驱动学习整理

目录

[热插拔 6](#_Toc489876470)

[参考资料 10](#_Toc489876471)

[Sysfs以及设备模型 11](#_Toc489876472)

[内核空间与用户空间的映射关系 11](#_Toc489876473)

[设备模型底层容器 11](#_Toc489876474)

[struct kobj 11](#_Toc489876475)

[struct kobj\_type 12](#_Toc489876476)

[struct kset 12](#_Toc489876477)

[struct kobj\_attribute 12](#_Toc489876478)

[摘录自wowo 13](#_Toc489876479)

[设备模型上层容器 13](#_Toc489876480)

[总线类型bus\_type 13](#_Toc489876481)

[Struct device 15](#_Toc489876482)

[Struct device\_driver 19](#_Toc489876483)

[从面向对象的角度 20](#_Toc489876484)

[Kobject，kset，ktype结构图 21](#_Toc489876485)

[Kobj type 24](#_Toc489876486)

[参考资料 25](#_Toc489876487)

[最小子系统+ramdisk+initrd+系统启动过程 26](#_Toc489876488)

[ramdisk+initrd 26](#_Toc489876489)

[BIOS(Basic Input Output System) 26](#_Toc489876490)

[系统引导 27](#_Toc489876491)

[MBR 27](#_Toc489876492)

[grub 28](#_Toc489876493)

[关于boot.img解压缩以后就是kernel+ramdisk.gz两个文件 29](#_Toc489876494)

[参考资料 33](#_Toc489876495)

[Paltform 34](#_Toc489876496)

[Platform设备驱动 34](#_Toc489876497)

[Platform模块的软件架构 34](#_Toc489876498)

[Platform模块向其它模块提供的API汇整 35](#_Toc489876499)

[总结大多数驱动框架 35](#_Toc489876500)

[Platform led驱动 35](#_Toc489876501)

[Platform input驱动 35](#_Toc489876502)

[Platform i2c驱动 36](#_Toc489876503)

[Platform spi驱动 36](#_Toc489876504)

[参考资料 37](#_Toc489876505)

[中断子系统 38](#_Toc489876506)

[中断 38](#_Toc489876507)

[IRQ编号 39](#_Toc489876508)

[中断处理程序 39](#_Toc489876509)

[上半部与下半部 40](#_Toc489876510)

[在驱动程序中申请中断 41](#_Toc489876511)

[flags:与中断相关的标志 42](#_Toc489876512)

[request\_threaded\_irq的工作流程 43](#_Toc489876513)

[下半部的几种机制 45](#_Toc489876514)

[软中断(softirqs) 45](#_Toc489876515)

[Tasklet 46](#_Toc489876516)

[work queue 46](#_Toc489876517)

[内核定时器 46](#_Toc489876518)

[参考资料 47](#_Toc489876519)

[ARM概念梳理：Architecture, Core, CPU，SOC 48](#_Toc489876520)

[概念梳理 48](#_Toc489876521)

[ARM 64bit 48](#_Toc489876522)

[参考资料 49](#_Toc489876523)

[pmic与pmu的区别 49](#_Toc489876524)

[Linux中SD/MMC设备驱动流程 50](#_Toc489876525)

[SD（Secure Digital）与 MMC（Multimedia Card） 50](#_Toc489876526)

[SDIO（Secure Digital I/O） 50](#_Toc489876527)

[SD/SDIO/MMC/SPI之间的关系 50](#_Toc489876528)

[同一个控制器，支持SD/MMC/SDIO卡，如何区分是什么卡 51](#_Toc489876529)

[SD/SDIO 的传输模式 51](#_Toc489876530)

[参考资料 52](#_Toc489876531)

[内核通知链 53](#_Toc489876532)

[notifier chain 定义和接口 53](#_Toc489876533)

[1、Atomic notifier chains 53](#_Toc489876534)

[2、Blocking notifier chains 54](#_Toc489876535)

[3、Raw notifier chains 54](#_Toc489876536)

[4、SRCU notifier chains 55](#_Toc489876537)

[5、notifier\_call\_chain 55](#_Toc489876538)

[notifier chain 使用方法 56](#_Toc489876539)

[参考资料 56](#_Toc489876540)

[Socket通信机制 57](#_Toc489876541)

[socket套接字： 57](#_Toc489876542)

[套接字描述符： 57](#_Toc489876543)

[文件描述符和文件指针的区别： 58](#_Toc489876544)

[基本的SOCKET接口函数 59](#_Toc489876545)

[socket()函数 60](#_Toc489876546)

[bind()函数 60](#_Toc489876547)

[网络字节序与主机字节序 61](#_Toc489876548)

[listen()、connect()函数 62](#_Toc489876549)

[accept()函数 62](#_Toc489876550)

[read()、write()等函数 63](#_Toc489876551)

[close()函数 64](#_Toc489876552)

[Netlink机制及其关键技术 64](#_Toc489876553)

[3.1 Netlink机制 64](#_Toc489876554)

[3.2 Netlink优点 65](#_Toc489876555)

[建立Netlink会话过程如下： 66](#_Toc489876556)

[其他相关说明 67](#_Toc489876557)

[参考资料 76](#_Toc489876558)

[Linux工作队列 77](#_Toc489876559)

[1. 什么是workqueue 77](#_Toc489876560)

[2. 数据结构 77](#_Toc489876561)

[3. 创建工作 79](#_Toc489876562)

[4. 调度 81](#_Toc489876563)

[5. 示例 82](#_Toc489876564)

[6. workqueue的API 83](#_Toc489876565)

[共享队列的使用的顺序： 84](#_Toc489876566)

[参考资料 85](#_Toc489876567)

[同步/异步与阻塞/非阻塞的区别消息 86](#_Toc489876568)

[同步/异步与阻塞/非阻塞的区别. 86](#_Toc489876569)

[I/O 模型 88](#_Toc489876570)

[同步阻塞 I/O 88](#_Toc489876571)

[同步非阻塞 I/O 89](#_Toc489876572)

[异步阻塞 I/O 90](#_Toc489876573)

[异步非阻塞 I/O（AIO） 91](#_Toc489876574)

[异步通知范例 92](#_Toc489876575)

[应用层 93](#_Toc489876576)

[设备驱动 94](#_Toc489876577)

[内核API 95](#_Toc489876578)

[参考资料 96](#_Toc489876579)

[Linux内核的动态电压和电流控制接口 97](#_Toc489876580)

[2 Consumer的API 98](#_Toc489876581)

[3 电压的API 98](#_Toc489876582)

[4 电流的API 99](#_Toc489876583)

[5 校准器的驱动和系统配置 99](#_Toc489876584)

[6 应用 100](#_Toc489876585)

[1 动态的电压和频率调节 100](#_Toc489876586)

[2 挂起和恢复 101](#_Toc489876587)

[3 支持电源管理的驱动程序 101](#_Toc489876588)

[4 CPU空闲调节 102](#_Toc489876589)

[5 应用设计策略 102](#_Toc489876590)

[参考资料 102](#_Toc489876591)

[led子系统 103](#_Toc489876592)

[数据结构 103](#_Toc489876593)

[触发器的结构体 104](#_Toc489876594)

[平台设备相关的led数据结构 104](#_Toc489876595)

[平台设备相关的gpio led数据结构 105](#_Toc489876596)

[注册struct led\_classdev： 105](#_Toc489876597)

[注销struct led\_classdev： 107](#_Toc489876598)

[sysfs中的属性文件： 107](#_Toc489876599)

[led\_classdev全局链表： 107](#_Toc489876600)

[参考资料 109](#_Toc489876601)

[固件子系统 110](#_Toc489876602)

[参考资料 117](#_Toc489876603)

[/sys目录下各个子目录的具体说明 118](#_Toc489876604)

[参考资料 119](#_Toc489876605)

[U-boot如何向kernel传递参数 + kernel如何读取参数 120](#_Toc489876606)

[u-boot怎么传入一个自定义参数给内核？？然后内核又是怎样解析？ 128](#_Toc489876607)

[参考资料 129](#_Toc489876608)

[linux中mdelay()与msleep()的区别 130](#_Toc489876609)

[参考资料 130](#_Toc489876610)

[用户空间内核空间ipc总结 131](#_Toc489876611)

[getsockopt/setsockopt： 131](#_Toc489876612)

[用户态函数： 131](#_Toc489876613)

[内核态函数： 131](#_Toc489876614)

[mmap /shm： 132](#_Toc489876615)

[Mmap的优点 133](#_Toc489876616)

[netlink/socket： 134](#_Toc489876617)

[与BSD的Routing socket的关系 135](#_Toc489876618)

[proc/seq/sys文件系统： 135](#_Toc489876619)

[copy\_from\_user/copy\_to\_user ： 135](#_Toc489876620)

[参考资料 135](#_Toc489876621)

[电源管理 137](#_Toc489876622)

[手机关机的做法 137](#_Toc489876623)

[▆ Hibernate（冬眠）和Sleep（睡眠） 138](#_Toc489876624)

[▆ Suspend 138](#_Toc489876625)

[▆ Standby 138](#_Toc489876626)

[▆ Wakeup 138](#_Toc489876627)

[参考资料 139](#_Toc489876628)

[linux内核中的GPIO系统 140](#_Toc489876629)

[自旋锁和互斥锁区别 140](#_Toc489876630)

[自旋锁(Spin lock) 140](#_Toc489876631)

[参考资料 142](#_Toc489876632)

[kmalloc、vmalloc、malloc的区别 142](#_Toc489876633)

[kmalloc函数详解 142](#_Toc489876634)

[kzalloc与kmalloc区别 143](#_Toc489876635)

[参考资料 143](#_Toc489876636)

# 热插拔

一个完整的**热插拔系统包括热插拔系统的硬件，支持热插拔的软件和操作系统**，**支持热插拔的设备驱动程序和支持热插拔的用户接口**。

**硬件支持：**插入->先数据，后电源；拔出->先电源，后数据(具体硬件机理不详)

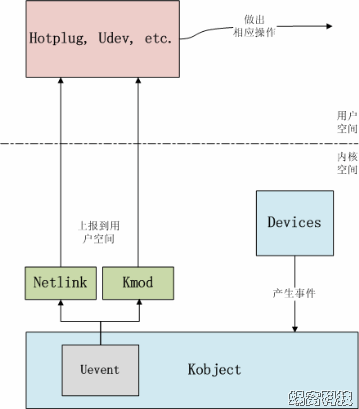
**软件支持：**uevent, user space event. 内核与用户空间的一种通信机制，netlink通信机制

**Uevent**是**Kobject**的一部分，用于在Kobject状态发生改变时，例如增加、移除等，通知用户空间程序。用户空间程序收到这样的事件后，会做相应的处理。

该机制通常是用来支持热拔插设备的，例如U盘插入后，USB相关的驱动软件会动态创建用于表示该U盘的device结构（相应的也包括其中的kobject），并告知用户空间程序，为该U盘动态的创建/dev/目录下的设备节点，更进一步，可以通知其它的应用程序，将该U盘设备mount到系统中，从而动态的支持该设备。

**Uevent的机制：**设备模型中任何设备有事件需要上报时，会触发Uevent提供的接口。Uevent模块准备好上报事件的格式后，可以通过两个途径把事件上报到用户空间：一种是通过kmod模块，直接调用用户空间的可执行文件；另一种是通过netlink通信机制，将事件从内核空间传递给用户空间。

如图来自蜗居科技：http://www.wowotech.net/linux\_kenrel/uevent.html



##### 内部动作

通过kobject.h，uevent模块提供了如下的API（这些API的实现是在"lib/kobject\_uevent.c”文件中）：

1: /\* include/linux/kobject.h, line 206 \*/

2: int kobject\_uevent(struct kobject \*kobj, enum kobject\_action action);

3: int kobject\_uevent\_env(struct kobject \*kobj, enum kobject\_action action, char \*envp[]);

6: \_\_printf(2, 3)

7: int add\_uevent\_var(struct kobj\_uevent\_env \*env, const char \*format, ...);

9: int kobject\_action\_type(const char \*buf, size\_t count, enum kobject\_action \*type);

kobject\_uevent\_env，以envp为环境变量，上报一个指定action的uevent。环境变量的作用是为执行用户空间程序指定运行环境。具体动作如下：

* 查找kobj本身或者其parent是否从属于某个kset，如果不是，则报错返回（注2：由此可以说明，如果一个kobject没有加入kset，是不允许上报uevent的）
* 查看kobj->uevent\_suppress是否设置，如果设置，则忽略所有的uevent上报并返回（注3：由此可知，可以通过Kobject的uevent\_suppress标志，管控Kobject的uevent的上报）
* 如果所属的kset有uevent\_ops->filter函数，则调用该函数，过滤此次上报（注4：这佐证了3.2小节有关filter接口的说明，kset可以通过filter接口过滤不希望上报的event，从而达到整体的管理效果）
* 判断所属的kset是否有合法的名称（称作subsystem，和前期的内核版本有区别），否则不允许上报uevent
* 分配一个用于此次上报的、存储环境变量的buffer（结果保存在env指针中），并获得该Kobject在sysfs中路径信息（用户空间软件需要依据该路径信息在sysfs中访问它）
* 调用add\_uevent\_var接口（下面会介绍），将Action、路径信息、subsystem等信息，添加到env指针中
* 如果传入的envp不空，则解析传入的环境变量中，同样调用add\_uevent\_var接口，添加到env指针中
* 如果所属的kset存在uevent\_ops->uevent接口，调用该接口，添加kset统一的环境变量到env指针
* 根据ACTION的类型，设置kobj->state\_add\_uevent\_sent和kobj->state\_remove\_uevent\_sent变量，以记录正确的状态
* 调用add\_uevent\_var接口，添加格式为"SEQNUM=%llu”的序列号
* **如果定义了"CONFIG\_NET”，则使用netlink发送该uevent**
* **以uevent\_helper、subsystem以及添加了标准环境变量（HOME=/，PATH=/sbin:/bin:/usr/sbin:/usr/bin）的env指针为参数，调用kmod模块提供的call\_usermodehelper函数，上报uevent。**  
  其中uevent\_helper的内容是由内核配置项CONFIG\_UEVENT\_HELPER\_PATH(位于./drivers/base/Kconfig)决定的(可参考lib/kobject\_uevent.c, line 32)，该配置项指定了一个用户空间程序（或者脚本），用于解析上报的uevent，例如"/sbin/hotplug”。   
  call\_usermodehelper的作用，就是fork一个进程，以uevent为参数，执行uevent\_helper。

kobject\_uevent，和kobject\_uevent\_env功能一样，只是没有指定任何的环境变量。

add\_uevent\_var，以格式化字符的形式（类似printf、printk等），将环境变量copy到env指针中。



kobject\_action\_type，将enum kobject\_action类型的Action，转换为字符串。

上面所说的过程可以去除driver部分，简化如下：  
① 外设插入；  
② 总线发现（中断？ usb有中断hub\_irq，pci的中断没找到）新设备，调用device\_add，添加新设备到设备管理系统；  
③ device\_add中调用kobject\_uevent(, KOBJ\_ADD)，向userspace广播新设备加入event通知；这里发出通知的方式，就是netlink；  
④ 用户空间运行的daemon(udevd)收到event事件广播；udevd (udev 后台程序)  
⑤ udevd根据消息和环境变量，查询/sys的变化，按照规则(/etc/udev/rules.d/\*)，在/dev目录下自动创建设备节点；

## 参考资料

http://www.wowotech.net/linux\_kenrel/uevent.html 主要讲述的是头文件热插拔事件从内核空间到用户空间

http://blog.csdn.net/bingqingsuimeng/article/details/7924300 有源代码的讲解，比较清晰

http://www.cnblogs.com/image-eye/archive/2011/08/19/2145858.html 内核启动过程的驱动加载

http://blog.chinaunix.net/uid-25721104-id-3023525.html 讨论的关于热插拔

http://blog.csdn.net/zirconsdu/article/details/8792184 图解很清晰

http://blog.chinaunix.net/uid-27666459-id-5747129.html讲解call\_usermodehelper

ps：[**kobject\_uevent\_env**](http://lxr.free-electrons.com/ident?i=kobject_uevent_env)**里面有定义是通过**

**#ifdef CONFIG\_UEVENT\_HELPER**

**#if defined(CONFIG\_NET)**

**两种机制的哪一种**

# Sysfs以及设备模型

Sysfs被加载在 /sys/目录下,它的子目录包括

1）Block:在系统中发现的每个块设备在该目录下对应一个子目录。每个子目录中又包含一些属性文件,它们描述了这个块设备的各方面属性,如:设备大小。(loop块设备是使用文件来模拟的)

2）Bus:在内核中注册的每条总线在该目录下对应一个子目录,如: ide pci scsi usbpcmcia 其中每个总线目录内又包含两个子目录:devices和drivers ,devices目录包含了在整个系统中发现的属于该总线类型的设备,drivers目录包含了注册到该总线的所有驱动。

