- 全面剖析进程 / 线程、内存管理、Binder 机制、GUI 显示系统、多媒体管理、输入系统、Android Dalvik/Art 虚拟机、Android 的安全机制、Gradle 自动化构建工具等核心知识在 Android 系统中的设计思想
- 通过大量图片与实例来引导读者学习，以求尽量在源码分析外，为读者提供更易于理解的思维路径
- 由浅入深，由总体框架再到细节实现，帮助读者彻底理解 Android 内核的实现原理

异步图书
www.epubit.com.cn

深入理解 Android 内核设计思想

第 2 版 | 上册

林学森 ◆ 著

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

深入理解 Android 内核（第 2 版）

——Thinking In Android（上册）

林学森　编著

人 民 邮 电 出 版 社
北　京

内 容 提 要

全书从操作系统的基础知识入手，全面剖析进程/线程、内存管理、Binder 机制、GUI 显示系统、多媒体管理、输入系统、虚拟机等核心技术在 Android 中的实现原理。书中讲述的知识点大部分来源于工程项目研发，因而具有较强的实用性，希望可以让读者“知其然，更知其所以然”。本书分为编译篇、系统原理篇、应用原理篇、系统工具篇，共 4 篇 25 章，基本涵盖了参与 Android 开发所需具备的知识，并通过大量图片与实例来引导读者学习，以求尽量在源码分析外为读者提供更易于理解的思维方式。

本书既适合 Android 系统工程师，也适合于应用开发工程师来阅读，从而提升 Android 开发能力。读者可以在本书潜移默化的学习过程中更深刻地理解 Android 系统，并将所学知识自然地应用到实际开发难题的解决中。

◆ 编　　著　林学森
责任编辑　张　涛
责任印制

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷

◆ 开本：800×1000　1/16
印张：39.5
字数：1037 千字　　2017 年 7 月第 2 版
印数：1 –　000 册　　2017 年 7 月北京第 1 次印刷

定价：　　元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

第 2 版前言

Android 系统的诞生地——美国硅谷。

Google 大楼前摆放着 Android 的最新版本雕塑，历史版本则被放置在 Android Statues Park 中

写第 2 版前言时，笔者刚好在美国加州硅谷等地公事出差访问。其间我一直在思考的问题是，美国硅谷（Silicon Valley）在近几十年时间里长盛不衰的原因是什么？技术的浪潮总是一波接着一波的，谁又会在不远的将来接替 Google 的 Android 系统，在操作系统领域成为下一轮的弄潮儿？我们又应该如何应对这种“长江后浪推前浪”的必然更迭呢？

从历史的长河来看，新技术、新事物的诞生往往和当时的大背景有着不可分割的关系。如果我们追溯硅谷的发展史，会发现其实它相对于美国很多传统地区来说还是非常年轻的。“硅谷”这个词是在 1971 年的“Silicon Valley in the USA”系列报导文章中才首次出现的。20 世纪四五十年代开始，硅谷就像一匹脱了缰的野马一般，“一发不可收拾”。从早期的 Hewlett-Packard 公司，到仙童、AMD、Intel 以及后来的 Apple、Yahoo!等众多世界一流企业，硅谷牢牢把握住了科技界的几次大变革，成功汇集了美国 90%以上的半导体产业，逐步呈现出“生生不息”的景象。

但为什么是硅谷，而不是美国其他地区成为高科技行业的“发动机”呢？

古语有云，“天时、地利、人和，三者不得，虽胜有殃”。

现在我们回过头来看这段历史，应该说硅谷早期的发展和当时的世界大环境有很大关系——更确切地说，正是美国国防工业的发展诉求，才给了硅谷创业初期的“第一桶金”。只有“先活下来，才有可能走得更远”。而接下来社会对半导体工业需求的爆炸式增长，同样让硅谷占据了“天时”的优势，再接再厉最终走上良性循环。

密密麻麻的硅谷大企业

（引用自 cdn.com）

硅谷的“地利”和“人和”，可能主要体现在：

（1）Stanford University

斯坦福大学校园

Stanford University 在硅谷的发展过程中起到了非常关键的作用。20 世纪 50 年代的时候，这所大学还并不是很起眼，各方面条件都比较糟糕，她的毕业生也多数会去东海岸寻求就业机会。后来她的一位教授 Frederick Terman 看到了产业和学术的接合点，从学校里划分出一大块空地来鼓励学生创业，并且指导其中两位学生创立了 Hewlett-Packard 公司。随后的几年他又成立了

Stanford Research Park，这也同时是后来全球各高科技园区的起点，并吸引了越来越多的公司加入。在那段时间里，相信起到核心催化作用的是“产”+“学”的高度结合——将科技产品不断推陈出新产生经济效益，然后再回馈到研究领域。在几十年的跨度里，很多顶尖公司（Google、Yahoo!、HP 等）的创始人都出自该校。有统计显示 Standford 师生及校友创造了硅谷一半以上的总产值，其影响力可见一斑。

（2）便利的地理环境

整个硅谷地区面积并不是很大，属于温带海洋性气候，全年平均温度在 13℃～24℃，污染很小。同时，它依林傍海，陆、海、空都可以很好地与外界相连，这样一来自然有利于人才的引入。

（3）鼓励创新，完善的专利保护机制

从法律上讲，硅谷每年有超过 4000 项的专利申请，工程师和律师的比例达到了 10∶1。在创新点得到保护的同时，也使得初创公司能够得到进一步的发展，从而避免它们被扼杀在摇篮中。从观念上来说，硅谷人对知识产权还是非常尊重的，他们大多认为剽窃是没有技术含量的，相当于“涸泽而渔”。

（4）完善的风投体系，并容忍失败

事实上在硅谷创业，其成本和失败率都很高——其中能存活 3～5 年的公司只有 10%～20%。一方面，风险投资方需要高度容忍这样的失败率；另一方面，在允许快速试错的同时，风险投资方又可以从某些成功中获得巨大收益——硅谷就是一个可以达到这种矛盾平衡的神奇所在地。

“三十年河东，三十年河西”，技术的浪潮总是在不断演进的。从 Symbian、Black Berry，到 Android、iOS，历史经验告诉我们没有一项技术是会永远一成不变的。所以我们在技术领域的探索过程中，既要拿“鱼”，更要学会“渔”——前者是为当前的工作而努力，后者则是为我们的未来做投资。以 Android 操作系统为例，事实上我们除了“知其然”外，还更应该学习它的内部设计思想——即“知其所以然”。当我们真正地理解了那些“精华”所在以后，那么相信以后再遇到任何其他的操作系统，就都可以做到“触类旁通”了。也只有这样，或许才能在快速变革的科技领域中把握住脉搏，立于不败之地。

林学森

于美国硅谷

关于本书第 2 版

在第 1 版上市的这两年时间里，不断有读者来信分享他们阅读本书时的感想和心得，笔者首先要在这里衷心地向大家说声感谢！正是你们的支持和肯定，才有了《深入理解 Android 内核设计思想》（第 2 版）的诞生。

其中有不少读者提到了他们希望在本书后续更新中看到的内容，包括 Android 虚拟机的内部实现原理、Android 的安全机制、Gradle 自动化构建工具等——这些要求都在本次版本更新中得到了体现。

需要特别说明的是，第 2 版中的所有新增和有更新的部分都是基于 Android 最新的 N 版本展开的。由于 Android 版本的更新换代很快，且版本间的差异巨大，导致书中很多内容几乎需要全部重写。另外笔者写书都是在下班后的业余时间进行的，所以即便是每晚奋笔疾书到深夜，再加上周末和节假日时间（如果没有加班工作的话），最后发现更新全书所需时间依然要大于 Android 系统的发布间隔。为了让读者可以早日阅读到大家感兴趣的内容，本次版本的部分章节保留了第 1 版的原有内容——本书下一次再版时会争取将它们更新到 Android 的最新版本。这一点希望得到大家的谅解，谢谢！

致谢

感谢我目前任职公司的领导和同事们，是你们的帮助和支持，才让我更快地融入到了这个大家庭中。在一个到处都是“聪明人”和具有“狼性奋斗者”精神的公司里，每天的进步和知识积累都是让人愉悦的。

感谢人民邮电出版社的编辑，你们的专业态度和处理问题的人性化，是所有作者的“福音”。

感谢我的家人林进跃、张建山、林美玉、杨惠萍、林惠忠、林月明，没有你们的鼓励与理解，就没有本书的顺利出版。

感谢我的妻子张白杨的默默付出，是你工作之外还无怨无悔地在照顾着我们可爱的宝宝，才让我有充足的时间和精力来写作。

感谢所有读者的支持，是你们赋予了我写作的动力。另外，因为个人能力和水平有限，书中难免会有不足之处，希望读者不吝指教，一起探讨学习，作者的联系方式是：**xuesenlin@alumni.cuhk.net**。编辑联系和投稿邮箱是：zhangtao@ptpress.com.cn。本书读者交流 QQ 群为 216840480。

作者

第1版前言

写本书的原因

4 次大幅改版，N 次修订，前后历时近 3 年，本书终于要与读者见面了。

在这 3 年的时间里，Android 系统不断更新换代，书本内容也尽可能紧随其步伐——我总是会在第一时间下载到工程源码，然后系统性地比对和研究每次改版后的差异。可以说本书伴随着 Android 的高速发展，完整地见证了它给大家带来的一次又一次惊喜。

在这么长的写作跨度中，有一个问题始终萦绕在我的脑海中，即“为什么写这本书”？

市面上讲解操作系统的著作很多，主要风格有两种。

- 理论型

高校中采用的操作系统教材多数属于这种类型。它们主要阐述通用的计算机理论与原理，一般不会针对某个具体的操作系统做详细剖析。这类书籍是我们进入计算机科学的“敲门砖”。只有基础打得扎实，研究市面上任何一款操作系统才能做到“有的放矢”。

- 实用型

这类书籍以讲解某个具体的操作系统为主，如市面上就有非常多的关于 Windows 和 Linux 系统的。前者因为不开源，谁也不可能深入代码级别进行讲解；而后者则恰恰相反，任何人都能轻松获取到完整的内核源码。在 Linux 之父经典名言“Read the f***king Source Code”的鼓励下，无数有志之士投入到“代码汪洋”的分析中，从中细细感受大师们的设计艺术。

那么本书属于什么类型呢？个人认为更贴切地说，就是上面两种的结合。

本书的一个主要宗旨是希望读者可以由浅入深地逐步理解 Android 系统的方方面面。因而在每章节内容的编排上，采用由整体到局部的线索铺展开来——先让读者有一个直观感性的认识，明白“是什么”“有什么用”，然后才剖析“如何做到的”。这样的一个好处是读者在学习过程中不容易产生困惑；否则如果直接切入原理，长篇大论地分析代码，仅一大堆函数调用就可能让人失去学习的方向。这样的结果往往是，读者花了非常多的时间来理清函数关系，但始终不明白代码编写者的意图，甚至连这些函数想实现什么功能都无法完全理解。

本书希望可以从更高的层次，即抽象的、反映设计者思想的角度去理解系统。而在思考的过程中，大部分情况下我们都将从读者容易理解的基础知识开始讲起。就好比画一张素描画一样——先给出一张白纸，勾勒出整体框架，然后针对重点部位细细加工，最后才能还原出完整的画面。另外，本书在对系统原理本身进行讲解的同时，也最大程度地结合工程项目中可能遇到的难点，理论联系实际地进行解析。希望这样的方式既能让读者真正学习到 Android 系统的设计思想，也能学有所用，增加一些实际的项目开发经验和技巧。

本书的主要内容

细心的读者会发现本书章节中包含了“Android 和 OpenGL ES”“信息安全基础概述”等看似与本书无关的内容——有些人可能会产生疑问，是否有此必要？

根据我们多年的 Android 项目开发和培训经验，答案就是“非常有必要”。举个例子，Android 的显示系统是围绕 OpenGL ES 来展开的，后者是它的“根基”。但另外，并非所有开发人员都深谙 OpenGL ES。这样导致的结果就是他们在学习显示系统的过程中，有一种“四处碰壁”的感觉——实践证明，正是这些因素直接打击到了大家学习 Android 系统的信心。

因此我们在讲解系统实现原理之前，会最大程度地为读者提炼出所需的背景知识。有了这样的铺垫，相信对大家学习 Android 内核大有裨益。

本书在内容选择上依据的是“研发人员（包括系统开发和应用程序开发）参与实际 Android 项目所需具备的知识”，因而具有较强的实用性。全书共分为 4 篇，涵盖了编译、系统原理、应用原理和系统工具等多个方面。

其中第一篇不仅详细介绍了 Android 源码的下载及编译过程，为读者呈现了“Hello World”式的入门向导——更为重要的是，结合编译系统的架构和内部原理，为各厂家定制自己的 Android 产品提供了参考范例。

Android 本质上只是市面上众多主流的操作系统之一。所以在系统原理的讲解过程中，我们将首先引导读者从计算机体系结构、经典的操作系统理论（比如进程/线程管理、进程间通信等）的角度来思考问题——包括 Android 在内的任何操作系统内核在实现过程中都“逃”不出这些经典的理论范畴。本书虽然是剖析 Android 系统的，但更希望读者可以从中学到“渔”，而不仅仅是“鱼”。

从动态运行的角度来理解，Android 内核是由众多系统服务组成的，如 ActivityManagerService、GUI 系统中的 SurfaceFlinger、音频系统中的 AudioFlinger、输入系统 InputManagerService 等。而各服务之间通信的基础就是 Binder 机制。本书在阐述它们错综复杂的关系中，遵循由“整体到局部”“由点及面”的科学方法，将知识点深入浅出地铺展开来，希望为读者全面理解 Android 内核提供“思维捷径”。

与其他讲解 Android 应用程序的书籍不同，本书在分析 APK 应用程序时的立足点是它的内部实现原理。如 Intent 匹配规则、应用程序的资源适配过程、字符编码的处理、Widget 机制、应用程序的编译打包等都是应用开发人员在工作中经常会遇到的难题。通过系统性地解析隐藏在这些实现背后的原理，有助于他们彻底摆脱困惑，加深对应用开发的理解。

不论是系统工程师还是应用开发人员，Android 调试工具都至关重要。但我们在实际工作中发现，不少研发人员对这些工具“只知其一，不知其二”。因而系统工具篇中将针对常用调试工具进行全面解析，希望由此可以让大家学习到如何“举一反三”，真正把它们的作用发挥得“淋漓尽致”。

本书的主要特点

（1）通过大量情景图片与实例引导读者学习，以求尽量在源码分析外为读者提供更易于理解的思维路径。

（2）作者在展开一个话题时，通常会由浅入深、由总体框架再到细节实现。这样可以保证读者能跟得上分析的节奏，并且“有根有据可循”，尽可能防止部分读者阅读技术书籍时“看了后面忘了前面”的现象。

（3）目前市面上不少 Android 书籍仍停留在 Android 2.3 或者更早期的版本。虽然原理类似，但对于开发人员来说，他们需要与项目研发相契合的技术书籍。本书希望尽可能紧随 Android 的更新步伐，为读者了解最新的 Android 技术提供帮助。

（4）本书的出发点仍是操作系统的经典原理，并以此为根基扩展分析 Android 中的具体实现机制——贯穿其中的是经久不衰的理论知识。

（5）本书所阐述的知识点大部分来源于工程项目研发的经验总结，因而具有较强的实用性，希望可以让读者“知其然，更知其所以然”。做到真正贴近读者，贴近开发需求。

致谢

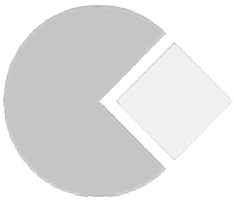
感谢王益民董事长、钟宝英女士长期以来的关心、信任和支持——你们在很多方面都是我们学习的楷模。衷心祝愿王总企业蒸蒸日上、再创辉煌；衷心祝愿钟小姐事事顺心如意、永葆青春。

感谢人民邮电出版社的编辑，你们的专业态度和处理问题的人性化，是所有作者的“福音”。

感谢我的家人、长辈和朋友林进跃、林美玉、林惠忠、刘冰、林月明、温艳，感谢你们长期以来对我工作和生活上无微不至的关心和支持。

感谢所有读者的支持，是你们赋予了我写作的动力。另外，因为个人能力和水平有限，书中可能还有不足之处，希望读者不吝指教，一起探讨学习，作者的联系方式是：**xuesenlin@alumni.cuhk.net**。编辑联系和投稿邮箱是：zhangtao@ptpress.com.cn。

目　录



第 1 篇

Android 编译篇

第 1 章　Android 系统简介

美国当地时间 2015 年 5 月 28 日，“Google I/O 2016”大会在旧金山市的 Moscone Center 举行。会议公布的官方数据如下：

- 全球已经激活的 Android 设备达到 9 亿次；
- Google Play 中收录了超过 70 万的应用程序；
- 应用程序安装量达到 480 亿次；
- 132 个以上的国家或地区销售 Android 设备；
- 超过 190 个国家或地区可以下载到免费的 Android 应用程序。

2016 年，Google 则直接把大会地址从传统的 Moscone Center 改到了 Shoreline 公园的户外，吸引了成千上万来自全球各地的科技爱好者。

从 2008 年 9 月 Google 发布 Android 1.0 版本开始，Android 已经走过了 8 个年头。在这短短的几年间，这个以机器人为 Logo 的操作系统不仅席卷了全球各地的手机市场，而且与 iOS、Windows Phone 形成三足鼎立之势，更渗透到传统与新兴电子产业的方方面面。越来越多的电子产品已开始采用 Android 系统，如 Android 电视、平板电脑、MP4 等与人们日常生活息息相关的电子设备。

那么，Android 势不可当的魅力从何而来呢？本章将试着以 Android 系统的发展历史为主线，先为读者提供最直观的背景知识，从而为以后的“透过现象看本质”打下一定的基础。

1.1 Android 系统发展历程

“Android”一词先天就充满着天才们改变世界的梦想味。虽然一件杰出的作品并不能只靠“名号”，但毋庸置疑的是，一个叫得响又耐人寻味的名称总会使人产生不自觉的亲近感。这或许就是每个 Android 版本都会有个代号的原因。下面来看看各个版本对应的 Android “外号”，如表 1-1 所示。

表 1-1　　Android 各版本的代号

Code name	Version	API level
(no code name)	1.0	API level 1
(no code name)	1.1	API level 2
Cupcake（纸杯蛋糕）	1.5	API level 3, NDK 1
Donut（甜甜圈）	1.6	API level 4, NDK 2
Éclair（松饼）	2.0	API level 5
Éclair	2.0.1	API level 6
Éclair	2.1	API level 7, NDK 3
Froyo（冻酸奶）	2.2.x	API level 8, NDK 4

续表

Gingerbread（姜饼）	2.3 - 2.3.2	API level 9, NDK 5
Gingerbread	2.3.3 - 2.3.7	API level 10
Honeycomb（蜂巢）	3.0	API level 11
Honeycomb	3.1	API level 12, NDK 6
Honeycomb	3.2.x	API level 13
Ice-creamSandwich（冰激凌三明治）	4.0.1 - 4.0.2	API level 14, NDK 7
Ice-creamSandwich	4.0.3 - 4.0.4	API level 15
Jelly Bean（果冻豆）	4.1.x	API level 16
Jelly Bean	4.2.x	API level 17
Jelly Bean	4.3.x	API level 18
KitKat	4.4.x	API level 19
KitKat with wearable extensions	4.4W	API level 20
Lollipop	5.0.1	API level 21
Lollipop	5.1.1	API level 22
Marshmallow	6.0	API level 23
Nougat	7.0	API level 24
Nougat	7.1.1	API level 25

“Android”一词来源于法国作家 Auguste Villiers de l'Isle-Adam 的科幻小说《L'ève future》（未来夏娃），是机器人的意思。因此，最初每个系统版本的命名也都是以全球著名的机器人为参考的，如“AstroBoy”。后来由于版权问题，才改为以食物的方式取名。不过 Android 的 Logo 仍然是机器人的形象，如图 1-1 所示。

▲图 1-1 Android 官方 Logo

和很多著名的科技企业一样，Android 的创始人 Andy Rubin 也是一个技术狂人。在创立 Android 公司前，他曾完成多项当时被称为“过于超前”的产品研发，并取得了一定的成绩。而创办 Android，最初的目的是提供一款开放式的移动平台系统。从 2003 年 10 月 Andy Rubin 开始启动这一系统的研究，Android 便正式走上历史的舞台。以下是关于这个系统的一些重要历史事件。

- 2003 年 10 月，Andy Rubin 在加利福尼亚州成立 Android 公司。
- 2005 年 4 月，Google 收购 Android。
- 2007 年 11 月，Google 成立 OHA（Open Handset Alliance）联盟，成员包括 Google、Broadcom、HTC、Intel、LG、Marvell、Motorola、NVIDIA、Qualcomm、Samsung 等通信行业和芯片制造领域的巨头。随后几年，这个联盟又陆续有不少公司加入，如著名的 Arm 公司、中国的华为等。
- 2007 年 11 月，Google 成立“Android Open Source Project”（AOSP）。这一项目的起步标志着 Android 系统首次公开面向全世界的开发者与使用者。

AOSP 的宗旨是：

Android Open Source Project is to create a successful real-world product that improves the mobile experience for end users。

因为是开源开放的组织，所以意味着每个人都可以参与进来，并为整个项目的发展添砖加瓦。如果读者有意愿成为其中的一员，可以参考该组织的相关说明（http://source.android.com/source/index.html）。

- 2007 年 11 月，Android Beta 版本发布。
- 2007 年 11 月，Android 第一个 SDK 版本发布。
- 2008 年 9 月，Android 1.0 版本正式发布。

至此，Android 版本的发布驶入正常轨道，保持着每年多次升级的速度，并加入越来越多的创新功能。

关于 Android 每个版本的特性及改版后一些重要变化的详细说明，请参阅官方网站（http://source.android.com/source/overview.html）。相信经过仔细比对各个版本的变化，读者会发现 Android 系统确实一直在秉承其“为终端用户提供更好的移动设备体验”的宗旨。

1.2 Android 系统特点

在这一节中，我们将从观察者的角度来客观评价 Android 系统某些突出的特点。这些分析中既包含了其值得肯定的诸多优点，也不吝指出其需要持续改善的地方。只有正确全面地了解一个系统所存在的优缺点，才可能在开发的过程中“知己知彼，百战不殆”，真正让系统为我们所用。

1. 开放与扩展性

相对于 iOS 和 Windows Phone 阵营，Android 操作系统最大的特点就是开放性；而且有别于个别开源项目的“藏藏掖掖”和“犹抱琵琶半遮面”，Android 几乎所有源码都可以免费下载到。无论是公司组织，还是个人开发者，Android 对于下载者基本没有限制，也没有下载权限的认证束缚。关于如何下载系统源码的完整描述，请参考下一章节。

当然，这并不代表开发者可以随意使用 Android 源码。事实上，Android 遵循的是 Apache 开源软件许可证。因此，所有跨越许可证规定范畴的行为都将是被禁止的。希望了解更多 Apache 协议条款详情的读者，可以自行查阅其官方说明（http://www.apache.org/）。

不过在大部分情况下，Android 操作系统仍然被认为是“高度自由”的。这也是越来越多的厂商选择 Android 作为下一代产品基础平台最主要的原因之一。可以想象，在其他操作系统对其授权的设备动辄收取每台高达几十甚至数百美元专利费的情况下，采用 Android 开源系统理论上就意味着降低成本。

另外，由于整个操作系统是开源的，从而给诸多产品制造商、软件开发商提供了创新的土壤环境。各厂商可以根据自己的需求，来完成对原生态系统的修改。大多数情况下，这种修改只是基于上层 UI 交互的“二次包装”，而保留底层系统的大框架。这就好比 Google 为大家免费提供了已经盖好的办公大楼，虽然是毛坯房，但相较于“万丈高楼平地起”的艰辛，显然已经为我们节约了大量的项目时间；而且我们可以通过“装修”把主要精力倾注在用户看得到的地方，从而更大限度地摆脱“产品同质化”。事实上，目前全球范围内已经有非常多这样的“装修范例”。大到跨国企业、运营商，小到一些初创的设计公司，都选择在 Android 系统上进行“界面”改造，再冠以新的操作系统名号。其中也不乏一些成功者，根据不同的地域环境、文化差异、使用习惯而定制出新的系统——这些具有“本地化”风格的“办公楼”往往比原生态系统更贴近当地消费者，因此受到热烈追捧。

而这一切，都要归功于 Android 系统的开放性。

2. 合理的分层架构

要学习 Android 系统，就不得不提它的分层架构。早期版本的 Android 系统框架包括 4 层，即 Linux Kernel、Library and Runtime、Application Framework 及 Application。后来因为版权相关

原因在 Kernel 层之上新增了一个 Hardware Abstraction Layer。我们会在后续小节对各层功能适当地展开讨论。

由此可见，Android 系统是一个“杂合体”，即便说其“包罗万象”也一点不为过。它包括了 prebuilt、bionic 等在内的不少开源项目。管理这些项目显然不是件容易的事，这也是 Android 系统提供 Repo 工具，而不是直接使用 Git 来进行版本管理的原因之一（详见后续章节的描述）。

在面对这么多独立项目的时候，合理的分层架构就显得异常重要——既要保证系统功能的完整性，也要确保各项目的相对独立性。Android 系统成功地做到了这一点，整个软件栈条理清晰，分工明确。一方面，它将底层复杂性与移植难度尽可能隐藏起来；另一方面，则提供尽可能方便的上层 API 接口，为开发者设计实现各种应用程序打下了坚实的基础。

3. 易用强大的 SDK

SDK（Software Development Kit）是操作系统与开发者之间的接口，也可以看成一个系统对外的窗口。对于广大的开发者而言，能否借助这个工具在尽可能短的周期内设计出符合需求而又稳定可靠的应用程序，是评判一个操作系统 SDK 好坏的重要标准之一。

Android 系统的大部分应用程序可以基于 Java 来开发。如果读者曾参与过大型的 C/C++研发项目（特别是面向嵌入式系统的），一定不会忘记加班加点解决内存泄露或者空指针异常的那些无眠夜。Java 语言对这些软件开发中最令人头疼的问题进行了强有力的改造，不但提供了垃圾回收机制，而且彻底隐藏了指针的使用。即便程序出现了崩溃，通常情况下也可以根据调用栈及各种 Log 来定位出问题的根源。这无疑为我们快速解决问题、保证程序稳定性提供了很好的平台基础。

Android 系统通过总结应用程序的开发规律，提供了 Activity、Service、Broadcast Receiver 及 Content Provider 四大组件；并且和 MFC 类似，设计了人性化的向导模式来帮助开发者便捷地生成工程原型。可以说，这些都为项目开发节约了不少宝贵时间。

另外，Android SDK 覆盖面相当广，且仍在持续扩充中。从线程管理、进程间通信等程序设计基础到各种界面组件的应用，只要是开发者能想到的，几乎都可以在 SDK 中找到现成的调用接口。而对一些界面特效的封装，使得开发者可以高效地设计出各种绚丽的 UI 效果，进而让 Android 系统加分良多。

4. 不断改进的交互界面

Android 版本的更迭是一件让无数人兴奋的事。除了那些令人眼前一亮的新功能外，不断改进的用户交互界面也是吸引用户的一个重要因素。我们可以明显地从新老版本的对比中寻找到 Google 追求绝佳用户体验的决心。

下面先来看看 Gingerbread（2.3 版本）的 Launcher 与 Camera 界面，如图 1-2 所示。

然后来看看后续版本上的变化，如图 1-3 所示。

▲图 1-2 Launcher 和 Camera 界面

▲图 1-3 后续 Android 版本变化

可以发现，新的版本相较于以前，不但在 UI 界面的色彩搭配、布局上有了很大提高，用户交互也更趋于人性化。这种对于用户最直观的“艺术盛宴”展示，促使越来越多的人投身到 Android 阵营中。

5. 逐步完善的生态系统

IT 业界长期以来都有一个共识——开发一个操作系统（OS）并不是最难的，而基于这个新的操作系统建立完整的生态系统才是最大的难点。用一句老话来说，颇有点“打江山易，守江山难”的味道。

那么，什么是基于 OS 的生态系统（ecosystem）呢？虽然我们一再听到媒体在大肆宣扬这个词，但目前还没有人能给出权威、严谨的解释来阐述这个特殊“生态系统”的定义与形成。本书下面所提出的释义也未必能完整解读这个词，读者可以带着自己的理解深入思考。

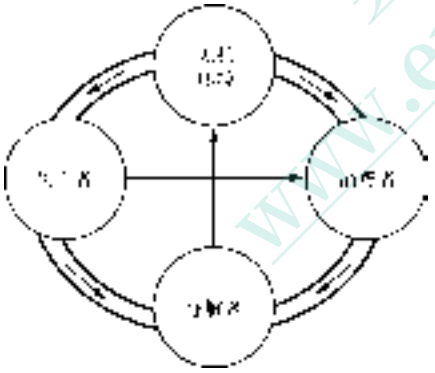
Ecosystem 原本是生态学中的一个概念。简单而言，它体现了一定时间和空间内能量的可循环平衡流动，例如，自然界的生态系统组成如下。

- 无机环境，包括太阳、有机物质、无机物质等非生物环境。
- 生产者。
- 消费者。
- 分解者。

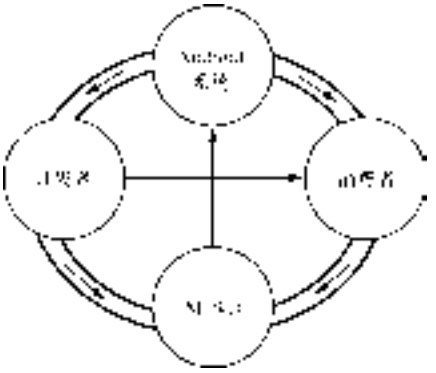
因此，它们之间所体现出的能量循环如图 1-4 所示。

生产者依靠无机环境制造食物，并实现自养；而消费者则需要消耗其他生物来生存发展；分解者最终将有机物分解为可被生产者重新利用的物质。这样，就构成了整个生态链的循环。

针对 Android 生态系统，我们可以得到以下的类比，如图 1-5 所示。



▲图 1-4　自然生态系统的能量循环

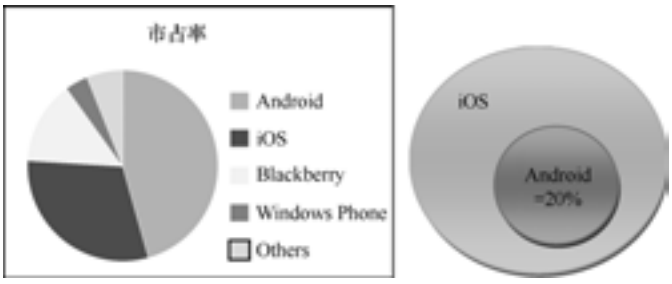


▲图 1-5　Android 生态系统假想

在 Android 生态系统中：

- Android 系统提供了底层基础平台；
- 开发者通过研发新产品来获取利润，或者提供相应服务；
- 消费者使用这些产品或服务来满足自身需求；
- Market 提供了开发者与消费者间资金支付与交易的平台，加快了生物间的“能量”流动。

当然，这只是本书对 Android 生态系统的一个初步设想，实际情况一定更复杂。但客观来说，Google 一方面既在努力打造“双赢”的市场机制，以吸引更多的开发者介入；另一方面也在提高为消费者服务的能力，如图 1-6 所示。



▲图 1-6 各操作系统平台占有率及开发者赢利对比

虽然最新的调查报告显示，依靠 Android 软件赢利的开发者寥寥无几，还远远比不上 iOS 系统赢利模式成熟（见图 1-6）。但同时也应该看到，随着整个 Android 市场占有率的提升以及 Google 一系列措施的实行，Android 生态系统正在逐步完善，前景一片光明。

6. 阵营良莠不齐

开源是一把“双刃剑”，它带来的一个突出问题就是阵营混乱。和一些操作系统需要收取高额的加盟费用不同，Android 的免费开源大大降低了开发商的准入门槛。因此，出现了“人人都可以做手机”的局面。无论是资本、研发实力雄厚的大型企业，还是初出茅庐没有太多经验的小公司，都在不断进入这个生态圈。这既是 Android 系统的优势，同时也是隐患。

因为每个厂商都可以根据自己的需求来改造原生态系统，从而难免造成整个 Android 阵营的分裂；而且，有的开发商也在努力基于 Android 建立自己的生态系统。这无疑会对 Android 的整体发展产生一定的影响。如果这些状况在今后一段时间里无法得到解决，那么很可能阻滞 Android 的进一步发展壮大。

