关联以显示共享对象的修订历史记录。

4.6.2 缺省版本

尽管共享对象可能具有同一符号的多个版本，但共享对象的客户机只能绑定到最新版本上。

它称为符号的*缺省版本*。

4.6.3 创建区分版本的符号

缺省情况下，链接器不会为非 BPABI 共享对象创建区分版本的符号。

嵌入的符号

可以将特别命名的符号添加到输入对象中，导致链接器创建符号版本。这些符号的格式为：

- `name@version`，适用于非缺省符号版本
- `name@@version`，适用于缺省符号版本。

您必须在函数或数据地址中定义要导出的符号。符号名称分为两个部分：符号名称 *name* 和版本定义 *version*。*name* 会添加到动态符号表中并成为共享对象的接口的一部分。Version 创建名为 *ver* 的版本 （如果还不存在），并将 *name* 与名为的 *ver* 版本关联起来。

有关如何创建版本符号的详细信息，请参阅：

- 《编译器用户指南》中第2-27 页的*添加符号版本*
- 《汇编器指南》中第 2 章 *编写ARM 汇编语言*

示例 4-5 将符号 foo@ver1、 foo@@ver2 和 bar@@ver1 放到对象符号表中：

示例 4-5 创建区分版本的符号、嵌入的符号

```
int old_function(void) __asm__("foo@ver1");
int new_function(void) __asm__("foo@@ver2");
int other_function(void) __asm__("bar@@ver1");
```

链接器读取这些符号，并创建版本定义 ver1 和 ver2。符号 foo 与 ver1 的非缺省版本以及 ver2 的缺省版本相关联。符号 bar 与 ver1 的缺省版本相关联。

不能通过这种方法在版本之间创建关联。

控制文件

可以在命令行选项 --symver_script=*file* 指定的脚本文件中嵌入命令以生成符号版本。通过使用此选项，可自动启用符号版本控制。

脚本文件支持与 GNU *ld* 链接器相同的语法。

通过使用脚本文件，您可以将版本与以前的版本相关联。

除了嵌入符号方法之外，还可以提供控制文件。有关详细信息，请参阅第4-12 页的*控制文件格式*。如果选择这样做，则脚本文件必须与嵌入的符号相匹配并使用 Backus-Naur 格式 (BNF) 表示法：

```
version_definition ::=

  version_name "{" symbol_association* "}" [depend_version] ";"
```

version_name 是一个包含版本名称的字符串。*depend_version* 是一个包含此 *version_name* 所依赖版本的名称的字符串。您必须已在脚本文件中定义了此版本。版本名称并不重要，但有助于选择可读性较高的名称，例如：

```
symbol_association ::=

  "local:" | "global:" | symbol_name ";"
```

其中：

- "local:" 指示此版本定义中的所有后续 symbol_name 均是共享对象的本地名称，并且不区分版本。
- "global:" 指示所有后续 symbol_name 属于此版本定义。

  每个版本定义开头都有一个隐式 "global:"。
- symbol_name 是静态符号表中的全局符号的名称。

示例 4-6 说明了一个控制文件，它对应于嵌入的符号示例 （第4-16 页的示例 4-5），并添加了相关性信息以使 ver2 依赖于 ver1：

示例 4-6 创建区分版本的符号、控制文件

```
ver1
{
    global:
        foo; bar;
    local:
        *;
};

ver2
{
    global:
        foo;
} ver1;
```

错误和警告

如果使用脚本文件，版本定义与其关联的符号必须匹配。如果链接器检测到任何不匹配的情况，则会发出警告。

文件名

可以使用命令行选项 --symver_soname 打开隐式符号版本控制。如果需要对符号进行版本控制以强制进行静态绑定，而并不关心为其指定的版本号，则可以使用此选项。

如果符号没有已定义的版本，链接器将使用所链接的文件的 SONAME。

无法将此选项与嵌入的符号或脚本文件配合使用。

第 5 章
使用分散加载描述文件

本章介绍如何将 ARM® 链接器 armlink 与分散加载描述文件配合使用以创建复杂映像。本章分为以下几节：

- 第5-2 页的*关于分散加载*
- 第5-9 页的*指定区和节地址的示例*
- 第5-31 页的*简单映像的等效分散加载描述*

5.1 关于分散加载

映像由区和输出节组成。映像中的每个区可以包含不同的加载和执行地址。有关详细信息，请参阅第3-2 页的*指定映像结构*。

要构建映像的内存映射，链接器必须具有：

- 描述如何将输入节划分到输出节和区中的分组信息
- 描述区位于内存映射中的地址的位置信息

通过使用分散加载机制，您可以使用文本文件中的描述为链接器指定映像的内存映射。分散加载为您提供了对映像组件分组和位置的全面控制。分散加载可以用于简单映像，但它通常仅用于具有复杂内存映射的映像，即多个区在加载和执行时分散在内存映射中。

5.1.1 为分散加载定义的符号

当链接器使用分散加载描述文件创建映像时，它会创建一些与区相关的符号。第4-3 页的*与区相关的符号*对这些符号进行了介绍。仅当代码引用这些特殊符号时，链接器才会创建它们。

未定义的符号

请注意，在使用分散加载描述文件时，*不会* 定义以下符号：

- `Image$$RW$$Base`
- `Image$$RW$$Limit`
- `Image$$RO$$Base`
- `Image$$RO$$Limit`
- `Image$$ZI$$Base`
- `Image$$ZI$$Limit`

有关详细信息，请参阅第4-3 页的*访问链接器定义的符号*。

如果使用分散加载描述文件，但没有指定任何特殊区名称，也没有重新实现 `__user_initial_stackheap()`，则库会生成错误消息。

有关详细信息，请参阅：

- 《库和浮点支持指南》中第2-67 页的*调整运行时内存模型*
- 《开发指南》中第3-13 页的*放置堆栈和堆*

5.1.2 使用分散加载描述文件指定堆栈和堆

ARM C 库提供了 __user_initial_stackheap() 函数的多个实现，可以根据分散加载描述文件中给出的信息自动选择正确的函数实现。

要选择两个区内存模型，请在分散加载描述文件中定义两个名为 ARM_LIB_HEAP 和 ARM_LIB_STACK 的特殊执行区。两个区均具有 EMPTY 属性。这导致库选择使用以下符号值的非缺省 __user_initial_stackheap() 实现：

