



Underlying Software Development and Practice of Android

# Android 底层 开发实战

周庆国 郑灵翔 康筱彬 刘同山◎编著



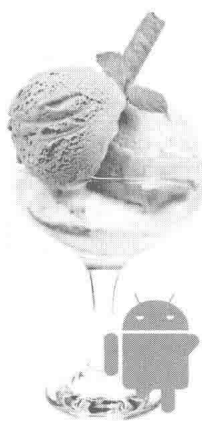
机械工业出版社  
China Machine Press

移动开发

Underlying Software Development and Practice of Android

# Android底层 开发实战

周庆国 郑灵翔 康筱彬 刘同山◎编著



机械工业出版社  
China Machine Press



## 图书在版编目 (CIP) 数据

Android 底层开发实战 / 周庆国等编著. —北京: 机械工业出版社, 2015.9  
(电子与嵌入式系统设计丛书)

ISBN 978-7-111-51611-8

I. A… II. 周… III. 移动终端—应用程序—程序设计 IV. TN929.53

中国版本图书馆 CIP 数据核字 (2015) 第 225084 号

本书主要面向对 Android 设备驱动开发有浓厚兴趣的人员, 无论是具有一定经验的开发人员还是初学者, 都能根据本书所提供的案例进行学习。

书中介绍了嵌入式系统的定义、Linux 系统和 Android 开发环境搭建等基础知识; 分析了 Android 系统底层源码和内核结构, 介绍了 init、Zygote、Binder、Ashmem、Low Memory Killer 和 Logger 等模块; 剖析了系统开发工具 Dalvik、JNI、Boot Loader 的原理和工作方式; 对驱动程序设计中 NDK 的编程方法以及 HAL 层的调用进行了详细阐述, 并通过 Camera 与 WiFi 驱动的实现进行进一步的说明。书中理论部分的介绍旨在呈现一个清晰的开发索引, 同时, 也有利于读者后续进行深度开发。

本书一大特色是具有针对不同架构的实例, 涵盖了当前较流行的 ZedBoard、pcDuino、BeagleBoard 三类开发板。通过对三类开发板实施系统移植、驱动设计、内核跟踪预测 3 个项目, 使得读者对书中所涉及的所有知识都能有更加深入的了解和应用。

## Android 底层开发实战

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

责任编辑: 张国强

责任校对: 董纪丽

印刷: 北京市荣盛彩色印刷有限公司

版次: 2015 年 10 月第 1 版第 1 次印刷

开本: 186mm × 240mm 1/16

印张: 16

书号: ISBN 978-7-111-51611-8

定价: 59.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东



# 推 荐 序

自从2008年谷歌公司发布了Android系统之后，全球便开启了智能机的研究热潮。与此同时，谷歌联合OHA（手机开放联盟）共同推进Android智能机的发展。Android系统的开源及开放等特性，不仅改变了手机移动智能行业，也深深影响了其他科技产业，如嵌入式物联网等，从而掀起了新的一波移动浪潮。

如今市面上介绍Android的书籍比较多，但从底层入手由浅入深地介绍Android开发的比较少。而本书正是从底层的角度出发，引导大家重新审视Android的控件，深入挖掘Android中底层的内容。当然，本书也不是什么大全之作，并没有深入剖析每个知识点，不过应对一般的开发应该足够了。

本书适合有一定编程基础（至少了解C/C++或Java）并对Android开发比较感兴趣的读者。本书首先从Android嵌入式系统及Linux开发常用基础入手，详细介绍Android的系统开发环境及源码结构，并对内核及相关环境库进行讲解，接着讲述相对比较高级的Android驱动开发设计，最后通过一系列开发实例将之前的内容融会贯通，从而让读者循序渐进、由浅入深地学习Android系统的底层开发知识。

学完本书后，读者可以在实际开发中自然而然地体会并运用所学知识，知道如何发现和解决问题以及为什么这样做。希望大家能够从本书中有所收获，从而对工作、学习或者研究起到帮助作用。

谷歌大学合作部安卓培训课程主讲人 郭国勇

2015年8月

# 自序

随着4G网络在全国范围内大规模部署，以及各种无线热点如雨后春笋般在城市遍地开花，甚至在乡村开枝散叶，无线宽带变得越来越普遍，这使得移动设备的应用场景变得更加丰富多彩，人与人之间能够更加方便地随时随地通过多种方式进行沟通，文字、图片已经不再新鲜，视频通信、视频点播也不在话下。智能手机在性能、功能上不断提升，而价格却越来越亲民。如今，在国内大部分人都有一部智能手机。手机犹如钱包、钥匙一样，已经成为每个人随身携带的必备物。智能手机已经不仅仅是一部移动电话，更是一个移动平台，既能当作语音、视频的通信平台，又能当作游戏、娱乐的平台，甚至还可以在手机上处理业务。

如今，传统的互联网已经转向移动互联网，以前还是PC之间的互联，如今变成了移动设备之间的互联。而“万物联网”的概念也已经炒得非常火热，不管是英特尔、三星、高通这样的集成芯片巨头，还是像微软这样的软件巨头，都在推出自己相应的软硬件产品，以趁这个发展势头在物联网时代占得自己的一席之地。将来，所有的“物”都会更加智能，能够更好地与人互动，与其他“物”通信。那时的网络会成为一个人与人、人与物、物与物之间有着万亿连接的庞大网络。

而在物联网时代，移动设备，尤其是智能手机，势必会成为一个核心的接入点，去连接和感知周围环境中的一切。比如通过连接家里的空调、电视机、空气净化器等家用电器，手机会告知甚至自动为用户调节家里的温度和湿度，或控制家里的电视、灯具；通过连接身上携带的传感器并跟踪用户的身体参数，如心跳、呼吸、运动量等，手机可以告知健康状况并给出专业的建议。此外，智能手机也会成为车联网的一大重要接入点，可以远程开启车门，远程反馈汽车的整体状态。总而言之，在万物联网的时代，智能手机会扮演举足轻重的角色。

2007年11月5日，Google推出开放的Android操作系统，同时宣布建立一个全球性的联盟组织，该组织由34家手机制造商、软件开发商、电信运营商以及芯片制造商共同组成。这一联盟将支持Google发布的手机操作系统以及应用软件，将共同开发Android系统的开放源代码。其他公司和个人也可以免费获取源代码并进行SDK的开发。正是因为Android的开放性以及优异的性能，使它得到了众多厂商的支持。前有三星、摩托罗拉、HTC、索尼，后有异军

突起的小米、联想、华为等，纷纷推出了自己定制的Android平台手机。国际研究及顾问机构Gartner公布的最新统计数据表明2014年全球智能手机销售量总计12亿部。另据市场调研公司IDC提供的数据，2014年全年，Android全球智能机所占市场份额为81.5%；其次才是苹果公司的iOS，占14.8%。众星拱月式的拥护，使Android在短短5年内超越iOS以及曾经辉煌一时的塞班系统，顺利雄踞智能手机市场占有率榜首。

不管从功能、性能上，还是从市场占有率及开发者数量上，都说明Android会成为物联网时代一个重要的操作系统。

对于Android底层开发者来说，需要面对复杂多变的应用场景、纷繁的外围设备、不同标准的互联通信，这些无疑都是巨大的挑战。这就要求底层开发者除了掌握软件编程，还要熟悉底层的硬件设备，需要在实践中锻炼自己的技能。目前在市场上也出现了兼容Android操作系统的开源硬件平台，比如BeagleBone、pcDuino等，这对于很多爱好Android的创客来说，是一大利好。就像开源软件Linux和开源硬件Arduino，正是依靠庞大的社区，才使其能够在全世界得到飞快的传播。庞大的社区，不管是对于开发者本身的学习和开发过程，还是对于Android的发展，都会带来积极的促进作用。

在编写本书的过程中，我们与创客社团和Google开发者社区（GDG）的成员沟通时发现，无论是初学者还是开发者，都偏重于软件而缺乏硬件基础，所以非常希望更深入地了解底层的硬件。而且，兼容Android的外围设备越来越多，也迫使底层开发人员去掌握基本的硬件知识和底层驱动开发。

因此，在Google大学合作部的大力支持下，几位从事嵌入式开发的教师合作编写了本书。本书在编写过程中也融入了几位老师的研究课题以及教学经验，希望能对从事和学习Android底层开发的人员有所帮助。

编者

2015年8月

# 前 言

自第一款搭载Android系统的智能手机HTC G1发布至今已近6年，凭借日益完善与强大的功能、完全开放的内核源码以及Google公司在网络应用领域的无缝支持，Android系统从初出茅庐的行业新锐，已经成长壮大为手持设备智能化产业的推动者和市场的领导者，这一点已经毋庸置疑。

不可否认，众多独立软件开发者与商业软件公司的加入使Android系统不仅仅作为终端产品，还作为开发平台深入到各行业的技术开发中，包括学校、公司，乃至开发人员的工作、生活的各方面，这是Android系统旺盛生命力的表现，同时也是Android系统持续发展壮大的保障。近期，支持Android系统的开发板如Zynq、pcDuino等在功能不断完善的前提下成本也在大幅下降，助力了Android的推广与拓展；可穿戴设备以及嵌入式芯片的应用又将为Android系统及其软硬件开发提供新的发展机遇。接触Android领域、使用Android产品、学习Android系统，现在已经成为计算机软件、电子、自动化控制及相关专业的学生和已经参加工作的软硬件工程师的首选。

但是目前市场上同类的Android相关书籍中，多是介绍基于Android SDK的单纯应用程序（APP）的开发，且种类繁杂，对于Android的底层源码与系统内核的分析、驱动程序设计与存储优化、平台移植与内核测试等较为深入的内容则甚少涉及。为了对这些底层领域相关知识稍作弥补，作者编写了本书。

本书而用浅显易懂的语言向广大Android爱好者和开发人员讲解Android系统下嵌入式开发板的设计。

全书共8章，前3章为预备知识，简要介绍嵌入式系统的定义与软硬件开发以及Android开发环境的搭建。这一部分为基础知识，有一定Linux基础和Android开发经验的读者可以选读。

第4~6章为系统结构，主要介绍Android系统的源码结构、内核与相关工具以及环境库。内容包括：Android源码结构、init初始化脚本、Zygote、Android系统编译；Android内核启动、Binder框架、Ashmem内存管理、系统日志Logger实现；Android开发工具、Dalvik虚拟机、JNI、Boot Loader。

第7章为驱动设计，主要介绍Android系统中常用外接设备的驱动架构以及实现。内容包括NDK编程、Android中HAL模型架构与实现流程。作为教学实例，最后还分析了Android系统中的Camera与WiFi两个功能部件的驱动设计。

第8章为实例分析，介绍具体开发板硬件结构的编程原理以及系统级的实例。在简要介绍系统底层开发流程的基础上，首先分析主流的Zynq和pcDuino平台上开发环境的搭建、Linux内核以及Android系统的编译、下载；然后结合前文学习过的知识点，从零开始设计LED显示系统的Linux内核驱动、Android HAL支持和服务层设计、顶层App的实现；作为进阶部分，最后介绍了对Android内核进行跟踪调试与性能测试的工具软件、工作流程以及结果分析。

## 致谢

在此要特别感谢华章公司的策划编辑张国强，是他对Android和嵌入式系统开发的关注促成了本书的出版。笔者在撰写书稿时，他也对本书提出了宝贵的写作建议，并进行了仔细的审阅。

本书的编写同样离不开许多朋友的支持，在此特别感谢兰州大学信息科学与工程学院的陈华明的大力协助，这本书的出版离不开他的贡献。

还要感谢兰州大学的高博、郭守超、王小强和朱芳芳，感谢他们在本书编写过程中所给予的帮助与建议。

# 目 录

推荐序

自 序

前 言

## 第1章 Android嵌入式系统导论 ..... 1

### 1.1 Android嵌入式系统概述 ..... 1

#### 1.1.1 嵌入式系统定义 ..... 1

#### 1.1.2 基于Android的嵌入式系统 构成 ..... 5

#### 1.1.3 移动电话系统 ..... 9

#### 1.1.4 基于ARM的移动电话硬件 结构 ..... 10

### 1.2 嵌入式系统实例 ..... 17

#### 1.2.1 pcDuino部分硬件功能介绍 ..... 17

#### 1.2.2 基于Android的嵌入式系统 ..... 30

## 第2章 Linux系统详解 ..... 34

### 2.1 系统简介 ..... 34

### 2.2 基础命令 ..... 35

#### 2.2.1 cd和ls命令 ..... 35

#### 2.2.2 touch和mkdir命令 ..... 37

#### 2.2.3 rm和rmdir命令 ..... 38

#### 2.2.4 cp和mv命令 ..... 38

#### 2.2.5 find和awk命令 ..... 39

#### 2.2.6 vim编辑器的使用 ..... 43

### 2.3 Bash Shell ..... 44

#### 2.3.1 Bash Shell简介 ..... 44

#### 2.3.2 Bash Shell脚本简介 ..... 44

### 2.4 Linux源码与Android源码介绍 ..... 45

#### 2.4.1 Linux源码简介 ..... 45

#### 2.4.2 Android源码简介 ..... 45

## 第3章 Android系统开发环境 搭建 ..... 47

### 3.1 编译前奏——Android上的开发 工作 ..... 47

#### 3.1.1 Android的移植开发 ..... 47

#### 3.1.2 系统开发 ..... 48

#### 3.1.3 应用开发 ..... 49

### 3.2 Android的系统架构 ..... 49

#### 3.2.1 软件结构 ..... 49

#### 3.2.2 源代码的结构 ..... 51

### 3.3 搭建开发环境 ..... 54

#### 3.3.1 搭建编译环境 ..... 54

#### 3.3.2 使用repo ..... 64

#### 3.3.3 Android的编译 ..... 66

## 第4章 Android系统底层源码结构 分析 ..... 69

### 4.1 源码结构分析 ..... 69



4.1.1 底层库结构介绍 .....	71	5.4.3 Binder的机制和原理 .....	125
4.1.2 C基础函数库bionic .....	73	5.5 Ashmem内存管理方式 .....	128
4.1.3 C语言底层库libcutils .....	74	5.5.1 概述 .....	128
4.1.4 C++工具库libutils .....	74	5.5.2 Ashmem初始化 .....	128
4.1.5 底层文件系统库system .....	75	5.5.3 内存的创建和释放 .....	131
4.1.6 增加本地库的方法 .....	76	5.5.4 内存的映射 .....	135
4.2 Android编译系统介绍 .....	78	5.5.5 内存的锁定和解锁 .....	137
4.2.1 build系统 .....	78	5.6 低内存管理 .....	139
4.2.2 SDK .....	79	5.7 Logger .....	145
4.3 init初始化脚本语言介绍 .....	82	5.7.1 Logger概述 .....	145
4.3.1 概述 .....	82	5.7.2 Logger实现原理 .....	146
4.3.2 init进程源码分析 .....	82		
4.3.3 脚本文件的创建与分析 .....	85	<b>第6章 Android系统相关工具及</b>	
4.3.4 创建设备节点文件 .....	89	<b>运行环境 .....</b>	<b>156</b>
4.3.5 子进程的创建与终止 .....	92	6.1 Android开发工具分类及介绍 .....	156
4.3.6 属性服务 .....	93	6.1.1 应用程序开发工具 .....	156
4.4 Zygote .....	95	6.1.2 框架开发工具 .....	157
4.4.1 Zygote概述 .....	95	6.1.3 交叉编译工具 .....	159
4.4.2 AppRuntime分析 .....	95	6.1.4 内核开发工具 .....	159
4.4.3 system_server分析 .....	100	6.2 Dalvik虚拟机 .....	160
<b>第5章 Android系统内核分析 .....</b>	<b>109</b>	6.2.1 概述 .....	160
5.1 Linux内核基础 .....	109	6.2.2 dex文件 .....	163
5.1.1 概述 .....	109	6.2.3 Dalvik内存管理 .....	165
5.1.2 Linux内核的主要子系统 .....	110	6.2.4 Dalvik编译器 .....	169
5.1.3 Linux启动过程分析 .....	111	6.3 JNI .....	171
5.2 Android内核概况 .....	112	6.3.1 概述 .....	171
5.3 Android启动过程分析 .....	113	6.3.2 JNI的架构 .....	171
5.4 Binder框架分析 .....	115	6.3.3 JNI的实现方式 .....	172
5.4.1 概述 .....	116	6.4 Boot Loader .....	173
5.4.2 Binder的系统架构 .....	117	6.4.1 概述 .....	173
		6.4.2 Boot Loader 的操作模式 .....	174

6.4.3 启动过程 .....	175	8.2.4 SD卡的准备以及Android 系统的启动 .....	205
6.5 busybox的使用 .....	176	8.3 移植讲解——基于pcDuino的 Android移植 .....	205
<b>第7章 Android驱动程序设计</b> .....	177	8.3.1 pcDuino介绍 .....	206
7.1 Android驱动概述 .....	177	8.3.2 环境搭建 .....	208
7.2 Android NDK编程 .....	178	8.3.3 编译内核 .....	209
7.3 Android系统中的HAL层 .....	183	8.3.4 编译Android .....	210
7.3.1 HAL_legacy和HAL对比 .....	184	8.3.5 烧录镜像 .....	211
7.3.2 HAL module 架构分析 .....	184	8.4 Android LED驱动设计 .....	212
7.3.3 HAL实现流程 .....	186	8.4.1 硬件原理 .....	212
7.4 Android系统Camera与WiFi实现 .....	187	8.4.2 Linux驱动设计 .....	212
7.4.1 Android中的Camera实现 .....	187	8.4.3 Android HAL层驱动 .....	220
7.4.2 Android系统WiFi实现 .....	190	8.4.4 硬件服务层 .....	224
<b>第8章 Android底层开发实例 讲解</b> .....	193	8.4.5 App应用编写 .....	229
8.1 底层开发相关技术概览 .....	193	8.5 进阶讲解——针对Android系统 的内核跟踪与测试 .....	231
8.2 实例讲解——基于Zynq的 Android移植 .....	193	8.5.1 使用平台简介 .....	231
8.2.1 主机开发环境的搭建 .....	194	8.5.2 测试环境的建立 .....	232
8.2.2 Linux内核的编译 .....	194	8.5.3 测试工具 .....	235
8.2.3 Android文件系统的编译 .....	203	8.5.4 Android内核调试与性能 测试 .....	239

# 第 1 章

## Android 嵌入式系统导论

本章主要围绕 ARM 嵌入式系统进行总体讲解，并对相应的硬件系统和软件系统进行讲解。

### 1.1 Android 嵌入式系统概述

#### 1.1.1 嵌入式系统定义

Android 是一款以 Linux 为基础的开源移动设备操作系统，一直由 Google 公司领导和开发。Google 对于 Android 系统所持有的开放态度，令 Android 系统一经发布就风靡全球。Google 于 2007 年 11 月 5 日正式发布 Android 系统，该平台由操作系统、中间件、用户界面和应用软件组成，是首个为移动终端打造的真正开放的和完整的移动设备软件。2012 年 11 月数据显示，Android 占据全球智能手机操作系统市场的 76%，在中国市场的占有率为 90%。据 2013 年数据显示，全世界采用这款系统的设备数量已经达到 10 亿台。

这里所说的 Android 系统，是基于 Linux 内核发展起来的嵌入式操作系统。Android 系统基于 Linux 内核所做的改进将在后续章节进行详细介绍。

本章将详细介绍嵌入式系统，主要涉及 Android 系统所运行的平台、处理器芯片以及相关的技术。

#### 1. 什么是嵌入式系统

嵌入式系统 (Embedded System) 是一种完全嵌入受控器件内部、为特定应用而设计的专用计算机系统。嵌入式系统的本质就是计算机系统，因而它也是由软件以及硬件构成的。与普通计算机不同，嵌入式系统通常仅拥有非常有限的硬件资源，这种配置使它们的成本大幅下降，但也对软件的优化提出更高的要求。嵌入式系统一般运行固定的程序或固定的操作系统，再加上可变的应用程序。有些为工业系统，仅用于某个特定的控制目的；有些因为有应用程序的加入而更显灵活，一般用于手机、平板电脑。如图 1-1 所示。

第一个被大家认可的现代嵌入式系统是麻省理工学院仪器研究室的查尔斯·斯塔克·德雷珀开发的阿波罗导航计算机。在两次月球飞行中，太空驾驶舱和月球登陆舱都使用了这种

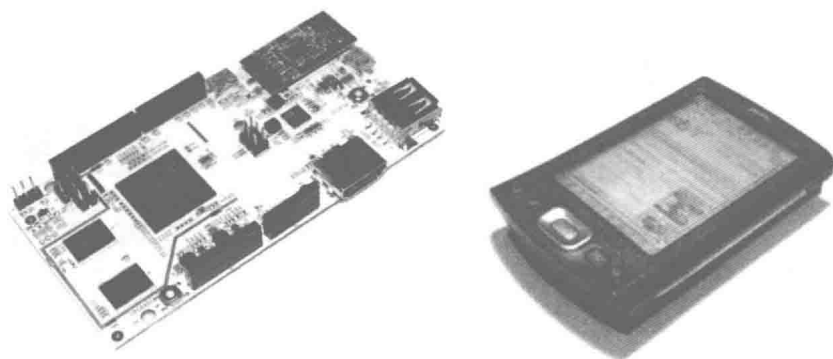


图 1-1 典型的嵌入式系统

惯性导航系统。在计划刚开始的时候，阿波罗导航计算机被认为是阿波罗计划中风险最大的部分。为了减小尺寸和重量而使用的当时最新的单片集成电路，这更加大了阿波罗计划的风险。第一款大批量生产的嵌入式系统是美国军方于 1961 年发布的民兵 I 导弹上的 D-17 自动导航控制计算机。它是由独立的晶体管逻辑电路构成的，并带有一个作为主内存的硬盘。当民兵 II 导弹在 1966 年开始生产的时候，D-17 由第一次使用大量集成电路的更新计算机所替代。仅这个项目就将与非门集成电路模块的价格从每个 1000 美元降低到了每个 3 美元，使集成电路的商用成为可能。民兵导弹的嵌入式计算机有一个重要的设计特性：它能够在项目后期对制导算法重新编程以获得更高的制导精度，并且能够使用计算机测试导弹，从而减轻测试所用的电缆和接头的重量。这些 20 世纪 60 年代的早期应用使嵌入式系统得到了长足发展，它的价格开始下降，同时处理能力和功能获得了巨大的提高。

英特尔 4004 是第一款微处理器，它在计算器和其他小型系统中找到了用武之地。但是，它仍然需要外部存储设备和外部支持芯片。1978 年，美国国家工程制造商协会（NEMA）发布了可编程单片机的“标准”，包括几乎所有以计算机为基础的控制器，如单片机、数控设备，以及基于事件的控制器。随着单片机和微处理器价格的下降，使一些消费性产品用单片机的数字电路取代昂贵模拟组件成为可能。到了 20 世纪 80 年代中期，许多以前是外部系统的组件被集成到了处理器芯片中，这种结构的微处理器得到了更广泛的应用。到了 20 世纪 80 年代末期，微处理器已经出现在几乎所有的电子设备中。

现代的嵌入式系统一般分为简单嵌入式系统和复杂嵌入式系统。简单嵌入式系统一般被认为是由单片集成控制器作为硬件核心的嵌入式系统，其核心只有一片芯片，却集成了处理器、闪存、内存、数字和模拟外设这些设备，这样的系统开发难度低。然而，由于种种限制，其性能一般，仅适合于自动化、运动控制、电源控制等简单的控制类应用。与之相反，复杂嵌入式系统一般由独立的处理器和闪存构成，处理器本身不集成大量的外设，仅执行处理任务，类似于传统计算机的 CPU。这样的系统灵活多变，性能优异，但是成本高昂，普遍用于人机接口、智能设备、手机等性能要求高的场合，如图 1-2 所示。

以下是一些嵌入式系统的典型应用：

- ATM 取款机
- 航空制导系统
- 交换机、路由器、ADSL 终端
- 可穿戴传感器
- 计算机硬盘
- 工业控制器
- 计算器
- 机顶盒
- MP3、MP4
- GPS 导航仪
- 手机、平板电脑
- 智能测试测量仪器仪表

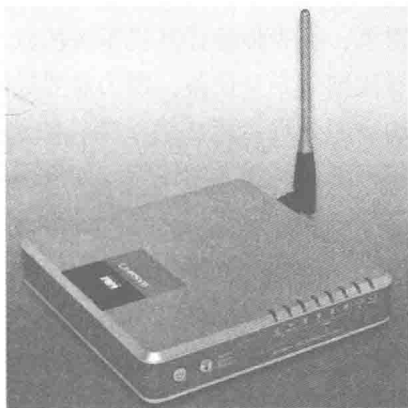


图 1-2 嵌入式系统应用

## 2. 为什么选择嵌入式系统

很多人觉得没有接触过嵌入式系统，嵌入式系统离他们很遥远。其实，嵌入式系统无处不在。从白色家电到大型网络系统，嵌入式系统时时刻刻为我们服务着。与传统计算机比较，嵌入式系统尽管有开发难度大、通用性差和人机接口普遍落后等劣势，却有着传统计算机所没有的关键优势。

嵌入式系统可以做到极低的成本。一般来说，用于简单工业控制和白色家电的单片芯片集成了复杂的模拟外设、数字外设，而且不用外界任何存储设备，如图 1-3 所示。这样的一片芯片往往售价不超过 10 元人民币，更加令人惊奇的是，这种芯片可以在很宽的电源电压范围内工作，又有良好的可靠性，从而进一步降低了对外部环境的要求，使微电脑控制技术得以广泛普及。

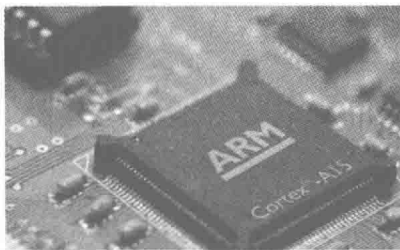


图 1-3 嵌入式系统处理器

嵌入式系统极其可靠。一般说来，系统中串联工作的部件越多，系统的可靠性越差；系统中并联工作的部件越多，系统的可靠性越好。这里的串联指的是相互依赖的工作方式，并联指的是互为冗余的工作方式。嵌入式系统往往有更精简的结构，从而使其有更少的出错机会。从硬件上讲，各模块之间的依赖关系更加清晰，模块数量精简，从而使串联部件减少。从软件上讲，由于使用了定制的操作系统和应用程序，甚至没有操作系统，从而使得软件组件大幅减少，也减少了串联部件。而且，由于结构精简，从而可以留出更多成本预算来做冗余。这样，在减少串联部件的同时增加并联部件，使得嵌入式系统可以提供传统计算机系统所难以比拟的高可靠性。

嵌入式系统极其高效。虽然绝大多数的嵌入式系统拥有较差的计算资源，它们通常仍能完成任务。与通用计算机不同，嵌入式系统的硬件、软件都可以根据实际需求而加以定制，

这使得系统得以精简。除了提高系统的可靠性，精简的系统还能减少硬件资源，尤其是 CPU 资源和内存资源的浪费。一台主频只有 16MHz 的计算机是什么样子？然而，多数用于工业控制的单片机的主频不超过 16MHz，却能井井有条地控制大型机械设备。对于某些超高性能需求，因为传统计算机系统的低效、臃肿，嵌入式系统是唯一的选择。比如说大型电信路由器或网络安全设备，因为需要处理大量的数据请求，还不能有太多的网络延迟或丢包，高效的专用处理器就成为了唯一的选择。它们被制成功能单一的芯片，只能做一种简单的任务，却有着极其强大的性能。如果采用通用计算机来支撑网络社会，可能会因为成本过高而无法实现。

嵌入式系统体积小、功耗低。手机、平板电脑都是嵌入式系统，却也有着不逊于传统计算机系统的性能。一般来说，嵌入式系统的功耗不会大于 20 瓦。多数的单片机功耗在几十毫瓦左右，而多数复杂嵌入式系统的功耗不过几百毫瓦。即使是最先进的手机系统，其峰值功耗不过二三瓦，而其平均功耗不过二三百毫瓦或更低。反观传统计算机，即使是最省电的笔记本电脑也要消耗数十瓦的功率。在减小功耗的同时，嵌入式系统的散热问题也随之消失，更简单的电源管理和几乎不使用散热装置，使得嵌入式系统的体积更小，如图 1-4 所示。

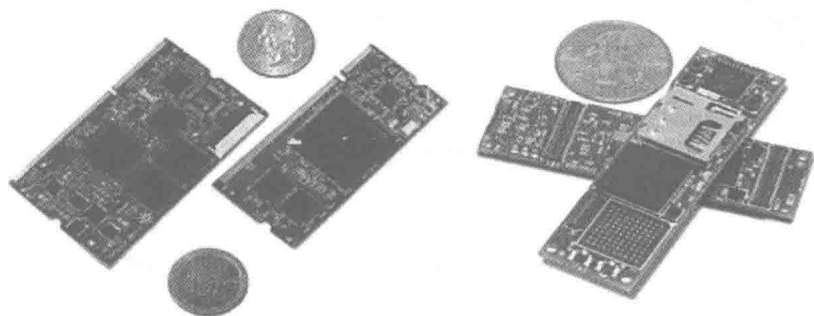


图 1-4 典型的嵌入式系统体积

### 3. 如何选择

如前文所述，嵌入式系统可分成简单嵌入式系统和复杂嵌入式系统。简单嵌入式系统一般为单片机，比如 MCS51 系列、PIC 系列、AVR 系列和新兴的 MSP430 系列。这些单片机成本低廉，外设丰富，而且在上电后可以立即运行，适合于对性能要求不高的控制类应用，如智能仪表、电机控制、可穿戴传感器和数字电源等。

复杂嵌入式系统的构成则要复杂得多，其性能、成本也高得多。一般来说，该类系统包括基于 DSP 的嵌入式系统、基于 ARM 的嵌入式系统、基于 MIPS 的嵌入式系统和基于 x86 的嵌入式系统。基于 DSP 的嵌入式系统一般用于处理大量的数据，典型应用为语音处理、雷达信号处理等。基于 MIPS 的嵌入式系统一般应用于通用计算，因为 MIPS 在开发之初就被用于通用处理器。基于 x86 的嵌入式系统实际上就是把传统的计算机压缩、精简，一般因其强大的性能与兼容性被用于对成本、功耗要求不高的场合，比如工业计算机等。

复杂嵌入式系统中最常见的要属基于 ARM 的嵌入式系统（以下简称 ARM 系统）。ARM

系统通常拥有足够的硬件资源和相对较低的功耗,这使得 ARM 处理器非常适用于复杂的工业环境和移动终端,这当中绝大部分就基于 Android 操作系统。

### 1.1.2 基于 Android 的嵌入式系统构成

#### 1. 嵌入式系统架构概述

嵌入式系统由软件和硬件构成。硬件部分包括微处理器、内存、闪存/硬盘、外部设备等;软件部分则包括引导器、操作系统、文件系统、用户程序等。对于简单嵌入式系统来说,硬件部分通常只是一块单片机芯片,集成了所需的全部硬件资源,而软件部分则只是一个单一的二进制文件,包含全部的代码、常数、存储分配结构等。本书的研究对象是基于 ARM 处理器和 Android 操作系统的复杂嵌入式系统。

#### 2. Android 系统的发展与版本演变

Android 一词的本义指“机器人”,该平台由操作系统、中间件、用户界面和应用软件组成。Android 操作系统最初由 Andy Rubin 开发完成,是一种基于 Linux 内核的自由及开放源代码的操作系统,主要应用于移动设备,如智能手机和平板电脑。2005 年 8 月由 Google 收购并注资。2007 年 11 月,Google 与 34 家硬件制造商、软件开发商及电信运营商组建开放手机联盟,共同研发改进 Android 系统,并且以 Apache 开源许可证的授权方式,发布了 Android 的源代码。第一部 Android 智能手机 HTC G1 发布于 2008 年 10 月。随后 Android 操作系统逐渐扩展到平板电脑及其他领域,如电视、数码相机、游戏机等。2011 年第一季度,Android 在全球的市场份额首次超过塞班系统,跃居第一。

2008 年,在 Google I/O 大会上,Google 提出了 Android HAL 架构图,在同年 8 月 18 日,Android 获得了美国联邦通信委员会的批准。在 2008 年 9 月,Google 正式发布了 Android 1.0 系统,这也是 Android 系统最早的版本。

2009 年 4 月,Google 发布了 Android 1.5 版本,并开始以甜品的名字对各个版本进行命名,Android 1.5 命名为 Cupcake。该版本与 1.0 版相比有了很大的改进。

2009 年 9 月,Google 发布了 Android 1.6 的正式版,并且推出了搭载 Android 1.6 正式版的 HTC G3 手机。凭借着出色的外观设计以及全新的 Android 1.6 操作系统,HTC G3 成为当时全球最受欢迎的手机。Android 1.6 也有一个有趣的甜品名称,即 Donut。

2010 年 2 月,Linux 内核开发者 Greg Kroah-Hartman 将 Android 的驱动程序从 Linux 内核 staging tree 上除去。从此,Android 与 Linux 开发主流分道扬镳。同年 5 月,Google 正式发布了 Android 2.2 操作系统,并将其命名为 Froyo。

2010 年 10 月,Google 宣布 Android 系统达到了第一个里程碑,即在电子市场上获得官方数字认证的 Android 应用数量已经达到了 10 万个,Android 系统的应用增长非常迅速。在 2010 年 12 月,Google 正式发布了 Android 2.3 操作系统 Gingerbread。

2011 年 1 月,Google 宣布每日新增的 Android 设备用户数达到 30 万部,到 2011 年 7 月,

这个数字增长到 55 万部，Android 系统设备的用户总数更是达到了 1.35 亿。Android 系统已经成为智能手机市场占有率最高的系统。

2011 年 8 月 2 日，Android 手机已占据全球智能机市场 48% 的份额，并在亚太地区市场占据统治地位，终结了塞班系统的霸主地位，跃居全球第一。

2011 年 9 月份，Android 系统的应用数目已经达到了 48 万，而在智能手机市场，Android 系统的占有率已经达到了 43%，继续排在移动操作系统首位。全新的 Android 4.0 操作系统于 2011 年 9 月 23 日发布，这款系统被 Google 命名为 Ice Cream Sandwich。

2012 年 1 月 6 日，Google Android Market 已有 10 万开发者推出超过 40 万活跃的应用，大多数的应用程序为免费的。Android Market 应用程序商店目录在新年首个周末突破 40 万基准，距离突破 30 万仅用了一个月。在 2011 年早些时候，Android Market 应用从 20 万增加到 30 万花了 4 个月。

2013 年 9 月，Google 证实，已有超过 10 亿 Android 设备被激活，应用下载次数超过 480 亿次。Android 系统不仅支持手机和平板电脑，还成功进军相机、电视、手表甚至是冰箱等消费品领域。同年 10 月发布了 Android 4.4，代号为 KitKat（奇巧巧克力）。新的 Android 4.4 操作系统为开发者提供了两种编译模式，一种仍是默认的 Dalvik 模式，另外一种则是 ART 模式。ART 模式发行的应用在用户安装时就进行预编译操作，将原本在程序运行中的编译动作提前到应用安装时，在省去解释代码这一过程之后，应用的运行效率更高。Android 被人所诟病的虚拟机解释编译时代一去不复返。

2014 年，随着更多的手机生产商的加入，Android 智能手机总出货量为 13 亿台，这意味着 Android 智能手机占全年手机销售量的 81.2%。

2014 年 10 月 16 日，Google 正式发布 Android 5.0，并将系统镜像发布到 Nexus 4、5、7 和 10 的设备上。主要更新包括：支持 64 位处理器，支持蓝牙 4.1，采用全新的 Material Design 界面，全面由 Dalvik 虚拟机转用 Android RunTime（ART）编译虚拟机等。

2015 年 4 月 22 日，Google 发布最新版本 Android 5.1.1，在 Android 5.0 基础上主要增加了新功能和完全性，例如多 SIM 卡和高清语音的支持、快速设置 WiFi 和蓝牙等

### 3. Android 系统的优势

#### （1）开放性

Android 平台最大的优势就是其开放性。开放的平台允许任何移动终端厂商加入 Android 联盟中来。显著的开放性可以使其拥有更多的开发者，随着用户数量的增多和应用的日益丰富，一个崭新的平台也将很快走向成熟。

#### （2）运营商的鼎力支持

在国内，三大运营商卯足了力量推出 Android 智能机。联通的“0 元购机”，电信的千元 3G，移动的索爱 A8i 定制机等，都显示了运营商对 Android 智能机的期望。在美国，T-Mobile、Sprint、AT&T 和 Verizon 全部推出了 Android 手机。此外，KDDI 及 NTT DoCoMo（日本）、



Telecom Italia (意大利电信)、T-Mobile (德国)、Telefónica (西班牙) 等众多运营商都是 Android 的支持者。有这么多的运营商支持 Android, 自然会使它占据巨大的市场份额。相对于 Android 的运营商联盟, 只有 AT&T 一家运营商销售 iPhone。

### (3) 丰富的硬件选择

基于 Android 平台的开放性, 众多厂商推出了多种产品。这些产品虽然功能上各有特色, 却不会影响数据同步, 以及软件的兼容。例如将一款诺基亚塞班风格的手机更换为苹果 iPhone, 不仅可以将塞班中优秀的软件移植到 iPhone 上继续使用, 而且联系人等资料也可以方便地转移。现在, 世界绝大部分智能手机厂商都加入了 Android 阵营, 并推出了一系列的 Android 智能机。摩托罗拉、三星、HTC、LG、Lumigon 等厂商都与 Google 建立了 Android 平台技术联盟。加盟的厂商越多, 手机终端就越多, 其市场潜力就越大。Android 智能机最近 6 个月在美国市场的占有率足以说明这一点。

### (4) 开发商不受任何限制

随着 Android 的推广与普及, 应用程序的数量呈指数级增长, Android 应用在可预见的未来是有能力与苹果竞争的。而来自 Android 应用商店最大的优势是, 不对应用程序进行严格的审查, 开发不受限制。

### (5) 无缝结合 Google 的应用

如今互联网的 Google 已经走过 10 年的历程, 从搜索巨人发展到全面的互联网渗透, Google 服务如地图、邮件、搜索等已经成为连接用户和互联网的重要纽带, Android 平台则将这些优秀的 Google 服务无缝结合到系统中。

## 4. 基于 Android 的嵌入式系统硬件构成

为了支持 Android 系统的正常运行, 一个典型的系统应具备 ARM 嵌入式处理器、内存、闪存、电源管理系统、音频子系统、显示屏和触摸屏。

图 1-5 以基于全智科技 A10 应用处理器的嵌入式系统为例介绍了一个无线 IP 电话硬件系统的构成。Android 操作系统运行在基于 ARM 核心的 A10 处理器上, GPU 部分运行图像加速算法。ARM 处理器控制片上外设, 片上外设控制系统中的其他设备, 比如音频控制器、背光控制器等。A10 SOC 通过 USB 等接口对外实现有线和无线通信。

## 5. Android 嵌入式系统软件构成

嵌入式系统软件包含引导器、操作系统、文件系统和用户程序等, 如图 1-6 所示。以 Android 系统为例, 系统引导器 (通常是 U-Boot) 在系统通电之后首先运行, 该程序负责处理器、内存、闪存的初始化, 对系统内核实行解压缩操作, 然后将控制权交给操作系统。操作系统包含内核和用户态程序。Android 系统的内核是 Linux, Linux 在加载完成后挂载文件系统, 并从中加载用户态程序。Android 的用户态程序包含启动管理、Java 虚拟机、系统库函数等。在完成 Android 系统的加载后, Android 会自动加载默认的桌面应用程序。至此, Android 系统完成启动过程。

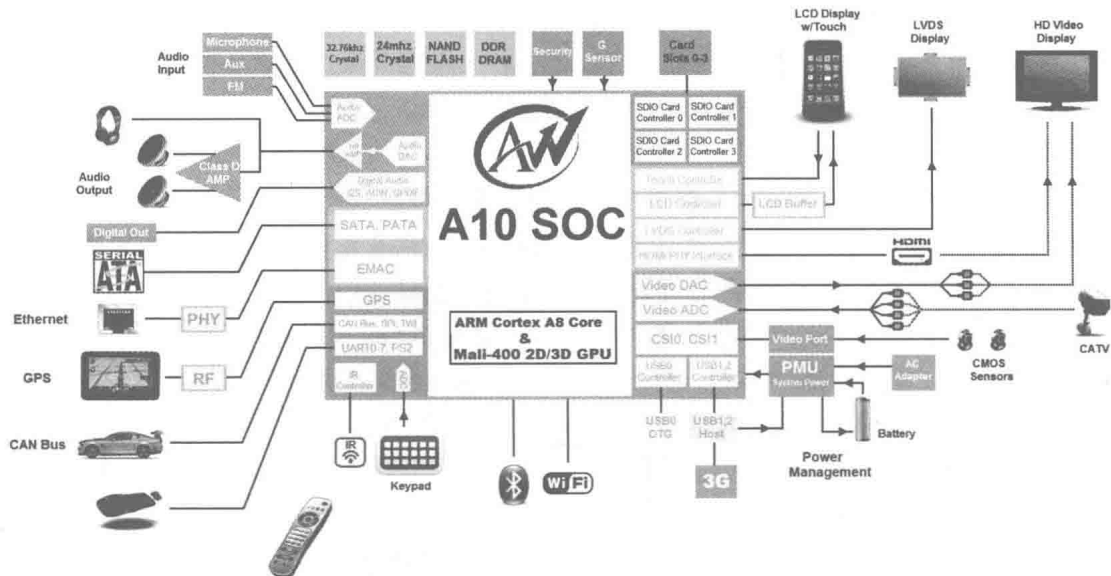


图 1-5 Android 嵌入式系统的硬件

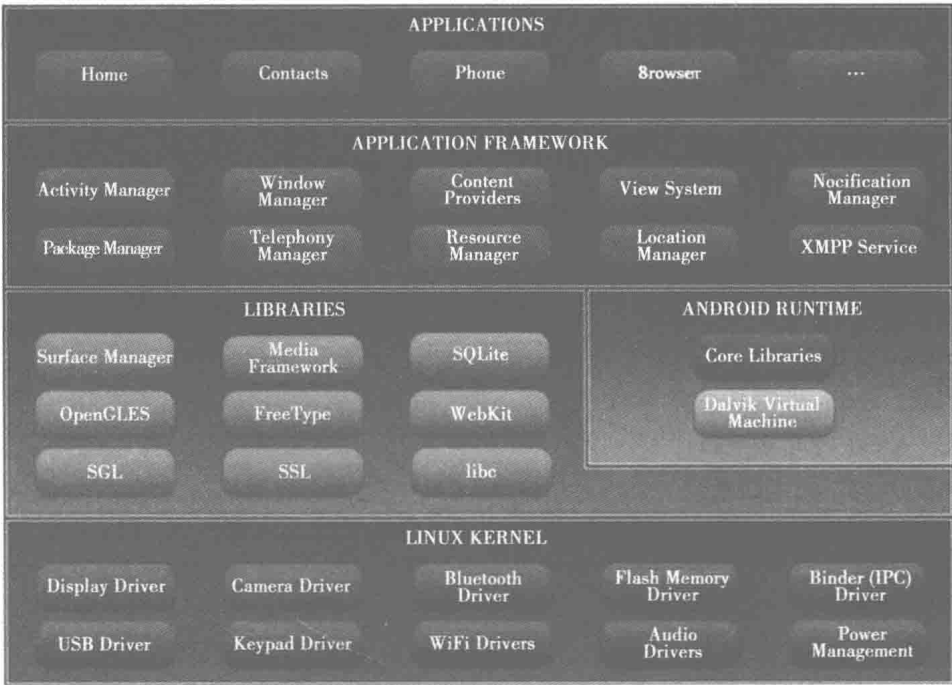


图 1-6 Android 系统软件构成

引导器包含设备相关的、与底层打交道的代码，这部分常用汇编语言和 C 语言编写。Linux 内核部分包含部分设备相关代码，比如 CPU 的优化指令等，以及大量的外设驱动。这部分将是移植 Android 系统的重点。Android 虚拟机和 Linux 用户态库文件提供应用程序与

Linux 内核沟通的机制,并实现大量的扩展功能。如果要开发自定义的 Android 系统,就少不了开发指定的用户态库文件。如果 Android 系统是用于特定目的,一般还要为应用程序(APK)开发对应的底层库文件及 JNI 库。这也是 Android 系统移植的重点。最高层是应用程序框架,该部分提供应用程序可以调用的 Java 包,它们实现了多数的 Android API,是应用开发的重点。最上层是应用程序,该部分由第三方开发,与系统移植无关。

### 1.1.3 移动电话系统

#### 1. 移动开发

移动开发也称为手机开发,或叫做移动互联网开发,是指以手机、PDA、UMPC 等便携终端设备为基础进行相应的开发工作。由于这些随身设备基本都采用无线上网的方式,因此,业内也称作为无线开发。

移动应用开发类似于 Web 应用开发,是为小型无线计算设备编写软件的流程和程序的集合,起源于更为传统的软件开发,其独特之处在于通常根据一个具体移动设备所提供的性能编写软件。

#### 2. 未来移动电话的功能

根据目前科学技术的发展,未来的移动电话可能包含如下功能:

##### (1) 更智能的语音接口

未来手机会大范围应用语音功能。不同于现在的语音检索,未来的语音功能会更加完善,功能更多,识别更准确,而且可以与用户的生活融为一体。

##### (2) 互动投影

如果想把手机做得更小,又能提供丰富的娱乐体验,那就只能把触摸屏幕换成互动投影仪,这种变更会使得手机变得更小,而显示面积却变得更大。

##### (3) 云计算

受电池和发热的限制,手机很难拥有非常豪华的硬件配置。随着超高带宽的 WiFi 和 4G 的普及,软件开发者可以把复杂但是带宽略低的应用运行在服务器上,而手机只负责最终的用户界面。

##### (4) 新型显示屏

影响手机厚度的因素之一就是屏幕。如果屏幕可以是简单的一块玻璃或者聚合物,那么手机的整体厚度就会大幅减少。随着柔性显示技术的发展,可以弯折的屏幕将进一步扩大显示面积,降低重量。

##### (5) 高级图像技术

现在有越来越多的数码相机提供 WiFi 分享等功能。手机也可以集成高性能的图像传感器,凭借手机的处理能力和联网能力,实现对动态图像捕捉、处理和分享。现在集成相机级别的摄像功能的手机已经问世。

### (6) 模块化技术

在购买手机时，常常会因为手机的某个功能不够强大而考虑其他的型号。模块化的设计使得手机最核心的部件以外的大多数部件都可以由用户自行更换。这种设计类似 PC 机，用户可以买来选配的模块来升级系统或为系统添加新的功能。

## 1.1.4 基于 ARM 的移动电话硬件结构

### 1. ARM 处理器简介

ARM 架构是 32 位元精简指令集中央处理器架构，广泛地使用在许多嵌入式系统设计中。由于节能这一特点，ARM 处理器非常适合应用于移动通信领域，符合移动设备低功耗的设计目标。

截至 2009 年，ARM 架构处理器占据了市面上 32 位嵌入式 RISC 处理器 90% 的份额。ARM 处理器可以在很多消费类电子产品中看到，从便携式设备（PDA、移动电话、多媒体播放器、掌上型电子游戏和计算机）到电脑外设（硬盘、桌面型路由器），甚至在导弹的弹载计算机等军用设施中都有应用。同时还有一些基于 ARM 设计的派生产品，例如 Marvell 的 XScale 架构和德州仪器的 OMAP 系列。

2011 年，ARM 的客户报告了 79 亿 ARM 处理器出货量，占有了当年智能手机市场的 95%，硬盘驱动器市场的 90%，数字电视和机顶盒市场的 40%，单片机市场的 15% 和移动电脑市场的 20%。2012 年，微软与 ARM 科技生产了新的 Surface 平板电脑，同年，AMD 宣布将于 2014 年开始生产基于 ARM 核心的 64 位服务器芯片。

现行的（基于 ARMv6、7、8 架构）ARM 处理器分为以下 3 类别：

- 用于应用处理器的 Cortex-A 系列
- 用于实时控制器的 Cortex-R 系列
- 用于高级单片机的 Cortex-M 系列

ARM 处理器支持多核计算，可以有多个核同时执行不同的指令，从而加快系统运行的速度。常见的多核系统有双核 Cortex A9、四核 Cortex A15 等。同时，ARM 也支持异构多核，即多个不同的 ARM 核所构成的多核系统，通常是由 Cortex-A7 和 Cortex-A15 构成的面向低功耗、高性能的异构多核系统。最近，TI 公司推出了基于双核 Cortex-R4 的 Hercules 系列实时处理器。

在任何时刻，CPU 只能运行在一种模式，但由于外部事件（中断）或编程方式，能够切换到其他的模式。CPU ARM 架构指定了以下的 CPU 模式：

- 用户模式，唯一的非特权模式。
- 系统模式，通过非异常的方法进入的特权模式。
- SVC 模式，执行 SWI 指令或处理器复位后进入的模式。
- Abort 模式，缓存没有命中时进入的模式。
- 未定义模式，执行非法指令时进入的模式。

- 干预模式，处理 IRQ 的模式。
- 快速干预模式，处理 FRQ（快速 IRQ）的模式。
- Hyp 模式，Cortex-A15 专用的虚拟化模式。

ARM 架构包含了以下精简指令集处理器的特性：

- 读取 / 存储架构。
- 正交指令集。
- 大量的  $16 \times 32$  位寄存器阵列。
- 固定的 32 位操作码长度，降低编码数量所产生的耗费，减轻解码和流水线化的负担。
- 大多均为一个 CPU 周期运行。
- 大部分指令可以条件运行，降低在分支时产生的负重，弥补分支预测器的不足。
- 算术指令只会在要求时更改条件码。
- 32 位简型位移器可用来运行大部分的算术指令和寻址计算，而不会损失性能。
- 强大的索引寻址模式。
- 精简但快速的双优先级中断子系统，具有可切换的暂寄存器组。

ARM 处理器支持丰富的指令集，包括传统的 ARM 指令集、高密度的 Thumb/Thumb2 指令集、为浮点运算优化的 VFP 指令集等，部分 ARM 处理器甚至能直接执行部分 Java 字节码，这就为以 Java 作为主要编程语言的 Android 系统带来了极大的便利。

## 2. ARM SoC 架构及选型

ARM 虽然性能强大，但是 ARM 仅仅是一种处理器核，要组成一个通用处理器，还需要有总线系统、存储系统、I/O 系统等部件的配合。为了协调各种部件，ARM 公司推出了为 ARM 处理器定制的、行业标准的 AHB 和 APB 总线。常见的基于 ARM 的 SoC 包含至少一个 ARM 内核、一个 AHB 总线控制器、一个内存控制器、一个闪存控制器、至少一个 APB 总线控制器和一些外设。ARM SoC 的性能取决于 ARM 核和 RAM 子系统，ARM SoC 的功能取决于外设的种类和数目。

基于 Cortex-M 系列核的 SoC 通常主频不超过 200MHz，内置 Flash 与 RAM，Flash 不超过 1MB，RAM 不超过 256KB，面向传统的单片机市场以及 PLC、现场总线等控制类应用。基于 Cortex-M 系列核的 SoC 因直接与外部 I/O 打交道，所以拥有丰富的片内外设，如大量的定时器、丰富的 ADC 和 DAC 以及比较器等模拟外设、UART 总线和 CAN 总线等。这类单片机对外部电路依赖小，低功耗，低成本，可以使用复杂的、大型的实时操作系统，能处理复杂的网络协议，目前正在取代传统的 8 位单片机。Cortex-M 家族中的 Cortex-M0 面向低功耗市场，Cortex-M1 面向 FPGA 嵌入式软核市场，Cortex-M3 面向通用控制和通信市场，Cortex-M4 有可选的 FPU，并能运行在高达 200MHz 的频率下，面向高性能控制、通信和人机接口市场。

面向高可靠性、高可用性、低延时的关键类应用，ARM 公司推出了 Cortex-R 系列实时控制器。与 Cortex-M 系列相比，该类产品拥有双机备份的能力，即在一个芯片内实现两个 CPU 核，双机同时工作，另有一个独立的模块负责监督，在发生故障时屏蔽出错的核，由另一个核继续工作。这种热备份、冗余的结构使得 Cortex-R 系列的可靠性非常高，又因为 Cortex-R 系列的总线、RAM 控制器和 Flash 控制器都支持高级的 ECC 等校验机制，使得 SEU 在多数情况下不会对系统造成严重的影响。另外，基于 Cortex-R 系列内核的实时控制器拥有更大的片内 Flash 和片内 RAM 以及更高的运行频率，为冗余、安全操作系统和控制程序提供了更好的运行环境，在软件层面上也加强了系统的可靠性。与 Cortex-M 系列核的微控制器相同的是，Cortex-R 核的实时控制器也包含丰富的外设，因为片内外设的可靠性一般来说比片外外设要高，而且很多集成外设也包含一定程度的安全特征。

Cortex-A 系列核代表了当今性能最强的 ARM 处理器，该系列包含 Cortex-A5、Cortex-A7、Cortex-A8、Cortex-A9、Cortex-A15 等多款处理器核。所有的 Cortex-A 系列核都是为高性能而设计的，因而都可以用作移动设备或者瘦客户机等高性能设备的处理器核。其中，Cortex-A8 是最早发售的型号，被用作通用应用处理器；Cortex-A9、Cortex-A15 因良好的多核支持，被用于高性能系统，如手机和 ARM 服务器。后来的 Cortex-A7 则是精简版的 Cortex-A8，拥有与 Cortex-A8 相似或略高的性能和更小的硅片面积，同时支持多核，并且耗电更少。Cortex-A5 则是 ARM 公司对 ARM9、ARM11 产品推出的升级产品，与 Cortex-A8、Cortex-A8 和 Cortex-A9 完全应用兼容，是 ARM 公司的低成本、低功耗产品。所有的 Cortex-A 系列核均含有 MMU 等系统管理模块，因此都可以运行 Linux 等大型操作系统。部分处理器还有可选的 NEON 单元、Java 虚拟机、虚拟化等高级功能，这些功能为 ARM 作为高性能移动处理器或高效服务器处理器打下了坚实的基础。通常，Cortex-A 系列的处理器都能运行在至少 800MHz 的频率上，有的甚至能超过 2000MHz。

### 3. ARM 指令集

ARM 处理器属于精简指令集处理器（RISC），这是 ARM 与 x86 处理器最大的不同。实际上，几乎所有的 x86 兼容处理器以外的处理器都是 RISC 处理器，只有 x86、x86-64 等传统的、面向桌面市场的处理器还在使用复杂指令集处理器（CISC）。以常用的 Intel 桌面处理器为例，传统的 CISC 处理器指令集支持丰富的操作，指令可以访问内存，提供大量的内存缓存，因此可以在同样的指令周期内执行更多的命令。然而，CISC 处理器的结构臃肿，需要很大程度的手工优化和工艺优化来提升运行频率，由于内部逻辑单元和缓存太多，耗电量也远高于 RISC 处理器。

RISC 处理器旨在使用简单的指令集实现与 CISC 一样的功能。因为指令集的减少，很多操作需分步进行，从而减少了相同周期内执行指令的个数。尽管如此，RISC 处理器因为结构简单，可以在设计阶段使用自动化工具进行逻辑综合、布局和布线，从而大幅度降低了设计门槛。又因结构简化，RISC 处理器可以在同样的工艺和优化成本上实现更高的运行频率和

更低的电能消耗。举例来说，一颗 Intel 的、基于 Ivy Bridge 架构的至强处理器（E3-1230 v2）拥有 4 个核，每个核运行频率为 3.3GHz，耗电 69W。而同一时期的三星公司生产的 ARM SoC，Exynos 5420，拥有 4 个能运行在 1.9GHz 的核和 4 个能运行在 1.3GHz 的核，而耗电不超过 3W。虽然指令集效率可能没有基于 CISC 且拥有 SSE/SSE/AVX 的至强处理器高，但是每瓦特性能（即能效）远高于后者，因此 RISC 处理器常用于移动设备、嵌入式设备等对功耗有严格要求的领域。

ARM 处理器支持 ARM 和 Thumb 指令集。以最新、最简单的 Cortex-M0 为例，该处理器占用硅片面积小，在 90 纳米工艺下约 0.04 平方毫米，功耗低至 16 微瓦 MHz，是理想的 8 位或 16 位单片机的替代品。在低功耗的同时，Cortex-M0 处理器还集成了丰富的指令，支持 ARM v6 Thumb 指令集，包括 Thumb II 扩展指令集。该处理器的高性能版本甚至还支持单周期 32 位乘法。Cortex-M0 处理器共支持 56 条指令，分为本原指令（Intrinsic Instructions）、内存操作指令（Memory Access Instructions）、通用数据处理指令（Generic Data Processing Instructions）、分支和控制指令（Branch and Control Instruments）和其他指令（Miscellaneous Instruments）。

本源指令和其他指令是标准 C 语言不用的，但是在操作系统或应用程序中可能用到用于控制 CPU 工作的指令。为了在 C 语言中使用这些 CPU 相关的扩展指令，可以使用 CMSIS 接口或者内嵌汇编代码。以下是 Cortex-M0 核所支持的本源指令和其他指令。

CPSIE i: 使能 IRQ

CPSID i: 禁止 IRQ

ISB: 代码缓存同步

DSB: 数据缓存同步

MSB: 内存同步

NOP: 等待一个周期

REV: 32 位按位反转

REV16: 16 位按位反转

REVSH: 32 位低半字反转

SEV: 发送中断

WFE: 等待事件

WFI: 等待中断

MRS: 从特殊寄存器移动至通用寄存器

MSR: 从通用寄存器移动至特殊寄存器

BKPT: 设置断点

SVC: 调用 SVC 例程

内存操作指令用于读、写内存、生成地址和堆栈管理。以下是 Cortex-M0 支持的内存操作指令。

ADR: 生成相对 PC 的地址

LDM: 装载多个寄存器

STM: 存储多个寄存器

LDR 系列: 装载一个寄存器

STR 系列: 存储一个寄存器

PUSH: 入栈

POP: 出栈

通用数据处理指令对数据进行算术运算。Cortex-M0 支持如下通用数据处理指令。

ASRS: 算术右移

LSLS: 逻辑左移

LSRS: 逻辑右移

ROR: 循环右移

ADCS: 带进位加法

ADDS: 无进位加法

ANDS: 逻辑与

BICS: 位清除

CMN: 负数比较

CMP: 比较

EORS: 异或

MOVS: 复制数据

MULS: 乘法

MVNS: 取非并移动

ORRS: 逻辑或

RSBS: 取反相减

SBCS: 带进位减法

SUBS: 无进位减法

SXTB: 设置扩展字节

SXTH: 设置扩展双字节

UXTB: 取消扩展字节

UXTH: 取消扩展双字节

TST: 按位与并比较

分支控制指令改变 PC 中的值, 从而实现代码跳转。跳转可以有条件的, 也可以是无条件的。有条件跳转的条件存放于状态寄存器。Cortex-M0 支持如下分支控制指令。

B: 条件跳转

BL: 带返回跳转



BX: 交换指令跳转

BLX: 带返回的交换指令跳转

#### 4. ARM 处理器的外围电路

ARM 处理器通常作为 SoC 的核存在, 本节中所采用的 ARM 处理器都是基于 ARM 处理器的 SoC。近年来常用的 ARM 核为 ARM7、ARM9、ARM11、Cortex-M0、Cortex-M1、Cortex-M3、Cortex-M4、Cortex-R4、Cortex-R7、Cortex-A5、Cortex-A6、Cortex-A8、Cortex-A9 和 Cortex-A15 等。由于处理器市场定位、性能区间和设计的相似性, 本节将 ARM7、Cortex-M0、Cortex-M1、Cortex-M3 和 Cortex-M4 称为微控制器系列, 将 Cortex-R4 和 Cortex-R7 称为实时处理器系列, 将 ARM9、ARM11、Cortex-A5、Cortex-A7、Cortex-A8、Cortex-A9 和 Cortex-A15 称为应用处理器系列。

通常微控制器系列的 ARM 处理器不需要外接 Flash 和 RAM, 其内部的 Flash 和 RAM 存储器能够满足绝大部分应用需求。有些早期的 ARM7 处理器支持外接 SDRAM, 如三星公司的 S3C44B0, 此类处理器定位于高端工控而非微控制器, 本书不介绍。一般的微控制器集成度很高, 仅需要提供合适的电源即可工作。有的芯片可能还需要输入一个外部复位信号。多数的处理器支持电路编程, 即芯片在安装之后通过 ISP/ICP 接口进行编程。

综上所述, 多数微控制器类的 ARM 处理器的电路设计十分简单, 最小系统仅需要电源稳压器、退耦电容以及 ISP/ICP 相关电路即可。如图 1-7 所示为一个实际的应用电路——STM32F103T8U6 Cortex-M3 处理器为例。

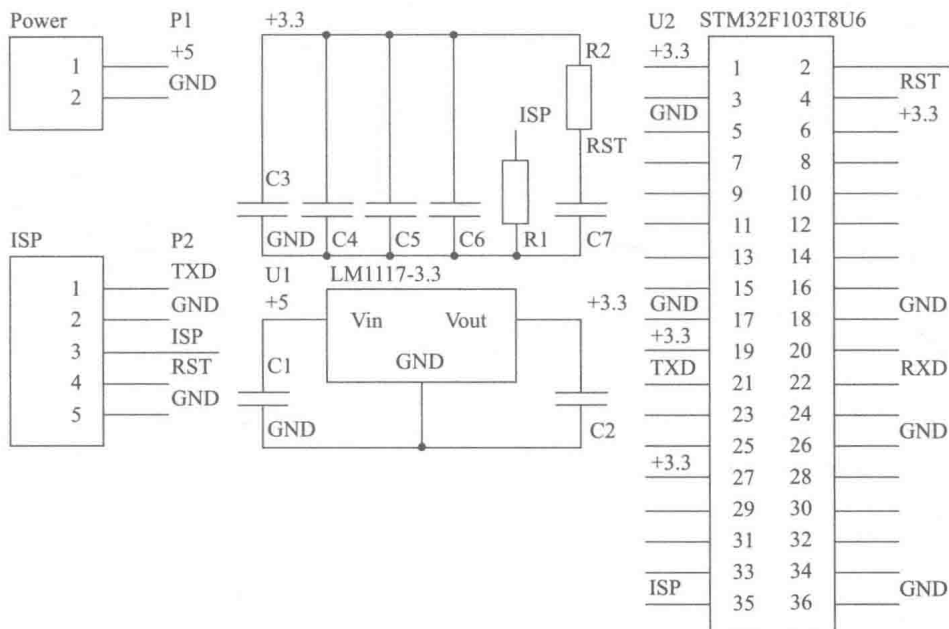


图 1-7 STM32F103T8U6 Cortex-M3 处理器应用电路

实时处理器，即 Cortex-R4 和 Cortex-R7 等，因为面向高可靠市场，所以一般集成 Flash 与 RAM，用法与微控制器类似。由于对可靠性的要求高，所以配备相应的片外监控芯片与更可靠的电源供应。常见的片外外设看门狗定时器、电源监控和复位芯片、电平转换与静电防护芯片等。供电方面应该配备多重可并联的冗余供电，以防止因供电问题导致可靠性降低。

应用处理器定位高性能市场，内部集成的 Flash 和 RAM 不能满足需求。几乎所有应用处理器都要外加代码存储器和 RAM。代码存储器通常由 Flash 闪存提供，也可以是硬盘等其他存储介质。为了保证高峰值运算能力与低平均功耗，应用处理器一般会采用复杂的电源管理系统，因此外围电路非常复杂，可能需要六层或八层电路板来实现。虽然有些应用处理器通过封装叠加或多芯片封装的方法把 Flash 和 RAM 集成到一个封装内，目的在于降低布线的难度，但其外设电路仍然复杂。下面将专门介绍应用处理器在 Android 智能设备上的硬件架构。

### 5. 基于 ARM 的 Android 智能设备硬件架构

常见的 Android 手机 / 平板架构，其中包含应用处理器、基带处理器和射频前端、内存、闪存、音频 Codec、电源管理、传感器、触摸屏幕等组件。

应用处理器通常是一个 ARM 处理器或 MIPS 处理器，负责运行 Android 程序代码。一般情况下，应用处理器集成了图形处理器、浮点运算协处理器和 Java 字节码处理器。图形处理器用于通过 OpenGL ES 接口提供高速的图像渲染、3D 场景重构。浮点运算协处理器和 Java 字节码处理器能在降低 CPU 负载的情况下完成复杂的运算和 Java 字节码的执行，实现快速的 Java 虚拟机。这种组合决定了基于 Java 的 Android 几乎可以和基于 C++ 的移动系统一样快。

基带处理器一般是一个高性能的 DSP 处理器，它负责无线通信的基带算法。所有的基于蜂窝系统的通信均通过基带系统实现。射频前端分为接收和发射两大部分：接收部分包含低噪声放大器、射频混频器、本机振荡器、带通滤波器和模数转换器；发射部分包含数模转换器、本机振荡器、射频混频器和功率放大器。其中模数转换器和数模转换器一般被集成在一个芯片里，叫做模拟前端。模拟前端直接与基带处理器连接，由基带处理器负责通信信号的处理。基带处理器还负责语音的处理，包括消回音算法、信噪比增强和 AMR/ADPCM 编解码等。通过基带处理器与应用处理器通信，实现了完整的蜂窝系统设备。

内存、闪存是支撑应用处理器和基带处理器的必需部件。内存提供临时、高速的数据缓冲，用以支撑高速的处理器数据请求。一般通过 DDR3、LPDDR3 或 LPDDR2 与处理器连接。DDR3 的优势是高性能，LPDDR3 的优势是低功耗，LPDDR2 的优势是在功耗较低的前提下提供快速的突发访问能力。闪存则提供永久的数据保存，用来存储程序和数据。传统上分为 NOR Flash 和 NAND Flash。NOR Flash 一般用来存储系统程序，而 NAND Flash 用来存储用户程序和数据。由于 NAND Flash 的成本下降、性能提升，现在的手机几

乎完全使用 NAND Flash。一种新型的基于 NAND Flash 的 eMMC/iNAND 正在慢慢占领新的市场,这种新的闪存器件需要更少的 I/O 资源,拥有不低于传统闪存的性能,从而获得更多的青睐。

音频 Codec,全称是音频编解码器,负责采集来自话筒的音频信号以及从耳机接口或扬声器输出的音频信号。由于包含了数模转换和模数转换,所以被称为 Codec,即编码器和解码器。

电源管理部分负责电池的充放电、电池电量、寿命监测,并为各种处理器等系统部件提供稳定的电源。手机等移动设备需要高度省电,因此很多电源管理系统都需要配合应用处理器实现各种低功耗功能,比如动态电压频率调节、低功耗模式等。

传感器与触摸屏提供了最直观的输入输出功能。通常智能手机包含加速度传感器、陀螺仪、温度传感器、光线传感器、接近传感器和地磁传感器等多个传感器部件。这些传感器会占用大量的 I/O 资源。为了节约 I/O 资源,会用一个单片机将全部的传感器数据汇总,一并发给应用处理器。这个单片机就是 Sensor Hub。触摸屏可以分成触摸控制器和液晶显示屏。触摸控制器一般与各种传感器一起连接至 Sensor Hub,而液晶显示屏直接连接至应用处理器的 LCD 接口。

## 1.2 嵌入式系统实例

pcDuino 是一款迷你的 PC 机,只有信用卡那么大小,却能够运行 Ubuntu 和 Android 的 ICS 系统。最新版本的 pcDuino3 采用了全志 A20 双核 1.5GHz ARM Cortex A7 处理器,性能优异。下面以该系统为实例,介绍一些与嵌入式系统软硬件相关的内容。

### 1.2.1 pcDuino 部分硬件功能介绍

#### 1. I2C

I2C (Inter-Integrated Circuit) 总线是由 PHILIPS 公司开发的两线式串行总线,用于连接微控制器及其外围设备,产生于 20 世纪 80 年代,最初的目的是进行音频和视频设备开发,现在主要应用于服务器管理,包括单个组件状态的通信。例如管理员可对各个组件进行查询,以管理系统的配置和掌握组件的功能状态,如电源和系统风扇,可随时监控内存、硬盘、网络、系统温度等多个参数,增加了系统安全性,方便管理。由于接口直接在组件之上,因此 I2C 总线占用的空间非常小,减少了电路板的空间和芯片管脚的数量,降低了互联成本,集中体现了 I2C 总线最主要的优点——简单和有效。

具体地说,I2C 总线是由数据线 SDA 和时钟 SCL 构成的串行总线,各种被控制器件均并联在这条总线上,每个器件都有唯一的地址识别,可以作为总线上的一个发送器件或接收器件(具体由器件的功能决定)。I2C 总线的接口电路结构如图 1-8 所示。

I2C 总线的几种信号状态具体分为以下几种。

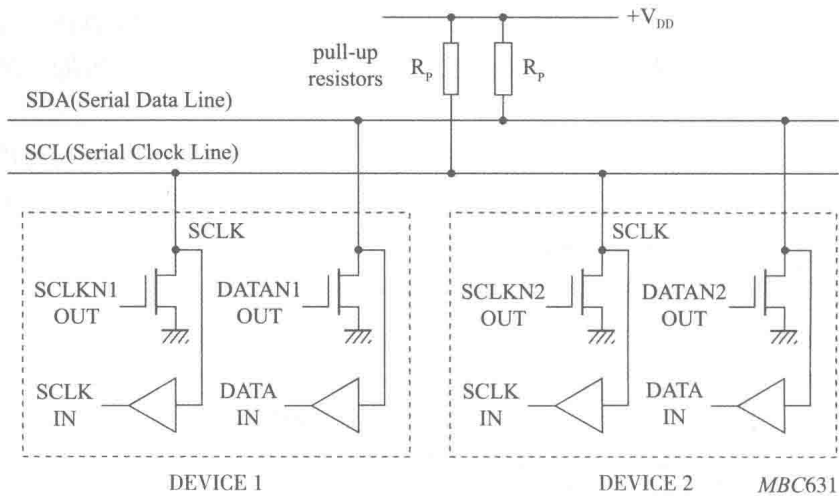


图 1-8 I2C 接口电路结构图

- 空闲状态：SDA 和 SCL 都为高电平。
- 开始条件 (S)：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。
- 结束条件 (P)：SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。
- 数据有效：在 SCL 的高电平期间，SDA 保持稳定，数据有效。SDA 的改变只发生在 SCL 的低电平期间。
- ACK 信号：数据传输的过程中，接收器件每接收一个字节数据要产生一个 ACK 信号，向发送器件发出特定的低电平脉冲，表示已经收到数据。

针对 I2C 总线的基本操作主要包括读、写及控制。主器件（通常为微控制器）控制 I2C 总线，产生串行时钟（SCL）、控制总线的传输方向、产生开始和停止条件。数据传输过程中，主器件产生开始条件，随后是器件的控制字节（前 7 位是从器件的地址，最后一位为读写位），接下来是读写操作的数据，以及 ACK 响应信号。数据传输结束时，主器件产生停止条件。具体的过程如图 1-9 所示。

