

揭示安卓驱动具体实现过程 分享驱动开发的移植技巧

Broadview
www.broadview.com.cn



Android

底层开发技术实战详解

——内核、移植和驱动

王振丽 编著



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

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

内 容 简 介

本书从底层原理开始讲起,结合真实的案例向读者详细介绍了 Android 内核、移植和驱动开发的整个流程。全书分为 19 章,依次讲解驱动移植的必要性,何为 HAL 层深入分析,Goldfish、MSM、MAP 内核和驱动解析,显示系统、输入系统、振动器系统、音频系统、视频输出系统的驱动,OpenMax 多媒体、多媒体插件框架,传感器、照相机、Wi-Fi、蓝牙、GPS 和电话系统等。在每一章中,重点介绍了与 Android 驱动开发相关的底层知识,并对 Android 源码进行了剖析。

本书适合 Android 研发人员及 Android 爱好者学习,也可以作为相关培训学校和大专院校相关专业的教学用书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Android 底层开发技术实战详解:内核、移植和驱动/王振丽编著. —北京:电子工业出版社,2012.8
(Android 移动开发技术丛书)
ISBN 978-7-121-17593-0

I. ①A… II. ①王… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字(2012)第 157997 号

策划编辑:张月萍

责任编辑:高洪霞

印 刷:

装 订:

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱

邮编:100036

开 本:787×1092 1/16

印张:33.75

字数:864 千字

印 次:2012 年 8 月第 1 次印刷

印 数:3500 册 定价:69.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn,盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线:(010) 88258888。

前言

随着 3G 的到来，无线带宽越来越高，使得更多内容丰富的应用程序装入手机成为可能，如视频通话、视频点播、移动互联网冲浪和内容分享等。为了承载这些数据应用及快速部署，手机功能将会越来越智能，越来越开放。为了实现这些需求，必须有一个好的开发平台来支持，在此由 Google 公司发起的 OHA 联盟走在了业界的前列，2007 年 11 月推出了开放的 Android 平台，任何公司及个人都可以免费获取源代码及开发 SDK。由于其开放性和优异性，Android 平台得到了业界广泛的支持，其中包括各大手机厂商和著名的移动运营商等。继 2008 年 9 月第一款基于 Android 平台的手机 G1 发布之后，三星、摩托罗拉、索爱、LG 等主流手机制造商都推出了自己的 Android 平台手机。在 2011 年底，Android 超越了塞班和 iOS，雄踞智能手机市场占有率榜首的位置。

毕竟 Android 平台被推出的时间才短短 5 年，了解 Android 平台软件开发技术的程序员还不多，如何迅速地推广和普及 Android 平台软件开发技术，让越来越多的人参与到 Android 应用的开发中，是整个产业链都在关注的一个话题。为了帮助开发者更快地进入 Android 开发行列，笔者特意精心编写了本书。本书系统讲解了 Android 底层驱动开发和移植的基本知识，图文并茂地帮助读者学习和掌握各种驱动的开发常识，详细讲解了 Android 源代码的方方面面。

从技术角度而言，Android 是一种融入了全部 Web 应用的平台。随着版本的更新，从最初的触屏到现在的多点触摸，从普通的联系人到现在的数据同步，从简单的 Google Map 到现在的导航系统，从基本的网页浏览到现在的 HTML 5，这都说明 Android 已经逐渐稳定，而且功能越来越强大。此外，Android 平台不仅支持 Java、C、C++ 等主流的编程语言，还支持 Ruby、Python 等脚本语言，甚至 Google 专为 Android 的应用开发推出了 Simple 语言，这使得 Android 有着非常广泛的开发群体。

本书的内容

在本书的内容中，详细讲解了 Android 底层技术和驱动开发的基本知识。本书内容新颖、知识全面、讲解详细，全书分为 19 章，具体内容分布如下：

章	主要内容
第 1 章	什么是驱动以及 Linux 内核源代码简单剖析
第 2 章	搭建 Linux 开发环境，分析及编译 Android 源代码，在 Linux 环境下运行模拟器
第 3 章	Android 移植的内容、驱动开发所要完成的任务，三种类型的驱动程序
第 4 章	传感器在 HAL 层的表现，HAL 层的源代码与移植
第 5 章	Goldfish 下的 staging、Ashmen、Pmem、Alarm 和 Android Paranoid 驱动
第 6 章	MSM 内核与 MSM 的移植
第 7 章	OMAP 内核与移植
第 8 章	显示系统的移植、调试与驱动程序实现
第 9 章	MSM 处理器和 OMAP 处理器平台中输入驱动的实现
第 10 章	振动器的系统结构与移植



(续表)

章	主要内容
第 11 章	音频系统的层次、移植与不同平台下的实现
第 12 章	视频输出系统 Overlay 的分析、实现、调用
第 13 章	OpenMax 多媒体框架的层次和实现
第 14 章	OpenCore 引擎和 Stagefright 引擎的代码结构与扩展
第 15 章	传感器系统的结构、移植与实现
第 16 章	照相机系统的结构、移植与实现
第 17 章	Wi-Fi 系统、蓝牙系统和定位系统的移植
第 18 章	开发电话系统
第 19 章	Alarm 警报器系统、Lights 光系统和 Battery 系统的移植与实现

全书内容都采用了理论加实践的教学方法，每个实例先提出制作思路及包含知识点，在实例最后补充总结知识点并出题让读者举一反三。

本书特色

本书内容相当丰富，实例内容覆盖全面，满足 Android 技术人员成长道路上的方方面面。我们的目标是通过一本图书，提供多本图书的价值，读者可以根据自己的需要有选择地阅读，以完善本人的知识和技能结构。在内容的编写上，本书具有以下特色：

(1) 结构合理

从用户的实际需要出发，科学安排知识结构，内容由浅入深，叙述清楚，具有很强的知识性和实用性，反映了当前 Android 技术的发展和水平。同时全书精心筛选的最具代表性、读者最关心典型知识点，几乎包括 Android 底层和驱动技术的各个方面。

(2) 易学易懂

本书条理清晰、语言简洁，可帮助读者快速掌握每个知识点；每个部分既相互连贯又自成体系，使读者既可以按照本书编排的章节顺序进行学习，也可以根据自己的需求对某一章节进行针对性的学习。

(3) 实用性强

本书彻底摒弃枯燥的理论和简单的操作，注重实用性和可操作性，详细讲解了各个部分的源代码知识，使用户在掌握相关操作技能的同时，还能学到相应的基础知识。

参加本书编写的人员有：王振丽、王东华、熊斌、朱桂英、周秀、邓才兵、罗红仙、王石磊、孙宇、程娟、王文忠、王梦、陈强、于洋、管西京。本团队由于时间和水平所限，书中难免有不足之处。如有纰漏和不尽如人意之处，诚请读者提出意见或建议，以便修订并使之更臻完善。另外，为了更好地为读者服务，我们专门提供了技术支持网站 www.topchuban.com，欢迎读者光临论坛，无论是书中的疑问，还是学习过程中的疑惑，本团队将尽力为大家解答。

编 者

2012 年 5 月

目 录

第 1 章	Android 底层开发基础	1	2.3.3	运行 Android 源代码	42
1.1	什么是驱动	1	2.3.4	实践演练——演示编译 Android 程序的两种方法	43
1.1.1	驱动程序的魅力	1	2.4	编译 Android Kernel	47
1.1.2	电脑中的驱动	2	2.4.1	获取 Goldfish 内核代码	47
1.1.3	手机中的驱动程序	2	2.4.2	获取 MSM 内核代码	50
1.2	开源还是不开源的问题	3	2.4.3	获取 OMAP 内核代码	50
1.2.1	雾里看花的开源	3	2.4.4	编译 Android 的 Linux 内核	50
1.2.2	从为什么选择 Java 谈为什么不 开源驱动程序	3	2.5	运行模拟器	52
1.2.3	对驱动开发者来说是一把双刃剑 ...	4	2.5.1	Linux 环境下运行模拟器的方法	53
1.3	Android 和 Linux	4	2.5.2	模拟器辅助工具——adb	54
1.3.1	Linux 简介	5	第 3 章	驱动需要移植	57
1.3.2	Android 和 Linux 的关系	5	3.1	驱动开发需要做的工作	57
1.4	简析 Linux 内核	8	3.2	Android 移植	59
1.4.1	内核的体系结构	8	3.2.1	移植的任务	60
1.4.2	和 Android 密切相关的 Linux 内核知识	10	3.2.2	移植的内容	60
1.5	分析 Linux 内核源代码很有必要 ...	14	3.2.3	驱动开发的任务	61
1.5.1	源代码目录结构	14	3.3	Android 对 Linux 的改造	61
1.5.2	浏览源代码的工具	16	3.3.1	Android 对 Linux 内核文件的 改动	62
1.5.3	为什么用汇编语言编写内核 代码	17	3.3.2	为 Android 构建 Linux 的操作 系统	63
1.5.4	Linux 内核的显著特性	18	3.4	内核空间和用户空间接口是一 个媒介	64
1.5.5	学习 Linux 内核的方法	26	3.4.1	内核空间和用户空间的相互 作用	64
第 2 章	分析 Android 源代码	31	3.4.2	系统和硬件之间的交互	64
2.1	搭建 Linux 开发环境和工具	31	3.4.3	使用 Relay 实现内核到用户空 间的数据传输	66
2.1.1	搭建 Linux 开发环境	31	3.5	三类驱动程序	70
2.1.2	设置环境变量	32	3.5.1	字符设备驱动程序	70
2.1.3	安装编译工具	32	3.5.2	块设备驱动程序	79
2.2	获取 Android 源代码	33			
2.3	分析并编译 Android 源代码	35			
2.3.1	Android 源代码的结构	35			
2.3.2	编译 Android 源代码	40			



