



深度阐释Linux操作系统原理的里程碑之作，由拥有超过10年研发经验的资深
Linux专家撰写

以从零开始构建一个完整的Linux操作系统的过程为依托，宏观上全面厘清了构成Linux操作系统的各个组件以及它们之间的关系，微观上深入探讨了核心组件的基本原理以及相互间的协作关系，指引读者在富有趣味的实践中参透操作系统的本质

华章  精品

深度探索 Linux操作系统 系统构建和原理解析

Inside the Linux Operating System

王柏生 著



 机械工业出版社
China Machine Press

深度探索 Linux 操作系统

系统构建和原理解析

王柏生 著



图书在版编目 (CIP) 数据

深度探索 Linux 操作系统：系统构建和原理解析 / 王柏生著. —北京 : 机械工业出版社, 2013.10
(原创精品系列)

ISBN 978-7-111-43901-1

I . 深… II . 王… III . Linux 操作系统 IV . TP316.89

中国版本图书馆 CIP 数据核字 (2013) 第 208809 号

版权所有•侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书是探索 Linux 操作系统原理的里程碑之作，在众多的同类书中独树一帜。它颠覆和摒弃了传统的从阅读 Linux 内核源代码着手学习 Linux 操作系统原理的方式，而是基于实践，以从零开始构建一个完整的 Linux 操作系统的过程为依托，指引读者在实践中去探索操作系统的本质。这种方式的妙处在于，让读者先从宏观上全面认清一个完整的操作系统中都包含哪些组件，各个组件的作用，以及各个组件间的关系，从微观上深入理解系统各个组件的原理，帮助读者达到事半功倍的学习效果，这是作者潜心研究 Linux 操作系统 10 几年的心得和经验，能避免后来者在学习中再走弯路。此外，本书还对编译链接技术（尤其是动态加载和链接技术）和图形系统进行了原理性的探讨，这部分内容非常珍贵。

全书一共 8 章：第 1 章介绍了如何准备工作环境。在第 2 章中构建了编译工具链，这是后面构建操作系统各个组件的基础。在这一章中，不仅详细讲解了工具链的构建过程，而且还通过对编译链接过程的探讨，深入讨论了工具链的组成及各个组件的作用，理解工具链的工作原理对理解操作系统至关重要。第 3~4 章，从零开始构建了一个具备用户字符界面的最小操作系统，详细讲解了构建的过程以及涉及的技术细节。第 5 章从理论的角度探讨了这一过程，从内核的加载、解压一直讨论到用户进程的加载，包括用户空间的动态链接器为加载程序所作的努力。第 6~7 章首先构建了操作系统的基础图形系统，然后在此基础上构建了桌面环境。第 8 章深入探讨了计算机图形的基础原理，包含 2D 和 3D 程序的渲染、软件渲染、硬件渲染等内容，同时也从操作系统的角度审视了 Pipeline。

华早图书

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑：姜 影

印刷

2013 年 10 月第 1 版第 1 次印刷

186mm×240mm • 27.25 印张

标准书号：ISBN 978-7-111-43901-1

ISBN 978-7-89405-088-5 (光盘)

定价：89.00 元 (附光盘)

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

谨以此书献给恩师李明树先生。



前　　言

为什么要写这本书

真正认真开始学习计算机是在 2000 年，当时书店里到处充斥着一系列如“21 天精通 xxx”、“7 天掌握 xxx”之类的图书，更有甚者宣称“24 小时学会 xxx”。既是高科技，又这么容易学，谁会拒绝呢？于是我走上了这一行。最初，确实如这些书所说，只要按照书中描述，将类似于 Visual Studio 等 IDE 安装到机器上，然后像搭积木一样，拖拽几个控件，再添加几行代码，一个程序就完成了。

短暂的兴奋后，好奇心驱使我想更深层次地探索这一切是如何发生的。于是我开始关注更多的书籍、更多的文章、更多的编程参考，国内的、国外的。但是，结果让我很沮丧，如果依然是用积木来举例子，我发现它们的区别就像一盒 10 块的积木和一盒 100 块的积木，只有量的变化，没有质的区别。有人说 Win32 编程更底层，于是我抛开 MFC，研究 Win32 编程。但是，结局一样让我失望。其实它们也没有本质区别，只不过如果把 MFC 比作大块积木，Win32 是小块积木而已。其间我又遍寻那些 Windows 内幕的书进行研读，也是铩羽而归，似乎前方已无路可走……

2003 年 4 月毕业后，我到了中科院软件所工作，开始从事与 Linux 相关的开发。经历了从 Windows 到 Linux 转型的阵痛后，我开始喜欢上了 Linux，因为它是开源的，我似乎看到了曙光。于是我开始疯狂地购买 Linux 方面各种各样的书籍，阅读各种权威资料，基本上网络上各种权威专家推荐的书籍在我的书桌上全部可以找到。其中，绝大部分是关于内核源码分析的书，于是我一头扎进讲解内核源码分析的书中。但是我很快淹没在庞大的内核代码中，几次都到了难以坚持的程度，但是我强迫自己坚持，强制自己接受作者的灌输。但是，最终的结果是：看的时候似乎明白，但是看完后感觉又什么也没有看。现在回头看，当初很有点像“盲人摸象”这个典故所描述的，在我还没有看清整个“大象”的时候，我就直接去研究“大象”的某些部分的构造了。

彷徨中，我又看到了另外一条路，低版本的内核。我就像一个在沙漠中饥渴难忍的人突然看到了绿洲，我甚至将低版本的内核打印出纸版，然后就像拿着伟人语录一样，只要觅得空隙，就虔诚地潜心研读。但是这条新路除了代码量小了点，与之前的相比并没有太多本质的区别，而且还有一个致命的缺点——早期版本的内核不能和工作中使用的 Linux 很好地结合。

2005 年，我从软件所被派到了中科红旗。最初从事桌面操作系统的开发，使用的是基于 Qt 的 KDE，因为比较成熟，所以当时做得更多的是一些维护工作。但是在我的探索过程中依然重复着上面的故事，没有任何的起色。转折大概出现在 2007 年，Intel 因为一个低功耗平台项目开始和中科红旗合作，他们要在低功耗平台上开发一套 Linux 操作系统，我接手了这项工作。因为这个平台的处理器性能相对要低，所以对于操作系统的要求比较高。同时因为用于消费类电子产品，用户体验要求也与普通的 PC 环境完全不同。所以，基于已有的桌面系统几乎是不可能了。于是，我们开始从头开发和定制。

这个从零开始的过程，让我彻底认识了整个 Linux 操作系统，而不仅仅是 Linux 的内核。曾经对内核中很多做法和模块不明了，通过构建整个操作系统，我豁然开朗。比如，内核中的 DRM 模块，其全称是 Direct Rendering Manager，从字面上看是直接渲染管理，这到底是什么意思？如果你仅仅从内核的角度来理解，相信我，你永远也不能正确理解它。恰恰是在构建系统时，亲手组装和调试图形环境，包括 X、OpenGL、2D/3D 图形驱动，让我明白了 DRM 的用途。这样的例子举不胜举。

经过这个过程中，我深刻认识到，学习操作系统，有三件最重要的事：第一是实践，第二依然是实践，第三还是实践。老祖宗说“纸上得来终觉浅”，唯物主义者说“实践是检验真理的唯一标准”，两句话中都蕴含着同一个道理——追求真理离不开实践。只是阅读、分析源码还远远不够，我们要动手实践，从实践中学习，实践反过来再促进思考。而且，实践也使学习不再是一个枯燥乏味的负担，而是一个乐趣。

通过这个过程，我也体会到，即使只为了学习内核，也不能将目光全部放在内核上。从整个操作系统的角度，从各个组件间关系的角度理解内核，效果反而更好。当对整个系统有了深入的理解后，再去理解组成操作系统的各个组件，会事半功倍。一旦从总体上理解了系统，你就会“艺高人胆大”，就可以尽情地“折腾”Linux 系统了，因为每一个组件尽在你的掌握之中。而恰恰在这不断的“折腾”中，理论又得到不断的提高，从此进入一个良性循环。

很早我就想把这种方法整理成书，和更多的读者分享，希望帮助所有有志于操作系统、又尚在门外徘徊的年轻人少走些弯路。但是因为忙于生计，只能在有限的业余时间写作，所以直到 2013 年中期，才基本把整个书稿写完。

对于计算机而言，操作系统的重要性不言而喻，但它也是我们心中的痛，我将为此求索一生。如果有生之年没能成功，请将我埋在后来者脚下。

读者对象

对于如同笔者一样怀揣操作系统梦的爱好者，希望本书能帮他们顺利地迈进操作系统这扇门；对于正在或者准备学习操作系统理论的大学生，本书将帮助他们感性地触摸那些“高居庙堂之上”的抽象理论；对于高级读者，本书中的很多内容对他们也很有用处，比如动态

链接部分的讨论、Linux 图形原理部分的讨论等。

除了以上的读者外，本书适合以下相关从业人员阅读：

- ❑ 系统程序员。要想成为一个合格的系统程序员，操作系统和编译链接技术是必不可少的技能，本书对此有较深入的讨论。
- ❑ 嵌入式 Linux 工程师。作为一名嵌入式 Linux 工程师，应该知道如何使用交叉编译工具链、配置编译内核、裁剪系统、搭建图形系统，甚至定制桌面环境，这些相关知识读者在本书中都可以找到。
- ❑ Linux 发行版工程师。作为制作发行版的工程师，更需要彻底熟悉操作系统的每个组件以及组件间的关系，本书可以满足他们这方面的需求。
- ❑ Linux 应用开发工程师。对于应用开发程序员，也推荐阅读本书，因为越深入地理解操作系统和编译链接原理，就越能写出高效而简洁的程序。

如何阅读本书

本书围绕着构建一个完整的 Linux 操作系统这一主线展开，除了第 1 章外，其余各章环环相扣，所以请读者严格按照章节顺序阅读。

工欲善其事，必先利其器。尤其是对于这样一本实践丰富的书来说，工作环境是后续内容的基础。因此，第 1 章介绍了如何准备工作环境。但是类似安装 Linux 发行版这样的内容，相关参考随处可见，因此书中并没有浪费篇幅去一一介绍，而是仅仅指出其中需要特别注意之处。

工具链是后面进行构建的基础，因此，接下来在第 2 章中构建了工具链。工具链是整个操作系统中非常重要的一部分，理解工具链的工作原理，对理解操作系统至关重要，所以第 2 章中并没有仅仅停留在构建的层次，还通过探讨编译链接过程，讨论了工具链的组成以及各个组件的作用。

在第 3 章和第 4 章，我们从零开始，构建了一个具备用户字符界面的最小操作系统。同时在第 5 章，我们从更深层次的角度探讨了这一切是如何发生的。我们从内核的加载、解压一直讨论到用户进程的加载，包括用户空间的动态链接器为加载程序所做的努力。

在第 6 章和第 7 章，我们首先构建了系统的基础图形系统，然后在其上构建了桌面环境。在第 8 章，我们深入探讨了计算机图形的基础原理，讨论了 2D 和 3D 程序的渲染、软件渲染、硬件渲染，我们也从操作系统的角度审视了 Pipeline。

笔者强烈建议读者在真实的计算机上安装一个 Linux 操作系统，让它成为你日常的工作机。然后将书中的，尤其是与实践相关的所有命令实际运行一遍。之后再尝试脱离本书，自己争取从头再构建一遍，相信你一定会在这个过程中受益匪浅的。

勘误和支持

由于作者水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者提出宝贵意见，批评指正。来信请发送至邮箱 baisheng_wang@163.com，笔者会尽自己最大的努力给出回复。

致谢

首先感谢恩师李明树先生，是他将我带进了操作系统这扇大门。

感谢机械工业出版社华章公司的策划杨福川，在他身上我看到了专业精神，这也是我在与几个出版团队沟通后，毫不犹豫地决定请他们出版的原因。

感谢机械工业出版社华章公司的姜影编辑，她清晰的思路让我深深折服。每每在遇到困惑不知如何表达时，她都能通过简单的几句话点醒梦中人。

感谢我的父母，感谢他们的养育之恩。感谢我的哥哥，为了让我受到更好的教育，在他刚刚毕业不久，就顶着生活的压力，将我从农村接到了城里接受教育，为我的学业奔波操劳。感谢我的嫂子在生活上给予我的无微不至的照顾。把最后一份感谢留给我的妻子，是她在我工作这些年，承担了照顾父母、操持家务的重任，是她的无私付出让我能全身心地投入到工作和学习中。

王柏生

北京

目 录

前 言

第 1 章 准备基本环境 1

1.1 安装 VirtualBox	1
1.2 创建虚拟计算机.....	2
1.3 安装 Linux 系统.....	2
1.4 使用 root 用户	5
1.5 启用自动登录	5
1.6 挂载实验分区	6
1.7 安装 ssh 服务器	6
1.8 更改网络模式	7
1.9 安装增强模式	8
1.10 使用 Xephyr.....	8

第 2 章 工具链 10

2.1 编译过程	10
2.1.1 预编译.....	12
2.1.2 编译	14
2.1.3 汇编	17
2.1.4 链接	31
2.2 构建工具链.....	39
2.2.1 GNU 工具链组成	40
2.2.2 构建工具链的过程	40
2.2.3 准备工作	43
2.2.4 构建二进制工具.....	45
2.2.5 编译 freestanding 的交叉 编译器.....	46
2.2.6 安装内核头文件.....	49
2.2.7 编译目标系统的 C 库	50

2.2.8 构建完整的交叉编译器	52
2.2.9 定义工具链相关的环境 变量	54
2.2.10 封装“交叉” pkg-config	54
2.2.11 关于使用 libtool 链接库 的讨论	56
2.2.12 启动代码	57

第 3 章 构建内核 62

3.1 内核映像的组成	62
3.1.1 一级推进系统—— setup.bin	63
3.1.2 二级推进系统——内核 非压缩部分	65
3.1.3 有效载荷——vmlinux	65
3.1.4 映像的格式	66
3.2 内核映像的构建过程	68
3.2.1 kbuild 简介	68
3.2.2 构建过程概述	71
3.2.3 vmlinux 的构建过程	71
3.2.4 vmlinux.bin 的构建过程	75
3.2.5 setup.bin 的构建过程	80
3.2.6 bzImage 的组合过程	81
3.2.7 内核映像构建过程总结	82
3.3 配置内核	86
3.3.1 交叉编译内核设置	86
3.3.2 基本内核配置	87
3.3.3 配置处理器	88
3.3.4 配置内核支持模块	90

3.3.5 配置硬盘控制器驱动	91	文件	144
3.3.6 配置文件系统	96	4.6.8 启动 udevd 和模拟热插拔	146
3.3.7 配置内核支持 ELF 文件格式	97		
3.4 构建基本根文件系统	99	4.7 挂载并切换到根文件系统	147
3.4.1 根文件系统的基本目录结构	99	4.7.1 挂载根文件系统	147
3.4.2 安装 C 库	100	4.7.2 切换到根文件系统	149
3.4.3 安装 shell	101		
3.4.4 安装根文件系统到目标系统	102		
第 4 章 构建 initramfs	104	第 5 章 从内核空间到用户空间	154
4.1 为什么需要 initramfs	104	5.1 Linux 操作系统加载	154
4.2 initramfs 原理探讨	105	5.1.1 GRUB 映像构成	155
4.2.1 挂载 rootfs	106	5.1.2 安装 GRUB	160
4.2.2 解压 initramfs 到 rootfs	110	5.1.3 GRUB 启动过程	165
4.2.3 挂载并切换到真正的根目录	116	5.1.4 加载内核和 initramfs	170
4.3 配置内核支持 initramfs	117	5.2 解压内核	181
4.4 构建基本的 initramfs	118	5.2.1 移动内核映像	182
4.5 将硬盘驱动编译为模块	121	5.2.2 解压	186
4.5.1 配置 devtmpfs	121	5.2.3 重定位	187
4.5.2 将硬盘控制器驱动配置为模块	126	5.3 内核初始化	190
4.6 自动加载硬盘控制器驱动	130	5.3.1 初始化虚拟内存	190
4.6.1 内核向用户空间发送事件	131	5.3.2 初始化进程 0	201
4.6.2 udev 加载驱动和建立设备节点	136	5.3.3 创建进程 1	206
4.6.3 处理冷插拔设备	139	5.4 进程加载	209
4.6.4 编译安装 udev	141	5.4.1 加载可执行程序	211
4.6.5 配置内核支持 NETLINK	142	5.4.2 进程的投入运行	223
4.6.6 配置内核支持 inotify	143	5.4.3 按需载入指令和数据	234
4.6.7 安装 modules.alias.bin		5.4.4 加载动态链接器	243
		5.4.5 加载动态库	246
		5.4.6 重定位动态库	250
		5.4.7 重定位可执行程序	268
		5.4.8 重定位动态链接器	271
		5.4.9 段 RELRO	274
第 6 章 构建根文件系统	278		
6.1 初始根文件系统	278		
6.2 以读写模式重新挂载文件系统	280		
6.3 配置内核支持网络	282		

6.3.1 配置内核支持 TCP/IP 协议	282	7.1.3 主要数据结构	328
6.3.2 配置内核支持网卡	283	7.1.4 初始化	331
6.4 启动 udev	285	7.1.5 为窗口“落户”	334
6.5 安装网络配置工具并配置网络	285	7.1.6 构建窗口装饰	337
6.6 安装并配置 ssh 服务	287	7.1.7 绘制装饰窗口	341
6.7 安装 procps	291	7.1.8 配置窗口	343
6.8 安装 X 窗口系统	291	7.1.9 移动窗口	345
6.8.1 安装 M4 宏定义	292	7.1.10 改变窗口大小	348
6.8.2 安装 X 协议和扩展	292	7.1.11 切换窗口	348
6.8.3 安装 X 相关库和工具	294	7.1.12 最大化 / 最小化 / 关闭 窗口	351
6.8.4 安装 X 服务器	296	7.1.13 管理已存在的窗口	354
6.8.5 安装 GPU 的 2D 驱动	297	7.2 任务条和桌面	356
6.8.6 安装 X 的输入设备驱动	297	7.2.1 标识任务条的身份	357
6.8.7 运行 X 服务器	300	7.2.2 更新任务条上的任务项	358
6.8.8 一个简单的 X 程序	302	7.2.3 激活任务	359
6.8.9 配置内核支持 DRM	303	7.2.4 高亮显示当前活动任务	360
6.9 安装图形库	307	7.2.5 显示桌面	361
6.9.1 安装 GLib 和 libffi	307	7.2.6 桌面	362
6.9.2 安装 ATK	307		
6.9.3 安装 libpng	308		
6.9.4 安装 GdkPixbuf	308		
6.9.5 安装 Fontconfig	308		
6.9.6 安装 Cairo	311		
6.9.7 安装 Pango	311		
6.9.8 安装 libXi	311		
6.9.9 安装 GTK	312		
6.9.10 安装 GTK 图形库的善后 工作	312		
6.9.11 一个简单的 GTK 程序	313		
6.10 安装字体	315		
第 7 章 构建桌面环境	317		
7.1 窗口管理器	317	8.1 渲染和显示	364
7.1.1 基本原理	318	8.1.1 渲染	365
7.1.2 创建编译脚本	325	8.1.2 显示	365
8.2 显存	366	8.2 动态显存技术	367
8.3 2D 渲染	375	8.2.1 Buffer Object	370
8.3.1 创建前缓冲	377	8.3 2D 渲染	375
8.3.2 GPU 渲染	381	8.3.1 创建前缓冲	377
8.3.3 CPU 渲染	386	8.3.2 GPU 渲染	381
8.4 3D 渲染	388	8.3.3 CPU 渲染	386
8.4.1 创建帧缓冲	390	8.4 动态帧缓冲	388
8.4.2 渲染 Pipeline	399	8.4.1 创建帧缓冲	390
8.4.3 交换前缓冲和后缓冲	414	8.4.2 渲染 Pipeline	399
8.5 Wayland	421	8.4.3 交换前缓冲和后缓冲	414



第 1 章

准备基本环境

在开始 Linux 操作系统的探索旅程之前，我们首先需要准备一下环境，读者最好在真实的计算机上安装一个 Linux 操作系统作为工作机。毫无疑问，使用是最好的学习方法，如果日常工作系统也是 Linux，那么这无疑有助于更好地理解 Linux 操作系统。但是这不是必须的，也可以安装一台虚拟机作为工作机。鉴于现在的 Linux 发行版的安装过程非常友好和自动化，本章无意浪费版面介绍其安装过程。

另外，在构建操作系统时，需要频繁重启系统，因此强烈建议读者不要使用工作机作为实验机，而是另外安装一台虚拟机作为实验机。本章将介绍如何创建一个虚拟的裸机以及如何在其上安装 Linux 操作系统，并且介绍为了后面的开发和调试，在虚拟机上需要进行的一些必要的准备。

因为桌面环境可以利用一个模拟的小 X 服务器 Xephyr 来调试，所以我们可以先在宿主机的 Xephyr 上进行开发和调试，然后再到构建的真实系统上调试。因此，本章的最后一部分介绍了如何使用 Xephyr。

1.1 安装 VirtualBox

笔者建议在真实的计算机上安装一个 Linux 操作系统，这个系统作为工作机，主要进行编译、构建和开发，另外辅助提供做一些实验及阅读源代码等。理论上使用哪家的发行版或者哪个版本都可以，但是为了避免意外的麻烦，建议使用和笔者相同的环境。在写作这本书的最后，笔者使用 Ubuntu12.10 将构建过程全部验证了一遍，所以建议读者也使用这个版本。

另外，我们当然不希望使用工作机调试我们构建的操作系统，因为这样需要频繁的启动。所以我们需要一个虚拟机，笔者使用的虚拟机是 VirtualBox。在 Ubuntu12.10 下，使用如下命令安装 VirtualBox：

```
root@baisheng:~# apt-get install virtualbox
```

因为我们是从零开始构建系统，因此虚拟机上还需要一个额外的 Linux 系统作为

桥梁。鉴于其只是一个桥梁，所以使用什么版本没有关系，比如笔者虚拟机上使用的是 Ubuntu11.10。

1.2 创建虚拟计算机

在安装 Linux 操作系统之前，我们需要从硬件层面创建一个虚拟的计算机。VirtualBox 启动后，主界面如图 1-1 所示。

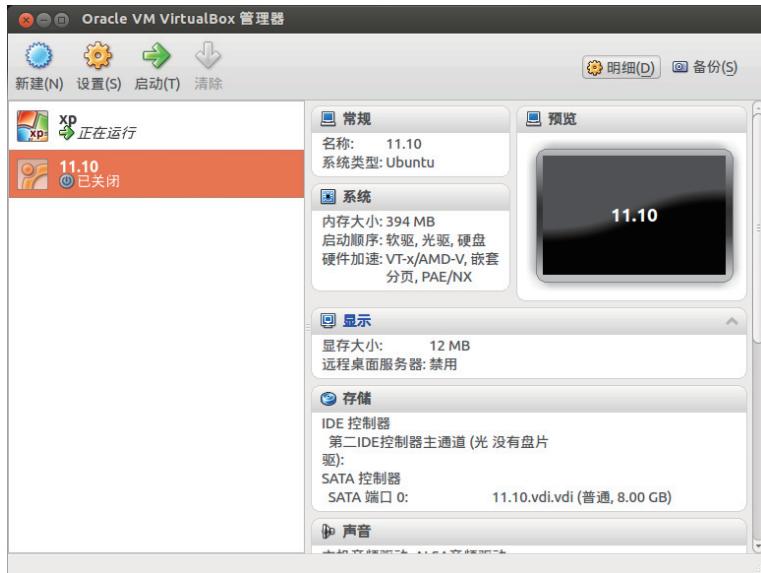


图 1-1 VirtualBox 主界面

单击图 1-1 中 VirtualBox 主界面工具条中的“新建”按钮，新建虚拟机的向导将启动。这个过程非常简单，读者按照新建向导一路执行下去就好。读者只需要注意在安装过程中要选择安装 Linux 操作系统，其他全部默认即可。

创建好虚拟机后，在 VirtualBox 主界面中将出现新建的虚拟裸机，如图 1-2 所示，其中，ubuntu11.10 就是笔者新创建的虚拟机。

1.3 安装 Linux 系统

本节我们将在 1.2 节创建的裸机上安装 Linux 操作系统。

在图 1-2 所示的工具栏上单击“设置”按钮，当然要确保在左侧的列表中选中的是刚刚创建的裸机，出现如图 1-3 所示的界面。

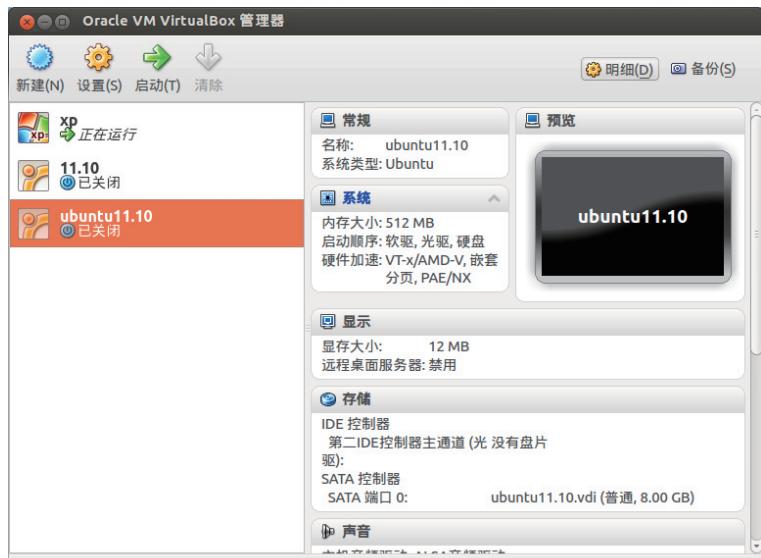


图 1-2 新建的虚拟裸机

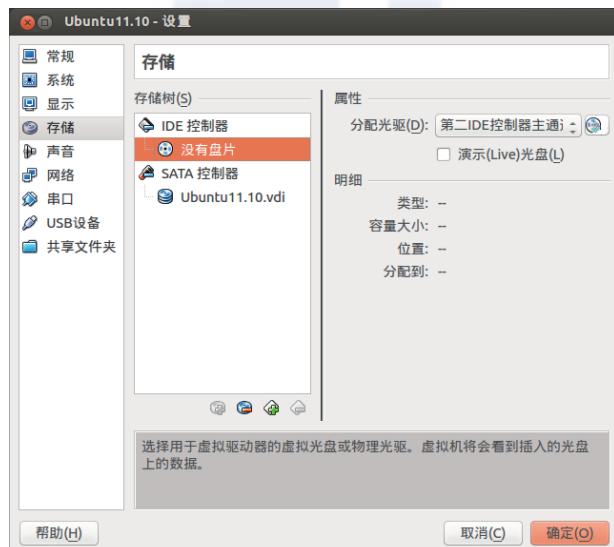


图 1-3 载入虚拟光盘映像

在图 1-3 中，首先在左侧的列表中选择“存储”。在默认情况下，我们会看到虚拟机已经添加了一个空的虚拟光驱。如果 VirtualBox 没有自动添加，读者手动添加即可。至于是 SATA 接口还是 IDE 接口，是没有关系的，毕竟是虚拟的。然后，选中虚拟光驱，即图 1-3 所示的 IDE 控制器下的“没有盘片”，然后单击“分配光驱”文本框旁的带有光盘图片的按

钮。VirtualBox 将打开一个文件选择对话框，读者找到 Linux 操作系统的光盘映像即可。这个过程与我们将物理光盘放入光驱道理完全相同。

在将光盘映像放入虚拟光驱后，在图 1-2 所示的界面中单击工具栏上的“启动”按钮，启动 Linux 系统的安装过程。鉴于现在的发行版的安装过程非常友好且全程自动化，我们就不再浪费太多版面逐一介绍。其中需要读者重点关注的是一定要从硬盘中为我们即将构建的系统划分出一块分区，基本上 2GB 就足够了，并将其格式化为 EXT4 类型，当然后面这一步也可以在系统安装完成后进行。

在安装过程中，在选择安装类型（installation type）这一步，务必要选择“Something else”，如果选择了使用中文简体安装，这里显示的可能是“其他选项”，总之，要选择这个允许我们为硬盘分区的选项，如图 1-4 所示。

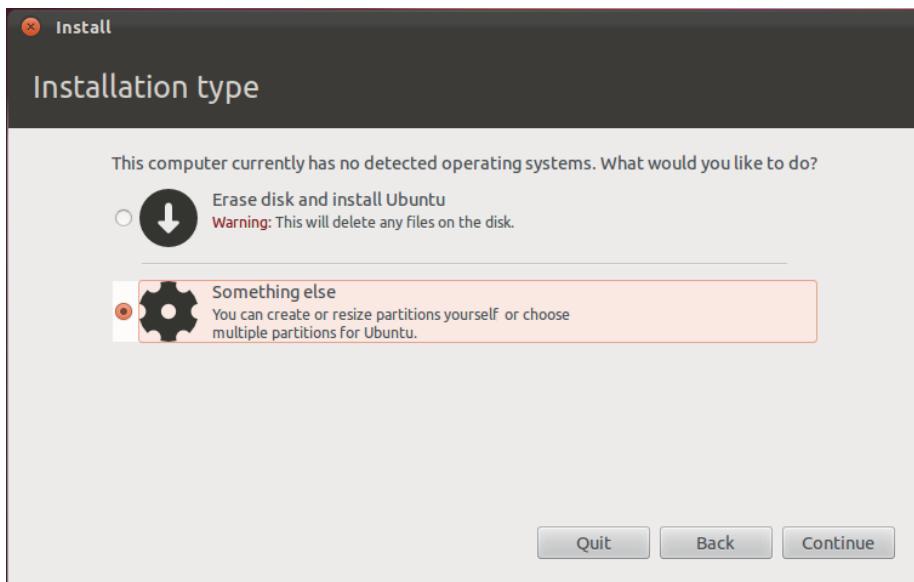


图 1-4 选择安装类型

单击图 1-4 中的继续（Continue）按钮，将出现硬盘分区的界面。基本上划分两个分区就可以了，一个用来安装操作系统，另外一个作为“实验田”，留给我们构建的操作系统用于实验。划分好的分区大致如图 1-5 所示。

另外，还有一处需要提醒读者，在安装的后期，安装程序可能会通过网络更新系统，因为这个虚拟机上的系统只是一个桥梁，没有太多工作要做，一个基本的系统就足够了，所以完全没有必要浪费时间等待其下载更新，直接略过（skip）即可。

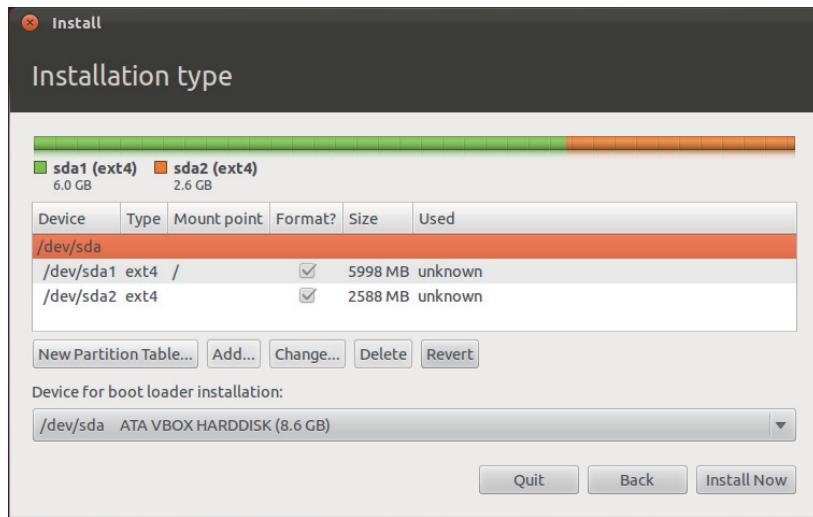


图 1-5 硬盘分区

1.4 使用 root 用户

很多发行版由于安全原因，默认使用普通用户登录，因此当要执行一些需要特权的操作时，往往需要通过“sudo”命令使自己临时成为root用户。但是这对于我们希望研究操作系统的人来说，当然有点不方便了，所以，笔者建议使用root用户登录。

既然打算使用root用户，当然要知道root用户的密码了，但是Ubuntu默认的root用户密码是什么呢？不必理会这个问题，直接改成我们自己的即可。以普通用户登录虚拟机后，启动一个终端，执行如下命令修改root用户密码：

```
sudo passwd root
```

然后就可以使用root用户了，或者使用命令“su”切换用户，或者登录时使用root用户。

1.5 启用自动登录

在安装步骤中，在添加用户这一步的界面中，有一个可选项，即“自动登录”(log in automatically)，这个选项默认是没有选中的。如果没有选中，那么在启动时，每次登录都需要输入登录密码，非常麻烦。所以，建议读者开启自动登录。

如果安装时没有选中，也不必重新安装。读者可以修改登录管理器lightdm的配置文件lightdm.conf，在其中添加下面一行：

```
/etc/lightdm/lightdm.conf:
```

```
autologin-user=root
```

如果读者实在不愿意敲击键盘输入这几个字母，那么可以在系统设置中，打开“用户账户”（User Accounts），将普通账户的“自动登录”（Automatic Login）开启，然后在配置文件 lightdm.conf 中将多出类似下面一行：

```
/etc/lightdm/lightdm.conf:
```

```
autologin-user=baisheng
```

读者将其中的普通用户的登录名改为 root 即可。

如此，即可免除每次登录时输入密码之苦，也无需手动切换用户，而是自动以 root 身份登录。

1.6 挂载实验分区

假设在虚拟机上为构建的操作系统划分的分区是 /dev/sda2，那么我们使用如下命令将其挂载在根目录的 vita 下：

```
mkdir /vita  
mount /dev/sda2 /vita
```

为了避免每次开机后都需要手工挂载，我们将其写入 fstab 文件中，开机后由操作系统自动挂载：

```
/etc/fstab:  
  
/dev/sda2 /vita ext4 defaults 0 0
```

1.7 安装 ssh 服务器

我们使用 ssh 服务从宿主系统向虚拟机复制构建的实验系统。因此，在虚拟机系统上需要安装 ssh 服务器。ssh 服务器需要通过网络从源服务器下载。以笔者使用的 VirtualBox 版本为例，默认其为虚拟机开启了网络，并且使用的是 NAT 模式，要访问互联网，无需设置 IP、路由等，但是要自己设置 DNS。或者直接可以进入设置，将虚拟机的网络改为桥接模式，这样在 DHCP 的网络环境中，无须做任何修改即可访问互联网。

