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

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

附光盘

机械工业出版社
China Machine Press

Inside the Linux Operating System

一直以来，学习Linux操作系统原理都被认为是一件非常复杂的事情，于是这些年来，市场上出版了很多探讨Linux操作系统原理的著作，它们都试图把操作系统的原理讲清楚，其中确实不乏经典之作。如果我们仔细分析一下这些著作，都无一例外地是在尝试着从阅读Linux内核源码的角度着手，以至于让我们认为内核就是作操作系统。

但是大量Linux操作系统学习者的经历告诉我们，通过阅读Linux内核源码来学习Linux操作系统原理的方式，并不见得是最理想的方式。如果将操作系统比作一个大象，那么只从阅读内核源码入手犹如“盲人摸象”，我们在尚未了解完整的大象，即Linux操作系统各个部分的作用以及相关间的协作关系时，就一头抱住了大象的腿开始钻研，其效果可想而知。所以，我们大多都有这样的体会：“看（内核源代码）的时候似乎明白，但是看完后似乎又什么都没有看。”

本书的作者在探索Linux操作系统的过程中也有过这样的经历，深知这种学习方式的弊病，他在一次从零开始构建一个完整的Linux操作系统的过程中领悟到了学习Linux操作系统原理的一种全新的方式：实践和理论有机结合，而不应仅仅停留在阅读内核源码上。因此，本书从一个全新的视角，以实践为导向，以从零开始构建一个完整的Linux操作系统为依托，带领读者从宏观上厘清构成操作系统的各个组件以及它们之间的关系，形成对操作系统的整体认识，并伴随整个构建过程探索各个组件的本质。与所有探讨Linux操作系统原理的书相比，本书在这一点上可谓独树一帜，奠定了其在该领域里程碑的地位。

附光盘

客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
投稿热线：(010) 88379604

数字阅读：www.hzmedia.com.cn
华章网站：www.h z b o o k .com
网上购书：www.china-pub.com

上架指导：计算机/操作系统
ISBN 978-7-111-43901-1
9 787111 439011 >
定价：89.00元（附光盘）

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

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

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。

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
北京市荣盛彩色印刷有限公司印刷
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）

华章计算机
HZBOOKS | Computer Science and Technology

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。

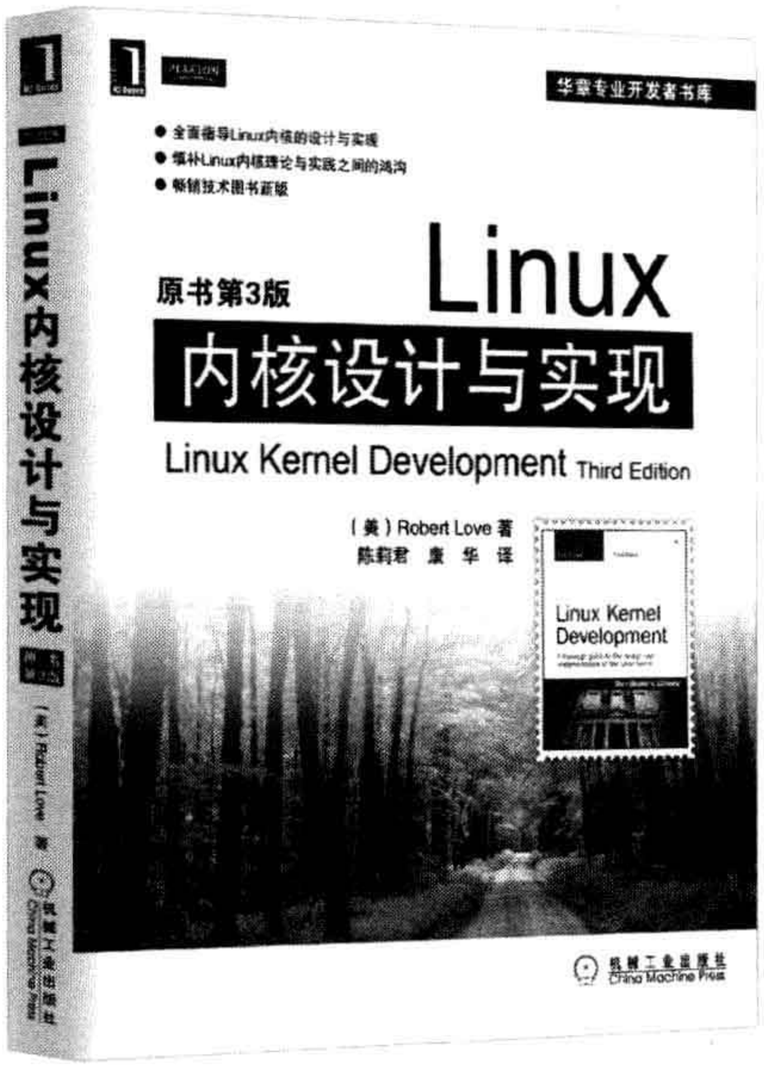
感谢机械工业出版社华章公司的姜影编辑，她清晰的思路让我深深折服。每每在遇到困惑不知如何表达时，她都能通过简单的几句话点醒梦中人。

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

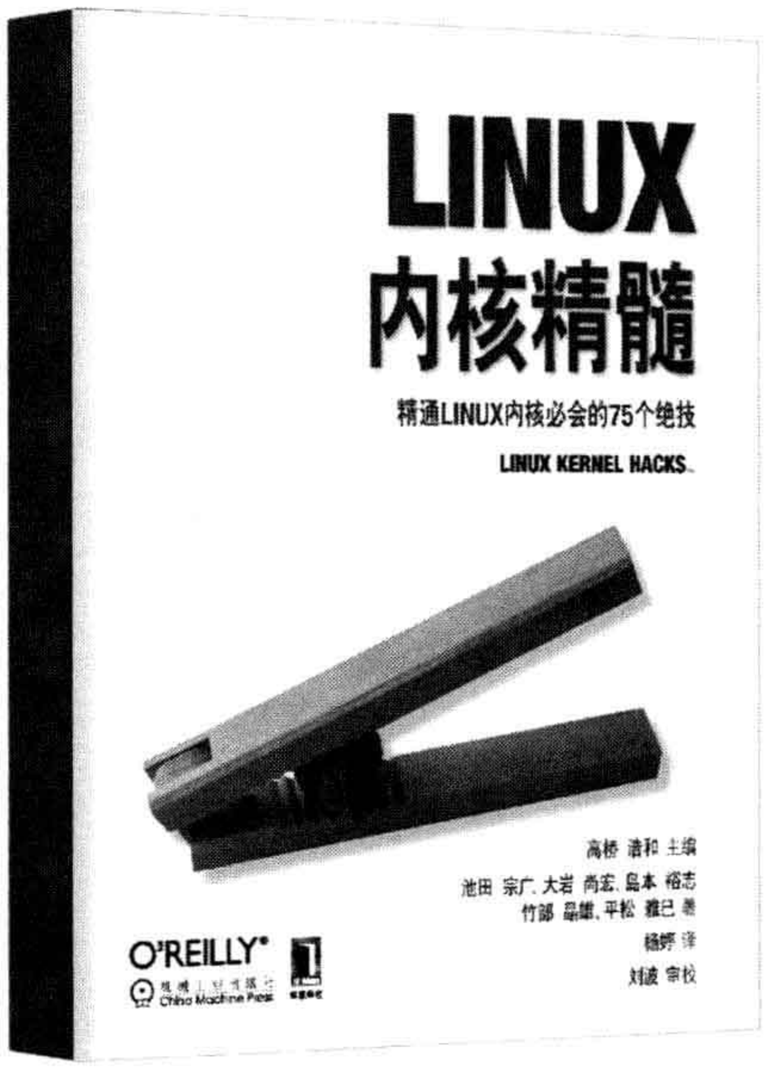
北京

推荐阅读



Linux内核设计与实现（原书第3版）

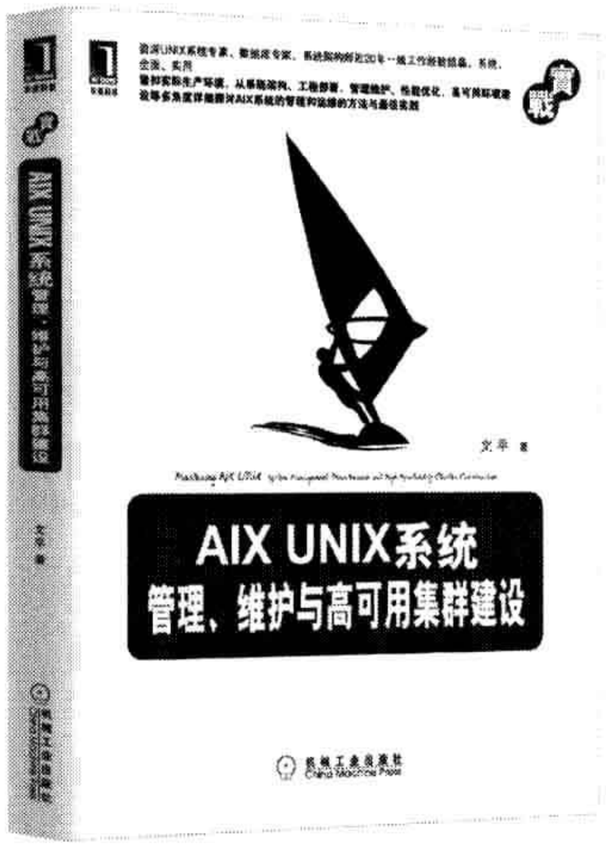
世界范围内公认的Linux内核经典著作，畅销全球多个国家

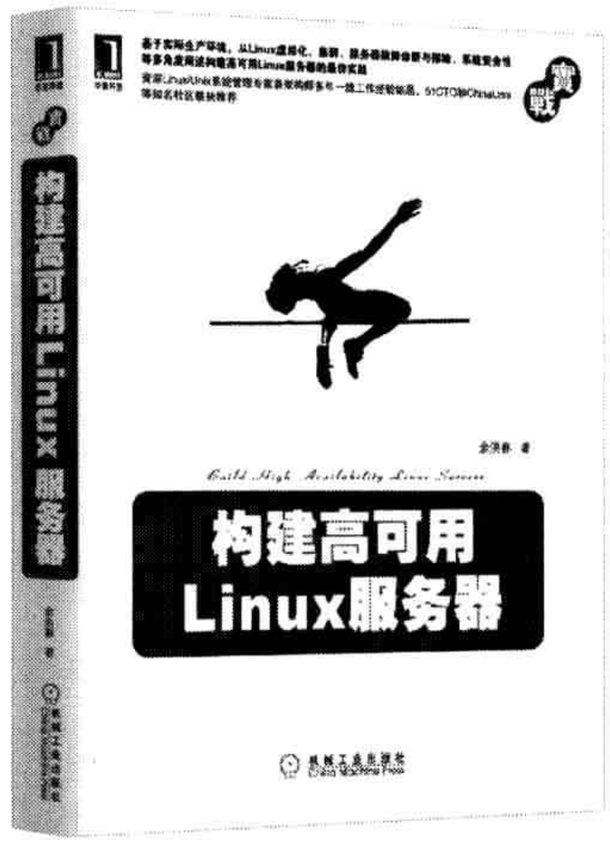


Linux内核精髓

畅销书，一线内核技术专家经验和智慧结晶，深刻解读Linux内核的资源管理、文件系统、网络、虚拟化、省电技术、调试、性能调优、分析与追踪等核心主题

推荐阅读





AIX UNIX系统：管理、维护与高可用集群建设

作者：文平 ISBN：978-7-111-35951-7 定价：79.00元

资深UNIX系统专家、数据库专家、
系统架构师近20年一线工作经验结晶，系统、全面、实用

紧扣实际生产环境，从系统架构、工程部署、管理维护、性能优化、
高可用环境建设等多角度详细探讨AIX系统的管理和运维的方法与最佳实践

构建高可用Linux服务器

作者：余洪春 ISBN：978-7-111-36055-1 定价：79.00元

基于实际生产环境，从Linux虚拟化、集群、服务器故障诊断与排除、
系统安全性等多角度阐述构建高可用Linux服务器的最佳实践

资深Linux/Unix系统管理专家兼架构师多年一线工作经验结晶，
51CTO和ChinaUnix等知名社区联袂推荐

目　录

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。

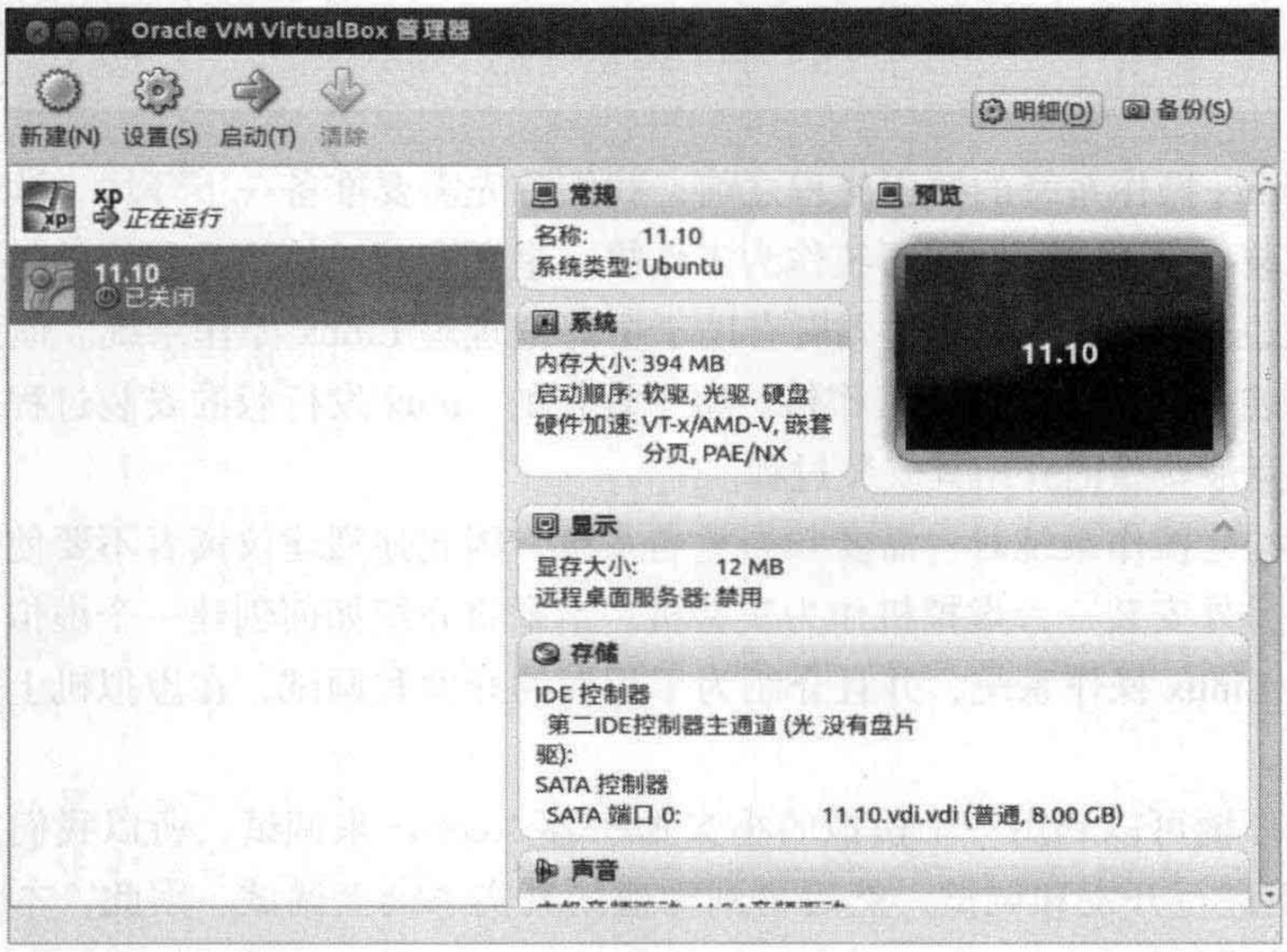


图1-1 VirtualBox主界面

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。

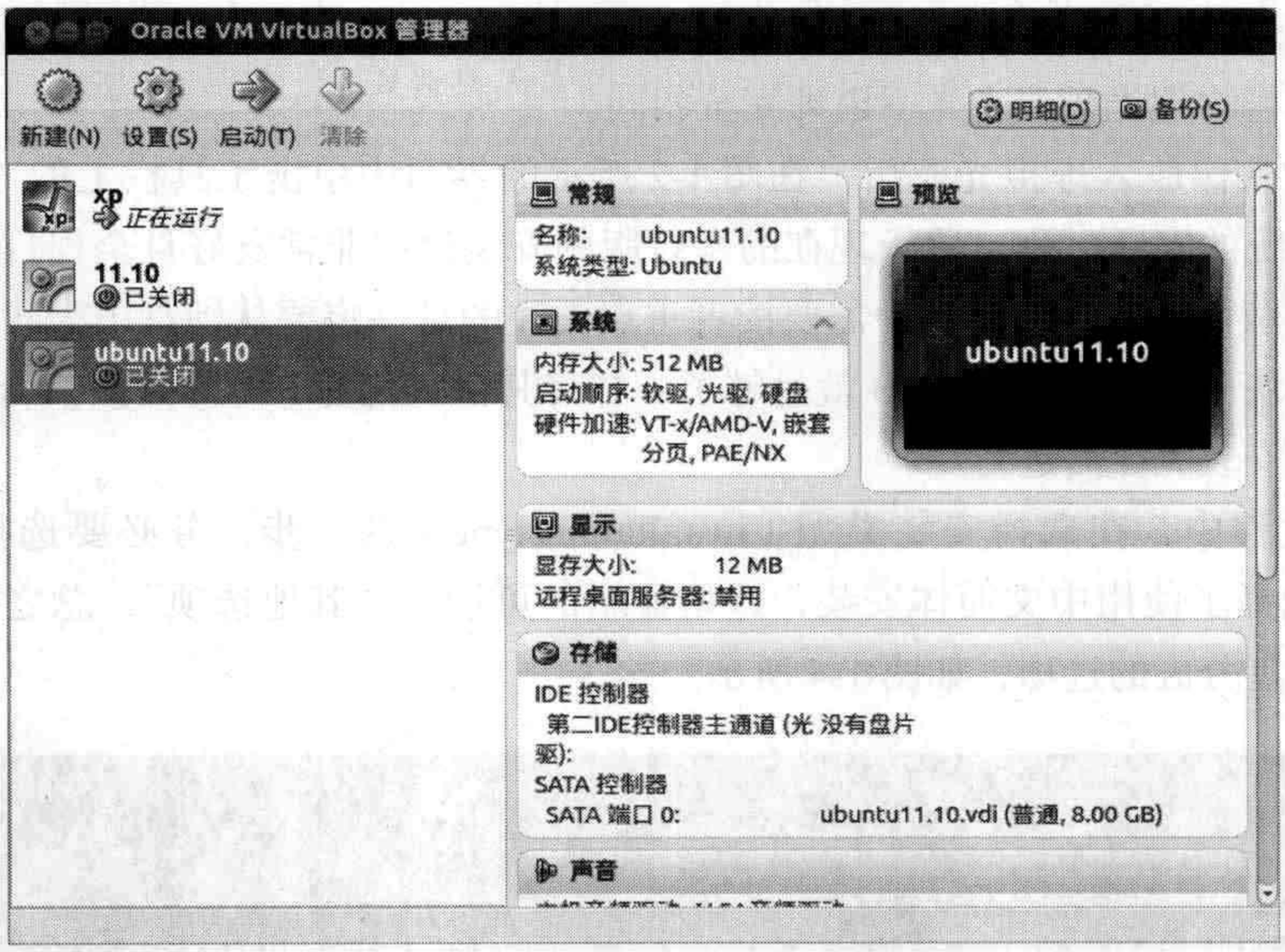


图 1-2 新建的虚拟裸机

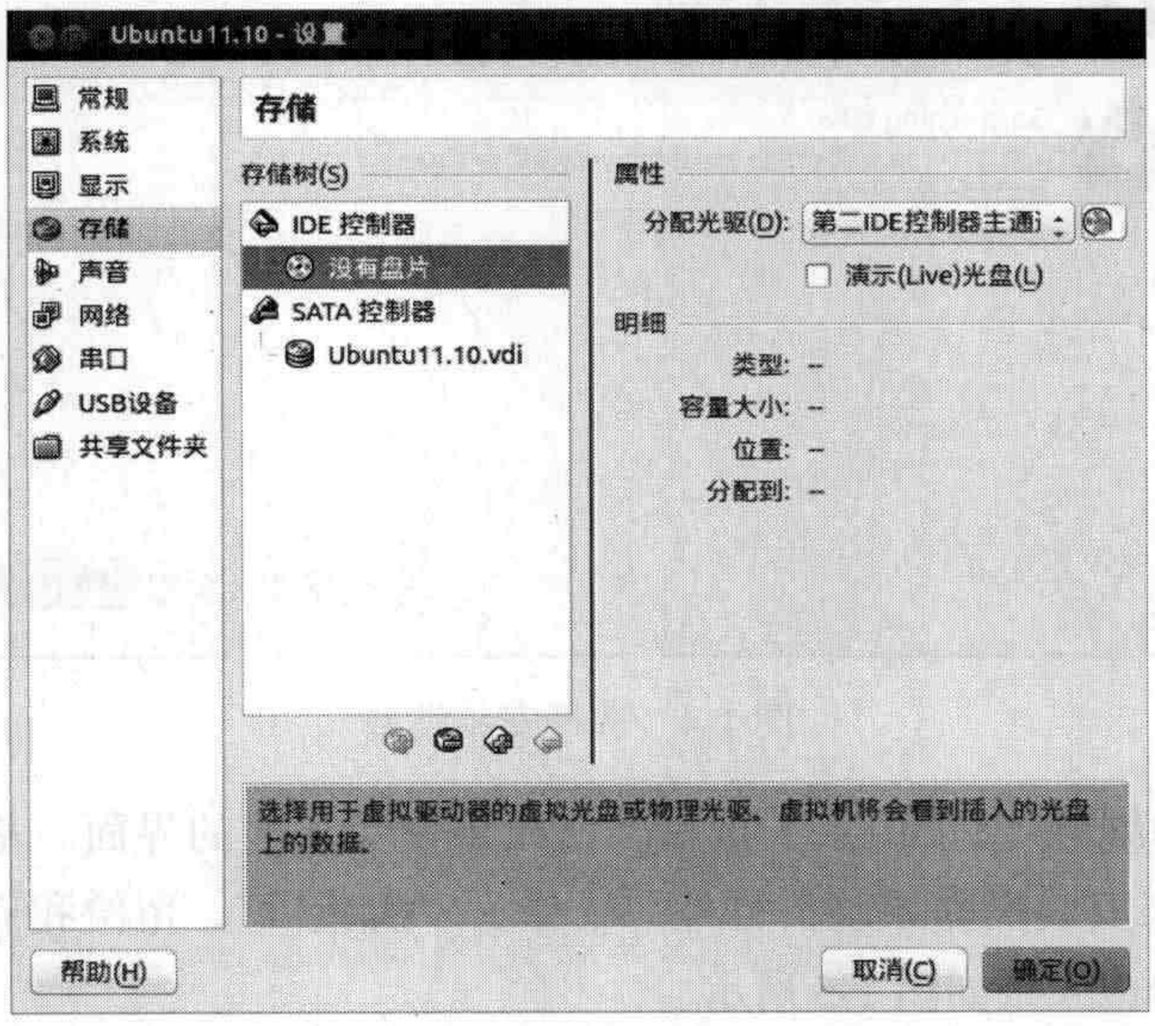


图 1-3 载入虚拟光盘映像

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。

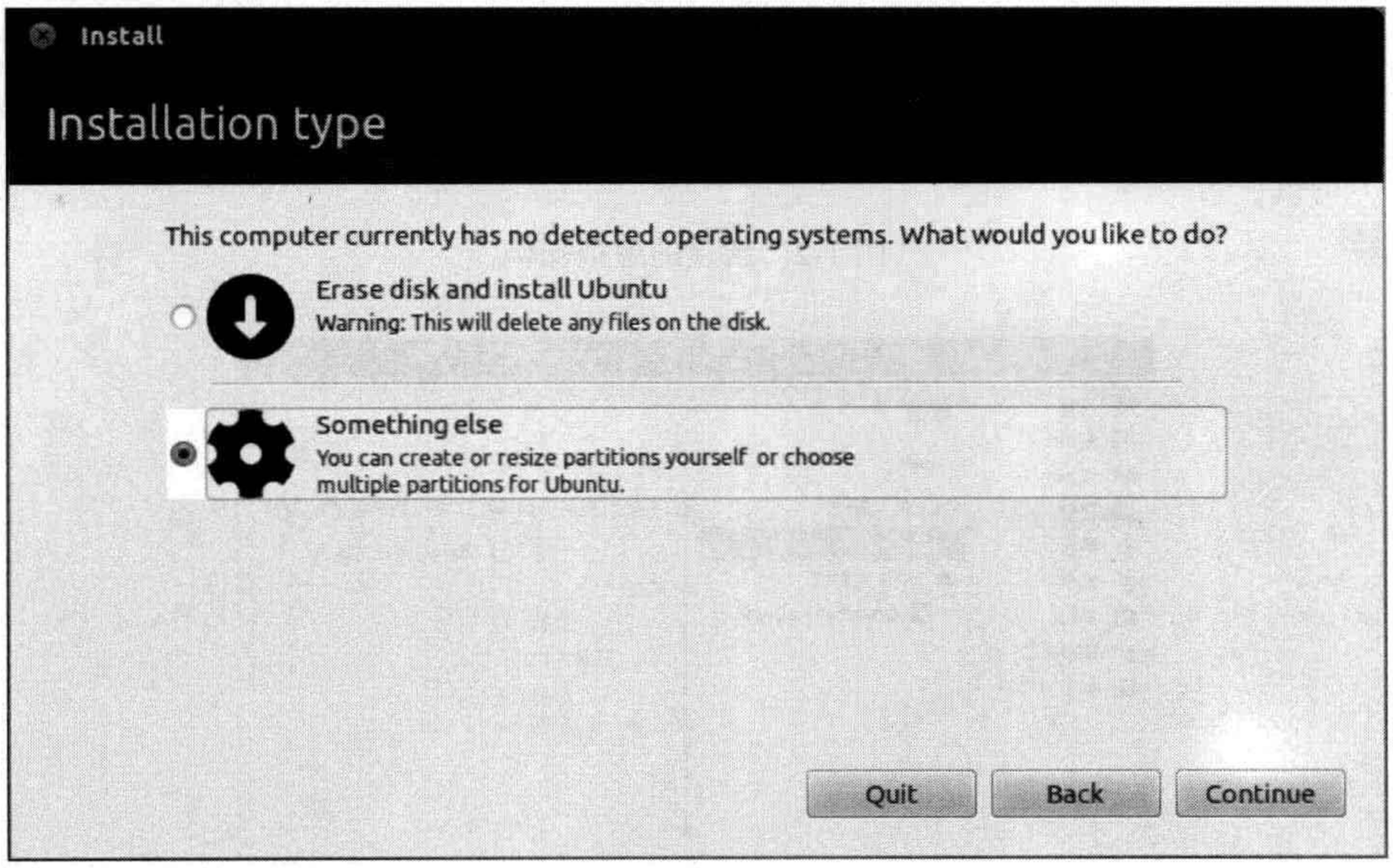


图 1-4 选择安装类型

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。

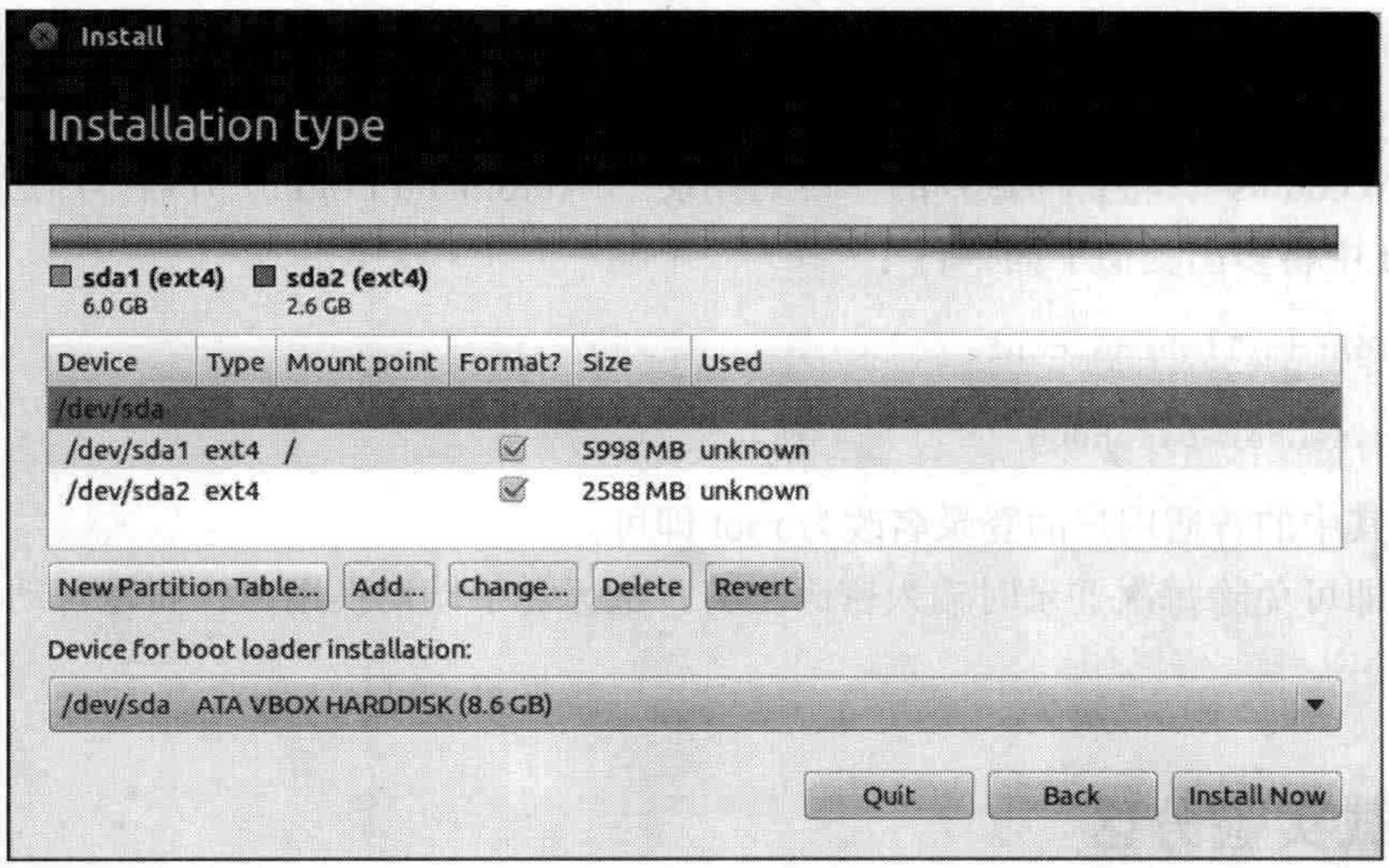


图 1-5 硬盘分区

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。另外一个问题是，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。

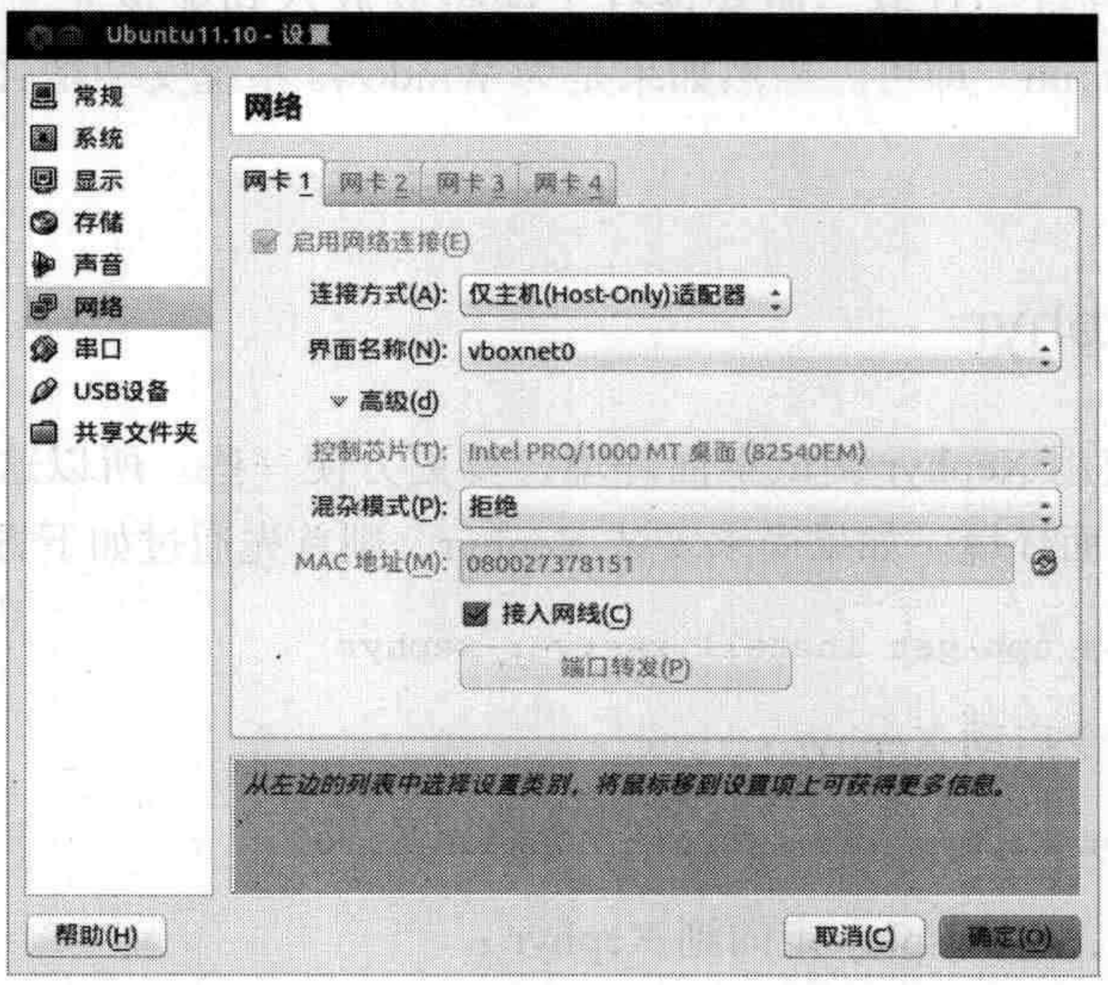


图 1-6 设置虚拟机网络模式

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。

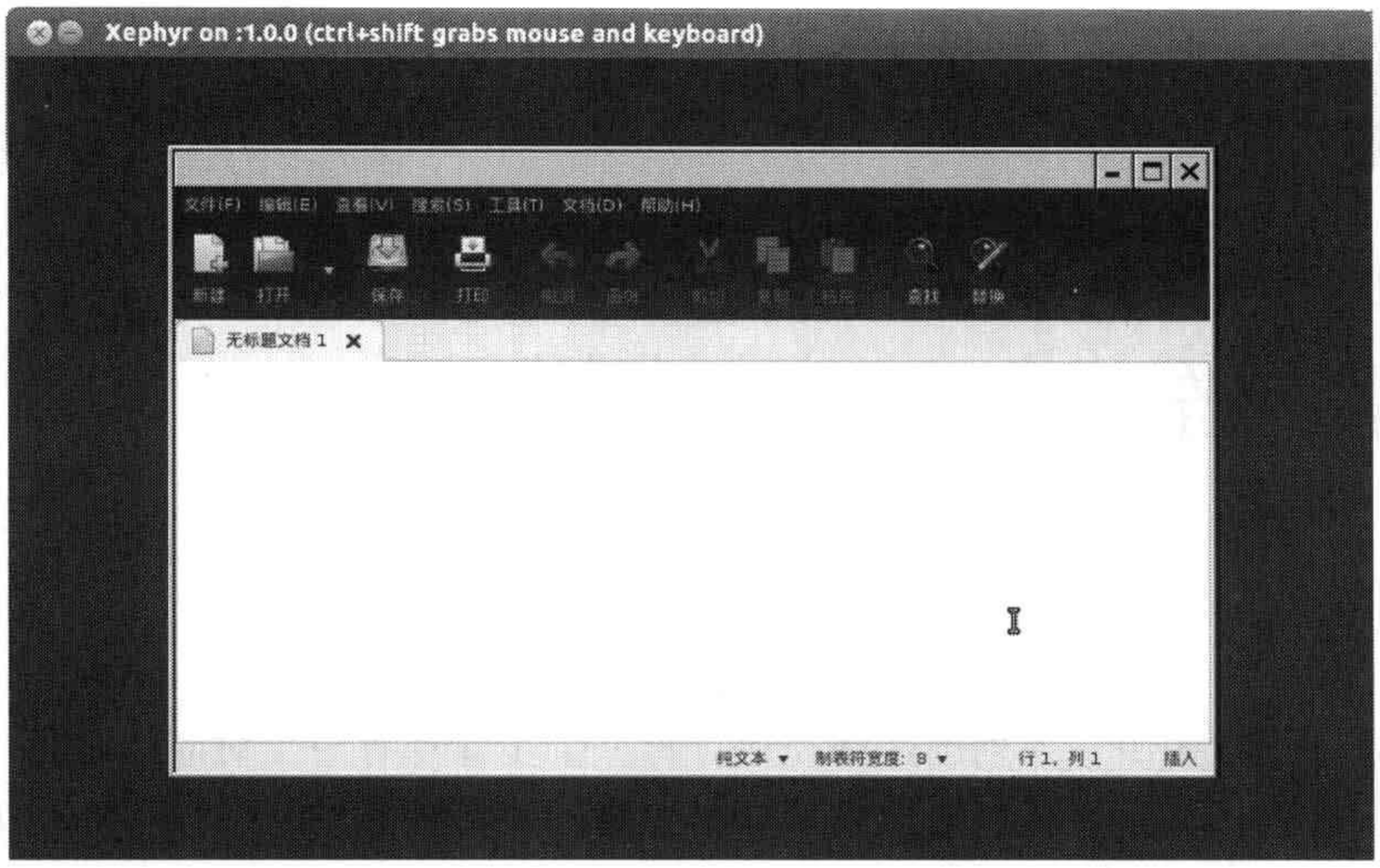

Xephyr on :1.0.0 (ctrl+shift grabs mouse and keyboard)
文件(F) 编辑(E) 查看(V) 搜索(S) 工具(T) 文档(D) 帮助(H)
新建 打开 保存 打印 撤销 重做 剪切 复制 粘贴 查找 替换
无标题文档 1
纯文本 制表符宽度: 8 行 1，列 1 插入

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

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/crt1.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/../../../../lib
-L/lib/i386-linux-gnu -L/lib/../lib -L/usr/lib/i386-linux-gnu
-L/usr/lib/../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 foo1 = 10;

void foo1_func()
{
```

```
    int ret = foo1;
}

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;
}
```

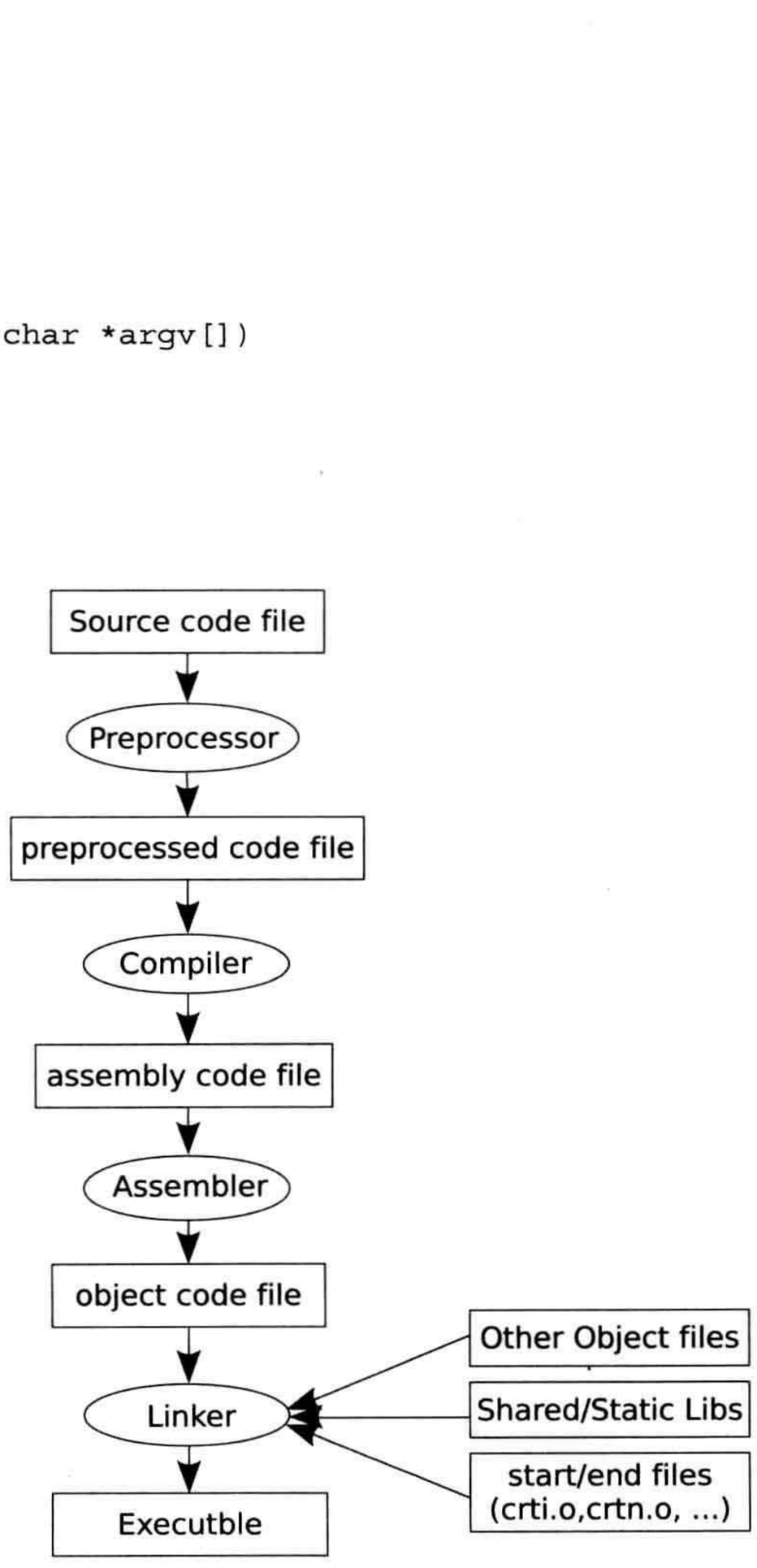


图 2-1 C 程序的构建过程

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“#incude”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 均指向栈桢的底部，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（该函数实现在启动代码中），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：

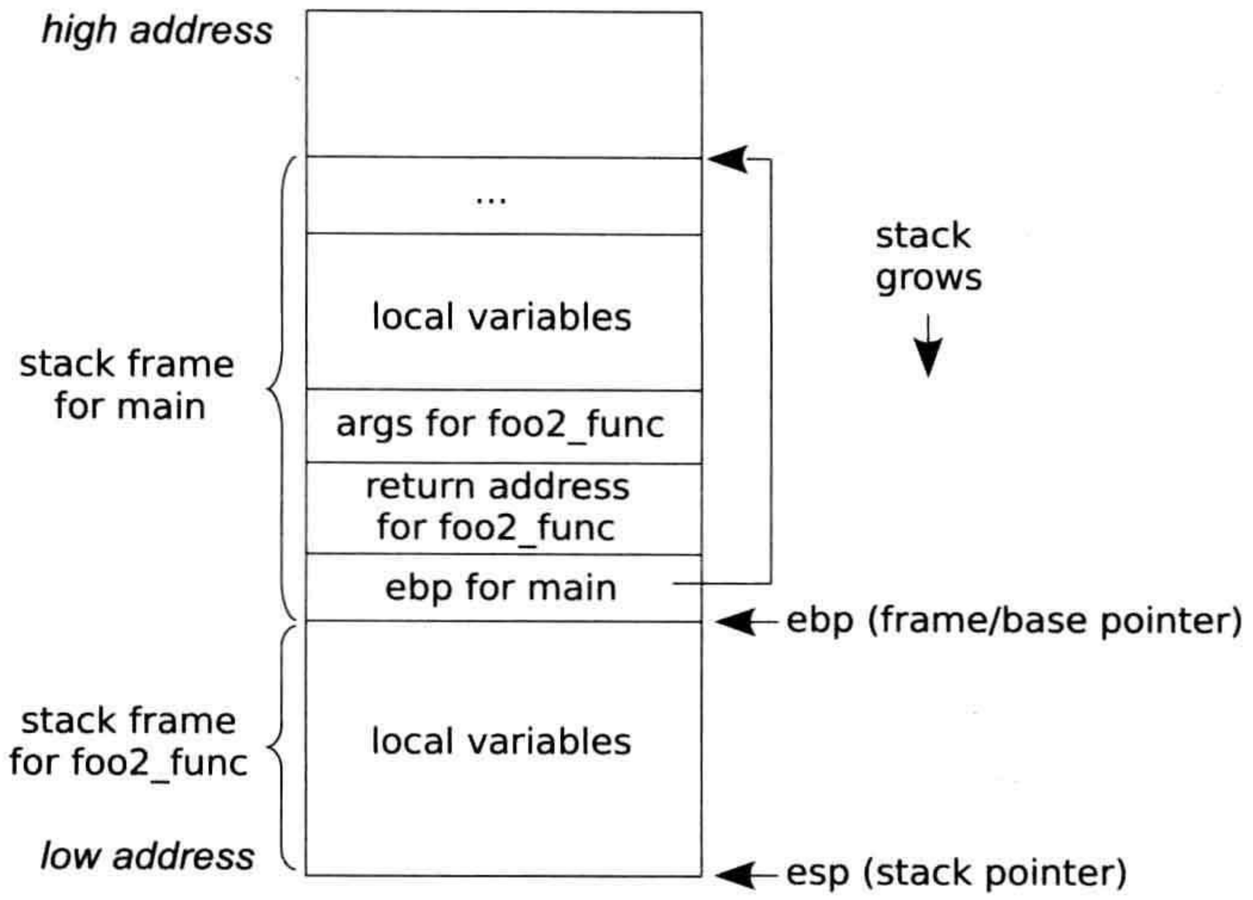


图 2-2 函数调用中的栈帧

```
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，指向当前栈桢的栈底，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 001024 000004
  [26] .comment          PROGBITS        00000000 001024 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。

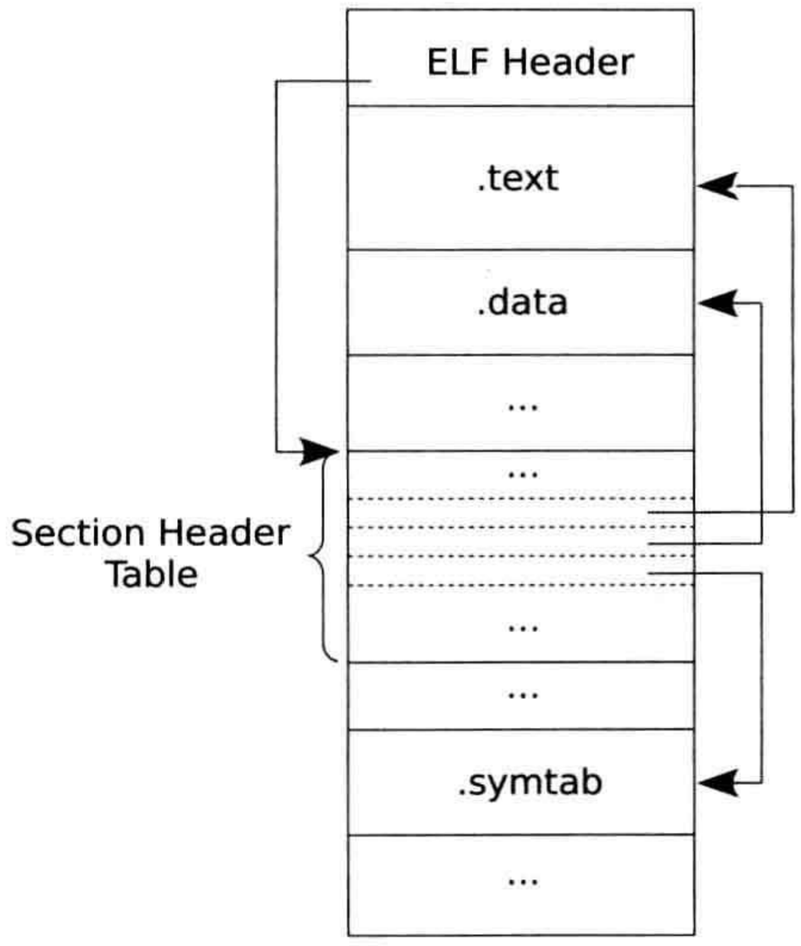


图 2-3 目标文件

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。

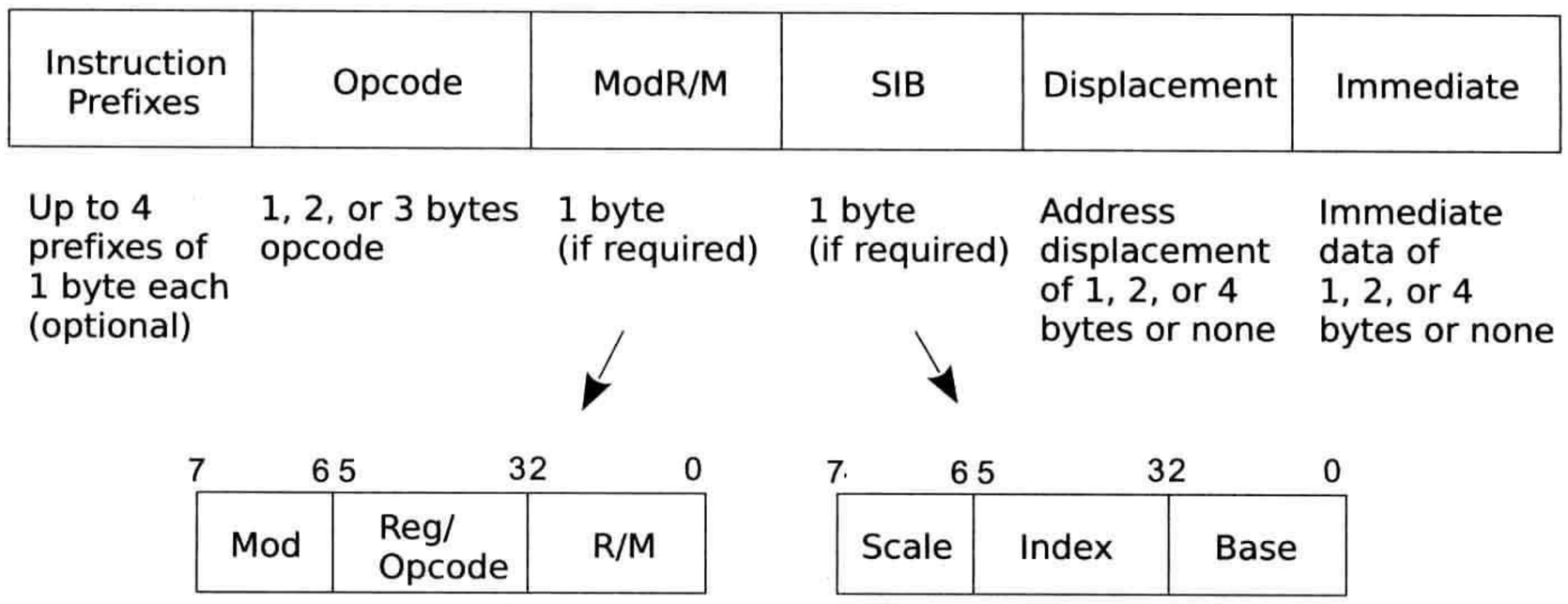


图 2-4 IA32 机器指令的格式

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，没有关系，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	01	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		000
16	…	11	…
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 00 34 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。下面我们将会看到这个 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了使用了一点后者。

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    $0xfffffff0,%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“fcffffff”，因为 IA32 使用的是 little-endian 字节序，补码“fffffffc”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。

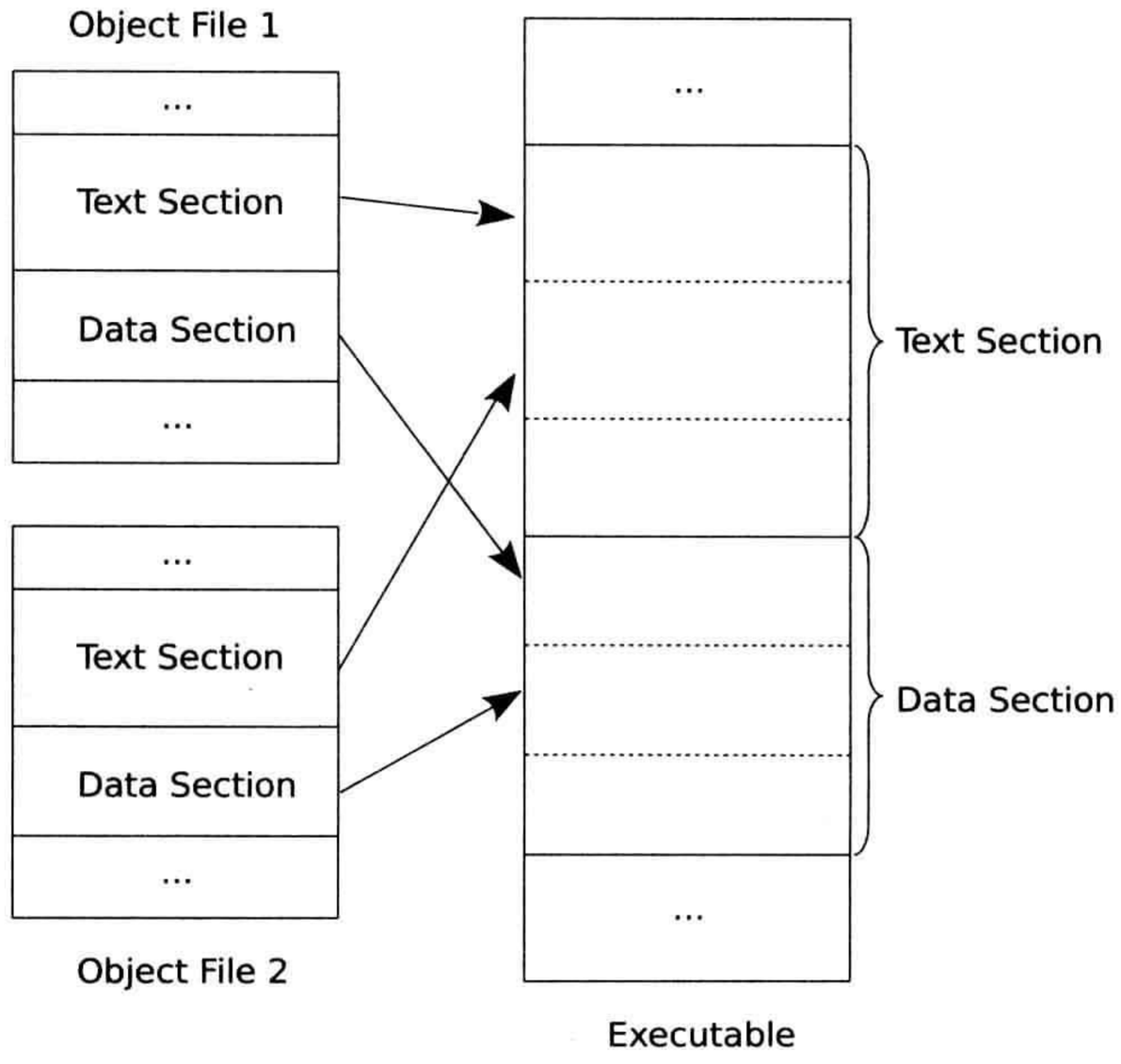


图 2-5 合并目标文件

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，其实就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            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    $0xfffffff0,%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    $0xfffffff0,%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                   xchg   %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 = 0x08048414 + (-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：

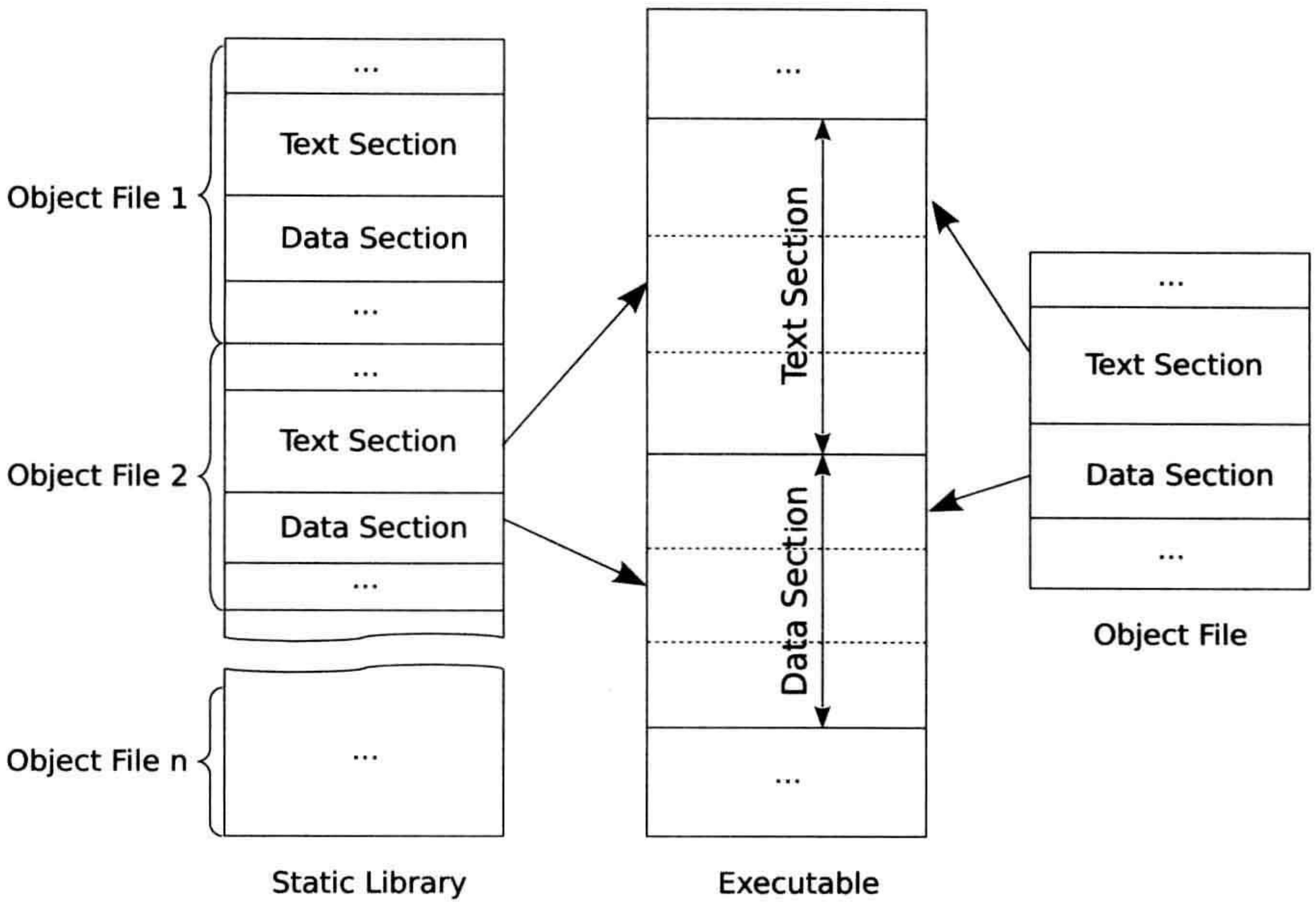


图 2-6 链接静态库

```
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_fiunc，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 loder/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 enviroment，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。

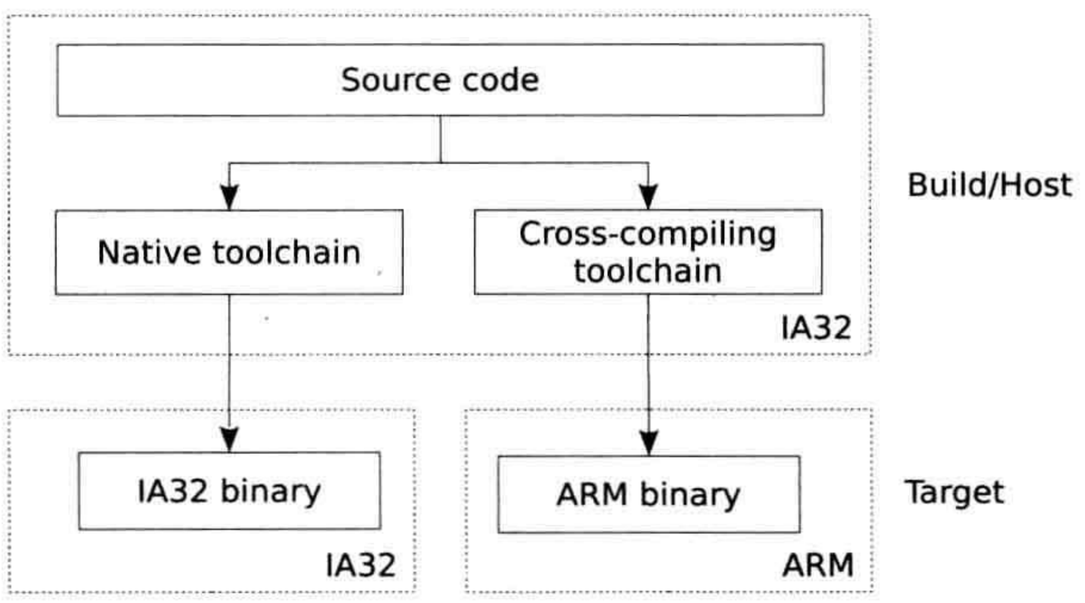


图 2-7 本地编译和交叉编译

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。但是如果没有 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(CRTSTUFFT_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-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.  */
#ifdef __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/headers_install
```

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 用来编译 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 使用了一些隐示的预定义变量，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
...
```

这些隐示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-confi
```

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/blkid
    -I/usr/include/uuid -lmount
```

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

```
vita@baisheng:~$ pkg-config --cflags --libs mount
    -I/vita/sysroot/usr/include/libmount
    -I/vita/sysroot/usr/include/blkid
    -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 $0xfffffff0, %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|funsafe-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:gcrt1.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 -shared -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。

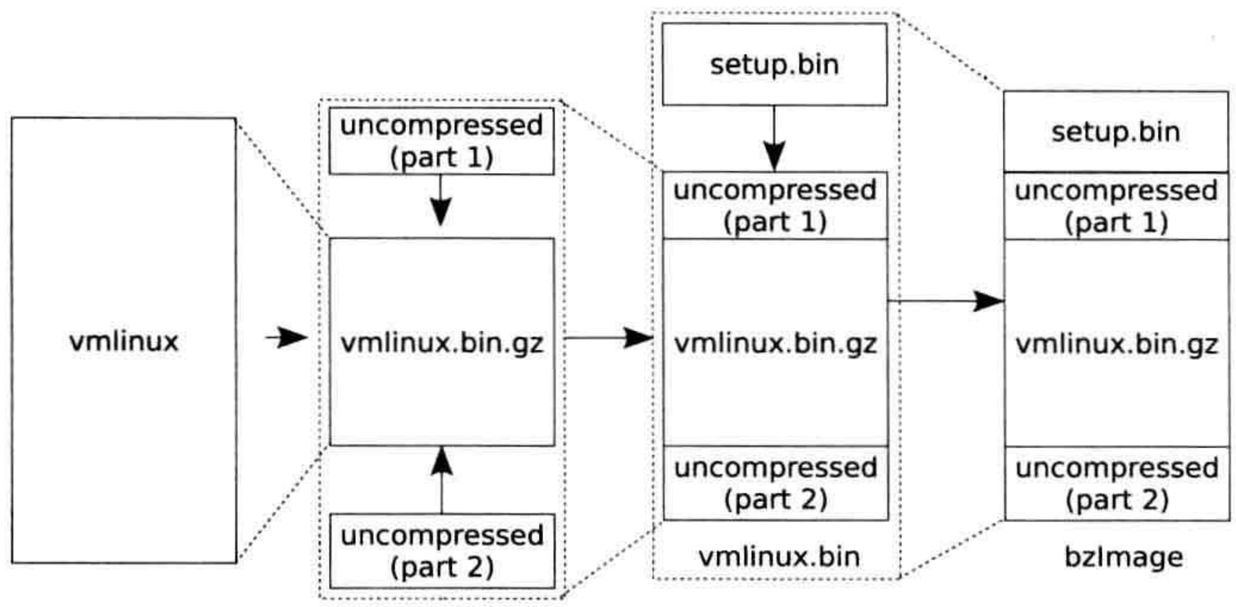


图 3-1 内核映像 bzImage 的组成

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，其占用的内存在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:

asmlinkage 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，kbuid 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 ${1}
    else
        ...
    fi
}
...
vmlinux_link "${kallsymso}" 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-dirs    := $(patsubst %/,%,$(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) $(KBUILD_VMLINUX_INIT)
                $(KBUILD_VMLINUX_MAIN)
...
$(sort $(vmlinux-deps)): $(vmlinux-dirs) ;
...
$(vmlinux-dirs): 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-dirs，我们继续跟踪目标 vmlinux-dirs 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，最终形成完整地 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-ym) $(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-ym 和 always。我们先来看 builtin-target 和 lib-target：

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

ifneq ($(strip $(lib-y) $(lib-m) $(lib-n) $(lib-)),)
lib-target := $(obj)/lib.a
endif

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

根据上述脚本片断可见，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 mkpiggy、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 builtin.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_RELOCS) += \
    $(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，如果不需要使用 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 cpucheck.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.efl 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，然后将他们链接为 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。

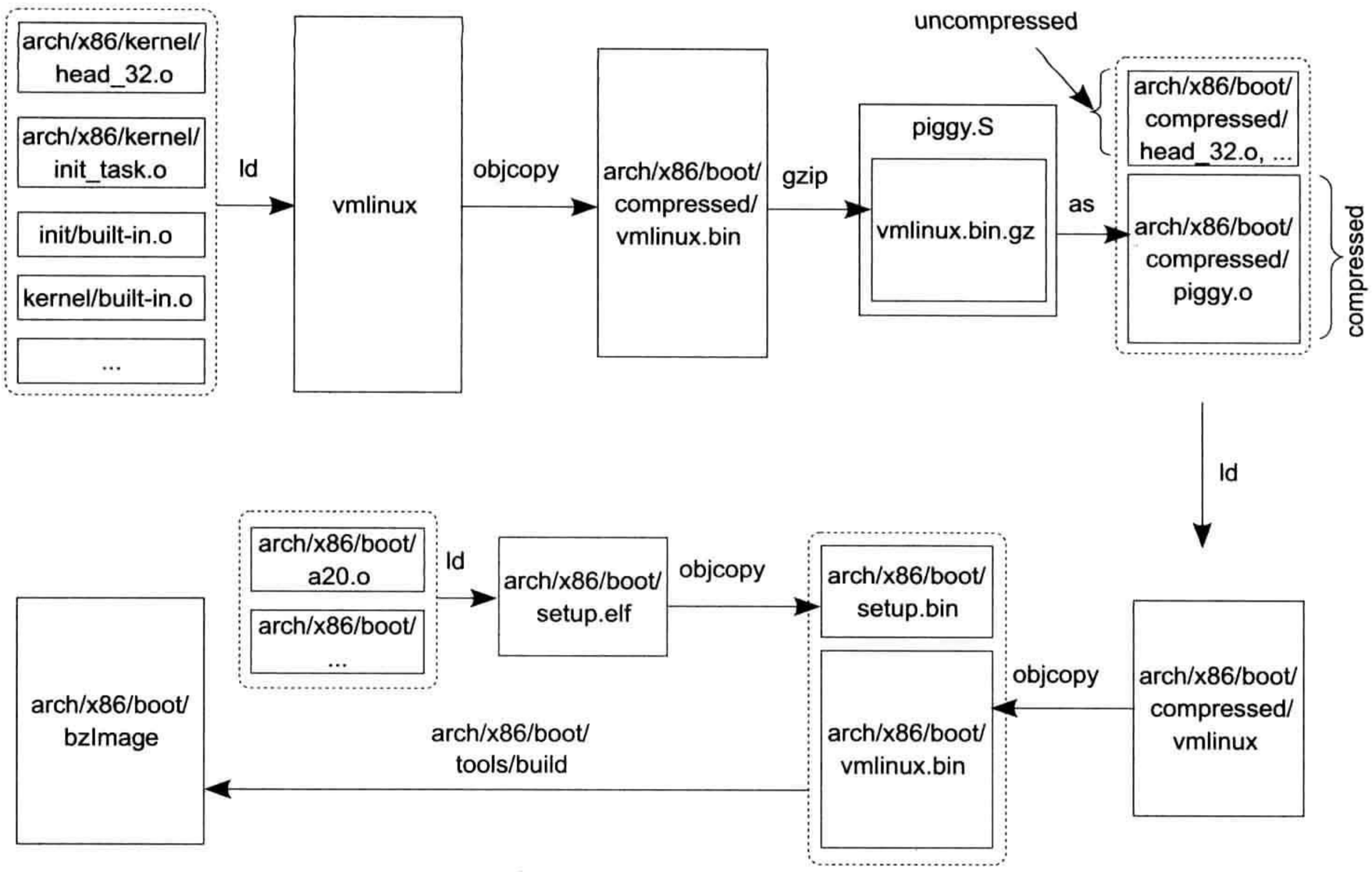


图 3-2　内核映像构建过程

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 = .;
        }
    #ifdef 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。

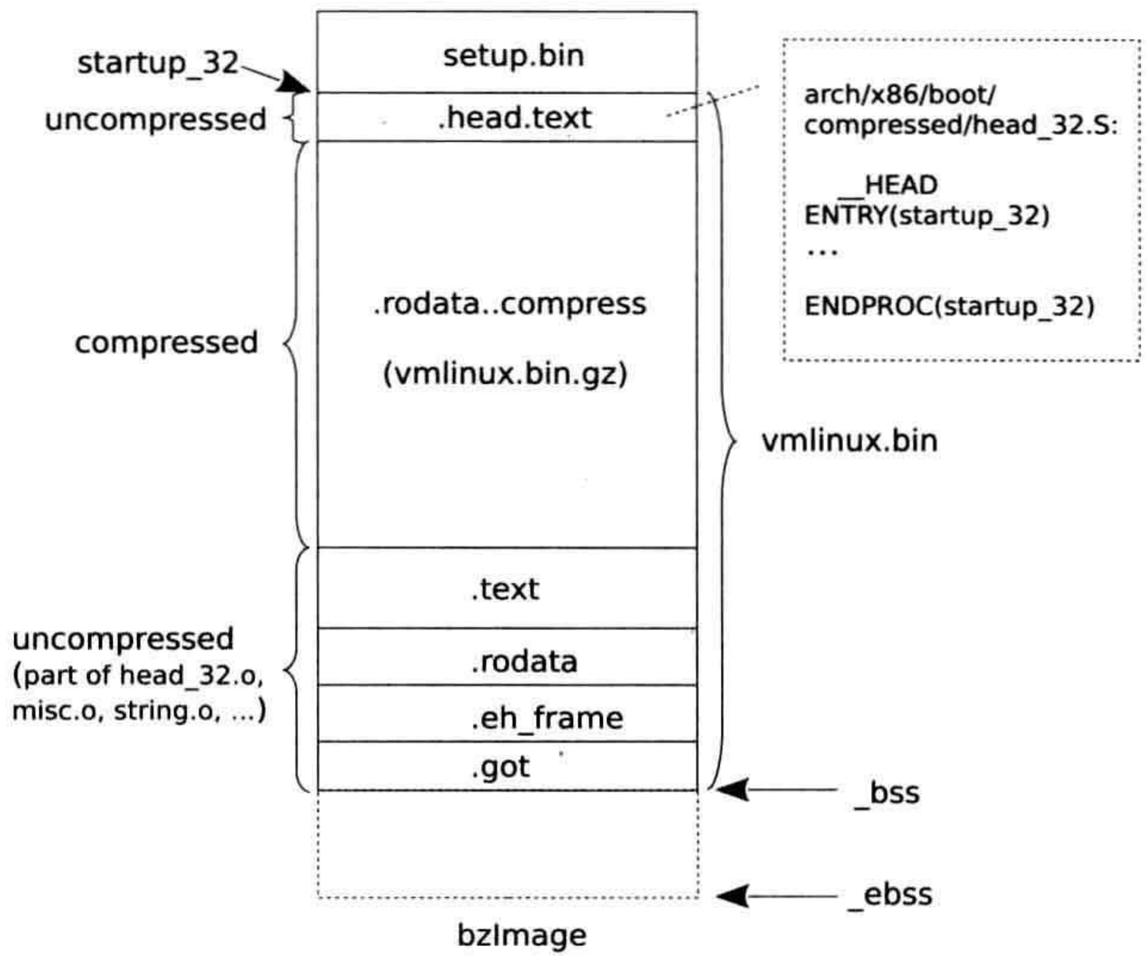


图 3-3 内核映像 bzImage 的布局

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 NVIDA 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了，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。

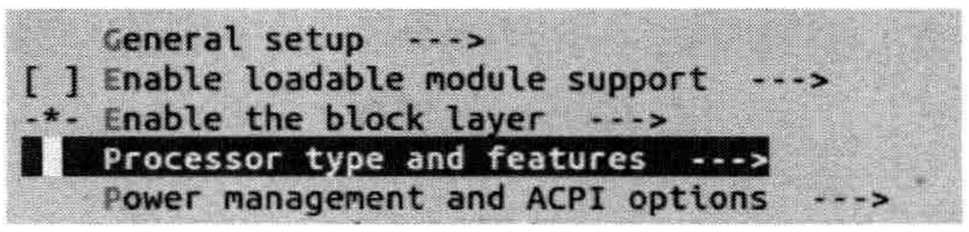


图 3-4 配置处理器型号（1）

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。

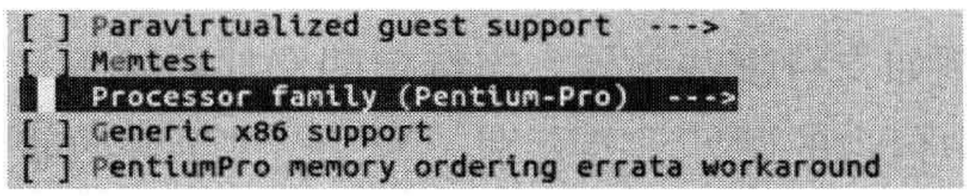


图 3-5 配置处理器型号（2）

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

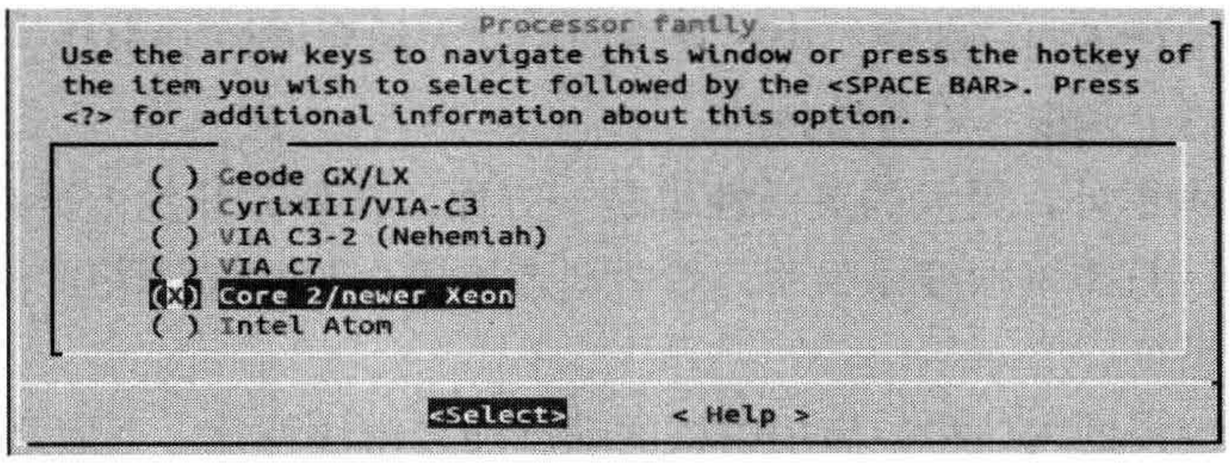


图 3-6 配置处理器型号（3）

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。

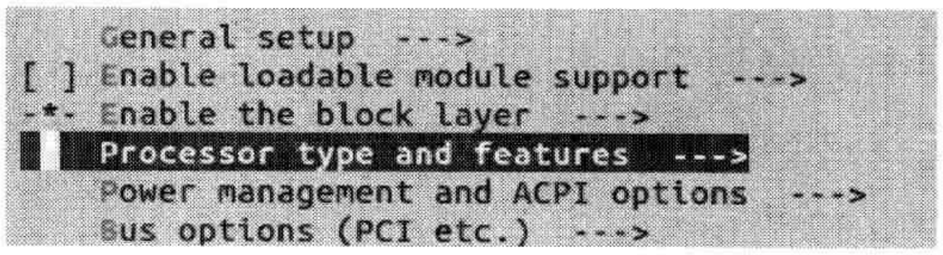


图 3-7 配置 SMP（1）

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。

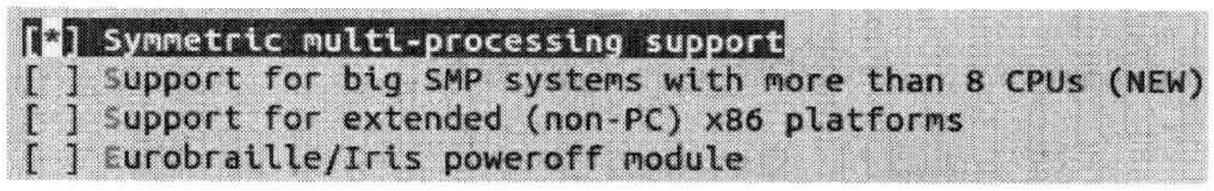


图 3-8 配置 SMP（2）

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。

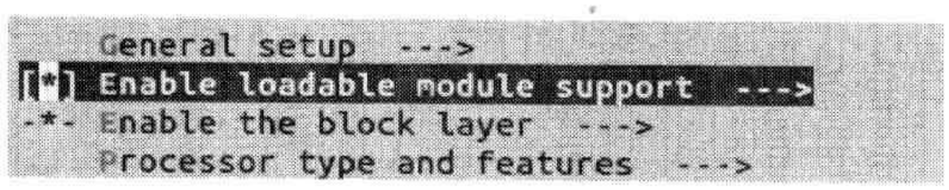


图 3-9 配置内核支持模块（1）

图 3-10 配置内核支持模块（2）

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。

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：

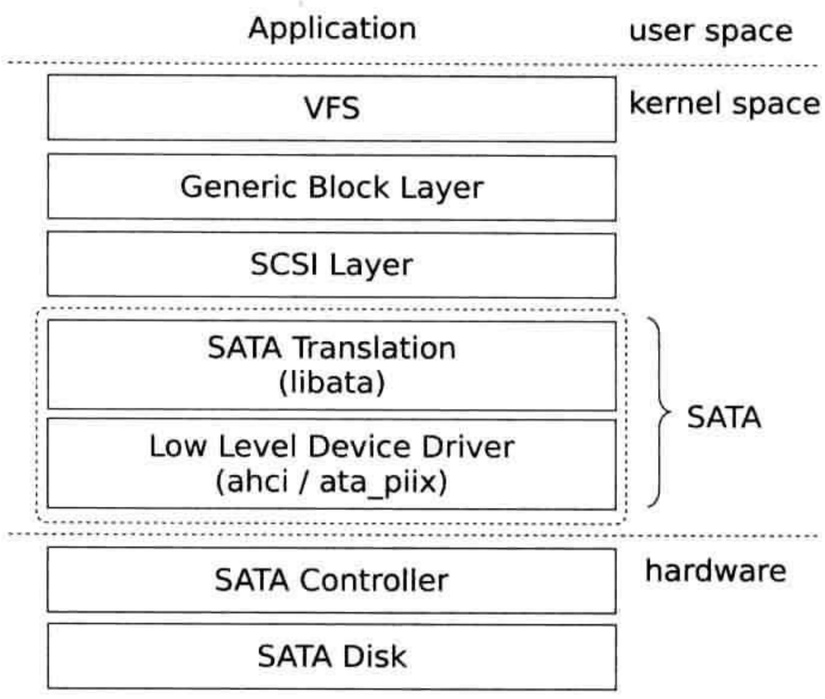


图 3-11 SATA 子系统结构

- 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。

```
    Processor type and features  --->
    Power management and ACPI options  --->
    Bus options (PCI etc.)  --->
    Executable file formats / Emulations  --->
```

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。

```
[*] PCI support
        PCI access mode (Any)  --->
[ ]   PCI Express support (NEW)
[ ] Enable PCI resource re-allocation detection (NEW)
```

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。

```
    Executable file formats / Emulations  --->
[ ] Networking support  --->
    Device Drivers  --->
    Firmware Drivers  --->
```

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。

```
[ ] Block devices  --->
    Misc devices  --->
< > ATA/ATAPI/MFM/RLL support (DEPRECATED)  --->
    SCSI device support  --->
< > Serial ATA and Parallel ATA drivers  --->
[ ] Multiple devices driver support (RAID and LVM)  --->
[ ] Fusion MPT device support  --->
```

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。

```
< > RAID Transport Class
<*> SCSI device support
[*] Legacy /proc/scsi/ support (NEW)
    *** SCSI support type (disk, tape, CD-ROM) ***
<*> SCSI disk support
< > SCSI tape support (NEW)
```

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。

```
    Executable file formats / Emulations  --->
[ ] Networking support  --->
    Device Drivers  --->
    Firmware Drivers  --->
```

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。

```
< > ATA/ATAPI/MFM/RLL support (DEPRECATED)  --->
    SCSI device support  --->
<*> Serial ATA and Parallel ATA drivers  --->
[ ] Multiple devices driver support (RAID and LVM)  --->
```

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。

```
<*>   AHCI SATA support
< >   Platform AHCI SATA support (NEW)
< >   Initio 162x SATA support (NEW)
< >   ACard AHCI variant (ATP 8620) (NEW)
< >   Silicon Image 3124/3132 SATA support (NEW)
[*]   ATA SFF support (for legacy IDE and PATA) (NEW)
        *** SFF controllers with custom DMA interface ***
< >     Pacific Digital ADMA support (NEW)
< >     Pacific Digital SATA QStor support (NEW)
[*]     ATA BMDMA support (NEW)
          *** SATA SFF controllers with BMDMA ***
<*>       Intel ESB, ICH, PIIX3, PIIX4 PATA/SATA support
```

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。

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。

```
    Device Drivers  --->
    Firmware Drivers  --->
    File systems  --->
    Kernel hacking  --->
```

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。

```
< > Ext3 journalling file system support
<*> The Extended 4 (ext4) filesystem
[*]   Use ext4 for ext2/ext3 file systems (NEW)
[*]   Ext4 extended attributes (NEW)
[ ]     Ext4 POSIX Access Control Lists (NEW)
[ ]     Ext4 Security Labels (NEW)
[ ]   EXT4 debugging support (NEW)
< > Reiserfs support
```

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。

```
    General setup  --->
[*] Enable loadable module support  --->
-*- Enable the block layer  --->
    Processor type and features  --->
```

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。

```
--- Enable the block layer
[*]   Support for large (2TB+) block devices and files
[ ]   Block layer SG support v4
[ ]   Block layer SG support v4 helper lib
```

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。

```
    Power management and ACPI options  --->
    Bus options (PCI etc.)  --->
    Executable file formats / Emulations  --->
[ ] Networking support  --->
```

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。

```
[*] Kernel support for ELF binaries
[*] Write ELF core dumps with partial segments (NEW)
< > Kernel support for a.out and ECOFF binaries
< > Kernel support for MISC binaries
```

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。

```
11.10 [正在运行] - Oracle VM VirtualBox
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): INFO: recovery required on readonly filesystem
EXT4-fs (sda2): write access will be enabled during recovery
EXT4-fs (sda2): recovery complete
EXT4-fs (sda2): mounted filesystem with ordered data mode. Opts: (null)
VFS: Mounted root (ext4 filesystem) readonly on device 8:2.
Freeing unused kernel memory: 336k freed
Kernel panic - not syncing: No init found.  Try passing init= option to kernel.
See Linux Documentation/init.txt for guidance.
Pid: 1, comm: swapper/0 Not tainted 3.7.4 #30
Call Trace:
 [<c11ed8ab>] ? panic+0x7d/0x158
 [<c11e7138>] ? kernel_init+0x238/0x270
 [<c12a93c2>] ? do_early_param+0x77/0x77
 [<c11f2377>] ? ret_from_kernel_thread+0x1b/0x28
 [<c11e6f00>] ? rest_init+0x60/0x60
                                                          右 Ctrl
```

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
/bin	保存系统管理员与用户均会使用的重要的命令
/boot	系统开机使用的文件，如内核映像和 boot loader 的相关文件
/dev	设备文件
/etc	系统配置
/lib	重要的库文件及内核模块
/media	可移动存储介质的挂载点
/mnt	临时挂载点，当然用户也可以自行选择一些临时挂载点
/opt	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 /usr/local 目录下
/sbin	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%s => %s\n" "" $1 $found
else
    printf "%8s%s => (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。

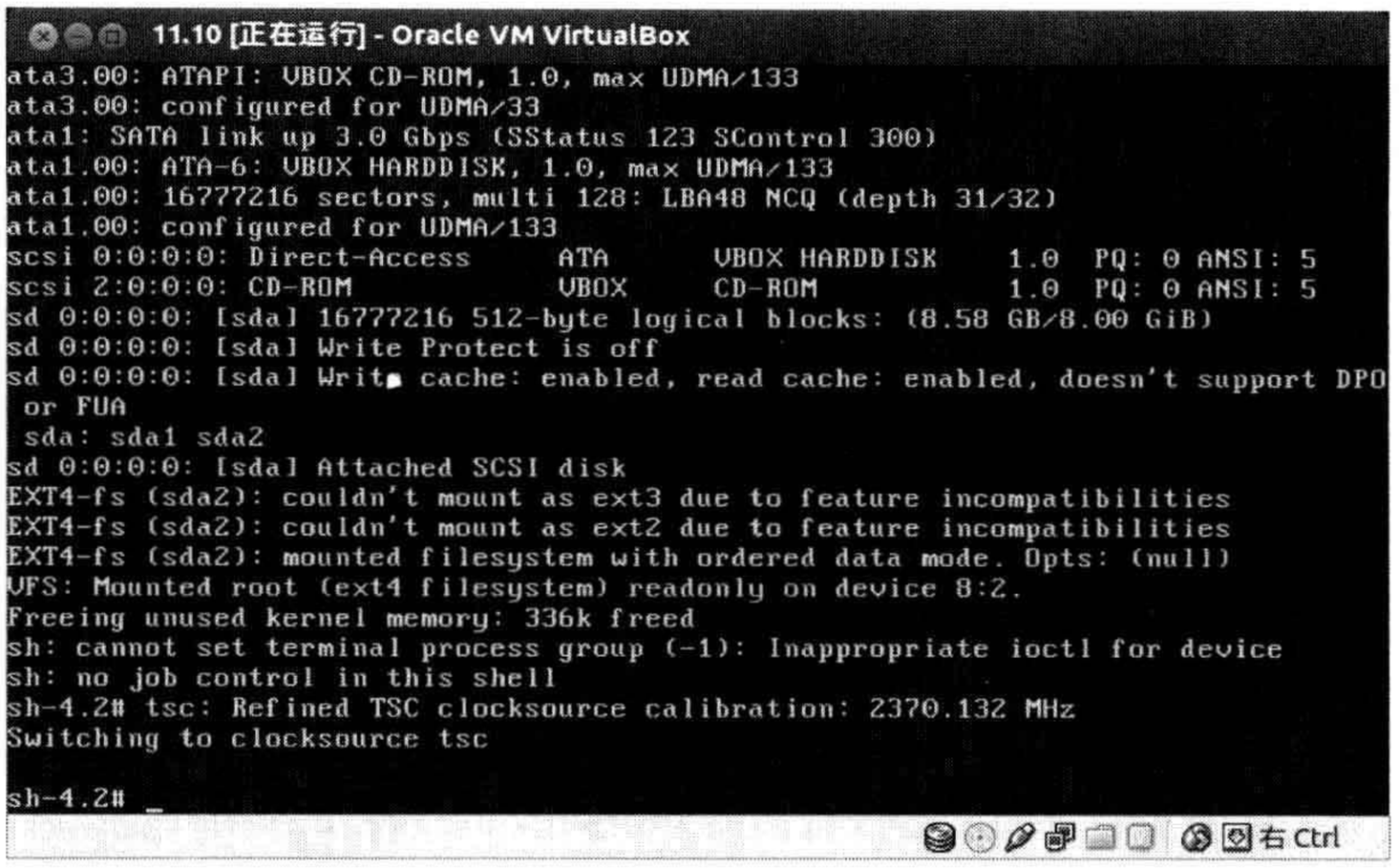


图 3-28 系统启动后进入 shell

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。

第 4 章

构建 initramfs

一般而言，桌面、服务器等通用系统都使用 initramfs。部分嵌入式系统中，也会使用 initramfs，甚至有的使用 initramfs 作为最终的根文件系统。那么什么是 initramfs 呢？很难用一句话将 initramfs 的作用描述清楚，或许可以将 initramfs 定位为内核通往根文件系统的桥梁。

但是，在上一章中我们看到，在没有使用 initramfs 的情况下，内核也已经成功挂载了根文件系统并进入了用户空间。因此，我们不禁会产生疑问：既然内核不用借助这座桥梁就能到达彼岸，为什么还要使用 initramfs 呢？是不是多此一举？

这一章就来回答这个问题。本章先后讲述了为什么要使用 initramfs；接着阐述了 initramfs 的工作原理；然后从零开始，构建了一个 initramfs，其中将讨论内核如何做到动态加载用户空间的驱动，如何让冷插拔（coldplug）设备也如热插拔（hotplug）设备一样，可以动态加载用户空间的驱动等等；最后讨论如何从 initramfs 切换到真正的根文件系统。

4.1 为什么需要 initramfs

在引导过程的最后，内核启动第一个用户进程，内核需要访问根文件系统，加载相应的可执行程序。这就要求内核能够正确驱动根文件系统所在设备。所以在上一章中，我们将 SATA 磁盘驱动编译进了内核，为的就是内核可以正确驱动硬盘。

寻找根文件系统的过程中，我们需要考虑以下两种情况。

第一，除非是一个专用系统，目标系统的硬件平台是固定不变的，否则，对于一个通用操作系统，比如 Linux 的桌面发行版，将运行在各种不同的硬件平台上。因此，根文件系统可能存储在各种各样的介质上，比如 IDE 硬盘、SATA 硬盘、SCSI 硬盘、Flash 存储器，以及随着技术的发展，不断出现的新的存储设备。为了能够兼容更多的硬件平台，显然系统需要支持尽可能多的存储设备。但是如果将所有这些设备的驱动全部编译进内核，显然不是一个好办法。因为对于某个特定的硬件平台，可能只需要一个驱动即可，内核中的其他驱动根本用不上，将它们编译进内核只会徒增内核的尺寸、占用内存空间，尤其对于一些内存或者存储介质空间有限的设备，这个问题尤为明显。于是将这些驱动编译为模块，存储在根文件

系统中，按需载入内存是一个解决问题的办法。

第二，根文件系统可能不在一个简单的硬盘上，比如当使用磁盘阵列 RAID 时，根文件系统可能横跨几个存储设备，或者根文件系统在某个网络设备上。以使用 NFS 挂载根文件系统为例，除了要支持网卡驱动外，还要进行网络配置，甚至还要进行网络认证。某些根文件系统经过压缩、加密，在挂载前需要解压缩、解密等操作。如果这些都由内核处理，将会使内核变得异常复杂，继而可能导致内核的稳定性、可靠性、灵活性等一系列的问题。因此，将复杂的操作移到用户空间是解决上述问题的一个思路。

但是，无论是将驱动编译为模块，还是将处理如 RAID 挂载的程序存储在文件系统上，都会导致一个鸡和蛋的问题：内核要加载这些模块或者运行这些程序才能正确识别根文件系统所在的设备，但是保存这些模块或者程序的根文件系统又存储在这些设备上。

为了解决上述问题，内核开发者们设计了 initramfs 机制。initramfs 是一个临时的文件系统，其中包含了必要的设备如硬盘、网卡、文件系统等的驱动以及加载驱动的工具及其运行环境，比如基本的 C 库，动态库的链接加载器等等。同时，那些处理根文件系统在 RAID、网络设备上的程序也存放在 initramfs 中。由第三方程序（如 Bootloader）负责将 initramfs 从硬盘装载进内存。以驱动硬盘为例，内核就不必再从硬盘，而是从已经加载到内存的 initramfs 中获取硬盘控制器等相关驱动了，继而可以驱动硬盘，访问硬盘上的根文件系统，从而解决了前面提到的鸡和蛋的矛盾。

在初始化的最后，内核运行 initramfs 中的 init 程序，该程序将探测硬件设备、加载驱动，挂载真正的文件系统，执行文件系统上的 /sbin/init，进而切换到真正的用户空间。真正的文件系统挂载后，initramfs 即完成了使命，其占用的内存也会被释放。

4.2 initramfs 原理探讨

在 2.4 以及更早版本的内核中，内核使用的是 initrd。initrd 是基于 ramdisk 技术的，而 ramdisk 就是一个基于内存的块设备，因此 initrd 也具有块设备的一切属性。比如 initrd 容量是固定的，一旦创建 initrd 时设定了一个大小，就不能再进行动态调整。而且，如同块设备一样，initrd 需要按照一定的文件系统格式进行组织，因此制作 initrd 时需要使用如 mke2fs 这样的工具“格式化”initrd，访问 initrd 时需要通过文件系统驱动。更重要的是，虽然 initrd 是一个伪块设备，但是从内核的角度看，其与真实的块设备并无区别，因此，内核访问 initrd 也需使用缓存机制，显然这是多此一举的，因为本身 initrd 就在内存中。

鉴于 ramdisk 机制的种种限制，Linus Torvalds 提出了一个想法：能否将 cache 当作一个文件系统直接挂载使用？基于这个想法，Linus Torvalds 基于已有的缓存机制实现了 ramfs。ramfs 与 ramdisk 有着本质的区别，ramdisk 本质上是基于内存的一个块设备，而 ramfs 是基于缓存的一个文件系统。因此，ramfs 去除了前述块设备的一些限制。比如，ramfs 根据其中

包含的文件大小可自由伸缩；增加文件时，自动分配内存；删除文件时，自动释放内存。更重要的是，ramfs 是基于已有的缓存机制，因此不必再像 ramdisk 那样需要和缓存之间进行多余的复制一环。

伴随着 ramfs 的出现，从 2.6 开始，内核开发人员基于 ramfs 开发了 initramfs 替代 initrd。那么 initramfs 是怎样工作的呢?

当 2.6 版本的内核引导时，在挂载真正的根文件系统之前，首先将挂载一个名为 rootfs 的文件系统，并将 rootfs 的根作为虚拟文件系统目录树的总根。那么为什么要使用 rootfs 这么一个中间过程呢？原因之一还是为了解决鸡和蛋的问题。内核需要根文件系统上的驱动以及程序来驱动和挂载根文件系统，但是这些驱动和程序有可能没有编译进内核，而在根文件系统上。如果不借助第三方，内核是没有办法挂载真正的根文件系统的。而 rootfs 虽然名称为 rootfs，但是并不是什么新的文件系统，事实上，其就是一个 ramfs，只不过换了一个名称。换句话说，rootfs 是在内存中的，内核不需要特殊的驱动就可以挂载 rootfs，所以内核使用 rootfs 作为一个过渡的桥梁。

在挂载了 rootfs 后，内核将 Bootloader 加载到内存中的 initramfs 中打包的文件解压到 rootfs 中，而这些文件中包含了驱动以及挂载真正的根文件系统的工具，内核通过加载这些驱动、使用这些工具，实现了挂载真正的根文件系统。此后，rootfs 也完成了历史使命，被真正的根文件系统覆盖（overmount）。但是 rootfs 作为虚拟文件系统目录树的总根，并不能被卸载。但是这没有关系，前面我们已经谈到了，rootfs 基于 ramfs，删除其中的文件即可释放其占用的空间。

4.2.1 挂载 rootfs

在讨论具体的挂载 rootfs 时，因为涉及了一些文件系统相关的概念，因此，为了更好地理解文件系统相关的概念，我们有必要先来了解一下文件系统的物理组织结构，以期对这些抽象的概念有个具体的认识。

以 ExtX（X=2,3,4）文件系统为例，其在存储介质上按照图 4-1 所示进行组织。虽然用于不同操作系统的文件系统其物理存储结构是不同的，但是 Linux 的虚拟文件系统通过为这些文件系统建立中间适配层，模拟这里介绍的概念来实现对这些文件系统的支持。

ExtX 文件系统使用块（Block）作为基本存储单元。ExtX 支持 1024、2048 和 4096 字节大小的块，块的大小是在创建文件系统时指定的，如果没有明确指出，mke2fs 将使用默认大小。ExtX 文件系统将整个分区分成多个块组（Block Group），除了最后一个块组，其他块组都包含相同数量的块。下面介绍每个块组包含的部分。

（1）超级块（Super Block）

超级块描述整个文件系统的信息，包括 Inode 总数，空闲 Inode 数量，每个块组包含的 Inode 的数量；块的总数，空闲块的数量，每个块组包含的块的数量，块的大小；挂载的次

数、最近一次挂载的时间等。

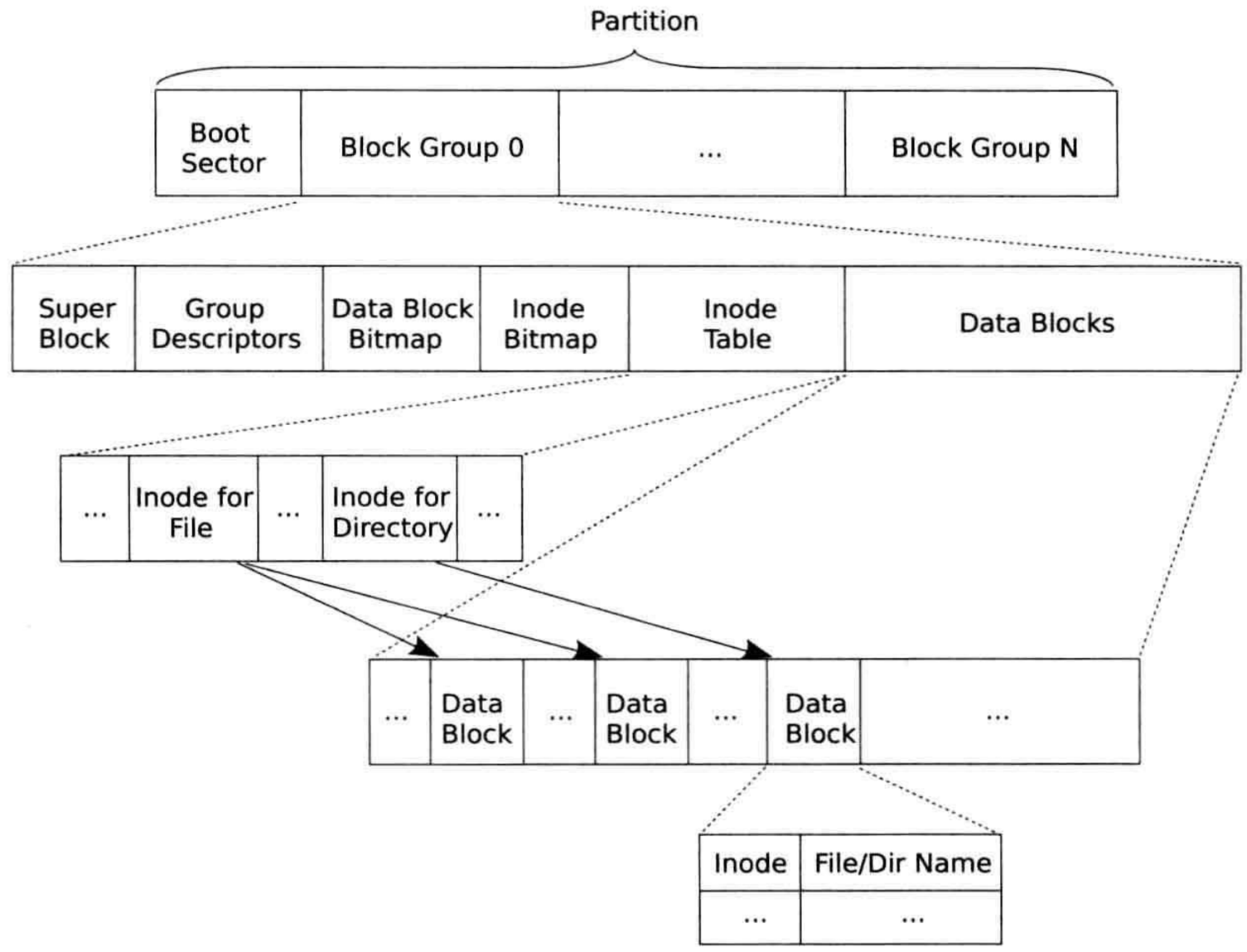


图 4-1 ExtX 文件系统的组织结构

（2）块组描述符（Group Descriptors）

块组描述符包含所有块组的描述。每个块组描述符存储一个块组的描述信息，包括块组中块位图所在的块、索引节点位图所在的块、索引节点表所在的块等等。

（3）块位图（Block Bitmap）

块位图用来描述块组中哪些块已用、哪些块空闲。其中每个位对应本块组中的一个块，这个位为 1 表示该块已用，为 0 表示该块空闲可用。

（4）索引节点位图（Inode Bitmap）

和块位图类似，索引节点位图用来描述索引节点表中哪些 Inode 已用、哪些 Inode 空闲。其中每个位对应索引节点位图中的一个 Inode，这个位为 1 表示该 Inode 已用，为 0 表示该 Inode 空闲可用。

（5）索引节点表（Inode Table）

一个文件除了需要存储数据以外，一些描述信息也需要存储，如文件类型（常规文件、目录等）、权限、文件大小、创建 / 修改 / 访问时间等，这些信息存储在 Inode 中而不是数据块中。每个文件都有一个对应的 Inode，一个块组中的所有 Inode 组成了索引节点表。除了文件属性信息外，Inode 中还记录了存储文件数据的数据块。

（6）数据块（Data Block）

数据块中存储的就是文件的数据。但是对于不同的文件类型，数据块中存储的内容是不

同的，以常规文件和目录为例：

- 对于常规文件，数据块中存储的是文件的数据。
- 对于目录，数据块中存储的是该目录下的所有文件名和子目录名。

当然了，不是所有的文件都需要数据块，如设备文件、socket 等特殊文件将相关信息全部保存在 Inode 中，就不需要数据块存储数据。

因为超级块和块组描述符是文件系统的关键信息，所以每个块组中都包含一份冗余的备份。ExtX 文件系统也允许在某些特殊的情况下，除了第 0 个块组外，其余的块组可以不包含超级块和块组描述符的备份。用户在创建文件系统时可以通过命令行参数告诉工具 mke2fs。

Linux 的虚拟文件系统将文件系统组织为树形结构。在初始化阶段，内核挂载 rootfs 文件系统，虚拟文件系统从无到有，rootfs 的根作为虚拟文件系统这棵大树中的第一个节点，自然成为所有后来创建的节点的祖先。也就是说，虚拟文件系统目录树的根就是 rootfs 的根。

本质上，rootfs 就是一个 ramfs 文件系统，根据下面 rootfs 的文件系统类型的定义就可看出这一点：

```
linux-3.7.4/fs/ramfs/inode.c:

static struct file_system_type rootfs_fs_type = {
    .name        = "rootfs",
    .mount       = rootfs_mount,
    .kill_sb     = kill_litter_super,
};

static struct dentry *rootfs_mount(struct file_system_type
       *fs_type, int flags, const char *dev_name, void *data)
{
    return mount_nodev(fs_type, flags|MS_NOUSER, data,
                       ramfs_fill_super);
}
```

根据 rootfs 中 mount 的具体实现——rootfs_mount 可见，创建 rootfs 超级块使用的函数恰恰是创建 ramfs 文件系统的函数 ramfs_fill_super，这从侧面表明了 rootfs 就是 ramfs。

在内核引导过程中，将调用 mnt_init 挂载 rootfs，代码如下所示：

```
linux-3.7.4/fs/namespace.c:

void __init mnt_init(void)
{
    ...
    init_rootfs();
    init_mount_tree();
}
```

mnt_init 首先调用 init_rootfs 向内核中注册了 rootfs 文件系统，代码如下所示：

```
linux-3.7.4/fs/ramfs/inode.c:
```

```
int __init init_rootfs(void)
{
    ...
    err = register_filesystem(&rootfs_fs_type);
    ...
}
```

然后，mnt_init 调用 init_mount_tree 挂载 rootfs，代码如下所示：

linux-3.7.4/fs/namespace.c：

```
static void __init init_mount_tree(void)
{
    struct vfsmount *mnt;
    struct mnt_namespace *ns;
    struct path root;

    mnt = do_kern_mount("rootfs", 0, "rootfs", NULL);
    ...
    root.mnt = mnt;
    root.dentry = mnt->mnt_root;

    set_fs_pwd(current->fs, &root);
    set_fs_root(current->fs, &root);
}
```

挂载 rootfs 的过程是由 do_kern_mount 来完成的，该函数所作的工作主要有以下几个方面。

（1）创建代表 rootfs 的 mount

Linux 的文件系统是按照树形组织的，不同的文件系统都可以挂载到这个树中的任何一个目录上来。内核使用数据结构 mount 记录具体的文件系统与虚拟文件系统这棵大树之间的关系，mount 起到一个承上启下的作用，其中的 mnt_mountpoint 指向文件系统的挂载点，mnt_parent 指向挂载点所在文件系统的 mount，mnt_root 指向要挂载的文件系统的根。所以 do_kern_mount 需要为 rootfs 创建一个 mount，因为 rootfs 是整个虚拟文件系统中第一个挂载的文件系统，所以这个 mount 实例是没有父亲的，其指向父亲 mount 成员的 mnt_parent 指向其自身，指向 rootfs 挂载点的成员 mnt_mountpoint 指向 rootfs 自己的根 mnt_root。

（2）创建 rootfs 的超级块

超级块用于描述整个文件系统信息，某种意义上，超级块就代表了整个文件系统，所以挂载文件系统时，需要创建超级块。对于一个常规的文件系统，内核将从存储介质上读入超级块信息。但是 ramfs 是一个基于内存的文件系统，并不存在所谓的存储介质，但是前面我们讨论的 ExtX 的文件系统的基本概念依然是适用的，ramfs 虽然不能从存储介质上读入超级块信息，但是会模拟出一个超级块。

（3）创建 rootfs 根节点的 Inode

内核也需要从文件系统中读入 rootfs 文件系统根节点的 Inode 信息。但是，与创建超级块同样的道理，对于 ramfs 来说，也是在内存中模拟一个根节点的 Inode 信息。

（4）创建 rootfs 根节点的 dentry

虽然在文件系统中，每个文件都有一个 Inode（对于那些没有的，Linux 将模拟 Inode，以使这些文件系统能够挂载到虚拟文件系统中），但是这个 Inode 主要是记录文件的数据块以及属性信息，而并没有记录文件间关系的信息。所以，内核设计了结构体 dentry，dentry 中记录了该文件的父节点和子节点，从而可以将文件挂载到虚拟文件系统树中。dentry 在物理存储介质中并没有对应的实体，而只存在于内存中。为了提高搜索文件的效率，内核会缓存文件系统中最近访问的 dentry。

挂载 rootfs 后，内核初始虚拟文件系统的结构如图 4-2 所示。

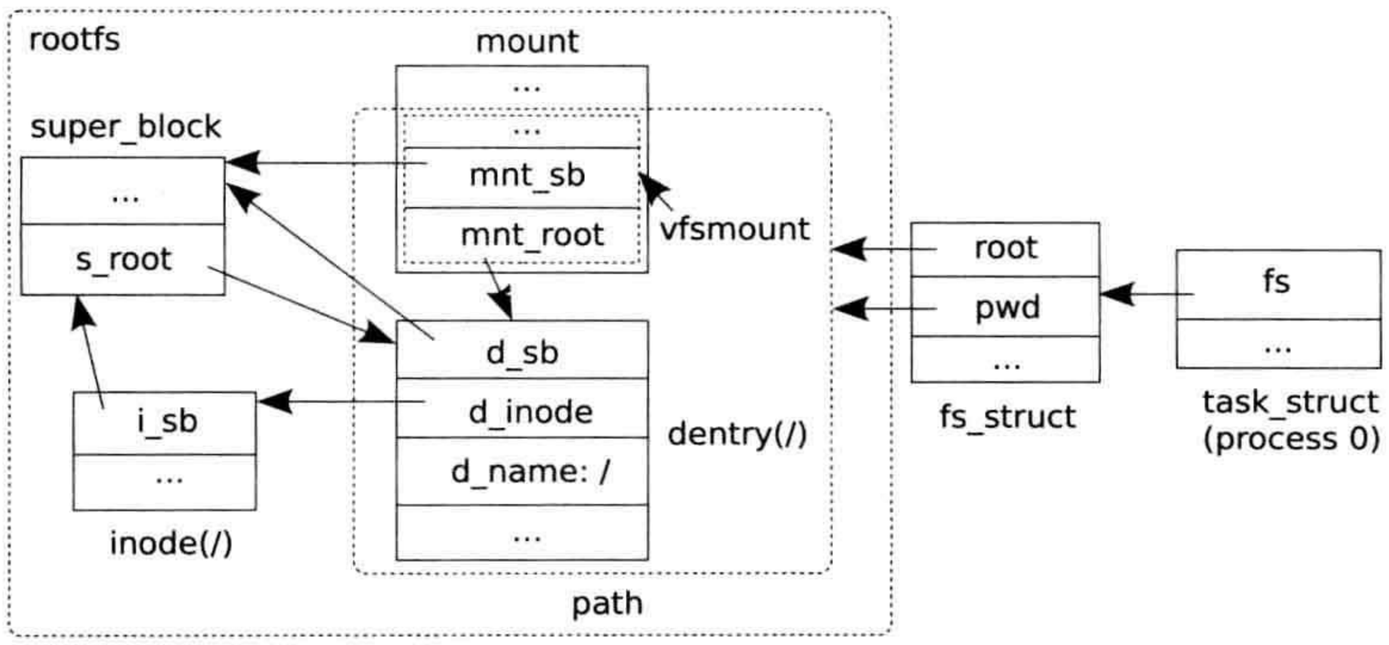


图 4-2 挂载 rootfs 后内核初始虚拟文件系统的结构

在虚拟文件系统中，通过文件系统的超级块、文件系统的根节点，再加上文件的 dentry，就可以在虚拟文件系统中唯一确定文件的位置了。为了程序实现上的方便，内核中设计了结构体 vfsmount 和 path。vfsmount“封装”了文件系统的超级块、文件系统的根节点等。path“封装”了 mount 和 dentry。

事实上，虚拟文件系统这棵代表整个文件系统的大树的根对用户并不可见，我们平时在进程中所见到的根目录，仅是这棵树上的一个分支而已。因此我们看到内核中文件系统中有 namespace 的概念，就是每个进程都有属于自己的文件系统空间，现实中多数进程的文件系统的 namespace 都是相同的。进程在任务结构体 task_struct 中的 fs_struct 中记录进程的文件系统的根，也就是进程的文件系统的 namespace，init_mount_tree 调用 set_fs_root 就是这个目的。当然，此时的 current 指向的是内核的原始进程，即进程 0。

至此，通过挂载 rootfs，虚拟文件系统的根目录已经建立起来，根目录已经可以容纳文件了。所以，接下来内核解压 initramfs 的内容到虚拟文件系统的根中，利用 initramfs 中的内容挂载并切换到真正的根文件系统。

4.2.2 解压 initramfs 到 rootfs

在挂载了 rootfs 后，内核将 Bootloader 加载到内存中的 initramfs 中的文件解压到 rootfs 中。而这些文件中包含了驱动以及挂载真正的根文件系统的工具，内核通过加载这些驱动、

使用这些工具实现挂载真正的根文件系统。

一旦配置内核支持 initramfs，那么内核将编译文件 initramfs.c，脚本如下所示：

```
linux-3.7.4/init/Makefile

obj-$(CONFIG_BLK_DEV_INITRD)    += initramfs.o
```

而在文件 initramfs.c 中，我们可看到如下代码：

```
linux-3.7.4/init/initramfs.c

rootfs_initcall(populate_rootfs);
```

宏 rootfs_initcall 告诉编译器将函数 populate_rootfs 链接在段“.initcall”部分。在内核初始化时，函数 do_basic_setup 调用 do_initcalls，而 do_initcalls 执行段“.initcall”中包含的函数，所以 initramfs 在此时被解压到 rootfs 中。

populate_rootfs 解压 initramfs 的代码如下所示：

```
linux-3.7.4/init/initramfs.c

static int __init populate_rootfs(void)
{
    char *err = unpack_to_rootfs(__initramfs_start, \
                                 __initramfs_size);
    if (err)
        panic(err); /* Failed to decompress INTERNAL initramfs */
    if (initrd_start) {
#ifdef CONFIG_BLK_DEV_RAM
...
#else
        printk(KERN_INFO "Unpacking initramfs...\n");
        err = unpack_to_rootfs((char *)initrd_start,
            initrd_end - initrd_start);
        ...
#endif
    }
    return 0;
}
```

根据 populate_rootfs 的代码可见：

1）populate_rootfs 首先调用 unpack_to_rootfs 将内核内置的 initramfs 解压到 rootfs 中；

2）接下来，如果变量 initrd_start 不为 0，那么说明还有一个外部的 initramfs 通过 Bootloader 加载了，内核将这个外部的 initramfs 也释放到 rootfs 中。其中 CONFIG_BLK_DEV_RAM 是对应于使用 ramdisk 机制的情况，我们不关心这种情况。initrd_start 是 initramfs 被加载到内存中的起始位置。initramfs 通常作为一个独立的外部文件存在，并通过 Bootloader 加载到内存。

事实上，内核也允许将 initramfs 和内核映像构建在一起，统一通过内核加载，使用的方法是：首先将 initramfs 的内容保存到一个目录；然后将这个目录指定给 kbuild，kbuild 将

使用自带的辅助程序 gen_init_cpio 将其压缩为 initramfs；最后链接到内核映像的段“.init.ramfs”中。这种方法的配置方式如图 4-3 所示。

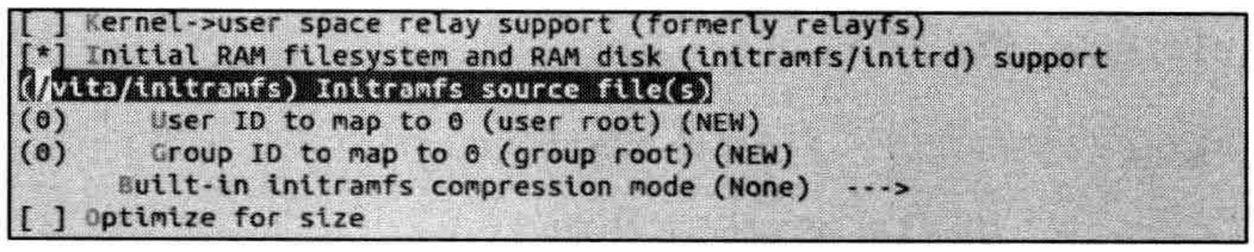


图 4-3 指定 initramfs 的源文件所在目录

但是，即使我们没有指定将 initramfs 包含进内核映像中，内核也会构建一个内置的 initramfs。这也是我们看到 populate_rootfs 代码中，第一个 unpack_to_rootfs 是无条件执行的原因。

接下来，我们就具体看看这个内置的 initramfs 的创建过程。

首先，内核的链接脚本告诉链接器将段“.init.ramfs”中包含的内容链接到内核映像的“Init code and data”部分，链接脚本如下所示：

```
linux-3.7.4/arch/x86/kernel/vmlinux.lds.S

SECTIONS
{
    ...
/* Init code and data - will be freed after init */
    . = ALIGN(PAGE_SIZE);
    .init.begin : AT(ADDR(.init.begin) - LOAD_OFFSET) {
        __init_begin = .; /* paired with __init_end */
    }
    ...
    INIT_DATA_SECTION(16)
    ...
    /* freed after init ends here */
    .init.end : AT(ADDR(.init.end) - LOAD_OFFSET) {
        __init_end = .;
    }
    ...
}
```

在“.init.begin”和“.init.end”之间的部分是内核初始化时使用的代码，在内核初始化完成后，将再无用处，因此，内核初始化完成后，这部分的代码将被释放。内核内置的 initramfs 就被包含在这里。我们先来看一下宏 INIT_DATA_SECTION 以及 INIT_RAM_FS 的定义：

```
linux-3.7.4/include/asm-generic/vmlinux.lds.h

#define INIT_DATA_SECTION(initsetup_align)                  \
    .init.data : AT(ADDR(.init.data) - LOAD_OFFSET) {       \
        INIT_DATA                           \
        INIT_SETUP(initsetup_align)                 \
        INIT_CALLS                          \
```

```
        CON_INITCALL                            \
        SECURITY_INITCALL                       \
        INIT_RAM_FS                          \
    }

#define INIT_RAM_FS                             \
    . = ALIGN(4);                               \
    VMLINUX_SYMBOL(__initramfs_start) = .;              \
    *(.init.ramfs)                              \
    . = ALIGN(8);                               \
    *(.init.ramfs.info)
```

由上可见，段“.init.ramfs”被链接到了内核映像的“Init code and data”部分；函数 unpack_to_rootfs 中使用的符号 __initramfs_start 指向段“.init.ramfs”的开头。

段“.init.ramfs”的具体内容在文件 initramfs_data.S 中，代码如下：

```
linux-3.7.4/usr/initramfs_data.S

.section .init.ramfs,"a"
__irf_start:
.incbin __stringify(INITRAMFS_IMAGE)
__irf_end:
.section .init.ramfs.info,"a"
.globl VMLINUX_SYMBOL(__initramfs_size)
VMLINUX_SYMBOL(__initramfs_size):
```

通过伪指令“.section .init.ramfs”，链接器将 initramfs 链接进段“.init.ramfs”。其中的 INITRAMFS_IMAGE 在 Makefile 中定义：

```
linux-3.7.4/usr/Makefile

AFLAGS_initramfs_data.o += \
  -DINITRAMFS_IMAGE="usr/initramfs_data.cpio$(suffix_y)"
```

initramfs 可以采用不同压缩方式，suffix_y 是对应的后缀。比如，如果使用的是 gzip 压缩方式，则后缀为“.gz”；如果使用的是 bzip2 压缩方式，则后缀是“.bz2”；等等。当然也可以不必压缩，因为内核最终会被压缩。

显然，initramfs_data.S 就是 initramfs 的内容（initramfs_data.cpio）的封装。另外在这个文件中定义了代表 initramfs 大小的符号 __initramfs_size，这个符号也是函数 populate_rootfs 解压内置的 initramfs 时需要的。

接下来，我们就来看看具体的 initramfs 的内容 initramfs_data.cpio，其创建规则的脚本如下所示：

```
linux-3.7.4/usr/Makefile

$(obj)/initramfs_data.cpio$(suffix_y): $(obj)/gen_init_cpio \
                                       $(deps_initramfs) klibcdirs
    ...
    $(call if_changed,initfs)
```

关注创建 initramfs_data.cpio 规则的命令，其中 if_changed 这个表达式在讨论内核的构建时我们已见过多次，其核心就是执行命令 cmd_$1，这里对应的就是 cmd_initfs，该命令定义如下：

```
linux-3.7.4/usr/Makefile

cmd_initfs = $(initramfs) -o $@ $(ramfs-args) $(ramfs-input)
```

其中涉及的三个参数的定义如下：

```
linux-3.7.4/usr/Makefile

initramfs   := $(CONFIG_SHELL) \
    $(srctree)/scripts/gen_initramfs_list.sh
ramfs-input := \
    $(if $(filter-out "",$(CONFIG_INITRAMFS_SOURCE)), \
        $(shell echo $(CONFIG_INITRAMFS_SOURCE)),-d)
ramfs-args  := \
    $(if $(CONFIG_INITRAMFS_ROOT_UID), -u \
        $(CONFIG_INITRAMFS_ROOT_UID)) \
        $(if $(CONFIG_INITRAMFS_ROOT_GID), -g \
        $(CONFIG_INITRAMFS_ROOT_GID))
```

其中：

- initramfs 是 scripts 目录下的脚本 gen_initramfs_list.sh。
- ramfs-input 指定了创建 initramfs 的输入。如果配置内核时指定了构成 initramfs 的源所在的目录，则使用这个源目录下的文件创建 initramfs；否则，只传递一个参数“-d”给脚本 gen_initramfs_list.sh，该脚本则用内核默认的内容创建 initramfs。
- ramfs-args 这个参数只有在用户自己指定了构建 initramfs 的文件时才有效，默认是“-u 0 –g 0”，是告诉内核将这些文件的用户 ID 和组 ID 都设置为 root。

综上所述，当指定了 initramfs 的源目录时，假设源目录为 /vita/initramfs，那么构建 initramfs 的命令展开后大致如下：

```
scripts/gen_initramfs_list.sh -o usr/initramfs_data.cpio \
    -u 0 -g 0 /vita/initramfs
```

脚本 gen_initramfs_list.sh 将 /vita/initramfs 目录下的文件的 UID 和 GID 全部设置为 0，即 root 用户和组的 ID，然后将目录下的内容打包为 initramfs_data.cpio。

当不指定 initramfs 的源目录时，创建内置的 initramfs 的命令展开后大致如下：

```
scripts/gen_initramfs_list.sh -o usr/initramfs_data.cpio -d
```

通过命令行参数“-d”，即“default initramfs”，告诉脚本 gen_initramfs_list.sh 创建一个默认的 initramfs，其包含的内容如下：

```
linux-3.7.4/scripts/gen_initramfs_list.sh
```

```
default_initramfs() {
    cat <<-EOF >> ${output}
        # This is a very simple, default initramfs

        dir /dev 0755 0 0
        nod /dev/console 0600 0 0 c 5 1
        dir /root 0700 0 0
        # file /kinit usr/kinit/kinit 0755 0 0
        # slink /init kinit 0755 0 0
    EOF
}
```

这个默认的 initramfs 非常简单，仅包括一个 /dev 目录，一个 /dev/console 设备节点以及一个 /root 目录。

最后还要指出的一点是，事实上，即使配置内核不支持 initramfs，内核在内部依然会构建一个最小的 initramfs。根据 init 目录下的 Makefile，当配置内核不支持 initramfs 时，内核链接 init 下的 noinitramfs.c，如下脚本所示：

```
linux-3.7.4/init/Makefile

ifneq ($(CONFIG_BLK_DEV_INITRD),y)
obj-y                          += noinitramfs.o
else
```

文件 noinitramfs.c 的代码如下：

```
static int __init default_rootfs(void)
{
    int err;

    err = sys_mkdir((const char __user __force *) "/dev", 0755);
    if (err < 0)
        goto out;

    err = sys_mknod((const char __user __force *) "/dev/console",
            S_IFCHR | S_IRUSR | S_IWUSR,
            new_encode_dev(MKDEV(5, 1)));
    if (err < 0)
        goto out;

    err = sys_mkdir((const char __user __force *) "/root", 0700);
    if (err < 0)
        goto out;

    return 0;

out:
    printk(KERN_WARNING "Failed to create a rootfs\n");
    return err;
}
rootfs_initcall(default_rootfs);
```

由代码“rootfs_initcall(default_rootfs)”可见，在没有配置内核支持 initramfs 的情况下，

内核初始化时，依然会执行 default_rootfs。而该函数将在 rootfs 中创建 /dev、/root 目录以及 /dev/console 节点。

可见，无论在什么情况下，内核都将确保有一个 initramfs。那么内核为什么要这么做呢？因为第一个进程如果打不开控制台设备（/dev/console），那么其将异常终止，最终导致内核 panic。所以，这个默认的 initramfs 确保了内核不会因为第一个进程打不开控制台设备而 panic。从某种意义上，也可以将其看作内核虚拟文件系统的一个 Bootstrap，也就是说，如果没人给内核提供一个最小的文件系统的内容，那么内核就自己创建一个。

4.2.3 挂载并切换到真正的根目录

将 initramfs 成功解压后，挂载真正的根文件系统所需的驱动、程序等已经全部俱备，可以挂载真正的根文件系统了。假设真正的根文件系统在第一块硬盘的第一个分区，即 /dev/sda1，并假设挂载点为 /root，那么挂载完成后，文件系统的目录树如图 4-4 所示。

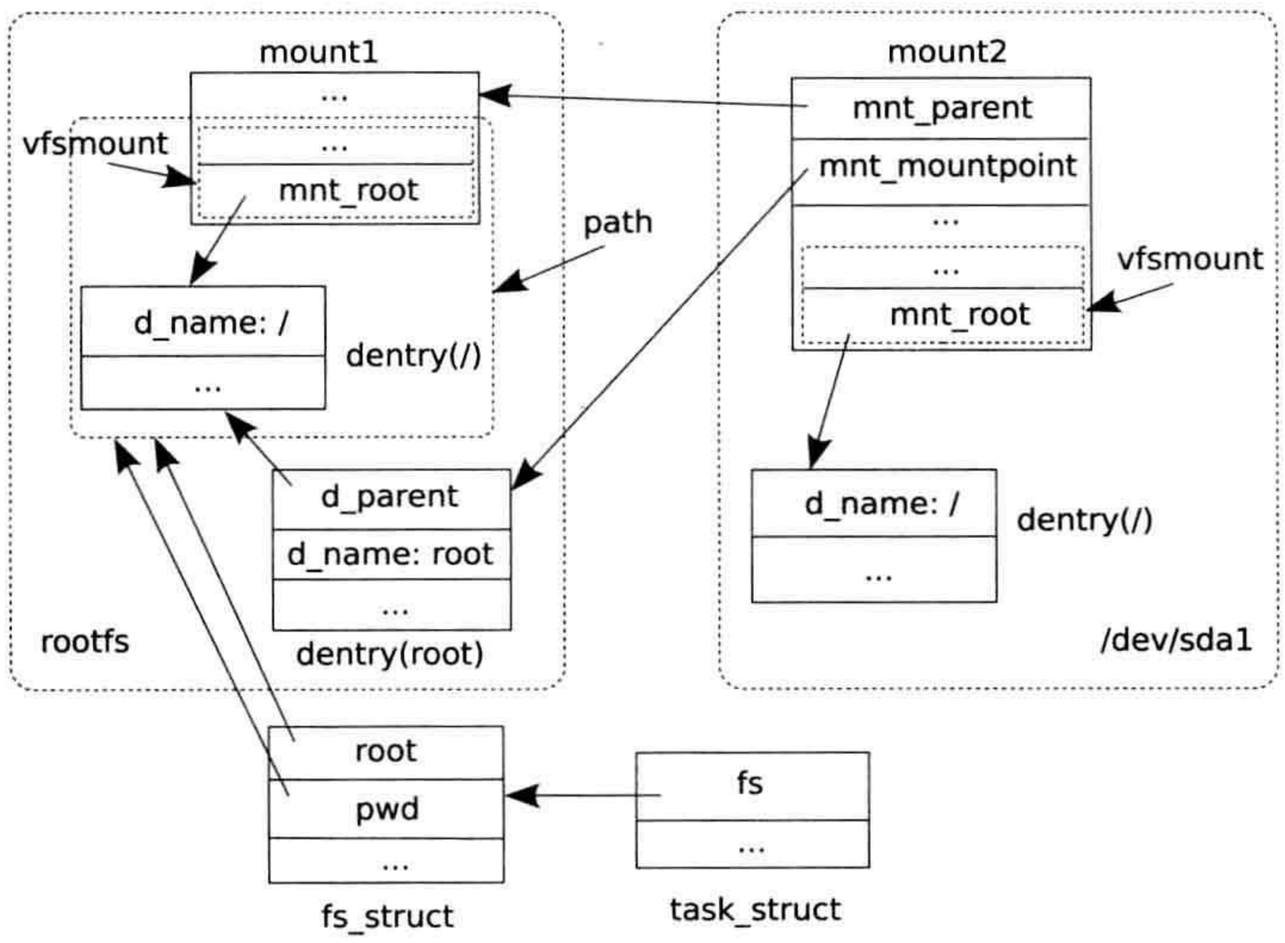


图 4-4 挂载根文件系统后虚拟文件系统结构

挂载真正的根文件系统，即 /dev/sda1 时，内核将为其创建一个 mount 对象，为了行文方便，这里用 mount2 指代。为了方便查找，mount2 会被内核加入到一个 Hash 表中。mount2 中的 mnt_parent 指向了代表 rootfs 的 mount1。mount2 中的 mnt_mountpoint 指向该文件系统的挂载点，显然，这里是 rootfs 中的 /root 目录。mount2 中的 mnt_root 指向代表根文件系统的根节点的 dentry。

此时，进程 0 的任务结构体中的 fs 的 root 和 pwd 均指向 rootfs 的根节点，当然，这个根节点是由 mount 和 rootfs 的根目录的 dentry 的共同标识的。也就是说，此时进程的文件系统的 namespace 是以 rootfs 的根目录作为根的目录树。

挂载真正的根文件系统后，rootfs 中的内容已经没有保留的意义，但是并不能将 rootfs

卸载，因为 rootfs 是整个虚拟文件系统的根。因此，为了不占用内存空间，将 rootfs 中的内容（文件）释放掉即可，然后将真正的根文件系统移动到虚拟文件系统的根（即 rootfs 的根）下，最后再将进程的文件系统的 namespace 切换到真正的根文件系统。切换后，虚拟文件系统中的相关数据结构间的关系如图 4-5 所示。

