嵌入式 Android 开发

huobaiyu@126.com

前言

Android 的增长已经是一个十分引人注目的现象。目前它是市场上占有率最高的手机操作系统。很明显，Android 集开源许可，富有侵略性的市场策略和时尚的界面于一身，而这都是 Google Android 团队一力打造的。无疑，因 Android 而产生的海量用户已经不能被手机制造商，移动网络运营商，芯片制造商和应用程序开发人员所忽视。现在的产品，应用程序和设备都必须“为 Android 设计”、“和 Android 设备兼容”或“基于 Android 开发”。

除了在手机上非常成功之外，Android 也被另外一个不被注意的人群所关注-嵌入式系统开发人员。虽然大量的嵌入式设备几乎没有人机界面，但是还是有相当数量被认为是“嵌入式”的设备，有用户界面。对于大量的现代设备，除了单纯的技术功能外，开发面向用户设备的技术人员必须考虑人机接口要素。因此，设计人员要么向用户呈现他们熟悉的界面，要么冒点风险要求客户学习他们不熟悉或者说是全新的界面。在 Android 出现之前，用户接口的选择是非常有限的。

很显然，嵌入式开发者更喜欢为用户提供了一个他们已经熟悉了的接口。因此在过去，接口都是基于视窗软件的。苹果的 iOS 和谷歌的 Android 已经永远使基于触摸屏的、类似于 iPhone 那样的图形界面被大众所接受。这样的变化使 Android 在嵌入式的世界里越来越被人所重视。

与 Android 应用程序开发人员比起来 ,许多想要做各种 Android 平台工作的开发人员，包括想要移植 Android 到嵌入式设备中的开发人员都面临一个共同的问题就是缺少文档来告诉他们如何去做。尽管谷歌为应用程序开发人员提供了大量的在线文档 ,同时也出版了一些对应主题的书籍，但是嵌入式开发人员仍然缺少有效的文档，使他们在移植 Android 到嵌入式设备时不得不直接从 Android 的系统源码开始入手。

这本书的目的是纠正这种情况，使您能在任何设备中嵌入 Android。因此，您将学习有关 Android 的架构，如何定位它的源代码，如何修改它的各个组成部分，以及如何为特定设备创建你自己的版本。此外，您将学习如何将 Android 集成到 Linux 内核中，以及如何利用 Android 的 Linux 的遗产。例如 ,我们将讨论如何把“经典”的 Linux 组件 ,如 glibc 和 BusyBox 打包成 Android 的一部分。同时，您将学习日常的提示和技巧，比如如何使用 Android 的 repo 工具以及集成或修改 Android 的编译系统。

第一章 介绍

将 Android 在嵌入式设备上的，是一项复杂的任务，需要对 Android 内部有非常深刻的理解，并且对于 Android 开源项目和它所运行的 Linux 内核要求有非常巧妙的修改。在了解嵌入式 Android 开发的细节之前，需要对一些 Android 开发的背景知识做一下了解，比如 Android 硬件需求，它的框架和其在嵌入式环境下所隐含的东西。

History

故事要追溯到早在 2002 年谷歌的拉里·佩奇和谢尔盖·布林出席一个在斯坦福大学所举行的演讲。演讲是由 Danger 公司带来的关于新一代 Sidekick 手机的开发。演讲者是当时 Danger 公司的 CEO 安迪·鲁宾，而 Sidekick 是第一个多功能并支持互联网的设备。演讲之后，拉里去了看该设备，并很高兴地看到，谷歌是默认的搜索引擎。不久后，拉里和谢尔盖都成为 Sidekick 用户。

尽管 Sidekick 非常新奇，用户也非常热情。但是该设备并没有取得商业上的成功。到 2003 年，鲁宾和 Danger 的董事会都同意，这是他离开的时候了。在尝试做了一些其它事情后，鲁宾还是决定回到他想做的手机操作系统业务中。他使用了以前拥有的域名 -android.com，并着手为手机制造商创建一个开放的操作系统。鲁宾把大部分的积蓄投资到该项目，并获得了一些额外的种子基金钱后，之后他得到了谷歌的投资。在 2005 年 8 月，谷歌收购了 Android 公司，但是很少大张旗鼓地宣传。从谷歌收购之后到 2007 年 11 月向世界公布之间这段时间内，几乎没有关于 Android 的信息。取而代之的是，整个 OS 开发团队在幕后疯狂的工作。最初的消息是由开放手机联盟（OHA）发布的，这是一个由多家公司组成的组织，其标榜的使命就是为移动设备开发公开的标准，而 Android 手机是它的第一个产品。一年后，于 2008 年 9 月，第一个开源版本的 Android 1.0 被发布。

从那时起，Android 已经发布了多个版本，并且操作系统的演进和开发明确更加公开。然而稍后将会看到，在 Android 上的许多工作都是“关起门”来完成的。

Android Versions

版本	发布日期	代号	主要功能	是否开源
1.0	2008 年 9 月	未知		是
1.1	2009 年 2 月	未知		是
1.5	2009 年 4 月	Cupcake	屏幕上的软键盘	是
1.6	2009 年 9 月	Donut	屏幕上显示电池使用率和 VPN 支持	是
2.0,2.0.1,2.1	2009 年 10 月	Eclair	Exchange 支持	是
2. 2	2010 年 5 月	Froyo	Just-In-Time (JIT) compile	是
2.3	2010 年 12 月	Gingerbread	SIP 和 NFC 支持	是
3.0	2011 年 1 月	Honeycomb	平板外形支持	否
3.1	2011 年 5 月	Honeycomb	支持 USB 主机和 API	否
4.0	2011 年 12 月	Ice-Cream Sandwich	合并手机和平板尺寸规格支持	是

功能及特点

谷歌公布关于 Android 有以下特点：

Application framework

应用程序开发人员用 *Application framework* 来创建 Android 应用程序。该框架的使用可以参考 http://developer.android.com 和 O'Reilly 的 Learning Android 这样的书。

Dalvik Virtual Machine

Android 使用 clean-room 字节编码翻译器来代替 Sun Java 的虚拟机。和 Java 虚拟机翻译 .class 和 .jar 文件不同，Dalvik 翻译 .dex 文件。这些文件都根据 java 编译器产生的 .class 文件生产，并被 dx 工具使用 。

Integrated browser

Android 包括了一个基于 WebKit 的浏览器 ,包括在其标准的应用程序列表中。在自己的应用程序中，应用程序开发人员可以使用 WebView 类来使用 WebKit 引擎。

Optimized graphics

Android 提供了自己的 2D 图形库 ,但依赖于 OpenGL ES 提供 3D 的能力。

SQLite

这是标准的 SQLite 数据库，在 http://www.sqlite.org 中可以找到。它通过应用程序框架提供给应用程序开发人员。

Media support

Android 通过其自定义的 StageFright 媒体框架支持广泛的媒体格式。2.2 之前，Android 依靠的是 PacketVideo 的 OpenCore 的框架。

GSM telephony support

电话支持是依赖于硬件，并且设备制造商必须提供的 HAL 模块，以便使 Android 访问他们的硬件。HAL 模块将在下一章中讨论。

Bluetooth, EDGE, 3G, and WiFi

Android 支持大多数无线连接技术。虽然有些是 Android 特定的方式实现的，如 EDGE 和 3G，而另外一些则和 Linux 实现的一样，如蓝牙和 WiFi。

Camera, GPS, compass, and accelerometer

和用户环境交互是 Android 的关键。在应用程序框架中有各种 API 来访问这些设备，一些 HAL 模块被提供来进行设备支持。

Rich development environment

这可能是 Android 的最大的资产之一。提供给开发者的开发环境，可以非常容易地开始进行 Android 软件开发。一个完整的 SDK 是免费提供下载，并带有一个仿真器，一个 Eclipse 插件，和一些调试和分析工具。

当然 Android 还有更多功能可以被列出来，如 USB 支持，多任务，多点触摸，SIP，網絡共享，语音控制等。但是上述列表让人更好的理解了 Android 有哪些重要的功能。另外请注意，每一个新的 Android 版本都带来了它自己的一套新的功能。通过查看每个版本的 Platform Highlights, 可以了解到更多的功能信息。

除了基本功能集，Android 系统平台拥有一些有助于它在嵌入式设备中使用的特征。下面是一个简单的总结：

Broad app ecosystem

在写这篇文章的时候，有 20 万个应用程序在 Android Market 上。与此相比 ,相当受欢迎苹果的 App 市场有 35 万个应用程序 ,这确保了预装在 Android 设备上的应用程序有一个很大的候选名单。

Consistent app APIs

应用程序框架提供的所有 API 都是向前兼容的。因此，开发人员为嵌入式设备开发的定制应用程序在未来的 Android 版本上都应该能顺利的运行。相比之下，对 Android 的源代码进行的修改将不能保证一直有效，也可能无法在下

一个 Android 发布版本中运行。

Replaceable components

由于 Android 是开源的，它的架构的一个好处就是组件可以完全更换。举例来说，如果你不喜欢默认的应用程序启动器（主屏幕），你可以写你自己的。对 Android 更深层次的改变也是可以做的。比如 GStreamer 的开发人员可以用 GStreamer 来代替 Android 默认的媒体框架 StageFright ,而无需改变应用 API。

Extendable

Android 的开放性和它的架构的另一个好处是，添加额外的功能和硬件的支持是比较简单的。你只需要模拟平台正在为其他相同类型的硬件或功能做什么。例如，您可以通过为 HAL 添加一些文件来对自定义硬件提供支持。

Customizable

如果你宁愿使用现有的组件，如现有的启动应用程序，您还可以根据自己的喜好自定义他们。无论是调整这些组件的行为，或改变它们的外观和感觉，都可以根据需要自由修改 AOSP。

开发模型

在考虑是否采用 Android 时，至关重要的一点就是，开发人员要理解对开发流程所做的任何改变所带来的后果 ,以及这些流程内部所需要的所有依赖关系。

和传统开源项目的不同

Android 的开源性质是其最大力宣传的特点之一。事实上，正如我们所看到的，很多开源软件工程的好处都适用于 Android。

但是，Android 的许可是不同于大多数开源项目，因此它的开发工作基本上都是谷歌自己完成的。例如，绝大多数的开源项目有公共的邮件列表和论坛，借此主要开发人员可以发现彼此，并进行互动，通过主要开发分支可以找到公共源代码库。但是 Android 什么都没有。

安迪·鲁宾（Andy Rubin）自己做了最好的总结："开源项目并不意味着就一定是社区驱动的。Android 在某种程度上讲，更强调开源而不是社区驱动。

不管我们喜不喜欢，Android 都主要是由谷歌的 Android 开发团队完成开发的。公众不参与内部讨论和开发工作。谷歌每 6 个月会推出一款安装了新版本 Android 的硬件设备。例如，三星 Nexus S 发布于 2010 年 12 月，它运行的是 Android2.3 Gingerbread，这个版本可以通过 http://android.git.kernel.org/

访问到。

Android 开发模式和大多数传统意义上的开源项目是不同的，但是它还被冠以“开源”的头衔，总会让人感到不舒服，尤其是在考虑到它还是如此的流行。开源社区在历史上采用类似开发模式的项目都不太成功。

抛开政治因素，一个开发人员在现有 Android 开发模式下所作出的贡献是极其有限的。除非你能加入谷歌内部的 Android 开发团队，否则不可能对 Android 开发分支上的最新版本做出任何贡献。此外，公众也不可能和核心开发团队成员一对一的讨论。但是，可以自由提出改进和对 AOSP 代码进行修正，并保存在 http://android.git.kernel.org/。

谷歌的做法最糟糕的副作用是绝对没有办法获得 Android 开发团队所做出的关于平台决定的内幕信息。如果在 AOSP 添加新的功能，例如，如果做了核心部件的修改，则只能通过阅读源代码才能找出有多少改变和它们的影响。但是真正的开源项目会有一个公共邮件列表来记录所有的信息，至少会给出指向这些信息的链接。

话虽这么说，但是我们同样要记住谷歌把 Android 放在开源许可证下是一个重大的贡献。尽管从开源社区的角度来看，开发模式很尴尬。但是让谷歌继续投入到 Android 上，对于广大开发人员来说仍然是天赐良机。此外，Android 成就了以往任何一个开源项目都没能做成的事，就是创建了一个迄今为止最成功的 Linux 发现版。因此，我们很难指责 Android 开发团队所做的工作。Android 的成功无疑得益于谷歌的能力。

包括的功能，路线图和新版本

简单地说，未来的 Android 版本的特性和功能没有公开的路线图。在最好的情况下，谷歌将提前宣布下一个版本的名称和可能的发布日期。通常情况下，在五月举行的谷歌 I/ O 大会上会发布一个新的 Android 版本，另一个版本会在年底发布。

然而，通常情况下，谷歌将选择一个单一制造商的来一起推出包含下一个 Android 版本的移动设备。在此期间，谷歌将与此制造商的工程师紧密合作以便让新的 Android 版本在即将到来的旗舰产品上工作。因此该制造商的工程师可以访问新 Android 版本的开发分支。一旦设备被投放到市场上，相应的源代码转储的公共仓库中。在接下来的版本中，他们选择其他制造商，并重新开始。

对于这种周期有一个显着的例外：Android 的 3.x/Honeycomb。这个特定的案例中，谷歌还没有发布源代码给相应的旗舰产品，摩托罗拉 XOOM。所有

的迹象表明，这些代码可能永远不会公开。原因似乎是 Android 开发团队为了使 Android 平板电脑得到一个工作版本，以便尽快投放市场而建立了一个代码分支。因此，在该版本不能很好的做到向后兼容。鉴于此，谷歌并不希望代码公布，以避免其平台的碎片化。相反，无论是手机和平板电脑的支持将合并到推出的 Android4.0/Ice-Cream 三明治发行版中。

生态系统

截至 2011 年 6 月：

- 从 2010 年 8 月的 20 万部 ,12 月的 30 万部上升到每一天 40 万部 Android 手机被激活。
- Android 的市场包含大约 20 万的应用程序。相比较而言，苹果的 App 商店有 35 万的应用程序。
- 所有在美国销售的手机中超过三分之一都是基于 Android 的。

显然，Android 是在上升。事实上，Gartner 在 2011 年 4 月预测，Android 到 2015 年将占领约 50%的智能手机市场。就像 10 年前 Linux 进入嵌入式市场一样，Android 正准备同一领域留下自己的印记。Android 将不仅在移动手机领域成为颠覆者，同样在以用户为中心的嵌入式设备领域中，Android 同样会成为事实上的 UI 标准。

因此，围绕着 Android 迅速建立起了一个完整的生态系统。芯片及系统芯片（SoC）的制造商，如 ARM，TI，高通，飞思卡尔，NVIDIA，TI 已经让他们的产品增加了对 Android 的支持，同样的，手机和平板电脑厂商，如摩托罗拉，三星，HTC，索尼爱立信，LG，爱可视，DELL，ASUS 等，他们 Android 的设备的出货量不断增加。这个生态系统还包括越来越多的不同领域的玩家，如亚马逊，Verizon，Sprint 和 Barnes&Nobles，他们都在创建自己的应用程序市场。

围绕 Android 许多民间的社区和项目纷纷涌现。尽管 Android 本身是封闭开发的。干这种事的人都是一些电话的爱好者，他们黑进了制造商提供的二进制文件，创建自己的修改或变型，而其他人有更多的开源的色彩，依靠 Android 的源代码，建立自己的分支。例如，XDA-developers.com 是一个网上论坛，是爱好者常常浏览的一个网站。而 cyanogenmod.com 网站主持了一个 Android 的分支通过修改 Android 的源代码来提供额外的功能和增强功能。其

它的 Android 分支包括 Replicant (http://replicant.us) 尽可能多的用免费软件来替代 Android 组件，还有就是 MIUI (http://en.miui.com/),它提供了一些很酷的 UI。

开放手机联盟宣言

正如前文提到的，OHA 是最早宣布 Android 的组织。在其官网上是这样描绘自己的：“我们是由 82 家技术和手机公司组成，成立的目的在于加快手机领域的技术创新步伐，并为消费者提供更丰富，更便宜，更好的移动体验。我们一起开发了 Android™，它是第一个完整，开放，免费的手机平台。“

然而，除了最初的公告，目前还不清楚 OHA 扮演什么样的角色。例如，在 2010 谷歌开发者大会上，一个参与了“Android 团队座谈”环节的工程师问道，他们公司作为 OHA 的成员应该享有哪些权限？但是座谈的人员回答到他们不知道，因为他们不是 OHA。因此对于 Android 开发团队本身来讲 OHA 成员的好处是不明确的。

OHA 的角色很模糊，因为它不是一个全职的组织，没有董事会和长期工作人员。相反，它只是一个“联盟”。此外，在任何谷歌的 Android 公告中都没有提及 OHA，也没有任何新的 Android 公告从 OHA 发布出来。总之，人们会忍不住猜测，谷歌有可能把 OHA 主要是作为营销手段，以显示行业对 Android 的支持有多少，但在实践中它几乎不会影响 Android 的发展。

得到 "Android"

要让 Android 顺利的运行在你的嵌入式设备上，需要得到 2 个东西：兼容 Android 的 Linux 内核和 Android 平台。

在写这本书的时候 ,还不能从 kernel.org 下载"vanilla"内核来运行 Android 平台。相反，只能使用 AOSP 提供的内核或者给"vanilla"内核打补丁，让它兼容 Android 的内核。不幸的是，合并 Android 修改到主线内核的努力都失败了。据预计，长期来看，主线内核支持 Android 这个悬而未决的问题将得到解决。然而，就目前而言，我们必须接收的事实就是和 Android 兼容的内核只能作为主线一个分支而存在。

Android 平台本质上是一个定制的 Linux 发行版，它包含了一些运行在用户空间的包。表 1-1 中列出的版本实际上是 Android 平台的版本。我们将在下一章讨论 Android 平台的内容和架构。需要记住一点，Android 平台发布版就

和 Ubuntu 或 Fedora 这样的标准 Linux 发布版类似。它是一套完整的套装软体，一旦建成，就提供了一个特定的用户体验、特定的工具，界面和开发用的 API。

法律框架

和其他的软件一样，Android 的使用和分发许可受到一些许可、知识产权限制和市场现实的压制。下面是其中的一些限制：

代码许可证

正如前面所讨论的，“Android”有两部分：兼容 Android 的 Linux 内核和一个 AOSP 版本。尽管 Linux 内核被修改后才能运行的 AOSP，但它仍然要遵守 GNU GPLv2 许可。因此，请记住，对内核的任何修改都只能在 GPL 的许可下分发，其它的许可都不行。因此，如果从 android.git.kernel.org 获取了一个内核版本，并修改它以便运行在自己的系统中时，只要遵守 GPL 就可以在自己的设备中分发内核文件。同时，对应的源码也要遵守 GPL。

由 Linus Torvalds 在其被复制的文件清楚地表明了只有内核是受 GPL 约束，在它上面运行的应用程序不被视为“衍生产品”。因此，可以按照自己选择的许可自由创建和分发运行在 Linux 内核上的应用程序。

这些规则及其适用性已经被开放源码界和大多数选择支持的 Linux 内核公司普遍理解和接受。除了内核，大量基于 Linux 发行版的重要部件，都通常有这种或那种形式的 GPL 许可。例如，GNU C 库（glibc）和 GNU 编译器（GCC）都分别获有在 LGPL 和 GPL 授权。

然而并不是每个人都喜欢 GNU GPL，事实上，它对衍生产品的限制，并考虑到大型企业开发部门的不同地理分布，文化差异以及对外部分包商的依赖后，会发现这些都构成了严峻的挑战。在北美地区，制造商把产品推到市场销售之前，不得不应对几十、上百家的供应商。这些供应商会提供一份可能包含或不包含 GPL 的代码。此外，流程必须到位，以确保工作基于 GPL 项目的工程师，是遵守许可证的。

当谷歌与制造商一起工作开发“开放式”的手机操作系统时，很明显 GPL 必须尽可能地避免。事实上，除了 Linux 外其它内核也曾经被考虑过，但是最终 Linux 被选中，其中主要原因是它得到了产业的强有力支持，尤其是 ARM 芯片厂商的支持，另一个原因是它很好的和系统的其它部分隔离开，因此 GPL 许可的影响很小。

Branding Use

尽管 Google 慷慨的公布了 Android 的源代码，但是对品牌元素的使用却严格的得多。下面来看看这些元素和使用它们的相关条款。官方的正式条款清单，可以查看 http://www.android.com/branding.html。

Android Robot

所有和 Android 相关的东西上都可以看见这个绿色的机器人。它的作用类似 Linux 的企鹅标志，同时它的使用权限也是一样的。事实上谷歌宣布“这个标示可以被使用、复制和根据营销传播的需要自由修改。”唯一的要求就是要遵守 Creative Commons Attribution 许可。

Android Logo

这是一组自定义的字体，由字母 A-N-D-R-O-I-D 拼出来的。并出现在设备和仿真器启动的时候，也出现在 android.com 网站上。该商标在任何情况下都不能使用。

Android Custom Typeface

这是自定义的字体，用来呈现 Android 标志，并只能用于 Logo 使用。

"Android" in Official Names

这个词只能谷歌用。如果其他厂商要用就必须和谷歌达成协议。

"Droid" in Official Names

不能单独使用“Droid”这词。

"Android" in Messaging

在文本中可以使用 Android。

谷歌自己的 Android 应用程序

AOSP 有一个基本应用程序集。Android 手机还包含一个额外的程序包，如 Android Market、YouTube、组，“地图和导航”和 Gmail 等。如果想在自己的设备中包含这些应用程序，厂商需要和谷歌签订协议，并遵守 ACP。同时要在产品名字中包含 Android。

其它的 APP 市场

可以用 Android 公开的 API 和开源许可建立自己的 APP 市场。至少有一

个开源项目，FDroid（http://f-droid.org/repository/）在 GPL 许可下，提供了一个 App market 的应用程序和相应的服务器后台。

Oracle v Google

甲骨文收购 Sun 公司后拥有了 Java 的知识产权。就 Java 知识产权问题甲骨文起诉谷歌侵犯了它的知识产品。不管这个案子本身是神马，Android 严重依赖 Java 是不争的事实。Java 主要有 3 个部分：Java 语言和它的语义、虚拟机和类库。甲骨文发布的 Java 组件官方版包括 JDK 和 JRE。

Android 用了 JDK 的编译器，虚拟机是 Dalvik-是 Android 定制版，类库用的是 Apache Harmony

硬件及合规要求

原则上，Android 应该运行在任何可以运行 Linux 的硬件上。这些硬件架构包括 ARM、x86、MIPS、SuperH 和 PowerPC。所以如果要移植 Android 到自己的硬件设备之前，必须先把 Linux 移植过去。此外，还需要一些额外的硬件，比如显示和方便用户交互的输入设备。如果自己的设备没有 AOSP 需要的外设，就需要修改 AOSP。比如如果没有 GPS，就必须向仿真器一样提供一个模拟的 GPS HAL 模块。同时，设备的内存和 CPU 都要足够好。

总而言之，要让 Android 在自己的设备上跑起来是有一些限制。其中之一就是要通过 ACP，ACP 有两个独立但又互补的部分：合规定义文件（CDD）和合规性测试套件（CTS）。每一个 Android 版本都有自己的 CDD 和 CTS。因此，开发者要注意使用正确的 CDD 和 CTS。

ACP 的目的是确保用户和应用程序开发人员有统一的生态系统。更多的细节请查看 http://source.android.com/compatibility/。

合规定义文件

CDD 是 ACP 的一部分，可以在 ACP 的网站上找到。它主要指出了设备要达到合规性必须满足的一些条件。CDD 的语言是基于 RFC2119，因此大量使用“必须、应该、可能”等字眼来描述各种属性。该文档一共 25 页涵盖了设备软硬件能力的各个方面。该文档提及的各个方面并不能被 CTS 自动测试。下面快速浏览一下 CDD 文档。