确保虚拟机可以访问互联网后，我们就可以安装 ssh 服务器了。当然首次从源安装软件时，需要更新源。更新源和安装 ssh 服务的命令如下：

```
apt-get update  
apt-get install openssh-server
```

1.8 更改网络模式

在 VirtualBox 的各种网络模式中，允许宿主机和虚拟机通信的常用网络模式是桥接模式和 Host-Only 模式。但是桥接模式有两个问题，一个是宿主机一定要时刻连网，因为在桥接模式下，虚拟机在局域网内被模拟为与宿主机同等地位的一台主机，所以如果宿主机没有接入局域网，何谈虚拟机和宿主机通信？虽然现在网络很普及，但是毕竟会存在未接入网络的情况。另外一个问题，我们也不想让开着 ssh 服务器的虚拟机暴露在互联网上。所以，笔者建议虚拟机的网络使用 Host-Only 模式，设置方法如图 1-6 所示。

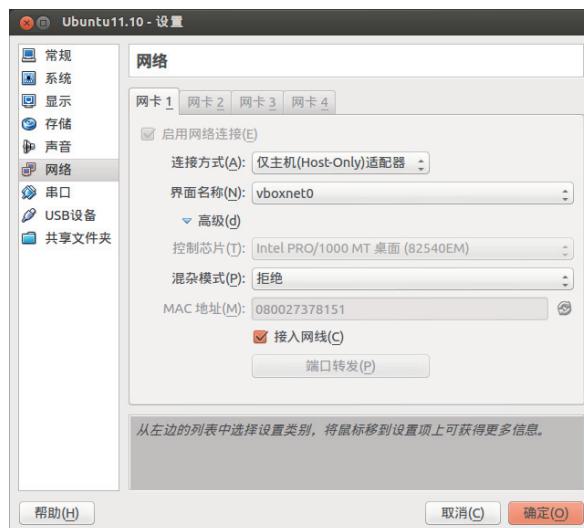


图 1-6 设置虚拟机网络模式

在图 1-6 中，首先选中左侧列表中的“网络”，然后将“连接方式”更改为“Host-Only”模式。

确定后，宿主系统将多出一个网络接口，用于与虚拟机通信，默认一般是 vboxnet0，其地址被设置为 192.168.56.1，虚拟机的地址被设置为 192.168.56.101。当然读者可以自己修改，但是这没有任何必要。

然后在虚拟机上我们就可以使用如下命令启动 ssh 服务器了：

```
/usr/sbin/sshd
```

在宿主系统上，我们可以远程登录到虚拟机，命令如下：

```
ssh 192.168.56.101
```

也可以将宿主系统的文件（比如 a）复制到虚拟机，命令如下：

```
scp a 192.168.56.101:/root/
```

1.9 安装增强模式

当没有安装增强模式时，虚拟机只能使用固定的分辨率，那么可能不支持全屏这样的功能。如果需要全屏功能，可以选择安装 VirtualBox 的增强功能来解决这一问题。

在 VirtualBox 的菜单中，首先选择“设备”菜单；然后在下拉菜单中选择“安装增强功能”。

增强功能也在一个光盘映像中，所以如果是首次安装增强功能，VirtualBox 将首先从网络上下载这个光盘映像到宿主机。下载完成后，这个光盘映像一般会被自动装入到虚拟光驱，如果没有自动挂载，需要读者手动将其放入到虚拟光驱，然后，运行其中的“VBoxLinuxAddtions.run”即可。当然如果是为 Windows 系统安装增强功能，需要运行相应的 Windows 版本。

1.10 使用 Xephyr

在宿主系统上使用 Xephyr 调试桌面环境，要更方便一些。所以这一节，我们介绍如何在宿主系统上调试桌面环境。如果尚未安装 Xephyr，则首先通过如下方法安装 Xephyr：

```
root@baisheng:~# apt-get install xserver-xephyr
```

然后使用如下命令启动 Xephyr：

```
root@baisheng:~# Xephyr -ac -screen 800x480 :1.0
```

在另外的终端中，将 Display 定向到 Xephyr：

```
root@baisheng:~# export DISPLAY=:1.0
```

如此，在这个终端中，所有需要 X 服务器渲染程序都将使用 Xephyr。当然，为了方便，我们可以开启任意个终端，并将它们的 Display 都定向到 Xephyr。

比如我们可以在一个终端中运行窗口管理器 winman：

```
root@baisheng:~# export DISPLAY=:1.0
root@baisheng:/vita/build/winman/src# ./winman
```

在另外一个终端中运行任务条：

```
root@baisheng:~# export DISPLAY=:1.0
root@baisheng:/vita/build/taskbar/src# ./taskbar
```

而在第三个终端中运行 Desktop 程序：

```
root@baisheng:~# export DISPLAY=:1.0
root@baisheng:/vita/build/desktop/src# ./desktop
```

图 1-7 就是在笔者的宿主机上运行 winman 以及一个 gedit 后的 Xephyr。

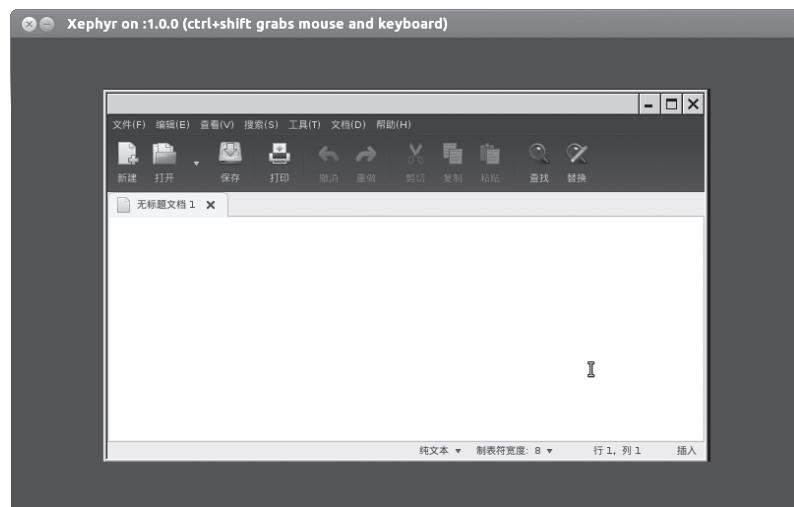


图 1-7 Xephyr

第 2 章

工具链

软件的编译过程中由一系列的步骤完成，每一个步骤都有一个对应的工具。这些工具紧密地工作在一起，前一个工具的输出是后一个工具的输入，像一根链条一样，因此，人们也把这些工具的组合形象地称为工具链。

在本书中，我们将从源码开始，逐步构建一个基本的 Linux 操作系统。显然，工具链是我们首先需要考虑的，因为工具链是编译包括内核在内的操作系统各个组件的基础。正所谓“物有本末，事有终始，知所先后，则近道矣。”因此，在本章中，我们并没有匆忙切入正题——构建工具链，而是首先结合具体的例子，借助宿主系统中的工具，尽可能地将工具链的工作过程更具体地展示给读者。希望通过这个探讨过程，读者可以明白工具链包含哪些组件以及这些组件的基本工作原理。然后基于 GNU 工具链的源码，手工从源码构建一套工具链。后面，我们将会使用这一章构建的工具链编译内核以及操作系统的各个组件。

2.1 编译过程

华章图书

在 Linux 系统上，通常，只需使用 `gcc` 就可以完成整个编译过程。但不要被 `gcc` 的名字误导，事实上，`gcc` 并不是一个编译器，而是一个驱动程序（driver program）。在整个编译过程中，`gcc` 就像一个导演一样，编译过程中的每一个环节由具体的组件负责，如编译过程由 `cc1` 负责、汇编过程由 `as` 负责、链接过程由 `ld` 负责。

我们可以通过传递参数 “`-v`” 给 `gcc` 来观察一个完整的编译过程中包含的步骤，下面是一个典型的编译过程中 `gcc` 的输出信息，为了更清楚地看到编译过程中的主要步骤，对输出信息进行了适当删减。

```
root@baisheng:~/demo# gcc -v main.c
...
/usr/lib/gcc/i686-linux-gnu/4.7/cc1 -quiet -v -imultiarch
i386-linux-gnu main.c -quiet -dumpbase main.c -mtune=generic
-march=i686 -auxbase main -version -fstack-protector -o
tmp/ccYBInzt.s
...
```

```

as -v --32 -o /tmp/ccj54pkM.o /tmp/ccYBInzt.s
...
/usr/lib/gcc/i686-linux-gnu/4.7/collect2 --sysroot=/ --build-id
--no-add-needed --as-needed --eh-frame-hdr -m elf_i386
--hash-style=gnu -dynamic-linker /lib/ld-linux.so.2 -z relro
/usr/lib/gcc/i686-linux-gnu/4.7/../../../i386-linux-gnu/crti.o
/usr/lib/gcc/i686-linux-gnu/4.7/../../../i386-linux-gnu/crti.o
/usr/lib/gcc/i686-linux-gnu/4.7/crtbegin.o
-L/usr/lib/gcc/i686-linux-gnu/4.7
-L/usr/lib/gcc/i686-linux-gnu/4.7/../../../../i386-linux-gnu
-L/usr/lib/gcc/i686-linux-gnu/4.7/../../../../lib
-L/lib/i386-linux-gnu -L/lib/.. -L/usr/lib/i386-linux-gnu
-L/usr/lib/.. -L/usr/lib/gcc/i686-linux-gnu/4.7/../../..
/tmp/ccj54pkM.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc
--as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/i686-linux-gnu/4.7/crtend.o
/usr/lib/gcc/i686-linux-gnu/4.7/../../../../i386-linux-gnu/crtn.o

```

根据 gcc 的输出可见，对于一个 C 程序来说，从源代码构建出可执行程序经过了三个阶段：

(1) 编译

gcc 调用编译器 cc1 进行编译，产生的汇编代码保存在目录 /tmp 下的文件 ccYBInzt.s 中。

(2) 汇编

gcc 调用汇编器 as，汇编编译过程产生汇编文件 cca2nBio.s，产生的目标文件保存在目录 /tmp 下的文件 ccj54pkM.o 中。

(3) 链接

我们看到，gcc 并没有如我们想象的那样直接调用 ld 进行链接，而是调用 collect2 进行链接。实际上，collect2 只是一个辅助程序，最终它仍将调用链接器 ld 完成真正的链接过程。举个例子，对于 C++ 程序来说，在执行 main 函数前，全局静态对象必须构造完成。也就是说，在 main 之前程序需要进行一些必要的初始化，gcc 就是使用 collect2 安排初始化过程中如何调用各个初始化函数的。根据链接过程可见，除了 main.c 对应的目标文件 ccj54pkM.o 外，ld 也链接了 libc、libgcc 等库，以及所谓的包含启动代码（start code）的启动文件（start/startup file），包括 crt1.o、crti.o、crtbegin.o、crtend.o 和 crtn.o。

事实上，对于 C 程序来说，编译过程也可以拆分为两个阶段：预编译（或称为编译预处理）和编译。所以，软件构建过程通常分四个阶段：预编译、编译、汇编以及链接，如图 2-1 所示。

在接下来讨论编译过程的章节中，如无特殊说明，都将以下面的程序为例。

```

foo1.c:

int fool = 10;

void fool_func()
{

```

```

        int ret = fool;
    }

foo2.c:

int foo2 = 20;

void foo2_func(int x)
{
    int ret = foo2;
}

hello.c:

#include <stdio.h>

extern int foo2;

int main(int argc, char *argv[])
{
    foo2 = 5;
    foo2_func(50);
    return 0;
}

```

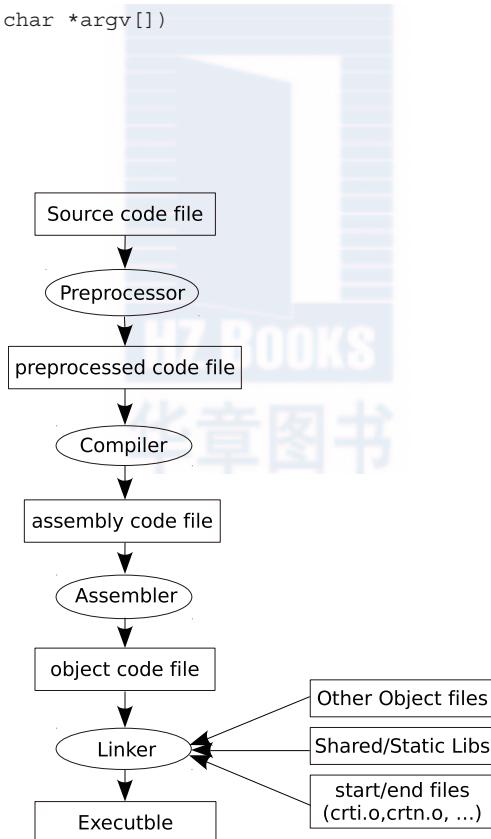


图 2-1 C 程序的构建过程

2.1.1 预编译

在预编译阶段，预编译器将处理源代码中的预编译指令。一般而言，C 语言中的预编译

指令以“#”开头，常用的预编译指令包括文件包含命令“#include”、宏定义“#define”，以及条件编译命令“#if”、“#else”、“#endif”等。

在工具链中，一般都提供单独的预编译器，比如GCC中提供的预编译器为cpp。但是，因为预编译也可以看作编译过程的第一遍（first pass），是为编译做的一些准备工作，所以通常编译器中也包含了预编译的功能。如在前面的编译过程中，我们看到gcc并没有单独调用cpp，而是直接调用cc1进行编译，原因就在于此。

以下面的程序为例，该程序使用了典型的预编译指令，我们通过观察这个程序的预处理的结果，来直观体会预编译过程。

```
foo.h:
#ifndef _FOO_H_
#define _FOO_H_

#define PI 3.1415926
#define AREA

struct foo_struct {
    int a;
};

#endif

hello.c:
#include "foo.h"

int main(int argc, char *argv[])
{
    int result;
    int r = 5;

#ifdef AREA
    result = PI * r * r;
#else
    result = PI * r * 2;
#endif
}
```

我们可以使用选项-E告诉编译器仅作预处理，不进行编译、汇编和链接，具体命令如下：

```
gcc -E hello.c -o hello.i
```

预编译后的结果保存在文件hello.i中，其内容如下：

```
struct foo_struct {
    int a;
};
```

```

int main(int argc, char *argv[])
{
    int result;
    int r = 5;

    result = 3.1415926 * r * r;
}

```

根据预编译后的结果可见，典型的预编译指令按照如下方式进行处理。

(1) 文件包含

文件包含命令指示预编器将一个源文件的内容全部复制到当前源文件中。在上面的代码中，hello.c 使用命令“#include”指示预编器包含文件 foo.h。而在预处理的输出文件 hello.i 中，结构体 foo_struct 的定义确实已经被复制到了文件 hello.i 中，也就是说，文件 foo.h 中的内容被包含到了文件 hello.i 中。

(2) 宏定义

宏可以提高程序的通用性和易读性，减少不一致性和输入错误，便于维护。在预处理过程中，预编器将宏名替换为具体的值。比如，在 hello.c 的 main 函数中，经过预处理后，宏名 PI 已经被替换为具体的值 3.1415926。

(3) 条件编译

在大多数情况下，源程序中所有的语句都参加编译，但有的时候用户希望按照一定的条件去编译源程序的不同部分，这时可以使用条件编译。比如在函数 main 中，当定义了变量 AREA 时，编译器将编译“#ifdef”块的代码，否则编译“#else”块的代码。在上面代码中，因为在 foo.h 中定义了变量 AREA，所以，在经过预处理的文件 hello.i 中，条件编译指令中的“#else”块的代码从源代码中被删除了，只保留了“#ifdef”块的代码。

2.1.2 编译

编译程序对预处理过的结果进行词法分析、语法分析、语义分析，然后生成中间代码，并对中间代码进行优化，目标是使最终生成的可执行代码执行时间更短、占用的空间更小，最后生成相应的汇编代码。

以 foo2.c 为例，我们可以使用如下命令指定编译过程只进行编译，不进行汇编和链接。

```
root@baisheng:~/demo# gcc -S foo2.c
```

编译后产生的汇编文件为 foo2.s，其内容如下：

```

.file   "foo2.c"
.globl  foo2
.data
.align 4
.type   foo2, @object
.size   foo2, 4
foo2:

```

```

.long    20
.text
.globl  foo2_func
.type   foo2_func, @function
foo2_func:
.LFB0:
.cfi_startproc
pushl   %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl    %esp, %ebp
.cfi_def_cfa_register 5
subl    $16, %esp
movl    foo2, %eax
movl    %eax, -4(%ebp)
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size   foo2_func, .-foo2_func
.ident  "GCC: (Ubuntu/Linaro 4.7.2-2ubuntu1) 4.7.2"
.section .note.GNU-stack,"",@progbits

```

在文件 foo2.c 中，除定义了一个全局变量 foo2 外，仅定义了一个函数 foo2_func，而该函数体中也只有区区一行代码，但为什么产生的汇编代码如此之长？事实上，仔细观察可以发现，文件 foo2.s 中相当一部分是汇编器的伪指令。伪指令是不参与 CPU 运行的，只指导编译链接过程。比如，代码中以“.cfi”开头的伪指令是辅助汇编器创建栈帧（stack frame）信息的。

在终端上调试程序的程序员一般都会有这样的经历：某个程序出现 Segment Fault 了，然后终端中会输出回溯（backtrace）信息。或者，我们在调试程序时，也经常需要回溯，查找一些变量或查看函数调用信息。这个过程，就是所谓的栈的回卷（unwind stack）。事实上，在每个函数调用过程中，都会形成一个栈帧，以 main 函数调用 foo2_func 为例，形成的栈帧如图 2-2 所示。

frame pointer 和 base pointer 均指向栈帧的底部，只是叫法不同，在 IA32 架构中，通常使用寄存器 ebp 保存这个位置。因为 main 并不是程序中第一个运行的函数，所以 main 也是一个被调函数，其也有栈帧。事实上，即使程序中第一个被调用的函数_start（该函数实现启动代码中），也会自己模拟一个栈帧。

理论上，调试器或异常处理程序完全可以根据 frame pointer 来遍历调用过程中各个函数的栈帧，但是因为 gcc 的代码优化，可能导致调试器或异常处理很难甚至不能正常回溯栈帧，所以这些伪指令的目的就是辅助编译过程创建栈帧信息，并将它们保存在目标文件的段“.eh_frame”中，这样就不会被编译器优化影响了。

去掉这些伪指令后，函数 foo2_func 中 CPU 真正执行的代码如下：

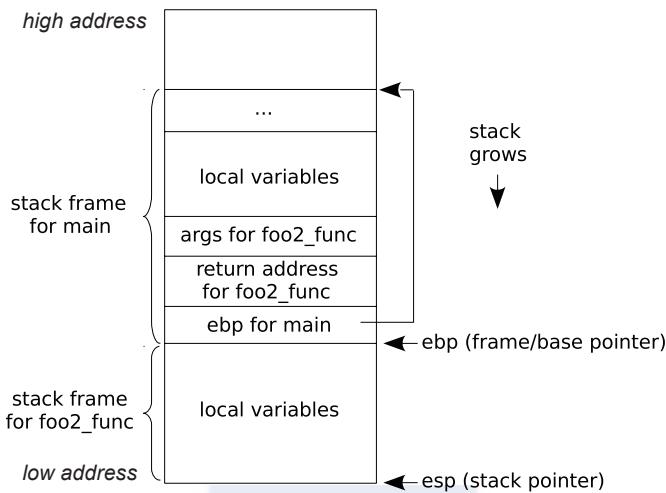


图 2-2 函数调用中的栈帧

```

foo2_func:
1    pushl  %ebp
2    movl  %esp, %ebp
3    subl $16, %esp
4    movl foo2, %eax
5    movl %eax, -4(%ebp)
6    leave
7    ret

```

在汇编语言中，在函数的开头和结尾处分别会插入一小段代码，分别称为 Prologue 和 Epilogue，如 foo2_func 中的第 1、2、3 行代码就是 Prologue，第 6、7 行代码就是 Epilogue。

Prologue 保存主调函数的 frame pointer，这是为了在子函数调用结束后，恢复主调函数的栈帧。同时为子函数准备栈帧。其主要操作包括：

- 保存主调函数的 frame pointer，如第 1 行代码所示，就是将保存在寄存器 ebp 中的 frame pointer 压栈。在退出子函数时可以从栈中恢复主调函数的 frame pointer。
- 将 esp 赋值给 ebp，即将子函数的 frame pointer 指向主调函数的栈顶，如第 2 行代码所示。换句话说，这行代码的意义就是记录了子函数的栈帧的底部，从这里就开始了子函数的栈帧。
- 修改栈顶指针 esp，为子函数的本地变量分配栈空间，如第 3 行代码。注意虽然这里的 foo2_func 中只有一个局部变量 ret，占据 4 字节，但是还是预留了 16 字节的栈空间，这根据的是 IA32 的 ABI (Application Binary Interface) 的 16 字节的对齐要求。

Epilogue 功能与 Prologue 恰恰相反，如果说 Prologue 相当于构造函数，那么 Epilogue 就相当于析构函数。其主要操作包括：

- 将栈指针 esp 指向当前子函数的栈帧的 frame pointer，也就是说，指向当前栈帧的栈底，而在这个位置，恰恰是 Prologue 保存的主调函数的 frame pointer。然后，通过

指令 pop 将主调函数的 frame pointer 弹出到 ebp 中，如此，一方面释放了被调函数 foo2_func 的栈帧，同时也回到了主调函数 main 的栈帧。IA32 提供了指令 leave 来完成这个功能，即第 6 行代码，这个指令相当于：

```
movl %ebp, %esp
pop %ebp
```

- 将调用子函数时 call 指令压栈的返回地址从栈顶 pop 到 EIP 中，并跳转到 EIP 处继续执行。如此，CPU 就返回到主调函数继续执行。IA32 提供了指令 ret 来完成这个功能，即第 7 行代码。

除了 Prologue 和 Epilogue，foo2_func 的核心代码就剩下第 4 行和第 5 行两行了。这两行代码对应的就是 C 语言中的赋值语句 “int ret = foo2”。首先，即第 4 行代码，CPU 从数据段中读取全局变量 foo2 的值到寄存器 EAX 中。然后，即第 5 行代码，将 eax 中的内容，即 foo2 的值，复制到栈中的局部变量 ret 的位置。代码中根据局部变量相对于栈的 frame pointer（在 ebp 中保存）的偏移来访问局部变量，如变量 ret 位于相对于栈底偏移为 -4 的内存处。

2.1.3 汇编

汇编器将汇编代码翻译为机器指令，每一条汇编语句几乎都对应一条机器指令，所以汇编器的汇编过程相对于编译器来讲比较简单，它没有复杂的语法，也没有语义，也不需要做指令优化，只是根据汇编指令和机器指令的对照表进行翻译就可以了。当然，汇编器的工作不仅包括翻译汇编指令到机器指令，除了生成机器码外，汇编器还要在目标文件中创建辅助链接时需要的信息，包括符号表、重定位表等。

1. 目标文件

汇编过程的产物是目标文件，同前面的预编译和编译阶段产生的文本文件不同，目标文件的格式更复杂，其中包括链接需要的信息，所以在理解汇编过程前，我们需要了解一下目标文件的格式。Linux 下的二进制文件包括可执行文件、静态库和动态库等，均采用 ELF 格式存储，目标文件的格式也不例外，也采用 ELF 格式存储。

对于 32 位的 ELF 文件来说，其最前部是文件头部信息，描述了整个文件的基本属性，除了包括该文件运行在什么操作系统中、运行在什么硬件体系结构上、程序入口地址是什么等基本信息外，最重要的是记录了两个表格的相关信息，如表格所在的位置、其中包括的条目数等。这两个表格一个是 Section Header Table，主要是供编译时链接使用的，表格中定义了各个段的位置、长度、属性等信息；另外一个是 Program Header Table，主要是供内核和动态加载器从磁盘加载 ELF 文件到内存时使用的。对于目标文件，由于其只是编译过程的一个中间产物，不涉及装载运行，因此，在目标文件中不会创建 Program Header Table。

在后续内容中，我们将 Segment 和 Section 都翻译为段，读者可根据上下文区分。在有的上下文中，段指的是真正加载到内存中的 Segment，而有的指的是 ELF 中链接时使用的 Section。

下面我们通过命令 `readelf` 列出目标文件 `foo2.o` 的 ELF 头信息。

```
root@baisheng:~/demo# gcc -c hello.c foo1.c foo2.c
root@baisheng:~/demo# readelf -h foo2.o
ELF Header:
  Magic:    7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:          ELF32
  Data:           2's complement, little endian
  Version:        1 (current)
  OS/ABI:         UNIX - System V
  ABI Version:   0
  Type:           REL (Relocatable file)
  Machine:       Intel 80386
  Version:        0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 264 (bytes into file)
  Flags:          0x0
  Size of this header: 52 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 40 (bytes)
  Number of section headers: 12
  Section header string table index: 9
```

`foo2.o` 的 ELF 头占用了 52 字节，通过 ELF 头可见该文件是 32 位的 ELF 文件；使用“little endian”字节序存储字节；ABI 遵循 UNIX-System V 标准；运行在类 UNIX 系统上；该文件是一个“REL (Relocatable file)”类型的文件，通常，可执行文件的类型是“EXEC (Executable file)”，动态共享库的类型是“DYN (Shared object file)”，静态库和目标文件的类型是“REL (Relocatable file)”；该目标文件是为 IA32 架构编译的；因为是目标文件，不存在执行的概念，所以程序入口“Entry point address”在这里不适用（同样的道理，Program Header Table 也不适用）；`foo2.o` 中的 Section Header Table 在偏移 264 字节处，Section Header Table 中的每个 Section Header 占用 40 字节，Section Header Table 共包含 12 个 Section Header。

在文件头信息后，就是各个段了。毫不夸张地说，ELF 文件就是段的组合。大体上，段可以分为如下几类：一类是存储指令的，通常称为代码段；第二类是存储数据的，通常称为数据段。但是存储数据的又细分为两个段，已经初始化的全局数据存放在“`.data`”段中，未初始化的全局数据存储在“`.bss`”段。不要被 BSS 这个令人困惑的名称迷惑，这个名称不是非常贴切，完全是历史遗留的，“`.data`”段和“`.bss`”段本质并没有什么不同，但是因为未初始化的变量不包含数据，所以在 ELF 文件中不需要占用空间，程序装载时在内存中即时分配就可以了。所以，为了节省存储器空间，人为地将存储数据的部分划分为两个段。除了最重要的代码段和数据段外，汇编器还将在目标文件中创建辅助链接段，存储如符号表、重定位表等。

我们考察目标文件 `foo2.o` 的 Section Header Table，因为排版篇幅的关系，删除了后面几列，这不影响我们讨论。有兴趣的读者，可以自行查看完整的命令输出。

```
root@baisheng:~/demo# readelf -S foo2.o
There are 12 section headers, starting at offset 0x104:
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size
[0]		NULL	00000000	000000	000000
[1]	.text	PROGBITS	00000000	000034	000010
[2]	.rel.text	REL	00000000	00039c	000008
[3]	.data	PROGBITS	00000000	000044	000004
[4]	.bss	NOBITS	00000000	000048	000000
[5]	.comment	PROGBITS	00000000	000048	00002b
[6]	.note.GNU-stack	PROGBITS	00000000	000073	000000
[7]	.eh_frame	PROGBITS	00000000	000074	000038
[8]	.rel.eh_frame	REL	00000000	0003a4	000008
[9]	.shstrtab	STRTAB	00000000	0000ac	000057
[10]	.symtab	SYMTAB	00000000	0002e4	0000a0
[11]	.strtab	STRTAB	00000000	000384	000017

根据输出可见，目标文件 foo2.o 的 Section Header Table 中包含 12 个 Section Header：

- “.text” 段存储在文件中偏移 0x34 处，占据 0x10 个字节。读者不要将 “.text” 段和进程的代码段混淆，进程的代码段不仅包括 “.text” 段，在后面链接时，我们还会看到，包括 .init、.fini 等段存储的代码都属于代码段。这些段都被映射到 Program Header Table 中的一个段，在 ELF 加载时，统一作为进程的代码段。
- “.data” 段存储在文件中偏移 0x44 字节处，占据 0x4 字节空间。
- 如我们在前面讨论的，虽然目标文件的 Section Header Table 中包含 “.bss” 段，但是因为其不必记录数据，所以 “.bss” 段在文件中只占据 Section Header Table 中的一个 Section Header，而并没有对应的段。在加载程序时，加载器将依据 “.bss” 段的 Section Header 中的信息，在内存中为其分配空间。考察程序 hello 的 Section Header Table：

```
root@baisheng:~/demo# readelf -S hello
There are 30 section headers, starting at offset 0x1198:
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size
...					
[25]	.bss	NOBITS	0804a024	<u>001024</u>	000004
[26]	.comment	PROGBITS	00000000	<u>001024</u>	00006b
...					

根据输出可见，“.bss” 段在文件中偏移为 0x001024，但是占用的空间（Size）并不是 0 字节，而是 0x4 个字节，这是为什么呢？而我们再观察 “.comment” 段在文件中的偏移，也为 0x001024。也就是说，正如我们前面讨论的，“.bss” 段在磁盘文件中并未占据任何空间，“.bss” 段的 Size 只是告诉程序加载器在加载程序时，在内存中为该段分配的内存空间。

- “.symtab” 段记录的是符号表。因为符号的名字字串长度可变，所以目标文件将符号

的名字字符串剥离出来，记录在另外一个段“.strtab”中，符号表使用符号名字的索引在段“.strtab”中的偏移来确定符号的名字。

- ❑ 同样的道理，“.shstrtab”中记录的是段的名字（sh是section header的简写）。
 - ❑ 以“rel”开头的，如“.rel.text”、“.rel.eh_frame”，记录的是段中需要重定位的符号。
 - ❑ “.eh_frame”段中记录的是调试和异常处理时用到的信息。
 - ❑ “.comment”、“.note.GNU-stack”等段如其名字所示，都是一些“comment”和“note”，无论是链接还是装载都不会用到，我们不必关心。

综上所述，目标文件的格式如图 2-3 所示。

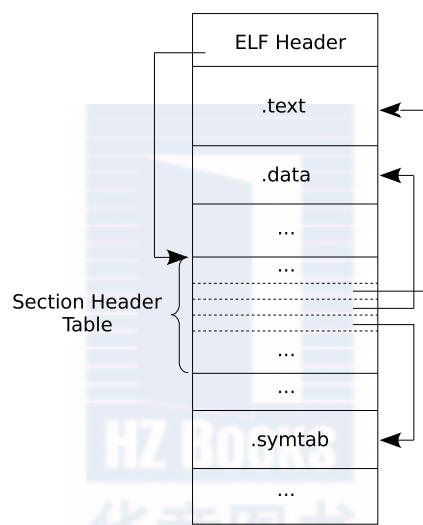


图 2-3 目标文件

2. 翻译机器指令

机器指令由操作码和操作数组成，操作码指明该指令所要完成的操作，即指令的功能，例如数据传送、加法运算等基本操作。操作数是参与操作的数据，主要以寄存器或存储器地址的形式指明数据的来源或者计算结果存放的位置等。机器指令使用计算机可以识别的 0 和 1 编码，可想而知，这对程序员来说编码难度非常大。因此，为了更容易编制出程序，就出现了汇编指令。汇编指令非常接近机器指令，但是机器指令中操作码和操作数都使用更接近自然语言的符号来代替，这类自然语言符号分别称为操作码助记符和操作数助记符。人们习惯将助记符省略，直接将操作码助记符称为操作码，将操作数助记符称为操作数，读者可根据上下文区分。

汇编过程就是将助记符翻译为对应的以 0 和 1 表示的机器指令，我们也将其称为操作码和操作数的编码过程。对于 IA32 架构，其机器指令的格式如图 2-4 所示。

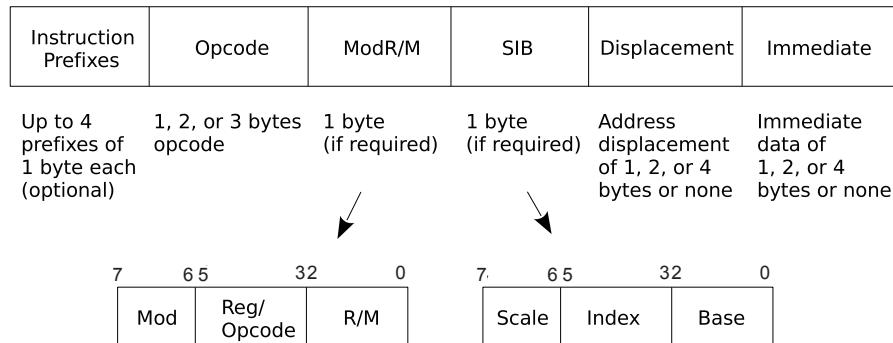


图 2-4 IA32 机器指令的格式

由图 2-4 可见，操作码 Opcode 直接嵌在指令中。操作码的翻译过程相对简单，将汇编指令中的操作码助记符翻译为相应的操作码即可，操作码助记符与操作码的对应关系可根据 CPU 的指令手册确定。

将操作数助记符翻译为操作数的机器码相对要复杂一些，操作数并没有直接嵌在指令编码中，而是根据汇编指令使用的具体寻址方式，设置 ModR/M、SIB、Displacement 和 Immediate 各项的值，这个过程称为操作数的编码。CPU 根据 ModR/M、SIB、Displacement 和 Immediate 的值，解码出操作数。

典型的操作数的编码方式包括下面几种。如果读者不太理解下面的抽象描述，没关系，后面将结合具体的 foo2.c 中的函数 foo2_func 探讨机器指令的翻译，读者可以前后结合起来理解。

(1) 操作数地址通过 ModR/M 中的 Mod + R/M 指定

ModR/M 占用 1 字节，包含三个域：Mod、Reg/Opcode 和 R/M，其中 Mod 占两位、R/M 占 3 位，Reg/Opcode 占 3 位。操作数可以使用 ModR/M 中的 Mod 和 R/M 字段联合起来定义，寻址模式与 Mod 和 R/M 联合编码的对应关系如表 2-1 所示。