3.5.3	网络设备驱动程序.....	82	5.7	Android Paranoid 驱动程序	153
第 4 章	HAL 层深入分析	84	5.8	Goldfish 设备驱动.....	154
4.1	认识 HAL 层.....	84	5.8.1	FrameBuffer 驱动	155
4.1.1	HAL 层的发展	84	5.8.2	键盘驱动	159
4.1.2	过去和现在的区别.....	86	5.8.3	实时时钟驱动程序	160
4.2	分析 HAL 层源代码.....	86	5.8.4	TTY 终端驱动程序.....	161
4.2.1	分析 HAL moudle	86	5.8.5	NandFlash 驱动程序	162
4.2.2	分析 mokoid 工程	89	5.8.6	MMC 驱动程序.....	162
4.3	总结 HAL 层的使用方法.....	98	5.8.7	电池驱动程序	162
4.4	传感器在 HAL 层的表现.....	101	第 6 章	MSM 内核和驱动解析	164
4.4.1	HAL 层的 Sensor 代码	102	6.1	MSM 基础	164
4.4.2	总结 Sensor 编程的流程	104	6.1.1	常见 MSM 处理器产品	164
4.4.3	分析 Sensor 源代码看 Android API 与硬件平台的衔接	104	6.1.2	Snapdragon 内核介绍	165
4.5	移植总结.....	116	6.2	移植 MSM 内核简介	166
4.5.1	移植各个 Android 部件的方式... ..	116	6.3	移植 MSM	168
4.5.2	移植技巧之一——不得不说的 辅助工作	117	6.3.1	Makefile 文件.....	168
第 5 章	Goldfish 下的驱动解析	125	6.3.2	驱动和组件.....	170
5.1	staging 驱动	125	6.3.3	设备驱动	172
5.1.1	staging 驱动概述	125	6.3.4	高通特有的组件.....	174
5.1.2	Binder 驱动程序	126	第 7 章	OMAP 内核和驱动解析	177
5.1.3	Logger 驱动程序	135	7.1	OMAP 基础	177
5.1.4	Lowmemorykiller 组件	136	7.1.1	OMAP 简析	177
5.1.5	Timed Output 驱动程序	137	7.1.2	常见 OMAP 处理器产品	177
5.1.6	Timed Gpio 驱动程序.....	139	7.1.3	开发平台	178
5.1.7	Ram Console 驱动程序.....	139	7.2	OMAP 内核.....	178
5.2	wakelock 和 early_suspend.....	140	7.3	移植 OMAP 体系结构	180
5.2.1	wakelock 和 early_suspend 的 原理.....	140	7.3.1	移植 OMAP 平台	180
5.2.2	Android 休眠.....	141	7.3.2	移植 OMAP 处理器	183
5.2.3	Android 唤醒.....	144	7.4	移植 Android 专用驱动和组件 ...	188
5.3	Ashmem 驱动程序	145	7.5	OMAP 的设备驱动	190
5.4	Pmem 驱动程序.....	148	第 8 章	显示系统驱动应用	195
5.5	Alarm 驱动程序.....	149	8.1	显示系统介绍.....	195
5.5.1	Alarm 简析	149	8.1.1	Android 的版本	195
5.5.2	Alarm 驱动程序的实现	150	8.1.2	不同版本的显示系统.....	195
5.6	USB Gadget 驱动程序.....	151	8.2	移植和调试前的准备.....	196
			8.2.1	FrameBuffer 驱动程序	196
			8.2.2	硬件抽象层.....	198
			8.3	实现显示系统的驱动程序.....	210

8.3.1 Goldfish 中的 FrameBuffer 驱动程序	210	11.2.3 本地代码	284
8.3.2 使用 Gralloc 模块的驱动程序... ..	214	11.2.4 JNI 代码	288
8.4 MSM 高通处理器中的显示驱动实现	224	11.2.5 Java 代码	289
8.4.1 MSM 中的 FrameBuffer 驱动程序	225	11.3 移植 Audio 系统的必备技术	289
8.4.2 MSM 中的 Gralloc 驱动程序... ..	227	11.3.1 移植 Audio 系统所要做的 工作	289
8.5 OMAP 处理器中的显示驱动实现... ..	235	11.3.2 分析硬件抽象层	290
第 9 章 输入系统驱动应用	239	11.3.3 分析 AudioFlinger 中的 Audio 硬件抽象层的实现	291
9.1 输入系统介绍	239	11.4 真正实现 Audio 硬件抽象层	298
9.1.1 Android 输入系统结构元素 介绍	239	11.5 MSM 平台实现 Audio 驱动系统... ..	298
9.1.2 移植 Android 输入系统时的 工作	240	11.5.1 实现 Audio 驱动程序	298
9.2 Input (输入) 驱动	241	11.5.2 实现硬件抽象层	299
9.3 模拟器的输入驱动	256	11.6 OSS 平台实现 Audio 驱动系统... ..	304
9.4 MSM 高通处理器中的输入驱动 实现	257	11.6.1 OSS 驱动程序介绍	304
9.4.1 触摸屏驱动	257	11.6.2 mixer	305
9.4.2 按键和轨迹球驱动	264	11.7 ALSA 平台实现 Audio 系统	312
9.5 OMAP 处理器平台中的输入驱 动实现	266	11.7.1 注册音频设备和音频驱动	312
9.5.1 触摸屏驱动	267	11.7.2 在 Android 中使用 ALSA 声卡	313
9.5.2 键盘驱动	267	11.7.3 在 OMAP 平台移植 Android 的 ALSA 声卡驱动	322
第 10 章 振动器系统驱动	269	第 12 章 视频输出系统驱动	326
10.1 振动器系统结构	269	12.1 视频输出系统结构	326
10.1.1 硬件抽象层	271	12.2 需要移植的部分	328
10.1.2 JNI 框架部分	272	12.3 分析硬件抽象层	328
10.2 开始移植	273	12.3.1 Overlay 系统硬件抽象层的 接口	328
10.2.1 移植振动器驱动程序	273	12.3.2 实现 Overlay 系统的硬件抽 象层	331
10.2.2 实现硬件抽象层	274	12.3.3 实现接口	332
10.3 在 MSM 平台实现振动器驱动... ..	275	12.4 实现 Overlay 硬件抽象层	333
第 11 章 音频系统驱动	279	12.5 在 OMAP 平台实现 Overlay 系统	335
11.1 音频系统结构	279	12.5.1 实现输出视频驱动程序	335
11.2 分析音频系统的层次	280	12.5.2 实现 Overlay 硬件抽象层	337
11.2.1 层次说明	280	12.6 系统层调用 Overlay HAL 的 架构	342
11.2.2 Media 库中的 Audio 框架	281	12.6.1 调用 Overlay HAL 的架构的 流程	342



12.6.2	S3C6410 Android Overlay 的 测试代码.....	346	15.2.1	移植驱动程序.....	417
15.2.2	移植硬件抽象层	418	15.2.3	实现上层部分	419
第 13 章	OpenMax 多媒体框架	349	15.3	在模拟器中实现传感器.....	424
13.1	OpenMax 基本层次结构.....	349	第 16 章	照相机系统	430
13.2	分析 OpenMax 框架构成	350	16.1	Camera 系统的结构	430
13.2.1	OpenMax 总体层次结构.....	350	16.2	需要移植的内容.....	433
13.2.2	OpenMax IL 层的结构	351	16.3	移植和调试.....	433
13.2.3	Android 中的 OpenMax	354	16.3.1	V4L2 驱动程序.....	433
13.3	实现 OpenMax IL 层接口	354	16.3.2	硬件抽象层	441
13.3.1	OpenMax IL 层的接口	354	16.4	实现 Camera 系统的硬件抽象层 ..	446
13.3.2	在 OpenMax IL 层中需要做 什么.....	361	16.4.1	Java 程序部分	446
13.3.3	研究 Android 中的 OpenMax 适配层	361	16.4.2	Camera 的 Java 本地调用 部分	447
13.4	在 OMAP 平台实现 OpenMax IL	363	16.4.3	Camera 的本地库 libui.so	448
13.4.1	实现文件	364	16.4.4	Camera 服务 libcameraservice.so	449
13.4.2	分析 TI OpenMax IL 的核心....	365	16.5	MSM 平台实现 Camera 系统	454
13.4.3	实现 TI OpenMax IL 组件 实例	368	16.6	OMAP 平台实现 Camera 系统 ...	457
第 14 章	多媒体插件框架	373	第 17 章	Wi-Fi 系统、蓝牙系统和 GPS 系统.....	459
14.1	Android 多媒体插件.....	373	17.1	Wi-Fi 系统	459
14.2	需要移植的内容	374	17.1.1	Wi-Fi 系统的结构	459
14.3	OpenCore 引擎	375	17.1.2	需要移植的内容	461
14.3.1	OpenCore 层次结构	375	17.1.3	移植和调试	461
14.3.2	OpenCore 代码结构	376	17.1.4	OMAP 平台实现 Wi-Fi.....	469
14.3.3	OpenCore 编译结构	377	17.1.5	配置 Wi-Fi 的流程	471
14.3.4	OpenCore OSCL	381	17.1.6	具体演练——在 Android 下 实现 Ethernet	473
14.3.5	实现 OpenCore 中的 OpenMax 部分	383	17.2	蓝牙系统.....	475
14.3.6	OpenCore 的扩展	398	17.2.1	蓝牙系统的结构	475
14.4	Stagefright 引擎	404	17.2.2	需要移植的内容	477
14.4.1	Stagefright 代码结构	404	17.2.3	具体移植.....	478
14.4.2	Stagefright 实现 OpenMax 接口	405	17.2.4	MSM 平台的蓝牙驱动.....	480
14.4.3	Video Buffer 传输流程	409	17.3	定位系统.....	482
第 15 章	传感器系统	415	17.3.1	定位系统的结构	483
15.1	传感器系统的结构.....	415	17.3.2	需要移植的内容	484
15.2	需要移植的内容	417	17.3.3	移植和调试	484