7. 系统运行速度有待改善

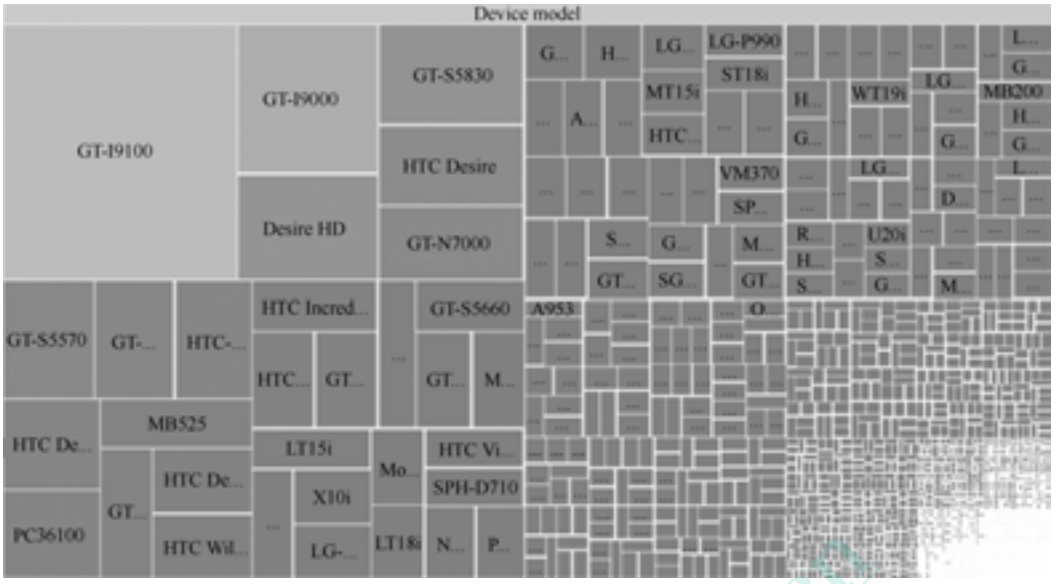
使用过 Android 相关产品的用户一定有这样的体验，那就是开机慢。一个针对目前市面上主流 Android 设备的不完全统计显示，Android 产品的平均开机时间超过了 1 分钟，有的甚至达到 5 分钟以上。对于某些需要快速实时响应的电子设备而言（比如车载电子导航一体机，往往需要在汽车启动后非常短的时间内完成操作系统开机，以显示倒车影像），这样的“龟速”显然是很难让人接受的。

值得欣慰的是，Google 也正致力于运行速度的改进。随着新版本的不断发布，我们已经可以明显地感受到 Android 在这方面所做的努力与成效。

8. 兼容性问题

对于 Android 平台的应用开发者而言，最头疼的恐怕并不是某项创新功能的研发，而是对市面上多种设备的适配。在这方面，iOS 的开发人员有绝对的优势，因为他们面对的往往只是一款机器（如 iPhone6、iPhone7），而且屏幕尺寸、分辨率等系统属性也都是固定已知的，如图 1-7 所示。

Android 系统由于开源、生产商众多，致使产品形态五花八门。以手机为例，为消费者所熟识的全球大型 Android 手机开发商就已经超过了 20 个。而这些厂商还有各自不同的产品型号——这也就意味着屏幕大小、分辨率等各种硬件参数的差异。按照目前行业的普遍经验，开发一款成熟的 Android 手机应用软件，需要适配 200 款以上不同厂商的手机，以保证软件发布后不至于出现大规模的用户投诉。如果是面向海外市场，则需要兼容的终端产品数量可能还会更多。



▲图 1-7　OpenSignalMaps 对市面上主流 Android 手机品牌的跟踪结果

虽然 Android 针对这一问题有一定的解决方法（详见本书应用原理篇的相关章节），但以实际开发经验来看，暂时还没有很好地解决难题的方法。

1.3 Android 系统框架

Android 系统框架如图 1-8 所示。

▲图 1-8　Android 系统 5 层框架图

注意	引用自 2008 年的 Google I/O 大会《Anatomy & Physiology of an Android》主题演讲，作者 Patrick Brady。

前面说过，Android 系统是由众多子项目组成的。从编程语言的角度来看，这些项目主要是使用 Java 和 C/C++来实现的；从整体系统框架而言，分成内核层、硬件抽象层、系统运行库层、应用程序框架层以及应用程序层。本书的一个主要宗旨是希望读者可以由浅入深地逐步理解 Android 系统的方方面面。因而在每章节内容的编排上，我们采用了由整体到局部的线索来铺展开——先让读者有一个直观感性的认识，明白“是什么”“有什么用”，然后才剖析“如何做到的”。这样做的一个好处是读者在学习的过程中不容易产生困惑。否则如果直接切入原理，长篇大论地分析代码，仅一大堆函数调用就可能让人失去学习的方向。这样的结果往往是读者花了非常多的时间来理清函数关系，但始终不明白代码编写者的意图，甚至连这些函数想实现什么功能都无法完全理解。

本书希望可以从更高的层次，即抽象的、反映代码设计思想和设计者初衷的角度去理解系统。而在思考的过程中，大部分情况下我们都将从读者容易理解的基础开始讲起。就好比画一张素描，先给出一张白纸，勾勒出整体的框架，然后针对重点部位细细加工，最后才能还原出完整的画面。另外，本书在对系统原理本身进行讲解的同时，也最大程度地结合工程项目中可能遇到的问题，理论联系实际地进行解析。希望这样的方式既能让读者真正学习到 Android 系统的设计思想，也能学有所用，增加一些实际的项目开发经验和技巧。

接下来的内容将对 Android 系统的 5 层框架做一个简单描述。

- 内核层

Android 的底层是基于 Linux 操作系统的。从严格意义上来讲，它属于 Linux 操作系统的一个变种。Android 选择在 Linux 内核的基础上来搭建自己的运行平台有几个好处。

首先，避开了与硬件直接打交道。Linux 经过多年的发展，这方面工作正是它的强项，其表现可以说相当优秀。更为难能可贵的是，Linux 本身也是开源的，所以 Android 系统没有必要花费额外的时间去做重复工作。

其次，基于 Linux 系统的驱动开发可扩展性很强。这对于嵌入式系统而言非常重要，因为每款产品在硬件上或多或少都会有差异，如果驱动开发不能做到高度可扩展和易用性，那么 Android 系统的移植工作将是无止境的噩梦。

值得一提的是，Android 的工程项目中并没有包括内核源码——内核源码的具体下载方式可以参见下一章节。

- 硬件抽象层

大家可能都有这样的疑问，既然 Linux 内核是专职与硬件打交道的，为什么又杀出个“程咬金”硬件抽象层（HAL）呢？没错，这个“人物”一开始并没有出现在 Android 的“剧本”中，其出场是有一定历史原因的。

HAL 的第一次亮相要追溯到 2008 年的 Google I/O 大会上。当时 Google 员工 Patrick Brady 发表了一篇名为《Anatomy & Physiology of an Android》的演讲，并在其中提出了带 HAL 的 Android 新架构。根据这份文档的描述，HAL 是：

（1）User space C/C++ library layer；

（2）Defines the interface that Android requires hardware “drivers” to implement；

（3）Separates the Android platform logic from the hardware interface。

也就是说，它希望通过定义硬件“驱动”的接口来进一步降低 Android 系统与硬件的耦合度。另外，由于 Linux 遵循的是 GPL 协议（注意，Android 开源项目基于 Apache 协议），意味着其下的所有驱动都应该是开源的——这点对于部分厂商来说是无法接受的。因而，Android 提供了一种“打擦边球”的做法来规避这类问题。我们会在后续章节中继续讲解关于 HAL 的更多知识。

- 系统运行库层

这一层中包含了支撑整个系统正常运行的基础库。由于这些库多数由 C/C++实现，因此也被一些开发人员称为“C 库层”，以区别于应用程序框架层。Android 中很多系统运行库实际上都是成熟的开源项目，如 WebKit、OpenGL、SQLite 等。我们并不要求读者去理解所有库的内部原理，这样做不现实。重点在于 Android 系统是如何有机地与这些库建立联系，从而保证整个设备的稳定运作的。

- 应用程序框架层

与系统运行库被称为“C 库层”相对应，应用程序框架层往往被冠以“Java 库”的称号。这是因为框架层所提供的组件一般都用 Java 语言编写而成，它们一方面为上层应用程序提供了 API 接口；另一方面也囊括了不少系统级服务进程的实现，是与 Android 应用程序开发者关系最直接的一层。

- 应用程序层

目前 Android 的软件开发分为两个方向，即系统移植与应用程序的开发。

对于一名出色的应用程序员而言，不仅要了解该使用哪些系统 API 接口去完成一个功能，还要尽可能了解这些接口及其下的系统底层框架是如何实现的。虽然理解系统运行原理对于应用开发者来说并不是必需的，但在很多情况下却可以极大地提高程序员分析问题的能力，也可在产品性能优化方面产生积极的作用。

第 2 章　Android 源码下载及编译

在分析 Android 源码前，首先要学会如何下载和编译系统。本章将向读者完整地呈现 Android 源码的下载流程、常见问题以及处理方法，并从开发者的角度来理解如何正确地编译出 Android 系统（包括原生态系统和定制设备）。

后面，我们将在此基础上深入到编译脚本的分析中，以“庖丁解牛”的方式来还原一个庞大而严谨的 Android 编译系统。

2.1 Android 源码下载指南

2.1.1　基于 Repo 和 Git 的版本管理

Git 是一种分布式的版本管理系统，最初被设计用于 Linux 内核的版本控制。本书工具篇中对 Git 的使用方法、原理框架有比较详细的剖析，建议读者先到相关章节阅读了解。

Git 的功能非常强大，速度也很快，是当前很多开源项目的首选工具。不过 Git 也存在一定的缺点，如相对于图形界面化的工具没那么容易上手、需要对内部原理有一定的了解才能很好地运用、不支持断点续传等。

为此，Google 提供了一个专门用于下载 Android 系统源码的 Python 脚本，即 Repo。

在 Repo 环境下，版本修改与提交流程是：

- 用 Repo 创建新的分支，通常情况下不建议在 master 分支上操作；
- 开发者根据需求对项目文件进行修改；
- 利用 git add 命令将所做修改进行暂存；
- 利用 git commit 命令将修改提交到仓库；
- 利用 repo upload 命令将修改提交到代码服务器上。

由此可见，Repo 与我们在工具篇中讨论的 Git 流程有些许不同，差异主要体现在与远程服务仓库的交互上；而本地的开发仍然是以原生的 Git 命令为主。下面我们讲解 Repo 的一些常用命令，读者也可以拿它和 Git 进行仔细比较。

1. 同步

同步操作可以让本地代码与远程仓库保持一致。它有两种形式。

如果是同步当前所有的项目：

```
$ repo sync
```

或者也可以指定需要同步的某个项目：

```
$ repo sync [PROJECT1] [PROJECT2]…
```

2. 分支操作

创建一个分支所需的命令：

```
$ repo start <BRANCH_NAME>
```

也可以查看当前有多少分支：

```
  $ repo branches
```

或者：

```
$ git branch
```

以及切换到指定分支：

```
$ git checkout <BRANCH_NAME>
```

3. 查询操作

查询当前状态：

```
$ repo status
```

查询未提交的修改：

```
$ repo diff
```

4. 版本管理操作

暂存文件：

```
$git add
```

提交文件：

```
$git commit
```

如果是提交修改到服务器上，首先需要同步一下：

```
$repo sync
```

然后执行上传指令：

```
$repo upload
```

2.1.2　Android 源码下载流程

了解了 Repo 的一些常规操作后，这一小节接着分析 Android 源码下载的全过程。这既是剖析 Android 系统原理的前提，也是让很多新手感到困惑的地方——源码下载可以作为初学者了解 Android 系统的“Hello World”。

值得一提的是，Android 官方建议我们务必确保编译系统环境符合以下几点要求：

- Linux 或者 Mac 系统

在虚拟机上或是其他不支持的系统（例如 Windows）上编译 Android 系统也是可能的，事实上 Google 鼓励大家去尝试不同的操作系统平台。不过 Google 内部针对 Android 系统的编译和测试工作大多是在 Ubuntu LTS(14.04)上进行的。因而建议开发人员也都选择同样的操作系统版本来开展工作，经验告诉我们这样可以少走很多弯路。

如果是在虚拟机上运行的 Linux 系统，那么理论上至少需要 16GB 的 RAM/Swap 才有可能完成整个 Android 系统的编译。

- 对于 Gingerbread(2.3.X)及以上的版本，64 位的开发环境是必需的。其他旧的 Android 系统版本可以采用 32 位的开发环境。
- 需要至少 100GB 以上的磁盘空间才能完成系统的一系列编译过程——仅源码大小就已经将近 10GB 了。
- Python 2.6-2.7，开发人员可以从 Python 官网上下载：www.python.org。
- GNU Make 3.81-3.82，开发人员可以从 Gnu 官网上下载：www.gnu.org。
- 如果是编译最新版本的 Android N 系统，那么需要 Java8(OpenJDK)。后续编译章节我们还会专门介绍。
- Git 1.7 以上版本，开发人员可以从 Git 官网上下载：http://git-scm.com。

要特别提醒大家的是，以下所有步骤都是在 Ubuntu 操作系统中完成的（“#”号后面表示注释内容）。

1. 下载 Repo

```
$ cd ~  #进入 home 目录
$ mkdir bin #创建 bin 目录用于存放 Repo 脚本
$ PATH=~/bin:$PATH #将 bin 目录加入系统路径中
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo #curl
#是一个基于命令行的文件传输工具，它支持非常多的协议。这里我们利用 curl 来将 repo 保存到相应目录下
$ chmod a+x ~/bin/repo
```

注：网上有很多开发者（中国大陆地区）反映上面的地址经常无法成功访问。如果读者也有类似困扰，可以试试下面这个：

```
$curl http://android.googlesource.com/repo > ~/bin/repo
```

另外，国内不少组织（特别是教育机构）也对 Android 做了镜像，如清华大学提供的开源项目（TUNA）的 mirror 地址如下：

```
https://aosp.tuna.tsinghua.edu.cn/
```

下面是 TUNA 官方对 Android 代码库的使用帮助节选：

```
Android 镜像使用帮助
参考 Google 教程 https://source.android.com/source/downloading.html,将 https://android.google
source.com/全部使用 git://aosp.tuna.tsinghua.edu.cn/android/代替即可。
本站资源有限，每个 IP 限制并发数为 4，请勿使用 repo sync-j8 这样的方式同步。
替换已有的 AOSP 源代码的 remote。
如果你之前已经通过某种途径获得了 AOSP 的源码（或者你只是 init 这一步完成后），你希望以后通过 TUNA 同步
AOSP 部分的代码,只需要将.repo/manifest.xml 把其中的 AOSP 这个 remote 的 fetch 从 https://android.
googlesource.com 改为 git://aosp.tuna.tsinghua.edu.cn/android/。

<manifest>
   <remote  name="aosp"
-           fetch="https://android.googlesource.com"
+           fetch="git://aosp.tuna.tsinghua.edu.cn/android/"
            review="android-review.googlesource.com" />
   <remote  name="github"
这个方法也可以用来在同步 Cyanogenmod 代码的时候从 TUNA 同步部分代码
```

下载 repo 后，最好进行一下校验，各版本的校验码如下所示：

```
对于 版本 1.17, SHA-1 checksum 是：ddd79b6d5a7807e911b524cb223bc3544b661c28
对于 版本 1.19, SHA-1 checksum 是：92cbad8c880f697b58ed83e348d06619f8098e6c
对于 版本 1.20, SHA-1 checksum 是：e197cb48ff4ddda4d11f23940d316e323b29671c
对于 版本 1.21, SHA-1 checksum 是：b8bd1804f432ecf1bab730949c82b93b0fc5fede
```

2. Repo 配置

在开始下载源码前，需要对 Repo 进行必要的配置。

如下所示：

```
$ mkdir source #用于存放整个项目源码
$ cd source
$ repo init -u https://android.googlesource.com/platform/manifest
############以下为注释部分#########
init 命令用于初始化 repo 并得到近期的版本更新信息。如果你想获取某个非 master 分支的代码，需要在命令最后加上-b 选项。如：
$ repo init -u https://android.googlesource.com/platform/manifest -b android-4.0.1_r1
完成配置后，repo 会有如下提示：
repo initialized in /home/android
这时在你的机器 home 目录下会有一个.repo 目录，用于记录 manifest 等信息##########
######
```

3. 下载源码

完成初始化动作后，就可以开始下载源码了。根据上一步的配置，下载到的可能是最新版本或者某分支版本的系统源码。

```
$ repo sync
```

由于整个 Android 源码项目非常大，再加上网络等不确定因素，运气好的话可能 1～2 个小时就能品尝到“Android 盛宴”；运气不好的话，估计一个礼拜也未必能完成这一步——如果下载一直失败的话，读者也可以尝试到网上搜索别人已经下载完成的源码包，因为通常在新版本发布后的第一时间就有热心人把它上传到网上了。

可以看到在 Repo 的帮助下，整个下载过程还是相当简单直观的。

提示：如果你在下载过程中出现暂时性的问题（如下载意外中断），可以多试几次。如果一直存在问题，则很可能是代理、网关等原因造成的。更多常见问题的描述与解决方法，可以参见下面这个网址。

```
http://source.android.com/source/known-issues.html
```

典型的 repo 下载界面如图 2-1 所示。

▲图 2-1　原生 Android 工程的典型下载界面

Android 系统本身是由非常多的子项目组成的，这也是为什么我们需要 repo 来统一管理 AOSP 源码的一个重要原因，如图 2-2 所示（部分）。

▲图 2-2 子项目

另外，不同子项目之间的 branches 和 tags 的区别如图 2-3 所示。

▲图 2-3 Android 各子项目的分支和标签

（左：frameworks/base，中：frameworks/native，右：/platform/libcore）

当我们使用 repo init 命令初始化 AOSP 工程时，会在当前目录下生成一个 repo 文件夹，如图 2-4 所示。

▲图 2-4 repo 文件

其中 manifests 本身也是一个 Git 项目，它提供的唯一文件名为 default.xml，用于管理 AOSP 中的所有子项目（每个子项目都由一个 project 标签表示）：

```
<project path="art" name="platform/art" groups="pdk" />
<project path="bionic" name="platform/bionic" groups="pdk" />
<project path="bootable/recovery" name="platform/bootable/recovery" groups="pdk" />
<project path="cts" name="platform/cts" groups="cts,pdk-cw-fs,pdk-fs" />
```

另外，default.xml 中记录了我们在初始化时通过-b 选项指定的分支版本，例如“android-n-preview-2”：

```
<default revision="refs/tags/android-n-preview-2"
         remote="aosp"
         sync-j="4" />
```

这样当执行 repo sync 命令时，系统就可以根据我们的要求去获取正确的源码版本了。

> 友情提示：经常有读者询问阅读 Android 源码可以使用哪些工具。除了著名的 Source Insight 外，另外还有一个名为 SlickEdit 的 IDE 也是相当不错的（支持 Windows、Linux 和 Mac），建议大家可以对比选择最适合自己的工具。

2.2 原生 Android 系统编译指南

任何一个项目在编译前，都首先需要搭建一个完整的编译环境。Android 系统通常是运行于类似 Arm 这样的嵌入式平台上，所以很可能涉及交叉编译。

什么是交叉编译呢？

简单来说，如果目标平台没有办法安装编译器，或者由于资源有限等无法完成正常的编译过程，那就需要另一个平台来辅助生成可执行文件。如很多情况下我们是在 PC 平台上进行 Android 系统的研发工作，这时就需要通过交叉编译器来生成可运行于 Arm 平台上的系统包。需要特别提出的是，“平台”这个概念是指硬件平台和操作系统环境的综合。

交叉编译主要包含以下几个对象。

宿主机（Host）：指的是我们开发和编译代码所在的平台。目前不少公司的开发平台都是基于 X86 架构的 PC，操作系统环境以 Windows 和 Linux 为主。

目标机（Target）：相对于宿主机的就是目标机。这是编译生成的系统包的目标平台。

交叉编译器（Cross Compiler）：本身运行于宿主机上，用于产生目标机可执行文件的编译器。

针对具体的项目需求，可以自行配置不同的交叉编译器。不过我们建议开发者尽可能直接采用国际权威组织推荐的经典交叉编译器。因为它们在 release 之前就已经在多个项目上测试过，可以为接下来的产品开发节约宝贵的时间。表 2-1 所示给出了一些常见的交叉编译器及它们的应用环境。

表 2-1　　常用交叉编译器及应用环境

交叉编译器	宿　主　机	目　标　机
armcc	X86PC(windows)，ADS 开发环境	Arm
arm-elf-gcc	X86PC(windows)，Cygwin 开发环境	Arm
arm-linux-gcc	X86PC(Linux)	Arm

2.2.1 建立编译环境

本书所采用的宿主机是 X86PC（Linux），通过表 2-1 可知在编译过程中需要用到 arm-linux-gcc 交叉编译器（注：Android 系统工程中自带了交叉编译工具，只要在编译时做好相应的配置即可）。

接下来我们分步骤来搭建完整的编译环境，并完成必要的配置。所选取的宿主机操作系统是 Ubuntu 的 14.04 版本 LTS（这也是 Android 官方推荐的）。为了不至于在编译过程中出现各种意想不到的问题，建议大家也采用同样的操作系统环境来执行编译过程。

Step1. 通用工具的安装

表 2-2 给出了所有需要安装的通用工具及它们的下载地址。

表 2-2 通用编译工具的安装及下载地址

通 用 工 具		安装地址、指南
Python 2.X		http://www.python.org/download/
GNU Make 3.81 -- 3.82		http://ftp.gnu.org/gnu/make/
JDK	Java 8 针对 Kitkat 以上版本	最新的 Android 工程已经改用 OpenJDK，并要求为 Java 8 及以上版本。这点大家应该特别注意，否则可能在编译过程中遇到各种问题。具体安装方式见下面的描述
	JDK 6 针对 Gingerbread 到 Kitkat 之间的版本	http://java.sun.com/javase/downloads/
	JDK 5 针对 Cupcake 到 Froyo 之间版本	
Git 1.7 以上版本		http://git-scm.com/download

对于开发人员来说，他们习惯于通过以下方法安装 JDK（如果处于 Ubuntu 系统下）：

Java 6：

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

Java 5：

```
$ sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu hardy main multiverse"
$sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu hardy-updates main
multiverse"
$ sudo apt-get update
$ sudo apt-get install sun-java5-jdk
```

但是随着 Java 的版本变迁及 Sun（已被 Oracle 收购）公司态度的转变，目前获取 Java 的方式也发生了很大变化。基于版权方面的考虑（大家应该已经听说了 Oracle 和 Google 之间的官司恩怨），Android 系统已经将 Java 环境切换到了 OpenJDK，安装步骤如下所示：

```
$ sudo apt-get update
$ sudo apt-get install openjdk-8-jdk
```

首先通过上述命令 install OpenJDK 8，成功后再进行如下配置：

```
$ sudo update-alternatives --config java
$ sudo update-alternatives --config javac
```

如果出现 Java 版本错误的问题，make 系统会有如下提示：

```
************************************************************
You are attempting to build with the incorrect version
of java.

Your version is: WRONG_VERSION.
The correct version is: RIGHT_VERSION.

Please follow the machine setup instructions at
    https://source.android.com/source/download.html
************************************************************
```

Step2. Ubuntu 下特定工具的安装

注意，这一步中描述的安装过程是针对 Ubuntu 而言的。如果你是在其他操作系统下执行的编译，请参阅官方文档进行正确配置；如果你是在虚拟机上运行的 Ubuntu 系统，那么请至少保留 16GB 的 RAM/SWAP 和 100GB 以上的磁盘空间，这是完成编译的基本要求。

- Ubuntu 14.04

```
   $ sudo apt-get install bison g++-multilib git gperf libxml2-utils make zlib1g-
dev:i386 zip
```

- Ubuntu 12.04

所需的命令如下：

```
   $ sudo apt-get install git gnupg flex bison gperf build-essential \
   zip curl libc6-dev libncurses5-dev:i386 x11proto-core-dev \
   libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 \
   libgl1-mesa-dev g++-multilib mingw32 tofrodos \
   python-markdown libxml2-utils xsltproc zlib1g-dev:i386
$ sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-gnu/libGL.so
```

- Ubuntu 10.04 - 11.10

需要安装的程序比较多，不过我们还是可以通过 apt-get 来轻松完成。

具体命令如下：

```
$ sudo apt-get install git-core gnupg flex bison gperf build-essential \
  zip curl zlib1g-dev libc6-dev lib32ncurses5-dev ia32-libs \
  x11proto-core-dev libx11-dev lib32readline5-dev lib32z-dev \
  libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown \
  libxml2-utils xsltproc
```

注意，如果以上命令中存在某些包找不到的情况，可以试试以下命令：

```
$ sudo apt-get install git-core gnupg flex bison gperf libsdl-dev libesd0-dev libwxg
tk2.6-dev build-essential zip curl libncurses5-dev zlib1g-dev openjdk-6-jdk ant gcc-
multilib g++-multilib
```

如果你的操作系统刚好是 Ubuntu 10.10，那么还需要：

```
$ sudo ln -s /usr/lib32/mesa/libGL.so.1 /usr/lib32/mesa/libGL.so
```

如果你的操作系统刚好是 Ubuntu 11.10，那么还需要：

```
$ sudo apt-get install libx11-dev:i386
```

Step3. 设立 ccache（可选）

如果你经常执行“make clean”，或者需要经常编译不同的产品类别，那么 ccache 还是有用的。它可以作为编译时的缓冲，从而加快重新编译的速度。

首先，需要在.bashrc 中加入如下命令。

```
export USE_CCACHE=1
```

如果你的 home 目录是非本地的文件系统（如 NFS），那么需要特别指定（默认情况下它存放于～/.ccache）：

```
export CCACHE_DIR=<path-to-your-cache-directory>
```

在源码下载完成后，必须在源码中找到如下路径并执行命令：

```
prebuilt/linux-x86/ccache/ccache -M 50G
#推荐的值为 50-100GB，你可以根据实际情况进行设置
```

Step4. 配置 USB 访问权限

USB 的访问权限在我们对实际设备进行操作时是必不可少的（如下载系统程序包到设备上）。在 Ubuntu 系统中，这一权限通常需要特别的配置才能获得。

可以通过修改/etc/udev/rules.d/51-android.rules 来达到目的。

例如，在这个文件中加入以下命令内容：

```
# adb protocol on passion (Nexus One)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e12", MODE="0600", OWNER
="<username>"
# fastboot protocol on passion (Nexus One)
SUBSYSTEM=="usb", ATTR{idVendor}=="0bb4", ATTR{idProduct}=="0fff", MODE="0600", OWNER
="<username>"
# adb protocol on crespo/crespo4g (Nexus S)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e22", MODE="0600", OWNER
="<username>"
# fastboot protocol on crespo/crespo4g (Nexus S)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e20", MODE="0600", OWNER
="<username>"
# adb protocol on stingray/wingray (Xoom)
SUBSYSTEM=="usb", ATTR{idVendor}=="22b8", ATTR{idProduct}=="70a9", MODE="0600", OWNER
="<username>"
# fastboot protocol on stingray/wingray (Xoom)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="708c", MODE="0600", OWNER
="<username>"
# adb protocol on maguro/toro (Galaxy Nexus)
SUBSYSTEM=="usb", ATTR{idVendor}=="04e8", ATTR{idProduct}=="6860", MODE="0600", OWNER
="<username>"
# fastboot protocol on maguro/toro (Galaxy Nexus)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e30", MODE="0600", OWNER
="<username>"
# adb protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d101", MODE="0600", OWNER
="<username>"
# fastboot protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d022", MODE="0600", OWNER
="<username>"
# usbboot protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d00f", MODE="0600", OWNER
="<username>"
# usbboot protocol on panda (PandaBoard ES)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d010", MODE="0600", OWNER
="<username>"
```

如果严格按照上述 4 个步骤来执行，并且没有任何错误——那么恭喜你，一个完整的 Android 编译环境已经搭建完成了。

2.2.2 编译流程

上一小节我们建立了完整的编译环境，可谓“万事俱备，只欠东风”，现在就可以执行真正的编译操作了。

下面内容仍然采用分步的形式进行讲解。

Step1. 执行 envsetup 脚本

脚本文件 envsetup.sh 记录着编译过程中所需的各种函数实现，如 lunch、m、mm 等。你可以根据需求进行一定的修改，然后执行以下命令：

```
$ source ./build/envsetup.sh
```

也可以用点号代替 source：

```
$ . ./build/envsetup.sh
```

Step2. 选择编译目标

编译目标由两部分组成，即 BUILD 和 BUILDTYPE。表 2-3 和表 2-4 给出了详细的解释。

表 2-3　　BUILD 参数详解

BUILD	设　备	备　注
Full	模拟器	全编译，即包括所有的语言、应用程序、输入法等
full_maguro	maguro	全编译，并且运行于 Galaxy Nexus GSM/HSPA+ ("maguro")
full_panda	panda	全编译，并且运行于 PandaBoard ("panda")

可见 BUILD 可用于描述不同的目标设备。

表 2-4　　BUILDTYPE 参数详解

BUILDTYPE	备　注
User	编译出的系统有一定的权限限制，通常用来发布最终的上市版本
userdebug	编译出的系统拥有 root 权限，通常用于调试目的
Eng	即 engineering 版本

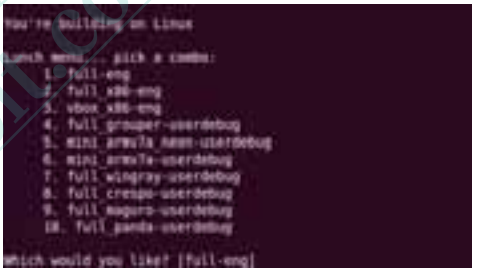
可见 BUILDTYPE 可用于描述各种不同的编译场景。

选择不同的编译目标，可以使用以下命令：

```
$ lunch BUILD-BUILDTYPE
```

如我们执行命令“lunch full-eng”，就相当于编译生成一个用于工程开发目的，且运行于模拟器的系统。

如果不知道有哪些产品类型可选，也可以只敲入“lunch”命令，这时会有一个列表显示出当前工程中已经配置过的所有产品类型（后续小节会讲解如何添加一款新产品）；然后可以根据提示进行选择，如图 2-5 所示。



▲图 2-5　使用“lunch”来显示所有产品

Step3. 执行编译命令

最直接的就是输入如下命令：

```
$ make
```

对于 2.3 以下的版本，整个编译过程在一台普通计算机上需要 3 小时以上的时间。而对于 JellyBean 以上的项目，很可能会花费 5 小时以上的时间（这取决于你的宿主机配置）。

如果希望充分利用 CPU 资源，也可以使用 make 选项“-jN”。N 的值取决于开发机器的 CPU 数、每颗 CPU 的核心数以及每个核心的线程数。

例如，你可以使用以下命令来加快编译速度：

```
$ make -j4
```

有个小技巧可以为这次编译轻松地打上 Build Number 标签，而不需要特别更改脚本文件，即在 make 之前输入如下命令：

```
$ export BUILD_NUMBER=${USER}-'date +%Y%m%d-%H%M%S'
```

在定义 BUILD_NUMBER 变量值时要特别注意容易引起错误的符号，如“$”“&”“:”“/”“\”“<”“>”等。

这样我们就成功编译出 Android 原生态系统了——当然，上面的“make”指令只是选择默认的产品进行编译。假如你希望针对某个特定的产品来执行，还需要先通过上一小节中的“lunch”进行相应的选择。

接下来看看如何编译出 SDK。这是很多开发者，特别是应用程序研发人员所关心的。因为很多时候通过 SDK 所带的模拟器来调试 APK 应用，比在真机上操作要来得高效且便捷；而且模拟器可以配置出各种不同的屏幕参数，用以验证应用程序的“适配”能力。