表 2-1 寻址模式对应的 Mod 和 R/M 联合编码

No.	Effective Address	Mod	R/M
1	[EAX]	00	000
2	[ECX]		001
3
4	disp32		101
5
6	[EDI]		111
7	[EAX]+disp8		000
8

(续)

No.	Effective Address	Mod	R/M
9	SIB+disp8		100
10	[EBP]+disp8		101
11
12	[EAX]+disp32	10	000
13
14	[EDI]+disp32		111
15	EAX/AX/AL/MM0/XMM0	11	000
16
17	EDI/DI/BH/MM7/XMM7		111

其中第 2 列表示寻址方式生成的有效地址；第 3 列和第 4 列表示对应于某个寻址方式，Mod 和 R/M 分别对应的编码。表 2-1 中列出了包含直接寻址、寄存器寻址、寄存器间接寻址、基址寻址及基址变址寻址等寻址方式下 ModR/M 中 Mod 和 R/M 的对应的编码。如果汇编指令使用的是基址变址寻址，那么机器指令中也需要字段 SIB。

以表 2-1 中第 7 行为例，假设汇编指令使用的寻址方式是 “[EAX]+disp8”，那么 Mod 应该取值 01，R/M 应该取值 000。偏移 disp8 表示 8 位的 Displacement，根据机器指令的格式，Displacement 直接嵌在指令中即可。Displacement 根据表示的值的范围可以使用 8 位、16 位或者 32 位，这主要是出于尺寸方面的考虑。另外，在机器指令中，Displacement 需要使用补码形式。也就是说，在 CPU 执行指令时，当解析到 ModR/M 这个字节时，一旦发现 Mod 的值是 01，R/M 的值是 000，那么 CPU 就到寄存器 EAX 中取出其中的内容，然后再取出嵌在指令中的 8 位的偏移 Displacement，将这两个值相加作为操作数的内存地址，从而完成操作数的解码过程。

(2) 操作数通过 ModR/M 中的 Reg/Opcode 指定

ModR/M 中的字段 Reg/Opcode 占据 3 位。如果在汇编指令中使用了寄存器作为操作数，那么编码时也可以使用 Reg/Opcode 指定操作数使用的寄存器。如果操作数不需要使用字段 Reg/Opcode 编码，字段 Reg/Opcode 也可以用于操作码的编码。表 2-2 列出了 32 位寄存器与字段 Reg/Opcode 取值的对应关系。

表 2-2 32 位寄存器对应的 Reg/Opcode 的编码

Register	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
Reg/Opcode	000	001	010	011	100	101	110	111

(3) 操作数地址直接嵌入在机器指令中

如果在汇编指令中直接使用了操作数的地址，即所谓的直接寻址方式，那么在翻译为机器指令时，直接使用机器指令中的 Displacement 字段表示操作数的地址。

(4) 操作数直接嵌入在指令中

如果在汇编指令中，操作数就是参与计算的数据，即所谓的立即寻址，那么在翻译为机器指令时，直接使用机器指令中的 Immediate 字段表示操作数。

(5) 操作数隐含在 Opcode 中

还有一种方式，保存操作数的寄存器直接隐含在操作码 Opcode 中，即所谓的隐含寻址。

根据图 2-4 可见，除了操作码和操作数外，还有一项“Instruction Prefixes”。很难用一段话准确地描述“Instruction Prefixes”，我们可以打个比方：“Instruction Prefixes”对于机器指令类似于“Modifier key”对于键盘上的按键。Shift 键作为键盘上的“Modifier key”之一，如对于数字键 3，当同时按下 Shift 键时，其值就变为了符号“#”。如同 Shift 键只对键盘上的某些键有修饰作用一样，“Instruction Prefixes”也只对部分指令有效。

比如对于下面的两类指令，它们的功能相同，都是在两个操作数之间传递数据。只不过第一类是在两个 16 位操作数之间传递数据，第二类是在两个 32 位操作数之间传递数据。

```
mov r/m16,r16
mov r/m32,r32
```

Intel 并没有为上述两类操作分别定义两个操作码，而是使用了同一个操作码，但是使用 Instruction Prefixes 区分指令中的操作数是 16 位的还是 32 位的。比如在 32 位环境下使用了 16 位的操作数，那么就需要在指令前使用 0x66 进行标识。

以下面的汇编代码为例，汇编文件 a.s 中的第一条汇编代码使用了 16 位的寄存器，第二条汇编代码在 32 位寄存器间传递数据。

```
a.s :
mov %ax, %bx
mov %eax, %ebx
```

将 a.s 编译为目标文件，并查看对应的机器指令：

```
root@baisheng:~/demo# gcc -c a.s
root@baisheng:~/demo# objdump -d a.o

a.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:   66 89 c3          mov    %ax,%bx
 3:   89 c3             mov    %eax,%ebx
```

我们观察第一条指令和第二条指令的区别，因为笔者使用的是 32 位的计算环境，所以第一条指令多了前缀 0x66。也就是说，在使用 32 位操作数的环境下，对于使用了 16 位操作数的机器指令，指令前面需要加上前缀 0x66。

Intel 规定了四组指令前缀：Lock and repeat prefixes、Segment override prefixes 和 Branch

hints、Operand-size override prefix，以及Address-size override prefix。前面我们讨论的是 Operand-size override prefix，其他几个不在这里讨论了，有需要的读者可以参考 Intel 手册。

在基本理解了从汇编指令翻译为机器指令的原理后，下面我们就结合 foo2_func 中的两条汇编指令具体探讨一下将汇编指令翻译为机器指令的过程。

```
movl    foo2, %eax
movl    %eax, -4(%ebp)
```

这两条指令使用的都是 mov 指令，IA-32 架构的 mov 指令说明如表 2-3 所示，限于篇幅，我们仅列出了部分。表 2-3 中有两列需要特别关注，一列是“Opcode”，这个无需解释了，指令对应的操作码；另外一列是“Op/En”，“Op/En”是“Operand-Encoding”的简写，根据列的名称相信读者已经猜出来了，该列表示操作数的编码方式。

表 2-3 mov 指令参考（部分）

No.	Opcode	Instruction	Op/En	Description
1	88 /r	MOV r/m8,r8	A	Move r8 to r/m8.
2	REX + 88 /r	MOV r/m8,r8	A	Move r8 to r/m8.
3	89 /r	MOV r/m16,r16	A	Move r16 to r/m16.
4	89 /r	MOV r/m32,r32	A	Move r32 to r/m32.
5
6	A1	MOV AX,moffs16	C	Move word at (seg:offset) to AX.
7	A1	MOV EAX,moffs32	C	Move doubleword at (seg:offset) to EAX.
8
9	B8+ rd	MOV r32, imm32	E	Move imm32 to r32.
10

我们看到，对于 MOV 指令，不仅仅只有一个操作码。对于同一类操作，可能使用不同的操作数，操作数可能是寄存器，也可能是内存地址，同时操作数还会有长度之分，比如 8 位、16 位或者 32 位。Intel 采取的策略是为同一类指令设计了多个操作码来细分这些指令。比如下面一段代码：

```
a.c

void main()
{
    char x, a;
    int y, b;

    x = a;
    y = b;
}
```

我们编译并考察其机器指令：

```

gcc -c a.c
objdump -d a.o

a.o:      file format elf32-i386

Disassembly of section .text:

00000000 <main>:
 0:    55                      push   %ebp
 1:    89 e5                   mov    %esp,%ebp
 3:    83 ec 10                sub    $0x10,%esp
 6:    0f b6 45 f6              movzbl -0xa(%ebp),%eax
 a:    88 45 f7                mov    %al,-0x9(%ebp)
 d:    8b 45 f8                mov    -0x8(%ebp),%eax
 10:   89 45 fc                mov    %eax,-0x4(%ebp)
 13:   c9                      leave 
 14:   c3                      ret    

```

根据反汇编的输出可见，两条赋值语句，对应都是汇编中的 MOV 指令。但是，对于语句“`x = a`”，即偏移 a 处，因为操作数是 8 位的，所以对应的机器码是 0x88，也就是表 2-3 中的第 1 行。对于语句“`y = b`”，对应偏移 0x10 处，因为操作数是 32 位的，所以机器码用的是 0x89，即表 2-3 中的第 4 行。

表 2-3 中值得关注的另外一列“Op/En”指明了对应一个指令的操作数的编码方式。每一类指令都有自己的操作数编码方式，对于 MOV 指令，其操作数的编码方式有 6 类，分别用 A~F 来代表，如表 2-4 所示。

表 2-4 MOV 指令的操作数编码说明

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	AL/AX/EAX/RAX	Displacement	NA	NA
D	Displacement	AL/AX/EAX/RAX	NA	NA
E	reg (w)	imm8/16/32/64	NA	NA
F	ModRM:r/m (w)	imm8/16/32/64	NA	NA

根据表 2-4 可见，如果采用 A 类编码方式，第一个操作数使用 ModRM 中的 R/M 指明，第二个操作数使用 ModRM 中的 Reg/Opcode 指明。如果使用 B 类编码方式，恰恰相反，第二个操作数使用 ModRM 中的 R/M 指明，第一个操作数使用 ModRM 中的 Reg/Opcode 指明。

看过了 MOV 指令的基本说明后，我们来讨论如下指令：

```
movl foo2, %eax
```

这里需要特别注意一点，编译器生成的汇编代码使用的是 AT&T 的格式，其操作数的顺序与 Intel 的汇编指令正好相反，所以这条指令中的第一个操作数“`foo2`”是 Intel 语法中的第二个操作数，这条指令中的第二个操作数“`%eax`”是 Intel 语法中的第一个操作数。

根据这条指令的两个操作数，参照表 2-3，匹配表中的第 7 行，即“MOV EAX, moffs32”，根据该行指令的说明，操作码 0xA1 隐含地指出了指令中的第一个操作数是寄存器 EAX，也就是寻址方式中所谓的操作数隐含寻址。

根据表 2-3 的“Op/En”列可见，该指令的操作数的编码方式是 C，参考表 2-4 可见，C 类编码方式并不需要 ModR/M，当然也不需要 SIB 了，而且也没有使用立即数作为操作数，亦不需要特殊的指令前缀进行修饰。而且，第一个操作数寄存器 EAX 是通过操作码隐含指明。所以，该条汇编代码最后转换为如下形式的机器指令：

Opcode + Displacement

第二个操作数“foo2”通过 Displacement 表示。这里，因为还没有链接，foo2 的地址尚未确定，所以暂时填充 0 占位，在链接时将根据实际地址修改。因为是运行在 32 位环境下，所以地址是 32 位的，Displacement 占用 4 字节。综上所述，该指令的机器码翻译为：

a1 00 00 00 00

再来看指令：

```
mov    %eax, -0x4(%ebp)
```

根据这条指令的两个操作数，参照表 2-3 可见，该指令匹配表中的第 4 行，即“MOV r/m32,r32”，该指令的操作码为 0x89。在确定了操作码后，我们再来看操作数的编码方式，根据表 2-3 中该指令的列“Op/En”可见，该指令使用了 A 类操作数编码方式。根据表 2-4 可见，A 类编码中的第一个操作数由 ModR/M 中的 Mod 和 R/M 共同指明，第二个操作数由 ModR/M 中的 Reg/Opcode 指明。

指令的第一个操作数“-0x4(%ebp)”，相当于 [EBP]+disp8，这里 Displacement 为什么用 8 位，而不是 32 位呢？因为对于 -4，用 1 字节表示足够了，使用 4 字节只能徒增二进制文件的尺寸。根据 A 类编码方式的要求，第一个操作数使用的寄存器需要由 ModR/M 中的 Mod 和 R/M 共同指明，参照表 2-1，根据寻址模式可匹配第 10 行，该行中 Mod 为 01，R/M 为 101，且第一个操作数中的偏移 -4 由 Displacement 表示，在机器指令中需要使用数的补码形式，-4 的补码为 fc。

根据 A 类编码方式的要求，第二个操作数由 ModR/M 中的 Reg/Opcode 指明。汇编指令的第二个操作数使用的寄存器是 EAX，对照表 2-2，寄存器 EAX 对应的 Reg/Opcode 值为 000。

综上所述，该汇编指令对应的机器编码格式为：

Opcode + ModR/M + Displacement

其中 Opcode 为 0x89，ModR/M 的二进制值为 01 000 101，用十六进制表示为 0x45，Displacement 为 fc，该汇编代码的机器编码为：

```
89 45 fc
```

至此，我们通过 foo2_func 中的赋值语句讨论了汇编指令到机器指令的翻译过程。相信读者对机器指令（包括汇编指令）已经有了更好的理解。

我们来查看一下目标文件 foo2.o 中这两条汇编指令对应的真正的机器指令：

```
root@baisheng:~/demo# objdump -d foo2.o

foo2.o:      file format elf32-i386

Disassembly of section .text:

00000000 <foo2_func>:
 0:    55                      push   %ebp
 1:    89 e5                   mov    %esp,%ebp
 3:    83 ec 10                sub    $0x10,%esp
 6:    a1 00 00 00 00          mov    0x0,%eax
 b:    89 45 fc               mov    %eax,-0x4(%ebp)
 e:    c9                      leave 
 f:    c3                      ret
```

其中偏移地址 0x6 和 0xb 处，就是我们前面讨论的两条汇编指令。根据输出可见，与我们的讨论结果完全吻合。

objdump 的输出是经过加工的，我们使用工具 hexdump 原汁原味地将目标文件 foo2.o 转存（dump）出来，查看其代码段部分和数据段部分。我们使用了更精确的参数控制 hexdump 的输出，“%04_ax” 表示使用 4 位十六进制显示偏移；“16/1” 表示每行显示 16 字节，逐字节解析；“%02x” 表示以十六进制显示，每个字符占据两位。为了方便，读者使用参数“-C” 即可。总之，要控制 hexdump 逐个字节解析，避免 hexdump 以双字节为单位进行解析，并且避免使用 little-endian 进行显示可能给读者造成的困惑。下面是我们截取的 foo2.o 的“.text” 段和”.data” 段的片段：

```
root@baisheng:~/demo# hexdump -e '"%04_ax:" 16/1 " %02x" "\n"' foo2.o
0000: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0010: 01 00 03 00 01 00 00 00 00 00 00 00 00 00 00 00
0020: bc 00 00 00 00 00 00 34 00 00 00 00 00 00 28 00
0030: 0a 00 07 00 55 89 e5 83 ec 10 a1 00 00 00 00 89
0040: 45 fc c9 c3 14 00 00 00 00 47 43 43 3a 20 28 55
...
```

其中起始于偏移 0x34 处、占据 0x10 字节的加粗斜体部分正是 objdump 输出的 foo2_func 的机器指令。

注意起始于偏移 0x44 字节处、占据 0x4 字节空间的下划线标识的 4 字节。注意，IA32 架构上，数据是按照 little-endian 顺序存放的，所以这 4 字节表示的数据是 0x14，而不是 0x14000000。十六进制的 0x14 正好是 foo2.c 中的全局变量 foo2 的初始值 20。

这里转存出的“.text” 段和“.data” 段的信息与 foo2.o 中的 Section Header Table 中输出的关于“.text” 段和“.data” 段的信息也完全吻合，即“.text” 段起始于 0x34，占据 0x10 字

节；“.data”段起始于 0x44，占据 0x4 字节。f002.0 的 Section Header Table 中关于这两个段的信息如下：

```
root@baisheng:~/demo# readelf -S foo2.o
There are 12 section headers, starting at offset 0x104:

Section Headers:
[Nr] Name           Type      Addr     Off      Size
...
[ 1] .text          PROGBITS  00000000 000034 000010
...
[ 3] .data          PROGBITS  00000000 000044 000004
...
```

3. 重定位表

在进行汇编时，在一个模块（这里我们将一个.c 文件称为一个模块）内，如果引用了其他模块或库中的变量或者函数，汇编器并不会解析引用的外部符号。因为在汇编时，模块是独立编译的，所以对于引用的外部的符号一无所知。而且退一步说，在汇编时并没有为符号分配运行时地址（行文中有时也称为虚拟地址），所以即使汇编器找到了这些符号，也没有任何意义，这些符号的地址只是临时的，在进行链接时链接器才会为这些符号分配运行时地址。

因此，在目标文件的机器指令中，汇编器基本上是留“空”引用的外部符号的地址。然后，在链接时，在符号地址确定后，链接器再来修订这些位置，这个修订过程被称为重定位。当然除了编译时重定位，还有加载和运行时重定位，本章讨论前者，我们在第 5 章讨论后者。事实上，为了辅助链接器在链接时计算修订值，这些需要修订的位置并不是全部都置为 0，有时这里填充的是一个 Addend，这就是之所以使用引号将空引用起来的原因。下面我们将看到这个 Addend。

但是链接器并不能聪明到可以自动找到目标文件中引用外部符号的地方，所以在目标文件中需要建立一个表格，这个表格中的每一条记录对应的就是一个需要重定位的符号，这个表格通常称为重定位表，汇编器将为可重定位文件中每个包含需要重定位符号的段都建立一个重定位表。ELF 标准规定，重定位表中的表项可以使用如下两种格式：

```
glibc-2.15/elf/elf.h:

typedef struct
{
    Elf32_Addr    r_offset;        /* Address */
    Elf32_Word    r_info;         /* Relocation type and symbol index */
} Elf32_Rel;

typedef struct
{
    Elf32_Addr    r_offset;        /* Address */
    Elf32_Word    r_info;         /* Relocation type and symbol index */
    Elf32_Sword   r_addend;       /* Addend */
} Elf32_Rela;
```

这两种格式唯一的不同是成员 `r_addend`。这个成员一般是个常量，用来辅助计算修订值。如果使用了第一种格式，那么 `r_addend` 将被填充在引用外部符号的地址处，也就是前面所说的留“空”处。具体的体系结构可以选择适合自己的一种格式，或者两种格式都使用，只不过在不同的上下文中使用更合适的格式。IA32 主要使用了前者，但是也在个别的情况下使用了一点后者。

`r_offset` 为需要重定位的符号在目标文件中的偏移。需要注意的是，对于目标文件与可执行文件或者动态库，这个值是不同的。对于目标文件，`r_offset` 是相对于段的，是段内偏移；而对于可执行文件或者动态库，`r_offset` 是虚拟地址。

`r_info` 中包含重定位类型和此处引用的外部符号在符号表中的索引。根据符号在符号表中的索引，链接器就可以从符号表中解析出符号的地址。因为指令中包含多种不同的寻址方式，并且还要针对不同的情况，所以有多种不同的重定位类型。不同的重定位类型，重定位的方法也不同。在 2.1.4 节中讨论“符号重定位”时，我们将讨论编译时使用的典型的重定位类型，包括 `R_386_32` 和 `R_386_PC32`。在第 5 章讨论动态重定位时，我们将讨论加载和运行时使用的典型的重定位类型 `R_386_GLOB_DAT` 和 `R_386 JMP_SLOT` 等。

了解了重定位的基本理论后，下面我们来看一下具体的实例。使用工具 `readelf` 查看目标文件 `hello.o` 的重定位表：

```
root@baisheng:~/demo# readelf -r hello.o

Relocation section '.rel.text' at offset 0x3c4 contains 2 entries:
  Offset      Info      Type            Sym.Value  Sym. Name
0000000b  00000901 R_386_32        00000000    foo2
0000001b  00000a02 R_386_PC32     00000000    foo2_func

Relocation section '.rel.eh_frame' at offset 0x3d4 contains 1 entries:
  Offset      Info      Type            Sym.Value  Sym. Name
00000020  00000202 R_386_PC32     00000000    .text
```

根据输出可见，`hello.o` 中“`.text`”段和“`.eh_frame`”段中都有符号需要重定位，所以建立了两重定位表。

在“`.text`”段的重定位表中，我们看到，目标文件 `hello.o` 引用的外部符号 `foo2` 和 `foo2_func` 分别占据表中的第一条和第二条重定位记录。根据前面目标文件 `hello.o` 的反汇编结果，`foo2` 在偏移 `0xb` 处，`foo2_func` 在偏移 `0x1b` 处，与这里的输出完全一致。

看过重定位表后，我们再来看看汇编器在目标文件 `hello.o` 中引用符号 `foo2` 和 `foo2_func` 处填充的 `Addend` 是什么。我们使用工具 `objdump` 查看目标文件 `hello.o`：

```
root@baisheng:~/demo# objdump -d hello.o

hello.o:      file format elf32-i386

Disassembly of section .text:
```

```
00000000 <main>:
 0:    55                      push   %ebp
 1:    89 e5                   mov    %esp,%ebp
 3:    83 e4 f0                and    $0xffffffff, %esp
 6:    83 ec 10                sub    $0x10,%esp
 9:    c7 05 00 00 00 00 05    movl   $0x5,0x0
10:   00 00 00
13:   c7 04 24 32 00 00 00    movl   $0x32,(%esp)
1a:   e8 fc ff ff ff         call   1b <main+0x1b>
1f:   c9                      leave
20:   c3                      ret
```

根据 objdump 的输出可见：

- 在偏移 0xb 处，对应的就是变量 foo2 的地址，汇编器填充的 Addend 是 0。
- 在偏移 0x1b 处，对应的是函数 foo2_func 的地址，汇编器填充的 Addend 是 “fcfffffc”，因为 IA32 使用的是 little-endian 字节序，补码 “fffffc” 对应的原码是 4。

在引用符号 foo2 的位置，填充 0 是比较容易理解的，链接器只需要找到符号 foo2 的运行时地址替换这里的 0 就好了。但是在引用符号 foo2_func 的位置，为什么使用 -4 呢，这究竟是一个什么魔数？我们在 2.1.4 节中讨论“符号重定位”时，再讨论这个 -4 的由来。

4. 符号表

既然在链接时，需要重定位目标文件中引用的外部符号，显然，链接器需要知道这些符号的定义在哪里，为此汇编器在每个目标文件中创建了一个符号表，符号表中记录了这个模块定义的可以提供给其他模块引用的全局符号。可以使用工具 readelf 查看文件中的符号表，如目标文件 foo2.o 的符号表如下：

```
root@baisheng:~/demo# readelf -s foo2.o
Symbol table '.symtab' contains 10 entries:
Num: Value  Size Type Bind Vis Ndx Name
 0: 00000000    0 NOTYPE LOCAL DEFAULT UND
 1: 00000000    0 FILE   LOCAL DEFAULT ABS foo2.c
 2: 00000000    0 SECTION LOCAL DEFAULT      1
 3: 00000000    0 SECTION LOCAL DEFAULT      3
 4: 00000000    0 SECTION LOCAL DEFAULT      4
 5: 00000000    0 SECTION LOCAL DEFAULT      6
 6: 00000000    0 SECTION LOCAL DEFAULT      7
 7: 00000000    0 SECTION LOCAL DEFAULT      5
 8: 00000000     4 OBJECT  GLOBAL DEFAULT    3 foo2
 9: 00000000    16 FUNC   GLOBAL DEFAULT     1 foo2_func
```

根据输出可见，foo2.o 符号表包含 10 个符号。Value 列表示的是符号的地址。前面我们提到，链接时链接器才会为符号分配地址，所以我们看到的符号的地址全部是 0。Size 列表示符号对应的实体占据的内存大小，如变量 foo2 占据 4 字节，函数 foo2_func 占据 16 字节。Type 列表示符号的类型，如 foo2 类型为 OBJECT，表示变量；foo2_func 类型为 FUNC，表示函数。Bind 列表示符号绑定的相关信息，LOCAL 表示模块内部符号，对外部不可见；

GLOBAL 表示全局符号，foo2 和 foo2_func 都属于全局变量。Ndx 列表示该符号在哪个段，如 foo2 在第 3 个段，即 “.data” 段，foo2_func 在第 1 个段，即 “.text” 段。Name 列表示符号的名称。

除了模块定义的符号外，符号表中也包括了模块引用的外部符号，如模块 hello 的符号表如下：

```
root@baisheng:~/demo# readelf -s hello.o

Symbol table '.symtab' contains 11 entries:
Num: Value    Size Type Bind Vis Ndx Name
 0: 00000000    0 NOTYPE LOCAL DEFAULT UND
 1: 00000000    0 FILE   LOCAL DEFAULT ABS hello.c
 2: 00000000    0 SECTION LOCAL DEFAULT 1
 3: 00000000    0 SECTION LOCAL DEFAULT 3
 4: 00000000    0 SECTION LOCAL DEFAULT 4
 5: 00000000    0 SECTION LOCAL DEFAULT 6
 6: 00000000    0 SECTION LOCAL DEFAULT 7
 7: 00000000    0 SECTION LOCAL DEFAULT 5
 8: 00000000   33 FUNC   GLOBAL DEFAULT 1 main
 9: 00000000    0 NOTYPE GLOBAL DEFAULT UND foo2
10: 00000000    0 NOTYPE GLOBAL DEFAULT UND foo2_func
```

符号 foo2 和 foo2_func 都在模块 foo2 中定义，对于模块 hello 来说是外部符号，没有在任何一个段中，所以在列 Ndx 中，foo2 和 foo2_func 的值是 UND。UND 是 Undefined 的缩写，表示符号 foo2、foo2_func 是未定义的。

在链接时，对于模块中引用的外部符号，链接器将根据符号表进行符号的重定位。如果我们将符号表删除了，那么链接器在链接时将找不到符号的定义，从而不能进行正确的符号解析。如我们将 foo2.o 中的符号表删除，再次进行链接，则链接器将因找不到符号定义而终止链接，如下所示：

```
root@baisheng:~/demo/tmp# strip foo2.o
root@baisheng:~/demo# gcc -o hello *.o
/usr/bin/ld: error in foo2.o(.eh_frame); no .eh_frame_hdr table will be created.
hello.o: In function 'main':
hello.c:(.text+0xb): undefined reference to 'foo2'
hello.c:(.text+0x1b): undefined reference to 'foo2_func'
collect2: error: ld returned 1 exit status
```

2.1.4 链接

链接是编译过程的最后一个阶段，链接器将一个或者多个目标文件和库，包括动态库和静态库，链接为一个单独的文件（通常为可执行文件、动态库或者静态库）。链接器的工作可以分为两个阶段：

- 第一阶段是将多个文件合并为一个单独的文件。对于可执行文件，还需要为指令及符号分配运行时地址。
- 第二阶段进行符号重定位。

1. 合并目标文件

合并多个目标文件其实就是将多个目标文件的相同类型的段合并到一个段中，如图 2-5 所示。

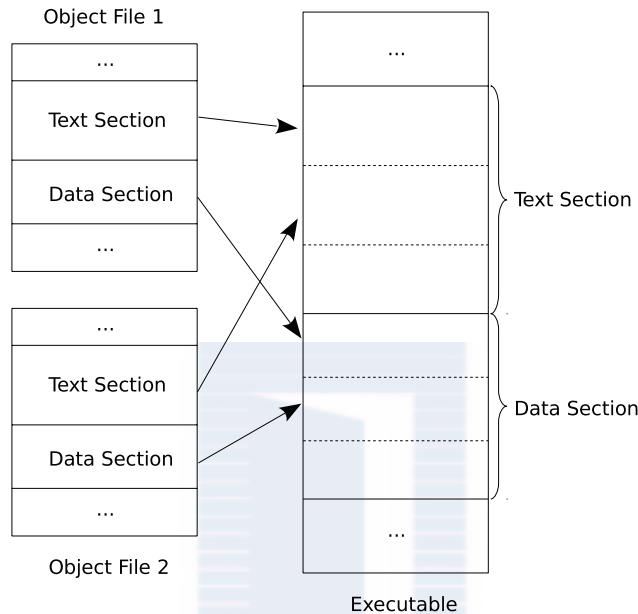


图 2-5 合并目标文件

我们来看一下目标文件和链接后的可执行文件的 “.text” 段。下面分别列出了目标文件 hello.o、foo1.o、foo2.o 以及可执行文件 hello 的段表中的 “.text” 段的相关信息。由于篇幅限制，我们删除了输出的后面几列。

```
root@baisheng:~/demo# readelf -S hello.o
There are 12 section headers, starting at offset 0x118:

Section Headers:
  [Nr] Name          Type      Addr     Off      Size 
  [ 0]             NULL      00000000 000000 000000
  [ 1] .text        PROGBITS 00000000 000034 000026
...
root@baisheng:~/demo# readelf -S foo1.o
There are 12 section headers, starting at offset 0x104:

Section Headers:
  [Nr] Name          Type      Addr     Off      Size 
  [ 0]             NULL      00000000 000000 000000
  [ 1] .text        PROGBITS 00000000 000034 000010
...
root@baisheng:~/demo# readelf -S foo2.o
There are 12 section headers, starting at offset 0x104:
```

```

Section Headers:
[Nr] Name           Type      Addr      Off       Size
[ 0]             NULL      00000000 000000   000000
[ 1] .text         PROGBITS  00000000 000034   000010
...
root@baisheng:~/demo# readelf -S hello
There are 30 section headers, starting at offset 0x1198:

Section Headers:
[Nr] Name           Type      Addr      Off       Size
...
[13] .text         PROGBITS  080482f0 0002f0   0001b8
...

```

根据上面的输出结果可见，对于目标文件，并没有为目标文件中的机器指令及符号分配运行时地址。而对于可执行文件 hello，链接器已经为其机器指令及符号分配了运行时地址，如对于可执行文件 hello 的 “.text” 段，其在进程地址空间中起始地址为 “0x080482f0”，占据了 0x1b8 字节。

按照前面我们提到的目标文件合并理论，理论上三个目标文件 hello.o、foo1.o、foo2.o 的 “.text” 段的尺寸加起来应该与可执行文件 hello 的 “.text” 段的尺寸大小相等。但是，通过 readelf 的输出可见，三个目标文件的 “.text” 段的尺寸加起来是 0x46 (0x26 + 0x10 + 0x10) 字节，远小于可执行文件 hello 的 “.text” 段的大小 0x1b8。如果读者在编译时向 gcc 传递了参数 -v，仔细观察 gcc 的输出可以发现，实际上在链接时链接器自作主张地链接了一些特别的文件，包括 crt1.o、crti.o、crtn.o、crtbegin.o 及 crtend.o 等，其实是我们前面提到的启动文件。所以多出来的尺寸都是合并这些文件的 “.text” 导致的。

下面我们手动调用 ld，不链接这些启动文件，再来对比一下 “.text” 段的尺寸。在默认情况下，链接器将使用函数 “_start” 作为可执行文件的入口，但是这个函数的实现在启动文件 (crt1.o) 中，因此，在这里我们通过给链接器 ld 传递参数 “-e main”，明确告诉链接器不使用默认的 “_start” 了，否则链接器会找不到符号 “_start”，而直接使用函数 main 作为可执行文件的入口。当然 main 函数中并没有实现启动代码的功能，在这里我们只是为了查看 “.text” 段的尺寸。具体如下：

```

root@baisheng:~/demo# ld -e main -o hello1 hello.o foo1.o foo2.o
root@baisheng:~/demo# readelf -S hello1
There are 8 section headers, starting at offset 0x1bc:

Section Headers:
[Nr] Name           Type      Addr      Off       Size
[ 0]             NULL      00000000 000000   000000
[ 1] .text         PROGBITS  08048094 000094   000048
...

```

我们看到，如果不链接那些特殊的文件，按照上面的链接方法，可执行文件的 “.text” 段的大小是 0x48 字节，依然不是 0x46 字节，为什么还是差了 2 字节？我们尝试更换一下链

接时目标文件的次序：

```
root@baisheng:~/demo# ld -e main -o hello2 foo1.o foo2.o hello.o
root@baisheng:~/demo# readelf -S hello2
There are 8 section headers, starting at offset 0x1bc:

Section Headers:
[Nr] Name           Type      Addr     Off      Size
[ 0] NULL          NULL      00000000 000000 000000
[ 1] .text          PROGBITS 08048094 000094 000046
...

```

这次我们看到，最终可执行文件“.text”段的尺寸与目标文件的“.text”段的尺寸和完全相同了。为什么呢？原因是在32位机器上，包括“.text”、“.data”等段有4字节对齐的要求。hello.o的“.text”段是0x26，如果按照4字节对齐，需要填充2字节。而foo1.o和foo2.o的“.text”段本身长度都是4字节对齐的，所以在合并时，如果hello.o在前面，那么其“.text”段需要使用0填充两字节，使其对齐到0x28。所以，最终“.text”的长度就是 $0x28 + 0x10 + 0x10$ ，为0x48字节。而如果hello在最后，那么合并后的“.text”的长度就是 $0x10 + 0x10 + 0x26$ ，即0x46字节。

2. 符号重定位

链接时，在第一阶段完成后，目标文件已经合并完成，并且已经为符号分配了运行时地址，链接器将进行符号重定位。

模块hello.o中有两处需要重定位，一处是偏移0xb处的变量foo2，另外一处是偏移0x1b处的函数foo2_func。汇编器已经将这两处需要重定位的符号记录在了重定位表中。

```
root@baisheng:~/demo# readelf -r hello.o

Relocation section '.rel.text' at offset 0x3c8 contains 2 entries:
  Offset    Info    Type            Sym.Value  Sym. Name
 0000000b  00000901 R_386_32       00000000  foo2
 0000001b  00000a02 R_386_PC32    00000000  foo2_func
...