第 18 章 电话系统.....	498	19.1.1 Alarm 系统的结构	514
18.1 电话系统基础	498	19.1.2 需要移植的内容	515
18.1.1 电话系统简介.....	498	19.1.3 移植和调试.....	516
18.1.2 电话系统结构.....	500	19.1.4 模拟器环境的具体实现.....	518
18.2 需要移植的内容	501	19.1.5 MSM 平台实现 Alarm.....	518
18.3 移植和调试	502	19.2 Lights 光系统.....	519
18.3.1 驱动程序	502	19.2.1 Lights 光系统的结构.....	520
18.3.2 RIL 接口	504	19.2.2 需要移植的内容	521
18.4 电话系统实现流程分析	507	19.2.3 移植和调试.....	521
18.4.1 初始启动流程.....	507	19.2.4 MSM 平台实现光系统	523
18.4.2 request 流程	509	19.3 Battery 电池系统	524
18.4.3 response 流程	512	19.3.1 Battery 系统的结构	524
第 19 章 其他系统.....	514	19.3.2 需要移植的内容	526
19.1 Alarm 警报器系统	514	19.3.3 移植和调试.....	526
		19.3.4 在模拟器中实现电池系统	529

第 3 章 驱动需要移植

我们开发的 Android 驱动程序是基于 Linux 内核的，这些驱动程序需要在 Android 系统中使用，上述工作需要系统移植的知识。本章将详细介绍移植 Android 系统的基本知识，以及系统移植的基本原理，为后面学习驱动开发和移植打下坚实的基础。



3.1 驱动开发需要做的工作

Android 作为当前最流行的手机操作系统之一，受到了广大开发人员和商家的青睐。Android 正在逐渐形成一个蓬勃发展的产业，带来了无限商机。既然 Android 这么火爆，我们程序员可以学习它的哪一方面的内容呢？本书的驱动开发又属于哪一领域呢？接下来将为读者奉上这两个问题的答案。

Android 是一个开放的系统，这个系统的体积非常庞大，开发人员无须掌握整个 Android 体系中的开发知识，只需熟悉其中某一个部分即可收获自己的未来。

从具体功能上划分，Android 开发主要分为如下三个领域。

1. 移植开发移动电话系统

移植开发的目的是构建硬件系统，并且移植 Android 的软件系统，最终形成手机产品。

2. Android 应用程序开发

应用程序开发的目的是开发出各种 Android 应用程序，然后将这些应用程序投入 Android 市场，进行交易。

Android 的应用程序开发是 Android 开发的另一个方面。从开发的角度来看，这种形式的开发可以基于某个硬件系统，在没有硬件系统的情况下也可以基于 Linux 或者 Windows 下的 Android 模拟器来开发。这种类型的开发工作在 Android 系统的上层。

事实上，在 Android 软件系统中，第 3 个层次（Java 框架）和第 4 个层次（Java 应用）之间的接口也就是 Android 的系统接口（系统 API）。这个层次是标准的接口，所有的 Android 应用程序都是基于这个层次的接口开发出来的。Android 系统的第 4 个层次就是一组内置的 Android 应用程序。

Android 应用程序开发者开发的应用程序和 Android 系统的第 4 个层次的应用程序其实是一个层次的内容。例如，Android 系统提供了基本的桌面程序，开发者可以根据 Android 的系统接口，实现另外一个桌面程序，提供给用户安装使用。根据 Android 系统的接口开发游戏，也是



Android 应用程序开发的一个重要方向。

上述两种类型的开发结构如图 3-1 所示。



图 3-1 Android 开发的领域

3. Android 系统开发

系统开发的目的是升级或改造 Android 中已经存在的应用和架构，开发出有自己特色的手机系统。例如联想手机乐 Phone 就是在 Android 基础上打造的一款适合国人使用习惯的手机系统，如图 3-2 所示。



图 3-2 乐 Phone

Android 系统开发的一个比较典型的示例就是当系统需要某种功能时，为了给 Java 层次的应用程序提供调用的接口，需要从底层到上层的整体开发，具体步骤如下所示。

- step 1** 增加 C 或者 C++ 和本地库。
- step 2** 定义 Java 层所需要的类（系统 API）。
- step 3** 将所需要的代码封装成 JNI。
- step 4** 结合 Java 类和 JNI。
- step 5** 应用程序调用 Java 类。

一定要慎重对待对 Android 系统 API 的改动工作，因为系统 API 的稍微变动就可能会涉及 Android 应用程序的兼容问题。

Android 系统本身的功能也处于增加和完善的过程中，因此 Android 系统的开发也是一个重要的方面。这种类型的开发涉及 Android 软件系统的各个层次。在更多的时候，Android 系统开发只是在不改变系统 API 的情况下修正系统的缺陷，增加系统的稳定性。

从商业模式的角度来看，第一种类型的开发和第二种类型的开发是 Android 开发的主流。事实上，移动电话的制造者主要进行第一种类型的开发，产品是 Android 实体手机；公司、个人和团体都可以进行第二种类型的开发，其产品是不同的 Android 应用程序。

在 Android 的开发过程中，每一种类型的开发都只涉及整个 Android 系统的一个子集。在 Android 系统中有着众多开发点，这些开发点相互独立，又有内在联系。在开发的过程中，需要重点掌握目前开发点涉及的部分。

背景说明：Android API 的接口是用 Java 语言编写的，通常更改接口函数的格式（参数、返回值）、常量的值等内容就相当于更改系统 API。

Android 是一个开放的系统，适用于从最低端直到最高端的智能手机。核心的 Android API 在每部手机上都可使用，但仍然有一些 API 接口有一些特别的适用范围：这就是所谓的“可选 API”。

在为某手机编写 Android 应用程序时，需要多少地对 Android API 进行修改，然后实现我们需要的功能。例如使用 Android API 添加蓝牙程序和 Wi-Fi 程序。在更改 Android API 时，通常更改其接口函数的格式（参数、返回值）和常量值等内容。但是 Android API 毕竟是谷歌推出的一系列标准，为了方便以后系统的升级，建议大家不改变 Android API 的格式，而是只改变 Android API 的具体行为，也就是说为这些固定的 Android API 编写各种各样的应用程序。



3.2 Android 移植

本书讲解的是 Android 驱动方面的开发知识，由图 3-1 可知，驱动开发是底层的应用，属于 Linux 内核层的工作。因为驱动是系统和硬件之间的载体，涉及不同硬件的应用问题，所以需要系统移植工作。本节将简要介绍系统移植方面的有关问题。



3.2.1 移植的任务

Android 移植开发的最终目的是开发手机产品。从开发者的角度来看，这种类型的开发以具有硬件系统为前提，在硬件系统的基础上构建 Android 软件系统。这种类型的开发工作在 Android 系统的底层。在软件系统方面，主要的工作集中在以下两个方面。

(1) Linux 中的相关设备驱动程序

驱动程序是硬件和上层软件的接口。在 Android 手机系统中，需要基本的屏幕、触摸屏、键盘等驱动程序，以及音频、摄像头、电话的 Modem、Wi-Fi、蓝牙等多种设备驱动程序。

(2) Android 本地框架中的硬件抽象层

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。Android 系统对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。

上述两个部分综合起来相互结合，共同完成了 Android 系统的软件移植。移植成功与否取决于驱动程序的品质和对 Android 硬件抽象层接口的理解程度。Android 移植开发的工作由核心库、Dalvik 虚拟机、硬件抽象层、Linux 内核层和硬件系统协同完成，具体结构如图 3-3 所示。

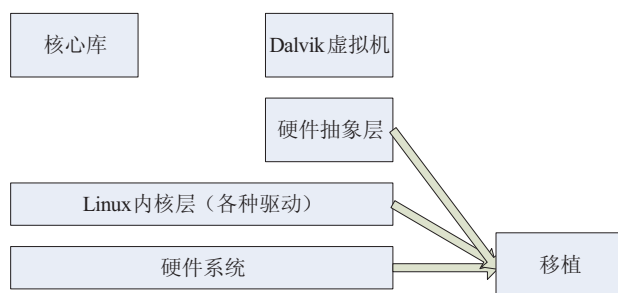


图 3-3 Android 移植结构

3.2.2 移植的内容

在 Android 系统中，在移植过程中主要移植驱动方面的内容。Android 移植主要分为如下几个类型。

- 基本图形用户界面(GUI)部分：包括显示部分、用户输入部分和硬件相关的加速部分，还包括媒体编解码和 OpenGL 等。
- 音视频输入输出部分：包括音频、视频输出和摄像头等。
- 连接部分：包括无线局域网、蓝牙、GPS 等。
- 电话部分：包括通话、GSM 等。
- 附属部件：包括传感器、背光、振动器等。

具体来说主要移植下面的内容。

- Display 显示部分：包括 FrameBuffer 驱动和 Gralloc 模块。
- Input 用户输入部分：包括 Event 驱动和 EventHub。
- Codec 多媒体编解码：包括硬件 Codec 驱动和 Codec 插件，例如 OpenMax。
- 3D Accelerator（3D 加速器）部分：包括硬件 OpenGL 驱动和 OpenGL 插件。

- Audio 音频部分：包括 Audio 驱动和 Audio 硬件抽象层。
- Video Out 视频输出部分：包括视频显示驱动和 Overlay 硬件抽象层。
- Camera 摄像头部分：包括 Camera 驱动（通常是 v4l2）和 Camera 硬件抽象层。
- Phone 电话部分：包括 Modem 驱动程序和 RIL 库。
- GPS 全球定位系统部分：包括 GPS 驱动（例如串口）和 GPS 硬件抽象层。
- Wi-Fi 无线局域网部分：包括 Wlan 驱动和协议和 Wi-Fi 的适配层。
- Blue Tooth 蓝牙部分：包括 BT 驱动和协议及 BT 的适配层。
- Sensor 传感器部分：包括 Sensor 驱动和 Sensor 硬件抽象层。
- Vibrator 振动器部分：包括 Vibrator 驱动和 Vibrator 硬件抽象层。
- Light 背光部分：包括 Light 驱动和 Light 硬件抽象层。
- Alarm 警告器部分：包括 Alarm 驱动和 RTC 系统和用户空间调用。
- Battery 电池部分：包括电池部分驱动和电池的硬件抽象层。

注意：在 Android 系统中有很多组件，但并不是每一个组件都需要移植，例如那些纯软的组件就不需要移植。像浏览器引擎虽然需要下层的网络支持，但是实际上并不需要直接为其移植网络接口，而是通过无线局域网或者电话系统数据连接来完成标准的网络接口。

3.2.3 驱动开发的任务

前面介绍了 Android 系统的基本知识和移植内容，那么究竟在驱动开发领域需要做什么工作呢？我们的任务就是为某一个将要在 Android 系统上使用的硬件开发一个驱动程序。因为 Android 是基于 Linux 的，所以开发 Android 驱动其实就是开发 Linux 驱动。