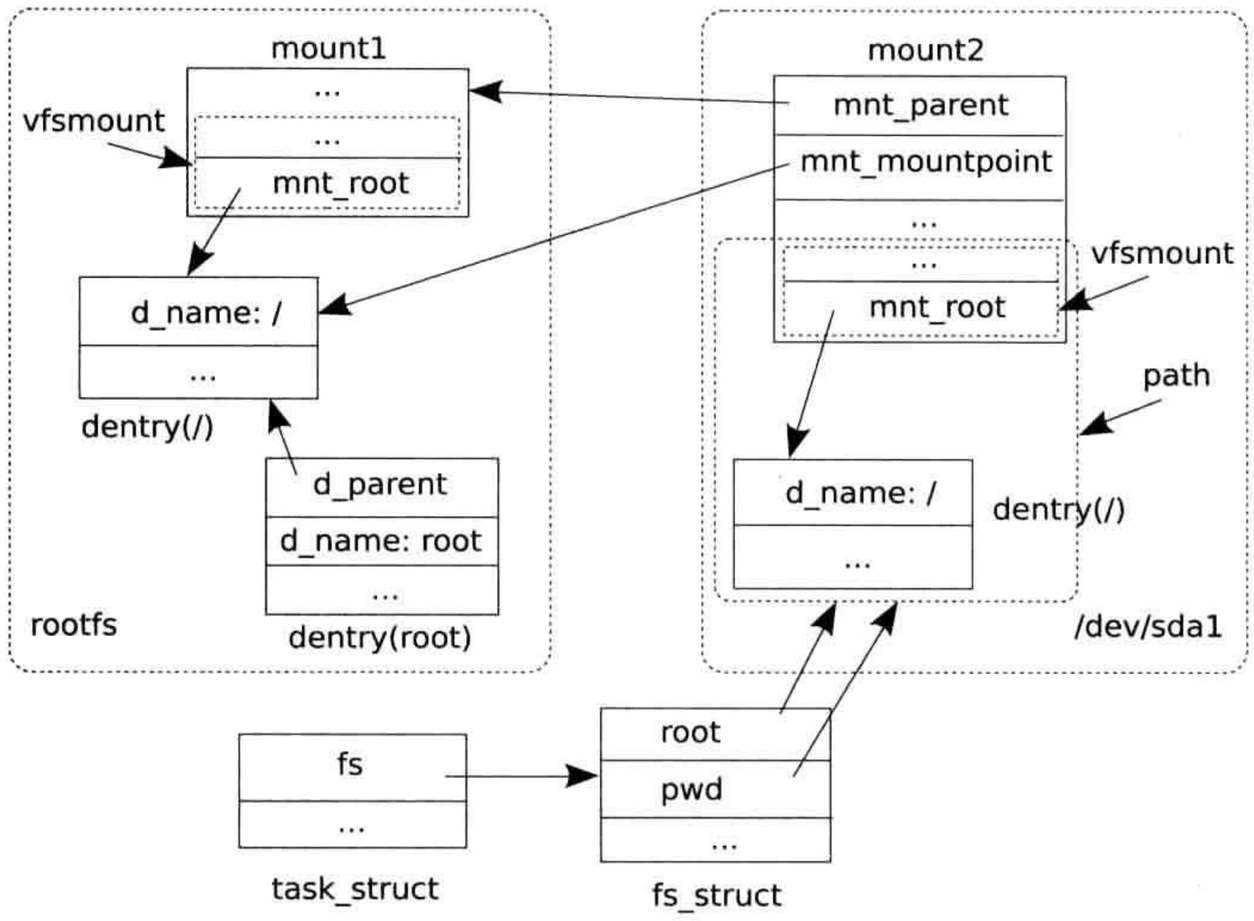


图 4-5 切换根目录后虚拟文件系统结构

4.3 配置内核支持 initramfs

要使用 initramfs，首先需要配置内核支持 initramfs，配置步骤如下：

1）执行 make menuconfig，出现如图 4-6 所示的界面。

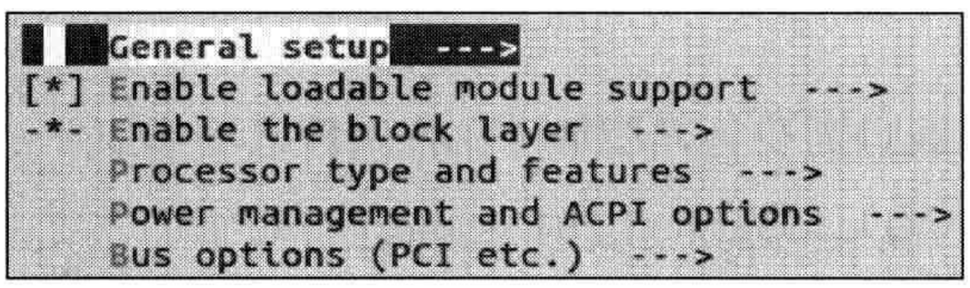


图 4-6 配置内核支持 initramfs（1）

2）在图 4-6 中选中“General setup”，出现如图 4-7 所示的界面。

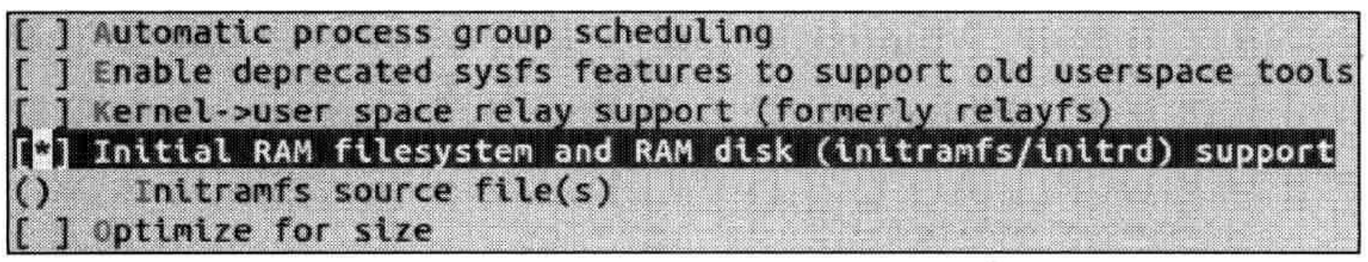


图 4-7 配置内核支持 initramfs（2）

3）在图 4-7 中选中“Initial RAM filesystem and RAM disk (initramfs/initrd) support”后，在该选择项的下方将出现一个子项“Initramfs source file(s)”。可以在这里指定 initramfs 所在的目录，内核编译时，将会把 initramfs 所在的目录压缩并链接到内核的一个特殊的数据段 .init.ramfs 中。

当然，我们也可以不将 initramfs 链接到内核中，而是将其作为一个单独的文件，由 Bootloader 将其加载到内存，而这也是通常的做法。这时，不必设置选项“Initramfs source file(s)”。

编译支持 initramfs 的内核，并将其保存到 /vita/sysroot/boot 目录下。

4.4 构建基本的 initramfs

在这一节，我们先建立一个 initramfs 的原型，用来验证内核配置以及这个 initramfs 原型。我们在 /vita 目录下创建一个 initramfs 目录，initramfs 的内容保存在这个目录中。

```
vita@baisheng:/vita$ mkdir initramfs
```

如果没有在传递给内核的命令行参数中指定“rdinit”，内核启动后，执行的 initramfs 中第一个程序是根目录下的 init。那我们就从创建 init 程序开始，来创建我们的 initramfs。

基本上所有的 init 程序都是采用 shell 脚本编写的，我们这里也不例外，当然你也可以采用其他语言（比如 C）来编写。这里要注意的一点是，编写 init 脚本时，用来指明脚本使用的解释器的字符串“#!/bin/bash”一定要从第一行的左侧第一个字符开始，因为内核中的脚本加载器将根据脚本文件的前两个字符判断使用什么解释器。具体如下：

```
/vita/initramfs/init:

#!/bin/bash
echo "Hello Linux!"
exec /bin/bash
```

编写完这个脚本后，我们需要为其增加可执行属性，具体如下：

```
vita@baisheng:/vita/initramfs$ chmod a+x init
```

这个脚本首先使用 echo 输出“Hello Linux!”，echo 是 shell 内置的命令，不需要再额外安装其他程序。然后运行一个交互式 shell，与用户进行交互。

shell 提供两种运行模式：一种是非交互模式，另外一种是交互模式。在运行解释 init 脚本文件时，最后会转化为形如“/bin/bash /init”，即将脚本文件 init 作为参数传给 bash 程序，这是典型的非交互方式，即 bash 的输入不是通过用户输入，而是保存在一个 shell 脚本文件中。

但是，我们需要通过 shell 与内核进行交互，因此，最后通过命令启动了一个新的 shell，这个 bash 程序没有任何输入，自然就以交互模式运行，因为其需要从用户获得输入。这里使

用 exec 的目的是后面的 shell 进程直接代替前面的 shell 进程，而不是复制出另外一个进程，也就是确保这个进程依然是 pid 为 1 的进程。

init 是需要 shell 解释器来解释运行的，因此，除了 init 脚本文件外，initramfs 中还需要 bash 程序。安装 bash 程序的命令如下：

```
vita@baisheng:/vita/initramfs$ mkdir bin
vita@baisheng:/vita/initramfs$ cp ../sysroot/bin/bash bin/
```

我们需要检查 bash 依赖的动态库，命令如下：

```
vita@baisheng:/vita/initramfs$ ldd 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
```

bash 依赖于 libc、libdl 以及 libgcc_s.so.1，因此，我们需要在 initramfs 中安装这三个库，以及安装加载动态库的动态加载 / 链接器。安装命令如下：

```
vita@baisheng:/vita/initramfs$ mkdir lib
vita@baisheng:/vita/initramfs$ cp -d /vita/sysroot/lib/libdl* \
   lib/
vita@baisheng:/vita/initramfs$ cp \
   /vita/sysroot/lib/libc-2.15.so lib/
vita@baisheng:/vita/initramfs$ cp -d \
   /vita/sysroot/lib/libc.so.6 lib/
vita@baisheng:/vita/initramfs$ cp \
   /vita/cross-tool/i686-none-linux-gnu/lib/libgcc_s.so.1 lib/
vita@baisheng:/vita/initramfs$ cp -d /vita/sysroot/lib/ld-* lib/
```

我们还需要检查它们的依赖。

```
vita@baisheng:/vita/initramfs$ ldd lib/libdl.so.2
        libc.so.6 => /vita/sysroot/lib/libc.so.6
        ld-linux.so.2 => /vita/sysroot/lib/ld-linux.so.2

vita@baisheng:/vita/initramfs$ ldd lib/libc.so.6
        ld-linux.so.2 => /vita/sysroot/lib/ld-linux.so.2

vita@baisheng:/vita/initramfs$ ldd lib/ld-linux.so.2

vita@baisheng:/vita/initramfs$ ldd lib/libgcc_s.so.1
        libc.so.6 => /vita/sysroot/lib/libc.so.6
```

根据依赖关系可见，libdl 依赖 libc 和动态链接器，libgcc 只依赖 libc，libc 仅依赖动态链接器，而动态链接器不依赖其他任何库，因此，我们不再需要安装其他库到 initramfs 中。

至此，基本的 initramfs 已经准备完成，打包前，我们可以使用 find 命令最后再检查一遍 initramfs 中的内容：

```
vita@baisheng:/vita/initramfs$ find .
.
```

```
./lib
./lib/libc.so.6
./lib/libc-2.15.so
./lib/ld-linux.so.2
./lib/libdl.so.2
./lib/libdl-2.15.so
./lib/ld-2.15.so
./lib/libgcc_s.so.1
./init
./bin
./bin/bash
```

最后我们将 initramfs 打包并压缩，保存在 /vita/sysroot/boot 目录下。根据内核要求，需要使用 cpio 压缩，并且压缩的格式为“newc”，具体命令如下：

```
vita@baisheng:/vita/initramfs$ find . | cpio -o -H newc \
    | gzip -9 > /vita/sysroot/boot/initrd.img
```

我们将 bzImage 和 initrd.img 复制到虚拟机：

```
vita@baisheng:/vita/sysroot/boot$ scp bzImage initrd.img \
        root@192.168.56.101:/vita/boot/
```

然后更改虚拟机的 GRUB 的配置文件 grub.cfg，告知 GRUB 加载 initrd。

```
menuentry 'vita' {
    set root='(hd0,2)'
    linux   /boot/bzImage root=/dev/sda2 ro
    initrd  /boot/initrd.img
}
```

重启系统，进入 vita 系统，运行结果如图 4-8 所示。

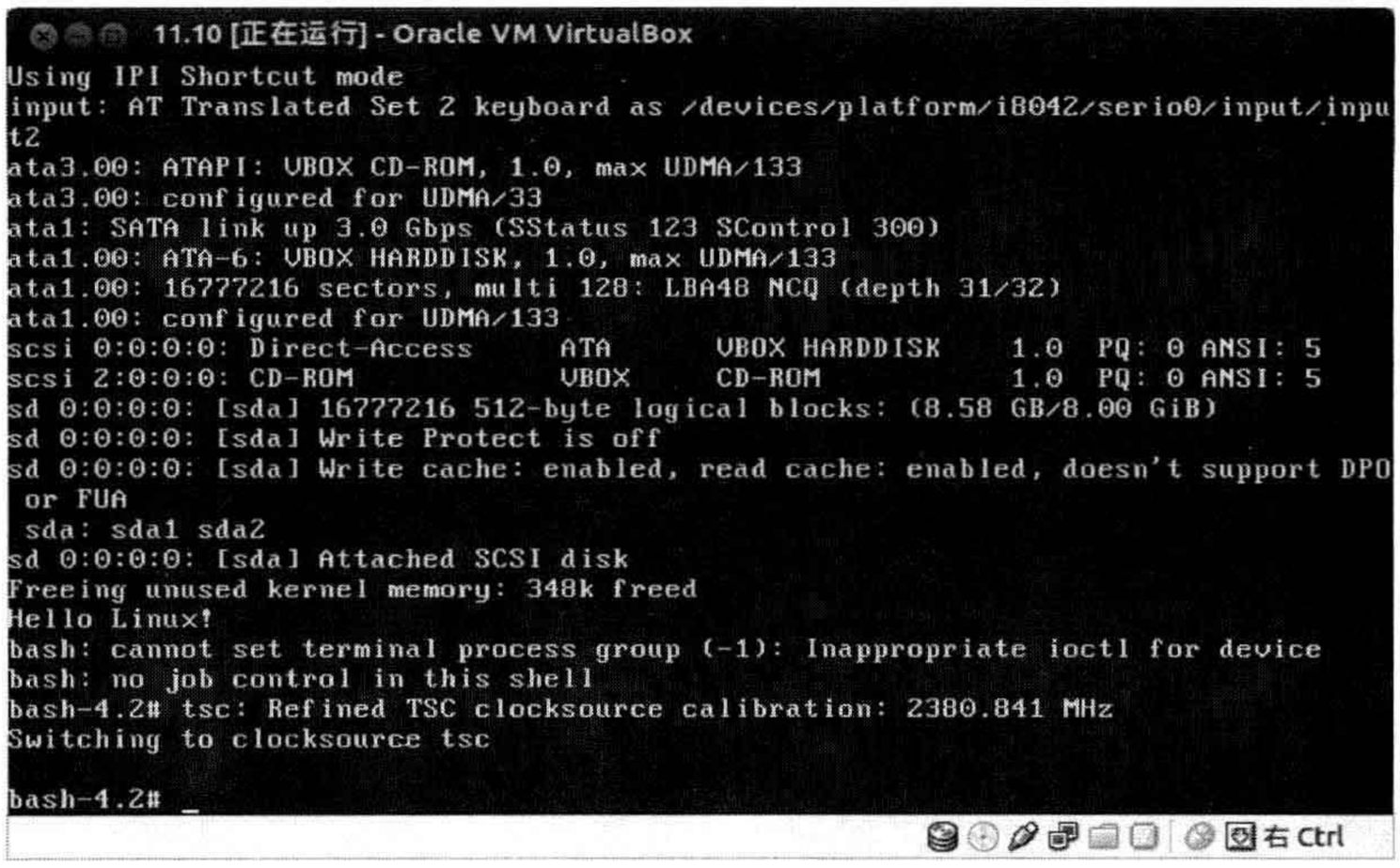


图 4-8 运行到 initramfs 中的 bash

由图 4-8 可见，initramfs 中的 init 输出了“Hello Linux!”后，启动了一个交互式的 shell。

4.5 将硬盘驱动编译为模块

initramfs 的重要作用之一就是允许内核将保存根文件系统的存储设备的驱动不再编译进内核。上一节，我们体验了基本的 initramfs，这一节，我们就将硬盘驱动编译为模块。

4.5.1 配置 devtmpfs

既然提到设备，而且 Linux 将设备也抽象为文件，这里就不得不讨论一下设备文件或者说设备节点。通常情况下，某些需要从用户空间访问的设备都会在文件系统中建立一个设备文件，作为用户空间访问设备的接口。得益于 Linux 中虚拟文件系统的设计，用户空间的程序可以像访问普通文件一样，使用标准的文件访问接口实现与设备的交互。

根据 FHS 的规定，设备文件存放在 /dev 目录下。在 Linux 系统的早期，设备文件是静态创建的，所有的设备节点是手动、事先创建的。笔者还记得在早期制作 Linux 发行版时，安装系统时，需要静态安装大量的设备文件，把所有可能的设备节点一并创建出来。但是，这样带来的一个问题就是，随着设备的种类越来越多，这个目录会越来越大，对于某一台具体的机器来说，dev 目录下充斥着大量无用的设备文件，因为某些设备在某些机器上根本就不存在。而且，这种方法会逐渐耗尽设备号，虽然可以通过扩展设备号的位数来增加设备号，但终究不是长久之计。

鉴于静态创建设备文件的种种问题，开发人员开发了 devfs。devfs 虽然解决了按需创建设备节点的机制，但是还是有很多问题，比如设备文件的名称依然由驱动开发人员在代码中指定，而不能由系统管理员指定。因此，后来又出现了 udev，使得设备命名策略、权限控制等都在用户空间完成。如此，设备文件不再是静态创建，而是由 udev 根据内核检测到的实际连接的设备，创建相应的设备文件。

对于动态创建设备文件，推荐在 /dev 目录下挂载一个基于内存的文件系统。基于内存的文件系统会完全驻留在 RAM 中，读写可以瞬间完成。除了性能优势之外，基于内存的文件系统的另外一个特点就是没有持久性，基于内存的文件系统中的数据在系统重新启动之后不会保留。这看起来可能不像是个积极因素，然而，对于动态创建设备节点这种情况来说，这实际上是一个优势。在系统关闭后，所有的设备节点无须保留，系统重启后，udev 将根据内核检测到的实际设备创建设备文件。

Linux 从 2.6.18 开始采用 udev，/dev 目录使用了基于内存的文件系统 tmpfs 管理设备文件。

2009 年初，开发人员又提出了 devtmpfs，并在同年年底被 Linux 2.6.32 正式收录。内核引导时，devtmpfs 将所有注册的设备在 devtmpfs 中建立相应的设备文件，一旦进入用户空间，在启动 udev 前，就可以将 devtmpfs 挂载到 /dev 目录下。也就是说，在启动 udev

前，devtmpfs 中已经建立了初步的设备文件，一般启动程序不必再等待 udev 建立设备节点，甚至在某些嵌入式系统上，不再需要 udev 创建设备节点，因为这个基本的 /dev 已经足够，从而缩短了系统的启动时间。同 rootfs 类似，devtmpfs 也不是新设计的文件系统，如果内核配置支持 tmpfs，那么其就是 tmpfs；否则，devtmpfs 就是 ramfs，只不过换了一个名字而已。

下面我们就实际体验一下 devtmpfs。为此，我们需要在 initramfs 中安装工具 ls 和 mount。工具 ls 在 coreutils 中，所以首先编译安装 coreutils：

```
vita@baisheng:/vita/build$ tar \
   xvf ../source/coreutils-8.20.tar.xz
vita@baisheng:/vita/build/coreutils-8.20$ ./configure \
   --prefix=/usr
vita@baisheng:/vita/build/coreutils-8.20$ make install
```

下面使用 ldd 检查 ls 依赖的库：

```
vita@baisheng:/vita$ ldd sysroot/usr/bin/ls
        librt.so.1 => /vita/sysroot/lib/librt.so.1
        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

vita@baisheng:/vita$ ldd sysroot/lib/librt.so.1
        libc.so.6 => /vita/sysroot/lib/libc.so.6
        libpthread.so.0 => /vita/sysroot/lib/libpthread.so.0
        ld-linux.so.2 => /vita/sysroot/lib/ld-linux.so.2

vita@baisheng:/vita$ ldd sysroot/lib/libpthread.so.0
        libc.so.6 => /vita/sysroot/lib/libc.so.6
        ld-linux.so.2 => /vita/sysroot/lib/ld-linux.so.2
```

根据 ldd 的输出可见，ls 依赖的库除了 librt 和 libpthread 尚未安装到 initramfs 中外，其余已经安装，所以我们将 ls 以及 librt 和 libpthread 复制到 initramfs 中：

```
vita@baisheng:/vita$ cp sysroot/usr/bin/ls initramfs/bin/
vita@baisheng:/vita$ cp -d sysroot/lib/librt* initramfs/lib/
vita@baisheng:/vita$ cp -d sysroot/lib/libpthread* initramfs/lib/
```

工具 mount 在软件包 util-linux 中，我们首先来编译这个软件包：

```
vita@baisheng:/vita/build$ tar xvf \
       ../source/util-linux-2.22.tar.xz
vita@baisheng:/vita/build/util-linux-2.22$ ./configure \
       --prefix=/usr --disable-use-tty-group --disable-login \
       --disable-sulogin --disable-su --without-ncurses
vita@baisheng:/vita/build/util-linux-2.22$ make
vita@baisheng:/vita/build/util-linux-2.22$ make install
vita@baisheng:/vita/build/util-linux-2.22$ find $SYSROOT \
   -name "*.la" -exec rm -f '{}' \;
```

下面使用 ldd 检查 mount 依赖的库：

```
vita@baisheng:/vita$ ldd sysroot/bin/mount
        libmount.so.1 => /vita/sysroot/lib/libmount.so.1
        libblkid.so.1 => /vita/sysroot/lib/libblkid.so.1
        libuuid.so.1 => /vita/sysroot/lib/libuuid.so.1
        libc.so.6 => /vita/sysroot/lib/libc.so.6

vita@baisheng:/vita$ ldd sysroot/lib/libmount.so.1
        libblkid.so.1 => /vita/sysroot/lib/libblkid.so.1
        libuuid.so.1 => /vita/sysroot/lib/libuuid.so.1
        libc.so.6 => /vita/sysroot/lib/libc.so.6

vita@baisheng:/vita$ ldd sysroot/lib/libblkid.so.1
        libuuid.so.1 => /vita/sysroot/lib/libuuid.so.1
        libc.so.6 => /vita/sysroot/lib/libc.so.6

vita@baisheng:/vita$ ldd sysroot/lib/libuuid.so.1
        libc.so.6 => /vita/sysroot/lib/libc.so.6
```

根据 ldd 的输出可见，mount 依赖 libmount、libblkid、libuuid 以及 libc。libc 已经安装到了 initramfs 中，我们将 mount 和其余几个库复制到 initramfs：

```
vita@baisheng:/vita$ cp sysroot/bin/mount initramfs/bin/
vita@baisheng:/vita$ cp -d sysroot/lib/libmount.so.1* \
   initramfs/lib/
vita@baisheng:/vita$ cp -d sysroot/lib/libblkid.so.1* \
   initramfs/lib/
vita@baisheng:/vita$ cp -d sysroot/lib/libuuid.so.1* \
   initramfs/lib/
```

重新压缩 initramfs，并将其保存到 /vita/sysroot/boot/ 目录下。接下来，我们准备支持 devtmpfs 的内核，配置步骤如下：

1）执行 make menuconfig，出现如图 4-9 所示的界面。

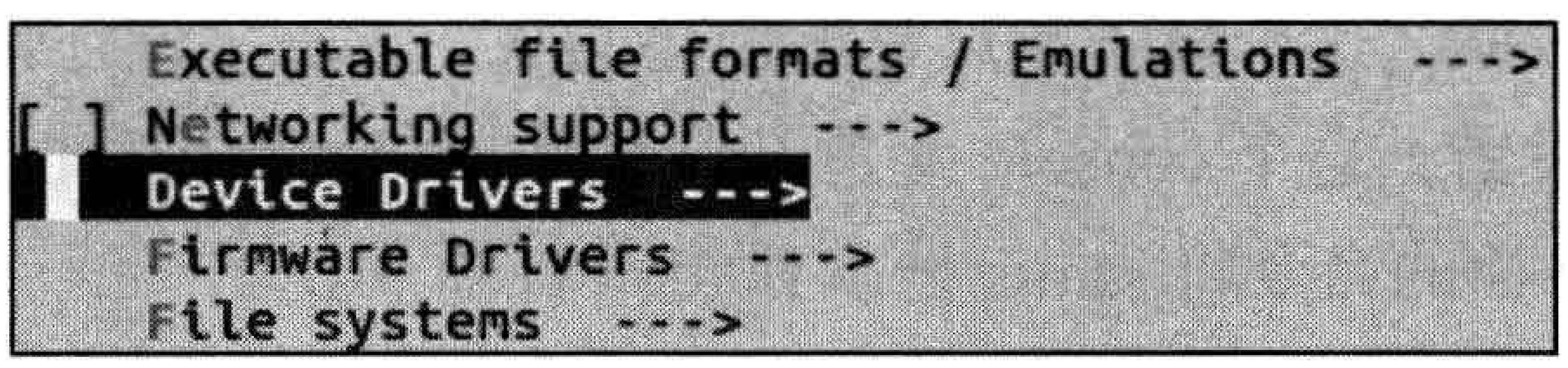


图 4-9 配置内核支持 devtmpfs（1）

2）在图 4-9 中，选择“Device Drivers”，出现如图 4-10 所示的界面。

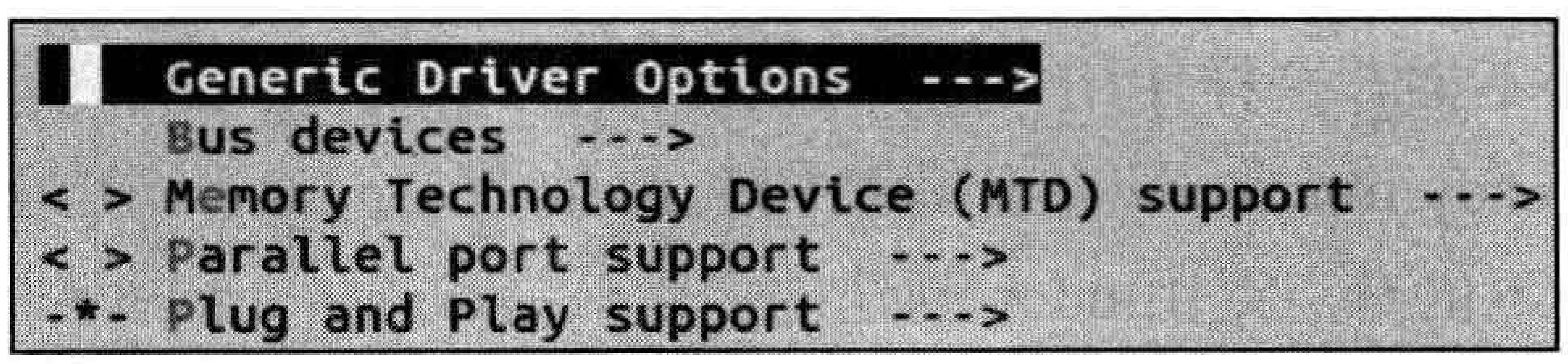


图 4-10 配置内核支持 devtmpfs（2）

3）在图 4-10 中，选择“Generic Driver Options”，出现如图 4-11 所示的界面。

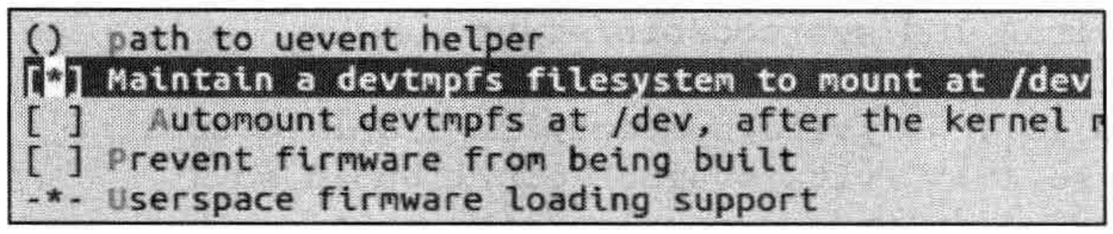


图 4-11　配置内核支持 devtmpfs（3）

4）在图 4-11 中，选中“Maintain a devtmpfs filesystem to mount at /dev”。

编译内核，并将内核与前面准备好的 initramfs 复制到虚拟机的目标系统上，然后重启进入 vita 系统。使用 ls 列出 vita 系统 /dev 下的设备文件，如图 4-12 所示。

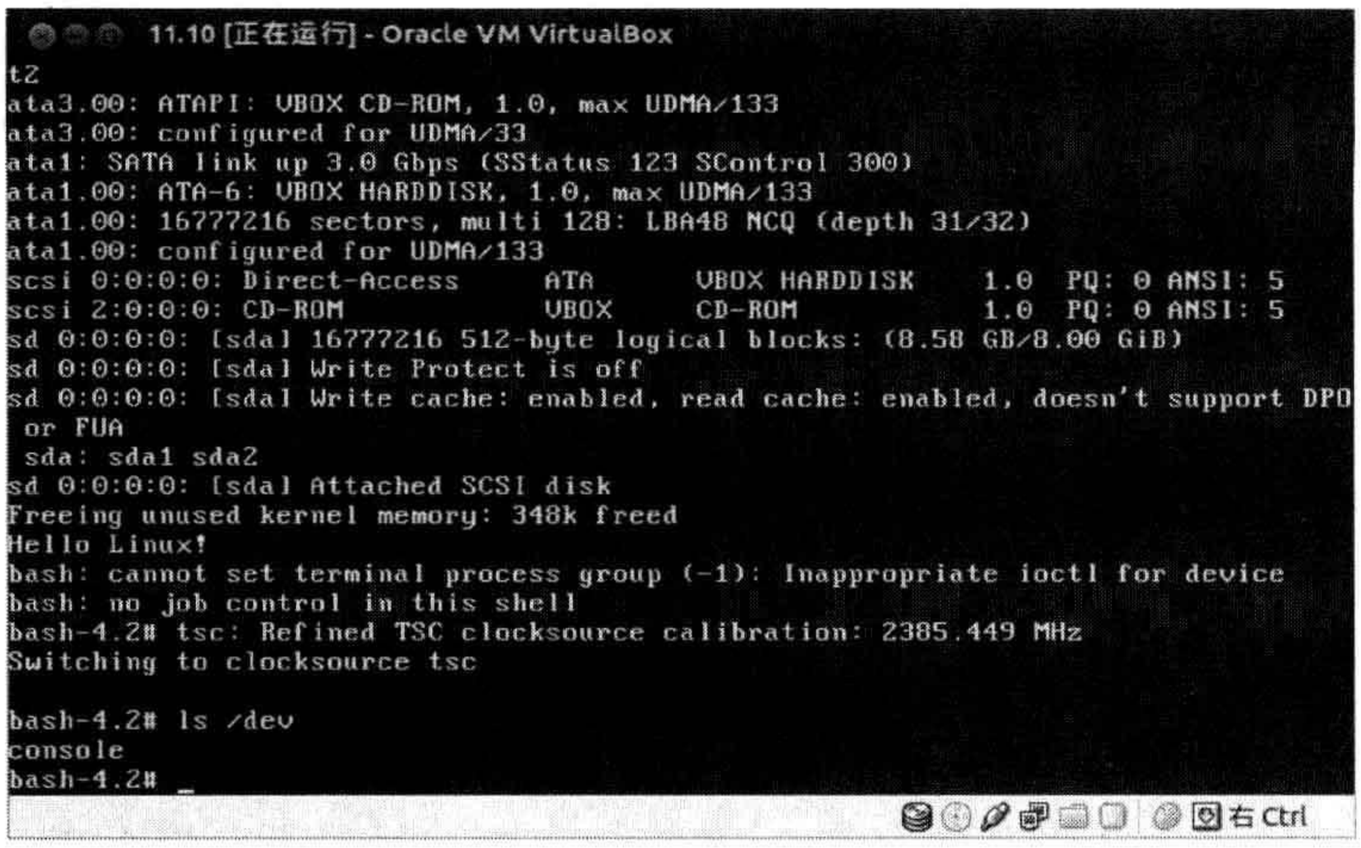


图 4-12　未挂载任何文件系统的 /dev 目录

根据图 4-12 可见，/dev 目录下包含一个设备节点 console。但是我们的 initramfs 中并没有包含这个设备节点，那么这个设备节点是谁创建的呢？大家一定还记得我们前面讨论过的内置的 initramfs 吧？没错，这个 console 就是由内置的 initramfs 创建的。

接下来我们使用下面的命令将 ramfs 挂载到 /dev 目录下。

```
mount -n -t ramfs none /dev
```

因为 ramfs 只是一个基于内存的文件系统，与设备无关，所以 mount 命令中的“device”参数可以使用任意字串描述。习惯上，对于这类没有具体设备的，一般使用字串“none”表示。但是 mount 命令不推荐这样使用，因为在某些情况下，某些提示很容易让用户费解，比如“mount : none already mounted”。这里我们暂时使用“none”，在最终系统中我们使用“udev”，表示该目录下的节点是由 udev 创建的。默认情况下，mount 命令会在文件 /etc/mtab 中维护一份当前已挂载的文件系统列表，因为我们的 initramfs 中没有创建 /etc/mtab 文件，initramfs 中也不需要，因此使用参数“-n”告诉 mount 不需维护这份列表。

挂载完成后，我们查看 /dev 下的设备文件。情况非常糟糕，/dev 下挂载了一个空的 ramfs 文件系统，原有的 console 设备节点也被覆盖了，如图 4-13 所示。

```
11.10 [正在运行] - Oracle VM VirtualBox
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
Freeing unused kernel memory: 348k freed
Hello Linux!
bash: cannot set terminal process group (-1): Inappropriate ioctl for device
bash: no job control in this shell
bash-4.2# tsc: Refined TSC clocksource calibration: 2385.449 MHz
Switching to clocksource tsc

bash-4.2# ls /dev
console
bash-4.2# mount -n -t ramfs none /dev
bash-4.2# ls /dev
bash-4.2# _
                                                          右 Ctrl
```

图 4-13　挂载 ramfs 文件系统的 /dev 目录

接下来我们使用下面的命令将 devtmpfs 挂载到 /dev 目录下：

```
mount -n -t devtmpfs none /dev
```

挂载完成后，我们再次查看 /dev 下的设备文件。可以看到，devtmpfs 下面已经建立了若干设备节点，如图 4-14 所示。

```
11.10 [正在运行] - Oracle VM VirtualBox
Freeing unused kernel memory: 348k freed
Hello Linux!
bash: cannot set terminal process group (-1): Inappropriate ioctl for device
bash: no job control in this shell
bash-4.2# tsc: Refined TSC clocksource calibration: 2385.449 MHz
Switching to clocksource tsc

bash-4.2# ls /dev
console
bash-4.2# mount -n -t ramfs none /dev
bash-4.2# ls /dev
bash-4.2# mount -n -t devtmpfs none /dev
bash-4.2# ls /dev
console             random  tty14  tty24  tty34  tty44  tty54  tty7
cpu_dma_latency     sda     tty15  tty25  tty35  tty45  tty55  tty8
full                sda1    tty16  tty26  tty36  tty46  tty56  tty9
input               sda2    tty17  tty27  tty37  tty47  tty57  urandom
kmsg                tty     tty18  tty28  tty38  tty48  tty58  vcs
mem                 tty0    tty19  tty29  tty39  tty49  tty59  vcs1
network_latency     tty1    tty2   tty3   tty4   tty5   tty6   vcsa
network_throughput  tty10   tty20  tty30  tty40  tty50  tty60  vcsa1
null                tty11   tty21  tty31  tty41  tty51  tty61  vga_arbiter
port                tty12   tty22  tty32  tty42  tty52  tty62  zero
ptmx                tty13   tty23  tty33  tty43  tty53  tty63
bash-4.2# _
                                                          右 Ctrl
```

图 4-14　挂载 devtmpfs 后的 /dev 目录

既然 devtmpfs 有这么多的好处，/dev 目录当然要使用 devtmpfs 文件系统了。因此，按照下面的脚本修改 initramfs 中的 init 文件：

```
/vita/initramfs/init

#!/bin/bash
echo "Hello Linux!"
```

```
mount -n -t devtmpfs udev /dev
exec /bin/bash
```

4.5.2 将硬盘控制器驱动配置为模块

与前面配置硬盘控制器驱动类似，只不过这里我们将“AHCI SATA support”和“Intel ESB, ICH, PIIX3, PIIX4 PATA/SATA support”配置为模块，如图 4-15 所示。

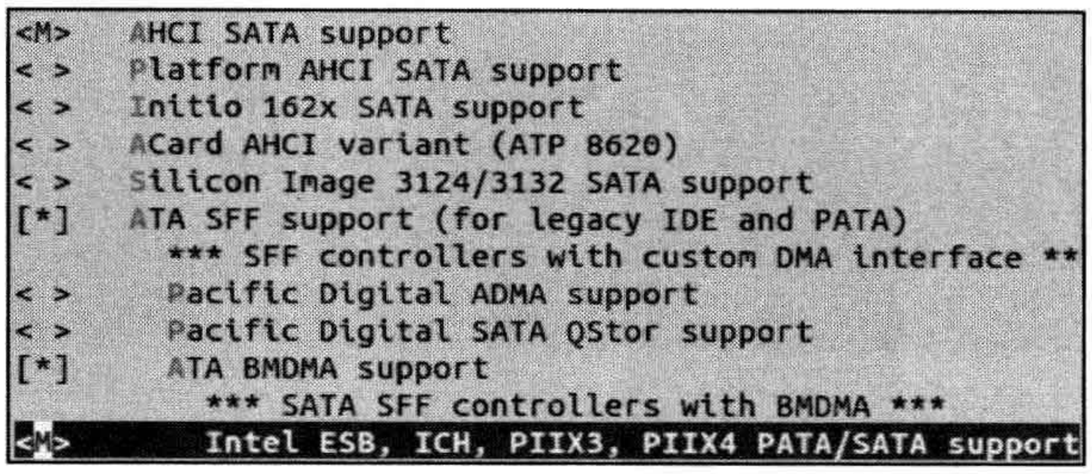


图 4-15 配置 SATA 控制器驱动为模块

接下来重新编译内核和模块。内核和模块可以使用单独的命令分开编译，也可以使用一条 make 命令同时编译内核和模块。编译完成后，将模块暂时安装在“/vita/sysroot/lib/modules”目录下。

```
vita@baisheng:/vita/build/linux-3.7.4$ make bzImage
vita@baisheng:/vita/build/linux-3.7.4$ make modules
vita@baisheng:/vita/build/linux-3.7.4$ make \
   INSTALL_MOD_PATH=$SYSROOT modules_install
```

最终安装的硬盘控制器驱动模块包括：

```
vita@baisheng:/vita$ ls \
   sysroot/lib/modules/3.7.4/kernel/drivers/ata/
ahci.ko  ata_piix.ko  libahci.ko
```

我们将其复制到 initramfs 中。

```
vita@baisheng:/vita$ mkdir -p \
   initramfs/lib/modules/3.7.4/kernel/drivers/ata
vita@baisheng:/vita$ cp \
   sysroot/lib/modules/3.7.4/kernel/drivers/ata/* \
   initramfs/lib/modules/3.7.4/kernel/drivers/ata/
```

为了加载内核模块，我们需要安装加载、卸载等管理模块的工具，这些工具在包 kmod 中：

```
vita@baisheng:/vita/build$ tar xvf ../source/kmod-12.tar.xz
vita@baisheng:/vita/build/kmod-12$ ./configure --prefix=/usr
vita@baisheng:/vita/build/kmod-12$ make
vita@baisheng:/vita/build/kmod-12$ make install
vita@baisheng:/vita/build/kmod-12$ find $SYSROOT \
   -name "*.la" -exec rm -f '{}' \;
```

检查 kmod 的依赖：

```
vita@baisheng:/vita$ ldd sysroot/usr/bin/kmod
        libkmod.so.2 => /vita/sysroot/usr/lib/libkmod.so.2
        libc.so.6 => /vita/sysroot/lib/libc.so.6

vita@baisheng:/vita$ ldd sysroot/usr/lib/libkmod.so.2
        libc.so.6 => /vita/sysroot/lib/libc.so.6
```

根据输出可见，kmod 及库 libkmod 只依赖 libc 库，而 libc 已经安装到 initramfs，所以复制 kmod 及库 libkmod 到 initramfs 即可，具体如下：

```
vita@baisheng:/vita/initramfs$ mkdir usr/bin
vita@baisheng:/vita$ cp sysroot/usr/bin/kmod initramfs/usr/bin/
vita@baisheng:/vita$ cp -d sysroot/usr/lib/libkmod.so.2* \
   initramfs/lib/
```

kmod 是 module-init-tools 的替代者，但是 kmod 是向后兼容 module-init-tools 的，虽然 kmod 只提供一个工具 kmod，但是通过符号链接的形式支持 module-init-tools 中的各个命令，而且目前来看，也只能使用这种方式来使用各种模块管理命令。因此，需要为各个模块管理命令建立符号链接，并将这些符号链接也复制到 initramfs 中：

```
vita@baisheng:/vita/sysroot/sbin$ ln -s ../usr/bin/kmod insmod
vita@baisheng:/vita/sysroot/sbin$ ln -s ../usr/bin/kmod rmmod
vita@baisheng:/vita/sysroot/sbin$ ln -s ../usr/bin/kmod modinfo
vita@baisheng:/vita/sysroot/sbin$ ln -s ../usr/bin/kmod lsmod
vita@baisheng:/vita/sysroot/sbin$ ln -s ../usr/bin/kmod modprobe
vita@baisheng:/vita/sysroot/sbin$ ln -s ../usr/bin/kmod depmod

vita@baisheng:/vita$ mkdir initramfs/sbin
vita@baisheng:/vita/sysroot/sbin$ cp -d insmod rmmod \
   modinfo lsmod modprobe depmod /vita/initramfs/sbin/
```

其中，insmod、rmmod、modprobe 用于加载 / 卸载模块；modinfo 用于查看模块信息；lsmod 用于查看已经加载的模块；depmod 用于创建模块间的依赖关系。注意，这里一定要将 modprobe 等命令放在 /sbin 目录下，因为后面的 udevd 将会到使用“/sbin/modprobe”的形式调用 modprobe 命令。最新的合并到 systemd 中的 udev 不再直接调用 modprobe 等工具，而是使用 libkmod 提供的库提供的 API 加载模块，但并无本质区别。

bash 的默认搜索命令的路径为 /usr/gnu/bin:/usr/local/bin:/bin:/usr/bin:.。我们当然可以使用全路径运行命令，比如 /sbin/insmod，但是为了方便，我们还是在 init 脚本中对搜索命令的路径进行一些调整。bash 中，保存搜索命令的环境变量为 PATH：

```
#!/bin/bash
echo "Hello Linux!"
export PATH=/usr/sbin:/usr/bin:/sbin:/bin
mount -n -t devtmpfs udev /dev
exec /bin/bash
```

压缩 initramfs，并将其和不包含硬盘驱动的 bzImage 复制到虚拟机，然后重启系统。进

入系统后，因为内核中已经没有硬盘控制器的驱动，所以在 /dev 目录下不会有类似 /dev/sdaX 的设备节点。为了识别硬盘，我们需要加载硬盘控制器的驱动。

前面提到，Intel 的 SATA 控制器可以运行在 Compatibility 模式和 AHCI 模式。笔者的机器的 SATA 控制器工作在 AHCI 模式，因此使用 ahci 驱动。但是在试图加载 ahci 模块时，ahci 模块在报告了若干找不到的符号后，加载以失败告终，如图 4-16 所示。

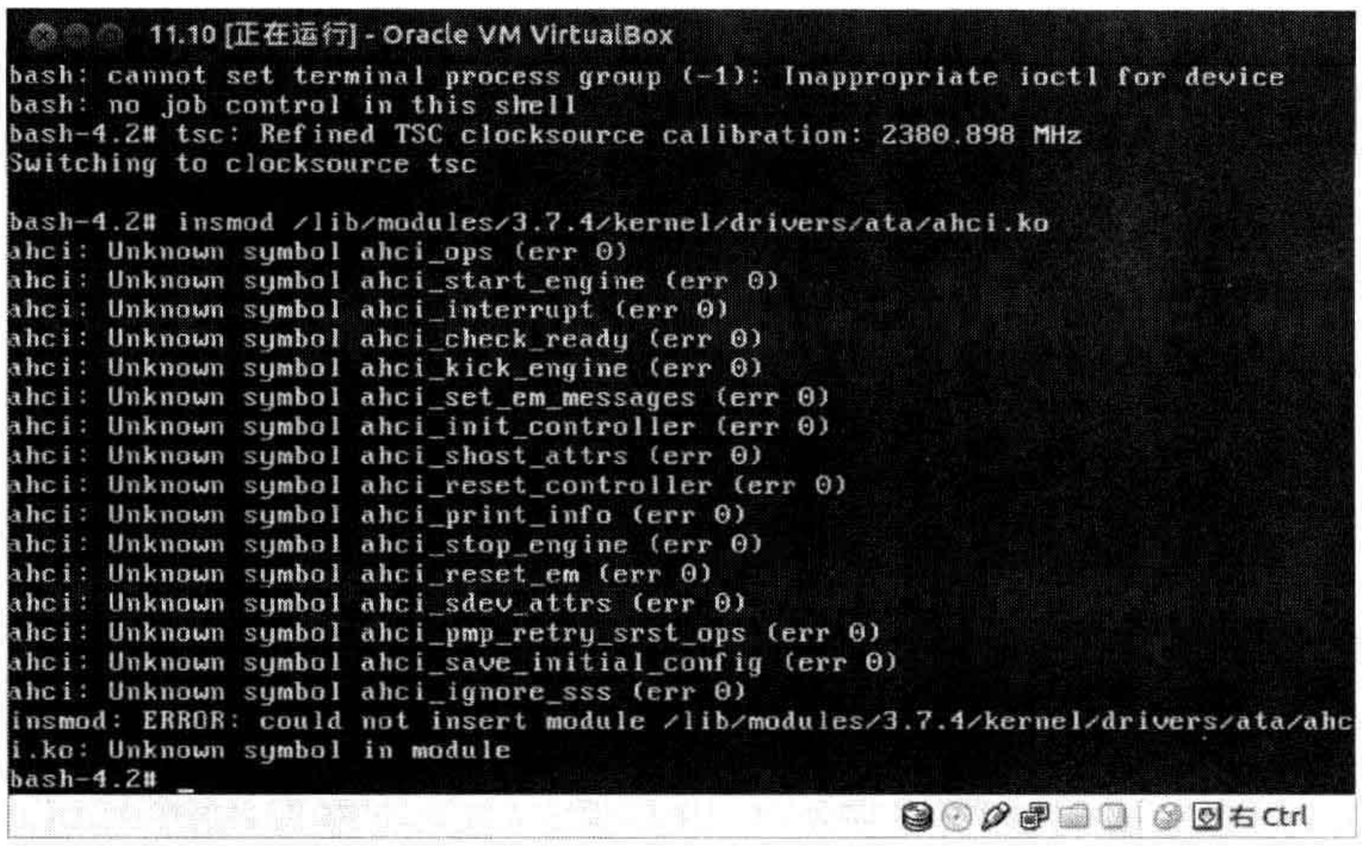


图 4-16 加载 ahci 模块失败

显然，这些找不到的符号应该定义在其他某个（些）未加载的模块中，我们需要使用命令 modinfo 查看一下模块 ahci 依赖了哪些模块，我们使用参数“-F depends”告诉 modinfo 仅显示 ahci 的依赖信息：

```
vita@baisheng:/vita$ modinfo -F depends \
   sysroot/lib/modules/3.7.4/kernel/drivers/ata/ahci.ko
libahci
```

根据 modinfo 的输出，我们可以看到，ahci 模块依赖 libahci 模块。因此，我们首先需要加载 libahci 模块，然后再加载 ahci 模块，如图 4-17 所示。

加载 ahci 模块后，该模块正确识别出了硬盘控制器，并且内核在 devtmpfs 中也建立了对应硬盘的设备节点。

虽然使用 insmod 可以完成加载模块的功能，但是我们发现必须要对模块的依赖关系非常清楚。幸运的是 ahci 只依赖 libahci，而且 libahci 不依赖其他模块。但是如果一个模块依赖多个模块，并且依赖的模块又依赖其他的模块，如此下去，可想而知，加载这样一个模块将是多么复杂。好在 kmod 中提供了另外一个加载 / 卸载模块的工具 modprobe，与 insmod 和 rmmod 相比，modprobe 可以自动加载 / 卸载模块依赖的其他模块，而模块间的依赖关系存储在 modules 目录下的 modules.dep 中。以硬盘驱动这几个模块为例，其在 modules.dep 中的内容如下：

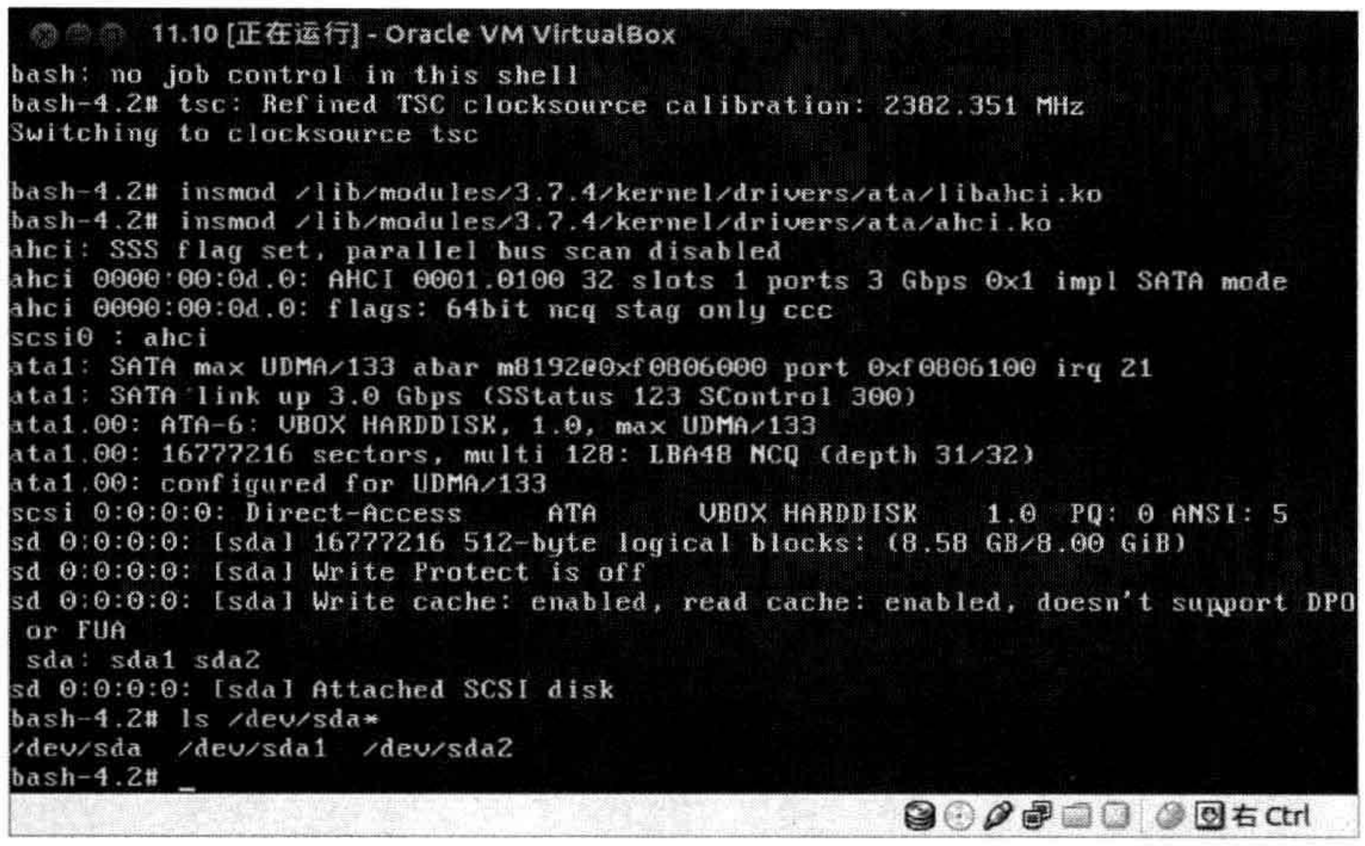


图 4-17 加载 ahci 模块成功

```
kernel/drivers/ata/ahci.ko: kernel/drivers/ata/libahci.ko
kernel/drivers/ata/libahci.ko:
kernel/drivers/ata/ata_piix.ko:
```

该片段表示模块 ahci.ko 依赖 libahci.ko，而模块 libahci.ko 和 ata_piix.ko 不依赖其他模块。

如果模块间依赖关系简单也罢，但是如果比较复杂，那么手动去创建 modules.dep 是不现实的，幸运的是，模块管理工具中也提供了相应的工具创建 modules.dep 文件，这个工具就是 depmod。在安装内核模块时，安装脚本将自动调用这个工具，创建 modules.dep 等文件。

为了加快搜索过程，modules.dep 通常使用更有效率的 Trie 树来组织，并命名为 modules.dep.bin。module-init-tools 中实现的 modprobe 上述两种格式都支持，当然首选使用 modules.dep.bin。但是 kmod 仅支持使用 Trie 树形式存储的 modules.dep.bin。

接下来我们就体验一下使用 modprobe 加载驱动模块。首先需要创建模块依赖关系文件。一般而言，对于通用系统，通常在安装系统时使用 depmod 创建依赖关系文件，然后如果模块有变动，可以使用 depmod 命令更新这些文件。

在这里，在安装内核模块时，安装脚本已经调用 depmod 创建了 modules.dep 和使用 Trie 树组织的 modules.dep.bin，注意需要将使用 Trie 树组织的 modules.dep.bin 复制到 initramfs。当然，如果使用了 module-init-tools 中的模块管理工具，那么这里完全可以体验一下手写 modules.dep 文件。

```
vita@baisheng:/vita$ cp \
   sysroot/lib/modules/3.7.4/modules.dep.bin \
   initramfs/lib/modules/3.7.4/
```

为了验证 modprobe 是否正确加载了模块，可以使用命令 lsmod 查看内核加载的模块。但是 lsmod 是通过 proc 和 sysfs 获取内核信息的，因此，为了使用 lsmod，首先需要挂载 proc 和 sysfs 文件系统。为此，我们需要在 initramfs 的根目录下创建 proc 和 sys 目录作为挂载点：

```
vita@baisheng:/vita/initramfs$ mkdir proc sys
```

同时修改 init 脚本，添加挂载 proc 和 sysfs 文件系统的脚本：

```
#!/bin/bash
echo "Hello Linux!"
export PATH=/usr/sbin:/usr/bin:/sbin:/bin
mount -n -t devtmpfs udev /dev
mount -n -t proc proc /proc
mount -n -t sysfs sysfs /sys
exec /bin/bash
```

重新压缩 initramfs，并将其复制到虚拟机，重新启动系统，使用命令 modprobe 安装 ahci 模块，并使用命令 lsmod 查看内核安装的模块，如图 4-18 所示。

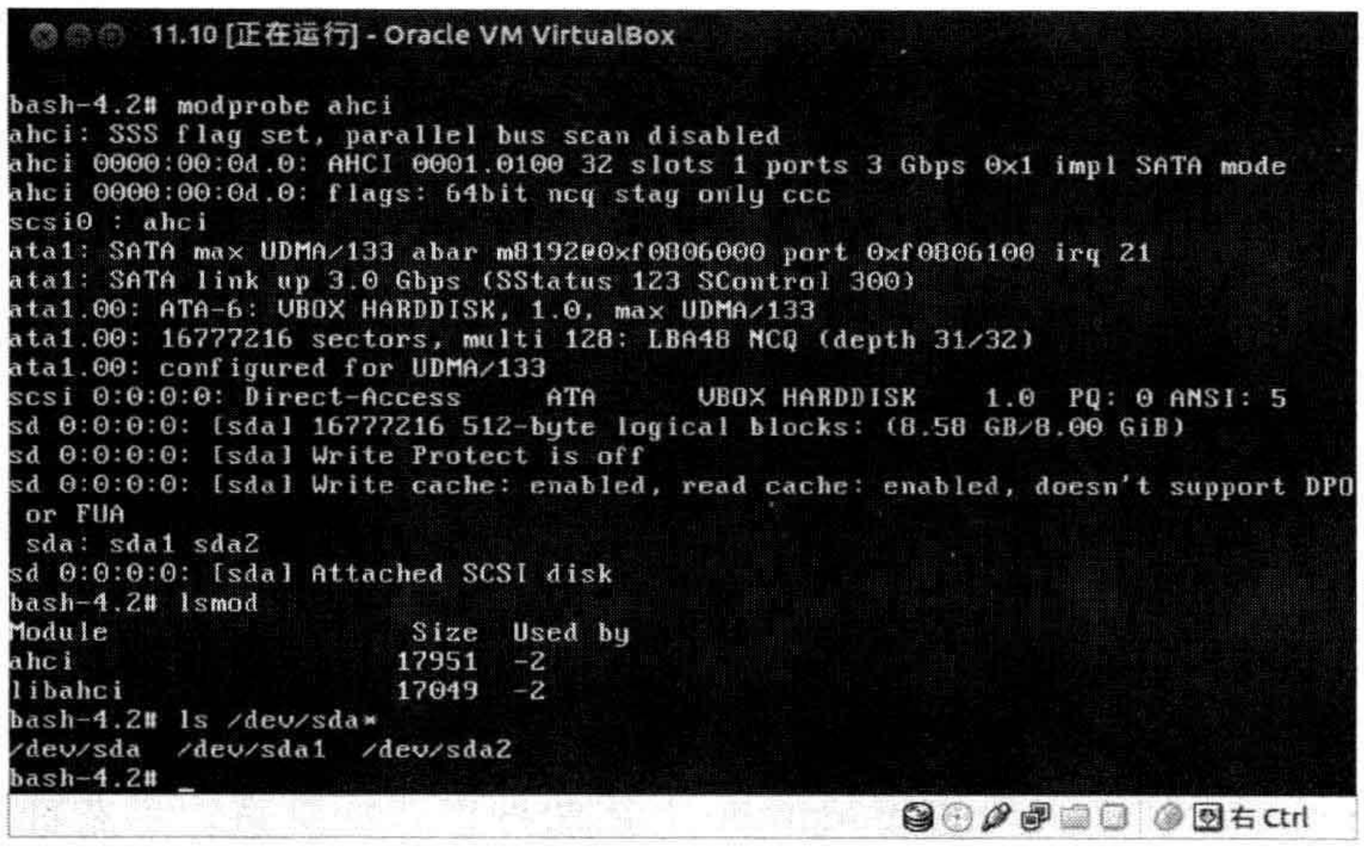


图 4-18 使用 modprobe 加载 ahci 模块

根据 lsmod 的输出可见，虽然我们并没有明确的指示 modprobe 加载模块 libahci，但是 modprobe 根据 modules.dep.bin 中记录的依赖关系，自动加载了 ahci 依赖的模块 libahci。

4.6 自动加载硬盘控制器驱动

在前面，我们以 Intel 的工作在 AHCI 模式下的 SATA 硬盘控制器为例，展示了如何加载硬盘控制器的驱动。但是，除非是为一款特定的嵌入式设备定制的系统，否则，对于一个通用设备来说，比如 PC，我们是不能假定硬件使用的硬盘控制器的。因此，合理的方法应该是根据具体的硬盘控制器加载对应的驱动模块。但是依靠用户自己手动来完成吗？姑且不提是否方便易用，除非专业用户，否则普通用户如何知道应该加载哪些驱动模块呢？

从 2.6 版内核开始，Linux 采用 udev 管理驱动模块的加载以及设备节点的管理。每当内核发现新的设备，便通过 NETLINK 向用户空间发送新设备事件，该事件中记录了设备的相关信息。用户空间的 udev 服务进程收到内核事件后，根据事件中携带的信息，首先判断该

设备的驱动是否已经加载，如果没有，则加载驱动。驱动加载后，内核会再次向用户空间报告发现新设备事件，这时设备已经成功驱动了，并且主次设备号等信息也已经准备好了，udev 收到事件后，或者为设备建立节点，或者执行某些特定的操作。整个过程如图 4-19 所示。

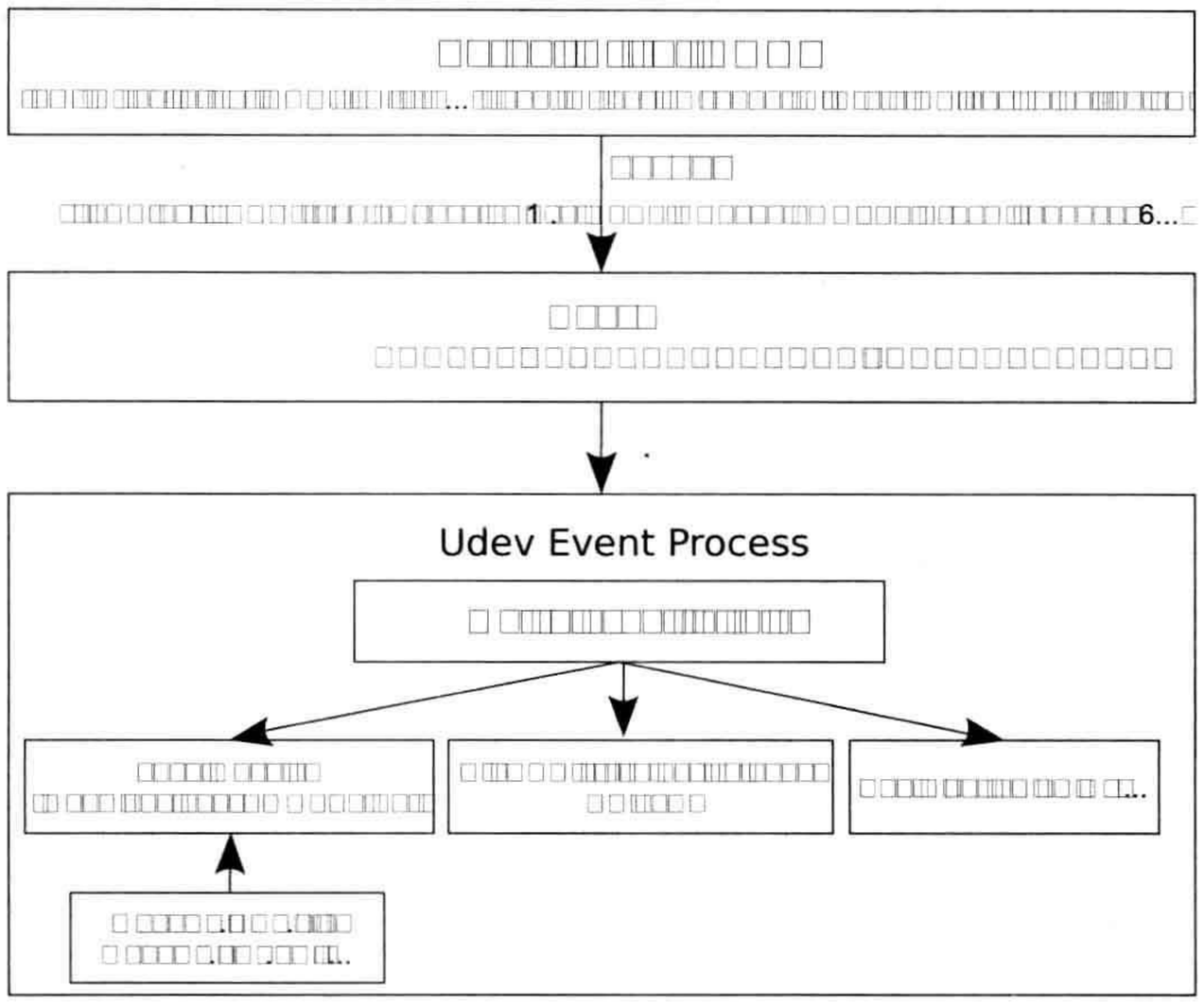


图 4-19　自动加载设备驱动过程

读者可能会有一点疑问：既然有了 devtmpfs，为什么还需要 udev？

首先，也是最重要的一点，devtmpfs 仅是记录了设备驱动注册的节点。udev 除了创建设备节点外，还要负责加载设备驱动。后者是 devtmpfs 所不能实现的，devtmpfs 仅是一个被动的记录数据的文件系统而已。

其次，使用 udev，在发现新设备或者设备发生了更新时，可以有机会执行某些特定的动作。比如在建立新设备时，为设备节点建立额外的符号链接。

下面我们就分别讨论一下上面描述的各个过程。

4.6.1　内核向用户空间发送事件

PC 机上的硬盘控制器，无论是 IDE 接口的，还是 SATA 接口的，一般都是通过 PCI 总线连接到计算机上的。内核在引导时，PCI 子系统将进行初始化，枚举总线上的设备，并尝试为设备匹配驱动；然后将收集到的设备相关信息组织为 uevent 事件；接着调用 kobject_uevent，通过 NETLINK 将组织好的 uevent 发送到用户空间，通知 udev 有新设备了。简单地讲，内核的工作就是探测并收集设备信息，将其包装到 uevent 事件中，然后发送到用户空间。

事实上，无论是发现新的设备，还是有新的驱动载入，抑或是用户向 sysfs 中的 uevent

写入字符串，内核都将调用函数 kobject_uevent 向用户空间发送事件，其代码如下所示：

```
linux-3.7.4/lib/kobject_uevent.c

int kobject_uevent(struct kobject *kobj, ...)
{
    return kobject_uevent_env(kobj, action, NULL);
}

int kobject_uevent_env(struct kobject *kobj, ...)
{
    struct kobj_uevent_env *env;
    ...
    const struct kset_uevent_ops *uevent_ops;
    ...
    /* search the kset we belong to */
    top_kobj = kobj;
    while (!top_kobj->kset && top_kobj->parent)
        top_kobj = top_kobj->parent;
    ...
    kset = top_kobj->kset;
    uevent_ops = kset->uevent_ops;
    ...
    env = kzalloc(sizeof(struct kobj_uevent_env), GFP_KERNEL);
    ...
    retval = add_uevent_var(env, "ACTION=%s", action_string);
    if (retval)
        goto exit;
    retval = add_uevent_var(env, "DEVPATH=%s", devpath);
    if (retval)
        goto exit;
    retval = add_uevent_var(env, "SUBSYSTEM=%s", subsystem);
    ...
    if (uevent_ops && uevent_ops->uevent) {
        retval = uevent_ops->uevent(kset, kobj, env);
        ...
    }
    ...
}
```

结构体 kobj_uevent_env 用来保存收集到的设备相关信息，所以在函数 kobject_uevent_env 中，首先为 kobj_uevent_env 申请了一块内存，即变量 env 指向的内存，用来临时存放准备发送到用户空间的设备相关信息。

然后向该内存中添加了三个默认的变量，包括 ACTION、DEVPATH 和 SUBSYSTEM。其中 ACTION 指的是热插拔的动作，如“add”，“remove”，“change”等。DEVPATH 指的是设备在 sysfs 文件系统中注册的设备路径，比如笔者的硬盘 sda 的 DEVPATH 是“/devices/pci0000:00/0000:00:1f.2/ata1/host0/target0:0:0/0:0:0:0/block/sda”。SUBSYSTEM 一般是指设备所在的总线，比如笔者的硬盘是挂在 PCI 总线上的，因此该变量的值是“pci”。

在 Linux 的设备模型中，除了总线、设备以及驱动这些对象外，还定义了集合，有某些相似特性的 kobject 将被组织到一个集合中。所以我们看到，在函数 kobject_uevent_env 的开

头，寻找硬盘控制器所属的集合，并在向 uevent 中添加了三个默认的变量后，调用硬盘控制器所属的集合的 uevent_ops 中的函数 uevent 继续向 uevent 中追加变量。对于硬盘控制器来说，其所属的集合是 devices_kset，这是 PCI 总线在初始化设备时设定的，相关定义如下：

```
linux-3.7.4/drivers/base/core.c

static const struct kset_uevent_ops device_uevent_ops = {
    .filter =   dev_uevent_filter,
    .name =     dev_uevent_name,
    .uevent =   dev_uevent,
};

static int dev_uevent(struct kset *kset, struct kobject *kobj,
              struct kobj_uevent_env *env)
{
    struct device *dev = kobj_to_dev(kobj);
    int retval = 0;

    /* add device node properties if present */
    if (MAJOR(dev->devt)) {
        const char *tmp;
        const char *name;
        umode_t mode = 0;

        add_uevent_var(env, "MAJOR=%u", MAJOR(dev->devt));
        add_uevent_var(env, "MINOR=%u", MINOR(dev->devt));
        name = device_get_devnode(dev, &mode, &tmp);
        if (name) {
            add_uevent_var(env, "DEVNAME=%s", name);
            kfree(tmp);
            if (mode)
                add_uevent_var(env, "DEVMODE=%#o", mode & 0777);
        }
}

if (dev->type && dev->type->name)
    add_uevent_var(env, "DEVTYPE=%s", dev->type->name);

if (dev->driver)
    add_uevent_var(env, "DRIVER=%s", dev->driver->name);
...
/* have the bus specific function add its stuff */
if (dev->bus && dev->bus->uevent) {
        retval = dev->bus->uevent(dev, env);
        ...
    }
    ...
}
```

dev_uevent 向 uevent 中继续增加一些设备相关的变量，包括设备节点的主次设备号、名称、设备节点的读、写和执行权限、设备的类型以及驱动模块的名称等准备发送到用户空间的变量。在设备枚举阶段，因为设备还没有被驱动，所以这些信息是没有的。只有当设备被

正确地驱动后，内核向用户空间发送的 uevent 中才包含这些信息。

除了设备信息外，设备所属的总线也可能需要向用户空间报告一些设备所在的总线的相关信息。因此，如果设备属于某个总线，函数 dev_uevent 则还要调用设备所属总线的 event 函数。PCI 的设备总线类型 pci_bus_type 及其中的 uevent 函数代码如下：

```
linux-3.7.4/drivers/pci/pci-driver.c

struct bus_type pci_bus_type = {
    .name       = "pci",
    .match      = pci_bus_match,
    .uevent     = pci_uevent,
    ...
};

linux-3.7.4/drivers/pci/hotplug.c

int pci_uevent(struct device *dev, struct kobj_uevent_env *env)
{
    struct pci_dev *pdev;
    ...
    pdev = to_pci_dev(dev);
    ...
    if (add_uevent_var(env, "PCI_CLASS=%04X", pdev->class))
        return -ENOMEM;

    if (add_uevent_var(env, "PCI_ID=%04X:%04X", pdev->vendor,
      pdev->device))
    return -ENOMEM;

if (add_uevent_var(env, "PCI_SUBSYS_ID=%04X:%04X",
      pdev->subsystem_vendor, pdev->subsystem_device))
    return -ENOMEM;

if (add_uevent_var(env, "PCI_SLOT_NAME=%s", pci_name(pdev)))
    return -ENOMEM;

if (add_uevent_var(env,
      "MODALIAS=pci:v%08Xd%08Xsv%08Xsd%08Xbc%02Xsc%02Xi%02x",
       pdev->vendor, pdev->device,
       pdev->subsystem_vendor, pdev->subsystem_device,
       (u8)(pdev->class >> 16), (u8)(pdev->class >> 8),
       (u8)(pdev->class)))
    return -ENOMEM;
    return 0;
}
```

pci_uevent 又向 uevent 中追加了 pci class、vendor id、device id 以及 MODALIAS 等变量，其中 MODALIAS 需要重点关注，其是由设备所在总线、vendor ID、device ID 等相关参数连接而成的一个字符串。在接下来的章节中，读者将看到，用户空间的 udev 恰恰就是根据这个变量为设备匹配驱动模块的。

除了总线外，如果硬盘控制器所属的 class 或者 type 也需要继续向 uevent 中追加变量，

则继续调用硬盘控制器所属的 class 或者 type 中的相应的函数，这里不再继续分析了。最终，内核向用户空间发送的 uevent 事件包含的大致的内容如下，其中不同变量之间使用“\0”进行分隔。

```
"add@/devices/pci0000:00/0000:00:1f.2\0
ACTION=add\0
DEVPATH=/devices/pci0000:00/0000:00:1f.2\0
SUBSYSTEM=pci\0
PCI_CLASS=10601\0
PCI_ID=8086:1C03\0
PCI_SUBSYS_ID=17AA:21CE\0
PCI_SLOT_NAME=0000:00:1f.2\0
MODALIAS=pci:v00008086d00001C03sv000017AAsd000021CEbc01sc06i01 \0
SEQNUM=1206\0"
```

当伴随着热插拔事件一同发往用户空间的变量准备完毕后，kobject_uevent_env 使用内核和用户空间的通信协议 NETLINK 向用户空间报告事件，代码如下：

```
linux-3.7.4/lib/kobject_uevent.c

int kobject_uevent_env(struct kobject *kobj, ...)
{
    ...
        struct sk_buff *skb;
    ...
    skb = alloc_skb(len + env->buflen, GFP_KERNEL);
    if (skb) {
        char *scratch;

        /* add header */
        scratch = skb_put(skb, len);
        sprintf(scratch, "%s@%s", action_string, devpath);

        /* copy keys to our continuous event payload buffer */
        for (i = 0; i < env->envp_idx; i++) {
            len = strlen(env->envp[i]) + 1;
            scratch = skb_put(skb, len);
            strcpy(scratch, env->envp[i]);
        }

        NETLINK_CB(skb).dst_group = 1;
        retval = netlink_broadcast_filtered(uevent_sock, skb,
                                            0, 1, GFP_KERNEL,
                                            kobj_bcast_filter,
                                            kobj);
                ...
            }
        ...
}
```

kobject_uevent_env 申请了一个结构体 sk_buff 类型的变量 skb，这个 skb 就是用来封装报文的。报文以形如“ACTION@DEVPATH”（如“add@/devices/pci0000:00/0000:00:1f.2”）

的格式开头，紧接着的消息体中封装的就是前面收集到的存储在变量 env 中的变量。

至此，对于加载硬盘控制器驱动这个任务，内核已经完成了它的使命：PCI 子系统获取硬盘控制器的信息，并将其通过 NETLINK 抛到了用户空间。接下来，该用户空间的 udev 出场了。

4.6.2 udev 加载驱动和建立设备节点

前面我们探讨了内核向用户空间报告 uevent 事件的过程。这一节，我们来讨论 udev 是如何根据内核报告的 uevent 事件加载硬盘控制器驱动以及建立设备节点的。

udev 是用户空间动态管理设备的机制，包括加载驱动、管理设备节点等。udev 机制的核心是其服务进程 udevd。当启动过程进入用户空间阶段后，udevd 将被启动。udevd 启动后，首先读取并分析所有的规则文件，并将其缓存在内存中。一般情况下，系统默认的规则文件存放在 /lib/udev/rules.d 目录下，用户自定义的规则存放在 /etc/udev/rules.d 目录下。每当动态地增加、删除或者改变某个规则文件时，udevd 将更新其缓存在内存中的规则。然后，udevd 通过 NETLINK 协议，监听并处理来自内核的 uevent 事件。每当 udevd 收到一个内核的 uevent，udevd 均创建一个单独的子进程处理 uevent。

对于每个内核报告的 uevent，udevd 根据 uevent 中的变量逐个匹配规则。规则文件通常以数字开头，数字小的先进行匹配。若每个规则文件中包含若干个规则，同一规则不允许断行，每个规则至少包含一个 key-value 对，每个 key-value 对之间使用逗号分隔。可以将规则理解为由匹配条件和赋值动作组成，当所有的匹配条件都满足后，赋值动作就会发生。规则中可以加载驱动模块；规定如何给设备接点命名、建立符号连接；设备连接和断开时分别执行指定的程序等。

前面我们看到内核在发现新设备时会将设备的一些信息通过 NETLINK 发送到用户空间，udev 接收到事件后，如果发现设备尚未被驱动，将尝试加载驱动模块。那么 udev 如何确定设备对应的驱动模块呢？一般而言，根据设备的 vender ID 和 device ID 就可以标识一类设备，当然有的也需要根据 subvendor ID 和 subdevice ID 进一步细分。而在驱动代码中，恰恰使用这些设备信息明确声明了其可以支持的设备。以驱动 AHCI 模式的 SATA 硬盘控制器驱动为例：

```
linux-3.7.4/vi drivers/ata/ahci.c

static const struct pci_device_id ahci_pci_tbl[] = {
    /* Intel */
    { PCI_VDEVICE(INTEL, 0x2652), board_ahci }, /* ICH6 */
    { PCI_VDEVICE(INTEL, 0x2653), board_ahci }, /* ICH6M */
    ...
    { PCI_VDEVICE(INTEL, 0x1c03), board_ahci }, /* CPT AHCI */
    ...
    /* AMD */
    { PCI_VDEVICE(AMD, 0x7800), board_ahci }, /* AMD Hudson-2 */
```

```
    ...
    /* NVIDIA */
    { PCI_VDEVICE(NVIDIA, 0x044c), board_ahci_mcp65 },   /* MCP65 */
    ...
};
```

ID table 中的每一项表示该驱动支持的一类设备，根据 PCI_VDEVICE 的定义：

```
#define PCI_VDEVICE(vendor, device)     \
    PCI_VENDOR_ID_##vendor, (device),   \
    PCI_ANY_ID, PCI_ANY_ID, 0, 0
```

以 ahci_pci_tbl 中的第一项为例，该项声明了该驱动支持 vender ID 为 PCI_VENDOR_ID_INTEL（0x8086），device ID 为 0x2652，subvendor ID、subdevice ID 为任意的 Intel SATA 控制器。

内核将 ID table 中的每一项中的信息按照一定的格式组合起来，作为驱动的一个别名。这些别名存储在编译好的驱动模块中，模块安装后，需要使用工具 depmod 将其提取出来并存储在 /lib/modules/'uname -r' 目录下的 modules.alias.bin/modules.alias 中，如同前面讨论的 modules.dep 和 modules.dep.bin 的关系一样，modules.alias.bin 与 modules.alias 完全相同，只不过 modules.alias.bin 是为了加快搜索速度采用 Trie 树存储的。很多读者可能会说，编译安装模块时从来没有显示执行 depmod 啊，那是因为 make 等安装脚本已经替我们调用了这个命令。

我们可以使用工具 modinfo 来查看驱动模块的相关信息，下面是查看驱动模块 ahci 的别名信息。

```
vita@baisheng:/vita$ modinfo -F alias \
   sysroot/lib/modules/3.7.4/kernel/drivers/ata/ahci.ko
pci:v*d*sv*sd*bc01sc06i01*
pci:v00001B21d00000612sv*sd*bc*sc*i*
pci:v00001B21d00000611sv*sd*bc*sc*i*
...
pci:v00008086d00002653sv*sd*bc*sc*i*
pci:v00008086d00002652sv*sd*bc*sc*i*
```

上述输出表示驱动模块 ahci 可以驱动别名为“pci:v*d*sv*sd*bc01sc06i01*”、“pci:v00001B21d00000612sv*sd*bc*sc*i*”等的设备，其中“*”表示可以匹配任意 ID。

通过 depmod 生成的典型的 modules.alias 文件如下所示：

```
alias pci:v00008086d00001C03sv*sd*bc*sc*i* ahci
alias pci:v00008086d00001C02sv*sd*bc*sc*i* ahci
...
alias pci:v00001101d00001622sv*sd*bc*sc*i* sata_inic162x
alias pci:v00001095d00003531sv*sd*bc*sc*i* sata_sil24
```

显然，这个文件就是简单地将别名和驱动名称对应起来。

前面讨论内核向用户空间发送 uevent 时，我们看到，内核将在 uevent 的消息体中封装

一个变量 MODALIAS，其值形如“pci:v00008086d00001C03sv000017AAsd000021CEbc01sc06i01”。看上去是不是与驱动的别名一致？没错，内核的设计者们设计了这个机制，内核创建变量 MODALIAS 和模块创建别名采用相同的算法。当 udevd 收到内核 uevent 后，从 uevent 中提取这个字符串，然后将这个字符串作为 modprobe 的参数。modprobe 首先查找文件 modules.alias.bin，将该别名对应的模块找到。以该别名为例，显然其会与上面 modules.alias 文件片断中的第一行匹配成功，而该行明确表明该别名对应的驱动模块是 ahci，因此，modprobe 将加载模块 ahci。

udev 设计了规则文件 80-drivers.rules 用来描述如何加载驱动模块，以 v173 版本的 udev 的 80-drivers.rules 为例：

```
udev-173/rules/rules.d/80-drivers.rules

ACTION=="remove", GOTO="drivers_end"

DRIVER!="?*", ENV{MODALIAS}=="?*", RUN+="/sbin/modprobe -bv
   $env{MODALIAS}"
...
LABEL="drivers_end"
```

我们先来看第一个规则，该规则表示如果 uevent 的动作是删除设备（remove），则忽略下面所有规则，什么也不用做。

第二个规则包含两个匹配条件，一个赋值动作。其中“?”匹配一个字符，“*”匹配 0 或多个字符。这个规则表达的含义是：当设备还没有加载驱动，即环境变量 DRIVER 的值为空，并且环境变量 MODALIAS 的值非空，那么调用 modprobe 加载驱动。我们看到这里加载模块的方式就是采用我们前面讨论的别名的方式。这里追加到环境变量 RUN 中的程序，如果不给出绝对路径，将在 /lib/udev 目录下寻找，如果这个程序不在 /lib/udev 目录下，必须给出绝对路径。

80-drivers.rules 也会包含对个别特殊 subsystem 类型的设备的特殊处理，我们这里不作过多讨论。

一旦驱动被正确加载，并且设备需要在用户空间建立设备节点，那么内核向用户空间再次报告的 uevent 中会包含创建设备节点需要的主次设备号以及节点的名称等环境变量，类似于下面的这个示例 uevent 事件。事实上，在发现设备、加载驱动过程中，内核一般会多次向用户空间报告 uevent 事件，只有设备和驱动匹配成功后发送的事件中才会包含主次设备号等变量。

```
    "add@/devices/pci0000:00/0000:00:1f.2/ata1/host0/target0:0:0/0:0:0:0/block/sda/
sda1\0
    ACTION=add\0
    DEVPATH=/devices/pci0000:00/0000:00:1f.2/ata1/host0/target0:0:0/0:0:0:0/block/
sda/sda1\0
    SUBSYSTEM=block\0
    DEVNAME=sda1\0
```

```
DEVTYPE=partition\0
MAJOR=8\0
MINOR=1\0
SEQNUM=1204\0"
```

该消息中，内核为 udev 创建设备节点提供了必要的变量，包括主设备号为 8，次设备号为 1，内核提供的该设备节点的名字为 sda1。当 udevd 收到的 uevent 消息中，如果 uevent 的变量中包含设备号，则使用系统调用 mknod 创建设备节点。

4.6.3 处理冷插拔设备

前面我们讨论了动态加载驱动的整个过程。但是不知道读者想过没有，对于磁盘这种非热插拔设备，如果驱动没有编译进内核，那么当内核引导枚举设备时，系统运行在内核空间，尚未进入用户空间，更谈不上启动用户空间的 udev 服务了，因此内核发送到用户空间的 uevent 自然会被丢掉，更别提加载硬盘驱动模块和建立设备节点了。

为了解决这个问题，开发人员基于 sys 文件系统设计了一种巧妙的机制。在 Linux 操作系统进入用户空间，udevd 启动后，通过 sys 文件系统请求内核重新发出 uevent。此时 udevd 已经启动了，就会收到 uevent，然后结合这些事件和规则，完成驱动的加载、设备节点的建立等。我们可以将这个过程看作是内核和 udev 导演的一出戏，对于冷插拔的设备，模拟了一遍热插拔的过程。

下面我们简单探讨一下这个机制的原理。

当新设备注册时，内核将调用 device_create_file 在 sys 文件系统中为设备注册一个名字为 uevent 的文件，当用户空间的程序读取该文件时，内核将调用函数 show_uevent 处理用户的读操作，而当用户空间的程序向该文件写入时，内核将调用函数 store_uevent 处理用户的写操作。我们以函数 store_uevent 为例，看看内核是如何处理用户的写操作的。函数 store_uevent 代码如下：

```
linux-3.7.4/drivers/base/core.c

static ssize_t store_uevent(struct device *dev, struct
    device_attribute *attr, const char *buf, size_t count)
{
    enum kobject_action action;

    if (kobject_action_type(buf, count, &action) == 0)
        kobject_uevent(&dev->kobj, action);
    else
        dev_err(dev, "uevent: unknown action-string\n");
    return count;
}
```

store_uevent 的参数 buf 指向复制自用户空间的用户写入的字符串。函数 kobject_action_type 根据 buf 中的字符串，来决定发送给用户空间的 uevent 的类型。写入的字符串和发送的事件类型间的对应关系的代码如下所示。

```
linux-3.7.4/include/linux/kobject.h

enum kobject_action {
    KOBJ_ADD,
    KOBJ_REMOVE,
    KOBJ_CHANGE,
    KOBJ_MOVE,
    KOBJ_ONLINE,
    KOBJ_OFFLINE,
    KOBJ_MAX
};

linux-3.7.4/lib/kobject_uevent.c

static const char *kobject_actions[] = {
    [KOBJ_ADD] =        "add",
    [KOBJ_REMOVE] =     "remove",
    [KOBJ_CHANGE] =     "change",
    [KOBJ_MOVE] =       "move",
    [KOBJ_ONLINE] =     "online",
    [KOBJ_OFFLINE] =    "offline",
};
```

也就是说，当用户空间的程序向该属性文件写入字符串“add”时，函数 kobject_action_type 认为用户空间的程序要求 KOBJ_ADD 类型的事件，于是调用 kobject_uevent 向用户空间发送 KOBJ_ADD 类型的 uevent。

利用这种机制，我们可以在用户空间的 udev 服务程序启动后，向所有设备的属性文件 uevent 写入“add”字符串，请求内核重新发送一遍 KOBJ_ADD 事件，模拟一遍热插拔动作。如此，udevd 就可以收到这些事件，完成驱动加载、设备节点创建等工作。

为此，udev 提供了一个管理工具 udevadm，我们可以使用这个工具请求内核重新发送设备相关事件。假设请求内核对全部设备模拟一遍热插拔，即重新发送事件 KOBJ_ADD，则使用如下命令：

```
udevadm trigger --action=add
```

我们来简单地看一下这个命令背后的代码：

```
udev-173/udev/udevadm-trigger.c

const struct udevadm_cmd udevadm_trigger = {
    .name = "trigger",
    .cmd = adm_trigger,
    .help = "request events from the kernel",
};

static int adm_trigger(struct udev *udev, int argc, char *argv[])
{
    ...
    switch (device_type) {
    ...
```

```
        case TYPE_DEVICES:
            udev_enumerate_scan_devices(udev_enumerate);
            exec_list(udev_enumerate, action);
        ...
        }
        ...
}

static void exec_list(struct udev_enumerate *udev_enumerate,
    const char *action)
{
    struct udev *udev = udev_enumerate_get_udev(udev_enumerate);
    struct udev_list_entry *entry;

    udev_list_entry_foreach(entry,
            udev_enumerate_get_list_entry(udev_enumerate)) {
        char filename[UTIL_PATH_SIZE];
        int fd;
        ...
        util_strscpyl(filename, sizeof(filename),
            udev_list_entry_get_name(entry), "/uevent", NULL);
        fd = open(filename, O_WRONLY);
        ...
        if (write(fd, action, strlen(action)) < 0)
            info(udev, "error writing '%s' to '%s': %m\n", action,
            filename);
        close(fd);
    }
}
```

根据上面代码可见，udevadm 的 trigger 命令对应的函数是 adm_trigger。当用户请求内核重新发送设备相关的事件时，adm_trigger 首先调用 udev_enumerate_scan_devices 在 sys 文件系统中寻找设备，使用 udevadm 的 trigger 命令时我们可以指定一些属性，匹配特定的设备。但是无论如何，会有多个设备满足匹配条件的情况，比如我们上面的命令，没有任何限制条件，那么内核将匹配所有设备。于是 udev 在结构体 udev_enumerate 中设计了一个链表，udev_enumerate_scan_devices 将找到的所有设备连接到结构体 udev_enumerate 中的设备链表中。

然后，adm_trigger 调用函数 exec_list 遍历这个链表，向这些设备在 sys 文件系统中注册的属性文件 uevent 写入用户请求内核重新发送的事件类型对应的字符串。比如，如果请求内核发送 KOBJ_ADD 类型的 uevent，则写入字符串“add”；如果请求内核发送 KOBJ_CHANGE 类型的 uevent，则写入字符串“change”，等等。

4.6.4 编译安装 udev

前面几节我们探讨了相关的工作原理，从本节开始，我们开始动手实践驱动模块的自动加载过程。

因为系统启动程序 systemd 和 udev 之间的依赖关系，为了方便开发编译，所以社区中已经将 udev 和 systemd 合并了。但是本书中我们不讨论 systemd，为了减少干扰，本书中使用

尚未合并前的 udev。合并前后，udev 本质上并没有什么差别。

使用如下命令编译安装 udev（我们采用的版本是 udev 173）：

```
vita@baisheng:/vita/build$ tar xvf ../source/udev-173.tar.xz
vita@baisheng:/vita/build/udev-173$ ./configure --prefix=/usr \
   --sysconfdir=/etc --sbindir=/sbin --libexecdir=/lib/udev \
   --disable-hwdb --disable-introspection \
   --disable-keymap --disable-gudev
vita@baisheng:/vita/build/udev-173$ make
vita@baisheng:/vita/build/udev-173$ make install
```

指定 --libexecdir 的目的是告诉安装脚本将 udev 的规则文件以及一些 helper 程序安装在 /lib/udev 目录下。我们使用 --disable 选项禁掉 udev 不必要的一些特性，也减少了 udev 对其他库的依赖和系统的复杂性。

接下来将 udevd、udevadm 以及相关的规则文件安装到 initramfs 中：

```
vita@baisheng:/vita$ ldd sysroot/sbin/udevd
        librt.so.1 => /vita/sysroot/lib/librt.so.1
        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

vita@baisheng:/vita$ ldd sysroot/sbin/udevadm
        librt.so.1 => /vita/sysroot/lib/librt.so.1
        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

vita@baisheng:/vita$ cp sysroot/sbin/udevd initramfs/bin/
vita@baisheng:/vita$ cp sysroot/sbin/udevadm initramfs/bin/
vita@baisheng:/vita$ mkdir -p initramfs/lib/udev/rules.d
vita@baisheng:/vita$ cp \
    sysroot/lib/udev/rules.d/80-drivers.rules \
    initramfs/lib/udev/rules.d/
```

udevd 和 udevadm 依赖的库在前面已经复制到 initramfs 中了，所以只需将 udevd、udevadm 和加载驱动的规则，即 80-drivers.rules，复制到 initramfs 即可。

4.6.5 配置内核支持 NETLINK

内核与 udevd 通过 Unix Domain Sockets 使用 NETLINK 协议进行通信，因此，我们需要配置内核支持 Unix Domain Sockets 与 NETLINK 协议。配置步骤如下：

1）执行 make menuconfig，出现如图 4-20 所示的界面。

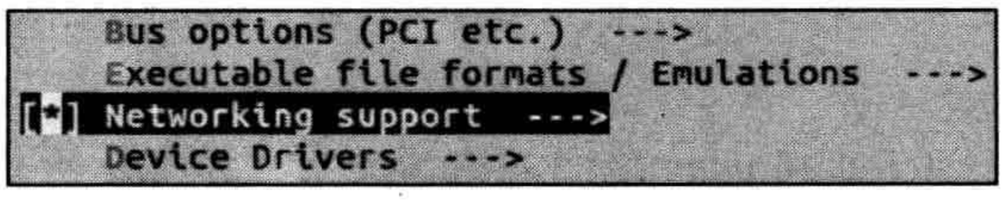


图 4-20 配置内核支持 Unix domain sockets 和 NETLINK 协议（1）

2）在图 4-20 中，选择“Networking support”，出现如图 4-21 所示的界面。

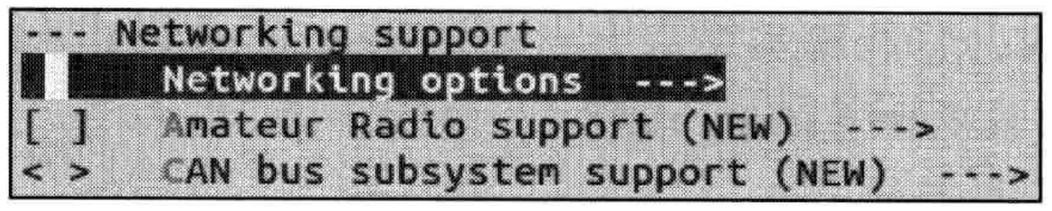


图 4-21 配置内核支持 Unix domain sockets 和 NETLINK 协议（2）

3）在图 4-21 中，选择“Networking options”，出现如图 4-22 所示的界面。

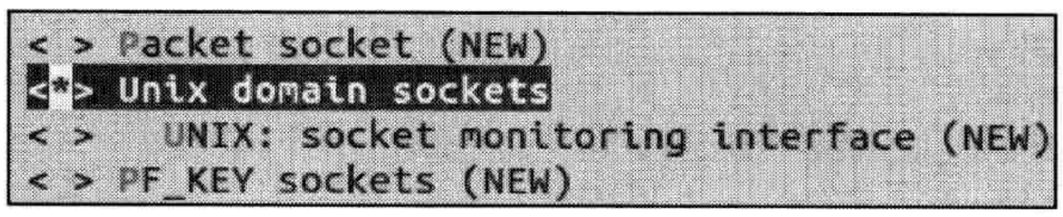


图 4-22 配置内核支持 Unix domain sockets 和 NETLINK 协议（3）

4）在图 4-22 中，选中“Unix domain sockets”。

对于，NETLINK 协议，只要配置内核支持网络，NETLINK 协议默认就被支持。通过 net 目录下的 Makefile 可以清楚地看到这一点：

```
linux-3.7.4/net/Makefile

obj-$(CONFIG_NET)        += ethernet/ 802/ sched/ netlink/
```

4.6.6 配置内核支持 inotify

因为 udev 使用 inotify 机制监测 udev 的规则文件是否发生变化，所以配置内核使其支持 inotify 机制，否则 udevd 将因为初始化 inotify 失败而退出。配置过程如下：

1）执行 make menuconfig，出现如图 4-23 所示的界面。

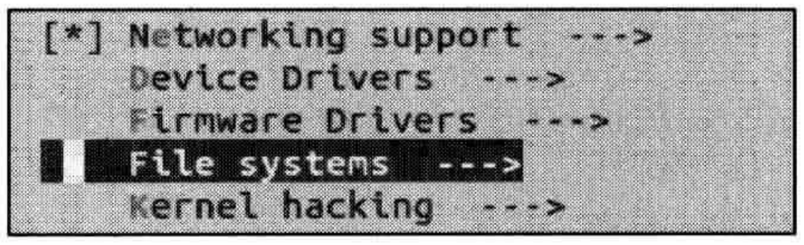


图 4-23 配置内核支持 inotify（1）

2）在图 4-23 中，选择“File systems”，出现如图 4-24 所示的界面。

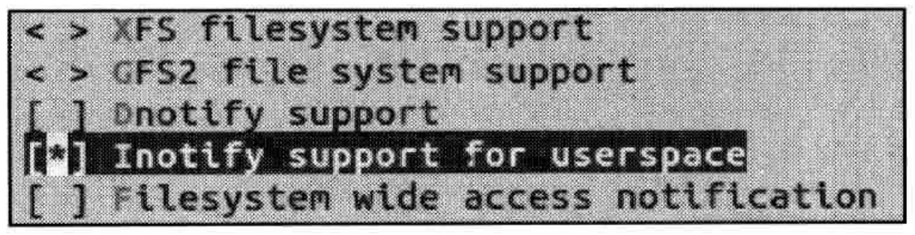


图 4-24 配置内核支持 inotify（2）

3）在图 4-24 中，选中“Inotify support for userspace”。

4.6.7 安装 modules.alias.bin 文件

在安装内核模块时，安装脚本最后会自动调用 depmod 创建 modules.alias.bin/modules.alias 文件。我们直接将其复制到 initramfs 即可：

```
vita@baisheng:/vita$ cp \
   sysroot/lib/modules/3.7.4/modules.alias.bin \
   initramfs/lib/modules/3.7.4/
```

如果你在某些特殊情况下，需要用手动执行 depmod 创建 modules.alias.bin、modules.dep.bin 等文件，相应命令如下：

```
vita@baisheng:/vita$ depmod -b /vita/sysroot/ 3.7.4
```

下面我们验证一下 modules.alias.bin 是否可以正确工作。我们需要安装两个工具：一个是 lspci，这个工具用来运行在目标系统上，查看硬盘设备在 PCI 总线上的位置，包括设备所在的总线、设备号等，这个工具在软件包 pciutils 中；另外一个是 coreutils 中的工具 cat，其已经编译并且安装在 /vita/sysroot 下了，我们将其直接复制到 initramfs 即可。

我们首先编译安装 lspci：

```
vita@baisheng:/vita/build$ tar \
   xvf ../source/pciutils-3.1.10.tar.xz
vita@baisheng:/vita/build/pciutils-3.1.10$ make PREFIX=/usr \
   ZLIB=no SHARED=yes PCI_COMPRESSED_IDS=0 all
vita@baisheng:/vita/build/pciutils-3.1.10$ make PREFIX=/usr \
   ZLIB=no SHARED=yes PCI_COMPRESSED_IDS=0 install
```

将 lspci 及其依赖的库安装到 initramfs，命令如下：

```
vita@baisheng:/vita$ ldd sysroot/usr/sbin/lspci
        libpci.so.3 => /vita/sysroot/usr/lib/libpci.so.3
        libc.so.6 => /vita/sysroot/lib/libc.so.6

vita@baisheng:/vita$ ldd sysroot/usr/lib/libpci.so.3
        libresolv.so.2 => /vita/sysroot/lib/libresolv.so.2
        libc.so.6 => /vita/sysroot/lib/libc.so.6

vita@baisheng:/vita$ ldd sysroot/lib/libresolv.so.2
        libc.so.6 => /vita/sysroot/lib/libc.so.6

vita@baisheng:/vita$ cp sysroot/usr/sbin/lspci initramfs/bin/
vita@baisheng:/vita$ cp -d sysroot/usr/lib/libpci.so.3* \
   initramfs/lib/
vita@baisheng:/vita$ cp -d sysroot/lib/libresolv* initramfs/lib/
```

lspci 将依次尝试通过 sysfs 文件系统、proc 文件系统以及直接访问端口的方式列出 PCI 总线上的设备。为了便于人们理解，社区中维护了一个 pci 数据库 pci.ids，该数据库中记录了 ID 到设备信息的映射。当 lspci 查找到设备 ID 时，其使用设备 ID 到 pci.ids 中去匹配设备信息。因此除了安装 lspci 及依赖的库外，我们还需要安装 pci 数据库 pci.ids，pci.ids 已经被

包含在软件包 pciutils 中，并且在安装 lspci 时已经安装到目标系统的根文件系统下，我们将其复制到 initramfs 中。

```
vita@baisheng:/vita$ mkdir -p initramfs/usr/share
vita@baisheng:/vita$ cp sysroot/usr/share/pci.ids \
   initramfs/usr/share/
```

coreutils 中的 cat 已经编译并且安装在 /vita/sysroot 下了，我们直接复制到 initramfs 即可。

```
vita@baisheng:/vita$ ldd sysroot/usr/bin/cat
        libc.so.6 => /vita/sysroot/lib/libc.so.6
vita@baisheng:/vita$ cp sysroot/usr/bin/cat initramfs/bin/
```

使用支持 NETLINK 和 inotity 的内核以及新的 initramfs 更新 vita 系统，重启后运行 lspci，运行结果如图 4-25 所示。根据 lspci 的输出可见，SATA 控制器挂在总线号为 0x00 的 PCI 总线上，设备号为 0x0d。

图 4-25　硬盘控制器的 uevent 中的环境变量

根据总线号和设备号就可以确定 SATA 控制器在 sysfs 文件系统中的路径。我们使用命令 cat 将 uevent 中的相关变量读出，根据输出结果可见，变量 MODALIAS 的值为“pci:v00008086d00002829sv00000000sd00000000bc01sc06i01”。我们使用这个 MODALIAS 的值加载模块，如图 4-26 所示。

根据输出的信息，我们清楚地看到，使用模块的别名，模块也被正确加载了，说明 modules.alias.bin 文件工作正常。事实上，通过文件 modules.alias.bin 中的别名和 MODALIAS 的对应关系，modprobe 将如下命令：

```
modprobe pci:v00008086d00002829sv00000000sd00000000bc01sc06i01
```

转换为了：

```
modprobe ahci
```

图 4-26 通过环境变量 MODALIAS 加载驱动

4.6.8 启动 udevd 和模拟热插拔

现在对于自动加载硬盘控制器驱动来说，是万事俱备，只欠东风了，让我们来扣响扳机。修改 init，在其中启动 udevd，并使用 udevadm 对冷插拔设备模拟热插拔。另外，udevd 需要保存某些运行时的信息，因此，我们需要建立 run 目录：

```
vita@baisheng:/vita$ mkdir initramfs/run
```

因为这个目录也是保存运行时信息的，关机后不再需要保存，因此我们也使用相对高效的基于内存的文件系统。修改后的 init 文件如下：

```
/vita/initramfs/init

#!/bin/bash
echo "Hello Linux!"
export PATH=/usr/sbin:/usr/bin:/sbin:/bin
mount -n -t devtmpfs udev /dev
mount -n -t proc proc /proc
mount -n -t sysfs sysfs /sys
mount -n -t ramfs ramfs /run
udevd --daemon
udevadm trigger --action=add
udevadm settle
exec /bin/bash
```

init 启动了 udev 的服务进程 udevd，然后使用命令 udevadm 遍历 sysfs 中的设备，向这些设备在 sysfs 文件系统中的文件 uevent 写入“add”字符串，请求内核重新发送 KOBJ_ADD 事件，相当于模拟了一次热插拔。

udevd 收到硬盘控制器的 uevent 后，将加载硬盘控制器驱动，并创建设备节点。当然

devtmpfs 也会创建设备节点，但是 udevd 与 devtmpfs 并不矛盾，udevd 可以在 devtmpfs 上进行用户空间的各种修饰。

命令“udevadm settle”的目的是等待 udevd 处理完内核向用户空间发送的 uevent 后再继续向下执行。否则，如果这里不进行等待，后续的操作有可能发生错误。举个例子，假如在 udevd 正在调用 modprobe 加载硬盘驱动模块时，init 后续的脚本可能已经并行地开始挂载根文件系统了，但是此时设备尚未被驱动，更别提设备节点了，所以挂载将会失败。

重新压缩 initramfs，更新到 vita 系统，重启系统。我们来检查一下硬盘控制器是否正确加载，如图 4-27 所示。通过 lsmod 和查看设备节点，显然硬盘控制器驱动已经成功自动加载。

图 4-27 查看加载模块以及建立的硬盘设备节点

4.7 挂载并切换到根文件系统

截止到目前，系统一直在使用 initramfs 作为临时的根文件系统，initramfs 的主要目的之一就是辅助系统顺利地切换到真正的根文件系统。既然现在已经正确的驱动了硬盘，那么接下来，我们就切换到硬盘上的真正的根文件系统。

4.7.1 挂载根文件系统

首先我们需要确定根文件系统所在的物理介质。以存储器为硬盘为例，需要确定文件系统储存在硬盘的哪个分区。内核在引导时已经将文件系统所在的介质等相关参数从 GRUB 复制到了内核中，所以我们现在可以通过内核获取这个参数。谈到用户空间与内核的通信，读者一定想到了 proc 与 sysfs 文件系统，接下来我们在 init 程序中通过 proc 文件系统取得文件系统所在的介质。

在 GRUB 的 cmdline 中，我们使用了“root=/dev/sda2”指定文件系统所在的介质，因此我们截取“root=”后面的值，将其保存在变量 ROOT 中，供后面挂载使用。

一般在准备挂载文件系统之前，将使用 fsck 检查文件系统。如果文件系统中存在错误，则试图修复。这个过程要求文件系统没有被挂载或者只能以只读方式挂载。因此，一般首先以只读方式 (ro) 挂载根文件系统，然后执行 fsck 检查修复后，再重新以读写方式（rw）挂载。这也是大家看到的在 GRUB 的配置文件 grub.cfg 中，内核的命令行参数要指定 ro 的原因。但是也不排除某些系统在启动时略过 fsck 的步骤，直接将文件系统以读写方式挂载。因此，在挂载前，我们首先查看内核命令行的这个参数。如果没有指定，默认我们以只读方式挂载。

对于文件系统的类型，可以通过 udev 来获取，但是这里我们偷个懒，直接让 mount 来猜测。在 init 中增加如下使用黑体标识的脚本将真正的根文件系统挂载到 /root 目录下，当然，不一定是挂载到 /root 目录下，也可以使用除了“/”外的任何目录作为挂载点。

```
/vita/initramfs/init

#!/bin/bash
echo "Hello Linux!"
export PATH=/usr/sbin:/usr/bin:/sbin:/bin
export ROOTMNT=/root
export ROFLAG=-r
mount -n -t devtmpfs udev /dev
mount -n -t proc proc /proc
mount -n -t sysfs sysfs /sys
mount -n -t ramfs ramfs /run
udevd --daemon
udevadm trigger --action=add
udevadm settle

for x in $(cat /proc/cmdline); do
    case $x in
    root=*)
        ROOT=${x#root=}
        ;;
    ro)
        ROFLAG=-r
        ;;
    rw)
        ROFLAG=-w
        ;;
    esac
done

mount ${ROFLAG} ${ROOT} ${ROOTMNT}

exec /bin/bash
```

使用修改的 initramfs 重新启动 vita 系统，查看 mount 的输出和 /root 目录下的内容，确定真正的根文件系统是否已经挂载，如图 4-28 所示。根据 mount 命令的输出可见，分区

“/dev/sda2”确实被挂载到了“/root”目录下，该目录下也不再是个空目录了，是根文件系统的内容。

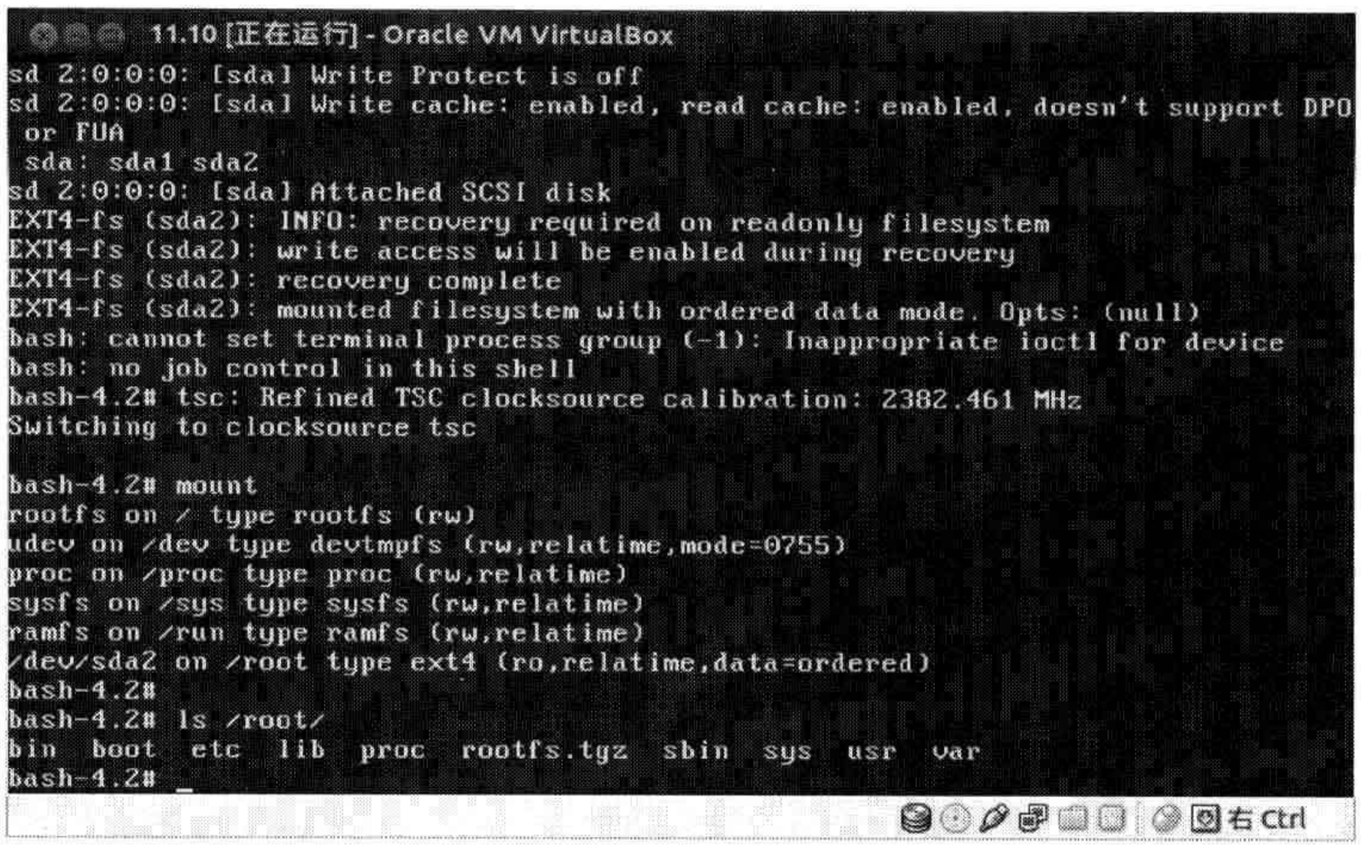


图 4-28 自动挂载根文件系统成功

4.7.2 切换到根文件系统

真正的根文件系统已经被挂载了，我们接下来就要切换到真正的根文件系统。

initramfs 中的这个 init 脚本也完成了它的历史使命，该退出舞台了。系统的第一个进程应该使用根文件系统中的一个程序了。无论是 SystemV 还是 systemd，都会提供一个 init 程序，通常这个程序都是使用二进制格式的。但是这里，我们为了不把事情搞得太复杂，真正的用户空间的 init 程序依然使用 shell 脚本。一般而言，init 存储在 /sbin 目录下，所以需要在根文件系统中建立 /sbin 目录。

```
vita@baisheng:/vita/rootfs$ mkdir sbin
```

init 脚本简单的执行一个交互式的 bash。

```
/vita/rootfs/sbin/init:

#!/bin/bash
exec /bin/bash
```

最后注意将该程序加上可执行权限

```
/vita/rootfs/sbin$ chmod a+x init
```

根文件系统准备好后，接下来开始向根文件系统切换，步骤如下：

1）删除 rootfs 文件系统中不再需要的内容，释放内存空间。

现在挂载在“/”下的 rootfs 中的内容是 initramfs 解压来的，在我们准备把磁盘文件系统

挂载到“/”前，需要删除 rootfs 中的内容，以释放其占用的内存空间。但是，在删除 rootfs 前，我们需要：

- 停止正在运行的进程，这里就是 udevd；
- 将 /dev、/run、/proc 和 /sys 目录移动到真正的文件系统上。因此，需要在根文件系统上建立如下目录：

```
vita@baisheng:/vita/rootfs$ mkdir sys proc dev run
```

2）将根文件系统从“/root”移动“/”下。

3）更改进程的文件系统 namespace，使其指向真正的根文件系统。因为当前进程就是进程 1，而后续进程都是从进程 1 复制的，所以后续进程的文件系统的 namespace 自然就是使用的真正的根文件系统。

4）运行真正的文件系统中的“init”程序。

这里提一个问题，前面的几个动作还可以用脚本继续实现吗？单个动作本身没有问题，但是不知道读者留意到没有，一旦步骤 1）执行了，rootfs 中就没有内容了，因此后面步骤中使用的命令已经不在了，被删除了，何谈步骤 2）和步骤 3）？因此，这里我们使用一个小技巧，将上面的步骤都封装到一个二进制程序中，将这个程序加载进内存后，我们再删除 rootfs 中的内容，如此一来，步骤 2）和步骤 3）都得以顺利完成。这个程序源码如下：

switch_root.c:

```
#include <errno.h>
#include <dirent.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mount.h>
#include <fcntl.h>
#include <unistd.h>

int delete_dir(char *directory);

void delete(char *what)
{
    if (unlink(what)) {
        if (errno == EISDIR) {
            if (!delete_dir(what))
                rmdir(what);
        }
    }
}

int delete_dir(char *directory)
{
    DIR *dir;
    struct dirent *d;
    struct stat st1, st2;
```

```
    char path[PATH_MAX];

    if (lstat(directory, &st1))
        return errno;

    if (!(dir = opendir(directory)))
        return errno;

    while ((d = readdir(dir))) {
        /* skip ./.. */
        if (d->d_name[0] == '.' &&
            (d->d_name[1] == '\0' ||
             (d->d_name[1] == '.' && d->d_name[2] == '\0')))
            continue;

        sprintf(path, "%s/%s", directory, d->d_name);
        lstat(path, &st2);
        /* do not recurse down mountpoint avoiding del realroot */
        if (st2.st_dev != st1.st_dev)
            continue;

        delete(path);
    }

    closedir(dir);
    return 0;
}

int main(int argc, char *argv[])
{
    int console_fd;

    /* change to the new root directory */
    chdir(argv[1]);

    /* delete rootfs contents */
    delete_dir("/");

    /* overmount the root */
    mount(".", "/", NULL, MS_MOVE, NULL);

    /* chroot, chdir */
    chroot(".");
    chdir("/");

    /* open /dev/console */
    console_fd = open("/dev/console", O_RDWR);
    dup2(console_fd, 0);
    dup2(console_fd, 1);
    dup2(console_fd, 2);
    close(console_fd);

    /* spawn init */
    execlp(argv[2], argv[2], NULL);
```

```
    return 0;
}
```

Makefile 文件如下：

```
switch_root: switch_root.c:

clean:
    rm -rf *.o
    rm -rf switch_root
```

switch_root 本身的逻辑比较简单，基本就是执行我们上面的步骤 1）~步骤 4），首先切换到新的文件系统，因此后面的“.”就是在新文件系统中，然后使用 chroot 命令，将“.”作为新的根文件系统，完成进程的文件系统 namespace 的切换，并重定向了 stdio、stdout 和 stderr。其中有一处需要特别注意，就是在执行删除前，需要做一个小的判断，以免把挂载在 ${ROOTMNT} 下的真正的文件系统也删除了。

编译后，将 switch_root 复制到 initramfs：

```
vita@baisheng:/vita/build/switch_root$ cp switch_root \
    /vita/initramfs/bin/
```

修改 initramfs 中的 init 脚本如下：

```
/vita/initramfs/init:

#!/bin/bash
echo "Hello Linux!"
export PATH=/usr/sbin:/usr/bin:/sbin:/bin
export ROOTMNT=/root
export ROFLAG=-r
mount -n -t devtmpfs udev /dev
mount -n -t proc proc /proc
mount -n -t sysfs sysfs /sys
mount -n -t ramfs ramfs /run
udevd --daemon
udevadm trigger --action=add
udevadm settle

for x in $(cat /proc/cmdline); do
    case $x in
    root=*)
        ROOT=${x#root=}
        ;;
    ro)
        ROFLAG=-r
        ;;
    rw)
        ROFLAG=-w
        ;;
    esac
done
```

```
mount ${ROFLAG} ${ROOT} ${ROOTMNT}

# Stop udevd
udevadm control --exit
# Move to the real filesystem
mount -n --move /dev ${ROOTMNT}/dev
mount -n --move /run ${ROOTMNT}/run
mount -n --move /proc ${ROOTMNT}/proc
mount -n --move /sys ${ROOTMNT}/sys

switch_root ${ROOTMNT} /sbin/init
```

最后，为了验证是否已经切换到根文件系统了，我们在 rootfs 中安装两个程序 cat 和 ls：

```
vita@baisheng:/vita$ cp sysroot/usr/bin/cat rootfs/bin/
vita@baisheng:/vita$ cp sysroot/usr/bin/ls rootfs/bin/
```

用修改过的根文件系统和 initramfs 更新 vita 系统，然后重新启动 vita 系统。使用命令 cat 打印文件 /proc/mounts 的内容，如图 4-29 所示。

图 4-29 成功切换到根文件系统

根据 /proc/mounts 的输出可见，存放根文件系统的分区“/dev/sda2”已经挂载到了“/”下。在这里，我们也看到，rootfs 还是作为整个虚拟文件系统的根存在的。

第5章

从内核空间到用户空间

前面，我们从无到有，编译了内核，构建了 initramfs 和一个基本的根文件系统，成功启动了用户空间的第一个进程。虽然我们只是迈出了一小步，但是这是关键的一步。在此基础上，我们可以放开手脚，去探索曾经的遥不可及。但是，我们也才刚刚破冰，学而不思则罔，因此，在继续构建一个完整的操作系统之前，我们先来更深入的探索一下这一切是如何发生的。

曾经不止一次，笔者在各个技术文章、书籍、甚至顶尖高校的讲义中，都看到类似的论述：内核首先进入实模式，然后从实模式跳入保护模式，事实果真如此吗？在这一章中，我们首先从 Linux 操作系统的加载谈起。

对于普通程序，它们运行在操作系统已经为其准备好的环境中，操作系统则没有这么幸运，其运行在裸机上。操作系统需要在裸机上自己引导自己，而且还要为运行进程搭建好环境。因此，本章的 5.2 节和 5.3 节将讨论内核是如何自解压以及如何初始化的。

操作系统最终的目的之一是承载进程。因此，在本章的最后，我们讨论了进程的加载和运行。提及进程的加载和运行，我们几乎将所有的关注都放在了内核上，却往往忽略了另外一个为进程辅以建立运行环境的重要角色：动态链接器。在进程加载中，相当一部分烦琐而又重要的工作由动态链接器完成。因此，除了讨论进程在内核中的加载过程外，我们也深入探讨了进程在用户空间的加载和链接过程。

5.1 Linux 操作系统加载

PC 上电或复位后，处理器跳转到 BIOS，开始执行 BIOS。BIOS 首先进行加电自检，初始化相关硬件，然后加载 MBR 中的程序到内存 0x7c00 处并跳转到该地址处，接着由 MBR 中的程序完成操作系统的加载工作。通常，MBR 中的程序也被称为 Bootloader。当然，鉴于现代操作系统的复杂性，Bootloader 已远远不止一个扇区大小。这一节，我们就以一个具体的 Bootloader——GRUB 为例，探讨操作系统的加载过程。为简单起见，我们只讨论典型的从硬盘加载操作系统的过程，所以后续的讨论全部是针对从硬盘启动的情况。

PC 上硬盘的传统分区方式是 MBR 分区方案。但是 MBR 最大能表示的分区大小为 2TB。因此，随着硬盘容量的不断扩大，为了突破 MBR 分区方式的一些限制，20 世纪 90 年代 Intel 提出了 GPT 分区方案。对于不同的分区方式，加载操作系统的方式还是有些许不同的。也是为了简单起见，我们结合现在依然广泛使用的传统的 MBR 分区方案进行讨论。

5.1.1 GRUB 映像构成

对于仅有 512 字节大小的 MBR，又要留给分区表 64 字节，在这么小的一个空间，已经很难容纳加载一个现代操作系统的代码。于是 GRUB 采取了分阶段的策略，MBR 中仅存放 GRUB 的第一阶段的代码，MBR 中的代码负责把 GRUB 的其余部分载入内存。

但是 GRUB 分成几段合适呢？要回答这个问题我们还得从 DOS 谈起。

DOS 的系统映像是不能跨柱面存放的，所以在 DOS 时代，磁盘的第一个分区索性并没有紧接在 MBR 的后面，而是直接从下一个柱面的边界开始。而且，按照柱面对齐，对系统的性能有很大好处，这对于现代操作系统同样适用。于是，在 MBR 与第一个分区之间，就出现了一块空闲区域。从那时起，这种分区方式成为了一个约定俗成，基本上所有的分区工具都把这种分区方式保留了下来。如果硬盘是 MBR 分区方案，用分区工具 fdisk 就可以看到这一点，以笔者的机器为例：

```
root@baisheng:~# fdisk -l

Disk /dev/sda: 500.1 GB, 500107862016 bytes
255 heads, 63 sectors/track, 60801 cylinders, total 976773168 ...
...
   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *          63   104872319    52436128+  83  Linux
/dev/sda2       104872320   188779814    41953747+  83  Linux
...
```

根据 fdisk 的输出可见，每个磁道划分为 63 个扇区。硬盘的第一个分区起始于第 63 个扇区（从 0 开始计数）。也就是说，对于第 0 个磁道，除了 MBR 占据的一个分区，其余 62 个分区是空闲的。

于是，GRUB 的开发人员就打算把 GRUB“嵌入”到这个空闲区域，这样做的好处就是相对来说比较安全。因为某些文件系统的一些特性或者一些修复文件系统的操作，有可能导致文件系统中的文件所在的扇区发生改变。因此，单纯依靠扇区定位文件是有一定的风险的。而对于 GRUB 来说，在其初始阶段，由于尚未加载文件系统的驱动，因此，它恰恰需要通过 BIOS 以扇区的方式访问 GRUB 的后续的阶段。但是，一旦 GRUB 嵌入到这个不属于任何分区的特殊区域，则将不再受文件系统的影响。当然将 GRUB 嵌入到这个区域也不是必须的，但是因为这个相对安全的原因，GRUB 的开发人员推荐将 GRUB 嵌入到这个区域。

但是这个区域的大小是有限的，通常，一个扇区 512 字节，一个柱面最多包含 63 个扇区。因此，除去 MBR，这个区域的大小是 62 个扇区，即 31KB。因此，嵌入到这里的

GRUB 的映像最大不能超过 31KB。为了控制嵌入到这个区域中的映像的尺寸不超过 31KB，GRUB 采用了模块化的设计方案。

GRUB 在嵌入的映像中包含硬件及文件系统的驱动，因此，一旦嵌入的映像载入内存，GRUB 即可访问文件系统。其他模块完全可以存储在文件系统上，通过文件系统的接口访问这些模块，避开了因为如修复文件系统而引起文件所在扇区的变化而带来的风险。另外也可以很好地控制嵌入到空闲扇区的映像的尺寸。

由上述内容可知，GRUB 将映像分为三个部分：MBR 中的 boot.img、嵌入空闲扇区的 core.img 以及存储在文件系统中的模块。这三个部分也对应着 GRUB 执行的三个阶段。在 MBR 分区模式下，以嵌入方式安装的 GRUB 的各个部分在硬盘上的分布如图 5-1 所示。

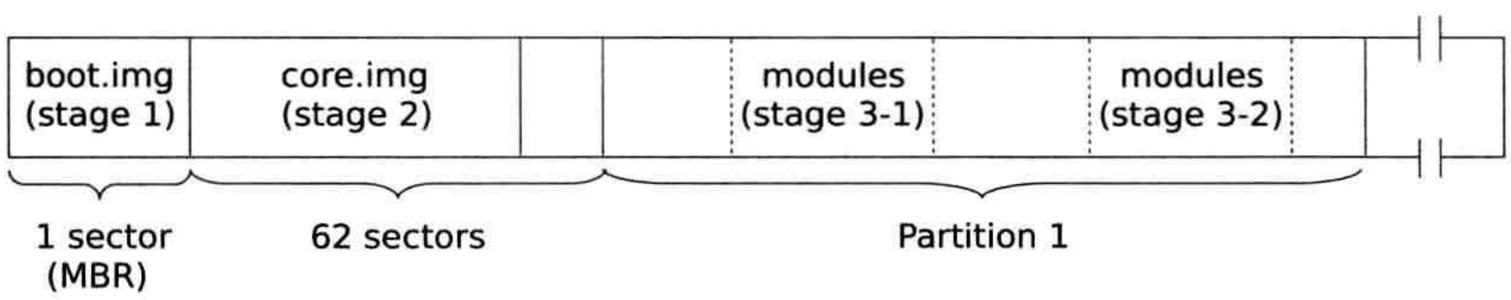


图 5-1 在 MBR 分区模式下以嵌入方式安装的 GRUB

boot.img 以及 core.img 分别以读写磁盘扇区的方式访问，它们不属于任何一个硬盘分区，所以不会受到文件系统的影响。第三阶段的这些模块是存储在文件系统上的，虽然文件所在的扇区可能会变动，但是 GRUB 不再通过扇区访问而是通过文件系统访问。

1. MBR 映像

boot.img 主要功能是将 core.img 中的第一个扇区载入内存。为什么只是加载 core.img 的第一个扇区而不是加载整个 core.img 呢？答案还是因为那可怜的区区 512 字节，除去 64 字节的分区域表信息以及最后的 2 字节的引导标识，还要给 BIOS 保留一段参数空间，boot.img 中可用的空间已经被瓜分得所剩无几。因此，索性 boot.img 中仅记录 core.img 的第一个扇区号，并仅将这个扇区号对应的扇区中的内容加载入内存，core.img 其余部分的加载留给 core.img 的第一个扇区的代码去考虑吧。

boot.img 对应的源文件是 boot.S，其中保存 core.img 的第一个扇区的位置如下：

```
grub-2.00/grub-core/boot/i386/pc/boot.S:
...
    . = _start + GRUB_BOOT_MACHINE_KERNEL_SECTOR
kernel_sector:
    .long   1, 0
...
grub-2.00/include/grub/i386/pc/boot.h:
#define GRUB_BOOT_MACHINE_KERNEL_SECTOR 0x5c
```

boot.S 中标号 kernel_sector 所在处，即 boot.img 中偏移 GRUB_BOOT_MACHINE_KERNEL_SECTOR，即 92 字节（0x5c）处，记录的就是 core.img 第一个扇区在硬盘上所在的扇区号。

后面讨论 GRUB 安装时，我们会看到，在安装 GRUB 时，GRUB 的安装程序将根据 core.img 的第一个扇区占据的实际硬盘扇区号修改这里。事实上，如果 GRUB 采用的是嵌入模式，那么这里的扇区就应该是 1，即紧接在 MBR 后面的一个扇区。

由于程序大小被限制在可怜的一个扇区内，不能奢望在这么小的程序内实现硬盘以及文件系统的驱动，所以，boot.img 只能利用 BIOS 提供的中断向量为 0x13 的基于扇区的磁盘读写服务。以支持 LBA 模式的硬盘为例，读取扇区的代码如下：

```
grub-2.00/grub-core/boot/i386/pc/boot.S:

...
lba_mode:
    ...
    movl    kernel_sector, %ebx
    ...
    movb    $0x42, %ah
    int $0x13
    ...
    /* boot kernel */
    jmp *(kernel_address)
```

boot.img 按照 BIOS 服务的要求，设置相应的寄存器，调用 BIOS 服务。BIOS 负责将地址 kernel_sector 处指示的扇区号所在扇区的内容载入内存。boot.img 最后把读入的扇区内容移动到符号 kernel_address 处指示的地址，并跳转到那里执行。符号 kernel_address 处的值为宏 GRUB_BOOT_MACHINE_KERNEL_ADDR，如下代码：

```
grub-2.00/grub-core/boot/i386/pc/boot.S:

kernel_address:
    .word   GRUB_BOOT_MACHINE_KERNEL_ADDR
```

这个宏的值是 0x8000，也就是说，GRUB 第二阶段映像被移动到了这里，并且从这里继续执行。后面在讨论 GRUB 启动时，读者会看到，链接器给 core.img 的最初 512 字节分配的地址，也确实是从 0x8000 开始的。

另外，读者并不会在 boot.S 中看到关于分区表的部分，因为在安装 GRUB 时，安装程序负责将分区表写到 boot.img 中。

2. GRUB 核心映像

core.img 包括多个映像和模块，以从硬盘启动为例，core.img 包含的内容如图 5-2 所示。

图 5-2　core.img 构成示意图

图 5-2 中 diskboot.img 占据 core.img 中的第一个扇区，它就是 boot.img 加载的 core.img 的所谓的第一个扇区。diskboot.img 用来加载 core.img 中除 diskboot.img 外的其余部分，与 boot.S 的实现本质上并无不同，也是借助 BIOS 的中断服务。只不过 boot.img 加载一个扇区进入内存，而 diskboot.img 加载多个扇区进入内存而已。

与 boot.img 类似，diskboot.img 也需要知道 core.img 的后续部分所在的扇区。显然，只有在将 GRUB 安装到磁盘时，才能知道 core.img 实际所占据的扇区。因此，在安装时，GRUB 的安装程序会将 core.img 占据的扇区号写入 diskboot.img 中。相关代码如下：

```
grub-2.00/grub-core/boot/i386/pc/diskboot.S:
...
    . = _start + 0x200 - GRUB_BOOT_MACHINE_LIST_SIZE
LOCAL(firstlist):
blocklist_default_start:
    .long 2, 0
blocklist_default_len:
    .word 0
blocklist_default_seg:
    .word (GRUB_BOOT_MACHINE_KERNEL_SEG + 0x20)
```

diskboot.img 的最后 12 字节记录的是一个 blocklist，每个 blocklist 代表一个连续的扇区，其对应的 C 语言的结构体如下：

```
grub-2.00/include/grub/offsets.h:

struct grub_pc_bios_boot_blocklist
{
  grub_uint64_t start;
  grub_uint16_t len;
  grub_uint16_t segment;
} __attribute__ ((packed));
```

其中 start 代表这个连续的扇区的起始扇区，len 表示扇区的数量，segment 表示扇区加载到内存的段地址。

在 diskboot.S 中，注意标号 blocklist_default_seg 处的宏 GRUB_BOOT_MACHINE_KERNEL_SEG，这是一个类似带参数的宏，对于使用 x86 架构的 PC，MACHINE 最后会被替换为“I386_PC”，展开后为 GRUB_BOOT_I386_PC_KERNEL_SEG：

```
grub-2.00/include/grub/offsets.h:

#define GRUB_BOOT_I386_PC_KERNEL_SEG    0x800
```

也就是说，diskboot.img 将 core.img 中除 diskboot.img 外的部分加载到内存的段地址为 0x820。在 diskboot.img 进行加载时，将段内偏移设置为了 0，所以最终 core.img 的其余部分被加载到了从内存地址 0x8200 开始的地方。在前面讨论 boot.img 时，我们看到，boot.img 将 diskboot.img 加载到了 0x8000 处。也就是说，diskboot.img 正好占据了一个扇区（0x200 字节）。

事实上，对于 MBR 分区方案，如果采用了嵌入式的安装方式，那么只要有一个 blocklist 就足够了。当使用非嵌入式的安装方式时，core.img 可能被分块存储在磁盘上，因此，diskboot.img 中可能存在多个 blocklist，每一个 blocklist 代表一段连续的扇区。第一个 blocklist 位于 diskboot.img 的最后，每增加一个 blockllist，向着 diskboot.img 开始的方向延伸。

为了控制 core.img 的体积，GRUB 将 core.img 进行了压缩。显然 diskboot.img 是不能压缩的，因为 boot.img 中没有任何解压代码。因此，GRUB 只将 core.img 中的 kernel.img 和模块进行了压缩。对于基于 x86 架构的 PC，GRUB 默认使用的是 lzma 压缩算法。当然安装 GRUB 前创建 core.img 时，用户也可通过命令行参数指定压缩算法，但是从 2.0 版本的代码来看，对于 x86 架构来说，只能使用 lzma 压缩算法。

既然有压缩部分，就要有负责解压的部分。GRUB 将 lzma 算法的解压缩代码编译为 lzma_decompress.img，连接在 diskboot.img 的后面。diskboot.img 将 core.img 加载进内存后，将跳转到 lzma_decompress.img，执行其中代码解压缩 core.img 后面的压缩部分。下面的代码就是 diskboot.img 加载完 core.img 后进行的跳转：

```
grub-2.00/grub-core/boot/i386/pc/diskboot.S:

...
LOCAL(bootit):
    ...
    ljmp    $0, $(GRUB_BOOT_MACHINE_KERNEL_ADDR + 0x200)
```

