第 1 章在宏观上介绍了 Android 的结构和原理，同时，搭建了后面我们将要使用的开发环境。本章将开始对 Android 的内核进行剖析，主要介绍 Android 和 Linux 之间的关系，以及 Android 系统在 Linux 系统之上扩展的部分功能和驱动。难度可能比上一章大一点，但是不用担心，我们会带着上述问题详细讲解的。

2.1　Linux 与 Android 的关系

虽然 Android 基于 Linux 内核，但是它与 Linux 之间还是有很大的差别，比如 Android 在 Linux 内核的基础上添加了自己所特有的驱动程序。下面我们就来分析一下它们之间究竟有什么关系？

2.1.1　为什么会选择 Linux

成熟的操作系统有很多，但是 Android 为什么选择采用 Linux 内核呢？这就与 Linux 的一些特性有关了，比如：

- 强大的内存管理和进程管理方案
- 基于权限的安全模式
- 支持共享库
- 经过认证的驱动模型
- Linux 本身就是开源项目

更多关于上述特性的信息可以参考 Linux 2.6 版内核的官方文档，这便于我们在后面的学习中更好地理解 Android 所特有的功能特性。接下来分析 Android 与 Linux 的关系。

2.1.2　Android 不是 Linux

看到这个标题大家可能会有些迷惑，前面不是一直说 Android 是基于 Linux 内核的吗，怎么现在又不是 Linux 了？迷惑也是正常的，请先看下面几个要点，然后我们将对每一个要点进行分析，看完后你就会觉得 Android 不是 Linux 了。

- 它没有本地窗口系统
- 它没有 glibc 的支持
- 它并不包括一整套标准的 Linux 使用程序
- 它增强了 Linux 以支持其特有的驱动

1．它没有本地窗口系统

什么是本地窗口系统呢？本地窗口系统是指 GNU/Linux 上的 X 窗口系统，或者 Mac OX X 的 Quartz 等。不同的操作系统的窗口系统可能不一样，Android 并没有使用（也不需要使用）Linux 的 X 窗口系统，这是 Android 不是 Linux 的一个基本原因。

2．它没有 glibc 支持

由于 Android 最初用于一些便携的移动设备上，所以，可能出于效率等方面的考虑，Android 并没有采用 glibc 作为 C 库，而是 Google 自己开发了一套 Bionic Libc 来代替 glibc。

3．它并不包括一整套标准的 Linux 使用程序

Android 并没有完全照搬 Liunx 系统的内核，除了修正部分 Liunx 的 Bug 之外，还增加了不少内容，比如：它基于 ARM 构架增加的 Gold-Fish 平台，以及 yaffs2 FLASH 文件系统等。

4．Android 专有的驱动程序

除了上面这些不同点之外，Android 还对 Linux 设备驱动进行了增强，主要如下所示。

1）Android Binder　基于 OpenBinder 框架的一个驱动，用于提供 Android 平台的进程间通信（InterProcess Communication，IPC）功能。源代码位于 drivers/staging/android/binder.c。

2）Android 电源管理（PM）　一个基于标准 Linux 电源管理系统的轻量级 Android 电源管理驱动，针对嵌入式设备做了很多优化。源代码位于：

- kernel/power/earlysuspend.c
- kernel/power/consoleearlysuspend.c
- kernel/power/fbearlysuspend.c
- kernel/power/wakelock.c
- kernel/power/userwakelock.c

3）低内存管理器（Low Memory Killer）　比 Linux 的标准的 OOM（Out Of Memory）机制更加灵活，它可以根据需要杀死进程以释放需要的内存。源代码位于 drivers/staging/android/lowmemorykiller.c。

4）匿名共享内存（Ashmem）　为进程间提供大块共享内存，同时为内核提供回收和管理这个内存的机制。源代码位于 mm/ashmem.c。

5）Android PMEM（Physical）　PMEM 用于向用户空间提供连续的物理内存区域，DSP 和某些设备只能工作在连续的物理内存上。源代码位于 drivers/misc/pmem.c。

6）Android Logger　一个轻量级的日志设备，用于抓取 Android 系统的各种日志。源代码位于 drivers/staging/android/logger.c。

7）Android Alarm　提供了一个定时器，用于把设备从睡眠状态唤醒，同时它还提供了一个即使在设备睡眠时也会运行的时钟基准。源代码位于 drivers/rtc/alarm.c。

8）USB Gadget 驱动　一个基于标准 Linux USB gadget 驱动框架的设备驱动，Android 的 USB 驱动是基于 gaeget 框架的。源代码位于 drivers/usb/gadget/。

9）Android Ram Console　为了提供调试功能，Android 允许将调试日志信息写入一个被称为 RAM Console 的设备里，它是一个基于 RAM 的 Buffer。源代码位于 drivers/staging/android/ram_console.c。

10）Android timed device　提供了对设备进行定时控制的功能，目前支持 vibrator 和 LED

设备。源代码位于 drivers/staging/android /timed_output.c(timed_gpio.c)。

11）**Yaffs2 文件系统**　Android 采用 Yaffs2 作为 MTD nand flash 文件系统，源代码位于 fs/yaffs2/目录下。Yaffs2 是一个快速稳定的应用于 NAND 和 NOR Flash 的跨平台的嵌入式设备文件系统，同其他 Flash 文件系统相比，Yaffs2 能使用更小的内存来保存其运行状态，因此它占用内存小。Yaffs2 的垃圾回收非常简单而且快速，因此能表现出更好的性能。Yaffs2 在大容量的 NAND Flash 上的性能表现尤为突出，非常适合大容量的 Flash 存储。

上面这些要点足以说明 Android 不是 Linux。本书的主要内容将围绕 Android 的这些特有的部分展开，我们的讲解会尽量通俗易懂，但还是建议大家先复习一下 Linux 内核的基本知识。在具体学习之前，我们还是先来总体浏览一下 Android 对 Linux 内核进行了哪些改动，在移植时就需要对这些改动加以调整。

2.2　Android 对 Linux 内核的改动

Android 从多个方面对 Linux 内核进行了改动与增强，下面将对此进行详细介绍和分析。

2.2.1　Goldfish

Android 模拟器通过运行一个 Goldfish 的虚拟 CPU.Goldfish 来运行 arm926t 指令集(arm926t 属于 armv5 构架)，并且仿真了输入/输出，比如键盘输入和 LCD 输出。这个模拟器其实是在 qemu 之上开发的，输入/输出是基于 libSDL 的。既然 Goldfish 是被模拟器运行的虚拟 CPU，那么当 Android 在真实的硬件设备上运行时，我们就需要去掉它，因此，只有知道 Google 对 Goldfish 做了哪些具体改动之后才能正确地去掉。据统计，Android 内核对 Goldfish 的改动主要涉及 44 个文件，具体汇总如下。

说明　本书中在被改动的文件前面加了 Chg 标记，在新增的文件前面加了 New 标记。

1	Chg	arch/arm/Makefile	添加 CONFIG_ARCH_GOLDFISH
2	New	arch/arm/configs/goldfish_defconfig	默认配置文件
3	New	arch/arm/mach-goldfish/Kconfig	为 Goldfish CPU 添加 Kernel 配置文件
4	New	arch/arm/mach-goldfish/Makefile	添加 board-goldfish.o
5	New	arch/arm/mach-goldfish/Makefile.boot	为 Goldfish CPU 进行启动配置
6	New	arch/arm/mach-goldfish/audio.c Audio	的输入/输出
7	New	arch/arm/mach-goldfish/board-goldfish.c	中断请求、输入/输出等
8	New	arch/arm/mach-goldfish/pdev_bus.c	设备总线
9	New	arch/arm/mach-goldfish/pm.c	电源管理
10	New	arch/arm/mach-goldfish/switch.c	Switch 控制
11	New	arch/arm/mach-goldfish/timer.c	获取和设置时间
12	Chg	arch/arm/mm/Kconfig	添加 ARCH_GOLDFISH 到支持列表
13	Chg	drivers/char/Makefile	添加 goldfish_tty