SDK 是运行于 Host 机之上的，因而编译过程根据宿主操作系统的不同会有所区别。详细步骤如下：

Mac OS 和 Linux

（1）下载源码，和前面已经讲过的源码下载过程没有任何区别。

（2）执行 envsetup.sh。

（3）选择 SDK 对应的产品。

```
$ lunch sdk-eng
```

提示：如果通过“lunch”没有出现“sdk”这个种类的产品也没有关系，可以直接输入上面的命令。

（4）最后，使用以下命令进行 SDK 编译：

```
$ make sdk
```

Windows

运行于 Windows 环境下的 SDK 编译需要基于上面 Linux 的编译结果（注意只能是 Linux 环境下生成的结果，而不支持 MacOS）。

（1）执行 Linux 下 SDK 编译的所有步骤，生成 Linux 版的 SDK。

（2）安装额外的支持包。

```
$ sudo apt-get install mingw32 tofrodos
```

（3）再次执行编译命令，即：

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

这样我们就完成 Windows 版本 SDK 的编译了。

当然上面编译 SDK 的过程也同样可以利用多核心 CPU 的优势。例如：

```
$ make -j4 sdk
```

面向 Host 和 Target 的编译结果都存放在源码工程 out 目录下，分为两个子目录。

- host：SDK 生成的文件存放在这里。例如：
  - MacOS
    out/host/darwin-x86/sdk/android-sdk_eng.<build-id>_mac-x86.zip
  - Windows
    out/host/windows/sdk/android-sdk_eng.${USER}_windows/
- target：通过 make 命令生成的文件存放在这里。

另外，启动一个模拟器可以使用以下命令。

```
$ emulator [OPTIONS]
```

模拟器提供的启动选项非常丰富，读者可以参见本书工具篇中的详细描述。

2.3 定制产品的编译与烧录

上一小节我们学习了原生态 Android 系统的编译步骤，为大家进一步理解定制设备的编译流程打下了基础。Android 系统发展到今天，已经在多个产品领域得到了广泛的应用。相信有一个问题是很多人都想了解的，那就是如何在原生态 Android 系统中添加自己的定制产品。

2.3.1 定制新产品

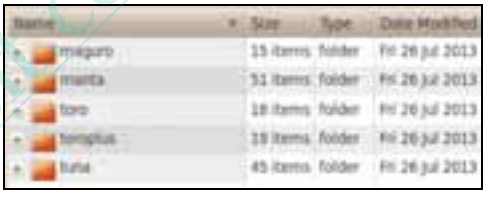
仔细观察整个 Android 源码项目可以发现，它的根目录下有一个 device 文件夹，其中又包含了诸如 samsung、moto、google 等厂商名录，如图 2-6 所示。

在 Android 编译系统中新增一款设备的过程如下。

Step 1. 和图 2-6 所列的各厂商一样，我们也最好先在 device 目录下添加一个以公司命名的文件夹。当然，Android 系统本身并没有强制这样做（后面会看到 vendor 目录也是可以的），只不过规范的做法有利于项目的统一管理。

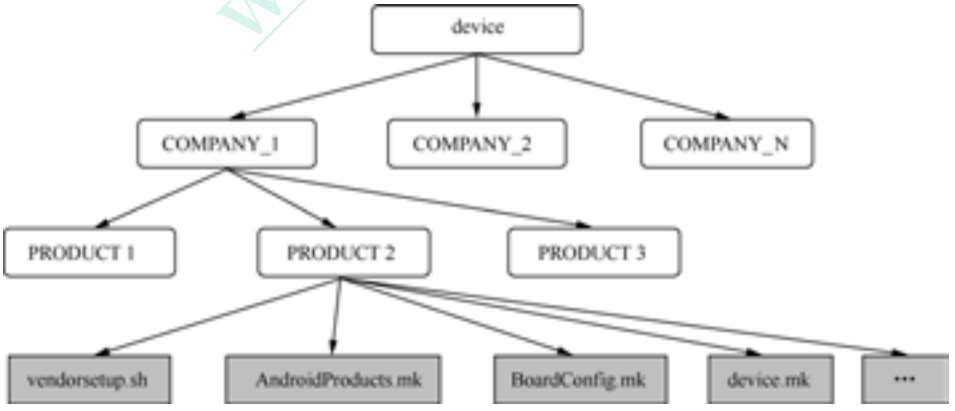
然后在这个公司名目录下为各产品分别建立对应的子文件夹。以 samsung 为例，其文件夹中包含的产品如图 2-7 所示。

▲图 2-6　device 文件夹下的厂商目录



▲图 2-7　一个厂商通常有多种产品

完成产品目录的添加后，和此项目相关的所有特定文件都应该优先放置到这里。一般的组织结构如图 2-8 所示。



▲图 2-8　device 目录的组织架构

由图 2-8 最后一行可以看出，一款新产品的编译需要多个配置文件（sh、mk 等）的支持。我们按照这些文件所处的层级进行一个系统的分类，如表 2-5 所示。

表 2-5 定制新设备所需的配置文件分类

层 级	作 用
芯片架构层（Architecture）	产品所采用的硬件架构，如 ARM、X86 等
核心板层（Board）	硬件电路的核心板层配置
设备层（Device）	外围设备的配置，如有没有键盘
产品层（Product）	最终生成的系统需要包含的软件模块和配置，如是否有摄像头应用程序、默认的国家或地区语言等

也就是说，一款产品由底层往上的构建顺序是：芯片架构→核心板→设备→产品。这样讲可能有点抽象，给大家举个具体的例子。我们知道，当前嵌入式领域市场占有率最高的当属 ARM 系列芯片。但是首先，ARM 公司本身并不生产具体的芯片，而只授权其他合作伙伴来生产和销售半导体芯片。ARM 架构就是属于最底层的硬件体系，需要在编译时配置。其次，很多芯片设计商（如三星）在获得授权后，可以在 ARM 架构的基础上设计出具体的核心板，如 S5PV210。接下来，三星会将其产品进一步销售给有需要的下一级厂商，如某手机生产商。此时就要考虑整个设备的硬件配置了，如这款手机是否要带有按键、触摸屏等。最后，在确认了以上 3 个层次的硬件设计后，我们还可以指定产品的一些具体属性，如默认的国家或地区语言、是否带有某些应用程序等。

后续的步骤中我们将分别讲解与这几个层次相关的一些重要的脚本文件。

Step 2. vendorsetup.sh

虽然我们已经为新产品创建了目录，但 Android 系统并不知道它的存在——所以需要主动告知 Android 系统新增了一个“家庭成员”。以三星 toro 为例，为了让它能被正确添加到编译系统中，首先就要在其目录下新建一个 vendorsetup.sh 脚本。这个脚本通常只需要一个语句。具体范例如下：

```
add_lunch_combo  full_toro-userdebug
```

大家应该还记得前一小节编译原生态系统的第一步是执行 envsetup.sh，函数 add_lunch_combo 就是在这个文件中定义的。此函数的作用是将其参数所描述的产品（如 full_toro-userdebug）添加到系统相关变量中——后续 lunch 提供的选单即基于这些变量产生的。

那么，vendorsetup.sh 在什么时候会被调用呢？

答案也是 envsetup.sh。这个脚本的大部分内容是对各种函数进行定义与实现，末尾则会通过一个 for 循环来扫描工程中所有可用的 vendorsetup.sh，并执行它们。具体源码如下：

```
# Execute the contents of any vendorsetup.sh files we can find.
for f in 'test -d device && find device -maxdepth 4 -name 'vendorsetup.sh' 2> /dev/null'
\
         'test -d vendor && find vendor -maxdepth 4 -name 'vendorsetup.sh' 2> /dev/null'
do
    echo "including $f"
    . $f
Done


unset f
```

可见，默认情况下编译系统会扫描如下路径来查找 vendorsetup.sh：

```
/vendor/
/device/
```

注：vendor 这个目录在 4.3 版本的 Android 工程中已经不存在了，建议开发者将产品目录统一放在 device 中。

打一个比方，上述步骤有点类似于超市的工作流程：工作人员（编译系统）首先要扫描仓库（vendor 和 device 目录），统计出有哪些商品（由 vendorsetup.sh 负责记录），并通过一定的方式（add_lunch_combo@envsetup.sh）将物品上架，然后消费者才能在货架上挑选（lunch）自己想要的商品。

Step 3. 添加 AndroidProducts.mk。消费者在货架上选择（lunch）了某样“商品”后，工作人员的后续操作（如结账、售后等）就完全基于这个特定商品来展开。编译系统会先在商品所在目录下寻找 AndroidProducts.mk 文件，这里记录着针对该款商品的一些具体属性。不过，通常我们只在这个文件中做一个“转向”。如：

```
/*device/samsung/toro/AndroidProducts.mk*/
PRODUCT_MAKEFILES := \
    $(LOCAL_DIR)/aosp_toro.mk \
    $(LOCAL_DIR)/full_toro.mk
```

因为 AndroidProducts.mk 对于每款产品都是通用的，不利于维护管理，所以可另外新增一个或者多个以该产品命名的 makefile（如 full_toro.mk 和 aosp_toro.mk），再让前者通过 PRODUCT_MAKEFILES“指向”它们。

Step4. 实现上一步所提到的某产品专用的 makefile 文件（如 full_toro.mk 和 aosp_toro.mk）。可以充分利用编译系统已有的全局变量或者函数来完成任何需要的功能。例如，指定编译结束后需要复制到设备系统中的各种文件、设置系统属性（系统属性最终会写入设备/system 目录下的 build.prop 文件中）等。以 full_toro.mk 为例：

```
/*device/samsung/toro/full_toro.mk*/
#将 apns 等配置文件复制到设备的指定目录中
PRODUCT_COPY_FILES += \
    device/samsung/toro/bcmdhd.cal:system/etc/wifi/bcmdhd.cal \
    device/sample/etc/apns-conf_verizon.xml:system/etc/apns-conf.xml \
…
# 继承下面两个 mk 文件
$(call inherit-product, $(SRC_TARGET_DIR)/product/aosp_base_telephony.mk)
$(call inherit-product, device/samsung/toro/device_vzw.mk)
# 下面重载编译系统中已经定义的变量
PRODUCT_NAME :=full_toro  #产品名称
PRODUCT_DEVICE := toro #设备名称
PRODUCT_BRAND := Android  #品牌名称
…
```

这部分的变量基本上以“PRODUCT_”开头，我们在表 2-6 中对其中常用的一些变量做统一讲解。

表 2-6　　PRODUCT 相关变量

变　　量	描　　述
PRODUCT_NAME	产品名称，最终会显示在系统设置中的“关于设备”选项卡中
PRODUCT_DEVICE	设备名称
PRODUCT_BRAND	产品所属品牌
PRODUCT_MANUFACTURER	产品生产商
PRODUCT_MODEL	产品型号
PRODUCT_PACKAGES	系统需要预装的一系列程序，如 APKs
PRODUCT_LOCALES	所支持的国家语言。格式如下： [两字节语言码]-[两字节国家码] 如 en_GB de_DE 各语言间以空格分隔
PRODUCT_POLICY	本产品遵循的“策略”，如： android.policy_phone android.policy_mid
PRODUCT_TAGS	一系列以空格分隔的产品标签描述

续表

变　量	描　述
PRODUCT_PROPERTY_OVERRIDES	用于重载系统属性。 格式：key=value 示例：ro.product.firmware=v0.4rc1 dalvik.vm.dexopt-data-only=1 这些属性最终会被存储在系统设备的/system/build.prop 文件中

Step 5. 添加 BoardConfig.mk 文件。这个文件用于填写目标架构、硬件设备属性、编译器的条件标志、分区布局、boot 地址、ramdisk 大小等一系列参数（参见下一小节对系统映像文件的讲解）。下面是一个范例（因为 toro 中的 BoardConfig 主要引用了 tuna 的 BoardConfig 实现，所以我们直接讲解后者的实现）：

```
#/*device/samsung/tuna/BoardConfig.mk*/
TARGET_CPU_ABI := armeabi-v7a ## eabi 即 Embedded application binary interface
TARGET_CPU_ABI2 := armeabi
…
TARGET_NO_BOOTLOADER := true ##不编译 bootloader
…
BOARD_SYSTEMIMAGE_PARTITION_SIZE := 685768704#system.img 分区大小
BOARD_USERDATAIMAGE_PARTITION_SIZE := 14539537408#userdata.img 的分区大小
BOARD_FLASH_BLOCK_SIZE := 4096 #flash 块大小
…
BOARD_WLAN_DEVICE      := bcmdhd #wifi 设备
```

可以看到，这个 makefile 文件中涉及的变量大部分以“TARGET_”和“BOARD_”开头，且数量众多。相信对于第一次编写 BoardConfig.mk 的开发者来说，这是一个不小的挑战。那么，有没有一些小技巧来加速学习呢？

答案是肯定的。

各大厂商在自己产品目录下存放的 BoardConfig.mk 样本就是我们学习的绝佳材料。通过比较可发现，这些文件大部分都是雷同的。所以我们完全可以先从中复制一份（最好选择架构、主芯片与自己项目相当的），然后根据产品的具体需求进行修改。

Step 6. 添加 Android.mk。这是 Android 系统下编译某个模块的标准 makefile。有些读者可能分不清楚这个文件与前面几个步骤中的 makefile 有何区别。我们举例说明，如果 Step1-Step5 中的文件用于决定一个产品的属性，那么 Android.mk 就是生产这个“产品”某个“零件”的“生产工序”。——要特别注意，只是某个“零件”而已。整个产品是需要由很多 Android.mk 生产出的“零件”组合而成的。

Step7. 完成前面 6 个步骤后，我们就成功地将一款新设备定制到编译系统中了。接下来的编译流程和原生态系统是完全一致的，这里不再赘述。

值得一提的是，/system/build.prop 这个文件的生成过程也是由编译系统控制的。具体处理过程在/build/core/Makefile 中，它主要由以下几个部分组成：

- /build/tools/buildinfo.sh

这个脚本用于向 build.prop 中输出各种<key> <value>组合，实现方式也很简单。下面是其中的两行节选：

echo "ro.build.id=$BUILD_ID"

echo "ro.build.display.id=$BUILD_DISPLAY_ID"

- TARGET_DEVICE_DIR 目录下的 system.prop
- ADDITIONAL_BUILD_PROPERTIES
- /build/tools/post_process_props.py

清理工作，将黑名单中的项目从最终的 build.prop 中移除。

开发人员在定制一款新设备时，可以根据实际情况将自己的配置信息添加到上述几个组成部分中，以保证设备的正常运行。

2.3.2　Linux 内核编译

不同产品的硬件配置往往是有差异的。比如某款手机配备了蓝牙芯片，而另一款则没有；即便是都内置了蓝牙模块的两款手机，它们的生产商和型号也很可能不一样——这就不可避免地要涉及内核驱动的移植。前面我们分析的编译流程只针对 Android 系统本身，而 Linux 内核和 Android 的编译是独立的。因此对于设备开发商来说，还需要下载、修改和编译内核版本。

接下来以 Android 官方提供的例子来讲解如何下载合适的内核版本。

这个范例基于 Google 的 Panda 设备，具体步骤如下。

Step1. 首先通过以下命令来获取到 git log：

```
$ git clone https://android.googlesource.com/device/ti/panda
$ cd panda
$ git log --max-count=1 kernel
```

这样就得到了 panda kernel 的提交值，在后续步骤中会用到。

Step2. Google 针对 Android 系统提供了以下可用的内核版本：

```
$ git clone https://android.googlesource.com/kernel/common.git
$ git clone https://android.googlesource.com/kernel/exynos.git
$ git clone https://android.googlesource.com/kernel/goldfish.git
$ git clone https://android.googlesource.com/kernel/msm.git
$ git clone https://android.googlesource.com/kernel/omap.git
$ git clone https://android.googlesource.com/kernel/samsung.git
$ git clone https://android.googlesource.com/kernel/tegra.git
```

上述命令的每一行都代表了一个可用的内核版本。

那么，它们之间有何区别呢？

- exynos，适用于 Samsung Exynos 芯片组；
- goldfish，适用于模拟平台；
- msm，适用于 ADP1，ADP2，Nexus One 以及 Qualcomm MSM 芯片组；
- omap，适用于 PandaBoard 和 Galaxy Nexus 以及 TI OMAP 芯片组；
- samsung，适用于 Nexus S 以及 Samsung Hummingbird 芯片组；
- tegra，适用于 Xoom 以及 NVIDIA Tegra 芯片组；
- common，则是通用版本。

由此可见，与 Panda 设备相匹配的是 omap.git 这个版本的内核。

Step3. 除了 Linux 内核，我们还需要下载 prebuilt。具体命令如下：

```
$ git clone https://android.googlesource.com/platform/prebuilt
$ export PATH=$(pwd)/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin:$PATH
```

Step4. 完成以上步骤后，就可以进行 Panda 内核的编译了：

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=arm-eabi-
$ cd omap
$ git checkout <第一步获取到的值>
$ make panda_defconfig
$ make
```

整个内核的编译相对简单，读者可以自行尝试。

2.3.3 烧录/升级系统

将编译生成的可执行文件包通过各种方式写入硬件设备的过程称为烧录（flash）。烧录的方式有很多，各厂商可以根据实际的需求自行选择。常见的有以下几种。

（1）SD 卡工厂烧录方式

当前市面上的 CPU 主芯片通常会提供多种跳线方式，来支持嵌入式设备从不同的存储介质（如 Flash、SD Card 等）中加载引导程序并启动系统。这样的设计显然会给设备开发商带来更多的便利。研发人员只需要将烧录文件按一定规则先写入 SD 卡，然后将设备配置为 SD 卡启动。一旦设备成功启动后，处于烧写模式下的 BootLoader 就会将各文件按照要求写入产品存储设备（通常是 FLASH 芯片）的指定地址中。

由此可见 Bootloader 的主要作用有两个：其一是提供下载模式，将组成系统的各个 Image 写入到设备的永久存储介质中；其二才是在设备开机过程中完成引导系统正常启动的重任。

一个完整的 Android 烧录包至少需要由 3 部分内容（即 Boot Loader，Linux Kernel 和 Android System）组成。我们可以利用某种方式对它们先进行打包处理，然后统一写入设备中。一般情况下，芯片厂商（如 Samsung）会针对某款或某系列芯片提供专门的烧录工具给开发人员使用；否则各产品开发商需要根据实际情况自行研发合适的工具。

总的来说，SD 卡的烧录手法以其操作简便、不需要 PC 支持等优点被广泛应用于工厂生产中。

（2）USB 方式

这种方式需要在 PC 的配合下完成。设备首先与 PC 通过 USB 进行连接，然后运行于 PC 上的客户端程序将辅助 Android 设备来完成文件烧录。

（3）专用的烧写工具

比如使用 J-Tag 进行系统烧录。

（4）网络连接方式

这种方式比较少见，因为它要求设备本身能接入网络（局域网、互联网），这对于很多嵌入式设备来说过于苛刻。

（5）设备 Bootloader+fastboot 的模式

这也就是我们俗称的“线刷”。需要特别注意的是，能够使用这种升级模式的一个前提是设备中已经存在可用的 Bootloader，因而它不能被运用于工厂烧录中（此时设备中还未有任何有效的系统程序）。

当然，各大厂商通常还会在这种模式上做一些“易用性的封装”（譬如提供带 GUI 界面的工具），从而在一定程度上降低用户的使用门槛。

迫使 Android 设备进入 Bootloader 模式的方法基本上大同小异，下面这两种是最常见的：

通过“fastboot reboot-bootloader”命令来重启设备并进入 Bootloader 模式；

在关机状态下，同时按住设备的“音量减”和电源键进入 Bootloader 模式。

（6）Recovery 模式

和前一种方式类似，Recovery 模式同样不适用于设备首次烧录的场景。“Recovery”的字面意思是“还原”，这也从侧面反映出它的初衷是帮助那些出现异常的系统进行快速修复。由于 OTA 这种得到大规模应用的升级方式同样需要借助于 Recovery 模式，使得后者逐步超出了原先的设计范畴，成为普通消费者执行设备升级操作的首选方式。我们将在后续小节中对此做更详细的讲解。

2.4 Android Multilib Build

早期的 Android 系统只支持 32 位 CPU 架构的编译，但随着越来越多的 64 位硬件平台的出现，这种编译系统的局限性就突显出来了。因而 Android 系统推出了一种新的编译方式，即 Multilib build。可想而知，这种编译系统上的改进需要至少满足两个条件：

- 支持 64-bit 和 32-bit

64 位和 32 位平台在很长一段时间内都需要“和谐共处”，因而编译系统必须保证以下几个场景。

Case1：支持只编译 64-bit 系统。

Case2：支持只编译 32-bit 系统。

Case3：支持编译 64 和 32bit 系统，64 位系统优先。

Case4：支持编译 32 和 64 位系统，32 位系统优先。

- 在现有编译系统基础上不需要做太多改动

事实上 Multilib Build 提供了比较简便的方式来满足以上两个条件，我们将在下面内容中学习到它的具体做法。

（1）平台配置

BoardConfig.mk 用于指定目标平台相关的很多属性，我们可以在这个脚本中同时指定 Primary 和 Secondary 的 CPU Arch 和 ABI：

与 Primary Arch 相关的变量有 TARGET_ARCH、TARGET_ARCH_VARIANT、TARGET_CPU_VARIANT 等，具体范例如下：

```
TARGET_ARCH := arm64
TARGET_ARCH_VARIANT := armv8-a
TARGET_CPU_VARIANT := generic
TARGET_CPU_ABI := arm64-v8a
```

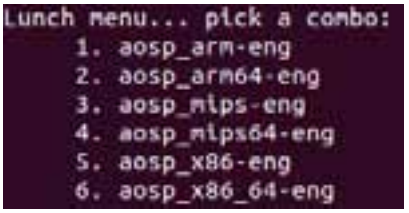
与 Secondary Arch 相关的变量有 TARGET_2ND_ARCH、TARGET_2ND_ARCH_VARIANT、TARGET_2ND_CPU_VARIANT 等，具体范例如下：

```
TARGET_2ND_ARCH := arm
TARGET_2ND_ARCH_VARIANT := armv7-a-neon
TARGET_2ND_CPU_VARIANT := cortex-a15
TARGET_2ND_CPU_ABI := armeabi-v7a
TARGET_2ND_CPU_ABI2 := armeabi
```

如果希望默认编译 32-bit 的可执行程序，可以设置：

```
TARGET_PREFER_32_BIT := true
```

通常 lunch 列表中会针对不同平台提供相应的选项，如图 2-9 所示。



▲图 2-9　相应的选项

当开发者选择不同平台时，会直接影响到 TARGET_2ND_ARCH 等变量的赋值，从而有效控制编译流程。比如图 2-10 中左、右两侧分别对应我们使用 lunch 1 和 lunch 2 所产生的结果，大家可以对比下其中的差异。

▲图 2-10　控制编译流程

另外，还可以设置 TARGET_SUPPORTS_32_BIT_APPS 和 TARGET_SUPPORTS_64_BIT_APPS 来指明需要为应用程序编译什么版本的本地库。此时需要特别注意：

- 如果这两个变量被同时设置，那么系统会编译 64-bit 的应用程序——除非你设置了 TARGET_PREFER_32_BIT 或者在 Android.mk 中对变量做了重载；
- 如果只有一个变量被设置了，那么就只编译与之对应的应用程序；
- 如果两个变量都没有被设置，那么除非你在 Android.mk 中做了变量重载，否则默认只编译 32-bit 应用程序。

那么在支持不同位数的编译时，所采用的 Tool Chain 是否有区别？答案是肯定的。

如果你希望使用通用的 GCC 工具链来同时处理两种 Arch 架构，那么可以使用 TARGET_GCC_VERSION_EXP；反之你可以使用 TARGET_TOOLCHAIN_ROOT 和 2ND_TARGET_TOOLCHAIN_ROOT 来为 64 和 32 位编译分别指定不同的工具链。

（2）单模块配置

我们当然也可以针对单个模块来配置 Multilib。

- 对于可执行程序，编译系统默认情况下只会编译出 64-bit 的版本。除非我们指定了 TARGET_PREFER_32_BIT 或者 LOCAL_32_BIT_ONLY。
- 对于某个模块依赖的库的编译方式，会和该模块有紧密关系。简单来讲 32-bit 的库或者可执行程序依赖的库，会被以 32 位来处理;对于 64 位的情况也同样如此。

需要特别注意的是，在 make 命令中直接指定的目标对象只会产生 64 位的编译。举一个例子来说，“lunch aosp_arm64-eng”→“make libc”只会编译 64-bit 的 libc。如果你想编译 32 位的版本，需要执行“make libc_32”。

描述单模块编译的核心脚本是 Android.mk，在这个文件里我们可以通过指定 LOCAL_MULTILIB 来改变默认规则。各种取值和释义如下所示：

- “first”

只考虑 Primary Arch 的情况

- “both”

同时编译 32 和 64 位版本

- “32”

只编译 32 位版本

- “64”

只编译 64 位版本

- “”

这是默认值。编译系统会根据其他配置来决定需要怎么做，如 LOCAL_MODULE_TARGET_ARCH，LOCAL_32_BIT_ONLY 等。

如果你需要针对某些特定的架构来做些调整，那么以下几个变量可能会帮到你：

- LOCAL_MODULE_TARGET_ARCH

可以指定一个 Arch 列表，例如“arm x86 arm64”等。这个列表用于指定你的模块所支持的 arch 范围，换句话说，如果当前正在编译的 arch 不在列表中将导致本模块不被编译：

- LOCAL_MODULE_UNSUPPORTED_TARGET_ARCH

如其名所示，这个变量起到和上述变量相反的作用。

- LOCAL_MODULE_TARGET_ARCH_WARN
- LOCAL_MODULE_UNSUPPORTED_TARGET_ARCH_WARN

这两个变量的末尾多了个“WARN”，意思就是如果当前模块在编译时被忽略，那么会有 warning 打印出来。

各种编译标志也可以打上与 Arch 相应的标签，如以下几个例子：

- LOCAL_SRC_FILES_arm, LOCAL_SRC_FILES_x86
- LOCAL_CFLAGS_arm, LOCAL_CFLAGS_arm64
- LOCAL_LDFLAGS_arm, LOCAL_LDFLAGS_arm64

我们再来看一下安装路径的设置。对于库文件来说，可以使用 LOCAL_MODULE_RELATIVE_PATH 来指定一个不同于默认路径的值，这样 32 位和 64 位的库都会被放置到这里。对于可执行文件来说，可以分别使用以下两类变量来指定文件名和安装路径：

- LOCAL_MODULE_STEM_32, LOCAL_MODULE_STEM_64

分别指定 32 位和 64 位下的可执行文件名称。

- LOCAL_MODULE_PATH_32, LOCAL_MODULE_PATH_64

分别指定 32 位和 64 位下的可执行文件安装路径。

（3）Zygote

支持 Multilib Build 还需要考虑一个重要的应用场合，即 Zygote。可想而知，Multilib 编译会产生两个版本的 Zygote 来支持不同位数的应用程序，即 Zygote64 和 Zygote32。早期的 Android 系统中，Zygote 的启动脚本被直接书写在 init.rc 中。但从 Lollipop 开始，这种情况一去不复返了。我们来看一下其中的变化：

```
/*system/core/rootdir/init.rc*/
import /init.${ro.hardware}.rc
import /init.${ro.zygote}.rc
```

根据系统属性 ro.zygote 的不同，init 进程会调用不同的 zygote 描述脚本，从而启动不同版本的“孵化器”。以 ro.zygote 为“zygote64_32”为例，具体脚本如下：

```
/*system/core/rootdir/init.zygote64_32.rc*/
service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system
-server --socket-name=zygote
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd

service zygote_secondary /system/bin/app_process32 -Xzygote /system/bin --zygote --
```

```
socket-name=zygote_secondary
    class main
    socket zygote_secondary stream 660 root system
    onrestart restart zygote
```

这个脚本描述的是 Primary Arch 为 64，Secondary Arch 为 32 位时的情况。因为 zygote 的承载进程是 app_process，所以我们可以看到系统同时启动了两个 Service，即 app_process64 和 app_process32。关于 zygote 启动过程中的更多细节，读者可以参考本书的系统启动章节，我们这里先不进行深入分析。

因为系统需要有两个不同版本的 zygote 同时存在，根据前面内容的学习我们可以断定，zygote 的 Android.mk 中一定做了同时编译 32 位和 64 位程序的配置：

```
/*frameworks/base/cmds/app_process/Android.mk*/
LOCAL_SHARED_LIBRARIES := \
     libcutils \
     libutils \
     liblog \
     libbinder \
     libandroid_runtime

LOCAL_MODULE:= app_process
LOCAL_MULTILIB := both
LOCAL_MODULE_STEM_32 := app_process32
LOCAL_MODULE_STEM_64 := app_process64
include $(BUILD_EXECUTABLE)
```

上面这个脚本可以作为需要支持 Multilib build 的模块的一个范例。其中 LOCAL_MULTILIB 告诉系统，需要为 zygote 生成两种类型的应用程序；而 LOCAL_MODULE_STEM_32 和 LOCAL_MODULE_STEM_64 分别用于指定两种情况下的应用程序名称。

2.5 Android 系统映像文件

通过前面几个小节的学习，我们已经按照产品需求编译出自定制的 Android 版本了。编译成功后，会在 out/target/product/[YOUR_PRODUCT_NAME]/目录下生成最终要烧录到设备中的映像文件，包括 system.img，userdata.img，recovery.img，ramdisk.img 等。初次看到这些文件的读者一定想知道为什么会生成这么多的映像、它们各自都将完成什么功能。

这是本小节所要回答的问题。

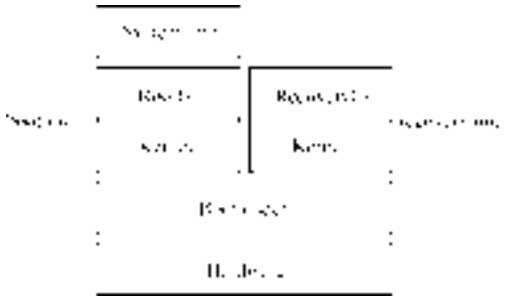
Android 中常见 image 文件包的解释如表 2-7 所示。

表 2-7　　Android 系统常见 image 释义

Image	Description
boot.img	包含内核启动参数、内核等多个元素（详见后面小节的描述）
ramdisk.img	一个小型的文件系统，是 Android 系统启动的关键
system.img	Android 系统的运行程序包（framework 就在这里），将被挂载到设备中的/system 节点下
userdata.img	各程序的数据存储所在，将被挂载到/data 目录下
recovery.img	设备进入“恢复模式”时所需要的映像包
misc.img	即“miscellaneous”，包含各种杂项资源
cache.img	缓冲区，将被挂载到/cache 节点中

它们的关系可以用图 2-11 来表示。

接下来对 boot、ramdisk、system 三个重要的系统 image 进行深入解析。



▲图 2-11　关系图

2.5.1　boot.img

理解 boot.img 的最好方法就是学习它的制作工具——mkbootimg，源码路径在 system/core/mkbootimg 中。这个工具的语法规则如下：

```
mkbootimg  --kernel <filename> --ramdisk <filename>
[ --second <2ndbootloader-filename>]  [ --cmdline <kernel-commandline> ]
 [ --board <boardname> ]  [ --base <address> ]
 [ --pagesize <pagesize> ]  -o|--output <filename>
```

--kernel：指定内核程序包（如 zImage）的存放路径；

--ramdisk：指定 ramdisk.img（下一小节有详细分析）的存放路径；

--second：可选，指第二阶段文件；

--cmdline：可选，内核启动参数；

--board：可选，板名称；

--base：可选，内核启动基地址；

--pagesize：可选，页大小；

--output：输出名称。

那么，编译系统是在什么地方调用 mkbootimg 的呢？

其一就是 droidcore 的依赖中，INSTALLED_BOOTI MAGE_TARGET，如图 2-12 所示。

其二就是生成 INSTALLED_BOOTIMAGE_TARGET 的地方（build/core/Makefile），如图 2-13 所示。

可见 mkbootimg 程序的各参数是由 INTERNAL_BOOTIMAGE_ARGS 和 BOARD_MKBOOTIMG_ARGS 来指定的，而这两者又分别取决于其他 makefile 中的定义。如 BoardConfig.mk 中定义的 BOARD_KERNEL_CMDLINE 在默认情况下会作为--cmdline 参数传给 mkbootimg；BOARD_KERNEL_BASE 则作为--base 参数传给 mkbootimg。

▲图 2-12　droidcore 的依赖

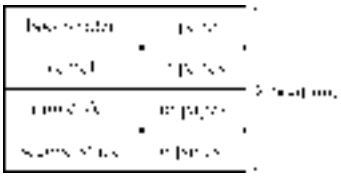
▲图 2-13　生成 INSTALLED_BOOTIMAGE_TARGET 的地方

按照 Bootimg.h 中的描述，boot.img 的文件结构如图 2-14 所示。

各组成部分如下：

1. boot header

存储内核启动“头部”——内核启动参数等信息，占据一个 page 空间，即 4KB 大小。Header 中包含的具体内容可以通过分析 Mkbootimg.c 中的 main 函数来获知，它实际上对应 boot_img_hdr 这个结构体：

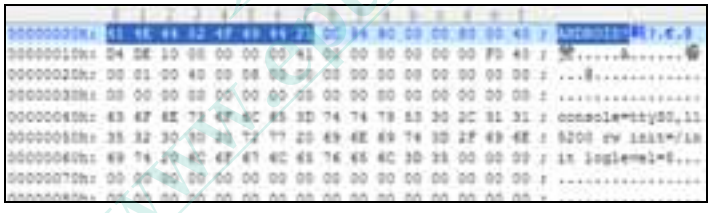


▲图 2-14 boot.img 的文件结构