软件

本节列出了对网站、虚拟机和用户接口方面的 Java 和原生 API 的兼容性需求。事实上，如果使用的是 AOSP，应该很容易符合本节的 CDD。

应用程序包的兼容性

本节规定，设备必须能够安装和运行.apk 文件。所有采用 Android SDK 开发的应用程序都被编译成.apk 文件，而这些都是通过 Android Market 分发并安装在用户的设备上的。

多媒体兼容性

这一节描述的是对设备的媒体编解码器（解码器和编码器），录音及音频延迟的要求。AOSP 包括 StageFright 多媒体框架，因此，使用 AOSP 的话就符合 CDD。然而，应该仔细阅读音频记录和延迟部分，因为它们含有可能会影响到设备必须配备的硬件类型或硬件配置的特定技术信息。

开发工具兼容性

本节列出了设备必须支持的 Android 专用工具。基本上，都是常见的应用程序开发和测试过程中使用的工具：adb、ddms 和 monkey。通常情况下，开发人员不直接使用这些工具。通常在 Eclipse 开发环境中进行软件开发和使用 Android 开发工具（ADT），ADT 是一个管理底层工具的插件。

硬件兼容性

本节对嵌入式开发人员来说是最重要的章节，因为他们对目标设备的任何设计决策都具有深远的影响。以下分章节进行简要介绍：

显示和图像化

- 设备的屏幕必须至少有 2.5”物理对角线尺寸。
- 其密度必须至少为 100dpi
- 其长宽比例必须是 4:3 和 16:9 之间

- 必须支持动态屏幕方向从纵向到横向，反之亦然。如果不能改变方向，它必须支持边框化①（letterboxing），因为应用程序可能会强制进行方向的变化。
- 必须支持 OpenGL ES 1.0，可以忽略 2.0 支持

输入设备

- 设备必须支持输入法框架，它允许开发人员创建自定义屏幕上的软键盘。
- 必须提供至少一个软键盘。
- 不能包含一个与 API 不相符的硬键盘。
- 必须提供“HOME”，“MENU”和“返回”按钮。
- 必须有一个触摸屏，无论是电容或电阻。
- 可能的话，它应该支持独立的跟踪点（多点触摸）。

传感器

传感器不是必配的，但是应该提供一份包含或不包含的传感器的清单。

数据连接

这里最重要的一点是 Android 可能被用于没有电话硬件的设备上。这是因为允许 Android 添加到平板设备上。此外，设备应该具有支持 802.11x，蓝牙和 NFC 的硬件。总之是要设备必须支持某种能支撑 200Kbits/s 带宽的网络制式。

摄像头

设备必须有后置摄像头，也可以有前置摄像头。

内存和存储设备

- 设备必须至少有 128MB 容量来存储内核和用户空间。
- 必须至少有 150MB，用于存储用户数据。
- 必须至少有 1GB 的“共享存储”，本质上这意味着 SD 卡

---

①常见的屏幕宽高比格式：
1.33:1 标准电视（电视机、电脑屏幕）；1.66:1 欧洲式宽高比；1.78:1 宽屏电视；1.85:1 美国式宽高比（传统电影）；2.35:1 宽屏电影。西尼玛斯科普式宽银幕立体声电影（大制作电影）

Pan&Scan：根据中心兴趣点裁剪画面
Letterboxing：保留画面完整性，上下留空隙带

- 还必须提供一种机制来和 PC 共享数据。换言之，该设备通过 USB 被连接时，SD 卡的内容必须在 PC 上能访问。

USB

这个需求可能是最能体现 Android 设备是如何以用户为中心的特征之一。因为基本上是假定用户拥有设备，因此，当它连接到电脑上时，需要让用户充分控制设备。但是在某些情况下是一个障碍，因为有可能设备制造商并不希望用户能把它的嵌入式设备连接到 PC 上。尽管如此，CDD 提出了下面的要求：

- 设备必须实现通过 USB-A 来实现连接的 USB 客户端。
- 必须实现 Android 调试桥（ADB）协议，这样通过 USB 可以执行 adb 命令。
- 它必须实现 USB 大容量存储，从而使设备的 SD 卡在 PC 上能被访问。

性能兼容性

虽然 CDD 不指定 CPU 速度的要求，但是指定了应用程序相关的时间限制，这将影响所选择的 CPU 速度。例如：

- •浏览器的应用程序启动时间必须小于 1300 毫秒。
- •MMS/ SMS 的应用程序启动时间必须小于 700 毫秒。
- •AlarmClock 的应用程序启动时间必须小于 650 毫秒。
- •重新启动已经运行的应用程序必须比第一次启动该程序花费的时间更少。

安全模式的兼容性

嵌入式设备必须符合 Android 应用程序框架，Dalvik 和 Linux 内核的安全环境。具体而言，应用程序必须能访问并遵守作为 SDK 文档一部分的权限模型。应用程序是要受”沙箱限制”，同样是作为独立进程运行并带有自己特有 UID。文件系统访问权限也必须和开发文档保持一致。最后即使自己的嵌入式设备不使用 Dalvik 虚拟机，那么其它被使用的虚拟机也必须要和 Dalvik 有相同的安全的行为。

软件兼容测试

设备必须通过 CTS，包括由人操作的 CTS 验证部分。此外，设备必须能够

图 2-1

运行 Android Market 上的特定应用程序。

软件更新

设备必须至少提供一种可更新的机制。一种可能的方式就是采用 OTA 空中下载技术进行离线更新，在重启设备后生效。另外一种方式可以是通过 USB 连接到 PC 完成更新。或者直接用一个可移除的存储设备来更新。

一致性测试套件

CTS 是 AOSP 的一部分，在第 10 章将会专门讨论。AOSP 包含一个特别的构建目标就是创建 CTS 命令行工具，该工具是控制测试套件的主要接口。CTS 依靠 adb 来对通过 USB 连接的设备进行测试。测试是建立在 Java 单元测试框架-JUnit 基础上的。主要测试框架的不同部分，例如，API，Dalvik 虚拟机，Intents, Per-missions 等。一旦测试完成，将生成一个 zip 文件，其中包含 XML 文件和屏幕截图，把这些文件发给 cts@android.com。

开发环境和工具

有两套独立的 Android 开发工具：用于应用程序开发的和用于平台开发的。如果想建立一个应用程序的开发环境，可以看 Learning Android 和谷歌在线文档。如果想建立平台开发环境，请查看本书，它们是完全不同的。

最基本的是需要一台 Linux 工作站来编译 AOSP。目前谷歌只支持 64 位的 Ubuntu 10.04。当然其它的 Ubuntu 版本或者 32 位系统也可能可以编译 AOSP。但是可能一些额外的工作需要做。如果不想重装环境可以选择一个虚拟机工具创建一个虚拟机-常用的工具推荐 VirtualBox，这个工具创建的客户机 OS 比较容易访问主机的串口。

未经编译的 AOSP 大约有 4GB 大，完成编译后有 10GB 大，如果为了测试需要创建多个 AOSP 版本的话，就需要几十上百 GB 的空间。如果用双核高端笔记本电脑从头编译 AOSP 的话需要 1 个小时，如果做了修改进行增量编译的话也至少要 5 分钟。因此要想开发 Android 嵌入式系统的话，配备的电脑要比较好。

图 2-1

第二章 内部探秘

我们已经了解了，Android 可以自由的下载、修改和安装到选定硬件设备上。事实上，获取 Android 代码，编译和在模拟器上运行都很简单。但要请给自己的设备定制 AOSP，就需要对 Android 内部的理解到一定的程度。在这一章我们将高屋建瓴的看看 Android 的内部，然后再在后面的章节中更详细的了解各个内部部件。

应用开发者的视角

由于，Android 的开发 API 是不同于任何其他现有的 API，包括在 Linux 中的任何 API。因此花些时间从应用开发者的角度来看看 Android 到底看起来像什么是很有必要的，即使那些修改过 AOSP 的人也不见得能看清楚。作为一名正在把 Android 嵌入到设备中的开发人员，可能并不一定会直接处理 Android 应用开发 API。这一章将把这些内容做一个总结，更加深入的内容请参看其它文章。

Android 的关键概念

应用程序开发人员必须在开发 Android 应用程序时考虑到几个关键概念。这些概念塑造了所有 Android 应用程序的体系结构，并决定了开发人员可以做什么和不能做什么。总体而言，它使大家的生活更美好，但有时也需要处理一些挑战。

组件

Android 应用程序是由一些松散的组件构成。一个应用程序的组件可以调用或使用其它应用程序的组件。最重要的是，一个 Android 应用程序并没有一个单一的入口点，没有 main()函数或其它类似的函数。相反会有一个预先定义好的事件-称为 intent，开发人员可以把他们的组件和这个 intent 进行绑定，当特定事件发生时，他们的组件就会被激活。举一个例子，一个负责管理用户通讯录数据库的组件，在用户拨号并按下通讯录按钮时就会被激活。因为一个应用程

序有多个组件所以它有多个入口点。有 4 个主要的组件类型：

Activities-活动

就像“窗口“是基于 windows 的系统中所有可视化交互的主要构造块一样，Activity 也是所有 Android App 的构建块。但和 Windows 不一样的是，Activity 不能被最大化、最小化或调整大小。相反 Activity 总是占据整个可视化区域，并彼此重叠在一起，像浏览器会记住网页被访问的顺序一样，用户也可以使用”回退”按钮回到上一个 Activity。大家应该还记得前面提过所有的 Android 设备都有一个”回退”按钮。但是和浏览器不一样的是，并没有一个”前进”按钮提供给用户。

有一个全局定义的 Android intent 允许应用程序在 app launcher 中显示为一个图标。因为绝大多数的应用程序都想要在主应用程序列表中出现，所以这些应用程序都要提供至少一个 activity 去处理这个 intent。通常情况下，用户在启动了一个特定的 activity 后，会在和这个 activity 有关的其他活动中来回移动，这样就会有一系列的活动被创建。这一系列的活动被称为一个任务。用户可以通过按 Home 键并在 app launcher 中启动一个新的 activity 序列来进行任务切换。

Services-服务

Android 服务是类似于 Unix 世界中的后台进程或守护进程。服务是在另一个组件需要它的服务的时候被启动，并在调用者的所需要的时间内都保持活动。不过，最重要的是，服务可以提供外部应用程序的组件使用，从而暴露出一些该应用程序的核心功能给其他的应用程序。服务一般都在后台运行，没有前台界面。

Broadcast Receivers-广播接收器

广播接收器是类似于中断处理程序。当一个按键事件发生时，一个广播接收器被触发，代表应用程序处理该事件。例如，一个应用程序可能想要在电池电量低或“飞行模式”（关闭无线连接）已被激活时被通知。如果不处理已经注册好的特定事件，广播接收器就会处于非活动状态。

Content Providers-内容提供器

内容提供器本质上就是数据库。通常情况下，一个应用程序如果需要将它的数据提供给其他 apps 访问就需要包括这个组件。所有内容提供器无论它们内部实现是怎样的，都提供相同的 API 给应用程序。大多数内容提供器都依靠 Android 内置的 SQLite 功能，但它们也可以使用文件或其他的存储类型。

Intents-事件

Intent 是 Android 中最重要的概念之一。他们是后期绑定机制，使组件进行交互。应用程序开发人员为活动发送 Intent 以便查看网页或 PDF 文档，这样即便发出请求的应用程序本身没有实现这个功能 ,用户也可以查看指定的 HTML 或 PDF 文件。这样通过 Intent 可以实现各种奇思妙想，不如开发人员可以发送一个特定的 Intent 以便触发打电话的行为。

可以把 Intent 看成是 Unix 中的多态信号 ,它们不一定必须是预定义或需要指定目标组件或应用程序。Intent 是一个被动对象。Intent 必须和接受它的组件类型进行绑定。如果一个 Intent 是发送给一个服务的话，那它就不能被活动或广播接收器接收。

在约束文件中可以用过滤器来声明组件要处理的特定 Intent 类型。此后，系统会把 intent 和过滤器进行匹配，并在运行时触发相应的组件。如果不想在组件过滤器中声明 Intent，则可以把 Intent 直接发送给组件。但是显式调用需要应用程序提前了解指定的组件，这种情况一般用于把 Intent 发送给位于同一个 app 内的组件。

组件生命周期

Android 另外一个关键点就是用户不能管理任务的切换，因此在 Android 没有任务栏或者类似的东西。取而代之的是 ,用户可以启动尽可能多的应用程序，并通过按 HOME 键回到主屏去启动任何其它的应用程序来进行切换。用户点击的应用程序可能是一个完全新的，或者是以前已经启动过，其活动堆栈（也称为一个“任务”）已经存在的应用程序。

这种设计的一个后果就是，随着启动的应用程序越多，占用的系统资源也将越多，这种状态不能永远持续下去。在某个临界点，系统就会开始从优先级最低

或最长时间未使用的组件那里回收资源，以便为新启动的应用分配需要的资源。当然这种资源回收对用户来说是完全透明的。换句话说就是当组件释放后，如果用户再返回来时，组件的状态应该和离开时的状态一致，就好像它一直在内存中一样。

为了使这种行为成为可能，Android 定义了一个标准的生命周期组件。应用程序开发人员必须管理其组件的生命周期，方法就是为每一个组件实现一系列额回调函数。这些回调函数会被和生命周期相关的事件所促发。比如一个活动不再出现前台时，OnPause()就会被触发。

管理组件的生命周期是应用程序开发人员所面临的最大挑战之一，因为他们必须在关键的过渡事件中小心地保存和恢复组件状态。理想的最终结果是，用户只需要做应用程序之间的“任务切换”而不会意识到前一个使用的 APP 已经被销毁以便为新启动的程序让路。

约束文件

如果必须有一个应用程序的“主”入口点，则约束文件可能是这样一个主入口点。约束文件告诉系统应用程序包含哪些组件，为了运行这个程序需要哪些能力，以及 API 的最低版本等。约束文件的格式是 XML，位于应用程序源代码的最顶端目录，叫做 AndroidManifest.xml。应用程序的组件通常是静态描述在约束文件中。事实上除了，广播接收器可以在运行时动态注册，其它组件都必须在编译时就在约束文件中被声明好。

进程和线程

每当一个应用程序的组件被系统或另一个应用程序激活时，就要启动一个进程以容纳该应用程序的组件。除非应用程序开发人员改变系统默认值，否则该应用程序第一个组件被激活后，其它组件都将运行在同一个进程中。换句话说，一个应用程序的所有组件都包含在一个单一的 Linux 进程中。因此，开发人员应在标准组件中避免长操作或阻塞操作，否则应该就用线程。

因为允许用户启动多个组件，所以一般都同时有多个进程在运行。当进程多到没有资源分配给新的应用程序时，Linux 内核的 Out-Of-Memory (OOM) 查杀机制将会启动，这时 Android 的内核中的 OOM 处理程序将调用，这个程序会决定那个进程必须被杀死以腾出空间。

简单地说，Android 的行为是基于低内存条件下的预测。如果应用程序开

发人员正确实现了组件的生命周期，用户应该看不到任何不良行为。

远程过程调用

像许多其他的系统组件 ,Android 的定义了自己的 RPC/ IPC 机制 :Binder。因此组件间的通讯并不是用传统的 System V IPC 或套接字来完成的。相反，组件使用内核中的粘合剂机制,通过/dev/binder 来实现通讯。

然而，应用程序开发人员，不直接使用粘合剂机制。他们必须定义和接口进行交互，接口要使用 Android 的接口定义语言（IDL）来定义。接口定义被存在一个.aidl 文件中，并由 aidl 工具进行处理后生产正确的存根对象（stub）以及编/解码代码，这样就可以通过 Binder 机制进行对象和数据的双向传输。

Framework 介绍

除了刚才讨论的概念，Android 也定义了自己的开发框架，它为开发人员提供了在其他开发框架中常见的功能。让我们来简单的介绍一下这个框架和它的功能。

User Interface

在 Android 中的 UI 元素，包括传统的小部件，如按钮，文本框，对话框，菜单，和事件处理程序。如果开发人员已经在任何其他 UI 框架下编过码，这部分就很好理解。

在 Android 中所有 UI 对象都派生于 View 类 ,并都在 ViewGroups 层次下进行组织。活动中的 UI 要么在 xml 文件中静态定义，要么就用 Java 语言进行动态生成。UI 也能根据需要在运行时用 Java 来更改。活动的 UI 会在它的内容被设置成 ViewGroup 层次结构的根目录时被显示。

Data Storage 数据存储

Android 为开发人员提供了多种存储选项。对于简单的存储需求，Android 提供了 shared preferences 机制 ,它允许开发人员把键-值对存储在数据集以便应用中的所有组件访问或者存在一个特定的对立文件中。开发人员可以直接操作这些文件。这些文件可以由应用程序私有存储，这样可以可以其它的应用访问权限，比如只读、可写或者完全不能访问。应用开发人员也可以使用 Android 内置的 SQLite 功能来管理他们自己的私有数据库。如果把数据托管到内容提供器中，那么其它应用程序也可以访问它。

Security and Permissions－安全性和权限

Android 中的安全性被强制在进程级别中执行。换句话说，Android 依赖于 Linux 的现有进程隔离机制，以实现自己的策略。为此，每一个安装后的应用程序都有自己的 UID 和 GID。从本质上讲，就好像每一个应用程序在系统中就是一个独立的“用户”。在任何多用户 Unix 系统，这些“用户”不能访问彼此的资源，除非授予明确地权限这样做。最终的结果就是，每个的应用程序都被放在自己单独的沙箱中。

要退出沙箱和访问关键的系统功能或资源，应用程序必须使用 Android 的权限机制 ,该机制要求开发人员在约束文件中静态的声明应用程序需要哪些权限。某些权限，如访问互联网（即使用套接字），拨打电话，或使用相机等，都是由 Android 预定义好的。其它的权限可以由开发人员自己定义，并由其它程序进行申请，以便和特定的应用程序组件进行通讯。当一个应用程序被安装时，会提示用户批准运行应用程序所需的权限。

访问执行是基于每个进程的操作，并且访问某个特定的 URI 或者给予一个特定的功能或资源的访问授权都是基于证书和用户提示。开发人员使用证书以便让他们开发的应用程序可以放在 Android Market 中。这样他们能对其它应用访问其功能时加以限制。Android 开发框架提供的功能比在这里提到要更多，可以访问 developer.android.com 网站以了解关于 2D 和 3D 图形，多媒体，地理位置和地图，蓝牙，NFC 等更多信息。

应用开发工具

开发 Android 应用程序的典型方法是使用免费提供的 Android 软件开发工具包（SDK）。SDK,加上 eclipse 和 ADT(Android 开发工具)插件，以及基于 QEMU 的 SDK 中的仿真器后，开发人员就可以开始进行程序开发了。开发人员在把应用程序放到 Android Market 之前，可能需要在整机上测试一下，毕竟整机和模拟器还是有不同。一些软件发行商做的比较极端，会在发布新版本前在几十款手机上进行测试。

即使不打算为嵌入式系统开发任何的应用程序 ,仍然强烈建议在自己的工作站上建立开发环境，这样开发人员就可以用基本测试程序来验证对 AOSP 的改变。

原生开发

虽然大多数的应用程序专门使用 Java 的开发环境开发，但是某些开发商需

要运行一些 C 代码。为此，谷歌已经提供给开发者原生开发套件（NDK）。主要原因是那些游戏开发人员需要把运行这些游戏的设备的性能发挥到极致。正因为如此，NDK 才能实现诸如图像渲染和传感器输入检索等事情。愤怒的小鸟就大量使用了原生代码编程。

NDK 的另外一个用途就是把现有的代码库移植到 Android 上。比如以前为旧的移动设备编写的大量 C 语言的程序，并不需要用 Java 再重新写一遍，只需用 NDK 重新编译然后和用 SDK 提供的一些 Android 特有的功能把编译好的文件与 Java 代码一起打包。Firefox 就是这么做的。

NDK 可以和 SDK 一起用，这样应用程序中就可以部分是 Java 部分是 C。关键要了解的是 NDK 只能访问 Android API 的一个非常有限的子集。比如不能从 NDK 编译的 C 代码中发送 Intent，这个只能在 SDK 中用 Java 来做。NDK 用到的 API 都是用来进行游戏开发的。

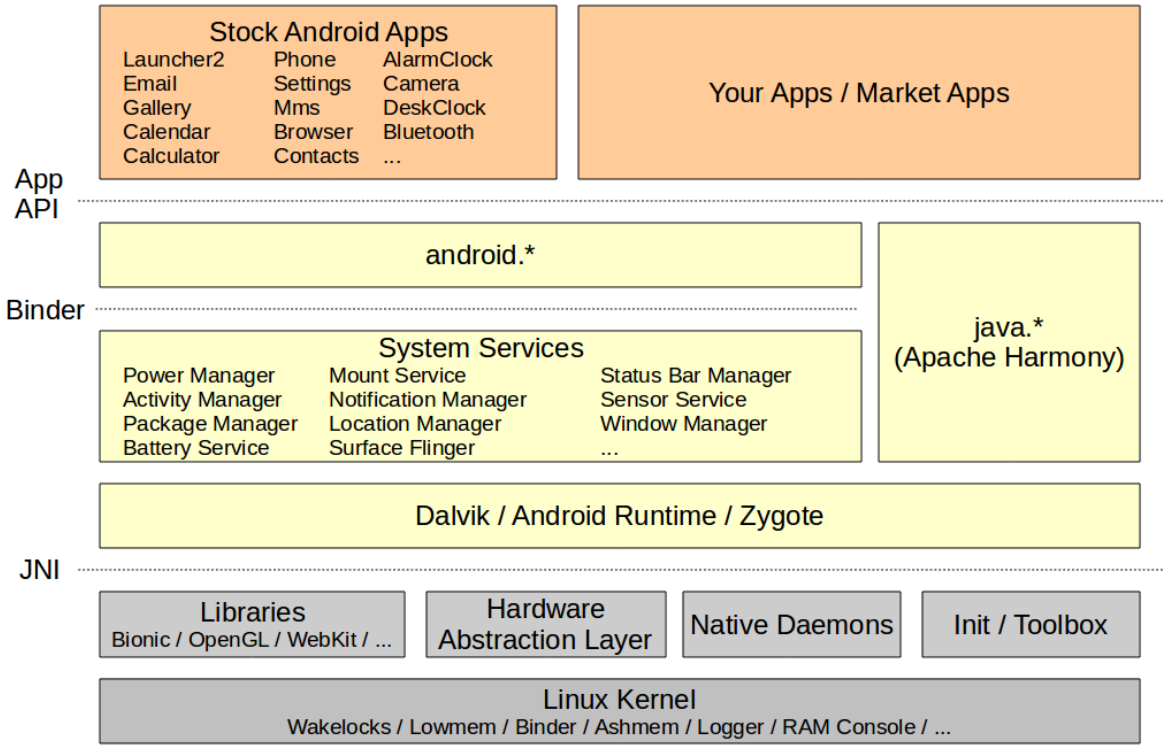

Figure 2–1. Android's architecture

有时嵌入式开发人员或者系统开发人员认为可以用 NDK 进行平台级的开发工作。其实 NDK 中的 Native（原生）这个词误导了大家，因为对于 Java 应用开发的限制和要求同样对 NDK 有效。因此嵌入式开发人员应该记住 NDK 的用途是应用开发人员可以在他们的 Java 代码中调用 C 的代码。除此之外，NDK

能起到的作用很小或者完全没用。

总架构

图 2-1[①]是本书中提出的最重要的图表之一，会经常提到它。虽然这是一个简化的视图，我们将来有机会充实它，但它给出了一个 Android 的架构，以及各个部分如何结合在一起的好的概念。