3）Class:将设备按照功能进行的分类,如/sys/class/net目录下包含了所有网络接口。

4）Devices:包含系统所有的设备。

5）Kernel:内核中的配置参数

6）Module:系统中所有模块的信息

7）Firmware:系统中的固件

8）Fs:描述系统中的文件系统

9）Power:系统中电源选项

## 内核空间与用户空间的映射关系

内核空间(internel) ——->用户空间(externel)  
内核对象(kernel objects) ——->目录(directories)  
对象属性(object attributes) ——->普通文件(regular files)  
对象关系(object relationships) ——->符号链接(symbolic links)

## 设备模型底层容器

### struct kobj

struct kobject {

const char \*name; //kobject的名称

struct list\_head entry; //kobject结构链表

struct kobject \*parent; //父kobject结构体

struct kset \*kset; //kset集合

struct kobj\_type \*ktype; //kobject的类型描述符

struct sysfs\_dirent \*sd; //sysfs文件目录

struct kref kref; //kobject引用计数

#ifdef CONFIG\_DEBUG\_KOBJECT\_RELEASE

struct [delayed\_work](http://lxr.free-electrons.com/ident?i=delayed_work) [release](http://lxr.free-electrons.com/ident?i=release);

#endif

unsigned int state\_initialized:1; //kobject是否初始化

unsigned int state\_in\_sysfs:1; //是否已经加入sysfs

unsigned int state\_add\_uevent\_sent:1;

unsigned int state\_remove\_uevent\_sent:1;

unsigned int uevent\_suppress:1;

};

### struct kobj\_type

struct kobj\_type {

void (\*release)(struct kobject \*kobj); //释放函数（驱动编写时提供），此函数会被kobject\_put函数调用

struct sysfs\_ops \*sysfs\_ops; //属性文件的操作函数（只有读和写操作）

struct attribute \*\*default\_attrs; //属性数组

const struct kobj\_ns\_type\_operations \*(\*child\_ns\_type)(struct kobject \*kobj);

const void \*(\*namespace)(struct kobject \*kobj);

};

### struct kset

struct kset {

struct list\_head list; //这个链表存放这个kset关联的所有kobject

spinlock\_t list\_lock; //维护此链表的锁

struct kobject kobj; //内嵌的kobject。这样kset本身也是一个kobject也被表现为一个目录

struct kset\_uevent\_ops \*uevent\_ops; //支持热插拔事件的函数集

};

### struct kobj\_attribute

struct kobj\_attribute {

struct attribute attr;

ssize\_t (\*show)(struct kobject \*kobj, struct kobj\_attribute \*attr,

char \*buf);

ssize\_t (\*store)(struct kobject \*kobj, struct kobj\_attribute \*attr,

const char \*buf, size\_t count);

};

### 摘录自wowo

* 每个设备（struct device）都会保存该设备的驱动（struct device\_driver）指针，以及该设备所在总线（struct bus\_type）的指针（具体参考“[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”）
* 设备驱动中有一个名称为“shutdown”的回调函数，用于在device\_shutdown时，关闭该设备（具体参考“[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”）
* 总线中也有一个名称为“shutdown”的回调函数，用于在device\_shutdown时，关闭该设备（具体参考“[Linux设备模型(6)\_Bus](http://www.wowotech.net/linux_kenrel/bus.html)”）
* 系统的所有设备，都存在于“/sys/devices/”目录下，而该目录由名称为“devices\_kset”的kset表示。而由“[Linux设备模型(2)\_Kobject](http://www.wowotech.net/linux_kenrel/kobject.html)”的描述可知，kset中会使用一个链表保存其下所有的kobject（也即“/sys/devices/”目录下的所有设备）。最终的结果就是，以“devices\_kset”为root节点，将内核中所有的设备（以相应的kobject为代表），组织成一个树状结构

## 设备模型上层容器

### 总线类型bus\_type

struct bus\_type {

const char \*name; /\* 总线类型名 \*/

struct bus\_attribute \*bus\_attrs; /\* 总线的属性 \*/

struct device\_attribute \*dev\_attrs; /\* 设备属性,为每个加入总线的设备建立属性链表 \*/

struct driver\_attribute \*drv\_attrs; /\* 驱动属性,为每个加入总线的驱动建立属性链表 \*/

/\* 驱动与设备匹配函数:当一个新设备或者驱动被添加到这个总线时，这个方法会被调用一次或多次，若指定的驱动程序能够处理指定的设备，则返回非零值。必须在总线层使用这个函数, 因为那里存在正确的逻辑，核心内核不知道如何为每个总线类型匹配设备和驱动程序 \*/

int (\*match)(struct device \*dev, struct device\_driver \*drv);

/\*在为用户空间产生热插拔事件之前，这个方法允许总线添加环境变量（参数和 kset 的uevent方法相同）\*/

int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

int (\*probe)(struct device \*dev); /\* \*/

int (\*remove)(struct device \*dev); /\* 设备移除调用操作 \*/

void (\*shutdown)(struct device \*dev);

int (\*suspend)(struct device \*dev, pm\_message\_t state);

int (\*resume)(struct device \*dev);

const struct dev\_pm\_ops \*pm;

struct subsys\_private \*p; /\* 一个很重要的域，包含了device链表和drivers链表 \*/

};

/\*\*

63 \* struct bus\_type - The bus type of the device

64 \*

65 \* @name: The name of the bus.

66 \* @dev\_name: Used for subsystems to enumerate devices like ("foo%u", dev->id).

67 \* @dev\_root: Default device to use as the parent.

68 \* @dev\_attrs: Default attributes of the devices on the bus.

69 \* @bus\_groups: Default attributes of the bus.

70 \* @dev\_groups: Default attributes of the devices on the bus.

71 \* @drv\_groups: Default attributes of the device drivers on the bus.

72 \* @match: Called, perhaps multiple times, whenever a new device or driver

73 \* is added for this bus. It should return a positive value if the

74 \* given device can be handled by the given driver and zero

75 \* otherwise. It may also return error code if determining that

76 \* the driver supports the device is not possible. In case of

77 \* -EPROBE\_DEFER it will queue the device for deferred probing.

78 \* @uevent: Called when a device is added, removed, or a few other things

79 \* that generate uevents to add the environment variables.

80 \* @probe: Called when a new device or driver add to this bus, and callback

81 \* the specific driver's probe to initial the matched device.

82 \* @remove: Called when a device removed from this bus.

83 \* @shutdown: Called at shut-down time to quiesce the device.

84 \*

85 \* @online: Called to put the device back online (after offlining it).

86 \* @offline: Called to put the device offline for hot-removal. May fail.

87 \*

88 \* @suspend: Called when a device on this bus wants to go to sleep mode.

89 \* @resume: Called to bring a device on this bus out of sleep mode.

90 \* @pm: Power management operations of this bus, callback the specific

91 \* device driver's pm-ops.

92 \* @iommu\_ops: IOMMU specific operations for this bus, used to attach IOMMU

93 \* driver implementations to a bus and allow the driver to do

94 \* bus-specific setup

95 \* @p: The private data of the driver core, only the driver core can

96 \* touch this.

97 \* @lock\_key: Lock class key for use by the lock validator

98 \*

99 \* A bus is a channel between the processor and one or more devices. For the

100 \* purposes of the device model, all devices are connected via a bus, even if

101 \* it is an internal, virtual, "platform" bus. Buses can plug into each other.

102 \* A USB controller is usually a PCI device, for example. The device model

103 \* represents the actual connections between buses and the devices they control.

104 \* A bus is represented by the bus\_type structure. It contains the name, the

105 \* default attributes, the bus' methods, PM operations, and the driver core's

106 \* private data.

107 \*/

108 struct bus\_type {

109 const char \*name;

110 const char \*dev\_name;

111 struct device \*dev\_root;

112 struct device\_attribute \*dev\_attrs; /\* use dev\_groups instead \*/

113 const struct attribute\_group \*\*bus\_groups;

114 const struct attribute\_group \*\*dev\_groups;

115 const struct attribute\_group \*\*drv\_groups;

116

117 int (\*match)(struct device \*dev, struct device\_driver \*drv);

118 int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

119 int (\*probe)(struct device \*dev);

120 int (\*remove)(struct device \*dev);

121 void (\*shutdown)(struct device \*dev);

122

123 int (\*online)(struct device \*dev);

124 int (\*offline)(struct device \*dev);

125

126 int (\*suspend)(struct device \*dev, pm\_message\_t state);

127 int (\*resume)(struct device \*dev);

128

129 const struct dev\_pm\_ops \*pm;

130

131 const struct iommu\_ops \*iommu\_ops;

132

133 struct subsys\_private \*p;

134 struct lock\_class\_key lock\_key;

135 };

### Struct device

709 /\*\*

710 \* struct device - The basic device structure

711 \* @parent: The device's "parent" device, the device to which it is attached.

712 \* In most cases, a parent device is some sort of bus or host

713 \* controller. If parent is NULL, the device, is a top-level device,

714 \* which is not usually what you want.

715 \* @p: Holds the private data of the driver core portions of the device.

716 \* See the comment of the struct device\_private for detail.

717 \* @kobj: A top-level, abstract class from which other classes are derived.

718 \* @init\_name: Initial name of the device.

719 \* @type: The type of device.

720 \* This identifies the device type and carries type-specific

721 \* information.

722 \* @mutex: Mutex to synchronize calls to its driver.

723 \* @bus: Type of bus device is on.

724 \* @driver: Which driver has allocated this

725 \* @platform\_data: Platform data specific to the device.

726 \* Example: For devices on custom boards, as typical of embedded

727 \* and SOC based hardware, Linux often uses platform\_data to point

728 \* to board-specific structures describing devices and how they

729 \* are wired. That can include what ports are available, chip

730 \* variants, which GPIO pins act in what additional roles, and so

731 \* on. This shrinks the "Board Support Packages" (BSPs) and

732 \* minimizes board-specific #ifdefs in drivers.

733 \* @driver\_data: Private pointer for driver specific info.

734 \* @power: For device power management.

735 \* See Documentation/power/devices.txt for details.

736 \* @pm\_domain: Provide callbacks that are executed during system suspend,

737 \* hibernation, system resume and during runtime PM transitions

738 \* along with subsystem-level and driver-level callbacks.

739 \* @pins: For device pin management.

740 \* See Documentation/pinctrl.txt for details.

741 \* @msi\_list: Hosts MSI descriptors

742 \* @msi\_domain: The generic MSI domain this device is using.

743 \* @numa\_node: NUMA node this device is close to.

744 \* @dma\_mask: Dma mask (if dma'ble device).

745 \* @coherent\_dma\_mask: Like dma\_mask, but for alloc\_coherent mapping as not all

746 \* hardware supports 64-bit addresses for consistent allocations

747 \* such descriptors.

748 \* @dma\_pfn\_offset: offset of DMA memory range relatively of RAM

749 \* @dma\_parms: A low level driver may set these to teach IOMMU code about

750 \* segment limitations.

751 \* @dma\_pools: Dma pools (if dma'ble device).

752 \* @dma\_mem: Internal for coherent mem override.

753 \* @cma\_area: Contiguous memory area for dma allocations

754 \* @archdata: For arch-specific additions.

755 \* @of\_node: Associated device tree node.

756 \* @fwnode: Associated device node supplied by platform firmware.

757 \* @devt: For creating the sysfs "dev".

758 \* @id: device instance

759 \* @devres\_lock: Spinlock to protect the resource of the device.

760 \* @devres\_head: The resources list of the device.

761 \* @knode\_class: The node used to add the device to the class list.

762 \* @class: The class of the device.

763 \* @groups: Optional attribute groups.

764 \* @release: Callback to free the device after all references have

765 \* gone away. This should be set by the allocator of the

766 \* device (i.e. the bus driver that discovered the device).

767 \* @iommu\_group: IOMMU group the device belongs to.

768 \*

769 \* @offline\_disabled: If set, the device is permanently online.

770 \* @offline: Set after successful invocation of bus type's .offline().

771 \*

772 \* At the lowest level, every device in a Linux system is represented by an

773 \* instance of struct device. The device structure contains the information

774 \* that the device model core needs to model the system. Most subsystems,

775 \* however, track additional information about the devices they host. As a

776 \* result, it is rare for devices to be represented by bare device structures;

777 \* instead, that structure, like kobject structures, is usually embedded within

778 \* a higher-level representation of the device.

779 \*/

780 struct device {

781 struct device \*parent;

782

783 struct device\_private \*p;

784

785 struct kobject kobj;

786 const char \*init\_name; /\* initial name of the device \*/

787 const struct device\_type \*type;

788

789 struct mutex mutex; /\* mutex to synchronize calls to

790 \* its driver.

791 \*/

792

793 struct bus\_type \*bus; /\* type of bus device is on \*/

794 struct device\_driver \*driver; /\* which driver has allocated this

795 device \*/

796 void \*platform\_data; /\* Platform specific data, device

797 core doesn't touch it \*/

798 void \*driver\_data; /\* Driver data, set and get with

799 dev\_set/get\_drvdata \*/

800 struct dev\_pm\_info power;

801 struct dev\_pm\_domain \*pm\_domain;

802

803 #ifdef CONFIG\_GENERIC\_MSI\_IRQ\_DOMAIN

804 struct irq\_domain \*msi\_domain;

805 #endif

806 #ifdef CONFIG\_PINCTRL

807 struct dev\_pin\_info \*pins;

808 #endif

809 #ifdef CONFIG\_GENERIC\_MSI\_IRQ

810 struct list\_head msi\_list;

811 #endif

812

813 #ifdef CONFIG\_NUMA

814 int numa\_node; /\* NUMA node this device is close to \*/

815 #endif

816 u64 \*dma\_mask; /\* dma mask (if dma'able device) \*/

817 u64 coherent\_dma\_mask;/\* Like dma\_mask, but for

818 alloc\_coherent mappings as

819 not all hardware supports

820 64 bit addresses for consistent

821 allocations such descriptors. \*/

822 unsigned long dma\_pfn\_offset;

823

824 struct device\_dma\_parameters \*dma\_parms;

825

826 struct list\_head dma\_pools; /\* dma pools (if dma'ble) \*/

827

828 struct dma\_coherent\_mem \*dma\_mem; /\* internal for coherent mem

829 override \*/

830 #ifdef CONFIG\_DMA\_CMA

831 struct cma \*cma\_area; /\* contiguous memory area for dma

832 allocations \*/

833 #endif

834 /\* arch specific additions \*/

835 struct dev\_archdata archdata;

836

837 struct device\_node \*of\_node; /\* associated device tree node \*/

838 struct fwnode\_handle \*fwnode; /\* firmware device node \*/

839

840 dev\_t devt; /\* dev\_t, creates the sysfs "dev" \*/

841 u32 id; /\* device instance \*/

842

843 spinlock\_t devres\_lock;

844 struct list\_head devres\_head;

845

846 struct klist\_node knode\_class;

847 struct class \*class;

848 const struct attribute\_group \*\*groups; /\* optional groups \*/

849

850 void (\*release)(struct device \*dev);

851 struct iommu\_group \*iommu\_group;

852

853 bool offline\_disabled:1;

854 bool offline:1;

855 };

### Struct device\_driver

/\*\*

231 \* struct device\_driver - The basic device driver structure

232 \* @name: Name of the device driver.

233 \* @bus: The bus which the device of this driver belongs to.

234 \* @owner: The module owner.

235 \* @mod\_name: Used for built-in modules.

236 \* @suppress\_bind\_attrs: Disables bind/unbind via sysfs.

237 \* @probe\_type: Type of the probe (synchronous or asynchronous) to use.

238 \* @of\_match\_table: The open firmware table.

239 \* @acpi\_match\_table: The ACPI match table.

240 \* @probe: Called to query the existence of a specific device,

241 \* whether this driver can work with it, and bind the driver

242 \* to a specific device.

243 \* @remove: Called when the device is removed from the system to

244 \* unbind a device from this driver.

245 \* @shutdown: Called at shut-down time to quiesce the device.

246 \* @suspend: Called to put the device to sleep mode. Usually to a

247 \* low power state.

248 \* @resume: Called to bring a device from sleep mode.

249 \* @groups: Default attributes that get created by the driver core

250 \* automatically.

251 \* @pm: Power management operations of the device which matched

252 \* this driver.

253 \* @p: Driver core's private data, no one other than the driver

254 \* core can touch this.

255 \*

256 \* The device driver-model tracks all of the drivers known to the system.

257 \* The main reason for this tracking is to enable the driver core to match

258 \* up drivers with new devices. Once drivers are known objects within the

259 \* system, however, a number of other things become possible. Device drivers

260 \* can export information and configuration variables that are independent

261 \* of any specific device.