对于大部分子系统来说，硬件抽象层和驱动程序都需要根据实际系统的情况来实现，例如传感器部分、音频部分、视频部分、摄像头部分和电话部分。另外也有一些子系统的硬件抽象层是标准的，只需实现 Linux 内核中的驱动程序即可，例如输入部分、振动器部分、无线局域网部分和蓝牙部分等。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。

随着 Android 系统的更新和发展，它已经不仅仅是一个移动设备的平台，也可以用于消费类电子和智能家电，例如 3.0 以后的版本主要是针对平板电脑的，另外电子书、数字电视、机顶盒、固定电话等都逐渐使用 Android 系统。在这些平台上，通常要实现比移动设备更少的部件。一般来说，包括显示和用户输入的基本用户界面部分是需要移植的，其他部分是可选的。例如电话系统、振动器、背光、传感器等一般不需要在非移动设备系统上实现，一些固定位置设备通常不需要实现 GPS 系统。



3.3 Android 对 Linux 的改造

Android 内核是基于 Linux 2.6 内核的，这是一个增强内核版本，除了修改部分 Bug 外，还提供了用于支持 Android 平台的设备驱动。Android 不但使用了 Linux 内核的基本功能，而且对 Linux 进行了改造，目的是实现更为强大的通信功能。



3.3.1 Android 对 Linux 内核文件的改动

在本书第 1 章的内容中已经讲解过 Linux 内核的基本知识，其实 Android 对 Linux 内核也进行了改动，这些改动保存在下面的文件中。

```
drivers/misc/kernel_debugger.c
drivers/misc/pmem.c
drivers/misc/qemutrace/qemu_trace_sysfs.c
drivers/misc/qemutrace/qemu_trace.c
drivers/misc/qemutrace/qemu_trace.h
drivers/misc/uid_stat.c
drivers/staging/android/lowmemorykiller.c
drivers/staging/android/logger.c
drivers/staging/android/timed_output.h
drivers/staging/android/ram_console.c
drivers/staging/android/timed_gpio.c
drivers/staging/android/logger.h
drivers/staging/android/binder.h
drivers/staging/android/binder.c
drivers/staging/android/timed_output.c
drivers/staging/android/timed_gpio.h
drivers/rtc/alarm.c
drivers/rtc/rtc-goldfish.c
drivers/net/pppolac.c
drivers/net/ppp_mppe.c
drivers/net/pppopns.c
drivers/video/goldfishfb.c
drivers/switch/switch_class.c
drivers/switch/switch_gpio.c
drivers/char/dcc_tty.c
drivers/char/goldfish_tty.c
drivers/watchdog/i6300esb.c
drivers/input/misc/gpio_event.c
drivers/input/misc/gpio_input.c
drivers/input/misc/gpio_output.c
drivers/input/misc/keychord.c
drivers/input/misc/gpio_axis.c
drivers/input/misc/gpio_matrix.c
drivers/input/keyreset.c
drivers/input/keyboard/goldfish_events.c
drivers/input/touchscreen/synaptics_i2c_rmi.c
drivers/usb/gadget/android.c
drivers/usb/gadget/f_adb.h
drivers/usb/gadget/f_mass_storage.h
drivers/usb/gadget/f_adb.c
drivers/usb/gadget/f_mass_storage.c
```

```

drivers/mmc/host/goldfish.c
drivers/power/goldfish_battery.c
drivers/leds/ledtrig-sleep.c
drivers/mtd/devices/goldfish_nand_reg.h
drivers/mtd/devices/goldfish_nand.c
kernel/power/earlysuspend.c
kernel/power/consoleearlysuspend.c
kernel/power/fbearlysuspend.c
kernel/power/wakelock.c
kernel/power/userwakelock.c
kernel/cpuset.c
kernel/cgroup_debug.c
kernel/cgroup.c
mm/ashmem.c
include/linux/ashmem.h
include/linux/switch.h
include/linux/keychord.h
include/linux/earlysuspend.h
include/linux/android_aid.h
include/linux/uid_stat.h
include/linux/if_pppolac.h
include/linux/usb/android.h
include/linux/wifi_tiwlan.h
include/linux/android_alarm.h
include/linux/keyreset.h
include/linux/synaptics_i2c_rmi.h
include/linux/android_pmem.h
include/linux/kernel_debugger.h
include/linux/gpio_event.h
include/linux/wakelock.h
include/linux/if_pppopns.h
net/ipv4/sysfs_net_ipv4.c
net/ipv4/af_inet.c
net/ipv6/af_inet6.c
net/bluetooth/af_bluetooth.c
security/commoncap.c
fs/proc/base.c

```

3.3.2 为 Android 构建 Linux 的操作系统

如果我们以一个原始的 Linux 操作系统为基础，改造成一个适合于 Android 的系统，所要做的工作其实非常简单，仅仅是增加适用于 Android 的驱动程序。在 Android 中有很多 Linux 系统的驱动程序，将这些驱动程序移植到新系统的步骤非常简单，具体来说有以下三个步骤。

- step 1** 编写新的源代码。
- step 2** 在 KConfig 配置文件中增加新内容。



step 3 在 Makefile 中增加新内容。

在 Android 系统中，通常会使用 FrameBuffer 驱动、Event 驱动、Flash MTD 驱动、Wi-Fi 驱动、蓝牙驱动和串口等驱动程序。并且还需要音频、视频、传感器等驱动和 sysfs 接口。移植的过程就是移植上述驱动的过程，我们的工作是在 Linux 下开发适用于 Android 的驱动程序，并移植到 Android 系统。

在 Android 中添加扩展驱动程序的基本步骤如下所示。

step 1 在 Linux 内核中移植硬件驱动程序，实现系统调用接口。

step 2 在 HAL 中把硬件驱动程序的调用封装成 Stub。

step 3 为上层应用的服务实现本地库，由 Dalvik 虚拟机调用本地库来完成上层 Java 代码的实现。

step 4 编写 Android 应用程序，提供 Android 应用服务和用户操作界面。



3.4 内核空间和用户空间接口是一个媒介

驱动程序是供系统使用硬件的，也就是说，驱动程序是介于系统和硬件之间的桥梁。在 Linux 下开发这些中间桥梁的驱动程序时，需要用到内核空间和用户空间之间的接口。

3.4.1 内核空间和用户空间的相互作用

现在，越来越多的应用程序需要编写内核级和用户级的程序来一起完成具体的任务。通常采用以下模式：首先，编写内核服务程序利用内核空间提供的权限和服务来接收、处理和缓存数据；然后，编写用户程序和先前完成的内核服务程序进行交互。具体来说，可以利用用户程序来配置内核服务程序的参数，提取内核服务程序提供的数据，当然也可以向内核服务程序输入待处理数据。

比较典型的应用包括 Netfilter（内核服务程序：防火墙）；Iptable（用户级程序：规则设置程序）；IPSEC（内核服务程序：VPN 协议部分）；IKE（用户级程序：VPN 密钥协商处理）；当然还包括大量的设备驱动程序及相应的应用软件。这些应用都是由内核级和用户级程序通过相互交换信息来一起完成特定任务的。

3.4.2 系统和硬件之间的交互

实现硬件和系统的交互是底层开发的主要任务之一。在 Linux 平台下有如下 5 种实现此功能的方式。

1. 编写自己的系统调用

系统调用是用户级程序访问内核最基本的方法。目前 Linux 大致提供了二百多个标准的系统调用（具体请参考内核代码树中的“include/asm-i386/unistd.h”和“arch/i386/kernel/entry.S”文件），并且允许我们添加自己的系统调用来实现和内核的信息交换。假如我们想建立一个系统调用日志系统，将所有的系统调用动作记录下来，以便进行入侵检测，此时可以编写一个内核服务程序，该程序负责收集所有的系统调用请求，并将这些调用信息记录到在内核中自建的缓冲里。我们无

法在内核里实现复杂的入侵检测程序，因此必须将该缓冲里的记录提取到用户空间。最直截了当的方法是自己编写一个新系统调用实现这种提取缓冲数据的功能。当内核服务程序和新系统调用都实现后，就可以在用户空间里编写用户程序执行入侵检测任务了，入侵检测程序可以定时、轮询或在需要的时候调用新系统调用从内核提取数据，然后进行入侵检测。

2. 编写驱动程序

Linux/UNIX 的一个特点就是把所有的东西都看做文件（every thing is a file）。系统定义了简洁完善的驱动程序界面，客户程序可以用统一的方法通过这个界面和内核驱动程序交互。而大部分系统的使用者和开发者已经非常熟悉这种界面及相应的开发流程了。

驱动程序运行于内核空间，用户空间的应用程序通过文件系统中“/dev/”目录下的一个文件来和它交互。这就是我们熟悉的文件操作流程：`open()`→`read()`→`write()`→`ioctl()`→`close()`。

注意：并不是所有的内核驱动程序都是这个界面，网络驱动程序和各种协议栈的使用就不大一致，比如套接口编程虽然也有 `open()`，`close()`等概念，但它的内核实现及外部使用方式都和普通驱动程序有很大差异。

这里先不谈设备驱动程序在内核中要做的中断响应、设备管理、数据处理等工作，在此先把注意力集中在它与用户级程序交互这一部分。操作系统为此定义了一种统一的交互界面，就是前面所说的 `open()`、`read()`、`write()`、`ioctl()`和 `close()`等。每个驱动程序按照自己的需要做独立实现，把自己提供的功能和服务隐藏在这个统一界面下。客户级程序选择需要的驱动程序或服务（其实就是选择“/dev/”目录下的文件），按照上述界面和文件操作流程，就可以跟内核中的驱动交互了。用面向对象的概念更容易解释，系统定义了一个抽象的界面（Abstract Interface），每个具体的驱动程序都是这个界面的实现（Implementation）。

由此可见，驱动程序也是用户空间和内核信息交互的重要方式之一。从本质上来说，`ioctl`、`read`、和 `write` 也是通过系统调用去完成的，只是这些调用已被内核进行了标准封装和统一定义。因此用户不必像添加新系统调用那样必须修改内核代码，重新编译新内核，使用虚拟设备只需要通过模块方法将新的虚拟设备安装到内核中（`insmod` 上）就能方便使用。