如果你熟悉某种形式的 Linux 开发，那么这个架构图除了 Linux 内核之外，其它的部分都和 Linux 和 Unix 不同。没有 glibc，没有 X window，没有 GTK，没有 BusyBox 等等。许多老牌 Linux 和嵌入式 Linux 从业者都指出，Android 的确感觉非常陌生。后面将谈到传统的或者是经典的 Linux 应用程序和工具如何与 Android 架构共存。

下面会从下往上介绍架构图中的各个部分。介绍完后本章结束，之后介绍 Android 启动流程。

Linux 内核

Android 和其它发行的 Linux 版本一样都是采用的"vanilla"内核，同样每个版本都在"vanilla"上打上自己的补丁以修复或增强某方面的功能。

Android 不同于标准的做法是它里面有各个核心功能是自己定制的，与"vanilla"内核自带的功能是完全不同的。事实上，一个 Linux 发行版附带的内核可以很容易地被 kernel.org 提供的内核所取代，而几乎不会影响到其余组件。但是 Android 如果不是自己的内核，则它用户空间的组件就无法工作，就像前面提到的那样，android 的内核是主线内核的一个分支。

虽然讨论 Linux 内核的内部超出了本书的范围，但是在这里仍然要了解一下 Android 为内核添加的一些主要功能。关于内核的详细信息建议参看 Robert Love 的《Linux 内核开发》一书（第三版）。也可以浏览 Linux Weekly News 网站，它会经常刊登一些相关文章。

注意以下小节只涵盖最重要的 Android 特有功能。Android 自有的内核是在标准内核上添加了几百个补丁，以提供给设备特定的功能，bug 修复和增强。可以用 git 工具把 http://android.git.kernel.org 提供的内核与主线内核做比较。另外，请注意在 Android 的内核中出现的一些功能，如 PMEM 驱动程序是某些设备特定的，不一定在所有 Android 设备中使用。

①图 2-1 和谷歌官方版本不一样。官方版本更适合应用开发人员。

Wakelocks

在 Android 所有的特定功能中是最有争议的，关于是否要并入主线内核的讨论不下 2000 封邮件，目前貌似已经并入主线内核了。

要了解 wakelocks 是什么，以及干了些什么，就必须先讨论在 Linux 中通常是如何进行电源管理的。Linux 的电源管理在笔记本电脑中最常见。在一个运行 Linux 的笔记本电脑的盖子扣上以后，电脑就进入挂起或休眠模式。在该模式下，系统的状态被保存在 RAM 中，而硬件都关闭。因此，电脑消耗的电池功率很小。当盖子被打开后，笔记本电脑就会苏醒，在很短的时间内用户就可以恢复使用。

这种模式对于笔记本电脑或者桌面电脑都工作的很好，但是却不适合于移动设备或者叫手持式设备。因此 Android 开发团队设计出一种机制，通过对规则略有改变来更匹配移动设备的需要。与 Linux 下由用户来决定系统休眠不同，Android 内核是尽可能的让系统休眠，但是为了防止在重要进程还在工作的时候或者正在等待用户输入的时候休眠，Wakelock 就会让系统保持清醒。

Wakelock 和前文提到了挂起功能都是建立在 Linux 现有的电源管理功能基础上的。但是他们的开发模式完全不同，因为应用程序开发人员或者驱动开发人员在执行关键操作或者等待用户输入的时候就必须明确获得 wakelock。通常，应用程序开发人员并不需要直接参与处理 wakelocks，因为他们使用抽象会自动处理所需的锁。如果它们需要 wakelocks 的时候，可以和电源管理服务进行通信。

在 LWN 中有下面一些文章详细解释了 wakelock，可以查看：

- Wakelocks and the embedded problem
- From wakelocks to a real solution
- Suspend block
- Blocking suspend blockers
- What comes after suspend blockers
- An alternative to suspend blockers

Low Memory Killer

如前文提到的，Android 的行为都是基于低内存的条件假设。因此对于如何处理内存不够的情况就尤为关键。因此 Android 开发团队增加了一个额外的

low memory killer 到内核中，它会在默认的 OOM killer 启动前开始生效。Android 的 low memory killer 使用的策略被描述在应用开发文档中，长期没有使用的进程或者优先级低的进程会被干掉。

OOM 的调节范围为-17~15，数字越高就意味着进程更有可能被干掉以释放资源。Android 的处理方式和 OOM 调节级别不同，Android 是按照进程正在运行的组件来表示它们的类型，每一种类型都有一个启动 killer 的阀值。OOM 只有在系统内存被耗尽时才启动，而通过阀值控制，则只要阀值达到 killer 就启动。

Binder

Binder 是一种 RPC/ IPC 机制类似于 Windows 中的 COM。它的历史非常悠久可以追溯到 BeOS 被 Palm 收购之前的时代。Binder 最终是以 OpenBinder 项目名义发布的。Android 中的 Binder 是受前人工作的启发，但是实现方式上却不是从原来的代码中继承来的。相反，它是重写了 OpenBinder 的部分功能。如果想要了解这种机制的底层框架和设计理念的话就必须阅读 OpenBinder Documentation。

本质上，Binder 提供的是在操作系统上层提供一个远程对象调用的功能。换句话说，代替重新设计传统的操作系统，Binder 试图避开和超越它们。因此开发人员即获得了把远程服务当成一个对象来处理的好处，又不必面对一种新的操作系统。因此，扩展系统的功能变得非常容易，只需要通过添加远程调用对象，而不用为了提供新的服务开发一个新的守护进程。远程对象能以任何熟悉的语言进行开发，并且既可以与其它远程服务共享一个进程空间，也可以拥有独立的进程。要调用远程对象则只需要知道接口定义和该对象的一个引用。

正如在图 2-1 中看见的，Binder 是 Android 架构中的一块基石。它使得应用程序可以和系统服务器进行通讯，也能使应用程序可以和其它服务组件进行通话。当然，应用程序开发人员并不是直接和 Binder 对话，而是通过 aidl 工具生成的接口和存根来调用 Binder。

Binder 机制中的内核驱动部分是一个字符驱动，通过访问/dev/binder.文件来实现。在参与通信的双方是通过调用 ioctl()函数来传递数据块的。这里面会涉及到 “Context Manager” 这一概念。后面会详细解释 “Context Manager”。

匿名共享内存(ashmem)

在大多数操作系统中另一个 IPC 机制是共享内存。在 Linux 中，通常是采用 POSIX 的 SHM 功能，但也有部分使用的是 System V IPC 机制。但是，如果看看 AOSP 中的 NDK/ DOCS /system/ libc/ SYSV IPC.html 文件，将发现 Android 开发团队似乎并不喜欢 SysV IPC。实际上，在那个文件中谈到 SysV IPC 机制可能导致内核中的资源泄漏，也可能被恶意软件攻击导致系统瘫痪。

尽管 Android 开发团队或者 ashmem 的相关文档没有明确说明，但是 ashmem 很可能并没有 SysV IPC 的缺陷，它更类似于 POSIX 的 SHM，但是在行为上有不同。比如，ashmem 会对引用进行计数，只有在所有引用了该内存的进程退出后，它才会被销毁。并且在系统需要内存的时候，ashmem 可以对内存进行压缩。Ashmem 有 2 种模式，当处于“Unpinning”时，可以对内存进行压缩，如果是“pinning”时，则不能。

一般 ashmem 使用方式是这样的，第一个进程使用 ashmem 创建一块共享内存区域，并且使用 Binder 共享对应的文件描述符给需要共享该内存区域的进程。比如，Dalvik's JIT 代码缓存就通过 ashmem 共享 Dalvik 实例。许多系统服务器组件，比如 surface Flinger 和音频 Flinger 通过 Imemory 接口来使用 ashmem。（IMemory 是 AOSP 的一个内部的接口，对应用程序开发人员不可见。应用程序开发使用 MemoryFile 类来执行类似的功能。）

告警-Alarm

被加入到内核中的告警驱动是默认内核提供的功能不能满足 Android 需要的又一个例证。Android 的报警驱动程序实际上位于内核提供的实时时钟（RTC）和高精计时器（HRT）之上的。内核的 RTC 功能为驱动开发人员提供了一个开发框架，以便开发特定 RTC 功能。而内核通过主 RTC 驱动暴露一个单一的硬件无关的接口。另一方面，内核的 HRT 的功能允许调用者在某个特定时间点醒来。

在"vanilla"内核的 Linux 中，应用程序开发人员通常使用 setitimer（）系统调用函数来获得一个给定时间值到期信号。这个系统调用函数运行使用多个定时器类型，其中之一就是 ITIMER_REAL,它是内核中高精度定时器（HRT）。但是这个功能在系统处于挂起的状态时是无效的。换句话说，就是如果应用程序使用 setitimer()来请求指定时间的唤醒，但是在中间时间点，系统被挂起，则该应用程序只能在设备被再次唤醒时才能获得它想要的信号。

与 setitimer( )系统调用函数不同 ,内核的 RTC 驱动是通过把文件 /dev/rtc 作为参数赋给 ioctl()函数来调用的，它会设置一个由系统中的 RTC 硬件设备触发的告警。该告警不管系统是否被挂起都会被触发。因为即使系统处于挂起状态，但是 RTC 设备也会处于激活的状态中。

Android 的告警驱动巧妙地结合了两者的优势。默认情况下，驱动使用内核的 HRT 为用户提供告警功能，但是如果系统将要处于挂起状态时，驱动就会变成 RTC 以便系统在合适的时间内个被唤醒。因此，无论什么时候用户空间中的应用程序需要告警，都需要使用 Android 的告警驱动去获得唤醒，而不管系统是否会在中间被挂起。

从用户空间看，告警驱动是以/dev/alarm 字符设备的形式出现，并且运行用户通过 ioctl()函数来建立告警和调整系统时间。

有几个依靠/dev/alarm 的关键 AOSP 组件 ,比如 ToolBox 和 SystemClock 类，都可以通过应用开发 API 访问，它们依赖/dev/alarm 来设置或获得系统时间。不过，最重要的是，系统服务器中的报警管理服务部分使用/dev/alarm 来为应用程序提供告警服务，开发人员通过 AlarmManager 类来获取该服务。当告警被触发时，应用程序获得了在系统挂起之前处理自身事务的机会。

日志

日志是 Linux 的另一个核心部件。通过事后或者实时分析系统日志可以发现错误和告警，这对于隔离关键错误，尤其是临时错误至关重要。默认情况下，大多数 Linux 发行版包括两个日志系统：内核自己的日志，通常可以通过 dmesg 命令访问；系统日志，通常存储在/ var / log 目录中的文件。内核日志的内容是内核中的核心内核代码或者设备驱动程序通过调用各种 printk()函数打印出的消息。系统日志包含的消息来自于系统中各种正在运行的守护程序和工具。事实上，可以使用 logger 命令发送自己的消息到系统日志。

考虑到 Android，内核日志的功能是一样的。然而在通常 Linux 中包含的系统日志软件包在 Android 中都没有。相反，Android 在添加到内核中的 Android 日志驱动的基础上定义了自己的日志机制。系统日志依赖通过套接字发送消息，并因此产生一个任务切换。它也使用文件来存储其信息，因此产生写入到存储设备的动作。相比之下，Android 的日志功能管理一些独立的内核缓存以记录用户空间来的日志数据。因此，记录每个事件都没有任务切换或文件写操作。相反，驱动程序保持循环缓冲区，记录每一个传入的事件，并立即返回给调用者。

由于其轻量级，高效的设计，用户空间中的组件可以在运行时使用 Android 的日志功能来记录有规律的日志事件。事实上，应用程序开发人员可以通过日志类（Log Class）或多或少直接调用 Logger 驱动写主要事件缓存。很显然，一切良好的东西可以被滥用。

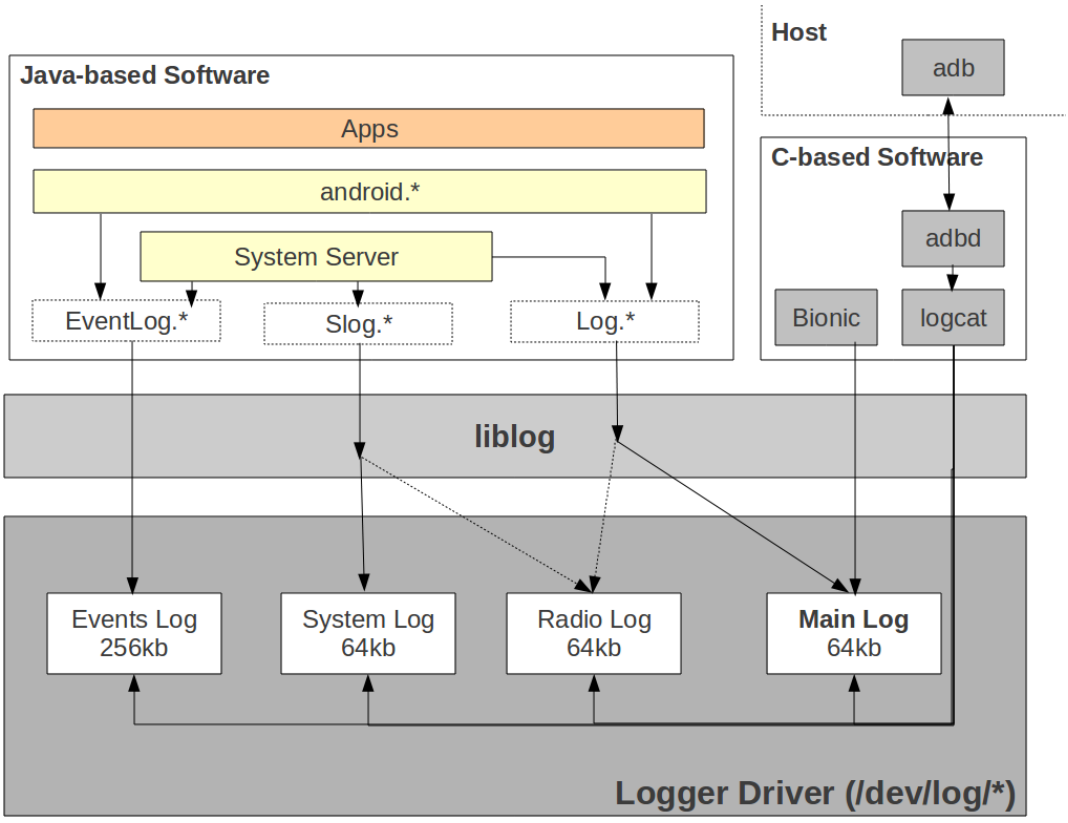

图 2-2 系统日志框架

图 2-2 更详细的描述了 Android 的日志框架。正如看到的，日志驱动是所有和日志相关功能的基础。日志驱动管理的每一个缓存在/dev/log/下被暴露成一个个独立的入口。然而，没有用户空间组件直接和驱动程序交互。相反，它们都依赖于 liblog 提供了许多不同的日志功能。根据使用的功能和传递的参数，事件将被记录到不同的缓冲区。比如，Log 类和 Slog 类使用 liblog 函数测试是否来自于无线相关模块的事件被分发。如果是的，事件被送到“radio”缓存中，如果不是，则 Log 类将把事件发送到“main”缓存中，Slog 类把事件发送到“system”缓存中。“main”缓存通过不带参数的 logcat 命令查看。

Log 类和 EventLog 类都可以通过应用 API 进行调用，但是 Slog 只用于 AOSP 内部。不过尽管应用开发人员可以使用 EventLog，但是在官方文档中明确指出 EventLog 主要是被系统集成商用，而不是应用开发人员使用。Log 类的使用可以参考各种在线文档和代码示例。通常，系统组件使用 EventLog 把二进制事件记录到 Android 的“event”缓存中。一些系统组件，尤其是系统服务器

托管服务，将 Log，Slog 和 EventLog 结合使用记录不同的事件。比如，与应用程序开发相关的事件会使用 Log 记录，与平台开发商或系统集成商相关的事件可能会用 Slog 或者 EventLog 记录。

需要注意的是 logcat 的工具依赖于 liblog 层，该工具主要被应用开发人员用于转储 Android 的日志。Liblog 不仅提供对日志驱动的访问，同时也起到了把事件格式化为合适的输出文本并过滤的作用。Liblog 的另一个特点就是，它记录的每一个事件都带有优先级 ,标签和数据。优先级有 4 种 :verbose、debug、info、warn、或 error。标签是一个唯一的字符串，用于标示写日志的组件或模块，而数据是实际需要被记录的信息。

拼图的最后一块是 adb 命令。正如我们将在后面讨论，AOSP 包括一个在 Android 设备中运行的守护进程-Android 调试桥（ADB），可以在调试主机上使用 adb 命令行工具进行访问。当在主机上敲击 adb logcat 命令，守护进程启动目标机本地的 logcat 命令，转储“main”buffer，然后传回主机并显示在终端上。

其它重要的 Android 组件

偏执网络（Paranoid Networking）

通常在 Linux 中，所有的进程都可以创建套接字，并与网络交互。然而按照 Android 的安全模式，访问网络的能力必须被控制。因此，有一个选项添加到内核以便对套接字的创立和网络的访问进行控制 ,该选项会对当前进程是否属于一组特定的进程或具有某些功能进行判定。这一原则适用于 IPv4 ,IPv6 和蓝牙。

RAM Console

正如前面提到的，内核管理自己的日志，并可以通过 dmesg 命令进行该日志的访问。这个日志的内容非常重要，它通常包含驱动程序和内核子系统的关键信息。当出现系统崩溃时，这个日志有助于做事后分析。因为相关的信息有可能在重启后就丢失了 ,Android 增加了一个驱动程序注册了基于 RAM 的控制台程序，即使重启这个控制台仍然存在，因此可以通过访问/proc/last_kmsg 文件来获取日志内容。

Physical Memory (pmem)

与 ashmem 一样，PMEM 驱动程序允许在进程间共享内存。然而，不像 ashmem ,PMEM 允许共享大块的连续的物理上的内存区域 ,而不是虚拟内存。此外，这些程序和驱动程序之间可以共享内存区域。例如 G1 手机，PMEM 堆用于 2D 硬件加速。但是请注意，PMEM 还没有在所有的设备中使用。

硬件支持

Android 的硬件支持的做法是与 Linux 内核和基于 Linux 的发行版的经典做法完全不同。具体来讲，不同点在于，硬件支持的实现方式，建立在硬件支持上的抽象和对于结果代码的许可和发布是完全不同的。

Linux 方法

Linux 提供支持新的硬件的通常做法是为该硬件开发驱动程序，并把驱动程序作为内核的一部分，或者在运行时作为模块加载。在用户空间一般可以通过/dev entry 访问到相应的硬件。Linux 的驱动程序模型定义了三种基本类型的设备：字符设备，字节流形式的块设备（基本上是硬盘）和网络设备。多年来，已添加相当多的额外的设备和子系统类型 ,如 USB 或 MTD 设备。然而 ,/dev entry 都会对应到一个特定的设备类型，与/dev entry 进行交互的 API 函数和方法已经非常标准化和稳定了。

因此，使得各种软件栈必须建立在/ dev 节点之上，以便直接与硬件交互，或者暴露一些通用 API 给应用程序已提供对硬件的访问能力。其实，绝大多数 Linux 发行版都有一组类似的核心库和子系统，如 ALSA 音频库和 X Window 系统，都是通过/dev 与相应的硬件设备进行交互。

许可和发布方面，“Linux”的一贯方针就是驱动程序应该被合并进主线，并作为主线的一部分进行维护，和主线内核一起按照 GPL 条款进行发布。尽管一些设备驱动程序是独立开发和维护的，并采用其它的许可进行发布，但是这些做法都不是首选的方法。事实上，非 GPL 的驱动程序一直是一个有争议的问题。因此，传统上如果用户和分销商要想得到最新的驱动程序就最好从 http://kernel.org 网站上获得最新的主线内核。

Android 的通用做法

虽然 Android 建立在内核的硬件抽象和能力的基础上，但其做法是非常不同的。从技术角度看，，最明显的区别是其子系统和库不依赖于标准的/ dev 条目才能正常工作。相反，Android 的堆栈通常依赖于制造商所提供的共享库与硬件交互。实际上，Android 依赖一个硬件抽象层（HAL），而不同类型的抽象硬件的行为和功能都是有很大的不同。

此外，大多数 Linux 发行版中与硬件交互的软件栈在 Android 中都没有。

例如 ,Andriod 中没有 X Window 系统 ,并且对 ALSA 的功能访问和标准 Linux 发布版本不一样。Android 中的 ALSA 驱动程序由硬件厂商提供共享库去实现对 HAL 的音频支持。

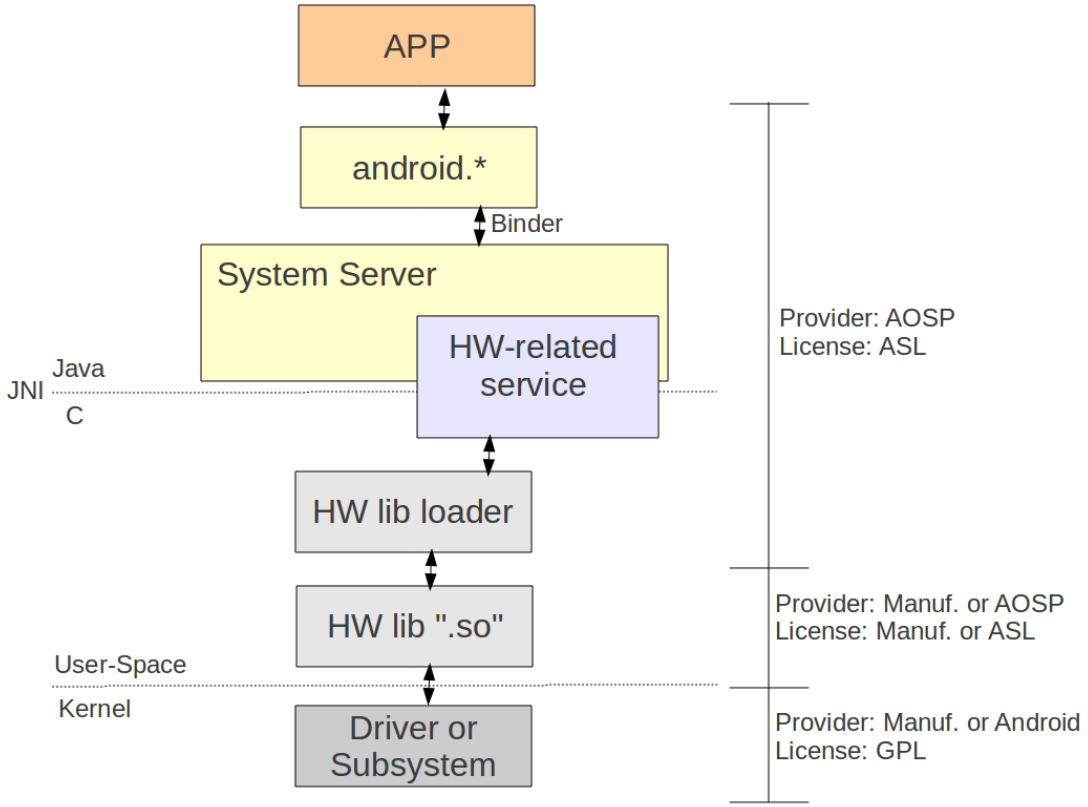

图 2-3 Android 硬件抽象层 （HAL）

图 2-3 给出在 Android 中的硬件抽象和支持的典型方式，以及相应的发布与许可。正如看到的，Android 仍然最终依赖于内核中的共享库来访问硬件。共享库由设备制造商或 AOSP 提供。

这种方法的主要特征之一就是发布的共享库使用许可依赖与硬件制造商。因此，设备制造商可以创建一个简单的设备驱动程序实现访问特定的硬件，该驱动程序处 GPL 许可之下。驱动程序通过 mmap()和 ioctl（ ）函数把硬件暴露给用户空间。

其实 Android 不指定共享库如何与驱动程序或内核子系统互动。Android 只指定共享库提供哪些 API 给上层。因此，由开发人员来确定最适合硬件的，具体驱动程序接口， 同时开发人员提供的共享库要实现合适的 API。在下一节中，我们将涉及 Android 用于硬件接口的典型方法。

Android 相对不一致的是用于硬件支持的共享库是由上层加载的。对于大多数硬件类型，必须有一个.so 文件，该文件要么由 AOSP 提供，或者由开发人员自己提供，否则功能不能正常运行。

无论用哪种机制来加载提供硬件支持的共享库 ,都有一个与特定硬件类型对应的系统服务被用来加载共享库并与之通信。该系统服务将负责与其他系统的服务进行交互和协调 ,使硬件的行为与该系统的其余部分和暴露给应用程序开发者的 API 相一致。如果增加一个特定硬件类型的支持，至关重要的是尽可能详细的了解内部硬件对应的系统服务。通常，系统服务会被分成两部分，其中一部分用 Java 实现大部分 Android 的具体功能 ,而另一部分用 C 来实现对共享库和其它底层函数的交互。

加载和接口方法（Loading and Interfacing Methods）

正如前面提到的，系统服务和 Android 有各种方式与共享库和硬件设备进行交互。很难完全理解为什么有这样多种的方法 ,幸运的是 ,方法正在趋于统一。

请注意，这里描述的方法并不一定是相互排斥的。它们常常结合使用。

dlopen()—loading through HAL

用于 GPS, Lights，Sensors 和显示

libhardware 库加载某些硬件共享库。Libhardware 库是 Android HAL 中的一部分，它提供 hw_get_module()函数给系统服务和子系统使用，用于加载一个特定的硬件共享库（在 HAL 中又被称为模块）。hw_get_module（）又后调用经典的 dlopen（）函数把库加载到调用者的地址空间。

Linker-loaded .so files

用于 Audio，Camera，wifi，震动器和电源管理。

在某些情况下，系统服务仅仅是在编译的时候链接一个.so 文件。因此，当相应的二进制运行时，动态连接器会自动加载共享库到进程的地址空间中。

Hardcoded dlopen()

用于 StageFright 和无线接口层(RIL)

在少数情况下，程序不通过 libhardware 库而直接调用 dlopen（）来获取共享库。使用这种方法的理由不得而知。

套接字-Sockets

用于蓝牙，网络管理，磁盘加载和无线接口层(RIL)

系统服务和框架组件有时使用套件字和远端与硬件进行交互的守护进程或服务进行通话。

Sysfs entries

用于震动器和电源管理

Sysfs（/sys）中的某些条目用于控制硬件和内核子系统的行为。在某些情况下，Android 使用这种方法，而不是/dev 条目来控制硬件。

/dev 节点

用于几乎所有的类型

可以说，任何硬件抽象必须在某些时候与/ dev 目录中的条目进行通信，主要是因为驱动程序是被暴露给用户空间的。有时这些通信对于 Android 是不可见的，因为它是直接和共享库进行交互。有时 AOSP 组件直接访问设备节点。被输入法管理器使用的 input 库就是这种情况。

D 总线 D-Bus.