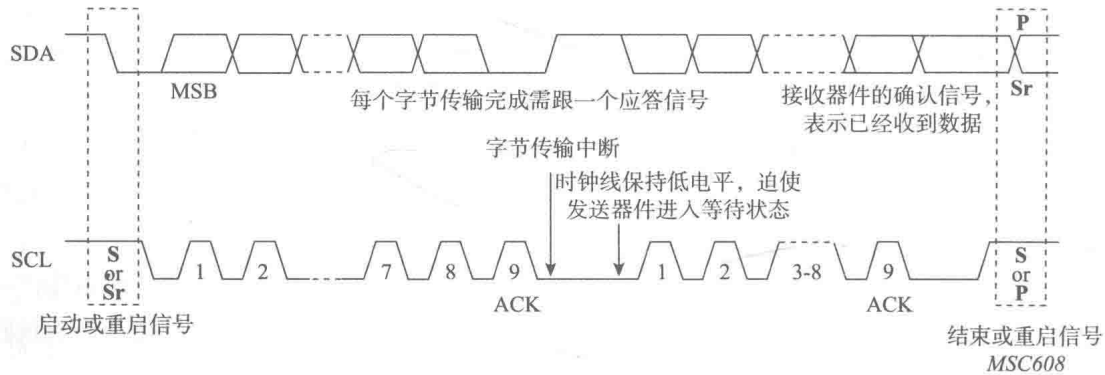


图 1-9 I2C 总线数据传输图

通过分析 I2C 源码可知, 在 drivers/I2C/ 目录下, 包含以下几个重要文件和目录。

- 1) 文件 I2C-core.c: I2C 子系统核心功能的实现。
- 2) 文件 I2C-dev.c: 通用的从设备驱动实现。
- 3) 目录 busses: 里面包括基于不同平台实现的 I2C 总线控制器驱动, A20 使用的源文件为 I2C-sun7i.c。

在 sys\_config.fex 中有 5 组 I2C 总线可供使用, 分别是 twi0、twi1、twi2、twi3 和 twi4。配置如下:

```
[twi0_para]
twi0_used=1
twi0_scl=port :PB0<2> <default> <default> <default>
twi0_sda=port :PB1<2> <default> <default> <default>

[twi1_para]
twi1_used=1
twi1_scl=port :PB18<2> <default> <default> <default>
twi1_sda=port :PB19<2> <default> <default> <default>

[twi2_para]
twi2_used=1
twi2_scl=port :PB20<2> <default> <default> <default>
twi2_sda=port :PB21<2> <default> <default> <default>
.....
```

其中常用的为 twi0、twi1、twi2, twi3 与 twi4 使用时按照 twi0 等格式进行添加即可。

若使用哪一组 I2C 总线, 将对应的 twiX\_used 置为 1 即可。通常情况下, twi0、twi1、twi2 均设置为 1。

对于 I2C 总线控制器的配置, 可通过命令 make ARCH = arm menuconfig 进入配置主界面, 并按以下步骤操作:

- 1) 选择 Device Drivers 选项, 进入下一级配置。
- 2) 选择 I2C support 选项, 进入下一级配置。
- 3) 选择 I2C HardWare Bus support 选项, 进入下一级配置。
- 4) 选择 Allwinner Technology SUN7I I2C interface 选项, 可选择直接编译进内核中, 也可以选择编译成模块, 如图 1-10 所示。

此外, 若需要获取指定 I2C 总线控制器相关的调试打印信息, 可选择 sun7i i2c print transfer information 选项, 并在 bus num id 选项中指定对应的 I2C 总线控制器编号, 可输入 0、1、2 和 3、4, 如图 1-11 所示。

## 2. I2C 体系结构描述

位于 drivers/I2C/busses 目录下的文件 I2C-sun7i.c, 是基于 SUN7I 平台实现的 I2C 总线



I2C 设备都需要一个 I2C\_client 来描述。I2C\_driver 对应一套驱动方法，其主要成员函数是 probe()、remove()、suspend()、resume() 等，另外 id\_table 是该驱动所支持的 I2C 设备的 ID 表。I2C\_driver 与 I2C\_client 的关系是一对多，一个 I2C\_driver 上可以支持多个同等类型的 I2C\_client。

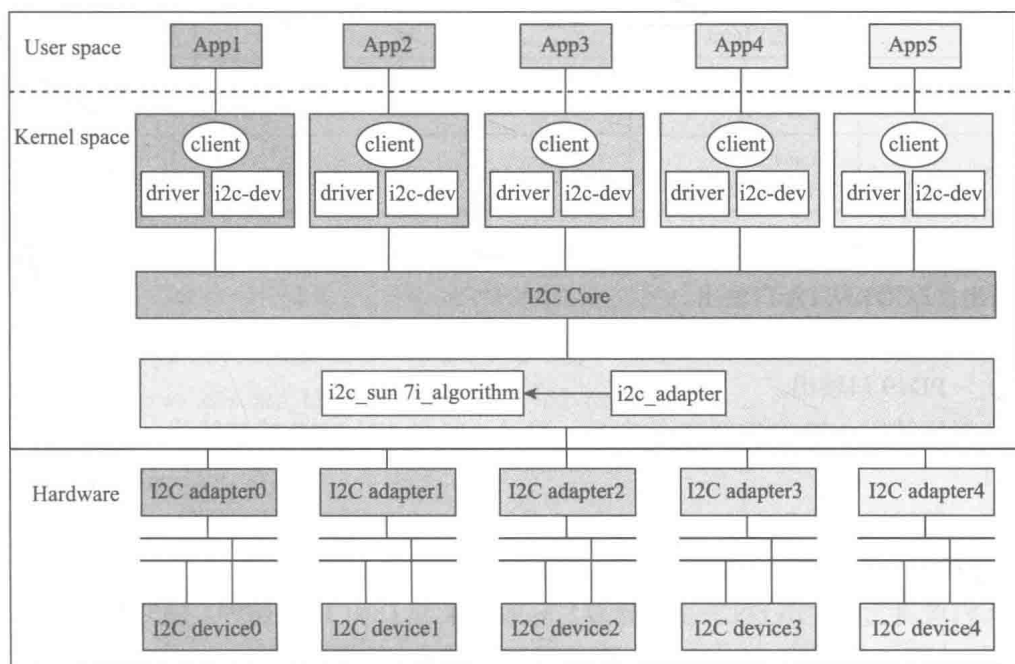


图 1-12 I2C 驱动层次架构图

I2C 常用接口如下所示：

- I2C\_add\_driver
- I2C\_del\_driver
- I2C\_set\_clientdata
- I2C\_get\_clientdata
- I2C\_master\_send
- I2C\_master\_send
- I2C\_master\_resv
- I2C\_transfer

### 3. A20 与 LCD

A20 有两路显示系统，支持双屏输出，支持 LCD 屏的接口形式及最大分辨率如表 1-1 所示。

表 1-1    LCD 输出 I/O 口

	接    口	分    辨    率	LCD0	LCD1
HVRGB	ParallelRGB	2048×1536	PD	PA
	SerialRGB/CCIR	1280×720	PD	PA
CPU/80	Parallel 18bit Parallel 16bit	1280 × 720	PD	PA
LVDS	Single Link Dual Link	1440×900	1920×1200 PD0 ~ PD19	PD0 ~ PD9 PD10 ~ PD19
DSI	4Lane (A20+SSD2828)	1920×1200	PD	PA

HVRGB 接口和 CPU/I80 接口用于并行数据输出，是 TTL 电平的屏接口，其中 LCD0 从 PD 口输出，LCD1 从 PA 口输出。

LVDS 接口用于串行输出，是差分信号的屏接口，LVDS0 从 PD0 ~ PD9 输出，LVDS1 从 PD10 ~ PD19 口输出。

DSI 接口使用 A20+SSD2828 的方式，通过 A20 的 HV RGB 接口输出，由转换 IC 转成 DSI 接口。

如果一路输出使用 Dual Link LVDS，占用了 LVDS 的所有引脚，另外一路只能使用其他接口形式输出。其他接口形式任意组合的双屏输出都支持。

以 LVDS 屏为例，LVDS 屏接口分为 Single Link 和 Dual Link 两种。LVDS 屏使用 LVDS 差分信号，LVDS Single Link 具有 1 组时钟对和 3 组或 4 组数据对。

如图 1-13 所示是一个典型 LVDS Single Link 屏的模组规格书的引脚定义。A20 与该 LCD 屏的引脚连接如表 1-2 所示。

表 1-2    LVDS Single Link 显示屏引脚定义

1	VDD	Power Supply, 3.3V typ	12	RXINO2-	D2-
2	VDD	Power Supply, 3.3V typ	13	RXUNO2+	D2+
3	VDD	Power Supply, 3.3V typ	14	GND	Ground
4	NC	NC	15	RXCLKIN-	CLK-
5	GND	Ground	16	RXCLKIN+	CLK+
6	RXINO-	D0-	17	GND	Ground
7	RXINO+	D0+	18	NC	NC
8	GND	Ground	19	NC	NC
9	RXINO1-	D1-	20	GND	Ground
10	RXINO1+	D1+	21	LVBIT	GND ( 6 or 8bit Change )
11	GND	Ground	22	DITHER	GND (FRC)



(续)

23	GND	GND	28	VLED	LED Power Supply, 3 ~ 5V
24	LED EN (PWM)	PWM	29	VLED	LED Power Supply, 3 ~ 5V
25	LVFMT	GND (MSB/LSB Change)	30	VLED	LED Power Supply, 3 ~ 5V
26	BIST	NC	31	NC	NC
27	VLED	LED Power Supply, 3 ~ 5V			

该 LCD 屏有 3 组数据对, 显示要求为 18bit 色深, 且不区分模式。故

$\text{lcd\_lvds\_colordepth} = 1$ ,  $\text{lcd\_lvds\_mode} = 0$

该 LCD 屏参数如表 1-3 所示。时序参数配置与 HVparallelRGB 类似。区别于 HVparallelRGB, 该 LCD 屏参数没有指定 BackPorch 和 SyncWidth。根据 A20 时序要求, LCD 控制器配置如下:

- $\text{lcd\_ht} > \text{lcd\_x} \times \text{cycle} + \text{lcd\_hbp}$ , 得  $\text{lcd\_hbp} < 64$ 。
- 取  $\text{lcd\_hbp} = 20$ ;  $\text{lcd\_hbp} > \text{lcd\_hspw}$ , 取  $\text{lcd\_hspw} = 10$ 。
- $\text{lcd\_vbp} = 20$ ,  $\text{lcd\_vspw} = 10$ 。

LCD I/O 必须配置为 LVDS。

表 1-3 LVDS Single Link 显示屏参数

ITEM			SYMBOL	MIN	TYP	MAX	UNIT
LCD Timing	Frame Rate		—	TBD	60	TBD	Hz
	DCLK		Frequency	$f_{\text{CLK}}$	TBD	66.7	TBD
	DENA	Horizontal	Horizontal total time	$t_{\text{H}}$	TBD	864	TBD
			Horizontal Active time	$t_{\text{HA}}$	TBD	800	TBD
			Horizontal Blank time	$t_{\text{HB}}$	TBD	64	TBD
		Vertical	Vertical total time	$t_{\text{V}}$	TBD	1288	TBD
			Vertical Active time	$t_{\text{VA}}$	TBD	1200	TBD
			Vertical Blank time	$t_{\text{VB}}$	TBD	8	TBD

### (1) LCD 硬件参数

1)  $\text{lcd\_if}$  为 LCD 接口选择, 参数值所对应的含义如下:

0—HVRGB 接口; 1—CPU/I80 接口; 2—Reserved; 3—LVDS 接口; 4—DSI 接口。

2)  $\text{lcd\_hv\_if}$ , 这个参数只有在  $\text{lcd\_if} = 0$  时才有效, 定义 RGB 同步屏下的几种接口类型。设置相应值的对应含义如下:

0—ParallelRGB; 8—SerialRGB; 10—DummyRGB; 11—RGBDummy。

3)  $\text{lcd\_hv\_s888\_if}$ , 这个参数只有在  $\text{lcd\_if} = 0$  且  $\text{lcd\_hv\_if} = 1$  (SerialRGB) 时才有效。

定义奇数行 RGB 输出的顺序如下。

```
0: OddlinesR→G→B: EvenlineR→G→B
1: OddlinesB→R→G
2: EvenlineR→G→B OddlinesG→B→R
4: EvenlineR→G→B OddlinesR→G→B
5: EvenlineB→R→G OddlinesB→R→G
6: EvenlineB→R→G OddlinesG→B→R
8: EvenlineB→R→G OddlinesR→G→B
9: EvenlineG→B→R OddlinesB→R→G
10:OddlinesG→B→R: EvenlineG→B→R
```

4) lcd\_hv\_syuv\_if, 这个参数用来设置 YUV 输出顺序的, 只有在 lcd\_if = 0 且 lcd\_hv\_if = 2 (SerialYUV) 时才有效。定义 YUV 输出格式如下:

0—YUYV; 1—YVYU; 2—UYVY; 3—VYUY。

5) lcd\_cpu\_if, 只有在 lcd\_if = 1 时, 即设置为 CPU/180 类型的接口, 这个参数才有效。设置相应值的对应含义如下:

0—18bit/1cycle parallel(RGB666); 4—16bit/1cycle parallel(RGB565); 6—18bit/3cycle parallel(RGB666); 7—16bit/2cycle parallel(RGB565)。

6) lcd\_lvds\_ch, 设置相应值的对应含义如下:

0—Single Link; 1—Dual Link。

LVDS 接口的 LCD 屏, 定义 1 组时钟对 +3/4 组数据对, 为 1 个 link。若有 2 组时钟对, 则为两个 link。

7) lcd\_lvds\_bitwidth, 设置相应值对应含义如下:

0—8 bit per color; 1—6 bit per color。

8) lcd\_lvds\_mode, 这个参数只有在 lcd\_lvds\_bitwidth = 0 时才有效。设置相应值对应含义如下:

0—NS mode; 1—JEDIA mode。

NS mode 和 JEDIA mode 的定义如图 1-14 所示。

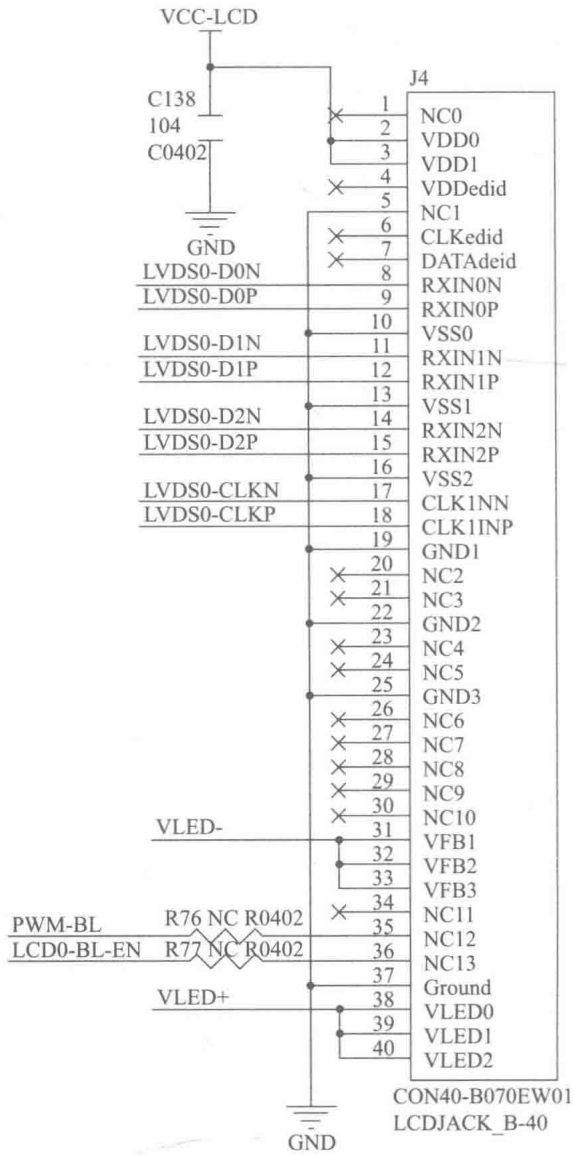


图 1-13 A20 与 LVDS Single Link 显示屏连接图

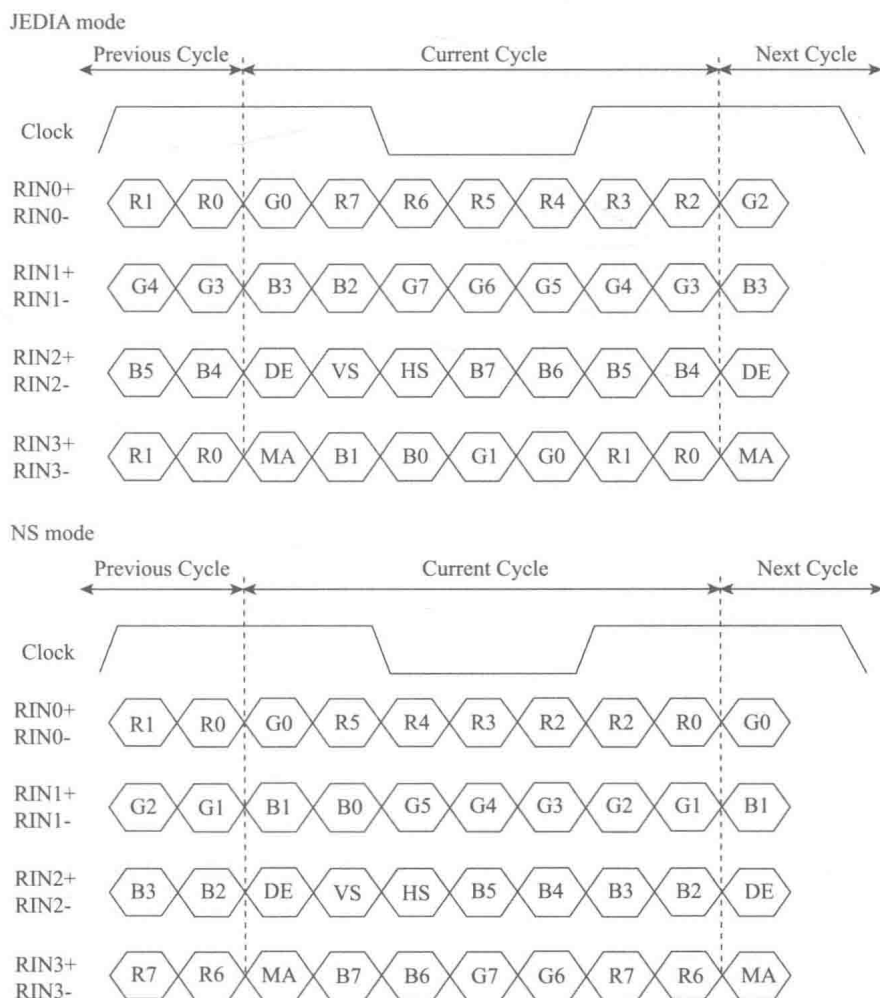


图 1-14 LVDS JEDIA mode 和 NS mode

## 9) lcd\_pin。

示例: `lcd0 = port:PD00<3><0><default><default>`

含义: lcd0 这个引脚, 即 PD0, 配置为 LVDS 输出。

第一个尖括号表示功能分配: 0 为输入, 1 为输出, 2 为 LCD 输出, 3 为 LVDS 接口输出, 7 为 disable。

第二个尖括号表示内置电阻: 0 表示内部电阻高阻态, 如果是 1 则内部电阻上拉, 2 代表内部电阻下拉。使用 default 代表默认状态, 即电阻上拉。其他数据无效。

第三个尖括号表示驱动能力: default 表示驱动能力是等级 1。

第四个尖括号表示默认值: 即是当设置为输出时, 该引脚输出的电平, 0 为低电平, 1 为高电平。

LCD PIN 的配置如下:

LCD 为 HVRGB 或 DSI 屏, CPU/I80 屏时, 必须定义相应的 I/O 口为 LCD 输出 (如果是 0 路输出, 第一个尖括号为 2; 如果是 1 路输出, 第一个尖括号为 3)。

LCD 为 LVDS 屏时, 必须定义 PD 口对应的 I/O 口为 LVDS 输出 (即第一个尖括号为 3)。

LCD PIN 的所有 I/O 均可通过注释方式去掉其定义, 显示驱动对注释 I/O 不进行初始化操作。

10) lcd\_gpiO\_x。

示例: `lcd_gpiO_0 = port:PA06<0><0><default><default>`

含义: lcd\_gpiO\_0 引脚为 PA06。

第一个尖括号表示功能分配: 0 为输入, 1 为输出。

第二个尖括号表示内置电阻: 0 内部电阻高阻态, 如果是 1 则内部电阻上拉, 2 就代表内部电阻下拉。使用 default 代表默认状态, 即电阻上拉。其他数据无效。

第三个尖括号表示驱动能力: default 表示驱动能力是等级 1。

第四个尖括号表示默认值: 即是当设置为输出时, 该引脚输出的电平, 0 为低电平, 1 为高电平。

A20 配置中, 共有 6 个可选的 lcd\_gpiO 引脚, lcd\_gpiO\_0、lcd\_gpiO\_1、lcd\_gpiO\_2、lcd\_gpiO\_3、lcd\_gpiO\_4、lcd\_gpiO\_5。

11) lcd\_bl\_en。

示例: `lcd_bl_en = port:PH07<1><0><default><1>`

含义: lcd\_power 引脚为 PH07, PH07 输出高电平时打开 LCD 背光: 上下拉不使能。

第一个尖括号表示功能分配: 1 为输出。

第二个尖括号表示内置电阻: 0 内部电阻高阻态, 如果是 1 则内部电阻上拉, 2 就代表内部电阻下拉。使用 default 代表默认状态, 即电阻上拉。其他数据无效。

第三个尖括号表示驱动能力: default 表示驱动能力是等级 1。

第四个尖括号表示输出有效所需电平: LCD 背光工作时的电平, 0 为低电平, 1 为高电平。

12) lcd\_power。

示例: `lcd_power = port:power2<1><0><default><1>`

含义: LCD 的供电定义 gpiO 控制。

13) lcd\_pwm。

示例: `lcd_pwm = port:PB02<2><0><default><default>`

含义: PB02 输出 PWM 信号。

A20 方案固定 PB02 为 PWM 信号输出引脚。建议使用此默认配置。

#### 4. PWM

PWM (Pulse Width Modulation, 脉冲宽度调制) 即脉宽调制, 是一种脉冲编码技术。一般

PWM 信号的周期不变,用占空比(有效电平在整个信号周期中的时间比率,为 0%~100%)来表示编码数值。PWM 可以用于对模拟信号电平进行数字编码方法,也可以通过控制高电平(或低电平)在整个周期中的时间来控制输出的能量,从而控制电机转速或 LED 亮度。

PWM 控制技术以其控制简单灵活和动态响应好的优点成为电力电子技术中最广泛应用的控制方式,也成为研究的热点。当今科学技术的发展已经模糊了学科界限,结合现代控制理论思想或无谐振波开关技术将会成为 PWM 控制技术发展的主要方向之一。如图 1-15 所示为典型的 PWM 波形。

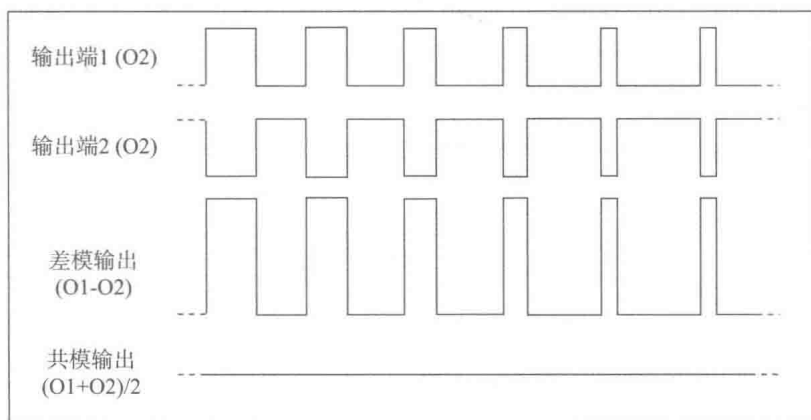


图 1-15 典型的 PWM 波形

PWM 信号一般由计数器和比较器产生。计数器以一定的频率自加,比较器中设定了一个阈值,当计数器中的数字小于这个阈值时,输出一种电平状态(如高电平);当数字大于这个阈值时,输出另一种电平状态(如低电平);当计数器计满后清零,又回到最初的电平状态。这样通过 I/O 引脚的周期翻转,就形成了 PWM 波形,如图 1-16 所示。

pcDuino 上的 PWM 信号如图 1-17 所示。

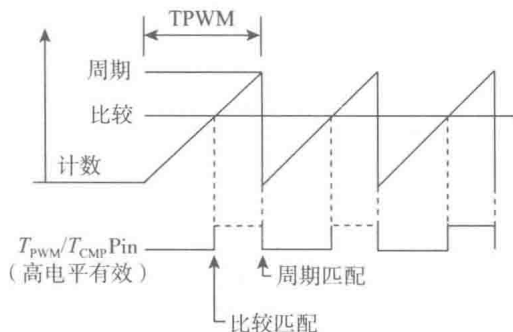


图 1-16 产生 PWM 原理

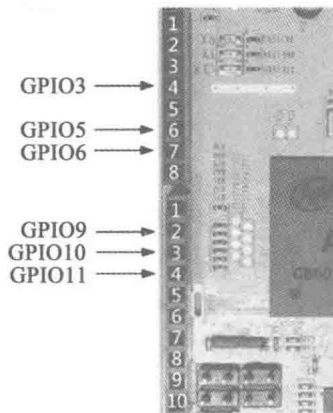


图 1-17 pcDuino 上的 PWM

可用于输出 PWM 信号的 GPIO 共有 6 个，其中 GPIO3、9、10、11 的 PWM 频率为 125Hz ~ 2kHz，GPIO5、6 的 PWM 频率只能设置为 195Hz、260Hz、390Hz、520Hz、781Hz 中的某个特定值。Arduino 库中提供了 analogWrite() 函数来设置 PWM 占空比，增加了 pwmfreq\_set() 函数，用于设置 PWM 信号周期。5、6 引脚为复用引脚。

5. G-Sensor

在人机交互过程中，G-Sensor 起着非常重要的作用，G-Sensor 作为输入设备，能感知当前其所处的空间状态，将它附着在终端上配合使用，能测量出终端在空间上的坐标状态，从而获知终端用户的操作意图，如横竖屏切换、转弯变向等。

通常 G-Sensor 通过 4 个引脚与主机连接，分别为 VCC、GND、SDA、SCL。引脚正常工作时候的高电平均为 3.3V。

在 G-Sensor 的硬件调试过程中，需要确认下列项：

- 1) 各个引脚与 HOST 正确连接。
- 2) 电源电压是否正常，即 VCC 接入电压为 3.3V，GND 电压为 0V。
- 3) I2C 引脚电平是否匹配。
- 4) 设备使用的 I2C 地址，特别是对一台设备进行多个地址设置时。

(1) G-Sensor 的体系结构 (如图 1-18 所示)

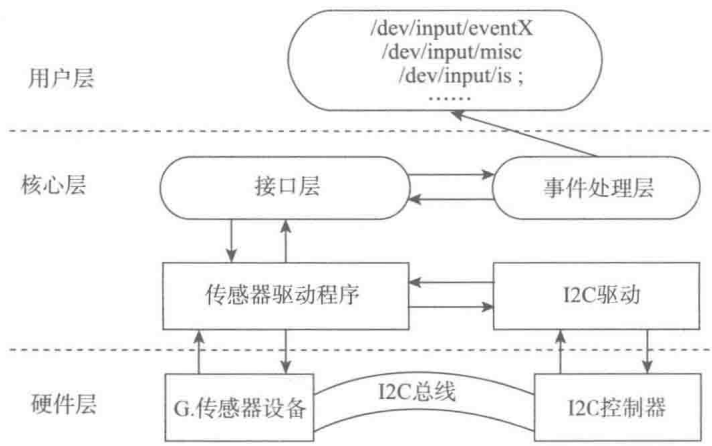


图 1-18 G-Sensor 体系结构

G-Sensor 设备为使用 I2C 总线进行通信的输入设备，G-Sensor driver 通过调用 I2C 驱动的相应接口来实现对 G-Sensor 设备的控制与通信，如 G-Sensor driver 对 G-Sensor 设备硬件和各寄存器的读写访问等。

G-Sensor driver 将底层硬件对用户输入访问的响应转换为标准的输入事件，再通过核心层 (Input Core) 提交给事件处理层；核心层向下提供了 G-Sensor driver 的编程接口，向上又提供了事件处理层的编程接口。事件处理层 (Input Event Drivers) 为用户空间的应用程序提

供了统一访问设备的接口，并对驱动层提交的事件进行处理；用户空间（User space）将根据设备的节点进行数据的读取以及相应的处理。

## （2）模块数据结构描述

1) struct I2C\_driver bma250\_driver：该变量会注册到 I2C\_driver 中，driver.name 为匹配设备名，probe 为设备的侦测函数，address\_list 为 I2C 的侦测地址，suspend 为休眠唤醒函数。

代码清单1-1: bma250\_driver注册

---

```
static struct I2C_driver bma250_driver = {
    .class = I2C_CLASS_HWMON,
    .driver = {
        .owner    = THIS_MODULE,
        .name= SENSOR_NAME,
    },
    .id_table= bma250_id,
    .probe= bma250_probe,
    .remove= bma250_remove,
#ifdef CONFIG_HAS_EARLYSUSPEND
#else
#ifdef CONFIG_PM
    .suspend = bma250_suspend,
    .resume = bma250_resume,
#endif
#endif
    .address_list = normal_I2C,
};
```

---

2) struct bma250\_data：代表了 G-Sensor 驱动所需要的信息的集合，用于帮助实现对采样信息的处理。

代码清单1-2: struct bma250\_data注册

---

```
struct bma250_data {
    struct I2C_client *bma250_client;
    atomic_t delay;
    atomic_t enable;
    unsigned char mode;
    struct input_dev *input;
    struct bma250acc value;
    struct mutex value_mutex;
    struct mutex enable_mutex;
    struct mutex mode_mutex;
    struct delayed_work work;
    struct work_struct irq_work;
#ifdef CONFIG_HAS_EARLYSUSPEND
```

---





这使得操作系统必须具有复杂的任务调度、请求处理能力。常见的高级操作系统有 GNU/Linux、Windows CE、Windows Phone、vxWorks 等，这些系统与前面所述的 FreeRTOS 等不同，拥有常见的计算机操作系统架构，能处理更加复杂的请求和动态加载代码。

很多处理器上电时仅有部分基本模块工作，为了成功加载操作系统内核，还必须借助引导器。以 Linux 内核来说，需要使用 Grub/Lilo 等引导器才能够在 x86 计算机上运行。同样，在手机等移动设备常用的 ARM 处理器上，Linux 需要 U-Boot、vivo、blob 等引导器才能启动。引导器通常不需要编写，通过修改原有的代码，使之运行在自定义的平台上即可。

## 2. 中间件选型

中间件是为系统软件和应用软件提供连接的软件，如数据库、Web 服务器等均属于中间件。为便于软件各部件之间的沟通，在现代信息技术应用框架中，如 Web 服务、面向服务的体系结构等，中间件的应用比较广泛。严格来讲，中间件技术已经不局限于应用服务器、数据库服务器。围绕中间件，Apache 基金会、IBM、Oracle、微软各自发展出了较为完整的软件产品体系。中间件技术创建在对应用软件部分常用功能的抽象上，将常用且重要的过程调用、分布式组件、消息队列、事务、安全、连接器、商业流程、网络并发、HTTP 服务器、Web Service 等功能集于一身，或者分别在不同品牌的不同产品中完成。

在嵌入式系统中，中间件一般不会处理复杂的大型应用。简单来说，嵌入式中间件就是运行在应用程序之下、操作系统之上，联系应用程序与操作系统，或应用程序与其他应用程序的软件。一般情况下，一个典型的系统中，图形用户界面、嵌入式数据库、运行库等都属于中间件。在传统开发过程中，软件通常由开发商自行开发完成。然而，随着嵌入式系统越来越大，功能越来越强，系统中从底层架构到上层需求之间或多或少存在重复，中间件就成为了一种低成本、高质量的选择。好的中间件选型会大幅度减少开发者的工作，同时减少测量、调试的工作。

一个完备的 Android 嵌入式系统中，成品的中间件几乎包含了支持 APK 程序的一切，从底层库函数到 Java 虚拟机，再到第三方的 JNI 库，最后是各种各样的 JAR 包。这些大量的中间件大幅度减少了 Android 系统开发的时间与成本。

## 3. 应用程序开发

作为运行在系统最顶层的程序——应用程序（简称 APP），需要直接与用户沟通，因此不仅需要具备完善的功能，还要具备美观大方的用户界面，以及舒适的用户体验。APP 开发往往依据常规软件工程的开发流程，在此不赘述。

## 4. pcDuino 的软件配置

pcDuino 是联斯普瑞电子科技有限公司发布的一款小巧而强大的 PC 平台，它结合了 ARM 架构的迷你 PC 和 Arduino 的优势，实现了开源软件 Linux 和开源硬件 Arduino 生态系统的完美结合。

pcDuino 支持 Ubuntu Linux 和 Android 4.0 ICS，采用标准的 Android SDK，支持 Python、C 语言、Java 等编程语言。pcDuino 的出厂默认设置是将 Ubuntu 安装在 NAND flash 里，Ubuntu 可以从 NAND flash 启动，或者通过可启动的 Micro SD 卡启动。用户可以根据自己的需要将 pcDuino 的系统更新为最新的系统。

pcDuino & Ubuntu 支持 Ubuntu Linux 12.04 版，该版本是为在具有 DRAM 限制并且支持 NAND flash 的 ARM Cortex 平台上运行而优化的定制版。设备可以通过支持 USB 接口的鼠标和键盘来操作，具体的支持程序清单如表 1-4 所示。

表 1-4 Ubuntu Linux 12.04 版的支持程序

项 目	详 细 说 明
终端	从桌面上使用内置 LXTerminal 应用程序 启动终端应用程序，运行标准 Linux 指令或者 Vi editor 等 启动 C/C++ 编译（使用 gcc），安装并执行
文件浏览器	从桌面上使用内置管理 用于典型的文件管理
互联网浏览器	在桌面上使用 Google 浏览器 支持 HTML5
Office	使用桌面上的 Document Viewer 文件 使用桌面上的 Gnumeric 文件 使用桌面上的 AbiWord 文件
视频播放器	从桌面启动 MPlayer 可以回放音频、视频和图像
服务器	VNC 服务器 SSH
开发	Arduino IDE for pcDuino

注：系统管理员和密码都是 ubuntu。

pcDuino 为支持 HD-TV 的输出显示设备对 Android 4.0 进行优化，形成定制版 Android 4.0。此显示设备需要通过支持 USB 接口的鼠标和键盘来操作，具体支持的部分程序清单如表 1-5 所示。

表 1-5 Android 4.0 定制版支持的部分程序

项 目	详 细 说 明
文件浏览器	使用内置文件管理 用于典型的文件管理
互联网浏览器	使用桌面上的网络浏览器或者 Google 搜索精灵 支持 HTML5
日历	Google 日历
Gmail	访问 Gmail 账户

(续)

项 目	详 细 说 明
联系人	Google 联系人 App
App 商店	使用 Google play store 下载第三方 Apps
音乐	Android 内置音乐播放器
视频播放器	内置超高清播放器支持用户播放本地文件

## 第 2 章

# Linux 系统详解

本章主要讲解 Linux 的相关命令和操作。

### 2.1 系统简介

操作系统是计算机必不可少的重要组成部分，只要使用计算机就一定会涉及操作系统。操作系统的功能用一句话来表述就是管理与控制计算机资源的软件。这里提到的“计算机资源”包括计算机硬件资源和软件资源。目前比较流行的操作系统包括 UNIX 系统、类 UNIX 系统、Windows，以及一些嵌入式的操作系统。目前用户数量比较多的可能就是 Windows 系统以及属于类 UNIX 的 Linux 系统。

Linux 系统的成长和发展就像一部小说一样有趣。在 1991 年，由于当时的 UNIX 厂商对 UNIX 源代码的限制，作为芬兰赫尔辛基大学学生的 Linus Torvalds 决定开发自己的操作系统。他首先编写了一个简单地终端仿真程序，用来连接到自己学校的大型 UNIX 系统上，在这个终端程序上经过了一年的开发、改进和完善，终于开发出了一个虽然没有 UNIX 功能那么完善和强大，但却是一个五脏俱全的类似 UNIX 的操作系统。在 1991 年底，他在网上发布了这个操作系统，并被命名为 Linux。

Linux 的设计都源于 UNIX 的设计，实现了 UNIX 操作系统的 API。与其他的类 UNIX 操作系统不同，Linux 系统并不是直接修改 UNIX 系统源代码而来，而是对 UNIX 系统的重新实现。

Linux 操作系统从发布之日起，就受到了很多人的追捧，其中一个非常重要的因素就是 Linux 允许其他的开发者对其代码自由地进行修改和完善。Linux 从诞生到现在经历了 20 多年，已经被广泛移植到各种硬件体系结构之上。

Linux 一诞生就决定了它光明的前途，它由互联网上的各个开发小组合作完成，每个人都可以向 Linux 提交代码，只要经过审核确定提交的代码符合规范，就可以成为 Linux 源代码的一部分。

Linux 的用途非常广泛，大到各种计算机集群，小到手机，甚至在手表中都可以看到 Linux 的存在。Linux 所支持的工具也非常完善，基本上各种工作都可以在 Linux 系统上建立并完成。

## 2.2 基础命令

### 2.2.1 cd 和 ls 命令

#### 1. 打开终端

和 Windows 一样，Linux 的各种发行版也一样实现了图形界面，但如果想随心所欲地使用 Linux，还需要了解和学习 Linux 的各种命令。

首先需要打开一个终端，终端是输入命令并查看命令执行结果的地方。

在 Ubuntu 系统上打开终端的方法是按组合键 `Ctrl+Alt+T`。如果这个方法不行的话，就先按 `Alt` 键，然后在光标提示处输入 `terminal`，按回车键后就会看到终端被打开了，如图 2-1 所示。

打开终端时出现的提示信息为 `author@asus-K43SJ: ~ $`。其中 `author` 表示当前登录系统的用户名，`asus-K43SJ` 是计算机名，图 2-1 中显示的是默认的计算机名字。冒号后面的 `~` 表示工作在当前用户的家目录之下（关于家目录后续内容会进行讲解），`$` 表示当前用户是普通用户，不是超级用户。

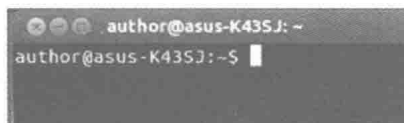


图 2-1 终端界面

#### 2. cd 命令

`cd` 命令的全称是 `change directory`，就是改变目录的意思。在改变目录之前首先需要了解当前的目录，如图 2-1 中当前用户的家目录，可以使用命令 `pwd` 来查看，如图 2-2 所示。

从图 2-2 可以看出当前用户的家目录为 `/home/author`。目录的更改可使用 `cd` 命令，如图 2-3 所示为将任一目录更改到根目录下。

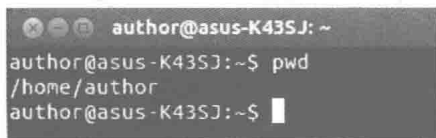


图 2-2 `pwd` 命令

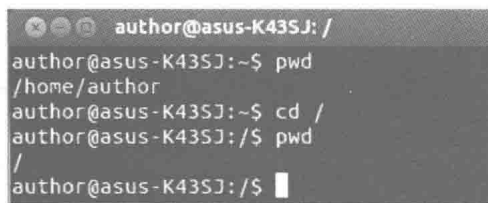


图 2-3 `cd /` 命令

使用 `pwd` 命令可查看当前的工作目录，之后使用 `cd /` 切换到根目录下。需要注意的是，根目录使用斜杠符号 `/` 来表示。最后再使用 `pwd` 查看当前的工作目录，发现目录已经切换到根目录之下了，同时冒号后面的 `/` 也暗示当前的工作目录是要根目录。

如果对已经切换的目录进行恢复操作需要使用命令 `cd -`，如图 2-4 所示。

如果需要将目录切换到包含当前目录的上层目录，则使用命令 `cd ..`，“`..`”表示当前工作目录的上层目录。比如当前目录为 `/home/author`，要切换到 `/home` 目录下，操作如图 2-5 所示。

```

author@asus-K43SJ: ~
author@asus-K43SJ:~$ pwd
/home/author
author@asus-K43SJ:~$ cd /
author@asus-K43SJ:/ $ pwd
/
author@asus-K43SJ:/ $ cd -
/home/author
author@asus-K43SJ:~$ pwd
/home/author
author@asus-K43SJ:~$

```

图 2-4 cd - 命令

```

author@asus-K43SJ: /home
author@asus-K43SJ:~$ pwd
/home/author
author@asus-K43SJ:~$ cd ..
author@asus-K43SJ:/home$ pwd
/home
author@asus-K43SJ:/home$

```

图 2-5 cd .. 命令

需要注意的是，根目录 / 是系统的最顶层目录。如果在根目录下执行 “cd ..” 命令的话，当前工作目录不会发生改变，如图 2-6 所示。

### 3. ls 命令

这个命令是显示目录内容。在终端中输入命令 ls，显示的结果如图 2-7 所示。

```

author@asus-K43SJ: /
author@asus-K43SJ:/ $ pwd
/
author@asus-K43SJ:/ $ cd ..
author@asus-K43SJ:/ $ pwd
/
author@asus-K43SJ:/ $

```

图 2-6 在根目录下执行 cd .. 命令

```

author@asus-K43SJ: ~
author@asus-K43SJ:~$ ls
examples.desktop
author@asus-K43SJ:~$

```

图 2-7 ls 命令

说明在当前用户的家目录之下只有一个叫作 example.desktop 的文件。切换到根目录之下，再使用 ls 命令查看一下，如图 2-8 所示。

```

author@asus-K43SJ: /
author@asus-K43SJ:/ $ ls
bin          error.3025  initrd.img  lib64       media      root       srv         var
boot         error.3065  initrd.img.old  libnss3.so  mnt        run        sys         vmlinuz
cdrom        etc         lib         log          opt         sbin       tmp         vmlinuz.old
dev          home       lib32       lost+found  proc        selinux    usr
author@asus-K43SJ:/ $

```

图 2-8 ls 命令

图 2-8 表示根目录下有许多的文件和子目录，而且颜色各不相同，每种颜色代表不同的文件种类，通过 ls -l 命令可以查看不同颜色代表文件种类的说明，如图 2-9 所示。

```

author@asus-K43SJ:/ $ ls -l
total 124
drwxr-xr-x  2 root root  4096 Mar  3 09:12 bin
drwxr-xr-x  3 root root  4096 Jul 15 11:51 boot
drwxr-xr-x  2 root root  4096 Dec 13 2013 cdrom
drwxr-xr-x 16 root root 4340 Aug 11 14:08 dev
-rw-r--r--  1 root root    2 Jul 15 22:03 error.3025
-rw-r--r--  1 root root    2 Jul 15 22:03 error.3065
drwxr-xr-x 153 root root 12288 Aug 11 08:51 etc

```

图 2-9 ls -l 命令

```

drwxr-xr-x  5 root root 4096 Aug 10 17:20 home
lrwxrwxrwx  1 root root  33 Mar 19 09:05 initrd.img -> /boot/initrd.img-3.8.0-38-generic
lrwxrwxrwx  1 root root  33 Mar  3 09:16 initrd.img.old -> /boot/initrd.img-3.8.0-37-generic
drwxr-xr-x 23 root root 4096 Apr 19 10:25 lib
drwxr-xr-x  2 root root 4096 Dec 23 2013 lib32
drwxr-xr-x  2 root root 4096 Dec 18 2013 lib64
lrwxrwxrwx  1 root root  36 Mar 19 09:06 libnss3.so -> /usr/lib/x86_64-linux-gnu/libnss3.so

```

图 2-9 (续)

第一行输出 124 表示该目录的大小为 124KB。后面 4 行根据第一个字母 d 可判断出所列文件为目录类型；接下来两行由“-”判断出所列文件为普通的文件；以 l 开头的行表示该文件是一个符号链接，类似于 Windows 中的快捷方式。

## 2.2.2 touch 和 mkdir 命令

### 1. touch 命令

touch 命令用于新建一个空的文件，如图 2-10 所示为使用 touch 命令创建一个名为 myfile 的文件。

新建的文件 myfile 是一个空文件，没有添加内容。可以使用 cat 命令来查看文件中的内容，如图 2-11 所示。

```

author@asus-K43SJ:~$ ls
examples.desktop
author@asus-K43SJ:~$ touch myfile
author@asus-K43SJ:~$ ls
examples.desktop  myfile
author@asus-K43SJ:~$

```

图 2-10 touch 命令

```

author@asus-K43SJ:~$ cat myfile
author@asus-K43SJ:~$

```

图 2-11 cat 命令

命令结果显示文件 myfile 的内容为空。向文件中写入内容或者编辑文件，需要借助文本编辑器，例如 vim、gedit 等。

### 2. mkdir 命令

mkdir 命令用于新建一个目录，如图 2-12 所示为新建了一个名为 mydir 的目录。

```

author@asus-K43SJ:~$ ls -l
total 12
-rw-r--r--  1 author asus 8445 Apr 17 2012 examples.desktop
-rw-r--r--  1 author asus  0 Aug 11 17:05 myfile
author@asus-K43SJ:~$ mkdir mydir
author@asus-K43SJ:~$ ls -l
total 16
-rw-r--r--  1 author asus 8445 Apr 17 2012 examples.desktop
drwxr-xr-x  2 author asus 4096 Aug 11 17:15 mydir
-rw-r--r--  1 author asus  0 Aug 11 17:05 myfile
author@asus-K43SJ:~$

```

图 2-12 mkdir 命令

可以改变路径到新建的目录 mydir 下，继续执行新建文件和子目录等操作。

## 2.2.3 rm 和 rmdir 命令

### 1. rm 命令

rm 命令用于删除指定的文件，例如删除上节新建的 myfile 文件，使用的命令如图 2-13 所示。

可以看到经过 rm 命令之后，myfile 文件已经被删除。rm -r 命令用来删除目录，如图 2-14 所示为删除上节所创建的目录 mydir。

```
author@asus-K43SJ:~$ ls
examples.desktop mydir myfile
author@asus-K43SJ:~$ rm myfile
author@asus-K43SJ:~$ ls
examples.desktop mydir
author@asus-K43SJ:~$
```

图 2-13 rm 命令

```
author@asus-K43SJ:~$ ls
examples.desktop mydir
author@asus-K43SJ:~$ rm -r mydir
author@asus-K43SJ:~$ ls
examples.desktop
author@asus-K43SJ:~$
```

图 2-14 rm -r 命令

### 2. rmdir 命令

rmdir 是专门执行删除目录操作的命令，如图 2-15 所示为先新建一个目录，然后使用 rmdir 命令删除它。

```
author@asus-K43SJ:~$ mkdir test_dir
author@asus-K43SJ:~$ ls
examples.desktop test_dir
author@asus-K43SJ:~$ rmdir test_dir
author@asus-K43SJ:~$ ls
examples.desktop
author@asus-K43SJ:~$
```

图 2-15 rmdir 命令

## 2.2.4 cp 和 mv 命令

### 1. cp 命令

cp 命令用于将一个已经存在的文件中的内容复制到另一个文件中，或者复制一个目录。例如，将在上节中建立的文件 myfile 中的内容复制到另一个文件 myfile\_copy 文件中，命令的使用如图 2-16 所示。

如图 2-17 所示为对目录的复制方法。新建目录 dir，在 dir 目录中新建一个文件 file\_in\_dir 和一个子目录 subdir\_in\_dir，将 dir 目录下的内容复制到 dir\_copy 目录中。

```
author@asus-K43SJ:~$ ls
examples.desktop myfile
author@asus-K43SJ:~$ cp myfile myfile_copy
author@asus-K43SJ:~$ ls
examples.desktop myfile myfile_copy
author@asus-K43SJ:~$ cat myfile
I like Linux!
He love Linux,too!
author@asus-K43SJ:~$
```

图 2-16 复制文件 myfile

```
author@asus-K43SJ:~$ ls
dir examples.desktop myfile myfile_copy
author@asus-K43SJ:~$ ls dir
file_in_dir subdir_in_dir
author@asus-K43SJ:~$ cp -r dir dir_copy
author@asus-K43SJ:~$ ls dir_copy
file_in_dir subdir_in_dir
author@asus-K43SJ:~$
```

图 2-17 复制目录 dir

### 2. mv 命令

mv 命令用和 cp 命令类似，但是执行的操作是移动而不是复制。对于图 2-16 和图 2-17 的演示，将 cp 命令换成 mv 命令，执行效果如图 2-18 和图 2-19 所示。

mv 命令的另一个作用就是给文件更名，使用方法如图 2-19 所示，将原文件 myfile 移动



并更名为 myfile\_move。

```
author@asus-K43SJ:~$ ls
dir  examples.desktop  myfile  myfile_copy
author@asus-K43SJ:~$ ls dir
file_in_dir  subdir_in_dir
author@asus-K43SJ:~$ cp -r dir dir_copy
author@asus-K43SJ:~$ ls dir_copy
file_in_dir  subdir_in_dir
author@asus-K43SJ:~$
```

图 2-18 移动文件

```
author@asus-K43SJ:~$ ls
examples.desktop  myfile
author@asus-K43SJ:~$ mv myfile myfile_move
author@asus-K43SJ:~$ ls
examples.desktop  myfile_move
author@asus-K43SJ:~$
```

图 2-19 移动目录

## 2.2.5 find 和 awk 命令

在底层的开发过程中，使用查找相关命令进行关键字、正则表达式等的查找匹配，对于整个流程的编写是很有帮助的。

在源码的学习中，提倡使用 `cscope+vim` 的方式进行代码的查看，这个方法对于源码中的函数调用、宏变量等可以得到具体的输出，方便进行浏览。而对于源码中相关模块和驱动在进行编写时建议使用 `find`、`awk` 这两个命令，可以达到事半功倍的效果。因此在本节，对这两个命令进行详细讲解。

### 1. find 命令

Linux 下的文件表达格式非常复杂，对于 Linux 新手而言，`find` 命令是了解和学习 Linux 文件特点的最佳方式之一。因为 Linux 发行版众多，各个发行版的升级很快，对于曾经熟悉的 Linux 中的某个配置文件所在位置，有时也无法按图索骥地找到。

虽然可以使用 `locate` 命令进行文件的查找，但仅仅是进行模糊匹配。`locate` 命令是对生成的数据库进行遍历，生成数据库的命令为 `updatedb`。这一特性决定了用 `locate` 查找文件的速度很快，但是相应的精确度就会差点。所以，这里重点介绍 `find` 命令。

若想查找系统配置文件 `fstab` 在哪个目录下，可以切换到根目录，并使用如图 2-20 所示命令。

图 2-20 中查找命令为 `find/-name fstab`。经过查找，输出结果显示 `fstab` 文件在 `/etc/` 目录下，如图 2-20 最后一行所示。

图 2-21 演示的是对不完整的文件名进行的查找。查找在 `/etc` 目录下所有的包含 `wireless` 字样的文件。

由图 2-21 可见，输出结果中有 `Permission denied` 的目录。产生这一输出的原因是，在 Linux 系统中，`find` 命令是所有系统用户都可以使用的命令，并不是 ROOT 系统管理员的专有权力。一旦 Linux 系统上系统管理员 ROOT 设定文件目录的权限为禁止访问，则 `find` 命令无法对这些文件进行读操作，就会出现 `Permission denied` 的字样。这时候通过特殊文件 `/dev/null`，可以重定向这些错误信息到这个文件，避免它们显示到输出中。`/dev/null` 文件用于表明空的或者错误的信息。

```

favorming@favorming-Lenovo:~$ find / -name fstab
/usr/share/doc/mount/examples/fstab
find: '/usr/share/doc/google-chrome-stable': Permission denied
find: '/lost+found': Permission denied
find: '/var/log/speech-dispatcher': Permission denied
find: '/var/log/gdm': Permission denied
find: '/var/cache/ldconfig': Permission denied
find: '/var/spool/rsyslog': Permission denied
find: '/var/spool/cron/crontabs': Permission denied
find: '/var/spool/cups': Permission denied
find: '/var/spool/exim4': Permission denied
find: '/var/lib/polkit-1': Permission denied
find: '/var/lib/sudo': Permission denied
find: '/var/lib/spamassassin/sa-update-keys': Permission denied
find: '/var/lib/spamassassin/.spamassassin': Permission denied
find: '/var/lib/gdm': Permission denied
find: '/var/lib/udisks2': Permission denied
find: '/run/udisks2': Permission denied
find: '/run/exim4': Permission denied
find: '/run/gdm': Permission denied
find: '/run/wpa_supplicant': Permission denied
find: '/run/cups/certs': Permission denied
find: '/home/lost+found': Permission denied
find: '/home/favorming/.cache/dconf': Permission denied
find: '/home/favorming/.dbus': Permission denied
/home/favorming/Desktop/git/kernel_source/android4.2/lichee/buildroot/fs/skeleton/etc/fstab
/home/favorming/Desktop/git/pcDuino/android4.2/lichee/buildroot/fs/skeleton/etc/fstab
find: '/home/favorming/.config/enchant': Permission denied
find: '/home/favorming/.gvfs': Permission denied
find: '/etc/ssl/private': Permission denied
find: '/etc/polkit-1/localauthority': Permission denied
/etc/fstab

```

图 2-20 find 命令查找 fstab 文件

```

favorming@favorming-Lenovo:~$ cd /etc/
favorming@favorming-Lenovo:/etc$ find ./ -name '*wireless*'
./network/if-post-down.d/wireless-tools
./network/if-pre-up.d/wireless-tools
find: './ssl/private': Permission denied
find: './polkit-1/localauthority': Permission denied
find: './cups/ssl': Permission denied
./acpi/ibm-wireless.sh
./acpi/events/tosh-wireless
./acpi/events/ibm-wireless
./acpi/events/asus-wireless-off
./acpi/events/asus-wireless-on
./acpi/asus-wireless.sh
./acpi/tosh-wireless.sh

```

图 2-21 find 命令查找带有 wireless 字样的文件

使用如下命令，可以方便地根据文件的特征进行文件查找。

```

find / -amin -10          #查找系统最后10分钟内访问的文件
find / -group favorming   #查找系统中属于favorming group的文件
find / -mmin -5          #查找系统中最后5分钟内修改过的文件

```

find 命令可以通过递归的方式使用，以便对一个文件夹中的每个子文件夹所包括的文件进行信息匹配。当然通过指定参数 `-maxdepth` 选项，可以对这个递归查找方式的深度进行限制。

图 2-22 演示了多个命令一起使用的状况，例如对于文件的查看操作。用户通常希望在知道路径的情况下同时看到它的属性，这时候可以采取这种方式。

```

favorming@favorming-Lenovo:/etc$ find ./ -name '*source*' -ls
1311330  4 drwxr-xr-x  2 root   root    4096 11月 14 16:03 ./apt/sources.list.d
1310919  4 -rw-rw-r--  1 root   root    3135  8月  2 12:16 ./apt/sources.list~
1312336  4 -rw-rw-r--  1 root   root    881  1月  5 09:43 ./apt/sources.list
1313356  4 -rw-rw-r--  1 root   root    881 11月 14 15:58 ./apt/sources.list.save
1310971  4 -rw-r--r--  1 root   root    878  3月 31 2010 ./X11/Xsession.d/30x11-c
1310953  4 drwxr-xr-x  2 root   root    4096 12月 11 11:38 ./X11/Xresources
find: './ssl/private': Permission denied
find: './polkit-1/localauthority': Permission denied
find: './cups/ssl': Permission denied
1312048  4 -rw-r--r--  1 root   root    387  5月 26 2012 ./ghostscript/cidmap.d
1312045  4 -rw-r--r--  1 root   root    475  5月 26 2012 ./ghostscript/cidmap.d/
1312049  4 -rw-r--r--  1 root   root    730  5月 26 2012 ./ghostscript/cidmap.d/
1312046  4 -rw-r--r--  1 root   root    504  5月 26 2012 ./ghostscript/cidmap.d/
1312047  4 -rw-r--r--  1 root   root    546  5月 26 2012 ./ghostscript/cidmap.d/

```

图 2-22 find 命令查找路径及查看文件属性

命令运行结果将所有包含有 source 的文件及其相关的属性都显示了出来。

## 2. awk 命令

awk 名称来源于它的 3 位创始人 Alfred Aho、Peter Weinberger、Brian Kemighan 的姓氏的首字母。相对于 grep 的查找，awk 命令具有更强大的功能，是一个强大的文本分析工具。

但更多时候 awk 是一种进行文本分析的编程语言，需要由用户自定义函数和动态正则表达式进行处理。

下文将介绍一些关于 awk 命令的使用。

awk 命令打印出 vivado.log 文件中所有包含 set 的行的内容，如图 2-23 所示。

```

favorming@favorming-Lenovo:~$ awk '/set/' vivado.log
# set version "2013.4"
# set carversion "v1"
# set project_name "zrobot_${carversion}_${version}"
# set part "xc7z020clg484-1"
# set bd_name "system"
# set board "em.avnet.com:zynq:zed:d"
# set_property board em.avnet.com:zynq:zed:d [current_project]
# set_property ip_repo_paths ../ipcores [current_fileset]
## set scripts_vivado_version 2013.4
## set current_vivado_version [version -short]
## set design_name system
## set errMsg ""
## set nRet 0
## set cur_design [current_bd_design -quiet]
## set list_cells [get_bd_cells -quiet]
## set errMsg "ERROR: Design <$design_name> already exists in your project"
## set nRet 1
## set errMsg "ERROR: Design <$design_name> already exists in your project"
## set nRet 3
## set parentCell [get_bd_cells /]
## set parentObj [get_bd_cells $parentCell]
## set parentType [get_property TYPE $parentObj]
## set oldCurInst [current_bd_instance .]

```

图 2-23 awk 命令匹配字符串



```
## cur_design
## list_cells
## errMsg
## nRet
## errMsg
## nRet
## parentCell
## parentObj
## parentType
## oldCurInst
## DDR
## FIXED_IO
## GPIO
## GPIO_0
## pwm_out
## Create
## AXI_PWM_0
## Create
## axi_gpio_0
## -dict
## Create
## processing_system7_0
## -dict
```

图 2-25 (续)

关于 `awk` 命令，更多的时候是将其看作一门编程语言进行操作，以实现更多更复杂的功能。若再加入正则表达式和函数定义，其功能将更加强大。

### 2.2.6 vim 编辑器的使用

`vim` 编辑器是迄今为止公认的最优秀的文本编辑工具之一，被广泛地应用在各种操作系统平台之上。`vim` 编辑器的安装步骤如下：

- 1) 打开一个终端，输入 `sudo apt-get install vim`。
- 2) 输入用户密码，开始进入安装环节，如图 2-26 所示。

```
author@asus-K43SJ:~$ sudo apt-get install vim
[sudo] password for author:
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  vim-doc vim-scripts
The following NEW packages will be installed:
  vim
0 upgraded, 1 newly installed, 0 to remove and 234 not upgraded.
```

图 2-26 安装 vim

安装完成之后，就可以使用 `vim` 对文件进行编辑。新建一个文件 `myfile`，使用命令 `vim myfile`，按回车键，进入编辑器。在编辑器中，输入 `i` 或者 `a` 命令进入编辑模式，屏幕左下角有 `INSERT` 提示，在里面输入以下内容：

```
I Like Linux!
He loves Linux,too!
```

操作界面如图 2-27 所示。

编辑完成之后，按下 Esc 按键，输入冒号，再输入 wq 命令保存并退出。

关于 vim 编辑器的详细使用方法，请参考相关的书籍。



图 2-27 编辑 myfile

## 2.3 Bash Shell

### 2.3.1 Bash Shell 简介

#### 1. 认识 Shell

计算机系统的软件和硬件资源是由操作系统进行管理的，操作系统提供一组接口与用户进行交互。Shell、操作系统和硬件的关系如图 2-28 所示。

用户在 Shell 中输入命令，Shell 对这个命令进行解释，并向操作系统发出相应的请求，操作系统再驱动硬件执行相应的操作。由此可见，Shell 是计算机系统和硬件设备之间的中间介质，是一个系统工具。学习 Shell 不仅仅是为了操作 Linux 系统，还是为了理解 Linux 的运行机制。



图 2-28 Shell 在计算机系统的位置

#### 2. 认识 Bash Shell

Linux 提供了很多 Shell，包括 C Shell、Korn Shell 等，而 Bash Shell 是 Linux 预设的 Shell。Bash 是 GNU 计划中非常重要的工具软件之一，是目前 Linux 系统的标准 Shell。Bash 的命令语法和其他 Shell 的语法非常相似。

Bash Shell 具有命令自动补齐功能。在输入命令时，只需要输入命令的前面部分，然后按 Tab 键就可以自动将命令补齐。

Bash Shell 可以将所要执行的命令写入一个称为脚本的文件里，可使所有的命令与这个文件一起被执行。

### 2.3.2 Bash Shell 脚本简介

编写 Shell 脚本是使用 Linux 必备的重要技能之一，使用 Shell 脚本可以极大地简化代码编写工作，让一部分系统的管理任务自动化。Shell 脚本是将一系列的命令写到一个文件中，并赋予这个文件可执行的权限，类似于 Windows 中的批处理文件。

例如：编写一个简单的 Shell 脚本，打开终端，使用 vim 编辑器新建一个 create\_dir.sh 文件，在里面输入如下代码：

代码清单2-1: Shell脚本运行实例

```
#!/bin/bash
mkdir level1
```

```
cd level1
mkdir level2
echo "finished creating directory"
```

保存并退出后，使用 `chmod a+x create_dir.sh` 命令来为 `create_dir.sh` 文件添加执行权限。使用 `./create_dir.sh` 运行这个脚本。输出结果为 `finished creating directory`，表明脚本已经执行完成。使用 `ls` 命令可以在当前的目录中查看新建的 `level1` 目录，使用 `cd` 命令进入 `level1` 目录会看到里面还有包含有 `level2` 目录。

## 2.4 Linux 源码与 Android 源码介绍

### 2.4.1 Linux 源码简介

Linux 开放源代码的最大好处就是给操作系统研究者提供了一个可以深入探索操作系统的机会。现在的操作系统非常庞大，若想把全部的 Linux 系统源代码从头到尾读一遍几乎是不可能的，因此首先需要了解 Linux 源代码的构造，在对整个系统有一个全面的了解后，再去了解相应的 Linux 源码。

Linux 的源代码目录及其说明如表 2-1 所示。

表 2-1 Linux 内核源码目录及说明

目 录	说 明	目 录	说 明
arch	与特定的硬件体系结构相关的代码	kernel	各种核心子系统
block	块设备	lib	库文件
crypto	加密 API	mm	内存管理子系统
Documentation	内核源代码的文档	net	网络管理子系统
drivers	设备驱动程序	samples	示例代码
firmware	驱动程序所需要的固件	scripts	编译内核所需要的脚本
fs	文件系统	security	与 Linux 安全相关的代码
include	内核头文件	sound	与语音相关的代码
init	与内核的初始化有关	tools	在 Linux 开发中用到的工具
ipc	进程间通信	virt	与虚拟化相关代码

用户可以根据需要去相应的目录中查阅所需的内核代码。如果读者想真正的了解 Linux 的工作机制，阅读 Linux 源代码是个好方法，它是一部学习计算机技术最好的“教科书”。

### 2.4.2 Android 源码简介

Android 系统是对 Linux 内核源代码的修改，Android 的内核源代码目录及其说明如表 2-2 所示。

表 2-2 Android 内核源码目录及说明

目 录	说 明
bionic	C 运行时支持，包括 libc、libm、libdl 以及动态 linker
bootloader/legacy	内核加载器
build	Build 系统
dalvik	Dalvik 虚拟机
development	上层开发和调试工具
framework/base	Android 核心的框架库
framework/policies/base	框架配置策略
hardware/libhardware	硬件抽象层库
hardware/ril	无线接口层
kernel	Linux 内核
prebuilt	预编译内核
system/core	最小化可启动环境
system/extras	底层调试和检查工具



## 第 3 章

# Android 系统开发环境搭建

本章主要讲解 Android 系统开发环境，介绍环境搭建的整个过程，让读者了解 Android 系统的开发环境。

### 3.1 编译前奏——Android 上的开发工作

#### 3.1.1 Android 的移植开发

Android 系统移植的主要目的是为了能在特定的硬件上运行 Android 系统。而在移植的过程中，一个重要的方面就是把握关键点，减少工作量。首先要熟悉硬件抽象层的接口，其次要集成和复用已有的驱动程序，主要的工作量集中在对硬件抽象层的实现中。为了更好地理解和调试系统，应该适当地对硬件抽象层的调用情况有所了解。

移植过程中主要的工作可分为两个部分：Linux 驱动；Android 系统硬件抽象层。

Linux 中的驱动工作在系统内核空间，Android 系统硬件抽象层工作在用户空间，有了这两个部分的结合，就可以让庞大的 Android 系统运行在特定的硬件平台上。在有了特定的硬件系统之后，需要在 Linux 中实现该硬件的驱动程序，这些驱动程序都是 Linux 的标准驱动程序，在 Android 平台和其他 Linux 平台上基本是相同的。

工作主要集中在 Android 系统中的硬件抽象层，硬件抽象层向下调用 Linux 中的驱动程序，向上提供接口，供 Android 系统之外的其他部分调用。

在 Android 系统中需要移植的主要包含以下内容：

- 显示部分
- 用户输入部分
- 多媒体编解码
- 3D 加速器部分
- 音频部分
- 视频输出部分
- 摄像头部分

- 电话部分
- GPS
- WiFi
- 蓝牙
- 传感器部分
- 振动器部分
- 背光部分
- 实时时钟
- 电池部分

Android 系统中包括很多组件，并不是每一个组件都需要移植。对于一些纯软件的组件，就没有移植的必要。对于诸如浏览器引擎等一些组件，虽然需要下层网络的支持，但是并非直接为其移植网络接口，而是通过无线局域网或电话系统数据的连接来完成标准的网络接口，这类组件也不需要移植。

Android 的移植可以分成以下几个主要的类型：基本图形用户界面（GUI）部分，包括显示部分和用户输入部分；与硬件相关的加速部分，包括媒体编解码和 OpenGL；音视频输入输出环节，包括音频、视频输出和摄像头部分；连接部分，包括无线局域网、蓝牙、GPS；电话部分；附属部件，包括传感器、背光、振动器等。

除此之外，电源管理也是一个非常重要的移植组件，它和 Android 系统的各个子系统都有关系。对于大部分子系统，硬件抽象层和驱动程序都需要根据实际系统的情况实现，例如传感器部分、音频部分、视频部分、摄像头部分、电话部分。也有一些子系统，硬件抽象层是标准的，只需要实现 Linux 内核中的驱动程序即可，例如输入部分、振动器部分、无线局域网部分、蓝牙部分等。对于有标准的硬件抽象层的系统，有的时候通常也需要做一些配置工作。

### 3.1.2 系统开发

Android 系统的开发主要就是中间件的开发。由于 Google 已经提供了绝大部分的中间件，因此系统开发的工作主要集中在对 JNI 库的开发。JNI 是 Java 应用程序访问其他语言，尤其是中低级语言的接口。通过 JNI，用户可以把对性能要求高的或者与操作系统底层直接打交道的程序封装在一个库文件中，通过 JNI 机制进行调用。JNI 的实现流程如图 3-1 所示。

如果是针对通用的 Android 设备开发，比如手机、平板电脑等，一般把电源管理、传感器等系统独有设备的用户态驱动程序放在 JNI 中，由上层的电源管理应用调用。除此之外，高级视频编解码算法、压缩解压等 Android 自

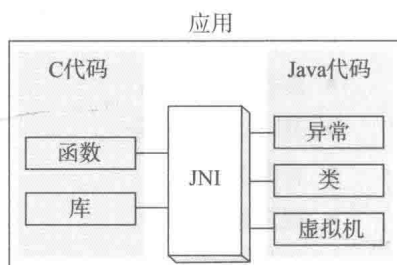


图 3-1 JNI 实现流程

身不提供的算法，也会被封装在 JNI 中。

如果针对专用的 Android 设备开发，比如视频监控设备、图像处理设备等，一般会把 OpenCV 等图像处理的算法封装在 JNI 模块中。

### 3.1.3 应用开发

应用程序是直接为用户打交道的程序。因为 Android 的分层特性，应用程序不与硬件或操作系统直接沟通。相反地，应用程序只与 Android 系统内的 Java 类和 JNI 库沟通，因而拥有良好的可移植性，能做到一次开发，处处运行。

开发 Android 应用程序需要使用 Android SDK，该 SDK 可以从 Android 的官方网站下载。如果要使用特殊的 JNI 库或针对特定手机开发的 Java 类库，则使用对应型号的手机 SDK，或自行编译 SDK。需要注意的是，非官方 SDK 开发的应用程序可能无法在其他平台上运行，因而推荐使用官方 SDK。仅当为特定机型开发时才应使用自定义 SDK。如果需要特定的 JNI 库，可以使用 Android NDK，并将 NDK 编译生成的 .so 文件封装到 .apk 文件中。注意，.so 文件不能跨平台，因此要为不同的平台编译不同的 .so 文件。

本书仅介绍 Android 系统底层的移植与开发，并不涉及 Android 应用开发。

## 3.2 Android 的系统架构

### 3.2.1 软件结构

#### 1. Linux 操作系统及驱动

Android 系统运行于 Linux kernel 之上，但并不等同于 GNU/Linux。通常 GNU/Linux 里支持的功能，Android 系统大都不支持，包括 Cairo、X11、Alsa、FFmpeg、GTK、Pango 及 Glibc 等都被 Android 系统移除掉了。

Android 系统中以 Bionic 取代 Glibc，以 Skia 取代 Cairo，以 OpenCORE 取代 FFmpeg。Android 为了达到商业应用，必须移除被 GNU GPL 授权证所约束的部分，例如 Android 系统将驱动程序移到 Userspace，使得 Linux drive 与 Linux kernel 彻底分开。Bionic/Libc/Kernel/并非标准的 Kernel header files。Android 系统的 Kernel header 是利用工具由 Linux Kernel header 所产生的，目的是为了保留常数、数据结构与宏。Android 系统的 Linux kernel 控制包括安全、存储器管理、程序管理、网络堆栈、驱动程序模型等。

#### 2. Android 系统程序库

操作系统与应用程序之间的沟通桥梁应分为两层：函数层和虚拟机。Bionic 是 Android 对 glibc 改良的版本。同时包含了 Webkit 和 Surface flinger。所谓的 Webkit 就是 Apple Safari 浏览器背后的引擎。Surface flinger 是将 2D 或 3D 的内容显示到屏幕上。Android 使用的工具链 (Toolchain) 为 Google 自制 Bionic Libc，采用 OpenCORE 作为基础多媒体框

架。Open CORE 可分 7 块：PVPlayer、PVAuthor、Codec、PacketVideo Multimedia Framework、Operating System Compatibility Library、Common、OpenMAX。使用 Skia 为核心图形引擎，搭配 OpenGL/ES，并采用 SQLite 数据库系统为多媒体数据库。Skia 与 Linux Cairo 功能相当，相较于 Linux Cairo，Skia 还只是雏形。2005 年，Skia 公司被 Google 收购，2007 年初，Skia GL 源码被公开，Skia 也成为了 Google Chrome 的图形引擎。