14	New	drivers/char/goldfish_tty.c	TTY 驱动
15	Chg	drivers/input/keyboard/Kconfig	为 Goldfish 的键盘事件添加配置文件
16	Chg	drivers/input/keyboard/Makefile	添加 goldfish_events 事件
17	New	drivers/input/keyboard/goldfish_events.c	Goldfish 键盘驱动
18	Chg	drivers/mmc/host/Kconfig	添加 Kernel 配置选项 Goldfish MMC 卡
19	Chg	drivers/mmc/host/Makefile	添加 Goldfish MMC 卡驱动
20	New	drivers/mmc/host/goldfish.c	多媒体驱动
21	Chg	drivers/mtd/devices/Kconfig	为 Goldfish 的 NAND flash device 添加 Kernel 配置选项
22	Chg	drivers/mtd/devices/Makefile	添加 goldfish_nand
23	New	drivers/mtd/devices/goldfish_nand.c	NAND flash 驱动
24	New	drivers/mtd/devices/goldfish_nand_reg.h	NAND flash 驱动
25	Chg	drivers/power/Kconfig	为 Goldfish 的 battery（电池）驱动添加 kernel 配置选项
26	Chg	drivers/power/Makefile	添加 Goldfish 电池
27	New	drivers/power/goldfish_battery.c	能源和电池状态驱动
28	Chg	drivers/rtc/Kconfig	为 Goldfish 的 rtc（时钟）驱动添加 Kernel 配置选项
29	Chg	drivers/rtc/Makefile	添加 rtc-goldfish
30	New	drivers/rtc/rtc-goldfish.c	实时时钟驱动
31	Chg	drivers/video/Kconfig	添加 Goldfish 的 framebuffer
32	Chg	drivers/video/Makefile	添加 Goldfish 的 framebuffer
33	New	drivers/video/goldfishfb.c	framebuffer 驱动
34	New	include/asm-arm/arch-goldfish/dma.h	
35	New	include/asm-arm/arch-goldfish/entry-macro.S	
36	New	include/asm-arm/arch-goldfish/hardware.h	
37	New	include/asm-arm/arch-goldfish/io.h	
38	New	include/asm-arm/arch-goldfish/irqs.h	
39	New	include/asm-arm/arch-goldfish/memory.h	
40	New	include/asm-arm/arch-goldfish/system.h	
41	New	include/asm-arm/arch-goldfish/timer.h	
42	New	include/asm-arm/arch-goldfish/timex.h	
43	New	include/asm-arm/arch-goldfish/uncompress.h	
44	New	include/asm-arm/arch-goldfish/vmalloc.h	

2.2.2 YAFFS2

不同于 PC 机（文件是存储在硬盘上的），手机使用 FLASH 作为存储介质。HTC 的 G1 使用的是 NANDFLASH——这种存储目前已经相当普及了，而且种类也颇多（如 SLC、MLC 等），存储密度也越来越高（已经出现几十 GB 大小的 NANDFLASH），价格也越来越低。

YAFFS2 是专门用在 FLASH 上的文件系统，YAFFS2 是“Yet Another Flash File System，2nd edition”的缩写。YAFFS2 为 Linux 内核提供了一个高效访问 NANDFLASH 的接口。但是 NANDFLASH 的支持并不包含在标准的 2.6.25 内核中，所以 Google 在其中添加了对 NANDFLASH 的支持。据统计，为了支持 YAFFS2，Google 一共改动和增加了以下 35 个文件：

1	Chg	fs/Kconfig	添加 YAFFS 配置
2	Chg	fs/Makefile	添加 YAFFS

以下为新增的 YAFFS2：

1	New	fs/yaffs2/Kconfig	18	New	fs/yaffs2/yaffs_mtddif2.h
2	New	fs/yaffs2/Makefile	19	New	fs/yaffs2/yaffs_nand.c
3	New	fs/yaffs2/devextras.h	20	New	fs/yaffs2/yaffs_nand.h
4	New	fs/yaffs2/moduleconfig.h	21	New	fs/yaffs2/yaffs_nandemul2k.h
5	New	fs/yaffs2/yaffs_checkptrw.c	22	New	fs/yaffs2/yaffs_packedtags1.c
6	New	fs/yaffs2/yaffs_checkprtw.h	23	New	fs/yaffs2/yaffs_packedtags1.h
7	New	fs/yaffs2/yaffs_ecc.c	24	New	fs/yaffs2/yaffs_packedtags2.c
8	New	fs/yaffs2/yaffs_ecc.h	25	New	fs/yaffs2/yaffs_packedtags2.h
9	New	fs/yaffs2/yaffs_fs.c	26	New	fs/yaffs2/yaffs_qsort.c
10	New	fs/yaffs2/yaffs_getblockinfo.h	27	New	fs/yaffs2/yaffs_qsort.h
11	New	fs/yaffs2/yaffs_guts.c	28	New	fs/yaffs2/yaffs_tagscompat.c
12	New	fs/yaffs2/yaffs_guts.h	29	New	fs/yaffs2/yaffs_tagscompat.h
13	New	fs/yaffs2/yaffs_mtdif.c	30	New	fs/yaffs2/yaffs_tagsvaliditiy.c
14	New	fs/yaffs2/yaffs_mtdif.h	31	New	fs/yaffs2/yaffs_tagsvalidity.h
15	New	fs/yaffs2/yaffs_mtddif1.c	32	New	fs/yaffs2/yaffsinterface.h
16	New	fs/yaffs2/yaffs_mtddif1.h	33	New	fs/yaffs2/yportenv.h
17	New	fs/yaffs2/yaffs_mtddif2.c			

2.2.3　蓝牙

在蓝牙通信协议栈里 Google 修改了 10 个文件。这些改动修复了一些与蓝牙耳机相关的明显的 Bug，以及一些与蓝牙调试和访问控制相关的函数，具体如下所示。

1	Chg	drivers/bluetooth/Kconfig	添加 HCI UART Debug
2	Chg	drivers/bluetooth/hci_II.c	如果 HCI UART Debug 定义在 Kernel 配置中，则添加 BT_DBG()宏
3	Chg	net/bluetooth/Kconfig	添加配置选项 L2CAP, HCI_CORE, HCI_SOCK, 以及通用接口和语音
4	Chg	net/bluetooth/af_bluetooth.c	如果 CONFIG_ANDROID_PARANOID_NETWORK 被定义，则添加蓝牙功能的安全检查
5	Chg	net/bluetooth/hci_event.c	修正蓝牙的加密 Bug 和增加语音的支持
6	Chg	net/bluetooth/rfcomm/core.c	修正 Bug
7	Chg	net/bluetooth/rfcomm/sock.c	修复 Bug
8	Chg	net/bluetooth/sco.c	禁用 SCO 链接
9	Chg	include/net/bluetooth/hci_core.h	禁用 LMP_ESCO
10	Chg	include/net/bluetooth/rfcomm.h	在 rfcomm_dlc 中添加“out”参数

2.2.4　调度器（Scheduler）

Android 内核还修改了与进程调度和时钟相关的策略。只改动了 5 个文件，如下：

1	Chg	kernel/sched.c	添加 NORMALIZED_SLEEPER
2	Chg	kernel/sched_fair.c	修改内核的调度方式
3	Chg	kernel/softirq.c	修改为 CPU 调度
4	Chg	kernel/time/tick-sched.c	修改为 CPU 调度
5	Chg	include/linux/tick.h	如果 CONFIG_NO_HZ 被定义，则添加 tick_nohz_ update_ stopped_ sched_tick()

