

*Embedded Android*



# 构建嵌入式 Android 系统

O'REILLY®  
中国电力出版社

Karim Yaghmour 著  
秦云川 肖淇 译

# 构建嵌入式Android系统

你想要把Android移植到其他嵌入式系统平台上吗？本书将告诉你Android是如何工作的，以及如何修改它以满足你的需求。你将会深入到Android的内部结构，并且学会如何阅读它的源码、修改它的各种组件、针对你的硬件设备创建你自己的Android版本。你会发现Android与它的源头Linux相比有多大的不同。

如果你是一个有经验的嵌入式系统开发人员并且熟悉Linux系统，那么本书将会帮你把Android看作是一个硬件平台，而不是仅仅是一个移动终端。

- 学习Android的开发模式和你需要用来运行Android的硬件。
- Android内部机制的快速入门，包括Linux内核和Dalvik虚拟机。
- 在没有硬件的情况下，通过模拟器镜像来开始学习和探索Android。
- 了解Android的无递归的构建系统，以及学习怎样来做你自定义的修改。
- 使用评估板来构建你的嵌入式Android的原型系统。
- 了解Android的本地用户空间，包括根文件系统的布局、adb工具，以及Android的命令行。
- 了解如何与Android框架进行交互或进行定制。

“这本书对所有想要基于Android构建系统的人来说是一本权威书籍。如果你不是在谷歌工作，但却使用最底层Android接口的话，这本书就很适合你。”

——Greg Kroah-Hartman  
Linux内核核心开发者

“有了这本书，我可以在了解Android的来龙去脉时，节约好几个月的时间。毫无疑问，在未来几年，这本书对于Android系统开发人员来说都是一本典型的参考书。”

——Tim Bird  
索尼网络娱乐的高级工程师

Karim Yaghmour是Opersys公司的CEO，该公司提供嵌入式Android和嵌入式Linux上的开发和培训服务。他是O'Reilly公司出版的《构建嵌入式Linux系统》一书的作者，Karim的文章已经提交并发表在同行评议的具有科学性的行业会议、杂志和在线出版物上。



**O'REILLY®**  
[oreilly.com.cn](http://oreilly.com.cn)

O'Reilly Media, Inc.授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行  
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



---

# 构建嵌入式 Android 系统

*Karim Yaghmour* 著  
秦云川 肖淇 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

## 图书在版编目 (CIP) 数据

构建嵌入式 Android 系统 / (美) 亚荷毛尔 (Yaghmour, K.) 著; 秦云川, 肖淇译.  
—北京: 中国电力出版社, 2015.6

书名原文: Embedded Android

ISBN 978-7-5123-7372-3

I. ①构… II. ①亚… ②秦… ③肖… III. ①移动终端—应用程序—程序设计  
IV. ① TN929.53

中国版本图书馆 CIP 数据核字 (2015) 第 050072 号

北京市版权局著作权合同登记

图字: 01-2014-5258 号

©2013 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2015.  
Authorized translation of the English edition, 2013 O'Reilly Media, Inc., the owner of all rights to publish and  
sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2013。

简体中文版由中国电力出版社出版 2015。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

封面设计 / Randy Comer, 张健

出版发行 / 中国电力出版社 (<http://www.cepp.sgcc.com.cn>)

地 址 / 北京市东城区北京站西街 19 号 (邮政编码 100005)

经 销 / 全国新华书店

印 刷 / 北京丰源印刷厂

开 本 / 787 毫米 × 980 毫米 16 开本 23.75 印张 452 千字

版 次 / 2015 年 6 月第一版 2015 年 6 月第一次印刷

印 数 / 0001 — 3000 册

定 价 / 68.00 元 (册)

## 敬 告 读 者

本书封底贴有防伪标签，刮开涂层可查询真伪

本书如有印装质量问题，我社发行部负责退换

版 权 专 有 翻 印 必 究

## 对本书的赞誉

“对于想要创建一个基于 Android 系统的人来说，这是一本权威书籍。如果你没有在谷歌工作，并且正在使用底层的 Android 接口，那么你需要这本书。”

——Greg Kroah-Hartman, 核心的 Linux 内核开发者

“如果你或你的团队正在致力于创建定制的 Android 镜像、设备或 ROM，那么你一定会想要这本书！除了源代码本身，关于 Android 是如何工作的，Android 是如何构建系统的，以及有关 Android 是如何集成总体视图，你会发现本书是唯一解释这些问题的地方。我特别喜欢关于构建系统和 Android 框架的几个章节（4、6 和 7），那里有很多从 AOSP 源代码中挖掘的信息，这是很难进行反向工程的。本书将为你和你的团队节省大量的时间。我希望在我们的团队两年前开始研究冷冻酸奶版本的 Android 的时候，就能拥有它。对于工作在用于英特尔参考手机的 Intel Android 堆栈的新团队成员来说，这本书将会成为他们的必读物。”

——Mark Gross, Android/Linux 内核架构师,  
平台系统集成部 / 移动与通信事业群 / 英特尔公司

“Karim 有条不紊地写出了很多对嵌入式系统开发人员来说的 Android 的未解之谜。对于工作在所有类型的设备，不仅仅是定制手机和平板电脑的开源软件项目的人来说，本书绝对是一本实战宝典。我个人很高兴看到有这么多提供在经济实惠的硬件，即 BeagleBone 上的例子，而不仅仅只是在仿真器上。”

——Jason Kridner, Sitara 软件架构经理,  
德州仪器公司以及 BeagleBoard.org 的创始人之一

“这本书所包含的内容，是我的工程师以前需要数百个小时才能发现的信息。对于所有想要加入我的团队做关于 Android 方面的新成员来说，这本书是他们的必读物。”

——Mark Micire 博士，空间和移动机器人领域的研究人员，卡耐基梅隆大学

“多亏了这本书，从事嵌入式系统开发的开发人员第一次可以应用这个开放的垂直整合堆栈，它包含他们构建基于 Linux 的稳定、高效产品的所有东西。Android 革命性的执行模型跨越了手机和平板电脑，其应用程序开发平台在行业特点和发展速度上是无与伦比的。这本书将会给开发人员提供宝贵资源，用于了解应用层和内核的所有东西，以及告诉开发人员如何扩展和改变一些东西来定制各种各样的 Android。”

——Zach Pfeffer, Linaro 公司 Android 团队的技术主管

“最后，这是一本从系统的角度来介绍 Android 平台的书！目前有大量关于创建 Android 应用程序的书籍，但很久没有一本单一的、全面的资源来介绍 Android 的内部信息。在本书中，Karim 已经收集了大量的资料，对于 Android 系统的程序员和集成商（虽然，可以肯定的是，应用程序开发人员来阅读本书也将受益）来说，这些都是很有必要且有很大帮助的。Karim 从他关于 Android 的丰富的经验和分析中，收集了丰富的例子、引用和解释。希望我曾经在索尼工作的时候，在学习 Android 的艰辛之路上时就能够拥有这本书。有了这本书，我可以节省自己几个月用于学习 Android 的来龙去脉的时间。毫无疑问，这将是近几年来对 Android 系统开发人员来说重要的参考书。”

——Tim Bird, 索尼娱乐网络副主任工程师，  
Linux 基金会的 CE 工作组体系结构组主席

“Karim 的书是那些希望进入基于 Android 的嵌入式项目及产品新兴领域的人的优秀导游。本书涵盖了从内核支持直至许可和商标问题的全部范围，也包括在“无领导”模式下运行 Android 系统的信息。这本书值得在每一个严谨的嵌入式 Android 开发者的书架上占有一席之地。”

——Paul E. McKenney, IBM 的杰出工程师，Linux 内核 RCU 维护者

“虽然 Android 通常是为手机和平板电脑领域而设计的，但它毫无疑问地考虑到了很多其他的产品领域，如汽车、类似 HMI 的 UI 面板、可穿戴的小设备等。强烈推荐这本书，因为它涵盖了所有必要的基础知识和概念，帮助开发者接触和研发基于 Android 的针对移动和非移动产品部分的解决方案。”

——Khasim Syed Mohammed, 德州仪器首席工程师

“这是一本伟大的书，不仅仅对于嵌入式 Android 开发人员来说是这样，对于那些要学习 Java 接口以下布线的 Android 应用程序开发者来说也是如此。”

——Lars Vogel, Vogella GmbH 公司的 CEO

“再一次，Karim 击中了我们的要害。如果你有兴趣将 Android 移植到一个新的设备上或者只是对 Android 是如何运行在一个硬件上的关键部分有兴趣，这一本书就是你一直在寻找的书。这本书将引导你完成所有包括构建环境启动，获取 AOSP 资源，增加你的硬件到 Android 源代码中，以及在你的硬件上部署一个新的 Android 的方方面面。它讨论了 Android 的基础，包括 HAL 以及如何使你的定制硬件得到 Android 的框架的支持。总之，对于所有关于 Android 的书来说，这是一本针对 Android 设备制造商，而不是 Android 应用程序开发人员或最终用户的书。我只是希望在我第一次接触 Android 的移植的时候，这本书就已经面世了。这可能会节约我几个月的时间来进行试验和错误尝试。”

——Mike Anderson, PTR 集团公司首席科学家

“本书已经是我们公司的一个很好的资源。当我们移植 Android 到新的硬件或在一个较低的水平整合新的功能时，这是一本必备的书。Karim 是一个伟大的指导者，他的写作很好地体现了他的风格。”

——Jim Steele, 传感器平台的工程副总裁

“本书对于那些想要认真研究 Android 内部结构以及在一个新平台上构建 Android 的人来说，是一本不可不看的书。它有助于指导你有关扩展的 AOSP 代码库，并了解整体架构和系统的设计。”

——Balwinder Kaur, Aptina 图像的高级会员，技术人员

“所以，你以为你了解关于 Android 的内部结构？好了，再想想！逐章看完后，你会发现其幕后还有什么，以及为什么 Android 不仅仅只是一个嵌入式 Linux 发行版。让自己准备好进入一个漩涡，因为本书对于每一个希望成为厉害的谷歌操作系统的黑客的人来说，就是一座金矿。”

——Benjamin Zores, Alcatel-Lucent 公司的 Android 平台架构师

“这本书肯定是关于 Android 系统堆栈的最有价值和最完整的资源之一。对于每一个 Android 系统工程师来说是一本必读物。”

——Maxime Ripard, Free Electrons 公司的 Android 的领导者

“当我接到一块运行 Linux 的开发板，并且被告知‘在上面运行一个 Android’的时候，想要找到有关如何在一个新设备上移植 Android 有关的更多信息，这是非常困难的。幸运的是，当我开始研究这个问题的同时，本书面世了。什么是救星！本书给了我很多指导，即我需要了解 Android 的基础，需要做什么来将 Android 移植到一个新的硬件上去。我喜欢这本书所有的细节和背景，从引导序列到构建系统。在读完了本书之后，我觉得我对 Android 以及它是如何与 Linux 内核交互的有了更好的理解。”

——Casey Anderson, Trendrl 的嵌入式系统架构师

# O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

致Anais、Thomas和Vincent。

愿你们的人生旅程中充满了分享与发现的喜悦。

# 目录

前言 .....	1
<b>第1章 概述 .....</b>	<b>11</b>
历史 .....	11
特点和特征 .....	12
开发模型 .....	15
生态系统 .....	17
获取“Android” .....	19
法律框架 .....	20
硬件与合规性要求 .....	27
开发工具及其环境搭建 .....	31
<b>第2章 内部结构入门 .....</b>	<b>33</b>
应用程序开发者的观点 .....	33
整体架构 .....	41
Linux内核 .....	42
硬件支持 .....	53
原生用户空间程序 .....	59
Dalvik以及Android上的Java .....	67
系统服务 .....	70
常见的AOSP包 .....	77

<b>第3章 AOSP入门</b>	<b>84</b>
开发主机设置	84
下载AOSP	85
AOSP的内部	91
构建的基础知识	96
运行Android	104
使用Android调试工具（ADB）	106
掌握模拟器的使用	110
<b>第4章 构建系统</b>	<b>115</b>
与其他构建系统的比较	115
体系结构	117
构建脚本	138
基本的AOSP修改技巧	147
<b>第5章 硬件基础</b>	<b>158</b>
典型的系统架构	158
片上系统中有什么？	164
内存布局与映射	167
评估板	172
<b>第6章 本地用户空间</b>	<b>176</b>
文件系统	176
Android的命令行	209
初始化	229
<b>第7章 Android框架</b>	<b>250</b>
Framework入门	251
工具和命令	266
支持守护进程	296
硬件抽象层	303

附录A 传统的用户空间 .....	305
附录B 为新硬件增加支持 .....	320
附录C 默认包列表的定制 .....	332
附录D 默认的init.rc文件 .....	335
附录E 资源 .....	358

---

# 前言

Android 的增长是显然易见的。在一个很短的时间内，它已经成功地成为市场上最顶尖的移动平台之一。显然，开源许可、积极走向市场以及时尚界面的独特组合，是 Google 公司的 Android 团队的成果。毫无疑问，Android 产生了如此大的用户量，使得手机厂商、移动通信网络运营商、芯片厂商以及应用开发人员都不能无视它。基于 Android 以及与 Android 兼容的产品、应用和设备正以前所未有的速度繁荣起来。

除了其在移动领域的成功以外，Android 也同样引起了另一个群体的注意——嵌入式系统开发者。虽然大多数嵌入式系统设备很少甚至没有用户界面，但是还是有很多传统“嵌入式”设备确实是有用户界面的。现在有很多机器除了有纯粹的功能以外，开发者还要为用户操作提供人机交互界面。所以，设计人员要么为用户提供一种其熟悉的界面体验，要么冒险为用户提供一种不熟悉甚至是全新的界面风格。在 Android 以前，可供这种设备开发人员选择的用户界面非常有限。

很明显，嵌入式系统开发人员更希望为用户提供一种更熟悉的界面体验。虽然以前可能是基于窗口的风格，所以很多嵌入式设备采用类似桌面或者基于桌面窗口风格的用户接口，但现在苹果的 iOS 和 Google 的 Android 展示了一种基于触摸的类似 iPhone 的图形接口。这种界面风格的变化，以及 Android 开放源代码的授权，使得很多人希望将它应用于嵌入式系统中。

与 Android 应用程序开发者不同，任何希望对 Android 框架开展工作的开发人员，不管是嵌入式系统移植还是修改 Android 框架的工程师，都会发现根本没有任何文档来阐述应该如何开展工作。对于应用开发者来说，有 Google 提供的在线文档可供参考，有大量的书籍可供学习，例如 O'Reilly 的《Learning Android》，但是对嵌入式系统

开发者来说，只能参考 Google 在 <http://source.android.com> 上面的很少的一点文档。那些想在系统中使用 Android 的嵌入式系统开发者只能从阅读 Android 的源代码开始。

本书的目的是想改变这种情况，使得你能够将 Android 嵌入到你的任何设备中。所以，你将会学习 Android 架构、浏览源码和修改各组件的方法，你将明白如何为你的设备创建你自己的版本。而且，你还会知道 Android 是如何集成 Linux 内核，以及如何利用 Android 的 Linux 特性。例如，我们会讨论如何将 glibc 和 Busybox 等传统 Linux 组件打包到 Android 中。另外，你还会学习到很多日常使用的技巧，例如 Android 的 repo 工具以及集成 / 修改 Android 的构建系统。

## 初识 Android

我从 20 世纪 90 年代中期就开始接触开源软件，很幸运地在它成为一种强势的软件运动之前就已开始对它进行研究，并且见证了其在 2000 年左右的迅速崛起。我在开源社区中做出了自己的贡献，当然也不能免俗地参与到很多口水战中。我之前还写过《构建嵌入式 Linux 系统》（O'Reilly 公司出版）。

所以当这个基于 Linux 的 Android 系统开始流行的时候，基于我对 Linux 历史和嵌入式 Linux 系统的认识，我意识到它很值得我进行投入。当时，我曾天真地以为：“我对 Linux 系统非常了解，而 Android 系统又是基于 Linux 的，那么对我来说它能难到哪里去？”直到真正地深入去了解 Android 的时候发现我错了，Android 就是不一样。我所知道的那些 Linux 知识以及嵌入式系统中常用的那些软件包很少在 Android 中遇到，而且就连 Android 对硬件适配所采用的抽象方式都觉得很怪异。

我花了很多时间（而且现在还在追寻）来搞清楚这些问题。例如，Android 是如何工作的？它跟普通的 Linux 有什么不同？如何对它进行定制？如何把它应用于嵌入式系统？如何编译构建它？应用程序 API 是如何转换成我所熟知的 Linux 用户空间接口的？诸如此类。而且我发现对 Android 学习得越深入，问题就越多。

我学习 Android 系统的第一件事就是到 <http://developer.android.com> 和 <http://source.android.com> 去打印所有我可以找到的内容，留做开发者 API 参考手册，结果打印出来的纸堆起来差不多就有 8 到 10 英尺高。我通读了这些材料，对重要的段落做了大量标注，结果得到了一个我无法回答的问题列表。同时，我又开始研究 Android 开放源码项目（AOSP）给出的源代码。老实说，我花了差不多 6 ~ 12 个月才对 AOSP 源码结构有了感觉。

你看到的这本书可以看作是我在 Android 上所做工作的一个总结，包括我所参与的一些项目，例如帮助各种开发团队对他们的嵌入式系统设计定制 Android 系统。但是，这本书绝不是我对 Android 了解的全部。关于 Android 及其内部机制还有太多东西本书没有涉及，也无法全部涉及，不过它却可以是满足你的需要而对 Android 系统进行定制的一个开始。

## 本书的读者对象

本书主要针对想基于 Android 系统开发嵌入式系统的开发者，或者想把 Android 定制后用于特定用途的开发人员。本书假设你对嵌入式系统开发有所了解，并且至少知道 Linux 的基本工作机制以及命令行交互方式。

我并不假设你懂得 Java，并且对定制 Android 的大部分任务你根本无需要懂 Java。当然，当你的工作在 Android 环境中开展以后，你会发现一定程度地熟悉 Java 还是很必要的。实际上，大部分 Android 的核心部分是采用 Java 开发的，所以为了增加需要的功能，你可能需要学习这门语言。

本书不会教你如何进行 App 开发或者 Java 编程，如果你对这些话题感兴趣，我建议你找一下其他资料，这样的书籍已经有很多了。本书也不讲述嵌入式系统，这个题材的书也有很多了。最后，本书也不是关于嵌入式 Linux 的，这个题材也有关于它的很多书籍。不过，熟悉在嵌入式系统中使用 Linux 会有助于学习 Android。实际上，Android 很多东西是从传统的“嵌入式 Linux”中演化而来的，创建嵌入式 Linux 系统的很多技术能引导或者帮助你构建嵌入式 Android 系统。

如果你对于了解 Android 的内部结构也有兴趣的话，这本书也将对你有帮助。事实上，定制一个在嵌入式系统中使用的 Android，需要至少知道一些关于它的内部结构的基础知识。因此，尽管关于探讨解释 Android 的源代码的讨论没有走向深入，本书的解释还是深入地展示了 Android 栈的各个部分是如何交互的。

## 本书的组织结构

像许多其他的主题一样，这本书复杂性也是循序渐进的，以前面章节的内容作为之后章节的背景材料。如果你是一名管理者，只是想要抓住其中的要领，或者如果你想知道哪些章节是在你能够开始跳过章节及有选择性地阅读材料的章节前你要通读的，我建议你至少要读前三个章节。这并不意味着其他的都是不相关的，只是之后的内容更模块化。

第 1 章概述，涵盖了你应该知道的关于在嵌入式系统中 Android 的使用的一般内容，比如它来自哪里，它的发展模式是怎样的，它区别于传统的开源项目的许可，以及运行 Android 需要的硬件类型。

第 2 章内部结构入门，深入挖掘 Android 的内部结构以及阐述它主要的抽象结构。我们首先介绍了应用程序开发人员都习惯了的应用程序开发模型，然后介绍了 Android 专用的内核修改、怎么将硬件支持加到 Android 上、Android 本地用户空间、Dalvik、系统服务器，以及系统启动的整体概况。

第 3 章 AOSP 入门，说明了怎样从谷歌上获取 Android 的源代码，如何将其编译成一个功能性的仿真器镜像，以及如何运行该镜像并 shell 进去。使用仿真器是一种简单的探索 Android 的基础，而无需实际硬件的方式。

第 4 章构建系统，提供关于 Android 构建系统的详细说明。事实上，与大多数开源项目不同，Android 的构建系统是非递归的。本章介绍了 Android 构建系统的体系结构，它通常在 AOSP 内是如何使用的，以及如何将自己的修改添加到 AOSP 中。

第 5 章硬件基础，介绍了 Android 设计所针对的硬件类型。这覆盖了通常使用 Android 的系统级芯片（SoC），典型的 Android 系统内存布局，Android 使用的典型的开发环境设置，以及一些你可以很容易地使用于嵌入式 Android 系统原型上的开发评估板。

第 6 章本地用户空间，包含根文件系统布局、adb 工具、Android 的命令行，以及它的自定义的 init。

第 7 章 Android 框架，讨论了 Android 框架是如何启动的，与之进行交互的工具和命令，以及支持它正常运行所需要的守护进程。

附录 A 传统的用户空间，解释了如何获得“嵌入式 Linux”软件传统堆栈，来与 Android 的用户空间共存。

附录 B 为新硬件增加支持，向你展示如何扩展 Android 栈以增加对新的硬件的支持。这包括如何增加一个新的系统服务，以及如何扩展 Android 的硬件抽象层（HAL）。

附录 C 默认包列表的定制，为你提供指导，以帮助你了解自定义默认包含在 AOSP 生成的镜像中的是什么。

附录 D 默认的 init.rc 文件，包含了一组关于版本 2.3/ 姜饼和版本 4.2/ 果冻豆使用的默认 init.rc 文件的备注。

附录 E 资源，列出了一系列你可以找得到的有用的资源，比如说网站、邮件列表、书籍和活动。

## 软件版本

如果当你拿起这本书时，你可能还没意识到，我们在这里提到的版本很可能落后于当前的 Android 版本。而且有可能这种情况会永远向前。其实，我并不指望这本书的任何版本能够适用 Android 的最新版本。原因很简单：Android 的发布每六个月进行一次。我花了将近两年的时间来写这本书，并由过去的经验得知，不管怎样也需要至少六个月到一年的时间，来更新现有标题使它涵盖软件的最新版本。

所以，要么你立即停止阅读并马上退回这本书，要么你继续阅读下去，并找到一个关于如何最好的使用这本书的有说服力的解释，尽管它几乎可以保证已经过时了。

尽管它有非常快速的发布周期，Android 的内部架构和用于构建它的程序仍然是自推出以来，大约五年了都几乎没有变化。因此，尽管这本书最初写的是版本 2.3/ 姜饼，它一直相对简单地进行一些更新，也包括了版本 4.2/ 果冻豆以及对其他版本的参考，有需要的时候还包括了 4.0/ 冰淇淋三明治和 4.1/ 果冻豆。因此，虽然新的版本添加了新的功能，并且许多我们在这里讨论的软件组件将在下一个新版本中变得更丰富，但基本程序和机制很可能仍然能够在相当一段长时间内适用。

因此，你尽管可以放心，我承诺将继续关注 Android 的开发并力所能及的经常更新，你也应该能够从这本书中包含的解释中得到收获，以用在好几个版本中而不仅仅是提到的最后一个版本。

实际上有些人期望版本 2.3/ 姜饼能够在市面上很长一段时间，因为它对硬件的要求比之后的更高版本更加温和。例如，在 2012 年 12 月的 AnDevCon IV 发布会上，从 Facebook 来的大会发言人解释说，预计将支持其运行在 2.3/ 姜饼的设备上的应用程序很长一段时间，因为该版本能够运行在比最近版本更廉价的硬件上。

## 本书的约定

本书使用了下列印刷约定：

斜体 (*Italic*)

表示新的术语、URL、邮件地址、文件名以及文件扩展。

### 等宽字体 (*Constant width*)

用于程序列表，以及段落中关于程序的元素，例如变量和函数名称、数据库、数据类型、环境变量、声明以及关键字。

### 加粗等宽字体 (**Constant width bold**)

用于命令和其他由用户输入的文字。

### 斜体等宽 (*Constant width italic*)

用于应该使用用户提供的内容替换的文字，或者由上下文确定的内容。

---

**注意：**表示一个提示，建议或一般注释。

---

**警告：**这个图标表示一个警告或注意事项。

---

## 使用代码示例

这本书的目的是帮助你完成工作。在一般情况下，你可以在你的程序和文档中使用这本书里的代码。你并不需要联系我们得到我们的许可，除非你复制了代码的很大一部分。例如，编程的时候使用的几段这本书中的代码并不需要获得得到我们的许可。销售或发行一个 CD-ROM，里面包含 O'Reilly 出版书籍中的例子确实需要得到许可。引用本书来回答问题以及引用示例代码不需要得到许可。将本书中大量的示例代码放到你的产品文档中则需要许可。

我们对引用表示赞赏，但不是必要的。一个引用通常包括标题，作者，出版商和 ISBN。例如：“Karim Yaghmour, *Embedded Android* (O'Reilly), Copyright 2013 Karim Yaghmour, 978-1-449-30829-2.”

如果你觉得你使用的代码示例超出正当使用范围或上面给出的权限，请随时联系我们 [permissions@oreilly.com](mailto:permissions@oreilly.com)。

## Safari® Book Online

Safari 在线图书馆是一个请求式的数字图书馆，可以让你轻松搜索超过 7500 项技术和创意参考书以及视频，以便于你快速找到需要的答案。

通过订阅，你可以从我们的库中读取到任何网页，以及观看到任何在线视频。你可以在你的手机和移动设备上阅读书籍。可以访问新的可以用于打印的主题，并独家获得

作者的开发手稿和后续反馈、复制和粘贴代码示例，组织你的收藏夹，下载章节，关键章节打上书签，创建笔记，打印页面，并从其他大量的省时功能中获益。

O'Reilly Media 已经上传这本书到 Safari 在线图书馆的服务器中。想要获得这本书和其他 O'Reilly 出版的类似话题的完全版的数字访问，请在 <http://my.safaribooksonline.com> 上免费注册。

## 怎么联系我们

请邮寄有关这本书的意见和问题到出版商：

美国：

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室 (100035)  
奥莱利技术咨询（北京）有限公司

我们这本书有一个网页，在网页上我们列出了勘误表，示例和其他所有信息。您可以通过 <http://oreil.ly/embedded-android> 访问此页面。

发表评论或询问有关这本书的技术问题，请发送电子邮件至：

[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

想要获取关于本书的更多信息，课程，会议，以及新闻，可以访问我们的网页：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

在 Facebook 也可以找到我们：<http://facebook.com/oreilly>

你也可以在 Twitter 上关注我们：<http://twitter.com/oreillymedia>

也可以在 YouTube 上观看我们：<http://www.youtube.com/oreillymedia>

# 致谢

这是我的第二本书，我的第一本书是在 10 年前。我对自我认知有些比较模糊：特别是我自己单独工作时，例如写这本书时。但我似乎清楚地知道我有一种倾向——被天真地吸引到探索未知领域。当 2001 年时我开始写我的第一本书《构建嵌入式 Linux 系统》时，还没有任何一本书是完全描述嵌入式 Linux 是什么。我花了两年时间才写下来一半的文稿，而我本来以为我只需要花一年的时间。同样的，当我在 2011 年开始写目前这本书时，几乎没有关于嵌入式 Android 的信息。有点巧合的是，这也是我花了两年时间完成你现在拿在手上（或者现在你正在看着的你的屏幕、平板电脑、手机或任何尚在我写作时还未出现的设备）的手稿。

总的来说，我发现写书感觉就像是一场消耗战。也许这是因为我的主题选择的问题，也许这只是我自己的怪癖。不过，类似于消耗战，写雄心勃勃主题的书并不是可以单独完成的。事实上，当我开始写这本书时，我只知道你会在这本书中发现的一小部分。然而你可以打赌，我已经做了大量的研究，我还要强调的是，你在这里看到的东西，是我与许多有才华的开发者进行大量的交流得到的结果，他们每一个人教一点比我当时知道的更多一点的知识。因此，如果你曾经在会议或课堂上问过我一个问题，或者如果你曾经向我解释过你在做与 Android 相关的什么，或者你告诉过我你正在遭遇的问题是什么，或者更好的是，当我迷失在 Android 中时，你有给我正确的方向，你的那部分可能现在出现在这本书里的某个地方。

没有出版商的支持，这种类型的书是不可能成功的。正如我的第一本书，每个 O'Reilly 出版社的人都是出类拔萃的。我想首先感谢迈克·亨德里克森相信这个项目在我的能力范围内能够完成它。我也是极大的荣幸再次得到了与作为编辑的安迪·奥拉姆工作的机会。他再一次在审批你正在阅读的这本书的工作中，指出了技术问题。除了安迪之外，我也想感谢 Rachel Roumeliotis 和玛丽亚·史泰龙温馨地提醒我继续向前写作这本书。

写这类书籍在确保技术精度的准确性方面是非常谨慎的。因此对我来说，拥有雄厚的技术审查小组是至关重要的。因此，我想首先感谢马格努斯柏克、马克·格罗斯和 Amit Pundir 很早就同意来这个项目中审查这本书，并在写这本书的很长时间以内都提供了慷慨的反馈。这个最初的小组，由其他许多有才华的人一路上加盟进来。硬件大师大卫·安德斯提供了硬件章节的重要反馈。Robert PJ Day 也确保对于那些从来没有接触过 Android 的人来说，这本书仍有它的意义，并在这项伟大工作中做出了杰出贡献。Benjamin Zores 解决了几个有关栈的内部方面的问题。最后，这本书早期版本的读者，如 Andrew Van Uitert 和 Maxime Ripard，优雅地与我分享他们在阅读的过程中发现的一些问题。

我想特别感谢 Linaro 的 Android 团队和 Bernhard Rosenkränzer，而后者几乎凭借一己之力指出这本书与之前版本的大部分的差异，本书是以版本 2.3/ 姜饼为中心，并兼顾版本 4.2/ 果冻豆。如果你幸福地抱着一本书，而这本书涵盖了两个主要的 Android 版本，其中一个是写本书的时候最新的一个版本，请感谢 Bernhard。他不仅监督我更新这本书，而且迄今为止他的投入是最广泛的，而且往往是最详细的。我还想衷心感谢 Zach Pfeffer 提供了他的团队的帮助，使 Bernhard 的贡献成为可能，还要感谢 Vishal Bhoj、Fahad Kunnathadi 和 YongQin Liu 等人。

正如之前所说的，我在会议上遇见的很多人都对本书的编写提供了帮助。因此，我想挑出两个以自己的方式使我能够参加他们的会议的组织。首先，我想感谢 BZ 媒体团队，他们自 2011 年初以来一直在组织 AnDevCon 会议，并且在早期就相信我可以介绍 Android 的内部结构，并持续邀请我参与。特别感谢 Alan Zeichick、Ted Bahr、Stacy Burris 和 Katie Serignese。我也想感谢 Linux 基金会给我机会进行主题演讲、讨论，并参与多项他们已经举办了多年的活动，包括 Android 建设者峰会，嵌入式 Linux 会议，以及嵌入式 Linux 欧洲大会。特别感谢 Mike Woster、Amanda McPherson、Angela Brown、Craig Ross、Maresa Fowler、Rudolf Streif、Dominic Duval、Ibrahim Haddad 和 Jerry Cooperstein。

也特别感谢 RevolutionLinux 的团队，尤其是 Benoit des Ligneris、Bruno Lambert 和 Patrick Turcotte，在早期就同意做我的试验品。你们的信任已取得丰硕的成果。

最后，特别感谢谷歌的 Android 团队，他们有一个我遇见过的具有最佳头脑的团队。我真诚地想说：研究这个操作系统一直是我所做的最有趣的事情之一。创造了一个惊人的软件，并使其在这样一个宽松的许可下能够被如此慷慨地使用，这对于整个团队来说是一种荣誉。虽然我知道这是一种非同寻常的开源项目，其透明度不会（有很好的理由）提上议事日程，我还是想感谢那些通过不同方式提供了帮助（或在某些情况下，至少试过）的 Android 开发者。感谢 Brian Swetland 每隔一段时间给 Chet Haase 填满一些关于 LWN 的空白。

这些鸣谢是不完整的，不足以完全覆盖那些亲密关心过我的人。谢谢 Sonia、Anais、Thomas 和 Vincent 自始至终的爱的耐心。*Les mains invisibles qui ont écrit les espaces entre les lignes sont les leurs et je leur en suis profondément reconnaissant.*<sup>注 1</sup>

---

注 1：行与行之间的空白，是那些无形的手所写，对此我深深地感谢他们。



# 概述

将 Android 放到一个嵌入式设备上是一项复杂的任务，涉及对它的内部构件，对 Android 开源项目（AOSP）聪明的混合修改，以及它所运行的内核——Linux 的复杂理解。在我们详细描述嵌入式 Android 系统之前，先介绍一些基本背景，包括嵌入式开发者需要考虑如何与 Android 打交道，比如说 Android 的硬件需求，围绕 Android 的合理的框架，以及它所牵涉的嵌入式环境。首先，我们来看一下 Android 是怎么产生的，以及它是如何发展的。

## 历史

故事追溯<sup>注1</sup>到 2002 年初期，Google 的 Larry Page 和 Sergey Brin 在斯坦福大学参加了一个讨论会，讨论关于 Danger 公司旗下的当时最新的 Sidekick 手机的发展。演讲者是 Andy Rubin，他是当时 Danger 公司的 CEO，Sidekick 手机是首批多功能一体的，网络驱动的设备之一。讨论会之后，Larry 前去看设备并且很高兴地发现 Google 是它的默认浏览器。不久之后，Larry 和 Sergey 都成为了 SideKick 手机的用户。

尽管它很新奇，也有对它很热情的用户，但是 SideKick 手机并没有取得商业上的成功。到 2003 年，Rubin 和 Danger 公司董事会都赞成放弃它。在做了一些尝试以后，Rubin 决定，他仍然想做手机操作系统的生意。通过他拥有的一个域名 ([android.com](http://android.com))，

注1：与 Android 在 2007 年 11 月初次公布的同时，纽约时报刊登了一篇题为“我，机器人：谷歌手机背后的的男人”的文章，这篇文章由约翰·马科夫撰写，他深刻而非常有见地的描述了安迪·鲁宾和他的职业生涯。推而广之，它描述了很多 Android 背后的有识之士的故事。本节的部分内容是基于该篇文章而写的。

打算为手机产商着手开发一个公开的操作系统。在投入了他的大部分积蓄和收到的一些额外的种子基金到项目中后，他打算去争取公司投资。不久之后的 2005 年 8 月，Google 悄然收购了 Android 公司。

从收购之后到 2007 年 11 月向世界宣布这段时间里，Google 几乎没有发布有关 Android 的任何信息。但是操作系统的开发团队仍在拼命地工作，而交易和技术原型都是在幕后进行的。这个消息最初是由开放手机联盟（OHA）宣布的，这个组织是一群公司，它们带着发展移动设备开放标准的使命来发布这个消息，Android 成为他们的第一个产品。一年以后，2008 年 9 月，Android 的第一个开源版本 1.0 可以正式使用。

从那之后连续发布了几个 Android 版本，操作系统的进展和发展明显变得更公开了。就像我们之后看到的那样，尽管大部分的 Android 开发工作仍然继续处于封闭式的开发中。表 1-1 是对 Android 各个发布版本的概要说明，其中最值得关注的特性是从相应的 Android 开源项目（AOSP）中得到的。

表 1-1：Android 版本

版本号	发布日期	研发代号	最值得关注特性	开源
1.0	2008 年 9 月	未知		是
1.1	2009 年 2 月	未知 <sup>a</sup>		是
1.5	2009 年 4 月	纸杯蛋糕	屏幕软键盘	是
1.6	2009 年 9 月	甜甜圈	电池用量显示和支持 VPN	是
2.0,2.0.1,2.1	2009 年 10 月	松饼	支持 Exchange	是
2.2	2010 年 5 月	冻酸奶	即时编译技术	是
2.3	2010 年 10 月	姜饼	支持 SIP 和 NFC	是
3.0	2011 年 1 月	蜂巢	支持平板形状	否
3.1	2011 年 5 月	蜂巢	支持 USB HOST 及其 API	否
4.0	2011 年 11 月	冰淇淋三明治	同时支持手机和平板电脑	是
4.1	2012 年 6 月	果冻豆	很多性能得到了优化	是
4.2	2012 年 11 月	果冻豆	支持多用户	是

a. 据说这个版本被叫做“花式小蛋糕”（Petit Four），更多信息可以看着 Google+ 的帖子 (<http://plus.google.com/107797272029781254158/posts/CABJIRdxH8G>)。

## 特点和特征

在版本 2.3.x/ 姜饼发布时，Google 在它的开发者网站中曾发布过以下几个关于 Android 的特征：

## 应用程序框架

应用开发者用来开发 Android app 的应用程序框架，在线文档说明描述了该框架的使用方法，O'Reilly 公司的《Learning Android》等书籍也有详细的说明。

## Davik 虚拟机

一个纯粹的字节代码解释器，在Android系统中用来代替Sun（现已被Oracle收购）的Java虚拟机（VM）。与Sun的Java虚拟机不同的是，Davik解释.dex文件，而Java虚拟机解释的是.class文件。这些文件由dx程序通过使用.class文件产生，而.class文件是由JDK的java编译器部分产生的。

## 集成浏览器

Android包含一个基于WebKit的浏览器作为它的应用程序标准列表中的一部分。应用开发者能够通过调用WebView类来使得他们自己的应用程序能用WebKit引擎。

## 优化的图像处理

Android提供它自带的2D图像处理数据库，但是它的3D功能是基于OpenGL ES<sup>注2</sup>的。

## SQLite数据库

在这里可以找到标准的SQLite数据库，这个数据库可以通过应用程序框架被应用开发者使用。

## 媒体支持

Android通过StageFright提供了大范围的媒体格式支持，StageFright是Android的定制媒体框架。在版本2.2之前，Android曾经依赖过PacktVideo的多媒体核心（OpenCore）框架。

## GSM技术支持<sup>注3</sup>

这种技术支持主要基于硬件的支持，设备制造商必须提供一个HAL模块来使得Android系统可以与它们的硬件相连接。HAL模块将会在下一章节进行讨论。

## 蓝牙（Bluetooth），EDGE，3G和WiFi

Android提供对很多无线连接技术的支持。其中一些可能以Android特性的形式实现，比如说EDGE和3G，另外一些实现的方式与在普通Linux中一样，比如说蓝牙和WiFi就是其中的例子。

---

注2：OpenGL ES是OpenGL标准针对嵌入式系统开发的一个版本。

注3：Android支持的不仅仅只是GSM电话，而且是特征的名字，就像是正式的广告一样。

## 相机, GPS, 指南针和加速计

与用户环境的接口技术是 Android 的关键技术。应用程序界面（API）用于方便应用程序框架访问设备，因此这些设备需要一些 HAL 模块来提供支持。

## 丰富的开发环境

这一点可以说是 Android 最伟大的资产之一。开发环境可以使得开发者非常容易地开始进行 Android 开发。完整的 SDK 可以免费下载，它连同一个模拟器，一个 Eclipse 插件和大量调试开发工具，用于进行 Android 开发。

当然，Android 还有很多其他特性可以列出来，例如 USB 支持、多任务处理、多点触摸、SIP、网络共享、声控命令等，但是之前列出来的特性可以让你对将在 Android 里发现的东西更有把握和理解。也请记住，每一个新发布的 Android 版本都会带来属于它自己的新的一组特性。想要知道 Android 特性和增强功能的更多信息，可以查看与每一个发布版本同时出版的平台特性。

除了它的基本功能集，Android 平台还有几个特性使得它成为嵌入式开发中的一个特别有趣的平台。以下是一个简单介绍：

## 广泛应用生态系统

在写这本书的时候，Google Play（以前也称 Android 市场）里总共有 700 000 个应用程序。这跟苹果应用程序市场的 700 000 个应用程序比起来也是毫不逊色，它能够保证你有很多的选择，在你想在嵌入式设备中预安装应用程序时。特别要记住的是，在你安装某些应用程序的时候，你可能需要同意这个应用程序开发者规定的某些协议。在 Google Play 中，一个应用程序的可用性并不意味着你有作为第三方将它重新分配的权利。

## 统一的应用程序 API

所有应用框架中提供的应用程序接口（API）都是可以向前兼容的。因此，用户在自己的嵌入式系统中开发的自定义应用程序包也是可以继续在将来的 Android 系统版本中工作的。与此相反，你对于 Android 系统源代码的修改并不能保证能够继续在下一个 Android 发布版本中应用甚至是运行起来。

## 可更换组件

由于 Android 是开放源代码的，并且作为其体系结构中的一个优点来说，它的组件中很多是能够完全被替换的。例如，如果你不喜欢默认的启动器应用程序（主界面），你可以编写一个自己喜欢的。甚至更大的变化也能够在 Android 中实现。

比如多媒体框架 GStreamer<sup>注 4</sup> 的开发者，可以用它来替代 Android 中默认的媒体框架 StageFright，而 GStreamer 没有修改应用程序的 API。

#### 可扩展性

Android 系统的开放性和它的体系结构的另一个好处是，增加了对附加特性的支持并且其硬件也相对简单。你只需要仿真平台为其他硬件或者同类型的特性做些什么。举例来说，用户可以自己添加一些文件到 HAL 中去，来增加对自定义硬件的支持，对于这部分的具体解释可以查看附录 B。

#### 用户可定制

如果你仍然希望使用现有的组件，比如说现有的启动器应用程序，用户还可以根据自己的喜好对启动器进行定制。无论是调整它们的行为还是改变其外观和使用感觉，用户可以根据需要对 AOSP 进行修改。

## 开发模型

当用户考虑是否使用 Android 的时候，非常重要的一点是需要明白其开发过程可能所做的任何修改以及其对所有依赖项产生的影响。

### 与“经典”开放源码项目的差别

Android 的开放源代码特性是它最值得称道的特性之一。实际上，就像刚刚看到的那样，很多软件工程的效益都来源于开源，这一点应用到了 Android 中。

暂时先不管 Android 的许可证，与大多数其他开源项目的不同在于，它的开发过程几乎都是在封闭式环境下进行的。绝大多数的开源项目，比如说都有公共的邮件列表和论坛，可以让主要开发人员与其他人进行相互交流，有公共的代码仓库并提供对主要代码开发分支的访问。但是这些东西在 Android 中都没有。

Andy Rubin 自己对这种情况作了最好的总结：“开放源代码跟一个社区驱动的项目不同。Android 不是社区驱动项目，但是属于开源项目。”

不管我们喜欢不喜欢，Android 主要是由 Android 的开发团队在 Google 开发的，对于公众来说，大家既不知道有关于它的内部讨论，也不知道每一个开发分支所需的费用。反而 Google 通常是每六个月会在一个新的设备上发布一版新的 Android 系统加入到

---

注 4：GStreamer 是大多数 Linux 桌面环境中使用的默认的媒体框架，包括 Gnome、KDE 和 XFCE。

Android 的队伍中，每一次都会发布代码。比如说自 2010 年 12 月发布三星 Nexus S 一段时间以后，Android 新版本的代码就可以获得了，Android2.3/ 姜饼公开发布在 <http://android.googlesource.com/>。

显然，在开源社区中有一定程度的不适合继续使用术语“开源”来描述这样一个项目，其发展模式违背了标准的开源项目手法，尤其是在 Android 非常普及的情况下。开源社区历史上还没有采用这样类似开发模式的项目。其他人则担心这种开发模式会受到 Google 商业目标的潜在变化的影响。

抛开政治因素，虽然 Android 的开发模式意味着作为一个开发者，你对 Android 做出贡献的能力是有限的。事实上，除非你成为 Google 里 Android 开发团队的一个成员，否则你是不能够对它的前端开发分支做出任何贡献的。此外，除了极少数的例外情况外，你想要跟核心开发团队的成员面对面地讨论如何增强 Android 的问题几乎是不可能的。然而，你仍然可以在 <http://android.googlesource.com/> 地自由提交对于 AOSP 代码的改进和修改。

Google 这样做最坏的影响是，你绝对没有办法拿到有关于 Android 开发团队对于平台决策的内部消息。比如说，如果有新的功能加入到了 AOSP 中，或者如果核心部件做了修改，你可以通过分析下一版的代码库，发现到底做了多少变化，以及它们如何影响你对之前版本所做的改变。此外，你不会有任何方法去了解造成这些改变和调整的内在需求、限制，以及所做的各种权衡。如果这是一个真正的开源项目，应该有一个公共邮件列表存档，存储所有的信息或者是指向它的链接。

话虽这么说，但重要的是要记住 Google 将 Android 作为一个开放源代码项目来发布，其贡献是多么的重大。尽管从一个开源社区的角度来看它的开发模型是很尴尬，但 Google 在 Android 上的工作对大量的开发人员来说仍然是一个天大的好事。另外，Google 完成了一件其他开源项目都没有达成的事情：成功创造了一个最大规模发布的 Linux 发行版。因此，我们很难再去苛求 Android 的开发团队。

此外，大家很容易地认为，从业务和进入市场的角度来看，一个社区驱动的过程肯定会阻碍到 Google 将要尝试发布的任何新产品的宣传公告，使得新产品的发布给人带来惊喜的效果成为不可能，因为它的每一种特性都是被公开开发的。也就是说，在社区驱动开发过程中，可以看到一群人花上几年时间才能统一，并用最好的方法来实现一个特定的功能集。而且，根据过去的成绩简单地分析，Android 的成功无疑是得益于 Google 能够迅速向前推动它，并通过发布一个很酷的新产品来引起公众的兴趣。

## 特性包含、线路图和最新版本

简单地说，没有公开可信的关于未来的 Android 版本的特性和功能的线路图。在最乐观的情况下，Google 将提前宣布下一个版本的名字和大概发布的日期。通常你可以期待一个新的 Android 版本会在 Google 的 I/O 大会上发布，这个大会通常在五月召开，或者另一个可能的发布时间是在一年的年底。新发布的版本会有什么？每一个人都在猜测。

然而，通常情况下，Google 将只选择一个制造商来合作下一个 Android 版本的发布。在此期间，Google 将会与那个生厂商的工程师密切合作，共同促成下一个 Android 版本在一个定向的即将成为领导者（或者是标杆性）的设备上运行。在此期间，只有制造商的团队，是可以访问 Android 开发分支的尖端信息。一旦这个设备被投放到市场上，相应的源代码库会转为一个公共存储库。在接下来的版本中，它会选择另一个生厂商并重新开始。

在这个周期中有一个明显的例外：Android3.x/ 蜂巢。在这个特别的例子中，Google 并没有发布源代码给相应的领导设备——摩托罗拉的 Xoom。其中的理由似乎是这样，Android 开发团队在分叉代码库的时候，希望尽快得到一个准备用于平板电脑的 Android 版本以反应市场。因此，在该版本中，很少考虑到与手机的外形因素保持向后的兼容性。鉴于此，Google 并不希望让这个版本的代码对外可用，以避免其平台的分裂。同时支持手机和平板电脑的外形因素随后在 Android4.0/ 冰淇淋三明治版本中得以实现。

## 生态系统

到 2013 年 1 月：

- 每天约有 130 万 Android 手机被激活，相比 2011 年 6 月的 40 万和 2010 年 8 月的 20 万有所增长。
- Google Play 包含将近 70 万个应用程序。相比之下，苹果的 App Store 有大约相同数量的应用程序。<sup>注 5</sup>
- Android 拥有全球智能手机市场的 72% 的份额。

Android 显然正处于上升之势。实际上，Gartner 曾经预测，到 2012 年 10 月，Adnroid 将成为占主导地位的操作系统，到 2016 年将击败可敬的对手 Windows。就像

---

注 5：在写本书时，是有史以来 Google Play 在应用程序数量上第一次赶上苹果的 App Store。

大约十年前 Linux 打乱嵌入式市场一样，Android 已经做好准备创造自己的市场。它不仅要翻转手机市场，挤占或者是消除一些甚至是强大的对手，而且在嵌入式领域它很可能将成为绝大多数以用户为中心的嵌入式设备中事实上的标准 UI。甚至有迹象表明，它很可能会在无显示（或者是不以用户为中心）设备中取代传统的“嵌入式 Linux”。

因此，一个完整的生态系统迅速建立在 Android 周围。硅和系统芯片（SoC）生产厂商，如 ARM、TI、Qualcomm、Freescale 和 Nvidia 在它们的产品中加入了 Android 支持，手机和平板电脑制造商比如摩托罗拉、三星、HTC、索尼、LG、Archos、戴尔和华硕等的加入使得预装 Android 的设备数量不断增多。这个生态系统还包括了越来越多的各种不同的参与者，比如 Amazon、Verizon、Sprint 和 Barnes & Noble 等都创建了自己的应用程序市场。

基层的社区和项目也开始围绕 Android，即使它是闭门开发出来的。就像传统的开源项目一样，通过公共的邮件列表和论坛做了很多努力。这种社区的工作通常开始于将官方的 Android 源代码的版本进行分叉，来生成它们自己的自定义特性和增强功能的 Android 产品。可能会出现这样的情况，比如 CyanogenMod 项目，它为有能力的用户提供了售后镜像文件。还有各种各样的硅提供商也做了其他努力，通过提供 Android 版本来启动或增强它们的平台。比如说，Linaro，一个由 ARM SoC 厂商创建的非营利性组织，通过提供它们自己的优化后的 Android 树来巩固他们的平台。还有一种力量来自于手机玩家，通过对制造商提供的二进制文件进行修改或者调整来创建自己的分支。想要获得完整的 AOSP 分叉列表和开发它们的社区，可以看一看附录 E。

## 关于开放手机联盟的一些话

正如我前面所提到的，OHA（手机联盟）是首次公布 Android 的最初的工具。它在自己的网页上这样描述，“一个由 82 个技术和移动公司组成的组织，它们聚集到一起来加速移动电话的创新并为消费者提供一个更丰富，更便宜和更好的移动体验。我们一起开发的 Android 将是第一个完整、公开且是免费的移动平台。”

然而，除了最初发布公告之外，目前还不清楚 OHA 到底扮演的是什么样的角色。例如，一个参与 Google 2010 年举办的“与 Android 团队的非正式座谈会”的与会者，曾经问过座谈小组，作为一名 OHA 成员公司的一名开发人员，他能够获得什么样的特权。通过与周围的座谈小组成员进行讨论后，发言者基本的回答是座谈小组不知道，因为他们不是 OHA 的成员。因此，就会出现这样的情况，对于 Android 开发团队本身来说，OHA 成员的优势并不明确。

OHA 所处的角色在现实中更是模糊不清，它似乎没有成为一个全职的组织，没有董事会成员和长期的员工。相反，它只是一个“联盟”。此外，在 Google 对于 Android 的每一次发布中都没有提及任何关于 OHA 的信息，也没有从 OHA 再发布过任何关于 Android 的新的公告。总之，人们会很自然地这样推测，Google 将 OHA 聚集在一起似乎主要是作为一个超前的市场营销，以显示行业对于 Android 的支持，但是在实际中，它对于 Android 的发展没有一点关系。

## 获取“Android”

想要让 Android 在你的嵌入式系统中运行，主要有两个方面的需要：一个兼容 Android 的 Linux 内核和 Android 平台。

在很长一段时间里，想要获得一个兼容 Android 的 Linux 内核是一件很困难的事情，编写本书时在某些情况下还是如此。<http://kernel.org> 下载的纯净内核运行不了平台，用户必须要么使用一个 AOSP 中可用的内核，要么修补一个纯净内核使它能够兼容 Android。根本的问题在于 Android 开发者为了允许他们自定义的平台能够运行，在内核中增加了很多东西。反过来，这些新增到官方主流内核的东西曾经遇到过很大的阻力。

我们会在下一章更详细地介绍有关内核的问题，自从 2011 年在布拉格召开的内核峰会开始，内核开发者决定积极寻求注入能够运行在主流 Linux 内核版本上的 Android 平台所需要的特性。因此，很多所需的功能被合并了，而另一些功能已经（或者在写本书的时候，即目前正在）被替换或者被其他的机制所取代。在本书的写作期间，想要获得一个预装 Android 内核最简单的方法是去询问你的 SoC 供应商。事实上，由于 Android 的普及，大多数主流的 SoC 供应商在它们的产品中对所有 Android 需要的组件提供积极的支持。

Android 平台本质上是一个定制的 Linux 发行版，包含了用来弥补通常被称为“Android”所需的用户空间包。表 1-1 所列举的版本实际上是平台的版本。我们将在下一章讨论平台的内容和架构。现在请记住，每一个平台版本都有它相应的作用，类似于标准的 Linux 发布版本中的 Ubuntu 或者 Fedora。这是一个自我连贯的一套软件包，一旦编译，将会通过特定的工具、接口和开发人员的 API 来提供特定的用户体验。

---

**注意：**我们把运行在 Android 兼容内核上的“Android 平台”源码称为“AOSP”，事实上这个词贯穿整本书，虽然它本身的意思是 Android 开放源代码项目。在其托管的这个网站 (<http://android.googlesource.com/>) 上，也包含了很多对于平台的补充内容，比如说参考 Linux 内核树和额外的软件包，采用普通的 *repo* 命令来获取平台时这些内容是下载不到的。

---

## 破解二进制文件

即使不能够访问 Android 源代码，这也不能阻止富有激情的玩家们按照他们的意愿破解和定制 Android 操作系统。例如，虽然 Android 3.x/蜂巢系统不提供 Barnes&Noble Nook 电子书的支持，玩家们仍然使其在该硬件上运行起来了。其原理是在 Android 3.x 系统 SDK 提供的模拟器镜像中提取二进制文件，放到 Nook 硬件上运行，这当然也会丧失硬件加速性能。类似的破解方法也应用到很多厂商没有提供源代码的实际设备上，玩家们用来“root”设备或者更新各种版本的 Android 组件。

## 法律框架

与其他任何软件代码一样，需要在一系列的授权和知识产权限制框架之下使用和发布 Android 系统。下面我们进行一些简单的介绍。

---

**警告：**显然我不是专业的律师，所以这不是专业的建议。所以，你应该和你的主管法律顾问来详细探讨这些条款和授权是如何应用到你的项目当中去的。当然，我参与到开放源代码社区也已经很多年了，所以你也可以参考一下我这个资深工程师的观点。

---

## 代码许可

如前所述，“Android”由两部分组成：一个兼容 Android 的 Linux 内核和一个 AOSP 版本。尽管 Linux 内核已经被修改为可以运行 AOSP，但是它仍然遵照于 GNU GPLv2 许可证。因此，请记住任何你对内核所做的修改都不允许在除了 GPL 以外的其他许可证许可下发布出去。那么，如果你从 <http://android.googlesource.com> 或者是从你的 SoC 供应商那里获得一个内核版本，并且对它做了修改使它能够在你的系统中运行，在遵守 GPL 许可的情况下允许你对修改后的内核镜像进行发布。这就意味着，你必须用源代码以及你对内核的修改生成镜像，并且在接受 GPL 条款的前提下最终用户才可以使用你的镜像。

内核的源代码中的 *COPYING* 文件包含一个由 Linus Torvalds 提供的注意事项，其中清晰地指出，只有内核是遵循 GPL 的，应用程序是运行在它上面的才不会被认为是“衍生的工作”。因此，你可以随意地创建应用程序，让它运行在 Linux 内核之上，并在你选择的许可证的允许下进行发布。

在开放源代码的圈子里，以及大部分选择支持 Linux 内核或者基于该内核研发产品的公司，对于这些条款及其适用范围通常来说是很容易理解并接受的。除了内核以外，基于 Linux 内核的发行版所包含的大量关键组件通常是采用某种形式的 GPL 授权的。例如，GNU C 语言函数库（glibc）和 GNU 编译器（GCC）分别是采用 LGPL 协议和 GPL 协议。嵌入式 Linux 系统中常用的 uClibc 和 Busybox 等重要的软件包也是采用 LGPL 和 GPL 授权。

然而，并不是所有人都适应 GNU GPL。事实上，它强加于许可上的限制所带来的衍生产品，给大型组织带来严峻的挑战，尤其是考虑到地理的分布，开发部门的不同位置带来文化的不同，以及对于外部的分包商的依赖。比如说在北美销售产品的一个制造商，可能必须面对即使不是几百个，也有几十个供应商。每一个供应商也许都会提供一块可能包含（也可能不包含）GPL 的代码。然而，制造商在销售产品给客户的时候，将一定会提供 GPL 组件的源码，不管是哪个供应商提供的组件。此外，项目的实施必须要到位，以保证工作在基于 GPL 项目的工程师都遵守许可证。

当 Google 开始与制造商联合开发其“开放”的手机操作系统时，很快就出现这样的情况，很明显尽可能地避免使用 GPL。事实上，除了 Linux 以外的其他内核显然也被考虑过，但是选择 Linux 是因为它已经有强大的产业支撑，特别是来自于 ARM 芯片制造商的支持，也是因为它的 GPL 许可能够非常不错地与系统的其他部分分离开来，所以它的影响不大<sup>注6</sup>。

尽管这样，最后决定将尽一切努力来保证大部分的用户空间组件所基于的许可证，并不构成与 GPL 相同的逻辑问题。这就是为什么大多数普通的 GPL- 和 LGPL- 许可的组件在嵌入式 Linux 系统中很常见，比如说 glibc、uClibc 和 BusyBox 都不包含在 AOSP 中。相反，Google 为 AOSP 创建的大部分组件都是基于 Apache License 2.0（也叫做 ASL）结合一些关键组件发布，比如说 Bionic library（glibc 和 uClibc 的一个替代）和 Toolbox utility（BusyBox 的一个替代），都是基于 BSD 许可证。一些经典的开源项目也包含其中，加入到 AOSP 中的 external/ 目录的大多数是在它们原来许可证许可

---

注 6：可以查看由一个 Android 内核开发团队的成员 Brian Swetland 写的 LWN post，获得关于这些选择背后的原理的更多信息。

下的源代码。这就意味着 AOSP 的一部分是既不基于 ASL，也不基于 BSD。实际上，AOSP 仍然包含 GPL 组件和 LGPL 组件。然而，二进制文件的发布都是通过编译这些组件得到，并不打算由 OEM 特别定制（即没有预想的衍生工作产生）以后发布，应该不会构成任何问题，这些用于 AOSP 中的组件的源代码都是现成的，所有的代码可以在 <http://android.googlesource.com> 上下载，因此在必要的情况下要遵守 GPL 的要求，再分配的衍生产品应该继续遵守 GPL。

与 GPL 不同，ASL 并不要求它的衍生产品在遵守一个特定的许可证下才能发布。事实上，你可以选择最适合你所做修改的方式。以下是 ASL 的相关部分（完整的许可证条款可以从 Apache 软件基金会 (<http://www.apache.org/licenses/>) 上获得）：

- “受制于这个许可证的条款和条件，每一个贡献者授予一个永久性，全球性，非排他性，不收费，免版税的不可撤销的版权许可，允许复制，准备衍生产品，公开展示，公开演示，再次许可和以源代码或者对象的形式发布你的作品和衍生产品。”
- “你可以添加关于你自己的修改部分的版权声明，并提供额外的或者不同的许可条款和条件来约束使用，复制或发布你的修改部分的情况，或者是将任何的衍生产品作为一个整体，提供了使用，复制和发布你的产品以及其他人在这个许可证中规定的情况。”

此外，ASL 明确规定了专利许可授权，这就意味着当你使用 ASL 许可的 Android 源代码时，你不需要任何 Google 授予的专利许可。除此之外它还规定了几个“管理”要求，比如说需要清楚的标记修改过的文件，提供接收者一份 ASL 许可证的副本，按照原始的样子保存 NOTICE 文件。不过从本质上来说，在符合你的意愿的条款下，可以自由授权你的修改。BSD 许可证覆盖了 Bionic 和 Toolbox，只允许类似的二进制发布的组件。

因此，制造商可以得到 AOSP 并根据需要对它进行定制，同时如果愿意可以持有这些修改的所有权，并且只要在继续遵守其余的 ASL 条款的条件下都有效。如果不出意外，这至少实现了一个用于跟踪所有修改的方法，并无需一定得按照 GPL 协议把修改交给客户。

## 增加 GPL 许可的组件

虽然已经尽最大努力使得 GPL 远离 Android 的用户空间，但是也有这样的情况，你可能明确地希望增加 GPL 许可的组件到你的 Android 发布版本中。比如说，你想要包含 glibc 或者是 uclibc，它们是兼容 POSIX 的 C 函数库，相较于不兼容 POSIX 的 Android 的 Bionic，因为你可能想要在 Android 上运行之前的 Linux 应用程序，但是又不想把它们移植到 Bionic 上。或者你想使用 BusyBox 而不是 Toolbox，因为后者在功能上比前者有更多的限制。

这些增加对你的开发环境来说也许是暂时的，可能在最终版本里会被删除，或者也可能永久地存在你所定制的 Android 系统中。不管你决定走什么样的路，无论是普通的 Android 或是一些带有额外 GPL 包的 Android，请记住，你必须遵循许可证的要求。

## 品牌的使用

虽然 Google 对 Android 的源代码非常大方，但是它对于 Android 相关的品牌元素的控制却很严格。让我们来了解一下这些元素和它们所用的相关术语。想要获得官方列表以及官方的术语，可以到这个网站 (<http://bit.ly/Zu5HCV>) 上看一下。

### Android 机器人

这个我们熟悉的绿色机器人在所有与 Android 有关的东西上都可以看见。它的作用类似于 Linux 的企鹅，使用权限也同样是任何人都许可的。实际上，Google 这样描述它，“在市场营销中是可以被使用，复制和随意修改”。唯一的要求是，相应的属性要遵守创作共享署名授权协议的条款。

### Android 商标

Android 的商标是一组自定义字体的字母，拼写为 A-N-D-R-O-I-D，它们会在设备或者模拟器启动时出现，也可以在 Android 的主页上看到。在任何情况下你都不能被授权使用这个商标。第 7 章将会告诉你怎样换掉开机的商标标志。

### Android 自定义字体

用于展现 Android 商标标志的是 Android 自定义字体，它的使用权限和 Android 商标一样严格。

### 正式命名中的“Android”和消息传送

就像 Google 说的那样，“‘Android’本身不能直接用于一个应用程序或者是其附件产品的命名，而应该是使用‘Android 版’来代替。”因此，你不能直接

命名为“Android 媒体播放器”，但是你可以命名为“Android 版本的媒体播放器”。Google 还指出，“Android 可以用作一个描述语，只要它后面是一个恰当的通用术语。”比如说“Android™ 应用程序。”当然，正确的商标归属权限必须遵守。总的来说，在没有 Google 的许可下，你不能给你的产品命名为“Android Foo”，但是“Android 版本的 Foo”是可以使用的。

### “Android” 品牌设备

就像 Android 兼容程序（ACP）常见问答 (<http://source.android.com/faqs.html#compatibility>) 里所说的，“如果制造商想要使用 Android 来命名他们的产品，它必须首先证明它们的设备是可兼容 Android 的。”如果你的设备品牌命名里有“Android”，那么 Google 就有监管的特权。从本质上讲，在你的设备能够向外公布命名为“Foo Android”前，你必须确保你的设备是符合要求的，然后和 Google 讨论并与之达成某种协议。我们将在本章后面部分介绍 Android 兼容程序。

### 正式命名中的“Droid”

你不能在一个名字里单独使用“Droid”，比如说“Foo Droid”。出于某种我还没有完全搞明白的原因，只知道“Droid”是 Lucas 电影公司的商标。在你能够使用它之前，你好像必须要达到非常了解的级别。

## 文字（品牌）游戏

在 Google 对于 Android 商标的使用控制非常严格的同时，ASL 用于批量许可 AOSP 的规定如下：“本许可并不允许使用商品的名称、商标、服务标志或许可证颁发者的产品名称，除了必要时可以用合理且习惯性使用的用法来描述初始工作和重述 NOTICE 文件的内容。”

这就清楚地表明，你没有权利使用相关的商标，“但是合理的且习惯性的用法可以用于描述初始工作”这一例外被很多人看作是允许你将你的设备描述为“基于 AOSP”。有人更进一步，将他们的产品简单地描述为“以 Android 为基础的”或者是“基于 Android”。你甚至可以找到一些聪明的营销商冒险地使用 Android 的机器人来为他们的产品做广告，但是却不提及“Android”这个词。

大概我所见过的最卑鄙的双关语之一是，有一个产品列举了以下的内容作为其中一个特点的一部分：“运行 Android 应用程序”。你可以跟自己打赌，如果它能够运行 Android 应用程序，几乎可以保证会以某种方式、形态或形式来包含 AOSP。

## Google 自己的 Android 应用程序

虽然 AOSP 包含了一组在 ASL 下有效的核心应用程序，“Android”品牌的手机里通常还包含一组额外的“Google”应用程序，它们不属于 AOSP，比如说 Play Store（“应用程序市场”里的应用），YouTube，“地图与导航”，Gmail 等。很明显，用户希望这些程序是 Android 的一部分，因此你可能就会想在你的设备上运行它们。如果是这样，你就必须遵守 ACP 许可条款，并与 Google 达成协议，这与你想要被允许在你的产品名称中使用“Android”要做的事一样。我们后面会介绍 ACP。

## 非主流的 Android 应用程序市场

虽然主流的应用程序市场（即 Google Play）是由 Google 主管的，用户可以通过应用商店（Play Store）将应用程序安装到“Android”品牌的设备上，其他用户还可以利用 Android 开放的应用程序接口和许可的开放源代码来提出非主流的应用程序市场。网上商户的情形就是这样，比如说亚马逊（Amazon）和 Barnes & Noble，同样的还有移动网络运营商，比如说 威尔森（Verizon）和斯普林特（Sprint）。事实上据我所知，没有什么可以妨碍你创建自己的应用程序商店。甚至有开源项目例如 Affero 许可的 F-Droid 版本库，既提供一个应用程序市场应用，也提供一个 GPL 许可的相应的服务器后端。

## Oracle 与 Google

作为收购 Sun Microsystems 的一个成员，Oracle 收购了 Sun 公司拥有的 Java 语言的知识产权。根据 Java 的创造者 James Gosling<sup>注7</sup> 所说，很明显在收购过程中，Oracle 从一开始就打算通过 Sun 的 Java 知识产权包来追赶 Google 的脚步。而在 2010 年 8 月 Oracle 就是这么做的，它对 Google 提起诉讼，声称 Google 侵犯了多项专利以及版权。

不用深入分析就很明显地知道，Android 确实严重依赖于 Java。很明显 Sun 创造了 Java 语言并且拥有大量围绕 Java 的知识产权。在似乎大家都在努力预测 Sun 将会针对 Android 提出什么样的索赔要求时，Android 的开发团队费尽心思地尽可能少地在 Android 操作系统中使用 Sun 的 Java。Java 实际上主要由三个部分组成：语言及其语义，能够运行由 Java 编译器生成的字节码的虚拟机，包含 Java 应用程序在运行时所用到的封装包的类库。

---

注 7：想要知道更多细节，可以到 Gosling 的博客上找关于该话题的文章：[http://nighthacks.com/roller/jag/entry/the\\_shit\\_finally\\_hits\\_the](http://nighthacks.com/roller/jag/entry/the_shit_finally_hits_the) 和 [http://nighthacks.com/roller/jag/entry/quite\\_the\\_firestorm](http://nighthacks.com/roller/jag/entry/quite_the_firestorm)。

Java 组件的官方版本是由 Oracle 提供，它是 Java 开发工具包（JDK）和 Java 运行环境（JRE）的一部分。从另一方面来说，Android 只依赖于 JDK 中的 Java 编译器来构建 AOSP 的部分内容，这不是 AOSP 生成的镜像的一部分。此外，它不用 Oracle 的 Java 虚拟机，Android 依赖于 Dalvik 虚拟机，它是专门为 Android 定制的。它不使用官方的类库，Android 依赖于 Apache Harmony 库，它是完全重新实现的类库。因此，Google 似乎在尽一切努力来避免任何版权或者版本发布的问题。

尽管如此，这起诉讼案的结果何去何从还有待观察。虽然在 2012 年 5 月，Google 在初次判决中获得了版权和专利方面的初次胜利，但是 Oracle 在同年的 10 月份对判决提出了上诉。当然由于牵扯到很多方面的利益，这起诉讼看似会需要很多年才能有眉目。如果你想要知道关于这起诉讼的最新的消息或者研究之前的情况，我建议你看一下 Groklaw 网站 (<http://groklaw.net/>)，以及查阅相关的维基百科条目 ([http://en.wikipedia.org/wiki/Oracle\\_v.\\_Google](http://en.wikipedia.org/wiki/Oracle_v._Google))。

另一个间接相关但是却关系重大的发展是，2010 年 10 月 IBM 加入了 Oracle 的 OpenJDK 的开发。IBM 是 Apache Harmony 项目背后的推动力量，这个项目研发的就是用在 Android 上的类库，IBM 的退出几乎意味着这个项目将变得孤立无援。在写作本书的时候，这个事情对于 Android 的影响还是未知的。

顺便说一句，James Gosling 在 2011 年 3 月加入了 Google，尽管他后来离开了。

## 手机专利

虽然上节已经做了一定程度的分析，但是在写本书时关于诉讼和法律的风波仍然是移动手机世界的冰山一角。手机的销售量已经超过了传统 PC，手机市场的增加导致大多数参与其中的人由于与他们的竞争对手的纠纷，都以某种方式被卷入到与法律的对抗中。甚至还还有一个维基百科的条目题目为“智能手机纠纷” ([http://en.wikipedia.org/wiki/Smartphone\\_wars](http://en.wikipedia.org/wiki/Smartphone_wars))，专门列出正在进行的手机纠纷。

很难说这些纠纷的结果会是怎么样。这似乎是没有尽头的，各个公司都会采用各种策略以确保它们从中获利。例如苹果和三星，在写本书的时候，在不少国家它们都涉及在相互针对对方的诉讼案件中。还盛传微软会接触各种制造商，要求获得 Android 系统的使用专利费，这个事件从 Barnes & Noble 向法院提交的一些文件中得到证实，Barnes & Noble 由于拒绝缴纳此费用而被微软起诉。

这些事件会对你自己的产品有什么样的影响还很难说。与往常一样，必要时可以向这方面的法律顾问进行咨询。通常这是一个量的问题。所以如果你的产品面向的是利基

市场，你可能只是一个小角色无关紧要。从另一方面来说，如果你要创建一个面向大众市场的产品，你就会想要确保你已经覆盖了的所有市场。

## 硬件与合规性要求

原则上，Android 操作系统应该是可以运行于任何 Linux 设备上的。ARM、x86、MIPS、SuperH 和 PowerPC 等所有 Linux 操作系统支持的体系架构上，Android 确实都已经能够运行。当然这同时也说明，如果你想要将 Android 移植到你的硬件上，必须首先移植 Linux 操作系统。除了需要运行 Linux 内核以外，AOSP 所需要的其他硬件要求就很少了。不过，AOSP 对于硬件还有一个逻辑上的需求，即需要提供某种显示和指针机制使得用户能够通过这个接口与设备进行交互。当然，如果使用了 AOSP 暂不支持的外设，你可能不得不修改 AOSP 才能使其在你的硬件配置上工作。例如，如果你的产品上没有 GPS 模块，你可能就需要像 Android 仿真器那样给 AOSP 提供一个模拟的 GPS HAL 模块。你当然也需要确保你的硬件拥有足够的存储空间来存储 Android 镜像，以及足够强大的 CPU 来给用户相当好的用户体验。

所以，总体来说，如果你仅仅是想将 AOSP 在你的硬件上运行起来，那么硬件方面的限制是非常少的。然而，如果你研制的设备必须带上“Android”标签，或者必须包含典型 Android 设备上标准的 Google 应用（例如 Google 地图或者 Play Store），你就需要通过之前提到的 Google 兼容性测试程序 (ACP, Application Compatibility Program)。ACP 有两个独立而又相辅相成的部分：合规性定义文档 (CDD, Compliance Definition Document) 和兼容性测试套件 (CTS, Compliance Test Suite)。即使不打算参与 ACP，你可能还是要看看 CDD 和 CTS，因为它们给出了一般思维定式下较好的体验方式，有助于理解你所使用的 Android 版本的设计理念。

---

**警告：**每一个 Android 发布版都有它自己的 CDD 和 CTS。所以，你必须要有和你最终产品准备使用的版本相匹配的 CDD 和 CTS。如果在项目过程中切换了 Android 版本（例如，因为你想拥有 Android 新发行版中出现的新的炫酷特性），你将需要确保你跟这个新版本的 CDD 和 CTS 一致。还需要记住的是，你需要和 Google 交流来确保兼容性。所以，切换是一个比较大的跳跃并且可能带来产品交付的延迟。

---

ACP 的总体目标（包括 CDD 和 CTS）就是要确保为用户和应用程序开发人员提供一个统一的生态系统。因此，在你被允许以“Android”品牌销售设备时，谷歌希望确保你所支持的不是支离破碎的 Android 生态系统，不会引入不兼容或缺陷产品。反过来，这对制造商也是有益的，因为当它们使用“Android”品牌时也会受益于别人的合规性。

可以看看在这个网站 (<http://source.android.com/compatibility/>) 上对于 ACP 有关的更多细节。

---

**警告：**请注意，谷歌有权拒绝你参加到 Android 生态系统，因此会阻止你的设备上搭载 Play Store 应用程序和使用“Android”品牌。如它在其网站上所述：“不幸的是，由于各种法律和商业方面的原因，我们不会自动许可在所有兼容设备上搭载 Google Play 应用。”

---

## 合规性定义文档

合规性定义文档 (CDD) 是 ACP 的政策部分，可在上述 ACP 网址中找到它。它指定了要考虑兼容的设备必须满足的要求。CDD 的语言是基于 RFC2119 的，大量使用了“必须”、“应该”、“可能”等来描述不同的属性。它的篇幅大约 25 页，涵盖了设备的硬件和软件功能的各个方面。从本质上讲，它涵盖了每一个不能简单地使用 CTS 自动测试的方面。让我们去了解一些 CDD 要求的内容。

---

**警告：**这里的讨论是基于 Android 2.3/ 姜饼系统的 CDD，你所使用的特定版本与这里的讨论可能存在一些细微的差别。

---

## 软件

本节列出了 Java、本地 API、网络、虚拟机和用户接口等方面兼容性要求。从本质上讲，如果你使用的是 AOSP，就应该已经符合这部分的 CDD 了。

### 应用程序安装包兼容性

本节规定你的设备必须能够安装和运行的 .apk 文件。所有使用 Android SDK 开发的 Android 应用程序都被打包为 .apk 文件，而这些文件通过 Google Play 发布并安装到用户设备上。

### 多媒体兼容性

这部分的 CDD 描述了设备在媒体编解码器（解码器和编码器）、录音、音频延迟等方面的要求。AOSP 中包含了 StageFright 多媒体框架，因此，你使用 AOSP 就应该很容易符合 CDD。然而，你还是应该读一下音频录制和延迟部分，因为它们含有的特定技术信息可能会影响到你采用的硬件类型或者硬件配置。

## 开发者工具兼容性

本节列出了你的设备必须支持的 Android 特定的工具。基本上，这些都是常见的应用程序开发和测试过程工具：*adb*、*ddms* 和 *mokey*。通常情况下，开发人员并不会与这些工具直接交互。相反，他们通常在 Eclipse 开发环境中开发和使用 Android 开发工具（ADT）插件，这个插件将负责与这些低级别工具进行交互。

## 硬件兼容性

这可能是对嵌入式系统开发人员来说最重要的部分，因为它对目标设备的设计决策有着深远的影响。这里对每个子条款所列内容做一个概述。

### 显示与图形

- 你的设备屏幕的物理尺寸必须至少 2.5 英寸以上。
- 它的密度必须至少 100dpi。
- 其纵横比必须介于 4:3 与 16:9 之间。
- 它必须支持屏幕方向从纵向到横向的动态变化，反之亦然。如果不能改变方向，那么它必须支持 letterboxing（即宽荧幕模式，改变方向时屏幕上出现黑边，使得显示纵横比发生变化），因为应用程序可能会作强制方向的变化。
- 必须支持 OpenGL ES 1.0，暂时还不需要 2.0 的支持。

### 输入设备

- 你的设备必须支持输入法框架，它允许开发者能够创建自定义的屏幕软件盘。
- 你的设备必须支持至少一种软键盘。
- 不能包含与 API 不兼容的硬件键盘。
- 必须提供 Home、Menu、Back 按钮。
- 必须有一个触摸屏，不管它是电容式还是电阻式的。
- 如果可能，就尽可能支持独立的跟踪点（多点触摸）。

### 传感器

虽然所有的传感器都使用了“应该”，意思是它们不是强制性的资格要求。你的设备必须准确地报告存在或不存在的传感器，并必须返回一个精确的所支持的传感器列表。

## 数据连接

这里最重要的内容是有一个明确的语句声明 Android 可用于没有电话硬件的设备上。添加这一条是为了允许 Android 运行于平板设备上。此外，你的设备应具有的硬件支持 802.11x、蓝牙和近场通信（NFC）。最终，你的设备必须支持某种提供 200kb/s 以上带宽的网络。

## 摄像头

你的设备应包括一个后置摄像头，并且可以有一个前置摄像头。

## 内存和存储

- 你的设备必须拥有至少 128MB 的内存用于内核和用户空间。
- 必须拥有至少 150MB 的空间用于存储用户数据。
- 必须拥有至少 1GB 的共享存储（Shared Storage）。它通常是（但不总是）一张可移动的 SD 卡。
- 它必须提供一种机制允许通过 PC 访问共享存储。也就是说，当设备通过 USB 连接时，SD 卡中的内容能够通过 PC 读取。

## USB

这一要求可能是“Android”品牌的设备“必须以用户为中心”理念的最深刻的体现，因为它本质的意思是如果用户拥有设备，那么就要求它在被用户连接到计算机上时，用户就能够完全控制设备。在某些情况下，这可能对你来说是个绊脚石，因为你可能不会真正想要或可能无法让用户将嵌入式设备连接到计算机上。尽管如此，接下来 CDD 必须做的事情有

- 你的设备必须事先一个 USB 客户端，能够通过 USB-A 连接。
- 它必须实现 adb 命令中使用 USB 调试所需要的 Android 调试桥（Android Debug Bridge, ADB）协议。
- 它必须实现 USB 大容量存储器设备（Usb Mass storage），因此允许设备的 SD 卡被主机访问。

新的 CDD 显然已经进一步演化了，自从 3.0 开始就不再需要拥有物理的 Home、Menu 和 Back 按钮了，而现在 OpenGL ES 2.0 成为了强制项。除了 USB Mass Storage 支持以外，设备现在还可以提供媒体传输协议（Media Transfer Protocol, MTP）支持。

## 性能兼容性

虽然 CDD 没有给出 CPU 速度的要求，但它却给出了 app 相关的时间限制，从而影响你选择 CPU 的速度。例如：

- 浏览器启动时间小于 1300ms。
- 彩信 / 短信应用启动时间小于 700ms。
- AlarmClock 时钟应用启动时间小于 650ms。
- 重启某个正在运行的 app 必须少于原启动时间。

## 安全模型兼容性

你的设备必须符合 Android 应用程序框架、Davik 和 Linux 内核所要求的安全环境。具体来说，就是应用程序必须能够访问并且遵从 SDK 文档中所描述的权限规则。应用程序必须被限制在相同的沙箱约束中，即不同的应用程序运行在不同的进程空间中，这些进程在不同的 Linux 用户（UID）身份下。文件系统访问权限必须同样遵守开发者文档的描述。最后，如果你不使用 Davik 虚拟机，你所使用的虚拟机也应当遵守与 Davik 相同的安全行为。

## 软件兼容性测试

你的设备必须通过 CTS，包括人工操作的 CTS 验证者部分。另外，你的设备必须能够运行 Google Play 中特殊的参考应用程序。

## 可更新软件

必须有一种机制来更新你的设备。你可以通过空中下载（OTA），也可以通过重新启动来执行离线更新。它还可以通过 USB 将设备连接到 PC 执行更新，或者使用移动存储器来执行离线更新。

## 相容性测试套件

CTS 是 AOSP 的一部分，而且我们会在第 4 章来讨论如何构建和使用它。在 AOSP 中有一个专门的构建目标是用来产生 CTS 的命令行工具，它是用于控制测试套件的主要工具。CTS 依赖于 adb 通过 USB 来推送和运行测试用例。这些测试用例基于 JUnit Java 测试框架，会检验框架的不同部分，例如 API、Davik、Intent、权限等。测试完成后，它们会生成一个 ZIP 文件，包含 XML 文件和屏幕快照，用于发送到 [cts@android.com](mailto:cts@android.com)。

## 开发工具及其环境搭建

有两套相对独立的 Android 开发工具：用于应用程序的开发和用于平台的开发。如果你要搭建应用程序的开发环境，那么可以看一下《Learning Android》或看看 Google

的在线文档。如果你想要做平台的开发，正如这本书将要讲述的，你的工具会有所不同，内容将会在本书后面介绍。

但是，在最基本的层面上，你需要有一个基于 Linux 的工作站用于构建 AOSP。事实上，在写本书的时候，Google 唯一支持的编译环境是 64 位的 Ubuntu 10.04。这并不意味着另一个 Ubuntu 版本甚至另一个 Linux 发行版将无法正常工作，现实情况是到姜饼为止的 Android 版本，你是无法在 32 位系统上构建 AOSP<sup>注8</sup>。但本质上，这只是反映了 Google 自己的 Android 编译配置情况。在不改变你工作站操作系统的情况下构建 AOSP 的一个简单的方法，是使用你最喜爱的虚拟机工具构建一个 Ubuntu 虚拟机。我通常使用 VirtualBox，因为我发现它可以在客户机操作系统中很容易地访问主机的串口。

---

**注意：**在某些情况下，对于某个特定的 Android 版本，即使没有 32 位的编译环境支持，也存在一些补丁能够使得这样的编译成为可能。至少对于姜饼系统来说是这样的。所以，即使官方的源码树不支持 32 位构建，你也可以找到其他的源码树能够支持它，也有相应的邮件列表的帖子来告诉你如何做到。尽管如此，较新的 AOSP 版本仍然是需要越来越强大的机器来在适当时间内执行完成构建过程，而这些机器通常都是 64 位的。因此，支持 32 位系统构建的动力随着每个新版本 Android 系统发布而逐渐消失。

---

无论你要配置的开发环境是什么，记住 AOSP 在构建前有好几个 GB 大小，而其构建后的最终规模要大得多。例如，姜饼（GingerBread）系统，编译前是大约 3GB 大小，一旦编译就增长到 10GB，而 4.2/ 果冻豆（JellyBean）未编译前是 6GB，而一旦编译就增长到约 24GB<sup>注9</sup>。当你可能要运行几个不同版本时（例如即使没有其他原因，也可能需要用于测试），你可能很快就会意识到你需要几十甚至数百 GB 才能使得几个 AOSP 工作。另外请注意，写这本书期间（2011 年至 2013 年），使用最高端的机器构建最新的 AOSP，一直需要在 30 分钟到 1 小时的时间。即使是轻微的修改，也可能会需要五六分钟时间运行，以完成构建或重新输出镜像。因此，你在开发基于 Android 的嵌入式系统时，可能需要确保有一个相当强大的机器。我们将在第 4 章中讨论 AOSP 的构建过程谈及更详细的要求。

---

注 8：更新的版本例如果冻豆（JellyBean）仅能使用 64 位系统构建。

注 9：这些未编译时空间的数字是在源码树中并没有包含 .git 和 .repo 目录时。包含这些目录时，2.3.7/ 姜饼（Gingerbread）未编译时的大小是 5.5GB，4.2/ 果冻豆（JellyBean）的是 18GB。

# 内部结构入门

如前所述，Android 的源代码提供免费下载，允许修改和安装在任何你选择的设备上。实际上，如果只是简单的获取代码，编译它并在你的 Android 模拟器上运行起来是相当的简单。但是要为你的设备和硬件定制专门的 AOSP，你可能首先必须要对 Android 内部结构有一定的了解。在本章中你将会看到关于 Android 的内部结构的一个高层示意图，并有机会在后面的章节中接触到更详细的内部结构信息，包括将所说的 AOSP 内部结构和实际的源代码联系起来。

---

注意：正如在前言中提到的那样，本书所说的系统主要是 Android2.3/ 姜饼。也可以说，自从 Android 升级到这个版本后，Android 的内部结构在其生命周期中保持得还算稳定，从这个版本到现在的 Android4.2/ 果冻豆改变的很少。不过，尽管内部结构大部分相对来说都保持不变，但是关键的变化能够带来突如其来的惊喜，这要感谢 Android 的封闭式开发过程。比如说，在 Android2.2 以及之前的版本中，状态栏作为系统服务中一个部分与系统服务是一个整体。在 Android2.3/ 姜饼中，状态栏从系统服务中分离出来，现在是独立运行的一个部分。<sup>注 1</sup>

---

## 应用程序开发者的观点

考虑到 Android 开发的 API 不同于其他已存在的 API，它包含 Linux 世界中发现的任何东西，花费一些时间从开发者的角度来理解“Android”的含义是非常重要的，虽然与正在破解 AOSP 的人理解的 Android 非常不一样。作为一个主要开发设备上的

---

注 1：有些人猜测，这种变化的产生是因为有一些应用程序开发人员在通知栏上做了太多花哨的东西，对系统服务产生了负面影响，Android 开发团队因此决定将状态栏从系统服务中分离出来成为单独的部分。

嵌入式 Android 系统的嵌入式开发者，你可能不会直接接触 Android 应用程序开发的 API 的特性，但是你的一些同行会接触到。如果不出意外，你可能也会跟其他应用程序开发人员一样都叫做应用程序开发者。当然，本节仅仅是一个总结，我建议你读一些 Android 应用程序开发的文章或书籍，可以获得更深一层的了解。

## Android 的概念

应用程序开发者在开发应用程序的时候，有几个关键的概念必须考虑到。这些概念定义了所有 Android 应用程序的体系结构，并规定开发者能够做什么不能够做什么。总而言之，它们让用户们的生活变得更美好，但是有时候又是有挑战性的。

### 组件

Android 应用程序由一些松散的组件组成。一个应用程序的组件可以涉及或者是用到其他应用程序的组件。最重要的是，一个 Android 的应用程序没有一个统一的入口：没有 `main()` 函数或者其他同等的函数。相反，有预定义的事件叫做意图（intents），开发者可以将他们的组件绑定到上面，从而使他们的组件在当前事件中被激活。举一个简单的例子，处理用户联系人数据库的组件，它是当用户在拨号器应用或其他应用程序中按下联系人按钮时调用的控件。因此，一个应用程序可以有与它的组件同等数量的入口。

主要有四个类型的组件：

#### 活动（Activities）

就像“窗口”是基于 Windows 的 GUI 系统中所有视觉交互的主要构件块，活动就是 Android 应用程序中的主要构件块。然而与窗口不同，活动不能被“最大化”，“最小化”或是“调整大小”。相反，活动通常是占满整个界面的，并且是一个堆叠一个进行存放，这种方式与一个浏览器为了记住它所访问过的网页而将它们都放在一个队列里一样，用户可以回到他之前访问过的界面。实际上，就像前面章节介绍的那样，所有的 Android 设备都有一个返回按钮，不管它是一个设备上的实际按钮还是显示在屏幕上的软按键，对用户来说都是有效的。尽管与网页浏览相比，没有相应的按钮对应“前进”浏览行为；只有“后退”是可以实现的。

一个全局定义的 Android Intent，允许一个 Activity 显示为应用程序启动器（设备上主要的应用程序列表）中的一个图标。由于绝大多数的应用程序想要出现在主应用程序列表中，它们至少要有一个活动是被定义为能够回应这个 Intent 的。通常情况下，用户将从一个特定的 Activity 开始启动，通过几个其他的 Activity

并以创建一个与他们最初启动的那个 Activity 相关的所有 Activities 的栈来结束，这个 Activities 栈就叫做任务（task）。用户可以通过按“Home”键来切换到另一个任务，这时开始从应用程序启动器创建另一个 Activity 栈。

## 服务（Services）

Android 的服务类似 UNIX 中的后台进程或者守护进程。从本质上来说，当其他组件需要一个服务并且由于它的调用者持续的需要使得它通常要保持活跃时，这个服务被激活。最重要的是，服务能够被一个应用程序外部的组件激活，从而暴露这个应用程序的一些核心功能给其他的应用程序。一个服务被激活时通常没有看得到的信号发出。

## 广播接收器（Broadcast receivers）

广播接收器类似中断处理程序。当一个关键的事件发生时，一个广播接收器被触发，代表这个应用程序来处理这个事件。例如，当电池电量很低时或者当“飞行模式”（关闭无线连接）开启时，一个应用程序可能希望得到通知。当广播接收器没有在处理一个它们注册了的特定事件时，它是无效的。

## 内容提供者（Content providers）

内容提供者本质上是数据库。通常，如果需要让数据可以被其他应用程序访问的话，一个应用程序将包括一个内容提供者。比如说，如果你正在构建一个 Twitter 客户端应用程序，可以通过一个内容提供者，给设备上想要访问 tweet 的其他应用程序提供你要展示给用户的信息。所有的内容提供者提供给应用程序相同的 API，不管它们实际上内部是怎么实现的。大部分内容提供者是基于包含在 Android 里的 SQLite 功能，但是也可以用文件或者其他方式进行数据存储。

## Intent

Intent 是 Android 里最重要的概念之一。它们采用的是后期绑定的机制，可以与组件进行交互。一个应用程序开发者可以发送一个活动的 Intent 来“浏览”网页或者“查看”一个 PDF 文件，从而使用户可以查看一个指定的 HTML 或者 PDF 文件，即使所请求的应用程序本身没有包含这样的功能。Intent 还有更神奇的用法。比如说，一个应用程序开发者可以发送一个指定的 Intent 来触发一次通话。

可以把 Intent 想象或是多态的 UNIX 信号，且不必进行预定义或指定特定的目标组件或应用程序。如果你熟悉 Qt，可以把 Intent 想象成一个 Qt 信号，虽然它们并不是完全一样。Intent 本身是一个被动的对象。其发送所带来的影响取决于它的内容，发送它所采用的机制，系统的内置规则以及所安装的应用程序集。比如说，系统规则其中的一条是，Intent 绑定的类型是它要发送到的组件的类型。比如说发送到服务的 Intent，只能由一个服务来接收，不能被一个 Activity 或者是一个广播接收器来接收。

组件可以在 *manifest* 文件中通过过滤器声明为能够处理给定的 Intent 类型。此后系统将找到这个过滤器匹配的 Intent，并触发相应的组件运行。这通常叫做一个“隐式”的 Intent。一个 Intent 也可以用一种“显式”的方式发送到一个特定的组件，不用在接收组件的过滤器中对这个 Intent 进行声明。由于这种显式的调用需要应用程序提前知道这个特定的组件，通常只适用于 Intent 是在同一个应用程序的组件间发送。

## 组件的生命周期

Android 的另一个核心宗旨是用户不用管理任务的切换。虽然有很多种方式进行任务的切换，包括一种内置机制，典型的做法就是长时间地按住“Home”键，还有就是一组在 Android 中使用的任务管理器应用程序，用户的体验并不是基于此。相反，用户希望尽可能多地打开想要打开的应用程序，并且通过按“Home”键返回主界面再单击打开其他应用程序来在它们之间进行切换。用户单击打开的应用程序可能是一个新的应用，或者是之前打开过的且在活动栈（又叫做“任务”）中已经存在的应用。

我们推测，之所以会有这样的设计决策是因为，应用程序被打开后会逐渐占用越来越多的系统资源，一个进程不可能一直持续运行下去。在某些时候，系统将不得不开始回收最近很少用或者是优先级较低的组件的资源，这也是为新激活的组件让路。资源回收应该是完全对用户透明的。换句话说，当一个组件为了给新的组件让路而被撤下时，用户想要回到原来的那个组件时，这个组件应该从它被撤下时的那个状态开始，就好像它一直是在内存中等着被启动一样。

为了实现以上的情形，Android 为每一种组件类型定义了标准的生命周期。一个应用程序开发者必须通过为每一个组件实现一系列的回调函数，来管理他的组件的生命周期。这些回调函数通常被与组件生命周期相关的事情触发。比如说，当一个活动不再在前台运行（并因此比在前台运行时更容易被回收资源），它的 `onPause()` 回调函数就会被触发。Google 采用了一种状态图 (<https://developer.android.com/images/training/basics/basic-lifecycle.png>) 来向应用程序开发者解释活动的生命周期。

管理组件的生命周期是应用程序开发者面临的最大挑战之一，因为他们必须小心地保存和恢复关键的过渡事件的组件状态。最理想的结果是用户从来不需要进行应用程序间的“任务切换”，或者是了解之前所用的应用程序的组件为了给他新开启的应用让路，已经被销毁了。

## Manifest 文件

如果一个应用程序必须要有一个“主”入口点，那么 manifest 文件就类似这个功能。基本上，它通知系统应用程序的组件运行所需要的功能，所需要的 API 的最低级别，

以及所需要的所有硬件等。`manifest` 文件采用的是 XML 文件格式，并处于应用程序源代码目录的最顶层，命名为 `AndroidManifest.xml`。在 `manifest` 文件中应用程序的组件通常被描述为静态的。实际上，除了广播接收器可以实时注册以外，所有其他组件必须在构建时就在 `manifest` 文件里声明。

## 进程和线程

不管应用程序的组件什么时候被激活，不管它是由系统或是其他应用程序激活，一个进程会被创建来储存应用程序的组件。并且除非是应用程序开发者做了什么来重写系统的默认值，否则这个应用程序的所有在初始组件激活以后建立的组件都会与初始组件运行在同一个进程中。换句话说，一个应用程序所有的组件都包含在一个单独的 Linux 进程中。因此，开发者应该避免在标准组件中有时间消耗长的或者是会出现阻塞的操作，而可以使用线程来实现。

理论上来说，用户可以尽可能多地激活他想要激活的组件，通常都会有几个 Linux 进程是随时处于激活状态的，为很多应用程序包括用户的组件服务。当有太多的进程运行而无法有资源让新的进程开始时，Linux 内核的处理内存匮乏（OOM）查杀机制将开始启动。在这个时候，Android 内核中的 OOM 处理器会被调用，并且它将决定哪些进程必须被杀死以腾出资源。

简而言之，所有 Android 行为都是以占用低内存为条件的。

如果进程被 Android 的 OOM 处理器杀死的应用程序的开发者可以恰当地定义他的组件的生命周期，用户应该看不到任何不友好的行为。事实上，在实际情况中，甚至都不应该让用户注意到封装应用程序组件的进程停止并且在之后不久又重新“自动”启动了。

## 远程过程调用（RPC）

像很多系统的其他组件一样，Android 定义了它自己的 RPC/IPC（远程过程调用 / 进程间通信）机制：*Binder*。所以组件之间的通信通常不使用普通的套接字（socket）或者是 System V IPC。相反，组件使用内核中的 Binder 机制，通过 `/dev/binder` 访问，这部分内容将在本章的后面介绍。

然而应用程序开发者并不直接使用 Binder 机制。相反，他们必须使用 Android 的接口定义语言（IDL）来定义接口并与它进行交互。接口的定义通常是存储在一个 `.aidl` 文件中，并且用 `aidl` 工具来处理，生成适当的存根（stub）和序列化 / 非序列化的代码，以便于使用 Binder 机制向后或向前传输对象或者数据。

## 框架概述

除了我们刚才讨论的概念，Android 也定义了自己的开发框架，它允许开发者访问在其他开放框架中常见的功能。让我们简单介绍这个框架及其功能。

### 用户界面

Android 中的 UI 元素包括传统的部件比如说按钮、文本框、对话框、菜单和事件处理程序。这部分的 API 相对简单，如果开发者已经编码过其他 UI 框架，则他们通常能够找到他们自己的方法来很容易地解决问题。

Android 中所有的 UI 对象都是作为 View 类的子类创建的，并且都是属于 ViewGroups 这个层次。一个 Activity 的 UI 实际上既可以在 XML 文件中静态声明（这是常用的方法），也可以用 Java 动态地声明。如果必要，UI 也可以用 Java 语言实时修改。如果一个 Activity 的 UI 的内容被设置在 ViewGroup 层级的根目录下面，那么它就会展现在屏幕上。

### 数据存储

Android 为开发者提供了几种存储方式。对于简单的存储需求，Android 提供了 *shared preferences*，允许开发者将键 / 值对或者存储在应用程序所有组件共享的一个数据集中，或者存储在指定的单独的文件中。开发者可以直接操作文件。这些文件可能被应用程序私有存储，所以它们是不能够被其他应用程序访问的，也不可以被其他应用程序读 / 写。应用程序开发者也可以用 Android 里的 SQLite 功能来管理他们自己的私有数据库。将这样一个数据库托管在一个内容提供者组件中将使得它对于其他应用程序来说也是可用的。

### 安全和许可

Android 的安全是在进程层实现的。换句话说，Android 是依赖于 Linux 的现有的进程隔离机制来实现它自己的策略。到最后，每个安装的应用程序获得一个自己的 UID 和组标识符 (GID)。从本质上来说，就好像每个应用程序是系统中一个独立的“用户”。而在任何的多用户 UNIX 系统中，这样的“用户”不能够访问其他用户的资源，除非获得了明确的权限来这样做。实际上，每个应用程序运行在它自己独立的空间里。

想要走出自己的空间去访问重要的系统功能或资源，应用程序必须使用 Android 的权限机制，这个机制需要开发者在该应用程序的 manifest 文件中静态地声明所需要的权限。有一些权限是由 Android 预定义的，比如说访问网络的权限（例如使用 socket），拨打电话，或者是使用相机。其他权限可以由应用程序开发者自己声明，之后被其他的应用程序用于与这个给定的应用程序的组件进行交互。当一个应用程序被安装时，会提示用户是否允许运行该应用程序所需的权限。

访问的执行是基于每一个进程的操作和对要访问的特定的 URI (通用资源标识符) 的请求，是否获得对一个特定的功能或者资源的访问的决定是基于证书以及用户请求的。这个证书就是应用程序开发者用于对应用程序进程签名以便于他们的应用程序通过 Google Play 被下载使用时所用的证书。因此，开发者可以约束他们自己之前所开发的其他应用程序对其功能的访问。

Android 开发框架提供了更多的功能，当然，我们在这里就不一一介绍了。我推荐大家阅读其他人的 Android 应用程序开发或者是访问网址 <http://developer.android.com> 来获得更多关于 2D 和 3D 图形、多媒体、位置和地图、蓝牙、NFC 等的信息。

## 应用程序开发工具

开发 Android 应用程序的典型方法是使用免费提供的 Android 软件开发包 (SDK)。这个 SDK [连同 Eclipse 及其相应的 Android 开发工具 (ADT) 插件，以及 SDK 中基于 QEMU 的模拟器] 允许开发者可以直接在他们自己的计算机终端上进行大部分的开发工作。在将应用程序放到 Google Play 里供别人下载使用前，开发者也通常会希望在实际的设备上测试他们的应用程序，通常应用程序运行在模拟器上和实际设备上会有一些行为上的差别。一些软件发行商将此做到了极致，它们在一个新的版本出货前会在几十台设备上测试它们的应用程序。

### 在几百台设备上做测试

显然，应用程序开发者不可能在他们的测试中安排对每一个可能的设备进行测试。因此，一些公司如雨后春笋般冒了出来，它们可以让应用程序开发者在几百台设备上测试他们的应用程序，只要简单地将他们的应用程序上传到这些公司的网站上就可以了。

这些公司通常都有一个网络接口，允许开发者通过这个接口上传他们要执行的应用程序到它们的设备场里。之后开发者就会得到关于出现的故障的详细报告，有时候还会相当明确的输出出现故障的设备上的日志。如果你想要这样的功能，可以看看 Apkudo (<http://www.apkudo.com/>)，Bitbar's Testdroid products (<http://testdroid.com/>) 和 LessPainful (<http://www.lesspainful.com/>)。

有趣的是，Apkudo 还提供了一种服务，它允许你在测试你的版本前先测试设备，通过运行几百个流行的应用程序来确定它运行 AOSP 是正常的。

即使你不打算在你的嵌入式系统上开发任何的应用程序，我还是强烈建议你在你的工作计算机上安装开发环境。如果不出意外，这可以让你用一些基础的测试应用程序来验证你对 AOSP 所做的修改。如果你打算扩展 AOSP 的 API 并创建和发布你自己定义的 SDK，安装开发环境也是必不可少的。

想要安装一个应用程序开发环境，可以按照 Google 提供的关于 SDK 的说明，或者看一下 Marko Gargenta (O'Reilly 出版) 写的书《Learning Android》 (<http://shop.oreilly.com/product/0636920010883.do>)。

## 本地开发环境

虽然大多数的应用程序都是在我们刚刚提到的开发环境中用 Java 来开发，但是也有某些开发者需要运行本地编译的代码。为此，Google 为他们提供了本地开发套件 (NDK) (<http://developer.android.com/tools/sdk/ndk/index.html>)。这个套件主要针对游戏开发者，他们需要充分用尽游戏运行设备的所有性能。正因为如此，NDK 中可用的 API 大多是面向图形绘制和传感器输入检索。比如说很著名的愤怒的小鸟 (Angry Birds) 游戏很大程度上依赖于本地运行的代码。

另一种可能用到 NDK 的地方显然是将已存在的代码库移植到 Android 上。如果你在几年间已经开发了大量遗留的 C 代码（一种常见的情况是之前开发过其他移动设备的应用程序），你不用将它们用 Java 重写。相反，你可以用 NDK 编译它们，并把它们与一些 Java 代码打包在一起，以便能够使用 SDK 提供的更多 Android 特定的功能。比如说 Firefox 浏览器，在很大程度上是依赖于 NDK 来使得它们的本地代码能够在 Android 上运行。

就像我刚刚介绍的那样，NDK 最好的功能是你可以将它与 SDK 结合在一起，那么你的应用程序里就有一部分代码是 Java 写的，一部分是 C 写的。不过很重要的是你要了解，NDK 只给你非常有限的访问部分 Android API 的权限。也就是说，目前还没有 API 可以允许你从 NDK 编译后的 C 代码中发出一个意图 (intent)，相反只能用 Java 的 SDK 才能够做到。再次说明，NDK 提供的 API 主要还是面向游戏开发。

有时候嵌入式开发和系统开发的开发者转入 Android 开发后，想要能够用 NDK 来实现平台级的工作。NDK 这组词中的“本地”在这方面可能会有误导，因为 NDK 的使用还涉及所有约束 Java 应用程序开发者的限制和要求。所以，作为一名嵌入式开发者，请记住 NDK 是让应用程序开发者用于运行他们的本地代码，以便他们能够调用 Java 代码的。除此之外，NDK 将对于你现在进行的工作几乎没有任何作用。

# 整体架构

图 2-1 可能是本书中给出的最重要的图表之一，我建议你找一个书签来标记这个位置，因为我可能会经常明里暗里地提到它。虽然它只是一个简单的视图（根据我们提到的内容，我们将会来充实它），它很好地展现了 Android 的结构，也介绍了各模块之间的组合方式。

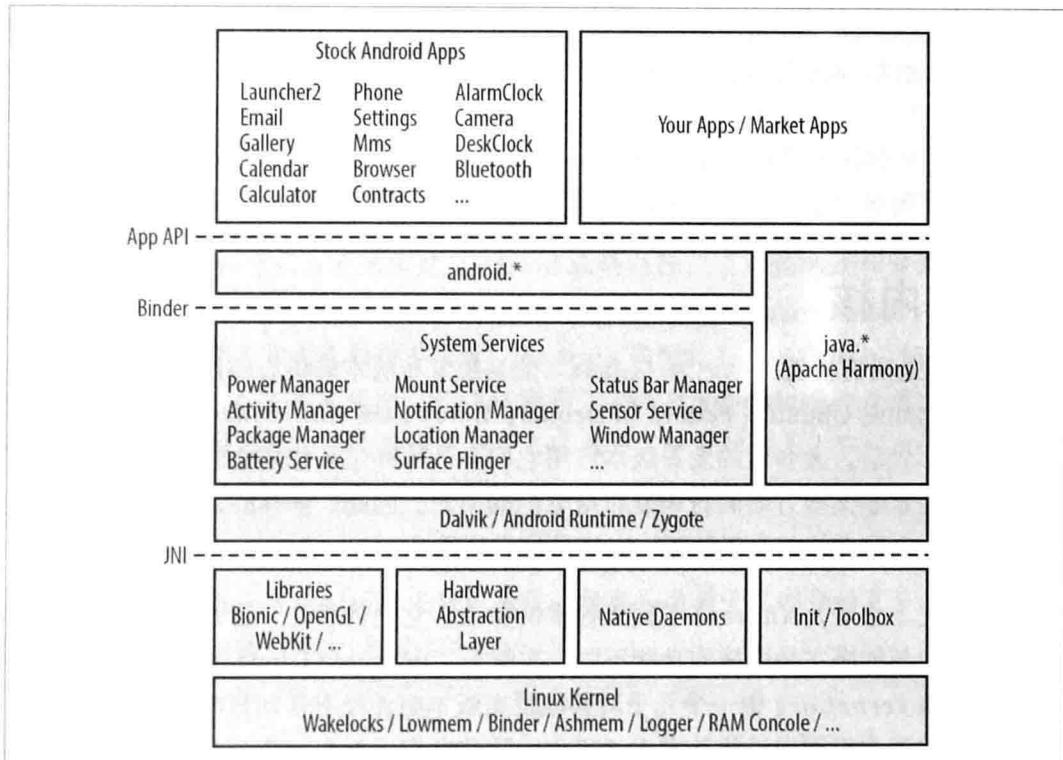


图 2-1: Android 的结构

如果你熟悉某种形式的 Linux 开发，那么首先第一件打击你的事就是在 Linux 内核之外，这个框架里很少有类似常用在 Linux 或者是 UNIX 开发中出现的东西。没有 glibc，没有 X Windows 系统，没有 GTK，没有 BusyBox，没有 bash shell 等。很多经验丰富的 Linux 和嵌入式 Linux 从业者甚至指出，Android 感觉非常陌生。虽然 Android 框架开始时在用户空间方面是一张白纸，我们将在附录 A 中讨论怎样得到“传统的”或者是“经典的” Linux 应用程序和工具，使之与 Android 框架一起共存。