Android 的中间层多以 Java 实现，并且采用特殊的 Dalvik 虚拟机。Dalvik 虚拟机是一种“暂存器形态”的 Java 虚拟机，变量皆存放于暂存器中，虚拟机的指令相对较少。Dalvik 虚拟机可以有多个实例，每个 Android 应用程序都由一个自属的 Dalvik 虚拟机来运行，让系统在运行程序时可达到优化。Dalvik 虚拟机并非运行 Java 字节码，而是运行一种称为 .dex 格式的文件。

Android 系统的硬件相关操作被封装到硬件抽象层之中。Android 的 HAL 能以封闭源码的形式提供硬件驱动模块。HAL 的目的是为了把 Android framework 与 Linux kernel 隔开，让 Android 不至于过度依赖 Linux kernel，以达成 Kernel independent 的概念，也让 Android framework 的开发能在不考虑驱动程序实现的前提下发展。

HAL Stub 是一种代理人的概念，Stub 以 \*.so 文件的形式存在。Stub 向 HAL “提供”操作函数，并在 Android 运行时向 HAL 取得 Stub 的操作，再调用这些操作函数。HAL 里包含了许多的 Stub。Runtime 只要说明类型，即 Module ID，就可以取得操作函数。

有些应用程序需要进行大量的数据计算。这种情况下 Java 的工作效率就会成为应用程序性能的瓶颈。为了使用 C/C++ 语言编写的高效函数库，Java 提供了 JNI 接口。从 Java1.1 开始，Java Native Interface (JNI) 标准成为 Java 平台的一部分，它允许 Java 代码与用其他语言编写的代码进行交互。

### 3. Android Java 程序库

Android 系统提供了如下的 Java 库。

android.util 涉及系统底层的辅助类库。

android.os 提供了系统服务、消息传输、IPC 管道。

android.graphics GPhone 图形库，包含了文本显示、输入输出、文字样式。

android.database 包含底层的 API 操作数据库 (SQLite)。

android.content 提供各种数据传输、服务、资源管理。

android.view 提供基础的用户界面接口框架。

android.widget 显示各种控件，如按钮、列表框、进度条等。

android.app 提供高层的程序模型及基本的运行环境。

android.provider 各种定义变量标准。

android.telephony 提供与拨打电话相关的 API 交互。

android.webkit 默认浏览器操作接口。

3.2.2 源代码的结构

1. Android 的工程目录

Android 的核心工程包含了对 Android 系统基本运行的支持，以及 Android 系统的编译系统，工程的内容包括 bionic、bootloader、build、dalvik、development、framework/base、framework/policies/base、hardware/libhardware、hardware/ril、kernel、prebuilt、system/core 和 system/extras 等。

Android 的扩展工程包含在 external 文件夹中，是一些经过修改后适应 Android 系统的开源工程。有一些工程在主机上运行，也有些在目标机上运行。工程名称及描述如表 3-1 所示。

表 3-1 工程名称及其含义

工程名称	描 述
aes	高级加密标准加解密库
apache-http	Http 服务器
bison	自动生成语法分析器程序，基本兼容 Yacc
bluez	蓝牙库
bsdif	用于为二进制文件生成补丁
bzip2	压缩文件工具
clearsilver	模板语言，包括 Python、Java、Perl、C 的 lib 支持
dbus	freedesktop 下开源的 Linux IPC 通信机制
dhcpcd	动态主机设定协定的工具
dropbear	ssh2 服务器和客户端
e2fsprogs	Ext2/3/4 文件系统的工具
elfcopy	ELF 工具
elfutils	ELF 工具
embunit	嵌入式 C 系统测试架构
emma	Java 代码覆盖检查工具
esd	仅头文件
expat	XML Parser
fdlibm	精确实现 IEE754 浮点数
freetype	C 语言实现字体光栅化引擎制作的软件库
Gdata	用于数据操作
genext2fs	Ext2 文件系统生成工具
giflib	GIF 工具库
googleclient	Google 客户端
grub	多重操作系统启动管理器
icu4c	IBM 支持软件国际化的开源项目

(续)

工 程 名 称	描 述
iptables	防火墙
jdifff	比较工具
jhead	Exif 编辑修改软件
jpeg	jpeg 工具库
libffi	外部函数接口
libpcap	网络数据包捕获函数包
libpng	PNG 工具库
libxml2	C 语言的 XML 解析库
netcat	读写用于网络连接的 TCP 或 UDP
netperf	网络性能测量工具
neven	人脸识别库
opencore	多媒体框架
openssl	C 语言的 SSL 工具
oprofile	Linux 内核支持的一种性能分析机制
ping	ping 工具
ppp	ppp 工具
protobuf	Google 工具, 利用 .proto 文件自动生成代码
qemu	仿真环境
safe-iop	跨平台的整数运算
skia	3D 图形库
sonivox	Sonic 嵌入式音乐合成器
sqlite	轻量级 SQL 嵌入式数据库
srec	motorola S-records 十六进制文件格式工具
strace	监视系统调用的工具
tagsoup	HTML 解析工具
tcpdump	分析被截获的传送数据包的头的工具
tinyxml	XML 工具
tremor	Ogg Vorbis 的播放器
webkit	开源浏览器引擎
wpa_supplicant	无线局域网 WiFi 工具
xdelta3	二进制文件比较工具
yaffs2	YAFFS 文件系统

## 2. Android 中的 Java 程序包

Android 中的 Java 程序包主要包括应用程序和内容提供器两个部分，应用程序在 `package/apps` 目录中，主要包括以下内容：

AlarmClock	闹铃
Browser	浏览器
Calculator	计算器
Calendar	日历
Camera	相机
Contacts	联系人
E-mail	电子邮件
GoogleSearch	Google 搜索服务
HTML Viewer	网页查看器
IM	即时通信
Launcher	程序加载器
Mms	短信息
Music	音乐
PackageInstaller	包管理器
Phone	电话
Settings	设置
SoundRecorder	录音机
Stk	SIM 卡工具
Sync	同步服务
Updater	自动更新
VoiceDialer	语音拨号

内容提供器在 `package/providers` 目录中，主要包括以下内容：

CalendarProvider  
 ContactsProvider  
 DownloadProvider  
 DrmProvider  
 GoogleContactsProvider  
 GoogleSubscribedFeedsProvider  
 ImProvider  
 MediaProvider  
 SettingsProvider  
 SubscribedFeedsProvider  
 TelephonyProvider

## 3.3 搭建开发环境

### 3.3.1 搭建编译环境

#### 1. 安装 VirtualBox

编译 Android 操作系统，一个真正的 Linux 环境是必不可少的。虽然使用 gcc 可以在 Windows 下交叉编译，但是 Android 自带的很多开发工具，如模拟器、ROM 生成脚本等，在 Windows 下进行移植十分困难。因此，若使用 Windows 作为工作系统，就需要通过虚拟机安装 Linux。Google 官方推荐 Ubuntu 系统作为 Android 的开发机。虽然其他的 Linux 发行版也可以作为开发机，但是需要手动配置大量的库。不同的 Android 版本有不同的 Ubuntu 版本，本章将以 Ubuntu 12.04 amd64 作为开发系统。如果已经安装了 Ubuntu 12.04 amd64，请跳到第 4 小点“安装所需的软件包”。

VirtualBox 是一个强大的 x86/amd64 虚拟化产品，如图 3-2 所示。选择 VirtualBox 作为 Windows 下虚拟 Linux 的虚拟化平台是因为 VirtualBox 是唯一免费、开源的虚拟机，同时 VirtualBox 还提供高性能的全硬件虚拟化功能。VirtualBox 提供对 Linux 客户机的良好的支持，支持文件共享、网络共享、USB 共享、鼠标与键盘间无缝切换等功能。本章将使用 VirtualBox 在 Windows 7 中虚拟 Ubuntu 12.04 环境。



图 3-2 VirtualBox 界面

安装 VirtualBox 和 Ubuntu 12.04 至少需要 20GB 硬盘空间（实际上，完整安装 Android 开发包，需要至少 100GB 空间），至少 4GB 内存（其中 2GB 分配给虚拟机），以及支持硬件虚拟化的 CPU。推荐的配置是 200GB 虚拟磁盘，4GB 以上的虚拟内存和酷睿 i5 以上的 CPU，以及 64 位操作系统。需要注意的是，Android 2.3 以上必须使用 64 位编译环境。

首先，从 [www.virtualbox.org](http://www.virtualbox.org) 下载 VirtualBox，依次单击 Download → VirtualBox x.x.x



for Windows hosts x86/amd64, 然后安装 VirtualBox, 如图 3-3 所示。

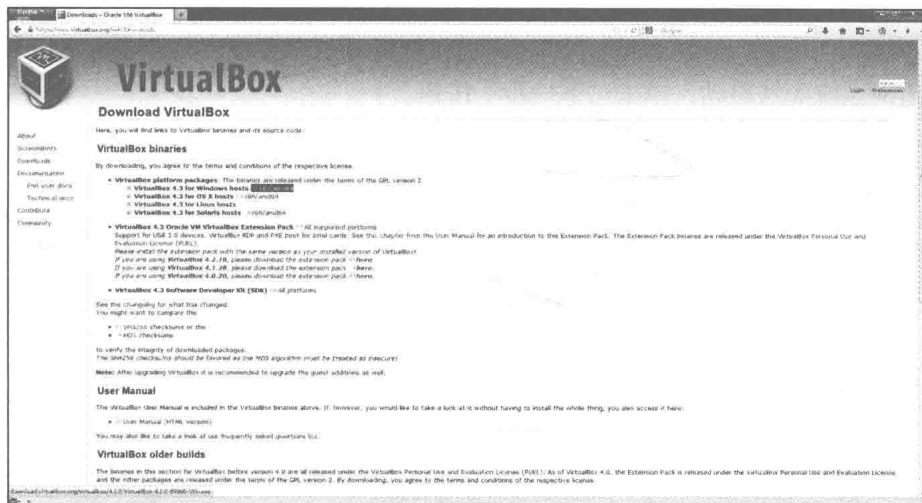


图 3-3 下载 VirtualBox

要使虚拟机直接访问 Android 设备, 例如程序下载, 则还需要安装 VirtualBox 扩展包。VirtualBox 扩展包可以在下载 VirtualBox 安装程序的页面下载, 单击 VirtualBox x.x.x Oracle VM VirtualBox Extension Pack All supported platforms 链接即可。下载完成后可将此文件拖入 VirtualBox 的界面, 完成安装。注意, 只有在阅读完所有许可文本后才能单击“我同意”, 如图 3-4 所示。

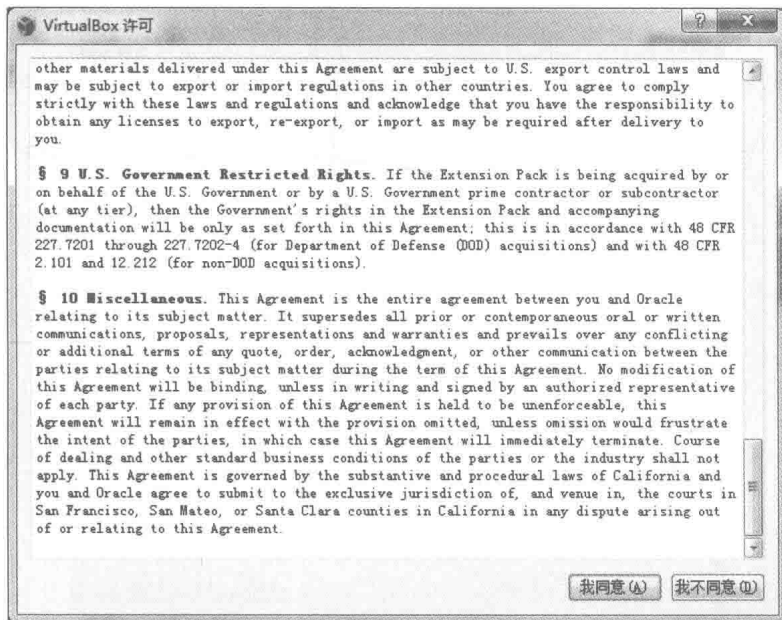


图 3-4 VirtualBox 许可文本

## 2. 安装 Ubuntu 12.04 LTS

在 VirtualBox 主界面里, 单击新建按钮, 如图 3-5 所示。在名称栏里输入 Ubuntu, VirtualBox 会自动选择 Ubuntu Linux 操作系统类型, 再手动将系统类型设置为 64 位, 否则后期可能会造成系统无法启动, 单击“下一步”按钮。

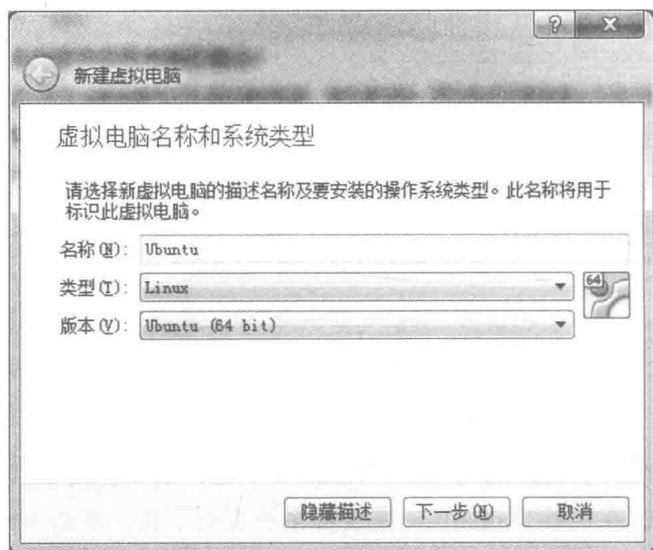


图 3-5 新建虚拟电脑

设置内存大小。推荐的内存大小为至少 4GB, 如图 3-6 所示。不推荐内存大小超过宿主机内存大小的一半, 因为这样设置可能会导致宿主机不稳定, 尤其是在虚拟内存不足的情况下。

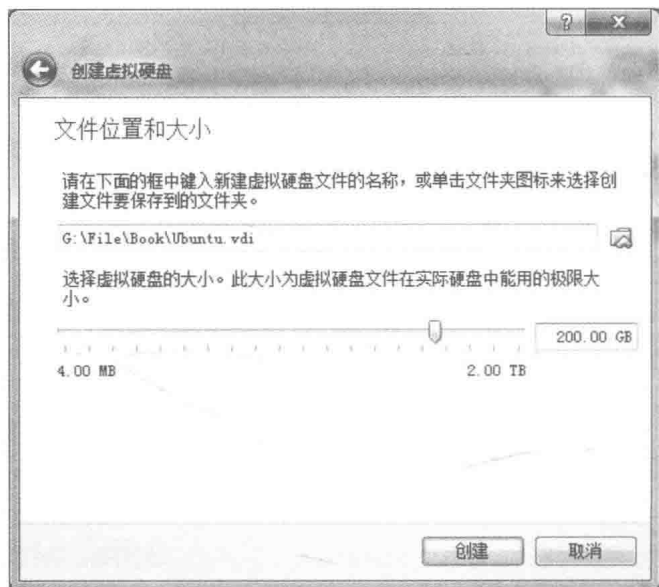


图 3-6 VirtualBox 中创建虚拟硬盘

在接下来的页面中选择“现在创建虚拟硬盘”，单击“创建”按钮。在下一页中选择虚拟磁盘的存放路径和大小，存放路径应该至少有 120GB 的可用空间，磁盘大小可设置为 120GB，如果需要同步更多的 Android 镜像，需要更大的硬盘空间。单击“创建”按钮进入下一页面。

至此虚拟机的创建已经完成。

安装系统时，首先配置虚拟机。单击窗口左侧的虚拟机图标，单击“设置”按钮，在“系统”选项卡中设置“芯片组”为 ICH9，如图 3-7 所示。



图 3-7 VirtualBox 中主板相关配置

在“显示”选项卡中设置“显存大小”为 128MB，勾选“启用 3D 加速”，如图 3-8 所示。如果显存设置过小，可能会使 Android 虚拟机工作不正常，或者 Ubuntu 系统工作不正常。



图 3-8 VirtualBox 中显示配置

在“存储”选项卡中将“IDE 控制器型号”设置为 ICH6，再单击 IDE “控制器”下方的“没有盘片”，在“分配光驱”处设置 Ubuntu 系统的 ISO 镜像文件，如图 3-9 所示。Ubuntu 操作系统可在 [www.ubuntu.com](http://www.ubuntu.com) 下载。在 [www.ubuntu.com](http://www.ubuntu.com) 首页单击 Download 按钮，在 Alternative download options 一栏里单击 Take a look at a full list of our previous versions and alternative downloads，选择合适的下载点以下网址：下载 Ubuntu 12.04 amd64 的镜像即可。实际上，国内的很多高校都提供 Linux 的镜像下载，可以尝试 <http://mirror.lzu.edu.cn/ubuntu-releases/12.04.3/ubuntu-12.04.3-desktop-amd64.iso>。



图 3-9 VirtualBox 中存储配置

在“网络”选项卡中配置网络设置。连接方式可选“网络地址转换”或者“桥接”，如图 3-10 所示。如果需要通过 TFTP 等网络协议将 Android ROM 烧录到设备中，则选择桥接模式。如果通过 SD 卡或者 USB DFU 烧录，则选择网络地址转换。网络地址转换能提供额外的安全性。



图 3-10 VirtualBox 网络配置

单击“启动”按钮，启动虚拟机。在初次启动以及驱动程序安装的过程中，会多次看到如图 3-11 所示的窗口，单击“不要再显示这个信息”，然后单击“捕获”或“确定”按钮即可。

在进入系统后，单击右上角的网络图标，断开虚拟机的网络连接。单击“继续”按钮，在安装类型页面中单击“其他”选项，单击“继续”按钮。选择 /dev/sda，单击“新建分区表”，如图 3-12 所示。

单击“继续”按钮，然后单击 /dev/sda 下方的“空闲”，单击“添加”按钮，创建一个至少 4GB 的交换空间（如果你的虚拟机内存不够，则要分配更大的交换空间），然后创建一个至少 100GB 的“ext4 日志文件系统”分区，将该分区“挂载点”设置为“/”，如图 3-13 所示。

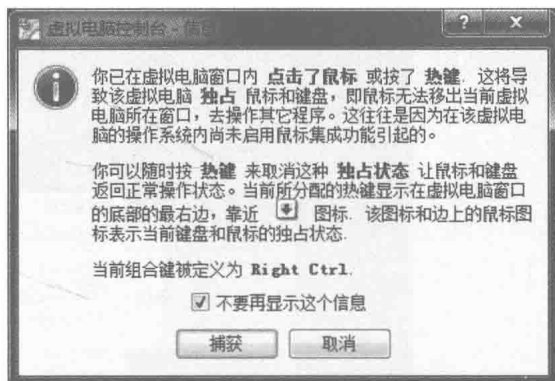


图 3-11 启动虚拟机



图 3-12 虚拟机中进行 Ubuntu 安装

单击“现在安装”按钮，选择时区（一般选择上海），单击“下一步”按钮，选择“键盘布局”，单击“下一步”按钮，然后设置用户名、密码等认证信息。系统安装完毕以后，重新启动虚拟机。



图 3-13 分配挂载点

### 3. 配置 Ubuntu 12.04 LTS

系统安装完毕以后，开始配置系统。首先，Ubuntu 采用软件源的概念。用户可以自己从互联网下载第三方软件，也可以从官方的软件源中下载软件。由于 Ubuntu 高度依赖软件源，必须先配置软件源。如果不使用教育网，可以跳过此环节。如果使用教育网，则使用中国高校提供的 Ubuntu 软件源，可大幅度缩短下载时间。使用教育网的软件源，在桌面按 Alt+F2 组合键，输入 `gnome-terminal`，打开终端。在终端输入以下命令：

```
sudo gedit /etc/apt/sources.list
```

输入密码，然后在 gedit 界面里面将 `cn.archive.ubuntu.com` 替换为 `mirror.lzu.edu.cn`，如图 3-14 所示。如果习惯使用 vi 或者 vim，也可以使用以下命令：

```
sudo vi /etc/apt/sources.list
```

然后输入

```
%s/cn.archive.ubuntu.com/mirror.lzu.edu.cn/g
```

完成替换，最后输入 `wq` 命令保存并退出。

除了 `mirror.lzu.edu.cn` 这个软件源，还可以使用北京交通大学、清华大学等各个高校提供的软件源。

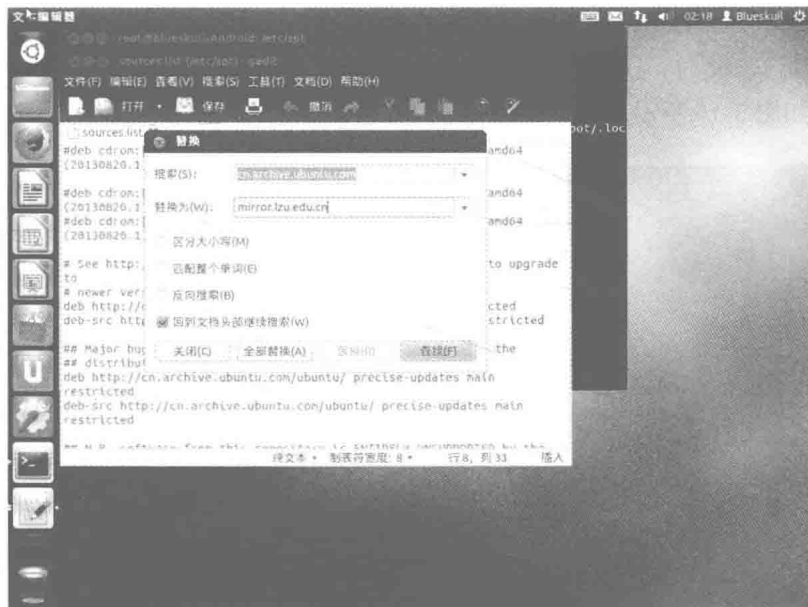


图 3-14 配置更新源

在完成了软件源更换后，执行 `sudo apt-get update` 命令从软件源同步新的软件包信息。

同步完成后，系统会提醒有系统更新，推荐全部安装。在此之后，可以安装完整的中文支持。单击右上角的齿轮图标，单击“系统设置”→“语言支持”，系统会提醒语言安装不完整，选择“安装”即可，如图 3-15 所示。中文支持不是必需的。



图 3-15 虚拟机中 Ubuntu 安装语言包

重新启动系统，安装 VirtualBox 客户机驱动程序。在运行的虚拟机窗口中单击“设备”，安装“增强功能”。此时，Ubuntu 系统会提醒有一个自动运行的光盘程序，单击 Run 按钮，并输入密码，如图 3-16 所示。等待安装完毕，重启虚拟机即可。

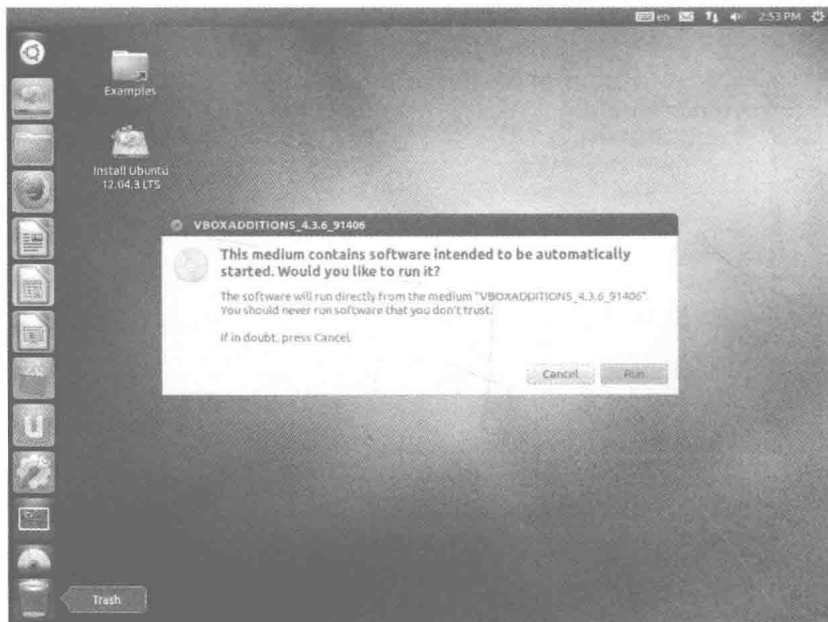


图 3-16 VirtualBox 开启增强功能

#### 4. 安装所需的软件包

为编译 Android，需要安装如下工具：

Oracle JDK 6

gitnupg

flex

bison

gperf

build-essential

zip

curl

libc6-dev

libncurses5-dev 32 位版本

x11proto-core-dev

libx11-dev 32 位版本

libreadline6-dev 32 位版本



libgl1-mesa-glx 32 位版本

libgl1-mesa-dev

g++-multilib

mingw32

tofrodo

python-markdown

libxml2-utils

xsltproc

zlib1g-dev 32 位版本

安装 JDK6。需要注意的是，Android 对 JDK 版本十分敏感，只有安装了正确的 JDK 版本才能正确编译 Android 源代码。本书以 Android 4.2 为例，Android 4.2 要求的 JDK 版本为 Java 6，而 Oracle 的官方最新版为 Java 7。安装 Java 7 会导致 Android 4.2 编译过程中出现错误。可以在以下网址下载到 JDK6：

<http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase6-419409.html>

单击 Java SE Development Kit 6u45，选择 Accept Licensing Agreement，选择 jdk-6u45-linux-x64.bin，即可下载 JDK 6。对于 Java 7 以前的版本，Oracle 可能会要求登录。如果没有进行过注册，需要先注册一个 Oracle 账号，登录后即可开始下载。

完成下载后，进入下载文件夹，运行以下程序：

```
chmod+x jdk-6u45-linux-x64.bin
./jdk-6u45-linux-x64.bin
```

得到一个名为 jdk1.6.0\_45 的文件夹。然后执行以下命令：

```
sudo mv jdk1.6.0_45 /usr/local/java
```

将 JDK 移动到 /usr/local/java 文件夹中。执行以下 6 条命令：

```
sudo update-alternatives --install "/usr/bin/java" "java" "/usr/lib/jvm/java-6-oracle/bin/java" 1
sudo update-alternatives --install "/usr/bin/javac" "javac" "/usr/lib/jvm/java-6-oracle/bin/javac" 1
sudo update-alternatives --install "/usr/bin/javaws" "java" "/usr/lib/jvm/java-6-oracle/bin/javaws" 1
sudo update-alternatives --config java
sudo update-alternatives --config javac
sudo update-alternatives --config javaws
```

安装完成以后，输入 java -version 来测试安装结果。正常输出应该如下：

```
java version "1.6.0_30"
```

```
Java(TM) SE Runtime Environment (build 1.6.0_30-b12)
Java HotSpot(TM) Client VM (build 20.5-b03, mixed mode, sharing)
```

为了让系统能更方便地访问 Java，可以添加一些环境变量。将以下语句添加到 `/etc/profile` 文件的结尾：

```
JAVA_HOME=/usr/lib/jvm/java-6-oracle
PATH=$PATH:$HOME/bin:$JAVA_HOME/bin
export JAVA_HOME
export JAVA_BIN
export PATH
```

执行 `/etc/profile`，重新加载环境变量。此时，JDK 就安装、配置完成。接下来，需要配置其他的软件包。执行以下命令：

```
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
```

安装过程中如果提示出错，执行以下命令：

```
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-dev g++-multilib mingw32 tofrodos python-markdown
libxml2-utils xsltproc zlib1g-dev:i386
```

最后，执行以下命令：

```
ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-gnu/libGL.so
（这里需要root权限）
```

### 3.3.2 使用 repo

#### 1. repo 简介

Android 使用 Git 作为代码管理工具，开发 Gerrit 进行代码审核以便更好地对代码进行集中式管理，还开发了 Repo 命令行工具，对 Git 部分命令封装，将多个 Git 库进行有效的组织。

所有的 Android 代码都可以通过 repo 进行下载。

#### 2. 初始化客户端

执行以下语句下载 repo 工具，并将 repo 存放于家目录的 bin 文件夹中。

```
mkdir ~/bin
PATH=~/bin:$PATH
curl http://android.git.kernel.org/repo > ~/bin/repo
chmod +x ~/bin/repo
```

设置 repo 的同步范围。执行以下命令：

```
repo init -u https://android.googlesource.com/platform/manifest
```

如果需要同步 Android 的一个分支，而不是 Master，可以使用 -b 选项。

```
repo init -u https://android.googlesource.com/platform/manifest -b 分支名称
```

具体的分支名称可以查询 <http://source.android.com/source/build-numbers.html> 中的 Source Code Tags and Builds。

### 3. 获取源代码

执行 repo sync 可以同步选定的 Android 源代码。该过程会同步十几 GB 的文件，所以会消耗很长时间。如果使用代理上网，可以在执行 repo sync 以前执行以下语句：

```
export HTTP_PROXY=http://用户名:密码@代理服务器:端口
export HTTPS_PROXY=http://用户名:密码@代理服务器:端口
```

如果没有用户名和密码，则输入：

```
export HTTP_PROXY=http://代理服务器:端口
export HTTPS_PROXY=http://代理服务器:端口
```

同步完成后，需要添加验证密钥。

执行 gpg --import，输入以下代码：

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
Version: GnuPG v1.4.2.2 (GNU/Linux)
```

```
mQGibEnnWD4RBAct9/h4v9xnnGDou13y3dvOx6/t43LPPIxeJ8eX9WB+8LLuROSv1FhpHawsVAcFlmi7
f7jdsRF+OvtZL9ShPKdLfWBJMNkU66/TZmPewS4m782ndtw78tR1cXb197Ob8kOfQB3A9yk2XZ4
ei4ZC3i6wVdqHLRxABdncwu5hOF9KXwCgkxMDu4PVgChaAJzTYJ1EG+UYBIUEAJmfearb0qRAN7
dEoff0FeXsEaUA6U90sEoVks0ZwNj96SA8BL+alOoEUUfpmhiHyLuQsftxisJxTh+2QclzDviDy
aTrkAnjdY7p2cq/HMdOY7LJlHaqtXmZxXjttw5Uc2QG8UY8aziU3IE9nTjSwCXeJnuyvoizl9/
Ils5jU5SA/9WwIps4SC84ielIXiGWEqq6i6/sk4I9qlYemZF2XVVKnmIlF4iCMtNKsR4MGSa1gA
8s4iQbsKNWPgp7M3a51JCVCu6l/8zTpA+uUGapw4tWCp4o0dpIvDPBEa9b/aF/ygcR8mh5hgUfp
F9IpXdknOsbKcV9M9lSSfRciETykZc4wrRCVGhlIEFuZHJvaWQgT3BlbiBTb3VyY2UgUHJvamVj
dCA8aW5pdGllbC1jb250cmllidXRpb25AYW5kcmluZC5jb20+igAEEeXECACAFaknnWD4CGwMGcWk
IBwMCBBUCCAMEFgIDAQIEAQIXgAAKCRDorT+BmrEOeNr+AJ42Yy6tEW7r3KzrJxnRX8mi9j9z8tg
CdFfQYiHpyNgkI2t09Ed+9Bm4gmEO5Ag0ESedYRBAIAKVW1JcMBWvV/0Bo9WiByJ9WJ5swMN36/
vAlQN4mWRhfzDOK/Rosdb0csAO/18Kz0gKQP0fObtyYjvI8JMC3rmi+LlvSUT9806UphisyEmmH
v6U8gUb/xHLIAnXGxwhYzjgeuAXVCsv+EvoPIHbY4L/KvP5x+oCJIDbkC2b1TvVk9PryzmE4BPI
QL/NtgRlOwLm/uWR9zRUfTbN4E11aMAN3qnAHBBMZzKMxLWBGE0znfRrnczI5p49i2YzJAjyX1
P2WzmScK49CV82dzLo7lMnrF6fj+Udtb5+OgTg7Cow+8PRaTkJEW5Y2JIZpnRUq0CYxAmHYX79E
MKHDSThf/8AAwUIAJPwSB/MpK+KMS/s3r6nJrnYLTfdZhtmQXimp0DMJg1zxmL8UfNUKiQZ6es
oAWtDgpgt7Y7sKZ81aHRARonte394hidZzM5nb6hQvpPjt20lPRsyqVxw4c/KsjAdtAuKW9/d8
phbN8bTyOJo856gg4o0EzKG9eeF7oaZTYBy33BTL0408sEBxiMior6b8LrZrAhkqDjAvUXRwm/
fFKgpsOysxC6xi553CxBUCH2omNV6KalLNmwzSp9ILz8jEGqmUtkBszoG1S8fXgE0Lq3cdDM/
GJ4QXP/p6LiwNF99faDMTV3+2SAOGvytOX6KjKVzKOSsfJQhN0DlsIw8hqJc0WISQQYEQIACQUC
```

```
SedYRAIbDAAKCRDorT+BmrEOeCUOAJ9qmR0lEXzeoxcdoafxqf6gZlJZlACgkWF7wi2YlW3Oa+
jv2QSTlrx4KLM==Wi5D
-----END PGP PUBLIC KEY BLOCK-----
```

按 Ctrl+D 组合键，完成密钥输入。输入完成后，可执行“git tag -v 分支名称”命令，来验证下载的完整性和可靠性。

#### 4. 创建本地代码库

如果很多客户都需要编译 Android，或者带宽紧张，则可以对远程 Android 服务器进行镜像，这样每次访问就不再需要从外网下载了。完整的镜像服务器需要下载的内容小于两个独立的客户端同步的下载量，但是却包含了更多的信息。

创建本地 Android 缓存，创建缓存文件夹，然后初始化 repo，最后同步镜像。

```
mkdir ~/and_mirror
cd ~/mirror
repo init -u https://android.googlesource.com/mirror/manifest -mirror
repo sync
```

Android 代码仓库将被放置于家目录下面的 and\_mirror 文件夹。一旦本地镜像同步完成，就可以在本地访问 Android 的代码仓库。执行以下命令：

```
mkdir ~/android
cd ~/android
repo init -u /usr/local/aosp/mirror/platform/manifest.git
repo sync
```

可以直接同步本地的 Android 代码库。如果需要更新，则先将镜像与 Google 服务器同步，再将客户端与镜像同步。执行以下命令：

```
cd ~/and_mirror
repo sync
cd ~/android
repo sync
```

如果希望通过网络访问 Android 代码库，可以通过 NFS 共享 and\_mirror 文件夹。需要注意的是，如果 NFS 服务器被安装在虚拟机里，虚拟机网络一定要选择桥接模式，NAT 模式下虚拟机不能被用作服务器。

### 3.3.3 Android 的编译

#### 1. 设置环境变量

编译 Android，需要设置代码缓存、输出文件夹和编译环境。代码缓存是指在编译过程中使用额外的硬盘空间，缓存编译所产生的目标文件。当使用 make clean 命令清理编译垃圾后，代码缓存可以大大提高二次编译的速度。设置代码缓存，将以下代码插入 ~/.bashrc 文

件的结尾。

```
export USE_CCACHE=1
```

默认的代码缓存存在家目录的 .ccache 文件夹中，要改变此设置，可在上句以后添加以下语句：

```
export CCACHE_DIR=代码缓存目录
```

默认的编译输出文件夹是 Android 代码目录下的 out 文件夹。out 文件夹又分为几个子文件夹，host 子文件夹存放编译完成的 SDK，target 文件夹存放编译完成的设备代码，即 rom 文件。要修改默认的输出文件夹位置，可在 ~ /.bashrc 文件结尾插入以下语句：

```
export OUT_DIR_COMMON_BASE=输出文件夹
```

重新开启一个终端，或执行 ~ /.bashrc 命令，完成环境变量设置。

以上修改和命令只需要执行一次，以后便不需要再运行。每次编译之前，需要在 Android 代码库根目录下执行 source build/envsetup.sh，来设置编译环境。

## 2. 编译 Android 源码

执行完 source build/envsetup.sh 后，执行 lunch 命令选择编译目标。目标名一般为 xxx\_yyy-zzz 的形式，xxx 为目标设备，一般为 full、mini 或 aosp；yyy 为目标平台，即处理器内核或设备型号；zzz 为编译选项，可选项为 user、userdebug 和 eng。user 选项下编译得到的 rom 文件不具备 root 权限，userdebug 选项下得到的文件具备 root 权限，eng 选项在 userdebug 的基础上提供更多的调试工具。

执行 make -jX 开始编译，X 为同时使用的进程数。对于双线程处理器，X 一般取 2 ~ 4，对于 4 线程处理器，X 取 4 ~ 6，对于 8 线程处理器，X 一般取 8 ~ 12。编译器占用的理论最大 CPU 资源为 X 与 CPU 线程数的比值。要编译用于 x86 的 VirtualBox 虚拟机镜像，可以执行 make android\_disk\_vdi -jX 命令。

编译过程中出现报错，需按照错误提示进行操作。常见错误有 JDK 版本不匹配、编译器版本太旧或系统库文件缺失等。如果出现单纯的代码语法错误，则很可能是下载的文件有损坏，使用“git tag -v 分支名称”命令验证。

编译得到的 rom 文件存放于 Android 编译输出文件夹 /target/product/ 设备型号文件夹中。编译结束后，如果编译目标是 aosp\_arm-zzz，可运行 emulator 命令启动模拟器运行。emulator 命令相关的环境变量会在编译完成时自动设置。

启动 emulator 模拟器，如图 3-17 所示。

## 3. 编译 SDK

在执行 source build/envsetup.sh 和 lunch 之后，可以使用以下命令：

```
make -jX sdk
```

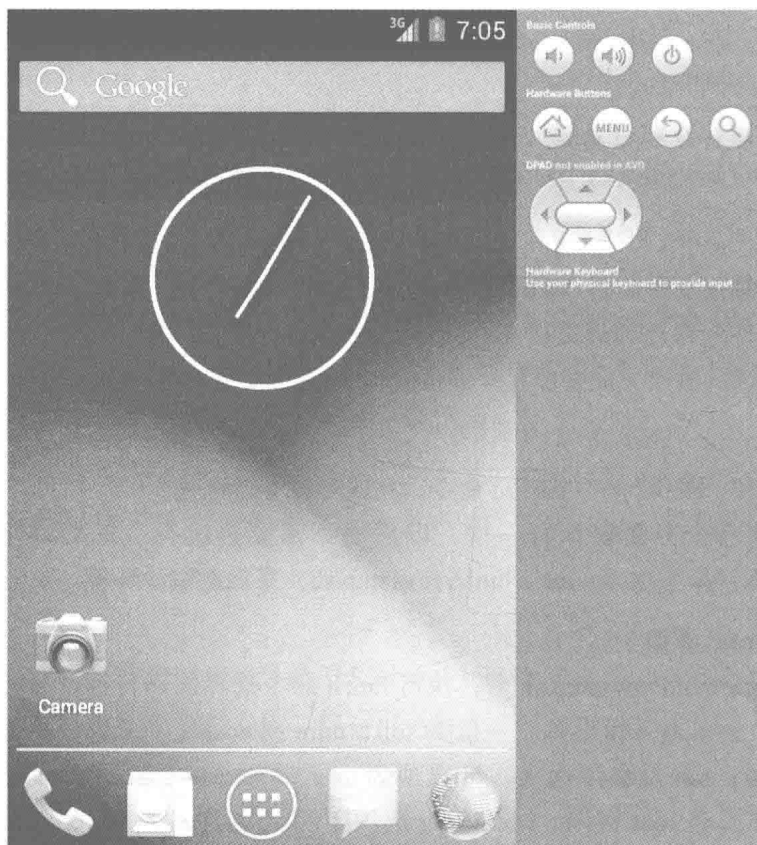


图 3-17 启动 emulator 模拟器

该命令编译对应 Linux 和 Mac OS X 版本的 SDK，供 Android 应用开发使用。得到的文件存放于 Android 编译输出文件夹 /host/ 设备型号 /sdk/android-xxx 文件夹中。

或使用以下命令：

```
make -jX win_sdk
```

命令编译用于 Windows 对应版本的 SDK，得到的文件存放于 Android 编译输出文件夹 /host/ 设备型号 /sdk/android-xxx 文件夹中。

## 第 4 章

# Android 系统底层源码结构分析

本章将向读者介绍 Android 系统源码结构，深入分析结构中的相关文件及工具。

### 4.1 源码结构分析

Android 是一个开放的软件系统，它借助 Linux 提供的硬件抽象接口运行在各种平台之上，实现存储管理、进程管理、网络接入以及其他操作系统服务。因此对 Android 系统的描述通常为：一个建立在 Linux 内核基础上，包含诸多源代码的开放式移动终端操作系统。Android 系统的层次划分如图 4-1 所示。

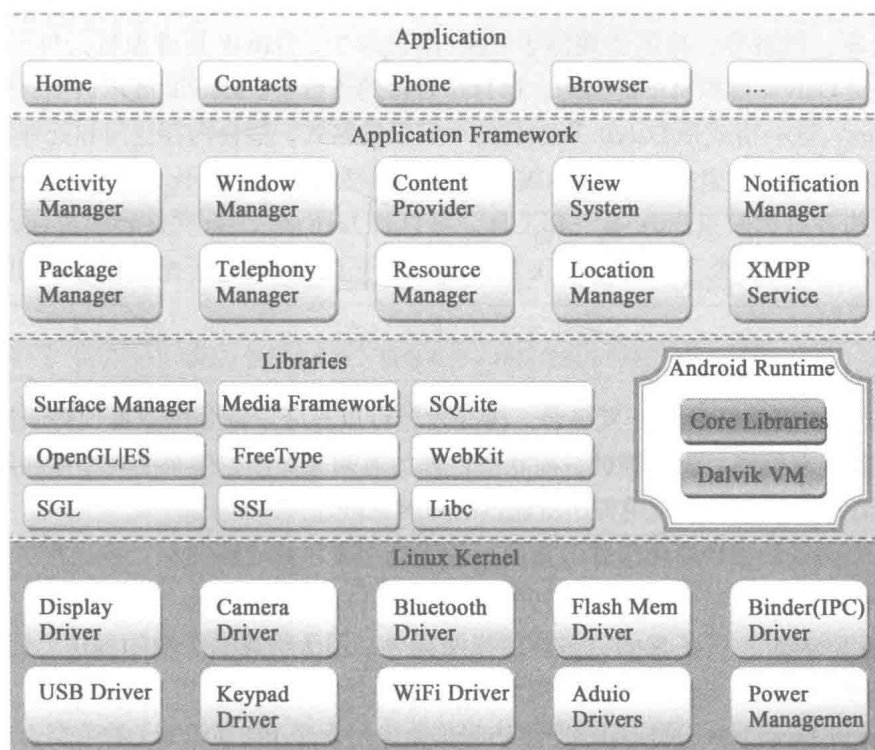


图 4-1 Android 系统层次

从图 4-1 中不难看出, Android 系统可以分为 4 个层次, 从下至上分别为:

- Linux Kernel、Linux 操作系统及驱动。
- 本地代码 (C/C++) 框架, 也称为 Libraries 层。
- Java 应用程序框架层 Application Framework。
- Java 应用程序层 Application。

Android 的第一层次由 C 语言实现, 第二层次由 C 和 C++ 实现, 第三、四层次主要由 Java 代码实现。

### 1. Linux Kernel

核心系统服务层, 完成诸如安全、内存管理、进程管理、网络堆栈、驱动模型等操作。Linux Kernel 也是硬件和软件之间的抽象层, 隐藏具体的硬件细节为上层提供统一的服务。分层的好处在于屏蔽本层及以下层的差异, 为上层提供统一的服务和接口 SAP (Service Access Point), 保证本层及以下层发生变化不会影响到上层。

### 2. Libraries

Android 包含一个 C/C++ 库的集合, 属于 Linux 层次的 C 库, 供 Android 系统各个功能组件调用。这些功能通过 Android 的应用程序框架暴露给开发者。Android Runtime 是 Android 的运行环境, 由两部分构成: Core Libraries 和 Dalvik VM。Core Libraries 由 Java 的基本类组成, 主要用于提供基本的 Java 类库的功能, 包括诸如基础数据结构、数学、I/O、工具、数据库、网络等, 具体实现位于 Libcore 目录中。Dalvik 是虚拟机, 每一个 Android 应用程序都是 Dalvik 虚拟机中的实例, 运行在自己的进程中。Dalvik 虚拟机生成的可执行文件格式为 .dex, 这一格式是 Dalvik 所独有的一种压缩格式, 适合内存空间和处理器速度非常有限的系统。大多数的虚拟机, 包括 JVM 都是基于栈的, 而 Dalvik 虚拟机则是基于寄存器的。两种架构各有优缺点。dx 是一套工具, 可以将 Java 的 .class 文件转换成 .dex 格式。一个 dex 文件通常会有多个 .class。Dalvik 虚拟机依赖于 Linux 内核所提供的基本功能, 如线程和底层内存管理。

### 3. Application Framework

Android 通过提供开放的开发平台, 使开发者可以充分发挥硬件设备优势, 执行访问位置信息、运行后台服务、设置闹钟、向状态栏添加通知等操作, 编制极其丰富和新颖的应用程序。开发者可以使用核心应用程序所提供的框架 APIs。

Java 框架的目的在于简化组件的重用, 任何应用程序都能使用和发布这些功能 (需要服从框架执行的安全限制)。主要功能列表如下:

- View System: 可扩展的、多样的视图集合, 用于构建一个应用程序、列表、网格、文本框、按钮, 甚至是内嵌的网页浏览器。
- Content Provider: 使应用程序能访问其他应用程序 (如通讯录) 的数据, 或共享自己的数据。



- Resource Manager: 提供访问非代码资源, 如本地化字符串、图形和布局文件。
- Notification Manager: 使所有的应用程序能够在状态栏显示自定义警告。
- Activity Manager: 管理应用程序生命周期, 提供通用的导航回退功能。

4. Applications

Java 核心应用程序集合, 包括电子邮件客户端、SMS 程序、日历、地图、浏览器、联系人等其他设置和应用。

通过以上分析可以看出 Android 系统属于分层架构, 层次清晰, 各层分工明确。

从 Linux 操作系统的角度来看, 第一层次和第二层次之间是内核空间与用户空间的分界线, 第一层次运行于内核空间, 其后的第二、三、四层次运行于用户空间。

第二层次和第三层次之间是本地代码层和 Java 代码层的接口。

第三层次和第四层次之间是 Android 的系统 API 的接口, 对于 Android 应用程序的开发, 第三层次以下的内容是不可见的, 仅考虑系统 API 即可。

在程序开发过程中, 通常会分为前端开发和底层开发两类。前端开发一般指针对客户端设备或应用进行的开发, 底层开发指的是针对硬件的开发, 例如接口程序、驱动程序等的开发, 使用的语言以 C、C++、汇编语言为主。对应于 Android 源码结构, 底层开发通常针对的就是第一层 Linux Kernel 层和第二层 Libraries 层。

4.1.1 底层库结构介绍

1. 底层函数库的基本结构

Android 系统源代码目录结构如表 4-1 所示。

表 4-1 Android 系统源代码目录结构

项 目	描 述
abi	applicationbinary interface 相关代码, 应用程序二进制接口
bionic	C runtime:libc、libm、libdl、dynamic linker bionic 含义为仿生, 这里面是一些基础的库的源代码
bootloader/legacy	Bootloader reference code 启动引导相关代码
build	Building system build 目录中的内容不是目标所用的代码, 而是编译和配置所需要的脚本和工具
cts	Android 兼容性测试套件标准
dalvik	Dalvik virtual machine Java 虚拟机
development	High-level development and debugging tools 程序开发所需要的模板和工具
device	各个厂商相关设备的代码

(续)

项 目	描 述
docs	编译 Android 文档相关
external	Android 的扩展工程
framework/base	Core Android app framework libraries 目标机器使用的一些库
frameworks/policies/base	Framework configuration policies
hardware	HAL 代码, 硬件抽象层适配层
libcore	核心库相关
ndk	Android Native Development Kit 是一系列的开发工具, 允许程序开发人员在 Android 应用程序中嵌入 C/C++ 语言编写的非托管代码
out	编译完成后的代码输出至此目录
kernel	Linux kernel Linux 2.6 的源代码
prebuilt	Binaries to support Linux and Mac OS builds x86 和 ARM 架构下预编译的一些资源
packages	Android 的各种应用程序
sdk	SDK 及模拟器
recovery	System recovery environment 与目标的恢复功能相关
system	Android 底层的一些库
vendor	厂商定制代码
makefile	工程编译管理总入口

Android 的本地实现层次具有基本的库和程序, 这些库和程序是 Android 基本系统运行的基础, 主要包括:

- 标准 C/C++ 库 bionic
- C 语言工具库 libcutils
- C++ 工具库 libutils
- Init 进程
- shell 工具

## 2. 底层函数库的功能

Android 包含一些 C/C++ 库, 这些库能被 Android 系统中不同的组件使用, 通过 Android 应用程序框架为开发者提供服务。以下是一些核心库。

- 系统 C 库: 从 BSD 继承来的标准 C 系统函数库 (libc), 专门为基于嵌入式 Linux 的设备定制的。

- 媒体库：基于 PacketVideo OpenCORE。支持多种常用的音频、视频格式回放和录制，同时支持静态图像文件。编码格式包括 MPEG4、H.264、MP3、AAC、AMR、JPG、PNG。
- Surface Manager：对显示子系统的管理，并且为多个应用程序提供了 2D 和 3D 图层的无缝融合。
- LibWebCore：一个 Web 浏览器引擎，支持 Android 浏览器和一个可嵌入的 Web 视图。
- SGL：底层的 2D 图形引擎。
- 3D libraries：基于 OpenGL ES 1.0 API 实现，可以使用硬件 3D 加速或者使用高度优化的 3D 软加速。
- FreeType：位图和适量字体显示。
- SQLite：一个对于所有应用程序可用、功能强劲的轻型关系型数据库引擎。

4.1.2 C 基础函数库 bionic

bionic 是一个专为嵌入式系统设计的轻量级标准库实现，优化和剪裁了一些使用频率低并且资源消耗比较高的函数，使其具有更小的体积和内存占用，同时也提高了执行效率，是整个系统的基础类库。bionic 支持标准 C/C++ 库的绝大部分功能，支持数学库及 NPTL 线程库。实现了用于动态库的创建和加载操作的 Linker 和 Loader，以及一套使用共享内存方式实现的 property 系统。bionic 不提供 libthread\_db 和 libm 的实现，也不完全支持 POSIX 标注。

bionic 函数库的结构如表 4-2 所示。

表 4-2 bionic 函数库结构

项 目	描 述	项 目	描 述
libc	C 库	libthread_db	多线程程序的调试器库
libdl	动态链接，提供访问动态链接库的功能	linker	动态链接器
libm	数学库的实现	arch	支持 ARM 和 x86 两种架构
libstdc++	C++ 实现库		

bionic 类似于嵌入式系统中常用的 uclib。除了性能、效率等方面的考虑，还由于商业用途的版权限制，即不被 LGPL 限制。

bionic 库中基础 C 库的源代码主要放在 libc 库中，libc 库的结构如表 4-3 所示。

表 4-3 libc 函数库结构

项 目	描 述
arch-arm	ARM 架构，包含系统调用汇编实现
arch-x86	x86 架构，包含系统调用汇编实现

(续)

项 目	描 述
bionic	由 C 实现的功能，与架构无关
docs	文档
include	头文件
inet	与 inet 相关的操作
kernel	Linux 内核中的头文件
netbsd	与 nesbsd 系统相关的操作
private	私有头文件
stdio	stdio 函数实现
stdib	stdib 函数实现
string	string 函数实现
tools	工具
tzcode	时区相关代码
unistd	unistd 函数实现
zoneinfo	时区信息

从表 4-3 可以看出，libc 函数库仅支持 ARM 架构和 x86 架构。

4.1.3 C 语言底层库 libcutils

Android 本地最基础的 C 语言工具库提供了 C 语言最基本的工具功能，基本上 Android 中所有的本地库和程序都连接了这个库。libcutils 的功能一般都是调用 Linux 标准库的封装实现的。在 C 语言工具库中，主要的头文件如下。

- Ashmem.h: 匿名共享内存的接口
- Atomic.h: 原子操作接口
- Array.h: 定义一个可动态改变大小的数组
- Threads.h: 线程接口
- Sockets.h: Android 的套接字接口
- Properties.h: Android 的属性系统接口
- Log.h: log 信息接口，定义 LOGV、LOGD、LOGI、LOGW、LOGE 等宏
- Mq.h: 消息队列接口
- Hashmap.h: 定义哈希表的工具

4.1.4 C++ 工具库 libutils

libutils 是 Android 的底层库，这个库以 C++ 实现，因此它提供的 API 也是 C++ 的。

Android 层次的 C 语言程序和库，大都基于 libutils 开发。

- libutils 的头文件：frameworks/base/include/utils。
- libutils 的源文件：frameworks/base/include/utils。
- Libutils 的库名称：libutils.so。

这个库可以分成两个部分，一部分是底层的工具，另一部分主要为实现 IPC 的 binder 机制。

binder 用于进程间的通信（IPC），它的实现基础是运行于 Kernel 空间的 binder 驱动。其优点是通过 mmap() 共享内存，不执行序列化，不复制内存。父子 Activity 间就是用 binder 传递数据的。

4.1.5 底层文件系统库 system

system 目录包含底层文件系统库、应用及组件，展开的两个级别的目录如表 4-4 所示。

表 4-4 system 目录结构

项 目	描 述	项 目	描 述
Bluetooth	蓝牙相关	extras	额外工具
core	系统核心工具箱接口	wlan	无线相关

core 与 extras 主要提供若干工具，其目录如表 4-5 和表 4-6 所示。

表 4-5 core 目录结构

项 目	描 述	项 目	描 述
adb	adb 调试工具	libpixelfinger	图形工具库
cpio	cpio 工具，创建 img	libsysutils	系统工具库
debuggerd	调试工具	libzipfile	zip 库
fastboot	快速启动相关	logcat	查看 log 工具
include	系统接口头文件	logwrapper	log 封装工具
init	init 程序源代码	mkbootimg	制作启动 boot.img 的工具盒脚本
libacc	轻量级 C 编译器	netcfg	网络配置 netcfg 源码
libctest	libc 测试相关	nexus	google 最新手机的代码
libcutils	libc 工具	rootdir	rootfs 包含一些 etc 下的脚本和配置
liblog	log 库	sh	shell 代码
libmncrypt	加密库	toolbox	toolbox，类似 busybox 的工具集
libnetutils	网络工具库	vold	SD 卡管理器

表 4-6 extras 目录结构

项 目	描 述	项 目	描 述
latencytop	软件开发工具	showslab	showslab 工具
libpagemap	pagemap 库	sound	声音相关
librank	Java Library Rank System 库	su	su 命令源码
procmem	pagemap 库	tests	一些测试工具
procrank	Java Library Rank System 相关	timeinfo	时区相关
showmap	showmap 工具		

可以说，bionic 文件和 system 文件完成了 Android 对 Linux 的封装，并且在二者的基础上构建了 Android 系统的两大核心模块：Dalvik 和 Framework。

4.1.6 增加本地库的方法

Android 中本地库的添加与库所在路径没有关系，只与 Android.mk 文件有关。Android.mk 文件是 Android 操作系统编译时使用的编译规则文件，是 GNU Makefile 的一小部分，可以生成多个可执行程序、静态库或者动态库。Android.mk 具有统一的格式，其中主要包含一些系统公共的宏。

1. 编译可执行程序模板

```
#Test Exe
LOCAL_PATH := $(call my-dir)
#include $(CLEAR_VARS)
LOCAL_SRC_FILES:= main.c
LOCAL_MODULE:= test_exe
#LOCAL_C_INCLUDES :=
#LOCAL_STATIC_LIBRARIES :=
#LOCAL_SHARED_LIBRARIES :=
include $(BUILD_EXECUTABLE)
```

在该模板中，“:=”代表赋值操作，“\$”代表引用变量的值，指定 LOCAL\_PATH 变量，用于查找源文件；CLEAR\_VARS 清除 LOCAL\_PATH 以外所有以 LOCAL\_ 开头的变量。

LOCAL\_SRC\_FILES 存入源文件路径；LOCAL\_MODULE 存入编译后的模块名称；LOCAL\_C\_INCLUDES 存入所需要包含的头文件路径；LOCAL\_STATIC\_LIBRARIES 存入所需链接的静态库 (\*.a) 的名称；LOCAL\_SHARED\_LIBRARIES 存入所需链接的动态库 (\*.so) 的名称；BUILD\_EXECUTABLE 表示以一个可执行程序的方式进行编译。

2. 编译静态库模板

```
#Test Static Lib
```

```

LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES:= /
    mk_static_library.c
LOCAL_MODULE:= libtest_static
#LOCAL_C_INCLUDES :=
#LOCAL_STATIC_LIBRARIES :=
#LOCAL_SHARED_LIBRARIES :=
include $(BUILD_STATIC_LIBRARY)

```

在该模板中，BUILD\_STATIC\_LIBRARY 表示编译一个静态库。

### 3. 编译动态库模板

```

#Test Shared Lib
LOCAL_PATH := $(call my-dir)
    include $(CLEAR_VARS)
LOCAL_SRC_FILES:= /
    mk_shared_library.c
LOCAL_MODULE:= libtest_shared
TARGET_PRELINK_MODULES := false
#LOCAL_C_INCLUDES :=
#LOCAL_STATIC_LIBRARIES :=
#LOCAL_SHARED_LIBRARIES :=
include $(BUILD_SHARED_LIBRARY)

```

在该模板中，BUILD\_SHARED\_LIBRARY 表示编译一个动态库。

可执行程序、动态库和静态库生成的文件目录如下：

```

out/target/product/*/obj/EXECUTABLE
out/target/product/*/obj/STATIC_LIBRARY
out/target/product/*/obj/SHARED_LIBRARY

```

其目标的文件夹分别为每个模块中 LOCAL\_MODULE 所定义的名字：

```

XXX_intermediates
XXX_shared_intermediates
XXX_static_intermediates

```

综合三类模板，Android.mk 的每个编译模块都是以 include \$(CLEAR\_VARS) 开始以 include \$(BUILD\_XXX) 结束。根据编译系统内的定义分为：

- include \$(BUILD\_STATIC\_LIBRARY) 表示编译成静态库。
- include \$(BUILD\_SHARED\_LIBRARY) 表示编译成动态库。
- include \$(BUILD\_EXECUTABLE) 表示编译成可执行程序。

在 Android.mk 文件中，用 LOCAL\_MODULE\_PATH 和 LOCAL\_UNSTRIPPED\_PATH 来指定目标的安装路径。不同的文件系统路径用以下的宏进行选择。

- TARGET\_ROOT\_OUT: 表示根文件系统。
- TARGET\_OUT: 表示 system 文件系统。
- TARGET\_OUT\_DATA: 表示 data 文件系统。

## 4.2 Android 编译系统介绍

Android 编译系统基于最新版本的 GNU Make，用以完成工具、二进制文件和文档的生成，主要目标是建立更可靠的依赖关系，提高编译效率。Android 系统的下载与编译的主要工作包括：准备阶段、下载源码、编译源码及内核源码。

Google 在开发 Android 系统的同时，为便于 Android 技术的推广使用，降低开发成本，开发了针对不同版本的模拟器，在没有目标开发板的情况下，要实现对硬件的操作可以使用模拟器的内核源码。

编译 Android 系统的步骤如下。

- 1) 初始化编译环境：\$ source build/envsetup.sh。
- 2) 选择编译选项：\$lunch。
- 3) 执行 make 命令开始编译：\$ make。

### 4.2.1 build 系统

Android 编译系统 (build system) 集中于 Android 源码下的 build/core 目录下，用来编译 Android 系统、Android SDK 以及相关文档。该系统主要由 make 文件，Shell 脚本以及 Python 脚本组成，支持多架构 (Linux-x86、Windows、ARM 等)、多语言 (汇编、C、C++、Java 等)、多目标、多编译方式。编译过程如图 4-2 所示。



图 4-2 编译过程



所有的编译生成的文件都将位于 /out 目录下, 该目录下主要有以下几个子目录。

1) /out/host/ : 该目录下包含了针对主机的 Android 开发工具的产物。即 SDK 中的各种工具, 例如: emulator、adb、aapt 等。

2) /out/target/common/ : 该目录下包含了针对设备的共通的编译产物, 主要是 Java 应用代码和 Java 库。

3) /out/target/product/<product\_name>/ : 包含了针对特定设备的编译结果以及平台相关的 C/C++ 库和二进制文件。其中, <product\_name> 是具体目标设备的名称。

4) /out/dist/ : 包含了为多种分发而准备的包, 通过 make disttarget 命令将文件复制到该目录, 默认的编译目标不会产生该目录。

由于 Build 系统中主要的处理逻辑在 make 文件中, Android 编译系统通过各种 .mk 文件和 Shell 脚本共同定义了一个编译框架, 这个框架基于 make 概念, 亦即 Android 的编译系统建立了一个位于 build 目录中编译框架, 其相互关系如图 4-3 所示。

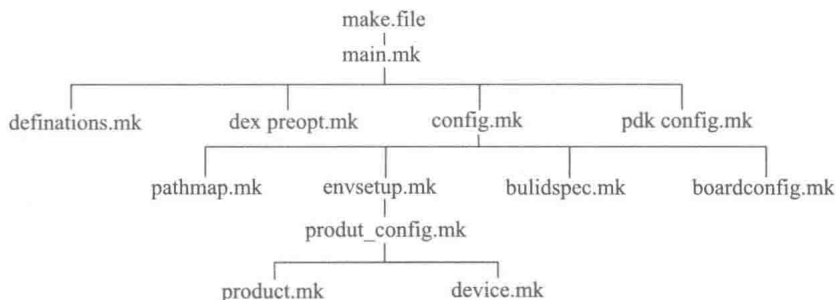


图 4-3 build 目录的编译框架

其中比较重要的 makefile 文件如下。

1) Android.mk : 是 module 和 package 的 makefile, 每个 module 和 package 目录下都会有这么一个文件。

2) AndroidProducts.mk : 为设定 product 配置, 一个 product 表示一个特定的产品版本。Android 通过编译不同的 product 来产生不同的软件配置内容。

3) BoardConfig.mk : 是为 product 主板做设定, 设置一些 Board 的参数, 如 kernel 的传递参数等。

4) \*.mk: 是针对选择的操作系统和 CPU 架构进行相关设定。

5) buidspec.mk: 是位于 source 根目录下为编译之外的设定, 如可以选择要产生的 product、平台、额外的 module/package 等。

## 4.2.2 SDK

### 1. SDK 概述

Android SDK 以 Java 语言为基础, 用户可以使用 Java 来开发 Android 平台上的应用软

件，通过 SDK 提供的一些工具将其打包成 Android 平台使用的 apk 文件，然后再使用 SDK 中的模拟器来模拟和测试该软件在 Android 上的运行效果。

Android SDK 目录及其功能如表 4-7 所示。

表 4-7 SDK 目录及功能

项 目	描 述
add-ons	保存附加库，比如 Google Maps，当然如果已经安装了 Ophone SDK，这里也会有一些类库在里面
docs	Android SDK API 参考文档，所有的 API 都可以在这里找到
market_licensing	为 Android Market 版权保护组件，一般发布付费应用到电子市场，可以用它来反盗版
platforms	每个平台的 SDK 真正的文件，里面会根据 API Level 划分 SDK 版本。这里以 Android 2.2 来说，进入后有一个 android-8 的文件夹，android-8 进入后是 Android 2.2 SDK 的主要文件，其中 ant 为 ant 编译脚本，data 保存着一些系统资源，images 是模拟器映像文件，skins 则是 Android 模拟器的皮肤，templates 是工程创建的默认模板，android.jar 则是该版本的主要 framework 文件，tools 目录里面包含了重要的编译工具，比如 aapt、aidl、逆向调试工具 dexdump 和编译脚本 dx
platform-tools	通用工具，比如 adb、aapt、aidl、dx 等文件，这里和 platforms 目录中 tools 文件夹有些重复
samples	Android SDK 自带的默认示例工程
tools	SDK 根目录下的 tools 文件夹，包含了重要的工具
usb_driver	Android 平台 Google 官方机型的驱动，如 Nexus One、Nexus S，同时也有一些老机型驱动的支持，比如 HTC dream、HTC magic 和 Motorola droid

2. SDK 工具

SDK 中包括各种各样的定制工具，最重要的是 Android 模拟器和 Eclipse 开发工具插件，以及各种在模拟器上针对不同使用场景的工具，具体如表 4-8 所示。

表 4-8 SDK 工具

项 目	描 述
Android 模拟器	运行在计算机上的虚拟移动模拟器，用于应用程序的设计、调试和测试
集成开发环境插件	用于 Eclipse 的开发工具插件，使得构建一个新的 Android 应用的过程自动化、简单化，提供 Android 代码编辑器等功能
调试监视服务	集成在 Dalvik 中，用于管理运行在模拟器或设备上的进程，并协助进行调试。用于选择特定程序进行调试、生成跟踪数据、查看堆和线程数据等
Android 调试桥	用于向模拟器或手机设备安装应用程序的 apk 文件和从命令行访问模拟器或手机设备。也可用于将标准的调试桥连接到运行在 Android 模拟器或手机设备上的应用代码
Android 资源打包工具	通过 aapt 工具创建的 apk 文件，这些文件包含 Android 应用程序的二进制文件和资源文件
Android 接口描述语言	用来生成进程间接口代码
SQLite3 数据库	便于开发者和使用者访问 Android 应用程序创建和使用的 SQLite 数据文件

(续)

项 目	描 述
跟踪显示工具	生成由 Android 应用程序产生的日志数据的图形分析图
创建 SD 卡工具	创建磁盘镜像
DX 工具	将 .class 文件编码为 .dex 文件
生成 ant 构建文件	一个脚本文件, 用于生成 ant 的构建文件。若安装了 ADT 插件的 Eclipse 环境则不需要该文件
Android 虚拟设备	具有独立内核、系统图像和数据分区的虚拟设备来运行 Android 平台

### 3. SDK 开发包

SDK 的开发包包括核心开发包和扩展开发包两类, 如表 4-9 和表 4-10 所示。

表 4-9 核心开发包

项 目	描 述
android.util	包含底层辅助类, 例如特定的容器类、XML 辅助工具类等
android.os	提供基本的操作服务、消息传递和进程间通信 IPC
android.graphics	核心渲染包, 提供图形渲染功能
android.text	提供一套丰富的文本处理工具
android.database	包含底层 API 处理数据库, 方便操作数据库表和数据
android.content	提供各种服务访问手机设备上的数据
android.view	核心用户界面框架
android.widget	提供标准用户界面元素, 例如, list (列表)、buttons (按钮)、layout managers (布局管理器) 等
android.app	提供高层应用程序模型, 实现 Activity
android.provider	提供方便调用系统提供的 content providers 的接口
android.telephony	提供 API 交互和手机设备的通话接口
android.webkit	包含一系列工作在基于 Web 内容的 API

表 4-10 扩展开发包

项 目	描 述
Location-Based Services	定位服务 Android 操作系统支持 GPS API-LBS, 通过集成 GPS 芯片来接收卫星信号, 获取当前手机坐标, 转换成为地图上的具体位置
Media APIs	多媒体接口 Android 平台上集成的影音解码器及相关多媒体 API, 支持 MP3、MP4、高清视频播放等处理
3D Graphics with OpenGL	图形处理 OpenGL Android 平台的游戏功能
Low-Level Hardware Access	底层硬件访问 主要用于控制手机的底层操作, 支持不同设备的操作管理

### 4.3 init 初始化脚本语言介绍

init 进程源自 Linux，是 Linux 内核启动后运行的第一个进程。其他所有系统运行所需的进程都由 init 来创建。在系统启动完成后，init 进程会作为守护进程监视其他进程。对于终结或者进入僵死状态的进程，init 会强制释放该进程所占用的所有系统资源。

作为以 Linux 内核为基础的 Android 系统，也不可避免地继承了 init 进程的诸多特点。

#### 4.3.1 概述

与 Linux 类似，init 是 Android 系统启动后由内核运行的第一个进程，启动过程如图 4-4 所示。

Linux 系统启动时依次执行 start\_kernel() 函数、init\_post() 函数和 run\_init\_process() 函数，之后开始运行 init 进程。Android 的 init 进程主要提供 4 个功能：

- 1) 分析启动 init.rc 脚本文件。
  - 2) 为应用程序访问设备驱动生成设备节点文件。
  - 3) 子进程的创建与终止。
  - 4) 提供 property service 属性服务，进行 Android 系统的属性管理。
- 启动过程具体代码如下所示。

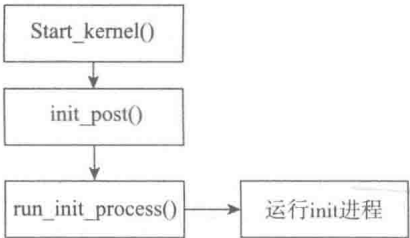


图 4-4 init 进程启动过程

代码清单4-1：启动init进程

```
Static noinline int init_post(void)
{
    if(execute_command){
        run_init_process(execute_command);
    }

    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");
    panic("No init found. Try passing init= option to kernel. "
        "See Linux Documentation/init.txt for guidance.");
}
```

本段代码中，init\_post() 函数通过 if 判断语句调用 run\_init\_process() 函数获取 execute\_command 中注册的进程文件路径。后续的 run\_init\_process() 函数分别在各个目录下查找 init 文件，并启动 init 进程。

#### 4.3.2 init 进程源码分析

同其他进程一样，init 进程也是由一个 main() 函数作为入口，代码如下所示。

代码清单4-2: main()函数

```

[-->init.c]
int main(int argc, char **argv)
{
    ..... act.sa_handler=sigchld_handler;
    act.sa_flags= SA_NOCLDSTOP;
    act.sa_mask= 0;
    act.sa_restorer= NULL;
    sigaction(SIGCHLD, &act, 0);

    .....

    mkdir("/dev", 0755);
    mkdir("/proc", 0755);
    mkdir("/sys", 0755);
    mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
    mkdir("/dev/pts", 0755);
    mkdir("/dev/socket", 0755);
    mount("devpts", "/dev/pts", "devpts", 0, NULL);
    mount("proc", "/proc", "proc", 0, NULL);
    mount("sysfs", "/sys", "sysfs", 0, NULL);

    open_devnull_stdio();
    klog_init();
    property_init();

    init_parse_config_file("/init.rc");
    get_hardware_name(hardware, &revision);
    snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);
    init_parse_config_file(tmp);

    action_for_each_trigger("early-init", action_add_queue_tail);
    drain_action_queue();

    Device_fd = device_init();
    property_init();
    keychord_fd = open_keychord();
    .....
    if (load_565rle_image(INIT_IMAGE_FILE) ) {
        .....
    }
    property_set("ro.bootloader", bootloader[0] ? bootloader : "unknown");
    .....

    action_for_each_trigger("init", action_add_queue_tail);
    drain_act ion_queue() ;
    property_set_fd = start_roproperty_service();

```

```

.....
action_for_each_trigger("early-boot",action_add_queue_tail);
action_for_each_trigger("boot", action_add_queue_tail);
drain_action_queue() ;
..... Ufds[0].fd = device_fd;
Ufds[0] .events = POLLIN;
Ufds[1] .fd = property_set_fd;
Ufds[1] .events = POLLIN;
Ufds[2] .fd = signal_recv_fd;
Ufds[2] .events = POLLIN;
fd_count = 3;
if (keychord_fd > 0) {
ufds[3] .fd = keychord_fd;
ufds[3] . events = POLLIN;
fd_count++;
}
.....