2.2.5 Android 新增的驱动

Android 在 Linux 的基础上新增了许多特有的驱动，如下所示。

1）IPC Binder　一种 IPC（进程间通信）机制。它的进程能够为其他进程提供服务——通过标准的 Linux 系统调用 API。IPC Binder 的概念起源于一家名为 Be.Inc 的公司，在 Google 之前就已经被 Palm 软件采用了。

2）Low Memory Killer　其实内核里已经有一个类似的功能，名称为 oom killer（out of memory killer）。当内存不够的时候，该策略会试图结束一个进程。

3）Ashmem　匿名共享内存。该功能使得进程间能够共享大块的内存。比如说，系统可以使用 Ashmem 保存一些图标，多个应用程序可以访问这个共享内存来获取这些图标。Ashmem 为内核提供了一种回收这些使用完的共享内存块的方法，如果某个进程试图访问这些已经被回收的内存块，它将会得到错误的返回值，以便它重新进行内存块分配和数据初始化。

4）RAM Console and Log Device　为了调试方便，Android 添加了一个功能，使调试信息可以输入到一个内存块中。此外，Android 还添加了一个独立的日志模块，这样用户空间的进程就能够读写日志消息，以及调试打印信息等。

5）Android Debug Bridge　嵌入式设备的调试的确比较麻烦，为了便于调试，Google 设计了这个调试工具，可以简称为 ADB，使用 USB 作为连接方式，ADB 可以看做是连接 Android 设备和 PC 机的一套协议。

除了这些主要的功能之外，Android 还增加了诸如 real-time clock、switch、timed GPIO 等功能，所有这些改动和增加包含在以下 28 个文件之中。

1	Chg	drivers/Kconfig	进入配置文件
2	Chg	drivers/Makefile	添加 switch，驱动等
3	New	drivers/android/Kconfig	添加 BINDER_IPC、POWER、POWER_STAT、POWER_ ALARM、LOGGER、RAM_CONSOLE、TIMED_GPIO、PARANOID_NETWORK 到配置中
4	New	drivers/android/Makefile	添加 binder.o、power.o、alarm.o、logger.o、ram_console.o、timed_gpio
5	New	drivers/android/alarm.c	系统硬件时钟和实时时钟管理
6	New	drivers/android/binder.c	IPC 机制（Binder）
7	New	drivers/android/logger.c	Google 的日志 API

8	New	drivers/android/ram_console.c	RAM 控制台和日志设备方便调试[⊖]
9	New	drivers/android/timed_gpio.c	Google 的 GPIO 定时驱动
10	New	drivers/switch/Kconfig	为 GPIO 添加配置选项
11	New	drivers/switch/Makefile	引入 GPIO 驱动
12	New	drivers/switch/switch_class.c	
13	New	drivers/switch/switch_gpio.c	
14	Chg	drivers/usb/gadget/Kconfig	添加 ADB 配置选项
15	Chg	drivers/usb/gadget/Makefile	编译 ADB 所需的配置选项
16	New	drivers/usb/gadget/android_adb.c	ADB 驱动
17	New	include/linux/android_aid.h	添加 AIDs、INET、networking
18	New	include/linux/android_alarm.h	时钟功能设置
19	New	include/linux/android_timed_gpio.h	GPIO 结构体
20	New	include/linux/ashmem.h	Android 共享内存
21	New	include/linux/binder.h	Binder IPC API 定义
22	New	include/linux/logger.h	Logger 定义
23	New	include/linux/switch.h	GPIO switch 接口
24	Chg	mm/Makefile	添加 ashmem.o
25	New	mm/ashmem.c	内存共享实现
26	Chg	drivers/misc/Kconfig	添加 LOW_MEMORY_KILLER 配置选项
27	Chg	drivers/misc/Makefile	添加 lowmemorykiller.c
28	New	drivers/misc/lowmemorykiller.c	当内存过低时，选择并结束进程

2.2.6　电源管理

电源管理（Power Management）对于移动设备来说相当重要，也是最为复杂和开发难度最高的一个功能。Google 添加了一个新的电源管理系统，不包含原有的 apm 和 dpm 等。这项改动主要涉及以下 5 个文件：

1	New	include/linux/android_power.h	定义电源管理 API
2	New	drivers/android/power.c	电源管理 API 实现
3	Chg	drivers/input/evdev.c	修改 Android 电源处理方式
4	Chg	fs/inotify_user.c	修改 Android 电源处理方式
5	Chg	kernel/power/process.c	修改 Android 电源处理方式

2.2.7　杂项

除了上述改动之外，还有一些小改动，如新增的额外调试功能、键盘背光控制、TCP 网络管理等，共涉及 36 个文件，如下所示。

⊖ 为了调试方便，Android 添加了一个功能，使得调试信息可以输入到一个内存块中。此外， Android 添加了一个独立的日志模块，这样用户空间的进程能够读写日志消息，调试打印信息等。

1	New	Documentation/vm/pagemap.txt	
2	Chg	arch/arm/Kconfig	添加 HAVE_LATENCYTOP_SUPPORT 和 ARCH_GOLDFISH
3	Chg	arch/arm/kernel/process.c	添加 dump_task_regs 方法
4	Chg	arch/arm/kernel/signal.c	解决系统无法重新启动的问题
5	Chg	arch/arm/kernel/stacktrace.c	改进调试栈跟踪
6	Chg	arch/arm/mm/abort-ev6.S	
7	Chg	drivers/char/Kconfig	添加 Memory device driver 和 Goldfish TTY driver
8	Chg	drivers/char/mem.c	使编译结果输出到/dev/kmem and /dev/mem
9	Chg	drivers/leds/Kconfig	当 CPU 运行时打开 LEDS，但是屏幕是关闭的
10	Chg	drivers/leds/Makefile	添加编译 ledtrig-sleep.o
11	New	drivers/leds/ledtrig-sleep.c	睡眠（当关闭屏幕后 CPU 仍然运行）
12	Chg	drivers/rtc/class.c	修正实时时钟误差的 Bug
13	Chg	fs/fat/dir.c	添加 VFAT_IOCTL_GET_VOLUME_ID 到 fat_dir_ioctl()
14	Chg	fs/fat/inode.c	
15	Chg	fs/proc/base.c	当内存不足时调整/proc 文件
16	Chg	fs/proc/proc_misc.c	修正 kpagecount_read 和 kpageflags_read 返回的一些错误
17	Chg	fs/proc/task_mmu.c	简化 add_to_pagemap 中的错误检查
18	Chg	include/asm-arm/elf.h	添加 ELF_CORE_COPY_TASK_REGS()宏调用 dump_task_ regs(...)
19	Chg	include/linux/mm.h	添加 shmem_set_file(...)函数原型
20	Chg	include/linux/msdos_fs.h	添加 VFAT_IOCTL_GET_VOLUME_ID 宏
21	Chg	kernel/hrtimer.c	修复 run_hrtimer_pending 错误
22	Chg	init/Kconfig	添加 PANIC_TIMEOUT 默认为 0
23	Chg	kernel/panic.c	设置默认的 panic_timeout：从 kernel 配置到 PANIC_TIMEOUT
24	Chg	kernel/power/console.c	修复虚拟控制台的错误信息
25	Chg	kernel/printk.c	修复 printk 错误
26	Chg	mm/filemap.c	修正 filemap_fault
27	Chg	mm/shmmem.c	重构 shmem_zero_setup
28	Chg	mm/tiny-shmem.c	重构 shmem_zero_setup
29	Chg	include/linux/sockios.h	添加 SIOCKILLADDR 控制
30	Chg	include/net/tcp.h	添加 tcp_v4_nuke_addr 函数
31	Chg	net/ipv4/Makefile	如果设置 CONFIG_SYSFS，编译 sysfs_net_ipv4
32	Chg	net/ipv4/af_inet_c	如果定义 CONFIG_ANDROID_PARANOID_NETWORK，则添加安全检查
33	Chg	net/ipv4/devinet.c	添加 SIOCKILLADDR
34	Chg	net/ipv4/sysfs_net_ipv4.c	控制 TCP 窗口长度
35	Chg	net/ipv4/tcp_ipv4.c	添加 tcp_v4_nuke_addr 函数
36	Chg	net/ipv6/af_inet6.c	如果定义 CONFIG_ANDROID_PARANOID_NETWORK，则添加安全检查