宏 GRUB_BOOT_MACHINE_KERNEL_ADDR 定义如下：

```
#define GRUB_BOOT_MACHINE_KERNEL_ADDR
   (GRUB_BOOT_MACHINE_KERNEL_SEG << 4)
```

刚刚我们已经看到了，宏 GRUB_BOOT_MACHINE_KERNEL_SEG 值为 0x800，于是左移 4 位后，宏 GRUB_BOOT_MACHINE_KERNEL_ADDR 的值为 0x8000。可见，加载完 core.img 剩余部分后，diskboot.img 跳转到了地址 0x8000 + 0x200 处，正是 lzma_decompress.img。lzma_decompress.img 解压后面的压缩的映像，最终跳转到 kernel.img。

根据其名字我们就可以猜到了，kernel.img 是 GRUB 的核心代码了。其中包括为底层具体的磁盘驱动以及文件系统驱动提供公共的服务层。kernel.img 的主入口函数是 grub_main。lzma_decompress.img 解压后正是跳转到这个函数，从某种意义上讲，这里才是 GRUB 的真正开始。

```
grub-2.00/grub-core/kern/main.c:

void __attribute__ ((noreturn)) grub_main (void)
{
  ...
  grub_load_modules ();
  ...
}
```

鉴于嵌入区域的尺寸有限，因此只有最关键的模块才能包含到 core.img 中，随着 core.img 一起嵌入到 MBR 后面的空闲扇区。那么哪些模块是关键模块呢？只有 core.img 支持文件系统，它才可以读入其他模块。所以，这就是磁盘驱动模块 biosdisk.mod、MBR 分区模式模块 part_msdos.mod 以及文件系统的驱动模块 ext2.mod（虽然其名字为 ext2，但是这个模块支持 EXT 系列文件系统）包含到 core.img 中的原因，它们的目的是驱动文件系统。

这几个模块虽然已经被 diskboot.img 加载进了内存，但是显然只是将它们简单地“放到”内存中还是不够的，因为这些模块就相当于目标文件，指令和数据地址都是从 0 开始分配的，比如以笔者机器上的 GRUB 的模块 ext2.mod 为例：

```
root@baisheng:/boot/grub/i386-pc# readelf -S ext2.mod

Section Headers:
  [Nr] Name              Type            Addr     Off    Size
  [ 0]                   NULL            00000000 000000 000000
  [ 1] .text             PROGBITS        00000000 000034 000cb7
  ...
```

所以需要为它们进行重定位，还是以这个模块为例，我们可以看到其有大量需要重定位的符号：

```
root@baisheng:/boot/grub/i386-pc# readelf -r ext2.mod

Relocation section '.rel.text' at offset 0x1330 contains 101 ...
 Offset     Info    Type            Sym.Value  Sym. Name
0000000a  00001702 R_386_PC32        00000000   grub_free
...
Relocation section '.rel.data' at offset 0x1658 contains 8 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000008  00000201 R_386_32         00000000   .rodata.str1.1
...
```

这就是函数 grub_main 调用 grub_load_modules 的目的。这个函数执行完毕后，GRUB 就支持文件系统了，访问磁盘上的文件时，无须再依靠原始的 BIOS 使用扇区的方式读写文件了。后续无论是读取 GRUB 的其他模块还是加载内核，都是通过文件系统的接口。

5.1.2 安装 GRUB

通常，在安装操作系统的最后，操作系统安装程序将会为用户安装 GRUB。当然，有时我们也会手动安装 GRUB。但是都是通过 GRUB 提供的工具，执行的命令如下：

```
grub-install /dev/sda
```

事实上，在这个安装命令的背后，GRUB 的安装过程分为两个阶段：第一阶段是创建 core.img，GRUB 为此提供的工具是 grub-mkimage；第二阶段是安装 boot.img 及 core.img 到硬盘，GRUB 提供的工具是 grub-setup。为了方便，GRUB 将这两个过程封装到脚本 grub-install 中。

在创建 core.img 时，grub-mkimage 需要获取需要加入 core.img 的模块，GRUB 也提供了相应的工具 grub-probe。grub-install 利用这个工具根据内核映像所在的介质，自动探测所需要的模块，并将它们传给 grub-mkimage。以笔者机器为例，使用 grub-probe 探测磁盘分区方式和文件系统的的方法如下：

```
root@baisheng:~# grub-probe --target=partmap -d /dev/sda1
msdos
root@baisheng:~# grub-probe --target=fs -d /dev/sda1
ext2
```

1. 创建映像

grub-install 首先调用 grub-mkimage 创建 core.img，我们结合其源代码来讨论 core.img 的创建过程。

grub-2.00/util/grub-mkimage.c:

```
01 static void generate_image (..., FILE *out, ..., char *mods[], ...)
02 {
03
04   path_list = grub_util_resolve_dependencies (dir,
05                   "moddep.lst", mods);
06
07   kernel_path = grub_util_get_path (dir, "kernel.img");
08   ...
09   kernel_img = load_image32 (kernel_path, ...);
10   ...
11   for (p = path_list; p; p = p->next)
12     {
13       ...
14       grub_util_load_image (p->name, kernel_img + offset);
15       ...
16     }
17   ...
18   compress_kernel (image_target, kernel_img, kernel_size +
19       total_module_size, &core_img, &core_size, comp);
20   ...
21   if (image_target->flags & PLATFORM_FLAGS_DECOMPRESSORS)
22     {
23       ...
24       switch (comp)
25         {
26         ...
27         case COMPRESSION_LZMA:
28           name = "lzma_decompress.img";
29           break;
30         ...
31         }
32       ...
33       decompress_img = grub_util_read_image (decompress_path);
34       ...
35       memcpy (full_img, decompress_img, decompress_size);
```

```
36
37        memcpy (full_img + decompress_size, core_img, core_size);
38        ...
39        core_img = full_img;
40        core_size = full_size;
41      }
42
43    switch (image_target->id)
44      {
45      case IMAGE_I386_PC:
46      case IMAGE_I386_PC_PXE:
47        {
48          ...
49          boot_path = grub_util_get_path (dir, "diskboot.img");
50          ...
51          boot_img = grub_util_read_image (boot_path);
52
53          {
54            struct grub_pc_bios_boot_blocklist *block;
55            block = (struct grub_pc_bios_boot_blocklist *)
56              (boot_img + GRUB_DISK_SECTOR_SIZE - sizeof (*block));
57            block->len = grub_host_to_target16 (num);
58            ...
59          }
60
61          grub_util_write_image (boot_img, boot_size, out,
62              outname);
63          ...
64        }
65        ...
66      }
67
68    grub_util_write_image (core_img, core_size, out, outname);
69    ...
70  }
```

根据上面的代码可见，创建 core.img 的主要过程如下：

1）generate_image 读取 kernel.img 到内存，见代码第 7~9 行。

2）除了 kernel.img 外，还要将一些模块合并到 core.img 中。传递给函数 generate_image 的参数 mods 是一个数组，其中记录的是每个要合并到 core.img 中的模块。但是这些模块可能还依赖其他模块，所以代码第 4~5 行是检查这些模块的依赖模块，并将这些模块记录到链表 path_list 中。然后，第 11~16 行的代码将这些模块全部加载到 kernel.img 的后面。

3）至此，kernel.img 和各个模块组成的 core.img 组装完成。为了在 62 个扇区中容纳下 core.img，所以代码第 18~19 行压缩 core.img。对于基于 IA32 架构的 PC，GRUB 使用的默认压缩方法是 LZMA。

4）既然压缩了 core.img，那么就得有人来负责解压缩。如同内核采用的方法，GRUB 也在压缩的 core.img 前面附加了一段未经压缩的指令，见代码第 21~37 行。如果 core.img 使用的是 LZMA 压缩方法，则 generate_image 读取 lzma_decompress.img，将其附加到 core.img

的前面。

5）如果 core.img 是为基于 IA32 的 PC 创建的，那么在 core.img 的最前面还要附加一个 diskboot.img，代码第 43~66 就是准备 diskboot.img。因为 diskboot.img 负责将 core.img 中除 diskboot.img 的部分读入内存，因此，diskboot.img 需要知道 core.img 所在的扇区信息。因为这里已经确定了 core.img 占用的大小，所以，generate_image 将 core.img 占用的扇区数写入了 diskboot.img 最后的 blocklist 中，这就是代码第 53~59 行的目的。准备好 diskboot.img 后，generate_image 将其写入了在磁盘上保存 core.img 的文件，见代码第 61~62 行，其中 out 就是对应的文件 core.img。

6）在将 diskboot.img 写入磁盘文件 core.img 后，generate_image 将内存中的 core.img 的其他部分，包括 lzma_decompress.img、kernel.img 以及需要合并到 core.img 中的模块，也写入磁盘上的文件 core.img 中，紧接在 diskboot.img 之后。至此，core.img 映像创建完毕。

2. 安装映像

创建完 core.img 映像后，grub-install 将调用 grub-setup 将 core.img（包括 boot.img）安装到硬盘。下面我们结合 grub-setup 的代码来讨论这一过程。

grub-2.00/util/grub-setup.c:

```
01 static void setup (...)
02 {
03
04   boot_img = grub_util_read_image (boot_path);
05   ...
06   core_img = grub_util_read_image (core_path);
07   ...
08     if (is_ldm)
09       err = grub_util_ldm_embed (dest_dev->disk, &nsec, maxsec,
10                                  GRUB_EMBED_PCBIOS, &sectors);
11     else if (dest_partmap)
12       err = dest_partmap->embed (dest_dev->disk, &nsec, maxsec,
13                                  GRUB_EMBED_PCBIOS, &sectors);
14     else
15       err = fs->embed (dest_dev, &nsec, maxsec,
16                        GRUB_EMBED_PCBIOS, &sectors);
17     ...
18     for (i = 1; i < nsec; i++)
19       save_blocklists (sectors[i] + grub_partition_get_start
20                (container), 0, GRUB_DISK_SECTOR_SIZE);
21     ...
22     write_rootdev (core_img, root_dev, boot_img, first_sector);
23     ...
24     for (i = 0; i < nsec; i++)
25       grub_disk_write (dest_dev->disk, sectors[i], 0,
26                        GRUB_DISK_SECTOR_SIZE,
27                        core_img + i * GRUB_DISK_SECTOR_SIZE);
28   ...
29   if (grub_disk_write (dest_dev->disk, BOOT_SECTOR,
30                        0, GRUB_DISK_SECTOR_SIZE, boot_img))
```

```
31    ...
32 }
```

根据上述代码可见，安装 GRUB 的主要过程如下：

1）grub-setup 首先读取 boot.img 和 core.img 到内存，见代码第 4~6 行。

2）boot.img 的安装位置是固定的，即 MBR，但是 grub-setup 需要确定 core.img 安装的扇区。对于不同的情况，确定的方法是不同的。代码第 8~10 行是针对多个磁盘组成的逻辑盘的情况。否则依次尝试使用具体分区方案以及文件系统提供的 embed 函数，见代码第 11~16 行。代表 MBR 分区方案的对象是 msdos，其中提供了获取安装 GRUB 所在的扇区函数 pc_partition_map_embed，该函数将计算 core.img 安装的扇区，并将结果保存到数组 sectors 中。变量 nsec 记录的是 core.img 占用扇区数。

3）在确定了 core.img 的安装扇区后，显然要将 diskboot.img 中的 blocklist 填充上，代码第 18~20 行就是在做这件事。

4）一旦确定了 core.img 安装的扇区，grub-setup 还要修订 boot.img。虽然对嵌入式安装而言，diskboot.img 就安装在第 2 个扇区，但是 GRUB 不能进行这样的假设，因为还有可能使用非嵌入的安装方式。因此，grub-setup 需要设置 boot.img 中 diskboot.img 所在的扇区，见代码第 22 行。其中所谓的 first_sector 就是 core.img 的第 1 个扇区，即 diskboot.img 占据的硬盘扇区。

5）映像准备好后，如果采用嵌入式安装，那么需要将 core.img 嵌入 MBR 后面的空闲扇区，即数组 sectors 中记录的扇区，见代码第 24~27 行。对于嵌入式安装，因为是嵌入在一块连续的扇区，所以 diskboot.img 中只需要记录一个 blocklist，如图 5-3 所示。

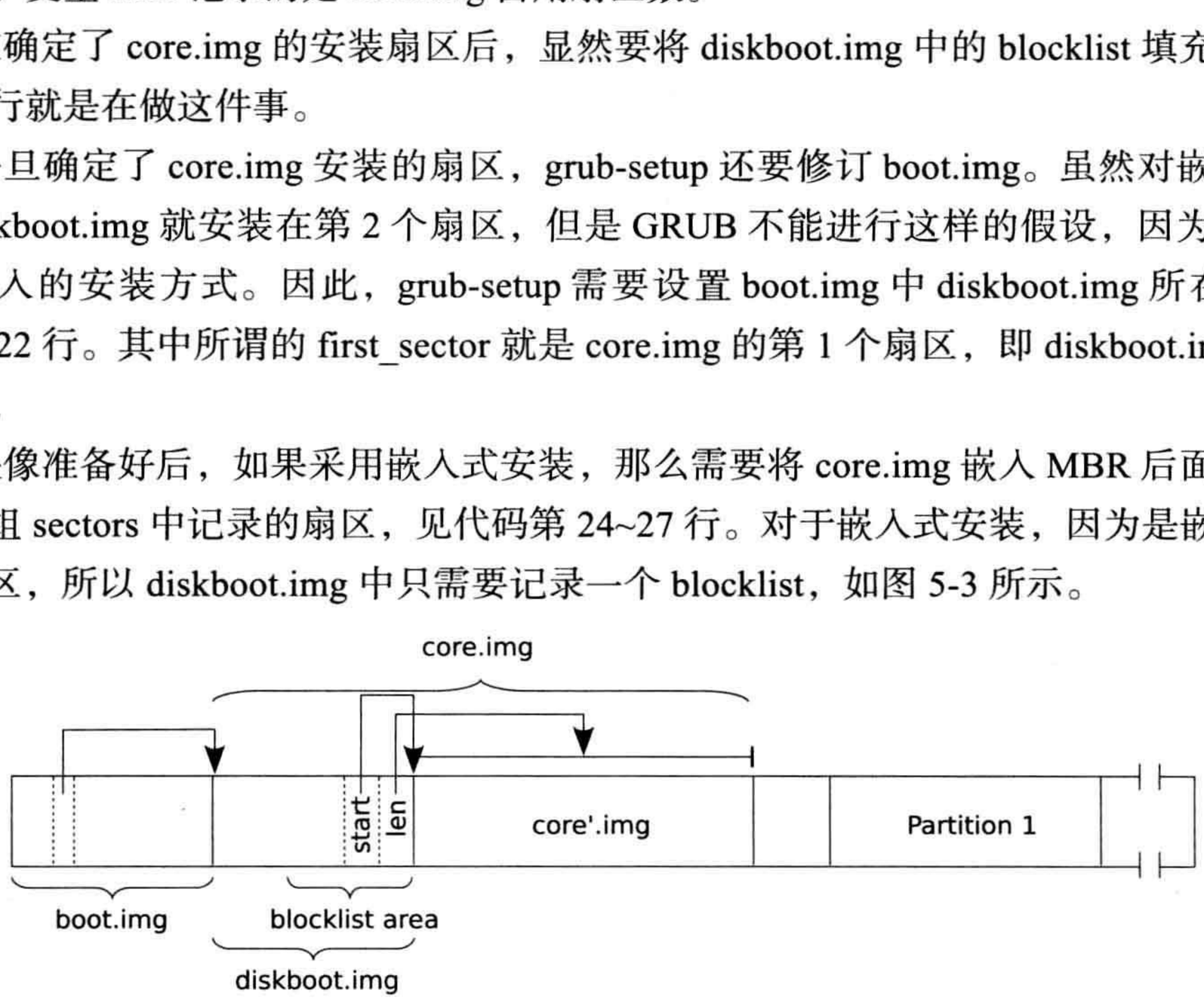


图 5-3　嵌入式安装 GRUB 示意图

6）最后，grub-setup 将 boot.img 写入 MBR，见代码第 29~30 行。

为简单起见，上面代码中略去了非嵌入式安装的情况。在非嵌入式安装情况下，GRUB 不需要将保存在文件系统中的 core.img 写入到 MBR 后面空闲的扇区中，而只需要将文件系统中的 core.img 所在的扇区写入 disboot.img 的 blocklist，将 diskboot.img 所在的扇区写入 boot.img 即可。因为 core.img 很有可能不是连续存储在硬盘上的，所以 diskboot.img 中需要

记录多个 blocklist，这就是 diskboot.img 后面预留了多个 blocklist 空间的原因，如图 5-4 所示。

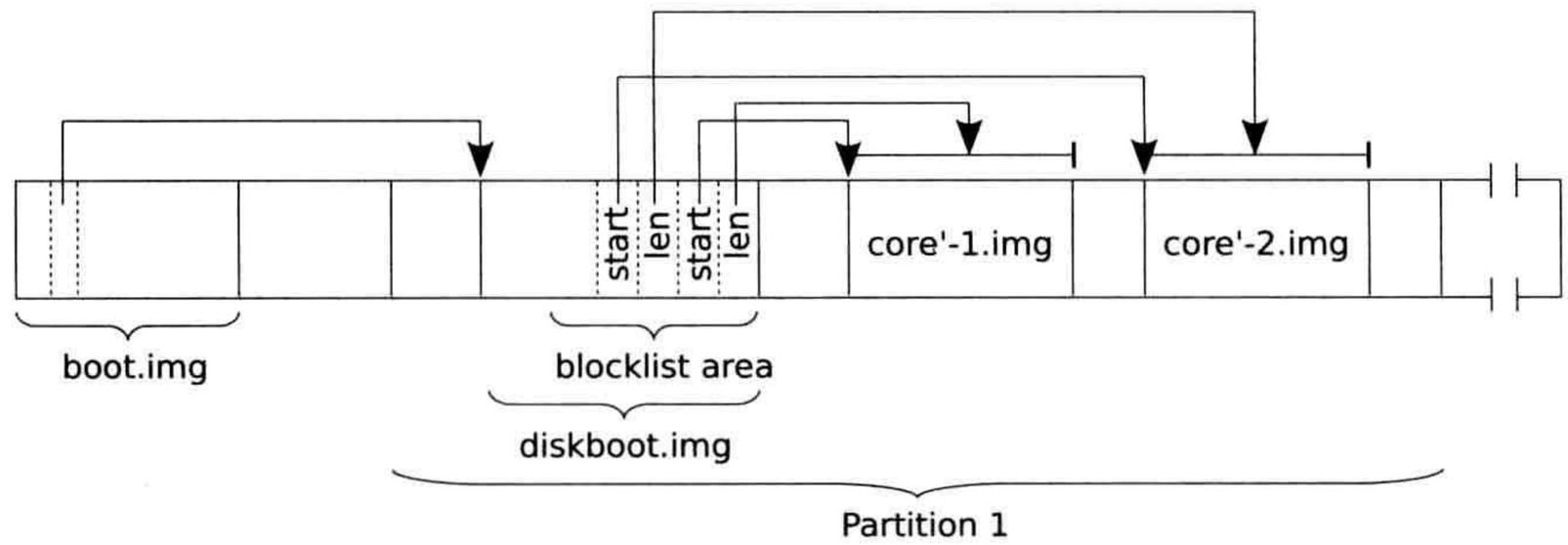


图 5-4　非嵌入式安装 GRUB 示意图

5.1.3　GRUB 启动过程

我们知道，在 PC 启动时，BIOS 会将 MBR 中的程序加载到内存的 0x7c00 处，并跳转到那里开始执行。对于 GRUB 来说，对应 MBR 中的映像是 boot.img，在编译 boot.img 时，编译脚本确实也是指导链接器从 0x7c00 开始为其指令和数据分配地址的，如下面编译脚本片段中使用黑体标识的部分：

```
grub-2.00/grub-core/Makefile.core.am:

if COND_i386_pc
platform_PROGRAMS += boot.image
...
boot_image_LDFLAGS  = $(AM_LDFLAGS) $(LDFLAGS_IMAGE) ... 0x7c00
...
endif
```

读者可能会注意到脚本中映像的后缀是“image”，而不是“img”，这是因为编译脚本最后会将 ELF 格式的 boot.image 转换为裸二进制格式，并命名为 boot.img。其余映像也是如此处理。

当跳转到 0x7c00 后，GRUB 开始执行，其启动过程大体如图 5-5 所示。

（1）boot.img 加载 diskboot.img

boot.img 使用 BIOS 中断号为 0x13 的基于扇区的磁盘读写服务加载 diskboot.img。GRUB 使用从 0x70000 开始处的一段内存作为 BIOS 读缓存，所以 BIOS 首先将 diskboot.img 读到内存 0x70000 处，然后 boot.img 再将其移动到内存 0x8000 处。根据下面脚本片段中黑色标识的部分可见，链接器确实是从 0x8000 为 diskboot.img 分配地址的：

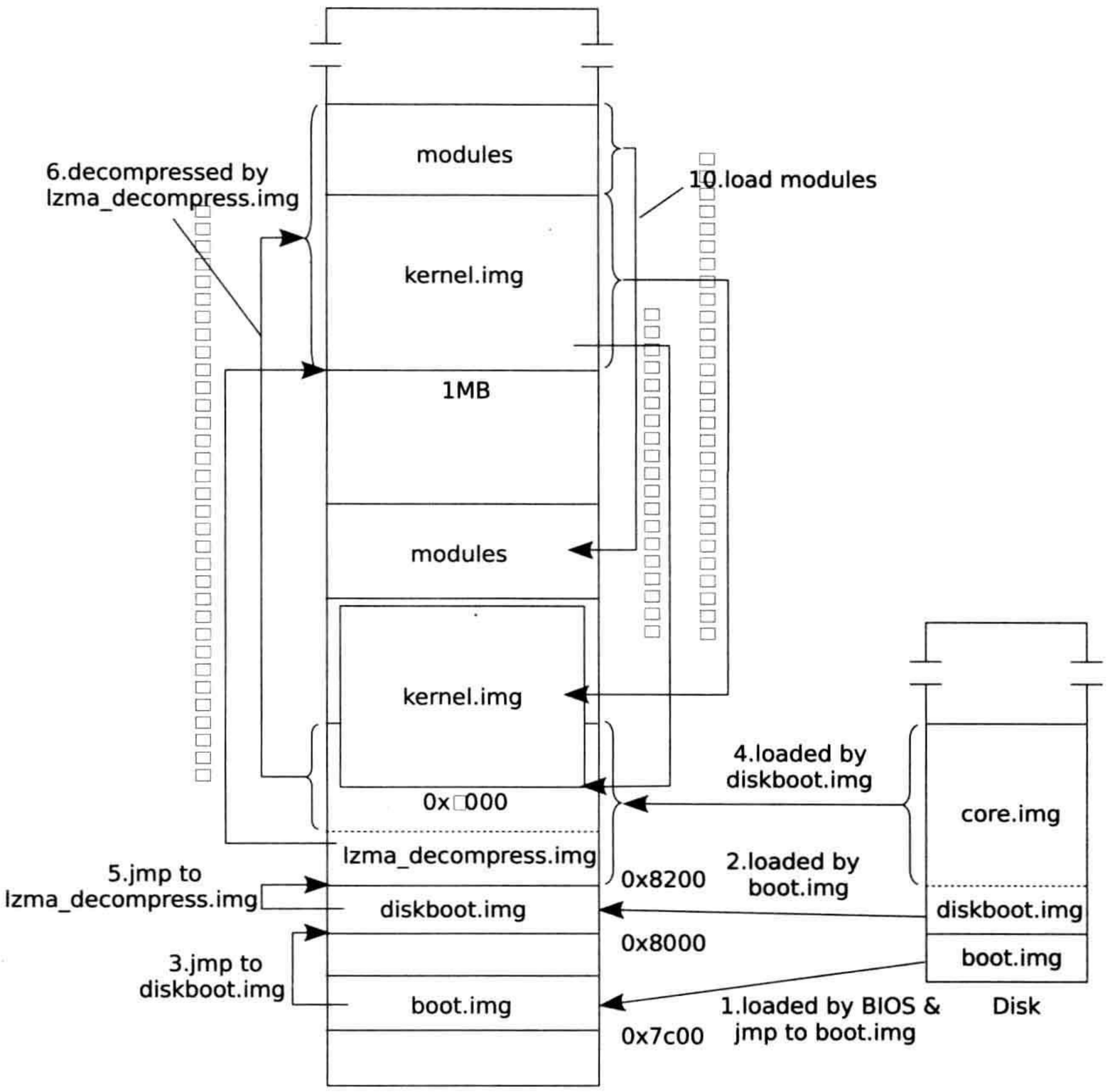


图 5-5 GRUB 启动过程

```
grub-2.00/grub-core/Makefile.core.am:

if COND_i386_pc
platform_PROGRAMS += diskboot.image
...
diskboot_image_LDFLAGS  = $(AM_LDFLAGS) $(LDFLAGS_IMAGE) ... 0x8000
...
diskboot_image_OBJCOPYFLAGS  = $(OBJCOPYFLAGS_IMAGE) -O binary
...
endif
```

boot.img 将 diskboot image 加载完成后，跳转到 diskboot 中的第一条指令处继续执行。

（2）diskboot.img 加载 core.img

与 boot.img 类似，diskboot.img 使用 BIOS 中断号为 0x13 的基于扇区的磁盘读写服务加载 core.img。BIOS 将 lore.img 读到缓存 0x70000 处，然后 diskboot.img 将其移动到 0x8200 处，最后跳转到 0x8200 处开始执行 lzma_decompress.img。

（3）core.img 自解压

在讨论 GRUB 创建映像时，我们看到，连接在 core.img 前端的 lzma_decompress.img 并没有被压缩，而这段没有被压缩的部分的作用就是负责解压 core.img 其余压缩的部分。我们来看看构建 lzma_decompress.img 的脚本片段：

```
grub-2.00/grub-core/Makefile.core.am:

if COND_i386_pc
platform_PROGRAMS += lzma_decompress.image
lzma_decompress_image_SOURCES   = boot/i386/pc/startup_raw.S
...
lzma_decompress_image_LDFLAGS   = $(AM_LDFLAGS) ... 0x8200
...
endif
```

core.img 被 diskboot.img 加载到 0x8200 处，lzma_decompress.img 作为 core.img 的开头，其地址应该从 0x8200 分配，根据上面的编译脚本，即可见这一点。

从脚本可见，lzma_decompress.img 的开头是 startup_raw.S，其正是解压 core.img 的地方，见下面的代码：

```
grub-2.00/grub-core/boot/i386/pc/startup_raw.S:

#ifdef ENABLE_LZMA
    movl    $GRUB_MEMORY_MACHINE_DECOMPRESSION_ADDR, %edi
    ...
    pushl   %edi
    ...
    push    %ecx
    call    _LzmaDecodeA
    pop %ecx
    /* _LzmaDecodeA clears DF, so no need to run cld */
    popl    %esi
#endif
    ...
    jmp *%esi

#ifdef ENABLE_LZMA
#include "lzma_decode.S"
#endif
```

其中，_LzmaDecodeA 就是进行解压的函数，其实现在文件 lzma_decode.S 中，所以 startup_raw.S 将这个文件包含进来。lzma_decompress.img 将 core.img 解压到内存 GRUB_MEMORY_MACHINE_DECOMPRESSION_ADDR 处，定义如下：

```
grub-2.00/include/grub/i386/pc/memory.h:

/* The area where GRUB is decompressed at early startup.  */
#define GRUB_MEMORY_MACHINE_DECOMPRESSION_ADDR  0x100000
```

因为低端内存捉襟见肘，所以 GRUB 使用从 1MB 开始的空间作为解压的缓冲区。

GRUB之所以能访问1MB以上的内存地址，是因为开启了CPU的保护模式。但是GRUB并没有启用分页，而是采用了段式寻址，而且还采用了特殊的平坦内存模型（flat model），即段基址为0。平坦内存模型的寻址比较简单，某种意义上就是短路了CPU的段机制，对于未开启分页的平坦内存模型，偏移地址就是最后的物理地址。GRUB中使用了平坦内存模型的GDT的设置如下：

grub-2.00/grub-core/kern/i386/realmode.S:

```
gdt:
    .word   0, 0
    .byte   0, 0, 0, 0

    /* -- code segment --
     * base = 0x00000000, limit = 0xFFFFF (4 KiB Granularity), present
     * type = 32bit code execute/read, DPL = 0
     */
    .word   0xFFFF, 0
    .byte   0, 0x9A, 0xCF, 0

    /* -- data segment --
     * base = 0x00000000, limit 0xFFFFF (4 KiB Granularity), present
     * type = 32 bit data read/write, DPL = 0
     */
    .word   0xFFFF, 0
    .byte   0, 0x92, 0xCF, 0

/* this is the GDT descriptor */
gdtdesc:
    .word   0x27            /* limit */
    .long   gdt           /* addr */
```

core.img解压完成后，lzma_decompress.img将跳转到解压的core.img处继续执行。根据前面讨论的core.img的构成，core.img的压缩部分包括kernel.img和必要的模块，所以经过这次跳转后，GRUB跳转到了映像kernel.img的开头。

（4）kernel.img将自己复制回0x9000

因为Linux内核和initramfs可能被加载到内存从1MB开始的任何地方，所以GRUB要给它们指路。为此，GRUB虽然使用了1MB以上的区域作为解压使用的缓冲区，但是解压后要移动回1MB以下的区域。

我们看一下kernel.img的构建脚本：

grub-2.00/grub-core/Makefile.core.am:

```
if COND_i386_pc
platform_PROGRAMS += kernel.exec
kernel_exec_SOURCES  = kern/i386/pc/startup.S
kernel_exec_SOURCES += kern/generic/rtc_get_time_ms.c
term/i386/vga_common.c kern/i386/pc/init.c ...
...
```

```
kernel_exec_LDFLAGS  = $(AM_LDFLAGS) $(LDFLAGS_KERNEL) ...,0x9000
...
kernel.img: kernel.exec$(EXEEXT)
    ...$(STRIP) $(kernel_exec_STRIPFLAGS) -o $@ $< ...
endif
```

根据 kernel.img 的编译脚本可见，kernel.img 的开头是 startup.S，移动 kernel.img 的代码就在这个文件中：

```
grub-2.00/grub-core/kern/i386/pc/startup.S:

start:
_start:
...
#ifdef __APPLE__
    movl    $EXT_C(_edata), %ecx
    subl    $LOCAL(start), %ecx
#else
    movl    $(_edata - _start), %ecx
#endif
    movl    $(_start), %edi
    rep
    movsb

    movl    $LOCAL (cont), %esi
    jmp *%esi
LOCAL(cont):
    ...
    call EXT_C(grub_main)
```

根据 startup.S 的代码可见：

1）startup.S 调用 x86 的指令 movsb 移动映像。

2）寄存器 esi 中的值是移动的源地址。在解压 core.img 时，解压后的 core.img 的地址，即 1MB，已经保存在寄存器 esi 中了。

3）寄存器 edi 的值是移动的目的地址。在代码中，寄存器 edi 的值被设置为符号 _start 的地址，这个符号地址是多少呢？注意编译脚本中传给链接器的参数 kernel_exec_LDFLAGS，其请求链接器从 0x9000 开始为 kernel.img 分配地址，而 _start 恰位于 kernel.img 的开头，所以符号 _start 的地址是 0x9000。因此，kernel.img 就是将自身移动到 0x9000 处。

4）寄存器 ecx 的值是移动的字节数。从代码中计算 ecx 的值来看，startup.S 只移动从 _edata 到 _start 的这段指令和数据，而 _edata 是链接器定义的代表 kernel.img 的数据段结束的位置，也就是说，startup.S 只是将 kernel.img 移动到了 0x9000 处，并没有移动模块。在讨论 core.img 的构成时，我们已经谈到模块需要重定位，所以不能简单地进行移动。

5）在移动完 kernel.img 后，startup.S 再次使用跳转指令 jmp 跳转到了移动后的位置继续执行，并转入了 GRUB 真正的核心部分，即 C 语言写的函数 grub_main 处。

```
grub-2.00/grub-core/kern/main.c:
```

```
void __attribute__ ((noreturn)) grub_main (void)
{
  ...
  grub_load_modules ();
  ...
  grub_load_normal_mode ();
  ...
}
```

函数grub_main调用函数grub_load_modules装配模块，然后调用grub_load_normal_mode加载normal模块，这个模块拉开了加载Linux内核和initramfs的大幕。

5.1.4 加载内核和initramfs

normal模块读取并解析GRUB配置文件grub.cfg，然后根据grub.cfg中的具体命令，加载相应的模块。命令和模块的关系记录在文件command.lst中，通常GRUB将该文件被安装在/boot/grub/i386-pc目录下，normal模块加载时将加载这个文件。command.lst包括两列，第一列是命令，第二列是该命令所在的模块：

```
/boot/grub/i386-pc/command.lst:

initrd16: linux16
initrd: linux
keymap: keylayouts
kfreebsd_loadenv: bsd
kfreebsd_module: bsd
kfreebsd_module_elf: bsd
...
linux16: linux16
linux: linux
```

以下面的GRUB配置文件中的片段为例：

```
set root=(hd0,1)
linux /boot/bzImage root=/dev/sda1 ro quiet splash
initrd /boot/initrd.img
```

当normal模块遇到命令如“linux”、“initrd”时，将到文件command.lst中查找这些命令所在的模块。根据command.lst可知，命令“linux”、“initrd”都在模块linux中，因此，normal模块将加载linux模块。然后，调用linux模块中的命令“linux”、“initrd”，完成Linux内核以及initramfs的加载。本节中，我们通过分析这两个命令对应的回调函数，来探讨Linux内核和initramfs的加载。

1. 引导协议

Bootloader负责加载内核，显然Bootloader和内核之间需要分享一些数据。典型的比如Bootloader需要知道内核的保护模式部分希望加载到什么位置？内核是不是可重定位内核？从内核的角度，则需要清楚Bootloader将initramfs加载到了内存的什么位置、initramfs的尺

寸是多少等。

因此，内核和引导程序之间需要有个约定，这个约定称为引导协议（boot protocol），也称为 16 位引导协议（16-bit boot protocol）。该协议约定了 Bootloader 和内核之间分享的数据存储的位置、大小以及哪些由内核提供给 Bootloader，哪些由 Bootloader 提供给内核等。

在进入保护模式后，内核将不会再切换到实模式，而硬件相关的参数必须在实模式下借助 BIOS 中断获取。为此，在早期的内核中，内核中包含了一部分实模式代码，即 setup.bin，其主要功能之一就是为保护模式部分的代码获取硬件的信息，也就是内核中所说的零页（zero-page）中规定的信息。

随着新的 BIOS 标准，如 EFI、LinuxBIOS 等的出现，出现了 32 位引导协议（32-bit boot protocol）。在 32 位引导协议下，除了传统的 16 位协议，Bootloader 取代内核中实模式部分负责收集硬件信息（即零页信息）的功能。而且 Bootloader 会将 CPU 切换为保护模式，在内核和 initramfs 加载完成后，Bootloader 不再跳转到内核实模式部分，而是直接跳转到内核的保护模式部分。

下面我们就来看看在 32 位引导协议下，内核和 Bootloader 之间是如何分享信息的。

（1）内核向 Bootloader 传递信息

内核中引导协议的相关部分在文件 arch/x86/boot/header.S 中：

```
linux-3.7.4/arch/x86/boot/header.S:

    .section ".header", "a"
...
ramdisk_image:  .long   0

ramdisk_size:   .long   0
...
kernel_alignment:  .long CONFIG_PHYSICAL_ALIGN

relocatable_kernel:    .byte 1
...
pref_address:      .quad LOAD_PHYSICAL_ADDR
...
```

上面列出了几个典型的信息，其中 ramdisk_image 和 ramdisk_size 是由 Bootloader 负责填充的，告诉内核 initramfs 被加载到了内存的什么位置，占据多大空间。而 kernel_alignment、relocatable_kernel、pref_address 则由内核负责填充，告知 Bootloader 内核加载的对齐要求、内核是否是可以重定位的以及内核希望的加载地址等信息。

引导协议规定，协议数据从内核映像的偏移 0x1F1 处开始，所以在 header.S 中使用汇编伪指令 .section “.header” 指示编译器将引导数据所在的段定义为 “.header”，并在 setup.bin 的链接脚本中将此段安排在内核映像偏移 0x1F1 处：

```
linux-3.7.4/arch/x86/boot/setup.ld:
```

```
SECTIONS
{
    ...
    . = 497;
    .header      : { *(.header) }
    ...
}
```

setup.ld 指示链接器将段“.header”链接在地址 497 处，其十六进制即 0x1F1，这恰是引导协议约定的位置。GRUB 加载内核时，将首先从 setup.bin 中读取引导协议相关的信息。

对于零页中规定的信息，并不需要从内核传递给 Bootloader，所以 setup.bin 中定义的依然是传统的 16 位引导数据。

（2）Bootloader 向内核传递信息

Bootloader 向内核传递的信息，要比内核向 Bootloader 的传递复杂一些。因为除了传统的 16 位引导信息外，还需要向内核传递零页信息。Bootloader 和内核均为此定义了一个数据结构，通常将这个结构体称为引导参数（boot parameters）。GRUB 中的定义如下：

```
grub-2.00/include/grub/i386/linux.h:

struct linux_kernel_params
{
  grub_uint8_t video_cursor_x;       /* 0 */
  grub_uint8_t video_cursor_y;
  ...
  grub_uint8_t padding9[0x1f1 - 0x1e9];

  grub_uint8_t setup_sects;  /* The size of the setup in sectors */
  grub_uint16_t root_flags;   /* If the root is mounted readonly */
  ...
  struct grub_e820_mmap e820_map[(0x400 - 0x2d0) / 20]; /* 2d0 */

} __attribute__ ((packed));
```

在结构体 linux_kernel_params 中，从偏移 0x1F1 处，即成员 setup_sects 处，保存的传统的 16 位引导协议的数据。除此之外，结构体 linux_kernel_params 中保存就是零页信息了，如显示相关的信息、内存相关的信息等。

GRUB 在启动内核前，将创建一个结构体 linux_kernel_params 类型的变量 linux_params，首先从 setup.bin 中读取 16 位的引导数据到这个变量中，代码如下：

```
grub-2.00/grub-core/loader/i386/linux.c:

static struct linux_kernel_params linux_params;

static grub_err_t grub_cmd_linux (grub_command_t cmd __attribute__
   ((unused)), int argc, char *argv[])
{
  grub_file_t file = 0;
  struct linux_kernel_header lh;
  struct linux_kernel_params *params;
```

```
...
file = grub_file_open (argv[0]);
...
if (grub_file_read (file, &lh, sizeof (lh)) != sizeof (lh))
...
params = (struct linux_kernel_params *) &linux_params;
grub_memset (params, 0, sizeof (*params));
grub_memcpy (&params->setup_sects, &lh.setup_sects,
     sizeof (lh) - 0x1F1);
  ...
  params->ext_mem = ((32 * 0x100000) >> 10);
  params->alt_mem = ((32 * 0x100000) >> 10);
  ...
}
```

grub_cmd_linux 是模块 linux 中的命令 linux 的回调函数，跟在命令 linux 后的第一个参数就是内核映像文件，因此，这里的 argv[0] 就是内核映像文件。函数 grub_cmd_linux 从内核映像的开头读取了结构体 linux_kernel_header 大小的一块数据，结构体 linux_kernel_header 定义的就是传统的 16 位的引导数据：

```
grub-2.00/include/grub/i386/linux.h:

/* For the Linux/i386 boot protocol version 2.10.  */
struct linux_kernel_header
{
  ...
  grub_uint8_t code2[0x01F1 - 0x0020 - 2 - 2];
  grub_uint8_t setup_sects;  /* The size of the setup in sectors */
  ...
  grub_uint64_t pref_address;
  grub_uint32_t init_size;
} __attribute__ ((packed));
```

然后，grub_cmd_linux 将其复制到变量 linux_params 中。

GRUB 除了使用从内核中读取的信息，比如内核希望加载的地址，也将实际加载的情况（比如 initramfs 加载的位置）填充到变量 linux_params 中。另外，GRUB 还要按照 32 位启动协议规定，检测零页定义的信息，填充到变量 linux_params 中。比如在上面列出的函数 grub_cmd_linux 的代码片段中设置 linux_kernel_params 中的 ext_mem 和 alt_mem 等。再举一个典型的例子，在引导 Linux 内核前，GRUB 会将探测到的内存的信息记录在变量 linux_params 中：

```
grub-2.00/grub-core/loader/i386/linux.c:

static grub_err_t grub_linux_boot (void)
{
  ...
      if (grub_e820_add_region (params->e820_map, &e820_num,
                                addr, size, e820_type))
  ...
}
```

在内核中，引导信息定义的数据结构如下：

```
linux-3.7.4/arch/x86/include/asm/bootparam.h:

struct setup_header {
    __u8    setup_sects;
    __u16   root_flags;
    ...
} __attribute__((packed));
...
struct boot_params {
    struct screen_info screen_info;          /* 0x000 */
    ...
    struct setup_header hdr;    /* setup header */  /* 0x1f1 */
    ...
    struct e820entry e820_map[E820MAX];      /* 0x2d0 */
    ...
}
```

其中，结构体 setup_header 中记录的就是传统的 16 位引导信息，结构体 boot_params 对应于 GRUB 中的结构体 linux_kernel_params，即记录的是传统的 16 位的引导信息和零页信息。在初始化时，内核会将 GRUB 准备好的这些信息复制到内核的地址空间中，代码如下：

```
linux-3.7.4/arch/x86/kernel/head_32.S:

ENTRY(startup_32)
    ...
    movl $pa(boot_params),%edi
    movl $(PARAM_SIZE/4),%ecx
    cld
    rep
    movsl
    movl pa(boot_params) + NEW_CL_POINTER,%esi
    andl %esi,%esi
    jz 1f            # No command line
    movl $pa(boot_command_line),%edi
    movl $(COMMAND_LINE_SIZE/4),%ecx
    rep
    movsl
1:
    ...
```

内核重复调用汇编指令 movsl 进行复制。复制的源寄存器 esi 在 GRUB 中启动内核前设置指向 linux_kernel_params 对象，目的地址是内核中定义的结构体 boot_params 类型的变量 boot_params。

如果用户通过 GRUB 向内核传递了参数，即我们所说的 grub.cfg 中的命令行参数，则 GRUB 将这些参数保存在一块内存中，并设置引导参数结构体中的字段 cmd_line_ptr 指向这块内存，内核也要将这些参数从 GRUB 复制到内核。

2. 加载内核及 initramfs

理解了引导协议后，接下来看看 GRUB 是如何加载内核以及 initramfs 到内存的。

模块 linux 初始化时注册了两个命令，一个是命令 linux，另外一个是 initrd。命令 linux 的作用是加载 Linux 内核，其对应的回调函数是 grub_cmd_linux；命令 initrd 的作用是加载 initramfs，其对应的回调函数是 grub_cmd_initrd，代码如下：

grub-2.00/grub-core/loader/i386/linux.c:

```
GRUB_MOD_INIT(linux)
{
  cmd_linux = grub_register_command ("linux", grub_cmd_linux,
                     0, N_("Load Linux."));
  cmd_initrd = grub_register_command ("initrd", grub_cmd_initrd,
                     0, N_("Load initrd."));
  my_mod = mod;
}
```

（1）加载内核

前面提到过，在 32 位引导协议下，内核的实模式部分已经退化为仅负责承载引导协议，其功能部分已经被 GRUB 取代了，所以实模式部分无须再加载到内存了。GRUB 只需将内核的保护模式加载到内存即可，相关代码如下：

grub-2.00/grub-core/loader/i386/linux.c:

```
01 static void *prot_mode_mem;
02 static grub_addr_t prot_mode_target;
03
04 static grub_err_t grub_cmd_linux (...)
05 {
06   ...
07   grub_uint64_t preffered_address = GRUB_LINUX_BZIMAGE_ADDR;
08   ...
09   setup_sects = lh.setup_sects;
10   ...
11   real_size = setup_sects << GRUB_DISK_SECTOR_BITS;
12   prot_file_size = grub_file_size (file) - real_size -
13       GRUB_DISK_SECTOR_SIZE;
14   ...
15   if (grub_le_to_cpu16 (lh.version) >= 0x020a)
16     {
17       ...
18       if (relocatable)
19         preffered_address = grub_le_to_cpu64 (lh.pref_address);
20       else
21         preffered_address = GRUB_LINUX_BZIMAGE_ADDR;
22     }
23   ...
24   if (allocate_pages (prot_size, &align, min_align,
25                relocatable, preffered_address))
26   ...
27   params->code32_start = prot_mode_target + lh.code32_start
28       - GRUB_LINUX_BZIMAGE_ADDR;
29   ...
30   grub_file_seek (file, real_size + GRUB_DISK_SECTOR_SIZE);
```

```
31    ...
32    len = prot_file_size;
33    if (grub_file_read (file, prot_mode_mem, len) != len && ...)
34    ...
35 }
```

在讨论函数 grub_cmd_linux 前，首先解释代码中出现的两个容易混淆的变量——prot_mode_mem 和 prot_mode_target，见代码第 1 行和第 2 行。这两个变量很容易让人困惑，但是仔细观察它们的变量类型可以发现，prot_mode_mem 是指向一块内存的指针，所以读写内存操作应该使用这个指针。而 prot_mode_target 记录的仅仅是一个内存地址，典型的用作跳转指令的操作数。比如跳转到内核的保护模式部分时，指令 jmp 后接的操作数是内存地址，而不应使用指向内存的指针。

函数 grub_cmd_linux 执行的主要操作如下：

1）既然准备将内核映像加载到内存，函数 grub_cmd_linux 首先确定内核希望加载的地址，见代码第 15~22 行。以大于 0x020a 版本的引导协议为例，如果内核支持重定位，那么 GRUB 将从引导协议中读取的 pref_address 作为内核加载的位置。否则，将内核加载到位置 GRUB_LINUX_BZIMAGE_ADDR，该宏在 GRUB 中定义为 1MB：

```
grub-2.00/include/grub/i386/linux.h:

#define GRUB_LINUX_BZIMAGE_ADDR     0x100000
```

2）确定了加载地址后，grub_cmd_linux 调用函数 allocate_pages 为内核映像分配内存，见代码第 24~25 行。同时，函数 allocate_pages 设置指针 prot_mode_mem 指向为内核分配的内存，而将该内存的地址记录在变量 prot_mode_target 中。

3）函数 grub_cmd_linux 也将内核加载的物理地址，即变量 prot_mode_target 的值，记录在引导参数的成员 code32_start 中，见代码第 27~28 行。后面启动内核时的跳转命令将以 code32_start 记录的物理地址作为操作数。这里，GRUB 的开发者们应该是考虑了某些特殊情况，因为针对我们的具体情况，setup.bin 中的 code32_start 定义为 1MB：

```
linux-3.7.4/arch/x86/boot/header.S:

code32_start:                   # here loaders can put a different
                        # start address for 32-bit code.
        .long   0x100000    # 0x100000 = default for big kernel
```

而宏 GRUB_LINUX_BZIMAGE_ADDR 也是 1MB，所以 lh.code32_start 和 GRUB_LINUX_BZIMAGE_ADDR 是相互抵消的，变量 code32_start 的值就是 prot_mode_target。

4）准备好了内核映像加载的内存后，下面就要准备从硬盘将内核映像读入内存。因为实模式部分不需要加载，所以读取前需要将映像文件的指针定位到保护模式。显然只有内核知道自己的实模式部分有多大，因此，GRUB 需要从承载引导协议的 setup.bin 中读取实模式部分的尺寸。

最初，内核为了支持从软盘启动，实模式部分分为 bootsect 以及 setup 两部分。后来引导统一由 Bootloader 来负责，因此，内核从 2.6 版本开始把 bootsect.S 文件和 setup.S 文件合成为一个文件——header.S 文件。但是为了向后兼容，引导协议中记录 setup 部分大小的成员 setup_sectors 依旧被内核设置为实模式部分的尺寸再减去一个扇区，代码如下所示：

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

    buf[0x1f1] = setup_sectors-1;
```

代码中第 9 行就是读取 setup 部分占据的扇区数，第 11 行是将扇区转换为字节。

5）获取了实模式部分的大小后，下面就要将内核映像文件定位到保护模式开始的地方，第 30 行代码就是做这件事。其中 GRUB_DISK_SECTOR_SIZE 就是在 setup 的基础上再增加一个扇区的 bootsect。

6）在最后读取之前，还有一件事要做，那就是确定保护模式部分的尺寸。一旦确定了实模式部分的大小，保护模式的尺寸就非常容易计算了，即整个映像的尺寸减去实模式的尺寸（包括 setup 部分和 bootsect 部分），就是保护模式的大小，见代码第 12~13 行。

7）一切就绪，第 33 行代码加载内核。此时，GRUB 已经加载了驱动硬盘和文件系统的模块，所以，GRUB 不再是通过 BIOS 中断，而是通过自身的文件系统驱动提供的接口 grub_file_read 读取的内核映像。

（2）加载 initramfs

与加载内核类似，命令 initrd 对应的回调函数 grub_cmd_initrd 首先确定 initramfs 的加载地址，为 initramfs 分配好内存。然后调用 GRUB 中的文件系统驱动提供的接口 grub_file_read 从硬盘加载 initramfs。相关代码如下：

```
grub-2.00/grub-core/loader/i386/linux.c:

01 static grub_err_t grub_cmd_initrd (...)
02 {
03   ...
04   if (grub_le_to_cpu16 (linux_params.version) >= 0x0203)
05     {
06       addr_max=grub_cpu_to_le32(linux_params.initrd_addr_max);
07         ...
08     }
09   else
10     addr_max = GRUB_LINUX_INITRD_MAX_ADDRESS;
11   ...
12   addr_min = (grub_addr_t) prot_mode_target + prot_init_space
13              + page_align (size);
14
15   /* Put the initrd as high as possible, 4KiB aligned.  */
16   addr = (addr_max - size) & ~0xFFF;
17   ...
18     err = grub_relocator_alloc_chunk_align (relocator, &ch,
19               addr_min, addr, size, 0x1000,
```

```
20                GRUB_RELOCATOR_PREFERENCE_HIGH, 1);
21      ...
22      initrd_mem = get_virtual_current_address (ch);
23      initrd_mem_target = get_physical_target_address (ch);
24    ...
25    ptr = initrd_mem;
26    for (i = 0; i < nfiles; i++)
27      {
28        ...
29        if (grub_file_read (files[i], ptr, cursize) != cursize)
30          ...
31      }
32    ...
33    linux_params.ramdisk_image = initrd_mem_target;
34    linux_params.ramdisk_size = size;
35    ...
36  }
```

函数 grub_cmd_initrd 执行的主要操作如下：

1）grub_cmd_initrd 首先要确定 initramfs 加载的位置，见代码第 4~13 行。从引导协议 0x0203 版本开始，内核定义了加载 initramfs 的上限，以 Linux 内核 3.7.4 版本为例，其规定的 initramfs 的上限为 0x7fffffff：

```
linux-3.7.4/arch/x86/boot/header.S:

ramdisk_max:    .long 0x7fffffff
```

如果引导协议小于这个版本，则 GRUB 只需自己作主即可，GRUB 将 initramfs 加载的上限设置为 GRUB_LINUX_INITRD_MAX_ADDRESS：

```
grub-2.00/include/grub/i386/linux.h:

#define GRUB_LINUX_INITRD_MAX_ADDRESS   0x37FFFFFF
```

而对于下限，内核没有要求。但是根据代码可见，GRUB 将 initramfs 加载在内核映像之后。

2）函数 grub_cmd_initrd 调用函数 grub_relocator_alloc_chunk_align 在这个范围内找一个合适的位置，见代码第 18~20 行。根据传给函数 grub_relocator_alloc_chunk_align 的参数 GRUB_RELOCATOR_PREFERENCE_HIGH 可见，GRUB 采用的策略是尽可能的将 initramfs 加载到高地址处。

为 initramfs 分配完内存之后，grub_cmd_initrd 将指针 initrd_mem 指向为加载 initramfs 分配的内存，并将这块内存的物理地址记录到变量 initrd_mem_target 中，见代码第 22~23 行。这两个变量与前面讨论加载内核时见到的变量 prot_mode_mem 和 prot_mode_target 类似。最后这个内存地址是要分享给内核的，当然不能将 GRUB 中的一个内存指针传递给内核了。

3）确定了地址，并分配了内存后，函数 grub_cmd_initrd 调用 grub_file_read 将 initramfs 加载到内存 initrd_mem 处。这里 GRUB 考虑了可能存在多个 initrd 的情况，所以有个 for 循环。

4）最后，GRUB 将 initramfs 的尺寸、加载的位置记录到引导参数中，供内核寻找 initramfs

时使用，见代码第 33~34 行。这里，我们看到，传递给内核的 initramfs 的加载地址就是前面分配的内存的物理地址 initrd_mem_target。

3. 将控制权交给内核

在加载完内核映像和 initramfs 后，GRUB 完成了其作为操作系统加载器的使命，其将跳转到加载的内核映像，将控制权交给内核。相关代码如下：

grub-2.00/grub-core/loader/i386/linux.c:

```
01 static grub_err_t grub_linux_boot (void)
02 {
03   ...
04   params = real_mode_mem;
05
06   *params = linux_params;
07   ...
08   state.esi = real_mode_target;
09   state.esp = real_mode_target;
10   state.eip = params->code32_start;
11   return grub_relocator32_boot (relocator, state, 0);
12 }
```

基本上，GRUB 是运行在保护模式的，只有在使用 BIOS 时才切换到实模式，所以记录引导参数的全局变量 linux_params 的位置是随机的。因此，在启动内核前，GRUB 在传统的低端内存中申请了一块区域，将引导参数放置到传统的实模式占据的位置。函数 grub_linux_boot 中指向这块内存的指针是 real_mode_mem，并将这块内存的物理地址记录在变量 real_mode_target。最终，在跳转到内核之前，GRUB 会将 real_mode_target 记录到寄存器 esi 中，内核启动后，将从寄存器 esi 记录的这个地址复制引导参数。

第 6 行代码就是将变量 linux_params 的值复制到这块内存区域。

第 10 行设置了指令指针的地址为 code32_start，我们在讨论加载内核映像时已经见到了 code32_start，其就是内核保护模式加载的地址。最后，函数 grub_relocator32_boot 将调用函数 grub_relocator32_start 跳转到内核的保护模式处，代码如下：

grub-2.00/grub-core/lib/i386/relocator32.S:

```
01 #define CODE_SEGMENT    0x10
02 ...
03 VARIABLE(grub_relocator32_start)
04     ...
05     RELOAD_GDT
06     ...
07     .byte   0xea
08 VARIABLE(grub_relocator32_eip)
09     .long   0
10     .word   CODE_SEGMENT
11     ...
12 LOCAL(gdt):
13     /* NULL.  */
```

```
14      .byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
15
16      /* Reserved.  */
17      .byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
18
19      /* Code segment.  */
20      .byte 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x9A, 0xCF, 0x00
21
22      /* Data segment.  */
23      .byte 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x92, 0xCF, 0x00
24  LOCAL(gdt_end):
```

上面代码片段中第5行装载gdt寄存器，gdt的内容在第12~24行代码中定义。根据gdt的定义可见，gdt中定义了两个段，一个是代码段，另外一个是数据段。这两个段的基址都是0，段的长度是32位CPU线性地址空间的范围，即4GB。这两个段的唯一区别是代码段是只读的，而数据段具有读写权限。

继续向下看第7行代码，其中有一个字节的“0xea”，这个正是x86指令集中的长跳转指令之一的操作码，如表5-1所示。

表5-1 x86 jmp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	EA	jmp ptr16:32	A	长跳转指令，跳转地址直接在操作数中给出

根据表5-1可见，指令jmp的操作数是48位的，前16位是代码段CS的内容，后32位是指令指针EIP的内容。

显然，上述代码片段中跟在0xea之后的第10行的word类型的变量就是CS段的内容，我们看到这2个字节处的宏CODE_SEGMENT在第1行代码处定义，值为0x10，展开二进制为：

```
0001 0000
```

在保护模式下，寄存器CS中保存的是段选择子（Segment Selector），其格式如图5-6所示。

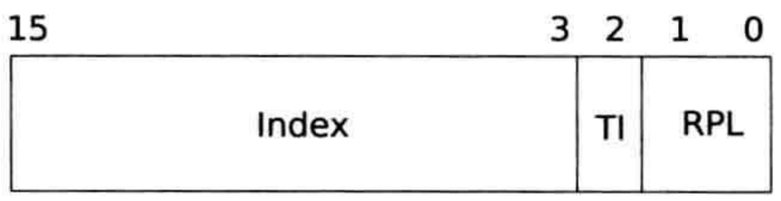


图5-6 段选择子格式

参照图5-6，除去最后三位，那么CS段在GDT中的索引就是二进制的10，十进制的2。而GDT表的下表从0开始，所以第2项正好是代码段。

第9行的long类型的变量就是EIP的内容，这个值是在函数grub_relocator32_boot中填充的，代码如下：

```
grub-2.00/grub-core/lib/i386/relocator.c:

grub_err_t grub_relocator32_boot (...)
{
  ...
  grub_relocator32_eip = state.eip;
```

```
    ...
}
```

看到“state.eip”是不是很熟悉？没错，它是在函数 grub_linux_boot 中设置的，值是 code32_start，也就是内核 32 位保护模式开始的地方。由此可见，函数 grub_relocator32_start 中的第 7~10 行代码，通过一个长跳转，GRUB 将控制权交给了内核。

5.2 解压内核

根据构建内核时的分析，我们知道，内核的保护模式部分包括非压缩部分以及压缩部分，压缩部分才是内核正常运转时的部分，而非压缩部分只是一个过客，其主要作用是解压内核的压缩部分，解压完成后，非压缩部分也将退出历史舞台。

内核的解压缩过程几经演进，现在的解压过程不再是首先将内核解压到另外的位置，然后再合并到最终的目的地址。而是采用了所谓的就地解压（in-place decompression）方法，内核解压时并不需要解压到另外的位置，从而避免覆盖其他部分的数据。

以不可重定位的内核的解压过程为例，其解压过程如图 5-7 所示。

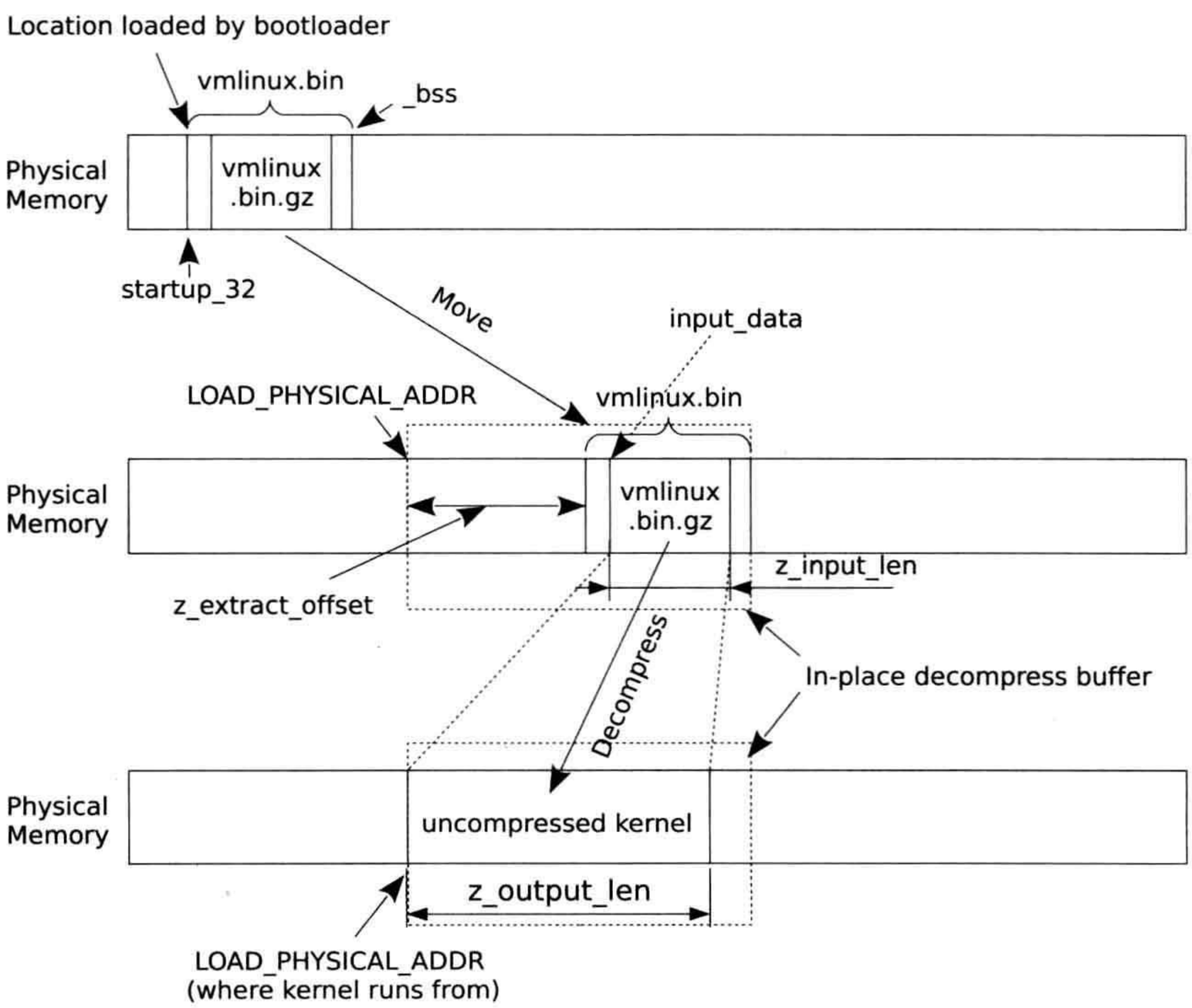


图 5-7 不可重定位内核的就地解压过程

对于不可重定位内核，最终解压后的的内核的起始位置是内核编译时设定的加载地址，

即 LOAD_PHYSICAL_ADDR。虽然解压的方式是就地解压，但是为了安全起见，解压过程所需要的内存空间并不完全等于解压后内核占据的空间，而是还预留有那么一点点安全空间。所以这个解压所需的空间，即图中标明的“In-place decompress buffer”的长度是解压后内核的长度 z_output_len 加上这个预留的安全空间。

为了确保在解压时，读取的位置永远在写入的位置的前面。内核首先移动到这个解压空间的最后。那么内核如何才能确保移动到这个空间的最后呢？内核只需从 LOAD_PHYSICAL_ADDR 向后移动 z_extract_offset，就确保了内核映像移动到了这个解压空间的最后。

那么 z_extract_offset 以及图 5-7 中的几个数据，包括解压后内核的长度 z_output_len 等数据，都是从哪里获取的呢？这些数据当然是压缩内核的时候最清楚了，因此这些早已在内核编译时，进行压缩时就已经计算好了，定义在内核映像中：

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

.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:
```

piggy.S 中定义的解压内核时需要的变量包括：

1）z_input_len，压缩内核的长度，即 vmlinux.bin.gz 的长度。

2）z_output_len，内核解压缩后的长度。

3）z_extract_offset，进行就地解压前，相对于解压后的位置，内核映像需要向后移动一段距离，为解压留出空间，避免解压的内核覆盖了压缩的内核，z_extract_offset 就是这个偏移的大小。

4）z_extract_offset_negative，这个是 z_extract_offset 的负数，是为了编程方便定义的。

5）input_data，标识内核映像中，压缩部分的起始位置。

在解压缩后，非压缩部分根据需要可能还要对内核进行重定位符号，然后跳转到解压后的内核的入口 startup_32。接下来，我们就具体讨论一下这个过程。

5.2.1 移动内核映像

1. 确定源地址

前面讨论 GRUB 时，我们看到虽然 GRUB 按照引导协议的规定，将内核保护模式部分

加载的地址，即 code32_start 写入了引导参数中，但是只有内核的“真正”部分投入运行时，内核才复制 GRUB 保存在低端内存的引导参数。也就是说，这个临时的负责解压部分，并没有复制 GRUB 保存的引导参数，因此，内核还需要自己计算映像被加载的地址。内核使用下面的代码获得当前被 Bootloader 加载的地址：

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

ENTRY(startup_32)
    ...
    call    1f
1:  popl    %ebp
    subl    $1b, %ebp
    ...
```

这里首先解释一下代码片段中的 1f。1 后面的 f 表示的是 forward，即以该条指令为参照，继续向前来寻找 1 这个标号；如果 1 的后缀是 b，则意义正好与此相反。

call 指令执行时，首先会将该调用返回后执行的下一条指令的地址压栈，这里就是标号 1 标识的指令运行时的地址。执行了 call 指令后，程序跳转到标号 1 所在的代码行处执行，标号 1 所在行的代码将栈顶的内容弹出到寄存器 ebp 中。而此时栈顶的内容恰恰是执行 call 调用前 CPU 压入的标号 1 处的指令的地址，也就是说，寄存器 ebp 中保存的就是标号 1 标识的指令在运行时所在的地址。接下来，减去标号 1 这行代码相对于程序开头处的偏移，即 $1b，就获得了函数 startup_32 的运行时地址。而函数 startup_32 就是内核开头，换句话说，就是获得了内核保护模式部分被 GRUB 实际加载的地址。

这段代码执行后，寄存器 ebp 中保存的就是内核的加载地址。

2. 确定目的地址

内核映像移动的目的地址针对内核是否支持重定位需要区别计算。

（1）内核被编译为可重定位

如果内核被编译为可重定位，理论上内核映像被加载的地址就可以作为最终的解压目的地址。但是出于效率角度的考虑，所以还要进一步检查 Bootloader 加载的地址是否符合内核的对齐要求。如果不符合要求，那么还要按照内核的对齐要求修正一下这个地址，然后再将其作为最终的内核解压的目的地址。代码如下所示：

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

#ifdef CONFIG_RELOCATABLE
    movl    %ebp, %ebx
    movl    BP_kernel_alignment(%esi), %eax
    decl    %eax
    addl    %eax, %ebx
    notl    %eax
    andl    %eax, %ebx
#else
    ...
```

```
#endif

    /* Target address to relocate to for decompression */
    addl    $z_extract_offset, %ebx
```

在上面代码中，#if 代码块的目的就是进行对齐修订。修订后的地址，也就是解压内核的目的地址，保存在寄存器 ebx 中。

然后，将内核解压后的地址再加上偏移 z_extract_offset，最后寄存器 ebx 中保存的即是内核映像在解压前应该移动到的位置。

如果需要配置一个可重定位的内核，则可按如下步骤进行：

1）执行 make menuconfig，出现如图 5-8 所示的界面。

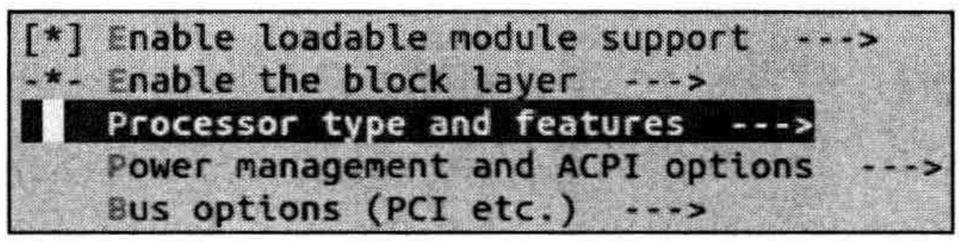


图 5-8　配置可重定位内核（1）

2）在图 5-8 中，选择菜单项“Processor type and features”，出现如图 5-9 所示的界面。

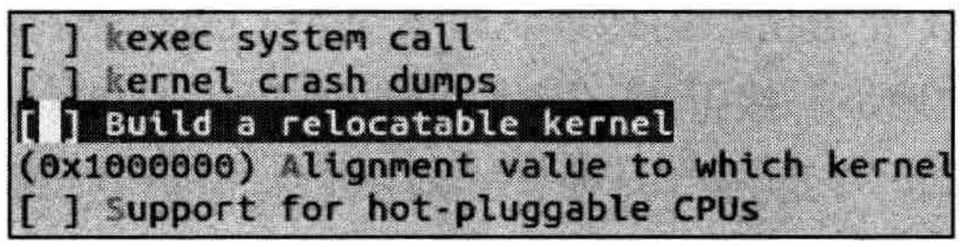


图 5-9　配置可重定位内核（2）

3）在图 5-9 中，选择“Build a relocatable kernel”。

在内核 3.7.4 版本中默认对齐为 0x1000000，当然也可以通过配置内核进行修改。

（2）内核不支持重定位

如果内核没有被编译为可重定位，那么表明内核不允许将其加载到其他位置，必须要加载到 LOAD_PHYSICAL_ADDR，因此内核的解压目的地址就是 LOAD_PHYSICAL_ADDR。代码如下：

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

#ifdef CONFIG_RELOCATABLE
    ...
#else
    movl    $LOAD_PHYSICAL_ADDR, %ebx
#endif

    /* Target address to relocate to for decompression */
    addl    $z_extract_offset, %ebx
```

然后，将解压的目的地址偏移 z_extract_offset，最后，寄存器 ebx 中保存的即是内核映像在解压前应该移动到的位置。

变量 LOAD_PHYSICAL_ADDR 是内核编译时指定的加载地址。其定义如下：

```
linux-3.7.4/arch/x86/include/asm/boot.h

#define LOAD_PHYSICAL_ADDR ((CONFIG_PHYSICAL_START \
                + (CONFIG_PHYSICAL_ALIGN - 1)) \
                & ~(CONFIG_PHYSICAL_ALIGN - 1))
```

可见，LOAD_PHYSICAL_ADDR 就是内核配置选项 CONFIG_PHYSICAL_START 按照内核的对齐要求修订后的值。

由这里可见，即使内核不允许重定位，那么事实上最后内核解压后的地址也是符合内核的对齐要求的，因为这里已经对编译时指定的加载地址进行了对齐处理。

内核 3.7.4 版本的默认加载地址是 0x1000000，用户可以通过配置指定内核加载的物理地址，步骤如下：

1）执行 make menuconfig，出现如图 5-10 所示的界面。

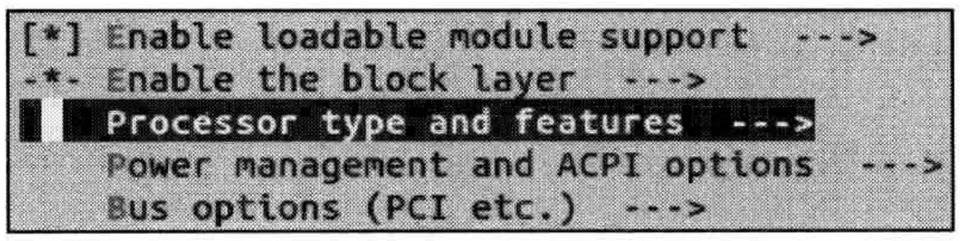


图 5-10 更改内核加载地址（1）

2）在图 5-10 中，选择菜单项“Processor type and features”，出现如图 5-11 所示的界面。

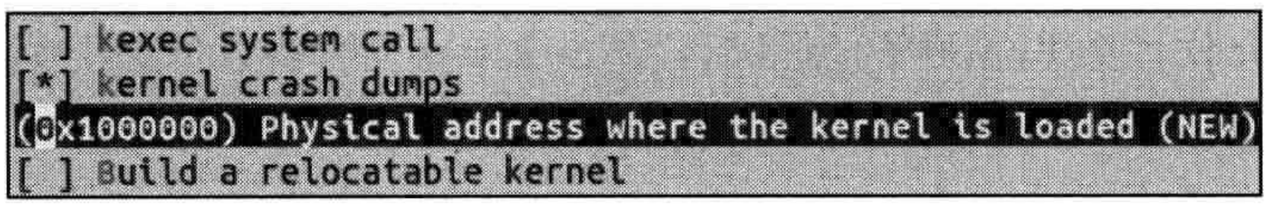


图 5-11 更改内核加载地址（2）

3）设置内核加载地址依赖于 CONFIG_EXPERT 或者 CONFIG_CRASH_DUMP，所以如果要修改内核依赖地址，必须选择这两项中的一项。可以在图 5-11 中选中“kernel crash dumps”，即 CONFIG_CRASH_DUMP，在该配置项的下面即可出现修改内核加载地址的配置项“Physical address where the kernel is loaded”，修改这一项的值即可达到修改内核加载地址的目的。

3. 移动内核映像

源地址和目的地址确定后，内核映像就开始了移动过程，代码如下：

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

01      pushl   %esi
02      leal    (_bss-4)(%ebp), %esi
03      leal    (_bss-4)(%ebx), %edi
04      movl    $(_bss - startup_32), %ecx
05      shrl    $2, %ecx
```

```
06      std
07      rep movsl
08      cld
09      popl    %esi
10      ...
11      leal    relocated(%ebx), %eax
12      jmp *%eax
13 ENDPROC(startup_32)
14
15      .text
16 relocated:
```

在上述代码中，符号 _bss 在链接 vmlinux.bin 时定义在 bss 段的开头。BSS 段被链接在 vmlinux.bin 的最后，而它在内核映像文件中并不占据空间，因此，符号 _bss 的地址就是内核保护模式的末尾。

寄存器 esi 是保存移动指令的源地址的，第 2 行代码就是设置这个寄存器的值，表达式：

```
(_bss-4)(%ebp)
```

展开后如下：

```
%ebp + _bss - 4
```

而寄存器 ebp 中保存的值是内核加载的地址，所以“%ebp + _bss”即为内核的末尾地址，-4 的目的当然是为第一次复制留出 4 字节的空间。

类似的，第 3 行代码是设置保存移动指令的目的地址的寄存器 edi。因为寄存器 ebx 中保存移动后的内核地址，所以 edi 中的值最后设置为移动后的内核的末尾地址，并为第一次复制留出 4 字节的空间。

同理，读者回忆一下 vmlinu.bin 的构建，链接脚本指定将函数 startup_32 链接在 vmlinux.bin 的最开头，因此，在第 4 行代码中，“_bss - startup_32”就是内核以字节为单位的长度了。因为，一次复制 4 字节，所以代表移动次数的寄存器 ecx 需要右移两位（即除以 4）。

第 6 行代码中，指令 std 的目的是表示每移动一次，esi 和 edi 分别减一（4 字节），也就是说复制是从内核尾部向着头部的方向复制。

内核移动结束后，显然需要重新装载指令指针 EIP 的值，跳转到移动后的内核中继续执行。这里是通过 jmp 指令修改指令指针的，即第 12 行代码，这条语句的目的是跳转到移动后的内核映像中标号 relocated 处继续执行。

5.2.2 解压

完成内核移动后，下一步就要开始解压内核了，代码如下：

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

01    leal    z_extract_offset_negative(%ebx), %ebp
02                /* push arguments for decompress_kernel: */
```

```
03    pushl   %ebp          /* output address */
04    pushl   $z_input_len    /* input_len */
05    leal    input_data(%ebx), %eax
06    pushl   %eax          /* input_data */
07    leal    boot_heap(%ebx), %eax
08    pushl   %eax          /* heap area */
09    pushl   %esi          /* real mode pointer */
10    call    decompress_kernel
```

我们看到，在 head_32.S 中，调用函数 decompress_kernel 解压内核，见第 10 行代码。decompress_kernel 是用 C 语言编写的，其函数定义在 misc.c 中：

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

asmlinkage void decompress_kernel(void *rmode, memptr heap,
                  unsigned char *input_data,
                  unsigned long input_len,
                  unsigned char *output)
```

我们结合函数 decompress_kernel 来看看文件 head_32.S 中的几个关键参数：

1）函数 decompress_kernel 的最后一个参数 output 是内核解压的目的地址，这个应该是第一个压栈的参数，见 head_32.S 代码第 1~3 行。在前面讨论移动内核时，我们看到，寄存器 ebx 中保存的是内核移动后的地址，而根据 piggy.S，z_extract_offset_negative 就是 z_extract_offset 的负数，所以表达式

```
z_extract_offset_negative(%ebx)
```

相当于

```
%ebx - z_extract_offset
```

参照图 5-7，显然，这就是内核解压的目的地址。

2）函数 decompress_kernel 的参数 input_len 表示压缩的内核映像的长度，这个变量对应 piggy.S 中定义的 z_input_len，这是第 2 个需要压栈的参数，见 head_32.S 代码片段第 4 行。

3）函数 decompress_kernel 的参数 input_data 表示压缩的内核映像的开头，对应 piggy.S 中定义的 input_data，这是第 3 个需要压栈的参数，见 head_32.S 代码片段第 5 行。

具体解压算法，我们不再分析，读者如果有兴趣，可自行阅读代码。

5.2.3 重定位

根据下面的内核链接脚本片段可见，内核中指令和数据的运行时地址是假定内核被解压到物理地址 LOAD_PHYSICAL_ADDR 处而分配的，我们称这个假定地址为理论加载地址。

```
linux-3.6.6/arch/x86/kernel/vmlinux.lds.S:

SECTIONS
{
```

```
#ifdef CONFIG_X86_32
        . = LOAD_OFFSET + LOAD_PHYSICAL_ADDR;
        phys_startup_32 = startup_32 - LOAD_OFFSET;
#else
    ...
}
```

如果内核被配置为可重定位的，那么尽管内核在引导协议中会将希望加载的地址（pref_address）设置为 LOAD_PHYSICAL_ADDR，如下代码：

```
linux-3.7.4/arch/x86/boot/header.S:

pref_address:     .quad LOAD_PHYSICAL_ADDR    # preferred load addr
```

但是内核并不能确保 Bootloader 将内核一定加载到内核建议的 LOAD_PHYSICAL_ADDR。如果 Bootloader 实际加载地址与理论加载地址不同，那么内核需要进行重定位。

对于可重定位内核，内核自身包含一个工具 relocs。在编译内核的最后，relocs 将 vmlinux 中需要重定位的符号导出，写入 vmlinux.relocs，然后 build 将其链接在内核的最后。简单来讲，vmlinux.relocs 就是一个数组，每一个元素记录的就是一个需要修订的位置。

head_32.S 中重定位的代码片段如下：

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

01 #if CONFIG_RELOCATABLE
02     ...
03     leal    z_output_len(%ebp), %edi
04     ...
05     movl    %ebp, %ebx
06     subl    $LOAD_PHYSICAL_ADDR, %ebx
07     jz  2f  /* Nothing to be done if loaded at compiled addr. */
08     ...
09 1: subl    $4, %edi
10     movl    (%edi), %ecx
11     testl   %ecx, %ecx
12     jz  2f
13     addl    %ebx, -__PAGE_OFFSET(%ebx, %ecx)
14     jmp 1b
15 2:
16 #endif
17     ...
18     jmp *%ebp
```

该代码片段执行的主要操作如下：

1）首先需要判断内核是否需要重定位，见代码第 5~7 行。在前面为解压缩函数准备参数时，寄存器 ebp 中记录了内核解压后的地址，所以这两行代码的目的就是比较内核解压后的地址与内核理论加载地址 LOAD_PHYSICAL_ADDR。如果相同，那么无须进行重定位，直接跳到标号 2 处，也就是跳过了标号 1 和标号 2 之间的重定位代码。

2）如果需要重定位，那么首先需要找到重定位表 vmlinux.relocs，见第 3 行代码。在编

译时，内核构建脚本将重定位表链接在映像的最后，而 z_output_len 代表内核解压后的长度，因此，%ebp + z_output_len 指向的就是重定位表的末尾。

3）找到重定位表后，就可以进行重定位了，代码第 9~14 行从后向前遍历重定位表，逐项进行修订。其中第 9~12 行代码判断重定位是否已经完成，重定位表以 0 开头，所以，当某一项的值为 0 时，就说明已经到了重定位表的表头，所有需要重定位的条目已经完成。具体的修订算法非常直接，就是在每个修订的位置，加上内核实际加载的地址与理论加载地址的差值，见第 13 行代码。但是这行指令的操作数使用了相对复杂一点的寻址方式，而且两次出现了寄存器 ebx，所以容易让人困惑。

首先来看一下寄存器 ebx 的值。事实上，在执行第 6 行代码时，内核实际的加载地址与理论加载地址的差值已经被保存到了寄存器 ebx 中。也就是说，用来修订的差值已经准备就绪。那么显然，第 13 行代码中指令 addl 的第二个操作数：

```
-__PAGE_OFFSET(%ebx, %ecx)
```

就是需要修订的位置，我们将其展开：

```
%ebx + %ecx - __PAGE_OFFSET
```

为了看得更清楚一点，我们换个写法：

```
(%ecx - __PAGE_OFFSET) + %ebx
```

我们来分析这个表达式，寄存器 ecx 就像一个局部变量，临时存储重定位表中每一项，即需要重定位的位置，那么为什么要从 ecx 中刨除 __PAGE_OFFSET 呢？这个问题的根源就在于重定位表 vmlinux.relocs 中记录的修订位置使用的是虚拟地址。

内核为了占据 3GB 以上的进程地址空间，所以在编译时，链接器为每个符号的地址增加了 3GB 的偏移，也就是这里的 __PAGE_OFFSET。在内核运行时，页式映射会将这个逻辑上的偏移消除，将符号映射到真正的物理地址。

但此时的麻烦是，CPU 尚未开启页式映射，而且 GRUB 将 CPU 设置工作在平坦内存模型下，段基址都为 0，虚拟地址经过 MMU 映射后，将原封不动地转换为物理地址。举个例子，假设内核最终加载到了 16MB 处，那么内核开头的虚拟地址是 0xc1000000，假设这里需要修订，那么重定位表中记录的修订位置是 0xc1000000。当进行重定位时，如果不做任何处理，这个地址经过 MMU 转换后的物理地址依然为 0xc1000000，显然多了 3GB 的偏移。因此，重定位代码需要事先将这个偏移消除掉。这就是在寄存器 ecx 的基础上再减去 __PAGE_OFFSET 的原因。

但是仅仅减去 __PAGE_OFFSET 还不够，不仅修订位置处的内容需要进行修订，修订位置本身也发生了变化，如图 5-12 所示，上面的虚线表示的是内核理论加载的地址，下面的实线表示的是内核实际加载的地址。

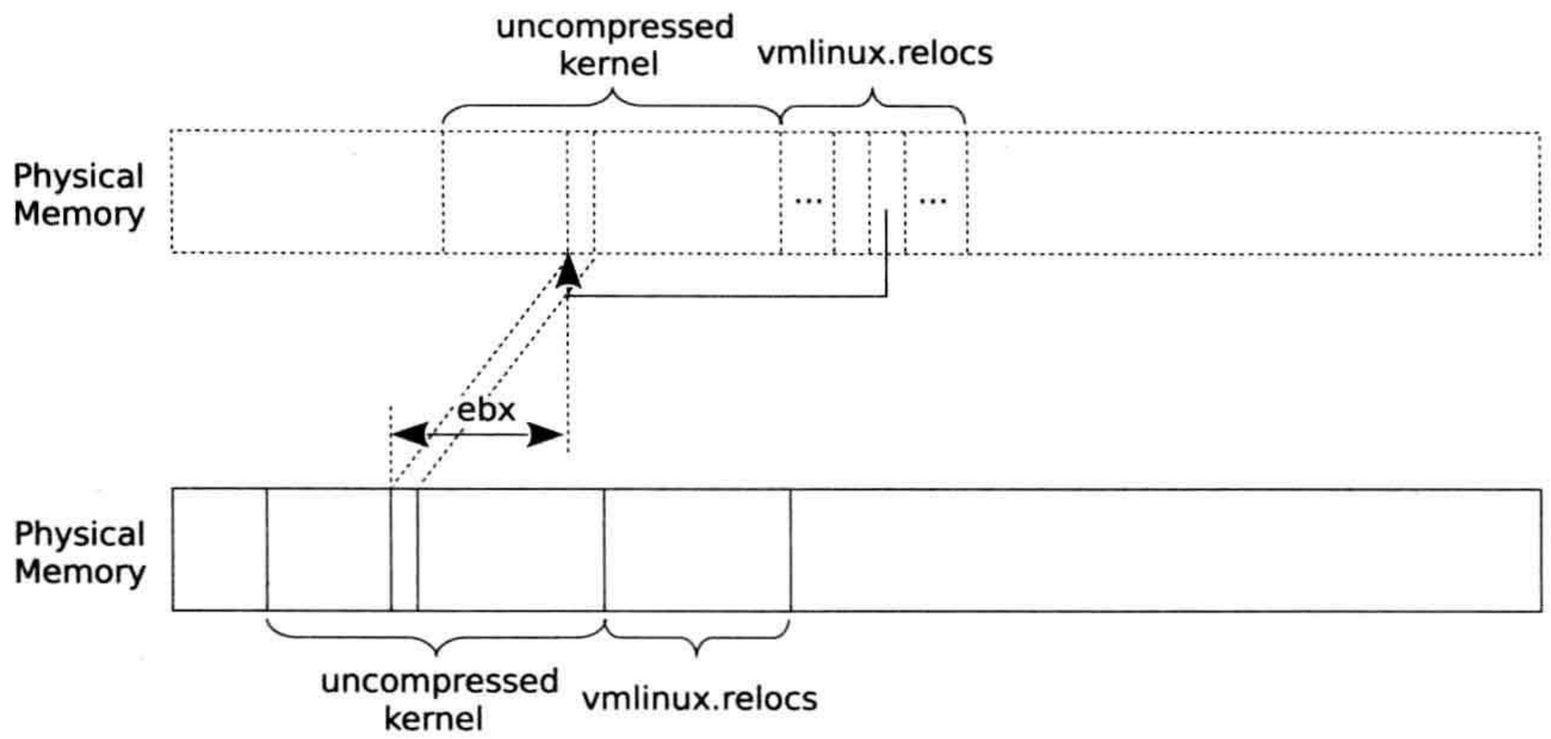


图 5-12 重定位位置的变化

因此，修订位置本身也需要进行修订。修订位置本身的偏移就是内核理论加载位置和实际加载位置的差值，而这个差值已经保存在寄存器 ebx 中，这就是为什么修订位置除了减去 __PAGE_OFFSET 外，又加上 ebx 的原因。

重定位完成之后，跳转到解压后的内核起始地址处继续执行，见重定位代码片段中的第 18 行。

5.3 内核初始化

虽然操作系统的功能包括进程管理、内存管理、设备管理等，但是操作系统的终极目标是创造一个环境，承载进程。但是由于进程运行时，可能需要和各种外设打交道，因此，操作系统初始化时，也会将这些外设等子系统进行初始化，这也导致内核初始化过程异常复杂。虽然这些过程很重要，但是忽略它们并不妨碍理解操作系统的本质。本节我们并不关心这些子系统的初始化，比如 USB 系统是如何初始化的，我们只围绕进程来讨论内核相关部分的初始化。

5.3.1 初始化虚拟内存

相对于单任务来说，多任务的好处无需赘言，但是多任务也对操作系统提出了更多的要求，其中一个主要问题就是如何在多个任务间互不干扰地共享同一物理内存。正如 Dennis DeBruler 说过的经典的一句话：计算机科学中所有问题都可以通过多一个间接层来解决。现代操作系统设计了虚拟内存机制支持多任务。

通过虚拟内存机制，多个进程之间就可以和平共享物理内存。每个进程都有了自己独立的虚拟地址空间，感觉就像自己独占物理内存一样。在某一个进程中访问任何地址都不可能访问到另外一个进程的数据，这使得任何一个进程由于执行错误指令或恶意代码导致的非法内存访问都不会意外改写其他进程的数据，也不会影响其他进程的运行，从而保证整个系统

的稳定性。进程本身不必关心虚拟地址是如何映射到物理内存以及存储在物理内存的什么位置，完全由操作系统替其打理。

为了支持虚拟内存，操作系统不必孤军奋战。现代的处理器几乎都从硬件的角度设计了支持虚拟内存的机制，以 x86 架构为例，其引入了 MMU 单元。但是与其他 CPU 几乎全部采用分页机制支持实现虚拟内存不同，对于 x86 体系架构来说，由于历史的原因，事情有点复杂。最初，8086 的寄存器是 16 位的，可以寻址 64KB 内存，为了在不改变寄存器和指令位数的情况下支持更大的寻址空间，Intel 的工程师们设计了一种段式寻址机制。后来，为了向后兼容，也就是保证为更早的体系结构开发的程序依旧可以在新的体系架构上运行，在后续的 x86 系列处理器上，Intel 保留了段式寻址机制，而且不能关闭段式机制。因此，对于 x86 架构来说，虚拟内存向物理内存的转换需要经过两个阶段，如图 5-13 所示。

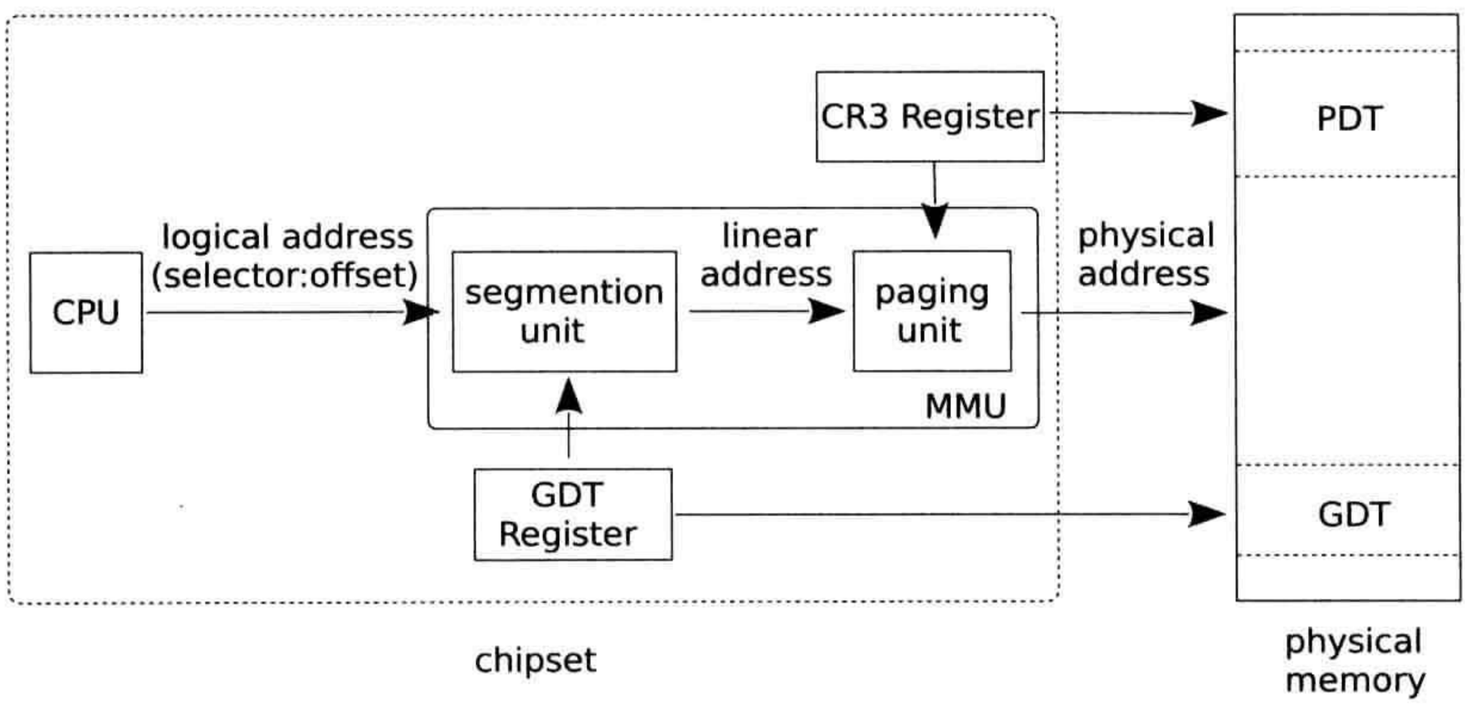


图 5-13　x86 架构虚拟内存向物理内存转换

1）逻辑地址转换为线形地址。CPU 将逻辑地址发送给 MMU。逻辑地址分为两部分：16 位的段选择子和 32 位的段内偏移。当把这 48 位地址传给 MMU 时，MMU 中的分段单元根据 16 位段选择子，从 GDT 表中获取对应段，取出段基址，再加上逻辑地址中的 32 位的偏移，就形成了线性地址。

2）线形地址转换为物理地址。分段单元将线性地址发送给分页单元，分页单元通过页表，将线性地址转换为物理地址。

通过虚拟内存，同一个虚拟地址可以映射到不同的物理内存。这也是多个进程共享同一个物理内存的理论基础，如图 5-14 所示。

显然，为了支持 MMU 进行地址转换，操作系统需要为 MMU 准备 GDT 以及页表，下面我们就讨论这两个过程。

1. 创建 GDT

分段机制是 x86 系列处理器演变发展过程中向后兼容的产物，更重要的是，页式映射已经完全可以非常好地支持虚拟内存机制了，除了增加实现的复杂度，分段机制已经没有存在的意义了。除了 IA 架构，其他体系结构几乎没有使用段机制的。但是为了向后兼容，又不

能关闭段机制，IA 架构提出了一种特殊的内存管理模型——平坦内存模型（flat model），如图 5-15 所示。

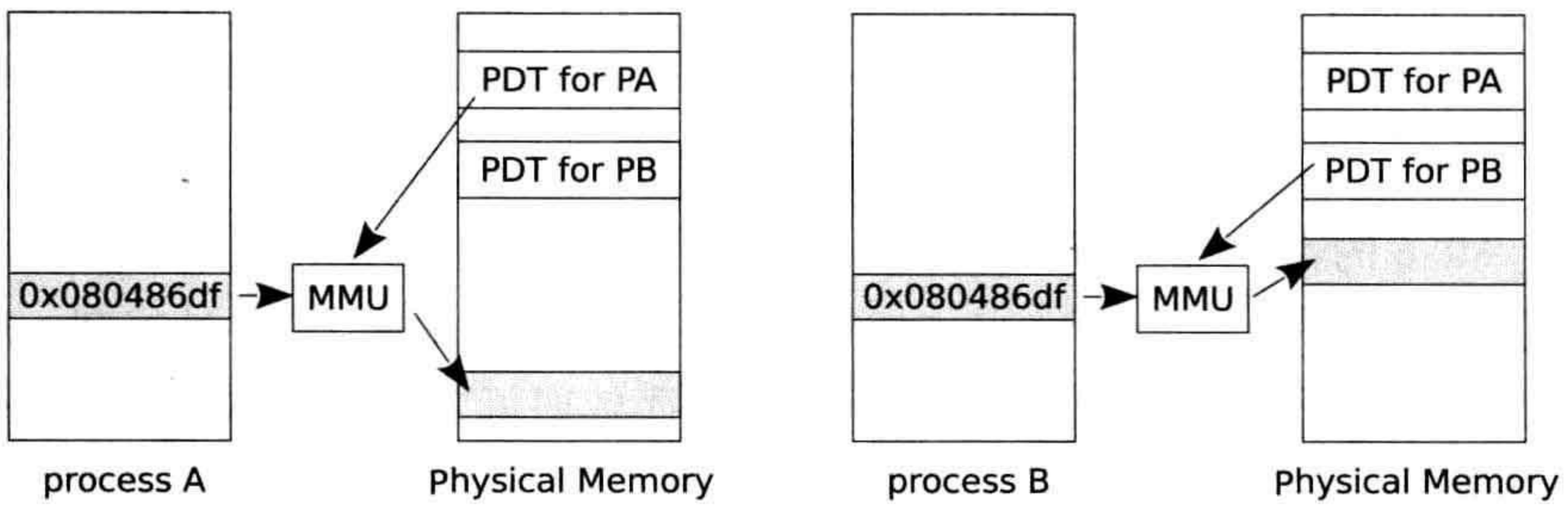


图 5-14　虚拟内存映射

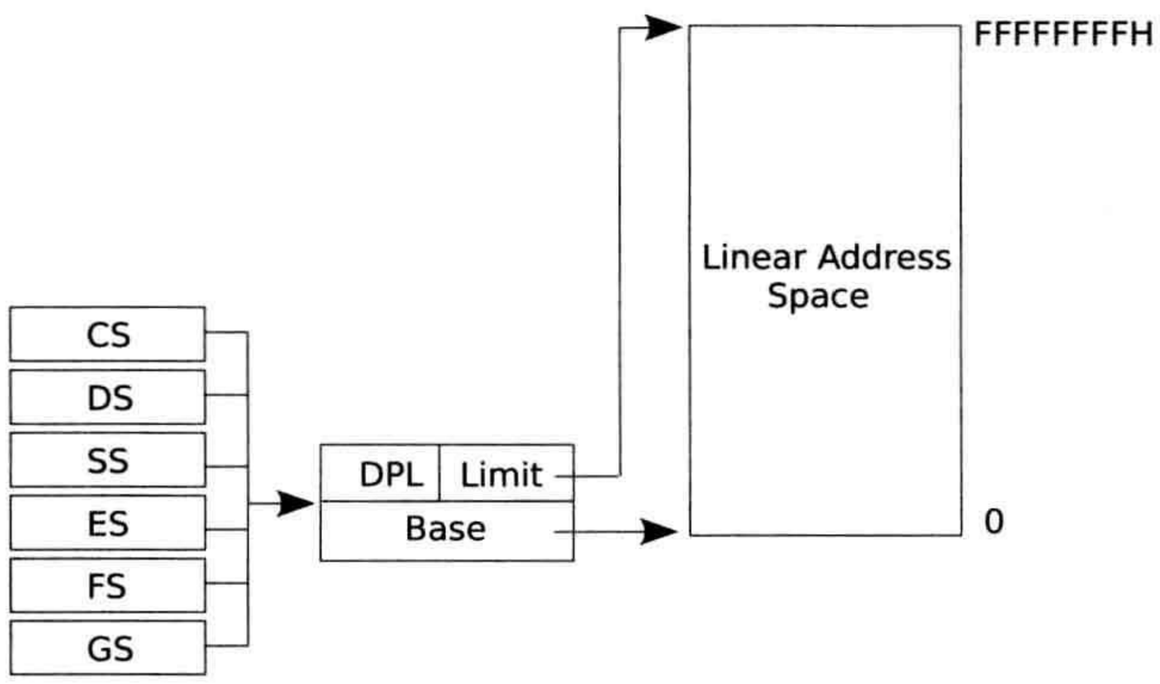


图 5-15　平坦内存模型

当使用平坦内存模型时，所有段的基址均为 0，段长为线性地址空间的整个长度。读者可能心存疑问：如果段基址相同，那么同一进程的不同段之间的地址是否会发生重叠？这点大可不必担心，虽然各个段的段基址都从 0 开始，但是在编译时，链接器会通过段内偏移控制各个段的内容不会彼此覆盖。

平坦内存模型就像功夫中的太极，将分段这个“麻烦”化解于无形。在平坦内存模型下，MMU 中的分段单元对地址的变换没有任何影响，编译好的二进制程序中的偏移地址（或者称为虚拟地址）完全等同于线性地址。平坦内存模型不仅简化了操作系统中的内存管理，而且也大大降低了编译器和链接器实现的复杂度。

本质上，平坦内存模型并不是一种什么特殊的模式，只是保护模式下的一种特例而已，其中的关键就在于段的基址和段的长度的设置。在内核初始化代码中，有两处设置并加载了 GDT。第一处是函数 startup_32，代码如下：

```
linux-3.7.4/arch/x86/kernel/head_32.S:

01 ENTRY(startup_32)
```

```
02      ...
03      testb $(1<<6), BP_loadflags(%esi)
04      jnz 2f
05
06      lgdt pa(boot_gdt_descr)
07      ...
08 2:
09      ...
10 boot_gdt_descr:
11      .word   BOOT DS+7
12      .long boot_gdt - __PAGE_OFFSET
13      ...
14 ENTRY(boot_gdt)
15      .fill GDT_ENTRY_BOOT_CS,8,0
16      .quad 0x00cf9a000000ffff /* kernel 4GB code at 0x00000000 */
17      .quad 0x00cf92000000ffff /* kernel 4GB data at 0x00000000 */
```

内核首先检查引导协议中的 loadflags 的第 6 位，即 KEEP_SEGMENTS 位。如果 Bootloader 没有设置这一位，那么内核需要重新装载各个段寄存器，包括 GDT 寄存器 gdtr，第 6 行代码就是重新设置 GDT 寄存器 gdtr，使其指向 boot_gdt_descr，而 boot_gdt_descr 中又包含了符号 boot_gdt。我们来看一下内核中的这两个符号的值：

```
vita@baisheng:/vita/build/linux-3.7.4$ readelf -s vmlinux \
                                       | grep boot_gdt
 24312: c1378d86     0 NOTYPE  GLOBAL DEFAULT   13 boot_gdt_descr
 29580: c1378dc0     0 NOTYPE  GLOBAL DEFAULT   13 boot_gdt
```

这两个符号的值均以 C 开头，显然都是虚拟地址了。但是，问题是此时 CPU 尚未开启分页，所以不能使用虚拟地址寻址，而只能使用物理地址寻址。所以在第 6 行代码中，使用宏 pa 将符号 boot_gdt_descr 的虚拟地址转化为物理地址，稍后我们会具体讨论这个宏，其主要作用就是将符号中的 3GB 偏移去掉。同理，注意第 12 行代码，在使用符号 boot_gdt 时，也去除了 3GB 偏移。因此，此后直到下一次重新装载，寄存器 gdtr 中将始终记录的是符号 boot_gdt_descr 的物理地址。

但是在 CPU 开启了分页后，应该使用虚拟地址寻址了。所以，在开启分页后，内核还需要重新装载寄存器 gdtr，将其中的物理地址替换为 GDT 描述符的虚拟地址，这就是内核两次加载 gdtr 寄存器的原因。因为这个 boot_gdt 只是临时的 GDT，够用就可以了，所以我们看到 boot_gdt 非常简单。

内核中第二次加载寄存器 gdtr 的代码如下：

```
linux-3.7.4/arch/x86/kernel/head_32.S:

01 ENTRY(startup_32)
02      ...
03      movl $pa(initial_page_table), %eax
04      movl %eax,%cr3       /* set the page table pointer.. */
05      movl %cr0,%eax
06      orl  $X86_CR0_PG,%eax
```

```
07      movl %eax,%cr0       /* ..and set paging (PG) bit */
08      ...
09      lgdt early_gdt_descr
10      ...
11 ENTRY(early_gdt_descr)
12      .word GDT_ENTRIES*8-1
13      .long gdt_page           /* Overwritten for secondary CPUs */
14      ...
```

在开启分页机制后，虚拟内存已经初始化完成，内核不必再使用汇编语言将虚拟地址使用宏 pa 手工转化为物理地址了，可以直接使用符号的虚拟地址了，如第 9 行代码使用的符号 early_gdt_descr 以及第 13 行代码使用的符号 gdt_page，使用的全部是符号的虚拟地址，MMU 会完成虚拟地址到物理地址的转换。换句话说，内核不必再使用汇编语言“精确”地指挥 CPU 了，可以使用更容易维护的 C 语言了，所以内核使用 C 语言重新定义了更完善的 GDT：

linux-3.7.4/arch/x86/kernel/cpu/common.c:

```
DEFINE_PER_CPU_PAGE_ALIGNED(struct gdt_page, gdt_page) =
{ .gdt = {
    [GDT_ENTRY_KERNEL_CS] = GDT_ENTRY_INIT(0xc09a, 0, 0xfffff),
    [GDT_ENTRY_KERNEL_DS] = GDT_ENTRY_INIT(0xc092, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER_CS] = GDT_ENTRY_INIT(0xc0fa, 0,
                                  0xfffff),
    [GDT_ENTRY_DEFAULT_USER_DS] = GDT_ENTRY_INIT(0xc0f2, 0,
                                  0xfffff),
...
} };
```

宏 GDT_ENTRY_INIT 用来构建一个全局描述符，定义如下：

linux-3.7.4/arch/x86/include/asm/desc_defs.h:

```
#define GDT_ENTRY_INIT(flags, base, limit) { { { \
 .a = ((limit) & 0xffff) | (((base) & 0xffff) << 16), \
 .b = (((base) & 0xff0000) >> 16) | (((flags) & 0xf0ff) << 8) | \
         ((limit) & 0xf0000) | ((base) & 0xff000000), \
    } } }
```

参照图 5-16 所示的段描述符的定义。

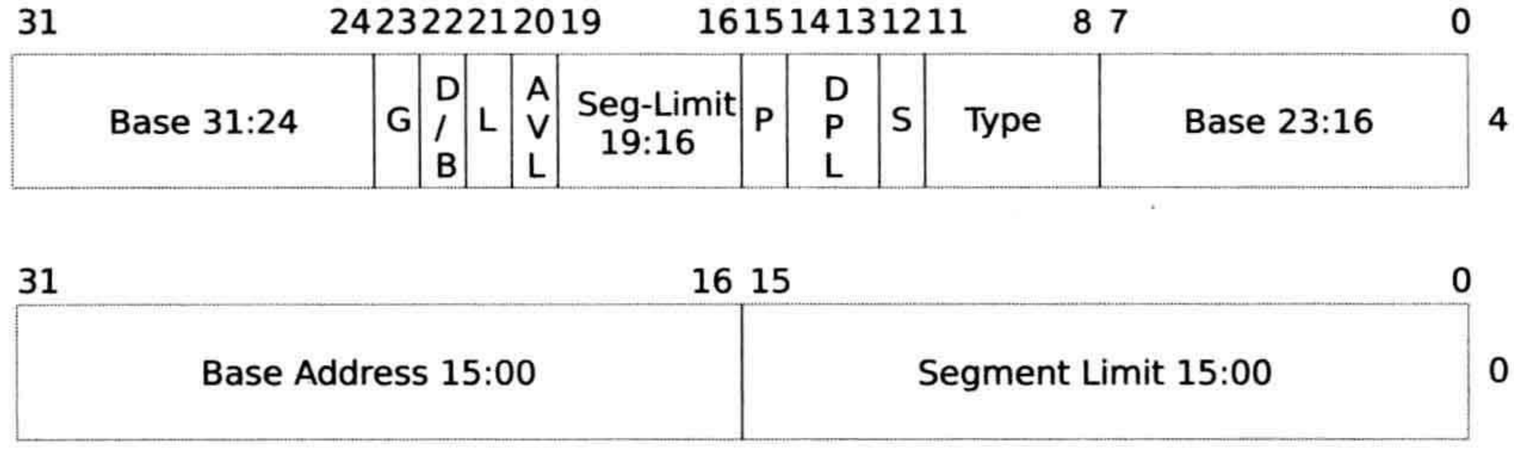


图 5-16 段描述符定义

可见，所有段的基址都是 0，上限都是 0xfffff。正如我们前面讨论的，Linux 使用了平坦内存模型。各个段仅有属性有一些差别，如表 5-2 所示。

表 5-2 内核代码段和数据段的属性

Segment/Flags	G	D/B	L	AVL	P	DPL	S	Type
Kernel CS / 0xc09a	1	1	0	0	1	00	1	1010
Kernel DS / 0xc092	1	1	0	0	1	00	1	0010
User CS / 0xc0fa	1	1	0	0	1	11	1	1010
User DS / 0xc0f2	1	1	0	0	1	11	1	0010

其中不同的字段是 DPL 和 Type。

DPL 表示段的特权级。内核的代码段和数据段的特权级是最高的 0，而用户代码和数据段的特权级是最低的 3。显然，从保护的角度，内核将段划分为内核空间和用户空间。

另外一个不同的是 Type。对于代码段，包括内核和用户空间的，其类型为 1010b，表示只具有可读权限。数据段的类型为 0010b，表示具有读写权限。显然这也是从保护角度考虑的，试图写代码段将激发 Segment Fault 类型的错误。

理论上，如果使用平坦内存模型，GDT 中只定义一个段描述符就可以了，所有段都使用这一个段描述符，但是出于保护的目的，代码是不允许随意改写的，因此内核定义了代码段和数据段。同样出于保护的目的，内核是不允许用户空间的程序随意访问内核空间的，所以内核又定义了内核段和用户段。最终内核定义了内核代码段、内核数据段、用户代码段和用户数据段。

正如同在平坦内存模型下，链接器通过偏移地址控制代码和数据占据的空间，链接器也需要通过偏移地址控制内核和用户程序占据的地址空间。在 Linux 系统上，约定内核占用 3GB~4GB 的地址空间，而应用程序使用 0~3GB。

那么内核是如何将自己的地址空间限制在 3GB 以上的呢？如同普通应用程序一样，内核符号的地址也是编译时链接器分配的，查看链接内核时链接器使用的链接器脚本：

```
linux-3.7.4/arch/x86/kernel/vmlinux.lds.S:

SECTIONS
{
#ifdef CONFIG_X86_32
        . = LOAD_OFFSET + LOAD_PHYSICAL_ADDR;
        phys_startup_32 = startup_32 - LOAD_OFFSET;
#else
    ...
}
```

其中 LOAD_PHYSICAL_ADDR 是假定的内核在物理内存中的实际加载位置，而 LOAD_OFFSET 就是人为让内核在线形地址空间中的偏移，其定义如下：

```
linux-3.7.4/arch/x86/kernel/vmlinux.lds.S:
```

```
#define LOAD_OFFSET __PAGE_OFFSET
```

```
linux-3.7.4/arch/x86/include/asm/page_32_types.h:
```

```
#define __PAGE_OFFSET        _AC(CONFIG_PAGE_OFFSET, UL)
```

```
linux-3.7.4/.config:
```

```
CONFIG_PAGE_OFFSET=0xC0000000
```

可见对于IA32来说，这个偏移默认是3GB。

我们看到，如果不增加偏移LOAD_OFFSET，内核中指令的起始地址就是LOAD_PHYSICAL_ADDR。如果内核在内存中也是实际加载到了LOAD_PHYSICAL_ADDR，那么指令或者数据的地址就是物理地址。

而在平坦内存模型下，在未开启分页时，CPU送给MMU的逻辑地址经过MMU转换后，偏移地址将原封不动的作为物理地址送到总线上：

物理地址 = 偏移地址 + 段基址（0）= 偏移地址

显然，这要求CPU送出的偏移地址就是物理地址。但事实上，内核中指令和数据的地址在链接时都增加了偏移LOAD_OFFSET。因此，在没有开启分页机制前，如果使用内核中的符号，必须要减去偏移LOAD_OFFSET。因此，内核中定义了宏pa，其目的就是将逻辑地址去除人为安排的3GB偏移。宏pa的定义如下：

```
linux-3.7.4/arch/x86/kernel/head_32.S:
```

```
#define pa(X) ((X) - __PAGE_OFFSET)
```

如在head_32.S中，寻址boot_gdt_descr、initial_page_table等符号时，因为尚未开启分页，所以均使用了宏pa。

而在CPU开启分页机制后，通过页表的映射，这个3GB的偏移将被消除。因此，在第二次加载gdtr，引用符号early_gdt_descr时，就不再需要进行任何手动的转换了。

2. 创建内核页表

前面我们讨论了内核为地址转换过程中MMU的分段单元准备GDT的过程。这里我们来讨论内核为MMU中负责第二阶段的地址转换的分页单元准备的页表。

如果操作系统永远在内核空间运行，那么页表只需要覆盖内核空间就可以了。但是，最终操作系统一定是要运行进程的，对于一个进程来说，它大部分时间运行在用户空间，但是有时也要在内核空间运行，因此进程访问的空间是整个线形地址空间，包括用户空间和内核空间。

虽然进程的用户空间“岁岁年年人不同”，但是内核空间却是“年年岁岁花相似”。操作系统只有一个内核，所以理论上，进程的页表只映射用户空间就可以了，进程运行在用户空间时，使用这个页表，而当进程切入内核空间时，CR3寄存器指向内核页表。但是，看似

只是一个寄存器的装载动作，其背后的代价却是非常高昂的。因为地址空间的切换，会导致TLB被清空，TLB中缓存的是虚拟地址到物理地址的映射，它可以大大提高虚拟地址到物理地址的转换速度。而且，进程在用户空间和内核空间的切换还是比较频繁的，比如，一个系统调用就会导致进程切换到内核空间。因此，Linux操作系统采用了这样一个冗余策略，在每个进程的页表中都包含了相同的内核空间映射部分。这样，当进程在用户空间和内核空间切换时，不必重新装载CR3寄存器。

在内核刚刚开始初始化时，内存尚未进行完全的初始化，所以内核将页表的初始化分成两个阶段进行。在开启页式映射前，内核还只能使用汇编语言。在准备基本的运行环境中，你愿意使用汇编语言考虑各种负责的情况吗？当然不愿意了，我们当然希望尽早地准备好可以使用C语言的环境。因此，这时内核仅建立一个小的够用的临时页目录和页表。在页式映射开启后，内核就可以毫无顾忌地使用C语言编写的代码了，于是内核进行内存的初始化。在内存子系统初始化完全完成后，内核再调用C语言函数建立完整的页表。

通常，内核创建的这个页目录和页表也被称为主内核页目录和页表，它们也作为进程的页目录和页表的模板。每当进程创建页表时，其将从主内核创建的这部分页目录和页表复制页目录项和页表项。当内核的内存映射发生变化时，内核将更新主内核页目录和页表，同时，也同步进程的页目录和页表中映射内核的页目录项和页表项。而对于页表的用户空间部分，因为与具体进程密切相关，因此由具体进程运行时按需创建。

在本小节中，我们讨论了内核第一阶段手工创建页表的过程，后面第二阶段使用C语言构建页表的过程与此原理完全相同，只不过高度自动化了，我们不再重复。

（1）页目录和页表的存储位置

最初，页目录的位置存放在BSS段中的变量swapper_pg_dir处。在建立页目录表时，除了需要将页目录中映射内核空间的部分映射到内核占据的物理内存外，还要把页目录中映射用户空间的最初部分也映射到内核占据的物理内存。内核为什么要这么做呢？这是x86架构明确要求的，在Intel的手册中明确规定了CPU切换到保护模式，并开启页式映射时的一个要求，具体如下[⊖]：

```
9.9.1 Switching to Protected Mode

6. If paging is enabled, the code for the MOV CR0 instruction
   and the JMP or CALL instruction must come from a page that
   is identity mapped (that is, the linear address before the
   jump is the same as the physical address after paging and
   protected mode is enabled). The target instruction for the
   JMP or CALL instruction does not need to be identity mapped.
```

准确的原因需要问Intel CPU的设计者了，但是原因之一是：在CPU设置了寄存器CR0开启分页机制后，显然所有地址都应该使用虚拟地址，而不再使用物理地址，因此内核将使

⊖ 来源：Intel 64 and IA-32 Architectures Software Developer's Manual，Volume 3A：System Programming Guide, Part 1. January 2011。

用一条长跳转指令，使 EIP 重新装载下一条指令的虚拟地址。但是，在寻址这条长跳转指令本身时，依然使用的是物理地址。显然，要确保经过页面映射后，这条指令依然可以正确映射到其所在的物理内存，因此页目录中映射用户空间的最初部分，即没有 3GB 偏移的部分，也映射到内核占据的物理内存。也就是说，物理地址经过页面映射，依然可以映射到正确的物理地址。这就是 Intel 手册中表达的所谓的恒等映射（identity map）。

曾经一段时间，这种机制工作得很好，也没有出现过什么问题。但是后来，在某些 32 位的 x86 处理器上将 swapper_pg_dir 作为页表激活其他 CPU（secondary CPU）时出现了一些 bug。引起这个 bug 的原因就是恒等映射。

为了解决这个问题，内核引入了变量 initial_page_table，其与 swapper_pg_dir 一样，也定义在内核的 BSS 段：

```
linux-3.7.4/arch/x86/kernel/head_32.S:

/*
 * BSS section
 */
...
ENTRY(initial_page_table)
    .fill 1024,4,0
...
ENTRY(swapper_pg_dir)
    .fill 1024,4,0
```

按照 x86 架构的要求，在 initial_page_table 中进行了恒等映射。但是 initial_page_table 只是在最初引导时使用，一旦引导完成，内核将 initial_page_table 处的内核页表复制到 swapper_pg_dir，但是只复制非恒等映射部分，然后将 swapper_pg_dir 装载到寄存器 CR3。也就是说，内核使用位置 swapper_pg_dir 处的页目录作为最终的页目录，代码如下：

```
linux-3.7.4/arch/x86/kernel/setup.c:

void __init setup_arch(char **cmdline_p)
{
    ...
    clone_pgd_range(swapper_pg_dir + KERNEL_PGD_BOUNDARY,
            initial_page_table + KERNEL_PGD_BOUNDARY,
            KERNEL_PGD_PTRS);

    load_cr3(swapper_pg_dir);
    ...
}
```

后续激活其他 CPU 时，内核也使用这个去除了恒等映射的 swapper_pg_dir 作为页目录，代码如下：

```
linux-3.7.4/arch/x86/kernel/smpboot.c:

/*
```

```
 * Activate a secondary processor.
 */
notrace static void __cpuinit start_secondary(void *unused)
{
    ...
    /* switch away from the initial page table */
    load_cr3(swapper_pg_dir);
    ...
}
```

除了要保存页目录外，内核中也要分配存储页表的地方。内核会将页表存储在 BSS 后面的 brk 段中从标号 __brk_base 开始的地方。__brk_base 在内核链接脚本 vmlinux.lds.S 中定义，代码如下：

```
linux-3.7.4/arch/x86/kernel/vmlinux.lds.S:

SECTIONS
{
    ...
    . = ALIGN(PAGE_SIZE);
    .brk : AT(ADDR(.brk) - LOAD_OFFSET) {
        __brk_base = .;
        . += 64 * 1024;      /* 64k alignment slop space */
        *(.brk_reservation) /* areas brk users have reserved */
        __brk_limit = .;
    }
    ...
}
```

内核中的 brk 概念与普通程序中的 brk 的概念基本相同，都表示动态内存分配的内存区域。

（2）建立页目录和页表

在创建页目录和页表时，第一步需要找到页目录和页表所在的位置，然后按照 IA32 架构的页目录项和页表项的格式约定，逐项填充各个表项。IA32 架构的页目录项和页表项的格式如图 5-17 所示。

Page-Directory Entry(4KB page)

Address of page table	ignored	PS	Ign	A	PCD	PWT	U/S	R/W	P

Page-Table Entry(4KB page)

Address of 4KB page frame	ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	P

图 5-17　页目录项和页表项的格式

内核初始化时创建页目录项和页表项的代码片段如下：

```
linux-3.7.4/arch/x86/kernel/head_32.S:
```

```
01 page_pde_offset = (__PAGE_OFFSET >> 20);
02
03      movl $pa(__brk_base), %edi
04      movl $pa(initial_page_table), %edx
05      movl $PTE_IDENT_ATTR, %eax
06 10:
07      leal PDE_IDENT_ATTR(%edi),%ecx      /* Create PDE entry */
08      movl %ecx,(%edx)                    /* Store identity PDE entry */
09      movl %ecx,page_pde_offset(%edx) /* Store kernel PDE entry */
10      addl $4,%edx
11      movl $1024, %ecx
12 11:
13      stosl
14      addl $0x1000,%eax
15      loop 11b
16      ...
17      movl $pa(_end) + MAPPING_BEYOND_END + PTE_IDENT_ATTR, %ebp
18      cmpl %ebp,%eax
19      jb 10b
```

上述代码中包含了一个二重循环：代码第6~19行是第一层循环，这层循环的目的是填充页目录项，直到建立的页表映射的地址可以覆盖 _end + MAPPING_BEYOND_END。其中符号 _end 在链接脚本 vmlinux.lds.S 中定义，标识内核映像的末尾。这里多映射了 MAPPING_BEYOND_END 目的是为后面第二阶段要建立完整的页表准备空间。代码第12~15行是第二层循环，这层循环每次循环1024次，目的是填充一个页表中的1024个页表项。

填充页目录项

先来看创建页目录项的第一层循环。第4行代码将页目录所在的位置 initial_page_table 存入寄存器 edx。第3行和第7行代码共同创建了第一个页目录项的内容，将其保存到寄存器 ecx。根据第3行代码可见，第一个页表位于符号 __brk_base 处，这个符号也在链接脚本 vmlinux.lds.S 中定义，基本上相当于普通进程的 Program break 处，也就是堆开始的地方。

在确定了页目录项所在的位置，并且也准备好了页目录项的内容后，第8行代码将准备好的页目录项的内容填充到页目录项所在的位置。

在建立初始引导使用的页目录表 initial_page_table 时，除了需要将页目录中映射内核空间的部分映射到内核占据的物理内存外，还要把页目中映射用户空间的最初部分也映射到内核占据的物理内存，也就是前面谈到的恒等映射，这里第9行代码就是做这件事的。

填充页表项

第二层循环完成页表项的填充。先来看第13行处的汇编指令 stosl，该指令将寄存器 eax 中的值存储到寄存器 edi 指示的内存处，然后将寄存器 edi 中的值增加4字节。显然，寄存器 edi 中保存的是页表项所在的位置，eax 中保存的是页表项的内容。

根据第3行代码可见，寄存器 edi 的初值被设置为 __brk_base，也就是说，第1个页表在符号 __brk_base 处。寄存器 eax 的初值在第5行代码中设置为 PTE_IDENT_ATTR：

linux-3.7.4/arch/x86/include/asm/pgtable_types.h:

```
#define PTE_IDENT_ATTR   0x003        /* PRESENT+RW */
```

因为 PTE_IDENT_ATTR 的高 20 位为 0，所以寄存器 eax 的高 20 位也为 0。也就是说，第一个页表项映射的内存页面是从内存 0 开始的一个页面。

因此，第二层循环从 __brk_base 处填充页表项，第一个页表项覆盖了从内存 0 开始的一个页面，以后每次循环后将 eax 指向的物理地址增加 4KB，见第 14 行代码，即指向下一个物理内存页面，依次类推。第 11 行代码设置循环的次数为 1024，所以第二层循环循环 1024 次，将第一个页表全部填充。最终，第一个页表映射了从 0 开始的 4MB 内存空间。

第二层循环结束后，代码将再次进入第一层循环，重新开始新一轮大循环。我们看一下新一轮循环页目录项和页表分别所在的位置。第 10 行代码将寄存器 edx 增加了 4 字节，即指向下一个页目录项。而在第二层循环结束后，寄存器 edi 恰好指向第一个页表后的末尾，这里也就是即将开始的第二个页表的起始位置。然后，内核开启填充第二个页目录项，然后进行第二个页表的过程……以此类推，当建立的页表已经可以映射到物理地址 _end + MAPPING_BEYOND_END 时，即完成了初始页表的建立。

最终，内核建立的页目录以及页表如图 5-18 所示。

（3）启动分页机制

页目录和页表准备好后，内核设置寄存器 CR3 指向页目录，设置寄存器 CR0 中的 PG 位，开启页式映射，代码片段如下：

linux-3.7.4/arch/x86/kernel/head_32.S:

```
movl $pa(initial_page_table), %eax
movl %eax,%cr3       /* set the page table pointer.. */
movl %cr0,%eax
orl  $X86_CR0_PG,%eax
movl %eax,%cr0       /* ..and set paging (PG) bit */
ljmp $__BOOT_CS,$1f /* Clear prefetch and normalize %eip */
```

5.3.2 初始化进程 0

POSIX 标准规定，符合 POSIX 标准的操作系统采用复制的方式创建进程，但是内核总得想办法创建第一个原始的进程，否则其他进程复制谁呢？因此，内核静态的创建了一个原始进程，因为这个进程是内核的第一个进程，Linux 为其分配的进程号为 0，所以也被称为进程 0。进程 0 不仅作为一个模板，在没有其他就绪任务时进程 0 将投入运行，所以其又称为 idle 进程。下面我们就看看内核是如何为进程 0 分配任务结构和内核栈这两个关键数据结构的。

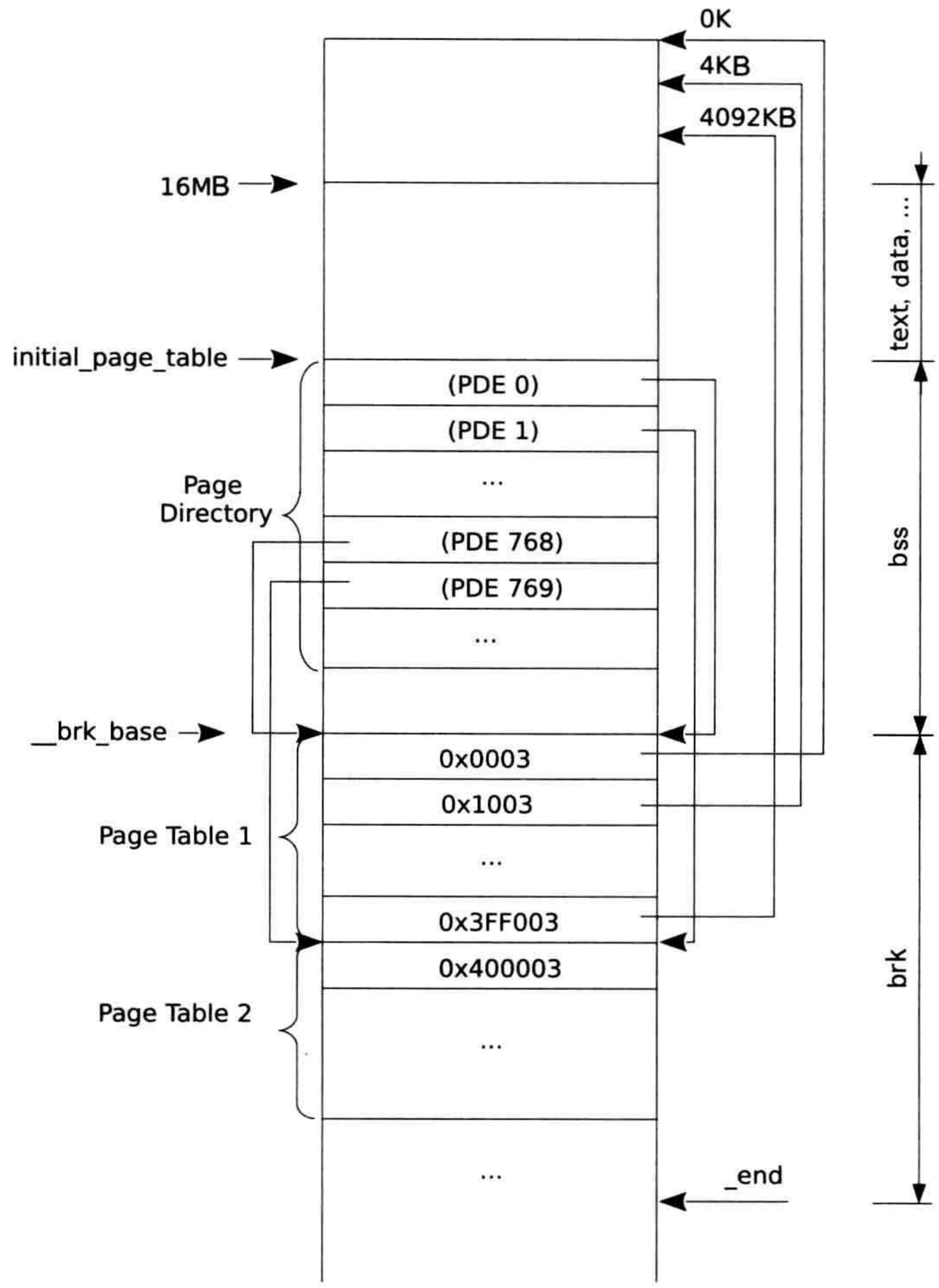


图 5-18　内核初始页表示意图

1. 创建任务结构

进程 0 的任务结构的定义如下：

linux-3.7.4/init/init_task.c:

```
struct task_struct init_task = INIT_TASK(init_task);
```

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

```
#define INIT_TASK(tsk)  \
{                                       \
    .state      = 0,                    \
    .stack      = &init_thread_info,            \
    .usage      = ATOMIC_INIT(2),               \
    ...
}
```

其中变量 init_task 所在的位置（在内核的数据段中）就是进程 0 的任务结构。

当前进程的任务结构是一个频繁使用的变量，为了方便获取它，内核中专门定义了一个宏 current。这个获取方法几经修改，现在的方式是定义了一个变量 current_task 指向当前进程的任务结构。在内核初始化时，内核将这个变量设置为指向 init_task，换句话说，当前进程是进程 0，代码如下：

```
linux-3.7.4/arch/x86/kernel/cpu/common.c

DEFINE_PER_CPU(struct task_struct *, current_task) = &init_task;
```

读者不必关心所谓的 PER_CPU，这是内核为了优化而定义的。为了在 SMP 情况下减少锁的使用，内核中为每颗 CPU 都定义了一个 current_task。

宏 current 就是通过读取 current_task 来获取当前进程的任务结构，定义如下：

```
linux-3.7.4/arch/x86/include/asm/current.h:

#define current get_current()

static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}
```

接下来在内核创建第一个真正意义上的进程（进程 1）时，内核将从 current 指向的进程进行复制，而此时这个 current 恰恰指向进程 0 的任务结构。

2. 进程 0 的内核栈

进程 0 不会切换到用户空间，所以无需用户空间的栈，只需为其安排好内核空间的栈即可。进程内核栈的数据结构抽象如下：

```
linux-3.7.4/include/linux/sched.h:

union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

这个抽象中的数组 stack 就是内核栈，对于 IA32，宏 THREAD_SIZE 定义为 8KB，可见内核为进程内核栈分配的大小为两个页面。那么为什么进程的内核栈与另外一个结构体 thread_info 定义在一起呢？我们后面再讨论这个问题，下面先来具体看一下进程 0 的内核栈：

```
linux-3.7.4/init/init_task.c:

union thread_union init_thread_union __init_task_data =
    { INIT_THREAD_INFO(init_task) };
```

其中，变量 init_thread_union 就是进程 0 的内核栈所在的位置，这个变量也是在内核的数据段中，当然其栈底是在 init_thread_union + THREAD_SIZE 处了，如图 5-19 所示。

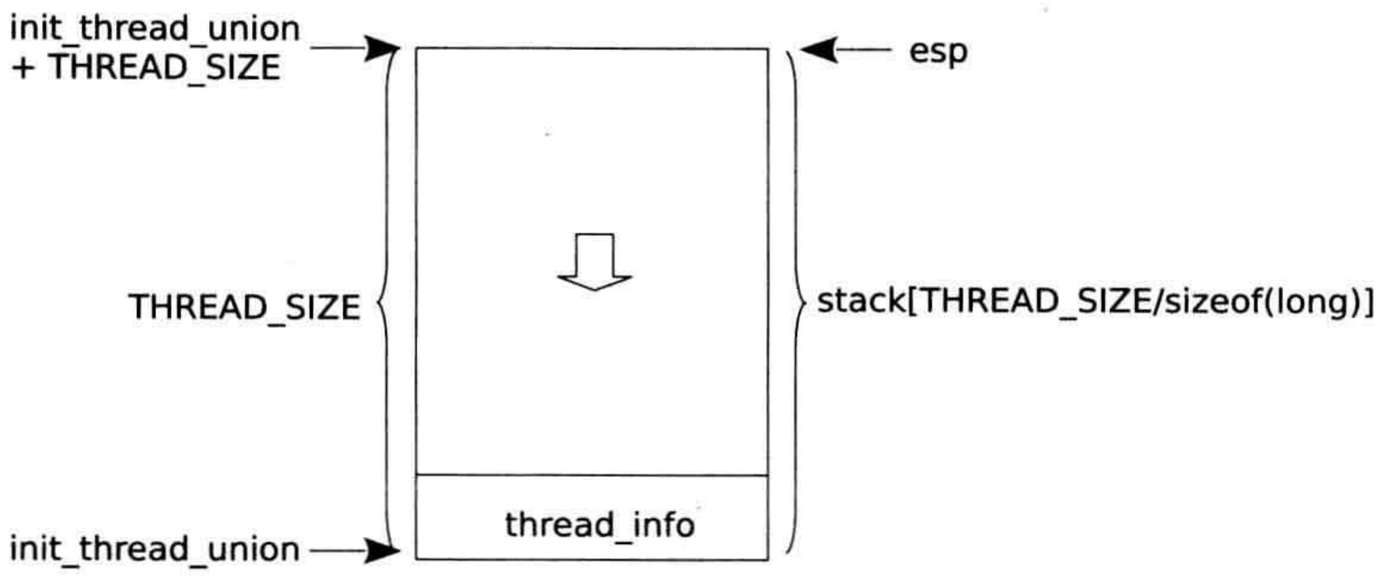


图 5-19 进程 0 的内核栈

在内核初始化时，设定了栈指针 esp 指向 init_thread_union + THREAD_SIZE，代码如下：

```
linux-3.7.4/arch/x86/kernel/head_32.S:

ENTRY(startup_32)
    movl pa(stack_start),%ecx
    ...
    leal -__PAGE_OFFSET(%ecx),%esp
    ...
ENTRY(stack_start)
    .long init_thread_union+THREAD_SIZE
```