```

符号foo2的重定位类型是R_386_32，ELF标准规定的计算修订值的公式是：

S + A

其中，S表示符号的运行时地址，A就是汇编器填充在引用外部符号处的Addend。

符号foo2_func的重定位类型是R_386_PC32，ELF标准规定的计算修订值的公式是：

S + A - P

其中S、A的意义与前面完全相同，P为修订处的运行时地址或者偏移。对于目标文件，P为修订处在段内的偏移。对于可执行文件和动态库，P为修订处的运行时地址。

首先我们先来确定S。运行时地址在链接时才分配，因此，变量foo2和函数foo2_func的运行时地址在链接后的可执行文件hello的符号表中：

```
root@baisheng:~/demo# readelf -s hello | grep foo2
 38: 00000000      0 FILE     LOCAL  DEFAULT  ABS foo2.c
 53: 0804a020      4 OBJECT   GLOBAL DEFAULT    24 foo2
 68: 08048414     16 FUNC     GLOBAL DEFAULT    13 foo2_func
```

可见，符号 foo2 的运行时地址为 0x0804a020，符号 foo2_func 的运行时地址是 0x08048414。

接下来，我们再来看看汇编器为这两个符号填充的 Addend 是多少。我们使用工具 objdump 反汇编 hello.o，其中黑体标识的分别是汇编器在引用 foo2 和 foo2_func 的地址处填充的 Addend：

```
root@baisheng:~/demo# objdump -d hello.o

hello.o:      file format elf32-i386

Disassembly of section .text:

00000000 <main>:
 0: 55                      push  %ebp
 1: 89 e5                   mov   %esp,%ebp
 3: 83 e4 f0                and   $0xffffffff0,%esp
 6: 83 ec 10                sub   $0x10,%esp
 9: c7 05 00 00 00 00 05    movl  $0x5,0x0
10: 00 00 00
13: c7 04 24 32 00 00 00    movl  $0x32,(%esp)
1a: e8 fc ff ff ff        call  1b <main+0x1b>
1f: b8 00 00 00 00        mov   $0x0,%eax
24: c9                     leave 
25: c3                     ret
```

根据输出可见，汇编器在引用符号 foo2 处填充的 Addend 是 0，在引用符号 foo2_func 处填充的 Addend 是 -4。

于是，可执行文件 hello 中引用符号 foo2 的位置的修订值为：

$S + A = 0x0804a020 + 0 = 0x0804a020$

我们反汇编可执行文件 hello，来验证一下引用符号 foo2 处的值是否修订为我们计算的这个值：

```
root@baisheng:~/demo# objdump -d hello

hello:      file format elf32-i386
...
080483dc <main>:
 80483dc: 55                      push  %ebp
 80483dd: 89 e5                   mov   %esp,%ebp
 80483df: 83 e4 f0                and   $0xffffffff0,%esp
 80483e2: 83 ec 10                sub   $0x10,%esp
 80483e5: c7 05 20 a0 04 08 05    movl  $0x5,0x804a020
 80483ec: 00 00 00
 80483ef: c7 04 24 32 00 00 00    movl  $0x32,(%esp)
 80483f6: e8 19 00 00 00        call  8048414
```

```

          <foo2_func>
80483fb: b8 00 00 00 00      mov    $0x0,%eax
8048400: c9                  leave
8048401: c3                  ret
8048402: 66 90              xchq   %ax,%ax
...

```

注意偏移 0x1b 处，确实已经被链接器修订为 0x0804a020 了。

对于符号 foo2_func 的修订值，还需要变量 P，即引用符号 foo2_func 处的运行时地址。根据可执行文件 hello 的反汇编代码可见，引用符号 foo2_func 的指令的地址是：

```
0x80483f6 + 1 = 0x80483f7
```

所以，可执行文件 hello 中引用符号 foo2_func 的位置的修订值为：

```
S + A - P = 0x8048414 + (-4) - 0x80483f7 = 0x19
```

观察 hello 的反汇编代码，从地址 0x80483f7 开始处的 4 字节，确实也已经被链接器修订为 0x19。

这里提醒一下读者，如果 foo2_func 占据的运行时地址小于 main 函数，那么这里 foo2_func 与 PC 的相对地址将是负数。在机器指令中，使用的是数的补码形式，所以一定要注意，以免造成困惑。

事实上，对于符号 foo2 使用的重定位类型 R_386_32，是绝对地址重定位，链接器只要解析符号 foo2 的运行时地址替换修订处即可。而对于符号 foo2_func，其使用的重定位类型是 R_386_PC32，这是一个 PC 相对地址重定位。而当执行当前指令时，PC 中已经加载了下一条指令的地址，并不是当前指令的地址，这就是在引用符号 foo2_func 处填充“-4”的原因。

我们看到，在链接时，链接器在需要重定位的符号所在的偏移处直接进行了编辑修订，所以人们通常也将链接器形象地称为“link editor”。

3. 链接静态库

如果在链接过程中有静态库，那么链接是如何进行的呢？静态库其实就是多个目标文件的打包，因此，与合并多个目标文件并无本质差别。但是有一点需要特别说明，在链接静态库时，并不是将整个静态库中包含的目标文件全部复制一份到最终的可执行文件中，而是仅仅链接库中使用的目标文件。如图 2-6 所示，在对可执行文件链接时，只使用了静态库中的“Object File 2”，所以链接器仅将“Object File 2”复制了一份到可执行文件中。

我们使用如下命令先将 foo1.c 和 foo2.c 编译为静态库 libfoo.a。然后将静态库 libfoo.a 链接到可执行程序 hello。

```

root@baisheng:~/demo# gcc -c foo1.c foo2.c
root@baisheng:~/demo# ar -r libfoo.a foo1.o foo2.o
root@baisheng:~/demo# gcc -o hello hello.c libfoo.a

```

我们来看一下静态库 libfoo.a 的符号表：

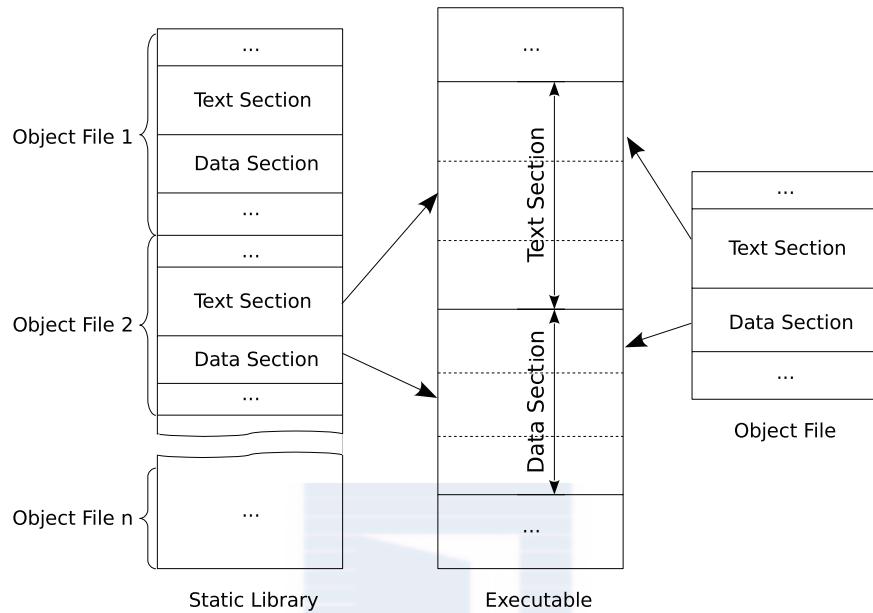


图 2-6 链接静态库

```

root@baisheng:~/demo# readelf -s libfoo.a

File: libfoo.a(foo1.o)

Symbol table '.symtab' contains 10 entries:
Num: Value Size Type Bind Vis Ndx Name
 0: 00000000    0 NOTYPE LOCAL DEFAULT UND
 1: 00000000    0 FILE   LOCAL DEFAULT ABS foo1.c
 2: 00000000    0 SECTION LOCAL DEFAULT 1
 3: 00000000    0 SECTION LOCAL DEFAULT 3
 4: 00000000    0 SECTION LOCAL DEFAULT 4
 5: 00000000    0 SECTION LOCAL DEFAULT 6
 6: 00000000    0 SECTION LOCAL DEFAULT 7
 7: 00000000    0 SECTION LOCAL DEFAULT 5
 8: 00000000    4 OBJECT  GLOBAL DEFAULT 3 foo1
 9: 00000000   19 FUNC   GLOBAL DEFAULT 1 foo1_func

File: libfoo.a(foo2.o)

Symbol table '.symtab' contains 10 entries:
Num: Value Size Type Bind Vis Ndx Name
 0: 00000000    0 NOTYPE LOCAL DEFAULT UND
 1: 00000000    0 FILE   LOCAL DEFAULT ABS foo2.c
 2: 00000000    0 SECTION LOCAL DEFAULT 1
 3: 00000000    0 SECTION LOCAL DEFAULT 3
 4: 00000000    0 SECTION LOCAL DEFAULT 4
 5: 00000000    0 SECTION LOCAL DEFAULT 6
 6: 00000000    0 SECTION LOCAL DEFAULT 7
 7: 00000000    0 SECTION LOCAL DEFAULT 5
 8: 00000000    4 OBJECT  GLOBAL DEFAULT 3 foo2

```

```
9: 00000000      19 FUNC      GLOBAL DEFAULT     1 foo2_func
```

我们看到，与代码中完全吻合，libfoo.a 的符号表中包含 4 个全局符号，分别是变量 foo1 和 foo2、函数 foo1_func 和 foo2_func。如果最终创建的可执行文件 hello 包含了整个 libfoo.a 的副本，那么 hello 的符号表中也应该包含这 4 个全局符号。但是，实际上 hello.c 中仅使用了目标文件 foo2.o 中的函数 foo2_func，所以按照我们前面的理论，hello 中应该仅仅包含 foo2.o 的副本，而不必包含没有使用的 foo1.o。我们查看一下 hello 的符号表：

```
root@baisheng:~/demo# readelf -s hello | grep foo
37: 00000000      0 FILE      LOCAL  DEFAULT  ABS foo2.c
52: 0804a01c      4 OBJECT    GLOBAL  DEFAULT    24 foo2
65: 080483f0     19 FUNC      GLOBAL  DEFAULT    13 foo2_func
```

以上 hello 的符号表仅包含了 foo2 和 foo2_func，显然，可执行文件 hello 中确实没有包含目标文件 foo1.o。至于链接静态库中的目标文件的方法，与我们前面讨论的目标文件的合并完全相同。

4. 链接动态库

我们知道，与静态库不同，动态库不会在可执行文件中有任何副本，那么为什么编译链接时依然需要指定动态库呢？原因包括下面几点：

1) 动态加载器需要知道可执行程序依赖的动态库，这样在加载可执行程序时才能加载其依赖的动态库。所以，在链接时，链接器将根据可执行程序引用的动态库中的符号的情况在 dynamic 段中记录可执行程序依赖的动态库。我们使用如下命令将 foo1.c 和 foo2.c 编译为动态库，并将 hello 链接到动态库 libfoo.so。

```
root@baisheng:~/demo# gcc -shared -fPIC foo1.c foo2.c -o libfoo.so
root@baisheng:~/demo# gcc hello.c -o hello -L./ -lfoo
```

我们来查看 hello 中的 dynamic 段：

```
root@baisheng:~/demo# readelf -d hello | grep Shared
0x00000001 (NEEDED)                         Shared library: [libfoo.so]
0x00000001 (NEEDED)                         Shared library: [libc.so.6]
```

显然，在 dynamic 段中，记录了 hello 依赖的动态链接库 libfoo.so。

2) 链接器需要在重定位表中创建重定位记录（Relocation Record），这样当动态链接器加载 hello 时，将依据重定位记录重定位 hello 引用的这些外部符号。重定位记录存储在 ELF 文件的重定位段（Relocation）中，ELF 文件中可能有多个段包含需要重定位的符号，所以可能会包含多个重定位段。以 hello 的重定位段为例：

```
root@baisheng:~/demo# readelf -r hello

Relocation section '.rel.dyn' at offset 0x3d4 contains 2 entries:
  Offset      Info      Type            Sym.Value  Sym. Name
 08049ffc  00000206  R_386_GLOB_DAT    00000000  __gmon_start__
```

```

0804a020 00000905 R_386_COPY          0804a020   foo2

Relocation section '.rel.plt' at offset 0x3e4 contains 3 entries:
  Offset      Info      Type            Sym. Value  Sym. Name
0804a00c 00000207 R_386_JUMP_SLOT    00000000  __gmon_start__
0804a010 00000307 R_386_JUMP_SLOT    00000000  __libc_start_main
0804a014 00000507 R_386_JUMP_SLOT    00000000  foo2_func

```

根据输出可见，可执行文件 hello 包含两个重定位段，“.rel.dyn” 段中记录的是加载时需要重定位的变量，“.rel.plt” 段中记录的是需要重定位的函数。

因此，虽然编译时不需要链接共享库，但是可执行文件中需要记录其依赖的共享库以及加载 / 运行时需要重定位的条目，在加载程序时，动态加载器需要这些信息来完成加载时重定位。

最后我们再来关注一下在 hello 中的全局符号 foo2 和 foo2_func。

```

root@baisheng:~/demo# nm hello | grep foo
0804a020 B foo2
U foo2_func

```

在符号表中，我们看到，foo2_func 是 Undefined 的，这没错，因为其确实不在 hello 中定义。但是注意变量 foo2，理论上它也应该是 Undefined 的，但是我们看到其在 hello 中是有定义的，而且其还在 BSS 段中。换句话说，虽然我们在 hello 中没有定义一个未初始化的全局变量，但是链接器却偷偷在 hello 中定义了一个未初始化的变量 foo2。那么，这个 foo2 与 libfoo.so 中的全局变量 foo2 是什么关系呢？为什么编译器要这样做？这也是和重定位有关的，事实上，这种重定位方式称为“Copy relocation”，后面我们在讨论用户进程的加载时将会进一步介绍。

2.2 构建工具链

虽然构建的目标系统是运行在 IA32 体系架构上的，但是我们不能使用宿主系统的工具链，否则可能会导致目标系统依赖宿主系统。在编译程序时，如果使用了宿主系统的链接器，那么链接器将在宿主系统的文件系统中寻找依赖的动态库，这势必会导致目标系统中的程序链接宿主系统的某些库，从而导致目标系统依赖宿主系统。其直观表现就是程序在编译时可能会顺利通过，但是当在目标系统中运行时，却可能出现未定义符号的错误。

除了上述的依赖问题外，目标系统使用的工具链的各个组件的版本，通常不同于宿主系统，因此，这也要求为目标系统构建一套新的工具链。

但是工具链在软件开发中占据极其重要的位置，包括编译、汇编、链接等多个组件在内的任一组件的问题都可能导致程序执行时出现问题，如执行效率低下，甚至带来安全问题。因此，在实际应用中，很多时候我们都是直接使用已经构建好的工具链，这类工具链一般都被广泛使用，所以在某种意义上其正确性是被实践检验过的，但是也有缺点，就是没有针对

具体的硬件平台进行优化。因此，有时我们也会借助某些辅助工具，针对我们的特定硬件，进行配置优化，“半自动”地为目标系统构建编译工具链。

在现实中，完全手工构建工具链的机会并不多，很多时候我们可能都是使用别人已经构建好的。但是，工具链中包含的组件可以说是除了操作系统内核之外的最底层的系统软件，无论是对理解操作系统，还是对开发程序来说，都有着重要的意义。即使永远不需自己手工编译工具链，但是了解工具链的构建过程，也可以帮助更高效灵活地运用已有的工具链，可以在多个现成的工具链中进行更好的选择，也有助于进行“半自动”地构建工具链。

2.2.1 GNU 工具链组成

编译过程分为 4 个阶段，分别是：编译预处理、编译、汇编以及链接。每个阶段都涉及了若干工具，GNU 将这些工具分别包含在 3 个软件包中：Binutils、GCC、Glibc。

- ❑ Binutils：GNU 将凡是与二进制文件相关的工具，都包括在软件包 Binutils 中。
Binutils 就是 Binary utilities 的简写，其中主要包括生成目标文件的汇编器（as），链接目标文件的链接器（ld）以及若干处理二进制文件的工具，如 objdump、strip 等。但是也不是 Binutils 中的所有的工具都是处理二进制文件的，比如处理文本文件的预编器（cpp）也包含在其中。
- ❑ GCC：GNU 将编译器包含在 GCC 中，包括 C 编译器、C++ 编译器、Fortran 编译器、Ada 编译器等。为简单起见，在本章中我们只构建 C/C++ 编译器。GCC 中还提供了 C++ 的启动文件。
- ❑ Glibc：C 库包含在 Glibc 中。除了 C 库外，动态链接器（dynamic loader/linker）也包含在这个包中。另外这个包中还提供 C 的启动文件。事实上，有很多 C 库的实现，比如适用于 Linux 桌面系统的 Glibc、EGlibc、uClibc；在嵌入式系统上，可以使用 EGlibc 或者 uClibc；对于没有操作系统的系统，也就是所说的 freestanding environment，可以选择 newlib、dietlibc，或者根本就不用 C 库。

除了这三个软件包外，工具链中还需要包括内核头文件。用户空间中的很多操作需要借助内核来完成，但是通常用户程序不必直接和内核打交道，而是通过更易用的 C 库。C 库中的很大一部分函数是对内核服务的封装。在某种意义上，内核头文件可以看作是内核与 C 库之间的协议。因此，构建 C 库之前，需要首先在工具链中安装内核头文件。

2.2.2 构建工具链的过程

针对我们的具体情况，目标系统与宿主系统都是基于 IA32 体系架构的，所以一种方式是利用宿主的编译工具链来构建一套独立于宿主系统的自包含的本地编译工具链；另外一种方式就是构建一套交叉编译工具链。在本章中，我们采用交叉编译的方式构建工具链，主要原因是：

- ❑ Linux 的主要应用的场景之一是嵌入式领域，嵌入式设备中存在多种不同的体系架构，受限于嵌入式设备的性能和内存等，像编译链接这种工作都在工作站或 PC 上进行。因此，使用交叉编译的方法，对读者进行嵌入式开发更有益处。
 - ❑ 再者，采用交叉编译的方式相对更有助于读者理解链接过程及文件系统的组织。所以，虽然我们的宿主系统和目标系统都是基于 IA32 的，但是我们利用交叉编译的方式构建编译工具链。

如果读者没有嵌入式开发的相关经验，也不必担心，交叉编译与本地编译本质上并无区别。交叉编译就是在目标机器与宿主机器体系结构不同时使用的编译方法。无论是本地编译还是交叉编译，工具链的各个组件均是运行在工作站或者 PC 上，只不过对于本地编译，我们编译出的程序运行在本地系统上，或者至少是运行在与宿主系统相同的体系架构的机器上。而对于交叉编译，编译出的程序是运行在目标机器上的。

如图 2-7 所示，如果目标机器与宿主系统相同均为 IA32 架构，那么就使用宿主机器上的本地编译工具链，编译出的二进制代码对应的也是 IA 架构的指令。如果目标机器是其他的体系结构的，比如 ARM，那么就需要使用宿主系统上的交叉编译工具链，编译出的二进制代码对应的也是目标体系架构的指令，如 ARM 指令。

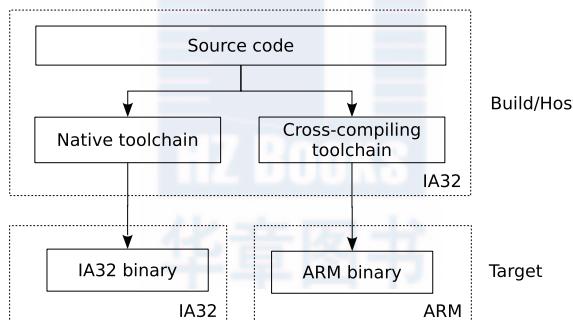


图 2-7 本地编译和交叉编译

GNU 将编译器和 C 库分开放在两个软件包里，好处是比较灵活，在工具链中可以选择不同的 C 库，比如 Glibc、uClibc 等。但是，也带来了编译器和 C 库的循环依赖问题：编译 C 库需要 C 编译器，但是 C 编译器也依赖 C 库。虽然理论上编译器不应该依赖 C 库，C 编译器只负责将源代码翻译为汇编代码，但是事实并非如此：

- ❑ C 编译器需要知道 C 库的某些特性，以此来决定支持哪些特性。所以，为了支持某些特性，C 编译器依赖 C 库的头文件。
 - ❑ C++ 的库和编译器需要 C 库支持，比如异常处理部分和栈回溯部分。
 - ❑ GCC 不仅包含编译器，还包含一些库，这些库通常依赖 C 库。
 - ❑ C 编译器本身也会使用 C 库的一些函数。

但是，幸运的是，C99 标准定义了两种运行环境，一种是“hosted environment”，针对

的是具有操作系统的环境，程序一般是运行在操作系统之上的，因此这个操作系统不仅是内核，还包括外围的 C 库，对于程序来说，就是一个“hosted environment”。另外一种是“freestanding environment”，就是程序不需要额外环境的支持，直接运行在裸机（bare metal）上，比如 Linux 内核，以及一些运行在没有操作系统的裸板上的程序，不再依赖操作系统内核和 C 库，所有的功能都在单个程序的内部实现。

针对这两种运行环境，C99 标准分别定义了两种实现：一种称为“hosted implementation”，支持完整的 C 标准，包括语言标准以及库标准；另外一种是“freestanding implementation”，这种实现方式支持完整的语言标准，但是只要求支持部分库标准。C99 标准要求“hosted implementation”支持“freestanding implementation”，通常是通过向编译器传递参数来控制编译器采用哪种方式进行编译。

通常“hosted implementation”的实现包含编译器（比如 GCC）和 C 库（比如 Glibc）。而“freestanding implementation”的实现通常只包含编译器，如 GCC，最多再加上一个简单的库，比如典型的 newlib。但是如果缺少 newlib 的支持，GCC 自己也可以自给自足。

“freestanding implementation”的实现，恰恰解决了我们提到的 GCC 和 Glibc 的循环依赖问题。我们可以先编译一个仅支持“freestanding implementation”的 GCC，因为在这种情况下，不需要 C 库的支持。但是“freestanding implementation”的 GCC 却可以编译 Glibc，因为 Glibc 也是一个自包含的，完全自给自足。事实上，Glibc 中也有小部分地方使用了 GCC 的代码，但是这不会带来依赖的麻烦，因为 GCC 一定是在 Glibc 之前编译的。

在编译目标系统的 C 库，甚至是编译 GCC 中包含的目标系统上的库时，都需要链接器，因此，Binutils 是编译器和 C 库共同依赖的。索性 Binutils 几乎没有任何依赖，只需要利用宿主系统的工具链构建一套交叉 Binutils 即可。

另外值得一提的是内核头文件。在 Linux 系统上，在编译 C 库前需要安装目标系统的内核头文件，从某种意义上讲，内核头文件就是 C 库和内核之间的一个协议（Protocol）。而且，C 库会根据内核头文件检查内核提供了哪些特性，以及需要在 C 库层面模拟哪些内核没有提供的服务。

综上所述，我们可以按照如下步骤构建工具链：

- 1) 构建交叉 Binutils，包括汇编器 as、链接器 ld 等。
- 2) 构建临时的交叉编译器（仅支持 freestanding）。
- 3) 安装目标系统的内核头文件。
- 4) 构建目标系统的 C 库。
- 5) 构建完整的交叉编译器（支持 hosted 和 freestanding）。

最后提醒读者注意一点，上面的目标平台也是 IA32 的，并且使用的 C 库是 Glibc。如果是其他平台的，或者是用了不同的 C 库，编译过程可能会略有差异，比如为了使最终编译 C 库的编译器具有更多的特性，有的编译过程将使用 freestanding 编译器编译一个简化版的 hosted 编译器，然后用这个 hosted 的 GCC 再编译 Glibc 等。但是，无论如何，交叉编译的

关键还是构建 freestanding 的编译器。freestanding 的编译器解决了鸡和蛋的问题（即 GCC 和 Glibc 的循环依赖），一旦解决了鸡和蛋的问题后，其他的问题就都迎刃而解。

2.2.3 准备工作

1. 新建普通用户 vita

为了避免误操作给宿主系统带来灾难性的后果，在编译过程中我们使用普通用户，避免不小心使用新编译的某些库覆盖宿主系统的库。我们新建一个普通用户 vita：

```
root@baisheng:~# groupadd vita
root@baisheng:~# useradd -m -s /bin/bash -g vita vita
```

我们还要新建一个组 vita，用户 vita 属于这个组。参数 “-m” 表示创建 vita 用户的属主目录，默认是 /home/vita；“-s /bin/bash” 表示使用 bash shell；“-g vita” 表示将 vita 加入组 vita。

在某些情况下，我们可能需要使用 vita 执行一些超级用户才有权限执行的命令，因此，我们让 vita 成为 sudoers，在 /etc/sudoers.d 目录下添加一个文件 vita，其内容如下：

```
vita ALL=(ALL) NOPASSWD: ALL
```

2. 建立工作目录

为了便于管理，我们需要建立一个工作目录，这个目录可以建立在任一目录下。笔者使用了一个单独的分区，并且挂载在 /vita 目录下。在该目录下，建立相应的工作目录的方法如下：

```
root@baisheng:/vita# mkdir source build cross-tool \
cross-gcc-tmp sysroot
root@baisheng:/vita# chown -R vita.vita /vita
```

其中，source 目录中存放的是源代码；build 目录用作编译；cross-tools 目录保存交叉编译工具。因为在整个编译过程中，编译器将被编译两次，所以 cross-gcc-tmp 用来保存临时的 freestanding 编译器，避免这个临时的 freestanding 编译器污染最后的工具链。编译好的目标机器上的文件安装在 sysroot 目录下，sysroot 目录相当于目标系统的根文件系统。另外，我们使用 chown 更改这些目录的属主和属组，使 vita 用户有权限使用这些目录及目录下的文件。

3. 定义环境变量

为了简化编译过程中的一些命令，我们需要定义一些环境变量。同时为了避免每次切换到 vita 用户时，都需要手动重新定义，我们将其定义在 /home/vita/.bashrc 中。

```
unset LANG
export HOST=i686-pc-linux-gnu
export BUILD=$HOST
export TARGET=i686-none-linux-gnu
export CROSS_TOOL=/vita/cross-tool
```

```
export CROSS_GCC_TMP=/vita/cross-gcc-tmp
export SYSROOT=/vita/sysroot
PATH=$CROSS_TOOL/bin:$CROSS_GCC_TMP/bin:/sbin:/usr/sbin:$PATH
```

如果使用的是中文环境的操作系统，那么为了避免不必要的麻烦，要在环境中将其设置英文环境，即上面的 unset LANG，因为，在中文环境下，有些工具，比如 readelf，在输出 ELF 文件的信息时，多此一举地将很多英文翻译为了中文，可能给有些脚本处理工具带来一些麻烦。

为了使后面的构建过程可以找到的交叉编译工具，我们将安装交叉编译工具的目录添加到环境变量 PATH 中，包括临时的 GCC 存储的目录。注意一点，临时的 GCC 存储的目录一定要在最终正式的工具链目录的后面，确保安装最终的交叉编译器后，在编译时将优先使用最终的交叉编译器。

Binutils、GCC 以及 Glibc 的配置脚本中均包含三个配置参数：HOST、BUILD 和 TARGET，这三个配置参数的值均是大致形如 ARCH-VENDOR-OS 三元组的组合。在编译前，可以通过配置选项设定这几个参数的值。如果配置时不显示指定这几个参数，编译脚本将自动探测编译所在的机器的相关值。

读者可以通过查看变量 MACHTYPE，或者查看编译时配置过程的结果，确定机器的三元组。以笔者的机器为例，该值为 i686-pc-linux-gnu，表示机器的 CPU 型号为 i686，vendor 为 none，操作系统为 linux-gnu。

```
root@baisheng:~# echo $MACHTYPE
i686-pc-linux-gnu
```

如果 HOST 的值和 TARGET 的值相同，那么编译脚本就构建本地编译工具。只有当 HOST 和 TARGET 的值不同时，编译脚本才构建交叉编译工具。因此，虽然目标平台也是 x86 架构的，但是为了使用交叉编译的方式，我们在配置时故意显示设置 TARGET 参数为 i686-none-linux-gnu，如此，TARGET 就会与编译脚本自行检测到的 HOST（对于笔者的机器来说，即 i686-pc-linux-gnu）不同，从而构建交叉编译工具。读者可根据自己的具体环境进行调整，总之，要使 TARGET 与 HOST 不同。为了方便在编译时设置配置参数，因此我们定义了环境变量 BUILD、HOST 和 TARGET。

4. 切换到 vita 用户

准备工作完成后，我们使用如下命令切换到 vita 用户：

```
root@baisheng:~# su - vita
```

注意，这里我们切换到 vita 用户使用的是“su -”而不是“su”。后者只是切换了身份，shell 的环境仍然是原用户的 shell 环境，而前者将 shell 环境也切换到了 vita。

2.2.4 构建二进制工具

Binutils 包含各种用来操作二进制目标文件的工具，其中包括 GNU 汇编器 as 和链接器 ld，处理静态库的工具 ar 和 ranlib，系统程序员常用的 objdump、readelf、nm、strings、stip 等。

Binutils 推荐使用单独的目录进行编译：

```
vita@baisheng:/vita/build$ tar xvf \
  ../source/binutils-2.23.1.tar.bz2
vita@baisheng:/vita/build$ mkdir binutils-build
vita@baisheng:/vita/build$ cd binutils-build
vita@baisheng:/vita/build/binutils-build$ 
  ./binutils-2.23.1/configure \
  --prefix=$CROSS_TOOL --target=$TARGET \
  --with-sysroot=$SYSROOT
```

下面介绍各个配置参数的意义。

- ❑ --prefix=\$CROSS_TOOL：通过参数 --prefix 指定安装脚本将编译好的二进制工具安装到保存交叉编译工具链的 \$CROSS_TOOL 目录下。
- ❑ --target=\$TARGET：因为没有显示指定参数 --host 和 --build，所以编译脚本将自动探测 HOST 和 BUILD 的值。对于笔者的机器来说，探测到的 HOST 和 BUILD 值相同，都为 i686-pc-linux-gnu。在前面设置环境变量时，我们故意将环境变量 TARGET 的值设置 i686-none-linux-gnu，与 HOST 自动探测的值不同，因此，编译脚本据此判断这是在构建交叉编译工具链，继而将指导宿主系统的工具链编译“运行在本机，但是最后编译链接的程序 / 库是运行在 \$TARGET 上”的交叉二进制工具。
- ❑ --with-sysroot=\$SYSROOT：我们通过参数 --with-sysroot 告诉链接器，目标系统的根文件系统放置在 \$SYSROOT 目录下，链接时到 \$SYSROOT 目录下寻找相关的库。

配置完成后，使用如下命令编译并安装：

```
vita@baisheng:/vita/build/binutils-build$ make
vita@baisheng:/vita/build/binutils-build$ make install
```

Binutils 将二进制工具安装在 \$CROSS_TOOL/bin 目录下，这里不浪费篇幅一一列举各个工具的具体功能了，读者可以使用 man 进行查看。

除了安装二进制工具外，Binutils 还安装了链接脚本，安装目录是：

```
$CROSS_TOOL/i686-none-linux-gnu/lib/ldscripts
```

其中 elf_i386.x 用于 IA32 上 ELF 文件的链接，elf_i386.xbn、elf_i386.xc 等分别对应 ld 使用不同的链接参数时使用的链接脚本，如果使用了“-N”参数，那么 ld 使用链接脚本 elf_i386.xbn。

Binutils 在 \$CROSS_TOOL/i686-none-linux-gnu/bin 目录下也安装了一些二进制工具，这些是编译器内部使用的，我们不必关心，其实这个目录下的工具与 \$CROSS_TOOL/bin 目录下的工具完全相同，只是名称不同而已。

2.2.5 编译 freestanding 的交叉编译器

正如我们前面讨论的，因为编译器和 C 库之间循环依赖的问题，我们需要找到一个办法解决这个鸡和蛋的问题。幸运的是，C 编译器提供了一个 freestanding 的实现，即一个不依赖 C 库的编译器。那么如何编译一个 freestanding 的编译器呢？

GCC 提供了一个编译选项 `--with-newlib`。这是一个让人困惑的 C 库参数，因为 newlib 本身就是一套 C 库的实现，所以容易让人误解为工具链中使用的 C 库是 newlib，而不是其他的 C 库。事实上，在构建交叉编译器时，其有着特殊的意义，文件 `configure.ac` 中的注释解释得很清楚。

```
gcc-4.7.2/gcc/configure.ac

# If this is a cross-compiler that does not
# have its own set of headers then define
# inhibit_libc

# If this is using newlib, without having the headers available now,
# then define inhibit_libc in LIBGCC2_CFLAGS.
# This prevents libgcc2 from containing any code which requires libc
# support.
: ${inhibit_libc=false}
if { { test x$host != x$target && test "x$with_sysroot" = x ; } ||
     test x$with_newlib = xyes ; } &&
   { test "x$with_headers" = x || test "x$with_headers" = xno ; } ;
then
    inhibit_libc=true
fi
```

注释中说明，在构建交叉编译器且尚未安装 C 库头文件的情况下，需要定义变量 `inhibit_libc`。一旦定义了该变量，将去掉 libgcc 库中对 C 库的一切依赖，转而使用 GCC 内部的实现。如下面的代码片段：

```
gcc-4.7.2/libgcc/crtstuff.c :

#if defined(OBJECT_FORMAT_ELF) \
&& !defined(OBJECT_FORMAT_FLAT) \
&& defined(HAVE_LD_EH_FRAME_HDR) \
&& !defined(inhibit_libc) && !defined(CRTSTUFF_O) \
&& defined(__FreeBSD__) && __FreeBSD__ >= 7
#include <link.h>
#define USE_PT_GNU_EH_FRAME
#endif
```

我们看到，如果没有定义 `inhibit_libc`，则 libgcc 库中可能会包含 `link.h`，而这恰恰是 glibc 提供的头文件。

换句话说，我们可以通过将变量 `inhibit_libc` 赋值为 `true`，告诉 GCC 编译为 freestanding 实现。但是，遗憾的是，GCC 并没有暴露一个直观的配置选项供配置时设置这个变量，相反需要通过另外相关的变量来控制变量 `inhibit_libc` 的值。

再次回顾文件 `gcc-4.7.2/gcc/configure.ac` 中的关于定义 `inhibit_libc` 的条件语句部分，`if` 中的条件如下：

- 1) 如果是交叉编译且未设置 `--with-sysroot`，或者设置了 `--with-newlib`。
- 2) 没有设置 `--with-headers`。

对于条件 1)，因为我们使用了 `sysroot` 的方式，所以要满足第一个条件，就需要设置 `--with-newlib`。对于条件 2)，因为我们没有指定头文件，所以自然成立。

看了前面的讨论，相信读者就比较清楚 `--with-newlib` 的意义了，使用 `--with-newlib` 并不是强行指定 GCC 使用 `newlib` 实现的 C 库。我们无从考究参数 `--with-newlib` 的出处，但是因为 `newlib` 的初衷就是作为 `freestanding` 环境中的 C 库，或许这个参数的名称来源于此。

下面，我们开始编译用于 `freestanding` 环境的 `gcc` 编译器，首先解开源码包：

```
vita@baisheng:/vita/build$ tar xvf ../source/gcc-4.7.2.tar.bz2
```