- Image$$ARM_LIB_STACK$$Base
- Image$$ARM_LIB_STACK$$ZI$$Limit
- Image$$ARM_LIB_HEAP$$Base
- Image$$ARM_LIB_HEAP$$ZI$$Limit

只能指定一个 ARM_LIB_STACK 或 ARM_LIB_HEAP 区，并且必须分配大小，例如：

```
ARM_LIB_HEAP 0x20100000 EMPTY 0x100000-0x8000  ; Heap starts at 1MB
                                               ; and grows upwards
ARM_LIB_STACK 0x20200000 EMPTY -0x8000         ; Stack space starts at the end
                                               ; of the 2MB of RAM
                                               ; And grows downwards for 32KB
```

注意

如果使用上面的堆栈函数，则还必须在汇编源代码中包含一个 IMPORT __use_two_region_memory，或在 C/C++ 源代码中包含一个 #pragma import(__use_two_region_memory)，因为不会自动选择双区模型。

可以使用 EMPTY 属性定义单个名为 ARM_LIB_STACKHEAP 的执行区，强制 __user_initial_stackheap() 使用合并的堆栈/堆区。这导致 __user_initial_stackheap() 使用符号 Image$$ARM_LIB_STACKHEAP$$Base 和 Image$$ARM_LIB_STACKHEAP$$ZI$$Limit 的值。

注意

如果重新实现 __user_initial_stackheap()，这将覆盖所有库实现。

5.1.3 何时使用分散加载

链接器的命令行选项提供了一些对数据和代码位置的控制，但要对位置进行全面控制，则需要使用比命令行中的输入内容更详细的指令。需要或最好使用分散加载描述的情况包括：

复杂内存映射

如果必须将代码和数据放在多个不同的内存区域中，则需要使用详细指令指定将哪个节放在哪个内存空间中。

不同类型的内存

许多系统都包含多种不同的物理内存设备，如闪存、 ROM、 SDRAM 和快速 SRAM。分散加载描述可以将代码和数据与最适合的内存类型相匹配。例如，可以将中断代码放在快速 SRAM 中以缩短中断响应时间，而将不经常使用的配置信息放在较慢的闪存中。

内存映射的外围设备

分散加载描述可以将数据节准确放在内存映射中的某个地址，以便能够访问内存映射的外围设备。

位于固定位置的函数

可以将函数放在内存中的相同位置，即使已修改并重新编译周围的应用程序。这有助于实现跳转表。

使用符号标识堆和堆栈

链接应用程序时，可以为堆和堆栈位置定义一些符号。

因此，几乎总是需要使用分散加载来实现嵌入式系统，因为这些系统使用 ROM、 RAM 和内存映射的外围设备。

——注意——

如果针对 Cortex™-M3 处理器进行编译，则会包括固定的内存映射，因此可以使用分散加载描述文件来定义堆栈和堆。在示例目录 *install_directory*\RVDS\Examples\..\Cortex-M3\Example3 中提供了这样一个示例。

5.1.4 分散加载命令行选项

用于分散加载的 armlink 命令行选项为：

`--scatter=`*`description_file`*

它指示链接器按照 *description_file* 中的描述构建映像内存映射。在《链接器参考指南》的第 3 章 *分散加载描述文件的形式语法*中介绍了描述文件的格式。

有关分散加载描述文件的其他信息，请参阅：

- 第5-9 页的*指定区和节地址的示例*
- 第5-31 页的*简单映像的等效分散加载描述*

5.1.5 具有简单内存映射的映像

图 5-1 中的分散加载描述将对象文件中的段加载到内存中，它与第5-6 页的图 5-2 中所示的映射相对应。区的最大大小设置是可选的，但是如果包含这些设置，则使链接器能够检查区是否没有溢出其边界。

在此示例中，可通过将 `--ro-base 0x0` 和 `--rw-base 0x10000` 指定为链接器的命令行选项来获得相同的结果。

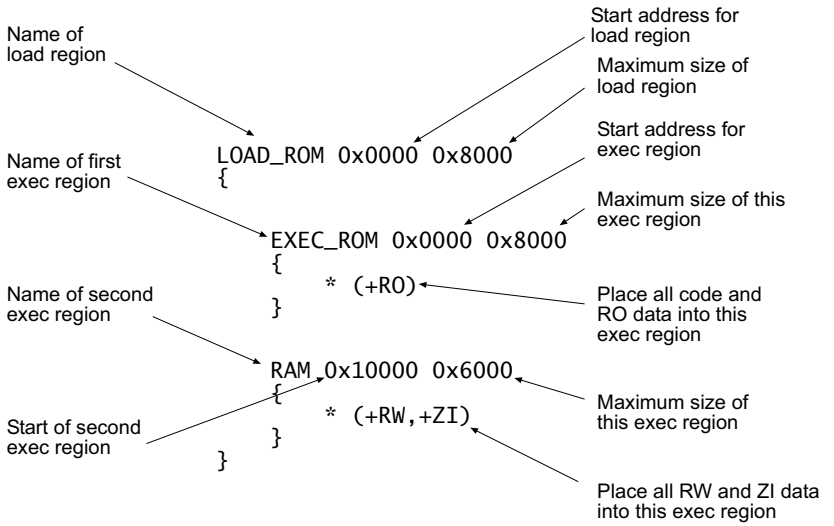

图 5-1 分散加载描述文件中的简单内存映射

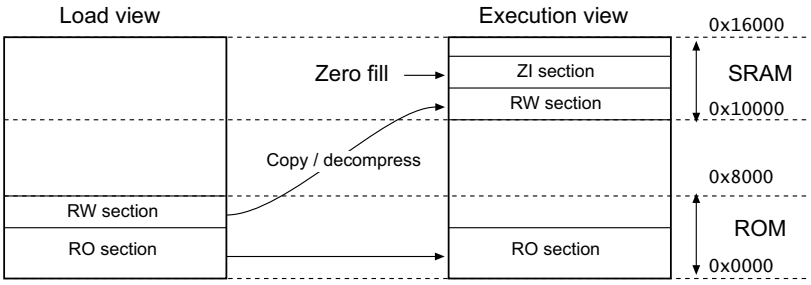

图 5-2 简单分散加载内存映射

5.1.6 具有复杂内存映射的映像

图 5-3 中的分散加载描述将 program1.o 和 program2.o 文件中的段加载到内存中，它与第5-8 页的图 5-4 中所示的映射相对应。

与第5-6 页的图 5-2 中所示的简单内存映射不同，不能只通过使用基本命令行选项为链接器指定此应用程序。

小心

图 5-3 中的分散加载描述仅指定 program1.o 和 program2.o 的代码和数据位置。如果链接其他模块 （如 program3.o）并使用此描述文件，则不会指定 program3.o 的代码和数据位置。