262 \*/

263 struct device\_driver {

264 const char \*name;

265 struct bus\_type \*bus;

266

267 struct module \*owner;

268 const char \*mod\_name; /\* used for built-in modules \*/

269

270 bool suppress\_bind\_attrs; /\* disables bind/unbind via sysfs \*/

271 enum probe\_type probe\_type;

272

273 const struct of\_device\_id \*of\_match\_table;

274 const struct acpi\_device\_id \*acpi\_match\_table;

275

276 int (\*probe) (struct device \*dev);

277 int (\*remove) (struct device \*dev);

278 void (\*shutdown) (struct device \*dev);

279 int (\*suspend) (struct device \*dev, pm\_message\_t state);

280 int (\*resume) (struct device \*dev);

281 const struct attribute\_group \*\*groups;

282

283 const struct dev\_pm\_ops \*pm;

284

285 struct driver\_private \*p;

286 };

## 从面向对象的角度

- struct kobj（及其相关结构如kset, ktype等）属于最抽象的基类，代码最简洁，最不具体； 通常我们并不关注 kobject 本身，而应该关注那些嵌入了 kobject 的那些结构体。；ktype 是嵌入了 kobject 的对象的类型。每个嵌入了 kobject 的对象都需要一个相应的 ktype。ktype 用来控制当 kobject 创建和销毁时所发生的操作。 这些 kobject 可以是同样的 ktype，也可以分别属于不同的 ktype。kset 是 kobject 集合的基本容器类型。kset 是 kobject 的一组集合。kset 也包含它们自己的 kobject，但是你可以放心的忽略这些 kobjects，因为 kset 的核心代码会自动处理这些 kobject。

- struct device（及其相关结构如device\_driver，device\_type等）是对kobj的封装，是第一层派生类；

- 再上层的结构（如platform\_device等），是在struct device的基础上再封装一次，是第二层派生类。

因此，例如我们创建了一个struct platform\_device的实例，使用完毕后要释放它。那么这个过程按道理应该是：

- 系统内部先调用platform\_device的remove函数，它只处理自己层特有的变量；

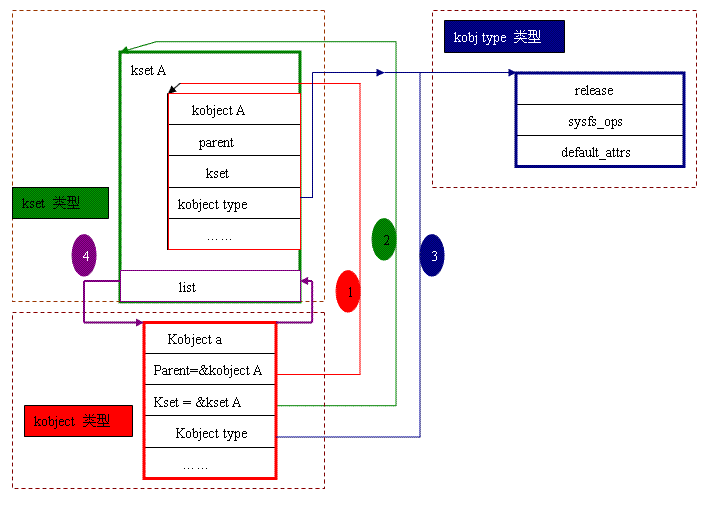
- 完毕后，系统调用第一层派生类struct device的release函数，处理了自己这一层的特有变量；

- 最后，kobject的release函数，将整个空间释放掉。

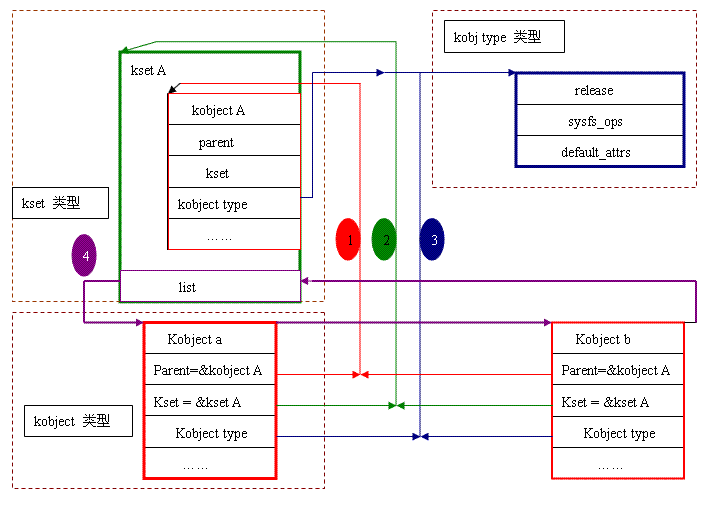
整个过程应该会跟C++析构过程比较类似，上述的“系统内部”也应该类似于C++编译器自动生成的代码，因为C++中析构函数的逆向调用是自动进行的，并没有在派生类的析构函数中显示调用。类似地，在此处上层的release中也不会显式调用下层release，都是由系统内部完成的。

## Kobject，kset，ktype结构图

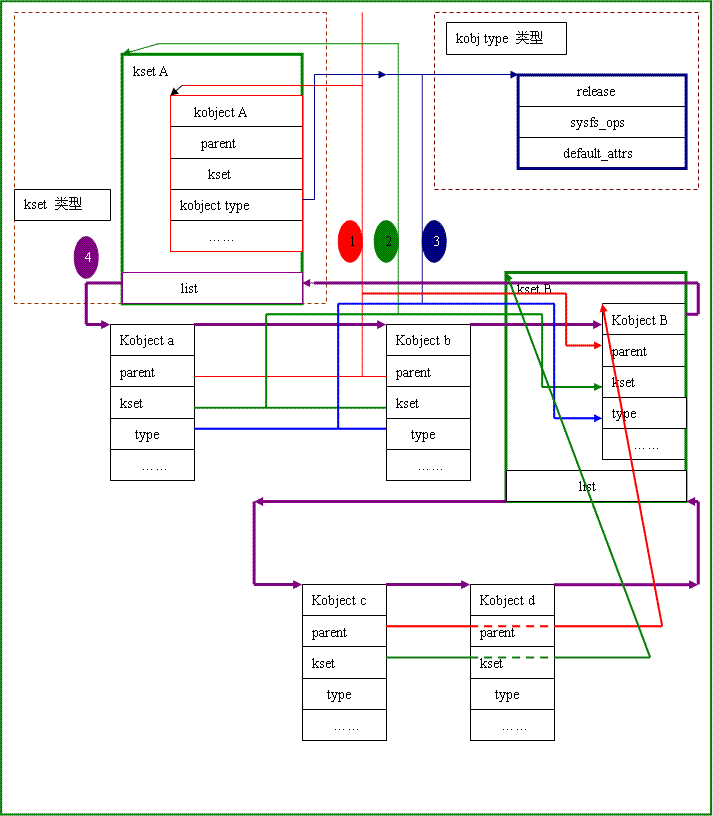
一个kobject结构如下图的kobject 类型部分，而一个kset结构如下图的kset 类型部分，一个kobject加入一个kset，主要是kobject结构体中的相关字段记录了对应的kset信息，①记录了kobject所对应 kset，其所指向的是kset所包含的kobject的地址，②记录了kobject所对应的kset的kset指针，③记录了kobject的类 型，④记录了kset所有的kobject的链子，这个链子是一个双向链表，每当有一个kobject加入到当前的kset,就会调用 list\_add\_tail()函数，把要加入kset的kobject连入链表的结尾，最终形成一个链表。



当有另外一个kobject要加入当前的kset，其中的①②③步跟第一个加入当前kset的kobject是一样的，即把要加入 的kobject的成员设置，使之指向当前的kset对应数据，而④需要把kobject添加到kset的list的尾部，下图表示了kobject b加入到kset A的图示：



当有一个kset，需要加入到当前的kset,其方法也跟一个kobject要加入到当前kset一样，即把要加入的kset中所 包含的kobject的成员设置，使这些成员指向对应的kset的对应数据。而当前kset要加入另一个kset，其方式也是跟一个kset加入到当前 kset一样，都是设备kset中的kobject，使kobject的成员指向要加入的kset的对应数据即可，下图显示了一个kset B加入到kset A中的图示。



由于一条总线要管理总线上的所有驱动，同时要管理总线上的有所设备，则需要再把所有设备和所有驱动都分开，分别设立一个设 备kset和一个设备驱动kset，用于管理所有的设备和设备驱动，如此，则总线kset实际上包含了两个kset（设备kset，设备驱动kset）， 设备kset又包含了所有的当前总线的设备的kobject，设备驱动kset包含了所有的当前总线的设备驱动的kobject；而所有的总线，又形成了 bus kset，归结起来就形成下图的层次关系：



经过上述的设备插入，或者驱动安装，系统就会出现只有设备，而没有设备驱动程序的情况，也会出现，只有设备驱动程序，没有对应的设备的情况，此时，设备或者设备驱动程序，就会暂时在各自的队列里等待，一旦有驱动程序安装，或新的设备插入，就都会自动的去扫描对应的链表，来检测是否有配对的可能。

综合上述三者的关系，如图：



## **Kobj type**

数据结构包含三个域：一个release方法用于释放kobject占用的资源；一个sysfs ops指针指向sysfs操作表和一个sysfs文件系统缺省属性列表。Sysfs操作表包括两个函数store()和show()。当用户态读取属性时，show()函数被调用，该函数编码指定属性值存入buffer中返回给用户态；而store()函数用于存储用户态传入的属性值。

## 参考资料

http://eeepage.info/sysfs/ 对新的sysfs的讲解非常到位

http://www.linuxidc.com/Linux/2012-05/60757.htm 同样讲解sysfs文件系统

http://blog.chinaunix.net/uid-24227137-id-3266449.html 讲解kobject,kset,sysfs

http://blog.csdn.net/xiahouzuoxin/article/details/8943863

# 最小子系统+ramdisk+initrd+系统启动过程

## ramdisk+initrd

ramdisk是一种基于内存的虚拟文件系统，通常用于放置内核的中间数据。

而initrd全称为"boot loader initialized RAM disk"，也就是由启动加载器所初始化的RamDisk设备，它的作用是完善内核的模块机制，让内核的初始化流程更具弹性；内核以及initrd，都由bootloader在机子启动后被加载至内存的指定位置，主要功能为按需加载模块以及按需改变根文件系统。initrd 大体上就是指包含根文件系统的ramdisk。initrd即initial ramdisk,它是在系统引导过程中挂载的一个临时根文件系统.激活系统所须加载的文件系统.



### BIOS(Basic Input Output System)

可以视为是一个永久地记录在ROM中的一个软件，是操作系统输入输出管理系统的一部分

包括：

**自检及初始化**

POST上电自检（如果内存没有插好，开机会滴滴的响，这就是上电自检查处了问题）

**初始化**

程序服务处理

硬件中断处理

枚举本地设备并对其初始化;’

BIOS在内存中建立中断向量表和中断服务程序

系统设置程序（开机按f2或者f12跳出来的配置界面就是配置这个参数）：准确的说法应是通过BIOS设置程序对CMOS参数进行设置

以硬盘启动为例，BIOS此时去读取硬盘驱动器的第一个扇区(MBR，512字节)，然后执行里面的代码；

Bios**将控制权转交给MBR**

**总结:检查硬件并且查找可启动设备[顺序可以是,网络,u盘,硬盘,等等]**

**什么样的是可启动设备呢?设备前512字节最后一个扇区是55aa**

### 系统引导

MBR，它是Master Boot Record的缩写。硬盘的0柱面、0磁头、1扇区称为主引导扇区。它由三个部分组成，**主引导程序(Bootloader)**、 **硬盘分区表DPT（Disk Partition table）**和**硬盘有效标志（55AA）**，其结构图如下所示：



        磁盘分区表包含以下三部分：

        1）Partition ID  （5：延伸  82：Swap   83：Linux   8e：LVM     fd：RAID）

        2）Partition起始磁柱

        3）Partition的磁柱数量

       通常情况下，诸如lilo、grub这些常见的引导程序都直接安装在MBR中。我们以grub为例来分析这个引导过程。

grub引导也分为两个阶段stage1阶段和stage2阶段(有些较新的grub又定义了stage1.5阶段)。

1. stage1：stage1是直接被写入到MBR中去的，也就是上图所看到的前446个字节空间中存放的是stage1的代码。BIOS将stage1载入内存中0x7c00处并跳转执行。stage1（/stage1/start.S）仅仅是将硬盘0头0道2扇区读入内存。而0头0道2扇区内容是源代码中的/stage2/start.S，编译后512字节，它是stage2或者stage1\_5的入口。

**Stage1其实就是MBR。512字节**

**Stage1\_5:用来添加对某种文件系统支持[fat,iso9600,e2fs等等很多很多的文件系统]**

         2)stage2：严格来说这里还应该再区分个stage1.5的，就一并把stage1.5放在这里一起介绍了，免得大家看得心里乱哄哄的。好的，我们继续说0头0到2扇区的/stage2/start.S文件，当它的内容被读入到内存之后，它的主要作用就是负责将stage2或stage1.5从硬盘读到内存中。如果是stage2，它将被载入到0x820处；如果是stage1.5，它将被载入到0x2200处。这里的stage2或者stage1\_5不是/boot分区/boot/grub目录下的文件，因为这个时候grub还没有能力识别任何文件系统。

        如果start.S加载stage1.5：stage1.5它存放在硬盘0头0道3扇区向后的位置，stage1\_5作为stage1和stage2中间的桥梁，stage1\_5有识别文件系统的能力，此后grub才有能力去访问/boot分区/boot/grub目录下的 stage2文件，将stage2载入内存并执行。

        如果start.S加载stage2：同样，这个stage2也不是/boot分区/boot/grub目录下的stage2，这个时候start.S读取的是存放在/boot分区Boot Sector的stage2。这种情况下就有一个限制：因为start.S通过BIOS中断方式直接对硬盘寻址（而非通过访问具体的文件系统），其寻址范围有限，限制在8GB以内。因此这种情况需要将/boot分区分在硬盘8GB寻址空间之前。

        假如是情形2，我们将/boot/grub目录下的内容清空，依然能成功启动grub；假如是情形1，将/boot/grub目录下stage2删除后，则系统启动过程中grub会启动失败。



**最终grub启动kernel内核，启动ramdisk【主要是一些驱动模块，非必要模块】**

**而最终 boot loader 的功能就是加载 kernel（内核）文件**

 Linux 内核是可以通过动态加载内核模块的 (就请想成驱动程序即可)，这些内核模块就放置在 /lib/modules/ 目录内。 由于模块放置到磁盘根目录内 (要记得 /lib 不可以与 / 分别放在不同的分区)，因此在启动的过程中内核必须要挂载根目录，这样才能够读取内核模块提供加载驱动程序的功能。而且为了避免影响到磁盘内的文件系统，因此启动过程中根目录是以只读的方式来挂载。

一般来说，非必要的功能且可以编译成为模块的内核功能，目前的 Linux distributions 都会将它编译成为模块。因此 U盘, SATA, SCSI... 等磁盘设备的驱动程序通常都是以模块的方式来存在的。现在来思考一种情况，假设你的 linux 是安装在 SATA 磁盘上面的，你可以通过 BIOS 的 INT 13 取得 boot loader 与 kernel 文件来启动，然后 kernel 会开始接管系统并且检测硬件及尝试挂载根目录来取得额外的驱动程序。

问题是，内核根本不认识 SATA 磁盘，所以需要加载 SATA 磁盘的驱动程序，否则根本就无法挂载根目录。但是 SATA 的驱动程序在 /lib/modules 内，你根本无法挂载根目录，又怎么读取到 /lib/modules/ 内的驱动程序？是吧！是不是有点进退两难？在这个情况之下，你的 Linux 是无法顺利启动的！ 那怎办？没关系，我们可以透过虚拟文件系统来处理这个问题。

虚拟文件系统 (Initial RAM Disk) 一般使用的文件名为 /boot/initrd ，这个文件的特色是，它也能够通过boot loader 来加载到内存中， 然后这个文件会被解压缩并且在内存当中仿真成一个根目录，且此仿真在内存当中的文件系统能够提供一个可执行的程序，通过该程序来加载启动过程中所最需要的内核模块，通常这些模块就是 U盘, RAID, LVM, SCSI 等文件系统与磁盘接口的驱动程序。等载入完成后，会帮助内核重新调用 /sbin/init 来开始后续的正常启动流程，如图



一张图片关于bios以后的bootloader 这张图片 讲解的非常详细

Ps：**最终grub启动kernel内核，启动【initrd】ramdisk【主要是一些驱动模块，非必要模块】**

### 关于boot.img解压缩以后就是kernel+ramdisk.gz两个文件

bzimage包含以下目标文件 bootsect.o + setup.o + misc.o + piggy.o .

bootsect：这个程序是linuxkernel的第一个程序，包括了linux自己的bootstrap程序，主要进行开机后加载真正内核镜像之前的各种准备工作.注意它是用来load bzimage，不是bzimage的一部分（这句话存在质疑）。

setup:进行实模式设置

misc：

piggy.o 包含被压缩的vmlinux

bzimage的解剖图：



kernel 的几种格式

1、vmlinux 编译出来的最原始的内核文件，未压缩。用于kernel-debug，产生system.map符号表，不能用于直接加载，不可以作为启动内核。只是启动过程中的中间媒体。vmlinux.bin ： The same as vmlinux, but in a binary file format.