用于蓝牙

D-BUS 是一个大多数 Linux 发行版中都有的典型消息传递系统，用于促进各种桌面组件之间的通信。Android 中的非 GPL 组件也用 D 总线与 Android 中的蓝牙协议栈(BlueZ)进行通话 ,(BlueZ 也是 Linux 中的默认蓝牙协议栈)。D-Bus 是双许可-AFL 和 GPL。要了解更多 D-Bus 请查看 http://dbus.freedesktop.org。

设备支持细节

表 2-1 总结了 Android 支持的每种硬件类型。你会发现，有多种机制和接口组合的组合。如果打算实现对特定类型硬件的支持，最好的办法是从现有的示例开始实现。AOSP 通常包括一些旗舰手机的硬件支持代码。有时硬件支持来源是非常广泛的。

唯一不能公开可用硬件类型是无线接口层，这个需要开发自己实现。由于种种原因，不能让人人都可以操作电波，因此制造商不提供这样的实现。相反，如果想自己实现一个 RIL，谷歌提供了一个 RIL 实现的参考。

Table 2-1. Android's 硬件支持方法和接口

硬件	系统服务	用户空间硬件支持接口	硬件接口
Audio	Audio Flinger	链接加载 libaudio.so	由硬件制造商决定，通常是 ALSA（高级 Linux 声音架构）.
Bluetooth	蓝牙服务	Socket/D-Bus to BlueZ	BlueZ stack
Camera	Camera Service	链接加载 libcamera.so	由硬件制造商决定，通常采用 Video4Linux
Display	Surface Flinger	HAL-加载 gralloc 模块	/dev/fb0 or /dev/graphics/fb0
Input	Input Manager	原生库	Entries in /dev/input/
Lights	Lights Ser	HAL-加载 lights 模块	由硬件制造商决定

Media	N/A, StageFright 媒体服务中的框架	dlopen on libstagefrighthw.so	由硬件制造商决定
Network interfaces	Network Management Service	Socket to netd	ioctl() on interfaces
Power Management	Power Manager Service	链接加载 libhardware_legacy.so	Entries in /sys/android_power/ or /sys/power/
Radio (Phone)	N/A, entrypoint is telephony Java code	Socket to rild, which itself does adlopen()on manufacturer-provided .so	由硬件制造商决定
Storage	Mount Service	Socket to vold	System calls
Sensors	Sensor Service	HAL-加载 Sensor 模块	由硬件制造商决定
Vibrator	Vibrator Service	链接加载 libhardware_legacy.so	由硬件制造商决定
Wifi	Wifi Service	链接加载 libhardware_legacy.so	Classic wpa_supplicantb

原生用户空间

现在，已经介绍完了 Android 低层的内容，让我们开始往上介绍相关的内容。首先，将介绍 Android 的原生用户空间环境。 “原生用户空间”的意思是所有在 Dalvik 虚拟机以外运行的用户空间组件。这包括了不少运行在目标机 CPU 架构上的二进制代码。这些二进制代码要么是自动启动的，要么是初始化进程根据配置文件启动的，或者是开发人员通过命令行启动的。这些二进制文件可以直接访问根文件系统和操作系统中包含的原生库。它们的这种能力不会受到像 Android 框架对普通应用程序施加的那样限制的影响。

请注意，Android 的用户空间的设计是全新的，和标准的 Linux 发布版本大大的不同。因此，下面会试着解释 Android 的用户空间到底与其它 Linux 系统有哪些不同？哪些相似点？

文件系统布局

像任何其他 Linux 一样，Android 使用根文件系统存储应用程序，库和数据。然而，不同的是 Android 的根文件系统的布局与文件系统层次标准不一致（FHS-是一个社区的标准，介绍了 Linux 根文件系统内各种目录的内容和用途。）。内核本身没有要求必须使用 FHS，但是绝大多数为 Linux 开发的软件包都是假

设这软件运行的根文件系统都是与 FHS 一致。因此，如果打算把准的 Linux 应用程序移植到 Android，首先要做的工作就是确保软件依赖的文件路径仍然是有效的。

由于大部分在 Android 用户空间中运行的软件包都是专门为 Android 开发的，因此不一样的文件系统布局对 Android 本身没有什么影响。后面会谈到，这样的做法反而还有一些好处。不过，重要的是要学会如何浏览 Android 的根文件系统。如果不出意外，很可能要花费相当长的一段时间才能完成 Android 的定制化工作以确保它能运行在自己的的硬件上。

Android 经常使用 2 个主要的文件夹/system 和 /data。这些目录并不来自 FHS。事实上 ,任何主流的 Linux 发行版都不使用这些文件夹。这些都是 Android 开发团队自己的设计。这暗示了可以把 Android 和一个普通的 Linux 并行托管到同一个 root 文件系统中。后面会详细介绍这种可能性。

/system 是主要的 Android 夹，用于存储在编译 AOSP 中产生的一些固化组件。包括原生二进制文件，原生库和框架包以及 stock 应用程序。它是来自于根文件系统中的一个独立镜像，而根文件系统又是来自于一个 RAM 磁盘镜像。/data 是另一个主要的 Android 文件夹，主要保存数据和变换的应用程序。包括用户安装的应用程序所产生和存储的数据，以及在运行时由 Android 系统组件所产生的数据。它也通常由自己独立的映像所加载。

Android 也包含一些 Linux 常见的文件夹，比如：/dev, /proc, /sys, /sbin, /root, /mnt, /etc 等。这些文件夹的作用和 Linux 中的类似，但并不完全相同。

有趣的是，Android 不包括任何/ bin 或/ lib 文件夹。这些文件夹在 Linux 系统中通常是关键的，它们分别包含的是必要的二进制文件和二进制库。这也是 Android 系统可以和标准的 Linux 组件共存的一个因素。

对于 Android 根文件系统要更详细的介绍。例如，刚才提到的文件夹包含它们自己的层次结构。另外，Android 的根文件系统包含其他这里没有提及的文件夹。在第 5 章的时候，会更加详细的介绍根文件系统。

库-Libraries

Android 依赖于大约一百个动态加载库，它们都存储在/ system/ lib 文件夹中。其中一些库来自于一些外包项目然后被合并入 Android 代码库中，这样就能在 Android 中使用它们的功能。但是大部分的库是 AOSP 自己开发的。表 2-2 列出了在 AOSP 中从外部项目并入的库，而表 2-3 总结了 AOSP 内自己开发的库。

Table 2-2. 来源于外部项目的库文件

库名	外部项目名	最初位置	许可
libcrypto.so and libssl.so	OpenSSL	http://www.openssl.org	Custom, BSD-like
libdbus.so	D-Bus	http://dbus.freedesktop.org	AFL and GPL
libexif.so	Exif Jpeg header manipulation tool	http://www.sentex.net/~mwandel/jhead/	Public Domain
libexpat.so	Expat XML Parser	http://expat.sourceforge.net	MIT
libFFTEm.so	neven face recognition library	N/A	ASL
libicui18n.so and libicuuc.so	International Components for Unicode	http://icu-project.org	MIT
libiprouteutil.so and libnetlink.so	iproute2 TCP/IP networking and traffic control	http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2	GPL
libjpeg.so	libjpeg	http://www.ijg.org	Custom, BSD-like
libnfc_ndef.so	NXP Semiconductor's NFC library	N/A	ASL
libskia.so and libskiagl.so	skia 2D graphics library	http://code.google.com/p/skia/	ASL
libsonivox	Sonic Network's Audio Synthesis library	N/A	ASL
libsqlite.so	SQLite database	http://www.sqlite.org	Public Domain
libSR_AudioIn.so and libsrec_jni.so	Nuance Communications' Speech Recognition engine	N/A	ASL
libstlport.so	Implementation of the C++ Standard Template Library	http://stlport.sourceforge.net	Custom, BSD-like
libttspico.so	SVOX's Text-To-Speech speech synthesizer engine	N/A	ASL
libvorbisidec.so	Tremolo ARM-optimized Ogg Vorbis decompression library	http://wss.co.uk/pinknoise/tremolo/	Custom, BSD-like
libwebcore.so	WebKit Open Source Project	http://www.webkit.org	LGPL and BSD
libwpa_client	Library based on wpa_supplicant	http://hostap.epitest.fi/wpa_supplicant/	GPL and BSD
libz.so	zlib compression library	http://zlib.net	Custom, BSD-like

Table 2–3. Android-specific libraries generated from within the AOSP

类型	库	描述
仿生-Bionic	libc.so	C library
	libm.so	Math library
	libdl.so	Dynamic linking library
	libstdc++.so	Standard C++ library
	libthread_db.so	Threads library
核心-Core	libbinder.so	The Binder library
	libutils.so, libcutils.so, libnetutils.so, and libsysutils.so	Various utility libraries
	libsystem_server.so, libandroid_servers.so, libaudioflinger.so, libsurfaceflinger.so,	System-services-related libraries
	linsensorservice.so, and libcameraservice.so	Client libraries for certain system services
	libcamera_client.so and libsurfaceflinger_client.so	The PixelFlinger library
	libpixelflinger.so	
	libui.so	Low-level user-interface-related functionalities, such as user input events handling and dispatching and graphics buffer allocation and manipulation
	libgui.so	Sensors-related functions library
	liblog.so lib	The logging library
	android_runtime.so	The Android runtime library
虚拟机-Dalvik	libdvm.so	The Dalvik VM library
	libnativehelper.so	JNI-related helper functions
硬件-Hardware	libhardware.so	The HAL library that provides hw_get_module() uses dlopen() to load hardware support modules (i.e. shared libraries that provide hardware support to the HAL) on demand.
	libhardware_legacy.so	Library providing hardware support for wifi, power-management and vibrator
	Various hardware-supporting shared libraries.	Libraries that provide support for various hardware components, some of which are loaded using through the HAL, while others are loaded automatically by the linker
媒体-Media	libmediaplayerservice.so	The Media Player service library
	libmedia.so	The low-level media functions used by the Media Player service
	libstagefright*.so	The many libraries that make-up the StageFright media framework

	libeffects.so and the libraries in the soundfx/ directory libdrm1.so and libdrm1_jni.so	The sound effects libraries The DRM(Digital Rights Management) framework libraries
Open GL	libEGL.so, libETC1.so, libGLESv1_CM.so, libGLESv2.so, and egl/ligGLES_android.so	Android's OpenGL implementation

初始化进程-Init

在内核启动完的时候，它会启动一个初始化进程（Init）。初始化进程负责产生系统中的其它所有执行关键操作的进程和服务，比如重启。通常由 Linux 发布版提供的软件包都是 SystemV init, 但是最近几年一些发行版本开始使用自己的包，比如 Unbantu 使用的就是 Upstart。在嵌入式 Linux 系统中，提供初始化软件的包是 BusyBox。Android 使用的是自己自定义额初始化包，它包含了一些新奇的功能。

配置语言

传统的初始化进程是按照当前的运行级别或者按照用户请求来执行脚本 ,但是 Android 不一样，它定义自己的配置语义并主要依赖于全局属性的变化来触发具体的指令。初始化主配置文件通常存储为 init.rc，同时与设备相关的配置文件被存储为/init.device_name.rc，与设备相关的脚本被存储为 /etc/init.device_name.sh, 这里的 device_name 就是设备的名字。开发人员可以高度控制系统的启动，通过改变这些文件来改变系统启动的行为。比如在启动的时候可以禁用 Zygote，然后再通过 adb 进入系统，手动来启动它。

全局属性

Android 的 init 的一个非常有趣的方面就是管理全局属性集的方式。这些属性集可以从系统的许多地方进行访问和修改，只要有合适的权限。其中一些属性在编译的时候被设置，，有的则是在 init 的配置文件中被设置，还有一些是在运行时设置。有些属性也保存到了磁盘以便永久使用。由于 Init 管理这些属性，它可以检测到它们的任何变化，并触发执行一组配置命令。

比如，前面提到的 OOM 调整,就是在就是在启动的时候，按照 init.rc 文件进行设置。网络属性的设置也是一样。在编译时候进行设置的属性被保存在

/system/build.prop 文件中 ,该文件还保存编译的时间和相关细节。在运行时，系统将有超过一百个不同的属性，范围从 IP 和 GSM 配置参数，到电池的级别等等。可以使用 getprop 指令来获得当前的属性及其值列表。

udev events

正如前文提及的，在 Linux 中对设备的访问是通过访问/dev 文件夹下的节点来实现的。在早期的 Linux 发布版中，在/dev 文件夹下放置了几千个条目以便包含所有可能了设备配置。后来出现了几套方案被用来进行节点的动态创建。目前，系统使用 udev，它依赖运行时由内核在硬件被添加或移除时所产生的事件。

在大多数 Linux 发布版中 ,udev hotplug 事件是由 udevd 守护进程处理。在 Android 中 ,这些事件是由 ueventd 守护进程来处理 ,该进程是 Android 初始化进程的一部分，并通过/sbin/ueventd 中的符号链接可以访问。eventd 依赖/ueventd.rc 和 /ueventd.device_name.rc 文件。

工具箱

就像根文件系统的目录层次结构，大多数 Linux 系统中的/ bin 和/ sbin 目录下有许多必不可少的二进制文件。在大多数 Linux 发行版中，在这些目录中的二进制文件都来自网络上不同站点上的软件包。在嵌入式系统中，并不必须要处理这么多包，也不一定有那么多不同的二进制文件。

经典的 BusyBox 包所采取的方法是建立一个单一的二进制文件，这个文件有很多 switch-case，用于检查命令行的第一个参数并执行相应的功能。所有的命令，都是象征性的链接 busybox 的命令。所以，当你输入 ls，你实际上是调用了 BusyBox。不过由于 BusyBox 的行为根据命令行中的第一个参数执行相应的功能，而该参数是 LS，它的行为就好像从一个标准的 Linux shell 中运行该命令。

Android 不使用 BusyBox，但包括其自己的工具，ToolBox，它同样采用符号链接到 toolbox 命令。但是功能完全不同，如果你以前使用过 BusyBox，那么再用 TooBox 将会比较失望。似乎从许可的角度看时，从头创建一个工具是有意义的，BusyBox 采用 GPL 许可。此外，一些 Android 开发者都表示，他们的目标是创建一个最小化的工具用于从 shell 进行调试，而不是提供一个 BusyBox 的完全替代品。无论如何，ToolBox 是 BSD 许可的，因此，制造商可

以修改和分发它,而无需跟踪开发人员作出的修改或把任何源代码提供给他们的客户。

您可能仍然希望包括 BusyBox，以便从它的功能中受益。如果你因为它的许可而不想将它作为您的最终产品的一部分，那么在开发过程中可以暂时包括，在最终的产品发布阶段把它剥离出来。我们将在后面详细介绍。

Daemons-守护进程

作为系统启动的一部分，Android 的 init 启动几个关键守护进程，它们在整个系统生命周期内都会一直运行。一些后台程序，如 adbd，按命令启动，取决于全局属性的变化。

Table 2-4. Native Android daemons

Daemon	描述
servicemanager	The Binder Context Manager. 作为系统中运行所有 Binder 服务的索引
vold	The volume manager-卷管理器，处理安装卷和图像的挂载和格式化.
netd	The network manager-网络管理器 ,处理 tethering, NAT, PPP, PAN, 和 USB RNDIS.
debuggerd	调试器守护程序。当一个进程崩溃做事后分析时通过仿生连接器来调用。允许 gdb 来连接主机。
Zygote	Zygote 进程。负责启动系统高速缓存，并启动系统服务器。我们将在本章稍后更详细讨论。
mediaserver	媒体服务器，主持大多数媒体相关服务，我们将在本章稍后更详细讨论。
dbus-daemon	D-Bus 消息守护进程。在 D-Bus 用户之间起到连接的作用。有关更多信息，看看它的 man page。
bluetoothd	蓝牙守护进程。管理蓝牙设备。通过 D-BUS 提供服务。
installd	APK 安装守护程序。负责安装和卸载 apk 文件和管理相关的文件系统条目。
keystore	密钥库守护进程。管理加密密钥键值对，存储加密密钥，例如 SSL 证书。
system_server	Android 系统服务器。这个守护进程承载绝大多数在 Android 系统运行的服务。
adbd	ADB 守护进程。管理目标机和主机 adb 命令之间连接的各个方面。

命令行工具

超过 150 个命令行实用工具散落在 Android 的根文件系统。/system/bin/下面包含其中的绝大多数，一些额外的工具位于/system/xbin 和/sbin 中。在/system/bin/中的命令行工具中,有大约 50 个是/system/bin/toolbox 的字符链接。余下的大部分来自于 Android 基础框架，要么是外部项目合并到 AOSP 或者来自于 AOSP 的其它部分。在第五章将会介绍各种 AOSP 中的二进制文件。

Dalvik 虚拟机和 Android's Java

简而言之，Dalvik 是 Android 的 Java 虚拟机。它使 Android 可以运行由 Java 应用程序和 Android 自身组件所产生的字节代码，同时虚拟机也提供了与系统其它部分交互所需的接口和环境，包括原生库和原生用户空间。对于 Dalvik 虚拟机和 Android Java 有很多内容可以介绍，但是在此之前先讲讲 Java 基础。

Java 的历史就不在这里累述了。反正在 Android 冒出来之前，Java 已经很流行了，你需要记住的重点就是，Java 组件和我们以前的开发语言 C、C++是不同的。

根据设计，Java 是一种解释型语言。C/C++是把源代码编译成二进制汇编指令，由与编译器指定架构匹配的 CPU 来执行，Java 源代码被 Java 编译器编译成与架构无关的字节代码，在运行时由字节代码翻译器（就是虚拟机）执行。正是这种机制使得 Java 开发语言具备了早期开发语言所不具备的特征，比如反射和匿名类。与 C/C++不同的是，Java 不仅不需要开发人员跟踪其所分配的对象，反而希望开发人员不管对象回收工作，它自己有垃圾回收器来确保所有的对象在任何代码不再引用它们时被销毁。

在实践层面上，Java 是由几个不同的东西构成：Java 编译器，Java 的字节码解释器通常被称为 Java 虚拟机（JVM）和 Java 开发人员常用的 Java 库。这些东西，开发人员都可以从 Oracle 免费提供的 JDK 中获取。Android 实际上用了 Java 编译器，但是没有用 JDK 中的虚拟机和开发库。Andriod 用 Dalvik 替代了 JVM，用 Apache Harmony project 替代了 JDK 库。Apache Harmony project 是在整个 Apache 项目下。

据 Dalvik 开发者丹·伯恩斯坦说，Dalvik 与 JVM 的区别就在于它是专为嵌入式系统设计的。也就是说运行 Dalvik 的系统一般 CPU 比较慢，RAM 少，没有交换空间，并采用电池供电。

JVM 运行的文件是.class 文件，但是 Dalvik 运行的文件是.dex 后缀的。不过.dex 文件是把 Java 编译器生成的.class 文件用 Android 的 dx 工具再处理后得到的。一个没有压缩的.dex 文件比原始的.jar 文件小 50%，另外一个不同是 Dalvik 是基于注册的，而 JVM 是基于堆栈的。要了解更多的信息，请参看下面的论文“Virtual Machine Showdown: Stack Versus Registers”，作者 Shietal，该文章于 2005 年 6 月 11 日发表。

Dalvik 一个非常值得关注的特性是，自 2010 年以后，Dalvik 包含了为 ARM 芯片服务的 JIT 编译器，这意外者 Dalvik 把字节码转换成了二进制汇编指令，

这些指令可以直接在目标机的 CPU 上运行，而不再是由虚拟机在运行时逐条解释执行。这种转换的结果就是将其存储以供将来使用。因此，应用程序第一次加载的时候，需要更长的时间，一旦完成后，加载和运行就会更快。这里唯一需要注意的是，JIT 不适用于任何 ARM 以外的其他架构。总之目前运行 Android 最快的架构是 ARM。

作为一个嵌入式开发者，让 Dalvik 运行在自己的系统上，不太需要做什么具体的工作。Dalvik 被设计成与架构无关。

Java Native Interface (JNI)

尽管 Java 非常强大和有许多优点，但是它并不能总是运行在真空中，Java 语言开发的代码有时需要和其它语言开发的源代码进行接口。这种情形在 Android 这样的嵌入式环境中尤其常见，它们的底层功能不是太遥远。为此，Java 本地接口（JNI）机制被提供出来。它本质上是其它开发语言的调用接口。类似于.NET/C#中的 pinvoke。

应用程序开发人员有时在 JAVA 代码中用 JNI 调用 NDK 编译的本地代码，在 AOSP 中的大量 Java 代码和组件使用 JNI 与 Android 底层功能进行交互，这些底层功能大多是用 C 或者 C++编写的。比如用 Java 写的系统服务就会调用 JNI 去和匹配的本地代码进行通信，而本地代码会和对应服务的硬件进行交互。