除非对代码和数据位置的要求非常严格，否则建议使用 * 或 .ANY 说明符来放置其余代码和数据。有关详细信息，请参阅第5-11 页的*将区放在固定地址中*。

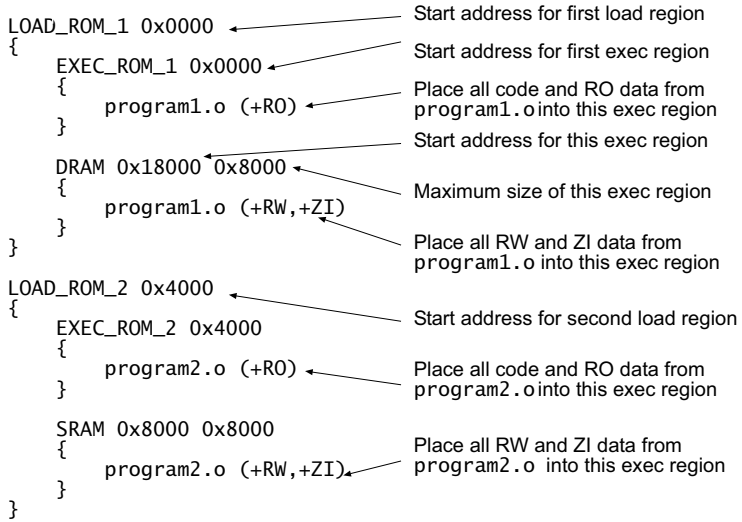

图 5-3 分散加载描述文件中的复杂内存映射

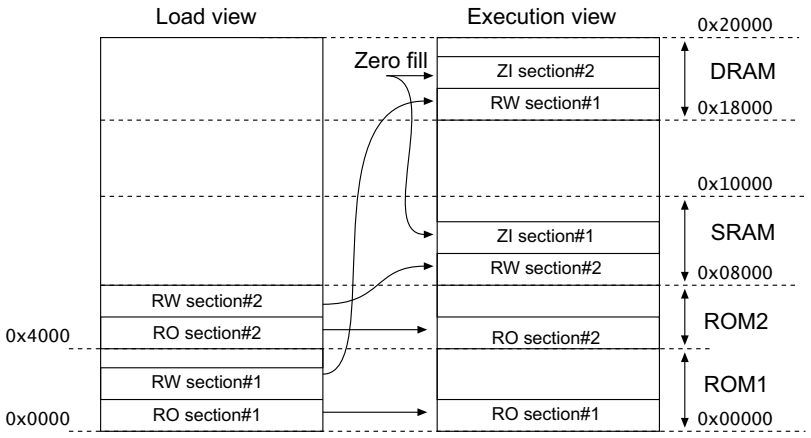

图 5-4 复杂分散加载内存映射

5.2 指定区和节地址的示例

本节介绍输入和执行节、区和预处理指令。有关访问位于固定地址中的数据和函数的示例，请参阅 《开发指南》中的第 3 章 *嵌入式软件开发*。

5.2.1 在分散加载描述中选择中间代码输入节

中间代码用于在 ARM 和 Thumb 代码之间切换，或者执行比一条指令中指定的跳转范围更大的程序跳转。请参阅第3-17 页的*中间代码*。可以使用分散加载描述文件来放置链接器生成的中间代码输入节。分散加载描述文件中的一个执行区最多可以包含 `*(Veneer$$Code)` 节选择器。

如果此操作是安全的，链接器则会将中间代码输入节放到 `*(Veneer$$Code)` 节选择器识别的区中。由于地址范围问题或执行区大小限制，可能无法将中间代码输入节分配到区中。如果不能将中间代码添加到指定的区中，则会将它添加到包含生成中间代码的重定位输入节的执行区中。

注意

在早期版本的 ARM 工具中，分散加载描述文件中的 `*(IWV$$Code)` 实例自动转换为 `*(Veneer$$Code)`。应在新描述中使用 `*(Veneer$$Code)`。

如果执行区中的代码数量超过以下数量，则会忽略 `*(Veneer$$Code)`：Thumb 代码 4Mb、 Thumb-2 代码 16Mb 以及 ARM 代码 32Mb。

5.2.2 创建根执行区

如果指定了映像的初始入口点，或者由于您仅使用一个 `ENTRY` 指令而使链接器创建了一个初始入口点，则必须确保该入口点位于根区中。根区是指加载地址和执行地址相同的区。如果初始入口点不在根区中，则链接会失败，且链接器会生成错误消息。

要在分散加载描述文件中将区指定为根区，您可以执行以下任一操作：

- 显式地将 `ABSOLUTE` 指定为执行区属性 （或将其作为缺省设置），并且第一个执行区及其所在的加载区使用相同的地址。要使执行区地址与加载区地址相同，请执行以下任一操作：
  - 为执行区基址和加载区基址指定相同的数值
  - 为加载区中的第一个执行区指定 `+0` 偏移。

    如果为加载区中的所有后续执行区指定 0 偏移 (`+0`)，则它们均为根区。

请参阅示例 5-1。

- 使用 FIXED 执行区属性以确保特定区的加载地址和执行地址是相同的。请参阅示例 5-2 和第5-11 页的图 5-5。

可以使用 FIXED 属性将任何执行区放在 ROM 中的特定地址。有关详细信息，请参阅第5-11 页的*将区放在固定地址中*。

示例 5-1 指定相同的加载和执行地址

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        * (+RO)        ; all RO sections (must include section with
                       ; initial entry point)
    }
    ...                ; rest of scatter description
}
```

示例 5-2 使用 FIXED 属性