nr = poll(ufds, fd_count, timeout);
if (nr <= 0)
    continue;

for (i = 0; i < fd_count; i++) {
    if (ufds[i].revents == POLLIN) {
        if (ufds[i].fd == get_property_set_fd())
            handle_property_set_fd();
        else if (ufds[i].fd == get_keychord_fd())
            handle_keychord();
        else if (ufds[i].fd == get_signal_fd())
            handle_signal();
    }
}

return 0;
}

```

本段代码中首先注册 `sigaction()` 函数为信号处理器，信号实质上是进程之间通信时相互发送的消息。每个进程在处理消息时都要注册信号处理器程序。

接下来通过 `mkdir()` 函数创建系统运行时所需的文件目录：`/dev`、`/proc` 和 `/sys`。这些目录由 `init` 进程生成，当系统终止时被撤销。

后续代码中，各函数的作用做如下的说明：`klog_init()` 函数设置 `init` 的日志输出设备文件为 `/dev/_kmsg`；`init_parse_config_file()` 函数用于解析 `init.rc` 的配置文件；`get_hardware_name()` 函数通过读取 `/proc/cpuinfo` 获得硬件的 `hardware` 名称；`init_parse_config_file()` 函数用于解析与硬件相关的配置文件。`init` 将动作执行的时间分为 `early-init`、`init`、`early-boot`、`boot` 4 个阶段，因此 `action_for_each_trigger()` 函数根据参数的不同，执行处于各个阶段的动作。

`property_init()` 函数初始化与属性相关的资源；`keychord_fd` 调用 `open_keychord()` 函数初始化 `/dev/keychord` 设备；`load_565rle_image()` 函数将 `INIT IMAGE FILE` 定义为 `/initlogo.rle`；`property_set()` 函数设置包括属性名和属性值的属性项；`start_roproperty_service()` 函数启动属性服务；`action_for_each_trigger()` 函数根据传入参数不同，分别执行位于 `init`、`early-boot` 和 `boot` 阶段的动作。

`init` 会关注以下事件：`device_fd` 用于监听来自内核的 `uevent` 事件；`property_set_fd` 用于监听来自属性服务器的事件；`signal_recv_fd` 由 `socketpair` 创建，它的事件来自另外一个 `sockel`；如果 `keychord_fd` 设备初始化成功，则 `init` 也会关注来自这个设备的事件。

`poll()` 函数用于监听、等待事件发生，通过 `for` 循环语句完成对 `uevent` 事件、属性服务器的事件、`signal` 事件和 `keychord` 事件的处理。

由上段代码可知，`init` 进程的事件处理如图 4-5 所示。

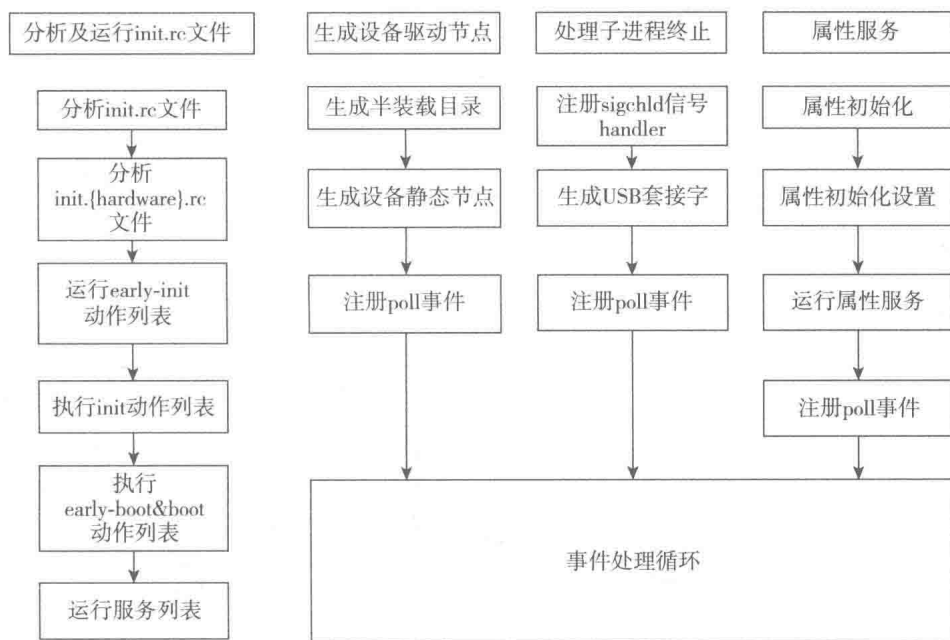


图 4-5 `init` 进程的事件处理

### 4.3.3 脚本文件的创建与分析

`init.rc` 文件是 `init` 启动后执行的启动脚本文件，该文件具有记录 `init` 进程的执行功能。在 Linux 系统中，`init` 进程位于系统的 `/etc/rc.d` 目录下，并保存着设置环境变量的脚本，是启动时的可执行文件。

在 Android 系统中，定义与执行环境变量由 `init.rc` 和 `init.{hardware}.rc` 两个文件实现。`init.rc` 在系统运行过程中用于通用的环境变量设置及与进程相关的定义，`init.{hardware}.rc` 文

件用于定义 Android 在不同硬件平台下的特定进程和环境设置等。两个文件处理方式和语法相同，因此，下文仅分析 init.rc 文件。

### 1. init.rc 文件

init.rc 文件大致可以分为两部分：以 on 开头的动作列表 action list，用于创建所需目录，以及为特定文件指定权限；以 service 开头的服务列表 service list，用来记录初始化程序所需要启动的一些程序。

### 2. action list

on 用来创建所需的目录，为特定文件设置权限，实现代码如下所示。

代码清单4-3: on init()

---

```
on init
sysclktz 0

loglevel 3

# setup the global environment
export PATH /sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin
export LD_LIBRARY_PATH /vendor/lib:/system/lib
export ANDROID_BOOTLOGO 1
export ANDROID_ROOT /system
export ANDROID_ASSETS /system/app
export ANDROID_DATA /data
export ANDROID_STORAGE /storage
export ASEC_MOUNTPOINT /mnt/asec
export LOOP_MOUNTPOINT /mnt/obb
export BOOTCLASSPATH /system/framework/core.jar: /system/framework/core-junit.
jar: /system/framework/bouncycastle.jar: /system/framework/ext.jar: /system/
framework/framework.jar: /system/framework/telephony-common.jar: /system/
framework/mms-common.jar: /system/framework/android.policy.jar: /system/
framework/services.jar: /system/framework/apache-xml.jar

on fs
# mount mtd partitions
# Mount /system rw first to give the filesystem a chance to save a checkpoint
mount yaffs2 mtd@system /system
mount yaffs2 mtd@system /system ro remount
mount yaffs2 mtd@userdata /data nosuid nodev
mount yaffs2 mtd@cache /cache nosuid nodev
on post-fs-data
# We chown/chmod /data again so because mount is run as root + defaults
chown system system /data
chmod 0771 /data
```

---

在本段代码中，on init 的环境变量设置部分主要设置运行根文件系统命令的目录，以及



程序编译时需要的库目录。在根文件系统主要挂载 /system 与 /data 两个目录。

### 3. service list

service 用来记录 init 进程启动。由 init 进程启动的子进程——daemon 进程启动，实现代码如下所示。

代码清单4-4: service init()

---

```
## Daemon processes to be run by init.
##
service console /system/bin/sh
    class core
    console
    disabled
    user shell
    group log
on property:ro.debuggable=1
    start console

# adbd is controlled via property triggers in init.<platform>.usb.rc
service adbd /sbin/adbd
    class core
    socket adbd stream 660 system system
    disabled
    seclabel u:r:adbd:s0
# adbd on at boot in emulator
on property:ro.kernel.qemu=1
    start adbd

service servicemanager /system/bin/servicemanager
    class core
    user system
    group system
    critical
    onrestart restart zygote
    onrestart restart media
    onrestart restart surfaceflinger
    onrestart restart drm
service vold /system/bin/vold
    class core
    socket vold stream 0660 root mount
    ioprio be 2
```

---

在本段代码中，service 后的第一个字符串表示服务的名称，第二个字符串表示服务的路径。第二行是服务的附加内容，主要包含运行权限、条件以及重启等相关选项。本段代码中的服务全部注册在服务列表中，init 进程从该列表中选择相应的服务进行启动。

#### 4. init.rc 脚本文件解析

init.rc 脚本文件的解析由 `init_parse_config_file()` 函数实现。该函数的参数用来指定待解析文件的存放路径。函数根据其参数指定的地址读取文件，生成连续的字符串，并对字符串进行解析。该函数主要调用两个函数：`read_file()` 函数和 `parse_config()` 函数，代码如下所示。

代码清单4-5: `init_parse_config_file()`

---

```
int init_parse_config_file(const char *fn)
{
    char *data;
    data = read_file(fn, 0);
    if (!data) return -1;
    parse_config(fn, data);
    DUMP();
    return 0;
}
```

---

在本段代码中，`read_file()` 函数用于读取文件，函数指定了文件读取的内存，并保存为字符串的形式，返回字符串在内存中的初始地址。`parse_config()` 函数分析 `read_file()` 函数的返回值（字符串），并生成动作列表和服务列表。`parse_config()` 函数实现对字符串的逐行分析，函数的实现代码如下所示。

代码清单4-6: `parse_config()`

---

```
static void parse_config(const char *fn, char *s)
{
    .....
    for (;;)
    {
        switch (next_token(&state))
        {
            case T_EOF:
                state.parse_line(&state, 0, 0);
                goto parser_done;
            case T_NEWLINE:
                state.line++;
                if (nargs)
                {
                    int kw = lookup_keyword(args[0]);
                    if (kw_is(kw, SECTION))
                    {
                        state.parse_line(&state, 0, 0);
                        parse_new_section(&state, kw, nargs, args);
                    }
                    else

```

---

```

        {
            state.parse_line(&state, nargs, args);
        }
        nargs = 0;
    }
    case T_TEXT:
        if (nargs < INIT_PARSER_MAXARGS)
        {
            args[nargs++] = state.text;
        }
        break;
    }
}
.....
}

```

本段代码中，`next_token()` 函数是以行为单位分割参数传递过来的字符串；`case T_EOF:` 语句用来遍历到文件结尾，使用 `goto` 语句解析 `import` 的 `.rc` 文件；`case T_NEWLINE:` 语句指定当前行为结束行；`lookup_keyword()` 函数用于返回 `init.rc` 脚本中每行首个单词在 `keyword_list` 结构体数组中的数组编号。`keyword_list` 结构体数组由 `KEYWORD` 列表构成，每个代码由 `KEYWORD` 宏生成。`kw_is(kw, SECTION)` 函数返回 `keyword_list` 结构体数组在 `KEYWORD` 列表中的编号；`parse_new_section()` 函数将 `kw_is()` 宏筛选出的命令注册到动作列表和服务列表中。

#### 4.3.4 创建设备节点文件

Android 与 Linux 一样通过设备驱动来访问硬件设备，设备节点文件是设备驱动的逻辑文件，应用程序通过设备节点文件来访问设备驱动程序，Linux 系统使用 `mknod` 进程来创建设备节点文件，Android 系统使用 `init` 进程来创建设备节点文件。

`init` 进程创建的设备节点文件分为静态节点文件和动态节点文件，分别对应于已经定义好的冷插拔（Cold Plug）设备和系统运行起来后插入的热插拔（Hot Plug）设备。

冷插拔设备使用静态节点文件，以预先定义的设备信息为基础，当 `init` 进程被启动运行时，统一创建设备节点文件，实现代码如下所示。

代码清单4-7: `init`节点创建过程

```

void device_init(void)
{
    suseconds_t t0, t1;
    struct stat info;
    int fd;

    device_fd = uevent_open_socket(64*1024, true);

```

```

    if(device_fd < 0)
        return;

    fcntl(device_fd, F_SETFD, FD_CLOEXEC);
    fcntl(device_fd, F_SETFL, O_NONBLOCK);
    if (stat(coldboot_done, &info) < 0) {
        .....
    }
static void do_coldboot(DIR *d)
{
    struct dirent *de;
    int dfd, fd;

    dfd = dirfd(d);

    fd = openat(dfd, "uevent", O_WRONLY);
    if(fd >= 0) {
        write(fd, "add\n", 4);
        close(fd);
        handle_device_fd();
    }

    while((de = readdir(d))) {
        DIR *d2;

        if(de->d_type != DT_DIR || de->d_name[0] == '.')
            continue;

        fd = openat(dfd, de->d_name, O_RDONLY | O_DIRECTORY);
        if(fd < 0)
            continue;

        d2 = fdopendir(fd);
        if(d2 == 0)
            close(fd);
        else {
            do_coldboot(d2);
            closedir(d2);
        }
    }
}

struct uevent {
    const char *action;
    const char *path;
    const char *subsystem;
    const char *firmware;

```

```

const char *partition_name;
int partition_num;
int major;
int minor;
};

void handle_device_fd()
{
    char msg[UEVENT_MSG_LEN+2];
    int n;
    while ((n = uevent_kernel_multicast_recv(device_fd, msg, UEVENT_MSG_LEN)) > 0) {
        if(n >= UEVENT_MSG_LEN) /* overflow -- discard */
            continue;

        msg[n] = '\0';
        msg[n+1] = '\0';

        struct uevent uevent;
        parse_event(msg, &uevent);

        handle_device_event(&uevent);
        handle_firmware_event(&uevent);
    }
}

static void handle_device_event(struct uevent *uevent)
{
    if (!strcmp(uevent->action, "add"))
        fixup_sys_perms(uevent->path);

    if (!strncmp(uevent->subsystem, "block", 5)) {
        handle_block_device_event(uevent);
    } else if (!strncmp(uevent->subsystem, "platform", 8)) {
        handle_platform_device_event(uevent);
    } else {
        handle_generic_device_event(uevent);
    }
}

static void make_device(const char *path,
                        const char *upath,
                        int block, int major, int minor)
{
    unsigned uid;
    unsigned gid;
    mode_t mode;
    dev_t dev;

```

```

mode = get_device_perm(path, &uid, &gid) | (block ? S_IFBLK : S_IFCHR);
dev = makedev(major, minor);
setegid(gid);
mknod(path, mode, dev);
chown(path, uid, -1);
setegid(AID_ROOT);
}

```

init 首先调用 `device_init()` 函数，创建一个 socket 来接收 uevent，再通过 `cold_boot()` 函数调用 `do_coldboot()` 函数对内核启动时注册到 `/sys` 下的驱动程序进行冷插拔处理。`do_coldboot()` 函数接收参数传递过来的目录路径，通过该路径查找保存的 uevent 文件，向文件中写入信息，而后强制启动 uevent。在 `handler_device_fd()` 函数中接收相关的 uevent 信息，并写入 uevent 结构体当中，调用 `handle_device_event()` 函数。此函数先检查 uevent 结构体的 `subsystem` 变量，而后在 `/dev` 目录下创建子目录。创建完成后调用 `make_device()` 函数创建节点文件。

其中 uevent 是内核向用户空间进程传递信息的信号系统，即在添加或删除设备时，内核使用 uevent 将设备信息传递到用户空间。uevent 包含设备名称、类别、主设备号、次设备号、设备节点文件创建的目录等信息。

init 对于热插拔的动态设备，一般在系统运行过程中，当有设备插入 USB 端口时，init 进程会收到这一事件，为插入的设备动态创建设备节点文件。

`init.c-main()` 是设备事件处理函数，借助循环的方式实现事件处理。首先 init 进程的事件处理循环调用 `poll()` 函数，监听来自驱动程序的 uevent，然后调用 `handle_device_fd()` 创建设备节点。

### 4.3.5 子进程的创建与终止

init 进程读取并分析 `init.rc` 文件，获得服务列表，并从中依次启动服务子进程。启动的主要进程如下。

- `sh`：搭载 Android 的机器终端，连接串口或 `adbd` 时，提供控制台输入输出的 Shell 程序。
- `adbd`：指 Android Debug Bridge，用来管理 QEMU 模拟器或实际机器的状态。该工具运行在目标机器上充当服务器，运行在 PC 上充当连接服务器的客户端。
- `servicemanager`：在 init 进程后，用来管理系统中的服务。
- `vuld`：指 volume dameon，用来挂载 / 管理 USB 存储或 SD 卡设备。
- `Playmp3`：指在 Android 启动时输出启动声音。

除此之外，init 还启动其他进程。若某个进程出现意外终止时，init 进程要对意外终止的进程重新启动。

如图 4-6 所示，当 init 的子进程意外终止时，会向父进程 init 进程传递 `sigchld` 信号，init

进程接收该信号，检查进程选项是否设置为 oneshot，若设置为 oneshot，init 进程将放弃重启进程，否则重启进程。

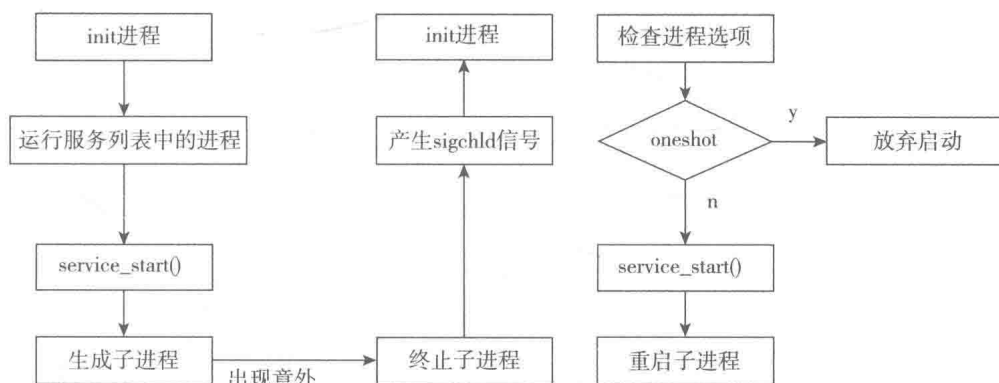


图 4-6 init 进程处理 sigchld 信号

### 4.3.6 属性服务

属性变更请求也属于 init 事件处理中的事件。在 Android 平台系统中开辟了属性存储区域，提供访问该区域的 API，使得运行中的进程能够共享系统运行时所需要的各种设置值。属性参数由 key 和 value 构成，其形式为 key = value。为提高访问的安全性，Android 系统增加了访问权限设置，即系统中所有运行的进程都可以访问属性值，但仅有 init 进程才能修改属性值。其他进程若要修改属性值，必须向 init 进程提出请求，最终由 init 进程负责修改。在此过程中，init 进程会先检查各属性的访问权限，而后再修改属性值。当属性值更改后，定义在 init.rc 文件中的某个特定条件得到满足，则与此条件相匹配的动作就会发生。每个动作都有一个触发器 trigger，它决定动作的执行时间，记录在 on property 关键字后的命令即被执行。

在 main() 函数中，调用了 property\_init() 函数，用来初始化属性域，实现代码如下所示。

代码清单4-8：属性初始化代码

```

void property_init(bool load_defaults)
{
    init_property_area();
    if (load_defaults)
        load_properties_from_file(PROP_PATH_RAMDISK_DEFAULT);
}

int main(int argc, char **argv)
{...
    property_init();
    ...
}
  
```

本段代码中为保存属性值，property\_init() 函数首先申请一块共享内存区域，用作共享内存 Ashmem (Android Shared Memory)。外部进程通过访问这个区域获取属性值，但是不能通过访问共享内存域的方式来更改属性值。

在进行属性变更的时候，init 进程会调用权限检查函数 handle\_property\_set\_fd()，实现代码如下所示。

代码清单4-9: 权限检查函数代码

---

```

void handle_property_set_fd()
{
    ...
    if (getsockopt(s, SOL_SOCKET, SO_PEERCRED, &cr, &cr_size) < 0)
    {
        close(s);
        ERROR("Unable to recieve socket options\n");
        return;
    }
    ...
    switch(msg.cmd) {
    case PROP_MSG_SETPROP:
        ...
        if(memcmp(msg.name, "ctl.", 4) == 0)
        {
            close(s);
            if (check_control_perms(msg.value, cr.uid, cr.gid))
            {
                handle_control_message((char*) msg.name + 4, (char*) msg.value);
            }
            else {
                ERROR("sys_prop: Unable to %s service ctl [%s] uid:%d gid:%d pid:%d\n",
                    msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);
            }
        }
        else {
            if (check_perms(msg.name, cr.uid, cr.gid))
            {
                property_set((char*) msg.name, (char*) msg.value);
            }
            else {
                ERROR("sys_prop: permission denied uid:%d name:%s\n",
                    cr.uid, msg.name);
            }
            close(s);
        }
        ...
    }
}

```

---



本段执行过程中, 首先获取 `SO_PEERCREC` 值, 用于检查传递信息的进程的访问权限。在结构体中存储传递信息进程的 `uid`、`pid`、`gid` 值。通过此结构体中的值及消息的类型, 检查进程的访问权限。在属性消息中, 以 `ctl` 开头的消息用来请求进程启动与终止。`check_control_perms()` 函数检查访问权限, 仅有 `system server`、`root` 以及相关进程才能使用 `ctl` 消息执行终止或启动进程。`property_set()` 函数用于更改属性值。

## 4.4 Zygot

### 4.4.1 Zygot 概述

Zygot 应用于 Dalvik 虚拟机的内存管理, 由 `init` 进程根据 `init.rc` 文件中的配置项创建。Zygot 最初的名字是由 `android.mk` 文件指定的 `app_process`, 在运行过程中由 Linux 下的 `pctl` 系统调用将名字改为 Zygot, 其对应的源文件依然是 `app_main.cpp`。

Zygot 创建了第一个 Java 虚拟机, 并且创建进程 `system_server`, 这是绝大多数系统服务的守护进程。之后为每一个应用程序创建一个相应的 Activity 进程。具体步骤如下:

- 1) 创建 `AppRuntime` 对象, 并且调用该对象的 `start`。由 `AppRuntime` 来控制后续的活动。
- 2) 调用 `startVM` 创建 Java 虚拟机, 由 `startReg` 来注册 JNI 函数。
- 3) 通过 JNI 调用 `com.android.internal.os.ZygotInit` 类的 `main` 函数, 开始进入 Java 世界。
- 4) 调用 `registerZygotSocket()`、`preloadClasses()` 和 `preloadResorces()` 三个函数, 响应其他应用程序。
- 5) 通过调用 `startSystemServer()` 分裂一个子进程 `system_server` 来提供服务。
- 6) 调用 `runSelectLoopMode()` 函数。

Zygot 启动新应用的过程如图 4-7 所示。Zygot service 接受启动新程序的请求, 通过调用 Zygot loop 函数, 使用 `fork` 机制为这个程序设置一个启动进程, 通过该进程启动程序。同时, 通过 COW (Copy on Write) 方式对运行在内存中的进程实现最大程度的复用, 并通过库共享有效地降低内存的使用量 (footprint)。

下面将对 Zygot 进行两方面的分析: `AppRuntime` 分析和 `system_server` 分析。

### 4.4.2 AppRuntime 分析

Zygot 启动应用程序的过程如图 4-8 所示。Zygot 进程调用 `fork()` 函数创建 Zygot' 子

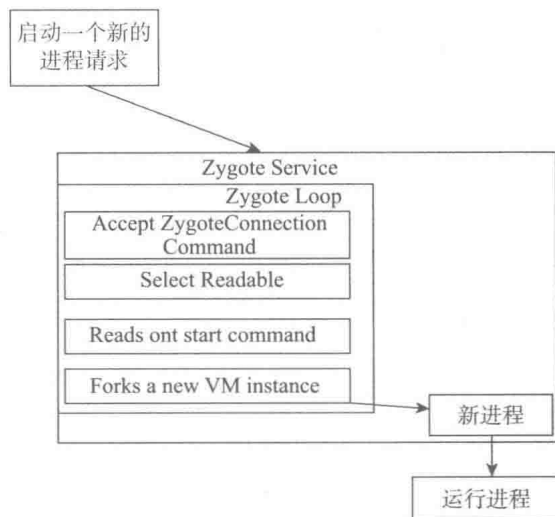


图 4-7 Zygot 接受请求启动应用程序过程

进程，子进程 `Zygote'` 共享父进程 `Zygote` 的代码区与连接信息。同时，新的 Android 应用程序 A 被动态地加载到复制出的 Dalvik 虚拟机上，而后 `Zygote'` 进程将执行流程交给应用程序 A 类，新的应用程序开始运行。新生的应用程序 A 会使用已有 `Zygote` 进程的库与资源的连接信息，所以运行速度快。图 4-8 描述了 `Zygote` 运行后，新的 Android 应用程序 A 的运行过程。

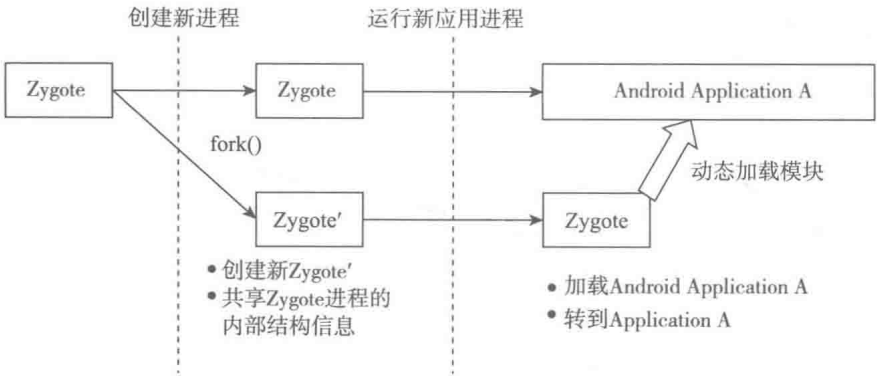


图 4-8 在 Android 平台上运行新应用程序的过程

代码清单4-10: AppRuntime对象的生成

```
int main(int argc, const char* const argv[])
{
    mArgC = argc;
    mArgV = argv;
    mArgLen = 0;
    for (int i=0; i<argc; i++)
    {
        mArgLen += strlen(argv[i]) + 1;
    }
    mArgLen--;
    AppRuntime runtime;
    int i = runtime.addVmArguments(argc, argv);
    if (i < argc)
    {
        runtime.mParentDir = argv[i++];
    }
    ....
}
```

本段代码的主要作用是完成 `AppRuntime` 对象的生成。代码中 `main` 函数的启动参数在 `init.rc` 中设置：`-Xzygote /system/bin --zygote --start-system-server`；`AppRuntime runtime` 定义了一个 `AppRuntime` 类的参数 `runtime`，`AppRuntime` 类继承自 `AndroidRuntime` 类，用于初始化和启动 Dalvik 虚拟机，为 Android 应用程序运行做好准备；`runtime.addVmArguments(argc,`

argv) 表明在 Dalvik 虚拟机运行前, 通过 AppRuntime 对象分析环境变量以及运行的参数, 并以此生成虚拟机选项; runtime.mParentDir 将 runtime 的 mParentDir 设置为 /system/bin。

而在下段代码中, 完成了 Zygote 的更名操作, 代码如下所示。

代码清单4-11: Zygote更名

---

```

if (i < argc)
{
    arg = argv[i++];
    if (0 == strcmp("--zygote", arg))
    {
        boolstartSystemServer = (i < argc) ?
            strcmp(argv[i], "--start-system-server") == 0 : false;
        setArgv0(argv0, "zygote");
        set_process_name("zygote");
        runtime.start("com.android.internal.os.ZygoteInit", startSystemServer);
    }
}
else
{
    return 10;
}

```

---

在本段代码中, strcmp("--zygote", arg) 用于检查类的名称是否为 "--zygote"; runtime.start() 是 AppRuntime 中 start() 成员函数, 生成并初始化虚拟机, 而后将 ZygoteInit 类加载至虚拟机中, runtime.start() 函数调用过程中传递的第一个参数 com.android.internal.os.ZygoteInit 是完全限定名 (Full Qualified Name, FQN)。FQN 中包含类所在的包以及类的名称。当 FQN 传递给类加载器时, 类加载器就会将包的名称解析为相应路径, 而后再在该路径下查找并加载名称为 ZygoteInit 的类。--startSystemServer 作为第二个参数传递给 app\_process, 而后 AppRuntime 中 start() 成员函数的第二个参数被设置为 true。

AppRuntime.cpp 源代码的存放路径是 framework/base/core/jni/AppRuntime.cpp, 当参数 className 的值是 com.android.internal.os.ZygoteInit、startSystemServer 的值是 true 时, 代码如下所示。

代码清单4-12: AppRuntime调用

---

```

Void Android Runtime::start(constchar*className,constboolstartSystemServer)
{
    char*slashClassName=NULL;
    char*cp;
    JNIEnv*env;
    blockSigpipe();
    .....
    constchar*rootDir=getenv("ANDROID_ROOT");
    if(rootDir==NULL){

```

---

```

rootDir="/system";
.....
setenv("ANDROID_ROOT",rootDir,1);
}
.....
if(startVm(&mJavaVm,&env)!=0)
goto bail;
.....
if(startReg(env)<0){
LOGE("Unable to register all android natives\n");
Goto bail;
}
.....
}

```

在这段代码中，blockSigpipe() 函数用于处理 sigpipe 信号；getenv("ANDROID\_ROOT") 函数用于将环境变量 ANDROID\_ROOT 设置为 /system。

在 AppRuntime.cpp 调用过程中，实现 Java 环境的构建的两个步骤：创建虚拟机——startVm；注册 JNI 函数——startReg。

### 1. 创建虚拟机——startVm

AndroidRuntime::startVm() 这个函数主要用于设置 Java 虚拟机的启动参数：JNI check 选项。JNI check 用于 Native 层实现对所调用的 JNI 函数的检查，例如，调用 NEWUTFString 函数时，检查传入的字符串是否符合 UTF-8 的要求。JNI check 还能检查资源是否被正确释放，但是这个选项在应用过程中也会影响系统运行的速度。具体的函数代码如下。

代码清单4-13：创建虚拟机

```

int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)
{
property_get("dalvik.vm.checkjni", propBuf, "");
if (strcmp(propBuf, "true") == 0) {
checkJni = true;
} else if (strcmp(propBuf, "false") != 0) {
/* property is neither true nor false; fall back on kernel parameter */
property_get("ro.kernel.android.checkjni", propBuf, "");
if (propBuf[0] == '1') {
checkJni = true;
}
}
.....
strcpy(heapsizeOptsBuf, "-Xmx");
property_get("dalvik.vm.heapsize", heapsizeOptsBuf+4, "16m");
opt.optionString = heapsizeOptsBuf;
mOptions.add(opt);
}

```

```

if (checkJni) {
    opt.optionString = "-Xcheck:jni";
    mOptions.add(opt);
    opt.optionString = "-Xjnimreflimit:2000";
    mOptions.add(opt);
}
if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
    LOGE("JNI_CreateJavaVM failed\n");
    goto bail;
}
result = 0;
bail:
free(stackTraceFile);
return result;
}

```

在这段代码中，`property_get("dalvik.vm.heapsize", heapsizeOptsBuf+4, "16m")` 函数设置虚拟机最大 `heapsize`，默认为 16M。在多数情况下，通常确保在进行大尺寸图片操作的过程中虚拟机能够正常运行，一般会将这个值设置为 32M；`JNI_CreateJavaVM()` 函数用于创建虚拟机，`pEnv` 返回当前线程的 `JNIEnv` 变量。

## 2. 注册 JNI 函数——startReg

在这个部分中，通过 `androidSetCreateThreadFunc()` 函数设置创建线程的函数为 `javaCreateThreadEtc`，具体的代码如下。

代码清单4-14：注册JNI

```

int AndroidRuntime::startReg(JNIEnv* env)
{
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);
    env->PushLocalFrame(200);
    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);
    createJavaThread("fubar", quickTest, (void*) "hello");
    return 0;
}

```

在上段代码中，通过 `register_jni_procs()` 函数注册了 JNI 函数，设置 `gRegJNI` 为全局数组。`register_jni_procs()` 函数和 `gRegJNI` 数组设置代码如代码清单 4-15 和代码清单 4-16 所示。

代码清单4-15：register\_jni\_procs()函数

```

static int register_jni_procs(const RegJNIRec array[], size_t count, JNIEnv* env)
{

```

```

for (size_t i = 0; i < count; i++)
{
    if (array[i].mProc(env) < 0)
    {
        return -1;
    }
}
return 0;
}

```

代码清单4-16: 设置gRegJNI数组

```

static const RegJNIRec gRegJNI[] = {
    REG_JNI(register_android_debug_JNITest),
    REG_JNI(register_com_android_internal_os_RuntimeInit),
    REG_JNI(register_android_os_SystemClock),
    REG_JNI(register_android_util_EventLog),
    REG_JNI(register_android_util_Log),
    .....
};

```

register\_jni\_procs() 函数中调用了数组元素的 mProc 函数, 为 Java 类注册一个 JNI 函数。gRegJNI 数组在设置过程中将所有的方法和变量都进行了定义。

至此, Java 的应用框架已经建立起来了。

#### 4.4.3 system\_server 分析

system\_server 进程由 ZygoteInit.main 函数中的 startSystemServer 函数启动。该函数定义在 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java 文件中, 代码如下所示。

代码清单4-17: system\_server进程启动

```

Public class ZygoteInit {
    .....
    String args[] = {
        "--setuid=1000",
        "--setgid=1000",
        "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,3001,3002,3003",
        "--capabilities=130104352,130104352",
        "--runtime-init",
        "--nice-name=system_server",
        "com.android.server.SystemServer",
    };
    ZygoteConnection.Arguments parsedArgs = null;
    int pid;
    Try {
        parsedArgs = new ZygoteConnection.Arguments(args);

```

```

.....
Pid=Zygote.forkSystemServer(
parsedArgs.uid,parsedArgs.gid,
parsedArgs.gids,debugFlags,null,
parsedArgs.permittedCapabilities,
parsedArgs.effectiveCapabilities);
.....}
If(pid==0){
handleSystemServerProcess(parsedArgs);
}
Returntrue;
}
.....
}

```

本段代码执行以下功能：定义 String[] 数组、调用 forkSystemServer() 函数和启动 SystemServer 组件。

String[] 数组主要包含启动进程的相关信息，其中最后一项 com.android.server.SystemServer 用于指定新进程启动后所装载的第一个 Java 类。

函数 forkSystemServer() 的作用与 forkAndSpecialize() 相似，是当前的 Zygote 进程孵化出新的进程。

SystemServer 组件由 Zygote.forkSystemServer() 函数创建的新进程启动，返回值 pid 等于 0 就会创建新进程，并执行 handleSystemServerProcess() 函数。

forkSystemServer() 函数是一个 native 函数，实现在 dalvik\_system\_zygote.c 中，代码如下所示。

代码清单4-18: forkSystemServer()函数

```

static void Dalvik_dalvik_system_Zygote_forkSystemServer(
    const u4* args, JValue* pResult)
{
    pid_t pid;
    pid = forkAndSpecializeCommon(args, true);
    if (pid > 0) {
        int status;

        LOGI("System server process %d has been created", pid);
        gDvm.systemServerPid = pid;
        if (waitpid(pid, &status, WNOHANG) == pid) {
            LOGE("System server process %d has died. Restarting Zygote!", pid);
            kill(getpid(), SIGKILL);
        }
    }
    RETURN_INT(pid);
}

```

forkAndSpecializeCommon() 函数完成子进程的创建和属性的设置；gDvm.systemServerPid 保存 system\_server 的进程 id；在函数结束前检查 system\_server 是否退出，如果退出，执行 kill() 函数。

SystemService 进程的启动和调用流程如图 4-9 所示。

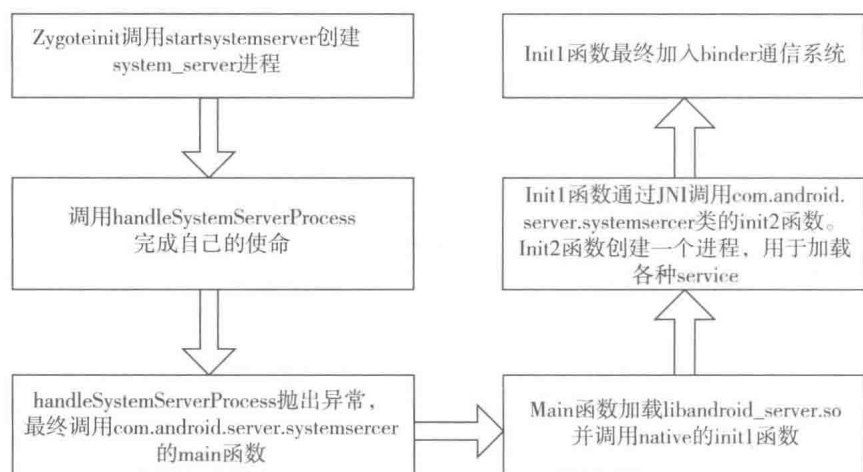


图 4-9 SystemServer 调用流程

handleSystemServerProcess() 函数定义可以在 ZygoteInit.java 文件中找到。该文件路径为 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java。

这个函数的主要功能是：调用 closeServerSocket() 函数关闭 Socket 服务端；调用 set Capabilities() 函数做一些额外的运行环境配置；调用函数 RuntimeInit.zygoteInit() 函数来进一步执行启动 SystemServer 进程的操作，代码如下所示。

代码清单4-19：启动SystemServer进程

```

PrivatestaticvoidhandleSystemServerProcess(
    ZygoteConnection.ArgumentsparsedArgs)
    ThrowsZygoteInit.MethodAndArgsCaller{
    closeServerSocket();
    set Capabilities (parsedArgs.permittedCapabilities,
        parsedArgs.effectiveCapabilitles);
    RuntimeInit.zygoteInit(parsedArgs.remainingArgs);
}
  
```

RuntimeInit.zygoteInit() 函数定义在 frameworks/base/core/java/com/android/internal/os/RuntimeInit.java 文件中，实现代码如下所示。

代码清单4-20：RuntimeInit.zygoteInit()函数

```

PublicclassRuntimeInit{
    .....
  
```



```

PublicStaticfinalvoidzygoteInit(String[]argv)
throwsZygoteInit.MethodAndArgsCaller{
.....
Commoninit();
zygoteInitNative();
.....
StringstartClass=argv[curArg++];
String[]startArgs=newString[argv.length-curArg];
System.arraycopy(argv,curArg,startArgs,0,startArgs.length);
invokeStaticMain(startClass,startArgs);
}
.....
}

```

RuntimeInit.zygoteInit() 函数通过 Commoninit() 函数完成常规属性的初始化操作；通过 zygoteInitNative() 函数完成 Native 层的初始化，与 binder 通信系统建立联系，函数实现在 AndroidRuntime.cpp 当中，如代码清单 4-21 所示。通过 invokeStaticMain() 函数，实现对 com.android.server.SystemServer 类的 main 函数的调用，如代码清单 4-22 所示。

代码清单4-21: zygoteInitNative()函数

```

static void com_android_internal_os_RuntimeInit_zygoteInit(JNIEnv* env, jobject clazz)
{
    gCurRuntime->onZygoteInit();
}

int main(int argc, const char* const argv[])
{
    ...
    AppRuntime runtime;
    ...
}

class AppRuntime : public AndroidRuntime
{
public:
    AppRuntime()
        : mParentDir(NULL)
        , mClassName(NULL)
        , mClass(NULL)
        , mArgC(0)
        , mArgV(NULL)
    {
    }

#ifdef 0
    const char* getParentDir() const
    {

```

```

        return mParentDir;
    }
#endif

    const char* getClassName() const
    {
        return mClassName;
    }

    virtual void onVmCreated(JNIEnv* env)
    {
        if (mClassName == NULL) {
            return;
        }
        char* slashClassName = toSlashClassName(mClassName);
        mClass = env->FindClass(slashClassName);
        if (mClass == NULL) {
            LOGE("ERROR: could not find class '%s'\n", mClassName);
        }
        free(slashClassName);

        mClass = reinterpret_cast<jclass>(env->NewGlobalRef(mClass));
    }

    virtual void onStart()
    {
        sp<ProcessState> proc = ProcessState::self();
        LOGV("App process: starting thread pool.\n");
        proc->startThreadPool();

        AndroidRuntime* ar = AndroidRuntime::getRuntime();
        ar->callMain(mClassName, mClass, mArgC, mArgV);

        IPCThreadState::self()->stopProcess();
    }

    virtual void onZygoteInit()
    {
        sp<ProcessState> proc = ProcessState::self();
        LOGV("App process: starting thread pool.\n");
        proc->startThreadPool();
    }

    virtual void onExit(int code)
    {
        if (mClassName == NULL) {
            // if zygote

```

```

        IPCThreadState::self()->stopProcess();
    }

    AndroidRuntime::onExit(code);
}

const char* mParentDir;
const char* mClassName;
jclass mClass;
int mArgC;
const char* const* mArgV;
};

```

---

#### 代码清单4-22: invokeStaticMain()函数

---

```

private static void invokeStaticMain(String className, String[] argv)
    throws ZygoteInit.MethodAndArgsCaller {
    Class<?> cl;

    try {
        cl = Class.forName(className);
    } catch (ClassNotFoundException ex) {
        throw new RuntimeException(
            "Missing class when invoking static main " + className,
            ex);
    }

    Method m;
    try {
        m = cl.getMethod("main", new Class[] { String[].class });
    } catch (NoSuchMethodException ex) {
        throw new RuntimeException(
            "Missing static main on " + className, ex);
    } catch (SecurityException ex) {
        throw new RuntimeException(
            "Problem getting static main on " + className, ex);
    }

    int modifiers = m.getModifiers();
    if (! (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
        throw new RuntimeException(
            "Main method is not public and static on " + className);
    }
    throw new ZygoteInit.MethodAndArgsCaller(m, argv);
}

```

---

main() 函数被启动后, 将加载 libandroid\_server.so 库, 这个库所包含的源码文件在文件夹 framework/base/services/jni 下, 在 main 函数中启动 init1() 函数, 代码如下所示。

代码清单4-23: 启动init1()函数

---

```
public static void main(String[] args) {
    ...
    System.loadLibrary("android_servers");
    init1(args);
}
```

---

init1 是 native 函数, 在 com\_android\_server\_SystemServer.cpp 中实现, 代码如下所示。

代码清单4-24: init1()的实现

---

```
extern "C" int system_init();
static void android_server_SystemServer_init1(JNIEnv* env, jobject clazz)
{
    system_init();
}
```

---

通过 system\_init() 调用另外一个函数, 这个函数的实现在 system\_init.cpp 中, 具体代码如下所示。

代码清单4-25: 创建SensorService服务

---

```
extern "C" status_t system_init()
{
    LOGI("Entered system_init()");

    sp<ProcessState> proc(ProcessState::self());

    sp<IServiceManager> sm = defaultServiceManager();
    LOGI("ServiceManager: %p\n", sm.get());

    sp<GrimReaper> grim = new GrimReaper();
    sm->asBinder()->linkToDeath(grim, grim.get(), 0);

    char propBuf[PROPERTY_VALUE_MAX];
    property_get("system_init.startsurfaceflinger", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        SurfaceFlinger::instantiate();
    }

    property_get("system_init.startsensorsservice", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        SensorService::instantiate();
    }

    LOGI("System server: starting Android runtime.\n");
}
```

---

```

AndroidRuntime* runtime = AndroidRuntime::getRuntime();

LOGI("System server: starting Android services.\n");
JNIEnv* env = runtime->getJNIEnv();
if (env == NULL) {
    return UNKNOWN_ERROR;
}
jclass clazz = env->FindClass("com/android/server/SystemServer");
if (clazz == NULL) {
    return UNKNOWN_ERROR;
}
jmethodID methodId = env->GetStaticMethodID(clazz, "init2", "()V");
if (methodId == NULL) {
    return UNKNOWN_ERROR;
}
env->CallStaticVoidMethod(clazz, methodId);

LOGI("System server: entering thread pool.\n");
ProcessState::self()->startThreadPool();
IPCThreadState::self()->joinThreadPool();
LOGI("System server: exiting thread pool.\n");

return NO_ERROR;
}

```

system\_server 进程创建了 SensorService 服务，并且调用线程添加到 binder 通信中。通过 JNI 调用了 com.android.server.SystemServer.cpp 类的 init2() 函数。init2() 函数代码如下所示。

代码清单4-26: init2()函数

```

public static final void init2() {
    Slog.i(TAG, "Entered the Android system server!");
    Thread thr = new ServerThread();
    thr.setName("android.server.ServerThread");
    thr.start();
}

```

init2() 启动了 ServerThread 线程。这个线程中的 run() 函数启动了 PowerManagerService (电源管理服务)、BatteryService (电池管理服务)、WindowManagerService (窗口管理服务)、ActivityManagerService (进程管理服务) 等各项系统启动和运行的重要服务。

代码清单4-27: run()函数

```

class ServerThread extends Thread {
    ...
    @Override

```

```

        public void run() {
            ...
            try {
                Slog.i(TAG, "Entropy Service");
                ServiceManager.addService("entropy", new EntropyService());
                Slog.i(TAG, "Power Manager");
                power = new PowerManagerService();
                ServiceManager.addService(Context.POWER_SERVICE, power);
                Slog.i(TAG, "Battery Service");
                battery = new BatteryService(context, lights);
                ServiceManager.addService("battery", battery);
                Slog.i(TAG, "Init Watchdog");
                Watchdog.getInstance().init(context, battery, power, alarm,
                    ActivityManagerService.self());
                Slog.i(TAG, "Window Manager");
                wm = WindowManagerService.main(context, power,
                    factoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL, !firstBoot);
                ServiceManager.addService(Context.WINDOW_SERVICE, wm);
                ActivityManagerService.self().setWindowManager(wm);
                ...
            }
            catch (RuntimeException e) {
                Slog.e("System", "*****");
                Slog.e("System", "***** Failure starting core service", e);
            }
            ...
            Looper.loop();
            Slog.d(TAG, "System ServerThread is exiting!");
        }
        ...
    }

```

---

由以上代码可知，Java 世界的核心 Service 都是由 init2() 函数启动完成的。

## 第 5 章

# Android 系统内核分析

本章主要讲解 Android 系统内核相关知识。通过对标准 Linux 内核的介绍，以及对现有 Android 系统内核的分析，让读者深入了解 Android 系统内核。

## 5.1 Linux 内核基础

### 5.1.1 概述

Android 使用标准的 Linux 内核，随着 Android 发布版本的升级，Linux 内核也在同步更新。Android 中内核的结构与标准的 Linux 内核基本相同，因功能需要还增加了一些驱动程序，这些驱动程序可以分为专用驱动和设备驱动两类。因此在学习 Android 内核之前，有必要了解 Linux 内核。

Linux 操作系统自上而下可以分为用户空间（User Space）和内核空间（Kernel Space），如图 5-1 所示。

用户空间也称为应用程序空间，是用户应用程序执行的地方。C Library（libc）用于应用程序和内核之间的相互转换，提供了连接内核的系统调用接口。

内核的主要用途是实现对设备硬件的编程控制和接口操作，调度对硬件资源的访问，为用户程序提供一个高级的执行环境和对硬件的虚拟接口，实现用户程序与硬件的交互。内核实现的是一个资源管理器功能。无论进程、内存还是硬件设备，内核负责管理并裁定多个竞争用户对资源的访问（既包括内核空间也包括用户空间）。内核提供两种结构模式，整体式的单内核模式和层次式的微内核模式。

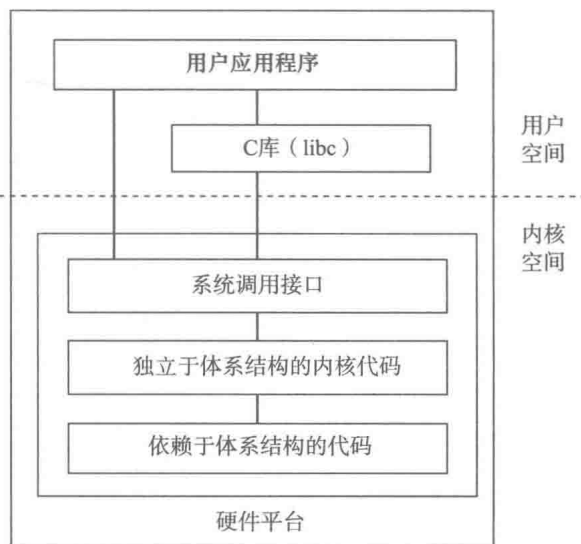


图 5-1 Linux 操作系统的基本结构

Android 所采用的 Linux 内核是基于单内核模式的。这种模式的优点在于代码紧凑，执行速度快。Linux 内核可以看作一个整体，也可以被划分为多个子系统。Linux 内核由上至下分别为：系统调用接口，实现系统的基本功能，如 read 和 write 功能；独立于体系结构的内核代码，这些代码是 Linux 所支持的所有处理器体系结构所通用的内核代码；依赖于体系结构的代码，这些代码用于特定体系结构的处理器和特定平台的代码，构成了通常称为 BSP（Board Support Package）的部分。

5.1.2 Linux 内核的主要子系统

Linux 内核可以划分为多个子系统，Linux 内核的主要组件如图 5-2 所示。



图 5-2 Linux 内核体系结构透视图

1. 系统调用接口 (System Call Interface)

SCI 层依赖于体系结构，定义了从用户空间到内核的函数调用接口，提供函数调用多路复用和多路分解服务。在 `./linux/kernel` 中可以找到 SCI 层的实现，在 `./linux/arch` 中可以找到依赖于体系结构的部分。

2. 进程管理 (Process Management)

进程管理的重点是进程的执行。内核通过 SCI 层提供了一个应用程序编程接口 (API) 用于新进程的创建和运行进程的停止操作，例如进程创建函数 `fork`、`exec` 和 `Portable Operating System Interface [POSIX]` 函数，进程终止函数 `kill`、`exit` 函数，并通过 `signal` 机制和 `POSIX` 机制实现进程之间的通信和同步。

进程管理还通过 `O(1)` 调度算法协调各活动进程之间共享 CPU 的需求。该算法支持多处理器（称为对称多处理器或 `SMP`），特点是调度多个进程所使用的时间和调度一个进程所使用的时间是相同的。因此不管有多少个进程竞争 CPU，该算法都可以在固定时间内完成操作。

3. 内存管理 (Memory Management)

Linux 内存管理机制中拥有物理内存和虚拟内存两种内存形式。物理内存就是系统硬件提供的内存大小，是真正的内存。虚拟内存是物理内存的扩展，以提升物理内存的利用率为目的，是利用磁盘空间虚拟出的一块逻辑内存。用作虚拟内存的磁盘空间被称为交换空间 (`Swap Space`)，工作原理是在物理内存不足时，根据内核中的“最近最经常使用”算法将暂时不用的页面文件信息写入交换空间，释放相关的物理内存，被释放的内存空间可以用于其他应用程序的申请。当系统需要调用这些页面文件时，这些信息重新从交换空间写入物理内存空间。内存管理的源代码可以在 `./linux/mm` 中找到。

4. 虚拟文件系统 (Virtual File System)

虚拟文件系统为文件系统提供了一个通用的抽象接口。VFS 在 SCI 和内核所支持的文件



系统之间提供了一个交换层如图 5-3 所示。

在 VFS 上面是对 API 的抽象，为 open、close、read 和 write 之类的函数提供一个通用的 API。在 VFS 下面是对文件系统的抽象，定义了上层函数的实现方式。它们是给定文件系统（超过 50 个）的插件。文件系统的源代码在 `./linux/fs` 中。

文件系统层之下是缓冲区缓存，它为文件系统层提供了一个通用函数集（与具体文件系统无关）。这个缓存层通过将数据保留一段时间（或者随机预先读取数据以便在需要时就可用）优化了对物理设备的访问。缓冲区缓存之下是设备驱动程序，实现特定物理设备的接口。

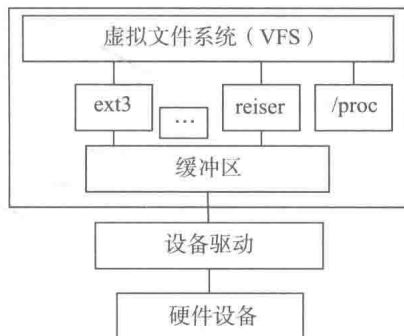


图 5-3 VFS 在用户和文件系统之间提供了一个交换层

## 5. 网络堆栈 (Network Stack)

网络堆栈在设计上遵循分层体系结构，是网络子系统的标准 API，它为各种网络协议提供了一个用户接口。从原始帧访问到 IP 协议数据单元 (PDU)，再到 TCP 和 UDP，socket 层提供了一种标准化的方法来管理连接，并在各个终点之间移动数据。网络堆栈的源代码在 `./linux/net` 中。

## 6. 设备驱动程序 (Device Drivers)

设备驱动程序是内核的重要组成部分，实现特定硬件设备的运转。Linux 源码树提供了一个驱动程序子目录，这个目录又进一步划分为各种支持设备，例如 Bluetooth、I2C、serial 等。设备驱动程序的代码可以在 `./linux/drivers` 中找到。

## 7. 依赖体系结构的代码 (Arch)

为实现更高效的操作，Linux 需要发挥不同体系结构的优点，在 `./linux/arch` 子目录中定义了内核源代码中依赖于体系结构的部分，包含了各种特定体系结构的子目录，例如典型桌面系统所使用的 i386 目录。每个子目录都包含了很多关注内核中某一个特定方面的目录，例如引导、内核、内存管理等。

### 5.1.3 Linux 启动过程分析

Linux 的开机启动过程可以分为 4 个步骤：启动驱动器、启动内核、初始化过程、启动用户登录界面。每个步骤中所执行的操作如图 5-4 所示。

1) 用户开机启动 PC 电源，CPU 开始运行，从地址编号 `0xFFFF0` 开始自动执行程序代码。这个地址通常是 ROM BIOS 中的地址。BIOS 首先利用 POST (PowerOnSelfTest, 上电自检) 程序来对内部各个设备进行检查，并按照 BIOS 中设置的启动设备进行启动。Linux 系统将 Grub 分两段进行引导，第一段存储于硬盘 MBR 中，第二段放置于操作系统内核所在

的分区上。Grub 根据 MBR 中第一段找到存放有 Grub 菜单等信息的第二段，然后继续引导，使得 Linux 获得启动权。

2) Linux 进行内核的引导。主要完成磁盘引导、读取机器系统数据、实模式和保护模式的切换、加载数据段寄存器以及重置中断描述符表等操作。

3) init 内核启动。加载内核之后，内核执行的 1 号进程就是 init 进程。init 进程首先读取配置文件 inittab，完成一系列初始化任务，确定用户的登录模式以及运行级别。由系统服务初始化脚本 sysinit 将 Linux 主机信息读入 Linux 系统，包括默认路径、主机名称、网络信息等内容。同时根据运行级别执行相应级别下的 init.d 脚本，具体目录为 /etc/rc.d/rcN.d，其中 N 代表不同的级别。最后执行 rc.local 文件，返回 init 程序。

4) 启动用户登录界面。init 初始化结束后，启动 mingetty，打开终端用户登录系统。用户登录成功后运行 Shell 脚本。至此 Linux 启动完成。

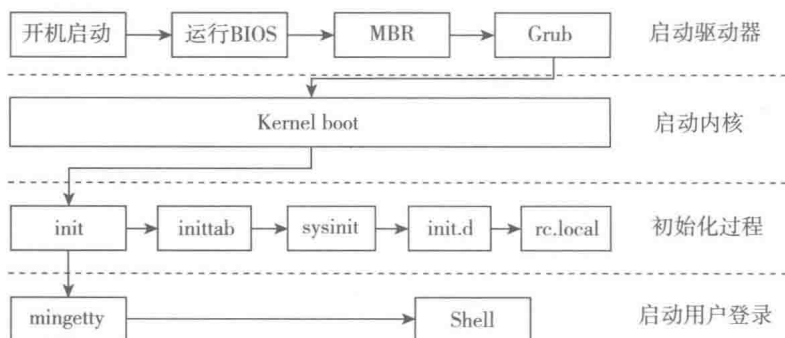


图 5-4 Linux 启动过程

## 5.2 Android 内核概况

Android 基于 Linux 操作系统，由硬件、系统内核、系统服务和应用程序 4 部分组成。其中内核是最核心的部分，与普通应用程序不同，内核拥有所有硬件设备的访问权限以及启动时划分受保护的内存空间，主要实现内存管理、进程调度、进程间通信等功能。

Android 内核是在标准的 Linux 内核的基础上修改而成的。为了适应嵌入式硬件环境和移动应用程序的开发，Android 对标准 Linux 内核进行了一定的修改。

### 1. 文件系统

由于移动设备大多采用 Flash 作为存储介质，因此 Android 内核中增加了专门用于 Flash 的文件系统——YAFFS2 文件系统，对 NAND Flash 芯片有良好的支持。YAFFS2 按层次结构设计，分为文件管理接口、内部实现层和 NAND。简化的系统接口设计使得该文件系统集成更为方便。经过测试证明，YAFFS2 性能比支持 NOR 型闪存的 JFFS2 文件系统更优秀。

## 2. 进程间通信机制

Android 提供进程间的通信机制 IPC Binder，在内核代码中 Binder 通过守护进程 Service Manager 管理系统中的服务，负责进程间的数据交换。各进程通过 Binder 访问同一块共享内存。从应用层的角度看，进程通过访问数据的守护进程获取程序框架接口，实现数据访问、数据交换、数据共享等操作。

## 3. 内存管理

Android 内核采用一种叫作 LMK (Low Memory Killer) 的低内存管理策略。这种管理机制将进程按照重要性进行分级、分组。内存不足时，将处于最低级别的进程关闭，而使处于较高级别的进程保持运行。例如，在移动设备中，UI 界面处于最高级别，所以该进程永远不会被中止，这种方式使得终端用户认为系统是稳定运行的。同时，Android 增加了一种内存共享的处理方式 Ashmem (Anonymous Shared Memory，匿名共享内存)。通过 Ashmem，进程间可以匿名自由共享同名的内存块。

## 4. 电源管理

由于 Android 主要用于移动设备，电源管理就显得尤为重要。目前采用的是一种较为简单的电源管理策略，通过开关屏幕、开关屏幕背光、开关键盘背光、开关按钮背光和调整屏幕亮度来实现电源管理，并没有实现休眠和待机功能。

有 3 种途径判断、调整电源管理策略：RPC 调用、电池状态改变和电源设置。通过广播 Intent 或直接调用 API 的方式来与其他模块进行联系。电源管理策略同时还有自动关机机制，当电力低于最低可接受程度时，系统将自动关机。Android 的电源管理模块还会根据用户行为自动调整屏幕亮度。

## 5. 其他

Android 内核添加了字符输出设备、图像显示设备、键盘输入设备、RTC 设备、USB 设备等相关设备驱动，增加了 Logger 系统。

# 5.3 Android 启动过程分析

Android 从 Linux 内核启动可分为 4 个步骤，启动框架如图 5-5 所示。

### 1. init 进程 (system/core/init)

init 进程是第一个进程，在内核完成初始化操作后，自行启动的进程。init 进程根据 init.rc 和 init.xxx.rc 脚本文件建立 servicemanager、zygote 等基本的服务，创建结束后承担 property service 的功能。

### 2. Zygote 框架建立

servicemanager 和 zygote 进程是 Android 的基础。启动代码位于 frameworks\base\cmds\

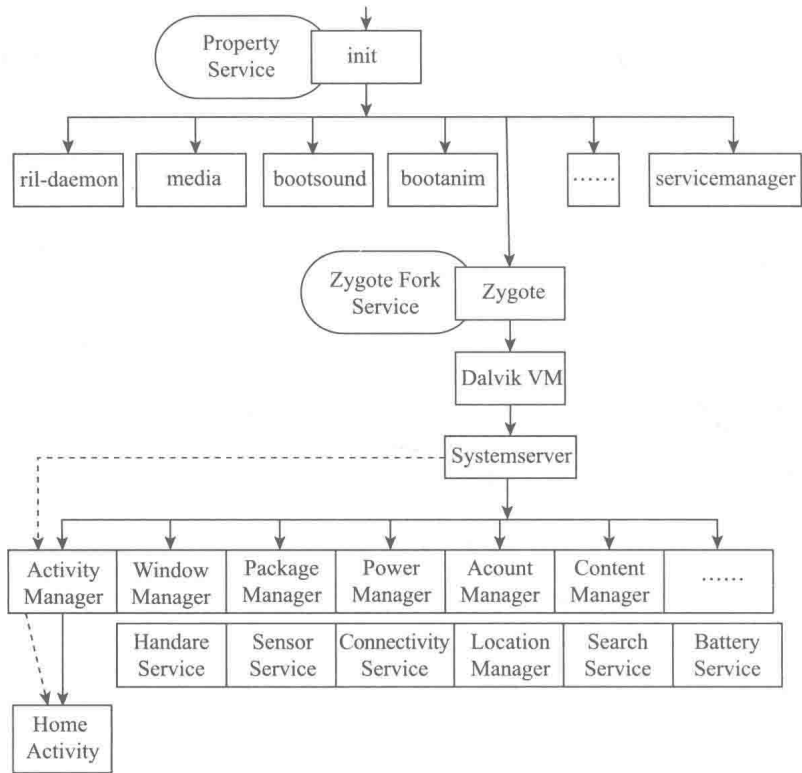


图 5-5 Android 启动框架图

app\_process\app\_main.cpp 文件中，由 main() 函数开始执行，代码如下所示：

代码清单5-1：Zygote框架建立main()函数

```
int main(int argc, const char* const argv[])
{
    mArgC = argc;
    mArgV = argv;
    .....
    runtime.mParentDir = parentDir;

    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit",
            startSystemServer ? "start-system-server" : "");
    } else if (className) {
        runtime.mClassName = className;
        runtime.mArgC = argc - i;
        runtime.mArgV = argv + i;
        runtime.start("com.android.internal.os.RuntimeInit",
            application ? "application" : "tool");
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
    }
}
```

```

    app_usage();
    LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
    return 10;
}
}

```

其中 `runtime.start()` 函数启动 Java 虚拟机。`runtime` 是 `AppRuntime` 的对象，定义在 `app_main.cpp` 中，这里的 `runtime.start()` 函数实际上调用的是 `AndroidRuntime::start()`。`AndroidRuntime::start()` 函数定义在 `/frameworks/base/core/jni/AndroidRuntime.cpp` 文件中，其中 `start()` 函数的作用是启动虚拟机，通过 `startVm()` 函数调用 JNI 创建 JavaVM，调用 `startReg()` 函数注册 JNI 接口，最后调用 `com.android.internal.os.ZygoteInit` 中的 `main()` 函数，注册用来接受请求的 Listen 端口，并启动 `SystemService`。至此，`Zygote` 建立完成。

`Zygote` 建立后，就可以利用 socket 的 Listen 端口进行通信，接收 `ActivityManagerService` 的请求，fork 应用程序。

### 3. System Server

`startSystemService` 是在 `Zygote` 上 fork 了以下进程：

```
com.android.server.SystemServer
```

Android 的所有服务循环框架都是建立在 `SystemService` 上。在 `SystemService.java` 中调用 `init2()` 函数，由 `init2()` 建立 Android 中所有用到的服务循环框架。

### 4. Home 启动

Home 是在 `ActivityManagerService.systemReady()` 通知的过程中建立的。系统在所有的 Android 服务启动完成后，会使用 `xxx.systemReady()` 函数通知各个 Service 系统已经就绪。

## 5.4 Binder 框架分析

Linux 系统是以进程为单位分配和管理资源的。出于保护机制，进程之间不能够进行资源的直接访问，当进程之间需要互通信息共享资源的时候，操作系统提供进程间的通信机制 IPC。Linux 中提供多种通信机制，如 `named pipe`、`message queue`、`signal`、`share memory`、`socket` 等。Android 继承了 Linux 对进程的保护机制，为进程间的通信提供更加简洁、快速、内存消耗小的 Binder 机制。

相比较其他的通信方式，Binder 主要提供以下功能：

- 利用驱动程序来推进进程间的通信。
- 通过共享内存来提高性能。
- 为进程请求分配每个进程的线程池。
- 引入了引用计数和跨进程的对象引用映射。

- 实现进程间同步调用。

同时 Binder 机制有效地避免了进程过载和安全漏洞等方面的风险。

### 5.4.1 概述

Binder 机制基于 Linux 的 OpenBinder 实现，由 Binder Driver 来实现，采用同步通信方式。通信是基于 Server 与 Client 的，所有需要 IBinder 通信的进程都必须创建一个 IBinder 接口。作为守护进程的 Service Manager 管理者，Android 系统服务端（Server）提供了各种服务（Service），当收到其他程序发来的请求时，Service Manager 进行响应。每个服务（Service）都要在 Service Manager 中注册，而请求服务的客户端（Client）要使用某个 Service，必须向 Service Manager 请求服务、进行查询、获取相关的信息；Client 根据得到的 Service 信息与 Service 所在的 Server 进程建立通信的通路，进行交互通信。工作过程可分为注册服务、查询服务、使用服务 3 个步骤，如图 5-6 所示。



图 5-6 工作过程

在 Android 虚拟机启动之前，系统先启动 Service Manager 进程，Service Manager 会打开 Binder 驱动，并通知 BinderKernel 驱动程序，这个进程将作为 System Service Manager，然后该进程将进入一个循环，等待处理来自其他进程的数据。

Binder 的实现大致分为如下几个部分：

#### （1）Binder 驱动

Binder 驱动是通信的主要完成模块。Binder 机制中，Server 和 Client 通过 Binder 驱动交互数据，执行数据的写入、读取等操作。

#### （2）Service Manager

守护进程的 Service Manager 是 Android 系统中所有服务的管理器。每个 Server 都需要在 Service Manager 中进行注册，Client 在 Service Manager 中查询服务并获取服务的信息。实现代码位于 `frameworks\base\cmds\servicemanager\service_manager.c`。

#### （3）Server

Server 即服务端，也是 Android 的系统服务，本质是响应 Client 的请求，负责在 Service Manager 中注册服务。同时，Server 还包括监听请求、处理请求、应答 Client 的过程。在 Android 中服务包括由 C++ 空间完成的 Native 服务，这些服务是由系统初始化时通过 `init.rc` 启动的；存在于 Android 空间，由 JVM 空间完成的 Android 服务，这些服务是由 `init2` 启动的，用于属性设置的 Init 空间服务。

#### （4）Client

Client 即客户端，一般是指 Android 系统的应用程序，是服务的请求方。

### (5) Proxy

Proxy 即服务代理对象，是在 Client 建立一个 Server 的“引用”，使得该代理对象具有 Server 的功能。从应用程序的角度看，代理对象和本地对象一样，都可以调用其方法，返回相应的结果。

## 5.4.2 Binder 的系统架构

### 1. Binder 的系统架构

Android 系统中，每个 Activity 和 Service 都是进程，两者之间的通信看起来就好像是一个进程进入另一个进程中执行代码，然后带着执行的结果返回。这种跨进程的通信方式依赖于 Binder 的用户空间为每个进程维护一个可用的线程池，IPC 以及进程的本地消息等交由线程池处理和执行。

在 Android 源码中，主要的 Binder 库由本地原生代码实现。应用程序通过 Binder 接口调用 Binder 原生库。Binder 的系统架构如图 5-7 所示。

Binder 驱动用于实现 Binder 的设备驱动，主要负责组织 Binder 的服务节点，调用 Binder 相关的处理线程，完成实际的 Binder 传输等，它位于 Binder 结构的最底层（即 Linux 内核层）。Binder Adapter 层是对 Binder 驱动的封装，实现 Binder 驱动的操作，实现包括 `IPCThreadState.cpp` 和 `ProcessState.cpp`，以及 `Parcel.cpp` 中的部分

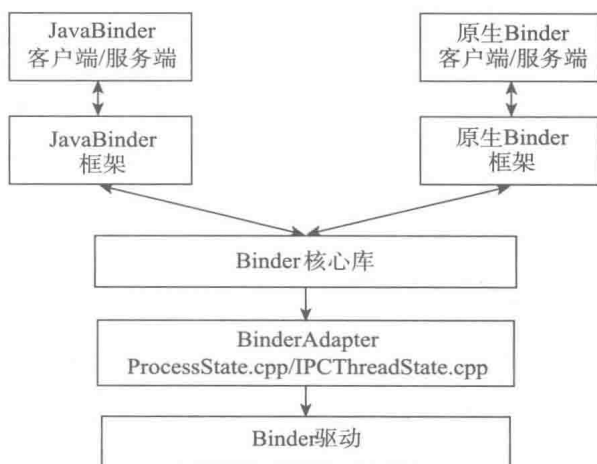


图 5-7 Android Binder 系统架构

内容。Binder 核心库是 Binder 构架的核心实现，主要包括 `IBinder`、`Binder`（服务器端）和 `BPBinder`（客户端）。最上面两层的 Binder 构架和具体的客户端/服务端都有 Java 和 C++ 两种实现方案，主要供应用程序使用，都是通过调用 Binder 的核心库来实现的。

### 2. Binder 驱动

Binder 驱动的实现遵循 Linux 设备驱动模型，通过 `binder_ioctl()` 函数与用户空间的进程交换数据。使用 `BINDER_WRITE_READ` 进行数据的读写操作，`binder_thread_write()` 函数发送请求或返回结果，`binder_thread_read()` 函数用于读取结果。实现代码主要涉及以下文件：`kernel/drivers/staging/binder.h` 和 `kernel/drivers/staging/binder.c`。

#### (1) binder\_init

`binder_init()` 是 Binder 驱动的初始化函数，位于 `kernel/drivers/staging/binder.c` 中，代码如下所示：

代码清单5-2: binder\_init()函数

---

```
static int_init binder_init(void)
{
    int ret;
    binder_proc_dir_entry_root = proc_mkdir("binder", NULL);
    if (binder_proc_dir_entry_root)
        binder_proc_dir_entry_proc = proc_mkdir("proc", binder_proc_dir_entry_root);
    ret = misc_register (&binder_miscdev);
    if (binder_proc_dir_entry_root) {
        create_proc_read_entry("state", S_IRUGO, binder_proc_dir_entry_root,
            binder_read_proc_state, NULL);
        create_proc_read_entry("stats", S_IRUGO, binder_proc_dir_entry_root,
            binder_read_proc_stats, NULL);
        create_proc_read_entry("transactions", S_IRUGO, binder_proc_dir_entry_root,
            binder_read_proc_transactions, NULL);
        create_proc_read_entry("transaction_log", S_IRUGO, binder_proc_dir_entry_root,
            binder_read_proc_transaction_log, &binder_transaction_log);
        create_proc_read_entry ("failed_transaction_log", S_IRUGO,
            binder_proc_dir_entry_root, binder_read_proc_transaction_log,
            &binder_transaction_log_failed) ;
    }
    return ret;
}
device_initcall(binder_init);
```

---

本段代码中, binder\_init() 函数由设备驱动接口函数 device\_initcall() 函数来调用。该函数使用 proc\_mkdir() 函数创建了 proc 文件系统的根节点和其他节点, 使用 misc\_register () 函数将自己注册为 misc 设备, 使用 create\_proc\_read\_entry() 函数创建各个只读 proc 文件, 并指定了这些文件的操作函数及所需的参数。

Binder 驱动除了提供只读文件的操作接口, 还提供了设备的操作接口。这些操作在结构体 file\_operations 中进行了定义, 代码如下所示:

代码清单5-3: Binder驱动操作定义

---