虚拟机完成了 Java 通过 JNI 和其它开发语言通信的绝大部分工作。在表 2-3 中提到的 libnativehelper.so 库就是作为 Dalvik 虚拟机的一部分，它的作用就是进行 JNI 调用。

在本书后面的部分，将有机会实际使用 JNI 进行 Java 和 C 语言的交互。JNI 是 Android 平台工作的核心，它的使用是一个相对复杂的机制，尤其是确保使用适当的调用语义和功能参数。

系统服务

系统服务是 Android 的幕后操控者，尽管在 Google 的应用开发文档中没有明确提及，但是大约有 50 多个重要的系统服务。这些服务共同合作建立起了一个在 Linux 基础之上的面向对象的操作系统。所有系统服务都使用 Binder。前文提到的用户空间都是为系统服务提供支持而设计的。因此了解系统服务的内容，以及它们彼此之间、它们与系统其它部分的交互方法就显得非常关键。这边部分我们在讨论硬件支持的时候提到了一些。

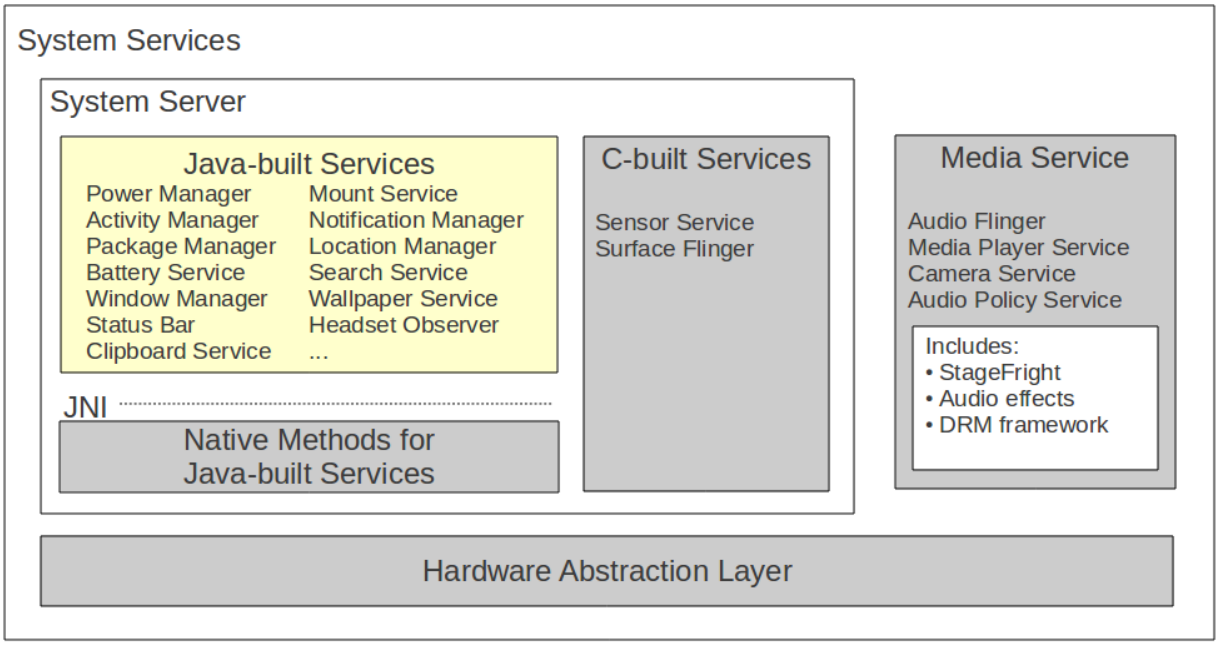

Figure 2–4. System Services

图 2-4 更详细地揭示了在图 2-1 中首次引入的系统服务概念。正如图中显示的，其实有 2 个主要的进程涉及到。最突出的是系统服务器，它的组件都运行在同一个进程中-system_server，其中大部分是由 Java 编编写的服务，有两项服务由 C/ C++编写的。系统服务器还包括一些原生代码允许一些基于 Java 的服务通过 JNI 与 Android 的下层进行交互。系统服务的其余部分是媒体服务，运行它们的进程为 mediaserver。

这些服务用 C/ C++编写的 ,并和相关媒体组件一起被打包 ,如 StageFright 和音频效果。

需要注意的是，尽管只有两个进程，以容纳整个 Android 的系统服务，但是它们都是独立运作，连接它们的服务需要使用 Binder。下面是在 Android 模拟器中使用 service 命令的输出：

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

不幸的是，没有多少文档描述了每个服务是如何运作的。必须通过阅读源代

码才能知道它是如何工作以及如何与其他服务交互。

反向工程源代码

要充分了解 Android 系统服务的内部是非常困难的。光 Java 代码行就超过 8 万 5 千行，分布在 100 多个文件中，这还不算 C/C++代码。雪上加霜的是，注释和设计文档寥寥无几。所以要想深入研究的话，就必须要很有耐心。

有一个小技巧，就是在 Eclipse 中建立一个 Java 项目，然后把源码导入到项目中。这些源码当然不能编译，但是借助 Eclipse 的 Java 浏览功能大大加快源码的理解。例如，你可以打开一个单独的 Java 文件，右键单击源码浏览滚动区域，选择折叠→折叠全部。

当然也可以使用一些商业源码分析工具，比如，Imagix, Rationale, Lattix,或 Scitools。也可以使用开源分析工具，不过这些工具都是用于定位 bug，而不是为了对源码进行反向工程分析用的。

服务管理器和 Binder 交互

正如前文解释的，Binder 机制是系统服务可以进行面向对象的远程方法调用的基础结构。系统中的一个进程如果通过 Binder 调用一个系统服务时，首先要先获取一个句柄。比如，应用开发人员请求电源管理器中的 wakelock 锁时，就必须通过 Binder 调用 Wakelock 内嵌类中的 acquire（）方法。在调用开始之前，需要先获得电源管理服务的句柄。下一节我们将看到，应用开发 API 实际上向开发人员隐藏了它获得句柄的细节。实际上由服务管理器完成了所有系统服务句柄的查询，如图 2-5.

可以把服务管理器看成是系统中所有有效服务的黄页书。如果一个服务没有向服务管理器注册的话，该服务对于系统其它部分就不可见。为了提供索引服务，服务管理器必须在其它服务启动之前被 init 进程先启动。服务管理器起来后会打开/dev/binder 并调用一个特殊的 ioctl（）函数把自己设置成 Binder 上下文管理器（Binder's Context Manager），如图 2-5 中的 A1，因此系统中的任何进程与 Binder ID 0 进行通信的时候，实际上是通过 Binder 与服务管理器（Service Manager）进行通信。

当系统服务器启动时，它会把实例化的每一个服务向服务管理器进行注册。之后，如果应用程序想要和一个服务进行通话时，比如和电源管理服务，应用程序首先向服务管理器请求服务（B1）的一个句柄，之后才能调用服务的方法（B2）。相反，调用一个运行在应用程序内的服务组件时，可以直接使用 Binder（C1），而不用通过服务管理器进行查询。

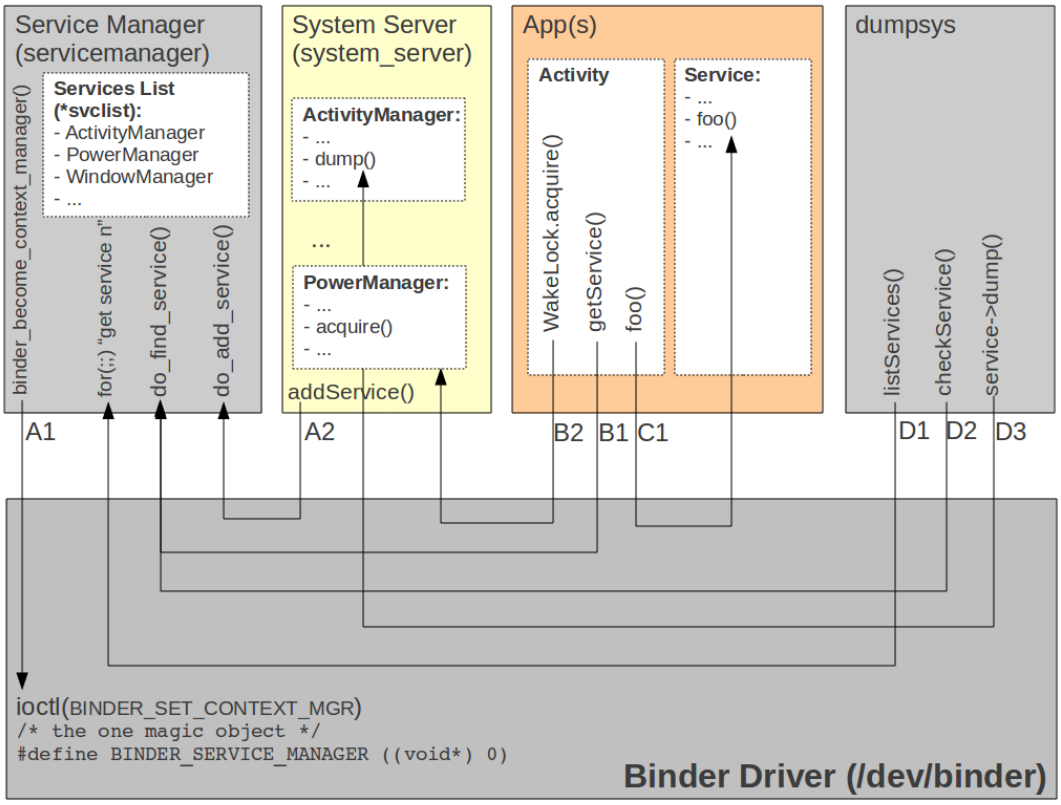

Figure 2-5. Service Manager and Binder interaction

服务管理器也有一种特殊的用法，就是使用 dumpsys 工具，通过该工具可以查询所有或单个系统服务的状态。为了得到所有系统服务的列表，dumpsys 会遍历获得每个服务，在每次迭代中会请求第 n+1 个服务，至到没有服务为止。为了获得每个服务，dumpsys 会请求服务管理器定位具体的服务（D2）。获得服务句柄后，dumpsys 调用服务的 dump()函数，获得服务的状态并显示在终端上。

调用服务

前文所述对用户来说是不可见的，下面是一个代码片段，开发人员使用常规应用程序开发 API 就可以在应用程序中获得 wakelock：

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

请注意这里没有任何关于服务管理器的内容，而只调用 getSystemService（）函数，并传递了一个 POWER_SERVICE 参数给它。在 getSystemService（）函数的内部，实际上是使用了服务管理器去定位电源管理服务，以便我们能

够创建 wakelock，请获得它。

一个服务示例: the Activity Manager

本书虽然无法介绍完每一个系统服务，但是活动管理器是需要研究一下的，它是关键的系统服务之一。活动管理器的源代码分散在 30 多个文件，代码行数超过 2 万行。如果 Android 内部有一个核心的话，活动管理器可以算是。活动管理器负责每个新组件的启动，比如活动和服务，以及内容提供器和消息广播。如果你碰到过 ANR(应用程序没有响应)的对话框，就需要知道，在该对话框背后是活动管理器在进行管理。活动管理器也涉及到 OOM 的调节。

例如，当用户点击一个图标来启动他的主屏幕上的应用程序时，首先发生的是，Launcher 的 Onclick（）回调函数被调用。为了处理该事件，Launcher 通过 Binder 调用活动管理器服务的 startActivity（）方法。服务然后调用 startViaZygote()方法，该方法会打开一个套接字到 Zygote，并请求它启动活动。本章的最后部分将会讲述的更详细。

AOSP 软件包

AOSP 提供了大多数 Android 设备需要的基本软件包。如上一章提到的，一些诸如地图，YouTube 和 Gmail 之类的应用程序并不是 AOSP 的一部分。现在我们一起来看看默认包含的一些软件包。表 2-5 是 AOSP 中的应用程序，表 2-6 中是 AOSP 中的内容提供器，表 2-7 中是 AOSP 中的输入法编辑器。

Table 2-5. Stock AOSP Apps

App in AOSP	显示在 Launcher 中的名字	描述
AccountsAndSettings	N/A	账号管理应用程序
Bluetooth	N/A	蓝牙管理器
Browser	浏览器	默认 Android 浏览器，包含书签部件
Calculator	计算器	计时器应用程序
Camera	照相机	照相机应用程序
CertInstaller	N/A	证书安装 UI
Contacts	通讯录	通讯录管理器
DeskClock	时钟	时钟和闹铃应用程序，包含时钟部件。
DownloadsUI	下载	DownloadProvider UI
Email	电邮	默认 Android 电邮程序
Development	开发工具	多个开发工具
Gallery	看图工具	默认的画廊风格的看图程序
Gallery3D	看图工具	更花哨的画廊风格的看图程序

HTMLViewer	N/A	看 HTML 文件的应用程序
Launcher2	N/A	默认的主屏
Mms	消息	短信/彩信应用程序
Music	音乐播放器	音乐播放器
PackageInstaller	N/A	应用程序安装/卸载 UI
Phone	电话	默认电话拨号程序
Protips	N/A	主屏提示
Provision	N/A	刚出厂时或者恢复出厂设置之后，一步一步引导用户完成各种设置的 Setup Wizard 程序
QuickSearchBox	搜索	搜索应用程序和部件
Settings	设置	设置应用程序，也可以通过主屏菜单访问
SoundRecorder	N/A	录音程序
SpeechRecorder	讲话录音	讲话录音程序
SystemUI	N/A	状态栏

Table 2-6. Stock AOSP Providers

Provider	描述
ApplicationProvider	查询安装好的程序
CalendarProvider	Android 日历
ContactsProvider	Android 通讯录
DownloadProvidera	下载管理、存储和访问
DrmProvider	受 DRM 保护的存储和访问管理
MediaProvider	媒体资源存储和访问
TelephonyProvider	运营商，短信/彩信存储和访问
UserDictionnaryProvider	用户定义的单词字典存储和访问。

Table 2-7. Stock AOSP Input Methods

输入法	描述
LatinIME	拉丁键盘
OpenWnn	日本键盘
PinyinIME	中文键盘

系统启动

把前面介绍的内容融合在一起了解的最好方式就是学习 Android 系统启动。在图 2-6 中，第一个转起来的齿轮是 CPU,它从一个硬编码的地址中获取到第一条执行指令。该地址通常指向具有引导程序的芯片。引导程序首先初始化 RAM,把硬件置于静止状态，加载内核和 RAM 磁盘，然后跳转到内核中。近几年的 SoC 设备（包含 CPU 和必要的外设在一个芯片上），都能直接从 SD 卡或类似

与 SD 卡的芯片上启动。比如最近的 PandaBoard 和 BeagleBoard 都没有板级闪存，因为它们是直接从 SD 卡上启动的。

最初的内核启动是非常依赖于硬件，但其目的是为了设置参数，以便使 CPU 可以尽早的开始执行 C 代码。一旦完成之后，内核就会跳转执行一个与架构无关的 start_kernel（）函数，通过该函数初始化各个子系统，并调用所有内置驱动程序的“init”函数。内核在启动过程中打印的绝大多数消息都是来源于这些步骤。之后内核会加载根文件系统并启动 init 进程。

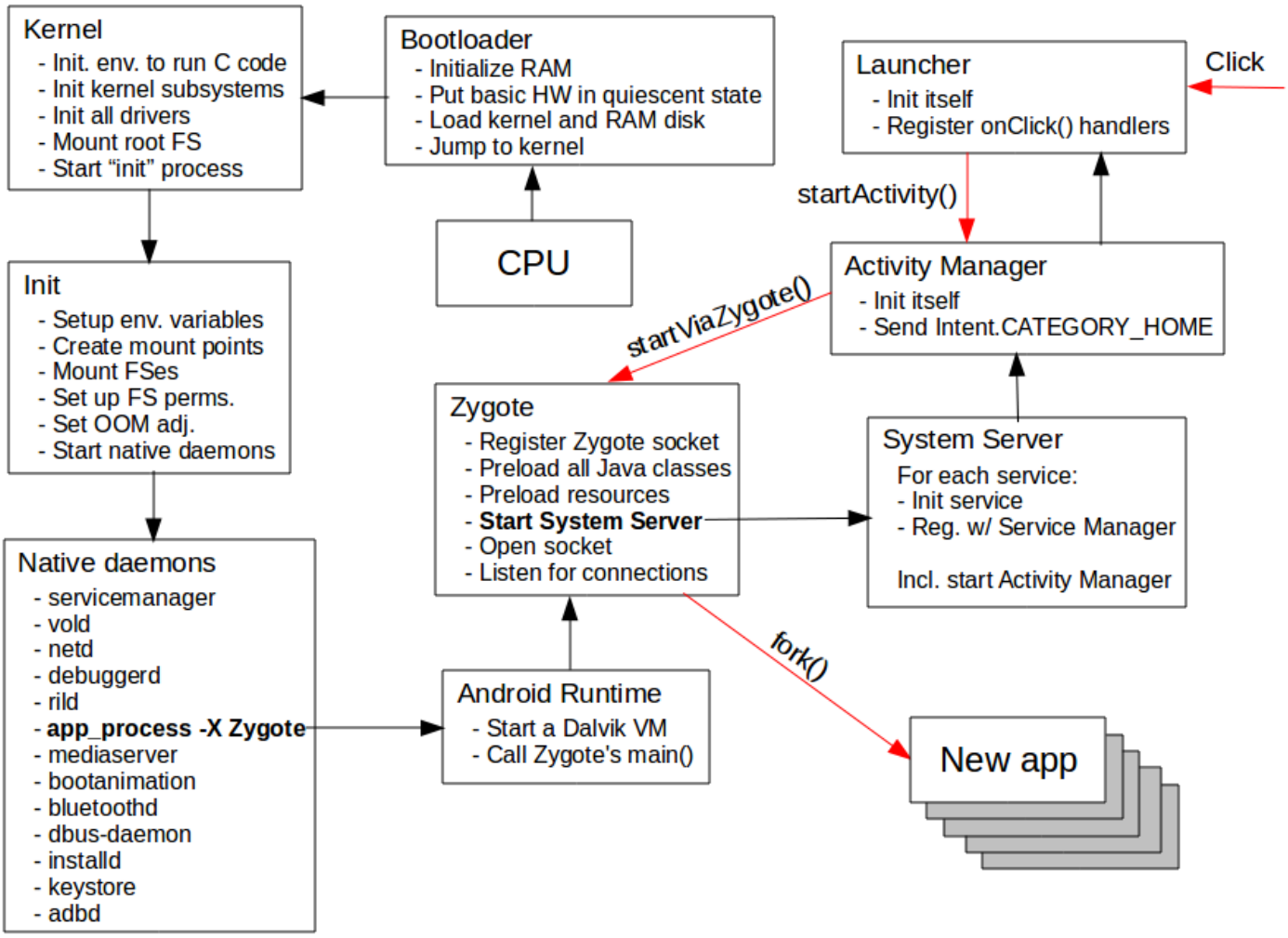

Figure 2-6. Android's boot sequence

Init 进程会执行存储在/init.rc 文件中的指令，该指令是设置诸如系统路径等环境变量，创建挂载点，加载文件系统，设置 OOM 调整并启动本地守护进程。前文中我们也谈到了一些 Andriod 本地的守护进程，在这里我们重点谈一下 Zygote。Zygote 是一个特别的守护进程，它的主要职责是启动应用程序。它的功能主要集中在统一所有应用程序共享组件并缩短它们的启动时间。Init 进程实

际上不直接启动 Zygote，Init 会通过 app_process 命令在 Android 运行时让 Zygote 启动。运行时启动系统第一个 Dalvik 虚拟机，并告诉它调用 Zygote 的 main（ ）函数。

Zygote 只有在一个新的 APP 需要被启动时才会被激活。为了让 APP 启动更快，Zygote 会把 app 运行时可能需要的所有 Java 类和资源预先加载到内存中。然后 Zygote 会监听套件字中的连接(/dev/socket/zygote)以便知道启动新 APP 的请求。当得到启动一个 app 的请求后，Zygote 会创建自己的子进程，并用该子进程启动新的 app。让所有 APP 都从 Zygote 复制的好处就是，虚拟机是全新的并有预先加载好的系统类和资源。换句话说就是，新的 app 不必等待资源加载好后才能运行。

上述的内容之所以有效率是因为 Linux 内核为 fork 实现了写时复制（COW）的策略。正如你可能知道，在 Unix 中 fork 将创建一个新的进程，它是父进程的完全拷贝。使用写时复制（COW）技术，Linux 并没有实际复制任何内容。相反，Linux 把新进程的内存页映射到父进程的内存页上。只有当新进程往内存页进行写操作时，才做复制。但事实上被加载的类和资源从来没有被写过，因为这些类和资源都是默认的，并在系统的生命期内几乎一成不变。因此所有直接从 Zygote 复制的进程实际上都是用的是 Zygote 的映射拷贝。所以无论有多少进程在系统中运行，都只有一份系统内和资源被加载进内存中。

尽管 Zygote 被设计成对连接进行监听以便收到新 app 的请求时创建子进程，但是有一个“app”是 Zygote 完全启动的：系统服务器。它是 Zygote 启动的第一个 app，它是运行在一个完全与父进程独立的进程中。系统服务器会初始化每一个它拥有的系统服务，并把它们注册到在它之前启动的服务管理器中。系统服务器启动的活动管理器（Activity Manager,）会发送一个消息类型（Intent.CATEGORY_HOME）来结束系统服务器的初始化工作。之后会启动 Launcher 程序，显示出所有 Android 用户熟悉的主屏幕。

当用户点击主屏幕上的图标后，系统服务进程激活。Launcher 会请求活动管理器启动应用程序进程，活动管理器会把请求传递给 Zygote，Zygote 会创建子进程，并启动应用程序，应用程序随之展示给用户。一旦系统完成启动，进程列表看起来像下面的情况：

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	mtdblockd
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	afd0b6fc	S	/system/bin/servicemanager
root	28	1	3864	308	ffffffff	afd0bdac	S	/system/bin/vold
root	29	1	3836	304	ffffffff	afd0bdac	S	/system/bin/netd
root	30	1	664	192	c01b52b4	afd0c0cc	S	/system/bin/debuggerd
radio	31	1	5396	440	ffffffff	afd0bdac	S	/system/bin/rild
root	32	1	60832	16348	c009b74c	afd0b844	S	zygote
media	33	1	17976	1104	ffffffff	afd0b6fc	S	/system/bin/mediaserver
bluetooth	34	1	1256	280	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	272	c00b8fec	afd0c51c	S	/system/bin/qemud
shell	40	1	732	204	c0158eb0	afd0b45c	S	/system/bin/sh
root	41	1	3368	172	ffffffff	00008294	S	/sbin/adbd
system	65	32	123128	25232	ffffffff	afd0b6fc	S	system_server
app_15	115	32	77232	17576	ffffffff	afd0c51c	S	com.android.inputmethod.latin
radio	120	32	86060	17952	ffffffff	afd0c51c	S	com.android.phone
system	122	32	73160	17656	ffffffff	afd0c51c	S	com.android.systemui
app_27	125	32	80664	22900	ffffffff	afd0c51c	S	com.android.launcher

app_5	173	32	74404	18024	ffffffff	afd0c51c	S	android.process.acore
app_2	212	32	73112	17032	ffffffff	afd0c51c	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

此输出实际上来自 Android 模拟器，所以它包含了一些模拟器特有的进程，比如 qemud 守护程序。在 Linux 中使用 prctl（）系统调用并带有 PR_SET_NAME 参数，告诉内核改变调用进程的名称。如果有兴趣，看看 prctl（）的使用手册页。还要注意的是由 init 启动的第一个进程实际上是 ueventd。在此之前，所有进程实际上是从内核子系统或驱动程序启动的。