```
LR_1 0x040000              ; load region starts at 0x40000
{                          ; start of execution region descriptions
    ER_RO 0x040000         ; load address = execution address
    {
        * (+RO)            ; RO sections other than those in init.o
    }
    ER_INIT 0x080000 FIXED ; load address and execution address of this
                           ; execution region are fixed at 0x80000
    {
        init.o(+RO)        ; all RO sections from init.o
    }
    ...                    ; rest of scatter description
}
```

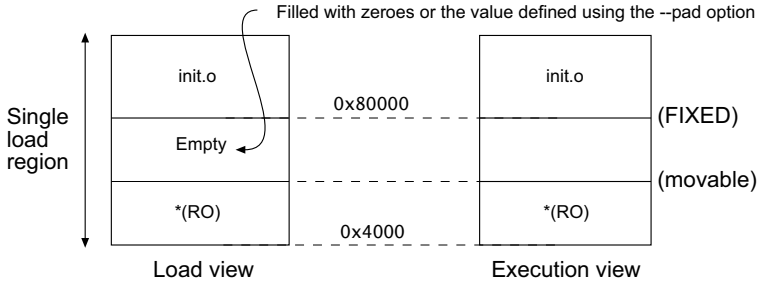

图 5-5 固定执行区的内存映射

将区放在固定地址中

可以在执行区分散加载描述文件中使用 FIXED 属性来创建根区，将在固定地址加载和执行这些区。

FIXED 用于在单个加载区内 （因而通常为单个 ROM 设备）创建多个根区。例如，可以使用此属性将函数或数据块 （如常数表或校验和）放到 ROM 中的固定地址，以便可以通过指针方便地对其进行访问。

例如，如果指定将某些初始化代码放在 ROM 开头，而将校验和放在 ROM 末尾，则可能不会使用某些内存内容。应使用 * 或 .ANY 模块选择器填充初始化块末尾和数据块开头之间的区。

注意

要使代码更易于维护和调试，请在分散加载描述文件中使用最少数量的位置说明，而由链接器提供函数和数据的详细位置信息。

无法指定已部分链接的组件对象。例如，如果部分链接 obj1.o、obj2.o 和 obj3.o 对象以生成 obj_all.o，则会在结果对象中弃用结果组件对象的名称。因此，不能按名称引用其中的某个对象，例如 obj1.o。只能引用合并的对象 obj_all.o。

将函数和数据放在特定地址中

通常，编译器通过单个源文件生成 RO、RW 和 ZI 节。这些区包含源文件中的所有代码和数据。要将单个函数或数据项放在固定地址中，您必须允许链接器单独处理该函数或数据，而与输入文件的其余部分分开。

链接器可以使用两种方法，将某个节放在特定地址中：

- 可以使用只选择一个节的节描述在所需地址创建执行区。

- 对于特别命名的节，链接器可以从节名称中获取放置地址。这些特别命名的节称为 __at 节。有关详细信息，请参阅第5-16 页的*使用 __at 节将节放在特定地址中*。

要将函数或变量放在特定地址中，必须将其放在自己的节中。可以使用几种方法来执行此操作：

- 将函数或数据项放在其自己的源文件中。
- 使用 --split_sections 编译器选项可为源文件中的每个函数分别生成一个 ELF 节。请参阅 《编译器参考指南》中第2-103 页的*--split_sections*。

  对于一些函数，此选项将稍微增大代码的大小，因为它减小了函数之间共享地址、数据和字符串文字的可能性。但是，通过指定 armlink --remove 以允许链接器删除未使用的函数，这有助于减小最终映像的总大小。
- 使用 __attribute__((section("*name*"))) 可创建多个命名节。请参阅 《编译器参考指南》中第4-45 页的*__attribute__((section))*。
- 使用汇编语言中的 AREA 指令。在汇编代码中，最小的可定位单元是 AREA。有关详细信息，请参阅 《汇编器指南》。

显式地使用分散加载放置已命名的节

示例 5-3 中的分散加载描述文件将：

- 初始化代码放在地址 0x0 中，其后是其余 RO 代码和除 data.o 对象中的 RO 数据之外的所有 RO 数据
- 所有全局 RW 变量放在 RAM 的 0x400000 中
- data.o 的 RO-DATA 表放在固定地址 0x1FF00 处。

示例 5-3 节放置

```
LR1 0x0 0x10000
{
    ER1 0x0 0x2000                ; Root Region, containing init code
    {                             ; place init code at exactly 0x0
        init.o (Init, +FIRST)
        * (+RO)                   ; rest of code and read-only data
    }
    RAM  0x400000                 ; RW & ZI data to be placed at 0x400000
    {
        * (+RW +ZI)
    }
    DATABLOCK 0x1FF00 FIXED 0xFF  ; execution region fixed at 0x1FF00
    {                             ; maximum space available for table is 0xFF
        data.o(+RO-DATA)          ; place RO data between 0x1FF00 and 0x1FFFF
    }
}
```

注意

在某些情况下，不适合将 FIXED 和单个加载区配合使用。下面是其他一些指定固定位置的方法：

- 如果加载程序可以处理多个加载区，请将 RO 代码或数据放在其自己的加载区中。
- 如果不需要将函数或数据放在 ROM 中的固定位置，请使用 ABSOLUTE 而不是 FIXED。加载程序随后会将数据从加载区复制到 RAM 中的指定地址。ABSOLUTE 是缺省属性。

- 要将数据结构放在内存映射的 I/O 位置，请使用两个加载区并指定 UNINIT。UNINIT 可确保不会将内存位置初始化为零。有关详细信息，请参阅 《开发指南》中第 3 章 *嵌入式软件开发*。

使用 __attribute__((section("name")))

标准编码方法是，将代码或数据对象放在其自己的源文件中，然后放置对象文件节。但是，也可以使用 __attribute__((section("name"))) 和分散加载描述文件来放置已命名的节。应创建一个模块 （如 adder.c）并显式地命名节，如示例 5-4 中所示。

示例 5-4 命名节

```
int variable __attribute__((section("foo"))) = 10;
```

可以使用分散加载描述文件指定已命名的节的放置位置，请参阅示例 5-5。如果代码和数据节的名称相同，则先放置代码节。

示例 5-5 放置节

```
FLASH 0x24000000 0x4000000
{
    ...                                 ; rest of code

    ADDER 0x08000000
    {
        adder.o (foo)                   ; select section foo from adder.o
    }
}
```

使用 __at 节将节放在特定地址中

可以为节指定一个特殊名称，以编码必须将其放置到的地址。您可以按以下方式指定名称：

.ARM.__at_*address*

其中：

address 是所需的节地址。可以按十六进制或十进制指定此地址。采用 .ARM.__at_*address* 格式的节是由缩写 __at 引用的。

在编译器中，可通过以下方式将变量分配给 __at 节：使用 __attribute__((section("*name*"))) 显式命名节或使用属性 __at 为您设置节的名称。请参阅示例 5-6。

注意

在使用 __attribute__((at(<address>))) 时， __AT 节名称中表示 *address* 的部分会标准化为 8 位十六进制数字。仅当您尝试在分散加载描述文件中按名称匹配节时，节的名称才有意义。

示例 5-6 将变量分配给 __at 节