---

**注意：**Google 的开发者文档中给出了与图 2-1 不同的结构图。前者可能非常适合于应用程序开发者，但是它忽略了嵌入式开发者必须要知道的关键信息。比如说，Google 的图和开发者文档，在写本书的时候，几乎很少提到系统服务。然而，作为一名嵌入式开发者，你需要知道那个组件是什么，因为它是 Android 中最重要的部分之一，你可能需要扩展它或是直接与它交互。

你可能会看到 Google 的图在一些文档或者是演示文稿中被展示或是复制，所以尤其重要的是你要理解这个图。如果没有错，请记住，系统服务的内部构件和意义如果要都解释给应用程序开发者听到是几乎不可能的，里面的大部分信息主要是针对应用程序开发者的，而不是做平台工作的开发者。

---

从图 2-1 的底部往上说起，让我们更深入地了解 Android 体系结构的每个部分。一旦我们完全了解各个组件，将会通过介绍系统的启动程序来结束这一章。

## Linux 内核

Linux 内核是所有传统上标记为“Linux”的系统发布版本的核心部件，包括主流的发布版本，比如说 Ubuntu、Fedora 和 Debian。Linux 内核 (<http://kernel.org/>) 发布为纯净主线形式后，大多数的发布版本应用它们自己的补丁来修复它的漏洞并增强了性能，或者是定制某些方面的行为发布给它们的用户。因此，同样的，Android 开发者们也为纯净主线内核打上了补丁，以满足他们的需求。

但是从历史上来看，Android 不同于标准的做法，它是依靠几个自定义的功能，这些功能明显地与纯净主线内核的功能不同。实际上，由于一个 Linux 发布版本装载的内核很容易被 *kernel.org* 中一个几乎对其他发布版本组件没有任何影响的内核替换，而 Android 的用户空间组件将基本上不工作，除非它们运行在一个“Android 版本”的内核上。在前面的章节中也提到，直到最近，Android 内核还不是主线内核中的重要部分。我还提到过，这种情况已经进步了很多，很多运行 Android 所需要的功能已经找到了它们进入主线内核的方式。

---

**注意：**希望当你读本书的时候，上面的情况可能已经进展顺利，你可以直接从网页 <http://kernel.org> 上获得一个内核，并且可以在这个内核上运行 AOSP。但是，如果过去的事只是一个前奏，嵌入式 Linux 的历史只是告诉我们它是怎么来的，那么你获得一个正确的、Android 兼容的内核以使它能够在你的硬件上运行的最好方式，可能是从你所用的 SoC 的厂商那里得到。

---

虽然 Linux 内核内部结构的讨论已经超出了本书的介绍范围，但还是让我们重新看一下加入到内核中的“Androidisms”。你可以从 Robert Love 写的《Linux Kernel

Development, 3rd》（Addison-Wesley Professional, 2010 年出版）这本书中获得关于内核内部结构的信息，也可以开始追看 Linux 每周新闻（LWN）网站。LWN 里有几篇关于内核内部结构开创性的文章，并提供有关 Linux 内核发展的最新消息。

请注意，以下几个段落介绍的仅仅是最重要的 Android 修改。Android 的内核通常包含几百个补丁，超过了标准的内核，它经常提供特定于设备的功能、修复和增强。你可以用 Git<sup>注 2</sup> 来对网页 <http://android.googlesource.com> 中的某一个内核和将它分叉出来的主线内核之间在提交增量方面做一个详尽的分析。同时，请注意在一些 Android 内核中的一些功能（比如说 PMEM 驱动）是针对特定设备的，并且没有必要用于所有的 Android 设备。

## 构建你自己的 Android 内核

如果你想知道如何从头开始构建 Android 版的内核，或者你的工作就是要构建它，或是因为你在为 SoC 供应商工作，那么可以看一下 Linaro 的工程师 Vishal Bhoj 在 2012 年 3 月发表的博客文章“Android 定制的 Linux 内核” (<http://bit.ly/16elk5l>)。在这篇文章中，Vishal 解释了怎么用 `git rebase` 命令来创建一个 Android 定制的内核。想要知道更多有关这些特定命令的信息，可以看一下相应的在线 Git 文档 (<http://git-scm.com/book/en/Git-Branching-Rebasing>)。

顺便说一句，Linaro 公司，其职责是为它的成员公司提供平台支持，维护着一种紧随 Linus 的想法的 Android 定制的内核。想要知道关于这部分的更多信息，可以看一下这个链接 (<http://bit.ly/Z5yGIm>)。

## wakelocks

对于所有的 Androidisms 来说，wakelocks 可能是最有争议的一点。对于它讨论的程度甚至超过了主线内核，产生了将近 2000 封电子邮件，即使是这样也没有讨论出明确的路径来合并 wakelocks 功能。2011 年的内核峰会后，内核开发者同意将大部分的 Androidisms 合并到主线中去，也做了很多努力来改造 wakelocks 机制，或者就像最后决定的那样，创建一个对于其他内核开发团体来说更容易接受的等效机制。

截至 2012 年 5 月底，等效于 wakelocks 的机制和它们相关的 early suspend 机制已经被加入到了主线线程里。提早暂停（early suspend）被替换为自动休眠（autosleep），

注 2：Git 是一个分布式源代码管理工具，它是由 Linus Torvald 开发的，用于管理内核代码。你可以在网页 <http://git-scm.com/> 上找到更多的信息。

而 wakelocks 机制则被替换为 `epoll()` 标志，叫做 EPOLLWAKEUP。API 也因而加入了不同于原来的功能，但是最终功能的效果还是一样。在写本书的时候，预期新的 AOSP 版本会开始使用新的机制来代替旧的机制。

想要了解 wakelocks 是什么和用来干什么，我们必须首先讨论 Linux 中的电源管理通常是怎么工作的。最常用到 Linux 的电源管理的例子是一台笔记本电脑。当一台运行 Linux 系统的笔记本电脑的盖子被合上时，它通常会进入“暂停”或是“休眠”模式。在这种模式下，系统的状态被保存在 RAM 中，但硬件的所有其他部分都是停止工作的。因此，笔记本电脑此时使用的电量会尽可能的少。当盖子被打开以后，这台笔记本电脑就“醒来”了，并且用户几乎可以瞬间恢复使用。

这种方法在笔记本电脑和桌面计算机设备使用得很好，但是并不适合移动设备，比如说手持设备。因此，Android 的开发团队设计了一种机制，对规则进行了些许变化以使得它更适合于移动设备。不是当用户发出命令时才让系统进入休眠状态，Android 定制的内核让系统尽快且尽可能经常地进入休眠。当正在处理重要的进程或者是一个应用程序正在等待用户的输入时，才阻止系统进入休眠状态，wakelocks 被用于使系统保持唤醒状态。

wakelocks 和提早暂停功能实际上是构建在 Linux 已存在的电源管理功能之上的。然而，它们引入了一个不同的开发模型，因为原来的功能要求应用程序和驱动开发者必须明确地抓取 wakelocks，即使是他们正在进行关键操作或者是必须要等待用户输入时。通常，应用程序开发者并不需要直接地处理 wakelocks，因为他们所使用的抽象对象自动地处理了需要的锁定操作。尽管如此，如果他们需要显式的 wakelocks 的时候，还是可以与电池管理服务进行通信。另一方面，驱动开发者能够调用内核里增加的 wakelocks 基元来获取和取消 wakelocks 功能。然而，在一个驱动程序中使用 wakelocks 的缺点是，使得这个驱动不可能被推到主线内核上，因为主线内核中不包含对 wakelocks 的支持。鉴于现在 wakelocks 的等效功能可以加入到主线内核中，因此这不再是一个问题。

---

**注意：**下面的 LWN 文章更详细地描述了 wakelocks，解释了围绕将它加入到主线内核中遇到的各种问题：

- Wakelocks 和嵌入式问题 (<http://lwn.net/Articles/318611/>)
  - 从 Wakelocks 到一个实际问题 (<http://lwn.net/Articles/319860/>)
  - Suspend 块 (<http://lwn.net/Articles/385103/>)
  - Blocking suspend blockers (<http://lwn.net/Articles/388131/>)
-

- 
- What comes after suspend blockers (<http://lwn.net/Articles/390369/>)
  - An alternative to suspend blockers (<http://lwn.net/Articles/416690/>)
  - KS2011: Patch review (<http://lwn.net/Articles/464298/>)
  - Bringing Android closer to the mainline (<http://lwn.net/Articles/472984/>)
  - Autosleep and wake locks (<http://lwn.net/Articles/479841/>)
  - 3.5 merge window part 2 (<http://lwn.net/Articles/498693/>)
- 

## 低内存管理

正如前面提到的，Android 的行为是以低内存条件为前提的。因此，内存溢出是很严重的。基于这个原因，Android 的开发团队在内核中增加了一个额外的低内存管理（low-memory killer），让它在默认的内核的 OOM 管理之前起作用。Android 的低内存管理应用了应用程序开发文档中的政策，淘汰掉占领组建很长一段时间没有使用的和优先级不高的进程。

Android 低内存管理是基于 OOM 的机制，在它的基础上做了更适合于 Linux 的调整，使得不同的进程有不同的 OOM 管理优先级。基本上，对 OOM 的调整包括允许用户空间控制一部分内核 OOM 管理的政策。OOM 调整的范围从 -17 ~ 15，越大的数字表示相关的进程在内存溢出时更容易被杀死。

因此，Android 根据不同进程所运行的组件和它自己的低内存管理为每一个类型的进程设置阀值的配置，给不同类型的进程分配了不同的 OOM 调整值。这有效地使它能够在内核自带的 OOM 管理（只有当系统没有内存的时候才会被激活）激活前发挥作用，只要达到了它设置的阀值，而不需要到系统的内存被消耗尽的时候。

用户空间的策略是它们自己是在启动时的初始化进程中应用的（见本章后面的“初始化”），在运行时间内被活动管理服务（系统服务的一部分）调整并部分执行。活动管理是系统服务中最重要的服务之一，它负责（还有很多其他事）实现前面提到的组件的生命周期。

---

**注意：**如果你想要获得更多关于内核的 OOM 管理和 Android 是如何在它的基础上构建的等方面的信息，你可以看一下“Taming the OOM killer” (<http://lwn.net/Articles/317814/>) 这篇 LWN 文章。

---

在写本书的时候，Android 的低内存管理器与其他很多 Android 特定驱动程序一样，都在内核的阶段源码树中。目前这个功能正处于重写过程中，采用了一种更通用的框

架来处理低内存情况。感兴趣的读者可以看看 Linux 内核邮件列表（LKML）上发布的“用户空间低内存管理守护进程”（<http://lwn.net/Articles/511731/>）这篇文章，以及 Linux-vmevent（<http://git.infradead.org/users/cbou/linux-vmevent.git>）补丁上关于低内存管理守护程序的工作进展，其目标是将低内存情况下的决策过程迁移到用户态的守护进程中。

## Android 和 Linux 的阶段源码树

在写本书的时候，许多运行 Android 所需要的驱动程序已经合并到内核的阶段源码树中了。这就意味着它们仍然可以在主线内核（网页 <http://kernel.org> 上可下载）中找到，这也意味着内核开发者相信，这些驱动程序应该处于在它们被认为足够成熟来与它们周围内核树中其余部分“干净”的驱动程序合并之前。

具体来说，目前很多的 Android 驱动程序放在 *drivers/staging/android* 的内核目录中。它们应该会一直放在这个目录里，直到它们的代码被重构或者重写，以符合被官方 Linux 驱动程序承认并放在 *drivers/* 目录的相关位置中这个条件为止。

如果你不是很熟悉阶段源码树，可以看一下 Greg Kroah-Hartman<sup>注3</sup> 在 2009 年 3 月开始发表的博客文章“The Linux Staging Tree, what it is and is not”（<http://www.kroah.com/log/linux/linux-staging-update.html>）：“Linux 的阶段源码树（或者只是从现在开始‘分阶段’）是用来保持独立的驱动程序以及文件系统，这些驱动程序和文件系统由于很多技术原因，现阶段还不打算合并到 Linux 内核树的主要部分中去。Linux 的阶段源码树包含在主要的 Linux 内核树中，使得用户可以比以前更方便地访问这些驱动，也为开发提供了一个共同访问的地方，解决了以前很多阶段源码树外的驱动程序都存在的‘上百个不同的下载网址’的问题。”

## Binder

Binder 是一种 RPC/IPC 机制，类似 Windows 下的 COM 组件。实际上其根源可以追溯到 Be 的资产被 Palm 收购之前所做的关于 BeOS 的工作。它在 Palm 中继续得到发展，其工作成果最终被发布为 *OpenBinder* 项目。虽然 OpenBinder 从来没有被当作是一个独立的项目，开发它的几个关键开发者，比如 Dianne Hackborn 和 Arve Hjønnevåg，最后离开了 Android 开发团队。

注 3：Greg 是顶级内核的开发者和维护者之一。

因而 Android 的 Binder 机制是由之前的工作延伸开来的，但是 Android 对它的实现并不是来自于 OpenBinder 的源代码。相反，它是对 OpenBinder 功能的一个子集完全的重写。OpenBinder 文档保留了一个必读文件，如果想了解这个机制的基础和设计理念，你可以看看 Dianne Hackborn 关于 Binder 是怎样在 Android 中实现的解释，LKML（内核邮件列表）也可以参考。

从本质上来说，Binder 试图提供在一个传统操作系统上的远程过程调用。换句话说，Binder 不是要重造传统的操作系统，而是“试图包含并超越它们。”因此，开发者可以更方便地将远程服务作为一个对象来处理，而不用处理一个新的操作系统。而扩展系统功能变得非常容易，通过增加远程调用的对象就可以实现，而不用像 UNIX 中通常用的那样执行新的守护进程来提供新的服务。因此这个远程对象可以用任何你想要用的语言来实现，并且可以跟其他的远程服务共享相同的进程空间，或者有它自己独立的进程。调用这个方法，所有要做的事情就是接口定义和一个对它的引用。

在图 2-1 中可以看到，Binder 是 Android 体系结构的基石。它允许应用程序与系统服务进行对话，应用程序调用它来与每一个其他的服务组件进行对话，不过就像我前面提到的，应用程序开发者实际上并不直接用 Binder。相反，他们调用由 *aidl* 工具生成的接口和存根。即使应用程序与系统服务相连接时，*android.\** 的 API 抽象了系统服务，开发者实际上从没见过 Binder 被调用。

---

**警告：**虽然听起来它们的语义相似，但是系统服务中运行的服务和开放给其他应用程  
序的服务之间有很大的不同，后者通过我在本章前面“组件”一节中介绍的“服务”组件  
模式，成为应用程序开发者可用的组件之一。最重要的是，服务组件和任何其他组件  
一样都从属于一样的系统机制。因此，它们是通过生命周期管理的，运行在同样的特  
定的与它们的一部分应用程序相关的沙箱里。另一方面，运行在系统服务里的服务，  
通常与系统权限一起运行，运行时间从开机引导时到重启时。这两种服务只共享以下  
的东西：①它们的名字；②使用连接器（Binder）与他们进行交互。

---

Binder 机制部分内核中的驱动是通过 */dev/binder* 访问的字符驱动。它通过调用 *ioctl()* 函数在通信双方之间传递数据包。它也允许进程指定它本身为“上下文管理器”。上下文管理器的重要性以及 Binder 驱动程序实际所用的用户空间，将会在本章的后面部分详细讨论。

在发布 Linux 内核 3.3 版本时，Binder 驱动加入到了阶段源码树中。目前还没有程  
序整理这个驱动或者是重写它，使得它适合于和 / 或能够用于更多标准的 Linux 桌  
面和服务系统中一般用途上。因此在可预见的将来，它可能会仍保留在目录 *drivers/*  
*staging/android/* 中。

## 匿名共享内存 (ashmem)

另一个在大多数操作系统中用到的 IPC 机制是共享内存。在 Linux 中，这通常是由 POSIX SHM 功能提供，这个功能是 SystemV 中的 IPC 机制的一部分。然而，如果看过 AOSP 中的 *bionic/libc/docs/SYSV-IPC.TXT* 文件，你会发现 Android 开发团队似乎不喜欢 SysV IPC。实际上，这个文件引起了争议，关于使用 Linux 中的 SysV IPC 机制可能会导致内核中的资源泄露，而对于会导致系统瘫痪的恶意或者行为不当的软件是无能为力的。

虽然这不是由 Android 开发者或者 ashmem 代码及其周边应用的任何文档描述的，但 ashmem 中很可能不会有部分 Android 开发团队发现的 SysV IPC 的缺陷。因此 Ashmem 虽然类似 POSIX SHM，“但是有不同的行为”。比如说，它使用引用计数来销毁占用内存的进程退出时的内存区块，并在系统需要内存时缩小到一个映射区域内。“非固定的”内存区块允许内存能够映射缩减，而“固定的”内存区块不允许内存映射缩减。

通常情况下，第一个进程用 ashmem 创建一个共享内存区域，并用连接器来与其他希望共享内存区域的进程共享相应的文件描述符。比如说 Dalvik 的 JIT 代码缓存是通过 ashmem 提供给 Dalvik 实例的。很多系统服务组件，比如说 Surface Flinger 和 Audio Flinger 都是依赖于 ashmem 的——通过 IMemory 接口而不是直接调用。

---

**注意：**IMemory 是 AOSP 中一个只在内部可用的接口，并不能被应用程序开发者调用。而开放给应用程序开发者的最接近的类是 MemoryFile。

---

在写本书的时候，ashmem 驱动已经包含在主线的 *drivers/staging/android/* 目录中，并且将要被重写了。

## alarm

alarm 驱动程序被加入到内核中，是另一个现有的默认的内核功能不能满足 Android 要求的例子。Android 的 alarm 驱动实际上是内核现有的实时时钟 (RTC) 和高分辨率计时器 (HRT) 功能上的。内核的 RTC 功能为驱动开发者提供了一个框架，来创建特定的 RTC 功能，而内核通过主 RTC 驱动程序开放一个单一的与硬件无关的接口。另一方面，内核的 HRT 功能允许调用者在一个非常特定的时间进行调用。

在主线版本的 Linux 中，应用程序开发者通常调用系统调用 `setitimer()` 函数来在一个给定的时间值到来时获得一个信号，想要获得这方面的更多信息，可以看一下 `setitimer()` 的手册页。这个系统调用允许少数类型的定时器，其中之一就是

`ITIMER_REAL`, 它用的是内核的 HRT。然而, 这个功能在系统暂停工作时并不发挥作用。换句话说, 如果一个应用程序用 `setitimer()` 来请求在给定的时间被激活, 之后在运行期间如果设备被暂停, 那么只有在设备被再次唤醒时应用程序才会获得信号。

除了 `setitimer()` 系统调用以外, 内核的 RTC 驱动程序可以通过 `/dev/rtc` 访问, 并且允许它的用户用 `ioctl()` 函数来打开一个计时器, 计时器在系统中由 RTC 硬件设备激活。不管系统是否暂停这个计时器都将启动, 这是因为它以 RTC 设备的行为为依据, 而 RTC 设备即使系统暂停时仍然能保持活跃状态。

Android 的 alarm 驱动巧妙地结合了两者的优势。默认情况下, 驱动程序使用内核的 HRT 功能将 alarm 提供给它的用户, 类似内核内置的定时器功能。然而, 当系统将要暂停时, 它会对 RTC 进行设置, 使得系统在合适的时间被唤醒。因此, 无论何时一个用户空间的应用程序需要一个特定的 alarm, 它只要在合适的时候用 Android 的 alarm 驱动将它唤醒, 不用管系统是否是处于暂停状态中。

从用户空间的角度来看, alarm 驱动程序表现为 `/dev/alarm` 里的字符设备, 允许用户通过 `ioctl()` 函数调用来设置 alarm 且调节系统时间 (实际时间)。有少量的关键 AOSP 组件是在目录 `/dev/alarm` 里。比如说, `Toolbox` 和 `SystemClock` 类, 可以通过应用程序 API 调用, 通过它来设置 / 获取系统时间。不过最重要的是, Alarm 管理器为部分系统服务提供服务, 用它可以为应用程序提供 alarm 服务, 通过 `AlarmManager` 类来开放给应用程序开发者。

无论什么时候只要满足 alarm 和其他 Android 的 wakelock 相关行为保持一致性这一条件, 驱动程序和 alarm 管理器就都可以使用 wakelock 机制。因此, 当一个 alarm 被触发以后, 如果有必要, 则它关联的应用程序在系统被再次暂停之前可以做任何需要的操作。

在写本书的时候, Android 的 alarm 驱动程序被加入到了内核的阶段代码树中。尽管它下一步的工作还悬而未决。

## 日志系统

日志是 Linux 系统也包括嵌入式的 Linux 系统中的另一个重要的组件。分析系统的日志可以找出系统曾经发生的或者实时的错误或警告, 这对解决系统的重大错误有很大的帮助, 特别是那些只在瞬间出现的错误。默认情况下, 大多数的 Linux 发布版本包括两种日志系统: 一种是内核自带的日志, 通常通过 `dmesg` 命令来访问, 另一种是系统日志, 通常将日志文件存储在 `/var/log` 目录下。内核日志通常包含的信息是由各种

`printf()` 调用在内核中产生的，这个调用函数或者是被主内核程序或者是被设备驱动调用。相比较而言，系统日志则是包含来自于各种运行在系统里的守护进程和实用程序产生的信息。实际上，你可以使用 `logger` 命令来将你的信息写到系统日志中。

在 Android 中，内核的日志功能得到运用。然而，几乎没有任何一个通常在大多数 Linux 发布版本中使用的系统日志软件包被发现使用于 Android 中。相反，基于加入到内核中的 Android 日志系统驱动，Android 定义了自己的日志机制。经典的系统日志依赖于套接字发送消息，并因此产生一个任务切换。它也用文件来存储信息，因此产生的信息都写入到一个存储设备中。相比之下，Android 的日志功能管理的是少量独立的由内核管理的缓冲区，这些缓冲区用于记录从用户空间发来的数据。因此，记录每一个事件时不需要任务切换或者是文件写入这样的操作。相反，对于每一个传入的事件，驱动程序维护 RAM 中的记录日志的循环缓冲区，并立即返回给调用者。

Android 里所用的在设置里禁止文件的写操作有很多好处。比如说，不同于一个桌面环境或者服务器环境，一个嵌入式系统中不一定希望有一个无限增长的日志文件。而它希望的是成为这样的系统，即使所用文件系统的类型都是只读的也能够记录日志。此外，大多数的 Android 设备依赖于固态的存储设备，这些设备有数量有限的擦除周期。在这个例子里，避免多余的写入是至关重要的。

由于其量级轻、高效和友好的嵌入式系统设计，Android 的日志系统可以被用户空间组件实时使用来定期地记录事件信息。实际上，开放给应用程序开发者的 `Log` 类多多少少都直接调用了日志系统驱动程序来写入信息到主要事件缓冲区。显然，所有好的主意都会被滥用，这也使得日志系统显得很优秀，但是仍然有这样的问题，通过应用程序 API 的 `Log` 类开放的日志系统的使用程度，连同 AOSP 本身的日志系统的使用程度一起，似乎很难承担，而且 Android 的日志记录是基于系统日志的。

图 2-2 详细地描述了 Android 的日志系统框架。如你所见，日志系统的驱动程序是所有其他日志相关功能依赖的核心构造块。它的每一个缓冲区是作为目录 `/dev/log` 里一个单独的条目开放给用户。然而，没有用户空间组件可以直接与这个驱动程序进行交互，都是基于 `liblog` 的，`liblog` 提供了很多不同的日志记录功能。根据使用的功能和传递的参数、时间将会被记录到不同的缓冲区中。比如说，`liblog` 功能被 `Log` 类和 `Slog` 类使用，将会测试来自于收音机相关模块的事件是否被调度了。如果是这个事件会被写到“收音机”缓冲区内；如果不是，`Log` 类将会把这个事件写入到“主”缓冲区内，不然 `Slog` 类将会把它写入“系统”缓冲区内。“主”缓冲区可以用 `logcat` 命令（如果它的事件不带任何参数的话）来将它存储的事件显示出来。

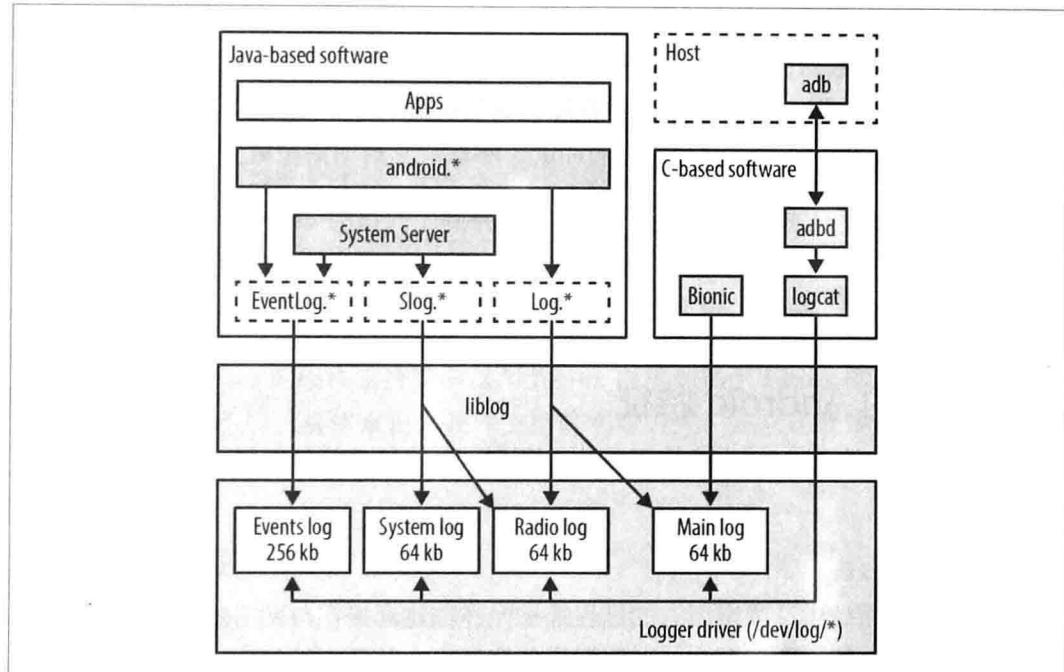


图 2-2: Android 的日志系统框架

Log 类和 EventLog 类都是通过应用程序 API 向用户开放的，而 Slog 类只在 AOSP 内部使用。不过，尽管 EventLog 提供给应用程序开发者使用，在文档中明确地指出它主要针对的是系统开发者而不是应用程序开发者。实际上，作为开发者文档的一部分，绝大多数的代码样本和用例都是用 Log 类提供的。通常，EventLog 类被系统组件用于记录二进制事件并写入 Android “事件” 缓冲区。一些系统组件，特别是系统中服务主导的服务，将结合 Log、Slog 和 EventLog 三个类来记录不同的事件。比如说有一个可能与应用程序开发者有关的事件，这个事件可能会用 Log 类来记录日志，另有一个平台开发者或者是系统集成者相关的事件，可能会用 Slog 或者是 EventLog 来记录日志。

请记住 logcat 工具，它通常被应用程序开发者用来复制 Android 日志，它也依赖于 liblog。liblog 除了提供对日志管理器驱动程序的访问功能之外，还提供格式化事件的功能，以便漂亮地打印和过滤。liblog 的另一个特点是它要求每一个被记录的事件都有一个优先级、一个标签（tag），以及数据。这个优先级是 verbose、debug、info、warn 或者是 error 中的一个。而标签（tag）是一串唯一的字符串以识别写入日志中的组件或者模块，数据则是需要被记录的实际信息。这样的表示方法实际上对于接触到应用程序开发 API 的人来说相当熟悉，这也正符合开发者文档列出的关于 Log 类的说明。

本节的最后一个谜题是 *adb* 命令。正如我们将在后面讨论的，AOSP 包括一个运行在 Android 设备上的 Android 调试桥接器（ADB）守护进程，可以用 *adb* 命令行工具从主机上进行访问。当你在主机上输入 *adb logcat*，守护进程启动目标主机本地的 *logcat* 命令，复制目标主机的“主”缓冲区，然后将它返回到主机并显示在终端上。

在写本书的时候，日志系统驱动程序已经被合并到了内核的 *drivers/staging/android/* 目录中。可以看一下 Mainline Android logger project ([http://elinux.org/Mainline\\_Android\\_logger\\_project](http://elinux.org/Mainline_Android_logger_project))，或者更多关于这个驱动的主线状态信息。

## 其他重要的 Android 特征

除了前面提到的这些 Android 特征以外，还有一些特征也值得一说，这里就做一个简要的介绍。

### *Paranoid networking*

在 Linux 系统中，通常所有的进程都被允许创建套接字与网络进行交互。但是，在 Android 的安全模型中，对于网络的访问能力是受到控制的。所以，它在内核中增加了一个选项来控制套接字的创建和网络接口的管理，它通过当前进程所属的进程组属性或该进程所拥有的能力来进行判定。适用于 IPv4、IPv6 和蓝牙等。

在写作本书的时候，这个特性还没有被整合进 Linux 的主线当中，其所在的路径也并不确定。你其实可以在一个不包含此特性的 Linux 内核上运行 AOSP，但此时 Android 的权限系统，特别是与网络有关的部分，就没法工作了。

### 内存控制台

正如我前面所述，内核管理自己的日志，可以通过 *dmesg* 命令来查看。这个日志的内容非常有用，它通常包含了内核和驱动的关键信息。当内核崩溃时，其信息可以用来做事后分析。但是这个信息通常在重启时就丢失了，所以 Android 增加了一个驱动程序并注册成为一个基于内存的控制台。该控制台在重启时不会丢失信息，并且其内容可以通过 */proc/last\_kmsg* 来访问。

在写作本书的时候，内存控制台的特性好像已经被整合到 Linux 主线当中，通过内核的 *pstore* 文件系统的 */pstore* 目录访问。

### 物理内存 (*pmem*)

和 *ashmem* 一样，*pmem* 驱动程序允许在进程间共享内存。然而，不像 *ashmem*，它可以共享大块连续的物理内存区域，而不是虚拟内存的共享。此外，这些内存区域可以在用户进程和驱动程序之间共享。例如，拿 G1 手机来说，*pmem* 堆被用于 2D 硬件加速。但是请注意，*pmem* 只在极少数的设备中使用。

事实上，根据 Android 内核开发团队的成员之一——Brian Swetland 所说，它是为了专门解决 G1 手机所用的 SoC 芯片——MSM7201A 的局限性而开发的。

在写作本书的时候，这个驱动程序已经被认为是过时的，且已被删除。它不在内核主线当中，且没有计划恢复。看来，pmem 使用的任何地方都有可能被 ION 内存分配器 (<http://lwn.net/Articles/480055/>) 所代替。

## 硬件支持

你会发现，Android 的硬件支持的做法与 Linux 内核和基于 Linux 的发行版的经典做法有着很明显的不同。具体来说，硬件支持的实现方式、建立在硬件上接口形式、代码的授权和发布理念等都不相同。

### Linux 的方法

创建提供新硬件支持的 Linux 设备驱动程序的通用方法是将驱动程序作为内核的一部分或作为运行时动态加载的模块。然后，相应的硬件一般可在用户空间通过在 `/dev` 中的条目访问。Linux 驱动程序模型定义了三种基本的设备类型：字符设备（设备呈现为字节流）、块设备（基本上是硬盘）和网络设备。多年来，已经添加了不少额外的设备和子系统类型，如 USB 设备或内存技术设备（MTD）设备。然而，对应于一个给定设备来说，其 API 和 `/dev` 项的接口方法仍然相当标准和稳定。

这使得各种软件栈可以建立在 `/dev` 节点上，它们要么直接与硬件交互，要么通过它暴露通用的 API 接口，使得用户应用程序通过它能访问其硬件。绝大多数 Linux 发行版其实提供了一组类似的核心库和子系统，如 ALSA 音频库和 X Window 系统，它们都是通过 `/dev` 暴露的接口来与硬件设备交互的。

在许可和发行方面，“Linux”的一贯方针是将驱动合并到内核主线中作为其一部分来维护，并在 GPL 条款下发布。所以，因为某些设备驱动程序开发和维护是一些个人和机构独立进行的，有的甚至是采用其他授权形式发布的，很明显，Linux 的这种方法就不是首选的方法。事实上，对于驱动程序的授权形式的问题，非 GPL 的许可一直是一个很有争议的问题。因此，传统的观点是，用户和发行商获得最新驱动程序的最好方法通常是从 <http://kernel.org> 获得最新的内核主线源代码。这种方式从内核创建初期开始到现在都是正确的，当然，现在内核也做了一些额外的补充以允许创建用户空间的驱动程序。

## Android 的通用方法

虽然 Android 是构建在内核的硬件抽象能力基础上的，但是其做法却非常不同。从纯粹技术层面来看，最明显的区别就是其各子系统和库不依赖于标准的 `/dev` 条目就能正常工作。相反，Android 的框架通常依赖于制造商所提供的共享库与硬件交互。实际上，Android 依赖一个可以被看作是硬件抽象层（HAL）的模块。正如我们将看到的，尽管不同类型的硬件模块，其抽象的接口、行为和功能存在很大的不同。

此外，大多数 Linux 发行版中与硬件交互的软件栈通常在 Android 中都没有。例如，没有 X Window 系统，而用不用 ALSA 驱动程序也不一定（这个决定留给了硬件制造商在实现 HAL 音频支持所提供的共享库时来做），访问这些的功能的方法与标准的 Linux 发行版也大不一样。例如，在 Linux 桌面环境中通常使用 ALSA 函数库来操作 ALSA 驱动接口，而官方的 AOSP 源码树却并不是这样。相反，近期的 Android 版本中包含了一个 BSD 许可 `tinyalsa` 库作为替代。

图 2-3 给出了在 Android 系统下典型的硬件抽象方式，以及与之相应的发布与许可方式。如你所见，Android 最终仍然依赖于内核来访问硬件。然而，它是通过共享的动态库来实现的，这个动态库要么由设备制造商提供，要么就是 AOSP 的一部分。一般来说，你可以把 HAL 层看作是图中硬件库装载器，不同的硬件类型定义了不同的头文件，这些头文件用于定义硬件库 `.so` 文件的 API 接口。

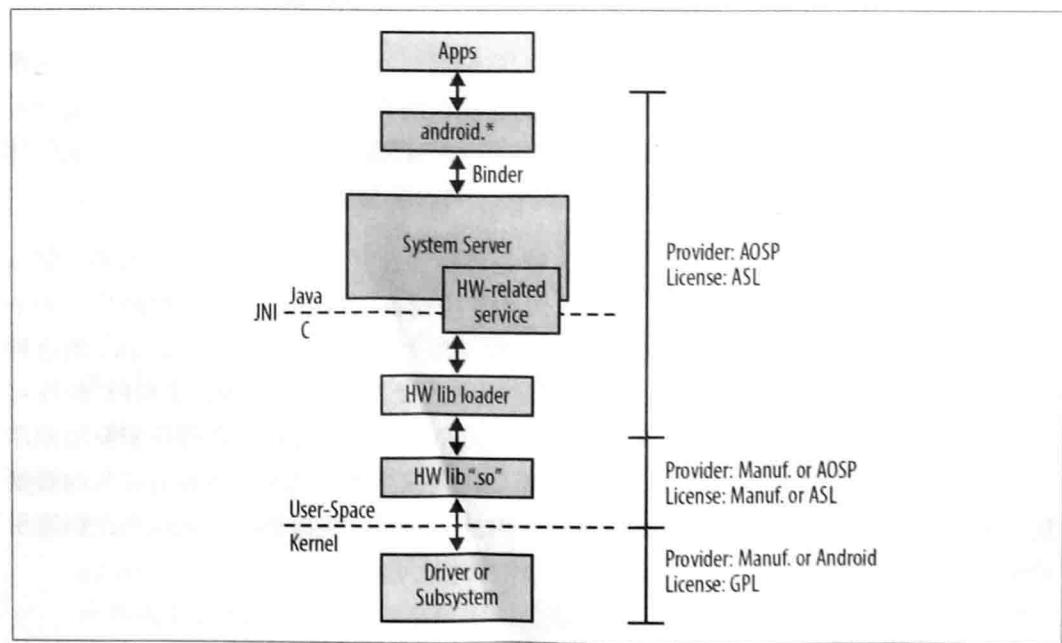


图 2-3: Android 的硬件抽象层

这种方式的主要特点在于，动态库发布所采用的许可方式可以由厂商自行决定。因此，设备制造商可以创建一个简单的设备驱动程序实现特定硬件的访问接口，该驱动程序在 GPL 下提供最基本的操作原语。这样不会暴露太多的硬件信息，因为驱动程序没有做任何实质的工作。这些硬件的驱动程序通过 `mmap()` 或 `ioctl()` 接口将硬件暴露给用户空间，大量智力劳动可以通过用户空间私有的动态库来实现，用户程序通过动态库的函数来驱动硬件。

实际上，Android 并不指定动态库、驱动程序或内核子系统如何交互，它只通过 HAL 指定了动态库应该提供的函数接口。因此，这就要由你根据你的硬件来确定最适合你的硬件具体的内核驱动程序接口，只要你提供的共享库实现了相应 API 就行。不过，我们将在下一节中介绍 Android 下典型的硬件接口。

Android 下各种共享库的装载方式是不同的。而现在只需要记住的是，对于大多数设备类型来说必须有一个 `.so` 文件才能工作，这个文件要么由 AOSP 提供，要么由你来提供。

无论加载硬件共享库的方法是什么，通常会有一个相对应的系统服务来负责加载并与共享库进行交互。这个系统服务负责与其他系统服务进行协调，使得硬件能够与系统其他部分的工作相一致，同时将 API 暴露给应用程序开发者。如果你要加入对一个特定类型硬件的支持，尽可能详细地理解与该硬件相关的系统服务的内部机制是至关重要的。通常情况下，系统服务被分割在两个部分：一部分采用 Java 实现 Android 特定的功能，另一部分采用 C/C++ 实现，它的工作主要是与 HAL、硬件的共享库，以及其他低级函数打交道。

## 装载和接口方法

正如我前面提到的，Android 以及系统服务操作硬件支持共享库的方式多种多样。很难完全理解为什么会有这样多的方法，我认为这主要是历史原因导致的。幸运的是，似乎是有一种潮流使得这些方法朝着更加统一的方式来做。鉴于 Android 的发展速度相当快，大家需要密切关注这个领域，因为它在可预见的将来可能会发生改变。

请注意，这里描述的方法并不一定相互排斥。在 Android 框架结构中，装载和使用动态库或软件模块通常组合了多种方法。在下节中，我会介绍具体硬件的做法。

### `dlopen()`, 通过 HAL 装载

适用于：GPS、灯光（light）、传感器（sensor）和显示。自 4.0/ 冰激凌三明治开始，也适用于音频和摄像头。

一些硬件支持动态库通过 *libhardware* 库装载。这个库是 Android HAL 的一个部分，它提供一个 `hw_get_module()` 函数接口，用于一些系统服务和子系统来装载特定的硬件支持动态库（这个动态库用 HAL 的术语来说，被称“模块”）。`Hw_get_module()` 本身是基于经典的 `dlopen()` 函数实现，这个函数用于将动态库装载到调用者地址空间。

---

**警告：**不要将 HAL 的“模块”与可装载内核模块相混淆，这两个是完全不同并且不相关的软件组件，虽然它们有一些相似的特点。

---

### 编译器，链接的 .so 文件

适用于：音频、摄像头、WIFI、振动器和电源管理。

在某些情况下，系统服务简单地与给定的 `.so` 文件在编译时进行链接。所以，当相应的二进制程序运行时，动态链接器会自动将相应的共享库装载到进程的地址空间。

### 硬编码的 `dlopen()`

适用于：StrageFright 和无线接口层（RIL）。

在少数情况下，代码直接调用 `dlopen()` 而不是通过 *libhardware* 来读取硬件相关动态库。为什么要这样做的原因尚不明确。

### 套接字

适用于：蓝牙、网络管理、磁盘挂载和无线接口层（RIL）。

系统服务和框架组件经常采用套接字来与守护进程或硬件接口服务进行通信。

### sysfs 条目

适用于：振动器和电源管理。

`sysfs` (`/sys`) 文件系统的一些条目可以用于控制硬件和内核子系统的行为。某些情况下，Android 使用该方法而不是 `/dev` 目录来控制硬件。使用 `sysfs` 而不是 `/dev` 是有道理的，例如，当需要在系统初始化时设置一些默认值，而此时框架还没有启动。

### `/dev` 节点

适用于：几乎所有类型的硬件。

可以说，任何硬件抽象都必须在某些时候与 `/dev` 目录中的某个条目进行通信，因为这是驱动程序与用户空间程序交互的接口。这种通信在某些时候可能被 Android 本身所隐藏了，因为它通过一个共享库来交互，但在某些情况下，

AOSP 本身的组件甚至都直接访问 `/dev` 下的设备节点。Android 输入管理器的输入函数库就是这样的一个例子。

#### D-BUS

适用于：蓝牙。

D-BUS 是一个典型的消息传递系统，在大多数的 Linux 发行版中它用于各种桌面组件之间的通信。它被包含在 Android 中是因为它被用于非 GPL 的组件跟 GPL 许可的 BlueZ 的蓝牙堆栈进行通信。Bluz 是 Linux 的默认蓝牙栈，这样做就不需要受到 GPL 的再发布的要求；D-BUS 本身是一个双许可组件——学术自由许可证（AFL）和 GPL 许可。更多信息可以参看 [freedesktop.org](http://freedesktop.org) 的 D-BUS 页面。考虑到自从 4.2/ 果冻豆开始，BlueZ 被从 AOSP 中移除，D-BUS 在将来的 Android 发布中是否会继续存在就不是很清楚了。

## 有关设备支持的细节

表 2-1 总结了 Android 系统中各种硬件支持的方法。你会发现，这些机制和接口存在多种组合方式。如果你打算实现某个特定类型的硬件支持，最好的办法是从现有样本开始来实现。在 AOSP 中通常包括针对少数手机的硬件支持代码，这些手机通常是谷歌用于开发新的 Android 版本，这些手机就成为了牵头的硬件设备。有些硬件设备的支持代码有很多，例如三星的 Nexus S（又名“Crespo”）在姜饼系统中，而 Galaxy Nexus（又名“Maguro”）和 Nexus7（又名“Grouper”）在果冻豆中。

你不可能完全靠自己就能找到公开可用的实现方法的，唯一的一种硬件类型是 RIL。由于种种原因，最好不要让大家都玩电波比较好。因此，制造商不会让这样的实现代码公开。相反，如果你希望实现一个 RIL 时，谷歌在 AOSP 中提供了一个参考的版本。

表 2-1：Android 的硬件支持方法与接口

硬件	系统服务	用户空间接口	硬件接口
音频	Audio Flinger	编译器链接 <sup>a</sup> 的 <i>libaudio.so</i>	虽然通常采用 ALSA，但这由硬件厂商确定
蓝牙	Bluetooth Service	与 Bluz <sup>b</sup> 通过套接字 或 D-Bus 连接	BlueZ 协议栈
摄像头	Camera Service	编译器链接 <sup>c</sup> 的 <i>libcamera.so</i>	由硬件厂商确定，有时用 Video4Linux
显示	Surface Flinger	HAL 装载的 <i>gralloc</i> 模块 <sup>d</sup>	由厂商决定，一般是 <i>/dev/fb0</i> 或 <i>/dev/graphics/fb0</i>

表 2-1：Android 的硬件支持方法与接口（续）

硬件	系统服务	用户空间接口	硬件接口
GPS	Location Manager	HAL 装载的 <i>gps</i> 模块	由厂商决定
输入	Input Manager	本地 <i>libui.so</i> <sup>e</sup> 库	<i>/dev/input/</i> 下的条目
灯光	Lights Service	HAL 装载的 <i>lights</i> 模块	由厂商确定
媒体	不适用，或 MediaService 中的 StageFright 框架	采用 <i>dlopen</i> 装载 <i>libstagefrighthw.so</i>	由厂商确定
网络接口 <sup>f</sup>	Network Management Service	通过套接字与 <i>netd</i> 通信	<i>ioctl</i> 接口
电源管理	Power Management Service	链接器装载的 <i>libhardware_legacy.so</i>	<i>/sys/power/</i> 下的条目，早期采 用 <i>/sys/android_power</i>
无线（电话）	Phone Service	与 <i>rild</i> 通过套接字 通信， <i>rild</i> 本身采用 <i>dlopen</i> 装载厂商 提供的 <i>.so</i> 文件	由硬件厂商决定
存储	Mount Service	与 <i>vold</i> 采用套接字 通信	系统调用或 <i>/sys</i> 下的条目
传感器	Sensor Service	HAL 装载的 <i>sensor</i> 模块	由厂商决定
振动器	Vibrator Service	链接器装载的 <i>libhardware_legacy.so</i>	由厂商决定
WiFi	WiFi Service	编译器装载的 <i>libhardware_legacy.so</i>	大多数情况下通常采用 <i>wpa_supplicant</i> <sup>g</sup>

- a. HAL 装载方式从 4.0/ 冰激凌三明治开始。
- b. 从 4.2 / 果冻豆版本开始，BlueZ 已经被移除了，高通公司提供的蓝牙协议栈 Bluedroid 代替了它。Bluedroid 这个新的协议栈跟大多数其他硬件类型一样是基于 HAL 装载。
- c. 这种 HAL 装载的方式是从 4.0 / 冰激凌三明治开始的。
- d. 从 4.0 / 冰激凌三明治开始，Surface Flinger 使用的模块就是 *hwcomposer*。
- e. 从 4.0 / 冰激凌三明治开始，它就被 *libinput.so* 代替了。
- f. 这是针对 Tether、NAT、PPP、PAN、USB RNDIS（Windows）的，不适用 WiFi。
- g. 桌面的 Linux 版本也使用同一个软件包 *wpa\_supplicant* 来管理 WiFi 网络与连接。

# 原生用户空间程序

现在，我们已经了解了 Android 系统依赖的系统硬件层，让我们开始往上层看。首先，我们将介绍 Android 原生用户空间的环境。原生用户空间，我指的是所有的用户空间中运行的 Dalvik 虚拟机以外的程序。这包括很多针对目标 CPU 编译的二进制代码。这些通常可以自启动或根据需要由 init 进程根据它的配置文件来启动，或者是开发人员通过 Shell 命令行调用。这种代码通常能直接访问根文件系统和本地系统中包含的库。它们的权限严格地受到赋予它们的文件系统权限和它们的有效 UID 和 GID 限制。它们不受任何典型的 Android 应用程序权限框架的限制，因为它们运行在它之外。

请注意，Android 系统的用户空间程序几乎是从空白开始设计而来的，因此它与一般的 Linux 发行版有很大的不同。因此，我会尽量详尽地来解释 Android 用户空间程序与普通基于 Linux 的系统之间的异同点。

## 文件系统布局

与其他基于 Linux 的发行版一样，Android 使用了一个根文件系统来存储应用程序、库文件和数据。然而，与主流基于 Linux 的发行版不同的是，Android 的根文件系统布局并不遵循文件系统层次化标准（Filesystem Hierarchy Standard, FHS）<sup>注4</sup>。内核本身并不强制使用 FHS，但是为 Linux 构建的大多数软件包都假设其运行所在的根文件系统符合 FHS。所以，如果你想移植一个标准的 Linux 应用程序到 Android 上来，你很可能需要做一些工作以确保它依赖的文件路径仍然有效，或者使用某种形式的“chroot”隔离它和根文件系统的其余部分（见 chroot 的手册页）。

考虑到大多数 Android 的用户空间程序都是从头开发的，不遵守标准对 Android 来说几乎没有影响。我们后面将会看到，它实际上还带来一些好处。当然，学会浏览 Android 根文件系统就变得比较重要。如果不出意外，你在你的硬件上把 Android 运行起来之前可能需要花费大量的时间在它上面。

Android 操作的两个主要目录是 */system* 和 */data*。这些目录并不来自 FHS。事实上，我想不出任何主流的 Linux 发行版使用这些目录。相反，它们反映了 Android 开发团队自己的设计。这也暗示了 Android 系统其实可以与一个普通的 Linux 发行版同时存在于一个根文件系统之上。可以看看附录 A 来了解创建这样一个混合系统的信息。

*/system* 是 Android 的主要目录，它存储了 AOSP 构建而成的持久化组件，包括原生的二进制文件、原生的库文件、框架的文件包以及大量的 app。它通常以只读形式从一

---

注 4：FHS 是一个描述 Linux 根文件系统中各目录内容的社区标准。

个独立的镜像中挂载到根文件系统上，而根文件系统本身又是从一个 RAM 磁盘镜像中挂载而来。*/data* 目录则是 Android 系统用于存储数据以及经常会变化的 app 的目录。它包括用户安装的应用程序所产生的临时数据，以及 Android 系统组件在运行时产生的数据。它通常也是从一个独立的镜像挂载到文件系统中的，不过是以可读 / 写的方式挂载的。

Android 也包括一些其他系统中常见的目录，例如 */dev*、*/proc*、*/sys*、*/sbin*、*/root*、*/mnt* 和 */etc*。这些目录的功能通常与其他 Linux 系统相同，虽然它们通常都经过了裁减，某些目录在一些情况下甚至是空的，例如 */root* 目录。

有意思的是，Android 系统没有 */bin* 目录和 */lib* 目录。这些目录对于 Linux 系统来说通常是很关键的，分别包括重要的二进制程序和库文件。这也为 Android 系统与标准 Linux 组件共存打开了一扇门。

Android 根文件系统当然远不止于此，前面提到的这些目录本身还存在自己的层次结构。同时，Android 根文件系统还包括这里没有提到的其他目录。我们将在第 6 章中再来揭开 Android 根文件系统的面纱并重新审视它。

## 库文件

Android 依赖于 100 多个动态库文件，都存储在 */system/lib* 目录下。在这些库中，很多都来自于外部项目，它们因为 Android 需要相关的特性而被整合进 Android 系统中。当然，大部分 */system/lib* 下的库文件实际上还是 AOSP 自己产生的。表 2-2 列出了 AOSP 中来源于外部项目的库文件，而表 2-3 列出了 AOSP 自己产生的 Android 特有的库文件。

表 2-2：AOSP 中来源于外部项目的库文件

库文件	外部项目	原位置	许可证
<i>audio.so</i> , <i>liba2dp</i> , <i>input.so</i> , <i>libbluetooth</i> 和 <i>libbluetoothd</i>	BlueZ <sup>a</sup>	<a href="http://www.bluez.org">http://www.bluez.org</a>	GPL
<i>libcrypto.so</i> 和 <i>libssl.so</i>	OpenSSL	<a href="http://www.openssl.org">http://www.openssl.org</a>	自定义 BSD 类型
<i>libdbus.so</i>	D-Bus	<a href="http://dbus.freedesktop.org">http://dbus.freedesktop.org</a>	AFL 和 GPL
<i>libexif.so</i> <sup>b</sup>	Exif JPEG header manipulation tool	<a href="http://www.sentex.net/~mwandel/jhead/">http://www.sentex.net/~mwandel/jhead/</a>	Public Domain
<i>libexpat.so</i>	Expat XML Parser	<a href="http://expat.sourceforge.net">http://expat.sourceforge.net</a>	MIT
<i>libFFTEm.so</i>	neven 面部识别库	N/A	ASL

表 2-2：AOSP 中来源于外部项目的库文件（续）

库文件	外部项目	原位置	许可证
<i>libicui18n.so</i> 和 <i>libicuuc.so</i>	International Components for Unicode	<a href="http://icu-project.org">http://icu-project.org</a>	MIT
<i>libiprouteutil.so</i> 和 <i>libnetlink.so</i>	iproute2 TCP/IP 网络和流量控制	<a href="http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2">http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2</a>	GPL
<i>libjpeg.so</i>	libjpeg	<a href="http://www.ijg.org">http://www.ijg.org</a>	自定义 BSD 类型
<i>libnfc_ndef.so</i>	NXP 的 NFC 库	N/A	ASL
<i>libskia.so</i> 和 Android 2.3/ 姜饼中的 <i>libskiagl.so</i>	skia 2D 图形库	<a href="http://code.google.com/p/skia/">http://code.google.com/p/skia/</a>	ASL
<i>libsonivox</i>	Sonic Network 的音频合成库	N/A	ASL
<i>libsqllite.so</i>	SQLite 数据库	<a href="http://www.sqlite.org">http://www.sqlite.org</a>	公共域
<i>libSR_AudioIn.so</i> 和 Android 2.3/ 姜饼中的 <i>libsrec_jni.so</i>	Nuance Communications 的语音识别引擎	N/A	ASL
<i>libstlport.so</i>	C++ 标准模板库实现	<a href="http://stlport.sourceforge.net">http://stlport.sourceforge.net</a>	自定义 BSD 类型
<i>libttspico.so</i>	SVOX 的 Text-to-Speech 语音识别引擎	N/A	ASL
<i>libvorbisidec.so</i>	Tremolo ARM-optimized Ogg Vorbis 解压库	<a href="http://wss.co.uk/pinknoise/tremolo/">http://wss.co.uk/pinknoise/tremolo/</a>	自定义 BSD 类型
<i>libwebcore.so</i>	WebKit 开源项目	<a href="http://www.webkit.org">http://www.webkit.org</a>	LGPL and BSD
<i>libwpa_client.so</i>	早期 HAL 用来与 wpa_supplicant 守护进程对话的库	<a href="http://hostap.epitest.fi/wpa_supplicant/">http://hostap.epitest.fi/wpa_supplicant/</a>	GPL and BSD
<i>libz.so</i>	zlib 压缩库	<a href="http://zlib.net">http://zlib.net</a>	自定义 BSD 类型

- a. BlueZ 已经被 ASL 许可证所取代，Broadcom 公司提供的蓝牙堆栈叫做 bluedroid，它也可以在 *external/* 中找到。由 bluedroid 生成的库不同于在表中列出的那些。
- b. 请记住，Android 的 *libexif.so* 的 API 非常不同于可以在传统的 Linux 发行版上运行的库的 API。

表 2-3：AOSP 自己产生的 Android 特有的库文件

类别	库文件	说明
核心 <sup>b</sup>	<i>libc.so</i>	C 标准库
	<i>libm.so</i>	数学标准库
	<i>libdl.so</i>	动态链接库
	<i>libstdc++.so</i>	C++ 支持标准库 <sup>a</sup>
	<i>libthread_db.so</i>	线程调试库
	<i>libbinder.so</i>	Binder 标准库
	<i>libutils.so, libcutils.so,</i> <i>libnetutils.so 和 libsysutils.so</i>	各种实用工具库
	<i>libsystem_server.so, libandroid_servers.so, libaudioflinger.so,</i> <i>libsurfaceflinger.so, libsensorservice.so</i> 和 <i>libcameraservice.so</i>	系统服务相关的库
	<i>libcamera_client.so</i> 和 Android2.3/姜饼中的 <i>libsurfaceflinger_client.so</i> <sup>c</sup>	对于特定系统服务的客户端库
	<i>libpixelflinger.so</i>	PixelFlinger 库
	<i>libui.so</i>	低层次的用户接口相关的功能，比如说用户输入事件的处理和分配，以及图形缓冲区的分配和处理
	<i>libgui.so</i>	与各传感器相关的函数库，从Android4.2/冰淇淋三明治开始，与Surface Flinger 进行的客户端通信
	<i>liblog.so</i>	日志库
	<i>libandroid_runtime.so</i>	Android 实时库
	<i>libandroid.so</i>	生命周期管理，输入事件，窗口管理，资产，存储管理的 C 接口
Dalvik	<i>libdvm.so</i>	Dalvik 虚拟机库
	<i>libnativehelper.so</i>	JNI 相关的帮助功能
硬件 (Hardware)	<i>Libhardware.so</i>	HAL 库，它提供 <code>hw_get_module()</code> 以及根据需求用 <code>dlopen()</code> 来装载硬件支持模块（即为 HAL 提供硬件支持的共享库）
	<i>libhardware_legacy.so</i>	为 WiFi、电源管理和振动器提供硬件支持的早期 HAL 库

表 2-3: AOSP 自己产生的 Android 特有的库文件 (续)

类别	库文件	说明
	各种硬件支持共享库	为各种硬件组件提供支持的库；有一些是通过 HAL 装载，有一些是自动被链接器装载
媒体 (Media)	<i>libmediaplayerservice.so</i> <i>libmedia.so</i>	媒体播放器服务库 媒体播放器服务所用的低级别的媒体功能
	<i>libstagefright*.so</i>	组成 StageFright 媒体框架的很多库
	<i>libeffects.so</i> 和目录 <i>soundfx/</i> 里的库	音效库
	<i>libdrm1.so</i> 和 <i>libdrm1_jni.so</i>	DRM (数字版权管理) 框架库
OpenGL	<i>libEGL.so</i> , <i>libETC1.so</i> , <i>libGLESv1_CM.so</i> , <i>libGLESv2.so</i> 和 <i>egl/libGLES_android.so</i>	Android 的 OpenGL 实现

- a. 有人说这个库类似标准 Linux 系统中的 *libsipc++.a* 的作用，而 Android 的 *libstlport.so* 类似传统 Linux 系统中的 *libsdc++.so*。
- b. 我用这个类别名来作为很多核心的 Android 功能的论点。
- c. 从版本 4.0/ 冰淇淋三明治开始，*libsurfaceflinger\_client.so* 库相应的功能已经合并到了 *libgui.so* 库中。

从 Android2.3/ 姜饼这个版本后，另外一些库文件都被加入到了 AOSP 中。表 2-4 和表 2-5 列出了你会在版本 4.1/ 果冻豆中看到的一些最明显增加的库文件。

表 2-4: 版本 4.1/ 果冻豆中来源于外部项目的重要的库文件

库文件	外部项目	原位置	许可证
<i>libtinyalsa.so</i>	tinyalsa	<a href="http://github.com/tinyalsa">http://github.com/tinyalsa</a>	ASL
<i>libmtp.so</i>	libmtp	<a href="http://libmtp.sourceforge.net/">http://libmtp.sourceforge.net/</a>	LGPL
<i>libchromium_net.so</i>	WebKit	<a href="http://webkit.org/">http://webkit.org/</a>	LGPL and BSD
<i>libmdnssd.so</i>	mDNSResponder	<a href="http://www.opensource.apple.com/tarballs/mDNSResponder/">http://www.opensource.apple.com/tarballs/mDNSResponder/</a>	ASL

表 2-5: 版本 4.1/ 果冻豆中重要的 Android 特有的库

类别	库文件	描述
核心	<i>libjnigraphics.so</i>	2D 图形系统的 C 接口
	<i>libcorkscrew.so</i>	调试库
	<i>libRS.so</i>	RenderScript 的接口

表 2-5：版本 4.1/ 果冻豆中重要的 Android 特有的库（续）

类别	库文件	描述
媒体	<i>libOpenMAXAL.so</i>	本地多媒体库，基于 Khronos OpenMAX AL
	<i>libOpenSLES.so</i>	兼容 Khronos OpenSL EL 的音频系统
	<i>libaudioutils.so</i>	回音消除和其他音频工具

## init

有一个 Android 没有改变过的东西，就是内核的启动程序。因此，你所知道的关于 Linux 的内核启动过程，同样应用在 Android 上。Android 中有所改变的是内核完成启动之后要做的事。事实上，在完成初始化本身和它所包含的驱动程序之后，内核只启动了一个用户空间进程，就是初始化进程。之后这个进程负责开启系统中其他所有进程和服务，并进行关键的操作，比如说重启。传统 Linux 发布版本都是依赖于 SystemV 的初始化程序来作为 init 进程的，虽然近年来很多发行版本已经创建了它们自己的变种版本。比如说 Ubuntu 用的是 Upstart，在嵌入式系统中，传统的初始化包是 BusyBox。

Android 推出了自己的定制初始化程序，这给 Android 带来了一些新奇。

## 配置语言

传统的初始化程序依据对当前运行级别的配置运行既定的脚本。与它不同的是，Android 的初始化程序定义了它自己的配置语义，且依赖于属性的变化来触发特定指令的执行。

初始化程序的主要配置文件通常存储在 */init.rc* 中，但是也有特定设备的配置文件存储为 */init.<device\_name>.rc*，其中 *<device\_name>* 是这个设备的名字。在一些例子中，比如说仿真器，也会有特定设备的脚本存储为 */system/etc/init.<device\_name>.sh*。你可以通过修改这些文件高度控制系统的启动和它的行为。比如说，你可以禁止 Zygote（一个关键的系统组件，我们将会在本章和第 7 章详细地介绍它）自动启动，等到在用 *adb* 将它注入设备之后再手动启动它。

我们将在第 6 章更深入地讨论初始化程序的配置语言。

## 通用属性

Android 的初始化程序中一个非常有趣的地方是它是如何用合适的权限来管理一个通用属性集的，这个属性集可以被访问且它来自于系统的很多部分。这些属性中有一些

是在编译时被设置，有一些是在初始化配置文件中被设置，还有一些是在运行时被设置。有些属性还被保存到了存储空间中可以永久使用。由于初始化程序管理着这些属性，因此它可以检测到任何的变化从而触发基于它的配置的一组命令的执行。

比如说之前提到了 OOM 的调整，它就是被 *init.rc* 文件设置启动的。网络属性也一样。有一些属性集是在编译时被设置，它们存储在 */system/build.prop* 文件中，包含编译日期和编译系统详细信息。在运行时，系统将有超过 100 个不同的属性，范围从 IP 和 GSM 配置参数到电池的电量。用 *getprop* 命令可以获得当前的属性列表和它们的值。

我们将在第 6 章中更详细地讨论初始化程序的通用属性，告诉你属性默认值的文件和相关的命令。

## udev 事件

就像我之前解释的那样，Linux 中对于设备的访问是在通过 */dev* 目录中的节点进行的。在之前的 Linux 发布版本中，这个目录会有成千上万的条目，以满足所有可能的设备配置。不过最终的解决方案是这些节点可以被动态创建。现在使用的系统已经由 udev 管理，它根据内核的实时事件做出响应，每当硬件增加或者是被从系统中移除时这些事件被触发。

在大多数的 Linux 发布版本中，*udev* 热插拔事件的处理是通过 *udevd* 守护进程来完成的。在 Android 中，这些事件的处理是通过将 *ueventd* 守护进程编译为 Android 初始化程序的一部分来完成的，它通过 */sbin/ueventd* 向 */init* 中的符号链接来访问。*ueventd* 在 */dev/* 目录里创建的条目依赖于 */ueventd.rc* 和 */ueventd.<device\_name>.rc* 文件。

我们将在第 6 章详细介绍 *ueventd* 和它的配置文件。

## 工具箱

就像根文件系统的目录层次结构一样，大多数 Linux 系统中有一些重要的二进制文件，在 */bin* 和 */sbin* 目录中由 FHS 列出。在大多数 Linux 发布版本中，这些目录中的二进制文件通过单独的数据包构建，这些数据包来源于网络中的不同可用项目。在一个嵌入式系统中，没有必要处理这么多包，也没有必要有这么多单独的二进制文件。

经典的 BusyBox 包所用的方法是构建一个单一的二进制文件，基本上相当于一个巨大的 *switch-case*，它检查命令行的第一个参数并执行相应功能。之后所有的命令都成为了 *busybox* 命令的符号链接。但是由于 BusyBox 的行为是依据于命令行的第一个

参数的，当这个参数是 *ls* 时，它表现的行为就好像你是在一个标准的 Linux shell 中运行这个命令一样。

Android 虽然不用 BusyBox，但是它有自己的工具 Toolbox，其基本的功能实现差不多，也是用字符与 Toolbox 链接的。不过，无论在任何地方 Toolbox 都不比 BusyBox 功能丰富。实际上，如果你曾经用过 BusyBox，可能你在用 Toolbox 的时候将会非常沮丧。在这种情况下，从头开始创建一个工具的基本原理看起来似乎可以从许可证的角度出发，BusyBox 是 GPL 许可的。此外，一些 Android 开发者表示他们的目的是要创建一个基于 shell 调试器的最简单的工具，而不是要一个全面的 shell 工具，比如 BusyBox 那样的。无论如何，Toolbox 是 BSD 许可的，因此制造商可以自行修改并发布它，不需要跟踪他们的开发者所做的修改或者为客户提供任何源代码。

你可能仍然想将 BusyBox 包含在 Toolbox 里，以便可以从它的功能中受益。如果因为许可证的关系你不想将它作为你最终产品的一部分，你可以在开发过程中临时包含它，并在最终产品版本中去掉它。我会在附录 A 中对此做更详细的介绍。

## 守护进程

作为系统启动程序的一部分，Android 的初始化程序启动了几个关键的守护进程，并在系统的整个生命周期里始终保持运行。一些守护进程（比如说 *adb*）是按需启动的，它的启动与否取决于编译选项和全局属性的变化。表 2-6 列出了一些比较突出的 Android 运行的守护进程。这些进程大部分会在第 6 章和第 7 章里进行更详细的讨论。

表 2-6：Android 的守护进程

守护进程	描述
<i>ueventd</i>	Android 里 <i>udev</i> 的替代进程
<i>servicemanager</i> (服务管理)	Binder Context 管理器。是所有运行在系统中的 Binder 服务的一个索引
<i>vold</i>	卷管理器。安装和格式化已安装的卷和图片
<i>netd</i>	网络管理器。处理 tethering、NAT、PPP、PAN 和 USB RNDIS
<i>debuggerd</i> (调试工具)	调试工具守护进程。当一个进程崩溃后做事后分析时 Bionic 的链接器调用。允许主机上的 <i>gdb</i> 链接
<i>rild</i>	RIL 守护进程。调节所有电话服务和基带处理器之间的通信
<i>Zygote</i>	<i>Zygote</i> 进程。它负责预热系统的缓冲区并启动系统服务。我们将在本章的后面详细讨论这个部分
<i>mediaserver</i> (媒体服务)	媒体服务器。主管大多数媒体相关的服务。我们将在本章的后面详细讨论这个部分

表 2-6：Android 的守护进程（续）

守护进程	描述
<i>dbus-daemon</i>	D-Bus 消息守护进程，是 D-Bus 用户之间的调解者。想要了解更多信息可以看一下它的主页
<i>bluetoothd</i> （蓝牙）	蓝牙守护进程。管理蓝牙设备。通过 D-Bus 提供服务。从版本 4.2/ 果冻豆开始不再包含在 AOSP 里，因为 BlueZ 堆栈被移除了
<i>installd</i>	.apk 安装守护进程。主要关注安装和卸载 .apk 文件以及管理相关的文件系统的目录
<i>keystore</i> （密钥库）	密钥库（KeyStore）守护进程。管理一对存储加密密钥的加密的 key-value 对，比如说 SSL 证书
<i>system_server</i>	Android 的系统服务。这个守护进程管理大多数运行在 Android 里的系统服务
<i>adb</i>	ADB 守护进程。管理目标主机和主机的 adb 命令间所有的通信

## 命令行工具

Android 的根文件系统有超过 150 个命令行工具。*/system/bin* 目录里包含其中的大多数，但是有一些“例外”包含在 */system/xbin* 目录中，还有极少数包含在 */sbin* 目录中。*/system/bin* 目录中大约有 50 个工具实际上是符号链接到 */system/bin/toolbox* 的。其余的大部分来自于 Android 的基本框架，或来自于合并到 AOSP 中的外部项目，或者是来自于 AOSP 中其余的各种部分。有机会我们将在第 6 章和第 7 章中更详细地介绍各种 AOSP 中的二进制文件。

## Dalvik 以及 Android 上的 Java

简单地说，Dalvik 是 Android 的 Java 虚拟机。它允许 Android 在上面运行基于 Java 的应用程序生成的字节码和 Android 自己的系统组件，并提供所需的钩子函数和环境用于与系统的其他部分相连接，包含了本地库和剩余的本地用户空间。虽然对于 Dalvik 以及 Android 上的 Java 品牌有很多要说的东西，但是在我深入介绍这部分之前，必须先讲述一些 Java 的基础知识。

我不想惹人烦地详细介绍 Java 语言的历史教训和来源，在这里我只想说 Java 是由 James Gosling 在 20 世纪 90 年代初期在 Sun 公司时创建的，它快速地流行起来，总的说来，在 Android 到来前它已经极具规模。从一个开发者的角度来看，请记住对

于 Java 有两个方面非常重要：它不同于传统的语言比如说 C 语言和 C++，以及我们一般称为“Java”的组成它的组件。

根据设计，Java 是一种解释型语言。不同于 C 和 C++，用这些语言写的源代码被一个编译器编译成为二进制指令集合，而且要在与编译器指定的体系结构相匹配的 CPU 上执行，但是用 Java 写的源代码是被一个 Java 编译器编译成架构独立的字节码，这些字节码是由字节码解释器实时执行的，这种字节码解释通常也可以称为“虚拟机”。这种操作方法，结合 Java 的语义，使得 Java 语言包含相当多的特点，这些特点在之前传统的语言中是找不到的，比如说反射和匿名的类。此外，与 C 和 C++ 不同，Java 并不要求你跟踪你所分配内存的对象。实际上，它不要求你知道所有未使用的对象的踪迹，这是因为它有一个综合垃圾收集器，以保证所有没有活跃的代码和被销毁的类都不再被引用。

在实践层面上，Java 实际上是由几种特别的东西组成的：Java 编译器，Java 字节码解释器（通常被称为 Java 虚拟机（JVM）），以及常常被 Java 开发者使用的 Java 库。总之，开发者通常可以从 Java 开发包（JDK）中获得这些，JDK 是由 Oracle 免费提供的。实际上 Android 在编译期间也依赖于 JDK 的 Java 编译器，但是它不用 JVM 或者 JDK 提供的库。Android 依赖于 Dalvik 虚拟机而不是 JVM，它依赖于 Apache Harmony 项目而不是 JDK 库文件，Apache Harmony 项目是 Java 库的一个洁净版本（clean-room），托管于 Apache 项目的保护伞下。

---

**注意：**AOSP 编译产生的镜像中没有任何的 JDK 组件。因此，在你的嵌入式系统中使用 Android 时，你不可以分布任何的 JDK 组件。

---

## Java 术语

Java 有自己的专业术语。如果你对这些术语不熟悉，下面的解释将会帮助你了解一些在书中的常用术语：

### 虚拟机

这个术语在 Java 出现时少有歧义，这是因为“虚拟机”软件产品比如说 VMware 和 VirtualBox 并不像现在这样常见。这些虚拟机所做的远远超过了像 Java 虚拟机那样仅仅解释字节码。

### 反射

这个功能的目的是询问一个对象是否实现一个特定的方法。

## 匿名 (anonymous) 类

将代码的片段像一个参数一样传递给被调用的方法。一个匿名类被调用，比如说作为一个注册回调方法，使开发者可以看到在同一个位置处理一个事件的代码，这个位置就是他调用注册回调方法的位置的源代码所在的地方。

## .jar 文件

.jar 文件实际上是包含很多 *.class* 文件的 Java ARchives (JAR)，每一个 *.class* 文件只包含一个单独的类。

根据其开发者 Dan Bornstein 的意思，Dalvik 将它自己和 JVM 区分开来，因为它是专门为嵌入式系统设计的。也就是说，它的目标系统有速度慢的 CPU 和相对小的 RAM，运行的操作系统不交换空间，使用电池供电。

JVM 处理的是 *.class* 文件，而 Dalvik 处理的是 *.dex* 生成文件。*.dex* 文件实际上是 Java 编译器生成的 *.class* 文件通过 Android 的 *dx* 工具生成的。排除其他因素，一个未压缩的 *.dex* 文件只有它原始 *.jar* 文件的 50% 大小。

想要知道更多关于 Dalvik 的特征和内部结构的信息，我强烈推荐你看一下 Dan Bornstein 在 Google I/O 2008 上演讲的题为“Dalvik 虚拟机内部结构”的报告。这个报告时长大概 1 小时，可以在 YouTube 上看到。你也可以到 YouTube 的主页中搜索“Dan Bornstein Dalvik”来找到这个视频。

---

**注意：**另一个有趣的传言是，Dalvik 是基于寄存器的，然而 JVM 是基于堆栈的，显然这可能对你来说一点意义都没有，除非你是对有关 VM 理论、体系结构和内部结构有强烈兴趣的学生。

如果你想要深入了解基于堆栈的 VM 和基于寄存器的 VM 之间各自的好处和取舍，你可以看一下名为“Virtual Machine Showdown: Stack Versus Registers”的文章，这篇文章由 Shi 等人撰写，刊登在 2005 年 6 月 11 日 ~ 12 日在芝加哥出版的 VEE'05 的第 153 ~ 163 页。

---

不过，Dalvik 有一个特点非常值得一提，就是从版本 2.2/ 冻酸奶开始，它包含了一个针对 Android 的 Just-in-Time (JIT) 编译器，之前 x86 和 MIPS 版本的已经增加了。在历史上，JIT 一直是很多虚拟机的一个定义特征，帮助它们缩小与非解释型语言的差距。事实上，有一个 JIT 意味着 Dalvik 将应用程序的字节码转换为二进制指令集合在目标主机的 CPU 上本地运行，而不是要用虚拟机逐个解释指令。之后这种转换的结果被存储起来以供将来使用。因此，应用程序在第一次启动时需要比较长的时间，但是一旦它们的系统被 JIT 化了，再次装载和运行就比之前快很多了。这里唯一需要注意的是，JIT 只能在几个体系结构中有效，分别是 ARM、x86 和 MIPS。

作为一名嵌入式开发者，你不太需要做什么具体的工作以使得 Dalvik 虚拟机能够运行在你的系统中。Dalvik 虚拟机是独立于体系结构的。据报道，一些早期的 Dalvik 移植会遭受了一些字节存储顺序的问题。然而，之后这些问题似乎都没有了。

## Java 本地接口（JNI）

先不管它的能力和优势，Java 不能一直只工作在一个不与外界相连接的环境中，用 Java 写的代码有时候需要与其他语言写的代码相连接。在一个嵌入式系统比如说 Android 系统中尤其如此，系统中低级别的功能相伴左右。为此提供了 Java 本地接口（JNI）机制。它本质上是一个与其他语言连接的桥梁，比如说 C 和 C++。它相当于 .NET/C# 里的 *P/Invoke*。

应用程序开发者在他们用 SDK 编译的常规 Java 代码中，有时候会用 JNI 来调用他们用 NDK 编译的本地代码。不过在内部，AOSP 主要依赖于 JNI 来使得 Java 编码的服务和与 Android 低层级功能连接的组件有效，这些低层级的功能大多数是用 C 和 C++ 写的。比如说 Java 写的系统服务经常用 JNI 来与匹配的本地代码进行通信，这部分代码与一个特定服务的相关硬件相连接。

允许 Java 通过 JNI 与其他语言进行通信的很大一部分繁重的工作是由 Dalvik 虚拟机来完成的。比如说如果你返回去看前一节的表 2-3，你会注意到 *libnativehelper.so* 库文件，它就是 Dalvik 提供来协助 JNI 调用的一部分。

附录 B 给出了一个使用 JNI 来连接 Java 和 C 代码的例子。就目前而言，请牢记 JNI 是 Android 平台工作的核心，它使用起来是一个相对复杂的机制，尤其要注意的是你要保证使用的是正确的调用语义和功能函数。

---

**注意：**不幸的是，学习开始使用 JNI 似乎是一件艰难的事情。换句话说，很难找到有关它的好的文档说明。这里有一本有关这个主题的权威的书，由 Sheng Liang (Addison-Wesley 出版社，1999 年出版) 写的《The Java Native Interface Programmer's Guide and Specification》（Java 本地接口程序员的指南和规范说明）。

---

## 系统服务

系统服务是 Android 中的幕后工作者。即使它没有明确的在 Google 的应用程序开发者文档中提到，那些 Android 中有趣的东西都要经过大约 50 到 70 个系统服务中的一个。这些系统服务一起合作共同为本质上相当于一个在 Linux 上编译的面向对象的操作系统提供服务，这实际上也是 Binder（所有系统服务构建的机制）的目的所在。我们刚

刚提到的本地用户空间实际上是为支持 Android 系统服务而设计的支撑环境。因此，理解有什么系统服务，它们之间如何交互，以及它们与系统的其他部分如何交互是至关重要的。我们在讨论 Android 的硬件支持时已经介绍过这方面的一些信息。

图 2-4 更详细地说明了图 2-1 中介绍的系统服务。如你所见，实际上其中涉及的主要是一组主进程。其中最突出的是系统服务，它的所有组件都运行在同一个进程即 *system\_server* 中，这个进程主要由 Java 编码的服务以及两个由 C/C++ 写的服务组成。系统服务也包含一些通过 JNI 访问的本地代码，以允许基于 Java 的服务与 Android 的底层相连接。另一组系统服务被封装成媒体服务（Media Service），运行为 mediaserver。这些服务都是用 C/C++ 编码的，都被与 Media 相关的组件一起打包，比如 StageFright 多媒体框架和音效效果。最后，手机应用程序中涉及的电话服务被单独分开来。自从版本 4.0/ 冰淇淋三明治以后，可以注意到 Surface Flinger 已经被分为一个单独的独立进程。

---

**警告：**这些术语不是我命名的，因此很不幸，它们比较混乱。“系统服务（System Server）”程序中设有多个系统服务在一个相同的进程里。“媒体服务（Media Service）”也一样。“系统服务”和“媒体服务”都是一个单一的描述，不管他里面包含多少个系统服务。当本书里用的是复数的“系统服务（system services）”时，它指的是系统里所有可用的的系统服务，不管它们所运行的进程是什么。所以，简单地说，“系统服务（System Server）”和“媒体服务（“Media Service”）”都不是“系统服务”的一部分。相反，它们是用来运行后者的进程。

---

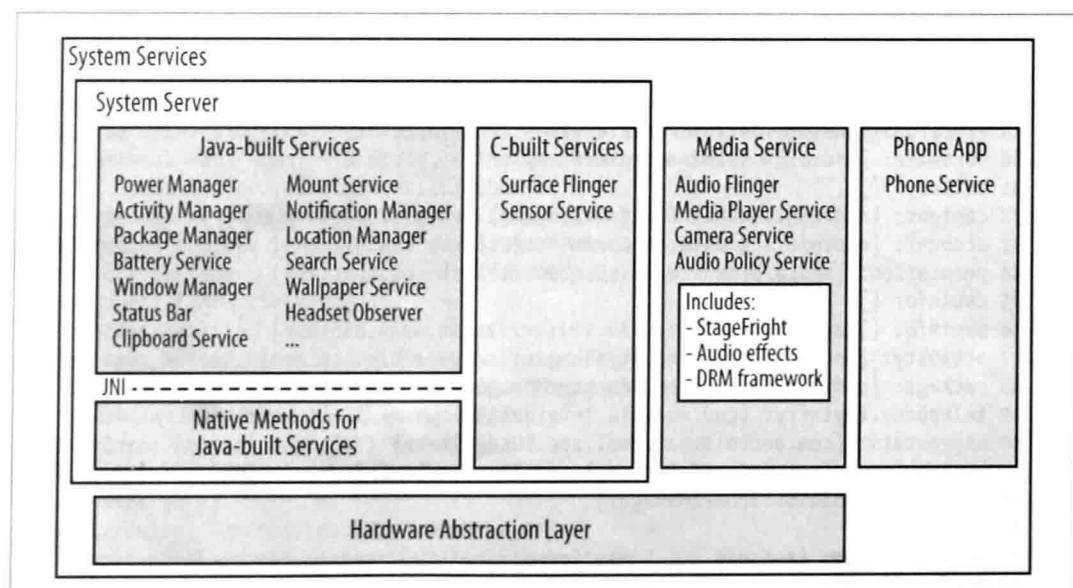


图 2-4：系统服务

请注意，尽管只有少数的几个进程来覆盖所有的 Android 系统服务，它们对于通过 Binder 与它们的服务连接的任何程序来说都是独立运作的。以下就是 Android2.3/ 姜饼仿真器上 service 实用工具的输出：

```
# service list
Found 50 services:
0 phone: [com.android.internal.telephony.ITelephony]
1 iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
2 simphonebook: [com.android.internal.telephony.IIccPhoneBook]
3 isms: [com.android.internal.telephony.ISms]
4 diskstats: []
5 appwidget: [com.android.internal.appwidget.IAppWidgetService]
6 backup: [android.app.backup.IBackupManager]
7 uimode: [android.app.IUiModeManager]
8 usb: [android.hardware.usb.IUsbManager]
9 audio: [android.media.IAudioService]
10 wallpaper: [android.app.IWallpaperManager]
11 dropbox: [com.android.internal.os.IDropBoxManagerService]
12 search: [android.app.ISearchManager]
13 location: [android.location.ILocationManager]
14 devicesstoragemonitor: []
15 notification: [android.app.INotificationManager]
16 mount: [IMountService]
17 accessibility: [android.view.accessibility.IAccessibilityManager]
18 throttle: [android.net.IThrottleManager]
19 connectivity: [android.net.IConnectivityManager]
20 wifi: [android.net.wifi.IWifiManager]
21 network_management: [android.os.INetworkManagementService]
22 netstat: [android.os.INetStatService]
23 input_method: [com.android.internal.view.IInputMethodManager]
24 clipboard: [android.text.IClipboard]
25 statusbar: [com.android.internal.statusbar.IStatusBarService]
26 device_policy: [android.app.admin.IDevicePolicyManager]
27 window: [android.view.IWindowManager]
28 alarm: [android.app.IAlarmManager]
29 vibrator: [android.os.IVibratorService]
30 hardware: [android.os.IHardwareService]
31 battery: []
32 content: [android.content.IContentService]
33 account: [android.accounts.IAccountManager]
34 permission: [android.os.IPermissionController]
35 cpuinfo: []
36 meminfo: []
37 activity: [android.app.IActivityManager]
38 package: [android.content.pm.IPackageManager]
39 telephony.registry: [com.android.internal.telephony.ITelephonyRegistry]
40 usagestats: [com.android.internal.app.IUsageStats]
41 batteryinfo: [com.android.internal.app.IBatteryStats]
42 power: [android.os.IPowerManager]
43 entropy: []
44 sensorservice: [android.gui.SensorServer]
45 SurfaceFlinger: [android.ui.ISurfaceComposer]
46 media.audio_policy: [android.media.IAudioPolicyService]
47 media.camera: [android.hardware.ICollectionService]
```

```
48 media.player: [android.media.IMediaPlayerService]
49 media.audio_flinger: [android.media.IAudioFlinger]
```

以下是版本 4.2/ 果冻豆仿真器上同样的输出：

```
root@android:/ # service list
Found 68 services:
0 phone: [com.android.internal.telephony.ITelephony]
1 iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
2 simphonebook: [com.android.internal.telephony.IIccPhoneBook]
3 isms: [com.android.internal.telephony.ISms]
4 dreams: [android.service.dreams.IDreamManager]
5 commontime_management: []
6 samplingprofiler: []
7 diskstats: []
8 appwidget: [com.android.internal.appwidget.IAppWidgetService]
9 backup: [android.app.backup.IBackupManager]
10 uimode: [android.app.IUiModeManager]
11 serial: [android.hardware.ISerialManager]
12 usb: [android.hardware.usb.IUsbManager]
13 audio: [android.media.IAudioService]
14 wallpaper: [android.app.IWallpaperManager]
15 dropbox: [com.android.internal.os.IDropBoxManagerService]
16 search: [android.app.ISearchManager]
17 country_detector: [android.location.ICountryDetector]
18 location: [android.location.ILocationManager]
19 devicestoragemonitor: []
20 notification: [android.app.INotificationManager]
21 updatelock: [android.os.IUpdateLock]
22 throttle: [android.net.IThrottleManager]
23 servicediscovery: [android.net.nsd.INsdManager]
24 connectivity: [android.net.IConnectivityManager]
25 wifi: [android.net.wifi.IWifiManager]
26 wifip2p: [android.net.wifi.p2p.IWifiP2pManager]
27 netpolicy: [android.net.INetworkPolicyManager]
28 netstats: [android.net.INetworkStatsService]
29 textservices: [com.android.internal.textservice.ITextServicesManager]
30 network_management: [android.os.INetworkManagementService]
31 clipboard: [android.content.IClipboard]
32 statusbar: [com.android.internal.statusbar.IStatusBarService]
33 device_policy: [android.app.admin.IDevicePolicyManager]
34 lock_settings: [com.android.internal.widget.ILockSettings]
35 mount: [IMountService]
36 accessibility: [android.view.accessibility.IAccessibilityManager]
37 input_method: [com.android.internal.view.IInputMethodManager]
38 input: [android.hardware.input.IInputManager]
39 window: [android.view.IWindowManager]
40 alarm: [android.app.IAlarmManager]
41 vibrator: [android.os.IVibratorService]
42 battery: []
43 hardware: [android.os.IHardwareService]
44 content: [android.content.IContentService]
45 account: [android.accounts.IAccountManager]
46 user: [android.os.IUserManager]
47 permission: [android.os.IPermissionController]
```

```
48 cpuinfo: []
49 dbinfo: []
50 gfxinfo: []
51 meminfo: []
52 activity: [android.app.IActivityManager]
53 package: [android.content.pm.IPackageManager]
54 scheduling_policy: [android.os.ISchedulingPolicyService]
55 telephony.registry: [com.android.internal.telephony.ITelephonyRegistry]
56 display: [android.hardware.display.IDisplayManager]
57 usagestats: [com.android.internal.app.IUsageStats]
58 batteryinfo: [com.android.internal.app.IBatteryStats]
59 power: [android.os.IPowerManager]
60 entropy: []
61 sensorservice: [android.gui.SensorServer]
62 media.audio_policy: [android.media.IAudioPolicyService]
63 media.camera: [android.hardware.ICameraService]
64 media.player: [android.media.IMediaPlayerService]
65 media.audio_flinger: [android.media.IAudioFlinger]
66 drm.drmManager: [drm.IDrmManagerService]
67 SurfaceFlinger: [android.ui.ISurfaceComposer]
```

不幸的是，并没有特别多的文档来说明这些服务中的每一个是怎么运行的。你必须通过查看每一个服务的源代码，才能知道它如何运行的以及它怎样与其他服务进行交互等精确信息。

## 逆向构建源代码

想要充分了解 Android 系统服务的内部结构，就像是试着吞咽一条鲸鱼。在版本 2.3/ 姜饼中，光系统服务大约有 85 000 行 Java 代码，分布在 100 个不同的文件中。而且这还不包括任何用 C/C++ 写的系统服务的代码。雪上加霜的是对这么多系统服务的评论非常少，并且其设计文档基本不存在。因此如果你想要进一步地深入了解，请准备好极大的耐心。

有一个技巧就是在 Eclipse 里创建一个新的 Java 项目，并把系统服务的代码导入到这个项目中。这些代码不用进行任何编译，但是它可以通过理解代码从而从 Eclipse 的 Java 浏览器功能中获得一些信息。比如说，你可以打开一个单独的 Java 文件，用鼠标右键单击浏览器功能的滚动条区域，选择 Folding → Collapse All。这个操作基本上将所有的方法折叠成一个单独的行，并且行的旁边有一个加号 (+)，这样你将看到整个代码框架树（一个接一个排着的方法名），而不是叶子（每一个方法的实际内容）。这样你就像是在一片森林中。

你可以尝试用一下供应商提供的应用市场中的商业源代码分析工具，比如说 Imagix、Rationale、Lattix 或者是 Scitools。虽然外面也有一些开放源代码的分析工具，但是大多数似乎都有对于定位的漏洞，也不能逆向构建你要分析的代码。仍然有一些报道说他们努力找到的 Ctags 和开源的 AndroidXRef 项目非常有用。

## 服务管理和 Binder 交互

正如我刚刚解释的，Binder 机制作为系统服务的基础结构，使得面向对象的 RPC/IPC 可用。系统中一个进程要通过 Binder 来调用一个系统服务，它首先必须要有一个句柄（handle）。比如说，Binder 允许一个应用程序开发者从电源管理服务（Power Manager）中，通过调用 `WakeLock` 嵌套类的 `acquire()` 方法，来请求一个 wakelock。然而在调用之前，开发者必须首先获得一个对电源管理服务的句柄。正如我们将在下一节看到的，应用程序 API 实际上通过对开发者抽象化来隐藏了它是如何获得句柄的细节，但是在这个隐藏下，所有的系统服务句柄的查找都通过服务管理器来完成，如图 2-5 所示。

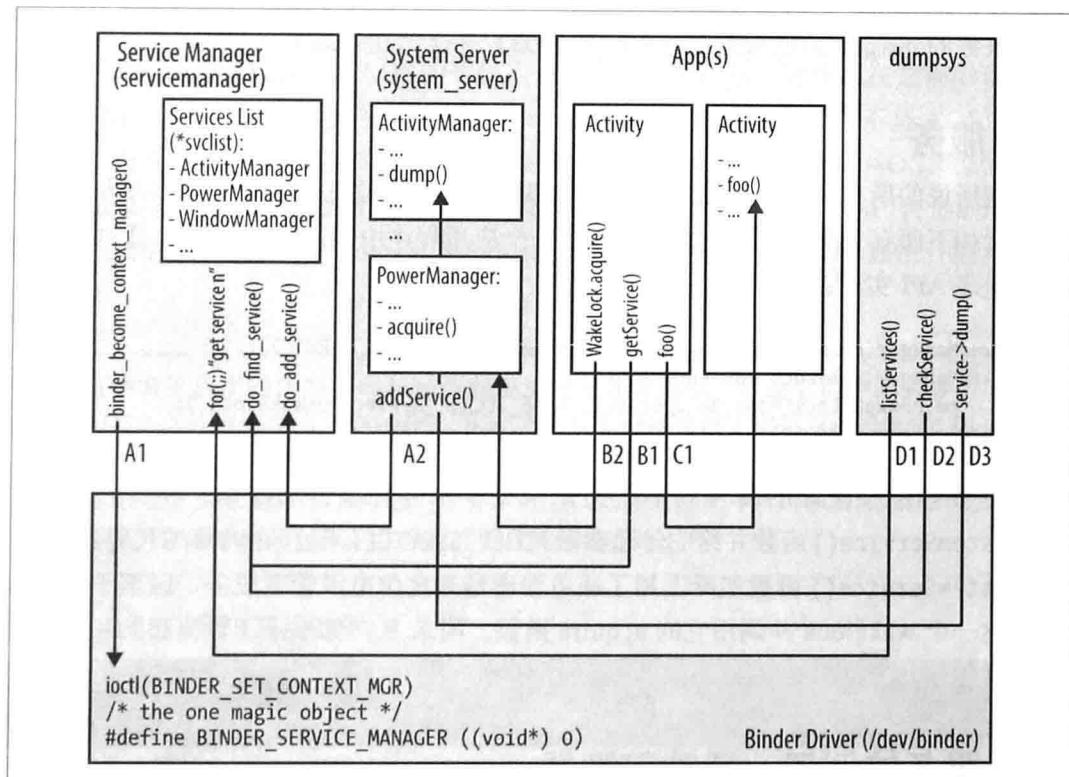


图 2-5：服务管理器与 Binder 交互

可以把服务管理器看作是所有系统中可用的服务的黄皮书。如果一个系统服务没有在服务管理器中注册，那么它在系统的其余地方是无效的。为了提供这种指引能力，服务管理器是被初始化程序在启动其他任何服务之前启动的。之后打开 `/dev/binder`，并用一个特殊的 `ioctl()` 调用来将它设置为 Binder 的上下文管理器（图 2-5 中的 A1）。此后，系统里的任何进程想要与 Binder ID 0（即“神奇的”Binder 或者是各个部分的代码中的“有魔法的对象”）进行通信，实际上就是通过 Binder 与服务管理器通信。

当系统服务器启动以后，比如说，它注册了每一个服务管理器实例化的单独的服务（A2）。之后，当有应用程序想要与一个系统服务通信时，比如说电源管理服务，首先会向服务管理器请求这个服务的句柄（B1），然后调用这个服务的方法（B2）。相比之下，对运行在一个应用程序中的服务组件的调用是直接通过 Binder 的，且不需要通过服务管理器查找。

服务管理器还有一种特殊的使用方式就是被很多命令行工具使用，比如 *dumpsys* 工具，它允许你查出一个或者是所有系统服务的状态。想要获得所有服务的列表，可以用 *dumpsys* 工具循环查找以获得每一个系统服务（D1），在每一个迭代中请求第  $n+1$  个服务，直到没有为止。想要获得其中一个服务，*dumpsys* 工具只要请求服务管理器定位到你要找的那一个就可以了（D2）。当一个服务的句柄在运行时，*dumpsys* 工具调用该服务的 *dump()* 函数来查出它的状态（D3）并显示在终端上。

## 调用服务

我刚刚所说的所有内容，就像之前提到的那样，几乎不被正规的应用程序开发者所看到。比如下面有一个片段，它允许我们在一个应用程序中抢占 wakelock，通过正规的应用程序 API 实现：

```
PowerManager pm = (PowerManager) getSystemService(POWER_SERVICE);
PowerManager.WakeLock wakeLock =
    pm.newWakeLock(PowerManager.FULL_WAKE_LOCK, "myPreciousWakeLock");
wakeLock.acquire(100);
```

请注意我们没在上面的源代码中看到任何关于服务管理器的暗示。相反，我们用 *getSystemService()* 函数并给它传递参数 *POWER\_SERVICE*。不过，在内部的代码实现中，*getSystemService()* 函数实际上用了服务管理器来定位电源管理服务，以至于我们可以创建一个 wakelock 并调用它的 *acquire* 函数。附录 B 向你展示了如何添加一个系统服务并通过 *getSystemService()* 函数让它可用。

## 一个服务的例子：活动管理器

虽然介绍每一个系统服务不在本书范围之内，但是我们可以快速浏览一下活动管理器（核心系统服务之一）。在版本 2.3/ 姜饼中，活动管理器的源代码实际上涵盖超过 30 个文件和 20000 行代码。如果 Android 的内部结构有一个核心的话，活动管理器服务就非常接近这个核心。它负责启动新的组件，比如活动（Activities）和服务（Services），以及获取内容管理器和 Intent 广播。如果你曾经遇到过可怕的 ANR（应用程序无响应）对话框，它背后就是活动管理器。它还涉及维护内核中低内存句柄使用的 OOM 调整、权限、任务管理等。

例如，如果用户在他的界面上单击一个图标来启动一个应用程序，第一件发生的事情就是启动器的 `onClick()` 回调函数被调用（启动器是与 AOSP 打包在一起的默认应用程序，它负责与用户交互的主要界面，即主界面）。想要处理事件，启动器将通过 Binder 调用活动管理服务的 `startActivity()` 函数。之后将调用 `startViaZygote()` 函数，该函数将打开一个到 Zygote 的套接字，并要求它打开活动（Activity）。在你读完本章最后一节时再来看这部分内容可能更有意义。

如果你熟悉 Linux 的内部结构，一个理解活动管理器的好方法是，它对于 Android 的意义就是内核源代码中的 `kernel/` 目录里的内容是什么对于 Linux 的意义。

## 常见的 AOSP 包

在大多数的 Android 设备中 AOSP 附带了一定数量的默认包。就像我在之前的章节里提到的那样，有一些应用程序，比如说 Maps、YouTube 和 Gmail 并不是 AOSP 中的一部分。让我们来看一下那些最引人注意的默认包。下面我们会看到 AOSP 包含的很多包。表 2-7 列出了版本 2.3/ 姜饼的 AOSP 中最重要的常见应用程序；表 2-8 列出了 AOSP 主要的内容提供者（content providers）；表 2-9 列出了相应的 IME（输入法编辑器）。

---

**警告：**虽然常见应用程序的编码与标准的应用程序非常类似，但是它们中的大多数不能在 AOSP 之外用标准的 SDK 编译。因此，如果你想创建以下应用程序的自定制版本（即分拆它），你必须要么在 AOSP 中实现，要么花一段时间来实现应用程序在 AOSP 外用标准的 SDK 编译。另外还有一件事就是，这些应用程序有时候要用 API，这些 API 要在 AOSP 中可用，但是并非通过标准的 SDK 导出。

---

表 2-7：常见的 AOSP 应用程序

AOSP 中的应用程序	显示在启动栏中的名字	描述
账户同步设置	N/A	账户管理应用程序
蓝牙	N/A	蓝牙管理器
浏览器	Browser	默认的 Android 浏览器，包含书签部件
计算器	Calculator	计算器应用程序
日历	Calendar	日历应用程序
照相机	Camera	照相机应用程序
证书安装程序	N/A	安装证书的用户界面

表 2-7：常见的 AOSP 应用程序（续）

AOSP 中的应用程序	显示在启动栏中的名字	描述
联系人	Contacts	联系人管理应用程序
桌面时钟	Clock	时钟和警报应用程序，包含时钟部件
下载管理器用户界面	Downloads	下载管理器的用户界面
开发工具	Dev Tools	各种开发工具
电子邮件	Email	默认的 Android 电子邮件应用程序
图库	Gallery	默认的浏览图片的图库应用程序
3D 图库	Gallery	神奇的更“生动”的用户界面的图库
HTML 阅读器	N/A	浏览 HTML 文件的应用程序
启动栏 2	N/A	默认的主屏幕
彩信	Messaging	SMS/MMS 应用程序
音乐	Music	音乐播放器
近场通信	N/A	NFC 配置用户界面和 NFC 系统服务
打包安装程序	N/A	应用程序安装 / 卸载用户界面
电话	Phone	默认的电话拨号器 / 用户界面和电话系统服务
主屏提示	N/A	主屏幕提示信息
Provision	N/A	设置一个标志指示一个设备是否被分配的应用程序
快速搜索	Search	搜索应用程序及部件
设置	Settings	设置应用程序，也可以通过主界面菜单访问
录音机	N/A	声音记录应用程序。当发送录音 Intent 时被激活，而不是由用户激活
通话录音	Speech Recorder	通话录音应用程序
状态栏	N/A	状态栏

表 2-8：常见的 AOSP 提供者

提供者	描述
ApplicationsProvider	搜索安装的应用程序的 Provider
CalendarProvider	主 Android 日历存储的 Provider
ContactsProvider	主 Android 联系人存储的 Provider
DownloadProvider <sup>a</sup>	下载管理、存储和访问
DrmProvider	DRM 保护存储的访问和管理

表 2-8：常见的 AOSP 提供者（续）

提供者	描述
MediaProvider	媒体存储的 Provider
TelephonyProvider	运营商和 SMS/MMS 存储的 Provider
UserDictionaryProvider	用户自定义字典的存储的 Provider

a. 有趣的是，这个包的源代码里包含一个设计文档，这在 AOSP 中是非常宝贵的。

表 2-9：常见的 AOSP 输入方法

输入方法	描述
LatinIME	Latin 键盘
OpenWnn	日文键盘
PinyinIME	中文键盘

AOSP 中包含了比上面表中列出的更多的包。事实上，如果你搜索源代码，就会发现一个 4.2/ 果冻豆发布版本中就有大约 500 个应用程序。它们中的很大一部分或者是测试用例或者是例子，并不值得我们重点讨论。这些应用程序中大约有四分之一值得放到最终的产品中，它们大多数也能在下面的 AOSP 目录中找到：

- *packages/apps/*
- *packages/inputmethods/*
- *packages/providers/*
- *packages/screensavers/* (更新到版本 4.2/Jelly Bean)
- *packages/wallpapers/*
- *frameworks/base/packages/*
- *development/apps/*

你可能想要看看这些目录的内容，结合上面表格中的内容来决定在你的项目里哪些包值得深入研究它的内容。当然，像 AOSP 中很多其他的事一样，这些包包含的内容随着时间改变而改变，它们所在的位置也是。这里总结一下在版本 2.3.4/ 姜饼和版本 4.2/ 果冻豆中，一些包位置的变化：

- 目录 *packages/apps/* 中删除了账户同步设置 (*AccountAndSyncSettings*) 和 3D 图形库 (*Gallery3D*)，并增加了以下的包：*CellBroadcastReceiver*、*SmartCardService*、*BasicSmsReceiver*、*Exchange*、*Gallery2*、*KeyChain*、*MusicFX*、*SpareParts*、*VideoEditor* 和 *LegacyCamera*。

- 目录 `frameworks/base/packages/` 中删除了 `TtsService` 和 `VpnServices`，并增加了以下的包：`BackupRestoreConfirmation`、`SharedStorageBackup`、`VpnDialogs`、`WAPPushManager`、`FakeOemFeatures`、`FusedLocation` 和 `InputDevices`。

## 系统启动项

将我们所介绍的所有东西结合在一起的最好方法是 Android 的启动过程。就像你看到的图 2-6，第一个要介绍的就是 CPU。通常从它的第一个指令中会取出一个硬编码地址。这个地址通常指向一个具有引导程序的芯片。之后这个引导程序（bootloader）初始化 RAM，将基本的硬件保持为静止状态，装载内核和 RAM 磁盘，并跳转至内核。现在很多的 Soc 设备包含一个 CPU 和在同一个芯片上的大量外设，它们实际上可以直接从一个正确格式的 SD 卡或者是类似于 SD 卡的芯片上启动。比如说 PandaBoard 和最新版本的 BeagleBoard，没有任何板载的闪存芯片，因为它们是从 SD 卡上启动的。

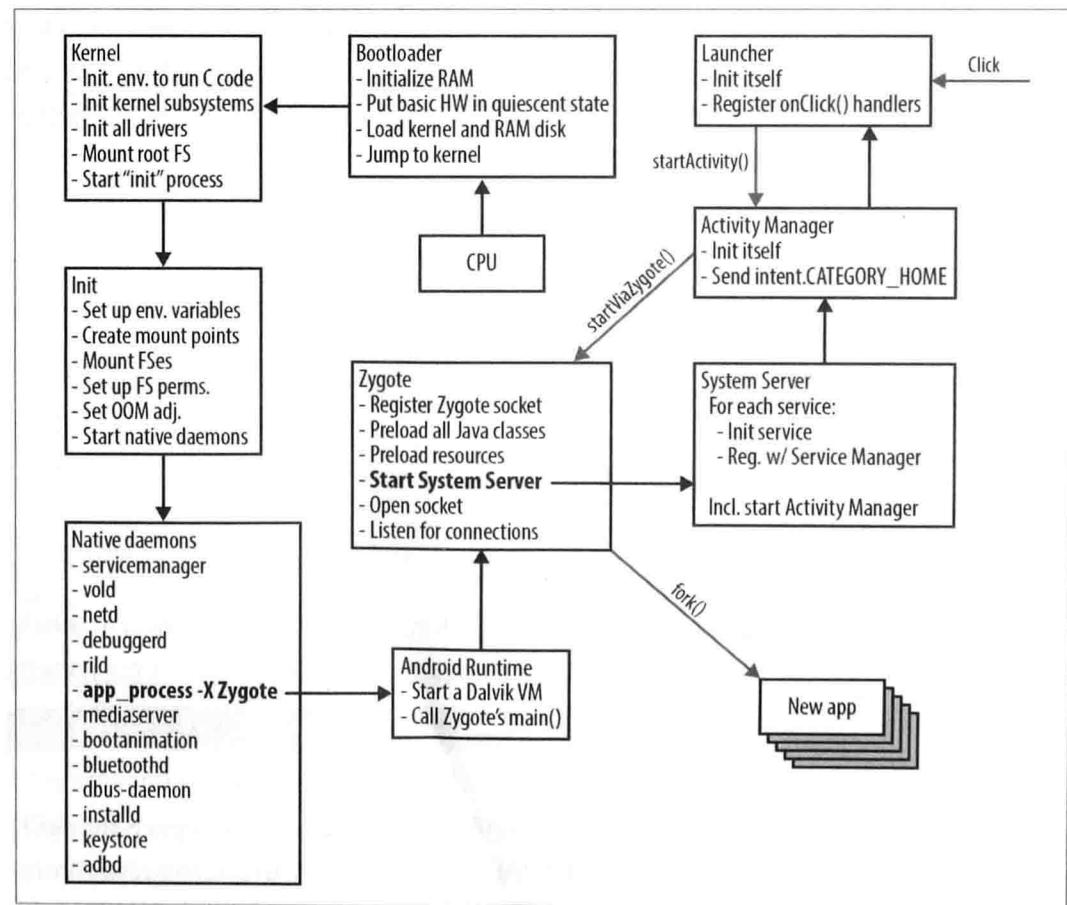


图 2-6：Android 的启动序列

最初的内核启动程序非常依赖于硬件，但它的目的是启动系统，以便 CPU 可以尽可能早地开始执行 C 代码。一旦完成了，内核跳转到体系结构独立的 `start_kernel()` 方法中，初始化它的各种子系统，并继续调用所有驱动内置的“初始化”函数。在启动阶段，这些步骤的大多数信息会被内核打印出来。之后内核挂载它的根文件系统并启动 Init 进程。

Android 的初始化程序开始启动并执行存储在 `/init.rc` 文件中的指令集，来设置环境变量，比如说系统路径、创建挂载位置、挂载文件系统、设置 OOM 调整，以及启动本地守护进程。我们几乎已经介绍了各种 Android 中的本地守护进程，但是还有一点值得关注，那就是 Zygote，Zygote 是一个特殊的守护进程，它的工作是启动应用程序。它的功能总体来说是为了统一所有应用程序共享的组件，以缩短它们的启动时间。初始化程序实际上并不直接启动 Zygote，而 Android Runtime 会调用 `app_process` 命令来启动 Zygote。之后 runtime 会启动系统的第一个 Dalvik 虚拟机，并告诉它调用 Zygote 的 `main()` 函数。

Zygote 只有在一个新的应用程序需要被启动时才处于激活状态。为了实现更快的应用程序启动，Zygote 通过预加载一个应用程序在运行时可能需要的所有 Java 类和源代码来启动。这样有效地将这些信息加载到了系统的 RAM 中。之后 Zygote 倾听它的套接字 (`/dev/socket/zygote`) 上是否有启动新应用程序的请求连接。如果有启动新应用程序的请求，它创建自己的子进程并启动新的应用程序。所有的应用程序都让 Zygote 创建一个子进程，这样做最好的一点是，这个“纯洁的”虚拟机中有一个应用程序可能需要预安装运行的所有系统类和源代码。换句话说，应用程序不需要等所有这些资源装载后才开始运行。

所有的这些都是因为 Linux 内核为创建子进程实现了写时才复制 (copy-on-write, COW) 策略。就像你所知道的，UNIX 中的分叉 (forking) 涉及创建一个新的进程，这个进程是它的父进程的完全复制。但是在 COW 中，Linux 实际上并没有复制任何东西。相反，它们映射新进程的页面到它们的父进程，并且只在新进程写页面时才复制。但是实际上，装载的类和源代码几乎从未被写过，这是因为它们都是默认的，在整个系统的生命周期中几乎都是一成不变的。所以所有直接从 Zygote 创建的进程基本上是使用它们自身的映射复制。因而，不管有多少数量的应用程序在系统中运行，只有一份系统类和源代码的复制加载到了 RAM 上。

虽然 Zygote 设计 forking 来实现新的应用程序请求的连接，但是还有一个“应用程序”实际上是由 Zygote 明确启动的：系统服务 (System Server)。这是 Zygote 启动的第一个应用程序，并且从它的父进程创建出来后一直以一个完全独立的进程存在。之后系统服务 (System Server) 应用程序开始初始化每一个它包含的系统服务，并用之前

启动的服务管理器来注册它们。它启动其中一个服务，即活动管理器，通过发送一个 Intent.CATEGORY\_HOME 类型的 Intent 来完成它的初始化。这样之后启动启动器应用程序，这个应用程序向所有 Android 用户展现他们熟悉的主界面。

当用户单击主界面上的一个图标时，我在本章前面的“一个服务的例子：活动管理器”一节中描述的进程启动。启动器请求活动管理器启动这个进程，按顺序向前请求启动 Zygote，它分叉新的进程并启动新的应用程序，之后这些应用程序就显示给用户。

系统完成启动以后，进程列表看起来如下所示：

```
# ps
USER     PID   PPID  VSIZE   RSS    WCHAN    PC          NAME
root      1     0    268    180  c009b74c 0000875c S /init
root      2     0     0     0  c004e72c 00000000 S kthreadd
root      3     2     0     0  c003fdc8 00000000 S ksoftirqd/0
root      4     2     0     0  c004b2c4 00000000 S events/0
root      5     2     0     0  c004b2c4 00000000 S khelper
root      6     2     0     0  c004b2c4 00000000 S suspend
root      7     2     0     0  c004b2c4 00000000 S kblockd/0
root      8     2     0     0  c004b2c4 00000000 S cqueue
root      9     2     0     0  c018179c 00000000 S kseriod
root     10     2     0     0  c004b2c4 00000000 S kmmcd
root     11     2     0     0  c006fc74 00000000 S pdflush
root     12     2     0     0  c006fc74 00000000 S pdflush
root     13     2     0     0  c0079750 00000000 D kswapd0
root     14     2     0     0  c004b2c4 00000000 S aio/0
root     22     2     0     0  c017ef48 00000000 S mtblockquote
root     23     2     0     0  c004b2c4 00000000 S kstriped
root     24     2     0     0  c004b2c4 00000000 S hid_compat
root     25     2     0     0  c004b2c4 00000000 S rpciod/0
root     26     1    232    136  c009b74c 0000875c S /sbin/ueventd
system   27     1    804    216  c01a94a4 afdb0b6fc S /system/bin/servicemanager
root     28     1   3864    308  ffffffff afdb0bdac S /system/bin/vold
root     29     1   3836    304  ffffffff afdb0bdac S /system/bin/netd
root     30     1    664    192  c01b52b4 afdb0c0cc S /system/bin/debuggerd
radio    31     1   5396    440  ffffffff afdb0bdac S /system/bin/rild
root     32     1  60832   16348  c009b74c afdb0b844 S zygote
media   33     1  17976   1104  ffffffff afdb0b6fc S /system/bin/mediaserver
bluetooth 34     1   1256    280  c009b74c afdb0c59c S /system/bin/dbus-daemon
root     35     1    812    232  c02181f4 afdb0b45c S /system/bin/installld
keystore 36     1   1744    212  c01b52b4 afdb0c0cc S /system/bin/keystore
root     38     1    824    272  c00b8fec afdb0c51c S /system/bin/qemuud
shell    40     1    732    204  c0158eb0 afdb0b45c S /system/bin/sh
root     41     1   3368    172  ffffffff 00008294 S /sbin/adbd
system   65     32  123128   25232  ffffffff afdb0b6fc S system_server
app_15   115    32  77232   17576  ffffffff afdb0c51c S com.android.inputmethod.
                             latin
radio    120    32  86060   17952  ffffffff afdb0c51c S com.android.phone
system   122    32  73160   17656  ffffffff afdb0c51c S com.android.systemui
app_27   125    32  80664   22900  ffffffff afdb0c51c S com.android.launcher
app_5    173    32  74404   18024  ffffffff afdb0c51c S android.process.acore
app_2    212    32  73112   17032  ffffffff afdb0c51c S android.process.media
```

```
app_19      284    32    70336   16672  ffffffff afd0c51c S com.android.bluetooth
app_22      292    32    72752   17844  ffffffff afd0c51c S com.android.email
app_23      320    32    70276   15792  ffffffff afd0c51c S com.android.music
app_28      328    32    70744   16444  ffffffff afd0c51c S com.android.quicksearchbox
app_14      345    32    69708   15404  ffffffff afd0c51c S com.android.protips
app_21      354    32    70912   17152  ffffffff afd0c51c S com.cooliris.media
root        366    41    2128    292    c003da38 00110c84 S /bin/sh
root        367    366   888     324    00000000 afd0b45c R /system/bin/ps
```

这些输出实际上来自于装载版本 2.3/ 姜饼系统的 Android 仿真器，所以它包含了一些仿真器特有的假象，比如 *qemud* 守护进程。请注意，运行中的应用程序都要有完全合格的包名，即使是被 Zygote 分叉出去以后也一样。有一个技巧可以用在 Linux 里，就是用 `prctl()` 系统调用 `PR_SET_NAME` 来告诉内核修改调用进程的名字。如果你对此感兴趣的话，可以看一下 `prctl()` 的手册页。还要注意的是，第一个被初始化程序启动的进程实际上是 *ueventd* 进程。在此之前的所有进程实际上都是在内核被子系统或者驱动启动的。

最重要的是，请注意 Zygote 进程的进程号(PID)为 32，且所有应用程序的父进程(PPID)的进程号也是 32。这证明了之前的说法是正确的，Zygote 是系统中所有应用程序的父进程。

# AOSP 入门

现在，我们对基础知识有了足够的了解，让我们开始接触 Android 的开源项目（AOSP）吧。我们的介绍从告诉你如何从 <http://android.googlesource.com/> 上获得 AOSP 发布版本开始。在真正编译和运行 AOSP 之前，我们将花一段时间来探究 AOSP 的内容，以及解释 AOSP 的源代码对我们之前章节提到的东西有怎样的影响。最后，在结束本章前，我们会介绍如何使用 adb 和仿真器，这两种工具对于完成任何平台工作都非常重要。

总之，这一章是非常有趣的。AOSP 是一个令人兴奋的具有大量创新思想的软件。好了，我承认它并不是完美的，它也有一些部分并不完善。但是除此之外的其他部分仍然令人赞叹。其中最令人惊奇的就是大家都可以下载它，修改它，并且在它的基础上定制用户自己的产品。所以，卷起你的袖子，让我们开始吧。

## 开发主机设置

就像我们在第 1 章“开发工具及其环境搭建”里讨论的那样，你需要一个基于 Ubuntu 的桌面以便来做有关 AOSP 的开发。虽然其他系统也可以用来开发，但 Ubuntu 是 Google 文档中支持的一个系统。我建议你翻回前面重新读一下上面提到的章节，复习一下如何进行 AOSP 开发所需的基本的主机设置。另外，我建议你看一下 Google 网页 <http://source.android.com> 上的“初始化一个编译环境” (<http://source.android.com/source/initializing.html>) 这一节，了解一下之后会介绍的怎样设置你的主机来编译 Android 的源代码。这个网页也介绍了如何配制 udev 来确保你的权限设置正确，从而让你顺利访问一个连接到你的主机的 Android 设备。

# 下载 AOSP

就如我前面提到的，官方的 AOSP 可以从 <http://android.googlesource.com> 上下载，这个网页介绍了图 3-1 中显示的 Gitweb 接口（git 的网络接口）。当你访问这个网页时，将会看到一个可用的相当大的 git 知识库。不用说，如果你手动地下载每一个资源将非常麻烦，至少有超过 100 个。而且实际上将它们所有的都下载下来也没有多大的作用，因为你只需要这些项目的一部分。最正确的下载 AOSP 的方法是用 *repo* 工具，这个工具非常适用于类似这样的情况。首先，你需要自己下载 *repo* 工具：

```
$ sudo apt-get install curl  
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo  
$ chmod a+x ~/bin/repo
```

The screenshot shows a Mozilla Firefox browser window displaying the 'android Git repositories' page on Google's Googlesource site. The URL in the address bar is <https://android.googlesource.com/>. The page lists numerous Android projects under the heading 'android Git repositories'. Each project entry includes a 'Name' (e.g., 'Kernel', 'Platform-Projects', 'Platform-Unselected-Projects', 'Project-Projects', etc.) and a 'Description' providing details about the project's purpose and maintainers. The interface includes standard browser controls like back, forward, and search, along with links for 'Code Review', 'Generate New Password', 'Revoke Prior Passwords', and 'Sign In'.