第三章 AOSP Jumpstart

到目前为止，相信大家对 Android 的基本概念已经有了深刻的认识。下面我们开始深入学习一下 AOSP，我们从 http://android.git.kernel.org/获取 AOSP 代码开始学习。在实际开始编译和运行 AOSP 之前，先花点时间探索一下 AOSP 的内容并解释一下前文提到的内容是如何反映到源代码中的。在本章结束的时候会介绍一下在做平台工作的时候会使用到的 2 个重要工具 adb 和模拟器。

AOSP 是一个令人兴奋的软件，包含了大量的创新。并且大家可以下载、修改并以它为基础交付定制化的产品。因此，卷起袖子，让我们开始吧。

获取 AOSP

正如前文提到的，官方的 AOSP 放在 http://android.git.kernel.org，当打开这个网站的时候，会看见大量的可以下载的 git 软件代码库。如果用手工的方式下载会非常繁琐，而且也完全没有必要，因为只有其中一部分是有用的。正确的方法是使用 repo 工具，首先我们需要先获得这个工具。

在 Linux 终端中执行下面的命令：

```
$ sudo apt-get install curl
$ curl https://android.git.kernel.org/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

尽管 repo 的结构是一个 shell 脚本，但实际上它是一个非常复杂的工具，稍后我们再详细研究一下它。现在可以把 repo 就看成是一个下载工具，它可以从多个 git 库中同步下载一个 Android 的发布版本。有一个 XML 格式的约束文件用以描述需要下载的工程项目和它们的位置。Repo 实际上是在 git 库的上层，它下载的每一个项目都是一个独立的 git 库。

容易产生混淆的是这里的约束文件和 android 应用程序中的约束文件是两会事。它们的格式和用途完全不同，不过幸运的是两者不会在同一范围内使用。

现在，已经有了 repo，可以开始取得自己的 AOSP 副本：

```
$ mkdir -p ~/android/aosp-2.3.x
$ cd ~/android/aosp-2.3.x
$ repo init -u git://android.git.kernel.org/platform/manifest.git -b
```

gingerbread
$ repo sync

最后的命令应该运行相当长的一段时间，因为它去获取在 manifest 文件中描述的所有项目的来源。毕竟， AOSP 未编译的情况下大约有 4GB 的大小。因此下载时间和网络带宽与延迟有很大的关系。还要注意的是，我们获取的是一个特定的分支的版本，姜饼。就是第三条命令中的“-b gingerbread”参数决定的。如果省略这个参数就会获得主分支。按照大多数人的经验，主分支往往不能编译和正确的运行，因为它常常包含一些开发分支的最新更新。那些带标签的分支常常工作状态不稳定。

AOSP 内容

现在，已经得到了 AOSP 副本，让我们开始看里面有什么，尤其要与前文提到的内容对应起来。如果太渴望让自己定制的 Android 运行起来，可以先跳过本节，先浏览下一节。表 3-1 总结了 AOSP 的顶层目录。

Table 3-1. AOSP content summary

文件夹	内容	大小（MB）
bionic	Andriod 定制的 C 库	14
bootable	参考启动引导和回复机制	4
build	编译系统	4
cts	兼容测试套件	78
dalvik	Dalvik 虚拟机	35
development	开发工具	65
device	设备相关的文件和组件	17
external	AOSP 中使用的外部项目的副本。	854
frameworks	核心组件-比如系统服务	361
hardware	HAL 和硬件支持库	27
libcore	Apache Harmony	54
ndk	原生开发工具库	13
packages	Android 应用程序，Provider 和 IME	117
prebuilt	预编译二进制文件，包括工具链	1389
sdk	软件开发包	14
system	包含 Android 的“嵌入式 Linux”平台，	32

正如所看到的，prebuild 和 external 是两个最大的文件夹，大约占了 75%的规模。有趣的是，这两个目录包含的内容绝大多数来源于其它开源项目比如，各种 GNU 工具链版本，内核映像，公共库和框架（Openssl 和 webkit 等，libcore

是另外一个开源项目 Apache Harmony。从本质上讲 ,这进一步证明了 Android 是多么依赖于其它开源生态系统中的项目。当然，Android 还是有大约 800MB 的原创代码。

为了更好地了解 Android 的源码，可以再会过头看看第二章的图 2-1，它描述了 Android 的架构。图 3-1 是图 2-1 的变种，它揭示了每一个 Android 组件在 AOSP 源码中的位置。显然，大量的关键部件被存放于 framework/base/ 下面，这个文件夹是大部分 Android “大脑”的存放位置。因此非常有必要研究一下这个文件夹和 system/core/文件夹。表 3-2 和表 3-3 包含了大部分嵌入式开发人员都非常感兴趣需要交互和修改的部分。

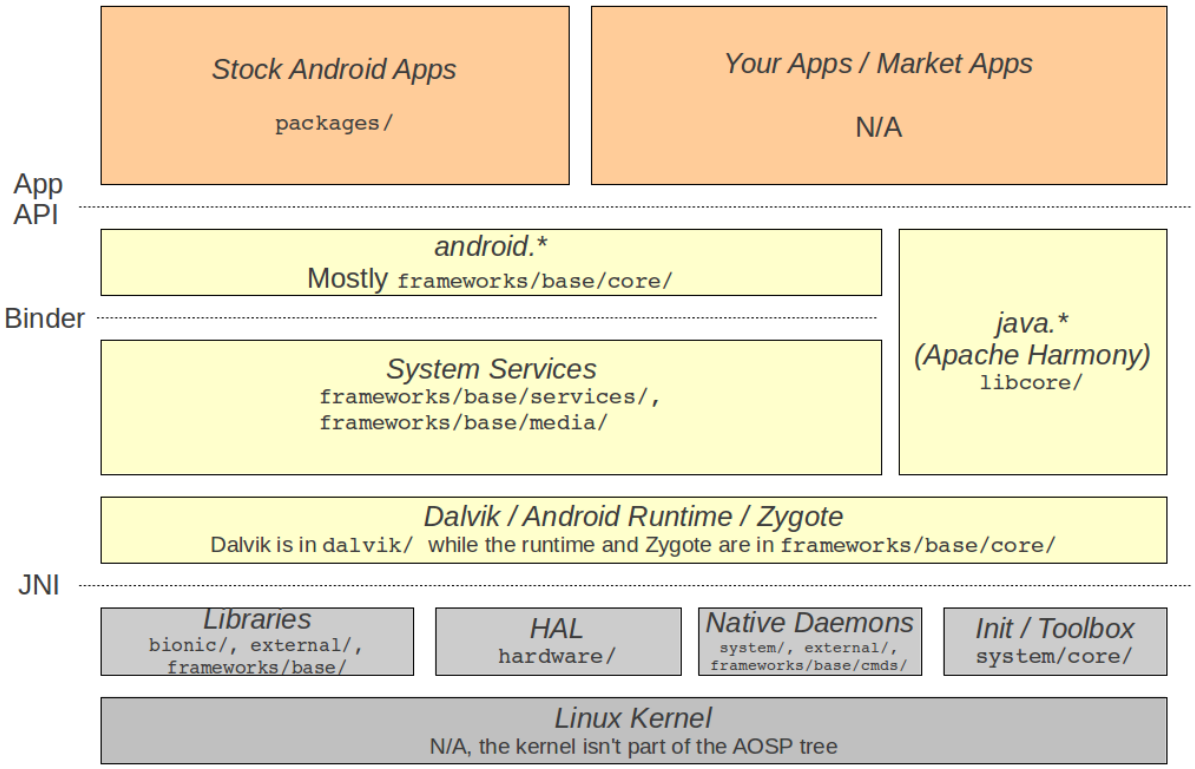

Figure 3–1. Android's architecture

Table 3–2. Content summary for frameworks/base/

文件夹	内容
cmds	原生的命令和守护程序
core	The android.* packages
data	字体和声音文件
graphics	2D 图形和渲染脚本
include	C 语言引用的文件
keystore	安全密钥存储
libs	C 语言库
location	位置 Provider

media	Media Service, StageFright, codecs 等
native	一些框架组件的原生代码
obex	蓝牙 obex
opengl	Opengl 库和 Java 代码
packages	一些核心包比如状态栏
services	系统服务
telephony	Telephony API, 和 rild 无线层接口进行通信
tools	一些核心工具，比如 aapt 和 aidl，
voip	RTP 和 SIP API
vpn	Vpn 管理器
wifi	Wifi 管理器和 API

Table 3-3. Content summary for system/core/

文件夹	内容
adb	Adb 守护进程和客户端
cpio	Mkbootfs 工具用于生成 RAM 磁盘镜像
debuggerd	第 2 章中提到的调试命令
fastboot	Fastboot 工具用于与 Android 引导程序进行通信，使用 fastboot 协议
include	C 语言与系统相关的头文件
init	Android 初始化程序
libacc	Renderscript 使用的类 C 语言的编译器库
libcutils	各种 C 的工具函数，不是标准 C 库的一部分
libdiskconfig	阅读和配置磁盘; vold 使用
liblinenoise	BSD 许可的 readline（ ）的替代函数，来源于 http://github.com/antirez/linenoise，主要由 Android 的 Shell 使用
liblog	日志库
libmincrypt	基本 RSA 和 SHA 功能，恢复机制和 mkbooting 工具会使用。
libnetutils	网络配置库，netd 会使用
libpixelflinger	底层图形渲染功能
libsysutils	与系统和框架中的各种组件进行通话的工具
libzipfile	处理 zip 文件的 zipb 封装，
logcat	Logcat 工具
logwrapper	重定向标准输出和错误输出到 android 日志时，使用的复制和运行命令工具
mkbootimg	创建启动映像的工具，该映像使用 RAM 磁盘和内核

Netcfg	网络配置工具
rootdir	默认 Android 的根文件夹结构和内容
run-as	以一个特定用户 ID 去运行程序的工具
sh	Android shell
Toolbox	Android 工具箱（BusyBox 的替代品）

除了 base/，frameworks/下面还包含一些其他的文件夹，但是它们都没有 base/这么基础。同意的，除了 core/之外，system/下面还包括一些其它文件夹，比如 netd/和 vold/,它们分别包含了 netd 和 vold 守护程序。

除了顶层的文件夹之外，根文件夹也包含了一个 makefile 文件。这个文件几乎是空的，它主要包含了 android 编译系统的入口点。

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

大家可能已经意识到 AOSP 包含的东西比前面介绍的要多的多。在姜饼这个版本中就包含了 14000 多个文件夹和 10 万个以上的文件。按照大多数标准衡量，这是一个相当大的项目。相比之下，早期 Linux 3.0.x 内核版本的目录有大约 2,000 个以及 35,000 个文件。随着本书的推进 ,大家可以学习到更多的 AOSP 的内容。

构建基础知识

目前 AOSP 已经下载完了，并且我们也了解了它里面包含的内容，现在可以尝试让它运行起来。在开始编译 AOSP 之前，我们还需要确认 Ubuntu 中已经把所有必要的包安装好了。下面的所有命令都是基于 Ubuntu11.04。即使使用的是旧的或较新的一些基于 Debian 的 Linux 发行版本 ,指令将是相当类似的。

编译系统建立

首先，得到开发系统上安装一些基本的包。如果因为其它开发工作已经把一些包安装好，那么 Ubuntu 包管理器将会忽略安装这些包的请求。

```
$ 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
```

可能还需要解决几个符号链接：

```
$ 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，需要安装 Sun 的 JDK：

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

构建 Android

现在已经准备好构建 Android。让跳转到 Android 被下载到的目录下并配置构建系统：

```
$ cd ~/android/aosp-2.3.x
$ . 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
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
============================================
```

请注意，先敲击一个. 接着一个空格然后是 build/envsetup.sh。这迫使 shell 在当前 shell 中运行 envsetup.sh 脚本。如果只运行脚本，shell 会催生一个新的 shell，并在新的 shell 中运行脚本。如果在新的 shell 中运行这个脚本的话就对构建 Android 没有用。

我们将在后面详细研究 envsetup.sh 和 lunch。不过，就目前而言，需要注意的是 generic-eng 组合，这样我们可以配置构建系统以便创建可以在 Android 模拟器中运行的镜像。这是与应用程序开发人员测试他们应用程序时所使用的 QEMU 模拟器相同，尽管这里运行的是我们自己定制的镜像，而不是随 SDK 附带的默认镜像。这也是 Android 开发团队在没有真实设备情况下，开发 Android 所使用的同一个仿真器。因此，虽然它不是真正的硬件，也不是一个完美的目标机，但是也足以满足我们的开发需求。一旦目标机确定下来，只要按照本书的内容进行修改就可以把定制的 Android 镜像加载到定制设备中，并通过硬件将其启动起来。

现在环境以及设置好，可以开始构建 Android，

```
$ 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
...
```

构建工作会花费很长一段时间，它和电脑的硬件配置关系很大，如果笔记本电脑配置四核英特尔带超线程技术的酷睿 i7 处理器和 8GB 的 RAM，这个执行命令将需要大约 20 分钟。如果它的配置是 CENTRO2 双核处理器和 2GB 的内存，将需要大约一个小时来构建相同的 AOSP。请注意，-j 参数允许指定多少个工作可以以并行方式运行。一般是处理器的数量乘以 2。但是也有人建议以处理器数加 2 的数量进行设置。

关于构建有几个其他的事情要考虑。首先注意的是，打印出构建配置和实际构建的第一个输出（"host Java: apicheck (out/host/common/o..."）之间有一个很长的时间间隔什么都不会打印出来。在后面会详细解释这种延迟，简短的说就是在这段时间内构建系统正在找出 AOSP 每个部分的构建原则。

还要注意的是，在构建过程中会看到大量的警告声明。这是相当“正常”，这些告警和软件质量无关，也对构建过程不会产生什么影响。

在虚拟机或非 Ubuntu 系统上构建

经常有人问我如何在虚拟机中进行 AOSP 构建; 最常见的原因是开发团队或者他们的 IT 部门配置的都是 Windows 电脑。虽然用虚拟机也可以构建，但是花费的时间是用 Ubuntu 系统构建时间的两倍。因此建议大家还是用 Ubuntu 来构建。

越来越多的开发商喜欢MacOS X胜过Linux和Windows，包括Google内部有些人也是这样的。因此在http://source.android.com网站上发布的官方指令上也描述了怎样在Mac电脑上进行构建。尽管由于Mac OS的更新，导致这些指导不是特别好用，但是幸运的是在Mac电脑上进行开发的人很多，也很热情，因此常常可以在一些社区或论坛上找到如何在新的Mac OS版本上进行AOSP的构建。

如果选择走虚拟机路线，请在配置 VM 时，确保可以使用多个 CPU。绝大多数 BIOS 的默认配置都是禁止了多 CPU 的虚拟化使用。比如 在 VirtualBox 中如果分配多 CPU 给一个映像时就会弹出错误窗口，必须先到 BIOS 中把相关开关打开后才能正常使用虚拟机。

运行 Android

随着构建完成，就可以启动模拟器来运行自己定制的映像：

```
$ emulator &
```

这将启动如图 3-2 所示的模拟器窗口，模拟器将引导至一个完整的 Android 环境。

Figure 3-2. Android 运行定制映像的模拟器

现在可以像在真实手机上一样的操作模拟器上的 Android，虽然电脑屏幕并不是触屏的，但是可以用鼠标来模拟手指，鼠标点击一下就相当于点击触屏，保持鼠标按下并移动就相当于手指滑动。同时在模拟器右边是一个完整的键盘，就相当于一个带 QWERTY 键盘的

手机，当然也可以通过电脑键盘完成文本输入。

尽管模拟器已经非常接近真实手机，但是它毕竟不是完美的，比如模拟器启动的时间毕竟长，尤其是第一次启动的时候，主要原因是 Dalvik 虚拟机要为在手机上运行的程序创建一个 JIT 缓存。在使用一段时间的模拟器后，你会发现它有点笨重，尤其是在调试修改阶段。同时，模拟器也不能完美的模拟一切，比如在使用 F11 or F12 模拟横竖屏切换时，就不是很好。

不管因为什么原因关闭了当前构建 AOSP 的 shell，在启动一个新的 shell 后必须先运行 envsetup.sh 脚本，以便重新建立环境变量。下面是一个新的 shell 命令：

```
$ cd ~/android/aosp-2.3.x
$ emulator &
No command 'emulator' found, did you mean:
Command 'qemulator' from package 'qemulator' (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，第二次启动模拟器时，shell 并没有再提示命令丢失。像 adb 这些命令也是同样的情况，还要注意，我们并没有执行任何 make 命令，这是因为我们已经构建过 AOSP，因此我们只需要保证重新正确设置了环境变量就可以了。

使用 ADB

Android 开发团队提供的开发环境最有趣的的一点就是可以使用 Usb 和 adb 工具，并通过 Shell 进入模拟器，或者任何真实的设备。

```
$ adb shell  “该命令让模拟器在同一个 shell 中启动”
* daemon not running. starting it now on port 5037 *
```

```
* daemon started successfully *
# cat /proc/cpuinfo  “这是目标机的 shell，cat 实际上是在“目标机”上运行（比如，在模拟器上）”
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
```

正如所看到的，在模拟器上运行的内核报告它发现了 ARM 处理器，ARM 也是事实上的 Android 运行主平台。内核也报告称它运行的平台名称为 Goldfish。这是模拟器的代号名称，在后面很多地方也可以看见。

到目前为止，已经以 root 权限通过 shell 连接到了模拟器上，这样就可以运行任何命令。图 3-3 显示了 ADB 的组件和它的连接方式。

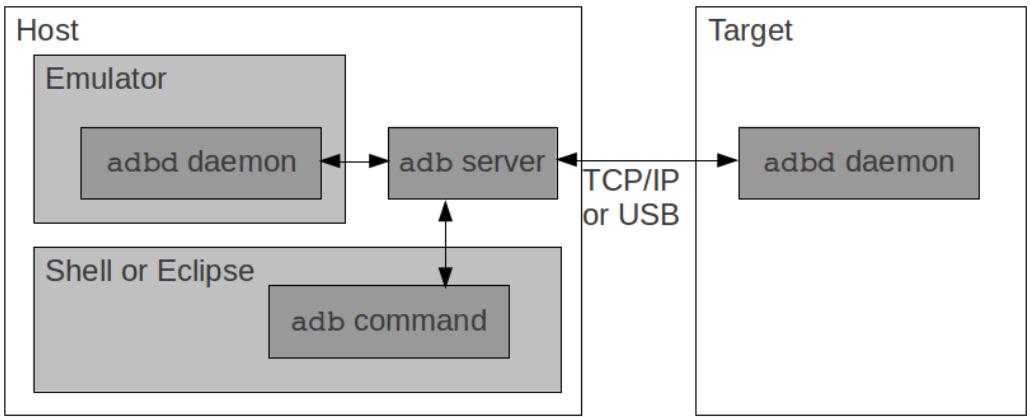

Figure 3-3. ADB 部件互通方式

输入 CTRL-D 可以退出 adb shell：

当第一次在主机上启动 ADB 时，adb 将会在后台启动一个服务器，该服务器的工作就是管理所有连到主机的 Android 设备的连接。

下面的输出可以看到一个守护进程在端口 5037 被启动。实际上可以通过守护进程查看什么设备被连接。

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

下面的输出显示有多个设备被连接，一个时通过 USB 连接的真实手机，另一个是模拟器。在多个设备被连接的时候，可以通过-s 标志可以和具体的设备进行通话，但是要带设

备的串口号：

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

请注意这里的输入提示符是$而不是#。原因是这里并非以 root 权限连接到设备中，这也可以通过 id 命令看出来。因为它连接的是一个真实的商业设备 ,因此无法获得 root 权限，即使通过 su 命令也不行。没有 root 权限的话，对设备的修改就很有限。目前很多手机厂商都不希望让开发人员获得它们手机的 root 权限 ,但是摩托和 HTC 似乎有意让开发人员可以分很容易的获得 root 权限，但毕竟不是主流。

Adb 可以做许多事情，强烈建立输入不带参数的 adb 命令，看看它的输出：

```
$ 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
```

也能通过 adb 命令把在主日志缓存中的数据显示在终端上：

```
$ 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
...
```

这对于观察系统关键部件在运行时的行为非常重要，也包括系统服务器启动的服务。

同时也可以在设备和主机之间进行文件拷贝：

```
$ 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)
```

因为 ADB 几乎是 android 开发的中心，建议大家读读 adb 的相关文档，这后面我们也会经常使用到它。记住 ADB 有一些古怪的表现 ,首先就是在主机端的守护进程有些问题，比如它有时不能正确的显示被连接设备的状态 ,对于已经处于连接状态的设备它会显示为连接。或者设备已经启动完毕并可以接收 adb 命令时还一直处于挂起状态，对于这种情况一般就是把主机端的守护进程给干掉。

不要担心的是，下一次发出任何 adb 命令，守护程序将会自动重新启动。目前还不清楚是什么原因导致这种行为，也许这个问题会在未来得到解决。因此如果在使用 ADB 时看到一些奇怪的行为时，就把主机端的守护进程给干掉。

谷歌 Android 开发文档中的 Android Debug Bridge 部分，建议大家认真看看。

掌握模拟器

正如前文所述，使用在进行平台开发过程中使用模拟器将会非常有用。模拟器有效模拟了带有最小硬件集合的目标机。本小节将会涉及到模拟器的一些高级特性。正如许多 Android 文章介绍的，模拟器本身就是一个非常复杂的软件。但是通过查看模拟器的一些关键功能可以很好的了解它。

前文中，我们通过下面的命令启动了模拟器：

```
$ emulator &
```

但模拟器令也能带几个参数。在命名行中增加 –help 标志可以查看在线帮助：

```
$ 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>/userdata.img
-initdata <file> same as '-init-data <file>'
-data <file> data image (default <datadir>/userdata-qemu.img
-partition-size <size> system/data partition size in MBs
    ...
```

一个特别有用的标志是 *-kernel*。该标志告诉模拟器使用另一个制定内核，而不是默认的预构建好的内核，一般默认内核都在 *prebuilt/android-arm/kernel/*文件夹下：

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

如果想要使用带模块支持特性的内核，就需要构建自己的自定义内核，因为预构建的内核默认是不带模块支持的。此外，默认情况下，模拟器将不会显示内核启动信息。然而，可以通过-show-kernel 标志来看这些信息：

```
$ emulator -show-kernel
Uncompressing Linux..... 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 cpuacct
Initializing cgroup subsys freezer
CPU: Testing write buffer coherency: ok
```

...

同时也可以通过 –verbose 标志让模拟器打印出它自己的执行情况，这样就可以查看模拟器正在使用那些映像文件：

```
$ 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/product/generic
emulator: locking user data image at /home/karim/android/aosp-2.3.x/out/target/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/emulator/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"
...
```

前面本书曾把 QEMU 和模拟器这两个词互换使用，但是实际情况是模拟器命令并不完全等同于 QEMU，模拟器实际上是 Android 开发团队创建的一个 QEMU 的定制封装。然而，大家可以通过 –qemu 标志和模拟器的 QEMU 进行交互。在该标志后面传递的任何东西都被送到 QEMU 而不是模拟器封装。

```
$ 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 -...
```

上一节我们看到如何使用 adb 与运行在模拟器上的 AOSP 进行交互，本节我们又看到如何使用各种选项来改变模拟器的启动方式。有趣的是，我们还可以通过 telnet 进入模拟器来控制模拟器的运行时行为。每个模拟器在启动时都被赋予一个主机上的端口号。回到图 3-2，看看模拟器窗口的左上角，那里的数字（5554）就是模拟器实例监听的端口号。下一个被同时启动的模拟器实例就会得到 5556 的端口号，下下一个就是 5558 端口号，依此类推。为了访问模拟器的控制台，我们可以使用正常的 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
```