```
; place variable1 in a section called .ARM.__at_0x00008000
int variable1 __attribute__((at(0x8000))) = 10;

; place variable2 in a section called .ARM.__at_0x8000
int variable2 __attribute__((section(".ARM.__at_0x8000"))) = 10;
```

有关详细信息，请参阅 《编译器参考指南》中第4-42 页的 *__attribute__((at(address)))* 和第4-45 页的 *__attribute__((section))*。

限制

- __at 节地址范围不能重叠，除非将重叠节放在不同的重叠区中
- 不允许在与位置无关的执行区中使用 __at 节
- 不能引用 __at 节的链接器定义的符号 $$Base、$$Limit 和 $$Length
- 不能在 System V 和 BPABI 可执行文件和 BPABI DLL 中使用 __at 节
- __at 节地址必须是其对齐边界的倍数
- __at 节忽略所有 +FIRST 或 +LAST 排序约束。

自动放置

此模式是使用链接器命令行选项 --autoat 启用的。有关详细信息，请参阅 《链接器参考指南》中第2-4 页的*--[no_]autoat*。

使用 --autoat 选项进行链接时，分散加载选择器不会放置 __at 节。相反，链接器将 __at 节放在兼容区中。如果找不到兼容区，链接器将为 __at 节创建加载和执行区。

链接器使用 --autoat 创建的所有执行区均具有 UNINIT 分散加载属性。如果需要对 ZI __at 节进行零初始化，则必须将其放在兼容区中。链接器使用 --autoat 创建的执行区必须具有至少 4 字节对齐的基址。如果有任何区未正确对齐，则链接器会产生错误消息。

兼容区是指：

- __at 地址位于执行区基址和限制范围内，其中限制是指基址 + 最大执行区大小。如果未设置最大大小，则假定限制为无限大的值。
- 加载区至少满足以下条件之一：
  - 它包含一个按标准分散加载规则与 __at 节相匹配的选择器
  - 它至少具有一个类型与 __at 节相同的节 （RO、 RW 或 ZI）
  - 它没有 EMPTY 属性。

    注意

    链接器将类型为 RW 的 __at 节视为与 RO 兼容。

示例 5-7 说明了具有如下类型的节：.ARM.__at_0x0 RO、 .ARM.__at_0x2000 RW、.ARM.__at_0x4000 ZI 和 .ARM.__at_0x8000 ZI。

示例 5-7 __at 节的自动放置

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)      ; .ARM.__at_0x0 lies within the bounds of ER_RO
    }
    ER_RW 0x2000 0x2000
    {
        *(+RW)      ; .ARM.__at_0x2000 lies within the bounds of ER_RW
    }
    ER_ZI 0x4000 0x2000
```

```
    {
        *(+ZI)      ; .ARM.__at_0x4000 lies within the bounds of ER_ZI
    }
}

; the linker creates a load and execution region for the __at section
; .ARM.__at_0x8000 because it lies outside all candidate regions.
```

手动放置

可以使用标准分散加载规则来确定用于放置 __at 节的执行区。

示例 5-8 说明了只读节 .ARM.__at_0x2000 和读写节 .ARM.__at_0x4000 的放置方式。在手动模式下，不会自动创建加载区和执行区。如果不能将 __at 节放在执行区中，则会产生错误。

示例 5-8 __at 节的手动放置

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)               ; .ARM.__at_0x0 is selected by +RO
    }
    ER_RO2 0x2000
    {
        *(.ARM.__at_0x2000)  ; .ARM.__at_0x2000 is selected by .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW +ZI)           ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

将键放在闪存中

某些闪存设备需要将键写入到地址中以激活某些功能。__at 节提供了一种简单方法，将值写入到特定地址中。

假定设备的闪存范围从 0x8000 到 0x10000，并且需要将键放在地址 0x8000 中。若要对 __at 节执行此操作，必须声明一个变量，以便编译器可以生成名为 .ARM.__at_0x8000 的节。请参阅第5-16 页的示例 5-6。

示例 5-9 演示了一个手动放置闪存执行区的分散加载描述文件。

示例 5-9 手动放置闪存执行区

```
ER_FLASH 0x8000 0x2000
{
    *(+RO)                      ; other code, read-only data, and padding if
reqd
    *(.ARM.__at_0x8000)         ; key
}
```

示例 5-10 演示了一个自动放置闪存执行区的分散加载描述文件。可以使用链接器命令行选项 --autoat 来启用自动放置。

示例 5-10 自动放置闪存执行区

```
ER_FLASH 0x8000 0x2000
{
    *(+RO)                ; other code and read-only data, the
                          ; __at section is automatically selected
}
```

将结构映射到外围寄存器上

要将未初始化的变量放在外围寄存器上，您可以使用 ZI __at 节。假定有一个寄存器可在 0x10000000 处使用，则可定义一个名为 .ARM.__at_0x10000000 的 ZI __at 节。例如：

```
int foo __attribute__((section( ".ARM.__at_0x10000000" ), zero_init));
```

示例 5-11 演示了一个手动放置 ZI __at 节的分散加载描述文件。

示例 5-11 手动放置 ZI __at 节

```
ER_PERIPHERAL 0x10000000 UNINIT
{
    *(.ARM.__at_0x10000000)
}
```

使用自动放置时，假定 0x10000000 附近没有其他执行区，链接器将在 0x10000000 处自动创建一个包含 UNINIT 属性的区。

5.2.3 使用重叠区放置节

可以在分散加载描述文件中使用 OVERLAY 属性将多个执行区放置在同一地址处。需要使用重叠区管理器以确保一次只实例化一个执行区。ARM RealView 编译工具不提供重叠区管理器。

示例 5-12 在 RAM 中定义了一个静态节，后跟一系列重叠区。此处，一次只能实例化其中的一个节。

示例 5-12 指定根区

```
EMB_APP 0x8000
{
    .
    .
    STATIC_RAM 0x0                  ; contains most of the RW and ZI code/data
    {
            * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY    ; start address of overlay...
    {
            module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
            module2.o (+RW,+ZI)
    }
    ...                             ; rest of scatter description...
}
```