GCC 依赖包括浮点计算、复数计算的几个数学库 `GMP`、`MPFR` 和 `MPC`。可以先单独编译这些库，然后通过 GCC 的配置选项如 `--with-mpc`、`--with-mpfr`、`--with-gmp` 告知 GCC 这几个库的位置。也可以将这几个库的源码解压到 GCC 的源码目录下，在编译时，GCC 会自动探测并编译。这里我们采用后者；

```
vita@baisheng:/vita/build/gcc-4.7.2$ tar xvf \
  ../../source/gmp-5.0.5.tar.bz2
vita@baisheng:/vita/build/gcc-4.7.2$ mv gmp-5.0.5/ gmp
vita@baisheng:/vita/build/gcc-4.7.2$ tar xvf \
  ../../source/mpfr-3.1.1.tar.bz2
vita@baisheng:/vita/build/gcc-4.7.2$ mv mpfr-3.1.1/ mpfr
vita@baisheng:/vita/build/gcc-4.7.2$ tar xvf \
  ../../source/mpc-1.0.1.tar.gz
vita@baisheng:/vita/build/gcc-4.7.2$ mv mpc-1.0.1/ mpc
```

GCC 要求在单独的目录编译，因此我们创建编译目录 `gcc-build`，配置如下：

```
vita@baisheng:/vita/build$ mkdir gcc-build
vita@baisheng:/vita/build$ cd gcc-build
vita@baisheng:/vita/build$ ../gcc-4.7.2/configure \
  --prefix=$CROSS_GCC_TMP --target=$TARGET \
  --with-sysroot=$SYSROOT \
  --with-newlib --enable-languages=c \
  --with-mpfr-include=/vita/build/gcc-4.7.2/mpfr/src \
  --with-mpfr-lib=/vita/build/gcc-build/mpfr/src/.libs \
  --disable-shared --disable-threads \
  --disable-decimal-float --disable-libquadmath \
  --disable-libmudflap --disable-libgomp \
  --disable-nls --disable-libssp
```

下面介绍各个配置参数的意义。

- ❑ `--prefix=$CROSS_GCC_TMP`：`freestanding` 的 GCC 与最终的 `hosted` 的 GCC 还是有些差别的，这里的 `freestanding` 的 GCC 只是一个临时的 GCC，并不会用作最终的交叉

编译器。所以，为了避免污染最后的工具链，这里将 freestanding 的 GCC 安装在一个临时的目录 \$CROSS_GCC_TMP 中。

- ❑ --target=\$TARGET：与在 Binutils 中指定参数 --target 同样的道理，告诉编译脚本构建的预处理器、编译器等是运行在本机上的，但是最后编译的程序或库是运行在目标体系结构 \$TARGET 上的，即构建交叉编译器。
- ❑ --with-sysroot=\$SYSROOT：配置参数 --with-sysroot 告诉 GCC 目标系统的根文件系统存放在 \$SYSROOT 目录下，编译时到 \$SYSROOT 目录下查找目标系统的头文件以及库。
- ❑ --enable-languages=c：编译 C 库只需要 C 编译器，所以这个临时的 freestanding 编译器只支持 C 编译器。而且像 C++ 编译器，即使想编译也是有心无力，因为其依赖目标系统的 C 库，所以目前也没有条件进行编译。
- ❑ --disable-shared：除了编译器外，软件包 GCC 中也包含有一个运行时库 libgcc。该库主要包括一些目标处理器不支持的数学运算、异常处理，以及一些小的比较复杂的便利函数。在默认情况下，会既编译 libgcc 的静态库版本，也编译动态库版本。但是动态库与静态库不同，加载器在加载动态库后需要进行一些初始化，比如初始化变量，而这些相关的代码是在 C 库的启动文件中实现的，包括 crt1.o、crti.o 等，因此，编译 libgcc 的动态版本时将会链接启动文件。但是此时目标机器的 C 库尚未编译，链接将发生类似“找不到 crt1.o 文件”的错误。因此，这里通过配置选项 --disable-shared 告诉编译脚本不要编译 libgcc 的动态库，仅编译静态库。
- ❑ --with-mpfr-include 和 --with-mpfr-lib：对于 MPFR 这个库，其目录结构与 GCC 的默认设定有一些差异，因此我们需要明确指定，否则编译时会报找不到 libmpfr 的错误。这就是配置时指定配置选项 --with-mpfr-include 和 --with-mpfr-lib 的原因。

另外我们还通过形如 --disable-xxx 这样的参数禁止了一些库的编译，也关闭了编译器的一些特性，因为目前这个 freestanding 的交叉编译器根本不需要这些特性，我们只需要一个基本的能够将 C 库中的代码翻译为目标机器的指令这样一个基本的编译器即可。而且，最重要的是，某些库和特性中可能会依赖 C 库，因此，临时的 freestanding 编译器不支持不必要的特性，也不编译不必要的库。

编译完成后，使用如下命令进行安装：

```
vita@baisheng:/vita/build/gcc-build$ make
vita@baisheng:/vita/build/gcc-build$ make install
```

在使用 --disable-shared 禁止编译 libgcc 的动态库后，GCC 的编译脚本将不再编译库 libgcc_eh.a。但是后面编译 Glibc 时，Glibc 将链接 libgcc_eh.a，Glibc 的 Thread cancellation 使用了 GCC 中的异常处理部分的实现，这里 eh 就是 exception handling 的缩写。我们可以直接修改 Glibc 中的 Makeconfig 文件，或者通过建一个指向 libgcc.a 的符号链接 libgcc_eh.a 来

解决这个问题。因为 libgcc.a 中包含 libgcc_eh.a 所包含的全部内容。我们采用后者来解决这个问题。

```
vita@baisheng:/vita/cross-gcc-tmp$ ln -s libgcc.a \
lib/gcc/i686-none-linux-gnu/4.7.2/libgcc_eh.a
```

2.2.6 安装内核头文件

应用程序很少直接通过内核提供的接口使用内核提供的服务，而通常都是用 C 库使用内核提供的服务。C 库的主要内容之一是对内核服务的封装。以系统调用 _exit 为例：

```
glibc-2.15/sysdeps/unix/sysv/linux/i386/_exit.S:

...
_exit:
    movl    4(%esp), %ebx
    /* Try the new syscall first. */
#ifndef __NR_exit_group
    movl    $__NR_exit_group, %eax
    ENTER_KERNEL
#endif
    /* Not available. Now the old one. */
    movl    $__NR_exit, %eax
    /* Don't bother using ENTER_KERNEL here. If the exit_group
       syscall is not available AT_SYSINFO isn't either. */
    int $0x80
    ...


```

Glibc 中使用的系统调用号 __NR_exit_group 和 __NR_exit 都是在内核中定义的。因此，在编译目标系统的 C 库之前，我们首先需要安装内核头文件。

首先解压内核源码，并清理内核。

```
vita@baisheng:/vita/build$ tar xvf ../source/linux-3.7.4.tar.xz
vita@baisheng:/vita/build/linux-3.7.4$ make mrproper
```

我们可以通过变量 ARCH 指出目标系统的架构，在默认情况下，make 将自动探测宿主系统的架构，并认为目标系统的架构与宿主系统的架构相同。对于 IA32 来说，其 ARCH 值是 i386。另外，在安装前，还需要对内核头文件进行一些合法化检查。

```
vita@baisheng:/vita/build/linux-3.7.4$ make ARCH=i386 \
headers_check
vita@baisheng:/vita/build/linux-3.7.4$ make ARCH=i386 \
INSTALL_HDR_PATH=$SYSROOT/usr/include
```

完成安装后，我们可以看到的内核定义的系统调用号在文件 unistd_32.h 中。Glibc 就可以包含该头文件，并使用诸如 __NR_exit 等宏定义。

```
/vita/sysroot/usr/include/asm/unistd_32.h:
```

```
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
...
#define __NR_exit_group 252
...
```

2.2.7 编译目标系统的 C 库

作为 Linux 操作系统中最底层的 API，几乎运行于 Linux 操作系统上的任何程序都会依赖于 C 库。Glibc 除了封装 Linux 内核所提供的系统服务外，也提供了 C 标准规定的必要功能的实现，如字符串处理、数学计算等。

在 Ubuntu12.10 中，系统默认安装的 awk 是 mawk，我们需要另外安装 gawk，因为 mawk 与 Glibc 中使用的 awk 脚本在兼容上有一些问题。

```
root@baisheng:~# apt-get install gawk
```

解压源码，并打开修复编译错误的 patch。

```
vita@baisheng:/vita/build$ tar xvf ../source/glibc-2.15.tar.xz
vita@baisheng:/vita/build$ cd glibc-2.15
vita@baisheng:/vita/build/glibc-2.15$ patch -p1 \
    < ../../source/glibc-2.15-cpuid.patch
vita@baisheng:/vita/build/glibc-2.15$ patch -p1 \
    < ../../source/glibc-2.15-s_frexp.patch
```

Glibc 要求在单独的目录编译，我们新建目录 glibc-build 来编译 Glibc。

```
vita@baisheng:/vita/build$ mkdir glibc-build
vita@baisheng:/vita/build$ cd glibc-build
vita@baisheng:/vita/build/glibc-build$ ../glibc-2.15/configure \
    --prefix=/usr --host=$TARGET \
    --enable-kernel=3.7.4 --enable-add-ons \
    --with-headers=$SYSROOT/usr/include \
    libc_cv_forced_unwind=yes libc_cv_c_cleanup=yes \
    libc_cv_ctors_header=yes
```

下面介绍各个配置参数的意义。

- ❑ **--host=\$TARGET**：注意这里与 Binutils 和 GCC 编译时指定的是 target 参数不同，Glibc 指定的是 host 参数，但这里 host 的值是 \$TARGET，也就是说 C 库运行所在的 host 是 \$TARGET。换句话说，就是告诉刚刚编译的交叉编译器、汇编器、链接器等编译一个运行在 \$TARGET 平台的 C 库。
- ❑ **--enable-kernel=3.7.4**：除非是制作发行版，需要一个兼容更早内核的 C 库，否则我们没有必要向后兼容较早版本的内核，因为这样只会降低 C 库的效率，包括增加 C 库的体积，甚至影响运行速度。本书构建的系统使用的内核版本为 3.7.4，因此，C 库只支持 3.7.4 及以后版本的内核就可以了。当然，如果这个 C 库运行在早于 3.7.4 版本

的内核上，将报类似于“FATAL: kernel too old”的致命错误，拒绝运行。

- --enable-add-ons：编译 C 库源码目录下全部的 add-on，如 libidn、nptl。
- --with-headers=\$SYSROOT/usr/include：告诉编译脚本内核头文件所在的目录。
- libc_cv_forced_unwind=yes 和 libc_cv_c_cleanup=yes：Glibc 中的 NPTL 将检测 C 编译器对线程的支持，而 freestanding 的 GCC 是不支持线程的，因此，我们这里欺骗一下 Glibc 中的 NPTL，告诉它编译器是支持线程的，采用的方法是设置这样两个参数。
- libc_cv_ctors_header=yes：临时的 freestanding 的 C 编译器不支持启动代码与构造函数支持，因此，这里我们再次欺骗一下 Glibc，人为地告诉 Glibc 编译器是支持启动代码的，也是支持构造函数的。

配置完成后，进行编译安装。我们通过指定参数 install_root 为 \$SYSROOT，将 C 库安装到 \$SYSROOT，即 /vita/sysroot 目录下。

```
vita@baisheng:/mnt/vita/build/glibc-build$ make
vita@baisheng:/mnt/vita/build/glibc-build$ make \
    install_root=$SYSROOT install
```

下面介绍一下 Glibc 安装的主要文件。

(1) C 库

Glibc 除了将最基本、最常用的函数封装在 libc 中外，又将功能相近的一些函数封装到一些子库里，比如将线程相关函数封装在 libpthread 中，将与加密算法相关的函数封装在 libcrypt 中，等等。

Glibc 除了安装库文件本身外，还建立了符号链接，包括：

- 动态链接时使用的共享库符号链接。其命名格式一般为：libLIBRARY_NAME.so.MAJOR_REVISION_VERSION。
- 开发时使用的共享库的符号链接。其命名格式一般为：libLIBRARY_NAME.so。

比如数学库的共享库及其符号链接如下：

```
vita@baisheng:/vita/sysroot/lib$ ls -l libm*
-rwxr-xr-x 1 vita vita 792815 Jan 23 10:29 libm-2.15.so
lrwxrwxrwx 1 vita vita      12 Jan 23 10:29 libm.so.6 -> libm-2.15.so
-rwxr-xr-x 1 vita vita  42195 Jan 23 10:29 libmemusage.so
lrwxrwxrwx 1 vita vita      17 Jan 29 17:17 libmount.so.1 ->
                           libmount.so.1.1.0
-rwxr-xr-x 1 vita vita 746758 Jan 29 17:17 libmount.so.1.1.0
```

其中，libm-2.15.so 是数学库的共享库本身，libm.so.6 是运行时使用的符号链接，libm.so 是编译链接时使用的符号链接。Glibc 将运行时使用的库安装在 \$SYSROOT/lib 目录下，其中包括共享库文件本身及动态链接器需要的符号链接。将开发时使用的库安装在 \$SYSROOT/usr/lib 目录下，包括开发时需要的符号链接及静态库等。

(2) 动态链接器

Glibc 亦提供了加载共享库的工具——动态加载器。2.15 版的 Glibc 提供的动态加载器为 ld-2.15.so，其符号链接是 ld-linux.so.2，也安装在 \$SYSROOT/lib 目录下。

(3) 头文件

Glibc 为应用程序的开发提供了头文件，安装在 \$SYSROOT/usr/include 目录下。

(4) 工具

Glibc 也提供了一些可执行的便利工具，这类工具一般安装在 sbin、usr/bin、usr/sbin 目录下，比如用来转换文件字符编码的工具 iconv，在 usr/lib/gconv 目录下安装了工具 iconv 使用的进行字符编码转换的各种库（如支持 GB18030 的 GB18030.so），如果不打算在目标系统上转换文件的字符编码，完全不必安装该工具。另外还有比如查看共享库依赖的工具 ldd，创建共享库缓存以提高共享库搜索效率的 ldconfig 程序等。

除此之外，usr 目录下还有支持国际化、时区设置需要的文件等。

(5) 启动文件

Glibc 提供了启动文件，包括 crt1.o、crti.o、crtn.o 等，这类文件在编译链接时将被链接器链接到最后的可执行文件中，Glibc 将其安装在 \$SYSROOT/usr/lib 目录下。

2.2.8 构建完整的交叉编译器

现在目标系统的 C 库已经构建完成，我们有条件编译完整的编译器了。进入 GCC 的编译目录，清除临时编译的文件，重新配置 GCC，与第一阶段的配置并无本质区别，但是把第一阶段禁掉的一些特性打开了。

```
vita@baisheng:/vita/build/gcc-build$ rm -rf *
vita@baisheng:/vita/build/gcc-build$ ../gcc-4.7.2/configure \
--prefix=$CROSS_TOOL --target=$TARGET \
--with-sysroot=$SYSROOT \
--with-mpfr-include=/vita/build/gcc-4.7.2/mpfr/src \
--with-mpfr-lib=/vita/build/gcc-build/mpfr/src/.libs \
--enable-languages=c,c++ --enable-threads=posix
```

注意，这次是编译最终的交叉编译器，所以安装在 \$CROSS_TOOL 目录下，而不是 \$CROSS_GCC_TMP 目录下。虽然 GCC 支持多种编译器，但是我们只需要 C 和 C++ 编译器。另外，我们要求编译器支持 posix 线程。

在配置完成后，使用如下命令编译并安装：

```
vita@baisheng:/vita/build/gcc-build$ make
vita@baisheng:/vita/build/gcc-build$ make install
```

最终的交叉编译器安装的主要文件如下：

(1) 驱动程序

GCC 安装的最主要的是交叉编译器的驱动程序，包括 i686-none-linux-gnu-gcc、i686-

none-linux-gnu-g++ 等。

(2) 目标系统的库和头文件

GCC 中也包含了一些用于目标系统的运行时库及头文件，它们安装在 \$CROSS_TOOL/i686-none-linux-gnu 目录下。在该目录下，子目录 lib 存放包括目标系统的运行时库以及供目标系统编译程序使用的静态库，子目录 include 下包含开发目标系统上的程序需要的 C++ 头文件。

(3) helper program

前面我们提到，gcc 仅仅是一个驱动程序，它将调用具体的程序完成具体的任务，这些程序被 GCC 安装在 libexec 目录下，典型的有编译器 cc1，链接过程调用的 collect2 等。

libexec 与 sbin/bin 目录下存放的可执行文件的一个区别是：sbin/bin 目录下的可执行文件一般是用户使用的；而 libexec 目录下的可执行文件一般是由某个程序或工具使用的，所以一般称为“helper program”。

(4) freestanding 实现文件

前面我们提到，C99 标准定义了两种实现方式：一种称为“hosted implementation”，支持全部 C 标准，包括语言标准以及库标准；另外一种是“freestanding implementation”。在 lib 目录下的头文件即为“freestanding implementation”实现标准要求的头文件。

(5) 启动文件

与 C++ 相关的启动文件在 GCC 中，包括 crtbegin.o、crtend.o 等。

讨论完 C 库和编译器后，我们看到，无论是 C 库，还是 GCC 都各自安装了头文件、运行库，GCC 还安装了一些内部使用的可执行程序。那么在编译程序时，GCC 是怎么找到这些文件的呢？答案就是 GCC 内部定义的两个环境变量 LIBRARY_PATH 和 COMPILER_PATH。GCC 会根据用户的一些配置参数，包括 --target、--with-sysroot 等设置这些环境变量的值。我们可以在编译程序时，使用参数 “-v” 查看这两个变量的值。

```
vita@baisheng:~$ i686-none-linux-gnu-gcc -v hello.c
...
COMPILER_PATH=/vita/cross-tool/libexec/gcc/i686-none-linux-gnu/4.6.1:/vita/
cross-tool/libexec/gcc/i686-none-linux-gnu/4.6.1:/vita/cross-tool/libexec/gcc/
i686-none-linux-gnu:/vita/cross-tool/lib/gcc/i686-none-linux-gnu/4.6.1:/vita/
cross-tool/lib/gcc/i686-none-linux-gnu:/vita/cross-tool/lib/gcc/i686-none-linux-
gnu/4.6.1/../../../../i686-none-linux-gnu/bin/
LIBRARY_PATH=/vita/cross-tool/lib/gcc/i686-none-linux-gnu/4.6.1:/vita/cross-
tool/lib/gcc/i686-none-linux-gnu/4.6.1/../../../../i686-none-linux-gnu/lib:/vita/
sysroot/lib:/vita/sysroot/usr/lib/
...

```

比如库的搜索路径，根据 LIBRARY_PATH 的定义，显然，既包括 GCC 安装的库的路径 /vita/cross-tool/i686-none-linux-gnu/lib，又包括 Glibc 安装的库的路径 /vita/sysroot/usr/lib。

2.2.9 定义工具链相关的环境变量

GNU Make 使用了一些隐式的预定义变量，并且这些变量都有对应的默认值。如 CC 代表编译器，默认值是程序 cc，这也是为什么 Linux 各个发行版中一般都有一个符号连接“cc”指向真正的编译器的原因。再比如 AR 代表汇编器，默认值为 ar。读者可以使用下面的命令输出 make 的数据库，进一步查看 make 数据库中的信息，比如查看交叉编译环境中的编译器。

```
vita@baisheng:~$ make -p | grep CC
...
CC = i686-none-linux-gnu-gcc
CPP = $(CC) -E
...
```

这些隐式的预定义变量可以通过环境变量覆盖，或者在 makefile 中显示重新定义。为了避免在编译每一个软件包时，都需要显示指定使用我们构建的交叉工具链，我们在环境变量中定义编译过程使用的相关变量。我们将相关变量定义在 /home/vita/.bashrc 中，确保在每次切换到 vita 用户时，这些变量定义自动生效。

```
/home/vita/.bashrc

export CC="$TARGET-gcc"
export CXX="$TARGET-g++"
export AR="$TARGET-ar"
export AS="$TARGET-as"
export RANLIB="$TARGET-ranlib"
export LD="$TARGET-ld"
export STRIP="$TARGET-strip"
```

在后面安装编译程序时，一般我们均通过给 make 传递变量 DESTDIR 指定 make 将它们安装到目标系统的根文件系统下，即 \$SYSROOT 目录下。为了避免每次都需要指定 DESTDIR 变量，我们也在 .bashrc 中定义这个变量。

```
/home/vita/.bashrc

export DESTDIR=$SYSROOT
```

为了使设置生效，定义变量后需要退出并重新切换到 vita 用户。

注意，如果需要重新构建交叉编译工具链，在构建前，要注释掉这一节的变量定义，在构建完成工具链后再重新启用这里的变量定义。

2.2.10 封装“交叉” pkg-config

在 GNU 中大部分的软件都使用 Autoconf 配置，Autoconf 通常借助工具 pkg-config 去获取将要编译的程序依赖的共享库的一些信息，比如库的头文件存放在哪个目录下，共享库存放在哪个目录下以及链接哪些共享库等，我们将其称为库的元信息。通常，这些信息都被

保存在一个以软件包的名称命名，并以“.pc”作为扩展名的文件中。而 pkg-config 会到特定的目录下寻找这些 pc 文件，一般而言，其首先搜索环境变量 PKG_CONFIG_PATH 指定的目录，然后搜索默认路径，一般是 /usr/lib/pkgconfig、/usr/share/pkgconfig、/usr/local/lib/pkgconfig 等。显然，使用环境变量 PKG_CONFIG_PATH 不能满足我们的要求。因为在交叉编译环境中，我们是不能允许正在编译的程序链接到宿主系统的库上的，也就是说，我们除了告诉 pkg-config 到目标系统的文件系统中寻找外，还要禁止它搜索默认的宿主系统的路径。而另外一个环境变量 PKG_CONFIG_LIBDIR 可以满足我们这个需求，一旦设置了 PKG_CONFIG_LIBDIR，其将取代 pkg-config 默认的搜索路径。因此，在交叉编译时，这两个变量的设置如下：

```
/home/vita/.bashrc

unset PKG_CONFIG_PATH
export PKG_CONFIG_LIBDIR=$SYSROOT/usr/lib/pkgconfig:\
$SYSROOT/usr/share/pkgconfig
```

注意 如果需要重新构建交叉编译工具链，在构建前，也需要注释掉此处的变量定义，在构建完成工具链后再重新启用这里的变量定义。

除了 pkg-config 寻找 pc 文件的搜索路径需要调整外，从 pc 文件中获取的 cflags 和 libs 也需要追加 sysroot 作为前缀。因此，这里我们包装一下 host 系统的 pkg-config，将为交叉编译定制的 pkg-config 放在 \$SYSROOT/bin 下。

```
/vita/cross-tool/bin/pkg-config :

#!/bin/bash
HOST_PKG_CFG=/usr/bin/pkg-config

if [ ! $SYSROOT ] ; then
    echo "Please make sure you are in cross-compile environment!"
    exit 1
fi

$HOST_PKG_CFG --exists $*
if [ $? -ne 0 ] ; then
    exit 1
fi

if $HOST_PKG_CFG $* | sed -e "s/-I/-I\\vita\\sysroot/g; \
s/-L/-L\\vita\\sysroot/g"
then
    exit 0
else
    exit 1
fi
```

并为 pkg-config 增加执行权限：

```
vita@baisheng:/vita/cross-tool/bin$ chmod a+x pkg-config
```

下面是宿主系统自身的 pkg-config 获得的 libmount 库的 --cflags 和 --libs :

```
vita@baisheng:~$ /usr/bin/pkg-config --cflags --libs mount
-I/usr/include/libmount -I/usr/include/bkfst
-I/usr/include/uuid -lmount
```

下面是经过我们包装的 pkg-config 得到的 libmount 库的 --cflags 和 --libs :

```
vita@baisheng:~$ pkg-config --cflags --libs mount
-I/vita/sysroot/usr/include/libmount
-I/vita/sysroot/usr/include/bkfst
-I/vita/sysroot/usr/include/uuid -lmount
```

显然，经过我们包装的 pkg-config 不再到宿主系统的文件系统下寻找依赖的库，而是到目标系统的根文件系统下去寻找依赖的共享库及头文件等。

2.2.11 关于使用 libtool 链接库的讨论

GNU 中的大部分软件包都使用 libtool 处理库的链接。通常，大部分的软件在包发布时都已经包含了 libtool 所需的脚本工具等。但是如果一旦准备使用 autoconf、automake 重新生成编译脚本，且这些脚本中包含了 libtool 提供的 M4 宏，则需要安装 libtool。可使用如下命令安装 libtool。

```
root@baisheng:~# apt-get install libtool
```

在交叉编译环境中使用 libtool 处理库的链接时，依然还有个不大不小的问题，如同 pkg-config 的麻烦一样，如果使用宿主系统的 libtool，那么编译库时生成的库的 la 文件中，记录库本身安装的位置以及依赖库的安装位置的路径将依然指向宿主系统的根文件系统，比如一个典型的 la 文件：

```
dependency_libs=' /usr/lib/libxcb.la /usr/lib/libXau.la'
libdir='/usr/lib'
```

而实际上，目标系统的根文件系统在 \$SYSROOT 下。显然，如果使用 libtool 链接，将会找错库的安装位置。

我们可以修改宿主系统的 libtool，使其在交叉编译环境下能够创建合适的 la 文件；或者直接修改 la 文件，将类似 “/usr/lib/*” 的路径调整为 “\$SYSROOT/usr/lib/*”；或者如 pkg-config 一样，封装一个 libtool。但是我们采用更简单的方式，使用如下命令将 la 文件删除：

```
find $SYSROOT -name "*.la" -exec rm -f '{}' \;
```

删除库的 la 文件后，链接相应的库时将不再使用 libtool 去寻找库的位置，而是依靠链接器去寻找库的位置。虽然 libtool 不建议这样做，但这样做最简单，且不容易发生错误，因此，后续我们采用这种方法。

2.2.12 启动代码

启动代码是工具链中 C 库和编译器都提供了的重要部分之一，但是由于应用程序员很少接触它们，因此非常容易引起程序员的困惑，所以我们特将其单独列出，使用一点篇幅加以讨论。

不知读者是否留意过这个问题：无论是在 DOS 下、Windows 下，还是在 Linux 操作系统下，程序员使用 C 语言编程时，几乎所有程序的入口函数都是 main，这是因为启动代码的存在。在“hosted environment”下，应用程序运行在操作系统之上，程序启动前和退出前需要进行一些初始化和善后工作，而这些工作与“hosted environment”密切相关，并且是公共的，不属于应用程序范畴的事情，这些应用程序员无需关心。更重要的一点是，有些初始化动作需要在 main 函数运行前完成，比如 C++ 全局对象的构造。有些操作是不能使用 C 语言完成的，必须要使用汇编指令，比如栈的初始化。于是编译器和 C 库将它们抽取出来，放在了公共的代码中。

这些公共代码被称为启动代码，其实不只是程序启动时，也包括在程序退出时执行的一些代码，我们统称它们为启动代码，并将启动代码所在的文件称为启动文件。对于 C 语言来说，Glibc 提供启动文件。显然，对于 C++ 语言来说，因为启动代码是和语言密切相关的，所以其启动代码不在 C 库中，而由 GCC 提供。这些启动文件以“crt”（可以理解为 C RunTime 的缩写）开头、以“.o”结尾。

我们查看可执行程序 hello 的入口函数：

```
root@baisheng:~/demo# readelf -h hello | grep Entry
Entry point address: 0x80482f0
```

根据 ELF 的头可见，可执行文件 hello 的入口地址为 0x80482f0。但该地址对应的函数是 main 吗？

```
root@baisheng:~/demo# readelf -s hello | grep 80482f0
61: 080482f0      0 FUNC    GLOBAL DEFAULT  13 _start
```

结果显然让我们很失望，可执行文件的入口不是我们熟悉的 main 函数，而是一个陌生的_start 函数，而且凭我们的职业直觉，这个函数的定义很像汇编语言的函数名。我们再来看一下可执行文件 hello 的代码段的起始地址：

```
root@baisheng:~/demo# readelf -S hello
There are 30 section headers, starting at offset 0x1198:
Section Headers:
[Nr] Name          Type        Addr     Off      Size
...
[13] .text         PROGBITS   080482f0 0002f0 0001b8
...
```

根据代码段的起始地址可见，hello 的代码段的最开头的函数确实是函数_start，而不是我们熟悉的 main 函数。那么 main 函数在哪里呢？

```
root@baisheng:~/demo# readelf -s hello | grep main
64: 080483fc      38 FUNC    GLOBAL DEFAULT  13 main
```

我们做个减法运算：

```
0x080483fc - 0x080482f0 = 0x10c = 268
```

也就是说，在代码段中，偏移 268 字节处才是 main 函数的代码，代码段的前 268 字节都是启动代码，当然，程序启动时的启动代码不仅限于这 268 字节，因为函数 _start 中可能还会调用 C 库中的一些函数。

如果用户的程序中，没有明确指明使用自己定义的启动代码，那么链接器将自动使用 C 库和 C 编译器中提供的启动代码。链接器将函数 “_start” 作为 ELF 文件的默认入口函数。函数 _start 的相关代码如下：

```
glibc-2.15/sysdeps/i386/elf/start.S

_start:
...
popl %esi          /* Pop the argument count. */
movl %esp, %ecx   /* argv starts just at the current stack top.*/
...
andl $0xffffffff0, %esp
pushl %eax         /* Push garbage because we allocate
                      28 more bytes. */
...
pushl %esp
pushl %edx         /* Push address of the shared library
                      termination function. */
...
/* Push address of our own entry points to .fini and .init. */
pushl $_libc_csu_fini
pushl $_libc_csu_init

pushl %ecx          /* Push second argument: argv. */
pushl %esi          /* Push first argument: argc. */

pushl $BP_SYM (main)

/* Call the user's main function, and exit with its value.
   But let the libc call main. */
call BP_SYM (_libc_start_main)
```

_start 函数先作了一些初始化，接着就是调用 __libc_start_main 压栈参数，包括程序进入 main 函数之前的初始化函数 __libc_csu_init、退出时可能执行的善后函数 __libc_csu_fini 以及 main 函数的参数，最后调用 __libc_start_main。

```
glibc-2.15/csu/libc-start.c:

STATIC int LIBC_START_MAIN (...)
{
```

```

...
if (init)
    (*init) (argc, argv, __environ MAIN_AUXVEC_PARAM);
...
result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
...
}

```

进入函数 `__libc_start_main` 后，将调用函数 `__libc_csu_init` 等初始化函数进行各种初始化操作、准备程序运行环境，最后才进入我们熟知的 `main` 函数。

函数 `_start` 包含在启动文件 `crt1.o` 中。根据启动文件 `crt1.o` 的符号表也可看出这一点。

```

vita@baisheng:/vita$ readelf -s /vita/sysroot/usr/lib/crt1.o
...
18: 00000000      0 FUNC     GLOBAL DEFAULT    2 _start
...

```

通过前面的简要分析，我们直观地感受到了所谓“启动代码”的意义。函数 `_start` 才是第一个从“hosted environment”进入到应用程序时运行的第一个函数，是名副其实的入口函数。从系统的角度看，`main` 函数与普通函数无异，并不是什么真正的入口函数，`main` 只是程序员的入口函数。因此，通过更改启动代码，这个程序员的入口函数也完全可以使用其他的函数名称而不是什么 `main`，比如 MFC 中就不用 `main` 这个名字。

在链接时，`gcc` 使用内置的 `spec` 文件来控制链接的启动文件。编译时，可以通过给 `gcc` 传递参数 `-specs=file` 来覆盖 `gcc` 内置的 `spec` 文件。我们可以传递参数 `-dumpspec` 来查看 `gcc` 内置的 `spec` 文件规定链接时链接哪些启动文件：

```

vita@baisheng:~$ i686-none-linux-gnu-gcc -dumpspec
...
*endfile:
%{Ofast|ffast-math|unsafe-math-optimizations:crtfastmath.o%s}
%{mpc32:crtprec32.o%s}
%{mpc64:crtprec64.o%s}
%{mpc80:crtprec80.o%s}
%{shared|pie:crtendS.o%s;:crtend.o%s}
crtn.o%s
...
*startfile:
%{!shared: %{pg|p|profile:g crt1.o%s; pie:Scrt1.o%s; :crt1.o%s}}
crti.o%s
%{static:crtbeginT.o%s; shared|pie:crtbeginS.o%s; :crtbegin.o%s}
...