上面这些看似简单，但是非常重要，当大家进行系统级应用开发和程序移植时都需要研究这些文件。对于每个文件的具体改动方式和实现，我们需要进一步查看 Android 的内核源代码，这是后面将要详细讲解的内容。

2.3 Android 对 Linux 内核的增强

2.2 节介绍了 Android 对 Linux 内核的改动，这一节将重点介绍 Android 对 Linux 内核的增

强，主要包括 Alarm（硬件时钟）、Ashmem（匿名内存共享）、Low Memory Killer（低内存管理）、Logger（日志设备），等等。

2.3.1　Alarm（硬件时钟）

Alarm 就是一个硬件时钟，前面我们已经知道它提供了一个定时器，用于把设备从睡眠状态唤醒，同时它也提供了一个在设备睡眠时仍然会运行的时钟基准。在应用层上，有关时间的应用都需要 Alarm 的支持，源代码位于“drivers/rtc/alarm.c”。

Alarm 的设备名为“/dev/alarm”。该设备的实现非常简单，我们首先打开源码，可以看到 include <linux/android_alarm.h>，其中定义了一些 Alarm 的相关信息。Alarm 的类型枚举如下：

```
enum android_alarm_type {
    ANDROID_ALARM_RTC_WAKEUP,
    ANDROID_ALARM_RTC,
    ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP,
    ANDROID_ALARM_ELAPSED_REALTIME,
    ANDROID_ALARM_SYSTEMTIME,
    ANDROID_ALARM_TYPE_COUNT,
};
```

主要包括了 5 种类型的 Alarm，_WAKEUP 类型表示在触发 Alarm 时需要唤醒设备，反之则不需要唤醒设备；ANDROID_ALARM_RTC 类型表示在指定的某一时刻出发 Alarm；ANDROID_ALARM_ELAPSED_REALTIME 表示在设备启动后，流逝的时间达到总时间之后触发 Alarm；ANDROID_ALARM_SYSTEMTIME 类型则表示系统时间；ANDROID_ALARM_TYPE_COUNT 则是 Alram 类型的计数。

> **注意**　流逝的时间也包括设备睡眠的时间，流逝时间的计算点从它最后一次启动算起。

Alarm 返回标记的枚举类型如下：

```
enum android_alarm_return_flags {
    ANDROID_ALARM_RTC_WAKEUP_MASK = 1U << ANDROID_ALARM_RTC_WAKEUP,
    ANDROID_ALARM_RTC_MASK = 1U << ANDROID_ALARM_RTC,
    ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP_MASK =
                1U << ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP,
    ANDROID_ALARM_ELAPSED_REALTIME_MASK =
                1U << ANDROID_ALARM_ELAPSED_REALTIME,
    ANDROID_ALARM_SYSTEMTIME_MASK = 1U << ANDROID_ALARM_SYSTEMTIME,
    ANDROID_ALARM_TIME_CHANGE_MASK = 1U << 16
};
```

Alarm 返回标记会随着 Alarm 的类型而改变。最后还定义了一些宏，主要包括禁用 Alarm、Alarm 等待、设置 Alarm 等。下面我们来分析 Alarm 驱动的具体实现。

首先，Alarm 的初始化及退出由以下三个函数来完成：

- late_initcall(alarm_late_init);
- module_init(alarm_init);
- module_exit(alarm_exit);

其中 alarm_init 函数对 Alarm 执行初始化操作，alarm_late_init 需要在初始化完成之后进行调用，最后退出时需要调用 alarm_exit 来销毁和卸载 Alarm 接口及驱动。

1．alarm_init

在初始化过程中，首先需要初始化系统时间，通过 platform_driver_register 函数来注册 Alarm 驱动的相关参数，具体如下所示：

```
static struct platform_driver alarm_driver = {
    .suspend = alarm_suspend,
    .resume = alarm_resume,
    .driver = {
        .name = "alarm"
    }
};
```

该参数主要指定了当系统挂起（suspend）和唤醒（Desume）所需要实现的分别为 alarm_suspend 和 alarm_resume，同时将 Alarm 设备驱动的名称设置为了“alarm”。

如果设置正确，那么继续通过如下代码来初始化 SUSPEND lock，因为在使用它们之前必须执行初始化操作。

```
wake_lock_init(&alarm_wake_lock, WAKE_LOCK_SUSPEND, "alarm");
wake_lock_init(&alarm_rtc_wake_lock, WAKE_LOCK_SUSPEND, "alarm_rtc");
```

紧接着通过 class_interface_register 函数来注册 Alarm 接口信息，主要包括设备的添加和移除操作，内容如下：

```
static struct class_interface rtc_alarm_interface = {
    .add_dev = &rtc_alarm_add_device,
    .remove_dev = &rtc_alarm_remove_device,
};
```

如果在此过程中出现错误，那么需要销毁已经注册的 SUSPEND lock，并且卸载 Alarm 驱动，代码如下：

```
wake_lock_destroy(&alarm_rtc_wake_lock);
wake_lock_destroy(&alarm_wake_lock);
platform_driver_unregister(&alarm_driver);
```

> **注意** wake lock 是一种锁机制，只要有用户持有该锁，系统就无法进入休眠状态，该锁可以被用户态程序和内核获得。这个锁可以是超时的或者是没有超时的，超时的锁会在时间过期以后自动解锁。如果没有锁或者超时了，内核就会启动休眠机制进入休眠状态，后面在讲电源管理时还会进一步讲解该机制。

2．alarm_late_init

当 Alarm 启动之后，我们需要读取当前的 RCT 和系统时间，由于需要确保在这个操作过程中不被中断，或者在中断之后能告诉其他进程该过程没有读取完成，不能被请求，因此这里需要通过 spin_lock_irqsave 和 spin_unlock_irqrestore 来对其执行锁定和解锁操作。实现代码如下：

```
static int __init alarm_late_init(void)
{
    unsigned long   flags;
    struct timespec system_time;

    spin_lock_irqsave(&alarm_slock, flags);

    getnstimeofday(&elapsed_rtc_delta);
    ktime_get_ts(&system_time);
    elapsed_rtc_delta = timespec_sub(elapsed_rtc_delta, system_time);

    spin_unlock_irqrestore(&alarm_slock, flags);

    ANDROID_ALARM_DPRINTF(ANDROID_ALARM_PRINT_INFO,
        "alarm_late_init: rtc to elapsed realtime delta %ld.%09ld\n",
        elapsed_rtc_delta.tv_sec, elapsed_rtc_delta.tv_nsec);
    return 0;
}
```

3．alarm_exit

当 Alarm 退出时，就需要通过 class_interface_unregister 函数来卸载在初始化时注册的 Alarm 接口，通过 wake_lock_destroy 函数来销毁 SUSPEND lock，以及通过 platform_driver_unregister 函数来卸载 Alarm 驱动。实现代码如下：

```
static void  __exit alarm_exit(void)
{
    class_interface_unregister(&rtc_alarm_interface);
    wake_lock_destroy(&alarm_rtc_wake_lock);
    wake_lock_destroy(&alarm_wake_lock);
    platform_driver_unregister(&alarm_driver);
}
```