标记为 OVERLAY 的区不会在启动时由 C 库初始化。重叠区使用的内存的内容由重叠区管理器负责。因此，重叠区管理器必须复制任何代码和数据，并在实例化某个区时初始化任何 ZI。如果该区包含初始化的数据，则还需要使用 NOCOMPRESS 避免进行 RW 数据压缩。

链接器定义的符号可以用于获取复制代码和数据所需的地址，有关详细信息，请参阅第4-3 页的*访问链接器定义的符号*。

OVERLAY 属性可以在地址与另一个区不同的单个区上使用。因此，重叠区可以用于防止 C 库启动代码初始化特定的区。与任何重叠区一样，这些区必须在代码中手动初始化。

重叠区可以有相对基址。具有 +offset 基址的重叠区的行为取决于该区之前的区以及 +offset 的值 （+0 有特殊含义）。

表 5-1 显示了 +offset 在用于 OVERLAY 属性时的效果。在分散加载描述文件中，REGION1 出现在 REGION2 之前的相邻位置上。

表 5-1 在重叠区中使用相对偏移量

REGION1 是否设置了 OVERLAY	+OFFSET	REGION2 基址
否	*<offset>*	REGION1 Limit + *<offset>*
是	+0	REGION1 Base Address
是	*<non-0 offset>*	REGION1 Limit + *<non-0 offset>*

示例 5-13 演示了相对偏移量如何用于重叠区及其对执行区地址的作用。

如果非重叠区域的长度未知，可以使用 0 相对偏移量指定重叠区的开始地址，以便将其放在静态节末尾的相邻位置。

示例 5-13 重叠区中的相对偏移量示例

```
EMB_APP 0x8000
{
    CODE 0x8000
    {
        *(+RO)
    }

    # REGION1 Base = CODE limit
    REGION1 +0 OVERLAY
    {
        module1.o(*)
    }

    # REGION2 Base = REGION1 Base
    REGION2 +0 OVERLAY
    {
        module2.o(*)
    }

    # REGION3 Base = REGION2 Base = REGION1 Base
    REGION3 +0 OVERLAY
    {
        module3.o(*)
    }
```

```
    # REGION4 Base = REGION3 Limit + 4
    Region4 +4 OVERLAY
    {
        module4.o(*)
    }
}
```

5.2.4 为根区分配节

有许多 ARM 库节必须放置在根区中，例如 __main.o、__scatter*.o、__dc*.o 和 *Region$$Table。此列表在不同版本中可能有差异。链接器可以使用 InRoot$$Sections 自动放置所有这些节，而不会影响将来的使用。

可以使用分散加载描述文件按照与已命名节相同的方法指定根节。示例 5-14 使用节选择器 InRoot$$Sections 放置必须位于根区中的所有节。

示例 5-14 指定根区

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000      ; root region at 0x0
  {
    vectors.o (Vect, +FIRST)  ; Vector table
    * (InRoot$$Sections)      ; All library sections that must be in a
                              ; root region, for example, __main.o,
                              ; __scatter*.o, __dc*.o, and * Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)         ; all other sections
  }
}
```

5.2.5 保留空白区

可以在执行区分散加载描述中使用 EMPTY 属性，为堆栈保留一个空白内存块。

该内存块并不构成加载区的一部分，而是在执行时分配使用的。由于它是作为虚 ZI 区创建的，因此链接器使用以下符号对其进行访问：

- Image$$region_name$$ZI$$Base
- Image$$region_name$$ZI$$Limit
- Image$$region_name$$ZI$$Length.

如果指定的长度为负值，则将该地址作为区结束地址。它必须是绝对地址，而不是相对地址。例如，第5-26 页的示例 5-15 中说明的执行区定义 STACK 0x800000 EMPTY -0x10000 定义了一个名为 STACK 的区，它的开始地址是 0x7F0000，结束地址是 0x800000。

注意

在运行时，不会将为 EMPTY 执行区创建的虚 ZI 区初始化为零。

如果地址采用相对格式 (+*n*)，并且长度为负值，链接器将生成错误。

示例 5-15 为堆栈保留区

```
LR_1 0x80000                            ; load region starts at 0x80000
{
    STACK 0x800000 EMPTY -0x10000        ; region ends at 0x800000 because of the
                                        ; negative length. The start of the
region
                                        ; is calculated using the length.
    {
                                        ; Empty region used to place stack
    }
    HEAP +0 EMPTY 0x10000               ; region starts at the end of previous
                                        ; region. End of region calculated using
                                        ; positive length
    {
                                        ; Empty region used to place heap
    }
    ...                                 ; rest of scatter description...
}
```

第5-27 页的图 5-6 是此示例的图形表示形式。

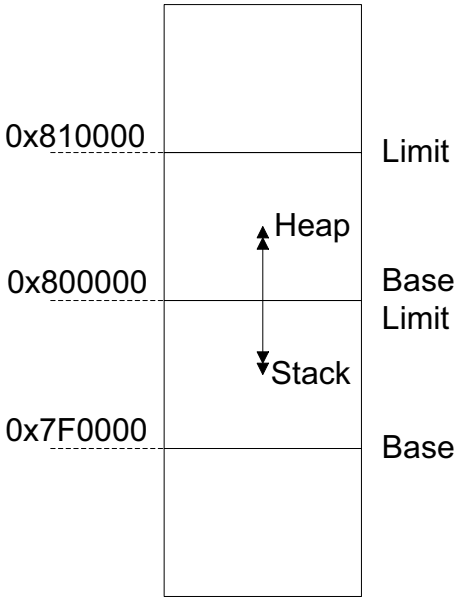

图 5-6 为堆栈保留区

在此示例中，链接器生成以下符号：

```
Image$$STACK$$ZI$$Base      = 0x7f0000
Image$$STACK$$ZI$$Limit     = 0x800000
Image$$STACK$$ZI$$Length    = 0x10000
Image$$HEAP$$ZI$$Base       = 0x800000
Image$$HEAP$$ZI$$Limit      = 0x810000
Image$$HEAP$$ZI$$Length     = 0x10000
```

注意

EMPTY 属性仅适用于执行区。如果在加载区定义中使用 EMPTY 属性，链接器将生成警告并忽略该属性。

链接器检查用于 EMPTY 区的地址空间是否与任何其他执行区重叠。

5.2.6 放置 ARM 库

可以将 ARM 标准 C 和 C++ 库中的代码放在分散加载描述文件中。应使用 *armlib 或 *armlib*，以使链接器能够解析分散加载文件中的库命名。例如：