大致可以将 Linux 中的设备分为如下三类。

- 字符设备：包括那些必须以顺序方式，像字节流一样被访问的设备。
- 块设备：指那些可以用随机方式，以整块数据为单位来访问的设备，如硬盘等。
- 网络接口：通常指网卡和协议栈等复杂的网络输入输出服务。

如果将我们的系统调用日志系统用字符型驱动程序的方式实现，整个过程就非常简单了。可以将内核中收集和记录信息的那一部分编写成一个字符设备驱动程序。虽然没有实际对应的物理设备，但是 Linux 的设备驱动程序本来就是一个软件抽象，它可以结合硬件提供服务，也完全可以作为纯软件提供服务。在驱动程序中，可以使用 `open()`来启动服务，用 `read()`返回处理好的记录，用 `ioctl()`设置记录格式等，用 `close()`停止服务，`write()`没有用到，那么我们可以不去实现它。然后在“/dev/”目录下建立一个设备文件，对应我们新加入内核的系统调用日志系统驱动程序。

3. 使用 proc 文件系统

`proc` 是 Linux 提供的一种特殊的文件系统，使用它的目的就是提供一种便捷的用户和内核间的交互方式。`proc` 以文件系统作为使用界面，使应用程序可以以文件操作的方式安全、方便地获



取系统当前运行的状态和其他一些内核数据信息。

`proc` 文件系统多用于监视、管理和调试系统，平常使用的 `ps` 和 `top` 等管理工具就是利用 `proc` 来读取内核信息的。除了读取内核信息外，`proc` 文件系统还提供了写入功能，所以我们可以利用它来向内核输入信息。比如通过修改 `proc` 文件系统下的系统参数配置文件“`/proc/sys`”，可以直接在运行时动态更改内核参数。

除了系统已经提供的文件条目，通过 `proc` 为我们留的接口，允许在内核中创建新的条目从而与用户程序共享信息数据。比如可以为系统调用日志程序（无论是作为驱动程序还是作为单纯的内核模块）在 `proc` 文件系统中创建新的文件条目，在此条目中显示系统调用的使用次数，每个单独系统调用的使用频率等。也可以增加另外的条目用于设置日志记录规则。

4. 使用虚拟文件系统（VFS）

很多内核开发者认为利用 `ioctl()` 系统调用往往会使得系统调用意义不明确，而且难控制。而将信息放入到 `proc` 文件系统中会使信息组织混乱，所以不赞成过多地使用此系统。他们的建议是实现一种孤立的虚拟文件系统来代替 `ioctl()` 和 `proc`。这是因为文件系统接口清楚，而且便于用户空间访问，同时利用虚拟文件系统使得利用脚本执行系统管理任务更加方便、有效。

下面举例来说如何通过虚拟文件系统修改内核信息。假设我们可以实现一个名为“`sagafs`”的虚拟文件系统，其中文件 `log` 对应内核存储的系统调用日志。此时就可以通过文件访问的普遍方法获得日志信息，命令如下所示。

```
# cat /sagafs/log
```

使用虚拟文件系统可以更加方便、清晰地实现信息交互。但是很多程序员认为 VFS 的 API 接口十分复杂。其实读者无须担心，因为从 Linux 2.5 内核开始就提供了一种叫做 `libfs` 的例子程序，它可以帮助不熟悉文件系统的用户封装实现了 VFS 的通用操作。

5. 使用内存映像

Linux 通过内存映像机制来提供用户程序对内存直接访问的能力。内存映像的意思是把内核中特定部分的内存空间映射到用户级程序的内存空间去。也就是说，用户空间和内核空间共享一块相同的内存。这样做有如下影响。

内核在这块地址内存储变更的任何数据，用户可以立即发现和使用，根本无须进行数据复制。

在使用系统调用交互信息时，在整个操作过程中必须有一步数据复制的工作，或者是把内核数据复制到用户缓冲区，或者是把用户数据复制到内核缓冲区。这样对于许多数据传输量大、时间要求高的应用来说很不科学，因为许多应用根本就无法忍受数据复制所耗费的时间和资源。

3.4.3 使用 Relay 实现内核到用户空间的数据传输

Relay 是一种从 Linux 内核到用户空间的高效数据传输技术。通过用户定义的 Relay 通道，内核空间的程序能够高效、可靠、便捷地将数据传输到用户空间。Relay 特别适用于内核空间有大量数据需要传输到用户空间的情形，目前已经广泛应用在内核调试工具如 `SystemTap` 中。

1. Relay 的原理

Relay 提供了一种机制，使得内核空间的程序能够通过用户定义的 Relay 通道（channel）将

大量数据高效地传输到用户空间。一个 Relay 通道由一组和 CPU 一一对应的内核缓冲区组成。这些缓冲区又被称为 Relay 缓冲区（buffer），其中的每一个在用户空间都用一个常规文件来表示，叫做 Relay 文件（file）。内核空间的用户可以利用 Relay 提供的 API 接口来写入数据，这些数据会被自动写入当前的 CPU ID 对应的那个 Relay 缓冲区；同时，这些缓冲区从用户空间看来，是一组普通文件，可以直接使用 `read()` 进行读取，也可以使用 `mmap()` 进行映射。Relay 并不关心数据的格式和内容，这些完全依赖于使用 Relay 的用户程序。Relay 的目的是提供一个足够简单的接口，从而使得基本操作尽可能高效。

Relay 实现了对数据读和写的分离，使得大量突发性数据写入的时候，不需要受限于用户空间相对较慢的读取速度，从而大大提高了效率。Relay 作为写入和读取的桥梁，也就是将内核用户写入的数据缓存并转发给用户空间的程序。这种转发机制正是 Relay 这个名称的由来。

2. Relay 的 API

在 Relay 中提供了许多 API 来支持用户程序完整地使用 Relay。这些 API 主要分为两大类，分别是面向用户空间和面向内核空间的，具体说明如下所示。

（1）面向用户空间的 API

此类 API 编程接口向用户空间程序提供了访问 Relay 通道缓冲区数据的基本操作的入口，主要包括如下方法。

- `open()`: 允许用户打开一个已经存在的通道缓冲区。
- `mmap()`: 使通道缓冲区被映射到位于用户空间的调用者的地址空间。要特别注意的是，我们不能仅对局部区域进行映射。也就是说，必须映射整个缓冲区文件，其大小是 CPU 的个数和单个 CPU 缓冲区大小的乘积。
- `read()`: 读取通道缓冲区的内容。这些数据一旦被读出，就意味着它们被用户空间的程序消费掉了，不能被之后的读操作看到。
- `sendfile()`: 将数据从通道缓冲区传输到一个输出文件描述符。其中可能的填充字符会被自动去掉，不会被用户看到。
- `poll()`: 支持 `POLLIN/POLLRDNORM/POLLERR` 信号。每次子缓冲区的边界被越过时，等待着的用户空间程序会得到通知。
- `close()`: 将通道缓冲区的引用数减 1。当引用数减为 0 时，表明没有进程或者内核用户需要打开它，从而这个通道缓冲区被释放。

（2）面向内核空间的 API

此类 API 接口向位于内核空间的用户提供了管理 Relay 通道、数据写入等功能，其中最为常用的如下所示。

- `relay_open()`: 创建一个 Relay 通道，包括创建每个 CPU 对应的 Relay 缓冲区。
- `relay_close()`: 关闭一个 Relay 通道，包括释放所有的 Relay 缓冲区，在此之前会调用 `relay_switch()` 来处理这些 Relay 缓冲区以保证已读取但是未满足的数据不会丢失。
- `relay_write()`: 将数据写入到当前 CPU 对应的 Relay 缓冲区内。由于它使用了 `local_irqsave()` 保护，因此也可以在中断上下文中使用。
- `relay_reserve()`: 在 Relay 通道中保留一块连续的区域来留给未来的写入操作。通常用于那些希望直接写入到 Relay 缓冲区的用户。考虑到性能或者其他因素，这些用户不希望先把



数据写到一个临时缓冲区中，然后再通过 `relay_write()` 进行写入。

3. 使用 Relay

在下面的内容中，将通过一个最简单的例子来介绍使用 **Relay** 的方法。本实例由如下两部分组成。

- 位于内核空间将数据写入 **Relay** 文件的程序，使用时需要作为一个内核模块被加载。
- 位于用户空间从 **Relay** 文件中读取数据的程序，使用时作为普通用户态程序运行。

(1) 实现内核空间

内核空间程序的主要操作如下所示。

- 当加载模块时，打开一个 **Relay** 通道，并且往打开的 **Relay** 通道中写入消息。
- 当卸载模块时，关闭 **Relay** 通道。

文件 `hello-mod.c` 的具体实现代码如下所示。

```
#include <linux/module.h>
#include <linux/relayfs_fs.h>
static struct rchan *hello_rchan;
int init_module(void)
{
    const char *msg="Hello world\n";
    hello_rchan = relay_open("cpu", NULL, 8192, 2, NULL);
    if(!hello_rchan){
        printk("relay_open() failed.\n");
        return -ENOMEM;
    }
    relay_write(hello_rchan, msg, strlen(msg));
    return 0;
}
void cleanup_module(void)
{
    if(hello_rchan) {
        relay_close(hello_rchan);
        hello_rchan = NULL;
    }
    return;
}
MODULE_LICENSE ("GPL");
MODULE_DESCRIPTION ("Simple example of Relay");
```

(2) 实现用户空间

用户空间的函数主要操作过程如下所示。

如果 `relayfs` 文件系统还没有被 `umount`（是一个命令）处理，则将其 `umount` 到“`/mnt/relay`”目录上。首先遍历每一个 `CPU` 对应的缓冲文件，然后打开文件，接着读取所有文件内容，然后关闭文件，最后，`umount` 掉 **Relay** 文件系统。