2、zImage是ARM Linux常用的一种压缩映像文件，zImage是一般情况下默认的压缩内核映像文件，压缩vmlinux，加上一段解压启动代码得到，只能从0X0地址运行。zImage是vmlinux经过gzip压缩后的文件。bzImage bz表示“big zImage”，不是用bzip2压缩的。两者的不同之处在于，zImage解压缩内核到低端内存(第一个640K)，bzImage解压缩内核到高端内存(1M以上)。如果内核比较小，那么采用zImage或bzImage都行，如果比较大应该用bzImage。

vmlinuz是bzImage/zImage文件的拷贝或指向bzImage/zImage的链接。vmlinuz是可引导的、压缩的内核。“vm”代表“VirtualMemory”。Linux 支持虚拟内存，不像老的操作系统比如DOS有640KB内存的限制。Linux能够使用硬盘空间作为虚拟内存，因此得名“vm”。

3、uImage是U-boot专用的映像文件，使用工具mkimage对普通的压缩内核映像文件（zImage）加工而得。它是在zImage之前加上一个长度为0x40的“头”，说明这个映像文件的类型、加载位置、生成时间、大小等信息。换句话说，如果直接从uImage的0x40位置开始执行，zImage和uImage没有任何区别。另外，Linux2.4内核不支持uImage，Linux2.6内核加入了很多对嵌入式系统的支持，但是uImage的生成也需要设置。u-boot使用bootm命令从任意地址解压启动内核。由于bootloader一般要占用0X0地址，所以，uImage相比zImage的好处就是可以和bootloader共存。

**Ramdisk.gz解压以后是initrd文件夹，里面包括**



ramdisk.img中比较重要的文件是"init","init.rc"，其中init是system/core/init/init.c编译而来，boot.img中ramdisk里的init.rc位于system/core/init/init.rc,而recovery.img中ramdisk里的init.rc位于bootable/recovery/etc/init.rc。

　　kernel加载结束以后第一个进程是执行init，init会解析init.rc文件，并起相应的服务。



首先CPU开机或重置之后执行Bootloader程式，其会进行初始化RAM，以载入kernel与RAM DISK并跳入kernel；初始化kernel非常依赖硬件，但它的目的是设定好一切，让CPU能够尽快执行C程式码。一旦完成，kernel便跳入架构独立的start\_kernel()函式，初始化它的各个子系统，并且开启呼叫全部内建驱动程式的"init"函式，接着kernel便挂载它的根档案系统（rootfs），并启动init行程。这时Android的init介入，并执行/ init.rc内得指令，以设定环境变数，建立挂载点，挂载档案系统，设定OOM调整，并且启动原生background处理程序。有很多原生background处理程序(也就是前述的Native Service )在此时被启动，把焦点放在Zygote；Zygote的工作就是启动APP，其主要功能就是聚焦于此，好让所有APP共用的组件都能一致并缩短其启动时间；但init实际上不直接启动Zygote，而是使用app\_process命令，让Zygote被Android Runtime启动，Android Runtime接着启动系统第一个Dalvik VM，并且要它呼叫Zygote的main()。Zygote启动System Server，这是Zygote启动的第一支APP，它以独立的父行程方式持续存活；接着，System Server开始初始化它所存放的每一个系统服务，并且向之前所启动的Server Manager注册它们；它们所开启的服务之一，Activity Manage，将透过传送Intent.CATEGORY\_HOME型别来结束它的初始化，以启动Launcher APP。

## 参考资料

<http://blog.chinaunix.net/uid-23069658-id-3142047.html>

<http://blog.sina.com.cn/s/blog_c70e10380102w9b5.html>

<http://blog.csdn.net/miss_acha/article/details/50004717>

<http://www.2cto.com/os/201603/494064.html>

http://blog.itpub.net/8111049/viewspace-680043

<http://www.ibm.com/developerworks/cn/linux/l-initrd.html>

<http://bbs.chinaunix.net/thread-2046548-1-1.html>

http://bbs.chinaunix.net/thread-2046548-1-1.html

<https://embeddedandroidsystem.wordpress.com/2014/03/30/%E4%B8%80%E8%B5%B7%E4%BE%86%E5%AD%B8%E5%B5%8C%E5%85%A5%E5%BC%8Fandroid%E7%B3%BB%E7%B5%B1-%E7%AC%AC%E4%BA%8C%E7%AB%A0%E5%BF%83%E5%BE%97%E4%B8%89/>

小知识：

\*.o 中间文件

\*.so 文件是动态链接库文件，相当于 win下的 .dll 文件。

\*.a 文件是静态库文件。

\*.ko 是内核模块文件，是内核加载的某个模块，一般是驱动程序。

具体的编译命令请百度

# Paltform

## Platform设备驱动

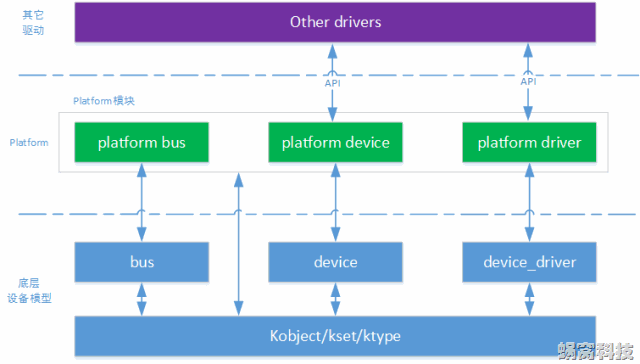
可以通过CPU bus直接寻址（例如在嵌入式系统常见的“寄存器”）。因此，由于这个共性，内核在设备模型的基础上（device和device\_driver），对这些设备进行了更进一步的封装，抽象出paltform bus、platform device和platform driver，以便驱动开发人员可以方便的开发这类设备的驱动。

Platform设备驱动包含三部分：**Platform总线**，**Platform设备**，**Platform设备驱动**

然而这三者是基于设备模型的概念：总线，设备与驱动。

## Platform模块的软件架构

内核中Platform设备有关的实现位include/linux/platform\_device.h和drivers/base/platform.c两个文件中，它的软件架构如下：



结合之前的知识讲解一下这幅图：每一个单独的结构体都是一个kobject（上图中的bus,device,device\_driver,platformbus,platform\_device,platform\_driver）；内核对这些东西又有一个整理，所有的bus是一个kset；所有的device是一个kset；所有的driver是一个kset；而ktype则是对这些kobject的附属操作。

Platform Bus，基于底层bus模块，抽象出一个虚拟的Platform bus，用于挂载Platform设备；

Platform Device，基于底层device模块，抽象出Platform Device，用于表示Platform设备；

Platform Driver，基于底层device\_driver模块，抽象出Platform Driver，用于驱动Platform设备。

其中Platform Device和Platform Driver会为其它Driver提供封装好的API

## Platform模块向其它模块提供的API汇整

Platform提供的接口包括：Platform Device和Platform Driver两个数据结构，以及它们的操作函数

用于抽象Platform设备的数据结构----“struct platform\_device”：

用于抽象Platform设备驱动的数据结构----“struct platform\_driver”：

具体的各类api请参考：<http://www.wowotech.net/device_model/platform_device.html>

## 总结大多数驱动框架

## Platform led驱动

最简单的了解platform平台的例子，可以理解为3部分，由驱动层，系统核心层，设备驱动三部分组成：

驱动层：硬件设备注册部分。

系统核心层：无

设备驱动层：设备端的实现，如led闪烁等

实际上之所以这里分成3部分，是为了与后面的设备驱动程序对应起来。

使用步骤示例：

(1)platform\_device\_register()：注册平台led设备。

(2)platform\_driver\_register()：注册平台led驱动。

## Platform input驱动

Linux系统提供了input子系统，按键、触摸屏、键盘、鼠标等输入都可以利用input接口函数来实现设备驱动。

在linux主要由驱动层，系统核心层(Input Core)和事件处理层（Event Handler）三部份组成。

驱动层：硬件设备注册部分，只是把输入设备注册到input子系统中，在驱动层的代码本身并不创建结点。对应文件如gpio\_key.c

Input core：向系统报告按键、触摸屏、键盘、鼠标等输入事件(event,通过input\_event结构体描述)，使得驱动层不需要关心文件操作接口。对应文件如Input.c

Event Handler：提供input设备接口。 对应文件如evdev.c,mousedev.c等。

**一般来说，如果要使用input子系统，只需要更改驱动层部分就可以了。**

## Platform i2c驱动

Linux系统中,i2c驱动由3部分组成，即i2c总线驱动、i2c core、i2c设备驱动。

I2c总线驱动：对i2c硬件体系结构中适配器端的实现，适配器可由CPU控制，或集成在CPU内部。对应文件如:i2c-at91.c

I2c core：提供了i2c总线驱动和设备驱动的注册、注销方法，i2c algorithm。与具体适配器无关的代码以及探测设备、检测设备地址的上层代码。对应文件如:i2c-core.c

I2c设备驱动：i2c体系硬件结构中设备端的实现，设备一般挂在受CPU控制的i2c适配器上，通过i2c适配器与CPU交换数据。对应文件如:at24.c,i2c-dev.c等。

对于常见的开发板来说，主芯片已经带了i2c总线，i2c总线驱动基本上提供了，不用怎么动。即使不带i2c总线，基本上也会提供io模拟的i2c，也就是说i2c总线驱动部分一般情况下不需要自己写或者更改。I2c core部分就更不用动了，呵呵。因此，写一个i2c设备的驱动，只需要写i2c设备驱动（这里对应于上面说的i2c驱动的3部分之一）就可以了。

大多数i2c设备驱动，内核已经提供了。而且简单的应用还可以利用i2c-dev.c来实现。

## Platform spi驱动

   Linux系统中，spi驱动由3部分组成，即spi总线驱动、spi core、spi设备驱动。

   Spi总线驱动:硬件spi驱动的实现，spi可为主芯片内部集成，也可以io口模拟。对应文件如：atmel\_spi.c

   Spi core:提供了spi总线驱动和设备驱动的注册、注销方法。

   Spi 设备驱动：spi体系结构中,spi设备端的实现。

综合上述几个比较简单的驱动可以看出一个共性：

这几个驱动基本都是由3部分组成：

（1） 总线驱动：与所选用的主芯片相关联，一般都有提供。

（2） 总线core：与具体的硬件无关，内核已经提供。

（3） 总线设备驱动：所操作的具体设备。根据实际应用需要，使用或更改内核已经提供的驱动，或者自己重新写一个驱动。

实际上，写一个设备驱动，我们所要做的工作基本上集中在第3部分，而这部分，内核也提供了大多数设备的驱动，即使没有提供，我们也可以根据已有的设备自己更改。

## 参考资料

<http://blog.chinaunix.net/uid-27041925-id-3884955.html>

<http://www.wowotech.net/device_model/platform_device.html>

http://blog.chinaunix.net/uid-25014876-id-111745.html

# 中断子系统

## 中断

Cpu在执行程序的过程中，出现某些突发时间急待处理，cpu必须暂时停止当前程序的执行，转去处理突发事件，处理完毕后又返回原程序被中断的位置继续执行。

外设产生中断是异步发生的，硬件设备生成中断的时候并不考虑与处理器的时钟同步——也就是说中断随时可以产生。内核随时可能因为新到来的中断而被打断。从物理学的角度看，中断是一种电信号，由硬件设备生成，并直接送入中断控制器的输入引脚。然后由中断控制器向处理器发送相应的信号。

中断是处理器和外设之间的一种通信机制，也是操作系统内核对外设进行管理的一种机制。外设通过发出特殊的电信号通知处理器发生了一次中断，处理器收到信号后就通知操作系统，然后操作系统负责做出相应的处理。

一个完整的设备中，与中断相关的硬件可以划分为3类，它们分别是：设备、中断控制器和CPU本身，下图展示了一个smp系统中的中断硬件的组成结构：



                          图  中断系统的硬件组成

**设备**  设备是发起中断的源，当设备需要请求某种服务的时候，它会发起一个硬件中断信号，通常，该信号会连接至中断控制器，由中断控制器做进一步的处理。在现代的移动设备中，发起中断的设备可以位于soc（system-on-chip）芯片的外部，也可以位于soc的内部，因为目前大多数soc都集成了大量的硬件IP，例如I2C、SPI、Display Controller等等。

Ps：中断优先寄存器，其英文缩写IP，实为“Interrupt Priority”的简写。

**中断控制器**  中断控制器负责收集所有中断源发起的中断，现有的中断控制器几乎都是可编程的，通过对中断控制器的编程，我们可以控制每个中断源的优先级、中断的电器类型，还可以打开和关闭某一个中断源，在smp系统中，甚至可以控制某个中断源发往哪一个CPU进行处理。对于ARM架构的soc，使用较多的中断控制器是VIC（Vector Interrupt Controller），进入多核时代以后，GIC（General Interrupt Controller）的应用也开始逐渐变多。

**CPU**  cpu是最终响应中断的部件，它通过对可编程中断控制器的编程操作，控制和管理者系统中的每个中断，当中断控制器最终判定一个中断可以被处理时，他会根据事先的设定，通知其中一个或者是某几个cpu对该中断进行处理，虽然中断控制器可以同时通知数个cpu对某一个中断进行处理，实际上，最后只会有一个cpu相应这个中断请求，但具体是哪个cpu进行响应是可能是随机的，中断控制器在硬件上对这一特性进行了保证，不过这也依赖于操作系统对中断系统的软件实现。在smp系统中，cpu之间也通过IPI（inter processor interrupt）中断进行通信

## IRQ编号

系统中每一个注册的中断源，都会分配一个唯一的编号用于识别该中断，我们称之为IRQ编号。IRQ编号贯穿在整个Linux的通用中断子系统中。在移动设备中，每个中断源的IRQ编号都会在arch相关的一些头文件中，例如arch/xxx/mach-xxx/include/irqs.h。驱动程序在请求中断服务时，它会使用IRQ编号注册该中断，中断发生时，cpu通常会从中断控制器中获取相关信息，然后计算出相应的IRQ编号，然后把该IRQ编号传递到相应的驱动程序中。

## 中断处理程序

      在响应一个特定中断的时候，内核会执行一个函数，该函数叫做中断处理程序或中断服务例程（interrupt service routine,ISR）。产生中断的每一个设备都有一个（中断处理程序通常不是和特定设备相关联，而是和特定中断相关联，也就是说，若一个设备可以产生 多种不同的中断，那么该设备就可以对应多个中断处理程序，相应的，该设备的驱动也就需要准备多个这样的函数。）相应的中断处理程序。一个设备的中断程序是它设备驱动程序的一部分——设备驱动程序是用于对设备进行管理的内核代码。

       在Linux中，中断处理程序看起来就是普普通通的C函数。只不过这些函数必须按照特定的类型声明，以便内核能够以标准的方式传递处理程序的信息。中断处理程序与其它内核函数的真正区别在于：中断处理程序是被内核调用来响应中断的，而它们运行于我们称之为中断上下文的特殊上下文中。

**执行在中断上下文中的代码需要注意的一些事项：**

**中断上下文中的代码不能进入休眠。**

**不能使用mutex，只能使用自旋锁，且仅当必须时。**

**中断处理函数不能直接与用户空间进行数据交换。**

**中断处理程序应该尽快结束。**

**中断处理程序不需要是可重入的，因为相同的中断处理函数不能同时在多个处理器上运行。**

**中断处理程序可能被一个优先级更高的中断处理程序所中断。为了避免这种情况，可以要求内核将中断处理程序标记为一个快速中断处理程序（将本地CPU上的所有中断禁用），不过在采取这个动作前要慎重考虑对系统的影响**

## 上半部与下半部

ISR的执行必须足够快速，这是由多种原因决定的。由于中断可以随时发生，因此ISR可以随时被执行，同时ISR打断了系统正常的流程，另外产生中断的外设也需要系统快速响应，最后由于linux不允许相同ISR中断嵌套，执行一个ISR时，当前中断线会在所有CPU上被屏蔽，这段屏蔽时间最好越短越好（否则会降低系统反应速度）。以上这些限制条件都要求ISR执行足够快速。由于速度的要求，ISR中不可能做太多的工作，而只能做一些必不可少的、跟硬件相关的工作，其他的工作就放到所谓的下半部里完成。

把中断处理划分为上半部和下半部是出于现实的考虑。上半部就是ISR，一接收到中断就立即执行，执行时禁止相同中断（必要时禁止所有中断），只做有严格时限且与硬件相关的工作，例如对接收的中断进行应答或复位硬件；而可以稍后执行或者与硬件无关的动作都放到下半部，在合适的时机下半部才开始执行。**上半部和下半部的关键区别是，上半部简单快速，执行时禁止一部分或者全部中断；下半部稍后执行，而且执行期间可以响应所有中断。**

举个网卡的例子，

当网卡收到来自网络的数据包时，会立刻产生中断来通知内核，

而内核收到中断后也会立刻调用网卡已注册的ISR，

ISR会通知网卡“你的中断已经被组织收到”，并拷贝网卡收到的数据到内存。

这些都是紧迫并与硬件相关的动作，如果内核不及时拷贝网卡缓存中的数据，网卡缓存很可能溢出，造成丢包现象。当数据都被拷贝到内存之后，ISR的任务算完成了，此时它将控制权转交给被它打断的程序。

而处理内存中的数据的任务则落在下半部里，在内核觉得合适的时机（不太繁忙），下半部将会执行。

**上半部和下半部的划分可以参考以下原则：**

**1) 如果一个任务对时间非常敏感，把它放在上半部（ISR）里；**

**2) 如果一个任务跟硬件相关，把它放在上半部里；**

**3) 如果一个任务要保证不被其他中断（特别是相同的中断）打断，把它放在上半部里；**

**4) 其他所有的任务，都放在下半部。**

下半部负责推后完成的工作，但是并不需要指明一个具体的时间，只是将任务稍微推迟，待到系统不是那么繁忙并且中断恢复之后执行就可以了（一般情况下，ISR一返回，下半部就会执行）。不同于ISR的最关键之处是，下半部执行的时候允许响应所有的中断。内核的策略是，当中断不是特别多的时候，及时处理中断，所以do\_irq会调用do\_softirq。

当系统中断过多时，do\_softirq才会被推迟到内核的ksoftirq内核线程中去。如何判断中断过多呢，linux的认为发生中断嵌套了，就是中断过多。do\_irq在调用do\_softirq时会以此为判断条件。

## 在驱动程序中申请中断

Linux中断子系统向驱动程序提供了一系列的API，其中的一个用于向系统申请中断：



request\_threaded\_irq()是Linux kernel 2.6.30 之后新加的irq handler API，如何确定可以用到 request\_threaded\_irq() ?