```
static struct file_operations binder_fops = {
    .owner = THIS_MODULE,
    .poll = binder_poll,
    .unlocked_ioctl = binder_ioctl,
    .mmap = binder_mmap,
    .open = binder_open,
    .flush = binder_flush,
    .release = binder_release,
};
```

---

该操作中定义了实现非阻塞 I/O 模型函数 binder\_poll、底层驱动调用操作 binder\_ioctl、



虚拟内存空间申请操作 `binder_mmap`、设备文件打开操作 `binder_open`、`binder_flush`、设备文件关闭操作 `binder_release`。

## (2) binder\_poll

所有支持非阻塞 I/O 操作的设备驱动都需要实现 `poll()` 函数。`binder_poll()` 函数提供 `proc_work` 和 `thread_work` 两种等待任务实现方式，实现代码如下所示：

代码清单5-4: `binder_poll()`函数

---

```
static unsigned int binder_poll(struct file *filp, struct poll_table_struct *wait)
{
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread = NULL;
    int wait_for_proc_work;

    mutex_lock(&binder_lock);
    thread = binder_get_thread(proc);

    wait_for_proc_work = thread->transaction_stack == NULL &&
        list_empty(&thread->todo) && thread->return_error == BR_OK;
    mutex_unlock(&binder_lock);

    if (wait_for_proc_work) {
        if (binder_has_proc_work(proc, thread))
            return POLLIN;
        poll_wait(filp, &proc->wait, wait);
        if (binder_has_proc_work(proc, thread))
            return POLLIN;
    } else {
        if (binder_has_thread_work(thread))
            return POLLIN;
        poll_wait(filp, &thread->wait, wait);
        if (binder_has_thread_work(thread))
            return POLLIN;
    }
    return 0;
}
```

---

本段代码中，通过 `binder_get_thread()` 函数获得当前进程信息。`if{}else{}` 语句根据进程的信息选择调用 `proc_work` 方式和 `thread_work` 方式。`poll` 操作则通过调用 `poll_wait()` 函数实现。

## (3) binder\_ioctl

用户空间通过 `ioctl()` 函数调用相应底层驱动的命令，该函数是 `binder` 功能的具体实现。`binder.h` 文件中为 `ioctl` 定义了 7 个命令，分别如下：

```
#define BINDER_WRITE_READ_IOWR('b', 1, struct binder_write_read)
```

```

#define BINDER_SET_IDLE_TIMEOUT_IOW('b', 3, int64_t)
#define BINDER_SET_MAX_THREADS_IOW('b', 5, size_t)
#define BINDER_SET_IDLE_PRIORITY_IOW('b', 6, int)
#define BINDER_SET_CONTEXT_MGR_IOW('b', 7, int)
#define BINDER_THREAD_EXIT_IOW('b', 8, int)
#define BINDER_VERSION_IOWR('b', 9, struct binder_version)

```

这 7 个命令中，仅有 5 个被实现。在 binder.c 文件中，ioctl() 函数实现代码如下所示：

代码清单5-5: binder\_ioctl()函数

---

```

static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int ret;
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread;
    unsigned int size = _IOC_SIZE(cmd);
    void __user *ubuf = (void __user *)arg;
    .....
    switch (cmd) {
    case BINDER_WRITE_READ: .....
    case BINDER_SET_MAX_THREADS: .....
    case BINDER_SET_CONTEXT_MGR: .....
    case BINDER_THREAD_EXIT: .....
    case BINDER_VERSION: .....
    }
}

```

---

ioctl() 函数首先根据接收到的参数 \*filp 查找需要操作的文件，将 cmd 对应为后续执行命令的代号，arg 为补充参数。通过 switch() 函数调用相应的命令操作。

BINDER\_WRITE\_READ 执行的是读写操作，其实现代码如下所示：

代码清单5-6: BINDER\_WRITE\_READ执行写操作

---

```

struct binder_write_read bwr;
if (size != sizeof(struct binder_write_read)) {
    ret = -EINVAL;
    goto err;
}
if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
    ret = -EFAULT;
    goto err;
}
if (binder_debug_mask & BINDER_DEBUG_READ_WRITE)
    printk(KERN_INFO "binder: %d:%d write %ld at %08lx, read %ld
        at %08lx\n",
        proc->pid, thread->pid, bwr.write_size, bwr.write_buffer,
        bwr.read_size, bwr.read_buffer);

```

---

```

if (bwr.write_size > 0) {
    ret = binder_thread_write(proc, thread, (void __user *)bwr.
        write_buffer, bwr.write_size, &bwr.write_consumed);
    if (ret < 0) {
        bwr.read_consumed = 0;
        if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
            ret = -EFAULT;
        goto err;
    }
}
if (bwr.read_size > 0) {
    ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer,
        bwr.read_size, &bwr.read_consumed, filp->f_flags & O_NONBLOCK);
    if (!list_empty(&proc->todo))
        wake_up_interruptible(&proc->wait);
    if (ret < 0) {
        if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
            ret = -EFAULT;
        goto err;
    }
}
if (binder_debug_mask & BINDER_DEBUG_READ_WRITE)
    printk(KERN_INFO "binder: %d:%d wrote %ld of %ld, read
        return %ld of %ld\n",
        proc->pid, thread->pid, bwr.write_consumed, bwr.write_size,
        bwr.read_consumed, bwr.read_size);
if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
    ret = -EFAULT;
    goto err;
}
break;
}
}

```

本段代码中通过 if 语句判断数据的完整性，利用 copy\_from\_user() 函数将数据从用户空间复制到 binder\_write\_read 的结构体中。接下来判断这个结构体中的 write\_size 和 read\_size 是否大于 0，如果 write\_size 大于 0 则调用 binder\_thread\_write() 函数执行写操作；如果 read\_size 大于 0 则调用 binder\_thread\_read() 函数执行读操作。最后利用 copy\_to\_user() 函数将操作完成的数据复制到用户空间。

BINDER\_SET\_MAX\_THREADS 命令用来告知 Binder 驱动接收方（通常是 Server 端）线程池中最大的线程数。命令实现代码如下所示：

#### 代码清单5-7: BINDER\_SET\_MAX\_THREADS命令实现

```

if (copy_from_user(&proc->max_threads, ubuf, sizeof(proc->max_threads))) {
    ret = -EINVAL;
}

```

```

        goto err;
    }

```

BINDER\_SET\_CONTEXT\_MGR 命令通常在初始化 Binder 驱动的过程中被调用，执行的操作是将一个进程（或线程）设置为 Context Manager，即 context\_mgr。命令的实现代码如下所示：

代码清单5-8: BINDER\_SET\_CONTEXT\_MGR命令实现

```

if (binder_context_mgr_node != NULL) {
    printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
    ret = -EBUSY;
    goto err;
}

if (binder_context_mgr_uid != -1) {
    if (binder_context_mgr_uid != current->cred->euid) {
        printk(KERN_ERR "binder: BINDER_SET_"
            "CONTEXT_MGR bad uid %d != %d\n",
            current->cred->euid,
            binder_context_mgr_uid);
        ret = -EPERM;
        goto err;
    }
} else
    binder_context_mgr_uid = current->cred->euid;
binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
if (binder_context_mgr_node == NULL) {
    ret = -ENOMEM;
    goto err;
}
binder_context_mgr_node->local_weak_refs++;
binder_context_mgr_node->local_strong_refs++;
binder_context_mgr_node->has_strong_ref = 1;
binder_context_mgr_node->has_weak_ref = 1;
break;

```

本段代码可分为 3 部分：由于系统中仅允许一个 Context Manager 出现，因此应当先判断 binder\_context\_mgr\_node 的值是否为空，如果为空说明当前没有 Context Manager 的节点，可以进行创建操作。其次判断 binder\_context\_mgr\_uid 是否存在，如果不存在，则将驱动中的全局变量 binder\_context\_mgr\_uid 设置为当前进程的 uid；如果存在，检查当前进程是否有执行命令的权限。最后创建并初始化一个 binder\_node 并赋值给全局变量 binder\_context\_mgr\_node。

BINDER\_THREAD\_EXIT 命令用来告知 Binder 驱动当前线程退出并释放相应的数据结构。命令的实现代码如下所示：

代码清单5-9: BINDER\_THREAD\_EXIT命令实现

---

```

if (binder_debug_mask & BINDER_DEBUG_THREADS)
    printk(KERN_INFO "binder: %d:%d exit\n",
           proc->pid, thread->pid);
binder_free_thread(proc, thread);
thread = NULL;
break;

```

---

BINDER\_VERSION 命令用来获取当前 Binder 驱动的版本号。实现代码如下所示:

代码清单5-10: BINDER\_VERSION命令实现

---

```

if (size != sizeof(struct binder_version)) {
    ret = -EINVAL;
    goto err;
}
if (put_user(BINDER_CURRENT_PROTOCOL_VERSION, &((struct binder_version *)
ubuf)->protocol_version)) {
    ret = -EINVAL;
    goto err;
}
break;

```

---

其中 put\_user() 函数是将当前 Binder 的版本返回给调用进程。

#### (4) binder\_mmap

mmap() 函数执行的是内存映射操作, 该操作在 5.5.4 节中将会详细介绍。Binder 中每个进程(或线程)执行一次映射操作, 所有的设备内存是在执行 mmap 操作时被分配的。分配时需要先申请内核虚拟映射表上的空间, 然后分配物理存储空间, 最后将分配的物理空间映射到虚拟映射表中。

#### (5) binder\_open

binder\_open() 函数用于 Binder 设备文件的打开操作, 实现代码如下所示:

代码清单5-11: binder\_open()函数

---

```

static int binder_open(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc;

    if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
        printk(KERN_INFO "binder_open: %d:%d\n", current->group_leader->pid,
               current->pid);

    proc = kzalloc(sizeof(*proc), GFP_KERNEL);
    if (proc == NULL)
        return -ENOMEM;
    get_task_struct(current);

```

---

```

proc->tsk = current;
INIT_LIST_HEAD(&proc->todo);
init_waitqueue_head(&proc->wait);
proc->default_priority = task_nice(current);
mutex_lock(&binder_lock);
binder_stats.obj_created[BINDER_STAT_PROC]++;
hlist_add_head(&proc->proc_node, &binder_procs);
proc->pid = current->group_leader->pid;
INIT_LIST_HEAD(&proc->delivered_death);
filp->private_data = proc;
mutex_unlock(&binder_lock);

if (binder_proc_dir_entry_proc) {
    char strbuf[11];
    snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
    remove_proc_entry(strbuf, binder_proc_dir_entry_proc);
    create_proc_read_entry(strbuf, S_IRUGO, binder_proc_dir_entry_proc,
        binder_read_proc_proc, proc);
}

return 0;
}

```

本段代码中 `kzalloc()` 函数为 `binder_proc` 分配空间，保存 Binder 数据；`get_task_struct()` 函数用于增加引用计数，并通过 `proc->tsk = current` 语句保存引用计数；`INIT_LIST_HEAD()` 函数完成 `binder_proc` 队列链表头的初始化操作；`init_waitqueue_head()` 完成等待队列 `wait` 的初始化，`proc->default_priority` 设置当前进程的 `nice` 值；`binder_stats.obj_created[]++` 完成对其参数对象的计数增加；`hlist_add_head()` 添加参数 `binder_procs` 到哈希表中；赋值语句 `proc->pid = current->group_leader->pid` 完成将当前进程或线程组的 `pid` 赋值给 `proc->pid`，并作为进程的 `id` 进行保存；最后通过 `if` 语句创建只读文件 `/proc/binder/proc/$pid`，`snprintf()` 函数输出当前对象的状态；`create_proc_read_entry()` 函数指定文件的函数接口。

#### (6) binder\_flush

`binder_flush()` 函数在关闭设备文件描述符复制时被调用。实现代码如下所示：

代码清单5-12: `binder_flush()`函数

```

static int binder_flush(struct file *filp, fl_owner_t id)
{
    struct binder_proc *proc = filp->private_data;
    binder_defer_work(proc, BINDER_DEFERRED_FLUSH);
    return 0;
}

```

本段代码主要通过 `binder_defer_work()` 函数调用 `BINDER_DEFERRED_FLUSH` 操作完成。

### (7) binder\_release

binder\_release() 函数在 Binder 驱动退出时释放所占用的空间并清除相关的数据, 实现代码如下所示:

代码清单5-13: binder\_release()函数

---

```
static int binder_release(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc = filp->private_data;
    if (binder_proc_dir_entry_proc) {
        char strbuf[11];
        snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
        remove_proc_entry(strbuf, binder_proc_dir_entry_proc);
    }
    binder_defer_work(proc, BINDER_DEFERRED_RELEASE);
    return 0;
}
```

---

本段代码中赋值语句 `binder_proc *proc = filp->private_data` 用于获取当前进程、线程的 `private_data` 数据, `snprintf()` 函数是找到并输出以 `pid` 命名的只读文件, `remove_proc_entry()` 函数对找到的文件进行删除, `binder_defer_work()` 函数释放 `binder_proc` 对象所占用的空间。

### 3. Binder Adapter

Binder Adapter 层的作用是完成对 Binder 驱动的封装, 实现 `IPCThreadState.cpp` 和 `ProcessState.cpp`, 以及 `Parcel.cpp` 中的部分内容。

`ProcessState` 属于 `singleton` 类型, 其作用是维护当前进程中的所有 `Proxy`。一个 `Client` 通常需要多个 `Service` 的服务, 这样就需要创建多个 `Service` 代理——`Proxy`, 客户端进程中的 `Process State` 对象负责维护 `Proxy`。

`IPCThreadState` 对象主要负责调用 Binder 设备的相关操作, 完成数据读取、写入和请求处理框架。主要实现 `talkWithDriver()` 读取 / 写入函数, `executeCommand()` 请求处理函数, `joinThreadPool()` 设备轮询函数。`IPCThreadState` 具体实现了所有进程 (包括客户端和服务端的进程) 与 Binder 设备的通信操作。

每个进程只有一个 `Process State` 对象, 每一个线程中都会有一个 `IPCThreadState` 对象, 其源代码位于 `frameworks\base\include\binder` 和 `frameworks\base\libs\binder` 两个文件夹中。

## 5.4.3 Binder 的机制和原理

### 1. Binder 的工作流程

Binder 的工作流程如下:

- 1) 创建 `Proxy`, `Client` 首先获取相应的 `Proxy`。
- 2) `Client` 通过调用 `Proxy` 发送请求。

- 3) Proxy 将请求通过 Binder 驱动发送给 Server 进程。
  - 4) Server 进程处理请求, 通过 Binder 驱动返回处理结果给 Proxy。
  - 5) Client 通过 Proxy 收到 Server 的返回结果。
- 至此, Binder 就完成了一次通信。

## 2. Binder 的机制

Binder 机制实际上就是一个类似于 C/S 的构架: Client 进程通过 Proxy 与 Server 进程通信, Proxy 通过 Binder 驱动将 Client 进程的请求发送给 Server 进程, 并返回结果。对于 Server 进程需要继承 BBinder (BnInterface), 因为 BBinder 对象有消息处理函数 onTransact(); Client 进程需要继承 BpBinder (BpInterface), 因为 BpBinder 对象有消息传递函数 transact()。

### (1) IBinder 接口

一个普通对象通常只能当前进程中访问, 若要被其他进程访问, 就属于跨进程通信, 需要实现 IBinder 接口。

IBinder 接口是跨进程对象的抽象, 在 framework\base\include\binder\IBinder.h 文件中定义了使用 Binder 机制来实现客户程序与服务端的通信协议。IBinder 接口可以指向本地对象, 也可以指向远程对象。如果 IBinder 指向 Proxy, transact() 函数负责把 Client 请求发送给 Server; 如果 IBinder 指向 Server, transact() 函数负责提供服务。

### (2) 服务端代理对象 BpBinder

IBinder 类中派生出的 BpBinder 和 BBinder 用以实现 Binder 通信, BpBinder 只能与相对应的 BBinder 进行交互通信。BpBinder 是 Server 用来实现交互的代理类, p 即 Proxy 的意思, BBinder 是 Server 代理。两者一一对应。

BpBinder 是 Server 的代理对象, 即远程对象在当前进程的代理, 是 IBinder 接口在 Client 进程的实现, 其 transact() 函数的实现代码如下所示:

代码清单5-14: Client中实现transact()函数

---

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
    return DEAD_OBJECT;
}
```

---

函数的参数分别为请求的 ID 号、请求的参数、返回的结果、额外的标识 (通常为 0)。在 BpBinder 中的 transact 函数中, 只是调用了 IPCThreadState::self()->transact 方法, 也就



是说，数据处理是在 `IPCThreadState` 类中的 `transact`。在 `transact` 中，它把请求的数据经过 `Binder` 设备发送给了 `Service`。`Service` 处理完请求后，又将结果原路返回给 `Client`。

### (3) 服务端 `BBinder`

`IBinder` 接口在 `Server` 的实现，其中 `transact()` 函数的实现代码如下所示：

代码清单5-15: `Server`中实现`transact()`函数

---

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);

    status_t err = NO_ERROR;
    switch (code) {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags);
            break;
    }

    if (reply != NULL) {
        reply->setDataPosition(0);
    }

    return err;
}
```

---

`PING_TRANSACTION` 请求用来检查对象是否还存在，这里简单地把 `pingBinder` 的返回值返回给调用者，其他的请求交给 `onTransact()` 函数处理。`onTransact()` 函数是 `BBinder` 里声明的一个 `protected` 类型的虚函数，代码如下所示：

代码清单5-16: `onTransact()`函数

---

```
Protected Boolean onTransact(int code, Parcel data, Parcel reply,
    Int flags) throws RemoteException{
    Log.i("TAG", "----->" + data.readInt());
    Log.i("TAG", "----->" + data.readString());
    reply.writeInt(2);
    reply.writeString("Rose");
    return super.onTransact(code, data, reply, flags);
}
```

---

重载 `onTransact()` 函数的主要内容是把 `onTransact()` 函数的参数转换为服务函数的参数。

## 5.5 Ashmem 内存管理方式

### 5.5.1 概述

内存资源是设备运行过程中的一种“稀缺资源”，将有限的内存资源进行更加充分的分配和使用是内存管理的首要任务。Ashmem 是 Android 在 Linux 内核基础上添加的匿名共享内存机制，主要工作是实现内存资源的分配、释放、回收等机制，目的是为每一个用户程序分配相应的内存空间。

Ashmem 通过应用程序框架层提供的 MemoryFile 接口来实现。MemoryFile 接口是 JNI 方法调用系统运行时库层中的匿名共享内存 C 接口，通过这些 C 接口来使用内核空间中的匿名共享内存驱动模块。

Ashmem 的工作原理是借助内核驱动提供的内存回收算法机制（pin/unpin），对应用进程中空闲的内存空间执行 unpin 操作。内核可以将这块内存所对应的物理空间回收并进行二次分配。但是 unpin 操作并不会改变已经 mmap 的地址空间，因此应用进程依然可以对已经 unpin 的内存空间再次进行访问，并且可以通过缺页 handler 再次获得这部分空间。

Android 依然保留了 Linux 内核的共享内存机制，因此需要向 Linux 内存管理系统的内存回收算法注册接口，由内存管理系统执行回收内存空间的操作。内存空间的释放由用户进程向 Ashmem 驱动提出，Ashmem 驱动通过注册的接口通知内存管理系统，最后由内存管理系统进行物理空间的回收。通过这种“用户—Ashmem 驱动程序—内存管理系统”三者的紧密合作，实现了有效的内存管理机制，适合移动设备内存小的特点。

### 5.5.2 Ashmem 初始化

打开 ashmem.h 可以看到以下宏和结构体：

代码清单5-16: ashmem.h中的宏和结构体

```
#define ASNMEM_NAME_DEF          "dev/ashmem"
#define ASHMEM_NOT_PURGED  0
#define ASHMEM_WAS_PURGED  1
#define ASHMEM_IS_UNPINNED 0
#define ASHMEM_IS_PINNED   1
Struct ashmem_pin{
    __u32  offset;
    __u32  len;
}
```

这段代码对设备的名称和 ASHMEM\_GET\_PIN\_STATUS 返回的值进行了设置。返回值是 1 表示 pin，返回值是 0 表示 unpin。同时 ashmem\_pin() 函数设置了 Ashmem 区域的偏移量 offset 和从偏移量开始的长度 len。

另外一些宏用于设置 Ashmem 的名称和状态，以及 pin 和 unpin 等操作。

ashmem.c 文件中包含了诸多功能函数,如图 5-8 所示。

Ashmem 的初始化操作由 init() 函数实现,退出操作由 exit() 函数实现。

init 的实现很简单,定义结构体 ashmem\_area 代表匿名共享内存区,定义结构体 ashmem\_range 代表 unpinned 页面的区域,代码如下:

代码清单5-17: 定义ashmem\_area结构体

```
Struct ashmem_area {
    Char name[ASHMEM_FULL_NAME_LEN];
    Struct list_head unpinned_list;
    Struct file *file;
    size_t size;
    unsigned long prot_mask;
};
```

ashmem\_area 的生命周期为文件的 open 和 release 操作之间的时间。name 表示为这块共享内存设置的名称,显示在 /proc/pid/maps 文件中; unpinned\_list 是所有共享内存的列表,用于将所有共享区连接起来; file 是共享内存中的备份文件,对应于临时文件系统 tmpfs 中的文件,当内核回收块共享内存对应的物理页面时,为保证进程的访问,会将该内存中的内容交换到临时文件中; size 表示这块共享内存的大小; prot\_mask 表示这块共享内存的访问保护位。

代码清单5-18: 定义ashmem\_range结构体

```
Struct ashmem_range {
    Struct list_head lru;
    Struct list_head unpinned;
    Struct ashmem_area *asma;
    size_t pgstart;
    size_t pgend;
    unsigned int purged;
};
```

ashmem\_range 定义了 LRU 列表、unpinned 列表、ashmem\_area 结构、开始和结束页面等,生命周期为 unpin 到 pin。

初始化时首先通过 kmem\_cache\_create 创建一个高速缓存 cache。如果创建成功,则返回指向 cache 的指针,如果创建失败,则返回 NULL。当对 cache 的新的页面分配成功时运行 ctor 构造函数,采用 unlikely 来对其创建结果进行判断。如果成功,就接着创建 ashmem\_range 的 cache。创建完成之后,通过 misc\_register 函数将 Ashmem 注册为 misc 设备。最后调用 register\_shrinker 注册回收函数 ashmem\_shrink,对所有创建的 cache 进行回收。ashmem\_shrink 定义在结构体 ashmem\_shrinker 当中。至此, Ashmem 初始化操作完成,实现代码如下所示:

```
● § lru_del(struct ashmem_range*) : void
● § range_alloc(struct ashmem_area*, struct as
● § range_del(struct ashmem_range*) : void
● § range_shrink(struct ashmem_range*, size_t
● § ashmem_open(struct inode*, struct file*) : ir
● § ashmem_release(struct inode*, struct file*)
● § ashmem_mmap(struct file*, struct vm_area
● § ashmem_shrink(int, gfp_t) : int
● § ashmem_shrinker : struct shrinker
● § set_prot_mask(struct ashmem_area*, unsig
● § ashmem_pin(struct ashmem_area*, size_t,
● § ashmem_unpin(struct ashmem_area*, size_
● § ashmem_get_pin_status(struct ashmem_ar
● § ashmem_ioctl(struct file*, unsigned int, unsi
● § ashmem_fops : struct file_operations
● § ashmem_misc : struct miscdevice
+ module_init()
+ module_exit()
```

图 5-8 ashmem.c 文件中的功能函数

代码清单5-19: ashmem\_init()函数

---

```
static int __init ashmem_init(void) {
    int ret;
    ashmem_area_cachep = kmem_cache_create("ashmem_area_cache",
        sizeof(struct ashmem_area), 0, 0, NULL);
    if (unlikely(!ashmem_area_cachep)) {
        printk(KERN_ERR "ashmem: failed to create slab cache\n");
        return -ENOMEM;
    }
    ret = misc_register(&ashmem_misc);
    if (unlikely(ret)) {
        printk(KERN_ERR "ashmem: failed to register misc device!\n");
        return ret;
    }
    register_shrinker(&ashmem_shrinker);
    printk(KERN_INFO "ashmem: initialized\n");
    return 0;
}
```

---

Ashmem 初始化的同时还需要注册 Ashmem 设备, 代码如下所示:

代码清单5-20: 注册Ashmem设备

---

```
Static struct file_operations ashmem_fops={
    .owner = THIS_MODULE,
    .open = ashmem_open,
    .release = ashmem_release,
    .mmap = ashmem_mmap,
    .unlocked_ioctl = ashmem_ioctl,
    .compat_ioctl = ashmem_ioctl,
};
Static struct miscdevice ashmem_misc = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "ashmem ",
    .fops = & ashmem_fops,
}
```

---

代码中定义了打开和释放 ashmem 的函数 ashmem\_open 和 ashmem\_release, 定义了 mmap() 函数和 ioctl() 函数。其中 mmap() 函数的作用是将文件描述符 fd 所指定的文件区域映射至调用进程的内存区域, 以便内存进行访问和修改。ioctl() 函数的作用是在设备驱动程序中对设备的 I/O 通道进行管理。

代码清单5-21: ashmem\_exit()函数

---

```
Ashmem退出时调用的是ashmem_exit()函数, 代码如下所示:
static void __exit ashmem_exit(void){
    int ret;
```

---

```

unregister_shrinker(&ashmem_shrinker);
ret = misc_deregister(&ashmem_misc);
if (unlikely(ret))
    printk(KERN_ERR "ashmem: failed to unregister misc device!\n");
kmem_cache_destroy(ashmem_range_cachep);
kmem_cache_destroy(ashmem_area_cachep);
printk(KERN_INFO "ashmem: unloaded\n");
}

```

本段代码中，`unregister_shrinker` 卸载注册的回收函数，`misc_deregister` 卸载 Ashmem 创建的设备 `misc`，`kmem_cache_destroy` 卸载释放生成的高速缓存 `cache`。

### 5.5.3 内存的创建和释放

创建内存就是为应用程序分配一定的内存空间，主要由 `open` 操作实现，工作流程可描述为以下 3 个步骤：

- 1) 打开 `/dev/ashmem` 文件。
- 2) 通过 `ioctl` 来设置名称和尺寸等。
- 3) 调用 `mmap` 将 Ashmem 分配的空间映射到进程空间。

通常只要打开 `/dev/ashmem` 设备并 `mmap`，就会获得内存空间。本节着重介绍前两个步骤，第三步将在 5.5.4 节中进行详细的描述。

#### 1. 内存的创建

在 Android 应用程序框架层提供的 `MemoryFile` 类的构造函数中，创建了匿名共享内存，这个构造函数位于 `frameworks/base/core/java/android/os/MemoryFile.java` 文件中，代码如下所示：

代码清单5-22: `MemoryFile`类

```

Public class MemoryFile
{
    .....
    Private static native FileDescriptor native_open(String name, int length) throws
        IOException;
    .....
    Private FileDescriptor mFD;
    .....
    Private int mLength;
    .....
    Public MemoryFile(String name, int length) throws IOException {
        mLength = length;
        mFD = native_open(name, length);
        .....
    }
    .....
}

```

本段代码中，执行创建操作的是 `native_open()` 函数。这部分代码实现在 `frameworks/base/core/jni/adroid_os_MemoryFile.cpp` 文件中，代码如下所示：

代码清单5-23: `native_open()`函数

---

```
Static jobject android_os_MemoryFile_open(JNIEnv* env, jobject clazz, jstring name,
    jint length) {
    Const char* namestr = (name ? env->GetStringUTFChars(name, NULL) : NULL);
    Int result = ashmem_create_region(namestr, length);
    If (name)env->ReleaseStringUTFChars(name, namestr);
    If (result < 0) {
        jniThrowException(env, "java/io/IOException", "ashmem_create_region failed");
        return NULL;
    }
    Return jniCreateFileDescriptor(env, result);
}
```

---

代码中创建匿名共享内存的操作又通过调用 `ashmem_create_region()` 函数来实现，这个函数实现在 `system/core/libcutils/ashmem-dev.c` 文件中，代码如下所示：

代码清单5-24: `ashmem_create_region()`函数

---

```
Int ashmem_create_region(const char *name, size_t size){
    Int fd, ret;
    Fd = open(ASHMEM_DEVICE, O_RDWR);
    If (fd < 0)
        Return fd;
    If (name) {
        Char buf[ASHMEM_NAME_LEN];
        strcpy(buf, name, sizeof(buf));
        ret = ioctl(fd, ASHMEM_SET_NAME, buf);
        if (ret < 0)
            goto error;
    }
    Ret = ioctl(fd, ASHMEM_SET_SIZE, size);
    if (ret<0)
        goto error;
    return fd;
error:
    close(fd);
    return ret;
}
```

---

通过一个 `open()` 函数和两个 `ioctl()` 函数与 `Ashmem` 驱动程序进行交互，前者是打开设备文件 `ASHMEM_DEVICE`，后者分别是设置匿名共享内存的名称和大小。下文将对这 3 个操作做进一步的分析。

### (1) open() 函数

open() 函数执行打开文件的操作，其参数 ASHMEM\_DEVICE 就是匿名共享内存设备文件 /dev/ashmem。open() 函数通过 ashmem\_open() 函数进入 Ashmem 驱动程序，代码如下所示：

代码清单5-25: ashmem\_open()函数

---

```
Static int ashmem_open(struct inode *inode, struct file *file)
{
    Struct ashmem_area *asma;
    Int ret;
    Ret = nonseekable_open(inode, file);
    If (unlikely(ret))
        Return ret;
    Asma = kmem_cache_zalloc(ashmem_area_cachep, GFP_KERNEL);
    If (unlikely(!asma))
        Return -ENOMEM;
    INIT_LIST_HEAD(&asma->unpinned_list);
    memcpy(asma->name, ASHMEM_NAME_PREFIX, ASHMEM_NAME_PREFIX_LEN);
    asma->prot_mask = PROT_MASK;
    file->private_data = asma;
    return 0;
}
```

---

首先是通过 nonseekable\_open 函数来设置这个文件不可以执行定位操作，即不可以执行 seek 文件操作。通过 kmem\_cache\_zalloc 函数从 ashmem\_area\_cachep 创建包含地址空间信息的 ashmem\_area 结构体，并保存在变量 asma 中，完成 asma 的初始化。最后将 asma 变量保存在 file 结构的 private\_data 域中，以便排除进程间共享内存的可能性。

至此，新的共享内存空间分配完成。

### (2) ioctl() 函数

ashmem\_create\_region 函数的两个 ioctl() 函数为新建的共享内存设置名称和大小。在 kernel/comon/mm/include/ashmem.h 文件中，ASHMEM\_SET\_NAME 和 ASHMEM\_SET\_SIZE 的定义如下：

```
#define ASHMEM_NAME_LEN 256
#define __ASHMEMIOC 0x7
#define ASHMEM_SET_NAME_IOW(__ASHMEMIOC, 1, char[ASHMEM_NAME_LEN])
#define ASHMEM_SET_SIZE_IOW(__ASHMEMIOC, 3, size_t)
```

ASHMEM\_SET\_NAME 命令中的 ioctl() 函数调用代码如下所示：

代码清单5-26: ASHMEM\_SET\_NAME和ASHMEM\_SET\_SIZE的定义

---

```
Static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
```

---

```

Struct ashmem_area *asma = file->private_data;
Long ret = -ENOTTY;
Switch (cmd) {
Case ASHMEM_SET_NAME:
Ret = set_name(asma, (void __user *) arg);
break;
.....
}
Return ret;
}

```

通过 `set_name()` 函数实现命名操作，该函数的代码如下所示：

代码清单5-27: `set_name()`函数

```

Static int set_name(struct ashmem_area *asma, void __user *name)
{
Int ret = 0;
mutex_lock(&ashmem_mutex);
if (unlikely(asma->file)) {
ret = -EINVAL;
goto out;
}
If (unlikely(copy_from_user(asma->name + ASHMEM_NAME_PREFIX_LEN,
name, ASHMEM_NAME_LEN)))
ret = -EFAULT;
asma->name[ASHMEM_FULL_NAME_LEN-1] = '\0';
out:
mutex_unlock(&ashmem_mutex);
return ret;
}

```

本段代码是将用户空间传进来的匿名共享内存的名称映射到 `asma->name` 域中去。需要说明的是匿名共享内存块的名称分为两部分，前缀部分由 `open` 操作默认添加 `ASHMEM_NAME_PREFIX`，即 `dev/ashmem/`；后一部分由用户自行设置，也可以调用 `ASHMEM_SET_NAME` 命令来设置。

`ASHMEM_SET_SIZE` 命令中 `ioctl()` 函数的调用，代码如下所示：

代码清单5-28: `ioctl()`函数

```

Static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
Struct ashmem_area *asma = file->private_data;
Long ret = -ENOTTY;
Switch (cmd) {
.....
Case ASHMEM_SET_SIZE:

```



```

        Ret = -EINVAL;
        If (!asma->file) {
            Ret = 0;
            asma->size = (size_t) arg; }
        break;
        ..... }
        Return ret;
    }

```

本段代码是将用户空间传进来的内存空间大小的值保存在对应的 `asma->size` 域中。

这样，`ashmem_create_region` 函数就执行完成了，层层返回，最后回到应用程序框架层提供的接口 `Memory` 的构造函数中，整个匿名共享内存的创建过程就完成了。

## 2. 内存的释放

`ashmem_release()` 函数执行的操作是将指定的地址空间从链表中进行删除，并释放相关的物理空间，代码如下所示：

代码清单5-29: `ashmem_release()`函数

```

static int ashmem_release(struct inode *ignored, struct file *file)
{
    struct ashmem_area *asma = file->private_data;
    struct ashmem_range *range, *next;

    mutex_lock(&ashmem_mutex);
    list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned)
        range_del(range);
    mutex_unlock(&ashmem_mutex);

    if (asma->file)
        fput(asma->file);
    kmem_cache_free(ashmem_area_cachep, asma);

    return 0;
}

```

上段代码中 `list_for_each_entry_safe()` 函数存储指定释放空间的下一地址，避免因某一地址空间指针的删除导致断链。`range_del()` 执行链表指针的删除操作。

## 5.5.4 内存的映射

Ashmem 驱动程序不提供 `read` 和 `write` 操作，所有对于共享内存的访问操作都需要先将访问的内容映射到自己的进程空间才可以进行。在 `MemoryFile` 类的构造函数中，实现了匿名共享内存的创建，还需要把匿名共享内存设备文件映射到进程空间，代码如下：

代码清单5-30: MemoryFile类的构造函数

---

```

Public class MemoryFile
{
    .....
    Private static native int native_mmap(FileDescriptor fd, int length, int mode)
    Throws IOException;
    .....
    Private int mAddress;
    .....
    Public MemoryFile(String name, int length) throws IOException
    {
        .....
        mAddress = native_mmap(mFD, length, PROT_READ|PROT_WRITE);
        .....
    }
}

```

---

映射操作通过 `native_mmap()` 函数来实现, 该函数的相关代码实现在 `frameworks/base/core/jni/adroid_os_MemoryFile.cpp` 文件中, 代码如下:

代码清单5-31: native\_mmap()函数

---

```

static jint android_os_MemoryFile_mmap(JNIEnv* env, jobject clazz, jobject
    fileDescriptor,
    jint length, jint prot)
{
    int fd = jniGetFDFFromFileDescriptor(env, fileDescriptor);
    jint result = (jint)mmap(NULL, length, prot, MAP_SHARED, fd, 0);
    if (!result)
        jniThrowException(env, "java/io/IOException", "mmap failed");
    return result;
}

```

---

文件的映射操作由 `mmap()` 函数实现, 该函数最终由 Ashmem 驱动程序的 `ashmem_mmap()` 函数实现, 函数代码如下所示:

代码清单5-32: ashmem\_mmap()函数

---

```

Static int ashmem_mmap(struct file *file, struct vm_area_struct *vma)
{
    Struct ashmem_area *asma = file->private_data;
    Int ret = 0;
    mutex_lock(&ashmem_mutex);
    if (unlikely(!asma->size)) {
        ret = -EINVAL;
        goto out;
    }
    If (unlikely((vma->vm_flags & ~asma->prot_mask) & PROT_MASK)) {
        Ret = -EPERM;
    }
}

```

---

```

Goto out;
}
If (!asma->file) {
Char *name = ASHMEM_NAME_DEF;
Struct file *vmfile;
If (asma->name[ASHMEM_NAME_PREFIX_LEN] != '\0')
Name = asma->name;
Vmfile = shmem_file_setup(name, asma->size, vma->vm_flags);
If (unlikely(IS_ERR(vmfile))) {
Ret = PTR_ERR(vmfile);
Goto out;
}
asma->file = vmfile;
}
get_file(asma->file);
if (vma->vm_flags & VM_SHARED)
shmem_set_file(vma, asma->file);
else {
if (vma->vm_file)
fput(vma->vm_file);
vma->vm_file = asma->file;
}
vma->vm_flags |= VM_CAN_NONLINEAR;
out:
mutex_unlock(&ashmem_mutex);
return ret;
}

```

ashmem\_mmap() 函数实现比较简单，主要是调用 Linux 内核提供的 shmem\_file\_setup() 函数。由此可见，最终实现 Ashmem 驱动操作的依然是 Linux 内核程序。

ashmem\_mmap() 函数执行完成后，将虚拟空间的起始地址层层返回到 JNI 方法 native\_mmap() 函数中，最终返回到应用程序框架层的 MemoryFile 类的构造函数中，并保存在成员变量 mAddress 中。至此，进程的匿名共享内存申请注册完成。

### 5.5.5 内存的锁定和解锁

匿名共享内存的锁定和解锁由函数 ashmem\_pin\_region() 和 ashmem\_unpin\_region() 实现在 system/core/libcutils/ashmem-dev.c 文件中，代码如下所示：

代码清单5-33: ashmem\_pin\_region()函数

```

Int ashmem_pin_region(int fd, size_t offset, size_t len)
{
Struct ashmem_pin pin={offset, len};
Return ioctl(fd, ASHMEM_PIN, &pin);
}

```

```

Int ashmem_unpin_region(int fd, size_t offset, size_t len)
{
    Struct ashmem_pin pin={offset, len};
    Return ioctl(fd, ASHMEM_UNPIN, &pin);
}

```

两个操作均由 `ioctl()` 函数实现，其中的参数 `ASHMEM_PIN` 和 `ASHMEM_UNPIN` 在 `kernel/common/include/linux/ashmem.h` 文件中被定义。

代码清单5-34：参数`ASHMEM_PIN`和`ASHMEM_UNPIN`

```

#define __ASHMEMIOC 0x77
#define ASHMEM_PIN_IOW(__ASHMEMIOC, 7, struct ashmem_pin)
#define ASHMEM_UNPIN_IOW(__ASHMEMIOC, 8, struct ashmem_pin)

```

包含的参数类型为一个结构体变量 `ashmem_pin`，实现代码如下所示：

代码清单5-35：`ashmem_pin`结构体

```

Struct ashmem_pin{
    __u32 offset;
    __u32 len; };

```

该结构体表示锁定或者解锁对象在内存空间的起始位置以及大小。

`ashmem_ioctl` 函数中对于 `ASHMEM_PIN` 和 `ASHMEM_UNPIN` 的操作代码如下所示：

代码清单5-36：`ashmem_ioctl()`函数

```

Static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    Struct ashmem_area *asma = file->private_data;
    Long ret = -ENOTTY;
    switch (cmd) {
        .....
        Case ASHMEM_PIN:
        Case ASHMEM_UNPIN:
            Ret = ashmem_pin_unpin(asma, cmd, (void__user*)arg);
            break;
        .....
    }
    Return ret;
}

```

这两个操作是由 `ashmem_pin_unpin()` 函数来完成，代码如下所示：

代码清单5-37：`ashmem_pin_unpin()`函数

```

Static int ashmem_pin_unpin(struct ashmem_area *asma,
    unsigned long cmd, void __user *p){
    struct ashmem_pin pin;

```

```

size_t pgstart, pgend;
--
.....
pgstart = pin.offset / PAGE_SIZE;
pgend = pgstart + (pin.len / PAGE_SIZE) - 1;
mutex_lock(&ashmem_mutex);
switch (cmd) {
case ASHMEM_PIN:
ret = ashmem_pin(asma, pgstart, pgend);
break;
case ASHMEM_UNPIN:
ret = ashmem_unpin(asma, pgstart, pgend);
break;
.....
}
mutex_unlock(&ashmem_mutex);
return ret;
}

```

首先是获得用户空间传进来的参数，并保存在本地变量 `pin` 中，由于 `struct ashmem_pin` 类型的变量对起始地址和内存大小都是以字节为单位进行描述的，因此需要转换成以页面为单位，并且保存在本地变量 `pgstart` 和 `pgend` 中。如果从用户空间传进来的内存块的大小值为 0，则认为是要 `pin/unpin` 整个匿名共享内存。

函数根据传递过来的参数 `ASHMEM_PIN` 或 `ASHMEM_UNPIN`，分别执行 `ashmem_pin()` 函数或 `ashmem_unpin()` 函数。需要说明的是，执行 `ASHMEM_PIN` 操作时，目标对象必须是一块当前处于 `unpinned` 状态的内存块。

Ashmem 的源代码实现相对比较简单，用户仅需要通过 `iocl()` 函数就可以设置内存空间的名称和大小，并执行 `pin` 和 `unpin` 等操作，借助 Linux 内核中已有的工具，来实现高效的内存管理。

## 5.6 低内存管理

内存资源的紧缺是任何一个系统都面临的问题，尤其是现在大多数的操作系统都是多任务处理系统。Android 作为一个多任务处理系统，为了加快程序启动的速度，退出时，系统通常不会彻底关闭该程序。随着系统中保留的程序越来越多，内存肯定会出现不足，此时需要强制释放一部分程序所占用的内存资源，也就是通常所说的 `kill` 掉一个程序。在 Linux 中内存是以页面为单位分配的，如果申请页面在分配时出现内存不足，则会通过名为 `OOM` (Out Of Memory) 的机制来选择一个进程，将其 `kill` 掉，释放其所占用的内存资源。Android 则使用了一个新的机制——`Low Memory Killer` (低内存管理) 来完成同样的任务。

`Low Memory Killer` 机制与标准的 Linux `OOM` 机制类似，但实现方式略有不同。Linux

中 OOM Killer 机制在 mm/oom\_kill.c 中实现，在 mm/page\_alloc.c 中分配内存时，被 `_alloc_pages_may_oom` 调用。oom\_kill.c 最主要的函数是 `out_of_memory()` 函数，负责选择 bad 进程进行 kill。两种机制相同的是 kill 方法都会发送 SIGKILL 信号。在 `out_of_memory` 中通过调用 `select_bad_process` 来选择一个进程 kill，选择的依据在 `badness` 函数中实现，基于多个标准来给每个进程评分，评分最高的被选中并 kill。一般而言，占用内存越多，oom\_adj 就越大，也就越有可能被选中。

1. Low Memory Killer 的原理

Low Memory Killer 基于 Linux 的 OOM 机制改进而来，OOM 机制通过一些比较复杂的评分机制对进程进行打分，然后将分数高的进程判定为 bad 进程，kill 掉并释放内存。OOM 机制只有当系统内存不足的时候才会启动检查，而 Low Memory Killer 定时进行检查。

Low Memory Killer 是根据进程的重要性和释放这个进程可获取的空闲内存数量来决定释放的进程。进程的内存通过 `get_mm_rss()` 函数获取，在相同的 oom\_adj 下，内存大的优先被 kill。进程的重要性由 `task_struct->signal_struct->oom_adj` 决定。Android 将程序分成几类，按照重要性依次降低的顺序，如表 5-1 所示。

表 5-1 重要性顺序

名 称	oom_adj	解 释
FOREGROUD_APP	0	前台程序，即正在使用的程序
VISIBLE_APP	1	用户可见的程序
SECONDARY_SERVER	2	后台服务
HOME_APP	4	HOME，即主界面
HIDDEN_APP	7	被隐藏的程序
CONTENT_PROVIDER	14	内容提供者
EMPTY_APP	15	空程序，既不提供服务，也不提供内容

每类程序都会有一个 oom\_adj 值，这个值越小，程序越重要，被 kill 掉的可能性越低。

Low Memory Killer 通过进程的 oom\_adj 来判定进程的重要程度。oom\_adj 的大小和进程的类型以及被调用的次序有关。Low Memory Killer 为用户空间指定了一组临界值，若进程描述中的 oom\_adj 值在其中某个值的范围内，该进程将被 kill 掉。其中 oom\_adj 的最小值定义在 `/sys/module/lowmemorykiller/parameters/adj` 中，储存空闲页面的数量定义在 `/sys/module/lowmemorykiller/parameters/minfree` 中，所有的值都用一个逗号将其隔开，且以升序排列。代码如下所示：

代码清单5-38：储存空闲页面的数量定义

```
static int lowmem_adj[6] = {
    0,
```

```

    1,
    6,
    12,
};
static int lowmem_adj_size = 4;
static size_t lowmem_minfree[6] = {
    3*512, // 6MB
    2*1024, // 8MB
    4*1024, // 16MB
    16*1024, // 64MB
};
static int lowmem_minfree_size = 4;

```

Lowmeme\_adj 中各项数值代表阈值的警戒级数，lowmem\_minfree 代表对应级数的剩余内存。由以上两段代码可知，若系统的剩余内存小于 6MB，警戒级数为 0；若系统内存剩余小于 8M 而大于 6M，警戒级数为 1；若系统内存剩余小于 16M 而大于 8M，警戒级数为 6；若系统内存小于 64M 大于 16MB，警戒级数为 12。

Low Memory Killer 的规则就是根据当前系统的剩余内存多少来获取当前的警戒级数，如果进程的 oom\_adj 大于警戒级数并且最大，进程将会被 kill。oom\_adj 越小，代表进程的重要性越高。由于前台进程的 oom\_adj 值会比较小，后台服务的 omm\_adj 值会比较大，因此当内存不足的时候，Low Memory Killer 会选择 kill 掉后台服务。

## 2. Low Memory Killer 的具体实现

Low Memory Killer 驱动的实现非常简单，位于 drivers/misc/lowmemorykiller.c。代码如下：

代码清单5-39: Low Memory Killer驱动

```

static int __init lowmem_init(void)
{
    register_shrinker(&lowmem_shrinker);
    return 0;
}
static void __exit lowmem_exit(void)
{
    unregister_shrinker(&lowmem_shrinker);
}
module_init(lowmem_init);
module_exit(lowmem_exit);

```

初始化函数 lowmem\_init() 通过 register\_shrinker() 函数注册了 shrinker 为 lowmem\_shrinker。退出时调用 lowmem\_exit() 函数，通过 unregister\_shrinker() 函数来卸载被注册的 lowmem\_shrinker。shrinker 的主要作用是在内存分页回收时根据规则释放内存，代码如下所示：

代码清单5-40: 注册shrinker

---

```
static struct shrinker lowmem_shrinker = {
    .shrink = lowmem_shrink,
    .seeks = DEFAULT_SEEKS * 16
};
```

---

`lowmem_shrink` 为回调函数的指针，是这个驱动的核心实现，当内存不足时就会调用 `lowmem_shrink` 方法来 kill 掉某些进程。下面来分析其具体实现，实现代码如下：

代码清单5-41: `lowmem_shrink()`函数

---

```
static int lowmem_shrink(int nr_to_scan, gfp_t gfp_mask)
{
    struct task_struct *p;
    struct task_struct *selected = NULL;
    int rem = 0;
    int tasksize;
    int i;
    int min_adj = OOM_ADJUST_MAX + 1;
    int selected_tasksize = 0;
    int selected_oom_adj;
    int array_size = ARRAY_SIZE(lowmem_adj);
    int other_free = global_page_state(NR_FREE_PAGES);
    int other_file = global_page_state(NR_FILE_PAGES);
    if(lowmem_adj_size < array_size)
        array_size = lowmem_adj_size;
    if(lowmem_minfree_size < array_size)
        array_size = lowmem_minfree_size;
    for(i = 0; i < array_size; i++) {
        if (other_free < lowmem_minfree[i] &&
            other_file < lowmem_minfree[i]) {
            min_adj = lowmem_adj[i];
            break;
        }
    }
    if(nr_to_scan > 0)
        lowmem_print(3, "lowmem_shrink %d, %x, ofree %d %d, ma %d\n", nr_to_scan,
            gfp_mask, other_free, other_file, min_adj);
    rem = global_page_state(NR_ACTIVE_ANON) +
        global_page_state(NR_ACTIVE_FILE) +
        global_page_state(NR_INACTIVE_ANON) +
        global_page_state(NR_INACTIVE_FILE);
    if (nr_to_scan <= 0 || min_adj == OOM_ADJUST_MAX + 1) {
        lowmem_print(5, "lowmem_shrink %d, %x, return %d\n", nr_to_scan, gfp_mask,
            rem);
        return rem;
    }
}
```

---



```

selected_oom_adj = min_adj;
read_lock(&tasklist_lock);
for_each_process(p) {
    struct mm_struct *mm;
    int oom_adj;
    task_lock(p);
    mm = p->mm;
    if (!mm) {
        task_unlock(p);
        continue;
    }
    oom_adj = mm->oom_adj;
    if (oom_adj < min_adj) {
        task_unlock(p);
        continue;
    }
    tasksize = get_mm_rss(mm);
    task_unlock(p);
    if (tasksize <= 0)
        continue;
    if (selected) {
        if (oom_adj < selected_oom_adj)
            continue;
        if (oom_adj == selected_oom_adj && tasksize <= selected_tasksize)
            continue;
    }
    selected = p;
    selected_tasksize = tasksize;
    selected_oom_adj = oom_adj;
    lowmem_print(2, "select %d (%s), adj %d, size %d, to kill\n",
        p->pid, p->comm, p->oomkilladj, tasksize);
}
if(selected != NULL) {
    lowmem_print(1, "send sigkill to %d (%s), adj %d, size %d\n",
        selected->pid, selected->comm,
        selected->oomkilladj, selected_tasksize);
    force_sig(SIGKILL, selected);
    rem -= selected_tasksize;
}
lowmem_print(4, "lowmem_shrink %d, %x, return %d\n", nr_to_scan, gfp_mask, rem);
read_unlock(&tasklist_lock);
return rem;
}

```

本段代码中首先确定 `lowmem_adj_size` 和 `lowmem_minfree_size` 数组的大小（元素个数）是否一致，如果不一致则以最小的为基准。通过比较 `lowmem_minfree_size` 中的空闲储存空间的值，以确定最小 `min_adj` 值。检测 `min_adj` 的值是否是初始值 `OOM_ADJUST_MAX + 1`，

如果是表示没有满足条件的 `min_adj` 值，否则进入下一步，使用循环对每一个进程块进行判断，通过 `min_adj` 来寻找满足条件的具体进程。最后，对找到的进程进行 `NULL` 判断，通过 `force_sig(SIGKILL, selected)` 发送一条 `SIGKILL` 信号到内核，kill 掉被选中的 `selected` 进程。

通过比较获得 `min_adj` 与 `selected_oom_adj` 值的代码如下所示：

代码清单5-42：获得`min_adj`与`selected_oom_adj`值

---

```
int array_size = ARRAY_SIZE(lowmem_adj);
int other_free = global_page_state(NR_FREE_PAGES);
int other_file = global_page_state(NR_FILE_PAGES);

if (lowmem_adj_size < array_size)
    array_size = lowmem_adj_size;
if (lowmem_minfree_size < array_size)
    array_size = lowmem_minfree_size;
for (i = 0; i < array_size; i++) {
    if (other_free < lowmem_minfree[i] && other_file < lowmem_minfree[i]) {
        min_adj = lowmem_adj[i];
        break;
    }
}
selected_oom_adj = min_adj;
```

---

在两段代码中多次使用了 `global_page_state()` 函数。该函数定义在 `linux/vmstat.h` 中，其参数使用 `zone_stat_item` 枚举，定义在 `linux/mmzone.h` 中，具体代码如下：

代码清单5-43：`zone_stat_item`枚举

---

```
enum zone_stat_item {
    NR_FREE_PAGES,
    NR_LRU_BASE,
    NR_INACTIVE_ANON = NR_LRU_BASE,
    NR_ACTIVE_ANON,
    NR_INACTIVE_FILE,
    NR_ACTIVE_FILE,
#ifdef CONFIG_UNEVICTABLE_LRU
    NR_UNEVICTABLE,
    NR_MLOCK,
#else
    NR_UNEVICTABLE = NR_ACTIVE_FILE,
    NR_MLOCK = NR_ACTIVE_FILE,
#endif
    NR_ANON_PAGES,
    NR_FILE_MAPPED,
    NR_FILE_PAGES,
    NR_FILE_DIRTY,
    NR_WRITEBACK,
```

```

NR_SLAB_RECLAIMABLE,
NR_SLAB_UNRECLAIMABLE,
NR_PAGETABLE,
NR_UNSTABLE_NFS,
NR_BOUNCE,
NR_VMSCAN_WRITE,
NR_WRITEBACK_TEMP,
#ifdef CONFIG_NUMA
    NUMA_HIT,
    NUMA_MISS,
    NUMA_FOREIGN,
    NUMA_INTERLEAVE_HIT,
    NUMA_LOCAL,
    NUMA_OTHER,
#endif
    NR_VM_ZONE_STAT_ITEMS };

```

其中 NR\_ANON\_PAGES 表示匿名映射页面, NR\_FILE\_MAPPED 表示映射页面, NR\_WRITEBACK\_TEMP 表示使用临时缓冲区, NUMA\_HIT 表示在预定节点上分配, NUMA\_MISS 表示在非预定节点上分配, NUMA\_LOCAL 表示从本地页面分配, NUMA\_OTHER 表示从其他节点分配。

## 5.7 Logger

### 5.7.1 Logger 概述

Logger 是 Android 系统提供的日志设备, 在开发应用的过程中可以使用 Log 信息来调试程序。引入 Log 信息基于以下 3 个目的:

- 1) 监视代码中变量的变化情况, 周期性地记录到文件中, 供其他应用进行统计分析工作。
- 2) 跟踪代码运行时轨迹, 作为日后审计的依据。
- 3) 担当集成开发环境中的调试器的作用, 向文件或控制台打印代码的调试信息。

Log 信息的实现需要 Android 的 Logger 驱动为用户层提供 Log 支持。无论是底层的源代码还是上层的应用, 都可以通过 Logger 来进行调试。Logger 一共包括以下 3 个设备节点:

- /dev/log/main
- /dev/log/event
- /dev/log/radio

3 个设备节点的驱动源文件位于:

- include/linux/logger.h
- include/linux/logger.c

### 5.7.2 Logger 实现原理

Logger 驱动中创建了结构体 `logger_entry`，其中定义了每一条日志信息的属性，代码如下所示，位于 `include/linux/logger.h` 文件中。

代码清单5-44：结构体 `logger_entry`

---

```
struct logger_entry {
    __u16      len;
    __u16      __pad;
    __s32      pid;
    __s32      tid;
    __s32      sec;
    __s32      nsec;
    char        msg[0];
};
```

---

其中，`len` 表示日志信息的有效长度；`__pad` 没有实质作用，但是需要使用两个字节来占位；`pid` 表示日志信息的生成进程的 `pid`；`tid` 表示日志信息的生成进程的 `tid`；`sec` 表示日志生成的时间，单位是秒；`nsec` 表示用纳秒来计算日志生成的时间，仅在 `sec` 不足 1 秒时调用；`msg` 储存该日志的有效信息。

对应于 Logger 的 3 个不同的设备节点定义了代表不同设备事件的宏，代码如下所示：

代码清单5-45：Logger的3个设备节点

---

```
#define LOGGER_LOG_MAIN "log_main"
#define LOGGER_LOG_EVENTS "log_events" /
#define LOGGER_LOG_RADIO "log_radio" /* 任何事件 */
```

---

分别表示任何事件、系统硬件事件、无线相关消息。

创建结构体 `logger_log`，其中定义了每一个日志设备的相关信息。将 `radio`、`events` 和 `main` 都用 `logger_log` 结构体来表示，代码如下所示：

代码清单5-46： `logger_log` 结构体

---

```
struct logger_log {
    unsigned char *      buffer;
    struct miscdevice     misc;
    wait_queue_head_t    wq;
    struct list_head      readers;
    struct mutex          mutex;
    size_t               w_off;
    size_t               head;
    size_t               size;
};
```

---

其中，`buffer` 表示该设备储存日志的环形缓冲区；`misc` 代表日志设备的 `miscdevice`，在

注册设备的时候需要使用；wq 表示一个等待队列，等待在该设备上读取日志的进程 readers；readers 表示读取日志的 readers 链表；mutex 则是用于多线程同步和保护该结构体的 mutex；w\_off 代表当前写入日志的位置，即在环形缓冲区中的偏移量；head 是一个读取日志的新的 readers，表示读取的起始位置，指环形缓冲区的偏移量；size 则代表该日志的大小，即环形缓冲区的大小。

日志的读取需要一个用于读取日志的 readers 来完成。readers 结构体的定义位于文件 include/linux/logger.c 中，代码如下所示：

代码清单5-47: logger\_reader 结构体

```
struct logger_reader {
    struct logger_log *    log;
    struct list_head      list;
    size_t                r_off;
};
```

其中 log 代表当前要读取数据的日志设备 (logger\_log)；list 指向日志设备的读取进程 (readers)；r\_off 则表示开始读取日志的一个偏移量，即日志设备中将要被读取的 buffer 的偏移量。

logger 驱动的工作流程如图 5-9 所示。

驱动首先启动 logger\_init() 函数完成初始化，通过 logger\_open() 函数完成日志文件的打开，由 logger\_poll() 函数判断是否可以对日志设备进行操作，由 logger\_read() 函数和 logger\_aio\_write() 函数完成对日志文件的读写操作，最后由 logger\_release() 函数关闭日志并释放所占用的空间。各个函数的具体实现下文将进一步进行介绍。

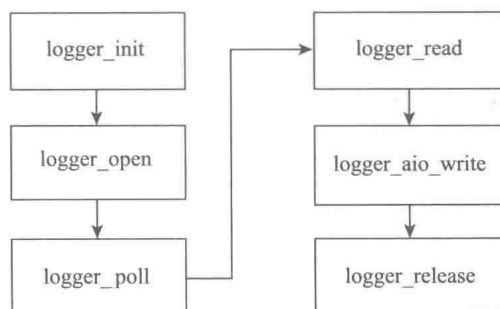


图 5-9 logger 驱动的工作流程

### (1) logger\_init()

当系统内核启动后，系统 init 过程会调用 device\_initcall 所指向的 logger\_init() 函数来初始化日志设备。logger\_init() 函数代码如下所示：

代码清单5-48: logger\_init()函数

```
static int __init logger_init(void)
{
    int ret;
    ret = init_log(&log_main);
    if (unlikely(ret))
        goto out;
    ret = init_log(&log_events);
    if (unlikely(ret))
        goto out;
}
```

```

    ret = init_log(&log_radio);
    if (unlikely(ret))
        goto out;
    out:
    return ret;
}
device_initcall(logger_init);

```

logger\_init() 函数调用 init\_log() 函数完成日志系统 3 个设备节点的初始化。init\_log() 函数的代码实现如下:

代码清单5-49: init\_log()函数

```

static int __init init_log(struct logger_log *log)
{
    int ret;
    ret = misc_register(&log->misc);
    if (unlikely(ret)) {
        printk(KERN_ERR "logger: failed to register misc "
                "device for log '%s'!\n", log->misc.name);
        return ret;
    }
    printk(KERN_INFO "logger: created %luK log '%s'\n",
            (unsigned long) log->size >> 10, log->misc.name);
    return 0;
}

```

misc\_register() 函数来实现日志设备 miscdevice(logger\_log->misc) 的初始化。具体的日志设备初始化的操作由 DEFINE\_LOGGER\_DEVICE 宏来完成。DEFINE\_LOGGER\_DEVICE 的实现代码如下所示:

代码清单5-50: DEFINE\_LOGGER\_DEVICE宏

```

#define DEFINE_LOGGER_DEVICE(VAR, NAME, SIZE)
static unsigned char _buf_ ## VAR[SIZE];
static struct logger_log VAR = {
    .buffer = _buf_ ## VAR,
    .misc = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = NAME,
        .fops = &logger_fops,
        .parent = NULL,
    },
    .wq = __WAIT_QUEUE_HEAD_INITIALIZER(VAR .wq),
    .readers = LIST_HEAD_INIT(VAR .readers),
    .mutex = __MUTEX_INITIALIZER(VAR .mutex),
    .w_off = 0,
    .head = 0,
}

```

```
.size = SIZE,
};
```

DEFINE\_LOGGER\_DEVICE 中传入的 3 个参数 (VAR, NAME, SIZE), 作用就是使用 NAME 作为名称和 SIZE 作为尺寸来创建一个日志设备。通常 SIZE 的大小必须为 2 的幂, 并且要大于 LOGGER\_ENTRY\_MAX\_LEN, 小于 LONG\_MAX-LOGGER\_ENTRY\_MAX\_LEN。在 logger.h 文件中对表示日志的长度宏进行了定义, 同时还定义了 LOGGER\_ENTRY\_MAX\_PAYLOAD 表示日志的最大有效长度。

代码清单5-51: 日志长度宏定义

```
#define LOGGER_ENTRY_MAX_LEN (4*1024)
#define LOGGER_ENTRY_MAX_PAYLOAD
(LOGGER_ENTRY_MAX_LEN - sizeof(struct logger_entry))
```

对 3 个日志设备进行初始化的代码如下所示:

代码清单5-52: 日志设备初始化

```
DEFINE_LOGGER_DEVICE(log_main, LOGGER_LOG_MAIN, 64*1024)
DEFINE_LOGGER_DEVICE(log_events, LOGGER_LOG_EVENTS, 256*1024)
DEFINE_LOGGER_DEVICE(log_radio, LOGGER_LOG_RADIO, 64*1024)
```

在初始化过程中, 设备对应的 file\_operations 结构体有如下定义:

代码清单5-53: file\_operations 结构体

```
static struct file_operations logger_fops = {
    .owner = THIS_MODULE,
    .read = logger_read,
    .aio_write = logger_aio_write,
    .poll = logger_poll,
    .unlocked_ioctl = logger_ioctl,
    .compat_ioctl = logger_ioctl,
    .open = logger_open,
    .release = logger_release,
};
```

这段代码中包括了关于日志设备的各种操作函数和接口, 比如读取日志的 logger\_read、打开日志设备文件的 logger\_open、读取数据的 logger\_read 等。至此, 日志设备的初始化工作完成。

## (2) logger\_open()

该函数为打开日志设备文件的方法, 实现代码如下所示:

代码清单5-54: logger\_open()函数

```
static int logger_open(struct inode *inode, struct file *file)
{
```

```

struct logger_log *log;
int ret;
ret = nonseekable_open(inode, file);
if (ret)
    return ret;
log = get_log_from_minor(MINOR(inode->i_rdev));
if (!log)
    return -ENODEV;
if (file->f_mode & FMODE_READ) {
    struct logger_reader *reader;

    reader = kmalloc(sizeof(struct logger_reader), GFP_KERNEL);
    if (!reader)
        return -ENOMEM;
    reader->log = log;
    INIT_LIST_HEAD(&reader->list);
    mutex_lock(&log->mutex);
    reader->r_off = log->head;
    list_add_tail(&reader->list, &log->readers);
    mutex_unlock(&log->mutex);
    //保存数据到private_data
    file->private_data = reader;
} else
    file->private_data = log;

return 0;
}

```

代码中通过 `get_log_from_minor()` 函数根据日志设备的 `misc.minor` 参数和 `minor` 参数判断需要打开的日志设备的类型。实现代码如下所示：

代码清单5-55: `get_log_from_minor()`函数

```

static struct logger_log * get_log_from_minor(int minor)
{
    if (log_main.misc.minor == minor)
        return &log_main;
    if (log_events.misc.minor == minor)
        return &log_events;
    if (log_radio.misc.minor == minor)
        return &log_radio;
    return NULL;
}

```

`logger_open()` 函数获得日志设备的类型之后，需要判断读写模式。对于只读模式，需要创建一个 `logger_reader`，之后对其所需的数据进行初始化，包括指定日志设备、`mutex`、读取偏移量 `r_off` 等操作，最后将该 `logger_reader` 保存到 `file->private_data` 中。对于读写模式或



者写模式，则直接将日志设备 log 保存到 file->private\_data 中，便于在读写过程中通过 file->private\_data 来取得 logger\_reader 和 logger\_log。

### (3) logger\_poll()

该函数用来判断是否可以对日志设备进行操作，其代码如下所示：

代码清单5-56: logger\_poll()函数

---

```
static unsigned int logger_poll(struct file *file, poll_table *wait)
{
    struct logger_reader *reader;
    struct logger_log *log;
    unsigned int ret = POLLOUT | POLLWRNORM;
    if (!(file->f_mode & FMODE_READ))
        return ret;
    reader = file->private_data;
    log = reader->log;
    poll_wait(file, &log->wq, wait);
    mutex_lock(&log->mutex);
    //判断是否为空
    if (log->w_off != reader->r_off)
        ret |= POLLIN | POLLRDNORM;
    mutex_unlock(&log->mutex);
    return ret;
}
```

---

可以看出，POLLOUT 表示进程可以执行写操作。读操作中若是以 FMODE\_READ 模式打开日志设备的进程，需要借助 if() 函数判断当前日志缓冲区是否为空，只有日志缓冲区不为空才能实现日志的读操作。

### (4) logger\_read()

读数据的操作，实现代码清单如下：

代码清单5-57: logger\_read()函数

---

```
static ssize_t logger_read(struct file *file, char __user *buf,
                          size_t count, loff_t *pos)
{
    struct logger_reader *reader = file->private_data;
    struct logger_log *log = reader->log;
    ssize_t ret;
    DEFINE_WAIT(wait);
start:
    while (1) {
        prepare_to_wait(&log->wq, &wait, TASK_INTERRUPTIBLE);
        mutex_lock(&log->mutex);
        ret = (log->w_off == reader->r_off);
        mutex_unlock(&log->mutex);
    }
```

---

```

    if (!ret)
        break;
    if (file->f_flags & O_NONBLOCK) {
        ret = -EAGAIN;
        break;
    }
    if (signal_pending(current)) {
        ret = -EINTR;
        break;
    }
    schedule();
}
finish_wait(&log->wq, &wait);
if (ret) return ret;
mutex_lock(&log->mutex);
if (unlikely(log->w_off == reader->r_off)) {
    mutex_unlock(&log->mutex);
    goto start;
}
ret = get_entry_len(log, reader->r_off);
if (count < ret) {
    ret = -EINVAL;
    goto out;
}
ret = do_read_log_to_user(log, reader, buf, ret);
out:
mutex_unlock(&log->mutex);
return ret;
}

```

函数中通过 `file->private_data` 获取 `logger_reader` 及其日志设备 `logger_log`。通过 `prepare_to_wait()` 函数将当前进程添加到等待队列 `log->wq` 之中。通过偏移量 `ret` 来判断当前日志的 buffer 是否为空，如果为空，则停止运行。如果指定了非阻塞模式，则直接返回 `EAGAIN`。`while()` 循环不断地对 buffer 进行扫描，直到 buffer 中有可供读取的日志为止。通过 `get_entry_len()` 函数读取下一条日志，并通过 `do_read_log_to_user()` 将日志文件复制到用户空间。至此，读操作结束。

#### (5) `logger_aio_write()`

写操作支持同步、异步以及 `scatter` 等方式，本操作的实现基于 buffer 的环形缓冲区。当日志缓冲区被写满之后，环形的结构会用新写入的数据将最初的数据覆盖。因此为避免原来的数据被覆盖造成数据丢失，需要对原数据进行保护，方式是执行写操作的同时引入读操作。写操作的代码实现如下所示：

代码清单5-58: logger\_aio\_write()函数

---

```

ssize_t logger_aio_write(struct kiocb *iocb, const struct iovec *iov,
                        unsigned long nr_segs, loff_t ppos)
{
    struct logger_log *log = file_get_log(iocb->ki_filp);
    size_t orig = log->w_off;
    struct logger_entry header;
    struct timespec now;
    ssize_t ret = 0;
    now = current_kernel_time();
    header.pid = current->tgid;
    header.tid = current->pid;
    header.sec = now.tv_sec;
    header.nsec = now.tv_nsec;
    header.len = min_t(size_t, iocb->ki_left, LOGGER_ENTRY_MAX_PAYLOAD);
    if (unlikely(!header.len))
        return 0;
    mutex_lock(&log->mutex);
    fix_up_readers(log, sizeof(struct logger_entry) + header.len);
    do_write_log(log, &header, sizeof(struct logger_entry));
    while (nr_segs-- > 0) {
        size_t len;
        ssize_t nr;
        len = min_t(size_t, iov->iov_len, header.len - ret);

        nr = do_write_log_from_user(log, iov->iov_base, len);
        if (unlikely(nr < 0)) {
            log->w_off = orig;
            mutex_unlock(&log->mutex);
            return nr;
        }
        iov++;
        ret += nr;
    }
    mutex_unlock(&log->mutex);
    wake_up_interruptible(&log->wq);
    return ret;
}

```

---

本段代码中通过 `file_get_log()` 函数获取日志设备。对执行写操作的日志执行初始化操作, 包括进程的 `pid`、`tid` 和时间等日志数据的初始化。为避免原数据被覆盖, 在写操作之前执行 `fix_up_readers()` 函数, 来修正其偏移量 (`r_off`)。 `do_write_log()` 函数真正执行写入操作, `do_write_log_from_user()` 函数执行从用户空间写入日志的操作。

`fix_up_readers()` 函数的实现代码如下所示:

代码清单5-59: fix\_up\_readers()函数

---

```
static void fix_up_readers(struct logger_log *log, size_t len)
{
    size_t old = log->w_off;
    size_t new = logger_offset(old + len);
    struct logger_reader *reader;
    if (clock_interval(old, new, log->head))
        //查询下一个
        log->head = get_next_entry(log, log->head, len);
    //遍历reader链表
    list_for_each_entry(reader, &log->readers, list)
        if (clock_interval(old, new, reader->r_off))
            reader->r_off = get_next_entry(log, reader->r_off, len);
}
```

---

本段代码中通过 `log->w_off` 获取当前写操作的偏移量, `logger_offset(old+len)` 为写入长度为 `len` 的数据后的偏移量。执行 `clock_interval()` 函数进行 `new` 复制时, 将会覆盖 `log->head`, 因此通过 `get_next_entry()` 函数查询下一个节点, 使其作为 `head` 节点。`list_for_each_entry()` 函数执行 `reader` 链表的遍历, 如果 `reader` 在覆盖范围内, 执行对当前 `reader` 位置的调整, 将 `reader` 的位置调到下一个 `log` 数据区。

`fix_up_readers()` 函数只是为原数据提供一个临时存放的区域, 不能彻底避免数据覆盖。因此对于写入前的原数据仍然需要及时读取, 否则仍然会造成数据丢失。

#### (6) logger\_release()

完成了上述的几个操作后, 需要对空间进行释放。操作的代码实现如下所示:

代码清单5-60: logger\_release()函数

---