图 3-1：Android Git 知识库主页

---

**警告：**在 Ubuntu 中，当你登录以后，`~/bin` 是自动加入到路径中的，前提是如果这个目录存在的话。所以，如果你的主目录里没有 `bin/` 目录，就创建一个，然后退出登录再登录进去，它就会加入到你的路径中了。否则，shell 就找不到 `repo` 工具，即使你像我刚刚展示的那样操作也不行。

如果这样也不行，不管是在 Ubuntu 系统还是你用的其他系统中都一样，就请手动在 `~/.profile` 文件中加入一句 `PATH=$PATH:~/bin`，之后退出登录再重新登录。

---

**注意：**你不用将 `repo` 放到 `~/bin` 中，但是它必须要在你的路径中。因此，无论你把它放在哪里，只要保证命令行中它在文件系统中的位置是可用的。

---

尽管 `repo` 工具的结构只是一个单一的 shell 脚本，但是它实际上仍然是一个相当复杂的工具。它可以同时下载多个 `git` 知识库到本地来创建一个 Android 发布版本。它下载的这些知识库都是通过 `manifest` 文件来发挥作用，`manifest` 文件是一个 XML 文件，记录了需要下载的项目和它们的位置。实际上 `repo` 工具位于 `git` 的层面之上，它下载的每一个项目都是一个独立的 `git` 知识包。你想要找到更多关于是什么推动了 Google 创建 `repo` 工具的信息，可以看一下 Gerrit 和 Repo 的博客文章，Android 的源代码管理工具 (<http://google-opensource.blogspot.com/2008/11/gerrit-and-repo-android-source.html>)，文章是在 2008 年 11 月发表的，也就是在 Android 的第一个开源版本发布之后不久。

---

**警告：**请注意，令人费解的是，`repo` 工具的“`manifest`”文件与应用程序开发者用来向系统描述他们的应用程序的“`manifest`”文件完全没有关系。它们的格式和用途完全不同。幸运的是，它们很少被用于同一范围内，所以你应该记住，在接下来的描述中我们没有必要担心和解释太多。

---

在使用 `repo` 工具之前，你需要确保 `git` 已经安装在你的系统里了，因为它可能不是默认安装的：

```
$ sudo apt-get install git
```

现在已经安装好 `repo` 和 `git` 了，我们来下载一个 AOSP 的副本：

```
$ mkdir -p ~/android/aosp-2.3.x
$ cd ~/android/aosp-2.3.x
$ repo init -u https://android.googlesource.com/platform/manifest.git
-b gingerbread
$ repo sync
```

最后一条命令应该会运行相当长一段时间，因为它要下载 `manifest` 文件中的所有项

目的源代码。毕竟，AOSP 在未编译时有几个 GB 大小，这在第 1 章的“开发工具及其环境搭建”已经提到过。要注意的是网络宽带和延迟对所花时间的影响非常大。还要注意的是我们提取的是所有内容里一个特定的分支，即姜饼（Gingerbread）。也就是第三个命令中的 `-b gingerbread` 部分。如果你去掉命令中的这部分，你会得到所有的分支。很多人的经验是，主分支并不一定都是正常编译和运行的，因为它包含了开放开发分支的信息。另一方面，标记的分支大多数是在框架外运行。如果你想要给 AOSP 做一些修改，那么请注意，Google 只接受对于主分支的修改意见。

如果你想要获得更多关于 `repo` 性能的信息，可以用它的在线帮助：

```
$ repo help
usage: repo COMMAND [ARGS]

The most commonly used repo commands are:

abandon      Permanently abandon a development branch
branch       View current topic branches
branches     View current topic branches
checkout     Checkout a branch for development
cherry-pick   Cherry-pick a change.
diff         Show changes between commit and working tree
download    Download and checkout a change
grep         Print lines matching a pattern
init         Initialize repo in the current directory
list          List projects and their associated directories
overview    Display overview of unmerged project branches
prune        Prune (delete) already merged topics
rebase      Rebase local branches on upstream branch
smartsync   Update working tree to the latest known good revision
stage        Stage file(s) for commit
start        Start a new branch for development
status       Show the working tree status
sync         Update working tree to the latest revision
upload      Upload changes for code review

See 'repo help <command>' for more information on a specific command.
See 'repo help --all' for a complete list of recognized commands.
```

上面的输出表明，你还可以获得有关 `repo` 的任何子命令的更多信息：

```
$ repo help init
Summary
-----
Initialize repo in the current directory

Usage: repo init [options]

Options:
-h, --help           show this help message and exit
```

```
Logging options:  
  -q, --quiet      be quiet  
  
Manifest options:  
  -u URL, --manifest-url=URL  
                        manifest repository location  
  -b REVISION, --manifest-branch=REVISION  
                        manifest branch or revision  
  -m NAME.xml, --manifest-name=NAME.xml  
                        initial manifest file  
  --mirror          create a replica of the remote repositories rather  
                    than a client working directory  
  --reference=DIR    location of mirror directory  
  --depth=DEPTH     create a shallow clone with given depth; see git clone  
  -g GROUP, --groups=GROUP  
                        restrict manifest projects to ones with a specified  
                        group  
  -p PLATFORM, --platform=PLATFORM  
                        restrict manifest projects to ones with a specified  
                        platform group [auto|all|none|linux|darwin|...]  
  
repo Version options:  
  --repo-url=URL    repo repository location  
  --repo-branch=REVISION  
                    repo branch or revision  
  --no-repo-verify   do not verify repo source code  
  
Other options:  
  --config-name     Always prompt for name/e-mail
```

#### Description

-----  
The 'repo init' command is run once to install and initialize repo. The latest repo source code and manifest collection is downloaded from the server and is installed in the .repo/ directory in the current working directory.

The optional -b argument can be used to select the manifest branch to checkout and use. If no branch is specified, master is assumed.

The optional -m argument can be used to specify an alternate manifest to be used. If no manifest is specified, the manifest default.xml will be used.

The --reference option can be used to point to a directory that has the content of a --mirror sync. This will make the working directory use as much data as possible from the local reference directory when fetching from the server. This will make the sync go a lot faster by reducing data traffic on the network.

#### Switching Manifest Branches

-----  
To switch to another manifest branch, `repo init -b otherbranch` may be

used in an existing client. However, as this only updates the manifest, a subsequent `repo sync` (or `repo sync -d`) is necessary to update the working directory files.

当你在看 *repo sync* 在线帮助时，很可能你会想要更进一步地了解一个标记 *-j*，因为它允许并行的同步几个 *git* 树。这对你来说非常有用，如果你有了一个高速的企业网络连接，想要加快你下载 AOSP 的速度——默认的，*repo* 只会开启四个线程并行下载：

```
$ repo sync -j8
```

获取其他分支和标签也非常简单。这里是版本 4.2/ 果冻豆的例子：

```
$ mkdir -p ~/android/aosp-4.2
$ cd ~/android/aosp-4.2
$ repo init -u https://android.googlesource.com/platform/manifest
-b android-4.2_r1

$ repo sync
```

与前面的命令不同，我用了一个特殊的版本号而不是版本名。开发代号、标签以及编译号 (<http://source.android.com/source/build-numbers.html>) 提供了官方标签和版本号的完整列表。你可以像下面这样做来找到你要用的标签和分支<sup>注1</sup>：

```
$ mkdir ~/android/aosp-branches-tags
$ cd ~/android/aosp-branches-tags
$ git clone https://android.googlesource.com/platform/manifest.git
$ cd manifest
$ git tag
android-1.6_r1.1_
android-1.6_r1.2_
android-1.6_r1.3_
android-1.6_r1.4_
android-1.6_r1.5_
android-1.6_r1_
android-1.6_r2_
android-2.0.1_r1_
android-2.0_r1_
android-2.1_r1_
android-2.1_r2.1p2_
android-2.1_r2.1p_
...
android-4.1.1_r6
android-4.1.1_r6.1
android-4.1.2_r1
android-4.2.1_r1_
android-4.2_r1_
android-cts-2.2_r8
android-cts-2.3_r10
```

---

注 1：感谢 Linaro 写的 Bernhard Rosenkranzer 给我们指出了这个很有用的技巧。

```
android-cts-2.3_r11
...
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/android-1.6_r1
  remotes/origin/android-1.6_r1.1
  remotes/origin/android-1.6_r1.2
  remotes/origin/android-1.6_r1.3
  remotes/origin/android-1.6_r1.4
  remotes/origin/android-1.6_r1.5
  remotes/origin/android-1.6_r2
  remotes/origin/android-2.0.1_r1
  remotes/origin/android-2.0_r1
  remotes/origin/android-2.1_r1
  remotes/origin/android-2.1_r2
  remotes/origin/android-2.1_r2.1p
  remotes/origin/android-2.1_r2.1p2
...
  remotes/origin/android-4.1.1_r6.1
  remotes/origin/android-4.1.2_r1
  remotes/origin/android-4.2.1_r1
  remotes/origin/android-4.2_r1
  remotes/origin/android-cts-2.2_r8
  remotes/origin/android-cts-2.3_r10
  remotes/origin/android-cts-2.3_r11
...
  remotes/origin/android-sdk-support_r11
  remotes/origin/froyo
  remotes/origin/gingerbread
  remotes/origin/gingerbread-release
  remotes/origin/ics-mr0
  remotes/origin/ics-mr1
  remotes/origin/ics-plus-aosp
  remotes/origin/jb-dev
  remotes/origin/jb-mr1-dev
  remotes/origin/jumper-stable
  remotes/origin/master
  remotes/origin/master-dalvik
  remotes/origin/tools_r20
  remotes/origin/tools_r21
  remotes/origin/tools_r21.1
  remotes/origin/tradefed
```

当然，以上所说的所有都仅限于官方的 AOSP。你可以看一下附录 E，那里有一个可能与你的工作有关的其他 AOSP 树的列表，比如那些由 Linaro 和 CyanogenMod 维护的 AOSP 树。有趣的是，大多数这些非官方的树也依赖于 *repo* 工具，这就使我们有更多的理由去学习如何掌握这个工具。

# AOSP 的内部

现在，我们已经获得了一个 AOSP 的副本，让我们来看看它的内部，更重要的是，跟之前我们在前面章节所说的内容做一个联系。如果你随意跳过这一节，在下一节之后如果你还是不能让你自己定制的 Android 系统运行的话，再回来看这一节。如果你还在看这一节的话，请你看一下表 3-1，它是版本 2.3.7/ 姜饼和版本 4.2/ 果冻豆对 AOSP 的顶层目录的一个总结。其中有目录的某个版本的“大小”这一列填的信息是“N/A”，这是因为这个目录在该版本中并不存在。同样，“大小”这一列所填的大小并不包含 .git 目录，因为这个目录底下可能包含任何指定的条目。

表 3-1：AOSP 的内容总结

目录	内容	在版本 2.3.7 的大小 (MB)	在版本 4.2 的 大小 (MB)
<i>abi</i>	支持最小的 C++ 实时类型信息	N/A	0.1
<i>bionic</i>	Android 定制的 C 类库	14	18
<i>bootable</i>	OTA，恢复机制和参考引导程序	4	4
<i>build</i>	编译系统	4	5
<i>cts</i>	性能测试套件	77	136
<i>dalvik</i>	Dalvik 虚拟机	35	40
<i>development</i>	开发工具	64	87
<i>device</i>	特定设备文件和组件	17	43
<i>docs</i>	<a href="http://source.android.com">http://source.android.com</a> 的内容	N/A	6
<i>external</i>	导入到 AOSP 中的外部项目	849	1595
<i>frameworks</i>	核心组件，比如系统服务等	360	1150
<i>gdk</i>	未知 <sup>a</sup>	N/A	5
<i>hardware</i>	HAL 和硬件支持库	27	52
<i>libcore</i>	Apache Harmony 项目	54	40
<i>libnativehelper</i> <sup>b</sup>	使用 JNI 的帮助功能	N/A	0.1
<i>ndk</i>	本地开发包	13	31
<i>packages</i>	常见 Android 应用程序的提供者和 IME	115	278
<i>pdk</i>	平台开发包	N/A	0.3
<i>prebuilt</i>	预编译库，包括工具链	1389	N/A
<i>prebuilts</i>	替换 <i>prebuilt</i>	N/A	2387

表 3-1：AOSP 的内容总结（续）

目录	内容	在版本 2.3.7 的大小 (MB)	在版本 4.2 的 大小 (MB) )
<i>sdk</i>	软件开发包	14	54
<i>system</i>	装载 Android 系统的“嵌入式 Linux”平台	32	9
<i>tools</i>	各种 IDE 工具	N/A	34

- a. 尽管多次尝试，这个目录除了有一些关于 NDK 和 LLVM 的文件以外，我还是一直无法确定这个目录的目的是什么。即使是 git 日志也没有暗示它的缩写表示的是什么。它在将来可能是作为实验代码来用。
- b. 这是版本 2.3.2 中 *dalvik/* 的子目录。

如你所见，*prebuilt*（在版本 4.2/ 果冻豆中是 *prebuilts*）和 *external* 是树中最大的两个目录，在版本 2.3.7/ 姜饼中它们的大小所占的比例接近 75%，且在版本 4.2/ 果冻豆中它们的大小所占的比例超过 65%。有趣的是，这些目录中的大多数都是由其他的开源项目的内容组成的，包括的内容比如说各种 GNU 工具链版本，内核映像，公共库和框架（例如 OpenSSL 和 WebKit 等）。*libcore* 也是从其他的开源项目（即 Apache Harmony）中得来。本质上，这也进一步证明了 Android 是如何依赖于存在的其余开放源代码生态系统的。Android 中仍然还包含了一部分“原始的”（或接近原始的）源代码：在版本 2.3.7/ 姜饼中超过 800MB；在版本 4.2/ 果冻豆中超过 2GB。

为了更好地了解 Android 的源代码，重新返回去看一下图 2-1 非常有意义，图 2-1 描述了 Android 的体系结构。图 3-2 在图 2-1 的基础上做了改变，它说明了 AOSP 源代码中每个 Android 组件的位置。显然，很多关键的组件来自于目录 *frameworks/base/*，这个目录是大半 Android 的“大脑”所在的位置。实际上这个目录和目录 *system/core/* 值得更仔细的研究，分别见表 3-2 和表 3-3，因为它们中包含了很大一块有意思的部分，可能你会感兴趣或者在嵌入式开发中修改它们。

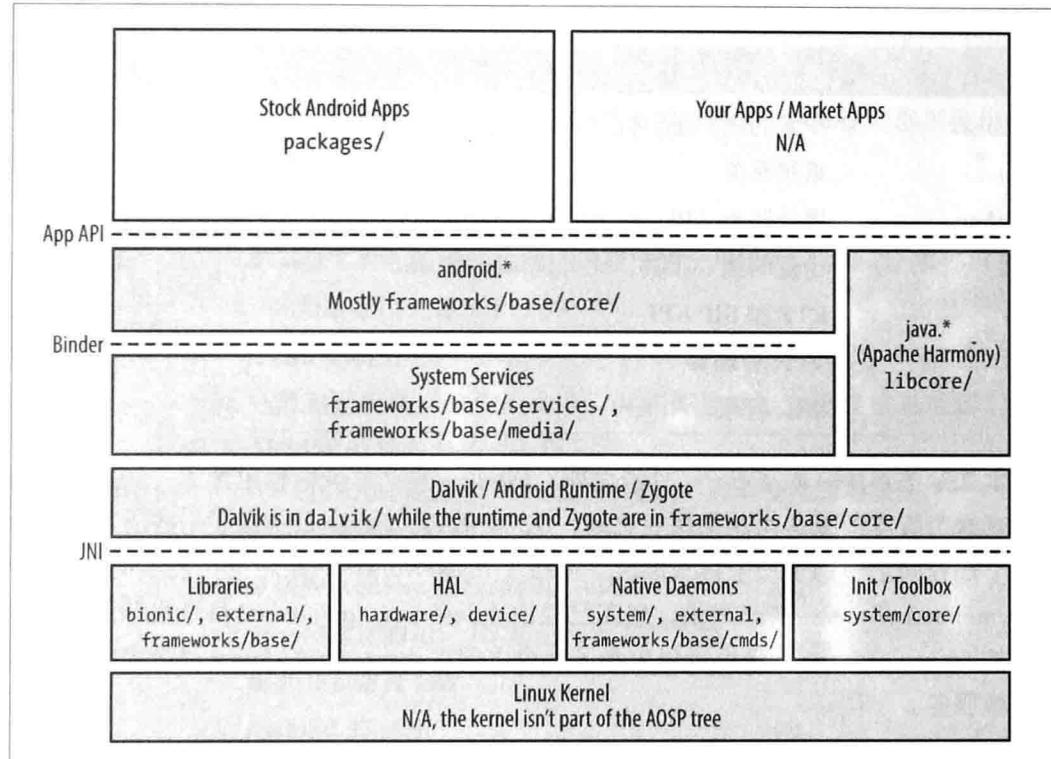


图 3-2: Android 的体系结构

表 3-2: 版本 2.3/ 姜饼中目录 frameworks/base/ 的内容总结

目录	内容
<i>cmds</i>	框架相关的命令和守护进程
<i>core</i>	<i>android.*</i> 包
<i>data</i>	字体和声音
<i>graphics</i>	2D 图库和 Renderscript
<i>include</i>	C 语言头文件
<i>keystore</i>	安全密钥存储
<i>libs</i>	C 函数库
<i>location</i>	位置提供者
<i>media</i>	媒体服务, StageFright, 编码解码器等
<i>native</i>	一些框架组件的本地代码
<i>obex</i>	蓝牙 Obex
<i>opengl</i>	OpenGL 库和 Java 代码

表 3-2：版本 2.3/ 姜饼中目录 frameworks/base/ 的内容总结（续）

目录	内容
<i>packages</i>	少量的核心包，比如说状态栏
<i>services</i>	系统服务
<i>telephony</i>	拨号服务 API，与 <i>rild</i> 无线电层接口进行通信
<i>tools</i>	少量的核心工具，比如说 <i>aapt</i> 和 <i>aidl</i>
<i>voip</i>	RTP 和 SIP API
<i>vpn</i>	VPN 管理器
<i>wifi</i>	Wifi 管理器和 API

在版本 2.3/ 姜饼中，除了 *base/* 目录之外，*frameworks/* 目录中也包含了一些其他的目录。在这个版本到版本 4.2/ 果冻豆之间，*frameworks/base/* 目录做了一些清理，其中的几个部分向上移动了一个目录层次，放在了 *frameworks/* 目录中（见表 3-4）。比如说 *frameworks/base/media/* 目录，现在已经由 *frameworks/av/media/* 目录代替。此外，现在的 *frameworks/native/* 目录中包含了几个之前在 *frameworks/base/* 目录中的本地库和系统服务。

表 3-3：版本 2.3/ 姜饼中的 system/core/ 目录内容总结

目录	内容
<i>adb</i> <sup>a</sup>	ADB 守护进程和客户端
<i>cpio</i>	<i>mkbootfs</i> 工具，用于生成 RAM 磁盘镜像 <sup>b</sup>
<i>debuggerd</i>	第 2 章提到以及第 6 章要介绍的 <i>debuggerd</i> 命令
<i>fastboot</i>	<i>fastboot</i> 使用程序，通过“快速启动（fastboot）”协议与 Android 的引导程序进行通信
<i>include</i>	所有“系统（system）”相关的 C 语言头文件
<i>init</i>	Android 的初始化程序（ <i>init</i> ）
<i>libacc</i>	用于编译类 C 代码的“几乎是”C 编译器的库；在版本 2.3/ 姜饼 <sup>c</sup> 中被 RenderScript 使用
<i>libcutils</i>	各种 C 语言公用函数，不是标准 C 库的一部分；在树中贯穿使用
<i>libdiskconfig</i>	用于读和配置磁盘；通过 <i>vold</i> 使用
<i>liblinenoise</i>	从 <a href="http://github.com/antirez/linenoise">http://github.com/antirez/linenoise</a> 替换的 BCD 许可的 <i>readline()</i> 函数；Android 的 shell 使用
<i>liblog</i>	与图 2-2 中所看到的 Android 内核日志记录器相连接的日志库；在树中贯穿使用

表 3-3：版本 2.3/ 姜饼中的 system/core/ 目录内容总结（续）

目录	内容
<i>libmncrypt</i>	基本的 RSA 和 SHA 函数，被恢复机制和 <i>mkbootimg</i> 功能所使用
<i>libnetutils</i>	网络配置库；被 <i>netd</i> 使用
<i>libpixelflinger</i>	低层次的图形渲染函数
<i>libsutils</i>	与系统各种不同组件包括框架通信的公用函数，被 <i>netd</i> 和 <i>vold</i> 使用
<i>libzipfile</i>	<i>zlib</i> 封装器以处理 ZIP 文件
<i>logcat</i>	<i>logcat</i> 工具
<i>logwrapper</i>	分派和运行传递给它的命令，并将标准输出和错误信息重新定向到 Android 的日志系统的工具
<i>mkbootimg</i>	用一个 RAM 磁盘和一个内核创建一个启动镜像的功能
<i>netcfg</i>	网络配置功能
<i>rootdir</i>	默认的 Android 根目录结构和内容
<i>run-as</i>	由一个指定的用户 ID 来运行程序的功能
<i>sdcard</i>	用 FUSE 仿真 FAT
<i>sh</i>	Android 的 shell
<i>toolbox</i>	Android 的工具箱 Toolbox（代替 BusyBox）

- a. 由于目前没有被任何部分的 AOSP 所使用，有一些目录已经被省略了。它们很可能成为遗留的组件。
- b. 这是用于创建默认的用于启动 Android 的 RAM 磁盘镜像和恢复镜像的。
- c. 除非你知道什么是 RenderScript，否则这个描述对你来说没有任何意义。你可以看一下 Google 文档中关于 RenderScript 的部分，那里有关 *libacc* 的内容会更清楚。

表 3-4：版本 2.3/ 姜饼和版本 4.2/ 果冻豆之间有关 system/core/ 目录的主要补充

目录	内容
<i>charger</i>	全屏电池状态显示
<i>fs_mgr</i>	文件系统管理器
<i>gpttool</i>	处理 GPT（UEFI）分区表的工具
<i>libcorkscrew</i>	调试（debugging）/ 回溯（backtrace）库
<i>libion</i>	与 ION 驱动连接的库
<i>libnl_2</i>	处理 NetLink 套接字的库
<i>libsuspend</i>	连接内核电源管理功能的库，包括自动休眠
<i>libsync</i>	连接目录 <i>/dev/sw_sync</i> 的库
<i>libusbhost</i>	USB 主机模式处理的库

除了 `core/` 目录以外，`system/` 目录里也包含了更多的目录，比如说 `netd/` 和 `vold/`，它们分别包含了 `netd` 和 `vold` 守护进程。

除了顶层的目录外，根目录也包含了一个单独的 `Makefile` 文件。然而，这个文件大部分是空的，它的主要用途是包含 Android 的编译系统的切入点：

```
### DO NOT EDIT THIS FILE ###
include build/core/main.mk
### DO NOT EDIT THIS FILE ###
```

你可能已经想到了，除了我刚刚跟你们提到的以外，还有很多关于 AOSP 的内容没有提到。毕竟，在版本 2.3/ 姜饼中有超过 14000 个目录和 100000 个文件，在版本 4.2/ 果冻豆中，有超过 40000 个目录和 265000 个文件。按照大多数的衡量标准来说，这是一个相当大的项目。相比之下，早期的 Linux 内核 3.0.x 发布版本有大约 2000 个目录和 35000 个文件。我们继续往后看的话，一定会有机会探讨更多关于 AOSP 的功能和源代码的部分。不过，我强烈建议你尽早地开始研究和实践这些源代码，因为可能会花几个月的时间，你才可以用自己的方式轻松地浏览它们。

## 构建的基础知识

现在，我们有了一个 AOSP 以及对它的内部结构有了一定的了解，让我们启动它让它运行起来吧。然而在我们构建它之前，还有最后一件重要的事情。我们必须确认我们的 Ubuntu 里安装了所有需要用到的包。这里有 64 位的 Ubuntu11.04 版本的说明，假设我们构建的是版本 2.3/ 姜饼。即使你用的是一个更老或者更新版本的一些基于 Debian 的 Linux 发布版本，这个说明也基本类似（可以看一下本章后面的“在虚拟机或者是非 Linux 系统上构建”，它介绍了可以构建 AOSP 的其他系统）。就像我之前提到过的，涉及为 Google 的最新版本的包初始化一个构建环境，需要在一个更新的 Ubuntu 版本上构建最新的 AOSP。

## 构建的系统设置

首先，让我们将一些基本的包安装在我们的开发系统中。可能这些包的其中一部分已经被你在做其他开发工作的时候安装好了，这非常好。Ubuntu 的包管理系统将会忽略你安装这部分包的请求。

---

**注意：**请注意，下面的命令为了适应这本书的宽度被分为了几行。在 shell 中每一行的结尾使用 \ 字符作为分隔符，表示要从下一行重新开始（也就是以 > 字符开始的部分），可以让你继续输入你的命令。这样说来，你可以在下面的命令的行尾输入 \ 字符，但是后一行开始位置的 > 字符不需要你输入，它是被 shell 自动插入的。本书的其他命令也使用同样的方法以便于给大家演示。

---

```
$ sudo apt-get install bison flex gperf git-core gnupg zip tofrodos \
> build-essential g++-multilib libc6-dev libc6-dev-i386 ia32-libs mingw32 \
> zlib1g-dev lib32z1-dev x11proto-core-dev libx11-dev \
> lib32readline5-dev libgl1-mesa-dev lib32ncurses5-dev
```

你可能还需要安装一些符号链接：

```
$ sudo ln -s /usr/lib32/libstdc++.so.6 /usr/lib32/libstdc++.so
$ sudo ln -s /usr/lib32/libz.so.1 /usr/lib32/libz.so
```

最后，你需要安装 Sun 的 JDK；“官方”不建议在 AOSP 构建中用 OpenJDK（可以看一下这个页面 (<https://groups.google.com/forum/?fromgroups=#!topic/android-building/IGCVGp9huLg>)，由 Google 的 Jean-Baptiste Queru 发布），虽然有些人能够成功地用它（看下面的侧边栏），但是 gcj 不行。在 Ubuntu 中，你可以通过使用下面的命令序列来获得 JDK：

```
$ sudo add-apt-repository "deb http://archive.canonical.com/ natty partner"
$ sudo apt-get update
$ sudo apt-get install sun-java6-jdk
```

不幸的是，Canonical (Ubuntu 幕后的公司) 和 Oracle 之间似乎有一些分歧，在写本书的时候这些指令不再能够正常运行。相反，你应该参考 Ubuntu 的指令，以获取 JDK6 来运行在你的主机中。请注意在写本书的时候，版本 7 还不能使用于 AOSP 中。从本质上来说，Ubuntu 的指令说明你需要从 Oracle 的网页上获取 JDK 二进制文件并安装。这里有一个对目前公布的指令稍微修改后的版本，它使得你可能不得不适应最新版本的 JDK：

```
$ chmod u+x jdk-6u38-linux-x64.bin
$ ./jdk-6u38-linux-x64.bin
$ sudo mkdir -p /usr/lib/jvm
$ sudo mv jdk1.6.0_38 /usr/lib/jvm/
$ sudo update-alternatives --install "/usr/bin/java" "java" \
> "/usr/lib/jvm/jdk1.6.0_38/bin/java" 1
$ sudo update-alternatives --install "/usr/bin/javac" "javac" \
> "/usr/lib/jvm/jdk1.6.0_38/bin/javac" 1
$ sudo update-alternatives --install "/usr/bin/javah" "javah" \
> "/usr/lib/jvm/jdk1.6.0_38/bin/javah" 1
$ sudo update-alternatives --install "/usr/bin/javadoc" "javadoc" \
```

```
> "/usr/lib/jvm/jdk1.6.0_38/bin/javadoc" 1  
$ sudo update-alternatives --install "/usr/bin/jar" "jar" \  
"/usr/lib/jvm/jdk1.6.0_38/bin/jar" 1
```

你之后将必须运行以下的命令，并选择你刚刚安装的版本：

```
$ sudo update-alternatives --config java  
There are 2 choices for the alternative java (providing /usr/bin/java).  
  
Selection Path Priority Status  
-----  
* 0      /usr/lib/jvm/java-6-openjdk-amd64/jre/bin/java 1061      auto mode  
  1      /usr/lib/jvm/java-6-openjdk-amd64/jre/bin/java 1061      manual mode  
  2      /usr/lib/jvm/jdk1.6.0_38/bin/java           1      manual mode  
  
Press enter to keep the current choice[*], or type selection number: 2  
$ sudo update-alternatives --display java  
java - manual mode  
link currently points to /usr/lib/jvm/jdk1.6.0_38/bin/java  
...  
$ sudo update-alternatives --config javac  
...  
$ sudo update-alternatives --config javah  
...  
$ sudo update-alternatives --config javadoc  
...  
$ sudo update-alternatives --config jar  
...  
...
```

就像你看到的，Oracle 的 JDK 和 OpenJDK 可以共存于同一个 Ubuntu 安装版本中。你只需要确定你所需要的指向正确 JDK 的默认指针。上面的指令指导你安装了 Oracle 的 JDK，并改变了默认的一些命令以便能够使用它的包中的二进制文件，而不是 Ubuntu 中的默认安装。没有人阻止你将 Oracle 的 JDK 安装在你的 home 目录下的某个位置，以及将 PATH 变量指向由运行中的 Oracle 安装的二进制文件提取的 bin/ 目录。

## 使用 OpenJDK 代替 Oracle 的 JDK

如果一直按照规则来做的话，有时候会比较无聊。尽管官方建议使用 Oracle 的 JDK，但是实际上还是有很多人成功地用 OpenJDK 来构建 AOSP。以下是 Linaro 的 Bernhard Rosenkranzer 所做的补丁，允许你用 OpenJDK 来构建 AOSP：

```
diff --git a/core/main.mk b/core/main.mk  
index 87488f4..32e3aec 100644  
--- a/core/main.mk  
+++ b/core/main.mk  
@@ -125,7 +125,14 @@ endif  
# Check for the correct version of java  
java_version := $(shell java -version 2>&1 | head -n 1 | grep '^java .*[  
"]1\.6[.\$"]')
```

```
ifeq ($(shell java -version 2>&1 | grep -i openjdk),)
-java_version :=
+$(warning ****)
+$(warning AOSP errors out when using OpenJDK, saying you need to use)
+$(warning Java SE 1.6 instead.)
+$(warning A build with OpenJDK seems to work fine though - if you)
+$(warning run into any Java errors, you may want to try using the)
+$(warning version required by AOSP though.)
+$(warning ****)
+#java_version :=
endif
ifeq ($(strip $(java_version)),)
$(info ****)
```

一些 Linaro 的工程师报告说他们不管是用这种方法编译 AOSP，还是运行产生的镜像文件都没有问题。其他一些人似乎报告了一些 javadoc 的问题，就像 Google 的 Jean-Baptiste Queru (<https://groups.google.com/forum/?fromgroups#!topic/android-building/IGCVGp9huLg>) 提示的那样。我们希望今后会有更多的人努力来为 OpenJDK 的可行性提供进一步的证明。

现在你的系统已经准备好构建 AOSP 了。显然，你不需要在每次构建 Android 的时候都进行包的安装。你只需要做一次就行了，用于你建立的每一个 Android 开发系统。

## 构建 Android

我们现在已经准备好构建 Android 了。现在进入到我们下载 Android 目录的位置，并配置构建系统：

```
$ cd ~/android/aosp-2.3.x
$ . build/envsetup.sh
including device/acme/coyotepad/vendorsetup.sh
including device/htc/passion/vendorsetup.sh
including device/samsung/crespo4g/vendorsetup.sh
including device/samsung/crespo/vendorsetup.sh
$ lunch
You're building on Linux
Lunch menu... pick a combo:
1. generic-eng
2. simulator
3. full_passion-userdebug
4. full_crespo4g-userdebug
5. full_crespo-userdebug
Which would you like? [generic-eng] ENTER
=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
```

```
TARGET_BUILD_TYPE=release  
TARGET_BUILD_APPS=  
TARGET_ARCH=arm  
HOST_ARCH=x86  
HOST_OS=linux  
HOST_BUILD_TYPE=release  
BUILD_ID=GINGERBREAD  
=====
```

版本 4.2/ 果冻豆中，在 Ubuntu12.04 上做同样的操作来替代：

```
$ cd ~/android/aosp-4.2  
$ . build/envsetup.sh  
including device/asus/grouper/vendorsetup.sh  
including device/asus/tilapia/vendorsetup.sh  
including device/generic/armv7-a-neon/vendorsetup.sh  
including device/generic/armv7-a/vendorsetup.sh  
including device/generic/mips/vendorsetup.sh  
including device/generic/x86/vendorsetup.sh  
including device/lge/mako/vendorsetup.sh  
including device/samsung/maguro/vendorsetup.sh  
including device/samsung/manta/vendorsetup.sh  
including device/samsung/toroplus/vendorsetup.sh  
including device/samsung/toro/vendorsetup.sh  
including device/ti/panda/vendorsetup.sh  
including sdk/bash_completion/adb.bash  
$ lunch
```

You're building on Linux

Lunch menu... pick a combo:

1. full-eng
2. full\_x86-eng
3. vbox\_x86-eng
4. full\_mips-eng
5. full\_grouper-userdebug
6. full\_tilapia-userdebug
7. mini\_armv7a\_neon-userdebug
8. mini\_armv7a-userdebug
9. mini\_mips-userdebug
10. mini\_x86-userdebug
11. full\_mako-userdebug
12. full\_maguro-userdebug
13. full\_manta-userdebug
14. full\_toroplus-userdebug
15. full\_toro-userdebug
16. full\_panda-userdebug

Which would you like? [full-eng] ENTER

```
=====  
PLATFORM_VERSION_CODENAME=REL  
PLATFORM_VERSION=4.2  
TARGET_PRODUCT=full  
TARGET_BUILD_VARIANT=eng  
TARGET_BUILD_TYPE=release
```

```
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-35-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JOP40C
OUT_DIR=out
=====
=====
```

以上两种情况下,请注意输入一个点号,一个空格符,然后输入 `build/envsetup.sh`。这是强制 shell 在当前的 shell 窗口中执行 `envsetup.sh` 脚本。如果我们想只运行脚本文件,那么 shell 会产生一个新的 shell 窗口,脚本文件就在这个新的 shell 窗口中运行。这样做无非是因为 `envsetup.sh` 定义了新的 shell 命令,比如说 `lunch`,并且设置之后构建所需要的环境。

我们之后会对 `envsetup.sh` 脚本文件和 `lunch` 命令做详细的解释。但是就目前而言,请注意版本 2.3/ 姜饼中的 `generic-eng` 选项和版本 4.2/ 果冻豆中的 `full-eng` 选项,它们意味着我们配置的构建系统是用于创建运行在 Android 仿真器上的镜像。类似 QEMU 模拟器软件,当在一个工作站上使用 SDK 开发时,这个软件被开发者用来测试他们的应用程序。在这里它被我们用来运行我们自定义的镜像,而不是运行 SDK 中装载的默认镜像。这里的仿真器和 Android 开发团队在没有设备的情况下用来开发 Android 的是同样的仿真器。因此,虽然它不是真正的硬件,因此达不到一个完美的目标,但是它对于我们要介绍的内容已经足够了。一旦你知道你的具体目标,你应该能够了解本书之后会出现的指令,而《构建嵌入式 Linux 系统》这本书对你或许会有一些帮助,它帮助你将自定义的 Android 镜像装载到你的设备上,用你的硬件来启动它们。

现在环境已经设置好了,我们可以构建 Android 了:

```
$ make -j16
=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
=====
```

```
Checking build tools versions...
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
find: `frameworks/base/frameworks/base/docs/html': No such file or directory
find: `out/target/common/docs/gen': No such file or directory
host Java: apicheck (out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes)
Header: out/host/linux-x86/obj/include/libexpat/expat.h
Header: out/host/linux-x86/obj/include/libexpat/expat_external.h
Header: out/target/product/generic/obj/include/libexpat/expat.h
Header: out/target/product/generic/obj/include/libexpat/expat_external.h
Header: out/host/linux-x86/obj/include/libpng/png.h
Header: out/host/linux-x86/obj/include/libpng/pngconf.h
Header: out/host/linux-x86/obj/include/libpng/pngusr.h
Header: out/target/product/generic/obj/include/libpng/png.h
Header: out/target/product/generic/obj/include/libpng/pngconf.h
Header: out/target/product/generic/obj/include/libpng/pngusr.h
Header: out/target/product/generic/obj/include/libwpa_client/wpa_ctrl.h
Header: out/target/product/generic/obj/include/libsonivox/eas_types.h
Header: out/target/product/generic/obj/include/libsonivox/eas.h
Header: out/target/product/generic/obj/include/libsonivox/eas_reverb.h
Header: out/target/product/generic/obj/include/libsonivox/jet.h
Header: out/target/product/generic/obj/include/libsonivox/ARM_synth_constants_gnu.inc
host Java: clearsilver (out/host/common/obj/JAVA_LIBRARIES/clearsilver_intermediates/classes)
target Java: core (out/target/common/obj/JAVA_LIBRARIES/core_intermediates/classes)
host Java: dx (out/host/common/obj/JAVA_LIBRARIES/dx_intermediates/classes)
Notice file: frameworks/base/libs/utils/NOTICE -- out/host/linux-x86/obj/NOTICE_FILES/src//lib/libutils.a.txt
Notice file: system/core/libcutils/NOTICE -- out/host/linux-x86/obj/NOTICE_FILES/src//lib/libcutils.a.txt
...

```

---

**警告：**请注意，特别是在末尾输出的那几行，它们都是因为超过了本书页面所允许的宽度而被顺延到了下一行。你可能还会看到几个由本书打印的限制出现的这种情况。我尽量保持一行 80 个字符，虽然有时候只要不是太明显我会多几个字。

总的来说，在本书之后的部分请你注意输出内容的换行。

---

现在正好是合适的时间去吃点小吃或者是看看今晚的曲棍球比赛——这是加拿大人的事，我就不能了。但值得注意的是，你的构建时间显然是取决于你的系统的性能。在一台配备四核英特尔酷睿 i7 处理器，且支持多处理器，8GB 的 RAM 的笔记本电脑上，这些命令如果是构建版本 2.3/ 姜饼需要耗时约 20 分钟，如果构建的是版本 4.2/ 果冻

豆则需要 80 分钟。在一台配备双核英特尔迅驰 2 处理器，2GB 的 RAM 的旧的笔记本电脑上，一个 `make -j4` 命令需要消耗 1 小时来构建版本 2.3/ 姜饼，我没有试过在这样一台笔记本电脑上构建版本 4.2/ 果冻豆。请注意 `make` 的 `-j` 参数允许你指定同时并行多少个作业。有人说，这个值最好为你的处理器数目的两倍，我在这里也是这样做的。另一种说法是，这个值最好是在你的处理器数上加上 2，我会用 10 和 4 来代替 16 和 4。

一般来说，AOSP 是软件构建中非常重要的一块。我强烈建议你使用你能够找到的配置最好的系统，毫无保留的。还有一个很重要的建议就是找一个 RAM 很大的系统。实际上，如果整个 AOSP 树都可以被内核装载在 RAM 的文件系统缓存中，那么这也会减少你的构建时间。你也可以用固态硬盘来代替普通的硬盘驱动，这些已经被证明是明显能够降低 AOSP 的构建时间的。

## 构建在虚拟机或者是非 Linux 系统上

经常有人问我关于在虚拟机上构建 AOSP 的问题，很多开发团队或者是他们的 IT 部门是以 Windows 为标准的。虽然我已经注意到了这项工作，且已经将镜像放在一起自己来做，但你得到的结果可能会有所不同。相较于在本地进行构建，在同一系统的 VM 上构建所需的时间通常会是两倍甚至是更多。所以如果你想要在 AOSP 上进行很多开发，我强烈建议你在本地进行构建。是的，这涉及是否有台 Linux 设备在手。

越来越多的开发者喜欢 Mac OS X 超过喜欢 Linux 和 Windows，包括很多在 Google 的人。因此，网页 <http://source.android.com> 上的官方说明中也描述了怎么在一台 Mac 上构建 AOSP。虽然这些指令会在 Mac OS 更新后被破坏掉，但是对于基于 Mac 开发的开发者来说幸运的是，他们人数很多，而且相当热心。因此，你会在网页上或者是各种各样的 Google 论坛中看到有关如何在你的新版本的 OS X 上构建 AOSP 的更新说明。这里有一个解释如何在 OS X Lion 上构建姜饼版本的帖子：在 OS X Lion 上构建姜饼版本 (<http://groups.google.com/group/android-building/msg/4b9e6168ecae68a5>)。然而请记住，就像我在第 1 章提到的那样，Google 自己的 Android 构建环境是基于 Ubuntu 系统的。如果你想要在 OS X 上构建，你可能就要一直追赶它的更新。最坏的情况下，你可以用一个 VM，就像在 Windows 下那样做。

如果你选择用 VM 这条路，请确保你配置的 VM 能够使用尽可能多的你系统中可用的 CPU。我见过的大多数 BIOS 默认禁止允许多 CPU 虚拟化的指令集设置。比如说 VirtualBox，当你尝试分配大于一个 CPU 给一个镜像，而指令集是不允许这样做时，它就会报一些模糊的错误出来。你必须进入到 BIOS 中，将有关你的 VM 软件的那些选项设置为可以允许你获得用户机 OS 的多 CPU。

关于构建还有几个问题需要考虑。首先请注意，在打印构建的配置信息和打印实际构建的第一次输出（打印输出为 host Java: apicheck (out/host/common/o...) 中，都会有一个相当长时间的延迟且什么都不打印出来，除非有“没有这样的文件或目录”这样的警告信息。我会在后面详细解释这种延迟，但是这足以说明，构建系统时正是需要这个时间来弄清楚如何构建 AOSP 每一个部分的规则。

还要注意的是，你可能会看到大量的警报声明。这是很“正常”的，虽然没有在维护软件质量方面出现的那么多，但是它们在 Android 的构建中是无处不在的。它们通常不会对编译产生的最终产品有影响。所以，与作为最好的软件工程师的直觉相反，我建议你完全不要理会这些警报，只要修改错误就行了。当然，除非这些警报是因为你自己添加的软件。无论如何，希望你摆脱这些警报。

## 运行 Android

当构建完成以后，你要做的事就是启动仿真器来运行你自己构建的镜像：

```
$ emulator &
```

这个命令将会启动仿真器窗口，然后启动如图 3-3 所示的完整的 Android 环境（显示的是版本 2.3/ 姜饼）。

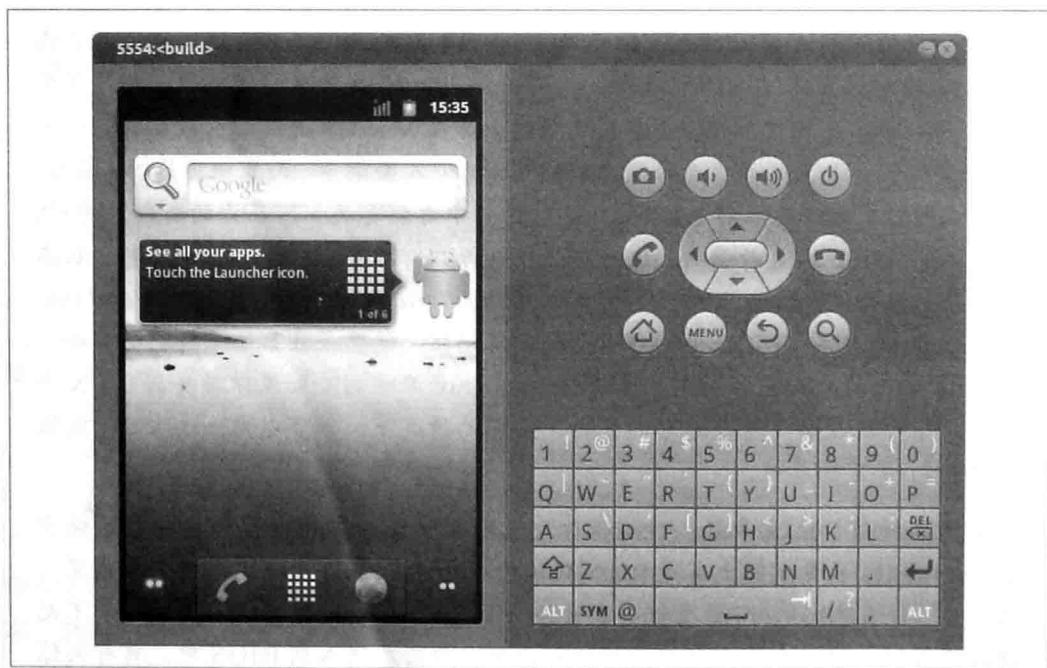


图 3-3：运行用户定制镜像的 Android 仿真器

之后，你就可以与你刚刚构建的 AOSP 进行交互了，就像它是运行在一台真正的设备上一样。当然，你的显示器可能不是一个触摸屏，所以你需要用你的鼠标来当作你的手指。鼠标的单击就表示手指触摸，按住鼠标键左右移动表示左右滑动屏幕，放开鼠标键表示你的手离开了触摸屏。你还有一个全键盘可以使用，所有的按键跟你在手机上看到的 QWERTY 键盘一样，不过你还是可以用普通的键盘输入内容到文本框中。

先不管它的特点以及实际的用途，Android 的仿真器还是有它的问题存在。一方面，它需要一定的时间来启动。它在第一次启动的时间消耗最长，这是因为 Dalvik 虚拟机要为手机上运行的应用程序创建一个 JIT 缓存。请注意 Dalvik 的缓存创建并不是专门针对于仿真器的。不管你的 Android 系统运行在什么类型的设备上，现代的 Dalvik 虚拟机都需要一个 JIT 缓存，无论它是在启动时创建还是像在第 7 章会看到的那样在构建时创建。

不过在第一次启动以后，你会发现你的仿真器非常笨重，特别是你在修改→编译→测试这个循环中时更明显。此外，仿真器并不能完美地模拟一切。比如说，通常当使用 F11 或者 F12 旋转屏幕是非常艰难的。不过这主要是应用程序开发者面临的问题。

如果由于某些原因，你关闭了你已经配置好、编译好的 shell，并开始启动 Android，或者如果你需要启动一个新的 shell 且要访问所有的从构建中创建的工具和二进制文件时，你必须再次调用 *envsetup.sh* 脚本文件和 *lunch* 命令来设置环境变量。比如说以下就是一个新的 shell 中输入的命令：

```
$ cd ~/android/aosp-2.3.x
$ emulator &
No command 'emulator' found, did you mean:
Command 'qemu emulator' from package 'qemu' (universe)
emulator: command not found
$ . build/envsetup.sh
$ lunch
```

You're building on Linux

```
Lunch menu... pick a combo:
1. generic-eng
2. simulator
3. full_passion-userdebug
4. full_crespo4g-userdebug
5. full_crespo-userdebug
```

Which would you like? [generic-eng] ENTER

```
=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
```

```
TARGET_BUILD_VARIANT=eng  
...  
=====  
$ emulator &  
$
```

请注意，我们第二次运行 *emulator* 命令时，shell 不再警告这个命令找不到了。这同样适用于其他的 Android 工具，例如我们要介绍的 *adb* 命令。还要注意的是，我们并没有执行任何的 *make* 命令，这是因为已经构建好 Android 了。在这种情况下，我们只需要保证环境变量设置正确，使得以前构建的结果可以让我们再次使用。

## 使用 Android 调试工具（ADB）

Android 开发团队设计的开发环境还有一个最有趣的地方是，你可以通过 USB 连接，或者说是 *adb* 工具，用 shell 进入到正在运行的仿真器中，或是任何真实的设备中：

```
$ adb shell ❶  
* daemon not running. starting it now on port 5037 *  
* daemon started successfully *  
# cat /proc/cpuinfo ❷  
Processor : ARM926EJ-S rev 5 (v5l)  
BogoMIPS : 405.50  
Features : swp half thumb fastmult vfp edsp java  
CPU implementer : 0x41  
CPU architecture: 5TEJ  
CPU variant : 0x0  
CPU part : 0x926  
CPU revision : 5  
  
Hardware : Goldfish  
Revision : 0000  
Serial : 0000000000000000
```

- ❶ 当你启动仿真器时，这个命令同样要运行在 shell 中。
- ❷ 这是目标机的 shell，*cat* 命令实际上运行在“目标机”（即仿真器）上。

正如你所看到的那样，模拟器中运行的内核据说是一个 ARM 处理器，实际上这个处理器是 Android 最常用的平台。而且，这个内核运行在一个叫做 Goldfish（金鱼）的平台上。这是一个模拟器的代码名称，你有可能看到它的地方很少。

现在，你可以在模拟器上看到一个 shell 命令行解释器，你可以 root 进去，这在模拟器中是默认的，之后如果你 shell 到了一个远程机器上，或者是一个传统的网络连接的嵌入式 Linux 系统上时，就可以运行任何你想要运行的命令。Android 调试工具（ADB）使之成为可能。如果想要退出 ADB shell 会话，你要做的就是按下组合键 Ctrl-D：

```
# [CTRL-D] ①  
$ ②
```

① 这是在目标 shell 中做的操作。

② 这是回到了主机中。

当你在主机上第一次打开 *adb* 的时候，它在后台启动一个服务程序，这个程序的工作主要是管理连接到主机上的所有 Android 设备连接。这就是较早的所谓的从端口 5073 启动的守护进程输出的一部分。事实上你还可以询问这个守护进程它检测到了什么设备：

```
$ adb devices  
List of devices attached  
emulator-5554 device  
0000021459584822 device  
emulator-5556 offline
```

这个输出表示的是一个模拟器实例正在运行，一台设备正通过 USB 连接，以及另一个模拟器实例正在启动。如果有多台设备连接，你可以通过 *-s* 标志来选择你要操作的是哪一台设备，这个标志用来识别设备的序列号。

```
$ adb -s 0000021459584822 shell  
$ id  
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input), ...  
$ su  
su: permission denied
```

在这种情况下请注意，我在我的 shell 里得到的是一个 \$ 符号，而不是一个 #。这就意味着与之前的互动情况不同，我没有用 root 的身份登录，也同样能够从输出中看到命令的 *id* 号。其实，这是一个真正的商用 Android 手机，我无法通过上面的操作用 *su* 命令来获得 root 权限。因此，我对于这个设备进行修改的权限是非常有限的。当然，除非我找到一些其他的方法来获得这个手机的 root 权限（比如说获得 root 访问）。

一直以来，设备制造商由于各种不同的原因都很不愿意将它们设备的 root 权限开放给用户，并且即使知道是不可能，还是为此制定了很多条款以使之变得尽可能的困难。这就是为什么很多有能力的用户和黑客将“root”设备当作就像是举起一座奖杯一样。截至 2013 年年初，一些制造商，包括摩托罗拉，HTC 以及索尼手机，已经清楚地向我们表明政策发生了变化，其似乎是旨在使得用户可以更方便地获得他们设备的 root 权限，当然也是要在有警告的情况下，但是这还并不是主流。而且不幸的是，它不得不受限于网络运营商，因为某些运营商希望仍然可以决定是否锁定这些被手机制造商解锁的设备。

---

**警告：**你可能忍不住想要 root 一个商业手机或者设备来试验 Android 平台的开发。我会建议你认真考虑一下。虽然表面上有很多说明书解释了怎样将你的标准镜像用通常称为“自定义 ROM”的工具来替换，比如 CyanogenMod 或者其他，你必须注意的是，操作期间任何错误的步骤都有可能使你的设备变为“砖头”（例如你的设备无法开机或者是被删除了关键的启动代码）。那么你将会拥有一台昂贵的压纸器（因此称之为“砖头”）而不是一台手机。

如果你想要在真实的硬件上实验运行你自定义的 AOSP，我建议你找一些比如 BeagleBoard xM 开发板或者是熊猫板（PandaBoard）的工具。对这些开发板进行一些改进。如果不出意外，它们没有一个让你有可能会损坏的内置的闪存芯片。相反，这些设备上的片上系统（SoC）直接从 SD 卡上启动。因此，修复损坏的镜像只需要简单地将 SD 卡从板子上拔下来，连接到你的工作计算机上，对它重新编程，再重新插回板子上就可以了。

一些商业手机或者设备允许你“解锁”固件，通常通过 *fastboot oem unlock* 命令来完成，这样你就能够烧录自定义的镜像到你的设备上，而你的设备变为“砖头”的风险也减小了。尽管如此，这种情况下的引导装载程序（bootloader）会成为单点故障，如果由于某种原因你将它损坏了，你的设备最后也将变为“砖头”。最好的配置是有一个你能够重新编程所有存储设备的地方，这样不管你输入了什么命令都没有关系了。

---

当然，*adb* 可以做的不仅仅是给你一个 shell 编辑器，我建议你不带任何参数地打开一个 shell，看一下它的用法输出：

```
$ adb
Android Debug Bridge version 1.0.26

-d           - directs command to the only connected USB device
              returns an error if more than one USB device is
              present.
-e           - directs command to the only running emulator.
              returns an error if more than one emulator is
              running.
-s <serial number> - directs command to the USB device or emulator with
                     the given serial number. Overrides ANDROID_SERIAL

...
device commands:
adb push <local> <remote>      - copy file/dir to device
adb pull <remote> [<local>]       - copy file/dir from device
adb sync [<directory> ]          - copy host->device only if changed
                                         (-l means list but don't copy)
                                         (see 'adb help all')
adb shell                         - run remote shell interactively
adb shell <command>                - run remote shell command
adb emu <command>                 - run emulator console command
...
```

例如，你可以用 *adb* 导出记录在主日志缓冲区内的数据：

```
$ adb logcat
I/DEBUG   ( 30): debuggerd: Sep 10 2011 13:44:19
I/Netd    ( 29): Netd 1.0 starting
I/Vold    ( 28): Vold 2.1 (the revenge) firing up
D/qemud   ( 38): entering main loop
D/Vold    ( 28): USB mass storage support is not enabled in the kernel
D/Vold    ( 28): usb_configuration switch is not enabled in the kernel
D/Vold    ( 28): Volume sdcard state changing -1 (Initializing) -> 0 (No-Media)
D/qemud   ( 38): fdhandler_accept_event: accepting on fd 9
D/qemud   ( 38): created client 0xe078 listening on fd 10
D/qemud   ( 38): client_fd_receive: attempting registration for service 'boot-properties'
D/qemud   ( 38): client_fd_receive: -> received channel id 1
D/qemud   ( 38): client_registration: registration succeeded for client 1
I/qemu-props( 54): connected to 'boot-properties' qemud service.
I/qemu-props( 54): receiving..
I/qemu-props( 54): received: qemu.sf.lcd_density=160
I/qemu-props( 54): receiving..
I/qemu-props( 54): received: dalvik.vm.heapsize=16m
I/qemu-props( 54): receiving..
D/qemud   ( 38): fdhandler_event: disconnect on fd 10
I/qemu-props( 54): exiting (2 properties set).
D/AndroidRuntime( 32):
D/AndroidRuntime( 32): >>>>> AndroidRuntime START com.android.internal.os.ZygoteInit <<<<<
D/AndroidRuntime( 32): CheckJNI is ON
I/        ( 33): ServiceManager: 0xad50
...
...
```

这对于观察关键系统组件运行时的行为非常有帮助，包括由系统服务器运行的服务。

你也可以复制文件到设备中或者是从设备中复制文件出来：

```
$ adb push data.txt /data/local
1 KB/s (87 bytes in 0.043s)
$ adb pull /proc/config.gz
95 KB/s (7087 bytes in 0.072s)
```

再次说明，鉴于 *adb* 在 Android 开发的核心地位，我推荐读者研读 *adb* 的使用。我们将在整本书中使用它，并且会在第 6 章做更为详细的介绍。虽然如此，但还是请记住，*adb* 也有它的奇特的地方。首先，很多使用者发现他们主机端的守护进程有一点奇怪。由于某种原因，它有时候不能正确地识别链接到它的设备的状态，即使你重新尝试链接设备也还是继续显示设备是离线的。或者 *adb* 只是在命令行中挂起等待设备，直到设备是完全激活的状态并且能够接收 *adb* 命令。这些问题的解决方法几乎总是杀死主机端的守护进程<sup>注2</sup>：

```
$ adb kill-server
```

---

注 2：实际上有点意思是，Android 开发团队感到有必要建立这样的功能来帮助正常地进入 *adb*。显然，他们遇到了守护进程本身的问题。

不用担心，当你下一次发出任何 *adb* 命令的时候，守护进程就会自动重启。目前还不清楚是什么原因导致了这种行为，也许这个问题在将来的某个时候会得到解决。在此期间请记住，当你在使用 ADB 的时候如果看到一些怪异的行为，杀死主机端的守护进程通常是你在研究解决其他潜在问题之前要尝试的解决方法。

正如我前面所说的那样，我们将在第 6 章详细讨论 ADB。不过，另一个关于 *adb* 的信息来源是 Google 的 Android 开发指南中的 Android Debug Bridge 部分。正如 Tim Bird<sup>注3</sup> 所提到的，你需要打印一个副本并把它放在你的枕头底下。

## 掌握模拟器的使用

正如我之前所说的，在平台开发中，简单地使用模拟器可以使你在开发的路上走很远。它用最少的硬件设备有效地模拟了一个 ARM 目标板，以及近似一个 x86 目标板。在这里我们会花一些时间深入了解模拟器先进的方面。和 Android 很多其他部分一样，模拟器的里里外外也是一个相当复杂的软件。不过，我们可以通过观察一些关键特性来深入了解它的性能。

在开始的时候我们先通过简单的输入以下的命令来启动模拟器：

```
$ emulator &
```

但是模拟器命令也可以有很多参数。你可以通过在命令后添加 *-help* 标志从而在命令行中看到在线帮助：

```
$ emulator -help
Android Emulator usage: emulator [options] [-qemu args]
  options:
    -sysdir <dir>          search for system disk images in <dir>
    -system <file>           read initial system image from <file>
    -datadir <dir>           write user data into <dir>
    -kernel <file>           use specific emulated kernel
    -ramdisk <file>          ramdisk image (default <system>/ramdisk.img)
    -image <file>            obsolete, use -system <file> instead
    -init-data <file>         initial data image (default <system>/
    -initdata <file>          same as '-init-data <file>'
    -data <file>              data image (default <datadir>/userdataqemu.img
    -partition-size <size>   system/data partition size in MBs
...
...
```

其中一个特别有用的标志是 *-kernel*。它可以允许你告诉模拟器使用另一个内核，而不是存放在文件夹 *prebuilt/android-arm/kernel/* 里默认的那个：

---

注 3：Tim 是 <http://elinux.org> 的维护者，就是那个在嵌入式 Linux 会议后面的家伙，他是 Linux 基金会 CE 工作组的主席，曾经在 Sony 做了很多很酷的东西。

```
$ emulator -kernel path_to_your_kernel_image/zImage
```

比如，如果你想使用一个有模块支持的内核，你需要自己构建一个，因为默认的内核不支持模块。另外，在默认情况下，模拟器不会显示给你内核的启动信息。然而你可以通过输入 `-show-kernel` 标志来查看它们：

```
$ emulator -show-kernel
Uncompressing Linux.....done, booting the kernel.
Initializing cgroup subsys cpu
Linux version 2.6.29-00261-g0097074-dirty (digit@digit.mtv.corp.google.com) (gcc
version 4.4.0 (GCC) ) #20 Wed Mar 31 09:54:02 PDT 2010
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00093177
CPU: VIVT data cache, VIVT instruction cache
Machine: Goldfish
Memory policy: ECC disabled, Data cache writeback
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 24384
Kernel command line: qemu=1 console=ttyS0 android.checkjni=1 android.qemud=ttyS1
    android.ndns=3
Unknown boot option `android.checkjni=1': ignoring
Unknown boot option `android.qemud=ttyS1': ignoring
Unknown boot option `android.ndns=3': ignoring
PID hash table entries: 512 (order: 9, 2048 bytes)
Console: colour dummy device 80x30
Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
Memory: 96MB = 96MB total
Memory: 91548KB available (2616K code, 681K data, 104K init)
Calibrating delay loop... 403.04 BogoMIPS (lpj=2015232)
Mount-cache hash table entries: 512
Initializing cgroup subsys debug
Initializing cgroup subsys cpacct
Initializing cgroup subsys freezer
CPU: Testing write buffer coherency: ok
...
```

同样，你也可以在模拟器中输入 `-verbose` 标志来打印出你自己的执行信息，从而你能够看到比如说它使用的是哪一个镜像之类的信息：

```
$ emulator -verbose
emulator: found Android build root: /home/karim/android/aosp-2.3.x
emulator: found Android build out: /home/karim/android/aosp-2.3.x/out/target/pr
oduct/generic
emulator: locking user data image at /home/karim/android/aosp-2.3.x/out/targ
et/product/generic/userdata-qemu.img
emulator: selecting default skin name 'HVGA'
emulator: found skin-specific hardware.ini: /home/karim/android/aosp-2.3.x/sdk/e
mulator/skins/HVGA/hardware.ini
emulator: autoconfig: -skin HVGA
emulator: autoconfig: -skindir /home/karim/android/aosp-2.3.x/sdk/emulator/skins
emulator: keyset loaded from: /home/karim/.android/default.keyset
emulator: trying to load skin file '/home/karim/android/aosp-2.3.x/sdk/emulator/
skins/HVGA/layout'
```

```
emulator: skin network speed: 'full'
emulator: skin network delay: 'none'
emulator: no SD Card image at '/home/karim/android/aosp-2.3.x/out/target/product/generic/sdcard.img'
emulator: registered 'boot-properties' qemud service
emulator: registered 'boot-properties' qemud service
emulator: Adding boot property: 'qemu.sf.lcd_density' = '160'
emulator: Adding boot property: 'dalvik.vm.heapsize' = '16m'
emulator: argv[00] = "emulator"
emulator: argv[01] = "-kernel"
emulator: argv[02] = "/home/karim/android/aosp-2.3.x/prebuilt/android-arm/kernel/kernel-qemu"
emulator: argv[03] = "-initrd"
emulator: argv[04] = "/home/karim/android/aosp-2.3.x/out/target/product/generic/ramdisk.img"
emulator: argv[05] = "-nand"
emulator: argv[06] = "system,size=0x4200000,initfile=/home/karim/android/aosp-2.3.x/out/target/product/generic/system.img"
emulator: argv[07] = "-nand"
emulator: argv[08] = "userdata,size=0x4200000,file=/home/karim/android/aosp-2.3.x/out/target/product/generic/userdata-qemu.img"
emulator: argv[09] = "-nand"
...
...
```

到目前为止，我已经可以相互交换的使用 QEMU 和模拟器了。在现实中，虽然模拟器命令不是真正的 QEMU：它是一个被 Android 开发团队创建的自定义的包装所包围的。但是，你仍然可以通过 -qemu 标志来与模拟器的 QEMU 交互。你在输入这个标志之后的所有输入信息都是在 QEMU 中进行的，而不是在模拟器中：

```
$ emulator -qemu -h
QEMU PC emulator version 0.10.50Android, Copyright (c) 2003-2008 Fabrice Bellard
usage: qemu [options] [disk_image]

'disk_image' is a raw hard image image for IDE hard disk 0

Standard options:
-h or -help      display this help and exit
-version        display version information and exit
-M machine      select emulated machine (-M ? for list)
-cpu cpu        select CPU (-cpu ? for list)
-smp n          set the number of CPUs to 'n' [default=1]
-numa node[,mem=size][,cpus=cpu[-cpu]][,nodeid=node]
-fda/-fdb file  use 'file' as floppy disk 0/1 image
-hda/-hdb file  use 'file' as IDE hard disk 0/1 image
...
$ emulator -qemu -...
```

前面了解了如何用 *adb* 与运行在模拟器内的 AOSP 交互，我们刚刚也看到了怎么通过不同的选项来改变模拟器的启动方式。有趣的是，我们还可以通过远程登录来实时监

控模拟器的行为。每一个启动的模拟器实例都在主机上被分配了一个端口号。大家再看一下图 3-3，检查一下模拟器窗口的左上角。在这个位置上有一个数字（在这个例子中是 5554），这就是模拟器实例正在监听的端口号。下一个同时运行的模拟器将会分配到 5556 的端口号，再下一个 5558 等。想要访问模拟器的专用控制台，你可以使用常规的 *telnet* 命令：

```
$ telnet localhost 5554
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^].
Android Console: type 'help' for a list of commands
OK
help
Android console command help:

help|h|?      print a list of commands
event          simulate hardware events
geo            Geo-location commands
gsm            GSM related commands
kill            kill the emulator instance
network        manage network settings
power          power related commands
quit|exit      quit control session
redir          manage port redirections
sms            SMS related commands
avd             manager virtual device state
window         manage emulator window

try 'help <command>' for command-specific help
OK
```

使用这个控制台，你可以做一些特别有技巧性的处理，比如说重定向一个从主机到目标机的端口号：

```
redir add tcp:8080:80
OK
redir list
tcp:8080 => 80
OK
```

从这以后，任何对你主机上 8080 端口的访问，实际上都将与那台 Android 模拟器上的所有对端口 80 的监听进行对话。在 Android 中默认对该端口没有监听，但是你可以，比如说让 BusyBox 的 *httpd* 运行在 Android 上并通过这种方式连接上它。

在模拟 Android 时模拟器也暴露了一些“神奇的”IP。比如说 IP 地址 10.0.2.2，就是你的工作站 127.0.0.1 的别名。如果你的工作站中正在运行 Apache，你可以打开模拟器的浏览器，输入 <http://10.0.2.2>，就可以浏览到 Apache 上提供的所有内容。

如果你想要知道更多关于如何操作模拟器的信息，你可以看看 Google 的 Android 开发团队指南（Android Developers Guide）中的“使用 Android 模拟器”这一章的内容。这是为应用程序开发者写的一个指南，不过即使你正在做的是平台工作，这个指南同样对你非常有帮助。

# 构建系统

前一章的目的是为了让你能够运行并且尽可能快地运行自定义的 AOSP。在现在这个时候，没有任何事情阻止你合上这本书，去深入挖掘并修改你的 AOSP 树以适应你的需求。想要测试你所做的修改，你所需要做的事仅仅是重新编译一下你的 AOSP，再次启动模拟器，如果需要的话，你可以用 ADB 重新 shell 回到修改的地方。然而，如果你想让你的努力发挥最大的效果，你可能会想要了解一些 Android 构建系统的知识。

尽管 Android 的编译系统是模块化的，但它还是相当复杂并且与它之外的任何主流的构建系统都不像，至少这些主流构建系统中没有一个是开源的项目。特别是，它的 *make* 命令的使用采取了一种非常规的方式，并且不提供任何基于 *menuconfig* 配置的排序（或者是其他等价的方式）。Android 中有很多它自己的构建模式，需要花一些时间去适应它。所以抓住其中的一两个要点，事情变得严肃起来。

---

**警告：**就像 AOSP 其余的部分一样，构建系统是一个变化的系统。所以尽管以下的信息在很长一段时间内是有效的，但是你还是要找出你所用的 AOSP 版本做了哪些改变。

---

## 与其他构建系统的比较

在我开始解释 Android 的构建系统如何工作之前，请允许我首先强调它与你已经知道的其他构建系统之间有什么不同。首先，不同于很多基于 *make* 的构建系统，Android 的构建系统不依赖于递归的 *makefile* 文件。比如，与 Linux 内核不同，没有一个顶层的 *makefile* 文件来递归地调用子目录的 *makefile* 文件。取而代之的是，它有一个脚本

来搜索所有的目录和子目录，直到找到一个 *Android.mk* 文件，之后会停止搜索，不再搜索这个文件位置之下的子目录，除非这个 *Android.mk* 文件发现还有构建系统其他的指令。请注意，Android 不依赖 *makefile*。相反，它依赖的是 *Android.mk* 文件，这个文件指定了本地的“模块”是怎么构建的。

Android 构建的“模块”与内核“模块”没有任何关系。在 Android 构建系统的环境中，“模块”就是指任何需要构建的 AOSP 的组件。它可能是一个二进制文件，可能是一个 app 包，一个库等，它可能是为主机或者目标机构构建的，但是它对于构建系统来说仍然是一个“模块”。

## 有多少构建模块？

可以思考一下 AOSP 可以构建多少个模块，你可以在你的 AOSP 树中试着运行一下以下命令：

```
$ find . -name Android.mk | wc -l
```

这个命令会查找所有的 *Android.mk* 文件，并且计算一共有多少个。在版本 2.3.7/ 姜饼中一共有 1143 个，在版本 4.2/ 果冻豆中有 2037 个。

另外一个 Android 特别的地方是它的构建系统的配置方式。然而，我们中的大多数人都习惯使用基于内核风格带有菜单配置或者 GNU 自动工具（即自动配置 *autoconf*，自动构造 *automake* 等）的系统，Android 依赖于一组变量，它们或者是通过 *envsetup.sh* 和 *lunch* 的方式被动态设置为 shell 环境的一部分，或者是在一个 *buildspec.mk* 文件中被提前静态的定义好。另外，对于初学者来说似乎会有一个惊喜，Android 的编译系统的配置水平是相当有限的。因此，尽管你可以为想要创建的 AOSP 这个目标指定属性，并且在一定程度上，它的应用程序应该被默认地包含在 AOSP 结果中，所以你没有办法去启用或者是禁用大部分的属性，就像是一个点菜菜单配置。比如，你不能决定你不想用电源管理支持，或者是你不想用默认启动的定位服务。

此外，构建系统不会生成目标文件或者是像源文件一样的在相同位置上有任何形式的中间输出。比如说你不会在源代码树中的 *.c* 源文件旁边找到 *.o* 文件。事实上，没有一个现有 AOSP 目录使用任何的输出。相反，构建系统创建一个 *out/* 目录，这个目录用于存储它所产生的所有东西。因此，一个 *make clean* 命令跟一个 *rm -rf out/* 命令产生相同的效果。换句话说，移除了 *out/* 目录就是删除掉所有这个目录所创建的内容。

在我们开始探索有关构建系统更详细的内容之前，最后要说的关于构建系统的是，它严重依赖使用 *GNU make* 命令版本。而且，现在是 3.81 版本，即使在较新的 3.82 版本中，它跟许多 AOSP 版本也不兼容，除非打了补丁。构建系统，其实在很大程度上依赖于使用 *GNU make* 命令特有的功能，如 `define`、`include` 和 `ifndef` 指令。

## Android 构建系统设计中的一些背景

如果你想获得更多关于如何把 Android 的编译系统放在一起的设计选择方面的想法，你可以查看 AOSP 中的 `build/core/build-system.html` 文件。这可以追溯到 2006 年 5 月，似乎已经绕到了 Android 开发团队中获得对构建系统的返工共识文件。有些信息和假设是过时的或已过时的，但目前大多数构建系统的亮点都在那里。我发现了更深远的事情，就是文件由 Android 的开发团队创建，更有见地的是关于原始的动机和技术背景。较新的文件往往会被“清理”和抽象，如果它们存在的话。

如果你想了解为什么 Android 的编译系统不使用递归的 *make* 的技术基础，看看题为“Recursive Make Considered Harmful”这篇文章，这是由 AUUG 公司出版的 AUUG 杂志中彼得·米勒编写的。这篇文章围绕如何使用递归 `makefile` 文件的问题进行了探讨，并解释了不同的方法，其中涉及使用一个单一的全局的 `makefile` 文件来构建基于模块 `.mk` 文件的项目，而这正好与 Android 是一样的。

## 体系结构

如图 4-1 所示，使构建系统更有意义的关键点是在 `build/core/` 目录中的 `main.mk` 文件中，这个文件通过顶层 `Makefile` 调用，正如我们之前所看到的一样。`build/core/` 目录实际上包含了构建系统的大部分，我们将从密钥文件开始介绍。请再次记住，Android 的构建系统将所有的一切整合成一个单一的 `makefile` 文件，而且它不是递归的。因此，你最终会看到每个 `.mk` 文件成为一个巨大的 `makefile` 文件的一部分，它包含用于构建系统中所有的规则。

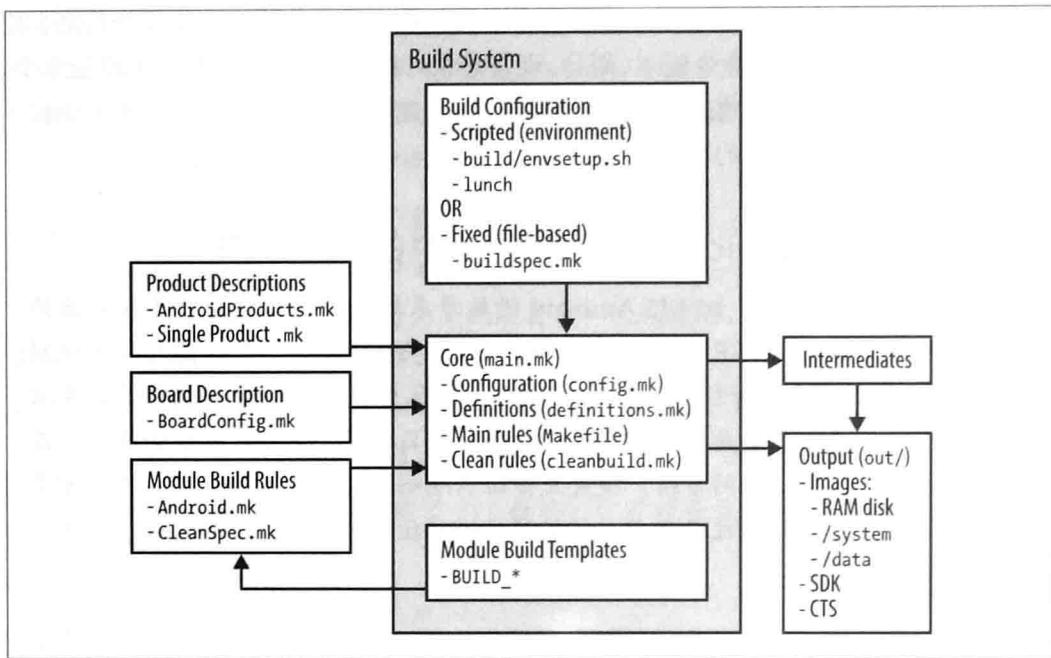


图 4-1: Android 的构建系统

## 为什么挂起 make 命令

任何时候你输入 *make* 命令，你就见证了 MK 文件聚集成一个单一的组合，这看起来像什么恼人的构建人工制品一样：构建系统打印出生成配置，似乎挂起了一段时间，但在屏幕上不打印任何东西。屏幕静默这些长的时间后，它居然又开始启动，建立 AOSP 的各部分，此时在屏幕上就是你所看到的，你期望从任何正常的构建系统中看到的常规输出。任何构建 AOSP 的人一直想知道世界上的构建系统在这段时间做的是什么事情。它所做的事情就是整合每一个你能够在 AOSP 中找到的 *Android.mk* 文件。

如果你想看到这个动作，你可以编辑 *build/core/main.mk* 文件并替换这行：

```
include $(subdir_makefiles)
```

更改为：

```

$(foreach subdir_makefile, $(subdir_makefiles), \
$(info Including $(subdir_makefile)) \
$(eval include $(subdir_makefile)) \
)
subdir_makefile :=
```

下一次你输入 *make* 的时候，你就会真正的看到发生了什么：

```
$ make -j16
=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
...
=====
Including ./bionic/Android.mk
Including ./development/samples/Snake/Android.mk
Including ./libcore/Android.mk
Including ./external/elfutils/Android.mk
Including ./packages/apps/Camera/Android.mk
Including ./device/htc/passion-common/Android.mk
...
```

## 配置

其中构建系统做的第一件事就是把 *config.mk* 文件包含的内容拉到构建配置里。构建既可以通过使用 *envsetup.sh* 和 *lunch* 命令，也可以通过在顶层目录中提供一个 *buildspec.mk* 文件来进行配置。在任何一种情况下，以下的一些变量都需要进行设置：

### TARGET\_PRODUCT

要构建的 Android 的风格。每个配方都可以，比如，包括一组不同应用程序或区域设置或构建树。看看各种各样的单独产生的 *.mk* 文件，比如说在 2.3/Gingerbread 版本的系统中，它包含在 *build/target/product*, *device/samsung/crespo*/ 目录，以及 *device/htc/passion/* 目录的 *AndroidProducts.mk* 文件中。而在 4.2/Jelly Bean 版本的系统中，可以找一下 *device/asus/grouper/* 目录和 *device/samsung/amgnuro/* 目录，而不是 crespo 和 passion。它的值包括以下内容：

#### generic

属于“香草”这一种类，它是你可以拥有的最基础的 AOSP 构建类型。

#### full

属于“光鲜亮丽”这一类，大多数应用程序和主要的语言环境都支持。

#### full\_crespo

跟 full 一样，但是主要针对 crespo（三星的 Nexus S）。

#### full\_grouper

跟 full 一样，但是主要针对 Grouper（华硕的 Nexus 7）。

#### sim

Android 的模拟器（请参阅 “The Simulator: A Piece of Android’s History”）

这本书）。但是这是适用于 2.3/ 姜饼这个版本的，这个目标在 4.2/Jelly Bean 这个版本中已经被删除了。

### sdk

就是 SDK，包含大量的本地设置。

### TARGET\_BUILD\_VARIANT

选择哪些模块来进行安装。每一个模块都应该在其 *Android.mk* 里有一个 LOCAL\_MODULE\_TAGS 变量设置，以支持以下的至少一个<sup>注1</sup>：用户 (user)，调试 (debug)，工程 (eng)，测试 (tests)，可选 (optional) 或样品 (samples)。通过选择这些变量，你会告诉编译系统应该包含哪些模块子集，但唯一的例外是包（例如，模块生成的 .apk 文件），这些规则不适用于包。特别的是：

#### eng

包括所有模块标记，比如说 user，debug 或者 eng。

#### userdebug

包括的模块标记是 user 和 debug。

#### user

包括的模块标记只有 user。

### TARGET\_BUILD\_TYPE

用来决定特别的构建标志是否应该在代码里被使用或者 DEBUG 变量是否要在代码中定义。这里的值可能是 release 或 debug。最值得注意的是，*frameworks/base/Android.mk* 文件是根据这个变量是否被设置为 debug 来在文件夹 *frameworks/base/core/config/debug* 或者文件夹 *frameworks/base/core/config/ndebug* 中进行选择的。前者导致 *ConfigBuildFlags.DEBUG* 的 Java 常量被设置为 true，而后者导致它被设置为 false。比如说，系统服务部分的一些代码，在 DEBUG 下是有条件的。通常，TARGET\_BUILD\_TYPE 都会设置为 release。

### TARGET\_TOOLS\_PREFIX

默认情况下，编译系统将使用附带在 *prebuilt/* 目录下的交叉开发工具链，在版本 4.2/Jelly Bean 中是 *prebuilts/* 目录。不过，如果你想让它使用另一种工具链，你可以设置一个值，使其指向这个工具链的位置。

---

注 1：如果不提供一个值，将使用默认值。举例来说，所有的应用程序都默认设置为可选 (optional)。此外，一些模块在 *user\_tags.mk* 文件中是 GRANDFATHERED\_USER\_MODULES 的一部分。LOCAL\_MODULE\_TAGS 没有必要为它们指定值，它们总是包括在内的。

## `OUT_DIR`

默认情况下，编译系统将会把所有构建的输出放到 `out/` 目录下。你可以使用这个变量来提供备用输出目录。

## `BUILD_ENV_SEQUENCE_NUMBER`

如果使用 `built/buildspec.mk.default` 里的模板来创建自己的 `buildspec.mk` 文件，这个值将被正确设置。但是，如果用一个比较老的 AOSP 的发布版本来创建一个 `buildspec.mk` 文件，并尝试把它使用在包含重要改变的新的 AOSP 发布版本的构建系统中，会有不同的值，这个变量将作为一个安全网。导致构建系统会通知你，你的 `buildspec.mk` 文件与构建系统不匹配。

## 模拟器：Android 的一部分历史

如果你回到了目录中第 3 章的“构建 Android”这一节的有关于“2.3/ 姜饼的 lunch”部分，你会发现有一个条目叫做模拟器（simulator）。事实上，你会发现对模拟器的一些引用出现在版本 2.3/ 姜饼的很多地方，包括很多的 `Android.mk` 文件和子目录树里。你需要了解有关模拟器的最重要的是，它与仿真器没有任何关系。它们是两个完全不同的事情。

这就是说，模拟器似乎是 Android 团队在创造 Android 的时候残留下来的早期作品。因为在当时甚至没有将 Android 在 QEMU 上运行，他们用自己的桌面操作系统和 `LD_PRELOAD` 机制来模拟一个 Android 设备，因此称为“模拟器”。很明显后来一旦 Android 在 QEMU 上运行成为可能，他们停止了使用模拟器。它继续在 AOSP 中使用，直到版本 4.0/ 冰淇淋三明治，虽然，这是潜在的用在 AOSP 的构建部分，用于开发和测试的开发人员工作站。比如说，在版本 4.2/ 果冻豆中，就不再有模拟器了。

在版本 2.3/ 姜饼中，模拟器构建目标的存在和历史并不意味着你可以在你的桌面上运行 AOSP。事实上，你不能这样做，如果仅仅是因为你需要一个包含 Binder 的内核，你需要一直使用 Bionic，而不是系统缺省的 C 库。但是，如果你想在桌面上运行一个从 AOSP 中构建出来的部分，这款产品的目标产物允许你这样做。

在版本 2.3/ 姜饼中，如果目标是模拟器的话，不同部分代码的构建非常不同。比如说，在浏览代码时，你有时会发现 `HAVE_ANDROID_OS` C 宏定义的条件编译，这是在模拟器编译时才会定义的。与 Binder 对话的部分代码就是其中之一。如果 `HAVE_ANDROID_OS` 没有定义，这部分代码就会返回一个错误给调用者，而不是继续尝试与 Binder 的驱动程序进行对话。

如果你想要了解模拟器背后完整的故事，你可以看一看 Android 开发者 Andrew McFadden 的对于题为“Android 的模拟器环境”一文的回应，刊登在 2009 年 4 月的 android 移植邮件列表中。

除了要选择构建 AOSP 的哪一部分，以及构建它们需要选择什么样的选项以外，构建系统还需要了解它所构建的目标。这可以通过一个 *BoardConfig.mk* 文件来提供信息，这个文件将指定类似于要提供给内核什么样的命令行，内核应该加载的基地址是什么，或者是最适合该主板的 CPU（*TARGET\_ARCH\_VARIANT*）的指令集版本是什么等信息。你可以看一下 *build/target/board/* 目录里的一组目标中每一个目标的目录，它们都包含一个 *BoardConfig.mk* 文件。你也可以看看包含在 AOSP 里的各种 *device/\*/TARGET\_DEVICE/BoardConfig.mk* 文件。后者比前者要丰富得多，这是因为后者包含了更多的特定于硬件的信息。设备名称（即 *TARGET\_DEVICE*）来源于 *product.mk* 文件中指定的 *PRODUCT\_DEVICE*，这个文件由配置里的 *TARGET\_PRODUCT* 决定。比如说在版本 2.3/ 姜饼中，*device/samsung/crespo/AndroidProducts.mk* 文件包含 *device/samsung/crespo/full\_crespo.mk* 文件，这个文件将 *PRODUCT\_DEVICE* 设置为 *crespo*。因此，构建系统会在 *device/\*/crespo/* 文件夹中寻找 *BoardConfig.mk* 文件，在这个位置恰好是相同的。同样地在版本 4.2/ 果冻豆中，文件 *device/asus/grouper/full\_grouper.mk* 中的 *PRODUCT\_DEVICE* 被设置为 *grouper*，从而将构建系统指向 *device/\*/grouper/BoardConfig.mk* 文件。

最后一个要说明的问题就是，配置用于构建 Android 的 CPU 特定的选项。对于 ARM 来说，这些都包含在 *build/core/combo/arch/arm/armv\*.mk* 文件中，由 *TARGET\_ARCH\_VARIANT* 来确定实际要使用的是什么文件。每个文件都列出了特定于 CPU 的交叉编译器和用于构建 C/C++ 文件的交叉链接器。它们还包含了一些 *ARCH\_ARM\_HAVE\_\** 变量，使得 AOSP 的其他部分能够根据一个给定的 ARM 特征是否可以在目标 CPU 中找到来有条件的构建代码。

## envsetup.sh

之前已经介绍了各种输入到构建系统需要的配置，接着我们可以讨论关于 *envsetup.sh* 的作用的更多细节了。正如其名称所暗示的，*envsetup.sh* 在 Android 中实际上是用来启动构建环境的，虽然它只是做了其中的一部分工作。更主要的是，它定义了一系列的 shell 命令，它们对任何形式的 AOSP 运行都有用：

```
$ cd ~/android/aosp-2.3.x
$ . build/envsetup.sh
$ help
Invoke ". build/envsetup.sh" from your shell to add the following functions to
your environment:
- croot:    Changes directory to the top of the tree.
- m:        Makes from the top of the tree.
- mm:       Builds all of the modules in the current directory.
- mmm:      Builds all of the modules in the supplied directories.
- cgrep:    Greps on all local C/C++ files.
- jgrep:    Greps on all local Java files.
- resgrep: Greps on all local res/*.xml files.
```

- godir: Go to the directory containing a file.