Linux kernel config 需要定义CONFIG\_GENERIC\_HARDIQS

kernel config 才有支援threaded irq ；Moving interrupts to threads 介绍request\_threaded\_irq() 的由来

http://lwn.net/Articles/302043/

从realtime tree 移植而来，为了减少kernel 因为要等待每一个硬件中断处理的时间 ，就另外交给kernel thread 处理中断后续工作。

 优点：

1 减少 kernel 延迟时间

2 避免处理中断时要分辨是在硬体中断或软体中断？

3 更容易为kernel 中断处理除错，可能可完全取代tasklet

原本的中断处理分上半部(硬体中断处理，必须关闭中断无法处理新的中断)跟下半部( 软体中断处理)，因此上半部的硬体中断处理必须尽可能简短，让系统反应速度更快。  request\_threaded\_irq 是在将上半部的硬件中断处理缩短为只确定硬体中断来 自我们要处理的装置，唤醒kernel thread 执行后续中断任务。

  缺点：

对于非irq 中断的kernel threads ，需要在原本task\_struct 新增struct irqaction 多占 4/8 bytes 记忆体空间，linux kernel 2.6.29 之后(2.6.30)加入request\_threaded\_irq 跟传统top/bottom havles 的差异是threaded\_irq 受Linux kernel system 的 process scheduling 控制，不会因为写错的bottom half 代码造成整个系统 延迟的问题。

也可以透过RT/non RT 跟nice 等工具调整各个thread 优先权，丢给使用率较低的  cpu 以及受惠于kernel 原本可以对threads 做的各种控制，包括但不限于sleep, lock, allocate 新的记忆体区块。

受惠最大的是shared irq line 的多个中断处理。除了可以加速共享中断造成的延迟;threaded\_irq 也可以降低在同一段程式码处理多个装置中断的复杂度。 threaded irq 在使用性上也比tasklet(接着top half 直接执行，无法sleep) /workqueue(kernel context?) 等需要在top half 增加跟bottom half 连结与沟通  的麻烦。

**Ps:最简单直观的理解，handler是顶半部，thread\_fn是底半部（个人猜测，不确定正确）**

**现在看来上面那句话确实不准确。感觉这个函数把中断重新定义了。不能说什么顶半部和底半部。**

理解自宋宝华：申请一个线程化的IRQ，kernel会为中断的底版本创建一个名字为irq/%d-%s的线程，%d对应着中断号。其中顶半部（硬中断）**handler**在做完必要的处理工作之后，会返回IRQ\_WAKE\_THREAD，之后kernel会唤醒irq/%d-%s线程，而该kernel线程会调用**thread\_fn**函数，因此，该线程成为底半部。

该机制目前在kernel中使用已经十分广泛，可以认为是继softirq（含tasklet）和workqueue之后的又一大中断底半部方式。

**irq**  需要申请的irq编号，对于ARM体系，irq编号通常在平台级的代码中事先定义好，有时候也可以动态申请。

**handler**  中断服务回调函数，该回调运行在中断上下文中，并且cpu的本地中断处于关闭状态，所以该回调函数应该只是执行需要快速响应的操作，执行时间应该尽可能短小，耗时的工作最好留给下面的thread\_fn回调处理。

**thread\_fn**  如果该参数不为NULL，内核会为该irq创建一个内核线程，当中断发生时，如果handler回调返回值是IRQ\_WAKE\_THREAD，内核将会激活中断线程，在中断线程中，该回调函数将被调用，所以，该回调函数运行在进程上下文中，允许进行阻塞操作。

**flags**  控制中断行为的位标志，IRQF\_XXXX，例如：IRQF\_TRIGGER\_RISING，IRQF\_TRIGGER\_LOW，IRQF\_SHARED等，在include/linux/interrupt.h中定义。

**name**  申请本中断服务的设备名称，同时也作为中断线程的名称，该名称可以在/proc/interrupts文件中显示。

**dev**  当多个设备的中断线共享同一个irq时，它会作为handler的参数，用于区分不同的设备。

## flags:与中断相关的标志

IRQF\_TRIGGER\_RISING：上升沿触发

IRQF\_TRIGGER\_FALLING：下降沿触发

IRQF\_TRIGGER\_HIGH：高电平触发

IRQF\_TRIGGER\_LOW：低电平触发

IRQF\_SAMPLE\_RANDOM：为系统随机发生器提供支持

IRQF\_SHARED：中断可在设备间共享

IRQF\_DISABLED：是否快速中断

IRQF\_ONESHOT:选项说明该中断已经被线程化了（而且是特殊的one shot类型的）

（虽然kernel的注释上是这样说，但是request\_threaded\_irq时还是必须要设定IRQF\_ONESHOT，否者它就会被错误的强制线程化了。

其实我觉得这个ONESHOT的命名也是很不好，它不仅仅表示one shot，而且还是thread irq的标志。）

## **request\_threaded\_irq的工作流程**

函数先是根据irq编号取出对应的irq\_desc实例的指针，然后分配了一个irqaction结构，用参数handler，thread\_fn，irqflags，devname，dev\_id初始化irqaction结构的各字段，同时做了一些必要的条件判断：该irq是否禁止申请？handler和thread\_fn不允许同时为NULL，最后把大部分工作委托给\_\_setup\_irq函数：



图 request\_threaded\_irq（部分）

进入\_\_setup\_irq函数，如果参数flag中设置了IRQF\_SAMPLE\_RANDOM标志，它会调用rand\_initialize\_irq，以便对随机数的生成产生影响。如果申请的不是一个线程嵌套中断（关于线程嵌套中断，请参阅[Linux中断（interrupt）子系统之三：中断流控处理层](http://blog.csdn.net/droidphone/article/details/7489756)中的handle\_nested\_irq一节），而且提供了thread\_fn参数，它将创建一个内核线程;这个函数会判断handler是否可以线程化，如果可以，就会将它线程化：



图 \_\_setup\_irq（部分—与最新内核有区别）

**最新内核:**

**\_setup\_irq->setup\_irq\_thread->kthread\_create->**[**kthread\_create\_on\_node**](http://lxr.free-electrons.com/ident?i=kthread_create_on_node)

如果irq\_desc结构中断action链表不为空，说明这个irq已经被其它设备申请过，也就是说，这是一个共享中断，所以接下来会判断这个新申请的中断与已经申请的旧中断的以下几个标志是否一致：

一定要设置了IRQF\_SHARED标志

电气触发方式要完全一样（IRQF\_TRIGGER\_XXXX）

IRQF\_PERCPU要一致

IRQF\_ONESHOT要一致

检查这些条件都是因为多个设备试图共享一根中断线，试想一下，如果一个设备要求上升沿中断，一个设备要求电平中断，当中断到达时，内核将不知如何选择合适的流控操作。完成检查后，函数找出action链表中最后一个irqaction实例的指针。

如果这不是一个共享中断，或者是共享中断的第一次申请，函数将初始化irq\_desc结构中断线程等待结构：wait\_for\_threads，disable\_irq函数会使用该字段等待所有irq线程的结束。接下来设置中断控制器的电气触发类型，然后处理一些必要的IRQF\_XXXX标志位。如果没有设置IRQF\_NOAUTOEN标志，则调用irq\_startup()打开该irq，在irq\_startup()函数中irq\_desc中的enable\_irq/disable\_irq嵌套深度字段depth设置为0，代表该irq已经打开，如果在没有任何disable\_irq被调用的情况下，enable\_irq将会打印一个警告信息。

接着，设置cpu和irq的亲缘关系

然后，把新的irqaction实例链接到action链表的最后

最后，唤醒中断线程，注册相关的/proc文件节点

至此，irq的申请宣告完毕，当中断发生时，处理的路径将会沿着：irq\_desc.handle\_irq，irqaction.handler，irqaction.thread\_fn（irqaction.handler的返回值是IRQ\_WAKE\_THREAD）这个过程进行处理。下图表明了某个irq被申请后，各个数据结构之间的关系：



附注：此函数申请一个中断处理函数（若赋值为NULL，则使用默认的）和线程，当中断发生时，中断处理函数运行于中断上下文，判断是否是自己所属的中断，若是则关闭自身中断并返回IRQ\_WAKE\_THREAD来唤醒线程函数的执行。在线程执行中，可以睡眠，执行后返回IRQ\_HANDLE

## 下半部的几种机制

## 软中断(softirqs)

软中断是一组在编译期间静态分配的下半部接口，一共有个32个。一个软中断不会抢占另一个软中断，唯一能够抢占软中断的是ISR。不同的软中断可以同时在多个CPU上执行，甚至同一个软中断也可以在所有CPU上同时执行，所以软中断必须设计成可重入函数，同时往往需要使用锁机制来保护共享数据

软中断保留给系统中对时间要求最严格以及最重要的下半部使用。在2.6版本中，只有两个子系统（网络和SCSI）直接使用了软中断。此外内核定时器和tasklet都是建立在软中断基础上的。

软中断运行时可以响应中断，但是不能休眠（因为此时仍然处于中断上下文）。当一个CPU正在执行某个软中断时，这个CPU会禁止软中断，但是别的CPU仍然可以执行新触发的软中断（相同的或不同的都可以），这也是软中断的优势——可以在多CPU上并行执行；但相应的tradeoff是，你要小心的保护好共享数据以避免竞争。在实践中，大部分软中断处理程序都采用per-CPU变量来避免显示的加锁，从而提高性能。

## Tasklet

tasklet是一种基于软中断的延时处理机制，是中断底半部的一种处理方式。基本上要使用就是申请中断，通过宏创建tasklet与处理函数的关联，在顶半部调用tasklet\_schedule使系统在适当的时候进行调（就是把tasklet\_struct结构体挂到tasklet\_vec链表或者挂接到tasklet\_hi\_vec链表上，并调度软中断TASKLET\_SOFTIRQ或者HI\_SOFTIRQ；Tasklet\_action在软中断TASKLET\_SOFTIRQ被调度到后会被执行，它从tasklet\_vec链表中把tasklet\_struct结构体都取下来，然后逐个执行。如果t->count的值等于0，说明这个tasklet在调度之后，被disable掉了，所以会将tasklet结构体重新放回到tasklet\_vec链表，并重新调度TASKLET\_SOFTIRQ软中断，在之后enable这个tasklet之后重新再执行它。）；

## work queue

前面提到的**软中断和tasklet都运行于中断上下文**，因此不可以睡眠；如果需要在下半部中睡眠，那就只能使用工作队列了。工作队列可以把下半部的操作交给一个内核线程来执行——也就是说运行在进程上下文中，因而是可以睡眠的。

Ps：**参考request\_threaded\_irq;此时如果使用request\_threaded\_irq申请的中断，handler不是在中断上下文里执行，而是在新创建的线程里执行，这样，该handler非常像执行workqueue，拥有所有workqueue的特性，但是省掉了创建,初始化，调度workqueue的繁多步骤。处理起来非常简单**

**主要区别是tasklet是在中断上下文执行,而workqueue是在process上下文，因此可以执行可能sleep的操作。**

## 内核定时器

前面提到的下半部机制都是将操作推迟到除了现在之外的其他时间，而内核定时器可以确保将操作推迟到某个确定的时间来执行。如果必须确保在某一个确定的时间间隔以后再运行下半部操作，那么需要使用内核定时器。内核定时器其实也是在软中断基础上实现的。

## 参考资料

http://blog.csdn.net/21cnbao/article/details/8090398 request\_threaded\_irq

http://www.kuqin.com/shuoit/20140104/337421.html 讲述了什么是tasklet

http://blog.csdn.net/lizuobin2/article/details/51793911 这篇讲的比较好，

把中断对应机制分的很详细；但是只是做了一个详细的划分，机制原理讲述不清

http://blog.csdn.net/songjinshi/article/details/23262923 讲述调用调度程序的时机

http://blog.sina.com.cn/s/blog\_70a9dd840100uqfh.html 讲述了什么是isr，

也涉及了中断上下文以及中断上半部和下半部

http://blog.sina.com.cn/s/blog\_65373f1401018w15.html

http://blog.sina.com.cn/s/blog\_510ac74901015fgz.html

http://blog.csdn.net/DroidPhone/article/details/7445825 中断子系统的一系列文章

http://blog.csdn.net/lickylin/article/details/12657373

http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=21977330&id=3755609

ps 三个有用的调试函数

#define hardirq\_count() (preempt\_count() & HARDIRQ\_MASK)  
#define softirq\_count() (preempt\_count() & SOFTIRQ\_MASK)  
#define irq\_count() (preempt\_count() & (HARDIRQ\_MASK | SOFTIRQ\_MASK \  
     | NMI\_MASK))

/\*  
 \* Are we doing bottom half or hardware interrupt processing?  
 \* Are we in a softirq context? Interrupt context?  
 \*/  
#define in\_irq()   (hardirq\_count())      //判断当前是否在硬件中断上下文  
#define in\_softirq()  (softirq\_count())  //判断当前是否在软件中断上下文  
#define in\_interrupt()  (irq\_count())    //判断当前是否在硬件、软件、底半部中断上下文

# ARM概念梳理：Architecture, Core, CPU，SOC

### 概念梳理

1）ARM architecture

ARM architecture，是指ARM公司开发的、基于精简指令集架构（RISC, Reduced Instruction Set Computing architecture）的指令集架构（Instruction set architecture）。我们常说的ARMv7、ARMv8、ARMv8-A，就是指ARM architecture。类似的基于RISC的architecture也有很多，例如MIPS、AVR、Blackfin等等，都是这个概念。

2）ARM core

ARM core是基于ARM architecture开发出来的IP core，它是介于architecture和最终的CPU(MCU)之间的中间产品，这也是ARM商业模式的独特之处。

有两种类型的ARM core：一种是ARM公司自己发布的，如我们耳熟能详的ARM7、ARM9、ARM Cortex M3、ARM Cortex A57等等；另一种是ARM授权其它公司开发的Core，如苹果的A6/A6X等等。下面链接是维基百科上的ARM core的列表，共大家参考：

<http://en.wikipedia.org/wiki/List_of_ARM_microarchitectures>

3）ARM CPU（MCU）

其它的芯片厂商，如Phillips、ST、TI等，会基于ARM公司发布的Core，开发自己的ARM处理器，这称作ARM CPU（也可称为MCU）。这些是我们工作过程中接触最多的，如LPCxxxx、STM32xxx、OMAPxxxx、S3Cxxxx等等。

4）ARM Soc

对于一些比较专业的应用场景，如视频、音频等，为了追求更小的size、更低的功耗，厂商会在芯片上，集成除处理器之外的东西，如视频编解码器、DSP等。这些集成了其它功能的芯片，称作片上系统（SOC），如TI的DM37x Video SOC。

注1：其实ARM的技术和商业模式，正体现了软件工程中抽象和封装的思想。

### ARM 64bit

我们以一款64bit ARM CPU为例，反向阐述一下ARM处理的诞生过程，同时罗列一些学习、研究方向。

1）我们熟悉一个CPU（假设它的型号是WW9000）的第一手资料，是芯片厂家发布的Datasheet，例如WW9000\_SPEC.pdf。

2）WW9000是基于ARM Cortex-A57 Core封装而来的，该ARM core的资料可以从下面链接下载

<http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488g/DDI0488G_cortex_a57_mpcore_trm.pdf>

3）ARM Cortex-A57 Core又是基于ARMv8-A architecture，该结构的资料可以通过如下方式获取：

Go to [ARM Infocenter](http://infocenter.arm.com/) and navigate through ARM architecture / Reference Manuals