```
static int logger_release(struct inode *ignored, struct file *file)
{
    if (file->f_mode & FMODE_READ) {
        struct logger_reader *reader = file->private_data;
        list_del(&reader->list);
        kfree(reader);
    }
    return 0;
}
```

---

代码中首先判断是否为只读模式, 如果是只读模式, 通过 `file->private_data` 取得其对应的 `logger_reader`, 删除队列并释放即可。由于写操作没有额外分配空间, 因此不需要处理。

至此 Android 系统的 `Logger` 驱动实现完成。

补充介绍 `logger_ioctl()` 函数, 该函数主要用于对一些命令进行操作, 支持的命令操作如下。

- `LOGGER_GET_LOG_BUF_SIZE`: 得到日志环形缓冲区的尺寸。

- `LOGGER_GET_LOG_LEN`: 得到当前日志 buffer 中未被读出的日志长度。
- `LOGGER_GET_NEXT_ENTRY_LEN`: 得到下一条日志长度。
- `LOGGER_FLUSH_LOG`: 清空日志。

这些操作分别对应于 `include/linux/logger.h` 中所定义的宏。

代码清单5-61: `logger.h`中的宏定义

---

```
#define LOGGER_GET_LOG_BUF_SIZE_IO(__LOGGERIO, 1)
#define LOGGER_GET_LOG_LEN_IO(__LOGGERIO, 2)
#define LOGGER_GET_NEXT_ENTRY_LEN_IO(__LOGGERIO, 3)
#define LOGGER_FLUSH_LOG_IO(__LOGGERIO, 4)
```

---

这些操作的具体实现可以参考 `include/linux/logger.c` 文件中的 `logger_ioctl` 函数。

## 第 6 章

# Android 系统相关工具及运行环境

本章主要讲解 Android 系统开发工具的原理与实现机制。

### 6.1 Android 开发工具分类及介绍

工欲善其事，必先利其器。在 Android 开发过程中，各种开发工具是不可或缺的。诸多开发工具可以分成 4 类：应用程序开发工具，框架开发工具，交叉编译工具，内核开发工具。

#### 6.1.1 应用程序开发工具

Android SDK 提供了一系列可帮助开发者设计、创建、测试和发布 Android 应用程序的强大工具，以下是 10 款最常用的开发工具。

##### 1. Eclipse/ADT

虽然 Eclipse 并非唯一可用于开发 Android 应用程序的 Java 开发环境，但它是目前最欢迎的工具，很大程度上是因为它的成本很低（免费）。但最主要的原因还是它与其他 Android 工具的强大组合功能，最典型的表现就是它与 Android Development Tools (ADT) 插件的组合。

##### 2. Android SDK and AVD Manager

这个工具可提供多种重要的功能，它提供不同版本的 Android SDK，以及第三方附件、工具、设备驱动程序和文件。另外它还能管理用来安装模拟器实体的 Android Virtual Device 配置 (AVD)。

##### 3. Android Debug Bridge (ADB)

该工具可将其他工具接入模拟器和设备，除了可以让其他工具（尤其是 Eclipse/ADT 插件）功能生效以外，还可以使用命令行上传或下载文件，安装或卸载程序包，通过进入设备或模拟器的 Shell 环境访问许多其他功能。

##### 4. Dalvik Debug Monitor Server (DDMS)

无论是通过独立应用程序还是 Eclipse perspective 访问 DDMS，DDMS 都能提供检查、

调试、与模拟器及设备实体交互的便利功能。开发者可使用 DDMS 检查运行程序和线程，查找文件系统，搜集堆栈和其他内存信息。通过模拟器，开发者还可以模拟电话接听和发送 SMS 等状态。

### 5. 模拟器和实际移动设备

如果开发者创建完成了一款应用程序，就必须针对自己的目标设备进行测试。可以将模拟器与 AVD 结合在一起，模拟目标移动设备的运行环境，但要想更全面地进行测试，还是需要有一个真正的移动设备。因为模拟器虽然功能强大，但它毕竟不是实际使用的手机，用户也不可能使用模拟器运行应用程序，所以实际移动设备也是测试环节必不可少的工具。

### 6. LogCat

LogCat 是 Android 日志系统的名称，用户可以通过 Eclipse、ADB 读取 LogCat 数据，它可以提供系统中相关事件的诊断信息。开发者可以由此将应用程序的调试和诊断信息发送到 LogCat。

### 7. The Hierarchy Viewer

开发者可通过独立应用程序或者 Eclipse perspective 访问 The Hierarchy Viewer，它的作用是在运行过程中查看程序的 UI 布局，提供了一个图表显示应用程序布局和视图层级的情况，开发者可依此判断程序 UI 布局存在的问题。

### 8. Draw 9-Patch

Draw 9-Patch 可帮助开发者更方便地完成应用程序的图形设计，该工具支持开发者将传统的 PNG 图像文件转化成更具灵活性、更有效地运用于手机应用开发过程的可扩展图像文件。这项工具可以在快速显示效果的环境中简化 9-Patch 文件的创建过程。

### 9. The Monkey Test Tools

它包括 Monkey 试验程序和 MonkeyRunner 工具，这两项工具可用于自动测试应用程序。前者可在强度测试过程中将其中发生的事件随机发送到应用程序中，而后者可使用 Python 脚本通过截屏自动测试和检查相关结果，以此测试应用的稳定性。

### 10. ProGuard

它是典型的 Android 应用开发过程中必不可少的一个环节，为开发者提供了一个发布产品后保护知识产权的有效方法。ProGuard 这种混淆器可用于模糊相关信息，并用无意义的字符序列来替换其中重要内容，使其难以进行逆向工程。通过 ProGuard 可得到更精简的文件，这就意味着网络传输更省时，装载速度更快，占用内存空间更小。

## 6.1.2 框架开发工具

### 1. Spring for Android

Spring for Android 是 Spring 框架在 Android 平台上的扩展，旨在简化 Android 原生应用

的开发流程，提高开发者的工作效率。Spring for Android 可以帮助开发者简化应用与服务器端交互和 Auth 授权验证。

很多 Android 应用都要与服务器进行交互，而现在很多应用服务器都会提供 REST 服务，数据格式一般是 JSON、XML、RSS 等，而使用 Spring for Android 将会大大地方便 Android 应用与服务器端的交互。Spring for Android 能够简化 JSON 的解析工作。截至目前，Spring for Android 支持 3 个 JSON 第三方库（Jackson JSON Processor、Jackson 2.x 和 Google Gson）。另外，Spring for Android 中的 Simple XML Serializer 也可以帮助开发者解析 XML 文件。

现在很多应用都提供开放的 API 服务，Android 应用往往要经过授权才能接入这些服务，而如今大多应用都采用 Auth 授权认证，而使用 Spring for Android 可以帮助开发者快速地进行授权处理。

## 2. GreenDroid

GreenDroid 是一款高效的 Android 开发类库，可以为开发者提供一个更为轻便的 Android 开发环境。

Android 的开放性使得各种应用的 UI 设计基本上丧失了一致性。不管是官方应用还是第三方应用，都选择使用自己的 UI 交互，开发各种非标准的按钮和控件。GreenDroid 可以使开发者的应用与 Android 生态系统保持一致，并试图为开发者打造界面结构与风格一致的开发环境。

GreenDroid 能够很好地利用 Android 框架所提供的功能，帮助开发者提高应用质量，还能允许开发者随时对应用功能进行优化。

XML 作为承载数据的一个重要角色，使得利用它成为 Android 开发中一项重要的技能。GreenDroid 可以把 XML 文件解析到库中，帮助开发者充分利用 XML。

## 3. Ignition

开发者通过使用 Ignition 所提供的即用组件和样板文件的实用类，可以让所开发的 Android 应用快速起步。

Ignition 涵盖的区域包括：

- Widget、Adapter、Dialog 等 UI 组件。
- 允许编写简单却强大的网络代码的 HTTP Wrapper 库。
- 加载远程 Web 图像并进行缓存的类。
- 简单但有效的缓存框架（将对所有对象树做出响应的 HTTP 缓存到内存或硬盘中）。
- Intents、diagnostics 等几个能让 API 级别更容易向后兼容的帮助类。
- 更友好、更强大的 AsyncTask 实现。

Ignition 包括以下 3 个子项目：

- Ignition-core 是一个可以直接编译到 App 中的 Android 库项目。
- Ignition-support 是一个标准的 Java 库项目，被部署为一个普通的 JAR，包含了大部



分实用工具类。开发者可以独立使用该工程的核心模块。

- Ignition-location 是一个可以直接编译到应用程序中的 Android AspectJ 库项目。能够让定位应用在不需要 Activity 位置更新处理的情况下获取最新的位置信息。

#### 4. DroidParts

DroidParts 是 Android 开发中一组常用的开发组件，可以给开发者带来许多意想不到的便利。DroidParts 主要包括以下几个方面：

- DI——DroidParts 在注入 Views、Services 方面真正做到了“开箱即用”效果，并且自定义依赖关系可以定义在 DependencyProvider 类中。
- ORM——借助于 Cursors 和 Fluent API, DroidParts 可以帮助开发者实现高效模型持久化。
- JSON——DroidParts 中含有简单的 JSON 生成和解析器，实现了对嵌套对象的高效处理。
- ImageFetcher——可以将图片异步加载到 ImageViews 中，同时支持淡入和淡出效果。
- RESTClient——可以向服务器发送各种 HTTP 请求（用户也可以自定义请求方式和提交 JSON 对象），并显示服务器响应。

此外，DroidParts 改进了 AsyncTask 和 IntentService，支持应用的异常处理和结果报告。

### 6.1.3 交叉编译工具

Android 所用的 Toolchain（即交叉编译工具链）可从下面的网址下载：<http://android.kernel.org/pub/android-toolchain-20081019.tar.bz2>。

如果下载了完整的 Android 项目的源代码，则可以在 /prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin 目录下找到交叉编译工具，比如 Android 所用的 arm-eabi-gcc-4.2.1。Android 并没有采用 glibc 作为 C 库，而是采用了 Google 自己开发的 Bionic Libc，它的官方 Toolchain 也是基于 Bionic Libc 而并非 glibc 的。这使得使用或移植其他 Toolchain 来用于 Android 要比较麻烦：在 Google 公布用于 Android 的官方 Toolchain 之前，多数的 Android 爱好者使用的 Toolchain 是在：[http://www.codesourcery.com/gnu\\_toolchains/arm/download.html](http://www.codesourcery.com/gnu_toolchains/arm/download.html) 网页

下载得到的一个通用的 Toolchain，用它来编译和移植 Android 的 Linux 内核是可行的，因为内核并不需要 C 库，但是开发 Android 的应用程序时，直接采用或者移植其他的 Toolchain 都比较麻烦，其他 Toolchain 编译的应用程序只能采用静态编译的方式才能运行于 Android 模拟器中，这显然是实际开发中所不能接受的方式。目前尚没有成功地移植其他交叉编译器来编译 Android 应用程序的资料。

### 6.1.4 内核开发工具

研究 Android，尤其是 Android 系统核心或者是驱动的开发，首先需要做的就是本地克隆并建立一套 Android 版本库管理机制。

Android 使用 Git 作为代码管理工具，开发了 Gerrit 进行代码审核，以便更好地对代码进行集中式管理，还开发了 Repo 命令行工具，对 Git 部分命令封装，将 100 多个 Git 库有效地进行组织。但要想克隆和管理这么多 Git 库，可不是一件简单的事情。

## 6.2 Dalvik 虚拟机

顾名思义，虚拟机并非一台真实的机器，通常是在一台实际机器上添加虚拟化软件实现的一种功能，它拥有的资源与实际机器不同，并且进程的指令集也不同于真实机器。将一台真实机器变为虚拟机通常需要两个步骤：将底层机器的资源映射成为虚拟机的资源；用真实机器的指令集或者系统调用执行虚拟机定义的指令集或者系统调用。

### 6.2.1 概述

Dalvik 虚拟机是 Google 公司为 Android 平台设计的虚拟机。Android 平台主要基于移动设备的应用，而移动设备的特点是较少的 RAM、低频率的 CPU、慢速的闪存以及使用电量有限的电池。因此为了加快程序的运行速度，减少应用程序的占用空间，Dalvik 采用寄存器架构，而不像大部分虚拟机那样基于栈的架构。

#### 1. Dalvik 虚拟机的主要功能

Dalvik 虚拟机主要完成对象生命周期的管理、堆栈的管理、线程管理、安全和异常的管理，以及垃圾回收等重要功能，它是一个面向 Linux、为嵌入式操作系统设计的虚拟机。Dalvik 充分利用 Linux 进程管理的特点，进行了面向对象的设计，使其可以同时运行多个进程，即在 Android 上能够同时运行多个程序，并且每个程序对应一个独立的 Dalvik 进程。同时，为了达到优化的目的，底层很多操作都直接与系统内核有关，或者直接调用内核接口，这使得移植 Dalvik 的工作量变大。

由于 Android 的核心系统服务依赖于 Linux 2.6 内核，如安全性、内存管理、进程管理、网络协议栈和驱动模型，Linux 内核也同时作为硬件和软件栈之间的抽象层，因此 Dalvik 虚拟机从层次结构上与程序库位于同一层。该库包含了 Java 编程语言核心库的大多数功能，每个程序都是在 Dalvik 虚拟机上执行的，如果这个程序要使用程序库中的内容，则需要 Dalvik 调用本地程序接口，执行程序库中的函数。虚拟机是 Java 运行的基础，其将 Java 中的文件转换成 dex 格式文件，具体步骤如图 6-1 所示。



图 6-1 应用程序在 Dalvik 虚拟机中的执行流程

Dalvik 虚拟机的输入是已经编译好的 class 文件或者 jar 文件, Dalvik 虚拟机首先使用 dx 工具将文件转换为 dex 文件, 在这个过程中 Java 指令被翻译成 Dalvik 指令, 文件占用的空间会显著缩小。接下来 Dexopt 工具对 dex 格式文件进行验证和优化, 这个过程中提前计算的数据会加入文件头中。文件下一步加载到内存中, 在运行前进行错误指令和语法准确性的验证。最后是优化和解释, 包括内联函数替换等。该操作完成后生成目标机器码, 交给硬件设备执行。

在 Android 源码中, Dalvik 虚拟机位于 dalvik/ 目录下, 其中 dalvik/vm 是虚拟机的实现部分, 编译结果为 libdvm.so; dalvik/libdex 的编译结果为 libdex.a, 静态库作为 dex 工具库; dalvik/dexdump 是 dex 文件的反编译工具; 虚拟机的可执行程序位于 dalvik/ dalvikvm 中, 编译结果为 dalvikvm 可执行程序。

一个应用转换成 Dalvik 虚拟机可以执行的 dex 文件后, 由解释器根据指令集对 Dalvik 字节码进行解释、执行。最后由 dalvik/vm/Dvm.mk 中的 dvm\_arch 来选择编译的目标机体系结构, 代码如 6-1 所示

代码清单6-1: dalvik/vm/Dvm.mk片段

---

```

ifeq ($(dvm_arch),arm)
    #dvm_arch_variant := armv7-a
    #LOCAL_CFLAGS += -march=armv7-a -mfloat-abi=softfp -mfpu=vfp
    LOCAL_CFLAGS += -Werror
    MTERP_ARCH_KNOWN := true
    # Select architecture-specific sources (armv5te, armv7-a, etc.)
    LOCAL_SRC_FILES += \
        arch/arm/CallOldABI.S \
        arch/arm/CallEABI.S \
        arch/arm/HintsEABI.cpp \
        mterp/out/InterpC-$(dvm_arch_variant).cpp.arm \
        mterp/out/InterpAsm-$(dvm_arch_variant).S
ifeq ($(WITH_JIT),true)
    LOCAL_SRC_FILES += \
        compiler/codegen/RallocUtil.cpp \
        compiler/codegen/arm/$(dvm_arch_variant)/Codegen.cpp \
        compiler/codegen/arm/$(dvm_arch_variant)/CallingConvention.S \
        compiler/codegen/arm/Assemble.cpp \
        compiler/codegen/arm/ArchUtility.cpp \
        compiler/codegen/arm/LocalOptimizations.cpp \
        compiler/codegen/arm/GlobalOptimizations.cpp \
        compiler/codegen/arm/ArmRallocUtil.cpp \
        compiler/template/out/CompilerTemplateAsm-$(dvm_arch_variant).S
endif
endif
ifeq ($(dvm_arch),x86)
    ifeq ($(dvm_os),linux)

```

```

MTERP_ARCH_KNOWN := true
LOCAL_CFLAGS += -DDVM_JMP_TABLE_MTERP=1
LOCAL_SRC_FILES += \
    arch/${dvm_arch_variant}/Call1386ABI.S \
    arch/${dvm_arch_variant}/Hints386ABI.cpp \
    mterp/out/InterpC-${dvm_arch_variant}.cpp \
    mterp/out/InterpAsm-${dvm_arch_variant}.S
ifeq ($(WITH_JIT),true)
    LOCAL_SRC_FILES += \
        compiler/codegen/x86/Assemble.cpp \
        compiler/codegen/x86/ArchUtility.cpp \
        compiler/codegen/x86/ia32/Codegen.cpp \
        compiler/codegen/x86/ia32/CallingConvention.S \
        compiler/template/out/CompilerTemplateAsm-ia32.S
endif
endif
endif
ifeq ($(dvm_arch),sh)
    MTERP_ARCH_KNOWN := true
    LOCAL_SRC_FILES += \
        arch/sh/CallSH4ABI.S \
        arch/generic/Hints.cpp \
        mterp/out/InterpC-allstubs.cpp \
        mterp/out/InterpAsm-allstubs.S
endif

```

若目标机是 ARM 体系结构，则使用 ARM 汇编语言及其相关的 C 语言；如果目标机是 x86 体系结构，则使用 x86 汇编语言及其相关的 C 语言。有关目标机体系结构的代码位于 dalvik/vm/arch 目录中。

## 2. Dalvik 虚拟机与 Java 虚拟机的区别

Dalvik 虚拟机与 Java 虚拟机除了指令集和类文件格式不同以外，二者拥有差不多的特性，例如，它们都是解释执行，并且支持即时编译（JIT）、垃圾收集（GC）、Java 本地方法调用（JNI）和 Java 远程调试协议（JDWP）等。

二者最显著区别是它们具有不同的类文件格式以及指令集。Dalvik 虚拟机使用的是 dex（Dalvik Executable）格式类文件，而 Java 虚拟机运行的是 Java 字节码。在 Java SE 程序中，Java 类会被编译成一个或者多个字节码的 class 格式类文件，然后打包到 jar 文件，而后 Java 虚拟机会从相应的 dex 文件和 jar 文件中获取相应的字节码。Android 需要将生成的 dex 文件通过一个工具 dx 转换为 dex 文件。

一个 dex 文件可以包含若干个类，而一个 class 文件只包括一个类。由于一个 dex 文件可以包含若干个类，因此它可以将各个类中重复的字符串和其他常数只保存一次，从而节省了空间，适合在内存和处理器速度有限的手机系统中使用。一般来说，包含有相同类的未压缩

dex 文件稍小于一个已经压缩的 jar 文件。

Dalvik 虚拟机使用的指令是基于寄存器的，而 Java 虚拟机使用的指令集是基于堆栈的。基于堆栈和基于寄存器的指令集各有优劣，一般而言，执行同样的功能，前者需要更多的指令（主要是 load 和 store 指令），而后者需要更多的指令空间。需要更多指令意味着要多占用 CPU 时间，而需要更多指令空间意味着指令缓冲（i-cache）更易失效。

综上所述，Dalvik 虚拟机具有如下特点：

- 自定义字节码格式 dex 文件，不兼容现有 Java 字节码格式。
- 运行效率高，代码密度小，节省资源。
- 常量池只使用 32 位的索引。
- 内存限制。
- 默认栈的大小是 12KB（3 页，每页 4KB）。
- 堆默认启动大小是 2MB，默认最大值为 16MB。
- 堆支持最小启动是 1MB，支持的最大值是 1024MB。
- 堆和栈参数可以通过 -Xms 和 -Xmx 更改。

每个 Android 应用都运行在一个 Dalvik 虚拟机实例里，而每个虚拟机实例都是一个独立的进程空间。虚拟机的线程机制、内存分配和管理、互斥等都是依赖底层操作系统来实现的，因而虚拟机可以更多地依赖操作系统的线程调度和管理机制。所有 Android 应用的线程都对应一个 Linux 线程，不同的应用在不同的进程空间里运行，而且对不同来源的应用都由不同的 Linux 用户来运行，这可以最大限度地保护应用的安全和独立运行。由此，每个 Android Dalvik 应用程序都被赋予了一个独立的 Linux PID（app\_\*）。

## 6.2.2 dex 文件

### 1. dex 文件格式

Dalvik 可执行文件由 Dalvik 虚拟机编译，并且被压缩成 apk（Android Package）文件放在设备上。dex 文件是由 class 文件通过 Android SDK 中的 dx 工具翻译而来。一个 dex 文件可以分为以下 7 个部分：

- 1) header 部分描述 dex 的头信息。
- 2) string\_ids 部分是 string 标识符列表，包含在此 dex 文件中用到的所有 string 标识符，既包括内部命名标识符，也包括对象应用。string\_ids 部分中的标识符按照 string 内容排序，使用 UTF-16 编码方式。
- 3) Type\_ids 部分是类型标识符列表，包含所有类型（类、数组以及原始类型）的标识符。
- 4) Proto\_ids 部分是方法原型标识符列表，文件中使用到的方法原型都要参考此列表，此列表的排序第一索引是返回值类型，第二索引是参数的数量。
- 5) Field\_ids 部分是 field 标识符列表，这个列表的排序第一索引是定义类型，第二索引是字段名称，第三索引是类型。

6) Method\_ids 部分是方法标识符列表, 列表的排序第一索引是定义类型, 第二索引是方法名称, 第三索引是方法原型。

7) Data 部分存放上面所说的所有数据, 不同的部分具有不同的数据存放要求, 所以在必要时需要一些填充字节。

dex 文件可以分成不同的常量池区域, 每个常量池区域存储一种类型数据。在 dex 文件的底部是类定义区域, 一个单独的 dex 文件可以包含多个类定义, 这点与 Java 的 class 文件明显不同。

## 2. dex 文件转换

Android 平台上的一个应用程序首先被编译成为多个 class 文件, 通过 dx 工具, class 文件转化为 Dalvik 字节码, 并以 dex 文件格式安排文件内容。通常 dex 文件中每个类文件都有一个常量池, 这样就会产生不少冗余的信息。而 dex 文件中会将所有的 class 文件内容整合到一个文件中, 将所有类中的常量放入一个单独的常量池中, Dalvik 仅维护这个常量池, 这样既减少整体的文件尺寸和 I/O 次数, 也提高了类的查找速度。此外, Dalvik 虚拟机还增加了对新的操作码的支持; 其文件结构尽量简洁, 使用等长的指令以提高解析速度; 而且尽量扩大只读结构的大小, 以提高跨进程的数据共享。

dex 文件转换的转换过程如图 6-2 所示。

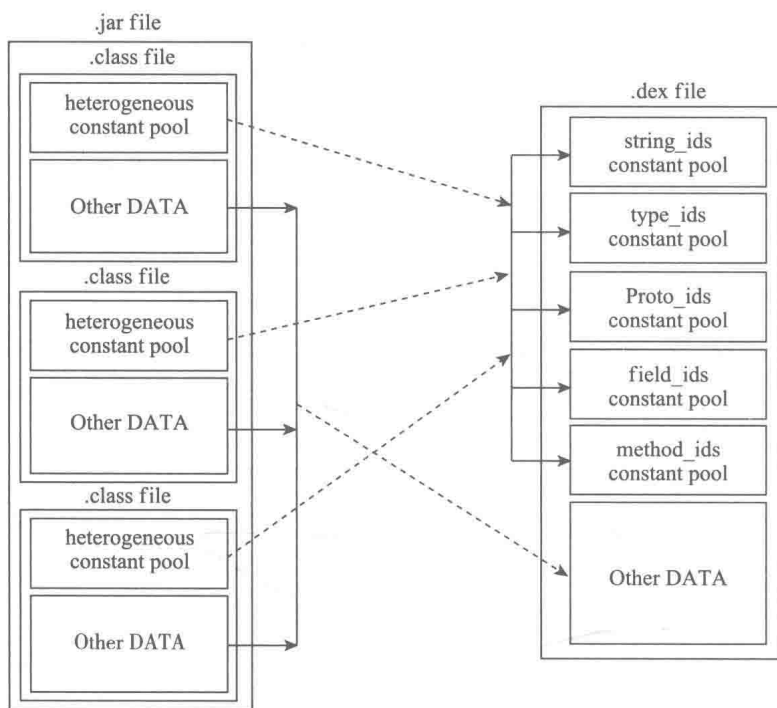


图 6-2 class 文件转换为 dex 文件

虽然 dex 文件的结构很紧凑,但是为进一步提高运行时的性能,可以对 dex 文件进行进一步的优化。主要针对以下几个方面:

- 1) 调整所有字段的字节序 (LITTLE\_ENDIAN) 并对齐结构中的每一个域。
- 2) 验证 dex 文件中的所有类。
- 3) 对特定类进行优化,对方法里的操作码进行优化。

经过优化后的 dex 文件会增大到原文件的 1 ~ 4 倍。对于内置应用,一般在系统编译后生成优化文件 (.odex)。这样,在发布时除了 APK 文件外,还会有一个相应的 .odex 文件。因此,一个内置 Android 应用程序在运行前会经历如图 6-3 所示的编译过程。

对于非内置应用,包含在 APK 文件里的 dex 文件会在运行时被优化,优化后的文件 (dex 格式) 将保存在缓存中,虚拟机会直接执行该文件。如果应用包文件不发生变化, dex 文件不会重新生成。

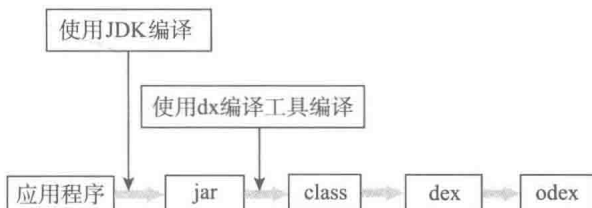


图 6-3 Android 应用程序编译过程

## 6.2.3 Dalvik 内存管理

### 1. 栈架构与寄存器架构

基于栈架构的虚拟机与基于寄存器架构的虚拟机都是针对虚拟机的指令集而言的,区别在于当虚拟机运行时,指令操作数是从栈中获得或者是从寄存器中获得。在基于栈架构的虚拟机中,栈不是实际机器中内存上的内存栈,而是虚拟机模拟出的操作数栈 (Operand Stack)。基于寄存器架构的虚拟机,寄存器一般会被虚拟机映射到实际机器的寄存器上。Java 虚拟机是基于栈架构的虚拟机,而 Dalvik 是基于寄存器架构的虚拟机。

#### (1) 基于栈架构

Java 虚拟机的指令是零地址指令,指令的源和目标都存放在操作数栈上。基于栈架构的虚拟机的指令执行过程:一个指令将常量从常量池中取出,并压到操作栈上;下一条指令将数据从操作数栈中取出,通过算术逻辑单元 (ALU) 执行,运算结束后,算术逻辑单元会把数据重新放入操作数栈。

程序计数器 PC 用于记录程序当前执行的位置。Java 程序每个线程都有自己的 PC,PC 以字节为单位记录当前运行位置方法开头的偏移量。每个线程都有一个 Java 栈,用于记录 Java 方法调用的活动记录 (Activation Record)。Java 栈帧为单位线程的运行状态,每调用一个方法就分配一个新的栈帧压入 Java 栈顶,每次从一个方法返回则弹出并撤销相应的栈帧。

每个栈帧中包括局部变量区、操作数栈以及其他的一些信息。局部变量区用于存储方法的参数与局部变量。每个方法所需要的局部变量区与求值栈大小都能够在编译时确定,并且记录在 class 文件里。

## (2) 基于寄存器架构

Dalvik 虚拟机指令以两个字节为单位，每个线程都有自己的 PC 和调用栈，方法调用的活动记录以帧为单位保存在调用栈上。PC 记录的偏移量是以 16 位为单位而不是以字节为单位的。与 Java 虚拟机不同，Dalvik 虚拟机指令的操作数是一组虚拟机寄存器。

在 Dalvik 虚拟机中常用的虚拟寄存器是 V0 ~ V15，ARM9 也是 16 个寄存器，这样所有的虚拟寄存器都可以映射到硬件寄存器上。也有少数指令可以访问 V0 ~ V255 范围内的 256 个虚拟寄存器，每个方法所需要的虚拟寄存器在编译时就可以确定，并且存到 dex 文件中。

Dalvik 虚拟机里有些很重要的辅助数据经常被访问，这些数据也放在实际机器的寄存器里。

## 2. 内存管理

Dalvik 虚拟机的初始化由 dalvik/vm/init.c 中的 dvmStartup() 函数完成，初始化操作由 dvmGcStartup() 函数开始。dvmGcStartup() 函数代码如下所示：

代码清单6-2: dvmGcStartup()函数

---

```
bool dvmGcStartup()
{
    dvmInitMutex(&gDvm.gcHeapLock);
    pthread_cond_init(&gDvm.gcHeapCond, NULL);
    return dvmHeapStartup();
}
```

---

其中，dvmInitMutex() 函数用于初始化一个 Mutex，它调用的 dvmHeapStartup() 函数代码如下：

代码清单6-3: dvmHeapStartup()函数

---

```
bool dvmHeapStartup()
{
    GcHeap *gcHeap;

    if (gDvm.heapGrowthLimit == 0) {
        gDvm.heapGrowthLimit = gDvm.heapMaximumSize;
    }

    gcHeap = dvmHeapSourceStartup(gDvm.heapStartingSize,
                                   gDvm.heapMaximumSize,
                                   gDvm.heapGrowthLimit);

    if (gcHeap == NULL) {
        return false;
    }
    gcHeap->ddmHpifWhen = 0;
```

---



```

gcHeap->ddmHpSgWhen = 0;
gcHeap->ddmHpSgWhat = 0;
gcHeap->ddmNhSgWhen = 0;
gcHeap->ddmNhSgWhat = 0;
gDvm.gcHeap = gcHeap;

/* Set up the lists we'll use for cleared reference objects.
 */
gcHeap->clearedReferences = NULL;

if (!dvmCardTableStartup(gDvm.heapMaximumSize)) {
    LOGE_HEAP("card table startup failed.");
    return false;
}

return true;
}

```

函数中，gDvm.heapGrowthLimit 实现对 Headmemory 大小的配置，Android 的默认值是 16M；gDvm.gcHeap 实现将分配好的 Heap source 指定给 gDvm.gcHeap，为后续动作做准备；设定一个 list 来记录回收的“参考”对象。

dvmHeapStartup() 函数中的 gcHeap 由 dvmHeapSourceStartup() 函数创建，代码如下：

代码清单6-4：dvmHeapSourceStartup()函数

```

GcHeap* dvmHeapSourceStartup(size_t startSize, size_t maximumSize,
                             size_t growthLimit)
{
    GcHeap *gcHeap;
    HeapSource *hs;
    mspace msp;

    .....

    msp = createMspace(base, startSize, maximumSize);
    if (msp == NULL) {
        goto fail;
    }
    gcHeap = (GcHeap *)malloc(sizeof(*gcHeap));
    if (gcHeap == NULL) {
        LOGE_HEAP("Can't allocate heap descriptor");
        goto fail;
    }
    memset(gcHeap, 0, sizeof(*gcHeap));
    hs = (HeapSource *)malloc(sizeof(*hs));

    .....

    if (!addInitialHeap(hs, msp, growthLimit)) {

```

```

        LOGE_HEAP("Can't add initial heap");
        goto fail;
    }

    .....

    gcHeap->heapSource = hs;

    gHs = hs;
    return gcHeap;

    .....
}

```

本段代码中，createMspace() 函数建立一个用来管理存储器的机制 msp；msp 其实是 dlmalloc 中用来管理和设定可分配范围的；createMspace() 函数传入的参数 startSize 和 maximumSize 用于设定分配空间的大小。创建 msp 后调用 addInitialHeap() 函数，加入 HeapSource 中。

具体的空间分配函数由 HeapSource.c 中的 dvmHeapSourceAlloc() 和 dvmHeapSourceAllocAndGrow() 函数来实现，由 msp 分配空间，调用 countAllocation() 进行标记，使得 GC 可以进行回收。

在 heap.c 中分配空间由 dvmMalloc() 函数中的 tryMalloc() 函数实现，代码如下：

代码清单6-5: dvmMalloc()函数

```

void* dvmMalloc(size_t size, int flags)
{
    void *ptr;
    .....

    dvmLockHeap();
    ptr = tryMalloc(size);

    .....
}

```

执行流程为：tryMalloc() 函数调用 gcForMalloc() 函数，由 gcForMalloc() 函数继续调用 dvmCollectGarbageInternal() 函数。其中 dvmCollectGarbageInternal() 函数代码如下：

代码清单6-6: dvmCollectGarbageInternal()函数

```

void dvmCollectGarbageInternal(const GcSpec* spec)
{
    dvmSuspendAllThreads(SUSPEND_FOR_GC);

    dvmMethodTraceGCBegin();
    if (!dvmHeapBeginMarkStep(spec->isPartial)) {
        LOGE_HEAP("dvmHeapBeginMarkStep failed; aborting");
    }
}

```

```

        dvmAbort();
    }

    dvmHeapMarkRootSet();

    assert(gcHeap->softReferences == NULL);
    assert(gcHeap->weakReferences == NULL);
    assert(gcHeap->finalizerReferences == NULL);
    assert(gcHeap->phantomReferences == NULL);
    assert(gcHeap->clearedReferences == NULL);

    dvmHeapScanMarkedObjects();
    .....
    dvmHeapSweepSystemWeaks();

    dvmHeapSourceSwapBitmaps();

    dvmHeapSweepUnmarkedObjects(spec->isPartial, spec->isConcurrent,
                                &numObjectsFreed, &numBytesFreed);

    dvmHeapFinishMarkStep();

    dvmHeapSourceGrowForUtilization();
    dvmMethodTraceGCEnd();
    dvmResumeAllThreads(SUSPEND_FOR_GC);
}

```

LOGE\_HEAP() 函数设置了 MarkStep 标记, 用来作为追踪 HeapBitmap 中的 bitmark; dvmHeapMarkRootSet() 函数标记所有的 root object; assert() 函数将 heap memory 中的所有 reference 全部都设为 NULL, 使其对象失去参考对象; dvmHeapScanMarkedObjects() 函数对所有已标记的 object 进行扫描, 若 reference 到其他未标记的 object, 加入标记并且在 bitmap memory 中的位作设定, 之后将 object 压入 Mark Stack 中; dvmHeapSweepSystemWeaks() 函数将所有 Heap memory 中没被标记的系统对象全部回收; dvmHeapSourceSwapBitmaps() 函数对于活动对象如果在 mark bitmap 中有一个位设定, 就将 live bitmap 跟 mark bitmap 互换, 同时将有标记与没标记的活动对象作分类; dvmHeapSweepUnmarkedObjects() 函数的作用是释放无标记对象; dvmHeapFinishMarkStep() 函数将 bitmap memory 里的位全部重新初始化, 且删除 Mark Stack; dvmHeapSourceGrowForUtilization() 函数利用 Heap Source 重新调整 Heap memory 的大小。

#### 6.2.4 Dalvik 编译器

对于任何虚拟机, 编译器是核心部分, 所有的 Java 代码都通过编译器编译执行。编译器的速度影响虚拟机的性能。在 Android 中, 有关 Dalvik 编译器接口的部分位于 dalvik/vm/

interp 目录下，Andriod 提供 C 语言版和各种汇编语言版本，位于 dalvik/vm/mterp 目录下。如图 6-4 所示。




























	armv5te	2013/11/20 23:30	文件夹
	armv6	2013/11/20 23:30	文件夹
	armv6t2	2013/11/20 23:30	文件夹
	armv7-a	2013/11/20 23:30	文件夹
	arm-vfp	2013/11/20 23:30	文件夹
	c	2013/11/20 23:30	文件夹
	common	2013/11/20 23:30	文件夹
	cstubs	2013/11/20 23:30	文件夹
	out	2013/11/20 23:30	文件夹
	portable	2013/11/20 23:30	文件夹
	x86	2013/11/20 23:30	文件夹
	x86-atom	2013/11/20 23:30	文件夹
	config-allstubs	2013/11/20 23:30	文件
	config-armv5te	2013/11/20 23:30	文件
	config-armv5te-vfp	2013/11/20 23:30	文件
	config-armv7-a	2013/11/20 23:30	文件
	config-armv7-a-neon	2013/11/20 23:30	文件
	config-portable	2013/11/20 23:30	文件
	config-x86	2013/11/20 23:30	文件
	config-x86-atom	2013/11/20 23:30	文件
	gen-mterp.py	2013/11/20 23:30	PY 文件
	Makefile-mterp	2013/11/20 23:30	文件
	Mterp.cpp	2013/11/20 23:30	CPP 文件
	Mterp.h	2013/11/20 23:30	H 文件
	NOTES.txt	2013/11/20 23:30	文本文档
	README.txt	2013/11/20 23:30	文本文档
	rebuild.sh	2013/11/20 23:30	SH 文件

图 6-4 dalvik/vm/mterp 目录

图 6-4 中每个目录对应一种不同的实现，c 子目录为 C 语言版本的实现，其他则对应不同的 CPU 的汇编语言实现。里面都是源代码，需要通过该目录下的 Python 工具即 gen-mterp.py 文件生成目标代码才能进行编译，生成文件都放在 out 目录下。

gen-mterp.py 使用的命令格式如下：

```
gen-mterp.py target-arch output-dir
```

其中 target-arch 为目标体系平台，output-dir 为输出文件的目录。

负责编译器工作流程的是 cstubs 目录。编译器通常是循环执行，入口函数调用处理程序执行的第一条指令，其后的每条指令执行时均会通过函数指针调用处理程序引出下

一条指令。每个处理程序都有一个粘合参数，其中包含了各种参数类型。编译器的入口由 `dvmInterpret()` 函数实现，函数位于 `\dalvik\vm\interp\interp.c` 文件中。

## 6.3 JNI

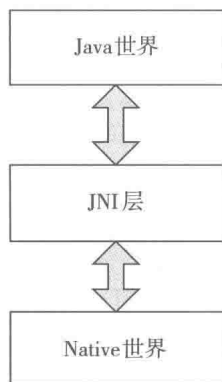
### 6.3.1 概述

JNI 是 Java Native Interface 的缩写，中文含义为“Java 本地调用”。通俗地说，JNI 是一种技术，通过这种技术可以做到以下两点：

- Java 程序中的函数可以调用 Native 语言写的函数，Native 一般指的是 C/C++ 编写的函数。
- Native 程序中的函数可以调用 Java 层的函数，也就是说在 C/C++ 程序中可以调用 Java 的函数。

Native 又是做什么的呢？众所周知，程序开发离不开具体的硬件平台，但为了保证平台无关，引入虚拟机。Java 世界的虚拟机是用 Native 语言写的。

早在 Java 语言诞生前，很多程序和模块功能都是用 Native 语言编写实现的。Java 一经推出便得到迅速发展，为了避免进行重复性的工作，Java 与 Native 合作进行功能的调用，实现两者合作的桥梁便是 JNI，即在 Java 中通过 JNI 技术可以直接使用 Native 模块的相关功能。三者之间的关系如图 6-5 所示。



由图 6-5 可以看出，JNI 是连接 Java 世界与 Native 世界的重要环节。图 6-5 三者关系

### 6.3.2 JNI 的架构

Android 应用程序框架层中的 API 大部分调用了 Native 方法，这些方法也都由本地代码来实现，然后注册到系统中。注册过程需要使用 Dalvik 中的一个工具库 `nativehelper`，该库主要用于注册 Java 本地调用的函数（即通过 JNI 方式向上层提供接口）。

实现 `nativehelper` 库的源码位于 `\dalvik\libnativehelper\` 目录中，最终将被编译为 `libnativehelper.so` 动态链接库。编译设置动态链接库的主要原因在于使得虚拟机可以随时进行库的加载和调用。通常在使用 JNI 方式时需要使用该库，其主要头文件如下：

- `\dalvik\libnativehelper\include\nativehelper\jni.h`（基于 JNI 标准的头文件）
- `\dalvik\libnativehelper\include\nativehelper\JNIHelp.h`（提供 JNI 注册功能的头文件）

具体的 JNI 代码的实现主要位于 `\frameworks\base\core\jni\` 目录中，实现了 Android API 的大部分功能，最终将被编译为 `libandroid runtime.so` 动态链接库，放置在目标文件系统的 `system\lib\` 目录中。

另外，也有部分 JNI 实现不在该目录中，而是位于每个模块的本地实现目录的 jni 目录中，包括硬件抽象层、Android 扩展库、功能库等都有对应的 JNI 实现目录。比如媒体部分的 JNI 实现位于 frameworks\base\media\jni\ 目录中，最终将被编译为 libmedia\_jni.so 动态链接库。

6.3.3 JNI 的实现方式

1. 概述

JNI 的实现实际上就是将 Java 代码中声明的原生方法在本地实现，并注册到系统中。原生方法由系统中各个功能模块进行实现，本节主要介绍原生方法在本系统中的注册，并使得 Java 层能够进行调用。由上节内容可知，注册即将原生方法写入 Native 层。从 nativehelper 库的 jni.h 中可以看到如代码清单 6-7 所示的结构体，其中，JNINativeMethod 表示一个原生方法。

代码清单6-7：JNINativeMethod定义

```
typedef struct {
    const char* name;
    const char* signature;
    void* fnPtr;
} JNINativeMethod;
```

其中 name 表示 JNI 函数的名称，对应于 Java 程序中声明的原函数名称；fnPtr 表示函数中的函数指针；signature 表示声明函数的参数和返回值，如图 6-6 所示。

JNI 的类型通过 typedef 定义实现，而对应的字母则通过 typedef union jvalue 进行定义。

2. 实例分析

本节通过媒体扫描器 MediaScanner 分析 JNI 的实现过程。MediaScanner 纵跨 3 个层次，Java 层的 MediaScanner 通过 JNI 层的 libmedia\_jni.so 调用 Native 层的 libmedia.so 库，完成实际的功能。MediaScanner 源码实现位于 frameworks\base\media\java\android\media\MediaScanner.java。

Java 的类型	JNI 的类型	对应的字母
boolean	jboolean	Z
byte	jbyte	B
char	jchar	C
short	jshort	S
int	jint	I
long	jlong	J
float	jfloat	F
double	jdouble	D
object	jobject	L
void	jvoid	V

图 6-6 JNI 函数参数和返回值

代码清单6-8：MediaScanner源码片段

```
public class MediaScanner
{
    static {
        System.loadLibrary("media_jni");
    }
}
```

```

        native_init();
    }
    .....
    public void scanDirectories(String[] directories, String volumeName) {
        .....
    }
    .....
    private native void processDirectory(String path, MediaScannerClient client);
    private native void processFile(String path, String mimeType,
MediaScannerClient client);
    public native void setLocale(String locale);

    public native byte[] extractAlbumArt(FileDescriptor fd);

    private static native final void native_init();
    private native final void native_setup();
    private native final void native_finalize();
    .....
}

```

本段代码主要完成两个功能：加载 JNI 库和 Java native 函数的声明。static 语句中，加载库 media\_jni 由函数 System.loadLibrary() 实现，native\_init() 实现将 Native 对象的指针保存到 Java 对象中；public void scanDirectories() 完成扫描文件的设置。由于 native 是 Java 的关键字，因此后续代码均为 Java 层中 native 函数的声明。

JNI 库的加载通常不会受到限制。加载方式可以进行套用，如代码 6-8 所示。在 static 语句中完成，调用 System.loadLibrary() 函数实现，函数的参数就是动态库的名字。系统会自动根据 vBulletin 不同的平台拓展成为真实的动态库文件名。

上段代码中，通过在函数前添加 Java 关键字 native 表示函数由 JNI 层实现。

因此，开发者使用 JNI 时，只需完成对应库的加载和 native 关键字函数的声明即可。

## 6.4 Boot Loader

### 6.4.1 概述

U-Boot，全称为 Universal Boot Loader，即通用 Boot Loader，是在操作系统内核运行之前运行的一段小程序。通过这段程序，可以进行硬件设备的初始化、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

通常，Boot Loader 是严重依赖于硬件而实现的，建立一个通用的 Boot Loader 几乎是不可能的。每种不同的 CPU 体系结构都有不同的 Boot Loader。有些 Boot Loader 也支持多种体系结构的 CPU，比如 U-Boot 就同时支持 ARM 体系结构和 MIPS 体系结构。

除了依赖于 CPU 的体系结构外, Boot Loader 也依赖于具体的嵌入式板级设备的配置。对于两块不同的嵌入式板,即使是基于同一种 CPU 而构建的, Boot Loader 程序在移植过程中也都需要修改源程序,才能运行在另一块板子上。

Boot Loader 的功能大致可以分为如下几部分:

1) 对 PLL 时钟进行初始化。处理器在启动时,为了获得更好的设备兼容性,其工作频率都很低,在 Boot Loader 程序中会提高处理器的时钟频率,以加快运行速度,速度一旦调好就不会发生改变。随着 PLL 时钟频率的增高,运行速度的加快,系统耗电量也随着增加,因此对于处理器所支持的“节电模式”,也会使得 PLL 时钟频率发生变化。

2) 初始化 SDRAM 内存控制器。Boot Loader 自身也需要用到内存,大多 Boot Loader 都会将自己加载到内存中。内存的配置一般是包括行地址和列地址的配置以及自动刷新频率的配置。

3) 初始化中断控制器和中断服务程序。

4) 初始化各地址空间的片选地址寄存器和读写时序。

5) 初始化堆栈寄存器。例如,在 x86 中初始化 ESP 寄存器,在 PowerPC 中需要初始化 r1 寄存器等。

6) Boot Loader 中需要访问的其他硬件设备进行初始化。例如,作为控制台(console)的串口就需要在 Boot Loader 中进行相应的初始化,才能够接受用户的命令,响应用户的请求。由此可见 Boot Loader 中存在着一定的命令处理程序。

7) 将 Boot Loader 加载到内存的过程中,如果需要解压,还得完成解压操作。

8) 加载需要运行的应用程序,并最终运行被加载的应用程序。

## 6.4.2 Boot Loader 的操作模式

大多数 Boot Loader 都包含两种不同的操作模式:“启动加载”模式和“下载”模式,这种区别仅对于开发人员才有意义。从用户的角度看, Boot Loader 的作用就是用来加载操作系统,而并不存在所谓的启动加载模式与下载模式。

1) 启动加载(Boot loading)模式:这种模式也称为“自主”(Autonomous)模式,即 Boot Loader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行,整个过程无需用户介入。这种模式是 Boot Loader 的正常工作模式,因此在嵌入式产品发布的时候, Boot Loader 都工作在这种模式下。

2) 下载(Downloading)模式:在这种模式下,目标机上的 Boot Loader 将通过串口连接或网络连接等通信手段从主机(Host)下载文件,例如下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 Boot Loader 保存到目标机的 RAM 中,然后再被 Boot Loader 写到目标机上的 Flash 类固态存储设备中。

Boot Loader 的这种模式通常在第一次安装内核与根文件系统时被使用,以后的系统更新也会使用。这种模式下的 Boot Loader 通常都会向它的终端用户提供一个简单的命令行接口。



像 Blob 或 U-Boot 等这样功能强大的 Boot Loader 通常同时支持这两种工作模式，而且允许用户在这两种工作模式之间进行切换。例如 Blob 在启动时处于正常的启动加载模式，但是它会延时 10 秒等待终端用户按下任意键而将 Blob 切换到下载模式。如果在 10 秒内用户没有按键，则 Blob 继续启动 Linux 内核。

### 6.4.3 启动过程

U-Boot 启动内核的过程可以分为两个阶段，两个阶段的功能如下。

第一阶段的功能有：硬件设备初始化，加载 U-Boot 第二阶段代码到 RAM 空间，设置栈（包括设置指针和置空 BSS 段），跳转到第二阶段代码入口，如图 6-7 所示。

文件位于 `cpu/arm920t/start.S` 和 `board/samsung/mini2440/lowlevel_init.S`。

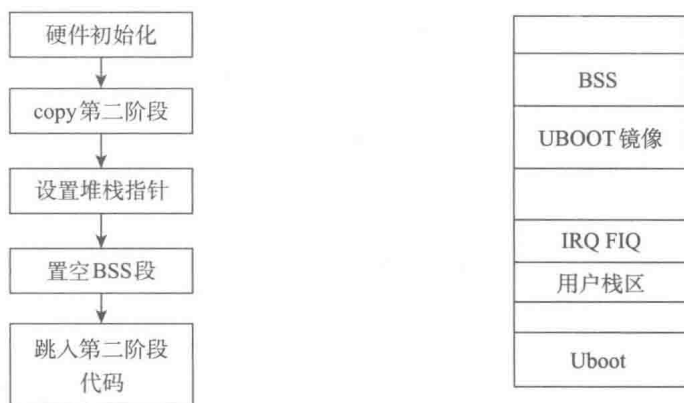


图 6-7 U-Boot 第一阶段流程

第二阶段的启动流程如图 6-8 所示。

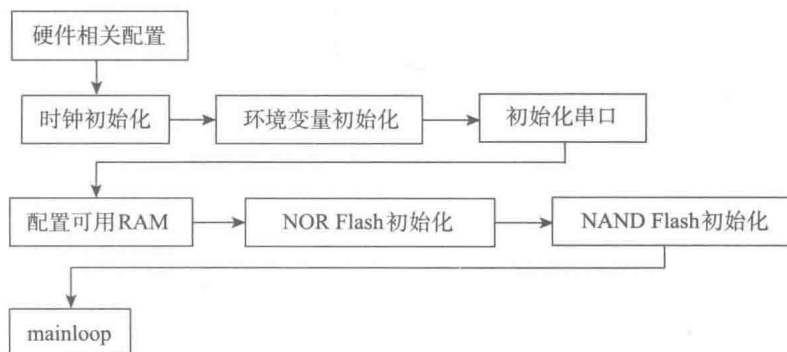


图 6-8 U-Boot 第二阶段流程

调用的函数分别为硬件配置 `board_init`，时钟初始化 `timer_init`，环境变量初始化 `env_init`，串口初始化 `init_baudrate`、`serial_init`、`console_init_f`，配置 RMA `dram_init`，并且使用 `display_dram_init` 显示 RAM 的大小，NOR Flash 初始化 `flash_init`，NAND Flash 初始化 `nand_init`。

## 6.5 busybox 的使用

Google 没有使用传统的标准的 glibc 来作为 C 库使用，而是重新开发了 bionic 来作为其 Android 上的 C 库使用。同时，Google 官方公布的 Toolchain 工具链也只是基于 bionic C 库而开发的。多数有关 Android busybox 方面的移植内容一般都是基于 CodeSourcery 的交叉编译工具链，来完成相应的编译开发工作的，通常来说这种方式不太合理。

与 glibc 相比，Bionic Libc 有如下特点：

- 采用 BSD License，而不是 glibc 的 GPL License。
- 大小只有大约 200k，比 glibc 差不多小一半，且比 glibc 更快。
- 实现了一个更小、更快的 pthread。
- 提供了一些 Android 所需要的重要函数，如 getprop、tprop 等。
- 不完全支持 POSIX 标准，比如 C++ exceptions、wide chars 等。
- 不提供 libthread\_db 和 libm 的实现。

有些基于静态链接编译的 busybox 命令在 Android 上无法使用或使用功能不完全，例如 ping 命令。对 busybox 的移植策略并不是静态地链接到标准的 glibc 库上，而是通过 Android 的 bionic C 库和 Toolchain 交叉编译工具链进行移植。

基于 CyanogenMod 的 busybox 源代码移植工作如下：

- 1) 下载 CM 的 busybox 源代码，将其解压并放入 Android 源代码的 external/busybox 目录中。
- 2) 添加 stdio.h 头文件到 external/busybox 目录中的下列文件之中，coreutils/df.c、util-linux/mount.c、util-linux/umount.c
- 3) 如果是 froyo 之前的版本，需在 bionic/libc/include/sys/resource.h 中添加下列定义语句：typedef unsigned long rlim\_t。
- 4) 在 Android.mk 中设置 CYANOGEN\_BIONIC 变量开关的值 CYANOGEN\_BIONIC := true。
- 5) 编译 busybox。

```
derek@derek-ThinkPad-Edge: ~/Android/wholeplatform/froyo_blcr$ make ake busybox
# mount -o remount,rw /dev/block/mtdblock0 /system
# cat /sdcard/busybox > /system/xbin/busybox
# mount -o remount,ro /dev/block/mtdblock0 /system
# cd system/xbin
# chmod 755 busybox
# busybox sh
# busybox ls -lh system/xbin/busybox
-rwxr-xr-x    1 root    root      401.4K Jul  8 21:23 busybox
```

由编译移植过程可知，基于 bionic C 库动态移植编译的 busybox（约 400k）比基于 bionic C 库静态链接编译（约 800k）和基于 glibc 静态链接编译的 busybox（约 1.8M）都要小很多，这对于存储空间非常有限的嵌入式系统来说是非常重要的。

## 第 7 章

# Android 驱动程序设计

本章主要讲解 Android 系统中的设备驱动的程序设计。

### 7.1 Android 驱动概述

在以往熟悉的 Windows 系统中，安装主板、光驱、显卡、声卡这些硬件都需要相应的驱动程序，如果需要外接其他硬件设备，也需要安装相应的驱动程序。和 Windows 一样，在安装有 Android 系统的设备上，也需要为一些外部硬件（如蓝牙耳机、摄像头等）安装对应的驱动程序。其实驱动程序是添加到操作系统的一段代码，包含了硬件相关的设备信息和操作接口。用户可以通过设备的操作接口来操作硬件。

Android 是在标准的 Linux 内核基础上开发出来的操作系统，它将系统的驱动分成两层，实现对硬件设备的支持，分别为内核空间的 Linux 内核驱动层和用户空间的 Android 硬件抽象层（HAL）。Android 系统的驱动需要在理解硬件设备工作原理的基础上开发，其主要的工作集中在以下两个方面：

#### （1）Linux 设备驱动程序

驱动程序是硬件和上层软件的接口，在 Android 系统中，需要基本的 LCD 屏幕、触摸屏、键盘等驱动，以及音频、摄像头、电话的 Modem、WiFi、蓝牙等多种设备驱动程序。

#### （2）Android 硬件抽象层

在 Android 中，硬件抽象层工作在用户空间，介于驱动程序和 Android 系统之间。HAL 层向下屏蔽硬件驱动的实现细节，向上提供硬件的访问接口。Android 系统的硬件抽象层通常都有标准的接口定义，在开发过程中，实现这些接口也就给 Android 系统提供了硬件抽象层。

Android 驱动开发相关的结构框图如图 7-1 所示。

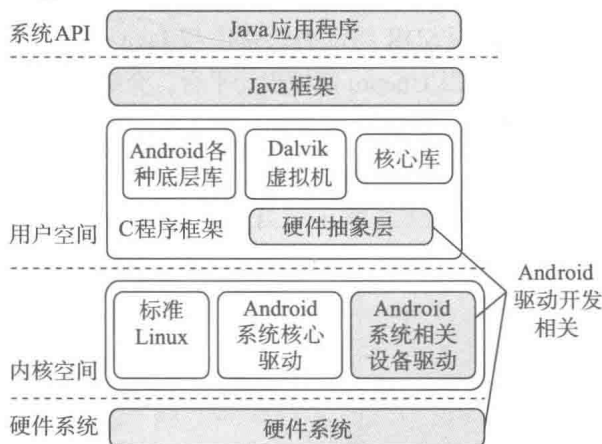


图 7-1 Android 驱动开发相关结构

## 7.2 Android NDK 编程

NDK 是一系列工具的集合，全称为 Android Native Development Kit，这个工具包帮助开发者快速开发 C（或 C++）的动态库或本地应用。

谈到 NDK，就不得不提到 JNI。JNI 为 Java Native Interface 的缩写，中文即为 JAVA 本地调用，它允许 Java 代码与其他语言的代码进行交互，尤其是 C 与 C++。

在应用中，Java 代码、C 代码与 JNI 的关系如图 7-2 所示。

作为一名使用 Java 语言开发 Android 应用程序的人员，需要知道如何通过 JNI 接口调用 C 语言编写的函数库。接下来将介绍 NDK 程序开发，其流程如图 7-3 所示。

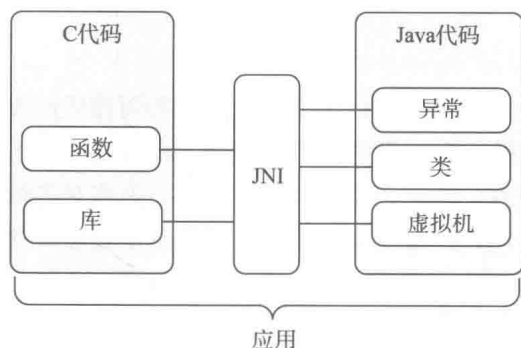


图 7-2 Java 语言、C 语言与 JNI 的关系

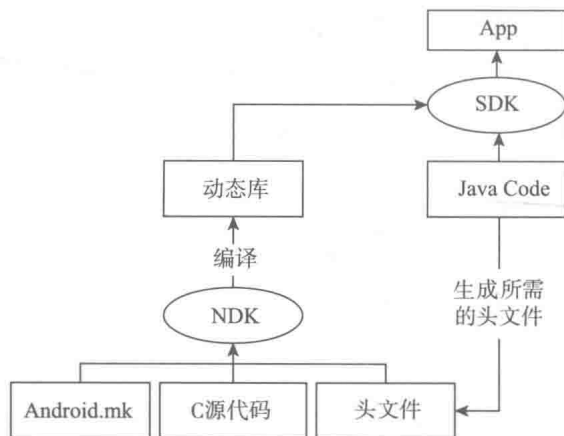


图 7-3 NDK 开发流程图

- 1) 编辑 Java 代码，并编译。
- 2) 在 Java 程序中添加动态库加载的代码，给出 C 函数的定义，并在代码中引用。
- 3) 生成符合 JNI 样式的 C 语言头文件，用工具自动生成。
- 4) 编辑 C 语言代码，并生成 C 共享库（.so 动态链接库）。
- 5) 使用 SDK 将动态链接库与 Java 程序一起打包成 App。

接下来以 Ubuntu 虚拟机为平台，介绍如何安装 NDK 开发环境以及如何使用 NDK 来编程。从网上下载最新的 NDK 开发包，此处以 android-ndk-r6-linux-x86.tar.bz2 为例。

下载地址是 <http://developer.android.com/sdk/ndk/index.html>。将 NDK 开发包 android-ndk-r6-linux-x86.tar.bz2 解压到当前用户的 NDK 目录，如图 7-4 所示。

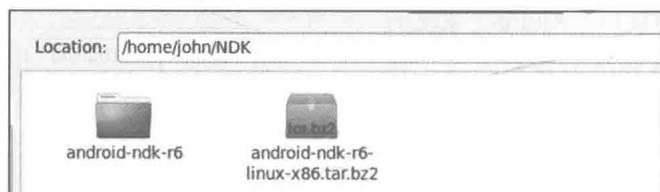


图 7-4 NDK 解压后的目录

解压完成后需要将 NDK 加入环境变量中。在终端的用户目录下，输入 `sudo gedit .bashrc` 后，在打开的文件的最后位置输入如下内容，保存退出后，通过 `source .bashrc` 使配置的环境变量生效。

```
#set NDK env
NDKROOT=~/.NDK/android-ndk-r6
export PATH=$NDKROOT:$PATH
```

下面将以实例的方式详细讲述如何通过 NDK 完成 C 语言动态链接库的生成，再通过 JNI 来调用该动态库返回运算结果。主要工作是：编写 Java 代码，使用 `native` 声明需要本地实现的方法 `add`；通过 `javah -jni` 命令生成带有 JNI 样式的头文件；使用 C 代码实现该函数并编写相应的编译规则文件 `Android.mk`；最后由 `ndk-build` 命令生成动态链接库，并将其打包到应用程序。接下来将描述具体步骤：

### (1) 编写 Java 源代码

在 `src` 的 `com/example/myjni` 目录下新建 Java 源文件。对代码作简要分析：定义整型 `a` 作为保存 `AddNum` 反馈的结果，并最终显示在 `TextView` 上，代码第 16 行的 `add` 方法在使用 NDK 开发的 `.so` 动态链接库中完成。在 `src/com/` 目录下，新建一个 `addnum` 包，Java 源代码中 `native` 声明 `add` 为本方法，比如使用 C/C++ 实现。一般使用 `static` 块进行加载动态库，该块中使用 `System.loadLibrary()` 加载库，`myjni` 为动态库的名字。

代码清单7-1

---

```
package com.example.myjni;
import com.addnum.AddNum;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        int a = new AddNum().add(5, 4); //需要相加的两个数字
        TextView tv1 = new TextView(this);
        tv1.setText("a = " + a);          //在屏幕上显示a的结果
        setContentView(tv1);

    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        //Inflate the menu; this adds items to the action bar if it is
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

```

    }
}

package com.addnum

public class AddNum {
    public native int add(int a, int b);
    static
    {
        //装载lib*.so文件
        System.loadLibrary("myjni");
    }
}

```

## (2) 生成 JNI 头文件

进入 <myjni 工程目录>/bin/classes 目录，并输入命令 `javah -jni com.addnum.AddNum`，生成 JNI 样式的头文件，其中 `com_addnum_AddNum.h` 中含有 JNI 函数的声明。

代码清单7-2

```

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_addnum_AddNum */

#ifndef Include_com_addnum_AddNum
#define Include_com_addnum_AddNum
#ifdef _cplusplus
extern "C" {
#endif
/*
 * Class:      com_addnum_Add_Num
 * Method:     add
 * Signature:  (II)I
 */
JNIEXPORT jint JNICALL Java_com_addnum_AddNum_add
    (JNIEnv *, jobject, jint, jint);

#ifdef _cplusplus
}
#endif
#endif

```

`Java_com_addnum_AddNum_add(JNIEnv *, jobject, jint, jint)` 即是本地方法中需要实现的方法，需要注意保持方法名一致，否则将导致错误。native 对应的函数名要以 `Java_` 开头，后面分别跟着 `com_addnum`（包名）、`AddNum`（类名）、`add`（函数名）。

这里注意到参数 JNIEnv \*, jobject。JNIEnv 结构体指针是 JNI 的核心数据，它包含诸多 JNI 接口函数指针，使开发者可以使用 JNI 所定义的接口。jobject 是调用这个 JNI 函数的 Java 对象，类似 C++ 中的 this 指针。

JNI 编程涉及到上层的 Java 层与下层的 C/C++ 层（本地代码），两种语言之间的数据类型定义是不一样的，JNI 解决了数据类型问题。对应的数据类型关系如图 7-5 所示。

Java 类型	Native 类型	符号属性	字长
boolean	jboolean	无符号	8 位
byte	jbyte	无符号	8 位
char	jchar	无符号	16 位
short	jshort	有符号	16 位
int	jint	有符号	32 位
long	jlong	有符号	64 位
float	jfloat	有符号	32 位
double	jdouble	有符号	64 位

Java 引用类型	Native 类型	Java 引用类型	Native 类型
all objects	jobject	char[]	jcharArray
java.lang.Class 实例	jclass	short[]	jshortArray
java.lang.String 实例	jstring	int[]	jintArray
object[]	jobjectArray	long[]	jlongArray
boolean[]	jbooleanArray	float[]	jfloatArray
byte[]	jbyteArray	double[]	jdoubleArray
Java.lang.Throwable 实例	jthrowable		

图 7-5 Java 类型与 Native 类型的数据转换

(3) 编写 C 代码和 Android.mk

在 myjni 工程目录中新建 jni 文件夹，进入 jni 目录，将 bin/classes/ 目录下生成的头文件复制到该目录下，并编写 C 代码 AddNum.c，直接通过 return a+b 实现加法功能。

代码清单7-3

```
#include "com_addnum_AddNum.h"

JNIEXPORT jint JNICALL Java_com_addnum_AddNum_add
    (JNIEnv *env, jobject thiz, jint a, jint b)
{
    return a + b;
}
```

再增加相应编译规则 Android.mk 文件。

代码清单7-4

---

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

LOCAL_MODULE := myjni
LOCAL_SRC_FILES := AddNum.c

include $(BUILD_SHARED_LIBRARY)
```

---

LOCAL_PATH := \$(call my-dir)	获取当前路径，并保存在LOCAL_PATH中
include \$(CLEAR_VARS)	由编译系统提供，GNU MAKEFILE清除LOCAL_XXX变量，除了LOCAL_PATH以外
LOCAL_MODULE	指定编译出的库名
LOCAL_SRC_FILES	指定编译源文件
include \$(BUILD_SHARED_LIBRARY)	指定编译成动态链接库，BUILD_STATIC_LIBRARY 为静态链接库

---

#### (4) 生成动态链接库

在终端，进入 myjni 工程目录，输入 ndk-build 命令，根据 Android.mk 中规定的编译规则将 AddNum.c 编译成 libmyjni.so。如果编译成功，会在工程中生成 libs 和 obj 目录，在 libs/armebabi 目录中包含 libmyjni.so 动态库，如图 7-6 所示。

```
john@wscec-desktop:~/workspace/myjni$ ndk-build
Compile thumb : myjni <= AddNum.c
SharedLibrary : libmyjni.so
Install       : libmyjni.so => libs/armebabi/libmyjni.so
john@wscec-desktop:~/workspace/myjni$
```

图 7-6 ndk-build 指令编译本地代码

NDK 工具通过 ndk-build 解析 Android.mk 文件，将相应的本地代码 AddNum.c 编译成 libmyjni.so，这个库将被 Java 程序所载入。

#### (5) 运行测试。

打开 AVD 运行该 App，则可以看到通过调用本地方法实现加法功能，输出结果正确，如图 7-7 所示。



图 7-7 NDK 编程结果



## 7.3 Android 系统中的 HAL 层

在 Android 系统中, HAL 层(硬件抽象层)是为保护一些硬件提供商的知识产权而提出的,位于 Linux 内核层之上。HAL 实现了硬件抽象化,将硬件平台的差异隐藏,为上层提供统一的硬件平台,从而加快开发人员在不同的硬件平台上进行代码移植。HAL 层在 Android 中具有重要的位置,通过 Android 系统框架了解其重要地位,如图 7-8 所示。

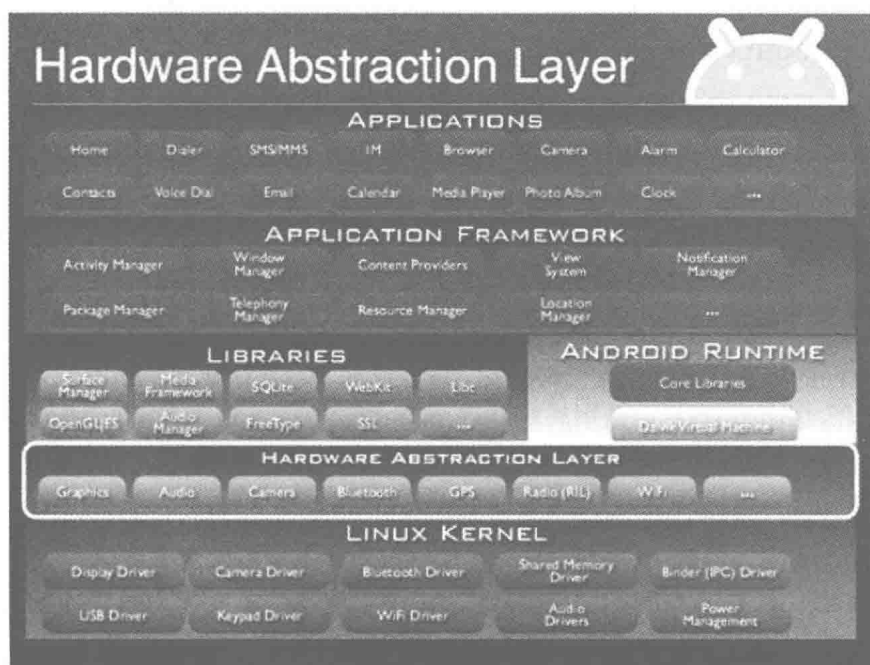


图 7-8 Android 系统框架

就驱动开发的角度而言, HAL 将 Android 框架(Android Framework)与 Linux 内核(Linux Kernel)隔离。主要原因有两方面:由于 Linux 内核要遵循 GPL,根据此许可证,如果对 Linux 内核源码进行改动就必须公开其源码。如果 Android 系统的硬件驱动都在 Linux 的驱动模块中实现,就要将驱动源码完全公开。但是硬件设备商并不愿意将具体的细节公开,此类做法有损其利益。因此,Android 系统源码采用 Apache License 许可,它允许各厂家对 Android 系统源码进行改动但不公开。另一方面是由于 Android 对某些硬件有特殊的要求,并没有标准的 Linux 接口。

Android 的 HAL 层考虑 Linux 系统的设计要求。由于 Linux 对硬件的支持实现不能完全在用户空间,只有内核空间才具有特权对硬件设备进行操作。基于上述因素,Android 驱动将对硬件的支持放在内核空间与用户空间,内核空间使用 Linux 驱动模块,但只提供硬件访问通道,而用户空间则以 HAL 模块的方式,在 HAL 层对硬件细节与参数进行封装,既实现 Linux 系统对代码公开的要求,也保护移动设备厂家各自的利益。

Android 系统中的 HAL 层具有 5 个方面的特点，这些特点也是设计 HAL 层的原则所在：

- 硬件抽象层具有与硬件设备的操作无关性。
- 硬件抽象层与硬件有着密切的相关性。
- 定义的接口要满足一定的硬件操作与系统要求。
- 接口定义较为简单，接口若过多则会增加软件模拟的难度。
- 具有可测试性，能够进行软件硬件测试和系统集成。

### 7.3.1 HAL\_legacy 和 HAL 对比

从 Android 源码上来分析，HAL 层主要对应源码目录如下。

1) libhardware\_legacy：旧架构，采取的链接库模块的思想设计。

2) libhardware：新架构，采用 HAL Stub 的方式来实现设计。

libhardware\_legacy 架构中 HAL 层采用的是共享库的方式，也就是直接函数调用的方式使用 HAL 层的 module，这种调用方式没有经过封装，上层直接操作底层的硬件设备。但其明显的缺点是接口不统一，如果出现多个进程同时调用，由于需要对各进程空间进行映射，会造成存储空间的浪费，同时存在代码的安全输入问题，容易造成系统崩溃。

libhardware 架构中 HAL 层使用的是 HAL Stub 模式，Stub 以动态链接库的方式存在。就操作而言，HAL 层实现对库的隐藏。Android 系统的 HAL 层包含很多的 Stub，采用统一的调用方式对硬件进行操作。上层在 HAL 层调用 Stub 的函数，然后再回调这些操作函数。通过这样的模式，实现接口统一，而且各 Stub 没有关系，上层若要对某个硬件设备进行操作，只要提供模块 ID，就能对相关设备进行操作。

图 7-9 和图 7-10 直观展示旧架构和新架构的不同之处。

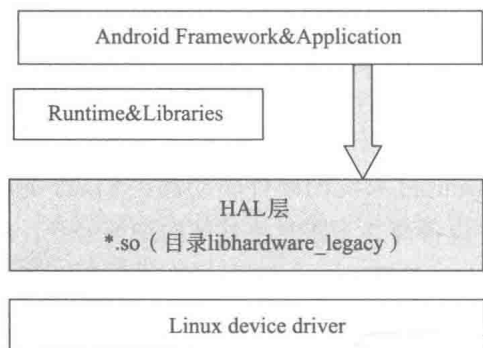


图 7-9 libhardware\_legacy HAL 层架构

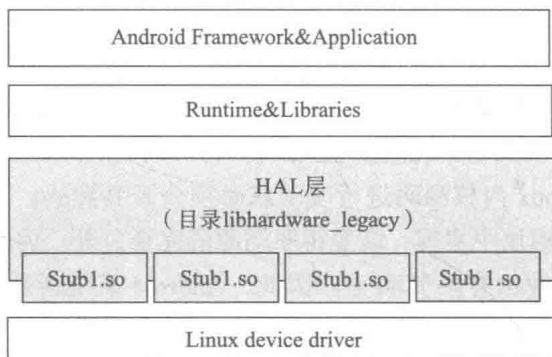


图 7-10 libhardware HAL 层架构

### 7.3.2 HAL module 架构分析

在 HAL module 架构中，必须提到 3 个重要结构体：hw\_module\_methods\_t、hw\_module\_t 和 hw\_device\_t。

### (1) hw\_module\_methods\_t 结构体

hw\_module\_methods\_t 用来定义模块初始化的方法。每个设备都需要有一个自定义的结构体, 并使用该结构体定义一个 HAL\_MODULE\_INFO\_SYM 变量。该自定义结构体的第一个成员必须为 hw\_module\_t 类型的变量。

该结构体只定义一个 open 指针, 它是一个必须实现 callback API, 负责数据填充、注册接口、初始化硬件设备等操作。

代码清单7-5

---

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
                struct hw_device_t** device);
} hw_module_methods_t;
```