Look at the source to view more functions. The complete list is:

```
add_lunch_combo cgrep check_product check_variant choosecombo chooseproduct choo
setype choosevariant cproj croot findmakefile gdbclient get_abs_build_var getbug
reports get_build_var getprebuilt gettop godir help isviewserverstarted jgrep lu
nch m mm mmm pgrep pid printconfig print_lunch_menu resgrep runhat runtest set_j
ava_home setpaths set_sequence_number set_stuff_for_environment settitle smokete
st startviewserver stopviewserver systemstack tapas tracedmdump
```

在版本 4.2/ 果冻豆中, *hmm* 替代了 *help*, 而且你可以使用的命令组合也得到了扩展:

```
$ cd ~/android/aosp-4.2
$ . build/envsetup.sh
$ hmm
Invoke ". build/envsetup.sh" from your shell to add the following functions to y
our environment:
- lunch:   lunch <product_name>-<build_variant>
- tapas:   tapas [<App1> <App2> ...] [arm|x86|mips] [eng|userdebug|user]
- croot:   Changes directory to the top of the tree.
- m:       Makes from the top of the tree.
- mm:      Builds all of the modules in the current directory.
- mmm:     Builds all of the modules in the supplied directories.
- cgrep:   Greps on all local C/C++ files.
- jgrep:   Greps on all local Java files.
- resgrep: Greps on all local res/*.xml files.
- godir:   Go to the directory containing a file.
```

Look at the source to view more functions. The complete list is:

```
addcompletions add_lunch_combo cgrep check_product check_variant choosecombo cho
oseproduct choosetype choosevariant cproj croot findmakefile gdbclient get_abs_b
uild_var getbugreports get_build_var getlastscreenshot getprebuilt getscreenshot
path getsdcardpath gettargetarch gettop godir hmm isviewserverstarted jgrep key_
back key_home key_menu lunch _lunch m mm mmm pid printconfig print_lunch_menu re
sgrep runhat runtest set_java_home setpaths set_sequence_number set_stuff_for_en
vironment settitle smoketest startviewserver stopviewserver systemstack tapas tr
acedmdump
```

你很可能会发现 *croot* 和 *godir* 命令在遍历树的时候非常有用。该命令的某些部分是相当有深度的, 它可以使用 Java 且要求是, 存放在目录树里的包必须命名为相应的完全合格的包名称, 该名称的每个子部分与层次结构名称相同。例如, `com.foo.bar` 包的文件部分必须存放在 `com/foo/bar/` 目录下。因此, 在你自己的 AOSP 的顶级目录下找到 7 到 10 层的目录不是很难, 并且在输入类似 `cd.. ../../...` 命令来返回到目录树的最上层的时候它很快变得乏味起来。

*m* 和 *mm* 也非常有用, 因为它们分别允许你从顶层开始构建模块, 无论你当前在什么目录下, 或只是构建当前目录中的模块。例如, 如果你对 *Launcher* 做了一个修改, 但当前在 *packages/apps/Launcher2* 目录下, 可以通过输入 *mm* 重新构建那个模块, 而

不用再输入 `cd` 回到顶层目录，再输入 `make` 来实现。请注意，`mm` 并不是重新构建整个目录树，因此，不会再重新生成 AOSP 镜像，即使相关模块发生了变化。但是 `m` 可以做到这一点。不过，`mm` 可以用来测试你的本地修改是否破坏构建，直到你准备重新生成完整的 AOSP。

在线帮助中没有提到 `lunch`，它是由 `envsetup.sh` 定义的命令之一。当运行 `lunch` 时不带任何参数，它就会提示你一个潜在参数的列表。这是在版本 2.3/ 姜饼中提示的列表：

```
$ lunch  
You're building on Linux  
  
Lunch menu... pick a combo:  
1. generic-eng  
2. simulator  
3. full_passion-userdebug  
4. full_crespo4g-userdebug  
5. full_crespo-userdebug  
  
Which would you like? [generic-eng]
```

以下是在版本 4.2/ 果冻豆中提示的列表：

```
$ lunch  
You're building on Linux  
  
Lunch menu... pick a combo:  
1. full-eng  
2. full_x86-eng  
3. vbox_x86-eng  
4. full_mips-eng  
5. full_grouper-userdebug  
6. full_tilapia-userdebug  
7. mini_armv7a_neon-userdebug  
8. mini_armv7a-userdebug  
9. mini_mips-userdebug  
10. mini_x86-userdebug  
11. full_mako-userdebug  
12. full_maguro-userdebug  
13. full_manta-userdebug  
14. full_toroplus-userdebug  
15. full_toro-userdebug  
16. full_panda-userdebug  
  
Which would you like? [full-eng]
```

这些选择不是一成不变的。它们大多数是根据在 `envsetup.sh` 运行时所在 AOSP 中的内容来确定的。事实上脚本中定义的他们是通过使用 `add_lunch_combo()` 函数来单

独添加的。比如说，在版本 2.3/ 姜饼中，*envsetup.sh* 默认的增加了 `generic-eng` 和 `simulator`：

```
# add the default one here
add_lunch_combo generic-eng
# if we're on linux, add the simulator. There is a special case
# in lunch to deal with the simulator
if [ "$(uname)" = "Linux" ] ; then
    add_lunch_combo simulator
fi
```

在版本 4.2/ 果冻豆中，`simulator` 不再是一个有效的目标，而 *envsetup.sh* 将它取代如下：

```
# add the default one here
add_lunch_combo full-eng
add_lunch_combo full_x86-eng
add_lunch_combo vbox_x86-eng
add_lunch_combo full_mips-eng
```

*envsetup.sh* 还包括所有它可以找到的供应商提供的脚本。以下演示了在版本 2.3/ 姜饼中它是如何运行的：

```
# Execute the contents of any vendorsetup.sh files we can find.
for f in `/bin/ls vendor/*/vendorsetup.sh vendor/*/*/build/vendorsetup.sh device/*
/*/vendorsetup.sh 2> /dev/null`
do
    echo "including $f"
    . $f
done
unset f
```

而以下是它在版本 4.2/ 果冻豆中的运行情况：

```
# Execute the contents of any vendorsetup.sh files we can find.
for f in `/bin/ls vendor/*/vendorsetup.sh vendor/*/*/*/vendorsetup.sh device/*/*/*/
endorsetup.sh 2> /dev/null`
do
    echo "including $f"
    . $f
done
unset f
```

在版本 2.3/ 姜饼中的 `device/samsung/crespo/vendorsetup.sh` 文件，比如说是这样的：

```
add_lunch_combo full_crespo-userdebug
```

同样的，在版本 4.2/ 果冻豆中则是这样的：

```
add_lunch_combo full_grouper-userdebug
```

下面告诉你如何结束我们前面看到的菜单。请注意，菜单会要求你选择一个端口。本质上来说，这是 TARGET\_PRODUCT 和 TARGET\_BUILD\_VARIANT 的组合，除了版本 2.3/ 姜饼中的模拟器以外。该菜单提供了默认的组合，但其他的仍然有效，并可以在命令行中作为参数传递给 *lunch* 命令。比如说在版本 2.3/ 姜饼中，你可以这样做：

```
$ lunch generic-user
```

```
=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=user
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
=====
```

```
$ lunch full_crespo-eng
```

```
=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=full_crespo
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
=====
```

一旦 *lunch* 与 generic-eng 组合运行结束，它将在当前的 shell 中设置表 4-1 中所述的环境变量，来为构建系统提供所需的配置信息。

表 4-1：通过 *lunch* 设置的环境变量，用于版本 2.3/ 姜饼中默认的构建目标（即 generic-eng）设置（没有特定的顺序）

变量	值
PATH	\$ANDROID_JAVA_TOOLCHAIN:\$PATH:\$ANDROID_BUILD_PATHS
ANDROID_EABI_TOOLCHAIN	aosp-root/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin

表 4-1：通过 lunch 设置的环境变量，用于版本 2.3/ 姜饼中默认的构建目标（即 generic-eng）设置（没有特定的顺序）（续）

变量	值
ANDROID_TOOLCHAIN	\$ANDROID_EABI_TOOLCHAIN
ANDROID_QTOOLS	<i>aosp-root/development/emulator/qtools</i>
ANDROID_BUILD_PATHS	<i>aosp-root/out/host/linux-x86:\$ANDROID_TOOLCHAIN:\$ANDROID_QTOOLS:\$ANDROID_TOOLCHAIN:\$ANDROID_EABI_TOOLCHAIN</i>
ANDROID_BUILD_TOP	<i>aosp-root</i>
ANDROID_JAVA_TOOLCHAIN	\$JAVA_HOME/bin
ANDROID_PRODUCT_OUT	<i>aosp-root/out/target/product/generic</i>
OUT	ANDROID_PRODUCT_OUT
BUILD_ENV_SEQUENCE_NUMBER	10
OPROFILE_EVENTS_DIR	<i>aosp-root/prebuilt/linux-x86/oprofile</i>
TARGET_BUILD_TYPE	release
TARGET_PRODUCT	generic
TARGET_BUILD_VARIANT	eng
TARGET_BUILD_APPS	empty
TARGET_SIMULATOR	false
PROMPT_COMMAND	\\"\\033]0;[\${TARGET_PRODUCT}- \${TARGET_BUILD_VARIANT}] \$ {USER}@\${HOSTNAME}: \${PWD}\\007\\"
JAVA_HOME	/usr/lib/jvm/java-6-sun

## 缓存的使用

如果你在读这本书的时候，已经做过一些构建 AOSP 的事情，那么可能你已经注意到了这个过程是多么漫长。显然，除非你可以构造一个自定义的很前沿的构建环境，否则对当前的硬件任何形式的加速都将增值。Android 开发团队自己可能也感受到了构建时间如此之长所带来的痛苦，所以他们已经增加了对 *ccache* 的支持。*ccache* 代表的是 *Compiler Cache*，它是 Samba 项目的一部分。这是一种机制，是由基于该预处理器的输出编译器生成的目标文件形成的高速缓存。因此，如果两个独立的构建预处理器处理的输出是相同的，那么使用 *ccache* 将会产生第二次构建，这次构建实际上没有使用编译器来生成文件。相反，缓存的对象文件将被复制到目的地，编译器的输出即是如此。

要启用 `ccache`, 你需要做的就是在你开始构建之前确保 `USE_CCACHE` 环境变量被设置为 1:

```
$ export USE_CCACHE=1
```

在你首次运行时, 不会得到任何加速, 这是因为缓存在那个时候是空的。不过, 从这之后的所有从头开始的构建, 缓存都将有助于加速整个构建的过程。唯一的缺点是, `ccache` 仅仅是针对 C/C++ 文件的。因此, 它不能加速任何 Java 文件的编译, 这一点我必须遗憾的补充一下。在版本 2.3/ 姜饼的 AOSP 中, 大约有 15000 的 C/C++ 文件和 18000 的 Java 文件。而在版本 4.2/ 果冻豆中, 这些数字是 27000 和 29000。因此, 虽然缓存不是万能的, 但它还是聊胜于无。

如果想了解更多关于 `ccache` 的知识, 你可以看看一篇刊登在 IBM 的 developerWorks 网站上的, 由马丁·布朗编写的名为 “*Improve collaborative build times with ccache*” (用 `ccache` 来协助提高构建时间) 的文章。文章还探讨了 `distcc` 的使用, 它允许将构建工作分布在几台机器上, 这样你就可以将你团队的工作站缓存集中起来。

尽管 `ccache` 带来诸多好处, 但是一些开发商使用 `ccache` 的时候, 在某些情况下会产生奇怪的错误。举例来说, 在维护我自己的 AOSP 框架时, 就遇到过这样的问题。首先, 从我的工作站中得到了一个版本的 AOSP, 并构建它, 创建了一个不错的缓存。之后我上传构建树到 <http://github.com> 上。最后, 我在刚刚上传的树中做了一个 `repo sync` (往回同步) 的操作, 但是在我的工作站中得到的是另一个目录, 而不是原来上传的那一个目录。使用 `diff` 命令来比较两个目录树, 结果显示这两个目录树是相同的。然而, 原来的构建在缓存中是好的, 而第二次构建的版本仍然继续构建失败, 直到缓存被擦除。

当然, 如果你厌倦了总是输入 `build/envsetup.sh` 和 `lunch`, 你需要做的事就是复制 `build/buildspec.mk.default` 文件到顶层目录中去, 重命名为 `buildspec.mk`, 并编辑它使它的内容能够与那些通过运行这些命令而得到的配置相匹配。该文件已经包含了所有你需要提供的变量, 这只是一个取消相应行的注释并设置相应值的问题。一旦你这样做了, 你所要做的就是到 AOSP 所在的目录, 并直接调用 `make` 命令。这样你就可以跳过 `envsetup.sh` 和 `lunch` 了。

## 函数定义

因为构建系统是相当庞大的——它仅仅在 *build/core* 目录下就有超过 40 个的 *.mk* 文件——能够尽可能多的重复使用代码是最好的。这就是为什么构建系统在 *definitions.mk* 文件中定义了大量的函数。该文件在构建系统中实际上是最大的一个文件，大小约 60KB 左右，其中在版本 2.3/ 姜饼中约有 140 个函数，约 1800 行生成文件代码。它在版本 4.2/ 果冻豆中仍然是构建系统中最大的文件，大小约 73KB，有 170 个函数，以及约 2100 行的生成文件代码。函数提供了多种操作，包括文件的查找（例如，`all-makefiles-under` 和 `all-C-files-under`），转换（例如，`transform-c-to-o` 和 `transform-java-to-classes.jar`），复制（例如，`copy-file-to-target`），以及示例程序（例如，`my-dir`）。

这些函数的使用不仅作为核心库贯穿了整个构建系统其余的组件，而且它们有时也直接用在模块的 *Android.mk* 文件中。下面是一个实例片断，它来自计算器应用程序的 *Android.mk* 文件：

```
LOCAL_SRC_FILES := $(call all-jar-files-under, src)
```

虽然详细的描述 *definitions.mk* 文件超出了本书的范围，但如果你自己去研究，对你来说应该是一件很容易的事情。如果不出意外，它的大部分函数前面都加了一个注释，用来说明它们的功能是什么。下面是版本 2.3/ 姜饼中的一个例子：

```
#####
## Find all of the java files under the named directories.
## Meant to be used like:
## SRC_FILES := $(call all-jar-files-under,src tests)
#####

define all-jar-files-under
$(patsubst .%,%,\n    $(shell cd $(LOCAL_PATH) ; \n        find $(1) -name "*.java" -and -not -name ".*") \
)
endef
```

## 主 make 方法

在这个时候，你可能会想知道所有生成的东西都在哪里。各种各样的镜像，比如说 RAM 磁盘是如何产生的，或者 SDK 是如何集成在一起的，举例说明一下？好吧，我希望你不会介意，因为我把最好的东西留在了最后。那么事不宜迟，我们来看看 *build/core/* 目录下的 *Makefile* 文件（不是顶层目录的那一个）。这个文件以一个看似平淡无奇的注释开始：

```
# Put some miscellaneous rules here
```

但是，不要被愚弄，这就是一些精华所在。下面是生成 RAM 磁盘的一些片段，比如说，在版本 2.3/ 姜饼中是这样的：

```
# -----
# the ramdisk
INTERNAL_RAMDISK_FILES := $(filter $(TARGET_ROOT_OUT)%, \
$(ALL_PREBUILT) \
$(ALL_COPIED_HEADERS) \
$(ALL_GENERATED_SOURCES) \
$(ALL_DEFAULT_INSTALLED_MODULES))

BUILT_RAMDISK_TARGET := $(PRODUCT_OUT)/ramdisk.img

# We just build this directly to the install location.
INSTALLED_RAMDISK_TARGET := $(BUILT_RAMDISK_TARGET)
$(INSTALLED_RAMDISK_TARGET): $(MKBOOTFS) $(INTERNAL_RAMDISK_FILES) | $(MINIGZIP)
$(call pretty,"Target ram disk: $@")
$(hide) $(MKBOOTFS) $(TARGET_ROOT_OUT) | $(MINIGZIP) > $@
```

而下面的是测试无线（OTA）更新的生成包的一些生成片段，跟前面的是在同一个 AOSP 版本下：

```
# -----
# Build a keystore with the authorized keys in it, used to verify the
# authenticity of downloaded OTA packages.
#
# This rule adds to ALL_DEFAULT_INSTALLED_MODULES, so it needs to come
# before the rules that use that variable to build the image.
ALL_DEFAULT_INSTALLED_MODULES += $(TARGET_OUT_ETC)/security/otacerts.zip
$(TARGET_OUT_ETC)/security/otacerts.zip: KEY_CERT_PAIR :=
$(DEFAULT_KEY_CERT_PAIR)
$(TARGET_OUT_ETC)/security/otacerts.zip: $(addsuffix .x509.pem,
$(DEFAULT_KEY_CERT_PAIR))
$(hide) rm -f $@
$(hide) mkdir -p $(dir $@)
$(hide) zip -qj $@ $<

.PHONY: otacerts
otacerts: $(TARGET_OUT_ETC)/security/otacerts.zip
```

当然，除了我之前说的之外，还有很多内容可以在这里介绍，但你可以看看 *Makefile* 文件，了解下面的所有操作是如何创建的：

- 属性（包括目标系统的 */default.prop* 和 */system/build.prop*）。
- RAM 磁盘。
- 启动镜像（结合 RAM 磁盘和一个内核镜像）。

- *NOTICE* 文件：有一些是 AOSP 的 Apache 软件许可证 (ASL) 的需要使用的文件。想要了解更多 ASL 相关的详细信息，你可以看一看 *NOTICE* 文件。
- OTA 关键存储。
- 恢复镜像。
- 系统镜像（目标系统的 */system* 目录）。
- 数据分区镜像（目标系统的 */data* 目录）。
- OTA 更新包。
- SDK。

然而，有一些东西没有在这个文件中：

### 内核镜像

不要试图找任何规则来构建它。官方 AOSP 版本中没有内核部分，有的第三方项目会在附录 E 中列出来，然而，他们实际上只是将内核源代码直接封装到它们构建的 AOSP 中。你需要找到一个 Android 版本的内核来作为你的目标，在 AOSP 中单独构建它，并将结果输出到 AOSP。你可以在 *device/* 目录中找到设备的这几个例子。例如，在版本 2.3/ 姜饼中，目录 *device/samsung/crespo/* 中包括一个内核映像（称为内核文件），以及一个用于 Crespo 的 WiFi (*bcm4329.ko* 文件) 的可加载模块。这两个都是在 AOSP 之外构建的，并以二进制的形式复制到包含构建其余部分的树中。

### NDK

构建 NDK 的代码在 AOSP 中，它完全从目录 *build/* 下的 AOSP 的构建系统中独立出来。相反，NDK 的构建系统是在 *ndk/build/* 目录下的。我们将在不久之后简单的讨论如何建立 NDK。

### CTS

构建 CTS 的规则是在 *build/core/tasks/ct.mk* 中。

## 清除

正如我前面提到的那样，*make clean* 目标等价于清除 *out/* 目录下的东西。*clean* 这个目标本身是定义在 *main.mk* 文件中的。然而，还有其他的用作清理的目标也被定义了。最值得注意的是 *installclean*，它在 *cleanbuild.mk* 文件中定义，每当你修改 *TARGET\_PRODUCT*、*TARGET\_BUILD\_VARIANT* 或 *PRODUCT\_LOCALES* 时它会自动被调用。举例来说，如果我先构建版本 2.3/ 姜饼的 *generic-eng* 组合，然后用 *lunch* 命令来将组合切换

到 full-eng，那么当我下次运行 *make* 命令时，一些编译的输出将会被自动调用的 *installclean* 删除：

```
$ make -j16
=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
...
=====
*** Build configuration changed: "generic-eng-{mdpi,nodpi}" -> "full-eng-{en_US,
en_GB,fr_FR,it_IT,de_DE,es_ES,mdpi,nodpi}"
*** Forcing "make installclean"...
*** rm -rf out/target/product/generic/data/* out/target/product/generic/data-qem
u/* out/target/product/generic/userdata-qemu.img out/host/linux-x86/obj/NOTICE_F
ILES out/host/linux-x86/sdk out/target/product/generic/*.img out/target/product/
generic/*.txt out/target/product/generic/*.xlb out/target/product/generic/*.zip
out/target/product/generic/data out/target/product/generic/obj/APPS out/target/p
roduct/generic/obj/NOTICE_FILES out/target/product/generic/obj/PACKAGING out/tar
get/product/generic/recovery out/target/product/generic/root out/target/product/
generic/system out/target/product/generic/dex_bootjars out/target/product/generi
c/obj/JAVA_LIBRARIES
*** Done with the cleaning, now starting the real build.
```

跟 *clean* 命令相比，*installclean* 并不会彻底删除整个 *out/* 目录的内容。相反，它只会删除一些需要重新构建的带来组合配置变更的部分。还有一个 *clobber* 目标，本质上跟 *clean* 一样。

## 模块化构模板

刚才所描述的主要是构建系统的体系结构及其核心部件的机制。看了上面的内容，你应该更了解 Android 是如何从自上而下的角度构建起来的。但是我们之前很少渗透到 AOSP 模块的 *Android.mk* 文件的层级。事实上该系统已经被构建，以至于模块的构建方法更多地独立于构建系统的内核。相反，模块的模板已经提供给大家，让模块作者们能够更恰当的构建他们的模块。每一个模板都是为特定类型的模块定做的，并且模块作者可以使用一组记录的变量，这些变量都是带有 *LOCAL\_* 前缀的，来调整模板的行为和输出。当然，模板及相关支持文件（主要是 *base\_rules.mk* 文件）与构建系统的其余部分紧密合作，来正确地处理每个模块的构建输出。但是，这对于模块的作者来说是无形的。

这些模板本身与构建系统其余的部分一样，都在同一位置即 *build/core/* 目录下。*Android.mk* 文件通过 *include* 指令来访问它们。这里有一个例子：

```
include $(BUILD_PACKAGE)
```

正如你所看到的，*Android.mk* 文件从名字上来看实际上并不包括*.mk* 模板。相反，它包括被设置为相应的*.mk* 文件的变量。表 4-2 提供了可选模块模板的完整列表。

表 4-2：模块构建模板列表

变量	模版	构建的是什么	最突出的应用
BUILD_EXECUTABLE	<i>executable.mk</i>	目标二进制文件	本地命令和守护进程
BUILD_HOST_EXECUTABLE	<i>host_executable.mk</i>	主二进制文件	开发工具
BUILD_RAW_EXECUTABLE	<i>raw_executable.mk</i>	在裸机上运行的目标二进制文件	<i>bootloader/</i> 目录下的代码
BUILD_JAVA_LIBRARY	<i>java_library.mk</i>	目标 Java 文件	Apache Harmony 项目和 Android 框架
BUILD_STATIC_JAVA_LIBRARY	<i>static_java_library.mk</i>	目标静态 Java 库	N/A，一些用这个的模块
BUILD_HOST_JAVA_LIBRARY	<i>host_java_library.mk</i>	主 Java 库	开发工具
BUILD_SHARED_LIBRARY	<i>shared_library.mk</i>	目标共享库	大量的模块，包括很多 <i>external/</i> 目录和 <i>frameworks/base/</i> 目录下的模块
BUILD_STATIC_LIBRARY	<i>static_library.mk</i>	目标静态库	大量的模块，包括很多 <i>external/</i> 目录下的模块
BUILD_HOST_SHARED_LIBRARY	<i>host_shared_library.mk</i>	主共享库	开发工具
BUILD_HOST_STATIC_LIBRARY	<i>host_static_library.mk</i>	主静态库	开发工具
BUILD_RAW_STATIC_LIBRARY	<i>raw_static_library.mk</i>	在裸机上运行的目标静态库	<i>bootloader/</i> 目录下的代码
BUILD_PREBUILT	<i>prebuilt.mk</i>	预编译的目标文件副本	配置文件和二进制文件
BUILD_HOST_PREBUILT	<i>host_prebuilt.mk</i>	预编译的主文件副本	<i>prebuilt/</i> 目录下的工具和配置文件
BUILD_MULTI_PREBUILT	<i>multi_prebuilt.mk</i>	多样化但知道类型的预编译模块的副本，例如 Java 库或者是可执行文件	很少使用

表 4-2：模块构建模版列表（续）

变量	模版	构建的是什么	最突出的应用
BUILD_PACKAGE	<i>package.mk</i>	AOSP 中构建的应用程序（即以 .apk 结尾的任何东西）	AOSP 中的所有应用程序
BUILD_KEY_CHAR_MAP	<i>key_char_map.mk</i>	设备字符映射表	AOSP 中所有的字符映射表

这些编译模板通常使得 *Android.mk* 文件变得无足轻重：

```
LOCAL_PATH := $(call my-dir) ①
include $(CLEAR_VARS) ②

LOCAL_VARIABLE_1 := value_1 ③
LOCAL_VARIABLE_2 := value_2

...
include $(BUILD_MODULE_TYPE) ④
```

- ① 告诉编译模板当前模块所在的位置。
- ② 清除所有先前设置的可能已设置为其他模块的 LOCAL\_\* 变量。
- ③ 设置各种 LOCAL\_\* 变量模块特定的值。
- ④ 调用对应于当前模块类型的构建模板。

---

**注意：**注意，*clear\_vars.mk*<sup>注2</sup> 文件里定义的 CLEAR\_VARS 非常重要。我们知道，构建系统包含了一个巨大的 *makefile*，它由所有的 *Android.mk* 文件形成。其中 CLEAR\_VARS 变量确保在你的模块包含的 *Android.mk* 文件之前设置 LOCAL\_\* 值。另外，一个 *Android.mk* 可以逐个描述多个模块。因此，CLEAR\_VARS 变量确保前一个模块定义不影响后面的。

---

这里以 2.3/ 姜饼系统 Service Manager 的 *Android.mk* 文件为例 (*frameworks/base/cmds/servicemanager/*)<sup>注3</sup>：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := service_manager.c binder.c
```

---

注 2：这个文件包括了一系列以 LOCAL\_ 开头的变量，CLEAR\_VAR 不会清除该文件中没有定义的变量。

注 3：这个版本做了一点清理（清除了一些注释，做了一点格式调整）。

---

```
LOCAL_MODULE := servicemanager
ifeq ($(BOARD_USE_LVMX),true)
    LOCAL_CFLAGS += -DLVMX
endif

include $(BUILD_EXECUTABLE)
```

这里还有一个 2.3/ 姜饼系统中 Desk Clock 应用的例子 (*packages/app/DesktopClock/*)<sup>注4</sup>:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-java-files-under, src)
LOCAL_PACKAGE_NAME := DeskClock
LOCAL_OVERRIDES_PACKAGES := AlarmClock
LOCAL_SDK_VERSION := current

include $(BUILD_PACKAGE)

include $(call all-makefiles-under,$(LOCAL_PATH))
```

可以看到，即使两个文件指定了不一样的输入，产生不一样的输出，却使用了几乎一模一样的结构。同样也可以注意到，Desk Clock 的 *Android.mk* 文件的最后一行，包含了所有子目录的 *Android.mk* 文件。前面已经说过，构建系统查找目录下的第一个 makefile 并执行它，但不会继续查找其子目录，所以需要手工加入。很明显，这里最后一行就是寻找所有子目录下的 makefile。不过，AOSP 中的某些部分也有明确列出其子目录或者根据配置条件选择其子目录的做法。

之前的 <http://source.android.com> 上的文档提供了所有 *LOCAL\_\** 变量的含义和用途的详尽清单。不幸的是，在编写本书的时候，这个清单不再可用。而 *build/core/build-system.html* 文件，包含了这个清单的早期版本。在最新的清单再次出现之前，可以先参考这个文件。下面是一些最常见的 *LOCAL\_\** 变量：

#### LOCAL\_PATH

当前模块的路径，通常是调用 `$(call my-dir)` 得到的。

#### LOCAL\_MODULE

模块构建输出的名称。实际的文件名、输出以及位置取决于你所包含的构建模板。例如，如果这个变量被设置为 *foo*，输出的可执行文件会是一个叫做 *foo* 的命令，其位置在目标机的 */system/bin/* 下。而如果 *LOCAL\_MODULE* 设置为 *libfoo*，并且引用 *BUILD\_SHARED\_LIBRARY* 而不是 *BUILD\_EXECUTABLE*，构建系统会产生一个 *libfoo.so* 并放在 */system/lib/* 目录下。

---

注 4： 同样做了一点整理，删除了空格和注释。

需要注意的是，对于任何一种模块类型（或者说是构建模板类型）来说，所有的名称必须唯一。也就是说，不可以有两个 *libfoo.so* 库文件。在将来模块名称估计会要求全局唯一。

#### LOCAL\_SRC\_FILES

构建模块所需要的源文件。可以使用构建系统定义的函数来提供给该变量，例如 Desk Clock 使用的 *all-java-files-under*，或者像 Service Manager 那样显式列出所有文件。

#### LOCAL\_PACKAGE\_NAME

跟其他模块不一样，应用程序使用这个变量而不是 LOCAL\_MODULE 来设置名字，你比较一下前面的两个 *Android.mk* 文件就可以看出来。

#### LOCAL\_SHARED\_LIBRARIES

给出该模块依赖的所有库文件。前面的例子可以看到，Service Manager 使用这个变量给出了其对 liblog 的依赖。

#### LOCAL\_MODULE\_TAGS

如前面所提到的，这个变量允许你控制在何种 TARGET\_BUILD\_VARIANT 下执行构建，通常它应该设置为 optional。

#### LOCAL\_MODULE\_PATH

用于覆盖你所构建的模块类型默认的安装路径。

查找更多 LOCAL\_\* 变量的一个好方法是查看 AOSP 中现有的 *Android.mk* 文件。同样，*clear\_vars.mk* 包含被清除的所有变量列表。虽然它没有给出每个变量的含义，但至少把它们都列了出来。

同样，除了清除影响所有 AOSP 的变量外，每个模块还可以通过提供 *CleanSpec.mk* 定义自己的清除规则，就像每个模块都提供 *Android.mk* 文件一样。不过，每个模块都需要提供 *Android.mk* 文件，却不一定需要 *CleanSpec.mk* 文件。默认情况下，构建系统对每种模块类型都有清除规则。但是，如果你的模块构建了构建系统默认情况下不构建的东西时，可以提供自己的 *CleanSpec.mk* 文件，因为构建系统默认情况下不知道如何清除它们。

## 输出

现在已经知道构建系统如何工作以及模块如何使用构建模板了，我们来看看它在 *out*/ 目录下产生的是什么内容。从一个高层视角来看，有三个步骤涉及输出，存在两种模

式（主机和目标机）：

1. 由模块的源代码产生中间文件。中间文件的格式和位置取决于模块的源码。例如，C/C++ 代码产生的是 `.o` 文件，Java 代码产生的是 `.jar` 文件。
2. 构建系统使用中间文件创建实际的二进制文件和文件包：以 `.o` 文件为例，将它们链接成实际的二进制文件。
3. 二进制文件和包被组装为构建系统需要的最终输出。例如，二进制文件被拷贝到包含根和 `/system` 的文件系统中，这个文件系统最后被用于产生设备的镜像文件。

`out/` 主要分为两个目录，反映它的两种操作模式：`host/` 和 `target/`。在每个目录下，你都可以看到一些 `obj` 目录，该目录里有很多构建时产生的中间文件。这些文件在构建系统时，存储在像 `BUILD_*` 宏名一样的特定辅助目录的子目录中：

- `EXECUTABLES/`
- `JAVA_LIBRARIES/`
- `SHARED_LIBRARIES/`
- `STATIC_LIBRARIES/`
- `APPS/`
- `DATA/`
- `ETC/`
- `KEYCHARS/`
- `PACKAGING/`
- `NOTICE_FILES/`
- `include/`
- `lib/`

你可能最感兴趣的目录是 `out/target/product/PRODUCT_DEVICE/`。镜像输出的位置会根据产品配置 `mk` 文件中定义的 `PRODUCT_DEVICE` 来确定。表 4-3 对这个目录进行了说明。

表 4-3：产品输出

项目	说明
<code>android-info.txt</code>	包含这个产品配置的代码名称
<code>clean_steps.mk</code>	包括清理代码树所需要执行的步骤，通过调用 <code>add-clean-step</code> 函数来提供在 <code>CleanSpec.mk</code> 文件中

表 4-3：产品输出（续）

项目	说明
<i>data/</i>	目标的 <i>/data</i> 目录
<i>installed-files.txt</i>	安装在 <i>data/</i> 和 <i>system/</i> 目录下的文件列表
<i>obj/</i>	目标产品的中间文件
<i>previous_build_config.mk</i>	最后的构建目标；在下次 <i>make</i> 时用于检查 <i>config</i> 是否有变化，从而强制执行 <i>installclean</i> 命令
<i>ramdisk.img</i>	基于 <i>/root/</i> 目录产生的 RAM 磁盘镜像
<i>root/</i>	目标根文件系统的内容
<i>symbols/</i>	放入根文件系统和 <i>/system</i> 目录下的二进制文件的原态版本
<i>system/</i>	目标机的 <i>/system</i> 目录
<i>system.img</i>	基于 <i>/system</i> 目录内容的镜像
<i>userdata.img</i>	基于 <i>/data</i> 目录内容的镜像

回头复习一下第 2 章关于根文件系统 */system* 和 */data* 的内容。从本质上讲，当内核启动时，它会挂载 RAM 磁盘镜像，然后执行找到的 */init* 文件。这个二进制文件，将运行 */init.rc* 脚本，并把 */system* 和 */data* 镜像挂载到其各自位置。我们会在第 6 章再来看看根文件系统的布局和启动时的系统操作。

## 构建脚本

了解构建系统的架构和功能之后，我们来看看那些最常见的以及不那么常见的构建脚本。我们只会稍微介绍一下使用这些脚本的结果，但是你在开始了解之前应该具备足够的知识。

### 默认的 droid 构建

前面我们已经多次使用 *make* 命令，但是并没有解释它的默认目标。实际上，直接运行 *make* 跟使用下面命令是一样的<sup>注5</sup>：

```
$ make droid
```

**droid** 实际上是 *main.mk* 中定义的默认构建目标。通常不用手动指定该目标。这里提到它只是为了完整性考虑，让读者了解而已。

---

注 5： 这里假设已经运行过 *envsetup.sh* 和 *lunch*。

## 查看构建命令

当构建 AOSP 时，你会注意到它不会显示实际运行时的命令。相反，它仅仅输出每个步骤的信息汇总。如果你想查看它运行的所有信息，就像 `gcc` 命令一样，可以在命令行加入 `showcommands` 目标：

```
$ make showcommands
...
host Java:apicheck (out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes)
for f in ; do if [ ! -f $f ]; then echo Missing file $f; exit 1; fi; unzip -qo $f -d out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes; (cd out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes && rm -rf META-INF); done
javac -J-Xmx512M -target 1.5 -Xmaxerrs 9999999 -encoding ascii -g -extdirs ""
-d out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes @out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/java-source-list-uniq || ( rm -rf out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes ; exit 41 )
rm -f out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/java-source-list
rm -f out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/java-source-list-uniq
jar -cfm out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/javalib.jar build/tools/apicheck/src/MANIFEST.mf -C out/host/common/obj/JAVA_LIBRARIES/apicheck_intermediates/classes .
Header: out/host/linux-x86/obj/include/libexpat/expat.h
cp -f external/expat/lib/expat.h out/host/linux-x86/obj/include/libexpat/expat.h
Header: out/host/linux-x86/obj/include/libexpat/expat_external.h
cp -f external/expat/lib/expat_external.h out/host/linux-x86/obj/include/libexpat/expat_external.h
Header: out/target/product/generic/obj/include/libexpat/expat.h
cp -f external/expat/lib/expat.h out/target/product/generic/obj/include/libexpat/expat.h
...
...
```

考虑前一节所讲的，它跟下面命令是一致的：

```
$ make droid showcommands
```

正如你很快就会看到的，使用这个命令会产生大量的输出从而让你没法去看。所以，你如果想要分析构建 AOSP 时运行的实际命令，你可能希望将标准输出和标准错误输出保存到文件里：

```
$ make showcommands > aosp-build-stdout 2> aosp-build-stderr
```

你也同样可以把所有输出合并到一个文件中：

```
$ make showcommands 2>&1 | tee build.log
```

有些人更喜欢使用 `nohup` 命令：

```
$ nohup make showcommands
```

## 针对 Linux 和 Mac OS 的 SDK 构建

Android 的 SDK 在官方网站 <http://developer.android.com> 上可以找到。当然，如果你扩展了核心 API 来增加新的功能，你想把它们发布给开发者们使用，也可以使用 AOSP 来构建你自己的 SDK。你可以通过使用下面命令组合来执行 SDK：

```
$ . build/envsetup.sh  
$ lunch sdk-eng  
$ make sdk
```

构建完成后，Linux 环境下的结果会输出到 *out/host/linux-x86/sdk/*，Mac 机器上会输出到 *out/host/darwin-x86/sdk/*。它们会以两份文件形式存在，一个是跟 <http://developer.android.com> 网站上一样的 zip 文件，另一个未压缩版可直接使用。

假设你已经使用 <http://developer.android.com> 上的指示来配置 Eclipse，并进行 Android 开发，你还需要两个额外的步骤以便能够使用新构建的 SDK。首先，你需要告诉 Eclipse 新的 SDK 的位置。要做到这一点，需要选择 Window → Preferences → Android，在 SDK 位置框里输入新 SDK 的路径，然后单击确定。另外，我在写本书的时候，以下做法的原因尚不完全清楚，你还需要到 Window → Android SDK Manager 中，取消选择 Tool 下除了前两个之外的所有被选择的项目，然后单击“Install 2 Packages.....”。做完之后，你就可以创建一个使用新 SDK 的新项目，并使用新开发的 API。如果你不做第二步，也可以创建新的 Android 项目，但 Java 库解析会存在问题，并因此而无法编译。

## 针对 Windows 系统的 SDK 构建

在 Windows 下构建的指令与 Linux 和 Mac OS 稍微有点不同：

```
$ . build/envsetup.sh  
$ lunch sdk-eng  
$ make win_sdk
```

输出结果会放在 *out/host/windows/sdk/* 目录里。

## 构建 CTS

如果你想构建 CTS，不需要使用 *envsetup.sh* 或 *lunch*。你可以直接输入：

```
$ make cts  
...  
Generating test description for package android.sax  
Generating test description for package android.performance  
Generating test description for package android.graphics
```

```
Generating test description for package android.database
Generating test description for package android.text
Generating test description for package android.webkit
Generating test description for package android.gesture
Generating test plan CTS
Generating test plan Android
Generating test plan Java
Generating test plan VM
Generating test plan Signature
Generating test plan RefApp
Generating test plan Performance
Generating test plan AppSecurity
Package CTS: out/host/linux-x86/cts/android-cts.zip
Install: out/host/linux-x86/bin/adb
```

*cts* 命令有自己的在线帮助，这里是 2.3/ 姜饼下输出的例子：

```
$ cd out/host/linux-x86/bin/
$ ./cts
Listening for transport dt_socket at address: 1337
Android CTS version 2.3_r3
$ cts_host > help
Usage: command options
Available commands and options:
Host:
  help: show this message
  exit: exit cts command line
Plan:
  ls --plan: list available plans
  ls --plan plan_name: list contents of the plan with specified name
  add --plan plan_name: add a new plan with specified name
  add --derivedplan plan_name -s/--session session_id -r/--result result_type:
  derive a plan from the given session
  rm --plan plan_name/all: remove a plan or all plans from repository
  start --plan test_plan_name: run a test plan
  start --plan test_plan_name -d/--device device_ID: run a test plan using the
  specified device
  start --plan test_plan_name -t/--test test_name: run a specific test
...
$ cts_host > ls --plan
List of plans (8 in total):
Signature
RefApp
VM
Performance
AppSecurity
Android
Java
CTS
```

一旦你的目标机运行起来后，例如 emulator 运行后，你可以启动测试套件并使用 *adb* 在目标机上运行测试：

```
$ ./cts start --plan CTS
Listening for transport dt_socket at address: 1337
Android CTS version 2.3_r3 Device(emulator-5554) connected
cts_host > start test plan CTS

CTS_INFO >>> Checking API...

CTS_INFO >>> This might take several minutes, please be patient...
...
```

## 构建 NDK

前面已经提到，NDK 有自己的构建系统、设置和帮助系统，你可以这样运行：

```
$ cd ndk/build/tools
$ export ANDROID_NDK_ROOT=aosp-root/ndk
$ ./make-release --help
Usage: make-release.sh [options]

Valid options (defaults are in brackets):

--help                  Print this help.
--verbose               Enable verbose mode.
--release=name          Specify release name [20110921]
--prefix=name            Specify package prefix [android-ndk]
--development=path       Path to development/ndk directory [/home/karim/
                        opersys-dev/android/aosp-2.3.4/development/ndk]
--out-dir=path           Path to output directory [/tmp/ndk-release]
--force                 Force build (do not ask initial question) [no]
--incremental            Enable incremental packaging (debug only). [no]
--darwin-ssh=hostname   Specify Darwin hostname to ssh to for the build.
--systems=list           List of host systems to build for [linux-x86]
--toolchain-src-dir=path Use toolchain sources from path
```

当你准备构建 NDK 时，可以像这样运行 *make-release*，并且可以看到它有力的警告：

```
$ ./make-release
IMPORTANT WARNING !!

This script is used to generate an NDK release package from scratch
for the following host platforms: linux-x86

This process is EXTREMELY LONG and may take SEVERAL HOURS on a dual-core
machine. If you plan to do that often, please read docs/DEVELOPMENT.TXT
that provides instructions on how to do that more easily.

Are you sure you want to do that [y/N]
y
Downloading toolchain sources...
...
```

## 更新 API

构建系统为了防止你修改 AOSP 的核心 API 而设置了安全措施，如果你做了修改，构建系统默认情况下会像这样显示失败：

```
*****
You have tried to change the API from what has been previously approved.

To make these errors go away, you have two choices:
 1) You can add "@hide" javadoc comments to the methods, etc. listed in the
    errors above.

 2) You can update current.xml by executing the following command:
    make update-api

  To submit the revised current.xml to the main Android repository,
  you will need approval
*****
```

make: \*\*\* [out/target/common/obj/PACKAGING/checkapi-current-timestamp] Error 38  
make: \*\*\* Waiting for unfinished jobs....

正如错误信息所述，为了让构建系统继续，你需要这样做：

```
$ make update-api
...
Install: out/host/linux-x86/framework/apicheck.jar
Install: out/host/linux-x86/framework/clearsilver.jar
Install: out/host/linux-x86/framework/droiddoc.jar
Install: out/host/linux-x86/lib/libneo_util.so
Install: out/host/linux-x86/lib/libneo_cs.so
Install: out/host/linux-x86/lib/libneo_cg.so
Install: out/host/linux-x86/lib/libclearsilver-jni.so
Copying: out/target/common/obj/JAVA_LIBRARIES/core_intermediates/emma_out/lib/cl
asses-jarjar.jar
Install: out/host/linux-x86/framework/dx.jar
Install: out/host/linux-x86/bin/dx
Install: out/host/linux-x86/bin/aapt
Copying: out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/emma_ou
t/lib/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/ext_intermediates/emma_out/lib/cla
sses-jarjar.jar
Install: out/host/linux-x86/bin/aidl
Copying: out/target/common/obj/JAVA_LIBRARIES/core-junit_intermediates/emma_out/
lib/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/emma_out/
lib/classes-jarjar.jar
Copying current.xml
```

下次开始 *make* 时，你就不会看到关于 API 修改的错误信息了。当然，此时你就跟官方系统不再兼容了，因此也不太可能通过 Google 的“Android”认证。

## 编译一个单独模块

现在，你已经知道如何构建整个目录树了。你也可以构建一个单独的模块。例如，你可以要求构建系统构建 Launcher2 模块（即 Home 界面）：

```
$ make Launcher2
```

也可以单独清除一个模块：

```
$ make clean-Launcher2
```

如果你希望构建系统重新产生一个包含你更新后模块的系统镜像，可以在 *make* 命令行中增加 *snod* 目标：

```
$ make Launcher2 snod
=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=generic
...
target Package: Launcher2 (out/target/product/generic/obj/APPS/Launcher2_interme
diates/package.apk)
  'out/target/common/obj/APPS/Launcher2_intermediates//classes.dex' as 'classes.d
ex'...
Install: out/target/product/generic/system/app/Launcher2.apk
Install: out/host/linux-x86/bin/mkyaffs2image
make snod: ignoring dependencies
Target system fs image: out/target/product/generic/system.img
```

## 目录树外编译

如果你希望在目录树外采用 AOSP 和 Bionic 库编译代码，你可以参考下面的 *makefile* 来完成<sup>注6</sup>：

```
# Paths and settings
TARGET_PRODUCT = generic
ANDROID_ROOT   = /home/karim/android/aosp-2.3.x
BIONIC_LIBC   = $(ANDROID_ROOT)/bionic/libc
PRODUCT_OUT    = $(ANDROID_ROOT)/out/target/product/$(TARGET_PRODUCT)
CROSS_COMPILE  = \
    $(ANDROID_ROOT)/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin/arm-eabi-
# Tool names
AS            = $(CROSS_COMPILE)as
AR            = $(CROSS_COMPILE)ar
CC            = $(CROSS_COMPILE)gcc
```

---

注 6：这个 *makefile* 是受 Row Boat 开发者 Amit Pundir 博客的启发，并基于《构建嵌入式 Linux 系统》第 4 章的 *makefile* 示例。

```

CPP      = $(CC) -E
LD       = $(CROSS_COMPILE)ld
NM       = $(CROSS_COMPILE)nm
OBJCOPY  = $(CROSS_COMPILE)objcopy
OBJDUMP  = $(CROSS_COMPILE)objdump
RANLIB   = $(CROSS_COMPILE)ranlib
READELF  = $(CROSS_COMPILE)readelf
SIZE     = $(CROSS_COMPILE)size
STRINGS  = $(CROSS_COMPILE)strings
STRIP    = $(CROSS_COMPILE)strip

export AS AR CC CPP LD NM OBJCOPY OBJDUMP RANLIB READELF \
        SIZE STRINGS STRIP

# Build settings
CFLAGS    = -O2 -Wall -fno-short-enums
HEADER_OPS = -I$(BIONIC_LIBC)/arch-arm/include \
            -I$(BIONIC_LIBC)/kernel/common \
            -I$(BIONIC_LIBC)/kernel/arch-arm
LDFLAGS   = -nostdlib -Wl,-dynamic-linker,/system/bin/linker \
            $(PRODUCT_OUT)/obj/lib/crtbegin_dynamic.o \
            $(PRODUCT_OUT)/obj/lib/crtend_android.o \
            -L$(PRODUCT_OUT)/obj/lib -lc -ldl

# Installation variables
EXEC_NAME = example-app
INSTALL   = install
INSTALL_DIR = $(PRODUCT_OUT)/system/bin

# Files needed for the build
OBJS      = example-app.o

# Make rules
all: example-app

.c.o:
    $(CC) $(CFLAGS) $(HEADER_OPS) -c $<
example-app: ${OBJS}
    $(CC) -o $(EXEC_NAME) ${OBJS} $(LDFLAGS)

install: example-app
    test -d $(INSTALL_DIR) || $(INSTALL) -d -m 755 $(INSTALL_DIR)
    $(INSTALL) -m 755 $(EXEC_NAME) $(INSTALL_DIR)

clean:
    rm -f *.o $(EXEC_NAME) core

distclean:
    rm -f *~
    rm -f *.o $(EXEC_NAME) core

```

在这种情况下，你就不再需要关心 *envsetup.sh* 或者 *lunch*，你可以直接输入下面这个魔法咒语：

```
$ make
```

显然，这将不会把二进制文件添加到任何由 AOSP 产生的镜像中。即使这里的 `install` 目标只有当你安装目标的文件系统 NFS 开启，并仅在调试期间才能体现其价值，但这却正是这个 makefile 被认为是有用的地方。在某种程度上，它也可以争辩说，使用这样的 makefile 文件实际上是适得其反，因为如果这段代码被添加为 AOSP 的模块组成部分，它将导致远比同等的 `droid.mk` 文件。

不过，这种情况也可以有用武之地。比如说在某些情况下，通过使用一个相当大的代码库而不是用它之外的 AOSP 来构建工程，通过这种做法来修改常规的构建系统，可能是有意义的。替代的做法是将该项目复制到 AOSP 中，并创建 `Android.mk` 文件来重现原来的常规的构建系统，就它本身而言，这可能被证明实质上是很有用的做法。

## 构建递归树

如果你真想尝试的话，可以在 AOSP 中给自己建一个 makefile 来构建一个组件，这个组件是基于递归的 makefiles 构建的，而不是像我在最后一节中建议的那样，试图用 `Android.mk` 文件复制相同的功能。在附录 E 中提到的几个 AOSP 分支，比如说包括在 AOSP 顶层的内核源代码和修改 AOSP 的主要的 makefile 文件来调用内核现有的编译系统。

这里有另一个例子，由 Linaro 的 Bernhard Rosenkranz 创建的用于构建 ffmpeg（这是一个依赖于 GNU 的自动工具类脚本），使用其原有构建文件的 `Android.mk` 文件：

```
include $(CLEAR_VARS)
FFMPEG_TCDIR := $(realpath $(shell dirname $(TARGET_TOOLS_PREFIX)))
FFMPEG_TCPREFIX := $(shell basename $(TARGET_TOOLS_PREFIX))
# FIXME remove -fno-strict-aliasing once the aliasing violations are fixed
FFMPEG_COMPILER_FLAGS = $(subst -I , -I../../,$(subst -include \
system/core/include/arch/linux-arm/AndroidConfig.h,,$(subst -include \
build/core/combo/include/arch/linux-arm/AndroidConfig.h,, \
$(TARGET_GLOBAL_CFLAGS)))) -fno-strict-aliasing -Wno-error=address \
-Wno-error=format-security
ifneq ($(strip $(SHOW_COMMANDS)),)
FF_VERBOSE="V=1"
endif

.PHONY: ffmpeg
droidcore: ffmpeg
systemtarball: ffmpeg

REALTOP=$(realpath $(TOP))

ffmpeg: x264 $(PRODUCT_OUT)/obj/STATIC_LIBRARIES/libvpx_intermediates/libvpx.a
mkdir -p $(PRODUCT_OUT)/obj/ffmpeg
cd $(PRODUCT_OUT)/obj/ffmpeg && \
export PATH=$(FFMPEG_TCDIR):$(PATH) && \
```

```
$(REALTOP)/external/ffmpeg/configure \
--arch=arm \
--target-os=linux \
--prefix=/system \
--bindir=/system/bin \
--libdir=/system/lib \
--enable-shared \
--enable-gpl \
--disable-avdevice \
--enable-runtime-cpudetect \
--disable-libvpx \
--enable-libx264 \
--enable-cross-compile \
--cross-prefix=$(FFMPEG_TCPREFIX) \
--extra-ldflags="-nostdlib -Wl,-dynamic-linker, \
/system/bin/linker,-z,muldefs$(shell if test $(PRODUCT_SDK_VERSION) -lt 16; \
then echo -n ', -T$(REALTOP)/$(BUILD_SYSTEM)/armelf.x'; fi),-z,nocopy reloc, \
--no-undefined -L$(REALTOP)/$(TARGET_OUT_STATIC_LIBRARIES) \
-L$(REALTOP)/$(PRODUCT_OUT)/system/lib \
-L$(REALTOP)/$(PRODUCT_OUT)/obj/STATIC_LIBRARIES/libvpx_intermediates -ldl -lc" \
--extra-cflags="$(FFMPEG_COMPILER_FLAGS)" \
-I$(REALTOP)/bionic/libc/include -I$(REALTOP)/bionic/libc/kernel/common \
-I$(REALTOP)/bionic/libc/kernel/arch-arm \
-I$(REALTOP)/bionic/libc/arch-arm/include -I$(REALTOP)/bionic/libm/include \
-I$(REALTOP)/external/libvpx -I$(REALTOP)/external/x264" \
--extra-libs="-lgcc" && \
$(MAKE) \
TARGET_CRTBEGIN_DYNAMIC_O=$(REALTOP)/$(TARGET_CRTBEGIN_DYNAMIC_O) \
TARGET_CRTEND_O=$(REALTOP)/$(TARGET_CRTEND_O) $(FF_VERBOSE) && \
$(MAKE) install DESTDIR=$(REALTOP)/$(PRODUCT_OUT)
```

## 基本的 AOSP 修改技巧

买本书的读者很可能是希望对 AOSP 进行修改以符合自己的要求。接下来这几页，我们将深入介绍一些你很可能想试着修改的地方。当然这里主要涉及的是编译构建环节，无论如何这都是必须首先搞清楚的地方。

---

**注意：**接下来的内容主要是基于 2.3/ 姜饼系统展开的，这些地方跟 4.2/ 果冻豆几乎是一样的，后续的大多数系统也将如此。事实上，这些机制在今后很长时间内都不会发生变化。当然，在 4.2/ 果冻豆里的更改我们会明显标注。

---

### 增加一个设备

读者阅读本书的原因之一（如果不是最重要原因的话）很可能是希望增加一个自定义设备。接下来我将解释如何做到这一点，因此你很可能需要在这一节设置个书签。当然，这里实际上只会告诉你如何调整构建系统。将 Android 系统移植到新的硬件上还

需要做大量的工作。但是，增加一个设备到构建系统很明显会是你需要做的第一件事。幸运的是，这个事情比较简单。

为了更方便地解释这个小实验，这里假设你在为一个叫做 ACME 的公司工作，并且你接受了一个任务——发布最新的玩具：CoyotePad，定位为玩鸟类游戏中最好的产品。作为开始，我们首先在 *device/* 目录下创建这个新设备：

```
$ cd ~/android/aosp-2.3.x  
$ . build/envsetup.sh  
$ mkdir -p device/acme/coyotepad  
$ cd device/acme/coyotepad
```

接下来，我们需要一个 *AndroidProducts.mk* 文件来描述为 CoyotePad 构建的各种 AOSP 产品：

```
PRODUCT_MAKEFILES := \  
$(LOCAL_DIR)/full_coyotepad.mk
```

虽然我们可以描述多个产品（可以把 *build/target/product/AndroidProducts.mk* 作为例子），但是大多数情况还是像这个例子一样只指定一个，这里就在 *full\_coyotepad.mk* 中进行描述。

```
$(call inherit-product, $(SRC_TARGET_DIR)/product/languages_full.mk)  
# If you're using 4.2/Jelly Bean, use full_base.mk instead of full.mk  
$(call inherit-product, $(SRC_TARGET_DIR)/product/full.mk)  
  
DEVICE_PACKAGE_OVERLAYS :=  
  
PRODUCT_PACKAGES +=  
PRODUCT_COPY_FILES +=  
  
PRODUCT_NAME := full_coyotepad  
PRODUCT_DEVICE := coyotepad  
PRODUCT_MODEL := Full Android on CoyotePad, meep-meep
```

很有必要仔细看看这个 makefile。首先，我们使用了继承的方式来告诉构建系统把其他产品描述拉进来作为我们产品描述的基础。这使得我们能够基于其他人的工作来构建这个 makefile，而无需从头一点点来指定包括的 AOSP 内容。*languages\_full.mk* 会拉进来大量的字库，而 *full.mk* 会确保我们使用 *full-eng* 组合构建时有一样多的模块。

这里介绍一下这些变量：

#### DEVICE\_PACKAGE\_OVERLAYS

允许我们指定一个目录，这个目录将覆盖原来的目录，会被应用到 AOSP 的源代码中，从而使我们能用特定设备的资源来替换默认包资源。比如说，如果你想

要为 Launcher2 或其他应用程序设置自定义的布局或颜色，你会发现这个是很有用的。我们将在下一节中讲述它是如何使用的。

#### PRODUCT\_PACKAGES

这个变量允许我们指定除了我们已经继承的产品以外的包。例如，如果在 *device/acme/cyotepad/* 目录下有定制的应用、二进制程序和库文件，我们就希望把它们加入到这里以便在产生系统镜像时加进去。注意，这里使用的 `+=` 符号，它使我们可以在现有的值上面增加内容而不是替代现有的值。

#### PRODUCT\_COPY\_FILES

设置希望被拷贝到目标文件系统中的文件以及文件路径。每个源 / 目的地对用分号分开，地址对本身用空格分开。这个变量对配置文件和预编译文件（例如固件镜像和内核模块等）很有用。

#### PRODUCT\_NAME

配置 `TARGET_PRODUCT`，用于你选择 *lunch* 组合或者给 *lunch* 传递组合参数。例如：

```
$ lunch full_coyotepad-eng
```

#### PRODUCT\_DEVICE

产品被销售给最终客户使用的名称。`TARGET_DEVICE` 变量就是从这个变量推导出来的。`PRODUCT_DEVICE` 与 *device/acme/* 下面的目录项匹配，因为构建系统会在这个目录项下找到对应的 *BoardConfig.mk* 文件。在这个例子中，这个变量就跟已经存在的目录名一样。

#### PRODUCT\_MODEL

在产品的设置程序中，“关于本机”中显示的“Model Number”。这个变量实际上存储在全局属性 `ro.product.model` 里，可以通过程序获取。

在 4.2/ 果冻豆版本中，还包括了一个 `PRODUCT_BRAND` 变量，它通常设置为 `Android`。这个变量通过全局属性 `ro.product.brand` 来获取。这个全局属性被系统的其他部分用于根据设备厂商来执行设备相关代码。

描述完产品后，我们也必须提供一些关于设备主板的基本信息，这些信息通过 *BoardConfig.mk* 文件提供：

```
TARGET_NO_KERNEL := true  
TARGET_NO_BOOTLOADER := true  
TARGET_CPU_ABI := armeabi  
BOARD_USES_GENERIC_AUDIO := true  
  
USE_CAMERA_STUB := true
```

这是一个保证能编译通过的最简单的 *BoardConfig.mk* 文件。想要看看更真实的版本，参见 2.3/ 姜饼版本中的 *device/samsung/crespo/BoardConfigCommon.mk*，或者 4.2/ 果冻豆系统中的 *device/asus/grouper/BoardConfigCommon.mk*。

为了编译在 *device* 目录下的模块，你还需要提供一个传统的 *Android.mk* 文件以构建这些模块：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

ifeq ($(filter coyotepad,$(TARGET_DEVICE)),)
include $(call all-makefiles-under,$(LOCAL_PATH))
endif
```

把所有设备特定的应用程序、二进制文件和库都放在 *device* 目录，而不是全局 AOSP 其余部分中，这是一种首选的做法。正如我刚才解释的，如果你在这里添加了模块，别忘了也把它们添加到 *PRODUCT\_PACKAGES* 中。否则，如果你只是把它们添加在这里，为它们提供有效的 *Android.mk* 文件，它们将构建，但他们不会被包含在最后的镜像中。

如果有多个产品共用同一套软件包，你可能希望创建一个包含共享包的 *device/acme/common/* 目录。你可以看到在 4.2/ 果冻豆系统中的 *device/generic* 下的例子。在这个版本中，还可以看看 *device/samsung/maguro/device.mk* 是如何继承 *device/samsung/tuna/device.mk* 的，这是一个设备基于其他设备的例子。

最后一个步骤，为了让我们刚才添加的东西在 *envsetup.sh* 和 *lunch* 中可用。要做到这一点，你需要添加一个 *vendorsetup.sh* 在你的设备的目录中：

```
add_lunch_combo full_coyotepad-eng
```

你同样需要确保它的权限符合可执行要求：

```
$ chmod 755 vendorsetup.sh
```

我们现在可以返回 AOSP 的根目录，开始为我们全新的 ACME CoyotePad 启动编译：

```
$ croot
$ . build/envsetup.sh
$ lunch

You're building on Linux

Lunch menu... pick a combo:
 1. generic-eng
 2. simulator
 3. full_coyotepad-eng
 4. full_passion-userdebug
 5. full_crespo4g-userdebug
 6. full_crespo-userdebug
```

```

Which would you like? [generic-eng] 3

=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=2.3.4
TARGET_PRODUCT=full_coyotepad
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=GINGERBREAD
=====

$ make -j16

```

正如你所看到的,AOSP现在找到了我们的新设备并打印出了相关信息。当编译结束时,我们同样得到了跟其他AOSP构建时一样类型的输出,只是这个输出将是一个产品的特定的目录。

```

$ ls -al out/target/product/coyotepad/
total 89356
drwxr-xr-x 7 karim karim 4096 2011-09-21 19:20 .
drwxr-xr-x 4 karim karim 4096 2011-09-21 19:08 ..
-rw-r--r-- 1 karim karim 7 2011-09-21 19:10 android-info.txt
-rw-r--r-- 1 karim karim 4021 2011-09-21 19:41 clean_steps.mk
drwxr-xr-x 3 karim karim 4096 2011-09-21 19:11 data
-rw-r--r-- 1 karim karim 20366 2011-09-21 19:20 installed-files.txt
drwxr-xr-x 14 karim karim 4096 2011-09-21 19:20 obj
-rw-r--r-- 1 karim karim 327 2011-09-21 19:41 previous_build_config.mk
-rw-r--r-- 1 karim karim 2649750 2011-09-21 19:43 ramdisk.img
drwxr-xr-x 11 karim karim 4096 2011-09-21 19:43 root
drwxr-xr-x 5 karim karim 4096 2011-09-21 19:19 symbols
drwxr-xr-x 12 karim karim 4096 2011-09-21 19:19 system
-rw----- 1 karim karim 87280512 2011-09-21 19:20 system.img
-rw----- 1 karim karim 1505856 2011-09-21 19:14 userdata.img.

```

同样,可以看看`system/`目录下的`build.prop`文件,它包括了目标设备上运行时可以获得的跟我们配置和构建相关的全局属性:

```

#begin build properties
# autogenerated by buildinfo.sh
ro.build.id=GINGERBREAD
ro.build.display.id=full_coyotepad-eng 2.3.4 GINGERBREAD eng.karim.20110921.190849 test-keys
ro.build.version.incremental=eng.karim.20110921.190849
ro.build.version.sdk=10
ro.build.version.codename=REL
ro.build.version.release=2.3.4
ro.build.date=Wed Sep 21 19:10:04 EDT 2011

```

```
ro.build.date.utc=1316646604
ro.build.type=eng
ro.build.user=karim
ro.build.host=w520
ro.build.tags=test-keys
ro.product.model=Full Android on CoyotePad, meep-meep
ro.product.brand=generic
ro.product.name=full_coyotepad
ro.product.device=coyotepad
ro.product.board=
ro.product.cpu.abi=armeabi
ro.product.manufacturer=unknown
ro.product.locale.language=en
ro.product.locale.region=US
ro.wifi.channels=
ro.board.platform=
# ro.build.product is obsolete; use ro.product.device
ro.build.product=coyotepad
# Do not try to parse ro.build.description or .fingerprint
ro.build.description=full_coyotepad-eng 2.3.4 GINGERBREAD eng.karim.20110921.190
849 test-keys
ro.build.fingerprint=generic/full_coyotepad/coyotepad:2.3.4/GINGERBREAD/eng.kari
m.20110921.190849:eng/test-keys
# end build properties
...
```

---

**警告：**在为真实设备进行构建之前，你可能需要仔细地审查默认的属性。很多开发者都遇到过因为默认值导致的严重问题。例如，在 2.3/ 姜饼系统和 4.2/ 果冻豆系统中全局属性 `ro.com.android.data roaming` 很多都设置为 `true`。所以如果你在针对一个连接到移动网络的设备进行开发的话，把这个值设置为 `false` 就会节省不少钱。

---

正如你能够想到的那样，为了确保 AOSP 能够在我们的硬件上运行，在这里有很多事情需要完成。但是前面的步骤是我们的出发点。然而，通过将板子的特定改变隔离在一个单独的目录中，这种结构将简化 CoyotePad 支持 AOSP 的下一个发布版本。事实上，到时候只是需要将相应的目录复制到新 AOSP 的 `device/` 目录以及调整其中代码来适应新的 API。

## 增加应用

添加一个应用程序到你的板子上是比较简单的。首先，试着用 Eclipse 和默认的 SDK 创建一个 HelloWorld 应用程序。在 Eclipse 中的所有新的 Android 工程在默认情况下都是一个 HelloWorld 工程。然后从 Eclipse 工作区中复制该应用程序到它的目的地：

```
$ cp -a ~/workspace/HelloWorld ~/android/aosp-2.3.x/device/acme/coyotepad/
```

之后你要在 *osp-root/device/acme/coyotepad>HelloWorld/* 目录中创建一个 *Android.mk* 文件，来编译这个应用程序：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-java-files-under, src)
LOCAL_PACKAGE_NAME := HelloWorld

include $(BUILD_PACKAGE)
```

由于我们把这个模块标记为 *optional*，因此默认情况下 AOSP 构建时不会包含该模块。为了把该模块包含进去，你需要把它加入到 CoyotePad 设备的 *full\_coyotepad.mk* 文件下的 *PRODUCT\_PACKAGES* 变量的列表中。

如果你不仅仅想把这个应用添加到某一块板上，而是希望把它加入到与 AOSP 相关的所有产品中，你就需要把它放在 *packages/apps/* 目录中。你同样也需要修改内置的 *.mk* 文件，例如 *aosp-root/build/target/product/core.mk*，以便你的应用被一同构建。但是，由于需要对每一个新发布的 AOSP 版本进行修改，所以这个方案不具备可移植性，我们不推荐采用这种方式。如前所述，最好的方式还是把你的定制修改尽量隔离在 *device/acme/coyotepad/* 目录中。

## 添加应用程序覆盖

有时候，你可能不希望完全新增一个应用程序，而是对 AOSP 默认包含的应用程序进行修改。应用程序覆盖（App Overlay）就是做这个事情的。*overlay* 是 AOSP 中用于设备厂商修改默认资源而不用修改 AOSP 中原有资源的机制。要使用这个特性，你必须创建一个覆盖目录树并告诉构建系统。*overlay* 最简单的位置是放在设备相关目录里，就像我们上一节创建的目录一样：

```
$ cd device/acme/coyotepad/
$ mkdir overlay
```

为了让编译 / 构建系统识别这个目录，我们需要修改 *full\_coyotepad.mk* 文件：

```
DEVICE_PACKAGE_OVERLAYS := device/acme/coyotepad/overlay
```

如果仅仅是这样，*overlay* 也没有什么用。让我们来介绍修改 Laucher2 的默认字符串。可以作如下修改：

```
$ mkdir -p overlay/packages/apps/Launcher2/res/values  
$ cp aosp-root/packages/apps/Launcher2/res/values/strings.xml \  
> overlay/packages/apps/Launcher2/res/values
```

这个时候你就可以修改你自己的 *strings.xml* 文件来覆盖你修改的字符串。最重要的是，你的设备就会有一个采用你定制字符串的 Launcher2，而默认 Launcher2 会保持原始字符串。所以，其他人如果依赖同一个 AOSP 源就看到的是原始字符串。你可以通过这种方式修改大多数资源，包括图片和 XML 文件。只要你把文件放在 *device/acme/coyotepad/overlay/* 中的同一个层次，就会被构建系统识别。

---

**警告：** overlay 只可以对资源使用，不能覆盖源代码。例如，如果你想定制 Android 的内部部件，你仍然不得不在原地修改。至少在目前，还没有办法将你板子的定制修改隔离起来。

---

## 添加本地工具或守护进程

跟前面为你的板子添加应用的例子一样，你可以在 *device/acme/coyotepad/* 目录中添加你定制的本地工具程序或者守护进程。显然，你需要在包含代码的目录里提供一个 *Android.mk* 文件来构建这个模块。

```
LOCAL_PATH:= $(call my-dir)  
include $(CLEAR_VARS)  
  
LOCAL_MODULE := hello-world  
LOCAL_MODULE_TAGS := optional  
LOCAL_SRC_FILES := hello-world.cpp  
  
LOCAL_SHARED_LIBRARIES := liblog  
  
include $(BUILD_EXECUTABLE)
```

在这个例子中，你还需要确保 *hello-world* 是 CoyotePad 的 *PRODUCT\_PACKAGES* 中的一部分。

如果你打算让你的二进制程序添加到所有产品中，而不仅仅是当前开发的主板，那么你就需要知道目录树中本地工具程序和守护程序的位置。以下是最重要的一些路径：

### *system/core* 和 *system/*

定制的 Android 二进制程序，也意味着这些程序是独立于 Android 框架运行或者是独立的应用程序。

### *frameworks/base/cmds*

与 Android 框架紧密耦合的二进制程序，例如，Service Manager 和 *installd* 就放在这里。

*external/*

AOSP 中导入的第三方项目产生的二进制程序。例如，*strace* 就放在这里。

一旦确定好你的二进制程序应该放在上面列表中的哪一个目录之后，你还需要添加到其中的全局 *.mk* 文件里，例如 *aosp-root/build/target/product/core.mk*。然而，如前所述，这样的全局修改是不推荐的，因为它们在移植到新的 AOSP 发布版时会很麻烦。

## 添加本地库文件

与 App 和二进制应用一样，你还可以为你的主板添加本地库文件。假设跟之前一样，编译库的源文件放在子目录 *device/acme/coyotepad/* 中，你将需要修改 *Android.mk* 来编译它：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := libmylib
LOCAL_MODULE_TAGS := optional
LOCAL_PRELINK_MODULE := false
LOCAL_SRC_FILES := $(call all-c-files-under,.)

include $(BUILD_SHARED_LIBRARY)
```

---

注意：请注意，从 4.0/Ice-Cream Sandwich 开始 *LOCAL\_PRELINK\_MODULE* 被清除，并且不再需要使用。

---

要使用这个库，你必须把它加入到依赖它的二进制程序的 *Android.mk* 文件的相应项目里：

```
LOCAL_SHARED_LIBRARIES := libmylib
```

你还很可能需要添加相应的头文件目录（例如 *include/* 目录），它们应该在放置你的库文件的时候一起放到了相应目录中，使得跟你的库文件链接的代码能够找到这些头文件，例如 *device/acme/coyotepad/include*。

如果你希望你的库文件能应用于所有的 AOSP，而不仅仅是你当前的设备，你还需要代码树中各路径的更多信息。首先你必须知道，与二进制程序不一样，大多数的库文件只在一个单一应用里使用，其他地方根本用不到。所以，这些库文件基本上应该放在程序里而不是库文件的常用位置，这些位置会使得系统全局可访问。这些位置主要有：

### *system/core/*

这个目录里的库文件被系统的很多部分用到，包括 Android 框架以外的一些组件。例如，liblog 就放在这里。

### *frameworks/base/libs/*

与框架紧密关联的库文件放在这里，libbinder 就在这。

### *frameworks/native/libs*

在 4.2/ 果冻豆系统中，很多 2.3/ 姜饼系统放在 *frameworks/base/libs/* 的库文件被移到了 *frameworks/native/libs* 目录。

### *external/*

AOSP 导入的外部工程产生的库文件放在这个目录，例如 OpenSSL 的 libssl 就在这。

同样，如果不希望使用 CoyotePad 相关的库头文件目录，你可以使用 *system/core/include/* 或者 *frameworks/base/include/* 这样的全局目录，以及在 4.2/ 果冻豆中的 *frameworks/native/include/* 中。如前所述，你需要仔细评估这样的全局修改是否真的必要，因为它意味着在移植到新版本系统时增加额外的移植工作。

## Library Prelinking

如果你仔细看了前面为这个示例库提供的 *Android.mk* 文件，你会注意到 *LOCAL\_PRELINK\_MODULE* 变量。为了减少装载库文件的时间，Android 版本 2.3 以前的系统采用了 *prelink* 技术来处理大多数的库文件。Prelinking 提前指定变量 / 函数的实际地址而不是等到运行期来计算。2.3/ 姜饼指定地址的文件是 *build/core/prelink-linux-arm.map*，执行映射的工具名为 *apriori*。它包含了以下一些项目：

```
#core system libraries
libdl.so          0xAF00000 # [<64K]
libc.so           0xAFD0000 # [~2M]
libstdc++.so      0xAF00000 # [<64K]
libm.so           0xAFB0000 # [~1M]
liblog.so         0xAFA0000 # [<64K]
libcutils.so      0xAF90000 # [~1M]
libthread_db.so   0xAF80000 # [<64K]
libz.so            0xAF70000 # [~1M]
libevent.so       0xAF60000 # [???]
libssl.so          0xAF40000 # [~2M]
...
# assorted system libraries
libssqlite.so     0xA8B0000 # [~2M]
libexpat.so        0xA8A0000 # [~1M]
```

```
libwebcore.so      0xA8300000 # [~7M]
libbinder.so       0xA8200000 # [~1M]
libutils.so        0xA8100000 # [~1M]
libcameraservice.so 0xA8000000 # [~1M]
libhardware.so     0xA7F00000 # [<64K]
libhardware_legacy.so 0xA7E00000 # [~1M]
...
...
```

如果你想在 2.3/ 姜饼系统中添加定制的库文件，你要么把它加入到 *prelink-linux-arm.map* 的列表里，或者设置 `LOCAL_PRELINK_MODULE` 为 `false`。如果你忘了执行这个步骤，编译就会失败。

库文件的 prelink 技术从 4.0/ 冰淇淋三明治系统开始就放弃了。

# 硬件基础

现在，你已经对 Android 的编译构建系统有所了解了，而接下来我们将探索编译得到的镜像文件是如何在目标板上运行的。而在开始之前，我们先要回过头来看看 Android 系统通常运行在何种硬件配置上。事实上，虽然 Android 被设计为可以运行在广泛类型的嵌入式系统上，但是它仍然着重用于消费类电子产品，或者更明显的是手机。

我们会以支持 Android 系统运行的典型硬件架构开始进行介绍，然后会讨论用于运行 Android 系统的典型 SoC（System On Chip，片上系统）架构，并提供几个典型 SoC 的大致介绍。我们同样也会说一下虚拟地址空间和物理地址空间的差异以及典型的主机—目标机调试系统的建立。最后会以一系列的评估板来结束这一章，你可以通过这些板子来设计你的嵌入式 Android 系统原型。

## 典型的系统架构

正如我们在第 1 章所讨论的，Android 本应该可以运行在任何支持 Linux 系统的硬件上。然而，Android 并不是凭空开发出来的。它的设计源于手机，而且它现在的架构仍然可以看出这一点。图 5-1 描述了针对 Android 的原型嵌入式系统架构框图。你的实际目标机器可能有差异，并且很可能差异非常大。但是，作为讨论的基础，这张图应该就足够了。

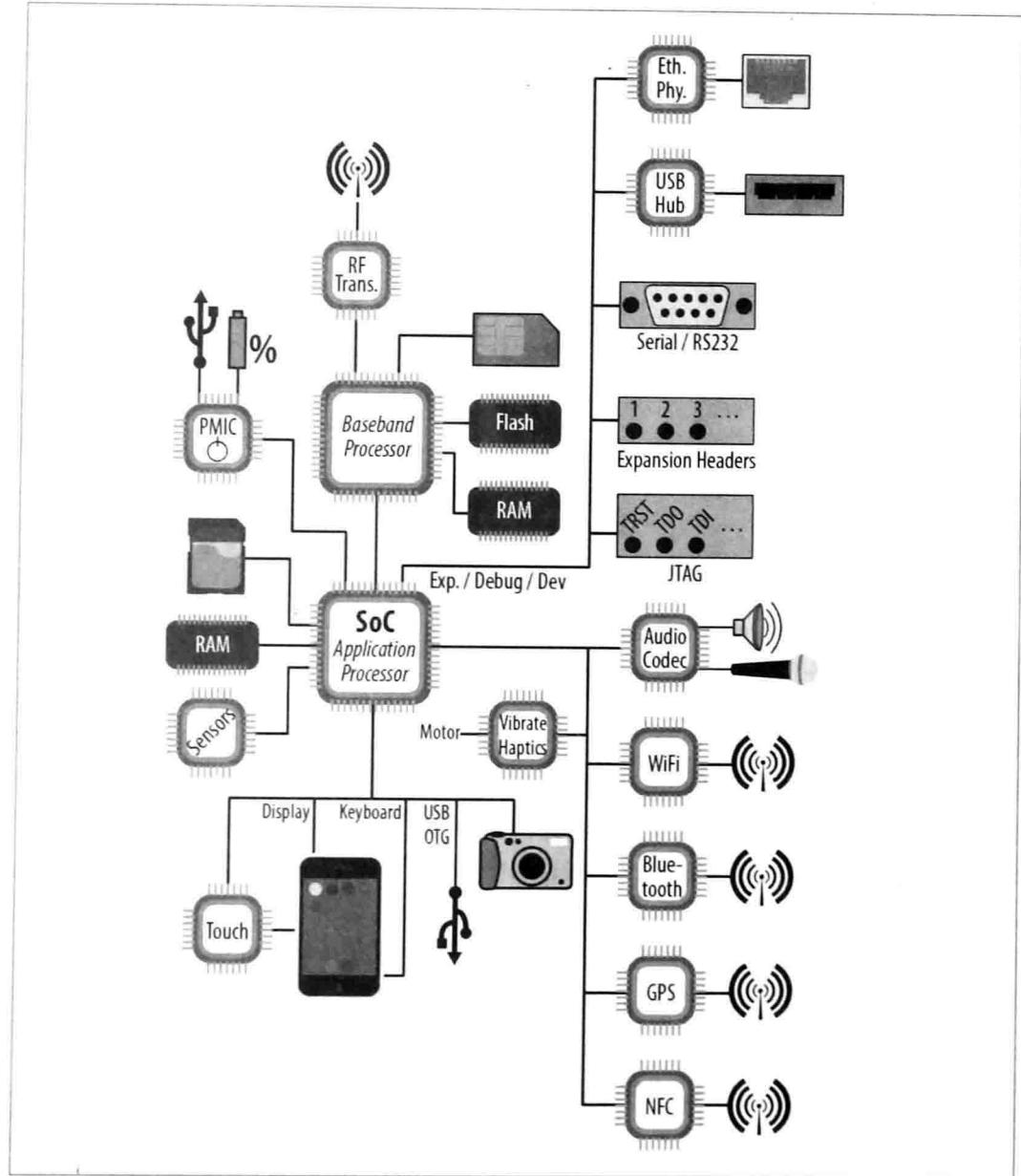


图 5-1：典型系统架构框图

需要注意的最重要的事情是，这个系统的中心就是 SoC。我们会在下一节更多地来讨论 SoC 的细节。简单地说，SoC 就是一个 CPU 和一系列外设控制器集成在一个集成电路（IC）硅片上。目标板上的所有其他组件都通过这样或那样的方式最后连接到 SoC 上。Android 系统主要就运行在 SoC 上，然后控制并访问板上的所有外设。

## 基带处理器

下一个需要关注的组件是基带处理器（Baseband Processor）。市场上大多数的手机采用了相互独立的处理单元来分别处理用户界面软件和射频功能。这就是我们常说的应用处理器（Application Processor, AP）和基带处理器（Baseband Processor, BP）。

你可能会奇怪，为什么需要两个独立的处理器而不是由一个来完成。这个原因包括法律和技术两个方面。首先，在美国，法律要求软件定义无线电（SDR）设备必须被美国通信委员会（FCC）认证，而这个认证有一个要求就是控制无线信号的软件不可以受到未经授权的更改。基本上，它的意思是终端用户在任何情况下都不允许修改无线调制方式和工作频率。同时，无线功能的很多操作有很强的实时性限制，因此控制无线信号的软件跟用户界面操作系统跑在同一个CPU上就不是一个很好的选择。另外还有一个好处是，可以在AP休眠的同时保持BP继续运行。

当然，这里只是说了一些皮毛，关于这个话题有太多的东西要说。这里为了讨论方便，假设没有任何办法能够让Android系统和控制无线信号的软件运行在同一个处理器上。显然，如果你的嵌入式系统不是一个手机，或者根本就没有无线功能，可以就直接忽略框图里面的BP以及跟它相关的部分。

但是，理解BP以及它与AP交互的方式还是很有必要的，因为Android的RIL跟其依赖的硬件是紧耦合的关系。简单地说，BP与AP交互的方式主要是基于串口的AT指令集。可以看到，BP有它自己的Flash和RAM，这就保证了运行在BP上的认证软件与AP上的软件相互隔离。运行在BP上的实时操作系统（RTOS, Real-time OS）只关注一件事情：无线协议处理，例如运行GSM协议栈。同样需要知道的是，SIM卡和RF收发器是接到BP上的。射频收发器处理与铁塔之间的实际射频发送和接收工作，而SIM卡用于蜂窝网络运营商识别手机用户身份。

---

**注意：**电信和无线通信技术是一个复杂的领域，我这里说的只是皮毛。实际的设计比我这里说的要复杂得多。例如，现代AP-BP之间的交互很可能并不是真的使用串口线或者AT指令，而是使用映射内存或私有的握手协议。不过，对于当前的话题，这个简化还是足够的。

如果你想得到关于智能手机无线架构的更多信息，建议你看看Harald Welte的“Anatomy of contemporary GSM cellphone hardware”并浏览在xda-developers上的这个帖子。

---

## 核心器件

虽然我们讨论的很多器件很可能存在，也很有可能不在你的嵌入式系统中，但是有一

些器件基本上一定会在所有嵌入式系统中，不管是 Android 系统：RAM 和存储。对于 RAM 没有太多可以说的，但是存储可能就有很多种形式了。

大多数传统的嵌入式系统都包含 Flash，不管是 NOR 还是 NAND，以及基于其之上的 Flash 文件系统，用于管理芯片资源并且实现各种功能。然而目前的主流趋势是改为采用嵌入式多媒体卡（embedded MultiMediaCard，eMMC）。本质上，这些芯片跟 SD 卡差不多，在 Linux 内核中被看作传统的块设备（打个比方，就跟传统的 ATA 硬盘一样）。所以，这些系统就不再有 NOR 或者 NAND，就只有一个 eMMC 芯片。这些系统的 SoC 芯片都有响应的模块来执行基本的 eMMC 接口读操作并具备从这个接口设备上引导系统的能力。

而且，系统中的存储设备很可能不止有一个。实际上，Android 系统会区分“内部”（internal）存储和“外部”（external）存储。内部存储一般就是指板上的 eMMC，而外部存储则是手机或者平板上用于可插拔的 SD 卡。内部存储中包含了 Android 系统本身用于引导，以及基本的文件系统操作。而外部存储则主要用于存储图片以及其他多媒体内容。当然，如果你的设备既不是手机也不是平板，这样的区分其实就没有什意义，但是 Android 应用开发的 API 却反映出了 Android 手机的特性，而且对于这两种存储做了比较明确的区分。

---

注意：需要注意的是，在一些比较近期的设备上，外部存储采用了 FUSE（Filesystem in User Space，用户空间文件系统）将系统内部存储的内容挂载到文件系统外部存储位置。所有 Nexus 设备就是这样，例如 Galaxy Nexus、Nexus 4、7 和 10。

---

在电池供电的设备上，你很可能看到的另一个部件是电源管理芯片（PMIC，Power Management IC）。PMIC 的作用是管理电池的各个方面，包括稳压和充电。PMIC 通常连接到电池上，为板子提供 DC 电源。在大多数消费类设备中，DC 电源来源于 USB 的 OTG 连接器，它作为电源充电器的插件，使充电器加倍。对于非移动设备来说（甚至对于某些移动设备也适用），外部电源并不通过 USB 提供，而是通过其他形式的连接器，例如 barrel 端子。

PMIC 与 SoC 一般通过 SPI、I2C 或者 GPIO 连接。当电压过低或者接上充电器时，PMIC 会产生相应的中断。它不仅包括电源管理功能（而且越来越多是这样），例如它可以包括实时时钟（RTC）、音频编解码器，以及 USB 收发器等。

## 现实交互

很明显，Android 主要是一个面向用户的系统。正如兼容性定义文档（CDD）中所建议的，

基于 Android 的系统应该允许丰富的用户交互和包括丰富的硬件组件，以响应直接物理环境。这也意味着有相当多的硬件部件专门从事该项任务。

首先要有直接的用户交互部件，诸如显示器、触摸输入和键盘。手机通常直接使用 SoC 集成的显示功能，具有较大的显示器，像平板电脑还通常有驱动 LVDS 的 LCD 桥。还有通常用来处理屏幕交互传感器的触摸控制器，以及用来处理设备上的键盘或物理按键的电路。

第二，还有一些用于与现实交互的器件，包括被 SoC 控制的摄像头（或者多个摄像头，例如某些设备既有一个前置摄像头，还有一个后置摄像头），以及被音频编解码 IC 控制的音频 IO。而且，设备也还包括一系列的元件用于传感现实环境的物理参数并与之进行物理交互。

在 Android 设备里面，还可以看到很多传感器，例如加速度传感器、陀螺仪、温度计、气压计、光度计、磁力传感器和接近传感器。为了简化这个框图，我在图里的板子上只画了一个传感器芯片。还有很多元件用于提供震动以及触摸反馈，而且涉及大量元器件。

## 连接性

Android 的一个重要特性是连接性，其依赖的硬件包括控制器、连接器以及天线，包括 USB、WIFI、Bluetooth、GPS 和 NFC 等标准。而且这些功能越来越倾向于封装到一颗芯片中而不是多颗 IC。

市面上大多数消费类的 Android 设备只提供了一个 USB 的 OTG 连接器用于将其连接到计算机上或者插入另一个 USB 设备，例如摄像头或者 U 盘。有很多一些设备也会让 USB 连接器用作 USB 主设备。甚至少数设备还提供独立的 USB 主连接器用于插入外设，就像你通常在 PC 或者 Mac 上看到的 USB 主接口一样。

## 扩展、开发与调试

除了我在前面提到的主流 Android 设备上看到的主流器件以外，SoC 还能提供很多其他器件和外设，而大多数消费类手机或平板上都没有。它们很明显可以用于其他基于 Android 的嵌入式系统中。这些外设有些得到 Android 支持，而有些不支持。而这些又恰恰就是我们想集成到嵌入式系统中的，对吧？要走其他开发者没走过的路？

你会在开发板上很容易看到支持以太网、USB Host、串口（RS-232）、JTAG 以及扩展槽的器件。例如，很流行的 BeagleBoard 和 PandaBoard 就有大量这些器件。JTAG

是一种硬件级别的调试接口，所以不需要 Linux 内核或 Android 框架的任何软件支持。开发板上还有扩展槽以允许外设板（附加模块通过扩展槽接到板上）连接到 SoC 的引脚上，例如 I<sup>2</sup>C 或者 GPIO 脚。你需要确保你装载了相应的设备驱动程序以使得 Linux 能够跟附加模块上的外设进行通信。

串口连接由 Linux 内核的 TTY 层提供支持。只要你的内核支持串口控制台（如果 Linux 运行在 SoC 上基本都会支持），这个功能基本上是“即插即用”的。串口连接对于嵌入式系统来说至关重要，特别是在调试板子的时候，因为它是主机与目标板通信既简单又有效的方法。

如果你使用 Android 3.1 以上的系统，它会支持 USB Host 模式。早期的版本，包括姜饼系统，在 Android 框架里不支持 USB Host 模式。但是，它并不能排除其依赖 Linux 内核支持，它仅仅是说 Android 3.1 系统以后所提供的 USB Host 模式 API 在之前的系统中不存在。

同样的情况也适用于以太网。虽然你可以通过以太网连接到网上，但是 Android 框架并不能将以太网当成是一个有效的通信链路，只有 WIFI 和包交换（例如，移动网络的数据连接）才是。所以，有些应用在以太网连接时可以正常工作，但是有些就不能。

## 为 Android 增加以太网支持

Android 系统默认情况对以太网接口的支持不太好，但也并不绝对。如果你对这个功能感兴趣，可以看看以下这些资料：

- Fabien Brisset 和 Benjamin Zores 已经整合了对 4.0/ 冰淇淋三明治系统和 4.1/ 果冻系统的以太网补丁。这些补丁在 GitHub (<https://github.com/gxben/aosp-ethernet>) 上可以找到，而且你还可以找到 Benjamin 在 2012 年 10 月欧洲嵌入式 Linux 系统会议（Embedded Linux Conference Europe, <http://www.slideshare.net/gxben/elce-2012-dive-into-android-networking-adding-ethernet-connectivity>）上对这项工作的报告。
- Linaro 也创建了针对这个功能的一系列补丁。这些补丁可以在这里 (<http://review.android.git.linaro.org/#change, 2599>)，这里 (<http://review.android.git.linaro.org/#change, 2598>) 和这里 (<http://review.android.git.linaro.org/#change, 2554>) 找到。

AOSP 官方不提供以太网连接支持的原因很容易理解：它在 Google 推动的任何类型的 Android 设备中都不是常见的连接方式。如果 Android 将来针对其他类型的设备了，那这个情况可能会发生变化。

# 片上系统中有什么？

到现在为止，我们都是把 SoC 当成一个黑箱子。现在，我们来看一眼它内部是什么样。图 5-2 是一个典型 SoC 内部的结构。

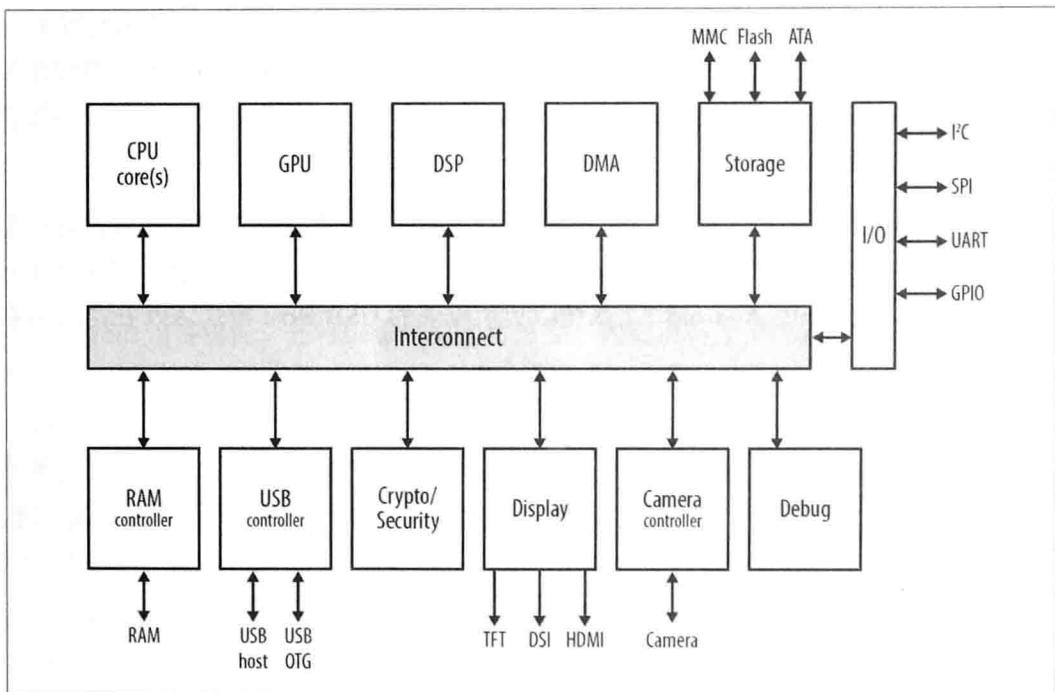


图 5-2：典型的片上系统（SoC）

正如你看到的，这里面除了 CPU 内核以外还有很多东西。SoC 可以看作是电路板的一种形式，它通过总线将各种组件连接到一起（通常称之为互联逻辑）。组件的数量和复杂度取决于 SoC 及其厂商。在这一点上，根本没有标准可言。不过市面上的大多数 SoC 虽然来自不同的厂商，但却包含了一系列相似的基本部件使得它们能够通用。正如在前面系统架构框图中所说的那样，这些组件的大多数可以被组合在一起，也可以被独立成更多的模块。所以，这里只是一个简化的视图。并且请注意，并非 SoC 中的所有组件都工作在同一个时钟频率下。所以打个比方，虽然可能说明书所说的 CPU 可以达到或者超过 1GHz，图形处理单元（GPU）很可能运行在几百兆赫兹而已。

---

**注意：** GPU 的时钟通常是由 CPU 的时钟分频而来。例如，如果 CPU 的时钟是 1GHz，GPU 可能就运行在 250MHz。虽然它们运行得很慢，但是 GPU 有大量的并行计算单元。即使 CPU 是双核的，GPU 却可能是 16 核或者 64 核。

---

表 5-1 列出了写这本书时用于 Android 系统的最流行 SoC。正如你所看到的，市场上越来越多的是双核 Android 设备，而且四核设备也已上市，厂商在飞速增加核的数量。虽然你的嵌入式 Android 系统并不需要这么强的计算能力，但是它却使得在不远的将来，多核 SoC 的成本也可以满足你的设计预算。

表 5-1：SoC 阵容

SoC	厂商	CPU	速度	GPU
OMAP3	德州仪器 (TI)	ARM Cortex A8	600MHz ~ 1.2GHz	PowerVR SGX530
OMAP4	TI	双核 ARM Cortex A9	1 ~ 1.8GHz	PowerVR SGX54x
OMAP5	TI	双核 ARM Cortex A15	2GHz	PowerVR SGX544
i.MX51	飞思卡尔	Cortex A8	800MHz	OpenGL ES 2.0 兼容 <sup>a</sup>
i.MX31	飞思卡尔	Cortex A8	1GHz	OpenGL ES 2.0 兼容
i.MX6	飞思卡尔	双 / 四核 Cortex A9	1GHz	OpenGL ES 2.0 兼容
Tegra 2	英伟达	双核 ARM Cortex A9	1 ~ 1.2GHz	GeForce
Tegra 3	英伟达	四核 ARM Cortex A9	1.3GHz	GeForce
骁龙 S2	高通	Scorpion <sup>b</sup>	800MHz ~ 1.5GHz	Adreno 205
骁龙 S3	高通	双核 Scorpion	1.2 ~ 1.5GHz	Adreno 220
骁龙 S4	高通	双核 Krait <sup>c</sup>	1 ~ 1.7GHz	Adreno 220 或 320
Exynos	三星	单 / 双核 Cortex A8	1 ~ 1.5GHz	PowerVR SGX540 或 ARM Mali-400
Exynos 4	三星	四核 Cortex A9	1.4 ~ 1.6GHz	ARM Mali-400 MP4
Exynos 5	三星	四核 Cortex A15	2GHz	ARM Mali-658
Atom	因特尔	单核 x86	1.6 ~ 2GHz	PowerVR SGX540
MT6575	联发科	单核 Cortex A9	1GHz	PowerVR Series5 SGX
MT6577	联发科	双核 Cortex A9	1GHz	PowerVR Series5 SGX

a. 在飞思卡尔的手册上没有关于它来源的更多信息。

b. 高通特有，据维基百科介绍，其跟 ARM Cortex A8 相似。

c. 高通特有，据维基百科介绍，其跟 ARM Cortex A15 相似。

Linux 内核从很早开始就已经支持对称多处理器了，所以对于支持多核 SoC 方面你不用太担心。作为手机的操作系统，Android 系统运行于多核处理器的时间并不长。而且，它也仅仅是受益于 Linux 的多核处理能力，其框架本身并没有任何的多核优化。所以，如果你的代码必须并行运行于多个处理器上，你可能需要手动对线程的 CPU 类似的属性进行恰当的设置。

传统意义上，Android 主要用在基于 ARM 的 SoC 上，就如上表所反映的那样。但是我们也很早就注意到，它已经被运行于一系列支持 Linux 的体系结构上，例如 x86、MIPS、SuperH 以及 PowerPC。实际上，很多企业已经开始销售基于 Intel 处理器的设备，例如摩托罗拉和联想。Google 和 Intel 实际上已经展开了合作，以推动 x86 体系结构进入主流的 AOSP。但是，现在在网络上看到的大量工具、文档仍然是以 ARM 为主的。

在 SoC 里面另一个重要的组件是 GPU，负责设备显示的图形加速功能。虽然大多数 Android 的 SoC 都是基于 ARM 核心的，但是对于 GPU 来说 SoC 厂商并没有什么标准的 GPU 可供选择。相反，正如表 5-1 所看到的，每家厂商都使用了不一样的 GPU。如前所述，即使跟 CPU 内核在一顆晶原上，虽然 CPU 运行速率达到或超过 1GHz，通常 GPU 运行频率在数百兆赫兹（300 ~ 500MHz）这样的速度上。

除了 CPU 和 GPU 以外的其他大部分 SoC 组件的角色就可以比较容易讲清楚了：

#### ***RAM 控制器***

与板载 RAM 的接口。

#### ***DMA 控制器***

自动处理 RAM 与内存映射硬件之间的数据传输。

#### ***USB 控制器***

实现硬件部分的设备 USB 连接管理。

#### ***DSP***

为 JPEG 编码等数字信号处理提供硬件加速。

#### ***显示控制器***

使得 SoC 能够驱动各种类型的显示设备。

#### ***摄像头控制器***

用于 SoC 与摄像头之间的接口。

#### ***存储控制器***

管理用于 SoC 的各种类型存储设备的 IO。

#### ***Debug***

使得 SoC 能够通过各种机制连接到硬件调试工具，例如 JTAG。

SoC 还可能包含一些加密和安全功能。这可能只是加密功能的简单硬件加速，也可能是 SoC 厂商提供给设备制造商用于锁定设备和用于防止未经授权的代码执行的安全机制。这种机制经常被用来实现数字版权管理（DRM），这可以让那些想重新刷机的人

们感到非常沮丧。然而不幸的是，消费者并不是 SoC 制造商的直接客户，并且使用这种技术的道德问题远远超过本书的范畴。

最后，SoC 很可能还包括连接其他外部 IC 的各种总线和接口。例如，这就是为什么前面描述的器件可以通过 PCB 上的导线与 SoC 连接的原因。这些总线和接口可能包括 I<sup>2</sup>C、SPI、UART 和 GPIO，但还可能包括其他一些机制。

SoC 厂商会在向设备厂商、OS 和设备驱动开发者提供的技术手册中对于每一个具体的 SoC 的功能和组件进行描述。而且 SoC 厂商也会对 SoC 片内组件提供一系列的驱动程序，例如 GPU。大部分的 SoC 厂商倾向于更进一步地为它们即插即用的芯片评估版提供 AOSP 源代码树。

## 内存布局与映射

为了让前面提到的硬件能够有用，它们必须提供一种软件操作的方法。大多数情况下，软件是通过 Linux 内核的设备驱动程序来进行访问的。应用程序通过这些驱动程序提供的标准接口与底层硬件进行通信。图 5-3 给出了一个示意图。

对于任何 CPU 来说，其连接的总线至少有一条是地址总线。连接这个总线使得 CPU 可以通过不同的地址范围来访问连接到 CPU 上的不同组件。实际上，大多数的组件会占据多个，而且经常是连续的地址区域。CPU 实际访问外设的地址通常又被称为物理地址，因为其表达的是实际的物理单元。当 CPU 使用任何地址时，实际上在印制板（PCB）上的 CPU 地址总线上就会产生实际的电子信号，特定的 IC 器件通过这些电子信号来识别和响应 CPU 的请求。

每个元件在物理地址空间的实际位置被叫做物理地址映射。这个映射由设备的开发者在设计 PCB 时通过 SoC 与器件之间的布线关系来决定。两块器件完全一样的电路板可以有完全不一样的物理内存映射。而重要的是每个驱动程序需要知道其通信的设备地址。有些时候，器件与驱动程序自己通过实际总线进行通信，这时候会有一个器件作为总线桥使得其他器件可以通过该总线连接到 SoC，例如 I<sup>2</sup>C 总线就是这样的例子。

---

**注意：**如果你想在运行时看看内核看到的物理地址映射，只需要在命令行里面输入 `cat /proc/iomem`。这个映射可能不会包括所有实际板上的外设，但是它包括了内核看见的所有内容。有些 IC 和外设没有列出来的原因可能是驱动程序没有注册或处理它们。

---

CPU 通过内存管理单元（Memory Management Unit, MMU）管理了两个完全独立的地址空间。通过 MMU，CPU 可以使用虚拟地址空间运行应用程序，而与通过地址总线连接的外设仍然通过物理地址进行通信。

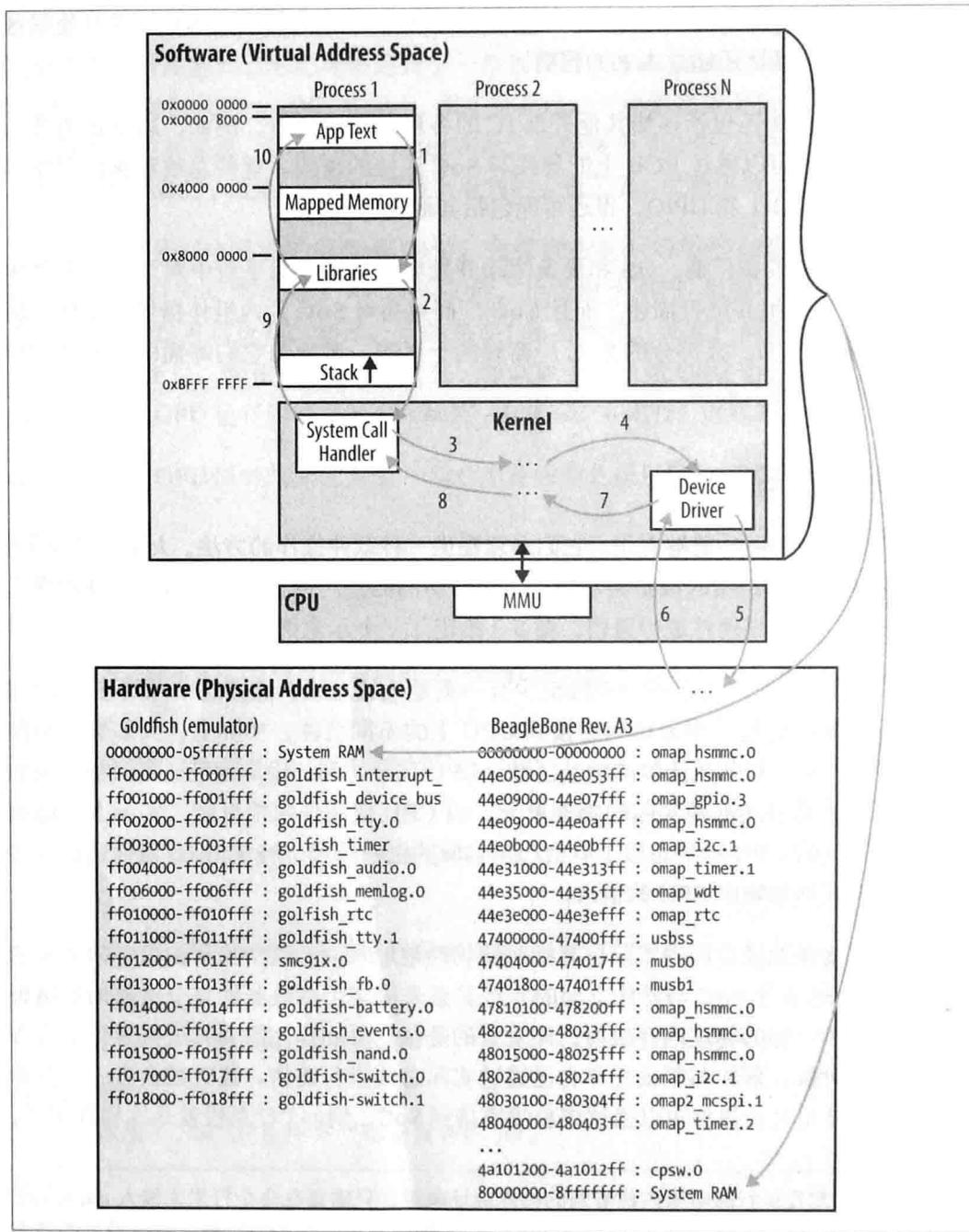


图 5-3：虚拟地址与物理地址空间的对比

位于物理地址空间的其中一个组件就是系统 RAM。正如你在图 5-3 中所看到的，RAM 在物理地址空间中位置差异很大。很显然，RAM 用于存储所有活动的软件代码和数据。然而，这些代码和数据几乎不使用实际地址空间位置。相反，OS 跟 MMU

一起实现了一个虚拟地址空间，使得每个进程看到几乎相同的地址环境。虚拟地址最终映射到实际物理地址，不过这个变换是由 MMU 基于 OS 维护的页表自动执行的。

对分页技术和 MMU 操作进行完整地解释就超出了本书的范畴，但是记住一点就可以：你在应用程序里看到的地址范围与 CPU 访问代码和地址在总线上给出的实际地址几乎没有固定关系。图 5-3 给出了 Android 进程所在的虚拟地址空间，注意这个布局不是成比例的。有些对象可能比它们显示得更大或者更小。内核通常占据以 0xC000 0000 起始的高地址，相反，Android 应用则占据 0xC000 0000 以下的所有地址。

实际应用程序的“text”段，也就是应用程序的代码，在虚拟地址空间开始的位置附近。然后是映射的内存区域，这些区域要么指向进程间通信的共享 RAM，要么是通过驱动程序的 mmap 函数映射到进程地址空间的物理地址区域。

将物理地址区域映射为一个进程的地址空间使得进程可以直接驱动 IC 器件或者其他设备，而无需每个操作都要经由内核或驱动程序来执行。这对图形显示等性能敏感的操作非常重要。而且它也是一个保密驱动程序的知识产权的有效方法，因为可以将其从内核的 GPL 要求中脱离出来。实际上，这是在 Android 的 HAL 层实现核心驱动程序代码的有效方式。

最后，库函数地址从 0x8000 0000 开始，而进程的栈空间从进程的最高位置开始向下递增。除非你的软件使用内存映射寄存器和映射区域来操作硬件设备，否则你的程序调用硬件的路径应该是这样的：

1. 你的代码调用一个函数，该函数与硬件相关的设备描述符相交互。被调用的中间代码实际上是被映射到你进程地址空间的某一个系统库文件。这个函数通常是一些内核系统调用的包装。
2. 库函数做一些基本处理并最后调用相应的系统调用。
3. 系统调用句柄做进一步处理，然后调用内核中多个功能。
4. 最终内核的某个部分调用与你的应用程序持有的文件描述符相匹配的相应设备的驱动程序。
5. 设备驱动程序通过某种方法与硬件进行交互，其结果当然是硬件依赖的。某些情况下，设备驱动程序能够读回来一个设备状态并立刻返回。而另一些情况下，硬件只能在一段时间之后才能反馈。当然，还有一些情况根本就没有任何反馈信息。
6. 假设硬件确实提供反馈信息给驱动程序或者产生一个中断来通知之前操作的状态，调用路径就会返回到它被调用的地方。

7. 调用路径从驱动程序返回到被调用的地方。
8. 调用路径返回到系统调用句柄。
9. 调用路径返回到系统函数库。
10. 调用路径返回到你的代码。

在这个调用路径中，涉及物理地址的只有在设备驱动程序与指定的硬件设备通信的时候，其他所有调用和数据交换都是采用虚拟地址空间进行的。

## 建立开发环境

只要你有一定的原型硬件，并准备将它运行起来并进行开发，最实际的方式是让你的目标板连接到你的工作机上。图 5-4 示意了一个通用的主机—目标机调试环境。你的实际连接可能会有所差异，但是这个图代表了一种理想情况。

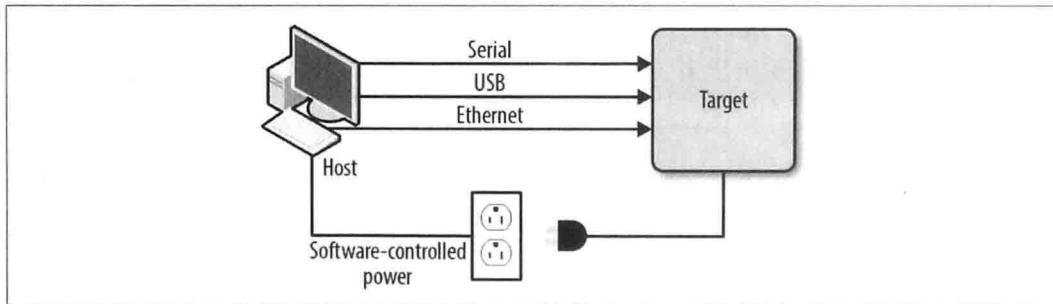


图 5-4：主机—目标机调试环境

在这里，主机与目标机之间的通信有时起到了好几个作用。通过将电源接到主机软件控制的电源上，主机可以通过脚本控制目标机的开机和关机，从而可以对板上的多个软件版本进行测试。市场上有多个脚本可以让你建立这样的环境。

目标机与主机最经典的连接方式是通过串口连接，通常是 RS-232。它可以使得你跟板上的 bootloader 进行交互，上传和下载小的文件，作为其他手段不能用时的通用交互手段。显然，这种连接非常缓慢，所以它也主要是用于基本的交互；大数据传输还是适合用像以太网这样的方式。

以太网连接会允许主机向目标板提供大量的服务，如图 5-5 所示。例如，为了简化调试过程，最好的方式是让目标板通过 DHCP 来取得它的 IP 配置，使用 TFTP 来装载内核映像，并通过 NFS 来挂载其根文件系统。如果你这样做，你在主机上对你的工程所做的任何修改最差也会在目标机复位时被部署进去。而最好的情况是，你修改了

NFS 挂载的根文件系统中的文件，所有你需要做的就仅仅是重启命令行并运行这个新的版本。在大多数情况下，你可以避免每次修改时都要对目标机的存储进行重新刷机。

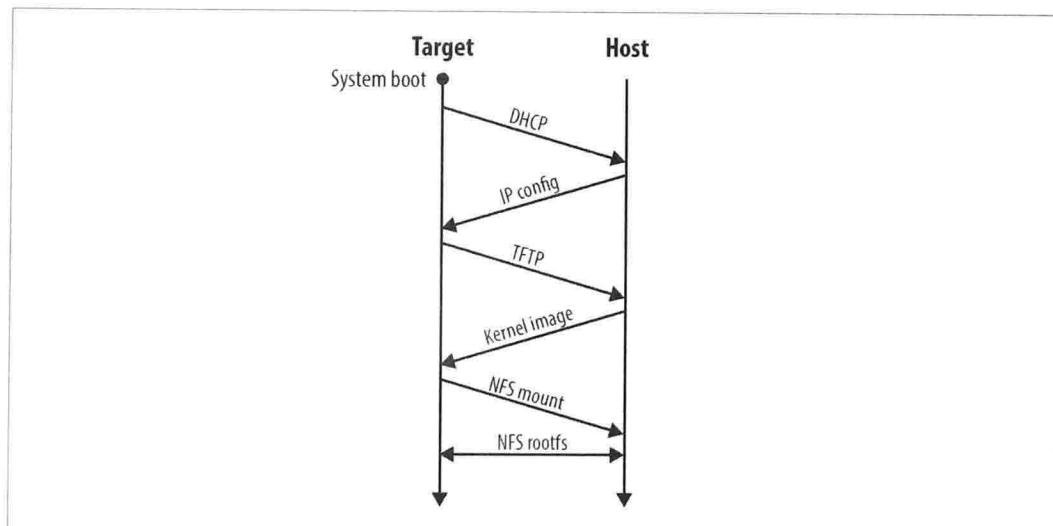


图 5-5：开发的启动安装

最后，特别是对于 Android 系统来说，USB 会非常有用。事实上，在 Android 系统中，你可以依靠 USB 来连接到消费者的手机或者平板电脑上进行应用程序调试。可以使用所有的 ADB 命令，包括目标机 Shell、端口转发以及上传文件等。你可以基于 IP 来配置 ADB，也就是基于以太网，不过采用 USB 是一个更好的选择，因为它基本上是“即插即用”的。

---

**注意：**设置基于 IP 的 ADB 实际上很简单：如果它现在基于 USB，那么你只需要很少一些命令来达到目的。大多数情况下，USB 是网络上文档讲得最多的形式。我们会在下一章中进一步阐述。

---

你的具体设置将最有可能包含自己的特点，这里给一个配置的总体思路。串口支持通常是由引导程序和内核提供的。除非你的板子是基于一个全新的 CPU，否则你应该已经可以通过串行端口进行通信了。支持以太网将需要一个正确的驱动程序来驱动以太网芯片，这可能需要你做一些工作。最后，USB 支持将取决于内核是否支持你的 USB 硬件。如果使用的是常见的 SoC，这应该不是一个问题。如果需要帮助来为你的目标板建立一个 DHCP、TFTP 或 NFS 服务器，可以看看 O'Reilly 出版的《构建嵌入式 Linux 系统（第二版）》。

# 评估板

如果你仍然处于开发进程的早期状态，或者你仅仅想简单评估 Android 系统，你很可能希望依赖一块开发板。这里有一些因素在你选择的时候需要考虑：

## SoC

评估板的 SoC 是否跟你最后设计使用的一样？或者是一个系列的？或者是厂商刚刚发布的新处理器的前一代芯片。

## 社区

是否围绕这个板子有一个现成的社区，或者厂商是否是该板子的唯一技术支持？社区是否活跃？这个社区是否支持该板子的一个系列？

## 成本

什么是板子的前期成本，该价格包含些什么内容？有多少附加或扩展成本？低端的版本和高端版本之间的价格相差多少，有哪些功能上的差异？

## 特性

板子包含 / 显露哪些功能？SoC 可以支持越来越广的功能。然而，板子支持的 SoC 功能越多，它往往就越昂贵。因此，需要检查下你所看的板子有没有显露你所需要的功能。

## 可扩展性

如果板子提供的基本特性可能满足你想要完成的一部分功能，那么板子是否允许你增加扩展板以达到你想完成的最终功能。

## 可获取性

板子到手容易吗？虽然有些板子在纸上看起来很不错，但是供应商却很不稳定。

## 授权

你能使用这个板子作为最终产品吗？有些厂商会禁止你这么做。你能拿到 BOM 清单或者原理图吗？如果你想基于评估板来制作一块电路板，这些就很重要。

## 配件目录

板子是否有配件目录？有些板子没有配件目录，你就需要自己找到相应的供应商。通常情况下，厂商只会将其器件销往大批量买家，这样一些部件就让小项目没办法采购。

## 第三方元器件

有些 SoC 厂商在板上包含了一些第三方元器件。你要确保你的相应器件有类似的参数。需要记住一点，如果在你的设计中使用了第三方的器件，你就需要从这些供应商那里拿到跟他们对 SoC 厂商一样的技术支持。

## 软件支持

板子对 Android 的支持怎么样？由谁来提供支持？是厂商？还是第三方？它支持 Android 的哪个版本？这些支持的长期承诺是什么？

你肯定还可以有关于你的项目的更多条件。然而，如果你在制作你自己的硬件，你的起点通常还会是 SoC，因为无论从硬件角度还是软件角度来说这都是一个关键决策点，它涉及你公司的一些利益相关者。下一步就是去 SoC 厂商的网站去看一下它为这个 SoC 推荐的评估板。如果你只是想到手一块不错的板子用于做一些 Android 系统的试验，你最好是通过你常用的搜索引擎花几个小时找几块板子看看。另外，也可以看看表 5-2 中所列的早期最流行的一些板子。

表 5-2：评估板阵容

板子	SoC	速度	RAM	IO	价格 <sup>a</sup>
BeagleBone	Sitara AM3358	500MHz (USB 供电) / 720MHz (DC 供电)	256MB	USB OTG, USB host, Ethernet, 板载串口, 板载 JTAG, 扩展座, microSD	\$89
BeagleBoard xM	Davinci DM3730	1GHz	512MB	USB OTG, USB host, Ethernet, 串口, JTAG, 扩展座, microSD, DVI-D, LCD 座子, S-Video, camera 座子, stereo in/out	\$149
iMX53 Quick Start Board	i.MX53	1GHz	1GB	USB OTG, USB host, Ethernet, serial, JTAG, 扩展座, SD, microSD, SATA, VGA, LCD 座, stereo in/out	\$149
PandaBoard ES	OMAP4 四核	1.2GHz	1GB	USB OTG, USB host, Ethernet, WLAN, 蓝牙, 串口, JTAG, 扩展座, SD, HDMI, DVI, LCD 座, camera 座, stereo in/out	\$182
AM335x Starter Kit	Sitara AM3358	720MHz	256MB	USB OTG, USB host, Ethernet, WLAN, 蓝牙, onboard serial, 板载 JTAG, 扩展座, microSD, 电容触摸 LCD, 加速度计, stereo out	\$199

表 5-2：评估板阵容（续）

板子	SoC	速度	RAM	IO	价格 <sup>a</sup>
Nitrogen6X	i.MX6 双核	1GHz	1GHz	USB OTG, USB host, 串口, JTAG, SATA, SD, CAN, LCD 座	\$199
OrigenBoard	Exynos 4210 双核	1.2GHz	1GB	USB OTG, USB host, WLAN, 蓝牙, 串口, JTAG, SD, HDMI, LCD 座, 摄像头座, stereo in/out	\$199
Origen 4 Quad	Exynos 4 四核	1.4GHz	1GB	USB OTG, USB host, Ethernet, SD, JTAG, 串口, HDMI, 板载 LCD 座, audio	\$199
DragonBoard APQ8060A	枭龙双核	1.2GHz	1GB	USB OTG, USB host, Ethernet, WLAN, 蓝牙, GPS, FM 收音机, 加速度计, 陀螺仪, 指南针, 磁力计, 压力传感器, eMMC, SATA HDMI, 摄像头, stereo out, 串口, 电容触摸 LCD, JTAG	\$499
SABRE	i.MX53	1GHz	1GB	USB OTG, USB host, Ethernet, WLAN, Bluetooth, GPS, ZigBee, 加速度计, 光传感器, 串口, JTAG, eMMC, SD, SATA, NOR flash, VGA, HDMI, LCD 面板, 摄像头, stereo in/out	\$999
Snapdragon MDP	枭龙 S4 双核	1.5GHz	1GB	USB OTG, WLAN, Bluetooth, GPS, 加速度计, 陀螺仪, 指南针, 接近传感器, 温度传感器, SD, HDMI, LCD panel, camera, stereo out	\$999

a. 写作本书时的常见价格。

除了最后两个板子以外，表 5-2 中列出的其他评估板名副其实：一块贴了芯片的 PCB 以及一些光秃秃的连接器在上面。很少一部分还包括了 LCD 面板，不过大多数板子可以插入一个液晶面板，价格大约 100 ~ 200 美元。列出的最后两个评估板是以平板

或者手机形式提供的，满足其对应的外观和机械要求。如果你试图想做一个最终产品的演示设备并展示给最终客户看，这两个板子很可能就比一堆飞线要好看得多。当然，正如你可能已经注意到的，它们的价格也就更高一些。

# 本地用户空间

到这个时候，你可能已经自己动手尝试了一些这样或那样的操作，或者你非常渴望使用一个真正的 Android 系统了。正如你所接触过的任何其他嵌入式系统一样，你的典型目标就是获得一个有最基本功能的系统，然后开始添加越来越多的硬件和功能，直到满足你的需求。

显然，要获得一个具有最基本功能的 Android 系统，你首先需要把内核下载到你的板子上。正如我之前提到的那样，想要获得一个你自己的兼容 Android 的内核的最好方法就是跟你的 SoC 供应商索要，内核移植和开发板启用的一些内容不包括在本书的范围之内。然而，一旦你已经有了一个自己的具有最基本功能的内核，首先你必须要处理的 Android 组件就是它的本地用户空间。

如第 2 章所说的，这些基础服务为 Android 栈的所有上层，包括 Dalvik 虚拟机以及它所运行的服务和应用程序提供环境。它也是 Android 的硬件支持的一部分实现。因此，现在我们可以来仔细看看 Android 的本地用户环境了。毫无疑问，对于这部分的学习完全不同于最经典的嵌入式 Linux 系统，它值得单独讨论。

## 文件系统

在第 4 章中，我们讨论了构建系统是如何运行的以及它生成了什么。具体来说，表 4-3 提供了通常由编译系统创建的镜像的详细清单。相反，图 6-1 说明了这些镜像彼此之间在运行时是如何相互关联的。除了我们将在后面讨论的少数例外之外，文件系统的布局在版本 2.3/ 姜饼和版本 4.2/ 果冻豆中本质上是相同的。

想要了解我们如何从构建系统生成的镜像到图 6-1 显示的运行时配置的，你需要回到第 2 章中关于系统启动部分的解释，更具体地说，你需要参考图 2-6 所示的引导过程启动的流程图。从本质上来说，内核挂载到由构建系统生成的 RAM 磁盘镜像上，该镜像作为它的根文件系统，并且启动该镜像中的初始化进程。这个初始化进程的配置文件，我们将在本章的后面部分介绍，将使得一些额外的镜像和虚拟文件系统被安装到现有的根文件系统的目录条目中。

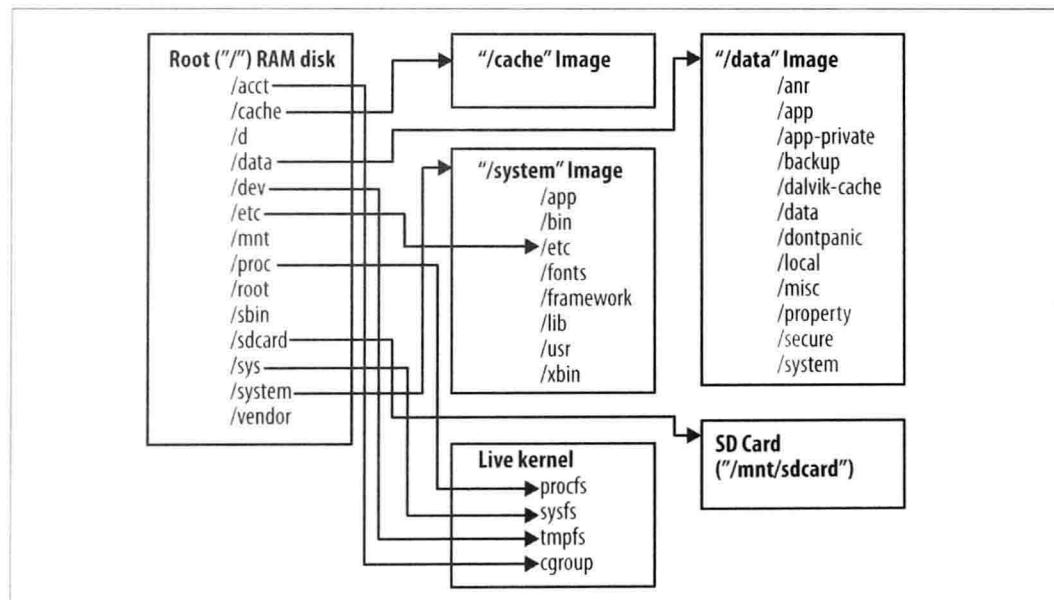


图 6-1：Android 根文件系统

你要问的第一个问题可能就是，“为什么有这么多的文件系统？”实际上就是说，为什么不仅仅用一个文件系统镜像来存储所有的东西呢？这个问题的答案在于每一种镜像有不同的作用，伴随着存储设备或被使用的技术的本质差异。例如 RAM 磁盘镜像，它尽可能的小，其唯一作用就是提供系统所需的初始架构。它通常存储为一个压缩的镜像，在被内核加载到 RAM 里然后被挂载为一个只读的根文件系统之前，通常存储在一些媒介中。

从另一方面来说，/cache、/data 和 /system 通常被从不同的分区挂载在实际的存储介质中。通常 /cache 和 /data 会被挂载为读-写权限的，而 /system 会被挂载为只读的。

## 使用单一的文件系统

没有什么能够阻止你使用单个文件系统，用于所有 Android 的构建输出而不是使用独立存储分区。例如，Texas Instruments 公司的 RowBoat 就完全是这样做的。它生成一个单独的根文件系统镜像，这是用于在目标存储设备上进行编程使用。例如说，对于 BeagleBone 或 BeagleBoard，在其整体程序中的根文件系统，被编入 microSD 卡的一个单独的分区中，用于引导启动以及作为主要设备的存储设备。

但是，通过整合为一个单一的文件系统，你会认为你可以一下子一起更新文件系统。总之，创建一个用于你的系统的故障安全更新程序是很难做到的。在 RowBoat 对 Beagles 的支持的情况下，这可能不是一个问题，因为它们都是开发板，但在你的实际产品中必须涉及这一个领域，这可能就会成为一个问题。

在 Android 的 2.2 版本及之前的版本中，所有三个目录通常被安装在 YAFFS2 格式的 NAND 闪存分区。由于手机制造商已经开始慢慢转向 eMMC 闪存，用来代替 NAND，在 Google 的 Android2.3 的先进设备三星的 Nexus S 中，YAFFS2 已经被 ext4 替换了。从那时起，它被假定所有基于 Android 的手机应该使用 ext4 来替代 YAFFS2。但是，没有任何事情阻止你使用另一种文件系统类型。你只需要修改编译系统的 makefile 文件来生成这些镜像并更新，使用 mount 命令来更新参数，这个命令是 init 配置文件的一部分，我们很快就会看到。

## eMMC VS NOR 或者 NAND 闪存

正如《构建嵌入式 Linux 系统》这本书中介绍的那样，Linux 的 MTD 层用于管理、操作和在 Linux 下访问闪存设备，它包括 NOR 和 NAND 闪存。然而在 MTD 层的顶部使用的不同的文件系统，比如说 JFFS2、UBIFS 或 YAFFS2，使得闪存设备或分区作为 Linux 的虚拟文件系统交换器（VFS.）的一部分是可访问的。这些闪存的文件系统通常实现损耗均衡和坏块管理，妥善处理底层的闪存设备。

如第 5 章所述，一个 eMMC 的设备是一个传统的块设备。从本质上讲，它包含一个微控制器和一些内存，允许它做必要的损耗均衡和透明的坏块管理。因此，操作系统可以使用常规的磁盘文件系统，比如说 EXT4。显然现在的趋势是使用 eMMC，根据 Android 开发者 Brian Swetland 所说，能够促进减少 PCB 上的引脚，因此整体成本得到了减少，使用这种类型的设备也有一些额外的好处。

首先，它可以让你使用所有你已经习惯了在常规的 Linux 文件系统中使用的传统的命令和方法。MTD 子系统功能尽管十分强大，在人们可以有效地使用它之前

也总是需要一些时间来适应。此外，闪存文件系统往往倾向于被设计成单处理器系统，这是因为在 Linux 中，多处理器系统都不得不花相当长的一段时间来处理磁盘文件系统。因此，它们很可能会更适合多核 Android 设备未来的需要。

SD 卡总是显示为一个块设备，且通常有一个 VFAT 文件系统在里面。这应该是预料之中的，因为用户需要能够将它从 Android 设备中取出，然后能够插入到自己的普通电脑上，无论电脑正在运行的操作系统是什么。*/proc*、*/sys* 和 */acct* 能够分别使用 procfs、sysfs 和 cgroupfs 来挂载。但是在传统的基于 Linux 的系统中，*/proc* 和 */sys* 目录被挂载在相同的位置，而 cgroup 通常在 Linux 中被挂载为 */cgroup*，而在 Android 中被挂载为 */acct*。还要注意的是，*/dev* 被挂载为 tmpfs。这意味着它的内容是动态创建的，并且不驻留在任何的永久存储中。这很好，因为 Android 依赖于 Linux 上的 udev 的机制，来在 */dev* 中动态创建入口表示设备被插入和（或）驱动程序被加载或初始化。

procfs、sysfs、tmpfs 和 cgroup 都是由系统中当前运行的内核所维护的虚拟文件系统。它们不具有任何相应的存储，实际上，其数据结构是在内核中维护的。procfs 是内核输出与自己相关信息给用户空间的传统方式。通常情况下，procfs 的入口被看作是文本文件，或包含文本文件的目录，它可以被转存到命令行中，用于从内核提取给定的信息。例如说，如果你正在寻找你所运行的系统的 CPU 类型，你可以转存 */proc/cpuinfo* 文件中的内容。

tmpfs 允许你创建一个虚拟的 RAM-only 的文件系统，仅用于存储临时文件。只要电源被应用到 RAM 中，内核就会允许你读取和写入这些文件。但是在重新启动后，这一切都消失了。cgroupfs 是除了内核外一个相对较新的用于管理对照组的功能，它在 Linux 的 2.6.24 版本中添加。总之，cgroup 允许你将特定的进程和它们的子进程进行分组，以及分配资源限制和优先权到这些组中。Android 使用 cgroup 来优先前台任务。

## 根目录

正如我们在第 2 章所说，文件系统目录标准（Filesystem Hierarchy Standard, FHS）指定了一个 Linux 根文件系统的经典结构。显然 Android 的文件系统是不会受这个标准所束缚，相反，它在很大程度上依赖于 */system* 和 */data* 目录来实现它的大部分关键功能。

Android 的根目录是从 *ramdisk.img* 挂载（mount）的，这个文件由 AOSP 构建系统产生。通常，*ramdisk.img* 将与内核一起存储在设备的主存储中，由引导程序在系统启动时加载。表 6-1 详细介绍了装载后的根目录内容。

表 6-1：Android 的根目录

名称	类型	描述
<code>/acct</code>	目录	cgroup 挂载点
<code>/cache</code>	目录	非关键数据以及正在下载中文件存放用的临时目录
<code>/d</code>	符号链接	指向 <code>/sys/kernel/debug</code> , 即 debugfs 的常用挂载点 <sup>a</sup>
<code>/data</code>	目录	<code>data</code> 分区的挂载点, 通常来说 <code>userdata.img</code> 的内容会挂载在这个地方
<code>/dev</code>	目录	<code>tmpfs</code> 文件系统, 包含了 Android 需要用到的各种设备节点
<code>/etc</code>	符号链接	指向 <code>/system/etc</code>
<code>/mnt</code>	目录	用作临时挂载点
<code>/proc</code>	目录	<code>procfs</code> 的挂载点
<code>/root</code>	目录	在传统的 Linux 系统中, 该目录用于 <code>root</code> 用户的根目录。在 Android 系统中, 这基本上就是一个空目录
<code>/sbin</code>	目录	在 Linux 系统中, 这个目录保存了系统管理员需要用到的各种关键工具。而在 Android 系统中, 它只给出了 <code>ueventd</code> 和 <code>adb</code>
<code>/sdcard</code>	目录	SD 卡的挂载点
<code>/sys</code>	目录	<code>sysfs</code> 文件系统的挂载点
<code>/system</code>	目录	<code>system</code> 分区的挂载点, <code>system.img</code> 就挂载在这里
<code>/vendor</code>	符号链接	一般来说就是指向 <code>/system/vendor</code> 的符号链接, 不是所有的设备都有这个目录
<code>/init</code>	文件	内核初始化完成时时调用的二进制程序
<code>/init.rc</code>	文件	<code>/init</code> 的主要配置文件
<code>/init.&lt;device_name&gt;.rc</code>	文件	<code>/init</code> 的板级配置文件
<code>/ueventd.rc</code>	文件	<code>ueventd</code> 的主要配置文件
<code>/ueventd.&lt;device_name&gt;.rc</code>	文件	<code>ueventd</code> 的板级配置文件
<code>/default.prop</code>	文件	系统使用的默认全局属性, 它会被 <code>init</code> 进程在启动时自动装载

a. `debugfs` 是一个基于 RAM 的很灵活的文件系统, 用于导出从内核空间到用户空间的所有调试信息, 一般量产设备中不存在这个目录。

作为版本 4.2/ 果冻豆的一部分, 你将会发现更多根文件系统的条目, 如表 6-2 所列。

表 6-2：4.2/ 果冻豆系统根目录中额外的项目

项目名称	类型	描述
/config	目录	configfs 的挂载点 <sup>a</sup>
/storage	目录	从 4.2/ 果冻豆系统开始，这个目录用于挂载外部存储设备。例如，/storage/sdcard0 是一个假的 <sup>b</sup> 外部存储，而 /storage/sdcard1 是第一个真实的外部 SD 卡
/charger	文件	一个显示充电状态的全屏的本地二进制应用
/res	目录	/charger 用到的资源文件

a. 有关 configfs 的更多信息请查看 <http://lwn.net/Articles/148973/>。

b. “假的”含义是该存储实际上是一个 FUSE 挂载的内部目录，但是在文件系统中显得像是外部存储设备。

## /system

正如前面提到的，/system 包含了 AOSP 编译系统产生的只读内容。为了进一步说明这一点，图 6-2 把第 2 章介绍 Android 体系结构图再次拿来，并且加上了各自在文件系统中位置。

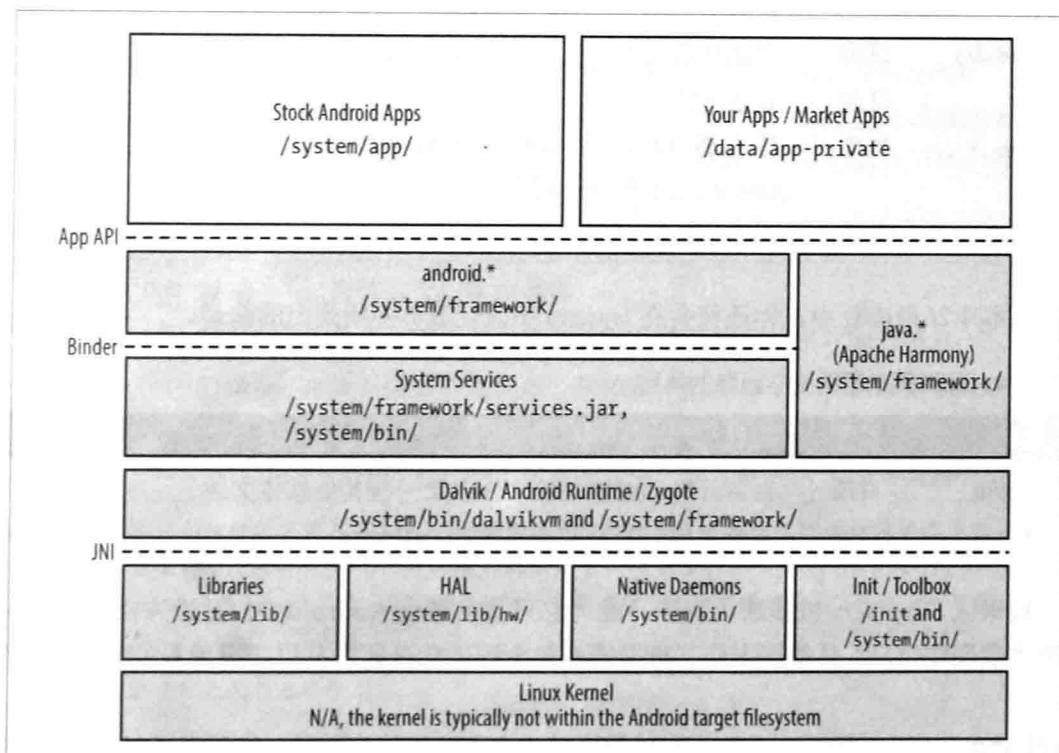


图 6-2：关键 Android 部件在文件系统中的位置

如你所见，一旦 *system.img* 挂载好，大部分的部件都可以在 */system* 下找到。表 6-3 进一步详细描述了每个条目。也可以对比图 6-2 和图 3-2，看看每个组件在 AOSP 源代码中的位置与最终的文件系统中位置的差异。

表 6-3：*/system* 目录下的内容

名称	类型	描述
<i>/app</i>	目录	AOSP 中的 app，例如浏览器、email、日历等。所有采用 BUILD_PACKAGE 生成的都在这里
<i>/bin</i>	目录	所有 AOSP 编译的本地二进制程序和守护进程都在这里，即采用 BUILD_EXECUTABLE 生成的都在这里。不过 <i>abdb</i> 是一个例外，它的 LOCAL_MODULE_PATH 设置为 <i>/sbin</i> ，所以就安装到了 <i>/sbin</i> 目录
<i>/etc</i>	目录	保存守护进程和工具的配置文件，可能包括用于某个 <i>init</i> 配置文件启动时装载的 <i>init.&lt;device_name&gt;.sh</i> 脚本
<i>/fonts</i>	目录	Android 用到的字体
<i>/framework</i>	目录	框架的 <i>.jar</i> 文件
<i>/lib</i>	目录	系统的本地二进制库文件，主要包括采用 BUILD_SHARED_LIBRARY 编译的组件。这里需要再次强调，Android 不使用 <i>/lib</i> 目录，而只是用 <i>/system</i> 下的 <i>lib</i> 目录
<i>/modules</i>	目录	可选目录，用于存放内核的可装载模块
<i>/usr</i>	目录	存放传统 Linux 系统中经典 <i>/usr</i> 目录的最小集合
<i>/sbin</i>	目录	保存跟 AOSP 一起编译的额外的非关键二进制程序，例如 <i>strace</i> 、 <i>ssh</i> 和 <i>sqlite3</i> 等
<i>/build.prop</i>	目录	AOSP 构建过程产生的一系列属性，这些属性会在 <i>init</i> 启动时装载

在版本 4.2/ 果冻豆中，你还将会在 */system* 中发现表 6-4 所列出的条目。

表 6-4：4.2/ 果冻豆系统中新增的条目

名称	类型	描述
<i>/media</i>	目录	存放开机动画相关的文件以及一些其他媒体文件
<i>/tts</i>	目录	存放文字—语音转换引擎相关的文件

一般来说，*/system* 被挂载为只读，因为它只会在整个 Android OS 更新为较新的版本时才会变化。OTA 更新脚本作二进制修补时会假定此分区自从出厂后就没发生过改变。

## */data*

正如前面讨论的，*/data* 包含了所有数据和应用程序，这些数据可以随着时间而改变。

例如，Google Play 下载的应用程序都在这里可以找到。由 AOSP 生成系统生成的 *userdata.img* 镜像文件其实大多是空的，所以此目录开始时包含很少的东西或甚至没有。当然，从系统一开始使用，此目录的内容就会发生改变，即使重新启动也不会使得这些改变消失。这就是为什么 */data* 目录通常以读 / 写模式挂载，主要用于持久性存储。表 6-5 显示了该目录的内容。

表 6-5：*/data* 目录内容

名称	类型	描述
<i>/anr</i>	目录	ANR 跟踪数据
<i>/app</i>	目录	app 默认的安装目录
<i>/app-private</i>	目录	采用 <i>Forward locking</i> <sup>a</sup> 的 app 安装在这里，这个机制已经被一个 API 所代替，这个 API 允许 APP 开发者检查该运行的 app 是否是从 Google Play 上取得的合法版本。对于该问题有兴趣的可以看看 app 开发者手册中关于应用程序授权的那一节
<i>/backup</i>	目录	用于系统服务 <i>BackupManager</i>
<i>/davik-cache</i>	目录	保存所有的 dex 文件的 JIT <sup>b</sup> 版本
<i>/data</i>	目录	这里为每一个安装的 app 分配了一个子目录，实际上它是每个 app 的 home 目录
<i>/dontpanic</i>	目录	上次崩溃时的输出（控制台或者线程），用于 <i>dumpstate</i>
<i>/local</i>	目录	shell 可写的目录。也就是说，任何通过 shell 登录设备的用户，例如用 <i>adb shell</i> ，可以复制任何包括二进制的文件到这个目录，在系统重启时这个目录的内容会保持不变
<i>/misc</i>	目录	用于 WiFi、蓝牙或者 VPN 等功能的杂项数据
<i>/property</i>	目录	包含持久化的系统属性
<i>/secure</i>	目录	如果设备采用加密文件系统，该目录存放用户帐户信息
<i>/system</i>	目录	存放系统级的数据库，例如账户数据库和已安装应用该列表
<i>/tombstones</i>	目录	当本地二进制程序崩溃时，一个叫做 <i>tombstone_NNNNN</i> ( <i>N</i> 是一个序列号) 的文件就会产生，该文件包含了该崩溃相关的信息

- a. 当一个独立的软件开发商在 Google Play 中发布它的应用时，可以选择版权保护选项的设置。该选项设置为 off 时，用户可以将该应用从设备中拷贝出来，而设置为 on 则不行。本质上，设置为 on 意味着 app 安装在 */data/app-private*，而设置 off 意味着安装在 */data/app* 目录。
- b. davik 有一个即时编译器能够将 .dex 中的字节代码编译为 CPU 的本地指令。这个动作只做一次，然后就缓存起来之后直接使用。

在版本 4.2/ 果冻豆中，你还会找到表 6-6 中列出的条目。

表 6-6：版本 4.2/ 果冻豆中新出现的 /data 目录条目

名称	类型	描述
/app-asec	目录	加密的 app
/drm	目录	DRM 加密的数据，以及用于 Forward locking 的控制文件
/radio	目录	射频系统的固件
/resource-cache	目录	app 资源的缓存目录
/user	目录	多用户系统中的用户相关数据

## 多用户支持

4.2/ 果冻豆系统中添加的最重要的特征之一是多用户支持。事实上，一些人认为这是一个分水岭，它将 Android 开放到了一些新的用例环境下。虽然它只在平板模式下可用，但是它允许多个用户共享同一个设备。具体来说，它意味着每个用户可以登录到设备后有自己独立帐户数据及其应用程序的数据。

为了达到这个目标，系统数据的存储机制略有修改。例如，/data/data 现在是设备的所有者（即“管理员”）应用程序数据的目录，而所有其他用户的数据存储在 /data/user/<user\_id> 中。以下是版本 4.2/ 果冻豆仿真器<sup>注1</sup> 下的 /data/user 目录中的内容。