因为此时尚未开启分页机制，而符号 stack_start 以及 init_thread_union 均使用的是加了偏移（0xc0000000）的虚拟地址，所以这里都要去掉这个偏移。而在开启页式映射后，把这个偏移又加了回来，如下面代码中使用黑体标识的部分：

```
linux-3.7.4/arch/x86/kernel/head_32.S:

ENTRY(startup_32)
    ...
    movl %cr0,%eax
    orl  $X86_CR0_PG,%eax
    movl %eax,%cr0       /* ..and set paging (PG) bit */
    ljmp $__BOOT_CS,$1f /* Clear prefetch and normalize %eip */
1:
    /* Shift the stack pointer to a virtual address */
    addl $__PAGE_OFFSET, %esp
```

最后，为了在进程切换时可以找到进程 0 的内核栈，还要将其保存在进程 0 的任务结构的结构体 thread_struct 的对象 thread 中，代码如下：

```
linux-3.7.4/include/linux/init_task.h:

#define INIT_TASK(tsk)                \
{                                     \
    ...
    .thread     = INIT_THREAD,    \
```

```
    ...
}
```

linux-3.7.4/arch/x86/include/asm/processor.h:

```
#define INIT_THREAD  {                                          \
    .sp0            = sizeof(init_stack) + (long)&init_stack, \
    ...
}
```

linux-3.7.4/arch/x86/include/asm/thread_info.h:

```
#define init_stack      (init_thread_union.stack)
```

结构体 thread_struct 中的 sp0 就是记录进程内核栈的栈指针。

3. 宏 current 与进程内核栈

这一小节我们来回答为什么进程的内核栈与另外一个结构体 thread_info 定义在一起的问题。

在 2.4 版本以前，内核直接将任务结构嵌入在堆栈的最下方。但是鉴于任务结构也占据不小的空间，而且要把任务结构放在栈底，还需要把任务结构复制到栈中。因此，在 2.6 版本时，内核设计了结构体 thread_info，取而代之的是 thread_info 放在了栈底。通过对寄存器 esp 进行对齐运算，即可方便地找到当前进程的 thread_info：

linux-3.7.4/arch/x86/include/asm/thread_info.h:

```
register unsigned long current_stack_pointer asm("esp") __used;

static inline struct thread_info *current_thread_info(void)
{
    return (struct thread_info *)
        (current_stack_pointer & ~(THREAD_SIZE - 1));
}
```

而 thread_info 中有一个指针指向进程的任务结构，因此获取当前进程的任务结构的方法如下：

linux-3.7.4/include/asm-generic/current.h:

```
#define get_current() (current_thread_info()->task)
#define current get_current()
```

但是，内核开发者还是认为计算 thread_info 位置时间过长，于是采用了以空间换时间的办法，从 2.6.22 版本开始，内核在内存中定义了一个变量 current_task 记录当前进程的任务结构。内核不再通过计算，而是直接通过一条访存指令来设置或者读取当前进程的任务结构。

我们在前面看到，在内核初始化时，current_task 指向进程 0 的任务结构 init_task。以后每次切换进程时，调度函数设置 current_task 指向下一个投入运行的进程的任务结构：

linux-3.7.4/arch/x86/kernel/process_32.c:

```
__notrace_funcgraph struct task_struct *
__switch_to(struct task_struct *prev_p, struct task_struct
            *next_p)
{
    ...
    this_cpu_write(current_task, next_p);
    ...
}
```

5.3.3 创建进程 1

在内核初始化的最后，将调用 kernel_thread 创建进程 1，代码如下：

```
linux-3.7.4/init/main.c:

static noinline void __init_refok rest_init(void)
{
    ...
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    ...
}

linux-3.7.4/kernel/fork.c

pid_t kernel_thread(int (*fn)(void *), void *arg, ...)
{
    return do_fork(flags|CLONE_VM|CLONE_UNTRACED,
      (unsigned long)fn, NULL, (unsigned long)arg, NULL, NULL);
}
```

根据 kernel_thread 代码可见，进程 1 是通过复制进程 0 而来的。在复制了进程后，将执行 kernel_init，相关代码如下：

```
linux-3.7.4/init/main.c:

static int __ref kernel_init(void *unused)
{
    ...
        if (!run_init_process(ramdisk_execute_command))
    ...
}

static int run_init_process(const char *init_filename)
{
    argv_init[0] = init_filename;
    return kernel_execve(init_filename, argv_init, envp_init);
}

linux-3.7.4/fs/exec.c:

int kernel_execve(const char *filename, ...)
{
    ...
    ret = do_execve(filename, ...);
```

```
    ...
}
```

根据代码可见，我们已经看清楚了，第一个进程的创建过程与我们在用户空间创建一个进程并无本质区别，就是我们惯用的套路：fork+exec。

创建进程 1 后，内核调用函数 sechedule 让进程 1 投入运行。在讨论进程 1 的投入运行前，我们先来了解一下内核的基本调度原理，如图 5-20 所示。

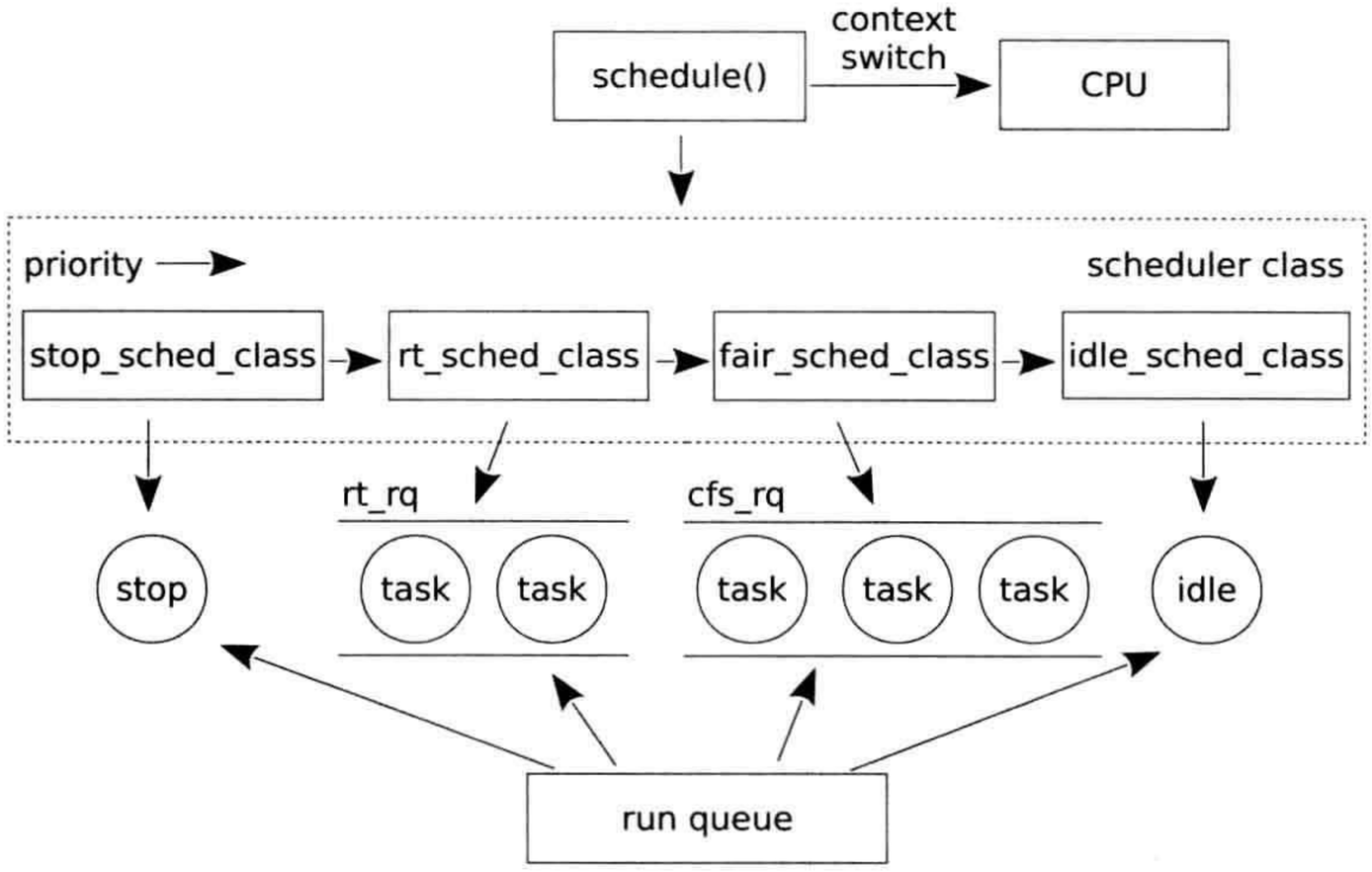


图 5-20 内核调度机制示意图

内核采用模块化的方法，将任务分成四类，优先级从高到低分别是停止类（stop_sched_class）、实时类（rt_sched_class）、公平类（fair_sched_class）和空闲类（idle_sched_class）。从名字我们就可以判断出了这几个类中归属的任务类型了。实时类中记录的是实时任务，一般的任务都归类在公平类中，而停止类和空闲类中记录的是两个特殊的任务。

在没有其他任务就绪时，CPU 将运行空闲类中的任务，该任务将 CPU 置于停机状态，直到有中断将其唤醒。而停止类中的任务是用于负载均衡或者进行 CPU 热插拔时使用的任务，顾名思义，其目的是为了停止正在运行的 CPU，以进行任务迁移或者插拔 CPU。每个 CPU 分别只有一个停止任务和空闲任务。

实时类和公平类分别有一个就绪队列 rt_rq 和 cfs_rq，维护着可以投入运行的任务。每个就绪队列有自己的排队算法，比如公平类采用红黑树对就绪的任务进行排队。

这几个类组成了一个链表，其中最高优先级的停止类作为表头。每个 CPU 有一个就绪队列（run queue），通过该队列，可以访问实时队列、公平队列以及停止任务和空闲任务。

每当调度发生时，调度函数 schedule 调用函数 pick_next_task 按照优先级依次遍历各个类，找出下一个投入运行的任务，代码如下：

linux-3.7.4/kernel/sched/core.c:

```
static inline struct task_struct *pick_next_task(struct rq *rq)
{
    const struct sched_class *class;
    struct task_struct *p;
    ...
    if (likely(rq->nr_running == rq->cfs.h_nr_running)) {
        p = fair_sched_class.pick_next_task(rq);
        if (likely(p))
            return p;
    }

    for_each_class(class) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
    }
    ...
}
```

先看函数 pick_next_task 中后面的 for 循环，显然这是在遍历调度类。pick_next_task 从优先级最高的停止类开始查找，每个类提供了各自的函数 pick_next_task，从就绪队列中选择需要投入运行的任务。

除非用在特定的领域，否则大部分任务应该属于公平类，所以内核开发人员对调度算法进行了一个小小的优化：如果目前系统就绪的任务都属于公平类，则直接从公平类中挑选下一个任务。这就是 for 循环前面的代码片段的作用。

那么进程 0 和进程 1 分别都是属于哪个调度类呢？看下面的代码：

```
linux-3.7.4/kernel/sched/core.c:

void __init sched_init(void)
{
    ...
    current->sched_class = &fair_sched_class;
    ...
}
```

在内核初始化时，在调度相关的初始化函数 sched_init 中，进程 0 的调度类被设置为公平类，因此，在从进程 0 复制后，进程 1 也是公平类。而在复制完成进程 1 后，内核将进程 0 的调度类设置为空闲类，代码如下：

```
linux-3.7.4/init/main.c:

static noinline void __init_refok rest_init(void)
{
    ...
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    ...
    init_idle_bootup_task(current);
    ...
}
```

```
linux-3.7.4/kernel/sched/core.c:

void __cpuinit init_idle_bootup_task(struct task_struct *idle)
{
    idle->sched_class = &idle_sched_class;
}
```

在调用 init_idle_bootup_task 设置了进程 0 的调度类后，内核调用函数 schedule_preempt_disabled 进行调度，代码如下：

```
linux-3.7.4/init/main.c:

static noinline void __init_refok rest_init(void)
{
    ...
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    ...
    init_idle_bootup_task(current);
    schedule_preempt_disabled();
    /* Call into cpu_idle with preempt disabled */
    cpu_idle();
}
```

作为公平类中的进程 1 显然要排在属于空闲类的进程 0 的前面，因此，在这次调度后，进程 1 将被调度函数选中，作为下一个投入运行的任务。而当系统没有其他就绪任务时，将返回到函数 schedule_preempt_disabled 中，继续执行进入函数 cpu_idle。cpu_idle 就是一个无限的循环，循环主体就是 CPU 停机，等待下一次被唤醒执行任务。可见，进程 0 的主体最后就退化为一个无限的 while 循环。

5.4 进程加载

根据 POSIX 标准的规定，操作系统创建一个新进程的方式是进程调用操作系统的 fork 服务，复制当前进程作为一个新的子进程，然后子进程使用操作系统的服务 exec 运行新的程序。前面，我们看到内核已经静态地创建了一个原始进程，进程 1 复制这个原始进程，然后加载了用户空间的可执行文件。这一节，我们就来探讨用户进程的加载过程，大致上整个加载过程包括如下几个步骤：

1）内核从磁盘加载可执行程序，建立进程地址空间；

2）如果可执行程序是动态链接的，那么加载动态链接器，并将控制权转交到动态链接器；

3）动态链接器重定位自身；

4）动态链接器加载动态库到进程地址空间；

5）动态链接器重定位动态库、可执行程序，然后跳转到可执行程序的入口处继续执行。

在本节中，我们使用下面的例子探讨用户进程的加载。

```
foo2.c:
```

```
int foo2 = 20;

void foo2_func() {}

foo1.c:

extern int foo2;
int foo1 = 10;
int dummy = 20;

void foo1_func()
{
    int a = foo2;
    int b = dummy;
    int c = foo1;

    foo2_func();
}

hello.c:

#include <stdlib.h>

extern int foo1;
int a[2048];
int dummy = 10;

void main()
{
    char *m = malloc(1024);
    foo1 = 5;

    foo1_func();
    while (1) sleep(1000);
}
```

我们分别将 foo1.c 和 foo2.c 编译为动态库 libf1.so 和 libf2.so，将 hello.c 编译为一个可执行程序，命令如下：

```
root@baisheng:~/demo# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
root@baisheng:~/demo# gcc -shared -fPIC -o libf2.so foo2.c
root@baisheng:~/demo# gcc -shared -fPIC -o libf1.so foo1.c -L. -lf2
root@baisheng:~/demo# gcc -o hello hello.c -L. -lf1
```

因为 hello 要链接当前目录下的动态库 libf1.so 和 libf2.so，所以这里将当前目录添加到了环境变量 LD_LIBRARY_PATH 中，告诉链接器寻找动态库时，也包括当前工作目录。当然读者也可将这个定义添加到文件 .bashrc 中，每次登录 shell 时将自动定义这个变量，避免每次都需要手工进行定义，实现代码如下：

```
/root/.bashrc:

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

5.4.1 加载可执行程序

一个进程的所有指令和数据并不一定全部要用到，比如某些处理错误的代码。某些错误可能根本不会发生，如果也将这些错误代码加载进内存，就是白白占据内存资源。而且对于某些特别大的程序，如果启动时全部加载进内存，也会使启动时间延长，让用户难以忍受。因此，内核初始加载可执行程序（包括动态库）时，并不将指令和数据真正的加载进内存，而仅仅将指令和数据的“地址”加载进内存，通常我们也将这个过程形象地称为映射。

对于一个程序来说，虽然其可以寻址的空间是整个地址空间，但是这只是个范围而已，就比如某个楼层的房间编号可能是 4 位的，但是并不意味着这个楼层 0000~9999 号房间都可用。对于某个进程而言，一般也仅仅使用了地址空间的一部分。那么一个进程如何知道自己使用了哪些虚拟地址呢？这个问题就转化为是谁为进程分配的运行时地址呢？没错，是链接器分配的，那么当然从 ELF 程序中获取了。所以内核首先将磁盘上 ELF 文件的地址映射进来。

除了代码段和数据段外，进程运行时还需要创建保存局部变量的栈段（Stack Segment）以及动态分配的内存的堆段（Heap Segment），这些段不对应任何具体的文件，所以也被称为匿名映射段（anonymous map）。对于一个动态链接的程序，还会依赖其他动态库，在进程空间中也需要为这些动态库预留空间。

通过上述的讨论可见，进程的地址空间并不是铁板一块，而是根据不同的功能、权限划分为不同的段。某些地址根本没有对应任何有意义的指令或者数据，所以从程序实现的角度看，内核并没有设计一个数据结构来代表整个地址空间，而是抽象了一个结构体 vm_area_struct。进程空间中每个段对应一个 vm_area_struct 的对象（或者叫实例），这些对象组成了“有效”的进程地址空间。进程运行时，首先需要将这个有效地址空间建立起来。

内核支持多种不同的文件格式，每种不同格式的加载都实现为一个模块。比如，加载 ELF 格式的模块是 binfmt_elf，加载脚本的模块是 binfmt_script，它们都在内核的 fs 目录下。对于每个要加载的文件，内核都读入其文件头部的一部分信息，然后依次调用这些模块提供的函数 load_binary 根据文件头的信息判断其是否可以加载。前面，initramfs 中的 init 程序是使用 shell 脚本写的，显然，它是由内核中负责加载脚本的模块 binfmt_script 加载。模块 binfmt_script 中的函数指针 load_binary 指向的具体函数是 load_script，代码如下：

```
linux-3.7.4/fs/binfmt_script.c:

static int load_script(struct linux_binprm *bprm, ...)
{
    ...
    if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!') || ...))
        return -ENOEXEC;
    ...
    for (cp = bprm->buf+2; (*cp == ' ') || (*cp == '\t'); cp++);
    ...
```

```
    file = open_exec(interp);
    ...
    bprm->file = file;
    ...
    return search_binary_handler(bprm,regs);
}
```

linux_binprm 是内核设计的一个在加载程序时，临时用来保存一些信息的结构体。其中，buf 中保存的就是内核读入的要加载程序的头部。函数 load_script 首先判断 buf，也就是文件的前两个字符是否是“#!”。这就是脚本必须以“#!”开头的原因。

如果要加载的程序是一个脚本，则 load_script 从字符“#!”后的字符串中解析出解释程序的名字，然后重新组织 bprm，以解释程序为目标再次调用函数 search_binary_handler，开始寻找加载解释程序的加载器。而脚本文件的名字将被当作解释程序的参数压入栈中。

对于 initramfs 中的 init 程序，其是使用 shell 脚本编写的，所以加载 init 的过程转变为加载解释程序“/bin/bash”的过程，而 init 脚本则作为 bash 程序的一个参数。

可见，脚本的加载，归根结底还是 ELF 可执行程序的加载。

ELF 文件“一人分饰二角”，既作为链接过程的输出，也作为装载过程的输入。在第 2 章中，我们从链接的角度讨论了 ELF 文件格式，当时我们看到 ELF 文件是由若干 Section 组成的。而为了配合进程的加载，ELF 文件中又引入了 Segment 的概念，每个 Segment 包含一个或者多个 Section。相应于 Section 有一个 Section Header Table，ELF 文件中也有一个 Program Header Table 描述 Segment 的信息，如图 5-21 所示。

Program Header Table 中有多个不同类型的 Segment，但是如果仔细观察图 5-21，我们会发现，两个类型为 LOAD 的 Segment 基本涵盖了整个 ELF 文件，而一些 Section，如“.comment”、“.symtab”等，包括 Section Header Table，只是链接时需要，加载时并不需要，所以没有包含到任何 Segment 中。基本上，这两个类型为 LOAD 的 Segment，在映射到进程地址空间时，一个映射为代码段，一个映射为数据段：

- 代码段（code segment）具有读和可执行权限，但是除了保存指令的 Section 外，一些仅具有只读属性的 Section，比如记录解释器名字的“.interp”，动态符号表“.dynsym”，以及重定位表“.rel.dyn”、“.rel.plt”，甚至是 ELF Header、Program Header Table，也包含到了这个段中。这些是程序加载和重定位时需要的信息，随着讨论的深入，我们慢慢就会理解它们的作用。
- 数据段（data segment）具有读写权限，除了典型保存数据的 Section 外，一些具有读写权限的 Section，如 GOT 表，也包含到这个段中。

除了这两个 LOAD 类型的 Segment 外，ELF 规范还规定了几个其他的 Segment，它们都是辅助加载的。仔细观察 Program Header Table，我们会发现，其他类型的 Segment 都包括在 LOAD 类型的段中。所以，在加载时，内核只需要加载 LOAD 类型的 Segment。

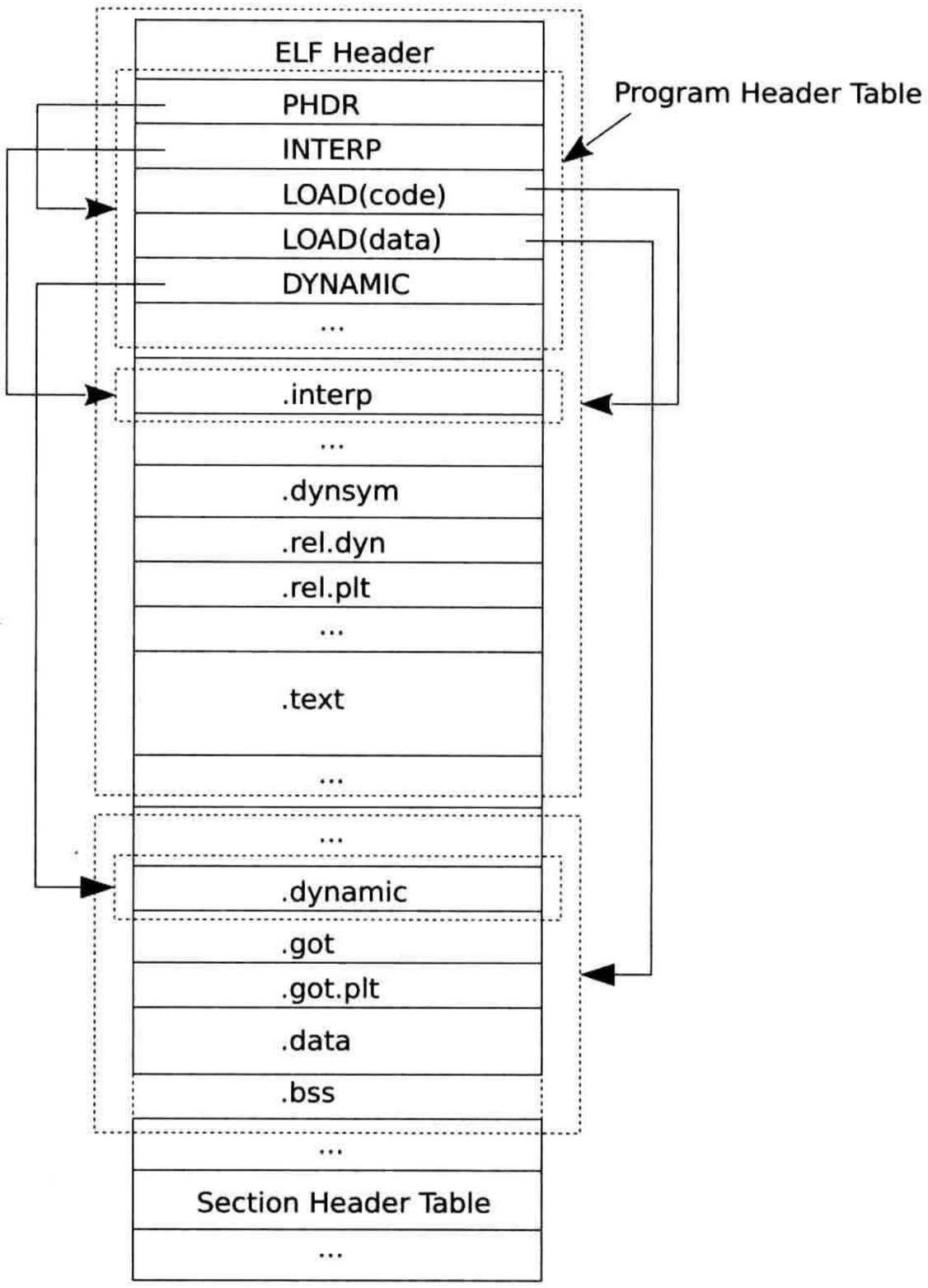


图 5-21 ELF 文件中的 Segment 与 Section

内核中加载 ELF 可执行文件的代码如下：

```
linux-3.7.4/fs/binfmt_elf.c:

static int load_elf_binary(struct linux_binprm *bprm, ...)
{
    ...
    if (memcmp(loc->elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
        goto out;
    ...
    retval = kernel_read(bprm->file, loc->elf_ex.e_phoff,
                (char *)elf_phdata, size);
    ...
    for(i = 0, elf_ppnt = elf_phdata;
        i < loc->elf_ex.e_phnum; i++, elf_ppnt++) {
        ...
        if (elf_ppnt->p_type != PT_LOAD)
            continue;
        ...
        error = elf_map(bprm->file, load_bias + vaddr, elf_ppnt,
```

```
                    elf_prot, elf_flags, 0);
            ...
        }
        ...
    }
```

1）函数 load_elf_binary 首先检测文件头部信息，判断是否是 ELF 类型的文件，包括进一步检测是否是 ELF 的可执行文件或者动态库等。

2）经过一致性检查，如果确认是 ELF 可执行文件，load_elf_binary 读入 Program Header Table。

3）load_elf_binary 遍历 Program Header Table，调用函数 elf_map 映射类型为 PT_LOAD 的段到进程地址空间。elf_map 为每个段创建一个 vm_area_struct 对象，其第二个参数就是段在进程地址空间中映射的地址，这个地址在编译时链接器就已经分配好了。加载偏移 load_bias 是用于动态库的，对于可执行文件来说，load_bias 值是 0。

事实上，除了映射 ELF 文件中的段到进程地址空间外，内核还创建了其他几个进程运行时必不可少的段，包括 BSS、栈和堆三个匿名段，以及为动态库及文件映射预留的内存映射区域，这个区域中一般包含多个段。

（1）栈段

起初，内核将栈安排在用户空间的最顶端，即栈底在 0xc0000000。后来为了安全起见，Linux 使用了 ASLR（Address Space Layout Randomization）技术。ASLR 是一种针对缓冲区溢出的安全保护技术，在进程的地址空间中，堆、栈、内存映射等段不再分配固定的地址，而是在每次进程启动时，在原来的位置上加上一个随机的偏移，增加攻击者确定这些段的位置的难度，从而达到阻止溢出攻击的目的。

创建栈段的 vm_area_struct 对象的代码如下：

```
linux-3.7.4/fs/exec.c:

static int __bprm_mm_init(struct linux_binprm *bprm)
{
    ...
    bprm->vma = vma = kmem_cache_zalloc(vm_area_cachep, ...);
    ...
    vma->vm_end = STACK_TOP_MAX;
    vma->vm_start = vma->vm_end - PAGE_SIZE;
    ...
}
```

函数 __bprm_mm_init 为栈创建了一个 vm_area_struct 对象，栈的初始大小是一个页面（PAGE_SIZE），栈底在 STACK_TOP_MAX。宏 STACK_TOP_MAX 的值如下：

```
linux-3.7.4/arch/x86/include/asm/processor.h:

#define TASK_SIZE       PAGE_OFFSET
#define TASK_SIZE_MAX       TASK_SIZE
```

```
#define STACK_TOP        TASK_SIZE
#define STACK_TOP_MAX        STACK_TOP
```

其中 PAGE_OFFSET 的值就是内核在进程空间中的偏移，即 0xc0000000，也就是用户空间的最顶端。但是接下来在将参数、环境变量所在的页面映射到新进程的栈空间时，内核对栈段的位置进行了随机化处理，代码如下：

```
linux-3.7.4/fs/binfmt_elf.c:

static int load_elf_binary(struct linux_binprm *bprm, ...)
{
    ...
    retval = setup_arg_pages(bprm,
       randomize_stack_top(STACK_TOP), executable_stack);
    ...
}

linux-3.7.4/fs/exec.c:

int setup_arg_pages(struct linux_binprm *bprm, ...)
{
    ...
    stack_top = arch_align_stack(stack_top);
    ...
}
```

x86 架构的函数 arch_align_stack 的代码如下：

```
linux-3.7.4/arch/x86/kernel/process.c:

unsigned long arch_align_stack(unsigned long sp)
{
    if (!(current->personality & ADDR_NO_RANDOMIZE) &&
            randomize_va_space)
        sp -= get_random_int() % 8192;
    return sp & ~0xf;
}
```

根据其中使用黑体标识的部分可见，栈段的地址被进行了随机处理。另外，注意 if 条件中的变量 randomize_va_space，用户可以通过 proc 文件系统中的接口改变这个变量，从而可以动态控制内核的这个特性。

在程序运行时，当进行压栈操作时，如果栈空间不足，将引起缺页中断。缺页中断处理函数调用函数 expand_stack 扩展栈段，代码如下：

```
linux-3.7.4/arch/x86/mm/fault.c:

static void __kprobes __do_page_fault(struct pt_regs *regs, ...)
{
    ...
    if (unlikely(expand_stack(vma, address))) {
    ...
}
```

（2）BSS 段

BSS 段保存的是未初始化的数据，所以 BSS 段并不需要从文件中读取数据，BSS 也并不需要映射到文件，故 BBS 段也是一个匿名映射段。但是注意一点，并不是每个进程都需要创建 BSS 段。如果程序中根本就没有未初始化数据，那么自然就不需要创建 BSS 段。或者程序中未初始化数据占据的空间被数据段的对齐部分覆盖，也不需要创建数据段。假设可执行文件中数据段的结束地址为：

```
0x804a028
```

按照数据段的页对齐要求，在进程地址空间中对齐后，数据段的结束地址为：

```
0x804b000
```

如果从 0x804a028 到 0x804b000 之间的这段空间已经覆盖了全部的未初始化数据，那么就不必再创建 BSS 段了。

函数 load_elf_binary 中创建 BSS 段的相关代码如下：

```
linux-3.7.4/fs/binfmt_elf.c:

01 static int load_elf_binary(...)
02 {
03      ...
04      struct elf_phdr *elf_ppnt, *elf_phdata;
05      ...
06      elf_bss = 0;
07      elf_brk = 0;
08      ...
09      for(i = 0, elf_ppnt = elf_phdata;
10          i < loc->elf_ex.e_phnum; i++, elf_ppnt++) {
11          ...
12          k = elf_ppnt->p_vaddr + elf_ppnt->p_filesz;
13
14          if (k > elf_bss)
15              elf_bss = k;
16          ...
17          k = elf_ppnt->p_vaddr + elf_ppnt->p_memsz;
18          if (k > elf_brk)
19              elf_brk = k;
20      }
21      ...
22      retval = set_brk(elf_bss, elf_brk);
23      ...
24 }
```

代码第 9~20 行遍历 ELF 文件的 Program Header Table，其中 elf_phdr 是指向表 Program Header Table 中的 Program Header 的指针，elf_ppnt->p_vaddr 是段在进程地址空间中的起始地址，elf_ppnt->p_filesz 是段在 ELF 文件中占据的尺寸，elf_ppnt->p_memsz 记录的是段在内存中占据的尺寸。

对于 ELF 可执行程序而言，这个 for 循环将循环两次，第一次映射代码段，第二次

映射数据段。因此，在第二次循环后，第 12 行代码中的变量 k 的值是数据段的起始地址（VirtAddr）与数据段（不包含 BSS）的大小（FileSiz）的和，并在第 15 行代码将这个值记录在变量 elf_bss 中。第 17 行代码中变量 k 的值是数据段的起始地址（VirtAddr）与数据段（包含 BSS）的大小（MemSiz）的和，并在第 19 行记录在变量 elf_brk 中。显然，elf_bss 指向不包含 BSS 数据的数据段的结束位置，而 elf_brk 指向包含 BSS 数据的数据段的结束位置。然后，load_elf_binary 调用函数 set_brk 比较 elf_bss 和 elf_brk。看到 brk 这个词是不是有点似曾相识？没错，brk 就是 program break，即代表程序动态申请地址的上限。那么 BSS 段和 brk 有关系吗？为什么在映射 BSS 段时出现了 brk？当然有，因为 BSS 段的末尾就是 brk 的起始地址。函数 set_brk 的代码如下：

```
linux-3.7.4/fs/binfmt_elf.c:

static int set_brk(unsigned long start, unsigned long end)
{
    start = ELF_PAGEALIGN(start);
    end = ELF_PAGEALIGN(end);
    if (end > start) {
        unsigned long addr;
        addr = vm_brk(start, end - start);
        if (BAD_ADDR(addr))
            return addr;
    }
    current->mm->start_brk = current->mm->brk = end;
    return 0;
}
```

set_brk 对比经过页对齐后的 elf_bss 和 elf_brk。如果对齐后前者不能涵盖后者，则调用函数 vm_brk 创建单独的 BSS 段，为 BSS 段创建一个 vm_struct_area 对象。

结构体 mm_struct 中的 start_brk 用来记录堆段的起始位置，变量 brk 记录堆段的结束位置。根据函数 set_brk 的代码可见，初始化时，堆的起始位置和结束位置都是 BSS 段的结束位置。在程序动态申请内存时，内核再按需扩展堆的大小。

（3）堆段

堆段映射的内存是进程运行时动态分配的，所以在建立进程的地址空间时，只需确定堆段的起始位置即可。根据前面讨论的函数 set_brk，初始时，堆的起始位置和结束位置都指向 BSS 段的结束位置。在进程运行时，根据程序动态申请内存情况动态的调整堆的大小。比如程序调用 C 库的 malloc/free 函数动态分配和释放内存时，事实上就是通过内核的系统调用 brk/sbrk 动态改变堆的大小。

出于安全的原因，堆段也使用了 ASLR 技术，所以这个位置一般并不紧接在 BSS 的后面，而是又加了一个随机的偏移，代码如下：

```
linux-3.7.4/fs/binfmt_elf.c:
static int load_elf_binary(struct linux_binprm *bprm, ...)
{
```

```
    ...
    if((current->flags & PF_RANDOMIZE)&&(randomize_va_space > 1)){
        current->mm->brk = current->mm->start_brk =
            arch_randomize_brk(current->mm);
        ...
    }
    ...
}
```

根据上面的代码可见，如果变量 randomize_va_space 的值大于 1，则调用体系结构相关的函数 arch_randomize_brk 将 start_brk 和 brk 调整到一个随机的值。 IA32 架构中的函数 arch_randomize_brk 实现如下：

```
linux-3.7.4/arch/x86/kernel/process.c:

unsigned long arch_randomize_brk(struct mm_struct *mm)
{
    unsigned long range_end = mm->brk + 0x02000000;
    return randomize_range(mm->brk, range_end, 0) ? : mm->brk;
}
```

内核在 proc 中为用户提供了一个接口，允许用户修改变量 randomize_va_space，从而可以动态控制内核的这个特性。

（4）内存映射区域

进程空间中还专门留有一个区域用于内存映射，比如文件映射、共享内存等，动态库就映射在这个区域。内存映射区域一般包含多个 vm_struct_area 对象。比如一个程序依赖多个动态库，那么就会有多个动态库映射到这里。而且即使是同一个动态库，也存在着如代码段、数据段等多个段。

对于 x86 架构，在 2.4 版本时，内存映射区域的起始地址是固定的，在内核用户空间的 1/3 处，即 0xc0000000/3=0x40000000。从 2.6 版本以后，内核将这个区域安排在了栈段的下方。x86 架构下确定内存映射区域的基址的函数如下：

```
linux-3.7.4/arch/x86/mm/mmap.c:

void arch_pick_mmap_layout(struct mm_struct *mm)
{
    if (mmap_is_legacy()) {
        mm->mmap_base = mmap_legacy_base();
        ...
    } else {
        mm->mmap_base = mmap_base();
        ...
    }
}
```

在函数 arch_pick_mmap_layout 中，if 代码块中对应的就是内核传统的（2.4 版本）确定内存映射区域起始位置的方法，而 else 代码块对应的则是从 2.6 版本开始使用的方法。根据

代码可见，在 2.6 版本下，确定这个位置的函数是 mmap_base，其代码如下：

linux-3.7.4/arch/x86/mm/mmap.c:

```
01 static unsigned long mmap_base(void)
02 {
03     unsigned long gap = rlimit(RLIMIT_STACK);
04
05     if (gap < MIN_GAP)
06         gap = MIN_GAP;
07     else if (gap > MAX_GAP)
08         gap = MAX_GAP;
09
10    return PAGE_ALIGN(TASK_SIZE - gap - mmap_rnd());
11 }
```

根据第 3 行代码可见，内核取出进程的栈的上限，然后将内存映射区域安排在栈的下方。内核默认进程栈的大小是 8MB，但是每个进程都可以通过系统调用 ulimit 设置进程的各项资源，包括栈的大小。所以在分配内存映射的基址时，内核首先尊重进程的意愿，调用 rlimit 读取了进程设置的栈的上限。但是，内核可不能由着用户的性子来，毕竟资源有限，内核还要判断用户设置的栈空间是否合理，这就是代码第 5~8 行的目的。我们看到，内核要求进程空间中，栈的最小尺寸是 MIN_GAP，而最大尺寸是 MAX_GAP。这两个宏的定义如下：

linux-3.7.4/arch/x86/mm/mmap.c:

```
#define MIN_GAP (128*1024*1024UL + stack_maxrandom_size())
#define MAX_GAP (TASK_SIZE/6*5)
```

可见，内核给栈预留的空间最小是 128MB，最大是 TASK_SIZE/6*5=3GB/6*5=2.5GB。

最后，内核调用函数 mmap_rnd 计算了一个随机的偏移，加在了内存映射的基址上，见第 10 行代码。也就是说，内存映射区域，内核也使用了 ASLR 技术。

综上，进程的地址空间大致如图 5-22 所示。

在图 5-22 中，进程地址空间中有效的部分使用实线标出，虚线部分是尚未映射的部分。因为数据段可能涵盖了 BSS，所以映射 BSS 的 vm_area_struct 对象也使用虚线标出，表示在程序映射时，可能并不会建立 BSS 段。另外，在内存映射区域，图中只示意性地列出了 C 库和动态链接器中的部分段的映射，其他段的映射并没有列出，所以也使用了一个虚线标出的 vm_area_struct 对象代表其他的映射。

最后，我们以可执行程序 hello 为例，具体观察一下进程的地址空间。

首先来看一下 hello 的 Program Header Table：

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

Program Headers:
  Type      Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR      0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
  INTERP    0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
```

```
    [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD    0x000000 0x08048000 0x08048000 0x0079c 0x0079c R E 0x1000
LOAD    0x000f00 0x08049f00 0x08049f00 0x0012c 0x02160 RW 0x1000
DYNAMIC    0x000f0c 0x08049f0c 0x08049f0c 0x000f0 0x000f0 RW  0x4
NOTE        0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
GNU_EH_FRAME 0x0006a8 0x080486a8 0x080486a8 0x00034 0x00034 R 0x4
GNU_STACK  0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
GNU_RELRO  0x000f00 0x08049f00 0x08049f00 0x00100 0x00100 R   0x1
```

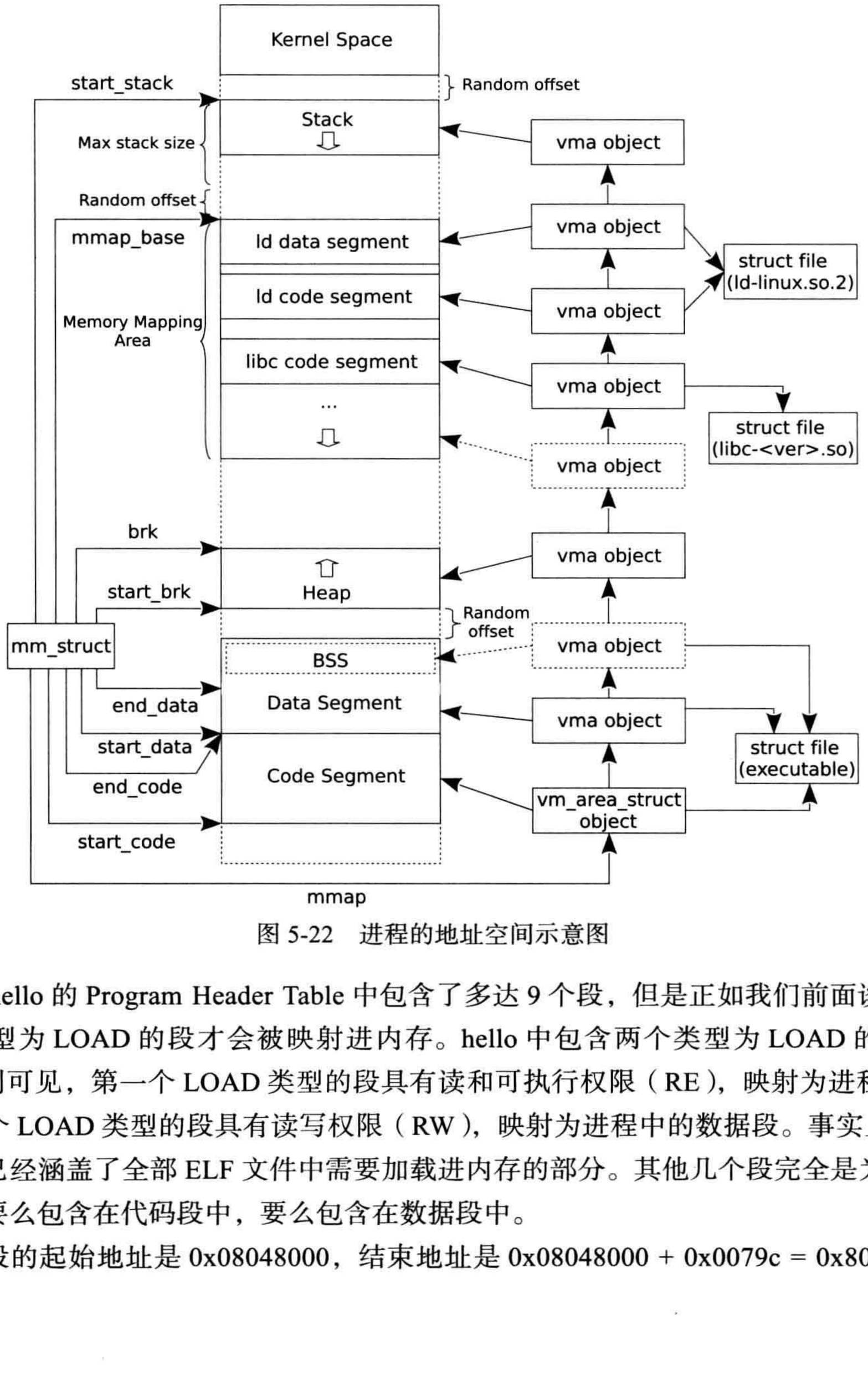


图 5-22 进程的地址空间示意图

虽然 hello 的 Program Header Table 中包含了多达 9 个段，但是正如我们前面谈到的，其中只有类型为 LOAD 的段才会被映射进内存。hello 中包含两个类型为 LOAD 的段，根据“Flg”一列可见，第一个 LOAD 类型的段具有读和可执行权限（RE），映射为进程中的代码段；第二个 LOAD 类型的段具有读写权限（RW），映射为进程中的数据段。事实上，LOAD 类型的段已经涵盖了全部 ELF 文件中需要加载进内存的部分。其他几个段完全是为了辅助加载用的，要么包含在代码段中，要么包含在数据段中。

代码段的起始地址是 0x08048000，结束地址是 0x08048000 + 0x0079c = 0x804879c。根

据列“Align”可见代码段要求 4KB 对齐，起始地址已经是 4KB 对齐的，无须调整，而结束地址则需要从 0x804879c 调整为 0x8049000。

数据段的起始地址是 0x08049f00，结束地址是 0x08049f00 + 0x0012c = 0x0804a02c。根据列“Align”可见数据段也要求 4KB 对齐，所以数据段的起始地址需要调整为 0x08049000，结束地址需要调整为 0x0804b000。hello 的 BSS 段为 8244 字节（0x02160 ~ 0x0012c），显然数据段对齐的那部分已经不能涵盖未初始化数据了，因此需要创建一个单独的 BSS 段。

下面我们将 hello 程序运行起来，结合前面的理论分析，实际观察一下其进程地址空间的映射。

```
root@baisheng:~/demo# ./hello &
[1] 4822
root@baisheng:~/demo# cat /proc/4822/maps
08048000-08049000 r-xp 00000000 08:01 1054223    /root/demo/hello
08049000-0804a000 r--p 00000000 08:01 1054223    /root/demo/hello
0804a000-0804b000 rw-p 00001000 08:01 1054223    /root/demo/hello
0804b000-0804d000 rw-p 00000000 00:00 0
0985e000-0987f000 rw-p 00000000 00:00 0          [heap]
b75a4000-b75a5000 rw-p 00000000 00:00 0
b75a5000-b75a6000 r-xp 00000000 08:01 1054350 /root/demo/libf2.so
b75a6000-b75a7000 r--p 00000000 08:01 1054350 /root/demo/libf2.so
b75a7000-b75a8000 rw-p 00001000 08:01 1054350 /root/demo/libf2.so
b75a8000-b75a9000 rw-p 00000000 00:00 0
b75a9000-b774c000 r-xp 00000000 08:01 523958
      /lib/i386-linux-gnu/libc-2.15.so
b774c000-b774d000 ---p 001a3000 08:01 523958
      /lib/i386-linux-gnu/libc-2.15.so
b774d000-b774f000 r--p 001a3000 08:01 523958
      /lib/i386-linux-gnu/libc-2.15.so
b774f000-b7750000 rw-p 001a5000 08:01 523958
      /lib/i386-linux-gnu/libc-2.15.so
b7750000-b7753000 rw-p 00000000 00:00 0
b7768000-b7769000 r-xp 00000000 08:01 1047105 /root/demo/libf1.so
b7769000-b776a000 r--p 00000000 08:01 1047105 /root/demo/libf1.so
b776a000-b776b000 rw-p 00001000 08:01 1047105 /root/demo/libf1.so
b776b000-b776d000 rw-p 00000000 00:00 0
b776d000-b776e000 r-xp 00000000 00:00 0          [vdso]
b776e000-b778e000 r-xp 00000000 08:01 523936
      /lib/i386-linux-gnu/ld-2.15.so
b778e000-b778f000 r--p 0001f000 08:01 523936
      /lib/i386-linux-gnu/ld-2.15.so
b778f000-b7790000 rw-p 00020000 08:01 523936
      /lib/i386-linux-gnu/ld-2.15.so
bf92a000-bf94b000 rw-p 00000000 00:00 0          [stack]
```

根据输出可见：

1）地址范围 0x08048000 ~ 0x08049000 具有读和可执行权限，显然就是进程的代码段。

2）地址范围 0x0804a000 ~ 0x0804b000 具有读写权限，是进程的数据段。

3）在代码段和数据段之间映射了一个只读的段：0x08049000 ~ 0x0804a000。虽然这

个段是读的，但是广义上其属于数据段，这个只不过是从数据段中划分的一个子段，称为段RELRO，读者可暂不关心，我们在5.4.9节会讨论进程空间映射这个段的缘由。

4）地址范围0x0804b000 ~ 0x0804d000具有读写权限，而且是个匿名映射段，紧接在数据段后面，而且正好占据8KB大小的空间，我想读者已经猜到了，其就是保存程序hello中未初始化的全局数组a[2048]的BSS段。读者可以做个实验，去掉程序hello.c中的这个数组，然后重新编译运行，就可发现hello进程将无需再映射单独的BSS段。

5）地址范围0x0985e000 ~ 0x0987f000具有读写权限，显然也是保存数据的，根据后面的字串“heap”，读者应该可以猜到，这个段是hello进程的堆段。读者也可作个实验，去掉程序hello.c中使用mallo动态分配的1024个字节，然后重新编译运行，就可发现堆段也将不再被映射。前面我们谈到过，堆是程序运行时动态分配的，所以如果程序中尚未在堆中申请变量，内核将不会主动为进程映射堆段，只是首先确定好堆的基址，在需要时按需动态映射。

6）接下来，就是一个大的内存映射区域了：0xb75a4000 ~ 0xb7790000，在这个区域中，映射了C库、动态链接器以及动态库libf1和libf2。对于每个动态库来说，其映射过程与可执行程序并无本质差别，仔细观察，可以发现，每个动态库也有自己的代码段、数据段等，其具体映射过程我们在加载动态库一节再讨论。

7）进程空间中最后映射的一个段：0xbf92a000 ~ 0xbf94b000，具有读写权限，也是保存数据的，根据后面输出字串“stack”可知，这个段就是栈段。

最后，我们留意栈段的起始地址：0xbf94b000。理论上，栈底应该在用户空间的最顶端，即0xc0000000，但是为什么不是这个地址呢？请读者回想一下我们前面谈到的ASLR技术，栈底在0xc0000000的基础上减去了一个随机的偏移。

与此相仿的还有堆段和内存映射部分。读者可以做个实验，多次启动hello程序，你会发现，每次这几个段的起始地址全都不同。每次进程启动时，都会在原来的理论地址上，再加了一个随机的偏移。

内核在proc文件系统中为用户提供了一个接口，允许用户动态控制是否使用ASLR技术。这个接口可以接收3个参数：0代表关闭ASLR技术；1代表内存映射区域、栈和vdso段的起始地址是随机的；2表示堆段起始地址也是随机的。以笔者的机器为例，其默认值为2：

```
root@baisheng:~# cat /proc/sys/kernel/randomize_va_space
2
```

读者可以采用下面的方法，关闭ASLR：

```
root@baisheng:~# echo 0 > /proc/sys/kernel/randomize_va_space
```

然后，即使多次启动同一个程序，但是这几个段的地址也不再随机变动了。

内核还保留了2.4版本的分配内存映射区域的方法，用户也可以通过下面的方法使用传

统的方法：

```
root@baisheng:/proc/sys# echo 1 > /proc/sys/vm/legacy_va_layout
```

使用下面的命令关闭传统的内存映射机制：

```
root@baisheng:/proc/sys# echo 0 > /proc/sys/vm/legacy_va_layout
```

限于篇幅，我们就不再一一列出开启和关闭这些参数后，进程空间的映射情况，读者可自行进行这些非常有趣的实验。

5.4.2 进程的投入运行

丑媳妇总是要见公婆的，进程最终一定是要切换到用户空间的，进程 1 也躲不过去。在内核创建进程 1 时，进程 0 是当前进程，因此，进程 1 要“回到”用户空间，需要经过两个步骤：

1）要将进程 0 赶出 CPU。也就是在内核空间，进程 1 要“恢复”为当前进程，这是进程 1“返回”用户空间的前提条件。

2）进程 1 从内核空间“回到”用户空间。

显然，从进程 0“恢复”到进程 1 需要进程 1 在内核空间的现场；进程 1 从内核空间“回到”用户空间需要进程 1 在用户空间的现场。但是，事实上，不仅是进程 1，对于所有刚刚创建的进程，它们并没有经历过从用户空间切到内核空间，然后在内核空间被其他进程抢占的过程，哪里来的保护现场？所以，就需要操作系统助它们一臂之力，人为地为新创建的进程伪造现场。

这一节，我们首先来看看在这两次转换过程中，保护现场的原理。然后，我们再来讨论内核是如何在原理的指引下伪造这两个现场的。事实上，不仅进程 1 如此，其他进程也如此，这里的讨论适用于所有进程。

1. 用户现场的保护

我们通过讨论一个进程从用户空间切换到内核空间来观察用户现场是如何保护的。

（1）从用户栈切换到内核栈

当一个进程正在用户空间运行时，一旦发生中断，那么进程将从用户空间切换到内核空间运行。进程在内核空间运行时，CPU 各个寄存器同样将被使用，因此，为了在处理完中断后，程序可以在用户空间的中断处得以继续执行，需要在穿越的一刻保护这些寄存器的值，以免被覆盖，即所谓的保护现场。Linux 使用进程的内核栈保存进程的用户现场。因此，在中断时，CPU 做的第一件事就是将栈从用户栈切换到内核栈，如图 5-23 所示。

Intel 从硬件层面设计了 TSS 段（task-state segment）支持任务的管理，其中记录了任务的状态信息。既然是一个段，每个 TSS 也在 GDT 中占据一个表项。如同代码段将段选择子保存在寄存器 CS 中，CPU 要求将 TSS 段选择子保存在专用寄存器 TR（Task Register）中。

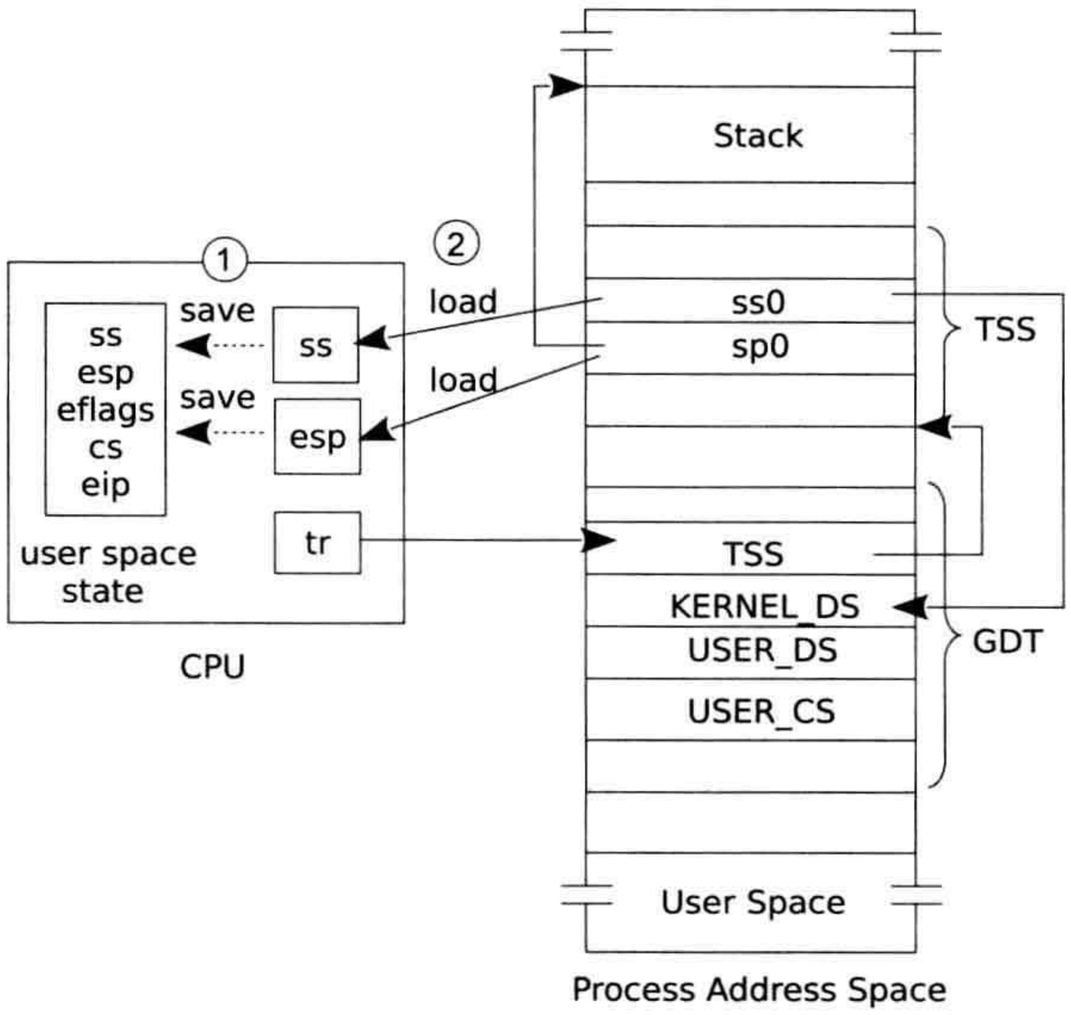


图 5-23　从用户栈切换到内核栈

Intel 建议为每一个进程准备一个独立的 TSS 段，每当任务切换时，更换 CPU 中 TR 寄存器指向当前任务的 TSS 段。天下没有免费的午餐，方便的代价就是效率的低下。而事实上，任务切换时，本不必保存如 TSS 中包含的如此复杂的上下文，而且 TR 寄存器的格式比较特殊，远非直接装入一个地址那么简单，还需要进行一些计算，因此切换 TR 的代价也比较大。因此，Linux 并没有使用 TSS 段保存任务状态信息。

但是当中断发生时，CPU 自动从任务寄存器 TR 中找到 TSS 段，然后从该段中装载 ss 和 esp。“我的地盘听我的”，所以 Linux 还要遵从 Intel 的“霸王”条款，必须得使用 TSS。但是 Linux 处理得比较巧妙，其并没有为每个任务设计一个 TSS 段，而是为每个 CPU 准备一个 TSS 段。内核只是在初始化阶段设置 TR，使之指向一个 TSS 段，从此以后永不改变 TR 的内容。TSS 段的初始内容如下：

linux-3.7.4/arch/x86/include/asm/processor.h:

```
#define INIT_TSS  {                                             \
    .x86_tss = {                                                \
        .sp0          = sizeof(init_stack) + (long)&init_stack, \
        .ss0          = __KERNEL_DS,                            \
        .ss1          = __KERNEL_CS,                            \
        .io_bitmap_base = INVALID_IO_BITMAP_OFFSET,             \
     },                                                         \
    .io_bitmap      = { [0 ... IO_BITMAP_LONGS] = ~0 },         \
}
```

内核初始化时，在函数 cpu_init 中初始化了 TR 寄存器，代码如下：

linux-3.7.4/arch/x86/kernel/cpu/common.c:

```
void __cpuinit cpu_init(void)
{
    ...
    struct tss_struct *t = &per_cpu(init_tss, cpu);
    ...
    set_tss_desc(cpu, t);
    load_TR_desc();
    ...
}
```

其中，函数 set_tss_desc 设置 GDT 中的 TSS 段的描述符，函数 load_TR_desc 设置 TR 寄存器。

虽然 TSS 不需要切换了，但是 TSS 中的 ss 和 sp0 却需要随着任务的切换，走马灯式地更换，记录下一个投入运行的任务的内核栈的 ss 和 sp0。这样，就保证了在中断发生时，CPU 可以正确地从 TSS 中加载当前进程内核栈的 ss 和 esp。在宏 INIT_TSS 中，TSS 段中记录的内核栈的 ss0 被设置为 __KERNEL_DS，所以这里 ss0 永远不需要改变，只需切换 sp0 即可。可见，TSS 是“铁打的营盘，流水的兵”。

但是不知道读者思考过没有，当中断发生时，当 CPU 从 TSS 段中加载 ss0 和 sp0 分别到 ss 和 esp 时，尚未保存用户现场，那么此时保存在 ss 和 esp 中的用户栈的信息岂不是被覆盖了？ Intel 的工程师们当然清楚这一点，事实上，CPU 在加载内核栈信息前，会将寄存器 ss 和 esp 中的值首先临时保存到 CPU 内部，除了保存寄存器 ss 和 esp 的值外，CPU 临时保存的还包括寄存器 eflags、cs、eip 中的值。

经过这一步后，进程已经完成了栈的切换，进程在向内核空间前进的道路上迈出了第一步。

（2）保存用户空间的现场

切换完栈后，CPU 在进程的内核栈中保存了进程在用户空间执行时的现场信息，包括 eflags、cs、eip、ss 和 esp，如图 5-24 所示。

在进程退出内核空间时，中段处理函数最后会调用 x86 的指令 iret 将 CPU 压入的这几个值恢复到对应的寄存器中。

（3）穿越中断门

接下来，进程就将进行最后的穿越了，当然，内核在初始化时就已经为 CPU 初始化了中断相关的部分，代码如下：

```
linux-3.7.4/arch/x86/kernel/head_32.S:

01 ENTRY(startup_32)
02      ...
03      lidt idt_descr
04      ...
05 setup_once:
06      movl $idt_table,%edi
```

```
07      movl $early_idt_handlers,%eax
08      movl $NUM_EXCEPTION_VECTORS,%ecx
09 1:
10      movl %eax,(%edi)
11      movl %eax,4(%edi)
12      movl $(0x8E000000 + __KERNEL_CS),2(%edi)
13      addl $9,%eax
14      addl $8,%edi
15      loop 1b
16
17      movl $256 - NUM_EXCEPTION_VECTORS,%ecx
18      movl $ignore_int,%edx
19      movl $(__KERNEL_CS << 16),%eax
20      movw %dx,%ax            /* selector = 0x0010 = cs */
21      movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */
22 2:
23      movl %eax,(%edi)
24      movl %edx,4(%edi)
25      addl $8,%edi
26      loop 2b
27      ...
28 idt_descr:
29      .word IDT_ENTRIES*8-1           # idt contains 256 entries
30      .long idt_table
```

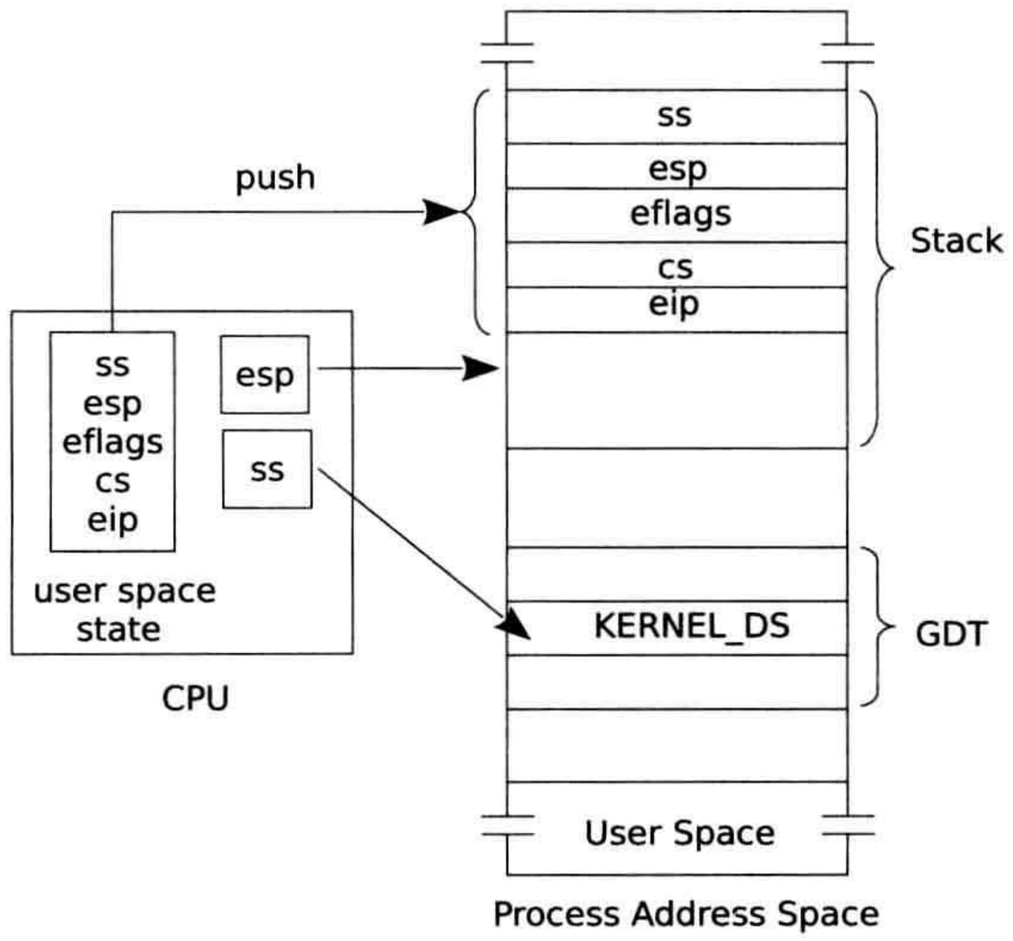


图 5-24 保存用户空间的现场

代码第 6~26 行初始化中断描述符表 idt_table，其包含 256 项，每一项均是一个 64 位的描述符。CPU 运行过程中，可能有多种情况需要中断正在执行的指令，转而先去处理中断。包括外部设备来的信号，或者是执行指令时发生了异常，如发生了除数是 0 的情况。因此，中断描述符表中包括几种不同的描述符，但是大同小异。这些描述符也被称为门描述符（gate descriptor），以中断门为例，其格式如图 5-25 所示。

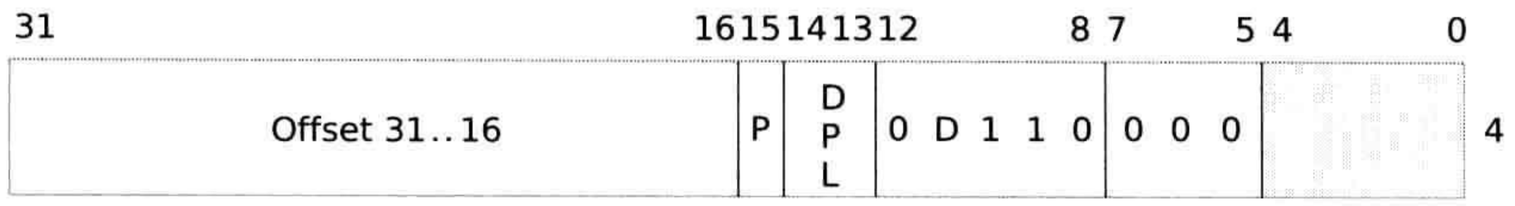


图 5-25 中断门格式

对于图 5-25，重点关注其中两个字段，一个是“Segment Selector”，这个是段选择子，也就是对应这个中断的处理函数所在的段；另外一个是“Offset”，其表示的是中断处理函数在段内的偏移。因为 Linux 使用的是平坦内存模式，段基址为 0，所以实际上这个段内偏移就是中断处理函数的地址。

上面代码中包含两个 loop 循环，填充了 256 项中断描述符。每个门的段选择子都是 __KERNEL_CS，只有中断处理函数不同。前 NUM_EXCEPTION_VECTORS 项对应的中断处理函数是 early_idt_handlers，其余项的中断处理函数是 ignore_int。这两个函数都是内核初始化早期的临时中断处理函数，在内核建立好基本环境后，会使用真正的中断处理函数替换这些临时的，代码如下：

```
linux-3.7.4/arch/x86/kernel/traps.c:

void __init trap_init(void)
{
    ...
    set_intr_gate(X86_TRAP_TS, &invalid_TSS);
    set_intr_gate(X86_TRAP_NP, &segment_not_present);
    ...
}
```

中断描述符表构建完成后，内核还需要将其地址告诉 CPU，CPU 中为此设计了一个专用寄存器 idtr。除了中断描述表的地址外，当然还需要将这个表长度也载入这个寄存器。x86 设计了指令 lidt 来加载 idtr 寄存器，见函数 startup_32 代码中的第 3 行。

了解了中断门的数据结构后，我们就很容易理解在穿越中断门的一刹那，CPU 的所作所为了。CPU 首先将根据寄存器 IDTR，找到中断描述符表。然后以中断向量作为下标，在中断描述符表中找到对应的门，CPU 将其中的段选择子加载到寄存器 cs，将其中的偏移地址加载到寄存器 eip，如图 5-26 所示。

经过这一步后，进程彻底的穿越到了内核空间。

因为 Linux 没有使用 TSS，所以除了 CPU 自动将 cs、eip 等几个寄存器压入栈外，中断处理函数需要将其他的寄存器也压入内核栈，保存进程完整的用户空间的现场。在进程退出内核空间时，中段处理函数负责将其压入栈的这些值再恢复到相应的寄存器中。

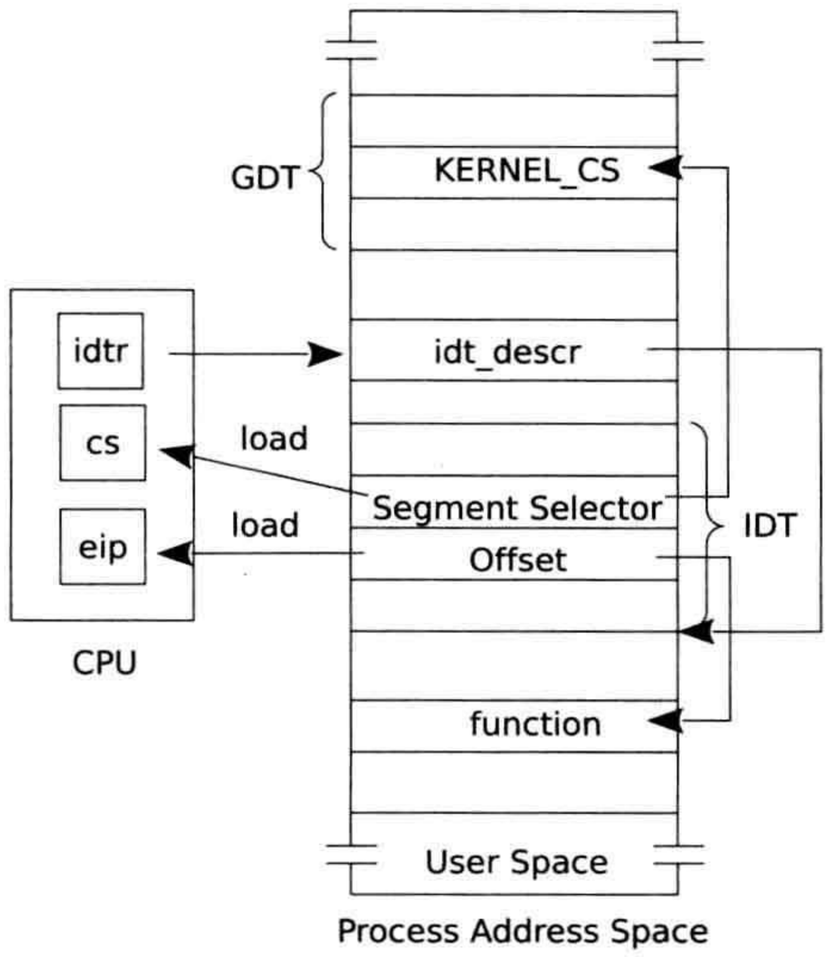


图 5-26 穿越中断门

2. 内核现场的保护

当进程在内核空间运行时，在发生进程切换时，依然需要保护切换走的进程的现场，这是其下次运行的起点。那么进程的内核现场保存在哪里合适呢？前面我们看到进程的用户现场保存在进程的内核栈，那么进程的内核现场当然也可以保存在进程的内核栈。

但是，当调度函数准备切换到下一个进程时，下一个进程的内核栈的栈指针从何而来？在前面讨论进程从用户空间切换到内核空间时，我们看到，CPU 从进程的 TSS 段中获取内核栈的栈指针。那么当在内核空间发生切换时，调度函数如何找到准备切入进程的内核栈的栈指针？

除了进程的内核栈外，进程在内核中始终存在另外一个数据结构——进程的户口，即任务结构。因此，进程的内核栈的栈指针可以保存在进程的任务结构中。在任务结构中，特意抽象了一个结构体 thread_struct 来保存进程的内核栈的栈指针、返回地址等关键信息。

调度函数使用宏 switch_to 切换进程，我们来仔细观察以下这段代码，为了看起来更清晰，删除了代码中的注释：

linux-3.7.4/arch/x86/include/asm/switch_to.h:

```
01 #define switch_to(prev, next, last)                    \
02 do {  ...                                              \
03     asm volatile("pushfl\n\t"                          \
04                  "pushl %%ebp\n\t"                     \
05                  "movl %%esp,%[prev_sp]\n\t"           \
06                  "movl %[next_sp],%%esp\n\t"           \
07                  "movl $1f,%[prev_ip]\n\t"             \
08                  "pushl %[next_ip]\n\t"                \
09                  __switch_canary                       \
```

```
10              "jmp __switch_to\n"                                    \
11              "1:\t"                                                 \
12              "popl %%ebp\n\t"                                       \
13              "popfl\n"                                              \
14              ...                                                    \
15              : [prev_sp] "=m" (prev->thread.sp),                    \
16                [prev_ip] "=m" (prev->thread.ip),                    \
17                ...                                                  \
18              : [next_sp]  "m" (next->thread.sp),                    \
19                [next_ip]  "m" (next->thread.ip),                    \
20                ...);                                                \
21 } while (0)
```

在每次进程切换时，调度函数将准备切出的进程的寄存器 esp 中的值保存在其任务结构中，见第 5 行代码。然后从下一个投入运行的进程的任务结构中恢复 esp，见第 6 行代码。除了栈指针外，程序下一次恢复运行时的地址也有一点点复杂，不仅仅是简单的保存 eip 中的值，有一些复杂情况需要考虑，比如稍后我们会看到对于新创建的进程，其恢复运行的地址的设置。所以调度函数也将 eip 保存到了任务结构中，第 7 行代码就是保存被切出进程下次恢复时的运行地址。第 8 行代码和第 10 行的 jmp，以及函数 __switch_to 最后的 ret 指令联手将投入运行的进程的地址，即 next->thread.ip，恢复到寄存器 eip 中。

除了 eip、esp 外，宏 switch_to 将其他寄存器如 eflags、ebp 等保存到了进程内核栈中。

每次中断时，CPU 会从 TSS 段中取出当前进程的内核栈的栈指针，因此，当发生任务切换时，TSS 段中的 esp0 的值也要更新为下一个投入运行任务的内核栈的栈指针。在宏 switch_to 中，即上面第 10 行代码处，调用函数 __switch_to 的目的就是设置 TSS 段中的 esp0：

```
linux-3.7.4/arch/x86/kernel/process_32.c:

__notrace_funcgraph struct task_struct *__switch_to(...)
{
    ...
    load_sp0(tss, next);
    ...
}

linux-3.7.4/arch/x86/include/asm/processor.h:

static inline void load_sp0(...)
{
    native_load_sp0(tss, thread);
}

static inline void native_load_sp0(...)
{
    tss->x86_tss.sp0 = thread->sp0;
    ...
}
```

综上，进程在内核中的切换过程如图 5-27 所示，其中 next 表示即将投入运行的任务，

prev 表示当前任务，但是马上将被切出。被切出进程下一次恢复运行时的地址并不一定是就是当前指令指针中的地址，所以图中 eip 使用了虚线，其表达的意图就是进程恢复运行时的地址也保存在了进程的任务结构中。

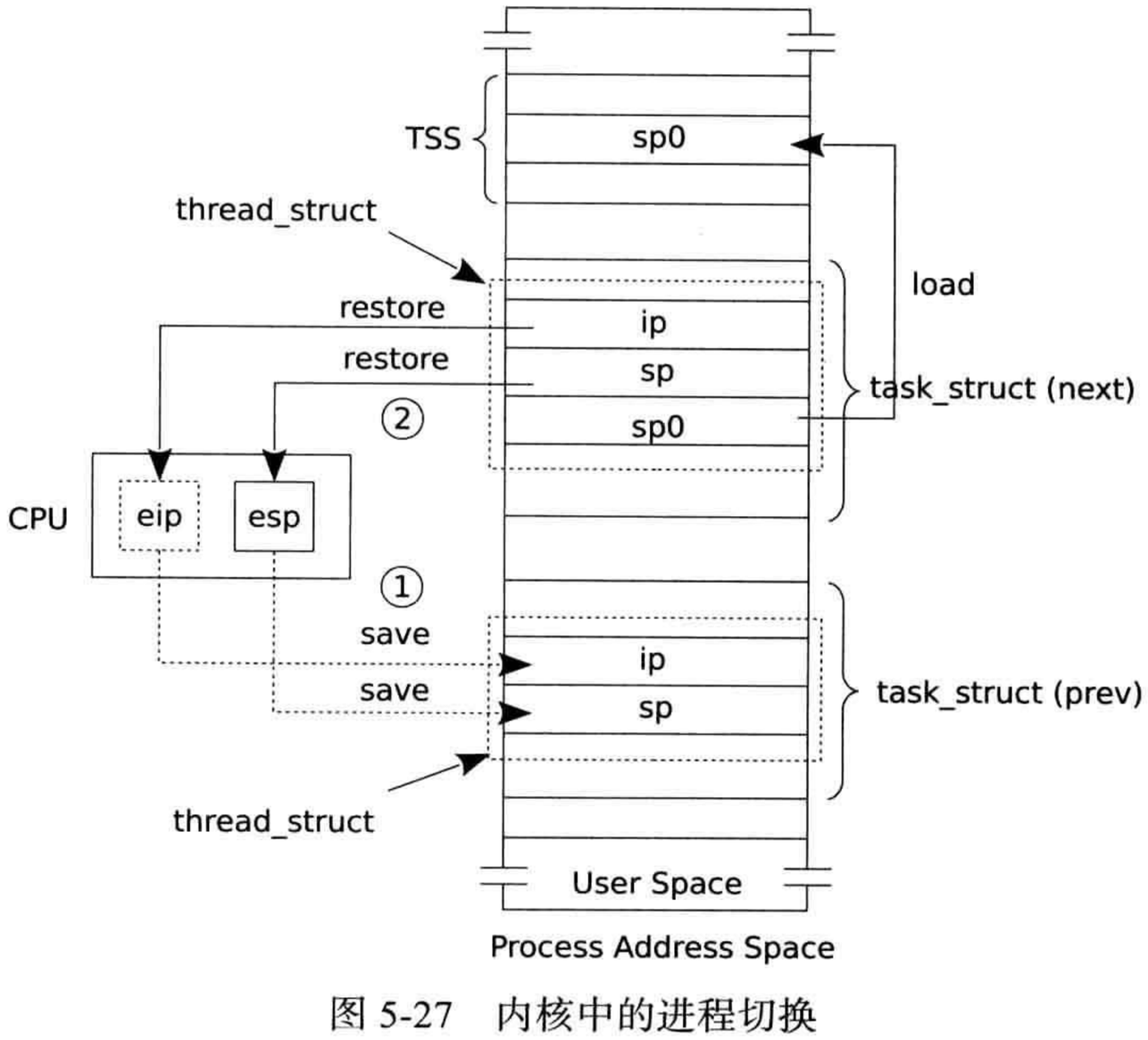


图 5-27　内核中的进程切换

3. 伪造现场

我们先来看看内核是如何伪造内核现场的。其实伪造内核现场只需要伪造三个关键的地方：进程恢复运行时的地址，即 eip；进程的内核栈的栈指针，这个无需解释了，进程当然需要知道目前的栈顶在哪里；最后还要准备内核栈的栈底，为了日后在进程返回到用户空间后，发生中断时，CPU 可以找到内核栈，从内核栈的栈底开始保存用户现场。

根据前面的讨论，如图 5-27，在调度器调度下一个进程投入运行时，即宏 switch_to 中，将从下一个投入运行的进程的任务结构的 thread_struct 中加载 sp 到寄存器 esp；加载 ip 到寄存器 eip；并在这个宏调用的函数 __switch_to 中，将 TSS 段中的 sp0 指向 thread_struct 中的 sp0。因此，要伪造这三个数据，只需设置任务结构中的结构体 thread_struct 中下面几个值即可：

```
linux-3.7.4/arch/x86/include/asm/processor.h:

struct thread_struct {
    ...
    unsigned long       sp0;
    unsigned long       sp;
    ...
    unsigned long       ip;
```

```
    ...
};
```

进程的任务结构在复制进程时创建，因此，这几个数据在复制进程时伪造是再合适不过了。复制进程时，与结构体 thread_struct 相关的复制函数为 copy_thread，其代码如下：

linux-3.7.4/arch/x86/kernel/process_32.c:

```
01 int copy_thread(..., struct task_struct *p, struct pt_regs *regs)
02 {
03     struct pt_regs *childregs = task_pt_regs(p);
04     ...
05     p->thread.sp = (unsigned long) childregs;
06     p->thread.sp0 = (unsigned long) (childregs+1);
07
08     if (unlikely(!regs)) {
09         ...
10         p->thread.ip = (unsigned long) ret_from_kernel_thread;
11         ...
12     }
13     ...
14     p->thread.ip = (unsigned long) ret_from_fork;
15     ...
16 }
```

先来看一下结构体 pt_regs，这个结构体就是为了解释内核栈底部保存的进程的用户现场而设计的，其中的字段完全按照压栈的各个寄存器的顺序设计。第 3 行代码中的宏 task_pt_regs 就是获取内核栈中 pt_regs 的，并使用 childregs 指向这个区域。

显然，第 5 行代码是在伪造栈指针。第 6 行代码是在为 TSS 段伪造内核栈的栈底。但是这两个变量的值可能让人有些困惑，我们通过图 5-28 来直观展示一下。

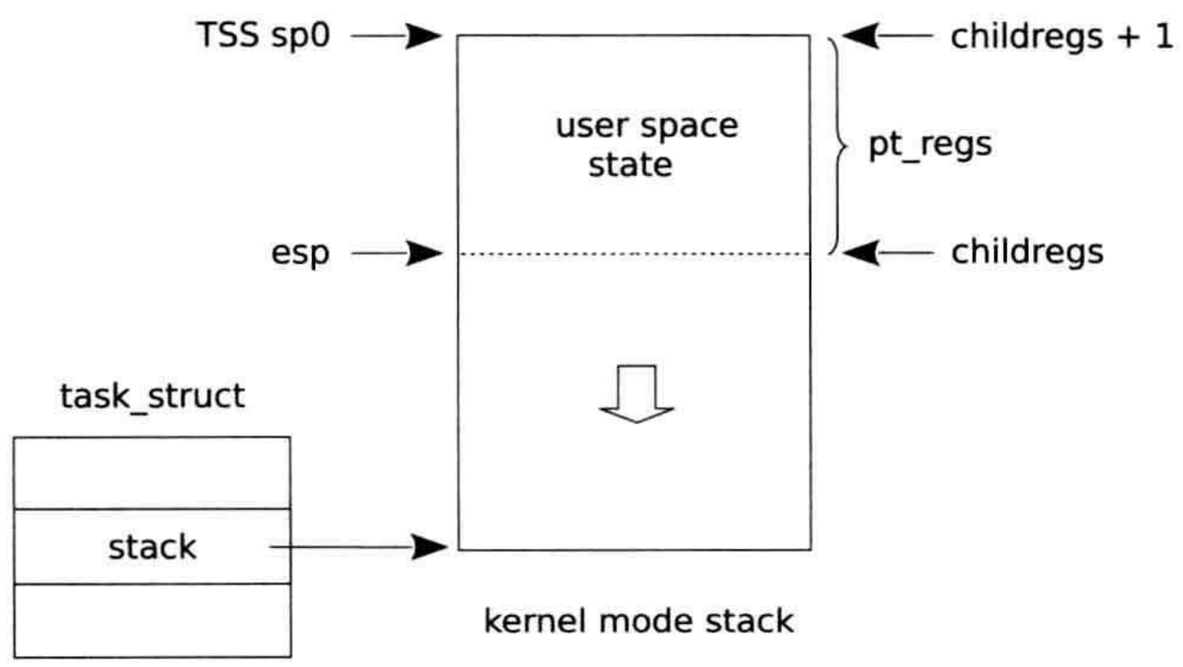


图 5-28 进程内核栈

根据图 5-28 可见，childregs 是进程在内核态运行时使用的内核栈的栈底。childregs+1 是在 childregs 的基础上，向地址增大方向偏移一个 pt_regs 的大小。也就是说，从 childregs 到 childregs+1 正是给伪造用户现场预留的空间。

函数 copy_thread 中的第 10 行和第 14 行都是在为新复制的进程伪造返回时的运行地

址。只不过第10行是针对内核线程的，从进程0复制进程就属于这种情况。函数ret_from_kernel_thread与ret_from_fork唯一的不同是，ret_from_kernel_thread在返回到用户空间前，其将执行一个函数，然后才恢复进程的用户现场，具体如下：

```
linux-3.7.4/arch/x86/kernel/entry_32.S:

ENTRY(ret_from_kernel_thread)
    ...
    call *PT_EBX(%esp)
    ...
ENDPROC(ret_from_kernel_thread)
```

PT_EBX(%esp)就是pt_regs中寄存器ebx处的值，显然，这个值是一个函数地址。那么，这个新复制的进程，在返回用户空间之前到底执行了一个什么函数呢？在函数copy_thread伪造eip时，其实已经设置了寄存器ebx的值：

```
linux-3.7.4/arch/x86/kernel/process_32.c:

01 int copy_thread(unsigned long clone_flags, unsigned long sp,
02   unsigned long arg, struct task_struct *p, struct pt_regs *regs)
03 {
04     struct pt_regs *childregs = task_pt_regs(p);
05     ...
06     if (unlikely(!regs)) {
07         ...
08         p->thread.ip = (unsigned long) ret_from_kernel_thread;
09         ...
10         childregs->bx = sp; /* function */
11         ...
12     }
13     ...
14 }
```

寄存器ebx中保存的是函数copy_thread的第2个参数sp，我们再来看看这个参数是什么：

```
linux-3.7.4/kernel/fork.c:

static struct task_struct *copy_process(unsigned long clone_flags,
                    unsigned long stack_start, ...)
{
    ...
    retval = copy_thread(clone_flags, stack_start, ...);
    ...
}

long do_fork(unsigned long clone_flags,
          unsigned long stack_start, ...)
{
    ...
    p = copy_process(clone_flags, stack_start, ...);
    ...
}
```

我们来回顾一下进程 1 的创建过程：

```
linux-3.7.4/init/main.c:

static noinline void __init_refok rest_init(void)
{
    ...
    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    ...
}

linux-3.7.4/kernel/fork.c

pid_t kernel_thread(int (*fn)(void *), void *arg, ...)
{
    return do_fork(flags|CLONE_VM|CLONE_UNTRACED,
            (unsigned long)fn, NULL, ...);
}
```

可见，寄存器 ebx 中保存的是函数 kernel_init 的地址。而恰恰是 kernel_init，开启了创建进程的第二阶段，即 exec 的过程。也就是说，在复制进程后，在返回用户空间之前，内核开启了 exec 过程，加载可执行程序。与我们编写普通程序时，在复制之后，使用 exec 执行新的程序异曲同工。

显然，在加载可执行程序之后，是一个伪造用户现场的合理时机。因为，只有这个时候，才知道新执行的程序的入口地址，也就是进程切换到用户空间后的执行地址。而且，也只有在进程的地址空间建立好之后，才能知道进程的用户栈的位置。相关代码如下：

```
linux-3.7.4/fs/binfmt_elf.c:

static int load_elf_binary(struct linux_binprm *bprm, ...)
{
    ...
    start_thread(regs, elf_entry, bprm->p);
    ...
}
```

在加载了可执行文件后，函数 load_elf_binary 确定了进程在用户空间的入口 elf_entry，也确定了进程的用户栈所在的位置 bprm->p，然后调用函数 start_thread 伪造进程的用户空间的现场，代码如下：

```
linux-3.7.4/arch/x86/kernel/process_32.c:

void start_thread(struct pt_regs *regs, unsigned long new_ip,
    unsigned long new_sp)
{
    set_user_gs(regs, 0);
    regs->fs        = 0;
    regs->ds        = __USER_DS;
    regs->es        = __USER_DS;
    regs->ss        = __USER_DS;
```

```
    regs->cs        = __USER_CS;
    regs->ip        = new_ip;
    regs->sp        = new_sp;
    ...
}
```

在函数 start_thread 中，各个段寄存器均被设置为了用户空间的段，进程的栈也指向了用户空间的栈。于是，在 ret_from_kernel_thread 最后，从进程的内核栈恢复用户现场时，进程被彻底打回原形，从天上掉到了人间，进程又作回了凡人，在用户空间从入口地址 elf_entry 处开始执行。

5.4.3 按需载入指令和数据

在建立进程的地址空间时，我们看到，内核仅仅是将地址映射进来，没有加载任何实际指令和数据到内存中。这主要还是出于效率的考虑，一个进程的所有指令和数据并不一定全部要用到，比如某些处理错误的代码。某些错误可能根本不会发生，如果也将这些错误代码加载进内存，就是白白占据内存资源。而且特别对于某些大型程序，如果启动时全部加载进内存，也会使启动时间延长，让用户难以忍受。所以，在实际需要这些指令和数据时，内核才会通过缺页中断处理函数将指令和数据从文件按需加载进内存。这一节，我们就来具体讨论这一过程。

1. 获取引起缺页异常的地址

IA32 架构的缺页中断的处理函数 do_page_fault 调用函数 __do_page_fault 处理缺页中断，相关代码如下：

```
linux-3.7.4/arch/x86/mm/fault.c:

01 static void __kprobes __do_page_fault(struct pt_regs *regs, ...)
02 {
03     ...
04     address = read_cr2();
05     ...
06     vma = find_vma(mm, address);
07     if (unlikely(!vma)) {
08         bad_area(regs, error_code, address);
09         return;
10     }
11     if (likely(vma->vm_start <= address))
12         goto good_area;
13     if (unlikely(!(vma->vm_flags & VM_GROWSDOWN))) {
14         bad_area(regs, error_code, address);
15         return;
16     }
17     ...
18     if (unlikely(expand_stack(vma, address))) {
19         bad_area(regs, error_code, address);
20         return;
21     }
```

```
22      ...
23 good_area:
24      ...
25      fault = handle_mm_fault(mm, vma, address, flags);
26      ...
27 }
```

在发生缺页中断时，寄存器 CR2 中保存的是引起缺页中断的线性地址。因此，第 4 行代码首先到寄存器 CR2 中读取线性地址。然后，函数 __do_page_fault 检查这个地址的合法性，判断条件就是这个地址是否在某个 vma 的范围内。注意这里的函数 find_vma，它从映射进程地址空间底部的 vm_area_struct 对象开始遍历，当找到第一个结束地址恰好要大于这个异常地址、但是却是最接近这个异常地址的 vm_area_struct 对象时，就将这个对象返回。如果找不到这样一个 vm_area_struct 对象，那么就说明这个地址是非法的，内核将向进程发送 SIGSEGV 信号，这是程序运行时出现 segment fault 的原因之一，见代码第 6~10 行。

如果找到了一个 vm_area_struct 对象，并且引起异常的这个地址也大于这个 vm_area_struct 对象的起始地址，那么就说明这个地址恰好在其涵盖的范围内，直接跳转到标号 good_area 处，见代码第 11~12 行。

但是，假如引起异常的这个地址小于 vm_area_struct 对象的起始地址，那也不能一棍子打死，我们还要给它一次机会。在前面讨论栈段的创建时，我们已经看到，内核初始只为进程分配了一个页面大小的空间，因此，其完全有可能是一次压栈操作，但是栈空间尚未映射具体的物理地址，如图 5-29 所示。

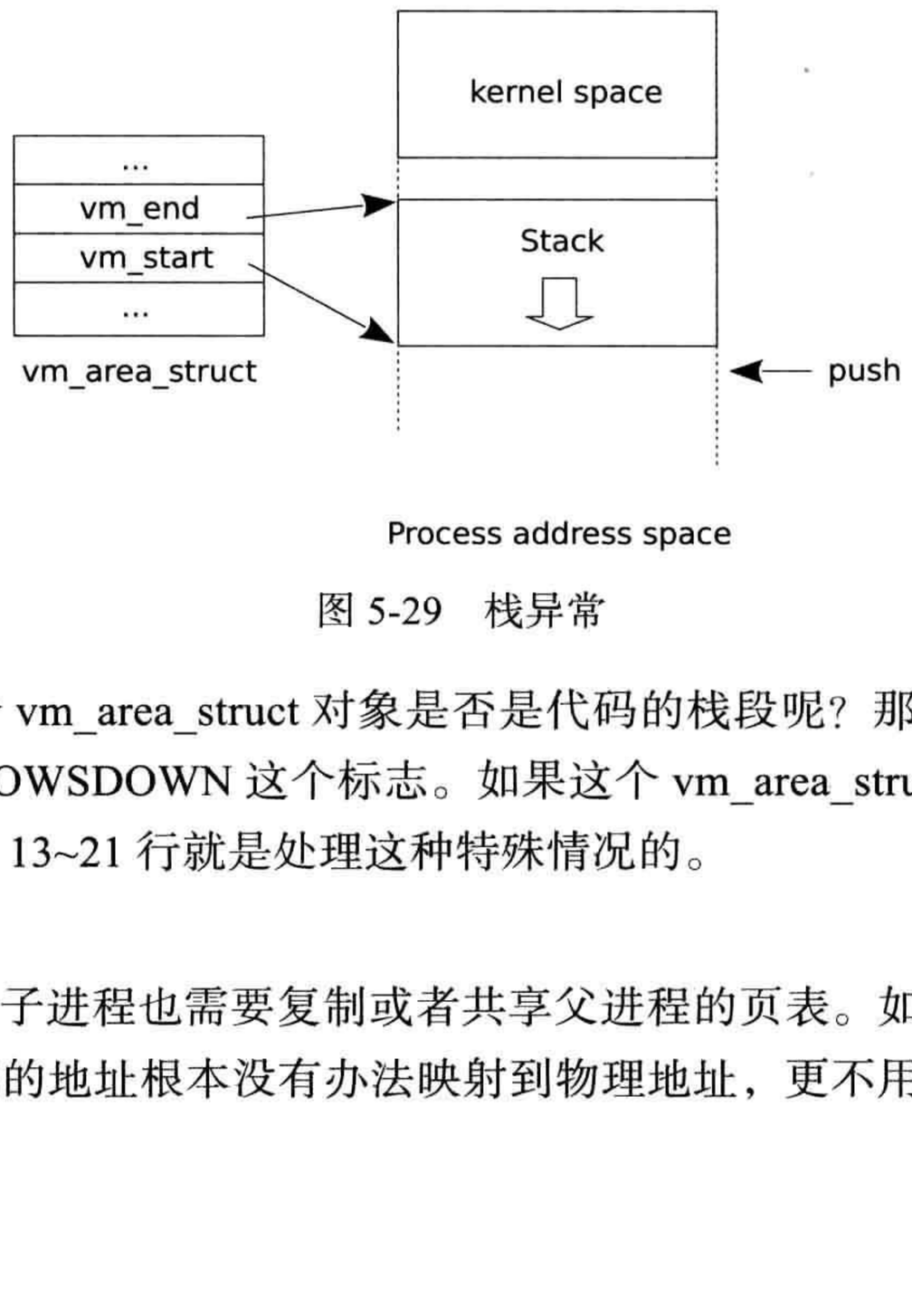


图 5-29 栈异常

那么如何判断这个 vm_area_struct 对象是否是代码的栈段呢？那就要看其成员 vm_flags 中是否设置了 VM_GROWSDOWN 这个标志。如果这个 vm_area_struct 对象是栈段，则首先对栈进行扩展。代码第 13~21 行就是处理这种特殊情况的。

2. 更新页表

在复制子进程时，子进程也需要复制或者共享父进程的页表。如果没有页表，子进程寸步难行，指令或者数据的地址根本没有办法映射到物理地址，更不用提从物理内存读取指令

了。当子进程替换（exec）为一个新的程序时，无论子进程是共享或者复制了父进程的页表，子进程都需要创建新的页表。创建页表的函数如下：

```
linux-3.7.4/arch/x86/mm/pgtable.c:

pgd_t *pgd_alloc(struct mm_struct *mm)
{
    ...
    pgd = (pgd_t *)__get_free_page(PGALLOC_GFP);
    ...
    pgd_ctor(mm, pgd);
    ...
}

static void pgd_ctor(struct mm_struct *mm, pgd_t *pgd)
{
    if (PAGETABLE_LEVELS == 2 || ...) {
        clone_pgd_range(pgd + KERNEL_PGD_BOUNDARY,
                swapper_pg_dir + KERNEL_PGD_BOUNDARY,
                KERNEL_PGD_PTRS);
    }
    ...
}
```

函数 pgd_alloc 申请了一个物理内存页面，然后调用函数 pgd_ctor 将存储在 swapper_pg_dir 处的页目录中的映射内核空间的页目录项复制过来。我们看到，这里没有复制映射用户空间的页目录项，而且也不需要复制，因为用户空间需要映射到一个新的程序。于是，页目录中映射用户空间的这些页目录项的自然为空，更不用提那些还没有影儿的页表了。当访问地址落在这些空的页目录项映射范围内时，自然就引发了缺页异常。那么在缺页异常处理函数中，自然就需要分配页面、分配页表、更新页目录、更新页表项等。

为了可以映射更大的地址空间，Linux 中使用多个级别的页面映射。因此，我们先来理解一下内核中页表的管理。比如缺页异常处理函数调用的函数 handle_mm_fault，代码如下：

```
linux-3.7.4/mm/memory.c:

int handle_mm_fault(...)
{
    pgd_t *pgd;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;
    ...
    pgd = pgd_offset(mm, address);
    pud = pud_alloc(mm, pgd, address);
    if (!pud)
        return VM_FAULT_OOM;
    pmd = pmd_alloc(mm, pud, address);
    ...
}
```

从上述代码中我们看到了 pgd、pud、pmd 和 pte。读者应该可以猜出来，内核使用了

4 级页表机制。但是这 4 级页表是如何与物理上的页表结合的呢？我们以没有开启 PAE 的 IA32 架构为例，来探讨这一过程。

对于没有启用 PAE 的 IA32，其映射的地址空间为 4GB，所以理论上内核使用与 IA32 物理上相同的 2 级页表就足够了。但是为了代码的一致性，内核依然保留了 pud 和 pmd 等概念。只不过通过一些巧妙的定义，绕过了中间的 pud 和 pmd。当配置内核时，如果未开启支持 IA32 的 PAE 特性，内核实质上将使用 2 级页表。内核使用了文件 pgtable-nopud.h 和 pgtable-nopmd.h 中有关 pud 和 pmd 的一些定义：

```
linux-3.7.4/arch/x86/include/asm/pgtable_types.h:

#if PAGETABLE_LEVELS > 3
...
#else
#include <asm-generic/pgtable-nopud.h>
...
#endif

#if PAGETABLE_LEVELS > 2
...
#else
#include <asm-generic/pgtable-nopmd.h>
...
#endif
```

从文件名字我们就已经看出了内核的意图，就是要屏蔽 pud 和 pmd 层。下面，我们就结合这两个文件中的定义，来看看内核是如何绕过 pud 和 pmd 的。以函数 handle_mm_fault 中使用的函数 pud_alloc 为例：

```
linux-3.7.4/include/linux/mm.h:

static inline pud_t *pud_alloc(struct mm_struct *mm, pgd_t *pgd,
                               unsigned long address)
{
    return (unlikely(pgd_none(*pgd)) && __pud_alloc(mm, pgd,
            address))? NULL: pud_offset(pgd, address);
}
```

在文件 pgtable-nopud.h 中函数 pgd_none 的定义为：

```
linux-3.7.4/include/asm-generic/pgtable-nopud.h:

static inline int pgd_none(pgd_t pgd)        { return 0; }
```

也就是说，函数 pgd_none 永远返回 0，那么 pud_alloc 中的函数 __pud_alloc 就不需要执行了，而且 pud_alloc 返回的值就是函数 pud_offset 的返回值，这个函数当然也对应的是文件 pgtable-nopud.h 中的实现：

```
linux-3.7.4/include/asm-generic/pgtable-nopud.h:
```

```
static inline pud_t * pud_offset(pgd_t * pgd, unsigned long address)
{
    return (pud_t *)pgd;
}
```

看到函数 pud_offset 的实现，我想读者一定会恍然大悟。显然，pud 就是 pgd。pmd_alloc 亦是如此处理的，读者可以仿照上面的方法自行查看。也就是说在使用了 2 级页表的情况下，pud 和 pmd 就像透明的空气，根本不存在，内核绕了一个圈后又回到原点。虽然代码中还有所谓的 pud、pmd 等，但是所谓的 pud 和 pmd 都是 pgd。

读者亦不要被诸如 pgd_t、pud_t、pmd_t 等这些封装的数据类型所迷惑，说白了，它们就是一些表项中的值。再直白一点，就是一个整数，或者至多是个整数的数组而已。这里之所以使用了一个结构体将这些整数封装起来，就是为了屏蔽这些表项的细节，避免日后的改动影响更多的代码。

了解了内核的页表管理机制后，下面我们就来具体看一下函数 handle_mm_fault。

```
linux-3.7.4/mm/memory.c:

01 int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct
02     *vma, unsigned long address, unsigned int flags)
03 {
04     pgd_t *pgd;
05     pud_t *pud;
06     pmd_t *pmd;
07     pte_t *pte;
08     ...
09     pgd = pgd_offset(mm, address);
10     pud = pud_alloc(mm, pgd, address);
11     if (!pud)
12         return VM_FAULT_OOM;
13     pmd = pmd_alloc(mm, pud, address);
14     ...
15     if (unlikely(pmd_none(*pmd)) && __pte_alloc(mm, vma, pmd,
16                                         address))
17     ...
18 }
```

函数 handle_mm_fault 首先确定引起异常的地址是由哪个页目录项映射的，见第 9 行代码。

在确定了页目录项 pgd 后，如前面讨论，我们可以无视第 10~13 行关于 pud 和 pmd 的部分。后面的 pud 和 pmd 都是 pgd。

确定了页目录项后，第 15 行代码就要判断页目录项是否为空。如果页目录项为空，显然还要分配页表。前面我们已经看到，对于一个新加载的程序，页目录表中映射用户空间的页目录项是空的。所以 pmd_none 的值一定是 True。进而继续执行函数 __pte_alloc 分配页表，代码如下：

```
linux-3.7.4/mm/memory.c:
```

```
int __pte_alloc(struct mm_struct *mm, struct vm_area_struct *vma,
        pmd_t *pmd, unsigned long address)
{
    pgtable_t new = pte_alloc_one(mm, address);
    ...
    if (likely(pmd_none(*pmd))) {  /* Has another populated it ? */
        mm->nr_ptes++;
        pmd_populate(mm, pmd, new);
        new = NULL;
    } else if (unlikely(pmd_trans_splitting(*pmd)))
        ...
}
```

函数 __pte_alloc 首先调用 pte_alloc_one 分配了一个物理页面承载页表，然后调用函数 pmd_populate，将这个页表的地址填充进页目录表中对应的表项。这里，对于使用 2 级页表的情况，pmd 表就是页目录表，所以函数 pmd_populate 也不是在填充什么 pmd 表，而是填充页目录表。

3. 从文件载入指令和数据

页表准备就绪后，handle_mm_fault 最后准备载入指令和数据了：

```
linux-3.7.4/mm/memory.c:

int handle_mm_fault(...)
{
    ...
    pte = pte_offset_map(pmd, address);

    return handle_pte_fault(mm, vma, address, pte, pmd, flags);
}
```

handle_mm_fault 首先调用函数 pte_offset_map 取得引起异常的地址在页表中的页表项。毫无疑问，这个页表项 pte 也是空的。然后 handle_mm_fault 调用函数 handle_pte_fault 从文件载入指令和数据：

```
linux-3.7.4/mm/memory.c:

01 int handle_pte_fault(struct mm_struct *mm,
02     struct vm_area_struct *vma, unsigned long address,
03     pte_t *pte, pmd_t *pmd, unsigned int flags)
04 {
05     pte_t entry;
06     spinlock_t *ptl;
07
08     entry = *pte;
09     if (!pte_present(entry)) {
10         if (pte_none(entry)) {
11             if (vma->vm_ops) {
12                 if (likely(vma->vm_ops->fault))
13                     return do_linear_fault(mm, vma, address,
14                         pte, pmd, flags, entry);
15             }
```

```
16              return do_anonymous_page(...);
17          }
18          if (pte_file(entry))
19              return do_nonlinear_fault(...);
20          return do_swap_page(...);
21      }
22      ...
23  }
```

handle_pte_fault 首先调用 pte_present 检查这个 pte 是否存在，见第 9 行代码。

如果不存在，那么可能有两种情况：第一种情况是页面是首次访问，也就是这个页表项是彻底的空的，而不仅仅是没有置位 present，在这种情况下，需要建立页面映射，见代码第 10~17 行；第二种情况是页面映射已经建立了，只不过是目前交换出内存了，即代码第 18~20 行处理的情况。我们只讨论第一种情况。

对于第一种情况，也存在两种情况：一种是如果 vm_area_struct 对象中的成员 vm_ops 存在，并且 vm_ops 中提供了函数 fault，那么说明段是映射到文件的，见代码第 11~15 行；否则是匿名映射。这里我们只讨论典型的从文件加载的情况，代码如下：

```
linux-3.7.4/mm/memory.c:

static int do_linear_fault(struct mm_struct *mm, struct
          vm_area_struct *vma, unsigned long address,
          pte_t *page_table, pmd_t *pmd, ...)
{
    pgoff_t pgoff = (((address & PAGE_MASK)
          - vma->vm_start) >> PAGE_SHIFT) + vma->vm_pgoff;

    pte_unmap(page_table);
    return __do_fault(mm, vma, address, pmd, pgoff, flags,
orig_pte);
}
```

do_linear_fault 这个函数非看似简单，仅一条计算指令，计算了变量 pgoff 的值，然后就将后续处理丢给了函数 __do_fault。但是小计算大智慧，不要小看这条计算指令，它计算出的 pgoff 是从文件载入指令和数据的关键。

我们知道，每次从文件加载指令或者数据时，都是以页面为单位的，所以我们可以将文件想象为多个连续的页面。那么如何确定引起异常的这个地址对应于文件中的哪个页面呢?

事实上，当从文件中将段映射到进程地址空间时，创建的段的 vm_area_struct 对象中的成员 vm_pgoff 已经记录了段在文件中的偏移，而且是以页为单位的。一个段可以占据一个或者多个页面。

当发生缺页异常时，虽然不能确定引起异常的地址是在文件中的哪一个页面，但是可以计算出这个地址相对于段的起始地址的差值。将这个差值转换为以页为单位，再加上段在文件中的偏移，即可确定这个地址在文件中的哪个页面上。

我们用图 5-30 来更直观地表示一下这个过程。图 5-30 表示数据段及其相应的 vm_area_

struct 对象，其中使用虚线框起来的页是数据段所映射的范围。

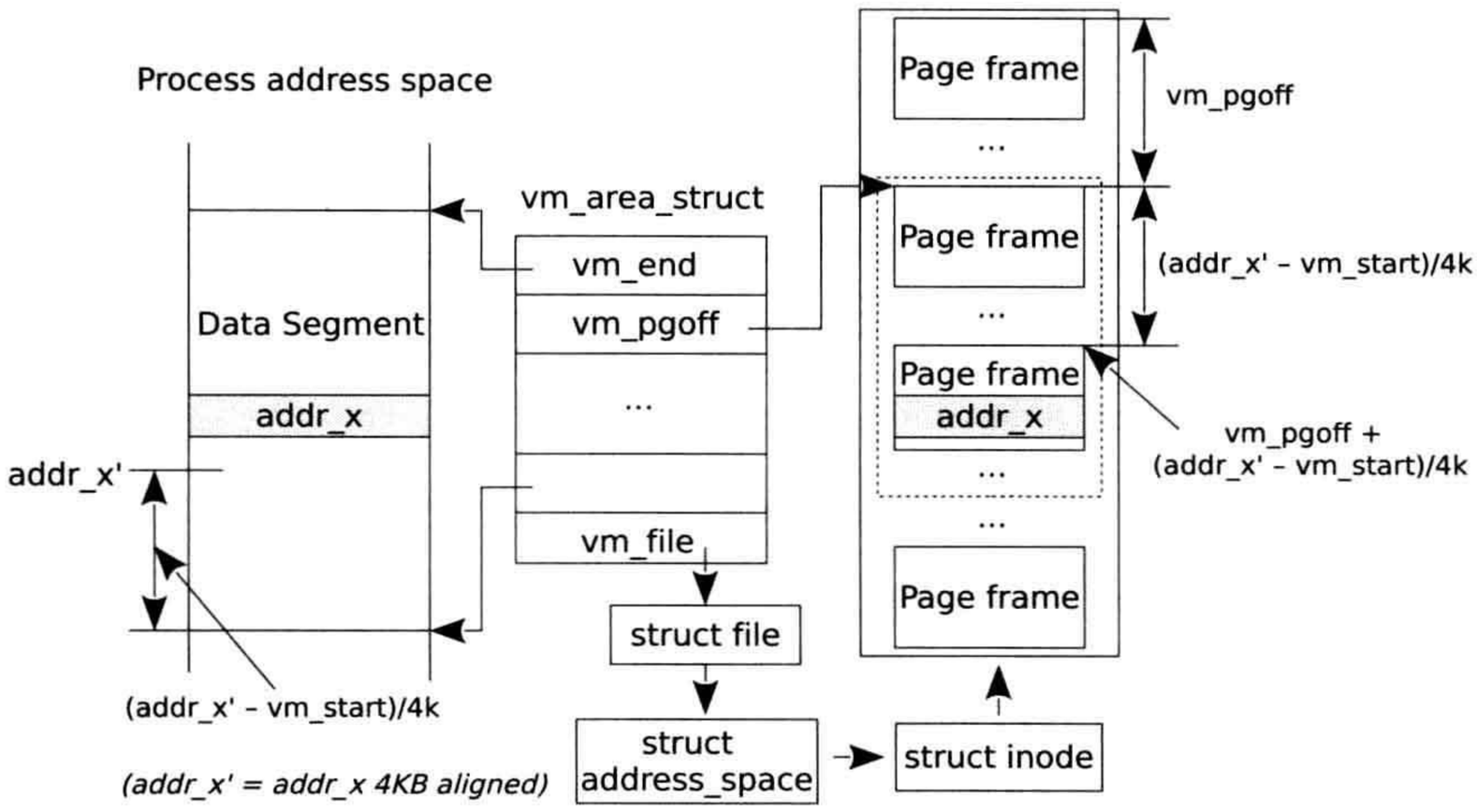


图 5-30 异常地址到页面的计算

我们以下面程序中变量 g_a 为例，具体体验一下这个偏移的计算过程。

```
int g_a = 100;

void main()
{
}
```

为了更具有代表性，我们使用静态链接，这样编译出的可执行文件尺寸大一点，页面偏移可以多一点。

```
root@baisheng:~/demo# gcc -static -o hello main.c
```

（1）段在文件的偏移（vm_pgoff）

因为变量 g_a 在数据段，所以我们看看数据段在可执行文件中的偏移：

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

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg
  LOAD           0x000000 0x08048000 0x08048000 0xa517d 0xa517d R E
  LOAD           0x0a5fa4 0x080eefa4 0x080eefa4 0x00cdc 0x023c8 RW
  ...
```

我们看到数据段在文件中的偏移是 0x0a5fa4，按照页面对齐后，数据段应该从文件中偏移 0x0a5000 处开始映射。而 0x0a5000/0x1000 = 165，也就是说，数据段映射的文件的起始位置是第 165 个页。

（2）引起异常的地址在段内的偏移

一个段可能会映射到文件中的多个页，所以我们还要计算具体的地址在段内的偏移（以

页为单位）。

```
root@baisheng:~/demo# readelf -s hello | grep g_a
  2184: 080ef068     4 OBJECT  GLOBAL DEFAULT   24 g_a
```

变量 g_a 的地址为 0x080ef068。因为映射是以页为单位的，所以这个地址应该包含在从 0x080ef000 到 0x080f0000 一个页面中。因此，使用地址 0x080ef000 与段的起始地址 0x080ee000 做差，从而得出这个地址所在页在段内的偏移：

```
0x080ef000 - 0x080ee000 = 0x1000
```

即偏移一个页。也就是说，在段在文件中的偏移的基础上，再偏移一个页就可以了，即载入文件第 166（165 + 1）页的数据到内存。

根据上面的讨论可见，do_linear_fault 这个函数的主要目的正如同其名字一样，是处理这个线性的缺页异常地址，将其从线性地址转换为相应的页单元。偏移这个参数准备好了，我们继续往下看 __do_fault。

```
linux-3.7.4/mm/memory.c:

static int __do_fault(struct mm_struct *mm, struct vm_area_struct
    *vma, unsigned long address, pmd_t *pmd, pgoff_t pgoff, ...)
{
    ...
    struct vm_fault vmf;
    ...
    vmf.virtual_address = (void __user *)(address & PAGE_MASK);
    vmf.pgoff = pgoff;
    ...
    ret = vma->vm_ops->fault(vma, &vmf);
    ...
    page = vmf.page;
    ...
    page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
    ...
        entry = mk_pte(page, vma->vm_page_prot);
        ...
        set_pte_at(mm, address, page_table, entry);
    ...
}
```

函数 __do_fault 调用具体文件系统中的 fault 函数，将所需的文件数据读入到内存。至于读入哪个页面，当然要使用刚刚计算的 pgoff 了。以 ext4 文件系统为例，其为 vma 提供的 vm_operations_struct 如下：

```
linux-3.7.4/fs/ext4/file.c:

static const struct vm_operations_struct ext4_file_vm_ops = {
    .fault      = filemap_fault,
    .page_mkwrite   = ext4_page_mkwrite,
    .remap_pages    = generic_file_remap_pages,
};
```

ext4 文件系统中的 filemap_fault 将指定偏移处的页面读入内存，其中参数 vmf 中的 page 就是指向从文件载入的页面。

载入页面后，还有最后一步要做：更新页表项。函数 __do_fault 锁定页表中映射这个页面的页表项，然后调用函数 mk_pte 创建页表项的值，最后调用 set_pte_at 将页表项的值填充到页表中对应的页表项。

5.4.4 加载动态链接器

在现代操作系统中，绝大部分程序都是动态链接的。对于动态链接的程序，除了加载可执行程序外，其依赖的动态库也要加载。对于动态链接的程序和库，编译时并不能确定引用的外部符号的地址，因此在加载后，还要进行符号重定位。

为了降低内核的复杂度，上述工作并没有包含在内核中，而是转移到了用户空间，由用户空间的程序来完成这个过程。这个程序被称为动态加载 / 链接器（dynamic linker/loader），一般也将其简称为动态链接器。后续行文中，凡是没有使用“动态”二字修饰的链接器，均指编译时的链接器。内核只负责将动态链接器加载到内存，其他的都交由动态链接器去处理。

为了更大的灵活性，内核不会假定系统中使用动态链接器，而是由可执行程序主动告诉内核谁是动态链接器。当编译一个可执行程序时，链接器将创建一个类型为“INTERP”的段，这个段非常简单，就是包含一个字符串，这个字符串就是动态链接器的名字，以可执行程序 hello 为例：

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

Program Headers:
  Type           Offset   VirtAddr   PhysAddr    FileSiz  MemSiz
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013
      [Requesting program interpreter: /lib/ld-linux.so.2]
  ...
```

由上可见，类型为“INTERP”的段就是一个 19（0x13）个字符长的字串“/lib/ld-linux.so.2”，正是动态链接器。

当内核加载可执行程序时，其将检查可执行程序的 Program Header Table 中是否包含有类型为“INTERP”的段，代码如下：

```
linux-3.7.4/fs/binfmt_elf.c:

static int load_elf_binary(struct linux_binprm *bprm, ...)
{
    struct file *interpreter = NULL; /* to shut gcc up */
    ...
    char * elf_interpreter = NULL;

    for (i = 0; i < loc->elf_ex.e_phnum; i++) {
        if (elf_ppnt->p_type == PT_INTERP) {
```

```
        ...
        elf_interpreter = kmalloc(elf_ppnt->p_filesz,
                        GFP_KERNEL);
        ...
        retval = kernel_read(bprm->file, elf_ppnt->p_offset,
                    elf_interpreter, elf_ppnt->p_filesz);
        ...
        interpreter = open_exec(elf_interpreter);
        ...
}
        elf_ppnt++;
    }
    ...
}
```

如果有 INTERP 的段，那么说明这个 ELF 文件是一个动态链接的可执行程序。Linux 中动态链接器以动态库的方式实现，于是内核需要将动态链接器这个动态库加载到进程的地址空间，代码如下：

```
linux-3.7.4/fs/binfmt_elf.c:

static int load_elf_binary(struct linux_binprm *bprm, ...)
{
    ...
    if (elf_interpreter) {
        ...
        elf_entry = load_elf_interp(&loc->interp_elf_ex,
                    interpreter, &interp_map_addr, load_bias);
        if (!IS_ERR((void *)elf_entry)) {
            ...
            interp_load_addr = elf_entry;
            elf_entry += loc->interp_elf_ex.e_entry;
        }
    ...
    }
    ...
    start_thread(regs, elf_entry, bprm->p);
    ...
}
```

加载动态链接器与加载可执行程序的过程基本完全相同，函数 load_elf_interp 就是一个简化版的 load_elf_binary，这里我们不再赘述。完成动态链接器加载后，需要跳转到动态链接器的入口继续执行。那么，如何确定动态链接器的入口地址呢？动态链接器的 ELF 头中将记录一个入口地址：

```
root@baisheng:/vita/sysroot/lib# readelf -h ld-2.15.so |
                                    grep -i entry
  Entry point address:              0x1050
```

难道编译时链接器计算错了？ 0x1050 不太像进程地址空间的虚拟地址。没错，0x1050 是虚拟地址，只不过是因为在编译时不能确定动态库的加载地址，所以动态库中地址分配从

0 开始，见下面动态库的 Program Header Table：

```
root@baisheng:/vita/sysroot/lib# readelf -l ld-2.15.so

Program Headers:
  Type           Offset   VirtAddr    PhysAddr    FileSiz MemSiz
  LOAD           0x000000 0x00000000 0x00000000 0x1f47c 0x1f47c
  LOAD           0x01fcc0 0x00020cc0 0x00020cc0 0x00bb8 0x00c78
  ...
```

函数 load_elf_interp 返回的是动态链接器在进程地址空间中的映射的基址，所以在这个基址加上入口地址 0x1050 后才是动态链接器的入口的真正的运行时地址。计算好动态链接器的入口地址后，内核调用函数 start_thread，伪造了用户现场。在进程切换到用户空间时，将跳转到动态链接器的入口处开始执行。

我们看看动态链接器入口地址对应的符号：

```
root@baisheng:/vita/sysroot/lib# readelf -s ld-2.15.so | grep 1050
   443: 00001050     0 NOTYPE  LOCAL  DEFAULT   10 _start
```

可见，动态链接器的入口是符号 _start：

```
glibc-2.15/sysdeps/i386/dl-machine.h:

_start:\n\
    # Note that _dl_start gets the parameter in %eax.\n\
    movl %esp, %eax\n\
    call _dl_start\n\
_dl_start_user:\n\
    # Save the user entry point address in %edi.\n\
    movl %eax, %edi\n\
    ...
    jmp *%edi\n\
    ...
```

函数 _start 调用 _dl_start 在进行一些自身的必要的准备工作。其中最重要的一点是动态链接器也是一个动态库，其在进程地址空间中的地址也是加载时才确定的，因此动态链接器也需要重定位，我们将在 5.4.8 节讨论这一过程。

然后，_dl_start 调用函数 dl_main 加载动态库以及重定位工作。其中，加载动态库的过程在 5.4.5 节讨论，重定位动态库的过程在 5.4.6 节讨论，有关重定位可执行程序的部分将在 5.4.7 节讨论。

在完成加载及重定位后，函数 _dl_start 将返回可执行程序的入口地址。因此，汇编指令从寄存器 eax 中取出可执行程序的入口地址，并临时保存到寄存器 edi。在这段程序的最后，通过指令“jmp *%edi”跳转到可执行程序的入口处开始执行可执行程序。

另外，我们再留意一下上面代码中的标号 _dl_start_user。从这个标号处开始，到最后跳转到可执行程序的入口前，动态链接器将调用动态库相关的一些初始化函数。前面在第 2 章中最后在动态库的初始化部分添加的那个函数，就是在这里执行的。

我们以一个具体的例子看看动态链接器在进程地址空间中映射的情况：

```
root@baisheng:~# cat /proc/self/maps
...
08048000-08053000 r-xp 00000000 08:01 261656     /bin/cat
...
b7736000-b7756000 r-xp 00000000 08:01 523936
                                /lib/i386-linux-gnu/ld-2.15.so
b7756000-b7757000 r--p 0001f000 08:01 523936
                                /lib/i386-linux-gnu/ld-2.15.so
b7757000-b7758000 rw-p 00020000 08:01 523936
                                /lib/i386-linux-gnu/ld-2.15.so
...
```

可见，对于这个进程，动态链接器被映射到进程地址空间从0xb7736000开始的地方，这个就是我们前面提到的动态库在进程地址空间中映射的基址。其中0xb7736000~0xb7756000这个段的权限是“rx”，显然这个段应该是代码段和一些只读的数据；0xb7756000~0xb7757000和0xb7757000~0xb7758000都对应的是数据段。但是为什么数据段被划分为两个段？其实不只是动态链接器如此，包括其他动态库和动态链接的可执行程序都是如此，具体原因我们将在5.4.9节讨论。

5.4.5 加载动态库

加载动态库前，首先需要知道这个可执行程序依赖的动态库，当然也包括这些动态库依赖的动态库，因此，这是一个递归的过程。那么动态链接器是如何知道这些依赖的动态库呢？动态链接器不是一个人在战斗，在编译时，链接器已经为动态链接做了很多铺垫，其中之一就是在ELF文件中创建了一个段“.dynamic”，保存的全部是与动态链接相关的信息。

我们观察一下可执行程序hello中的段“.dynamic”：

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

Dynamic section at offset 0xf0c contains 25 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libf1.so]
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 ...
0x00000003 (PLTGOT)                      0x804a000
 0x00000002 (PLTRELSZ)                   40 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x8048430
 0x00000011 (REL)                        0x8048420
...
```

段“.dynamic”中记录了多组与动态库有关的信息，每一组信息都使用如下格式保存：

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

typedef struct
```

```
{
  Elf32_Sword   d_tag;                    /* Dynamic entry type */
  union
    {
      Elf32_Word d_val;                   /* Integer value */
      Elf32_Addr d_ptr;                   /* Address value */
    } d_un;
} Elf32_Dyn;
```

可见，每组信息使用的是 tag/value 的形式保存，只不过 value 有的是个整数值，有的是地址而已。

其中类型（Type）为“NEEDED”的项记录的就是可执行程序依赖的动态库。可以看到，hello 依赖动态库 libc.so.6 和 libf1.so。

动态链接器设计了一个数据结构来代表每个加载到内存的动态库（包括可执行程序），定义如下：

```
glibc-2.15/include/link.h:

struct link_map
  {
    ElfW(Addr) l_addr;

    char *l_name;

    ElfW(Dyn) *l_ld;

    struct link_map *l_next, *l_prev;
    ...
    ElfW(Dyn) *l_info[DT_NUM + DT_THISPROCNUM + DT_VERSIONTAGNUM
                      + DT_EXTRANUM + DT_VALNUM + DT_ADDRNUM];
    ...
  };
```

这个数据结构中记录了动态库重定位需要的关键两项信息：l_addr 和 l_ld。l_addr 记录的是动态库在进程地址空间中映射的基址，有了这个参照，动态链接器才可以修订符号的运行时地址；l_ld 指向动态库的段“.dynamic”，通过这个参数，动态链接器可以知道一切与动态重定位相关的信息。为了方便，结构体 link_map 中定义了一个数组 l_info，将段“.dynamic”中的信息记录在这个数组中，就不必每次使用时再去重新解析“.dynamic”了。

当内核将控制权转交给动态链接器时，链接器首先为即将处理的可执行程序创建一个 link_map 对象，在动态链接器代码中将其命名为 main_map。然后，动态链接器找到这个可执行程序依赖的动态库，当然也包括其依赖的动态库也依赖的动态库，依次链接在 main_map 的后面，形成一个 link_map 对象链表。动态链接器作为动态库依赖的一个动态库，自然也包含在这个链表中。沿着这个链表，动态链接器将动态库映射进进程地址空间，并进行重定位。

函数 dl_main 调用 _dl_map_object_deps 加载可执行程序依赖的所有动态库，代码如下：

```
glibc-2.15/elf/rtld.c:

static void dl_main (const ElfW(Phdr) *phdr, ...)
{
  ...
  _dl_map_object_deps (main_map, ...);
  ...
}
glibc-2.15/elf/dl-deps.c:

void internal_function
_dl_map_object_deps (struct link_map *map, ...)
{
  ...
  for (runp = known; runp; )
    {
      ...
            int err = _dl_catch_error (&objname, &errstring,
                                       &malloced, openaux, &args);
...
    }
  ...
}

static void openaux (void *a)
{
  ...
  args->aux = _dl_map_object (args->map, args->name, ...);
}
```

函数 dl_main 给函数 _dl_map_object_deps 传递了一个参数 main_map，前面提到过这个参数，就是可执行程序的 link_map 对象。函数 _dl_map_object_deps 遍历可执行程序依赖的所有动态库，对每一个动态库调用函数 _dl_map_object 将这些动态库全部映射到进程的地址空间，代码如下：

```
glibc-2.15/elf/dl-load.c:

struct link_map * internal_function _dl_map_object (...)
{
  ...
  return _dl_map_object_from_fd (name, fd, &fb, realname,
                 loader, type, mode, &stack_end, nsid);
}
```

因为动态库可能已经被加载到内存了，所以 _dl_map_object 首先从已经映射的动态库中寻找。如果没有找到，则调用函数 _dl_map_object_from_fd 从文件系统加载，代码如下：

```
glibc-2.15/elf/dl-load.c:

struct link_map * _dl_map_object_from_fd (...)
{
  ...
        l->l_map_start = (ElfW(Addr)) __mmap ((void *) mappref,
```

```
                maplength, c->prot, MAP_COPY|MAP_FILE, fd, c->mapoff);
    ...
          l->l_addr = l->l_map_start - c->mapstart;
    ...
}
```

_dl_map_object_from_fd 调用函数 __mmap 映射文件中的段到进程地址空间，并将映射基址记录到 link_map 中的 l_addr。至于函数 __mmap，读者应该已经隐隐猜出来了，没错，这个函数就是我们编写应用程序时使用的 C 库的函数 mmap，只不过这是在 C 库内部调用，所以函数名称略有差别，其实现如下：

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

ENTRY (__mmap)
    ...
    movl $SYS_ify(mmap2), %eax  /* System call number in %eax.*/

    /* Do the system call trap.  */
    ENTER_KERNEL
    ...
```

_mmap 首先将系统调用号 __NR_mmap2 装入寄存器 eax，然后就向内核请求服务。这里请求内核服务时之所以没有使用 0x80 中断，原因是为了在支持快速系统调用（Fast System Call）指令 sysenter/sysexit 的 CPU 上使用这些比 0x80 中断更优化的系统调用指令。

在映射了程序段后，函数 _dl_map_object_from_fd 调用了函数 __mprotect 设置段的读、写以及可执行权限，__mprotect 使用的是内核调用号为 __NR_mprotect 的服务。

我们看到，动态链接器并没有发明什么新的魔法，它只是使用内核提供的系统调用将动态库映射到进程的地址空间。也就是说，虽然动态库是由动态链接器在用户空间进程映射的，但是本质上的映射动作还是由内核完成的。

最后，_dl_map_object_from_fd 将 link_map 中的成员 l_ld 指向了段“.dynamic”所在的位置：

```
glibc-2.15/elf/dl-load.c:

struct link_map * _dl_map_object_from_fd (const char *name, ...)
{
  ...
    for (ph = phdr; ph < &phdr[l->l_phnum]; ++ph)
      switch (ph->p_type)
        {
        ...
        case PT_DYNAMIC:
          l->l_ld = (void *) ph->p_vaddr;
        ...
        }
  ...
    l->l_ld = (ElfW(Dyn) *) ((ElfW(Addr)) l->l_ld + l->l_addr);
  ...
}
```

函数 _dl_map_object_from_fd 从 Program Header Table 中取出类型为“DYNAMIC”的段的地址，然后再加上动态库的映射基址。

最后，我们结合图 5-31 来直观地看一下多个进程是如何共享一个动态库的。

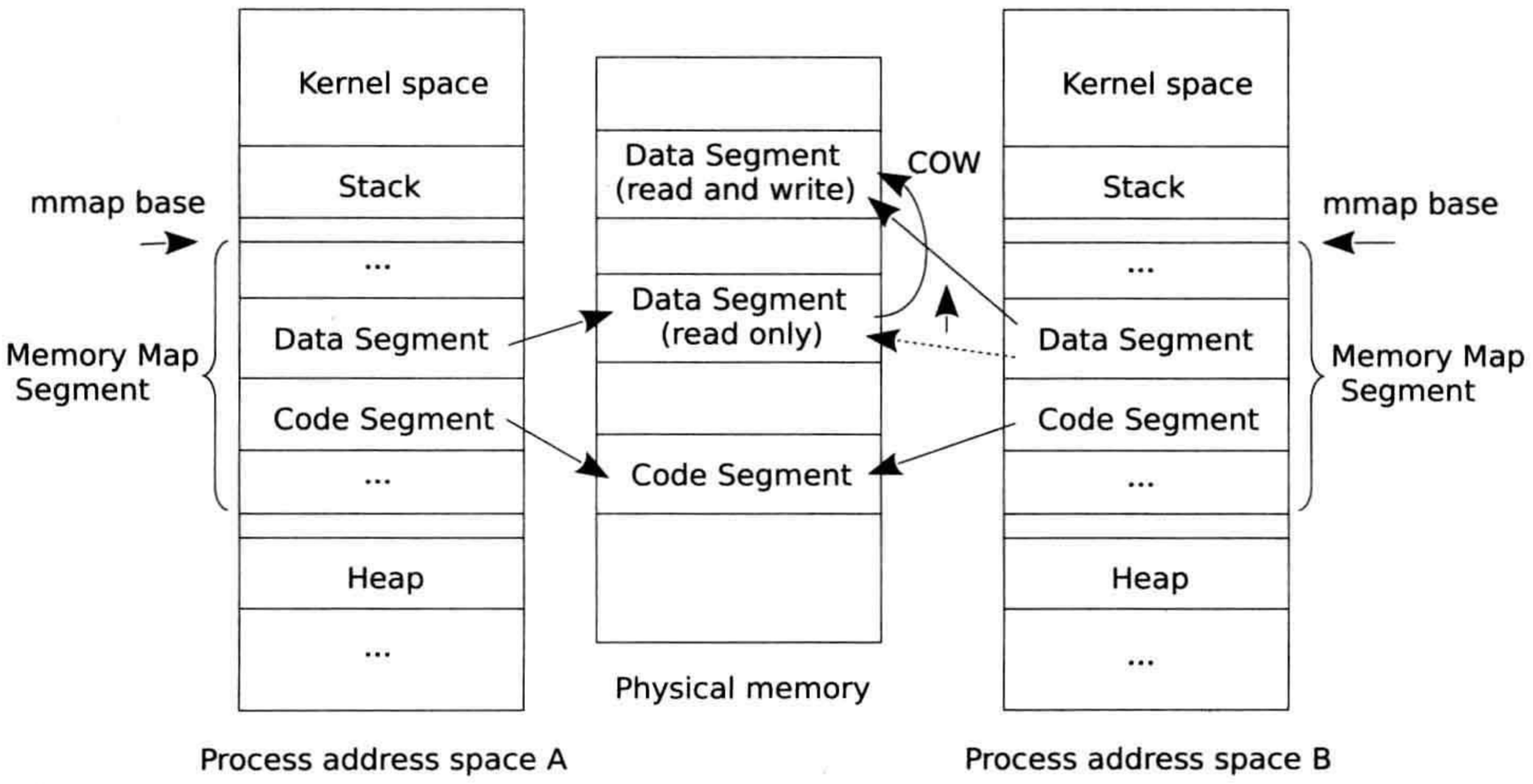


图 5-31 动态库的映射

最初，当共享库映射进内存后，代码段和数据段在物理内存中分别都只有一份副本，并且都是只读的，进程 A 和进程 B 共享只读的代码段和数据段。在进程运行过程中，当任一个进程试图修改数据段时，则内核将为这个进程复制一份私有的数据段的副本，而且这个数据段的权限被设置为可读写的。这里使用的策略就是所谓的写时复制（COW，Copy On Write）。但是这个复制动作不会影响进程的地址空间，对进程是透明的，只是同一段地址通过页面表映射到不同的物理页面而已。

5.4.6 重定位动态库

动态库在编译时，链接器并不知道最后被加载的位置，所以在编译时，共享库的地址是相对于 0 分配的。以动态库 libf1.so 为例：