注2：[ARM Infocenter](http://infocenter.arm.com/)中资料是非常全面的，没事时可以多逛逛。

## 参考资料

<http://www.wowotech.net/armv8a_arch/arm_concept.html>

ps：

 Linux内核支持众多体系结构，内核把与设备无关的代码放在arch目录，对应的头文件放在include/asm-<体系名称>目录下。这样的划分代码结构清晰，同时提高了代码的复用率。**在arch目录里，每个子目录对应一种体系结构，存放这种体系结构对应的代码，如果代码较多会单独建立一个目录**，例如arch/arm目录下，有一个kernel目录，存放的是kernel目录中在arm体系结构上特有的函数或者实现方法；在arch/i386目录存放了Intel X86体系结构的代码，不仅有kernel目录，而且还有多个目录，例如mm目录包含了x86体系上内存管理的实现方法，math-emu包含了x86体系上浮点数模拟的实现等。读者在阅读内核代码的时候可以从一个体系结构代码入手，对不同体系结构移植代码的主要工作是arch里面的代码。

##### Architecture和Machine的概念

内核代码中最常见的目录结构就是：arch/xxx/mach-xxx/（例如arch/arm/mach-bcm/）。由该目录结构可知，Architecture（简称arch）是指具体的体系结构，如ARM、X86等等。Machine呢，是指具体体系结构下的一个或一系列的SOC，如bcm等。

## pmic与pmu的区别

pmu是cpu这边的，pmic是外设；pmic是供电给主芯片的，pmu负责把这些电分配给cpu的各个单元，也负责问pmic要合适的电；pmu是cpu内部负责电源管理的单元

ps:个人猜测pmic对应很多单元

# Linux中SD/MMC设备驱动流程

SD（Secure Digital）与 MMC（Multimedia Card）

SD 是一种 flash memory card 的标准，也就是一般常见的 SD 记忆卡，而 MMC 则是较早的一种记忆卡标准，目前已经被 SD 标准所取代。在维基百科上有相当详细的 SD/MMC 规格说明：

## SDIO（Secure Digital I/O）

SDIO 是目前我们比较关心的技术，SDIO 故名思义，就是 SD 的 I/O 接口（interface）的意思，不过这样解释可能还有点抽像。更具体的说明，SD 本来是记忆卡的标准，但是现在也可以把 SD 拿来插上一些外围接口使用，这样的技术便是 SDIO。

所以 SDIO 本身是一种相当单纯的技术，透过 SD 的 I/O 接脚来连接外部外围，并且透过 SD 上的 I/O 数据接位与这些外围传输数据，而且 SD 协会会员也推出很完整的 SDIO stack 驱动程序，使得 SDIO 外围（我们称为 SDIO 卡）的开发与应用变得相当热门。 现在已经有非常多的手机或是手持装置都支持 SDIO 的功能（SD 标准原本就是针对 mobile device 而制定），而且许多 SDIO 外围也都被开发出来，让手机外接外围更加容易，并且开发上更有弹性（不需要内建外围）。目前常见的 SDIO 外围（SDIO 卡）有：

 Wi-Fi card（无线网络卡）

 CMOS sensor card（照相模块）

   GPS card

 GSM/GPRS modem card

 Bluetooth card

 Radio/TV card（很好玩）

SDIO 的应用将是未来嵌入式系统最重要的接口技术之一，并且也会取代目前 GPIO 式的 SPI 接口。

## **SD/SDIO/MMC/SPI之间的关系**

SD/SDIO/MMC是三种不同的卡，这3种卡的管脚很类似，卡的样子也类似。最开始是先由MMC 卡，然后在MMC的基础上做了修改有了SD卡，SD卡的基础上有了SDIO卡。所以SD/SDIO/MMC的协 议大同小异，只有少量差异。 SPI在SD的大军里，只是一种传输模式。若是慢速的情况下，可选择SPI传输模式，这样SD接口 就变成SPI协议了。Host也要根据SPI的协议对卡进行操作。 具体的区别需参考各协议。

## **同一个控制器，支持SD/MMC/SDIO卡，如何区分是什么卡**

这个需要软件来区分。软件不停的循环访问卡，对卡按照一定的顺序发出对应格式的命令，那么不同的卡会做出不同的反应。几个CMD发完，根据不用的反应就可以分出是什么卡了。比如已经判断出是SD卡了，但是不知道是什么类型的SD卡。那么我们就发一个CMD8，如果 没有response，那么这个卡就是SD1.X的卡；如果有response，那么就是SD2.0以上的卡。

## SD/SDIO 的传输模式

SD 传输模式有以下 3 种：

 SPI mode（required）

 1-bit mode

 4-bit mode

SDIO 同样也支持以上 3 种传输模式。依据 SD 标准，所有的 SD（记忆卡）与 SDIO（外围）都必须支持 SPI mode，因此 SPI mode 是「required」。此外，早期的 MMC 卡（使用 SPI 传输）也能接到 SD 插糟（SD slot），并且使用 SPI mode 或 1-bit mode 来读取。 SD 的 MMC Mode SD 也能读取 MMC 内存，虽然 MMC 标准上提到，MMC 内存不见得要支持 SPI mode（但是一定要支持 1-bit mode），但是市面上能看到的 MMC 卡其实都有支持 SPI mode。因此，我们可以把 SD 设定成 SPI mode 的传输方式来读取 MMC 记忆卡。 SD 的 MMC Mode 就是用来读取 MMC 卡的一种传输模式。不过，SD 的 MMC Mode 虽然也是使用 SPI mode，但其物理特性仍是有差异的：  MMC 的 SPI mode 最大传输速率为 20 Mbit/s；  SD 的 SPI mode 最大传输速率为 25 Mbit/s。 为避免混淆，有时也用 SPI/MMC mode 与 SPI/SD mode 的写法来做清楚区别



## 参考资料

http://blog.csdn.net/evilcode/article/details/7408921

http://blog.csdn.net/evilcode/article/details/7418323

<http://wenku.baidu.com/link?url=lPg26KHW_IUD6de7wxooapFMfzJNXAhuyRabcnj0WpmlT23LpGG5KRwEXIrUFqe8UfIqL1_i3CMUIZYxrNNsOmz1Dsvv_7-foXWH0ZwZIAa>

# 内核通知链

大多数内核子系统都是相互独立的，因此某个子系统可能对其它子系统产生的事件感兴趣。为了满足这个需求，也即是让某个子系统在发生某个事件时通知其它的子系统，Linux内核提供了通知链的机制。通知链表**只能够在内核的子系统之间使用**，而不能够在内核与用户空间之间进行事件的通知。通知链表是一个**单链表的函数链表**，链表上的每一个节点都注册了一个函数。当某个事情发生时，链表上所有节点对应的函数就会被执行。所以对于通知链表来说有一个通知方与一个接收方。在通知这个事件时所运行的函数由被通知方决定，实际上也即是被通知方注册了某个函数，在发生某个事件时这些函数就得到执行。通知链技术可以概括为：事件的接收者将事件发生时应该执行的操作通过函数指针方式保存在链表中，然后当事件发生时通知者依次执行链表中每一个元素的回调函数。

## ****notifier chain 定义和接口****

1. **struct** notifier\_block {
2. **int** (\*notifier\_call)(**struct** notifier\_block \*, unsigned **long**, **void** \*);  // 回调函数接口
3. **struct** notifier\_block \*next;  // 指向下一个通知结构
4. **int** priority;                 // 当前通知链的优先级 ，一般默认为0。
5. };

可以看到通知链的基础数据结构比较简单，有回调函数接口、下一节点指针、优先级三个成员，其中回调函数的三个参数分别为：指向当前结构的指针、事件类型、参数。

内核提供了4种常用的通知链：

## ****1、Atomic notifier chains****

原子通知链：通知链元素的回调函数（当事件发生时要执行的函数）只能在中断上下文中运行，不允许阻塞

1. **struct** atomic\_notifier\_head {
2. spinlock\_t lock;
3. **struct** notifier\_block \*head;
4. };

由宏 ATOMIC\_NOTIFIER\_HEAD(name) 初始化链表头，其注册、注销及通知接口分别为：

1. **int** atomic\_notifier\_chain\_register(**struct** atomic\_notifier\_head \*nh, **struct** notifier\_block \*n);
2. **int** atomic\_notifier\_chain\_unregister(**struct** atomic\_notifier\_head \*nh, **struct** notifier\_block \*n);
3. **int** atomic\_notifier\_call\_chain(**struct** atomic\_notifier\_head \*nh, unsigned **long** val, **void** \*v);

## ****2、Blocking notifier chains****

可阻塞通知链：通知链元素的回调函数在进程上下文中运行，允许阻塞

1. **struct** blocking\_notifier\_head {
2. **struct** rw\_semaphore rwsem;
3. **struct** notifier\_block \*head;
4. };

由宏 BLOCKING\_NOTIFIER\_HEAD(name) 初始化链表头，其注册、注销及通知接口分别为：

1. **int** blocking\_notifier\_chain\_register(**struct** blocking\_notifier\_head \*nh, **struct** notifier\_block \*nb);
2. **int** blocking\_notifier\_chain\_unregister(**struct** blocking\_notifier\_head \*nh, **struct** notifier\_block \*nb);
3. **int** blocking\_notifier\_call\_chain(**struct** blocking\_notifier\_head \*nh, unsigned **long** val, **void** \*v);

## 3、Raw notifier chains

原始通知链：对通知链元素的回调函数没有任何限制，所有锁和保护机制都由调用者维护

1. **struct** raw\_notifier\_head {
2. **struct** notifier\_block \*head;
3. };

由宏 RAW\_NOTIFIER\_HEAD(name) 初始化链表头，其注册、注销及通知接口分别为：

1. **int** raw\_notifier\_chain\_register(**struct** raw\_notifier\_head \*nh, **struct** notifier\_block \*nb);
2. **int** raw\_notifier\_chain\_unregister(**struct** raw\_notifier\_head \*nh, **struct** notifier\_block \*nb);
3. **int** raw\_notifier\_call\_chain(**struct** raw\_notifier\_head \*nh, unsigned **long** val, **void** \*v);

## 4、SRCU notifier chains

可阻塞通知链的一种变体

1. **struct** srcu\_notifier\_head {
2. **struct** mutex mutex;
3. **struct** srcu\_struct srcu;
4. **struct** notifier\_block \*head;
5. };

该链表头必须动态申请 srcu\_init\_notifier\_head 和释放 srcu\_cleanup\_notifier\_head，其注册、注销及通知接口分别为：

1. **int** srcu\_notifier\_chain\_register(**struct** srcu\_notifier\_head \*nh, **struct** notifier\_block \*nb);
2. **int** srcu\_notifier\_chain\_unregister(**struct** srcu\_notifier\_head \*nh, **struct** notifier\_block \*nb);
3. **int** srcu\_notifier\_call\_chain(**struct** srcu\_notifier\_head \*nh, unsigned **long** val, **void** \*v);

## ****5、notifier\_call\_chain****

当有事件触发时，通知者调用 notifier\_call\_chain 函数通知事件的到达，这个函数会遍历nl指向的通知链中所有的元素，然后依次调用每一个的回调函数，完成通知动作。

1. **static** **int** \_\_kprobes notifier\_call\_chain(**struct** notifier\_block \*\*nl,
2. unsigned **long** val, **void** \*v,
3. **int** nr\_to\_call, **int** \*nr\_calls)

参数nl是通知链的头部，val表示事件类型，v用来指向通知链上的函数执行时需要用到的参数，一般不同的通知链，参数类型也不一样，例如当通知一个网卡被注册时，v就指向net\_device结构，nr\_to\_call表示准备最多通知几个，-1表示整条链都通知，nr\_calls非空的话，返回通知了多少个。每个被执行的 notifier\_block 回调函数的**返回值**可能取值为以下几个：  
NOTIFY\_DONE：表示对相关的事件类型不关心  
NOTIFY\_OK：顺利执行  
NOTIFY\_BAD：执行有错  
NOTIFY\_STOP：停止执行后面的回调函数  
NOTIFY\_STOP\_MASK：停止执行的掩码  
notifier\_call\_chain 把最后一个被调用的回调函数的返回值作为它的返回值。

## notifier chain 使用方法

在常见的环境中，我们通常会对 notifier chain 做一定的封装再使用。比如在电源管理子系统中，做了如下的定义和封装：

1. **static** BLOCKING\_NOTIFIER\_HEAD(pm\_chain\_head);         // 初始化链表头部
3. **int** register\_pm\_notifier(**struct** notifier\_block \*nb)   // 注册函数
4. {
5. **return** blocking\_notifier\_chain\_register(&pm\_chain\_head, nb);
6. }
7. EXPORT\_SYMBOL\_GPL(register\_pm\_notifier);
9. **int** unregister\_pm\_notifier(**struct** notifier\_block \*nb)  // 注销函数
10. {
11. **return** blocking\_notifier\_chain\_unregister(&pm\_chain\_head, nb);
12. }
13. EXPORT\_SYMBOL\_GPL(unregister\_pm\_notifier);
15. **int** pm\_notifier\_call\_chain(unsigned **long** val)          // 通知函数
16. {
17. **return** (blocking\_notifier\_call\_chain(&pm\_chain\_head, val, NULL)
18. == NOTIFY\_BAD) ? -EINVAL : 0;
19. }

**Ps :blocking\_notifier\_chain\_register一次性执行两遍会使原本为单链的call变成一个环，死循环**

## 参考资料

<http://blog.csdn.net/g_salamander/article/details/8081724>

http://blog.csdn.net/wuhzossibility/article/details/8079025

# Socket通信机制

## ****socket****套接字：

     socket起源于Unix，而Unix/[Linux](http://lib.csdn.net/base/linux)基本哲学之一就是“一切皆文件”，都可以用“打开open –> 读写write/read –> 关闭close”模式来操作。Socket就是该模式的一个实现，socket即是一种特殊的文件，一些socket函数就是对其进行的操作（读/写IO、打开、关闭）.  
     说白了Socket是应用层与TCP/IP协议族通信的中间软件抽象层，**它是一组接口**。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

       注意：其实socket也没有层的概念，它只是一个facade设计模式的应用，让编程变的更简单。是一个软件抽象层。在网络编程中，我们大量用的都是通过socket实现的。

## ****套接字描述符：****

     其实就是一个整数，我们最熟悉的句柄是0、1、2三个，0是标准输入，1是标准输出，2是标准错误输出。0、1、2是整数表示的，对应的FILE \*结构的表示就是stdin、stdout、stderr

套接字API最初是作为UNIX操作系统的一部分而开发的，所以套接字API与系统的其他I/O设备集成在一起。特别是，当应用程序要为因特网通信而创建一个套接字（socket）时，操作系统就返回一个小整数作为描述符（descriptor）来标识这个套接字。然后，应用程序以该描述符作为传递参数，通过调用函数来完成某种操作（例如通过网络传送数据或接收输入的数据）。

在许多操作系统中，套接字描述符和其他I/O描述符是集成在一起的，所以应用程序可以对文件进行套接字I/O或I/O读/写操作。

当应用程序要创建一个套接字时，操作系统就返回一个小整数作为描述符，应用程序则使用这个描述符来引用该套接字需要I/O请求的应用程序请求操作系统打开一个文件。操作系统就创建一个文件描述符提供给应用程序访问文件。从应用程序的角度看，文件描述符是一个整数，应用程序可以用它来读写文件。下图显示，操作系统如何把文件描述符实现为一个指针数组，这些指针指向内部[**数据结构**](http://lib.csdn.net/base/datastructure)。



     对于每个程序系统都有一张单独的表。精确地讲，系统为每个运行的进程维护一张单独的文件描述符表。当进程打开一个文件时，系统把一个指向此文件内部数据结构的指针写入文件描述符表，并把该表的索引值返回给调用者 。应用程序只需记住这个描述符，并在以后操作该文件时使用它。操作系统把该描述符作为索引访问进程描述符表，通过指针找到保存该文件所有的信息的数据结构。

**针对套接字的系统数据结构：**

   1）、套接字API里有个函数socket，它就是用来创建一个套接字。套接字设计的总体思路是，单个系统调用就可以创建任何套接字，因为套接字是相当笼统的。一旦套接字创建后，应用程序还需要调用其他函数来指定具体细节。例如调用socket将创建一个新的描述符条目：



   2）、虽然套接字的内部数据结构包含很多字段，但是系统创建套接字后，大多数字字段没有填写。应用程序创建套接字后在该套接字可以使用之前，必须调用其他的过程来填充这些字段。

## ****文件描述符和文件指针的区别：****

**文件描述符：**在linux系统中打开文件就会获得文件描述符，它是个很小的正整数。每个进程在PCB（Process Control Block）中保存着一份文件描述符表，文件描述符就是这个表的索引，每个表项都有一个指向已打开文件的指针。

**文件指针：**[**C语言**](http://lib.csdn.net/base/c)中使用文件指针做为I/O的句柄。文件指针指向进程用户区中的一个被称为FILE结构的数据结构。FILE结构包括一个缓冲区和一个文件描述符。而文件描述符是文件描述符表的一个索引，因此从某种意义上说文件指针就是句柄的句柄（在Windows系统上，文件描述符被称作文件句柄）。