4．添加和移除设备

接下来是 rtc_alarm_add_device 和 rtc_alarm_remove_device 函数的实现。添加设备时，首先将设备转换成 rtc_device 类型，然后，通过 misc_register 函数将自己注册成为一个 Misc 设备。其包括的主要特性如下面的代码所示：

```
static struct file_operations alarm_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = alarm_ioctl,
```

```
    .open = alarm_open,
    .release = alarm_release,
};

static struct miscdevice alarm_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "alarm",
    .fops = &alarm_fops,
};
```

其中 alarm_device 中的“.name”表示设备文件名称，而 alarm_fops 则定义了 Alarm 的常用操作，包括打开、释放和 I/O 控制。这里还需要通过 rtc_irq_register 函数注册一个 rtc_task，用来处理 Alarm 触发的方法，其定义如下：

```
static struct rtc_task alarm_rtc_task = {
    .func = alarm_triggered_func
};
```

其中“alarm_triggered_func”则是 Alarm 需要触发的方法。

> **注意** 如果在添加设备的过程中出现错误，我们需要对已经执行的操作进行释放、销毁和卸载。但是，移除一个设备时同样需要判断设备是否是 Alarm 设备，然后再执行卸载等操作。另外，在处理挂起操作时，我们首先就需要对设备进行锁定，然后根据 Alarm 的类型执行不同的操作，同时要保存时间。

alarm_open 和 alarm_release 的实现很简单。最后需要说明的是，对于 I/O 操作而言，主要需要实现：设置时间、设置 RTC、获取时间、设置 Alarm 等待等。

本小节主要对 Android 中最简单的设备驱动——Alarm 的实现流程进行了分析，大家应该可以自己绘制出一个流程图来了吧。对于 Alarm 的具体实现，大家可以参考源代码“drivers/rtc/alarm.c”中的实现方式。

2.3.2 Ashmem（匿名内存共享）

Ashmem 是 Android 的内存分配与共享机制，它在 dev 目录下对应的设备文件为/dev/ashmem，其实现的源文件为：

- include/linux/ashmem.h
- kernel/mm/ashmem.c

相比于 malloc 和 anonymous/named mmap 等传统的内存分配机制，其优势是通过内核驱动提供了辅助内核的内存回收算法机制（pin/unpin）。什么是 pin 和 unpin 呢？具体来讲，就是当你使用 Ashmem 分配了一块内存，但是其中某些部分却不会被使用时，那么就可以将这块内存 unpin 掉。unpin 后，内核可以将它对应的物理页面回收，以作他用。你也不用担心进程无法对

unpin 掉的内存进行再次访问，因为回收后的内存还可以再次被获得（通过缺页 handler），因为 unpin 操作并不会改变已经 mmap 的地址空间。下面就来分析 Ashmem 的内核驱动是如何完成这些功能。

首先，打开其头文件（ashmem.h），可以看到定义了以下一些宏和结构体：

```
//设备文件名称
#define ASHMEM_NAME_DEF        "dev/ashmem"
//从 ASHMEM_PIN 返回的值，判断是否需要清楚
#define ASHMEM_NOT_PURGED 0
#define ASHMEM_WAS_PURGED 1
//从 ASHMEM_GET_PIN_STATUS 返回的值，判断是 pin 还是 unpin
#define ASHMEM_IS_UNPINNED0
#define ASHMEM_IS_PINNED  1
struct ashmem_pin {
     __u32 offset;//在 Ashmem 区域的偏移量
     __u32 len;    //从偏移量开始的长度
};
```

另外一些宏用于设置 Ashmem 的名称和状态，以及 pin 和 unpin 等操作。接下来看一下 Ashmem 的具体实现，打开（ashmem.c）文件，首先大致预览一下它有哪些功能函数，如图 2-1 所示。

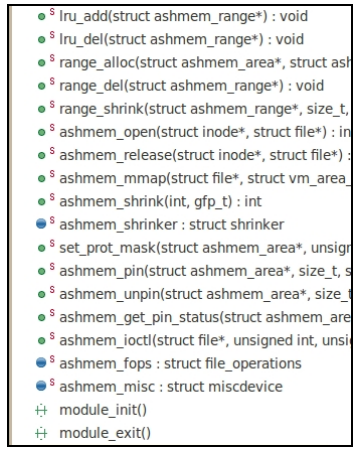

图 2-1　Ashmem 实现函数列表

可以看到 Ashmem 是通过以下代码来管理其初始化和退出操作的，我们分别需要实现其初始化函数 ashmem_init 和退出函数 ashmem_exit。

```
module_init(ashmem_init);
module_exit(ashmem_exit);
```

ashmem_init 的实现很简单，首先，定义一个结构体 ashmem_area 代表匿名共享内存区；然后，定义一个结构体 ashmem_range 代表 unpinned 页面的区域，代码如下：

```
struct ashmem_area {
    char name[ASHMEM_FULL_NAME_LEN];/* 用于/proc/pid/maps 中的一个标识名称 */
    struct list_head unpinned_list; /* 所有的匿名共享内存区列表 */
    struct file *file;              /* Ashmem 所支持的文件 */
    size_t size;                    /* 字节数 */
    unsigned long prot_mask;        /* vm_flags */
};
struct ashmem_range {
    struct list_head lru;           /* LRU 列表 */
    struct list_head unpinned;      /* unpinned 列表 */
    struct ashmem_area *asma;       /* ashmem_area 结构 */
    size_t pgstart;                 /* 开始页面 */
    size_t pgend;                   /* 结束页面 */
    unsigned int purged;   /* 是否需要清除(ASHMEM_NOT_PURGED 或者 ASHMEM_WAS_PURGED) */
};
```

ashmem_area 的生命周期为文件的 open()和 release()操作之间，而 ashmem_range 的生命周期则是从 unpin 到 pin，初始化时首先通过 kmem_cache_create 创建一个高速缓存 cache，所需参数如下：

- name　　用于/proc/slabinfo 文件中来识别这个 cache
- size　　在对应的 cache 中所创建的对象的长度
- align　　对象对齐尺寸
- flags　　SLAB 标志
- ctor　　构造函数

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。当针对 cache 的新的页面分配成功时运行 ctor 构造函数，然后采用 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（实现原理与 ashmem_area 一样）。创建完成之后，通过 misc_register 函数将 Ashmem 注册为 misc 设备。这里需要注意，我们对所创建的这些 cache 都需要进行回收，因此，再紧接着需调用 register_shrinker 注册回收函数 ashmem_shrinker。而从图 2-1 可以看出，ashmem_shrinker 实际上是一个结构体，真正的回收函数是在 ashmem_shrinker 中定义的 ashmem_shrink。到这里，初始化操作则完成了，实现代码如下：

```
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;
    }
```

```
    ashmem_range_cachep = kmem_cache_create("ashmem_range_cache",
                          sizeof(struct ashmem_range),
                          0, 0, NULL);
    if (unlikely(!ashmem_range_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;
    }
    /* 注册回收函数 */
    register_shrinker(&ashmem_shrinker);
    printk(KERN_INFO "ashmem: initialized\n");

    return 0;
}
```

当 Ashmem 退出时，又该执行什么操作呢？下面是 Ashmem 退出时需要执行的 ashmem_exit 函数的具体实现：

```
static void __exit ashmem_exit(void)
{
    int ret;
    /* 卸载回收函数 */
    unregister_shrinker(&ashmem_shrinker);
    /* 卸载 Ashmem 设备 */
    ret = misc_deregister(&ashmem_misc);
    if (unlikely(ret))
         printk(KERN_ERR "ashmem: failed to unregister misc device!\n");
    /* 卸载 cache */
    kmem_cache_destroy(ashmem_range_cachep);
    kmem_cache_destroy(ashmem_area_cachep);
    printk(KERN_INFO "ashmem: unloaded\n");
}
```