```
root@baisheng:~/demo# readelf -l libf1.so

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz   MemSiz  Flg
  LOAD           0x000000 0x00000000 0x00000000 0x00658 0x00658 R E
  LOAD           0x000eec 0x00001eec 0x00001eec 0x00138 0x0013c RW
  ...
```

根据动态库 libf1.so 的 Program Header Table，注意列 VirtAddr，显然地址是从 0 开始分配的。因此，在映射到具体进程的地址空间后，需要修订其中那些通过绝对方式引用的符号的地址，代码如下：

```
glibc-2.15/elf/rtld.c:

static void dl_main (...)
{
  ...
      /* Now we have all the objects loaded. Relocate them all ...*/
      ...
      unsigned i = main_map->l_searchlist.r_nlist;
      while (i-- > 0)
        {
          struct link_map *l = main_map->l_initfini[i];
          ...
          if (l != &GL(dl_rtld_map))
            _dl_relocate_object (l, l->l_scope, ...);
          ...
        }
  ...
}
```

函数 dl_main 从 main_map 开始，调用函数 _dl_relocate_object 重定位 link_map 链表中的所有动态库和可执行程序，顺序是从后向前。如果有符号重定义了，那么后面发现的符号的地址将覆盖掉前面的符号地址。换句话说，链接时排在前面的动态库中的符号将被优先使用。另外，还有一点要注意，这个列表中的动态链接器将不再需要重定位，因为其已经在前面自己重定位好了。

常用的重定位方式有两种：加载时重定位（Load-time relocation）和 PIC 方式。

加载时重定位与编译时的重定位非常相似。动态链接器在加载动态库后，遍历动态库的重定位表，对于重定位表中的每一项记录，解析这个记录中指明的符号的地址，然后使用解析到的地址修订这个记录中指定的偏移处，当然这个偏移需要加上动态库映射的基址。

但是动态库是多个进程共享的，不同的进程映射的动态库的地址不同，因此，如果某个进程按照动态库在自己进程空间中映射的基址修改了动态库的代码段，那么这个动态库的显然就不能被其他进程所共享了，除非所有的进程映射动态库的位置相同，但是这又带来太多的限制和问题。

基于以上原因，开发者们又设计了另外一种方式——PIC（Position-Independent Code）。PIC 基于两个关键的事实：

- **数据段是可写的**。既然代码段是不能更改的，但是数据段总是可以更改的。于是 PIC 把重定位战场从代码段转移到数据段，在数据段中增加了一个 GOT（GLOBAL OFFSET TABLE）表。在编译时，链接器将所有需要重定位的符号在这个表中分配一项，其中记录了符号以及其实际所在的地址。重定位时，动态链接器只修改 GOT 表中的值。
- **代码和数据的相对位置不变**。在代码中凡是引用 GOT 表中的符号，只需要找到 GOT 表的地址，再加上变量在 GOT 表中的偏移即可。但是，如此还是没有避开代码段被修改的命运，因为动态库在进程地址空间中的位置只有在加载时才能确定，所以，

GOT 表的地址在加载时也需要重定位。但是，我们也注意到这样一个事实：对动态库来说，虽然其映射的地址在编译时不确定，但是在映射到进程的地址空间时，代码段和数据段依然按照编译时分配好的地址映射，也就是说，指令和数据的相对位置却是固定的。因此，GOT 表作为数据段中的一员，代码段中的任一指令与 GOT 表基址之间的偏移是固定的，在编译时就可以确定。PIC 恰恰是基于这个事实，在代码中凡是访问 GOT 表的地方，都是使用这个固定的相对偏移来引用 GOT 表以及其中的变量，因此，代码中引用 GOT 表的地址不再需要重定位，从而避开了代码段被修改的问题。接下来我们结合具体的实例进一步解释这个过程。

1. GOT 表

显然，PIC 技术中，GOT 表是一个非常重要的数据结构，在继续深入探讨前，我们先来认识一下这个数据结构，如图 5-32 所示。

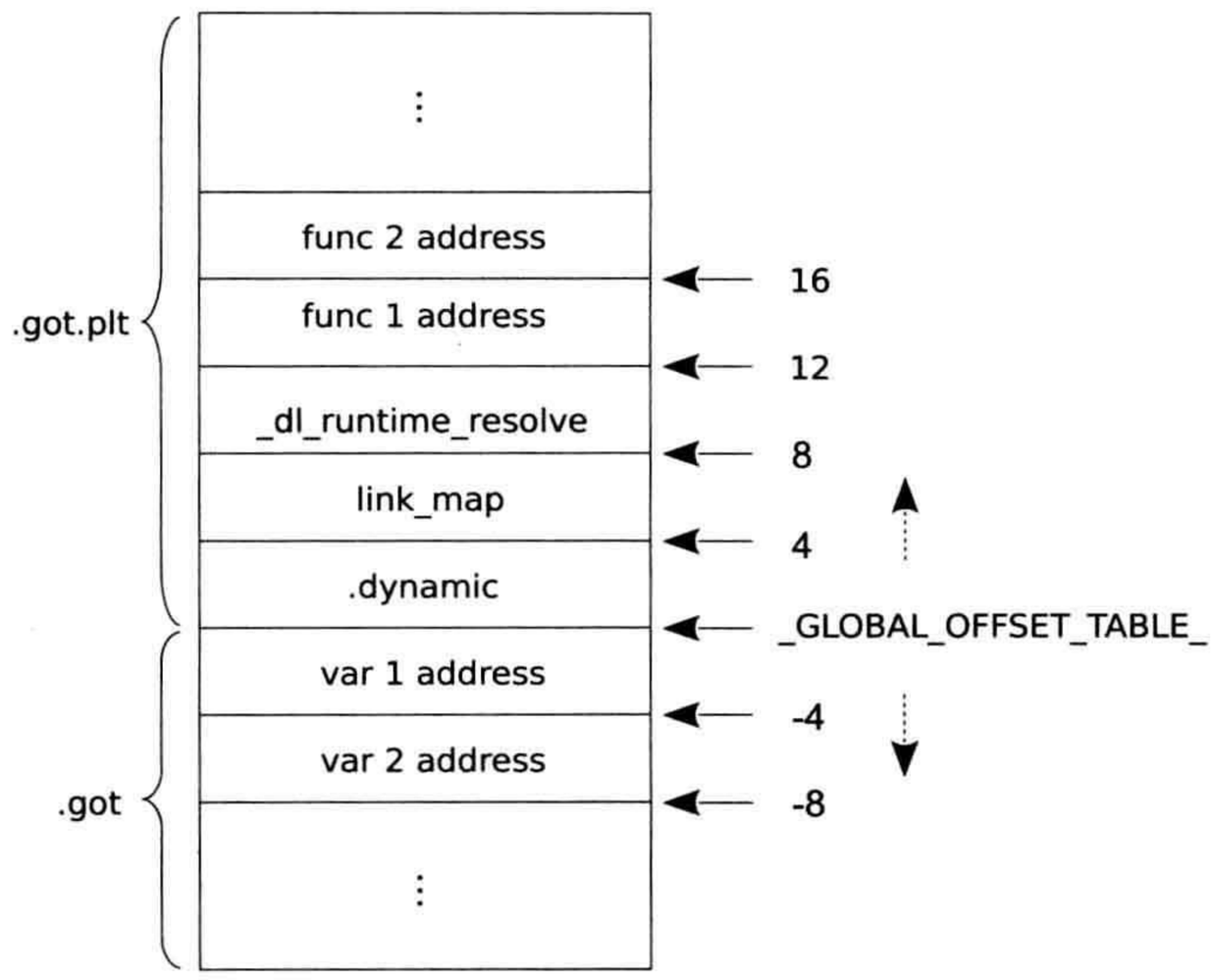


图 5-32 GOT 表

由图 5-32 可见，这么大名鼎鼎的 GOT 表却如此简单，其就是一个一维数组。对于 32 位 CPU 来说，每个数组元素就是 32 位的地址。GOT 表分成两个部分：.got 和 .got.plt。.got 中存储的是变量的地址。.got.plt 中存储的是函数的地址。在 5.4.9 节中我们将讨论 GOT 表一分为二的原因。

在编译时，链接器将定义一个符号 _GLOBAL_OFFSET_TABLE_，指向 .got 和 .got.plt 的连接处，凡是访问 GOT 表中的地址时，都使用基于这个符号的偏移。比如，访问变量 var 1，那么使用：

```
_GLOBAL_OFFSET_TABLE_ - 4
```

访问函数 func1 则使用：

```
_GLOBAL_OFFSET_TABLE_ + 12。
```

GOT 表中除了记录变量和函数的地址外，还有另外三个特殊的表项，我们在图 5-32 中也已经标出，它们就是 .got.plt 的前三项。其中第 1 项记录的是动态库或者可执行文件的 .dynamic 段的地址；第 2 项记录的是代表动态库或者可执行文件的 link_map 对象；第 3 项记录的是动态链接器提供的解析符号地址的函数 _dl_runtime_resolve 的地址。我们以动态库 libf1.so 为例，看看在一个已经编译好的动态库中，这三项的值：

```
root@baisheng:~/demo# readelf -x .got.plt libf1.so

Hex dump of section '.got.plt':
  0x00002000 f81e0000 00000000 00000000 36040000 ............6...
  0x00002010 46040000 56040000                    F...V...
```

从地址 0x2000 处起，就是 .got.plt 开始的地方。其中使用黑体标识的 3 个 32 位地址就分别是这三项的值。可见，除了第 1 项被赋予了具体的值外，其余两项全部是 0。原因是段 .dynamic 的地址是编译时就确定的。我们查看动态库 libf1.so 的段 .dynamic 的值：

```
root@baisheng:~/demo# readelf -S libf1.so

Section Headers:
[Nr] Name              Type            Addr     Off      Size
...
[18] .dynamic          DYNAMIC         00001ef8 000ef8 0000e8
...
```

上面，使用“-x”显示段 .got.plt 的内容时，是以 little-endian 表示的，所以 .dynamic 段的地址“00001ef8”被显示为“f81e0000”。

记录动态库信息的 link_map 是在加载后创建的，编译时当然不知道这个运行时创建的对象的地址。同理，因为动态链接器也是以动态库的形式加载到进程地址空间的，其映射地址也是加载时才确定的，所以动态链接器中的函数 _dl_runtime_resolve 的地址也是在动态链接器加载后才能确定。因此，与段 .dynamic 的地址在编译时就可确定不同，这两项是由动态链接器动态填充的，代码如下：

```
glibc-2.15/sysdeps/i386/dl-machine.h:

1  static inline int __attribute__ ((unused, always_inline))
2  elf_machine_runtime_setup (struct link_map *l, ...)
3  {
4    Elf32_Addr *got;
5    ...
6        got = (Elf32_Addr *) D_PTR (l, l_info[DT_PLTGOT]);
7
8        got[1] = (Elf32_Addr) l; /*Identify this shared object.*/
9
10     got[2] = (Elf32_Addr) &_dl_runtime_resolve;
```

```
11    ...
12  }
```

其中第6行语句将相关宏进行替换后，展开如下：

```
got = (Elf32_Addr *) l->l_info[DT_PLTGOT].d_un.d_ptr;
```

前面，讨论结构体link_map时，我们提到过，这个结构体中的数组l_info就是为了方便存储段.dynamic的信息的。因此，这条语句的目的就是从段.dynamic中取得GOT表的基地址，也就是got.plt的基址。

接下来的第8行和第10行语句的目的是在获得了.got.plt的基址之后，分别设置其中第2项和第3项的值。很明显，一个是代表动态库的link_map对象，另外一个就是函数_dl_runtime_resolve的地址。

读者这里了解GOT表中这特殊的三项就可以了，更具体的我们后面会讨论。其中第1项主要是动态链接器重定位自己时使用，我们将在5.4.8节讨论；第2项和第3项主要是用在函数的延迟绑定中使用，我们在5.4.6节中讨论。

2. 重定位变量

变量的重定位在动态库加载时进行，注意不要将这里的加载时与前面特指的“加载时重定位”混淆，这里指的是使用PIC技术在加载时进行的变量重定位的过程。我们分别从代码中引用变量以及动态链接器修订GOT表两个角度来讨论PIC中的变量重定位。

（1）代码中引用变量

我们以库libf1中的函数foo1_func引用库libf2中的符号foo2为例，具体看一下PIC中的变量重定位。我们反汇编动态库libf1.so，其中引用全局变量foo2的反汇编代码片段如下：

```
root@baisheng:~/demo# objdump -d libf1.so

0000057b <__x86.get_pc_thunk.bx>:
 57b:      8b 1c 24                mov    (%esp),%ebx
 57e:      c3                      ret
 57f:      90                      nop

00000580 <foo1_func>:
 580:      55                      push   %ebp
 581:      89 e5                   mov    %esp,%ebp
 583:      53                      push   %ebx
 584:      83 ec 14                sub    $0x14,%esp
 587:      e8 ef ff ff ff          call   57b
              <__x86.get_pc_thunk.bx>
 58c:      81 c3 74 1a 00 00       add    $0x1a74,%ebx
 592:      8b 83 e8 ff ff ff       mov    -0x18(%ebx),%eax
 598:      8b 00                   mov    (%eax),%eax
 ...
```

1）获取下一条指令的运行时地址。注意偏移0x587处的指令，其调用了偏移0x57b处的函数__x86.get_pc_thunk.cx。在调用这个函数时，call指令会将下一条指令的地址0x58c压

入到栈中。而在进入函数 __x86.get_pc_thunk.cx 后，其将栈顶的值取出到寄存器 ebx 中，然后返回。显然，调用这个函数的目的就是取得下一条指令的运行时地址。这里之所以这么做，是因为 x86 指令集中没有提供获取指令指针值的指令，不得以才采用的一个小技巧。

2）计算 GOT 表的运行时地址。现在，下一条指令的绝对地址保存在寄存器 ebx 中，而下一条指令与 GOT 之间的偏移又是固定的，因此寄存器 ebx 加上这个固定的偏移后，就确定了 GOT 表在运行时所在的地址。

编译时，链接器定义了一个变量 _GLOBAL_OFFSET_TABLE_ 代表 GOT 表的基址，库 libf1 中该符号地址如下：

```
root@baisheng:~/demo# readelf -s libf1.so

Symbol table '.symtab' contains 58 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
    ...
    42: 00002000   0 OBJECT LOCAL DEFAULT   20 _GLOBAL_OFFSET_TABLE_
    ...
```

因此，库 libf1 中偏移 0x58c 处的指令到 GOT 表所在位置的差为：0x2000 – 0x58c = 0x1a74，这就是地址 0x58c 处的值 0x1a74 的由来。也就是说，这个 0x1a74 就是指令与 GOT 表之间的那个固定偏移。

3）计算符号 foo2 在 GOT 表中的偏移。取得了 GOT 表的绝对地址后，如要访问变量 foo2，还要加上变量 foo2 在 GOT 表中的偏移。那这个偏移是多少呢？我们看看动态库 libf1 的重定位表：

```
root@baisheng:~/demo# readelf -r libf1.so

Relocation section '.rel.dyn' at offset 0x38c contains 11 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
...
00001fe8  00000206 R_386_GLOB_DAT    00000000   foo2
...
```

根据重定位表可见，符号 foo2 在偏移 0x00001fe8 处。而 GOT 表基址在 0x2000 处，因此，根据这两个值之差就可以确定符号 foo2 在 GOT 表中的偏移：0x1fe8 – 0x2000 = –0x18，也就是说，变量 foo2 相对 GOT 表的偏移是 –0x18。根据 ELF 文件中段的布局：

```
root@baisheng:~/demo# readelf -S libf1.so

Section Headers:
  [Nr] Name              Type            Addr     Off    Size
  ...
  [19] .got              PROGBITS        00001fe0 000fe0 000020
  [20] .got.plt          PROGBITS        00002000 001000 000018
  ...
```

可见，GOT 表的基址是介于 .got 和 .got.plt 之间的。对于 .got 部分来说，GOT 表的基址

位于 .got 部分的底部，这就是偏移为负的原因。之所以将 GOT 表的基址设置在 .got 和 .got.plt 之间，并无特别的目的，这样访问 .got.plt 就是正值了。所以，我们看到在库 libf1 的地址 0x592 处在 ebx 的基础上又加了偏移 –0x18。

（2）动态链接器修订 GOT 表

我们还是以库 libf1 中引用的库 libf2 中的符号 foo2 为例，来看看在加载时，动态链接器是如何解析这个符号并修订 GOT 表的。

1）获取动态库 libf1 的重定位表。重定位信息保存在重定位表中，因此，动态链接器首先要找到重定位表。段 .dynamic 中类型为 REL 的条目记录的就是重定位表的位置，动态库 libf1 段 .dynamic 中记录的重定位表如下：

```
root@baisheng:~/demo# readelf -d libf1.so

Dynamic section at offset 0xefc contains 25 entries:
  Tag        Type                         Name/Value
  ...
  0x00000011 (REL)                        0x38c
  ...
```

可见，保存重定位变量的表位于 0x38c 处。因此，动态链接器按照如下公式计算重定位表的地址：

```
link_map->l_addr + 0x38c
```

2）根据重定位表，确定需要修订的位置。确定重定位表后，动态链接器就遍历重定位表中的每一条记录。以 libf1.so 中的引用的全局变量 dummy、foo2 和 foo1 的重定位记录为例：

```
root@baisheng:~/demo# readelf -r libf1.so

Relocation section '.rel.dyn' at offset 0x38c contains 11 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
...
00001fe0  00000806 R_386_GLOB_DAT    00002020   dummy
...
00001fe8  00000206 R_386_GLOB_DAT    00000000   foo2
...
00001ff8  00000c06 R_386_GLOB_DAT    0000201c   foo1
...
```

其中第一条重定位记录表示需要使用符号 dummy 的值修订下面位置处的值：

```
link_map->l_addr + 0x1fe0
```

第二条重定位记录表示需要使用符号 foo2 的值修订下面位置处的值：

```
link_map->l_addr + 0x1fe8
```

第三条重定位记录表示需要使用符号 foo1 的值修订下面位置处的值：

```
link_map->l_addr + 0x1ff8
```

3）寻找动态符号表。需要修订的位置确定后，那么接下来就需要解析符号的值。动态链接器从 link_map 这个链表的表头，即代表可执行程序的 main_map 开始，依次在它们的动态符号表中查找符号。所以，要解析符号的地址，首先要确定动态符号表的地址。以动态库 libf2 为例，动态链接器确定其动态符号表的过程如下。

动态链接器根据代表库 libf2 的 link_map 中的字段 l_ld 找到段 .dynamic，然后在该段中取出动态符号表的地址：

```
root@baisheng:~/demo# readelf -d libf2.so

Dynamic section at offset 0xf0c contains 24 entries:
  Tag        Type                         Name/Value
  ...
  0x00000006 (SYMTAB)                     0x178
  ...
```

段 .dynamic 中类型为 SYMTAB 的项记录的是动态符号表的地址。可见，libf2 的动态符号表的地址是 0x178，因此，其在运行时的绝对地址使用如下公式计算：

```
link_map->l_addr + 0x178
```

4）解析符号地址。动态链接器找到了动态符号表后，进一步在动态符号表中查找符号的地址。以全局变量 foo2 为例，动态链接器将在库 libf2 的动态符号表中找到这个符号的信息：

```
root@baisheng:~/demo# readelf -s libf2.so

Symbol table '.dynsym' contains 13 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     ...
     9: 00002018     4 OBJECT  GLOBAL DEFAULT   21 foo2
     ...
```

上述动态符号表中符号的地址是相对于 0 的，因此需要加上 libf2 在进程地址空间中映射的基址，所以符号 foo2 的运行时地址是：

```
link_map_libf2->l_addr + 0x2018
```

然后，动态链接器使用上述这个地址，修订前面确定的需要修订的位置。

前面是静态的分析，下面我们将这个例子运行起来，动态地观察一下全局变量 foo2 的重定位过程。

```
root@baisheng:~/demo# gdb ./hello
(gdb) b main
Breakpoint 1 at 0x80485cf
 (gdb) r
Starting program: /root/demo/hello

Breakpoint 1, 0x080485cf in main ()
```

我们在另外一个终端中查看动态库 libf2 在进程 hello 的地址空间中映射的基址：

```
root@baisheng:~/demo# ps -C hello -o pid=
 2897
root@baisheng:~/demo# cat /proc/2897/maps
08048000-08049000 r-xp 00000000 08:01 1054223    /root/demo/hello
...
b7e15000-b7e16000 r-xp 00000000 08:01 1054350 /root/demo/libf2.so
...
b7fd8000-b7fd9000 r-xp 00000000 08:01 1047105 /root/demo/libf1.so
...
```

可见，库 libf1 和 libf2 在 hello 进程的地址空间中映射的基址分别是 0xb7fd8000 和 0xb7e15000。那么 libf1 中需要修订的地址是：

```
0xb7fd8000 + 0x1fe8 = 0xb7fd9fe8
```

符号 foo2 的地址是：

```
0xb7e15000 + 0x2018 = 0xb7e17018
```

下面我们使用 gdb 查看内存 0xb7fd9fe8 处的值，如果计算正确，那么该内存处的值应该已经被动态链接器修订为 0xb7e17018：

```
(gdb) x 0xb7fd9fe8
0xb7fd9fe8:        0xb7e17018
```

根据输出结果可见，内存 0xb7fd9fe8 处输出的值与我们理论上计算的符号 foo2 的地址完全吻合。

综上可知，变量 foo2 的重定位过程如图 5-33 所示。

不知道读者注意到没有，在例子中，我们在可执行文件 hello 和动态库 libf1 中分别定义了全局变量 dummy。这不是我们的笔误，而是故意为之。不知读者想过没有，对于变量 foo2，其定义在动态库 libf2 中，编译时动态库 libf1 对其一无所知，所以在加载时进行重定位，我们没有任何疑义。但是，对于变量 dummy，其在动态库 libf1 中已经定义了，既然指令和数据的相对位置是固定的，那么为什么不采用与寻址 GOT 表一样的方法，编译时就直接定义好位置，而还是通过 GOT 表，在加载时进行重定位呢?

我们先反过来问读者一个问题：动态库 libf1 中函数 foo1_func 中引用的变量 dummy 是动态库 libf1 中定义的，还是可执行程序 hello 中定义的? 答案是后者。对于一个全局符号，包括函数，其可能在本地定义，但在其他库中、甚至包括使用动态库的可执行程序中也可能有定义。在动态链接器解析符号时，将沿着以可执行程序的 link_map 对象 main_map 开头的这个链表依次查找动态符号表，使用最先找到的符号值。如我们的例子中，可执行程序 hello 的动态符号表将先于动态库 libf1 的动态符号表被查找，所以，库 libf1 中的函数 foo1_func 将使用可执行程序 hello 中 dummy 的定义。

除此之外，还有一种所谓的 Copy Relocation，也要求即使引用同一个动态库中定义的全

局变量，也要使用重定位的方式，我们在 5.4.7 节讨论这种重定位情况。

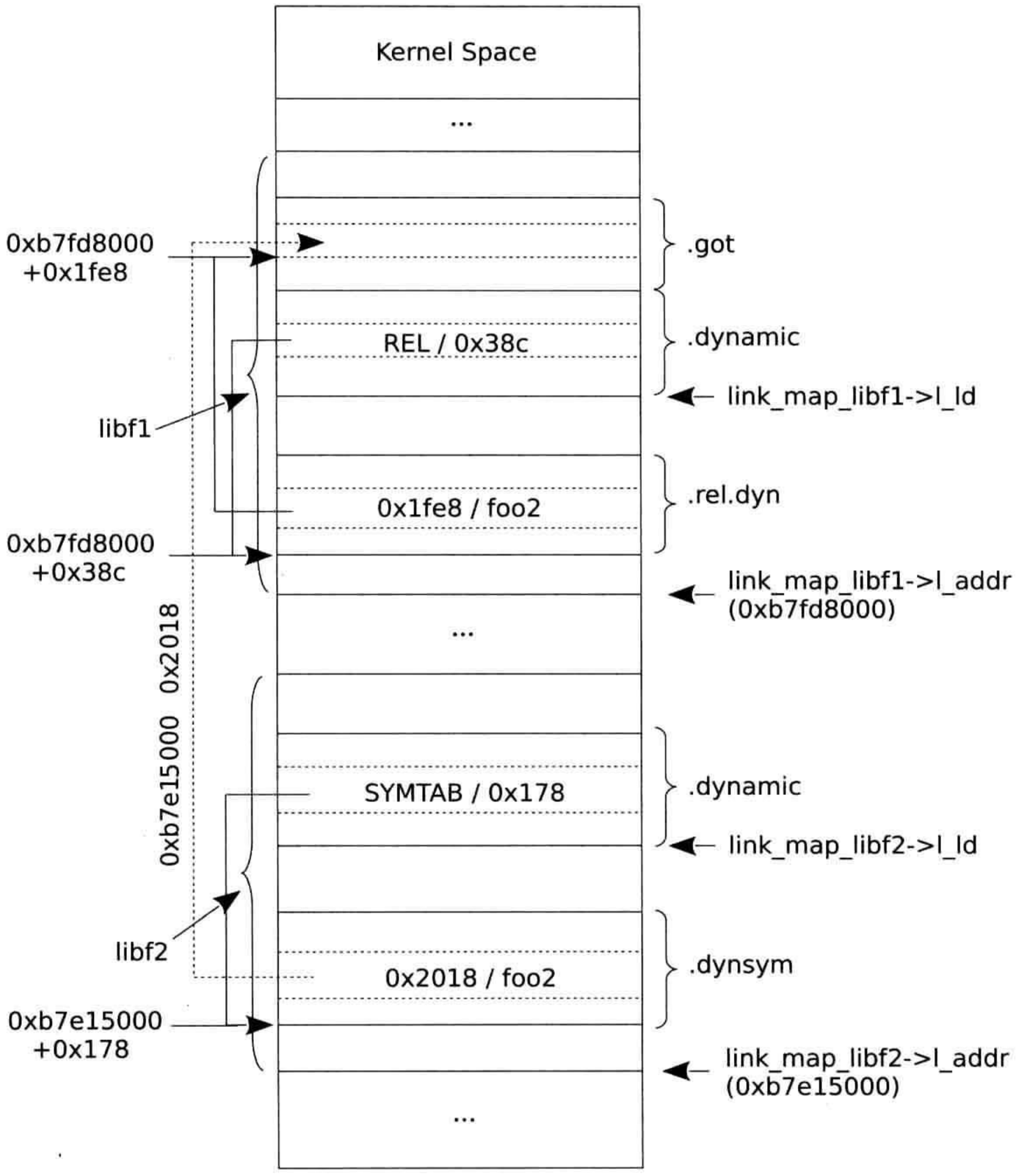


图 5-33　变量 foo2 的重定位过程

3. 重定位函数

前面我们讨论了变量的重定位，本小节我们讨论函数的重定位。理论上，函数的重定位使用与变量相同的方法即可。但是，因为相对比较少的全局变量的引用，函数引用的数量可能要大得多，因此函数重定位的时间不得不考虑。

事实上，读者回想一下我们日常开发的程序，其实很多代码不一定能全部执行，比如有些分支、错误处理等。而且，即使可执行程序本身使用的函数数量并不大，但是可执行程序依赖的动态库可能还会引用其他动态库中的函数，这些动态库再依赖其他的动态库，如此，需要重定位的函数的数量不容小觑。更重要的是，可执行程序可能根本就用不到这些动态库中的函数，因此，加载时重定位函数只会延长程序启动的时间，但是重定位的某些函数却可能根本就用不到。出于以上考虑，PIC 对于函数的重定位引入了延迟绑定技术（lazy binding）。

也就是说，在加载时，动态链接器不解析任何一个需要重定位的函数的地址，而是在运

行时真正调用时，再去重定位。为此，开发者们引入了 PLT（Procedure Linkage Table）机制。在 GOT 表的巧妙配合下，PIC 将函数地址的解析推迟到了运行时。

在编译时，链接器在代码段中插入了一个 PLT 代码片段，每个外部函数在 PLT 中都占据着一小段代码。我们可以将这些片段看作外部函数在本地代码中的代理。代码段中所有引用外部函数的地方，全部指向其相应的本地代理。其他具体的事情就交由本地代理去处理。

PLT 的代码片段的逻辑如图 5-34 所示。

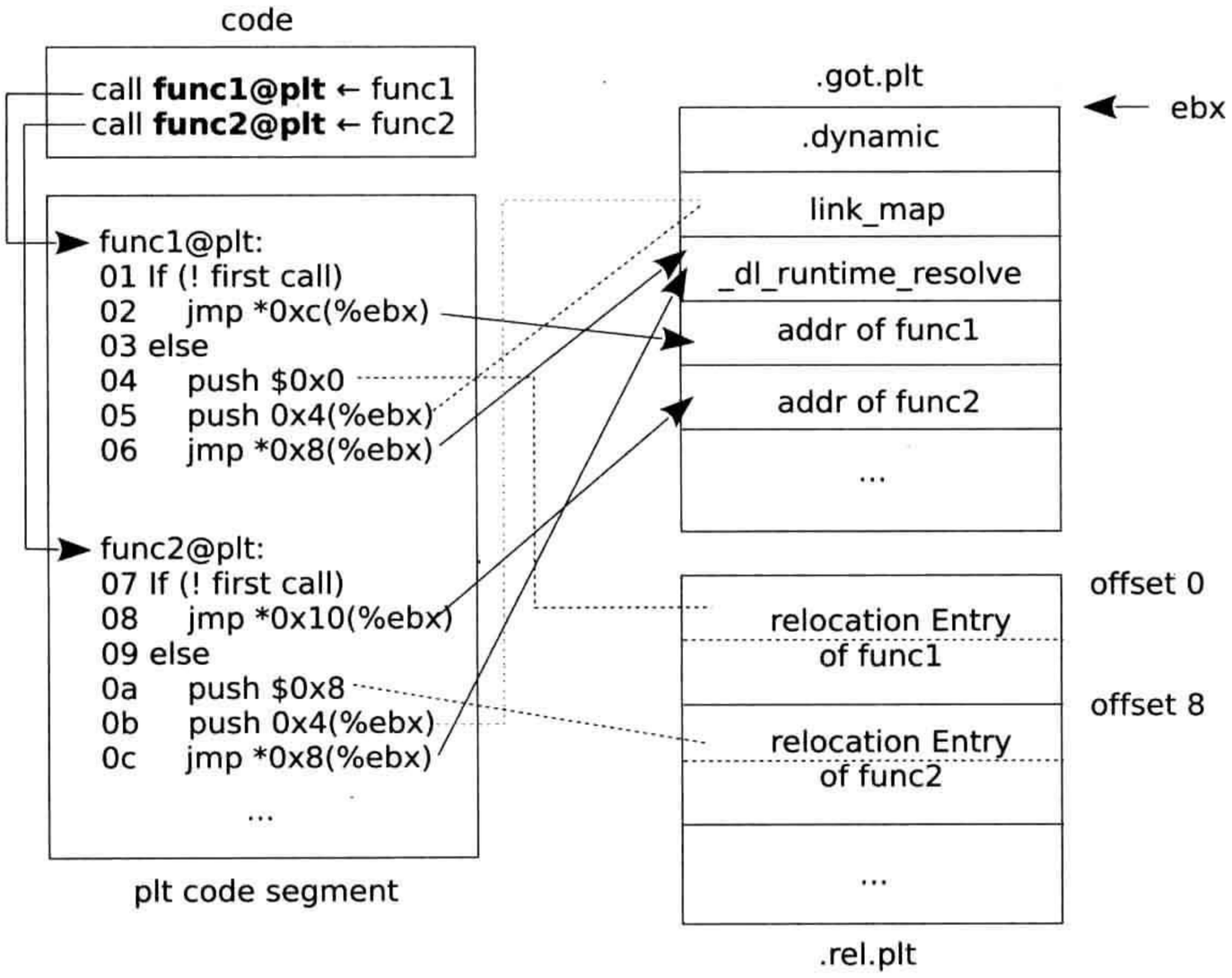


图 5-34　PLT 代码片段

由图 5-34 可见：

1）代码中所有引用函数如 func1、func2 的地方全部替换为指向 PLT 中的代码片段。因为这里使用的是相对寻址，所以运行时代码段无须再进行任何修订，也就是说，代码段不需要重定位了。保证了代码段的可读属性，从而在多个进程间可以共享。

2）PLT 中每个函数的代码片段除了两处数据外，基本完全相同。以调用函数 func1 为例，它的基本逻辑是：如果不是第一次调用 func1，就说明函数 func1 的地址已经被解析，并且 GOT 表中对应的 func1 的地址的项也已经被正确修订了，那么直接跳转到 GOT 表中对应的项即可，也就是说，这样就直接跳转到了函数 foo2 的开头。这里，因为 GOT 表的前 3 项有特殊的用途，所以 func1 的地址占据 GOT 表的第 4 项。ELF 标准规定，在调用 PLT 中的代码片段前，主调函数需要将 GOT 表的基址装载进寄存器 ebx，所以，PLT 中凡是访问 got 的地方，都使用 ebx，*0xc（%ebx）就是 GOT 表中第 4 项的值，即函数 func1 的地址。读者可以回顾一下前面讨论的重定位变量一节，那里讨论确定 GOT 的基地址时，正是将 GOT 表的地址装入了寄存器 ebx。

3）如果是第一次调用，那么将调用动态链接器提供的函数 _dl_runtime_resolve 解析函数 foo1 的地址。这里显然不能将函数 _dl_runtime_resolve 的地址直接写在 PLT 代码中，如果这样的话，那么 PLT 也需要重定位这个函数，除非使用前面提到的加载时重定位，但前面已经提到了其种种弊端。因此，动态链接器在加载库时，将函数 _dl_runtime_resolve 的地址填充到动态库的 GOT 表的第 3 项，而在 PLT 表中，则直接跳转到 GOT 表中第 3 项保存的地址，即 *0x8（%ebx）。

4）在跳转到函数 _dl_runtime_resolve 的地址前，有两条 push 指令，它们就是为函数 _dl_runtime_resolve 准备参数的。在具体看这两条直指令前，我们先来看一下修订 GOT 表中的函数地址时需要的信息：

- 第一个需要的信息是当前重定位的函数在重定位表中的偏移。根据这个偏移，_dl_runtime_resolve 找到相应的重定位条目，从而确定需要解析的符号的名字，以及需要修订的位置。对于函数在重定位表中的偏移，这个在编译时就可以确定，所以我们看到 PLT 中直接使用了确定的数字。如函数 func1 在重定位表中占据第 1 个条目，那么偏移就是 0x0，这就是汇编指令“push $0x0”的作用。而对于函数 foo2，因为其在重定位表中占据第 2 个条目，所以偏移就是 0x8。
- 第二个是需要个代表当前动态库的 link_map 对象。要获得重定位表，当然需要知道动态库映射的基址以及段 .dynamic 所在的地址，而这些信息记录在库的 link_map 对象中。在查找符号时，其需要遍历可执行程序的 link_map 链表，因此，函数 _dl_runtime_resolve 要根据动态库的 link_map 对象找到 link_map 链表。而 link_map 也是在动态链接器加载库时填充到 GOT 表中的，它占据 GOT 表的第 2 项，这就是 PLT 代码中汇编语句“push 0x4（%ebx）”的作用。

5）准备好参数后，_dl_runtime_resolve 将开始寻找符号，最后修订 GOT 表中的地址。相关代码如下：

```
glibc-2.15/sysdeps/i386/dl-trampoline.S:

_dl_runtime_resolve:
    ...
    movl 16(%esp), %edx # Copy args pushed by PLT in register.  Note
    movl 12(%esp), %eax # that `fixup' takes its parameters in regs.
    call _dl_fixup      # Call resolver.
    ...
    movl %eax, (%esp)    # Store the function address.
    ...
    ret $12           # Jump to function address.
    cfi_endproc
    .size _dl_runtime_resolve, .-_dl_runtime_resolve
```

_dl_runtime_resolve 中核心的是调用函数 _dl_fixup 进行符号解析，并修订 GOT 表。这里使用的是寄存器传参，所以 _dl_runtime_resolve 在调用 _dl_fixup 前，将动态库的 link_map 存储在寄存器 eax 中，作为传给 _dl_fixup 的第 1 个参数；将重定位函数在重定位表中的偏移

存储在寄存器 edx，作为传给 _dl_fixup 的第 2 个参数。

然后，在 _dl_fixup 执行完毕后，会将解析的函数的地址返回。这个返回值会放在寄存器 eax 中，所以我们看到 _dl_runtime_resolve 在 _dl_fixup 执行完毕后，会将保存在寄存器 eax 中的值放到栈顶，然后调用 ret 指令，将这个返回地址弹出到指令指针之中，从而跳转到解析后的地址运行。

下面我们再简要看一下解析函数地址的函数 _dl_fixup：

```
glibc-2.15/elf/dl-runtime.c:

01 #ifndef reloc_offset
02 # define reloc_offset reloc_arg
03 # define reloc_index  reloc_arg / sizeof (PLTREL)
04 #endif
05
06 __attribute ((noinline)) ARCH_FIXUP_ATTRIBUTE
07 _dl_fixup ( struct link_map *__unbounded l, ElfW(Word) reloc_arg)
08 {
09   const ElfW(Sym) *const symtab
10     = (const void *) D_PTR (l, l_info[DT_SYMTAB]);
11   const char *strtab = (const void *) D_PTR (l,
12                         l_info[DT_STRTAB]);
13
14   const PLTREL *const reloc = (const void *) (D_PTR (l,
15       l_info[DT_JMPREL]) + reloc_offset);
16   const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
17   void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
18   lookup_t result;
19   DL_FIXUP_VALUE_TYPE value;
20   ...
21       result = _dl_lookup_symbol_x (strtab + sym->st_name, l,
22             &sym,l->l_scope, version, ELF_RTYPE_CLASS_PLT, ...);
23       ...
24       value = DL_FIXUP_MAKE_VALUE (result, ...);
25   ...
26   return elf_machine_fixup_plt (l, result, reloc, rel_addr,
27             value);
28 }
29
30 glibc-2.15/sysdeps/i386/dl-machine.h:
31
32 static inline Elf32_Addr
33 elf_machine_fixup_plt (struct link_map *map, lookup_t t,
34     const Elf32_Rel *reloc, Elf32_Addr *reloc_addr,
35     Elf32_Addr value)
36 {
37   return *reloc_addr = value;
38 }
```

先看函数 _dl_fixup 的两个参数，第一个参数 l 就是传递进来的动态库的 link_map；第 2 个参数 reloc_arg 就是重定位表的偏移，根据第 2 行代码的宏定义可见，函数体中使用的变量 reloc_offset 就是 reloc_arg。

代码第 9~12 行根据传递来的 link_map，首先取得动态库的动态符号表，包括 SYMTAB 和 STRTAB。

代码第 14~15 行根据传进来的函数在重定位表中的偏移，从重定位表中获取对应的重定位记录 reloc。

第 16 行代码根据重定位记录 reloc 中符号在动态符号表中的索引，从动态符号表 symtab 中取出符号的名字。

第 17 行代码根据重定位记录 reloc 中的记录的偏移，加上库映射的基址，计算出需要修订的位置。当然这个位置对应的是 GOT 表中的某一项。

代码第 21~22 行调用 _dl_lookup_symbol_x 遍历 link_map 链表，查找符号的地址。

代码第 26~27 行调用 elf_machine_fixup_plt 修订 GOT 表中对应的项，函数 elf_machine_fixup_plt 中就一条代码，如代码第 37 行，就是给 GOT 表的某一项赋个符号地址而已。

理论上，函数的重定位过程可以就此完成了。但是，上述方法还有些瑕疵：

- 在 PLT 代码片段中，需要设计标志来表示函数是否是第一次调用。
- 在 PLT 代码片段中，编译器的实现者们不想做那个多余的 if 判断，即函数是否是第一次调用的判断。尽管这可能只是一次跳转和一次访存，但是编译器的实现者们还是想把它们节省下来。

于是，编译器的设计者们在上述基础上，做出了更进一步的改进，如图 5-35 所示。

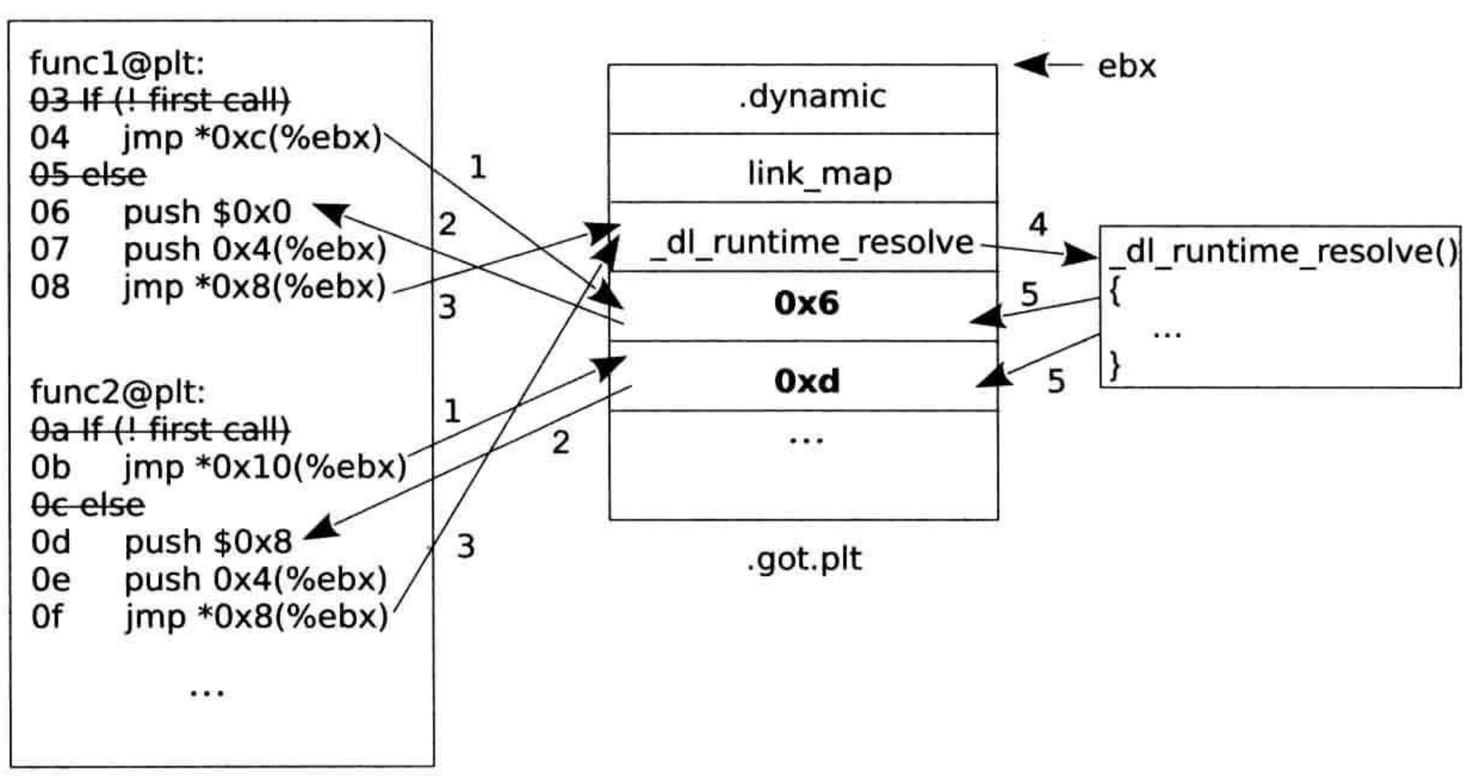


图 5-35 PLT 代码片段

我们看到，PLT 中的代码片段不再进行任何判断，而是直接跳转到 GOT 表中用来保存解析的函数的地址的表项。这里面最关键的一个技巧就是图 5-35 中用黑体标识的 GOT 表中的两项。编译时，编译器将函数对应的项的地址初始化为 PLT 代码片段中 jmp 语句的下一条地址。在动态库加载时，动态器会在此基础上，再加上动态库的映射的基址。如此，当第一次执行这个函数时，jmp 语句并没有跳转到真正的函数的地址处，而是直接相当于执行 PLT 代码片段中的下一条语句，即压栈参数，然后调用 _dl_runtime_resolve 解析函数地址，使用解

析的符号的地址修订 GOT 表中的项，然后跳转到解析的函数的地址，执行函数。

这里不知是否有读者有过这样的设想：程序加载时，将函数的 GOT 表项直接填写为函数 _dl_runtime_resolve 的地址，是不是更合理？非也，GOT 表一项只有 4 字节，只能保存一个地址，而调用 _dl_runtime_resolve 之前，还需要其他指令准备参数。

经过第一次调用后，GOT 表中的函数对应的项已经变为真正的函数的地址，下次再次调用时，将直接跳转到函数的地址继续执行，如图 5-36 所示。

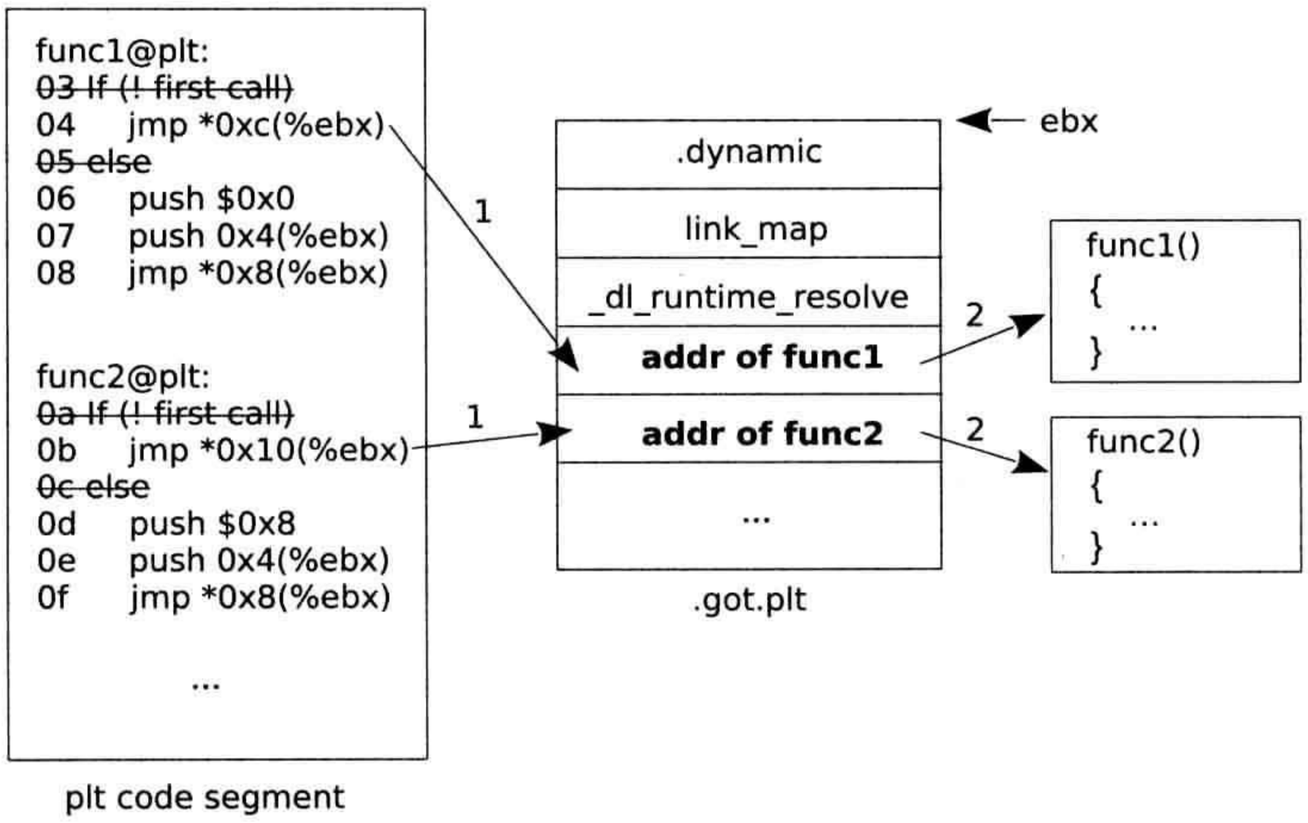


图 5-36　PLT 代码片段

观察图 5-36 会发现，PLT 中 func1@plt 中的地址为 0x7 和 0x8 处两行的代码，以及 func2@plt 中地址 0xe 和 0xf 处的代码完全一样。事实上，所有函数的 PLT 片段的最后两行都完全相同。于是，PLT 将这两行代码独立为一个“子函数”plt0。进一步改进后 PLT 的代码如图 5-37 所示。

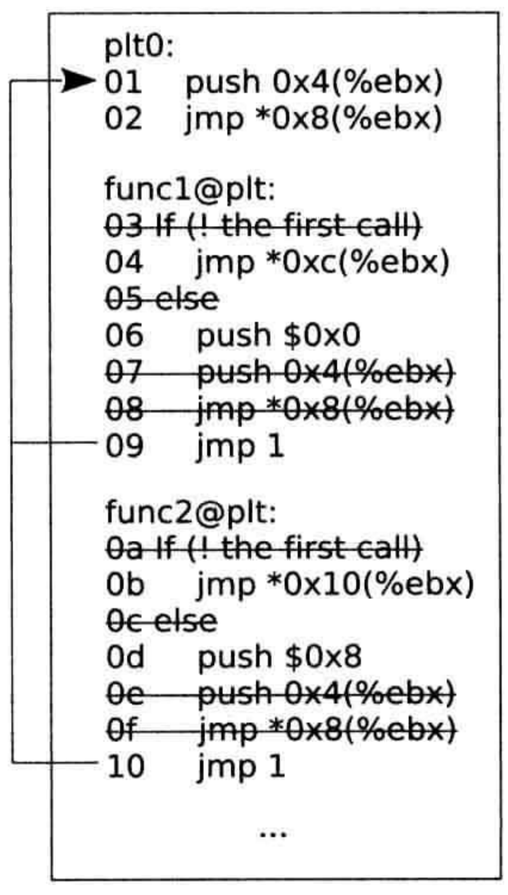


图 5-37　PLT 代码片段

下面我们以库 libf1 中的函数 foo1_func 调用库 libf2 中的函数 foo2_func 为例，来具体体会一下前面的理论分析。反汇编库 libfoo2，并截取引用函数 foo2_func 的有关部分：

```
root@baisheng:~/demo# objdump -d libf1.so

Disassembly of section .plt:

00000420 <__cxa_finalize@plt-0x10>:
 420:      ff b3 04 00 00 00      pushl  0x4(%ebx)
 426:      ff a3 08 00 00 00      jmp    *0x8(%ebx)
...
00000450 <foo2_func@plt>:
 450:      ff a3 14 00 00 00      jmp    *0x14(%ebx)
 456:      68 10 00 00 00         push   $0x10
 45b:      e9 c0 ff ff ff         jmp    420 <_init+0x24>

Disassembly of section .text:

00000460 <deregister_tm_clones>:
 460:      55                     push   %ebp
...
00000580 <foo1_func>:
...
5b3:      e8 98 fe ff ff         call   450 <foo2_func@plt>
5b8:      83 c4 14               add    $0x14,%esp
...
```

先来看地址 0x5b3 处的指令。汇编指令 call 的操作数 0xfffffe98（补码）对应的原码是 -0x168，call 指令的操作数是一个相对寻址，因此 -0x168 是目标地址和下一条指令的差值。因为下一条指令的地址是 0x5b8，所以跳转的目的地址是：

```
0x5b8 - 0x168 = 0x450
```

地址 0x450 处正是 PLT 中对应函数 foo2_func 的片段。我们看到地址 0x450 处的汇编指令跳转到 GOT 表中偏移为 0x14 处中的值表示的地址处。那么 GOT 表中这个位置处保存的是什么呢？我们需要到记录函数重定位的表——.rel.plt 中寻找答案：

```
root@baisheng:~/demo# readelf -r libf1.so

Relocation section '.rel.plt' at offset 0x3e4 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000200c  00000307 R_386_JUMP_SLOT   00000000   __cxa_finalize
00002010  00000407 R_386_JUMP_SLOT   00000000   __gmon_start__
00002014  00000607 R_386_JUMP_SLOT   00000000   foo2_func
```

动态库 libf1 的 GOT 表的基址为 0x2000，所以偏移 0x14 处的地址即为 0x2014，也就是重定位表中的第 3 条记录。可见，这条重定位记录要求动态链接器使用符号 foo2_func 的值填充地址为 0x2014 处的 GOT 表项。根据前面的理论分析，初始时，这个地址指向下一条 push 指令，即地址 0x456 处的指令。所以，当首次调用 foo2_func 时，地址 0x450 处的指令跳转到了地址 0x456 处。

地址 0x456 处的指令压栈了一个立即数 0x10。根据前面的理论分析，这是为符号解析函数 _dl_runtime_resolve 压栈的一个参数，即需要重定位的函数在重定位表中的偏移。根据重定位表中的信息，函数 _dl_runtime_resolve 就可以找到与重定位函数相关的信息，如重定位函数的符号名称、需要修订的位置等。0x10 用十进制表示是 16，也就是从重定位表 .rel.plt 开始偏移 16 字节，重定位表中每个条目占据 8 字节，因此偏移 16 字节处的第 3 条重定位记录正是记录函数 f002-func 的重定位信息。

继续看下一条指令，即地址 0x45b 处的指令。也是一条相对跳转指令，补码 0xffffffc0 的原码是 -0x40，所以跳转的目的地址是：

```
0x460 - 0x40 = 0x420
```

objdump 工具虽然显示地址 0x420 处的函数的名字是“__cxa_finalize@plt-0x10”，实际上与函数“__cxa_finalize”没有任何关系，这里解析的有一点 bug，忽略即可。地址 0x420 处就是 PLT 表的第 0 项。我们看到 plt0 首先将 GOT 表中偏移 0x4 处，即 GOT 表第 2 项的值（库 libf1 的 link_map）压栈，显然是给解析函数传参。然后跳转到 GOT 表的偏移 0x8 处，即第 3 项，也就是解析函数 _dl_runtime_resolve 的地址处执行，该函数解析符号 foo2_func，然后使用解析得到的符号 f002-func 的运行时地址修订 GOT 表中偏移 0x14 处，即第 6 项，然后跳转到函数 foo2_func 执行。

首次调用函数 foo2_func 后，GOT 表中第 6 项保存的就是 foo2_func 的地址了。以后再次调用该函数时，PLT 中的 foo2_func@plt 将不再跳转到函数 _dl_runtime_resolve 处解析函数了，而是直接跳转到函数 foo2_func 处。

在静态分析后，下面我们再动态观察一下函数 foo2_func 的重定位过程。

我们首先来看一下编译时库 libf1 的 GOT 表中第 6 项，即偏移 0x2014 处，保存的内容是什么，前面我们已经讨论过了，理论上这里应该是 foo2_func@plt 中 push 指令的地址：

```
root@baisheng:~/demo# readelf -r libf1.so

Relocation section '.rel.plt' at offset 0x3e4 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
...
00002014  00000607 R_386_JUMP_SLOT   00000000   foo2_func

root@baisheng:~/demo# objdump -D -j .got.plt libf1.so

Disassembly of section .got.plt:

00002000 <_GLOBAL_OFFSET_TABLE_>:
    ...
    2014:  56                      push   %esi
    2015:  04 00                   add    $0x0,%al
              ...

root@baisheng:~/demo# objdump -d -j .plt libf1.so
...
```

```
00000450 <foo2_func@plt>:
 450:      ff a3 14 00 00 00      jmp    *0x14(%ebx)
 456:      68 10 00 00 00         push   $0x10
 45b:      e9 c0 ff ff ff         jmp    420 <_init+0x24>
```

注意上面使用黑体标识的部分，编译时偏移 0x2014 处的 4 字节初始化为 0x0456，正是 foo2_func@plt 中 push 指令的地址。

我们将 hello 运行起来，观察一下 GOT 表中第 6 项的变化情况：

```
root@baisheng:~/demo# gdb ./hello
 (gdb) b foo1_func
Breakpoint 1 at 0x80484d0
 (gdb) r
Starting program: /root/demo/hello

Breakpoint 1, 0xb7fd8584 in foo1_func () from libf1.so
```

我们在另外一个终端中查看库 libf1 在进程 hello 的地址空间中映射的基址：

```
root@baisheng:~# ps -C hello -o pid=
 3122
root@baisheng:~# cat /proc/3122/maps
08048000-08049000 r-xp 00000000 08:01 1054223    /root/demo/hello
...
b7fd8000-b7fd9000 r-xp 00000000 08:01 1047105 /root/demo/libf1.so
...
```

根据输出可见库 libf1 在进程 hello 的地址空间中映射基址是 0xb7fd8000。虽然说函数 foo2_func 的地址是在使用时再去重定位，但是加载时动态链接器还是要做一个重定位。读者不禁要问，重定位什么呢？我们以 GOT 表的第 6 项，即偏移 0x2014 处的值为例。在编译时，我们看到链接器将此处的地址填充为 0x0456，即 jmp 后的 push 指令的地址。但是不知读者是否注意到，这个地址是相对于 0 的地址，在加载后，当动态库 libf1 的映射基址确定为 0xb7fd8000 后，显然需要修订这个地址为：

```
0xb7fd8000 + 0x456 = 0xb7fd8456
```

我们通过 gdb 看一下实际的输出：

```
(gdb) x 0xb7fd8000 + 0x2014
0xb7fda014:        0xb7fd8456
```

可见，GOT 表中的这一项在加载时确实修订了。

在 foo2_func 第一次执行后，这个 GOT 表中的地址就应该修订为 foo2_func 的地址，我们看一下库 libf2 中为 foo2_func 分配的地址：

```
root@baisheng:~/demo# readelf -s libf2.so

Symbol table '.dynsym' contains 13 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     ...
```

```
     8: 00000500     5 FUNC    GLOBAL DEFAULT   11 foo2_func
     ...
```

而动态库 libf2 在进程 hello 的地址空间中映射的基址是：

```
root@baisheng:~/demo# ps -C hello -o pid=
 3122
root@baisheng:~/demo# cat /proc/3122/maps
08048000-08049000 r-xp 00000000 08:01 1054223    /root/demo/hello
...
b7e15000-b7e16000 r-xp 00000000 08:01 1054350 /root/demo/libf2.so
...
```

所以，符号 foo2_func 的运行时地址是：

```
0xb7e15000 + 0x500 = 0xb7e15500
```

我们通过 gdb 来查看一下 foo2_func 执行一次后，GOT 表中的保存这个函数的地址被修订成了什么：

```
(gdb) n
0x080485f4 in main ()
(gdb) x 0xb7fd8000 + 0x2014
0xb7fda014:        0xb7e15500
```

可见，在首次调用后，GOT 表中的值已经修订为符号 foo2_func 的运行时地址。

5.4.7 重定位可执行程序

可执行程序如果引用的是自身定义的函数和变量，这些符号在编译时就已经确定，不需要任何重定位。即使其他动态库中也定义了与可执行程序中相同的符号，链接器也优先使用可执行程序自身定义的函数和变量。

如果引用了动态库中的函数和全局变量，那么编译时可执行程序根本不知道这些符号最终的地址，在重定位了动态库之后，可执行程序也需要重定位这些符号。可执行程序的重定位与共享库原理基本一致，只有一点差别，我们这里简单讨论一下它们之间的差别。

（1）重定位引用的动态库中的函数

我们以 hello 中引用动态库 libf1 中的函数 foo1_func 为例，来看关于函数的重定位。可执行程序 hello 中调用 foo1_func 的反汇编代码如下：

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

080485cc <main>:
...
80485ef:   e8 dc fe ff ff         call   80484d0 <foo1_func@plt>
...
```

可见，可执行程序也使用了延迟绑定的技术。再来看看 PLT 部分的代码：

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

```
Disassembly of section .plt:

08048480 <sleep@plt-0x10>:
 8048480:  ff 35 04 a0 04 08      pushl  0x804a004
 8048486:  ff 25 08 a0 04 08      jmp    *0x804a008
...
080484d0 <foo1_func@plt>:
 80484d0:  ff 25 1c a0 04 08      jmp    *0x804a01c
 80484d6:  68 20 00 00 00         push   $0x20
 80484db:  e9 a0 ff ff ff         jmp    8048480 <_init+0x28>
```

与动态库不同，可执行程序的地址在编译时就已经分配好了，所以，GOT 的地址在编译时就确定了，不必再如动态那样在运行时动态获取 GOT 表的基址。我们来看看 hello 的 GOT 表的基址：

```
root@baisheng:~/demo# readelf -s hello | grep OFFSET_TABLE
44: 0804a000   0 OBJECT  LOCAL  DEFAULT  23 _GLOBAL_OFFSET_TABLE_
```

GOT 表的基址为 0x0804a000，所以任何以 GOT 表基址为参照的偏移，直接使用这个地址即可。比如访问 GOT 表中的第 3 项，即函数 _dl_runtime_resolve 时，直接在此地址上加两个 4 字节偏移即可（因为 _dl_runtime_resolve 占据 GOT 表的第 3 项，所以偏移 8 字节）：

```
0x0804a000 + 0x4*2 = 0x0804a008
```

观察 hello 中 plt0 部分，即地址 0x8048486 处，我们看到，指令中也确实是这么做的，jmp 的目标地址在编译时就计算好了，就是 *0x804a008。

除 GOT 表的基址固定外，可执行程序函数的重定位与动态库中函数的重定位完全一致。

（2）重定位引用的动态库中的变量

可执行程序与动态库不同，一般而言，其地址是编译时分配好的，是固定的（这里我们不考虑为了安全而使用 PIE 技术）。如果编译时没有传给编译器参数“-fPIC”，那么对于引用的外部的全局变量，可执行程序不使用 GOT 表的方式寻址。换句话说，可执行程序引用的变量，在编译链接时就需要在编译链接时确定好地址，不能在加载时再进行重定位。

但是，编译时动态库都不能确定自己的变量的最终加载地址，更别提可执行程序了。那怎么办呢？于是 ELF 标准定义了一种新的重定位类型——R_386_COPY。对于这种重定位类型，编译器、链接器和动态链接器是这样协作的：编译时，编译器将偷偷地在可执行程序的 BSS 段创建了一个变量，这样就解决了编译时，变量地址不确定的问题。在程序加载时，动态链接器将动态库中的变量的初值复制到可执行程序的 BSS 段中来。然后，动态库（包括其他动态库）在引用这个变量时，因为可执行程序在 link_map 的最前面，所以解析符号都将使用可执行程序中的这个偷偷创建的变量。

下面我们结合 hello 引用动态库 libf1 中的变量 foo1 来具体的讨论一下。先来看一下 hello 的动态符号表：

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

```
Symbol table '.dynsym' contains 17 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     ...
     12: 0804a040     4 OBJECT  GLOBAL DEFAULT   25 foo1
     ...
```

虽然我们没有在可执行程序中定义变量 foo1，但是根据动态符号表可见，可执行程序 hello 中却定义了变量 foo1，其所在地址是 0x0804a028，而且在第 25 个段中。我们来看看第 25 个段是什么：

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

Section Headers:
  [Nr] Name              Type            Addr     Off    Size
  ...
  [25] .bss              NOBITS          0804a040 00102c 002020
  ...
```

可见，第 25 个段是 .bss。也就是说，编译时，链接器为可执行程序 hello 定义了一个未初始化的全局变量 foo1。而 hello 中，使用的恰恰是 hello 自己的 foo1，而不是库 libf1 中的 foo1。观察下面中引用的符号 foo1 的地址，正是 hello 中定义的符号 foo1 的地址：

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

080485cc <main>:
...
80485e5:   c7 05 40 a0 04 08 05   movl   $0x5,0x804a040
...
```

链接器将 hello 的重定位表中 foo1 的重定位类型设置为 R_386_COPY，当处理这个类型的重定位时，动态链接器将在加载时，将库 libf1 中变量 foo1 的值复制到 hello 中的 foo1：

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

Relocation section '.rel.dyn' at offset 0x420 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000406 R_386_GLOB_DAT    00000000   __gmon_start__
0804a040  00000c05 R_386_COPY        0804a040   foo1
```

下面我们将程序运行起来，动态观察一下 R_386_COPY 类型的重定位过程。

```
root@baisheng:~/demo# gdb ./hello
(gdb) b main
Breakpoint 1 at 0x80485cf
 (gdb) r
Starting program: /root/demo/hello

Breakpoint 1, 0x080485cf in main ()
```

理论上，动态链接器应该将库 libf1 中的 foo1 的初值 10 复制到 hello 中定义的 foo1 处。我们将 hello 中定义变量 foo1 所在地址实际的值打印出来：

```
(gdb) x 0x0804a040
0x804a040 <foo1>: 0x0000000a
```

可见，hello 中的 foo1 已经被赋值为库 libf1 中的 foo1 的初值 10 了。

另外，库 libf1 中 GOT 表中保存的 foo1 的地址，也应该指向 hello 中定义的 foo1 的地址，而不是库 libf1 中的变量 foo1 的地址。原因是链接时，可执行程序排在链表 link_map 的表头，所以 hello 中的符号 foo1 当然要优先于库 libf1 中的 foo1。我们来实际验证一下这一点，首先找到库 libf1 中变量 foo1 所在位置：

```
root@baisheng:~/demo# readelf -r libf1.so

Relocation section '.rel.dyn' at offset 0x38c contains 11 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
 ...
00001ff8  00000c06 R_386_GLOB_DAT    0000201c   foo1
...
```

在另外一个终端中查看库 libf1 在进程 hello 的地址空间中映射的基址：

```
root@baisheng:~/demo# ps -C hello -o pid=
 3346
root@baisheng:~/demo# cat /proc/3346/maps
08048000-08049000 r-xp 00000000 08:01 1054223    /root/demo/hello
...
b7fd8000-b7fd9000 r-xp 00000000 08:01 1047105 /root/demo/libf1.so
...
```

库 libf1 的 GOT 表中记录符号 foo1 的地址是：

```
0xb7fd8000 + 0x1ff8 = 0xb7fd9ff8
```

我们打印一下 GOT 表中的值：

```
(gdb) x 0xb7fd9ff8
0xb7fd9ff8:     0x0804a040
```

根据输出可见，地址 0x0804a040 正是 hello 中定义的符号 foo1 的地址。可见，动态库 libf1 中使用的 foo1 变量是可执行程序中创建的这个副本。显然，虽然这个副本仅仅是编译器为其偷偷分配的，但是实际已经取代了库 libf1 中的 foo1，已经转正了。

当然，在编译可执行程序时也可以给其传递参数“-fPIC”，如此，可执行程序中对外部变量的应用也将采用 GOT 表的方式，但是这对可执行程序没有任何意义。

5.4.8 重定位动态链接器

在 Linux 中，动态链接器被实现为一个动态库的形式，而且这个动态库是自包含的（self-contained），没有引用其他库的符号，但是与普通动态库一样的道理，它在编译时也不知道自己的确切位置，所以它也难逃重定位的命运。事实上，当 C 库加载后，动态链接使用了 C 库中的内存管理相关函数替换了自身的实现。

查看一下动态链接器的重定位表就可见其需要重定位的符号：

```
root@baisheng:/vita/sysroot/lib# readelf -r ld-2.15.so

Relocation section '.rel.dyn' at offset 0x6d0 contains 13 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00020e48  00000008 R_386_RELATIVE
...
00020ffc  00000606 R_386_GLOB_DAT    00015780   free

Relocation section '.rel.plt' at offset 0x738 contains 6 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0002100c  00000b07 R_386_JUMP_SLOT   000155e0   __libc_memalign
00021010  00001607 R_386_JUMP_SLOT   000156f0   malloc
00021014  00000d07 R_386_JUMP_SLOT   00015720   calloc
00021018  00000707 R_386_JUMP_SLOT   00015860   realloc
0002101c  00001207 R_386_JUMP_SLOT   00011f50   ___tls_get_addr
00021020  00000607 R_386_JUMP_SLOT   00015780   free
```

但是，与动态库和可执行程序不同，它们有动态链接器负责为它们重定位，而动态链接器则没有这么好的命。在内核跳转到动态链接器时，它是非常残酷的，并没有给动态链接器如 link_map 信息。好在动态链接器不依赖其他的动态库，只需要确定自己被加载的基地址，然后找到动态链接需要的段 .dynamic 就可以解决问题，后续的重定位过程与动态库的过程基本完全相同。因此，动态链接器重定位自己的关键是：

- 确定自己被加载的基地址；
- 找到段 .dynamic。

动态链接器被加载的地址就相当于 link_map 中的 l_addr 了。运行后，动态链接器可以获取到某个符号的地址，但是这并不足以计算出动态链接器在进程地址空间中映射的基址，只有对比，才能求出基址。因此，动态链接器还是需要编译时的链接器作一点小小的配合。在编译时，链接器定义了一个符号“_DYNAMIC”：

```
root@baisheng:/vita/sysroot/lib# readelf -s ld-2.15.so
Symbol table '.symtab' contains 584 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
    ...
    353: 00020f3c     0 OBJECT  LOCAL  DEFAULT   15 _DYNAMIC
   ...
   511: 00021000  0 OBJECT  LOCAL DEFAULT 17 _GLOBAL_OFFSET_TABLE_
   ...
```

定义这个符号的目的就是为了标识段 .dynamic 所在的地址，看下面动态链接器的 Section Header Table：

```
root@baisheng:/vita/sysroot/lib# readelf -S ld-2.15.so

Section Headers:
  [Nr] Name              Type            Addr     Off     Size
  ...
  [15] .dynamic          DYNAMIC         00020f3c 01ff3c 0000b8
```

...

由上可见，符号 _DYNAMIC 的地址正是段 .dynamic 的地址。在运行时，动态链接器使用如 x86 指令 lea 读取符号 _DYNAMIC 的运行时地址，实际就是读取运行时段 .dynamic 的地址。

除了定义了这个符号外，在编译时，段 .dynamic 的地址也被装载到了 GOT 表中的第 1 项。读者回忆一下在 5.4.6 节讨论 GOT 表时的内容。其中，第 2 项的 link_map 和第 3 项的解析函数我们都已经看到其作用了，但是尚未看到第 1 项的意义。在重定位动态链接器时，这一项发挥了关键作用。前面我们就已经看到过，编译时定义了另外一个符号 _GLOBAL_OFFSET_TABLE_，目的与 _DYNAMIC 相似，是为了标识 GOT 表的地址。因此，动态链接器就可以使用符号 _GLOBAL_OFFSET_TABLE_ 找到 GOT 表，从而取出 GOT 表中第 1 项的值。

然后，使用取得的符号 _DYNAMIC，也就是段 .dynamic 的运行时地址，与 GOT 表第一项在编译时保存的段 .dynamic 的地址（其是相对于 0 的）做差，得出的就是动态链接器在进程地址空间映射的基址了。相关代码如下：

```
glibc-2.15/elf/rtld.c:

static ElfW(Addr) __attribute_used__ internal_function
_dl_start (void *arg)
{
  ...
  bootstrap_map.l_addr = elf_machine_load_address ();

  /* Read our own dynamic section and fill in the info array.  */
  bootstrap_map.l_ld = (void *) bootstrap_map.l_addr +
                               elf_machine_dynamic ();
  elf_get_dynamic_info (&bootstrap_map, NULL);
  ...
}
```

注意变量 bootstrap_map，相信从名字读者已经猜出来了，相当于代表普通动态库和执行程序的 link_map。而且根据这个变量的名字，我们也可以揣摩到开发者的用意是在表达这是动态链接器的自举过程。变量 bootstrap_map 中的关键两项读者应该非常熟悉了，l_addr 是代表动态链接器自己被映射的地址，l_ld 代表动态链接器的段 .dynamic 所在的地址。找到段 .dynamic 后，动态链接器调用 elf_get_dynamic_info 读取了这个段的信息。

我们来看看获取 l_addr 和 l_ld 这两个地址的函数：

```
glibc-2.15/sysdeps/i386/dl-machine.h:

static inline Elf32_Addr __attribute__ ((unused, const))
elf_machine_dynamic (void)
{
  extern const Elf32_Addr _GLOBAL_OFFSET_TABLE_[]
                                        attribute_hidden;
  return _GLOBAL_OFFSET_TABLE_[0];
}
```

```
static inline Elf32_Addr __attribute__ ((unused))
elf_machine_load_address (void)
{
  extern Elf32_Dyn bygotoff[] asm ("_DYNAMIC") attribute_hidden;
  return (Elf32_Addr) &bygotoff - elf_machine_dynamic ();
}
```

函数 elf_machine_dynamic 利用在编译时定义的符号 _GLOBAL_OFFSET_TABLE_ 读取 GOT 表中第 0 项的值。

函数 elf_machine_load_address 计算动态链接器加载的地址。其首先取得符号 _DYNAMIC 的运行时地址，对于 x86 来说，可以使用指令 lea，然后与 GOT 表中保存的编译时的地址做差，从而得出动态库在进程地址空间中映射的基址。

事实上，动态连接器重定位表中的那些动态内存管理的函数，如 malloc、free 等，最初动态链接器使用的是自己内部的实现：

```
glibc-2.15/elf/dl-minimal.c:

...
void * weak_function malloc (size_t n)
{
  ...
}

void weak_function free (void *ptr)
{
  ...
}
...
```

但是一旦 C 库加载后，动态链接器将再次重定位这几个函数，使用 C 库中的相应实现。

5.4.9 段 RELRO

最初，编译时链接器并没有过多考虑 ELF 文件中各个段的布局，一个 ELF 文件各个段的大致布局如图 5-38 所示。

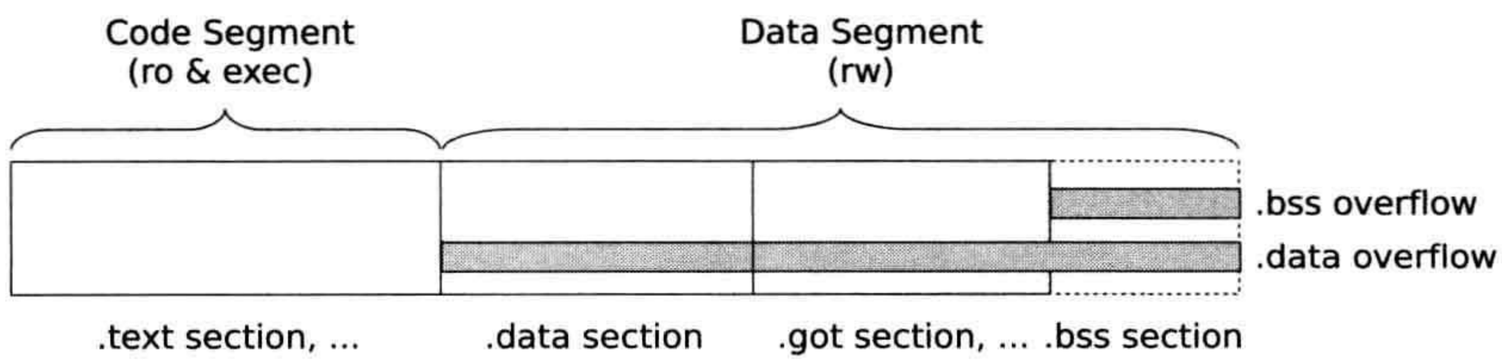


图 5-38 早期 ELF 文件段的布局

可见，动态链接器重定位涉及的 GOT 表、段 .dynamic 都位于数据段的后面，一旦数据段发生溢出，动态链接器使用的 GOT 表、段 .dynamic 都可能受到破坏，尤其是作为函数跳

转表的 GOT 表，更容易被攻击者利用。而事实上，除了函数被延迟到运行时重定位外，变量等的重定位在加载时就已经完成了，后续动态链接器不再会对这些段进行写操作，也就是说完全可以在完成加载时重定位后，把这部分数据修改为只读。

因此，如今的链接器重新安排了各个段的布局，将动态链接器涉及到的段提到了数据段的前面，并将 GOT 拆分为两个部分：.got 和 .got.plt。.got 部分用于记录需要重定位的变量，.got.plt 部分用于记录需要重定位的函数。在加载时完成重定位后，除了 .got.plt 仍然保留可写属性，允许在运行时进行重定位外，包括 .got 在内的其余部分全部更改为只读，减少被攻击的可能。

这些在重定位后更改为只读的段被称为 RELRO 段。从 Program Header Table 的角度看，段 RELRO 仍然包含于数据段中，只不过是数据段开头部分一块只读的数据而已。经过上述调整后，一个 ELF 文件的大致布局演化为如图 5-39 所示的形式。

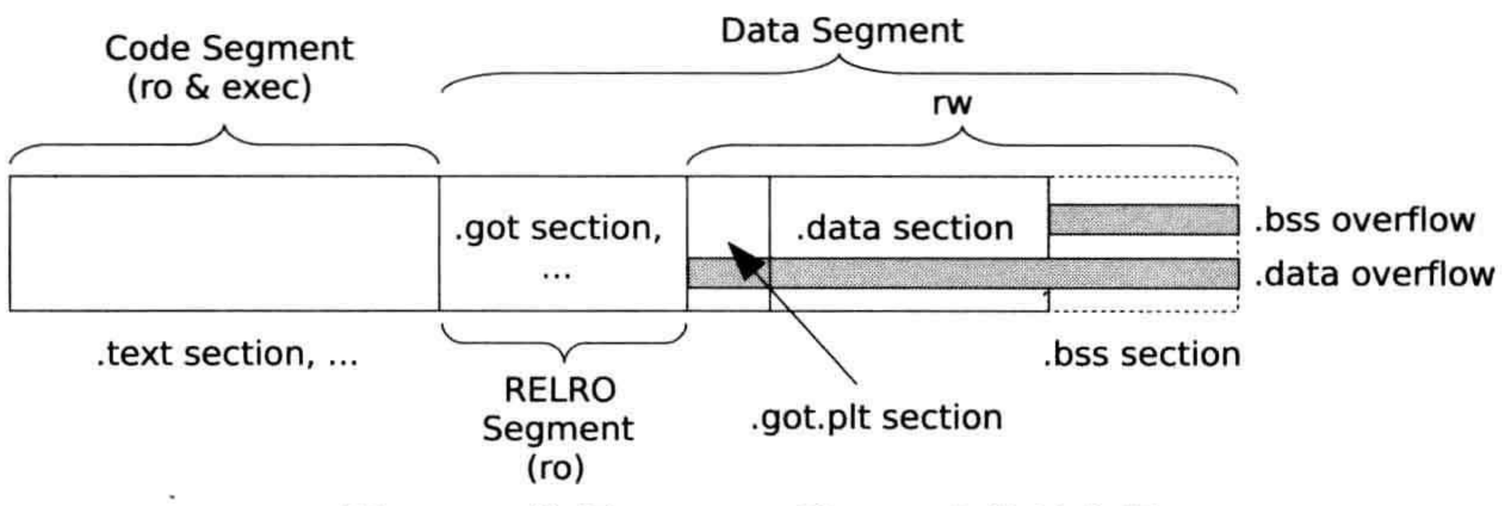


图 5-39 使用 RELRO 后 ELF 文件的布局

在加载时完成重定位后，动态链接器将检查 ELF 文件的 Program Header Table 中是否存在段 RELRO。如果这个段存在，则将这个段更改为只读，从而达到保护更多数据的目的。相关代码如下：

```
glibc-2.15/elf/dl-reloc.c:

void _dl_relocate_object (struct link_map *l, ...)
{
  ...
  if (l->l_relro_size != 0)
    _dl_protect_relro (l);
}

void internal_function _dl_protect_relro (struct link_map *l)
{
  ...
  if (start != end
      && __mprotect ((void *) start, end - start, PROT_READ) < 0)
    ...
}
```

其中 _dl_relocate_object 就是动态链接中负责加载时重定位的函数。在这个函数的最后，也就是加载时重定位完成后，这个函数调用 _dl_protect_relro 修改段 RELRO 的权限为只读。

函数 _dl_protect_relro 逻辑非常简单，就是通过函数 __mprotect 请求内核更改段 RELRO 的属性为 PROT_READ。

编译时链接器并没有强制使用 RELRO 这个特性，如果需要使用这个特性，在链接时需要向链接器传递参数“-z relro”。以笔者使用的 Ubuntu12.10 为例，可以看到在编译时编译器确实给链接器传递了这个参数，注意下面使用黑体标识的部分：

```
root@baisheng:~# gcc -dumpspecs
...
*link_command:
%{!fsyntax-only:%{!c:%{!M:%{!MM:%{!E:%{!S: %(linker) ...-z relro ...
```

在我们构建的工具链中，为简单起见，并没有默认启用 RELRO 特性。

理解了 RELRO 的设计动机以及理论背景后，我们结合一个实例来具体体验一下这个特性。以下面程序为例：

```
hello.c:

#include <stdlib.h>

void main()
{
    while(1) sleep(1000);
}
```

我们使用如下命令分别编译不支持 RELRO 特性和支持 RELRO 特性的两个可执行程序：

```
vita@baisheng:~$ i686-none-linux-gnu-gcc -o hello hello.c
vita@baisheng:~$ i686-none-linux-gnu-gcc -Wl,-z,relro \
                    -o hello_relro hello.c
```

其中，hello 是不支持 RELRO 特性的，hello_relro 是支持 RELRO 特性的。我们首先对比一下这两个程序的 Program Header Table，hello 的 Program Header Table 如下：

```
vita@baisheng:~$ readelf -l hello

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100
  ...
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000
```

hello_relro 的 Program Header Table 如下：

```
vita@baisheng:~$ readelf -l hello_relro

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
  ...
  LOAD           0x000000 0x08048000 0x08048000 0x00608 0x00608
  LOAD           0x000f20 0x08049f20 0x08049f20 0x00100 0x00108
  ...
```

```
 GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000
 GNU_RELRO      0x000f20 0x08049f20 0x08049f20 0x000e0 0x000e0

Section to Segment mapping:
 Segment Sections...
  ...
  08     .ctors .dtors .jcr .dynamic .got
```

留意 hello_relro 的 Program Header Table 中使用黑体标识的部分。显然，相比于程序 hello，hello_relro 中多了段“GNU_RELRO”。

读者可能会有个疑问，前面不是提到内核只加载 ELF 文件中类型为 LOAD 的段吗，那么这个类型为 GNU_RELRO 的段会被加载吗？请仔细观察段 RELRO 与第 2 个类型为 LOAD 的段（即数据段）的 Offset 一列，可见，RELRO 段在 hello_relro 中偏移与数据段在 hello_relro 文件中的偏移相同。换句话说，RELRO 段正是数据段的开头部分，所以在加载数据段时，已经隐含着将段 RELRO 加载了。

接下来，我们再来动态的观察一下特性 RELRO。这里偷个懒，因为目标系统也是 x86 的，所以笔者直接在宿主系统上运行了，读者当然可以将 hello_relro 复制到目标系统做这个试验。hello 的进程地址空间的映射情况如下：

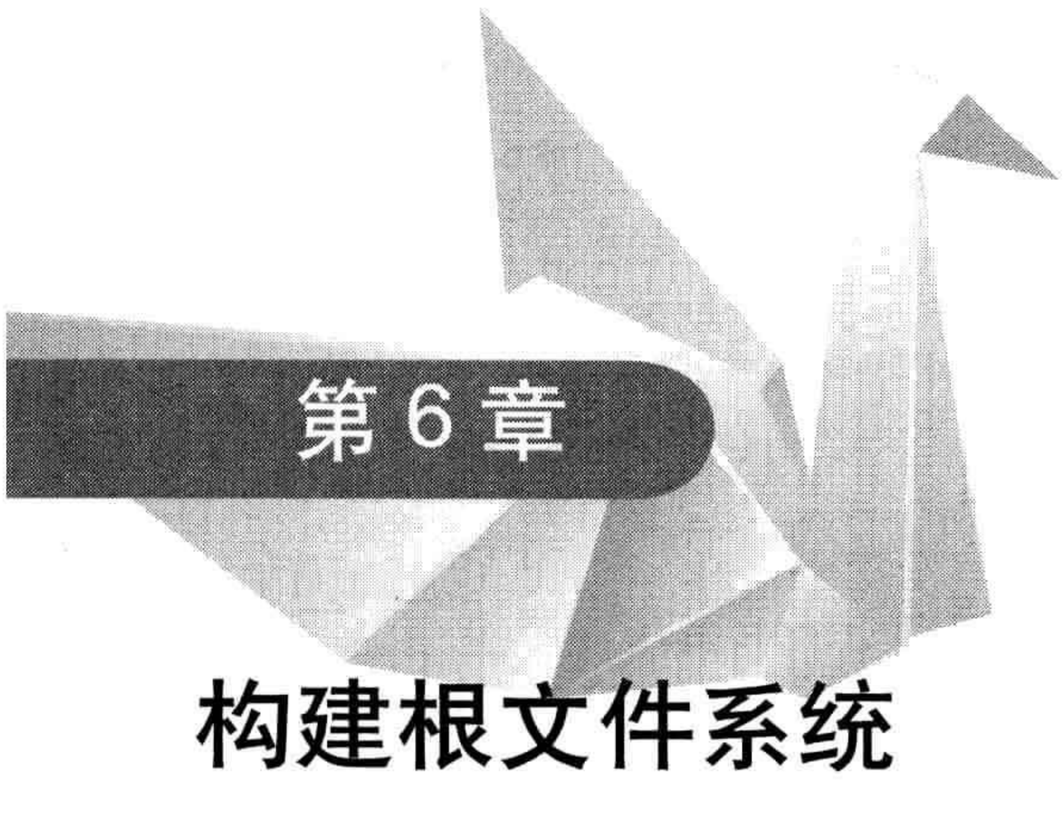
```
vita@baisheng:~$ ./hello &
[1] 4320
vita@baisheng:~$ cat /proc/4320/maps
08048000-08049000 r-xp 00000000 08:03 9447286    /home/vita/hello
08049000-0804a000 rw-p 00000000 08:03 9447286    /home/vita/hello
b753e000-b753f000 rw-p 00000000 00:00 0
...
```

hello_relro 的进程地址空间的映射情况如下：

```
vita@baisheng:~$ ./ hello_relro &
[2] 4328
vita@baisheng:~$ cat /proc/4328/maps
08048000-08049000 r-xp 00000000 08:03 9447303
                                  /home/vita/hello_relro
08049000-0804a000 r--p 00000000 08:03 9447303
                                  /home/vita/hello_relro
0804a000-0804b000 rw-p 00001000 08:03 9447303
                                  /home/vita/hello_relro
b7559000-b755a000 rw-p 00000000 00:00 0
...
```

对比 hello 和 hello_relro 的进程空间的映射情况，注意 hello_relro 映射中使用黑体标识的部分，可见，hello_relro 在 0x08049000~0x0804a000 多映射了一个只读的段，没错，这个段就是段 RELRO 。显然，因为其与后面的数据段权限不同，所以内核为这个段单独分配了一个 vm_struct_area 对象。

事实上，不仅是可执行程序，动态库也是如此。读者可以自己做些对比试验，这里不再赘述。



第6章 构建根文件系统

在第3章中，我们通过手工的方式展示了从零构建根文件系统的过程。在本章中，我们将构建一个相对完善的根文件系统，但是我们不再从零开始，毕竟一旦熟悉了原理后，余下的就是简单的重复了。第2章编译工具链时曾通过参数“--with-sysroot”指定了目标系统的文件安装的目录，后续所有的为目标系统编译的文件全部安装到了这个目录下。因此，在本章中，我们就基于这个目录下的文件构建运行在真实系统上的根文件系统。

为了更高效地开发调试，我们首先打通了目标系统的网络，建立了宿主系统与目标系统的桥梁，包括配置内核支持网络协议以及网卡驱动，并安装了用户空间的网络配置工具。如此，我们就可以远程登录到目标系统上进行调试，并且可以动态更新文件（除内核和initramfs外）而不必再每次都重启系统。

几乎所有的现代操作系统都提供图形用户界面，Linux也不例外。麻省理工的开发者们为UNIX系统开发了X窗口系统（X Window System，简称X或者X11）作为图形环境。除了X外，另外一个需要关注的图形环境是Wayland。虽然Wayland的目标是替代X，并且开源社区也支持Wayland向着这个方向发展，但是Wayland距广泛使用还有一段路要走。因此，在本章中，我们依然以目前广泛使用的X构建基础的图形环境，并安装了GTK作为更上层的图形库。事实上，Wayland更像是X的一次整合或者重构，在第8章探讨Linux的图形原理时，我们会拿出一点篇幅讨论Wayland，在那里我们会看到，Wayland和X之间并无本质区别。

6.1 初始根文件系统

因为我们使用的是vita用户进行编译过程，所以$SYSROOT目录下的所有文件的属主和属组都是vita，如果对安全问题有顾虑，在最终将其作为根文件系统时，可以将该目录下的所有文件，包括目录的属主和属组，更改为root。

另外，为简单起见，我们也没有考虑文件系统的大小。如果是为一个真实的系统制作根文件系统，那么可以考虑进行一些优化，比如对二进制文件和动态库使用命令strip删除一些

运行时不需要的信息和符号表等；删除那些只是在编译时使用的头文件和静态库；等等。

上面讨论的都不是必须的，如果仅作为一个用于测试的系统，完全可以不必理会，下面是必须要做的几件事。

（1）安装 GCC 库

在前面编译 GCC 时，我们已经看到，GCC 也将部分底层函数封装到库中，很多程序会使用 GCC 的这些库，因此，我们也将这部分程序安装到根文件系统中。我们只安装运行时使用的动态库及对应的运行时符号链接，当然，系统中并不一定会用到全部这些库，但是简单起见，这里全部安装了：

```
vita@baisheng:/vita$ cp -d \
        cross-tool/i686-none-linux-gnu/lib/lib*.so.*[0-9] \
        rootfs/lib/
```

（2）建立相关目录

在前面讨论从 initramfs 切换到根文件系统时，我们看到，切换程序将最初挂载到文件系统 rootfs 中的 /dev、/run、/proc 和 /sys 目录移动到真正的根文件系统，因此，我们需要在根文件系统上建立这几个目录。另外我们也为 root 用户建立一个属主 root 目录：

```
vita@baisheng:/vita/sysroot$ mkdir sys proc dev run root
```

（3）构建程序 /sbin/init

在内核初始化的最后，启动的第一个进程要装载用户空间的程序从而切入用户空间，通常这个程序是 /sbin 目录下的 init，因此我们要准备这个程序。为简单起见，我们也使用 shell 脚本编写：

```
/vita/sysroot/sbin/init:

#!/bin/bash
export HOME=/root
exec /bin/bash -l
```

init 启动了一个交互式的 shell。其中传递的参数“-l”是告诉 bash 以登录方式启动，这样可以使 bash 读取在 /etc/profile、~/.profile 等文件中定义的环境变量。同时要确保 init 程序具有可执行权限：

```
vita@baisheng:/vita/sysroot/sbin$ chmod a+x init
```

为了让 shell 提示符看上去友好一些，更重要的是为了后面当从宿主系统远程登录到 vita 系统时，方便区分本地终端和登录到 vita 的终端，我们在全局范围的 profile 文件中定义了环境变量 PS1 来控制 shell 提示符的显示内容和风格：

```
/vita/sysroot/etc/profile:

export PS1="\[\e[31;1m\]\u@vita:\[\e[35;1m\]\w# \[\e[0m\]"
```

其中，“\u”告诉 shell 显示当前用户名；“\w”告诉 shell 显示完整的工作路径；我们将主机名直接硬编码为 vita，为了便于区分是本地的终端还是登入 vita 的终端；接下来我们给提示符加一点漂亮的颜色，“\e[”与“m”之间的内容表示颜色值，在它们之外包围的“\[”与“\]”是保证其内的非打印字符，不占用任何空间。颜色设置的格式为“\[\e[F;B;Cm\]”，其中 F 是前景色，B 是背景色，C 是一些表示特殊效果的代码，如下划线、闪烁等。

具体到我们这个例子，其中 31 表示红色，因此，“用户名 @vita”将以红色显示；35 表示洋红，因此工作路径将以洋红色显示。最后，在提示符结束的位置，我们通过“\e[0m”将颜色值设定为零，也就是通知终端将前景、背景重置为它们的默认值，以使后续的文字以非彩色显示。

接下来将 $SYSROOT 目录整个复制到虚拟机，因为命令 scp 会跟随符号链接，所以我们采用先压缩、再复制的办法，相关命令如下：

```
vita@baisheng:/vita/sysroot$ tar zcvf ../sysroot.tgz *
vita@baisheng:/vita/sysroot$ scp ../sysroot.tgz \
                                    root@192.168.56.101:/root/
```

在虚拟机上执行如下命令解压根文件系统：

```
root@baisheng-vb:/vita# tar xvf /root/sysroot.tgz
```

6.2 以读写模式重新挂载文件系统

一般在挂载文件系统之前，将使用工具 fsck 检查文件系统的一致性。如果文件系统中存在错误，则 fsck 会试图修复它们，但是这个过程要求文件系统没有挂载或以只读方式挂载。因此我们在 GRUB 的配置文件 grub.cfg 中经常看到内核的命令行参数中有这么一个字串“ro”，其是“read only”的简写，目的是告诉内核或 initramfs 最初以只读方式挂载根文件系统。在 Linux 系统进入用户空间、使用工具 fsck 检查文件系统后，然后再以读写方式重新挂载根文件系统。这里我们忽略文件系统检查这一过程，直接以读写模式重新挂载根文件系统。

/vita/sysroot/sbin/init:

```
#!/bin/bash
mount -o remount,rw /dev/sda2 /

export HOME=/root
exec /bin/bash -l
```

如果读者没有更改根文件系统中文件的属主和属组为 root，那么更新 vita 系统的 /sbin/init 程序后，重启系统，我们来检查一下根文件系统是否以读写方式成功挂载了。以笔者的 vita 系统为例，如图 6-1 所示。

图 6-1 重新挂载文件系统失败

使用 cat 命令查看“/proc/mounts”，发现根文件系统依然是以只读方式挂载的。使用命令 touch 试图尝试创建一个文件，创建也以失败告终，再次证明根文件系统确实是以只读方式挂载的。我们手动再次尝试以读写方式重新挂载根文件系统，还是失败了，但是我们看到 mount 命令提示了一个非常有用的信息：effective UID is 1001。看上去似乎是权限出了问题。mount 命令只允许以 root 的身份进行挂载操作，所以 EUID 应该是 0 才对，但是这里的 EUID 却是 1001，这个 1001 是哪来的呢?

在安装时，安装脚本设置了工具 mount 和 umount 的 SUID，相关脚本如下：

```
util-linux-2.22/Makefile

install-exec-hook-mount:
    chmod 4755 $(DESTDIR)$(bindir)/mount
    chmod 4755 $(DESTDIR)$(bindir)/umount
```

使用 ls 命令查看这两个文件的信息可见，SUID 确实被设置了：

```
vita@baisheng:/vita/sysroot/bin$ ls -l *mount
-rwsr-xr-x 1 vita vita 103045 Jan 29 17:17 mount
-rwsr-xr-x 1 vita vita  40612 Jan 29 17:17 umount
```

因此，虽然进程 1 的 EUID 是超级用户 root，但是一旦执行 mount 命令时，其 EUID 将降为 vita 用户的 UID，而在笔者的宿主系统上，vita 的 UID 正是 1001，这就是上面“effective UID is 1001”的来源。而 mount 命令是要求以超级用户 root 运行的，但是此时进程 1 被降级为 1001，mount 命令自然拒绝执行挂载任务。

因此，我们需要修改 mount 和 umount 的属主和属组为 root，命令如下：

```
root@baisheng:/vita/sysroot/bin# chown root.root mount umount
```

更改属主和属组会导致这两个程序的 SUID 也会被丢弃，但是对我们来说，这没有什么

问题。如果很介意 mount 和 umount 的 SUID，执行下面的命令重新设置即可：

```
root@baisheng:/vita/sysroot/bin# chmod 4755 mount umount
```

读者可能会问，第 4 章在讨论 initramfs 时，也使用了 mount，那时为什么没有这个权限问题？原因是我们在从 $SYSROOT 复制到那个手工搭建的基本的根文件系统时，SUID 被丢弃了。很多有安全要求的领域经常使用 SUID 这个技巧，比如管理员通常会设置一些网络服务器程序的 SUID，这样一旦被人攻破，也不能获得超级用户 root 的权限。SUID 是个比较抽象的概念，通过这个例子，我们切实体验了一次 SUID。

6.3　配置内核支持网络

为了方便宿主系统与目标系统传输文件，并且可以从宿主系统登录到目标系统，目标系统需要支持网络。为此，需要配置目标系统的内核支持 TCP/IP 协议和网卡驱动。

6.3.1　配置内核支持 TCP/IP 协议

我们首先配置内核使其支持 TCP/IP 协议，步骤如下：

1）执行 make menuconfig，出现如图 6-2 所示的界面。

```
    Bus options (PCI etc.)  --->
    Executable file formats / Emulations  --->
[*] Networking support  --->
    Device Drivers  --->
```

图 6-2　配置内核支持 TCP/IP（1）

2）在图 6-2 中，选择菜单项“Networking support”，出现如图 6-3 所示的界面。

```
--- Networking support
    Networking options  --->
[ ] Amateur Radio support (NEW)  --->
< > CAN bus subsystem support (NEW)  --->
```

图 6-3　配置内核支持 TCP/IP（2）

3）在图 6-3 中，选择菜单项“Networking options”，出现如图 6-4 所示的界面。

```
<*> Unix domain sockets
< >   UNIX: socket monitoring interface
< > Transformation user configuration int
< > PF_KEY sockets
[*] TCP/IP networking
[ ]   IP: multicasting (NEW)
[ ]   IP: advanced router (NEW)
```

图 6-4　配置内核支持 TCP/IP（3）

4）在图 6-4 中，选中“TCP/IP networking”，TCP/IP 协议配置完成。

6.3.2 配置内核支持网卡

接下来我们配置内核中的网卡驱动。既然是为网卡配置驱动，首先当然需要知道系统使用的是什么网卡。那么我们如何查看目标系统的网卡型号呢？对于普通的 PC 来说，网卡一般是连接在 PCI 总线上的 PCI 设备，这里我们不考虑通过其他接口（如 USB）转换的网卡。既然是连接在 PCI 总线上，读者一定想到了前面使用的查看硬盘控制器信息的工具 lspci。

在目标系统上运行 lspci，查看与以太网相关的设备。确定设备所在的 PCI 总线上的位置后，查看这个网卡的环境变量 MODALIAS，如图 6-5 所示。

图 6-5 查看网卡控制器信息

根据命令 lspci 的输出可见，在总线号为 0x00 的 PCI 总线上，设备号为 0x03 的设备就是 Intel 的型号为 82540EM 千兆以太网卡。

根据内核通过 sys 文件系统报告的 uevent 事件，我们可以清楚地看到，环境变量 MODALIAS 的值为：

```
pci:v00008086d0000100Esv00008086sd0000001Ebc02sc00i00
```

以设备 ID“100E”在内核的 drivers/net 目录下搜索，结果如下：

```
vita@baisheng:/vita/build/linux-3.7.4/drivers/net$ grep "100E" \
                                                    -Ir *

ethernet/chelsio/cxgb/elmer0.h:   ELMER0_XC2S100E_6TQ144_C
ethernet/intel/e1000/e1000_main.c:
                INTEL_E1000_ETHERNET_DEVICE(0x100E),
ethernet/intel/e1000/e1000_hw.h:#define E1000_DEV_ID_82540EM
                                        0x100E
fddi/defxx.h:#define PI_ITEM_K_SMT_HI_VERS_ID    0x100E
fddi/skfp/pmf.c:   { SMT_P100E,AC_G,
MOFFSS(fddiSMTHiVersionId),       "S"     } ,
```

```
fddi/skfp/h/smt_p.h:#define       SMT_P100E       0x100e
wireless/adm8211.c:        ADM8211_CSR_WRITE(MMIWA,  0x100E0C0A);
```

根据上面输出结果中使用黑体标识的部分可见，驱动 e1000 声明对设备 ID 为“100E”的设备负责。也就是说，驱动 e1000 是 Intel 82540EM 千兆以太网卡的驱动。因此，我们需要配置内核支持 e1000 驱动，配置步骤如下：

1）执行 make menuconfig，出现如图 6-6 所示的界面。

```
    Executable file formats / Emulations  --->
[ ] Networking support  --->
    Device Drivers  --->
    Firmware Drivers  --->
```

图 6-6 配置内核支持网卡驱动（1）

2）在图 6-6 中，选择菜单项“Device Drivers”，出现如图 6-7 所示的界面。

```
< > I2O device support  --->
[ ] Macintosh device drivers  --->
[*] Network device support  --->
    Input device support  --->
    Character devices  --->
```

图 6-7 配置内核支持网卡驱动（2）

3）在图 6-7 中，选中菜单项“Network device support”，出现如图 6-8 所示的界面。

```
< >   ARCnet support (NEW)  --->
      *** CAIF transport drivers ***
[*]   Ethernet driver support (NEW)  --->
< >   FDDI driver support (NEW)
< >   General Instruments Surfboard 1000 (
```

图 6-8 配置内核支持网卡驱动（3）

4）在图 6-8 中，选中菜单项“Ethernet driver support”，出现如图 6-9 所示的界面。

```
[*]   Intel devices (NEW)
< >     Intel(R) PRO/100+ support (NEW)
<M>     Intel(R) PRO/1000 Gigabit Ethernet support
< >     Intel(R) PRO/1000 PCI-Express Gigabit Ether
< >     Intel(R) 82575/82576 PCI-Express Gigabit Et
```

图 6-9 配置内核支持网卡驱动（4）

5）在图 6-9 中，将“Intel(R) PRO/1000 Gigabit Ethernet support”配置为模块，网卡驱动配置完成。

重新编译内核和模块，并将内核映像以及内核模块安装到 /vita/sysroot 目录下。

6.4 启动 udev

前面我们将网卡驱动编译为模块，为了自动加载网卡驱动，需要启动 udev，为此需要修改 init 脚本：

```
/vita/sysroot/sbin/init:

#!/bin/bash
mount -o remount,rw /dev/sda2 /

udevd --daemon
udevadm trigger --action=add
udevadm settle

export HOME=/root
exec /bin/bash -l
```

这几条命令我们在讨论 initramfs 时已经见过了。事实上，这里启动 udev 服务，不仅是为了加载网卡驱动模块。initramfs 中往往只包含存储介质相关的驱动，而其他大量设备的驱动，大部分还是保存在根文件系统中，所以，在挂载了根文件系统后，需要重新模拟一遍热插拔，从根文件系统中加载相关设备的驱动模块。

6.5 安装网络配置工具并配置网络

在用户空间中，我们使用工具 ip 来配置网络，工具 ip 包含在软件包 iproute2 中。所以我们首先来编译安装软件包 iproute2。

```
vita@baisheng:/vita/build$ tar \
     xvf ../source/iproute2-3.8.0.tar.xz
```

iproute 中包含很多网络管理工具，但是其中一些工具我们构建的 vita 系统并不需要。而编译这些不必要的工具还需要引入一些额外的库或者工具，比如网络流量控制工具和套接字统计工具要求系统安装工具 bison。因此，我们只安装和网络配置相关的工具。为此，在 iproute2 的顶层目录下的 Makefile 中，将下面的编译目标：

```
SUBDIRS=lib ip tc bridge misc netem genl man
```

修改为：

```
SUBDIRS=lib ip
```

执行如下命令编译安装：

```
vita@baisheng:/vita/build/iproute2-3.8.0$ make install
```

为了验证我们的网络是否配置正确，我们安装 ping 工具，该工具在软件包 iputils 中。

```
vita@baisheng:/vita/build$ tar \
     xvf ../source/iputils-s20121221.tar.bz2
```

我们只编译 IPv4 的 ping 工具，在 iputils 的顶层目录下的 Makefile 中，将下面的编译目标：

```
IPV4_TARGETS=tracepath ping clockdiff rdisc arping tftpd rarpd
IPV6_TARGETS=tracepath6 traceroute6 ping6
                TARGETS=$(IPV4_TARGETS) $(IPV6_TARGETS)
```

修改为：

```
IPV4_TARGETS=ping
IPV6_TARGETS=tracepath6 traceroute6 ping6
                TARGETS=$(IPV4_TARGETS)
```

我们构建的 vita 系统中目前没有安装 Capability 相关的库，因此我们去掉 ping 对库 Capability 的依赖，我们也不需要 ping 的这个特性。因此，在 iputils 的顶层目录下的 Makefile 中，将下面的变量：

```
USE_CAP=yes
```

修改为：

```
USE_CAP=no
```

执行如下命令编译安装：

```
vita@baisheng:/vita/build/iputils-s20121221$ make
vita@baisheng:/vita/build/iputils-s20121221$ cp ping \
       /vita/sysroot/bin/
```

更新 vita 系统的根文件系统并重新启动，然后使用如下命令查看网络接口：

```
ip link show
```

如果网卡被正确驱动了，那么应该可以看到网络接口。笔者机器的网络接口为 eth0，因此在后面的命令中使用的是 eth0，读者可能需要根据自己的具体情况调整。一般而言，第一块有线网卡接口都为 eth0。

在配置网络前，如果网络接口的状态是“down”，那么首先使用如下命令将网络接口状态设置为“up”：

```
ip link set eth0 up
```

然后使用如下命令设置网卡的 IP 地址：

```
ip addr add 192.168.56.2/24 dev eth0
```

具体的 IP 地址需要根据读者自己的实际情况调整，总之，需要和宿主系统在一个网段上。

设置了 IP 地址后，工具 ip 自动增加了路由，可以使用如下命令查看：

```
ip route show
```

图 6-10 是在笔者构建的 vita 系统上配置网络的过程。

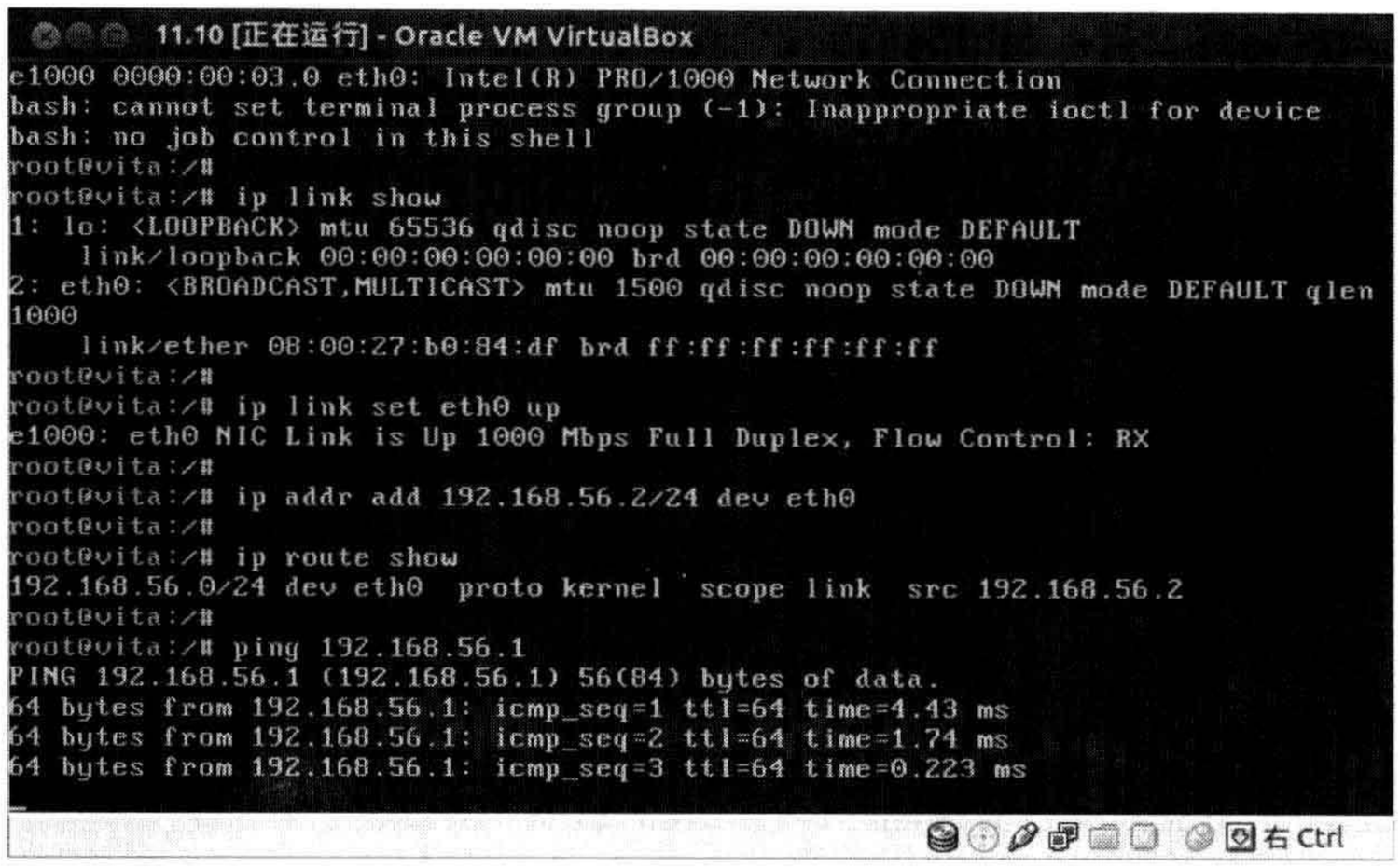


图 6-10 网络配置过程

配置完成后，可以使用命令 ping 确认网络是否已经成功配置。

最后，为了不必每次重启系统后都手动重复执行这些网络配置命令，我们将其添加到 init 中：

```
/vita/sysroot/sbin/init:

#!/bin/bash
mount -o remount,rw /dev/sda2 /

udevd --daemon
udevadm trigger --action=add
udevadm settle

ip link set eth0 up
ip addr add 192.168.56.2/24 dev eth0

export HOME=/root
exec /bin/bash -l
```

6.6 安装并配置 ssh 服务

既然网络已经配置好了，一般情况下就不必再通过第三方系统（即虚拟机）更新 vita 系统了，可以直接通过网络和 vita 系统打交道了。当然如果是更新内核、initramfs 或者整个文件系统，还是要通过虚拟机系统的。我们在宿主系统和 vita 系统之间使用 ssh 服务进行通信。因此在这一节，我们为 vita 系统安装并配置 ssh 服务。

我们使用 ssh 协议的开源实现 openssh，其依赖 zlib 和 openssl，因此首先编译安装这两个软件包。

使用如下命令编译安装 zlib：

```
vita@baisheng:/vita/build$ tar xvf ../source/zlib-1.2.7.tar.bz2
vita@baisheng:/vita/build/zlib-1.2.7$ ./configure --prefix=/usr
vita@baisheng:/vita/build/zlib-1.2.7$ make
vita@baisheng:/vita/build/zlib-1.2.7$ make install
vita@baisheng:/vita/build/zlib-1.2.7$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;
```

使用如下命令编译安装 openssl：

```
vita@baisheng:/vita/build$ tar xvf \
     ../source/openssl-1.0.1e.tar.gz
vita@baisheng:/vita/build/openssl-1.0.1e$ ./config --prefix=/usr
     --openssldir=/etc/ssl
vita@baisheng:/vita/build/openssl-1.0.1e$ make
vita@baisheng:/vita/build/openssl-1.0.1e$ make install \
     MANDIR=/usr/share/man INSTALL_PREFIX=$SYSROOT
vita@baisheng:/vita/build/openssl-1.0.1e$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;
```

openssh 的依赖已经安装完成，下面安装 openssh，命令如下：

```
vita@baisheng:/vita/build$ tar xvf ../source/openssh-6.1p1.tar.gz
vita@baisheng:/vita/build/openssh-6.1p1$ \
     LD=i686-none-linux-gnu-gcc ./configure \
     --prefix=/usr --sysconfdir=/etc/ssh \
     --without-openssl-header-check
vita@baisheng:/vita/build/openssh-6.1p1$ make install \
     DESTDIR=$SYSROOT
```

在 openssh 的编译脚本中，调用链接器时传递了参数 -fstack-protector-all。链接器不允许链接可执行文件时使用以“-f”开头的参数，以“-f”开头的参数只能用于链接动态库。解决这个问题的方法之一就是避免直接调用链接器进行链接，而是通过 gcc 间接调用链接器。这就是在配置 openssh 时设定 LD=i686-none-linux-gnu-gcc，覆盖系统环境变量中定义的 LD=i686-none-linux-gnu-ldd 的目的。

读者可能会有个疑问：在宿主系统上编译 openssh 时并不会遇到类似问题啊！那是因为在非交叉编译环境下，一般系统环境变量中不会定义 LD，而如果环境变量中没有定义，那么 openssh 的编译脚本则将 LD 定义为编译器，从而绕过了这个问题，脚本如下：

```
openssh-6.1p1/configure.ac:

if test -z "$LD" ; then
    LD=$CC
fi
```

当然读者不必纠结这个问题，这仅是 openssh 在交叉编译环境中的一个小插曲而已。安

装完 ssh 后，下面我们开始配置 ssh 服务。

openssh 支持一种安全机制，称为特权分离（Privilege Separation），这个机制是默认开启的。但是这个机制要求一些附加操作，比如建立非特权用户等。为简单起见，我们关掉了这个机制。

为了方便，vita 系统允许 ssh 服务使用 root 用户登录。同样为了方便，笔者将 vita 系统的 root 密码设置为空，因此也需要配置 ssh 服务允许登录用户密码为空。

最终，ssh 服务的配置文件 sshd_config 中的相关变量按照如下进行修改：

```
/vita/sysroot/etc/ssh/sshd_config:

UsePrivilegeSeparation no
PermitRootLogin yes
PermitEmptyPasswords yes
```

除了配置 ssh 服务外，根据 ssh 协议 2.0 的要求，还需要为 ssh 服务创建 dsa、rsa 和 ecdsa 三种类型的密钥。而创建密钥需要一些账户信息，因此，我们首先要为 vita 系统添加账户信息。

用户信息保存在文件 /etc/passwd 中，格式为：

```
name:password:uid:gid:comment:home:shell
```

其中，name 是用户名；password 是用户密码；uid 是用户 ID；gid 是用户所属的组；comment 保存如用户的真实姓名等一些信息；home 是用户的属主目录；shell 是用户登录后执行的命令。

组信息保存在文件 /etc/group 中，格式为：

```
group_name:password:gid:user_list
```

其中，group_name 是组名；password 是组的密码；gid 是组 ID；user_list 部分记录属于该组的所有用户（用户之间使用逗号分隔）。

我们在 vita 系统上创建的的具体的 passwd 和 group 文件分别如下：

```
/vita/sysroot/etc/passwd:
root::0:0::/root:/bin/bash

/vita/sysroot/etc/group:
root::0:
```

一切准备就绪后，更新 vita 的文件系统。重启后，在 vita 系统上使用命令 ssh-keygen 创建密钥，这个工具也是软件包 openssh 提供的。

在默认情况下，dsa、rsa 和 ecdsa 分别存储在文件 /etc/ssh/ssh_host_dsa_key、/etc/ssh/ssh_host_rsa_key 以及 /etc/ssh/ssh_host_ecdsa_key 中，当然也可以在 ssh 服务的配置文件 sshd_config 中修改这些默认的设置。创建密钥的命令分别如下：

```
ssh-keygen -t dsa -f /etc/ssh/ssh_host_dsa_key
ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key
ssh-keygen -t ecdsa -f /etc/ssh/ssh_host_ecdsa_key
```

当 ssh-keygen 提示输入“passphrase”时，直接按回车即可。图 6-11 是在笔者构建的 vita 系统上创建 dsa 密钥的过程：

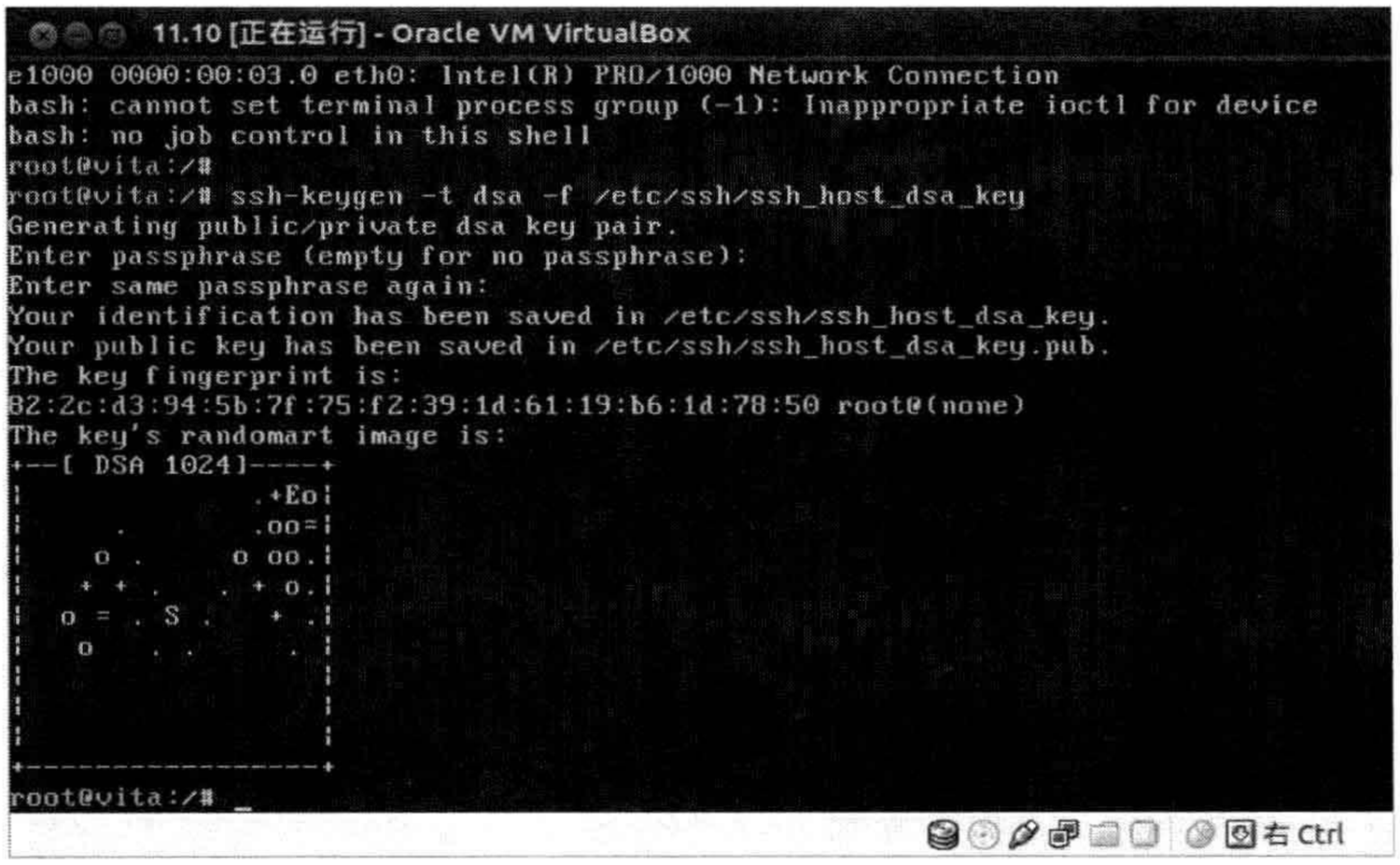


图 6-11 创建 dsa 密钥

其他两个密钥 rsa 和 ecdsa 按照 dsa 的创建如法炮制即可。

密钥创建一次即永久保存在文件系统了，如果删除了 vita 系统的根文件系统，需要再次重新创建这几个密钥。

从宿主系统远程登录 vita 系统时，vita 系统需要为登录的用户分配伪终端（PTY），而伪终端设备节点建立在 /dev/pts 目录下，并且 /dev/pts 要求挂载 devpts 文件系统。如果没有挂载 devpts, 登录将失败，报错信息类似如下：

```
PTY allocation request failed on channel 0
```

为此，修改 init 程序，挂载 devpts，方法如下：

```
/vita/sysroot/sbin/init:

#!/bin/bash
mount -o remount,rw /dev/sda2 /

udevd --daemon
udevadm trigger --action=add
udevadm settle

ip link set eth0 up
ip addr add 192.168.56.2/24 dev eth0
```

```
mkdir /dev/pts
mount -n -t devpts devpts /dev/pts

export HOME=/root
exec /bin/bash -l
```

一切就绪，在 vita 系统启动时，默认启动 ssh 服务：

```
/vita/sysroot/sbin/init:

#!/bin/bash
mount -o remount,rw /dev/sda2 /

udevd --daemon
udevadm trigger --action=add
udevadm settle

ip link set eth0 up
ip addr add 192.168.56.2/24 dev eth0

mkdir /dev/pts
mount -n -t devpts devpts /dev/pts

/usr/sbin/sshd

export HOME=/root
exec /bin/bash -l
```

更新 vita 系统的 init 程序，重启后，就可以顺利地从宿主系统登录到 vita 系统了。一旦可以远程登录到 vita，工作的效率就会大大提高，我们可以使用宿主系统的强大的终端。而且，更重要的一点是，除非对根文件系统进行彻底的更新，或者更新内核，否则不再需要频繁地重启系统。

6.7 安装 procps

为了方便后面调试，我们在 vita 系统上安装 procps。该软件包中包含了常用的一些工具，如 ps、kill 等。因为 vita 系统中没有安装 ncurses 库，为简单起见，我们只编译不需要使用 ncurses 库绘制界面的程序，这就是编译 procps 时传递参数“--without-ncurses”的目的。

```
vita@baisheng:/vita/build$ tar \
     xvf ../source/procps-ng-3.3.6.tar.xz
vita@baisheng:/vita/build/procps-ng-3.3.6$ ./configure \
     --prefix=/ --without-ncurses
vita@baisheng:/vita/build/procps-ng-3.3.6$ make
vita@baisheng:/vita/build/procps-ng-3.3.6$ make install
```

6.8 安装 X 窗口系统

UNIX 系统的主要目标就是多用户、多任务，而且允许多个用户远程登录并发执行任务。

这种设计哲学同样被带到了X窗口系统中。X的实现者将X设计为客户/服务器的架构，应用程序相当于客户端，它们不需要关心具体的显示和用户输入，而由X服务器负责管理显示设备和输入设备。应用程序只需要将请求，比如“绘制一条直线从点A到点B”，发送给X服务器，而由X服务器负责将其绘制到具体的显示设备上。X服务器也会将用户的输入（包括鼠标、键盘等输入事件），转发给对应的应用。

X将协议相关实现封装到了一个库中，开发者将这个库称为Xlib。后来因为效率问题，又开发了xcb来替代Xlib。Xlib中封装的只是X的核心协议，X使用扩展的方式扩充X协议，其他扩展协议可以在单独的库中实现。

作为类UNIX的图形系统的基础，X的复杂是难以避免的。也恰恰是因为X的复杂，很多人提及X的安装就会谈虎色变。虽然X系统非常庞大，实际上它也是有章可循的。本节笔者就带领读者从头安装一个X窗口系统。鉴于X的安装过程比较烦琐和复杂，我们提供了一个安装脚本build-X11.sh。但是笔者建议读者尽量使用手动的方式安装，这样可以在思考和解决问题中不断提高。遇到自己实在解决不了的问题时再参考这个脚本，从而达到更好的学习效果。

6.8.1 安装M4宏定义

X定义了一些公用的M4宏，并将它们放在软件包util-macros中。X的各个组件的配置脚本中将使用M4宏，因此我们首先来安装M4宏，方法如下：

```
vita@baisheng:/vita/build$ tar xvf \
     ../source/X7.7/util-macros-1.17.tar.bz2
vita@baisheng:/vita/build/util-macros-1.17$ ./configure \
     --prefix=/usr
vita@baisheng:/vita/build/util-macros-1.17$ make install
```

在第7章讨论窗口管理器的构建脚本时，我们介绍了M4宏，读者可以参考那里的讨论。

6.8.2 安装X协议和扩展

X包含了多种协议和扩展，为简单起见，Vita系统不必全部安装。比如禁掉了记录事件的扩展Record，支持扩展屏幕的协议Xinerama及用于屏保的Screensaver，禁掉了已经过时的DRI1等。下面是vita系统安装的协议，安装这些协议时没有先后顺序要求。如果不要求X服务器支持DRI2，那么可以安装更少的协议，比如去掉glproto、dri2proto、damageproto等。

（1）核心协议

Xlib中的绝大部分编程接口，如XCreateWindow、XMapWindow、XDrawRectange、XCopyArea等都是由X核心协议定义的。核心协议的定义在软件包xproto中。

（2）基本扩展

X的基本扩展包括：DOUBLE-BUFFER（DBE）、DPMS、Extended-Visual-Information

（EVI）、Generic Event Extension、LBX、MIT-SHM、MIT-SUNDRY-NONSTANDARD、Multi-Buffering、SECURITY、SHAPE、SYNC、TOG-CUP、XC-APPGROUP、XTEST。它们的定义在软件包 xextproto 中。

（3）键盘扩展

键盘扩展定义了键盘的模型、布局，如对于不同的键盘模型，某个键值对应的字符。键盘扩展的定义在软件包 kbproto 中。

（4）输入扩展

输入扩展是为一些特殊的输入设备定义的协议。通过这个扩展，输入设备可以模拟出与鼠标、键盘等核心输入设备相同格式的事件。输入扩展的定义在软件包 inputproto 中。

（5）XCB 协议

鉴于 Xlib 的效率，开发者们开发了更高效的 XCB 来替代 Xlib。XCB 协议是用于这个库的协议，其以 XML 形式定义，并提供 python 程序将这些 XML 描述文件转换为相应的程序代码。XCB 协议的定义在软件包 xcb-proto 中。

（6）GLX 扩展

GLX 扩展定义了 OpenGL 和 X 之间通信的协议。该扩展的定义在软件包 glxproto 中。

（7）DRI2 扩展

DRI2 扩展是 DRI 的第 2 个版本，定义了应用不通过 X 服务器直接使用硬件进行渲染的协议。DRI2 扩展的定义在软件包 dri2proto 中。

（8）XFixes 扩展

从这个扩展的名字也可以看出，这个扩展其实是为解决 X 核心协议存在的各种限制的。该扩展的定义在软件包 fixesproto 中。

（9）Damage 扩展

Damage 扩展是 X 服务器用来记录那些离屏的、发生了变化的绘制区域的协议。Damage 扩展的定义在软件包 damageproto 中。

（10）XC-MISC 扩展

应用可以通过 XC-MISC 扩展获取 X 服务器可用的资源 ID，如 GetXIDRange、GetXIDList 等。该扩展的定义在软件包 xcmiscproto 中。

（11）BIG-REQUESTS 扩展

BIG-REQUESTS 扩展提供了对大于 262140 字节的请求的支持。该扩展的定义在软件包 bigreqsproto 中。

（12）RANDR 扩展

RANDR 扩展定义了动态调整屏幕尺寸、旋转屏幕以及镜像屏幕的协议。X 提供的工具 xrandr 就是这个协议的一个典型使用者。该扩展的定义在软件包 randrproto 中。

（13）RENDER 扩展

RENDER 扩展是 X 使用的较新的渲染模型，用于合成多个绘制区域，相对于原始的通

过复制进行合成的模型其更有效率。该扩展的定义在软件包 renderproto 中。

（14）字体扩展

字体扩展定义了 X 中与字体处理相关的协议。字体扩展的定义在软件包 fontsproto 中。

（15）视频扩展

视频扩展定义了 X 的视频输出相关的协议。该扩展的定义在软件包 videoproto 中。

（16）复合扩展

复合扩展是为了 X 支持窗口特效设计的扩展。在没有这个扩展之前，所有的在窗口上的绘制操作都“实时”显示在屏幕上。而复合扩展允许窗口可以先在离屏的区域进行绘制。复合扩展的定义在软件包 compositeproto 中。

（17）资源扩展

资源扩展定义了应用程序查询 X 服务器各种资源使用情况的协议。该扩展的定义在软件包 resourceproto 中。

（18）直接图形访问扩展

顾名思义，直接图形访问扩展也是为了直接访问图形硬件设计的协议，不过其功能非常有限，目前基本已经停止开发，但 vesa 驱动还在使用这个扩展。该扩展的定义在软件包 xf86dgaproto 中。

这些协议的配置安装都非常简单，而且安装命令完全相同。以 xproto 为例，安装命令如下：