详细内容请看[linux文件系统](http://blog.csdn.net/hguisu/article/details/6122513#t7)：<http://blog.csdn.net/hguisu/article/details/6122513#t7>

## 基本的SOCKET接口函数

在生活中，A要电话给B，A拨号，B听到电话铃声后提起电话，这时A和B就建立起了连接，A和B就可以讲话了。等交流结束，挂断电话结束此次交谈。  打电话很简单解释了这工作原理：“open—write/read—close”模式。



    服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，等待客户端连接。在这时如果有个客户端初始化一个Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

**这些接口的实现都是内核来完成。具体如何实现，可以看看linux的内核**

## socket()函数

int **socket**(int protofamily, int type, int protocol);//返回sockfd

  sockfd是描述符。

  socket函数对应于普通文件的打开操作。普通文件的打开操作返回一个文件描述字，而**socket()**用于创建一个socket描述符（socket descriptor），它唯一标识一个socket。这个socket描述字跟文件描述字一样，后续的操作都有用到它，把它作为参数，通过它来进行一些读写操作。

      正如可以给fopen的传入不同参数值，以打开不同的文件。创建socket的时候，也可以指定不同的参数创建不同的socket描述符，socket函数的三个参数分别为：

protofamily：即协议域，又称为协议族（family）。常用的协议族有，AF\_INET(IPV4)、AF\_INET6(IPV6)、AF\_LOCAL（或称AF\_UNIX，Unix域socket）、AF\_ROUTE等等。协议族决定了socket的地址类型，在通信中必须采用对应的地址，如AF\_INET决定了要用ipv4地址（32位的）与端口号（16位的）的组合、AF\_UNIX决定了要用一个绝对路径名作为地址。

type：指定socket类型。常用的socket类型有，SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等等（socket的类型有哪些？）。

protocol：故名思意，就是指定协议。常用的协议有，IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC等，它们分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议（这个协议我将会单独开篇讨论！）。

**注意**：并不是上面的type和protocol可以随意组合的，如SOCK\_STREAM不可以跟IPPROTO\_UDP组合。当protocol为0时，会自动选择type类型对应的默认协议。

当我们调用**socket**创建一个socket时，返回的socket描述字它存在于协议族（address family，AF\_XXX）空间中，但没有一个具体的地址。如果想要给它赋值一个地址，就必须调用bind()函数，否则就当调用connect()、listen()时系统会自动随机分配一个端口。

## bind()函数

正如上面所说bind()函数把一个地址族中的特定地址赋给socket。例如对应AF\_INET、AF\_INET6就是把一个ipv4或ipv6地址和端口号组合赋给socket。

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

函数的三个参数分别为：

sockfd：即socket描述字，它是通过socket()函数创建了，唯一标识一个socket。bind()函数就是将给这个描述字绑定一个名字。

addr：一个const struct sockaddr \*指针，指向要绑定给sockfd的协议地址。这个地址结构根据地址创建socket时的地址协议族的不同而不同，如ipv4对应的是：

struct sockaddr\_in {

sa\_family\_t sin\_family; /\* address family: AF\_INET \*/

in\_port\_t sin\_port; /\* port in network byte order \*/

struct in\_addr sin\_addr; /\* internet address \*/

};

/\* Internet address. \*/

struct in\_addr {

uint32\_t s\_addr; /\* address in network byte order \*/

};

ipv6对应的是：

struct sockaddr\_in6 {

sa\_family\_t sin6\_family; /\* AF\_INET6 \*/

in\_port\_t sin6\_port; /\* port number \*/

uint32\_t sin6\_flowinfo; /\* IPv6 flow information \*/

struct in6\_addr sin6\_addr; /\* IPv6 address \*/

uint32\_t sin6\_scope\_id; /\* Scope ID (new in 2.4) \*/

};

struct in6\_addr {

unsigned char s6\_addr[16]; /\* IPv6 address \*/

};

Unix域对应的是：

#define UNIX\_PATH\_MAX 108

struct sockaddr\_un {

sa\_family\_t sun\_family; /\* AF\_UNIX \*/

char sun\_path[UNIX\_PATH\_MAX]; /\* pathname \*/

};

addrlen：对应的是地址的长度。

通常服务器在启动的时候都会绑定一个众所周知的地址（如ip地址+端口号），用于提供服务，客户就可以通过它来接连服务器；而客户端就不用指定，有系统自动分配一个端口号和自身的ip地址组合。这就是为什么通常服务器端在listen之前会调用bind()，而客户端就不会调用，而是在connect()时由系统随机生成一个。

## 网络字节序与主机字节序

**主机字节序**就是我们平常说的大端和小端模式：不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，这个叫做主机序。引用标准的Big-Endian和Little-Endian的定义如下：

　　a) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

　　b) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

**网络字节序**：4个字节的32 bit值以下面的次序传输：首先是0～7bit，其次8～15bit，然后16～23bit，最后是24~31bit。这种传输次序称作大端字节序。**由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。**字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序，一个字节的数据没有顺序的问题了。

所以：在将一个地址绑定到socket的时候，请先将主机字节序转换成为网络字节序，而不要假定主机字节序跟网络字节序一样使用的是Big-Endian。由于这个问题曾引发过血案！公司项目代码中由于存在这个问题，导致了很多莫名其妙的问题，所以请谨记对主机字节序不要做任何假定，务必将其转化为网络字节序再赋给socket。

## listen()、connect()函数

如果作为一个服务器，在调用socket()、bind()之后就会调用listen()来监听这个socket，如果客户端这时调用connect()发出连接请求，服务器端就会接收到这个请求。

int listen(int sockfd, int backlog);

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

listen函数的第一个参数即为要监听的socket描述字，第二个参数为相应socket可以排队的最大连接个数。socket()函数创建的socket默认是一个主动类型的，listen函数将socket变为被动类型的，等待客户的连接请求。

connect函数的第一个参数即为客户端的socket描述字，第二参数为服务器的socket地址，第三个参数为socket地址的长度。客户端通过调用connect函数来建立与TCP服务器的连接。

## accept()函数

TCP服务器端依次调用socket()、bind()、listen()之后，就会监听指定的socket地址了。TCP客户端依次调用socket()、connect()之后就向TCP服务器发送了一个连接请求。TCP服务器监听到这个请求之后，就会调用accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络I/O操作了，即类同于普通文件的读写I/O操作。

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen); //返回连接connect\_fd

参数sockfd

参数sockfd就是上面解释中的监听套接字，这个套接字用来监听一个端口，当有一个客户与服务器连接时，它使用这个一个端口号，而此时这个端口号正与这个套接字关联。当然客户不知道套接字这些细节，它只知道一个地址和一个端口号。

参数addr

这是一个结果参数，它用来接受一个返回值，这返回值指定客户端的地址，当然这个地址是通过某个地址结构来描述的，用户应该知道这一个什么样的地址结构。如果对客户的地址不感兴趣，那么可以把这个值设置为NULL。

参数len

如同大家所认为的，它也是结果的参数，用来接受上述addr的结构的大小的，它指明addr结构所占有的字节个数。同样的，它也可以被设置为NULL。

如果accept成功返回，则服务器与客户已经正确建立连接了，此时服务器通过accept返回的套接字来完成与客户的通信。

注意：

      accept默认会阻塞进程，直到有一个客户连接建立后返回，它返回的是一个新可用的套接字，这个套接字是连接套接字。

此时我们需要区分两种套接字，

       监听套接字: 监听套接字正如accept的参数sockfd，它是监听套接字，在调用listen函数之后，是服务器开始调用socket()函数生成的，称为监听socket描述字(监听套接字)

       连接套接字：一个套接字会从主动连接的套接字变身为一个监听套接字；而accept函数返回的是已连接socket描述字(一个连接套接字)，它代表着一个网络已经存在的点点连接。

        一个服务器通常通常仅仅只创建一个监听socket描述字，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接socket描述字，当服务器完成了对某个客户的服务，相应的已连接socket描述字就被关闭。

        自然要问的是：为什么要有两种套接字？原因很简单，如果使用一个描述字的话，那么它的功能太多，使得使用很不直观，同时在内核确实产生了一个这样的新的描述字。

连接套接字socketfd\_new 并没有占用新的端口与客户端通信，依然使用的是与监听套接字socketfd一样的端口号

## read()、write()等函数

万事具备只欠东风，至此服务器与客户已经建立好连接了。可以调用网络I/O进行读写操作了，即实现了网咯中不同进程之间的通信！网络I/O操作有下面几组：

read()/write()

recv()/send()

readv()/writev()

recvmsg()/sendmsg()

recvfrom()/sendto()

我推荐使用recvmsg()/sendmsg()函数，这两个函数是最通用的I/O函数，实际上可以把上面的其它函数都替换成这两个函数。它们的声明如下：

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int fd, const void \*buf, size\_t count);

#include <sys/types.h>

#include <sys/socket.h>

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags,

const struct sockaddr \*dest\_addr, socklen\_t addrlen);

ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags,

struct sockaddr \*src\_addr, socklen\_t \*addrlen);

ssize\_t sendmsg(int sockfd, const struct msghdr \*msg, int flags);

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

read函数是负责从fd中读取内容.当读成功时，read返回实际所读的字节数，如果返回的值是0表示已经读到文件的结束了，小于0表示出现了错误。如果错误为EINTR说明读是由中断引起的，如果是ECONNREST表示网络连接出了问题。

write函数将buf中的nbytes字节内容写入文件描述符fd.成功时返回写的字节数。失败时返回-1，并设置errno变量。 在网络程序中，当我们向套接字文件描述符写时有俩种可能。1)write的返回值大于0，表示写了部分或者是全部的数据。2)返回的值小于0，此时出现了错误。我们要根据错误类型来处理。如果错误为EINTR表示在写的时候出现了中断错误。如果为EPIPE表示网络连接出现了问题(对方已经关闭了连接)。

其它的我就不一一介绍这几对I/O函数了，具体参见man文档或者baidu、Google，下面的例子中将使用到send/recv。

## close()函数

在服务器与客户端建立连接之后，会进行一些读写操作，完成了读写操作就要关闭相应的socket描述字，好比操作完打开的文件要调用fclose关闭打开的文件。

#include <unistd.h>

int close(int fd);

close一个TCP socket的缺省行为时把该socket标记为以关闭，然后立即返回到调用进程。该描述字不能再由调用进程使用，也就是说不能再作为read或write的第一个参数。

注意：close操作只是使相应socket描述字的引用计数-1，只有当引用计数为0的时候，才会触发TCP客户端向服务器发送终止连接请求。

## Netlink机制及其关键技术

### 3.1 Netlink机制

Linux操作系统中当CPU处于内核状态时，可以分为有用户上下文的状态和执行硬件、软件中断两种。其中当处于有用户上下文时，由于内核态和用户态的内存映射机制不同，不可直接将本地变量传给用户态的内存区；处于硬件、软件中断时，无法直接向用户内存区传递数据，代码执行不可中断。针对传统的进程间通信机制，他们均无法直接在内核态和用户态之间使用，原因如下表：

|  |  |
| --- | --- |
| 通信方法 | 无法介于内核态与用户态的原因 |
| 管道（不包括命名管道） | 局限于父子进程间的通信。 |
| 消息队列 | 在硬、软中断中无法无阻塞地接收数据。 |
| 信号量 | 无法介于内核态和用户态使用。 |
| 内存共享 | 需要信号量辅助，而信号量又无法使用。 |
| 套接字 | 在硬、软中断中无法无阻塞地接收数据。 |

    解决内核态和用户态通信机制可分为两类：

1. 处于有用户上下文时，可以使用Linux提供的copy\_from\_user()和copy\_to\_user()函数完成，但由于这两个函数可能阻塞，因此不能在硬件、软件的中断过程中使用。
2. 处于硬、软件中断时。

2.1   可以通过Linux内核提供的spinlock自旋锁实现内核线程与中断过程的同步，由于内核线程运行在有上下文的进程中，因此可以在内核线程中使用套接字或消息队列来取得用户空间的数据，然后再将数据通过临界区传递给中断过程.

2.2   通过Netlink机制实现。Netlink 套接字的通信依据是一个对应于进程的标识，一般定为该进程的 ID。Netlink通信最大的特点是对对中断过程的支持，它在内核空间接收用户空间数据时不再需要用户自行启动一个内核线程，而是通过另一个软中断调用用户事先指定的接收函数。通过软中断而不是自行启动内核线程保证了数据传输的及时性。

### 3.2 Netlink优点

Netlink相对于其他的通信机制具有以下优点：

1. 使用Netlink通过自定义一种新的协议并加入协议族即可通过socket API使用Netlink协议完成数据交换，而ioctl和proc文件系统均需要通过程序加入相应的设备或文件。[ioctl与proc都需要依赖与文件的创建]
2. Netlink使用socket缓存队列，是一种异步通信机制，而ioctl是同步通信机制，如果传输的数据量较大，会影响系统性能。
3. Netlink支持多播，属于一个Netlink组的模块和进程都能获得该多播消息。
4. Netlink允许内核发起会话，而ioctl和系统调用只能由用户空间进程发起

## 建立Netlink会话过程如下：



内核使用与标准socket API类似的一套API完成通信过程。首先通过netlink\_kernel\_create()创建套接字，该函数的原型如下：

struct sock \*netlink\_kernel\_create(struct net \*net,  
int unit,unsigned int groups,  
void (\*input)(struct sk\_buff \*skb),  
struct mutex \*cb\_mutex,  
struct module \*module);

其中net参数是网络设备命名空间指针，input函数是netlink socket在接受到消息时调用的回调函数指针，module默认为THIS\_MODULE.

然后用户空间进程使用标准Socket API来创建套接字，将进程ID发送至内核空间，用户空间创建使用socket()创建套接字，该函数的原型如下：

int socket(int domain, int type, int protocol);

其中domain值为PF\_NETLINK，即Netlink使用协议族。protocol为Netlink提供的协议或者是用户自定义的协议，Netlink提供的协议包括NETLINK\_ROUTE, NETLINK\_FIREWALL, NETLINK\_ARPD, NETLINK\_ROUTE6和 NETLINK\_IP6\_FW。

接着使用bind函数绑定。Netlink的bind()函数把一个本地socket地址(源socket地址)与一个打开的socket进行关联。完成绑定，内核空间接收到用户进程ID之后便可以进行通讯。

用户空间进程发送数据使用标准socket API中sendmsg()函数完成，使用时需添加struct msghdr消息和nlmsghdr消息头。一个netlink消息体由nlmsghdr和消息的payload部分组成，输入消息后，内核会进入nlmsghdr指向的缓冲区。

内核空间发送数据使用独立创建的sk\_buff缓冲区，Linux定义了如下宏方便对于缓冲区地址的设置，如下所示：

#define NETLINK\_CB(skb) (\*(struct netlink\_skb\_parms\*)&((skb)->cb))

在对缓冲区设置完成消息地址之后，可以使用netlink\_unicast()来发布单播消息，netlink\_unicast()原型如下：

int netlink\_unicast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, int nonblock);

参数sk为函数netlink\_kernel\_create()返回的socket，参数skb存放消息，它的data字段指向要发送的netlink消息结构，而skb的控制块保存了消息的地址信息，前面的宏NETLINK\_CB(skb)就用于方便设置该控制块，参数pid为接收消息进程的pid，参数nonblock表示该函数是否为非阻塞，如果为1，该函数将在没有接收缓存可利用时立即返回，而如果为0，该函数在没有接收缓存可利用时睡眠。

内核模块或子系统也可以使用函数netlink\_broadcast来发送广播消息：

void netlink\_broadcast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, u32 group, int allocation);

前面的三个参数与netlink\_unicast相同，参数group为接收消息的多播组，该参数的每一个代表一个多播组，因此如果发送给多个多播组，就把该参数设置为多个多播组组ID的位或。参数allocation为内核内存分配类型，一般地为GFP\_ATOMIC或GFP\_KERNEL，GFP\_ATOMIC用于原子的上下文（即不可以睡眠），而GFP\_KERNEL用于非原子上下文。

接收数据时程序需要申请足够大的空间来存储netlink消息头和消息的payload部分。然后使用标准函数接口recvmsg()来接收netlink消息

### 其他相关说明

    Netlink 是一种特殊的 socket，它是 Linux 所特有的，类似于 BSD 中的AF\_ROUTE 但又远比它的功能强大，目前在最新的 Linux 内核（2.6.14）中使用netlink 进行应用与内核通信的应用很多，包括：路由 daemon（NETLINK\_ROUTE），1-wire 子系统（NETLINK\_W1），用户态 socket 协议（NETLINK\_USERSOCK），防火墙（NETLINK\_FIREWALL），socket 监视（NETLINK\_INET\_DIAG），netfilter 日志（NETLINK\_NFLOG），ipsec 安全策略（NETLINK\_XFRM），SELinux 事件通知（NETLINK\_SELINUX），iSCSI 子系统（NETLINK\_ISCSI），进程审计（NETLINK\_AUDIT），转发信息表查询（NETLINK\_FIB\_LOOKUP），netlink connector(NETLINK\_CONNECTOR),netfilter 子系统（NETLINK\_NETFILTER），IPv6 防火墙（NETLINK\_IP6\_FW），DECnet 路由信息（NETLINK\_DNRTMSG），内核事件向用户态通知（NETLINK\_KOBJECT\_UEVENT），通用 netlink（NETLINK\_GENERIC）。