通过该控制台，可以使用一些小技巧，比如重定向一个从主机到目标机的端口：

```
redir add tcp:8080:80
```

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

执行该命令后，任何访问主机 8080 端口的程序实际上都是和模拟器上任何正在监听 80 端口的程序进行通话。在 Android 上默认是没有任何东西在监听该端口，但是可以让 BusyBox 的 httpd 运行在 Android 上，并通过该方式与它进行连接。

模拟器也暴露了一些“魔术”IP 给模拟运行的 Android。比如 IP 地址 10.0.2.2 就是工作站 127.0.0.1 的别名。如果 Apache 运行在工作站，就可以打开模拟器的浏览器，输入 http://10.0.2.2,就可以浏览 Apache 服务器发布的任何内容。

更多关于模拟器和各种选项的详细信息可以查看谷歌 Android 开发者文档中的 Android 模拟器部分。虽然该文档的阅读对象是 Android 应用开发者，但是它同样对平台开发者有用。

第四章 构建系统

本章开始将深入探讨构建系统，尽管构建系统是模块化的，但是却非常复杂而且和大多数主流的构建系统不同，一般的构建系统不会用于开源系统。具体来讲，就是易用性比较差没有任何菜单配置。Android 有自己的构建范式，需要花费一些时间来习惯。

和其它构建系统对比

在开始解释 Android 的构建系统是如何工作之前，首先要了解它和其它的构建系统是如何的不同。首先，不像大多数基于 make 的构建系统，Android 构建系统不依赖于递归的 makefile。例如不同于 Linux 内核的是，它没有一个顶层 Makefile 递归调用子目录中的 makefile。相反，有一个脚本将会遍历所有的文件夹和子文件夹直到找到一个 Android.mk 文件。需要注意的是 Android 不依赖于 makefile 文件。相反，正是 Android.mk 文件指定本地的“模块”应该如何构建。

> 注：Android 编译“模块”和内核“模块”没有任何关系。在 Android 的构建系统的范围内，“模块”可以是任何需要被编译的 AOSP 组件，它可能是二进制文件，程序包或者是库等。

另一个 Android 特殊之处是构建系统的配置的方式。虽然绝大多数人都习惯内核风格的 menuconfig 或者是 GNU 的 autoconf/ automake，但是 Android 依赖的是一组变量，这些变量可以在 shell 中通过 envsetup.sh 脚本动态设定，也可以在编译时通过 buildspec.mk 文件进行静态设置。对于新手 Android 构建系统的可配置程度是非常有限的。因此可以通过指定一些属性让 AOSP 包含哪些应用程序，但是没有办法通过一拉 menuconfig 就启用或禁用功能。比如不能决定不支持 wifi 或不想默认启动位置服务。

此外，构建系统不在源文件的同一位置下生成目标文件或任何形式的中间输出。事实上，没有任何已存在的 AOSP 目录用于存放编译输出。相反，构建系统创建一个 out/目录，存储了构建系统产生的一切。因此 make clean 命令和 rm -rf out/命令是一个效果。换句话说，如果把 out/文件夹删除就等同于删掉了编译的所有产出。

开始更详细的研究构建系统之前，最后要介绍的一件事是它严重依赖于 GNU make。而且，更重要的是，需要的版本是 3.81 或更新版本。构建系统其实很大程度上依赖于许多 GNU make 的特定功能，如 define，include 和 ifndef 指令。

架构

如图 4-1 所示，构建系统的入口点是 build/core/文件夹下面的 main.mk 文件，它被顶层的 makefile 文件调用。build/core/文件夹实际上包含了大量的构建系统文件，从现在开始我们会涉及到其中的关键文件。再次提醒，Android 的构建系统把所有的东西都放入了一个单一的 makefile 中，它不是递归的。因此，每一个.mk 文件都最终成为一个巨大的 makefile 中的一部分，该 makefile 包含了系统中所有东西的构建原则。

> **为什么 make 命令会挂起**
>
> 每次键入 make 命令，构建系统会打印出编译配置信息，并挂起一段时间，什么信息都不会显示在屏幕上。经过一段漫长的屏幕沉默时间之后，构建系统会再次开始活动，并编译 AOSP 的每一个部分，这个时候可以看见屏幕上有正常的打印输出。任何人在编译 AOSP 时，都会好奇编译系统在那段时间在干嘛？其实，这段时间它做的事情就是把在 AOSP 中找到的所有.mk 文件进行合并。

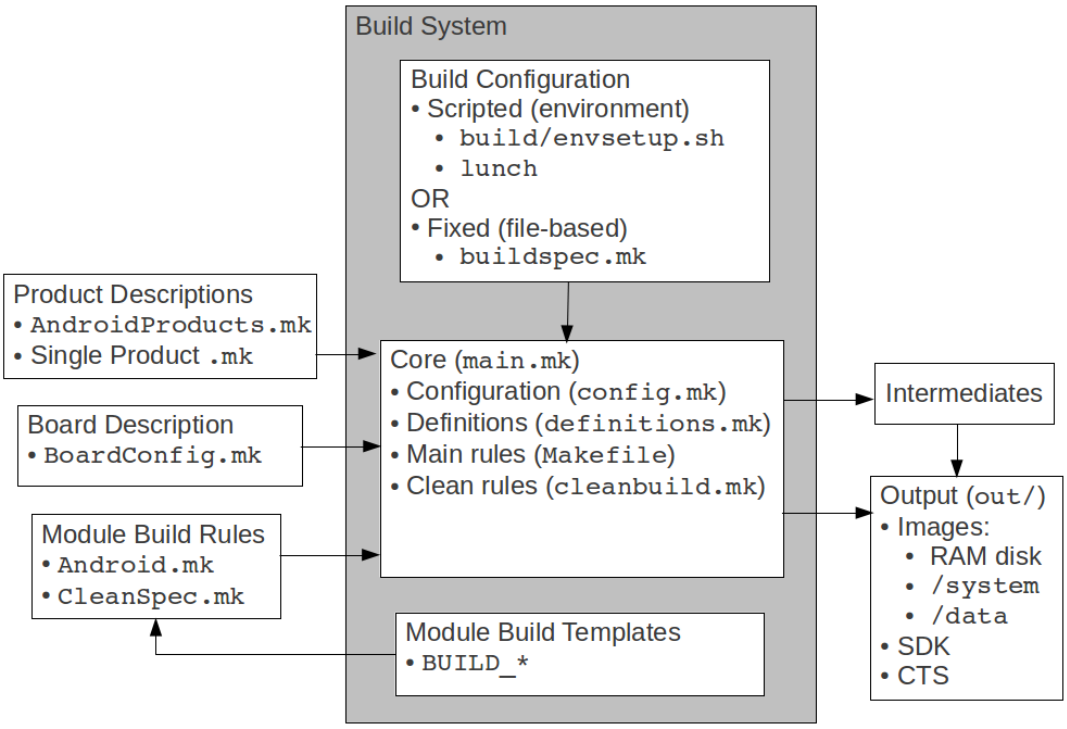

Figure 4–1. Android's 构建系统

如果想要看合并的动作，可以编辑 *build/core/main.mk*，并替换下面的语句：

include $(subdir_makefiles)

替换的语句为：

$(foreach subdir_makefile, $(subdir_makefiles), \

$(info Including $(subdir_makefile)) \

$(eval include $(subdir_makefile)) \

)

subdir_makefile :=

下次在输入 make 命令后，可以看见下面的输出：

$ 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

构建系统做的第一件事就是通过包含 config.mk 文件来获取编译配置信息。编译的配置方式可以有 2 种：要么使用 envsetup.sh 脚本文件，在顶层文件夹中包含 buildspec.mk。这 2 种方式的任一种，都需要设置一些环境变量：

TARGET_PRODUCT

每一个编译版本可以包含不同的应用程序、区域或者编译工程的不同部分。可以看一看在 *build/target/product/*, *device/samsung/crespo/*, 和 *device/htc/passion/*文件夹中的 *AndroidProducts.mk*，它包含有不同的产品.mk文件。这个变量可以被设置成下面的值：

generic – ASOP的最基本编译。

full – 大多数应用程序和主要语言环境被启用。

full_crespo – 和"full"一样，除了Crespo（比如三星的Nexus S.）

sim – Android simulator

sdk – 包括大量的语言环境

TARGET_BUILD_VARIANT

选择需要被安装的模块。每一个模块都应该有一个 LOCAL_MOD ULE_TAGS 变量在 Android.mk 文件中被设置成：* user, debug, eng, tests, optional, or samples 中的一个或多个。通过变量选择，可以告诉构建系统那个模块的子集应该给包括。具体做法是：

eng–包括所有标记为user，debug或eng的模块。

userdebug – 包括同时标记为user和debug的模块。

user –包括只标记为user的模块。

TARGET_BUILD_TYPE

决定是否使用特殊构建标志或是否在代码中定义调试变量。可能的值要么是 release 或者 debug。最值得注意的是，frameworks/base/Android.mk 文件会在 frameworks/base/core/config/debug 和 frameworks/base/core/config/ndebug 两者之间选择一个，选择的依据就是这个变量是否被设置成 debug。前者会把 ConfigBuildFlags.DEBUG 这个 Java 常数设置成 true，后者会把它设置成 false。在系统服务中的部分代码有 DEBUG 条件分支，一般 TARGET_BUILD_TYPE 会被设置成 release。

TARGET_TOOLS_PREFIX

默认情况下，构建系统将使用附带的 prebuilt/目录下面的交叉开发工具链。然而，如果你想使用另一种工具链，你可以设置这个值指向其位置。

OUT_DIR

默认情况下，构建系统会把所有构建成果输出到 out/目录。可以使用这个变量提供一个备用的输出目录。

BUILD_ENV_SEQUENCE_NUMBER

如果使用模板 build/buildspec.mk.default 创建自己的 buildspec.mk 文件，这个值将被正确设置。但是，如果用旧的 AOSP 版本创建一个 buildspec.mk 文件，并且将来会在新的 AOSP 版本上使用，那么这个变量将会成为一道安全网，这将导致构建系统通知，buildspec.mk 文件不匹配现有的构建系统。

除了选择 AOSP 的哪个部分进行构建和使用哪些选项进行构建以外，构建系统也需要知道它将要构建的目标。*BoardConfig.mk* 文件会指定诸如提供给内核的命令行，内核应该被加载到的基地址，或者最匹配主板 *CPU* 的指令集版本（*TARGET_ARCH_VARIANT*）等。看看 *build/target/board/*这个文件夹，它下面有每一个目标的文件夹，并且每一个目标文件夹都有 *BoardConfig.mk*。大家也可以看看在 AOSP 中的各个 *device/*/TAR GET_DEVICE/BoardConfig.mk* 文件。后者比前者丰富得多，因为它们包含了很多硬件的具体信息。设备名（即 *TARGET_DEVICE*）来自于 *product .mk* 文件中的

PRODUCT_DEVICE，它是为配置中的 *TARGET_PRODUCT* 组提供的。例如，*device/samsung/crespo/AndroidProducts.mk* 包括 *device/samsung/crespo/full_crespo.mk*，它会设置 PRODUCT_DEVICE 为 crespo。因此构建系统会在 *device/*/crespo/*中寻找 *BoardConfig.mk*。而且也能在这个位置找到该文件。

关于配置内容的最后一部分就是用于 *Android* 构建的 CPU 选项。对于 ARM，这些选项都包含在 *build/core/combo/arch/arm/armv*.mk* 中，*TARGET_ARCH_VARIANT* 变量会决定哪个文件会被使用。每个文件会列出用于构建 C / C ++文件的与具体 CPU 相关的交叉编译器和交叉连接器标志。它们还包含了一些 *ARCH_ARM_HAVE_** 变量，这些变量会在目标机 CPU 中有特定的 ARM 功能被发现时影响 AOSP 部份代码的编译。

Andriod Simulator

如果回到“构建 Android“那一节，你会看到由 lunch 打出的菜单中有一个 simulator 的选项。事实上有很多地方都可以找到 simulator，包括 *Android.mk* 文件中和项目树上的一些文件夹中。关于 simulator 最重要的一件事就是，它和 emulator 完全没有关系，它们完全是 2 个东西。

这么说吧，simulator 似乎是 Android 开发团队在创建 Android 过程中早期工作的残余。由于当时他们没有在 QEMU 中运行的 Android，他们使用自己的桌面操作系统和 LD_PRELOAD 机制来模拟 Android 设备，因此称为“模拟器”。一旦在 QEMU 上运行 Android 成为可能，simulator 就被停止使用了。但是，simulator 仍然存在，因为为了在工作站上进行开发和测试工作，它仍然对 AOSP 的构建有用。

但这并不意味着可以在桌面电脑上运行 AOSP。因为运行 AOSP 意味着电脑的内核应该有 Binder, 并且要把系统中的 C 库替换成 Bionic。不过如果在桌面上只运行 AOSP 其中的一部分是可以的，因为产品的设计考虑到这一点。

如果构建目标是 simulator 的话，那么各个部分代码的构建会非常不同。例如，如果浏览代码会发现一些条件编译分支是围绕着 HAVE_ANDROID_OS 这个宏。如果没有定义 HAVE_ANDROID_OS ,该代码将返回一个错误给其调用者，而不是试图与 Binder 驱动进行通话。

如果要全面了解 simulator 的话，可以看看 Andrew McFadden 的文章“Android Simulator Environment ”。

envsetup.sh

现在大家已经理解了构建系统需要的配置输入，因此我们可以更加深入的探讨一下 envsetup.sh 这个脚本文件。正如它名字说暗示的，envsetup.sh 实际上是为 Android 设立一个构建环境。它的唯一工作就是为任何形式的 AOSP 工作定义一系列的 shell 命令：

```
$ 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.
```

通过查看源文件可以看见更多的函数。完整的列表如下：

add_lunch_combo cgrep check_product check_variant choosecombo chooseproduct choosetype choosevariant cproj croot findmakefile gdbclient get_abs_build_var getbugreports get_build_var getprebuilt gettop godir help isviewserverstarted jgrep lunch m mm mmm pgrep pid printconfig print_lunch_menu resgrep runhat runtest set_java_home setpaths set_sequence_number set_stuff_for_environment settitle smoketest startviewserver stopviewserver systemstack tapas tracedmdump

读者可能会发现 CRoot 以及 godir 命令对于遍历文件夹树非常有用。因为文件夹的有些目录层次非常深，有些文件夹距离 AOSP 顶层目录有 10 层，因此为了返回到文件夹的上几层会不停的输入 cd ../../../ ... 这样的命令是非常令人乏味的。

m 和 mm 们也非常有用，因为它们可以从顶层开始进行构建而不管用户当前在那一层文件夹中，同时它们也允许用户只构建当前文件夹中的模块。比如，开发人员对 Launcher 做了修改，并且当前的路径在 packages/apps/Launcher2 下面，那么就可以通过 mm 命令重新构建该模块，而勿需通过 cd 命名返回顶层文件夹再通过 make 命令进行构建。请注意 mm 命令不会重建整个树，因此不会重新生成 AOSP 映像，即便一个依赖模块已经被修改。但是 m 命令可以这么做。Mm 命令对于测试本地修改是否会中断构建是非常有用的。同时它对于验证是否可以重新生成整个 AOSP 也是有用的。

虽然联机帮助没有提及 *lunch*,但它确实是 envsetup.sh 定义的命令之一。如果不带参数的运行 *lunch*，会打印出潜在的选择列表：

```
$ 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]
```

这些选择不是一成不变的。大部分取决于运行 AOSP 时的 envsetup.sh 的内容。这些选项是脚本文件中的函数 add_lunch_combo()单独添加的。举例来说，默认情况下，envsetup.sh 增加 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
```

envsetup.sh 还包括所有供应商提供的脚本：

```
# 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
```

device/samsung/crespo/vendorsetup.sh 文件会完成下面的工作：

```
add_lunch_combo full_crespo-userdebug
```

所以这就是我们前面看到的菜单，需要注意的是菜单会让你选择一个组合。实质上，这是一个 TARGET_PRODUCT 和 TARGET_BUILD_VARIANT 的组合，菜单提供了默认的组合，但其他组合仍然保持有效，并且可以通过命令行把参数传给 *lunch*：

```
$ 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 完成 generic-eng 组合的运行，在表 4-1 中的环境变量就会在当前 shell 中被建立，以便向构建系统提供需要的配置信息。

当然，如果厌倦了总是输入 build/envsetup.sh 及 lunch 命令，可以把 build/buildspec.mk.default 拷贝到顶层文件夹，然后重新命名它为 buildspec.mk，并编辑这个文件以匹配运行这些命令所设置的配置项。该文件已经包含了所有你需要提供的变量；完成了这个工作后，可以直接到 AOSP 的文件夹，并直接调用 make 命令。这样就跳过了 envsetup.sh 和 lunch。

Table 4-1. Environment variables set by lunch (in no particular order)

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

使用 ccache

如果在读完这是资料后，大家已经尝试过构建 AOSP，相信大家已经意识到这个过程有多长。如果能够提升硬件配置，这个过程可以缩短。很明显 Android 开发团队也痛苦地认识到了这个过程很长，所以他们增加了多 ccache 的支持。Ccache 的意思就是编译器缓存(Compiler Cache)，它是 Samba 项目的一部分。它的机制是基于预编译的输出来缓存编译器生成的目标对象。因此对于两个独立的构建，如果预处理器的输出是相同的，ccache 将导致第 2 个构建实际上并没有使用编译器以生成文件。。相反，被缓存的对象文件将被复制到编译器输出存放的目标文件夹中。

要启用 ccache 功能，需要确保 USE_CCACHE 环境变量设置为 1，然后再开始构建：

$ export USE_CCACHE=1

第一次运行不会获得任何加速，因为高速缓存在那个时间是空的。但是之后，缓存将有助于加速构建过程。唯一的缺点是 ccache 仅适用于 C / C ++文件。因此，不能加速任何 Java 文件构建，不幸的是，AOSP 有大约 15,000 个 C / C ++文件和 18000 Java 文件。因此，尽管缓存也不是万能的，但它却是非常有意义。

如果想了解更多关于 ccache，看看由马丁·布朗撰写发表在 IBM developerWorks 网站上一篇题为“使用 ccache 提高协同构建时间“的文章。文章还探讨了 distcc 的使用，它允许多台机器上进行分布式构建，并允许把团队工作站的缓存汇总在一起。

指令定义

由于构建系统是相当大的，在 build/core/文件夹中就有大约 40 个文件，能够重复使用尽可能多的代码是非常重要的。这就是为什么构建系统在 definitions.mk 文件中定义了大量指令（makefile 中的指令类似于编程语言中的函数）。这个文件实际上是构建系统中最大的一个文件，约有 60KB 大小，1800 行 makefile 代码和 140 条指令。指令提供了多种功能，包括文件查找（例如，所有的 makefile 和所有 C 文件），转换（例如，变换 C 文件为 o 文件， java 文件转为 class.jar），拷贝和工具。

这些指令不仅会被整个构建系统的组件使用和扮演核心库的角色，而且有时它们也直接用于模块的 Android.mk 文件。下面是一个代码片段示例，来源于计算器程序的 Android.mk 文件。

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

虽然彻底描述 definitions.mk 是这本书的范围之外，自己研究这个文件也是很容易的。如果不出意外，大部分指令的前面有注释说明他们的作用，下面是一个例子：

```
###########################################################
## Find all of the java files under the named directories.
## Meant to be used like:
## SRC_FILES := $(call all-java-files-under,src tests)
###########################################################
define all-java-files-under
$(patsubst ./%,%, \
$(shell cd $(LOCAL_PATH) ; \
find $(1) -name "*.java" -and -not -name ".*") \
)
endef
```

主要的 make 食谱

到这儿大家可能好奇像 RAM 磁盘映像是怎么被生成出来的？SDK 是如何集成的？不要着急，好东西总是在最后，现在我们一起来看看 build/core/下面的 makefile（不是顶层的 makefile）。这个文件的开头是一句不起眼的注释：

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

但是，不要被愚弄，这是其中一块最好的肉。下面的代码片段负责生成 RAM 磁盘：

```
# ------------------------------------------------------------------
# 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) > $@
```

下面的片段创建证书包用以检查 OTA 更新（无线更新）：

```
# ------------------------------------------------------------------
# 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
```

查看 makefile 文件了解下面内容是如何被创建：

- Properties (including the target's */default.prop* and */system/build.prop*)
- RAM disk
- Boot image (combining the RAM disk and a kernel image)
- *NOTICE* files
- OTA keystore
- Recovery image
- System image (the target's */system* directory)
- Data partition image (the target's */data* directory)
- OTA update package
- SDK

然而，这个文件中不包含如下内容：

Kernel images

不要查询任何构建内核映像的规则。AOSP 没有内核部分。相反开发人员为自己的目标机找到 Android 可以使用的内核，并且在 AOSP 之外单独构建它，然后合并入 AOSP 中。在 device/文件夹下可以找到一些例子，比如 device/samsung/crespo/，它有一个内核映像（名叫 kernel 的文件），以及一个拥有 Crespo's Wifi 的可加载模块(bcm4329.ko 文件)。这两个文件都是在 AOSP 之外构建的，并以二进制文件的形式被并入项目树中。

NDK

虽然构建 NDK 的代码是在 AOSP 中，但是 *NDK* 是完全独立于 build/中的 AOSP 构建系统。NDK 的构建系统是在 *ndk/build/*。我们将简短地讨论如何构建 *NDK*。

CTS

构建 *CTS* 的规则是在 *build/core/tasks/cts.mk* 文件中。

清理-Cleaning

正如前面提到的，*make clean* 就几乎等同于清除 *out*/文件夹。*Clean* 目标是在 *main.mk* 文件中定义的。但是，有其他的清理目标。最值得注意的是，*installclean*，只要修改 *TARGET_PRODUCT* 或者 *TARGET_BUILD_VARIANT* 时，就会被自动调用。举例来说，如果先使用 *generic-eng* 组合构建 AOSP，然后使用 *lunch* 切换组合到 *full-eng*，那么下次启动 *make* 并使用 *installclean* 时，一些构建输出将自动被删除。

```
$ 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-qemu/*
out/target/product/generic/userdata-qemu.img out/host/linux-x86/obj/NOTICE_FILES
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/product/generic/obj/NOTICE_FILES out/target/product/generic/obj/PACKAGING
out/target/product/generic/recovery out/target/product/generic/root
out/target/product/generic/system out/target/product/generic/dex_bootjars
out/target/product/generic/obj/JAVA_LIBRARIES
*** Done with the cleaning, now starting the real build.
```

对比 *clean*，*installclean* 不会清除整个 *out/*。相反，它只是清除在组合配置被改变时需要重新构建的那部分。另外，*clobber* 和 *clean* 基本一样。

模块构建模板

刚才所描述的是构建系统的架构，以及核心组件的方法。读过上面的文章之后，从自上而下的角度来理解 *Android* 是如何被构建的会有更好的认识。然而，却很少能对 AOSP 模块的 Android.mk 文件有深入的理解。事实上系统的架构设计让模块构建方法是于独立构建系统的内部。相反，通过提供构建模板让模块的作者能正确的构建自己的模块。每个模板会根据模块的具体类型进行裁剪，并且模块作者能使用一组都带有 *LOCAL_* 前缀的变量来调节模板的行为和输出。当然，模板及相关支持文件（主要是 *base_rules.mk*）密切与构建系统的其余部分进行互动，以便妥善处理每个模块的构建输出。但是这一切对于模块作者都是不可见的。模板放在 *build/core/*中，*Android.mk* 通过 *include* 指令可以访问这些模板，下面是一个例子：

```
include $(BUILD_PACKAGE)
```