实现文件 `audience.c` 的具体实现代码如下所示。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mount.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>
#include <stdio.h>
#define MAX_BUFLEN 256
const char filename_base[]="/mnt/relay/cpu";
// implement your own get_cputotal() before compilation
static int get_cputotal(void);
int main(void)
{
    char filename[128]={0};
    char buf[MAX_BUFLEN];
    int fd, c, i, bytesread, cputotal = 0;
    if(mount("relayfs", "/mnt/relay", "relayfs", 0, NULL)
        && (errno != EBUSY)) {
        printf("mount() failed: %s\n", strerror(errno));
        return 1;
    }
    cputotal = get_cputotal();
    if(cputotal <= 0) {
        printf("invalid cputotal value: %d\n", cputotal);
        return 1;
    }
    for(i=0; i<cputotal; i++) {
        // open per-cpu file
        sprintf(filename, "%s%d", filename_base, i);
        fd = open(filename, O_RDONLY);
        if (fd < 0) {
            printf("fopen() failed: %s\n", strerror(errno));
            return 1;
        }
        // read per-cpu file
        bytesread = read(fd, buf, MAX_BUFLEN);
        while(bytesread > 0) {
            buf[bytesread] = '\0';
            puts(buf);
            bytesread = read(fd, buf, MAX_BUFLEN);
        };
        // close per-cpu file
        if(fd > 0) {
            close(fd);
            fd = 0;
        }
    }
}
```



```
    }  
}  
if(umount("/mnt/relay") && (errno != EINVAL)) {  
    printf("umount() failed: %s\n", strerror(errno));  
    return 1;  
}  
return 0;  
}
```

上述实例演示了使用 Relay 的过程。虽然上述代码并没有实际用处，但是形象地描述了从用户空间和内核空间两个方面使用 Relay 的基本流程。实际应用中对 Relay 的使用当然比这复杂得多，有关更多用法的实例请参考 Relay 的主页。



3.5 三类驱动程序

在 Linux 系统中主要有三类设备驱动程序，分别是字符设备驱动程序、块设备驱动程序和网络设备驱动程序。本节将简要讲解上述三类设备驱动程序的基本知识。

3.5.1 字符设备驱动程序

字符设备是指在 I/O 传输过程中以字符为单位进行传输的设备，例如键盘、打印机等。请注意，以字符为单位并不意味着是以字节为单位，因为有的编码规则规定，1 个字符占 16 比特，合 2 个字节。字符设备驱动程序的结构如图 3-4 所示。

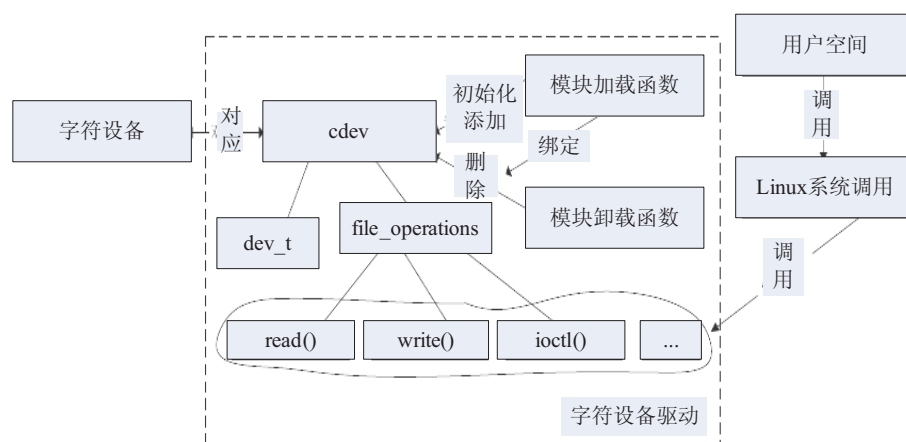


图 3-4 字符设备驱动程序的结构

在 Linux 系统中，字符设备以特别文件方式在文件目录树中占据位置并拥有相应的 i 节点。在 i 节点中的文件类型指明该文件是字符设备文件。可以使用与普通文件相同的文件操作命令对字符设备文件进行操作，例如打开、关闭、读、写等。概括来说，字符设备驱动程序主要做如下三件事。

- 定义一个结构体 `static struct file_operations` 变量，在其中定义一些设备的打开、关闭、读、

写、控制函数。

- 在结构体外分别实现结构体中定义的这些函数。
- 向内核中注册或删除驱动模块。

由此可见，实现字符设备驱动程序的首要任务是定义一个结构体。字符设备提供给应用程序流控制接口有 `open`、`close`、`read`、`write` 和 `ioctl`。添加一个字符设备驱动程序的过程，实际上是给上述操作添加对应的代码的过程，Linux 对这些操作统一做了抽象。

定义结构体 `file_operations` 的格式如下所示。

```
static struct file_operations myDriver_fops = {
    owner: THIS_MODULE,
    write: myDriver_write,
    read: myDriver_read,
    ioctl: myDriver_ioctl,
    open: myDriver_open,
    release: myDriver_release,
};
```

在此结构体中规定了驱动程序向应用程序提供的操作接口，主要有实现以下几个功能的接口。

- 实现 `write` 操作，就是从应用程序接收数据送到硬件。例如下面的代码。

```
static ssize_t myDriver_write(struct file *filp, const char *buf, size_t count, loff_t
*f_pos) {
    size_t fill_size = count;
    PRINTK("myDriver write called!\n");
    PRINTK("\tcount=%d, pos=%d\n", count, (int)*f_pos);
    if(*f_pos >= sizeof(myDriver_Buffer))
    {
        PRINTK("[myDriver write]Buffer Overlap\n");
        *f_pos = sizeof(myDriver_Buffer);
        return 0;
    }
    if((count + *f_pos) > sizeof(myDriver_Buffer))
    {
        PRINTK("count + f_pos > sizeof buffer\n");
        fill_size = sizeof(myDriver_Buffer) - *f_pos;
    }
    copy_from_user(&myDriver_Buffer[*f_pos], buf, fill_size);
    *f_pos += fill_size;
    return fill_size;
}
```

在上述代码中，函数 `u_long copy_from_user(void *to, const void *from, u_long len)` 用于把用户态的数据复制到内核态，实现数据的传送。

- 实现 `read` 操作，即从硬件读取数据并交给应用程序。例如下面的代码。



```
static ssize_t myDriver_read(struct file *filp, char *buf, size_t count, loff_t
*f_pos) {
    size_t read_size = count;
    PRINTK("myDriver read called!\n");
    PRINTK("\tcount=%d, pos=%d\n", count, (int)*f_pos);
    if(*f_pos >= sizeof(myDriver_Buffer))
    {
        PRINTK("[myDriver read]Buffer Overlap\n");
        *f_pos = sizeof(myDriver_Buffer);
        return 0;
    }
    if((count + *f_pos) > sizeof(myDriver_Buffer))
    {
        PRINTK("count + f_pos > sizeof buffer\n");
        read_size = sizeof(myDriver_Buffer) - *f_pos;
    }
    copy_to_user(buf, &myDriver_Buffer[*f_pos], read_size);
    *f_pos+= read_size;
    return read_size;
}
```

在上述代码中，函数 `u_long copy_to_user(void * to, const void *from, u_long len)` 用于实现把内核态的数据复制到用户态。

- 实现 `ioctl` 操作，即为应用程序提供对硬件行为的控制。例如下面的代码。

```
static int myDriver_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
unsigned long arg){
    PRINTK("myDriver ioctl called(%d)!\n", cmd);
    if(_IOC_TYPE(cmd) != TSTDRV_MAGIC)
    {
        return -ENOTTY;
    }
    if(_IOC_NR(cmd) >= TSTDRV_MAXNR)
    {
        return -ENOTTY;
    }
    switch(cmd)
    {
        case MYDRV_IOCTL0:
            PRINTK("IOCTRL 0 called(0x%lx)!\n", arg);
            break;
        case MYDRV_IOCTL1:
            PRINTK("IOCTRL 1 called(0x%lx)!\n", arg);
            break;
        case MYDRV_IOCTL2:
            PRINTK("IOCTRL 2 called(0x%lx)!\n", arg);
            break;
    }
```

```

case MYDRV_IOCTL3:
    PRINTK("_IOCTL 3 called(0x%lx)!\n", arg);
    break;
}
return 0;
}

```

- 实现 **open** 操作。

当应用程序打开设备时对设备进行初始化，使用 `MOD_INC_USE_COUNT` 增加驱动程序的使用次数。例如下面的代码。

```

static int myDriver_open(struct inode *inode, struct file *filp){
    MOD_INC_USE_COUNT;
    PRINTK("myDriver open called!\n");
    return 0;
}

```

- 实现 **release** 操作。

当应用程序关闭设备时处理设备的关闭操作。使用 `MOD_DEC_USE_COUNT` 来增加驱动程序的使用次数。例如下面的代码。

```

static int myDriver_release(struct inode *inode, struct file *filp){
    MOD_DEC_USE_COUNT;
    PRINTK("myDriver release called!\n");
    return 0;
}

```

- 驱动程序初始化函数。

Linux 在加载内核模块时会调用初始化函数，初始化驱动程序本身使用 `register_chrdev` 向内核注册驱动程序，该函数的第三个参数是指向包含有驱动程序接口函数信息的 `file_operations` 结构体。例如下面的代码。

```

#ifdef CONFIG_DEVFS_FS
devfs_handle_t devfs_myDriver_dir;
devfs_handle_t devfs_myDriver_raw;
#endif
static int __init myModule_init(void)
{
    /* Module init code */
    PRINTK("myModule_init\n");
    /* Driver register */
    myDriver_Major = register_chrdev0(DRIVER_NAME, &myDriver_fops);
    if(myDriver_Major < 0)
    {
        PRINTK("register char device fail!\n");
        return myDriver_Major;
    }
}

```



```
}
PRINTK("register myDriver OK! Major = %d\n", myDriver_Major);
#ifdef CONFIG_DEVFS_FS
    devfs_myDriver_dir = devfs_mk_dir(NULL, "myDriver", NULL);
    devfs_myDriver_raw = devfs_register(devfs_myDriver_dir, "raw0", DEVFS_FL_DEFAULT,
myDriver_Major, 0, S_IFCHR | S_IRUSR | S_IWUSR, &myDriver_fops, NULL);
    PRINTK("add dev file to devfs OK!\n");
#endif
return 0;
}
```

在上述代码中，函数 `module_init()` 的功能是向内核声明当前模块的初始化函数。

- 驱动程序退出函数。

Linux 在卸载内核模块时会调用退出函数释放驱动程序使用的资源，使用 `unregister_chrdev` 从内核中卸载驱动程序。将驱动程序模块注册到内核，内核需要知道模块的初始化函数和退出函数，才能将模块放入自己的管理队列中。例如下面的代码。