```

当然，编译时也可以根据实际情况传递参数如 `-nostartfiles`、`-nostdlib`、`-ffreestanding` 等给链接器，告诉链接器不要链接系统中提供的启动代码，而是使用自己程序中提供的。

最后，让我们以一个小例子，结束本章。回顾上面的函数 `__libc_start_main`，在其调用 `main` 函数前，启动代码中的函数 `__libc_start_main` 将调用 `init` 函数，而 `_start` 传递给 `__libc_start_main` 的 `init` 函数指针指向的是 `__libc_csu_init`：

```
glibc-2.15/csu/elf-init.c:

void __libc_csu_init (int argc, char **argv, char **envp)
{
    ...
#ifndef LIBC_NONSHARED
    /* For static executables, preinit happens right before init. */
    {
        const size_t size = __preinit_array_end -
__preinit_array_start;
        size_t i;
        for (i = 0; i < size; i++)
            (*__preinit_array_start [i]) (argc, argv, envp);
    }
#endif

    _init ();

    const size_t size = __init_array_end - __init_array_start;
    for (size_t i = 0; i < size; i++)
        (*__init_array_start [i]) (argc, argv, envp);
}
```

根据函数可见，`__libc_csu_init` 将先后调用段“`.preinit_array`”、“`.init_array`”中包含的函数指针指向的函数。因此，如果打算在程序执行 `main` 函数前或者在动态库被加载时做点什么，那么我们可以定义一个函数，并告诉链接器将函数指针存储到段“`.preinit_array`”或“`.init_array`”中。示例代码如下：

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

void myinit(int argc, char **argv, char **envp)
{
    printf("%s\n", __FUNCTION__);
}

__attribute__((section(".init_array"))) typeof(myinit) *_myinit =
myinit;

void test()
{
    printf("%s\n", __FUNCTION__);
}

bar.c
#include <stdio.h>

void main()
{
    printf ("Enter main./n");
    test();
}
```

我们通过关键字“`__attribute__((section(".init_array")))`”指定链接器将函数 myinit 的地址放置到段“`.init_array`”中，那么在库 libfoo 被加载时，函数 myinit 会被 `__libc_csu_init` 调用。

使用如下命令编译并运行程序：

```
root@baisheng:~/demo# gcc -fPIC foo.c -o libfoo.so
root@baisheng:~/demo# gcc bar.c -o bar -L./ -lfoo
root@baisheng:~/demo# LD_LIBRARY_PATH= ./bar
myinit
Enter main.
test
```

根据程序 bar 的输出可见，函数 myinit 在进入函数 main 之前就被调用了。也就是说，库 libfoo 在加载时，函数 myinit 就被启动代码调用了。



第 3 章

构建内核

内核的构建系统 kbuild 基于 GNU Make，是一套非常复杂的系统。我们本无意着太多笔墨来分析 kbuild，因为作为开发者可能永远不需要去改动内核映像的构建过程，但是了解这一过程，无论是对学习内核，还是进行内核开发都有诸多帮助。所以在构建内核之前，本章首先讨论了内核的构建过程。

对于编译内核而言，一条 make 命令就足够了。因此，构建内核最困难的地方不是编译，而是编译前的配置。配置内核时，通常我们都能找到一些参考。比如，对于桌面系统，可以参考主流发行版的内核配置。但是，这些发行版为了能够在更多的机器上运行，几乎选择了全部的配置选项，编译了全部的驱动，不仅增加了内核的体积，还降低了内核的运行速度。再比如，对于嵌入式系统，BSP（Board Support Package）中通常也提供内核，但他们通常也仅是个可以工作的内核而已。显然，如果要一个占用空间更小、运行更快的内核，就需要开发人员手动配置内核。而且，也确实存在着在某些情况下，我们找不到任何合适的参考，这时我们只能以手动方式从零开始配置。

但是，面对内核中成千上万的配置选项，开发人员通常不知从何下手。但正所谓万事开头难，一旦迈过了这个坎，读者就不会在内核前望而却步。因此，在本章中，我们摸着石头过河，带领读者以手动的方式配置内核。

在内核启动的最后，内核要从根文件系统加载用户空间的程序从而转入用户空间。因此，在本章的最后，我们准备了一个基本的根文件系统来配合内核的启动。我们也采用手动的方式构建这个根文件系统，通过手动的方式，读者将会更透彻地了解到动辄几个 GB 的根文件系统是如何组织和安排的。

3.1 内核映像的组成

在讨论内核构建前，我们先来简单了解一下内核映像的组成，如图 3-1 所示。

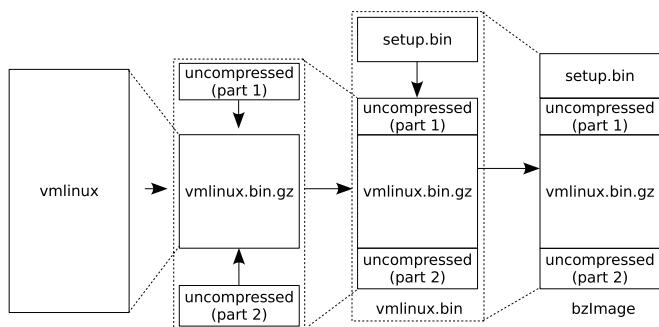


图 3-1 内核映像 bzImage 的组成

如果将内核的映像比作航天器，则 setup.bin 部分就类似于火箭的一级推进子系统。最初，这部分负责将内核加载进内存，并为后面内核保护模式的运行建立基本的环境。但后来加载内核的功能被分离到 Bootloader 中，setup.bin 则退化为辅助 Bootloader 将内核加载到内存。

紧接着，包围在 32 位保护模式部分外的是非解压缩部分。这部分可以看作是火箭的二级推进子系统，负责将压缩的内核解压到合适的位置，并进行内核重定位，在完成这个环节后，其从内核映像脱离。

最后是内核的 32 位保护模式部分 vmlinux。这部分相当于航天器的有效载荷，即类似于最后运行的卫星或者宇宙飞船，只有这部分最后留在轨道内（内存中）运行。内核构建时，将对有效载荷 vmlinux 进行压缩，然后与二级推进系统装配为 vmlinux.bin。

下面我们就来看看内核映像的各个组成部分。

3.1.1 一级推进系统——setup.bin

在进行内核初始化时，需要一些信息，如显示信息、内存信息等。曾经，这些信息由工作在实模式下的 setup.bin 通过 BIOS 获取，保存在内核中的变量 boot_params 中，变量 boot_params 是结构体 boot_params 的一个实例。如 setup.bin 中收集显示信息的代码如下：

```
linux-3.7.4/arch/x86/boot/video.c :

static void store_video_mode(void)
{
    struct biosregs ireg, oreg;
    ...
    initregs(&ireg);
    ireg.ah = 0x0f;
    intcall(0x10, &ireg, &oreg);
    ...
    boot_params.screen_info.orig_video_mode = oreg.al & 0x7f;
    boot_params.screen_info.orig_video_page = oreg.bh;
}
```

store_video_mode 首先调用函数 intcall 获取显示方面的信息，并将其保存在 boot_params

的 screen_info 中。intcall 是调用 BIOS 中断的封装，0x10 是 BIOS 提供的显示服务（Video Service）的中断号，代码如下：

```
linux-3.7.4/arch/x86/boot/bioscall.s:

intcall:
    /* Self-modify the INT instruction. Ugly, but works. */
    cmpb    %al, 3f
    je     1f
    movb    %al, 3f
    jmp    1f    /* Synchronize pipeline */
1:
...
    .byte   0xcd      /* INT opcode */
3: .byte   0
...
```

在代码中我们并没有看到熟悉的调用 BIOS 中断的身影，如“int \$0x10”，但是我们看到了一个特殊的字符——0xcd。正如其后面的注释所言，0xcd 就是 x86 汇编指令 INT 的机器码，如表 3-1 所示。

表 3-1 x86 INT 指令说明（部分）

序号	操作码（Opcode）	指令（Instruction）	操作数编码方式（Op/En）	描述
1	CD ib	INT imm8	B	跟在操作码后面的 8 位立即数指定中断号

根据 x86 的 INT 指令说明，0xcd 后面跟着的 1 字节就是 BIOS 中断号，这就是上面代码中标号为 3 处分配 1 字节的目的。

在函数 intcall 的开头，首先比较寄存器 al 中的值与标号 3 处占用的 1 字节，若相等则直接向前跳转至标号 1 处，否则将寄存器 al 中的值复制到标号 3 处的 1 个字节空间。那么寄存器 al 中保存的是什么呢？

在默认情况下，GCC 使用栈来传递参数。但是我们可以使用关键字“`__attribute__(regparm(n))`”修饰函数，或者通过向 GCC 传递命令行参数“`-mregparm=n`”来指定 GCC 使用寄存器传递参数，其中 n 表示使用寄存器传递参数的个数。在编译 setup.bin 时，kbuild 使用了后者，编译脚本如下所示：

```
linux-3.7.4/arch/x86/boot/Makefile :

KBUILD_CFLAGS := ... -mregparm=3 ...
```

如此，函数的第一个参数通过寄存器 eax/ax 传递，第二个参数通过 ebx/bx 传递，等等，而不是通过栈传递了。因此，上面的寄存器 al 中保存的是函数 intcall 的第一个参数，即 BIOS 中断号。

在完成信息收集后，setup.bin 将 CPU 切换到保护模式，并跳转到内核的保护模式部分执行。如我们前面讨论的，setup.bin 作为一级推进系统，即将结束历史使命，所以内核将

setup.bin 收集的保存在 setup.bin 的数据段的变量 boot_params 复制到 vmlinux 的数据段中。

但是随着新的 BIOS 标准的出现，尤其是 EFI 的出现，为了支持这些新标准，开发者们制定了 32 位启动协议（32-bit boot protocol）。在 32 位启动协议下，由 Bootloader 实现收集这些信息的功能，内核启动时不再需要首先运行实模式部分（即 setup.bin），而是直接跳转到内核的保护模式部分。因此，在 32 位启动协议下，不再需要 setup.bin 收集内核初始化时需要的相关信息。但是这是否意味着可以彻底放弃 setup.bin 呢？

事实上，除了收集信息功能外，setup.bin 被忽略的另一个重要功能就是负责在内核和 Bootloader 之间传递信息。例如，在加载内核时，Bootloader 需要从 setup.bin 中获取内核是否是可重定位的、内核的对齐要求、内核建议的加载地址等。32 位启动协议约定在 setup.bin 中分配一块空间用来承载这些信息，在构建映像时，内核构建系统需要将这些信息写到 setup.bin 的这块空间中。所以，虽然 setup.bin 已经失去了其以往的作用，但还不能完全放弃，其还要作为内核与 Bootloader 之间传递数据的桥梁，而且还要照顾到某些不能使用 32 位启动协议的场合。

3.1.2 二级推进系统——内核非压缩部分

内核的保护模式部分是经过压缩的，因此运行前需要解压缩，但是谁来负责内核映像的解压呢？解铃还须系铃人，既然内核在构建时自己压缩了自己，当然解压缩也要由内核映像自己完成。

内核在压缩的映像外包围了一部分非压缩的代码，Bootloader 在加载内核映像后跳转至外围的这段非压缩部分。这些没有经过解压缩的指令可以直接送给 CPU 执行，由这段 CPU 可执行的指令负责解压内核的压缩部分。

除了解压以外，非压缩部分还负责内核重定位。内核可以配置为可重定位的（relocatable），所谓可重定位即内核可以被 Bootloader 加载到内存任何位置。但是在链接内核时，链接器需要假定一个加载地址，然后以这个假定地址为参考，为各个符号分配运行时地址。显然，如果加载地址和链接时假定的地址不同，那么需要对符号的地址进行重新修订，这就是内核重定位。

内核非压缩部分工作在保护模式下，其占用的内存完成使命后将会被释放。

3.1.3 有效载荷——vmlinux

在编译时，kbuild 分别构建内核各个子目录中的目标文件，然后将它们链接为 vmlinux。为了缩小内核体积，kbuild 删除了 vmlinux 中一些不必要的信息，并将其命名为 vmlinux.bin，最后将 vmlinux.bin 压缩为 vmlinux.bin.gz。在默认情况下，内核使用 gzip 压缩，当然也可以在配置时指定使用 lzma 等压缩格式。gzip 的压缩比相对较小，但是压缩速度相对较快。

那么为什么内核要进行压缩呢？

1) 最初，因为在某些体系架构上，特别是 i386，系统启动时运行于实模式状态，可以寻址空间只能在 1MB 以下，如果内核尺寸过大，将无法正常加载，因此，对内核进行了压缩。在内核加载完毕后，CPU 切换到保护模式，可以寻址更大的地址空间，于是就可以将压缩过的内核展开了。

2) 另外一个原因是，2.4 及更早版本的内核，需要可以容纳在一张软盘上，所以内核也要进行压缩。

以上都是历史原因了，如今有些 Bootloader，如 GRUB，在加载内核期间就已经将 CPU 切换到保护模式了，寻址空间的限制早已不是问题。而且，如今软盘基本已经被其他介质替代，容量已不是问题。

但是内核的压缩还是保留了下来，毕竟还要考虑到某些尺寸受限的情况。而且，现代 CPU 解压的速度要远大于 IO 的速度，在启动时虽然解压要耗费一点时间，但是更小的内核也减少了加载时间。

3.1.4 映像的格式

不知读者留意到没有，无论是 setup.bin、vmlinux.bin，还是 vmlinux.bin.gz，命名中都包含“bin”的字样，这是开发者有意为之，还是机缘巧合？显然，这个 bin 不是开发人员随意杜撰的，而是 binary 的缩写，表示文件格式是裸二进制（raw binary）的。

读者可能有个困惑，在 Linux 操作系统中二进制文件的格式不是使用 ABI（Application Binary Interface）规定的 ELF 吗？

没错，在 Linux 作为操作系统的 hosted environment 环境下，二进制文件使用 ELF 格式，操作系统也提供 ELF 文件的加载器。但是，操作系统本身确是工作在 freestanding environment 环境下。操作系统显然不能强制要求 Bootloader 也提供 ELF 加载器。而且，操作系统映像也没有必要使用 ELF 格式来组织，将代码和数据顺次存放即可，即所谓的裸二进制格式。所以，内核映像都采用裸二进制格式进行组织。

但是，从 Linux 2.6.26 版本开始，内核的压缩部分，即有效载荷部分，采用了 ELF 格式。至于为什么采用 ELF 格式，Patch 的提交者给出了原因：

```
This allows other boot loaders such as the Xen domain builder the
opportunity to extract the ELF file.
```

我们知道，在解压内核映像后，将会跳转到解压映像的开头执行。但是，ELF 文件的开头并不是代码段的开始，而是 ELF 文件头，也就是说，并不是 CPU 可执行的机器指令。显然，当内核映像不是裸二进制格式时，我们需要有一个 ELF 加载器来将 ELF 格式的内核映像转化为裸二进制格式。那么谁来充当这个 ELF 加载器呢？

正所谓“螳螂捕蝉，黄雀在后”。内核的非压缩部分调用函数 decompress 解压内核后，紧接着就调用了函数 parse_elf 来处理 ELF 格式的内核映像，代码如下：

```

linux-3.7.4/arch/x86/boot/compressed/misc.c :

asmlinkage void decompress_kernel(...)
{
    ...
    decompress(input_data, input_len, ...);
    parse_elf(output);
    ...
}

static void parse_elf(void *output)
{
    ...
    for (i = 0; i < ehdr.e_phnum; i++) {
        phdr = &phdrs[i];

        switch (phdr->p_type) {
            case PT_LOAD:
#ifdef CONFIG_RELOCATABLE
                dest = output;
                dest += (phdr->p_paddr - LOAD_PHYSICAL_ADDR);
#else
                dest = (void *) (phdr->p_paddr);
#endif
                memcpy(dest, output + phdr->p_offset, phdr->p_filesz);
                break;
            default: /* Ignore other PT_* */ break;
        }
    }

    free(phdrs);
}

```

在 ELF 文件中，存放代码和数据的段的类型是 PT_LOAD，因此，仅处理这个类型的段即可。在函数 parse_elf 中，对于类型是 PT_LOAD 的段，其按照 Program Header Table 中的信息，将它们移动到链接时指定的物理地址处，即 p_paddr。当然，如果内核是可重定位的，还要考虑内核实际加载地址与编译时指定的加载地址的差值。

事实上，如果 Bootloader 不是所谓的“the Xen domain builder”，我们完全没有必要保留内核的压缩部分为 ELF 格式，并略去启动时进行的“parse_elf”。具体方法如下：

(1) 将压缩部分链接为裸二进制格式

将传递给命令 objcopy 的参数追加“-O binary”，如下面使用黑体标识的部分：

```
linux-3.7.4/arch/x86/boot/compressed/Makefile:
```

```

OBJCOPYFLAGS_vmlinux.bin := -R .comment -S -O binary
$(obj)/vmlinux.bin: vmlinux FORCE
    $(call if_changed,objcopy)

```

(2) 注释掉 parse_elf

既然内核压缩部分已经是裸二进制格式的了，解压后自然不再需要调用函数 parse_elf 了。

```
linux-3.7.4/arch/x86/boot/compressed/misc.c:

asm linkage void decompress_kernel(...)

{
    ...
    decompress(input_data, input_len, ...);
    /* parse_elf(output); */
    ...
}
```

3.2 内核映像的构建过程

3.2.1 kbuild 简介

虽然内核有自己的构建系统 kbuild，但是 kbuild 并不是什么新的东西。我们可以把 kbuild 看作利用 GNU Make 组织的一套复杂的构建系统，虽然 kbuild 也在 Make 基础上作了适当的扩展，但是因为内核的复杂性，所以 kbuild 要比一般项目的 Makefile 的组织要复杂得多。虽然 kbuild 很复杂，但也是有章可循的，下面两点是理解 kbuild 的关键。

1. Makefile 的包含

Makefile 的包含是很多复杂的项目中常用的方法之一。通常的做法是将共同使用的变量或规则定义在一个文件中，在需要使用的 Makefile 中使用关键字“include”来包含这个文件。kbuild 中多处使用了包含的方式，其中关键的两处我们需要特别指出。

(1) 顶层 Makefile 包含平台相关的 Makefile

为了 Linux 能够方便地支持多平台，kbuild 必须方便添加对新平台的支持，同时上层的 Makefile 不需要做大的改动，甚至不需要改动。所以，kbuild 将与平台无关的变量、规则等放到了顶层的 Makefile 中，平台相关的部分定义在各个平台的“顶层” Makefile 中。所谓各个平台的“顶层” Makefile，即 arch/\$(SRCARCH) 目录下的 Makefile。在顶层的 Makefile 中包含平台的“顶层” Makefile，脚本如下所示：

```
linux-3.7.4/Makefile :

include $(srctree)/arch/$(SRCARCH)/Makefile
```

其中变量 SRCARCH 的值就是平台相关部分所在的目录，对于 IA32 架构，SRCARCH 的值为 x86。顶层 Makefile 包含了平台的“顶层” Makefile 后，才组成了真正的 Makefile。这也是为什么我们在顶层目录执行“make bzImage”这样的命令时，可以编译内核映像，却在顶层目录下的 Makefile 中找不到目标 bzImage 的原因，因为其在平台的“顶层” Makefile 中。

(2) Makefile.build 包含各个子目录下的 Makefile

为了方便 Linux 开发者能够编写 Makefile，kbuild 考虑得不可谓不周到，在牺牲自己的同时（kbuild 的实现非常烦琐），确实让 Linux 的开发者们享受了便捷。比如，kbuild 将所有

与编译过程相关的公共的规则和变量都提取到 scripts 目录下的 Makefile.build 中，而具体的子目录下的 Makefile 文件则可以写得非常简单和直接。

kbuild 定义了若干变量，如 obj-y、obj-m 等，用于记录参与编译过程的文件。这些变量就像钩子或者回调函数，各个子目录 Makefile 只需为其赋值，设置参与编译的文件即可，其他事情都由 Makefile.build 处理。甚至最简单的 Makefile 可以简单到只有一行语句：

```
linux-3.7.4/fs/notify/dnotify/Makefile :
obj-$(CONFIG_DNOTIFY)      += dnotify.o
```

在编译时，Makefile.build 会指导 make 将要编译的子目录下的 Makefile 文件包含到 Makefile.include 中动态地组成完整的 Makefile 文件，脚本如下：

```
linux-3.7.4/scripts/Makefile.build :
kbuild-dir := $(if $(filter /%, $(src)), $(src), $(srctree)/$(src))
kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),
    $(kbuild-dir)/Kbuild, $(kbuild-dir)/Makefile)
include $(kbuild-file)
```

理论上，要包含 Makefile 文件，一条 include 命令就够了，为什么这里实现得如此复杂？

一是因为 src 的值是相对于顶层目录的，所以在顶层目录执行 make 没有任何问题。但是如果 make 不是在顶层目录执行的，那么就需要使用绝对路径来定位编译的子目录了。这就是为什么既然有了 src，还要定义变量 kbuild-dir。src 是 kbuild 中定义的一个变量，始终指向需要构建的目录，kbuild-dir 则是加上了绝对路径的 src。make 使用内嵌函数 filter 来判断编译所在的目录的路径是否是以绝对路径表示，即以 “/” 开头，如果不是，则冠以 \$(srctree)。srctree 记录内核顶层目录的绝对路径。以笔者的环境为例，srctree 的值是 /vita/build/linux-3.7.4。在一般情况下，构建都发生在顶层目录下，在子目录下构建是为内核开发人员提供的特性。

二是因为子目录下的“Makefile”文件毕竟不是一个真正意义上的 Makefile，所以 kbuild 的设计者的初衷是希望使用 Kbuild 这个名字。所以，我们看到，在确定 kbuild-file 时，使用 make 的内嵌函数 wildcard 首先尝试匹配子目录下是否存在 Kbuild 这个文件。如果有则优先使用 Kbuild，否则使用 Makefile。但是事实上，在内核目录的绝大部分子目录下，人们还是更习惯使用 Makefile 这个名字。

2. 使用指定 Makefile 的方式进行递归

通常，很多使用 make 进行构建的项目，当存在多级目录时，使用递归方式构建，例如：

```
cd subdir && make
```

也就是说，在子目录下构建时，首先要切换当前工作目录到子目录，然后再启动一个 make 进程解释执行当前目录下的 Makefile。在编译一些规模稍大一点的软件时，我们经常看到 make 不断通过 cd 命令切换目录，原因就在于此。

但是，kbuild 并没有采用切换目录的方式。在 kbuild 中，make 的当前工作目录永远是顶层目录，当编译子目录时，kbuild 通过命令行选项 -f 将子目录的 Makefile 传递给 make，从而达到编译子目录的目的。kbuild 使用的典型方式如下：

```
$ (MAKE) $(build)=<subdir> [target]
```

其中 MAKE 是 make 的内部变量，读者把它理解为 make 即可。变量 build 在 Makefile.build 中定义：

```
linux-3.7.4/scripts/Kbuild.include:

# Shorthand for $(Q)$ (MAKE) -f scripts/Makefile.build obj=
# Usage:
# $(Q)$ (MAKE) $(build)=dir
build := -f $(if $ (KBUILD_SRC),$(srctree)/)scripts/Makefile.build
obj
```

只有当在子目录进行 make 时，变量 KBUILD_SRC 才会被设置为子目录，否则，在顶层目录进行 make 时，该变量值为空，所以 make 的内嵌函数 if 的返回值为 else 部分。但是因为 if 函数的 else 部分为空，所以该函数返回值为空。正如注释中所说，变量 build 相当于下面这段脚本的简写：

```
-f scripts/Makefile.build obj=
```

我们进一步把上面的 make 命令展开：

```
make -f scripts/Makefile.build obj=<subdir> [target]
```

也就说，通过命令行参数 -f，指定 Makefile 为 scripts 目录下的 Makefile.build。而当 make 解释执行 Makefile.build 时，再将子目录中的 Makefile 包含到 Makefile.include 中来，动态地组成子目录的真正的 Makefile。

既然通过指定 Makefile 的方式编译多级目录，而 make 又始终工作在顶层目录下，那么必然要在顶层工作目录中跟踪编译所在的子目录。为此，kbuild 定义了两个变量：src 和 obj。其中，src 始终指向需要构建的目录；obj 指向构建的目标存放的目录。并约定，在引用源码树中业已存在的对象时使用变量 src，引用编译时动态生成的对象使用变量 obj。kbuild 在脚本中小心地维护着这两个变量的值。实际上，因为构建的目标存放的目录与源文件经常在同一个目录下，所以大部分情况下这两个变量均指向同一个目录。

理解了 kbuild 中这两个变量的意义后，读者一定看明白了上述 make 命令中参数 “obj=<subdir>” 的意义，就是设置变量 obj 的值，记录编译所在的子目录。而在 Makefile.build 的一开头，变量 src 的值也被设置为 \$(obj)。

```
linux-3.7.4/scripts/Kbuild.include:
```

```
src := $(obj)
```

```
PHONY := __build
__build:
...
```

下面，我们就结合构建 IA32 架构下的内核映像 bzImage，探讨内核映像的具体构建过程。

3.2.2 构建过程概述

在编译内核时，通常我们只需要执行“make bzImage”，或者 make 后面不接任何目标。在没有接目标时，构建的内核映像也是 bzImage。读者自然会问：我们并没有指定构建 vmlinux、vmlinux.bin 和 setup.bin，最后的 bzImage 是怎么来的呢？

虽然我们没有显示指定这几部分的构建，但是读者想必已经猜出来了，这是 Makefile 的依赖的魔法。下面是构建 bzImage 的规则，我们暂且不讨论它的由来，先把焦点放在 bzImage 的依赖关系上：

```
linux-3.7.4/arch/x86/boot/Makefile:
$(obj)/bzImage: $(obj)/setup.bin $(obj)/vmlinux.bin \
$(obj)/tools/build FORCE
```

根据构建规则可见，bzImage 依赖于 setup.bin 和 vmlinux.bin，所以在构建 bzImage 前，make 将自动先去构建它们，以此类推，vmlinux 的构建也是同样的道理。因此，组成内核映像的各个部分的构建顺序如下：

- 1) 构建有效载荷 vmlinux，并将其压缩为 vmlinux.bin.gz；
- 2) 构建二级推进系统，并将二级推进系统装配到有效载荷上，组成 vmlinux.bin；
- 3) 构建一级推进系统，即构建 setup.bin；
- 4) 将 setup.bin 和 vmlinux.bin 组合为 bzImage。

接下来我们就依次讨论各个部分的构建过程。

3.2.3 vmlinux 的构建过程

所有的体系结构都需要构建 vmlinux，所以 vmlinux 的构建规则在顶层的 Makefile 中。

```
linux-3.7.4/Makefile:
cmd_link-vmlinux = $(CONFIG_SHELL) $< $(LD) $(LDFLAGS) \
$(LDFLAGS_vmlinux)

vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
+$ (call if_changed, link-vmlinux)
```

注意，构建 vmlinux 的命令使用了 make 的内置函数 call。这是一个比较特殊的内置函数，make 使用它来引用用户自己定义的带有参数的函数。if_changed 是 kbuild 定义的一个函

数，这里通过 call 引用这个函数，传递的实参是 link-vmlinux。函数 if_changed 的定义如下：

```
linux-3.7.4/scripts/Kbuild.include:

if_changed = $(if $(strip $(any-prereq) $(arg-check)), \
    @set -e;
    $(echo-cmd) $(cmd_$(1));
    echo 'cmd_@ := $(make-cmd)' > $(dot-target).cmd)
```

在 if_changed 中，any-prereq 检查是否有依赖比目标新，或者依赖还没有创建；arg-check 检查编译目标的命令相对上次是否发生变化。如果两者中只要有一个发生改变，就执行 if 函数的 if 块。注意 if 块中的使用黑体标识的部分，其中“1”代表的就是传给 if_changed 的第一个实参。由此可见，if_changed 核心功能就是当目标的依赖或者编译命令发生变化时，执行表达式“cmd_\$(1)”展开后的值。

这里，传给 if_changed 的第一个实参是 link-vmlinux，因此，cmd_\$(1) 展开后为 cmd_link-vmlinux。注意 cmd_link-vmlinux 中的第二项“\$<”，这是 make 的自动变量，翻译自“Automatic Variable”，意指变量名相同，但是 make 根据具体上下文，将其自动替换为合适的值。这里，make 会将这个自动变量替换为构建 vmlinux 中的规则中的第一个依赖，即 shell 脚本文件 scripts/link-vmlinux.sh，该脚本文件中负责 vmlinux 链接的脚本如下：

```
linux-3.7.4/scripts/link-vmlinux.sh:

vmlinux_link()
{
    local lds="${objtree}/.${KBUILD_LDS}"
    if [ "${SRCARCH}" != "um" ]; then
        ${LD} ${LDFLAGS} ${LDFLAGS_vmlinux} -o ${2} \
            -T ${lds} ${KBUILD_VMLINUX_INIT} \
            --start-group ${KBUILD_VMLINUX_MAIN} --end-group ${1}
    else
        ...
    fi
}
...
vmlinux_link "${kallsyms}" vmlinux
```

根据函数 vmlinux_link 的实现，如果平台不是“um”，那么就调用链接器将变量 KBUILD_VMLINUX_INIT、KBUILD_VMLINUX_MAIN 中记录的目标文件链接为 vmlinux。我们看看这两个变量的定义：

```
linux-3.7.4/Makefile:

# Externally visible symbols (used by link-vmlinux.sh)
export KBUILD_VMLINUX_INIT := $(head-y) $(init-y)
export KBUILD_VMLINUX_MAIN := $(core-y) $(libs-y) $(drivers-y) \
    $(net-y)
```

我们以 core-y 为例来分析变量 KBUILD_VMLINUX_MAIN 的值。

```
linux-3.7.4/Makefile :

core-y      := usr/
...
core-y      += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
...
core-y      := $(patsubst %/, %/built-in.o, $(core-y))
```

patsubst 是 make 的内置函数，功能是在输入的文本中查找与模式匹配的字符串，然后使用特定字符串进行替换。具体到这里，其目的就是在变量 core-y 的值中将字符串 “/” 替换为 “/built-in.o”。经过函数 patsubst 替换后，最后变量 core-y 的值如下：

```
core-y := user/built-in.o kernel/built-in.o mm/built-in.o fs/built-in.o \
          ipc/built-in.o security/built-in.o crypto/built-in.o \
          block/built-in.o
```

除了各个子目录下的 built-in.o，有些子目录（如 lib）下还会编译 lib.a。总之，vmlinux 就是由这些目录下的 built-in.o、lib.a 等链接而成的。

那么这些子目录下面的目标文件 built-in.o 或者 lib.a 是在什么时机构建的呢？我们来回顾一下 vmlinux 的构建规则；

```
linux-3.7.4/Makefile :

vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
```

我们看到，除了依赖 scripts/link-vmlinux.sh，vmlinux 的另外一个依赖是 vmlinux-deps，其构建规则也在顶层 Makefile 中定义：

```
linux-3.7.4/Makefile :

vmlinux-deps    := $(patsubst %/,%, $(filter %/, $(init-y) \
                                         $(init-m) $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
                                         $(net-y) $(net-m) $(libs-y) $(libs-m))) )
...
vmlinux-deps := $(KBUILD_LDS) $(VMLINUX_INIT)
                 $(VMLINUX_MAIN)
...
$(sort $(vmlinux-deps)): $(vmlinux-deps) ;
...
$(vmlinux-deps): prepare scripts
  $(Q) $(MAKE) $(build)=$@
```

我们首先看看变量 vmlinux-deps，显然，其记录的就是我们前面讨论的最终链接为 vmlinux 的内核子目录下的目标文件的名字，如 built-in.o 等。也就说，vmlinux 的构建规则表达得很清楚，要最后链接 vmlinux，首先需要构建这些目标文件。但是注意目标 vmlinux-deps 的构建规则，其“规则体”是空的，也就是说这个构建规则下没有任何命令可执行，但是可以看到这些目标文件依赖于另外一个目标 vmlinux-deps，我们继续跟踪目标 vmlinux-deps 的构建。

我们来关注一下变量 vmlinux-dirs 的值。注意该变量的赋值脚本，其中函数 filter 也是 make 的内置函数，其功能是过滤掉输入文本中不以“/”结尾的字符串。前面我们看到，输入到 filter 的这些变量，比如 core-y，其中所有的子目录都以“/”结尾，因此，这里 filter 的目的是过滤掉这些变量中的非目录。patsubst 这个 make 的内置函数我们刚刚讨论过，显然是将过滤出来的子目录后面的字符“/”去掉。因此，正如其名字所揭示的，变量 vmlinux-dirs 的值是多个目录，所以构建 vmlinux-dirs 的规则也是一个多目标规则，等价于：

```
init: prepare scripts
    $(Q) $(MAKE) $(build)=$@
kernel: prepare scripts
    $(Q) $(MAKE) $(build)=$@
...
...
```

规则中的命令展开后为：

```
make -f script/Makefile.build obj=$@
```

其中“\$@”是 make 的自动变量，表示规则的目标，所以这里会被 make 自动替换为构建的子目录，如 init、kernel 等，即相当于逐个编译这些子目录，使用的 Makefile 是 Makefile.build。如 3.2.1 节讨论的那样，Makefile.build 将包含构建目录中的 Makefile 或 Kbuild，最终形成完整的 Makefile。make 命令中没有显式指定构建目标，因此，将构建 Makefile.build 中默认的目标。Makefile.build 中的默认目标是 __build，脚本如下所示：

```
linux-3.7.4/scripts/Makefile.build:
src := $(obj)
PHONY := __build
__build:
...
__build: $(if $(KBUILD_BUILTIN),$(builtin-target) \
    $(lib-target) $(extra-y)) \
    $(if $(KBUILD_MODULES),$(obj-m) $(modorder-target)) \
    $(subdir-y) $(always)
```

目标 __build 涵盖了内核映像和模块，这里我们只关注内核映像的构建，不关注模块的构建。对于编译内核映像来说，目标 __build 依赖 builtin-target、lib-target、extra-y、subdir-y 和 always。我们先来看 builtin-target 和 lib-target：

```
linux-3.7.4/scripts/Makefile.build:
ifeq ($(strip $(lib-y) $(lib-m) $(lib-n) $(lib-)),)
lib-target := $(obj)/lib.a
endif

ifeq ($(strip $(obj-y) $(obj-m) $(obj-n) $(obj-) $(subdir-m) \
    $(lib-target)),)
builtin-target := $(obj)/built-in.o
endif
```

根据上述脚本片段可见，`builtin-target` 代表的就是子目录下的 `built-in.o`，`lib-target` 代表的就是子目录下的 `lib.a`。对于构建的子目录，如果变量 `obj-y` 等值非空，那么就构建 `built-in.o`。如果变量 `lib-y` 等的值非空，那么就构建 `lib.a`。我们来看看 `built-in.o` 和 `lib.a` 的构建：

```
linux-3.7.4/scripts/Makefile.build:

cmd_link_o_target = $(if $(strip $(obj-y)), \
    $(LD) $(ld_flags) -r -o $@ $(filter $(obj-y), $^) \
    $(cmd_secanalysis), \
    rm -f $@; $(AR) rcs$(KBUILD_ARFLAGS) $@)

$(builtin-target): $(obj-y) FORCE
    $(call if_changed,link_o_target)
...
cmd_link_l_target = rm -f $@; $(AR) rcs$(KBUILD_ARFLAGS) $@ $(lib-y)

$(lib-target): $(lib-y) FORCE
    $(call if_changed,link_l_target)
```

前面已经讨论过函数 `if_changed`，如果理解了这个函数，就很容易理解 `built-in.o` 和 `lib.a` 的构建过程了，对于 `built-in.o`，就是调用链接器将变量 `obj-y` 中记录的各个目标文件链接为 `built-in.o`。对于 `lib-target`，就是调用创建静态库的程序 `AR` 将变量 `lib-y` 中的各个目标文件链接为 `lib.a`。

编译内核时需要一些临时工具，比如我们前面用到的 `mkgiggy`、`build` 等，这些是一定需要编译的，因为构建内核时会用到。因此，`kbuild` 中定义了一个变量 `always`，其中记录的就是必须要编译的构建目标。

另外，可能有多层目录嵌套的情况，因此 `_build` 依赖列表中有这么一项：`subdir-ym`。目标 `subdir-ym` 的规则如下：

```
linux-3.7.4/scripts/Makefile.build:

$(subdir-ym):
    $(Q) $(MAKE) $(build)= $@
```

上面的代码看上去是不是很熟悉？没错，它和前面处理 `vmlinux-dirs` 的规则完全相同，显然，这是在处理目录中还有子目录的情况。

至此，链接 `vmlinux` 的目标文件经构建完成。回顾一下 `vmlinux` 的构建过程，`kbuild` 将依次构建 `Makefile` 中指定的子目录，生成 `built-in.o`、`lib.a` 等目标文件，然后调用链接器将这些目标文件链接为 `vmlinux`，并保存在顶层目录下。

3.2.4 vmlinux.bin 的构建过程

根据图 3-1 可知，`kbuild` 将有效载荷与内核的非压缩部分装配为 `vmlinux.bin`。我们前面已经看到了有效载荷 `vmlinux` 的构建过程，这一节我们讨论二级推进系统的构建，并看看二级推进系统是如何与有效载荷进行装配的。构建 `vmlinux.bin` 的规则在 `arch/x86/boot` 目录下

的 Makefile 中：

```
linux-3.7.4/arch/x86/boot/Makefile :

OBJCOPYFLAGS_vmlinux.bin := -O binary -R .note -R .comment -S
$(obj)/vmlinux.bin: $(obj)/compressed/vmlinux FORCE
    $(call if_changed,objcopy)
```

根据前面对 kbuild 自定义函数 `if_changed` 的讨论可知，这里将执行命令 `cmd_objcopy`。
`cmd_objcopy` 的定义如下：

```
linux-3.7.4/scripts/Makefile.lib :

cmd_objcopy = $(OBJCOPY) $(OBJCOPYFLAGS) $(OBJCOPYFLAGS_$(@F)) \
    $< $@
```

其中 `OBJCOPY` 就是二进制工具 `objcopy`，当然，因为我们使用的是交叉工具链，所以 `objcopy` 是 `i686-none-linux-gnu-objcopy`，前面构建工具链时构建组件 `Binutils` 时已经构建：

```
linux-3.7.4/Makefile :

OBJCOPY      = $(CROSS_COMPILE) objcopy
```

这里使用这个工具的目的是将 ELF 格式的文件转化为裸二进制格式。

`cmd_objcopy` 中的 “\$<”、“\$@”、“\$(@F)” 都是 make 的自动变量，“\$<” 表示规则的依赖列表中的第一个依赖，这里是 `arch/x86/boot/compressed/vmlinux`；“\$@” 表示规则的目标，这里是 `arch/x86/boot/vmlinux.bin`；“\$(@F)” 表示构建目标去除目录后的文件名，这里是 `vmlinux.bin`，因此变量 `OBJCOPYFLAGS_$(@F)` 展开为 `OBJCOPYFLAGS_vmlinux.bin`。替换各个变量后，`cmd_objcopy` 最后展开大致为：

```
cmd_objcopy = i686-none-linux-gnu-objcopy -O binary -R .note \
    -R .comment -S arch/x86/boot/compressed/vmlinux \
    arch/x86/boot/vmlinux.bin
```

上述代码的意义已经显而易见了：`arch/x86/boot` 目录下的 `vmlinux.bin` 是由 `arch/x86/boot/compressed` 目录下的 `vmlinux` 通过工具 `i686-none-linux-gnu-objcopy` 复制而来。

为了指导加载器加载 ELF 文件，ELF 文件中附加了很多信息，如 ELF 文件头、Program Header Table、符号表、重定位表等。但是这些对内核是没有意义的，Bootloader 加载内核时不需要 ELF 文件中附加的这些信息，变量 `OBJCOPYFLAGS_vmlinux.bin` 中的 “-O binary” 指定 `objcopy` 将复制后内核转换为裸二进制格式；选项（如 “.note”、“.comment”）则表明将这些段也删除。读者可能会问，转化为裸二进制格式时还会保留如 “.note”、“.comment” 等段吗？当然了，转化为裸二进制格式只不过把为 ELF 格式附加的东西去除掉了，比如 ELF 的头、Section Header Table、Program Header Table 等，但是并不会删除保存具体内容的段。

显然，构建的焦点转换为 `arch/x86/boot/compressed` 下的 `vmlinux`，其构建规则如下：

```
linux-3.7.4/arch/x86/boot/Makefile :
```

```
$(obj)/compressed/vmlinux: FORCE
$(Q) $(MAKE) $(build)=$(obj)/compressed $@
```

构建命令展开为：

```
make -f scripts/Makefile.build obj=arch/x86/boot/compressed
      arch/x86/boot/compressed/vmlinux
```

Makefile.build 将 arch/x86/boot/compressed 目录下的 Makefile 包含到 Makefile.build 中，生成完整的 Makefile。但是这次，make 没有如同构建各个子目录一样使用默认的构建目标，而是指定了构建目标为 arch/x86/boot/compressed/vmlinux，其构建规则在 arch/x86/boot/compressed 目录下的 Makefile 中定义：

```
linux-3.7.4/arch/x86/boot/compressed/Makefile :

VMLINUX_OBJS = $(obj)/vmlinux.lds $(obj)/head_$(BITS).o \
               $(obj)/misc.o $(obj)/string.o $(obj)/cmdline.o \
               $(obj)/early_serial_console.o $(obj)/piggy.o
...
$(obj)/vmlinux: $(VMLINUX_OBJS) FORCE
$(call if_changed, ld)
```

对于 32 位系统，变量 BITS 为 32。由上述 Makefile 可见，arch/x86/boot/compressed 目录下的 vmlinux 是由该目录下的 head_32.o、misc.o、string.o、cmdline.o、early_serial_console.o 以及 piggy.o 链接而成的。其中 vmlinux.lds 是指导链接过程的脚本。

在一份刚刚解压且没有进行任何编译动作之前的内核源码中，除了 piggy.o，我们可以找到上述依赖列表中任何一个目标文件的源文件，比如 head_32.o 对应源文件 head_32.S，misc.o 对应源文件 misc.c 等。而我们却找不到 piggy.o 对应的源文件，比如 piggy.c 或 piggy.S 亦或其他。但是仔细观察，我们会发现在 arch/x86/boot/compressed 目录下的 Makefile 中有一个创建文件 piggy.S 的规则：

```
linux-3.7.4/arch/x86/boot/compressed/Makefile :

cmd_mkpiggy = $(obj)/mkpiggy $< > $@ || ( rm -f $@ ; false )
$(obj)/piggy.S: $(obj)/vmlinux.bin.$(suffix-y) $(obj)/mkpiggy \
                 FORCE
$(call if_changed, mkpiggy)
```

看到上面的规则，我们恍然大悟，原来 piggy.o 是由 piggy.S 汇编而来，而 piggy.S 是编译内核时动态创建的，这就是我们找不到它的原因。piggy.S 的第一个依赖 vmlinux.bin.\$(suffix-y) 中的 suffix-y 表示内核压缩方式对应的后缀：

```
linux-3.7.4/arch/x86/boot/compressed/Makefile :

suffix-$(CONFIG_KERNEL_GZIP)      := gz
suffix-$(CONFIG_KERNEL_BZIP2)      := bz2
...
```

如果配置内核时指定采用 gzip 压缩方式，则 suffix-y 值为 gz；如果指定 bzip2 压缩方式，则 suffix-y 值为 bz2；等等。在本书中，我们配置的内核使用默认的压缩方式 gzip，因此，suffix-y 的值为 gz。那么 vmlinux.bin.gz 是什么呢？我们看下面的脚本：

```
linux-3.7.4/arch/x86/boot/compressed/Makefile :

vmlinux.bin.all-y := $(obj)/vmlinux.bin
vmlinux.bin.all-$(CONFIG_X86_NEED_RELLOCS) += \
    $(obj)/vmlinux.relocs

$(obj)/vmlinux.bin.gz: $(vmlinux.bin.all-y) FORCE
    $(call if_changed,gzip)
```

看到这里，相信读者已经不需要看 cmd_gzip 的定义了。根据变量 vmlinux.bin.all-y 的值，vmlinux.bin.all-y 中包括 arch/x86/boot/compressed 目录下的 vmlinux.bin。如果内核被配置为可重定位的，那么 vmlinux.bin.all-y 中还包括记录重定位信息的 vmlinux.relocs。也就是说，如果内核被配置可重定位，则 vmlinux.bin.gz 是由 vmlinux.bin 和 vmlinux.relocs 压缩而来的，否则只是 vmlinux.bin 由压缩而来。

那么 arch/x86/boot/compressed 目录下的 vmlinux.bin 又是如何创建的？看下面的脚本：

```
linux-3.7.4/arch/arch/x86/boot/compressed/Makefile :

OBJCOPYFLAGS_vmlinux.bin := -R .comment -S
$(obj)/vmlinux.bin: vmlinux FORCE
    $(call if_changed,objcopy)
```

我们再次看到了熟悉的 objcopy，也就是说，arch/x86/boot/compressed 目录下的 vmlinux.bin 是由 vmlinux 复制而来。而 vmlinux 没有任何修饰前缀，这说明其就是最顶层目录下的有效载荷。但是在这里我们也看到，这次复制过程只是删除了 “.comment” 段，以及符号表和重定位表（通过参数 -S 指定），而有效载荷 vmlinux 的格式依然是 ELF 格式的，如果不使用 ELF 格式的内核，这里追加一个 “-O binary” 即可。

至此，我们明白了 vmlinux.bin.gz 就是有效载荷的压缩。

接着我们再来看构建目标 piggy.S 的命令。进行变量替换后，cmd_mkpiggy 展开为：

```
cmd_mkpiggy = arch/x86/boot/compressed/mkpiggy \
    arch/x86/boot/compressed/vmlinux.bin.gz \
    > arch/x86/boot/compressed/piggy.S
```

其中 mkpiggy 是内核自带的一个工具程序，源码如下：

```
linux-3.7.4/arch/x86/boot/compressed/mkpiggy.c :

int main(int argc, char *argv[])
{
    ...
    printf(".section \".rodata..compressed\\\", \\\"a\\\", \\
@progbits\\n");
```

```

printf(".globl z_input_len\n");
printf("z_input_len = %lu\n", ilen);
printf(".globl z_output_len\n");
printf("z_output_len = %lu\n", (unsigned long)olen);
printf(".globl z_extract_offset\n");
printf("z_extract_offset = 0x%lx\n", offs);
...
printf(".incbin \"%s\"\n", argv[1]);
...
}

```

mkpiggy 向屏幕打印了一堆文本。习惯上，我们会认为标准输出就是屏幕，但是回头再仔细观察一下 cmd_mkpiggy 的定义，其将标准输出重定向到了文件 piggy.S，所以这里 printf 实际上是在组织汇编语句，然后输出到 piggy.S 中。也就是说，mkpiggy 就是在“写”一个汇编程序。

根据代码可见，这个 piggy.S 非常简单，其使用汇编指令 incbin 将压缩的有效载荷 vmlinux.bin.gz 不加更改地直接包含进来。除了包含了压缩的内核映像外，piggy.S 中还定义了解压 vmlinux.bin.gz 时需要的各种信息，包括压缩映像的长度、解压后的长度等，在解压内核时，解压代码将需要这些信息。下面是 mkpiggy 生成的一个具体的 piggy.S 示例：

```

.section ".rodata..compressed", "a", @progbits
.globl z_input_len
z_input_len = 1721557
.globl z_output_len
z_output_len = 3421472
.globl z_extract_offset
z_extract_offset = 0x1b0000
.globl z_extract_offset_negative
z_extract_offset_negative = -0x1b0000
.globl input_data, input_data_end
input_data:
.incbin "arch/x86/boot/compressed/vmlinux.bin.gz"
input_data_end:

```

终于结束了这个让人眩晕的过程，让我们来回顾一下 vmlinux.bin 的构建过程：

- 1) kbuild 使用 objcopy，将顶层 Makefile 构建好的内核映像 vmlinux 复制到 arch/x86/boot/compressed 目录下，删除了“.comment”段、符号表和重定位表，并命名为 vmlinux.bin；

- 2) kbuild 压缩内核映像 vmlinux.bin，笔者采用默认的压缩方式 gzip，所以压缩后的内核映像为 vmlinux.bin.gz；

- 3) kbuild 借助内核自带的程序 mkpiggy 构建一个汇编程序 piggy.S，该汇编程序就是 vmlinux.bin.gz 加上一些解压内核时需要的信息；

- 4) kbuild 将 head_32.o、misc.o 以及包含压缩映像的 piggy.o 等目标文件链接为 vmlinux.bin，保存到 arch/x86/boot 目录下。

可见，vmlinux.bin 由压缩的 vmlinux 加上以 head_32.o 为代表的一小部分非压缩代码组成。vmlinux 就是我们提到的有效载荷，而这部分非压缩代码就是我们所谓的二级推进系统。

3.2.5 setup.bin 的构建过程

构建 setup.bin 的规则也在 arch/x86/boot 目录下的 Makefile 中：

```
linux-3.7.4/arch/x86/boot/Makefile:

setup-y += a20.o bioscall.o cmdline.o copy.o cpu.o cpuchek.o
...
SETUP_OBJS = $(addprefix $(obj)/,$(setup-y))
...
LDFLAGS_setup.elf := -T
$(obj)/setup.elf: $(src)/setup.ld $(SETUP_OBJS) FORCE
    $(call if_changed,ld)

OBJCOPYFLAGS_setup.bin := -O binary
$(obj)/setup.bin: $(obj)/setup.elf FORCE
    $(call if_changed,objcopy)
```

根据 setup.bin 的构建命令可见，setup.bin 是由 setup.elf 经过 objcopy 复制而来的。根据构建 setup.elf 的规则可见，构建 setup.elf 的命令为 cmd_ld，其定义如下：

```
linux-3.7.4/scripts/Makefile.lib:

cmd_ld = $(LD) $(LDFLAGS) $(ldflags-y) $(LDFLAGS_$(@F)) \
    $(filter-out FORCE,$^) -o $@
```

这里 LD 就是链接器 i686-none-linux-gnu-ld，定义在顶层 Makefile 中：

```
linux-3.7.4/Makefile:

LD      = $(CROSS_COMPILE)ld
```

链接器的输出就是规则的目标（“\$@”），这里就是 setup.elf。“\$^”也是 make 的一个自动变量，表示规则的全部依赖，所以这里链接器的输入即是规则的依赖，但是使用 make 的内置函数 filter-out 过滤掉了依赖中的伪目标 FORCE，因此输入是 arch/x86/boot/setup.ld 和 \$(SETUP_OBJS)。其中，setup.ld 是传递给链接器的链接脚本；SETUP_OBJS 对应的则是变量 setup-y 中记录的目标文件，只不过使用 make 的内置函数 addprefix 在这些文件前面中添加了一个前缀，目的是在顶层目录中能找到这些目标文件。

这里我们看到了 kbuild 的一个约定，虽然都在 arch/x86/boot 目录下，但是引用原本就存在的文件 setup.ld 使用的是变量 src，而引用动态创建的 SETUP_OBJS 则使用了变量 obj。

将上述变量替换到 cmd_ld，cmd_ld 最后展开为：

```
cmd_ld = i686-none-linux-gnu-ld -T arch/x86/boot/setup.ld \
    arch/x86/boot/a20.o arch/x86/boot/bioscall.o ... \
    -o arch/x86/boot/setup.elf
```

也就说，链接器依照链接脚本 setup.ld，将 arch/x86/boot 目录下的目标文件 a20.o、bioscall.o 等链接为 setup.elf。

但是 setup.elf 也是 ELF 格式的，ELF 附加的一些信息对内核是没有意义的，所以 kbuild 也将 ELF 格式的 setup.elf 转换为裸二进制格式的 setup.bin。至此，一级推进系统准备完成。

3.2.6 bzImage 的组合过程

一级推进系统和包括有效载荷的二级推进系统都已就绪，这一节，我们就来讨论一级推进系统和二级推进系统的组合。组合的规则定义在平台的“顶层” Makefile 中：

```
linux-3.7.4/arch/x86/Makefile:

boot := arch/x86/boot
...
KBUILD_IMAGE := $(boot)/bzImage
...
bzImage: vmlinux
...
$(Q) $(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
```

在将各个变量进行替换后，构建 bzImage 的命令展开为：

```
make -f scripts/Makefile.build obj=arch/x86/boot
      arch/x86/boot/bzImage
```

Makefile.build 将包含在 arch/x86/boot 目录下的 Makefile 文件组成为最终的 Makefile。构建目标 arch/x86/boot/bzImage 的规则在 arch/x86/boot 下的 Makefile 中：

```
linux-3.7.4/arch/x86/boot/Makefile:

$(obj)/bzImage: $(obj)/setup.bin $(obj)/vmlinux.bin \
               $(obj)/tools/build FORCE
    $(call if_changed, image)
```

我们来看看构建 bzImage 的命令 cmd_image，其在 arch/x86/boot/Makefile 中定义：

```
linux-3.7.4/arch/x86/boot/Makefile:

cmd_image = $(obj)/tools/build $(obj)/setup.bin \
            $(obj)/vmlinux.bin > $@
```

根据 cmd_image 的定义，表面上就是执行程序 build，并传递给程序 build 两个参数，分别是 arch/x86/boot 目录下的 setup.bin 和 vmlinux.bin，同时将程序 build 的标准输出 stdout 重定向到规则的目标（\$@），即 bzImage。那么程序 build 究竟做了什么呢？我们来看看它的源码：

```
linux-3.7.4/arch/x86/boot/tools/build.c:

1 /* This must be large enough to hold the entire setup */
2 u8 buf[SETUP_SECT_MAX*512];
3 ...
4 int main(int argc, char ** argv)
5 {
```

```

6      ...
7      void *kernel;
8      ...
9      file = fopen(argv[1], "r");
10     ...
11     c = fread(buf, 1, sizeof(buf), file);
12     ...
13     fd = open(argv[2], O_RDONLY);
14     ...
15     kernel = mmap(NULL, sz, PROT_READ, MAP_SHARED, fd, 0);
16     ...
17     if (fwrite(buf, 1, i, stdout) != i)
18     ...
19     if (fwrite(kernel, 1, sz, stdout) != sz)
20     ...
21 }

```

1) argv[1] 对应的是 setup.bin, 所以第 9 行代码就是将文件 setup.bin 打开, 第 11 行代码是将其内容读到数组 buf 中。由第 1 行的注释可见, 数组 buf 就是用来存放 setup.bin 的。

2) argv[2] 对应的是 vmlinux.bin, 所以第 13 行代码是将文件 vmlinux.bin 打开。因为 vmlinux.bin 尺寸较大, build 并没有使用与 setup.bin 相同的方式读取 vmlinux.bin, 而是将 vmlinux.bin 映射到 build 的进程空间中, 变量 kernel 指向了 vmlinux.bin 映射的基址, 如代码第 15 行所示。也就是说, build 通过内存映射的方式读取文件 vmlinux.bin。

3) 第 17 行代码是将读取到 buf 中的 setup.bin 写入到标准输出 (stdout)。而根据 cmd_image 的定义, build 程序已经将其标准输出重定向为 bzImage, 所以这里并不是将 setup.bin 显示到屏幕上, 而是写入到文件 bzImage 中。

4) 同理第 19 行代码是将 vmlinux.bin 写入到文件 bzImage 中。

可见, 程序 build 就是将 setup.bin 和 vmlinux.bin 简单地连接为 bzImage。

3.2.7 内核映像构建过程总结

前面我们简要讨论了内核映像的构建, 内核映像的构建过程大体上可以概括为“三次编译链接, 一次组合”, 如图 3-2 所示。

(1) 第一次编译链接

kbuild 分别编译各个子目录下的目标文件, 如 built-in.o、lib.a (如果有) 等, 然后将它们链接为 ELF 格式的 vmlinux, 并存放在顶层目录中。这一步相当于构建有效载荷。

(2) 第二次编译链接

kbuild 使用工具 objcopy, 将顶层目录的 vmlinux 复制到 arch/x86/boot/compressed 目录下, 去掉其中的符号信息、重定位信息, 删除段 “.comment”, 并命名为 vmlinux.bin。然后, kbuild 将其压缩为 vmlinux.bin.gz (假设内核采用核默认的 gzip 压缩方式), 封装到 piggy.S 中, 并调用汇编器将其编译为 piggy.o, 这一步是对有效载荷进行了压缩。

同时, kbuild 也调用编译器编译 arch/x86/boot/compressed 目录下的 head_32.c、misc.c 等

作为内核的非压缩部分，这一步相当于构建二级推进系统。

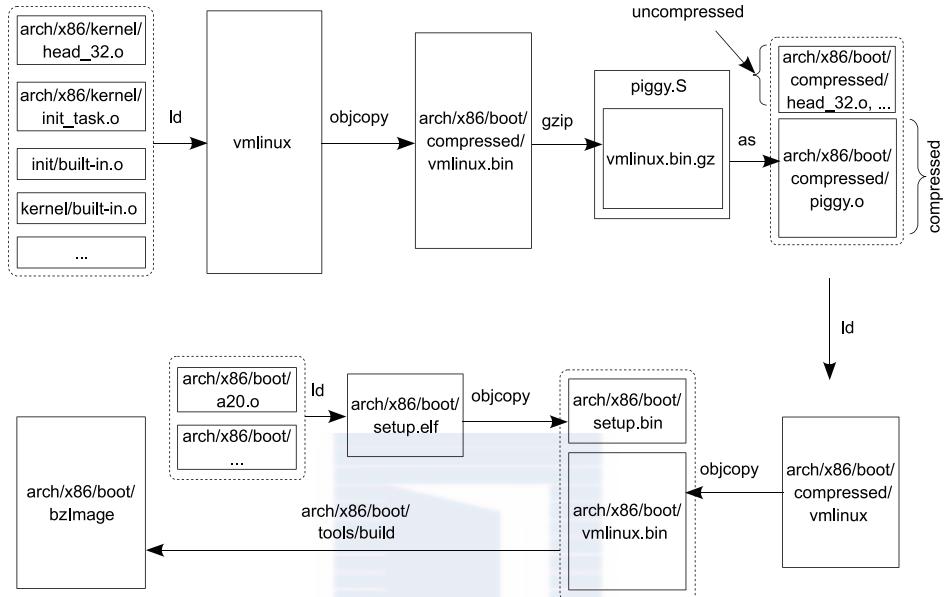


图 3-2 内核映像构建过程

然后，kbuild 调用链接器将压缩的有效载荷和二级推进系统链接为 vmlinux。注意这里文件名虽然也是 vmlinux，但是不要与顶层目录下的 vmlinux 混淆，arch/x86/boot/compressed 目录下的 vmlinux 是二级推进系统和有效载荷的组合，与顶层目录下的 vmlinux 是包含的关系。

最后，kbuild 调用 objcopy 将 arch/x86/boot/compressed 目录下的 vmlinux 复制到 arch/x86/boot 目录下，同时将其转换为裸二进制格式，并命名为 vmlinux.bin，为在 arch/x86/boot 目录下进行的最后的组装做好准备。

(3) 第三次编译链接

kbuild 将 arch/x86/boot 下的 a20.o、bioscall.o 等目标文件链接为 setup.elf，使用 objcopy 将其转换为裸二进制格式，并命名为 setup.bin。这一步，相当于构建一级推进系统。

(4) 一次组合

最后，kbuild 调用内核自带的程序 build，将 vmlinux.bin 和 setup.bin 合并为 bzImage。至此，航天器的一级推进系统和包含有效载荷的二级推进系统装配完毕。

在 3.1 节，我们曾粗略讨论了内核映像的组成。在了解了内核的构建过程后，让我们近距离的再观察一下 bzImage。以下是 bzImage 的链接脚本：

linux-3.7.4/arch/x86/boot/compressed/vmlinux.lds.S:

```

SECTIONS
{
    . = 0;
    .head.text : {

```

```

        _head = . ;
        HEAD_TEXT
        _ehead = . ;
    }
.rodata..compressed : {
    *(.rodata..compressed)
}
.text : {
    ...
}
.rodata : {
    ...
}
.got : {
    ...
}
.data : {
    ...
}
.= ALIGN(L1_CACHE_BYTES);
.bss : {
    _bss = . ;
    ...
    _ebss = . ;
}
#endif CONFIG_X86_64
...
#endif
_end = . ;
}

```

首先来看链接脚本中的段“.head.text”，其中宏 HEAD_TEXT 的定义为：

```
linux-3.7.4/include/asm-generic/vmlinux.lds.h:
#define HEAD_TEXT *(.head.text)
```

而结合文件 head_32.S：

```
linux-3.7.4/arch/x86/boot/compressed/head_32.S:
.text

#include <linux/init.h>
...
__HEAD
ENTRY(startup_32)
...
ENDPROC(startup_32)

.text
relocated:

/*
 * Clear BSS (stack is currently empty)
 */
```

```

xorl    %eax, %eax
...

```

以及宏 `_HEAD` 的定义：

```

linux-3.7.4/include/linux/init.h:

#define _HEAD      .section ".head.text", "ax"

```

可见，在 `head_32.S` 中，函数 `startup_32` 通过宏 `_HEAD` 明确要求链接器将函数 `startup_32` 链接到段 “`.head.text`”。而根据 `bzImage` 的链接脚本，段 “`.head.text`” 被安排在了内核映像的起始位置，也就是说，函数 `startup_32` 被链接到了内核映像的开头。

接下来的段 “`.rodata..compressed`”，想必读者一定猜出来了，这里就是放置内核的压缩映像部分。根据 `piggy.S` 的内容即可见这一点：

```

linux-3.7.4/arch/x86/boot/compressed/piggy.S:

.section ".rodata..compressed", "a", @progbits
.globl z_input_len
z_input_len = 1721556
...
.incbin "arch/x86/boot/compressed/vmlinux.bin.gz"
input_data_end:

```

在 `piggy.S` 中，明确定义了内核压缩部分所在的段为 “`.rodata..compressed`”。

接下来的 “`.text`”、“`.data`” 等段就是保存内核非压缩部分的代码和数据了，包括 `misc.o`、`string.o`、`cmdline.o`、`early_serial_console.o` 以及 `head_32.o` 中的不属于段 “`.head.text`” 的部分。因为内核非压缩部分被编译为位置无关（PIC）代码，所以我们看到其包含 `got` 表。

综上所述，`bzImage` 的布局如图 3-3 所示。

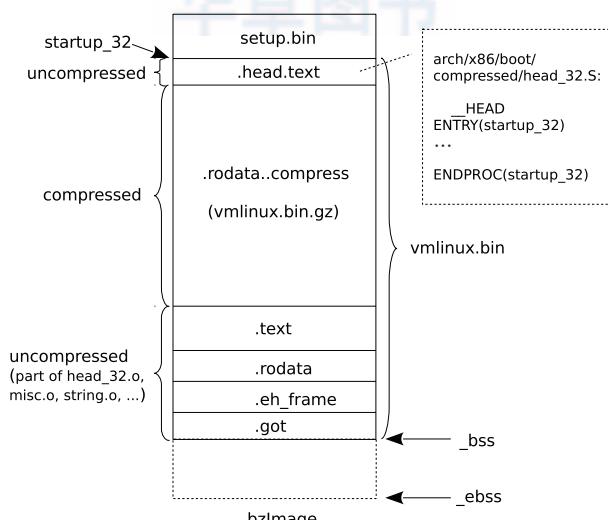


图 3-3 内核映像 `bzImage` 的布局

3.3 配置内核

内核提供了 make menuconfig、make xconfig、make gconfig 等具有图形界面的配置方式。make menuconfig 是图形界面配置方式中最简陋的一种，但是却非常方便易用，依赖也最小。其他如 make xconfig、make gconfig 需要 QT、GTK+ 等库的支持。在本书中，我们使用 make menuconfig 配置内核，其简单地基于终端的图形界面是使用 ncurses 编写的，因此需要安装 libncurses5-dev，安装方法如下：

```
root@baisheng:~# apt-get install libncurses5-dev
```

3.3.1 交叉编译内核设置

在默认情况下，内核构建系统默认内核是本地编译，即编译的内核是运行在与宿主系统相同的体系架构上。如果是为其他的架构编译内核，即交叉编译，我们需要设置两个变量：ARCH 和 CROSS_COMPILE。其中：

- ❑ ARCH 指明目标体系架构，即编译好的内核运行在什么平台上，如 x86、arm 或 mips 等。
- ❑ CROSS_COMPILE 指定使用的交叉编译器的前缀。对于我们的交叉工具链来说，其前缀是 i686-none-linux-gnu-。

在顶层的 Makefile 中，我们可以看到工具链中的编译器、链接器等均以 \$(CROSS_COMPILE) 作为前缀：

```
linux-3.7.4/Makefile :
AS      = $(CROSS_COMPILE)as
LD      = $(CROSS_COMPILE)ld
CC      = $(CROSS_COMPILE)gcc
CPP     = $(CC) -E
AR      = $(CROSS_COMPILE)ar
NM      = $(CROSS_COMPILE)nm
STRIP   = $(CROSS_COMPILE)strip
OBJCOPY = $(CROSS_COMPILE)objcopy
OBJDUMP = $(CROSS_COMPILE)objdump
```

可以使用多种方式定义这两个变量，比如通过在环境变量中定义 ARCH、CROSS_COMPILE；或者每次执行 make 时，通过命名行为这两个变量的赋值，如：

```
make ARCH=i386 CROSS_COMPILE=i686-none-linux-gnu-
```

也可以直接更改顶层 Makefile。这种方法比较方便，但是要小心，以免破坏 Makefile 文件。本书中我们采用这种方式，将顶层 Makefile 中的如下脚本：

```
linux-3.7.4/Makefile :
ARCH      ?= $(SUBARCH)
CROSS_COMPILE ?= $(CONFIG_CROSS_COMPILE:"%"=%)
```

更改为：

```
linux-3.7.4/Makefile :
ARCH      ?= i386
CROSS_COMPILE ?= i686-none-linux-gnu-
```

3.3.2 基本内核配置

编译内核的第一步是配置内核，但是在我们使用的这一版的内核中，有成千上万的配置项，并且很多配置项彼此之间存在着非常紧密的依赖关系，如果从零开始一项一项地配置，显然不是一个好办法。

幸运的是，在很多情况下，我们都会有一个目标系统的老版本内核配置文件，而不必每次都从零开始。在此种情况下，首先将已有的内核配置文件复制到顶层目录下，并命名为`.config`；然后运行`make oldconfig`，其将会询问用户如何处理变动的内核配置；最后用户可以使用`make menuconfig`进行微调。虽然内核提供`make oldconfig`的方法，但是这些方法并不是完美的，读者需要小心处理新内核中新增或改变的配置项。

但是也有很多情况，已有配置并不理想，我们需要进行更彻底定制，或者我们根本找不到一个合适的已有配置。难道我们就别无选择，只能从零开始了吗？当然不是，内核构建系统已经为开发者考虑了这些。

一方面内核为很多平台附带了默认配置文件，保存在`arch/<arch>/configs`目录下，其中`<arch>`对应具体的架构，如`x86`、`arm`或者`mips`等。比如，对于`x86`架构，内核分别提供了32位和64位的配置文件，即`i386_defconfig`和`x86_64_defconfig`；对于`arm`架构，内核提供了如NVIDIA的Tegra平台的默认配置`tegra_defconfig`，Samsung的S5PV210平台的默认配置`s5pv210_defconfig`等。

如果我们打算使用`x86`的32位的默认配置，执行下面命令即可：

```
make i386_defconfig
```

如果想使用Samsung的S5PV210平台的默认配置，则使用如下命令：

```
make ARCH=arm s5pv210_defconfig
```

如果对这些内核内置的默认配置依然不满意，`kbuild`还提供了创建一个最小配置的方法，从某种意义上讲，这是最彻底的定制方式了，命令如下：

```
make allnoconfig
```

执行该命令后，内核除了选中必选项外，其余全部不选。我们举个例子来展示这个配置方式，例如某`Kconfig`文件中有如下配置：

```
config A
  def_bool y
```

```

config B
    def_bool y if X86_64

config C
    def_tristate y
    select D

config D
    bool

config E
    bool "config E"

config F
    bool "config F"
    default y

```

如果我们在 IA32 上执行“make allnoconfig”，则内核构建系统基本按照如下规则处理上述各配置项。

- config A：无条件选中。
- config B：不会被选中，因为平台不是 X86_64 架构。
- config C：无条件选中。另外，因为该选项明确要求选中 D，所以选项 D 也会被选中。
- config E：不会被选中。
- config F：不会被选中。虽然该选项指出默认值“default y”，但是注意“default y”和“def_bool y”是有本质区别的，“def_bool y”是无条件选中，“default y”只是建议。

执行 make allnoconfig 后，生成的配置文件 .config 如下：

```

CONFIG_A=y
CONFIG_C=y
CONFIG_D=y
# CONFIG_E is not set
# CONFIG_F is not set

```

在本书中，我们基于 make allnoconfig 的结果开始配置内核，命令如下：

```
vita@baisheng:/vita/build/linux-3.7.4$ make allnoconfig
```

接下来各节中，我们以这个基本配置为基础，按照需要进行具体的配置。希望读者可以通过这个过程的学习，能够做到举一反三，在具体的项目中进行最优的配置。

3.3.3 配置处理器

1. 选择处理器型号

对于 x86 架构来说，其具有向后兼容性，较新的处理器都支持较早的处理器的指令。因此，为较早的处理器开发的程序都可以在较新的处理器上运行，但是反过来则不一定是，因为较早的处理器当然不会支持较新的处理器中的一些指令。

因此，选择支持越早的处理器，则内核就可以在更多的机器上运行。对于很多 Linux 发行版，通常就是依照这个原则。比如 Ubuntu12.10 的内核配置支持的处理器型号为 Pentium Pro (686)，这样理论上可以确保 Ubuntu12.10 可以运行在 Pentium Pro 以后的所有系列机器上。

如此选择虽然带来了兼容性的好处，但是付出的代价可能就是丧失了速度。比如，为 Pentium Pro 编译的内核，只针对 Pentium Pro 进行了优化，显然不能使用最新处理器中的更高级的指令。

对于其他架构亦如此，甚至有过之而无不及，比如都是 ARM 处理器，但是如果目标平台是 Freescale iMX 系列，处理器型号显然不能选择 Samsung 的 S3C 系列。

在 Linux 操作系统中，可以使用如下命令查看处理器的具体型号：