---

### (2) hw\_module\_t 结构体

该结构体作为 HAL 结构体重要的一员, 含有 hw\_module\_method\_t, 可以通过 hw\_get\_module 打开该 HAL module。hw\_module\_t 最重要的成员为 id, 它是该 HAL 模块的唯一标识符。

代码清单7-6

---

```
typedef struct hw_module_t {
    /** tag must be initialized to HARDWARE_MODULE_TAG */
    uint32_t tag;

    /** major version number for the module */
    uint16_t version_major;

    /** minor version number of the module */
    uint16_t version_minor;

    /** Identifier of module */
    const char *id;

    /** Name of this module */
    const char *name;

    /** Author/owner/implementor of the module */
    const char *author;

    /** Modules methods */
    struct hw_module_methods_t *methods;

    /** module's dso */
```

---

```
void *dso;

/** padding to 128 bytes, reversed for future use */
uint_32_t reversed[32-7];

}hw_module_t;
```

---

### (3) hw\_device\_t 结构体

hw\_device\_t 用于设备的描述，同样每一个模块都需要有一个自定义的结构体来描述该设备，该自定义结构体的第一个成员必须是 hw\_device\_t，并含有相应的操作接口。hw\_device\_t 有一个 hw\_module\_t 类型的成员变量，它在模块初始化时与 HAL\_MODULE\_INFO\_SYM 中 hw\_module\_t 类型的变量进行绑定。

该结构含有 hw\_module\_t，它表示该硬件设备，以及定义该硬件设备的属性、方法。

代码清单7-7

```
typedef struct hw_device_t{
/** tag must be initialized to HARDWARE_DEVICE_TAG */
uint_32_t tag;

/** version number for hw_device_t */
uint_32_t version;

/** reference to the module this device belongs to */
struct hw_module_t* module;

/** padding reversed for future use */
uint_32_t reversed[12];

/** Close this device */
int (*close)(struct hw_device_t* device);

}hw_device_t;
```

---

## 7.3.3 HAL 实现流程

1) 如果 Native Service 需要获取 HAL Stub，使用调用函数 hw\_get\_module(const char \*id, const struct hw\_module\_t \*\*module) 创建实例。id 为该设备唯一标识符，一般通过宏定义 ID 的方式进行传参，module 为所要操作的 hw\_module\_t。

2) 在 hw\_module\_t 实例中需要使用到 hw\_module\_methods\_t 实例中的指针，因此在创建 hw\_module\_t 实例前要先声明或者创建一个 hw\_module\_methods\_t 实例。

3) 在 hw\_module\_method\_t 中需要使用 open 方法，因此在创建 hw\_module\_methods\_t 实例前要先声明一个 open 方法。

4) 实现该 HAL Stub 的 open 方法, 在函数中对设备结构体进行初始化工作。可借助此方法调用 C 库的 open 方法打开 Linux 设备文件, 获取该设备的文件描述符, 进而达到操作设备的目的。HAL 通过 hw\_module\_t->methods->open 获取 hw\_device\_t 的指针。

5) 在包含 hw\_device\_t 的结构体中添加所需的操作函数指针, 并在源码中实现, 通过调用该接口实现对内核层驱动的操作。

## 7.4 Android 系统 Camera 与 WiFi 实现

### 7.4.1 Android 中的 Camera 实现

在 Android 中, Camera 提供了取景、视频录制和拍摄照片等功能, 并且还具有各种控制类的接口, Camera 系统包括了 Camera 驱动程序、Camera 硬件抽象层、AudioService、Camera 本地库、Camera 的 Java 框架类和 Java 应用对 Camera 系统的调用, 如图 7-11 所示。

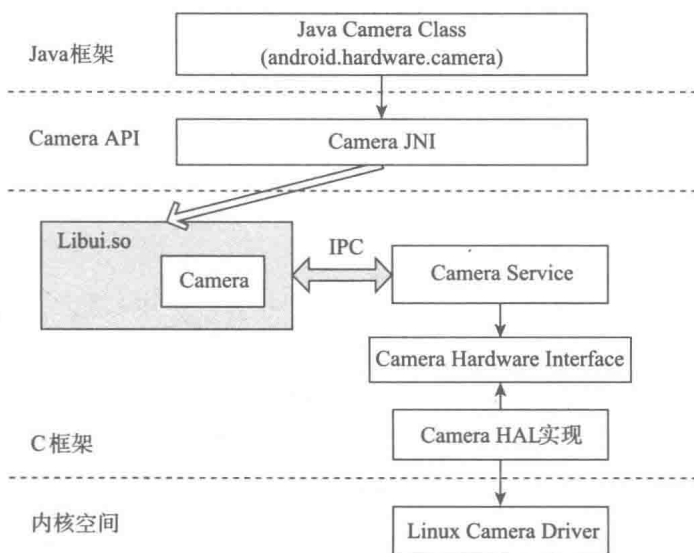


图 7-11 Camera 的系统结构

#### 1. 应用层

Camera 的应用程序为调用 API, 其代码路径在 package/apps/Camera, 通过对参数的设定, 实现特定的功能。假设要在一个 Android 应用中使用 Camera 类, 则需要 Manifest.xml 中加入 Camera 权限的声明。

#### 2. Framework 层硬件服务

Camera 应用框架层则将应用与底层的实现隔离, 方便应用与底层的开发和移植。对 Camera 而言, Framework 主要分为 4 部分: Java 接口、JNI 部分、Camera Client、Camera Service。

### (1) Java 接口

代码 Camera.java (frameworks/base/core/java/android/hardware/Camera.java) 声明了很多 native 方法, 通过 JNI 接口调用本地方法, 还有一些是自己实现, 这部分将编译成 framework.jar, 成为 SDK API。

Camera 构造方法如下:

代码清单7-8

---

```
Camera(int cameraId) {
    mShutterCallback = null;
    mRawImageCallback = null;
    mJpegCallback = null;
    mPreviewCallback = null;
    mPostviewCallback = null;
    mZoomListener = null;

   Looper looper;
    if ((looper = Looper.myLooper()) != null) {
        mEventHandler = new EventHandler(this, looper);
    } else if ((looper = Looper.getMainLooper()) != null) {
        mEventHandler = new EventHandler(this, looper);
    } else {
        mEventHandler = null;
    }

    native_setup(new WeakReference<Camera>(this), cameraId);
}
```

---

初始化回调接口的成员变量, 调用 native\_setup(), 在 JNI 层完成初始化操作。完成初始化后, 便可以通过 setPreviewDisplay 对 Camera 进行操作。

### (2) JNI 接口

Camera 的 Java 本地调用 (即 JNI 接口), 代码路径如下:

frameworks/base/core/jni/android\_hardware\_Camera.cpp

一方面注册 JNI 方法表, 使得 Java 层可以进行调用, 另一方面, 将 Camera Service 的信息传递给 Java 层, 这部分的内容编译成 libandroid\_runtime.so。

### (3) Camera Client

Camera 本地框架, 源码路径: frameworks/base/libs/camera。

该路径下主要包含 5 个源文件, 被编译成 libcamera\_client.so。该库是作为 Camera 框架中的客户端, 并通过 Binder 与服务端进行通信。

### (4) Camera Service

Camera 的服务端, 源码路径: frameworks/base/services/camera/libcameraservices。

CameraService 是 Camera 框架的中间层, 用于连接 Camera 硬件接口和 Camera 客户端,

通过调用 HAL 层的接口实现。这部分将被编译成 libcameraservice.so。

Camera 在运行时，分成 Client 和 Server 两部分，通过 Binder 机制进行通信，通过 Client 调用接口，而实际在 Server 中实现功能。

### 3. Camera HAL

这个层为用户空间的驱动代码，HAL 层继承了 CameraHardwareInterface 接口，根据 V4L2 规范实现底层硬件驱动，生成 libcamera.so 供上层调用。

V4L2(Video4linux2) 是 Linux 中关于视频设备的内核驱动。在 Linux 中，视频设备是设备文件，摄像头在 /dev/video0 下。

CameraHardwareInterface 定义 Camera 硬件的接口，代码如下所示：

代码清单7-9

---

```
class CameraHardwareInterface : public virtual RefBase {
public:
    virtual ~CameraHardwareInterface() { }
    virtual sp<IMemoryHeap>      getPreviewHeap() const = 0;
    virtual status_t      startPreview(preview_callback cb, void* user) = 0;
    virtual void          stopPreview() = 0;
    virtual status_t      autoFocus(autofocus_callback,
                                   void* user) = 0;
    virtual status_t      takePicture(shutter_callback,
                                     raw_callback,
                                     jpeg_callback,
                                     void* user) = 0;
    virtual status_t      cancelPicture(bool cancel_shutter,
                                       bool cancel_raw,
                                       bool cancel_jpeg) = 0;
    virtual status_t      setParameters(const CameraParameters& params) = 0;
    virtual CameraParameters getParameters() const = 0;
    virtual void release() = 0;
    virtual status_t dump(int fd, const Vector<String16>& args) const = 0;
};
```

---

setParameters() 函数通过参数传递通知 HAL 使用的硬件摄像头，以及它工作的参数。同时在 HAL 层分配存储 preview 数据的 buffers。

纵观整个 CameraHardwareInterface，所有的方法都是基于一个类型为 camera\_device\_t 的变量，在 hardware/libhardware/include/hardware/camera.h 中定义。该结构体定义如下：

代码清单7-10

---

```
typedef struct camera_device {
    hw_device_t common;
    camera_device_ops_t *ops;
    void *priv;
} camera_device_t
```

---

对 `hw_device_t` 进行封装，用以设备初始化。

回到 `CameraService.cpp` 的类定义中，有一个 `connect` 方法，它是通过以下方式获取 HAL 的模块的：

代码清单7-11

```
hw_get_module(CAMERA_HARDWARE_MODULE_ID,
              (const hw_module_t **) &mModule)
```

在 `hardware\ti\omap3\camera` 下主要有 3 个文件，`V4l2camera.cpp` 对照相过程中进行抓取图片和保存图片；`CameraHal_Module.cpp` 文件实现了上面分析的 `hw_device_t` `common` 和所有的 `open`、`ioctl`、`mmap`、`close` 等操作，加载模块库所使用的识别 `IDCAMERA_HARDWARE_MODULE_ID` 和其他信息都在此文件中定义。

#### 4. Camera Linux 驱动

对于 Camera 而言，一般遵守 V4L2 规范，将 Camera 的原子功能通过 `ioctl` 提供给 HAL 调用。驱动目录为 `kernel/driver/media/video`。

### 7.4.2 Android 系统 WiFi 实现

WiFi 是一种可将个人电脑、手持设备（手机或 PDA）等终端以无线方式相互连接的技术。WiFi 系统的上层接口包括数据部分和控制部分，数据部分通常是一个和以太网卡类似的网络设备，控制部分用于实现接入点操作和安全验证处理。在软件层，WiFi 系统包括 Linux 内核程序与协议，还包括本地部分、Java 框架类，WiFi 系统向 Java 应用程序提供了控制类接口。

Android 平台中 WiFi 系统的基本层次如图 7-12 所示。

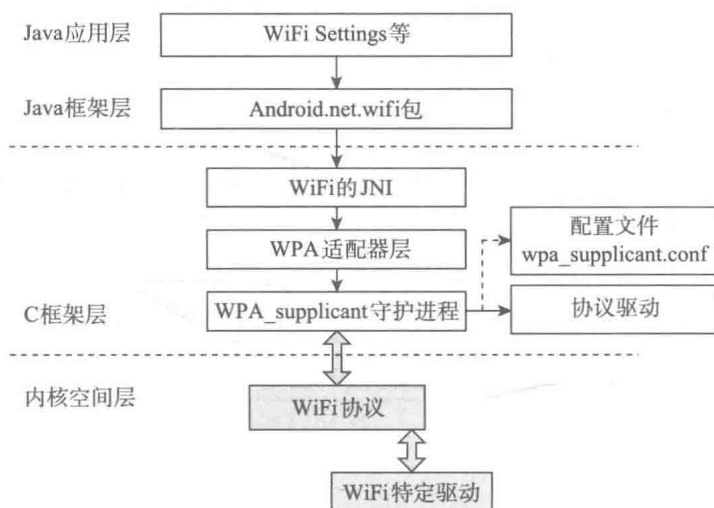


图 7-12 WiFi 的系统结构



由图 7-12 可知, Android 平台中 WiFi 系统主要包括驱动程序和协议、wpa\_supplicant 守护进程、Android 适配器层、WiFi JNI、WiFi 框架层、Settings 应用。

### 1. wpa\_supplicant

wpa\_supplicant 是一个开源的项目, 经过 Google 的修改后加入 Android 系统源码中, 通过 socket 与驱动进行交互, 并将数据传递给用户。总而言之, 它是 WPA 的应用层认证的客户端, 负责认证的登录、加密等。在 Android 系统中, 其源码路径为:

external/wpa\_supplicant/

external/wpa\_supplicant\_6/

external/wpa\_supplicant\_8/

wpa\_supplicant 的核心是消息循环, 在这个循环中处理 WPA 状态机、驱动、配置、控制等。该工程编译后将生成动态链接库 libwpaclient.so 和守护进程 wpa\_supplicant。

### 2. 适配器层

在 Android 系统中, 适配器层是通用的 wpa\_supplicant 的封装, 作为 WiFi 的抽象层来使用。该适配层主要实现与 wpa\_supplicant 的通信工作, 实现 Framework 层的加载、控制等功能。

代码路径: hardware/libhardware\_legacy/wifi/。

wpa\_supplicant 适配层包含一些驱动加载和卸载、wpa\_supplicant 启动等, 同时也实现了简单的接口, 比如 wifi\_command, 这个函数被 JNI 调用, 将上层发送的命令传递给 wpa\_supplicant, 而 wifi\_command 再调用 wifi\_send\_command 函数具体实现; wifi\_wait\_for\_event() 实现阻塞等待 wpa\_supplicant 消息。

### 3. WiFi JNI

WiFi 的 JNI 接口实现源码路径如下:

frameworks/base/core/jni/android\_net\_wifi\_Wifi.cpp

在该代码中, 本地函数通过调用适配层的接口实现。

### 4. WiFi Framework 层

WiFi 系统分为 WiFi 客户端与 WiFi 服务端, 两者通过 Binder 机制进行通信。

其源码路径如下:

frameworks/base/services/java/com/android/server/

frameworks/base/wifi/java/android/net/wifi/

在 Android 系统启动后, 会调用 ServiceManager.addService 注册 WifiService。WifiService 是该 Framework 层的主要部分, 它实现了驱动加载、开启 wpa\_supplicant、扫描 AP 等功能。该服务可以根据客户端的具体命令, 对底层进行相应的控制操作。

WifiManager 是 WiFi 与用户的接口, 用户可以通过它发送相应的操作指令。

## 5. WiFi Settings

通过 Android 系统的 Settings 应用程序实现对 WiFi 的配置、控制等操作。该部分的代码目录为：`packages/apps/Settings/src/com/android/settings/wifi/`。在 Settings 应用程序中，用户可以通过 WiFi 的设置界面实现 WiFi 的开关、扫描等功能。

## 第 8 章

# Android 底层开发实例讲解

本章主要对 Android 底层开发内容进行讲解，分别对相关技术进行总结，并从具体的芯片入手进行实例讲解。

### 8.1 底层开发相关技术概览

针对 Android 的底层开发技术，已经通过前 6 章的讲解进行了详尽的讲述，这当中涉及以下内容：

- Android 系统的架构分析及其环境搭建
- Android 系统底层源码的结构分析
- Android 系统内核的相关分析
- Android 系统设备驱动的相关分析和实现展示
- Android 系统相关的工具讲解

从事底层开发的人员所要打交道的主要有 3 项：内核、移植和驱动。

由于在嵌入式系统开发中人机交互简单，因此内核更加受到重视。一般情况下，需要直接对内核进行操作，这时需要考虑针对不同架构、不同平台的内核移植问题。

嵌入式 Linux 的可移植性在嵌入式系统领域具有无可比拟的优势地位。内核代码中将近 85% 的代码均为 ARM、PPC、x86、Microblaze 等一系列处理器平台的驱动代码，这就说明这一问题。

在嵌入式系统领域，另外一个重要的部分就是驱动程序，即用于与硬件交互的相关部分，它实现了硬件操作的抽象化，使程序员可以更专注于上层应用的开发。

### 8.2 实例讲解——基于 Zynq 的 Android 移植

这一节将深入探索 Zynq 这块 Xilinx 费尽心血而开发出来的芯片。借助搭载的高性能 ARM Cortex9 硬核，在 Zynq 芯片上运行 Android 以及 Linux 不存在任何问题。Zynq-7000 平台是第一个搭载 Zynq 芯片的开发平台，即是搭载了 ARM+FPGA 体系结构的 SoC 平台。不

同于以往搭载的 PPC 硬核，这次搭载的 ARM Cortex9 硬核对于外设的管理控制以及对于 FPGA 资源的管理上都体现出了更多的优势。

针对 Android 的运行，Zynq-7000 平台可以成功地通过编译并运行起来。但鉴于 Zynq-7000 平台售价过高，本文选用了 Digilent-Avnet 最新开发的 ZedBoard 开发板进行讲解。考虑到硬件平台有所变动，因此编译过程中将会有所修改。

对于 Android 的移植工作主要包括以下几个步骤：

- 1) 主机开发环境的搭建。
- 2) Linux 内核的编译。
- 3) Android 文件系统编译。
- 4) 准备 SD 卡以启动 Android 系统。

### 8.2.1 主机开发环境的搭建

Linux 系统主机进行开发环境的搭建包括以下步骤：

- 1) 安装 Xilinx ISE 14.4 以及 CodeSourcery Lite ARM 交叉编译工具链。
- 2) 安装 Oracle JDK1.6。
- 3) 检查 gcc 版本是否为 4.4 版本。

这里对于交叉编译工具链的声明，推荐修改 `~/.bashrc` 文件，这是一劳永逸的方法。

除了这些工具外，针对源码的下载需要用到 git，交叉编译中所需用到的工具包如下：

```
fakeroot build-essential crash kexec-tools makedumpfile kernel-wedge git-core
libncurses5 libncurses5-dev libelf-dev asciidoc binutils-dev curl
```

对于 ZedBoard 开发板，需要使用 Digilent 针对 ZedBoard 的 Linux 开发所做的硬件设计的相关文件。这些文件可以从以下网址下载：

[http://www.digilentinc.com/Data/Products/ZedBoard/ZedBoard\\_Linux\\_Design.zip](http://www.digilentinc.com/Data/Products/ZedBoard/ZedBoard_Linux_Design.zip)

接下来就开始内核的编译之旅了。

### 8.2.2 Linux 内核的编译

前面提到，对于 Zynq-7000 而言，在发布之初就对 Android 系统进行了支持。这里简单回顾一下 Zynq-7000 搭载 Android 系统的步骤。

#### 1. 从官方仓库中克隆相关资源

这当中包括 Linux 源代码、Android 源代码。相关仓库内容如下：

```
git clone git://git.iveia.com/scm/xilinx/android/kernel/zynq.git
```

对于 Android 源代码，这里需要详细说明一下相关内容的下载。

对于所需的 Android 版本的选择，可以通过 IVEIA 提供的官方仓库进行确定，步骤如图 8-1 所示。

```

favorming@favorming-D5LabPC: ~/Desktop/git/zedboard/manifest
File Edit View Search Terminal Help
favorming@favorming-D5LabPC:~/Desktop/git/zedboard$ git clone git://git.iveia.com/scm/xilinx/android/platform/manifest.git
Cloning into 'manifest'...
remote: Counting objects: 229, done.
remote: Compressing objects: 100% (99/99), done.
remote: Total 229 (delta 65), reused 229 (delta 65)
Receiving objects: 100% (229/229), 77.55 KiB | 21 KiB/s, done.
Resolving deltas: 100% (65/65), done.
favorming@favorming-D5LabPC:~/Desktop/git/zedboard$ ls
manifest
favorming@favorming-D5LabPC:~/Desktop/git/zedboard$ cd manifest/
favorming@favorming-D5LabPC:~/Desktop/git/zedboard/manifest$ ls
base-for-3.0-gpl.xml base-for-3.1-gpl.xml base-for-3.2-gpl.xml default.xml
favorming@favorming-D5LabPC:~/Desktop/git/zedboard/manifest$ git branch -r
origin/HEAD -> origin/master
origin/android-1.6_r1
origin/android-1.6_r1.1
origin/android-1.6_r1.2
origin/android-1.6_r1.3
origin/android-1.6_r1.4
origin/android-1.6_r1.5
origin/android-1.6_r2
origin/android-2.0.1_r1
origin/android-2.8_r1

```

图 8-1 仓库克隆步骤

确定了所需的版本之后，依照 Android 官方源代码的下载方式进行下载，通过 repo 工具进行下载更新。这里要注意的是，-b 之后所选择的分支即为通过 IVEIA 官方仓库进行选择的版本。本文选择 android-zynq-1.0，具体如图 8-2 所示。

```

favorming@favorming-D5LabPC: ~/Desktop/git/zedboard/manifest
File Edit View Search Terminal Help
origin/android-4.0.1_r1.2
origin/android-4.0.2_r1
origin/android-4.0.3_r1
origin/android-cts-2.2_r8
origin/android-cts-2.3_r10
origin/android-cts-2.3_r11
origin/android-cts-4.0_r1
origin/android-cts-verlfter-4.0_r1
origin/android-sdk-4.0.3-tools_r1
origin/android-sdk-4.0.3_r1
origin/android-sdk-adt-r16.0.1
origin/android-zynq-0.5
origin/android-zynq-0.6
origin/android-zynq-0.7
origin/android-zynq-1.0
origin/froyo
origin/gingerbread
origin/gingerbread release
origin/lcs-mr0
origin/lcs-mr1
origin/master
origin/tradedef
origin/tradefed

```

图 8-2 仓库版本选择

## 2. 进行编译工作

在源代码的文件夹中发现内核配置文件 xilinx\_android\_defconfig，如图 8-3 所示。

```

favorming@favorming-DSLAbPC: ~/Desktop/git/zedboard/zynq/arch/arm/configs
File Edit View Search Terminal Help
h7202_defconfig      tct_hammer_defconfig
hackkit_defconfig    tegra_defconfig
imote2_defconfig     trizeps4_defconfig
integrator_defconfig u300_defconfig
iop13xx_defconfig    u8500_defconfig
iop32x_defconfig     usb-a9260_defconfig
iop33x_defconfig     versatile_defconfig
lveia_atlas_i_lpe_defconfig vexpress_defconfig
lpx2000_defconfig    viper_defconfig
lpx23xx_defconfig    xcep_defconfig
lpx4xx_defconfig     xilinx_amp_cpu0_defconfig
jornada720_defconfig xilinx_amp_cpu1_defconfig
kirkwood_defconfig  xilinx_android_defconfig
ks8695_defconfig     xilinx_defconfig
lart_defconfig       xilinx_palladium_defconfig
loki_defconfig       xilinx_palladium_smp_defconfig
lpd270_defconfig     xilinx_qemu_defconfig
lubbock_defconfig    xilinx_test_defconfig
mackerel_defconfig  xilinx_test_smp_defconfig
magician_defconfig  xilinx_xylon_defconfig
mainstone_defconfig xilinx_xylon_defconfig_golden
mini2440_defconfig  xilinx_zynq_defconfig
mmp2_defconfig      zeus_defconfig
favorming@favorming-DSLAbPC:~/Desktop/git/zedboard/zynq/arch/arm/configs$

```

图 8-3 内核配置文件

依据这个配置文件对 Linux 内核进行配置，继而编译所需的第一个文件 zImage。命令如图 8-4 所示。

```

favorming@favorming-DSLAbPC: ~/Desktop/git/zedboard/zynq
File Edit View Search Terminal Help
favorming@favorming-DSLAbPC:~/Desktop/git/zedboard/zynq$ export ARCH=arm
favorming@favorming-DSLAbPC:~/Desktop/git/zedboard/zynq$ export CROSS_COMPILE=ar
m-xilinx-linux-gnueabi-
favorming@favorming-DSLAbPC:~/Desktop/git/zedboard/zynq$ make xilinx_android_def
config
#
# configuration written to .config
#
favorming@favorming-DSLAbPC:~/Desktop/git/zedboard/zynq$ make zImage

```

图 8-4 编译 zImage 文件

编译完成后得到的 zImage 文件位于 arch/arm/boot/zImage。

这是最重要的镜像文件，在进一步熟悉 Linux 内核的配置流程后，可以对 xilinx\_android\_defconfig 进行深入的学习了解。

完成了镜像文件，还需要对设备树进一步了解。

说到 ARM 中的设备树，这当中还有点渊源。

2011 年 3 月 17 日，Linus Torvalds 在 Linux 邮件列表中称：this whole ARM thing is a fucking pain in the ass。也正是这句话，引发了 ARM 社区对于并入 Linux 源代码中的代码的大量修正。在这之前，ARM 将大量的板级细节代码写在了 arch/arm/ 文件夹下，导致了在内核中出现了大量本不该出现的代码，有人做过相关统计，这个代码数量在数万行左右。

鉴于这种情况，ARM 社区的开发者们开始使用设备树即 Device Tree 这一概念。Device

Tree 是一种描述设备硬件的数据结构。采用 Device Tree 之后,许多硬件的细节都可以通过这一文件传递给 Linux,不需要再在内核源代码中保留冗余代码。

而这也是 Linux 2.6 版本与 Linux 3+ 版本的巨大差别之一。

借助这个名称,可以形象地想象到,bootloader 将这棵树上所描述的各类信息,包括 CPU、内存、总线等,传递给了内核,由内核展开进行下一步分配,并将资源绑定到展开的相应设备。

现在所讲的 Device Tree 在内容上主要涵盖的文件是每个开发平台所对应的 dts 文件。这个文件可以在内核源代码的相应架构下的 boot/dts 文件夹下找到。

这个文件是一种 ASCII 文本格式的 Device Tree 描述。基本上一个 .dts 文件对应一个开发平台。对于这个 .dts 文件,需要使用工具对其进行编译。.dts 格式文件里面所阐述的内容格式规范,便于进行阅读和修改,但对于机器而言,需要先转换为二进制格式的描述文件才能由机器执行。这里就需要用到 DTC (Device Tree Compiler)。DTC 的源代码可以在 Linux 的内核源代码中找到,位于 scripts/dtc 目录下。

通过编译内核得到了 zImage 文件,通过使用 DTC 将 .dts 编译为 .dtb。dtb 文件是对二进制格式 Device Tree 的描述,可以交由 Linux 进行内核解析。一般 Bootloader 在启动时进行 kernel 的引导前会将这个 .dtb 文件读取到内存中。

那么如何进行 Zynq 开发板上的 .dtb 文件的编译呢?还是使用 DTC 工具,具体的参数如下所示:

```
user@local-machine $ scripts/dtc/dtc \
-I dts -O dtb \
-o devicetree.dtb \
arch/arm/boot/dts/zynq-zc702-xylon-hdmi.dts
```

当两个文件都已经生成后,还需要生成 Boot Loader。关于 Boot Loader 的生成,在只考虑 PS 部分的情况下,需要两个文件——FSBL 的 elf 文件及 u-boot.elf 文件。对于 System.bit 这个文件需要说明,仅在将 Zynq 认为是一块 ARM 开发板的情况下,可以不需要 System.bit 文件,但一旦添加了外设,就需要在 XPS 中对这个文件进行额外的生成。

下面分两个部分讲解这两个必要的文件的生成过程。

对于 u-boot.elf,首先需要下载 u-boot 的相关源代码。

```
git clone git://git.xilinx.com/u-boot-xlnx.git
cd u-boot-xlnx (此处略去设置环境变量的语句)
make zynq_zed_config
make
```

在编译完成后会提示有一个文件生成,即 u-boot,将其重新命名为 u-boot.elf。

对于 zynq\_fsbl\_0.elf 文件,需要通过 Xilinx 的 SDK 进行编译得到。进入 XPS,生成一个工程。通过 XPS 的 Export Design 进入 SDK,在这里新建一个 C 工程,如图 8-5 所示。这

个工程选择 First Stage Bootloader (FSBL), 用于实现 u-boot 之前的所有初始化与启动工作, 如图 8-6 所示。



图 8-5 新建一个 C 工程

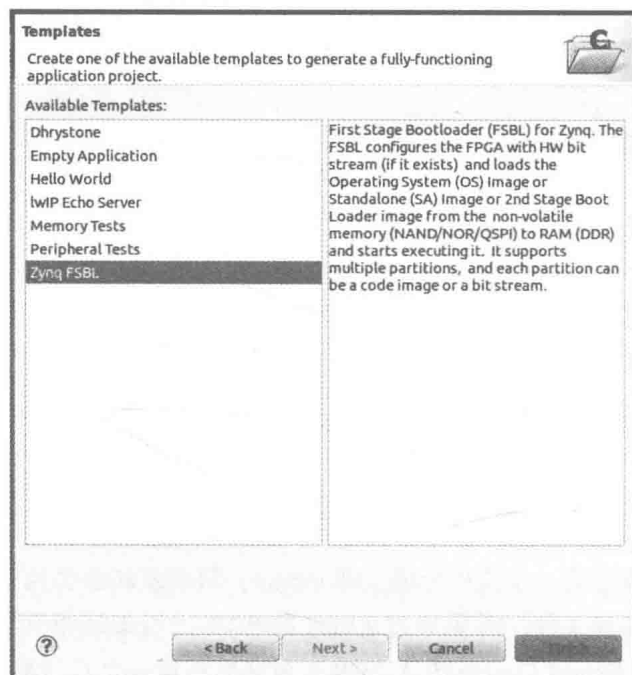


图 8-6 生成 FSBL



有了这几个文件，在 SDK 的上方菜单栏中找到栏目 Xilinx Tools，选择下拉列表中的 Create Boot Image，将这两个文件加入，并选择路径进行编译，会在相关路径中得到一个 u-boot.bin 的文件，将其重命名为 BOOT.bin。这就是启动所需要的文件。如图 8-7 所示即为 SDK 中生成 u-boot.bin 的步骤。

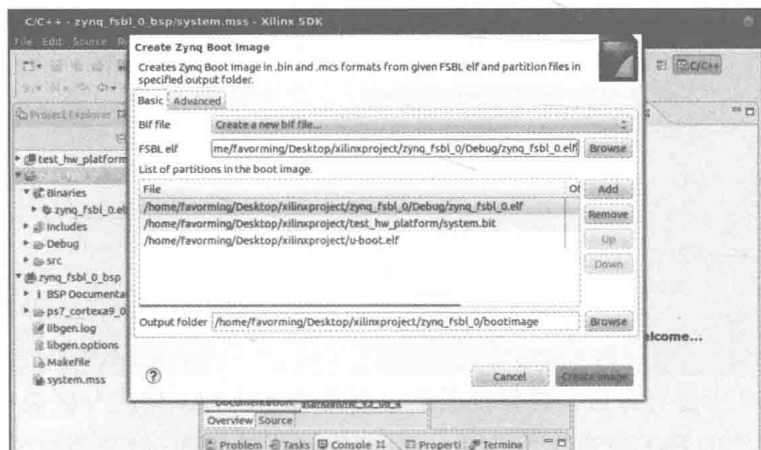


图 8-7 SDK 中 u-boot.bin 的生成

此外，也可以通过 <http://git.iveia.com/support> 网站得到上述所有文件。

### 3. Android 源代码编译

在上述 Linux 内核源代码的编译工作结束后，对 Android 的源代码进行编译。

#### 1) 对源代码直接进行编译。

##### 代码清单8-1

```
user@local-machine $ cd /DIR/TO/android
user@local-machine $ . build/envsetup.sh
user@local-machine $ lunch generic-eng
user@local-machine $ make -j 4
```

#### 2) 安装必要软件

```
sudo apt-get install genext2fs
```

#### 3) 建立编译规则文件。在根目录文件夹下建立一个编译规则文件，命名为 Makefile.zynq。

文件内容如下：

##### 代码清单8-2

```
OUT_DIR:=out/target/product/generic
ROOT_IMG:=root.img
ROOTFS:=rootfs
ROOT_DIRS=lib/modules tmp media
.PHONY: dummy
```

```

$(ROOT_IMG): dummy
    rm -rf $@
    sudo rm -rf $(ROOTFS)
    cp -r $(OUT_DIR)/root $(ROOTFS)
    cp -r $(OUT_DIR)/system $(ROOTFS)
    cd $(ROOTFS) && mkdir -p $(ROOT_DIRS)
    sudo chown -R root:root $(ROOTFS)
    sudo genext2fs -d $(ROOTFS) -b $$((80*1024)) -m 0 -N $$((64*1024)) $(ROOT_IMG)
    sudo chown $(shell id -u):$(shell id -g) $(ROOT_IMG)
# Phony target forces the rootfs image file to be rebuilt on each make
dummy:

```

4) 编译 root.img 文件。执行编译命令，得到 root.img 文件。

make -f Makefile.zynq

5) 启动 Zynq 上的 Android 系统。这里使用上文编译出的文件，以及 Xylon 提供的 ramdisk8M.image.gz 文件启动系统。

还需要将 SD 卡进行分区。要保证最后，有足够大的 FAT 格式分区装载所有的启动文件，以及一个额外的 FAT 格式分区留给 Android 作为外部存储。这里假设 SD 卡在电脑中显示的设备节点号为 /dev/sdb，而后进行如下操作，一般情况下显示的节点号即为 /dev/sdb。

SD 卡的分区删除操作如图 8-8 所示。

```

favorming@favorming-DSLAbPC: ~
File Edit View Search Terminal Help
favorming@favorming-DSLAbPC:~$ sudo /sbin/fdisk /dev/sdb
sdb sdb1 sdb2
favorming@favorming-DSLAbPC:~$ sudo /sbin/fdisk /dev/sdb

Command (m for help): p

Disk /dev/sdb: 3987 MB, 3987734528 bytes
4 heads, 16 sectors/track, 121696 cylinders, total 7788544 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00016f61

   Device Boot      Start         End      Blocks    Id  System
/dev/sdb1            16        976575       488280    c   W95 FAT32 (LBA)
/dev/sdb2        976576       7788543      3405984    83   Linux

Command (m for help): d
Partition number (1-4): 1

Command (m for help): d
Selected partition 2

Command (m for help):

```

图 8-8 对 SD 卡删除分区操作

在删除分区之后，对 SD 卡进行重新分区，如图 8-9、图 8-10 所示。

选定 b 为所要的分区类型，具体如图 8-11 所示。

如图 8-12 所示，对分区 2 进行分区分配与选型。

```

favorming@favorming-DSLAPC: ~
File Edit View Search Terminal Help
Command (m for help): p

Disk /dev/sdb: 3987 MB, 3987734528 bytes
4 heads, 16 sectors/track, 121696 cylinders, total 7788544 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00016f61

   Device Boot      Start         End      Blocks   Id  System
Command (m for help): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-7788543, default 2048):
Using default value 2048
Last sector, +sectors or +size[K,M,G] (2048-7788543, default 7788543): 3895295
Command (m for help): t
Selected partition 1

```

图 8-9 对 SD 卡重新分区

```

favorming@favorming-DSLAPC: ~
File Edit View Search Terminal Help
Hex code (type L to list codes): L

```

0	Empty	24	NEC DOS	81	Minix / old Lin	bf	Solaris
1	FAT12	27	Hidden NTFS Win	82	Linux swap / So	c1	DRDOS/sec (FAT-
2	XENIX root	39	Plan 9	83	Linux	c4	DRDOS/sec (FAT-
3	XENIX usr	3c	PartitionMagic	84	OS/2 hidden C:	c6	DRDOS/sec (FAT-
4	FAT16 <32M	40	Verix 80286	85	Linux extended	c7	Syrinx
5	Extended	41	PPC PreP Boot	86	NTFS volume set	da	Non-FS data
6	FAT16	42	SFS	87	NTFS volume set	db	CP/M / CTOS /
7	HPFS/NTFS/exFAT	4d	QNX4.x	88	Linux plaintext	de	Dell Utility
8	AIX	4e	QNX4.x 2nd part	8e	Linux LVM	df	BootIt
9	AIX bootable	4f	QNX4.x 3rd part	93	Amoeba	e1	DOS access
a	OS/2 Boot Manag	50	OnTrack DM	94	Amoeba BBT	e3	DOS R/O
b	W95 FAT32	51	OnTrack DM6 Aux	9f	BSD/OS	e4	SpeedStor
c	W95 FAT32 (LBA)	52	CP/M	a0	IBM Thinkpad hi	eb	BeOS fs
e	W95 FAT16 (LBA)	53	OnTrack DM6 Aux	a5	FreeBSD	ee	GPT
f	W95 Ext'd (LBA)	54	OnTrackDM6	a6	OpenBSD	ef	EFI (FAT-12/16/
10	OPUS	55	EZ-Drive	a7	NeXTSTEP	f0	Linux/PA-RISC b
11	Hidden FAT12	56	Golden Bow	a8	Darwin UFS	f1	SpeedStor
12	Compaq diagnost	5c	Priam Edisk	a9	NetBSD	f4	SpeedStor
14	Hidden FAT16 <3	61	SpeedStor	ab	Darwin boot	f2	DOS secondary
16	Hidden FAT16	63	GNU HURD or Sys	af	HFS / HFS+	fb	VMware VMFS
17	Hidden HPFS/NTF	64	Novell Netware	b7	BSDI fs	fc	VMware VMKCORE
18	Hidden HPFS/NTF	64	Novell Netware	b7	BSDI fs	fc	VMware VMKCORE
18	AST SmartSleep	65	Novell Netware	b8	BSDI swap	fd	Linux raid auto

图 8-10 分区分类及对应代码

```

favorming@favorming-DSLAPC: ~
File Edit View Search Terminal Help

```

12	Compaq diagnost	5c	Priam Edisk	a9	NetBSD	f4	SpeedStor
14	Hidden FAT16 <3	61	SpeedStor	ab	Darwin boot	f2	DOS secondary
16	Hidden FAT16	63	GNU HURD or Sys	af	HFS / HFS+	fb	VMware VMFS
17	Hidden HPFS/NTF	64	Novell Netware	b7	BSDI fs	fc	VMware VMKCORE
18	AST SmartSleep	65	Novell Netware	b8	BSDI swap	fd	Linux raid auto
1b	Hidden W95 FAT3	70	DiskSecure Mult	bb	Boot Wizard hid	fe	LANstep
1c	Hidden W95 FAT3	75	PC/IX	be	Solaris boot	ff	BBT
1e	Hidden W95 FAT1	80	Old Minix				

图 8-11 选定分区类型

```

Hex code (type L to list codes): b
Changed system type of partition 1 to b (W95 FAT32)

Command (m for help): p

Disk /dev/sdb: 3987 MB, 3987734528 bytes
4 heads, 16 sectors/track, 121696 cylinders, total 7788544 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00016f61

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1             2048      3895295      1946624    b   W95 FAT32

Command (m for help):

```

图 8-11 (续)

```

favorming@favorming-DSLAbPC: ~
File Edit View Search Terminal Help
Command (m for help): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p): p
Partition number (1-4, default 2): 2
First sector (3895296-7788543, default 3895296):
Using default value 3895296
Last sector, +sectors or +size{K,M,G} (3895296-7788543, default 7788543):
Using default value 7788543

Command (m for help): t
Partition number (1-4): 2
Hex code (type L to list codes): b
Changed system type of partition 2 to b (W95 FAT32)

```

图 8-12 分区 2 分配与选型

最后通过 `w` 命令直接将相关配置写入 SD 卡，至此 SD 卡的格式化与分区工作完成，如图 8-13 所示。

```

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: Re-reading the partition table failed with error 16: Device or resource
busy.
The kernel still uses the old table. The new table will be used at
the next reboot or after you run partprobe(8) or kpartx(8)

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.

```

图 8-13 写入 SD 卡

最后进行 SD 卡分区的收尾工作，如图 8-14 所示。

```

favorming@favorming-DSLAbPC:~$ sudo partprobe /dev/sdb

favorming@favorming-DSLAbPC:~$ sudo /sbin/mkdosfs -F 32 -n "Android" /dev/sdb1
mkdosfs 3.0.12 (29 Oct 2011)
favorming@favorming-DSLAbPC:~$ sudo /sbin/mkdosfs -F 32 -n "ExternalStorage" /dev/sdb2
mkdosfs 3.0.12 (29 Oct 2011)

```

图 8-14 分区收尾工作

而后将各个文件复制至第一个分区中。

插入 SD 卡，正确设置 Zynq 开发板上的跳线，包括 J27: 1-2, J28: 1-2, J21: 2-3, J20: 2-3, J22: 1-2, J25: 1-2, J26: 2-3。接上 HDMI 线，启动开发板后就会发现 Android 出现在了屏幕中。通过 git 工具下载相关内核源码。

```
git clone https://github.com/Digilent/linux-digilent.git
```

下载完成后，不需要直接进行编译。由于 .dts 文件并不适用于 Zedboard 开发板，因此需要从下列网站下载相关的 dts 文件。

[http://www.digilentinc.com/Data/Products/ZedBoard/ZedBoard\\_Linux\\_Design.zip](http://www.digilentinc.com/Data/Products/ZedBoard/ZedBoard_Linux_Design.zip)

对下载的文件进行解压，找到其中的 devicetree.dts 文件，替换到 arch/arm/boot/dts/digilent-zed.dts 文件中。

接下来开始进入内核的编译流程。

这里要提到的是，对于编译前的配置过程，选择的配置文件为 digilent\_zed\_defconfig 文件，如图 8-15 所示。

```
favorming@favorming-DSLAPC:~/Desktop/glt/zedboard/linux-digilent$ ls arch/arm/c
onfigs/digilent_zed_defconfig
arch/arm/configs/digilent_zed_defconfig
favorming@favorming-DSLAPC:~/Desktop/glt/zedboard/linux-digilent$ make digilent
_zed_defconfig
HOSTCC scripts/basic/fixdep
HOSTCC scripts/kconfig/conf.o
SHIPPED scripts/kconfig/zconf.tab.c
SHIPPED scripts/kconfig/zconf.lex.c
SHIPPED scripts/kconfig/zconf.hash.c
HOSTCC scripts/kconfig/zconf.tab.o
HOSTLD scripts/kconfig/conf
#
# configuration written to .config
#
favorming@favorming-DSLAPC:~/Desktop/glt/zedboard/linux-digilent$ make menuconf
lg
```

图 8-15 选择配置文件

而后通过 make menuconfig 命令进入图形配置界面，选择 Device Drivers，将 Staging drivers 选择为 Y，按回车键进入 Android 选项界面，将其中的所有选项都勾选上，勾选的方式为按空格键，如图 8-16 所示。

保存并退出图形配置界面，使用命令 make 进行编译。与上述一致的是，在编译完成后，同样在 arch/arm/boot/zImage 中会出现一个 zImage 文件。这是需要的 zImage 文件。

### 8.2.3 Android 文件系统的编译

对于 Zedboard 对应的 Android 源代码的获取，与之前对于 Zynq 的 Android 源码获取过程并无不同，都是获取 android-zynq-1.0 分支内的代码内容。

这里讲解一下如何修改 Android 源码中的内容，使其支持 USB OTG 模式下的键盘使用。

修改的文件为 PhoneWindowManager.java，位于 frameworks/base/policy/src/com/android/internal/policy/impl/。

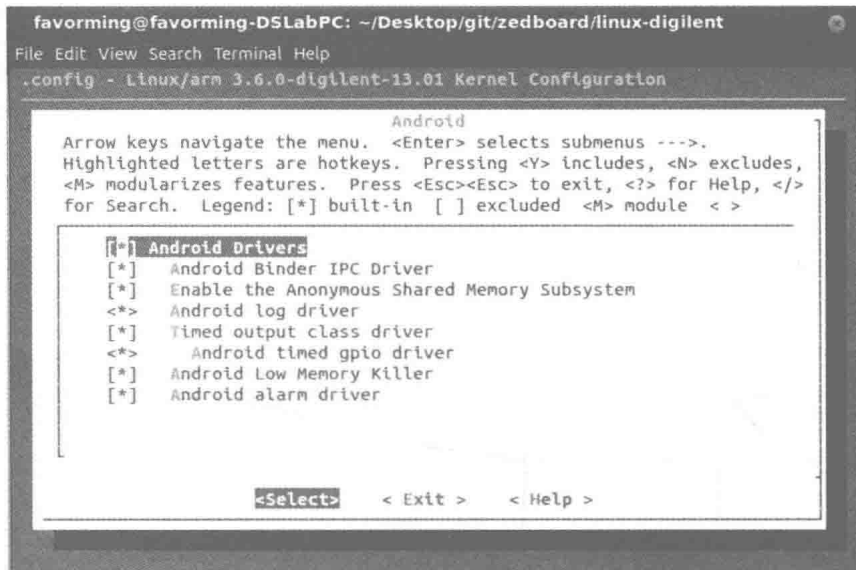


图 8-16 图形配置界面

将其中的

```
if (isScreenOn || isInjected) { (line 1770)
```

替换为

```
if (isScreenOn || isInjected || true) {
```

之后就可以对 Android 源码进行编译。

确认 gcc 版本为 4.4，安装必需的安装包。

```
git gnupg flex bison gperf build-essential zip curl libc6-dev
libncurses5-dev:i386 x11proto-core-dev libx11-dev:i386 libreadline6-dev:i386
libgll-mesa-glx:i386 libgll-mesa-dev g++-multilib mingw32 tofrodos
python-markdown libxml2-utils xsltproc zlib1g-dev:i386
```

在编译前，执行如下命令，确保环境变量配置正确。

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

而后执行以下命令：

```
lunch generic-eng
make
```

等待编译完成。而后的步骤就与先前 Zynq 的一致了，建立编译规则 Makefile.zynq，进行编译。

最后就会得到一个文件 root.img，大小在 80M 左右。

### 8.2.4 SD 卡的准备以及 Android 系统的启动

工作准备完成，还缺少一个 `ramdisk8M.image.gz`。为了保证 Android 顺利自启动，需要对这个压缩包进行修改。

```
gunzip ramdisk8M.image.gz
sudo mount ramdisk8M.image.gz -o loop ramdisk
```

修改 `etc/init.d/rcS` 文件，将下列内容添到文件最后：

```
echo "Mounting SD card to /mnt/sd"
mkdir -p /mnt/sd
mount /dev/mmcblk0p1 /mnt/sd
echo "++ Starting startup.sh script on SD card"
if [ -f /mnt/sd/startup.sh ]; then
    . /mnt/sd/startup.sh
fi
```

而后重新打包。

```
sudo umount ramdisk
gzip ramdisk8M.image
```

建立一个脚本文件，用以启动系统。在 SD 卡的根目录中建立 `starup.sh` 文件，文件内容为代码清单 8-3。

代码清单8-3

---

```
echo "++ Preparing for Android"
mkdir /mnt/root
mount -o loop /mnt/sd/root.img /mnt/root/
mount -t proc proc /mnt/root/proc
mount -t sysfs sys /mnt/root/sys
mount -t tmpfs tmp /mnt/root/data
mkdir -p /mnt/root/tmp/sd
mount /mnt/sd /mnt/root/tmp/sd
echo "++ Starting Android"
chroot /mnt/root /init
```

---

这时所有文件就都已经准备妥当了。接上 HDMI 线，依据先前的规则进行跳线帽的安装，就可以看到 Android 系统也顺利启动了。

## 8.3 移植讲解——基于 pcDuino 的 Android 移植

在本节将结合现在流行的开发套件 pcDuino，讲解常见的 ARM 开发平台上 Android 操作系统移植，从平台和环境搭建入手，一步一步完成开发的准备工作，对内核编译和 Android

编译进行通用的步骤讲解，最后完成移植。

通过本节，读者将得到关于 pcDuino 最新版本的开发移植案例。

### 8.3.1 pcDuino 介绍

作为一个新晋的单板 PC 机成员，pcDuino 可谓红火，但鲜有人提及其设计及创造者——联斯普瑞电子科技有限公司。pcDuino 如同联斯普瑞孕育的“孩子”，整个团队在其身上倾注了大量的心血，目的就是为广大开发人员打造一个高性能的开源硬件平台。

2010 年，刘靖峰作为第七批“千人计划”的创业人才，与其余 4 名团队成员在武汉成立了 LinkSprite 公司，主攻芯片研发，并为国外市场生产一些开源硬件模块。硬件开发关键在于掌握客户的实际需求。刘靖峰对 Arduino 平台的计算能力有所不满，“Arduino 能够快速地应用于简单的电子系统，但它毕竟已经是过去的技术，更适合艺术创意的实现。”同时，市场上还有另一款主流开源硬件平台树莓派，树莓派与 Arduino 看起来很像，都是小型的电路板，有一些芯片和管脚在上面，但两者的运算水平的平台却是完全不同的。“在我看来，Arduino 是开源硬件，而树莓派与其说是开源硬件，不如说是采用了开源软件 Linux 的开发板。这两种开源开发板针对不同的用户群。”刘靖峰说，很遗憾树莓派无法直接兼容已有的 Arduino 生态。

“会硬件的软件工程师不多，会软件的硬件工程师也不多。”刘靖峰希望将这两大充满创意、却又相互隔阂的群体联系在一起，让软件长在硬件里面。

于是，刘靖峰看到了商机，决定研发 pcDuino。这是一款类似 Arduino 与树莓派的开源开发板，被称为“Arduino 与迷你 PC 的合体”，且比树莓派配置更高，速度更快。

在经过无数日夜的艰苦奋斗后，pcDuino 终于诞生了，它于 2013 年 3 月上线，受到了广大电子爱好者的一致追捧。pcDuino 作为硬件开发平台，被用于多种 DIY 项目，其应用数不胜数。目前，pcDuino 有 4 个版本：

- pcDuino V1
- pcDuino V2
- pcDuino Lite
- pcDuino Lite WiFi

pcDuino V2 是 pcDuino 的升级版，外观与 pcDuino 差别不大，除了增加了自带 WiFi 模块的特点外，pcDuino V2 的最大改进是板载硬件接口实现了对 Arduino Uno 接口的完全兼容（可能是解决了授权等问题）。市面上的 Arduino 扩展板可以直接插在 pcDuino V2 上，而且代码也是完全兼容的。pcDuino 实物图如图 8-17 所示。

pcDuino 是一种高性能、高性价比的迷你 PC 平台，而且配备简单易用的编译工具，能够完全运行 PC 操作系统，如 Ubuntu 和 Android ICS 等。它完全兼容现在非常流行的 Arduino 的接口和软件系统，是一款可以通过 Arduino 接口完美兼容目前市面上的所有的 Arduino shield 的产品。著名的在线开源电子网站 Sparkfun 强烈推荐这款有创意的结合，目



前在国外 Sparkfun 网站 pcDuino 的价格大约为 60 美元, 国内售价 370 元人民币左右。

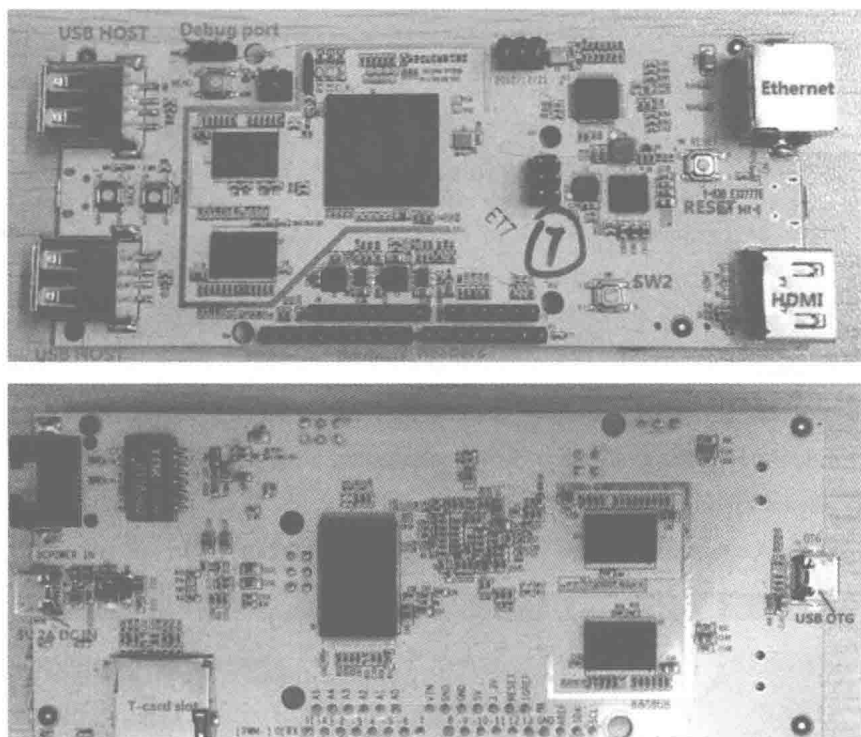


图 8-17 pcDuino 实物图

pcDuino 与 Arduino、树莓派 (Raspberry Pi) 等相比, 具有明显的特点。

### (1) 更快的 CPU

树莓派上的博通 BCM2835 采用了上一代的 ARM 架构 (ARM V6, 即 ARM 11), 速度和兼容性都受到了很大的限制。pcDuino 采用了 Allwinner A10 处理器, 即 ARM v7 处理器, 运行速度达到 1GHz。

### (2) 更大的内存

首先, pcDuino 上的内存达到 1GB 容量。开发板上采用了 4 颗海力士的 DDR3 SDRAM H5TQ2G83BFR。其次, pcDuino 和树莓派一样也有 Micro SD 卡接口, 但是 pcDuino 的系统并不是存放在 SD 卡上, 而是放在板载的 2GB NAND Flash 存储器中, 这样, 系统运行速度更快。存储器也采用海力士的 NAND Flash H27UAG8T2C。Micro SD 卡主要作为外部存储空间, 其最大容量达到 32GB。

### (3) 更多的接口

pcDuino 最大的特点就是它的扩展接口与 Arduino 的信号兼容, 融入了 Arduino 的生态系统, 可以使用各种 Arduino 扩展板, 并且可以直接使用 Arduino 的开发环境和应用程序, 且大大提升了性能, 能产生更多有创意的应用, 如扩展网络、多媒体功能等。

### 8.3.2 环境搭建

通过前面的学习，已经知道 pcDuino 采用的架构是基于全志 ARM 处理器的开发平台，主要的芯片是中央的 A10 处理器。后期还将会会有 A80 的八核版本，在目前的应用开发中，pcDuino V3 版本已能满足大众开发的需求。

对于全志 ARM 处理器，其相关的开发文档在网上已经全部开源，因此在开发过程中可以方便地找到相关资料。相应地，在环境的搭建上，也将主要借鉴 A10 的开发环境需求进行整理。

这里的 HOST 主机开发平台为 Ubuntu 12.04 版本，按照需求，将通过 apt-get 命令安装如下依赖包：gnupg, flex, bison, gperf, build-essential, zip, curl, libc6-dev, libncurses5-dev, x11proto-core-dev, libx11-dev, g++-multilib, mingw32, tofrodos, python-markdown, libxml2-utils, xsltproc, zlib1g-dev, lib32r, eadline6-dev。

这里做的是 Android 移植，因此对于 Java 的需求是必须的。通过以下下载页面 <http://www.oracle.com/technetwork/java/javase/downloads/index.html>，可以下载到需要的 Java 版本，本文采用的是 jdk-6u45-linux-x64 版本。

下载到的是一个 zip 压缩包，将其解压后，将得到一个 bin 可执行文件，如图 8-18 所示。

```
favorming@favorming-Lenovo:~/Downloads$ unzip jdk-6u45-linux-x64.zip
```

图 8-18 解压 Java JDK 压缩包

通过 chmod 命令，改变这个文件的执行权限，而后执行该可执行文件，如图 8-19 所示。

```
favorming@favorming-Lenovo:~/Downloads$ sudo chmod u+x jdk-6u45-linux-x64.bin
favorming@favorming-Lenovo:~/Downloads$ ./jdk-6u45-linux-x64.bin
```

图 8-19 执行 JDK bin 文件得到文件

至此已经得到了所需要的 Java 文件。将其复制到用户目录 /usr/lib 下，执行如图 8-20 所示命令。此时即使没有 jvm 这个文件夹，也会自动创建该文件夹。

```
favorming@favorming-Lenovo:~/Downloads$ ls jdk1.6.0_45/
bin  COPYRIGHT  db  include  jre  lib  LICENSE  man  README.html  src.zip  THIRDPARTYLICENSEREADME.txt
favorming@favorming-Lenovo:~/Downloads$ sudo mv jdk1.6.0_45/ /usr/lib/jvm
favorming@favorming-Lenovo:~/Downloads$ ls /usr/lib/jvm/
bin  COPYRIGHT  db  include  jre  lib  LICENSE  man  README.html  src.zip  THIRDPARTYLICENSEREADME.txt
```

图 8-20 设置 Java 环境

做完这步，环境已经搭建完成了。由于 Java 版本的限制，以及考虑到后续开发的方便，可以将 Ubuntu 的环境变量一次性设定好，将 Java 配置为所期望的版本，达到一劳永逸的效果，如图 8-21 所示。

```
favorming@favorming-Lenovo:~/Downloads$ vim ~/.bashrc
favorming@favorming-Lenovo:~/Downloads$ source ~/.bashrc
favorming@favorming-Lenovo:~/Downloads$ java -version
java version "1.6.0_45"
Java(TM) SE Runtime Environment (build 1.6.0_45-b06)
Java HotSpot(TM) 64-Bit Server VM (build 20.45-b01, mixed mode)
favorming@favorming-Lenovo:~/Downloads$
```

图 8-21 设置 Java bashrc 环境变量

打开~/.bashrc 文件，往里面添加内容，如图 8-22 所示。



图 8-22 修改 bashrc 文件

保存并退出。通过 source 命令，即可更新当前环境变量。

用 java -version 命令就可以查看到相应的 Java 版本已经改变为需要的版本，如图 8-23 所示。

```

favorming@favorming-Lenovo:~$ java -version
java version "1.6.0_45"
Java(TM) SE Runtime Environment (build 1.6.0_45-b06)
Java HotSpot(TM) 64-Bit Server VM (build 20.45-b01, mixed mode)
favorming@favorming-Lenovo:~$
  
```

图 8-23 查看 Java 版本

### 8.3.3 编译内核

pcDuino 的开发者们将 Android 源码分为 3 个文件存放，首先需要对这 3 个文件进行内容的提取，如图 8-24 所示。

```

favorming@favorming-Lenovo:~/Desktop/git/pcDuino$ cd android4.2/
favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2$ ls
android lichee
  
```

图 8-24 文件内容提取

通过 ls 命令，发现全志将源码的存放为两个文件夹，分别为 lichee 和 android。对于内核的编译在 lichee 中执行，这一步需要得到 u-boot、rootfs 等文件。cd 进入相应的文件夹，可以看到有一个 build.sh 的脚本文件。这个脚本文件为编译提供了很大的方便，避免了重复输入命令的操作，也略去了编译的步骤，加快了编译速度。现在通过这个脚本文件进行编译，如图 8-25 所示。

执行脚本文件，设置环境进行编译。编译完成，将得到如图 8-26 所示结果。

```

favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2$ cd lichee/
favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2/lichee$ ./build.sh -p sun7i_android
mkscript current setting:
  chip: sun7i
  Platform: android
  Board:
  Output Dir: /home/favorming/Desktop/git/pcDuino/android4.2/lichee/out/android/common
INFO: build lichee ...
INFO: build buildroot ...
Installing external toolchain
please wait for a few minutes ...
INFO: build buildroot OK.
INFO: build kernel ...
INFO: prepare toolchain ...
Building kernel

```

图 8-25 通过 build.sh 编译

```

arm-linux-gnueabi-objcopy -O srec u-boot u-boot.srec
arm-linux-gnueabi-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
make[1]: Leaving directory '/home/favorming/Desktop/git/pcDuino/android4.2/lichee/u-boot'
INFO: build u-boot OK.
INFO: build rootfs ...
INFO: skip make rootfs for android
INFO: build rootfs OK.
INFO: build lichee OK.
favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2/lichee$

```

图 8-26 编译结果

### 8.3.4 编译 Android

进入 Android 源代码的编译。

导入编译所需的环境变量，如图 8-27 所示。

```

favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2/android$ cd ../android/
favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2/android$ source build/envsetup.sh
including device/asus/grouper/vendorsetup.sh
including device/asus/tilapia/vendorsetup.sh
including device/generic/armv7-a-neon/vendorsetup.sh
including device/generic/armv7-a/vendorsetup.sh
including device/generic/mips/vendorsetup.sh
including device/generic/x86/vendorsetup.sh
including device/samsung/maguro/vendorsetup.sh
including device/samsung/manta/vendorsetup.sh
including device/samsung/toroplus/vendorsetup.sh
including device/samsung/toro/vendorsetup.sh
including device/softwinner/common/vendorsetup.sh
including device/softwinner/wing-k70/vendorsetup.sh
including device/softwinner/wing-n71j/vendorsetup.sh
including device/softwinner/wing-nck70/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2/android$

```

图 8-27 导入编译环境变量

通过 lunch 命令启动编译，选择 15 号硬件，如图 8-28 所示。

```

favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2/android$ lunch
You're building on Linux
Lunch menu... pick a combo:
  1. full-eng
  2. full_x86-eng
  3. vbox_x86-eng
  4. full_mips-eng

```

图 8-28 选择 15 号硬件进行编译

```

5. full_grouper-userdebug
6. full_tilapia-userdebug
7. mini_armv7a-neon-userdebug
8. mini_armv7a-userdebug
9. mini_mips-userdebug
10. mini_x86-userdebug
11. full_maguro-userdebug
12. full_manta-userdebug
13. full_toroplus-userdebug
14. full_toro-userdebug
15. wing_k70-eng
16. wing_n71j-eng
17. wing_nck70-eng
18. full_panda-userdebug

Which would you like? [full-eng] 15

```

图 8-28 (续)

使用 make 命令进行编译。如果环境以及系统依赖软件都安装成功,在编译过程中就不会出现错误,如图 8-29 所示。在 out 文件夹中就会得到所要的文件。

```

Favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2/android$ extract-bsp
/home/favorming/Desktop/git/pcDuino/android4.2/android/device/*/*wing-k70/bImage copied!
/home/favorming/Desktop/git/pcDuino/android4.2/android/device/*/*wing-k70/modules copied!
Favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2/android$ 
Favorming@favorming-Lenovo:~/Desktop/git/pcDuino/android4.2/android$ make -j 8
=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.2.2
TARGET_PRODUCT=wing_k70
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.13.11.4-x86_64-with-Ubuntu-14.04-trusty
HOST_BUILD_TYPE=release
BUILD_ID=JDQ39
OUT_DIR=out
=====

```

图 8-29 执行 make 命令进行编译

### 8.3.5 烧录镜像

有了一个可用的镜像,此时要做的就是按照官方手册进行烧录。

这里用到的软件是 Phoenix Card 软件。在 pcDuino 的网站能找到这个软件。

将 micro SD 卡插到电脑上,启动软件,选择先前制作好的镜像,待其烧录完毕,就得到一个可用的 micro SD 卡了。

将这张 micro SD 卡插到开发板上,上电开机。由于要往 NAND 闪存中复制相关镜像内容,因此花费时间较多,4 分钟左右。

这里要注意的是,要在 4 分钟之后,先拔下 micro SD 卡,而后再重启设备。

在重启后就进入自己编译的 Android 系统了。

## 8.4 Android LED 驱动设计

本小节将结合 LED 设备的具体实例说明 Android 驱动设计要点，介绍应用程序如何通过 Android 驱动的接口，控制 LED 灯的亮 / 灭。主要工作是：根据硬件原理图了解 LED 灯控制的硬件原理，在此基础上完成 Linux 驱动设计，提供应用层访问的接口；实现 Linux 驱动后，在 Android 的 HAL 层实现对底层驱动的封装，提供与硬件无关的接口给硬件服务层；HAL 的上层为硬件服务层，通过 JNI 接口实现硬件访问服务，并保证硬件的互斥访问；应用 App 调用硬件服务层的接口，实现 LED 灯的控制。

在本小节的学习中，读者将用到 JNI 的相关知识，加深对硬件抽象层的理解。

### 8.4.1 硬件原理

LED 底层硬件结构较简单，CPU 处理器（本实例是在 TI 公司的 DM3730 处理器为基础的实验箱上运行）通过 GPIO 控制 LED 灯。

LED 灯的硬件的连接示意如图 8-30 所示。

GPIO 口分别对应实验箱上的 6 个 LED 灯，可以通过 GPIO 口的输出的电平高低实现点亮或者熄灭 LED 灯，如图 8-31 所示。

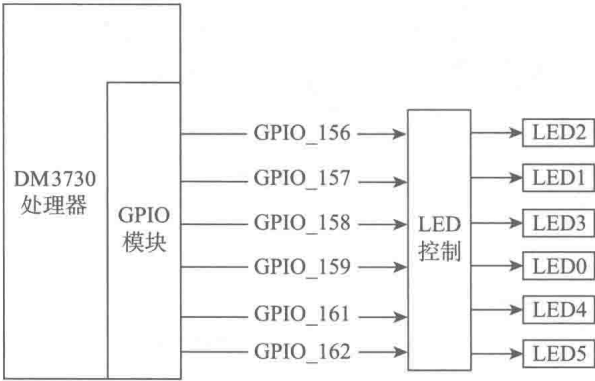


图 8-30 DM3730 与 LED 灯的硬件连接示意图

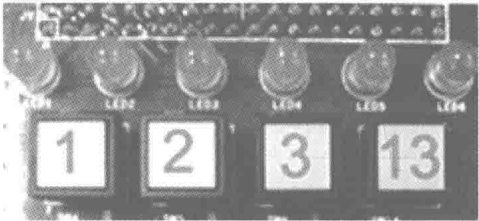


图 8-31 实验箱上 6 个 LED 灯

### 8.4.2 Linux 驱动设计

在嵌入式系统中，大量的外部设备是相对独立的设备。在 Linux-2.6 内核中，加入了虚拟的平台总线（platform\_bus），处理器在逻辑上将所有的独立设备视为连接在虚拟的平台总线上，这些设备被称为平台总线设备。若想实现设备驱动的平台无关性，提高设备驱动对象的可移植性，则需要使用平台总线对象，它减少了设备驱动在新的平台上的移植工作量。本小节将说明如何通过平台总线设备实现对 LED 驱动的封装。

#### 1. LED 平台总线设备对象

Android 内核驱动程序和一般 Linux 内核驱动程序的编写方法是一样的，都是以 Linux

模块的形式实现的。本例将所需的 LED 驱动模块以静态编译的方式编译进内核中。

LED 平台总线设备对象使用 struct platform\_device 结构体描述, 它包含该对象的设备名称以及相关的信息。platform\_device 结构体中 platform\_data 为 gpio\_led\_platform\_data 结构体类型, 该结构体包含 gpio\_led 结构体, 并通过 num\_leds 描述 LED 设备的数量。gpio\_leds 为 gpio\_led 结构体类型的数组, 具体描述 LED 设备的设备节点名、触发方式、GPIO 口等信息。

代码清单8-4

---

```
static struct gpio_led_platform_data gpio_led_info = {
    .leds = gpio_leds,
    .num_leds = ARRAY_SIZE(gpio_leds),
};

static struct platform_device leds_gpio={
    .name = "leds-gpio",
    .id = -1,
    .dev ={
        .platform_data = &gpio_led_info,
    },
};

static struct platform_device *omap3_wscec_devices[] __initdata = {
    &leds_gpio,
    &keys_gpio,
    &wscec_dss_device,
    &usb_mass_storage_device,
    &omap3wscec_dm9000_device,
};
```

---

平台总线设备对象使用 platform\_add\_device() 函数注册。

```
platform_add_devices(omap3_wscec_devices, ARRAY_SIZE(omap3_wscec_devices));
```

了解 LED 总线设备对象的框架后, 要将 LED 加入总线设备。首先在 kernel (文件路径 kernel/arch/arm/mach-omap2/board-omap3wscec.c) 中对 GPIO 端口进行宏定义。

```
#define OMAP3_WSCEC_GPIO_LED3      158
```

同时在该源码中的结构体 static struct gpio\_led gpio\_led[] = {} 中加入 LED 设备描述, 完成对 LED 平台总线设备注册操作。

代码清单8-5

---

```
static struct gpio_led gpio_leds[] = {
    .....
    .....
    .....
    {
```

---

```

        .name           =       "led3" ,
        .default_trigger =       "default-on" ,
        .gpio           =       "OMAP3_WSCEC_GPIO_LED3" ,
        .active_low     =       "false" ,
    },
    .....
    .....
    .....
};

.name           指定leds目录下生成的子目录的名称
.default_trigger 指定触发方式, 这里采用默认
.gpio           指定特定的IO口, 根据实际硬件电路修改
.active_low     当active_low为真时, 对led设置值时, 会进行一个取反操作, 这里active_
                low=false, 则不会进行取反操作

```

---

## 2. LED 平台总线设备驱动分析

平台总线设备驱动对象描述了该设备的驱动, 对应的源码在 kernel 目录下 kernel /driver/ leds 的 leds-gpio.c, 主要完成的功能是注册创建 LED 设备节点与 LED 相关操作函数。

### (1) 内核模块框架

module\_init 宏声明了模块加载函数, module\_exit 声明了模块卸载函数。当设备驱动加载时, 使用 platform\_driver\_register() 函数进行注册, 当驱动卸载时使用 platform\_driver\_unregister() 函数移除。由于本例使用静态编译的模式, 因此并不会对模块进行移除。