```
/*system/core/mkbootimg/Bootimg.h*/
struct boot_img_hdr
{
    unsigned char magic[BOOT_MAGIC_SIZE];
    unsigned kernel_size;  /* size in bytes */
    unsigned kernel_addr;  /* physical load addr */
    unsigned ramdisk_size; /* size in bytes */
    unsigned ramdisk_addr; /* physical load addr */
    unsigned second_size;  /* size in bytes */
    unsigned second_addr;  /* physical load addr */
    unsigned tags_addr;    /* physical addr for kernel tags */
    unsigned page_size;    /* flash page size we assume */
    unsigned unused[2];    /* future expansion: should be 0 */
    unsigned char name[BOOT_NAME_SIZE]; /* asciiz product name */
    unsigned char cmdline[BOOT_ARGS_SIZE];
    unsigned id[8]; /* timestamp / checksum / sha1 / etc */
};
```

这样讲有点抽象，下面举个实际的 boot.img 例子，我们可以用 UltraEditor 或者 WinHex 把它打开，如图 2-15 所示。

可以看到，文件最起始的 8 个字节是“ANDROID!”，也称为 BOOT_MAGIC；后续的内容则包括 kernel_size，kernel_addr 等，与上述的 boot_img_hdr 结构体完全吻合。



▲图 2-15 boot header 实例

2. kernel

内核程序是整个 Android 系统的基础，也被“装入”boot.img 中——我们可以通过--kernel 选项来指定内核映射文件的存储路径。其所占据的大小为：

```
n pages=(kernel_size + page_size - 1) / page_size
```

由此可以看出，boot.img 中的各元素必须是页对齐的。

3. ramdisk

不仅是 kernel，boot.img 中也包含了 ramdisk.img。其所占据大小为：

```
m pages=(ramdisk_size + page_size - 1) / page_size
```

可见也是页对齐的。

其他关于 ramdisk 的详细描述请参照下一小节，这里先不做解释。

4. second stage

这一项是可选的。其占据大小为：

```
o pages= (second_size + page_size - 1) / page_size
```

这个元素通常用于扩展功能，默认情况下可以忽略。

2.5.2　ramdisk.img

无论什么类型的文件，从计算机存储的角度来说都只不过是一堆“0”“1”数字的集合——它们只有在特定处理规则的解释下才能表现出意义。如 txt 文本用 Ultra Editor 打开就可以显示出里面的文字；jpg 图像文件在 Photoshop 工具的辅助下可以让用户看到其所包含的内容。而文本与 jpeg 图像文件本质上并没有区别，只不过存储与读取这一文件的“规则”发生了变化——正是这些“五花八门”的“规则”才创造出成千上万的文件类型。

另外，文件后缀名也并不是必需的，除非操作系统用它来鉴别文件的类型。而更多情况下，后缀名的存在只是为了让用户有个直观的认识。如我们会认为“*.txt”是文本文档、“*.jpg”是图片等。

Android 的系统文件以“.img”为后缀名，这种类型的文件最初用来表示某个 disk 的完整复制。在从原理的层面讲解这些系统映像之前，可以通过一种方式来让读者对这些文件有个初步的感性认识（下面的操作以 ramdisk.img 为例，其他映像文件也是类似的）。

首先对 ramdisk.img 执行 file 命令，得到如下结果：

```
$file ramdisk.img
ramdisk.img: gzip compressed data, from Unix
```

这说明它是一个 gZip 的压缩文件。我们将其改名为 ramdisk.img.gz，再进行解压。具体命令如下：

```
$gzip –d ramdisk.img.gz
```

这时会得到另一个名为 ramdisk.img 的文件，不过文件类型变了：

```
$file ramdisk.img
ramdisk.img: ASCII cpio archive (SVR4 with no CRC)
```

由此可知，这时的 ramdisk.img 是 CPIO 文件了。

再来执行以下操作：

```
$cpio -i -F ramdisk.img
3544 blocks
```

这样就解压出了各种文件和文件夹，范例如图 2-16 所示。

▲图 2–16　范例

可以清楚地看到，常用的 system 目录、data 目录以及 init 程序（系统启动过程中运行的第一个程序）等文件都包含在 ramdisk.img 中。

这样我们可以得出一个大致的结论，ramdisk.img 中存放的是 root 根目录的镜像（编译后可以在 out/target/product/[YOUR_PRODUCT_NAME]/root 目录下找到）。它将在 Android 系统的启动过程中发挥重要作用。

2.5.3 system.img

要将 system.img 像 ramdisk.img 一样解压出来会相对麻烦一些。不过方法比较多，除了以下提到的方式，读者还可以尝试使用 unyaffs（参考 http://code.google.com/p/unyaffs/或者 http://code.google.com/p/yaffs2utils/）来实现。

这里我们采取 mount 的方法，这是目前最省时省力的解决方式。

步骤如下：

- simg2img

编译成功后，这个工具的可执行文件在 out/host/linux-x86/bin 中。

源码目录 system/extras/ext4_utils。

将此工具复制到与 system.img 同一目录下。

执行如下命令可以查询 simg2img 的用法：

```
$ ./simg2img --h
Usage: simg2img <sparse_image_file><raw_image_file>
```

对 system.img 执行：

```
$ ./simg2img system.img system.img.step1
```

- mount

将上一步得到的文件通过以下操作挂载到 system_extracted 中：

```
$ mkdir system_extracted
$ sudo mount -o loop system.img.step1 system_extracted
```

最终我们得到如图 2-17 所示的结果。

▲图 2-17 结果图

这说明该 image 文件包含了设备/system 节点中的相关内容。

2.5.4 Verified Boot

Android 领域的开放性催生了很多第三方 ROM 的繁荣（例如市面上“五花八门”的 Recovery、定制的 Boot Image、System Image 等），同时也给系统本身的安全性带来了挑战。

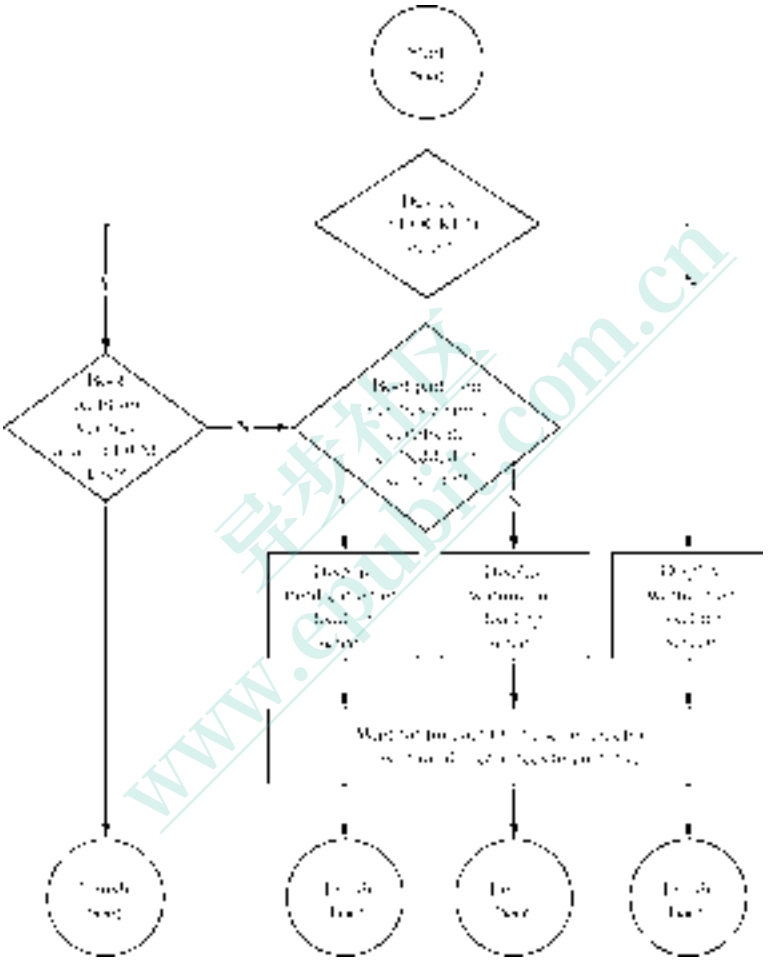
从 4.4 版本开始，Android 结合 Kernel 的 dm-verity 驱动能力实现了一个名为“Verified Boot”的安全特性，以期更好地保护系统本身免受恶意程序的侵害。我们在本小节将向大家讲解这一特性的基本原理，以便读者们在无法成功利用 fastboot 写入 image 时可以清楚地知道隐藏在背后的真正原因。

我们先来熟悉表 2-8 所示的术语。

当设备开机以后，根据 Boot State 和 Device State 的状态值不同，有如图 2-18 所示几种可能性。

表 2-8　　Verified Boot 相关术语

术　　语	释　　义
dm-verity	Linux kernel 的一个驱动，用于在运行时态验证文件系统分区的完整性（判断依据是 Hash Tree 和 Signed metadata）
Boot State	保护等级，分为 GREEN、YELLOW、ORANGE 和 RED 四种
Device State	表明设备接受软件刷写的程度，通常有 LOCKED 和 UNLOCKED 两种状态
Keystore	公钥合集
OEM key	Bootloader 用于验证 boot image 的 key

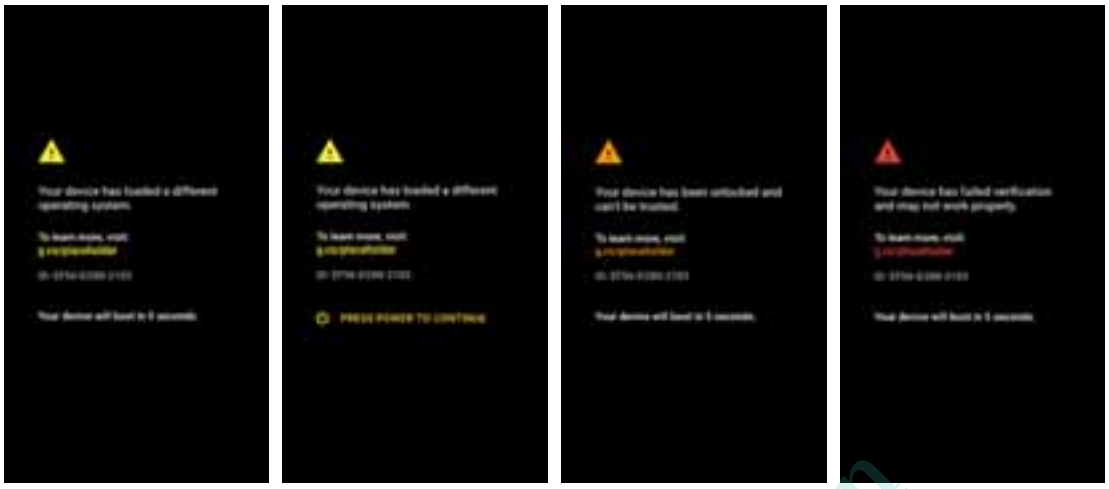


▲图 2-18　Verified Boot 总体流程

（引用自 Android 官方文档）

最下方的 4 个圆圈颜色分别为：GREEN、YELLOW、RED 和 ORANGE。例如当前设备的 Device State 是 LOCKED，那么就首先需要经历 OEM KEY Verification——如果通过的话 Boot State 是 GREEN，表示系统是安全的；否则需要进入下一轮的 Signature Verification，其结果决定了 Boot State 是 YELLOW 或者是 RED（比较危险）。当然，如果当前设备本身就是 UNLOCKED 的，那就不用经过任何检验——不过它和 YELLOW、RED 一样的地方是，都会在屏幕上显式地告诫用户潜在的各种风险。部分 Android 设备还会要求用户主动做出选择后才能正常启动，如图 2-19 所示典型示例。

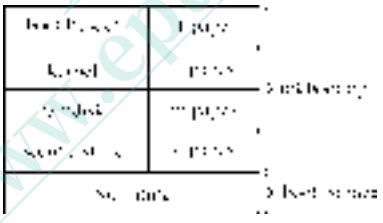
如果设备的 Device State 发生切换的话（fastboot 就提供了类似的命令，只不过大部分设备都需要解锁码才能完成），那么系统中的 data 分区将会被擦除，以保证用户数据的安全。



▲图 2-19　典型示例

我们知道，Android 系统在启动过程中要经过 Bootloader->Kernel->Android 三个阶段，因而在 Verified Boot 的设计中，它对分区的看护也是环环相扣的。具体来说，Bootloader 承担 boot 和 recovery 分区的完整性校验职责；而 Boot Partition 则需要保证后续的分区，如 system 的安全性。另外，Recovery 的工作和 Boot 是基本类似的。

不过，由于分区文件大小有差异，具体的检验手段也是不同的。结合前面小节对 boot.img 的描述，其在增加了 verified boot 后的文件结构变化如图 2-20 所示。



▲图 2-20　文件结构变化

除了 mkbootimg 来生成原始的 boot.img 外，编译系统还会调用另一个新工具，即 boot_signer（对应源码目录 system/extras/verity）来在 boot.img 的尾部附加一个 signature 段。这个签名是针对 boot.img 的 Hash 结果展开的，默认使用的 key 在/build/target/product/security 目录下。

而对于某些大块分区（如 System Image），则需要通过 dm-verity 来验证它们的完整性。关于 dm-verity 还有非常多的技术细节，限于篇幅我们不做过多讨论，但强烈建议读者自行查阅相关资料做进一步深入学习。

2.6 ODEX 流程

ODEX 是 Android 旧系统的一个优化机制。对于很多开发人员来说，ODEX 可以说是既熟悉又陌生。熟悉的原因在于目前很多手机系统，或者 APK 中的文件都从以前的格式变成了如图 2-21 和图 2-22 所示的样子。

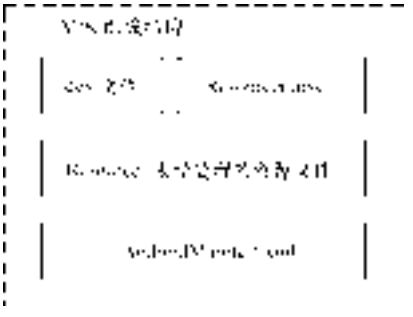
而陌生的原因在于有关 ODEX 的资料并不是很多，不少开发人员对于 ODEX 是什么，能做什么以及它的应用流程并不清楚——这也是我们本小节所要向大家阐述的内容。

▲图 2-21　系统目录 system/framework 下的文件列表

ODEX 是 Optimized Dalvik Executable 的缩写，从字面意思上理解，就是经过优化的 Dalvik 可执行文件。Dalvik 是 Android 系统（目前已经切换到 Art 虚拟机）中采用的一种虚拟机，因而经过优化的 ODEX 文件让我们很自然地想到可以为虚拟机的运行带来好处。

事实上也的确如此——ODEX 是 Google 为了提高 Android 运行效率做出努力的成果之一。我们知道，Android 系统中不少代码是使用 Java 语言编写的。编译系统首先会将一个 Java 文件编译成 class 的形式，进而再通过一个名为 dx 的工具来转换成 dex 文件，最后将 dex 和资源等文件压缩成 zip 格式的 APK 文件。换句话说，一个典型的 Android APK 的组成结构如图 2-23 所示。

▲图 2-22　系统目录/system/app 下的文件列表



▲图 2-23　APK 的组成结构

本书的 Android 应用程序编译和打包章节将做更为详细介绍。现在大家只要知道 APK 中有哪些组成元素就可以了。当应用程序启动时，系统需要提取图 2-23 中的 dex（如果之前没有做过 ODEX 优化的话，或者/data/dalvik-cache 中没有对应的 ODEX 缓存），然后才能执行加载动作。而 ODEX 则是预先将 DEX 提取出来，并针对当前具体设备做了优化工作后的产物，这样做除了能提高加载速度外，还有如下几个优势：

- 加大了破解程序的难度

ODEX 是在 dex 基础上针对当前具体设备所做的优化，因而它和生成时所处的具体设备有很大关联。换句话说，除非破解者能提供与 ODEX 生成时相匹配的环境文件（比如 core.jar、ext.jar、framework.jar、services.jar 等），否则很难完成破解工作。这就在无形中提高了系统的安全性。

- 节省了存储空间

按照 Android 系统以前的做法，不仅 APK 中需要存放一个 dex 文件，而且/data/dalvik-cache 目录下也会有一个 dex 文件，这样显然会浪费一定的存储空间。相比之下，ODEX 只有一份，而且它比 dex 所占的体积更小，因而自然可以为系统节省更多的存储空间。

2.7 OTA 系统升级

前面我们讨论了系统包烧录的几种传统方法，而 Android 系统其实还提供了另一种全新的升级方案，即 OTA（Over the Air）。OTA 非常灵活，它既可以实现完整的版本升级，也可以做到增量升级。另外，用户既可以选择通过 SD 卡来做本地升级，也可以直接采用网络在线升级。

不论是哪种升级形式，都可以总结为 3 个阶段：

- 生成升级包；
- 获取升级包；
- 执行升级过程。

下面我们来逐一分析这 3 个阶段。

2.7.1 生成升级包

升级包也是由系统编译生成的，其编译过程本质上和普通 Android 系统编译并没有太大区别。如果想生成完整的升级包，具体命令如下：

```
$make otapackage
```

> **注意** 生成 OTA 包的前提是，我们已经成功编译生成了系统映像文件（system.img 等）。

最终将生成以下文件：

```
out/target/product/[YOUR_PRODUCT_NAME]/[YOUR_PRODUCT_NAME]-ota-eng.[UID].zip
```

而生成差分包的过程相对麻烦一些，不过方法也很多。以下给出一种常用的方式：

➢ 将上一次生成的完整升级包复制并更名到某个目录下，如～/OTA_DIFF/old_target_file.zip；

➢ 对源文件进行修改后，用 make otapackage 编译出一个新的 OTA 版本；

➢ 将本次生成的 OTA 包更名后复制到和上一个升级包相同的目录下，如～/OTA_DIFF/new_target_file.zip；

➢ 调用 ota_from_target_files 脚本来生成最终的差分包。

这个脚本位于：

```
build/tools/releasetools/ota_from_target_files
```

值得一提的是，完整升级包的生成过程其实也使用了这一脚本。区分的关键就在于使用时是否提供了-i 参数。

其具体语法格式是：

```
ota_from_target_files [Flags] input_target_files output_ota_package
```

所有 Flags 参数释义如表 2-9 所示。

表 2-9　　ota_from_target_files 参数

参　　数	说　　明
-b (--board_config)　<file>	在新版本中已经无效
-k (--package_key) <key>	<key>用于包的签名 默认使用 input_target-files 中的 META/misc_info.txt 文件 如果此文件不存在，则使用 build/target/product/security/testkey
-i (--incremental_from)　<file>	该选项用于生成差分包
-w (--wipe_user_data)	由此生成的 OTA 包在安装时会自动擦除 user data 分区
-n (--no_prereq)	忽略时间戳检查
-e (--extra_script)　<file>	将<file>内容插入 update 脚本的尾部
-a (--aslr_mode)　<on|off>	是否开启 ASLR 技术 默认为开

在这个例子中，我们可以采用以下命令生成一个 OTA 差分包：

```
./build/tools/releasetools/ota_from_target_files-i  ~ /OTA_DIFF/old_target_file.zip ~ /OTA_DIFF/new_target_file.zip
```

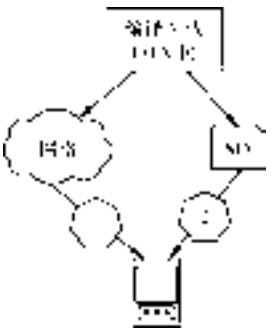
这样生成的 update.zip 就是最终可用的差分升级包。一方面，差分升级包体积较小，传输方便；但另一方面，它对升级的设备有严格要求，即必须是安装了上一升级包版本的那些设备才能正常使用本次的 OTA 差分包。

2.7.2　获取升级包

如图 2-24 所示，有两种常见的渠道可以获取到 OTA 升级包，分别是在线升级和本地升级。

1. 在线升级

开发者将编译生成的 OTA 包上传至网络存储服务器上，然后用户可以直接通过终端访问和下载升级文件。通常我们把下载到的 OTA 包存储在设备的 SD 卡中。



▲图 2-24　获取 OTA 升级包的两种方式

在线升级的方式涉及两个核心因素。

- 服务器端的架构

设备厂商需要架构服务器来存放、管理 OTA 包，并为客户端提供包括查询在内的多项服务。

- 客户端与服务器的交互方式

客户终端如何与服务器进行交互，是否需要认证，OTA 包如何传输等都是需要考虑的。

由此可见，在线升级方式要求厂商提供较好的硬件环境来解决用户大规模升级时可能引发的问题，因而成本较高。不过这种方式对消费者来说比较方便，而且可以实时掌握版本的最新动态，所以对凝聚客户有很大帮助。目前很多主流设备生产商（如 HTC）和第三方的 ROM 开发商（如 MIUI）都提供了在线升级模式。

服务器和客户端的一种理论交互方案可以参见图 2-25 所示的图例。

步骤如下：

➢　在手动升级的情况下，由用户发出升级的指令；而在自动升级的情况下，则由程序根据一定的预设条件来启动升级流程。比如设定了开机自动检查是否有可用的更新，那么每次机器启动后都会去服务器取得最新的版本信息。

➢ 无论是手动还是自动升级，都必须通过服务器查询信息。与服务器的连接方式是多种多样的，由开发人员自行决定。在必要的情况下，还应该使用加密连接。

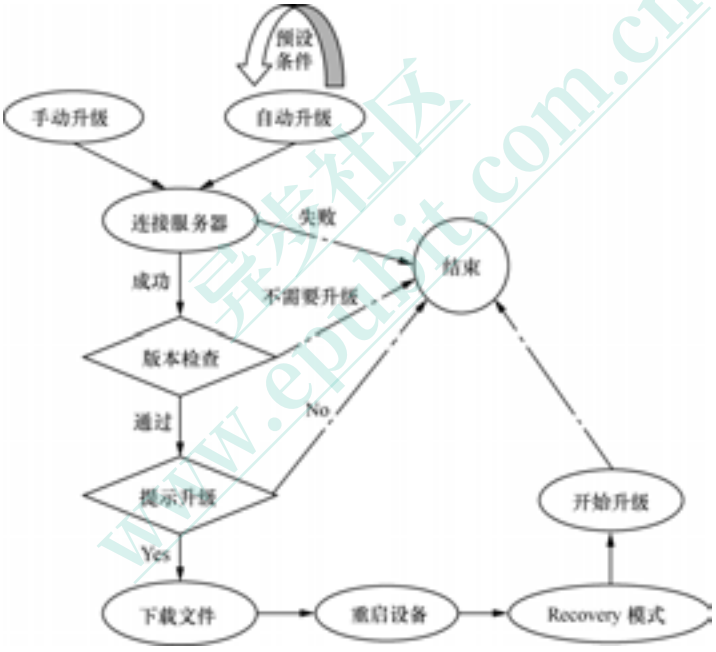
➢ 如果一切顺利，我们就得到了服务器上最新升级文件的版本号。接下来需要将这个版本号与本地安装的系统版本号进行比较，决定是否进入下一步操作。

➢ 如果服务器上的升级文件要比本地系统新（在制定版本号规则时，应尽量考虑如何可以保证新旧版本的快速比较），那么升级继续；否则中止升级流程——且若是手动升级的情况，一定要提示用户中止的原因，避免造成不好的用户体验。

➢ 升级文件一般都比较大（Android 系统文件可能达到几百 MB）。这么大的数据量，如果是通过移动通信网络（GSM\WCDMA\CDMA\TD-SCDMA 等）来下载，往往不现实。因此如果没有事先知会用户而自动下载的话，很可能会引起用户的不满。“提示框”的设计也要尽可能便利，如可以让用户快捷地启用 Wi-Fi 通道进行下载。

➢ 下载后的升级文件需要存储在本地设备中才能进入下一步的升级。通常这一文件会直接被放置在 SD 卡的根目录下，命名为 update.zip。

➢ 接下来系统将自动重启，并进入 RecoveryMode 进行升级。



▲图 2-25 在线升级图例

2. 本地升级

OTA 升级包并非一定要通过网络在线的方式才可以下载到——只要条件允许，就可以从其他渠道获取到升级文件 update.zip，并复制到 SD 卡的根目录下，然后手动进入升级模式（见下一小节）。

在线升级和本地升级各有利弊，开发商应根据实际情况来提供最佳的升级方式。

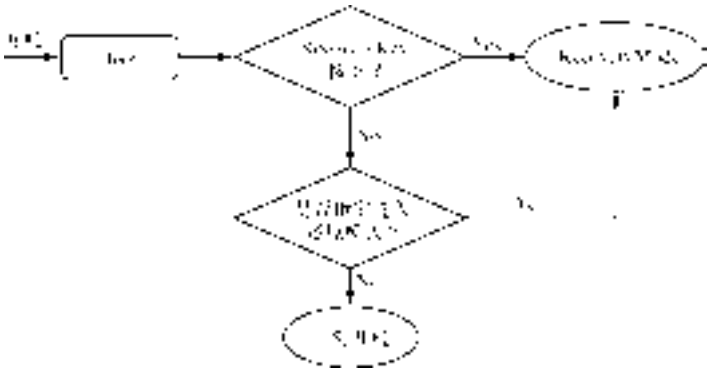
2.7.3 OTA 升级——Recovery 模式

经过前面小节的讲解，现在我们已经准备好系统升级文件了（不论是在线还是本地升级），接下来就进入 OTA 升级最关键的阶段——Recovery 模式，也就是大家俗称的“卡刷”。

Recovery 相关的源码主要在工程项目的如下目录中：

\bootable\recovery

因为涉及的模块比较多，这个文件夹显得有点杂乱。我们只挑选与 Recovery 刷机有关联的部分来进行重点分析。



▲图 2-26　进入 RecoveryMode 的流程

图 2-26 所示是 Android 系统进入 RecoveryMode 的判断流程，可见在如下两种情况下设备会进入还原模式。

- 开机过程中检测到 RecoveryKey 按下

很多 Android 设备的 RecoveryKey 都是电源和 Volume+的组合键，因为这两个按键在大部分设备上都是存在的。

- 系统指定进入 RecoveryMode

系统在某些情况下会主动要求进入还原模式，如我们前面讨论的“在线升级”方式——当 OTA 包下载完成后，系统需要重启然后进入 RecoveryMode 进行文件的刷写。

当进入 RecoveryMode 后，设备会运行一个名为“Recovery”的程序。这个程序对应的主要源码文件是/bootable/recovery/ recovery.cpp，并且通过如下几个文件与 Android 主系统进行沟通。

（1）/cache/recovery/command　　INPUT

Android 系统发送给 recovery 的命令行文件，具体命令格式见后面的表格。

（2）/cache/recovery/log　　OUTPUT

recovery 程序输出的 log 文件。

（3）/cache/recovery/intent　　OUTPUT

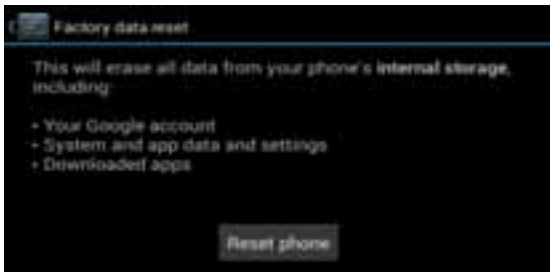
recovery 传递给 Android 的 intent。

当 Android 系统希望开机进入还原模式时，它会在/cache/recovery/command 中描述需要由 Recovery 程序完成的“任务”。后续 Recovery 程序通过解析这个文件就可以知道系统的“意图”，如表 2-10 所示。

表 2-10　　CommandLine 参数释义

Command Line	Description
--send_intent=anystring	将 text 输出到 recovery.intent 中
--update_package=path	安装 OTA 包
--wipe_data	擦除 user data，然后重启
--wipe_cache	擦除 cache（不包括 user data），然后重启
--set_encrypted_filesystem=on|off	enable/disable 加密文件系统
--just_exit	直接退出，然后重启

由表格所示的参数可以知道 Recovery 不但负责 OTA 的升级，而且也是“恢复出厂设置”的实际执行者，如图 2-27 所示。



▲图 2-27 系统设置中的“恢复出厂设置”

接下来分别讲解这两个功能在 Recovery 程序中的处理流程。

恢复出厂设置。

（1）用户在系统设置中选择了“恢复出厂设置”。

（2）Android 系统在/cache/recovery/command 中写入“--wipe_data”。

（3）设备重启后发现了 command 命令，于是进入 recovery。

（4）recovery 将在 BCB（bootloader control block）中写入“boot-recovery”和“--wipe_data”，具体是在 get_args()函数中——这样即便设备此时重启，也会再进入 erase 流程。

（5）通过 erase_volume 来重新格式化/data。

（6）通过 erase_volume 来重新格式化/cache。

（7）finish_recovery 将擦除 BCB，这样设备重启后就能进入正常的开机流程了。

（8）main 函数调用 reboot 来重启。

上述过程中的 BCB 是专门用于 recovery 和 bootloader 间互相通信的一个 flash 块，包含了如下信息：