```
static void __exit myModule_exit(void){
    /* Module exit code */
    PRINTK("myModule_exit\n");
    /* Driver unregister */
    if(myDriver_Major > 0)
    {
        #ifdef CONFIG_DEVFS_FS
            devfs_unregister(devfs_myDriver_raw);
            devfs_unregister(devfs_myDriver_dir);
        #endif
        unregister_chrdev(myDriver_Major, DRIVER_NAME);
    }
    return;
}
```

在上述代码中，函数 `module_exit()` 的功能是向内核声明当前模块的退出函数。

经过上述分析，可以总结出开发字符设备驱动程序的基本步骤如下所示。

step 1 确定主设备号和次设备号。

step 2 实现字符驱动程序，先实现 `file_operations` 结构体，然后实现初始化函数并注册字符设备，接下来实现销毁函数并释放字符设备。

step 3 创建设备文件节点。

接下来给出一个通用的字符设备驱动程序。此程序由如下两个文件构成。其中文件 `tst-driver.h` 的实现代码如下所示。

```
#ifndef __TST_DRIVER_H__
#define __TST_DRIVER_H__
#define TSTDRV_MAGIC          0xd0
#define GPIO_IN                0
```

```

#define GPIO_OUT                1//_IO(TSTDRV_MAGIC, 1)
#define GPIO_SET_BIT            2//_IO(TSTDRV_MAGIC, 2)
#define GPIO_CLR_BIT            3//_IO(TSTDRV_MAGIC, 3)
#define TSTDRV_MAXNR            4
#endif //ifndef __TST_DRIVER_H__

```

另一个构成文件 `tst-driver.c` 的实现代码如下所示。

```

#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/init.h> /* __init __exit */
#include <linux/types.h> /* size_t */
#include <linux/fs.h> /* file_operation */
// #include <linux/errno.h> /* Error number */
// #include <linux/delay.h> /* udelay */
#include <asm/uaccess.h> /* copy_to_user, copy_from_user */
#include <asm/hardware.h>
#include "tst-driver.h"
#define DRIVER_NAME "myDriver"
// #undef CONFIG_DEVFS_FS
#ifdef DEBUG
#define PRINTK(fmt, arg...) printk(KERN_NOTICE fmt, ##arg)
#else
#define PRINTK(fmt, arg...)
#endif
/*
    KERN_EMERG 用于紧急事件,一般是系统崩溃前的提示信息
    KERN_ALERT 用于需要立即采取动作的场合
    KERN_CRIT 为临界状态,通常设计验证的硬件或软件操作失败
    KERN_ERR 用于报告错误状态。设备驱动程序通常会用它报告来自硬件的问题
    KERN_WARNING 就可能出现的问题提出警告。这些问题通常不会对系统造成严重破坏
    KERN_NOTICE 有必要提示的正常情况。许多安全相关的情况用这个级别汇报
    KERN_INFO 提示性信息。有很多驱动程序在启动时用这个级别打印相关信息
    KERN_DEBUG 用于调试的信息
*/
static int myDriver_Major = 0; /* Driver Major Number */
/* Virtual Driver Buffer */
static unsigned char myDriver_Buffer[1024*1024];
/* Driver Operation Functions */
static int myDriver_open(struct inode *inode, struct file *filp)
{

```




```
// int Minor = MINOR(inode->i_rdev);
// filp->private_data = 0;
MOD_INC_USE_COUNT;
PRINTK("myDriver open called!\n");
return 0;
}
static int myDriver_release(struct inode *inode, struct file *filp)
{
// int Minor = MINOR(inode->i_rdev);
MOD_DEC_USE_COUNT;
PRINTK("myDriver release called!\n");
return 0;
}
static ssize_t myDriver_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
    size_t read_size = count;
    PRINTK("myDriver read called!\n");
    PRINTK("\tcount=%d, pos=%d\n", count, (int)*f_pos);
    if(*f_pos >= sizeof(myDriver_Buffer))
    {
        PRINTK("[myDriver read]Buffer Overlap\n");
        *f_pos = sizeof(myDriver_Buffer);
        return 0;
    }
    if((count + *f_pos) > sizeof(myDriver_Buffer))
    {
        PRINTK("count + f_pos > sizeof buffer\n");
        read_size = sizeof(myDriver_Buffer) - *f_pos;
    }
    copy_to_user(buf, &myDriver_Buffer[*f_pos], read_size);
    *f_pos += read_size;
    return read_size;
}
static ssize_t myDriver_write(struct file *filp, const char *buf, size_t count, loff_t
*f_pos)
{
    size_t fill_size = count;
    PRINTK("myDriver write called!\n");
    PRINTK("\tcount=%d, pos=%d\n", count, (int)*f_pos);
    if(*f_pos >= sizeof(myDriver_Buffer))
    {
        PRINTK("[myDriver write]Buffer Overlap\n");
        *f_pos = sizeof(myDriver_Buffer);
        return 0;
    }
    if((count + *f_pos) > sizeof(myDriver_Buffer))
```

```

    {
        PRINTK("count + f_pos > sizeof buffer\n");
        fill_size = sizeof(myDriver_Buffer) - *f_pos;
    }
    copy_from_user(&myDriver_Buffer[*f_pos], buf, fill_size);
    *f_pos += fill_size;
    return fill_size;
}

static int myDriver_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
unsigned long arg)
{
    PRINTK("myDriver ioctl called(%d)!\n", cmd);
    if(_IOC_TYPE(cmd) != TSTDRV_MAGIC)
    {
        return -ENOTTY;
    }
    if(_IOC_NR(cmd) >= TSTDRV_MAXNR)
    {
        return -ENOTTY;
    }
    switch(cmd)
    {
        case MYDRV_IOCTL0:
            PRINTK("IOCTRL 0 called(0x%lx)!\n", arg);
            break;
        case MYDRV_IOCTL1:
            PRINTK("IOCTRL 1 called(0x%lx)!\n", arg);
            break;
        case MYDRV_IOCTL2:
            PRINTK("IOCTRL 2 called(0x%lx)!\n", arg);
            break;
        case MYDRV_IOCTL3:
            PRINTK("IOCTRL 3 called(0x%lx)!\n", arg);
            break;
    }
    return 0;
}

/* 驱动操作结构 */
static struct file_operations myDriver_fops = {
    owner: THIS_MODULE,
    write: myDriver_write,
    read: myDriver_read,
    ioctl: myDriver_ioctl,
    open: myDriver_open,
    release: myDriver_release,
};

```



```
#ifndef CONFIG_DEVFS_FS
devfs_handle_t devfs_myDriver_dir;
devfs_handle_t devfs_myDriver_raw;
#endif

static int __init myModule_init(void)
{
    /* Module init code */
    PRINTK("myModule_init\n");
    /* Driver register */
    myDriver_Major = register_chrdev(0, DRIVER_NAME, &myDriver_fops);
    if(myDriver_Major < 0)
    {
        PRINTK("register char device fail!\n");
        return myDriver_Major;
    }
    PRINTK("register myDriver OK! Major = %d\n", myDriver_Major);
#ifdef CONFIG_DEVFS_FS
    devfs_myDriver_dir = devfs_mk_dir(NULL, "myDriver", NULL);
    devfs_myDriver_raw = devfs_register(devfs_myDriver_dir, "raw0", DEVFS_FL_DEFAULT,
myDriver_Major, 0, S_IFCHR | S_IRUSR | S_IWUSR, &myDriver_fops, NULL);
    PRINTK("add dev file to devfs OK!\n");
#endif
    return 0;
}

static void __exit myModule_exit(void)
{
    /* Module exit code */
    PRINTK("myModule_exit\n");
    /* Driver unregister */
    if(myDriver_Major > 0)
    {
#ifdef CONFIG_DEVFS_FS
        devfs_unregister(devfs_myDriver_raw);
        devfs_unregister(devfs_myDriver_dir);
#endif
        unregister_chrdev(myDriver_Major, DRIVER_NAME);
    }
    return;
}

MODULE_AUTHOR("SXZ");
MODULE_LICENSE("Dual BSD/GPL");
module_init(myModule_init);
module_exit(myModule_exit);
```

3.5.2 块设备驱动程序

块设备 I/O 与字符设备操作的主要区别如下所示。

- 块设备只能以块为单位接收输入、返回输出，而字符设备则以 Byte 为单位。大多数设备是字符设备，它们不需要缓冲并且不以固定块大小进行操作。
- 块设备对于 I/O 请求有对应的缓冲区，所以它们可以选择以什么顺序进行响应。字符设备无须缓冲且被直接读写。
- 字符设备只能被顺序读写，块设备可以随机访问。

1. 结构体 block_device_operations

在文件“include/linux/fs.h”中定义了结构体 block_device_operations，此结构体描述了对块设备的操作的集合，具体代码如下所示。

```
struct block_device_operations {
    int (*open) (struct inode *, struct file *); /*打开*/
    int (*release) (struct inode *, struct file *); /*释放*/
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned, unsigned long);
    int (*direct_access) (struct block_device *, sector_t, unsigned long *);
    int (*media_changed) (struct gendisk *); /*介质被改变?*/
    int (*revalidate_disk) (struct gendisk *); /*使介质改变*/
    int (*getgeo) (struct block_device *, struct hd_geometry *); /*填充驱动器信息*/
    struct module *owner; /*模块拥有者,一般初始化为 THIS_MODULE*/
};
```

2. 结构体 gendisk

结构体 gendisk 的功能是描述一个独立的磁盘设备或分区，具体代码如下所示。