正如你可以看到，*Android.mk* 实际上没有通过名字来包括 *.mk* 模板。相反，它包括一个变量，这个变量被设置为相应的 *.mk* 文件。表 *4-2* 列出了可用的模块模板的完整列表：

Table 4-2. 模块构建模板列表

变量	模板	模板构建内容	最常见的使用
BUILD_EXECUTABLE	*executable.mk*	Target binaries	Native commands and daemons
BUILD_HOST_EXECUTABLE	*host_executable.mk*	Host binaries	Development tools
BUILD_RAW_EXECUTABLE	*raw_executable.mk*	Target binaries that run on bare metal	Code in the *bootloader/* directory
BUILD_JAVA_LIBRARY	*java_library.mk*	Target Java libaries	Apache Harmony and Android framework
BUILD_STATIC_JAVA_LIBRARY	*static_java_library.mk*	Target static Java libraries	N/A, few modules use this
BUILD_HOST_JAVA_LIBRARY	*host_java_library.mk*	Host Java libraries	Development tools
BUILD_SHARED_LIBRARY	*shared_library.mk*	Target shared libraries	A vast number of modules, including many in *external/* and *frameworks/base/*
BUILD_STATIC_LIBRARY	*static_library.mk*	Target static libraries	A vast number of modules, including many in *external/*
BUILD_HOST_SHARED_LIBRARY	*host_shared_library.mk*	Host shared libraries	Development tools
BUILD_HOST_STATIC_LIBRARY	*host_static_library.mk*	Host static libraries	Development tools
BUILD_RAW_STATIC_LIBRARY	*raw_static_library.mk*	Target static libraries that run on bare metal	Code in *bootloader/*
BUILD_PREBUILT	*prebuilt.mk*	For copying prebuilt target	Configuration files and

		files	binaries
BUILD_HOST_PREBUILT	*host_prebuilt.mk*	For copying prebuilt host files	Tools in *prebuilt/* and configuration files
BUILD_MULTI_PREBUILT	*multi_prebuilt.mk*	For copying prebuilt modules of multiple but known type, like Java libraries or executables	Rarely used
BUILD_PACKAGE	*package.mk*	Built-in AOSP apps (i.e. anything that ends up being an *.apk*	All stock AOSP apps
BUILD_KEY_CHAR_MAP	*key_char_map.mk*	Device character maps	All device character maps in AOSP

这些构建模板允许 Android.mk 文件以轻量级的方式被使用。

```
LOCAL_PATH := $(call my-dir) //告诉构建模板当前模块的位置
include $(CLEAR_VARS)// 清除所有以前为其他模块设置的LOCAL_*变量
LOCAL_VARIABLE_1 := value_1//设置各种LOCAL_*变量为模块特定的值。
LOCAL_VARIABLE_2 := value_2
...
include $(BUILD_MODULE_TYPE)//根据模块类型调用相应的构建模板
```

需要注意的是*CLEAR_VARS*，这是由*clear_vars.mk*提供的，是非常重要的。回想一下，构建系统把所有的Android.mk并入到一个单一的巨大的makefile中。另外，一个单一的Android.mk可以依次描述多个模块，因此，CLEAR_VARS确保以前模块的食谱不污染后续的模块。

下面是服务管理器的Android.mk实例(frameworks/base/cmds/servicemanager/):

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := service_manager.c binder.c
LOCAL_MODULE := servicemanager
ifeq ($(BOARD_USE_LVMX),true)
LOCAL_CFLAGS += -DLVMX
endif
include $(BUILD_EXECUTABLE)
```

一个桌面时钟应用程序的示例*(packages/app/DeskClock/)*:

```
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))
```

可以看出两个模块基本上使用了相同的结构，即使它们提供了非常不同的输入，并且产生了非常不同的输出。还要注意座钟程序的*Android.mk*的最后一行，基本上包括所有子目录中的*Android.mk*文件。正

如刚才所说，构建系统查找层次结构中的第一个*makefile*，并且在找到后不会再查找下一层的文件夹，因此需要手动调用这些。显然，这里的代码只是查询所有的下层*makefile*，然而，*AOSP*要么明确列出子目录或有条件的基于配置选择它们。

在*http://source.android.com*的文档提供一个详尽的*LOCAL_*"的变量清单，指明了它们的含义和用法。不幸的是，在写这篇文章时，这个名单已不再可用。但是，*build/core/build-system.html*文件包含了该名单的早期版本，可以先参考一下。下面是一些最经常遇到的*LOCAL_*变量：

LOCAL_PATH

当前模块源文件的路径，通过调用*$(call my-dir)*来使用。

LOCAL_MODULE

此模块的构建输出属性的名称。实际的文件名或输出以及它的位置将取决于包括的构建模板。例如，如果该变量被设置为*foo*，并构建一个可执行文件，最终的可执行文件将是一个名为*foo*的命令，它将被放置在目标机的*/system/bin/*下面。如果*LOCAL_MODULE*被置为*libfoo*并且包括*BUILD_SHARED_LIBRARY*来代替*BUILD_EXECUTABLE*，那么构建系统将产生*libfoo.so*并把他放在*/system/lib/*下面。

LOCAL_SRC_FILES

用于构建模块的源文件。可以通过使用一个构建系统的指令来提供这些源文件，比如桌面时钟程序就使用*all-java-files-under*，或者可以像服务管理器那样明确列出文件。

LOCAL_PACKAGE_NAME

与所有其他模块不同的是，应用程序使用这个变量来代替*LOCAL_MODULE*去指定他们的名字，通过比较两个*Android.mk*可以看出来。

LOCAL_SHARED_LIBRARIES

使用此变量列出所有模块依赖的库。正如前面提到的，服务管理器依赖liblog而不是这个变量。

LOCAL_MODULE_TAGS

正如前面提到的，这个变量让模块在TARGET_BUILD_VARIANT控制下进行构建。

LOCAL_MODULE_PATH

使用这个变量去覆盖需要构建的模块的默认安装位置。

了解LOCAL_*变量的一个好办法是查看现有的AOSP中的Android.mk文件。此外，clear_vars.mk包含被清除的变量的完整列表。尽管它没有解释每个变量的含义，但是确实列出了所有变量。

此外，在除了全面影响AOSP的cleaning targets之外，每个模块都可以通过提供一个CleanSpec.mk定义它自己的cleaning规则，很像模块提供的Android.mk文件。但是，与后者不同的是，前者不是必须的。默认情况下，构建系统有每个类型模块的cleaning规则。但是，如果模块构建出了一些构建系统默认情况下不产生的东西，那么构建系统是不知道如何clean它们的，因此在CleanSpec.mk中可以指定自己的规则

输出

现在我们已经看了构建系统的工作原理以及模块如何使用构建模板，现在来看看构建系统保存在 *out/*下的输出内容。从总体上来看，构建输出分成 3 个阶段，2 种模式，一种模式适用于主机，另一种适用于目标机：

由模块源码产生的中间文件。这些中间文件的形式和位置取决于模块的源代码。如果是 *C/C++*文件则生成*.o* 文件，如果是 *Java* 代码则生成*.jar* 文件。

1. 构建系统会使用这些中间文件生成真正的二进制文件和包：比如把*.o*文件链接成一个真正的二进制执行文件。

二进制文件和包被组装成构建系统需要的最终成果物。例如，二进制文件被复制到包含*/root* 和*/system* 文件系统的文件夹中，并且这些文件系统的映像会被用到真实的设备上。

*out/*下有两个主要目录，反映了其经营模式：*host/*和 *target/*。在每一个目录下都有 *obj/*文件夹，它

包含了构建过程中产生的各种中间文件。这些中间文件都被保存在其下的各个子文件夹中。这些文件夹的名字都类似于 *BUILD_* 这样的宏名：

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

很可能最感兴趣的文件夹是 *out/target/product/PRODUCT_DEVICE/*。这个文件夹存放了产品映像文件，该映像文件是由对应产品的 *configuration's .mk* 文件中定义的 *PRODUCT_DEVICE* 指定的。*Table 4-3* 解释了该文件夹的内容。

Table 4-3. Product output

条目	描述
android-info.txt	*包含了为产品配置的样板代号*
clean_steps.mk	*包含了清理树所必须的一系列步骤，通过调用add-clean-step指令*
data/	*目标机的data/文件夹*
installed-files.txt	*安装在 data/和 system/文件夹中的所有文件清单*
obj/	*目标产品的中间。*
previous_build_config.mk	*最后构建目标；会在下一次的make中用于检查是否配置已经被改变，如果是，强制installclean。*
ramdisk.img	*基于root/文件夹内容生成的*RAM磁盘映像。
root/	*目标机root文件系统的内容*
symbols/	*放在root文件系统和/system目录下的二进制文件*
system/	*目标机的system /文件夹*
system.img	*/system映像，基于system/文件夹的内容*
userdata.img	*/data映像，基于data/文件夹的内容*

可以回头复习一下第二章的*root*文件系统、*/system*和*/data*。从本质上讲，在内核引导时，会安装*RAM*磁盘映像并执行初始化程序。反过来，映像文件会运行*/init.rc*脚本，用于加载*/system*和*/data*到它们对应的位置。

构建食谱-Build Recipes

处于理解构建架构和功能的目的，我们来看看最常见的，还有一些稍微少见的构建食谱。我们只稍微接触一下使用每个配方的结果，因为该主题是最好的别处讨论，但是应该有足够的信息开始学习。

默认的 droid 构建

此前，我们浏览了一些纯*make*命令，但从来没有真正解释默认目标。当运行纯*make*时，就好像输入了：

$ *make droid*

droid事实上是定义在*main.mk*中的默认目标。通常并不需要手动指定这个目标。

构建命令

当构建AOSP时，并没有实际显示构建使用的命令。相反，只能打印出在每一步的总结。如果想看构建做的每一件事，需要添加*showcommands*到命令行中：

```
$ 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
...
```

下面的命令也是一样的：

```
$ make droid showcommands
```

很容易发现该命令会产生很多输出，因此很难看清楚。如果要分析用来构建AOSP的命令，可以把标准输出和标准错误保存到文件中。

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

为 linux 和 Mac OS 构建 SDK

官方 Android SDK 可以从 http://developer.android.com 中找到。但是，可以使用 AOSP 来构建自己的 SDK。例如，扩展核心 API 提供新的功能，并且发布给开发人员，就需要选择一个特殊的组合：

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

构建一完成， linux版本的SDK会被存在out/host/linux-x86/sdk/文件夹下面， Mac OS版本的会被放在out/host/darwin-x86/sdk/下面。将有两个副本，一个是zip文件，很像 http://developer.android.com中的发布版本，另一个是未压缩的，可以直接使用并准备使用。

如果已经配置好Eclipse，需要进行两次额外的步骤来使用新建的SDK。首先，需要告诉Eclipse新的SDK的位置。此外，也需要去"Window→"Android SDK and AVD Manager"→"Installed Packages" 并且点击 "Update All..." 这将显示一个向导。只选择第一个项目"Android SDK Tools, revision api_level",并且点击 "Install."。完成后就可以使用新的SDK创建工程项目，并访问任何新的API。如果不做第二个步骤，能够创建新的Android项目，但是不能进行编译。

为 Windows 构建 SDK

为Windows构建SDK稍微有一些不同：

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

结果输出保存在out/host/windows/sdk/。

构建 CTS

如果想构建CTS，需要用到envsetup.sh或lunch：

```
$ 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命令包含了自己的联机帮助：

```
$ 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
Avaiable 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
```

一旦有目标机启动并开始运行，比如像模拟器，就可以启动测试工具包，它提供adb工具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

正如前面提到的，NDK有自己单独的编译系统，拥有自己的设置和帮助系统，你可以像这样调用：

```
$ 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

当准备构建*NDK*时，可以像下面一样调用*make-release*，并能看见一些告警：

```
$ ./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...
Using git clone prefix: git://android.git.kernel.org/toolchain
downloading sources for toolchain/binutils
...
```

更新 API

构建系统针对修改 AOSP 的核心 API 的情况有安全机制。如果真的动了核心 API，默认情况下，构建失败，并产生如下告警：

```
******************************
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....
```

如错误信息提示，为构建能继续，需要做这样的事情：

```
$ 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_cgi.so

Install: out/host/linux-x86/lib/libclearsilver-jni.so

Copying:

out/target/common/obj/JAVA_LIBRARIES/core_intermediates/emma_out/lib/classes-jar.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_out/lib/classes-jar.jar

Copying: out/target/common/obj/JAVA_LIBRARIES/ext_intermediates/emma_out/lib/classes-jar.jar

Install: out/host/linux-x86/bin/aidl

Copying:

out/target/common/obj/JAVA_LIBRARIES/core-junit_intermediates/emma_out/lib/classes-jar.jar

Copying:

out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/emma_out/lib/classes-jar.jar

Copying current.xml

下次启动*make*时，不会再得到任何有关*API*的变化的错误提示。

构建单个模块

到现在为止，已经浏览了构建整个项目树的情况。但是我们还可以构建单个模块。下面的示例说明如何让构建系统构建*Launcher2*模块（即：主屏幕）：

$ make Launcher2

当然也可以单独*clean*模块：

$ make clean-Launcher2

如果想迫使编译系统重新生成系统映像，以便包括新后的模块，可以添加*snod*目标到命令行：

$ 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_intermediates/package.apk)

'out/target/common/obj/APPS/Launcher2_intermediates//classes.dex' as 'classes.dex'...

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

构建树外的内容

如果想构建 AOSP 和仿生库之外的代码，但是又不想把代码并入 AOSP 中，就可以使用下面的命令：

```
# 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
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
```

这种情况下就不用管*envsetup.sh*或*lunch*。只需要键入：

```
$ make
```

显然，这不会增加二进制文件到*AOSP*产生的任何映像中。

AOSP 黑客基础

你最有可能因为一件事买了这本书：破解*AOSP*，以适应您的的需求。在接下来的几页中，将开始介绍一些最明显的黑客技术，你可能会想尝试。当然，我们只是浅显的介绍，深入的内容会在下章介绍。

增加 APP

如果在现有程序栈中想添加一个默认的应用程序，首先需要在*packages/apps/*中为该程序创建一个文件夹。举一个简单的例子，我们在*Eclipse*中创建“*HelloWorld*!”的程序，然后复制该应用程序从*Eclipse*工作区到它的目的地：

```
$ cp -a ~/workspace/HelloWorld ~/android/aosp-2.3.x/packages/apps/
```

然后，在*aosp-rootpackages/apps/HelloWorld*中创建一个*Android.mk*,以便构建这个*app*。

```
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)
```

鉴于我们标记这个模块作为可选，默认情况下它不会被*AOSP*构建。为了包括它，需要把它加入到*aosp-root/build/target/product/core.mk*中的*PRODUCT_PACKAGES*中。

目前介绍的是把*app*加入到所有的产品中，但是读者可能只想把*app*加入到自己的产品中，那么应该增加*app*到*device*/下的产品条目中，而是*packages*/*apps*/中。

增加原生工具和守护进程

原生工具和守护进程在 AOSP 中被存放在多个位置：

system/core/ 和 *system/*

定制的 *Android* 二进制文件，在 *Android* 框架以外可以使用。

frameworks/base/cmds/

与 *Android* 框架紧耦合的二进制文件。比例服务管理器。

external/

外部项目所产生的二进制文件，比如 *strace*。

现在应该知道生成二进制文件的代码应该放在什么地方呢，同时要提供一个 *Android.mk*。

```
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)
```

同样，还需要确保 *hello-world* 被列入 *aosp-root/build/target/product/core.mk* 中的 *PRODUCT_PACKAGES* 默认清单中。如果让 app 成为产品定制的程序，应该把二进制文件放入 device/ 下面和产品相关的文件夹中。

增加原生库

与二进制文件类似，库在 AOSP 中也被存放在多个位置。和二进制文件不一样的是许多库在一个单一的模块中被使用。因此，这些库通常与该模块的代码放在一起。而系统级的库文件一般是放在下面文件夹中：

system/core

系统级核心库，一般包含了 *Android* 框架之外的一些库，比如 *liblog* 就是。

frameworks/base/libs/

与框架联系紧密的库，比如 libbinder。

external/

由外部项目生成，并被引入到 *AOSP* 中的库，比如：OpenSSL's libssl。

无论库物文件是在以上的文件夹，或是模块文件夹还是在产品的 *device/*中，都需要 *Android.mk* 来编译它们。

```
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)
```

要使用编译好的库，必须把它添加到 Android.mk 文件列出的二进制文件依赖的库清单中。

```
LOCAL_SHARED_LIBRARIES := libmylib
```

您还可能需要添加相应的头文件到库文件所在的文件夹中的 include/下面，以便需要连接库文件的源码可以找到相应的头文件，比如 system/core/include/和 frameworks/base/include/。

库的预连接

为了减少加载库所花费的时间，Android 会预连接它的大多数的库。方法就是提前指定库将被加载到的地址位置，而不是在运行时才指定。指定地址位置的文件是 build/core/prelink-linux-arm.map，完成映射的工具是 apriori。它包含了下面的一些条目：

```
# core system libraries
libdl.so                0xAFF00000 # [<64K]
libc.so                 0xAFD00000 # [~2M]
libstdc++.so            0xAFC00000 # [<64K]
libm.so                 0xAFB00000 # [~1M]
liblog.so               0xAFA00000 # [<64K]
libcutils.so            0xAF900000 # [~1M]
libthread_db.so         0xAF800000 # [<64K]
libz.so                 0xAF700000 # [~1M]
libevent.so             0xAF600000 # [???]
libssl.so               0xAF400000 # [~2M]
...
# assorted system libraries
libsqlite.so                0xA8B00000 # [~2M]
libexpat.so                 0xA8A00000 # [~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]
...
```

如果想添加一个自定义的原生库，需要将它添加到prelink-linux-arm.map文件中的库列表中或者设置LOCAL_PRELINK_MODULE为false。如果不这么做构建会失败。

增加设备

添加一个自定义的设备是阅读这本书最有可能的目的之一。本节只介绍完成该工作的构建部分，要把Android移植到一个新的设备上还有很多步骤，它们会在本书的其它章节中介绍。尽管如此，添加新的设备到构建系统中仍然是首先要完成的工作之一。

为了方便进行练习，现在假设你正为一家名为ACME的公司工作，你的任务是交付它的最新发明：：CoyotePad，发明它的目的是打造一款玩所有鸟游戏的最佳平台。首先我们在device/下面创建我们新设备的条目：

```
$ cd ~/android/aosp-2.3.x
$ . build/envsetup.sh
$ mkdir -p device/acme/coyotepad
$ cd device/acme/coyotepad
```

然后我们需要一个AndroidProducts.mk用以描述为CoyotePad构建的各种AOSP产品：

```
PRODUCT_MAKEFILES := \
        $(LOCAL_DIR)/full_coyotepad.mk
```

full_coyotepad.mk文件内容如下：

```
$(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 +=
PRODUCT_NAME := full_coyotepad
PRODUCT_DEVICE := coyotepad
PRODUCT_MODEL := Full Android on CoyotePad, meep-meep
```

应该认真看看这个makefile文件。首先，使用了inherit-product指令来告诉构建系统纳入其它产品的描述作为我们产品的基础。这让我们可以在其他人成果的基础上进行构建，而不是开始每一个位和一块AOSP的指定。languages_full.mk这边文件纳入了大量的语言环境，并且full.mk确保我们得到相同的一组模块，就如同我们已使用了full-eng组合完成了构建。

关于其他的变量：

DEVICE_PACKAGE_OVERLAYS

允许指定一个目录，形成一个将覆盖AOSP源文件的基础，使我们能够用设备特定的资源替代默认包资源。例如，如果想为Launcher2或其它应用程序设置自定义的布局或颜色，这个有用会非常有用。在下一节来看看如何使用这个变量。

PRODUCT_PACKAGES

允许指定除了那些已经继承的软件包之外，还需要包括的指定的软件包。如果有自定义的应用程序，二进制文件或位于device/acme/coyotepad/下的库文件，想添加它们到最终生成的映像文件中时，要用到这个变量。

PRODUCT_COPY_FILES

允许列出复制到目标文件系统的特定的文件和需要被复制到的位置。每个源/目的对都是冒号分隔，并且对于对之间有空格。它对配置文件和预编译的二进制文件，比如固件映像和内核模块，有用。

PRODUCT_NAME

就是TARGET_PRODUCT，有2种方式进行设置：选择lunch组合或者传递这个变量给lunch：

```
$ lunch full_coyotepad-eng
```

PRODUCT_DEVICE

最终交付给客户的产品名称。TARGET_DEVICE来自这个变量。PRODUCT_DEVICE必须与device/acme/中的条目匹配，因为构建系统会查找相应的BoardConfig.mk文件。

PRODUCT_MODEL

手机设置里面“关于手机”部分显示的型号。这个变量实际上被存储在设备上的全局属性ro.product.model中。

现在，我们已经描述了产品，还必须通过BoardConfig.mk文件提供有关设备主板的一些信息：

```
TARGET_NO_KERNEL := true
TARGET_NO_BOOTLOADER := true
TARGET_CPU_ABI := armeabi
BOARD_USES_GENERIC_AUDIO := true
USE_CAMERA_STUB := true
```

这个文件非常小，只是为了保证构建成功。现实中的文件可以参看*device/samsung/crespo/Board-ConfigCommon.mk*。

还需要提供常规的Android.mk为了构建包含在设备的目录中的所有模块：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
ifneq ($(filter coyotepad,$(TARGET_DEVICE)),)
include $(call all-makefiles-under,$(LOCAL_PATH))
endif
```

事实上首选的做法是把所有设备特定的应用程序，二进制文件，和库放入到设备文件夹。如果在这里添加模块，别忘了也把它们添加到PRODUCT_PACKAGES。如果你只是把它们放在这里，并提供有效的Android.mk文件，可以构建成功，但将不会在最终映像中。

最后，需要添加一个vendorsetup.sh在设备的目录中，并确保，它是可执行文件：

```
add_lunch_combo full_coyotepad-eng
$ chmod 755 vendorsetup.sh
```

然后执行下面的操作：

```
$ 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
```

可以看到，现在的AOSP认识新设备，并打印信息。当编译完成后，输出类型与其它AOSP构建一样，只不过输出是在特定产品文件夹中。

```
$ 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

此外，看看system/build.prop文件，它包含各种全局属性，这些属性与配置和构建相关。

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.20110921.190849

```
test-keys
    ro.build.fingerprint=generic/full_coyotepad/coyotepad:2.3.4/GINGERBREAD
    /eng.karim.20110921.190849:eng/test-keys
    # end build properties
    ...
```

为了让AOSP运行在新的设备上，工作还远远没有结束。

添加一个替换应用程序

覆盖是AOSP一种机制，允许设备制造商改变提供的资源（如应用程序），而无需实际修改包含在AOSP中的原始资源。要使用此功能，必须创建一个覆盖树，并告诉它构建系统。创建一个覆盖最简单的位置就是设备特定目录：

```
$ cd device/acme/coyotepad/
$ mkdir overlay
```

为了告诉构建系统考虑覆盖，需要修改 full_coyotepad.mk：

```
DEVICE_PACKAGE_OVERLAYS := device/acme/coyotepad/overlay
```

比方说，我们要修改的launcher2的默认字符串:

```
$ mkdir -p overlay/packages/apps/Launcher2/res/values
$ cp aosp-root/packages/apps/Launcher2/res/values/strings.xml \
> overlay/packages/apps/Launcher2/res/values/
```

然后，随意修改副本的strings.xml以符合需求。设备将有一个的launcher2的自定义字符串，但默认的launcher2仍会有其原有的字符串。所以，如果有人依赖于您使用相同的AOSP源用于其它产品的构建，他们仍然会得到原来的字符串。当然，可以取代大多数这样的资源，包括图像和XML文件。因此，只要文件被放在device/acme/coyotepad/overlay/中，构建系统会使用它们。