```
struct bootloader_message {
    char command[32];
    char status[32];
    char recovery[1024];
};
```

依据前面对 Android 系统几大分区的讲解，BCB 数据应该存放在哪个 image 中呢？没错，是 misc。

OTA 升级具体如下。

（1）OTA 包的下载过程参见前一小节的介绍。假设包名是 update.zip，存储在 SDCard 中。

（2）系统在/cache/recovery/command 中写入"--update_package=[路径名]"。

（3）系统重启后检测到 command 命令，因而进入 recovery。

（4）get_args 将在 BCB 中写入"boot-recovery" 和 "--update_package=..." ——这样即便此时设备重启，也会尝试重新安装 OTA 升级包。

（5）install_package 开始安装 OTA 升级包。

（6）finish_recovery 擦除 BCB，这样设备重启后就可以进入正常的开机流程了。

（7）如果 install 失败的话：

- prompt_and_wait 显示错误，并等待用户响应；
- 用户重启（比如拔掉电池等方式）。

（8）main 调用 maybe_install_firmware_update，OTA 包中还可能包含 radio/hboot firmware 的更新，具体过程略。

（9）main 调用 reboot 重启系统。

总体来说，整个 Recovery.cpp 源文件的逻辑层次比较清晰，读者可以基于上述流程的描述来对照并阅读代码。

2.8 Android 反编译

目前我们已经学习了 Android 原生态系统及定制产品的编译和烧录过程。和编译相对的，却同样重要的是反编译。比如，一个优秀的“用毒”高手往往也会是卓越的“解毒”大师，反之亦然。大自然的一个奇妙之处即万事万物都是“相生相克”的，只有在竞争中才能不断地进步和发展。

首先要纠正不少读者可能会持有的观点——“反编译”就是去“破解”软件。应该说，破解一款软件的确需要用到很多反编译的知识，不过这并不是它的全部用途。比如笔者就曾经在开发过程中利用反编译辅助解决了一个 bug，在这里和读者分享一下。

问题是这样的：开发人员 A 修改了 framework 中的某个文件，然后通过正常的编译过程生成了 image，再将其烧录到了机器上。但奇怪的是，文件的修改并没有体现出来（连新加的 log 也没有打印出来）。显然，出现问题的可能是下列步骤中的任何一个，如图 2-28 所示。

▲图 2-28　可能出现问题的几个步骤

可疑点为：

- 程序没有执行到打印 log 的地方

因为加 log 的那个函数是系统会频繁调用到的，而且 log 就放在函数开头没有加任何判断，所以这个可能性被排除。

- log 被屏蔽

打印 log 所用的方法与此文件中其他地方所用的方法完全一致，而且其他地方的 log 确实成功输出了，所以也排除这一可能性。

- 修改的文件没有被编译到

虽然 Android 的编译系统非常强大，但是难免会有 bug，因而这个可能性还是存在的。那么如何确定我们修改的文件真的被编译到了呢？此时反编译就有了用武之地了。

- 文件确实被编译到，但是烧录时出现了问题

这并不是空穴来风，确实发生过开发人员因为粗心大意烧错版本的“事故”（对于某些细微修改，编译系统不会主动产生新的版本号）。通过反编译机器上的程序，然后和原始文件进行比较，我们可以清楚地确认机器中运行的程序是不是预期的版本。

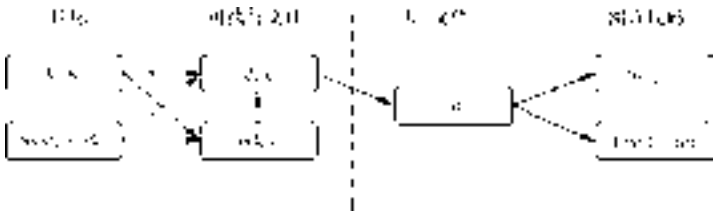
由上述分析可知，反编译是确定该问题最直接的方式。

Android 反编译过程按照目标的不同分为如下两类（都是基于 Java 语言的情况）。

- APK 应用程序包的反编译。
- Android 系统包（如本例中的 framework）的反编译。

不论针对哪种目标对象，它们的步骤都可以归纳为如图 2-29 所示。

APK 应用安装包实际上是一个 Zip 压缩包，使用 Zip 或 WinRAR 等软件打开后里面有一个“classes.dex”文件——这是 Dalvik JVM 虚拟机支持的可执行文件（Dalvik Executable）。关于这个文件的生成过程，可以参见本书应用篇中对 APK 编译过程的介绍。换句话说，classes.dex 这个文件包含了所有的可执行代码。



▲图 2-29 反编译的一般流程

由前面小节的学习我们知道，odex 是 classes.dex 经过 dex 优化（optimize）后产生的。一方面，Dalvik 虚拟机会根据运行需求对程序进行合理优化，并缓存结果；另一方面，因为可以直接访问到程序的 odex，而不是通过解压缩包去获取数据，所以无形中加快了系统开机及程序的运行速度。

针对反编译过程，我们首先是要取得程序的 dex 或者 odex 文件。如果是 APK 应用程序，只需要使用 Zip 工具解压缩出其中的 classes.dex 即可（有的 APK 原始的 classes.dex 会被删除，只保留对应的 odex 文件）；而如果是包含在系统 image 中的系统包（如 framework 就是在 system.img 中），就需要通过其他方法间接地将其原始文件还原出来。具体步骤可以参见前一小节的介绍。

取得 dex/odex 文件后，我们将它转化成 Jar 文件。

- odex

目前已经有不少研究项目在分析 Android 的逆向工程，其中最著名的就是 smali/baksmali。可以在这里下载到它的最新版本：

```
http://code.google.com/p/smali/downloads/list
```

“smali”和“baksmali”分别对应冰岛语中“assembler”和“disassembler”。为什么要用冰岛语命名呢？答案就是 Dalvik 这个名字实际上是冰岛的一个小渔村。

如果是 odex，需要先用 baksmali 将其转换成 dex。具体语法如下：

```
$ baksmali -a <api_level> -x <odex_file> -d <framework_dir>
```

-a 指定了 API Level，-x 表示目标 odex 文件，-d 指明了 framework 路径。因为这个工具需要用到诸如 core.jar，ext.jar，framework.jar 等一系列 framework 包，所以建议读者直接在 Android 源码工程中 out 目录下的 system/framework 中进行操作，或者把所需文件统一复制到同一个目录下。

范例如下（1.4.1 版本）：

```
$ java -jar  baksmali-1.4.1.jar  -a 16  -x  example.odex
```

如果是要反编译系统包中的 odex（如 services.odex），请参考以下命令：

```
$java -Xmx512m -jar baksmali-1.4.1.jar -a 16 -c:core.jar:bouncycastle.jar:ext.jar:framework.
jar:android.policy.jar:services.jar:core-junit.jar -d framework/ -x services.odex
```

更多语法规则可以通过以下命令获取：

```
$ java -jar  baksmali-1.4.1.jar  --help
```

执行结果会被保存在一个 out 目录中，里面包含了与 odex 相应的所有源码，只不过由 smali 语法描述。读者如果有兴趣的话，可以阅读以下文档来了解 smali 语法：

```
http://code.google.com/p/smali/wiki/TypesMethodsAndFields
```

当然对于大部分开发人员来说，还是希望能反编译出最原始的 Java 语言文件。此时就要再将 smali 文件转化成 dex 文件。具体命令如下：

```
$ java -jar  smali-1.4.1.jar out/ -o  services.dex
```

于是接下来的流程就是 dex→Java，请参考下面的说明。

- dex

前面我们已经成功将 odex“去优化”成 dex 了，离胜利还有一步之遥——将 dex 转化成 jar 文件。目前比较流行的工具是 dex2jar，可以在这里下载到它的最新版本：

http://code.google.com/p/dex2jar/downloads/list

使用方法也很简单，具体范例如下：

```
$ ./dex2jar.sh  services.dex
```

上面的命令将生成 services_dex2jar.jar，这个 Jar 包中包含的就是我们想要的原始 Java 文件。那么，选择什么工具来阅读 Jar 中的内容呢？在本例中，我们只是希望确定所加的 log 是否被正确编译进目标文件中，因而可以使用任何常用的文本编辑器查阅代码。而如果希望能更方便地阅读代码，推荐使用 jd-gui，它是一款图形化的反编译代码阅读工具。

这样，整个反编译过程就完成了。

顺便提一下，目前，几乎所有的 Android 程序在编译时都使用了“代码混淆”技术，反编译后的结果和原始代码还是有一定差距，但不影响我们理解程序的主体架构。“代码混淆”可以有效地保护知识产权，防止某些不法分子恶意剽窃，或者篡改源码（如添加广告代码、植入木马等），建议大家在实际的项目开发中尽量采用。

2.9 NDK Build

我们知道 Android 系统下的应用程序主要是由 Java 语言开发的，但这并不代表它不支持其他语言，比如 C++和 C。事实上，不同类型的应用程序对编程语言的诉求是有区别的——普通 Application 的 UI 界面基本上是静态的，所以，利用 Java 开发更有优势；而游戏程序，以及其他需要基于 OpenGL（或基于各种 Game Engine）来绘制动态界面的应用程序则更适合采用 C 或者 C++语言。

伴随着 Android 系统的不断发展，开发者对于 C/C++语言的需求越来越多，也使得 Google 需要不断完善它所提供的 NDK 工具链。NDK 的全称是 Native Development Kit，可以有效支撑 Android 系统中使用 C/C++等 Native 语言进行开发，从而让开发者可以：

- 提高程序运行效率

完成同样的功能，Java 虚拟机理论上来说比 C/C++要耗费更多的系统资源。因而，如果程序本身对运行性能要求很高的话，建议利用 NDK 进行开发。

- 复用已有的 C 和 C++库

好处是显而易见，即最大程度地避免重复性开发。

NDK 的官方网址是：

```
https://developer.android.com/ndk/index.html
```

它的安装很简单，在 Windows 下只要下载一个几百 MB 的自解压包然后双击打开它就可以了。NDK 文件夹可以被放置到磁盘中的任何位置，不过为了操作方便，建议开发者可以设置系统环境变量来指向其中的关键程序。NDK 既支持 Java 和 C/C++混合编程的模式，也允许我们只开发纯 Native 实现的程序。前者需要用到 JNI 技术(即 Java Native Interface)，它的神奇之处在于可以让两种看似没有瓜葛的语言间进行无缝的调用。例如下面是一个 JNI 的实例：

```
public class MyActivity extends Activity {
  /**
   * Native method implemented in C/C++
```

```
    */
    public native void jniMethodExample();
}
```

MyActivity 是一个 Java 类，它的内部包含一个声明为 Native 的成员变量，即 jniMethodExample。这个函数的实现是通过 C/C++完成的，并被编译成 so 库来供程序加载使用。更多 JNI 的细节，我们将在后续章节进行详细介绍。

本小节我们将通过一个具体实例来着重讲解如何利用 NDK 来为应用程序执行 C/C++的编译。

在此之前，请确保你已经下载并解压了 NDK 包，并为它设置了正确的系统环境变量。这个例子中将包含如下几个文件，我们统一放在一个 JNI 文件夹中：

- Android.mk；
- Application.mk；
- testNative.cpp。

Android.mk 用于描述一个 Android 的模块，包括应用程序、动态库、静态库等。它和我们本章节讲解的用法基本一致，因而不再赘述。

Application.mk 用于描述你的程序中所用到的各个 Native 模块（可以是静态或者动态库，或者可执行程序）。这个脚本中常用的变量不多，我们从中挑选几个核心的来讲解：

1. APP_PROJECT_PATH

指向程序的根目录。当然，如果你是按照 Android 系统默认的结构来组织工程文件的话，这个变量是可选的。

2. APP_OPTIM

用于指示当前是 release 或者 debug 版本。前者是默认的值，将会生成优化程度较高的二进制文件;调试模式则会生成未优化的版本，以便保留更多的信息来帮助开发者追踪问题。在 AndroidManifest.xml 的<application>标签中声明 android:debuggable 会将默认值变更为 debug，不过 APP_OPTIM 的优先级更高，可以重载 debuggable 的设置。

3. APP_CFLAGS

设置对全体 module 有效的 C/C++编译标志。

4. APP_LDFLAGS

用于描述一系列链接器标志，不过只对动态链接库和可执行程序有效。如果是静态链接库的情况，系统将忽略这个值。

5. APP_ABI

用于指示编译所针对的目标 Application Binary Interface，默认值是 armeabi。可选值如表 2-11 所示。

表 2-11 可选值

指 令 集	ABI 值
Hardware FPU instructions on ARMv7 based devices	APP_ABI := armeabi-v7a
ARMv8 AArch64	APP_ABI := arm64-v8a
IA-32	APP_ABI := x86
Intel64	APP_ABI := x86_64

续表

指　令　集	ABI 值
MIPS32	APP_ABI := mips
MIPS64 (r6)	APP_ABI := mips64
All supported instruction sets	APP_ABI := all

文件 testNative.cpp 中的内容就是程序的源码实现，对此 NDK 官方提供了较为完整的 Samples 供大家参考，涵盖了 OpenGL、Audio、Std 等多个方面，有兴趣的读者可以自行下载分析。

那么有了这些文件后，如何利用 NDK 把它们编译成最终产物呢？

最简单的方式就是采用如下的命令：

```
cd <project>
$ <ndk>/ndk-build
```

其中 ndk-build 是一个脚本，等价于：

```
$GNUMAKE -f <ndk>/build/core/build-local.mk
<parameters>
```

<ndk>指的是 NDK 的安装路径。

可见使用 NDK 来编译还是相当简单的。另外，除了常规的编译外，ndk-build 还支持多种选项，譬如：

“clean”表示清理掉之前编译所产生的各种中间文件；

“-B”会强制发起一次完整的编译流程；

“NDK_LOG=1”用于打开 NDK 的内部 log 消息；

……

2.10 第三方 ROM 的移植

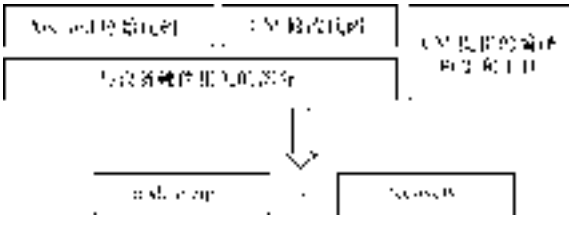
除了本章所描述的 Android 原生代码外，开发人员也可以选择一些知名的第三方开源 ROM 来进行学习，譬如 CyanogenMod。

CyanogenMod（简称 CM）的官方网址如下：

http://www.cyanogenmod.org/

它目前的最新版本是基于 Android 6.0 的 CM 13，并同时支持 Google Nexus、HTC、Huawei、LG 等多个品牌的众多设备。CyanogenMod 的初衷是将 Android 系统移植到更多的没有得到 Google 官方支持的设备中，所以有的时候 CM 针对某特定设备的版本更新时间可能比设备厂商来得还要早。

那么 CyanogenMod 是如何做到针对多种设备的移植和适配工作的呢?我们将在接下来的内容中为大家揭开这个问题的答案。图 2-30 是 CM 的整体描述图。



▲图 2-30　CM 的整体描述

下面我们分步骤进行讲解。

Step1. 前期准备

在做 Porting 之前，有一些准备工作需要我们去完成。

（1）获取设备的 Product Name、Code Name、Platform Architecture、Memory Size、Internal Storage Size 等信息

这些数据有很多可以从/system/build.prop 文件中获得，不过前提条件是手机需要被 root。

（2）收集设备对应的内核源码

根据 GPL 开源协议的规定，Android 厂商必须公布受 GPL 协议保护的内容，包括内核源码。因而实现这一步是可行的，只是可能会费些周折。

（3）获取设备的分区信息

Step2. 建立 3 个核心文件夹

分别是：

- device/[vendor]/[codename]/

设备特有的配置和代码将保存在这个路径下。

- vendor/[vendor]/[codename]/

这个文件夹中的内容是从原始设备中拉取出来的，由此可见主要是那些没有源代码可以生成的部分，例如一些二进制文件。

- kernel/[vendor]/[codename]/

专门用于保存内核版本源码的地方。

CM 提供了一个名为 mkvendor.sh 的脚本来帮助创建上述文件夹的“雏形”，有兴趣的读者可以参见 build/tools/device/mkvendor.sh 文件。不过很多情况下还需要开发者手工修改其中的部分文件，例如 device 目录下的 BoardConfig.mk、device_[codename].mk、cm.mk、recovery.fstab 等核心文件。

Step3. 编译一个用于测试的 recovery image

编译过程和普通 CM 编译的最大区别在于选择 make recoveryimage。如果在 recovery 模式下发现 Android 设备的硬件按键无法使用，那么可以尝试修改/device/[vendor]/[codename]/recovery/recovery_ui.cpp 中的 GPIO 配置。

Step4. 为上述的 device 目录建立 github 仓库，以便其他人可以访问到。

Step5. 填充 vendor 目录

可以参考 CM 官网上成熟的设备范例提供的 extract-files.sh 和 setup-makefiles.sh，并据此完成适合自己的这两个脚本。

Step6. 通过 CM 提供的编译命令最终编译出 ROM 升级包，并利用前面生成的 recovery 来将其刷入到设备中。这个过程很可能不是“一蹴而就”的，需要不断调试和修改，直至成功。

当然，限于篇幅我们在本小节只是讲解了 CM 升级包的核心制作过程，读者如果有兴趣的话可以查阅 http://www.cyanogenmod.org/来获取更多细节详情。

第 3 章　Android 编译系统

3.1 Makefile 入门

如果读者曾在 Linux 环境下开发过程序，那么对 Makefile 一定不会陌生。简单而言，Makefile 提供了一种机制，让使用者可以有效地组织“工作”。注意这里只使用“工作”这个词，而不是“编译”。这是因为 Makefile 并不一定是用来完成编译工作。事实上它本身只是一种“规则”的执行者，而使用者具体通过它来做什么则没有任何限制。如既可以用它来架构编译系统，也能用来生成文档，或者单纯地打印 log 等。

从中可以看出，理解 Makefile 的“规则”才是我们学习的重点。只要学会了“渔”，自然就什么“鱼”都不缺了。

和 shell、python 等类似，Makefile 也是一个脚本，由 make 程序来解析。目前软件行业有多款优秀的 make 解析程序，如 GNU make（Android 中采用的）、Visual Studio 中的 nmake 等。尽管在表现形式上会有些差异，但“万变不离其宗”，它们都是通过以下基础规则扩展起来的：

```
TARGET: PREREQUISITES
    COMMANDS
```

注意　每个 COMMAND 前都必须有一个 TAB 制表符。

这个看起来简单得不能再简单的规则，在经过一次次的扩展修饰后，便构建成最终我们看到的各种庞大工程的编译系统。在 Makefile 规则中，TARGET 是需要生成的目标文件，PREREQUISITES 代表了目标所依赖的所有文件。当 PREREQUISITES 文件中有任何一个比 TARGET 新时，那么都会触发下面 COMMAND 命令的执行。COMMAND 的具体内容取决于使用者的需求，如调用 GCC 编译器、生成某个文档等。

下面我们通过一个简单的例子（Linux 环境下，采用 GCC 编译器来生成一个可运行程序）来讲解这个规则的使用方法。

范例项目包括的主要工程文件和说明如表 3-1 所示。

表 3-1　工程文件和说明

文 件 名	描　　述
main.c	主函数所在文件
utility.h	提供了一个测试函数（getNumber）的声明
utility.c	提供了函数 getNumber 的实现。getNumber 函数只是简单地返回一个值，没有其他特别的功能
Makefile	用于编译整个工程

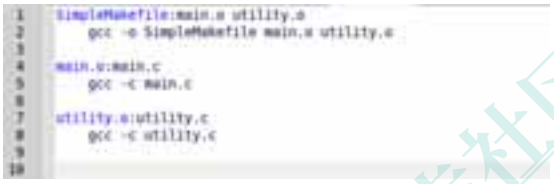
我们先来看一下各工程文件中主要的源码节选。

（1）main.c 中将打印一条语句，输出 getNumber()函数的返回值。

（2）utility.h 中，仅仅是对 getNumber 进行了函数声明。

（3）utility.c 中，给出了 getNumber 的函数实现。

（4）Makefile 文件，是编译过程的“规划者”。



根据前面提到的“基础规则”，这个脚本共有 3 个 TARGET，即 SimpleMakefile、main.o 及 utility.o。其中 SimpleMakefile 依赖于后两者，而 main.o 和 utility.o 则又分别由对应的 C 文件编译生成。

最后我们通过调用 make 命令来生成 SimpleMakefile 可执行文件。运行结果如下：

这是一个非常简单的 Makefile 示例，但“麻雀虽小，五脏俱全”，它向我们清晰地展示了一个 Makefile 的编写过程。Make 工具本身是非常强大的，有很多隐含规则来帮助开发者快速构建一个复杂的编译体系。如利用它的自动推导功能，可以简化对象间的依赖关系；还可以加入各种变量以避免多次重复输入。如下面就是 SimpleMakefile 程序的另一个 Makefile 版本，和前一个相比显然更简洁明了。

另外，我们还需要不断熟悉 Android 编译系统中常见的一些 Makefile 的高级用法。例如下面这两种。

（1）Target-Specific Variable

这是一种局部作用域的变量，它只对特定的目标起效果。譬如下面这个例子中的 PRIVATE_DEX_FILE 变量：

```
/*build/core/package_internal.mk*/
ifdef LOCAL_DEX_PREOPT
$(built_odex): PRIVATE_DEX_FILE := $(built_dex)
# Use pattern rule - we may have multiple built odex files.
$(built_odex) : $(dir $(LOCAL_BUILT_MODULE))% : $(built_dex)
     $(hide) mkdir -p $(dir $@) && rm -f $@
```

```
        $(add-dex-to-package)
        $(hide) mv $@ $@.input
        $(call dexpreopt-one-file,$@.input,$@)
        $(hide) rm $@.input
    endif
```

我们知道，Make 会首先构建出依赖树，然后才会根据用户选择目标来执行相应的编译操作。换句话说，当 Make 执行到上述脚本语句时，并不会触发 COMMANDS 部分的执行。那么如果我们直接在 COMMANDS（在这个场景中，add-dex-to-package 会调用到 PRIVATE_DEX_FILE）中使用$(built_dex)这种全局变量会发生什么事呢？

因为 package_internal.mk 是 APK 编译模板的内部实现（可以参考后续小节的介绍），所以它在整个系统编译过程中会被多次调用。换句话说，$(built_dex)的值是不断在变化中的。而当原先的$(built_odex)在执行自己的 COMMANDS 时，$(built_dex)的值早已经“物是人非”了，所以，在 COMMANDS 中直接调用它有可能导致不可预期的错误。例如 PRIVATE_DEX_FILE 就是属于$(built_odex)规则中的特定私有变量（这也是变量名中“PRIVATE”所表达的含义）。因而，无论外界环境如何变化，PRIVATE_DEX_FILE 的值在定义它的目标规则中都是不变的，这样一来就有效保证了编译执行阶段的准确性。

（2）Static Pattern Rules

另一个 Android 编译脚本中常见的 Makefile 语法是 Static Pattern Rules。它的典型格式中带有两个“:”，如下所示：

```
TARGETS ...: TARGET-PATTERN: DEP-PATTERNS ...
    COMMANDS
```

静态模式语法经常被应用于多目标的场景。如下所示就是一个简单的范例：

```
foo.o tcc.o bar.o : %.o : %.c
```

在这个例子中，TARGET-PATTERN 带有一个“%”，它与 TARGETS 集合中的元素进行匹配，便可以得到词干（STEM）：foo tcc bar;得到的词干再和 DEP-PATTERNS 中的“%”进行组合，从而为不同目标生成正确的依赖对象（foo.c tcc.c bar.c）。

除此之外，Makefile 实际上还有很多高级用法和规则，希望读者能自行查阅相关资料做进一步了解，以便为后续更好地理解 Android 编译系统打下坚实的基础。

3.2 Android 编译系统

很多人在初次分析 Android 的编译系统时，都会有一种感觉——这是一头让人“胆寒的怪兽”。因为它正好符合怪兽的两个特点，即功力深厚且体积庞大，甚至可以说有点臃肿。这在一定程度上也加重了我们分析它内部结构的难度。就如同漫步在崇山峻岭或深山野林一般，如果缺乏很明确的指引，那么稍有不慎便会迷失方向。

事实上 Android 编译系统也是经过了数次不断地迭代演进，才成长为今天大家所看到的“怪兽”。特别是在 Google 收购 Android 系统不久后的 2006 年，他们曾对编译系统动过一次“大手术”。核心目标有两个：

- 让依赖关系分析更为可靠，以保证可以正确判断出需要被编译的模块；
- 让不必要被编译的模块也可以被准确判断出来，以提高编译效率。

在那次重构中，Android 遵循了多个设计原则和策略，包括但不限于：

（1）同一套代码支持编译出不同的构建目标。例如既可以编译出运行于设备端的软件包，也可以编译出 Host 平台上的各种工具（模拟器、辅助工具等）。

（2）Non-Recursive Make: 1997 年的时候有一篇非常有名的论文，名为《Recursive Make Considered Harmful》，其核心思想是我们在大型项目中应该采用唯一的 Makefile 来组织所有文件的自动化编译，并对比了它和传统的多 Makefile 的优势。相信这篇论文对当年重构过程起到了很大的影响，可以说直到今天，Android 编译系统的主体框架还是采用 Non-Recursive 的方式。有兴趣的读者可以阅读一下这篇论文。

（3）可以对项目中的任意模块进行单独的编译验证。比如我们只更改了 Art 虚拟机所对应的 libart.so，那么就应该能做到只以这个 so 库为中心来开展编译工作，而不需要每次都是整个项目编译。

（4）编译所产生的中间过程文件，以及最终的编译结果和源代码需要在存储目录上分离等。

在本节内容的组织上，我们将采取由上而下、由整体到局部的方式将 Android 编译系统所涉及的各个重要方面一一理顺。希望通过这种分析手法，让读者可以逐步摸清整个编译机制的内部架构，而不致牵绊于各种宏、函数、变量、目录结构等细枝末节。另外，希望读者在阅读接下来内容的过程中还可以同步思考一下编译系统的几个设计原则具体是如何实现的，以期达到触类旁通的效果。

3.2.1 Makefile 依赖树的概念

细心的读者应该已经注意到了，前一小节的 SimpleMakefile 中各 TARGET 的依赖关系实际上可以组成一棵树，本书将其称为“Makefile 依赖树”。

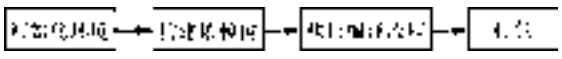
我们仍以上一小节的工程项目为例，给出它的依赖树，如图 3-1 所示。

▲图 3-1 SimpleMakefile 的依赖树

这种树型结构为我们按照由上而下的顺序分析 Android 编译系统提供了可能，接下来几个小节的内容就是由此展开的。

3.2.2 Android 编译系统抽象模型

我们对 Android 编译系统进行抽象，其顶层模型如图 3-2 所示。



▲图 3-2 Android 编译系统的抽象模型

因为是基于 Makefile 实现的，所以，整个编译系统的核心仍然是如何有效地构建出依赖树；Android 系统的编译过程涉及 Java、C/C++等多种语言，而且还分为 Host 和 Target 等不同的目标平台，因而它的运行环境相对复杂，需要我们在初始化环节做好环境的搭建工作（例如当前的 JDK 和 Make 的版本是否满足要求）；编译的执行过程本质上和传统的 Make 实现没有太大差异，只不过因为 Android 工程非常庞大，所以初学者往往很难在短时间内全部掌握；编译系统的另一个重要任务则是打包，包括 system.img、boot.img、userdata.img 等，我们在后续小节会有更深入的分析。

3.2.3　树根节点 droid

树根节点一定是整个编译系统的最终目标产物吗？

答案既可能是肯定的，也可能是否定的。

肯定的一面，是因为有些树根节点（比如 SimpleMakefile）确实是整个工程项目的“目标产物”——它是真实存在的“生成物”；而更多情况下，特别是对 Android 系统这种庞然大物而言，它的树根节点往往只是一个“伪目标”——它的确是代表了编译系统的“终极目标意愿”，本身却不是一个真正的“TARGET”。

我们先从 Android 源码工程的根目录分析起，其下的 Makefile 文件是其编译系统的起点。可以看到，它只是一个简单的文件转向，直接引用了另一个 Makefile 文件（build/core/main.mk）。具体如下所示：

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

这个被引用的 main.mk 文件有上千行代码，对于 Makefile 文档来说是比较大的。另外，它又进一步引用了很多其他脚本文件，使得整个文件的内部结构看起来杂乱无章。即便有不少注释，仍让很多初学者“望而却步”。

基于这个原因，如果一开始我们就照着 Makefile 文件一行行解释，很可能会“事倍功半”。建议大家牢记 Makefile 依赖树的概念，以树根节点作为切入点寻求突围。当然，Android 系统的依赖树可不像 SimpleMakefile 例子那样一目了然。如何找出它的根节点，并以此为基础逐步构建出一棵完整的依赖树还是需要花点工夫的。这要求我们对 Make 的工作原理有一定程度的理解。所需知识点总结如下：

（1）需要强调的是，编译系统中往往有不止一棵依赖树存在。比如我们在 Android 系统下使用“make”命令和“make sdk”的编译结果会迥然不同，这是因为它们分别执行了两棵不同的依赖树。

在没有显式指定编译目标的情况下（使用不带任何参数的“make”命令来执行编译），那么第一个符合要求的目标会被 Make 作为默认的依赖树根节点。这条规则也同样提醒我们在书写 Makefile 时一定要注意各 Target 的排放顺序。如果不慎将 clean 等目标放在最开始的位置，很可能会导致异常情况。

（2）原则上 Make 程序会对 makefile 中的内容按照顺序进行逐条解析。一个典型的解析过程分为三大步骤，即：

- 变量赋值、环境检测等初始化操作；
- 按照规则生成所有依赖树；
- 根据用户选择的依赖树，从叶到根逐步生成目标文件。

掌握了上面 Makefile 的基本规则后，对照 main.mk 就可以很快找出，“make”命令对应的依赖树的根节点是 droid。它是这样定义的：

```
/*build/core/Main.mk*/
# This is the default target.  It must be the first declared target.
.PHONY: droid
DEFAULT_GOAL := droid
$(DEFAULT_GOAL):
```

从注释中也可以佐证我们的猜测，即 droid 就是“default target”，而且“It must be the first declared target”——只不过这里的 droid 还是一个空的“规则”，相当于是我们预先占了个位置以保证它是第一个出现的目标。

那么，droid 的真正“规则”是在哪里定义的呢？

仔细观察的话，会发现 main.mk 后续内容还有多个地方对 droid 进行了定义。而且根据 TARGET_BUILD_APPS 值的不同，出现了两个分支。如下源码所示：

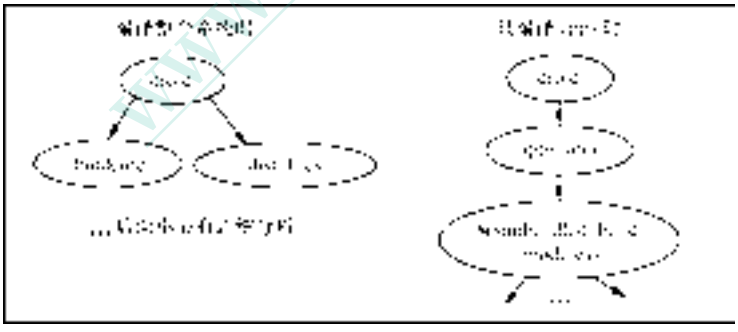
```
ifneq ($(TARGET_BUILD_APPS),) ##只编译 APP，而不是整个系统
  …
.PHONY: apps_only
apps_only: $(unbundled_build_modules)
droid: apps_only ##这种情况下 droid 只依赖于 apps_only

else #非 TARGET_BUILD_APPS 的情况，此时需要编译整个系统
  $(call dist-for-goals, droidcore, \
    $(INTERNAL_UPDATE_PACKAGE_TARGET) \
    $(INTERNAL_OTA_PACKAGE_TARGET) \
    $(SYMBOLS_ZIP) \
    $(INSTALLED_FILES_FILE) \
    $(INSTALLED_BUILD_PROP_TARGET) \
    $(BUILT_TARGET_FILES_PACKAGE) \
    $(INSTALLED_ANDROID_INFO_TXT_TARGET) \
    $(INSTALLED_RAMDISK_TARGET) \
    $(INSTALLED_FACTORY_RAMDISK_TARGET) \
$(INSTALLED_FACTORY_BUNDLE_TARGET) \
)
  …
droid: droidcore  dist_files##当编译整个系统时，droid 的依赖关系
endif # TARGET_BUILD_APPS
```

从上面这段代码可以看出，我们既可以编译整个系统，也可以选择只编译 App。第二种情况使用的场景比较少——此时 droid 只依赖于 apps_only，而后者又进一步取决于$(unbundled_build_modules)变量，有兴趣的读者可以自行分析。

编译整个系统是我们的首选，即上述代码中“else”部分的内容。很显然，这时候 droid 有两个 PREREQUISITES: droidcore 和 dist_files，如图 3-3 所示。

我们将在后续几个小节具体分析 droidcore 和 dist-files。



▲图 3-3 droid 的依赖树

3.2.4 main.mk 解析

讲解 droidcore 和 dist_files 之前，我们先完整分析下 main.mk 脚本文件的架构。除了构建 droid 等依赖树外，main.mk 有一大半的内容是为了完成以下几点。

➢ 对编译环境的检测

比如 Java 版本是否符合要求，当前机器上的 make 环境必须高于特定版本等。如果这些检查没有通过，一般情况下系统会终止编译。

➢ 进行一些必要的前期处理

比如说整个项目工程是否需要先进行清理工作，部分工具的安装等。

➢ 引用其他 Makefile 文件

这点在整个 main.mk 中处处可见，如引用 config.mk，cleanbuild.mk 等。

➢ 设置全局变量

这些全局变量决定了编译的具体实现过程。

➢ 各种函数的实现

Android 编译系统中定义了不少函数，它们提供了各种问题的统一解决方案。比如 print-vars 函数用于打印一串变量列表，my-dir 可以知道当前所处的路径等。这些函数对我们自己编写 Makefile 文件也有一定的指导意义。

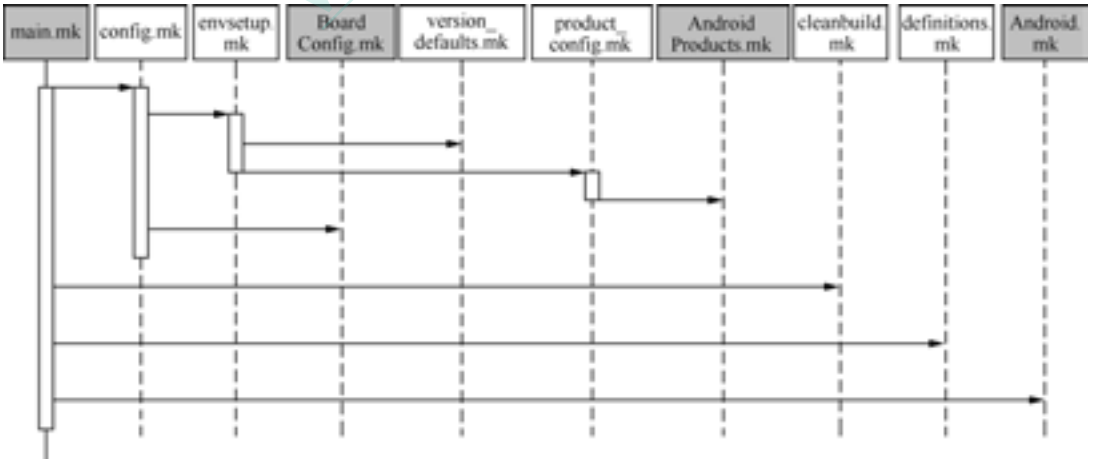
根据前一章节内容的讲解，为 Android 系统添加一款定制设备需要涉及如下几个脚本文件。