现在我们已经很清楚 Ashmem 的初始化和退出操作了，接下来我们将分析使用 Ashmem 对内存进行分配、释放和回收等机制的实现过程。在了解这些实现之前，我们先看看 Ashmem 分配内存的流程：

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。

由于 Ashmem 支持 pin/unpin 机制，所以还可以通过 ioctl 来 pin 和 unpin 某一段映射的空间。Ashmem 的作用就是分配空间，打开多少次/dev/ashmem 设备并 mmap，就会获得多少个不同的空间。

下面来分析如何通过打开设备文件来分配空间,并对空间进行回收。我们在初始化 Ashmem 时注册了 Ashmem 设备，其中包含的相关方法及其作用如下面的代码所示。

```
static struct file_operations ashmem_fops = {
    .owner = THIS_MODULE,
    .open = ashmem_open,                /* 打开 Ashmem */
    .release = ashmem_release,          /* 释放 Ashmem */
    .mmap = ashmem_mmap,                /* mmap 函数 */
    .unlocked_ioctl = ashmem_ioctl,     /* ioctl */
    .compat_ioctl = ashmem_ioctl,
};
static struct miscdevice ashmem_misc = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "ashmem",
    .fops = &ashmem_fops,
};
```

其中，ashmem_open 方法主要是对 unpinned 列表进行初始化，并将 Ashmem 分配的地址空间赋给 file 结构的 private_data，这就排除了进程间共享的可能性。ashmem_release 方法用于将指定的节点的空间从链表中删除并释放掉。需要指出的是,当使用 list_for_each_entry_safe(pos, n, head,member)函数时，需要调用者另外提供一个与 pos 同类型的指针 n，在 for 循环中暂存 pos 节点的下一个节点的地址，避免因 pos 节点被释放而造成断链。ashmem_release 函数的实现如下：

```
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);/* 删除 */
    mutex_unlock(&ashmem_mutex);
    if (asma->file)
        fput(asma->file);
    kmem_cache_free(ashmem_area_cachep, asma);

    return 0;
}
```

接下来就是将分配的空间映射到进程空间。在 ashmem_mmap 函数中需要指出的是，它借助了 Linux 内核的 shmem_file_setup（支撑文件）工具，使得我们不需要自己去实现这一复杂的过程。所以 ashmem_mmap 的整个实现过程很简单，大家可以参考它的源代码。最后，我们还将分析通过 ioctl 来 pin 和 unpin 某一段映射的空间的实现方式。ashmem_ioctl 函数的功能很多，它可以通过其参数 cmd 来处理不同的操作，包括设置（获取）名称和尺寸、pin/unpin 以及获取 pin 的一些状态。最终对 pin/unpin 的处理会通过下面这个函数来完成：

```
//pin/unpin 处理函数
static int ashmem_pin_unpin(struct ashmem_area *asma, unsigned long cmd,void __user *p)
//如果页面是 unpinned 和 ASHMEM_IS_PINNED,则返回 ASHMEM_IS_UNPINNED 状态
static int ashmem_get_pin_status(struct ashmem_area *asma, size_t pgstart,size_t pgend)
//unpin 指定区域页面，返回 0 表示成功
//调用者必须持有 ashmem_mutex
static int ashmem_unpin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
//pin ashmem 指定的区域
//返回是否曾被清除过(即 ASHMEM_WAS_PURGED 或者 ASHMEM_NOT_PURGED)
//调用者必须持有 ashmem_mutex
static int ashmem_pin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
```

最后需要说明：回收函数 cache_shrinker 同样也参考了 Linux 内核的 slab 分配算法用于页面回收的回调函数。具体实现如下：

```
static int ashmem_shrink(int nr_to_scan, gfp_t gfp_mask)
{
    struct ashmem_range *range, *next;
    if (nr_to_scan && !(gfp_mask & __GFP_FS))
        return -1;
    if (!nr_to_scan)
        return lru_count;
    mutex_lock(&ashmem_mutex);
    list_for_each_entry_safe(range, next, &ashmem_lru_list, lru) {
        struct inode *inode = range->asma->file->f_dentry->d_inode;
        loff_t start = range->pgstart * PAGE_SIZE;
        loff_t end = (range->pgend + 1) * PAGE_SIZE - 1;
        vmtruncate_range(inode, start, end);
        range->purged = ASHMEM_WAS_PURGED;
        lru_del(range);
        nr_to_scan -= range_size(range);
        if (nr_to_scan <= 0)
            break;
    }
    mutex_unlock(&ashmem_mutex);
    return lru_count;
}
```

cache_shrinker 同样先取得了 ashmem_mutex，通过 list_for_each_entry_safe 来确保其被安全释放。该方法会被 mm/vmscan.c :: shrink_slab 调用，其中参数 nr_to_scan 表示有多少个页面对象。如果该参数为 0，则表示查询所有的页面对象总数。而“gfp_mask”是一个配置，返回值为被回收之后剩下的页面数量；如果返回–1，则表示由于配置文件（gfp_mask）产生的问题，使得 mutex_lock 不能进行安全的死锁。

Ashmem 的源代码实现很简单，注释和代码总共不到 700 行。主要因为它借助了 Linux 内核已经有的工具，例如 shmem_file_setup（支撑文件）和 cache_shrinker（slab 分配算法用于页

面回收的回调函数）等，实现了高效的内存使用和管理，但是用户需进行额外的 ioctl 调用来设置名字和大小，以及执行 pin 和 unpin 操作等。

到这里，对 Ashmem 驱动的分析已经结束了。因为我们讲述的是实现的原理和机制，所以没有将代码全部贴出来，建议大家参考源代码进行理解。

2.3.3 Low Memory Killer（低内存管理）

对于 PC 来说，内存是至关重要。如果某个程序发生了内存泄漏，那么一般情况下系统就会将其进程 Kill 掉。Linux 中使用一种名称为 OOM（Out Of Memory，内存不足）的机制来完成这个任务，该机制会在系统内存不足的情况下，选择一个进程并将其 Kill 掉。Android 则使用了一个新的机制——Low Memory Killer 来完成同样的任务。下面首先来看看 Low Memory Killer 机制的原理以及它是如何选择将被 Kill 的进程的。

1．Low Memory Killer 的原理和机制

Low Memory Killer 在用户空间中指定了一组内存临界值，当其中的某个值与进程描述中的 oom_adj 值在同一范围时，该进程将被 Kill 掉。通常，在“/sys/module/lowmemorykiller / parameters/adj”中指定 oom_adj 的最小值，在“/sys/module/lowmemorykiller/parameters/minfree”中储存空闲页面的数量，所有的值都用一个逗号将其隔开且以升序排列。比如：把“0,8”写入到/sys/module/lowmemorykiller/parameters/adj 中，把“1024,4096”写入到/sys/module/lowmemory-killer/parameters/minfree 中，就表示当一个进程的空闲存储空间下降到 4096 个页面时，oom_adj 值为 8 或者更大的进程会被 Kill 掉。同理，当一个进程的空闲存储空间下降到 1024 个页面时，oom_adj 值为 0 或者更大的进程会被 Kill 掉。我们发现在 lowmemorykiller.c 中就指定了这样的值，如下所示：

```
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;
```

这就说明，当一个进程的空闲空间下降到 3×512 个页面时，oom_adj 值为 0 或者更大的进程会被 Kill 掉；当一个进程的空闲空间下降到 2×1024 个页面时，oom_adj 值为 10 或者更大的

进程会被 Kill 掉，依此类推。其实更简明的理解就是满足以下条件的进程将被优先 Kill 掉：

- task_struct->signal_struct->oom_adj 越大的越优先被 Kill。
- 占用物理内存最多的那个进程会被优先 Kill。

进程描述符中的 signal_struct->oom_adj 表示当内存短缺时进程被选择并 Kill 的优先级，取值范围是–17~15。如果是–17，则表示不会被选中，值越大越可能被选中。当某个进程被选中后，内核会发送 SIGKILL 信号将其 Kill 掉。