```
root@android:/ # ls -l /data/user/
lrwxrwxrwx root      root 2012-11-30 20:46 0 -> /data/data/
drwxrwxr-x system   system 2012-12-04 23:38 10
root@android:/ # ls -l /data/user/0/
drwxr-x--x u0_a27  u0_a27          2012-11-30 20:46 com.android.backupconfirm
drwxr-x--x bluetooth bluetooth    2012-11-30 20:46 com.android.bluetooth
drwxr-x--x u0_a17  u0_a17          2012-12-14 18:01 com.android.browser
drwxr-x--x u0_a43  u0_a43          2012-11-30 20:46 com.android.calculator2
drwxr-x--x u0_a20  u0_a20          2012-11-30 20:47 com.android.calendar
drwxr-x--x u0_a33  u0_a33          2012-11-30 20:46 com.android.certinstaller
drwxr-x--x u0_a0   u0_a0           2012-11-30 20:47 com.android.contacts
drwxr-x--x u0_a25  u0_a25          2012-11-30 20:46 com.android.defcontainer
drwxr-x--x u0_a6   u0_a6           2012-11-30 20:47 com.android.deskclock
...
root@android:/ # ls -l /data/user/10/
drwxr-x--x u10_system u10_system    2012-12-04 23:38 android
drwxr-x--x u10_a27  u10_a27        2012-12-04 23:38 com.android.backupconfirm
drwxr-x--x u10_bluetooth u10_bluetooth 2012-12-04 23:38 com.android.bluetooth
drwxr-x--x u10_a17  u10_a17        2012-12-04 23:38 com.android.browser
drwxr-x--x u10_a43  u10_a43        2012-12-04 23:38 com.android.calculator2
drwxr-x--x u10_a20  u10_a20        2012-12-04 23:38 com.android.calendar
drwxr-x--x u10_a33  u10_a33        2012-12-04 23:38 com.android.certinstaller
drwxr-x--x u10_a0   u10_a0         2012-12-04 23:38 com.android.contacts
drwxr-x--x u10_a25  u10_a25        2012-12-04 23:38 com.android.defcontainer
```

注 1：默认情况下模拟器是不支持多用户的，需要做一些 hack 工作才能添加一个虚拟的用户进去。

```
drwxr-x--x u10_a6 u10_a6          2012-12-04 23:38 com.android.deskclock  
...
```

同样的，现在可以为每个用户存储各自的互联网用户帐户凭据。在 4.2/ 果冻豆之前，系统中只有一个 *data/system/accounts.db* 文件来保存所有帐户的信息。现在每个用户都有了一个这样的文件：

```
root@android:/ # ls /data/system/users/ -l  
drwx----- system system 2013-01-19 01:03 0  
-rw----- system system 155 2012-11-30 20:46 0.xml  
drwx----- system system 2013-01-19 01:03 10  
-rw----- system system 166 2012-12-04 23:38 10.xml  
-rw----- system system 141 2013-01-19 01:03 userlist.xml  
root@android:/ # ls /data/system/users/0 -l  
-rw-rw---- system system 57344 2012-11-30 20:47 accounts.db  
-rw----- system system 8720 2012-11-30 20:47 accounts.db-journal  
-rw----- system system 534 2013-01-19 01:03 appwidgets.xml  
-rw-rw---- system system 549 2013-01-19 01:03 package-restrictions.xml  
-rw----- system system 97 2013-01-19 01:03 wallpaper_info.xml  
root@android:/ # ls /data/system/users/10 -l  
-rw-rw---- system system 57344 2012-12-04 23:39 accounts.db  
-rw----- system system 8720 2012-12-04 23:39 accounts.db-journal  
-rw-rw---- system system 129 2013-01-19 01:03 package-restrictions.xml
```

## SD 卡

正如前面所说，消费类设备通常有一个 microSD 卡，用户拔出来然后插到他电脑里。这个 SD 卡的内容对系统运行来说不是至关重要，事实上你可以移除它而且不会造成任何不良影响。但是，你需要了解一个真正用户使用设备时它上面存储了些什么内容，某些应用程序将它们的信息存储在 SD 卡，这些数据可能包含了用户需要的信息。表 6-7 详细介绍一些您可能会在 */sdcard* 目录中发现的内容。

表 6-7：*/sdcard* 目录内容的一个例子

名称	类型	说明
<i>/Alarm</i>	目录	下载的用于做响铃的音频文件
<i>/Android</i>	目录	包含应用程序“外部”数据和媒体文件。前者可能用于非关键文件和缓存，后者则是应用程序相关的媒体文件
<i>/DCIM</i>	目录	相机应用程序拍摄的照片和视频
<i>/Download</i>	目录	从 Web 下载的文件
<i>/Movie</i>	目录	下载的电影
<i>/Music</i>	目录	用户的音乐文件
<i>/Notification</i>	目录	下载的做通知铃声的文件

表 6-7: /sdcard 目录内容的一个例子 (续)

名称	类型	说明
/Pictures	目录	用户下载的图片
/Podcasts	目录	用户的播客
/Ringtones	目录	用户下载的电话铃声

因为 */sdcard* 是全局可写的，它的具体内容将取决于对该设备运行的应用程序，当然，还包括用户手动复制的内容。再一次说明，只是作为一个提醒，Android 的 API 区分了“内部”和“外部”存储，而 SD 卡是后者。此外，请注意一些升级程序在升级过程中使用 SD 卡作为更新镜像存储的位置。

## 构建系统和文件系统

第 4 章讲述了构建系统如何生成文件系统的各个部分。在这里，让我们进一步看看如何对构建系统进行控制来调整文件系统的生成。

### 构建模板和文件位置

在表 4-2 中，我们已经给出了可用的构建模板，而表 6-8 则详细介绍了每个模块的默认构建模板所用的安装位置。建议留意一下 */system* 的子目录中各文件是如何安装进去的。

表 6-8：构建模板及其相应的输出位置

模板	默认输出位置
BUILD_EXECUTABLE	<i>/system/bin</i>
BUILD_JAVA_LIBRARY	<i>/system/framework</i>
BUILD_SHARED_LIBRARY	<i>/system/lib</i>
BUILD_PREBUILT	没有默认位置。你要确保显式地指明了 LOCAL_MODULE_CLASS 或者 LOCAL_MODULE_PATH。BUILD_MULTI_PREBUILT 依赖于其复制的模块类型
BUILD_PACKAGE	<i>/system/app</i>
BUILD_KEY_CHAR_MAP	<i>/system/usr/keychars</i>

在内部，构建系统会根据模块的生成模板为每个模块自动生成 LOCAL\_MODULE\_PATH，即编译后输出的安装位置。可以通过更改你的 *Android.mk* 文件中的 LOCAL\_MODULE\_PATH 的变量值来覆盖默认值。假设，你有一个自定义工具，希望将其安装在主板的 */sbin* 而不是 */system/bin* 目录，你的 *Android.mk* 可能看起来像这样：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-c-files-under, src)
LOCAL_PACKAGE_NAME := calibratebirdradar
LOCAL_MODULE_PATH := $(TARGET_ROOT_OUT_SBIN)

include $(BUILD_PACKAGE)
```

请注意这里指定的是 `$(TARGET_ROOT_OUT_SBIN)`，而不是 `/sbin`。这样二进制文件会正确输出在 `out/target/product/PRODUCT_DEVICE/` 目录。所有的 `TARGET_ROOT_OUT_*` 以及很多默认安装位置的宏都定义在 `build/core/envsetup.mk` 中。针对我们的目标而言，下面是其相关的代码段：

```
TARGET_ROOT_OUT := $(PRODUCT_OUT)/root
TARGET_ROOT_OUT_BIN := $(TARGET_ROOT_OUT)/bin
TARGET_ROOT_OUT_SBIN := $(TARGET_ROOT_OUT)/sbin
TARGET_ROOT_OUT_ETC := $(TARGET_ROOT_OUT)/etc
TARGET_ROOT_OUT_USR := $(TARGET_ROOT_OUT)/usr
```

## 显式复制文件

有一些文件是不需要构建系统通过任何方式来生成，你只需要将文件复制到它生成的文件系统中。这就要用到 `PRODUCT_COPY_FILES` 宏，可以在产品相关的 `mk` 文件中使用它。例如，第 4 章中 CoyotePad 最新版本的 `full_coyote.mk` 文件如下所示：

```
$(call inherit-product, $(SRC_TARGET_DIR)/product/languages_full.mk)
$(call inherit-product, $(SRC_TARGET_DIR)/product/full.mk)

DEVICE_PACKAGE_OVERLAYS :=

PRODUCT_PACKAGES += 
PRODUCT_COPY_FILES += \
    device/acme/coyotepad/rfirmware.bin:system/vendor/firmware/rfirmware.bin \
    device/acme/coyotepad/rcalibrate.data:system/vendor/etc/rcalibrate.data

PRODUCT_NAME := full_coyotepad
PRODUCT_DEVICE := coyotepad
PRODUCT_MODEL := Full Android on CoyotePad, meep-meep
```

这是分别将文件 `rfirmware.bin` 和 `rcalibrate.data` 从 `device/acme/coyotepad/` 复制到目标的 `/system/vendor/firmware` 目录和 `/system/vendor/etc` 目录。

## 默认权限和所有者

到目前为止，如何配置 Android 文件系统中各目录和文件的权限和所有者是还没有介绍的一个方面。如果你不介意多输几行命令，我强烈建议你去看看 `system/core/`

*include/private/android\_filesystem\_config.h* 文件。这个文件没有什么公开资料<sup>注2</sup>，又找不到任何文档，然而它非常重要，因为它提供了预定义的系统用户，以及系统中所有文件的权限和所有者的列表。这里是 2.3/ 姜饼系统中定义的用户名 / 组名及其Uid/Gid 的部分列表：

```
#define AID_ROOT          0 /* traditional unix root user */
#define AID_SYSTEM         1000 /* system server */
#define AID_RADIO          1001 /* telephony subsystem, RIL */
#define AID_BLUETOOTH      1002 /* bluetooth subsystem */
#define AID_GRAPHICS        1003 /* graphics devices */
#define AID_INPUT           1004 /* input devices */
...
#define AID_RFU2           1024 /* RFU */
#define AID_NFC            1025 /* nfc subsystem */
#define AID_SHELL          2000 /* adb and debug shell user */
#define AID_CACHE          2001 /* cache access */
#define AID_DIAG           2002 /* access to diagnostic resources */
...
#define AID_MISC           9998 /* access to misc storage */
#define AID_NOBODY         9999
#define AID_APP            10000 /* first app user */
...
static const struct android_id_info android_ids[] = {
    { "root",      AID_ROOT, },
    { "system",    AID_SYSTEM, },
    { "radio",     AID_RADIO, },
    { "bluetooth", AID_BLUETOOTH, },
    { "graphics",  AID_GRAPHICS, },
    { "input",     AID_INPUT, },
...
}
```

如果你在登录的设备的 shell 里使用 *ps* 命令，会看到像这样的输出：

```
...
root      18048  1      61552  26700  c00a6548 afd0b844 S zygote
system   18090  18048  141756  50224  ffffffff afd0b6fc S system_server
system   18187  18048  75664   21828  ffffffff afdoc51c S com.android.systemui
app_16   18197  18048  78548   19292  ffffffff afdoc51c S com.android.inputmethod.
                    latin
radio    18200  18048  86400   19580  ffffffff afdoc51c S com.android.phone
app_19   18201  18048  78636   23472  ffffffff afdoc51c S com.android.launcher
app_1    18234  18048  83904   22232  ffffffff afdoc51c S android.process.acore
app_2    18281  18048  72364   16696  ffffffff afdoc51c S com.android.deskclock
...
```

---

注2：这个文件实际上之后会由 Magnus Back 对我指明，他是索尼爱立信公司的工程师，他帮助我复审此书，在我请求获得关于 Android 的文件系统权限管理资料时，他会在 Android 构建邮件列表中回复我。

你可以看到 *system\_server* 以 system 用户身份运行，每个应用程序以一个名为 *app\_N* 的用户身份运行，每个应用程序有一个独立的 N 值。很明显，内核是不提供这些名称的，bionic 则使用了前面的定义来提供 PID/GID 名称转换。对用户应用程序来说，每个应用程序作为一个独立的用户（应用程序的 UID/GID 从 10000 开始），安装的应用程序用户名称均以 *app\_* 开始，然后将应用程序的实际 UID/GID 减去 10000 得到的整数就是 N 值。对于 4.2/ 果冻豆及之后的系统来说，由于存在多用户的 support 而稍有不同。其应用程序的名称是 *uM\_appN* 形式，其中 M 是用户 ID，N 是应用程序 id。

AOSP 构建系统的其他部分可能会允许你将你的特定主板的修改隔离在 *device* 目录中，例如前面所说的 *device/acme/coyotepad*，但是这种办法不能用于修改 *android\_filesystem\_config.h*。以下片段中的粗体行，显示如何修改来添加一个 *birdradar* 用户：

```
...
#define AID_RFU2          1024 /* RFU */
#define AID_NFC           1025 /* nfc subsystem */
#define AID_BIRDRADAR    1999 /* Bird radar subsystem */

#define AID_SHELL         2000 /* adb and debug shell user */
#define AID_CACHE          2001 /* cache access */
#define AID_DIAG           2002 /* access to diagnostic resources */

...
static const struct android_id_info android_ids[] = {
    { "root", AID_ROOT, },
    { "system", AID_SYSTEM, },
    { "radio", AID_RADIO, },
    ...
    { "media", AID_MEDIA, },
    { "nfc", AID_NFC, },
    { "birdradar", AID_BIRDRADAR, },
    { "shell", AID_SHELL, },
    { "cache", AID_CACHE, },
...
}
```

---

注意：我们为新用户添加的用户 ID 是 1999 而不是 1026，这是为了避免在将来发行的 Android 版本中更新此整数，因为谷歌还可能会添加新用户。事实上，上面的代码段是针对 2.3/ 姜饼系统的，最后一个整数是 1025，而在 4.2/ 果冻豆里，默认最后一个编号就已经是 1028 了。

---

添加新的默认用户的原因可能包括为 Android 增加的一个新的、尚未被支持的 硬件类型，或者你希望在 Android 系统里运行你自定义的一套软件框架。当然，也可能想将守护进程独立地以某个用户权限来运行。

这里给出了 *android\_filesystem\_config.h* 中定义目录和文件权限的代码段：

```

static struct fs_path_config android_dirs[] = {
    { 00770, AID_SYSTEM, AID_CACHE, "cache" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/app" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/app-private" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/dalvik-cache" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/data" },
    ...
    { 00750, AID_ROOT, AID_SHELL, "sbin" },
    { 00755, AID_ROOT, AID_SHELL, "system/bin" },
    { 00755, AID_ROOT, AID_SHELL, "system/vendor" },
    ...
    { 00755, AID_ROOT, AID_ROOT, 0 },
};

...
static struct fs_path_config android_files[] = {
    { 00440, AID_ROOT, AID_SHELL, "system/etc/init.goldfish.rc" },
    { 00550, AID_ROOT, AID_SHELL, "system/etc/init.goldfish.sh" },
    ...
    { 00644, AID_SYSTEM, AID_SYSTEM, "data/app/*" },
    { 00644, AID_SYSTEM, AID_SYSTEM, "data/app-private/*" },
    { 00644, AID_APP, AID_APP, "data/data/*" },
    ...
    { 00755, AID_ROOT, AID_SHELL, "system/bin/*" },
    { 00755, AID_ROOT, AID_SHELL, "system/xbin/*" },
    { 00755, AID_ROOT, AID_SHELL, "system/vendor/bin/*" },
    { 00750, AID_ROOT, AID_SHELL, "sbin/*" },
    { 00755, AID_ROOT, AID_ROOT, "bin/*" },
    { 00750, AID_ROOT, AID_SHELL, "init*" },
    { 00644, AID_ROOT, AID_ROOT, 0 },
};

```

如果出于某种原因，你要添加一个新的目录或文件到文件系统中，其默认的所有者和访问权限将取决于刚才所显示数组中的最后一项，即 0 而不是路径。换句话说，一个新的目录将配置为 755 访问权限，文件会配置为 644 的访问权限，并被 AID\_ROOT 用户 / 组所拥有。

如果你想要像附录 A 那样添加 glibc 链接的二进制文件到你的设备上，你可能需要有一个 `/lib` 目录保存 glibc 库 (`/lib` 是传统 Linux 下的 C 库的默认路径)。然而在默认情况下，放在该目录下没有可执行权限，即使它们是在你的主机生成的<sup>注3</sup>，因此，用 glibc 链接的任何二进制文件都将无法运行。要解决此问题，你需要修改 `android_filesystem_config.h` 中的 `android_files` 数组，如下所示：

```

...
{ 00750, AID_ROOT, AID_SHELL, "sbin/*" },
{ 00755, AID_ROOT, AID_ROOT, "bin/*" },
{ 00755, AID_ROOT, AID_ROOT, "lib/*" },
{ 00750, AID_ROOT, AID_SHELL, "init*" },

```

<sup>注3</sup>: 这是因为构建系统忽略主机上文件的所有者和权限信息，相反构建系统只依赖于 `android_filesystem_config.h` 文件。

```
{ 00644, AID_ROOT,      AID_ROOT,      0 },  
};
```

这是你不能将其隔离到 *device/acme/coyotepad* 中的另一种修改。

请注意，通常情况下 */system/vendor* 目录是留给厂商做扩展的。事实上 *android\_filesystem\_config.h* 指出在 */system/vendor/bin* 中的所有二进制文件应该都是可执行文件。因此，如果你要将大量的文件添加到文件系统，你可以想想是否可以把你所添加的内容放到 */system/vendor* 目录中。这将是最干净的方法，不过谁说嵌入式和干净是同义词？嘿嘿！

---

**注意：**一般来说，如果你想要简化你的设备对将来 Android 版本的支持工作，那么在 Android 框架推荐的修改范围内修改是最好的。理论上讲，如果你将设备相关的代码隔离在 *device* 目录，移植到下一个版本时就只需将你的 *device* 目录复制到新的框架中，然后调整一下 API 相关的代码。

这个规则对手机来说特别适用，不过由于嵌入式系统通常是一次性的，如果前一个版本需要关键性更新，那么下一个版本的设计可能会完全换一个新的 SoC。因此，遵守“规则”在这种情况下实际上可能会起反作用，它就会施加不必要的制约和限制。在本书中我会不断指出我们做事的“Android 方式”以及所有其他的可能性，然后留给你来决定什么是最适合你自己的项目的方法。

---

## adb

文件系统的布局只是为 Android 系统其他部分提供了存储空间。在板子启动过程中，内核启动完成后最首要确保运行正常的软件就是 *adb* 了。在第 3 章，我们介绍了它的基本操作，接下来更深入地讨论这个话题。

### 基础概念

尽管 *adb* 用起来非常简单，但它对应用开发和平台开发来说却是一个至关重要的工具。虽然 Android 从传统嵌入式 Linux 系统中引入了很多东西，有些部分重新开发了相应功能，但是 *adb* 却是一个全新的东西，在 Android 之前没有任何项目或者软件包提供类似的功能（至少我没见过）。所以在主机与目标版之间的交互方面，*adb* 填补了一个重要的空白。

*adb* 实际上是由多个组件组成，这些组件又各自连接到其他系统组件上，它们组合在一起提供了一个集成的 *adb* 功能。图 6-3 给出了 *adb* 的连接和基本操作。有意思的是，*adb* 的主机端和目标端都是用同一套代码 (*system/coreadb/*) 编译出来的，从而保证了组件间的版本一致性。

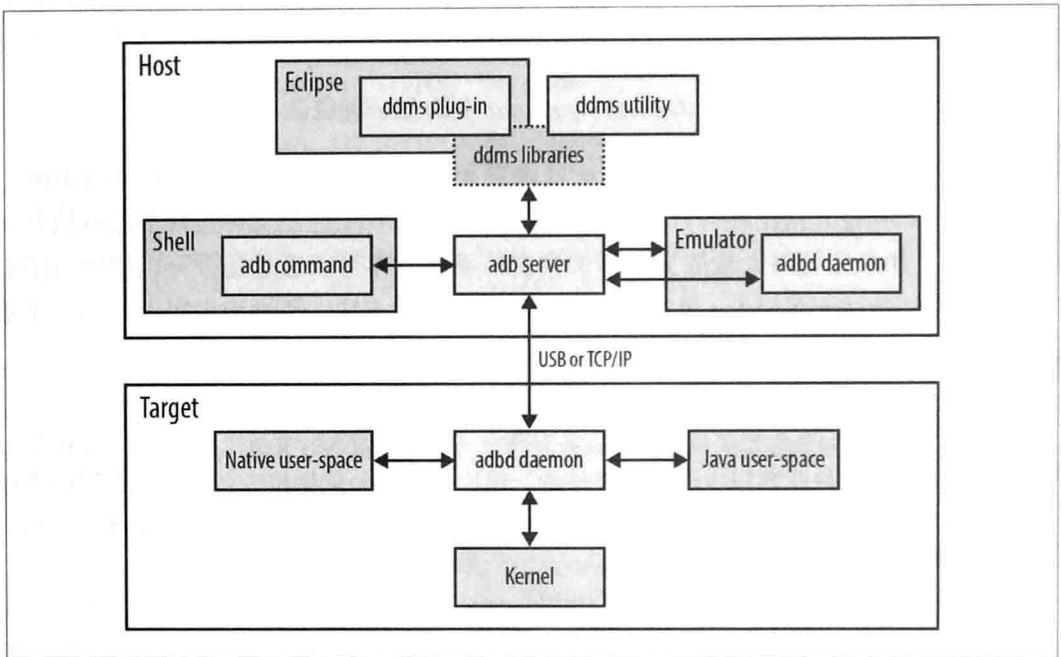


图 6-3: `adb` 及其相互连接

实际上，`adb`既是一种透明的数据传输机制，又是一个服务提供者。它的两个最重要的组件是运行在主机上的`adb`服务器和运行在目标机上的`adbd`守护程序。这两个组件有效地实现了一个代理协议，通过这个协议实现了所有的`adb`服务。它们可以通过USB或者普通TCP/IP连接到一起，两种方式提供的`adb`命令是一样的。

---

**注意：**Android这个地方的命名可能会让人迷惑。一般来说，服务器是运行在远端，客户端工具通过网络连接到服务器。而在这个例子中，`adb`的“服务器”实际上是在主机后台运行的一个程序，而`adbd`是目标板后台运行的另一个程序。

---

当我们在命令行中运行`adb`命令时，`adb`服务器会自动启动。它会监控连接上的设备并且维护与对方`adbd`守护程序的通信。`adbd`再通过接口与本地用户程序、Java用户程序，以及内核进行通信来提供相应功能。我们会在后面讨论`adb`功能时进一步介绍它的部分接口。

在主机侧，有两个软件模块会主动发起跟`adb`服务器的连接：`adb`命令行和Davik调试监控服务器（Dalvik Debug Monitor Server, ddms）库（`ddmlib`和`ddmuilib`）。`ddms`库本身被`ddms`工具和Eclipse的`ddms`插件使用（`ddms`工具是一些独立应用的小工具），它们在ADT安装的时候被自动安装到系统中。`ddms`库提供了跟`adb`服

务器通信的接口（ddmlib）以及显示 / 管理的 UI 部分（ddmuilib）。这就是为什么在 Eclipse 插件和在 ddms 工具中呈现的用户接口是一致的。

adb 命令和 ddms 库并不提供完全一样的 adb 服务器的功能。例如，adb 服务器能够抓取目标机帧缓存中的内容来提供屏幕截图，这个功能在 ddms 工具中有提供，但是 adb 命令行中却没有这个命令。

adb 服务器通过套接字在主机端口 5037 提供它的服务，遵守该协议的任何人都可以连接上去。如果你想实现直接跟 adb 服务器通信，可以看一看 system/core/adb 目录中的 OVERVIEW.TXT 和 SERVICES.TXT 文件。与连接远程目标机一样，adb 服务器也可以用同样的方式与模拟器中的 abd 守护进程进行交互。

adb 甚至还可以操作模拟器的控制台。每一个模拟器会在不同的端口上等待 telnet 连接，这个端口号从 5554 开始并且会显示在模拟器窗口的左上角。使用 telnet 连接后你就可以控制模拟器的行为，包括转发某个端口信息到模拟器，以及修改模拟器的窗口大小等，Google 的应用开发者手册中有详细介绍使用模拟器的方法。为了简化这个过程，adb 可以允许你通过它来发送这些命令给模拟器而无需打开 telnet。

## 主要的参数、标志和环境变量

我们在第 3 章中有提到（而且后面也会详细介绍）adb 提供了大量的命令。而且，adb 可以被同时用于多个设备以及不同的 AOSP 构建中。所以，如表 6-9 所示，有一些标志、参数和环境变量用于控制它的行为。如果只有一个设备或者一个模拟器实例在运行，adb 的操作就会相对比较简单，它假定这个实例就是你的命令要操作的对象。

表 6-9：adb 的标志、参数和环境变量

项目	描述
-d	这个标志告诉 adb 将命令发送给 USB 连接的设备。如果你同时有一个模拟器和一个 USB 连接的 Android 设备，这个标志可以确保 adb 在你的设备上运行你的命令，而不是模拟器。很明显，如果你有多个 USB 连接的设备时就没用了
-e	跟 -d 类似，这个标志告诉 adb 将命令只发给模拟器，不管是否有 USB 设备连接。同样，如果多个模拟器实例也不能区分
-s <序列号>	这是告诉 adb 连接到由给定的序列号指定的设备。尽管使用每个 adb 命令时必须输入设备的完整序列号是很乏味的，这是（以及下面的 ANDROID_SERIAL）唯一的区分方法，如果你有多个设备连接或运行了多个模拟器的话

表 6-9: adb 的标志、参数和环境变量（续）

项目	描述
-p<产品名称或路径>	有些adb的命令需要访问被用来构建目标的AOSP的源代码。如果你正运行的adb也是来自于与构建AOSP相同的shell，由于ANDROID_PRODUCT_OUT环境变量将被设置，这些都能够正确地被找到。如果不是这种情况的话，你就需要使用-p来指示AOSP源代码树中的产品的输出目录的路径
ANDROID_SERIAL	如果你不断地有多设备连接，并希望避免使用-s标志来指定一个你会非常频繁操作的特定串行设备，可以设置ANDROID_SERIAL环境变量为该设备的序列号，那么adb在默认情况下将始终连接到该设备上，除非你明确地使用-s来覆盖
ADB_TRACE	如果你想调试或监视主机上的adb服务和目标上的adb守护进程之间的交互，可以设置ADB_TRACE环境变量为一个或一系列以逗号、冒号、分号，或空格符号分隔的以下值的组合：1、all、adb、sockets、packets、rwx、usb、sync、sysdeps、transport、jdwp

## 基本的本地命令

让我们先从一些adb本地运行的基本命令开始。首先，如果你想手动启动adb服务，可以像下面这样做：

```
$ adb start-server
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
```

但是，当你在需要的时候输入了任何其他的adb命令，该服务将会自动启动。所以，你通常可以跳过手动启动服务这一项。不幸的是，在有些情况下，你实际上必须手动关闭该服务，通常需要这样做是在你的某个adb命令被挂起的时候：

```
$ adb kill-server
```

如果你想知道adb的功能，你可以不带任何参数地启动以下命令或输入：

```
$ adb help
Android Debug Bridge version 1.0.26

-d           - directs command to the only connected USB device
               returns an error if more than one USB device is
               present.
-e           - directs command to the only running emulator.
               returns an error if more than one emulator is
               Running.
```

```
-s <serial number>           - directs command to the USB device or emulator  
                                with the given serial number. Overrides  
                                ANDROID_SERIAL  
...  
device commands:  
adb push <local> <remote>    - copy file/dir to device  
adb pull <remote> [<local>]   - copy file/dir from device  
adb sync [ <directory> ]      - copy host->device only if changed  
                                (-l means list but don't copy)  
                                (see 'adb help all')  
adb shell                      - run remote shell interactively  
adb shell <command>            - run remote shell command  
adb emu <command>              - run emulator console command  
...
```

上面显示的帮助屏幕，将命令的版本号作为输出的一部分输出。但是你还可以让 *adb* 明确地打印出它的版本号：

```
$ adb version  
Android Debug Bridge version 1.0.26
```

像其余的 AOSP 一样，*adb* 是一个活动的目标。以下是在 4.2/ 果冻豆中的版本：

```
$ adb version  
Android Debug Bridge version 1.0.31
```

## 设备连接和状态

现在让我们来看看 *adb* 提供的用于管理与设备进行通信的命令。首先，如果你想知道哪些设备对于 *adb* 是可见的，可以输入：

```
$ adb devices  
List of devices attached  
emulator-5554 device  
0123456789ABCDEF device  
emulator-5556 device
```

如果你想连接到一个远程设备上，该设备的 *adbd* 守护进程是运行在 TCP/IP 连接上而不是 USB 连接的，你可以使用 *connect* 命令：

```
$ adb connect 192.168.202.79:7878  
connected to 192.168.202.79:7878  
$ adb devices  
List of devices attached  
emulator-5554 device  
0123456789ABCDEF device  
emulator-5556 device  
192.168.202.79:7878 device
```

*connect* 命令的格式化表示形式为（5555 是默认的端口）：

```
adb connect <host>[:port]
```

想要指定一个目标作为一个给定的命令显示的地方，只需要使用由 *adb device* 命令显示的 IP: PORT 信息作为序列号。比如，获取一个 shell：

```
$ adb -s 192.168.202.79:7878 shell
```

当你这样做完以后，可以断开设备连接，它就会在由 *adb* 服务可见的设备列表中停止显示：

```
$ adb disconnect 192.168.202.79:7878
```

*disconnect* 命令的格式化表示方式为（如果没有设备被指定，那么所有 TCP/IP 连接的设备都将断开连接）：

```
adb disconnect [<host>[:port]]
```

如果你想要 *adb* 挂起等待设备再上线，你可以输入如下的命令：

```
$ adb wait-for-device
```

那么 shell 将被暂停，直到设备上线为止。当该设备处于上线状态时 *adb* 将返回到 shell。这对于脚本编写是非常有用的，因为你可以让你的脚本在继续执行其他命令前等待设备做好准备。

如果你想要请求一个设备的状态，你可以输入：

```
$ adb -s 0123456789ABCDEF get-state  
device
```

状态包括引导程序（bootloader）、设备（device）、离线（offline）和一些未知状态（unknown）。这里的 device 值是设备处于上线状态的代名词。*offline* 的意思不言自明。*bootloader* 是指设备当前处于引导程序中。*unknown* 表示 *adb* 无法识别该设备的当前状态。

如果因为任何原因，你需要明确知道一个设备的序列号，比如当你在脚本编写 *adb* 命令时，可以这样做：

```
$ adb -d get-serialno  
0123456789ABCDEF
```

最后，如果你需要有一个 shell 窗口打开，来不断地报告当前设备的状态，你可以这样做：

```
$ adb -d status-window
```

这个操作将清除屏幕，并在终端的顶部上这样显示（状态报告在 State 旁边：作为设备的“实时”状态）：

```
Android Debug Bridge  
State: device
```

想要退出，你可以输入 Ctrl-C。

## 基本的远程命令

到现在为止，我们已经看到的命令并没有真正使我们能够在远程目标机上做任何事情或获取有关它的任何信息。因此，让我们开始做一些有趣的事。

### shell

很显然，如果你是一个像我一样的极客，你会想要做的第一件事就是到登录到你的设备上，做一些有趣或有用的事。在版本 2.3/ 姜饼中，你会得到这样的信息：

```
$ adb shell  
#
```

4.2/ 果冻豆有一个更丰富的 shell，如下所示：

```
$ adb shell  
root@android:/ #
```

在这两种情况下，*adb* 守护进程的命令结果是在目标机上生成一个 shell 来执行你输入的命令。所有命令的输入 / 输出（即 *stdin*、*stdout* 和 *stderr*）将被运行在主机上的 *adb* 服务器和运行在目标机上的 *adbd* 守护进程所代理。

想要从目标机的 shell 退出并返回到你的主机的 shell，只需按下 Ctrl-D。如果你也想要启动一个特定的命令，可以通过将它作为一个参数传递给 shell 命令。在这种情况下，可以打印出 BeagleBone 的 CPU 信息：

```
$ adb -d shell cat /proc/cpuinfo  
Processor      : ARMv7 Processor rev 2 (v7l)  
BogoMIPS      : 718.02  
Features       : swp half thumb fastmult vfp edsp thumbee neon vfpv3 tls  
CPU implementer : 0x41  
CPU architecture: 7  
CPU variant   : 0x3  
CPU part      : 0xc08  
CPU revision  : 2  
  
Hardware      : am335xevm
```

```
Revision      : 0000
Serial       : 0000000000000000
```

以下是 *shell* 的格式描述：

```
adb shell [ <command> ]
```

## 查看日志信息

如果你想要查看 Android 的日志缓冲区，可以输入如下的命令：

```
$ adb -d logcat
----- beginning of /dev/log/main
I/DEBUG ( 59): debuggerd: Mar 27 2012 05:30:39
----- beginning of /dev/log/system
I/Vold ( 57): Vold 2.1 (the revenge) firing up
D/Vold ( 57): USB mass storage support is not enabled in the kernel
D/Vold ( 57): usb_configuration switch is not enabled in the kernel
D/Vold ( 57): Volume sdcard state changing -1 (Initializing) -> 0 (No-Media)
D/Vold ( 57): Volume usb state changing -1 (Initializing) -> 0 (No-Media)
D/Vold ( 57): Volume sdcard state changing 0 (No-Media) -> 2 (Pending)
D/Vold ( 57): Volume sdcard state changing 2 (Pending) -> 1 (Idle-Unmounted)
I/Netd ( 58): Netd 1.0 starting
I/ ( 61): ServiceManager: Oxad50
W/AudioHardwareInterface( 61): Using stubbed audio hardware. No sound will be
produced.
D/AudioHardwareInterface( 61): setMode(NORMAL)
I/CameraService( 61): CameraService started (pid=61)
I/AudioFlinger( 61): AudioFlinger's thread 0xc638 ready to run
E/dhcpcd ( 65): timed out
D/AndroidRuntime( 224):
D/AndroidRuntime( 224): >>>>> AndroidRuntime START com.android.internal.os.Zyg
oteInit <<<<<
D/AndroidRuntime( 224): CheckJNI is ON
D/dalvikvm( 224): creating instr width table
I/SamplingProfilerIntegration( 224): Profiler is disabled.
I/Zygote ( 224): Preloading classes...
...
```

这个命令实际上等效以下的命令：

```
$ adb -d shell logcat
```

我们之后将进一步详细讨论 *logcat* 命令，但要知道，你可以在你输入的 *adb logcat* 之后排列一些类似的参数，就好像你是直接从目标机的命令行中运行 *logcat* 一样。所以，举例来说，如果你想查看“radio”缓冲区，而不是“main”的缓冲区，你可以这样做：

```
$ adb -d logcat -b radio
I/PHONE ( 394): Network Mode set to 0
I/PHONE ( 394): Cdma Subscription set to 1
```

```
I/PHONE ( 394): Creating GSMPhone
D/PHONE ( 394): mDoesRilSendMultipleCallRing=true
D/PHONE ( 394): mCallRingDelay=3000
W/GSM   ( 394): Can't open /system/etc/voicemail-conf.xml
W/GSM   ( 394): Can't open /system/etc/spn-conf.xml
D/GSM   ( 394): [DSAC DEB] registerForPsRestrictedEnabled
D/GSM   ( 394): [DSAC DEB] registerForPsRestrictedDisabled
D/GSM   ( 394): [GsmDataConnection-1] DataConnection constructor E
D/GSM   ( 394): [GsmDataConnection-1] clearSettings
D/GSM   ( 394): [GsmDataConnection-1] DataConnection constructor X
D/GSM   ( 394): [GsmDataConnection-1] Made GsmDataConnection-1
D/RILJ  ( 394): [0000]> RIL_REQUEST_REPORT_STK_SERVICE_IS_RUNNING
D/STK   ( 394): StkService: StkService: is running
...
...
```

如果是当你启动命令时该环境变量就在主机的 shell 中被设置的话，*adb* 还将兑现 `ANDROID_LOG_TAGS` 环境变量。`ANDROID_LOG_TAGS` 是由 *logcat* 考虑的，正如我们将在后面看到的，用于过滤它打印的输出。*logcat* 的格式化描述如下：

```
adb logcat [ <parameters> ]
```

## 带有 ddms 库的 logcat

如果你曾经使用过 *ddms* 或 Android 的 ADT 插件，你就会知道它们可以显示出与 *logcat* 命令在命令行中打印出的相同的 Android 日志信息。但是，它们在如何检索信息方面还是有一些差别。同时，正如我刚才介绍的，一个 *adb logcat* 命令实际上只是在目标机上运行 *logcat* 命令，并将输出信息代理显示回主机上，*ddms* 的库使用一个不同于用于代理 shell I/O 的 log service 的 *adb* 服务器机制。该服务将相关的 `/dev/log` 缓冲区的内容直接显示回主机上，而不通过目标机的 *logcat*。实际上有两种方法来解决问题，这只是其中一种。

*adb* 服务和它的客户端之间的协议其实是很丰富的，像我前面提到的那样。你需要深入了解 *adb* 的来源，来了解它的全貌，但我只想说服务器与目标机的 *adb* 守护进程之间通信，以提供多种类型的服务。所有通过 *adb* 命令行和 *ddms* 显示出来的 ADB 功能都依赖于这些服务。但是，你可以编写代码来直接与 *adb* 服务进行对话，来接入所有它所提供的服务。

## 获取调试报告

跟 *logcat* 目标机命令很相似——有一个 *adb* 的快捷方式，你不需要明确告诉它去调用 shell——*adb* 也为 *bugreport* 提供了一个快捷方式。后者是一个目标命令，可以查看系统状态，目的是用于调试报告（bug-reporting）。实际上，它使得 *dumpstate* 命令可在目标机上运行：

```
$ adb -d bugreport
=====
== dumpstate: 2000-01-01 05:05:08
=====

Build: beaglebone-eng 2.3.4 GRJ22 eng.karim.20120327.052544 test-keys
Bootloader: unknown
Radio: unknown
Network: (unknown)
Kernel: Linux version 3.1.0-g62911f8-dirty (a0131746@sditapps03) (gcc version 4.
4.3 (GCC) ) #1 Mon Nov 28 22:05:07 IST 2011
Command line: console=tty00,115200n8 androidboot.console=tty00 mem=256M root=/de
v/mmcblk0p2 rw rootfstype=ext3 rootwait init=/init ip=off

----- MEMORY INFO (/proc/meminfo) -----
MemTotal:      253264 kB
MemFree:       198308 kB
...
...
```

你可能会想，*bugreport* 命令调用 *dumpstate*，那为什么不直接用下面这样的命令来代替？

```
$ adb -d shell dumpstate
```

麻烦的是，*dumpstate* 需要以 root 身份来运行，而有些设备不允许它们的 shell 以 root 身份运行。这就是目前市场上的绝大多数手持设备的情况。因此，在这些设备上，输入上述命令将是不可能的，但它仍然可以使用 *bugreport* 命令。以下是我的手机上会发生的情况：

```
$ adb -s 4xxxxxxxxxxxxx shell dumpstate
dumpstate: permission denied
$ adb -s 4xxxxxxxxxxxxx shell bugreport
=====
== dumpstate: 2012-05-04 13:38:05
=====

Build: GINGERBREAD.UCKI3
Bootloader: unknown
Radio: unknown
...
...
```

从本质上讲，*bugreport* 会导致 *init* 以一种模式来启动 *dumpsys*，这种模式打开一个 UNIX 域套接字，并监听想要查看的输出的连接。接着 *bugreport* 连接到这个套接字上，并将其读取的内容复制到自己的标准输出中，然后通过 *adb* 代理显示到你的主机的 shell 中。因此，用户或技术人员可以为你的设备创建错误报告，即使你没有给他们 root 权限。

## 端口转发

*adb* 的另一个非常有趣的特点是，它可以在主机和目标机的端口之间进行转发。例如，以下命令将转发本地端口 8080 目标机的 80 端口：

```
$ adb -d forward tcp:8080 tcp:80
```

此后，任何到主机的端口 8080 的连接都将被重定向到目标机的 80 端口。例如，如果你在你的 Android 设备上正在运行一个 Web 服务器（默认运行在端口 80），就可以将主机的网络浏览器连接到 `localhost:8080` 来浏览你的设备。

但是，*adb* 的 *forward* 命令可做的事不止于此。实际上它能够转发主机端口到目标机的不仅仅是端口上。例如，可以转发本地端口 8000 到目标机的字符设备之一：

```
$ adb -d forward tcp:8000 dev:/dev/ttyUSB0
```

在这种情况下，端口 8000 进行的任何读 / 写操作都将导致远程的 `/dev/ttyUSB0` 上的读 / 写操作。表 6-10 列出了 *forward* 命令支持的连接类型，以及其形式化描述为：

```
adb forward <local> <remote>
```

表 6-10：adb 的 forward 连接类型

连接	描述
<code>tcp:&lt;port&gt;</code>	常规的 TCP 端口。这是一个非负的整数值
<code>localfilesystem:&lt;unix domain socket&gt;</code>	一个常规的 UNIX 域套接字。它显示为一个文件系统的条目
<code>localabstract:&lt;unix domain socket&gt;</code>	一个“抽象的” UNIX 域套接字。这就好比一个 UNIX 域套接字，但它是一个 Linux 特定的扩展。看看 UNIX 手册页了解更多细节： <code>man 7 unix</code>
<code>localreserved:&lt;unix domain socket&gt;</code>	Android 的“保留” UNIX 域套接字。它们都在 <code>/dev/socket</code> 文件夹中，并且有非常特定的用途，我们将随着前进的步伐在后面讨论它。这些用途包括 <code>dbus</code> 、 <code>installd</code> 、 <code>keystore</code> 、 <code>netd</code> 、 <code>property_service</code> 、 <code>rild</code> 、 <code>rild-debug</code> 、 <code>vold</code> 和 <code>zygote</code>
<code>dev:&lt;character devicename&gt;</code>	目标的实际设备。在文件系统中，必须为设备提供完整的路径
<code>jdwp:&lt;pid&gt;</code>	用于指定一个 Dalvik 进程的 PID 进行调试

## Dalvik 调试

对于 *forward* 的代理连接到 Dalvik 这个过程的能力，值得我们花多一点的篇幅来介绍。Dalvik 实现了 Java 调试有线协议 (JDWP)，从而使你可以使用常规的 Java 调试器

*jdb* 来调试应用程序。显然，这对于应用程序开发人员来说是封装在 Eclipse 里的，但如果你想在命令行中使用 *jdb*, *forward* 的重定向 Dalvik 进程的调试端口到你的主机的功能，就显得至关重要了。下面就是一个例子：

```
$ adb forward tcp:8000 jdwp:376
$ jdb -attach localhost:8000
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
>
```

想要知道哪一些 PID 是可通过 JDWP 调试的，你可以输入如下命令：

```
$ adb jdwp
271
376
386
389
390
425
473
480
...
```

实际上，*adb* 是在目标机上进行任何 Java 调试的一个重要组成部分。当 *adbd* 守护程序在目标机上启动时，它打开了“抽象的” UNIX 域套接字 *jdwpcontrol* 并等待连接。Dalvik 进程启动后连接到这个套接字，从而使得自己对于调试来说是“可见的”。为了让应用程序开发人员调试他们的应用程序，*ddms* Eclipse 插件通过 *ddmlib* 跟 *adb* 服务器进行对话，来调试应用程序。或者，正如我们刚才看到的，你可以在命令行中使用 *adb* 来进行调试。

请注意，实现这一切需要的是 *adbd* 要在任何 Dalvik 应用程序被启动之前就要运行在目标机上了，只有那些在 *adbd* 启动之后启动的 Dalvik 应用程序才是可调试的。

## 文件系统命令

*adb* 还允许你以各种方式对目标机的文件系统进行操作和交互。例如，如果想将文件复制到设备上，你可以使用 *push* 命令：

```
$ adb push acme_user_manual.pdf /data/local
```

以下命令将 *acme\_user\_manual.pdf* 文件复制到目标机的 */data/local* 文件夹中：

```
$ adb shell ls /data/local
acme_user_manual.pdf
```

你也可以将文件从目标机复制到主机上：

```
$ adb pull /proc/cpuinfo
$ cat cpuinfo
Processor : ARMv7 Processor rev 2 (v7l)
BogoMIPS : 718.02
...
```

正如我在本章前面介绍的那样，目标机的文件系统部件并非所有的都挂载有相同的权限。例如，*/system* 通常被挂载为只读的。如果你想要将它重新挂载为读 / 写模式，可以增加或修改一个它包含的文件，比如，你可以使用 *remount* 命令来这样做。下面是一个例子：

```
$ adb push acme_utility /system/bin
failed to copy 'acme_utility' to '/system/bin/acme_utility': Read-only file system
$ adb remount
remount succeeded
$ adb push acme_utility /system/bin
$
```

当然 *push* 命令的功能仅仅在复制一些文件的时候是非常有用的。如果你正在寻找更新目标机的 */data* 或是 */system* 分区的方法，你可以使用 *sync* 命令来实现。这将从根本上进行类似 *rsync* 命令的操作，确保目标机的文件与主机上的同步。如果在目标机构建 AOSP 所在的目录下运行 *adb sync* 命令，那么它会自动找到同步的文件，这是因为 *ANDROID\_PRODUCT\_OUT* 环境变量指向正确的目录（当然，这是假设你由于需要，已经在你的目标机上运行过了 *build/envsetup.sh* 和 *lunch* 的前提下）。否则，你需要像下面这样手动将其指向正确的输出目录：

```
$ adb -d -p ~/android/beaglebone/out/target/product/beaglebone/ sync
syncing /system...
push: /home/karim/android/beaglebone/out/target/product/beaglebone/system/xbin/crasher -> /system/xbin/crasher
push: /home/karim/android/beaglebone/out/target/product/beaglebone/system/xbin/scp -> /system/xbin/scp
push: /home/karim/android/beaglebone/out/target/product/beaglebone/system/xbin/opcontrol -> /system/xbin/opcontrol
push: /home/karim/android/beaglebone/out/target/product/beaglebone/system/xbin/tcpdump -> /system/xbin/tcpdump
push: /home/karim/android/beaglebone/out/target/product/beaglebone/system/xbin/oprofiled -> /system/xbin/oprofiled
push: /home/karim/android/beaglebone/out/target/product/beaglebone/system/xbin/timeinfo -> /system/xbin/timeinfo
push: /home/karim/android/beaglebone/out/target/product/beaglebone/system/xbin/cpueater -> /system/xbin/cpueater
...
491 files pushed. 0 files skipped.
1317 KB/s (81337934 bytes in 60.310s)
syncing /data...
push: /home/karim/android/beaglebone/out/target/product/beaglebone/data/app/gles
```

```
2_texture_stream.apk -> /data/app/gles2_texture_stream.apk
push: /home/karim/android/beaglebone/out/target/product/beaglebone/data/app/test
_iterator_host -> /data/app/test_iterator_host
push: /home/karim/android/beaglebone/out/target/product/beaglebone/data/app/test
_iostream_host -> /data/app/test_iostream_host
push: /home/karim/android/beaglebone/out/target/product/beaglebone/data/app/test
_string_host -> /data/app/test_string_host
...
25 files pushed. 0 files skipped.
2804 KB/s (4078615 bytes in 1.420s)
```

在这样的更新后你可能想要重新启动目标机，否则有可能会引用过时的文件。请注意，*sync* 仅同步 */system* 和 */data* 目录。它不同步别的任何东西。换句话说，你不能使用 *sync* 命令来同步挂载为目标机根文件系统的 RAM 磁盘的内容。即使它允许你进行同步，也不会有太大的用处，因为 RAM 盘只在 RAM 中有效，且它的内容不通过持久存储写入。

*sync* 命令也可以被用于只同步数据或系统分区，而不是两者都同步。简单地将你想同步的分区作为参数进行传递：

```
$ adb -e sync data
syncing /data...
...
```

*sync* 的形式描述如下：

```
adb sync [ <directory> ]
```

假如，你要做的不是复制单个文件或整个同步分区，而是安装新的应用程序，那么你应该使用 *install* 命令：

```
$ adb install FastBirds.apk
299 KB/s (13290 bytes in 0.043s) pkg: /data/local/tmp/FastBirds.apk
Success
```

从本质上来说，这将在目标机上调用 *pm*（简称“包管理器”）命令。它将自己与包管理器进行交互来运行你的应用程序的安装。想要从设备上删除它，你可以使用 *uninstall* 命令：

```
$ adb uninstall com.acme.fastbirds
Success
```

你可能已经注意到，*install* 命令依赖于文件名，而 *uninstall* 命令实际需要完整的包名。每个命令实际上可以带有一些标志，如表 6-11 解释了这些标志：

```
adb install [-l] [-r] [-s] <file>
adb uninstall [-k] <package>
```

表 6-11: install 和 uninstall 命令的标志

标志	描述
-l	告诉 <i>install</i> 确保该应用程序是转发锁定 (forward-locked) 的。换句话说，它不允许用户将 .apk 文件复制出设备。在实践中，这意味着应用程序被安装在 /data/app-private 中，而不是 /data/app 中
-r	告诉 <i>install</i> 要重新安装应用程序，并保留之前的数据
-s	告诉 <i>install</i> 在外部存储器 (SD 卡) 中，而不是内部存储器中安装应用程序
-k	告诉 <i>uninstall</i> 即使 .apk 被删除了，也还要保留应用程序的数据

## 状态修改命令

在这一节中，这一个类别缺乏一个更好的名字，对于所有命令以这种或那种方式显著地修改目标机的行为，我已经混为一谈了。它不像以前的命令一样不能或者没有修改目标机，在这里你会发现用很显著的方式来这样做。

### 重启

让我们以一个更明显的命令来开始说：

```
$ adb reboot
```

正如你已经猜到的，这是在重启目标机。实际上这是在目标机上的内核中调用了 `reboot()` 系统调用，同时传递给它适当的值以实现重启。也可以传递一个参数给 `reboot`，来告诉它是以引导装载程序 (bootloader) 还是以恢复 (recovery) 模式来进行重启：

```
$ adb reboot bootloader
```

以及：

```
$ adb reboot recovery
```

但是请注意，这个参数是传递到内核的。如何适当的处理这个参数，这将是你的支持内核代码的开发板的工作。如果你的支持内核代码的开发板不处理传递给 `reboot()` 函数的字符串，将会忽略它，所做的一切就是一个普通的重启。另一种进入引导装载程序的重启方式是：

```
$ adb reboot-bootloader
```

在这里很重要的是要强调，所有这些重启命令都会导致立即重启。期间没有任何正常关闭任何进程或系统服务的过程。因此，如果你需要做任何清理，最好在发出 `reboot` 命令之前做。

## 作为根用户运行

默认情况下，在开发板上，大部分的 *adb* 命令将没有任何问题的发挥自己的全部功能，这是因为目标机上的 *adbd* 守护进程很可能是以 root 身份运行的。但是，在一个类似于商用手持设备这样的生产系统中，很可能 *adbd* 不是以 root 身份运行的，而是作为 shell 用户运行，它只有很少的特权。因此，类似 *adb shell* 这样的命令，将只以 shell 的权限来运行。

*adbd* 守护进程的默认权限将取决于 AOSP 是如何建立的，以及它所运行的目标机。例如，如果它运行在模拟器上，*adbd* 总是以 root 身份运行。在其他情况下，*adbd* 的权限将取决于所选的构建 AOSP 的 TARGET\_BUILD\_VARIANT 参数。如果是 userdebug 或 user 的身份，*adbd* 将不会以 root 身份运行，它在开始时就会作为 shell 用户运行。在 userdebug 的情况下，你可以通过输入以下命令来要求它重新以 root 身份启动：

```
$ adb root  
restarting adbd as root
```

如果你是以 user 的身份构建的，发出一个相同的命令时，你会得到这样的提示，换句话说，你不能更改默认的情况：

```
$ adb root  
adbd cannot run as root in production builds
```

如果你是以 eng 的身份来构建的，很可能在开发过程中，*adbd* 是以 root 身份启动的，以下就是当你坚持这样做时会发生的事：

```
$ adb root  
adbd is already running as root
```

如果由于之前执行的 *adb root* 命令，系统已经用 root 身份运行 *adbd* 了，所发生的事情也是一样的。所有的这些行为已经被 ro.secure、ro.debuggable 和 service.adb.root 全局属性设定了限制。前两个是在构建时被设置的，而后者则是由 *adb* 的 root 命令来设置的。user 和 userdebug 使得 ro.secure 被设置为 1，仅仅 userdebug 和 eng 使得 ro.debuggable 被设置为 1。显然，这些全局属性不仅仅被 *adbd* 所检查。

## 切换连接方式

默认情况下，*adb* 服务器只检查运行在主机上或通过 USB 物理连接到主机上的设备上运行的模拟器实例。像我们前面看到的那样，你仍然可以用 *adb connect* 命令连接到带有自己的 *adbd* 守护进程并通过 TCP/IP 端口监听的，而不是使用 USB 连接的设备上。我们还没有讲到的是如何使用 TCP/IP 代替 USB 来获取 *adbd*。假设该设备已经通过 USB 连接，你可以要求它使用 TCP/IP 来代替，就像下面这样：

```
$ adb -s 0123456789ABCDEF tcpip 7878
restarting in TCP mode port: 7878
```

从本质上讲，这个命令将目标机的 `service.adb.tcp.port` 全局属性设置为 7878，并重新启动 `adb` 守护进程。在重新启动时，守护进程就会等待特定端口上的连接，而不是 USB 上的连接。然后，可以这样连接：

```
$ adb connect 192.168.202.79:7878
connected to 192.168.202.79:7878
```

想要转换回 USB 连接，你可以输入以下命令：

```
$ adb -s 192.168.172.79:7878 usb
restarting in USB mode
```

实际上，这个命令等同于输入了：

```
$ adb -s 192.168.172.79:7878 shell
# setprop service.adb.tcp.port 0
# ps
...
root      66      1      3412    164      ffffffff 00008294 S /sbin/adbd
...
# kill 66
```

在这两种情况下，`adb` 被退出，并由 `init` 自动重新启动它。之后检查 `service.adb.tcp.port` 并相应地启动。如果因为某些原因，你没有一个 USB 连接到你的设备上，可以随时在你的设备上手动预设 `service.adb.tcp.port`，以使 `adb` 总是从该端口号启动。我们将在后面讨论通过全局属性来设置的方式。连接的形式化描述是：

```
adb tcpip <port>
```

## 控制模拟器

正如之前说明的那样，你可以用 `telnet` 来连接每一个模拟器的控制台：

```
$ telnet localhost 5554
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^>'.
Android Console: type 'help' for a list of commands
OK
help
Android console command help:

  help|h|?      print a list of commands
  event          simulate hardware events
  geo            Geo-location commands
  gsm            GSM related commands
  kill           kill the emulator instance
```

```
network      manage network settings
power        power related commands
quit|exit    quit control session
redir        manage port redirections
sms          SMS related commands
avd          manager virtual device state
window       manage emulator window

try 'help <command>' for command-specific help
OK
```

谷歌的在线手册详细介绍了这些命令的使用细节。不幸的是，必须使用 *telnet* 来访问这些命令是很麻烦的，特别是当你想要将你需要做的一部分编写为脚本时。因此，*adb* 可以允许你启动这些像任何其他的命令一样的命令：

```
$ adb -e emu redir add tcp:8080:80
```

这个命令将重定向所有到主机端口 8080 的连接到目标机的 80 端口。*emu* 之后的命令行部分，正是与你通过 *telnet* 会话输入重定向端口的命令一样。

## 探讨 PPP

AOSP 中包含的外部项目之一就是标准 PPP 守护进程，它用在大多数基于 Linux 的发行版中，并在 <https://ppp.samba.org/> 上可以找到。你可以调用 *adb* 来启动一个主机和目标机之间的 PPP 连接。当你的主机和目标机之间仅有 USB 连接时，它可能是用于维护或者是创建主机与目标机之间的网络连接。下面是对 PPP 命令的正式定义：

```
adb ppp <adb service name> [ppp opts]
```

不幸的是，了解如何使用此命令本身是不够的，更糟糕的是，对于所有的 *adb* 命令来说，该命令的文档记录是最贫乏的。你可能使用这个命令更常见的方式是<sup>注4</sup>：

```
adb ppp "shell:pppd nodetach noauth noipdefault /dev/tty" nodetach noauth \
> noipdefault notty <local-ip>:<remote-ip>
```

从本质上讲，这里发生的事情是主机的 *pppd* 守护进程正在用以下参数进行启动：

```
nodetach noauth noipdefault notty <local-ip>:<remote-ip>
```

并且，目标机的 *pppd* 正在被使用以下参数来启动：

```
nodetach noauth noipdefault /dev/tty
```

---

注 4：请注意，这个命令太长而无法在本书的一个单行中显示，因此它是跨行的。第一行末尾的 \ 和第二行开始的 > 在这里只是为了表示换行。

*adb* 随后将代理两个 *pppd* 守护进程之间的通信，因此你可以在主机和目标机之间建立确定的网络连接。你可能需要做更多的调查工作来弄清楚你想要建立什么样的网络连接以及制定什么特定的 IP 参数。但从上述情况来看，你至少有一个良好的起点。我会鼓励你阅读你主机上的 *pppd* 的手册页，以获取关于它的全部功能的更多信息。

我也鼓励你仔细看一看下面这些网络上的文章，这些文章告诉你更多关于怎么使用 *adb* 功能的细节和例子：

- *adb* 上的 PPP（针对于 Linux/unix 用户） (<http://bit.ly/10qD9jM>) 。
- 出现 lsusb + *adb* 但不带 ifconfig 的设备 (<http://bit.ly/WPzHZE>) 。
- Xperia X10 Mini Pro 的 USB 范围 (<http://bit.ly/13JRrgh>) 。
- 通过 USB 连接创建一个我的 Ubuntu 开发机和运行在 Android 上的 BeagleBoard 之间的 PPP 连接 (<http://bit.ly/10qDhzM>) 。

## Android 的命令行

正如我刚才所说的，你很可能遇到的第一款 Android 专用工具之一是 *adb*，以及其最常见的用途之一就是 shell 到目标机上。而且，由于在开发板生产期间，在具有功能性 UI 之前，你可能有相当长的一段时间要使用命令行，它到现在为止只包含了 Android 命令行。事实上，你将可能会直接用 Android 的命令行来处理，也可能是通过一个串行控制台，甚至在 *adb* 之前这就是全部的功能：如果你的设备不具备 USB 功能，或者是还不具有一个功能性的 USB 驱动程序或可用的 TCP/IP 网络接口，就会是这种情况。

## 版本 2.3/ 姜饼及之前使用的 shell 工具

在 Android 版本中直到版本 2.3/ 姜饼采用的都是标准的 shell，它在源代码的 */system/core/sh* 中，其生成的二进制文件在目标的 */system/bin/sh* 中。不同于系统中的许多组件，这个 shell 是 Android 并没有推倒重来的一个。相反，Android 使用 NetBSD 的 sh 工具并只进行了很少的调整。实际上在 AOSP 上保留了 sh 手册页的原样，这样你就可以在你的主机上按下面这样做，来获取有关如何使用 shell 的更多信息：

```
$ man system/core/sh/sh.1
```

很不幸，这个 shell 比 bash 或 BusyBox 的 ash 更简单。比如，它没有完成标签或颜色编码的文件。如果不考虑其他原因，这些限制就是开发人员将 BusyBox 加入到他们的目标机中，至少在开发过程中加入的很好的理由。想要了解关于 UNIX 的 shell 的

一个全面的比较，除去 BusyBox，你可以看看阿尔诺·塔代伊的“shell 选择，一个 shell 的比较”。它的历史可以追溯到 1994 年，但它是少数几个讨论这个话题的文件之一。还有维基百科的比较，但它更浅。

除了比较之外，这里还有一个关于 shell 功能的概述：

- 输出改向用 > 和 <。
- 管道用 |。
- 后台运行命令用 &。
- 脚本编写用 if/then/fi、while/do/done、for/do/done、continue/break 和 case/in/pattern/esac。
- 环境变量。
- 参数扩展 \${...}。
- 命令替换 \$(...)。
- shell 模式 (\*, ?, ! 等)。

表 6-12 描述了 sh 的内置命令。

表 6-12: sh 的内置命令

命令	描述
<i>alias</i>	用一个命令来替代另一个命令
<i>bg</i>	在后台运行一个暂停任务
<i>command</i>	运行指定命令。当一个脚本与一个内置命令具有相同名称时非常有用
<i>cd</i>	换一个目录
<i>eval</i>	评估一个表达式
<i>exec</i>	使用指定的命令来代替运行的 shell
<i>exit</i>	退出 shell 进程
<i>export</i>	输出一个环境变量的值用于所有后续的命令
<i>fg</i>	将后台的作业移到前台
<i>getopts</i>	解析命令行选项
<i>hash</i>	打印出命令在 shell 缓存区的位置
<i>jobid</i>	打印属于作业 ID 的 PID 值
<i>jobs</i>	列出当前运行的作业
<i>pwd</i>	打印出工作目录

表 6-12: sh 的内置命令 (续)

命令	描述
<i>read</i>	从命令行中读取一个变量
<i>readonly</i>	将一个环境变量设置为只读的
<i>set</i>	列出环境变量当前的设置
<i>setvar</i>	将一个环境变量设置为给定的值
<i>shift</i>	向上移动命令行参数 (\$1 变为 \$2 等)
<i>trap</i>	当给定的 UNIX 信号接收到时执行一个行为
<i>type</i>	打印出一个命令或一个别名定义的文件系统位置
<i>ulimit</i>	打印 / 设置进程的范围 (使用 <code>sysctl()</code> )
<i>umask</i>	设置默认文件创建模式
<i>unalias</i>	删除一个给定的别名
<i>unset</i>	删除一个给定的环境变量
<i>wait</i>	等待一个给定的作业完成

如果你正在使用版本 2.3/ 姜饼或其他更低的 Android 版本，我鼓励你看看 sh 的手册页以获取更多关于如何使用它的每一个功能的信息。你也可以从 UNIX 的 shell 脚本的大量在线例子和教程中获得收益。所有这些方面都不是 Android 所独有的，或者并不是仅在嵌入式环境中使用 sh。

## 版本 4.0/ 冰淇淋三明治之后的 shell

从版本 4.0/ 冰淇淋三明治开始<sup>注5</sup>，Android 现在的版本都依赖于 MirBSD Korn Shell。它在主机的 `external/mksh/` 目录中，其二进制文件在目标机的 `/system/bin/mksh` 中。

---

**注意：**尽管 `mksh` 被包含在 4.2/ 果冻豆之前的版本中，但是当建立在仿真器上时，它是被禁用的。有一个构建系统中的 `TARGET_SHELL` 配置变量在默认情况下被设置为 `mksh`。但是，开发板上的配置可以将默认的值改为任何适合开发板上运行的值。在 4.2/ 果冻豆之前的版本中，这个变量被设置为 `ash`，这是取代了上节中所描述的可执行的 `sh` 命令的新名称。

---

<sup>注 5：</sup>这种变化显然是在 3.x 系列版本中做的，但该版本的源代码从来没有被作为正确标记的分支来使用，即使更新的 Android 版本中也会包含这些代码。

*mksh* 比 *sh* 更加强大。例如，它包括标签完成，虽然它不支持颜色编码的文件，并且具有 *bash/ksh93/zsh* 的扩展。它也有一个手册页，你可以通过输入以下命令在主机上看到：

```
$ man system/external/mksh/src/mksh.1
```

鉴于 *mksh* 有比 *sh* 更多的功能和内置命令，因此很难在本书中花更多的篇幅来介绍它。相反，我建议你看一下它的手册页和网站以获取更多的信息。例如，它包括对非常有用的 *history* 命令的实现，该命令列出了之前你输入在 shell 中的命令。

## 工具箱

像其他基于 Linux 的系统一样，Android 的 shell 只提供所要求的最低限度的有功能的命令行。而其余部分的功能则来自于单独的工具，这些工具提供可以单独从 shell 启动的特定功能。正如我们在第 2 章中讨论的那样，在 Android 中提供这些工具的包被称为工具箱，它在 BSD 许可证下发布。工具箱在 AOSP 的 *system/core/toolbox/* 目录下。它生成的二进制文件和符号链接放在实际目标机的 */system/bin* 中。

不幸的是，除了不像 BusyBox 软件那样功能丰富之外，工具箱也严重缺乏文档。幸运的是，它提供的大多数命令已经存在，虽然在标准 Linux 桌面上有功能更全面的形式。因此，你可以用你的开发设备的主页面作为基础来使用等效的工具箱命令。要注意，因为一些工具箱命令的变体与它们在标准 Linux 版本的命令行语义略有不同。

在某些情况下，命令很容易弄清楚，因为如果你给它传递了错误类型的参数，该命令将打印出它的用法。然而，并非所有的工具箱命令都提供在线帮助。在某些情况下，你甚至要深入挖掘工具箱的来源，以找出命令的参数究竟是如何进行处理的，以及该命令实际上在做什么。

### 常见的 Linux 命令

表 6-13 列出了在工具箱中能找到的常见的 Linux 命令。如果你喜欢的命令不在这个列表里，我建议你检查一下 BusyBox，很可能它在那里。我们将在附录 A 中讨论在同一个文件系统中如何与工具箱一同获取到 BusyBox。如果连 BusyBox 中都不包括你要寻找的工具，那么你可以为 Android 编译完整的 Linux 工具，也许可以通过将它导入到 AOSP 的 *external/* 目录中，并基于它现有的构建脚本或 *makefile* 文件为它派生出一个 *Android.mk* 文件。

---

**注意：**为求简洁，我省略了表 6-13 中的命令参数对于每个指令的完整列表。看一下 Linux 的主页来了解它们可能是一个什么的概念。

---

表 6-13：工具箱的常见 Linux 命令

命令	描述
<i>cat</i>	在一个标准输出中查看给定文件的内容
<i>chmod</i>	修改一个文件或目录的访问权限
<i>chown</i>	修改一个文件或目录的所有者
<i>cmp</i>	比较两个文件
<i>date</i>	输出当前日期和时间
<i>dd</i>	当转换和格式化内容时，复制一个文件
<i>df</i>	打印文件系统的磁盘使用情况
<i>dmesg</i>	查看内核的日志缓冲区
<i>hd</i>	将一个文件转储为十六进制格式
<i>id</i>	打印当前用户和组 ID
<i>ifconfig</i>	配置一个网络接口
<i>iftop</i>	实时监控网络流量
<i>insmod</i>	装载一个内核模块
<i>ionice</i>	获取 / 设置一个进程的 I/O 优先级
<i>ln</i>	创建一个符号连接
<i>kill</i>	发送 TERM 信号给进程
<i>ls</i>	列出一个目录内的内容
<i>lsmod</i>	列出当前装载的内核模块
<i>lsof</i>	列出当前打开的文件描述符
<i>mkdir</i>	创建一个目录
<i>mount</i>	打印挂载的文件系统列表或新挂载的列表
<i>mv</i>	给一个文件重命名
<i>netstat</i>	打印网络统计数
<i>printenv</i>	打印所有输出的环境变量
<i>ps</i>	打印正在运行中的进程
<i>reboot</i>	重启系统
<i>renice</i>	修改一个进程的“nice”值
<i>rm</i>	删除一个文件
<i>rmdir</i>	删除一个目录
<i>rmmod</i>	移除一个内核模块

表 6-13：工具箱的常见 Linux 命令（续）

命令	描述
<i>route</i>	打印 / 修改内核路由表
<i>sleep</i>	在给定数量的秒数内休眠
<i>sync</i>	强制将文件系统的缓冲区写入硬盘中
<i>top</i>	实时监测进程
<i>umount</i>	卸载一个文件系统
<i>uptime</i>	打印系统的正常运行时间
<i>vmstat</i>	打印系统的内存使用情况

这些命令中有一些命令的缺点很惹人讨厌。例如，直到版本 4.0/ 冰淇淋，*ls* 还是无法按字母顺序打印目录列表或为文件提供颜色编码，然而这些大多数是 Linux 系统的标准配备。按字母顺序已经被添加，但还是不能进行颜色编码。此外，与其典型的 Linux 或 BusyBox 软件相违背的是，使用 *ifconfig* 命令，如果不带任何参数调用的话，实际上并不能打印出当前网络配置，你必须使用 *netcfg* 来代替。表 6-14 列出了其他的你会在 4.2/ 果冻豆中发现的 Linux 命令。

表 6-14：版本 4.2/ 果冻豆中添加的公共 Linux 命令

命令	描述
<i>cp</i>	复制文件
<i>du</i>	展示文件空间使用情况
<i>grep</i>	在文件中查询字符串
<i>md5</i>	类似 Linux 中的 <i>md5sum</i> 命令，计算文件的 MD5 校验和
<i>touch</i>	更新一个文件的时间戳（如果文件不存在的话创建它）

## 全局属性

在第 2 章介绍过，Android 初始化功能之一是，它保留了一组可以从系统中的任何地方访问的全局属性。当然，工具箱提供了一些工具来与这些全局属性进行接口：

```
getprop <key>
setprop <key> <value>
watchprops
```

你可能想要做的第一件事是列出所有的属性及其当前值：

```
# getprop
[ro.ril.wake_lock_timeout]: [0]
```

```
[ro.secure]: [0]
[ro.allow.mock.location]: [1]
[ro.debuggable]: [1]
[persist.service.adb.enable]: [1]
[ro.factorytest]: [0]
[ro.serialno]: []
[ro.bootmode]: [unknown]
[ro.baseband]: [unknown]
[ro.carrier]: [unknown]
[ro.bootloader]: [unknown]
[ro.hardware]: [am335xevm]
[ro.revision]: [0]
[ro.build.id]: [GRJ22]
[ro.build.display.id]: [beaglebone-eng 2.3.4 GRJ22 eng.karim.20120504.160548
test-keys]
[ro.build.version.incremental]: [eng.karim.20120504.160548]
[ro.build.version.sdk]: [10]
...
...
```

它会打印出超过 100 个，如果没有更多了的话，你的系统设置的全局属性的值。如果你只是想打印出一个值，你可以这样做：

```
# getprop ro.hardware
am335xevm
```

你也可以直接从命令行中设置全局属性：

```
# setprop acme.birdradar.enable 1
# getprop acme.birdradar.enable
1
```

一旦一个属性被设置，可以使用 *setprop* 命令再次改变它的值。但是你不能删除你使用 *setprop* 命令“创造”的属性。不过该属性将在下次重新启动时消失，除非它的名字是以 *persist* 开始的。在这种情况下，带有属性全名的文件将在包含了属性的值的 */data/property* 文件夹中被创建。要删除此属性，你将需要删除这个文件或破坏数据分区。

你还可以实时监控被修改的属性，假设 *acme.bird radar.enable* 是在 *watchprop* 启动后被设置的：

```
# watchprops
946709853 acme.birdradar.enable = '1'
```

## 输入事件

Android 在很大程度上依赖于 Linux 的输入层，以获得用户的输入事件。正如我们在第 2 章所看到的，Linux 的输入层的设备都在 */dev/input* 目录下，它们是 Android 输入支持的基础。每当用户触摸或滑动屏幕或触摸设备上任何按钮时，都会产生一个事件。

虽然 Android 的系统服务器已经适当地处理了这些事件，但你可能想要么遵守或者是生成自己的事件。工具箱可以让你做到这一点：

```
getevent [-t] [-n] [-s <switchmask>] [-S] [-v [<mask>]] [-p] [-q] [-c <count>]
[-r]<device>
-t: show time stamps
-n: don't print newlines
-s: print switch states for given bits
-S: print all switch states
-v: verbosity mask (errs=1, dev=2, name=4, info=8, vers=16, pos. events=32)
-p: show possible events (errs, dev, name, pos. events)
-q: quiet (clear verbosity mask)
-c: print given number of events then exit
-r: print rate events are received
sendevent <device> <type> <code> <value>
```

想要观察这些事件，你可以输入如下的命令：

```
# getevent
/dev/input/event0: 0003 0000 0000007d
/dev/input/event0: 0003 0001 0000001b
/dev/input/event0: 0001 014a 00000001
/dev/input/event0: 0000 0000 00000000
/dev/input/event0: 0001 014a 00000000
/dev/input/event0: 0000 0000 00000000
/dev/input/event0: 0001 0066 00000001
/dev/input/event0: 0001 0066 00000000
...
...
```

*getevent* 连续显示产生的事件，直到你按下 Ctrl-C 为止。输出格式是事件类型、事件代码和事件的值。这可让你验证你的驱动程序是否上报了相应的信息反馈给 Android。

类似的方式，如果你想监控 Android 处理的事件，你可以像下面这样自己发送事件：

```
# sendevent /dev/input/event0 1 330 1
```

请注意，如果你同时运行 *getevent*，你会看到这个新的事件：

```
/dev/input/event0: 0001 014a 00000001
```

换句话说，这是因为 *getevent* 的输出是十六进制，*sendevent* 的输入是十进制。

## 控制服务

正如我们在第 2 章中所看到的那样，Android 的初始化因为不同的目的，启动了一些本地的守护进程。通常情况下，它们在初始化配置脚本中被叫做服务（service），初始化的“服务”与系统服务或提供给应用程序开发者的服务组件都无关。正如我们很

快就会看到那样，这些服务可以自动启动或标记为禁用。无论哪种方式，你都可以按下面的方式开始或停止服务：

```
start <servicename>
stop <servicename>
```

这些命令都不产生任何输出。有些遗憾的是，我们没有办法得到 Android 的正在运行的服务列表。相反，你需要足够了解初始化的配置脚本，能够知道哪些服务你可以启动和停止。举例来说，如果想要停止所有的系统 Java 组件，你可以这样做：

```
# stop zygote
```

请注意，这个特定的命令是一个非常激烈的措施，因为它会停止所有的应用程序并杀死系统服务。但在某些情况下，它可能正是你要寻找的。比如说，假设你想要阻止系统服务去访问一个给定的驱动程序，因为该驱动程序已经正常停止了，并且你想要在系统不再继续使用它时在上面运行一些诊断。

我们将在下一节中介绍 Android 的初始化，及其对服务的处理。

## 日志

另一个有趣的工具箱的功能是，它能够让你将自己的事件添加到 Android 的日志中：

```
log [-p <prioritychar>] [-t <tag>] <message>
prioritychar should be one of:
v,d,i,w,e
```

比如说：

```
# log -p i -t ACME Initiating bird tracking sequence
```

现在，如果你用 *logcat* 命令来查看日志，可以看见如下的打印信息：

```
# logcat
...
I/ACME ( 336): Initiating bird tracking sequence
...
```

如果你的 shell 脚本与 Android 软件开发包的其余部分一起执行，这会非常有用。另外，如果你使用 Android 的日志功能得到了自定义的代码，这里指的是在一个应用程序中或自定义的系统服务中，你就可以看到在这里生成的事件，以及从脚本或命令行中手动生成的事件的相关顺序。

## ioctl

正如我们在第 2 章所讨论的那样，设备表现为 `/dev` 中的条目。如果熟悉 Linux 的驱动程序模式，你就知道如果一个设备是由一个字符设备驱动程序控制的，那么只需打开 `/dev` 中该设备的目录，它的读取 / 写入也只需要调用它的 `read()`/`write()` 函数即可。所以，你想要读取一个字符设备，你可以像这样来做：

```
# cat /dev/birdlocator0
```

同样，你也可以像下面一样来写一个字符设备：

```
# echo "Fire" > /dev/birdlaser0
```

另一个在字符设备中使用的非常重要的文件操作是 `ioctl()`。然而，没有标准的 Linux 工具来调用该操作，因为它是由驱动指定的。然而，在嵌入式系统中，那些对系统的操作通常要么是驱动作者本身，要么是与它们密切合作的，因此有一个工具来使开发者能够调用驱动程序的 `ioctl()` 函数是很有意义的。工具箱提供了这一点：

```
ioctl [-l <length>] [-a <argsize>] [-rdh] <device> <ioctlnr>
-l <length> Length of io buffer
-a <argsize> Size of each argument (1-8)
-r          Open device in read only mode
-d          Direct argument (no iobuffer)
-h          Print help
```

很显然，这样的使用是特定于驱动程序的。你需要参考你的驱动程序文档和 / 或源代码，来确切地知道你需要传递给该命令什么参数，对它们将会有什么影响。

---

**警告：**`ioctl()` 是一个非常强大的驱动程序操作。使用它可以从良性状态报告到彻底的硬件毁灭状态。你要确保你了解，你即将在你指定的设备上发起的是什么特定的 I/O 控制操作。你可能只想在你写的驱动程序使用它。

---

## 擦除设备

在某些极端的情况下，擦除 Android 设备上的数据是有必要的。这种极端的和不可逆的操作，可以使用工具箱中的 `wipe` 命令来实现：

```
wipe <system|data|all>
```

system 表示 '/system'  
data 表示 '/data'

如果需要删除系统上的所有数据，你可以这样做：

```
# wipe data
Wiping /data
Done wiping /data
```

我敢肯定，你明白在这里的操作是不能够“撤销”的，所以你要小心一点。比如说，当你的设备上有敏感数据或二进制文件时，你可能想用这个操作作为故障保护，当你检测到有对关键系统部分的未经授权的访问的情况下，摧毁这些数据或文件。

## 其他 Android 特有的命令

工具箱还包括一些其他的 Android 特有的命令，我们简要地介绍一下，因为它们的用途要么很明显要么非常有限。

**nandread**。该工具用于读取 NAND 闪存设备的内容到一个文件中：

```
nandread [-d <dev>] [-f <file>] [-s <size>] [-vh]
-d <dev>    Read from <dev>
-f <file>    Write to <file>
-s <size>    Number of spare bytes in file (default 64)
-R          Raw mode
-S <start>   Start offset (default 0)
-L <len>     Length (default 0)
-v          Print info
-h          Print help
```

**newfs\_msdos**。这个命令允许你将设备格式化为 VFAT 文件系统：

```
newfs_msdos [ -options ] <device> [<disktype>]
where the options are:
-@ create file system at specified offset
-B get bootstrap from file
-C create image file with specified size
-F FAT type (12, 16, or 32)
-I volume ID
-L volume label
-N don't create file system: just print out parameters
-O OEM string
-S bytes/sector
-a sectors/FAT
-b block size
-c sectors/cluster
-e root directory entries
-f standard format
-h drive heads
-i file system info sector
-k backup boot sector
-m media descriptor
-n number of FATs
-o hidden sectors
-r reserved sectors
-s file system size (sectors)
-u sectors/track
```

`newfs_msdos` 是 `vold` 守护进程用来将设备格式化为 VFAT 的工具；`vold` 本身被挂载系统服务所使用，用于管理挂载的设备。

**notify**。此命令使用 inotify 系统调用一个 API 来监控要修改的目录或文件：

```
notify [-m <eventmask>] [-c <count>] [-p] [-v <verbosity>] <path> [<path> ...]
```

r。在版本 4.2/ 果冻豆中，你还会发现一个 `r` 命令。这是重复你之前输入到 shell 上的一条命令的简写。因此，你可以不通过按向上的箭头，然后回车，而是只输入 `r` 来执行之前输入的命令。下面是一个简单的例子：

```
root@android:/ # ls -l /proc/cpuinfo
-rw-r--r-- root      root          0 2013-01-19 10:34 cpuinfo
root@android:/ # r
ls -l /proc/cpuinfo
-rw-r--r-- root      root          0 2013-01-19 10:34 cpuinfo
```

**schedtop**。类似 `top`，`schedtop` 用于连续的、实时监控内核调度。不同于 `top` 仅报告了每个进程的实时的 CPU 使用百分比，该命令连续地报告了每个进程的累积执行时间：

```
schedtop [-d <delay>] [-bitamun]
-d refresh every <delay> seconds
-b batch - continuous prints instead of refresh
-i hide idle tasks
-t show threads
-a use alternate screen
-m use millisecond precision
-u use microsecond precision
-n use nanosecond precision
```

---

注意：这里给出的命令描述源于我所读的工具箱的源码。`schedtop` 本身不提供任何在线帮助，也没有关于其使用的任何文件。

---

**setconsole**。这个命令可以让你切换控制台：

```
setconsole [-d <dev>] [-v <vc>] [-gtncpoh]
-d <dev>   Use <dev> instead of /dev/tty0
-v <vc>    Switch to virtual console <vc>
-g         Switch to graphics mode
-t         Switch to text mode
-n         Create and switch to new virtual console
-c         Close unused virtual consoles
-p         Print new virtual console
-o         Print old virtual console
-h         Print help
```

**smd**。所有工具箱的命令中，这一个是最“神秘的”。我艰难地花费了很多精力才找

出关于 *smd* 的使用或实际使用例子的有用信息。看来，在确定的设备中，基带处理器显示为 */dev/smdN* 中的一个。此后，该工具允许你发送 AT 指令到基带处理器：

```
smd [<port>] <commands>
```

## 内核本地工具和守护进程

正如我在第 2 章中提到的那样，Android 有大约 150 个工具分布在它的文件系统中。在本章中，我们将通过独立的 Java 框架和服务来介绍它们。具体而言，我们将重点在这一节主要介绍那些在 */system/bin* 文件夹中的工具，该文件夹可以认为是 Android 的核心。也有一些工具在 */system/xbin* 中，但它们对于系统的正确运行来说不是必要的。

我们已经看到了在标准的 Linux 系统中工具箱的很多功能是如何实现的，也看到了其在 Android 中特有的功能。同样，核心的 Android 工具和守护进程也有两大类，其中一些是从外部项目中派生出来的，还有一些是 Android 特有的。表 6-15 列出了几个从 *external/* 目录的项目中编译出来的核心工具和守护进程。

表 6-15：外部项目的内核工具和守护进程

工具 / 守护进程	外部项目	原始位置
<i>bluetoothd</i> 、 <i>sdptool</i> 、 <i>BlueZ</i> <sup>a</sup> <i>avinfo</i> 、 <i>hciconfig</i> 、 <i>hctool</i> 、 <i>l2ping</i> 、 <i>hciattach</i> 和 <i>rfcomm</i>		<a href="http://www.bluez.org/">http://www.bluez.org/</a>
<i>dbus-daemon</i>	D-Bus	<a href="http://dbus.freedesktop.org">http://dbus.freedesktop.org</a>
<i>dnsmasq</i>	Dnsmasq	<a href="http://www.thekelleys.org.uk/dnsmasq/">http://www.thekelleys.org.uk/dnsmasq/</a>
<i>dhpcd</i> 和 <i>showlease</i>	<i>dhpcd</i>	<a href="http://roy.marples.name/projects/dhpcd/">http://roy.marples.name/projects/dhpcd/</a>
<i>fsck_msdos</i>	NetBSD <i>fsck_msdos</i>	<a href="http://cvsweb.netbsd.org/bsdweb.cgi/src/sbin/fsck_msdos/">http://cvsweb.netbsd.org/bsdweb.cgi/src/sbin/fsck_msdos/</a>
<i>gdbserver</i>	GNU 调试器	<a href="http://www.gnu.org/software/gdb/">http://www.gnu.org/software/gdb/</a>
<i>gzip</i>	gzip 工具	<a href="http://www.gzip.org/">http://www.gzip.org/</a>
<i>iptables</i>	Netfilter	<a href="http://www.netfilter.org/">http://www.netfilter.org/</a>
<i>ping</i>	iutils	<a href="http://www.skbuff.net/iutils/">http://www.skbuff.net/iutils/</a>
<i>pppd</i>	PPP	<a href="http://ppp.samba.org/">http://ppp.samba.org/</a>
<i>raccoon</i>	IPsec-Tools	<a href="http://ipsec-tools.sourceforge.net/">http://ipsec-tools.sourceforge.net/</a>
<i>tc</i>	<i>iproute2</i>	<a href="http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2">http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2</a>

表 6-15：外部项目的内核工具和守护进程（续）

工具 / 守护进程	外部项目	原始位置
wpa_supplicant 和 wpa_cli	WPA 客户端	<a href="http://hostap.epitest.fi/wpa_supplicant/">http://hostap.epitest.fi/wpa_supplicant/</a>

a. 从版本 4.2/ 果冻豆开始不再是 Android 的一部分。

并非所有的这些工具对于你系统的运行都是必需的。比如说，如果你的嵌入式系统并不支持 WiFi 或蓝牙，那么就没有必要有 *wpa\_supplicant* 或者是任何的 BlueZ 工具及守护进程。事实上，在这些特殊情况下，二进制文件并不会构建，除非开发板专用的 *.mk* 文件需要它。记住在版本 4.2/ 果冻豆中，BlueZ 已经被换成了另一个栈。

我们在接下来的小节中看一下核心的 Android 专用的工具和守护进程。实际上它们中的很多并不会被你在命令行中直接调用，但它们会被自动调用，而不是由系统的一部分或其他来实现。然而，有一些还是值得掌握的。

## logcat

也许这是你在 Android 里最常使用的一个命令，*logcat* 允许你在覆盖 *adb* 时打印出 Android 日志的缓冲区，就像我们之前看到的那样。以下是 *logcat* 的完整的在线帮助：

```
# logcat --help
Usage: logcat [options] [filterspecs]
options include:
  -s           Set default filter to silent.
               Like specifying filterspec '*:s'
  -f <filename> Log to file. Default to stdout
  -r [<kbytes>] Rotate log every kbytes. (16 if unspecified). Requires -f
  -n <count>   Sets max number of rotated logs to <count>, default 4
  -v <format>   Sets the log print format, where <format> is one of:
               brief process tag thread raw time threadtime long

  -c           clear (flush) the entire log and exit
  -d           dump the log and then exit (don't block)
  -t <count>   print only the most recent <count> lines (implies -d)
  -g           get the size of the log's ring buffer and exit
  -b <buffer>   request alternate ring buffer
               ('main' (default), 'radio', 'events')
  -B           output the log in binary
filterspecs are a series of
  <tag>[:priority]

where <tag> is a log component tag (or * for all) and priority is:
  V  Verbose
  D  Debug
  I  Info
  W  Warn
```

```
E      Error
F      Fatal
S      Silent (suppress all output)

'*' means '*:d' and <tag> by itself means <tag>:v
If not specified on the commandline, filterspec is set from ANDROID_LOG_TAGS.
If no filterspec is found, filter defaults to ':I'

If not specified with -v, format is set from ANDROID_PRINTF_LOG
or defaults to "brief"
```

使用这个帮助你应该能够了解大部分 *logcat* 的复杂情况，第 2 章也对 Android 日志进行了解释。例如，可以使用 -b 标志，选择你想要打印出哪个缓冲，默认值是 `main`。还可以设置 `ANDROID_LOG_TAGS` 环境变量来提供一个默认的输出滤波器。尽管如此，*logcat* 的一个更令人困惑的方面是过滤能力。事实上，在线帮助似乎表明，仅仅在命令之后指定一个 `<tag>[:priority]`，就足以将输出限制到属于 `tag` 的范围内。但这是不行的：

```
# logcat ActivityManager
----- beginning of /dev/log/main
I/DEBUG    ( 59): debuggerd: Mar 27 2012 05:30:39
----- beginning of /dev/log/system
I/Vold     ( 57): Vold 2.1 (the revenge) firing up
D/Vold     ( 57): USB mass storage support is not enabled in the kernel
D/Vold     ( 57): usb_configuration switch is not enabled in the kernel
D/Vold     ( 57): Volume sdcard state changing -1 (Initializing) -> 0 (No-Media)
)
D/Vold     ( 57): Volume usb state changing -1 (Initializing) -> 0 (No-Media)
D/Vold     ( 57): Volume sdcard state changing 0 (No-Media) -> 2 (Pending)
D/Vold     ( 57): Volume sdcard state changing 2 (Pending) -> 1 (Idle-Unmounted)
)
I/Netd     ( 58): Netd 1.0 starting
D/AndroidRuntime( 61):
D/AndroidRuntime( 61): >>>>> AndroidRuntime START com.android.internal.os.Zyg
oteInit <<<<<
D/AndroidRuntime( 61): CheckJNI is ON
D/dalvikvm( 61): creating instr width table
...
...
```

很显然，我们看到的是所有标签的输出，而不仅仅是一个匹配 `Activity Manager` 的输出。技巧是使用 -s 标志：

```
# logcat -s ActivityManager
----- beginning of /dev/log/main
----- beginning of /dev/log/system
I/ActivityManager( 128): Memory class: 16
I/ActivityManager( 128): Config changed: { scale=1.0 imsi=0/0 loc=md_US touch=1
keys=1/1/2 nav=1/1 orien=2 layout=268435491 uiMode=0 seq=1}
I/ActivityManager( 128): System now ready
I/ActivityManager( 128): Start proc com.android.systemui for service com.androi
d.systemui/.StatusBarService: pid=245 uid=1000 gids={3002, 3001, 3003}
```

```
I/ActivityManager( 128): Config changed: { scale=1.0 imsi=0/0 loc=md_US touch=1  
keys=1/1/2 nav=1/1 orien=2 layout=268435491 uiMode=17 seq=2}  
I/ActivityManager( 128): Start proc com.android.inputmethod.latin for service c  
om.android.inputmethod.latin/.LatinIME: pid=247 uid=10016 gids={}  
W/ActivityManager( 128): Unable to start service Intent { act=@0 }: not found  
W/ActivityManager( 128): Unable to start service Intent { act=@0 }: not found  
...
```

不幸的是，*logcat* 的在线帮助并没有指出这一点。

## netcfg

除了工具箱的 *ifconfig*，Android 还有另一个工具可以让你操纵网络接口：

```
netcfg [<interface> {dhcp|up|down}]
```

令人困惑的是，*netcfg* 和 *ifconfig* 的功能有些重叠。比如说，两者都可以建立或撤销接口。不过，*netcfg* 可以初始化 DHCP 客户端请求，并打印出当前接口的配置，而 *ifconfig* 两者都做不到。另一方面，*ifconfig* 命令可以设置一个接口的静态 IP 地址和子网掩码，而 *netcfg* 做不到这一点。

大多数情况下，*netcfg* 在打印出接口的配置信息方面是非常有用的：

```
# netcfg  
lo      UP    127.0.0.1        255.0.0.0        0x00000049  
etho    UP    10.0.2.15       255.255.255.0    0x00001043  
tunlo   DOWN  0.0.0.0        0.0.0.0          0x00000080  
greo    DOWN  0.0.0.0        0.0.0.0          0x00000080
```

## debuggerd

这个守护进程实际上是由 *init* 在启动初始时就启动了。它打开了 android:debuggerd 抽象 UNIX 域套接字<sup>注6</sup> 并等待连接。它保持休眠状态，直到用户空间进程崩溃。它是由仿真器的链接器激活，这个链接器设置了处理崩溃事故的信号处理程序，并在这种情况下连接到 *debuggerd*。然后 *debuggerd* 做两件事：在 */data/tombstones* 文件夹中创建一个 *tombstone* 文件；如果需要的话，可以通过 *gdbserver* 进行事后调试工作。

你不需要为 *tombstone* 文件的生成做什么特别事情。它们会被自动创建，并包含崩溃进程相关的信息，以便于在事后分析时有所发现。这里有一段我的 BeagleBone 上经常崩溃的 VNC 服务器上的信息：

```
# cat /data/tombstones/tombstone_06
```

---

注 6：与典型的 UNIX 域套接字在文件系统中以条目的形式出现不同，抽象的套接字在文件系统中是不可见的。

```

*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***

Build fingerprint: 'TI/beaglebone/beaglebone:2.3.4/GRJ22/eng.karim.20120504.1605
48:eng/test-keys'

pid: 4656, tid: 4656 >>> androidvncserver <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr deadbaad
r0 00000027 r1 deadbaad r2 a0000000 r3 00000000
r4 00000001 r5 00000000 r6 00069ad8 r7 0005e000
r8 00069cd8 r9 00000000 10 000003e8 fp 00000001
ip afd46668 sp beep4bd0 lr afd191d9 pc afd15ca4 cpsr 60000030
d0 2e302e302e373220 d1 206f742074636567
d2 0000000000000006f d3 0000000000000006e
...
#00 pc 00015ca4 /system/lib/libc.so
#01 pc 00013614 /system/lib/libc.so
#02 pc 000144da /system/lib/libc.so
#03 pc 00010290 /system/bin/androidvncserver
#04 pc 00010296 /system/bin/androidvncserver
#05 pc 0000fcbe /system/bin/androidvncserver
#06 pc 0000bc66 /system/bin/androidvncserver
#07 pc 0000a87e /system/bin/androidvncserver
#08 pc 00014b52 /system/lib/libc.so

code around pc:
afdf15c84 2c006824 e028d1fb b13368db c064f8df
afdf15c94 44fc2401 4000f8cc 49124798 25002027
afdf15ca4 f7f57008 2106ec7c edd8f7f6 460aa901
afdf15cb4 f04f2006 95015380 95029303 e93ef7f6
afdf15cc4 462aa905 f7f5e94a 2106ec68

code around lr:
afdf191b8 4a0e4b0d e92d447b 589c41f0 26004680
afdf191c8 686768a5 f9b5e006 b113300c 47c04628
afdf191d8 35544306 37fff117 6824d5f5 dief2c00
afdf191e8 e8bd4630 bf0081f0 00028344 ffffff88
afdf191f8 b086b570 f602fb01 9004460c a804a901

stack:
beeb4b90 0005e008
beeb4b94 6f000001
beeb4b98 6f2e6772
beeb4b9c 7069616e
beeb4ba0 afd4270c
beeb4ba4 afd426b8
beeb4ba8 00000000
beeb4bac afd191d9 /system/lib/libc.so
...