```
vendorsetup.sh
AndroidProducts.mk
BoardConfig.mk
Android.mk
```

其中 vendorsetup.sh 是在 envsetup.sh 中被调用的，其他几个 Makefile 的调用时序图及调用关系如图 3-4 所示。

从图 3-4 中可以清楚地看到，Android.mk 是最后才被 main.mk 调用的。换句话说，前面的步骤都是在决定选择“什么产品”以及“产品的属性”，而最后才是考虑该产品的“零件”组成——每个“零件”都由一个 Android.mk 描述。比如我们生产一台电视机，先是通过读取配置来获知电视的具体属性（多大尺寸、是否 LCD、是否壁挂等），最后才是考虑它的具体“零件”组成（某品牌的屏幕、螺丝、支架等）。

整个编译过程中起主导作用的是 main.mk，我们从它的文件名就可以看出来。和 main.mk 同样重要的还有 config.mk，这个专职“配置”的脚本可以说是“产品的设计师”——除了 BoardConfig.mk 和 AndroidProducts.mk 这两个“定制设备”必备的文件外，其他诸如 javac.mk（用于选取合适的 java 编译器）、envsetup.mk（负责环境变量的定义）等 Makefile 也都属于其管辖范围。另外，BUILD 系列变量（即 BUILD_HOST_STATIC_LIBRARY、BUILD_STATIC_LIBRARY）也是在这里赋值的。



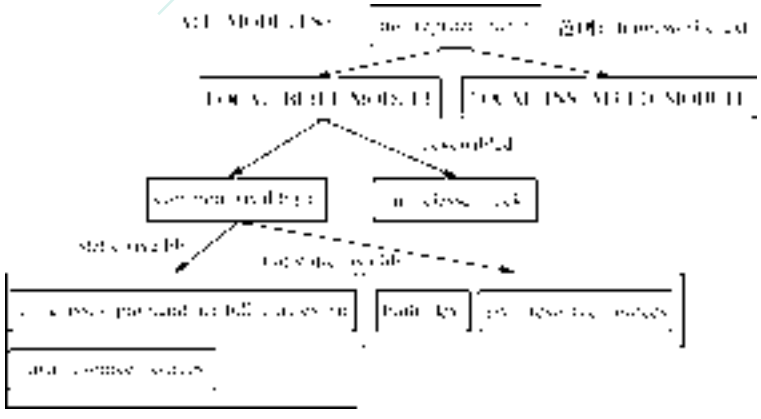
▲图 3-4　各主要 Makefile 的调用时序图

表 3-2 所示是对 Android 编译系统中涉及的主要 Makefile 文件的统一解释，可供有需要的读者参考。

表 3-2　　主要 Makefile 文件释义

Name	Description
main.mk	整个编译系统的主导文件
config.mk	产品配置的主导文件
base_rules.mk	编译系统需要遵循的基础规则定义。其中最重要的变量之一是 ALL_MODULES，它负责将各 Android.mk 中的 LOCAL_MODULE 添加到全局依赖树中，从而保证所有的模块都参与到整个系统的编译中。 另两个起到关键作用的变量是 LOCAL_BUILT_MODULE 和 LOCAL_INSTALLED_MODULE（添加到 ALL_MODULES 的是 my_register_name，而后者则依赖于这两个变量），典型的赋值如下： LOCAL_BUILT_MODULE=out/target/product/generic/obj/JAVA_LIBRARIES/framework_intermediates/javalib.jar LOCAL_INSTALLED_MODULE=out/target/product/generic/system/framework/framework.jar
build_id.mk	版本 id 号的定义
cleanbuild.mk	clean 操作的定义
clear_vars.mk	清空以 LOCAL 开头的相关系统变量，下一小节会看到它的应用场景
definitions.mk	提供了大量实用函数的定义
envsetup.mk	配置编译时的环境变量，注意要与 envsetup.sh 区分开来
executable.mk	负责 BUILD_EXECUTABLE 的具体实现
java.mk	负责与 java 语言相关的编译实现，是 java_library.mk 是基础。 参见表后面的示意图
host_executable.mk	负责 BUILD_HOST_EXECUTABLE 的具体实现
host_static_library.mk	负责 BUILD_HOST_STATIC_LIBRARY 的具体实现 另外，其他类型的 BUILD_XX 变量也都有其对应的 Makefile 文件实现，此处不再赘述
product_config.mk	产品级别的配置，属于 config 的一部分
version_defaults.mk	负责生成版本信息，默认格式为： BUILD_NUMBER := eng.$(USER).$(shell date +%Y%m%d.%H%M%S)

编译生成 Java 库的典型依赖关系如图 3-5 所示。



▲图 3-5　典型依赖关系

Java.mk 中定义了多个中间和最终产物的生成规则，包括：

```
LOCAL_INTERMEDIATE_TARGETS += \
    $(full_classes_compiled_jar) \
    $(full_classes_jarjar_jar) \
    $(full_classes_emma_jar) \
    $(full_classes_jar) \
    $(full_classes_proguard_jar) \
    $(built_dex_intermediate) \
    $(full_classes_jack) \
    $(noshrob_classes_jack) \
    $(built_dex) \
    $(full_classes_stubs_jar)
```

上述这些变量的具体描述如表 3-3 所示。

表 3-3　　具体的描述

变　　量	对应的具体产物	生 成 规 则
full_classes_compiled_jar	classes-no-debug-var.jar 或者 classes-full-debug.jar	$(**full_classes_compiled_jar**): \ $(java_sources) \ $(java_resource_sources) \ $(full_java_lib_deps) \ $(jar_manifest_file) \ $(layers_file) \ $(RenderScript_file_stamp) \ $(proto_java_sources_file_stamp) \ $(LOCAL_MODULE_MAKEFILE) \ $(LOCAL_ADDITIONAL_DEPENDENCIES) $(transform-java-to-classes.jar)
full_classes_jarjar_jar	classes-jarjar.jar	当存在 JarJar 规则时： $(**full_classes_jarjar_jar**): $(**full_classes_compiled_jar**) (LOCAL_JARJAR_RULES) | $(JARJAR) @echo JarJar: $@ $(hide) java -jar $(JARJAR) process $ (PRIVATE_JARJAR_RULES) $< $@
full_classes_emma_jar	lib/ classes-jarjar.jar	$(**full_classes_emma_jar**): $(**full_classes_jarjar_jar**) | $(EMMA_JAR) $(transform-classes.jar-to-emma)
full_classes_jar	classes.jar	$(**full_classes_jar**): $(**full_classes_emma_jar**) | $(ACP) @echo Copying: $@ $(hide) $(ACP) -fp $< $@
full_classes_proguard_jar	proguard.classes.jar 或者 noproguard.classes.jar	$(**full_classes_proguard_jar**) : $(**full_classes_jar**) $(extra_input_jar) $(my_support_library_sdk_raise) $(proguard_flag_files) | $(ACP) $(PROGUARD) $(call transform-jar-to-proguard)
built_dex_intermediate	no-local（或者 with-local）/ classes.dex	$(**built_dex_intermediate**): $(**full_classes_proguard_jar**) $(DX) $(transform-classes.jar-to-dex)
full_classes_jack	classes.jack	$(**full_classes_jack**): $(**jack_all_deps**) @echo Building with Jack: $@ $(java-to-jack)
built_dex	classes.dex	$(**built_dex**): $(**built_dex_intermediate**) | $(ACP) @echo Copying: $@ $(hide) mkdir -p $(dir $@) $(hide) rm -f $(dir $@)/classes*.dex $(hide) $(ACP) -fp $(dir $<)/classes*.dex $(dir $@) ifneq ($(GENERATE_DEX_DEBUG),) $(install-dex-debug) endif

请大家注意观察表 3-3 中的黑色加粗字体，不难发现 java.mk 中的这几个变量之间存在着相互依赖关系，可以说是“环环相扣”：

built_dex-> built_dex_intermediate -> full_classes_proguard_jar -> full_classes_jar->full_classes_emma_jar->full_classes_jarjar_jar->full_classes_compiled_jar->java_sources+java_resource_sources+full_java_lib_deps+jar_manifest_file…

当然，如果开启了 Jack 编译，那么依赖关系会有所不同。Jack 与传统编译方式一个重要的区别就是，它会直接生成最终的 dex 文件——不过在 Static Java Library 的情况下它还需要生成.jack 文件。

3.2.5 droidcore 节点

在编译整个 Android 系统（而不是只有 App）的情况下，droid 依赖于 droidcore 和 dist_files。这一小节先来分析下 droidcore 节点的生成过程。其规则定义如下：

```
# Build files and then package it into the rom formats
.PHONY: droidcore
droidcore: files \
    systemimage \
    $(INSTALLED_BOOTIMAGE_TARGET) \
    $(INSTALLED_RECOVERYIMAGE_TARGET) \
    $(INSTALLED_USERDATAIMAGE_TARGET) \
    $(INSTALLED_CACHEIMAGE_TARGET) \
    $(INSTALLED_VENDORIMAGE_TARGET) \
    $(INSTALLED_FILES_FILE)
```

可以看到，droidcore 依赖于如表 3-4 所示的几个 Prerequisites。

表 3-4 droidcore 的 prerequisites

Prerequisite	Description
files	代表其所依赖的“先决条件”的集合，没有实际意义
systemimage	将生成 system.img
$(INSTALLED_BOOTIMAGE_TARGET)	将生成 boot.img
$(INSTALLED_RECOVERYIMAGE_TARGET)	将生成 recovery.img
$(INSTALLED_USERDATAIMAGE_TARGET)	将生成 userdata.img
$(INSTALLED_CACHEIMAGE_TARGET)	将生成 cache.img
$(INSTALLED_VENDORIMAGE_TARGET)	将生成 vendor.img
$(INSTALLED_FILES_FILE)	将生成 installed-files.txt，用于记录当前系统中预安装的程序、库等模块

这几个“先决条件”的产生原理都类似，因而我们只挑选前几个来做重点分析。

1. files

定义如下：

```
# All the droid stuff, in directories
.PHONY: files
files: prebuilt \
        $(modules_to_install) \
        $(INSTALLED_ANDROID_INFO_TXT_TARGET)
```

（1）prebuilt。“prebuilt”也是一个伪目标，它依赖于$(ALL_PREBUILT)变量。ALL_PREBUILT 机制只用于早期的版本，目前已经被废弃，建议大家使用它的替代品 PRODUCT_COPY_FILES。

（2）modules_to_install。如其名称所示，这个变量描述了系统需要安装的模块。

```
modules_to_install := $(sort \
    $(ALL_DEFAULT_INSTALLED_MODULES) \
    $(product_FILES) \
    $(foreach tag,$(tags_to_install),$($(tag)_MODULES)) \
    $(call get-tagged-modules, shell_$(TARGET_BUILD)) \
    $(CUSTOM_MODULES) \
)
```

由此可见，这些模块被分为五部分，最核心的是下面两种。

- product_FILES

编译该产品涉及的相关文件。与 product_FILES 有直接联系的是 product_MODULES，后者则是基于 PRODUCT_PACKAGES 的处理结果。简而言之，product_FILES 是各 modules（比如 framework，Browser）需要安装文件的列表。

- tags_to_install 所对应的 Modules

在 Android 编译机制中，模块是可以指定编译标志的，如 user、debug、eng、tests。通过给各模块打上相应标志，可以在编译时有选择地避开某些无用的功能部件。

比如开发人员在当次编译时特别指定了 eng 标志，那么所有与此无关的模块都将被排除在最终生成的系统之外。具体而言，$(tag)_MODULES 变量赋值后得到的是 eng_MODULES，后者又由函数 get-tagged-modules 来进一步填充：

```
eng_MODULES := $(sort \
        $(call get-tagged-modules,eng) \
        $(call module-installed-files, $(PRODUCTS.$(INTERNAL_PRODUCT).PRODUCT_PACKAGES_ENG)) \
    )
debug_MODULES := $(sort \
        $(call get-tagged-modules,debug) \
        $(call module-installed-files, $(PRODUCTS.$(INTERNAL_PRODUCT).PRODUCT_PACKAGES_DEBUG)) \
    )
```

接下来我们分析 get-tagged-modules 的函数实现：

```
# Given an accept and reject list, find the matching
# set of targets.  If a target has multiple tags and
# any of them are rejected, the target is rejected.
# Reject overrides accept.
# $(1): list of tags to accept
# $(2): list of tags to reject
#TODO(dbort): do $(if $(strip $(1)),$(1),$(ALL_MODULE_TAGS))
#TODO(jbq): as of 20100106 nobody uses the second parameter
define get-tagged-modules
$(filter-out \
    $(call modules-for-tag-list,$(2)), \
      $(call modules-for-tag-list,$(1)))
endef
```

可以看到，这个函数并不只单纯寻找符合 tag 要求的候选者，而是综合考虑了“可接受”和“拒绝”两种情况。简单来说，当一个目标既有“可接受”标签，又有“拒绝”标签时，它仍然会被淘汰。

“可接受”对象的查找则由 modules-for-tag-list 来实现。其定义如下：

```
# Given a list of tags, return the targets that specify
# any of those tags.
# $(1): tag list
define modules-for-tag-list
$(sort $(foreach tag,$(1),$(ALL_MODULE_TAGS.$(tag))))
endef
```

需要注意的是，main.mk 在后续编译过程中还会对 modules_to_install 进行若干次过滤，目的就是去掉不符合条件和重复无效的部分。

（3）$(INSTALLED_ANDROID_INFO_TXT_TARGET)的生成流程和前两个变量相似，读者可以作为练习自行分析。

2. systemimage

systemimage 的规则定义在 build/core/Makefile 中：

```
$(INSTALLED_SYSTEMIMAGE): $(BUILT_SYSTEMIMAGE) $(RECOVERY_FROM_BOOT_PATCH) | $(ACP)
  @echo "Install system fs image: $@"
  $(copy-file-to-target)
  $(hide) $(call assert-max-image-size,$@ $(RECOVERY_FROM_BOOT_PATCH),$(BOARD_SYSTEMIMAGE_PARTITION_SIZE),yaffs)

systemimage: $(INSTALLED_SYSTEMIMAGE)
```

不难看出，这个规则中最重要的 Prerequisite 是$(BUILT_SYSTEMIMAGE)，它和 systemimage 一样也是在同一个 Makefile 文件中定义的。最终 system.img 文件就是通过它来生成的，采用的 COMMAND 是：

```
$(call build-systemimage-target,$@)
```

参数“$@”是 Makefile 定义的一个自动化变量，代表所有 targets 的集合。

这样就可以成功编译出 system.img 了。

总的来说，droidcore 负责生成系统的所有可运行程序包，包括 system.img，boot.img，recovery.img 等。

3.2.6 dist_files

根节点 droid 还有另一个依赖，即 dist_files。它在整个编译项目中只出现了一次，如：

```
# dist_files only for putting your library into the dist directory with a full build.
.PHONY: dist_files
```

按照 Android 的设计，当这个目标节点起作用时，会在 out 目录下产生专门的 dist 文件夹，用于存储多种分发包。而通常情况下，它既不做任何工作，也不会对我们理解整个编译系统产生影响，因此读者可以选择跳过。

3.2.7 Android.mk 的编写规则

一棵大树的繁茂和枝叶的多寡是息息相关的。一方面，只有茁壮的枝干才能提供足够的养分来供给它的众多细枝末叶；另一方面，树叶的光合作用同样可以滋养整棵大树，从而使其呈现出欣欣向荣之态。

我们曾不止一次地提到过，Android 系统是由非常多的子项目组成的。一款优秀的开放式操作系统，除了要能在原生态系统中预先兼容诸多第三方模块外，还应该具有良好的后期动态扩展性。比如一家互联网厂商希望将其自行研发的某个 APK 应用程序集成进 Android 系统中；或者某家主打“音乐概念”的手机公司打算把某个音频解析库编译进系统版本中。

那么，良好的动态扩展性对这两家公司而言就意味着不需要花太多时间去关心整个 Android 编译体系的运作原理，就能完成以上两个需求——这就像驾驶员适当地了解发动机内部结构能让他们在道路上更得心应手，但我们不能因此就强制要求大家必须理解发动机原理才能开车。

这就是 Android.mk 的一大意义所在。

Android.mk 在整个源码工程中随处可见，保守估计其总体数量在一千个以上。

那么如此之多的文件，又是如何整合进庞大的编译系统而保证不会出错的呢？下面的代码是将源码工程中所有 Android.mk 添加进编译系统的处理过程（在 main.mk 中定义）：

```
ifneq ($(ONE_SHOT_MAKEFILE),)
include $(ONE_SHOT_MAKEFILE)
CUSTOM_MODULES := $(sort $(call get-tagged-modules,$(ALL_MODULE_TAGS)))
FULL_BUILD :=
# Stub out the notice targets, which probably aren't defined
# when using ONE_SHOT_MAKEFILE.
NOTICE-HOST-%: ;
NOTICE-TARGET-%: ;

else # ONE_SHOT_MAKEFILE

#
# Include all of the makefiles in the system
#
# Can't use first-makefiles-under here because
# --mindepth=2 makes the prunes not work.
subdir_makefiles := \
    $(shell build/tools/findleaves.py --prune=out --prune=.repo --prune=.git $(subdirs) Android.mk)

include $(subdir_makefiles)

endif # ONE_SHOT_MAKEFILE
```

变量 ONE_SHOT_MAKEFILE 和编译选项有关。如果选择了编译整个工程项目，那么这个变量就是空的；否则如果使用了诸如"make mm"之类的部分编译命令时，那么在 mm 的实现里会对 ONE_SHOT_MAKEFILE 进行必要的赋值。

在编译整个工程的情况下，系统所找到的所有 Android.mk 将会先存入 subdir_makefiles 变量中，随后一次性 include 进整个编译文件中。

接下来，我们选取 adb 项目作为例子详细解释 Android.mk 的编写规则及一些注意事项。之所以挑选 adb 程序，是因为本书的工具篇中还会对其内部原理进行剖析——理解一个程序很重要的前提就是读懂它的 Makefile，这将为我们后期的学习打下一定的基础。

Adb 的源码路径在 system/core/adb 中，我们只摘抄其 Android.mk 中的核心实现。具体如下所示：

```
LOCAL_PATH:= $(call my-dir) /*LOCAL_PATH 的位置先于 CLEAR_VARS*/
include $(CLEAR_VARS)
/*CLEAR_VARS 的定义在/build/core/clear_vars.mk 中，它清除了上百个除 LOCAL_PATH 外的变量。因而
CLEAR_VARS 通常被认为是一个编译模块的开始标志*/

USB_SRCS :=
EXTRA_SRCS :=
/*adb 内部定义的两个变量，将在不同的操作系统环境下赋予不同的文件值。因为不同的操作系统所需的 usb 驱动
是不一样的，所以有此区分*/
ifeq ($(HOST_OS),linux) /*Linux 环境下*/
  USB_SRCS := usb_linux.c
  EXTRA_SRCS := get_my_path_linux.c
  LOCAL_LDLIBS += -lrt -ldl -lpthread
  LOCAL_CFLAGS += -DWORKAROUND_BUG6558362
endif
…/*省略其他操作系统下的类似处理*/
LOCAL_SRC_FILES := \
     adb.c \
     …
     $(EXTRA_SRCS) \
     $(USB_SRCS) \
     utils.c \
     usb_vendors.c
/*LOCAL_SRC_FILES 是一个很重要的变量，它定义了本模块编译所涉及的所有源文件。可以看到，adb 自定义
  的两个变量，即 USB_SRCS 和 EXTRA_SRCS 也在这里被加入到了编译列表中*/
…
ifneq ($(USE_SYSDEPS_WIN32),)
  LOCAL_SRC_FILES += sysdeps_win32.c/*根据实际情况来扩展 LOCAL_SRC_FILES 变量*/
else
  LOCAL_SRC_FILES += fdevent.c
endif

LOCAL_CFLAGS += -O2 -g -DADB_HOST=1  -Wall  -Wno-unused-parameter
LOCAL_CFLAGS += -D_XOPEN_SOURCE -D_GNU_SOURCE
/*上面两行用于添加编译标志，这在编译过程中会起作用*/
LOCAL_MODULE := adb/*所要生成的模块的名称*/
…
LOCAL_STATIC_LIBRARIES := libzipfile libunz libcrypto_static $(EXTRA_STATIC_LIBS)/*
编译过程中要用到的库*/
…
include $(BUILD_HOST_EXECUTABLE)
/*这个语句是整个 Android.mk 的重点。BUILD_HOST_EXECUTABLE 表示我们希望生成一个 HOST 可执行程序。
当然，你也可以根据需要来选择其他“BUILD_XXX”变量，详见后面的总结。每一个编译模块从 CLEAR_VARS 开始，
到这里结束*/
…
/*到目前为止的语句仅定义了 adb host 工具的生成过程,这个程序将运行于开发环境所处的机器上。如在 Windows
操作系统中的 Eclipse 集成开发环境中开展产品研发，那么 adb host 就运行在 Windows 中。它实际上分饰
adb client 和 adb server 两个角色(通过源码中的 ADB_HOST 宏进行区分)。我们从上面 LOCAL_SRC_FILES
所包含的具体文件中也可以看出一些端倪。更多 adb 的分析，请参照本书工具篇中相关章节。*/

/*以下内容是 adbd 的编译配置，这个程序运行于设备端*/
include $(CLEAR_VARS)/*第二个模块编译的开始标志*/
LOCAL_SRC_FILES := \
     adb.c \
```

```
    …
    Utils.c
/*从 LOCAL_SRC_FILES 变量可以发现，adb daemon 和 adb host 所涉及的文件是有重叠的*/
…/*省略和第一个模块中类似的语句*/
LOCAL_MODULE := adbd/*模块名称*/
…
LOCAL_STATIC_LIBRARIES := libcutils libc libmincrypt
include $(BUILD_EXECUTABLE)/*到这里为止，第二个模块的编译配置结束，最终生成 adbd 可执行程序*/
```

通过以上对 Android.mk 文件的分析，我们了解到 ADB 工具分为两部分，即 ADB Host（Client 和 Server）和 ADB Daemon。前者将和开发环境运行于同一机器平台中，后者则面向设备本身。

上述脚本代码中用到的所有 Android 编译系统提供的函数与变量，我们都特别标注了出来。从中可以看到，通过 Android.mk 添加一个编译模块到系统中的顺序如下：

- LOCAL_PATH；
- CLEAR_VARS；
- LOCAL_SRC_FILES；
- LOCAL_CFLAGS（可选）；
- LOCAL_MODULE；
- LOCAL_STATIC_LIBRARIES（可选）；
- BUILD_HOST_EXECUTABLE/BUILD_EXECUTABLE 等。

最后，总结下 Android.mk 的几个使用要点。

（1）我们可以把 Android.mk 的处理步骤与食品制作过程做个类比。

- 准备食材。包括 LOCAL_SRC_FILES，LOCAL_MODULE，LOCAL_STATIC_LIBRARIES 的收集。虽然每道菜都会有不同的原料需求，但细心的厨师（Android 编译系统）事先都考虑到了。他做好了合理的分类（调味类、肉类、蔬菜类等），并提供了完整的清单。我们所要做的工作就是按照这些清单逐项采购。

- 烹饪菜肴。准备好食材以后，就可以开始佳肴的烹饪了。得益于 Android 编译系统的用心，这个过程比我们想象中的简单很多。因为同一道菜的烧制工序是一样的，所以 Google 事先就把这些步骤集合起来，并做好了各种模板，如 BUILD_STATIC_LIBRARY，BUILD_SHARED_LIBRARY，BUILD_EXECUTABLE 等。一旦顾客点了哪道菜，我们只要调用这些固有模板就可以轻松完成整个烹饪过程，非常方便。

（2）每个 Android.mk 都允许同时煮几道菜，每个清单将以下面的语句开始：

```
include $(CLEAR_VARS)
```

并以如下语句结束：

```
include $(BUILD_XXX)
注：BUILD_XXX 代表上面所提及的各个编译模板
```

（3）表 3-5 所示是在编写 Android.mk 时所涉及的常用重要变量，供读者参考。

表 3-5　　Android.mk 中常用的重要变量解析

变　量　名	说　　明
LOCAL_PATH	用于确定源码所在的目录，最好把它放在 CLEAR_VARS 变量引用的前面。因为它不会被清除，每个 Android.mk 只需要定义一次即可
CLEAR_VARS	它清空了很多以“LOCAL_”开头的变量（LOCAL_PATH 除外）。由于所有的 Makefile 都是在一个编译环境中执行的，因此变量的定义理论上都是全局的，在每个模块编译开始前进行清理工作是必要的
LOCAL_MODULE	模块名，需保证在整个编译系统中是唯一存在的，而且中间不可以有空格
LOCAL_MODULE_PATH	模块的输出路径

续表

变　量　名	说　　明
LOCAL_SRC_FILES	模块编译过程所涉及的源文件。如果是 Java 程序，可以考虑调用 all-subdir-java-files 来一次性添加目录（包括子目录）下所有的 Java 文件 因为有 LOCAL_PATH，这里只需要给出文件名（相对路径）即可；而且编译系统有比较强的推导功能，可以自动计算依赖关系
LOCAL_CC	用于指定 C 编译器
LOCAL_CXX	用于指定 C++编译器
LOCAL_CPP_EXTENSION	用于指定特殊的 C++文件后缀名
LOCAL_CFLAGS	C 语言编译时的额外选项
LOCAL_CXXFLAGS	C++语言编译时的额外选项
LOCAL_C_INCLUDES	编译 C 和 C++程序所需的额外头文件
LOCAL_STATIC_LIBRARIES	编译所需的静态库列表
LOCAL_SHARED_LIBRARIES	编译所需的共享库列表
LOCAL_JAVA_LIBRARIES	编译时所需的 JAVA 类库
LOCAL_LDLIBS	编译时所需的链接选项
LOCAL_COPY_HEADERS	安装应用程序时所需复制的头文件列表，需要和 LOCAL_COPY_HEADERS_TO 变量配合使用
LOCAL_COPY_HEADERS_TO	上述头文件列表的复制目的地
BUILD_HOST_STATIC_LIBRARY BUILD_HOST_SHARED_LIBRARY BUILD_STATIC_LIBRARY BUILD_RAW_STATIC_LIBRARY BUILD_SHARED_LIBRARY BUILD_EXECUTABLE BUILD_RAW_EXECUTABLE BUILD_HOST_EXECUTABLE BUILD_PACKAGE BUILD_HOST_PREBUILT BUILD_PREBUILT BUILD_MULTI_PREBUILT BUILD_JAVA_LIBRARY BUILD_STATIC_JAVA_LIBRARY BUILD_HOST_JAVA_LIBRARY BUILD_DROIDDOC BUILD_COPY_HEADERS BUILD_KEY_CHAR_MAP	各种形式的编译模板，如生成设备端或者 Host 端的静态、动态库文件（Java 和 C/C++等），可执行文件，文档等 需要大家特别注意的是 BUILD_JAVA_LIBRARY 和 BUILD_STATIC_JAVA_LIBRARY 之间的区别。后者生成的 Java Library 既不会被安装，也不会被放到 Java 的 Classpath 中。而 BUILD_JAVA_LIBRARY 得到的 Java 库是具有“共享”性质的，可以为多个程序所共用（会被复制到/system/framework 中）

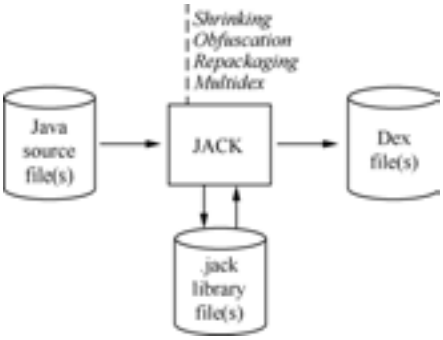
在本章的学习中，我们先从最基础的 Makefile 语法规则入手，导出了依赖树的概念；然后按照由上而下的顺序，逐步梳理出编译一个完整的 Android 版本所涉及的几个重要节点。

Android 编译系统是非常庞大的，所包括的节点和文件数量远不止本章所描述的。不过，我们的目的是让读者掌握一种有效的分析编译系统的方法——相信只要按照本章的分析方法，再一步步地进行推导论断，整个 Android 编译体系就会“水落石出”了。

3.3 Jack Toolchain

从 6.0 版本开始，Android 编译系统中一个比较大的变化是采用了全新的 Java 编译链，即 Jack（Java Android Compiler Kit）。

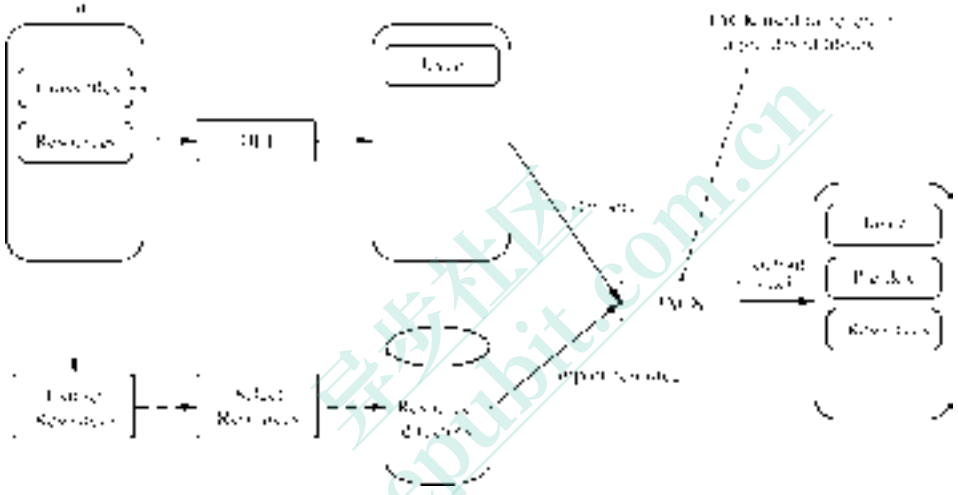
Jack 的主要任务是取代以前版本中的 javac、ProGuard、jarjar、dx 等诸多工具，以一种全新的方式来将 Java 源文件编译成 Android 的 Dex 字节码。它的优势在于可以加快编译速度，具有内建的 shrinking、obfuscation、repackaging 和 multidex 等功能，并且完全开源，如图 3-6 所示。



▲图 3-6 Jack 简图

（引用自 source.android.com）

Jack 有自己的文件格式，这就不可避免地需要利用一个工具来将它与传统的.jar 文件进行转换——即 Jill（Jack Intermediate Library Linker）。如图 3-7 所示。



▲图 3-7 Jill 工作流程

（引用自 source.android.com）

接下来我们重点讲解一下 Jack 在 Android 系统工程编译中的一些特点，以便大家可以学以致用。虽然 Android 官方承诺使用 Jack 的情况和以往的编译过程没有任何区别，但事实上总还是会出现因为 Jack 而导致编译失败的情况。譬如笔者就曾在 M 版本编译中遇到过如图 3-8 所示的问题。

```
Building with Jack: out/target/common/obj/JAVA_LIBRARIES/core-libart_intermediat
es/with-local/classes.dex