```
vita@baisheng:/vita/build$ tar xvf \
     ../source/X7.7/xproto-7.0.23.tar.bz2
vita@baisheng:/vita/build/xproto-7.0.23$ ./configure \
     --prefix=/usr
vita@baisheng:/vita/build/xproto-7.0.23$ make install
```

6.8.3 安装 X 相关库和工具

在安装 X 服务器前，我们需要安装 X 服务器依赖的库、这些库依赖的库以及 X 服务器使用的工具和相关数据。注意，某些库是有安装顺序要求的，比如，libX11 需要在 libxkbfile 前安装，安装 libXfont 前需要先安装 freetype，libdrm、expat 需要在 Mesa 前安装等。读者按照下面的顺序安装即可。

（1）pixman

pixman 是一个底层的像素操作的库，提供图形合成及光栅化等功能，是 X 中软件渲染的基础。

（2）xtrans

xtrans 封装了网络传输的基本功能，从开发角度讲，是 X 服务器和应用程序之间进行通信的基础。X 服务器、libX11 等 X 的相关组件都要用到这个库。

（3）libXau

libXau 是 X 服务器和应用程序之间认证授权使用的库。

（4）libX11、libxcb 和 libpthread-stubs

libX11 是为应用程序提供的 X 协议的实现，应用程序使用 libX11 中提供的 API 和 X 服务器进行通信。

因为 libX11 的效率问题，开发人员又开发了 libxcb 来替换 libX11。而反过来，libX11 也基于 xcb 进行了改进，所以在安装 libX11 前，需要安装 libxcb。

libxcb 依赖 libpthread-stubs，因此在安装 libxcb 前需要先安装 libpthread-stubs。

（5）libxkbfile、xkbcomp 和 xkeyboard-config

这三个包都与键盘扩展相关。X 服务器根据键盘扩展，确定不同键盘模型的键盘的布局、键值到字符的转换等。键盘相关的数据就包含在 xkeyboard-config 中。

而开发者将操作这些数据的功能封装在库 libxkbfile 中。

xkbcomp 包中提供了同名的工具 xkbcomp，该工具根据键盘映射的描述，将键盘映射编译为 X 服务器可以识别的指定格式。

（6）libXfont、libfontenc 和 freetype

这几个库都是与字体处理相关的。开发者将 X 使用的与字体相关的功能封装在库 libXfont 中。

而 libXfont 使用 freetype 进行字体渲染，使用 libfontenc 处理字体编码。所以安装 libXfont 前需要安装 libfontenc 和 freetype。

（7）pciaccess

早期版本的 GPU 的 2D 驱动，包括 X 服务器中的一些功能，不通过内核，而是直接访问 PCI 接口的 GPU，这就是这个库的由来。现在虽然 GPU 驱动都通过内核访问 GPU 硬件了，但是 X 服务器中并没有清理得特别干净，还残存着对 pciaccess 库的依赖。

库 libdrm 中也使用了部分 pciaccess 中的功能。比如通过读取 PCI 寄存器探测 BIOS 中给 GPU 分配的显存大小，libdrm 借助的就是库 pciaccess 中的函数。

（8）libdrm

用户空间的组件，如 GPU 的 2D 驱动和 3D 驱动、GLX 扩展（包括 X 服务器端和 Mesa 端的实现部分）等，都需要通过内核的 DRM 模块访问 GPU。为了方便用户空间的组件访问内核 DRM 模块，开发者开发了库 libdrm。

（9）Mesa、expat、libXext、libXdamage 和 libXfixes

如果配置 X 服务器支持 DRI2，那么必须要安装 Mesa，它是 3D 应用程序进行直接渲染的基础。

Mesa 中的 DRI 扩展使用 Damage 扩展告知 X 服务器绘制完成，因此需要安装 libXdamage。

Mesa 中的 DRI2 扩展使用 XFixes 扩展中的如 XFixesCreateRegion 创建发生了改变的区

域，也就是绘制发生的区域，因此也需要安装库 libXfixes。

而在安装扩展前，需要安装库 libXext。它是所有扩展的公共库。

另外，Mesa 使用 expat 解析 XML，所以安装 Mesa 前，还需要安装 expat。

在安装上述相关库之前，在宿主系统上还需安装几个辅助的软件包。一个是 xkeyboard-config 依赖的 intltool。另外是 Mesa 依赖的 xutils-dev、flex 和 bison，使用如下命令在宿主系统上安装这几个软件包：

```
root@baisheng:~# apt-get install intltool
root@baisheng:~# apt-get install xutils-dev
root@baisheng:~# apt-get install flex
root@baisheng:~# apt-get install bison
```

除了 Mesa 外，这些库的安装完全相同。以 pixman 为例，配置及安装命令如下：

```
vita@baisheng:/vita/build$ tar xvf ../source/pixman-0.28.0.tar.gz
vita@baisheng:/vita/build/pixman-0.28.0$ ./configure \
    --prefix=/usr
vita@baisheng:/vita/build/pixman-0.28.0$ make
vita@baisheng:/vita/build/pixman-0.28.0$ make install
```

Mesa 的配置要稍复杂一点，配置命令如下：

```
vita@baisheng:/vita/build/Mesa-8.0.3$ ./configure --prefix=/usr \
    --with-dri-drivers=swrast,i915,i965 \
    --disable-gallium-llvm --without-gallium-drivers
```

因为笔者的测试机器使用的是 Intel 的 GPU，因此为了简单，这里仅编译了 Intel GPU 的 3D 驱动，而且使用经典模式的 3D 驱动，不使用 Gallium3D 模式的驱动。

另外，我们不使用 libtool 查找依赖库，因此每次安装完库后，切记使用如下命令删除 la 文件，以避免 libtool 带来麻烦：

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

6.8.4 安装 X 服务器

万事俱备，现在我们开始安装 X 服务器，配置及安装命令如下：

```
vita@baisheng:/vita/build$ tar \
     xvf ../source/X7.7/xorg-server-1.12.2.tar.bz2
vita@baisheng:/vita/build/xorg-server-1.12.2$ ./configure \
     --prefix=/usr --enable-dri2 --disable-dri --disable-xnest \
     --disable-xephyr --disable-xvfb --disable-record \
     --disable-xinerama --disable-screensaver \
     --with-xkb-output=/var/lib/xkb --with-log-dir=/var/log
vita@baisheng:/vita/build/xorg-server-1.12.2$ make
vita@baisheng:/vita/build/xorg-server-1.12.2$ find $SYSROOT \
     -name "*.la" -exec rm -f '{}' \;
```

各项配置参数意义如下。

- --enable-dri2、--disable-dri：支持 DRI2 扩展，不支持已经过时的 DRI1 扩展。
- --disable-xnest、--disable-xephyr、--disable-xvfb：我们不需要模拟的 X 服务器 Xnest、Xephyr 和 Xvfb，所以没有必要浪费时间编译它们。
- --disable-record、--disable-xinerama、--disable-screensaver：为简单起见，我们不使用 X 服务器的这几个特性。
- --with-xkb-output=/var/lib/xkb：指定工具 xkbcomp 编译的键盘映射文件存放在 /var/lib/xkb 目录下。
- --with-log-dir=/var/log：指定 X 服务器将日志文件保存在 /var/log 目录下。

6.8.5 安装 GPU 的 2D 驱动

如果只是在虚拟机上运行目标系统，安装 vesa 驱动即可，安装命令如下：

```
vita@baisheng:/vita/build$ tar xvf \
     ../source/X7.7/xf86-video-vesa-2.3.1.tar.bz2
vita@baisheng:/vita/build/xf86-video-vesa-2.3.1$ ./configure \
     --prefix=/usr
vita@baisheng:/vita/build/xf86-video-vesa-2.3.1$ make
vita@baisheng:/vita/build/xf86-video-vesa-2.3.1$ make install
```

但是如果是在真实的机器上运行目标系统，最好安装相应 GPU 的 2D 驱动。在安装脚本 build-X11.sh 中，包含了安装 Intel GPU 的 2D 驱动的方法，读者如果需要，可以参考。PC 上使用的 GPU 一般都符合 VESA 标准，所以在通常情况下，用 vesa 也能勉强驱动，但是 vesa 驱动很多特性不支持，比如硬件加速。

6.8.6 安装 X 的输入设备驱动

看到输入设备驱动，读者可能会有个疑问：内核中不是包括了各种设备的驱动吗？怎么 X 中还要安装设备驱动？没错，输入设备的驱动是在内核中，X 中的所谓输入设备的驱动 evdev 谈不上是一个驱动了，只不过大家习惯这么称呼而已。evdev 模块并不面向任何具体输入设备，它只不过是接收和解析内核发送到用户空间的输入事件。仔细观察图 6-12 所示的 Linux 输入子系统的架构，读者自然就会明白。

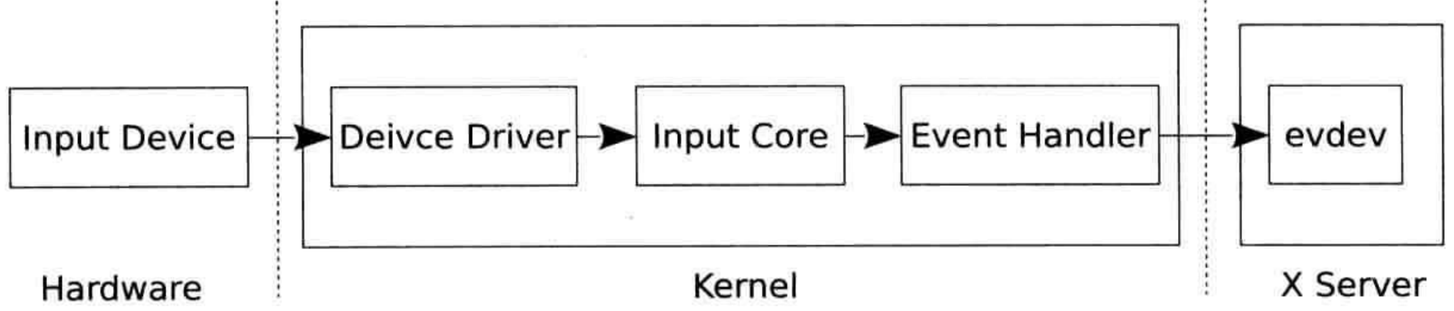


图 6-12 Linux 输入子系统架构

操作系统将面对各种各样的输入设备，如鼠标、键盘、触摸屏、游戏手柄等。由于这些输入设备大部分不遵循统一的标准，所以导致应用程序，比如 X 将不得不处理来自各种输入

设备的五花八门的输入事件。

因此，内核中抽象了一个输入子系统。在输入子系统中，设备驱动面对各种各样具体的硬件设备，而输入事件经过事件处理模块处理后，将以统一的格式发送给用户空间的应用，用户空间的应用无需再为各种各样的输入事件格式疲于奔命。

现在很多输入设备都使用 USB 接口，对于 USB 接口的输入设备，图 6-12 演化为图 6-13 所示。

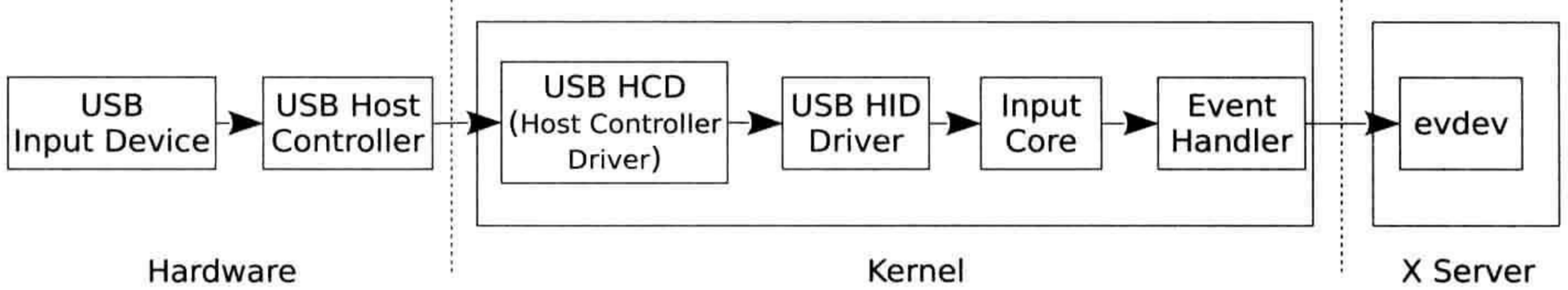


图 6-13 Linux USB 设备的输入子系统架构

USB 设备通过主控制器连接到主机，所以内核需要驱动 USB 主控制器。USB 流行的一个主要原因就是具有统一的标准，所以对于 USB 接口的输入设备，它们使用统一的设备驱动，即图 6-13 中的 USB HID 驱动。

通过上面的讨论可见，从操作系统的角度，安装 X 的输入设备驱动事实上有两件事需要做：一是需要配置内核的输入设备相关的驱动和模块；二是安装 X 的 evdev 模块。

1. 配置内核输入子系统相关驱动

如前文所说，现在大部分鼠标、键盘等输入设备都使用 USB 接口，所以这一节我们以 USB 接口的输入设备为例，来配置内核中的输入设备相关的驱动和模块。包括配置 USB 总线及控制器的驱动、USB 输入设备的驱动以及事件处理模块。

（1）配置 USB 总线以及 USB 主控制器驱动

配置 USB 总线及 USB 主控制器驱动的步骤如下：

1）执行 make menuconfig，出现如图 6-14 所示的界面。

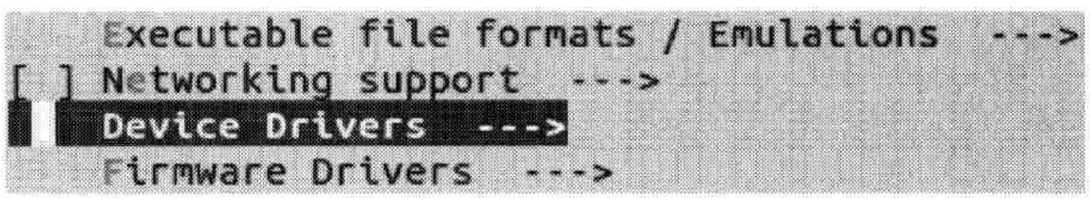


图 6-14 配置 USB 总线及 USB 主控制器驱动（1）

2）在图 6-14 中，选择菜单项“Device Drivers”，出现如图 6-15 所示的界面。

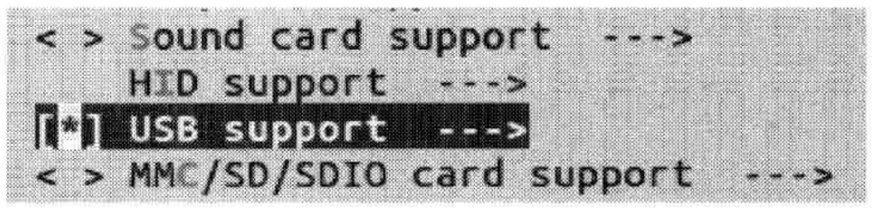


图 6-15 配置 USB 总线及 USB 主控制器驱动（2）

3）在图 6-15 中，选中菜单项“USB support”，出现如图 6-16 所示的界面。

```
--- USB support
<*>   Support for Host-side USB
[ ]     USB verbose debug messages (NEW)
[ ]     USB announce new devices (NEW)
        *** Miscellaneous USB options ***
[ ]     Dynamic USB minor allocation (NEW)
< >   USB Monitor (NEW)
< >   Support WUSB Cable Based Association (CBA) (NEW)
        *** USB Host Controller Drivers ***
< >   Cypress C67x00 HCD support (NEW)
<*>   xHCI HCD (USB 3.0) support
[ ]     Debugging for the xHCI host controller (NEW)
<*>   EHCI HCD (USB 2.0) support
[ ]   Root Hub Transaction Translators (NEW)
[*]   Improved Transaction Translator scheduling (NEW)
< >   OXU210HP HCD support (NEW)
< >   ISP116X HCD support (NEW)
< >   ISP 1760 HCD support (NEW)
< >   ISP1362 HCD support (NEW)
<*>   OHCI HCD support
[ ]     Generic OHCI driver for a platform device (NEW)
[ ]   Generic EHCI driver for a platform device (NEW)
<*>   UHCI HCD (most Intel and VIA) support
```

图 6-16　配置 USB 总线及 USB 主控制器驱动（3）

4）在图 6-16 中，选中“Support for Host-side USB”，并分别选中几个典型的 USB 主控制器的驱动，包括“xHCI HCD (USB 3.0) support”、“EHCI HCD (USB 2.0) support”、“OHCI HCD support”、“UHCI HCD (most Intel and VIA) support”。USB 总线及主控制器驱动配置完成。

（2）配置 USB 输入设备驱动

在配置了内核支持 USB 总线和主控制器之后，一般内核都会自动配置支持 USB HID 设备，至少我们使用的 3.7.4 版本的内核是这样的。如果读者使用了其他版本内核，请自己确认，配置过程为：首先进入“Device Drivers”界面，然后再进入“HID support”界面，查看 USB HID 是否已经被选中，一般情况下选中“Generic HID driver”即可。

（3）配置事件处理模块

1）执行 make menuconfig，出现如图 6-17 所示的界面。

```
    Executable file formats / Emulations  --->
[ ] Networking support  --->
    Device Drivers  --->
    Firmware Drivers  --->
```

图 6-17　配置事件处理模块（1）

2）在图 6-17 中，选择菜单项“Device Drivers”，出现如图 6-18 所示的界面。

```
[*] Network device support  --->
    Input device support  --->
    Character devices  --->
```

图 6-18　配置事件处理模块（2）

3）在图 6-18 中，选择菜单项“Input device support”，出现如图 6-19 所示的界面。

```
< >   Joystick interface
<*>   Event interface
< >   Event debugging
      *** Input Device Drivers ***
```

图 6-19 配置事件处理模块（3）

4）在图 6-19 中，选中“Event interface”，事件处理模块配置完成。

Linux 系统运行时，事件处理模块将为输入设备在 /dev/input 目录下建立相应的节点，一般形如 eventX，其中“X”是具体的数字。在调试 X 的鼠标、键盘、触摸屏等输入设备的驱动时，一旦遇到麻烦，可以先确认内核中的设备驱动和事件处理模块是否已经正确工作。方法之一就是通过直接读取这些输入设备的节点，命令如下（其中“X”根据具体的情况进行替换）：

```
cat /dev/input/eventX
```

然后操作鼠标或者键盘等输入设备，通过观察是否有数据输出，以确认内核的输入子系统部分是否已经正确工作。

2. 安装 evdev 模块

使用如下命令安装 X 服务器使用的 evdev 模块：

```
vita@baisheng:/vita/build$ tar xvf \
     ../source/X7.7/xf86-input-evdev-2.7.0.tar.bz2
vita@baisheng:/vita/build/xf86-input-evdev-2.7.0$ ./configure \
     --prefix=/usr
vita@baisheng:/vita/build/xf86-input-evdev-2.7.0$ make install
```

6.8.7 运行 X 服务器

X 服务器将建立一个套接字与应用程序进行通信，通常这个套接字被命名为“/tmp/.X11-unix/X0”，0 表示是第一个 X 服务器，如果再启动第二个 X 服务器，则为“/tmp/.X11-unix/X1”。除了建立套接字外，X 服务器还将在 /tmp 目录下建立一个锁文件，例如对于第一个 X 服务器，这个锁文件为“/tmp/.X0-lock”。另外，在前面编译时，我们指定 X 服务器将日志文件存放在 /var/log 目录下，因此，我们需要在根文件系统中建立这两个目录：

```
vita@baisheng:/vita/rootfs$ mkdir -p tmp var/log
```

为了使书中的截图不至于尺寸过大，笔者将 vita 系统的 X 服务器的分辨率设置为“640 × 480”。最初，X 服务器完全由用户通过书写配置文件的方式手动配置，在 udev 出现后，X 服务器采用了自动配置技术。但是 X 也给用户留有机会进行手动微调，并且用户手动配置的优先级还要更高。当然读者不必设置分辨率，由 X 服务器自动探测即可。通过 xorg.conf 设定分辨率的方法如下：

```
/vita/sysroot/etc/X11/xorg.conf:

Section "Screen"
    Identifier  "Screen0"
    SubSection  "Display"
        modes   "640x480"
    EndSubSection
EndSection
```

最初，X 服务器启动后将创建并显示鼠标指针。后来，X 的开发人员认为只有在应用程序明确表明需要与用户进行交互时，才应该显示鼠标指针。所以，这个默认行为发生了改变，在 X 服务器启动后，不再默认创建并显示鼠标指针，而是在第一个应用明确调用类似 XDefineCursor 这样的函数请求 X 服务器显示鼠标后，才显示鼠标指针。

但是 X 还是为用户留了余地，增加了一个命令行参数“-retro”。如果用户运行 X 服务器启动时即创建和显示鼠标，那么给 X 服务器传递这个参数即可。

在默认情况下，当最后一个 X 应用断开与 X 服务器的连接后，X 服务器默认自动重置。同样，X 也为这个行为提供了修正的机会，用户可以使用命令行参数“-noreset”关闭这个特性。vita 系统不需要这个特性，因此我们传递了“-noreset”参数给 X 服务器。

最后，使用如下命令运行 X 服务器：

```
root@vita:~# Xorg -retro -noreset &
```

在 X 服务器启动成功后，将创建一个根窗口，作为未来所有用户窗口的根。默认情况下，这个根窗口只以一个简单的灰色背景显示。并且我们看到，X 也按照我们的要求，创建并显示了鼠标指针，如图 6-20 所示。

图 6-20　X 服务器成功启动

6.8.8 一个简单的X程序

我们使用Xlib编写一个简单的X程序来确认X服务器是否已经正常工作。这个程序非常简单，就是创建一个窗口，并在其上显示字符串“Hello X Window !”，代码如下：

/vita/build/hello_x/hello_x.c:

```
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    Display *dpy;
    int screen_num;
    Window win;
    int x, y;
    unsigned int w, h;
    Atom atom_win_type, atom_win_type_normal;
    XEvent e;
    GC gc;
    char *s = "Hello X Window !";

    if (!(dpy = XOpenDisplay(NULL))) {
        fprintf(stderr, "Can't connect to X sever!\n");
        return -1;
}

screen_num = DefaultScreen(dpy);
x = y = 20;
w = DisplayWidth(dpy, screen_num) / 2;
h = DisplayHeight(dpy, screen_num) / 2;

win = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy),
        x, y, w, h, 2, BlackPixel(dpy, screen_num),
        WhitePixel(dpy, screen_num));

XStoreName(dpy, win, "Hello X11");

atom_win_type = XInternAtom(dpy, "_NET_WM_WINDOW_TYPE",
                    False);
atom_win_type_normal = XInternAtom(dpy,
            "_NET_WM_WINDOW_TYPE_NORMAL", False);
XChangeProperty(dpy, win, atom_win_type, XA_ATOM,
            32, PropModeReplace,
            (unsigned char *)&atom_win_type_normal, 1);

XSelectInput(dpy, win, ExposureMask);
gc = XCreateGC(dpy, win, 0, 0);
XMapWindow(dpy, win);

while (1) {
    XNextEvent(dpy, &e);
```

```
        switch (e.type) {
            case Expose:
                XDrawString(dpy, win, gc, 30, 30, s, strlen(s));
                break;
        }
    }
}
```

编译这个程序的 Makefile 如下：

```
/vita/build/hello_x/Makefile:

LDFLAGS=`pkg-config --libs x11`

hello_x: hello_x.o

clean:
    rm -rf hello_x  *.o
```

编译后通过 scp 命令将 hello_x 复制到 vita 系统，并通过 ssh 登录到 vita 系统，相应命令如下：

```
vita@baisheng:/vita/build/hello_x$ scp hello_x \
     root@192.168.56.2:/root/
     root@baisheng:~# ssh 192.168.56.2
```

在登录到 vita 的终端中，使用如下命令启动 X 服务器，并运行应用程序 hello_x：

```
root@vita:~# Xorg -retro &
root@vita:~# export DISPLAY=:0.0
root@vita:~# ./hello_x &
```

注意环境变量 DISPLAY 的设置，其格式如下：

```
hostname: displaynumber.screennumber
```

如果主机名（hostname）为空，则表示 X 服务器运行在本机。读者可以把 display 理解为一个 X 服务器，screen 这里无须解释。displaynumber 和 screennumber 均从 0 开始计数，如值为“:0.0”表示运行在本机的第一个 X 服务器接的第一块屏幕。vita 系统只启动了一个 X 服务器，并且只接一块屏。所以自然将环境变量 DISPLAY 设置为“:0.0”。

如果一切正常，则应用程序运行情况如图 6-21 所示。

6.8.9 配置内核支持 DRM

如果读者是在真实机器上调试的，那么为了使 GPU 的 2D 驱动和 3D 驱动都可以正常工作，内核中还需要进行相关的配置，因为用户空间的 GPU 驱动是通过内核中的 DRM 访问 GPU 的。GPU 用户空间的驱动（2D 和 3D 驱动）和内核空间的驱动（DRM 模块）之间的关系如图 6-22 所示。

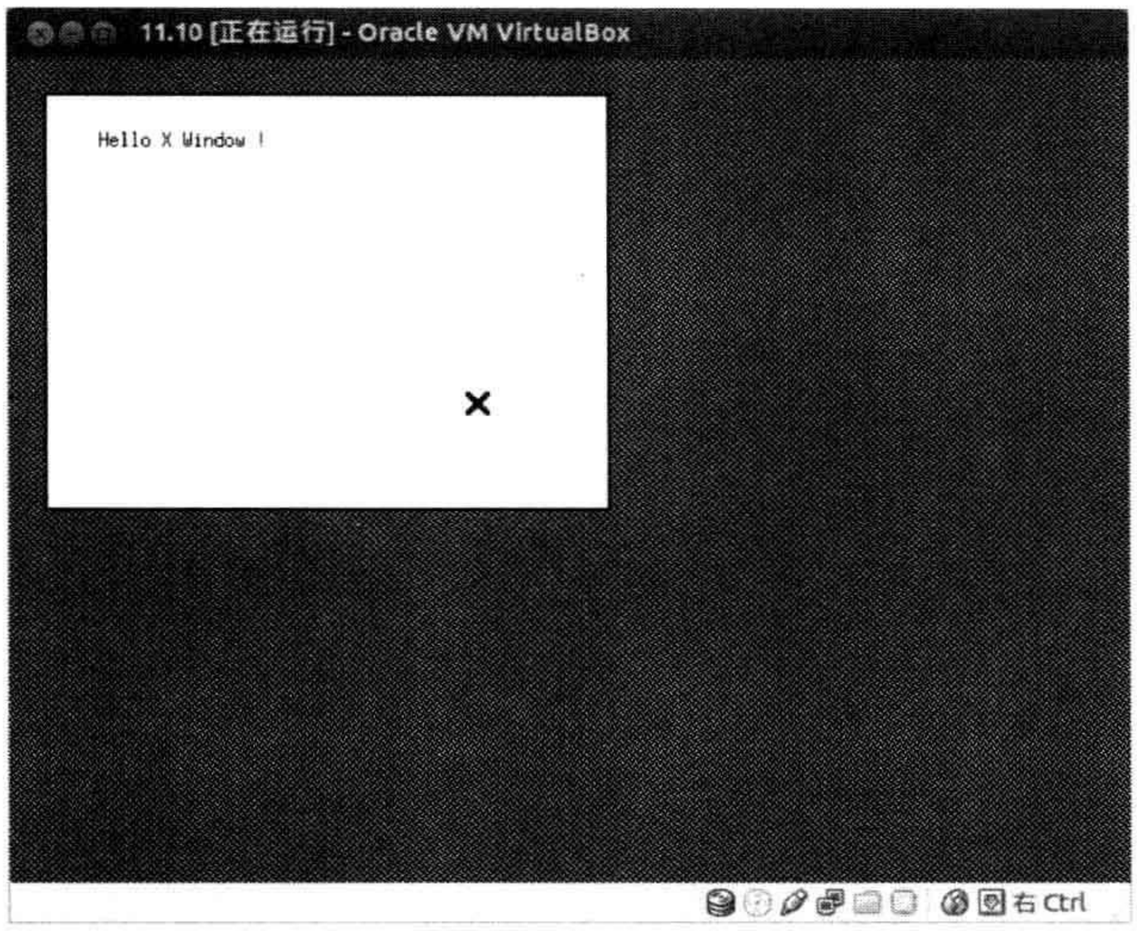


图 6-21　一个简单的使用 Xlib 编写的程序

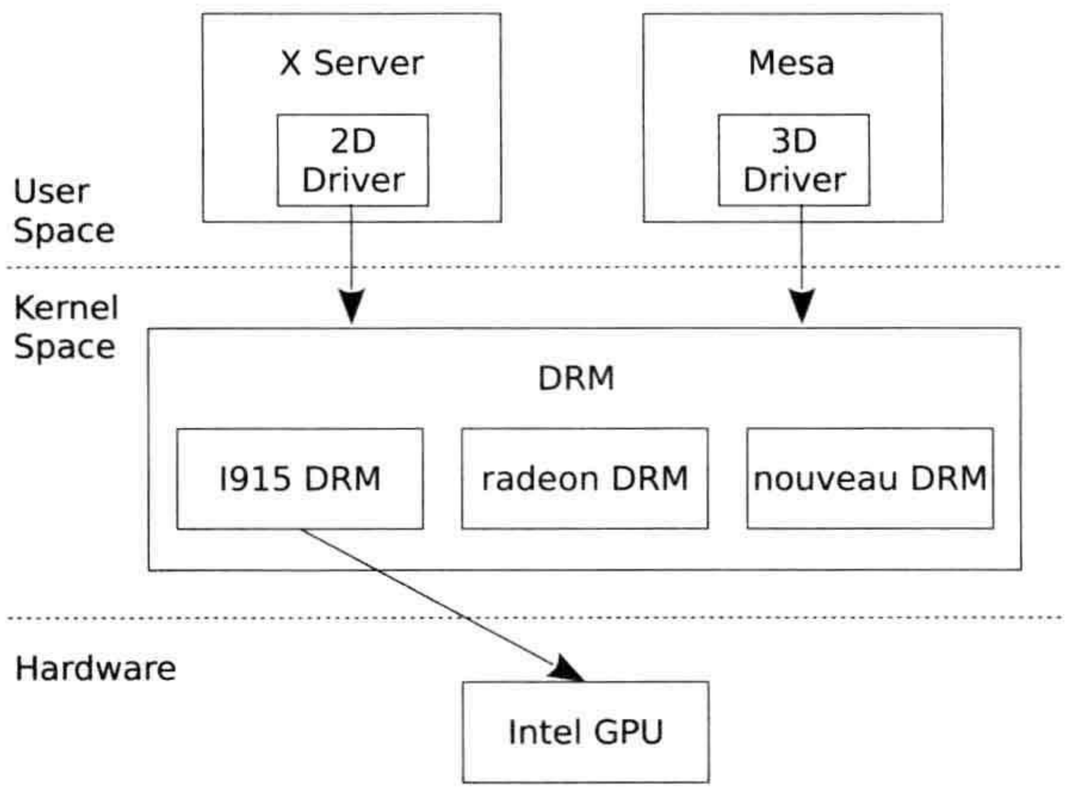


图 6-22　GPU 驱动各部分间的关系

在本节中，我们以 Intel 的 GPU 为例来讨论内核中关于 GPU 的相关配置。Intel 的 GPU 使用了 AGP 局部总线，所以在配置 GPU 的 DRM 模块前，需要首先配置内核支持 AGP 总线。

（1）配置 AGP 总线

配置 AGP 总线的步骤如下：

1）执行 make menuconfig，出现如图 6-23 所示的界面。

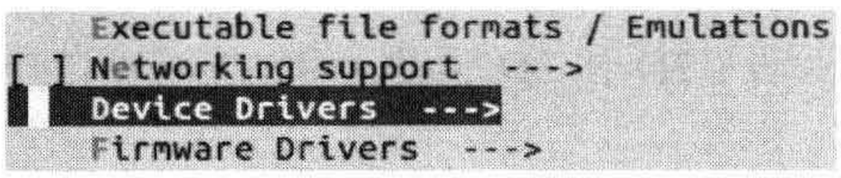


图 6-23　配置 AGP 总线（1）

2）在图 6-23 中，选择“Device Drivers”，出现如图 6-24 所示的界面。

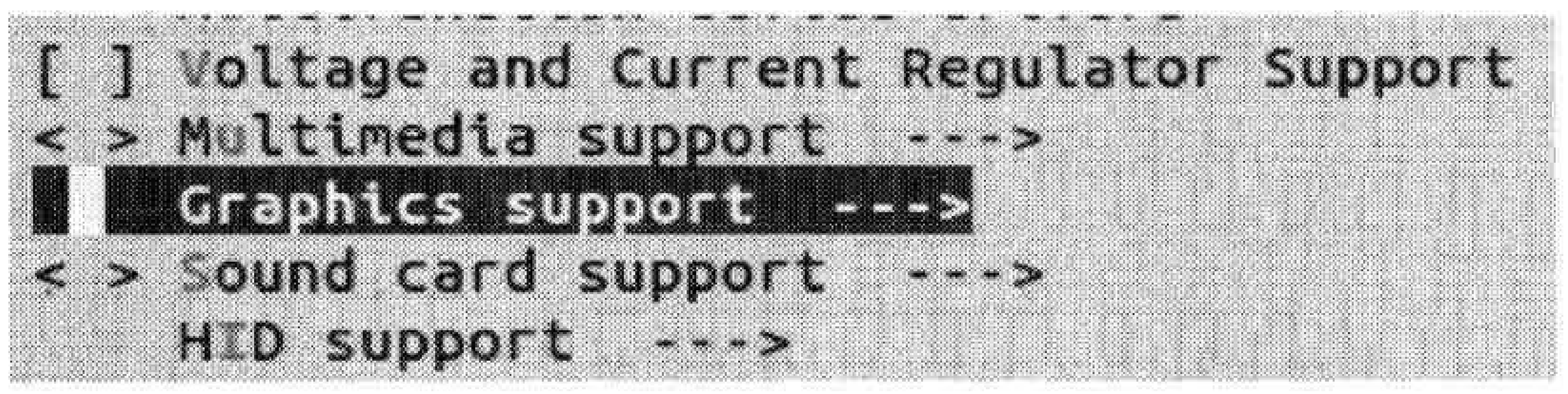


图 6-24　配置 AGP 总线（2）

3）在图 6-24 中，选择“Graphics support”，出现如图 6-25 所示的界面。

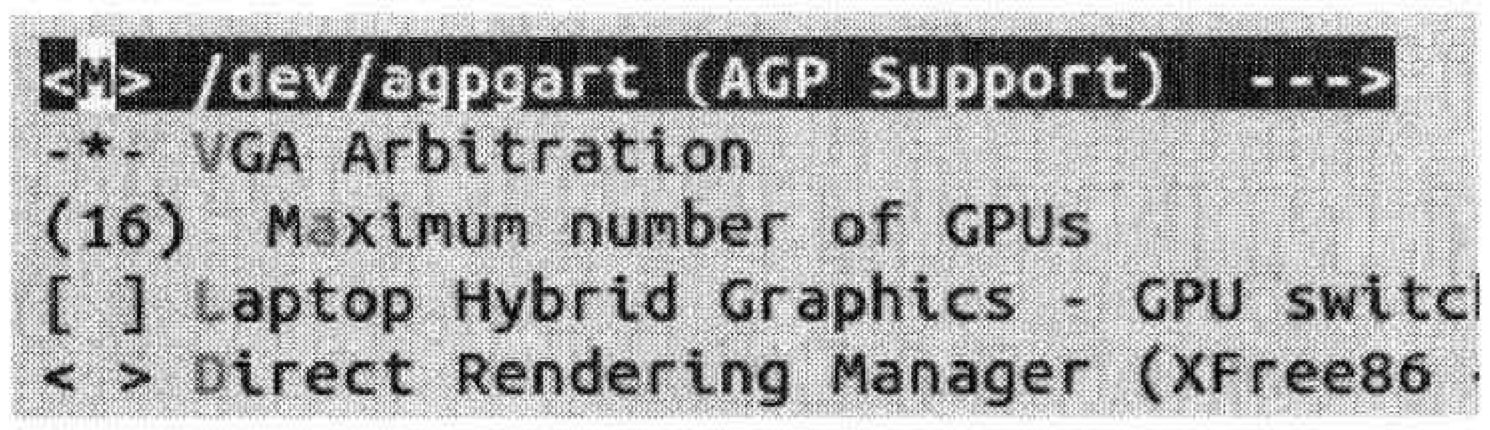


图 6-25　配置 AGP 总线（3）

4）在图 6-25 中，选中“/dev/agpgart (AGP Support)”，出现如图 6-26 所示的界面。

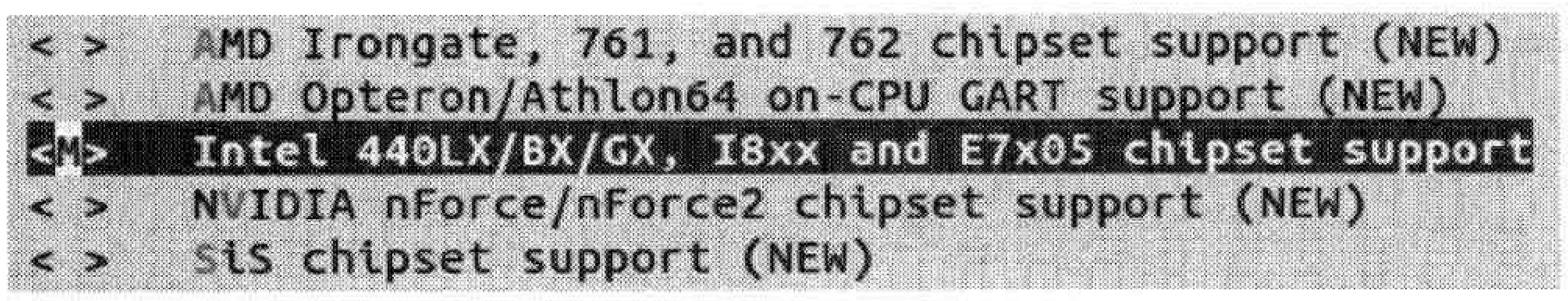


图 6-26　配置 AGP（4）

5）在图 6-26 中，选中“Intel 440LX/BX/GX, I8xx and E7x05 chipset support”。AGP 总线配置完毕。

（2）配置 DRM 模块

配置 DRM 的前两步与图 6-23 和 6-24 完全相同，在图 6-24 所示的界面选择“Graphics support”后，将出现如图 6-27 所示的界面。

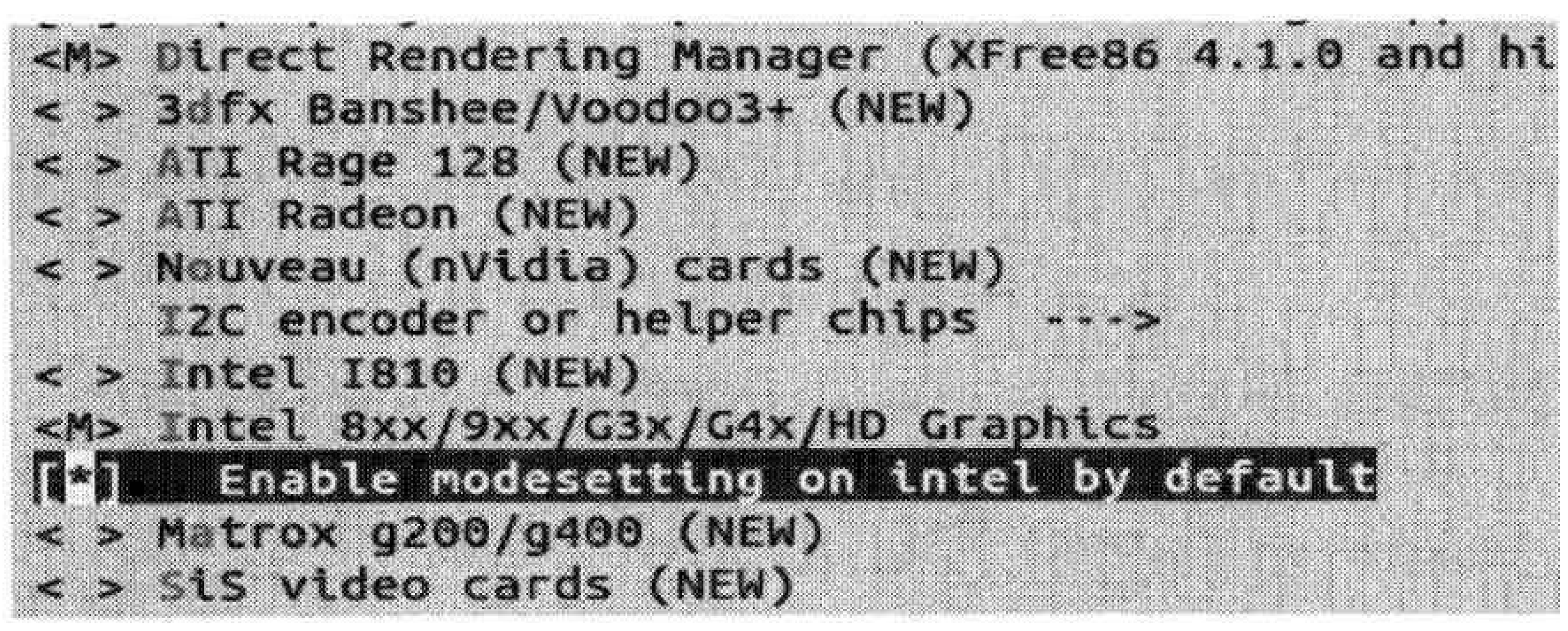


图 6-27　配置内核支持 DRM

在图 6-27 中，首先选中“Direct Rendering Manager”，表示需要内核支持 DRM，这是 DRM 的通用部分。然后还要选中驱动具体 GPU 的模块，比如适合笔者测试机器的 DRM 模块为“Intel 8xx/9xx/G3x/G4x/HD Graphics”，并且要勾选其下的子选项“Enable modesetting on intel by default”，这个特性就是所谓的 KMS（Kernel Mode Setting）。对于 Intel 的 GPU，这是

必须要选择的，否则用户空间的 2D 和 3D 驱动可能都会遇到麻烦。

一切配置好后，那么如何确定我们的配置是正确的，并且已经生效呢？我们可以按如下方法验证。

验证 2D 驱动

可以通过查看 X 的日志文件确认 GPU 的 2D 驱动是否已经被正确加载。比如在笔者使用的另外一台真实测试机上，Intel 的 2D 驱动成功加载后输出的初始化信息如下：

```
root@vita:~# cat /var/log/Xorg.0.log
...
[    48.560] (II) intel: Driver for Intel Integrated Graphics
Chipsets: i810,
...
[    49.277] (==) intel(0): Depth 24, (--) framebuffer bpp 32
[    49.287] (==) intel(0): RGB weight 888
[    49.287] (==) intel(0): Default visual is TrueColor
[    49.287] (II) intel(0): Integrated Graphics Chipset: Intel(R) 945GME
[    49.288] (--) intel(0): Chipset: "945GME"
...
 [     49.335] (II) intel(0): Modeline "800x600"x56.2    36.00   800 824 896 1024
600 601 603 625 +hsync +vsync (35.2 kHz d)
[    49.381] (II) intel(0): EDID for output VGA1
[    49.381] (II) intel(0): Output LVDS1 connected
[    49.381] (II) intel(0): Output VGA1 disconnected
...
```

验证 3D 驱动

可以使用 Mesa 提供的工具，如 glxinfo，查看 GPU 的 3D 驱动是否已经正确加载。读者可以从 Mesa 的网站下载并编译工具 glxinfo，或者偷个懒，从宿主系统复制过来一个，只要版本差别不是特别大，一般都可以正常运行。可以远程登录，或者在测试机器的本地运行一个终端，使用 glxinfo 命令查看 3D 驱动是否已经正常工作：

```
root@vita:~# export DISPLAY=:0.0
root@vita:~# ./glxinfo
...
OpenGL renderer string: Mesa DRI Intel(R) 945GME x86/MMX/SSE2
...
```

再比如在笔者的宿主机上使用命令 glxinfo 查看，显示如下：

```
root@baisheng:~# glxinfo | grep "renderer string"
OpenGL renderer string: Mesa DRI Intel(R) Sandybridge Mobile
x86/MMX/SSE2
```

仔细观察 glxinfo 的输出，可以看到在 3D 渲染时，使用的是 Intel GPU 的 3D 驱动。如果是软件渲染，则输出类似如下：

```
OpenGL renderer string: Software Rasterizer
```

6.9 安装图形库

前面，我们使用 Xlib 编写了一个小程序。但是我们也看到，Xlib 是多么的原始，使用 X 提供的库编写一个如此简单的程序是多么的复杂，更别提具有复杂图形用户界面的程序了。所以先辈开发者们前赴后继，尝试在 Xlib 的基础上为 X 开发更高级的图形库，这些图形库通常被称为 Widget Libraries 或 Toolkits，其中最著名的就是 GTK 和 QT。这些图形库引入了控件的概念，极大简化了程序开发，也提高了开发效率。

我们选择 GTK 作为 vita 系统的图形库。这一节，我们就来编译安装 GTK。相比于安装 X，图形库的安装过程相对要简单，但是我们也提供了一个编译脚本 build-gtk.sh。必要时，读者可以参考这个脚本。

6.9.1 安装 GLib 和 libffi

GLib 是 GTK+ 和 GNOME 工程的基础底层核心程序库，是一个实用的轻量级的库，它提供常用的数据结构、相关的处理函数和一些运行时支承机制，如事件循环、线程、对象系统等。因此安装 GTK+ 前首先需要安装 GLib。GLib 目前也由开发 GTK+ 的团队维护。

因为 GLib 提供的对象系统（GObject）可以绑定到多种语言，常见的如 C、Python、Ruby 等，因此，GLib 的对象系统借助库 libffi 处理不同语言间的函数调用。libffi 是专门设计的一个库，主要用于不同语言间的相互调用。因此，安装 GLib 前还需要安装 libffi。

libffi 和 GLib 的编译安装命令如下：

```
vita@baisheng:/vita/build$ tar xvf ../source/libffi-3.0.11.tar.gz
vita@baisheng:/vita/build/libffi-3.0.11$ ./configure \
     --prefix=/usr
vita@baisheng:/vita/build/libffi-3.0.11$ make install
vita@baisheng:/vita/build/libffi-3.0.11$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;

vita@baisheng:/vita/build$ tar xvf ../source/glib-2.32.4.tar.xz`
vita@baisheng:/vita/build/glib-2.32.4$ ./configure --prefix=/usr
vita@baisheng:/vita/build/glib-2.32.4$ make install
vita@baisheng:/vita/build/glib-2.32.4$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;
```

6.9.2 安装 ATK

ATK（Accessibility ToolKit）是 GTK 中实现辅助功能使用的库，包括辅助视觉、听觉、打字等。这个库也是别无选择，必须要安装的，安装命令如下：

```
vita@baisheng:/vita/build$ tar xvf ../source/atk-2.4.0.tar.xz
vita@baisheng:/vita/build/atk-2.4.0$ ./configure --prefix=/usr
vita@baisheng:/vita/build/atk-2.4.0$ make install
vita@baisheng:/vita/build/atk-2.4.0$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;
```

6.9.3 安装 libpng

图形库当然离不开图片格式处理的库，常用的图片格式有多种，比如 PNG、JPEG 等。但是为了简单起见，vita 系统只支持 PNG 图片格式。处理 PNG 图形格式的库是 libpng，安装命令如下：

```
vita@baisheng:/vita/build$ tar xvf ../source/libpng-1.5.12.tar.xz
vita@baisheng:/vita/build/libpng-1.5.12$ ./configure \
     --prefix=/usr
vita@baisheng:/vita/build/libpng-1.5.12$ make install
vita@baisheng:/vita/build/libpng-1.5.12$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;
```

6.9.4 安装 GdkPixbuf

GTK 使用 GdkPixbuf 进行图片的渲染，是 GTK 图形库的基本依赖之一，是必须安装的，安装命令如下：

```
vita@baisheng:/vita/build$ tar xvf \
     ../source/gdk-pixbuf-2.26.3.tar.xz
vita@baisheng:/vita/build/gdk-pixbuf-2.26.3$ patch -p1 \
     < ../../source/gdk-pixbuf-2.26.3-disable-test.patch
vita@baisheng:/vita/build/gdk-pixbuf-2.26.3$ ./configure \
     --prefix=/usr --without-libtiff --without-libjpeg
vita@baisheng:/vita/build/gdk-pixbuf-2.26.3$ make install
vita@baisheng:/vita/build/gdk-pixbuf-2.26.3$ find $SYSROOT \
     -name  "*.la" -exec rm -f '{}' \;
```

为简单起见，我们禁掉了其对 JPEG 和 TIFF 格式的支持，否则，还需要安装如库 libpng 一样操作 JPEG 和 TIFF 格式的库。所以，读者在 vita 上使用 GTK 开发程序时，也不要使用 JPEG 和 TIFF 格式的图片。当然除了 PNG 格式，GdkPixbuf 也默认支持其他一些格式，在安装后，读者可以使用如下命令查看其支持的图片格式：

```
vita@baisheng:/vita$ ls \
     /vita/sysroot/usr/lib/gdk-pixbuf-2.0/2.10.0/loaders/

libpixbufloader-ani.so  libpixbufloader-icns.so
libpixbufloader-png.so  libpixbufloader-ras.so
libpixbufloader-xbm.so libpixbufloader-bmp.so
...
libpixbufloader-wbmp.so
```

6.9.5 安装 Fontconfig

Linux 最初在我国的程序员中流行时，有很多程序员热衷于 Linux 的美化，其中优化文字的显示是其中主要内容之一，至今在各个 Linux 论坛仍然可见 Linux 美化的身影。文本渲染比较烦琐，除了技术原因外，文本处理机制不断的发展变化，从最初的 X 的核心字体，到 X 的字体服务器，再到现在广泛采用的客户端渲染，也给这个本身就不是特别容易理解的领

域增加了很多复杂性。

凡是涉及字体相关的地方，我们经常看到如 Fontconfig、Freetype、Pango，甚至更多，这些库在文本渲染中都担任什么角色？它们之间的关系又是什么？在我们埋头搭建系统时，还是要不时抬头看看路的。下面，我们就结合图 6-28 来简单地介绍一下文本的渲染。

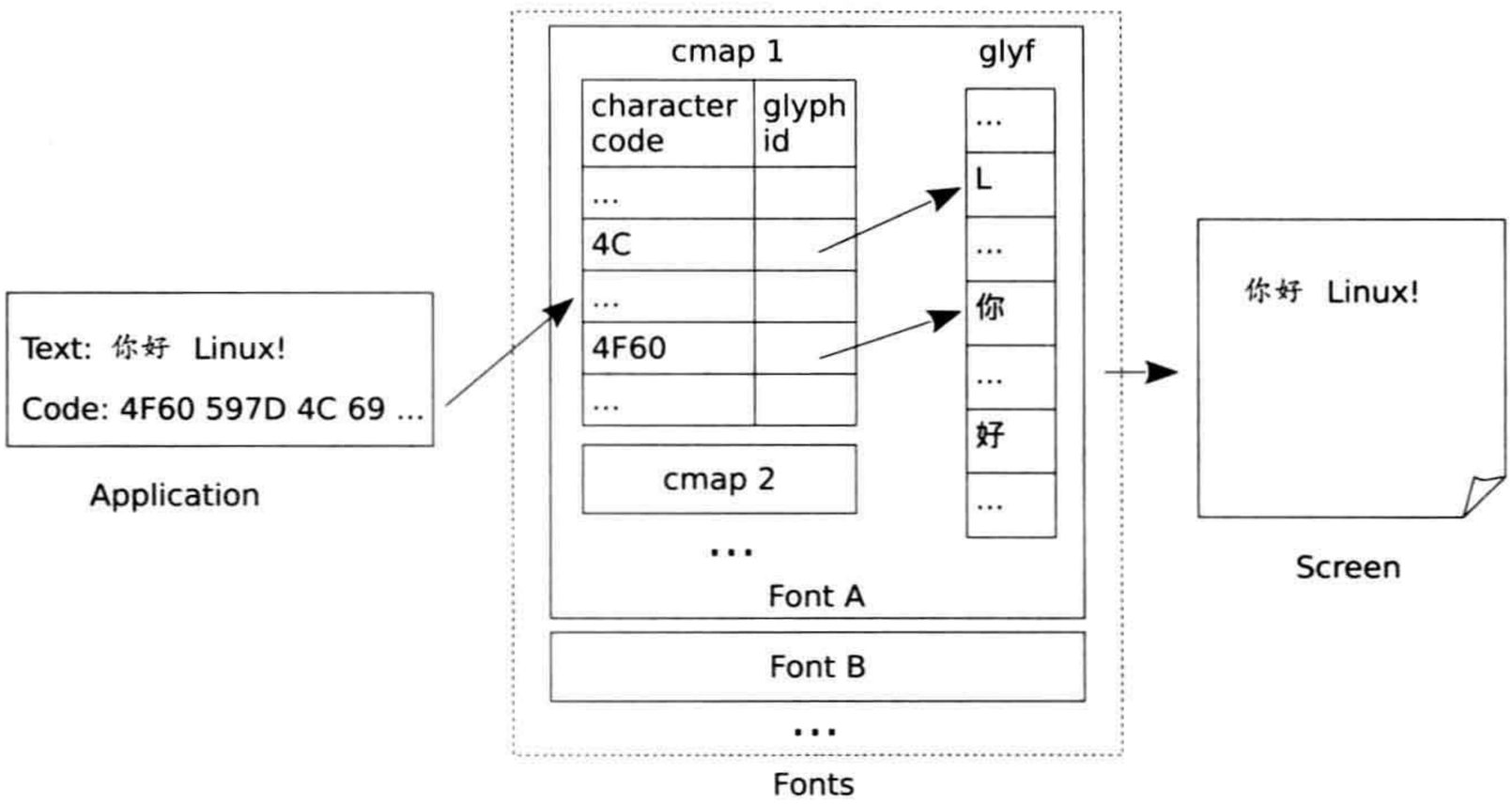


图 6-28 文本渲染过程示意图

（1）字符编码（character code）

虽然我们在写程序时，直接使用可读的字符，但事实上，在程序内部，是用字符的编码来代表字符的。字符的编码有多种标准，比如 ISO-8859 系列编码，Unicode 编码以及我国的 GB18030 等。

假设系统使用 UTF8 编码，当程序准备显示字符串“你好 Linux!”时，程序中将以编码“4F60 597D 4C 69 …”来记录这个字符串。

（2）字形（glyph）

字形是字的形体的简称，GB/T 16964《信息技术 · 字型信息交换》中关于字形的的定义为：一个可以辨认的抽象的图形符号，它不依赖于任何特定的设计。

这样解释读者可能依然会感到比较生疏，因为平时我们很少使用这个概念，但是提到字体，大家就一定比较熟悉了，因为操作系统中一定要安装字体文件的，否则是不能正确显示字符的。而所谓的这个字体文件，其实就是字形的集合。

以 TrueType 字体文件为例，其中包含两个关键的数据结构：

- 一个是字形表，也称为 glyf。字形表中每一项代表一个字形，使用字形索引访问其中的字形。TrueType 的字形表中，每个字形的描述并非如图 6-28 中的字形表（glyf）中显示的那样直观，字形表中描述的字形信息都是矢量的，字符的每一个笔画都是由多条曲线包围而形成的。一次曲线需要两个点来确定，二次需要三个点，三次就需要四个点。字体内部保存了这些点的坐标。

- 一个是字符编码到字形映射表（Character to Glyph Mapping），简称 cmap。读者可能会有个疑问，cmap 中的第二列为什么不是字形，而是字形索引呢？原因是字体文件可能使用在不同的编码环境中，所以字体文件可能包含多个 cmap 表，比如 UTF8 对应一个 cmap 表，GB18030 对应另外一个 cmap 表。另外，一个字体文件中也可能不只包含一种字体。

（3）排版（layout）

每每谈到文本渲染时，大家更多的关注在字体上，却往往忽略了文本的布局排版。实际上，文字的排版是重要而且复杂的。排版引擎需要将单个字符按照一定的间距美观的排列起来。

除了处理字形信息外，由于世界上有多种文字体系，因此，文本可能是多种语言混合的。而且，还有像阿拉伯文、希伯来文这种文字体系是从右向左书写，更别提布局规则极其复杂的印度系文字。

可见，排版引擎是一位真正的幕后英雄。而且，文本渲染的过程都是由排版引擎牵头开始的，不同的图形库可能使用不同的排版引擎，GTK 使用的排版引擎是 Pango。

（4）确定字体

在将字符编码转化为字符前，首先需要确定字体文件，否则巧妇也难为无米之炊。一个系统中可能安装了多个字体文件，因此，在众多的字体文件中要选择一个最合适的，这就是 Fontconfig 的主要任务之一。

进行文本渲染时，Pango 收集来自各方的字体信息，如系统主题中设置如下：

```
xxx {
    ...
    font: Italic 18;
    ...
}
```

程序自身的设置如下：

```
pango_font_description_from_string( "Serif bold 12" );
```

可能还有来自如图形库等其他方面的信息，总之，Pango 最后加工出一个字体描述，将它传递给 Fontconfig，这个描述称为模式（Pattern），Fontconfig 根据配置文件对这个模式进行进一步加工，Fontconfig 通常会修改或者增加一些属性。

最终，Fontconfig 以加工好的模式在众多的字体中匹配一个最合适的字体。

（5）光栅化

一旦字体确定后，Fontconfig 使用库 Freetype 提供的接口，确定字符编码对应的字形索引，依据的就是如 TrueType 字体文件中的 cmap 表。最后，Freetype 根据字形索引，从字体文件的字形表中获取描述字形的矢量信息，构建具体的字形，这个过程也叫光栅化。经过光栅化的字符编码，就是一普通图形了，接下来无论是显示到具体窗口中，还是进行其他处

理，都与处理普通的图形完全相同。

理解了各个库的作用后，下面我们开始安装这些库。

Freetype 在前面安装 X 时已经安装，接下来只需安装 Fontconfig 和 Pango。由于 Cairo 依赖于 Fontconfig，而 Pango 又基于 Cairo 进行字体渲染，所以，这里的安装顺序看上去有点奇怪。我们先安装 Fontconfig，中间插播 Cairo，然后才安装 Pango。

安装 Fontconfig 的命令如下：

```
vita@baisheng:/vita/build$ tar xvf \
     ../source/fontconfig-2.10.1.tar.bz2
vita@baisheng:/vita/build/fontconfig-2.10.1$ ./configure \
     --prefix=/usr --sysconfdir=/etc --localstatedir=/var \
     --disable-docs --without-add-fonts
vita@baisheng:/vita/build/fontconfig-2.10.1$ make install
vita@baisheng:/vita/build/fontconfig-2.10.1$ find $SYSROOT \
     -name "*.la" -exec rm -f '{}' \;
```

6.9.6 安装 Cairo

Cairo 是一个矢量图形库，GTK 使用其作为绘制后端。换句话说，GTK 的绘制动作由 Cairo 完成。看到这里，读者可能会非常困惑：X 上的应用不是由 X 服务器负责绘制吗？没错，暂且不提我们第 8 章讨论的 DRI。事实上，即使普通的 2D 应用也是可以自己绘制的，只不过，应用是将内容绘制在一个离屏的区域，但是最后还是要请求 X 服务器将绘制的内容显示到屏幕上。应用或者将绘制的内容复制到 X 服务器，或者使用 X 提供的 RENDER 扩展。当然，应用也可以将全部绘制请求 X 服务器完成，这就要看具体图形库采用的策略了。

安装 Cairo 的命令如下：

```
vita@baisheng:/vita/build$ tar xvf ../source/cairo-1.12.2.tar.xz
vita@baisheng:/vita/build/cairo-1.12.2$ ./configure \
     --prefix=/usr
vita@baisheng:/vita/build/cairo-1.12.2$ make install
vita@baisheng:/vita/build/cairo-1.12.2$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;
```

6.9.7 安装 Pango

安装 Pango 的命令如下：

```
vita@baisheng:/vita/build$ tar xvf ../source/pango-1.30.1.tar.xz
vita@baisheng:/vita/build/pango-1.30.1$ ./configure \
     --prefix=/usr --sysconfdir=/etc
vita@baisheng:/vita/build/pango-1.30.1$ make install
vita@baisheng:/vita/build/pango-1.30.1$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;
```

6.9.8 安装 libXi

图形库当然是要接收用户输入的，X 输入扩展协议的实现是库 libXi，安装命令如下：

```
vita@baisheng:/vita/build$ tar xvf \
     ../source/X7.7/libXi-1.6.1.tar.bz2
vita@baisheng:/vita/build/libXi-1.6.1$ ./configure --prefix=/usr
vita@baisheng:/vita/build/libXi-1.6.1$ make install
vita@baisheng:/vita/build/libXi-1.6.1$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;
```

6.9.9 安装 GTK

GTK 的基本依赖已经安装完成，只差完成最后一步了，安装 GTK 的命令如下：

```
vita@baisheng:/vita/build$ tar xvf ../source/gtk+-3.4.4.tar.xz
vita@baisheng:/vita/build/gtk+-3.4.4$ ./configure \
     --prefix=/usr --sysconfdir=/etc
vita@baisheng:/vita/build/gtk+-3.4.4$ make install
vita@baisheng:/vita/build/gtk+-3.4.4$ find $SYSROOT -name \
     "*.la" -exec rm -f '{}' \;
```

至此，图形库 GTK 的安装过程已经全部完成，读者可以将 /vita/sysroot 目录下的文件系统更新到 vita 的根文件系统了。

6.9.10 安装 GTK 图形库的善后工作

更新了 vita 系统的根文件系统后，在运行使用 GTK 编写的程序前，我们还要在 vita 系统上为图形库做一点收尾工作。注意下面两个操作需要在使用安装了 GTK 图形库的根文件系统重启 vita 系统后进行。

（1）为 Pango 创建语系和模块对应关系的文件

不同语系，对布局有不同的要求，全世界有各种各样的语系，如汉语、阿拉伯语、印度语等。Pango 采用模块化的方式提供对这些语系的支持。为了提高效率，在运行时，Pango 不会到文件系统中解析具体的模块，查看其支持的语系，而是直接读取 /etc/pango 目录下的文件 pango.modules，其中记录了每个模块及其支持的语系。因此，我们需要为 Pango 创建文件 pango.modules，命令如下：

```
root@vita:~# pango-querymodules > /etc/pango/pango.modules
```

下面是创建的文件 pango.modules 中的片段：

```
root@vita:~# cat /etc/pango/pango.modules

/usr/lib/pango/1.6.0/modules/pango-syriac-fc.so \
SyriacScriptEngineFc PangoEngineShape PangoRenderFc syriac:*
...
/usr/lib/pango/1.6.0/modules/pango-basic-fc.so \
BasicScriptEngineFc PangoEngineShape PangoRenderFc latin:* \
cyrillic:* greek:* armenian:* georgian:* runic:* ogham:* \
bopomofo:* cherokee:* coptic:* deseret:* ethiopic:* gothic:* \
han:* hiragana:* katakana:* old-italic:* canadian-aboriginal:*\
yi:* braille:* cypriot:* limbu:* osmanya:* shavian:* linear-b:*\
```

```
ugaritic:* glagolitic:* cuneiform:* phoenician:* common:
```

可见，模块 pango-syriac-fc.so 负责处理叙利亚语（syriac），模块 pango-basic-fc.so 负责处理拉丁语（latin）、希腊语（greek），包括我们的汉语（han）。

（2）为库 GdkPixbuf 创建模块信息文件

在安装库 GdkPixbuf 时，我们看到，GdkPixbuf 使用模块的形式支持各图形格式。因此，在这个库初始化时，需要加载这些模块。但是这些模块存储在文件系统的什么位置，每个模块又支持什么图形格式等，诸如此类信息从哪里获取呢？为了提高加载速度，GdkPixbuf 没有去再次扫描每个模块，而是直接从系统的一个文件中读取，因此，我们需要为 GdkPixbuf 创建这个文件，命令如下：

```
root@vita:~# gdk-pixbuf-query-loaders --update-cache
```

gdk-pixbuf-query-loaders 将到模块所在的目录去扫描各个模块，然后将模块信息记录下来，默认情况下，记录在下面这个文件中：

```
/usr/lib/gdk-pixbuf-2.0/2.10.0/loaders.cache
```

当然如果将模块信息写到其他文件了，需要通过环境变量 GDK_PIXBUF_MODULE_FILE 指定出来。

以 vita 系统为例，该文件中记录的信息大致如下：

```
root@vita:~# cat /usr/lib/gdk-pixbuf-2.0/2.10.0/loaders.cache
...
"/usr/lib/gdk-pixbuf-2.0/2.10.0/loaders/libpixbufloader-png.so"
"png" 5 "gdk-pixbuf" "The PNG image format" "LGPL"
"image/png" ""
"png" ""
"\211PNG\r\n\032\n" "" 100
...
```

6.9.11 一个简单的 GTK 程序

最后，我们使用一个简单的程序来测试我们的 GTK 是否工作正常，程序代码如下：

hello_gtk/hello_gtk.c:

```
#include <gtk/gtk.h>

int main(int argc, char *argv[] )
{
    GtkWidget *window;
    GtkWidget *lbl;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_default_size(GTK_WINDOW(window), 300, 200);
```

```
    lbl = gtk_label_new("Hello GTK!");
    gtk_container_add(GTK_CONTAINER(window), lbl);
    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

编译该程序的 Makefile 文件如下：

```
hello_gtk/Makefile:

CFLAGS=`pkg-config --cflags gtk+-3.0`
LDFLAGS=`pkg-config --cflags gtk+-3.0`

hello_gtk: hello_gtk.o

install:
    install -m 755 hello_gtk $(DESTDIR)/usr/bin/

clean:
    rm -rf hello_gtk *.o
```

可见，同样是显示一个简单的窗口，使用 GTK 编写就简单多了，那些烦琐的细节已经实现在如 GTK 等这些图形库中。编译这个程序，并将其复制到 vita 系统并运行，步骤与程序 hello_x 完全相同。

如果 GTK 安装正常，在 vita 系统上我们将看到类似图 6-29 所示的输出。

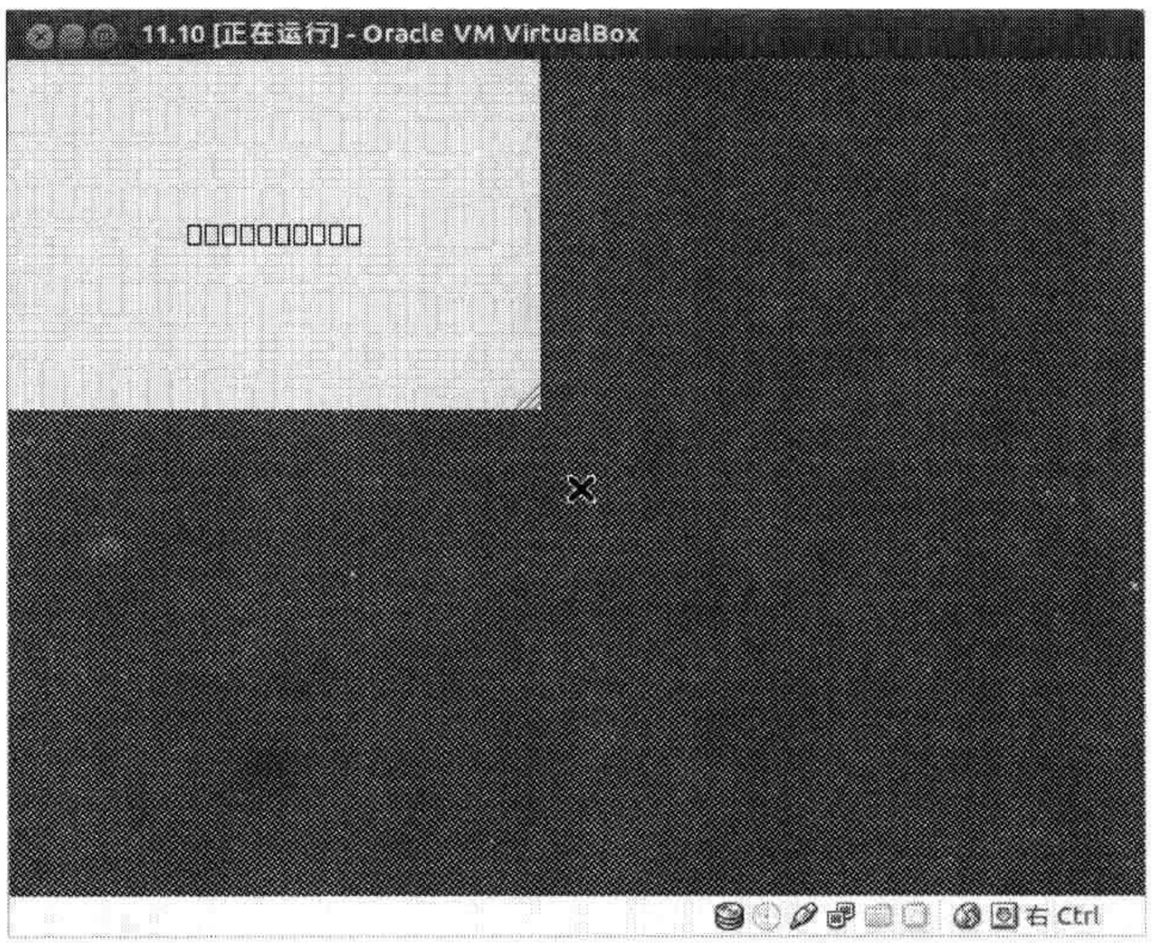


图 6-29 一个简单的 GTK 程序

但是，我们发现麻烦来了：文本并没有显示出来，而是显示了一串“方框”。如果对字体渲染有一些经验就会知道，这里所谓的“方框”是在找不到字体时使用的默认的一个特殊

字符。

观察终端上 Pango 输出：

```
root@vita:~# hello_gtk &

 (hello_gtk:249): Pango-WARNING **: failed to choose a font, expect ugly output.
engine-type='PangoRenderFc', script='latin'
```

Pango 的输出也印证了我们的推论，Pango 明确提示，找不到匹配的字体。

读者可能会再次陷入了困惑中：在测试 hello_x 时，为什么 hello_x 就可以找到字体呢？下一节，我们就来讨论这个问题，并为 vita 系统安装字体。

6.10 安装字体

对于基于 Xlib 编写的程序，一般简单的字符使用 X 中的内置字体就可以应付了。X 的内置字体在 libXfont 中：

```
libXfont-1.4.5/src/built-ins/fonts.c:

static const char file_6x13[] = {
    '\037', '\213', '\010', '\010', '\126', '\121', '\054', '\100',
    '\000', '\003', '\066', '\170', '\061', '\063', '\055', '\111',
    ...
    '\002', '\174', '\155', '\142', '\140', '\200', '\114', '\000',
    '\000',
};
```

其中 file_6x13 中记录的就是简单的点阵字体，又称位图字体，显然这个内置的点阵字体是把每一个字符都分成 6×23 个点，然后用每个点的虚实来表示字符的轮廓。

这也是为什么前面在没有安装字体的情况下，使用 Xlib 编写的例子可以显示字符的原因。但是既然有内置的字体，那为什么使用 GTK 的程序不能显示字符呢？原因是 GTK 程序的字体是在客户端绘制的，客户端绘制完成后，将字形位图传给 X 服务器。而 GTK 中并没有像 libXfont 那样内置了字体，所以如果系统中没有安装字体，当然应用就找不到字体了。因此，我们需要安装字体。

字体的安装非常简单，直接把字体文件复制到相关的目录下即可。但是安装在哪个目录下呢？前面我们已经看到，Linux 使用 Fontconfig 寻找字体，因此这个问题要问 Fontconfig。没错，Fontconfig 在其配置文件中明确指明了其寻找字体文件的目录：

```
/vita/sysroot/etc/fonts/fonts.conf:

<!-- Font directory list -->

    <dir>/usr/share/fonts</dir>

    <dir prefix="xdg">fonts</dir>
```

```
    <!-- the following element will be removed in the future -->
    <dir>~/.fonts</dir>
```

这里，我们使用文泉驿字体，并将其安装到 vita 系统的 /usr/share/fonts 目录下，命令如下：

```
root@vita:~# mkdir -p /usr/share/fonts/
root@baisheng:~# scp \
     /usr/share/fonts/truetype/wqy/wqy-microhei.ttc \
     192.168.56.2:/usr/share/fonts/
```

安装完字体后，再次执行 gtk_hello，就会发现字符不再是一个一个的“方框”了，如图 6-30 所示。

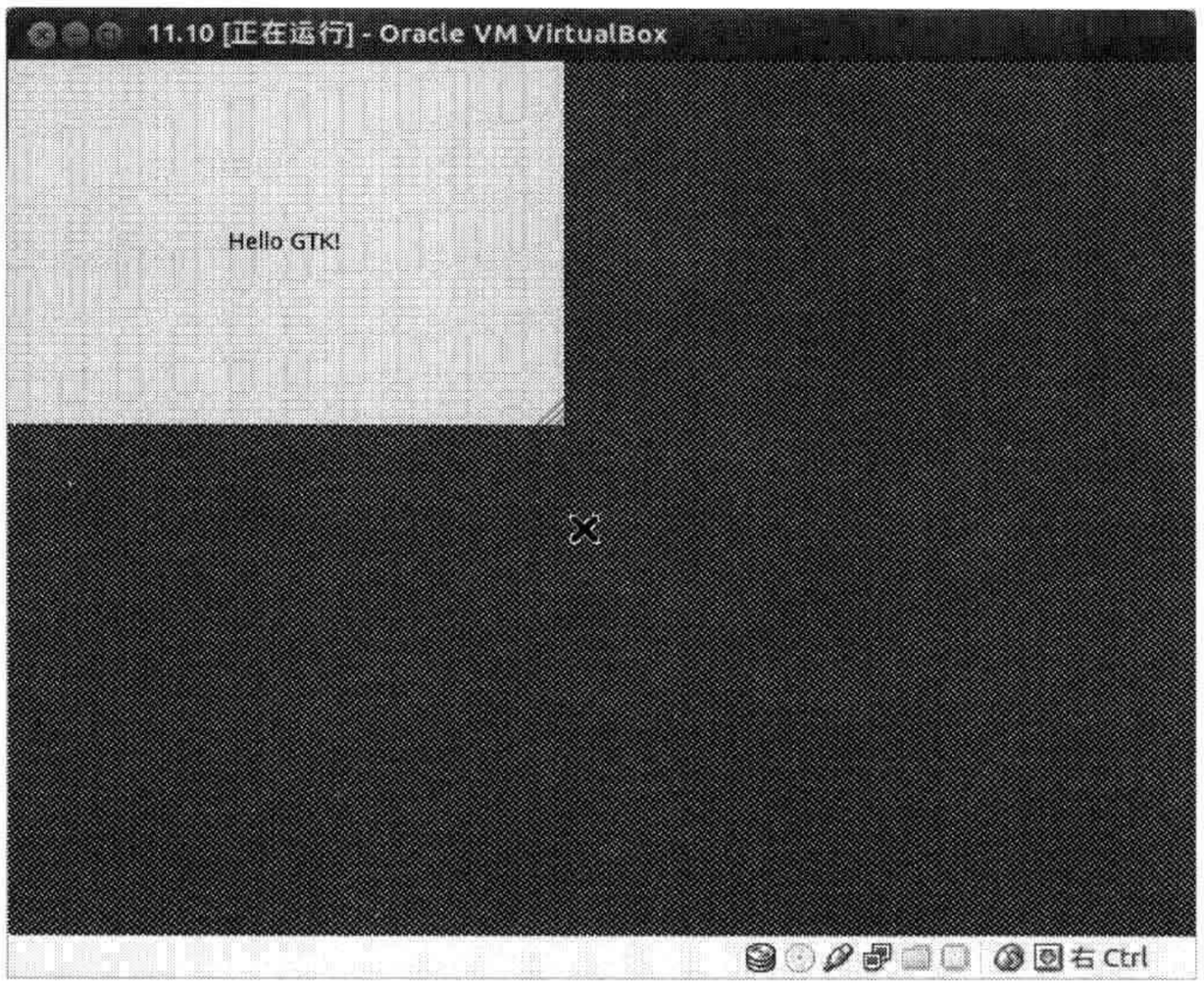


图 6-30 安装字体后的 GTK 程序

而且，可以在 vita 系统的 /var/cache/fontconfig 目录下看到类似如下的 Fontconfig 用于快速搜索字体的缓存文件：

```
root@vita:~# ls /var/cache/fontconfig/
3830d5c3ddfd5cd38a049b759396e72e-le32d4.cache-3
99e8ed0e538f840c565b6ed5dad60d56-le32d4.cache-3
7ef2298fde41cc6eeb7af42e48b7d293-le32d4.cache-3
```

至此，基础的图形环境已经安装完毕，可以运行基本的具有图形界面的程序了。但是我们也看到，这个图形环境是个裸环境，没有任务条、没有桌面背景。应用程序的窗口也是个裸窗口，没有标题栏、没有边框，不能最大化和最小化、不能关闭，也不能移动，甚至当启动多个应用时，我们也没有办法在多个应用间切换等。这些问题，我们留给下一章。

第 7 章

构建桌面环境

计算机领域中的桌面环境（Desktop Environment）其实是一种比喻的说法，即图形用户界面就像物理书桌一样，其上可以放置文件夹、文档等。桌面最初用来特指个人计算机（PC），但是现在不只个人计算机有图形界面环境，服务器、嵌入式设备等基本都提供桌面环境。桌面环境包括窗口管理器、任务条等基本组件，除了这些基本的组件外，有的桌面环境还提供文件管理器、控制面板等。

桌面环境是操作系统中人机交互的关键部分，理解它的基本运作原理，无论是对理解操作系统，还是对开发应用程序，都有极大的帮助。我们处于这样一个追求个性的年代，无论是用于消费类电子设备的移动系统，还是用于 PC 的中规中矩的桌面系统，人们都已不再满足于千篇一律的桌面。打造一个全新的个性化桌面，绝不只是停留在更改个背景图、换个主题这个层面，我们需要更大的革新。但是如果对桌面环境的基本原理都不甚了解，那又何谈去开发打造具有创造性的用户交互。

因此，在本章中我们带领读者从头构建一个基本的桌面环境，包括窗口管理器、任务条以及一个显示桌面背景的组件。为了使读者更能深刻体会 X 的客户 / 服务器模型，窗口管理器基于 Xlib 编写，而任务条等组件则展示了使用 GTK 图形库的编程方法。

限于篇幅，我们没有将全部源代码全部贴到书中，所以请读者结合随书光盘中附带的源代码进行阅读。另外，本章虽然涉及 Xlib 和 GTK 编程，但是为了不干扰主线——构建桌面环境，我们不会过多讨论它们的编程，其中涉及的 API，如有必要请参考 Xlib 和 GTK 各自的参考手册。

7.1 窗口管理器

本质上，窗口就是显示器上对应的一块区域。对于一个运行多任务的操作系统来讲，在一个有限的屏幕上可以同时存在多个窗口，因此，用户希望多个窗口之间可以协调布局和平共享同一个屏幕。可以将特定窗口切换为当前活动窗口；可以按需改变窗口尺寸；可以最大化、最小化以及关闭窗口。但是 X 的设计哲学是只提供机制，不提供策略，X 服务器只提供

窗口操作相关的函数，但不管如何去操作窗口。于是诞生了另外一个特殊的 X 应用：窗口管理器。

7.1.1 基本原理

1. X 的窗口

X 将所有窗口组织为一棵树。X 服务器启动后，将默认创建一个窗口，这个窗口充满整个屏幕，作为整个窗口树的根，称为根窗口（Root Window），所有应用的顶层窗口（Top-level Window）都是根窗口的子窗口。

假设在 X 中运行两个应用 A 和 B，A 包含 2 个窗口，B 应包含 3 个窗口，窗口之间的布局如图 7-1 所示。

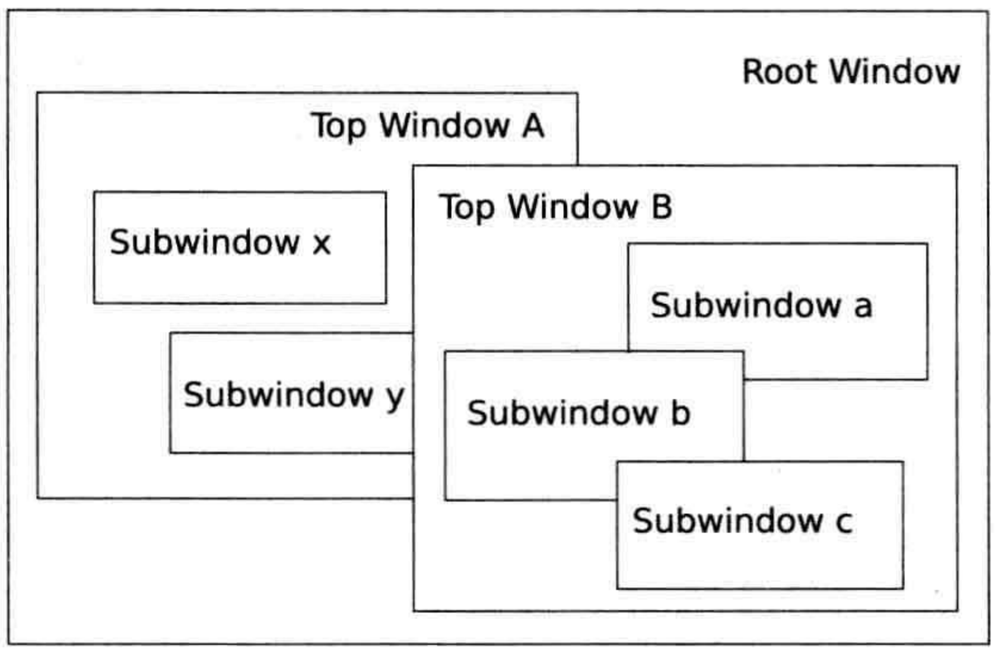


图 7-1 窗口布局示意图

它们之间的树形关系如图 7-2 所示。

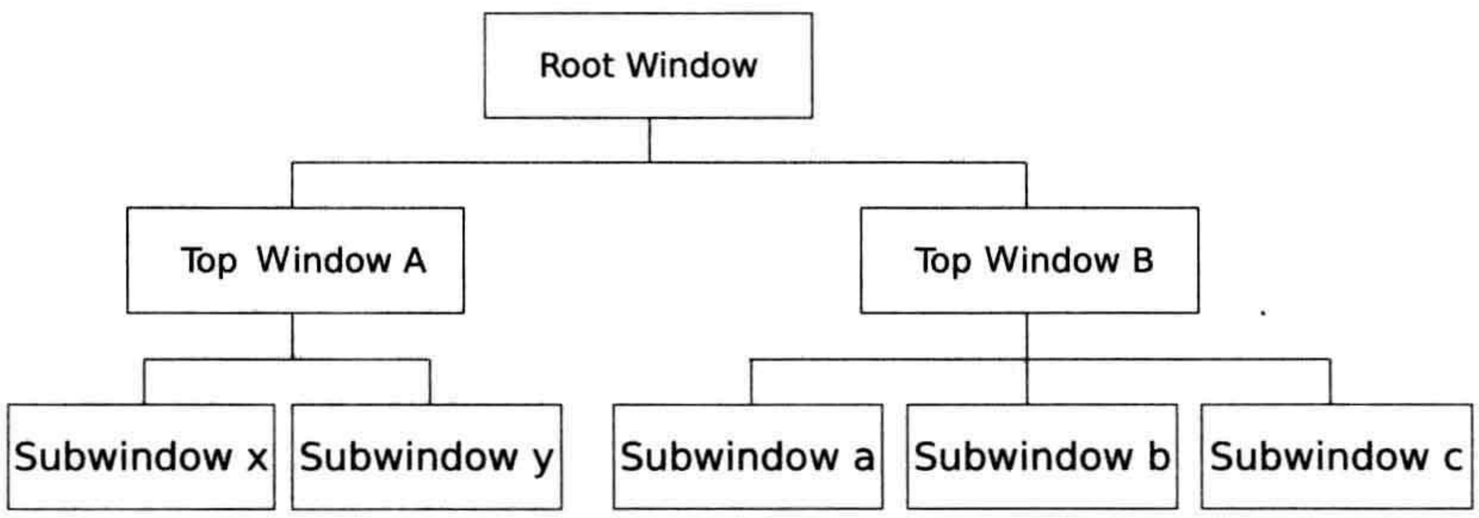


图 7-2 窗口树形关系示意图

窗口管理器仅管理应用的顶层窗口，即如图 7-2 中的“Top Window A”和“Top Window B”。一个应用可能有多个顶层窗口，除了应用的主窗口之外，对话框一般也是一个顶层窗口。而对于顶层窗口的子窗口，则由应用自己管理。

2. 窗口装饰

在第 6 章中，我们看到，无论是基于 Xlib 的程序，还是使用 GTK 编写的程序，在没有

窗口管理器的情况下，它们的窗口都以“素颜”示人，只是一个“裸”窗口。一个典型的桌面应用的窗口，一般而言，包括一个标题栏，标题栏上还可能显示窗口的名称、最大化、最小化和关闭按钮。另外，窗口一般还有一个边框。用户可以通过标题栏移动窗口，可以在边框处拖动鼠标改变窗口尺寸，可以分别通过最大化、最小化和关闭按钮最大化、最小化、关闭窗口。这些组件除了具备功能外，还具备美化的作用，比如可以设置窗口边框的颜色、阴影效果等，因此，它们也被称为窗口装饰。

显然，窗口装饰不应该由各个应用负责，暂且不提重复劳动，单单一致性就是个大问题。如果任由应用自己绘制，最后将导致窗口标题栏等装饰五花八门。因此，在 X 中，将窗口装饰提取为公共部分，由窗口管理器统一负责。通常的实现方式是：窗口管理器创建一个窗口，我们称这个窗口为 Frame，作为根窗口的子窗口，但是作为应用的顶层窗口的父窗口。其他装饰，或者直接绘制在 Frame 窗口上，或者创建新的装饰窗口，但是这些装饰窗口也作为 Frame 的子窗口，本章我们开发的窗口管理器采用后者。应用的顶层窗口和 Frame 窗口之间的关系如图 7-3 所示。

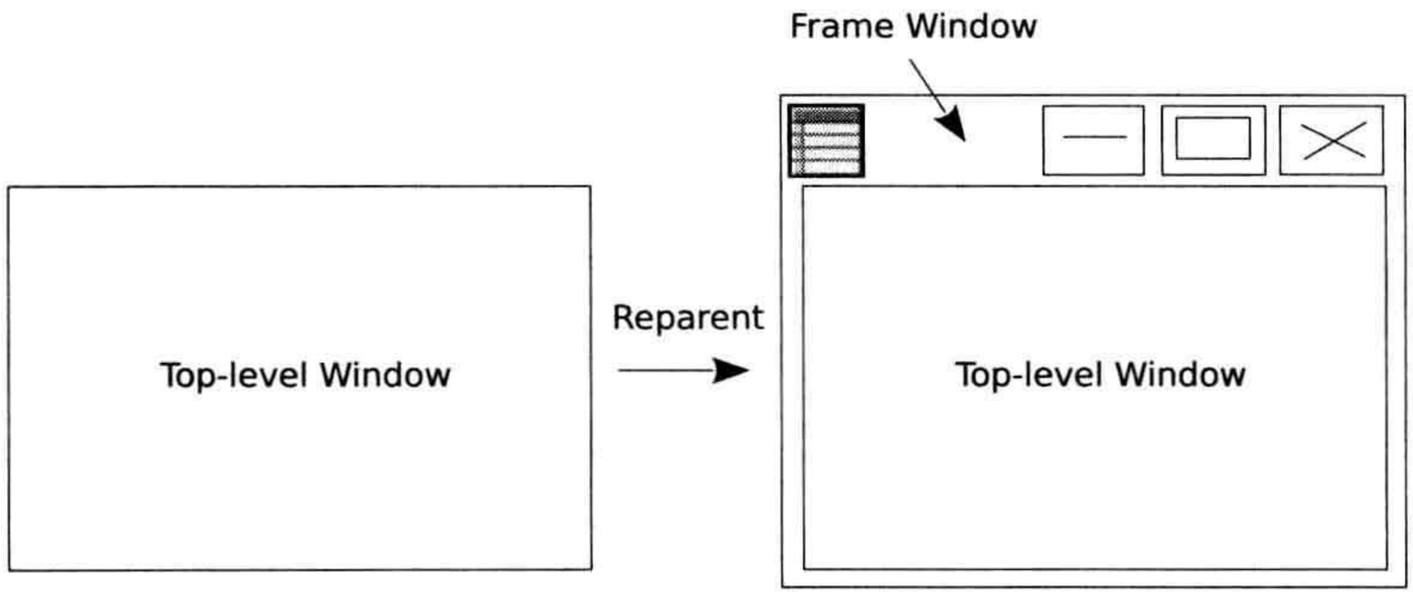


图 7-3　顶层窗口和 Frame 窗口的关系

3. 拦截事件

X 服务器维护一个事件队列，在该队列中按顺序保存着发生的各个事件，并周期地分发给应用。每个应用可以选择对发生在某些窗口上的哪些事件感兴趣，如果多个应用对同一个事件感兴趣，X 服务器将复制该事件的多个副本，并将其分发给各个对其感兴趣的应用，如图 7-4 所示。

Xlib 提供了函数 XSelectInput，应用程序可以使用该函数选择接收指定窗口的事件，其函数原型如下：

```
XSelectInput(Display *display, Window w, long event_mask)
```

其中参数 w 表示接收发生在窗口 w 上的事件，event_mask 表示对哪些事件感兴趣，如 ButtonPressMask 表示希望接收窗口 w 的 ButtonPress 事件。

在这些事件掩码中，有一个比较特殊——SubstructureRedirectMask，其含义是：当某个

应用选定了某个窗口的 SubstructureRedirectMask 时，该窗口的子窗口（Substructure）发送给 X 服务器的 MapRequest、ConfigureRequest 和 CirculateRequest 三类请求，都将被重定向给这个应用，这就是 X 的“Substructure Redirection”机制。

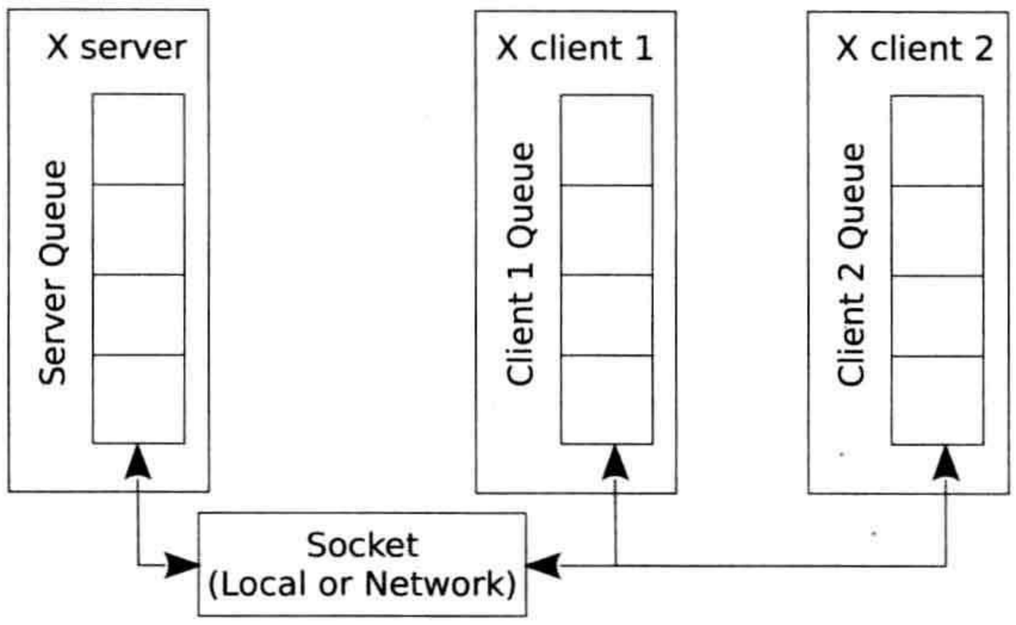


图 7-4 事件队列

窗口管理器恰恰利用了这个机制，对根窗口选择了 SubstructureRedirectMask，从而截获了应用的顶层窗口的请求。其中最关键的是 MapRequest，在窗口请求 X 服务器显示时，其将向 X 服务器发送 MapRequest 请求。在截获了 MapRequest 后，窗口管理器创建 Frame 窗口，作为根窗口的子窗口，然后暗渡陈仓，将应用的顶层窗口从根窗口脱离，而将其作为 Frame 窗口的子窗口，同时也创建其他窗口装饰。都伪装好后，窗口管理器再以 Frame 窗口的身份，请求 X 服务器显示 Frame 窗口。应用的顶层窗口作为 Frame 窗口的子窗口，当 Frame 窗口得以显示后，其自然也被显示。在某种意义上，窗口管理器通过 Frame 窗口控制了应用的顶层窗口，从而达到管理它们的目的。

在应用的顶层窗口作为 Frame 窗口的子窗口后，窗口管理器还是要关心它们发送给 X 服务器与窗口管理相关的请求，因此，如同设置根窗口的 SubstructureRedirectMask，窗口管理器也需要设置 Frame 窗口的 SubstructureRedirectMask。

不知读者是否考虑过这样一个问题：既然 X 服务器将其他应用的 MapRequest 请求重定向给窗口管理器，那么窗口管理器同样也作为 X 服务器的一个客户程序，它也需要向 X 服务器发送 MapRequest 请求，比如请求显示 Frame 等装饰窗口。如此这般，X 服务器岂不是将窗口管理器发送给它的请求再重定向给窗口管理器？如此往复，岂不是形成了死循环？

为此，窗口提供了一个属性：override_redirect。如果窗口的这个属性值为 True，则其明确告知 X 服务器自己不需要窗口管理器的管理，X 服务器就不会将这个窗口的请求重定向给窗口管理器。我们常用的鼠标右键菜单就是一个典型的将属性 override_redirect 设置为 True 的窗口。因此，窗口管理器在创建 Frame 等装饰窗口时，可以通过将它们的这个属性设置为 True 来解决我们刚刚谈到的死循环问题。事实上，即使不设置这个属性，也不会形成死循环，X 的开发者已经考虑了这个问题。

窗口管理器除了关心应用的顶层窗口的 SubstructureRedirectMask 涉及的请求外，另

外还要获得它们的某些通知事件。其中一个就是UnmapNotify，在收到这个通知后，窗口管理器需要清理所有为该窗口创建的对象，包括窗口装饰等。所以除了事件掩码SubstructureRedirectMask外，窗口管理器还要选择根窗口和Frame窗口的事件掩码SubstructureNotifyMask。

4. 窗口间通信

在一个标准的桌面环境下，存在多个不同的应用程序，除了普通的应用程序外，还有构成基本桌面环境的组件，如任务条等。而且，每个应用的窗口布局策略不尽相同，比如普通的X应用一般带有窗口装饰，但是我们有看到过构成桌面环境的任务条装饰着标题栏，并且标题栏上有最大化/最小化以及关闭等按钮吗？显然，这类组件不需要窗口装饰。我们还以任务条为例，在某些桌面环境上，任务条可以放置在屏幕的上方、下方、左侧以及右侧。再比如，对话框的窗口装饰中通常是没有最大化按钮的。

显然，窗口管理器需要获得窗口的相关信息，才能根据这些信息决定如何为这些窗口在同一个屏幕上协调的布局以及如何装饰这些窗口。为此，X提供了多种窗口间通信的机制，属性（Property）是窗口管理器和应用的窗口之间使用的主要通信机制。

X默认定义了一些属性，这些属性在窗口管理器规范中约定，但是应用也可以自定义属性。在X中，每个窗口都附着一个属性表，表中每一行大致就是属性的名字和其对应的值。应用可以设置自己创建的窗口的属性，也可以读取或者改变其他应用的窗口的属性，从而达到不同窗口间通信的目的。

属性保存在X服务器端。每个属性都有一个名字，为了便于使用属性，属性的名字是可读性更好的ASCII字符串而不是一串数字。然而，如果应用程序使用属性的名字引用属性，势必要通过套接字传递属性的名字给X服务器。但是字符串的数据量明显大于一个固定长度的整数，而且，还有一点，字符串的长度是可变的，也给协议的实现增加了复杂度。为此，X又为每个属性起了个小名，这个小名是一个整型数，与属性的名字间是一一对应的关系，X将其称为Atom，在应用与服务器之间通信时，使用这个小名而不是可变长度的字符串。

属性对应的Atom是动态创建的，当X服务器启动时，会为一些属性创建Atom，其他则是在首次使用时创建。Xlib提供了函数XInternAtoms和XInternAtom用来获取属性名对应的Atom。这两个函数基本相同，只不过一个是“批发”，一个是“零售”，相对于XInternAtom而言，XInternAtoms减少了应用和服务器之间的通信次数。XInternAtoms函数原型如下：

```
Status XInternAtoms(Display *display, char **names, int count,
                    Bool only_if_exists, Atom *atoms_return)
```

其中，参数names包含要转换的属性的名称，count表示转换的数量，转换后的Atom存储在数组atoms_return中。如果属性的Atom已经存在了，则直接获取其值即可，否则，是否为属性创建Atom要根据参数only_if_exists的值而定。只有only_if_exists为False时，才

创建 Atom。

Xlib 提供了函数 XGetWindowProperty 和 XChangeProperty 来读写窗口的属性，我们以 XGetWindowProperty 为例来讨论一下如何读取窗口属性，该函数原型如下：

```
int XGetWindowProperty(Display *display, Window w, Atom property,
     long long_offset, long long_length, Bool delete,
     Atom req_type, Atom actual_type_return,
     int *actual_format_return, unsigned long *nitems_return,
     unsigned long *bytes_after_return,
     unsigned char **prop_return)
```

1）参数 property 指的就是准备读取的窗口 w 的属性，根据该参数类型也印证了 X 没有使用属性的名字，而是使用了占用字节数更少的属性的 Atom。

2）属性的值可能是一个数组，比如窗口管理器规范 EWMH 规定属性 _NET_WM_WINDOW_TYPE 值就是一个 Atom 数组。数组就是在内存中的一块缓冲区了，从这个角度，就比较容易理解参数 long_offset 和 long_length 的意义了。XGetWindowProperty 为获取窗口属性提供了更大的灵活性，调用者可以通过参数 long_offset 和 long_length 读取存储属性值的缓冲区中指定偏移处的指定长度的值，这两个参数均以 32 位为单位。

3）在读取窗口的属性后，可以通过参数 delete 告诉 X 服务器是否删除窗口的这个属性，这也是为了节省内存空间考虑。

4）XGetWindowProperty 允许调用者传递参数 req_type 告诉服务器读取的属性值的类型，典型的包括 XA_ATOM、XA_CARDINAL 以及 XA_STRING 等，分别表示属性的值为 Atom、32 位整数以及字符串类型。当不确定属性的值的类型时，可以传递 AnyPropertyType 给 X 服务器，由 X 服务器将实际的类型通过参数 actual_type_return 返回给应用程序。

5）XGetWindowProperty 收到 X 服务器的返回值后，将动态申请一块内存，保存读取到的属性的值，并使用指针 prop_return 指向这块内存。既然是动态申请的内存，使用后需要用 Xlib 的函数 XFree 将其释放。

6）XGetWindowProperty 将实际读取的属性的值的类型保存在 actual_type_return 中；将实际读取的属性的值的格式保存在 actual_format_return 中，属性的值的格式可以是 8、16 或 32 三者之一，分别代表 char、short 以及 long；如果读取操作仅读取了保存属性值的缓冲区中的部分数据，则 XGetWindowProperty 将保存属性值的缓冲区中剩余的尚未读取的字节数存储在 bytes_after_return 中；nitems_return 中记录的是实际读取的属性的数量。

5. 捕捉窗口

我们设想这样一种场景，如图 7-5 所示，假设 X 服务器上已经在运行两个 X 应用 A 和 B，A 是当前活动的应用，B 是非活动应用。B 有两个顶层窗口，除了主窗口外，打开文件对话框也是一个顶层窗口，同时这个对话框也是应用 B 的临时（transient）窗口。正如其字面意义所言，所谓的“transient”就是临时的、短暂的，是一个相对的概念，是相对于某一窗口而言的。举个例子，如某些应用的“打开文件”对话框，是一个典型的临时窗口。但是

如果某个应用的主窗口就是一个对话框，那么这个对话框就不是临时窗口了。

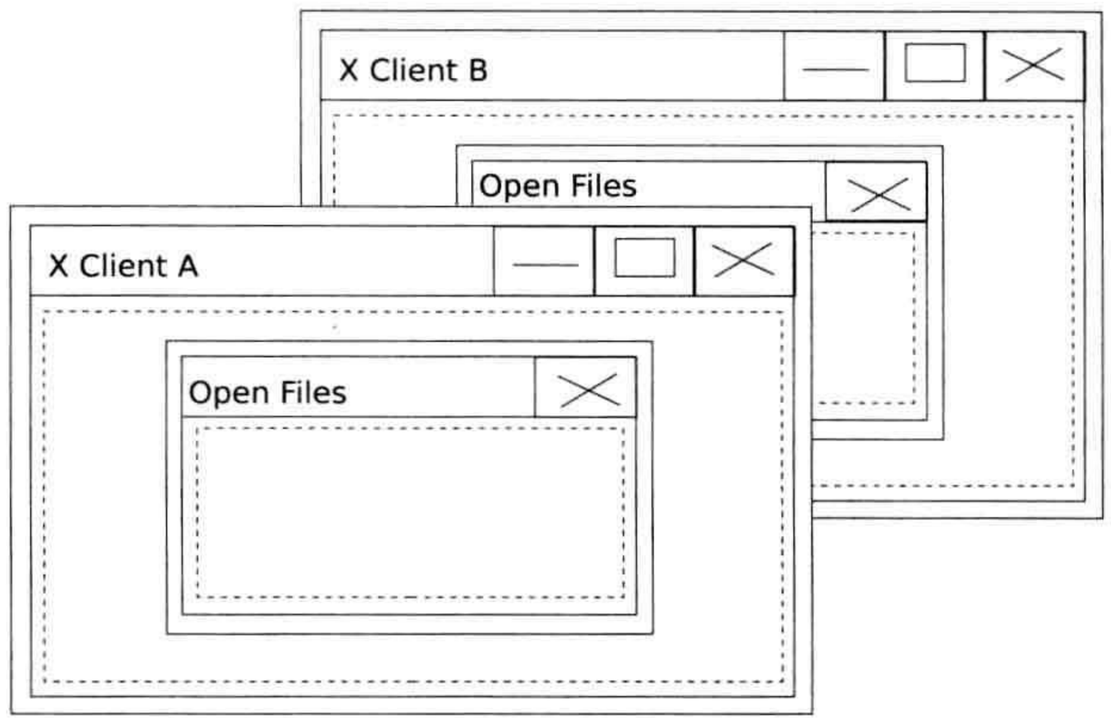


图 7-5 切换应用

当用户想要将应用 B 切换为当前活动的应用时，常用的方法之一是使用鼠标点击 B 应用的窗口。这时窗口管理器拦截鼠标事件，然后请求 X 服务器重新排列窗口栈序，具体细节见 7.1.11 节。总之窗口管理器必须要能接收到鼠标事件，如果接收不到鼠标事件，一切都无从谈起。

Frame 等装饰窗口是窗口管理器创建的，因此窗口管理器可以自如控制，比如我们可以设置 Frame 窗口的事件掩码中包含 ButtonPressMask。而对于应用的顶层窗口，我们肯定不能过多干涉。但是，我们又不能强制用户一定要点击到 Frame 窗口上未被应用顶层窗口覆盖的地方。而且一般情况下，用户一定是点击到顶层窗口或者其子窗口上，而不是 Frame 窗口上，毕竟 Frame 窗口未被应用顶层窗口遮挡的区域除了标题栏外，只有很小的边框了，也就是说能被点击到的区域很小。

根据 X 的事件传播机制，如果发生在一个窗口上的事件未被处理，在该窗口没有设置禁止事件继续向其父窗口传播的情况下，事件将沿着窗口树一直向着树的根部传播。很少有具有图形界面的程序不处理鼠标事件，否则就没有任何意义了，也就是说，鼠标事件几乎永远传递不到 Frame 窗口，都被应用自身消化了。如果不能接收鼠标事件，更何谈激活窗口了。那么怎么解决这个问题呢?

X 提供了鼠标捕捉机制，其又分为主动捕捉和被动捕捉。以图 7-5 为例，假设另外一个应用以被动机制捕捉应用 B 的顶层窗口时，当用户在应用 B 的顶层窗口范围内按下鼠标时，将激活捕捉机制，X 服务器将鼠标事件不再按照正常的事件传播路径传播了，而是转发给捕捉应用 B 的顶层窗口的 X 应用。窗口管理器恰恰是利用了这个机制，捕捉非活动窗口，从而捕获这些窗口的鼠标事件，实现不同应用间的切换。

Xlib 提供的用于捕捉的函数是 XGrabButton，其原型如下：

```
XGrabButton(Display *display, unsigned int button,
```

```
unsigned int modifiers, Window grab_window, Bool owner_events,
unsigned int event_mask, int pointer_mode,
int keyboard_mode, Window confine_to, Cursor cursor)
```

其中各个参数意义如下：

- button 表示捕捉鼠标哪个键，比如是捕捉左键还是捕捉右键等。
- modifiers 表示是否要求同时按下键盘上某个按键才能捕捉，也就是我们所说的修饰键。
- event_mask 表示捕捉事件的掩码，即捕捉什么事件，是捕捉按下鼠标事件还是捕捉释放鼠标事件等。
- confine_to 表示是否需捕捉的区域限制在某个范围，也就是说，当事件发生时，只有鼠标在这个区域才可以捕捉。
- cursor 表示当捕捉发生时，是否需要使用特定的鼠标指针形状，以给用户一个友好的提示。
- owner_events 主要是用于当应用捕捉自身创建的窗口时使用，与窗口管理器无关。
- 参数 grab_window 是最核心的一个参数，理解了这个参数就基本理解了整个函数，这个参数就是表明当鼠标按键发生在哪个窗口时进行捕捉。
- 最后来解释参数 pointer_mode。我们举个例子来解释这个参数，假设我们将捕捉比喻为窃，那么捕捉其他窗口的应用就是江洋大盗，被捕捉的窗口所属的应用就是受害人。不知读者是否有这样的疑问：当江洋大盗将事件窃走后，受害人还能否失而复得。X 再次将这个策略性的问题抛给了应用自己来决定。X 提供了两种捕捉模式：异步模式和同步模式。当使用异步模式时，受害人不要心存任何侥幸了。而当使用同步模式时，在取消对一个窗口的捕捉行为后，如果江洋大盗良心发现，X 则会给他一次浪子回头的机会。江洋大盗可以调用 Xlib 的函数 XAllowEvents 放行这个被截获的事件，这样受害者就可以失而复得了，但是可能不是那么新鲜了，要晚一点。

6. save-set

笔者没有找到一个恰当一点的词来表达 save-set 这个术语，所以我们就直接用英文了。根据其名字就可以猜出这是一个集合了。但是这个集合是做什么的呢?

我们设想这样一种情况，当窗口管理器异常终止时，窗口管理器创建的 Frame 等装饰窗口自然也被销毁。销毁这些窗口本身没有问题，但是它们带来了副作用：作为 Frame 窗口子窗口的应用的窗口也被销毁。这显然不是我们希望看到的。

每个 X 应用都有一个 save-set，其中保存的就是就是窗口的列表。当应用异常断开到 X 服务器的连接时，X 服务器将首先检查应用的 save-set，并安排根窗口领养 save-set 中的窗口，从而避免了在 save-set 中的这些窗口被销毁。

前面提到的窗口管理器的问题恰恰可以用这个方法解决。每当管理一个窗口时，窗口管理器就可以调用 Xlib 的函数 XAddToSaveSet 将其加入到自己的 save-set 中。一旦当窗口管理器异常终止，根窗口将领养应用的窗口，从而避免了 Frame 窗口被销毁时，应用的窗口也被销毁的命运。

7.1.2 创建编译脚本

不知道读者是否注意到，几乎前面编译的所有软件在进行安装时，仅仅通过定义环境变量 DESTDIR 为 $SYSROOT，就安装到了目录 /vita/sysroot 下。如果 Makefile 全部是由程序员手工写的，不知道是否能做到如此整齐划一？很多手写的 Makefile 中，目标 install 的规则更多的是形如下面这个样子：

```
install:
    install x /usr/bin
```

很难考虑到像下面这样周全：

```
install:
    install x $(DESTDIR)/usr/bin
```

因此，标准化的 Makefile 对 GNU 这种由来自世界各地的程序员共同参与开发的项目非常重要。

前面，我们已经领教了内核构建系统中的 Makefile，从其复杂程度可见，对于具有多级目录、多个目标的复杂项目，编写和维护一个 Makefile 是多么繁重的一件事情。

鉴于类 UNIX 系统版本的多样性，GNU 软件的源代码级的可移植就变得非常重要了，因此，编译脚本时必须要小心应对不同系统环境之间的差异。以我们的环境为例，同样一个软件，在不修改配置编译脚本的前提下，在宿主系统下，应该将编译器识别为 gcc，而在交叉编译环境下，应该将编译器识别为 i686-none-linux-gnu-gcc。这只是非常简单的一个例子，对于复杂的项目，情况要比这个糟糕得多。

于是饱受折磨的开发者们开发了 GNU 构建系统（GNU Build System），或者叫 GNU 自动构建工具（GNU Autotools），为了行文方便，我们简称其为 Autotools。Autotools 核心包括 Autoconf 和 Automake。这里要准确理解“自动构建工具”的意义，所谓 Autotools，并不是自动完成整个配置编译过程，而是自动构建配置脚本 configure 和 Makefile。

（1）Autoconf

Autoconf 的准确含义是自动创建自动配置脚本（automatically create automatic configuration scripts）。怎么理解自动配置脚本呢？简单来讲，就是自动探测各种不同系统的各种特性，如是本地编译还是交叉编译，系统中使用的编译器、链接器等程序是什么，编译以及链接程序时需要的头文件、动态库以及它们所在的路径，等等，达到自动动态适配，而不是硬编码到脚本中。

可以这样概括 Autoconf 的工作过程：将多个 shell 片段最终合并为一个完整的 shell 脚本，即 configure。Autoconf 使用宏来定义这些 shell 片段，开发者需要根据编译需要，使用这些宏组合 Autoconf 的元文件 configure.ac，这个元文件曾经命名为 configure.in，后来更改为 configure.ac，但是 Autoconf 也向后兼容 configure.in。然后 Autoconf 将元文件 configure.ac 中的宏展开为具体的配置脚本 configure。

Autoconf 程序本身使用 shell 脚本编写，但是 Autoconf 并没有使用 shell 完成宏展开功能，而是借助了 GNU 的 M4 来完成宏的展开。简单来讲，M4 就是将输入的宏名转换为宏定义，也就是说，M4 的输入是宏名，而输出是 shell 脚本片段。Autoconf 使用 M4 定义了一些内置的宏，并且基于 M4 之上又封装了一层宏，目的是为了更符合 Autoconf 的需求，Autoconf 封装的宏一般以“AC_”开头。其他程序可以使用 Autoconf 封装的这些宏，或者直接使用 M4 定义自己的宏，但是最终，本质上都是 M4 宏。

因为 M4 宏定义很多是第三方程序提供的，可能安装在系统的多个位置，因此 GNU 自动构建系统编写了程序 aclocal 负责将这些宏定义收集到文件 aclocal.m4，保存在源码的顶层目录下，供自动构建系统使用。

（2）Automake

同 Autoconf 类似，Automake 的准确含义是“automatically generate makefile.in”，开发人员只需编写一个简单的元文件，在其中描述必要的诉求：比如构建一个二进制程序，使用的源代码文件是什么，链接某某库等即可。其他的都交由 Automake 全权处理吧。Automake 将创建一个标准的 Makefile 文件，包括补全开发者不愿意编写的那些琐碎的规则，如 install、clean、distclean、dist 等。

Automake 的输出事实上是一个 Makefile 模板，命名为 Makefile.in。然后，configure 脚本使用探测到的值替换模板 Makefile.in 中的变量，创建最终的 Makefile。显然，这种方式要比我们将所有的变量定义全部硬编码到 Makefile 中的做法可移植性更好。

综上，使用 GNU Autotools 创建 Makefile 的过程可以分为如下几个步骤：

1）编写元文件 configure.ac。

2）执行 aclocal。aclocal 将扫描 configure.ac 中使用的 M4 宏，并到系统中收集这些宏的定义，然后将这些宏定义复制到源码顶层目录下的 aclocal.m4 中。

3）调用 autoconf，将 configure.ac 中的宏展开为 shell 脚本形式的 configure。

4）编写元文件 Makefile.am。

5）调用 automake。automake 根据 Makefile.am 创建 Makefile 的模板文件 Makefile.in。

6）执行脚本 configure。configure 探测系统环境，并使用探测到的值替换模板 Makefile.in 中的变量，生成具体的 Makefile。

从上面的讨论可以看出，对于开发者来说，主要的工作就是创建元文件 configure.ac 和 Makefile.am，其他的全部交给 Autotools。Autotools 极大地减轻了程序开发人员的负担，将烦琐的编写的 Makefile 任务转嫁给了 Autotools 的开发和维护者。

既然 Autotools 有如此多的优点，所以即使我们的迷你窗口管理器很小，我们还是可以借助它感同身受一下 Autotools 带来的好处。我们这里绝非“杀鸡用牛刀”，而是希望读者借助这个例子，可以切身体会一下 Autotools，这样无论是在大型项目中使用 Autotools，或者为 GNU 软件贡献源码，亦或基于使用 Autotools 的项目进行二次开发，都会大有益处。

1. 创建 configure

我们将这个迷你窗口管理器命名为 winman，使用 winman 作为顶层目录的名字，在顶层目录下创建一个子目录 src 用来存放源代码。我们基于 Xlib，使用 C 语言编写 winman。因此，configure.ac 中除了 Autoconf 要求的必选的宏外，最重要的就是检查 C 编译器和 X 的库了，其内容如下：

```
winman/configure.ac:

AC_INIT(winman, 0.1, baisheng_wang@163.com)
AM_INIT_AUTOMAKE(foreign)

AC_PROG_CC

PKG_CHECK_MODULES(X, x11)

AC_CONFIG_FILES(Makefile src/Makefile)
AC_OUTPUT
```

Autoconf 要求 configure.ac 以宏 AC_INIT 作为开头，该宏由 Autoconf 定义，接收一些基本信息，如软件包的名称，版本号，开发或者维护人员的 Email 等。制作发布的软件包时，将用到这些信息。

宏 AM_INIT_AUTOMAKE 由 Automake 定义，用来进行与 Automake 相关的初始化工作，只要使用 Automake，这个宏也是必选的。在默认情况下，Automake 会检查项目目录中是否包含 NEWS、README、ChangeLog 等文件，为简单起见，我们给 Automake 传递了“foreign”参数，明确告诉 Automake 我们的项目不需要包含这些文件。

宏 AC_PROG_CC 用来检测 C 编译器，根据该宏名的前缀“AC_”就可判断出其他由 Autoconf 定义。其将在系统内搜索 C 编译器，并定义变量 CC 指向搜索到的 C 编译器。

接下来，我们使用软件包 pkg-config 提供的宏 PKG_CHECK_MODULES 检测 X 的库。该宏将定义两个变量，分别是宏的第一个参数加上后缀“_CFLAGS”和“_LIBS”，这里就是 X_CFLAGS 和 X_LIBS。如果查看 congfigure 脚本就可以发现，这个宏定义的核心其实就是执行命令“pkg-config --cflags x11”和“pkg-config --libs x11”。

宏 AC_CONFIG_FILES 告诉 Automake 生成哪些 Makefile 模板文件 Makefile.in。这里，需要在顶层目录和 src 目录下分别创建 Makefile.in 文件。

在 configure.ac 的最后，Autoconf 要求必须以宏 AC_OUTPUT 结束 configure.ac。

准备好 configure.ac 后，我们使用如下命令生成脚本 configure：

```
vita@baisheng:/vita/build/winman$ aclocal
vita@baisheng:/vita/build/winman$ autoconf
```

2. 生成 Makefile

窗口管理器的源码保存在顶层目录下的子目录 src 中，我们在顶层目录和子目录 src 下面分别需要编写 Automake 元文件 Makefile.am。

顶层目录 winman 下的 Makefile.am 如下：

```
winman/Makefile.am:

SUBDIRS = src
```

因为顶层目录下基本没有任何操作，所以该 Makefile.am 非常简单，只是通过变量 SUBDIRS 告诉 Automake，需要递归编译子目录 src。

子目录 src 下的 Makefile.am 如下：

```
winman/src/Makefile.am:

bin_PROGRAMS = winman

winman_SOURCES = wm.h main.c ...

winman_CFLAGS = $(X_CFLAGS)
winman_LDADD = $(X_LIBS)
```

变量 bin_PROGRAMS 指定了编译最后创建的二进制可执行文件的名称，该变量由两部分构成，其中“bin”表示安装在目录 $prefix/bin 下，“PROGRAMS”指明最后创建的文件是一个可执行文件。

winman_SOURCES 表示创建 winman 需要的源文件；winman_CFLAGS 和 winman_LDADD 分别表示编译链接时需要传递给编译器和链接器的参数。X_CFLAGS 和 X_LIBS 已经在前面的 configure.ac 中由宏 PKG_CHECK_MODULES 定义了。

Automake 的元文件已经准备就绪，我们使用下面的命令创建 Makefile 的模板 Makefile.in：

```
vita@baisheng:/vita/build/winman$ automake --add-missing -copy
```

其中选项“--add-missing”和“--copy”是告诉 automake 将其需要的一些脚本文件，比如 install-sh 等，直接复制到项目目录中，而不是建立这些脚本文件的链接。这么做是为了分发到其他系统时，避免因为脚本位置不同或者系统中没有安装相应脚本而导致编译链接失败。

上述命令执行后，将分别在顶层目录和子目录 src 下创建 Makefile 的模板 Makefile.in。

最后，执行 confugure 脚本探测编译过程所需的各个变量，然后用探测到的具体的值替换 Makefile.in 中的变量，比如 X_CFLAGS、X_LIBS，生成 Makefile 文件：

```
vita@baisheng:/vita/build/winman$ ./configure --prefix=/usr
```

7.1.3 主要数据结构

在 winman 中，将为每个被管理的窗口创建一个对象，记录其相关信息，为此我们抽象了结构体 Client。另外，我们抽象了结构体 WinMan，其中记录了一些全局信息。在讨论具体的实现前，我们先来了解一下这两个数据结构。读者不必全部理解每一项的含义，后面具

体遇到时，读者可以再回到这里，前后结合进行理解。

1. 结构体 Client

结构体 Client 中主要包含窗口属性信息以及与窗口操作相关的函数，定义如下：

winman/src/wm.h:

```
typedef struct _Client {
    Window window;
    WinMan *wm;

    Window frame;
    Window titlebar;
    Window minimize_btn;
    Window maximize_restore_btn;
    Window close_btn;
    Window acting_btn;

    Window rsz_top_side;
    Window rsz_bottom_side;
    ...
    Window rsz_lr_angle;
    Window resizing_area;

    Bool moving;
    int anchor_x, anchor_y;

    int x, y, width, height;
    int min_width, min_height;
    unsigned int state;
    int restore_x, restore_y, restore_w, restore_h;

    struct _Client *trans_for;
    int ignore_unmap;

    struct _Client *above;
    struct _Client *below;

    void (* configure) (struct _Client *, XConfigureRequestEvent *);
    void (* reparent) (struct _Client *);
    ...
    void (* move_resize) (struct _Client *);
} Client;
```

我们结合图 7-6 来解释其中相关数据项。

- 每个窗口作为 X 服务器的一个资源，都有一个 ID 来唯一标识。这里的数据项 window 就是被管理的窗口的 ID。
- wm 指向全局的结构体 WinMan 的对象。
- frame 是 Frame 窗口的 ID；titlebar 是标题栏窗口的 ID；minimize_btn 是最小化按钮对应的窗口的 ID；maximize_restore_btn 是最大化 / 恢复按钮对应的窗口的 ID；close_btn 是关闭按钮对应的窗口的 ID。acting_btn 是一个为了编程方便定义的辅助变

量，用来记录用户点击了最大化、最小化以及关闭按钮中的哪一个。

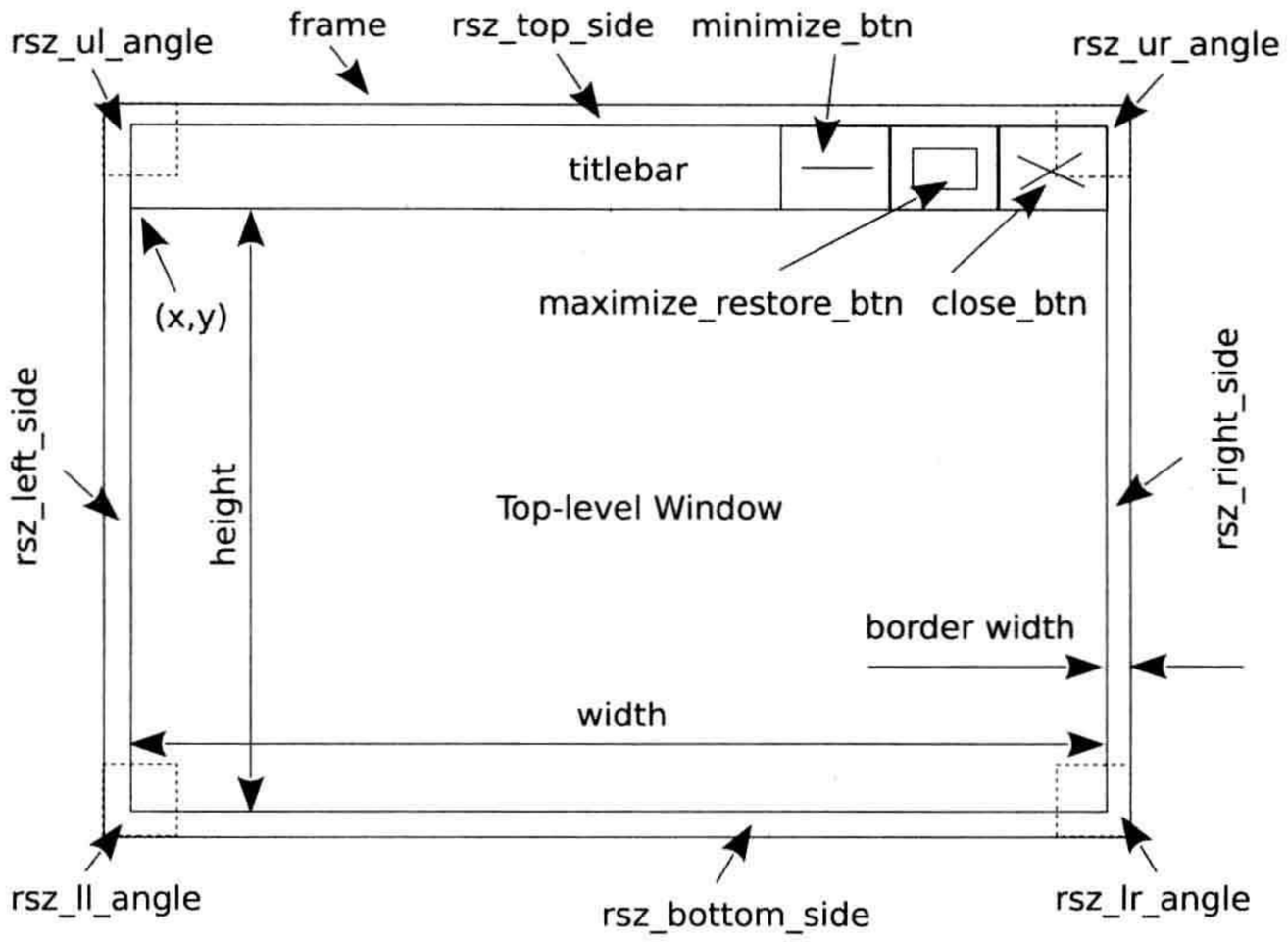


图 7-6 窗口相关参数

- 以“rsz_”（resize 的简写）开头的 8 个窗口，是在 Frame 窗口上创建的 8 个不可见窗口，它们分别位于 Frame 窗口的 4 个边和 4 个角，目的是为了便于判断鼠标是否落在调整窗口尺寸的区域。也就是说，一旦用户的鼠标落入这个窗口区域，程序的鼠标将使用特定的指针，提示用户可以进行改变窗口的尺寸了。resizing_area 也是一个辅助变量，用来记录鼠标落在了调整窗口尺寸的 8 个区域的哪个区域，也就是哪个窗口上。
- 变量 moving 用来标识用户是否正在移动窗口，anchor_x、anchor_y 记录用户在标题栏上按下鼠标左键、准备开始移动时的位置，目的是为了计算鼠标按下位置与当前位置的距离。
- x、y、width、height 记录窗口的位置及大小。min_width、min_height 表示允许用户改变窗口尺寸时允许的最小值，主要目的是避免用户不小心将窗口缩小的太小，导致窗口“丢失”了。
- state 记录窗口的状态，winman 只处理最大化及其标准状态。restore_x、restore_y、restore_w、restore_h 分别记录窗口在最大化之前的位置和尺寸，以便从最大化恢复为标准尺寸时使用。
- trans_for 的目的是为了记录某个窗口是否是临时窗口，如果是临时窗口，那么它是谁的临时窗口。在讨论窗口切换时，我们将看到这个数据项的意义。
- 变量 ignore_unmap 涉及的内容有点复杂，将在 7.1.12 节详细讨论。
- 每个窗口通过指针 above 和 below 链接到窗口栈中。

与窗口操作相关的一些函数的实现，后面章节中我们会讨论。

2. 结构体 WinMan

结构体 WinMan 中包含了全局需要使用的变量，定义如下：

winman/src/wm.h:

```
typedef struct _WinMan {
    Display *dpy;
    int screen;
    Window root;

    Atom atoms[ATOM_COUNT];

    struct _Client *stack_top;
    struct _Client *stack_bottom;
    int stack_items;

    struct _Client *active;
    struct _Client *desktop;
    struct _Client *taskbar;
} WinMan;
```

上述代码中各个参数含义如下：

- 第一个数据项 dpy 无需多说了，它是代表应用到 X 服务器的连接。一个 X 服务器可以支持多屏，每个屏上都会有一个根窗口。虽然我们不考虑多屏的情况，但是某些函数使用屏幕号和根窗口作为参数，为了避免每次使用时都要从 X 服务器获取，winman 在结构体中保留了一份副本，即数据项 screen 和 root，分别代表屏幕号和根窗口 ID。
- 数组 atoms 中记录的是用于窗口间通信的 Atom，winman 也不希望应用反复的从 X 服务器获取这些 Atom，所以将它们保存在 winman 的本地。
- winman 使用栈的方式记录窗口对象（即为每个窗口创建的结构体 Client 的实例），stack_top 和 stack_bottom 分别指向这个窗口栈的栈顶和栈底。距离用户最远的窗口记录在栈底，距离用户最近的窗口记录在栈顶。stack_items 记录栈中窗口对象的数量。
- active、desktop 以及 taskbar 分别指向当前活动的窗口、构成桌面环境的桌面组件以及任务条组件。

7.1.4 初始化

谈及初始化，大多给人的印象是进行一些琐碎的准备工作，但是 winman 中有几处却非常关键，相关代码如下：

winman/src/main.c:

```
int main(int argc, char *argv[])
{
    WinMan *wm;
```

```
    wm = malloc(sizeof(WinMan));
    memset(wm, 0, sizeof(WinMan));

    if (!(wm->dpy = XOpenDisplay(NULL))) {
        ...
    }

    wm->screen = DefaultScreen(wm->dpy);
    wm->root = RootWindow(wm->dpy, wm->screen);
    XSetErrorHandler(error_handler);

    atom_init(wm);

    XSelectInput(wm->dpy, wm->root, SubstructureRedirectMask
                 | SubstructureNotifyMask);

    init_clients(wm);

    wm_event_loop(wm);

    return 0;
}
```

下面介绍该函数主要执行的操作。

（1）连接 X 服务器

窗口管理器与普通的 X 应用并无本质区别，只是具有一点点特权，它也是 X 服务器的一个客户端。从服务器和客户端的体系架构角度而言，应用程序当然需要和服务器建立连接后才能通信，为此，Xlib 提供了函数 XOpenDisplay 用于建立它们之间的连接。

（2）错误处理

Xlib 将错误分为两类：一类错误是不可恢复的，这类错误是致命的，一旦发生后，应用基本不可能执行下去了，如应用程序和服务器的连接断开了，除了终止应用程序外别无选择；另外一类是协议错误，比如当应用读取某个窗口的属性时，这个窗口可能在服务器中已经不存在了。显然，这类错误不是致命的，应用完全可以自己决定是忽略错误还是终止执行。

Xlib 为这两类错误都设置了默认的错误处理函数，它们的行为均是打印错误提示并终止应用。但是 Xlib 也分别提供了接口 XSetIOErrorHandler 和 XSetErrorHandler 允许应用设置自己的致命错误和协议错误处理函数。winman 设置了协议错误处理函数为 error_handler，当发生协议错误时，error_handler 只打印错误信息，并不终止执行。

（3）创建 Atoms

函数 atom_init 使用 Xlib 的函数 XInternAtoms，一次性创建后面用到的属性的 Atom，并将 Atom 保存在结构体 WinMan 的 atoms 数组中。相关代码如下：

```
winman/src/main.c:
static void atom_init(WinMan *wm)
{
    char *atom_names[] = {
```

```
            "_NET_WM_WINDOW_TYPE",
            ...
        };

    XInternAtoms(wm->dpy, atom_names, ATOM_COUNT, False, wm->atoms);
}
```

最初，X 标准协会制定了 ICCC 通信协议。后来，随着现代桌面的发展，又制定了 EWMH，即对 ICCCM 进行的补充扩展。这两个协议中定义了大量的属性，在函数 atom_init 中，以 WM_ 开头的基本是 ICCCM 标准中定义的，以 _NET_ 开头的是 EWMH 中定义的。除了标准定义的属性外，应用也可以自定义属性，其中以 _CUSTOM_ 开头的就是 winman 自定义的属性。

（4）拦截事件

函数 XSelectInput 可以说是窗口管理器的画龙点睛之笔了。winman 按照前面的讨论，选择了根窗口的“SubstructureRedirectMask”和“SubstructureNotifyMask”。

（5）管理已存在的窗口

在窗口管理器启动之前，系统上可能已经有 X 应用在运行。因此，winman 启动时需要管理这些已存在的窗口，函数 init_clients 就是做这件事的，其具体细节请参见 7.1.13 节。

（6）事件循环

初始化完成后，窗口管理器进入事件循环，函数 wm_event_loop 调用 Xlib 的函数 XNextEvent 将窗口管理器对 X 的请求发送给 X 服务器，然后检查事件队列，调用相应的事件处理函数。接下来的的章节中，我们会陆续讨论这些事件处理函数。

需要特殊指出的是 Expose 事件，以图 7-7 所示窗口布局为例。

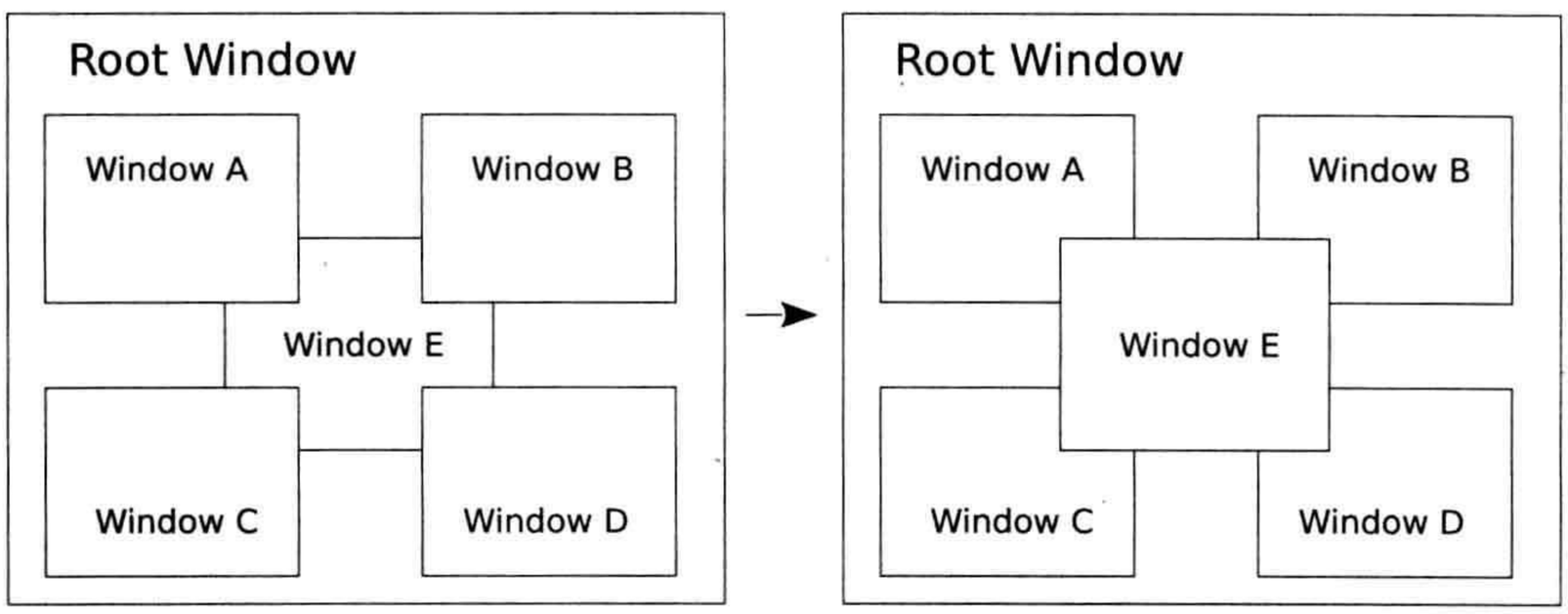


图 7-7　多个连续 Expose 事件发生情况

Window E 有四个区域分别被 Window A~D 遮挡，当 Window E 成为当前活动窗口时，X 服务器将为每一个被遮挡的部分都报告一个 Expose 事件，并将同一个动作引发的 Expose 事件连续的放到事件队列中。结构体 XExposeEvent 中的变量 count 就是用来记录一个 Expose 事件后面还有多少个 Expose 事件的。