```
root@baisheng:~# cat /proc/cpuinfo | grep "model name"
model name : Intel(R) Core(TM) i5-2430M CPU @ 2.40GHz
model name : Intel(R) Core(TM) i5-2430M CPU @ 2.40GHz
model name : Intel(R) Core(TM) i5-2430M CPU @ 2.40GHz
model name : Intel(R) Core(TM) i5-2430M CPU @ 2.40GHz
```

下面是配置内核支持的处理器型号的步骤。

1) 执行 make menuconfig，出现如图 3-4 所示界面。

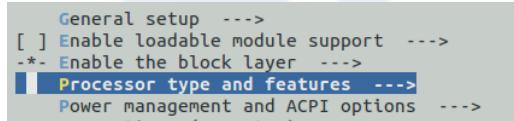


图 3-4 配置处理器型号 (1)

2) 在图 3-4 中，选择菜单项“Processor type and features”，出现如图 3-5 所示界面。

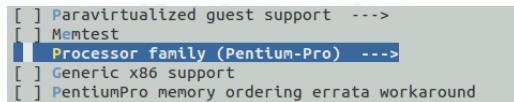


图 3-5 配置处理器型号 (2)

3) 在图 3-5 中，选择菜单项“Processor family”，出现如图 3-6 所示的界面。

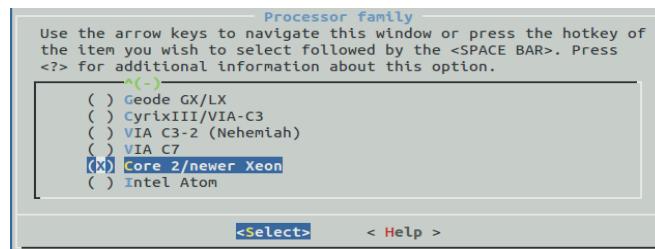


图 3-6 配置处理器型号 (3)

4) 以笔者的机器为例，根据前面察看的 CPU 信息，显然选择图 3-6 中的“Core 2/newer

Xeon”是最适合的。如果在列表中没有与实际CPU型号完全吻合的，可选择与它最接近的一项。

2. 配置内核支持 SMP

如果机器有多颗CPU（包括多核），为了更好地发挥多颗CPU的性能，需要配置内核支持SMP。下面是配置内核支持SMP的步骤。

- 1) 执行make menuconfig，出现如图3-7所示的界面。

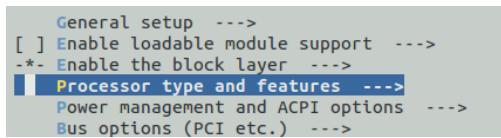


图3-7 配置SMP(1)

- 2) 在图3-7种，选择菜单项“Processor type and features”，出现如图3-8所示的界面。

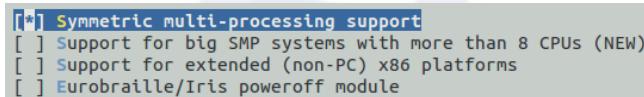


图3-8 配置SMP(2)

- 3) 在图3-8中，选中“Symmetric multi-processing support”。

3.3.4 配置内核支持模块

在嵌入式系统中，由于外围设备相对比较固定，因此，在编译内核时，基本可以确定内核需要支持哪些特性，例如支持哪些硬件、支持哪些文件系统等。而对于用在PC系统上的内核，因为个人计算机中包含的硬件千差万别，为了提供更好的兼容性，各家Linux发行版的内核都尽可能地包含更多的功能，支持更多的硬件。但是，如果所有的功能模块和驱动全部编译进内核映像，势必造成内核极其庞大。以作者使用的Ubuntu12.10发行版为例，其内核映像大小为5MB，而该发行版中包含的内核模块的尺寸约为100MB左右。也就是说，如果把全部的模块都编译进内核映像，内核映像的尺寸大约要增加100MB，而其中绝大部分模块在特定的一台机器上是根本不会用到的。

除了尺寸上的考虑外，更大的灵活性也是一方面。比如，开发人员在开发某个驱动时，如果使用模块机制，只需单独编译驱动，然后动态加载，即可进行调试；而不必重新编译整个内核，甚至重启系统。

因此，在我们编译的内核中，启用内核的动态加载模块特性。下面是配置内核支持模块机制的步骤。

- 1) 执行make menuconfig，出现如图3-9所示的界面。

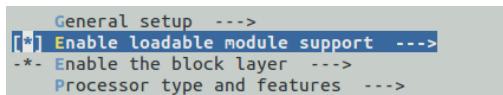


图 3-9 配置内核支持模块（1）

2) 在图 3-9 中, 选中菜单项“Enable loadable module support”, 允许内核动态加载模块, 出现如图 3-10 所示的界面。

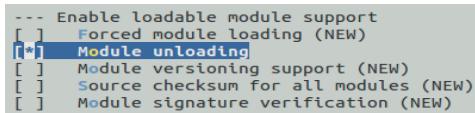


图 3-10 配置内核支持模块（2）

3) 在图 3-10 中, 选中“Module unloading”, 允许内核动态卸载模块。

3.3.5 配置硬盘控制器驱动

一般而言, PC 的根文件系统都保存在硬盘上, 因此, 我们需要配置内核的硬盘驱动。在笔者写作这本书时, 大多数现代 PC 都使用 SATA 接口的硬盘, SATA 硬盘基本已经全面取代了 IDE 硬盘。因此, 我们以 SATA 硬盘为例, 讨论内核中硬盘驱动的配置。关于 SATA 控制器驱动的配置, 需要从三个方面考虑。

(1) 硬盘控制器的接口

SATA 控制器使用的是 PCI 接口, 挂在 PCI 总线上, 所以首先需要配置内核支持 PCI 总线。

可以使用 `lspci` 命令查看 SATA 硬盘的相关信息。下面以笔者机器为例, 执行 `lspci` 命令输出的关于 SATA 控制器的相关信息 :

```
root@baisheng:~# lspci -v

00:1f.2 SATA controller: Intel Corporation 6 Series/C200 Series Chipset Family 6
port SATA AHCI Controller (rev 04) (prog-if 01 [AHCI 1.0])
...
Kernel driver in use: ahci
```

显然, 在 0 号 PCI 总线上, 有一个 SATA 控制器, 工作模式为 AHCI, 使用的内核驱动是 `ahci`。

(2) 与 SCSI 层之间的关系

在内核中, SATA 设备被实现为一个 SCSI 设备, 如图 3-11 所示。

因此, 虽然目标机器上可能没有 SCSI 设备, 但是如果要支持 SATA 控制器, 内核也要配置支持 SCSI。

(3) 底层设备驱动

在图 3-11 中, 内核将 SATA 驱动从逻辑上划分为两层 :

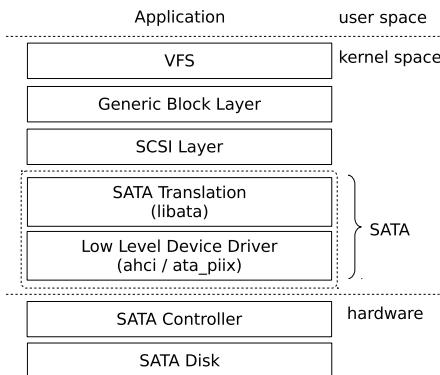


图 3-11 SATA 子系统结构

- ❑ SATA Translation，这一层负责 SCSI 和 SATA 协议之间的翻译，在 SATA 驱动中被封装为 libata 模块。
- ❑ Low Level Device Driver，在 SATA Translation 层下，是直接面对设备的底层驱动。很多 SATA 控制器均提供两种可选模式：一种是模拟传统的 IDE，通常称为 Compatibility 模式，这种模式是为了向后兼容那些较早的不支持 SATA 的操作系统；另外一种是 AHCI 模式，这种模式可以提供更好的性能及传输速度。对于 Intel 的 SATA 控制器来说，内核为这两种不同的模式分别实现了驱动：ata_piix 和 ahci。ata_piix 驱动用于 Compatibility 模式，ahci 驱动用于 AHCI 模式。

从 kbuild 中有关 SATA 的 Kconfig 中，我们也可以清楚地看到 SATA 控制器对 PCI 和 SCSI 的依赖：

```

linux-3.7.4/drivers/ata/Kconfig :
menuconfig ATA
    tristate "Serial ATA and Parallel ATA drivers"
    ...
    select SCSI
    ...
config SATA_AHCI
    tristate "AHCI SATA support"
    depends on PCI
    ...
config ATA_PIIX
    tristate "Intel ESB, ICH, PIIX3, PIIX4 PATA/SATA support"
    depends on PCI

```

先看配置项 ATA，正如配置中的描述“Serial ATA and Parallel ATA drivers”，这一项对应于 SATA 和 PATA 设备。注意其中用黑体标识的部分，这一配置项是依赖 SCSI 的而且 ATA 是选择（select）依赖 SCSI。也就是说，一旦配置内核支持 ATA 设备，kbuild 将自动选中内核的 SCSI 支持。

再来看配置项 SATA_AHCI 和 ATA_PIIX，这两项对应的就是前面所说的 SATA 控制器的

驱动，SATA_AHCI 用于驱动 AHCI 模式，ATA_PIIX 用于驱动 Compatibility 模式。根据上面我们用黑体标示的部分，可以清楚地看到，这两项均要求内核支持 PCI 总线。

下面我们就具体配置这三个部分。

1. 配置 PCI 总线

因为 SATA 控制器使用的是 PCI 接口，所以我们首先来配置内核支持 PCI 总线。

1) 执行 make menuconfig，出现如图 3-12 所示的界面。

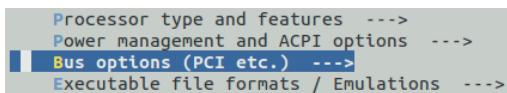


图 3-12 配置 PCI 总线 (1)

2) 在图 3-12 中，选择菜单项“Bus options”，出现如图 3-13 所示的界面。

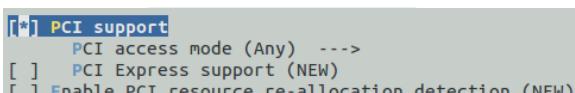


图 3-13 配置 PCI 总线 (2)

3) 在图 3-13 中，选中菜单项“PCI support”。PCI 总线配置完毕。

2. 配置 SCSI

接下来配置 SCSI。

1) 执行 make menuconfig，出现如图 3-14 所示的界面。

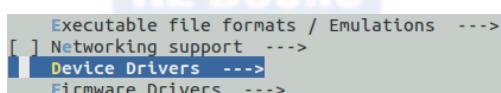


图 3-14 配置 SCSI (1)

2) 在图 3-14 中，选择菜单项“Device Drivers”，出现如图 3-15 所示的界面。

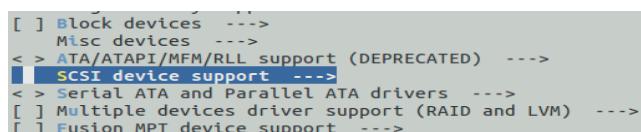


图 3-15 配置 SCSI (2)

3) 在图 3-15 中，选择菜单项“SCSI device support”，出现如图 3-16 所示的界面。

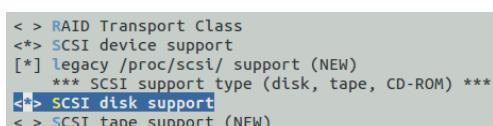


图 3-16 配置 SCSI (3)

4) 在图 3-16 中, 选中 “SCSI device support” 和 “SCSI disk support”, 注意将它们都编译进内核, 而不是编译为模块。SCSI 配置完毕。

3. 配置 SATA 控制器驱动

下面来配置 SATA 控制器驱动。

1) 执行 make menuconfig, 出现如图 3-17 所示的界面。

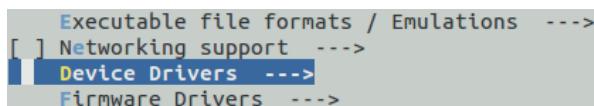


图 3-17 配置 SATA 控制器驱动 (1)

2) 在图 3-17 中, 选择菜单项 “Device Drivers”, 出现如图 3-18 所示的界面。

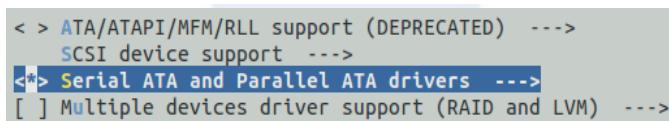


图 3-18 配置 SATA 控制器驱动 (2)

3) 在图 3-18 中, 选择 “Serial ATA and Parallel ATA drivers” (注意将它编译进内核, 而不是编译为模块), 出现如图 3-19 所示的界面。

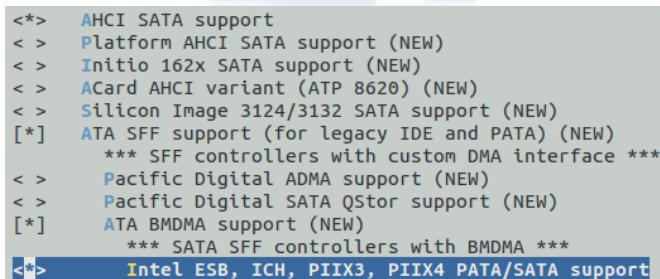


图 3-19 配置 SATA 控制器驱动 (3)

4) 笔者的机器使用的是 Intel SATA 控制器, 所以选择图 3-19 中的 “AHCI SATA support” 和 “Intel ESB, ICH, PIIX3, PIIX4 PATA/SATA support”。前者是工作在 AHCI 模式的 Intel SATA 控制器的驱动, 后者是工作在 Compatibility 模式的 Intel SATA 控制器的驱动。注意将它们也都编译进内核。

至此, SATA 控制器的驱动配置完成。接下来我们编译内核, 并将编译好的内核保存在目标系统的根文件系统的 boot 目录下。

```
vita@baisheng:/vita/build/linux-3.7.4$ make bzImage
vita@baisheng:/vita/build/linux-3.7.4$ mkdir /vita/sysroot/boot/
```

```
vita@baisheng:/vita/build/linux-3.7.4$ cp arch/x86/boot/bzImage \
/vita/sysroot/boot/
```

下面测试新编译的内核。首先在虚拟机的 sda2 分区上创建 boot 目录，用来存放内核映像：

```
root@baisheng-vb:/vita# mkdir boot
```

将新编译的内核复制到虚拟机：

```
vita@baisheng:/vita/sysroot/boot$ scp bzImage \
root@192.168.56.101:/vita/boot/
```

并在虚拟机 GRUB 的配置文件中添加如下启动项：

```
/boot/grub/grub.cfg
```

```
menuentry 'vita' {
    set root='(hd0,2)'
    linux /boot/bzImage root=/dev/sda2 ro
}
```

注意将虚拟机的 GRUB 的配置文件 grub.cfg 中的 timeout 都设置为一个正值，比如 5s，这样 GRUB 才会给我们机会选择引导哪个系统。

然后重新启动并进入 vita 系统，运行结果如图 3-20 所示。

```
11.10 [正在运行] - Oracle VM VirtualBox
sd 0:0:0:0: [sda] 16777216 512-byte logical blocks: (8.58 GB/8.00 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO or FUA
sda: sda1 sda2
sd 0:0:0:0: [sda] Attached SCSI disk
List of all partitions:
 0:8000      8388608 sda  driver: sd
 0:8001      4881408 sda1 00000000-0000-0000-0000-000000000000
 0:8002      3506176 sda2 00000000-0000-0000-0000-000000000000
No filesystem could mount root, tried:
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(8,2)
Pid: 1, comm: swapper/0 Not tainted 3.7.4 #1
Call Trace:
[<c11a500b>] ? panic+0x7d/0x158
[<c124caf6>] ? mount_block_root+0x228/0x239
[<c1002932>] ? sys_sigaction+0xa2/0xf0
[<c124cb52>] ? mount_root+0x4b/0x5f
[<c124cc74>] ? prepare_namespace+0x10e/0x14a
[<c109918f>] ? sys_access+0x1f/0x30
[<c119e7d4>] ? kernel_init+0x174/0x270
[<c124c3c2>] ? do_early_param+0x77/0x77
[<c11a9577>] ? ret_from_kernel_thread+0x1b/0x28
[<c119e660>] ? rest_init+0x60/0x60
```

图 3-20 配置 SATA 控制器驱动后内核运行情况

根据内核的输出信息可见，内核已经正确识别了 SATA 硬盘。但是因为没有找到合适的文件系统挂载 sda2 分区，所以在“抱怨”“No filesystem could mount root”后出现了“panic”。因此，在下一小节，我们配置内核对文件系统的支持。

3.3.6 配置文件系统

内核支持多种文件系统，但是由于我们仅使用 Ext4 文件系统，所以这里仅配置内核包含 Ext4 文件系统驱动模块。Ext4 驱动是向后兼容的，也就是说它也可以驱动 Ext3 和 Ext2 文件系统。下面是配置文件系统的步骤。

- 1) 执行 make menuconfig，出现如图 3-21 所示的界面。



图 3-21 配置文件系统 (1)

- 2) 在图 3-21 中，选择菜单项 “File Systems”，出现如图 3-22 所示的界面。

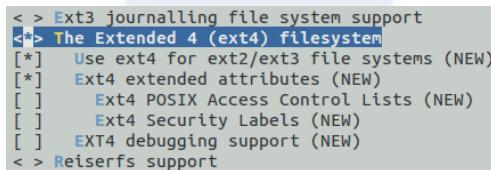


图 3-22 配置文件系统 (2)

- 3) 在图 3-22 中，选中配置项 “The Extended 4 (ext4) filesystem”，并将其直接编译进内核。

在格式化 Ext4 文件系统时，工具 mke2fs.ext4 会默认支持 “huge_file” 特性，而该特性要求内核支持大于 2TB 的块设备或文件，因此，我们配置内核支持这一特性。

- 1) 执行 make menuconfig，出现如图 3-23 所示的界面。

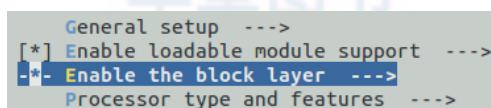


图 3-23 配置支持大于 2TB 的块设备和文件 (1)

- 2) 在图 3-23 中，选择菜单项 “Enable the block layer”，出现如图 3-24 所示的界面。

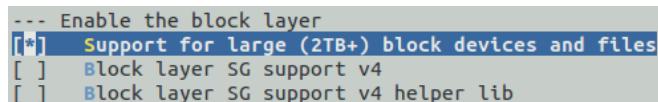


图 3-24 配置支持大于 2TB 的块设备和文件 (2)

- 3) 在图 3-24 中，选中配置项 “Support for large (2TB+) block devices and files”。

3.3.7 配置内核支持 ELF 文件格式

在上一节，我们配置了内核支持 Ext4 文件系统。但是内核从文件系统加载文件，仅支持文件系统还是不够的，内核还要支持具体的文件格式，当前 Linux 系统使用的标准二进制格式是 ELF，因此需要配置 Linux 支持 ELF 文件格式。以下是配置内核支持 ELF 文件格式的步骤。

1) 执行 make menuconfig，出现如图 3-25 所示的界面。

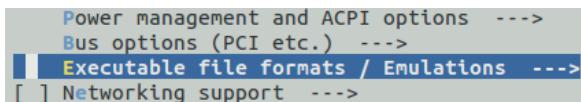


图 3-25 配置内核支持 ELF 文件格式 (1)

2) 在图 3-25 中，选择菜单项“Executable file formats / Emulations”，出现如图 3-26 所示的界面。

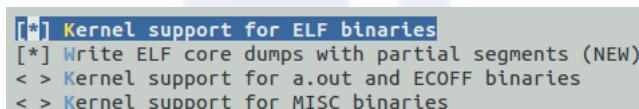


图 3-26 配置内核支持 ELF 文件格式 (2)

3) 在图 3-26 中，选中配置项“Kernel support for ELF binaries”。ELF 格式支持配置完毕。

在配置内核支持 ELF 文件格式后，我们重新编译内核并使用新编译的内核引导 vita 系统后，系统的输出如图 3-27 所示。

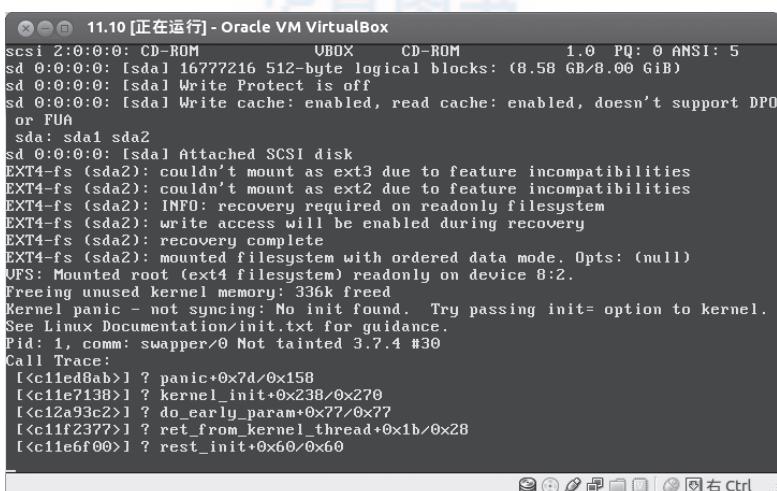


图 3-27 配置文件系统和文件格式后内核运行情况

根据内核输出的信息可见，内核已经识别出硬盘的分区，也识别出了 sda2 分区使用的文件系统，并使用 Ext4 文件系统驱动成功挂载了该分区。但是内核在“报怨”“No init found …”后，依然出现了“panic”。那么，这里的“init”指的是什么呢？我们看一下内核进入用户空间的过程：

```
linux-3.7.4/init/main.c:

static noinline void __init_refok rest_init(void)
{
    ...
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    ...
}

static int __ref kernel_init(void *unused)
{
    ...
    if (ramdisk_execute_command) {
        if (!run_init_process(ramdisk_execute_command))
            return 0;
        printk(KERN_WARNING "Failed to execute %s\n",
               ramdisk_execute_command);
    }
    ...
    if (execute_command) {
        if (!run_init_process(execute_command))
            return 0;
        printk(KERN_WARNING "Failed to execute %s. Attempting "
               "defaults...\n", execute_command);
    }
    if (!run_init_process("/sbin/init") ||
        !run_init_process("/etc/init") ||
        !run_init_process("/bin/init") ||
        !run_init_process("/bin/sh"))
        return 0;

    panic("No init found. Try passing init= option to kernel. "
          "See Linux Documentation/init.txt for guidance.");
}
}
```

函数 `kernel_init` 首先尝试执行 `initramfs` 中的字符串 `ramdisk_execute_command` 代表的命令。目前没有使用 `initramfs`，所以这个字符串为空，于是其继续尝试到根文件系统中寻找程序，首先其将尝试寻找字符串 `execute_command` 代表的命令。

根据下面代码：

```
linux-3.7.4/init/main.c:

static char *execute_command;

static int __init init_setup(char *str)
{
```

```

    unsigned int i;

    execute_command = str;
    ...
}

__setup("init=", init_setup);

```

字符串 `execute_command` 代表用户通过内核命令行参数 “`init`” 明确指定的程序，形如：

```

/boot/grub/grub.cfg

menuentry 'vita' {
    set root='(hd0,2)'
    linux   /boot/bzImage root=/dev/sda2 ro init=/bin/bash
}

```

在上面 `grub` 的配置文件中，使用黑体标识的部分就是明确告诉内核，第一个进程直接运行根文件系统中目录 `/bin` 下的 `bash`。如果用户没有通过命令行参数 “`init`” 指定第一个进程执行的程序，这个进程将依次尝试执行 `/sbin`、`/etc`、`/bin` 下的 `init`，最后尝试执行 `/bin/sh`。

由于目前根文件系统中除了内核的映像外没有任何文件，因此，内核找不到任何程序，所以在报告 “`No init found ...`” 后出现 “`panic`” 了。为了辅助验证内核的构建，在下一节我们将构建一个基本的根文件系统。

3.4 构建基本根文件系统

3.4.1 根文件系统的基本目录结构

Linux 的根文件系统的目录结构不是随意定义的，而是依照 Filesystem Hierarchy Standard Group 制定的 Filesystem Hierarchy Standard (FHS) 标准。从服务器、个人计算机到嵌入式系统，虽达不到完全符合，但大体上还是遵循这个标准的。

FHS 标准规定的根文件系统的顶层目录如表 3-2 所示。

表 3-2 FHS 根文件系统顶层的目录规范

目 录	内 容
<code>/bin</code>	保存系统管理员与用户均会使用的重要的命令
<code>/boot</code>	系统开机使用的文件，如内核映像和 boot loader 的相关文件
<code>/dev</code>	设备文件
<code>/etc</code>	系统配置
<code>/lib</code>	重要的库文件及内核模块
<code>/media</code>	可移动存储介质的挂载点
<code>/mnt</code>	临时挂载点，当然用户也可以自行选择一些临时挂载点
<code>/opt</code>	用户自行安装软件的位置，通常用户也会选择将软件安装在 <code>/usr/local</code> 目录下
<code>/sbin</code>	系统管理员使用的重要的系统命令

(续)

目 录	内 容
/tmp	主要是正在执行的程序存放的临时文件
/usr	包含系统中安装的主要程序的相关文件，类似于 MS Windows 操作系统中的“Program files”目录
/var	针对的主要是系统运行过程中经常发生变化的一些数据，比如 cache、log、临时的数据库、打印机的队列等
/home	用户目录保存的地方
/root	root 用户的用户目录
/srv	主要用在服务器版本上，是很多服务器软件用来保存数据的目录。比如，www 服务器使用的网页资料就可以放置在 /srv/www 目录下

FHS 标准已经将各个目录存放的内容解释得比较清楚了，但是还是有几个容易引起混淆的目录需要澄清一下。

根文件系统中主要有四处存放可执行程序的目录：/bin、/sbin、/usr/bin 和 /usr/sbin。系统管理员和普通用户都使用的重要命令保存在 /bin 目录下，而仅由系统管理员使用的重要命令则保存在 /sbin 目录下。相应的，不是很重要的命令则分别放置在 /usr/bin 和 /usr/sbin 目录下。

同样的道理，重要的系统库一般存放在 /lib 目录下，其他的库则存放在 /usr/lib 目录下。

3.4.2 安装 C 库

几乎所有程序都依赖 C 库，它是整个系统的基础，因此，我们首先安装 C 库到根文件系统。在 2.2.7 节讨论编译构建系统的 C 库时，我们看到，C 库包含函数库、各种工具程序，以及开发所需的头文件等。而这里的文件系统只是个临时系统，所以 C 库中的各种实用工具及 \$SYSROOT/usr/share 目录下的数据文件，都不需要安装。而且这个临时根文件系统亦不需要支撑开发，所以凡是开发时所需要的文件，包括头文件、静态库、启动文件等，也不需要安装。因此，最终我们只需要安装 \$SYSROOT/lib 目录下的动态库及相应的动态链接 / 加载器需要的符号链接。

我们新建一个保存目标系统的根文件系统的 rootfs 目录，并且按照 FHS 标准的规定，将 C 库安装在 rootfs/lib 目录下，命令如下：

```
vita@baisheng:/vita$ mkdir rootfs
vita@baisheng:/vita$ mkdir rootfs/lib
vita@baisheng:/vita$ cp -d sysroot/lib/* rootfs/lib/
```

除了 Glibc 中包含的 C 库外，在前面编译 GCC 时，我们也看到，GCC 也将部分底层函数封装到库中，有些程序会使用 GCC 的这些库，因此，我们也将这部分程序安装到 rootfs/lib 目录中。同样，我们也只安装动态库及其对应的运行时符号链接，命令如下：

```
vita@baisheng:/vita$ cp -d \
cross-tool/i686-none-linux-gnu/lib/lib*.so.* [0-9] rootfs/lib/
```

3.4.3 安装 shell

在安装 C 库后，构建基本的应用程序的基础已经具备了，接下来我们需要为内核准备用户空间的程序了。在 Linux 中，专门负责启动的软件包，如 System V init 和 Systemd 等都提供一个二进制程序作为第一个进程执行的用户空间的程序，但是为简单起见，我们使用 bash shell。安装 bash 的命令如下：

```
vita@baisheng:/vita/build$ tar xvf ../source/bash-4.2.tar.gz
vita@baisheng:/vita/build/bash-4.2$ ./configure --prefix=/usr \
    --bindir=/bin --without-bash-malloc
vita@baisheng:/vita/build/bash-4.2$ make
vita@baisheng:/vita/build/bash-4.2$ make install DESTDIR=$SYSROOT
```

这里有一点需要解释一下，我们虽然定义了环境变量 DESTDIR 为 \$SYSROOT，但是由于 bash 的 Makefile 中有如下脚本：

```
bash-4.2/Makefile :

DESTDIR =
```

而 Makefile 中的这个定义优先级要比环境变量的高，所以我们还需要通过命令行参数再次指定安装目录为 \$SYSROOT。

使用如下命令将 bash 安装到 rootfs 中：

```
vita@baisheng:/vita$ mkdir rootfs/bin
vita@baisheng:/vita$ cp sysroot/bin/bash rootfs/bin/
```

除了安装程序 bash 外，当然还需要安装 bash 依赖的动态库。因此，为了检查可执行程序或动态库的依赖，我们编写了一个脚本 ldd：

```
/vita/cross-tool/bin/ldd :

#!/bin/bash

LIBDIR="${SYSROOT}/lib ${SYSROOT}/usr/lib
${CROSS_TOOL}/${TARGET}/lib"

find() {
    for d in $LIBDIR; do
        found=""
        if [ -f "${d}/$1" ]; then
            found="${d}/$1"
            break
        fi
    done

    if [ -n "$found" ]; then
        printf "%8s%8s => %s\n" "" $1 $found
    else
        printf "%8s%8s => (not found)\n" "" $1
```

```

        fi
    }

readelf -d $1 | grep NEEDED \
| sed -r -e 's/.Shared library:[ ]+\[(.*)\]/\1/;' \
| while read lib; do
    find $lib
done

```

并为该脚本增加了可执行权限：

```
vita@baisheng:/vita/cross-tool/bin$ chmod a+x ldd
```

使用 ldd 脚本查看 bash 依赖的动态库：

```
vita@baisheng:/vita$ ldd rootfs/bin/bash
    libdl.so.2 => /vita/sysroot/lib/libdl.so.2
    libgcc_s.so.1 =>
        /vita/cross-tool/i686-none-linux-gnu/lib/libgcc_s.so.1
    libc.so.6 => /vita/sysroot/lib/libc.so.6
```

根据脚本 ldd 的输出可见，bash 依赖动态库 libdl、libc 和 libgcc_s.so.1，而这几个库都包含在 C 库中，我们都已经安装了。

在 3.3.7 节我们看到，如果用户没有通过内核命令行参数“init”指定第一个进程运行的用户空间的程序，则内核依次尝试执行目录 /sbin、/etc、/bin 下的 init，最后尝试执行目录 /bin 下的 sh。因此，我们在目录 /bin 下建立一个指向 bash 的符号链接 sh，而且，这个符号链接也是 FHS 标准要求的。

```
vita@baisheng:/vita/rootfs/bin$ ln -s bash sh
```

3.4.4 安装根文件系统到目标系统

接下来，我们需要将文件系统安装到虚拟机上，来配合内核进行启动。

当然，如果为了减少共享库和二进制可执行文件的大小，可以使用 i686-none-linux-gnu-strip 命令删除 ELF 中运行时不需要的符号，命令如下：

```
vita@baisheng:/vita$ i686-none-linux-gnu-strip rootfs/lib/* \
rootfs/bin/*
```

但是一定不要对 crt*.o 等这些启动文件进行 strip，因为这样会删除目标文件的符号表，导致链接器在链接时找不到符号。

接下来我们使用 scp 命令，将文件系统复制到虚拟机上。因为命令 scp 会跟随符号链接，因此，我们首先将文件系统打包，然后再使用 scp 命令进行复制：

```
vita@baisheng:/vita/rootfs$ tar zcvf ..../rootfs.tgz *
vita@baisheng:/vita/rootfs$ scp ..../rootfs.tgz \
root@192.168.56.101:/vita/
```

在复制完成后，在虚拟机上解开压缩包：

```
root@baisheng-vb:/vita# tar xvf rootfs.tgz
```

重启系统后，如果一切顺利，用户空间的程序 /bin/sh 会顺利运行，如图 3-28 所示。



```
11.10 [正在运行] - Oracle VM VirtualBox
ata3.00: ATAPI: VBOX CD-ROM, 1.0, max UDMA/133
ata3.00: configured for UDMA/33
ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
ata1.00: ATA-6: VBOX HARDDISK, 1.0, max UDMA/133
ata1.00: 16777216 sectors, multi 128: LBA48 NCQ (depth 31/32)
ata1.00: configured for UDMA/133
scsi 0:0:0:0: Direct-Access      ATA      VBOX HARDDISK    1.0 PQ: 0 ANSI: 5
scsi 2:0:0:0: CD-ROM            VBOX   CD-ROM        1.0 PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 16777216 512-byte logical blocks: (8.58 GB/8.00 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO
or FUA
sda: sda1 sda2
sd 0:0:0:0: [sda] Attached SCSI disk
EXT4-fs (sda2): couldn't mount as ext3 due to feature incompatibilities
EXT4-fs (sda2): couldn't mount as ext2 due to feature incompatibilities
EXT4-fs (sda2): mounted filesystem with ordered data mode. Opts: (null)
UFS: Mounted root (ext4 filesystem) readonly on device 8:2.
Freeing unused kernel memory: 336k freed
sh: cannot set terminal process group (-1): Inappropriate ioctl for device
sh: no job control in this shell
sh-4.2# tsc# Refined TSC clocksource calibration: 2370.132 MHz
Switching to clocksource tsc
sh-4.2# _
```

图 3-28 系统启动后进入 shell

至此，一个基本的内核已经构建完成了。它可以运行在 x86 体系架构上，可以驱动 Intel 的 SATA 硬盘，可以识别 EXT 系列文件系统，并内置 ELF 文件加载器，最后成功运行了用户空间的程序 bash。

当然，这仅仅是个开始，我们才刚刚上路。读者可以根据需要继续扩展内核功能，比如，后面为了支持网络，我们配置内核支持 TCP/IP 协议、配置内核支持网卡驱动等。但是，通过这一过程，我们也看到，从头开始编译一个内核并非如想象般困难。虽然内核包罗万象，支持不同的体系结构，有着成千上万的选项，包含数不清的驱动，这些都让内核看起来无比复杂，但是不要被这些表象迷惑，只要以目标为导向，再加上一点耐心，配置一个高效的内核不再是梦。