```
struct gendisk{
    /*前三个元素共同表征了一个磁盘的主、次设备号,同一个磁盘的各个分区共享一个主设备号*/
    int major; /*主设备号*/
    int first_minor; /*第一个次设备号*/
    int minors; /*最大的次设备数,如果不能分区,则为 1*/
    char disk_name[32];
    struct hd_struct** part; /*磁盘上的分区信息*/
    struct block_device_operations* fops; /*块设备操作,block_device_operations*/
    struct request_queue* queue; /* 请求队列,用于管理该设备 I/O 请求队列的指针*/
    void* private_data; /*私有数据*/
    sector_t capacity; /*扇区数,512 字节为 1 个扇区,描述设备容量*/
    //.....
};
```

3. 结构体 request 和 bio

(1) 请求 request

结构体 request 和 request_queue 在 Linux 块设备驱动中，使用 request 结构体来表征（显示出



来的现象；表现出来的特征）等待进行的 I/O 请求，用 `request_queue` 来表征一个块 I/O 请求队列。这两个结构体的定义代码如下所示。

```
struct request{
    struct list_head queuelist;
    unsigned long flags;
    sector_t sector; /*要传输的下一个扇区*/
    unsigned long nr_sectors; /*要传送的扇区数目*/
    unsigned int current_nr_sector; /*当前要传送的扇区*/
    sector_t hard_sector; /*要完成的下一个扇区*/
    unsigned long hard_nr_sectors; /*要被完成的扇区数目*/
    unsigned int hard_cur_sectors; /*当前要被完成的扇区数目*/
    struct bio* bio; /*请求的 bio 结构体的链表*/
    struct bio* biotail; /*请求的 bio 结构体的链表尾*/
    /*请求在物理内存中占据的不连续的段的数目*/
    unsigned short nr_phys_segments;
    unsigned short nr_hw_segments;
    int tag;
    char* buffer; /*传送的缓冲区,内核的虚拟地址*/
    int ref_count; /*引用计数*/
    ...
};
```

（2）请求队列 `request_queue`

请求队列跟踪等候块 I/O 的请求，功能是描述这个设备能够支持的请求的类型信息。请求队列还要实现一个插入接口，这个接口允许使用多个 I/O 调度器，I/O 调度器以最优性能的方式向驱动提交 I/O 请求。大部分 I/O 调度器是积累批量的 I/O 请求，并将其排列为递增/递减的块索引顺序后提交给驱动。另外 I/O 调度器还负责合并邻近的请求，当一个新的 I/O 请求被提交给调度器后，它会在队列里搜寻包含邻近的扇区的请求。如果找到一个并且此请求合理，则调度器会将这两个请求合并。

定义结构体 `request_queue` 的代码如下所示。

```
struct request_queue{
    ...
    /*自旋锁,保护队列结构体*/
    spinlock_t __queue_lock;
    spinlock_t* queue_lock;
    struct kobject kobj; /*队列 kobject*/
    /*队列设置*/
    unsigned long nr_requests; /*最大的请求数量*/
    unsigned int nr_congestion_on;
    unsigned int nr_congestion_off;
    unsigned int nr_batching;
    unsigned short max_sectors; /*最大扇区数*/
    unsigned short max_hw_sectors;
```



```

unsigned short max_phys_sectors; /*最大的段数*/
unsigned short max_hw_segments;
unsigned short hardsect_size; /*硬件扇区尺寸*/
unsigned int max_segment_size; /*最大的段尺寸*/
unsigned long seg_boundary_mask; /*段边界掩码*/
unsigned int dma_alignment; /*DMA 传送内存对齐限制*/
struct blk_queue_tag* queue_tags;
atomic_t refcnt; /*引用计数*/
unsigned int in_flight;
unsigned int sg_timeout;
unsigned int sg_reserved_size;
int node;
struct list_head drain_list;
struct request* flush_rq;
unsigned char ordered;
};

```

注意：在块设备模块中，还可以使用函数实现块设备驱动的模块卸载、加载、打开与释放操作，相关知识请参阅相关资料。

Android 的块设备驱动在目录“/dev/block”中，其主要内容如下所示。

```

brw----- root    root    179,  2 2010-03-29 23:33 mmcblk0p2
brw----- root    root    179,  1 2010-03-29 23:33 mmcblk0p1
brw----- root    root    179,  0 2010-03-29 23:33 mmcblk0
brw----- root    root    31,   6 2010-03-29 23:33 mtddb0ck6
brw----- root    root    31,   5 2010-03-29 23:33 mtddb0ck5
brw----- root    root    31,   4 2010-03-29 23:33 mtddb0ck4
brw----- root    root    31,   3 2010-03-29 23:33 mtddb0ck3
brw----- root    root    31,   2 2010-03-29 23:33 mtddb0ck2
brw----- root    root    31,   1 2010-03-29 23:33 mtddb0ck1
brw----- root    root    31,   0 2010-03-29 23:33 mtddb0ck0
brw----- root    root     7,   7 2010-03-29 23:33 loop7
brw----- root    root     7,   6 2010-03-29 23:33 loop6
brw----- root    root     7,   5 2010-03-29 23:33 loop5
brw----- root    root     7,   4 2010-03-29 23:33 loop4
brw----- root    root     7,   3 2010-03-29 23:33 loop3
brw----- root    root     7,   2 2010-03-29 23:33 loop2
brw----- root    root     7,   1 2010-03-29 23:33 loop1
brw----- root    root     7,   0 2010-03-29 23:33 loop0
brw----- root    root     1,   0 2010-03-29 23:33 ram0
brw----- root    root     1,   0 2010-03-29 23:33 ram1

```

在上述内容中，主设备号为 1 的是各个内存块设备，主设备号为 7 的是各个回环块设备，主设备号为 31 的是 mtd 设备中的块设备，mmcblk0 表示 SD 卡的块设备。

在 Android 系统中，可以使用 mount 命令来查看系统中被挂起的文件系统。使用 mount 命令



的格式如下所示。

```
mount [-t vfstype] [-o options] device dir
```

上述命令中的主要参数的具体说明如下所示。

① **-t vfstype**: 用于指定文件系统的类型, 通常不必指定。**mount** 会自动选择正确的类型。常用类型有如下 6 类。

- 光盘或光盘镜像: **iso9660**。
- DOS fat16 文件系统: **msdos**。
- Windows 9x fat32 文件系统: **vfat**。
- Windows NT ntfs 文件系统: **ntfs**。
- Mount Windows 文件网络共享: **smbfs**。
- UNIX(LINUX) 文件网络共享: **nfs**。

② **-o options**: 主要用来描述设备或档案的挂接方式。常用的参数有如下 4 类。

- **loop**: 用来把一个文件当成硬盘分区挂接上系统。
- **ro**: 采用只读方式挂接设备。
- **rw**: 采用读写方式挂接设备。
- **iocharset**: 指定访问文件系统所用字符集。

③ **device**: 要挂接 (**mount**) 的设备。

④ **dir**: 设备在系统上的挂接点 (**mount point**)。

在 Android 系统中, 可以使用 **df** 命令来查看系统中各个盘的使用情况。使用 **df** 命令的格式如下所示。

```
df [options]
```

参数 **options** 常用取值的具体说明如下所示。

- **-s**: 对每个 **Names** 参数只给出占用的数据块总数。
- **-a**: 递归地显示指定目录中各文件及子目录中各文件占用的数据块数。若既不指定 **-s**, 也不指定 **-a**, 则只显示 **Names** 中的每一个目录及其中的各子目录所占的磁盘块数。
- **-k**: 以 1024 字节为单位列出磁盘空间使用情况。
- **-x**: 跳过在不同文件系统上的目录, 不予统计。
- **-l**: 计算所有的文件大小, 对硬链接文件则计算多次。
- **-i**: 显示 **inode** 信息而非块使用量。
- **-h**: 以容易理解的格式输出文件系统大小, 例如 136KB、254MB、21GB。
- **-P**: 使用 POSIX 输出格式。
- **-T**: 显示文件系统类型。

3.5.3 网络设备驱动程序

Linux 网络设备驱动程序由 4 部分组成, 分别是网络设备媒介层、网络设备驱动层、网络设备接口层及网络协议接口层。网络设备媒介层包括各种物理网络设备和传输媒介。对于网络设备

接口层，Linux 系统用 `Net_device` 结构表示网络设备接口。`Net_device` 结构保存所有与硬件有关的接口信息，各协议软件主要通过 `Net_device` 结构来完成与硬件的交互。网络设备驱动层主要包括网络设备的初始化、数据包的发送和接收。网络协议接口层提供网络接口驱动程序的抽象接口。

Linux 网络驱动程序中的常用方法如下所示。

- 初始化 (`initialize`)：检测设备；配置和初始化硬件；初始化 `net_device` 结构；注册设备。
- 打开 (`open`)：这个方法在网络设备被激活的时候调用，进行资源的申请和硬件的激活等。`open` 方法另一个作用是如果驱动程序作为一个模块被装入，则要防止模块卸载时设备处于打开状态。在 `open` 方法里要调用 `MOD_INC_USE_COUNT` 宏。
- 关闭 (`close`)：释放某些系统资源。如果是作为模块装入的驱动程序，`close` 里应该调用 `MOD_DEC_USE_COUNT`，减少设备被引用的次数，以使驱动程序可以被卸载。
- 发送 (`hard_start_xmit`)：网络设备驱动程序发送数据时，系统调用 `dev_queue_xmit` 函数，发送的数据放在一个 `sk_buff` 结构中。一般的驱动程序将数据传输到硬件发出去，特殊的设备如 `loopback` 把数据组成一个接收数据再回送给系统，或如 `dummy` 设备直接丢弃数据。如果发送成功，则在 `hard_start_xmit` 方法里释放 `sk_buff`，返回 0，否则返回 1。
- 接收 (`reception`)：驱动程序并不存在一个接收方法。有数据收到应该是驱动程序来通知系统的。一般设备收到数据后都会产生一个中断，在中断处理程序中驱动程序申请一块 `sk_buff`，从硬件读出数据放置到申请好的缓冲区里。接下来填充 `sk_buff` 中的一些信息。最后调用 `netif_rx()` 把数据传送给上层协议层处理。

在 Android 系统中，可以使用命令 `ifconfig` 来查询系统中的网络设备，另外使用此命令也可以获取 Wi-Fi 网络和电话网络的信息。