因此，从效率的角度来讲，对于多个连续的 Expose 事件，应用应该忽略掉最后一个 Expose 事件前面所有的 Expose 事件，而在收到最后一个 Expose 事件时才进行绘制。

7.1.5 为窗口“落户”

一旦收到 X 服务器转发来的 MapRequest，就说明有应用的顶层窗口请求显示了，显然，这个时机是窗口管理器切入的最佳时机。winman 首先遍历窗口栈确认窗口是否已经被管理了，如果请求映射的窗口尚未被管理，则调用 wm_new_client 开始管理窗口，函数 wm_new_client 的代码如下：

```
winman/src/main.c:

static Client* wm_new_client(WinMan *wm, Window win)
{
    Atom type;
    int format, status;
    unsigned long n, extra;
    Atom *value = NULL;
    Client *c = NULL;

    status = XGetWindowProperty(wm->dpy, win,
                  wm->atoms[_NET_WM_WINDOW_TYPE],
                  0, 1, False, XA_ATOM, &type, &format, &n,
                  &extra, (unsigned char **)&value);

    if (status == Success && type == XA_ATOM
            && format == 32 && value) {
        if (value[0] == wm->atoms[_NET_WM_WINDOW_TYPE_NORMAL])
            c = normal_client_new(wm, win);
        else if (value[0] == wm->atoms[_NET_WM_WINDOW_TYPE_DIALOG])
        ...
    }
    ...
    c->reparent(c);
    c->show(c);

    return c;
}
```

该函数执行的主要操作如下：

1）如同我们每个人要有一个户口，在落户时需要提供各种自然人信息一样，窗口管理器也要收集窗口的各种“自然人”信息，为窗口在窗口管理器中“落户”。

2）绘制窗口装饰。

3）一切准备妥当后，申请 X 服务器显示应用的窗口，当然也包括窗口管理器附加的装饰。

这一节，我们先来讨论为窗口“落户”这一过程。

如前所述，在一个典型的 X 环境中，可能有多种类型的 X 应用程序，比如构成桌面环境

的任务条等组件，以及普通的应用程序。即使是普通的应用的窗口，也可分为标准的窗口以及对话框等。显然，不同的类型的窗口需要区别对待，我们不能给任务条也加个标题栏，那样就会闹出笑话。

EWMH 规定窗口需要设置属性 _NET_WM_WINDOW_TYPE 来表明自己的类型，函数 wm_new_client 依据的就是 EWMH 这个规定来判别窗口的类型。因此，函数 wm_new_client 调用 Xlib 的函数 XGetWindowProperty 获取窗口的属性 _NET_WM_WINDOW_TYPE 的值，根据窗口的不同类型，创建不同类型的窗口对象。

下面，我们以标准窗口为例，讨论其“落户”过程。

winman/src/normal_client.c:

```
Client* normal_client_new(WinMan *wm, Window win)
{
    Client *c;
    XWindowAttributes attr;
    XSizeHints *hints = NULL;
    long dummy;
    Window trans_for = None;

    c = malloc(sizeof(Client));
    memset(c, 0, sizeof(Client));
    c->window = win;
    c->wm = wm;

    XSetWindowBorderWidth(wm->dpy, c->window, 0);
    XGetWindowAttributes(wm->dpy, win, &attr);
    c->x = attr.x;
    c->y = attr.y;
    c->width = attr.width;
    c->height = attr.height;

    if (!(hints = XAllocSizeHints()))
        return;
    if (XGetWMNormalHints(wm->dpy, c->window, hints, &dummy)) {
        if (hints->flags & PMinSize) {
            c->min_width = hints->min_width;
            c->min_height = hints->min_height;
        }
    }
    XFree(hints);

    c->min_width = c->min_width > MIN_WIDTH ?
           c->min_width : MIN_WIDTH;
    c->min_height = c->min_height > MIN_HEIGHT ?
           c->min_height : MIN_HEIGHT;

    ewmh_get_net_wm_state(c);
    if (c->state & (NET_WM_STATE_MAXIMIZED_V
            | NET_WM_STATE_MAXIMIZED_H))
        custom_get_restore_size(c);
```

```
    XGetTransientForHint(wm->dpy, c->window, &trans_for);
    if (trans_for)
        c->trans_for = wm_find_client_by_window(wm, trans_for);
    if (!c->trans_for) {
        if (wm->active) {
            XGrabButton(wm->dpy, Button1, 0, wm->active->window,
                 True, ButtonPressMask, GrabModeSync,
                 GrabModeSync, None, None);

            Item *trans = normal_client_get_transients(wm->active);
            Item *i;
            for (i = trans; i; i = i->next) {
                XGrabButton(c->wm->dpy, Button1, 0,
                   i->client->window, True, ButtonPressMask,
                   GrabModeSync, GrabModeSync, None, None);
            }
            list_free(&trans);
        }

        wm->active = c;
        ewmh_set_net_active_window(wm->active);
    }

    c->configure = &normal_client_configure;
    c->reparent = &normal_client_reparent;
    ...
    stack_append_top(c);
        ewmh_update_net_client_list_stacking(wm);

        return c;
    }
```

下面介绍函数 normal_client_new 执行的主要操作。

（1）设置窗口边框

通常，窗口可以请求 X 服务器绘制边框。但是为了统一，我们调用 XSetWindowBorderWidth 人为地将窗口的自身的边框设置为 0，而是在 Frame 窗口上为被管理的窗口绘制统一的边框。在 winman 中，为简单起见，边框的宽度采用了一个固定的值。但是窗口管理器可以尊重窗口的诉求，在绘制窗口边框前，读取窗口属性中设定的边框宽度。

（2）获取窗口几何尺寸

接下来，我们读取窗口的几何尺寸，包括位置、宽度和高度，以及窗口所允许的最小的尺寸。这里我们分别使用了 Xlib 的函数 XGetWindowAttributes 及 XGetWMNormalHints，主要是因为 Xlib 不推荐通过 XGetWMNormalHints 获取的窗口的位置和大小，但是通过 XGetWindowAttributes 又不能获取窗口的最小宽度和最小高度。

（3）读取窗口状态

在 EWMH 中，规定了窗口的状态属性 _NET_WM_STATE 包括 _NET_WM_STATE_MODAL、_NET_WM_STATE_MAXIMIZED_VERT、_NET_WM_STATE_MAXIMIZED_HORZ、_NET_WM_STATE_FULLSCREEN 等。

为简单起见，winman 中只示例处理了窗口最大化的状态。函数 ewmh_get_net_wm_state 调用 Xlib 的接口 XGetWindowProperty 读取窗口的属性 _NET_WM_STATE。如果属性中包含 _NET_WM_STATE_MAXIMIZED_VERT 和 _NET_WM_STATE_MAXIMIZED_HORZ，那么就说明窗口是处于最大化状态，则 winman 尝试读取窗口中的标准状态（Restore 状态）下窗口的位置和尺寸信息，以便窗口从最大化切换到标准状态时使用。

读者可能会问，结构体 Client 中数据项 restore_x、restore_y、restore_w、restore_h 不是记录了窗口在最大化之前的位置和尺寸吗？但是设想这样一个场景：当窗口处于最大化时，窗口管理器异常退出了，那么当窗口管理器再次启动时，这个数据如何初始化？这就是 winman 为窗口自定义属性 _CUSTOM_WM_RESTORE_GEOMETRY 的目的，winman 将这些信息保存在窗口中，只要窗口在，这些信息就可以从窗口中读出来，可谓是“人在阵地就在”。

（4）捕捉“旧”窗口

当新的窗口出现后，如果这个新窗口不是一个临时窗口，其将成为当前活动的窗口，而上一个活动窗口将退居二线。因此，winman 需要捕捉这个退居二线的窗口，以便可以将其顺利切换回来。而且，这个退居二线的窗口可能还有临时窗口，而且临时窗口可能还有临时窗口，因此，函数 normal_client_get_transients 遍历窗口栈，返回这个退居二线窗口的临时窗口组成的链，捕捉这个链上的所有窗口。

（5）设置窗口对象的函数指针

创建了窗口对象后，显然需要设置操作窗口的函数指针。根据不同的窗口类型，设置这些指针指向不同的函数实现。对于标准窗口，设置这些指针指向标准窗口的实现。

（6）更新根窗口的属性

新窗口的“自然人”信息收集完毕后，就可以给其“落户”了。函数 normal_client_new 调用 stack_append_top 将新创建的窗口对象压入 winman 自己维护的窗口栈。

为了让其他应用知晓又有新成员加入了，当然需要更新一些状态信息，比如任务条就时刻关注着系统中应用的变化情况。一个是记录当前活动窗口的属性 _NET_ACTIVE_WINDOW；另外一个是记录 X 服务器中所有窗口的列表的属性 _NET_CLIENT_LIST_STACKING。这两个属性都是 EWMH 标准规定的，它们都是根窗口的属性。函数 normal_client_new 中调用的两个子函数 ewmh_set_net_active_window 和 ewmh_update_net_client_list_stacking 目的就是分别更新这两个属性。

7.1.6 构建窗口装饰

仅给窗口“落户”还是不够的，接下来我们还需要为窗口构建装饰。除了起到美化作用外，这些装饰还是用户和应用的窗口之间的桥梁。用户可以通过标题栏移动窗口位置，可以通过边框改变窗口尺寸，可以点击最大化按钮将窗口最大化，可点击最小化按钮将窗口最小

化，可以点击关闭按钮关闭窗口。

在创建了窗口对象后，函数 wm_new_client 中调用窗口对象中函数指针 reparent 指向的函数来构建窗口装饰。对于标准窗口来说，构建窗口装饰的函数是 normal_client_reparent，代码如下：

winman/src/normal_client.c：

```
static void normal_client_reparent(Client *c)
{
    XSetWindowAttributes attr;
    WinMan *wm = c->wm;
    int frame_x, frame_y;
    XColor tc, sc;

    if (normal_client_calc_geometry(c))
        XMoveResizeWindow(wm->dpy, c->window, c->x, c->y,
            c->width, c->height);

    XAllocNamedColor(wm->dpy, DefaultColormap(wm->dpy,
        wm->screen), LIGHTGRAY, &sc, &tc);

    attr.background_pixel = sc.pixel;
    attr.override_redirect = True;
    attr.event_mask = SubstructureRedirectMask
                    | SubstructureNotifyMask
                    | ExposureMask
                    | ButtonPressMask
                    | ButtonReleaseMask
                    | Button1MotionMask;
    c->frame = XCreateWindow(wm->dpy, wm->root,
            c->x - BORDER_WIDTH,
            c->y - BORDER_WIDTH - TITLEBAR_HEIGHT,
            c->width + BORDER_WIDTH * 2,
            c->height + TITLEBAR_HEIGHT + BORDER_WIDTH * 2, 0,
            CopyFromParent, CopyFromParent, CopyFromParent,
            CWOverrideRedirect | CWBackPixel | CWEventMask,
            &attr);
    XDefineCursor(wm->dpy, c->frame,
        XCreateFontCursor(wm->dpy, XC_arrow));

    attr.event_mask = ExposureMask;
    c->titlebar = XCreateWindow(...);
    ...
    c->rsz_ul_angle = XCreateWindow(wm->dpy, c->frame,
        0, 0, RSZ_ANGLE_SIZE, RSZ_ANGLE_SIZE, 0,
        CopyFromParent, InputOnly, CopyFromParent,
        CWOverrideRedirect, &attr);

    XDefineCursor(wm->dpy, c->rsz_ul_angle,
        XCreateFontCursor(wm->dpy, XC_ul_angle));
    XLowerWindow(wm->dpy, c->rsz_ul_angle);
    ...
    XAddToSaveSet(wm->dpy, c->window);
```

```
    XReparentWindow(wm->dpy, c->window, c->frame, BORDER_WIDTH,
        TITLEBAR_HEIGHT + BORDER_WIDTH);
}
```

函数 normal_client_reparent 执行的主要操作如下：

（1）调整窗口位置和尺寸

在显示窗口前，winman 调用函数 normal_client_calc_geometry 对窗口的位置和尺寸进行了一些合理性检查。对于某些不合理的值，函数 normal_client_calc_geometry 会进行简单的修正，比如不能允许窗口显示在屏幕的可见范围之外。如果经过函数 normal_client_calc_geometry 计算发现窗口确实需要修正，normal_client_reparent 则调用 Xlib 的函数 XMoveResizeWindow 请求 X 服务器对窗口进行调整。

（2）创建 Frame 窗口

正所谓“皮之不存，毛将焉附”，Frame 作为承载窗口装饰的载体，normal_client_reparent 首先调用 Xlib 的函数 XCreateWindow 来创建这个窗口。Frame 的父窗口是根窗口；几何尺寸基于应用的顶层窗口的尺寸，并为容纳窗口边框以及标题栏等预留出空间；其他如窗口类型等属性均采用与根窗口相同的值即可，也就是代码中的为函数 XCreateWindow 传递参数 CopyFromParent 的意图。

winman 通过结构体 XSetWindowAttributes 设置了 Frame 窗口的另外三个属性：override_redirect、background_pixel 和 event_mask。

override_redirect 表示窗口是否接受窗口管理。Frame 窗口，包括后面的标题栏等其他装饰窗口，当然不再需要窗口管理器管理，因此，这个属性设置为 True。

X 服务器中可以创建一个或者多个颜色映射（Colormap），每个颜色映射就是一个数组，数组中的每个元素代表一个颜色。颜色映射数组的大小，由显示器的色深决定。应用使用颜色时，首先要根据颜色的名字，在颜色映射中找到对应的索引，X 将这个过程称为分配颜色如图 7-8 所示。

函数 normal_client_reparent 使用 Xlib 的函数 XAllocNamedColor 分配了一个颜色，然后将这个颜色通过属性 background_pixel 指定为 Frame 窗口的背景色。

winman 通过属性 event_mask 选择接收 Frame 窗口的事件。winman 依然需要拦截应用的顶层窗口的请求、获取它们的某些通知，而应用的顶层窗口已经成为 Frame 的子窗口了，所以 winman 需要接收 Frame 的子窗口的事件，这就是选择 Frame 窗口的事件掩码 SubstructureRedirectMask 和 SubstructureNotifyMask 的目的。Frame 窗口也需要绘制，所以选择了 ExposureMask。事件掩码中最后选择接收的就是几个鼠标事件，winman 按照下面的逻辑处理 Frame 及作为其子窗口的各个装饰窗口间的鼠标事件。

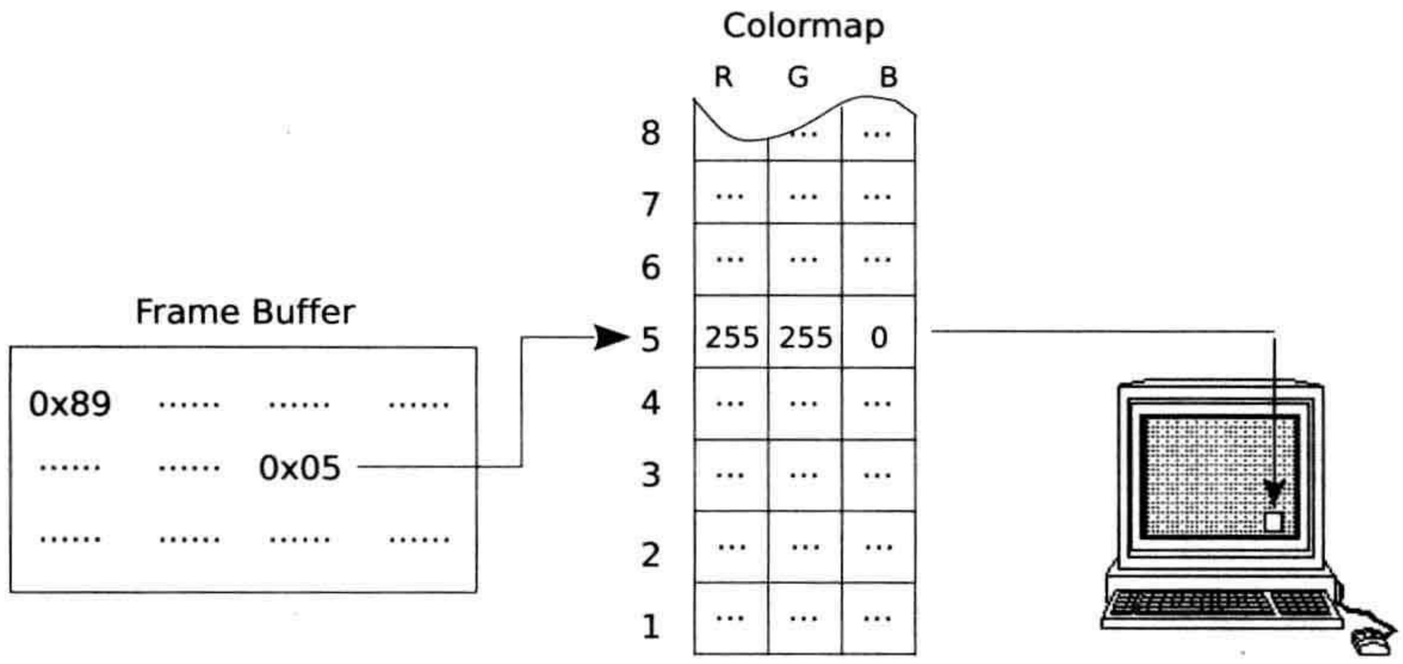


图 7-8　X 颜色映射

winman 仅接收 Frame 的鼠标事件，而不接受其他装饰窗口的鼠标事件。所以，当鼠标事件发生在任何装饰窗口上时，按照 X 的事件传播机制，鼠标事件最终都会在窗口树中向上传播到 Frame。也就是说，winman 最终接收到的是来自 Frame 窗口的鼠标事件，鼠标事件中的参数 window 是 Frame，但是 subwindow 则是鼠标事件发生时，鼠标指针所在的真实的装饰窗口。后面，当处理鼠标事件时，winman 就是利用鼠标事件的参数 subwindow 判断发生在哪个窗口装饰上了，从而判读出用户的意图，比如是关闭窗口，最小化窗口，抑或是移动窗口等。

（3）设置 Frame 窗口鼠标指针形状

Xlib 提供了函数 XDefineCursor 为窗口设置鼠标指针的形状，winman 将 Frame 窗口的指针设置为 XC_arrow，即常用的箭头形状。

（4）创建标题栏、最大化、最小化及关闭按钮

winman 为这几个窗口装饰分别创建了窗口，基本与创建 Frame 窗口相同。它们的父窗口是 Frame 窗口。winman 只接收这几个窗口的 Expose 事件，毕竟还需要在按钮上绘制图标。winman 也无需为这几个窗口定义鼠标指针，它们使用与父窗口 Frame 相同的鼠标指针。

（5）创建“改变窗口尺寸”指示区域

接下来 winman 还要创建几个幕后英雄，即前面提到的以 rsz_ 开头的几个窗口。这几个窗口的目的非常单纯，就是为了当鼠标进入这个区域时，显示特殊的鼠标指针形状，提示用户可以在这个区域改变窗口尺寸了。

这几个窗口不需要显示任何内容，因此窗口的类型设置为 InputOnly；而且也不需要接收任何事件，所以无须设置任何事件掩码；它们也不需要接受窗口管理器管理，所以窗口属性 override_redirect 设置为 True。

为了给用户一个直观的提示，这几个窗口的鼠标指针分别设置为不同的形状，比如窗口正上方的鼠标指针设置为 XC_top_side，左上角设置为 XC_ul_angle，等等。

对于位于四个角的窗口，winman 还调用了 Xlib 的函数 XLowerWindow，其目的是什么

呢？图 7-9 展示了放大了的窗口右上角的区域，窗口 rsz_ur_angle 是正方形形状的窗口，但是我们希望鼠标指针只有落在图中使用灰色标出的区域时才允许用户改变窗口尺寸。并且窗口 rsz_ur_angle 不应该遮挡关闭按钮，导致其失效，因此 winman 使用 XLowerWindow 将 rsz_ur_angle 置于窗口栈的底部，以免遮挡其他兄弟窗口。

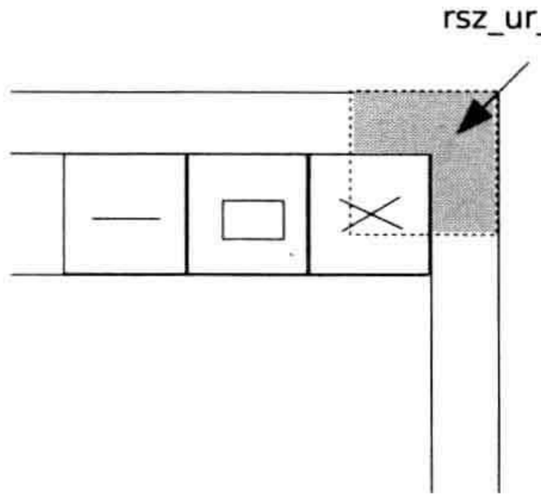


图 7-9 放大的窗口右上角区域

（6）将应用顶层窗口加入 save-set

如前面讨论的，我们不希望 winman 异常退出时，因为销毁 Frame 窗口而导致作为 Frame 子窗口的应用的顶层窗口也受牵连，因此 winman 调用 Xlib 的函数 XAddToSaveSet 将应用的顶层窗口加入到 save-set 中。

如果读者注释掉函数 normal_client_reparent 中的 XAddToSaveSet，然后尝试终止窗口管理器，就会发现应用的窗口也随之被销毁了。

当 save-set 中的窗口被销毁时，X 服务器负责从 save-set 中将它们移除。因此，当应用的顶层窗口“扬长而去”时，winman 无需调用 XRemoveFromSaveSet 从 save-set 中移除它们。

（7）将应用的顶层窗口作为 Frame 的子窗口

Frame 窗口以及作为其子窗口的其他装饰，全部准备完毕。最后，函数 normal_client_reparent 调用 Xlib 的 XReparentWindow 函数将 Frame 作为应用的顶层窗口的父窗口，完成最后一击。

7.1.7 绘制装饰窗口

在 7.1.6 节中，winman 创建了各个装饰窗口，但是并没有为各装饰窗口绘制内容。事实上，即使 winman 想去绘制，也是有心无力。基于 X 的原理，X 服务器并不保存窗口的内容，在窗口可见时，X 服务器会向应用报告 Expose 事件，应用收到这个事件后，开始绘制。否则即使应用在创建窗口时自说自话地进行了绘制，也会被丢掉。

因此，在函数 wm_new_client 中，在构建了窗口装饰后，调用了窗口对象中函数指针 show 指向的函数，请求 X 服务器进行显示。对于标准窗口来说，请求 X 服务器显示窗口是 normal_client_show，代码如下：

```
winman/src/normal_client.c:

static void normal_client_show(Client *c)
{
    WinMan *wm = c->wm;

    XMapWindow(wm->dpy, c->frame);
    XMapSubwindows(wm->dpy, c->frame);
}
```

在接受了 winman 的显示请求后，X 服务器将向 winman 发送 Expose 事件。收到 Expose 事件后，winman 将绘制窗口装饰。标准窗口的处理 Expose 事件的函数如下：

```
winman/src/normal_client.c:

static void normal_client_redraw(Client *c)
{
    WinMan *wm = c->wm;
    GC gc = XCreateGC(wm->dpy, wm->root, 0, 0);

    Pixmap cm_close = XCreateBitmapFromData(wm->dpy, wm->root,
        close_bits, close_width, close_height);
    ...
    XSetForeground(wm->dpy, gc, BlackPixel(wm->dpy, wm->screen));

    draw_raised(wm, c->titlebar, 0, 0,
        c->width - TITLEBAR_HEIGHT * 3, TITLEBAR_HEIGHT);
    draw_raised(wm, c->close_btn, 0, 0, TITLEBAR_HEIGHT,
       TITLEBAR_HEIGHT);
    XSetClipMask(wm->dpy, gc, cm_close);
    XSetClipOrigin(wm->dpy, gc, 2, 2);
    XFillRectangle(wm->dpy, c->close_btn, gc, 0, 0,
       TITLEBAR_HEIGHT, TITLEBAR_HEIGHT);
    XSetClipMask(wm->dpy, gc, None);
    ...
    draw_raised(wm, c->frame, 0, 0, c->width + BORDER_WIDTH * 2,
        c->height + TITLEBAR_HEIGHT + BORDER_WIDTH * 2);
    draw_lowered(wm, c->frame, BORDER_WIDTH - 2, BORDER_WIDTH - 2,
        c->width + 3, TITLEBAR_HEIGHT + c->height + 3);

    XFreeGC(wm->dpy, gc);
}
```

该函数执行的主要操作如下：

1）为标题栏绘制边框，使标题栏看上去更富立体感。

2）为标题栏上的关闭等各个按钮绘制图标，并为它们也绘制边框。

3）为窗口绘制边框。

事实上，所谓的绘制边框就是在窗口边上绘制线条，但是不同的线条使用不同的颜色，依据色差来产生立体感。函数 draw_raised 和 draw_lowered 就是做这件事的。

对于标题栏上关闭等按钮的图标，winman 使用了类似掩码的方法来绘制。以函数

XFillRectangle 为例，在默认情况下，将使用图形上下文（GC）中指定的前景色填充矩形。但是，如果设置了 GC 中的 clip_mask，那么 clip_mask 中凡是值为“1”的位，依然使用前景色填充，但是值为“0”的位则不会进行填充，如图 7-10 所示。

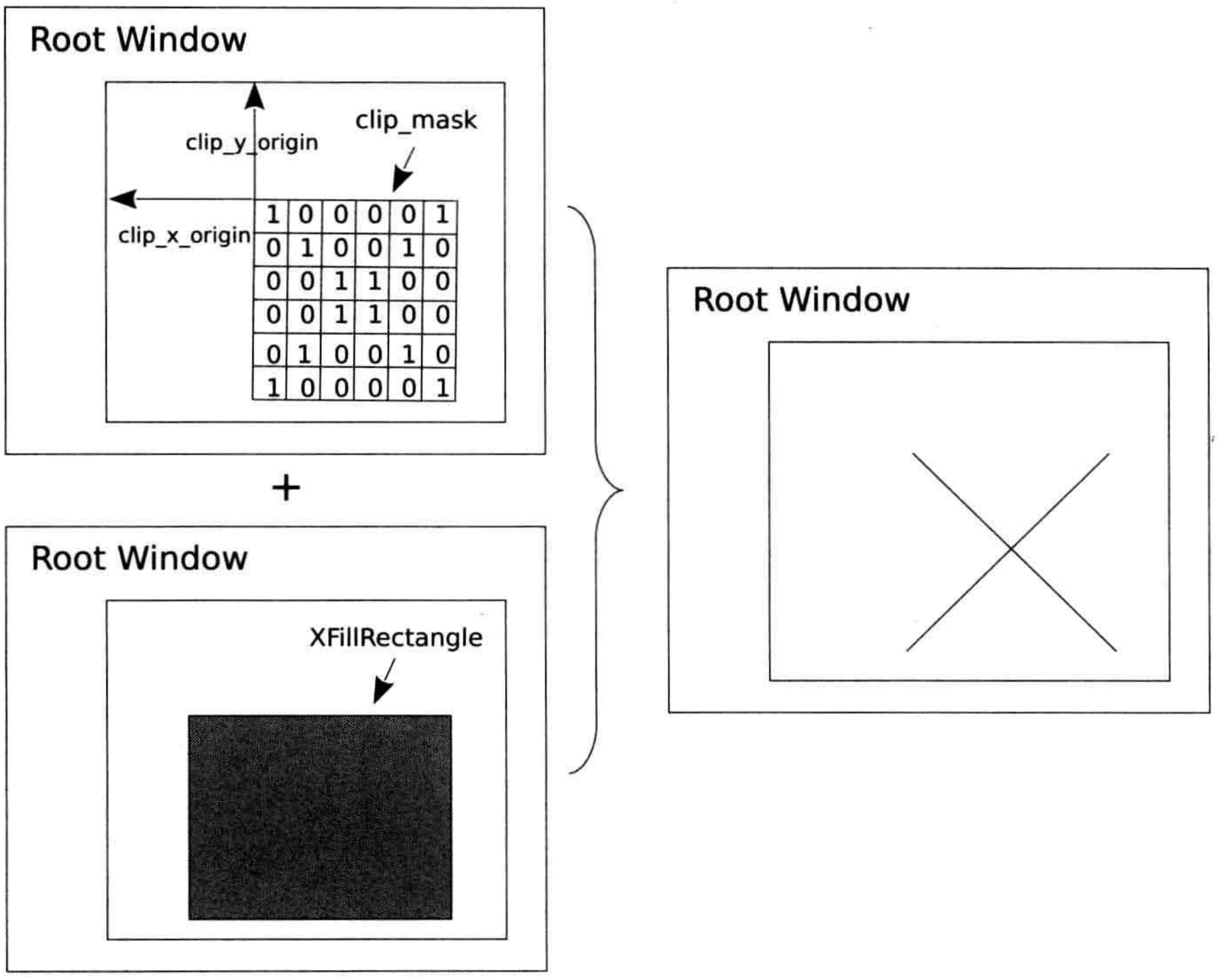


图 7-10　clip_mask 使用示意图

7.1.8　配置窗口

通常，我们在编写具有图形界面的应用程序时，在显示图形界面之前，一定会设置窗口的位置、尺寸或者边框宽度等。虽然读者可能反驳说，我们有时并没有设置这些啊？实际上，那是因为如 GTK、QT 等图形库已经帮我们做了。另外一种情况是在应用运行的某个中间时刻，应用也可能会改变窗口的这些信息。

X 将这些信息统称为窗口配置，包括窗口的位置、宽度和高度，边框的宽度以及在栈中的位置。

在上述两种情况下，应用都将产生配置请求，X 服务器也都会将它们重定向给窗口管理器。那么窗口管理器如何区分这两种情况呢？ winman 是这样处理的，当收到 X 服务器重定向来的配置请求时，winman 调用函数 wm_find_client_by_window 遍历窗口栈，如果窗口栈中没有一个窗口对象与发送请求的窗口匹配，就说明这个窗口尚未被管理，否则说明这个窗口已经被管理了。

对于尚未纳入管理的应用的窗口，winman 当然不能贸然管理，谁知道未来它是否需要管理呢。所以直接请求 X 服务器满足其需要。winman 从事件 XConfigureRequestEvent 中提取信息，不加任何修改，完全照搬原来的配置请求，使用 Xlib 的函数 XConfigureWindow 直接代替应用向 X 服务器发出配置请求。

对于已被管理的窗口，winman 调用具体窗口对象中处理配置的函数进行具体的配置。相关代码如下：

winman/src/main.c:

```
static void wm_handle_configure_request(WinMan *wm,
                XConfigureRequestEvent *e)
{
    Client *c;
    XWindowChanges xwc;
    int border_width;
    int screen_width, screen_height;

    c = wm_find_client_by_window(wm, e->window);

    if (!c) {
        xwc.x = e->x;
        xwc.y = e->y;
        xwc.width = e->width;
        ...
        XConfigureWindow(wm->dpy, e->window, e->value_mask, &xwc);
    } else
        c->configure(c, e);

    return;
}
```

我们看到，当 winman 不了解“敌情”时，它直接调用 Xlib 的函数 XConfigureWindow 将“皮球”又踢给了 X 服务器。而对于已经在自己掌控之下的窗口，则调用具体窗口对象的配置处理函数进行配置。以标准窗口对象为例，函数指针 configure 指向 normal_client_configure，其代码如下：

winman/src/normal_client.c

```
static void normal_client_configure(Client *c,
    XConfigureRequestEvent *e)
{
    if (e->value_mask & CWX)
        c->x = e->x;
    ...
    if (e->value_mask & (CWX | CWY | CWWidth | CWHeight)) {
        normal_client_calc_geometry(c);
        c->move_resize(c);
    }
}
```

```
static void normal_client_move_resize(Client *c)
{
    WinMan *wm = c->wm;

    XMoveResizeWindow(wm->dpy, c->frame, ...);
    XMoveResizeWindow(wm->dpy, c->window, ...);
    XMoveWindow(wm->dpy, c->close_btn, ...);
    ...
}
```

该函数的核心就是调用 Xlib 的 XMoveResizeWindow 系列函数，满足窗口的配置请求。但是有一点需要注意，因为应用的顶层窗口的配置改变了，所以所有的窗口装饰可能都需要进行调整。

7.1.9 移动窗口

对于一个典型的桌面应用来说，用户可以通过拖动窗口的标题栏来移动窗口。这里所谓的“拖动”的具体动作是：在标题栏上按下鼠标左键并保持，然后移动鼠标，直到释放鼠标。

因此，整个移动窗口的过程可以划分为三个阶段：

1）用户在标题栏上按下鼠标左键并保持；

2）用户移动鼠标；

3）用户释放鼠标，移动结束。

为此，结构体 Client 中设计了布尔变量 moving，当用户在标题栏内按下鼠标左键时，moving 将被置为 True。当鼠标移动事件发生时，如果变量 moving 为 True，那么我们就可以断定用户是在移动窗口。一旦用户释放了鼠标，winman 将 moving 更改为 False。

1. 按下鼠标

一旦收到鼠标按下事件，winman 首先要找到鼠标事件发生在哪个窗口对象上，winman 中封装的函数 wm_find_client_by_frame 就是做这件事的。找到了具体的窗口对象后，则调用这个窗口对象的指针 button_press 指向的函数，代码如下：

```
winman/src/main.c:

static void wm_handle_button_press(WinMan *wm, XButtonEvent *e)
{
    Client *c;
    ...
    if (c = wm_find_client_by_frame(wm, e->window))
        c->button_press(c, e);
}
```

我们还是以标准窗口对象为例，其处理鼠标按下事件的函数是 normal_client_button_press，代码如下：

```
winman/src/normal_client.c:
```

```
static void normal_client_button_press(Client *c, XButtonEvent *e)
{
    ...
    } else if (e->subwindow == c->titlebar) {
        c->moving = True;
        c->anchor_x = e->x;
        c->anchor_y = e->y;
    } else if (e->subwindow == c->rsz_top_side
    ...
}
```

函数 normal_client_button_press 检查鼠标事件中的成员 subwindow 是否是标题栏，如果是，则设置窗口对象的 moving 为 True。读者可能有个疑问，如果用户只是虚晃一枪，马上又释放了鼠标呢？那也没有什么影响，因为 winman 在鼠标释放的事件处理函数中，将把这个值重置为 False。

同时，函数 normal_client_button_press 将鼠标事件发生的位置记录到窗口对象中的 anchor_x 和 anchor_y 中。注意，winman 记录的位置使用的是窗口内部坐标，是相对于窗口原点的。后面将以这个点为参考，计算窗口移动后的新位置。

2. *移动鼠标*

在收到鼠标移动通知后，winman 首先还是要找到具体的窗口对象，然后调用这个窗口的函数指针 motion 指向的函数，代码如下：

winman/src/main.c:

```
static void wm_handle_motion_notify(WinMan *wm, XMotionEvent *e)
{
    Client *c = wm_find_client_by_frame(wm, e->window);

    if(c)
        c->motion(c, e);
}
```

以标准窗口对象为例，其处理鼠标移动事件的函数是 normal_client_motion，代码如下：

winman/src/normal_client.c:

```
void normal_client_motion(Client *c, XMotionEvent *e)
{
WinMan *wm = c->wm;
int x, y, width, height;
int frame_x, frame_y;

if (c->moving) {
    frame_x = e->x_root - c->anchor_x;
    frame_y = e->y_root - c->anchor_y;

    XMoveWindow(wm->dpy, c->frame, frame_x, frame_y);
    ...
} else if (c->resizing_area) {
```

```
        ...
}
```

函数 normal_client_motion 首先检查窗口对象中的变量 moving。如果 moving 为 True，则说明用户正在拖动标题栏。因此，其根据当前的鼠标所在的位置以及在鼠标按下时所在的位置，即窗口对象中的变量 anchor_x 和 anchor_y，计算出窗口新的位置，具体的计算方法参见图 7-11。然后调用 Xlib 的函数 XMoveWindow 移动窗口。

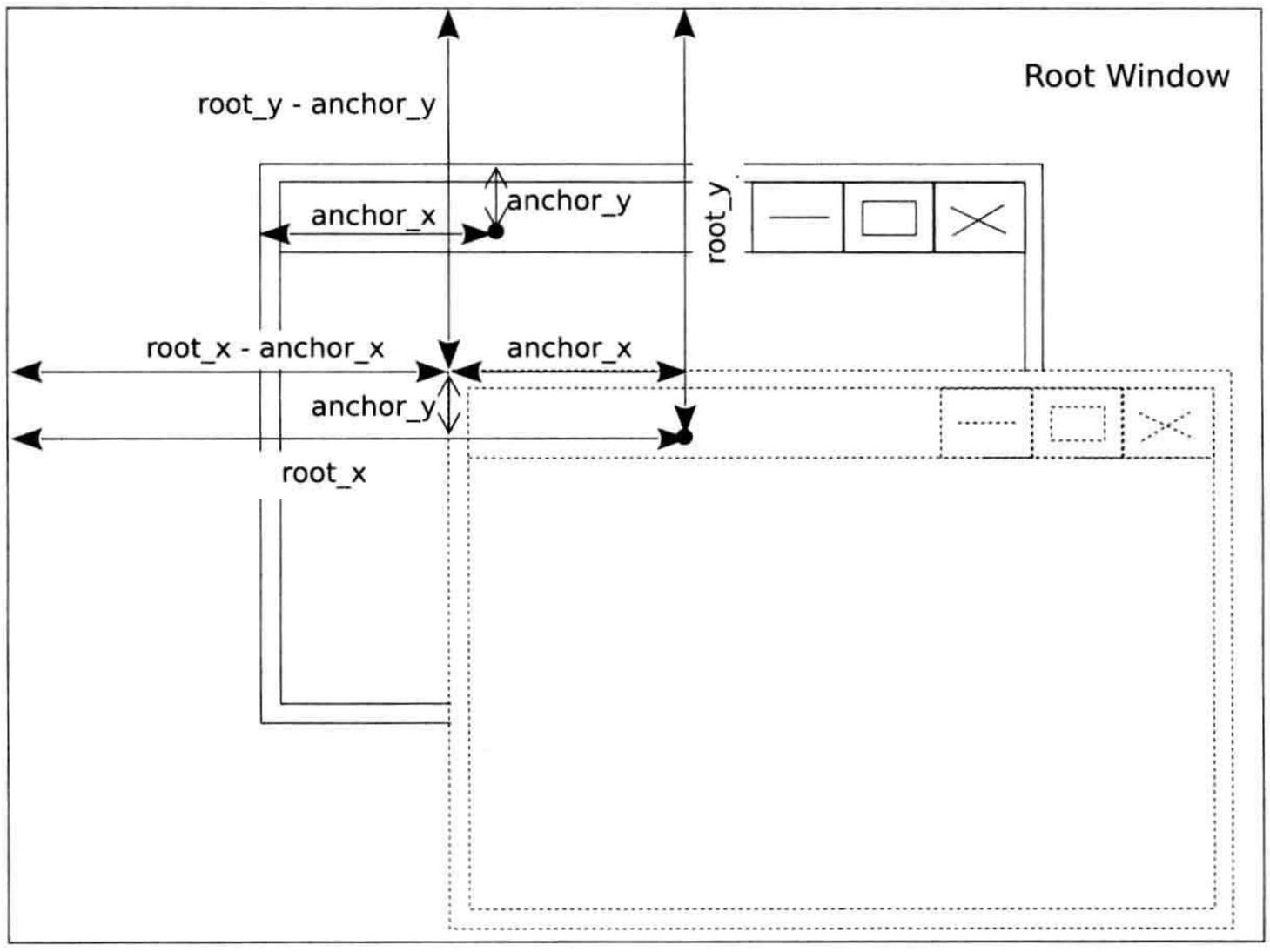


图 7-11 窗口移动位置计算

看过上面的实现，读者可能会有个疑问：一定要设置 moving 吗？不可以在移动时根据事件中的子窗口（subwindow）是否是标题栏来判断是否是在拖动标题栏吗？即将下面的语句：

```
if (c->moving)
```

更改为：

```
if (e->subwindow == c->titlebar )
```

答案是不可以。原因是，如果鼠标移动较慢，那么在移动时鼠标指针会一直落在标题栏中，这没有问题。但是如果用户移动得非常快，窗口移动的速度跟不上鼠标，这时鼠标所在的子窗口，即鼠标移动事件中的成员 subwindow，或者是 0，或者是其他窗口，总之，不再是标题栏了。

3. 释放鼠标

在收到鼠标释放事件后，winman 首先找到具体的窗口对象，然后调用这个窗口的函数指

针 button_release 指向的函数，代码如下：

```
winman/src/main.c:

static void wm_handle_button_release(WinMan *wm, XButtonEvent *e)
{
    Client *c = wm_find_client_by_frame(wm, e->window);

    if (c)
        c->button_release(c, e);
}
```

以标准窗口对象为例，其处理鼠标释放事件的函数是 normal_client_button_release，代码如下：

```
winman/src/normal_client.c:

static void normal_client_button_release(Client *c,
             XButtonEvent *e)
{
    ...
    } else if (c->moving) {
        c->moving = False;
    } else if (c->resizing_area) {
    ...
}
```

当鼠标释放时，表明用户已经停止移动窗口，winman 将窗口对象中的 moving 标志清除。

7.1.10 改变窗口大小

改变窗口大小与移动窗口的操作逻辑上基本相同，这里只简要讨论实现的逻辑，就不再列出具体代码了，请读者自行参考随书光盘中附带的源代码。

在用户按下鼠标事件时，将鼠标指针所在的标识移动区域的窗口，也就是结构体 Client 中以 rsz_ 开头的窗口，记录到窗口对象的成员 resizing_area 中。

然后，当收到鼠标移动事件时，如果窗口对象的成员 resizing_area 非 0，那就说明用户正在试图改变窗口大小。根据 resizing_area 与 8 个标识移动区域的窗口对比，推断出用户正在如何更改窗口的大小，然后计算出窗口改变后的几何信息，请求 X 服务器改变窗口大小。

当鼠标释放时，将 resizing_area 清 0。

7.1.11 切换窗口

我们以图 7-12 所示的场景为例来讨论窗口之间的切换。应用 A 和应用 B 分别为两个 X 应用，图中使用虚线标识的是应用创建的窗口，实线标出的是窗口管理器创建的窗口。A1 是应用 A 的标准类型的顶层窗口，A2 是对话框类型的顶层窗口，且 A2 是窗口 A1 的临时窗口。B1 是应用 B 的标准类型的顶层窗口，B2 是对话框类型的顶层窗口，且 B2 是窗口 B1 的临时

窗口。初始状态 X 时，应用 A 是当前活动的应用；在状态为 Y 时，应用 B 被切换为当前的活动应用。

当将应用 B 切换为当前应用时，窗口管理器需要考虑以下两点：

- 窗口管理器不应限制用户只有点击在窗口管理器完全控制的装饰窗口上才可以切换，即使鼠标指针落在应用自己创建的窗口上，如 B1 窗口、B2 窗口，窗口管理器也应将应用 B 切换为当前活动应用。
- 窗口管理器应以整个应用为单位进行切换，即将应用 B 的所有窗口都移动到 X 服务器窗口栈的顶端。切换完成后，窗口的栈序应该为 B2 → B1 → A2 → A1，而不是类似如 B2 → A2 → A1 → B1。

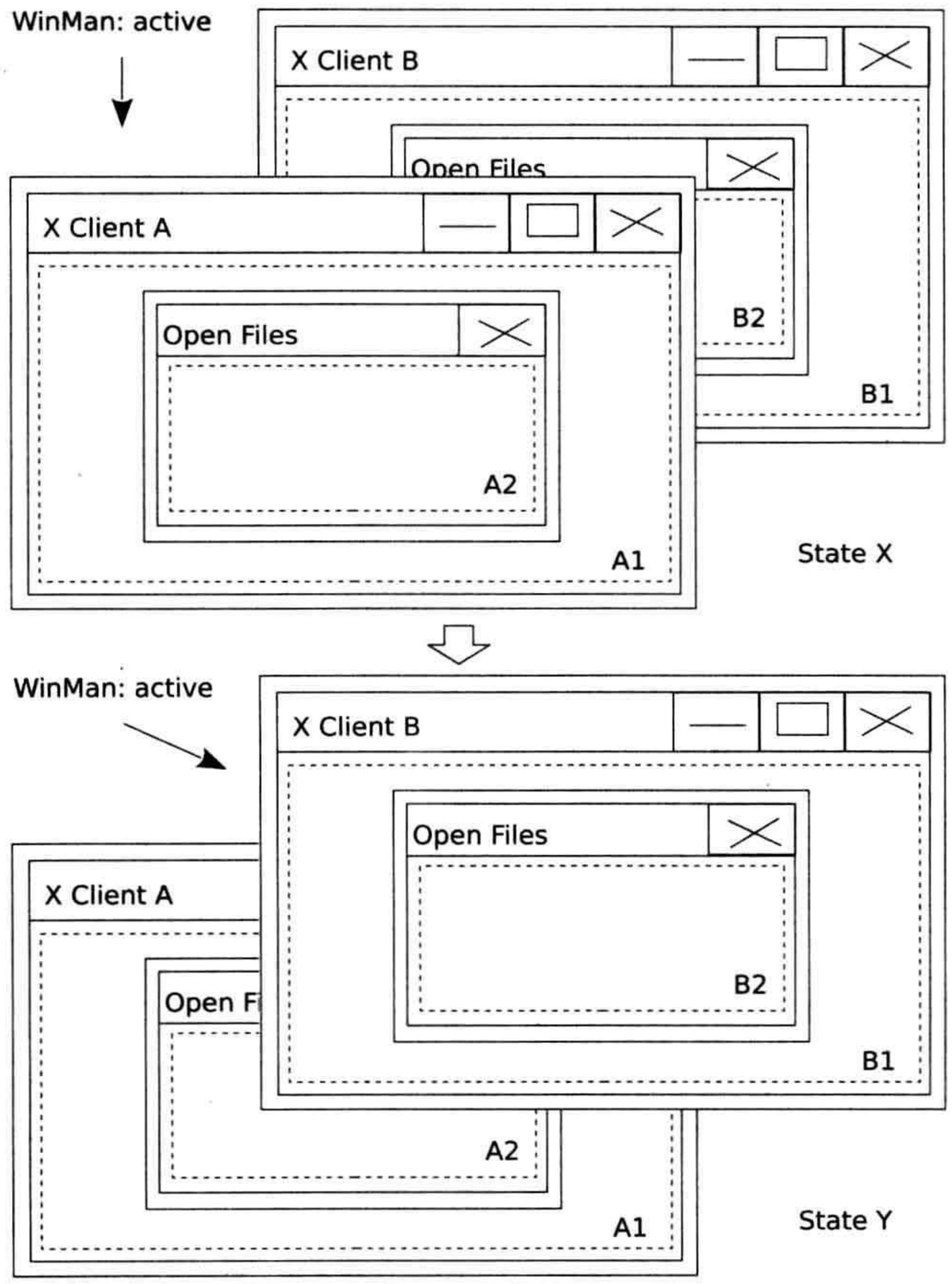


图 7-12 切换窗口

理解了上面两点后，我们来看具体的切换实现，代码如下：

winman/src/main.c:

```
static void wm_handle_button_press(WinMan *wm, XButtonEvent *e)
```

```
{
    Client *c;
    Client *topmost;

    if ((c = wm_find_client_by_frame(wm, e->window))
      || (c = wm_find_client_by_window(wm, e->window))) {
        if (!c->trans_for) {
            if (c != wm->active)
                c->activate(c);
        } else {
            topmost = transient_get_topmost(c);
            if (topmost != wm->active)
                c->activate(topmost);
        }
    }
    ...
}
```

函数 wm_handle_button_press 中与切换相关的主要操作如下：

1）winman 首先在窗口栈中寻找鼠标点击事件发生的窗口对象。winman 既考虑了鼠标可能点击在窗口管理器创建的装饰窗口，也考虑了鼠标可能点击在应用创建的窗口的情况。

2）如果鼠标点击的窗口不是临时窗口，并且也不是当前活动的窗口，则调用函数指针 activate 指向的函数进行切换。

3）但是如果用户点击在了临时窗口上，winman 调用函数 transient_get_topmost 一直找到非临时窗口，如果这个非临时窗口不是当前活动的窗口，那么调用函数指针 activate 指向的函数进行切换。

以标准窗口为例，函数指针 activate 指向函数为 normal_client_activate，代码如下：

```
winman/src/normal_client.c:

static void normal_client_activate(Client *c)
{
    WinMan *wm = c->wm;
    Item *trans, *i;

    stack_remove(c);
    stack_append_top(c);

    trans = normal_client_get_transients(c);
    for (i = trans; i; i = i->next) {
        stack_remove(i->client);
        stack_append_top(i->client);
        XUngrabButton(wm->dpy, Button1, 0, i->client->window);
    }
    list_free(&trans);

    wm_restack_clients(wm);

    XUngrabButton(wm->dpy, Button1, 0, c->window);
    XAllowEvents(wm->dpy, ReplayPointer, CurrentTime);
```

```
if (wm->active) {
    XGrabButton(wm->dpy, Button1, 0, wm->active->window,
          True, ButtonPressMask, GrabModeSync,
          GrabModeSync, None, None);

    trans = normal_client_get_transients(wm->active);
    for (i = trans; i; i = i->next) {
        XGrabButton(wm->dpy, Button1, 0, i->client->window,
              True, ButtonPressMask, GrabModeSync,
              GrabModeSync, None, None);
        }
        list_free(&trans);
    }

    wm->active = c;
    ewmh_set_net_active_window(wm->active);
}
```

该函数 normal_client_activate 执行的主要操作如下：

1）首先要调整窗口栈序，将准备激活的窗口放到窗口栈的最顶端。

2）如果窗口还有临时窗口，那么也要将临时窗口移动到栈的最顶端。因为临时窗口可能还有临时窗口，这就是代码中 for 循环的目的。

3）安排好窗口的栈序后，函数 normal_client_activate 调用函数 wm_restack_clients 请求 X 服务器重新调整窗口栈序。函数 wm_restack_clients 就是将 winman 的窗口栈中的窗口按照 Xlib 的函数 XRestackWindows 的参数格式组织好，然后调用 Xlib 的函数 XRestackWindows 向 X 服务器发出调整请求。从这里我们再次深刻地体会到 X 策略和机制的分离哲学：X 服务器负责具体的切换窗口的动作，但是，各个窗口的前后顺序如何排列这个策略由窗口管理器来负责。

4）切换窗口后，winman 还有一件事情要做，那就是取消捕捉刚刚晋升的活动窗口。捕捉的目的是为了接收非当前活动窗口的鼠标事件，而对于当前活动的窗口显然没有必要继续捕捉了。而且如果不取消捕捉窗口的鼠标事件，那么每次点击窗口时，鼠标事件都会送给 winman, 这显然是没有必要的，而且徒增 X 服务器和窗口管理器之间的通信量。

5）接下来，函数 normal_client_activate 调用 Xlib 的函数 XAllowEvents 放行捕捉的鼠标事件。这个事件就像一个接力棒一样，在 winman 中逗留了一圈，又回到其真正属于的应用，不至于造成事件丢失。但是前提是捕捉时必须使用同步模式。

6）有放就要有抓，这样才能张弛有度。winman 需要捕捉上一个活动但是马上就变为非活动的窗口，包括其临时窗口。可见 winman 不仅只“见新人笑”，也“闻旧人哭”。

7）最后，winman 更新了根窗口的属性 _NET_ACTIVE_WINDOW。因为有些应用可能会通过根窗口的这个属性获取当前的活动窗口，比如后面的窗口组件任务条。

7.1.12 最大化 / 最小化 / 关闭窗口

本节我们讨论最大化、最小化及关闭窗口的相关知识。

1. 最小化窗口

最小化窗口本质上就是取消窗口的显示，Xlib 为此提供了相应的函数 XUnmapWindow。当取消某个窗口的显示时，同时也需要取消其临时窗口的显示，代码如下：

winman/src/normal_client.c：

```
static void minimize_window(Client *c)
{
    WinMan *wm = c->wm;

    XUnmapWindow(wm->dpy, c->frame);

    Item *trans = normal_client_get_transients(c);
    Item *i;
    for (i = trans; i; i = i->next)
        XUnmapWindow(wm->dpy, i->client->frame);
    list_free(&trans);
}
```

如果取消了一个窗口的显示，那么如何再恢复它的显示呢？在 7.2 节，我们再来讨论这个问题。

2. 最大化 / 恢复窗口

所谓的最大化 / 恢复窗口，本质上就是使用 Xlib 的类似如 XMoveResizeWindow 的函数调整窗口位置和大小，winman 中代码中对应的实现是 maximize window 函数。

唯一需要指出的就是，winman 自定义了属性 _CUSTOM_WM_RESTORE_GEOMETRY，在最大化之前将窗口的几何信息，包括位置、高度和宽度，都记录到窗口的属性中。为什么要在窗口的属性中记录，不是都已经记录到窗口对象中了吗？试想一下，如果不在窗口的属性中记录，而只是记录在窗口管理器中，一旦窗口管理器异常退出，那么一切状态信息将随着窗口管理器灰飞烟灭。为了窗口管理器再次启动时能获得这些信息，将这些信息保存在窗口中是一个合理的办法。

类似地，在最大化 / 恢复窗口时，winman 也更新了窗口的另外两个属性 _NET_WM_STATE_MAXIMIZED_VERT 和 _NET_WM_STATE_MAXIMIZED_HORZ。

3. 关闭窗口

ICCCM 规范规定，当关闭窗口时，窗口管理器应该发送消息 WM_DELETE_WINDOW 给应用，而不是越俎代庖地请求 X 服务器去销毁应用的窗口。因为应用收到消息 WM_DELETE_WINDOW 后，可以做一些善后处理，然后在请求 X 服务器关闭窗口。

当然，有些应用程序不是很守规矩，尤其是早期使用 Xlib 编写的程序，它们不处理消息 WM_DELETE_WINDOW。对于这类窗口，也只能采用简单粗暴的方法了，直接使用 Xlib 提供的函数 XKillClient 断开应用程序到 X 服务器的连接，这也就意味着整个 X 应用彻底退出执行。

那么窗口管理器如何得知应用是否处理了事件 WM_DELETE_WINDOW？ ICCCM 规范

规定，如果窗口自己负责销毁，其应该在窗口的属性 WM_PROTOCOLS 中设置属性 WM_DELETE_WINDOW。属性 WM_PROTOCOLS 的值是个 Atom 数组，其中包括多个属性。

我们来看一下 winman 中的相关代码。当用户点击关闭按钮时，winman 将调用函数 icccm_delete_window，代码如下：

winman/src/ewmh_icccm.c:

```
void icccm_delete_window(Client *c)
{
    WinMan *wm = c->wm;
    Atom *protocols;
    int i, n, found = 0;
    XEvent ev;

    if (XGetWMProtocols(wm->dpy, c->window, &protocols, &n)) {
        for (i = 0; i < n; i++) {
            if (protocols[i] == wm->atoms[WM_DELETE_WINDOW]) {
                found++;
                break;
            }
        }
        if (protocols)
          XFree(protocols);
    }

    if (found) {
        memset(&ev, 0, sizeof ev);

        ev.xclient.type = ClientMessage;
        ev.xclient.window = c->window;
        ev.xclient.message_type = wm->atoms[WM_PROTOCOLS];
        ev.xclient.format = 32;
        ev.xclient.data.l[0] = wm->atoms[WM_DELETE_WINDOW];
        ev.xclient.data.l[1] = CurrentTime;

        XSendEvent(wm->dpy, c->window, False, 0L, &ev);
    } else
        XKillClient(wm->dpy, c->window);
}
```

函数 icccm_delete_window 检查窗口的属性 WM_PROTOCOLS，若其中包含属性 WM_DELETE_WINDOW，那么就发消息 WM_DELETE_WINDOW 给窗口，否则调用 Xlib 的函数 XKillClient 直接切断应用和 X 服务器的连接。

无论是采用哪种方式，最终窗口一定会离我们而去的。虽然它们“轻轻的走了，不带走一片云彩”，但是 winman 还是要做一些必要的善后处理的，最起码，要把代表窗口的对象释放了吧。X 服务器在销毁窗口后将会发送通知 UnmapNotify 给窗口管理器，winman 在这个事件的处理函数中为离去的窗口“销户”。以标准窗口为例，其对应的“销户”函数如下：

winman/src/normal_client.c:

```
static void normal_client_remove(Client *c)
{
    WinMan *wm = c->wm;

    stack_remove(c);

    XReparentWindow(wm->dpy, c->window, wm->root, c->x, c->y);

    if(c->frame)
        XDestroyWindow(wm->dpy, c->frame);

    if (c == wm->active) {
        wm->active = stack_get_first_nontransient(wm);

        if (wm->active) {
            XUngrabButton(wm->dpy, Button1, 0, wm->active->window);

            Item *trans = normal_client_get_transients(wm->active);
            Item *i;
            for (i = trans; i; i = i->next)
                XUngrabButton(c->wm->dpy, Button1, 0,
                              i->client->window);
            list_free(&trans);

            ewmh_set_net_active_window(wm->active);
        }
    }

    ewmh_update_net_client_list_stacking(wm);

    free(c);
}
```

函数 normal_client_remove 执行的主要操作如下：

1）首先从窗口栈中将窗口对应的窗口对象移除。

2）让根窗口收容应用的窗口。为什么要做这么一件事呢？因为有些窗口只是暂时取消显示，比如我们经常进行的最小化窗口。所以如果不将应用的窗口从 Frame 窗口拆除，那么接下来销毁 Frame 时，应用的窗口作为 Frame 窗口的子窗口，也将一并被销毁。

3）安全的将应用的窗口从 Frame 窗口脱离后，normal_client_remove 调用 Xlib 的函数 XDestroyWindow 销毁了 Frame 等装饰窗口。

4）如果销毁的窗口是当前的活动窗口，winman 将在窗口栈中试图找到下一个非临时窗口作为当前活动的窗口。如果找到了，那么余下要做的就和切换窗口类似了。

5）当前活动窗口变化了，窗口列表也变了。函数 normal_client_remove 调用相应的函数更新根窗口的属性。这方面的内容前面已多次见过，这里就不再讨论了。

7.1.13 管理已存在的窗口

在窗口管理器启动之前，可能有一些应用已经在运行。因此，在窗口管理器启动时，需

要管理这些已存在的窗口。这就是 winman 的初始化时调用函数 init_clients 的目的。init_clients 的实现代码如下：

winman/src/main.c:

```
static void init_clients(WinMan *wm)
{
    Client *c;
    XWindowAttributes attr;
Window root, parent, *children;
unsigned int n, i;

XQueryTree(wm->dpy, wm->root, &root, &parent, &children, &n);

for (i = 0; i < n; i++) {
    if (XGetWindowAttributes(wm->dpy, children[i], &attr)) {
        if (attr.override_redirect == False
                && attr.map_state == IsViewable) {
            c = wm_new_client(wm, children[i]);
            if (c)
                c->ignore_unmap++;
        }
    }
}

if (children)
        XFree(children);
}
```

函数 init_clients 执行的主要操作如下：

1）调用 Xlib 的函数 XQueryTree 查询根窗口的子窗口，参数 children 指向保存子窗口的内存区，变量 n 记录子窗口的数量。

2）遍历根窗口的子窗口，如果窗口的属性 override_redirect 的值为 False，则表明窗口需要窗口管理器管理；然后检查窗口的显示状态，如果值是 IsViewable，（IsViewable 表示的意思是“Window is viewable”，而不是“Is window viewable？”），则表示窗口是可见的，那么 init_clients 就调用函数 wm_new_client 开始管理窗口。

3）将变量 ignore_unmap 累加 1。为什么需要这么一个变量，并且这里将其累加 1 呢？对于这些窗口，在窗口管理器启动前，它们已经是可见的。而为了管理这些窗口，窗口管理器将使用 Frame 窗口作为它们的父窗口，因此它们当然要首先从根窗口剥离了。换句话说，X 服务器需要将窗口从窗口树中当前的位置删除，并插入到新的位置。X 当然不想让人看到它的这个小动作，因此，在执行这个过程前，X 服务器首先取消这些窗口的显示，也就是说，X 服务器先把窗口藏起来，让它们不可见，然后偷偷摸摸地把它们移动到窗口树中新的位置，移动完成后，再让窗口可见。恰恰是 X 服务器的这个将窗口设置为不可见的动作带来了麻烦。在 X 服务器取消窗口的可见状态时，窗口管理器将收到通知 UnmapNotify，如果不加任何甄别，将导致 init_clients 刚刚管理的窗口又被销毁。显然，这不是我们希望的，因此结

构体 Client 中设计了变量 ignore_unmap 来忽略这个特殊的 UnmapNotify 事件。

至此，我们的迷你窗口管理管理器开发完成了，我们将其复制到 vita 系统，并使用如下命令运行：

```
root@vita:~# Xorg -retro -noreset &
root@vita:~# export DISPLAY=:0.0
root@vita:~# ./winman &
root@vita:~# ./hello_gtk &
```

如果没有问题，将看到一个类似图 7-13 所示的窗口。

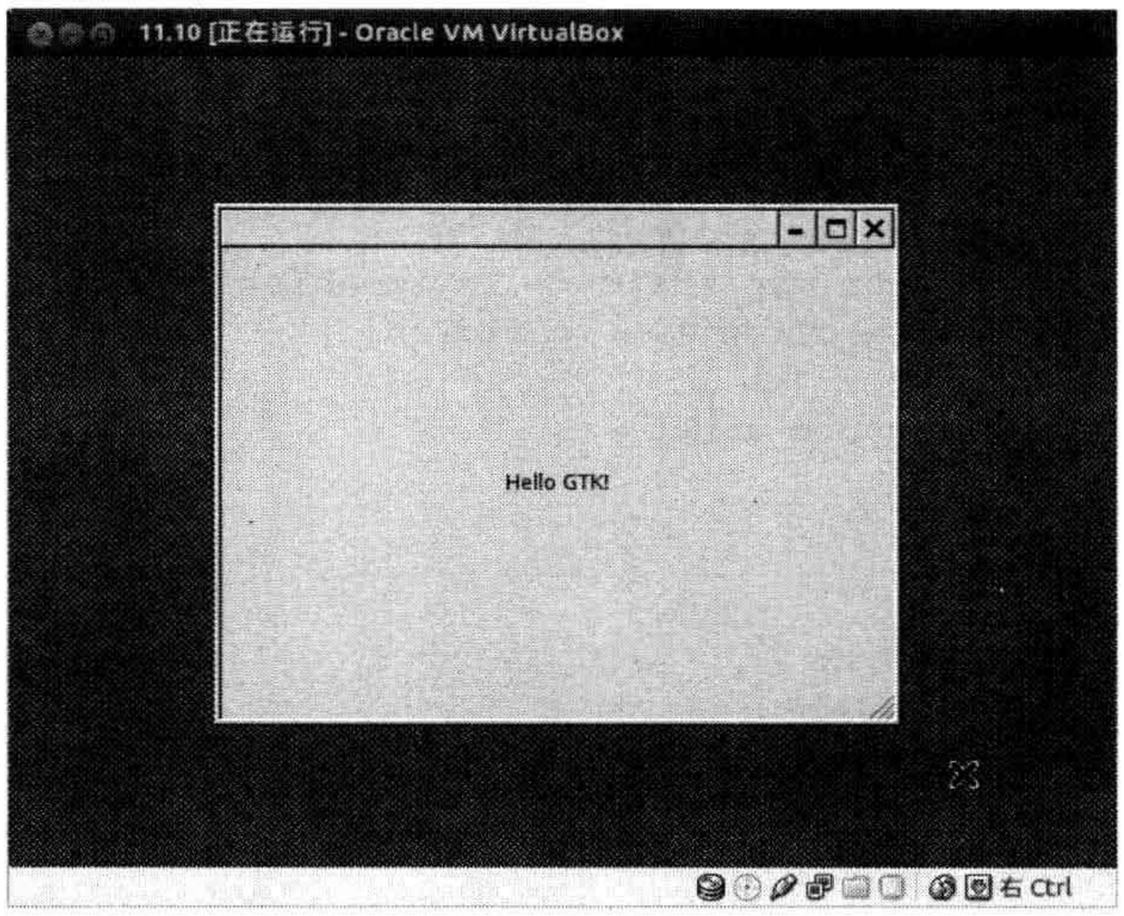


图 7-13 窗口管理器

根据图 7-13 中可见，hello_gtk 的窗口不再是一个光秃秃的裸窗口了，它被添加了各种装饰，看上去更有立体感了。而且我们可以拖住标题栏移动窗口，也可以改变窗口大小，可以关闭窗口，可以最大化窗口，但是一旦最小化窗口，就再也找不到它了。下一节，我们构建了任务条来解决这个问题。

7.2 任务条和桌面

从最初出现在桌面环境中发展到现在，任务条的风格也在不断地发生改变，但依然是桌面环境的重要组件之一，只不过表现形式并不一定是千篇一律。

典型的任务条从左至右包括“开始按钮”、“快速启动栏”、“任务项”以及“通知区域”。用户通过“开始按钮”可以启动应用程序；“快速启动栏”中放置用户常用的一些程序；每个启动的任务都有一个“任务项”；“通知区域”主要用来显示一些系统状态，比如显示当前的输入法、网络状态等。

除了任务条外，一般的桌面环境都有一个背景，并且在这个背景上面可以显示一些快捷

方式，可以显示一些很有个性的小插件。

本章中，我们实现了一个简单的任务条和一个桌面。不同的桌面环境，实现这些组件的逻辑不尽相同，有的是放在一个完整的程序中，有的是每个组件是一个单独的程序，我们采用后者。我们通过这两个程序向读者展示使用图形库（GTK）编程。相比于 Xlib，GTK 的编程理解起来要容易得多，而且 GTK 的官方文档写得也非常详尽，所以我们就不浪费篇幅讨论有关 GTK 的编程了，这里仅讨论其中与窗口管理器相关的部分。

7.2.1 标识任务条的身份

虽然任务条也是一个普通 X 应用，但是作为桌面环境中重要的一个组件，还是有一些特殊的地方。比如，在我们构建的桌面环境中，窗口管理器将其停靠在屏幕的最下方。但是任务条如何向 winman 亮明自己的任务身份呢？读者一定已经猜到了：属性。任务条自定义了属性 _CUSTOM_WM_WINDOW_TYPE_TASKBAR，在启动时，其将窗口的属性 _NET_WM_WINDOW_TYPE 设置为属性 _CUSTOM_WM_WINDOW_TYPE_TASKBAR，如下代码所示：

```
taskbar/src/main.c:

int main(int argc, char *argv[])
{
    ...
    XChangeProperty(taskbar->dpy, wid,
        taskbar->atoms[_NET_WM_WINDOW_TYPE],
        XA_ATOM, 32, PropModeReplace, (unsigned char *)
        &taskbar->atoms[_CUSTOM_WM_WINDOW_TYPE_TASKBAR], 1);
    ...
}
```

winman 一旦发现窗口的类型为 _CUSTOM_WM_WINDOW_TYPE_TASKBAR，则将其作为任务条管理，让我们回顾一下 winman 中给“窗口”落户的相关代码：

```
winman/src/main.c:

static Client* wm_new_client(WinMan *wm, Window win)
{
    ...
    status = XGetWindowProperty(wm->dpy, win,
                    wm->atoms[_NET_WM_WINDOW_TYPE],
                    0, 1, False, XA_ATOM, &type, &format, &n,
                    &extra, (unsigned char **)&value);
    ...
        else if (value[0] ==
                wm->atoms[_CUSTOM_WM_WINDOW_TYPE_TASKBAR])
            c = taskbar_client_new(wm, win);
    ...
}
```

如果 winman 发现窗口的类型是 _CUSTOM_WM_WINDOW_TYPE_TASKBAR，winman

将不再创建标准窗口的对象，而是创建一个任务条窗口对象，代码如下：

```
winman/src/taskbar_client.c:

Client* taskbar_client_new(WinMan *wm, Window win)
{
    ...
    c->x = 0;
    c->y = DisplayHeight(wm->dpy, wm->screen) - TASKBAR_HEIGHT;
    c->width = DisplayWidth(wm->dpy, wm->screen);
    c->height = TASKBAR_HEIGHT;
    ...
    XMoveResizeWindow(wm->dpy, win, c->x, c->y, c->width,
        c->height);
    ...
    c->reparent = &taskbar_client_reparent;
    ...
}
```

其中比较有趣的两个地方，我们需要特别关注：

1）winman 将任务条布局在特定的位置。其左起屏幕最左边，宽度为整个屏幕。高度为 TASKBAR_HEIGHT，这个宏在 winman 中定义，值为 30 像素。位于屏幕的底部。

2）任务条的构建窗口装饰的函数 taskbar_client_reparent，其实现如下：

```
winman/src/taskbar_client.c:

static void taskbar_client_reparent(Client *c)
{
}
```

函数体怎么是空的？没错，任务条不需要任何装饰。读者可能会问，那为什么定义这么一个空函数体？这个主要是出于面向对象实现上的考虑，当然，条条大路通罗马，聪明的读者也可以不需要定义这么个空函数，而是采用调用前判断函数指针是否为空，等等。

7.2.2 更新任务条上的任务项

前面我们看到，在 winman 中，每当为一个窗口“落户”时，winman 都将更新根窗口的属性 _NET_CLIENT_LIST_STACKING。因此，任务条利用的就是这个机制，监测根窗口属性的变化，从而跟踪系统中任务的变化，相关代码如下：

```
taskbar/src/main.c:

int main(int argc, char *argv[])
{
    ...
    gdk_window_set_events(root, GDK_PROPERTY_CHANGE_MASK);
    gdk_window_add_filter(root, root_window_event_filter, taskbar);
    ...
}
```

```
static GdkFilterReturn root_window_event_filter(GdkXEvent
    *xevent, GdkEvent *event, gpointer data)
{
    XPropertyEvent *prop;
    Taskbar *taskbar = data;
    ...
    prop = (XPropertyEvent*)xevent;
    ...
    } else if (prop->atom ==
           taskbar->atoms[_NET_CLIENT_LIST_STACKING]) {
        taskbar_setup_items(taskbar);
    } else if (prop->atom == taskbar->atoms[_NET_SHOWING_DESKTOP])
    ...
}
```

在任务条初始化时，其将选择根窗口事件掩码 PropertyChangeMask，并设置根窗口的属性变化事件的回调函数为 root_window_event_filter。如此，一旦根窗口的属性发生变化时，任务条都将洞悉。

每当根窗口的属性 _NET_CLIENT_LIST_STACKING 发生变化时，函数 taskbar_setup_items 就读取根窗口的该属性的值，获取目前系统中全部的窗口列表。然后遍历这个列表，更新任务栏。为了简单，该函数做了很多简化，比如只要窗口类型是 _NET_WM_WINDOW_TYPE_NORMAL，并且也没有判断窗口是否是其他窗口的临时窗口，任务条就为其在任务条上创建一个任务项。

作为桌面环境的核心组件之一，在桌面环境启动时，任务条是首先启动的核心组件之一。理论上，这个时候还没有应用启动，但是不排除系统运行过程中，任务条重新启动，谁也不能保证程序完全没有 bug。因此，无论如何，任务条还是有必要在启动时获取系统中正在运行的任务，并为它们在任务条上建立相应的任务项。这个过程请读者参考随书光盘中附带的源代码。

7.2.3 激活任务

任务条的另外一个主要任务就是将最小化的，或者将非活动的窗口激活为当前活动窗口。

EWMH 规范规定，如果一个 X 应用希望激活另外一个窗口，可以通过向根窗口发送消息 _NET_ACTIVE_WINDOW 来实现。因此，在我们的任务条中，当用户点击任务按钮时，在回调函数中将向根窗口发送 ClientMessage 事件，其中的消息类型为 _NET_ACTIVE_WINDOW，代码如下：

```
taskbar/src/taskbar_item.c:

static void taskbar_item_clicked_cb(TaskbarItem *item, ...)
{
    ewmh_send_net_active_window(item);
}
```

```
taskbar/src/ewmh.c:

void ewmh_send_net_active_window(TaskbarItem *item)
{
    XEvent e;

    memset(&e, 0, sizeof(e));
    e.xclient.type = ClientMessage;
    e.xclient.window = item->win;
    e.xclient.message_type =
        item->taskbar->atoms[_NET_ACTIVE_WINDOW];
    e.xclient.format = 32;

    XSendEvent(item->taskbar->dpy, GDK_ROOT_WINDOW(), False,
        SubstructureNotifyMask | SubstructureRedirectMask, &e);
}
```

事实上，任务条发给根窗口 ClientMessage 事件也被窗口管理器拦截了。读者可能有个疑问：窗口管理器会收到应用发给根窗口的类型为 ClientMessage 的事件吗？答案是肯定的。因为 EWMH 规定，ClientMessage 对应的事件掩码是 SubstructureNotifyMask 和 SubstructureRedirectMask，而窗口管理器恰恰选择了接收 SubstructureNotify 和 SubstructureRedirect。

winman 中处理事件 ClientMessage 的代码如下：

```
winman/src/main.c:

static void wm_handle_client_message(WinMan *wm,
   XClientMessageEvent *e)
{
    ...
    } else if (e->message_type == wm->atoms[_NET_ACTIVE_WINDOW]) {
        if (c = wm_find_client_by_window(wm, e->window)) {
            c->show(c);
            c->activate(c);
        }
    }
}
```

函数 wm_handle_client_message 检查消息的类型，如果是 _NET_ACTIVE_WINDOW，则调用窗口对象中函数指针 activate 指向的函数，将事件 XClientMessageEvent 中指定的窗口切换为当前活动窗口。具体的过程我们在 7.1.11 一节已经详细讨论了。

在调用 activate 前，函数 wm_handle_client_message 还调用了函数 show。目的是什么呢？原因是窗口可能上次被最小化了，因此首先需要请求 X 服务器显示这个窗口。

7.2.4 高亮显示当前活动任务

当某个任务成为当前活动任务时，任务条需要将对应的任务项特殊标识一下。那么任务条如何知道当前任务已经发生变化了呢？前面我们看到，在 winman 中，每当为将一个窗口设置为当前活动窗口时，winman 都将更新根窗口的属性 _NET_ACTIVE_WINDOW。看到这

里，读者一定明白了，任务条的处理过程与 7.2.2 节基本完全相同，相关代码如下：

```
taskbar/src/main.c:

static GdkFilterReturn root_window_event_filter(
    GdkXEvent *xevent, GdkEvent *event, gpointer data)
{
    XPropertyEvent *prop;
    Taskbar *taskbar = data;
    ...
    prop = (XPropertyEvent*)xevent;

    if (prop->atom == taskbar->atoms[_NET_ACTIVE_WINDOW]) {
        Window active_win = ewmh_get_net_active_window(taskbar);
    ...
}
```

函数 root_window_event_filter 用于检查根窗口发生变化的属性的值，如果是 _NET_ACTIVE_WINDOW，说明当前活动的窗口改变了，任务条从根窗口读取当前活动的窗口，然后将其在任务条上对应的项高亮显示。

7.2.5 显示桌面

当用户按下快速启动栏上的显示桌面按钮时，将把桌面显示到所有窗口的最前面。本章讨论到这里，我想读者应该已经大致可以猜出这个故事的脚本了：

1）任务条向根窗口发送类型为 ClientMessage 的事件，EWMH 规范规定这个事件中的消息类型为 _NET_SHOWING_DESKTOP。

2）winman 请求 X 服务器将将桌面这个组件显示到窗口栈的最上面。winman 中的实现与切换窗口基本完全相同。

下面就是任务条中当用户点击显示桌面按钮后发送消息的相关代码：

```
Taskbar/src/ewmh.c:

void ewmh_send_net_showing_destkop(Taskbar *taskbar)
{
    XEvent e;

    memset(&e, 0, sizeof(e));
    e.xclient.type = ClientMessage;
    e.xclient.message_type =
                taskbar->atoms[_NET_SHOWING_DESKTOP];
    e.xclient.format = 32;
    e.xclient.data.l[0] = 1;

    XSendEvent(taskbar->dpy, GDK_ROOT_WINDOW(), False,
        SubstructureNotifyMask | SubstructureRedirectMask, &e);
}
```

7.2.6 桌面

相比于任务条，这个示例的桌面程序要简单很多。而且，经过了前面任务条的讨论，我想读者应该不需要笔者再过多的啰唆了。同普通应用对比，其比较特殊的地方之一就是，要向窗口管理器亮明自己的身份，代码如下所示：

desktop/src/main.c:

```
int main(int argc, char *argv[])
{
    ...
    gtk_window_set_type_hint(GTK_WINDOW(win),
       GDK_WINDOW_TYPE_HINT_DESKTOP);
    ...
}
```

桌面程序使用标准的 EWMH 规范规定的属性 _NET_WM_WINDOW_TYPE_DESKTOP 标识该程序是一个桌面程序。GTK 中的函数 gtk_window_set_type_hint 就是对 Xlib 的函数 XChangeProperty 的更高层的封装，我们直接使用即可。

winman 将为桌面程序创建桌面窗口对象，并将其整个铺满在桌面背景上。同样，桌面窗口对象也不需要装饰，因此其函数 desktop_client_reparent 也是个空函数。其他细节，请读者参考随书光盘中附带的源代码。

至此，一个基本的桌面环境就已经搭建完毕了，读者将它们安装到 vita 系统，然后使用如下命令即可启动完整的桌面环境：

```
root@vita:~# Xorg -retro -noreset &
root@vita:~# export DISPLAY=:0.0
root@vita:~# ./winman &
root@vita:~# ./desktop &
root@vita:~# ./taskbar &
```

注意，桌面程序上的快捷方式“Hello World”的回调函数将到目录 /usr/bin 下寻找程序 hello_gtk，所以请将这个程序复制到目录 /usr/bin 下。另外，也请确保程序 taskbar、desktop 使用的 css 主题描述安装在正确的目录下。如果遇到麻烦，请读者参考随书光盘中附带的源代码。

如果一切正常，将看到一个类似图 7-14 所示的完整桌面环境。

这是一个经典的 PC 上的桌面环境。桌面下方是个任务条，其最左侧是个“开始”按钮。然后紧接着是快速启动栏，其中包含一个“显示桌面”的按钮。接下来当然就是各个任务项了。在图 7-14 所示的例子中，我们运行了两个 hello_gtk 程序，所以在任务条上有两个任务项。同时有一个程序专门负责桌面的背景，当然其上面还可以添加各种程序的快捷方式，比如这里添加了程序 hello_gtk 的快捷方式。

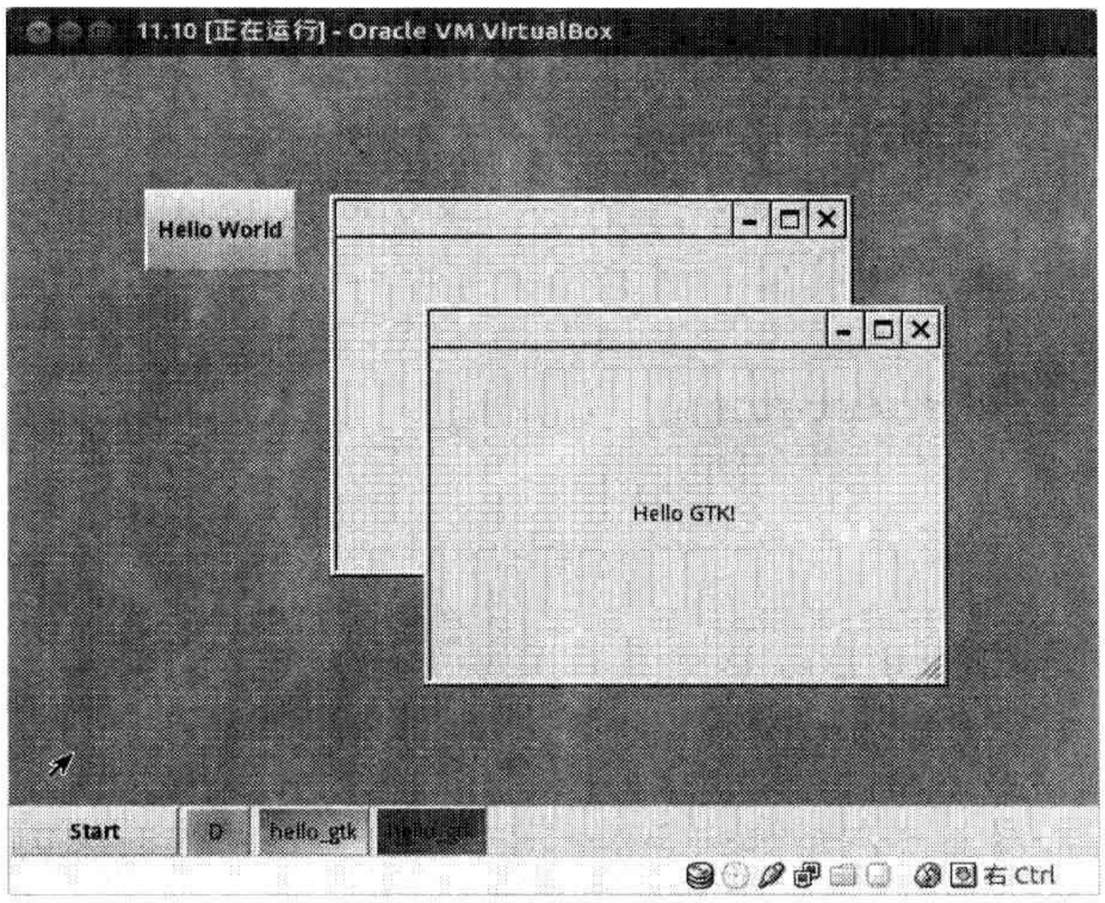


图 7-14 完整的桌面环境

什么？没有 Dashboard（Mac OS X 的很有特色的一个桌面组件）？是的，这是一个很简陋的桌面环境。但是通过这个简陋的桌面，我们已经清楚了桌面环境的基本组成及运行原理，接下来，你可以按照意愿随意改造，甚至可以添加一个新的桌面组件，但是记得告诉窗口管理器将其放在了哪一个特殊的位置。

虽然如今的桌面环境越来越个性化，就如同那个“开始”按钮都可能消失一样，但是事实上，这些都是表面现象。如果你对众多操作系统比较了解，你就会知道，它们本质上完全相同，只是表象不同而已，就看你是否足够艺高且敢于创造了。

第8章

Linux 图形原理探讨

在第 6 章和第 7 章中，我们揭示了在 Linux 操作系统中，图形系统及桌面环境的构成。这一章，我们进一步深入，尝试探讨 Linux 的图形原理。

本质上，谈及图形原理必会涉及渲染和显示两部分。但是显示过程比较简单和直接，而渲染过程要复杂得多，更重要的是，渲染牵扯到操作系统内部的组件更多，因此，本章我们主要讨论渲染过程。我们不想只浮于理论，结合具体的 GPU 进行讨论更有助于深度理解计算机的图形原理。相比于 NV 及 ATI 的 GPU，我们选择相对更开放一些的 Intel 的 GPU 进行讨论。Intel 的 GPU 也在不断的演进，本书写作时主要针对的是用在 Sandy Bridge 和 Ivy Bridge 架构上的 Intel HD Graphics。

显存是图形渲染的基础，也是理解图形原理的基础，因此，本章我们从讨论显存开始。或许读者会说，显存有什么好讨论的，不就是一块存储区吗？早已是陈词滥调。但是事实并非如此，通过显存的讨论，我们会注意到 CPU 和 GPU 融合的脚步，会看到它们是如何的和谐共享物理内存的。或许，已经有 GPU 和 CPU 完美地进行统一寻址了。

然后，我们分别讨论 2D 和 3D 的渲染过程。在其间，我们将看到到底何谓硬件加速，我们也会从更深的层次去展示 3D 渲染过程中所谓的 Pipeline。以往，很多教材都会为了辅助 OpenGL 的应用开发，多少从理论上谈及一点 Pipeline，而在这一章中，我们从操作系统角度和 Pipeline 进行一次亲密接触。

最后，我们讨论了很多读者认为神秘而陌生的 Wayland。其实，Wayland 既不神秘也不陌生，它是在 DRI 和复合扩展发展的背景下产生的，基于 DRI 和复合扩展演进的成果。从某个角度，Wayland 更像是去除了基于网络的服务器 / 客户端的 X 和复合管理器的一次整合。

8.1 渲染和显示

计算机将图形显示到显示设备上的过程，可以划分为两个阶段：第一阶段是渲染（render）过程，第二阶段是显示（display）过程，如图 8-1 所示。

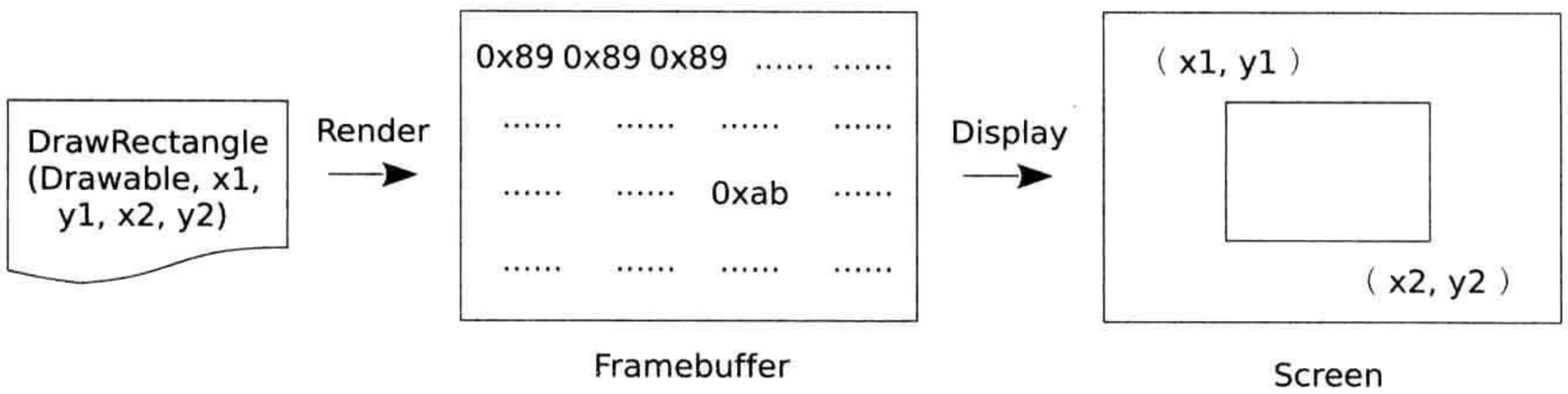


图 8-1 图形渲染显示过程示意图

8.1.1 渲染

所谓渲染就是将使用数学模型描述的图形，如“DrawRectangle(x1, y1, x2, y2)”，转化为像素阵列，或者叫像素数组。像素数组中的每个元素是一个颜色值或者颜色索引，对应图像的一个像素。对于 X 窗口系统，数组中的元素是一个颜色索引，具体的颜色根据这个索引从颜色映射（Colormap）中查询得来。

渲染通常又被分为两种：一种渲染过程是由 CPU 完成的，通常称为软件渲染，另外一种是由 GPU 完成的，通常称为硬件渲染，也就是我们常常提到的硬件加速。

谈及渲染，不得不提的另外一个关键概念就是帧缓冲（Framebuffer）。从字面意义上讲，frame 表示屏幕上的某个时刻对应的一帧图像，buffer 就是一段存储区了，因此，在狭义上，合起来的这个词就是指存储一帧屏幕图像像素数据的存储区。

但是从广义上，帧缓冲则是多个缓冲区的统称。比如在 OpenGL 中，帧缓冲包括用于输出的颜色缓冲（Color Buffer），以及辅助用来创建颜色缓冲的深度缓冲（Depth Buffer）和模板缓冲（Stencil Buffer）等。即使在 2D 环境中，帧缓冲这个概念也不仅指屏幕上的一帧图像，还包含用于存储命令的缓冲（Command Buffer）等。每一个应用都有自己的一套帧缓冲。

在 OpenGL 环境中，为了避免渲染和显示过程交叉导致冲突，从而出现如撕裂（tearing）以及闪烁（flicker）等现象，颜色缓冲又被划分为前缓冲（front buffer）和后缓冲（back buffer）。如果为了支持立体效果，则前缓冲和后缓冲又分别划分为左和右各两个缓冲区，我们不讨论这种情况。前缓冲和后缓冲中的内容都是像素阵列，每个像素或者是一个颜色值，或者是一个颜色索引。只不过前缓冲用于显示，后缓冲用于渲染。

2D 可以看作 3D 的一个特例，因此，我们将 2D 程序中用于输出的缓冲区称为前缓冲。为了避免歧义，在容易引起混淆的地方我们尽量不使用这个多义的帧缓冲一词。

8.1.2 显示

一般而言，显示设备也使用像素来衡量，比如屏幕的分辨率为 1366×768，那么其可以显示 1049 088 个像素，一个像素对应屏幕上的一个点，图像就是通过这些点显示出来的。通常，图像中一个像素对应屏幕上的一个像素，那么将图像显示到屏幕的过程就是逐个读取帧缓冲中存储的图像的像素，根据其所代表的颜色值，控制显示器上对应的点显示相应颜色的

过程。

通常，显示过程基本上要经过如下几个组件：显示控制器（CRTC）、编码器（Encoder）、发射器（Transmitter）、连接器（Connector），最后显示在显示器上。

（1）显示控制器

显示控制器负责读取帧缓冲中的数据。对于 X 来说，帧缓冲中存储的是颜色的索引，显示控制器读取索引值后，还需要根据索引值从颜色映射中查询具体的颜色值。显示控制器也负责产生同步信号，典型的如水平同步信号（HSYNC）和垂直同步信号（VSYNC）。水平同步信号目的是通知显示设备开始显示新的一行，垂直同步信号通知显示设备开始显示新的一帧。所谓同步，以垂直同步信号为例，我们可以这样来通俗地理解它：显示控制器开始扫描新的一帧数据了，因此它通过这个信号告诉显示器开始显示，跟上我，不要掉队，这就是同步的意思。以 CRT 显示器为例，这两个信号控制着电子枪的移动，每显示完一行，电子枪都会回溯到下一行的开始，等待下一个水平同步信号的到来。每显示完一帧，电子枪都会回溯到屏幕的左上角，等待一下垂直同步信号的到来。

（2）编码器

对于帧缓冲中每个像素，可能使用 8 位、16 位、32 位甚至更多的位来表示颜色值，但是对于具体的接口来说，却远没有这么多的数据线供使用，而且不同的接口有不同的格式规定。比如对于 VGA 接口来说，总共只有三根数据线，每个颜色通道占用一根数据线；对于 LVDS 来说，数据是串行传输的。因此，需要将 CRTC 读取的数据编码为适合具体物理接口的编码格式，这就是编码器的作用。

（3）发射器

发射器将经过编码的数据转变为物理信号。读者可以将其想象成：发射器将 1 转化为高电平，将 0 转化为低电平。当然，这只是一个形象的说法。

（4）连接器

连接器有时也被称为端口（Port），比如 VGA、LVDS 等。它们直接连接着显示设备，负责将发射器发出的信号传递给显示设备。

8.2 显存

Intel 的 GPU 集成到芯片组中，一般没有专用显存，通常是由 BIOS 从系统物理内存中分配一块空间给 GPU 作专用显存。一般而言，BIOS 会有个默认的分配规则，有的 BIOS 也会为用户留有接口，用户可以通过 BIOS 设置显存的大小。如对于具有 1GB 物理内存的系统来说，可以划分 256MB 内存给 GPU 用作显存。

但是这种静态的分配方式带来的问题之一就是如何平衡系统与显示占用的内存，究竟分配多少内存给 GPU 才能在系统常规使用和运行图形计算密集的应用（如 3D 应用）之间达到最优。如果分配给 GPU 的显存少了，那么在进行图形处理时性能必然会降低。而单纯提

高分配给 GPU 的显存，也可能会造成系统的整体性能降低。而且，过多的分配内存给显存，那么当不运行 3D 应用时，就是一种内存浪费。毕竟，用户的使用模式不会是一成不变的，比如对于一个程序员来说，在编程之余也可能会玩一些游戏。但是我们显然不能期望用户根据具体运行应用的情况，每次都进入 BIOS 修改内存分配给显存的大小。

为了最优利用内存，一种方式就是不再从内存中为 GPU 分配固定的显存，而是当 GPU 需要时，直接从系统内存中分配，不使用时就归还给系统使用。但是 CPU 和 GPU 毕竟是两个完全独立的处理器，虽然现在 CPU 和 GPU 正在走融合之路，但是它们依然有自己的地址空间。显然，我们不能允许 CPU 和 GPU 彼此独立地去使用物理内存，这样必然会导致冲突，也正是因为这个原因，才有了我们前面提到的 BIOS 会从物理内存中划分一块区域给 GPU，这样 CPU 和 GPU 才能井水不犯河水，分别使用属于自己的存储区域。

8.2.1 动态显存技术

为了解决这个矛盾，Intel 的开发者们开发了动态显存技术（Dynamic Video Memory Technology），相比于以前在内存中为 GPU 开辟专用显存，使用动态显存技术后，显存和系统可以按需动态共享整个主存。

动态显存中关键的是 GART（graphics address remapping table），也被称为 GTT（graphics translation table），它是 GPU 直接访问系统内存的关键。事实上，这是 CPU 和 GPU 的融合过程中的一个产物，最终，CPU 和 GPU 有可能完全实现统一的寻址。

GTT 就是一个表格，或者说就是一个数组，表格中的每一个表项占用 4 字节，或者指向物理内存中的一个页面，或者设置为无效。整个 GTT 所能寻址的范围就代表了 GPU 的逻辑寻址空间，如 512KB 大小的 GTT 可以寻址 512MB 的显存空间（512K/4 * 4KB = 512MB），如图 8-2 所示。

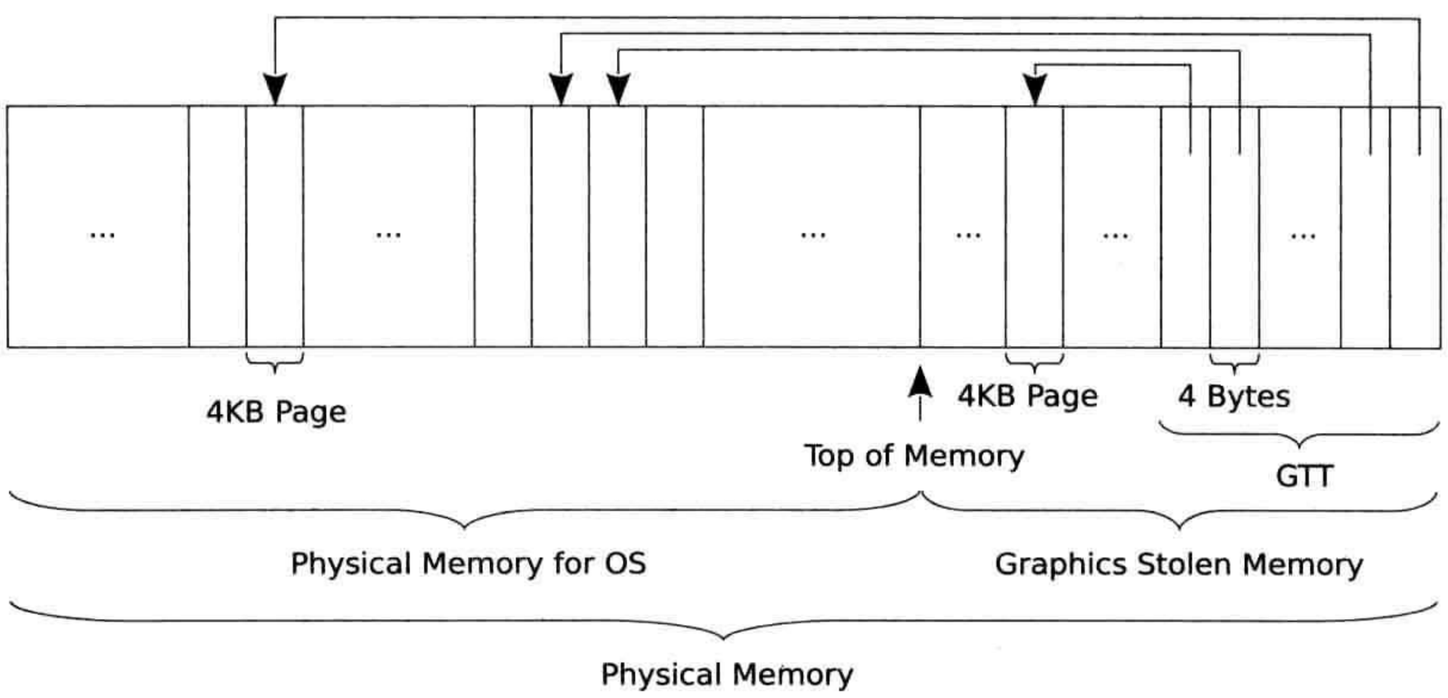


图 8-2 显存映射

这是一种动态按需从内存中分配显存的方式，GTT 中的所有表项不必全部都映射到实

际的物理内存，完全可以按需映射。而且当 GTT 中的某个表项指向的内存不再被 GPU 使用时，可以收回为系统所用。通过这种动态按需分配的方式，达到系统和 GPU 最优分享内存。内核中的 DRM 模块设计了特殊的互斥机制，保证 CPU 和 GPU 独立寻址物理内存时不会发生冲突。

我们注意到，GPU 是通过 GTT 访问内存的（内存中用作显存的部分），所以 GPU 首先要访问 GTT，但是，GTT 也是在内存中。显然，这又是一个先有鸡还是先有蛋的问题。因此，需要另外一个协调人出现，这个协调人就是 BIOS。在 BIOS 中，仍然需要在物理内存中划分出一块对操作系统不可见、专用于显存的存储区域，这个区域通常也称为 Graphics Stolen Memory。但是相比于以前动辄分配几百兆的专用显存给 GPU，这个区域要小多了，一般几 MB 就足矣，如我们前面讨论的寻址 512MB 的显存，只需要一个 512KB 的 GTT 表。

BIOS 负责在 Graphics Stolen Memory 中建立 GTT 表格，初始化 GTT 表格的表项，更重要的是，BIOS 负责将 GTT 的相关信息，如 GTT 的基址，写入到 GPU 的 PCI 的配置寄存器（PCI Configuration Registers），这样，GPU 可以直接找到 GTT 了。BIOS 中初始化 GTT 的代码大致如下：

```
01 InitGfxGtt(...) {
02     uint32_t gfxMemAddr, gttMemStart;
03     volatile uint32_t *pGttEntry;
04
05     PCI_WRITE(busno, deviceid, function number,
06                 PCI_REG_GTT, “GTT Base address” );
07
08     gfxMemAddr = “Graphics Stolen Memory Address” ;
09     gttMemStart = “GTT Base Address” ;
10     pGttEntry = “GTT Base Address” ;
11
12     while (gfxMemAddr < gttMemStart) {
13         *pGttEntry = (gfxMemAddr | 0x00000001);//addr + valid bit
14         pGttEntry++;    // next PTE
15         gfxMemAddr += (4 * KB); // next page
16     }
17
18     while (pGttEntry < pGttEnd) {
19         *pGttEntry = 0; // mark entry invalid
20         pGttEntry++;     // next PTE
21     }
22 }
```

在上面代码中，变量 gfxMemAddr 代表 Graphics Stolen Memory 的起始地址，gttMemStart 代表 GTT 的起始地址，指针 pGttEntry 指向 GTT 的表项。代码第 8~10 行初始化了这几个变量，在初始化时，pGttEntry 指向 GTT 表的开始，为后面填充 GTT 表作准备。

BIOS 从物理内存的最顶端分配一块区域作为 Graphics Stolen Memory，然后在这块区域中分配一块区域用作 GTT，并将 GTT 所在的地址写入 GPU 的 PCI 的配置寄存器，见代码第 5~6 行。操作系统启动后将从这个寄存器中读取 GTT 表的地址，其中 PCI_REG_GTT 表示

GPU 中用作保存 GTT 地址的 PCI 配置寄存器。

在初始化时，GTT 只需要映射 Graphics Stolen Memory 区域即可，当然 GTT 占用的空间就无需映射了。代码第 12~16 行就是映射 GSM 中除 GTT 以外的显存的，即 gfxMemAddr 到 gttMemStart 之间的部分。

显存的其余部分需要时动态按需分配，所以代码第 18~21 行的 while 循环就是将 GTT 中的表项设置为无效，亦即尚未分配显存。

在操作系统启动后，显存的分配和回收就由操作系统负责了，因此操作系统需要访问 GTT。但是，GTT 存储在操作系统不可见的 Graphics Stolen Memory 中，那么操作系统如何找到 GTT 呢？这就是 BIOS 将 GTT 的地址设置到 GPU 的 PCI 配置寄存器中的原因。在操作系统启动后，将从 GPU 的 PCI 配置寄存器中获取如 GTT 的基址等信息，代码如下：

```
linux-3.7.4/drivers/char/agp/intel-gtt.c:

static const struct intel_gtt_driver i915_gtt_driver = {
    .gen = 3,
    .has_pgtbl_enable = 1,
    .setup = i9xx_setup,
    ...
};

static const struct intel_gtt_driver sandybridge_gtt_driver = {
    ...
};

static int i9xx_setup(void)
{
    ...
    if (INTEL_GTT_GEN == 3) {
        u32 gtt_addr;

        pci_read_config_dword(intel_private.pcidev,
                              I915_PTEADDR, &gtt_addr);
        intel_private.gtt_bus_addr = gtt_addr;
    } else {
    ...
}
```

在内核中，对于 Intel 不同系列的 GPU，都有相应的 GTT 驱动，比如分别有针对 i8xx、i915、sandybridge 等型号的 GTT 驱动模块。

以 i915 系列的 GPU 为例，我们在其 GTT 驱动的函数 i9xx_setup 中可见，其使用 pci_read_config_dword 从 GPU 的 PCI 的配置寄存器 I915_PTEADDR 中读取 GTT 的基址等相关信息，然后将 i9xx_setup 读取的 GTT 的地址保存到 gtt_bus_addr，这个变量就是用来保存 GPU 的 GTT 的地址的，后面我们在讨论显存绑定时会再次见到这个变量。

当然在 GTT 的驱动模块中，也包含操作 GTT 表的函数，如更新 GTT 表项的函数 write_entry，后面在讨论 Buffer Object 绑定到 GTT 时，我们会看到这个函数。

8.2.2 Buffer Object

与 CPU 相比，GPU 中包含大量的重复的计算单元，非常适合如像素、光影处理、3D 坐标变换等大量同类型数据的密集运算。因此，很多程序为了能够使用 GPU 的加速功能，都试图和 GPU 直接打交道。因此，系统中可能有多个组件或者程序同时使用 GPU，如 Mesa 中的 3D 驱动、X 的 2D 驱动以及一些直接通过帧缓冲驱动直接操作帧缓冲的应用等。但是多个程序并发访问 GPU，一旦逻辑控制不好，势必导致系统工作极不稳定，严重者甚至使 GPU 陷入一个混乱的状态。

而且，如果每个希望使用 GPU 加速的组件或程序都需要在自身的代码中加入操作 GPU 的代码，也使开发过程变得非常复杂。

于是，为了解决这一乱象，开发者们在内核中设计了 DRM 模块，所有访问 GPU 的操作都通过 DRM 统一进行，由 DRM 来统一协调对 GPU 的访问，如图 8-3 所示。

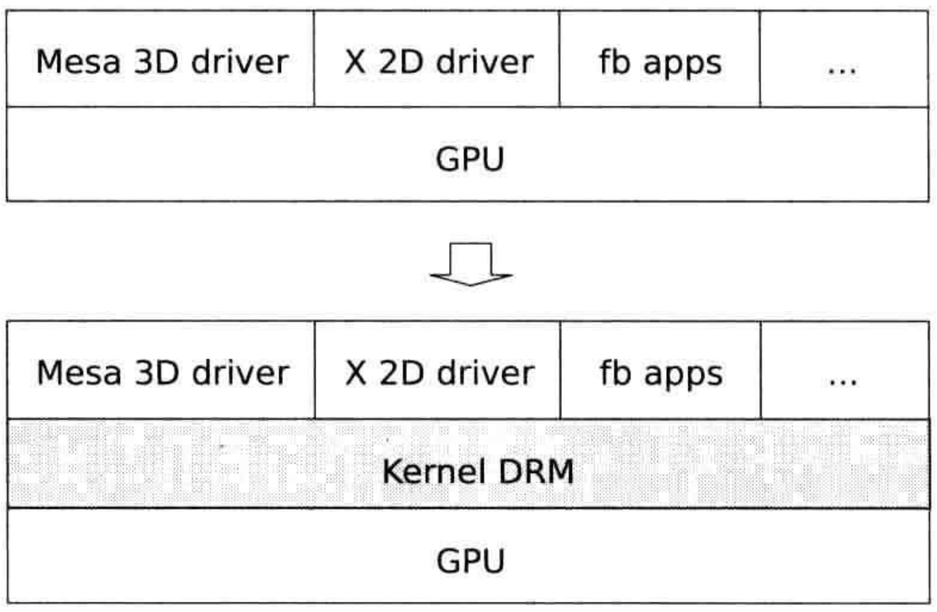


图 8-3 内核中的 DRM 模块

DRM 的核心是显存的管理，当前内核的 DRM 模块中包含两个显存管理机制：GEM 和 TTM。TTM 先于 GEM 开发，但是 Intel 的工程师认为 TTM 比较复杂，所以后来设计了 GEM 来替代 TTM。目前内核中的 ATI 和 NVIDA 的 GPU 驱动仍然使用 TTM，所以 GEM 和 TTM 还是共存的，但是 GEM 占据主导地位。

GEM 抽象了一个数据结构 Buffer Object，顾名思义，就是一块缓冲区，但是比较特别，是 GPU 使用的一块缓冲区，也就是一块显存。比如一个颜色缓冲的像素阵列保存在一个 Buffer Object，绘制命令以及绘制所需数据也分别保存在各自的 Buffer Object，等等。笔者实在找不到一个准确的中文词汇来代表 Buffer Object，所以只好使用这个英文名称。开发者习惯上也将 Buffer Object 简称为 BO，后续为了行文方便，我们有时也使用这个简称，其定义如下：

```
linux-3.7.4/include/drm/drmP.h:

struct drm_gem_object {
    ...
    struct file *filp;
```

```
    ...
    int name;
    ...
};
```

其中两个关键的字段是 filp 和 name。

对于一个 BO 来说，可能会有多个组件或者程序需要访问它。GEM 使用 Linux 的共享内存机制实现这一需求，字段 filp 指向的就是 BO 对应的共享内存，代码如下：

```
linux-3.7.4/drivers/gpu/drm/drm_gem.c:

int drm_gem_object_init(...)
{
    ...
    obj->filp = shmem_file_setup("drm mm object", size, ...);
    ...
}
```

既然多个组件需要访问 BO，GEM 为每个 BO 都分配了一个名字。当然这个名字不是一个简单的字符，它是一个全局唯一的 ID，各个组件使用这个名字来访问 BO。

BO 可以占用一个页面，也可以占用多个页面。但是，通常 BO 都是占用整数个页面，即 BO 的大小一般是 4KB 的整数倍。在 i915 的 BO 的结构体定义中，数据项 pages 指向的就是 BO 占用的页面的链表，这里并不是使用的简单的链表，结构体 sg_table 使用了散列技术。具体代码如下：

```
linux-3.7.4/drivers/gpu/drm/i915/i915_drv.h:

struct drm_i915_gem_object {
    ...
    struct sg_table *pages;
    ...
};
```

为了可以被 GPU 访问，BO 使用的内存页面还要映射到 GTT。这个映射过程也比较直接，就是将 BO 所在的页面填入到 GTT 的表项中。以 i915 为例，下面这个函数就是获取 BO 占据的页面：

```
linux-3.7.4/drivers/gpu/drm/i915/i915_gem.c:

static int
i915_gem_object_get_pages_gtt(struct drm_i915_gem_object *obj)
{
    ...
    mapping = obj->base.filp->f_path.dentry->d_inode->i_mapping;
    ...
    for_each_sg(st->sgl, sg, page_count, i) {
        page = shmem_read_mapping_page_gfp(mapping, i, gfp);
        ...
        sg_set_page(sg, page, PAGE_SIZE, 0);
    }
```

```
    obj->pages = st;
    ...
}
```

注意上面代码中使用黑体标识的 filp，它指向了 BO 对应的共享内存区。显然，获取 BO 的页面实际就是获取这块共享内存的页面，代码中函数 shmem_read_mapping_page_gfp 就是做这件事的。当然 BO 可能对应多个页面，所以这里是一个循环，并将每个获取的页面放到散列表中，最后使 BO 中的指针 pages 指向这个页面散列表。

将 BO 的对应页表写入到 GTT 的表项中的代码如下：

```
linux-3.7.4/drivers/gpu/drm/i915/i915_gem_gtt.c:

void i915_gem_gtt_bind_object(struct drm_i915_gem_object *obj,...)
{
    ...
    intel_gtt_insert_sg_entries(obj->pages, ...);
    ...
}
```

函数 intel_gtt_insert_sg_entries 在内核的 Intel 的 GTT 驱动模块中，其实现代码如下：

```
linux-3.7.4/drivers/char/agp/intel-gtt.c:

void intel_gtt_insert_sg_entries(struct sg_table *st, ...)
{
    ...
    for_each_sg(st->sgl, sg, st->nents, i) {
        len = sg_dma_len(sg) >> PAGE_SHIFT;
        for (m = 0; m < len; m++) {
            dma_addr_t addr = sg_dma_address(sg) + (m << PAGE_SHIFT);
            intel_private.driver->write_entry(addr, j, flags);
            j++;
        }
    }
    ...
}
```

函数 intel_gtt_insert_sg_entries 遍历 BO 对应页面的散列表，依次调用 GTT 驱动中的函数 write_entry 将页面的地址写入到 GTT 的表项中。GPU 当然不能理解 CPU 使用的虚拟地址了，所以函数 sg_dma_address 返回的是页面的物理地址。i915 系列 GPU 的 GTT 驱动的 write_entry 函数如下：

```
linux-3.7.4/drivers/char/agp/intel-gtt.c:

static const struct intel_gtt_driver i915_gtt_driver = {
    ...
    .write_entry = i830_write_entry,
    ...
};

static void i830_write_entry(dma_addr_t addr,unsigned int entry,...)
```

```
{
    ...
    writel(addr | pte_flags, intel_private.gtt + entry);
}
```

函数 i830_write_entry 逻辑非常简单，尤其是对于了解驱动的读者而言更是如此，其与我们向某个内存地址处赋值无本质区别。这里，addr 就是页面的地址，intel_private.gtt 是 GTT 的基址，entry 是 GTT 中具体的表项。

读者可能有个疑问：GTT 不是在 BIOS 划分给 GPU 专用的 Graphics Stolen Memory 中吗？那么 CPU 怎么可以寻址 GTT，更新 GTT 的表项呢？内核中的 GTT 驱动模块已经考虑到了这点，在 GTT 模块初始化时，其使用 ioremap 将 GTT 所在地址映射到了 CPU 的地址空间，代码如下所示：

```
linux-3.7.4/drivers/char/agp/intel-gtt.c:
static int intel_gtt_init(void)
{
    ...
    if (INTEL_GTT_GEN < 6 && INTEL_GTT_GEN > 2)
        intel_private.gtt = ioremap_wc(
                        intel_private.gtt_bus_addr, gtt_map_size);
    if (intel_private.gtt == NULL)
        intel_private.gtt = ioremap(intel_private.gtt_bus_addr,
                                    gtt_map_size);
    ...
}
```

看到变量 gtt_bus_addr 是不是很熟悉？没错，在前面讨论 i915 的 GTT 驱动中的函数 i9xx_setup 时我们看到，i9xx_setup 从 GPU 的 PCI 配置寄存器读取的 GTT 的基址就记录在这个变量中。

综上，我们看到，BO 本质上就是一块共享内存，对于 CPU 来说 BO 与其他内存没有任何差别，但是 BO 又是特别的，它被映射进了 GTT，所以它既可以被 CPU 寻址，也可以被 GPU 寻址，如图 8-4 所示。

为了方便程序使用内核的 DRM 模块，开发者们开发了库 libdrm。在库 libdrm 中 BO 的定义如下：

```
libdrm-2.4.35/intel/intel_bufmgr.h:
struct _drm_intel_bo {
    ...
    unsigned long offset;
    ...
    void *virtual;
    ...
};
```

其中两个重要的数据项是 offset 和 virtual。

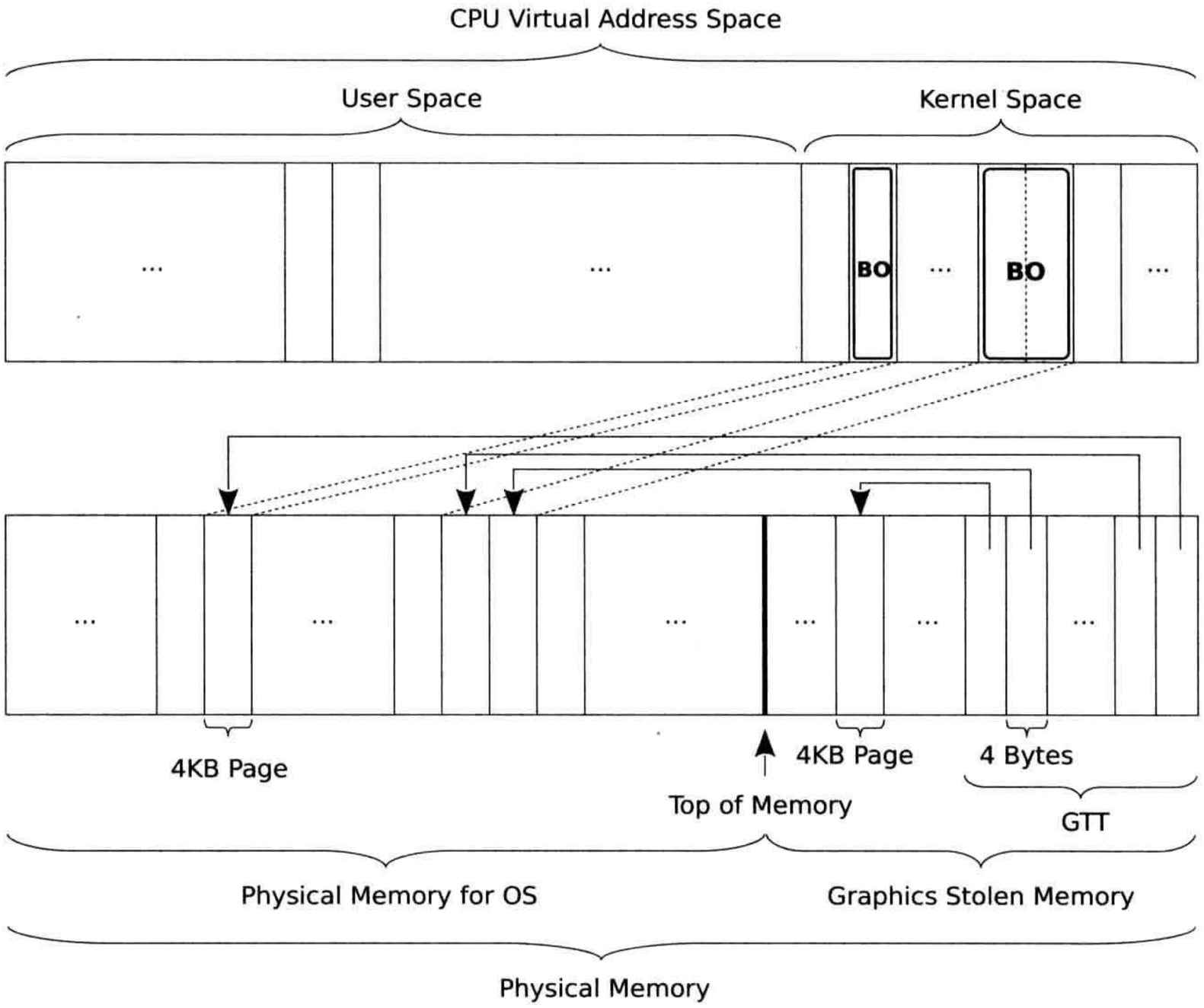


图 8-4　Buffer Object

事实上，BO 只是 DRM 抽象的在内核空间代表一块显存的一个数据结构。那么 GPU 是怎么找到 BO 的呢？如同 CPU 使用地址寻址一个内存单元一样，GPU 也使用地址寻址。GPU 根本不关心什么 BO，它只认显存的地址。因此，每一个 BO 在显存的地址空间中，都有一个唯一的地址，GPU 通过这个地址寻址，这就是 offset 的意义。offset 是 BO 在显存地址空间中的虚拟地址，显存使用线性地址寻址，任何一个显存地址都是从起始地址的偏移，这就是 offset 命名的由来。offset 通过 GTT 即可映射到实际的物理地址。当我们向 GPU 发出命令访问某个 BO 时，就使用 BO 的成员 offset。

有时需要将 BO 映射到用户空间，其中数据项 virtual 就是记录映射的基址。

前面，我们讨论了 BO 的本质。下面我们从使用的角度看一看 CPU 与 GPU 又是如何使用 BO 的。BO 是显存的基本单元，所以从保存像素阵列的帧缓冲，到 CPU 下达给 GPU 的指令和数据，全部使用 BO 承载。下面，我们分别从软件渲染和硬件渲染两个角度看看 BO 的使用。

（1）软件渲染

当 GPU 不支持某些绘制操作时，代表帧缓冲的 BO 将被映射到用户空间，用户程序直接在 BO 上使用 CPU 进行软件绘制。从这里我们也可以看出，DRM 巧妙的设计使得 BO 非常方便地在显存和系统内存之间进行角色切换。

（2）硬件渲染

当 GPU 支持绘制操作时，用户程序则将命令和数据等复制到保存命令和数据的 BO，然后 GPU 从这些 BO 读取命令和数据，按照 BO 中的指令和数据进行渲染。

库 libdrm 中提供了函数 drm_intel_bo_subdata 和 drm_intel_bo_get_subdata，在程序中一般使用这两个函数将用户空间的命令和数据复制到内核空间的 BO 读者也会见到 dri_bo_subdata 和 dri_bo_get_subdata。对于 Intel 的驱动来说，后面两个函数分别是前面两个函数的别名而已。后面讨论具体渲染过程时，我们会经常看到这几个函数。

8.3 2D 渲染

这一节，我们结合 X 窗口系统，讨论 2D 程序的渲染过程。我们可以形象地将 2D 渲染过程比喻为绘画，其中有两个关键的地方：一个是画布，另外一个是画笔。

X 服务器启动后，将加载 GPU 的 2D 驱动，2D 驱动将请求内核中的 DRM 模块创建帧缓冲，这个帧缓冲就相当于画布。然后 X 服务器按照绘画需要，从画笔盒子中挑选合适的画笔进行绘画。

X 的画笔保存在结构体 GCOps 中，其中包含了基本的绘制操作，如绘制矩形的 PolyRectangle，绘制圆弧的 PolyArc，绘制实心多边形的 FillPolygon，等等。代码如下：

```
xorg-server-1.12.2/include/gcstruct.h:
typedef struct _GCOps {
    ...
    void (*PolyRectangle) (DrawablePtr /*pDrawable */ , ...);

    void (*PolyArc) (DrawablePtr /*pDrawable */ , ...);

    void (*FillPolygon) (DrawablePtr /*pDrawable */ , ...);

    void (*PolyFillRect) (DrawablePtr /*pDrawable */ , ...);
    ...
} GCOps;
```

最初，这些绘制操作均由 CPU 负责完成，也就是我们通常所说的软件渲染。X 中的 fb 层就是软件渲染的实现，代码如下：

```
xorg-server-1.12.2/fb/fbgc.c:

const GCOps fbGCOps = {
    fbFillSpans,
    ...
    fbPolySegment,
    fbPolyRectangle,
    fbPolyArc,
    miFillPolygon,
    ...
};
```

但是随着 GPU 的不断发展，其计算能力越来越强。于是 X 的开发者们不断改进 X 的渲染部分，希望能充分利用 GPU 擅长的图形操作以大幅提高计算机的图形能力，而又可以解放 CPU，使其专心于控制逻辑。也就是说，X 的开发者们希望画笔盒子中的画笔更多地来自 GPU。

当然，任何事物都不是一蹴而就的，GPU 的渲染能力也是螺旋演进的，对于 GPU 尚未实现的或者相比来说 CPU 更适合的渲染操作还是需要 CPU 来完成，因此，X 的渲染架构也随着 GPU 的演进不断地改进。在 XFree86 3.3 的时候，X 的开发者设计了 XAA（XFree86 Acceleration Architecture）架构；在 X.Org Server 6.9 版本，开发者用改进的 EXA 取代了 XAA；当 DRM 中使用了 GEM 后，Intel 的 GPU 驱动开发者们重新实现了 EXA，并命名为 UXA（Unified Acceleration Architecture）；随着 Intel 推出 Sandy Bridge 及 ivy Birdge 芯片组，Intel 又开发了 SNA（SandyBridge's New Acceleration）。

后续，我们以成熟且稳定的 UXA 为例进行讨论。在 UXA 架构下，X 的画笔盒子如下：

```
xf86-video-intel-2.19.0/uxa/uxa-accel.c:

const GCOps uxa_ops = {
    uxa_fill_spans,
    ...
    uxa_poly_lines,
    uxa_poly_segment,
    miPolyRectangle,
    uxa_check_poly_arc,
    miFillPolygon,
    ...
};
```

我们看到 uxa_ops 包含在 Intel 的 GPU 驱动中，当然，这是非常合理的，因为只有 GPU 自己最清楚哪些渲染自己可以胜任，哪些还需要 CPU 来负责。在 uxa_ops 中，有一部分画笔来自 GPU，另外一部分来自 CPU。

对于每一个绘制操作，UXA 首先检查 GPU 是否支持这个绘制操作，或者说在某些条件下，对于这个绘制操作，GPU 渲染的比 CPU 更快。如果 GPU 支持这个绘制操作，UXA 首先将绘制的命令翻译为 GPU 可以识别的指令，并将这个指令、绘制所需的相关数据，以及保存像素阵列的 BO 在显存地址空间中的地址，一同保存在用户空间的批量缓冲（Batch Buffer），然后通过 DRM 将用户空间的批量缓冲复制到内核为批量缓冲创建的 BO，之后通知 GPU 从 BO 中读取指令和数据进行绘制。实际上，DRM 按照 Intel GPU 的要求在批量缓冲和 GPU 之间还组织了一个环形缓冲区（Ring buffer），但是我们暂时忽略它，这对于理解 2D 渲染过程没有任何影响，后面在讨论 3D 渲染过程时，我们会简单的讨论这个环形缓冲区。

如果 GPU 不支持这个绘制操作，那么 UXA 将代表帧缓冲的 BO 映射到 X 服务器的用户空间，X 服务器借助 fb 层中的实现，使用 CPU 进行绘制。

也就是说，UXA 在 fb 和 GPU 加速的上面封装了一层，其根据具体绘制动作选择使用来

自 GPU 的画笔或来自 CPU 的画笔。

综上，X 的 2D 渲染过程如图 8-5 所示。

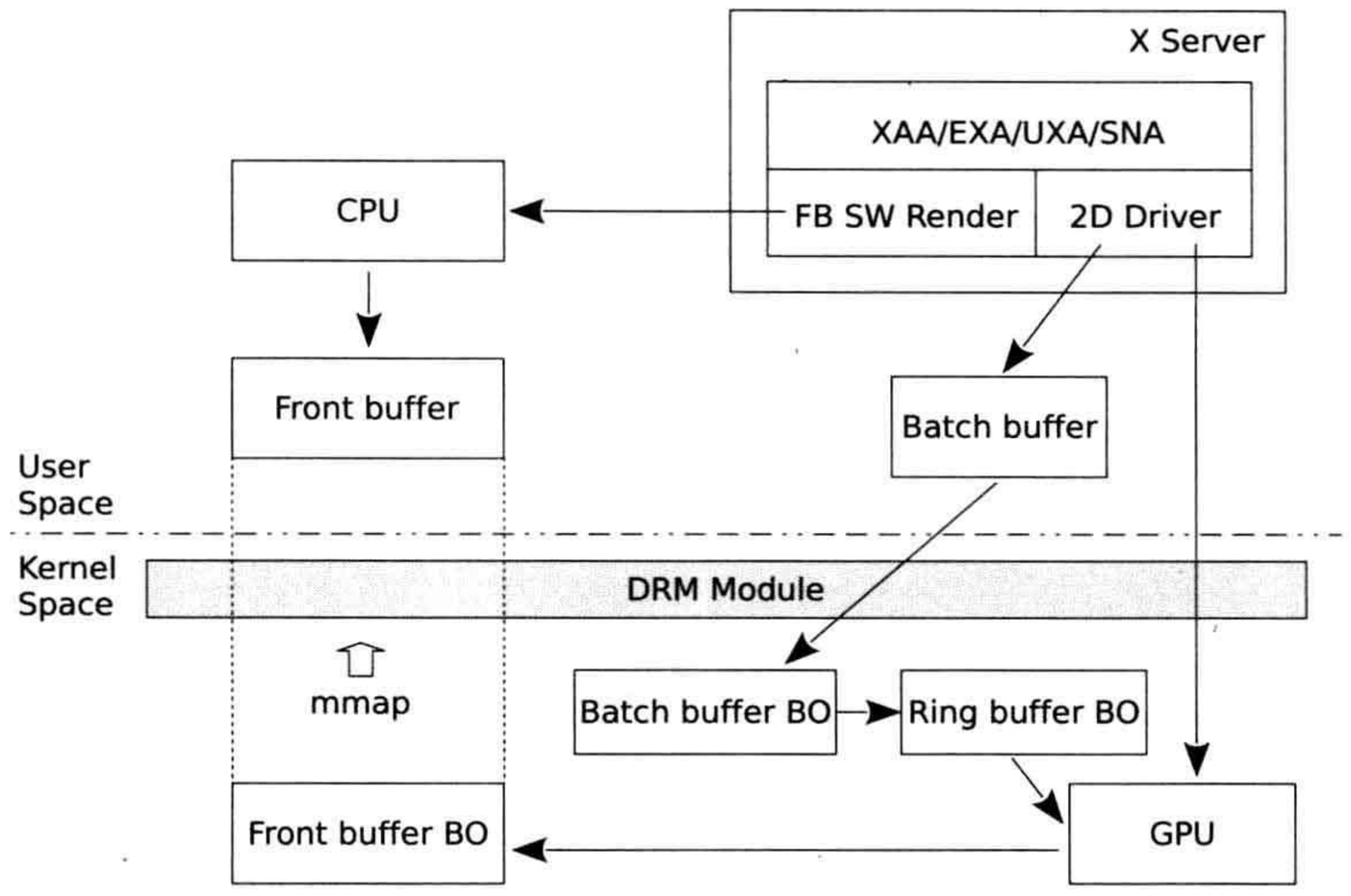


图 8-5 X 的 2D 渲染过程

不知读者是否注意到，无论是 fbGCOps，还是 uxa_ops，其中均有个别的绘制函数以“mi”开头。这些以“mi”开头的函数包含在 X 的 mi 层中。mi 是 Machine Independent 的缩写，顾名思义，是与机器无关的实现。笔者没有找到 X 中关于这个层的非常明确的解释，但是根据 mi 中的代码来看，其中的绘致函数根据不同的绘制条件，被拆分为调用其他 GCOps 中的绘制函数。

基本上，拆分的原因无外乎 GPU 支持的绘制原语有限，所以有些绘制操作需要分解为 GPU 可以支持的动作。或者出于绘制效率的考虑，将某些绘制操作拆分为效率更好的绘制原语。因此，X 将这些与具体绘制实现无关的代码剥离到一个单独的模块 mi 中。从这个角度或许能解释 X 为什么将这个层命名为 Machine Independent。

8.3.1 创建前缓冲

在 X 环境下，在不开启复合（Composite）扩展的情况下，所有程序共享一个前缓冲。对于 2D 程序，所有的绘制动作生成的图像的像素阵列最终都输出到这个前缓冲上，窗口只不过是前缓冲中的一块区域而已。

但是一旦开启了复合扩展，那么每个窗口都将被分配一个离屏（offscreen）的缓冲，类似于 OpenGL 环境中的后缓冲。应用将生成的像素阵列输出到这个离屏的缓冲中，在绘制完成后，X 服务器将向复合管理器（Composite Manager）发送 Damage 事件，复合管理器收到这个事件后，将离屏缓冲区的内容合成到前缓冲。为了避免复合扩展干扰我们探讨图形渲染的本质，在讨论 2D、包括后面的 3D 渲染时，我们都不考虑复合扩展开启的情况。

在 X 中，Window 和 Pixmap 是两个绘制发生的地方，Window 代表屏幕上的窗口，Pixmap 则代表离屏的一个存储区域。所以自然而然的，X 使用数据结构 Pixmap 来表示前缓冲。因为这个前缓冲对应整个屏幕，而且不属于某一个应用，因此开发者也将代表前缓冲的这个 Pixmap 称为 Screen Pixmap。后续为了行文方便，我们有时也使用 Screen Pixmap 这个词来代表前缓冲的这个 Pixmap 对象。显然，这个 Screen Pixmap 也是显示器（Screen）的资源，所以 X 将其保存到了代表显示器的结构体 _Screen 中：

```
xorg-server-1.12.2/include/scrnintstr.h:

typedef struct _Screen {
                ...
    pointer devPrivate;
                ...
} ScreenRec;
```

其中指针 devPrivate 指向这个前缓冲，后面看到如 GetScreenPixmap 的函数，就是从 _Screen 中获取前缓冲。

Pixmap 并不只是简单地抽象为像素阵列，还要包含一些解释像素数组所需要的信息，比如图形的高度、宽度、格式等，其定义如下：

```
xorg-server-1.12.2/include/pixmapstr.h:

typedef struct _Pixmap {
    DrawableRec drawable;
    PrivateRec *devPrivates;
    ...
} PixmapRec;
```