```
ER 0x2000
{
    *armlib/c_* (+RO)                   ; all ARM-supplied C libraries
    ...                                 ; rest of scatter description...
}
```

示例 5-16 说明了如何放置库代码。

注意

示例 5-16 在路径名中使用正斜杠，以确保在 Windows 和 Unix 平台上能够识别它们。

示例 5-16 放置 ARM 库代码

```
ROM1 0
{
    * (InRoot$$Sections)
    * (+RO)
    ROM2 0x1000
    {
        *armlib/c_* (+RO)         ; all ARM-supplied C library functions
    }
}
ROM3 0x2000
{
    *armlib/h_* (+RO)                   ; just the ARM-supplied __ARM_*
                                        ; redistributable library functions
}
RAM1 0x3000
{
    *armlib* (+RO)                      ; all other ARM-supplied library code
                                        ; e.g. floating-point libraries
}
RAM2 0x4000
{
    * (+RW, +ZI)
}
```

5.2.7 在页边界上创建区

ALIGN 指令生成一个 ELF 文件，该文件可以直接加载到目标中，其中每个执行区都始于一个页边界。

示例 5-17 假定页面大小为 65536，并生成一个 ELF 文件，其中每个区都始于一个新页面。

示例 5-17 在页边界上创建区

```
LR1 +4 ALIGN 65536              ; load region at 65536
{
    ER1 +0 ALIGN 65536          ; first region at first page boundary
    {
        *(+RO)                  ; all RO sections are placed consecutively here
    }
    ER2 +0 ALIGN 65536          ; second region at next available page boundary
    {
        *(+RW)                  ; all RW sections are placed consecutively here
    }
    ER3 +0 ALIGN 65536          ; third region at next available page boundary
   {
        *(+ZI)                  ; all ZI sections are placed consecutively here
    }
    ...                         ; rest of scatter description...
}
```

5.2.8 使用预处理指令

可以使用分散加载描述文件中的第一行指定链接器为处理该文件而调用的预处理器。此命令的格式如下：

```
#! <preprocessor> [pre_processor_flags]
```

通常情况下，此命令是 `#! armcc -E`。

链接器可以使用一组有限的运算符执行简单的表达式求值，即 +、-、 *、 /、AND、 OR 和括号。OR 和 AND 实现遵循 C 运算符优先级规则。

您可以将预处理指令添加到分散加载描述文件的顶部。例如：

```
#define ADDRESS 0x20000000
#include "include_file_1.h"
```

链接器解析预处理的分散加载描述文件，其中将这些指令视为注释并忽略。

举一个简单的例子：

```
#define AN_ADDRESS (BASE_ADDRESS+(ALIAS_NUMBER*ALIAS_SIZE))
```

使用以下指令：

```
#define BASE_ADDRESS 0x8000
#define ALIAS_NUMBER 0x2
#define ALIAS_SIZE 0x400
```

如果分散加载描述文件包含：

```
LOAD_FLASH AN_ADDRESS    ; start address
```

进行预处理后，将对其进行计算，结果为：

```
LOAD_FLASH ( 0x8000 + ( 0x2 * 0x400 ))  ; start address
```

在进行计算后，链接器解析分散加载文件以生成加载区：

```
LOAD_FLASH 0x8808 ; start address
```

有关详细信息，请参阅 《链接器参考指南》中第2-42 页的*--predefine="string"*。

5.3 简单映像的等效分散加载描述

第3-22 页的*使用命令行选项创建简单映像*中介绍了如何使用以下命令行选项创建简单映像类型：--reloc、--ro-base、--rw-base、--ropi、--rwpi 和 --split。通过使用 --scatter 命令行选项以及包含相应分散加载描述之一的文件，可以创建相同的映像类型。

5.3.1 类型 1，一个加载区和几个连续执行区

此类映像由加载视图中的单个加载区以及执行视图中的三个执行区组成。执行区是在内存映射中连续放置的。

--ro-base *address* 指定包含 RO 输出节的区的加载和执行地址。示例 5-18 演示了与使用 --ro-base 0x040000 等效的分散加载描述。

示例 5-18 单个加载区和几个连续执行区

```
LR_1 0x040000     ; Define the load region name as LR_1, the region starts at 0x040000.
{
    ER_RO +0      ; First execution region is called ER_RO, region starts at end of previous region.
                  ; However, since there is no previous region, the address is 0x040000.
    {
        * (+RO)   ; All RO sections go into this region, they are placed consecutively.
    }
    ER_RW +0      ; Second execution region is called ER_RW, the region starts at the end of the
                  ; previous region. The address is 0x040000 + size of ER_RO region.
    {
        * (+RW)   ; All RW sections go into this region, they are placed consecutively.
    }
    ER_ZI +0      ; Last execution region is called ER_ZI, the region starts at the end of the
                  ; previous region at 0x040000 + the size of the ER_RO regions + the size of
                  ; the ER_RW regions.
    {
        * (+ZI)   ; All ZI sections are placed consecutively here.
    }
}
```

示例 5-18 中所示的描述创建一个映像，其中包含一个名为 LR_1 的加载区，其加载地址为 0x040000。

该映像包含三个名为 ER_RO、ER_RW 和 ER_ZI 的执行区，它们分别包含 RO、 RW 和 ZI 输出节。RO 和 RW 是根区。ZI 是在运行时动态创建的。ER_RO 的执行地址是 0x040000。通过在执行区描述中使用 *+offset* 格式的基址指示符，可以在内存映射中连续放置所有三个执行区。这样，即可紧靠前一个执行区后面放置下一个执行区。

--reloc 选项用于生成可重定位的映像。单独使用时，--reloc 生成的映像类似于简单类型 1，但单个加载区具有 RELOC 属性。

修改后的 ropi 示例版本

在此版本中，执行区连续放置在内存映射中。但是，--ropi 将包含 RO 输出节的加载和执行区标记为与位置无关。

示例 5-19 演示了与使用 --ro-base 0x010000 --ropi 等效的分散加载描述。

示例 5-19 与位置无关的代码