```

此外，如果设置的一些 UID 的 debug.db.uid 比崩溃进程的更大（只是用一个大的整数，如  $32767 [2^{15} - 1]$ ），debuggerd 之后会使用 ptrace() 系统调用来附加到死亡程序上，让你能够启动 gdbserver 里进行控制。下面是我在我的 BeagleBone 上这样做的时候，debuggerd 打印出来的日志：

```

I/DEBUG      ( 59): ****
I/DEBUG      ( 59): * Process 4656 has been suspended while crashing. To

```

```
I/DEBUG    ( 59): * attach gdbserver for a gdb connection on port 5039:  
I/DEBUG    ( 59): *  
I/DEBUG    ( 59): *      adb shell gdbserver :5039 --attach 4656 &  
I/DEBUG    ( 59): *  
I/DEBUG    ( 59): * Press HOME key to let the process continue crashing.  
I/DEBUG    ( 59): *****
```

一旦 *gdbserver* 连接到死亡程序，你就可以使用作为 AOSP 的 *prebuilt/* 目录一部分的 *aem-eabigdb* 调试器中的一个，来依附到目标机上运行的 *gdbserver*，并对死亡程序进行调试。

## 其他 Android 特有的内核工具和守护进程

此外，你应该知道还有一些其他的内核工具和守护进程，虽然你可能不太会非常频繁地使用它们。

`check_prereq`。它可以让你检查当前运行的版本是否比给定的时间戳更旧：

```
# check_prereq 1336847591  
current build time: [1336162137] new build time: [1336847591]
```

这主要是用于升级的，你可以从 *adb* 中调用这个命令来查看当前的构建版本比运行在你的设备上的版本更旧还是更新。构建时间存储在 *system/* 分区里的 *build.prop* 文件的 *ro.build.date.utc* 全局属性中。

`linker`。这是仿真器的动态链接。你永远不需要手动调用此方法。每当仿真器链接的二进制文件执行时它会自动加载，它的任务就是加载该二进制文件需要的所有库文件。作为 GNU 工具链一部分的 *readelf* 工具，更深入地展示了在此过程中发生的事情：

```
$ arm-eabi-readelf -a logcat  
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: EXEC (Executable file)  
  Machine: ARM  
  Version: 0x1  
  Entry point address: 0x8ed0  
  Start of program headers: 52 (bytes into file)  
  Start of section headers: 13020 (bytes into file)  
  Flags: 0x5000000, Version5 EABI  
  Size of this header: 52 (bytes)  
  Size of program headers: 32 (bytes)  
...  
Program Headers:  
  Type          Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
```

```

PHDR          0x000034 0x00008034 0x00008034 0x000e0 0x000e0 R 0x4
INTERP        0x000114 0x00008114 0x00008114 0x00013 0x00013 R 0x1
    [Requesting program interpreter: /system/bin/linker] ①
LOAD          0x0000000 0x00008000 0x00008000 0x02470 0x02470 R E 0x1000
LOAD          0x003000 0x0000b000 0x0000b000 0x001cc 0x00608 RW 0x1000
DYNAMIC       0x003020 0x0000b020 0x0000b020 0x000c8 0x000c8 RW 0x4
GNU_STACK     0x0000000 0x00000000 0x00000000 0x00000 0x00000 RW 0
EXIDX         0x002410 0x0000a410 0x0000a410 0x00060 0x00060 R 0x4
...
Dynamic section at offset 0x3020 contains 25 entries:
Tag           Type                Name/Value
0x00000003 (PLTGOT)           0xb0fc
0x00000002 (PLTRELSZ)         376 (bytes)
...
0x00000001 (NEEDED)           Shared library: [liblog.so] ②
0x00000001 (NEEDED)           Shared library: [libc.so]
0x00000001 (NEEDED)           Shared library: [libstdc++.so]
0x00000001 (NEEDED)           Shared library: [libm.so]
...

```

- ① 这是二进制文件请求的链接。
- ② 这些都是必须被链接加载的库。

当然 `readelf` 的输出除了上面的以外还有更多，这些告诉你 `logcat` 的“程序解释器”是 `/system/bin/linker`，它需要以下库：`liblog.so`、`libc.so`、`libstdc+++.so` 和 `libm.so`。

`logwrapper`。该命令允许你运行另一个命令，并重定向它的标准输出（`stdout`）和标准错误（`stderr`）到 Android 日志中：

```
logwrapper [-x] <binary> [<args> ...]
```

在这种情况下使用的日志标签与 `binary` 的名字采用相同的字符串。当 `binary` 终止时，使用 `-x` 选项使 `logwrapper` 生成一个段错误（`SIGSEGV`），且错误地址是现有的二进制文件的 `wait()` 系统调用返回的状态。

`run-as`。允许你运行一个二进制文件，就好像它是伴随一个应用程序包的相关权限一起运行的：

```
run-as <package-name> <command> [<args>]
```

该命令将从与 `package-name` 相关联的目录中运行，该目录在带有应用程序 `UID/GID` 的 `/data/data` 的目录中。

`sdcard utility`。该工具使用 Linux 的用户空间文件系统（Filesystem in User Space，FUSE）中的 Linux 的文件系统，来在文件系统的任何目录下，仿真你会在所有 FAT 格式的 SD 卡中发现的权利和权限：

```
sdcard <path> <uid> <gid>
```

换句话说，指定目录下的文件和目录都将是可执行的，正如你所期望的 FAT 那样。`path` 所提供的路径将被挂载到 `/mnt/sdcard` 上。虽然 `sdcard` 命令必须由根用户下发，但它会作为 `uid/gid` 运行。这对于那些实际上并不具备可移动 SD 卡的设备是非常有用。在这种情况下，“外部”存储使用 `sdcard` 命令在“内部”的存储中仿真。

## 额外的本地工具和守护进程

Android 还配备了一定数量的额外工具和守护进程，它们本质上对系统运行不是必要的。它们中的大部分是在文件夹 `/system/xbin` 中，在某些情况下，它们可能会对你有用。表 6-16 和表 6-17 列出了这些实用程序和守护进程。

表 6-16：外部项目的额外工具和守护进程

工具 / 守护进程	外部项目	原始位置
<code>dbus-monitor</code> 和 <code>dbus-send</code>	D-Bus	<a href="http://dbus.freedesktop.org">http://dbus.freedesktop.org</a>
<code>ssh</code> 和 <code>scpnc</code>	Dropbear	<a href="http://matt.ucc.asn.au/dropbear/">http://matt.ucc.asn.au/dropbear/</a>
<code>nc</code>	Netcat	<a href="http://nc110.sourceforge.net/">http://nc110.sourceforge.net/</a>
<code>skia_text</code>	skia 2D 图形库	<a href="http://code.google.com/p/skia/">http://code.google.com/p/skia/</a>
<code>sqlite3</code>	SQLite	<a href="http://www.sqlite.org/">http://www.sqlite.org/</a>
<code>strace</code>	strace 工具	<a href="http://sourceforge.net/projects/strace/">http://sourceforge.net/projects/strace/</a>
<code>tcpdump</code>	tcpdump 工具	<a href="http://www.tcpdump.org/">http://www.tcpdump.org/</a>
<code>netperf</code> 和 <code>netserver</code>	netperf	<a href="http://www.netperf.org/netperf/">http://www.netperf.org/netperf/</a>
<code>oprofiled</code> 和 <code>opcontrol</code>	OProfile	<a href="http://oprofile.sourceforge.net/">http://oprofile.sourceforge.net/</a>

表 6-17：额外的 Android 特有的工具和守护进程

工具 / 守护进程	描述
<code>cpueater</code> 和 <code>daemonize</code>	<code>cpueater</code> 做了一个 <code>while(1)</code> 循环，尽可能的占用 CPU 资源， <code>daemonize</code> 允许你在后台将它以守护进程来运行
<code>crasher</code>	该工具与 <code>debuggerd</code> 可打包在一起，基本上模拟了一个崩溃的过程
<code>directiotest</code>	提供块设备的挂载，在块设备上做写 / 回读测试来测试它
<code>latencytop</code>	提供每个过程的潜在信息
<code>librank</code>	为映射到所有进程内存中的每个对象打印内存使用信息。这包括库和内存映射设备和地区
<code>procmem</code>	为一个运行中的 PID 的每一部分打印内存使用信息
<code>procrank</code>	内存使用排名的进程

表 6-17：额外的 Android 特有的工具和守护进程（续）

工具 / 守护进程	描述
<code>schedtest</code>	调度测试，看看它在要求的 1ms 休眠时间里及时唤醒任务有多可靠
<code>showmap</code>	打印出一个进程的内存映射
<code>showslab</code>	打印出 slab 分配器上的信息
<code>su</code>	允许 root 用户更改他的 UID/GID
<code>timeinfo</code>	报告实时数据，正常运行时间，清醒时间的百分比和休眠时间百分比到标准输出中

## 框架工具和守护进程

除了刚刚介绍的工具和守护程序之外，Android 还包含很多其他的紧紧地与系统服务和 Android 框架绑定的工具和守护进程，如 *ServiceManager*、*installd* 和 *dumpsyst*。我们将在下一章进行讨论。

## 初始化

系统最重要的任务之一就是，一旦内核完成初始化设备驱动程序及其本身的内部结构之后，要初始化用户空间环境。正如我们在第 2 章讨论的那样，这是 *init* 进程的工作，它被内核启动。而且正如我们之后会讨论的，Android 有它自己的自定义的 *init*，并带有自己特定的功能。我们已经介绍了系统启动后可用于本地用户空间的部分，现在让我们来仔细看看负责启动这一切的整个过程。

## 工作原理

图 6-4 显示了 *init* 是如何与 Android 组件的其余部分相整合的。在内核启动它之后，它会读取它的配置文件，打印引导标志或文字到屏幕上，为它的属性服务打开一个套接字，并启动所有支撑整个 Android 用户空间的守护进程和服务。当然对于具体每一步来说还有很多步骤。

### Android 的 *init* 与“普通”*init*

在一个典型的 Linux 系统中，*init* 的作用将仅限于启动守护进程，但是，如果仅仅看它的属性服务，Android 的 *init* 是特别的。但是就像其他任何 Linux 的 *init* 一样，Android 的 *init* 也希望永远不会死去。正如我们前面所讨论的那样，*init* 程序是第一个被内核启动的进程，正因为如此，它的 PID 始终为 1。如果它一旦死去，内核将会崩溃。

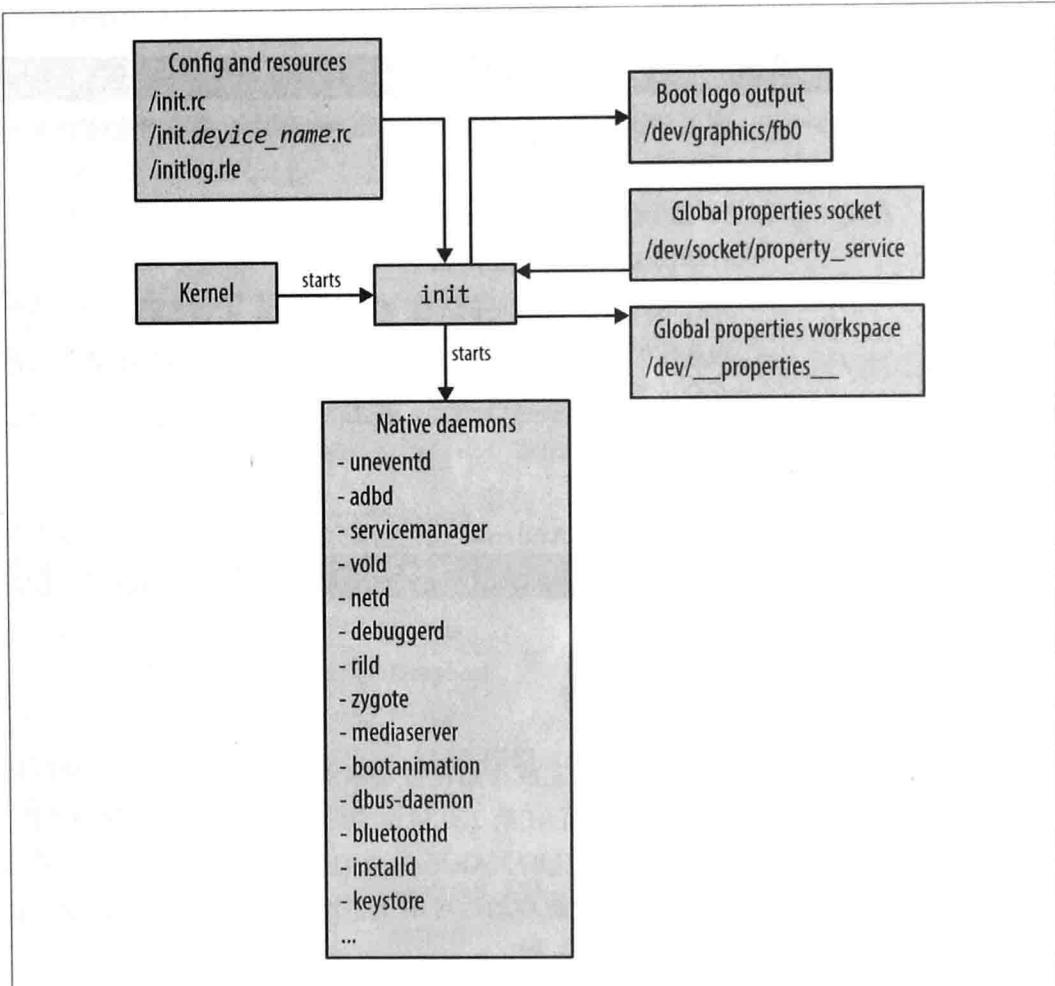


图 6-4: Android 的初始化

`init` 首先所做的事情中的一件就是检查它是否被调用为 `ueventd`。正如我在第 2 章中提到那样，`init` 包括一个 `udev` 热插拔事件处理程序的实现。因为这段代码被编译在 `init` 自己的代码中，`init` 会检查被用来调用它的命令行，如果它是通过 `/sbin/ueventd` 调用符号链接到 `/init` 上的，那么 `init` 会立即作为 `ueventd` 运行。

`init` 接下来会做的事情就是创建和挂载 `/dev`、`/proc` 和 `/sys`。这些目录和它们的条目对于 `init` 接下来会做的事情是至关重要的。之后 `init` 会读取 `/init.rc` 和 `/init.<device_name>.rc` 文件，解析其内容转换成其内部的结构，并基于其配置文件和内置规则混合系统来初始化系统。我们将在下节中更详细的讨论这个部分。

一旦所有的初始化完成后，`init` 会进入一个无限循环，它可能会重新启动已经退出的

但需要重新启动的所有服务，然后轮询其处理的文件描述符，比如说属性服务的套接字，以能够处理需要被处理的任何输入。这就是 *setprop* 属性设定请求是如何被服务的。

## 配置文件

控制 *init* 的行为的主要方式是通过它的配置文件来控制。鉴于 Android 有自己的 *init*，对于这些配置文件有太多的话要讲。让我们先看看它们的位置和语义。然后，将讨论主 *init.rc* 文件和开发板定制的配置文件。

### 位置

对于 *init*，所有东西的主要位置是根目录 (/)。在这里，你会找到实际的 *init* 二进制文件本身和它的两个配置文件：*init.rc* 和 *init<device\_name>.rc*。第一个文件的名字是固定的，而第二个文件的名称取决于硬件。

从本质上讲，*<device\_name>* 是从 */proc/cpuinfo* 中提取的。在本章前面，我们使用 *adb shell* 来为 BeagleBone 转存该文件的内容。在转存过程中，你会发现有一行是以 *Hardware* 开始的。这一行的内容是由 *init* 解析来检索 *<device\_name>*。在 BeagleBone 的情况下，这是 *am335xevm*，并且在仿真器的情况下，它是 *goldfish*。

---

注意：在最终的 *init.<device\_name>.rc* 被从磁盘获取前，*Hardware* 旁边显示的字符串被转换成小写字母。因此，虽然模拟器记录 *Goldfish* 作为 */proc/cpuinfo* 中的硬件，但被获取的文件是 */init.goldfish.rc*。

---

需要重点提到的一个非常重要的事情是，*init* 在执行任何指令前会读取两个文件。因此这对于将开发板定制的修改加入到主 *init.rc* 文件中，而不是开发板定制的 *.rc* 文件中有一点影响。此外，虽然 *.rc* 文件通常来说能够使能它们自己的执行权限，但是 *init* 本身并不真正检查它。

### 语义

*init* 的 *.rc* 文件包含了一系列的声明，归纳为两种类型：行为和服务。每个声明部分都是以一个表示类型的关键字开头的，*on* 表示一个行为，*service* 表示一种服务。之后跟了很多关于声明的更多详细信息的行：

```
on <trigger>
  <command>
  <command>
  <command>
...
...
```

```
service <name> <pathname> [ <argument> ]*
    <option>
    <option>
    <option>
    ...

```

---

**警告：** *init* 中的“服务”与系统服务或应用程序开发者所使用的服务组件都没有关系。

---

**注意：**有趣的是，在AOSP的*init*的源代码中有一个*readme.txt*文件。你可以在文件夹*system/core/init*下找到它。它所描述的一些内容可能已经设计，但实际上并没有在当前的*init*中实现，比如*device-added*和*device-removed*触发器。总体而言，它仍然是一个很好的参考。

---

当然，配置文件可以声明很多的行为和服务。通常情况下，行为和服务都是左对齐，并且紧随其后的命令或选项会被像上面例子那样缩进。行为和服务的声明在某些情况下是类似的，都是以下一个on或sevice关键字出现之时来作为前一次声明的结束。然而，仅有一个行为，是以命令的执行作为结果的。服务声明仅用于描述服务，它们没有真正启动任何事情。这些服务通常都是当一个行为被触发后启动或停止的。

有两种类型的行为触发器：预定义触发器和由于属性值的变化来激活的触发器。*init*定义了一组固定的预定义触发器，它们以一个特定的顺序来运行。但是，属性激活的触发器，只有当给定的属性值达到一个*init.rc*文件中定义的特定值时，才会被激活。下面列出了一个可以在*init*配置文件中使用的预定义触发器的列表：

- `early-init`
- `init`
- `early-fs`
- `fs`
- `post-fs`
- `early-boot`
- `boot`

每个触发器以及它们所包含命令的意义将在接下来的部分，也就是我们查看主*init.rc*文件的时候，变得更清楚。目前，这里有一个预定义触发器和内置行为的执行顺序，它们由*init*程序在解析完其配置文件后执行：

1. 执行 `early-init` 命令。
2. `coldboot`: 检查 `ueventd` 已经在 `/dev` 目录下了。
3. 初始化属性服务的内部数据结构。
4. 为 `keychord` 启动处理程序。
5. 初始化控制台并显示启动文本或图像。
6. 启动初始化属性，比如说 `ro.serialno`、`ro.baseband` 和 `ro.carrier`。
7. 运行 `init` 命令。
8. 运行 `early-fs` 命令。
9. 运行 `fs` 命令。
10. 运行 `post-fs` 命令。
11. 开始属性服务。
12. 准备接收 `SIGCHLD` 信号。
13. 确保属性服务套接字和 `SIGCGLD` 处理程序已经准备好。
14. 运行 `early-boot` 命令。
15. 运行 `boot` 命令。
16. 运行所有的基于当前属性值触发的属性触发器命令。

基于属性的触发器。行为也可以基于属性值的变化而执行：

```
on property:<name>=<value>
```

从本质上来说，这个公式表示当 `name` 属性被设置为 `value` 值时，你可以运行一组命令。这方面的一个很好的例子是，默认的 `init.rc` 文件启动或停止 `adbd` 守护进程，是基于“USB 调试”复选框的选项的来回切换：

```
on property:persist.service.adb.enable=1
    start adbd

on property:persist.service.adb.enable=0
    stop adbd
```

行为命令。在使用 `on` 关键字声明了一个新的动作之后，最重要的是什么命令作为这个行为的一部分被实际执行。`init` 包括大量的命令，作为其词汇的一部分。虽然它们中的很多非常类似命令行的使用，但是你应该能够认识到它们的用途，有些是 Android 系统特有的。表 6-18 列出了 `init` 的命令。

表 6-18：版本 2.3/ 姜饼中的 init 命令

命令	描述
<code>chdir &lt;directory&gt;</code>	与 <code>cd</code> 命令一样
<code>chmod &lt;octal-mode&gt; &lt;path&gt;</code>	修改 <code>path</code> 的访问权限
<code>chown &lt;owner&gt; &lt;group&gt; &lt;path&gt;</code>	修改 <code>path</code> 的所有者
<code>chroot &lt;directory&gt;</code>	设置进程的根目录为 <code>directory</code>
<code>class_start &lt;serviceclass&gt;</code>	启动所有属于 <code>serviceclass</code> 的服务
<code>class_stop &lt;serviceclass&gt;</code>	停止所有属于 <code>serviceclass</code> 的服务，并禁用它们
<code>copy &lt;path&gt; &lt;destination&gt;</code>	复制一个文件到 <code>destination</code> 目录下
<code>domainname &lt;name&gt;</code>	设置系统的域名
<code>exec &lt;path&gt; [ &lt;argument&gt; ]*</code>	分叉并执行一个程序。建议使用一个 <code>init</code> 的服务代替，因为这个操作是阻塞的
<code>export &lt;name&gt; &lt;value&gt;</code>	设置环境变量 <code>name</code> 的值为 <code>value</code>
<code>ifup &lt;interface&gt;</code>	启动 <code>interface</code>
<code>import &lt;filename&gt;</code>	导入一个额外的初始化配置文件到一个当前的分析中
<code>insmod &lt;path&gt;</code>	插入一个内核模块
<code>hostname &lt;name&gt;</code>	设置系统的主机名
<code>loglevel &lt;level&gt;</code>	设置当前日志级别
<code>mkdir &lt;path&gt; [mode] [owner] [group]</code>	创建 <code>path</code> 目录，并设置适当的权限和所有者
<code>mount &lt;type&gt; &lt;device&gt; &lt;dir&gt; [ &lt;mountoption&gt; ]*</code>	挂载 <code>device</code> 到 <code>dir</code> 目录上
<code>restart &lt;service&gt;</code>	停止然后重启服务 <code>service</code>
<code>setkey &lt;table&gt; &lt;index&gt; &lt;value&gt;</code>	设置键盘输入值
<code>setprop &lt;name&gt; &lt;value&gt;</code>	设置属性 <code>name</code> 的值为 <code>value</code>
<code>setrlimit &lt;resource&gt; &lt;cur&gt; &lt;max&gt;</code>	设置 <code>resource</code> 的限定
<code>start &lt;service&gt;</code>	启动服务 <code>service</code>
<code>stop &lt;service&gt;</code>	停止服务 <code>service</code>
<code>symlink &lt;target&gt; &lt;path&gt;</code>	创建一个符号连接
<code>sysclktz &lt;mins_west_of_gmt&gt;</code>	设置时间域
<code>trigger &lt;event&gt;</code>	启动名为 <code>event</code> 的行为
<code>wait &lt;path&gt;</code>	等待直到文件系统中出现一个文件
<code>write &lt;path&gt; &lt;string&gt; [ &lt;string&gt; ]*</code>	打开一个文件，将 <code>string</code> 的内容写入文件

---

**警告：**尽管许多 *init* 的命令类似工具箱中或其他地方的命令行的等价物，但是重要的是，请注意只有在表 6-18 中列出的是被认可的。*init* 将不会试图发出命令到命令行。未识别的命令被直接忽略。

---

版本 4.2/ 果冻豆中，还有一些额外的命令被 *init* 所认可，你可以在表 6-19 看到。

表 6-19：版本 4.2/ 果冻豆中新出现的 *init* 命令

命令	描述
<code>class_reset &lt;serviceclass&gt;</code>	类似 <code>class_stop</code> ，但是无法停止服务
<code>load_persist_props</code>	装载永久性属性
<code>mount_all &lt;path&gt;</code>	基于 <i>path</i> 文件中发现的所有信息，挂载所有的分区
<code>restorecon &lt;path&gt;</code>	恢复 SELinux 内容
<code>rm &lt;path&gt;</code>	删除文件
<code>rmdir &lt;path&gt;</code>	删除目录
<code>setcon &lt;string&gt;</code>	设置安全信息（SELinux）
<code>setenforce &lt;value&gt;</code>	使能或禁用安全策略执行（SELinux）
<code>setsebool</code>	设置 SELinux 为 BOOL 类型

正如你所看到的，有一些命令是为了支持 SELinux。有关 SEAndroid，也就是对 SELinux 工作延伸的更多信息，你可以看看相关的项目网站。

**服务声明。**初始化仅指服务的名称，无法识别文件的路径名使得进程运行。因此，任何从一个文件中被执行的进程，都必须首先被分配给一个服务。正如我们前面所看到的，服务的声明是这样的：

```
service <name> <pathname> [ <argument> ]*
```

在这里需要着重强调的是，一旦这一行被解析，初始化将通过名称 `name` 来知道该服务。而由路径名 `pathname` 指向的该二进制文件的实际名字将不会被认可。其中一个最好的例子就是 `Zygote`（注意，下面这行的排版是为了适应本书页面的宽度）：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start  
-system-server
```

实际上这里正在运行的二进制文件是 `app_process`。然而，这不是由主 `init.rc` 文件的其余部分所引用的服务。相反，你会发现对 `zygote` 的引用如下所示：

```
onrestart restart zygote
```

**服务选项**。就像行为 (action) 一样，服务声明之后往往跟着一些行，这些行能够提供关于服务选项和如何运行服务的更多信息。表 6-20 详细介绍了这些选项。

表 6-20：初始化的服务选项

选项	描述
<code>class &lt;name&gt;</code>	这个服务属于 <code>name</code> 类，默认的类属于 <code>default</code>
<code>console</code>	服务需要并运行在控制台
<code>critical</code>	如果此服务崩溃五次，重启进入恢复模式
<code>disabled</code>	不自动启动该服务。它需要手动使用 <code>start</code> 命令来启动
<code>group &lt;groupname&gt; [ &lt;group name&gt; ]*</code>	在给定的组里运行这个服务
<code>ioprio &lt;rt be idle&gt; &lt;ioprio 0-7&gt;</code>	设置服务的 I/O 调度程序和优先级 <sup>a</sup>
<code>keycodes &lt;keycode&gt; [ &lt;key code&gt; ]*</code>	当指定的 key code 被激活时启动该服务
<code>oneshot</code>	服务只运行一次。在退出时将服务设置为禁用
<code>onrestart &lt;command&gt;</code>	如果服务被重新启动，运行 <code>command</code> 的命令
<code>seclabel &lt;string&gt;</code>	设置服务的 SELinux 标签，从版本 4.1/ 果冻豆开始使用
<code>setenv &lt;name&gt; &lt;value&gt;</code>	在启动这个服务之前设置 <code>name</code> 环境变量
<code>socket &lt;name&gt; &lt;type&gt; &lt;perm&gt; [ &lt;user&gt; [ &lt;group&gt; ] ]</code>	创建一个 UNIX 域套接字，并将它的文件描述符传递给该过程作为启动的开始
<code>user &lt;username&gt;</code>	为用户 <code>username</code> 运行这个服务

a. 看看关于 `ioprio_set()` 的手册页了解更多信息。

显而易见，这些服务选项中的某一些的用途比其他的更加明显。在一个特定的用户下运行或作为一个组的一部分运行的服务是简单的。但是基于一组特定的键组合运行的服务可能是不太明显的。下面是一个例子，展示了在版本 2.3/ 姜饼中，开发板定制的 `.rc` 文件是如何使用这个选项来设置 Nexus S（又名“Crespo”）的：

```
# bugreport is triggered by holding down volume down, volume up and power
service bugreport /system/bin/dumpstate -d -v -o /sdcard/bugreports/bugreport
    disabled
    oneshot
    keycodes 114 115 116
```

## 主 init.rc 文件

正如我们前面所讨论的那样，*init* 读取两个 *.rc* 文件，来指出它的配置。在 AOSP 中，在默认情况下，两个文件中的一个是为了所有的开发板而提供的。你会在附录 D 中发现这个文件的两个版本：一个用于版本 2.3/ 姜饼，另一个用于版本 4.2/ 果冻豆。我非常强烈地建议你仔细阅读该附录，因为 *init.rc* 文件是很多系统行为的基础。如果有时间的话，你可以看看注释（即以 # 开头的行）。默认情况下，这两个默认的文件实际上注释相当好，你应该可以使用之前的表来作为指南，使它们的内容让人更容易理解。

由 *init.rc* 文件管理下的一些操作是微妙的，但对 Android 的各个部分都有深远的影响。给跟你有关的版本文件设置书签是明智的，每隔一段时间，当你试图弄清楚一件事或其他有关系统的事情时，你都可以把它拿出来看一看。虽然默认的 *init.rc* 文件通常很容易看懂，但是要理解某些特定部分正在做什么，往往需要很扎实地掌握有关系统的其余部分，以及其中执行初始化操作的顺序。

---

**警告：**永远记住，默认的 *init.rc* 文件中的行为、命令和服务的特定顺序，对系统的运行是至关重要的。可以尝试制定自己的 *init.rc* 文件，但你很快会发现，如果默认的步骤不保存的话，很多系统中的事情将打破。比如说一些服务，除非有相应的选项是用来启动这些操作的，否则就干脆不运行。你最好多调整一下你的 AOSP 所提供的默认 *init.rc* 文件，或者更好的是，添加上你自己的开发板定制的 *.rc* 文件，如果你需要特定的主板的行为或服务被启动的话。

---

请注意，并非所有预定义的行为都必须在你的 AOSP 的默认的 *init.rc* 中使用。比如说，在版本 2.3/ 姜饼中，既不用 *early-fs* 也不用 *early-boot*。因此，如果你需要在 *fs* 或 *boot* 行为中优先允许命令的话，你可以将它们使用在你自己的开发板定制的 *.rc* 文件中。

## 开发板定制的 *.rc* 文件

如果你需要为 *init* 添加特定于开发板的配置说明，最好的办法是使用一个 *init.<device-name>.rc* 作为你的系统的量身定制。它特别的地方在于你怎么使用它。不过，我建议你先看看作为你 AOSP 中一部分的开发板定制的 *.rc* 文件。例如，下面是在版本 2.3/ 姜饼中的文件：

- *system/core/rootdir/etc/init.goldfish.rc*
- *device/htc/passion/init.mahimahi.rc*
- *device/samsung/crespo4g/init.herring.rc*
- *device/samsung/crespo/init.herring.rc*

以下是版本 4.2/ 果冻豆中的一些文件：

- *system/core/rootdir/etc/init.goldfish.rc*
- *build/target/board/vbox\_x86/init.vbox\_x86.rc*
- *device/asus/tilapia/init.tilapia.rc*
- *device/asus/grouper/init.grouper.rc*
- *device/samsung/tuna/init.tuna.rc*
- *device/ti/panda/init.omap4pandaboard.rc*
- *device/lge/mako/init.mako.rc*

正如你所期望的，这些文件通常包含硬件特有的命令。比如说，很多时候，它们都会包含对主板的具体安装说明。以下来自版本 2.3/ 姜饼的 Cresp 特定文件中的示例：

```
on fs
    mkdir /efs 0775 radio radio
    mount yaffs2 mtd@efs /efs nosuid nodev
        chmod 770 /efs/bluetooth
        chmod 770 /efs/imei
    mount ext4 /dev/block/platform/s3c-sdhci.0/by-name/system /system wait ro
    mount ext4 /dev/block/platform/s3c-sdhci.0/by-name/userdata /data wait noatime nosuid nodev
```

正如你所看到的，上面是将板上 eMMC 中的 ext4 分区挂载到 */system* 和 */data* 上。另一个例子在之前的章节中，它展示了当你按到设备上的一个特定的组合键时，*bugreport* 命令就会被激活。

同样，正如我之前提到的那样，*init* 在执行它里面的任何行为之前，会读取主要的 *init.rc* 文件和开发板定制的 *.rc* 文件。因此，通过在你的主板专用文件中声明一个 **boot** 行为或是一个 **fs** 行为，它里面的命令将被排队，当主配置文件中相同行为的命令找到时，该命令立即执行。这些命令仍然是在它们所在的行为中运行。因此，**boot** 行为中的命令会在 **fs** 行为的命令之后运行，而不管是哪个文件设置的命令被声明。

比如说，以下就是一个 *init.coyotepad.rc* 文件：

```
on property:acme.birdradar.enable=1
    start birdradar

service birdradar /system/vendor/bin/bradard -d /system/vendor/etc/rcalibrate.data
    user birdradar
    group birdradar
    disabled
```

这说明，当属性 `acme.birdradar.enable` 被设置为 1 时，`birdradar` 服务应该被启动。在之前有关工具箱的介绍中，我们在命令行中使用 `setprop` 命令设置该属性值为 1。假如上面的 `init.coyotepad.rc` 文件在启动时被作为系统的一部分，即之前的 `setprop` 命令将因此能够导致 `bradard` 被启动。

## init.<device\_name>.sh 文件怎么样？

在某些情况下，在由 `init` 的配置文件运行的命令之外，再运行一个 shell 脚本是很有意义的。比如说，仿真器依赖于文件夹 `/system/etc` 下的 `init.goldfish.sh` 文件。先不管文件的名称是什么，`init` 本身并不承认这样的脚本，并且也没有代码来查找它们。相反，`board-specific.rc` 文件可用于运行 shell 脚本，就像它们会运行其他的任何服务一样。这里描述了 `init.goldfish.rc` 文件是如何获取 `init.goldfish.sh` 来执行的：

```
service goldfish-setup /system/etc/init.goldfish.sh  
    Oneshot
```

在这种特殊情况下，shell 脚本中可以运行可用于 shell 上的命令，但并不是 `init` 词典中的一部分。如果你需要一个 shell 脚本的话，这实际上就是一个很好的理由。

## 全局属性

虽然我已经提到过全局属性很多次了，但还没有更深入地介绍过 Android 的这个方面。正如我更早前暗示的那样，全局属性是 Android 整体框架的重要组成部分。与 Windows 注册表类似，Android 的全局变量往往成为了全局共享的微不足道而又相对稳定的，在堆栈中所有部分中全局使用的变量。

### 工作原理

正如我前面提到的那样，`init` 将维持一个属性服务作为它的其他职责中的一部分。正如你可以在图 6-4 看到的那样，将该属性服务暴露于系统的其余部分有两种方法：

`/dev/socket/property_service`

这是一个 UNIX 域套接字，该进程可以打开与属性服务的对话，并能够设置和/或改变全局属性的值。

`/dev/_properties_`

这是一个“不可见”的文件（比如说，如果你查找它的话，你不会在 `/dev` 文件夹中看到它），它是在 `tmpfs` 挂载的文件夹 `/dev` 中创建的，也是内存映射到由

*init* 启动的所有服务的地址空间。通过这个映射区域，*init* 的子节点（例如，系统中的所有用户空间程序）可以读取到全局属性。

## /dev/\_properties\_ 的不可见性

你无法在文件系统中找到 */dev/\_properties\_*，这也是由于 *init* 处理文件方式的原因。以下是在初始化过程中它对文件实际所做的事：

1. 以只读模式创建 */dev/\_properties\_*。
2. 设置它的大小为一个期望的全局属性的工作空间大小。
3. 内存映射该文件到 *init* 的地址空间。
4. 关闭文件描述符。
5. 以只读模式打开文件。
6. 从文件系统中删除文件。

把删除文件作为最后一步，任何人在寻找 */dev* 文件夹时，实际上都不会看到文件。然而，由于该文件是内存映射的，而它仍是以读/写模式打开的，因此 *init* 的属性服务能够继续写入内存映射文件。此外，由于在被删除之前它是以只读模式打开的，*init* 还有一个文件描述符可以传递给它的子节点，这样它们就可以依次把内存映射到文件中，该文件对它们保持只读状态。

正如在侧边栏解释的那样，属性服务本质上保持了基于 RAM 的工作区，该工作空间存储了所有的全局属性。由于它的设置方式，只有属性服务可以写入此工作空间，但任何进程都可以读取它。因此，我们有一个单写入 / 多读取的配置。这样的设计允许属性服务申请对写请求的权限检查，该写请求通过 */dev/socket/property\_service* 的 UNIX 域套接字提交。具体的设置一个全局属性的权限是需要硬编码的。下面是版本 2.3/ 姜饼中的 *system/core/init/property\_service.c* 文件中截取的相关代码片段：

```
/* White list of permissions for setting property services. */
struct {
    const char *prefix;
    unsigned int uid;
    unsigned int gid;
} property_perms[] = {
    { "net.rmneto.",      AID_RADIO,    0 },
    { "net.gprs.",        AID_RADIO,    0 },
    { "net.ppp",          AID_RADIO,    0 },
    { "ril.",             AID_RADIO,    0 },
    { "gsm.",             AID_RADIO,    0 },
    { "persist.radio",     AID_RADIO,    0 },
```

```
{ "net.dns",           AID_RADIO,      0 },
{ "net.",              AID_SYSTEM,     0 },
{ "dev.",              AID_SYSTEM,     0 },
{ "runtime.",          AID_SYSTEM,     0 },
{ "hw.",               AID_SYSTEM,     0 },
{ "sys.",              AID_SYSTEM,     0 },
{ "service.",          AID_SYSTEM,     0 },
{ "wlan.",             AID_SYSTEM,     0 },
{ "dhcp.",             AID_SYSTEM,     0 },
{ "dhcpc.",            AID_DHCP,      0 },
{ "vpn.",              AID_SYSTEM,     0 },
{ "vpn.",              AID_VPN,       0 },
{ "debug.",             AID_SHELL,     0 },
{ "log.",              AID_SHELL,     0 },
{ "service.adb.root",  AID_SHELL,     0 },
{ "persist.sys.",       AID_SYSTEM,     0 },
{ "persist.service.",   AID_SYSTEM,     0 },
{ "persist.security.", AID_SYSTEM,     0 },
{ NULL, 0, 0 }
};
```

想要了解每个 AID\_\* UID 的意义，请参考本章前面所讲的“构建系统和文件系统”的关于 *android\_filesystem\_config.h* 文件的讨论，里面定义了用户 ID 和其他核心文件系统属性。例如，上面说的只有系统用户运行的进程可以改变以 *sys.* 或 *hw.* 开头的属性，而只有 *radio* 用户运行的进程（比如说，*RILD*）可以改变以 *ril.* 或 *gsm.* 开头的属性。

请注意，以 *root* 身份运行的进程可以改变它们希望改变的任何属性。还需要注意的是，名字以 *ro.* 开头的属性，在权限被上述阵列检查之前，这三个字符会被从名字中剥离出来。然而，这样的属性只可以被设置一次。试图改变一个名字以 *ro.* 开头的现有属性的值将会失败。此外，如果为一个给定的属性（或属性集）设置的权限，没有被上述阵列明确地授予给其下有一个进程正在运行的用户，那么这一进程将不允许设置该属性。下面是一个从一个非 *root* 的 shell 中试图设置 *acme.birdradar.enable* 的例子：

```
$ setprop acme.birdradar.enable 1
[ 1992.292414] init: sys_prop: permission denied uid:2000 name:acme.birdradar
.enable
```

正如我们在工具箱这一节中讨论的那样，你可以在命令行中使用 *getprop*、*setprop* 和 *watchprops* 命令来与属性服务进行交互。也可以通过你构建的作为 AOSP 一部分的代码来与属性服务进行交互。如果是以 Java 编码的，你可以看看 *framework/base/core/java/android/os/SystemProperties.java* 类。要使用这个类，就需要导入 *android.os.SystemProperties*。如果你是用 C 语言编码的，你可以看看 *system/core/include/cutils/properties.h*。要使用该头文件里的功能，你需要在代码中加入 *include <cutils/properties.h>*。

---

**注意：**全局属性不能通过由 SDK 提供的常规应用程序开发 API 来访问。

---

## 术语和集合

就像你可以从前面所有对全局属性的讨论中看到的那样，它们似乎都遵循一定的命名约定，即名字的每一部分都是由一个句点字符（.）来分隔开，而且紧跟着这个句点符号之后的每一部分名字，都进一步缩小了该属性所属的子类别。除此之外，也还有一些约定。当然，我们之前看到过的权限阵列，有一些决定 root 属性的基本集合。正如我们很快就会看到的那样，只有少数的属性被创建作为构建系统的一部分。当然也有一些值得记住的特殊的属性。尽管如此，每个设备都有自己特定的一组全局属性。还有就是，没有可以跨 Android 设备的权威的字典或官方的全局属性列表是能被预期到的。

没有什么可以阻止你为你的嵌入式系统创建自己的一组全局属性。到现在为止，我已经使用 acme.birdradar.enable 属性来举了一些例子。我可以整体的给你展现一下 acme.\* 属性，以及它的每一个部分在我的系统中使用的独立的目的。你也可以根据你的目标机的需要，来修改一些现有的全局属性。请确保你充分调查过你所修改的特定全局属性，是如何被 Android 系统的其余部分所使用的，因为其中的一些属性是由差别巨大的堆栈的不同部分来读取或设置的。一个良好的贯穿整个属性代码库的 grep 工具应该能够帮你迅速地找出它的用户们。

---

**注意：**在系统初始引导程序完成以后，可以使用 `getprop` 命令来获取你的设备的基本属性列表。此外，还可以从属性文件中看到，在启动时加载的属性的默认列表。我们将在下一节中看到这些。

---

正如我所说的，还有一些特殊的属性，根据其前缀的不同而有不同的处理方式：

**ro.\***

以该前缀开始的属性表示它是只读的。因此，它们只能在系统的生命周期中被设定一次。想要改变它们值的唯一方法是改变设置它们的信息源并重新启动系统。比如说，这样的属性有 `ro.hardware` 和 `ro.build.id`。

**persist.\***

标记有该前缀的属性一旦被设置将会被永久的存储。这样的属性有 `persist.service.adb.enable`，它用来启动 / 停止 `adb`。

**crl.\***

它有一个 `ctl.start` 和一个 `ctl.stop`，并且，设置它们实际上并不会产生任何

被保存到全局属性组中的属性。相反，当属性服务接收到一个对它们其中一个进行设置的请求时，它开始 / 停止其名称被提供作为该属性值的服务。比如说，Surface Flinger 服务会执行以下操作作为其启动的一部分：

```
property_set("ctl.start", "bootanim");
```

bootanim 服务的有效结果将会被 *init* 所启动。该 bootanim 服务及其选项，我们在前面在介绍 *init.rc* 文件时描述过了。工具箱的启动和停止也依靠于 *ctl.\** 来启动 / 停止服务。

*net.change*

每当一个 *net.\** 属性被修改时，*net.change* 会被设置为该属性的名称。因此，*net.change* 总是包含最近被修改的 *net.\** 属性的名称。

## 存储

全局属性不会存储在一个单一的位置，或它们也不会从一个单一的位置被设置。相反，不同的系统部分负责设置不同的全局属性组，以及几个系统部件都参与创建最终的全局属性组，这个全局属性组在任何的 Android 设备都存在。

构建系统。构建系统生成了两个属性文件：

*/system/build.prop*

它包含构建本身的信息，比如说 Android 的版本以及它被构建的日期。

*/default.prop*

它包含了必要的关键属性的默认值，比如说我们之前看到过的 *persist.service.adb.enable* 属性。

这两个文件都能够在用于初始引导程序的目标根文件系统中被找到，并作为系统的基本属性集。你可以在文件夹 *root/* 和 *system/* 的 *out/target/product/PRODUCT\_DEVICE/* 的子目录中找到它们。

该文件包含单行的名称 - 值对。它们在 *init* 启动时，被属性服务启动初期读取和分析。这些文件的大部分内容，是由文件夹 *build/core/* 下的核心 AOSP 构建代码所生成。尽管如此，在从版本 2.3/ 姜饼的 Crespo 的 makefile 文件中截取的代码片段中，它里面中的一部分是由设备指定的：

```
PRODUCT_PROPERTY_OVERRIDES += \
    wifi.interface=eth0 \
    wifi.suplicant_scan_interval=15 \
    dalvik.vm.heapsize=32m
```

附加的属性文件。除了通过构建系统生成的文件之外，你可以添加自己的特定于目标机的 */system/default.prop* 和特定设备的 */data/local.prop* 属性文件，这两者可以通过属性服务来读取，放在我们刚刚讨论的通过构建系统生成的文件旁边。

.rc 文件。正如我们之前看到的那样，*init.rc* 文件和 *init.<device\_name>.rc* 文件都可以设置全局属性。*init.rc* 文件实际上用于设置很多关键的全局属性。

代码。代码的一些部分也用来设置属性。比如说，连接服务会这样做：

```
SystemProperties.set("net.hostname", name);
```

进一步使事情混乱的是，如果属性值没有被找到的话，代码的某些部分还试图读取全局属性并使用其默认值。以下是从 *frameworks/base/core/jni/AndroidRuntime.cpp* 中截取的：

```
property_get("dalvik.vm.heapsize", heapsizeOptsBuf+4, "16m");
```

在这种情况下，调用者试图获取 *dalvik.vm.heapsize*，如果它没有被找到的话，值 *16m* 将被默认的使用。

*/data/property*。迄今为止我们所见过的存储方法都需要手动干预，要么在构建之前需要修改 AOSP，要么需要编辑设备上的文件。显然，系统需要能够在运行时自动更新值，并让它们下次重新启动时可以使用。这就是 */data/property* 目录的条目的作用。事实上，所有以 *persist.* 开头的属性都会在那个目录中被存储为单独的文件。里面的每一个文件包含有分配给该属性的值。因此，*/data/property/persist.service.adb.enable* 文件包含 *persist.service.adb.enable* 的值。

在 */data/property* 中看到的属性都由属性服务在启动阶段读取并存储。正如我们在前面讨论工具箱的 *setprop* 时提到那样，要毁掉一个永久存储的属性的唯一方法就是从目录 */data/property* 中删除该文件。

## ueventd

正如前面所讨论的那样，*init* 包括处理内核热插拔事件的功能。当 */init* 的二进制文件通过 */sbin/ueventd* 符号链接调用时，它会立即切换它的身份，从作为常规 *init* 运行切换为作为 *ueventd* 身份运行。图 6-5 显示了 *ueventd* 的操作。

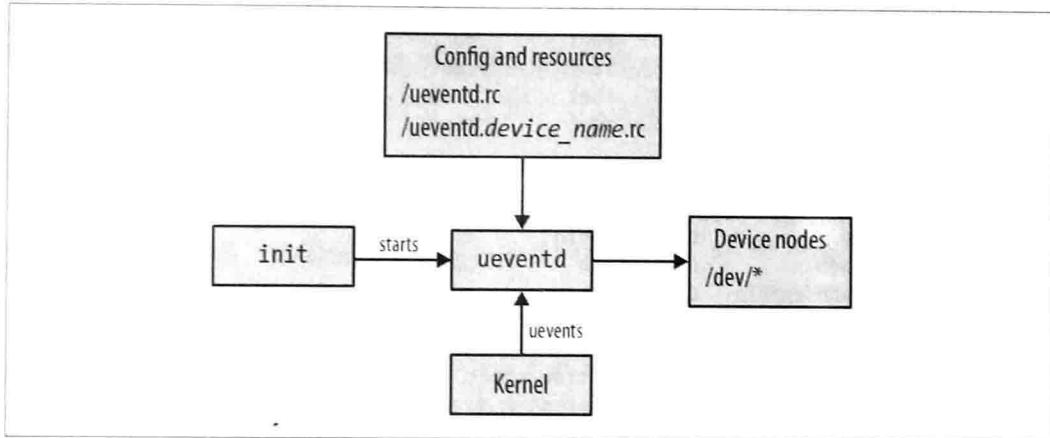


图 6-5: Android 的 ueventd

*ueventd* 是由默认的 *init.rc* 文件启动的最初的服务之一。开始时读取它的主要配置文件, */ueventd.rc* 和 */ueventd.<device\_name>.rc*<sup>注7</sup>, 重播所有内核 uevents (即热插拔事件), 然后等待, 监听所有的即将到来的 uevents。内核 uevents 通过一个网络连接套接字被传递到 *ueventd*, 该方法是特定内核功能与用户空间工具及其守护进程进行通信的常用方法。

基于 *ueventd* 接收事件和它的配置文件, 它会自动在目录 */dev* 下创建设备节点条目。而且, 由于后者是作为一个 tmpfs 文件系统被挂载的, 因此只活动在 RAM 中, 在每次重新启动时, 这些条目都从零开始, 基于 *ueventd* 的配置文件被重新创建。因此, *ueventd* 操作的关键是它的配置文件。

不同于 *init*, *ueventd* 的配置文件有一个相当简单的格式。本质上来说, 每一个设备条目的描述方式都是单行的如下所示的格式:

```
/dev/<node>      <mode>  <user>  <group>
```

当一个对应于 *node* (节点) 的 uevent 发生时, *ueventd* 会创建 */dev/node* 目录, 该目录带有对 *mode* (模式) 的访问权限组, 并分配条目给 *user/group*。权限和所有权是非常重要的, 因为关键的守护进程和服务必须能够访问相应的 */dev* 条目, 才能够操作正确。比如说系统服务器, 需要作为 *system* 用户运行。

比如说, 以下是从版本 2.3/ 姜饼中默认的 *ueventd.rc* 文件中截取的部分:

注 7: 这个文件的命名类似我们之前看到的 */init.<device\_name>.rc* 文件。

```
/dev/null          0666  root    root
/dev/zero          0666  root    root
/dev/full          0666  root    root
/dev/ptmx          0666  root    root
/dev/tty           0666  root    root
...
# these should not be world writable
/dev/diag          0660  radio   radio
/dev/diag_arm9      0660  radio   radio
/dev/android_adb     0660  adb    adb
/dev/android_adb_enable 0660  adb    adb
/dev/ttyMSM0         0600  bluetooth bluetooth
/dev/ttyHS0          0600  bluetooth bluetooth
/dev/uinput          0660  system  bluetooth
/dev/alarm           0664  system  radio
/dev/tty0            0660  root    system
/dev/graphics/*      0660  root    graphics
/dev/msm_hw3dm       0660  system  graphics
/dev/input/*         0660  root    input
/dev/eac             0660  root    audio
...
...
```

正如 *init* 一样，你需要将你的开发板定制的节点条目放到 *ueventd.<device\_name>.rc* 文件中。比如说，以下就是 *ueventd.coyotepad.rc* 中的一个设备条目：

```
/dev/bradar        0660  system  birdradar
```

请注意，某些 uevents 可能需要 *ueventd* 工具来加载代表内核的固件文件。没有任何配置选项可以用于 *ueventd* 的配置文件。相反，要确保这些固件文件是在 */etc/firmware* 中或是在 */vendor/firmware* 中。比如说，在 CoyotePad 的情况下，我们用 *PRODUCT\_COPY\_FILES* 把 *rfirmware.bin* 放入 */system/vendor/firmware* 文件夹中。

## 启动标志

先不管什么设备的引导程序可能会在启动时显示，Android 设备的屏幕在启动过程中通常经历了四个阶段：

### 内核启动屏幕

通常情况下，Android 设备在启动时将不在它的 LED 屏幕上显示内核引导信息。取而代之的是，内核可能要么保持黑屏直到 *init* 启动，要么它可能在帧缓冲区中显示一个静态的标志，构建作为内核镜像的一部分。所有这样的显示都超出了本书的范围。

### 初始化引导标志

当初始话控制台时，*init* 很早地会有一个文本字符串或图像显示。本节的目的就是要讨论 *init* 显示在这里的是什么。

## 开机动画

这是一系列动画图像，可能是循环显示的，它们是在 Surface Flinger 服务启动期间显示的。我们将在之后讨论 java 用户空间时再讨论开机动画。

## 主屏幕

这是启动器的开始屏幕，它是在引导序列执行结束的时候被激活的。如果想定制显示内容的话，你需要深入挖掘启动器的来源。

如果你回头看一下之前的“配置文件”中，关于 *init* 在预定义的行为上的强制执行顺序和内置命令的解释，你会发现，第五步是初始化控制台及显示启动文本或图像。在此步骤中，*init* 将尝试从 */initlogo.rle* 文件加载一个标志图像，并将它显示在屏幕上。如果没有找到这样的文件，它会显示由仿真器在启动时显示的文本相似的文本字符，如图 6-6 所示。

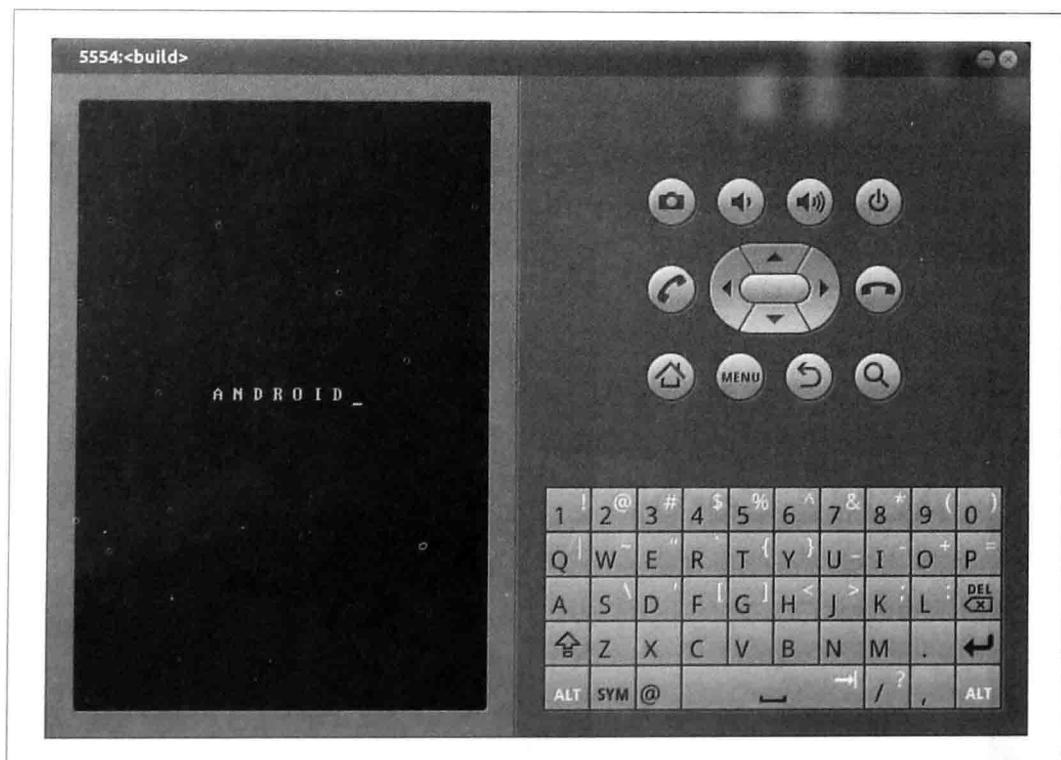


图 6-6: *init* 的启动标志

如果想改变这个字符串，你可以看看 *system/core/init/init.c* 文件中的 `console_init_action()`。如果你想有一个图形标志来代替文字的显示，需要创建一个适当的 *initlogo.rle*。让我们来看看是如何做的。

首先，你需要找出你的设备的屏幕大小。比如说，仿真器在构建后从 AOSP 的命令行启动时的默认分辨率为  $320 \times 480$  像素。假设你有一个这样大小的 PNG，首先需要将其转换为能被 *init* 所认可的图片格式。主机上有两个工具需要这样做：*convert*，它是 ImageMagick 软件包的一部分，以及 *rgb2565*，它是作为 AOSP 的一部分构建的<sup>注8</sup>：

```
$ cd device/acme/coyotepad  
$ convert -depth 8 acmelogos.png rgb:acmelogos.raw  
$ rgb2565 -rle < acmelogos.raw > acmelogos.rle  
153600 pixels
```

这将获取 *acmelogos.png*，并将其转换成一个 *acmelogos.rle*，然后你就可以通过修改 CoyotePad 的 *full\_coyote.mk* 来复制它并添加以下片段：

```
PRODUCT_COPY_FILES += \  
    device/acme/coyotepad/acmelogos.rle:root/initlogo.rle
```

当你重建了 AOSP 之后，更新你的设备，并重新启动它，你会看到如图 6-7 所显示的那样，而不再是以前的文本字符串。

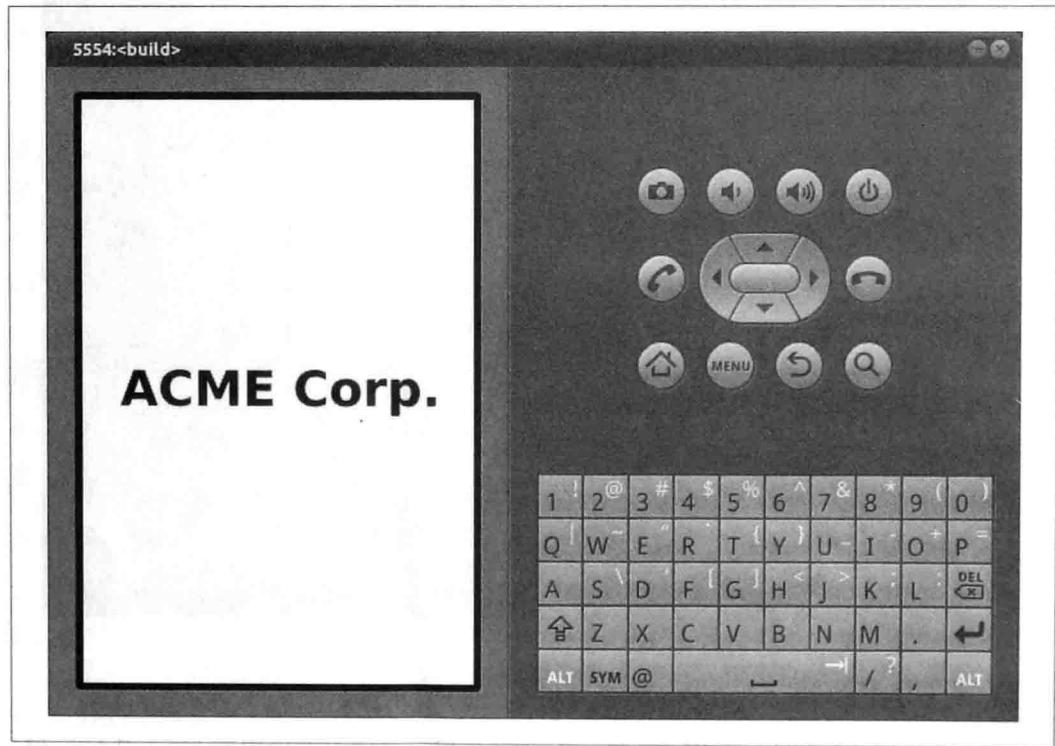


图 6-7：coyotePad 的启动标志

注 8：请记住，在路径被正确设置，来使用主机工具构建作为 AOSP 的一部分之前，你需要运行 *build/envsetup.sh* 和 *lunch*。

一般情况下，LCD 屏幕会保持不变，直到 Surface Flinger 服务启动，并在剩下的系统服务被启动时启动开机动画。

# Android 框架

你的最终目标是让你的嵌入式系统运行 Android 环境，而不是简单只是运行我们前面所说的原生用户程序，因为 Android 是一个用户和开发人员都已经习惯了的环境。这不仅包括全套系统服务和用于应用程序开发的标准 API 封装，还包括一些不太明显的组件，例如支持系统服务和硬件抽象层原生守护进程。本章将介绍在本机用户空间程序基础之上，Android 框架是如何运行的，并且将讨论如何与之进行交互以及如何对其进行定制。

请注意，跟前面介绍的 Android 组件不一样，那些组件可以有很多种方式进行修改，而大多数 Android 框架都必须那样用。例如，你不能选择只运行某些系统服务，因为它们不是通过脚本或者基于某个配置文件启动的。相反，修改 Android 框架通常需要深入到源代码当中修改 / 增加你自己的代码来对它的行为进行定制化。

因此，这种定制工作需要非常熟悉 Android 源码和版本依赖。尽管如此，我们这里还是会尽量覆盖要领，这足以让你开始靠你自己导航 Android 的内部。然而，这只是一个长期努力的开始，因为 Android 源码相当多，并且新版本正在以非常快的速度更新。

### 到底什么是“Android 框架”？

如果你回过头去看看图 2-1，Android 框架包括 `Android.*` 的包、系统服务、Android 运行时函数库以及本地守护程序。从源代码角度来说，Android 框架通常是指 AOSP 中 `frameworks/` 目录下的所有源代码。

在某种程度上，我使用“Android 框架”一词指的是运行于本地用户程序之上的

所有东西。所以，我这里的解释很多时候会超出 *frameworks* 的范围。当然，我将会讨论 Android 框架（例如 Dalvik 和 HAL 等）与生俱来内容。

## Framework 入门

第 6 章中，我们以 *init* 命令及其配置和用法结束。我在讲述 *init.rc* 的时候，只是简单介绍了一下通过 Zygote 启动 Android 框架的方式。对于这个话题，这里就需要更多地来讲一讲。在上一章里面我介绍的大部分东西可以很容易在嵌入式 Linux 世界中找到类似的东西，但是后面所介绍的就很难找到相应替代了。事实上，Android 开发人员对于移动计算世界的主要贡献在于他们在 BSD/ASL 授权的嵌入式 Linux 上构建的这套东西。

### 编译不包含 Framework 的 AOSP

也许看起来很奇怪，但是确实存在一些情况，你希望编译出来一个不包含基于 Java 的系统服务和应用的 AOSP，跟大家所知的 Android 完全不一样。不管你是希望在一个无显示系统（headless）上运行“Android”，还是仅仅是因为你在调试一个板子时希望有一个最简单的 AOSP 用于基本的工具和最小本地用户程序环境，AOSP 确实提供了这样一个编译策略：Tiny Android。

要使 AOSP 生成 Tiny Android，你只需要到 AOSP 的源代码目录输入以下指令：

```
$ BUILD_TINY_ANDROID=true; make -j16
```

这样会产生几个输出镜像文件，包含内核镜像和 Android 的最小组件集合，可以支持本地 Android 用户空间程序运行。大致上，你可以得到一个 Toolbox、Bionic、*init*、*adb*、*logcat*、*sh* 以及一些其他的关键二进制程序及库文件。没有任何的 Android 框架会包含在镜像中，包括任何的服务和应用程序。

也许，虽然这究竟是不是“Android”还是个问题，但是很多时候它很有用。

“Android”指代的是什么最终取决于你，毕竟仁者见仁、智者见智。

## 核心构建模块

Android 的框架依赖于一系列的核心构建模块：Service Manager、Android 运行时库、Zygote 和 Dalvik。缺少了这些模块，我们所知的 Android 没有任何组件可以正常工作。我们在第 2 章中在描述系统启动时已经对这些模块及其角色做了介绍。想要做一个更

深入的探讨，建议你回过头再阅读一下那一章的相关内容，特别是现在我们已经对 *init* 及其脚本也做过了阐述。实际上，在阅读后续内容时，你还可能需要翻一下本书附录 D 中关于 *init.rc* 的主要内容。

*servicemanager* 是 *init* 首先启动的服务中的一个。我前面也提到过，它是所有运行的系统服务的“黄页”或是目录。显然，在它启动时还没有任何系统服务被启动，而它必须尽早就绪以使得后续启动的服务可以向它注册，以使得系统其他部分可以找到它。

如果 *servicemanager* 没有运行，任何系统服务都没有办法提供服务，框架也就没有办法工作。所以，*servicemanager* 不是一个可选的组件，其在 *init.rc* 中的位置也不可以修改。你必须保留其在 *init.rc* 文件中本来的样子，不能有修改。

下一个启动的核心组件是 *Zygote*，以下是 *init.rc* 中启动它的代码：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
```

这一语句其实包含了大量的动作。首先需要注意的是，这里实际启动的命令是 *app\_process* 命令，其形式化的参数列表如下：

```
Usage: app_process [java-options] cmd-dir start-class-name [options]
```

*app\_process* 是一个很少有人知道的命令，但却非常重要，它使得你启动一个新的 Dalvik 虚拟机来运行命令行中指定的 Android 代码。不过这并不意味着你可以用它来在命令行启动普通的 Android 应用程序，后面你将学习到有另外一个命令可以达到这个目的：*am* 命令。一些核心的系统组件和工具必须在命令行里面启动且不能使用现有的 Dalvik 虚拟机实例，就比如说 *Zygote*，它是运行的第一个 Dalvik 进程。*am* 和 *pm* 也是这样，后面会进一步提到。

*app\_process* 依赖于 Android 运行时库来实现其功能。这个库是共享库文件形式，*libandroid\_runtime.so* 提供启动和管理 Dalvik 以运行 Android 字节代码的能力。另外，它还预装载 Android API 所需要的一系列库文件。例如，它会预装 Android 框架的字节代码依赖的所有本地调用程序，它们注册到 VM 以后，任何用 Java 编写的 Android 框架程序包可以通过它调用本地函数功能。

运行时库还包括一些辅助性的函数，这些函数通常对所有运行在 Dalvik 虚拟机上的 Android 应用都会用得到。实际上，你可以把 Dalvik 看作是一个非常粗糙、原始的低级 VM，它并不知道你在它上面运行的是 Android 代码。为了在 Dalvik 虚拟机上运行 Android 代码，运行时库要使用一些特定的参数来启动 Dalvik，这些参数根据 Java 代码依赖的 Android Java 应用程序接口进行了特殊的裁剪，这些应用程序接口可能是在

SDK 的开发者文档中公开发布的，也可能是作为 AOSP 中编译内部 Android 代码所用的内部 API。

运行时库又进一步依赖于很多本地程序空间的功能。例如，它使用了一些 *init* 维护的全局属性来控制 Dalvik 虚拟机的启动，使用 Android 的日志功能来记录 Dalvik 虚拟机启动过程的信息。除了初始化用于启动 Dalvik 虚拟机的参数以外，运行时库还在代码 `main()` 方法被调用前初始化 Java 和 Android 环境。最重要的是，它为刚刚初始化好的虚拟机中的所有线程提供了默认的异常处理句柄。

不过需要注意的是，运行时库并不会预装载 Java 类：这是 Zygote 在启动系统来运行 Android 应用时才做的工作。而且由于每次使用 `app_process` 就会导致产生一个新的独立 VM，所有非 Zygote 的 Dalvik 实例都会在使用时才装载 java 类，而不会在代码运行前装载。

## Dalvik 的全局属性

除了上一章我们讨论的 *init* 维护的全局属性以外，Dalvik 还进一步提供一些 Java 程序通过 `java.lang.System` 可以找到的全局系统属性。例如，如果浏览一些系统服务的源代码，你可能就会注意到对 `System.getProperty()` 函数或者 `System.setProperty()` 函数的调用。注意，这些函数调用以及其依赖的属性集合跟 *init* 维护的全局属性是相互独立的。

举个例子来说，包管理器服务会在启动时读取 `java.boot.class.path` 属性。而且，如果你在命令行使用 `getprop`，在 *init* 返回的属性列表中你压根就找不到这个属性。相反，这些变量是在每一个 Dalvik 实例中维护的，只能让运行在上面的 Java 代码读取和 / 或使用。例如，`java.boot.class.path` 属性就是在 `dalvik/vm/Property.c` 中使用 `init.rc` 脚本的 `BOOT CLASSPATH` 变量来设置的。

你还可以在 Java 官方文档中看到更多的 Java 系统属性。要注意的是，*init* 中使用的全局属性的变量名称跟 Java 系统属性中使用的变量名在语意上是非常接近的。

通过 `app_process` 启动的 Java 类开始执行以后，它就可以使用“普通”的 Android API 来跟现有的系统服务进行通信了。如果它被当作 AOSP 一部分来编译构建的，那么它在编译时很多 `android.*` 包都可以使用。`am` 和 `pm` 程序就是这样做的。你也可以这样完全用 Java，采用 Android API 编写你自己的命令行工具，与框架的其他部分独立启动。也就是说，它可以完全独立于 Zygote 启动和运行。

但是这并不是就能允许你使用 `app_process` 启动你编写的普通 Android 应用。Android 应用只可以通过 Activity 管理器采用 intent 来启动，Activity 管理器本身是 Zygote 本身启动后作为一个系统服务来启动的。这就又回到了前面讨论过的 Zygote 启动过程。

为了保证 Zygote 正常启动并启动剩下的系统服务，你必须保持 `init.rc` 中相应的 `app_process` 命令行不做任何变化，在其默认的位置。对于 Zygote 启动，没有什么东西你可以额外配置的。然而，你可以通过修改一些系统全局属性来影响 Android 运行时启动其 Dalvik 虚拟机的方式。可以看一看 2.3/ 姜饼和 4.2/ 果冻豆中位于 `frameworks/base/core/jni/AndroidRuntime.cpp` 文件的 `AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)` 函数，可以看出它在准备启动一个新的虚拟机时读取了哪些系统属性。要注意的是，任何使用这些属性来影响 Dalvik 虚拟机的方式都很可能是版本相关的。

一旦 Zygote 虚拟机启动完成，`com.android.internal.os.ZygoteInit` 类的 `main()` 函数就会被调用，它会预装载所有的 Android 包，然后启动系统服务器 System Server，接着就开始侦听 Activity 管理器的请求去 fork 和开启新的 Android 应用。同样的，对系统服务器 System Server 启动也没有什么可以配置的东西，你可以自己看看 `frameworks/base/core/java/com/android/internal/os/ZygoteInit.java` 文件中 `startSystemServer()` 函数的参数。我建议你不要修改这个，除非你对 Android 内部机制已经很精通了。

## 禁止 Zygote

虽然你不能够对 Zygote 启动进行配置，但是你可以在 `init.rc` 脚本中通过添加 `disabled` 选项完全禁止掉它的启动。这里是在 2.3/ 姜饼中的做法：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote  
--start-system-server  
    socket zygote stream 666  
    onrestart write /sys/android_power/request_state wake  
    onrestart write /sys/power/state on  
    onrestart restart media  
    onrestart restart netd  
    disabled
```

这会阻止 `init` 在引导时启动 Zygote，所以任何 Android 框架组件都不会启动，包括 System Server。这在你调试关键系统错误或者开发某个 HAL 模块过程中很有用，你必须手工配置调试工具、装载文件，以及在关键系统服务启动前监测系统行为。

最后，你还可以手工启动 Zygote 以及其他系统服务：

```
#start zygote
```

## 系统服务

就像在上一节我们看到的，System Server 是作为 Zygote 启动的一个部分。这一节，我们将进一步深入这个过程。不过在第 2 章中我们讨论过，还有系统服务是在 System Server 之外的进程启动的，这一节会进一步讨论。

从 4.0/ 冰淇淋三明治开始，最先启动的系统服务是 Surface Flinger。到 2.3/ 姜饼为止，它还只是作为系统服务器 System Server 的一部分来启动的。但是在 4.0/ 冰淇淋三明治中，它在 Zygote 之前启动，并且独立于系统服务器和其他系统服务运行。这里是 4.2/ 果冻豆系统中 *init.rc* 文件里启动 Zygote 前的代码片段：

```
service surfaceflinger /system/bin/surfaceflinger
    class main
    user system
    group graphics drmrpc
    onrestart restart zygote
```

Surface Flinger 的源码在 2.3/ 姜饼系统中位于 *frameworks/base/services/surfaceflinger/* 中，在 4.2/ 果冻豆系统中位于 *frameworks/native/services/surfaceflinger/* 中。它的角色是将 app 绘制的界面组合成最终的图像呈现给最终用户。所以，它是 Android 系统中最重要的基础构件块。

在 Android 4.0 中，由于 Surface Flinger 比 Zygote 先启动，所以系统的启动动画比之前版本出来的更早。我们会在这一章后续部分进一步讨论启动动画。

为了启动系统服务器 System Server，Zygote 分支了一个新进程运行 `com.android.server.SystemServer` 类的 `main()` 函数。然后装载 `libandroid_servers.so` 函数库，它包含了一些系统服务需要的 JNI 部分，然后调用 *frameworks/base/cmds/system\_server/library/system\_init.cpp* 中的本地代码，这就在 `system_server` 中调用了以 C 语言编写的系统服务。在 2.3/ 姜饼中，这包括 Surface Flinger 和传感器服务 Sensor Service。但是在 4.2/ 果冻豆中，正如我们看到的，由于 Surface Flinger 单独启动了，通过 `system_server` 启动的唯一一个 C 语言编写的系统服务就只剩下 Sensor Service 了。

然后系统服务器返回到 Java 并且继续初始化关键的系统服务，例如电源管理器、Activity 管理器和包管理器。然后它继续启动由它管理的其他系统服务并将它们注册到服务管理器中。这些动作都在 *frameworks/base/services/java/com/android/server/SystemServer.java* 中完成。这些都硬编码在 `SystemServer.java`，没有任何标志位或者参数可以传递进去告诉系统服务器 System Server 不启动某些系统服务。如果你想禁止某一个，你必须手工去注释掉相关的代码行。

---

**警告：**系统服务是相互依赖的，而且几乎所有的 Android 部分，包括 Android API，都假设所有构建进 AOSP 的系统服务任何时候都是可用的。正如我在第 2 章中所提到的，作为一个整体，这些系统服务构建在 Linux 系统上形成了一个面向对象的操作系统，这个系统的很多部分并没有把模块化作为其设计追求。所以，如果你移除某个系统服务，很明显某些 Android 部件就会挂掉。

这并不意味着那就没办法了。我在 2012 年 Android 构建者峰会上所做的名为“Headless Android”的报告 (<http://www.opersys.com/blog/headless-android-1>) 上介绍了一种方法，我成功地禁用了 Surface Flinger、Window Manager 以及一些其他关键系统服务，使得完整的 Android 运行在无显示系统上。我在报告里也警告了，这项工作还只是属于概念论证阶段，还需要大量的工作才能使得它能投入到产品中去<sup>注1</sup>。

所以，你可以任意修改，但是如果你打算深入到 Android 的内部去，你最好有所准备。

---

## 什么是 /system/bin/system\_server ?

当浏览目标机根文件系统时你可能注意到，在 /system/bin 目录下有一个二进制程序 `system_server`。然而这个二进制程序对于 System Server 以及其他系统服务启动过程没有任何作用。不知道这个二进制程序有什么作用，也许它是 Android 早期时候留下来的遗留应用。

这个字符串经常是迷惑的根源，因为简单看一下二进制程序列表或者 ps 程序的输出可能导致你相信 `system_server` 进程是由命令 `system_server` 启动的。我在阅读相关源码的时候，实际上我自己也相当怀疑，并且在 Android-building 邮件列表中询问了这个问题，不过随之而来的回答 (<https://groups.google.com/forum/?fromgroups#!topic/android-platform/x2ToX7x5Yzw>) 好像也印证了我对代码的理解。

System Server 除了启动 Surface Flinger 和一些系统服务以外，还有一些系统服务由 `mediaserver` 启动。这是 2.3/ 姜饼的 `init.rc` 文件的相关片段（4.2/ 果冻豆中实际上也差不多）：

```
service media /system/bin/mediaserver
    user media
    group system audio camera graphics inet net_bt net_bt_admin net_raw
        ioprio rt 4
```

---

注1：有趣的是，自从 4.1/ 果冻豆之后，在官方 AOSP 发行版中增加了一个新的全局属性 `ro.config.headless`。这个属性好像会允许 Android 程序在没有用户界面情况下运行。

在 2.3/ 姜饼中，Media 源码位于 `frameworks/base/media`，而在 4.2/ 果冻豆中，`mediaserver` 的源码位于 `frameworks/av/media`。`mediaserver` 启动的系统服务有 Audio Flinger、Media Player Service、Camera Service 以及 Audio Policy Service。同样，这是不能配置的，而且推荐你不要修改 `init.rc` 文件中这部分内容，除非你知道你的修改可能会引起的潜在问题。例如，如果你想要将 `mediaserver` 从 `init.rc` 中移除，或者使用 `disable` 选项来阻止它启动，你会在 `logcat` 日志中看到这样的消息：

```
...
I/ServiceManager( 56): Waiting for service media.audio_policy...
I/ServiceManager( 56): Waiting for service media.audio_policy...
I/ServiceManager( 56): Waiting for service media.audio_policy...
W/AudioSystem( 56): AudioPolicyService not published, waiting...
I/ServiceManager( 56): Waiting for service media.audio_policy...
I/ServiceManager( 56): Waiting for service media.audio_policy...
...
...
```

系统会挂起并且不停地输出上述信息，直到 `mediaserver` 启动起来。

请注意，`mediaserver` 是 `init` 服务中唯一一个使用 `ioprio` 选项的。很不幸，没有官方文档正式说明原因，我估计大概是希望确保媒体播放拥有一个合适的优先级，以避免媒体播放断断续续。

最后一个服务是手机领域的传统角色——电话应用，它提供了电话服务。总体来说，`app` 是不应该放到系统服务中的，因为 `app` 的生命周期是被管理的，所以可以被按照意愿停止或者重启。而系统服务则是希望从开机到重启期间一直运行，所以就不能在不影响系统其他部分的情况下中途停掉它。电话应用是个特例，它的 `manifest` 文件中应用 XML 元素中将 `android:persistent` 属性设置为 `true`。这就向系统表明了这个应用的生命周期不应该受到管理，因此就使得它可以容纳系统服务。这样同样使得它在 Activity Manager 初始化时被自动启动起来。

同样，电话应用通常也不是可配置的。然而，你可以相对简单地从 AOSP 构建中移除电话应用。当然结果就是任何依赖于这个系统服务的部分没办法正常工作。当然，你可能也希望让他保留，但是想将拨号图标从 HOME 屏幕删掉，同时还想删掉联系人应用。可能听起来毫无道理，Android 用户实际操作的电话拨号程序并不是电话应用的一部分，它其实是联系人应用的界面。

---

注意：另一个由应用程序容纳系统服务的例子是 NFC 应用，在 `packages/app/Nfc` 目录下可以找到它。

---

电话应用这种提供系统服务的方式很有趣，因为它为我们开启了一扇门，展示了以 app 方式增加系统服务的方式。这种方式允许我们在自己的 *device/acme/coyotepad/* 目录下增加系统服务，而不需修改 *framework/base/services* 下默认系统服务的代码。

## 开机动画

正如我在前面的章节讨论开机标志时解释所说的，Android 的 LCD 在启动过程中经历了四个阶段。其中之一就是开机动画。以下是版本 2.3/ 姜饼中的 *init.rc* 相应的条目（在版本 4.2/ 果冻豆中实际上是相同的）：

```
service bootanim /system/bin/bootanimation
    user graphics
    group graphics
    disabled
    Oneshot
```

请注意，该服务被标记为禁用。因此，*init* 实际上将不会马上启动该服务。相反，它必须在其他地方被明确启动。在这种情况下，它作为 Surface Flinger 服务，实际上是在通过设置 *ctl.start* 全局属性来完成它本身的初始后之后，才启动开机动画的。下面是来自 *SurfaceFlinger::readyToRun()* 函数的代码，该函数在版本 2.3/ 姜饼中的 *frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp* 文件中：

```
// start boot animation
property_set("ctl.start", "bootanim");
```

版本 4.2/ 果冻豆的 *frameworks/native/services/surfaceflinger/SurfaceFlinger.cpp* 文件中的代码也发挥同样的作用：

```
...
void SurfaceFlinger::startBootAnim() {
    // start boot animation
    property_set("service.bootanim.exit", "0");
    property_set("ctl.start", "bootanim");
}

...
status_t SurfaceFlinger::readyToRun()
{
    ...
    // start boot animation
    startBootAnim();

    return NO_ERROR;
}
...
```

考虑到 Surface Flinger 服务是第一批启动的系统服务之一（如果不是第一个的话），当系统的关键部分正在初始化时，开机动画停止连续显示。通常情况下，它只在手机

的主屏幕终于显示在前台时停止。我们来简要的看一看在开机动画期间发生了什么事情。

正如你在前面的 *init.rc* 文件片段中看到的，*bootanim* 服务对应于 *bootanimation* 二进制文件。后者的源代码在 *frameworks/base/cmds/bootanimation/* 文件夹中，如果深入挖掘一下的话，你会发现这个工具是直接通过连接器来与 Surface Flinger 服务进行交互，以呈现它的动画，因此 Surface Flinger 服务需要在动画启动前就存在。图 7-1 展示了通过 *bootanimation* 显示的默认的 Android 开机动画，并在 Android 的标志上带有移动的光反射投影。

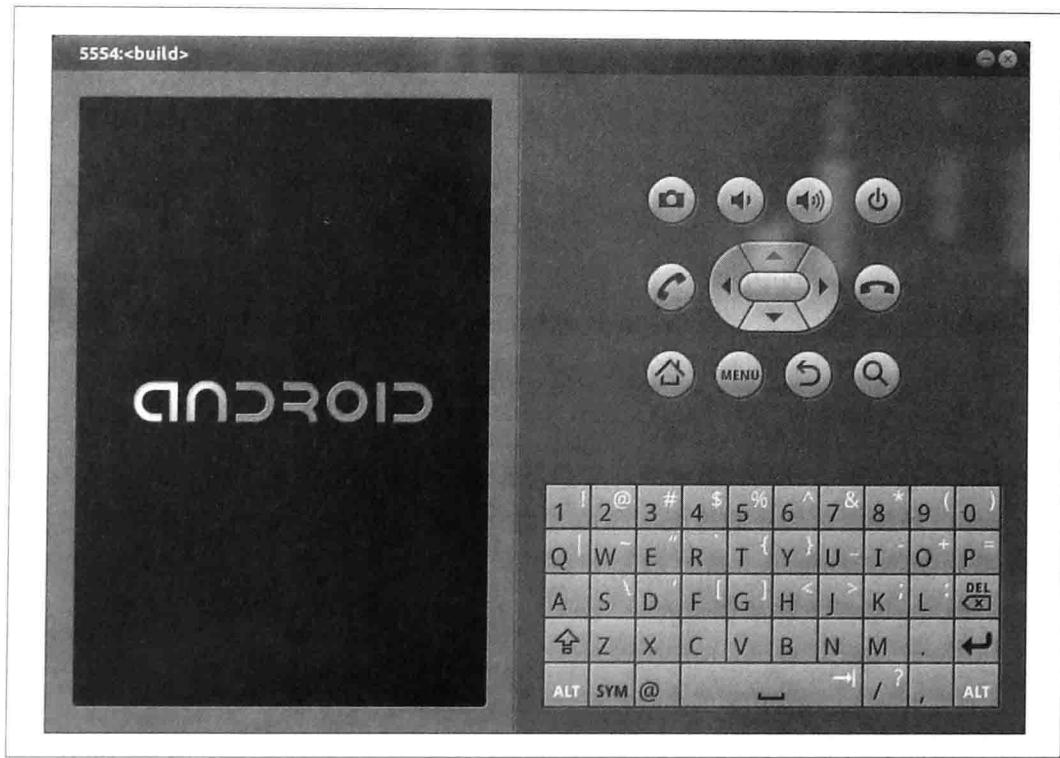


图 7-1：默认的开机动画

*bootanimation* 实际上有两种操作模式。在一种模式下，它会用 *frameworks/base/core/res/assets/images/* 文件夹下的图像创建默认的 Android 标志的开机动画。你最好不要尝试通过修改这些文件来更改开机动画。相反，通过提供或者是 */data/local/bootanimation.zip*，或者是 */system/media/bootanimation.zip*，你就可以迫使 *bootanimation* 进入它的另一种操作模式，该操作模式使用这些 ZIP 文件中的一个的内容，来呈现一个开机动画。这是值得花一些时间来研究的，并看看它是如何做到的，虽然本书不是一个完美的媒介来图解一个运行中的动画。

## bootanimation.zip

*bootanimation.zip* 是一个普通的，未压缩的 ZIP 文件，它包含有至少一个在顶级目录里的 *desc.txt* 文件和一堆包含 PNG 文件的目录。后者里是根据 *desc.txt* 文件中的规则排列的动画序列。需要注意的是，*bootanimation* 除了 PNG 文件之外不再支持其他格式。下面是 *desc.txt* 文件的语义：

```
<width> <height> <fps>
p <count> <pause> <path>
p <count> <pause> <path>
```

请注意，该文件的格式是非常简单的，并且不容许任何的错误。所以必须严格遵守上面的语义规定。第一行表示动画的宽度、高度和帧速率（每秒帧的数量）。每个随后的行就是动画的一部分。对于每一个组成部分，你必须提供这部分被播放（计数）的次数，帧的每个部分被播放之后暂停的帧数（暂停），并且还有动画的每一部分所在的目录（路径）。每个部分按照它们出现在 *desc.txt* 的顺序进行播放。

每个动画的部分，以及相关的目录，都是由几个 PNG 文件组成，这些文件的文件名表示这个帧在所有序列中的帧序号。比如说，文件可以被命名为 *001.png*、*002.png*、*003.png* 等。如果计数值被设为零，该部分将被循环播放，直到系统完成启动且启动栏启动。通常情况下，初始部分可能计数为 1，而最后的部分通常计数为 0，所以它是连续播放的，直至启动完成。

想要创建自己的开机动画，最好的办法是看看由其他人创建的现有的 *bootanimation.zip* 文件。如果在你最爱的搜索引擎中搜索一下该文件名，你应该可以找到几个相对容易的例子。比如说，看一看几个最新的开机动画，它们是 Android 发布的为 CyanogenMod 售后支持所创建的。

---

**警告：**再重申一遍，请确保你创建的 Zip 文件是未压缩的。否则它将不会工作。你可以看看 zip 命令的主页，特别是 -o 标志。

---

## 禁用开机动画

如果你不想要的话，也可以直接禁用开机动画。你只需使用 *init.rc* 中的 *setprop* 命令，将 *debug.sf.nobootanimation* 设置为 1：

```
setprop debug.sf.nobootanimation 1
```

在这种情况下，屏幕将会在引导标志显示之后的某一时刻变为黑屏，并保持黑屏直到启动栏应用程序显示到主屏幕上。

## 敏捷优化

其中一个在开机动画过程中启动的系统服务就是软件包管理器。我们还没有具体的讨论过其功能，但可以说，软件包管理器管理系统中所有的.apk文件。除此之外，它还会处理.apk的安装和卸载，并帮助活动管理来解决问题。

软件包管理器的职责之一是，确保在相应的Java代码执行之前，所有DEX字节码的JIT就绪版本都是可用的。为了达到这一目标，软件包管理器的构造（软件包管理系统服务是以一个Java类来实现的）遍历系统中所有的.apk文件和.jar文件，并请求install运行dexopt命令。

这个过程应该只发生在第一次启动时。随后，/data/dalvikcache目录将包含所有JIT就绪版本的.dex文件，并且启动顺序应该大幅加快。如果你在第一次启动时观察了logcat的输出，你实际上会看到以下的条目：

```
D/dalvikvm( 32): DexOpt: --- BEGIN 'core.jar' (bootstrap=1) ---
D/dalvikvm( 62): Ignoring duplicate verify attempt on Ljava/lang/Object;
D/dalvikvm( 62): Ignoring duplicate verify attempt on Ljava/lang/Class;
D/dalvikvm( 62): DexOpt: load 349ms, verify+opt 4153ms
D/dalvikvm( 32): DexOpt: --- END 'core.jar' (success) ---
D/dalvikvm( 32): DEX prep '/system/framework/core.jar': unzip in 405ms, rewrite 5337ms
D/dalvikvm( 32): DexOpt: --- BEGIN 'bouncycastle.jar' (bootstrap=1) ---
D/dalvikvm( 63): DexOpt: load 54ms, verify+opt 779ms
D/dalvikvm( 32): DexOpt: --- END 'bouncycastle.jar' (success) ---
D/dalvikvm( 32): DEX prep '/system/framework/bouncycastle.jar': unzip in 48ms, rewrite 1023ms
D/dalvikvm( 32): DexOpt: --- BEGIN 'ext.jar' (bootstrap=1) ---
D/dalvikvm( 64): DexOpt: load 129ms, verify+opt 1497ms
D/dalvikvm( 32): DexOpt: --- END 'ext.jar' (success) ---
D/dalvikvm( 32): DEX prep '/system/framework/ext.jar' : unzip in 91ms, rewrite 1923ms
...
D/installld( 35): DexInv: --- BEGIN '/system/framework/am.jar' ---
D/dalvikvm( 95): DexOpt: load 15ms, verify+opt 58ms
D/installld( 35): DexInv: --- END '/system/framework/am.jar' (success) ---
D/installld( 35): DexInv: --- BEGIN '/system/framework/input.jar' ---
D/dalvikvm( 96): DexOpt: load 5ms, verify+opt 28ms
D/installld( 35): DexInv: --- END '/system/framework/input.jar' (success) ---
D/installld( 35): DexInv: --- BEGIN '/system/framework/pm.jar' ---
D/dalvikvm( 97): DexOpt: load 12ms, verify+opt 64ms
D/installld( 35): DexInv: --- END '/system/framework/pm.jar' (success) ---
...
D/installld( 35): DexInv: --- BEGIN '/system/app/ApplicationsProvider.apk' ---
D/dalvikvm( 249): DexOpt: load 31ms, verify+opt 110ms
D/installld( 35): DexInv: --- END '/system/app/ApplicationsProvider.apk' (success) ---
D/installld( 35): DexInv: --- BEGIN '/system/app/UserDictionaryProvider.apk' ---
D/dalvikvm( 253): DexOpt: load 19ms, verify+opt 52ms
```

```
D/installld( 35): DexInv: --- END '/system/app/UserDictionaryProvider.apk' (success) ---
D/installld( 35): DexInv: --- BEGIN '/system/app/Settings.apk' ---
D/dalvikvm( 254): DexOpt: load 155ms, verify+opt 894ms
D/installld( 35): DexInv: --- END '/system/app/Settings.apk' (success) ---
D/installld( 35): DexInv: --- BEGIN '/system/app/Launcher2.apk' ---
D/dalvikvm( 256): DexOpt: load 178ms, verify+opt 581ms
D/installld( 35): DexInv: --- END '/system/app/Launcher2.apk' (success) ---
```

起初，软件包管理器服务尚未运行，因此我们可以看到 Dalvik 直接为某些 *.jar* 文件运行 *dexopt*，而不是当软件包管理器服务请求它发生时由 *installld* 来运行。一旦软件包管理器被启动，它会按照下列的顺序来运行其余的优化进程：

1. *.jar* 文件在 *init.rc* 文件中的 *BOOTCLASSPATH* 变量中列出。
2. *.jar* 文件列为 */system/etc/permission/platform.xml* 中的库文件。
3. *.apk* 和 *.jar* 文件被发现在文件夹 */system/framework* 中。
4. *.apk* 文件被发现在 */system/app* 中。
5. *.apk* 文件被发现在 */vendor/app* 中。
6. *.apk* 文件被发现在 */data/app* 中。
7. *.apk* 文件被发现在 */data/app-private* 中。

显然，这个过程需要一定的时间。在我的四核酷睿 i7 上，它装载新编译的 2.3/ 姜饼 AOSP 的模拟器镜像需要花费 75 秒的时间，来完成它的第一次全面启动（即到主屏幕上），而且需要 24 秒时间做后续启动。在生产系统中，例如电话，这样的启动时间是不可接受的。

因此，你会很高兴地听到，实际上可以在启动时停止此优化过程，并把它放到构建时再做。你只需要在构建 AOSP 时将 *WITH\_DEXPREOPT* 构建标志设置为 *true* 即可：

```
$ make WITH_DEXPREOPT=true -j16
```

你也可以在你的设备的 *BoardConfig.mk* 中设置这个变量，避免每一次都把它添加到 *make* 命令中。在模拟器中构建的情况下，这个操作在版本 2.3/ 姜饼中不是默认执行的，但在版本 4.2/ 果冻豆中是默认执行的。

当然，构建会花费更多的时间，但在第一次启动时会显著地变快。先前提到的相同的工作站中，需要 30 分钟来构建 2.3/ 姜饼，而不是带有 *WITH\_DEXPREOPT* 标志时的 20 分钟。然而，模拟器的图像在 40 秒内出现，而不是第一次启动时在 75 秒才出现。当选项被使用时，*/data/dalvikcache* 目录最终在第一次启动之后，在目标机上被清空。相反，

在构建时，*.odex* 文件被并排放置在同一文件系统路径下，就像其相应的 *.jar* 文件和 *.apk* 文件一样。

## 应用程序启动

当系统服务启动渐入尾声时，应用程序开始被激活，包括主屏幕。正如在第 2 章介绍的，活动管理器通过发送一个 Intent.CATEGORY\_HOME 类型的事件来结束其初始化，这将导致启动栏应用程序启动，主屏幕出现。虽然这只是故事的一部分。系统服务的启动将在事实上引起相当多的应用程序启动。以下是 *ps* 命令的部分输出，该命令在一个新启动的 2.3/ 姜饼的模拟器图像上执行：

```
# ps
...
root      32      1    60832   16240 c009b74c afd0b844 S zygote
media     33      1    17976   1056 ffffffff afd0b6fc S /system/bin/mediaserver
bluetooth 34      1    1256    220 c009b74c afd0c59c S /system/bin/dbus-daemon
root      35      1     812    232 c02181f4 afd0b45c S /system/bin/installd
keystore   36      1    1744    212 c01b52b4 afd0c0cc S /system/bin/keystore
root      38      1     824    268 c00b8fec afd0c51c S /system/bin/qemud
shell     40      1     732    200 c0158eb0 afd0b45c S /system/bin/sh
root      41      1    3364    168 ffffffff 000008294 S /sbin/adbd
system    61      32   124096   26352 ffffffff afd0b6fc S system_server
app_19    113     32   80336   17400 ffffffff afd0c51c S com.android.inputmethod.
latin
radio     121     32    87112   17972 ffffffff afd0c51c S com.android.phone
system    122     32    73160   18452 ffffffff afd0c51c S com.android.systemui
app_26    132     32    76608   20812 ffffffff afd0c51c S com.android.launcher
app_1     169     32    85368   20584 ffffffff afd0c51c S android.process.acore
app_12    234     32    70752   15748 ffffffff afd0c51c S com.android.quicksearchbox
app_8     242     32    73108   16908 ffffffff afd0c51c S android.process.media
app_10    266     32    70928   16572 ffffffff afd0c51c S com.android.providers.
calendar
app_29    300     32    72764   17484 ffffffff afd0c51c S com.android.email
app_18    315     32    70272   15428 ffffffff afd0c51c S com.android.music
app_22    323     32    69712   15220 ffffffff afd0c51c S com.android.protips
app_3     335     32    71432   16756 ffffffff afd0c51c S com.cooliris.media
...
...
```

所有具有 Java 风格进程名字<sup>注2</sup> 的进程，实际上都是被自动启动的应用程序，它们的启动无须用户干预，也无论系统启动的是什么。各种系统机制导致这些应用程序启动，给定它们各自的清单文件的内容。这是一个可喜的变化，因为控制应用程序的激活需要很多比它需要的更少的内部工作，来控制很多其他方面的启动，正如我们在上面看到的。相反，它的所有都是关于小心地创建精心设计的应用程序，用于打包到 AOSP 中。

---

注 2：这些都是以点号分隔的名字，比如说，com.android.launcher 就是启动栏应用程序。

当然，还会有你想要修改常用应用程序，使其行为或启动有所不同的情况，但至少我们已经进入了应用程序的世界，这里函数更松耦合且文档更容易获得。

引导我们讨论触发器的是它导致了应用程序被激活。

## 输入法

最早启动的一个类型的应用程序就是输入法。输入法管理器服务的构造函数执行，并激活所有具有 `android.view.InputMethod` 意图过滤器的应用程序服务。例如，这就是 LatinIME 应用程序如何被激活的，该应用程序作为 `com.android.inputmethod.latin` 进程运行。

正如你通过阅读“创建一个输入法”博文看到的——该文章发表在 Android 开发者博客的博文中，输入法实际上是由精心打造的服务。

## 持久的应用程序

带有 `android:persistent="true"` 属性的应用程序，该属性在它们的 manifest 文件的 `<application>` 元素中，会在启动阶段由活动管理器自动产生。其实，即使这样的应用程序会死掉，它也将会被活动管理自动重启。

正如我刚才解释的，与普通的应用程序不同，那些被标记为持久的应用程序不受活动管理器的生命周期的管理。相反，它们在整个系统的生命周期中保持存活。这就允许我们使用这种应用程序来实现特殊的功能。比如说，状态栏和手机应用程序，运行为 `com.android.system ui` 和 `com.android.phone` 进程，它们都是持久的应用程序。

---

**警告：** 虽然应用程序开发文档介绍了 `android:persistent` 这个角色，该属性的使用是为那些在 AOSP 内构建的应用程序而保留的。

---

## 主屏幕

通常情况下，只有一个主屏幕应用程序，它反映了 `Intent.CATEGORY_HOME` 事件，该事件是由活动管理器在系统服务启动结束后发送的。在 `development/samples/Home/` 目录下有一个示例主应用程序，但是真正的主应用程序被激活是在 `packages/apps/launcher2/` 中。下面是启动器的主要活动，及其在版本 2.3/ 姜饼中的事件过滤器（4.2/ 果冻豆中也基本相同）：

```
<activity
    android:name="com.android.launcher2.Launcher"
    android:launchMode="singleTask"
    android:clearTaskOnLaunch="true"
```

```
    android:stateNotNeeded="true"
    android:theme="@style/Theme"
    android:screenOrientation="nosensor"
    android:windowSoftInputMode="stateUnspecified|adjustPan">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.HOME" />
            <category android:name="android.intent.category.DEFAULT" />
            <category android:name="android.intent.category.MONKEY"/>
        </intent-filter>
    </activity>
```

很显然，如果你想启动一个自定义的应用程序来作为主屏幕，并代替 launcher2，你将需要删除后者，并添加自己的应用程序来对相同的时间做出反应。如果有多个应用程序响应该事件，用户将会看到一个对话框，询问他们想要用哪一个作为主屏幕。

请注意，该事件并不仅仅是在启动时被发送。根据系统的状态，活动管理器将在需要把主屏幕带回到前台时，发送此事件。

## BOOT\_COMPLETED 事件

活动管理器在启动阶段还广播 Intent.BOOT\_COMPLETED 事件。这是一个常用于通知应用程序系统已经完成启动的一个事件。AOSP 中许多常备的应用程序实际上依赖于这个事件，比如说媒体供应器、日历供应器、彩信应用程序和电子邮件应用程序。下面是在 2.3/ 姜饼中媒体供应器使用的广播接收器，连同其事件过滤器（4.2/ 果冻豆中也非常相似）：

```
<receiver android:name="MediaScannerReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.MEDIA_MOUNTED" />
        <data android:scheme="file" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.MEDIA_SCANNER_SCAN_
FILE" />
        <data android:scheme="file" />
    </intent-filter>
</receiver>
```

为了能够接收该事件，应用程序必须显示地要求有权限能够这样做：

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
```

## APPWIDGET\_UPDATE 事件

除了应用程序外，应用程序窗口部件服务（APP Widget Service），它本身也是一个系统服务，将自己注册为接收 Intent.BOOT\_COMPLETED 事件。它将收到这一事件作为触发器，来通过发送 Intent.APPWIDGET\_UPDATE 事件激活系统中的所有应用程序的部件。因此，如果你已经开发了一个应用程序窗口小部件来作为你的应用程序的一部分，你的代码将在这个时刻被激活。看一看 Android 开发者文档的“应用程序窗口小部件”这一节，了解关于如何编写自己的应用程序窗口小部件的更多信息。

一些常备的 AOSP 应用程序有应用程序小部件，如快速搜索框，音乐，主屏幕提示和媒体。例如，以下是快速搜索框的应用程序窗口小部件在 manifest 文件中的声明：

```
<receiver android:name=".SearchWidgetProvider"
          android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_
          UPDATE" />
    </intent-filter>
    <meta-data android:name="android.appwidget.provider" android:
      resource="@xml/search_widget_info" />
</receiver>
```

## 工具和命令

一旦应用程序的框架及基本设置启动并运行，将会有相当多的、你可以用于查询的或是系统服务与框架之间进行交互的命令。就像第 6 章中谈及的命令，一旦 shell 到设备上就可以在命令行中使用它们。但是这些命令没有实际意义，除非应用框架在运行，否则这些命令不会产生影响。当然当你把 Android 引入到新设备和（或）应用框架调试部分时，你会发现大多数这样的命令是有用的，有时甚至是至关重要的。就像本地用户空间的命令，就文档及功能方面而言，用于框架交互的可用工具有很大的不同。尽管它们提供在新的硬件上引进 Android 或从现有的产品移除它所需要的基本功能。让我们看一看能够为你与 Android 框架之间提供交互的命令集。

---

**注意：**许多命令位于 Android 开放源代码项目（下面简称 AOSP）源文件的 *frameworks/base/cmds/* 目录下，尽管在 Android 4.2 版本的源文件中一些命令已经被移到 *frameworks/nativecmds/* 目录下。我鼓励你在使用这些命令时，查阅参考一下这些资源，因为仅仅通过查看它们已有的在线帮助文档效果往往不明显。

---

## 通用工具

对比我们将要看到的一些工具，这些工具对于框架各部分之间的交互是非常有用的，其中有些的功能是非常强大的。

### service

*service* 命令允许我们与任意在服务管理器中注册了的系统服务进行交互：

```
# service -h
Usage: service [-h|-?]
    service list
    service check SERVICE
    service call SERVICE CODE [i32 INT | s16 STR] ...
Options:
    i32: Write the integer INT into the send parcel.
    s16: Write the UTF-16 string STR into the send parcel.
```

如你所见，该命令能用于查询，同样也能用于调用来自系统服务的方法。这里例举了它是如何查询 Android 2.3 版本中已有的系统服务列表的：

```
# service list
Found 50 services:
0 phone: [com.android.internal.telephony.ITelephony]
1 iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
2 simphonebook: [com.android.internal.telephony.IIccPhoneBook]
3 isms: [com.android.internal.telephony.ISms]
4 diskstats: []
5 appwidget: [com.android.internal.appwidget.IAppWidgetService]
6 backup: [android.app.backup.IBackupManager]
7 uimode: [android.app.IUiModeManager]
8 usb: [android.hardware.usb.IUsbManager]
9 audio: [android.media.IAudioService]
10 wallpaper: [android.app.IWallpaperManager]
11 dropbox: [com.android.internal.os.IDropBoxManagerService]
12 search: [android.app.ISearchManager]
13 location: [android.location.ILocationManager]
14 devicestoragemonitor: []
15 notification: [android.app.INotificationManager]
16 mount: [IMountService]
17 accessibility: [android.view.accessibility.IAccessibilityManager]
...
...
```

接口名称设置在方括号内，允许你浏览 AOSP 源文件来找到匹配的定义接口的 *.aidl* 文件。

你也可以使用如下的命令检测一项给定的服务是否存在：

```
# service check power
Service power: found
```

最有趣的是，你可以使用 service call 去直接地调用系统服务的 Binder exposed（暴露绑定）方法。为了做到这一点，你首先需要了解服务接口。这里例举了 Android 2.3 *frameworks/base/core/java/com/android/internal/statusbar/IStatusBarService.aidl* 中定义的 *IStatusBarService* 接口（Android 4.2 版本中的接口的名称是相同的，但 *setIcon()* 函数的函数原型已经改变）：

```
...
interface IStatusBarService
{
    void expand();
    void collapse();
    void disable(int what, IBinder token, String pkg);
    void setIcon(String slot, String iconPackage, int iconId, int iconLevel);
...
}
```

请注意，*service call* 实际上需要一个方法的代码，而不仅仅是一个方法的名字。为了找到与接口中定义的方法名称匹配的代码，你需要看一下基于接口定义的 *aidl* 工具生成的代码。这里例举 *IStatusBarService.java* 文件中的有关摘录，该文件在 *out/target/common/obj/JAVA\_LIBRARIES/framework\_intermediates/src/core/java/com/android/internal/statusbar/* 目录下生成：

```
...
static final int TRANSACTION_expand = (android.os.IBinder.FIRST_CALL_
TRANSACTION + 0);
static final int TRANSACTION_collapse = (android.os.IBinder.FIRST_CALL_
TRANSACTION + 1);
static final int TRANSACTION_disable = (android.os.IBinder.FIRST_CALL_
TRANSACTION + 2);
static final int TRANSACTION_setIcon = (android.os.IBinder.FIRST_CALL_
TRANSACTION + 3);
...