实际上，Low Memory Killer 驱动程序会认为被用于缓存的存储空间都要被释放，但是，如果很大一部分缓存存储空间处于被锁定的状态，那么这将是一个非常严重的错误，并且当正常的 oom killer 被触发之前，进程是不会被 Kill 掉的。

2．Low Memory Killer 的具体实现

在了解了 Low Memory Killer 的原理之后，我们再来看如何实现这个驱动。Low Memory Killer 驱动的实现位于 drivers/misc/lowmemorykiller.c。

该驱动的实现非常简单，其初始化与退出操作也是我们到目前为止见过的最简单的，代码如下：

```
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);
```

在初始化函数 lowmem_init 中通过 register_shrinker 注册了一个 shrinker 为 lowmem_shrinker；退出时又调用了函数 lowmem_exit，通过 unregister_shrinker 来卸载被注册的 lowmem_shrinker。其中 lowmem_shrinker 的定义如下：

```
static struct shrinker lowmem_shrinker = {
    .shrink = lowmem_shrink,
    .seeks = DEFAULT_SEEKS * 16
};
```

lowmem_shrink 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：

```
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 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;
}

read_lock(&tasklist_lock);
for_each_process(p) {
    if (p->oomkilladj < min_adj || !p->mm)
        continue;
    tasksize = get_mm_rss(p->mm);
    if (tasksize <= 0)
        continue;
    if (selected) {
        if (p->oomkilladj < selected->oomkilladj)
            continue;
        if (p->oomkilladj == selected->oomkilladj &&
           tasksize <= selected_tasksize)
            continue;
    }
```

```
            selected = p;
            selected_tasksize = tasksize;
            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;
    }
```

可以看出，其中多处用到了 global_page_state 函数。有很多人找不到这个函数，其实它被定义在了 linux/vmstat.h 中，其参数使用 zone_stat_item 枚举，被定义在 linux/mmzone.h 中，具体代码如下：

```
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, /* 避免编译错误*/
    NR_MLOCK = NR_ACTIVE_FILE,
#endif
    NR_ANON_PAGES,          /* 匿名映射页面*/
    NR_FILE_MAPPED,         /*映射页面*/
    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,      /* 使用临时缓冲区*/
#ifdef CONFIG_NUMA
```

```
        NUMA_HIT,               /* 在预定节点上分配*/
        NUMA_MISS,              /* 在非预定节点上分配*/
        NUMA_FOREIGN,
        NUMA_INTERLEAVE_HIT,
        NUMA_LOCAL,             /* 从本地页面分配*/
        NUMA_OTHER,             /* 从其他节点分配 */
    #endif
        NR_VM_ZONE_STAT_ITEMS };
```

再回过头来看 owmem_shrink 函数，首先确定我们所定义的 lowmem_adj 和 lowmem_minfree 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。因为我们需要通过比较 lowmem_minfree 中的空闲储存空间的值，以确定最小 min_adj 值（当满足其条件时，通过其数组索引来寻找 lowmem_adj 中对应元素的值）；之后检测 min_adj 的值是否是初始值“OOM_ADJUST_MAX + 1”，如果是，则表示没有满足条件的 min_adj 值，否则进入下一步；然后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（主要包括对 oomkilladj 和 task_struct 进行判断）；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。

关于 Low Memory Killer 的分析就到这里，在了解了其机制和原理之后，我们发现它的实现非常简单，与标准的 Linux OOM 机制类似，只是实现方式稍有不同。标准 Linux 的 OOM Killer 机制在 mm/oom_kill.c 中实现，且会被__alloc_pages_may_oom 调用（在分配内存时，即 mm/page_alloc.c 中）。oom_kill.c 最主要的一个函数是 out_of_memory，它选择一个 bad 进程 Kill，Kill 的方法同样是通过发送 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。

2.3.4 Logger（日志设备）

我们在开发 Android 应用的过程中可以很方便地使用 Log 信息来调试程序，这都归功于 Android 的 Logger 驱动为用户层提供的 Log 支持。无论是底层的源代码还是上层的应用，我们都可以使用 Logger 这个日志设备来进行调试。Logger 一共包括三个设备节点，它们分别是：

- /dev/log/main
- /dev/log/event
- /dev/log/radio

其驱动程序的实现源文件位于：

- include/linux/logger.h
- include/linux/logger.c

下面将对该驱动的实现进行分析，首先打开 logger.h 文件，我们可以看到如下所示的一个

结构体 logger_entry，它定义了每一条日志信息的属性。

```
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 目前没有什么实质作用，但是需要使用两个字节来占位；pid 表示生成该日志信息的进程的 pid；tid 表示生成该日志信息的进程的 tid；sec 表示生成该日志的时间，单位是秒；nsec 表示当生成该日志的时间不足 1 秒时，用纳秒来计算；msg 储存着该日志的有效信息，即我们前面说的长度为 len 的日志信息属于有效信息。

此外，还定义了代表不同设备事件的宏，分别对应于 Logger 的三个不同的设备节点，如下所示：

```
#define LOGGER_LOG_RADIO  "log_radio"       /* 无线相关消息 */
#define LOGGER_LOG_EVENTS "log_events"      /* 系统硬件事件 */
#define LOGGER_LOG_MAIN   "log_main"        /* 任何事件 */
```

接下来在 logger.c 中还定义了 logger_log 结构体，它定义每一个日志设备的相关信息。我们上面所说的 radio、events 和 main 都将使用 logger_log 结构体来表示，定义如下：

```
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，（为什么是环形缓冲区，后面将给大家解释）；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（buffer）的偏移量；head 是一个读取日志的新的 readers，表示从这里开始读取，同样指在环形缓冲区中（buffer）的偏移量；size 则代表该日志的大小，即环形缓冲区中（buffer）的大小。

根据上面这个日志设备结构 logger_log 可以得知，要读取日志还需要一个用于读取日志的 readers。下面我们来分析一下 readers 的定义，其结构体位于 logger.c 中的 logger_reader 结构体

中，代码如下：

```
struct logger_reader {
    struct logger_log *   log;
    struct list_head      list;
    size_t                r_off;
};
```

logger_reader 结构体的实现就很简单，其中 log 代表相关的日志设备，即当前将要读取数据的日志设备（logger_log）；list 用于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。

了解了这些数据结构之后，我们来分析一下该驱动是如何工作的，即该驱动的工作流程。

1．logger_init

首先还是来看其初始化方式，如下所示：

```
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);
```

当系统内核启动后，在 init 过程中就会调用 device_initcall 所指向的 logger_init 来初始化日志设备。我们可以看到，在 logger_init 函数中正好调用了 init_log 函数来初始化前面所提到的日志系统的三个设备节点。下面我们来看看 init_log 函数中究竟是如何初始化这些设备节点的。init_log 的实现如下：

```
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;
}
```

非常简单，通过调用 misc_register 来初始化每个日志设备的 miscdevice(logger_log->misc)。我们并没有看到具体的初始化日志设备的操作，那是因为这些工作都由 DEFINE_LOGGER_DEVICE 宏来完成了，DEFINE_LOGGER_DEVICE 的实现如下：

```
#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 需要我们传入三个参数，其作用就是使用参数 NAME 作为名称和使用 SIZE 作为尺寸来创建一个日志设备。这里需要注意：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。该宏的定义如下（源代码在 logger.h 文件中），表示日志的最大长度，同时还定义了 LOGGER_ENTRY_MAX_PAYLOAD 表示日志的最大有效长度。

```
#define LOGGER_ENTRY_MAX_LEN       (4*1024)
#define LOGGER_ENTRY_MAX_PAYLOAD
    (LOGGER_ENTRY_MAX_LEN - sizeof(struct logger_entry))
```

有了这些定义之后，现在要初始化一个日志设备就变得非常简单，以下代码初始化了三个不同的日志设备：