Launching background server java -Dfile.encoding=UTF-8 -Xms2560m -XX:+TieredComp
ilation -jar out/host/linux-x86/framework/jack-launcher.jar -cp out/host/linux-x
86/framework/jack.jar com.android.jack.server.JackSimpleServer
out/host/linux-x86/bin/jack: line 131: 24820 Killed                  $SERVER_PRO
 $SERVER_PORT_SERVICE $SERVER_PORT_ADMIN $SERVER_COUNT $SERVER_NB_COMPILE $SERVE
R_TIMEOUT >> $SERVER_LOG 2>&1
ERROR: Cannot launch Jack server
make: *** [out/target/common/obj/JAVA_LIBRARIES/core-libart_intermediates/with-l
ocal/classes.dex] Error 41
```

▲图 3-8 编译时出现的问题

想要解决上述这个错误，首先就得理解 Jack 在编译系统中的工作过程。

Jack 可以加快编译速度的一个重要原因是提供了一个能随时待命的 Server，与以往的 JRE JVM 相比，这种方式显然可以节省启动和初始化的时间（这有点类似于 Gradle 的做法）。当然，如果一直没有新的编译任务到达，一定时长后，Jack Server 也会自动关闭，以避免对系统资源的无端消耗。

Jack Server 对应的配置文件是$HOME/.jack，其中包含了如表 3-6 所示的一些重要信息。

表 3-6　　重要的信息

Config Item	Description
SERVER=true	启用 Jack Server 功能，默认情况下是打开的
SERVER_PORT_SERVICE=8072	设置 Server 用于编译功能的端口号
SERVER_PORT_ADMIN=8073	设置 Server 用于管理功能的端口号
SERVER_COUNT=1	目前还未用到
SERVER_NB_COMPILE=4	并行编译的最大数量
SERVER_TIMEOUT=60	没有编译任务多长时间后 Server 应该关闭自身
SERVER_LOG=${SERVER_LOG:=$SERVER_DIR/jack-$SERVER_PORT_SERVICE.log}	设置 Server 运行时 Log 文件的存储路径
JACK_VM_COMMAND=${JACK_VM_COMMAND:=java}	在 Host 机上启动一个 JVM 实例的默认命令

根据前述问题的 Log 描述，我们不难发现编译失败的缘由是无法启动 Jack Server。针对这种情况最可疑的原因就是 Server 所需的端口已经被 Host 机中其他程序占用了。所以解决办法就是在.jack 配置文件中对 Jack Server 的端口号进行重新调整（建议选一个高位不常用的端口号，避免冲突）。

不过很遗憾，更改端口号对笔者的这个编译错误没有产生任何效果。再仔细观察一下编译过程输出的 Log，可以发现另一个可疑点（开发人员也可以查看 Jack Server 输出的 Log 文件来定位问题），如图 3-9 所示。

▲图 3-9　发现问题

可以看到 Jack Server 为虚拟机申请了高达 2560m 的堆内存，而笔者给编译 Android 系统的虚拟机环境只配置了 2GB 的内存，所以显然也会导致失败。解决的办法也很简单，就是扩大内存配额，然后重试——这次问题确实得到了解决，并成功完成了整个编译过程，参见图 3-12。

我们再来观察一下 Jack 编译链下的中间文件。

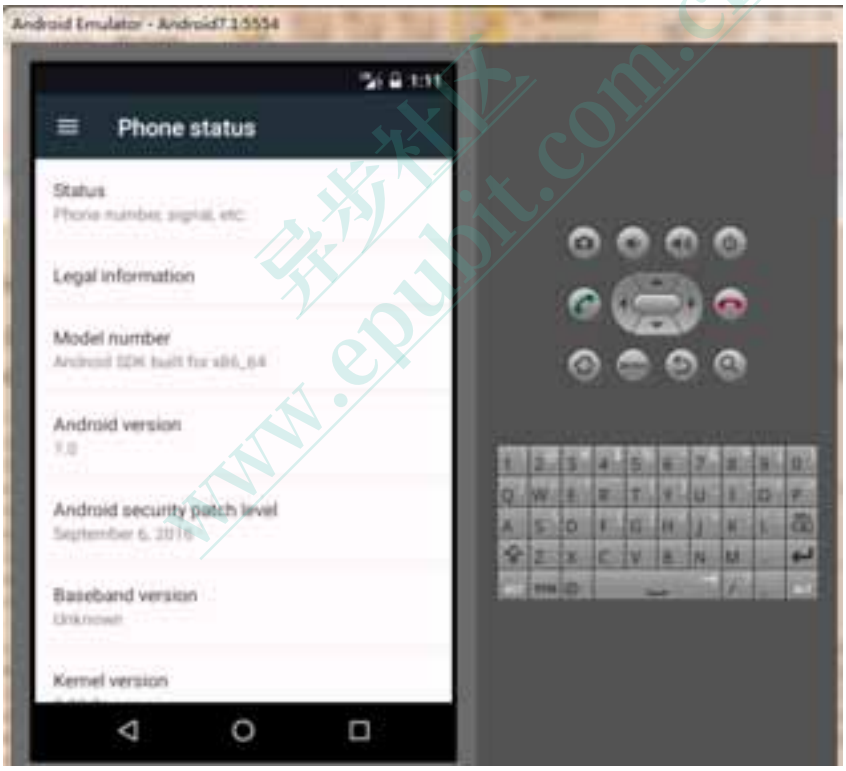
其中 BUILD_JAVA_LIBRARY 的中间文件状态如图 3-10 所示。

而 BUILD_STATIC_JAVA_LIBRARY 的中间文件状态如图 3-11 所示。

▲图 3-10　Jack 不再输出旧版本中的 classes.jar 等文件

▲图 3-11 中间文件夹状态

不难发现，Jack 并不输出中间状态的 jar 文件，而是直接得到最终的 dex 产物——这也是它会导致一些分析工具失效的原因，例如著名的 Jacoco 代码覆盖率工具。



▲图 3-12 Android 7.x 模拟器运行效果

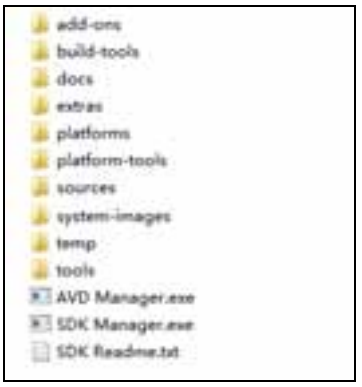
由于 Jack 是一项 experimental 的特性，或多或少都存在着一些 bug，所以，大家在开发过程中，（包括应用程序和系统开发）如果遇到了确实是由于 Jack 所引发的问题，而且这个问题暂时还无法得到解决的话，那么可以选择不用 Jack 来进行编译（注意：并不是所有情况下都适用，比如存在其他依赖关系的时候）。具体做法是在 Android.mk 中添加如下语句：

```
LOCAL_JACK_ENABLED := disabled
```

3.4 SDK 的编译过程

采用 Android 系统的设备厂商越来越多，不少公司便开始考虑如何编译出具有自己特色的扩展 SDK。这种做法除了满足开发人员的需求外，还可以通过 SDK 来开放设备自身的优势，从而吸引更多的用户，提高市场竞争力。因而我们觉得很有必要分析一下 SDK 的生成过程，以便大家可以根据自己的需求来进行定制修改。

简单来说，SDK 是 Android 系统提供给开发人员的一个“工具集”。它的典型文件结构大致如图 3-13 所示。



▲图 3-13　Android SDK 的典型文件结构

接下来我们按照编译 Android SDK 的步骤来进行讲解，并从编译脚本内部实现的角度来分析它的整个生成过程，同时也希望可以让读者借此更深入地理解 Android 编译系统。

3.4.1 envsetup.sh

相信通过前面章节的学习，大家都知道在编译 Android 系统之前，首先需要执行 build 目录下的 envsetup.sh 脚本。这个脚本中除了包含后续步骤所需的函数定义外，还会做很多准备工作。具体如下所示：

```
function hmm() {
cat <<EOF
Invoke ". build/envsetup.sh" from your shell to add the following functions to your
environment:
  - lunch: lunch <product_name>-<build_variant>
  - tapas: tapas [<App1> <App2> ...] [arm|x86|mips|armv5|arm64|x86_64|mips64] [eng|
userdebug|user]
  - croot: Changes directory to the top of the tree.
  - m: Makes from the top of the tree.
  - mm: Builds all of the modules in the current directory, but not their dependencies.
  - mmm: Builds all of the modules in the supplied directories, but not their dependencies.
  To limit the modules being built use the syntax: mmm dir/:target1,target2.
  - mma: Builds all of the modules in the current directory, and their dependencies.
  - mmma: Builds all of the modules in the supplied directories, and their dependencies.
  - cgrep: Greps on all local C/C++ files.
  - ggrep: Greps on all local Gradle files.
  - jgrep: Greps on all local Java files.
  - resgrep: Greps on all local res/*.xml files.
  - mangrep: Greps on all local AndroidManifest.xml files.
  - sepgrep: Greps on all local sepolicy files.
  - sgrep: Greps on all local source files.
  - godir: Go to the directory containing a file.
  Environemnt options:
  - SANITIZE_HOST: Set to 'true' to use ASAN for all host modules. Note that
  ASAN_OPTIONS=detect_leaks=0 will be set by default until the
  build is leak-check clean.
  Look at the source to view more functions. The complete list is:
  EOF
  T=$(gettop)
  local A
  A=""
  for i in 'cat $T/build/envsetup.sh | sed -n "/^[ \t]*function /s/function \([a-z_]
*\).*/\1/p" | sort | uniq'; do
  A="$A $i"
  done
  echo $A
}
```

从 hmm 函数可以大概看出 envsetup.sh 所提供的功能，如表 3-7 所示。

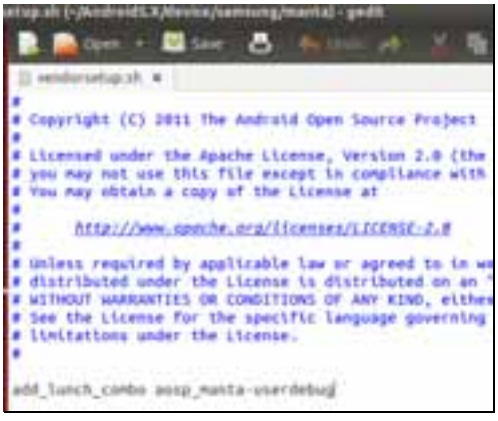
表 3-7　　envsetup.sh 脚本提供的主要函数释义

函　数	描　述
croot	回到树的根位置，即 AOSP 工程的根目录
m	从树根节点开始执行 make
mm	make 当前目录下的所有模块，但是不包括它们的依赖
mmm	make 指定目录下的模块，但是不包括它们的依赖
mma	make 当前目录下的所有模块，以及它们的依赖
mmma	make 指定目录下的模块，以及它们的依赖
cgrep	只针对所有 C/C++文件执行 grep 命令
ggrep	只针对所有 gradle 文件执行 grep 命令
jgrep	只针对所有 Java 文件执行 grep 命令
resgrep	只针对所有 res/*.xml 文件执行 grep 命令
mangrep	只针对所有 AndroidManifest.xml 文件执行 grep 命令
sepgrep	只针对所有 sepolicy 文件执行 grep 命令
sgrep	针对所有源代码文件执行 grep 命令
godir	转到包含指定文件的目录下

除了提供很多实用的函数外，envsetup.sh 在文件的最后还会扫描和加载 device 和 vendor 目录下的 vendorsetup.sh 文件，具体如下所示：

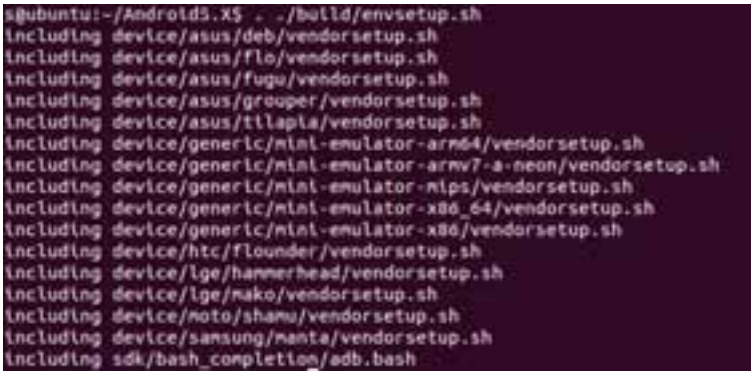
```
for f in 'test -d device && find -L device -maxdepth 4 -name 'vendorsetup.sh' 2> /
dev/null' \
        'test -d vendor && find -L vendor -maxdepth 4 -name 'vendorsetup.sh' 2> /
dev/null'
do
    echo "including $f"
    . $f
done
```

需要特别注意的是，上述脚本对 vendorsetup.sh 文件的扫描深度为 4。由前面章节的学习我们知道，vendorsetup.sh 会通过 add_lunch_combo 命令来为 lunch 添加一条加载项，如图 3-14 是*星手机的一个实例。



▲图 3-14　vendorsetup.sh 示例

当 envsetup.sh 执行结束后，将会打印出搜寻到的 vendorsetup.sh，如图 3-15 所示。



▲图 3-15　打印的信息

3.4.2　lunch sdk-eng

完成 envsetup.sh 的初始化后，我们就可以执行下一步命令了，即 lunch sdk-eng。

大家应该还记得 lunch 命令的用法：要么在 lunch 后加上 product-variant 参数，要么不带任何参数，从而让编译系统打印出当前可选的产品列表，如图 3-16 所示。

▲图 3-16　产品的列表

看了上述列表，不少读者可能会有疑惑——编译 SDK 使用的命令 lunch sdk-eng 并没有在列表中，那么它是合法的吗？

答案是肯定的。实际上 lunch 提供的列表是由开发者在 vendorsetup.sh 或其他文件中通过 add_lunch_combo 提供的，并不是必需的；而真正的重点在于 lunch product-variant 在执行过程中如何验证 product 和 variant 的合法性。

（1）lunch 针对 product 的合法性检查

```
function lunch()
{
```

```
local answer

if [ "$1" ] ; then  # #如果lunch命令带参数，则记录下来
   answer=$1
else
   print_lunch_menu  # #否则打印出产品列表供开发者选择
   echo -n "Which would you like? [aosp_arm-eng] "
   read answer  # #读取用户的选择
fi

local selection=

if [ -z "$answer" ]  # #如果用户什么也没选
then
   selection=aosp_arm-eng  # #默认选择一个
elif (echo -n $answer | grep -q -e "^ [0-9][0-9]*$")# #如果是数字，表示是从列表中选择的
then
   if [ $answer -le ${#LUNCH_MENU_CHOICES[@]} ]# #数字有效
   then
       selection=${LUNCH_MENU_CHOICES[$(($answer-1))]} # #读取相应的值
   fi
elif (echo -n $answer | grep -q -e "^[^\-][^\-]*-[^\-][^\-]*$")   # #lunch带参数
then
    selection=$answer
fi

if [ -z "$selection" ]    # #这时selection不应该为空了
then
    echo
    echo "Invalid lunch combo: $answer"
    return 1
fi

export TARGET_BUILD_APPS=

local product=$(echo -n $selection | sed -e "s/-.*$//")# #从参数中取出product
check_product $product  # #检验product是否合法，下面我们会有详细分析
if [ $? -ne 0 ]## “$?”用于检查上一个函数的退出码，这里是指check_product
then
   echo
   echo "** Don't have a product spec for: '$product'"
   echo "** Do you have the right repo manifest?"
   product=
fi

local variant=$(echo -n $selection | sed -e "s/^[^\-]*-//")# #参数中取variant
check_variant $variant  # #检验variant是否合法，下面我们会有详细分析
if [ $? -ne 0 ]## “$?”用于检查上一个函数的退出码，这里是指check_variant
then
   echo
   echo "** Invalid variant: '$variant'"
   echo "** Must be one of ${VARIANT_CHOICES[@]}"
   variant=
fi

if [ -z "$product" -o -z "$variant" ]
then
    echo
    return 1
fi

export TARGET_PRODUCT=$product  # #将结果值传递给全局变量
```

```
    export TARGET_BUILD_VARIANT=$variant
    export TARGET_BUILD_TYPE=release

    echo

    set_stuff_for_environment
    printconfig
}
```

可以看到，检查 product 的合法性主要依靠的是 check_product 这个函数，具体如下所示：

```
function check_product()
{
    T=$(gettop)
    if [ ! "$T" ]; then
       echo "Couldn't locate the top of the tree.  Try setting TOP." >&2
       return
    fi
       TARGET_PRODUCT=$1 \
       TARGET_BUILD_VARIANT= \
       TARGET_BUILD_TYPE= \
       TARGET_BUILD_APPS= \
       get_build_var TARGET_DEVICE > /dev/null
    # hide successful answers, but allow the errors to show
}
```

这个函数看起来很简单，什么也没做，奥妙就在于最后的 get_build_var TARGET_DEVICE。我们接下来看一下 get_build_var 又做了哪些工作。要特别注意的是，TARGET_PRODUCT 被赋予了第 1 个参数值，即$product，在我们这个例子中就是 sdk。

```
function get_build_var()
{
    T=$(gettop)##回到 AOSP 工程根目录
    if [ ! "$T" ]; then
       echo "Couldn't locate the top of the tree.  Try setting TOP." >&2
       return
    fi
    (\cd $T; CALLED_FROM_SETUP=true BUILD_SYSTEM=build/core \
     command make --no-print-directory -f build/core/config.mk dumpvar-$1)
}
```

这个函数首先回到工程的根目录，然后为几个变量赋值，它们都将在后续过程中发挥作用。最后，它调用了 make 命令，执行的脚本是/build/core/config.mk，而且将目标对象设置为 dumpvar-$1，其中$1 即 get_build_var 的调用参数 TARGET_DEVICE。那么这个 TARGET_DEVICE 的值是什么呢？

```
/*build/core/config.mk*/
…
include $(BUILD_SYSTEM)/envsetup.mk
…
include $(BUILD_SYSTEM)/combo/javac.mk
…
include $(BUILD_SYSTEM)/dumpvar.mk
```

可以看到，config.mk 的主要目的如其名所示，就是为了做各种配置工作，而这其中非常重要的一步配置是通过 envsetup.mk 这个脚本完成的。读者应该特别注意，将这个 envsetup.mk 与之前提到的 envsetup.sh 区分开来：

```
/*build/core/envsetup.mk*/
…
include $(BUILD_SYSTEM)/product_config.mk
…
board_config_mk := \
     $(strip $(wildcard \
          $(SRC_TARGET_DIR)/board/$(TARGET_DEVICE)/BoardConfig.mk \
          $(shell test -d device && find device -maxdepth 4 -path '*/$(TARGET_DEVICE
)/BoardConfig.mk') \
          $(shell test -d vendor && find vendor -maxdepth 4 -path '*/$(TARGET_DEVICE
)/BoardConfig.mk') \
     ))
```

从上面的脚本可以清楚地看到，BoardConfig.mk 的赋值与 TARGET_DEVICE 有关系，因而后者一定要在前者之前被确定下来，更确切来讲就是在 product_config.mk 中了：

```
/*build/core/product_config.mk*/
…
include $(BUILD_SYSTEM)/node_fns.mk
include $(BUILD_SYSTEM)/product.mk
include $(BUILD_SYSTEM)/device.mk

ifneq ($(strip $(TARGET_BUILD_APPS)),)
# An unbundled app build needs only the core product makefiles.
all_product_configs := $(call get-product-makefiles,\
    $(SRC_TARGET_DIR)/product/AndroidProducts.mk)
else
# Read in all of the product definitions specified by the AndroidProducts.mk
# files in the tree.
all_product_configs := $(get-all-product-makefiles)
endif
…
ifneq (,$(filter product-graph dump-products, $(MAKECMDGOALS)))
# Import all product makefiles.
$(call import-products, $(all_product_makefiles))
else
# Import just the current product.
ifndef current_product_makefile ##如果出现错误，编译脚本就会停止，这也是 lunch 检查的真正意义
$(error Can not locate config makefile for product "$(TARGET_PRODUCT)")
endif
ifneq (1,$(words $(current_product_makefile)))
$(error Product "$(TARGET_PRODUCT)" ambiguous: matches $(current_product_makefile))
endif
$(call import-products, $(current_product_makefile))
endif  # Import all or just the current product makefile
TARGET_DEVICE := $(PRODUCTS.$(INTERNAL_PRODUCT).PRODUCT_DEVICE)
…
```

脚本 node_fns.mk、product.mk 和 device.mk 引入了很多函数定义，它们会在后续查找产品定义的操作中产生作用。比如接下来的 get-product-makefiles 和 get-all-product-makefiles 的定义就在 product.mk 中。

如果 TARGET_BUILD_APPS 不为空，那么证明本次并不是全编译，所以只要把核心的产品定义引用进来就可以了。核心产品是由$(SRC_TARGET_DIR)/product/AndroidProducts.mk 提供的，即/build/target/product/AndroidProducts.mk：

```
ifneq ($(TARGET_BUILD_APPS),)
PRODUCT_MAKEFILES := \
    $(LOCAL_DIR)/aosp_arm.mk \
    $(LOCAL_DIR)/full.mk \
    $(LOCAL_DIR)/generic_armv5.mk \
    $(LOCAL_DIR)/aosp_x86.mk \
    $(LOCAL_DIR)/full_x86.mk \
    $(LOCAL_DIR)/aosp_mips.mk \
    $(LOCAL_DIR)/full_mips.mk \
    $(LOCAL_DIR)/aosp_arm64.mk \
```

```
    $(LOCAL_DIR)/aosp_mips64.mk \
    $(LOCAL_DIR)/aosp_x86_64.mk
else
PRODUCT_MAKEFILES := \
    $(LOCAL_DIR)/core.mk \
    $(LOCAL_DIR)/generic.mk \
    $(LOCAL_DIR)/generic_x86.mk \
    $(LOCAL_DIR)/generic_mips.mk \
    $(LOCAL_DIR)/aosp_arm.mk \
    $(LOCAL_DIR)/full.mk \
    $(LOCAL_DIR)/aosp_x86.mk \
    $(LOCAL_DIR)/full_x86.mk \
    $(LOCAL_DIR)/aosp_mips.mk \
    $(LOCAL_DIR)/full_mips.mk \
    $(LOCAL_DIR)/aosp_arm64.mk \
    $(LOCAL_DIR)/aosp_mips64.mk \
    $(LOCAL_DIR)/aosp_x86_64.mk \
    $(LOCAL_DIR)/full_x86_64.mk \
    $(LOCAL_DIR)/sdk_phone_armv7.mk \
    $(LOCAL_DIR)/sdk_phone_x86.mk \
    $(LOCAL_DIR)/sdk_phone_mips.mk \
    $(LOCAL_DIR)/sdk_phone_arm64.mk \
    $(LOCAL_DIR)/sdk_phone_x86_64.mk \
    $(LOCAL_DIR)/sdk_phone_mips64.mk \
    $(LOCAL_DIR)/sdk.mk \
    $(LOCAL_DIR)/sdk_x86.mk \
    $(LOCAL_DIR)/sdk_mips.mk \
    $(LOCAL_DIR)/sdk_arm64.mk \
    $(LOCAL_DIR)/sdk_x86_64.mk
endif
```

可以看到，在确定产品定义时仍然会区分当前是否是全编译的情况。不管是哪种情况，收集到的产品定义项都会放到 PRODUCT_MAKEFILES 中。需要特别注意的是，PRODUCT_MAKEFILES 实际上只是一个文件列表的集合，那么这些文件的具体内容会在什么时候进行解析呢？答案就是前述的 product_config.mk 中的 get-product-makefiles 和 get-all-product-makefiles。这两个函数会逐一取出 PRODUCT_MAKEFILES 列表中的每一个文件名，组成一个路径，然后通过 include 将脚本中的内容读取进来。我们以 sdk.mk 这个列表项为例，它的部分内容节选如下：

```
/*sdk.mk*/
$(call inherit-product, $(SRC_TARGET_DIR)/product/sdk_phone_armv7.mk)
PRODUCT_NAME := sdk

/*sdk_phone_armv7.mk*/
$(call inherit-product, $(SRC_TARGET_DIR)/product/sdk_base.mk)
# Overrides
PRODUCT_BRAND := generic
PRODUCT_NAME := sdk_phone_armv7
PRODUCT_DEVICE := generic

/*sdk_base.mk*/
PRODUCT_PROPERTY_OVERRIDES :=
PRODUCT_PACKAGES := \
     ApiDemos \
     CubeLiveWallpapers \
     CustomLocale \
     Development \
     DevelopmentSettings \
     Dialer \
     EmulatorSmokeTests \
…
```

上述的内容可谓“环环相扣”，颇有点“继承”的关系。这种实现方式可以很好地分离各个脚本的职责，值得大家借鉴。

再回到前面的 TARGET_DEVICE 的赋值中来。还记得 product_config.mk 中的最后一行是怎么给 TARGET_DEVICE 赋值的吗？我们再把它列出来：

```
TARGET_DEVICE := $(PRODUCTS.$(INTERNAL_PRODUCT).PRODUCT_DEVICE)
```

大家肯定会有疑问，这个 PRODUCTS 又是从何而来的呢？它是对所有产品的一个记录集。具体而言，就是调用了 product.mk 中的 import-products，后者又引用了 import-nodes——这个函数则会对 PRODUCTS 和 DEVICES 等变量进行赋值，从而才能保证 TARGET_DEVICE 可以取到正确的值。

总结来说，对 product 的整个检查过程无非是做了两件事：

- 对各种变量进行赋值，保证后续编译的正常运行；
- 在出现错误时直接终止脚本运行，从而有效保证了 product 的准确性。这一点才是体现“合法性检查”的关键核心;而且当 check_product 检查过程中出现错误时，$?不为 0，这样函数退出后就可以通过这一变量进行出错判断，如下所示：

```
if [ $? -ne 0 ]
    then
       echo
       echo "** Don't have a product spec for: '$product'"
       echo "** Do you have the right repo manifest?"
       product=
    fi
```

（2）lunch 对 variant 合法性的检查

和 check_product 类似，check_variant 用于检查 variant 的合法性，在我们这个例子中就是“eng”：

```
/*/build/envsetup.sh*/
VARIANT_CHOICES=(user userdebug eng)
function check_variant()
{
    for v in ${VARIANT_CHOICES[@]}
    do
        if [ "$v" = "$1" ]
        then
            return 0
        fi
    done
    return 1
}
```

可以看到，check_variant 的实现相当简单，直接从 VARIANT_CHOICES 数组（只有 user、userdebug 和 eng 三种）中逐一取出元素进行比较，一旦有匹配的就返回 0；函数结束时仍然没有找到匹配的则返回 1。后面一种情况下，$? -ne 0 会被判定为真，因而脚本在执行过程中将打印出错误，表示当前 variant 是无效的。

3.4.3 make sdk

本小节我们重点分析 make sdk 这一命令的处理流程，它和编译各种系统 image 有不小的差异。

不过和其他 image 一样，sdk 的编译脚本主体在 Main.mk 中。我们节选出其中的关键部分：

```
/*build/core/Main.mk*/

is_sdk_build :=       ##用于标志当前是否是 sdk 编译

ifneq ($(filter sdk win_sdk sdk_addon,$(MAKECMDGOALS)),)##判断是否为 sdk 编译
is_sdk_build := true
endif
…
ifdef is_sdk_build
```

```
# Detect if we want to build a repository for the SDK
sdk_repo_goal := $(strip $(filter sdk_repo,$(MAKECMDGOALS)))
MAKECMDGOALS := $(strip $(filter-out sdk_repo,$(MAKECMDGOALS)))

ifneq ($(words $(filter-out $(INTERNAL_MODIFIER_TARGETS) checkbuild emulator_tests
target-files-package,$(MAKECMDGOALS))),1)
$(error The 'sdk' target may not be specified with any other targets)
endif

# TODO: this should be eng I think.  Since the sdk is built from the eng
# variant.
tags_to_install := debug eng
ADDITIONAL_BUILD_PROPERTIES += xmpp.auto-presence=true
ADDITIONAL_BUILD_PROPERTIES += ro.config.nocheckin=yes
else # !sdk
endif
…
# Bring in all modules that need to be built.
ifeq ($(HOST_OS),windows)
SDK_ONLY := true
endif

ifeq ($(SDK_ONLY),true)
include $(TOPDIR)sdk/build/windows_sdk_whitelist.mk
include $(TOPDIR)development/build/windows_sdk_whitelist.mk

# Exclude tools/acp when cross-compiling windows under linux
ifeq ($(findstring Linux,$(UNAME)),)
subdirs += build/tools/acp
endif

else # !SDK_ONLY
#
# Typical build; include any Android.mk files we can find.
#
subdirs := $(TOP)
FULL_BUILD := true
endif # !SDK_ONLY
…
include $(BUILD_SYSTEM)/Makefile
…
.PHONY: sdk
ALL_SDK_TARGETS := $(INTERNAL_SDK_TARGET)
sdk: $(ALL_SDK_TARGETS)
$(call dist-for-goals,sdk win_sdk, \
    $(ALL_SDK_TARGETS) \
    $(SYMBOLS_ZIP) \
    $(INSTALLED_BUILD_PROP_TARGET) \
)
```

上面这段脚本的关键之一在于 dist-for-goals 这个函数，它的定义如下：

```
/*build/core/distdir.mk*/
define dist-for-goals
$(foreach file,$(2), \
  $(eval fw := $(subst :,$(space),$(file))) \
  $(eval src := $(word 1,$(fw))) \
  $(eval dst := $(word 2,$(fw))) \
  $(eval dst := $(if $(dst),$(dst),$(notdir $(src)))) \
  $(if $(filter $(_all_dist_src_dst_pairs),$(src):$(dst)),\ ##如果已经处理过
    $(eval $(call add-dependency,$(1),$(DIST_DIR)/$(dst))),\ ##那么只添加依赖
    $(eval $(call copy-one-dist-file,\  ##如果没有处理过的情况
      $(src),$(DIST_DIR)/$(dst),$(1)))\
      $(eval _all_dist_src_dst_pairs += $(src):$(dst))\
  )\
)
endef
```

这个函数用于记录 goal 所需要的全部 src:dst 对。它带有两个参数，即：

- $(1)

代表目标对象 goal 的集合，比如上面脚本中对应的是 sdk win_sdk。

- $(2)

为上述参数中的 goal 指定所需的 dist 文件列表。如果 dist 文件中包含冒号，那么“:”之后的表示它在 dist 文件夹中的名称。

dist-for-goals 函数会遍历$(2)中 dist 文件列表的所有 file，并执行以下操作。

（1）将 file 包含的冒号替换成空格。其中 subst 和 eval 都是 makefile 提供的函数，eval 的函数原型为：

```
$(eval arg).
```

这个函数比较特殊，它可以将其他变量和函数的解析结果添加到 make 的语法规则中。为了达到这一目标，eval 的参数会被扩展两次：第一次是由 eval 函数来扩展，然后当它被作为 makefile 语法解析时又会被扩展一次。

（2）fw 的空格前半部分是 src，后半部分是 dst。

（3）if 函数的语法规则为：

```
$(if <condition>,<then-part> ) 或者 $(if <condition>,<then-part>,<else-part> )
```

所以$(if $(dst),$(dst),$(notdir $(src)))的意思就是如果 dst 不为空，那么就取 dst 作为结果；否则取 src 路径中的文件名作为结果。这样就保证了用户没有特别指定 dst 的情况下它有一个默认值。

（4）filter 函数的语法规则为：

```
$(filter <pattern…>,<text> )
```

即应用 pattern 模式来过滤 text 中的内容，允许有多个 pattern 存在。

所以$(filter $(_all_dist_src_dst_pairs)，$(src):$(dst))用于判断 src:dst 这个组合对在_all_dist_src_dst_pairs 中是否已经存在。如果是的话，接下来就只调用 add-dependency 来为 goal 添加依赖关系；否则就需要将其复制到 dst 中。

我们再回到 main.mk 中。可以看到，sdk 这个目标依赖于$(ALL_SDK_TARGETS)，即$(INTERNAL_SDK_TARGET)，那么后面这个变量又是从哪来的呢？

答案就是 build/core/Makefile，它会被 include 到 main.mk 中：

```
include $(BUILD_SYSTEM)/Makefile
```

不光是 sdk，实际上大部分系统 image 的依赖关系都是在 Makefile 这个文件中指定的，可以说它是这些目标生成规则的“集大成者”。我们看一下与 sdk 相关的部分：

```
/*build/core/Makefile*/
INTERNAL_SDK_TARGET := $(sdk_dir)/$(sdk_name).zip ####Comment 1
…
$(INTERNAL_SDK_TARGET): $(deps)####Comment 2
     @echo "Package SDK: $@"
     $(hide) rm -rf $(PRIVATE_DIR) $@
     …
     if [ $$FAIL ]; then exit 1; fi ##如果执行过程中产生错误，直接结束
     $(hide) echo $(notdir $(SDK_FONT_DEPS)) | tr " " "\n"  > $(SDK_FONT_TEMP)/fontsInSdk.txt
     $(hide) ( \
          ATREE_STRIP="strip -x" \
          $(HOST_OUT_EXECUTABLES)/atree \ ####Comment 3
          $(addprefix -f ,$(PRIVATE_INPUT_FILES)) \
               -m $(PRIVATE_DEP_FILE) \
               -I . \
               -I $(PRODUCT_OUT) \
               -I $(HOST_OUT) \
               -I $(TARGET_COMMON_OUT_ROOT) \
```