代码清单8-6

---

```

static int __init gpio_led_init(void)
{
    int ret = 0;
#ifdef CONFIG_LEDS_GPIO_PLATFORM
    ret = platform_driver_register(&gpio_led_driver);
    if(ret)
        return ret;
#endif
#ifdef CONFIG_LEDS_GPIO_OF
    ret = of_register_platform_driver(&of_gpio_leds_driver);
#endif
#ifdef CONFIG_LEDS_GPIO_PLATFORM
    if (ret)
        platform_driver_unregister(&gpio_led_driver);
#endif
    return ret;
}

static void __exit gpio_led_exit(void)

```



```

{
#ifdef CONFIG_LEDS_GPIO_PLATFORM
platform_driver_unregister(&gpio_led_driver);
#endif
#ifdef CONFIG_LEDS_GPIO_OF
of_unregister_platform_driver(&of_gpio_leds_driver);
#endif
}

module_init(gpio_led_init);
module_exit(gpio_led_exit);

MODULE_AUTHOR("Raphael Assenat <raph@8d.com>, Trent Piepho <tpiepho@freescale.com>")
MODULE_DESCRIPTION("GPIO LED driver");
MODULE_LICENSE("GPL");

```

---

## (2) LED 设备探测与初始化

平台总线设备驱动对象通过 `platform_driver` 结构体进行描述，内嵌 `struct device_driver` 结构体类型的驱动对象 `driver`，若 `driver` 名与总线设备对象名称相同，则进行匹配绑定。

代码清单8-7

```

static struct platform_driver gpio_led_driver = {
    .probe = gpio_led_probe,
    .remove = __devexit_p(gpio_led_remove),
    .driver = {
        .name = "leds-gpio",
        .owner = THIS_MODULE,
    },
};

```

---

当设备和驱动匹配成功后，会运行 `probe` 所指定的函数，本文为 `gpio_led_probe`。为了完成驱动和设备的交互工作，需要从平台总线设备对象中获取其资源信息，并向内核进行注册。设备初始化函数 `gpio_led_probe` 主要完成 LED 的设备初始化工作，例如申请内存空间，创建设备（`create_gpio_led`），保存 LED 设备结构。而 `.remove` 所指定的函数完成的是设备的移除，主要完成了设备的注销以及内存的回收。

代码清单8-8

```

#ifdef CONFIG_LEDS_GPIO_PLATFORM
static int __devinit gpio_led_probe(struct platform_device *pdev)
{
    struct gpio_led_platform_data *pdata = pdev->dev.platform_data;
    struct gpio_led_data *leds_data;
    int i, ret = 0;
    if(!pdata)
        return -EBUSY;

```

```

    leds_data = kzalloc(sizeof(struct gpio_led_data) * pdata->num_leds, GFP_KERNEL);
    if (!leds_data)
        return -ENOMEM;
    printk(KERN_INFO "num_leds = %d\n", pdata->num_leds);
    for (i = 0; i < pdata->num_leds; i++) {
        ret = create_gpio_led(&pdata->leds[i], &leds_data[i], &pdev->dev, pdata-
            >gpio_blink_set);
        if (ret < 0)
            goto err;
    }
    platform_set_drvdata(pdev, leds_data);
    return 0;
err:
    for (i = i-1; i >= 0; i--)
        delete_gpio_led(&leds_data[i]);
    kfree(leds_data);
    return ret;
}

static int __devexit gpio_led_remove(struct platform_device *pdev)
{
    int i;
    struct gpio_led_platform_data *pdata = pdev->dev.platform_data;
    struct gpio_led_data *leds_data;

    leds_data = platform_get_drvdata(pdev);

    for (i = 0; i < pdata->num_leds; i++)
        delete_gpio_led(&leds_data[i]);
    kfree(leds_data);
    return 0;
}

```

---

LED 硬件初始化是在函数 `create_gpio_led()` 中完成的，该函数主要工作如下：

- 1) 对 `gpio` 口的有效性进行判断。
- 2) 申请 `gpio` 口。
- 3) 对设备进行赋值。
- 4) 设置 `IO` 口为输出并指定初值。
- 5) 针对 `gpio_led` 的原子操作。
- 6) 调用 `led_classdev_register` 对设备进行注册。

#### 代码清单8-9

---

```

static int __devinit create_gpio_led(const struct gpio_led *template, struct
    gpio_led_data *leds_data, struct device *parent, int (*blink_set)(unsigned,

```

```

    int , unsigned long *, unsigned long *))
{
    int ret, state;
    led_dat->gpio = -1;
    /* skip leds that aren't available */
    if (!gpio_is_valid(template->gpio)){
        printk(KERN_INFO "Skipping unavailable LED gpio %d (%s)\n", template-
            >gpio, template->name);
        return 0;
    }

    ret = gpio_request(template->gpio, template->name);
    if (ret < 0)
        return ret;

    led_dat->cdev.name = template->name;
    led_dat->cdev.default_trigger = template->default_trigger;
    led_dat->gpio = template->gpio;
    led_dat->can_sleep = gpio_cansleep(template->gpio);
    led_dat->active_low = template->active_low;
    led_dat->blinking = 0;
    if (blink_set){
        led_dat->platform_gpio_blink_set = blink_set;
        led_dat->cdev.blink_set = gpio_blink_set;
    }
    led_dat->cdev.brightness_set = gpio_led_set;
    if (template->default_state == LEDS_GPIO_DEFSTATE_KEEP)
        state = !!gpio_get_value(led_dat->gpio) ^ led_dat->active_low;
    else
        state = (template->default_state == LEDS_GPIO_DEFSTATE_ON);
    led_dat->cdev.brightness_set = state ? LED_FULL : LED_OFF;
    if (ret < 0)
        goto err;

    INIT_WORK(&led_dat->work, gpio_led_work);

    ret = led_classdev_register(parent, &led_dat->cdev);
    if (ret < 0)
        goto err;

    return 0;
err:
    gpio_free(led_dat->gpio);
    return ret;
}

```

---

### (3) LED 设备操作函数

通过 `gpio_led_set` 对相应的 LED 进行点亮或者熄灭操作, `active_low` 为 `value` 值是否取反的 `flag`。

代码清单8-10

```
static void gpio_led_set(struct led_classdev *led_cdev, enum led_brightness value)
{
    struct gpio_led_data *led_dat = container_of(led_cdev, struct gpio_led_data, cdev);
    int level;
    if (value == LED_OFF)
        level = 0;
    else
        level = 1;
    if (led_dat->active_low)
        level = !level;
    /* Setting GPIOs with I2C/etc requires a task context, and we don't seem to
     * have a reliable way to know if we're already in one; so let's just assume the worst.
     */
    if (led_dat->can_sleep){
        led_dat->new_level = level;
        schedule_work(&led_dat->work);
    }
    else {
        if (led_dat->blinking) {
            led_dat->platform_gpio_blink_set(led_dat->gpio, level, NULL, NULL);
            led_dat->blinking = 0;
        }
    }
    else{
        gpio_set_value(led_dat->gpio, level);
    }
}
```

### (4) 编译与测试

在终端输入 `make ARCH = arm CROSS_COMPILE = arm-eabi- menuconfig` 进入配置页面, 将 LED-GPIO 编译进内核, 如图 8-32 所示。

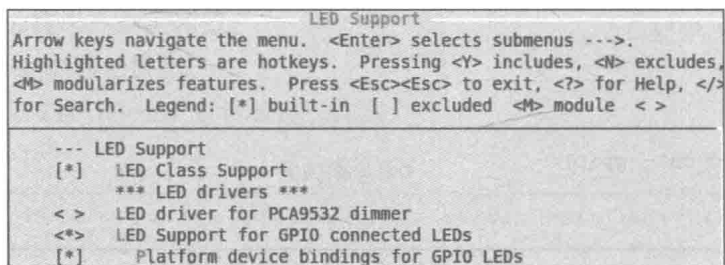


图 8-32 内核配置选项

配置完毕后, 通过 `make ARCH = arm CROSS_COMPILE = arm-eabi- uImage` 编译内核镜像, `uImage` 为 U-boot 可引导的内核镜像格式。通过 `ttip` 下载到实验箱 (DM3730), 进入 `/sys/class` 目录中就可以看到 `leds` 目录, 这个目录是在 Linux 内核创建 LED 设备时由函数 `led_classdev_register(parent, &led_dat->cdev)` 创建的。

代码清单8-11

---

```
root@android:/sys/class # ls
.....
i2c-adapter
i2c-dev
input
lcd
leds
mdio_bus
Mem
.....
```

---

进入任意 LED 设备目录 (如 `led3` 目录) 可以看到属性值 `brightness`。

代码清单8-12

---

```
root@android:/sys/class/leds # cd led3
root@android:/sys/class/leds/led3 # ls
brightness
device
max_brightness
power
subsystem
uevent
```

---

访问和修改属性文件 `brightness` 的值, 就可以控制 LED 灯的亮与灭。

代码清单8-13

---

```
root@android:/sys/class/leds/led3 # cat brightness
0
root@android:/sys/class/leds/led3 # echo 1 >brightness
root@android:/sys/class/leds/led3 # cat brightness
1
```

---

此时, `led3` 灯亮, 如图 8-33 所示。

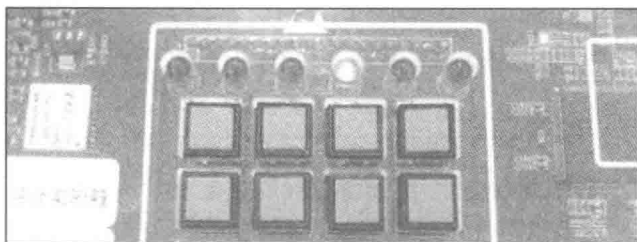


图 8-33 实验箱 LED 灯亮

至此，LED 的驱动部分在 Linux 内核层的实现已经完成，在 Linux 系统上，可以访问并控制 6 个 LED 灯的亮灭。

### 8.4.3 Android HAL 层驱动

LED 的 HAL 层驱动位于 Linux 的用户空间，HAL 对底层驱动进行封装，提供与硬件无关的接口给硬件服务层。在 7.3.2 节中介绍的 `hw_module_t` 和 `hw_device_t` 为 HAL 层的重要结构体，针对 LED 设备而言，当 Stub 被加载时 `hw_module_t` 提供 LED 设备的初始化操作，比如 `open` 操作。`hw_device_t` 提供 LED 设备必要的操作接口，比如点亮或者熄灭。LED 的 HAL 层代码会编译成动态链接库，当 Service 调用 `hw_get_module` 时会寻找 LED 的动态链接库，并提供相应的接口。

对于 HAL 层的工作，第一步要编写一个代理，即 Stub，其主要作用是根据特定的设备，使用代理完成工作，它是调用 Linux 驱动的第一层。

#### 1. 头文件 led.h

`led.h` 的主要作用是对结构体进行封装并定义 LED 模块的 ID。该文件放在源码根目录下的 `hardware/libhardware/include/hardware` 子目录下。

代码清单8-14

---

```

struct led_moudle_t{
    struct hw_moudle_t common;
};

struct led_control_device_t {
    struct hw_moudle_t common;
    /*attributes*/
    int fd;

    /* supporting control APIs go here */
    int (*set_on)(struct led_control_device_t *dev, int32_t led);
    int (*set_off)(struct led_control_device_t *dev, int32_t led);
};

struct led_control_context_t
{
    struct led_control_device_t device;
};

__END_DECLS
#endif
#define LED_HARDWARE_MODULE_ID "led"

```

---

`led_module_t` 对 `hw_module_t` 进行封装，套用 C++ 的说法是进行了“继承”。在 HAL 中，每个硬件都使用 `hw_module_t` 进行描述，记录本 Stub 的基本信息和 `open` 函数指针。

`led_control_device_t` 对 `hw_device_t` 进行封装与扩展。`fd` 为文件句柄，供 `open` 函数指针

打开后返回使用。set\_on 和 set\_off 为两个操作接口，分别对应点亮与熄灭 LED 灯。

led\_control\_context\_t 对 led\_control\_device\_t 进行分装，作为上下文结构。

LED\_HARDWARE\_MODULE\_ID 定义 HAL 模块的 ID，作为唯一识别标志。

## 2. 函数源码 led.c

led.c 中实现了 HAL 层的向下操作的接口，以及提供给上层硬件服务层的接口，这部分将被编译成动态链接库。进入 hardware/libhardware/modules 目录，新建 led 目录，并添加 led.c 文件。led.c 主要分为 3 大块：入口定义、open 与 close 和操作函数。通过入口定义可以找到特定的 Stub，而 open 与 close 函数实现打开与关闭设备，操作函数则规定了具体操作方法接口。

### (1) 入口定义

定义一个 led\_module\_t 结构体类型的 HAL\_MODULE\_INFO\_SYM(必须使用这个名称)，对这个变量进行初始化，最主要的是 id 与 methods 这两部分。

id 作为该设备的唯一标识符，上层的 Service 通过 id 可以找到该 Stub。而 methods 则定义了入口函数，结构体 led\_module\_methods 完成 open 函数指针指向函数 led\_device\_open 的操作。

代码清单 8-15

---

```
static struct hw_moudle_methods_t led_module_methods = {
    open: led_device_open
};

const struct led_moudle_t HAL_MOUDLE_INFO_SYM = {
    common: {
        tag: HARDWARE_MODULE_TAG,
        version_major: 1,
        version_minor: 1,
        id: LED_HARDWARE_MODULE_ID,
        name: "Sample LED Stub",
        author: "The xmu Open Source Project",
        methods: &led_module_methods,
    }
};
```

---

### (2) open 与 close 函数

led\_device\_open 函数主要完成的工作是：定义 hw\_device\_t 继承类的 dev 变量，并对其初始化操作。其中

- 1) close 指定了关联的关闭接口，即 led\_device\_close。
- 2) dev->set\_on = led\_on 指定了关联的操作接口，即点亮 LED 灯。
- 3) dev->set\_off = led\_off 指定了关联的操作接口，即关闭 LED 灯。

`*device = &dev->common` 则是对二级指针 `device` 初始化其指向的内容。

代码清单8-16

---

```
static int led_device_open(const struct hw_moudle_t* module, const char* name,
struct hw_device_t** device)
{
    struct led_control_device_t* dev;

    dev = (struct led_control_device_t*)malloc(sizeof(*dev));

    if(!dev)
    {
        LOGE("LED Stub:failed to alloc space");
        return -EFAULT;
    }
    memset(dev, 0, sizeof(*dev));

    dev->common.tag = HARDWARE_MODULE_TAG;
    dev->common.version = 0;
    dev->common.module = module;
    dev->common.close = led_device_close;

    dev->set_on = led_on;
    dev->set_off = led_off;

    if ((dev->fd = open(DEVICE_NAME, O_RDWR)) == -1)
    {
        LOGI("LED Stub:failed open file brightness in leds ---%s.",strerror(errno));
        free(dev);
        return -EFAULT;
    }

    *device = &dev->common;

    success:
    LOGI("LED Stub:open file brightness in led successfully.");
    return 0;
}
```

---

代码清单8-17

---

`led_device_close`完成了对设备的释放

```
int led_device_close(struct hw_device_t* device)
{
    struct led_control_device_t* ctx = (struct led_control_device_t*)device;
    if(ctx)
```



```

{
    free(ctx);
    close(ctx->fd);
}
return 0;
}

```

### (3) 操作接口 led\_on 与 led\_off

led\_on 与 led\_off 为操作接口函数，完成的功能为对设备写字符类型 ‘1’ 或者字符类型 ‘0’，即实现点亮或者关闭 LED 灯。

代码清单8-18

```

int led_on(struct led_control_device_t* dev,int32_t led)
{
    char ledOn = '1';
    write(dev->fd, &ledOn, 1);
    LOGI("LED Stub:set %d on.", led);
    return 0;
}

int led_off(struct led_control_device_t* dev, int32_t led)
{
    char ledOff = '0';
    write(dev->fd, &ledOff, 1);
    LOGI("LED Stub: set %d off.", led);
    return 0;
}

```

## 3. 编译规则 Android.mk

在 led 目录下新建编译规则文件 Android.mk，它规定如何将此模块编译成 .so 文件。

代码清单8-19

```

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

LOCAL_MODULE_TAGS := optional
LOCAL_PRELINK_MODULE := false
LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
LOCAL_SHARED_LIBRARIES := liblog
LOCAL_SRC_FILES := led.c
LOCAL_MODULE := led.default_state
include $(BUILD_SHARED_LIBRARY)

```

说明如下。

1) LOCAL\_MODULE\_TAG：该宏指定 Android 版本，有 user、eng、test 等，optional 表示该模块在所有版本都编译。

2) `LOCAL_PRELINK_MODULE`: 该宏指定是否使用事先链接模式, 该模式可以加快启动速度。`false` 表示不使用。

3) `LOCAL_MODULE_PATH`: 该宏指定目标安装路径。

4) `LOCAL_SHARED_LIBRARIES`: 该宏指定需要链接的动态库。

5) `LOCAL_MODULE`: 该宏指定目标库的库名。

编译成功后, 可以在 `out/target/product/wscecboard/system/lib/hw` 目录下看到 `led.default.so` 文件。

至此, Android 系统为自己的硬件增加了一个硬件抽象层模块, 但是现在 Java 应用程序还不能访问到此硬件。必须修改 Framework 层, 才能在 Java 应用程序中访问自己定制的硬件。

#### 8.4.4 硬件服务层

通过上面的学习, 对于 Android 系统的硬件驱动程序实现已经有所了解, 包括 Linux 驱动程序设计和用户空间实现硬件抽象层的接口。实现这两者的目的是为了给 Android 的 Application Framework 层提供硬件服务。众所周知, Android 的应用程序是通过 Java 语言实现的, 而底层的硬件驱动是通过 C 语言实现的, 两者之间的调用方式是通过前面介绍的 JNI 接口。为了让 Java 应用程序调用硬件抽象层接口, 硬件服务程序将 JNI 接口对 Android 硬件抽象层的功能进行封装, 提供给 Java 应用程序使用。接下来将介绍如何实现 LED 设备的硬件服务程序。

进入源码目录, 新建 JNI 接口文件 `com_android_server_LedService.cpp`, (`frameworks/base/services/jni`)。这里注意文件命名的方法, `com_android_server` 前缀表示的是包名, 表示硬件服务的 `LedService` 是放在 `frameworks/base/services/java` 目录下的 `com/android/server` 中, 即存在一个名为 `com.android.server.LedService` 的类。

##### 1. JNI 接口

JNI 接口主要实现 Native 函数, 通过 `JNINativeMethod` 完成 JNI 接口方法表的实现, 最后通过 `jniRegisterNativeMethods` 完成 JNI 的注册。

##### (1) Native 函数实现

Native 函数为 `xmu_init`、`xmu_setOn` 和 `xmu_setOff`。其中 `xmu_setOn` 和 `xmu_setOff` 较为简单, 只是通过调用 HAL 层对应的函数即可。这里主要分析 `xmu_init` 的实现。

在 Framework 层的 Java 文件 (`LedService.java` 后面会分析), 构造函数 `public LedService()` 会通过 `_init` 接口调用本函数 `xmu_init(JNIEnv* env, jclass clazz)` 来完成本模块的初始化工作。`xmu_init` 中通过 `hw_get_module` 打开设备 id 为 `LED_HARDWARE_MODULE_ID` 的硬件设备, 获得 LED Stub 的句柄并保存在 `module` 变量中。

`led_control_open` 通过该函数调用 Stub 中的 `open` 接口, 并将设备句柄保存到 `sLedDevice`,

完成了设备的初始化工作。

代码清单8-20

---

```
static inline int led_control_open(const struct hw_moudle_t* module, struct led_
    control_device_t** device)
{
    return module->methods->open(module, LED_HARDWARE_MODULE_ID, (struct hw_device_t**)
        device);
}

static jboolean xmu_init(JNIE* env, jclass clazz)
{
    led_moudle_t* module;
    LOGI("LedService JNI: Initialing.....");

    if(hw_get_module(LED_HARDWARE_MODULE_ID, (const hw_moudle_t**)&module)== 0)
    {
        LOGI("LedService JNI:Led Stub found.");
        if(led_control_open(&module->common, &sLedDevice) ==0)
        {
            LOGI("LedService JNI:Got Stub operations.");
            return 0;
        }
        LOGI("LedService JNI:failed to open led device.");
        return -1;
    }
    LOGI("LedService JNI:Get Stub operations failed.");
    return -1;
}
```

---

## (2) JNI 接口定义

JNINativeMethods 是 JNI 层的注册方法表, Framework 层则可以使用这些方法。具体解释如下:

- `_init`、`_set_on` 和 `_set_off` 是 Framework 层调用时的方法名。
- `()Z` 表示无参数且返回值为 boolean 型。
- `xmu_init`、`xmu_setOn` 和 `xmu_setOff` 为对应的 Native 函数名。

代码清单8-21

---

```
/*
 * Array of methods
 * Each entry has three filed: the name of the method, the methodsignature, and
 * a pointer to the native *implementation.
 */
static const JNINativeMethod gMethods[] = {
    { "_init", "()Z", (void *)xmu_init},
    { "_set_on", "()Z", (void *)xmu_setOn},
```

---

```
{ "_set_off",      "()Z",      (void *)xmu_setOff},
};
```

### (3) JNI 注册本地方法

Register\_android\_server\_LedService 调用 jniRegisterNativeMethod 向类中注册本 .so 中的 native 接口，而具体的接口定义在 gMethods 中。

### 2. 注册 JNI 接口

onload.cpp 在目录 framework/base/service/jni/onload.cpp 中，在里面添加 register\_android\_server\_LedService 函数的声明与调用。当系统加载 libandroid\_servers 模块时，会调用 onload.cpp 中的 JNI\_OnLoad 函数，将定义的 JNI 接口注册到 Java 虚拟机 (Dalvik) 中。

首先在 namespace android 中增加 register\_android\_server\_LedService 函数声明。

代码清单8-22

```
namespace android{
int register_android_server_AlarmManagerService(JNIEnv* env);
int register_android_server_BatteryService(JNIEnv* env);
int register_android_server_InputApplicationHandle(JNIEnv* env);
int register_android_server_InputWindowHandle(JNIEnv* env);
int register_android_server_InputManager(JNIEnv* env);
int register_android_server_LightsService(JNIEnv* env);
int register_android_server_PowerManagerService(JNIEnv* env);
int register_android_server_UsbDeviceManager(JNIEnv* env);
int register_android_server_UsbHostManager(JNIEnv* env);
int register_android_server_VibratorService(JNIEnv* env);
int register_android_server_SystemServer(JNIEnv* env);
int register_android_server_location_GpsLocationProvider(JNIEnv* env);
int register_android_server_connectivity_Vpn(JNIEnv* env);
int register_android_server_LedService(JNIEnv* env);    //add
};
```

在 JNI\_onLoad 中增加 register\_android\_server\_LedService 函数调用。这样，在 Android 系统初始化时，就会自动加载该 JNI 方法调用表。

代码清单8-23

```
register_android_server_PowerManagerService(env);
register_android_server_InputApplicationHandle(env);
register_android_server_InputWindowHandle(env);
register_android_server_InputManager(env);
register_android_server_LightsService(env);
register_android_server_AlarmManagerService(env);
register_android_server_BatteryService(env);
register_android_server_UsbDeviceManager(env);
register_android_server_UsbHostManager(env);
```

```

register_android_server_VibratorService(env);
register_android_server_SystemServer(env);
register_android_server_location_GpsLocationProvider();
register_android_server_connectivity_Vpn();
//add
register_android_server_LedService(env);

```

修改同目录下的 Android.mk 文件，在 LOCAL\_SRC\_FILES 中加入对 LedService 源代码的编译。

代码清单8-24

```

LOCAL_SRC_FILES:= \
com_android_server_AlarmManagerService.cpp \
com_android_server_BatteryService.cpp \
com_android_server_InputApplicationHandle.cpp \
com_android_server_InputManager.cpp \
com_android_server_InputWindowHandle.cpp \
com_android_server_LightsService.cpp \
com_android_server_PowerManagerService.cpp \
com_android_server_SystemServer.cpp \
com_android_server_UsbDeviceManager.cpp \
com_android_server_UsbHostManager.cpp \
com_android_server_VibratorService.cpp \
com_android_server_location_GpsLocationProvider.cpp \
com_android_server_connectivity_Vpn.cpp \
com_android_server_LedService.cpp \
onload.cpp

```

在 Android 源码目录下，输入 `mmm frameworks/base/services/jni` 进行编译，则完成了 JNI 接口的实现。

### 3. 硬件服务接口定义

硬件抽象的目的是为了使得最上层的应用程序能够使用这些硬件提供的服务，对 Android 系统上的应用软件来说，就是要在系统的 Application Frameworks 层为其提供硬件服务。前面介绍的 Linux 内核层、硬件抽象层提供的自定义硬件服务接口，这些接口都是通过 C 或者 C++ 语言来实现的。Android 系统的应用框架层（Application Frameworks）提供 Java 接口的硬件服务，硬件服务一般是运行在一个独立的进程中为各种应用程序提供服务，调用这些硬件服务的应用程序与这些硬件服务之间的通信需要通过代理来进行，为此，需要定义好相应的通信接口。

进入 `frameworks/base/core/java/android/os` 目录，新增 `ILedService.aidl` 接口定义文件，`ILedService` 接口主要提供了设备和获取硬件寄存器 `val` 值的功能，分别通过 `setOn` 和 `setOff` 两个函数来实现。

代码清单8-25

---

```
package android.os;
interface ILedService
{
    boolean setOn(int led);
    boolean setOff(int led);
}
```

---

返回 frameworks/base 目录，打开编译规则脚本文件 Android.mk 文件，修改 LOCAL\_SRC\_FILES 变量的值，增加 ILedService.aidl。

代码清单8-26

---

```
core/java/android/os/IPowerManager.aidl \
core/java/android/os/IRemoteCallback.aidl \
core/java/android/os/IVibratorService.aidl \
core/java/android/os/ILedService.aidl \
```

---

编译 ILedService.aidl 接口，生成相应的 Java 代码接口 (ILedService.Stub)。然后，进入目录 frameworks/base/services/java/com/android/server 中，新增文件 LedService.java。构造函数通过 \_init 函数指针调用 xmu\_init 函数，setOn 和 setOff 分别调用 xmu\_setOn 和 xmu\_setOff 接口，最后通过 native 关键字声明。

代码清单8-27

---

```
package com.android.server;
import android.util.Config;
import android.util.Log;
import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.RemoteException;
import android.IBinder;
import android.os.ILedService;

public final class LedService extends ILedService.Stub {
    public LedService() {
        Log.i("LedService", "Go to get Led Stub.");
        _init();
    }
    /*
     *xmu_LED native methods
     */
    public boolean setOn(int led){
        Log.i("xmuplatform", "Led On");
        return _set_on(led);
    }
}
```

---

```

public boolean setOff(int led){
    Log.i("xmuplatform", "Led Off");
    return _set_off(led);
}

private static native boolean _init();
private static native boolean _set_on(int led);
private static native boolean _set_off(int led);
}

```

修改同目录下的 SystemServer.java 文件，在 ServerThread::run 函数中增加加载 LedService 的代码。

代码清单8-28

```

try{
    Slog.i(TAG, "DiskStats Service");
    ServiceManager.addService("diskstats", new DiskStatsService(context));
}catch(Throwable e){
    Slog.e(TAG, "Failure starting DiskStats Service", e);
}

try {
    Slog.i(TAG, "Led Service");
    ServiceManager.addService("diskstars", new LedService());
}catch (Throwable e){
    Slog.e(TAG, "Fail starting Led Service", e);
}

```

完成以上步骤后，便可以通过 mmm 命令编译生成 LED 硬件访问服务。由于制作的文件系统需要对属性文件 brightness 进行读写，而该设备文件是由 device\_create 创建的，因此默认只有 root 用户可以进行读写，而 led\_device\_open 一般由上层应用调用，因此不具有 root 权限则无法对设备进行读写。解决的办法是在创建的文件系统的 init.rc 中增加以下一行：

```
chmod 666 /sys/class/leds/led3/brightness
```

将 brightness 修改为任何用户都具有读写的权限。

### 8.4.5 App 应用编写

更高层的设计是使用 Java 语言进行的 Application 层的开发。新建一个工程 LedClient，列出其主要的源码文件：src/com/xmu/ledclient/LedClient.java。程序通过 ServiceManager.getService("led") 来获得 java 层服务 LedService，通过 ILedService.Stub.asInterface 函数转换为 ILedService 接口。其中，服务名 led 是系统启动加载 LedService 时指定的。ILedService 接口定义在 android.os.ILedService 中，这个程序提供了简单的读写自定义硬件，有寄存器 val

的值的功能，通过 `ILedService.setOn` 和 `ILedService.setOff` 两个接口实现。

代码清单8-29

---

```

public class LedClient extends Activity implements OnClickListener
{
    private static final String LOG_TAG = "Xmu Led Service";
    private ILedService ledService = null;

    private TextView msg = null;
    private Button ledOn = null;
    private Button ledOff = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_led_client);
        ledService = ILedService.Stub.asInterface(
            ServiceManager.getService("led"));

        msg = (TextView) findViewById(R.id.tv1);
        ledOn = (Button) findViewById(R.id.bt1);
        ledOff = (Button) findViewById(R.id.bt2);

        ledOn.setOnClickListener(this);
        ledOff.setOnClickListener(this);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.led_client, menu);
        return true;
    }
}

```

---

接口监听 `Button`，当 `Button` 被按下时，执行相应的动作。

代码清单8-30

---

```

@Override
public void onClick(View v) {
    // TODO Auto-generated method stub
    if(v.equals(ledOn)) {
        try {
            ledService.setOn(3);
            msg.setText("Led 3 is On");
        } catch (RemoteException e) {
            Log.e(LOG_TAG, "Remote Exception while reading value from device.");
        }
    }
}

```

---



```

    }
    else if(v.equals(ledOff)) {
        try {
            ledService.setOff(3);
            msg.setText("Led 3 is Off.");
        } catch (RemoteException e) {
            Log.e(LOG_TAG, "Remote Exception while writing value to device.");
        }
    }
}
}
}

```

在实验箱上运行 App 后，按下 Button 点亮 LED 灯。

## 8.5 进阶讲解——针对 Android 系统的内核跟踪与测试

Android 操作系统除了商业领域的应用外，也可以作为一个很好的嵌入式系统教学平台。可以利用 Android 系统进行内核移植、驱动设计、内核定值等系统级开发。而在进行系统级开发时，总会遇到各种问题，比如一些不易排查的错误，或者是需要对系统性能做一些评估。由于 Android 内核与 Linux 内核有相似之处，也有不同之处，因此需要一款好的工具来解决这些问题。ftrace 作为一个系统内核跟踪工具，可以用来对 Android 内核进行进程、函数、堆栈等的追踪。本节将主要介绍在 ARM 平台运行 Android 内核，以及利用 ftrace 工具中的 3 个 tracer 对 Android 内核进行追踪的例子及其分析。

### 8.5.1 使用平台简介

#### 1. 模拟器平台

本文进行测试和使用的内核是 Google 所提供的 Goldfish 版本。使用 Goldfish 的步骤如下：

- 1) 从 Google 官方 git 仓库下载 goldfish 的分支，目前最新的版本是 Android-1.6 (icecake)。
- 2) 配置 Android 的开发环境，包括安装 Eclipse，添加插件 ADT、SDK，修改 Java 的环境变量等。
- 3) 编译下载下来的内核，期间会遇到不少问题，需要打一些补丁，最终完成编译。
- 4) 在 Android 自己的 Emulator 虚拟机上使用编译好的 zImage 了。

在实际测试中，虚拟机的使用存在一个 bug，即在进行 tracer 挂起时，不进行活动，因此本文使用真实的平台来运行 Android kernel。

#### 2. BeagleBoard

除了虚拟的平台，本文还将使用一个真实的 ARM 平台。

运行 Android Kernel 的 ARM 平台，选择 BeagleBoard XM 这款开发板。BeagleBoard 是

由 TI 与 Digi-Key 共同推出的, 具有功能强大、成本低廉、便携等优点, 目前已经在开源教育领域得到了应用和推广。这款平台基于 TI 的 OMAP3 处理器, 这是 ARM Cortex A8 的核心, 处理能力属于目前 ARM 系列的主流。

BeagleBoard 的主要硬件性能如下:

1) 处理器为 DM3730, 内含主频 1GHz 的 ARM Cortex A8, 主频 800MHz 的 TMS320C64+。

2) 512MB 的 LPDDR 内存, 采用 POP 封装。这使得板子的结构极为紧凑, 尺寸仅为 85mm×85mm。

3) 周边接口丰富, 而且有丰富的开源支持。在其官方主页上, 列出了很多基于此平台的开源项目, 相关资源非常多。

采用 BeagleBoard 的开源项目很多, 无论是 WinCE、Ubuntu、Symbian 或者 Android, 还是至今尚未正式部署的 Meego, 都有基于 BeagleBoard 的项目。Google Code 上有两个相关的 Android 项目, 一个是 0xdroid, 另一个是 Rowboat。其中 0xdroid 是由源于台湾地区的 0xlab 提供解决方案。

本文选择 Rowboat。目前最新的稳定版是基于 Gingerbread (Android 2.3) 的。在 TI 的官方主页上可以下载的 OMAP3 的 Android 镜像就是由 Rowboat-Gingerbread 编译得来的。与 0xdroid 相比较, Rowboat 的优点在于项目开发活跃度高, 能够紧跟 Google 的脚步。就全球范围而言, 使用 Rowboat 的用户也范围更广、人数更多。

## 8.5.2 测试环境的建立

下面以 32 位的 Ubuntu 11.10 操作系统为例, 说明如何在 Ubuntu 系统下获取 Android 内核、源码、编译内核, 以及搭建 ARM 测试平台。

Android 源码 (以 Android 2.3 为例) 的获取有两种方法, 一种是从 Android Developer 网站获得, 一种是从 TI 公司官网获得。

### 1. 从 Android 官网获得源码和内核

1) 需要提前安装的文件包有: git-core, nex, bison, gperf, lib8dl-dev, libesd0-dev, libwxgtk2.6-dev, build-essential, zip, curl。

2) 建立工作目录, 并且添加到环境变量的 \$PATH 中。

3) 下载 repo 脚本, 并修改其权限为可执行。

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

4) 建立一个存放源码的目录并进入该目录, 用如下命令同步 URL 清单, 它指定了包括 Android 各种相关文件的资料库。

```
repo init -u https://android.googlesource.com/platform/manifest
```

5) 查找需要版本的分支, 用 `-b` 命令。

```
repo init -u https://android.googlesource.com/platform/manifest -b android-2.3 (这里需要Android 2.3版本, 即gingerbread版本的源码)
```

6) 获取文件。Repo sync 将源码文件同步到本地 (需要一定的时间, 一般为 1 小时)。

7) 获得内核, 需要 Android\_OMAP 内核。

```
git clone https://android.googlesource.com/kernel/omap.git
```

8) 使用 git 工具将内核下载到本地。

## 2. 从 TI 官网获得内核和源码

在 TI 的官方网站的 Digital Signal Processors and ARM Microprocessors 部分能够找到 Android 开发组件。

选择 Android 2.3 Development Kit, 地址如下:

```
http://software-dl.ti.com/dsps/dsps_public_sw/sdo_tii/TI_Android_DevKit/TI_Android_GingerBread_2_3_4_DevKit_2_1/index_FDS.html
```

也可以用 repo 脚本, 从 TI 官网的 git 仓库同步源码及内核。

## 3. 编译 Android

### (1) 准备交叉编译链

交叉编译, 就是在一个平台上生成另一个平台上的可执行代码。平台包含了两个概念: 一是体系结构 (Architecture), 二是操作系统 (Operating System)。一般来说, 在同一个体系结构下可以运行不同的操作系统, 且同一个操作系统也可以在不同的体系结构上运行。通常目标平台的存储和运算能力是有限的, 所以进行编译就会显得很困难。再加上编译工具链 (compilation tool-chain) 需要很大的存储空间和很强的 CPU 运算能力, 所以在 ARM 平台本机编译十分困难。交叉工具链应运而生, 通过交叉编译工具链, 可以达到在 CPU 能力够强、存储空间足够大的主机平台上 (比如在 PC 机器上) 编译出针对其他平台的可执行程序的目的。

Google 公司提供了用于 Android 的交叉编译工具链。在源码的 prebuilt/linux-x86/toolchain/ 文件夹中有不同版本的交叉编译链。为此, 需要先将 Toolchain 的路径添加到环境变量中去。

命令如下:

```
$ export PATH=<DIR >/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin:$PATH
```

### (2) 编译 X-loader

X-loader 在 CPU 下主要完成了常规的 ARM 内核的初始化, 在 board 下的 omapbebm 下完成了对总线、时钟的初始化。

X-loader 还完成了 OneNand 的相应初始化, 并且把 Uboot 从 NAND 架构下复制到

BufrrtRam 下，再复制到 SDRAM 中。

X-loader 还支持从 SD、MMC 下引导 Uboot，它在 CPU 下有对 MMC 的初始化程序，可以直接把 uboot.bin 从 MMC 下复制到 SDRAM 中。

编译时使用的配置文件为 omap3beagle\_config，使用 make 命令编译。

代码清单8-31

---

```
$ make ARCH=arm CROSS_COMPILE=arm-eabi- distclean
$ make ARCH=arm CROSS_COMPILE=arm-eabi- omap3beagle_config
$ make ARCH=arm CROSS_COMPILE=arm-eabi-
```

---

编译后，利用 OMAP 提供的工具 signGP 将生成的 x-load.bin 转换成 MLO 文件 ./signGP ./x-load.bin。

### (3) 编译 u-boot

U-boot 的功能主要是系统引导、挂载文件系统、支持目标板存储、操作系统接口、设备驱动、CRC 校验等。它是 ARM 平台启动的必要步骤。

编译时使用的配置文件是 omap3\_beagle\_config。与上例相似，采用 make 语句编译，成功后会生成 u-boot.ini 文件。

### (4) 编译 kernel

编译 kernel 使用的配置文件是 omap3\_beagle\_android\_defconfig，命令如下：

```
$ make ARCH=arm CROSS_COMPILE=arm-eabi- uImage
```

在这个过程中，可能出现一些错误，大都是由于配置不当造成的。可以在网上搜索解决方法。

### (5) 编译文件系统

进入 TI\_Android\_GingerBread\_2\_3\_4Sources 目录，执行以下命令开始编译生成 Android 文件系统：

```
$ make TARGET_PRODUCT=beagleboard OMAPES=5.x -j8
```

其中，如果是 Beagleboard Rev Cx，参数为 3.x；如果是 Beagleboard XM A/B/C，参数为 5.x。本例所使用的型号是 Beagleboard XM A，所以用 5.x 作为参数。

编译需要很长时间，需要耐心的等待。

### (6) 打包 Android 文件系统

编译完成后进入 out/target/product/beagleboard 目录，新建一个目录（本例是 android\_rootfs）将需要的文件复制进去，然后通过 mktarball.sh 命令将文件系统打包。

代码清单8-32

---

```
$ cd out/target/product/beagleboard
$ mkdir android_rootfs
$ cp -r root/* android_rootfs
```

---

```
$ cp -r system android_rootfs
$ sudo ../../../../../../build/tools/mktarball.sh
../../../../host/linux-x86/bin/fs_get_stats android_rootfs . rootfs rootfs.tar.bz2
```

### (7) 生成 SDcard

最后一步是制作启动的 SD 卡，SD 卡大小为 2G ~ 4G，通过读卡器连接到 Ubuntu 上，挂载后找到设备名，如 /dev/sdc。

然后新建一个目录将需要的文件都复制进去，包括 MLO、u-boot.bin、boot.scr、uImage、root.tar.bz2 和 media\_clips 目录。

其中 MLO、u-boot.bin、uImage、root.tar.bz2 都是由以上步骤生成的，不包括 boot.scr 文件和 media\_clips 目录。

boot.scr 文件可以通过 mkbootscr 生成。

新建目标文件夹，所使用的命令如下：

#### 代码清单8-33

```
$ cp kernel/arch/arm/boot/uImage image_folder
$ cp u-boot/u-boot.bin image_folder
$ cp x-loader/MLO image_folder
$ cp Tools/mk-bootscr/boot.scr image_folder
$ cp out/target/product/omap3evm/rootfs.tar.bz2 image_folder
```

另外将 Devkit 中的 mkmmc-android.sh 命令复制到新建的目录中。

```
$ cp Tools/mk-mmc/mkmmc-android.sh image_folder
```

最后，执行命令 mkmmc-android，格式如下：

```
$ ./mkmmc-android <SD卡设备名，如/dev/sdb>
MLO u-boot.bin uImage boot.scr rootfs.tar.bz2 Media_Clips
```

SD 卡将被格式化，并进行重新分区，同时复制了相关文件。

到这里，一个可以启动的 MMC 就制作完成，将 SD 卡插入 Beagleboard 开发板，启动，即完成了在 ARM 平台上运行的 Android 内核的前期准备任务。

## 8.5.3 测试工具

测试 Android 系统的方法多种多样，本文的重点介绍内核性能的测试和分析，所以选用 ftrace 这个工具。ftrace 作为一款实时调试系统，能够有效地辅助开发人员定位系统的问题，找到瓶颈。

### 1. ftrace 简介

ftrace 是一个 Linux 内核提供的追踪工具。使用 ftrace 可以调试内核，或者分析内核中发生的问题。ftrace 提供了不同种类的追踪器，可以进行多种操作。比如跟踪内核函数调用、

对上下文跳转进行跟踪、查看中断被关闭的时长、跟踪内核态中的延迟以及性能等。通过 ftrace，开发者可以对内核进行跟踪调试，从而找到内核中出现问题的根源，方便对问题进行修复。

ftrace 有两大组成部分：框架（framework）和各式各样的追踪器（tracer）。tracer 负责完成功能，framework 负责对 tracer 进行管理。ftrace 的 trace 信息保存在 ring buffer 中，由 framework 负责管理。framework 利用 debugfs 系统在 /debugfs 下建立 tracing 目录，并提供了一系列的控制文件。

ftrace 利用 GCC 的 profile 特性，在所有内核函数的开始部分加入一段 stub 代码，ftrace 重载这段代码来实现 trace 功能。

大致工作过程如图 8-34 所示。

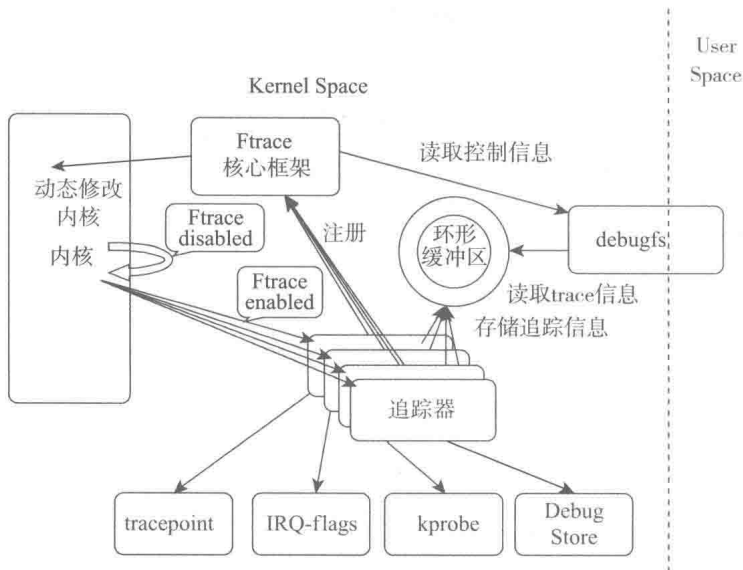


图 8-34 ftrace 工作流程

ftrace 本质上是静态代码插装技术，所以它不需要对某种编程接口支持，而让用户自定义 trace 行为。静态代码插装技术的优点在于它的可靠性，可以很大程度上避免了因为用户的不当使用而导致的内核崩溃。

主要的 tracer 主要有以下几项。

- Function tracer 和 Function graph tracer：跟踪函数调用。
- Schedule switch tracer：跟踪进程调度情况。
- Wakeup tracer：跟踪进程的调度延迟，即跟踪高优先级进程从进入 ready 状态，到获得 CPU 的延迟所用时间。该 tracer 只针对实时进程。
- Irqsoff tracer：在中断被禁止时，系统无法响应外部事件，时钟也无法产生中断，这意味着系统的响应延迟。通过使用 Irqsoff，可以跟踪到内核中哪些函数禁止了中断，

哪个函数对于其中断禁止时间最长，并且对这一信息进行记录，从而使开发者可以迅速定位造成响应延迟的问题所在。

- **Preemptoff tracer**：此 tracer 可以跟踪并记录禁止内核抢占的函数，并显示禁止抢占时间最长的那个内核函数。
- **Kernel memory tracer**：主要功能是跟踪 slab allocator 的分配情况。包括对 kfree、kmem\_cache\_alloc 等 API 的调用情况的追踪、记录。开发者可以根据 tracer 收集到的信息分析内部碎片情况，找出内存分配最频繁的代码片断等。
- **Workqueue statistical tracer**：是一个静态的 tracer，用于统计系统中所有的 workqueue 的工作情况。比如有多少个 work 被插入 workqueue，多少个已经被执行等。可以帮助开发者决定是使用 single threaded workqueue 还是 per-cpu workqueue。

Event tracer：跟踪系统事件，比如 timer、系统调用、中断等。

这里没有列出所有的 tracer。ftrace 是目前非常活跃的开发领域，新的 tracer 还在不断被加入内核。

## 2. ftrace 的使用

### (1) 让内核支持追踪

要使用 ftrace，先要将其编译进内核中。使用 make menuconfig 打开图形化配置界面，在 kernel hacking → Tracers 中选择，如图 8-35 所示。

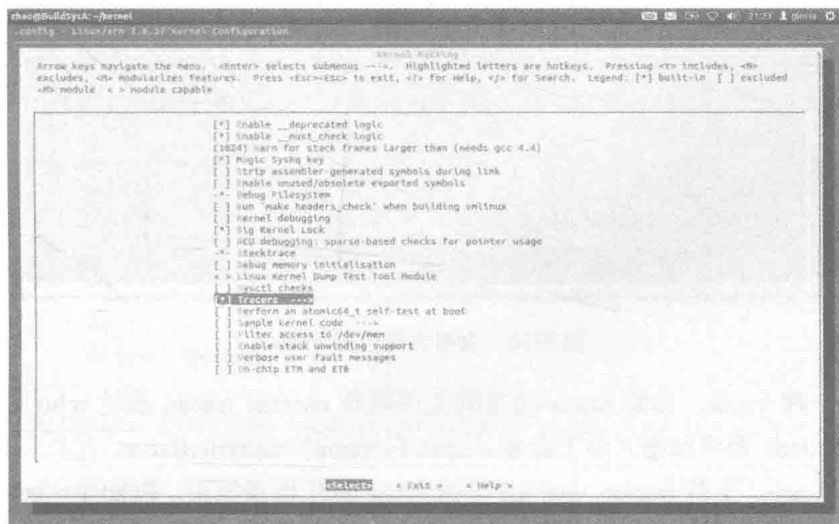


图 8-35 Kernel-Tracers 编译选项

可以看到 Tracers 菜单中的选项都是内核支持的 tracers。选中需要的 tracer，这里选中 function 和 sched\_switch。

在这里要特别注意，对于 32 位的 x86 机器，要去掉 Optimize for size 选项（在 general

setup 菜单下)。但遗憾的是,本例所使用的这个内核版本不支持 Function Graph Tracer 这一功能。

ftrace 通过 debugfs 向用户态提供了访问接口,所以需要在编译时将 debugfs 编译进内核。同样的,在 kernel hacking 目录下选中对 debugfs 文件系统的支持。激活 ftrace 后,编译内核时会使用编译器的 -pg 选项。-pg 选项会在编译得到的内核映像中加入大量的调试信息,所以一般用于调试阶段,最终用于发行版的内核会关闭 -pg 选项,这也意味着无法使用 ftrace。

## (2) 使用 ftrace 的步骤

首先需要挂载 debugfs,通常是将其挂载在 /sys/kernel 目录之下(另外一种常见的是挂载在 /mnt 目录下)。执行以下命令:

```
mount -t debugfs none /sys/kernel/debug
```

进入 /sys/kernel 目录下,会发现一个 tracing 文件夹。通过 cat available\_tracers 命令,查看当前可以选用的 tracer,如图 8-36 所示。



图 8-36 查看当前可用 tracer

1) 选择一种 tracer。选择 tracer 的控制文件叫作 current\_tracer,通过 echo 写入。比如要选择 function tracer,则可以键入如下命令: echo function > current\_tracer。

2) 使能 ftrace。文件 tracing\_enabled 控制 ftrace 的开始和结束。Echo 1 > tracing\_enabled,即打开追踪。

3) 执行需要追踪的应用程序,比如需要跟踪命令 cat 的执行情况,就运行 cat。

4) 关闭 ftrace: Echo 0 > tracing\_enabled。

5) 查看 trace 文件。使用 cat 查看,可以用 cat trace > /mnt/trace.txt 导出 trace 文件。

ftrace 的输出信息被保存在以下 3 个文件中:



- trace, 保存 ftrace 的输出信息, 可以通过 cat 直接查看, 或者导出后查看。
- latency\_trace, 主要保存系统中有关中断时间延迟的信息。
- trace\_pipe 是一个“管道”, 可以通过重定向, 方便应用程序读取 trace 的内容。

到此为止, 已经大致介绍了所要使用的工具 ftrace, 具体的使用及结果将在下面介绍。

## 8.5.4 Android 内核调试与性能测试

### 1. 使用 ftrace 调试 Android 内核模块

使用 function tracer 可以跟踪记录到函数的调度过程。通过文件 set\_ftrace\_filter 可以指定要跟踪的函数, 缺省目标为所有可跟踪的内核函数; 可以将感兴趣的函数通过 echo 写入该文件。为了方便使用, set\_ftrace\_filter 文件还支持简单格式的通配符。

在下面的内核模块中本文设计一个“除零错误”, 并使用 Ftrace 对这个错误进行跟踪。

#### (1) 模块设计

内核模块在 /sys/kernel/ 下创建了 kobj4ftrace 文件夹, kobj4ftrace 下有一个 data 文件。

读者可以向 data 输入一个整数 (echo 3 > data), 然后读出 data 的值,  $24/3 = 8$ 。

由于在内核模块中没有检查分母是否为零, 因此当输入 0, 再读出结果时, 将会触发一个“除零错误”, 即无法输出正常结果。

#### (2) 生成模块

命名这个文件为 test.c, 代码主要如下:

代码清单8-34

---

```
static int data = 0;
int devide(int data)
{
    return 24/data;
}
.....
my_kobj = kobject_create_and_add("kobj4ftrace", kernel_kobj);
if (!my_kobj)
    return -ENOMEM;
.....
/* Create the files associated with this kobject */
retval = sysfs_create_group(my_kobj, &attr_group);
if (retval)
    kobject_put(my_kobj);
return retval;
```

---

通过交叉编译链将 test.c 编译为模块 test.ko, 将 test.ko 复制到 MMC 的 /rootfs 目录下。

#### (3) 通过 Minicom 操作

打开 MMC 控制端, 即进入 Minicom, 在这里加上参数 -C, 保存运行中的日志 log。

挂载 debugfs 文件系统, 导入模块 test.ko

```
$mount -t debugfs nodev /mnt
$insmod test.ko
```

插入模块成功，可以通过 `lsmod` 命令查看。

#### (4) trace 步骤

1) 先设置 `set_trace_filter`。

```
echo ":mod:test" > set_ftrace_filter
```

2) 用 `cat` 指令查看是否设置成功，结果如下：

```
devide
caculater
data_store
data_show
```

可以看到，指定跟踪模块 `test` 中的函数将文件 `set_ftrace_filter` 的内容设置为只包含该模块中的函数。

3) 选择一个 tracer，在此选择 `function`，即 `echo function > current_tracer`。

使能 tracer，`echo 1 > tracing_enabled`。

转到 `/sys/kernel/kobj4ftrace` 目录下，先给 `data` 赋正常数值，比如 2，除法结果为  $24/2 = 12$ 。此时可以得到正常的 `trace` 结果，可以看到是完整的调用过程。

再给 `data` 赋值 0，此时除法出错。进行 `trace` 可以看到如下结果：

```
-----
0)  bash-3234      =>    cat-3688
-----
0)  9.898 us      |  data_show();
-----
0)  cat-3688      =>    bash-3234
-----
0)  8.028 us      |  data_store();
-----
0)  bash-3234      =>    cat-3689
-----
0) |  data_show() {
-----
```

此时调用到了 `data_show` 函数即停止。

#### (5) 结果分析

通过此例可以看到，在用 `function tracer` 时，可以跟踪到函数的调用情况，从而追踪执行过程。通过对模块的追踪，可以筛选出目标模块的各函数执行情况，从而判断出出错情况。

## 2. 使用 ftrace 测试系统性能

### (1) 使用 sched\_switch 追踪系统调度

sched\_switch 跟踪器可以对进程的调度切换以及唤醒操作进行跟踪。

使用 seched\_switch 和上例类似，将 sched\_switch 写入 current\_tracer 文件即可。

在 sched\_switch 跟踪器获取的跟踪信息中，记录了进程间的唤醒操作和调度切换信息，可以通过符号 ‘+’ 和 ‘==>’ 区分；唤醒操作记录给出了当前进程唤醒运行的进程，进程调度切换记录中显示了接替当前进程运行的后续进程。

跟踪结果的部分如下所示：

```
tracer: sched_switch
#
# TASK-PID      CPU#    TIMESTAMP      FUNCTION
#      ||        |           |
sh-850    [000]    405.958313:    850:120:S + [000] 56: 49: S irq/74-serial 1
sh-850    [000]    405.958313:    850:120:S + [000] 850: 120: S sh
sh-850    [000]    405.958344:    850:120:R + [000] 3: 120: S ksoftirqd/0
sh-850    [000]    405.958344:    850:120:R ==>[000] 56: 49: R
```

描述进程状态的格式为 Task-PID:Priority:Task-State，并且

Previous task                  Next Task

<pid>:<prio>:<state> ==> <pid>:<prio>:<state>

关于 state 状态，有以下几个参数。

- R - running: 在运行 / 想要运行 (可能不是正在运行)。
- S - sleep: 可中断睡眠进程。
- D - disk sleep: 不可中断睡眠进程 (与信号无关)。
- T - stopped: 挂起。
- t - traced: 被追踪的进程。
- Z - zombie: 进程将要被清理掉。

以示例跟踪信息中的第一条跟踪记录为例，可以看到进程 sh 的 PID 为 850，其对应的内核态优先级为 120，当前状态为 S (可中断睡眠状态)，当前 bash 并没有唤醒其他进程；从第 3 条记录可以看到，进程 sh 将被进程 ksoftirqd/0 唤醒，而在第 4 条记录中发生了进程调度，进程 sh 切换到进程 ksoftirqd/0 执行。

在 Linux 内核中，进程的状态在内核头文件 include/linux/sched.h 中定义，包括可运行状态 TASK\_RUNNING (对应跟踪信息中的符号 R)、可中断阻塞状态 TASK\_INTERRUPTIBLE (对应跟踪信息中的符号 S) 等。同时该头文件也定义了用户态进程所使用的优先级的范围，最小值为 MAX\_USER\_RT\_PRIO (值为 100)，最大值为 MAX\_PRIO - 1 (对应值为 139)，缺省为 DEFAULT\_PRIO (值为 120)

同时，将结果文件转换成 .vcd 格式，可以利用开源工具 GTKwave 查看调度的时序图，

如图 8-37 所示。

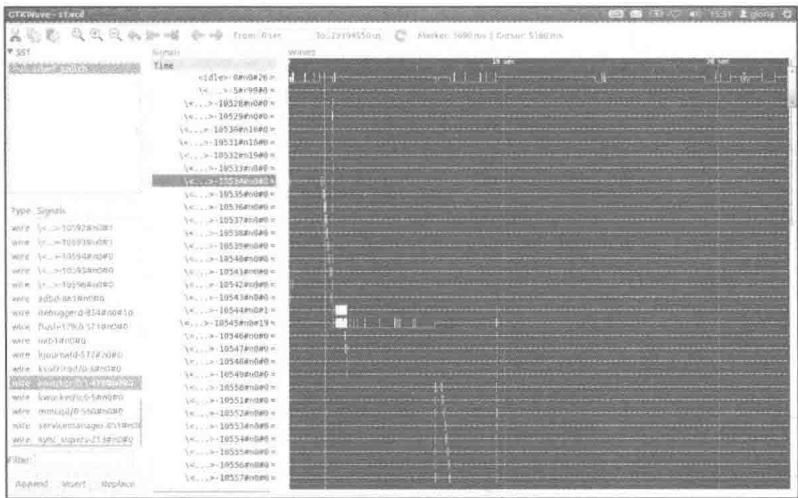


图 8-37 利用开源工具 GTKWave 查看调度的时序图

图 8-37 左侧列出各进程参数，可以通过添加显示在右侧。图 8-37 为调度时序图，可以进行放大、比较。

从输出结果来看，内核现有的调度策略，能够正确的实现优先级调度。

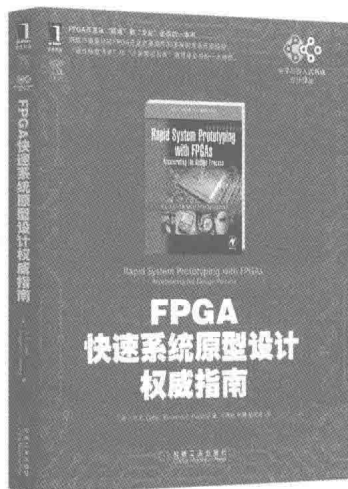
(2) 查看系统中断关闭延迟

当关闭中断时，CPU 会延迟对设备的状态变化做出反应，有时候这样做会对系统性能造成比较大的影响。irqsoff 跟踪器可以对中断被关闭的状况进行跟踪，有助于发现导致较大延迟的代码；当出现最大延迟时，跟踪器会记录导致延迟的跟踪信息，文件 tracing\_max\_latency 则记录中断被关闭的最大延时。下面列出追踪部分结果：

```
mmcqd/0-560      0d..1.    0us : blk_update_request <-blk_update_bidi_request
mmcqd/0-560      0d..2.    0us+: disk_map_sector_rcu <-blk_update_request
mmcqd/0-560      0d..1.    31us : req_bio_endio <-blk_update_request
mmcqd/0-560      0d..1.    31us : bio_endio <-req_bio_endio
mmcqd/0-560      0d..1.    31us : end_bio_bh_io_sync <-bio_endio
mmcqd/0-560      0d..1.    31us : journal_end_buffer_io_sync <-end_bio_bh_io_sync
mmcqd/0-560      0d..1.    31us : unlock_buffer <-journal_end_buffer_io_sync
mmcqd/0-560      0d..1.    31us : wake_up_bit <-unlock_buffer
mmcqd/0-560      0d..1.    31us : bit_waitqueue <-wake_up_bit
mmcqd/0-560      0d..1.    31us : __wake_up_bit <-wake_up_bit
mmcqd/0-560      0d..1.    31us : bio_put <-end_bio_bh_io_sync
mmcqd/0-560      0d..1.    31us : bio_fs_destructor <-bio_put
mmcqd/0-560      0d..1.    31us+: bio_free <-bio_fs_destructor
mmcqd/0-560      0d..1.    61us : mempool_free <-bio_free
mmcqd/0-560      0d..1.    61us : mempool_free_slab <-mempool_free
mmcqd/0-560      0d..1.    61us : kmem_cache_free <-mempool_free_slab
mmcqd/0-560      0d..1.    61us : req_bio_endio <-blk_update_request
```

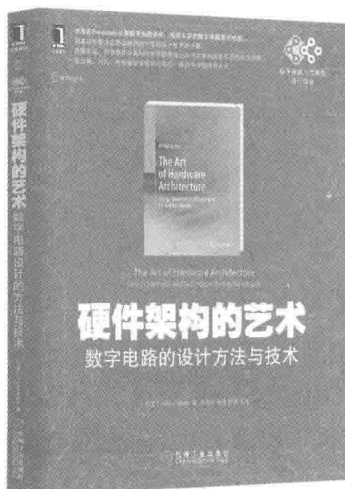


## 推荐阅读



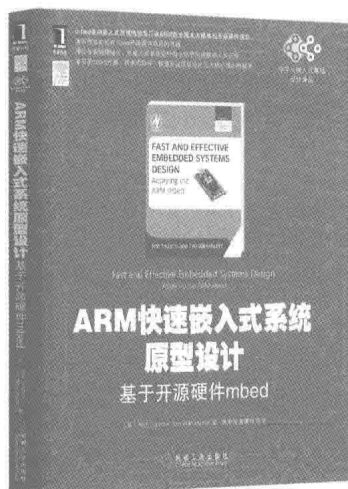
### FPGA快速系统原型设计权威指南

作者: R.C. Cofer 等 ISBN: 978-7-111-44851-8 定价: 69.00元



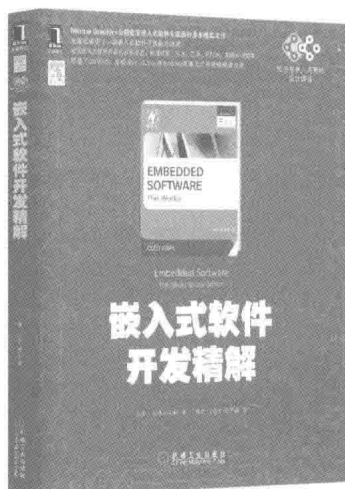
### 硬件架构的艺术: 数字电路的设计方法与技术

作者: Mohit Arora ISBN: 978-7-111-44939-3 定价: 59.00元



### ARM快速嵌入式系统原型设计: 基于开源硬件mbed

作者: Rob Toulson 等 ISBN: 978-7-111-46019-0 定价: 69.00元



### 嵌入式软件开发精解

作者: Colin Walls ISBN: 978-7-111-44952-2 定价: 79.00元

## 推荐阅读



### Arduino 高级开发权威指南 (原书第2版)

作者: Steven F. Barrett ISBN: 978-7-111-45246-1 定价: 59.00元



### 例说XBee无线模块开发

作者: Jonathan A. Tifus ISBN: 978-7-111-45681-0 定价: 59.00元



### Arduino 与LabVIEW 开发实战

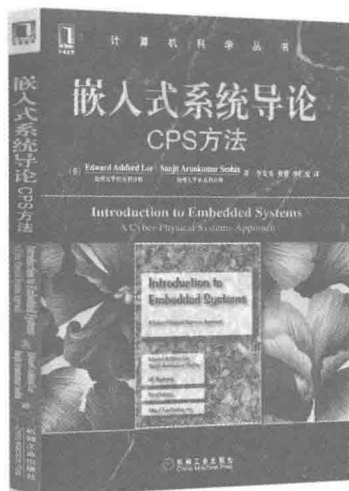
作者: 沈金鑫 ISBN: 978-7-111-45839-5 定价: 59.00元



### Arduino 开发实战指南: STM32篇

作者: 姚汉 ISBN: 978-7-111-44582-1 定价: 59.00元

## 推荐阅读



### 嵌入式系统导论：CPS方法

作者：Edward Ashford Lee 等 ISBN: 978-7-111-36021-6 定价：55.00元



### 嵌入式计算系统设计原理（第2版）

作者：Wayne Wolf ISBN: 978-7-111-27068-3 定价：55.00元



### 嵌入式微控制器与处理器设计

作者：Greg Osborn ISBN: 978-7-111-32281-8 定价：59.00元



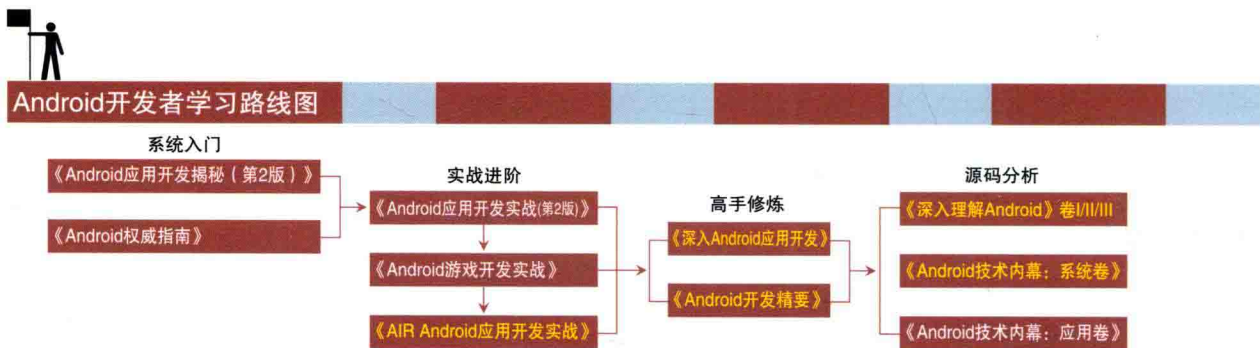
### 计算机组成与设计：硬件/软件接口（原书第4版）

作者：David A. Patterson 等 ISBN: 978-7-111-35305-8 定价：99.00元



## 掌握Android底层软件开发必备的工具书

开发一个Android应用往往非常简单,而要构建一个健壮、可靠、功耗和性能优异的软硬件一体系统确非易事。因此,市面上介绍Android应用开发的书籍很多,但是偏重底层软件开发的书却不多。Android如今已经是最为流行的嵌入式操作系统,如果想要从零开始定制一个全新的基于Android和ARM处理器的新系统,就必须要了解Android操作系统的底层软件。本书不仅覆盖了必要的嵌入式系统理论和Linux的基础知识,还对Android底层的源码进行分析,并用实例引导读者掌握底层开发的技能和调试方法。本书是掌握Android底层软件开发技术的必备工具书。



投稿热线: (010) 88379604  
客服热线: (010) 88379426 88361066  
购书热线: (010) 68326294 88379649 68995259

华章网站: [www.hzbook.com](http://www.hzbook.com)  
网上购书: [www.china-pub.com](http://www.china-pub.com)  
数字阅读: [www.hzmedia.com.cn](http://www.hzmedia.com.cn)



[General Information]

书名=Android底层开发实战

作者=周庆国，郑灵翔，康筱彬等编著

页数=246

SS号=13877507

DX号=

出版日期=2015.10

出版社=北京机械工业出版社

封面  
书名  
版权  
前言  
目录

## 第1章 Android嵌入式系统导论

### 1.1 Android嵌入式系统概述

#### 1.1.1 嵌入式系统定义

#### 1.1.2 基于Android的嵌入式系统构成

#### 1.1.3 移动电话系统

#### 1.1.4 基于ARM的移动电话硬件结构

### 1.2 嵌入式系统实例

#### 1.2.1 pcDuino部分硬件功能介绍

#### 1.2.2 基于Android的嵌入式系统

## 第2章 Linux系统详解

### 2.1 系统简介

### 2.2 基础命令

#### 2.2.1 cd和ls命令

#### 2.2.2 touch和mkdir命令

#### 2.2.3 rm和rmdir命令

#### 2.2.4 cp和mv命令

#### 2.2.5 find和awk命令

#### 2.2.6 vim编辑器的使用

### 2.3 Bash Shell

#### 2.3.1 Bash Shell简介

#### 2.3.2 Bash Shell脚本简介

### 2.4 Linux源码与Android源码介绍

#### 2.4.1 Linux源码简介

#### 2.4.2 Android源码简介

## 第3章 Android系统开发环境搭建

### 3.1 编译前奏——Android上的开发工作

#### 3.1.1 Android的移植开发

#### 3.1.2 系统开发

#### 3.1.3 应用开发

### 3.2 Android的系统架构

#### 3.2.1 软件结构

#### 3.2.2 源代码的结构

### 3.3 搭建开发环境

3.3.1 搭建编译环境

3.3.2 使用repo

3.3.3 Android的编译

## 第4章 Android系统底层源码结构分析

### 4.1 源码结构分析

4.1.1 底层库结构介绍

4.1.2 C基础函数库bionic

4.1.3 C语言底层库libcutils

4.1.4 C++工具库libutils

4.1.5 底层文件系统库system

4.1.6 增加本地库的方法

### 4.2 Android编译系统介绍

4.2.1 build系统

4.2.2 SDK

### 4.3 init初始化脚本语言介绍

4.3.1 概述

4.3.2 init进程源码分析

4.3.3 脚本文件的创建与分析

4.3.4 创建设备节点文件

4.3.5 子进程的创建与终止

4.3.6 属性服务

### 4.4 Zygote

4.4.1 Zygote概述

4.4.2 AppRuntime分析

4.4.3 system server分析

## 第5章 Android系统内核分析

### 5.1 Linux内核基础

5.1.1 概述

5.1.2 Linux内核的主要子系统

5.1.3 Linux启动过程分析

### 5.2 Android内核概况

### 5.3 Android启动过程分析

### 5.4 Binder框架分析

5.4.1 概述

5.4.2 Binder的系统架构

5.4.3 Binder的机制和原理

### 5.5 Ashmem内存管理方式

5.5.1 概述

- 5.5.2 Ashmem初始化
  - 5.5.3 内存的创建和释放
  - 5.5.4 内存的映射
  - 5.5.5 内存的锁定和解锁
- 5.6 低内存管理
- 5.7 Logger
  - 5.7.1 Logger概述
  - 5.7.2 Logger实现原理
- 第6章 Android系统相关工具及运行环境
  - 6.1 Android开发工具分类及介绍
    - 6.1.1 应用程序开发工具
    - 6.1.2 框架开发工具
    - 6.1.3 交叉编译工具
    - 6.1.4 内核开发工具
  - 6.2 Dalvik虚拟机
    - 6.2.1 概述
    - 6.2.2 dex文件
    - 6.2.3 Dalvik内存管理
    - 6.2.4 Dalvik编译器
  - 6.3 JNI
    - 6.3.1 概述
    - 6.3.2 JNI的架构
    - 6.3.3 JNI的实现方式
  - 6.4 Boot Loader
    - 6.4.1 概述
    - 6.4.2 Boot Loader的操作模式
    - 6.4.3 启动过程
  - 6.5 busybox的使用
- 第7章 Android驱动程序设计
  - 7.1 Android驱动概述
  - 7.2 Android NDK编程
  - 7.3 Android系统中的HAL层
    - 7.3.1 HAL\_legacy和HAL对比
    - 7.3.2 HAL module架构分析
    - 7.3.3 HAL实现流程
  - 7.4 Android系统Camera与WiFi实现
    - 7.4.1 Android中的Camera实现
    - 7.4.2 Android系统WiFi实现

## 第8章 Android底层开发实例讲解

- 8.1 底层开发相关技术概览
- 8.2 实例讲解——基于Zynq的Android移植
  - 8.2.1 主机开发环境的搭建
  - 8.2.2 Linux内核的编译
  - 8.2.3 Android文件系统的编译
  - 8.2.4 SD卡的准备以及Android系统的启动
- 8.3 移植讲解——基于pcDuino的Android移植
  - 8.3.1 pcDuino介绍
  - 8.3.2 环境搭建
  - 8.3.3 编译内核
  - 8.3.4 编译Android
  - 8.3.5 烧录镜像
- 8.4 Android LED驱动设计
  - 8.4.1 硬件原理
  - 8.4.2 Linux驱动设计
  - 8.4.3 Android HAL层驱动
  - 8.4.4 硬件服务层
  - 8.4.5 App应用编写
- 8.5 进阶讲解——针对Android系统的内核跟踪与测试
  - 8.5.1 使用平台简介
  - 8.5.2 测试环境的建立
  - 8.5.3 测试工具
  - 8.5.4 Android内核调试与性能测试

封底