```

同时注意，对于整型变量 *FIRST\_CALL\_TRANSACTION* 在 *frameworks/base/core/java/android/os/IBinder.java* 文件中有如下定义：

```
int FIRST_CALL_TRANSACTION = 0x00000001;
```

因此，*expand()* 的代码是 1，*collapse()* 的代码是 2。如下这个命令将引起状态栏的扩展：

```
# service call statusbar 1
```

与此同时如下这个命令将引起状态栏崩溃：

```
# service call statusbar 2
```

这是一种非常简单的情况，其作用相当明显且调用的方法不带任何参数。在其他情况

下，你需要更仔细地查看系统服务的 API 及理解设定的参数含义。另外，记住系统服务的接口不一定通过 `.aidl` 文件展示。在某些情况下，如活动管理，接口定义是直接通过硬编码在常规的 Java 文件中实现而不是自动生成。在基于 C 的系统服务的情况下，Binder 的编组和解组都是直接在 C 语言代码中完成。因此，尝试使用 `grep` 在 AOSP 的 `frameworks/` 目录（除了 `out/target/common/` 目录外）寻找所有的实例。

## dumpsys

另一个有趣的事情是查询系统服务的内部状态。事实上，每一个系统服务实现了一个 `dump()` 方法，并且可以通过 `dumpsys` 命令进行查询：

```
dumpsys [ <service> ]
```

默认情况下，如果没有系统服务名称作为参数，`dumpsys` 将先输出系统服务列表，然后解析它们的状态：

```
# dumpsys
Currently running services:
  SurfaceFlinger
  accessibility
  account
  activity
  alarm
  appwidget
  audio
  backup
  battery
  batteryinfo
  clipboard
  connectivity
  content
  cpuinfo
  device_policy
  devicestoragemonitor
  diskstats
  dropbox
  entropy
  hardware
  ...
-----
DUMP OF SERVICE SurfaceFlinger:
+ Layer 0x1e5788
  z=    21000, pos=( 0, 0), size=( 320, 480), needsBlending=0, needsDither=0, invalidate=0, alpha=0xff, flags=0x00000000, tr=[1.00, 0.00][0.00, 1.00]
    name=com.android.internal.service.wallpaper.ImageWallpaper
    client=0x1ed2a8, identity=3
    [ head= 1, available= 2, queued= 0 ] reallocMask=00000000, identity=3, status=0
    format= 4, [320x480:320] [320x480:320], freezeLock=0x0, bypass=0, dq-q-time=2034
us
  Region transparentRegion (this=0x1e5918, count=1)
```

```
[ 0, 0, 0]
Region transparentRegionScreen (this=0x1e57bc, count=1)
[ 0, 0, 0]
Region visibleRegionScreen (this=0x1e5798, count=1)
[ 0, 25, 320, 480]
+ Layer 0x268b70
    z= 21005, pos=( 0, 0), size=( 320, 480), needsBlending=1, needsDithering=1, invalidate=0, alpha=0xff, flags=0x00000000, tr=[1.00, 0.00][0.00, 1.00]
...
-----
DUMP OF SERVICE accessibility:
-----
DUMP OF SERVICE account:
Accounts: 0
Active Sessions: 0

RegisteredServicesCache: 1 services
ServiceInfo: AuthenticatorDescription {type=com.android.exchange}, ComponentInfo{com.android.email/com.android.email.service.EasAuthenticatorService}, uid 10029
-----
DUMP OF SERVICE activity:
Providers in Current Activity Manager State:
Published content providers (by class):
* ContentProviderRecord{4060d0e0 com.android.deskclock.AlarmProvider}
...

```

显然，其输出是非常详细的，最重要的是，它需要你理解相应的系统服务的内部情况。如果你实现你自己的系统服务，能够查询其在运行时的状态是至关重要的。当然，如果你对解析系统服务状态不是很感兴趣，你只需要提供你想获取信息的特定服务的名称作为 *dumpsys* 命令的一个参数：

```
# dumpsys power
Power Manager State:
mIsPowered=true mPowerState=1 mScreenOffTime=46793204 ms
mPartialCount=1
mWakeLockState=SCREEN_ON_BIT
mUserState=
mPowerState=SCREEN_ON_BIT
mLocks.gather=SCREEN_ON_BIT
mNextTimeout=94351 now=46880555 -46786s from now
mDimScreen=true mStayOnConditions=1
mScreenOffReason=0 mUserState=0
mBroadcastQueue={-1,-1,-1}
mBroadcastWhy={0,0,0}
mPokekey=0 mPokeAwakeonSet=false
...
```

## dumpstate

在某些情况下，你想要做的是获取整个系统的一个快照，而不仅仅只是系统服务。这是 *dumpstate* 命令所关注的。事实上，你可能会记得我们在第 6 章讲 *adb* 的错误返回

报告程序时关于这个命令的讨论，因为 *dumpstate* 提供了错误返回报告程序返回的信息。这里例举了在 Android 2.3 姜饼版本中 *dumpstate* 的详细帮助文档：

```
# dumpstate -h
usage: dumpstate [-d] [-o file] [-s] [-z]
-d: append date to filename (requires -o)
-o: write to file (instead of stdout)
-s: write output to control socket (for init)
-z: gzip output (requires -o)
```

在 Android 4.2 版本中，*dumpstate* 的功能已经得到扩展：

```
root@android:/ # dumpstate -h
usage: dumpstate [-b soundfile] [-e soundfile] [-o file [-d] [-p] [-z]] [-s] [-q]
-o: write to file (instead of stdout)
-d: append date to filename (requires -o)
-z: gzip output (requires -o)
-p: capture screenshot to filename.png (requires -o)
-s: write output to control socket (for init)
-b: play sound file instead of vibrate, at beginning of job
-e: play sound file instead of vibrate, at end of job
-q: disable vibrate
```

如果你不使用任何参数调用它，它照样可以查询系统的好几个部分来给你提供完整的系统状态的快照。

```
# dumpstate
=====
== dumpstate: 2012-10-10 03:15:26
=====

Build: generic-eng 2.3.4 GINGERBREAD eng.karim.20120913.141233 test-keys
Bootloader: unknown
Radio: unknown
Network: Android
Kernel: Linux version 2.6.29-00261-g0097074-dirty (digit@digit.mtv.corp.google.com)
(gcc version 4.4.0 (GCC) ) #20 Wed Mar 31 09:54:02 PDT 2010
Command line: qemu=1 console=ttyS0 android.checkjni=1 android.qemud=ttyS1 android.
ndns=1

----- MEMORY INFO (/proc/meminfo) -----
MemTotal: 94096kB
MemFree: 1296 kB
Buffers: 0 kB
Cached: 32424 kB
...
----- CPU INFO (top -n 1 -d 1 -m 30 -t) -----

User 2%, System 11%, IOW 33%, IRQ 0%
User 3 + Nice 0 + Sys 15 + Idle 67 + IOW 42 + IRQ 0 + SIRQ 0 = 127
```

PID	TID	CPU%	S	VSS	RSS	PCY	UID	Thread	Proc
-----	-----	------	---	-----	-----	-----	-----	--------	------

```

      339  339  13% R    976K   440K fg shell top          top
      121  121  0% S 86100K 18484K fg radio m.android.phone com.android.phone
      3     3  0% S      OK      OK fg root ksoftirqd/0
      4     4  0% S      OK      OK fg root events/0
----- PROC RANK (procrank) -----

```

PID	Vss	Rss	Pss	Uss	cmdline
61	25676K	25076K	10581K	8552K	system_server
124	21412K	21412K	6851K	4908K	com.android.launcher
122	19268K	19268K	5698K	4388K	com.android.systemui
121	18484K	18484K	4744K	3568K	com.android.phone
295	18176K	18176K	4337K	3132K	com.android.email
115	17836K	17836K	4118K	2960K	com.android.inputmethod.latin

```

...
----- VIRTUAL MEMORY STATS (/proc/vmstat) -----
nr_free_pages 553
nr_inactive_anon 6708
nr_active_anon 6068
nr_inactive_file 3449
nr_active_file 2062
...
----- VMALLOC INFO (/proc/vmallocinfo) -----
0xc684c000-0xc684e000    8192 __arm_ioremap_pfn+0x68/0x2fc ioremap
0xc6850000-0xc6852000    8192 __arm_ioremap_pfn+0x68/0x2fc ioremap
0xc6854000-0xc6856000    8192 __arm_ioremap_pfn+0x68/0x2fc ioremap
0xc6880000-0xc68a1000 135168 binder_mmap+0xb4/0x200 ioremap
...
----- SLAB INFO (/proc/slabinfo) -----
slabinfo - version: 2.1
# name           <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
: tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_s
labs> <sharedavail>
rpc_buffers        8     8   2048    2    1 : tunables    24   12   0 : sla
bdata      4     4     0
rpc_tasks         8     24   160    24   1 : tunables   120   60   0 : sla
bdata      1     1     0
rpc_inode_cache   0     0   416     9   1 : tunables    54   27   0 : sla
bdata      0     0     0
bridge_fdb_cache  0     0    64    59   1 : tunables   120   60   0 : sla
bdata      0     0     0
...
----- ZONEINFO (/proc/zoneinfo) -----
Node 0, zone Normal
  pages free    550
  min     312
  low    390
  high   468
  scanned 0 (aa: 0 ia: 0 af: 26 if: 0)
...
----- SYSTEM LOG (logcat -v time -d *:v) -----
10-10 01:38:02.762 I/DEBUG   ( 30): debuggerd: Feb 26 2012 21:06:53
10-10 01:38:02.882 I/Netd    ( 29): Netd 1.0 starting
10-10 01:38:02.932 D/qemud   ( 38): entering main loop
10-10 01:38:02.972 I/Vold    ( 28): Vold 2.1 (the revenge) firing up
10-10 01:38:02.972 D/Vold    ( 28): USB mass storage support is not enabled in

```

```

the kernel
...
----- VM TRACES JUST NOW (/data/anr/traces.txt.bugreport: 2012-10-10 03:15:26)
-----
----- pid 61 at 2012-10-10 03:15:26 -----
Cmd line: system_server

DALVIK THREADS:
(mutexes: tll=0 tsl=0 tscl=0 ghl=0 hwl=0 hwll=0)
"main" prio=5 tid=1 NATIVE
| group="main" sCount=1 dsCount=0 obj=0x4001f1a8 self=0xce48
| sysTid=61 nice=0 sched=0/0 cgrp=default handle=-1345006528
| schedstat=( 1116789165 392598071 782 )
at com.android.server.SystemServer.init1(Native Method)
at com.android.server.SystemServer.main(SystemServer.java:625)
...
----- EVENT LOG (logcat -b events -v time -d *:v) -----
10-10 01:38:03.642 I/boot_progress_start( 32): 5126
10-10 01:38:04.221 I/boot_progress_preload_start( 32): 5706
10-10 01:38:04.251 I/dvm_gc_info( 32): [8825198673194415294,-90644969689662529
97,-4012584086963399109,0]
10-10 01:38:04.281 I/dvm_gc_info( 32): [8825198673194406507,-92148046065296736
57,-4012584086963329465,0]
10-10 01:38:04.331 I/dvm_gc_info( 32): [8825198673194406993,-91348657131437777
12,-4012584086963259824,0]
10-10 01:38:04.371 I/dvm_gc_info( 32): [8825198673194415172,-91399322627244589
19,-4012584086963149223,0]
...
----- RADIO LOG (logcat -b radio -v time -d *:v) -----
10-10 01:58:04.988 D/AT      ( 31): AT< +CSQ: 7,99
10-10 01:58:04.988 D/AT      ( 31): AT< OK
10-10 01:58:04.988 D/RILJ    ( 121): [0114]< SIGNAL_STRENGTH {7, 99, 0, 0, 0
, 0, 0}
10-10 01:58:24.998 D/RILJ    ( 121): [0115]> SIGNAL_STRENGTH
10-10 01:58:25.008 D/RIL     ( 31): onRequest: SIGNAL_STRENGTH
...
----- NETWORK INTERFACES (netcfg) -----
*** exec(netcfg): Permission denied
*** netcfg: Exit code 255
[netcfg: 0.1s elapsed]

----- NETWORK ROUTES (/proc/net/route) -----
Iface Destination     Gateway       Flags  RefCnt   Use     Metric   Mask
          MTU      Window     IRTT
etho    0002000A        00000000    0001     0        0        0        00FFFFFF
          0          0          0
etho    00000000        0202000A    0003     0        0        0        00000000
          0          0          0

----- ARP CACHE (/proc/net/arp) -----
IP address      HW type      Flags      HW address      Mask      Device
10.0.2.2        0x1         0x2        52:54:00:12:35:02      *        eth0

----- SYSTEM PROPERTIES -----
[dalvik.vm.heapsize]: [16m]

```

```

[dalvik.vm.stack-trace-file]: [/data/anr/traces.txt]
[dev.bootcomplete]: [1]
[gsm.current.phone-type]: [1]
[gsm.defaultpdcontext.active]: [true]
...
----- KERNEL LOG (dmesg) -----
Initializing cgroup subsys cpu
Linux version 2.6.29-00261-g0097074-dirty (digit@digit.mtv.corp.google.com) (gcc
version 4.4.0 (GCC) ) #20 Wed Mar 31 09:54:02 PDT 2010
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00093177
CPU: VIVT data cache, VIVT instruction cache
Machine: Goldfish
Memory policy: ECC disabled, Data cache writeback
On node 0 totalpages: 24576
...
----- KERNEL WAKELOCKS (/proc/wakelocks) -----
name      count    expire_count    wake_count      active_since    total_time
sleep_time    max_time        last_change
"alarm"   106       0           0           1632946980       0           41697763
5822030632794
"KeyEvents" 27       0           0           123592046        0           94064309
    27084159991
"evento-61"  26       0           0           48780811        0           12891126
    27083608920
"radio-interface" 3       0           0           0           3472899963       0
1459986280     25362482435
...
----- KERNEL CPUFREQ (/sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state)
-----
*** /sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state: No such file or di
rectory

----- VOLD DUMP (vdc dump) -----
000 Dumping loop status
000 Dumping DM status
000 Dumping mounted filesystems
000 rootfs / rootfs ro 0 0
...
----- SECURE CONTAINERS (vdc asec list) -----
200 asec operation succeeded
[vdc: 0.1s elapsed]

----- PROCESSES (ps -P) -----
USER      PID      PPID     VSZIE    RSS      PCY      WCHAN      PC      NAME
root      1        0        268     180      fg      c009b74c  0000875c S /init
root      2        0        0        0      fg      c004e72c  00000000 S kthreadd
root      3        2        0        0      fg      c003fdc8  00000000 S ksoftirqd/0
root      4        2        0        0      fg      c004b2c4  00000000 S events/0
root      5        2        0        0      fg      c004b2c4  00000000 S khelper
root      6        2        0        0      fg      c004b2c4  00000000 S suspend
...
----- PROCESSES AND THREADS (ps -t -p -P) -----
USER      PID      PPID     VSZIE    RSS      PRIO     NICE     RTPRI     SCHED     PCY      WCHAN      PC
      NAME
root      1        0        268     180      20       0        0        0      fg      c009b74c  0000875c

```

```

S /init
root      2      0      0      0     15    -5      0      0      fg   c004e72c 00000000
      S kthreadd
root      3      2      0      0     15    -5      0      0      fg   c003fdc8 00000000
      S ksoftirqd/0
root      4      2      0      0     15    -5      0      0      fg   c004b2c4 00000000
      S events/0
root      5      2      0      0     15    -5      0      0      fg   c004b2c4 00000000
      S khelper
root      6      2      0      0     15    -5      0      0      fg   c004b2c4 00000000
S suspend
...
----- LIBRANK (librank) -----
RSStat      VSS      RSS      PSS      USS  Name/PID
16658K
          6980K    6980K    3218K    2896K  /dev/ashmem/dalvik-heap
          5208K    5208K    1371K    1048K  system_server [61]
          5272K    5272K    1343K    1012K  com.android.launcher [124]
          5272K    5272K    1343K    1012K  com.android.phone [121]
...
----- BINDER FAILED TRANSACTION LOG (/sys/kernel/debug/binder/failed_transaction_log) -----
*** /sys/kernel/debug/binder/failed_transaction_log: No such file or directory

----- BINDER TRANSACTION LOG (/sys/kernel/debug/binder/transaction_log) -----
*** /sys/kernel/debug/binder/transaction_log: No such file or directory

----- BINDER TRANSACTIONS (/sys/kernel/debug/binder/transactions) -----
*** /sys/kernel/debug/binder/transactions: No such file or directory

----- BINDER STATS (/sys/kernel/debug/binder/stats) -----
*** /sys/kernel/debug/binder/stats: No such file or directory

----- BINDER STATE (/sys/kernel/debug/binder/state) -----
*** /sys/kernel/debug/binder/state: No such file or directory

----- FILESYSTEMS & FREE SPACE (df) -----
Filesystem      1K-blocks      Used  Available  Use%  Mounted on
tmpfs           47048        32     47016     0%   /dev
tmpfs           47048        0     47048     0%   /mnt/asec
tmpfs           47048        0     47048     0%   /mnt/obb
/dev/block/mtdblock0  65536      65536        0 100%  /system
/dev/block/mtdblock1  65536      25292     40244    39%  /data
/dev/block/mtdblock2  65536      1156     64380     2%  /cache
[df: 0.is elapsed]

----- PACKAGE SETTINGS (/data/system/packages.xml: 2012-10-10 01:38:16) -----
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<packages>
<last-platform-version internal="10" external="0" />
...
----- PACKAGE UID ERRORS (/data/system/uiderrors.txt: 2012-09-24 21:06:14) -----
--
2012-09-24 21:06: No settings file; creating initial state

----- LAST KMSG (/proc/last_kmsg) -----

```

```
*** /proc/last_kmsg: No such file or directory

----- LAST RADIO LOG (parse_radio_log /proc/last_radio_log) -----
*** exec(parse_radio_log): Permission denied
*** parse_radio_log: Exit code 255
[parse_radio_log: 0.1s elapsed]

----- LAST PANIC CONSOLE (/data/dontpanic/apanic_console) -----
*** /data/dontpanic/apanic_console: No such file or directory

----- LAST PANIC THREADS (/data/dontpanic/apanic_threads) -----
*** /data/dontpanic/apanic_threads: No such file or directory

----- BLOCKED PROCESS WAIT-CHANNELS -----
----- BACKLIGHTS -----
LCD brightness=*** /sys/class/leds/lcd-backlight/brightness: No such file or directory
Button brightness=*** /sys/class/leds/button-backlight/brightness: No such file or directory
Keyboard brightness=*** /sys/class/leds/keyboard-backlight/brightness: No such file or directory
ALS mode=*** /sys/class/leds/lcd-backlight/als: No such file or directory
LCD driver registers:
*** /sys/class/leds/lcd-backlight/registers: No such file or directory

=====
== Android Framework Services
=====
----- DUMPSYS (dumpsys) -----
Currently running services:
    SurfaceFlinger
...
...
```

在大多数情况下，如你所见，*dumpstate* 命令实际上是调用其他的命令如 *logcat*、*dumpsys*、*ps* 来检索它的信息。同样其输出信息非常冗长。

## rawbu

在一些情况下，你可能想要备份和恢复 */data* 目录下的内容。你可以使用 *rawbu* 命令来完成这项工作，*rawbu* 命令用法如下：

```
# rawbu help
Usage: rawbu COMMAND [options] [backup-file-path]
commands are:
    help      Show this help text.
    backup    Perform a backup of /data.
    restore   Perform a restore of /data.
options include:
    -h        Show this help text.
    -a        Backup all files.
```

The *rawbu* command allows you to perform low-level

backup and restore of the /data partition. This is where all user data is kept, allowing for a fairly complete restore of a device's state. Note that because this is low-level, it will only work across builds of the same (or very similar) device software.

这里例举了如何用它创建一个备份：

```
# rawbu backup /sdcard/backup.dat
Stopping system...
Backing up /data to /sdcard/backup.dat...
Saving dir /data/local...
Saving dir /data/local/tmp...
Saving dir /data/app-private...
Saving dir /data/app...
Saving dir /data/property...
Saving file /data/property/persist.sys.localevar...
Saving file /data/property/persist.sys.country...
Saving file /data/property/persist.sys.language...
Saving file /data/property/persist.sys.timezone...
...
Backup complete! Restarting system...
```

该命令做的第一件事就是停止 Zygote，从而停止所有的系统服务。然后创建一个进程来从 /data 复制所有的数据，通过重启 Zygote 来结束复制，一旦数据阻塞，你能在稍后还原它：

```
# rawbu restore /sdcard/backup.dat
Stopping system...
Wiping contents of /data...
warning -- rmdir() error on '/data/system': Directory not empty
warning -- rmdir() error on '/data/system': Directory not empty
Restoring from /sdcard/backup.dat to /data...
Restoring dir /data/local...
Restoring dir /data/local/tmp...
Restoring dir /data/app-private...
Restoring dir /data/app...
...
Restore complete! Restarting system, cross your fingers...
```

显然，这个命令的输出意味着这是一个弱操作，同时你也应该意识到结果会变动。

## 特定服务工具

正如我们前面看到的，已经存在许多的系统服务。通常情况下，使用这些系统服务需要写代码来与其 Binder-exposed 的 API 以某种方式（形状或形式）进行交互。然而，在某些情况下，AOSP 包括命令行实用工具来直接与某些系统服务进行交互。这些工具是非常强大，允许我们通过命令行直接进入 Android 的功能。这开启了在成品或是开发期间使用大量下列工具作为脚本的一部分的先河。

## 绕过 Android 的权限系统

系统服务的 API 通常由 Android 的权限系统所保护，这就需要应用程序在清单文件中前期声明它们需要的权限。一般的，一项系统服务在响应调用者请求时会检查它的调用者是否具有相应的权限。这部分需要检查调用者的进程号（PID）及使用包管理服务来验证原始.apk 文件的正确性。

但是有一种情况绕过所有的保护：当调用者作为超级用户（root）运行。事实上，如果你看看活动管理者的权限检测代码，这些代码同样也用于其他系统服务检查权限，你会明白在文件 *frameworks/base/services/java/com/android/server/am/ActivityManagerService.java* 中的如下这段代码（在 Android 2.3/gingerbread 版本中）：

```
int checkComponentPermission(String permission, int pid, int uid,
...
    // root、system server 和我们自己的 process 开始工作
    if (uid == 0 || uid == Process.SYSTEM_UID || pid == MY_PID ||
        !Process.supportsProcesses()) {
        return PackageManager.PERMISSION_GRANTED;
    }
...
}
```

在 Android 4.2 (Jelly Bean) 版本中，你将发现取而代之的是如下代码：

```
int checkComponentPermission(String permission, int pid, int uid,
...
    if (pid == MY_PID) {
        return PackageManager.PERMISSION_GRANTED;
    }

    return ActivityManager.checkComponentPermission(permission, uid,
        owningUid, exported);
}
```

在文件 *frameworks/base/core/java/android/app/ActivityManager.java* 中 *ActivityManager.checkComponentPermission()* 函数定义如下：

```
public static int checkComponentPermission(String permission, int uid,
    int owningUid, boolean exported) {
    // root、system server 开始工作
    if (uid == 0 || uid == Process.SYSTEM_UID) {
        return PackageManager.PERMISSION_GRANTED;
    }
...
}
```

因此，在 AOSP 的所有版本中，任何你在这里看到的跟一个系统服务交互的命令通常可以毫无障碍地从系统服务中要求它们需要的东西。因此，当你以 root 的身份运行，与系统服务交互的时候你必须非常小心。同样，如果你写一个命令行工具，模仿本节讨论的与系统服务交互的许多命令的方式时你也要非常小心。

## am

正如之前提到过的，最重要的系统服务中的一个就是活动管理器（Activity Manager）。那么这里存在一个直接调用它功能的命令也就不足为怪了。这里例举它在 Android 2.3 版本中的一个在线帮助文档：

```
# am
usage: am [subcommand] [options]

start an Activity: am start [-D] [-W] <INTENT>
-D: enable debugging
-W: wait for launch to complete

start a Service: am startservice <INTENT>

send a broadcast Intent: am broadcast <INTENT>

start an Instrumentation: am instrument [flags] <COMPONENT>
-r: print raw results (otherwise decode REPORT_KEY_STREAMRESULT)
-e <NAME> <VALUE>: set argument <NAME> to <VALUE>
-p <FILE>: write profiling data to <FILE>
-w: wait for instrumentation to finish before returning

start profiling: am profile <PROCESS> start <FILE>
stop profiling: am profile <PROCESS> stop

start monitoring: am monitor [--gdb <port>]
--gdb: start gdbserve on the given port at crash/ANR

<INTENT> specifications include these flags:
[-a <ACTION>] [-d <DATA_URI>] [-t < MIME_TYPE >]
[-c <CATEGORY>] [-c <CATEGORY>] ...
[-e|--es <EXTRA_KEY> <EXTRA_STRING_VALUE> ...]
[--esn <EXTRA_KEY> ...]
[--ez <EXTRA_KEY> <EXTRA_BOOLEAN_VALUE> ...]
[-e|--ei <EXTRA_KEY> <EXTRA_INT_VALUE> ...]
[-n <COMPONENT>] [-f <FLAGS>]
[--grant-read-uri-permission] [--grant-write-uri-permission]
[--debug-log-resolution]
[--activity-brought-to-front] [--activity-clear-top]
[--activity-clear-when-task-reset] [--activity-exclude-from-recents]
[--activity-launched-from-history] [--activity-multiple-task]
[--activity-no-animation] [--activity-no-history]
[--activity-no-user-action] [--activity-previous-is-top]
[--activity-reorder-to-front] [--activity-reset-task-if-needed]
[--activity-single-top]
[--receiver-registered-only] [--receiver-replace-pending]
[<URI>]
```

---

**注意：**在 Android 4.2 果冻豆版本中，*am* 的功能被扩充了，同样也有在线帮助文档。既然后者的内容覆盖了三页，将它完整地印刷在本书中是不切实际的。前一段是足够目前使

用的介绍；不过，我仍然鼓励你阅读 *am* 命令在 Android 4.2 果冻豆版本中的在线帮助文档。

正如我们在第 2 章所见，已经有四种可用的应用程序开发的组件类型：活动、服务、广播接收器和内容提供者。前三个类型的组件是通过意图（Intent）激活，*am* 命令一个主要的特点是它拥有直接从命令行发送意图的能力。

这里例举了怎样使用 *am* 命令浏览特定网站和相关日志摘录：

```
# am start -a android.intent.action.VIEW -d http://source.android.com
Starting: Intent { act=android.intent.action.VIEW dat=http://source.android.com }

# logcat
...
D/AndroidRuntime( 786):
D/AndroidRuntime( 786): >>>> AndroidRuntime START com.android.internal.os.RuntimeInit <<<<
D/AndroidRuntime( 786): CheckJNI is ON
D/AndroidRuntime( 786): Calling main entry com.android.commands.am.Am
I/ActivityManager( 62): Starting: Intent { act=android.intent.action.VIEW dat=http://source.android.com flg=0x10000000 cmp=com.android.browser/.BrowserActivity } from pid 786
I/ActivityManager( 62): Start proc com.android.browser for activity com.android.browser/.BrowserActivity: pid=794 uid=10015 gids={3003, 1015}
D/AndroidRuntime( 786): Shutting down VM
D/dalvikvm( 786): GC_CONCURRENT freed 100K, 69% free 317K/1024K, external OK/OK , paused 1ms+1ms
D/jdwp ( 786): adbd disconnected
I/ActivityThread( 794): Pub browser: com.android.browser.BrowserProvider
I/BrowserSettings( 794): Selected search engine: ActivitySearchEngine{android.app.SearchableInfo@40593270}
D/dalvikvm( 794): GC_CONCURRENT freed 447K, 51% free 2909K/5831K, external 934K/1038K, paused 5ms+14ms
I/ActivityManager( 62): Displayed com.android.browser/.BrowserActivity: +1s924ms
D/dalvikvm( 794): GC_EXTERNAL_ALLOC freed 51K, 50% free 2953K/5831K, external 951K/1038K, paused 62ms
...
...
```

这是一个相当简单的例子。让我们来看一下一些定制化的情况。这里例举了来自一个定制的应用中的一个广播接收者：

```
<receiver android:name="FastBirdApproaching">
    <intent-filter>
        <action android:name="com.acme.coyotebirdmonitor.FAST_BIRD"/>
    </intent-filter>
</receiver>
```

下面是相应的代码：

```
public class FastBirdApproaching extends BroadcastReceiver {
    private static final String TAG = "FastBirdApproaching";

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub
        Log.i(TAG, "*****");
        Log.i(TAG, "Meep Meep!");
        Log.i(TAG, "*****");
    }
}
```

这里例举了如何使用 *am* 触发此广播接收器以及在日志中输出结果：

```
# am broadcast -a com.acme.coyotebirdmonitor.FAST_BIRD
Broadcasting: Intent { act=com.acme.coyotebirdmonitor.FAST_BIRD }
Broadcast completed: result=0

# logcat
...
I/ActivityManager( 62): Start proc com.acme.coyotebirdmonitor for broadcast co
m.acme.coyotebirdmonitor/.FastBirdApproaching: pid=466 uid=10029 gids={}
I/FastBirdApproaching( 466): ****
I/FastBirdApproaching( 466): Meep Meep!
I/FastBirdApproaching( 466): ****
...
...
```

正如你在在线帮助文档里面可以看到的，可以指定一个关于要发送的意图的细节。尽管前两个示例使用隐式意图，你也可以发显示的意图激活指定的组件：

```
# am start -n com.android.settings/.Settings
```

在这种情况下，这将启动应用程序在系统中的设置活动。更有趣的是 *am* 命令可以用一种你不能重复使用官方出版的应用开发的 API 的方式启动元件。这是因为它是作为 AOSP 的一部分创建的，因此它存在一些隐藏的只对创建 AOSP 代码可用的调用。

*am* 命令实际上是一种 shell 脚本，正如你在 *frameworks/based/cmds/am/am/* 目录下见到的一样：

```
# Script to start "am" on the device, which has a very rudimentary
# shell.
#
base=/system
export CLASSPATH=$base/framework/am.jar
exec app_process $base/bin com.android.commands.am.Am "$@"
```

上面这段脚本使用 *app\_process* 来启动 Java 代码实现 *am* 的功能。命令行里传递的所有参数实际上传递给了 Java 代码。

*am* 命令同样也可用于仪器、分析、监控。查看一下 Android 开发手册的“测试基础”以及“从其他的 IDE 测试”章节获取更多的关于 Android 测试和 *am instrument* 命令的使用。

*am profile* 命令允许我们生成的数据通过 *traceview* 命令在主机上进行可视化。在 Android 开发者手册中的相关部分你可以找到更多的关于 *traceview* 命令的信息。请注意，文档说有两种方法来创建跟踪文件，*am* 命令作为其中的一种并没有列举出来。

最后，*am monitor* 命令通过活动管理器允许我们监控程序运行。这里例举了一个场景，我先启动该命令然后启动几个应用：

```
# am monitor
Monitoring activity manager... available commands:
(q)uit: finish monitoring
** Activity starting: com.android.browser
** Activity resuming: com.android.launcher
** Activity starting: com.android.settings
** Activity resuming: com.android.launcher
** Activity starting: com.android.browser
** Activity starting: com.android.launcher
...
...
```

请注意，当你启动一个应用，单击回退键，该命令报告启动器正在重启状态 (resuming)，而如果你单击主页按钮，启动报告为开始状态 (starting)。这种监控能力也将让你捕获 ANRS (Application Not Responding，应用程序不响应)，使你附上 GDB 工具到一个崩溃进程上。

---

**注意：**不要让关于 *am* 的简短的报道误导你：这是一个非常强大且有用命令，你应该保持良好的心态。如果你需要脚本从命令行启动你的应用程序，你会发现这个命令是非常有用的。

---

## pm

另一个非常重要的系统服务是包管理器，跟活动管理器类似，它拥有它自己的命令行工具。这里例举在 Android 2.3 姜饼版本下它的在线帮助文档：

```
# pm
usage: pm [list|path|install|uninstall]
        pm list packages [-f] [-d] [-e] [-u] [FILTER]
        pm list permission-groups
        pm list permissions [-g] [-f] [-d] [-u] [GROUP]
        pm list instrumentation [-f] [TARGET-PACKAGE]
        pm list features
        pm list libraries
        pm path PACKAGE
```

```
pm install [-l] [-r] [-t] [-i INSTALLER_PACKAGE_NAME] [-s] [-f] PATH  
pm uninstall [-k] PACKAGE  
pm clear PACKAGE  
pm enable PACKAGE_OR_COMPONENT  
pm disable PACKAGE_OR_COMPONENT  
pm setInstallLocation [0/auto] [1/internal] [2/external]
```

The list packages command prints all packages, optionally only those whose package name contains the text in FILTER. Options:

- f: see their associated file.
- d: filter to include disabled packages.
- e: filter to include enabled packages.
- u: also include uninstalled packages.

The list permission-groups command prints all known permission groups.

The list permissions command prints all known permissions, optionally only those in GROUP. Options:

- g: organize by group.
- f: print all information.
- s: short summary.
- d: only list dangerous permissions.
- u: list only the permissions users will see.

The list instrumentation command prints all instrumentations, or only those that target a specified package. Options:

- f: see their associated file.

The list features command prints all features of the system.

The path command prints the path to the .apk of a package.

The install command installs a package to the system. Options:

- l: install the package with FORWARD\_LOCK.
- r: reinstall an existing app, keeping its data.
- t: allow test .apks to be installed.
- i: specify the installer package name.
- s: install package on sdcard.
- f: install package on internal flash.

The uninstall command removes a package from the system. Options:

- k: keep the data and cache directories around.
- after the package removal.

The clear command deletes all data associated with a package.

The enable and disable commands change the enabled state of a given package or component (written as "package/class").

The getInstallLocation command gets the current install location

- 0 [auto]: Let system decide the best location
- 1 [internal]: Install on internal device storage
- 2 [external]: Install on external media

```
The setInstallLocation command changes the default install location
0 [auto]: Let system decide the best location
1 [internal]: Install on internal device storage
2 [external]: Install on external media
```

---

注意：跟 *am* 类似，*pm* 的功能经过多个版本不断扩大，Android 4.2 果冻豆版本的关于该工具的在线帮助文档比本书介绍得更加合理、恰当。我仍然鼓励你去看一下。

---

幸运的是，该命令实际上是相当有据可查的，正如你在上面的输出中看到的一样。尽可能简洁地列出已安装的软件包，如下所示：

```
# pm list packages
package:android
package:android.tts
package:com.android.bluetooth
package:com.android.browser
package:com.android.calculator2
package:com.android.calendar
package:com.android.camera
package:com.android.certinstaller
package:com.android.contacts
package:com.android.defcontainer
...
...
```

安装一个应用程序（被 *adb* 使用的安装命令在最后一章节中讲述），下面以安装 *FastBirds.apk* 为例来进行展示如下：

```
# pm install FastBirds.apk
pkg: FastBirds.apk
Success
```

注意移除一个应用程序需要知道它的包的名称，而不是原始的 *.apk* 的名字，下面展示移除应用：

```
# pm uninstall com.acme.fastbirds
Success
```

*pm* 同样也是启动 Java 代码的 shell 脚本：

```
# Script to start "pm" on the device, which has a very rudimentary
# shell.
#
base=/system
export CLASSPATH=$base/framework/pm.jar
exec app_process $base/bin com.android.commands.pm.Pm "$@"
```

---

注意：正如 *am* 命令，*pm* 命令有许多的知识点，在本书中无法一一再去讲解。我鼓励你去深入了解它的更多用法，对开发或者产品中的脚本非常有用。

---

## SVC

不像前面的两个命令，*svc* 有点像瑞士军刀，旨在给你提供控制多个系统服务的能力。这里例举了在 Android 2.3 版本中关于该命令的在线帮助文档：

```
# svc
Available commands:
    help Show information about the subcommands
    power Control the power manager
    data Control mobile data connectivity
    wifi Control the Wi-Fi manager
```

在 Android 4.2 版本中关于该命令的在线帮助文档附加了对 USB 的处理：

```
root@android:/ # svc
Available commands:
    help Show information about the subcommands
    power Control the power manager
    data Control mobile data connectivity
    wifi Control the Wi-Fi manager
    usb Control Usb state
```

请注意，*svc* 命令的能力受限于允许和禁止指定的系统服务的行为：

```
# svc help power
Control the power manager

usage: svc power stayon [true|false|usb|ac]
        Set the 'keep awake while plugged in' setting.

# svc help data
Control mobile data connectivity

usage: svc data [enable|disable]
        Turn mobile data on or off.

        svc data prefer
        Set mobile as the preferred data network

# svc help wifi
Control the Wi-Fi manager

usage: svc wifi [enable|disable]
        Turn Wi-Fi on or off.

        svc wifi prefer
        Set Wi-Fi as the preferred data network
```

总而言之，我们应该了解 *svc*，但你不大可能会经常使用它。像 *am*、*pm* 命令，*svc* 也是一个使用 *app\_process* 来启动 Java 代码的脚本。

## ime

*ime* 指令让你与输入法系统服务通信来控制可用输入法的系统使用，在 Android 2.3 与 4.2 版本中，该命令是相同的：

```
# ime
usage: ime list [-a] [-s]
    ime enable ID
    ime disable ID
    ime set ID

The list command prints all enabled input methods. Use
the -a option to see all input methods. Use
the -s option to see only a single summary line of each.

The enable command allows the given input method ID to be used.

The disable command disallows the given input method ID from use.

The set command switches to the given input method ID.
```

这里例举了在 Android 2.3 版本模拟器中可用的输入法列表，例如：

```
# ime list
com.android.inputmethod.latin/.LatinIME:
    mId=com.android.inputmethod.latin/.LatinIME mSettingsActivityName=com.android.
inputmethod.latin.LatinIMESettings
    mIsDefaultResId=0x7f080001
    Service:
        priority=0 preferredOrder=0 match=0x108000 specificIndex=-1 isDefault=false
    ServiceInfo:
        name=com.android.inputmethod.latin.LatinIME
        packageName=com.android.inputmethod.latin
        labelRes=0x7f0c001f nonLocalizedLabel=null icon=0x0
        enabled=true exported=true processName=com.android.inputmethod.latin
        permission=android.permission.BIND_INPUT_METHOD
```

再次强调，*ime* 命令在一个脚本中使用 *app\_process* 启动 Java 代码。正如 *svc* 命令，*ime* 命令是一个值得记住、但可能不经常使用的命令。

## input

*input* 连接到窗口管理系统服务并将文本或按键事件注入系统。如下展示了在 Android 2.3 版本中 *input* 命令是如何运作的：

```
# input
Usage:  input [text|keyevent]
        input text <string>
        input keyevent <event_code>
```

如下展示了在 Android 4.2 版本中是如何运作的：

```
root@android:/ # input
Usage:input ...
    input text <string>
    input keyevent <key code number or name>
    input [touchscreen|touchpad] tap <x> <y>
    input [touchscreen|touchpad] swipe <x1> <y1> <x2> <y2>
    input trackball press
    input trackball roll <dx> <dy>
```

*input* 的功能很简单，但又不是这样的。例如，对于接收事件的什么信息都知道，只有这样，事件才能被发送到目前有焦点的任何接收者上面。因此，由你来确保所有需要得到输入都有焦点。显然，当你不在电脑前而又正在试图执行这样的动作是一件很困难的事。然而，*input* 给你一个工具可以从命令行提供原始输入。而且，在某些情况下，你发送的输入的意义不需要关注。这里给出了如何从命令行中单击主页按钮，例如：

```
# input keyevent 3
```

你可能会想我是怎么知道 3 是主页键。在 Android 2.3 版本的源文件中，可以查看一下 *frameworks/base/core/java/android/view/KeyEvent.java* 文件和 *frameworks/base/native/include/android/keycodes.h* 文件或者是 Android 4.2 版本中的 *frameworks/native/include/android/keycodes.h* 文件以获取被 android 识别的所有按键代码列表。例如，对于前者，包含代码如下：

```
...
    public static final int KEYCODE_HOME = 3;
    /** Key code constant: Back key. */
    public static final int KEYCODE_BACK = 4;
    /** Key code constant: Call key. */
    public static final int KEYCODE_CALL = 5;
    /** Key code constant: End Call key. */
    public static final int KEYCODE_ENDCALL = 6;
    /** Key code constant: '0' key. */
    public static final int KEYCODE_0 = 7;
...

```

像所有其他的命令一样，*input* 命令也是依赖 *app\_process* 的一个脚本。

## monkey

还有另一个工具允许你为 Android 提供输入。这就是所谓的 *monkey*，在名为“UI/Application Exerciser Monkey”的应用程序的开发文档中有一整段关于它的描述。正如文档所描述的，*monkey* 命令可以用于提供随机而又可重复的输入给你的应用程序。例如，这个命令将 50 个伪随机输入到浏览器的应用程序，可通过如下命令实现：

```
# monkey -p com.android.browser -v 50
```

*monkey* 命令可以实现更多功能。正如你在 Android 2.3 版本（4.2 版本中非常相似）能够看到的输出如下：

```
# monkey
usage: monkey [-p ALLOWED_PACKAGE [-p ALLOWED_PACKAGE] ...]
               [-c MAIN_CATEGORY [-c MAIN_CATEGORY] ...]
               [--ignore-crashes] [--ignore-timeouts]
               [--ignore-security-exceptions]
               [--monitor-native-crashes] [--ignore-native-crashes]
               [--kill-process-after-error] [--hprof]
               [--pct-touch PERCENT] [--pct-motion PERCENT]
               [--pct-trackball PERCENT] [--pct-syskeys PERCENT]
               [--pct-nav PERCENT] [--pct-majornav PERCENT]
               [--pct-appswitch PERCENT] [--pct-flip PERCENT]
               [--pct-anyevent PERCENT]
               [--pkg-blacklist-file PACKAGE_BLACKLIST_FILE]
               [--pkg-whitelist-file PACKAGE_WHITELIST_FILE]
               [--wait-dbg] [--dbg-no-events]
               [--setup scriptfile] [-f scriptfile [-f scriptfile] ...]
               [--port port]
               [-s SEED] [-v [-v] ...]
               [--throttle MILLISEC] [--randomize-throttle]
               [--profile-wait MILLISEC]
               [--device-sleep-time MILLISEC]
               [--randomize-script]
               [--script-log]
               [--bugreport]
               COUNT
```

最有趣的是，你可以为 *monkey* 提供一个脚本运行一组预定义的输入而不是提供随机输入。这是用于开发、测试、现场诊断的一个非常有用的功能。不幸的是，几乎没有任何文档描述 *monkey* 命令这一非常强大的功能。因此，作为参考，这里给出了一个脚本文件示例：

```
# This is a sample test script
# Lines starting with '#' are comments

# This part is the "header"
# monkey doesn't actually look for 'type', but does require 'count', 'speed' and
# 'start data >>'
type= custom
count= 100
speed= 1.0
start data >>

# These are the actual instructions to carry out
LaunchActivity(com.android.contacts,com.android.contacts.TwelveKeyDialer)
# Use this instead in 4.2/Jelly Bean (line-wrap is for book, remove to run)
# LaunchActivity(com.android.contacts,com.android.contacts.activities.Dialtact
#   sActivity)
UserWait(2500)
DispatchPress(KEYCODE_1)
```

```
UserWait(200)
DispatchPress(KEYCODE_8)
UserWait(200)
DispatchPress(KEYCODE_0)
UserWait(200)
DispatchPress(KEYCODE_0)
UserWait(200)
DispatchPress(KEYCODE_8)
UserWait(200)
DispatchPress(KEYCODE_8)
UserWait(200)
DispatchPress(KEYCODE_9)
UserWait(200)
DispatchPress(KEYCODE_8)
UserWait(200)
DispatchPress(KEYCODE_9)
UserWait(200)
DispatchPress(KEYCODE_6)
UserWait(200)
DispatchPress(KEYCODE_9)
UserWait(200)
DispatchPress(KEYCODE_ENTER)
UserWait(10000)
DispatchPress(KEYCODE_ENDCALL)
UserWait(200)
RunCmd(input keyevent 3)
UserWait(1000)
RunCmd(service call statusbar 1)
UserWait(2000)
RunCmd(service call statusbar 2)
```

使用如下的命令行来运行该脚本：

```
# monkey -f myscript 1
```

这个脚本将启动标准拨号，拨号 1-800-889-8969 等待 10 秒<sup>注 3</sup>，挂起，返回到主屏幕，然后展开和折叠状态栏。请注意，最后一部分采用 RunCmd 指令使脚本直接从命令行运行命令。顺便说一句，这些都是我们之前看到的命令。当然，这是一个相当短并且简单的脚本。你甚至可以将多个这样的脚本的调用整合为更复杂的 shell 脚本。

关于用作脚本语言的 *monkey* 命令更为详细的了解以及每个命令可以携带的参数，我请你看一看 *monkey* 的脚本解释代码，在文件 *development/cmds/monkey/src/com/android/commands/monkey/MonkeySourceScript.java* 中并查看 *EVENT\_KEYWORD\_*。你会发现事件的关键词，如 DispatchPress, UserWait 以及许多其他的关键词。

为了实现其功能与魔力，*monkey* 命令与活动管理器、窗口管理器、包管理器通信。它也是一个依赖于 *app\_process* 来启动 Java 代码实现其功能的 shell 脚本。

---

注 3： 如果你想知道的话，这是出版者的手机号。

---

**警告：**如果你看看 *development/cmds/monkey/* 目录下的源文件，你会找到一个文件名为 *example\_script.txt* 的文件，其中似乎包含了一些脚本指令。目前还不清楚为什么这个文件在源文件目录中，因为该文件中的语义没有对应到 *monkey* 工具预期的实际语义。

---

## bmgr

从 Android 2.2/ 冻酸奶版本之后，Android 拥有了备份功能，允许用户将他们拥有的数据备份到云中，以便以后如果丢失或更换它们的设备后，还能对数据进行还原。谷歌本身通过扮演一种可能的传输方式<sup>注4</sup>，也提供了这样的功能，但其他的可以提供交替的传输方式。Android 内部提供的 API 及应用程序开发者的 API 是独立于传输的。然而，它仍然是一个针对 Android 手机和平板电脑使用的非常具体的以及可能不需要在嵌入式环境中的使用的行为<sup>注5</sup>：

```
# bmgr
usage:bmgr [backup|restore|list|transport|run]
        bmgr backup PACKAGE
        bmgr enable BOOL
        bmgr enabled
        bmgr list transports
        bmgr list sets
        bmgr transport WHICH
        bmgr restore TOKEN
        bmgr restore PACKAGE
        bmgr run
        bmgr wipe PACKAGE
```

The 'backup' command schedules a backup pass for the named package.  
Note that the backup pass will effectively be a no-op if the package  
does not actually have changed data to store.

The 'enable' command enables or disables the entire backup mechanism.  
If the argument is 'true' it will be enabled, otherwise it will be  
disabled. When disabled, neither backup or restore operations will  
be performed.

The 'enabled' command reports the current enabled/disabled state of  
the backup mechanism.

The 'list transports' command reports the names of the backup transports  
currently available on the device. These names can be passed as arguments  
to the 'transport' command. The currently selected transport is indicated  
with a '\*' character.

The 'list sets' command reports the token and name of each restore set

---

注 4：这里的“传输”是与给定云服务接口的一个所需的引擎。

注 5：对应 Android 2.3 的输出，Android 4.2 类似。

available to the device via the current transport.

The 'transport' command designates the named transport as the currently active one. This setting is persistent across reboots.

The 'restore' command when given a restore token initiates a full-system restore operation from the currently active transport. It will deliver the restore set designated by the TOKEN argument to each application that had contributed data to that restore set.

The 'restore' command when given a package name initiates a restore of just that one package according to the restore set selection algorithm used by the `RestoreSession.restorePackage()` method.

The 'run' command causes any scheduled backup operation to be initiated immediately, without the usual waiting period for batching together data changes.

The 'wipe' command causes all backed-up data for the given package to be erased from the current transport's storage. The next backup operation that the given application performs will rewrite its entire data set.

如果这与你使用 Android 相关，看看应用程序开发者手册中关于数据备份章节以及由谷歌提供的自己备份传输的有关信息。像我们看到其他的许多命令一样，`app_process` 用于启动实际的与备份管理器服务接口对应的 Java 代码。

## stagefright

Android 的一个关键特征是其丰富的介质层，AOSP 包含的工具使你能与它进行交互。更具体地说，`stagefright` 命令能与多媒体播放器服务交互，允许你做多媒体播放。下面是它在 Android 2.3 中的在线帮助文档（在 4.2 版本中稍有扩展）：

```
# stagefright -h
usage:stagefright
  -h(help)
  -a(audio)
  -n repetitions
  -l(list) components
  -m max-number-of-frames-to-decode in each pass
  -b bug to reproduce
  -p(rofiles) dump decoder profiles supported
  -t(humbnail) extract video thumbnail or album art
  -s(oftware) prefer software codec
  -o playback audio
  -w(rite) filename (write to .mp4 file)
  -k seek test
```

下面例举了你是如何播放一个 .mp3 文件的：

```
# stagefright -a -o /sdcard/trainwhistle.mp3
```

你可能还需要使用 *record* 和 *audioloop* 工具，它们在 Android 2.3 版本中的目录 *frameworks/base/cmds/stagefright/* 或者 Android 4.2 版本中的目录 *frameworks/av/cmds/stagefright/* 中的 *stagefright* 的源文件中。虽然少量的关于它们使用的几个例子可以在网上或其他地方找到，但它们的参考资料严重缺乏是一个事实。有趣的是，所有这三个工具都是用 C 编码实现，不像我们目前看到的大多数的主要用 Java 和脚本中使用 *app\_process* 编码的系统特定服务工具。同时 *stagefright* 命令与媒体服务直接通信，而 *record* 和 *audioloop* 命令使用 OMXClient 方便地与相同的服务通信。

## Dalvik 工具

我们已经看到了如何通过 *am* 命令来发送事件，因而触发新的应用程序启动，这些应用程序中的每一个都有自己的 Zygote-forked Dalvik 实例。我们也看到 *app\_process* 命令是如何使用 Android 的运行时刻 (Runtime) 来启动 Java 编码的命令行工具的。然而，某些情况下，你可能想放弃所有的 Android 特定的层，并直接参与到 Dalvik 中。以下是可以允许你这么做的命令<sup>注6</sup>。

```
# dalvikvm -help

dalvikvm: [options] class [argument ...]
dalvikvm: [options] -jar file.jar [argument ...]

The following standard options are recognized:
  -classpath classpath
  -Dproperty=value
  -verbose:tag ('gc', 'jni', or 'class')
  -ea[:<package name>... |:<class name>]
  -da[:<package name>... |:<class name>]
    (-enableassertions, -disableassertions)
  -esa
  -dsa
    (-enablesystemassertions, -disablesystemassertions)
  -showversion
  -help

The following extended options are recognized:
  -Xrunjdwp:<options>
  -Xbootclasspath:bootclasspath
  -Xcheck:tag (e.g. 'jni')
  -XmsN (min heap, must be multiple of 1K, >= 1MB)
  -XmxN (max heap, must be multiple of 1K, >= 2MB)
  -XssN (stack size, >= 1KB, <= 256KB)
  -Xverify:{none,remote,all}
  -Xrs
  -Xint (extended to accept ':portable', ':fast' and ':jit')
```

---

注 6：这是 Android 2.3/ 姜饼的输出，Android 4.2/ 果冻豆的输出也类似。

These are unique to Dalvik:

```
-Xzygote  
-Xdexopt:{none,verified,all}  
-Xnoquithandler  
-Xjnígreflimit:N (must be multiple of 100, >= 200)  
-Xjniopts:{warnonly,forcecopy}  
-Xjnítrace:substring (eg NativeClass or nativeMethod)  
-Xdeadlockpredict:{off,warn,err,abort}  
-Xstacktracefile:<filename>  
  
-Xgc:[no]precise  
-Xgc:[no]preverify  
-Xgc:[no]postverify  
-Xgc:[no]concurrent  
-Xgc:[no]verifycardtable  
-Xgenregmap  
-Xcheckdexsum  
-Xincludeselectedop  
-Xjitop:hexopvalue[-endvalue][,hexopvalue[-endvalue]]*  
-Xincludeselectedmethod  
-Xjitthreshold:decimalvalue  
-Xjitblocking  
-Xjitmap:signature[,signature]* (eg Ljava/lang/String\;replace)  
-Xjitcheckcg  
-Xjitverbose  
-Xjitprofile  
-Xjitedisableopt
```

Configured with: debugger profiler hprof jit(armv5te) show\_exception=1

Dalvik VM init failed (check log file)

*dalvikvm* 实际上是一个原始的没有连接到任何“Android”上的 Dalvik 虚拟机。它不依赖于 Zygote，同时它也不包含 Android 的运行时库。它只是启动一个虚拟机来运行你提供的所有类或 JAR 文件。实际上 AOSP 本身并不经常使用它，可能是因为在 AOSP 中没有太多不运行在“Android”上下文中的情况。例如，在 Android 2.3/ 姜饼版本中的预装载 Java 库，在文件 *frameworks/base/tools/preload/MemoryUsage.java* 中使用它，连同 adb 工具来检查目标设备中的一个类的内存使用量。

## dvz

另一种启动 Dalvik 虚拟机的方法是使用 *dvz* 命令：

```
# dvz --help  
usage: dvz [--help] [-classpath <classpath>]  
[additional zygote args] fully.qualified.java.ClassName [args]
```

请求一个新的从 Zygote 进程中生成的 Dalvik 虚拟机实例。*stdin*、*stdout* 和 *stderr* 都被与之关联。在产生的虚拟机实例保持运行的时间内，这个进程会被保留，并提交一些信号。VM 实例的退出代码被丢弃。

正如描述所暗示的，*dvz* 命令实际上的作用类似活动管理器，它通过请求 Zygote 进程调用 *fork* 产生一个新的进程。在这里，唯一不同的是由此产生的进程不受活动管理器的管理。相反，它是非常独立的。

这是否意味着该工具被大量使用还不清楚，因为在 Android 2.3 版本中，它唯一的使用实例是在测试代码中，特别是在 *dalvik/tests/etc/push-and-run-test-jar* 中，在 Android 4.2 版本中，它甚至没有包含在默认的构建中。然而不管怎样，在你的库里面的实例中使用该命令有可能非常有用。

## 启动 Dalvik 的多种方法

到目前为止，我们已经了解了四种启动 Dalvik 的方法。值得我们花点时间正确地把他们统一的整理一下。表 7-1 描述启动 Dalvik 虚拟机的每个方式、在虚拟机里包含了什么及它是如何启动的。

表 7-1：Dalvik 启动的方式

命令	Dalvik VM	Android Runtime	Zygote	Activity Manager	机制
<i>dalvikvm</i>	X				使用 <i>libdvm.so</i>
<i>app_process</i>	X	X			使用 <i>libandroid_runtime.so</i>
<i>DVZ</i>	X	X	X		使用 <i>libcutils<sup>a</sup></i>
<i>am</i>	X	X	X	X	Talks to Activity Manager service

a. *asystem/core/libcutilc/zygote.c* 与活动管理器服务（Activity Manager Service）对话文件中包含了 *zygote\_run\_wait()* 函数和一个 *zygote\_run\_oneshot()* 函数。

*am* 是唯一的一个为我们提供由活动管理器控制的 Dalvik 虚拟机实例的命令。在所有其他情况下，虚拟机是独立的，并没有它的生命周期管理。*am* 命令也是唯一的一个使我们能够自动触发，*apk* 文件中代码执行的命令。而所有其他的命令需要我们提供了一个特定的类或 JAR 文件。

## dexdump

如果你想要反向开发 Android 应用程序或者 JAR 文件，你可以通过 *dexdump* 命令实现：

```
#dexdump  
dexdump: no file specified  
Copyright (C) 2007 The Android Open Source Project  
  
dexdump: [-c] [-d] [-f] [-h] [-i] [-l layout] [-m] [-t tempfile] dexfile...
```

```
-c : verify checksum and exit  
-d : disassemble code sections  
-f : display summary information from file header  
-h : display file header details  
-i : ignore checksum failures  
-l : output layout, either 'plain' or 'xml'  
-m : dump register maps (and nothing else)  
-t : temp file name (defaults to /sdcard/dex-temp-*)
```

下面例举了如何在 JAR 文件上使用该命令：

```
# dexdump /system/framework/services.jar  
Processing '/system/framework/services.jar'...  
Opened '/system/framework/services.jar', DEX version '035'  
Class #0 -  
    Class descriptor : 'Lcom/android/server/AccessibilityManagerService$1;'  
    Access flags : 0x0000 ()  
    Superclass : 'Landroid/os/Handler;'  
    Interfaces -  
    Static fields -  
    Instance fields -  
        #0 : (in Lcom/android/server/AccessibilityManagerService$1;)  
            name : 'this$0'  
            type : 'Lcom/android/server/AccessibilityManagerService;'  
            access : 0x1010 (FINAL SYNTHETIC)  
    Direct methods -  
        #0 : (in Lcom/android/server/AccessibilityManagerService$1;)  
            name : '<init>'  
            type : '(Lcom/android/server/AccessibilityManagerService;)V'  
            access : 0x10000 (CONSTRUCTOR)  
            code -  
            registers : 2  
            ins : 2  
            outs : 1  
            insns size : 6 16-bit code units  
            catches : (none)  
            positions :  
                0x0000 line=113  
            locals :  
                0x0000 - 0x0006 reg=0 this Lcom/android/server/AccessibilityManagerService$1;  
    Virtual methods -  
        #0 : (in Lcom/android/server/AccessibilityManagerService$1;)  
            name : 'handleMessage'  
...
```

同时你也可以让它反汇编代码：

```
# dexdump -d /system/app/Launcher2.apk  
...  
00ea5c:                                     [[00ea5c] com.android.common.ARRAY  
yListCursor.<init>:([Ljava/lang/String;Ljava/util/ArrayList;)V  
00ea6c: 1206                                |0000: const/4 v6, #int 0 // #0  
00ea6e: 1a07 e804                            |0001: const-string v7, "_id" //
```

```
string@04e8
00ea72: 7010 b400 0800 |0003: invoke-direct {v8}, Landro
id/database/AbstractCursor;.<init>:()V // method@00b4
00ea78: 2190 |0006: array-length v0, v9
00ea7a: 1201 |0007: const/4 v1, #int 0 // #
00ea7c: 1202 |0008: const/4 v2, #int 0 // #
00ea7e: 3502 0f00 |0009: if-ge v2, v0, 0018 // +000
00ea82: 4604 0902 |000b: aget-object v4, v9, v2
00ea86: 1a05 e804 |000d: const-string v5, "_id" //
string@04e8
00ea8a: 6e20 dd07 7400 |000f: invoke-virtual {v4, v7}, L
java/lang/String;.compareToIgnoreCase:(Ljava/lang/String;)I // method@07dd
00ea90: 0a04 |0012: move-result v4
00ea92: 3904 3e00 |0013: if-nez v4, 0051 // +003e
00ea96: 5b89 3600 |0015: igure-object v9, v8, Lcom/a
ndroid/common/ArrayListCursor;.mColumnNames:[Ljava/lang/String; // field@0036
00ea9a: 1211 |0017: const/4 v1, #int 1 // #
00ea9c: 3901 1400 |0018: if-nez v1, 002c // +0014
00eaa0: d804 0001 |001a: add-int/lit8 v4, v0, #int
1 // #01
00eaa4: 2344 d901 |001c: array-array v4, v4, [Ljava/l
ang/String; // class@01d9
00eaa8: 5b84 3600 |001e: input-object v4, v8, Lcom/a
ndroid/common/ArrayListCursor;.mColumnNames:[Ljava/lang/String; // field@0036
00eaac: 5484 3600 |0020: igure-object v4, v8, Lcom/a
ndroid/common/ArrayListCursor;.mColumnNames:[Ljava/lang/String; // field@0036
...
...
```

显然关于 Android 反汇编代码的主题远远超出本书所描述的范围，但是如果你对这个话题很感兴趣，我建议你在你最喜欢的在线书店看一看专门讲 Android 的安全和取证方面的书籍。

## 支持守护进程

大量的 Android 的智能之处是基于系统服务实现的，有的情况下，一个系统服务作为本地守护进程的中介，但实际上却完成了要求的关键操作。这种方法不同于由一个系统的服务器直接地进行实际操作的指导，它受青睐可能有两个主要原因：安全性和可靠性。

正如我在第 1 章中阐述，Android 权限模型要求需要调用特定操作的应用程序开发人员在创建期间就请求特定的权限。典型的，在一个应用程序的 manifest 文件中这些权限将类似于下面这样的事情：

```
...
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
...
...
```

在这种情况下，这些权限需要有打开套接字并抓住 wakelocks 的能力。显然有很多比这更多的权限。看一下应用开发文档提供的可用权限的完整列表。没有这些权限，一个应用程序无法进行一些最为关键的 Android 操作。主要原因是应用程序运行在非特权用户模式下，不能调用任何需要 root 权限的系统调用或访问 /dev 目录下最关键的设备。应用程序必须请求系统服务代理，反过来，系统服务在执行它们得到的任何请求之前检查应用权限。

然而，系统服务本身不是以 root 身份运行。*system\_server* 进程以 system 身份运行；媒体服务器进程以 media 身份运行；手机应用程序以 radio 身份运行。如果你检查 /dev，你会看到一些条目只属于这样一些用户。你也会发现相当多的条目属于 root 用户。因此，就像应用程序，系统服务一般不会使用需要 root 权限的系统调用或者访问 /dev 的关键设备。

相反，许多关键的操作需要系统服务借助于目录 /dev/socket/ 下的 UNIX 域套接字与以 root 身份运行或是特定用户进行特权操作的本地守护进程进行通信来完成。许多这些守护进程是 Android 特定的，尽管其中一些我们早已在第 6 章中作为传统的 Linux 守护进程进行阐述，如在 Android 4.2 之前版本中的 *bluetoothd*。

在某些特定情况下，例如 *rild*，需要关注与基带处理器间的通信，看来选择作为一个单独的进程运行可能在可靠性方面还有很多要做的。事实上，智能手机的电话功能是非常关键的，这就使得你要确保其操作独立于任何潜在的、驻扎于 *system\_server* 进程中可能影响系统服务的问题。

让我们看看系统服务所使用的主要支持的守护进程，以及它们的配置及相关的命令行工具。请注意，我们不会重述之前已经讲解的守护进程，如 Zygote；或者那些与系统服务不相关的，如 *ueventd* 和 *dumpsyst*；或者不是 Android 特定的，如 *bluetoothd* 或 *wpa\_supplicant*。

## installd

包管理服务的工作就是管理 .apk 文件，它不需要适当的特权来执行许多操作以及 / 或者启动一个应用程序使其运行的操作。相反，对于关键的文件系统操作和命令，它依赖于 *installd*，其在 Android 2.3 版本中以 root 身份运行，在 Android 4.2 版本中以 *install* 用户运行。例如，在一个 .apk 的文件中运行 *dexopt*，为 Dalvik 生成 JIT-optimized .dex 文件是由 *installd* 命令在代表软件包管理器的安装时间里完成的。

在 Android 2.3 中，*installd* 由 *init.rc* 文件中以下这部分来启动（Android 4.2 版本做的事情相当类似）：

```
service installd /system/bin/installld
    socket installd stream 600 system system
```

然后打开 `/dev/socket/installld` 设备监听连接，接着再监听来自包管理器的命令。它没有配置文件，同时它也不带任何命令行参数。没有任何命令行工具独立于软件包管理器之外与其通信。因此，从命令行激活 `installld` 的唯一途径是使用 `pm` 命令，它将与包管理器进行通信，这将，接着实现与 `installld` 的通信，如果需要的话。

`installld` 的源代码在目录 `frameworks/base/cmds/installld/` 下，你可能需要看看 `install.c` 和 `commands.c` 文件。前者包含被 `installld` 确认的命令列表，后者包含这些命令的具体实现。以下是 Android 2.3 版本的 `install.c` 文件，其中列出了被 `installld` 确认的命令（4.2 版本的命令比这个稍多一点）：

```
struct cmdinfo cmds[] = {
    { "ping",                      0, do_ping },
    { "install",                    4, do_install },
    { "dexopt",                     3, do_dexopt },
    { "movedex",                   2, do_move_dex },
    { "rmdex",                      1, do_rm_dex },
    { "remove",                     2, do_remove },
    { "rename",                     3, do_rename },
    { "freecache",                  1, do_free_cache },
    { "rmcache",                    2, do_rm_cache },
    { "protect",                    2, do_protect },
    { "getsize",                    4, do_get_size },
    { "rmuserdata",                 2, do_rm_user_data },
    { "movefiles",                  0, do_movefiles },
    { "linklib",                     2, do_linklib },
    { "unlinklib",                  1, do_unlinklib },
};

};
```

注意，就像下面我们会看到的许多其他守护进程一样，`installld` 与软件包管理器之间的有线协议是基于字符串的。因此，上面的代码片段每个命令包含三项：命令的字符串被“有线地”发送，预期的参数数目，以及当命令被接收时调用 `install.c` 中的功能。

## vold

`vold` 注重很多的挂载服务所需的关键操作，如挂载和格式化卷。不同于 `installld` 命令，`vold` 命令在 Android 2.3 和 4.2 版本中都是以 `root` 身份运行，而挂载服务只是系统服务的一部分。在 Android 2.3 中 `vold` 通过 `init.rc` 的以下代码段启动（4.2 版本中的代码段也类似）：

```
service vold /system/bin/vold
    socket vold stream 0660 root mount
        ioprio be 2
```

不同于文中剩余的待阐述的支持守护进程，*vold* 实际上有一个配置文件 */etc/vold.fstab*。下面是摘自描述文件语义的目录 *system/core/rootdir/etc/* 下的 *vold.fstab* 文件中的一段：

```
#####
## Regular device mount
##
## Format: dev_mount <label> <mount_point> <part> <sysfs_path1...>
## label           - Label for the volume
## mount_point    - Where the volume will be mounted
## part            - Partition # (1 based), or 'auto' for first usable partition.
## <sysfs_path>   - List of sysfs paths to source devices
#####
```

下面这一段是与模拟器中的 SD 卡相关的一段代码：

```
dev_mount sdcard /mnt/sdcard auto /devices/platform/goldfish_mmc.0 /devices/plat
form/msm_sdcc.2/mmc_host/mmc1
```

当 *vold* 命令开始后，它将解析这个文件，然后打开 */dev/socket/vold* 来侦听连接和命令。不同于 *installd* 命令，有一个命令行工具可以与 *vold* 直接进行通信：

```
usage: vdc <monitor>|<cmd> [arg1] [arg2...]
```

在命令行上 *vdc* 命令期待的实际参数与当 *vold* 命令连接到指定的套接字时期待从挂载服务获取的参数是相同的。不幸的是，没有文档或在线帮助完整地描述了命令设置。相反地，你必须看看目录 *system/vold/* 下的 *CommandListener.cpp* 文件了解 *vold* 命令设置的实现。

例如，你可以像下面这样查看 *vold* 的内部状态：

```
# vdc dump
000 Dumping loop status
000 Dumping DM status
000 Dumping mounted filesystems
000 rootfs / rootfs rw 0 0
000 tmpfs /dev tmpfs rw,mode=755 0 0
000 devpts /dev/pts devpts rw,mode=600 0 0
000 proc /proc proc rw 0 0
000 sysfs /sys sysfs rw 0 0
000 none /acct cgroup rw,cpuacct 0 0
000 tmpfs /mnt/asec tmpfs rw,mode=755,gid=1000 0 0
000 tmpfs /mnt/obb tmpfs rw,mode=755,gid=1000 0 0
000 none /dev/cpuctl cgroup rw,cpu 0 0
000 /dev/block/mtdblock0 /system yaffs2 ro 0 0
000 /dev/block/mtdblock1 /data yaffs2 rw,nosuid,nodev 0 0
000 /dev/block/mtdblock2 /cache yaffs2 rw,nosuid,nodev 0 0
200 dump complete
```

在某些情况下，*vdc* 实际上是提供在线帮助的：

```
# vdc volume format  
500 Usage: volume format <path>
```

为了在 Android 4.2 版本中为你的设备自定义一个存储设备列表，你需要看看文件 *frameworks/base/core/res/res/xml/storage\_list.xml*。你可能想在您设备的目录 *device/acme/coyotepad/overlay/* 下创建那个文件的一个覆盖版本以实现对你设备的定制。

## netd

网络管理服务中关键的网络配置操作依赖于 *netd* 命令，如配置网络接口、建立限度、运行 *pppd*。在这种情况下，*netd* 以 root 身份运行，而网络管理服务只是系统服务的一部分。在 Android 2.3 版本，*netd* 由 *init.rc* 中的以下部分启动：

```
service netd /system/bin/netd  
    socket netd stream 0660 root system
```

然而在 Android 4.2 版本中，声明发生了变化：

```
service netd /system/bin/netd  
    class main  
    socket netd stream 0660 root system  
    socket dnsproxyd stream 0660 root inet  
    socket mdns stream 0660 root system
```

*netd* 命令打开 */dev/socket/netd* 并监听连接和命令。它不接收任何命令行参数，也不依赖于任何配置文件。与 *vold* 一样，它有一个命令行工具直接与它通信。下面例举了在 Android 2.3 版本中该命令的在线帮助文档：

```
# ndc  
Usage: ndc <monitor>|<cmd> [arg1] [arg2...]
```

在 Android 4.2 版本中该命令的在线帮助文档：

```
root@android:/ # ndc  
Usage: ndc [sockname] <monitor>|<cmd> [arg1] [arg2...]
```

像 *vdc* 一样，*ndc* 命令期待的命令行参数与 *netd* 在它的套接字中期待的参数相同。同 *vold* 类似，你需要看看在目录 *netd/system/netd/* 下的 *CommandListener.cpp* 文件来了解它的指令语义。

同 *vdc* 类似，你可以通过 *ndc* 命令请求 *netd* 的状态信息：

```
# ndc interface list  
110 lo
```

```
110 eth0
110 tun0
110 gre0
200 Interface list completed
```

## vold 及 netd 命令的设置

*vold* 及 *netd* 命令都是使用 *libsy sutils* 提供的相同的 C++ 机制构建，都依赖于文件 *CommandListener.cpp* 解析和调度发送给它们的命令。通过查看文件 *CommandListener.cpp* 中的构造函数，了解这两个相互被接受的特定命令：

```
CommandListener::CommandListener() :
    FrameworkListener("...") {
    ...
}
```

每一个都包含对函数 *registerCmd()* 的调用，其注册的对象定义在下面的同一个文件中。下面是来自 Android 2.3 版本中 *vold* 指令对 *dump* 的摘录：

```
CommandListener::CommandListener() :
    FrameworkListener("vold") {
    registerCmd(new DumpCmd());
    registerCmd(new VolumeCmd());
    ...
CommandListener::DumpCmd::DumpCmd() :
    VoldCommand("dump") {
}

int CommandListener::DumpCmd::runCommand(SocketClient *cli,
                                         int argc, char **argv) {
    cli->sendMsg(0, "Dumping loop status", false);
    if (Loop::dumpState(cli)) {
        cli->sendMsg(ResponseCode::CommandOkay, "Loop dump failed", true);
    }
    ...
}
```

被 *vold* 或 *netd* 接受的每个命令具有相应的 *runCommand()* 函数来解析传递到该命令的参数。正如我们之前做过的，通过在命令行中运行 *vdc dump*，我们在上面代码中调用 *runCommand()* 函数。相反，在命令行中输入 *vdc volume list* 将调用下面的函数，并传递 *list* 作为参数的一部分：

```
int CommandListener::VolumeCmd::runCommand(SocketClient *cli,
                                             int argc, char **argv) {
    ...
}
```

## rild

托管在手机应用程序中的电话系统服务，使用 *rild* 命令与基带处理器进行通信。*rild* 命令本身使用 *dlopen()* 函数加载一个 *baseband-specific .so* 文件到实际的基带硬件接

口。正如我之前提到的，*rild* 的存在可能就是为了确保即使剩余堆栈发生问题，手机端的系统仍然保持活跃。

在 Android 2.3 版本中，使用模拟器的情况下，*rild* 命令由 *init.rc* 文件的以下部分启动（4.2 版本的几乎相同）：

```
service ril-daemon /system/bin/rild
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc audio sdcard_rw
```

*rild* 命令没有配置文件，但它本身可以携带几个命令行参数。

```
usage: rild -l <ril impl library> [-- <args for impl library>]
```

如果命令行中没有提供 RIL 实现库，*rild* 将尝试使用 *rild.libpath* 全局属性来定位库。如果不指定，它会认为系统回路中没有无线电信号，它会调用 *sleep()* 函数。如果是在模拟器中，系统依赖于 */system/lib/libreference-ril.so* 文件，正如它的名字一样意味着一个实现真正的 RIL 链接库的操作的参考实现。

有两个 UNIX 域套接字被 *rild* 使用：用于电话系统服务的 */dev/socket/rild*、用于 *radiooptions* 命令进行交互的 */dev/socket/rild-debug*。事实上，后者是一个与 *rild* 进行通信的命令行工具：

```
usage: radiooptions [option] [extra_socket_args]
    0 - RADIO_RESET,
    1 - RADIO_OFF,
    2 - UNSOL_NETWORK_STATE_CHANGE,
    3 - QXDM_ENABLE,
    4 - QXDM_DISABLE,
    5 - RADIO_ON,
    6 apn- SETUP_PDP apn,
    7 - DEACTIVE_PDP,
    8 number - DIAL_CALL number,
    9 - ANSWER_CALL,
   10 - END_CALL
```

如果想知道更多关于 *rild* 和 *radiooptions* 命令的相关知识，参看 *hardware/ril/rild* 目录下的源码。RIL 的参考实现位于 *hardare/ril/referenceril/* 目录下。

## keystore

与到目前为止我已经给出了的其余的守护进程不太一样，*keystore* 实际上并没有服务于任何的系统服务。相反，它被各种不同的系统使用，用于键值对的存储和检索。

它保存的值主要是用于连接到网络或网络等基础设施的安全密钥，例如访问点及 VPNs，安全密钥意味着是一个用户定义的密码。显然，采用一个单独的守护进程来进行这个信息的存储增加了系统的整体安全性。

在 Android 2.3 版本中，*keystore* 是由 *init.rc* 文件的以下部分启动（4.2 版本中大致相同）：

```
service keystore /system/bin/keystore /data/misc/keystore
    user keystore
    group keystore
    socket keystore stream 666
```

*keystore* 命令没有配置文件，但它确实会期待一个目录来存储每个密钥对的值。正如你在前面看到的，这个目录是 */data/misc/keystore*。然后 *keystore* 在 */dev/socket/keystore* 监听连接和命令。几个本地守护进程连接到 *keystore* 来检索键值，如 *wpa\_supplicant*、*mtpd* 及 *racoon*。而“设置”这一应用程序也可以连接到 *keystore*，列出并插入新的键值。

还有一个与 *keystore* 通信的命令行工具：

```
usage: keystore_cli action [parameter ...]
```

在 Android 2.3 版本的目录 *frameworks/basecmds/keystore/* 下以及 4.2 版本的目录 *system/security/keystore/* 下你将找到命令 *keystore* 和 *keystore\_cli* 的源代码。

## 其他支持守护进程

还有一些额外的守护进程扮演了一个次要角色，在这里，我们无法一一讲述，如 *mtpd* 和 *racoon*。前者用于 VPN，能在目录 *external/mtpd/* 下找到，后者用于 IPSec，能在目录 *external/ipsec-tools/* 下找到。

当然，有可能在你的系统上运行的其他守护进程有特定的用途，和 / 或你可能要添加你自己的自定义守护进程。看看在第 4 章关于如何在你的 AOSP 构建系统中添加你自定义库的解释。记住，如果你想要一个守护进程在 *init* 启动时开始启动，你需要在主 *init.rc* 文件或是特定板级初始化文件 *init.<device\_name>.rc* 中为其添加一个 *service* 声明。

## 硬件抽象层

正如我在第 2 章中所述，Android 依赖于硬件抽象层（HAL）来跟硬件进行交互。实际上，系统服务几乎不直接跟设备通过 */dev* 目录项进行交互，而是通过 HAL 模块，常规的共享库来跟硬件进行通信，如表 2-1 中所列。

Android 的 HAL 实现可以在 `hardware/` 目录中找到。最重要的是，你可以在 `hardware/libhardware/include/hardware/` 和 `hardware/libhardware_legacy/include/hardware_legacy` 目录的头文件中找到系统框架与 HAL 模块之间的接口定义。这里的头文件提供了每一种 Android 所支持硬件的 API。你还可以看到这些 HAL 模块一些示例实现的源代码，在 `device/` 目录下。

理想情况下，你需要避免为现有的系统服务实现你自己的 HAL 模块。事实上，你应该向你的 SoC 或者开发板提供商请求这样的模块。编写 HAL 模块要求对相应系统服务的内部复杂机制有很清晰的理解，学会正确地编写这样的代码是一个很耗时的过程，特别是 HAL 接口需要根据每个新的 Android 版本去改进。所以，我强烈建议你使用厂商或者 SOC 供应商已经完成 HAL 开发的组件 / 开发板。

通常来说，考虑到 Android 所取得的市场成功，部件及 SoC 供应商多数已经为 Android 在它们产品上正常运行付出了很多努力。也就是说，它们要么可以针对它们的评估板向你提供完整可用的 AOSP 以及支持 Android 的内核，或者针对它们的模块提供 HAL 模块及 Linux 驱动程序。所以，针对你的硬件类型实现你自己的 HAL 模块可能仅仅是最后一种备选方案。事实上，你应该尽快找你的 SoC 或者部件供应商拿到使你能够运行 Android 的 HAL 模块和驱动程序。

---

**注意：**所有主流的 SoC 供应商针对它们的评估板以这种或那种方式提供可以直接使用的 AOSP 及内核。例如 TI、高通、飞思卡尔、三星以及很多其他厂商都是如此。如果你正在设计的你自己的电路板是基于它们设计中的一个，我建议你获取他们的 AOSP 代码树并且根据你的需求对其进行定制。试图在你的硬件上直接使用 Google 提供的 AOSP 代码树从头移植 Android 不是一个好方法，估计也不符合你的产品上市时间要求。

---

如果你确实需要为现有的系统服务实现一个自己的 HAL 模块，那么建议参考我前面提到的头文件，它们定义了每一种 HAL 模块需要的 API，并且尽量从 `device/` 目录下参考的 HAL 实现中得到更多的灵感。例如，针对 2.3/ 姜饼系统，看一看 `device/samsung` 下的 `lib*` 目录。而对于 4.2/ 果冻豆，建议看看 `device/asus/grouper/` 和 `device/samsung/tuna/`。

# 传统的用户空间

正如我在第 2 章所阐述的，尽管是基于 Linux 内核的，但 Android 不像任何其他 Linux 系统。事实上，从图 2-1 你可以看到，Android 的用户空间，我们在第 6 章和第 7 章中探讨了的，是一种谷歌的自定义的创作。因此，如果你熟悉“传统”的 Linux 系统或有嵌入式 Linux 的背景，你可能会发现自己会回忆起经典的、已使用了很长一段时间的 Linux 工具和组件。本附录将告诉你如何使得传统的 Linux 用户空间与 AOSP 共存于同一个 Linux 内核顶层。

## 基础

首先，我们需要就传统 Linux 用户空间是什么达成一致。在目前的讨论中，我们假定我们谈论的是一个文件系统层次结构标准（FHS）兼容的根文件系统。正如我前面提到的，Android 的根文件系统不是 FHS 兼容的，它没有使用关键的 FHS 目录（如 `/bin` 和 `/lib`），这允许我们添加与之并列的、确实使用这些目录的一个根文件系统。

现在，我不是说你将会使用这些指令来构建同时包含 AOSP 和一个大的如 Ubuntu 一样的发布系统。这里有更多关于 Ubuntu 作为分布系统以及你需要考虑的 AOSP 细节，而不是解析如何去匹配少量顶层的根文件系统目录。然而，如果你熟悉如何为嵌入式 Linux 系统创建一个基本的根文件系统，它将使你对如何把你最喜欢的工具和库，如 Busybox、glibc，与 AOSP 装载在同一根文件系统了解得相对来说比较清晰。如果你有更多的兴趣与计划，例如，使 Ubuntu 或 Fedora 与 AOSP 共存于同一根文件系统，这些解释提供了一个很好的入门介绍。

踏上这条路之前，回答“何必呢”这一问题的做法是值得的。事实上，为什么要花时

间去使得那些传统 Linux 软件包与 AOSP 在同一内核共处呢？既然它已经有了一个 C++ 库、命令行工具、丰富的用户空间等，为什么不直接使用 AOSP？难道 AOSP 无法做所需要做的一切？不？

开发者想要 Android 保留传统 Linux 用户空间的主要原因是能够使得现有 Linux 应用能接入运行 Android 的系统上而不用必须把它们接入 Android 中。例如，如果你有传统代码只能在 glibc 上运行得很好，那么在你的根文件系统中引入 glibc 比你把你的传统代码接入 Bionic 可能更容易。事实上，你可以看到通过阅读 Bionic 在 *bionic/libc/* 目录下自带的文档，尤其是在 *docs/* 目录下的文档，相比更主流的东西，如 glibc，Bionic 有许多的局限性和差异性。例如，它不是 Posix 兼容的，也没有体现 System V 中 IPC 调用。依托著名的 C 库如 glibc，你可以避免这些附带问题。

重用经典的 Linux 系统组件的另一个原因是为了避免处理 Android 的构建系统。正如我们在第 4 章中看到的，Android 的构建系统是非递归的。因此，如果想重用大的、传统的软件包，你通常不得不将其构建系统转换成使用 Android 的构建系统的 *.mk* 文件。事实上，一些非常著名的包在导出到 AOSP 的 *external/* 目录时已重建构建文件以供 AOSP 使用。例如，传统的基于 *autoconf* 和 *automake* 的 D-BUS，有个 *Android.mk* 文件添加到源文件目录 *external/dbus/* 下它才得以在 AOSP 上构建。最初没有文件用于构建，例如，只有当它在 AOSP 建立时，配置脚本才开始使用。一个简单的方法来解决这一问题是，为那些你需要的传统包生成一个独立的 AOSP 的根文件系统，然后将结果合并到 AOSP 中。

另一方面，重用现有的传统资源构建系统是有利的。比如说，没有理由不使用像 Yocto 或 BuildRoot 这样的工具产生适合你需要的根文件系统，然后将结果与 AOSP 合并。事实上，有许多现有的构建系统和包系统，它们使用传统的方法与 AOSP 混合可以产生非常有用的输出。在某些情况下，成本 / 效益方程可能会让你不可思议地把包的构建系统接入到 AOSP 的构建系统，仅仅是根据原项目的代码库大小。

---

**注意：**目前的说明都没有阻止你尝试构建你的传统代码到 Bionic。但仅有轻微的可能性那就是所做的变化是最低限度的。另外，正如我在第 4 章中说的，你可以把调用现有的、递归的、基于 make 的构建脚本的 *Android.mk* 文件放在一起。

不过，知道如何规避 Bionic 是一个非常有用的技巧。所以我鼓励你接着读。

---

## 操作理论

一旦你决定你想要使传统的 Linux 用户空间的组件与 AOSP 共同工作，接下来的问题是如何做。这实际上是两个问题。首先，我们如何将传统的用户空间和 AOSP 整合到

同一个映像文件系统？第二，这种传统用户空间如何与 AOSP 的组件进行交互？让我们以解决前一个问题开始。

假设你正在用《构建嵌入式 Linux 系统》一书中所讲述的一种方法生成一个基于 glibc 根文件系统，图 A-1 阐述了如何构建根文件系统用以与 AOSP 进行交互的一般方法。本质上，项目环境变量 PRJROOT 用于掌握基于 glibc 的根文件系统的创建。AOSP 构建系统之后被修改以复制根文件系统的内容到 AOSP 产生的镜像中。既然 AOSP 最开始不包含 /bin 和 /lib 目录，那么这些目录将被创建并包含基于 glibc 的根文件系统的内容。

---

注意：接下来的这些解释都假设你已经有了一个真正的你想用 AOSP 进行合并的根文件系统或者你知道如何去创造一个。如果你没有或者不知道如何去创建一个，我建议你看看《构建嵌入式 Linux 系统》（这本书实际上是由你们自己写的）。

---

一旦传统组件整合到 AOSP 这一问题解决了，另一个关键的问题是讨论如何使用这些组件和 / 或与它们在 AOSP 中进行交互。简单地说，所有的命令行工具和二进制文件可以直接从 Android 的命令行拿来使用。例如，如果在 Android 路径中有 /bin/foo 和 /bin 目录，你可以继续，并输入类似 adb shell 这样的命令，然后在命令行键入 foo 来运行二进制文件。有可能你会想做这样的事，例如，集成到 Android 的 init 操作中，而我们不久就会讨论它。

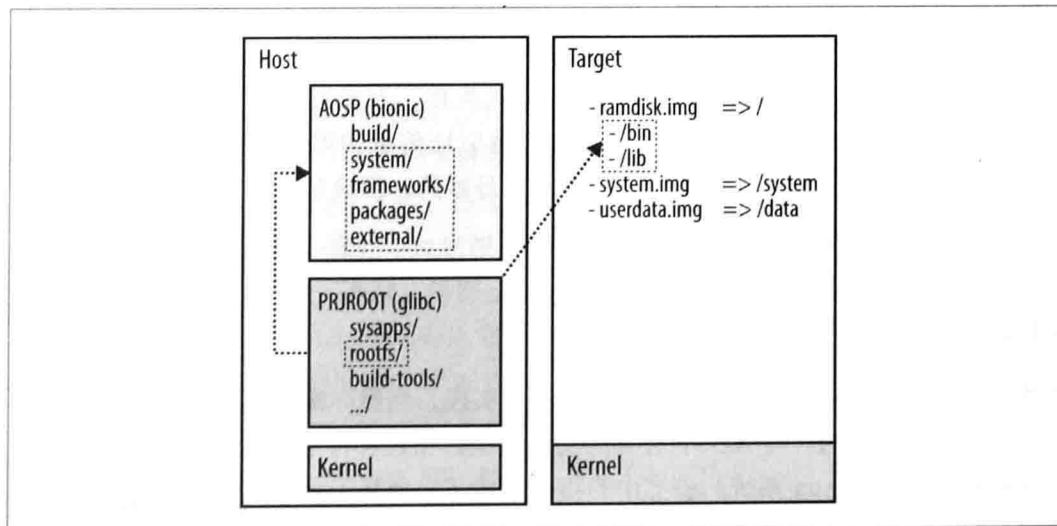


图 A-1：传统 Linux 用户空间与 AOSP 合并

除基本的命令行操作和初始化配置之外，一个更基本的讨论点是如何使运行在不同 C 库上的组件一起通信。比如说，如何将守护程序链接到 glibc，如何同步一个链接到

Bionic 的守护进程？或链接到 glibc 的命令行工具如何与链接到 Bionic 的守护进程进行通信？

记住，尽管被链接到不同的 C 库，一切资源仍运行在相同的内核。因此，在内核中不管 IPC 机制如何存在，仍然可以被任何在其上运行的二进制文件使用。正如你可以在图 A-2 看到的，有一个基于 glibc 的组件使用常规的 IPC 机制与一个基于 Bionic 的组件在 AOSP 中进行通信是完全可行的。例如，套接字就是一个最好的选择，考虑到它们基于 glibc 和 Bionic 均有实现。另一方面，System V 的 IPC 机制只有在 glibc 中可用。你也可以考虑采用 Binder，尽管这需要你将 libbinder 编译到 glibc 中。

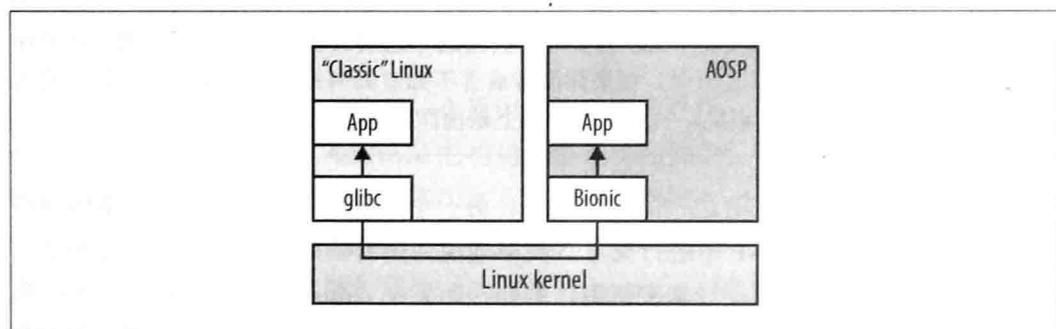


图 A-2：一种基于 glibc 的堆栈和 AOSP 之间的通信

例如，与我工作过的许多开发团队，多年来已经开发了大量的基于 glibc 的堆栈，它们通常运行在嵌入式 Linux 系统。虽然致力于将 Android 整合到他们的产品线，但他们经常面临这样一个抉择，是将堆栈及其控制逻辑接入到 Bionic，还是找出一种方法使那些传统的堆栈与 AOSP 以一种友好的方式共存。大多数这些团队的一个潜在的方法是创建一个像我刚刚描述的安装文件，然后让传统的堆栈的控制逻辑与新建的 Android 组件使用套接字进行通信。这不是灵丹妙药，但这是一个有用的技巧用于掌控应用于你的设计或设计的一部分的情况。

## 与 AOSP 合并

既然已经阐述了要点，让我们把这个方法付诸实践。当然你需要做的第一件事是一个有多功能的、传统的、与 AOSP 合并的文件系统。在这种特定的情况下，假设我沿用了《构建嵌入式 Linux 系统》一书中的指令创建了一个基于 glibc 的包含 busybox 的文件系统。因此，我们有如下这样的一些资源：

```
$ ls -l ${PRJROOT}/rootfs
total 16
drwxr-xr-x 2 karim karim 4096 2012-10-26 23:12 bin
drwxr-xr-x 2 karim karim 4096 2012-10-26 23:12 lib
```

```
lrwxrwxrwx 1 karim karim 11 2012-10-26 23:12 linuxrc -> bin/busybox  
drwxr-xr-x 2 karim karim 4096 2012-10-26 23:12 sbin  
drwxr-xr-x 4 karim karim 4096 2012-10-26 23:12 usr
```

为了简单点，可将那个根文件系统复制到我的 AOSP 中的新目录：

```
$ cp -a ${PRJROOT}/rootfs path_to_my_aosp/rootfs-glibc/
```

现在在我的 AOSP 顶层有一个 *rootfs-glibc* 目录。这个目录不会被过多使用，但是，由于没有考虑到 *Android.mk* 文件，如果你在这个时候构建 AOSP，它将会被完全忽略。为了解决这个问题，我们可以创建这样一个 *Android.mk*，迫使 AOSP 构建系统复制我们的基于 glibc 的根文件系统内容。下面是我的 *rootfs-glibc/Android.mk* 文件，基于 Android 2.3 版本，作为完成这项工作的一个例子：

```
LOCAL_PATH:= $(call my-dir)  
include $(CLEAR_VARS)  
  
# This part is a hack, we're doing "addprefix" because if we don't,  
# this dependency will be stripped out by the build system  
GLIBC_ROOTFS := $(addprefix $(TARGET_ROOT_OUT)/, rootfs-glibc)  
  
$(GLIBC_ROOTFS):  
	mkdir -p $(TARGET_ROOT_OUT)  
	cp -af $(TOPDIR)rootfs-glibc/* $(TARGET_ROOT_OUT)  
	rm $(TARGET_ROOT_OUT)/Android.mk  
	# The last command just gets rid of this very .mk since it's copied as is  
  
ALL_PREBUILT += $(GLIBC_ROOTFS)
```

这将导致 *rootfs-glibc* 中的内容被合并到由 AOSP 产生的 *ramdisk.img* 文件。虽然说那不足以使基于 glibc 的堆栈准确地作用于产生的根文件系统。事实上，正如我在第 6 章所说的，*rootfs* 中所有文件的文件系统权限只取决于 *system/core/include/private/android\_filesystem\_config.h* 文件，该文件已被修改，以保持 */lib* 目录下文件的可执行性。否则，把基于 glibc 的组件放到根文件系统的 */lib* 目录，但却是不可执行的，以致所有的链接到 glibc 的二进制文件都会运行失败。因此，正如我在第 6 章中所做的，你需要在 *android\_filesystem\_config.h* 文件中找到 *android\_files* 阵列并且修改它，使它看起来有些像 Android 2.3 中的下列形式：

```
...  
{ 00750, AID_ROOT, AID_SHELL, "sbin/*" },  
{ 00755, AID_ROOT, AID_ROOT, "bin/*" },  
{ 00755, AID_ROOT, AID_ROOT, "lib/*" },  
{ 00750, AID_ROOT, AID_SHELL, "init*" },  
{ 00644, AID_ROOT, AID_ROOT, 0 },  
};
```

通过这些修改，我们链接到 glibc 的二进制文件将在由 AOSP 生成的根文件系统中工

作得很好。然而，这并不理想，因为我们使用 Android 的 shell 和 Toolbox 的命令时，相比 BusyBox 的功能，这两者的能力都是严重受限。理想的情况下，我们应该使用 BusyBox 的 shell 程序和它的命令行工具。需要更多的变化使之成为现实。首先，我们需要修改 *init.rc* 以使新增到 */bin* 目录的内容包含 busybox 的命令，优先于 */system/bin* 出现在 PATH 中，其中包含工具箱的命令。下面是 Android 2.3 中 *system/core/rootdir/init.rc* 修改后的文件：

```
...
# setup the global environment
export PATH /bin:/sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin
export LD_LIBRARY_PATH /vendor/lib:/system/lib
export ANDROID_BOOTLOGO 1
export ANDROID_ROOT /system
...
...
```

最后，至少在 Android 2.3 的情况下，我们会想用 Busybox 的 shell 程序代替默认的 Android 的 shell 程序。那样做有两件事要改变，首先，我们需要修改 *init.rc*，使其使用 BusyBox 的 shell 作为控制台。默认情况下，下面是 *init.rc* 如何启动控制台的：

```
service console /system/bin/sh
...
```

使用 BusyBox 的 shell 程序，而不是默认的 Android 的 shell 程序，我们需要做的是使 *init.rc* 文件运行 */bin/sh*，而不是 */system/bin/sh*：

```
service console /bin/sh
...
```

同时，如果 *adb* 的 shell 程序允许我们访问 BusyBox 的 shell 程序，这也将很棒。运行在目标设备上的 *adbd* 的 shell 程序定义在 *system/core/adb/services.c* 文件中：

```
...
#if ADB_HOST
#define SHELL_COMMAND "/bin/sh"
#else
#define SHELL_COMMAND "/system/bin/sh"
#endif
...
...
```

在这里我们需要做的是注释掉默认值，使 *adbd* 运行 */bin/sh*：

```
...
#if ADB_HOST
#define SHELL_COMMAND "/bin/sh"
#else
//#define SHELL_COMMAND "/system/bin/sh"
#define SHELL_COMMAND "/bin/sh"
#endif
...
...
```

...