```
                -v "PLATFORM_NAME=android-$(PLATFORM_VERSION)" \
                -v "OUT_DIR=$(OUT_DIR)" \
                -v "HOST_OUT=$(HOST_OUT)" \
                -v "TARGET_ARCH=$(TARGET_ARCH)" \
                -v "TARGET_CPU_ABI=$(TARGET_CPU_ABI)" \
                -v "DLL_EXTENSION=$(HOST_SHLIB_SUFFIX)" \
                -v "FONT_OUT=$(SDK_FONT_TEMP)" \
                -o $(PRIVATE_DIR) && \
        $(PRIVATE_DIR)/system-images/android-$(PLATFORM_VERSION)/$(TARGET_CPU_ABI)/NOTICE.txt && \
             cp -f $(tools_notice_file_txt) $(PRIVATE_DIR)/platform-tools/NOTICE.txt && \
             HOST_OUT_EXECUTABLES=$(HOST_OUT_EXECUTABLES) HOST_OS=$(HOST_OS) \
                  development/build/tools/sdk_clean.sh $(PRIVATE_DIR) && \
             chmod -R ug+rwX $(PRIVATE_DIR) && \
             cd $(dir $@) && zip -rq $(notdir $@) $(PRIVATE_NAME) \ ###压缩 sdk 包
        ) || ( rm -rf $(PRIVATE_DIR) $@ && exit 44 )
```

Comment 1：将 INTERNAL_SDK_TARGET 赋值为$(sdk_dir)/$(sdk_name).zip，其中 sdk_dir 指的是 $(HOST_OUT)/sdk/$(TARGET_PRODUCT)，即 /out/host/sdk/$(TARGET_PRODUCT); 而 sdk_name 则为 android-sdk_$(FILE_NAME_TAG)，典型情况下的值为：

```
#     linux-x86   --> android-sdk_12345_linux-x86
#     darwin-x86  --> android-sdk_12345_mac-x86
#     windows-x86 --> android-sdk_12345_windows
```

Comment 2：INTERNAL_SDK_TARGET := $(sdk_dir)/$(sdk_name).zip，下面是这个变量在典型情况下的值为：out/host/linux-x86/sdk/sdk/android-sdk_eng.s_linux-x86.zip。

由前面的分析可知，sdk 这个伪目标依赖于 INTERNAL_SDK_TARGET，后者则实际上依赖于$(deps)——这个变量是一个生成物的列表，如下截图是部分节选：

它代表了生成 sdk 需要编译的所有中间产物。那么这么多文件内容又是如何产生的呢？

```
/*build/core/Makefile*/

deps := \
     $(target_notice_file_txt) \
     $(tools_notice_file_txt) \
     $(OUT_DOCS)/offline-sdk-timestamp \
     $(SYMBOLS_ZIP) \
     $(INSTALLED_SYSTEMIMAGE) \
     $(INSTALLED_USERDATAIMAGE_TARGET) \
     $(INSTALLED_RAMDISK_TARGET) \
     $(INSTALLED_SDK_BUILD_PROP_TARGET) \
     $(INSTALLED_BUILD_PROP_TARGET) \
     $(ATREE_FILES) \
```

```
    $(sdk_atree_files) \
    $(HOST_OUT_EXECUTABLES)/atree \
    $(HOST_OUT_EXECUTABLES)/line_endings \
    $(SDK_FONT_DEPS)
```

我们知道，SDK 包含的东西比较多，除了各种文档和 framework 中间件（android.jar）外，还有平台工具、字体，以及各种 image 文件（如 system 和 userdata，这些都是模拟器运行所必需的）等。

因为 deps 变量所涉及的内容比较多，我们接下来会挑选 android.jar 这个最重要的文件作为例子进行讲解，读者也可以根据需要自行分析其他产物。

Comment 3: 可以看到 deps 变量中有不少与 atree 相关的文件，它们是 sdk 最终产物的描述文件。比如下面这个范例：

```
/*development/build/sdk.atree*/
…
# host tools from out/host/$(HOST_OS)-$(HOST_ARCH)/
bin/adb                                 strip platform-tools/adb
bin/fastboot                            strip platform-tools/fastboot
bin/sqlite3                             strip platform-tools/sqlite3
bin/dmtracedump                         strip platform-tools/dmtracedump
bin/etc1tool                            strip platform-tools/etc1tool
bin/hprof-conv                          strip platform-tools/hprof-conv
```

Atree 文件中的内容分为两列，左边一列表达的是“source”，右边则是“destination”。大家应该已经想到了，既然有 atree 格式的描述文件，那么一定会有工具来解析这些格式。

完成这一任务的是 atree，它所支持的主要参数选项及释义如表 3-8 所示。

表 3-8　　工具 atree 参数选项表

Option	Description
-f FILELIST	FILELIST 是文件列表，其中的文件用于记录需要被复制的一系列文件名
-I INPUTDIR	指定基准目录，帮助 atree 查找上述的被复制文件
-o OUTPUTDIR	指定复制的目标路径
-l	使用硬链接，而不是真的复制文件
-m DEPENDENCY	输出一个 make 格式的文件
-v VAR=VAL	在读取输入文件过程中，将$VAR 替换成 VAL
-d	调试模式，用于输出额外的信息

对照上述的参数选项列表，我们再来重点看下 Comment 3 部分的实现：

```
ATREE_STRIP="strip -x" \
          $(HOST_OUT_EXECUTABLES)/atree \ ####Comment 3
          $(addprefix -f ,$(PRIVATE_INPUT_FILES)) \ ####-f 选项
               -m $(PRIVATE_DEP_FILE) \
               -I . \
               -I $(PRODUCT_OUT) \
               -I $(HOST_OUT) \
               -I $(TARGET_COMMON_OUT_ROOT) \
               -v "PLATFORM_NAME=android-$(PLATFORM_VERSION)" \
               -v "OUT_DIR=$(OUT_DIR)" \
               -v "HOST_OUT=$(HOST_OUT)" \
               -v "TARGET_ARCH=$(TARGET_ARCH)" \
               -v "TARGET_CPU_ABI=$(TARGET_CPU_ABI)" \
               -v "DLL_EXTENSION=$(HOST_SHLIB_SUFFIX)" \
               -v "FONT_OUT=$(SDK_FONT_TEMP)" \
               -o $(PRIVATE_DIR) && \
```

其中，-f 选项所带的参数是$(PRIVATE_INPUT_FILES)，后者因为是一个文件列表，所以需要通过 addprefix 为它们全部加上-f 选项。那么$(PRIVATE_INPUT_FILES)具体包含了哪些文件呢？

由前面的脚本文件不难分析出，$(PRIVATE_INPUT_FILES)实际上等价于 sdk_atree_files，进一步来讲，就是包含了如下的文件：

```
sdk_atree_files := \
     $(atree_dir)/sdk.exclude.atree \
     $(atree_dir)/sdk-$(HOST_OS)-$(SDK_HOST_ARCH).atree
…
$(atree_dir)/sdk-android-$(TARGET_CPU_ABI).atree
…
$(atree_dir)/sdk.atree
```

其中 atree_dir 指定的是 development/build，这个目录下包含了不少 atree 文件，如下所示：

以我们关心的 android.jar 为例，与它相对应的描述语句如下所示：

```
# the uper-jar file that apps link against. This is the public API
${OUT_DIR}/target/common/obj/PACKAGING/android_jar_intermediates/android.jar
platforms/${PLATFORM_NAME}/android.jar
```

也就是说，通过 atree 文件可以将 out 目录下编译生成的 android.jar 复制到 sdk 目标路径下的 platforms/${PLATFORM_NAME}/文件夹中。现在的问题转变为，android.jar 又是如何生成的呢？

答案就在 development/build/Android.mk 中，我们一起来看一下：

```
# ===== SDK jar file of stubs =====
# A.k.a the "current" version of the public SDK (android.jar inside the SDK package).
sdk_stub_name := android_stubs_current
stub_timestamp := $(OUT_DOCS)/api-stubs-timestamp
include $(LOCAL_PATH)/build_android_stubs.mk

.PHONY: android_stubs
android_stubs: $(full_target)
…
# android.jar is what we put in the SDK package.
android_jar_intermediates := $(TARGET_OUT_COMMON_INTERMEDIATES)/PACKAGING/android_
jar_intermediates
android_jar_full_target := $(android_jar_intermediates)/android.jar

$(android_jar_full_target): $(full_target)
     @echo Package SDK Stubs: $@
     $(hide)mkdir -p $(dir $@)
     $(hide)$(ACP) $< $@

ALL_SDK_FILES += $(android_jar_full_target)
```

其中 android_jar_full_target 是 android.jar 中 out 目录中的绝对路径，即$(TARGET_OUT_COMMON_INTERMEDIATES)/PACKAGING/android_jar_intermediates/android.jar，而且它依赖于 full_target 变量——从名称上可以猜到，这个变量用于记录所有 android.jar 包中所需的文件。

根据 make 的规则，一旦 full_target 比目标 android_jar_full_target 新，那么就会执行以下的命令，包括：

- 打印出“Package SDK Stubs:[目标对象]”。
- 新建 android_jar_full_target 所指向的目录，即$(TARGET_OUT_COMMON_INTERMEDIATES)/PACKAGING/android_jar_intermediates。
- 调用$(ACP)执行实际的复制和打包工作。

$(ACP)是 Android 编译系统提供的专门用于跨平台复制的一个工具，源码路径是/build/tools/acp。它的用法如下：

```
acp [OPTION]  SOURCE  DEST
```

那么 SOURCE，即 full_target 包含了哪些内容呢？

我们需要在 build_android_stubs.mk 中寻找答案，如下所示：

```
/*development/build/build_android_stubs.mk*/
# Build an SDK jar file out of the generated stubs
intermediates := $(TARGET_OUT_COMMON_INTERMEDIATES)/JAVA_LIBRARIES/$(sdk_stub_name)_
intermediates
full_target := $(intermediates)/classes.jar
src_dir := $(intermediates)/src
classes_dir := $(intermediates)/classes
framework_res_package := $(call intermediates-dir-for,APPS,framework-res,,COMMON)/
package-export.apk

$(full_target): PRIVATE_SRC_DIR := $(src_dir)
$(full_target): PRIVATE_INTERMEDIATES_DIR := $(intermediates)
$(full_target): PRIVATE_CLASS_INTERMEDIATES_DIR := $(classes_dir)
$(full_target): PRIVATE_FRAMEWORK_RES_PACKAGE := $(framework_res_package)

$(full_target): $(stub_timestamp) $(framework_res_package)
     @echo Compiling SDK Stubs: $@
     $(hide) rm -rf $(PRIVATE_CLASS_INTERMEDIATES_DIR)  ###Step1
     $(hide) mkdir -p $(PRIVATE_CLASS_INTERMEDIATES_DIR)
     $(hide) find $(PRIVATE_SRC_DIR) -name "*.java" > \
        $(PRIVATE_INTERMEDIATES_DIR)/java-source-list ###Step2
     $(hide) $(TARGET_JAVAC) -encoding ascii -bootclasspath "" \
               -g $(xlint_unchecked) \
               -extdirs "" -d $(PRIVATE_CLASS_INTERMEDIATES_DIR) \
               \@$(PRIVATE_INTERMEDIATES_DIR)/java-source-list \
          || ( rm -rf $(PRIVATE_CLASS_INTERMEDIATES_DIR) ; exit 41 ) ###Step3
     $(hide) if [ ! -f $(PRIVATE_FRAMEWORK_RES_PACKAGE) ]; then \
          echo Missing file $(PRIVATE_FRAMEWORK_RES_PACKAGE); \
          rm -rf $(PRIVATE_CLASS_INTERMEDIATES_DIR); \
          exit 1; \
     fi; ###Step4
     $(hide) unzip -qo $(PRIVATE_FRAMEWORK_RES_PACKAGE) -d $(PRIVATE_CLASS_INTERMEDIATES_DIR)
     $(hide) (cd $(PRIVATE_CLASS_INTERMEDIATES_DIR) && rm -rf classes.dex META-INF)
     $(hide) mkdir -p $(dir $@)
     $(hide) jar -cf $@ -C $(PRIVATE_CLASS_INTERMEDIATES_DIR) .
     $(hide) jar -u0f $@ -C $(PRIVATE_CLASS_INTERMEDIATES_DIR) resources.arsc ###Step5
```

脚本 build_android_stubs.mk 有两个类似于入参的变量，分别是：

（1）sdk_stub_name：SDK stub 的名称。Stub 源代码应该已经生成在$(TARGET_OUT_COMMON_INTERMEDIATES)/JAVA_LIBRARIES/$(sdk_stub_name)_intermediates 目录中。在这个例子中对应的就是$(TARGET_OUT_COMMON_INTERMEDIATES)/JAVA_LIBRARIES/android_stubs_current_intermediates

（2）stub_timestamp：生成的源码所依赖的时间戳文件

总的来说，build_android_stubs.mk 脚本的目标是编译一个 classes.jar，换句话讲就是我们需要的 android.jar。而被编译的所有源文件必须已经生成到上述的 android_stubs_current_intermediates 文件夹中了，这是由 droiddoc.mk 来完成的，我们后续将会有进一步分析。

现在先来看一下如何编译和打包 classes.jar（即脚本中的 full_target 变量）。

Step1. 首先通过整个文件夹删除来清理掉$(intermediates)/classes 目录中的内容，然后再重新创建这个文件夹。

Step2. 查找$(intermediates)/src 目录下以“java”为后缀的所有文件，并把结果记录到$(intermediates)/java-source-list 中，以供后续使用。

Step3. 这是关键的一个步骤，即通过调用 javac 来完成编译工作。其中 encoding 参数用于指明文件的编码格式，以保证编译能顺利进行。其他参数的官方释义如图 3-17 所示。

▲图 3-17　官方释义

最后通过@<filename>来把前一步生成的 java-source-list 中记录的所有 Java 文件读取出来，然后执行编译过程，中间文件输出到$(intermediates)/classes 中。

Step4. 如果 PRIVATE_FRAMEWORK_RES_PACKAGE 文件缺失，则终止操作并报错。

Step5. 这一步主要是针对前述的步骤进行清理和打包工作。

首先将 framework_res_package 所指向的 apk 通过 unzip 命令进行解压，结果也同样输出到$(intermediates)/classes 中。在以后章节的学习中我们会发现，apk 实际上就是 zip 包。大家可以尝试在 Windows 操作系统下通过 WinRAR 等类似软件打开一个实际的 apk 自行验证下。紧接着删除掉解压后的 classes.dex 和 META-INF 文件夹，换言之我们要的是 apk 中的 res 目录和 resources.arsc。

一切准备就绪，现在就只剩下打包了。大家应该已经猜到了，classes.jar 就是$(intermediates)/classes 文件夹内容的集成。具体操作分为两步，即：

```
jar -cf $@ -C $(PRIVATE_CLASS_INTERMEDIATES_DIR) .
```

创建 classes.jar 文件，然后将$(intermediates)/classes 内容打包进去。

```
jar -u0f $@ -C $(PRIVATE_CLASS_INTERMEDIATES_DIR) resources.arsc
```

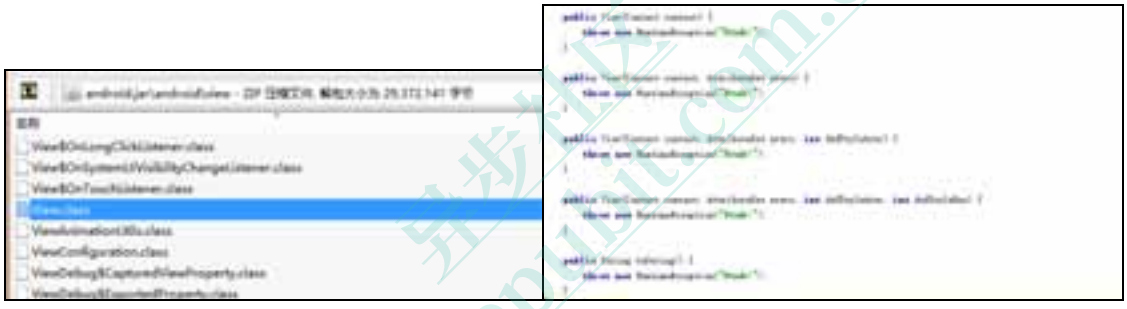
更新 jar 包。

接下来我们再分析一下$(intermediates)/src 中的源文件是如何生成的。

大家可以先思考一下，最有可能生成 stub source 的脚本是哪一个？回答这个问题，要先了解 android.jar 中的内容是什么。我们来看图 3-18 所示的界面。

▲图 3-18　android.jar 中包含的内容

是不是感觉非常熟悉？没错，它们和 framework 的组成基本上是一致的，只不过后者包含了实现体，而 android.jar 只是“空壳”，即 Stub（桩），如图 3-19 所示。



▲图 3-19　android.jar 中不包含实现体

（左：android/view 目录下的 View.class　右：View.class 中的所有函数体实现都只是抛出一个异常。）

所以生成 android.jar 所需 Stub 源码的地方很可能在 framework 中，即/frameworks/base/Android.mk。我们来实际验证一下：

```
# ====  the system api stubs ===================================
include $(CLEAR_VARS)

LOCAL_SRC_FILES:=$(framework_docs_LOCAL_API_CHECK_SRC_FILES)
LOCAL_INTERMEDIATE_SOURCES:=$(framework_docs_LOCAL_INTERMEDIATE_SOURCES)
LOCAL_JAVA_LIBRARIES:=$(framework_docs_LOCAL_API_CHECK_JAVA_LIBRARIES)
LOCAL_MODULE_CLASS:=$(framework_docs_LOCAL_MODULE_CLASS)
LOCAL_DROIDDOC_SOURCE_PATH:=$(framework_docs_LOCAL_DROIDDOC_SOURCE_PATH)
LOCAL_DROIDDOC_HTML_DIR:=$(framework_docs_LOCAL_DROIDDOC_HTML_DIR)
LOCAL_ADDITIONAL_JAVA_DIR:=$(framework_docs_LOCAL_API_CHECK_ADDITIONAL_JAVA_DIR)
LOCAL_ADDITIONAL_DEPENDENCIES:=$(framework_docs_LOCAL_ADDITIONAL_DEPENDENCIES)

LOCAL_MODULE := system-api-stubs

LOCAL_DROIDDOC_OPTIONS:=\
          $(framework_docs_LOCAL_DROIDDOC_OPTIONS) \
          -stubs $(TARGET_OUT_COMMON_INTERMEDIATES)/JAVA_LIBRARIES/android_system_
stubs_current_intermediates/src \
          -showAnnotation android.annotation.SystemApi \
```

```
            -api $(INTERNAL_PLATFORM_SYSTEM_API_FILE) \
            -removedApi $(INTERNAL_PLATFORM_SYSTEM_REMOVED_API_FILE) \
            -nodocs

LOCAL_DROIDDOC_CUSTOM_TEMPLATE_DIR:=build/tools/droiddoc/templates-sdk

LOCAL_UNINSTALLABLE_MODULE := true
include $(BUILD_DROIDDOC)
```

果然如此。具体生成 stub 借助的是 droiddoc，也就是 BUILD_DROIDDOC 变量所指向的 droiddoc.mk：

```
/*build/core/config.mk*/
…
BUILD_DROIDDOC:= $(BUILD_SYSTEM)/droiddoc.mk
```

在调用这个脚本之前，需要给几个重要的变量赋值，包括 LOCAL_MODULE_CLASS、LOCAL_SRC_FILES 和 LOCAL_DROIDDOC_OPTIONS 等。

这些变量将作为 droiddoc 最终产生目标“文档”产物的基础——在我们这个例子中，“文档”意味着 stub source。

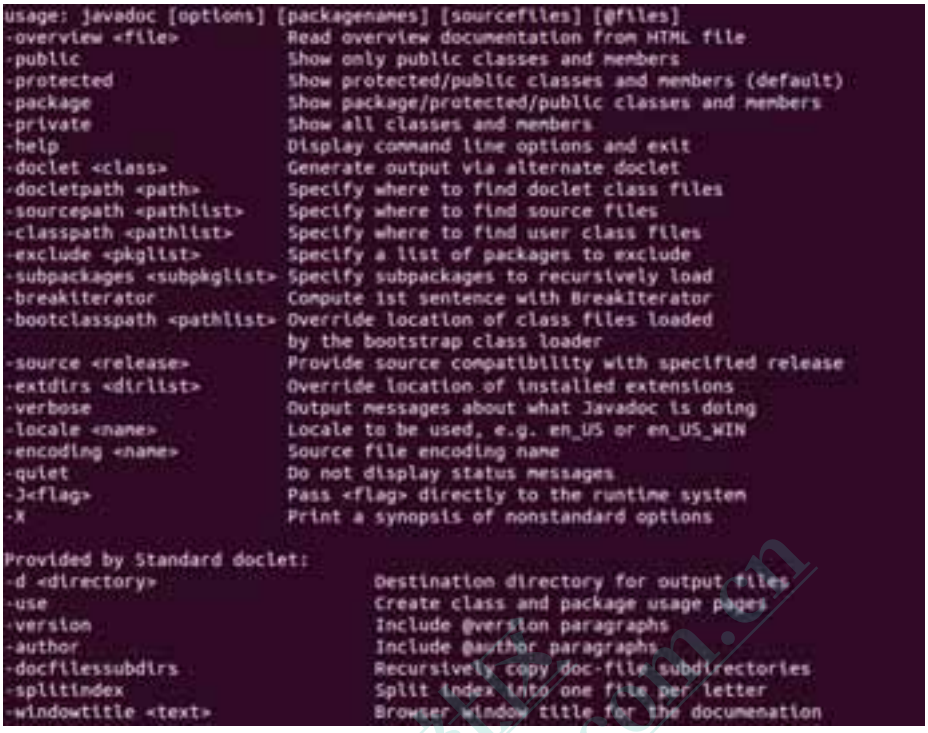
我们只节选 droiddoc.mk 中与最终产物生成有关联的语句，以便大家可以更好地理解整个过程：

```
ifneq ($(strip $(LOCAL_DROIDDOC_USE_STANDARD_DOCLET)),true) ###droiddoc 分支
…
else ###标准 doclet 分支
$(full_target): $(full_src_files) $(full_java_lib_deps)
     @echo Docs javadoc: $(PRIVATE_OUT_DIR)
     @mkdir -p $(dir $@)
     $(call prepare-doc-source-list,$(PRIVATE_SRC_LIST_FILE),$(PRIVATE_JAVA_FILES), \
               $(PRIVATE_SOURCE_INTERMEDIATES_DIR) $(PRIVATE_ADDITIONAL_JAVA_DIR))
     $(hide) ( \
          javadoc \ ###javadoc 是生成 doc 的关键，其余工作都是围绕它展开的
                -encoding UTF-8 \
                $(PRIVATE_DROIDDOC_OPTIONS) \
                \@$(PRIVATE_SRC_LIST_FILE) \
                -J-Xmx1024m \
                -XDignore.symbol.file \
                $(PRIVATE_PROFILING_OPTIONS) \
                $(addprefix -classpath ,$(PRIVATE_CLASSPATH)) \
                $(addprefix -bootclasspath ,$(PRIVATE_BOOTCLASSPATH)) \
                -sourcepath $(PRIVATE_SOURCE_PATH)$(addprefix :,$(PRIVATE_CLASSPATH)) \
                -d $(PRIVATE_OUT_DIR) \
                -quiet \
        && touch -f $@ \
    ) || (rm -rf $(PRIVATE_OUT_DIR) $(PRIVATE_SRC_LIST_FILE); exit 45)
endif
```

目前编译系统对 doclet 的实现方式进行了改进，从而出现了上述脚本中的 if 和 else 语句，即标准的 doclet 和 doclava。不过这些并不是我们这里需要关心的重点，读者有兴趣的话可以自行查阅相关资料进行分析。

相信大家应该都听说过 javadoc 这个工具，它的基本用法如图 3-20 所示。

这样一来 android.jar 所需的 stub 就可以成功生成了，其他的中间件的处理过程也是类似的。一旦它们都处于 Ready 状态后，make sdk 就可以将它们打包，并最终完成 Android SDK 的编译。



▲图 3-20 javadoc 界面

3.5 Android 系统 GDB 调试

GDB 是 GNU Project Debugger 的缩写，它也是很多开源软件的调试利器。对于 Android 这样一个庞大的系统，难免会在开发过程中遇到一些“难缠”的问题，这其中一个非常重要的解决办法就是通过 GDB 进行调试，因而我们觉得有必要向大家介绍一下 GDB 工具在 Android 系统中的典型用法。

实际上 Android 编译系统也已经为开发者使用 GDB 做了一些便利的工作，如/build/envsetup.sh：

```
function gdbclient()
{
   local OUT_ROOT=$(get_abs_build_var PRODUCT_OUT) ##得到 out 目录
   local OUT_SYMBOLS=$(get_abs_build_var TARGET_OUT_UNSTRIPPED)
   local OUT_SO_SYMBOLS=$(get_abs_build_var TARGET_OUT_SHARED_LIBRARIES_UNSTRIPPED)
   local OUT_VENDOR_SO_SYMBOLS=$(get_abs_build_var TARGET_OUT_VENDOR_SHARED_LIBRARIES
_UNSTRIPPED)
   local OUT_EXE_SYMBOLS=$(get_symbols_directory)
##得到 Symbols 目录，这是 GDB 将目标对象与源文件进行对应的关键

   local PREBUILTS=$(get_abs_build_var ANDROID_PREBUILTS)
    ##Prebuilts 目录下有很多 GDB 相关的工具
   local ARCH=$(get_build_var TARGET_ARCH) ##ARCH 决定了需要使用哪一个具体的 GDB 客户端
   local GDB

   case "$ARCH" in
       arm) GDB=arm-linux-androideabi-gdb;;
       arm64) GDB=arm-linux-androideabi-gdb; GDB64=aarch64-linux-android-gdb;;
       mips|mips64) GDB=mips64el-linux-android-gdb;;
       x86) GDB=x86_64-linux-android-gdb;;
```

```
        x86_64) GDB=x86_64-linux-android-gdb;;
        *) echo "Unknown arch $ARCH"; return 1;;
    esac ##通过 ARCH 来选定 GDB 客户端，通常这些客户端工具保存在 prebuilts 目录下

    if [ "$OUT_ROOT" -a "$PREBUILTS" ]; then
        local EXE="$1" ###EXE 代表的是需要被调试的应用程序
        if [ "$EXE" ] ; then
            EXE=$1
            if [[ $EXE =~ ^[^/].* ]] ; then
                EXE="system/bin/"$EXE
            fi
        else
            EXE="app_process" ##默认情况下调试 app_process
        fi

        local PORT="$2" ##端口号，必须与 gdbserver 保持一致
        if [ "$PORT" ] ; then
            PORT=$2
        else
            PORT=":5039" ##端口号默认为 5039
        fi

        local PID="$3" ##PID 表示被调试对象的进程号，也可以只提供进程名，后面情况下需要解析成进程号
        if [ "$PID" ] ; then
            if [[ ! "$PID" =~ ^[0-9]+$ ]] ; then
                PID=`pid $3`
                if [[ ! "$PID" =~ ^[0-9]+$ ]] ; then
                    PID='adb shell ps | \grep $3 | \grep -v ":" | awk '{print $2}''
                    if [[ ! "$PID" =~ ^[0-9]+$ ]]
                    then
                        echo "Couldn't resolve '$3' to single PID"
                        return 1
                    else
                        echo ""
                        echo "WARNING: multiple processes matching '$3' observed,
                    using root process"
                        echo ""
                    fi
                fi
            fi
            adb forward "tcp$PORT" "tcp$PORT" ##端口映射
            local USE64BIT="$(is64bit $PID)"
            adb shell gdbserver$USE64BIT $PORT --attach $PID & ##gdbserver attach
            sleep 2
        else
                echo ""
                echo "If you haven't done so already, do this first on the device:"
                echo "    gdbserver $PORT /system/bin/$EXE"
                    echo " or"
                echo "    gdbserver $PORT --attach <PID>"
                echo ""
        fi

        OUT_SO_SYMBOLS=$OUT_SO_SYMBOLS$USE64BIT
        OUT_VENDOR_SO_SYMBOLS=$OUT_VENDOR_SO_SYMBOLS$USE64BIT
        echo >|"$OUT_ROOT/gdbclient.cmds" "set solib-absolute-prefix $OUT_SYMBOLS"
        echo >>"$OUT_ROOT/gdbclient.cmds" "set solib-search-path $OUT_SO_SYMBOLS:$OUT_SO_
SYMBOLS/hw:$OUT_SO_SYMBOLS/ssl/engines:$OUT_SO_SYMBOLS/drm:$OUT_SO_SYMBOLS/egl:$OUT_SO_SYM
BOLS/soundfx:$OUT_VENDOR_SO_SYMBOLS:$OUT_VENDOR_SO_SYMBOLS/hw:$OUT_VENDOR_SO_SYMBOLS/egl"
        echo >>"$OUT_ROOT/gdbclient.cmds" "source $ANDROID_BUILD_TOP/development/
scripts/gdb/dalvik.gdb"
        echo >>"$OUT_ROOT/gdbclient.cmds" "target remote $PORT"
```

```
        ###将 GDB 需要的一些参数写入脚本文件中
        …

        local WHICH_GDB=
        # 64-bit exe found
        if [ "$USE64BIT" != "" ] ; then
            WHICH_GDB=$ANDROID_TOOLCHAIN/$GDB64
        # 32-bit exe / 32-bit platform
        elif [ "$(get_build_var TARGET_2ND_ARCH)" = "" ]; then
            WHICH_GDB=$ANDROID_TOOLCHAIN/$GDB
        # 32-bit exe / 64-bit platform
        else
            WHICH_GDB=$ANDROID_TOOLCHAIN_2ND_ARCH/$GDB
        fi
        ###WHICH_GDB 是 GDB 客户端的绝对路径

        gdbwrapper $WHICH_GDB "$OUT_ROOT/gdbclient.cmds" "$OUT_EXE_SYMBOLS/$EXE"
    else
        echo "Unable to determine build system output dir."
     fi

}
```

综合上面的分析，可以发现 gdbclient 函数的实现是围绕两个方面展开的。为了让大家可以更快地理解使用 gdb 的整个过程，下面我们以调试 surfaceflinger 为例来讲解。

（1）gdb server

这是运行在 Android 设备端（量产的设备中很可能会被移除）的一个监听服务，通常路径为 /system/bin/gdbserver。

Gdbserver 有两个使用场景。

- attach

如果被调试的程序已经在运行，那么我们就需要“attach”上它。在这种情况下，我们最好能先通过 ps 命令来获取到被测试对象的 PID。比如 attach surfaceflinger，那么先利用 adb shell ps 来找到 surfaceflinger 的进程号，接着在 adb shell 中使用如下命令：

```
gdbserver --attach :5039 [PID]
```

其中“:5039”表示 gdbserver 将在 localhost 的 5039 端口进行监听。

- 启动被调试对象

上述的 attach 用于调试已经在运行的被测对象，而如果希望从头开始调试目标对象，那么可以采用这个场景。以 surfaceflinger 为例，以下命令可以启动一个新的 surfaceflinger 程序来进行调试：

```
gdbserver :5039 /system/bin/surfaceflinger
```

（2）gdb client

当 gdbserver 处于监听状态下时，我们就可以启动 gdb 客户端了。

首先需要切换到 Android 工程的根目录下，然后执行：

```
source /build/envsetup.sh
```

这个步骤是为了保证我们可以正常使用各种函数。

接下来就可以启动 gdbclient 了，下面是调试 surfaceflinger 所用的命令：

```
gdbclient  surfaceflinger  :5039  [PID]
```

也就是对应前面看到的$1,$2,$3 各变量。其中“:5039”和[PID]都需要与 gdbserver 中的保持一致，才能保证正常连接。

一切顺利的话，此时 gdb client 和 gdb server 就已经建立连接了。我们可以通过 gdb 提供的一系列丰富的命令进行各种调试工作。下面是 gdb 提供的官方指导文档，供大家参考查询：

https://sourceware.org/gdb/download/onlinedocs/gdb/index.html