结构体 _Pixmap 中的指针 devPrivates 指向保存前缓冲的像素阵列的 BO。但是 Intel GPU 的 2D 驱动为了记录更多信息，在 BO 基础上封装了一层，封装后的数据结构为 intel_pixmap。所以，最终 Pixmap 中的 devPrivate 指向的并不是一个裸 BO，而是在 BO 上包围了一层的一个 intel_pixmap 对象。结构体 intel_pixmap 的定义如下：

```
xf86-video-intel-2.19.0/src/intel.h:

struct intel_pixmap {
    dri_bo *bo;
    ...
};
```

其中指针 bo 指向的就是保存前缓冲的像素阵列的 BO。

1. 创建前缓冲的 BO

前缓冲的 BO 是在 X 服务器启动过程中，2D 驱动初始化输出设备时，调用函数 intel_allocate_framebuffer 创建的，具体代码如下：

```
xf86-video-intel-2.19.0/src/intel_memory.c:
```

```
drm_intel_bo *intel_allocate_framebuffer(...)
{
    ...
    front_buffer = drm_intel_bo_alloc_tiled(intel->bufmgr,
        "front buffer", width, height, intel->cpp,
    &tiling_mode, &pitch, 0);
    ...
}
```

函数 drm_intel_bo_alloc_tiled 是库 libdrm 提供的 API，在库 libdrm 中对应的函数是 drm_intel_gem_bo_alloc_internal：

```
libdrm-2.4.35/intel/intel_bufmgr_gem.c:

static drm_intel_bo *drm_intel_gem_bo_alloc_internal(...)
{
    ...
        ret = drmIoctl(bufmgr_gem->fd,
                DRM_IOCTL_I915_GEM_CREATE, &create);
    ...
}
```

由代码可知，函数 drm_intel_gem_bo_alloc_internal 就是向内核中的 DRM 模块申请创建前缓冲的 BO。

2. 将前缓冲保存到屏幕对象

在创建了前缓冲的 BO 后，X 服务器为前缓冲创建了 Pixmap 对象，即 Screen Pixmap。2D 驱动则创建了封装前缓冲的 BO 的 intel_pixmap 对象。并且，X 也将各个对象关联了起来。

X 服务器创建 Pixmap 对象的代码如下：

```
xorg-server-1.12.2/mi/miscrinit.c:

Bool miCreateScreenResources(ScreenPtr pScreen)
{
    ...
        pPixmap = (*pScreen->CreatePixmap) (pScreen, ...);
    ...
        value = (pointer) pPixmap;
    ...
    pScreen->devPrivate = value;          /* pPixmap or pbits */
    return TRUE;
}
```

函数 miCreateScreenResources 首先创建了一个 Pixmap 对象，这个对象就是 Screen Pixmap。然后将屏幕对象中的指针 devPrivate 指向 Screen Pixmap。

2D 驱动中创建 intel_pixmap 对象的代码如下：

```
xf86-video-intel-2.19.0/src/intel_uxa.c:
```

```
Bool intel_uxa_create_screen_resources(ScreenPtr screen)
{
    ...
    dri_bo *bo = intel->front_buffer;
    ...
        PixmapPtr pixmap = screen->GetScreenPixmap(screen);
        intel_set_pixmap_bo(pixmap, bo);
    ...
}
```

在上面的代码中，函数 GetScreenPixmap 的目的就是获取 Screen Pixmap。其中函数 intel_set_pixmap_bo 的代码如下：

```
xf86-video-intel-2.19.0/src/intel_uxa.c:

void intel_set_pixmap_bo(PixmapPtr pixmap, dri_bo * bo)
{
    struct intel_pixmap *priv;
    ...
        priv = calloc(1, sizeof (struct intel_pixmap));
        ...
        priv->bo = bo;
  ...
    intel_set_pixmap_private(pixmap, priv);
}
```

函数 intel_set_pixmap_bo 首先创建了一个 intel_pixmap 对象，这个 intel_pixmap 对象中的指针 bo 指向的函数的第 2 个实参 bo 正是保存前缓冲像素阵列的 BO。然后调用函数 intel_set_pixmap_private 将 intel_pixmap 与该函数的第 1 个实参，即 Screen Pixmap 关联起来。

·3. 窗口与前缓冲的绑定

前缓冲已经建立起来了，但是，显然需要将窗口与前缓冲关联起来，否则在窗口上的绘制并不能体现到屏幕上。我们在编写具有图形界面的程序时，在绘制之前首先需要创建绘制所在的窗口。恰恰就是在创建窗口时，窗口与前缓冲绑定了。我们来看一下 X 中创建窗口的函数 fbCreateWindow：

```
xorg-server-1.12.2/fb/fbwindow.c:

Bool fbCreateWindow(WindowPtr pWin)
{
    dixSetPrivate(&pWin->devPrivates, fbGetWinPrivateKey(),
        fbGetScreenPixmap(pWin->drawable.pScreen));
    ...
}
```

fbCreateWindow 调用函数 fbGetScreenPixmap 获取 Screen Pixmap，并将窗口对象与 Screen Pixmap 绑定。

显然，所谓的创建窗口事实上就是将窗口与前缓冲关联起来，以后凡是发生在窗口上的绘制，都将直接绘制到前缓冲中。

8.3.2 GPU 渲染

GPU 渲染，也就是我们通常所说的硬件加速，从软件的层面所做的工作就是将数学模型按照 GPU 的规定，翻译为 GPU 可以识别的指令和数据，传递给 GPU，生成像素阵列等图像密集型计算则由 GPU 负责完成。可见，当使用 GPU 进行渲染时，在软件层面，实质上就是组织命令和数据而已。

Intel GPU 的 2D 驱动是如何将这些命令和数据传递给 GPU 的呢？读者一定想到了 BO。在 Intel GPU 的 2D 驱动中，定义了使用了一种所谓的批量缓冲来保存这些命令和数据，这里所谓的批量就是将驱动准备命令和数据放到这个缓冲，然后批量地让 GPU 来读取，这就是批量缓冲的由来。批量缓冲的相关定义在结构体 intel_screen_private 中：

```
xf86-video-intel-2.19.0/src/intel.h:

typedef struct intel_screen_private {
    ...
    uint32_t batch_ptr[4096];
    unsigned int batch_used;
    ...
    dri_bo *batch_bo;
    ...
} intel_screen_private;
```

在结构体 intel_screen_private 中，数组 batch_ptr 是 X（准确地说是 2D 驱动）在用户空间使用的组织命令和数据的地方，指针 batch_bo 则指向内核空间保存批量缓冲数据的 BO。2D 驱动将相关的命令和数据组织在数组 batch_ptr 中，然后将数组 batch_ptr 中的数据复制到 batch_bo 指向的内核空间中的 BO 中，供 GPU 来读取。

2D 驱动的代码中为了方便，也定义了几个操作批量缓冲的宏，典型的有如下的 OUT_BATCH 和 OUT_RELOC_PIXMAP_FENCED：

```
xf86-video-intel-2.19.0/src/intel_batchbuffer.h:

#define OUT_BATCH(dword) intel_batch_emit_dword(intel, dword)
#define OUT_RELOC_PIXMAP_FENCED(pixmap, reads, write, delta) \
          intel_batch_emit_reloc_pixmap(intel, pixmap, reads, \
                                        write, delta, 1)

static inline void intel_batch_emit_dword(
        intel_screen_private *intel, uint32_t dword)
{
    intel->batch_ptr[intel->batch_used++] = dword;
}

static inline void intel_batch_emit_reloc(
        intel_screen_private *intel, dri_bo * bo,...)
{
    ...
    intel_batch_emit_dword(intel, bo->offset + delta);
}
```

宏 OUT_BATCH 将命令或者数据写入数组 batch_ptr，宏 OUT_RELOC_PIXMAP_FENCED 将 BO 的在 GPU 地址空间中的虚拟地址写入数组 batch_ptr。

接下来，我们以具体的绘制方法 miPolyRectangle 为例，讨论 2D 驱动如何使用 GPU 进行绘制，也就是我们通常所说的硬件加速，具体代码如下：

xorg-server-1.12.2/mi/mipolyrect.c:

```
01 void miPolyRectangle(...)
02 {
03     ...
04     if (pGC->lineStyle == LineSolid && pGC->joinStyle ==
05             JoinMiter && pGC->lineWidth != 0) {
06         ...
07         (*pGC->ops->PolyFillRect) (pDraw, pGC, t - tmp, tmp);
08         free((pointer) tmp);
09     }
10     else {
11             ...
12            (*pGC->ops->Polylines) (...);
13             ...
14     }
15 }
```

根据不同的绘制条件，函数 miPolyRectangle 将绘制矩形的动作进行了拆分，拆分的目的是选择最合适的绘制方式进行绘制。这个拆分方法不依赖于任何具体硬件，因此，X 将这个拆分过程放到 mi 层中。

如果矩形的线性是实心填充的，且线段交汇处是尖角（JoinMiter）风格的，并且宽度不为 0，那么使用方法 PolyFillRect 绘制，见代码第 4~9 行。否则，使用方法 Polylines 绘制，如代码第 10~14 行所示。

以 PolyFillRect 为例，其在 UXA（即 uxa_ops）中对应的具体函数是 uxa_poly_fill_rect：

xf86-video-intel-2.19.0/uxa/uxa-accel.c:

```
01 static void uxa_poly_fill_rect(...)
02 {
03     ...
04     if (pGC->fillStyle != FillSolid && ...) {
05         goto fallback;
06     }
07     ...
08     if (!(*uxa_screen->info->prepare_solid) (pPixmap, ...)) {
09 fallback:
10         uxa_check_poly_fill_rect(pDrawable, pGC, nrect, prect);
11         goto out;
12     }
13     ...
14             (*uxa_screen->info->solid) (pPixmap,
15                 x1 + xoff, y1 + yoff, x2 + xoff, y2 + yoff);
16     ...
17 }
```

函数 uxa_poly_fill_rect 首先检查各种绘制条件以确认是否适合使用 GPU 进行绘制，如代码第 4~7 行所示。如果适合使用 GPU 进行绘制，则陆续调用函数 prepare_solid 和 solid 为 GPU 准备指令和数据，下面我们会重点讨论这两个函数。

否则，正如第 5 行代码所示，跳转到标签 fallback 处，即代码第 9 行，使用函数 uxa_check_poly_fill_rect 进行绘制，这个函数实际是使用 CPU 进行绘制的，我们将在 8.3.3 节进行讨论。

UXA 中的函数指针 solid 指向 intel_uxa_solid：

xf86-video-intel-2.19.0/src/intel_uxa.c:

```
01 static void intel_uxa_solid(PixmapPtr pixmap, int x1,
02     int y1, int x2, int y2)
03 {
04     ...
05     {
06         BEGIN_BATCH_BLT(6);
07
08         cmd = XY_COLOR_BLT_CMD;
09         ...
10         OUT_BATCH(cmd);
11
12         OUT_BATCH(intel->BR[13] | pitch);
13         OUT_BATCH((y1 << 16) | (x1 & 0xffff));
14         OUT_BATCH((y2 << 16) | (x2 & 0xffff));
15         OUT_RELOC_PIXMAP_FENCED(pixmap,
16            I915_GEM_DOMAIN_RENDER, I915_GEM_DOMAIN_RENDER, 0);
17         OUT_BATCH(intel->BR[16]);
18         ADVANCE_BATCH();
19     }
20 }
```

根据前面讨论的批量缓冲以及为操作批量缓冲封装的几个宏，读者一定已经看出来了，上面的代码是在组织批量缓冲在用户空间的数组 batch_ptr。那么函数 intel_uxa_solid 向 batch_ptr 中填入各项的意义是什么呢？这个显然要参考 Intel GPU 的指令格式。根据第 8 行代码可见，2D 驱动发送给 GPU 的指令是 XY_COLOR_BLT，该指令的功能是对目标区域以指定颜色填充。Intel GPU 的指令 XY_COLOR_BLT 的格式如表 8-1 所示。

下面我们结合表 8-1 来分析函数 intel_uxa_solid 组织批量缓冲的过程。

1）由表 8-1 可见，指令 XY_COLOR_BLT 包含 6 个双字（DWord），第 10 行代码填充的是第 0 个双字，其中“BR00”是什么意思呢？事实上，GPU 内部分为多个微核，处理不同的命令，处理 2D 指令的微核称为 BLT 引擎（Engine）。对于每个 2D 指令，每个双字实际上分别被送往 BLT 引擎的各个寄存器中。因此，这里的 BR 就是“BLT Register”的简写，如指令 XY_COLOR_BLT 中的第 1 个双字被送往 BLT 引擎的第 0 个寄存器，第 2 个双字被送往 BLT 引擎的第 13 个寄存器，等等。

表 8-1　Intel GPU 指令 XY_COLOR_BLT 的格式

双字（寄存器）	位	描　述
0 = BR00	31:29	02h – BLT 引擎
	28:22	指令操作码 : 50h
	…	…
1 = BR13	25:24	色深 : 00 = 8 位 ; 01 = 16 位 ; …
	15:00	目标图像的跨度
	…	…
2 = BR22	31:16	目标区域顶部坐标（Y1）
	15:00	目标区域左侧坐标（X1）
3 = BR23	31:16	目标区域底部坐标（Y2）
	15:00	目标区域右侧坐标（X2）
4 = BR09	31:00	目标区域基址
5 = BR16	31:00	填充颜色

（来源于“Intel OpenSource HD Graphics Programmer’s Reference Manual (PRM) Volume 1 Part 5: Graphics Core – Blitter Engine (SandyBridge) May 2011 Revision 1.0”）

因此，对于 GPU 指令来说，需要指明自己需要哪个微核来处理，这就是第一个双字中第 29~31 位的作用，0x2 表示 2D 指令、0x3 表示 3D 指令等。寄存器 BR00 中最重要的就是指令的操作码，即第 22~28 位，对于指令 XY_COLOR_BLT，其操作码是 0x50。其余位主要用于控制，如第 11 位用于控制是否打开 Tiling，等等。因此，寄存器 BR00 也被称为“BLT Opcode & Control Register”。

根据宏 XY_COLOR_BLT_CMD 的定义：

```
xf86-video-intel-2.19.0/src/i830_reg.h:

#define XY_COLOR_BLT_CMD          ((2<<29)|(0x50<<22)|(0x4))
```

其中从第 22 位开始的 0x50 正是指令 XY_COLOR_BLT 的指令操作码。第 29~30 位设置为 2，告诉 GPU 这个指令是一个 2D 指令，需要 GPU 定向给 BLT 引擎。

2）第 12 行代码填充的是第 1 个双字，对应 BLT 引擎的寄存器 BR13。其中“intel->BR[13]”是为了方便构建指令在程序中定义的一个变量，保存寄存器 BR13 的值。这就是函数 uxa_poly_fill_rect 在调用 solid 之前，调用函数 prepare_solid 的目的。在 UXA（uxa_ops）中，prepare_solid 对应的函数是 intel_uxa_prepare_solid：

```
xf86-video-intel-2.19.0/src/intel_uxa.c:

static Bool intel_uxa_prepare_solid(PixmapPtr pixmap, int alu,
                    Pixel planemask, Pixel fg)
{
                ...
    switch (pixmap->drawable.bitsPerPixel) {
```

```
    case 8:
        break;
    case 16:
        /* RGB565 */
        intel->BR[13] |= (1 << 24);
        break;
    case 32:
        /* RGB8888 */
        intel->BR[13] |= ((1 << 24) | (1 << 25));
        break;
    }
    intel->BR[16] = fg;

    return TRUE;
}
```

函数 intel_uxa_prepare_solid 根据图像实际使用的色深，设置相应的位。intel_uxa_prepare_solid 除了计算了寄存器 BR13 中的色深外，也计算了寄存器 BR16 的值，BR16 中的值是 GPU 进行填充时使用的颜色。

除了设置色深外，第 12 行代码也设置了图像的跨度（pitch），跨度是以字节为单位表示的图像的一行的长度。

3）第 13 行代码填充了第 2 个双字，对应 BLT 引擎的寄存器 BR22，这个寄存器中保存的是目标区域的左上角的坐标。

4）第 14 行代码填充了第 3 个双字，对应 BLT 引擎的寄存器 BR23，这个寄存器中保存的是目标区域的右下角的坐标。

5）第 15~16 行代码填充的第 4 个双字，对应 BLT 引擎的寄存器 BR09，这个寄存器中保存的是目标区域在 GPU 的显存空间中的地址。这里的 pixmap 就是 Screen Pixmap，所以宏 OUT_RELOC_PIXMAP_FENCED 就是将保存前缓冲的像素阵列的 BO 在显存空间中的地址填充到这个双字中。读者可以参见前面关于宏 OUT_RELOC_PIXMAP_FENCED 的介绍。

6）第 17 行代码填充了第 5 个双字，对应 BLT 引擎的寄存器 BR16，这个寄存器中保存的是填充使用的颜色。

在完成指令 XY_COLOR_BLT 的构建后，函数 intel_batch_submit 将用户空间的 batch_ptr 中的数据复制内核空间的，并通知 GPU，开始执行批量缓冲中的指令，代码如下：

xf86-video-intel-2.19.0/src/intel_batchbuffer.c:

```
void intel_batch_submit(ScrnInfoPtr scrn)
{
    ...
    ret = dri_bo_subdata(intel->batch_bo, 0, intel->batch_used*4,
                         intel->batch_ptr);
    if (ret == 0) {
        ret = drm_intel_bo_mrb_exec(intel->batch_bo,
                                    intel->batch_used*4, ...));
    }
```

```
    ...
}
```

其中函数 dri_bo_subdata，我们已经在 8.2.2 节讨论过，其负责将数据从用户空间复制到内核空间。所以这里就是 2D 驱动将组织在用户空间中的数组 batch_ptr 中的数据复制到批量缓冲在内核中对应的 BO。

函数 drm_intel_bo_mrb_exec 通知 GPU 开始执行批量缓冲中的指令。方式是通过写 GPU 的一个寄存器，具体过程请参考 8.4.2 节。

8.3.3 CPU 渲染

根据上节讨论的函数 uxa_poly_fill_rect，我们看到，GPU 并不是接收全部的绘制实心矩形的操作。对于不满足 GPU 条件的实心矩形，则将求助于 CPU 绘制，对应的函数是 uxa_check_poly_fill_rect：

```
xf86-video-intel-2.19.0/uxa/uxa-unaccel.c:

void uxa_check_poly_fill_rect(...)
{
    ...
    if (uxa_prepare_access(pDrawable, UXA_ACCESS_RW)) {
        ...
            fbPolyFillRect(pDrawable, pGC, nrect, prect);
        ...
    }
}
```

BO 是由 DRM 模块在内核空间分配的，因此运行在用户空间的 X（2D 驱动）要想访问这个内存，必须首先要将其映射到用户空间，这是由函数 uxa_prepare_access 来完成的。然后，X 使用 CPU 在映射到用户空间的 BO 上进行绘制。看到以 fb 开头的函数 fbPolyFillRect，读者一定猜到了，这就是 X 的 fb 层的函数，而 fb 层正是软件渲染的实现。

（1）映射 BO 到用户空间

函数 uxa_check_poly_fill_rect 调用 uxa_prepare_access 将 BO 映射到用户空间：

```
xf86-video-intel-2.19.0/uxa/uxa.c:

Bool uxa_prepare_access(DrawablePtr pDrawable, ...)
{
    ...
    if (uxa_screen->info->prepare_access)
        return (*uxa_screen->info->prepare_access) (pPixmap, ...);
    return TRUE;
}
```

在 UXA（uxa_ops）中，指针 prepare_access 指向的函数是 intel_uxa_prepare_access：

```
xf86-video-intel-2.19.0/src/intel_uxa.c:
```

```
static Bool intel_uxa_prepare_access(PixmapPtr pixmap, ...)
{
    ...
    struct intel_pixmap *priv = intel_get_pixmap_private(pixmap);
    dri_bo *bo = priv->bo;
    ...
    ret = drm_intel_gem_bo_map_gtt(bo);
    ...
}
```

函数 intel_uxa_prepare_access 通过 libdrm 库中的函数 drm_intel_gem_bo_map_gtt 申请内核中的 DRM 模块将保存前缓冲的像素阵列的 BO 映射到用户空间：

```
libdrm-2.4.35/intel/intel_bufmgr_gem.c:

int drm_intel_gem_bo_map_gtt(drm_intel_bo *bo)
{
    ...
    ret = map_gtt(bo);
    ...
}

static int map_gtt(drm_intel_bo *bo)
{
    ...
        bo_gem->gtt_virtual = mmap(0, bo->size, PROT_READ |
                    PROT_WRITE, MAP_SHARED, bufmgr_gem->fd, ...);
    ...
}
```

看到熟悉的函数 mmap，读者应该一切都明白了。从 CPU 的角度看，BO 与普通内存并无区别，所以，映射 BO 与映射普通内存完全相同。其中 bufmgr_gem->fd 指向的就是代表 BO 的共享内存。

（2）使用 CPU 在映射到用户空间的 BO 上进行绘制

我们再来简单看看软件渲染函数 fbPolyFillRect 的实现：

```
xorg-server-1.12.2/fb/fbfillrect.c:

void fbPolyFillRect(...)
{
    ...
            fbFill(...);
    ...
}

xorg-server-1.12.2/fb/fbfill.c:

void fbFill(...)
{
    ...
    switch (pGC->fillStyle) {
    case FillSolid:
```

```
#ifndef FB_ACCESS_WRAPPER
        if (and || !pixman_fill((uint32_t *) dst, dstStride, dstBpp,
                partX1 + dstXoff, partY1 + dstYoff,
                (partX2 - partX1), (partY2 - partY1), xor))
#endif
            fbSolid(...);
        break;
                ...
}
```

```
xorg-server-1.12.2/fb/fbsolid.c:

void fbSolid(...)
{
    ...
                WRITE(dst++, xor);
    ...
}

xorg-server-1.12.2/fb/fb.h:

#define WRITE(ptr, val) (*(ptr) = (val))
```

根据上面的代码可见，X 的软件渲染层（即 fb 这一层），或者借助库 pixman 中的 API，或者自己直接操作像素数组，完成图形的绘制。其原理非常简单，就是直接设置像素数组中的颜色值或索引。

经过对 2D 渲染的探讨，我们看到，所谓的软件渲染和硬件加速，本质上都是生成图像的像素阵列，只不过一个是由 CPU 来计算的，另外一个是由 GPU 来计算的。当然，对于硬件加速，CPU 要充当一个翻译，将数学模型按照 GPU 的要求翻译为其可以识别的指令和数据。

8.4 3D 渲染

运行在 X 上的 2D 程序，都将绘制请求发给 X 服务器，由 X 服务器来完成绘制。但是对于 3D 图形的绘制，X 应用需要通过套接字向 X 服务器传递大量的数据，这种机制严重影响了图形的渲染效率。为了解决效率问题，X 的开发者们设计了 DRI 机制，即 X 应用不再将绘制图形的请求发送给 X 服务器了，而是由应用自行绘制。

在 Linux 平台上，OpenGL 的实现是 Mesa，所以在本节中，我们结合 Mesa，探讨 3D 的渲染过程。我们可以认为 Mesa 分为两个关键部分：

- 一部分是一套兼容 OpenGL 标准的实现，为应用程序提供标准的 OpenGL API。
- 另外一部分是 DRI 驱动，通常也被称为 3D 驱动，其中包括 Pipleline 的软件实现，也就是说，即使 GPU 没有任何 3D 计算能力，那么 Mesa 也完全可以使用 CPU 完成 3D 渲染功能。3D 驱动还负责将 3D 渲染命令翻译为 GPU 可以理解并能执行的指令。不同的 GPU 有各自的“指令集”，因此，在 Mesa 中不同的 GPU 都有各自的 3D 驱动。

Pipeline 最后将生成好的像素阵列输出到帧缓冲，但是这还不够，因为最后的输出需要显示到屏幕上。而屏幕的显示是由具体的窗口系统控制的，因此，帧缓冲还需要与具体的窗口系统相结合。但是 X 的核心协议并不包含 OpenGL 相关的协议，因此，开发者们开发了 GL 的扩展 GLX（GL Extension）。为了支持 DRI，开发者们又开发了 DRI 扩展。显然，GLX 以及 DRI 扩展在 X 和 Mesa 中均需要实现。

基本上，运行在 X 窗口系统上的 OpenGL 程序的渲染过程，可以划分为三个阶段，如图 8-6 所示。

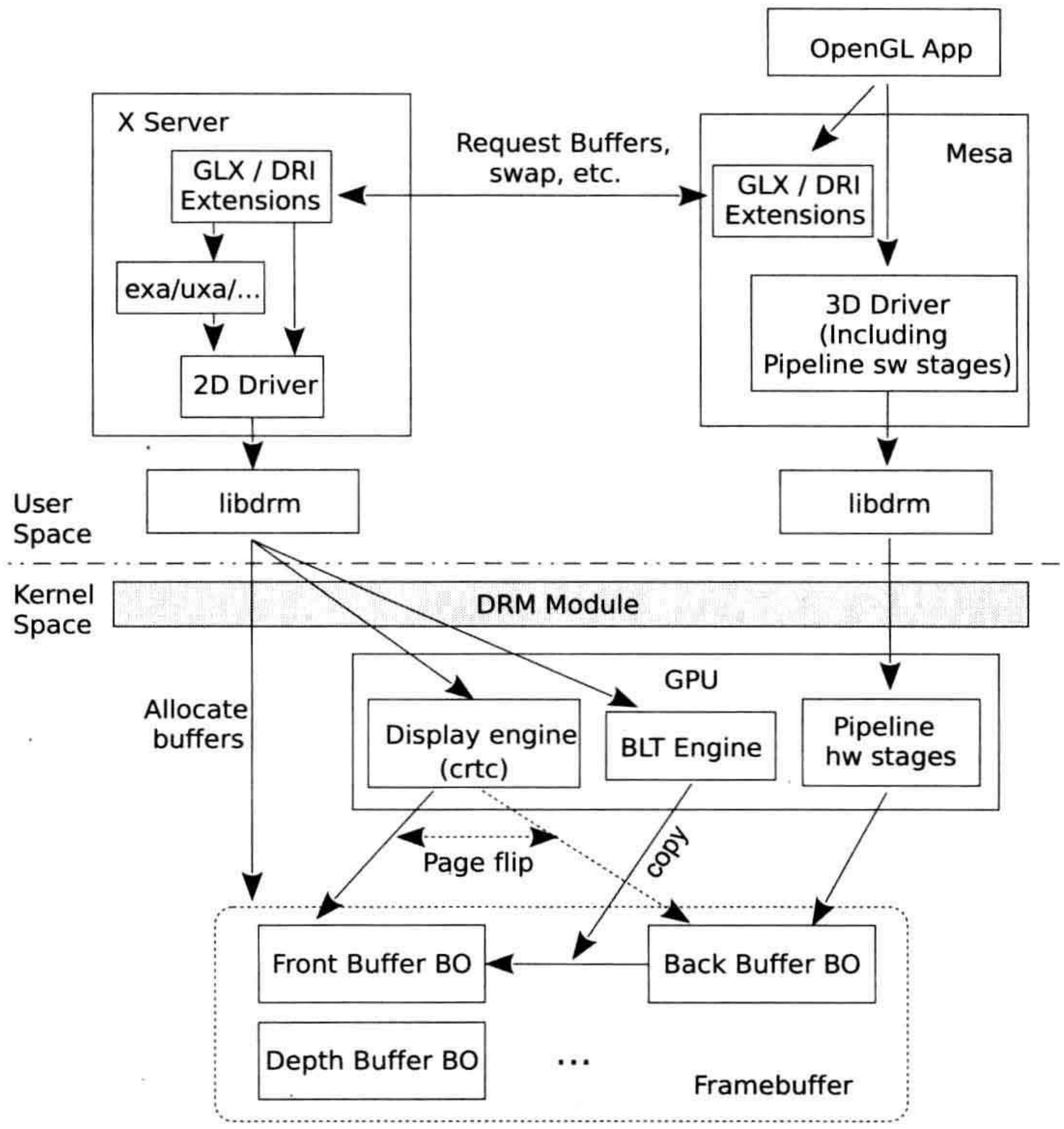


图 8-6　3D 渲染架构图

1）应用创建 OpenGL 的上下文，包括向 X 服务器申请创建帧缓冲。应用为什么不自己直接向内核的 DRM 模块请求创建帧缓冲呢？从技术上讲，应用自己请求 DRM 创建请求创建帧缓冲没有任何问题，但是为了将帧缓冲与具体的窗口系统绑定，应用只能委屈一下，放低姿态请求 X 服务器为其创建帧缓冲。这样，X 服务器就掌握了应用的帧缓冲的一手材料，在需要时，将帧缓冲显示到屏幕。帧缓冲是应用程序的“画板”，因此创建完成后，X 服务器需要将帧缓冲的 BO 的信息返回给应用。

2）应用程序建立数学模型，并通过 OpenGL 的 API 将数学模型的数据写入顶点缓冲（vertex buffer）；更新 GPU 的状态，如指定后缓冲，用来存储 Pipeline 输出的像素阵列；然

后启动 Pipeline 进行渲染。

3）渲染完成后，应用程序向 X 服务器发出交换（swap）请求。这里的交换有两种方式，一种是复制（copy），所谓复制就是将后缓冲中的内容复制到前缓冲，这是由 GPU 中 BLT 引擎负责的。但是复制的效率相对较低，所以，开发者们又设计了一种称为页翻转（page flip）的模式，在这种模式下，不需要复制动作，而是通过 GPU 的显示引擎控制显示控制器扫描哪个帧缓冲，这个被扫描的缓冲此时扮演前缓冲，而另外一个不被扫描的帧缓冲则作为应用的“画板”，也就是所说的后缓冲。

接下来我们就围绕这三个阶段，讨论 3D 程序的渲染过程。

8.4.1 创建帧缓冲

在 2D 渲染中，渲染过程都由 X 服务器完成，所以毫无争议，前缓冲由而且只能由 X 服务器创建。但是对于 DRI 程序来说，其渲染是在应用中完成，应用当然需要知道帧缓冲，但是 X 服务器控制着窗口的显示，所以 X 服务器也需要知道帧缓冲。所以，帧缓冲或者由 X 服务器创建，然后告知应用；或者由应用创建，然后再告知 X 服务器。X 采用的是前者。

虽然 OpenGL 中的帧缓冲的概念与 2D 相比有些不同，但本质上并无差别，帧缓冲中的每个缓冲都对应着一个 BO。为了管理方便，Mesa 为帧缓冲以及其中的各个缓冲分别抽象了相应的数据结构，代码如下：

```
Mesa-8.0.3/src/mesa/main/mtypes.h:

struct gl_framebuffer
{
   ...
   struct gl_renderbuffer_attachment Attachment[BUFFER_COUNT];
   ...
};
```

其中，结构体 gl_framebuffer 是帧缓冲的抽象。结构体 gl_renderbuffer 是颜色缓冲、深度缓冲等的抽象。gl_framebuffer 中的数组 Attachment 中保存的就是颜色缓冲、深度缓冲等。

在具体的 3D 驱动中，通常会以 gl_renderbuffer 作为基类，派生出自己的类。如对于 Intel GPU 的 3D 驱动，派生的数据结构为 intel_renderbuffer：

```
Mesa-8.0.3/src/mesa/drivers/dri/intel/intel_fbo.h:

struct intel_renderbuffer
{
   struct swrast_renderbuffer Base;
   struct intel_mipmap_tree *mt; /**< The renderbuffer storage. */
   ...
};
```

其中指针 mt 间接指向缓冲区对应的 BO。

如同在 Intel GPU 的 2D 驱动中，使用结构体 intel_pixmap 封装了 BO 一样，Intel GPU 的

3D 驱动也在 BO 之上包装了一层 intel_region。intel_region 中除了包括 BO 外，还包括缓冲区的一些信息，如缓冲区的宽度、高度等：

```
Mesa-8.0.3/src/mesa/drivers/dri/intel/intel_regions.h:

struct intel_region
{
   drm_intel_bo *bo;  /**< buffer manager's buffer */
   GLuint refcount; /**< Reference count for region */
   GLuint cpp;      /**< bytes per pixel */
   GLuint width;    /**< in pixels */
   ...
};
```

当 OpenGL 应用调用 glXMakeCurrent 时，就开启了创建帧缓冲的过程，这个过程可分为三个阶段：

1）OpenGL 应用向 X 服务器请求为指定窗口创建帧缓冲对应的 BO。帧缓冲中包含多个缓冲，所以当然是创建多个 BO 了。

2）X 服务器收到应用的请求后，为各个缓冲创建 BO。在创建完成后，将 BO 的名字等相关信息发送给应用。

3）应用收到 BO 信息后，将更新 GPU 的状态。比如告诉 GPU 画板在哪里。

1. 应用请求 X 服务器创建 BO

帧缓冲与具体的 GPU 密切相关，因此创建帧缓冲的发起在 3D 驱动中。以 i915 系列的 3D 驱动为例，发起创建帧缓冲的函数为 intelCreateBuffer：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_screen.c:

static GLboolean intelCreateBuffer(...)
{
   struct intel_renderbuffer *rb;
   ...
     struct gl_framebuffer *fb = CALLOC_STRUCT(gl_framebuffer);
      ...
      rb = intel_create_renderbuffer(rgbFormat);
      _mesa_add_renderbuffer(fb, BUFFER_FRONT_LEFT, ...);
   ...
}
```

函数 intelCreateBuffer 先后创建了帧缓冲对象和帧缓冲中包含的各个“子”缓冲对象，并将各“子”缓冲对象加入到帧缓冲对象的数组 Attachment 中。但是并不是 OpenGL 中规定的所有的缓冲对象都需要创建，所以函数 intelCreateBuffer 需要根据具体情况创建如前缓冲、后缓冲、深度缓冲等对象。注意，这里所谓的创建缓冲对象，仅仅是搭建起了一个空架子而已，帧缓冲尚未与具体的 BO 绑定。

一旦应用调用 glXMakeCurrent 切换自己为当前应用，glXMakeCurrent 将调用 3D 驱动中的函数 intel_update_renderbuffers 请求 X 服务器创建指定 X 窗口的各个缓冲区的 BO：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_context.c:

void intel_update_renderbuffers(__DRIcontext *context, ...)
{
   ...
   if (try_separate_stencil) {
      intel_query_dri2_buffers_with_separate_stencil(intel,
                  drawable, &buffers, &attachments, &count);
   } else {
      intel_query_dri2_buffers_no_separate_stencil(...);
   }
   ...
}
```

其中，函数intel_query_dri2_buffers_with/no_separate_stencil向X服务器申请为ID为drawable的窗口创建帧缓冲。以intel_query_dri2_buffers_with_separate_stencil为例：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_context.c:

static void intel_query_dri2_buffers_with_separate_stencil(...)
{
   ...
   struct gl_framebuffer *fb = drawable->driverPrivate;
   ...
   struct intel_renderbuffer *front_rb;
   struct intel_renderbuffer *back_rb;
   ...
   back_rb = intel_get_renderbuffer(fb, BUFFER_BACK_LEFT);
   ...
   if (back_rb) {
      (*attachments)[i++] = __DRI_BUFFER_BACK_LEFT;
      (*attachments)[i++] = intel_bits_per_pixel(back_rb);
   }
   ...
   *buffers = screen->dri2.loader->getBuffersWithFormat(drawable,
                                                ..., *attachments, ...);
   ...
}
```

函数intel_query_dri2_buffers_with_separate_stencil将帧缓冲中的各个缓冲组织为一个数组attachments，其格式是缓冲的ID加上缓冲的色深，后面组织X请求将使用这个数组attachments。然后调用getBuffersWithFormat向X服务器请求创建这些缓冲的BO。在Mesa端的DRI扩展中，getBuffersWithFormat最终调用的函数是DRI2GetBuffersWithFormat：

```
Mesa-8.0.3/src/glx/dri2.c:

DRI2Buffer * DRI2GetBuffersWithFormat(Display * dpy, XID drawable,
                           ..., unsigned int *attachments, ...)
{
   ...
   GetReqExtra(DRI2GetBuffers, count * (4 * 2), req);
   req->reqType = info->codes->major_opcode;
   req->dri2ReqType = X_DRI2GetBuffersWithFormat;
```

```
req->drawable = drawable;
req->count = count;
p = (CARD32 *) & req[1];
for (i = 0; i < (count * 2); i++)
   p[i] = attachments[i];

if (!_XReply(dpy, (xReply *) & rep, 0, xFalse)) {
...
for (i = 0; i < rep.count; i++) {
   _XReadPad(dpy, (char *) &repBuffer, sizeof repBuffer);
      buffers[i].attachment = repBuffer.attachment;
      buffers[i].name = repBuffer.name;
      buffers[i].pitch = repBuffer.pitch;
      buffers[i].cpp = repBuffer.cpp;
      buffers[i].flags = repBuffer.flags;
   }
   ...
}
```

函数 DRI2GetBuffersWithFormat 首先创建一个 X_DRI2GetBuffersWithFormat 类型的 X 请求，根据前面组织的数组 attachments，即申请创建的缓冲的信息，组织 X 请求的消息体，消息体中包含各缓冲的 ID 和色深。

然后调用 Xlib 的接口 _XReply 将请求发送给 X 服务器，并等待请求的返回。

在 X 服务器创建 BO 后，会将 BO 信息返回给应用，X 服务器创建 BO 的过程我们下节讨论。根据代码我们看到，在返回的 BO 信息中最关键的一项就是 BO 的名称。回忆 8.2.2 节的讨论，我们谈到无论是 X 服务器还是应用，均使用名称访问 BO。所以，这里返回的 BO 的名称就是为了使 DRI 应用通过这个名称访问 BO。看到名称，我们习惯上将其理解为字符串，实际上在内核的 DRM 模块中，为 BO 的名称分配的是一个数字。

2. X 服务器创建 BO

X 服务器中处理 OpenGL 应用为帧缓冲创建 BO 请求的函数是 ProcDRI2GetBuffersWithFormat：

```
xorg-server-1.12.2/hw/xfree86/dri2/dri2ext.c:

static int ProcDRI2GetBuffersWithFormat(ClientPtr client)
{
    ...
    attachments = (unsigned int *) &stuff[1];
    buffers = DRI2GetBuffersWithFormat(pDrawable, &width, &height,
                attachments, stuff->count, &count);

    return send_buffers_reply(client, pDrawable, buffers, ...);
}
```

函数 ProcDRI2GetBuffersWithFormat 首先从应用的请求中提取 attachments，然后调用函数 DRI2GetBuffersWithFormat 创建 BO，最后通过函数 send_buffers_reply 将 BO 的信息发送给应用。

函数 DRI2GetBuffersWithFormat 将调用函数 do_get_buffers 为帧缓冲创建 BO：

```
xorg-server-1.12.2/hw/xfree86/dri2/dri2.c:

static DRI2BufferPtr * do_get_buffers(...)
{
    ...
    for (i = 0; i < count; i++) {
        ...
        if (allocate or reuse buffer(...))
        ...
}
```

函数 do_get_buffers 中的变量 count 为应用请求创建 BO 的数量，显然，函数 do_get_buffers 是在循环为窗口的缓冲区创建 BO。其中 allocate_or_reuse_buffer 调用 I830DRI2CreateBuffer 为缓冲区创建 BO：

```
xf86-video-intel-2.13.0/src/intel_dri.c:

01 static DRI2Buffer2Ptr I830DRI2CreateBuffer(DrawablePtr
02                drawable, unsigned int attachment, ...)
03 {
04     ...
05     if (attachment == DRI2BufferFrontLeft) {
06         pixmap = get_front_buffer(drawable);
07         ...
08     }
09
10     if (pixmap == NULL) {
11         ...
12         pixmap = screen->CreatePixmap(...);
13         ...
14     }
15     ...
16     if ((buffer->name = pixmap_flink(pixmap)) == 0) {
17     ...
18 }
```

在前面讨论 2D 渲染时，我们已经看到，X 服务器启动时，2D 驱动在初始化输出设备时已经创建了前缓冲的 BO。因为各个窗口是共享这个前缓冲的，因此，如果 DRI 应用申请为前缓冲创建 BO，则 I830DRI2CreateBuffer 就不必创建了，其调用函数 get_front_buffer 直接查找前缓冲的 BO，如代码第 5~8 行所示。

如果函数 I830DRI2CreateBuffer 执行到第 10 行代码时，pixmap 依然空，则说明这次不是为前缓冲创建 BO，于是调用函数 CreatePixmap 为其他缓冲创建 BO。在 UXA 中，CreatePixmap 指向函数 intel_uxa_create_pixmap：

```
xf86-video-intel-2.19.0/src/intel_uxa.c:

static PixmapPtr intel_uxa_create_pixmap(...)
{
```

```
    ...
            priv->bo = drm_intel_bo_alloc_for_render(...);
    ...
}
```

函数 drm_intel_bo_alloc_for_render 是库 libdrm 提供的接口，其请求内核的 DRM 模块为缓冲区创建 BO。

创建好 BO 后，函数 I830DRI2CreateBuffer 使用库 libdrm 提供的接口 pixmap_flink，请求内核的 DRM 模块为 BO 命名，见第 16 行代码。

在创建完缓冲区的 BO 后，让我们回到函数 ProcDRI2GetBuffersWithFormat，其将调用 send_buffers_reply 将 BO 的相关信息发送给应用程序：

```
xorg-server-1.12.2/hw/xfree86/dri2/dri2ext.c:

static int send_buffers_reply(...)
{
    ...
    for (i = 0; i < count; i++) {
        xDRI2Buffer buffer;

        /* Do not send the real front buffer of a window to the client.*/
        if ((pDrawable->type == DRAWABLE_WINDOW)
            && (buffers[i]->attachment == DRI2BufferFrontLeft)) {
            continue;
        }

        buffer.attachment = buffers[i]->attachment;
        buffer.name = buffers[i]->name;
        ...
        WriteToClient(client, sizeof(xDRI2Buffer), &buffer);
    }
    return Success;
}
```

仔细观察 send_buffers_reply，可见，即使应用向 X 服务器发出了索要前缓冲的 BO 的申请，X 服务器也不会将真正的前缓冲的 BO 的信息发送给应用程序。事实上，对于运行在 X 窗口系统上的 OpenGL 应用来说，尽管应用程序有可能要求直接绘制在前缓冲上，但是 X 服务器发给 OpenGL 应用的只是一个伪前缓冲，和普通的后缓冲没有本质区别。从这里也可以看出，X 不允许 DRI 应用不通过 X 直接在前缓冲上绘制，X 不希望应用把屏幕显示搞乱，X 要对前缓冲有绝对的控制权。如果读者熟悉 Linux，一定知道第 1 版的 DRI，在开启符合管理器后，运行 DRI 应用时，那个著名的 glxgears 转动的齿轮不受复合管理器管理的 bug。

3. 更新 GPU 状态

系统中可能存在多个 OpenGL 程序并行运行但是只有一个 GPU 的情况。因此，GPU 要分时给不同的 OpenGL 程序使用。如同进程切换时，CPU 需要切换上下文一样，在对不同的 OpenGL 程序进行渲染时，GPU 也需要在不同程序之间切换。

以帧缓冲为例，每个 OpenGL 程序都有自己的帧缓冲。但是只有当前进行绘制的 OpenGL 应用的帧缓冲才是 GPU 的目标帧缓冲。因此，当不同的 OpenGL 程序进行切换时，GPU 需要切换记录帧缓冲地址的寄存器，使其指向当前正在进行绘制的程序的帧缓冲。

以 Intel i915 系列 GPU 为例，在其 3D 驱动中，对应 GPU 状态的结构体为 struct i915_hw_state：

Mesa-8.0.3/src/mesa/drivers/dri/i915/i915_context.h:

```
struct i915_hw_state
{
   GLuint Ctx[I915_CTX_SETUP_SIZE];
   GLuint Blend[I915_BLEND_SETUP_SIZE];
   GLuint Buffer[I915_DEST_SETUP_SIZE];
   ...
   struct intel_region *draw_region;
   ...
};
```

结构体 i915_hw_state 使用一系列的数组来记录 GPU 的状态，其中指针 draw_region 指向的就是保存输出的图像的像素阵列的 BO。

应用程序从 X 服务器获取了各个缓冲区的 BO 后，需要更新 GPU 中帧缓冲相关的状态。以 i915 的 3D 驱动中缓冲区更新为例，更新 GPU 的帧缓冲状态的函数是 i915_update_draw_buffer：

Mesa-8.0.3/src/mesa/drivers/dri/i915/i915_vtbl.c:

```
1 static void i915_update_draw_buffer(struct intel_context *intel)
2 {
3    ...
4       struct intel_renderbuffer *irb;
5       irb = intel_renderbuffer(fb->_ColorDrawBuffers[0]);
6       colorRegion = (irb && irb->mt) ? irb->mt->region : NULL;
7    ...
8    intel->vtbl.set_draw_region(intel, &colorRegion, ...);
9    ...
10}
```

第 5 行的变量 irb 显然是指向一个颜色缓冲区。

Intel GPU 的 3D 驱动中采用 Mipmap 的方式保存 intel_region，Mipmap 是一种为了加快渲染速度和减少图像锯齿，将贴图处理成由一系列被预先计算和优化过的图片的技术。因此，第 6 行代码中的“irb->mt->region”就是指向封装颜色缓冲 BO 的 intel_region 对象。

那么 _ColorDrawBuffers 中的第 0 个缓冲指向的是哪个颜色缓冲呢？看看下面代码片段：

Mesa-8.0.3/src/mesa/main/framebuffer.c:

```
void _mesa_initialize_window_framebuffer(...)
{
   ...
```

```
    if (visual->doubleBufferMode) {
        ...
        fb->ColorDrawBuffer[0] = GL_BACK;
        fb->_ColorDrawBufferIndexes[0] = BUFFER_BACK_LEFT;
        ...
    }
    ...
}
```

根据上面的代码片段可见，这个颜色缓冲就是后缓冲。

我们继续看函数 i915_update_draw_buffer 中的函数 set_draw_region，对于 i915 的 3D 驱动，该函数指针指向 i915_set_draw_region：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/i915_vtbl.c:

01 static void i915_set_draw_region(struct intel_context *intel,
02                        struct intel_region *color_regions[], ...)
03 {
04    ...
05    struct i915_hw_state *state = &i915->state;
06    ...
07       intel_region_reference(&state->draw_region,
08                              color_regions[0]);
09    ...
10    i915_set_buf_info_for_region(
11             &state->Buffer[I915_DESTREG_CBUFADDR0],
12             color_regions[0], BUF_3D_ID_COLOR_BACK);
13    ...
14    I915_STATECHANGE(i915, I915_UPLOAD_BUFFERS);
15 }
16
17 Mesa-8.0.3/src/mesa/drivers/dri/i915/i915_context.h:
18
19 #define I915_DESTREG_CBUFADDR0 0
```

其中，第 7~8 行代码中调用的函数 intel_region_reference 比较简单，就是将 i915_hw_state 中的 draw_region 设置为 color_regions[0]，其就是我们刚刚在函数 i915_update_draw_buffer 中讨论的后缓冲。

在考察第 10~12 行代码调用的函数 i915_set_buf_info_for_region 前，先来看一下传给这个函数的 3 个参数。根据第 19 行的宏定义，可见第 1 个参数就是 i915_hw_state 中数组 Buffer 的首地址；第 2 个参数 color_regions[0] 是后缓冲；第 3 个参数从名字上可以猜出大概是 GPU 用来标识后缓冲的 ID。了解了参数后，我们来看一下这个函数的具体代码：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/i915_vtbl.c:

void i915_set_buf_info_for_region(uint32_t *state,
            struct intel_region *region, uint32_t buffer_id)
{
   state[0] = _3DSTATE_BUF_INFO_CMD;
   state[1] = buffer_id;
```

```
    ...
}
```

显然，函数 i915_set_buf_info_for_region 就是设置 i915_hw_state 中数组 Buffer 的前两个元素的值。第一个元素被赋值为 GPU 指令 _3DSTATE_BUF_INFO_CMD；第二个元素被赋值为标识后缓冲的 ID。

更新了 i915_hw_state 中的状态信息后，函数 i915_set_draw_region 调用 I915_STATECHANGE 将状态信息组织到批量缓冲。GPU 将在进行绘制之前，从批量缓冲中读取这些信息，并更新自身的状态。宏 I915_STATECHANGE 最终调用函数 i915_emit_state 组织批量缓冲：

Mesa-8.0.3/src/mesa/drivers/dri/i915/i915_vtbl.c:

```
01 static void i915_emit_state(struct intel_context *intel)
02 {
03    ...
04       BEGIN_BATCH(count);
05       OUT_BATCH(state->Buffer[I915_DESTREG_CBUFADDR0]);
06       OUT_BATCH(state->Buffer[I915_DESTREG_CBUFADDR1]);
07       if (state->draw_region) {
08          OUT_RELOC(state->draw_region->bo,
09            I915_GEM_DOMAIN_RENDER, I915_GEM_DOMAIN_RENDER, 0);
10       } else {
11    ...
12 }
13
14 Mesa-8.0.3/src/mesa/drivers/dri/intel/intel_reg.h:
15
16 #define _3DSTATE_BUF_INFO_CMD (CMD_3D|(0x1d<<24)|(0x8e<<16)|1)
17 /* Dword 1 */
18 #define BUF_3D_ID_COLOR_BACK     (0x3<<24)
19 #define BUF_3D_ID_DEPTH          (0x7<<24)
20 ...
21 /* Dword 2 */
22 #define BUF_3D_ADDR(x)           ((x) & ~0x3)
```

根据第 5~6 行代码，批量缓冲中的前两个元素分别为 i915_hw_state 中 Buffer 数组中的第一个和第二个元素。我们刚刚讨论过，这两个元素分别是 GPU 指令 _3DSTATE_BUF_INFO_CMD 和 GPU 用来标识后缓冲的 ID 的。

笔者没有找到有关 GPU 指令 _3DSTATE_BUF_INFO_CMD 的参考，但是根据上面代码第 16~22 行的宏定义，我们可以猜出一二：

1）_3DSTATE_BUF_INFO_CMD 是个指令 ID，应该是告诉 GPU 更新相关缓冲的信息。

2）在指令码之后，紧接的第一个参数中至少应该包含要更新的缓冲区的 ID，这里 BUF_3D_ID_COLOR_BACK 应该是 GPU 内部用来标识后缓冲的 ID。我们看到这个 ID 大约占据从 24 位开始的几位，如 011 对应的是后缓冲，111 对应的是深度缓冲。

3）既然通知 GPU 更新后缓冲的地址，当然需要将后缓冲所在的 BO 告知 GPU 了。所以指令码之后的第二个参数应该是更新的缓冲的 BO。当然了，这里要使用 BO 在 GPU 地址空

间的地址。上面代码第 8~9 行的宏正是在批量缓冲中写入了后缓冲 BO 的地址。

事实上，除了更新了 GPU 中后缓冲的信息外，也更新了 GPU 的其他状态，这里不再一一讨论。

8.4.2 渲染 Pipleline

与 2D 渲染相比，3D 渲染要复杂得多。就如同有些复杂的绘画过程，要分成几个阶段一样，OpenGL 标准也将 3D 的渲染过程划分为一些阶段，并将由这些阶段组成的这一过程形象地称为 Pipleline。

应用程序建立基本的模型包括在对象坐标中的顶点数据、顶点的各种属性（比如颜色），以及如何连接这些顶点（如是连接成直线还是连接为三角形），等等，统一存储在顶点缓冲中，然后作为 Pipeline 的输入，这些输入就像原材料一样，经过 Pipeline 这台机器的加工，最终生成像素阵列，输出到后缓冲的 BO 中。

OpenGL 的标准规定了一个参考的 Pipeline，但是各家 GPU 的实现与这个参考还是有很多差别的，有的 GPU 将相应的阶段合并，有的 GPU 将个别阶段又拆分了，有的可能增加了一些阶段，有的又砍了一些阶段。但是，大体上整个过程如图 8-7 所示。

（1）顶点处理

OpenGL 使用顶点的集合来定义或逼近对象，应用程序建模实际上就是组织这些顶点，当然也包括顶点的属性。Pipeline 的第一个阶段就是顶点处理（vertex operations），顶点处理单元将几何对象的顶点从对象坐标系变换到视点坐标系，也就是将三维空间的坐标投影到二维坐标，并为每个顶点赋颜色值，并进行光照处理等。

（2）图元装配

显然，很多操作处理是不能以顶点单独进行处理的，比如裁减、光栅化等，需要将顶点组装成几何图形。Pipeline 将处理过的顶点连接成为一些最基本的图元，包括点、线和三角形等。这个过程成为图元装配（primitive assembly）。

任何一个曲面都是多个平面无限逼近的，而最基本的是三点表示一个平面。所以，理论上，GPU 将曲面都划分为若干个三角形，也就是使用三角形进行装配。但是也不排除现代 GPU 的设计者们使用其他的更有效的图元，比如梯形，进行装配。

（3）光栅化

我们前文提到，图形是使用像素阵列来表示的。所以，图元最终要转化为像素阵列，这个过程称为光栅化（rasterization），我们可以把光栅理解为像素的阵列。经过光栅化之后，图元被分解为一些片断（fragment），每个片段对应一个像素，其中有位置值（像素位置）、颜色、纹理坐标和深度等属性。

（4）片段处理

在 Pipeline 更新帧缓冲之前，Pipeline 执行最后一系列的针对每个片段的操作。对于每一个片断，首先进行相关的测试，比如深度测试、模板测试。以深度测试为例，只有当片段的

深度值小于深度缓存中与片段相对应的像素的深度值时，颜色缓冲、深度缓冲中的与片段相对应的像素的值才会被这个片段中对应的信息更新。

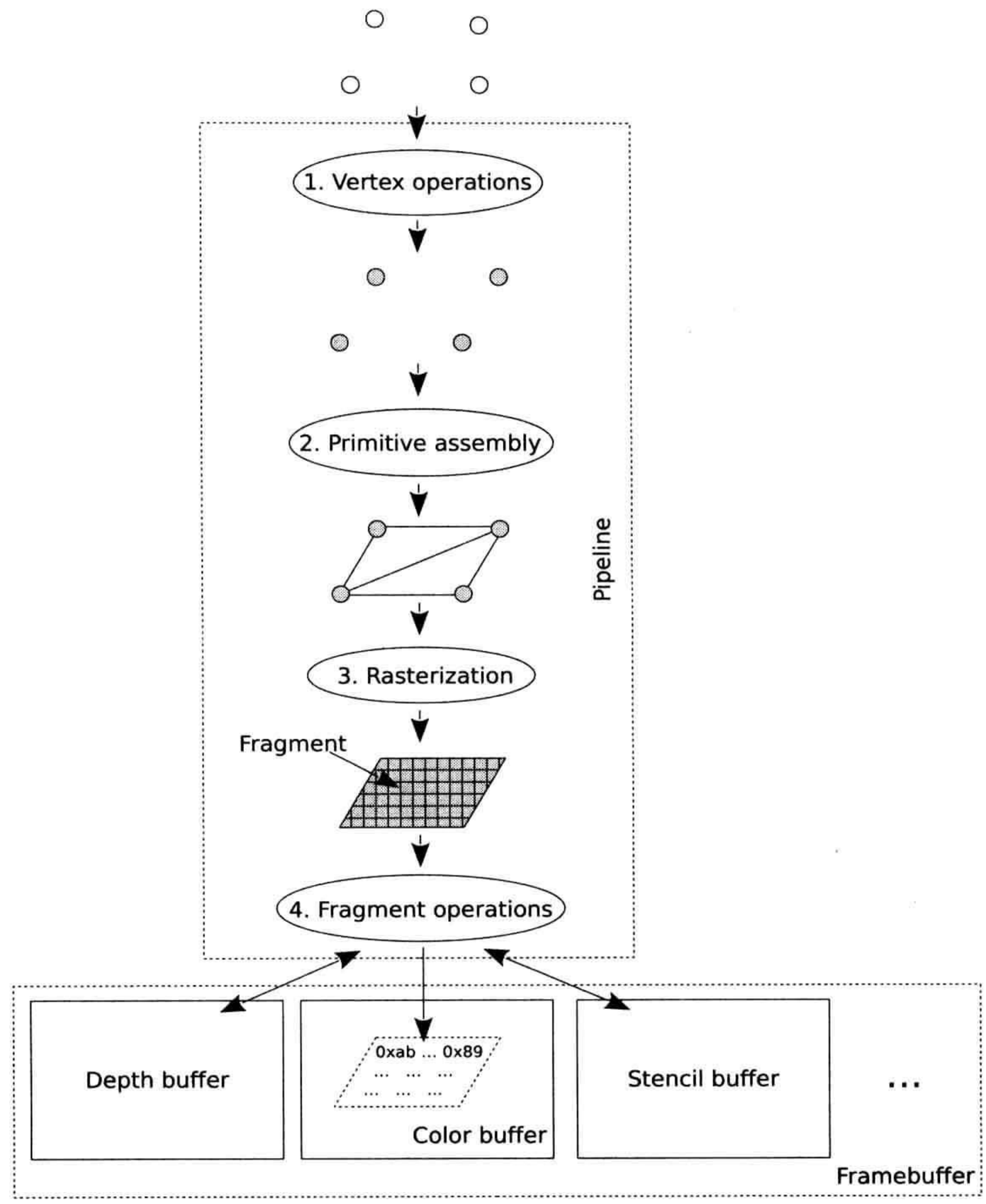


图 8-7 Pipeline

Pipeline 可全部由软件实现（CPU），也可全部由硬件实现（GPU），或者二者混合，这完全取决于 GPU 的能力。对于 GPU 没有 3D 计算能力的，则 Pipeline 完全由软件实现。比如，Mesa 中的 _tnl_default_pipeline，即是一个纯软件的 Pipeline，Pipeline 中的每一个阶段均由 CPU 负责渲染：

```
Mesa-8.0.3/src/mesa/tnl/t_pipeline.c:

const struct tnl_pipeline_stage *_tnl_default_pipeline[] = {
   &_tnl_vertex_transform_stage,
   &_tnl_normal_transform_stage,
   &_tnl_lighting_stage,
   &_tnl_texgen_stage,
```

```
    &_tnl_texture_transform_stage,
    &_tnl_point_attenuation_stage,
    &_tnl_vertex_program_stage,
    &_tnl_fog_coordinate_stage,
    &_tnl_render_stage,
    NULL
};
```

对于 3D 计算能力比较强的 GPU，如 ATI 的 GPU，Pipeline 完全由 GPU 实现。

而有些 GPU 能力不那么强大，那么 CPU 就要参与图形渲染了，因此，Pipeline 一部分由 CPU 实现，一部分由 GPU 实现，比如基于 Intel i915 GPU 的 Pipeline：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_render.c:

const struct tnl_pipeline_stage *intel_pipeline[] = {
    &_tnl_vertex_transform_stage,
    &_tnl_normal_transform_stage,
    &_tnl_lighting_stage,
    &_tnl_fog_coordinate_stage,
    &_tnl_texgen_stage,
    &_tnl_texture_transform_stage,
    &_tnl_point_attenuation_stage,
    &_tnl_vertex_program_stage,
#if 1
    &_intel_render_stage,
#endif
    &_tnl_render_stage,
    0,
};
```

相比于 _tnl_default_pipeline，intel_pipeline 使用 _intel_render_stage 替换了 _tnl_render_stage。

以 Intel GPU 为例，Pipeline 的渲染过程大致如图 8-8 所示。

1）首先，应用程序通过 glVertex 等 OpenGL API 将数据写入用户空间的顶点缓冲。

2）当程序显示调用 glFlush，或者，当顶点缓冲满时，其将自动激活 glFlush，glFlush 将启动 Pipeline。以 intel_pipeline 为例，Pipeline 的前几个阶段是 CPU 负责的，因此，所有的输入来自用户空间的顶点缓冲，计算结果也输出到用户空间的顶点缓冲；在最后的 _intel_render_stage 阶段，按照 intel GPU 的要求，从公共的顶点缓冲中读取数据，使用 intel GPU 的 3D 驱动中提供的函数，重新组织一个符合 intel GPU 规范的顶点缓冲。

3）glFlush 调用 3D 驱动中的函数 intel_glFlush。intel_glFlush 首先将顶点缓冲和批量缓冲复制到内核空间对应的 BO，实际上就是相当于复制到了 GPU 的显存空间，这样 GPU 就可以访问了。然后，内核的 DRM 模块将按照 Intel GPU 的要求建立一个环形缓冲区（ring buffer）。

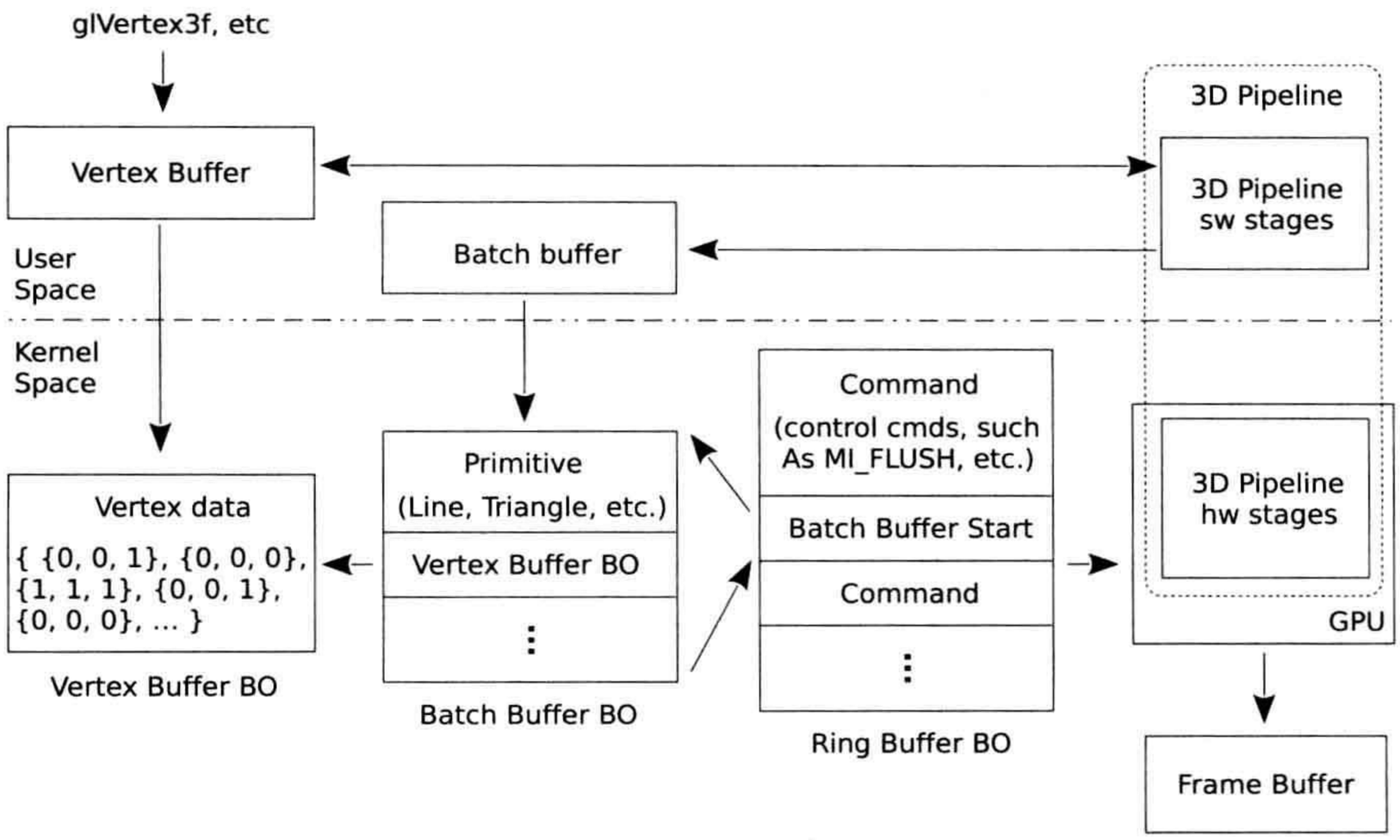


图 8-8　intel GPU 3D 渲染过程

4）准备好环形缓冲区后，内核中的 DRM 模块将环形缓冲区的信息，如缓冲区的头和尾的地址分别写入 GPU 的寄存器 Head Offset 和 Tail Offset 等。当 DRM 向寄存器 Tail Offset 写入数据时，将触发 GPU 读取并执行环形缓冲区中的命令，启动 GPU 中的 Pipeline 进行渲染。最后，GPU 的 Pipepline 将生成的像素阵列输入到帧缓冲。

1. 建立数学模型

使用 OpenGL 绘制，首先需要将绘制的内容使用数学模型描述出来，这个描述的过程的最终结果将保存在顶点缓冲中。我们以函数 glVertex3f 为例，来简单看看这个过程。

因为可能存在多个上下文，比如某个上下文使用的是软件渲染，另外一个上下文使用的是硬件渲染，因此，Mesa 采用分发函数表（dispatch table）实现访问当前上下文的 GL 函数。

GL 上下文中有一个指向结构体 _glapi_table 的指针 Exec，用于指向当前上下文的分发函数表，具体代码如下：

```
Mesa-8.0.3/src/mesa/main/mtypes.h:

struct gl_context
{
   ...
   struct _glapi_table *Exec;    /**< Execute functions */
   ...
};
```

函数表作为 GL 上下文的一部分，在创建上下文时进行初始化，具体代码如下：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_context.c:

bool intelInitContext(...)
```

```
{
    ...
    if (!_mesa_initialize_context(...) {
    ...
    _vbo_CreateContext(ctx);
    ...
}
```

其中，函数 _mesa_initialize_context 创建了函数表，并初始化了函数表中的部分 GL 函数，如 glFlush。函数 glVertex3f 是在初始化 VBO 时初始化的：

```
Mesa-8.0.3/src/mesa/vbo/vbo_exec.c:

void vbo_exec_init( struct gl_context *ctx )
{
    ...
    vbo_exec_vtx_init( exec );
    ...
}

Mesa-8.0.3/src/mesa/vbo/vbo_exec_api.c:

void vbo_exec_vtx_init( struct vbo_exec_context *exec )
{
    ...
    vbo_exec_vtxfmt_init( exec );
    ...
    _mesa_install_exec_vtxfmt( ctx, &exec->vtxfmt );
    ...
}

static void vbo_exec_vtxfmt_init( struct vbo_exec_context *exec )
{
    ...
    vfmt->Vertex3f = vbo_Vertex3f;
    ...
}
```

在函数 vbo_exec_vtxfmt_init 中，函数指针 Vertex3f 指向的函数最后会被 mesa_install_exec_vtxfmt 安装到函数表中，对应函数 glVertex3f。

我们先来看看函数 vbo_Vertex3f 的实现：

```
Mesa-8.0.3/src/mesa/vbo/vbo_exec_api.c:

#define TAG(x) vbo_##x

#include "vbo_attrib_tmp.h"

Mesa-8.0.3/src/mesa/vbo/vbo_attrib_tmp.h:

static void GLAPIENTRY TAG(Vertex3f)(GLfloat x, GLfloat y,
                                     GLfloat z)
{
```

```
    GET_CURRENT_CONTEXT(ctx);
    ATTR3F(VBO_ATTRIB_POS, x, y, z);
}
```

根据宏 TAG 的定义，显然，TAG(Vertex3f) 就是 vbo_Vertex3f 的函数实现。其中宏 ATTR3F 的定义如下：

```
Mesa-8.0.3/src/mesa/vbo/vbo_attrib_tmp.h:

#define ATTR3F( A, X, Y, Z )    ATTR( A, 3, X, Y, Z, 1 )

Mesa-8.0.3/src/mesa/vbo/vbo_exec_api.c:

#define ATTR( A, N, V0, V1, V2, V3 )                                    \
do {                                                                    \
   struct vbo_exec_context *exec = &vbo_context(ctx)->exec;             \
   ...                                                                  \
   {                                                                    \
      GLfloat *dest = exec->vtx.attrptr[A];                             \
      if (N>0) dest[0] = V0;                                            \
      if (N>1) dest[1] = V1;                                            \
      if (N>2) dest[2] = V2;                                            \
      if (N>3) dest[3] = V3;                                            \
   }                                                                    \
   ...
} while (0)
```

根据宏 ATTR 的定义可见，vbo_Vertex3f 就是将数学模型的相关数据写入顶点缓冲。

了解了函数 vbo_Vertex3f 的实现后，我们看看 _mesa_install_exec_vtxfmt 是如何将其安装到函数表的：

```
Mesa-8.0.3/src/mesa/main/vtxfmt.c:

void _mesa_install_exec_vtxfmt(struct gl_context *ctx,
                               const GLvertexformat *vfmt)
{
   if (ctx->API == API_OPENGL)
      install_vtxfmt( ctx->Exec, vfmt );
}

static void install_vtxfmt( struct _glapi_table *tab,
                            const GLvertexformat *vfmt )
{
   ...
   SET_Vertex3f(tab, vfmt->Vertex3f);
   ...
}
```

函数 SET_Vertex3f 的相关代码如下：

```
Mesa-8.0.3/src/mesa/main/dispatch.h:

static inline void SET_Vertex3f(struct _glapi_table *disp,
```

```
        void (GLAPIENTRYP fn)(GLfloat, GLfloat, GLfloat)) {
   SET_by_offset(disp, _gloffset_Vertex3f, fn);
}

#define _gloffset_Vertex3f 136

#define SET_by_offset(disp, offset, fn) \
    do { \
         ... \
           ((_glapi_proc *)(disp))[offset] = (_glapi_proc) fn; \
        } \
    } while(0)
```

因为宏 _gloffset_Vertex3f 的定义为 136，所以宏 SET_by_offset 设置函数表中第 136 项的函数指针指向函数 vbo_Vertex3f。我们来看看 GL 函数表中的第 136 项的函数指针：

```
Mesa-8.0.3/src/mapi/glapi/glapitable.h:

struct _glapi_table
{
   ...
   void (GLAPIENTRYP Vertex3f)(GLfloat x, GLfloat y, GLfloat z);
                                                        /* 136 */
   ...
};
```

我们看到函数表中的第 136 项是 Vertex3f，而不是 glVertex3f，是不是很困惑?

事实上，由于采用这种跳转函数表的方式，给 GL 函数调用带来许多不必要的开销，因此，Mesa 进行了必要的优化。比如，在 IA32 平台上，Mesa 使用汇编语言实现 OpenGL API 规定的这些函数。相比使用 C 语言，使用汇编语言实现的函数编译后的机器指令要更精简一些，相关代码如下：

```
Mesa-8.0.3/src/mapi/glapi/glapi_x86.S:

01 GLNAME(gl_dispatch_functions_start):
02         ...
03         GL_STUB(Vertex3f, 136, Vertex3f@12)
04         ...
05
06 #  define GL_STUB(fn,off,fn_alt)                          \
07 GL_PREFIX(fn, fn_alt):                                    \
08          ...                                              \
09 1:      CALL(_x86_get_dispatch) ;                        \
10          JMP(GL_OFFSET(off))
11
12 #  define GL_PREFIX(n,n2) GLNAME(CONCAT(gl,n))
```

因为要处理多种情况，再加上一些额外的汇编伪指令，所以代码比较复杂，为了增加可读性，笔者进行了必要的删减。

从第 1 行代码处开始，Mesa 使用宏 GL_STUB 开始定义 OpenGL API 规定的函数，其中第 3 行代码定义的就是函数 glVertex3f。

注意定义函数使用的宏 GL_STUB，其在第 6~10 行代码定义。其中第 7 行代码定义的是函数名，代码中宏 GL_PREFIX 在第 12 行代码定义，就是给函数名称前加个前缀 gl，所以

```
GL_PREFIX(Vertex3f, Vertex3f@12)
```

展开后为：

```
glVertex3f
```

可见，第 3 行代码使用宏 GL_STUB 定义的就是函数 glVertex3f。

我们再来看看宏 GL_STUB 定义的函数体。第 9 行代码获取函数表所在的基址，然后跳转到偏移 off 处，见第 10 行代码。以函数 glVertex3f 为例，根据第 3 行代码可见，这个偏移是 136。也就是说，当程序执行函数 glVertex3f 时，其将跳转到函数表中第 136 项指针指向的函数。

而前面函数 SET_Vertex3f 正是将函数 vbo_Vertex3f 安装到了函数表的第 136 项。也就是说，当执行函数 glVertex3f 时，实际跳转到的函数就是 vbo_Vertex3f。

2. *启动* Pipeline

在建模后，应用将顶点数据存入了顶点缓冲，加工需要的原材料已经准备好了，接下来就需要开动 Pipeline 这台加工机器了。那么，这个机器什么时候运转起来呢？通常是在程序中显示调用函数 glFlush 时。当然，一旦顶点缓冲已经充满了，也会自动调用 glFlush。读者可能有个疑问：我们编写程序时，有时并没有显示调用 glFlush 啊？没错，那是通常情况下，我们使用的都是启用了双缓冲的 OpenGL，即前缓冲和后缓冲。对于启用双缓冲的 OpenGL 程序，OpenGL 规定，当程序在后缓冲渲染完成后，请求交换到前后缓冲时，使用 OpenGL 的 API glXSwapBuffers，而实际上，函数 glXSwapBuffers 已经替我们调用了 glFlush。

当调用函数 glFlush 时，将通过函数表跳转到函数 _mesa_flush：

```
Mesa-8.0.3/src/mesa/main/context.c:

void _mesa_flush(struct gl_context *ctx)
{
   FLUSH_CURRENT( ctx, 0 );
   if (ctx->Driver.Flush) {
      ctx->Driver.Flush(ctx);
   }
}
```

函数 _mesa_flush 首先使用宏 FLUSH_CURRENT 启动 CPU 负责的 Pipeline。在 CPU 负责的 Pipeline 运行完毕后，_mesa_flush 调用驱动中的 Flush 函数将用户空间的顶点缓冲、批量缓冲的数据复制到内核空间，并启动 GPU 中的 Pipeline。

宏 FLUSH_CURRENT 调用函数 _tnl_draw_prims 开动 Pipeline，具体代码如下：

```
Mesa-8.0.3/src/mesa/tnl/t_draw.c:
```

```
void _tnl_draw_prims(...)
{
   ...
      for (i = 0; i < nr_prims;) {
         ...
         for (inst = 0; inst < prim[i].num_instances; inst++) {
            ...
            TNL_CONTEXT(ctx)->Driver.RunPipeline(ctx);
            ...
         }
      }
   ...
}
```

我们看到，对于每个绘制原语，函数 _tnl_draw_prims 分别启动 Pipeline 对其进行加工。对于 Intel GPU 的 3D 驱动，RunPipeline 指向的函数是 intelRunPipeline：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_tris.c:

static void intelRunPipeline(struct gl_context * ctx)
{
   ...
   _tnl_run_pipeline(ctx);
...
}
Mesa-8.0.3/src/mesa/tnl/t_pipeline.c:

void _tnl_run_pipeline( struct gl_context *ctx )
{
   ...
   for (i = 0; i < tnl->pipeline.nr_stages ; i++) {
      struct tnl_pipeline_stage *s = &tnl->pipeline.stages[i];
      if (!s->run( ctx, s ))
         break;
   }
   ...
}
```

函数 _tnl_run_pipeline 依次运行 Pipeline 中每个阶段的 run 函数，一旦某个阶段的函数 run 返回 False，则表明整个 Pipeline 运行结束。

3. Pipeline 中的软件计算阶段

所谓的软件计算阶段，是指计算过程是由 CPU 来负责的。CPU 从上下文中获取上个阶段的状态信息，进行计算，然后将计算结果保存到上下文中，作为下一个阶段的输入。上下文的数据抽象为结构体 TNLcontext，其中非常重要的一个成员是结构体 vertex_buffer：

```
Mesa-8.0.3/src/mesa/tnl/t_context.h:

typedef struct
{
   ...
   struct vertex_buffer vb;
```

```
    ...
} TNLcontext;
```

顾名思义，结构体 vertex_buffer 是保存顶点数据的。软件计算阶段的所有顶点数据来自这个 vertex_buffer，经过变换后的顶点数据也输出到这个 vertex_buffer 中。

以 intel_pipeline 中的 texgen 阶段为例：

Mesa-8.0.3/src/mesa/tnl/t_vb_vertex.c:

```
01 static GLboolean run_texgen_stage( struct gl_context *ctx,
02                                    struct tnl_pipeline_stage *stage )
03 {
04    struct vertex_buffer *VB = &TNL_CONTEXT(ctx)->vb;
05    struct texgen_stage_data *store = TEXGEN_stage_DATA(stage);
06    ...
07    for (i = 0 ; i < ctx->Const.MaxTextureCoordUnits ; i++) {
08       ...
09          store->TexgenFunc[i]( ctx, store, i );
10
11          VB->AttribPtr[VERT_ATTRIB_TEX0 + i] =
12              &store->texcoord[i];
13       }
14    ...
15 }
```

第9行代码计算纹理的坐标，并将结果保存到 store 的数组 texcoord 中。而在函数 TexgenFunc 的计算过程中，使用了来自 TNLcontext 中的结构体 vertex_buffer 中的各种状态信息。

计算完成后，函数 run_texgen_stage 也将这个阶段的计算结果保存到了 TNLcontext 中的结构体 vertex_buffer 中，如代码第 11~12 行所示。

4. Pipeline 中 GPU 相关的阶段

很难要求所有厂家的 GPU 都按照一个标准设计，所以在启动 GPU 中的硬件阶段之前，需要将 OpenGL 标准规定的标准格式的顶点缓冲中的数据按照具体的 GPU 的要求组织一下，然后再传递给 GPU。下面我们就以 Intel i915 系列 GPU 的 Pipeline 中的 _intel_render_stage 为例，看看其是如何为 GPU 准备批量缓冲的。

前面我们在函数 _tnl_run_pipeline 中看到，Pipeline 在运行时，是依次调用各个阶段的 run 函数来运行各个阶段的。_intel_render_stage 阶段的 run 函数是 intel_run_render：

Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_render.c:

```
01 static GLboolean intel_run_render(struct gl_context * ctx, ...)
02 {
03    ...
04    for (i = 0; i < VB->PrimitiveCount; i++) {
05       GLuint prim = _tnl_translate_prim(&VB->Primitive[i]);
06       GLuint start = VB->Primitive[i].start;
07       GLuint length = VB->Primitive[i].count;
```

```
08        ...
09        intel_render_tab_verts[prim & PRIM_MODE_MASK] (ctx,
10                                               start, start + length, prim);
11    }
12    ...
13    INTEL_FIREVERTICES(intel);
14    ...
15 }
```

其中，代码第 4~11 行的 for 循环，将依次调用特定 GPU 相关的函数按照 GPU 要求的格式重新组织顶点缓冲。以 Intel GPU 的 3D 驱动为例，其另外分配了与驱动相关的顶点缓冲存储重新组织顶点数据：

```
Mesa-8.0.3/src/mesa/drivers/dri/intel/intel_context.h:

struct intel_context
{
   ...
   struct
   {
      ...
      drm_intel_bo *vb_bo;
      uint8_t *vb;
      ...
   } prim;
   ...
}
```

在结构体 prim 中，vb 指向的是用户空间的顶点缓冲，vb_bo 指向的是内核空间创建的保存顶点数据的 BO。

intel i915 系列 GPU 的 3D 驱动中组织三角形的顶点缓冲的函数为 intel_draw_triangle：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_tris.c:

static void intel_draw_triangle(struct intel_context *intel,
       intelVertexPtr v0, intelVertexPtr v1, intelVertexPtr v2)
{
   GLuint vertsize = intel->vertex_size;
   GLuint *vb = intel_get_prim_space(intel, 3);
   int j;

   COPY_DWORDS(j, vb, vertsize, v0);
   COPY_DWORDS(j, vb, vertsize, v1);
   COPY_DWORDS(j, vb, vertsize, v2);
}
```

函数 intel_draw_triangle 使用宏 COPY_DWORDS 向顶点缓冲中指定偏移处写入顶点数据。对于每一个三角形来说都包括三个顶点数据，因此调用三次宏 COPY_DWORDS，将三角形的三个顶点写入了顶点缓冲。

处理完顶点缓冲后，函数 intel_run_render 就将开始为 GPU 组织批量缓冲。Intel GPU 的 3D 驱动中批量缓冲的数据抽象如下：

```
Mesa-8.0.3/src/mesa/drivers/dri/intel/intel_context.h:

struct intel_context
{
   ...
   struct intel_batchbuffer {
      drm_intel_bo *bo;
      ...
      uint32_t map[8192];
      ...
   } batch;
   ...
}
```

在结构体 intel_batchbuffer 中，数组 map 就是用户空间中的批量缓冲，bo 指向的就是内核空间中的保存批量数据的 BO。可见，3D 驱动中使用批量缓冲的方式与 2D 驱动中的基本相同。

函数 intel_run_render 在最后调用了宏 INTEL_FIREVERTICES，开启了批量缓冲的生成过程：

```
Mesa-8.0.3/src/mesa/drivers/dri/intel/intel_context.h:

#define INTEL_FIREVERTICES(intel)       \
do {                                    \
   if ((intel)->prim.flush)             \
      (intel)->prim.flush(intel);       \
} while (0)
```

函数指针 flush 指向函数 intel_flush_prim：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_tris.c:

01 void intel_flush_prim(struct intel_context *intel)
02 {
03    ...
04       BEGIN_BATCH(2+len);
05       if (cmd)
06          OUT_BATCH(_3DSTATE_LOAD_STATE_IMMEDIATE_1 | cmd | ...));
07       if (vb_bo != i915->current_vb_bo) {
08          OUT_RELOC(vb_bo, I915_GEM_DOMAIN_VERTEX, 0, 0);
09          i915->current_vb_bo = vb_bo;
10       }
11       ...
12       OUT_BATCH(_3DPRIMITIVE |
13                 PRIM_INDIRECT |
14                 PRIM_INDIRECT_SEQUENTIAL |
15                 intel->prim.primitive |
16                 count);
17       OUT_BATCH(offset / (intel->vertex_size * 4));
18       ADVANCE_BATCH();
19    ...
20 }
```

这里，我们再次看到与 2D 驱动中类似的宏定义（如 OUT_BATCH 等），它们基本与 2D 驱动中的定义完全相同，我们不再展开分析这些宏定义了。在上述组织批量缓冲的代码片段中：

1）第 6 行代码在批量缓冲中填充了发给 GPU 的 3D 命令的指令代码（opcode）；

2）第 8 行代码在批量缓冲中填充了引用的保存顶点数据的 BO；

3）第 12~16 行代码在批量缓冲中填写了渲染原语的相关信息，比如绘制的是三角形还是线段等；

4）第 17 行代码指明了绘制这个原语需要的顶点数据在保存顶点数据的 BO 中的偏移。

至此，用户空间中的批量缓冲也准备好了。下一步，就是将用户空间的数据复制到内核空间的 BO，并启动 GPU 中的 Pipeline。

5. 复制顶点数据和批量数据到内核空间

在 Pipeline 的软件阶段，所有阶段的计算结果都保存在用户空间，为了启动 Pipeline 的硬件阶段，显然需要将这些数据复制到内核空间的 BO，这样 GPU 才可以访问。_mesa_flush 最后将调用 3D 驱动中的函数 _intel_batchbuffer_flush 进行复制：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_batchbuffer.c:

int _intel_batchbuffer_flush(...)
{
   ...
   if (intel->vtbl.finish_batch)
      intel->vtbl.finish_batch(intel);
   ...
   ret = do_flush_locked(intel);
   ...
}
```

我们先来看一下函数 finish_batch。对于 i915 来说，其指向的函数是 intel_finish_vb：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_tris.c:

void intel_finish_vb(struct intel_context *intel)
{
   ...
   drm_intel_bo_subdata(intel->prim.vb_bo, 0, ...);
   ...
}
```

函数 drm_intel_bo_subdata 我们已经见过了，其将用户空间的顶点缓冲中的数据复制到内核空间中保存顶点数据的 BO。

接下来，再来看函数 _intel_batchbuffer_flush 中调用的 do_flush_locked：

```
Mesa-8.0.3/src/mesa/drivers/dri/i915/intel_batchbuffer.c:

static int do_flush_locked(struct intel_context *intel)
```

```
{
    ...
    ret = drm_intel_bo_subdata(batch->bo, 0, 4*batch->used, ...);
    ...
          ret = drm_intel_bo_mrb_exec(...);
    ...
}
```

函数 do_flush_locked 首先调用 drm_intel_bo_subdata 将用户空间的批量缓冲中的数据复制到内核空间中保存批量数据的 BO。至此，用户空间的顶点缓冲和批量缓冲中的数据都被复制到内核空间的 BO。

在将用户空间的数据复制到内核空间中的 BO 后，do_flush_locked 调用库 libdrm 中的函数 drm_intel_bo_mrb_exec 通知 GPU 启动其 Pipeline 开始渲染，这个过程我们下一节讨论。

6. *启动 GPU 中的* Pipeline

将数据复制到内核空间的 BO 后，接下来就需要通知 GPU 来读取这些数据，并执行 GPU 中的 Pipeline。以 Intel GPU 为例，其规定需要将批量数据组织到一个环形缓冲区中，然后 GPU 从环形缓冲区中读取并执行命令，如图 8-9 所示。

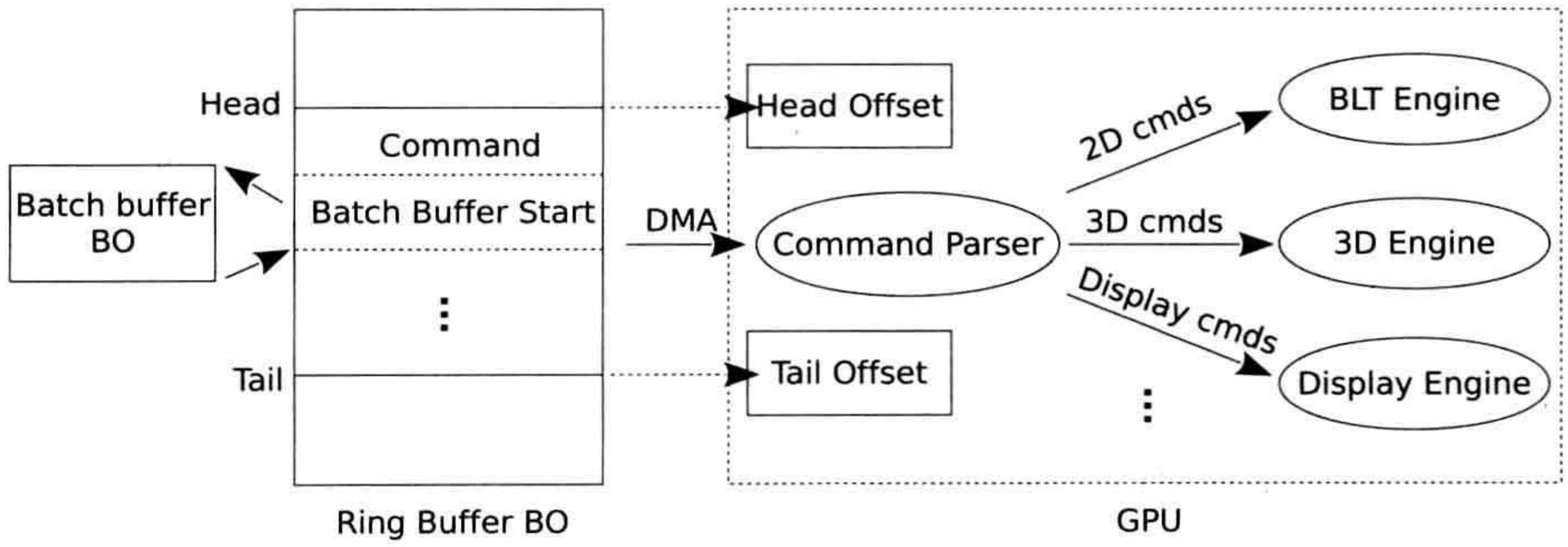


图 8-9 GPU 命令流

环形缓冲区也只是从内存中分配的一块用于显存的普通存储区，所以，当内核中的 DRM 模块组织好其中的数据后，GPU 并不会自动到环形缓冲区中读取数据，而是需要通知 GPU 来读取。

那么内核如何通知 GPU 呢？熟悉驱动开发的读者应该比较容易猜到，方法之一就是直接写 GPU 的寄存器。Intel GPU 为环形缓冲区设计了专门的寄存器，典型的包括 Head Offset、Tail Offset 等。其中寄存器 Head Offset 中记录环形缓存区中有效数据的起始位置，寄存器 Tail Offset 中记录的则是环形缓存区中有效数据的结束位置。

一旦内核中的 DRM 模块向寄存器 Tail Offset 中写入数据，GPU 就将对比寄存器 Head Offset 和 Tail Offset 中的值。如果这两个寄存器中的值不相等，那么就说明环形缓冲区中已经存在有效的命令了，GPU 中的命令解析单元（Command Parser）通过 DMA 的方式直接从环形缓冲区中读取命令，并根据命令的类型，定向给不同的处理引擎。如果是 3D 命令，则

转发给 GPU 中的 3D 引擎；如果是 2D 命令，则转发给 GPU 中的 BLT 引擎；如果是控制显示的，则转发给 Display 引擎；等等。

理解了相关原理后，下面我们就来看看 DRM 中具体的实现。在函数 drm_intel_bo_subdata 将数据复制到内核空间的 BO 后，do_flush_locked 调用了函数 drm_intel_bo_mrb_exec 向内核 DRM 模块发送命令 DRM_IOCTL_I915_GEM_EXECBUFFER 或者 DRM_IOCTL_I915_GEM_EXECBUFFER2（依据 GPU 的具体情况）。以 DRM 模块中处理命令 DRM_IOCTL_I915_GEM_EXECBUFFER2 的函数 i915_gem_execbuffer2 为例，组织并启动 GPU 读取环形缓冲区的相关代码如下：

```
linux-3.7.4/drivers/gpu/drm/i915/i915_gem_execbuffer.c:

int i915_gem_execbuffer2(...)
{
    ...
    ret = i915_gem_do_execbuffer(...);
    ...
}

static int i915_gem_do_execbuffer(...)
{
    ...
        ret = ring->dispatch_execbuffer(ring, ...);
    ...
}

linux-3.7.4/drivers/gpu/drm/i915/intel_ringbuffer.c:

static int i915_dispatch_execbuffer(...)
{
    ...
    intel_ring_emit(ring, MI_BATCH_BUFFER_START | MI_BATCH_GTT);
    intel_ring_emit(ring, offset | MI_BATCH_NON_SECURE);
    intel_ring_advance(ring);
    ...
}
```

注意函数 i915_dispatch_execbuffer 中的函数 intel_ring_emit，读者一定想到了组织批量缓冲的宏 OUT_BATCH 的定义，没错，这里就是在填充环形缓冲区。

在组织好环形缓冲后，i915_dispatch_execbuffer 调用了函数 intel_ring_advance 扣动了 GPU 的扳机，相关代码如下：

```
linux-3.7.4/drivers/gpu/drm/i915/intel_ringbuffer.c:

void intel_ring_advance(struct intel_ring_buffer *ring)
{
    ...
    ring->write_tail(ring, ring->tail);
}
```

```
static void ring_write_tail(struct intel_ring_buffer *ring,
                u32 value)
{
    ...
    I915_WRITE_TAIL(ring, value);
}

linux-3.7.4/drivers/gpu/drm/i915/intel_ringbuffer.h:

#define I915_WRITE_TAIL(ring, val) \
        I915_WRITE(RING_TAIL((ring)->mmio_base), val)
```

以 i915 系列为例，GPU 的相应寄存器在 CPU 地址空间中占据的地址如下：

```
linux-3.7.4/drivers/gpu/drm/i915/i915_reg.h:

#define RENDER_RING_BASE    0x02000
#define RING_TAIL(base)     ((base)+0x30)
#define RING_HEAD(base)     ((base)+0x34)
```

根据 Intel 的 GPU 的手册，地址“0x02000 + 0x30”恰恰就是 GPU 的寄存器 Tail Offset 在 CPU 的地址空间中分配的地址。

根据上述分析可见，内核的 DRM 模块通过写 GPU 的寄存器 Tail Offset 启动了 GPU 中的 Pipeline。最后 Pipeline 会将生成的图像的像素阵列输出到后缓冲的 BO。

8.4.3 交换前缓冲和后缓冲

应用程序绘制完成后，需要将后缓冲交换（swap）到前缓冲，其中有三个问题需要考虑。

（1）谁来负责交换

如果应用自己负责将后缓冲更新到前缓冲，那么当有多个应用同时更新前缓冲时如何协调？显然将交换动作交给更擅长窗口管理的 X 服务器统一协调更为合理。

如果 X 服务器开启了复合扩展，更需要知道应用已经更新前缓冲了，因为 X 服务器需要通知复合管理器重新合成前缓冲。

综上，应该由 X 服务器来负责交换前后缓冲。

对于 GPU 支持交换的情况，X 服务器通过 2D 驱动请求 GPU 进行交换。否则 X 服务器只能将前缓冲和后缓冲的 BO 映射到用户空间，使用 CPU 逐位复制。

（2）交换的时机

与 2D 应用不同，3D 程序通常涉及复杂的动画和图像，如果显示控制器正在扫描前缓冲的同时，X 服务器更新了前缓冲，那么可能会导致屏幕出现撕裂（tearing）现象。所谓的撕裂就是指本应该分为两桢显示在屏幕上的图像同时显示在屏幕上，上半部分是一帧的上半部分，而下半部分是另外一帧的下半部分，情况严重的将导致屏幕出现闪烁（flicker）。

以一个刷新率为 60Hz 的显示器为例，显示控制器每隔 1/60 秒从前缓冲读取数据传给显示器。每开始新的一帧扫描时，显示控制器都从前缓冲的最左上角的点，即第一行的第一个

点开始，逐行进行扫描，直到扫描到图像右下角的点，即最后一行的最后一个点。经过这样一个过程之后，就完成了一帧图像的扫描。然后显示控制器回溯（retrace）到第一行的第一个点的位置，等待下一帧扫描开始，如图 8-10 所示。

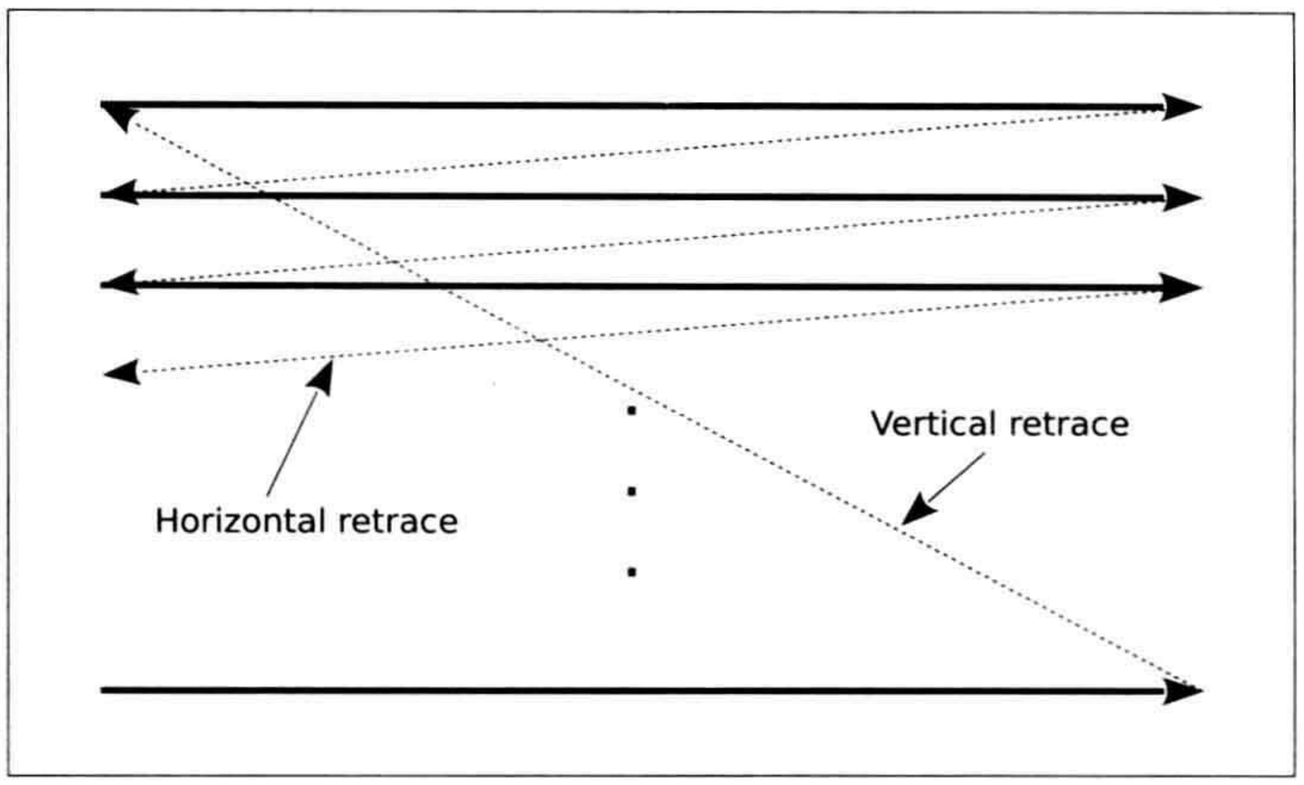


图 8-10　图像扫描示意图

更新一帧图像远不需要 1/60 秒，从更新完最后一行的最右侧一个点，到开始扫描下一帧之间的间隙被称为垂直空闲（vertical blank），简称为“vblank”。显然，如果在 vblank 这段时间更新前缓冲，就不会导致上述撕裂和闪烁现象的出现了。

（3）交换的方法

交换后缓冲和前缓冲通常有两种方法：第一是复制，在绘制完成后，X 服务器将后缓冲中的数据复制到前缓冲，如图 8-11 所示。

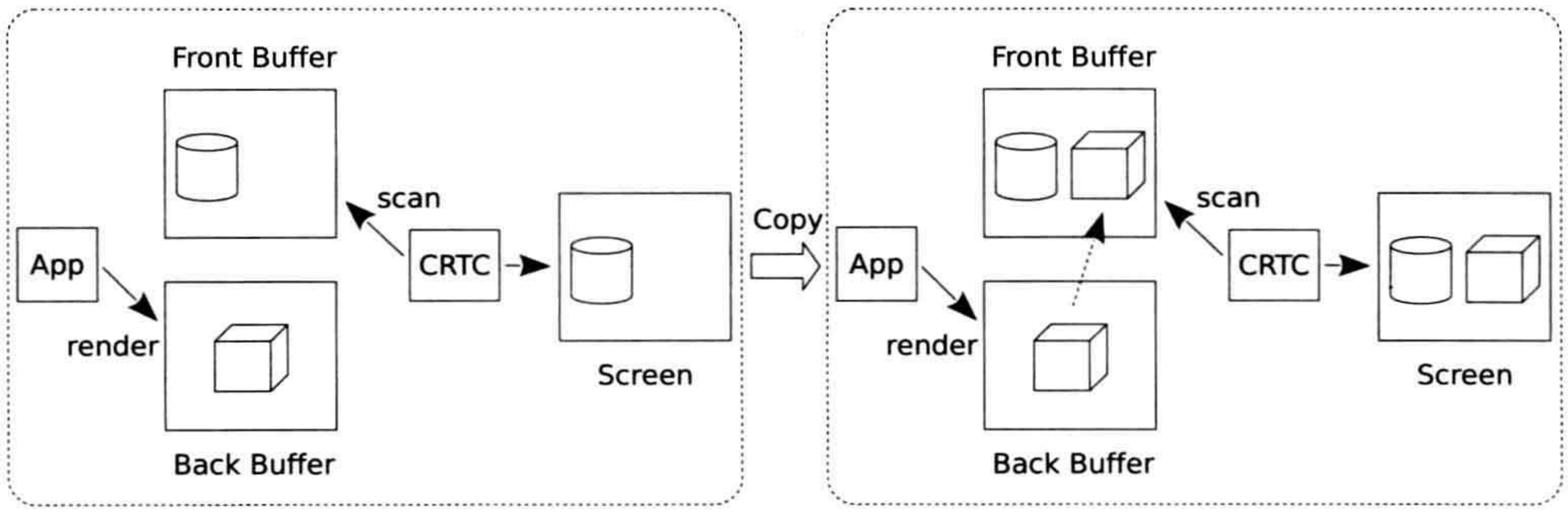


图 8-11　复制模式

但是这种方法效率相对较低，所以开发者们设计了页翻转模式（page flip）。页翻转模式不进行数据复制，而是将显示控制器指向后缓冲。后缓冲与前缓冲的角色进行互换，后缓冲摇身一变成为前缓冲，显示控制器将扫描后缓冲的数据到屏幕，而原来的前缓冲则变成了后缓冲，应用程序在前缓冲上进行绘制，如图 8-12 所示。

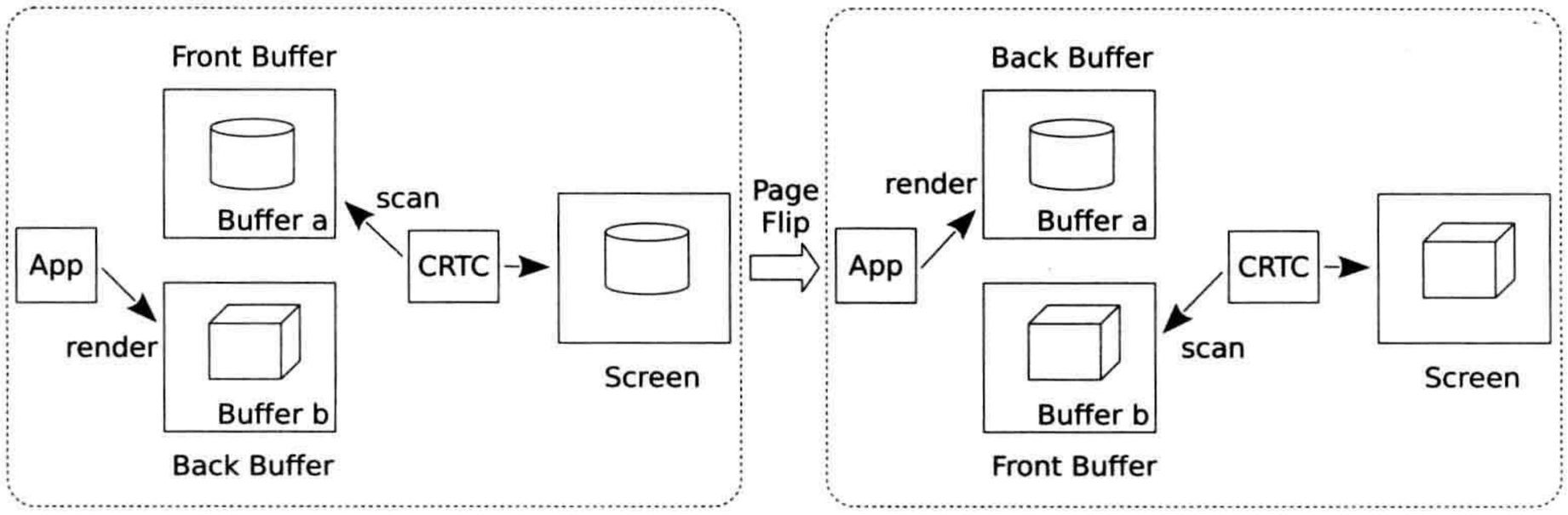


图 8-12 页翻转模式

页翻转模式虽然效率高，但也不是所有的情况都适用。典型的，当一个应用处于全屏模式时，可以采用页翻转模式互换前缓冲和后缓冲。但是这对于使用复合管理器的图形系统来说，其实已经大大的提升效率了，因为复合管理器控制着整个屏幕的显示，所以复合管理器可以使用页翻转模式交换前缓冲和后缓冲。

1. 应用发送交换请求

对于一个 OpenGL 程序来说，在绘制完成后，需要调用 GLX 扩展中的函数 glXSwapBuffers 向 X 服务器发出交换请求：

Mesa-8.0.3/src/glx/glxcmds.c:

```
_X_EXPORT void glXSwapBuffers(Display * dpy, GLXDrawable drawable)
{
    ...
            glFlush();
    ...
          (*pdraw->psc->driScreen->swapBuffers)(pdraw, 0, 0, 0);
    ...
}
```

glXSwapBuffers 首先调用 glFlush 启动 Pipeline 进行渲染。然后调用 DRI2 扩展的指针 swapBuffers 指向的函数向 X 服务器发出交换请求。DRI2 扩展中指针 swapBuffers 指向的函数是 DRI2SwapBuffers：

Mesa-8.0.3/src/glx/dri2.c:

```
void DRI2SwapBuffers(...)
{
    ...
    GetReq(DRI2SwapBuffers, req);
    req->reqType = info->codes->major_opcode;
    req->dri2ReqType = X_DRI2SwapBuffers;
    req->drawable = drawable;
    load_swap_req(req, target_msc, divisor, remainder);

    _XReply(dpy, (xReply *)&rep, 0, xFalse);
```

```
    ...
}
```

函数 DRI2SwapBuffers 创建了一个类型为 X_DRI2SwapBuffers 的 X 请求，然后调用函数 _XReply 将这个请求发送给 X 服务器。

2. X 服务器处理交换请求

X 服务器中处理来自 OpenGL 应用的请求在 DRI/GLX 的扩展模块中，对应的函数是 DRI2SwapBuffers：

```
xorg-server-1.12.2/hw/xfree86/dri2/dri2.c:

int DRI2SwapBuffers(...)
{
    ...
    for (i = 0; i < pPriv->bufferCount; i++) {
        if (pPriv->buffers[i]->attachment == DRI2BufferFrontLeft)
            pDestBuffer = (DRI2BufferPtr) pPriv->buffers[i];
        if (pPriv->buffers[i]->attachment == DRI2BufferBackLeft)
            pSrcBuffer = (DRI2BufferPtr) pPriv->buffers[i];
    }
    ...
    ret = (*ds->ScheduleSwap) (client, pDraw, pDestBuffer,
       pSrcBuffer, swap_target, divisor, remainder, func, data);
    ...
}
```

函数 DRI2SwapBuffers 首先获取请求更新的窗口的前缓冲和后缓冲。X 服务器在前面创建帧缓冲时已经将各个缓冲记录到了各个窗口中，所以这里取出即可。其中，pDestBuffer 指向前缓冲，pSrcBuffer 指向后缓冲。取得前缓冲和后缓冲后，具体的交换动作显然需要 2D 驱动来完成。DRI2SwapBuffers 调用 2D 驱动中的函数 ScheduleSwap 交换后缓冲和前缓冲。

在 Intel GPU 的 2D 驱动中，函数指针 ScheduleSwap 指向函数 I830DRI2ScheduleSwap：

```
xf86-video-intel-2.19.0/src/intel_dri.c:

01 static int I830DRI2ScheduleSwap(...)
02 {
03     ...
04     drmVBlank vbl;
05     ...
06     DRI2FrameEventPtr swap_info = NULL;
07     enum DRI2FrameEventType swap_type = DRI2_SWAP;
08     ...
09     if (can_exchange(draw, front, back)) {
10         swap_type = DRI2_FLIP;
11         flip = 1;
12     }
13
14     swap_info->type = swap_type;
15     ...
16     vbl.request.signal = (unsigned long)swap_info;
17     ret = drmWaitVBlank(intel->drmSubFD, &vbl);
```

```
18      ...
19 }
```

前面谈到 X 服务器应该在 vblank 时更新前缓冲，实现中也确实如此。I830DRI2ScheduleSwap 没有直接进行交换，而是调用库 libdrm 中的函数 drmWaitVBlank，这个函数告诉显示控制器，在 vblank 时，向内核发送 vblank 事件，如第 17 行代码所示。

函数 I830DRI2ScheduleSwap 需要做的另外一件事就是判断前缓冲和后缓冲的交换方式。默认的交换方式是复制，如第 7 行代码所示。第 9~12 行代码调用函数 can_exchange 来判断是否可以使用更高效的页翻转方式。

Intel GPU 的 2D 驱动在初始化时注册 vblank 事件的回调函数是 intel_vblank_handler：

```
xf86-video-intel-2.19.0/src/intel_display.c:

static void intel_vblank_handler(...)
{
    I830DRI2FrameEventHandler(frame, tv_sec, tv_usec, event);
}

xf86-video-intel-2.19.0/src/intel_dri.c:

void I830DRI2FrameEventHandler(..., DRI2FrameEventPtr swap_info)
{
    ...
    switch (swap_info->type) {
    case DRI2_FLIP:
        /* If we can still flip... */
        if (can_exchange(drawable, swap_info->front,
                swap_info->back) &&
                I830DRI2ScheduleFlip(intel, drawable, swap_info))
            return;

        /* else fall through to exchange/blit */
    case DRI2_SWAP: {
        ...
        I830DRI2CopyRegion(drawable, &region, swap_info->front,
                                swap_info->back);
        ...
    }
    ...
}
```

收到 vblank 事件后，函数 I830DRI2FrameEventHandler 首先判断等待 vblank 的交换请求希望使用的是页翻转模式还是复制模式。如果是页翻转模式，为了安全起见，再次使用函数 can_exchange 检查是否可以进行页翻转，确认没有问题后，则调用函数 I830DRI2ScheduleFlip 执行翻转。否则，则调用函数 I830DRI2CopyRegion 将后缓冲的内容复制到前缓冲。

（1）页翻转模式

进行页翻转的函数 I830DRI2ScheduleFlip 的相关代码如下：

```
xf86-video-intel-2.19.0/src/intel_dri.c:
```

```
static Bool I830DRI2ScheduleFlip(...)
{
    ...
        if (!intel_do_pageflip(intel, ...))
    ...
        I830DRI2ExchangeBuffers(intel, info->front, info->back);
    ...
}
```

I830DRI2ScheduleFlip 调用 2D 驱动中的函数 intel_do_pageflip 进行翻转。当然翻转后需要更新状态，包括更新当 Screen Pixmap 对应的 BO，这就是函数 I830DRI2ScheduleFlip 调用 I830DRI2ExchangeBuffers 的目的。2D 驱动中函数 intel_do_pageflip 的代码如下：

```
xf86-video-intel-2.19.0/src/intel_display.c:

Bool intel_do_pageflip(...)
{
    ...
        if (drmModePageFlip(...)) {
    ...
}
```

函数 intel_do_pageflip 并没有使用库 libdrm 提供的接口，如 drmModeSetCrtc 设置显示控制器扫描的缓冲，而是使用了接口 drmModePageFlip。相比于有点莽撞的 drmModeSetCrtc，函数 drmModePageFlip 能确保是在发生 vblank 时设置显示控制器扫描的缓冲。drmModePageFlip 将翻转的动作排队到下一个 vblank 事件发生时的处理队列中，在下个 vblank 发生时，设置显示控制器扫描的缓冲。

（2）复制模式

处理复制模式的函数 I830DRI2CopyRegion 的代码如下：

```
xf86-video-intel-2.19.0/src/intel_dri.c:

static void I830DRI2CopyRegion(DrawablePtr drawable, ...)
{
    ...
        gc->ops->CopyArea(src, dst, gc, ...);
    ...
}
```

看到 ops，读者一定非常熟悉了，没错，这就是我们前面讨论 2D 渲染时提及的画笔。在 UXA（uxa_ops）中，CopyArea 对应的函数是 intel_uxa_copy：

```
xf86-video-intel-2.19.0/src/intel_uxa.c:

01 static void intel_uxa_copy(...)
02 {
03     ...
04     {
05         BEGIN_BATCH_BLT(8);
```

```
06
07          cmd = XY_SRC_COPY_BLT_CMD;
08          ...
09          OUT_BATCH(cmd);
10
11          OUT_BATCH(intel->BR[13] | dst_pitch);
12          OUT_BATCH((dst_y1 << 16) | (dst_x1 & 0xffff));
13          OUT_BATCH((dst_y2 << 16) | (dst_x2 & 0xffff));
14          OUT_RELOC_PIXMAP_FENCED(dest, ...);
15          OUT_BATCH((src_y1 << 16) | (src_x1 & 0xffff));
16          OUT_BATCH(src_pitch);
17          OUT_RELOC_PIXMAP_FENCED(intel->render_source, ...);
18
19          ADVANCE_BATCH();
20      }
21 }
```

看到函数 intel_uxa_copy 的内容是否似曾相识？没错，指令 XY_SRC_COPY_BLT 与 8.3.2 节讨论的指令 XY_COLOR_BLT 非常相似，最大的不同是多了复制的源的信息。Intel GPU 的指令 XY_SRC_COPY_BLT 的格式如表 8-2 所示。

表 8-2 Intel GPU 指令 XY_SRC_COPY_BLT 的格式

双字（寄存器）	位	描 述
0 = BR00	31:29	02h – BLT 引擎
	28:22	指令操作码 (Opcode): 53h
	…	…
1 = BR13	25:24	色深 : 00 = 8 位 ; 01 = 16 位 ; …
	15:00	目标图像跨度
	…	…
2 = BR22	31:16	目标区域顶部坐标（Y1）
	15:00	目标区域左侧坐标（X1）
3 = BR23	31:16	目标区域底部坐标（Y2）
	15:00	目标区域右侧坐标（X2）
4 = BR09	31:00	目标区域基址
5 = BR26	31:16	源区域顶部坐标（Y1）
	15:00	源区域左侧坐标（X1）
6 = BR11	31:16	保留
	15:00	源区域图像跨度
7 = BR12	31:00	源区域基址

下面我们结合表 8-2 来分析函数 intel_uxa_copy 为 GPU 组织批量缓冲的过程。

1）第 9 行代码填充的是第 0 个双字，即 BLT 引擎的寄存器 BR00。这个寄存器中最重要的就是指令的操作码（Opcode），即第 22~28 位。对于指令 XY_SRC_COPY_BLT，其操作码是 0x53。观察宏 XY_SRC_COPY_BLT_CMD 的定义：

```
xf86-video-intel-2.19.0/src/i830_reg.h:

#define XY_SRC_COPY_BLT_CMD      ((2<<29)|(0x53<<22)|6)
```

其中从第 22 位开始的 0x53 正是指令 XY_SRC_COPY_BLT 的指令码。另外，第 29~30 位设置为 2，告诉 GPU 这个指令是一个 2D 指令，需要 GPU 定向给 BLT 引擎。

2）第 11 行代码填充的是第 1 个双字，对应 BLT 引擎的寄存器 BR13，其中“intel->BR[13]”在 8.3.2 节我们已经讨论过，表示色深。另外，dst_pitch 表示目标区域的跨度，所谓的跨度就是以字节为单位的图形的宽度。

3）第 12 行代码填充了第 2 个双字，对应 BLT 引擎的寄存器 BR22，这个寄存器中保存的是目标区域的左上角的坐标。

4）第 13 行代码填充了第 3 个双字，对应 BLT 引擎的寄存器 BR23，这个寄存器中保存的是目标区域的右下角的坐标。

5）第 14 行代码填充了第 4 个双字，对应 BLT 引擎的寄存器 BR09，这个寄存器中保存的是存储目标区域像素阵列的 BO，当然使用的是 BO 在 GPU 虚拟地址空间的地址，即 BO 的 offset。

6）第 15 行代码填充了第 5 个双字，对应 BLT 引擎的寄存器 BR26，这个寄存器中保存的是源区域的左上角的坐标。

7）第 16 行代码填充了第 6 个双字，对应 BLT 引擎的寄存器 BR11，这个寄存器中保存的是源区域的图形的跨度。

8）第 17 行代码填充了第 7 个双字，对应 BLT 引擎的寄存器 BR12，这个寄存器中保存的是存储源区域的像素阵列的 BO 的地址。

8.5 Wayland

将所有图形全部交由 X 服务器绘制的这种设计，在以 2D 应用为主的时代，一切还相安无事。但是随着基于 3D 的应用越来越多，效率问题逐渐凸显出来。与 2D 程序不同，3D 程序的数据量要大得多，所以应用与 X 服务器之间需要传递大量的数据。设想一下几个人过独木桥和万人争过独木桥的场景，显然，X 曾经引以为傲的设计——通过网络通信的客户 / 服务器架构，成为性能的瓶颈。

为了解决这个问题，X 的开发者们设计了 DRI 机制，即应用程序不再将绘制图形的请求发送给 X 服务器，而是由应用程序自行绘制。这种设计与 X 最初的设计原则虽然有些格格不入，但是从某种程度上确实缓解了 3D 应用的效率问题。

但是，好景不长，人们逐渐不再满足于看上去比较“呆板”的图形用户界面，人们追求具有更华丽的 3D 特效的图形用户界面，比如窗口弹出和关闭时的放大 / 缩小动画、窗口之间的透明等。于是开发者们为 X 设计了复合（Composite）扩展，并仿效窗口管理器设计了

一个所谓的复合管理器（Composite Manager）来实现这些效果。

我们以 2D 绘制过程为例来简要地看一下什么是复合扩展以及复合管理器，如图 8-13 所示。

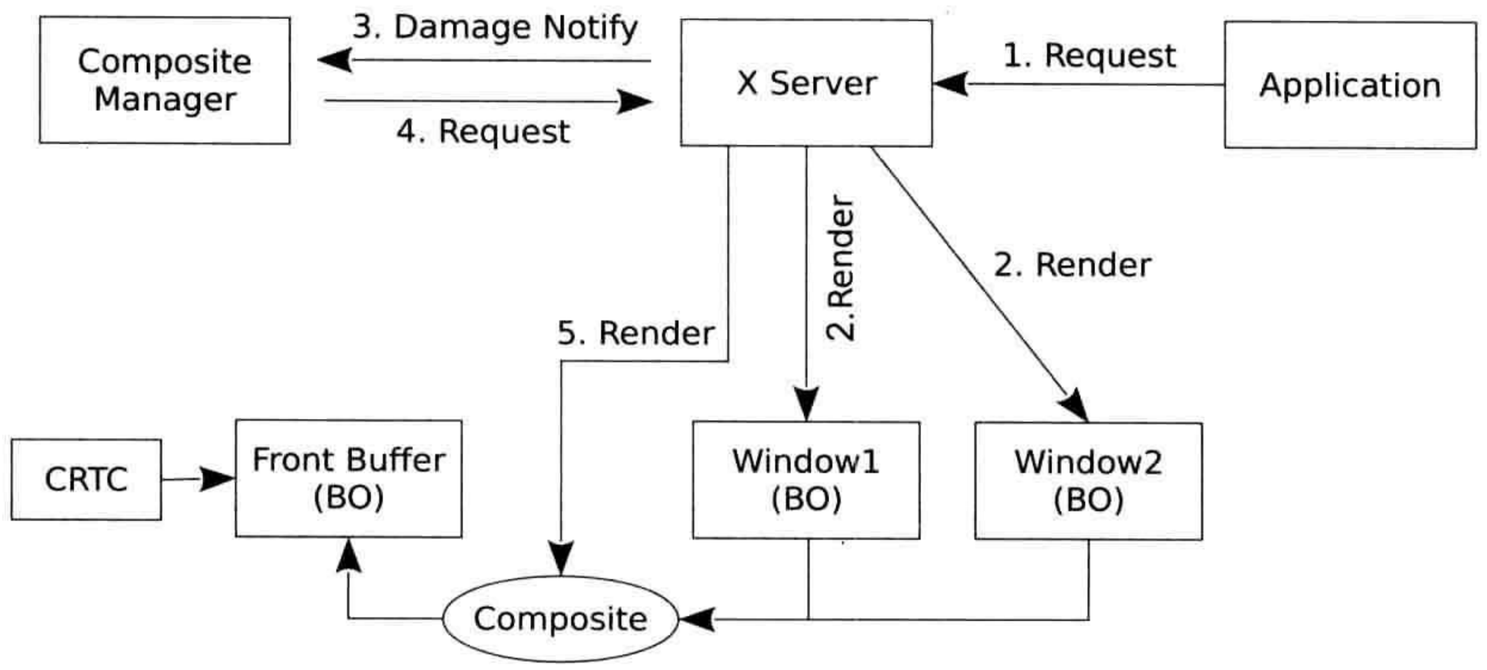


图 8-13 复合扩展

开启复合扩展后，最大的一个区别是所有的窗口都不再共享一个前缓冲，而是有了各自的离屏区域。X 服务器在各个窗口的离屏区域上进行绘制。在绘制好后，X 服务器向另外一个特殊的应用复合管理器（Composite Manager）发出 Damage 通知。然后由复合管理器请求 X 服务器对这些离屏的窗口的缓冲区进行合成，最后请求 X 服务器显示到前缓冲。

下面的代码片段展示了复合管理为窗口创建离屏缓冲的过程：

```
xorg-server-1.12.2/composite/compinit.c:

Bool compScreenInit(ScreenPtr pScreen)
{
    ...
    pScreen->CreateWindow = compCreateWindow;
    ...
}

xorg-server-1.12.2/composite/compwindow.c:

Bool compCreateWindow(WindowPtr pWin)
{
    ...
            compRedirectWindow( ... );
    ...
}
```

我们看到，在开启复合扩展后，屏幕中的指针 CreateWindow 已经指向了复合扩展中实现的函数 compCreateWindow。而在函数 compCreateWindow 中，其使用函数 compRedirectWindow 将窗口从前缓冲重定向到一个离屏区域。

在这个复合过程中，就是制造那些绚丽效果的地方。比如在合成的过程中，我们使用如图 8-14 的方法，就可以使窗口看起来是以放大效果出现的。

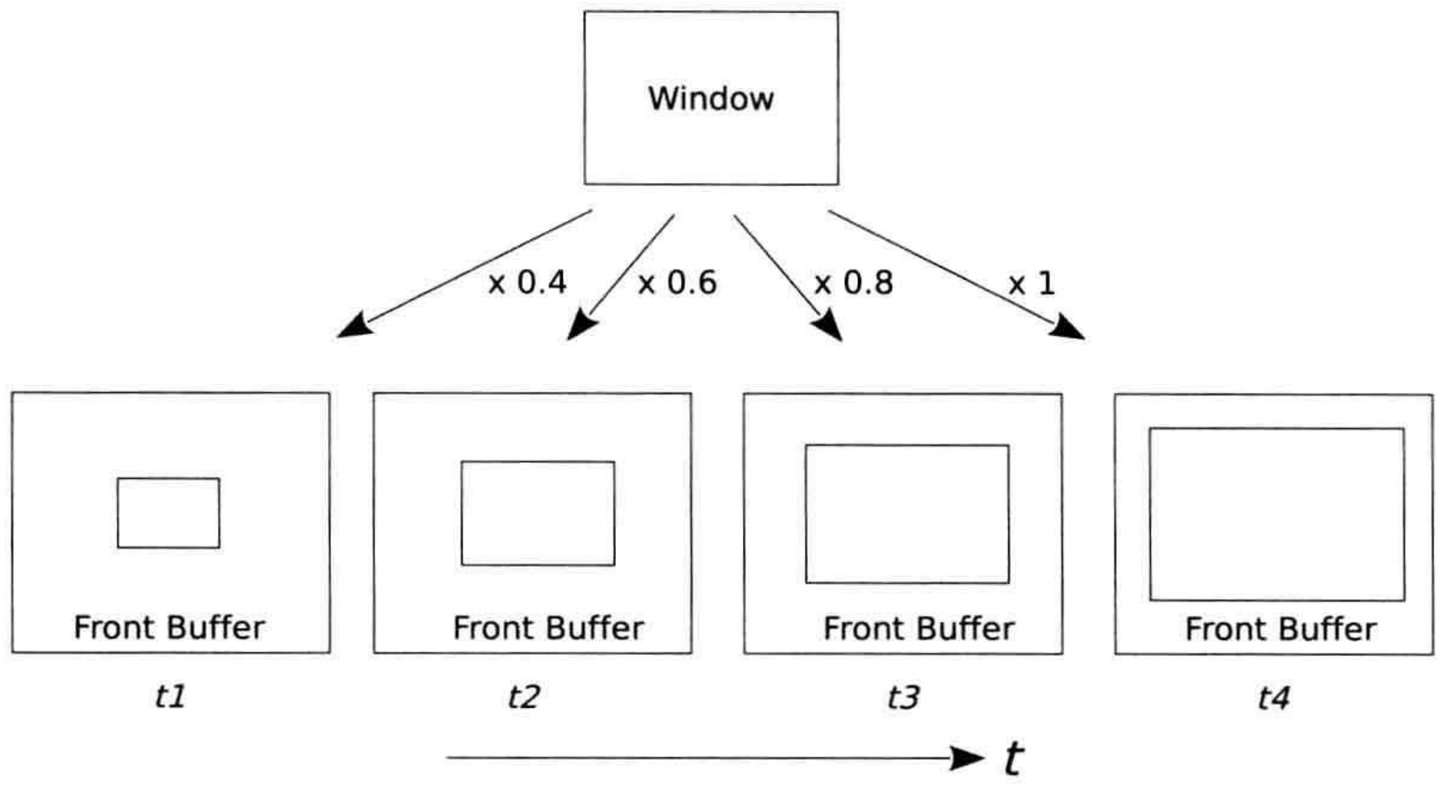


图 8-14　以放大效果出现的窗口

使用复合管理后，绚丽的效果有了，但是仔细观察图 8-13 会发现，X 被人诟病的基于网络通信的客户 / 服务器模式的问题又变得严重了。除了 X 服务器和应用之间的通信外，为了进行合成，X 服务器和复合管理器之间又多了一层通信关系。

事实上，在 DRI 的演进过程中，X 不断被拆分和瘦身，开发者从 X 中移除了大量与渲染有关的功能到内核和各种程序库中。慢慢的，人们发现，X 所做的事情已经大为减少，替代 X 已经不是一项不可能的任务。于是一部分开发者开始尝试为 Linux 开发替代 X 的窗口系统，Kristian Høgsberg 提出了 Wayland。事实上，这一个过程迟早要发生的，即使不是 Wayland，也会涌现出如 Yayland、Zayland 等。

Wayland 并不是一个全新的事物，它是站在 X 这个巨人的肩膀上，在 X 的不断演进中进化而来的。虽然从名字上看，Wayland 与 X 没有丝毫相干，但是实际上两者的联系可谓千丝万缕。Wayland 的开发者 Kristian Høgsberg 曾经是 X 的 DRI 的主要开发者之一。套用一句奔驰的广告语，“经典是对经典的继承，经典是对经典的背叛”，Wayland 去掉了 X 的客户 / 服务器架构，但是继承了 X 为提高绘制效率不懈努力的成果：DRI。除了逻辑上设计上不同外，Wayland 基本的渲染原理与我们前面讨论的 2D 和 3D 的渲染原理完全相同。基本上，基于 Wayland 的图形架构如图 8-15 所示。

Wayland 本身是一个协议，其具体的实现包括一个合成器（Compositor）以及一套协议实现库。当然，图形库为了与合成器进行通信，在图形库中需要加入 Wayland 协议的相关模块，也就是图 8-15 中的 Wayland backend 部分，当然这些都可以基于 Wayland 提供的库，而不必从头再将 wayland 协议实现一遍。

在 Wayland 下，所有的图形绘制完全由应用自己负责。其绘制过程与我们前面讨论的 2D 和 3D 的绘制过程完全相同，只不过 2D 的绘制部分也搬到图形库中了，绘制动作与合成器没有丝毫关系。而在绘制后，应用将前缓冲和后缓冲进行对调，并向合成器发送 Damage 通知，当然颜色缓冲不一定是前后两个，在具体实现中，有的图形系统可能使用 3 个、4 个

甚至更多。在收到 Damage 通知后，合成器将应用的前缓冲合成到自己的后缓冲中。而合成器的这个合成过程，与普通应用的绘制过程并无本质区别，也是通过图形库完成。

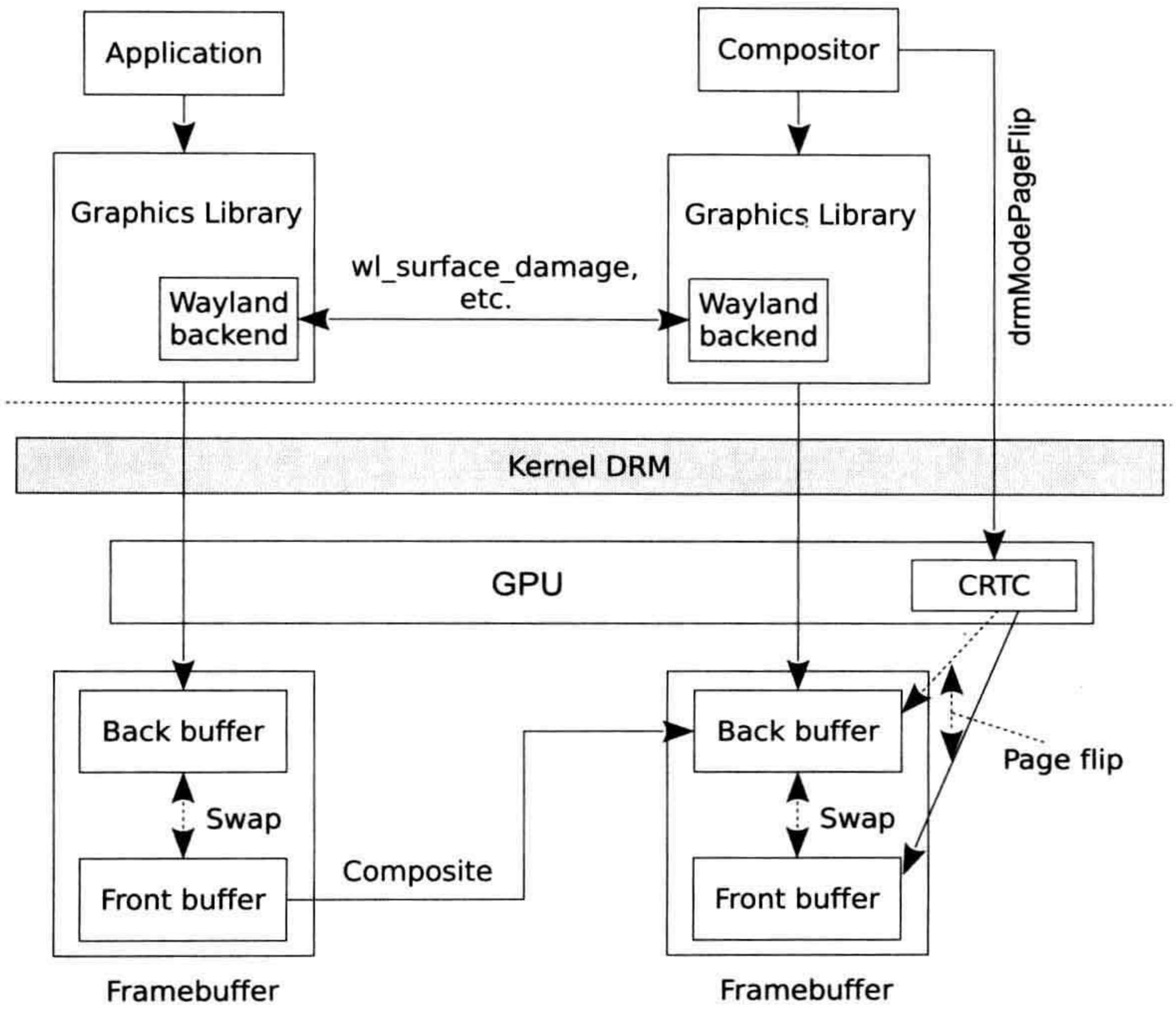


图 8-15　Wayland 体系架构

在合成完成后，合成器对调后缓冲与前缓冲，并设置显示控制器指向新的前缓冲，即原来的后缓冲。此前的前缓冲作为新的后缓冲，并作为合成器下一次合成的现场；而原来的后缓冲则变成现在的前缓冲，用于显示控制器的扫描输出。