这些变化的总和将给我们一个新的 AOSP 根文件系统，它包含 glibc 和 Busybox，并采用 BusyBox 的 shell 程序作为默认的 shell 程序以及 Busybox 的命令作为其默认的命令。

---

**警告：**如果你使用的是 Android 4.2/ 版本，替换默认的 shell 程序或 Toolbox 默认的命令可能不会像在 Android 2.3 版本中那么有用。原因是，AOSP 用 *mksh* 取代了旧的 *sh*，*mksh* 命令提供了许多现代 shell 程序的特点，以及一些 Toolbox 的基本命令，如 *ls*，它们已被固定以消除它们最明显的缺陷。

---

## 采用组合栈

一旦用新的根文件系统启动系统，你会得到 Busybox 和 glibc 带来的所有的好处。下面是在 Android 2.3 版本中与 Android 的 shell 程序以及 Toolbox 命令的 shell 会话：

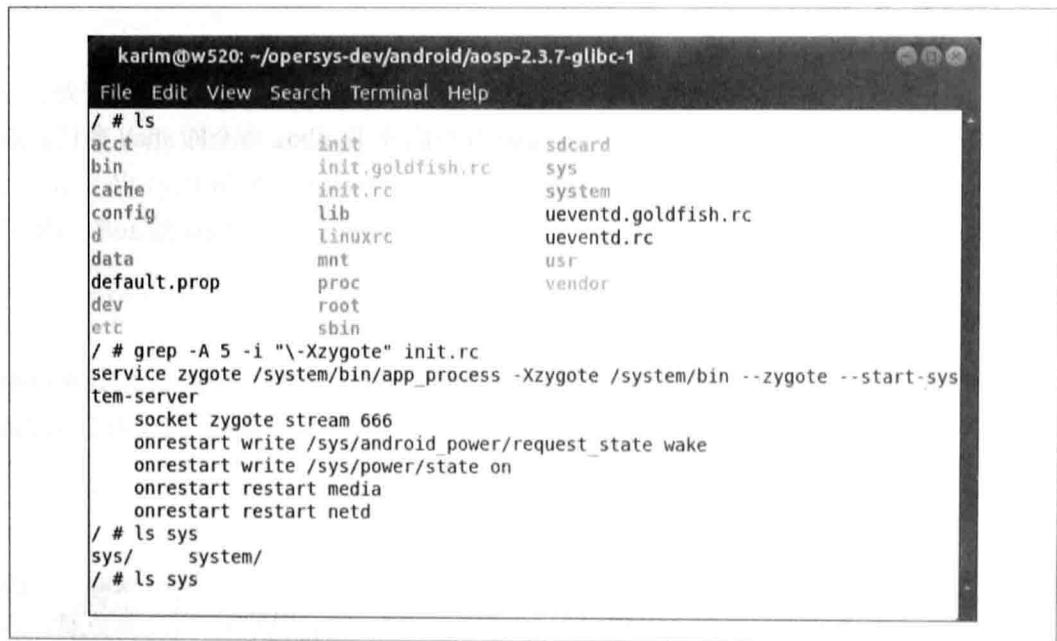
```
# ls
config
cache
sdcard
acct
mnt
vendor
d
etc
...
init
default.prop
data
root
dev
# grep -A 5 -i "\-Xzygote" init.rc
grep: not found
# ls sys[TAB] [TAB] [TAB]
```

你可以看到，*ls* 的输出没有按字母顺序排列，*grep* 是一个无法识别的命令，*tab* 输入根本不存在。下面是 Busybox 的相同命令：

```
/ # ls
acct           init          sdcard
bin            init.goldfish.rc  sys
cache          init.rc        system
config         lib           ueventd.goldfish.rc
d              linuxrc       ueventd.rc
data           mnt          usr
default.prop   proc          vendor
dev            root
etc            sbin
/ # grep -A 5 -i "\-Xzygote" init.rc
service zygote /system/bin/app_process -Xzygote /system/bin --zygote
```

```
--start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
/ # ls sys[TAB][TAB]
sys/ system/
/ # ls sys
```

此外，Android 的 shell 程序没有任何一种颜色编码来区分不同文件的类型或者文件所在的目录，但是 Busybox 的 shell 程序有，你可以在图 A-3 看到。



The screenshot shows a terminal window titled "karim@w520: ~/opersys-dev/android/aosp-2.3.7-glibc-1". The window contains the following command-line session:

```
File Edit View Search Terminal Help
/ # ls
acct           init          sdcard
bin            init.goldfish.rc   sys
cache          init.rc        system
config         lib           ueventd.goldfish.rc
d              linuxrc       ueventd.rc
data           mnt          user
default.prop   proc          vendor
dev             root
etc            sbin

/ # grep -A 5 -i "\-Xzygote" init.rc
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-sys
tem-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
/ # ls sys
sys/ system/
/ # ls sys
```

图 A-3: Busybox 的 shell 会话示例

但 BusyBox 没有停留在这里。除了包括命令如 *vi*，能让你直接在目标上编辑文件之外，Busybox 还包含一些常见的守护进程，如 *httpd* 和 *sendmail*。如果你试图在典型的 Android 设备上用浏览器连接到 80 端口，你会得到如图 A-4 的一些显示。

然而，如果 BusyBox 在你的目标设备上是可用的，你可以在 *init.rc* 文件为你的 *httpd* 添加一个服务声明：

```
service httpd /usr/sbin/httpd
    oneshot
```

然后你可以连接到它，如图 A-5 所示，404 消息事实上是从 Web 服务器发过来的正确的信息，表明没有 *index.html* 可用。

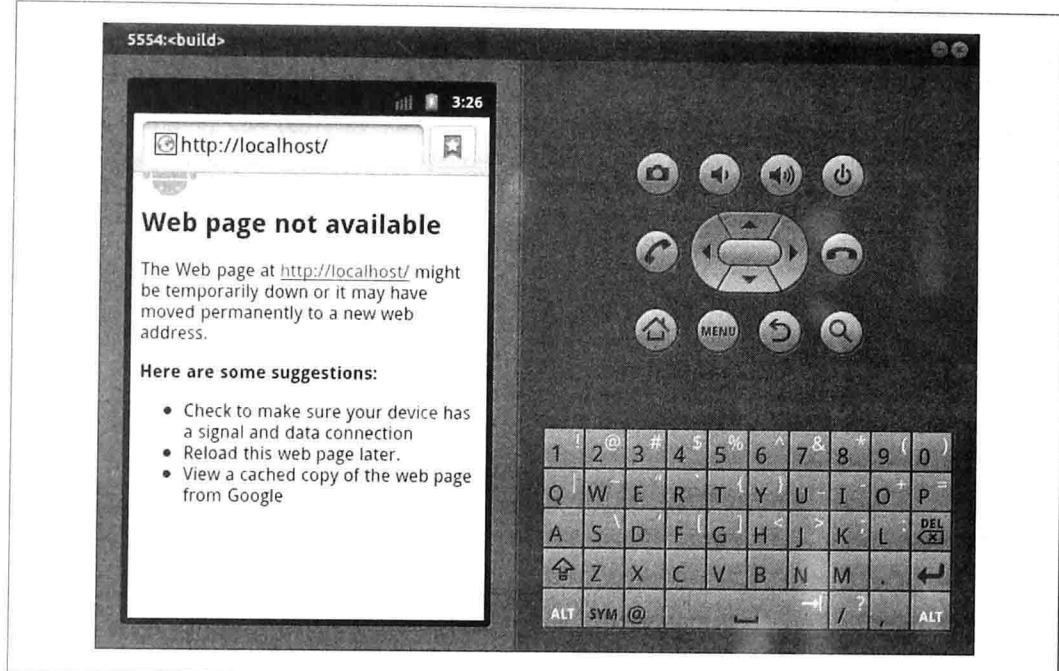


图 A-4：浏览器试图连接到本地主机

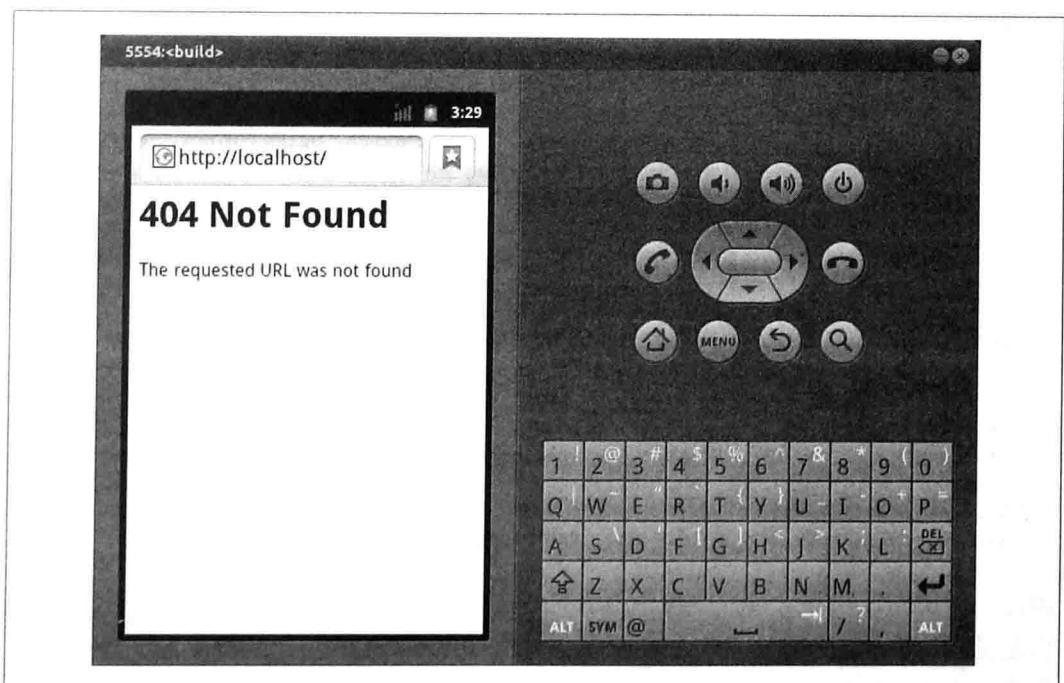


图 A-5：浏览器连接到 BusyBox 的 httpd

按照一般规则，Busybox的命令集是远远大于Toolbox的。下面是Toolbox在Android 2.3版本中的命令集，例如：

```
cat, chmod, chown, cmp, date, dd, df, dmesg, getevent, getprop, hd, id,  
ifconfig, iftop, insmod, ioctl, ionice, kill, ln, log, ls, lsmod, lsof, mkdir,  
mount, mv, nandread, netstat, newfs_msdos, notify, printenv, ps, reboot, renice,  
rm, rmdir, rmmod, route, schedtop, sendevent, setconsole, setprop, sleep, smd,  
start, stop, sync, toolbox, top, umount, uptime, vmstat, watchprops, wipe
```

在Android 4.2版本中有6倍多的命令。下面是BusyBox的命令集：

```
[, [[, acpid, add-shell, addgroup, adduser, adjtimex, arp, arping, ash, awk,  
base64, basename, beep, blkid, blockdev, bootchartd, brctl, bunzip2, bzipcat,  
bzip2, cal, cat, catv, chat, chattr, chgrp, chmod, chown, chpasswd, chpst,  
chroot, chrt, chvt, cksum, clear, cmp, comm, cp, cpio, crond, crontab, cryptpw,  
cttihack, cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devmem, df,  
dhcprelay, diff, dirname, dmesg, dnsd, dnsdomainname, dos2unix, du, dumpkmap,  
dumpleases, echo, ed, egrep, eject, env, envdir, envuidgid, ether-wake, expand,  
expr, fakeidentd, false, fbset, fbsplash, fdflush, fdformat, fdisk, fgconsole,  
fgrep, find, findfs, flock, fold, free, freeramdisk, fsck, fsck.minix, fsync,  
ftpd, ftpget, ftpput, fuser, getopt, getty, grep, gunzip, gzip, halt, hd,  
hdparm, head, hexdump, hostid, hostname, httpd, hush, hwclock, id, ifconfig,  
ifdown, ifenslave, ifplugged, ifup, ineted, init, insmod, install, ionice, iostat,  
ip, ipaddr, ipcalc, ipcrm, ipcs, iplink, iproute, iprule, iptunnel, kbd_mode,  
kill, killall, killall5, klogd, last, length, less, linux32, linux64, linuxrc,  
ln, loadfont, loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr,  
ls, lsattr, lsmod, lspci, lsusb, lzcat, lzma, lzop, lzopcat, makedevs, makemime,  
man, md5sum, mdev, mesg, microcom, mkdir, mkdosfs, mke2fs, mkfifo, mkfs.ext2,  
mkfs.minix, mkfs.vfat, mknod, mkpasswd, mkswap, mktemp, modinfo, modprobe, more,  
mount, mountpoint, mpstat, mt, mv, nameif, nbd-client, nc, netstat, nice,  
nmeter, nohup, nslookup, ntpd, od, openvt, passwd, patch, pgrep, pidof, ping,  
ping6, pipe_progress, pivot_root, pkill, pmap, popmaildir, poweroff, powertop,  
printenv, printf, ps, pscan, pwd, raidautorun, rdate, rdev, readahead, readlink,  
readprofile, realpath, reboot, reformime, remove-shell, renice, reset, resize,  
rev, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run-parts, runlevel,  
runsv, runsvdir, rx, script, scriptreplay, sed, sendmail, seq, setarch,  
setconsole, setfont, setkeycodes, setlogcons, setsid, setuidgid, sh, sha1sum,  
sha256sum, sha512sum, showkey, slattach, sleep, smemcap, softlimit, sort, split,  
start-stop-daemon, stat, strings, stty, su, slogin, sum, sv, svlogd, swapoff,  
swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, tcpsvd, tee, telnet,  
telnetd, test, tftp, tftpd, time, timeout, top, touch, tr, traceroute,  
traceroute6, true, tty, ttysize, tunctl, udhcpc, udhcpd, udpsvd, umount, uname,  
unexpand, uniq, unix2dos, unlzma, unlzop, unxz, unzip, uptime, usleep, uudecode,  
uuencode, vconfig, vi, vlock, volname, wall, watch, watchdog, wc, wget, which,  
who, whoami, xargs, xz, xzcat, yes, zcat, zcip]
```

因此，即使你仅在开发过程中使用Busybox，然后在成品镜像文件中丢弃它，其好处仍是显而易见的。事实上，如果你已经习惯了Busybox却被迫使用普通的Toolbox，那很可能是一种折磨。

同时，如果你看看 `/lib`，你会找到所有的常规的你熟悉的 glibc 组件。但如果你使用纯 AOSP 这些都是不存在的：

```
/ # ls /lib
ld-2.9.so          libm-2.9.so          libnss_nisplus-2.9.so
ld-linux.so.3       libm.so.6            libnss_nisplus.so.2
libBrokenLocale-2.9.so libmemusage.so    libpcprofile.so
libBrokenLocale.so.1 libnsl-2.9.so      libpthread-2.9.so
libSegFault.so     libnsl.so.1         libpthread.so.0
libanl-2.9.so      libnss_compat-2.9.so libresolv-2.9.so
libanl.so.1        libnss_compat.so.2   libresolv.so.2
libc-2.9.so        libnss_dns-2.9.so   librt-2.9.so
libc.so.6          libnss_dns.so.2     librt.so.1
libcrypt-2.9.so    libnss_files-2.9.so libthread_db-1.0.so
libcrypt.so.1      libnss_files.so.2    libthread_db.so.1
libdl-2.9.so       libnss_hesiod-2.9.so libutil-2.9.so
libdl.so.2         libnss_hesiod.so.2   libutil.so.1
libgcc_s.so        libnss_nis-2.9.so
libgcc_s.so.1      libnss_nis.so.2
```

## 注意事项及悬而未决的问题

现在你可以看到什么已经做了，让我们看看这个类型的配置需要什么。首先，新的 C 库和任何你添加的二进制文件都将使根文件系统更大。而由 2.3.x AOSP 构建的默认 `ramdisk.img` 文件大约是 144KB，那些包含 glibc 和 Busybox 的上述文件大概是 2.6MB。你当然可以裁剪基于 glibc 的根文件系统，嵌入式 Linux 开发者一直都是这样做的，通过消除不必要的 glibc 部件及使用 `strip` 命令实现裁剪。它可能也可以使存储在你的嵌入式系统成为小事一桩。毕竟，在同一构建中，`system.img` 文件是 66MB。

---

**注意：**当然，你可以在不同于 `/lib` 的另一个位置安装 glibc 库，但是避免使用目录 `/bin`。例如，你可以创建一个 `/legacy` 目录，然后把你所有的传统的内容复制到该目录，并将其从一个单独的镜像中挂载以使根文件系统内存的 RAM 磁盘使用最小，当然它是默认的。不过，这显然比只使用传统的 `/bin` 和 `/lib` 通过 FHS 来说明要简单得多。

---

事实上，你现在已经有了两个 C 库需要被加载到 RAM，分别是 Bionic 和 glibc。再次强调，这可能是你设计中的一桩小事，但你应该意识到这一点。一个地方添加库没有影响，这是 CPU 的性能。只有强行地装载你打包的额外的二进制文件，才会影响 CPU 性能。

一个更微妙的问题是如何处理 `/etc`。事实上，在 Android 的根文件系统中 `/etc` 是符号链接到 `/system/etc`。这是背离 FHS，但在 AOSP 中运行正常的。如果你有一个传统的嵌入式 Linux 文件系统，且你想要将它与 AOSP 的根文件系统合并，你将面临一个选择。或者是拷贝你的 `/etc` 目录的内容到 `/system/etc` 中且保持其原本的象征性链接，或者是

拷贝 */system/etc* 的内容到你的 */etc* 中。很令人困惑，但是并不妨碍你使用这里提到的技术。

在运行时，你可能会遇到一些问题，因为 Toolbox 的工具是基于工作于不同的假设下，不同于普通 Linux 中的使用。通常，例如，*ps* 使用 */etc/passwd* 匹配 UID。在 Android 的情况下，没有 */etc/passwd* 目录。相反，用户和组是硬编码在 *android\_filesystem\_config.h* 文件中的，我们以前有提到过此文件。因此，BusyBox 的 *ps* 是无法通过进程匹配用户名。

```
/ # ps
 PID  USER   TIME   COMMAND
 1 0      0:08 /init
 ...
 26 0      0:00 /sbin/ueventd
 27 1000    0:00 /system/bin/servicemanager
 28 0      0:00 /system/bin/vold
 29 0      0:00 /system/bin/netd
 30 0      0:00 /system/bin/debuggerd
 31 1001    0:00 /system/bin/rild
 32 0      0:10 zygote /bin/app_process -Xzygote /system/bin --zygote --s
 33 1013    0:00 /system/bin/mediaserver
 34 1002    0:00 /system/bin/dbus-daemon --system --nofork
 35 0      0:00 /system/bin/installld
 36 1017    0:00 /system/bin/keystore /data/misc/keystore
 38 0      0:00 /system/bin/qemud
 40 2000    0:00 /system/bin/sh
 41 0      0:00 /sbin/adbd
 64 1000    0:22 system_server
 116 10018   0:01 com.android.inputmethod.latin
 124 1001    0:03 com.android.phone
 125 1000    0:18 com.android.systemui
 ...
...
```

Toolbox 的 *ps* 没有这样的问题：

```
# ps
USER     PID   PPID  VSZ RSS   WCHAN   PC      NAME
root     1      0    268  180 c009b74c 0000875c S /init
...
root     26     1    232  136 c009b74c 0000875c S /sbin/ueventd
system   27     1    804  188 c01a94a4 afd0b6fc S /system/bin/servicemanager
root     28     1   3864  300 ffffffff afd0bdac S /system/bin/vold
root     29     1   3836  316 ffffffff afd0bdac S /system/bin/netd
root     30     1    664  176 c01b52b4 afd0c0cc S /system/bin/debuggerd
radio    31     1   5396  432 ffffffff afd0bdac S /system/bin/rild
root     32     1   60876 16396 c009b74c afd0b844 S zygote
media   33     1   17976 1000 ffffffff afd0b6fc S /system/bin/mediaserver
bluetooth 34     1   1256  216 c009b74c afd0c59c S /system/bin/dbus-daemon
root     35     1    812  220 c02181f4 afd0b45c S /system/bin/installld
keystore 36     1   1744  200 c01b52b4 afd0c0cc S /system/bin/keystore
root     38     1    824  260 c00b8fec afd0c51c S /system/bin/qemud
```

```

shell      40      1      732      192      c0158eb0 afd0b45c S /system/bin/sh
root      41      1      3364     168      ffffffff 00008294 S /sbin/adbd
system    64      32     119832    26144      ffffffff afd0b6fc S system_server
app_18    116      32     77272     17604      ffffffff afd0c51c S com.android.inputmethod.latin

radio     124      32     86120    17996      ffffffff afd0c51c S com.android.phone
system    125      32     73320     19012      ffffffff afd0c51c S com.android.systemui
...

```

有时 Toolbox 的命令有不同于传统的 Linux 命令的参数。例如，Toolbox 的 *ps* 接受一个参数 -t 列出添加到该进程中的所有线程：

```

# ps -t
...
system    64      32     119832    26144      ffffffff afd0b6fc S system_server
system    65      64     119832    26144      c0059e24 afd0c738 S HeapWorker
system    66      64     119832    26144      c0059e24 afd0c738 S GC
system    67      64     119832    26144      c0047be8 afd0bfec S Signal Catcher
system    68      64     119832    26144      c02181f4 afd0c22c S JDWP
system    69      64     119832    26144      c0059e24 afd0c738 S Compiler
system    70      64     119832    26144      c01a94a4 afd0b6fc S Binder Thread #
system    71      64     119832    26144      c01a94a4 afd0b6fc S Binder Thread #
system    72      64     119832    26144      c0059e24 afd0c738 S SurfaceFlinger
system    74      64     119832    26144      c0047be8 afd0bfec S DisplayEventThr
system    75      64     119832    26144      c00b8fec afd0c51c S er.ServerThread
system    77      64     119832    26144      c00b8fec afd0c51c S ActivityManager
system    81      64     119832    26144      c0059f2c afd0c738 S ProcessStats
system    82      64     119832    26144      c00b8fec afd0c51c S PackageManager
system    83      64     119832    26144      c00b7db0 afd0b45c S FileObserver
system    84      64     119832    26144      c00b8fec afd0c51c S AccountManagerS
system    86      64     119832    26144      c00b8fec afd0c51c S SyncHandlerThre
...

```

Busybox 的 *ps* 期待 -T (用大写代替小写 t) 替代：

```

/ # ps -t
ps: invalid option -- 't'
BusyBox v1.18.3 (2011-03-09 09:33:40 PST) multi-call binary.

Usage: ps [-o COL1,COL2=HEADER] [-T]

Show list of processes

Options: -o COL1,COL2=HEADER Select columns for display -T Show threads

```

在大多数情况下，这些不兼容会引起人的困惑，但不会有实际的破坏。而且，最终我们还没有摆脱 Toolbox 或任何默认的 AOSP 命令。所以你仍然可以通过提供完整的命令路径来调用任何 Toolbox 的命令：

```

# /system/bin/ps
USER      PID      PPID      VSIZE      RSS      WCHAN      PC      NAME

```

```
root      1      0      268    180    c009b74c 00000875c S /init
root      2      0      0       0    c004e72c 00000000 S kthreadd
root      3      2      0       0    c003fdc8 00000000 S ksoftirqd/0
root      4      2      0       0    c004b2c4 00000000 S events/0
root      5      2      0       0    c004b2c4 00000000 S khelper
root      6      2      0       0    c004b2c4 00000000 S suspend
root      7      2      0       0    c004b2c4 00000000 S kblockd/0
...
...
```

至少有一种情况我已经注意到，在 PATH 中将 Busybox 置于 Toolbox 前面会造成破坏。比如，在 *dumpstate* 的例子中，PATH 中默认的 *ps* 命令用来检索运行的线程列表。然而，由于 BusyBox 的 *ps* 期待 *-T* 代替 *-t*，*dumpstate* 命令相应部分的输出会有损坏。

另一个值得一提的实质性的区别是名称解析。事实上，Android 管理 DNS 的方式与它在 glibc 和 BusyBox 中完成的非常不同。所以这可能是你要关注的一个问题。

---

**警告：**一些人认为 Toolbox 的一个优势在于它很受限制的命令集：它限制了一个恶意的用户或第三方利用系统进行的攻击。从这一点出发，利用 Busybox 会导致安全风险的增加。要当心。

---

## 连接 Busybox 到 Bionic

在本节的介绍中，Busybox 相比 AOSP 的默认的命令行显得耀眼些。所以，事实上，很多人觉得需要用它来为他们的 AOSP 树工作。因此，现在从 <http://busybox.net> 获取的默认的树包含对 Android 的支持。即已添加补丁使能连接 BusyBox 到 Bionic 来运行，而同时又支持如 glibc 这样的库。此外，还有一个 android\_build 脚本在 BusyBox 的源代码的 examples/ 目录下用于构建它到一个给定的 AOSP 源代码组中。

然而，无论你是否把它链接到 Bionic 或 glibc，你仍然必须找到一个办法使它与 AOSP 的剩余部分在同一个文件系统中共存。因此，上述解释只是使得你可以忽略库的影响。

## 向前移动

显然，对于这种方法你还有比我给你多很多的东西可以做。例如，即使你不包括 Busybox 或你选择链接其他的库而不是 glibc，例如 uClibc 或 eglIBC，知道如何移植一个“经典”的 C 库到你的根文件系统是一个有用的技巧。

我鼓励你看看一些项目，如 BuildRoot 和 Yocto，了解你如何能利用它们的工作获得额外的工具和库以整合到你的 AOSP 中，得到一个更通用的结果。记住，Android 的愿景和开发方法限制了添加到 AOSP 的只有那些符合谷歌计划的软件包。事实上，你的项目可能已经与谷歌目前市场上的目标毫无共同之处，所以纯 AOSP 可能对于你的项目来说是严重不够的。

在这里介绍的并不是实现目标结果的唯一办法，还有许多方法来实现。一般来说，这种解释是让你明白，你可以建设性地突破 AOSP 严格的模板，从经典的嵌入式 Linux 工作中纳入你最终的根文件系统的元素。这是一项伟大的工作，因为它将开启多年对嵌入式 Linux 系统创建的庞大工作的利用之门。这可以利用包括邮件列表、会议、书籍等，最重要的是有一个非常强大的开发团体。

# 为新硬件增加支持

有很多情况下，你的嵌入式系统会包含 Android 所不支持的设备。增加对新类型硬件的支持是一件很棘手的事情，因为它需要对 Android 内部有一个深入的认识，不过 AOSP 的模块化使得它也是可行的。这个附录要给你展示的是如何扩展 Android 的各层来支持你自己类型的硬件。

---

**注意：**虽然你可能并不会实际为你的系统增加硬件，但是如果你想试着理解 Android 框架各层之间的复杂关系，你会发现这个附录还是很有启发性的。

同样的，虽然这个附录展示的是对 2.3/ 姜饼系统修改，但这个机制和对 Java 代码的修改跟 4.2/ 果冻豆中很接近，主要的差异会在下文中指出来。

---

## 基础知识

正如在第 2 章中所说，Android 系统跟“纯洁 Linux”相比，它需要的不仅仅是恰当的驱动程序来使得硬件能够工作。Android 定义了一个硬件抽象层（HAL），为每一种 Android 核心支持的每种硬件类型定义了 API 接口。为了一个硬件组件能够恰当地跟 Android 交互，必须要有一个硬件的“模块”（跟内核模块没关系）跟某个硬件类型的 API 兼容。

通常来说，每个 Android 支持的硬件类型都有一个系统服务和 HAL 定义。比如说，有一个灯光服务和灯光 HAL 定义，有 WiFi 服务和 WiFi 的 HAL 定义，电源管理、位置服务、传感器等等也都这样。图 2-3 展示了 Android 硬件支持系统的整体结构，当然，正如我们前面讨论过的，大量的系统服务是运行在 System Server 中的。

有两个通用 HAL 模块类型：显式装载（通过 `dlopen` 函数调用）和通过动态链接器自动装载（因为它们跟 `libhardware_legacy.so` 链接）。前者的 API 在 `hardware/libhardware/include/hardware` 中，而后者在 `hardware/libhardware_legacy/include/hardware_legacy`。Android 的趋势是将代码从 `legacy` 中移除。动态库 `.so` 文件和实际驱动程序之间的接口是通过 `/dev` 目录下的设备节点或者由厂商自行定义，Android 不关心这个，它只关心如何找到相应的 HAL 动态库文件。

我经常被问到一个问题：“如何在 Android 系统中增加自己类型的硬件支持？”为了展示这个答案，我创建了一个叫作 `opersys-hal-hw` 的设备类型并且把这种 HAL 类型的实现代码上传到了 GitHub (<https://github.com/opersys/opersys-hal-hw>)，还有一个非常基本的环形缓冲驱动程序 (<https://github.com/opersys/circular-dirver>)。

如果你复制 `opersys-hal-hw` 项目内容到现有的 AOSP 项目 2.3.7\_r1 版本，然后为模拟器构建一个镜像，包含 `opersys` 服务。后者依赖于环形缓冲驱动来实现非常基本的新硬件类型。显然，这仅仅是一个简单的架构程序，应该在增加新硬件类型方面给你些许灵感。当然，你的硬件很可能是完全不一样的接口。

## 系统服务

为了展示一个新的系统服务是如何实现的，我首先在 `frameworks/base/services/java/com/android/server/` 增加了一个 `OpersysService.java`。这个文件实现了 `OpersysService` 类，给外部提供了两个非常基本的调用。

```
public String read(int maxLength)
{
...
}

public int write(String mString)
{
...
}
```

如果阅读这个新硬件类型的代码，你会看到我如何在 Android 的每一层实现这些函数调用。所以，如果你看看系统服务的 `read` 函数，将会看到它是这样做的：

```
public String read(int maxLength)
{
    int length;
    byte[] buffer = new byte[maxLength];

    length = read_native(mNativePointer, buffer);
    return new String(buffer, 0, length);
}
```

这里最重要的工作是调用 `read_native` 函数来完成的，它本身在 `OpersysService` 类里面是这样声明的：

```
private static native int read_native(int ptr, byte[] buffer);
```

该方法通过 `native` 关键字向编译器声明该函数不在任何 Java 代码中实现，它需要 Davik 虚拟机运行时装载 JNI 的动态库文件。而且，实际上如果阅读 `frameworks/base/services/jni/` 目录，你会发现 `Android.mk` 和 `onload.cpp` 已经被修改，并把 `com_android_server_OopersysService.cpp` 放进来了。后者有一个 `register_android_server_OopersysService()` 函数，它在 `libandroid_servers.so` 装载时被调用，该动态库由刚刚提到的 `Android.mk` 产生。注册函数告诉 Davik 虚拟机 `com_android_server_OopersysService.cpp` 文件中为 `OpersysService` 实现的本地函数以及它们的调用方式：

```
static JNINativeMethod method_table[] = {
    { "init_native", "(I)V", (void*)init_native },
    { "finalize_native", "(I)V", (void*)finalize_native },
    { "read_native", "(I[B)I", (void*)read_native },
    { "write_native", "(I[B)I", (void*)write_native },
    { "test_native", "(II)I", (void*)test_native},
};

int register_android_server_OopersysService(JNIEnv *env)
{
    return jniRegisterNativeMethods(env, "com/android/server/OopersysService",
                                    method_table, NELEM(method_table));
}
```

上面结构体包含了三个域，第一个域是 Java 类中定义函数的名称，最后一个域是它对应的 C 语言实现的函数名。在这个例子中，两个名称是一样的，虽然大多数情况下我们会写成一样的名字，但这也不是必须的。中间这个参数看起来有一点神秘，括号里面字母是 Java 传递下来的参数类型描述，右括号后面字母是返回值的类型描述。例如，`init_native()` 函数不携带参数并返回一个整型值，而 `read_native()` 函数有两个参数，一个整型和一个字节数组，并且返回一个整型值。

---

**注意：**当你深入到 Android 内部，你会经常跟这样的 JNI 打交道。我建议你读一读 Sheng Liang 写的《Java Native Interface: Programmer's Guide and Specification》（Addison-Wesley 出版）以了解更多关于 JNI 使用的信息。

---

`read_native()` 函数的实现方法如下：

```
static int read_native(JNIEnv *env, jobject clazz, int ptr, jbyteArray buffer)
{
    opersyshw_device_t* dev = (opersyshw_device_t*)ptr;
```

```

jbyte* real_byte_array;
int length;

real_byte_array = env->GetByteArrayElements(buffer, NULL);

if (dev == NULL) {
    return 0;
}

length = dev->read((char*) real_byte_array, env->GetArrayLength(buffer));

env->ReleaseByteArrayElements(buffer, real_byte_array, 0);

return length;
}

```

首先，函数本身比 JNI 描述多两个参数。所有 JNI 调用都是以这两个参数开始的：一个叫作 `env` 的虚拟机句柄以及名为 `clazz` 的指向 `this` 对象的指针。然后要注意，数组的用法跟平时也不一样。需要使用 `env->GetByteArrayElements()` 函数来获得在 C 代码中可以使用的数组，并且用完后要使用 `env->ReleaseByteArrayElements()` 来释放掉。实际上，千万不要忘记 JNI 函数是在 C 语言的世界里操作 Java 类型的对象。虽然一些对象（例如整型）可以被直接使用，其他对象（例如数组）是需要在使用前后进行转换的。

十分重要的是，`read_native()` 函数实际操作是通过调用 `dev->read()` 来完成的。这个函数指针指向了哪里呢？为了理解这部分，你需要看一看 `init_native()` 函数：

```

static jint init_native(JNIEnv *env, jobject clazz)
{
    int err;
    hw_module_t* module;
    opersyshw_device_t* dev = NULL;

    err = hw_get_module(OPERSYSHW_HARDWARE_MODULE_ID, (hw_module_t const**)&module);
    if (err == 0) {
        if (module->methods->open(module, "", ((hw_device_t**) &dev)) != 0)
            return 0;
    }

    return (jint)dev;
}

```

这个函数里做了两件重要的事情。首先，调用 `hw_get_module()` 函数请求 HAL 装载支持 `OPERSYSHW_HARDWARE_MODULE_ID` 类型硬件的模块。然后，调用模块的 `open()` 函数。我们会在后面再看看这两个函数，但是在这里请注意，第一个动作导致 `.so` 文件被加载到系统服务的地址空间，第二个动作使得那个库文件中的硬件相关函数，例如 `read()` 和 `write()`，被 `com_android_server_OopersysService.cpp` 调用，本质上也就是 C 实现的系统服务被增加到系统中了。

# HAL 及其扩展

*Hardware/* 目录下的 HAL 提供了前面提到的 `hw_get_module()` 函数。如果你进一步阅读代码，你会看到 `hw_get_module()` 以经典的 `dlopen()` 来实现的，`dlopen` 函数的功能就是装载一个动态库到进程空间。

---

注意：在任何 Linux 工作站上输入 `man dlopen` 都可以得到有关 `dlopen` 及其用法的更多信息。

---

当然，HAL 不会仅仅就是装载一个动态库而已。当你请求一个特定硬件类型的时候，它会检查 `/system/lib/hw` 目录查找匹配的文件名，文件名由硬件类型和运行的设备相关。所以，以我们现在说的这个新类型设备为例，它会找一个叫做 `opersyshw.goldfish.so` 的文件，`goldfish` 是模拟器的代码。实际上用于文件名中间部分的设备名称取自下列全局属性中的一个：`ro.hardware`、`ro.product.board`、`ro.board.platform` 和 `ro.arch`。动态库必须包含一个提供 HAL 硬件信息的结构体，结构体要命名为 `HAL_MODULE_INFO_SYM_AS_STR`。后面我们会看到一个例子。

对于新硬件类型本身的定义是另外一个头文件，在这个例子中是 `opersyshw.h`，其他类型硬件的定义在 `hardware/libhardware/include/hardware/`：

```
#ifndef ANDROID_OPERSYSHW_INTERFACE_H
#define ANDROID_OPERSYSHW_INTERFACE_H

#include <stdint.h>
#include <sys/cdefs.h>
#include <sys/types.h>

#include <hardware/hardware.h>

__BEGIN_DECLS

#define OPERSYSHW_HARDWARE_MODULE_ID "opersyshw"

struct opersyshw_device_t {
    struct hw_device_t common;

    int (*read)(char* buffer, int length);
    int (*write)(char* buffer, int length);
    int (*test)(int value);
};

__END_DECLS

#endif // ANDROID_OPERSYSHW_INTERFACE_H
```

这个文件中除了定义了 `read` 和 `write` 原型以外，还定义了 `OPERSYSHW_HARDWARE_MODULE_ID`。后者成为了在文件系统中查找 HAL 模块文件名的基础。

# HAL 模块

Android 的思想是每个设备需要一个相应的 HAL 模块来支持特定的硬件类型。例如不同的硬件厂商的手机会使用不同的图形芯片，因此也就有不同的 gralloc 模块。通常来说，添加到 AOSP 源的 HAL 模块在 *device/<vendor>/<product>* 的 *lib\** 目录里。不过对于模拟器来说，它支持的虚拟硬件放在  *sdk/emulator* 目录下，所以我们的硬件类型在 goldfish 中的实现代码就放在这里。

opersyshw 硬件类型并不太复杂，所以它在 Goldfish 中就一个文件 *opersyshw\_qemu.c*。为了这个文件产生的库文件可以被真正的 HAL 模块识别，它需要以这这样的代码片段结尾：

```
static struct hw_module_methods_t opersyshw_module_methods = {
    .open = open_opersyshw
};

const struct hw_module_t HAL_MODULE_INFO_SYM = {
    .tag = HARDWARE_MODULE_TAG,
    .version_major = 1,
    .version_minor = 0,
    .id = OPERSYSHW_HARDWARE_MODULE_ID,
    .name = "Opersys HW Module",
    .author = "Opersys inc.",
    .methods = &opersyshw_module_methods,
};
```

这里有一个叫做 *HAL\_MODULE\_INFO\_SYM* 的结构体，还有 *opersyshw\_module\_methods* 及其包含的 *open* 函数指针。之前在 HAL 模块被装载的时候，*init\_native* 调用了一个很相似的 *open()* 函数。相应的 *open\_opersyshw()* 函数如下所示：

```
static int open_opersyshw(const struct hw_module_t* module, char const* name,
                           struct hw_device_t** device)
{
    struct opersyshw_device_t *dev = malloc(sizeof(struct opersyshw_device_t));
    memset(dev, 0, sizeof(*dev));

    dev->common.tag = HARDWARE_DEVICE_TAG;
    dev->common.version = 0;
    dev->common.module = (struct hw_module_t*)module;
    dev->read = opersyshw_read;
    dev->write = opersyshw_write;
    dev->test = opersyshw_test;
    *device = (struct hw_device_t*) dev;

    fd = open("/dev/circchar", O_RDWR);

    D("OPERSYS HW has been initialized");
```

```
    return 0;
}
```

这个函数的主要功能是初始化 `device` 结构体，该结构体的类型就是 `opersyshw.h` 文件中定义的 `opersyshw_device_t`，然后打开 `/dev` 目录下的设备节点，也就跟装载在内核中的底层设备驱动程序建立了连接。显然，有些设备驱动可能需要做一些初始化工作，但对我们演示而言，这就足够了。

最后，这里再看一下 `opersyshw_read()` 函数：

```
int opersyshw_read(char* buffer, int length)
{
    int retval;
    D("OPERSYS HW - read() for %d bytes called", length);
    retval = read(fd, buffer, length);
    return retval;
}
```

这里我们没有做错误检测，但是你的代码里应该需要做。例如，我们甚至没有判断设备文件 `fd` 有没有被成功打开。通常来说我们应该要检测的。现在，调用路径很清晰了。系统服务的 `read` 导致对 JNI 函数 `read_native()` 的调用，通过 HAL，又调用到 HAL 的 `opersyshw_read()`。

目前，现有系统服务和 HAL 组件调用路径几乎都差不多，不过在它们的系统服务中还有其他大量的函数调用，因此在支持它们特定硬件类型时牵涉到更多的地方。

## 调用系统服务

到现在为止，我们主要关注的是新系统服务跟底层之间的接口，还没有讨论系统服务如何通过 `binder` 被其他系统服务和应用所调用。系统服务要被 `Binder` 所调用，至少需要有一个接口定义。以 `opersys` 服务为例，我们可以增加 `IOpersysService.aidl` 到 `frameworks/base/core/java/android/os/`：

```
package android.os;
/**
 * {@hide}
 */
interface IOpersysService {
    String read(int maxLength);
    int write(String mString);
}
```

这个修改使得 AOSP 中编译的代码可以调用到我们的系统服务。例如，我们可以在 `device/acme/coyotepad` 或者 `packages/apps/` 目录下增加一个 app，并在其 `onCreate()` 回调函数中这么做：

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
  
    IOpersysService om =  
        IOpersysService.Stub.asInterface(ServiceManager.getService("opersys"));  
    try {  
        Log.d(DTAG, "Going to write to the \"opersys\" service");  
        om.write("Hello Opersys");  
        Log.d(DTAG, "Service returned: " + om.read(20));  
    }  
    catch (Exception e) {  
        Log.d(DTAG, "FAILED to call service");  
        e.printStackTrace();  
    }  
}
```

注意我们这里使用了 `ServiceManager.getService()` 来得到 Binder 句柄指向系统服务，然后通过 `IOpersysService.Stub.asInterface()` 来把它转换成我们可以使用的 `IOpersysService` 对象。这对跟 AOSP 编译的代码来说是可行的，但是对于普通 app 来说就行不通了。因为 `ServiceManager.getService()` 在 SDK 中不存在。同样，如果你对 app 开发熟悉，你会发现这不是跟系统交互的常用方法，app 应该调用的是 `getSystemService()`。

为了系统服务能够通过我们的 SDK 来调用（当然这个 SDK 由我们采用 AOSP 构建而来），需要执行几个额外的步骤。首先，我们需要创建一个 `manager` 类来进一步把 Binder 可用的服务包装起来。我们在 `frameworks/base/core/java/android/os/` 目录中增加一个文件 `OpersysManager.java`：

```
package android.os;  
  
import android.os.IOpersysService;  
  
public class OpersysManager  
{  
    public String read(int maxLength) {  
        try {  
            return mService.read(maxLength);  
        } catch (RemoteException e) {  
            return null;  
        }  
    }  
}
```

```
public int write(String mString) {
    try {
        return mService.write(mString);
    } catch (RemoteException e) {
        return 0;
    }
}

public OpersysManager(IVisualizerService service) {
    mService = service;
}

IVisualizerService mService;
}
```

这里可以看到调用如何通过 Binder 调用转到系统服务。大多数预定义的 manager 具有相似的语意，虽然大多数会在执行调用之前有一些额外的逻辑，有些还会定义一些额外的接口。这跟 C 函数库类似，大多数 C 函数的功能要由内核来提供。

要使得这个 manager 能通过 `getSystemService()` 调用获得，还有两个步骤要做。首先，需要修改 `frameworks/base/core/java/android/content/Context.java`，让它能识别新的系统服务。

```
/**
 * Use with {@link #getSystemService} to retrieve a
 * {@link android.os.OpersysManager} for using Opersys Service.
 *
 * @see #getSystemService
 */
public static final String OPERSYS_SERVICE = "opersys";
```

然后，我们要修改 `frameworks/base/core/java/android/content/app/ContextImpl.java` 以使得 `getSystemService()` 函数能够识别这个新服务：

```
@Override
public Object getSystemService(String name) {
    if (WINDOW_SERVICE.equals(name)) {
        return WindowManagerImpl.getDefault();
    } else if (LAYOUT_INFLATER_SERVICE.equals(name)) {
        synchronized (mSync) {
...
    } else if (DOWNLOAD_SERVICE.equals(name)) {
        return getDownloadManager();
    } else if (NFC_SERVICE.equals(name)) {
        return getNfcManager();
    } else if (OPERSYS_SERVICE.equals(name)) {
        return getOpersysManager();
    }
}
```

```
private OpersysManager getOpersysManager() {
    synchronized (mSync) {
        if (mOpersysManager == null) {
            IBinder b = ServiceManager.getService(OPERSYS_SERVICE);
            IOpersysService service = IOpersysService.Stub.asInterface(b);
            mOpersysManager = new OpersysManager(service);
        }
    }
    return mOpersysManager;
}
...

```

---

**警告：**在 4.2/ 果冻豆中，`getSystemService` 函数实现跟前面代码已经有非常大的区别。请看在 `ContextImpl.java` 文件中的 `ContextImpl` 类如何使用 `registerService()` 来声明新 manager 的。特别可以看一下 `POWER_SERVICE` 是怎么做的。你可以很容易采纳上面的代码来组合成 `getSystemService()` 要用的 `PowerManager` 对象。

---

现在，我们可以通过这个 AOSP 构建一个新的 SDK，然后创建一个 app 来像调用其他预定义服务一样调用这个新的系统服务：

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    OpersysManager om = (OpersysManager) getSystemService(OPERSYS_SERVICE);
    Log.d(DTAG, "Going to write to the \"opersys\" service");
    om.write("Hello Opersys");
    Log.d(DTAG, "Service returned: " + om.read(20));
}
```

## 启动系统服务

这个例子中还有最后一点我还没有介绍，系统服务是如何启动的。就如我在第 7 章所提到的，基于 Java 的系统服务是在 `SystemServer.java` 中启动的。所以，我们可以修改这个文件来实例化我们的系统服务并向 Service Manager 注册：

```
...
try {
    Slog.i(TAG, "DiskStats Service");
    ServiceManager.addService("diskstats",
        new DiskStatsService(context));
} catch (Throwable e) {
    Slog.e(TAG, "Failure starting DiskStats Service", e);
}
```

```
try {
    Slog.i(TAG, "Opersys Service");
    ServiceManager.addService(Context.OPERSYS_SERVICE,
                               new OpersysService(context));
} catch (Throwable e) {
    Slog.e(TAG, "Failure starting OpersysService Service", e);
}
...
...
```

## 建议及说明

上面为 AOSP 添加新类型硬件的方法和我给出的代码仅仅就是能工作而已。因为你需要修改一些文件，所以它是依赖于版本的。前面的这些为 AOSP 添加新硬件类型的代码本质上是假设它是稳定不变的。而通常来说，这并不适用于你的情况，所以正如我在第 4 章所建议的，定制化的扩展最好是添加到 `device/<manufacturer/product-name>/` 目录，这样你就可以把它复制到任意的 AOSP 代码树里面去。

尽管有它的缺点，这个方法的优势在于你可以把 AOSP 中大量现存的其他系统和 HAL 模块作为例子，并且可以很简单的复制这些代码来用，因为你所增加的代码位置跟其他内建组件是一样的。

要使得新类型的硬件对 app 和其他系统服务可用，还有些方法是把系统服务添加到产品相关的目录 `device/` 中。最直接的办法是创建一个 app 并且将它的 manifest 文件中的 `persistent` 标志设置为 `true`。我们前面说过，app 的生命周期是由 Activity Manager 管理的，所以在普通 app 中实现硬件支持会有问题，因为它可能在任意停止或重启。如果软件需要维护硬件的一些状态信息，这样的重启就会导致出现问题。设置 `persistent` 标志可以禁止 Activity Manager 对这样应用做生命周期管理。正如我在第 7 章所述，例如电话应用就是采用这个方法来提供电话服务的。

这个方法的缺点是 Activity Manager 的宿主程序 System Server 如果崩溃了，会把你的系统服务同时宕掉。我前面说的方法也有这个问题。另外，更实质的问题在于这个方法的例子很少。你同样不能使用官方 SDK，而需要使用前面介绍的方法对 AOSP 修改后创建新的 SDK。因为调用者不能使用标准的 `getSystemService()` 函数来取得跟系统服务通信的句柄，而默认系统服务是可以的。

你还可以采用 Java 创建一个独立的系统服务，采用 `app_process` 来启动，跟 `am` 和 `pm` 一样。这就可以使得你的系统服务跟 System Server 无关，它重启也不会对你产生影响。但是我目前还不能给你一个采用这个方法实现系统服务的例子。而且，即使你采用这个方法，对应用程序开发者而言，你的系统服务跟其他系统服务看起来就不一样。

最后，你还可以使用 C 这样的语言创建本地系统服务，跟 *mediaserver*一样的方式来启动。这种情况下，虽然好处是程序直接运行在 CPU 上，但是你就不能像 Java 一样使用 *aidl* 这样的工具来产生封装和解封装代码。相反，与 Binder 通信的所有数据你都只能手工进行封装和解封装，这是相当繁琐的过程。而且，你的系统服务同样跟标准系统服务看起来很不一样。

## 附录 C

# 默认包列表的定制

在第 4 章中我们看到，可以修改构建系统来增加增加一个新的包到构建过程中。但是我们在那一章中没有讲到构建系统在创建镜像时如何创建默认的包列表，以及如何对这个列表进行定制。很显然，修改默认包列表这样的基础 AOSP 功能是有很大风险的，你很可能弄完后产生一个不能用的镜像。不过，在这里我们仍然值得看一看修改的方法。将来如果有需要的时候，你也可以知道应该修改哪里。

## 整体依赖关系

在 2.3/ 姜饼系统中，两个主要的变量决定了 AOSP 中包含哪些内容：GRANDFATHERED\_USER\_MODULES 和 PRODUCT\_PACKAGES。前者是产生自 *build/core/user\_tags.mk* 的列表，它包含了 AOSP 中需要的大量核心包，例如 *abdb*、*system service* 和 *Bionic*。这个文件不能修改，文件开头就是这样的警告信息：

```
# This is the list of modules grandfathered to use a user tag  
# DO NOT ADD ANY NEW MODULE TO THIS FILE  
#  
# user modules are hard to control and audit and we don't want  
# to add any new such module in the system
```

实际上，GRANDFATHERED\_USER\_MODULES 中的包列表或多或少是固定的，我们需要关注的是添加到 PRODUCT\_PACKAGES 的包。实际上有一系列的文件帮助你添加更多的包到 PRODUCT\_PACKAGES 中，一个个完整的 *.mk* 文件列表被包含到这个变量中，而产品相关的文件则在 *device/<vendor>/<product>/* 中。

在 4.2/ 果冻豆中，`GRANDFATHERED_USER_MODULES` 和 `build/core/user_tags.mk` 都不存在了。相反，有一个经过大量裁剪的 `GRANDFATHERED_ALL_PREBUILT`，以及跟包含前面警告信息的 `build/core/legacy_prebuilt.mk`。2.3/GingerBread 的 `GRANDFATHERED_USER_MODULES` 变量中的大部分放在了 `build/target/product/base.mk`，或者放到了 `build/target/product/core.mk`，最后都被添加到了 `PRODUCT_PACKAGES` 变量中，用法跟 2.3/Gingerbread 中一样。

## 组装 PRODUCT\_PACKAGES

通常来讲，多数产品会使用 `inherit-product` 工具来导入之前 `PRODUCT_PACKAGES` 变量声明的其他 `.mk` 文件。

`PRODUCT_PACKAGES` 用到的核心文件是 `build/target/product/core.mk`。在 2.3/ 姜饼中，该文件不从任何 `.mk` 文件继承。然而在 4.2/ 果冻豆中，它继承自 `build/target/product/base.mk`。不管在哪个版本中，`build/target/product/core.mk` 都包含了 SSL 库和浏览器应用。除了用于构建 SDK 的其他大多数产品描述实际上都不会仅仅依赖于这个文件定义的包集合。相反，在 2.3/ 姜饼中至少还会依赖 `build/target/product/generic.mk`，而在 4.2/ 果冻豆中则会依赖 `build/target/product/generic_no_telephony.mk`，两个版本都会依赖 `core.mk` 来包括其他很多 app，例如 `Calendar`（日历）、`Launcher2`（桌面）和 `Settings`（设置）。例如，2.3/ 姜饼默认的模拟器就会依赖 `generic.mk`，TI 为 BeagleBone 提供的 AOSP 代码树也是一样，本书中我用到的部分代码就是它的。

不过，大多数产品会有更进一步。在 2.3/ 姜饼中，它们会使用 `build/target/product/full.mk` 来加入输入法（例如拼音输入法）和一些语言，该文件依赖于 `generic.mk`。例如，`full.mk` 就是 `device/samsung/crespo/`（Nexus S）的基础，我在第 4 章的 CoyotePad 中就是用的它。

在 4.2/ 果冻豆中，大多数产品会使用 `build/target/product/full_base.mk` 而不是 `build/target/product/full.mk`。前者基于 `generic_no_telephony.mk` 而不是 `generic.mk`。你可以在 `device/asus/groupie/` 和 `device/samsung/tuna/` 中看到 `full_base.mk` 的用法。

## 包的裁剪

我经常被开发者问到的问题是如何裁剪 AOSP 的大小。要解决这个问题，在 2.3/ 姜饼系统中你应该遍历一遍 `GRANDFATHERED_USER_MODULES` 中包含的包列表，当然在 4.2/ 果冻豆系统中应该是 `GRANDFATHERED_ALL_PREBUILT`，以及 `PRODUCT_PACKAGES`，看看这两个列表中有没有在你的系统中不需要的包。正如我前面所述，这是一个危险的动

作，因为你很可能产生一个无法使用的 AOSP。实际上，AOSP 的构建系统没有提供任何类型的包依赖检测工具。

当然，有一些基本的常识你可以遵守。通常来说，我不建议你修改 GRANDFATHERED 包以及 4.2/ 果冻豆中的 *base.mk*，除非你觉得你对 AOSP 内部机制的理解以及对你修改的结果很有信心。从 *core.mk* 开始，相对来说是更安全的。然后你可以根据 *core.mk* 中的包的依赖关系移除不需要的包，请确保移除的包不会导致 AOSP 崩溃。例如，你可以在 2.3/ 姜饼的 *generic.mk* 中（或 4.2/ 果冻豆的 *generic\_no\_telephony.mk* 中）移除 Launcher2，你仍然可以得到一个可用的 AOSP 包。你就不会有你习惯的那个 HOME 屏幕，但依然可以正常工作的。这个文件中的其他很多应用也是一样。

# 默认的 init.rc 文件

这个附录包含了 2.3/ 姜饼和 4.2/ 果冻豆中的默认 *init.rc* 文件<sup>注 1</sup>。我不喜欢在书里放 大段的文件，你也很少在我的书中看到。但是，*init.rc* 是向你解释一些东西的最好的方法。为了让你更轻松地按照文件中进行的操作，我增加了一些标注，提供了关键部分见解。用于 *init.rc* 的有关动作、触发器、命令、服务和服务选项的更多详细信息请 参见第 6 章。

## 2.3/Gingerbread 默认 init.rc

```
on early-init ❶
    start ueventd

on init ❷
    sysclk tz 0
    loglevel 3

# setup the global environment ❸
    export PATH /sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin
    export LD_LIBRARY_PATH /vendor/lib:/system/lib
    export ANDROID_BOOTLOGO 1
    export ANDROID_ROOT /system
    export ANDROID_ASSETS /system/app
    export ANDROID_DATA /data
    export EXTERNAL_STORAGE /mnt/sdcard
    export ASEC_MOUNTPOINT /mnt/asec
    export LOOP_MOUNTPOINT /mnt/obb
```

---

注 1： 两个文件都是 AOSP 源代码的一部分，因此也是 Apache 协议授权的。

```
export BOOTCLASSPATH /system/framework/core.jar:/system/framework/bouncycastle.jar:/system/framework/ext.jar:/system/framework/framework.jar:/system/framework/android.policy.jar:/system/framework/services.jar:/system/framework/core-junit.jar

# Backward compatibility
symlink /system/etc /etc
symlink /sys/kernel/debug /d

# Right now vendor lives on the same filesystem as system,
# but someday that may change.
symlink /system/vendor /vendor

# create mountpoints
mkdir /mnt 0775 root system
mkdir /mnt/sdcard 0000 system system

# Create cgroup mount point for cpu accounting
mkdir /acct
mount cgroup none /acct cpacct
mkdir /acct/uid

# Backwards Compat - XXX: Going away in G*
symlink /mnt/sdcard /sdcard

mkdir /system
mkdir /data 0771 system system
mkdir /cache 0770 system cache
mkdir /config 0500 root root

# Directory for putting things only root should see.
mkdir /mnt/secure 0700 root root

# Directory for staging bindmounts
mkdir /mnt/secure/staging 0700 root root

# Directory-target for where the secure container
# imagefile directory will be bind-mounted
mkdir /mnt/secure/asec 0700 root root

# Secure container public mount points.
mkdir /mnt/asec 0700 root system
mount tmpfs tmpfs /mnt/asec mode=0755,gid=1000

# Filesystem image public mount points.
mkdir /mnt/obb 0700 root system
mount tmpfs tmpfs /mnt/obb mode=0755,gid=1000

write /proc/sys/kernel/panic_on_oops 1 ❸
write /proc/sys/kernel/hung_task_timeout_secs 0
write /proc/cpu/alignment 4

write /proc/sys/kernel/sched_latency_ns 10000000
write /proc/sys/kernel/sched_wakeup_granularity_ns 2000000
write /proc/sys/kernel/sched_compat_yield 1
write /proc/sys/kernel/sched_child_runs_first 0
```

```

# Create cgroup mount points for process groups
mkdir /dev/cpuctl
mount cgroup none /dev/cpuctl cpu
chown system system /dev/cpuctl
chown system system /dev/cpuctl/tasks
chmod 0777 /dev/cpuctl/tasks
write /dev/cpuctl/cpu.shares 1024

mkdir /dev/cpuctl/fg_boost
chown system system /dev/cpuctl/fg_boost/tasks
chmod 0777 /dev/cpuctl/fg_boost/tasks
write /dev/cpuctl/fg_boost/cpu.shares 1024

mkdir /dev/cpuctl/bg_non_interactive
chown system system /dev/cpuctl/bg_non_interactive/tasks
chmod 0777 /dev/cpuctl/bg_non_interactive/tasks
# 5.0 %
write /dev/cpuctl/bg_non_interactive/cpu.shares 52

on fs 5
# mount mtd partitions
    # Mount /system rw first to give the filesystem a chance to save a checkpoint
    mount yaffs2 mtd@system /system
    mount yaffs2 mtd@system /system ro remount
    mount yaffs2 mtd@userdata /data nosuid nodev
    mount yaffs2 mtd@cache /cache nosuid nodev

on post-fs 6
    # once everything is setup, no need to modify /
    mount rootfs rootfs / ro remount

    # We chown/chmod /data again so because mount is run as root + defaults
    chown system system /data
    chmod 0771 /data

    # Create dump dir and collect dumps.
    # Do this before we mount cache so eventually we can use cache for
    # storing dumps on platforms which do not have a dedicated dump partition.

    mkdir /data/dontpanic
    chown root log /data/dontpanic
    chmod 0750 /data/dontpanic

    # Collect apanic data, free resources and re-arm trigger
    copy /proc/panic_console /data/dontpanic/panic_console
    chown root log /data/dontpanic/panic_console
    chmod 0640 /data/dontpanic/panic_console

    copy /proc/panic_threads /data/dontpanic/panic_threads
    chown root log /data/dontpanic/panic_threads
    chmod 0640 /data/dontpanic/panic_threads

    write /proc/panic_console 1

```

```
# Same reason as /data above
chown system cache /cache
chmod 0770 /cache

# This may have been created by the recovery system with odd permissions
chown system cache /cache/recovery
chmod 0770 /cache/recovery

#change permissions on vmallocinfo so we can grab it from bugreports
chown root log /proc/vmallocinfo
chmod 0440 /proc/vmallocinfo

#change permissions on kmsg & sysrq-trigger so bugreports can grab kthread
stacks
chown root system /proc/kmsg
chmod 0440 /proc/kmsg
chown root system /proc/sysrq-trigger
chmod 0220 /proc/sysrq-trigger

# create basic filesystem structure
mkdir /data/misc 01771 system misc
mkdir /data/misc/bluetoothd 0770 bluetooth bluetooth
mkdir /data/misc/bluetooth 0770 system system
mkdir /data/misc/keystore 0700 keystore keystore
mkdir /data/misc/vpn 0770 system system
mkdir /data/misc/systemkeys 0700 system system
mkdir /data/misc/vpn/profiles 0770 system system
# give system access to wpa_supplicant.conf for backup and restore
mkdir /data/misc/wifi 0770 wifi wifi
chmod 0770 /data/misc/wifi
chmod 0660 /data/misc/wifi/wpa_supplicant.conf
mkdir /data/local 0771 shell shell
mkdir /data/local/tmp 0771 shell shell
mkdir /data/data 0771 system system
mkdir /data/app-private 0771 system system
mkdir /data/app 0771 system system
mkdir /data/property 0700 root root

# create dalvik-cache and double-check the perms
mkdir /data/dalvik-cache 0771 system system
chown system system /data/dalvik-cache
chmod 0771 /data/dalvik-cache

# create the lost+found directories, so as to enforce our permissions
mkdir /data/lost+found 0770
mkdir /cache/lost+found 0770

# double check the perms, in case lost+found already exists, and set owner
chown root root /data/lost+found
chmod 0770 /data/lost+found
chown root root /cache/lost+found
chmod 0770 /cache/lost+found

on boot ⑦
# basic network init
```

```
ifup lo
hostname localhost
domainname localdomain

# set RLIMIT_NICE to allow priorities from 19 to -20
setrlimit 13 40 40

# Define the oom_adj values for the classes of processes that can be
# killed by the kernel. These are used in ActivityManagerService.
setprop ro.FOREGROUND_APP_ADJ 0
setprop ro.VISIBLE_APP_ADJ 1
setprop ro.PERCEPTEABLE_APP_ADJ 2
setprop ro.HEAVY_WEIGHT_APP_ADJ 3
setprop ro.SECONDARY_SERVER_APP_ADJ 4
setprop ro.BACKUP_APP_ADJ 5
setprop ro.HOME_APP_ADJ 6
setprop ro.HIDDEN_APP_MIN_ADJ 7
setprop ro.EMPTY_APP_ADJ 15

# Define the memory thresholds at which the above process classes will
# be killed. These numbers are in pages (4k).
setprop ro.FOREGROUND_APP_MEM 2048
setprop ro.VISIBLE_APP_MEM 3072
setprop ro.PERCEPTEABLE_APP_MEM 4096
setprop ro.HEAVY_WEIGHT_APP_MEM 4096
setprop ro.SECONDARY_SERVER_MEM 6144
setprop ro.BACKUP_APP_MEM 6144
setprop ro.HOME_APP_MEM 6144
setprop ro.HIDDEN_APP_MEM 7168
setprop ro.EMPTY_APP_MEM 8192

# Write value must be consistent with the above properties. ❸
# Note that the driver only supports 6 slots, so we have combined some of
# the classes into the same memory level; the associated processes of higher
# classes will still be killed first.
write /sys/module/lowmemorykiller/parameters/adj 0,1,2,4,7,15

write /proc/sys/vm/overcommit_memory 1
write /proc/sys/vm/min_free_order_shift 4
write /sys/module/lowmemorykiller/parameters/minfree 2048,3072,4096,6144,
7168,8192

# Set init its forked children's oom_adj.
write /proc/1/oom_adj -16

# Tweak background writeout
write /proc/sys/vm/dirty_expire_centisecs 200
write /proc/sys/vm/dirty_background_ratio 5

# Permissions for System Server and daemons.
chown radio system /sys/android_power/state
chown radio system /sys/android_power/request_state
chown radio system /sys/android_power/acquire_full_wake_lock
chown radio system /sys/android_power/acquire_partial_wake_lock
chown radio system /sys/android_power/release_wake_lock
```

```
chown radio system /sys/power/state
chown radio system /sys/power/wake_lock
chown radio system /sys/power/wake_unlock
chmod 0660 /sys/power/state
chmod 0660 /sys/power/wake_lock
chmod 0660 /sys/power/wake_unlock
chown system system /sys/class/timed_output/vibrator/enable
chown system system /sys/class/leds/keyboard-backlight/brightness
chown system system /sys/class/leds/lcd-backlight/brightness
chown system system /sys/class/leds/button-backlight/brightness
chown system system /sys/class/leds/jogball-backlight/brightness
chown system system /sys/class/leds/red/brightness
chown system system /sys/class/leds/green/brightness
chown system system /sys/class/leds/blue/brightness
chown system system /sys/class/leds/red/device/grpfreq
chown system system /sys/class/leds/red/device/grppwm
chown system system /sys/class/leds/red/device/blink
chown system system /sys/class/leds/red/brightness
chown system system /sys/class/leds/green/brightness
chown system system /sys/class/leds/blue/brightness
chown system system /sys/class/leds/red/device/grpfreq
chown system system /sys/class/leds/red/device/grppwm
chown system system /sys/class/leds/red/device/blink
chown system system /sys/class/timed_output/vibrator/enable
chown system system /sys/module/sco/parameters/disable_esc0
chown system system /sys/kernel/ipv4/tcp_wmem_min
chown system system /sys/kernel/ipv4/tcp_wmem_def
chown system system /sys/kernel/ipv4/tcp_wmem_max
chown system system /sys/kernel/ipv4/tcp_rmem_min
chown system system /sys/kernel/ipv4/tcp_rmem_def
chown system system /sys/kernel/ipv4/tcp_rmem_max
chown root radio /proc/cmdline
```

```
# Define TCP buffer sizes for various networks
# ReadMin, ReadInitial, ReadMax, WriteMin, WriteInitial, WriteMax,
    setprop net.tcp.buffersize.default 4096,87380,110208,4096,16384,110208
    setprop net.tcp.buffersize.wifi     4095,87380,110208,4096,16384,110208
    setprop net.tcp.buffersize.umts    4094,87380,110208,4096,16384,110208
    setprop net.tcp.buffersize.edge   4093,26280,35040,4096,16384,35040
    setprop net.tcp.buffersize.gprs   4092,8760,11680,4096,8760,11680

    class_start default ❾

## Daemon processes to be run by init. ❿
##  

service ueventd /sbin/ueventd
    critical

service console /system/bin/sh
    console
    disabled
    user shell
    group log
```

```
on property:ro.secure=0
    start console

# abd is controlled by the persist.service.adb.enable system property ⑪
service abd /sbin/abd
disabled

# abd on at boot in emulator
on property:ro.kernel.qemu=1
    start abd

on property:persist.service.adb.enable=1
    start abd

on property:persist.service.adb.enable=0
    stop abd

service servicemanager /system/bin/servicemanager ⑫
    user system
    critical
    onrestart restart zygote
    onrestart restart media

service vold /system/bin/vold
    socket vold stream 0660 root mount
    ioprio be 2

service netd /system/bin/netd
    socket netd stream 0660 root system

service debuggerd /system/bin/debuggerd

service ril-daemon /system/bin/rild
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc audio sdcard_rw

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-
server ⑬
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd

service media /system/bin/mediaserver ⑭
    user media
    group system audio camera graphics inet net_bt net_bt_admin net_raw
    ioprio rt 4

service bootanim /system/bin/bootanimation
    user graphics
    group graphics
    disabled
    oneshot
```

```
service dbus /system/bin/dbus-daemon --system --nofork
    socket dbus stream 660 bluetooth bluetooth
    user bluetooth
    group bluetooth net_bt_admin

service bluetoothd /system/bin/bluetoothd -n
    socket bluetooth stream 660 bluetooth bluetooth
    socket dbus_bluetooth stream 660 bluetooth bluetooth
    # init.rc does not yet support applying capabilities, so run as root and
    # let bluetoothd drop uid to bluetooth with the right linux capabilities
    group bluetooth net_bt_admin misc
    disabled

service hfag /system/bin/sdptool add --channel=10 HFAG
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service hsag /system/bin/sdptool add --channel=11 HSAG
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service opush /system/bin/sdptool add --channel=12 OPUSH
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service pbap /system/bin/sdptool add --channel=19 PBAP
    user bluetooth
    group bluetooth net_bt_admin
    disabled
    oneshot

service installd /system/bin/installld
    socket installd stream 600 system system
    service flash_recovery /system/etc/install-recovery.sh
    oneshot

service racoon /system/bin/racoon
    socket racoon stream 600 system system
    # racoon will setuid to vpn after getting necessary resources.
    group net_admin
    disabled
    oneshot

service mtpd /system/bin/mtpd
    socket mtpd stream 600 system system
    user vpn
    group vpn net_admin net_raw
    disabled
    oneshot
```

```
service keystore /system/bin/keystore /data/misc/keystore
    user keystore
    group keystore
    socket keystore stream 666

service dumpstate /system/bin/dumpstate -s ⑯
    socket dumpstate stream 0660 shell log
    disabled
    oneshot
```

- ❶ 正如第 6 章所述, *init* 按照触发器、动作列表执行动作, *early-init* 是最早执行的部分。正如你所看到的, 这里只运行了 *ueventd*。实际上, *init* 在初始化过程中的下一个步骤就会检查 *ueventd* 是否被正常启动了。
- ❷ *init* 运行的第一个大段代码是 *init* 动作, 它把时区设置为 GMT (格林尼治标准时间), 把日志等级设置为 3 级<sup>注 2</sup>, 配置核心环境变量, 然后对根文件系统做一系列操作。
- ❸ 这个部分的初始化相当重要。这里设置了所有二进制程序的默认路径 PATH, 动态链接器的默认搜索路径——LD\_LIBRARY\_PATH, 也在那里进行了配置。注意, /bin 没有在 PATH 中, /lib 也不在 LD\_LIBRARY\_PATH 里面。
- ❹ 在这里, 通过写入值到 /proc 中的条目的方式调整一些内核的参数。这就跟将值写入到 /sys 目录条目控制内核和驱动程序一样是常用方法。
- ❺ *fs* 动作就是挂载 /system、/data 和 /cache 等分区。这个默认的 config 文件是采用 YAFFS2 文件系统挂载 MTD 分区, 而你的板子可能既没有 MTD 分区也不使用 YAFFS, 这种情况下这些命令就会失败。不过这并没有问题, 因为并没有阻止你在你板级的 .rc 文件中包含一个 *fs* 动作来在你分区上挂载其他文件系统。
- ❻ *post-fs* 动作是所有依赖的文件系统已被挂载, 而且文件系统命令已被正确执行时触发。这里, 同样是大量的文件系统操作被触发的执行了。
- ❼ *boot* 动作是在所有文件系统已经被正确设置时执行了, 该动作执行完成后, 所有服务将会被启动。这一节设置基本网络功能、配置 OOM 参数、Activity 管理器和内核使用的内存阈值, 允许系统服务访问 /sys 里的目录项, 设置网络属性, 最后启动所有默认服务。
- ❽ 这些 /proc 和 /sys 操作是用户空间配置低内存驱动参数的方法。
- ❾ 这个看似无用的命令其实是这个文件中最重要的命令。这个文件后面的服务其实都是被这个命令启动的。实际上, .rc 文件中声明的服务都配置为其默认选项,

---

注 2: 想了解这个动作的细节信息, 请参见 klogctl 的 manpage 手册。

除非在 service 描述中配置了特殊的 class 选项。因为这个文件中列出的服务都没有 class 选项，而是自 class\_start 开始使用默认 class。

- ⑩ 到此时，大部分的动作已经被定义好了，这个文件剩下的部分主要在于描述需要运行的服务。由于这些都属于默认类别，它们会以文件中定义的次序启动。
- ⑪ 注意这里在启动时将 abdb 设置为禁止，除非 persist.service.adb.enable 属性被设置为 1。
- ⑫ 这是至关重要的 Service Manager，在第 2 章中也已介绍过。注意它被标注为 critical，它的重启会导致 System Server 和 Media Server 重新启动。
- ⑬ 这是 Zygote，在第 2 章中也介绍了。这里可以留意一下二进制程序是如何被 app\_process 启动的。app\_process 实际上是基于 C 开发的二进制程序，用于启动 Davik 虚拟机实例，Zygote 的 Java 类就从这里被启动。而在这里，System Server 被 Zygote 启动。
- ⑭ 这是配置媒体服务。可以留意一下如何设置 I/O 的 nice 值以模拟“实时”调度器以及如何设置优先级。
- ⑮ 这个 dumpstate 对于工具箱的 bugreport 命令正确处理是很关键的。想更详细了解，建议看看第 6 章关于 bugreport 与 dumpstate 操作的更多信息。

## 4.2/ 果冻豆默认 init 文件

与 2.3/ 姜饼系统有所不同，4.2/ 果冻豆有三个主要的 rc 文件：init.rc、init.usb.rc 和 init.trace.rc 文件。我们来看一看这几个文件：

### init.rc 文件

这里是 4.2/ 果冻豆系统中的主 init.rc 文件。正如你所看到的，跟 2.3/ 姜饼系统相比很多重要的部分都没变，不过在这个新版本中却出现了一些新的东西需要专门再说一下。

---

**注意：**即使你采用 4.2/ 果冻豆系统，我也仍然建议你先读一下上节中 2.3/ 姜饼系统的 init.rc 文件，我不会把前面介绍过的内容再做重复介绍。

---

```
# Copyright (C) 2012 The Android Open Source Project
#
# IMPORTANT: Do not create world writable files or directories.
# This is a common source of Android security bugs.
#
```

```
import /init.usb.rc ①
import /init.${ro.hardware}.rc
import /init.trace.rc

on early-init
    # Set init and its forked children's oom_adj.
    write /proc/1/oom_adj -16

    # Set the security context for the init process.
    # This should occur before anything else (e.g. ueventd) is started.
    setcon u:r:init:so ②

    start ueventd

# create mountpoints
    mkdir /mnt 0775 root system

on init
    sysclk tz 0
    loglevel 3

# setup the global environment
    export PATH /sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin
    export LD_LIBRARY_PATH /vendor/lib:/system/lib
    export ANDROID_BOOTLOGO 1
    export ANDROID_ROOT /system
    export ANDROID_ASSETS /system/app
    export ANDROID_DATA /data
    export ANDROID_STORAGE /storage
    export ASEC_MOUNTPOINT /mnt/asec
    export LOOP_MOUNTPOINT /mnt/obb
    export BOOTCLASSPATH /system/framework/core.jar:/system/framework/core-junit/
.jar:/system/framework/bouncycastle.jar:/system/framework/ext.jar:/system/framework/
framework.jar:/system/framework/telephony-common.jar:/system/framework/mms-c ommon.
jar:/system/framework/android.policy.jar:/system/framework/services.jar:/s ystem/
framework/apache-xml.jar

# Backward compatibility
    symlink /system/etc /etc
    symlink /sys/kernel/debug /d

# Right now vendor lives on the same filesystem as system,
# but someday that may change.
    symlink /system/vendor /vendor

# Create cgroup mount point for cpu accounting
    mkdir /acct
    mount cgroup none /acct cpualct
    mkdir /acct/uid

    mkdir /system
    mkdir /data 0771 system system
    mkdir /cache 0770 system cache
    mkdir /config 0500 root root
```

```
# See storage config details at http://source.android.com/tech/storage/
mkdir /mnt/shell 0700 shell shell
mkdir /storage 0050 root sdcard_r

# Directory for putting things only root should see.
mkdir /mnt/secure 0700 root root
# Create private mountpoint so we can MS_MOVE from staging
mount tmpfs /mnt/secure mode=0700,uid=0,gid=0

# Directory for staging bindmounts
mkdir /mnt/secure/staging 0700 root root

# Directory-target for where the secure container
# imagefile directory will be bind-mounted
mkdir /mnt/secure/asec 0700 root root

# Secure container public mount points.
mkdir /mnt/asec 0700 root system
mount tmpfs /mnt/asec mode=0755,gid=1000

# Filesystem image public mount points.
mkdir /mnt/obb 0700 root system
mount tmpfs /mnt/obb mode=0755,gid=1000

write /proc/sys/kernel/panic_on_oops 1
write /proc/sys/kernel/hung_task_timeout_secs 0
write /proc/cpu/alignment 4
write /proc/sys/kernel/sched_latency_ns 10000000
write /proc/sys/kernel/sched_wakeup_granularity_ns 2000000
write /proc/sys/kernel/sched_compat_yield 1
write /proc/sys/kernel/sched_child_runs_first 0
write /proc/sys/kernel/randomize_va_space 2
write /proc/sys/kernel/kptr_restrict 2
write /proc/sys/kernel/dmesg_restrict 1
write /proc/sys/vm/mmap_min_addr 32768
write /proc/sys/kernel/sched_rt_runtime_us 950000
write /proc/sys/kernel/sched_rt_period_us 1000000

# Create cgroup mount points for process groups
mkdir /dev/cpuctl
mount cgroup none /dev/cpuctl cpu
chown system system /dev/cpuctl
chown system system /dev/cpuctl/tasks
chmod 0660 /dev/cpuctl/tasks
write /dev/cpuctl/cpu.shares 1024
write /dev/cpuctl/cpu.rt_runtime_us 950000
write /dev/cpuctl/cpu.rt_period_us 1000000
mkdir /dev/cpuctl/apps
chown system system /dev/cpuctl/apps/tasks
chmod 0666 /dev/cpuctl/apps/tasks
write /dev/cpuctl/apps/cpu.shares 1024
write /dev/cpuctl/apps/cpu.rt_runtime_us 800000
write /dev/cpuctl/apps/cpu.rt_period_us 1000000

mkdir /dev/cpuctl/apps/bg_non_interactive
chown system system /dev/cpuctl/apps/bg_non_interactive/tasks
```

```
chmod 0666 /dev/cpuctl/apps/bg_non_interactive/tasks
# 5.0 %
write /dev/cpuctl/apps/bg_non_interactive/cpu.shares 52
write /dev/cpuctl/apps/bg_non_interactive/cpu.rt_runtime_us 700000
write /dev/cpuctl/apps/bg_non_interactive/cpu.rt_period_us 1000000

# Allow everybody to read the xt_qtaguid resource tracking misc dev.
# This is needed by any process that uses socket tagging.
chmod 0644 /dev/xt_qtaguid

on fs
# mount mtd partitions
# Mount /system rw first to give the filesystem a chance to save a      checkpoint
mount yaffs2 mtd@system /system
mount yaffs2 mtd@system /system ro remount
mount yaffs2 mtd@userdata /data nosuid nodev
mount yaffs2 mtd@cache /cache nosuid nodev

on post-fs
# once everything is setup, no need to modify /
mount rootfs rootfs / ro remount
# mount shared so changes propagate into child namespaces
mount rootfs rootfs / shared rec
mount tmpfs tmpfs /mnt/secure private rec

# We chown/chmod /cache again so because mount is run as root + defaults
chown system cache /cache
chmod 0770 /cache
# We restorecon /cache in case the cache partition has been reset.
restorecon /cache

# This may have been created by the recovery system with odd permissions
chown system cache /cache/recovery
chmod 0770 /cache/recovery

# This may have been created by the recovery system with the wrong context.
restorecon /cache/recovery

#change permissions on vmallocinfo so we can grab it from bugreports
chown root log /proc/vmallocinfo
chmod 0440 /proc/vmallocinfo
chown root log /proc/slabinfo
chmod 0440 /proc/slabinfo

#change permissions on kmsg & sysrq-trigger so bugreports can grab kthread
stacks
chown root system /proc/kmsg
chmod 0440 /proc/kmsg
chown root system /proc/sysrq-trigger
chmod 0220 /proc/sysrq-trigger
chown system log /proc/last_kmsg
chmod 0440 /proc/last_kmsg

# create the lost+found directories, so as to enforce our permissions
mkdir /cache/lost+found 0770 root root
```

```
on post-fs-data
    # We chown/chmod /data again so because mount is run as root + defaults
    chown system system /data
    chmod 0771 /data
    # We restorecon /data in case the userdata partition has been reset.
    restorecon /data

    # Create dump dir and collect dumps.
    # Do this before we mount cache so eventually we can use cache for
    # storing dumps on platforms which do not have a dedicated dump partition.
    mkdir /data/dontpanic 0750 root log

    # Collect apanic data, free resources and re-arm trigger
    copy /proc/panic_console /data/dontpanic/panic_console
    chown root log /data/dontpanic/panic_console
    chmod 0640 /data/dontpanic/panic_console

    copy /proc/panic_threads /data/dontpanic/panic_threads
    chown root log /data/dontpanic/panic_threads
    chmod 0640 /data/dontpanic/panic_threads

    write /proc/panic_console 1

    # create basic filesystem structure
    mkdir /data/misc 01771 system misc
    mkdir /data/misc/adb 02750 system shell
    mkdir /data/misc/bluedroid 0770 bluetooth net_bt_stack
    mkdir /data/misc/bluetooth 0770 system system
    mkdir /data/misc/keystore 0700 keystore keystore
    mkdir /data/misc/keychain 0771 system system
    mkdir /data/misc/sms 0770 system radio
    mkdir /data/misc/vpn 0770 system vpn
    mkdir /data/misc/systemkeys 0700 system system
    # give system access to wpa_supplicant.conf for backup and restore
    mkdir /data/misc/wifi 0770 wifi wifi
    chmod 0660 /data/misc/wifi/wpa_supplicant.conf
    mkdir /data/local 0751 root root

    # For security reasons, /data/local/tmp should always be empty.
    # Do not place files or directories in /data/local/tmp
    mkdir /data/local/tmp 0771 shell shell
    mkdir /data/data 0771 system system
    mkdir /data/app-private 0771 system system
    mkdir /data/app-asec 0700 root root
    mkdir /data/app-lib 0771 system system
    mkdir /data/app 0771 system system
    mkdir /data/property 0700 root root
    mkdir /data/ssh 0750 root shell
    mkdir /data/ssh/empty 0700 root root

    # create dalvik-cache, so as to enforce our permissions
    mkdir /data/dalvik-cache 0771 system system

    # create resource-cache and double-check the perms
    mkdir /data/resource-cache 0771 system system
```

```
chown system system /data/resource-cache
chmod 0771 /data/resource-cache

# create the lost+found directories, so as to enforce our permissions
mkdir /data/lost+found 0770 root root

# create directory for DRM plug-ins - give drm the read/write access to
# the following directory.
mkdir /data/drm 0770 drm drm

# If there is no fs-post-data action in the init.<device>.rc file, you
# must uncomment this line, otherwise encrypted filesystems
# won't work.
# Set indication (checked by vold) that we have finished this action
#setprop vold.post_fs_data_done 1

on boot
# basic network init
    ifup lo
    hostname localhost
    domainname localdomain

# set RLIMIT_NICE to allow priorities from 19 to -20
    setrlimit 13 40 40

# Memory management. Basic kernel parameters, and allow the high
# level system server to be able to adjust the kernel OOM driver
# parameters to match how it is managing things
    write /proc/sys/vm/overcommit_memory 1
    write /proc/sys/vm/min_free_order_shift 4
    chown root system /sys/module/lowmemorykiller/parameters/adj
    chmod 0664 /sys/module/lowmemorykiller/parameters/adj
    chown root system /sys/module/lowmemorykiller/parameters/minfree
    chmod 0664 /sys/module/lowmemorykiller/parameters/minfree

# Tweak background writeout
    write /proc/sys/vm/dirty_expire_centisecs 200
    write /proc/sys/vm/dirty_background_ratio 5

# Permissions for System Server and daemons.
    chown radio system /sys/android_power/state
    chown radio system /sys/android_power/request_state
    chown radio system /sys/android_power/acquire_full_wake_lock
    chown radio system /sys/android_power/acquire_partial_wake_lock
    chown radio system /sys/android_power/release_wake_lock
    chown system system /sys/power/autosleep
    chown system system /sys/power/state

    chown system system /sys/power/wakeup_count
    chown radio system /sys/power/wake_lock
    chown radio system /sys/power/wake_unlock
    chmod 0660 /sys/power/state
    chmod 0660 /sys/power/wake_lock
    chmod 0660 /sys/power/wake_unlock
```

```
chown system system /sys/devices/system/cpu/cpufreq/interactive/timer_rate
chmod 0660 /sys/devices/system/cpu/cpufreq/interactive/timer_rate
chown system system /sys/devices/system/cpu/cpufreq/interactive/min_sample_time
chmod 0660 /sys/devices/system/cpu/cpufreq/interactive/min_sample_time
chown system system /sys/devices/system/cpu/cpufreq/interactive/hispeed_freq
chmod 0660 /sys/devices/system/cpu/cpufreq/interactive/hispeed_freq
chown system system /sys/devices/system/cpu/cpufreq/interactive/go_hispeed_load
chmod 0660 /sys/devices/system/cpu/cpufreq/interactive/go_hispeed_load
chown system system /sys/devices/system/cpu/cpufreq/interactive/above_hispeed_delay
chmod 0660 /sys/devices/system/cpu/cpufreq/interactive/above_hispeed_delay
chown system system /sys/devices/system/cpu/cpufreq/interactive/boost
chmod 0660 /sys/devices/system/cpu/cpufreq/interactive/boost
chown system system /sys/devices/system/cpu/cpufreq/interactive/boostpulse
chown system system /sys/devices/system/cpu/cpufreq/interactive/input_boost
chmod 0660 /sys/devices/system/cpu/cpufreq/interactive/input_boost

# Assume SMP uses shared cpufreq policy for all CPUs
chown system system /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq
chmod 0660 /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq

chown system system /sys/class/timed_output/vibrator/enable
chown system system /sys/class/leds/keyboard-backlight/brightness
chown system system /sys/class/leds/lcd-backlight/brightness
chown system system /sys/class/leds/button-backlight/brightness
chown system system /sys/class/leds/jogball-backlight/brightness
chown system system /sys/class/leds/red/brightness
chown system system /sys/class/leds/green/brightness
chown system system /sys/class/leds/blue/brightness
chown system system /sys/class/leds/red/device/grpfreq
chown system system /sys/class/leds/red/device/grppwm
chown system system /sys/class/leds/red/device/blink
chown system system /sys/class/leds/red/brightness
chown system system /sys/class/leds/green/brightness
chown system system /sys/class/leds/blue/brightness
chown system system /sys/class/leds/red/device/grpfreq
chown system system /sys/class/leds/red/device/grppwm
chown system system /sys/class/leds/red/device/blink
chown system system /sys/class/timed_output/vibrator/enable
chown system system /sys/module/sco/parameters/disable_esco
chown system system /sys/kernel/ipv4/tcp_wmem_min
chown system system /sys/kernel/ipv4/tcp_wmem_def
chown system system /sys/kernel/ipv4/tcp_wmem_max
chown system system /sys/kernel/ipv4/tcp_rmem_min
chown system system /sys/kernel/ipv4/tcp_rmem_def
chown system system /sys/kernel/ipv4/tcp_rmem_max
chown root radio /proc/cmdline

# Define TCP buffer sizes for various networks
# ReadMin, ReadInitial, ReadMax, WriteMin, WriteInitial, WriteMax
setprop net.tcp.buffersize.default 4096,87380,110208,4096,16384,110208
setprop net.tcp.buffersize.wifi      524288,1048576,2097152,262144,524288,
                                    1048576
setprop net.tcp.buffersize.lte       524288,1048576,2097152,262144,524288,
                                    1048576
setprop net.tcp.buffersize.umts     4094,87380,110208,4096,16384,110208
```

```
setprop net.tcp.buffersize.hspa      4094,87380,262144,4096,16384,262144
setprop net.tcp.buffersize.hsupa     4094,87380,262144,4096,16384,262144
setprop net.tcp.buffersize.hsdpa     4094,87380,262144,4096,16384,262144
setprop net.tcp.buffersize.hspap     4094,87380,1220608,4096,16384,1220608
setprop net.tcp.buffersize.edge      4093,26280,35040,4096,16384,35040
setprop net.tcp.buffersize.gprs      4092,8760,11680,4096,8760,11680
setprop net.tcp.buffersize.evdo     4094,87380,262144,4096,16384,262144

# Set this property so surfaceflinger is not started by system_init
    setprop system_init.startsurfaceflinger 0

    class_start core 3
    class_start main

on nonencrypted
    class_start late_start

on charger
    class_start charger

on property:vold.decrypt=trigger_reset_main
    class_reset main

on property:vold.decrypt=trigger_load_persist_props
    load_persist_props

on property:vold.decrypt=trigger_post_fs_data
    trigger post-fs-data

on property:vold.decrypt=trigger_restart_min_framework
    class_start main

on property:vold.decrypt=trigger_restart_framework
    class_start main
    class_start late_start

on property:vold.decrypt=trigger_shutdown_framework
    class_reset late_start
    class_reset main

## Daemon processes to be run by init.
##
service ueventd /sbin/ueventd
    class core 4
    critical
    seclabel u:r:ueventd:so

on property:selinux.reload_policy=1
    restart ueventd
    restart installd

service console /system/bin/sh
    class core
    console
    disabled
```

```
user shell
group log

on property:ro.debuggable=1
    start console

# adbd is controlled via property triggers in init.<platform>.usb.rc ⑤
service adbd /sbin/adbd
    class core
    socket adbd stream 660 system system
    disabled
    seclabel u:r:adbd:s0

# adbd on at boot in emulator
on property:ro.kernel.qemu=1
    start adbd

service servicemanager /system/bin/servicemanager
    class core
    user system
    group system
    critical
    onrestart restart zygote
    onrestart restart media
    onrestart restart surfaceflinger
    onrestart restart drm

service vold /system/bin/vold
    class core
    socket vold stream 0660 root mount
    ioprio be 2

service netd /system/bin/netd
    class main ⑥
    socket netd stream 0660 root system
    socket dnsproxyd stream 0660 root inet
    socket mdns stream 0660 root system

service debuggerd /system/bin/debuggerd
    class main

service ril-daemon /system/bin/rild
    class main
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc audio log

service surfaceflinger /system/bin/surfaceflinger ⑦
    class main
    user system
    group graphics drmrpc
    onrestart restart zygote

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
```

```
class main
socket zygote stream 660 root system
onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd

service drm /system/bin/drmserver
    class main
    user drm
    group drm system inet drmrpc

service media /system/bin/mediaserver
    class main
    user media
    group audio camera inet net_bt net_bt_admin net_bw_acct drmrpc
    ioprio rt 4

service bootanim /system/bin/bootanimation
    class main
    user graphics
    group graphics
    disabled
    oneshot

service installd /system/bin/installld
    class main
    socket installd stream 600 system system

service flash_recovery /system/etc/install-recovery.sh
    class main
    oneshot

service racoon /system/bin/racoon
    class main
    socket racoon stream 600 system system
    # IKE uses UDP port 500. Racoon will setuid to vpn after binding the port.
    group vpn net_admin inet
    disabled
    oneshot

service mtpd /system/bin/mtpd
    class main
    socket mtpd stream 600 system system
    user vpn
    group vpn net_admin inet net_raw
    disabled
    oneshot

service keystore /system/bin/keystore /data/misc/keystore
    class main
    user keystore
    group keystore drmrpc
    socket keystore stream 666
```

```
service dumpstate /system/bin/dumpstate -s
    class main
    socket dumpstate stream 0660 shell log
    disabled
    oneshot

service sshd /system/bin/start-ssh
    class main
    disabled

service mdnsd /system/bin/mdnsd
    class main
    user mdnsr
    group inet net_raw
    socket mdnsd stream 0660 mdnsr inet
    disabled
    oneshot
```

- ❶ 4.2/ 果冻豆系统使用 `import` 机制来整合其他 `.rc` 文件。在这个例子中，导入了 3 个外部文件。因为 `init.usb.rc` 和 `init.trace.rc` 两个文件对所有设备来说是共有的，所以我在后面把它们放进来作为参考。`init.rc` 还根据 `ro.hardware` 全局属性来导入板级初始化文件 `init.${ro.hardware}.rc`，这个文件位于 `device` 目录中。
- ❷ 它跟 SEAndroid 项目相关，是新加入到 `init.rc` 的。关于 SEAndroid 的更多信息，可以看看 <http://selinuxproject.org/page/SEAndroid>。
- ❸ 在 2.3/ 姜饼系统的 `init.rc` 文件中，`class_start` 只用来启动服务的默认类，也就是默认同时启动 `init.rc` 文件中的所有服务。而在 4.2/ 果冻豆系统中，把文件中的服务分为了 `core` 和 `main` 两类，类别名称已经说明了其含义。在文件的后面部分你可以看到每个服务是如何被标识到相应类别的。
- ❹ 这是第一个采用 `class` 来指明服务类的定义，这里指定为 `core`。
- ❺ 跟 2.3/ 姜饼系统不同，启动和停止 `adb` 现在不再由 `persist.service.adb.enable` 属性所控制了。正如注释所写的，它现在由 `init.usb.rc` 文件控制。下一节我们再讨论这个话题。
- ❻ `netd` 是 `main` 类中第一个启动的服务。
- ❼ 正如我在第 2 章所说，Surface Flinger 不再是 System Server 的一部分了，它现在以一个独立的进程单独启动。

## init.usb.rc

这个文件处理 USB 相关的所有事务，为了更好地理解它的操作及其所设置的值，你需要看一下 `frameworks/base/services/java/com/android/server/usb/` 中 USB 系统服务的代码。

```
# Copyright (C) 2012 The Android Open Source Project
#
# USB configuration common for all android devices
#
on post-fs-data
    chown system system /sys/class/android_usb/android0/f_mass_storage/lun/file
    chmod 0660 /sys/class/android_usb/android0/f_mass_storage/lun/file
    chown system system /sys/class/android_usb/android0/f_rndis/ethaddr
    chmod 0660 /sys/class/android_usb/android0/f_rndis/ethaddr

# Used to disable USB when switching states
on property:sys.usb.config=none ❶
    stop adbd ❷
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/bDeviceClass 0
    setprop sys.usb.state ${sys.usb.config}

# adb only USB configuration
# This should only be used during device bringup
# and as a fallback if the USB manager fails to set a standard configuration
on property:sys.usb.config=adb
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18d1
    write /sys/class/android_usb/android0/idProduct D002
    write /sys/class/android_usb/android0/functions ${sys.usb.config}
    write /sys/class/android_usb/android0/enable 1
    start adbd ❸
    setprop sys.usb.state ${sys.usb.config}

# USB accessory configuration
on property:sys.usb.config=accessory
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18d1
    write /sys/class/android_usb/android0/idProduct 2d00
    write /sys/class/android_usb/android0/functions ${sys.usb.config}
    write /sys/class/android_usb/android0/enable 1
    setprop sys.usb.state ${sys.usb.config}

# USB accessory configuration, with adb
on property:sys.usb.config=accessory,adb
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18d1
    write /sys/class/android_usb/android0/idProduct 2d01
    write /sys/class/android_usb/android0/functions ${sys.usb.config}
    write /sys/class/android_usb/android0/enable 1
    start adbd
    setprop sys.usb.state ${sys.usb.config}

# audio accessory configuration
on property:sys.usb.config=audio_source
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18d1
    write /sys/class/android_usb/android0/idProduct 2d02
    write /sys/class/android_usb/android0/functions ${sys.usb.config}
```

```

write /sys/class/android_usb/android0/enable 1
setprop sys.usb.state ${sys.usb.config}

# audio accessory configuration, with adb
on property:sys.usb.config=audio_source,adb
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18d1
    write /sys/class/android_usb/android0/idProduct 2d03
    write /sys/class/android_usb/android0/functions ${sys.usb.config}
    write /sys/class/android_usb/android0/enable 1
    start adbd
    setprop sys.usb.state ${sys.usb.config}

# USB and audio accessory configuration
on property:sys.usb.config=accessory,audio_source
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18d1
    write /sys/class/android_usb/android0/idProduct 2d04
    write /sys/class/android_usb/android0/functions ${sys.usb.config}
    write /sys/class/android_usb/android0/enable 1
    setprop sys.usb.state ${sys.usb.config}

# USB and audio accessory configuration, with adb
on property:sys.usb.config=accessory,audio_source,adb
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18d1
    write /sys/class/android_usb/android0/idProduct 2d05
    write /sys/class/android_usb/android0/functions ${sys.usb.config}
    write /sys/class/android_usb/android0/enable 1
    start adbd
    setprop sys.usb.state ${sys.usb.config}

# Used to set USB configuration at boot and to switch the configuration
# when changing the default configuration
on property:persist.sys.usb.config=*
    setprop sys.usb.config ${persist.sys.usb.config} ④

```

- ① `sys.usb.config` 全局属性控制了 USB 连接的状态。该变量要么是由代码 `frameworks/base/services/java/com/android/server/usb/UsbDeviceManager.java` 设置的，要么就由这个配置文件后面部分根据 `persist.sys.usb.config` 设置的。
- ② 当 `sys.usb.config` 变为 `none` 时，`adbd` 就被关闭。
- ③ 基于 `sys.usb.config` 改变启动 `adbd` 的一个例子，文件中有好几个这种做法。
- ④ 只要 `persist.sys.usb.config` 被修改了，`sys.usb.config` 这里就会自动更新。然后，就会基于本文件前面声明的触发器触发相应动作。

## init.trace.rc

从 4.1/ 果冻豆系统开始，Android 系统就为应用开发人员提供了一个 `systrace` 的命

令。这个运行在主机上的工具实际上依赖于目标设备上的 *atrace* 工具，它通过 ADB 协议启动。*atrace* 采用了内核的 ftrace 功能来跟踪系统。这个 *init.trace.rc* 文件就是为 Android 的跟踪工具配置 ftrace。如果你在搜索引擎中搜索 ftrace，你可以找到很多关于其机制的文章。

```
## Permissions to allow system-wide tracing to the kernel trace buffer.  
##  
on boot  
  
# Allow writing to the kernel trace log.  
    chmod 0222 /sys/kernel/debug/tracing/trace_marker  
  
# Allow the shell group to enable (some) kernel tracing.  
    chown root shell /sys/kernel/debug/tracing/trace_clock  
    chown root shell /sys/kernel/debug/tracing/buffer_size_kb  
    chown root shell /sys/kernel/debug/tracing/options/overwrite  
    chown root shell /sys/kernel/debug/tracing/events/sched/sched_switch/enable  
    chown root shell /sys/kernel/debug/tracing/events/sched/sched_wakeup/enable  
    chown root shell /sys/kernel/debug/tracing/events/power/cpu_frequency/enable  
    chown root shell /sys/kernel/debug/tracing/events/power/cpu_idle/enable  
  
    chown root shell /sys/kernel/debug/tracing/events/power/clock_set_rate/enable  
    chown root shell /sys/kernel/debug/tracing/events/cpufreq_interactive/enable  
    chown root shell /sys/kernel/debug/tracing/tracing_on  
  
    chmod 0664 /sys/kernel/debug/tracing/trace_clock  
    chmod 0664 /sys/kernel/debug/tracing/buffer_size_kb  
    chmod 0664 /sys/kernel/debug/tracing/options/overwrite  
    chmod 0664 /sys/kernel/debug/tracing/events/sched/sched_switch/enable  
    chmod 0664 /sys/kernel/debug/tracing/events/sched/sched_wakeup/enable  
    chmod 0664 /sys/kernel/debug/tracing/events/power/cpu_frequency/enable  
    chmod 0664 /sys/kernel/debug/tracing/events/power/cpu_idle/enable  
    chmod 0664 /sys/kernel/debug/tracing/events/power/clock_set_rate/enable  
    chmod 0664 /sys/kernel/debug/tracing/events/cpufreq_interactive/enable  
    chmod 0664 /sys/kernel/debug/tracing/tracing_on  
  
# Allow only the shell group to read and truncate the kernel trace.  
    chown root shell /sys/kernel/debug/tracing/trace  
    chmod 0660 /sys/kernel/debug/tracing/trace
```

## 附录 E

# 资源

Android 还有更多的东西没法都在一本书内覆盖到。对于初学者来说，Android 的周围有一个活跃生态系统和大量的社区项目。本附录介绍了你在研究 Android 的工作过程中，会探索到的主要资源。

## 网站和社区

有大量的网站和社区与 Android 有直接或间接的关系。尽量有条理地给它们做如下的分类：

### Google

#### *Android 开源项目 (<http://source.android.com/>)*

谷歌是 Android 平台的主要网站。它在历史上包含关于系统的更多信息，但它们已经被删除了。但它仍然是如何获得源代码以及如何建立你的开发系统来构建 AOSP 的一个很好的参考。它还包含了 Android 兼容性计划的最新文档，其中包括符合定义文件。

#### *Android 开发者 (<http://developer.android.com/develop/index.html>)*

这是谷歌为应用程序开发人员建立的网站。与平台网站不同，这个网站有相当丰富的文档资源。它包含教程，API 参考，图形设计师的指导方针，以及更多其他内容。总之，如果你正在开发一个应用程序，你将从这个网站中得到良好的帮助。

#### *Android 工具项目网站 (<http://tools.android.com/>)*

这是一个包含 Android 开发工具信息的网站。包括 SDK、Eclipse 插件，NDK 等。

## Soc 供应商

用于 Sitara 的 TI Android 开发工具 ([http://www.ti.com/tool/android\\_sdk-sitara](http://www.ti.com/tool/android_sdk-sitara))

该开发套件包括一套 AOSP 源代码，它们被定制运行与基于 TI 的芯片，如 BeagleBone 的开发板上。你也可以在这里找到可供移植的信息。

Linaro Android (<https://wiki.linaro.org/Platform/Android>)

正如网站所说，“Linaro 的是一个不以营利为目的工程性组织，整合和优化了用于 ARM 架构的开源的 Linux 软件和工具。”实际上，这是一个服务于 Soc 厂商的组织，帮助它们开启新的平台。它们为会员维护了一个 Android 树，可供大家免费获得下载。

CodeAurora (<https://www.codeaurora.org/>)

这是 Linux 基金会实验室的一部分，并提供了基于高通芯片的各种开源项目。它维护了一个 Android 树。

## 其他分支

除了提供在它们网站上的信息之外，很多的这些分支都有公共的邮件列表，你会发现它们都非常有用。

CyanogenMod (<http://www.cyanogenmod.org/>)

这可能是最流行的 Android 分支。它本质上是一个 AOSP 发布的售后，针对技术人员和高级用户带来额外的特点和增强功能。最有趣的是，所有的开发工作都是开放完成的。

Android-x86 (<http://www.android-x86.org/>)

这是与 Intel 公司将对 x86 的支持合并到主 AOSP 树所不同的项目。相反，它面向将 Android 移植到电脑、上网本和笔记本电脑上。

RowBoat (<http://code.google.com/P/row boat/>)

这是由 TI 维护的社区项目，从 TI 的 Android 开发工具包进行了衍生。

Replicant (<http://replicant.us/>)

该项目旨在将 Android 组件尽可能多的替换为免费的软件。例如，它包括 F-Droid，一个免费的软件应用程序目录（本质上是谷歌市场的一个免费软件版本）。

除了上面的列表之外，也还有一个庞大且越来越多的封闭代码的 AOSP 分支。请记住，Android 的许可证是非常宽容的。

## 文档和论坛

*Linux 周末新闻 (<http://lwn.net/>)*

这是内核开发相关的所有事情的主要的新闻网站。当相关联的时候，Android 也被涉及，但它的重点肯定还是在经典的 Linux 发行版本和 Linux 内核。

*嵌入式 linux 维基百科 (<http://www.elinux.org/>)*

收集了大量有关嵌入式 Linux 信息的维基网站。一段时间以来，Android 也是其中的一部分。

*OMAPpedia ([http://omappedia.org/wiki/Main\\_Page](http://omappedia.org/wiki/Main_Page))*

这个维基百科网站包含有关在 TI 的 OMAP 处理器上使用 Linux 和 Android 的信息。有一些文章包括了很多详细的说明。

*xdadevelopers (<https://www.xda-developers.com/>)*

虽然这个网站传统来说经常被游戏制作者们访问，有时也包含一些很难获得的其他方面的信息。可以看一看 Android 部分，本书发现的有价值的信息很多都是来自于该网站的论坛里。

*Slideshare (<http://www.slideshare.net/>)*

这是一个共享幻灯片的通用网站。它包含了大量的 Android 有关的幻灯片，其中包括许多内部的或围绕其内部的组件。

*Vogella (<http://www.vogella.com/android.html>)*

这个网站是由拉尔斯·沃格尔维护，并提供有关 Android 应用程序开发的各种教程。这是一个对由谷歌发布的官方 Android 应用程序开发者信息的很好补充。

## 嵌入式 Linux 构建工具

*Buildroot (<http://buildroot.uclibc.org/>)*

该项目至今已经有十多年了，允许你构建一个目标嵌入式 Linux 根文件系统，以及基于一个使用基于菜单的系统来供给它的配置的工具。

*Yocto 项目 (<https://www.yoctoproject.org/>)*

与 BUILDROOT 类似，但其在目标上更加雄心勃勃。它包含生成整个嵌入式 Linux 发行版的框架和工具。

## 开放式硬件项目

*BeagleBoard 和 BeagleBone (<http://beagleboard.org>)*

目前市场上许多廉价的评估板。然而，BeagleBoard 和 BeagleBone 已建立了非常活跃的社区。原理图也可提供。

## 书籍

*Building Embedded Linux Systems, 2nd*, 由 arim Yaghmour, Jon Masters, Gilad Ben-Yossef 和 Philippe Gerum 创作 (O'Reilly, 2008)

这是关于嵌入式 Linux 话题的经典的书籍，最初是由我写起来的，后来在 Jon Masters 的领导下更新的。

*Embedded Linux Primer, 2nd*, 由 Christopher Hallinan 创作 (Prentice Hall, 2010)

另一本好的嵌入式 Linux 的书。

*Linux Device Drivers, 3nd*, 由 Jonathan Corbet, Alessandro Rubini 和 Greg Kroah-Hartman 创作 (O'Reilly, 2005)

先不管它的年份，这本书有很多对 Linux 设备驱动作者的建议和参考。

*Linux Kernel Development, 3nd*, 由 Robert Love 创作 (Addison-Wesley, 2010)

关于内核内部书籍中经得起时间考验的一本。

*Linux Kernel Architecture*, 由 Wolfgang Mauerer 创作 (Wrox, 2008)

另一本关于内核的书籍。

*Programming Android, 2nd*, 由 Zigurd Mednieks, Laird Dornin Blake Meike 和 Masumi Nakamura 创作 (O'Reilly, 2012)

一本关于应用程序开发的比较深入的书。

*Learning Android*, 由 Marko Gargenta 创作 (O'Reilly, 2011)

另一本关于应用程序开发的介绍性的书。

*Professional Android 4 Application Development*, 由 Reto Meier 创作 (Wrox, 2012)

一本应用程序开发的书，由在谷歌的 Android 开发者关系团队中技术领导者所写。

## 会议和活动

Android 构建者峰会 (<https://events.linuxfoundation.org/events/android-builders-summit>)

工作在 AOSP 堆栈的开发者的主要活动。

嵌入式 Linux 会议 (<https://events.linuxfoundation.org/events/embedded-linux-conference>)

与嵌入式 Linux 相关的所有事情的主要活动。

**嵌入式 Linux 欧洲会议** (<https://events.linux foundation.org/events/embedded-linux-conference-europe>)

欧洲开展的嵌入式 Linux 会议。

**Linaro 联络会** (<http://www.linaro.org/connect>)

Linaro 用于汇集其成员和开发者的活动。

**AnDevCon** (<http://www.and evcon.com/>)

应用程序开发者的主要会议，也有一些平台的讨论。

## 作者介绍

---

**Karim Yaghmour** 既是一系列企业的创业者，又是一个狂热的极客。他是 Opersys 公司的 CEO，该公司在嵌入式 Android 和嵌入式 Linux 方面提供开发和培训服务，但是他最广为人知的事是，他创作了 O'Reilly 出版社出版的《构建嵌入式 Linux 系统》一书，这本书成千上万地在世界各地销售，并已被翻译成多种不同的语言。

在 20 世纪九十年代末，Karim 率先在 Linux 的追踪领域里引入了 Linux 跟踪工具包 (LTT)。他一直维护着 LTT 到 2005 年，并有多家公司的开发人员加入了对该工具的开发，包括 IBM，惠普和英特尔。LTT 的用户包括谷歌，IBM，惠普，甲骨文，阿尔卡特，北电网络，爱立信，高通，美国航空航天局，波音公司，空中客车公司，索尼，三星，NEC，富士通，SGI 公司，红帽，泰雷兹公司，欧瑞康，公牛，摩托罗拉，ARVI 和 ST Micro。他的其他贡献包括 relayfs 和先进地球观测卫星。

Karim 的文章已经提交并发表在同行评议的具有科学性的行业会议，杂志和在线出版物上，包括 Usenix，Linux 内核峰会，嵌入式 Linux 大会，Android 构建者峰会，AnDevCon，嵌入式系统大会，渥太华的 Linux 研讨会，Linux 杂志，O'Reilly 网络，以及实时 Linux 研讨会。

## 封面介绍

---

本书的封面上的动物是摩尔壁虎 (*Tarentola mauritanica*)，它是壁虎的一种，原产于欧洲和北非的地中海西部地区，在北美和亚洲也发现过。据观察，它通常会在城市地区的墙壁上，这些城市主要是温暖的沿海地区，特别是在西班牙，尽管它也可以在内陆生存。收养这个物种作为宠物，已经成为美国佛罗里达州和其他地方的人们的爱好。

摩尔壁虎主要是昼伏夜出，或在黄昏出没，但它有时候也在白天活动，尤其是在冬季快要结束时，阳光明媚的日子。它在每年四月至六月左右会下两次蛋，两个几乎球形的蛋。4 个月后，小于 5 厘米长的小摩尔壁虎就出生了。它们生长比较缓慢，通常需要圈养 4 ~ 5 年的时间。

成年的摩尔壁虎据测量可长达 15 厘米，包括尾巴。它们有一个健壮的身体，平头，以及它的瘤状结非常的大，这也使得这个物种带刺，看起来像铠甲一样。它们是棕灰色或褐色，专颜色较深或较浅的斑点，根据光的强度这些色彩会发生变化。