```
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，我们为设备指定了对应的 file_operations，其定义如下：

```
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,
};
```

其中主要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，等等。下面，我们将分别对这些函数的实现进行分析。

2．logger_open

该方法为打开日志设备文件的方法，具体实现如下：

```
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;
    //判断类型
    log = get_log_from_minor(MINOR(inode->i_rdev));
    if (!log)
        return -ENODEV;
    //只读模式
    if (file->f_mode & FMODE_READ) {
        struct logger_reader *reader;

        reader = kmalloc(sizeof(struct logger_reader), GFP_KERNEL);
        if (!reader)
            return -ENOMEM;
        //指定日志设备
        reader->log = log;
        INIT_LIST_HEAD(&reader->list);
        //指定 mutex
        mutex_lock(&log->mutex);
        //指定读取偏移量
        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 //读写模式
        file->private_data = log;
```

```
    return 0;
}
```

该函数首先调用 get_log_from_minor 函数来判断需要打开的日志设备的类型，判断方法非常简单，直接判断日志设备的 misc.minor 参数和 minor 参数即可，实现代码如下：

```
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;
}
```

再回过头来看 logger_open 函数，在取得了日志设备的类型之后，我们需要判断其读写模式。如果是只读模式，则将创建一个 logger_reader，然后对其所需的数据进行初始化（指定日志设备、mutex、读取偏移量 r_off），最后将该 logger_reader 保存到 file->private_data 中；如果是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，这样做就方便我们在以后的读写过程中直接通过 file->private_data 来取得 logger_reader 和 logger_log。

3．logger_release

在分析了打开操作之后，我们再来看一下释放操作，具体实现如下：

```
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;
}
```

首先判断其是否为只读模式，如果是只读模式，则直接通过 file->private_data 取得其对应的 logger_reader，然后删除其队列并释放即可。写操作则没有额外分配空间，所以不需要处理。

4．logger_read

接下来分析一下读数据的操作方法，其实现代码如下：

```
static ssize_t logger_read(struct file *file, char __user *buf,
                size_t count, loff_t *pos)
{
    //通过 file->private_data 获取 logger_reader 及其日志设备 logger_log
    struct logger_reader *reader = file->private_data;
    struct logger_log *log = reader->log;
```

```
    ssize_t ret;
    DEFINE_WAIT(wait);
start:
    while (1) {
        //添加进程到等待队列
        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;
    }
    //读取下一条日志
    ret = get_entry_len(log, reader->r_off);
    if (count < ret) {
        ret = -EINVAL;
        goto out;
    }
    //复制到用户空间
    ret = do_read_log_to_user(log, reader, buf, ret);
out:
    mutex_unlock(&log->mutex);
    return ret;
}
```

整体过程比较简单，但是这里需要注意：我们首先是通过 prepare_to_wait 函数将当前进程添加到等待队列 log->wq 之中，通过偏移量来判断当前日志的 buffer 是否为空。如果为空，则调度其他的进程运行，自己挂起；如果指定了非阻塞模式，则直接返回 EAGAIN。然后，通过 while 循环来重复该过程，直到 buffer 中有可供读取的日志为止。最后，通过 get_entry_len 函数读取下一条日志，并通过 do_read_log_to_user 将其复制到用户空间，读取完毕。

5．logger_aio_write

分析了读操作，下面登场的应该是写操作了。在这里，我们终于可以清楚地向大家解释之前的疑问——为什么缓冲区是环形的。在写入日志时，当其日志缓冲区 buffer 被写满之后，我们就不能再执行写入操作了吗？答案是否定的，正因为 buffer 是环形的，在写满之后，新写入的数据就会覆盖最初的数据，所以我们需要采取一定的措施来避免原来的数据被覆盖，以免造成数据丢失。写操作的具体实现如下：

```
ssize_t logger_aio_write(struct kiocb *iocb, const struct iovec *iov,
              unsigned long nr_segs, loff_t ppos)
{
    //取得日志设备 logger_log
    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();
    //初始化日志数据 logger_entry
    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);
    //修正偏移量，避免被覆盖
    fix_up_readers(log, sizeof(struct logger_entry) + header.len);
    //写入操作
    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);
        //从用户空间写入日志
        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;
}
```

与读操作一样，首先，需要取得日志设备 logger_log，这里我们是通过 file_get_log 函数来获取日志设备；然后，对要写入的日志执行初始化操作（包括进程的 pid、tid 和时间等）。因为我们的写操作支持同步、异步以及 scatter 等方式（非常灵活），而且在进行写操作时读操作可能并没有发生，这样就会被覆盖，所以通过在写操作之前执行 fix_up_readers 函数来修正其偏移量（r_off），然后才执行真正的写入操作。

fix_up_readers 函数真正能修正其偏移量而使其不被覆盖吗？下面我们先看看该函数的具体实现，如下所示：

```
static void fix_up_readers(struct logger_log *log, size_t len)
{
    //当前写偏移量
    size_t old = log->w_off;
    //写入长度为 len 的数据后的偏移量
    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);
}
```

大家可以看到，在执行 clock_interval 进行 new 复制时，将会覆盖 log->head，所以我们使用 get_next_entry 来查询下一个节点，并使其作为 head 节点。通常在执行查询时，我们使用的都是要被写入的整个数据的长度（len），因为是环形缓冲区，所以会出现覆盖数据的情况，因此这里传入的长度为最大长度（即要写入的数据长度）；然后遍历 reader 链表，如果 reader 在覆盖范围内，那么调整当前 reader 位置到下一个 log 数据区。因此从这里我们可以看出，fix_up_readers 函数只是起到一个缓解的作用，也不能最终解决数据覆盖问题，所以写入的数据如果不被及时读取，则会造成数据丢失。

6. logger_poll

该函数用来判断当前进程是否可以对日志设备进行操作，其具体实现代码如下：

```
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;
}
```

我们可以看出，POLLOUT 总是成立的，即进程总是可以进行写入操作；读操作则不一样了，如果只是以 FMODE_READ 模式打开日志设备的进程，那么就需要判断当前日志缓冲区是否为空，只有不为空才能读取日志。

7．logger_ioctl

该函数主要用于对一些命令进行操作，它可以支持以下命令操作：

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

它们分别对应于 logger.h 中所定义的下面这些宏：

```
#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很简单，大家可以参考 logger.c 中的 logger_ioctl 函数。以上就是我们对 Logger 驱动的分析，大家可以对应源码来阅读，这样会更容易理解。

2.3.5　Android PMEM

我们都知道 DSP 这类设备只能工作于大块连续的物理内存区域之上，因此 Android PMEM 便产生了，其主要作用就是向用户空间提供连续的物理内存区域。因此，Android PMEM 的主要功能包括以下两点：

- 让 GPU 或 VPU 缓冲区共享 CPU 核心。
- 用于 Android service 堆。

听上去有些玄妙，其实 Android PMEM 驱动就是为了实现应用层与内核层之间共享大块连续物理内存而研发的一种内存分配机制。PMEM 内存区域的内存不会受到标准 Linux 内存管理机制的限制，因此 DSP、GPU、VPU 等设备便可以在其分配的内存之上完美地工作。Android PMEM 驱动需要 Linux 内核的 misc 设备和 platform 驱动框架的支持，下面我们就来分析 Android

Linux公社（LinuxIDC.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

LinuxIDC.com提供包括Ubuntu，Fedora，SUSE技术，以及最新IT资讯等Linux专业类网站。

并被收录到Google 网页目录-计算机 > 软件 > 操作系统 > Linux 目录下。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

包括:

Ubuntu专题

Fedora专题

RedHat专题

SUSE专题

红旗Linux专题

Android专题