```
LR_1 0x010000 PI        ; The first load region is at 0x010000.
{
    ER_RO +0            ; The PI attribute is inherited from parent.
                        ; The default execution address is 0x010000, but the code can be moved.
    {
        * (+RO)         ; All the RO sections go here.
    }
    ER_RW +0 ABSOLUTE   ; PI attribute is overridden by ABSOLUTE.
    {
        * (+RW)         ; The RW sections are placed next. They cannot be moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)         ; All the ZI sections are placed consecutively here.
    }
}
```

如示例 5-19 中所示， RO 执行区 ER_RO 从加载区 LR_1 继承 PI 属性。下一个执行区 ER_RW 被标记为 ABSOLUTE，并使用 *+offset* 格式的基址指示符。这可防止 ER_RW 从 ER_RO 继承 PI 属性。另外，由于 ER_ZI 区的偏移为 +0，因此它从 ER_RW 区继承 ABSOLUTE 属性。

5.3.2 类型 2，一个加载区和几个不连续的执行区

此类映像由加载视图中的单个加载区以及执行视图中的三个执行区组成。它与类型 1 映像相似，但 RW 执行区与 RO 执行区不相邻。

`--ro-base=`*address1* 指定包含 RO 输出节的区的加载和执行地址。`--rw-base=`*address2* 指定 RW 执行区的执行地址。

示例 5-20 演示了与使用 `--ro-base=0x010000 --rw-base=0x040000` 等效的分散加载描述。

示例 5-20 单个加载区和多个执行区

```
LR_1 0x010000        ; Defines the load region name as LR_1
{
    ER_RO +0         ; The first execution region is called ER_RO and starts at end of previous region.
                     ; Since there is no previous region, the address is 0x010000.
    {
        * (+RO)      ; All RO sections are placed consecutively into this region.
    }
    ER_RW 0x040000   ; Second execution region is called ER_RW and starts at 0x040000.
    {
        * (+RW)      ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0         ; The last execution region is called ER_ZI.
                     ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)      ; All ZI sections are placed consecutively here.
    }
}
```

此描述创建一个映像，其中包含一个名为 `LR_1` 的加载区，其加载地址是 `0x010000`。

该映像包含三个名为 `ER_RO`、`ER_RW` 和 `ER_ZI` 的执行区，它们分别包含 RO、 RW 和 ZI 输出节。RO 区是根区。`ER_RO` 的执行地址是 `0x010000`。

`ER_RW` 执行区与 `ER_RO` 不相邻。其执行地址是 `0x040000`。

`ER_ZI` 执行区紧靠上一个执行区 `ER_RW` 后面放置。

rwpi 示例变化版本

这与使用 --rw-base 的类型 2 映像相似， RW 执行区与 RO 执行区是分开的。但是，--rwpi 将包含 RW 输出节的执行区标记为与位置无关。

示例 5-21 演示了与使用 --ro-base=0x010000 --rw-base=0x018000 --rwpi 等效的分散加载描述。

示例 5-21 与位置无关的数据

```
LR_1 0x010000           ; The first load region is at 0x010000.
{
    ER_RO +0            ; Default ABSOLUTE attribute is inherited from parent. The execution address
                        ; is 0x010000. The code and ro data cannot be moved.
    {
        * (+RO)         ; All the RO sections go here.
    }
    ER_RW 0x018000 PI   ; PI attribute overrides ABSOLUTE
    {
        * (+RW)         ; The RW sections are placed at 0x018000 and they can be moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)         ; All the ZI sections are placed consecutively here.
    }
}
```

RO 执行区 ER_RO 从加载区 LR_1 继承 ABSOLUTE 属性。下一个执行区 ER_RW 被标记为 PI。另外，由于 ER_ZI 区的偏移为 +0，因此它从 ER_RW 区继承 PI 属性。

还可以编写类似的分散加载描述，以对应于 --ropi 和 --rwpi 与类型 2 和类型 3 映像的其他组合用法。

5.3.3 类型 3，两个加载区和几个不连续的执行区

类型 3 映像由加载视图中的两个加载区以及执行视图中的三个执行区组成。它们与类型 2 映像相似，但类型 2 映像中的单个加载区现在拆分为两个加载区。

可以使用以下链接器选项重定位并拆分加载区：

--reloc　　--reloc --split 组合生成的映像类似于简单类型 3，但两个加载区现在具有 RELOC 属性。

--ro-base=*address1*

指定包含 RO 输出节的区的加载和执行地址。

--rw-base=*address2*

指定包含 RW 输出节的区的加载和执行地址。

--split 将缺省的单个加载区 （包含 RO 和 RW 输出节）拆分为两个加载区。一个加载区包含 RO 输出节；另一个加载区包含 RW 输出节。

示例 5-22 演示了与使用 --ro-base=0x010000 --rw-base=0x040000 --split 等效的分散加载描述。

在本示例中：

- 此描述创建一个映像，其中包含两个名为 LR_1 和 LR_2 的加载区，它们的加载地址是 0x010000 和 0x040000。
- 该映像包含三个名为 ER_RO、ER_RW 和 ER_ZI 的执行区，它们分别包含 RO、RW 和 ZI 输出节。ER_RO 的执行地址是 0x010000。
- ER_RW 执行区与 ER_RO 不相邻。其执行地址是 0x040000。
- ER_ZI 执行区紧靠上一个执行区 ER_RW 后面放置。

示例 5-22 多个加载区

```
LR_1 0x010000     ; The first load region is at 0x010000.
{
    ER_RO +0      ; The address is 0x010000.
    {
        * (+RO)
    }
}
LR_2 0x040000     ; The second load region is at 0x040000.
{
    ER_RW +0      ; The address is 0x040000.
    {
        * (+RW)   ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0      ; The address is 0x040000 + size of ER_RW region.
    {
```

```
        * (+ZI)   ; All ZI sections are placed consecutively into this region.
    }
}
```

可重定位加载区示例变化版本

此类型 3 映像也由加载视图中的两个加载区以及执行视图中的三个执行区组成。但是，用于指定两个加载区的 --reloc 现在具有 RELOC 属性。

示例 5-23 演示了与使用 --ro-base 0x010000 --rw-base 0x040000 --reloc --split 等效的分散加载描述。

示例 5-23 可重定位的加载区

```
LR_1 0x010000 RELOC
{
     ER_RO + 0
     {
         * (+RO)
     }
}

LR2 0x040000 RELOC
{
     ER_RW + 0
     {
         * (+RW)
     }

     ER_ZI +0
     {
         * (+ZI)
     }
}
```