    Netlink 是一种在内核与用户应用间进行双向数据传输的非常好的方式，用户态应用使用标准的 socket API 就可以使用 netlink 提供的强大功能，内核态需要使用专门的内核 API 来使用 netlink。

Netlink 相对于系统调用，ioctl 以及 /proc 文件系统而言具有以下优点：

    1，为了使用 netlink，用户仅需要在 include/linux/netlink.h 中增加一个新类型的 netlink 协议定义即可， 如 #define NETLINK\_MYTEST 17 然后，内核和用户态应用就可以立即通过 socket API 使用该 netlink 协议类型进行数据交换。但系统调用需要增加新的系统调用，ioctl 则需要增加设备或文件， 那需要不少代码，proc 文件系统则需要在 /proc 下添加新的文件或目录，那将使本来就混乱的 /proc 更加混乱。

    2. netlink是一种异步通信机制，在内核与用户态应用之间传递的消息保存在socket缓存队列中，发送消息只是把消息保存在接收者的socket的接收队列，而不需要等待接收者收到消息，但系统调用与 ioctl 则是同步通信机制，如果传递的数据太长，将影响调度粒度。

    3．使用 netlink 的内核部分可以采用模块的方式实现，使用 netlink 的应用部分和内核部分没有编译时依赖，但系统调用就有依赖，而且新的系统调用的实现必须静态地连接到内核中，它无法在模块中实现，使用新系统调用的应用在编译时需要依赖内核。

    4．netlink 支持多播，内核模块或应用可以把消息多播给一个netlink组，属于该neilink 组的任何内核模块或应用都能接收到该消息，内核事件向用户态的通知机制就使用了这一特性，任何对内核事件感兴趣的应用都能收到该子系统发送的内核事件，在后面的文章中将介绍这一机制的使用。

    5．内核可以使用 netlink 首先发起会话，但系统调用和 ioctl 只能由用户应用发起调用。

6．netlink 使用标准的 socket API，因此很容易使用，但系统调用和 ioctl则需要专门的培训才能使用。

**用户态使用 netlink**

   用户态应用使用标准的socket APIs， socket(), bind(), sendmsg(), recvmsg() 和 close() 就能很容易地使用 netlink socket，查询手册页可以了解这些函数的使用细节，本文只是讲解使用 netlink 的用户应该如何使用这些函数。注意，使用 netlink 的应用必须包含头文件 linux/netlink.h。当然 socket 需要的头文件也必不可少，sys/socket.h。

   为了创建一个 netlink socket，用户需要使用如下参数调用 socket():

|  |
| --- |
| socket(AF\_NETLINK, SOCK\_RAW, netlink\_type) |

   第一个参数必须是 AF\_NETLINK 或 PF\_NETLINK，在 Linux 中，它们俩实际为一个东西，它表示要使用netlink，第二个参数必须是SOCK\_RAW或SOCK\_DGRAM，第三个参数指定netlink协议类型，如前面讲的用户自定义协议类型NETLINK\_MYTEST， NETLINK\_GENERIC是一个通用的协议类型，它是专门为用户使用的，因此，用户可以直接使用它，而不必再添加新的协议类型。内核预定义的协议类型有：

#define NETLINK\_ROUTE 0  
#define NETLINK\_W1 1  
#define NETLINK\_USERSOCK 2   
#define NETLINK\_FIREWALL 3  
#define NETLINK\_INET\_DIAG 4  
#define NETLINK\_NFLOG 5  
#define NETLINK\_XFRM 6   
#define NETLINK\_SELINUX 7   
#define NETLINK\_ISCSI 8   
#define NETLINK\_AUDIT 9   
#define NETLINK\_FIB\_LOOKUP 10   
#define NETLINK\_CONNECTOR 11   
#define NETLINK\_NETFILTER 12   
#define NETLINK\_IP6\_FW 13   
#define NETLINK\_DNRTMSG 14   
#define NETLINK\_KOBJECT\_UEVENT 15   
#define NETLINK\_GENERIC 16

   对于每一个netlink协议类型，可以有多达 32多播组，每一个多播组用一个位表示，netlink 的多播特性使得发送消息给同一个组仅需要一次系统调用，因而对于需要多拨消息的应用而言，大大地降低了系统调用的次数。

   函数 bind() 用于把一个打开的 netlink socket 与 netlink 源 socket 地址绑定在一起。netlink socket 的地址结构如下：

struct sockaddr\_nl {  
sa\_family\_t nl\_family;  
unsigned short nl\_pad;  
\_\_u32 nl\_pid;  
\_\_u32 nl\_groups;  
};

   字段 nl\_family 必须设置为 AF\_NETLINK 或着 PF\_NETLINK，字段 nl\_pad 当前没有使用，因此要总是设置为 0，字段 nl\_pid 为接收或发送消息的进程的 ID，如果希望内核处理消息或多播消息，就把该字段设置为 0，否则设置为处理消息的进程 ID。字段 nl\_groups 用于指定多播组，bind 函数用于把调用进程加入到该字段指定的多播组，如果设置为 0，表示调用者不加入任何多播组。

   传递给 bind 函数的地址的 nl\_pid 字段应当设置为本进程的进程 ID，这相当于 netlink socket 的本地地址。但是，对于一个进程的多个线程使用 netlink socket 的情况，字段 nl\_pid 则可以设置为其它的值，如：

|  |
| --- |
| pthread\_self() << 16 | getpid(); |

   因此字段 nl\_pid 实际上未必是进程 ID,它只是用于区分不同的接收者或发送者的一个标识，用户可以根据自己需要设置该字段。函数 bind 的调用方式如下：

|  |
| --- |
| bind(fd, (struct sockaddr\*)&nladdr, sizeof(struct sockaddr\_nl)); |

   fd为前面的 socket 调用返回的文件描述符，参数 nladdr 为 struct sockaddr\_nl 类型的地址。为了发送一个 netlink 消息给内核或其他用户态应用，需要填充目标 netlink socket 地址，此时，字段 nl\_pid 和 nl\_groups 分别表示接收消息者的进程 ID 与多播组。如果字段 nl\_pid 设置为 0，表示消息接收者为内核或多播组，如果 nl\_groups为 0，表示该消息为单播消息，否则表示多播消息。使用函数 sendmsg 发送 netlink 消息时还需要引用结构 struct msghdr、struct nlmsghdr 和 struct iovec，结构 struct msghdr 需如下设置：

struct msghdr msg;  
memset(&msg, 0, sizeof(msg));  
msg.msg\_name = (void \*)&(nladdr);  
msg.msg\_namelen = sizeof(nladdr);

其中 nladdr 为消息接收者的 netlink 地址。

   struct nlmsghdr 为 netlink socket 自己的消息头，这用于多路复用和多路分解 netlink 定义的所有协议类型以及其它一些控制，netlink 的内核实现将利用这个消息头来多路复用和多路分解已经其它的一些控制，因此它也被称为netlink 控制块。因此，应用在发送 netlink 消息时必须提供该消息头。

struct nlmsghdr {  
\_\_u32 nlmsg\_len;  
\_\_u16 nlmsg\_type;  
\_\_u16 nlmsg\_flags;  
\_\_u32 nlmsg\_seq;  
\_\_u32 nlmsg\_pid;  
};

字段 nlmsg\_len 指定消息的总长度，包括紧跟该结构的数据部分长度以及该结构的大小，字段 nlmsg\_type 用于应用内部定义消息的类型，它对 netlink 内核实现是透明的，因此大部分情况下设置为 0，字段 nlmsg\_flags 用于设置消息标志，可用的标志包括：

#define NLM\_F\_REQUEST 1   
#define NLM\_F\_MULTI 2   
#define NLM\_F\_ACK 4   
#define NLM\_F\_ECHO 8   
#define NLM\_F\_ROOT 0x100   
#define NLM\_F\_MATCH 0x200   
#define NLM\_F\_ATOMIC 0x400   
#define NLM\_F\_DUMP (NLM\_F\_ROOT|NLM\_F\_MATCH)   
#define NLM\_F\_REPLACE 0x100   
#define NLM\_F\_EXCL 0x200   
#define NLM\_F\_CREATE 0x400   
#define NLM\_F\_APPEND 0x800

标志NLM\_F\_REQUEST用于表示消息是一个请求，所有应用首先发起的消息都应设置该标志。

标志NLM\_F\_MULTI 用于指示该消息是一个多部分消息的一部分，后续的消息可以通过宏NLMSG\_NEXT来获得。

宏NLM\_F\_ACK表示该消息是前一个请求消息的响应，顺序号与进程ID可以把请求与响应关联起来。

标志NLM\_F\_ECHO表示该消息是相关的一个包的回传。

标志NLM\_F\_ROOT 被许多 netlink 协议的各种数据获取操作使用，该标志指示被请求的数据表应当整体返回用户应用，而不是一个条目一个条目地返回。有该标志的请求通常导致响应消息设置NLM\_F\_MULTI标志。注意，当设置了该标志时，请求是协议特定的，因此，需要在字段 nlmsg\_type 中指定协议类型。

标志 NLM\_F\_MATCH 表示该协议特定的请求只需要一个数据子集，数据子集由指定的协议特定的过滤器来匹配。

标志 NLM\_F\_ATOMIC 指示请求返回的数据应当原子地收集，这预防数据在获取期间被修改。

标志 NLM\_F\_DUMP 未实现。

标志 NLM\_F\_REPLACE 用于取代在数据表中的现有条目。

标志 NLM\_F\_EXCL\_ 用于和 CREATE 和 APPEND 配合使用，如果条目已经存在，将失败。

标志 NLM\_F\_CREATE 指示应当在指定的表中创建一个条目。

标志 NLM\_F\_APPEND 指示在表末尾添加新的条目。

内核需要读取和修改这些标志，对于一般的使用，用户把它设置为 0 就可以，只是一些高级应用（如 netfilter 和路由 daemon 需要它进行一些复杂的操作），字段 nlmsg\_seq 和 nlmsg\_pid 用于应用追踪消息，前者表示顺序号，后者为消息来源进程 ID。下面是一个示例：

#define MAX\_MSGSIZE 1024  
char buffer[] = "An example message";  
struct nlmsghdr nlhdr;  
nlhdr = (struct nlmsghdr \*)malloc(NLMSG\_SPACE(MAX\_MSGSIZE));  
strcpy(NLMSG\_DATA(nlhdr),buffer);  
nlhdr->nlmsg\_len = NLMSG\_LENGTH(strlen(buffer));  
nlhdr->nlmsg\_pid = getpid();  
nlhdr->nlmsg\_flags = 0;

结构 struct iovec 用于把多个消息通过一次系统调用来发送，下面是该结构使用示例：

struct iovec iov;  
iov.iov\_base = (void \*)nlhdr;  
iov.iov\_len = nlh->nlmsg\_len;  
msg.msg\_iov = &iov;  
msg.msg\_iovlen = 1;

在完成以上步骤后，消息就可以通过下面语句直接发送：

|  |
| --- |
| sendmsg(fd, &msg, 0); |

应用接收消息时需要首先分配一个足够大的缓存来保存消息头以及消息的数据部分，然后填充消息头，添完后就可以直接调用函数 recvmsg() 来接收。

#define MAX\_NL\_MSG\_LEN 1024   
struct sockaddr\_nl nladdr;  
struct msghdr msg;  
struct iovec iov;  
struct nlmsghdr \* nlhdr;  
nlhdr = (struct nlmsghdr \*)malloc(MAX\_NL\_MSG\_LEN);  
iov.iov\_base = (void \*)nlhdr;  
iov.iov\_len = MAX\_NL\_MSG\_LEN;  
msg.msg\_name = (void \*)&(nladdr);  
msg.msg\_namelen = sizeof(nladdr);  
msg.msg\_iov = &iov;  
msg.msg\_iovlen = 1;  
recvmsg(fd, &msg, 0);

注意：fd为socket调用打开的netlink socket描述符。

在消息接收后，nlhdr指向接收到的消息的消息头，nladdr保存了接收到的消息的目标地址，宏NLMSG\_DATA(nlhdr)返回指向消息的数据部分的指针。

在linux/netlink.h中定义了一些方便对消息进行处理的宏，这些宏包括：

|  |
| --- |
| #define NLMSG\_ALIGNTO 4  #define NLMSG\_ALIGN(len) ( ((len)+NLMSG\_ALIGNTO-1) & ~(NLMSG\_ALIGNTO-1) ) |

宏NLMSG\_ALIGN(len)用于得到不小于len且字节对齐的最小数值。

|  |
| --- |
| #define NLMSG\_LENGTH(len) ((len)+NLMSG\_ALIGN(sizeof(struct nlmsghdr))) |

宏NLMSG\_LENGTH(len)用于计算数据部分长度为len时实际的消息长度。它一般用于分配消息缓存。

|  |
| --- |
| #define NLMSG\_SPACE(len) NLMSG\_ALIGN(NLMSG\_LENGTH(len)) |

宏NLMSG\_SPACE(len)返回不小于NLMSG\_LENGTH(len)且字节对齐的最小数值，它也用于分配消息缓存。

|  |
| --- |
| #define NLMSG\_DATA(nlh) ((void\*)(((char\*)nlh) + NLMSG\_LENGTH(0))) |

宏NLMSG\_DATA(nlh)用于取得消息的数据部分的首地址，设置和读取消息数据部分时需要使用该宏。

|  |
| --- |
| #define NLMSG\_NEXT(nlh,len) ((len) -= NLMSG\_ALIGN((nlh)->nlmsg\_len), \  (struct nlmsghdr\*)(((char\*)(nlh)) + NLMSG\_ALIGN((nlh)->nlmsg\_len))) |

宏NLMSG\_NEXT(nlh,len)用于得到下一个消息的首地址，同时len也减少为剩余消息的总长度，该宏一般在一个消息被分成几个部分发送或接收时使用。

|  |
| --- |
| #define NLMSG\_OK(nlh,len) ((len) >= (int)sizeof(struct nlmsghdr) && \  (nlh)->nlmsg\_len >= sizeof(struct nlmsghdr) && \  (nlh)->nlmsg\_len <= (len)) |

宏NLMSG\_OK(nlh,len)用于判断消息是否有len这么长。

|  |
| --- |
| #define NLMSG\_PAYLOAD(nlh,len) ((nlh)->nlmsg\_len - NLMSG\_SPACE((len))) |

宏NLMSG\_PAYLOAD(nlh,len)用于返回payload的长度。

函数close用于关闭打开的netlink socket。

**netlink内核API**

netlink的内核实现在.c文件net/core/af\_netlink.c中，内核模块要想使用netlink，也必须包含头文件linux/netlink.h。内核使用netlink需要专门的API，这完全不同于用户态应用对netlink的使用。如果用户需要增加新的netlink协议类型，必须通过修改linux/netlink.h来实现，当然，目前的netlink实现已经包含了一个通用的协议类型NETLINK\_GENERIC以方便用户使用，用户可以直接使用它而不必增加新的协议类型。前面讲到，为了增加新的netlink协议类型，用户仅需增加如下定义到linux/netlink.h就可以：

|  |
| --- |
| #define NETLINK\_MYTEST 17 |

只要增加这个定义之后，用户就可以在内核的任何地方引用该协议。

在内核中，为了创建一个netlink socket用户需要调用如下函数：

|  |
| --- |
| struct sock \* netlink\_kernel\_create(int unit, void (\*input)(struct sock \*sk, int len)); |

参数unit表示netlink协议类型，如NETLINK\_MYTEST，参数input则为内核模块定义的netlink消息处理函数，当有消息到达这个netlink socket时，该input函数指针就会被引用。函数指针input的参数sk实际上就是函数netlink\_kernel\_create返回的struct sock指针，sock实际是socket的一个内核表示数据结构，用户态应用创建的socket在内核中也会有一个struct sock结构来表示。下面是一个input函数的示例：

|  |
| --- |
| void input (struct sock \*sk, int len)  {  struct sk\_buff \*skb;  struct nlmsghdr \*nlh = NULL;  u8 \*data = NULL;  while ((skb = skb\_dequeue(&sk->receive\_queue)) != NULL)  {  nlh = (struct nlmsghdr \*)skb->data;  data = NLMSG\_DATA(nlh);  }  } |

函数input()会在发送进程执行sendmsg()时被调用，这样处理消息比较及时，但是，如果消息特别长时，这样处理将增加系统调用sendmsg()的执行时间，对于这种情况，可以定义一个内核线程专门负责消息接收，而函数input的工作只是唤醒该内核线程，这样sendmsg将很快返回。

函数skb = skb\_dequeue(&sk->receive\_queue)用于取得socket sk的接收队列上的消息，返回为一个struct sk\_buff的结构，skb->data指向实际的netlink消息。

函数skb\_recv\_datagram(nl\_sk)也用于在netlink socket nl\_sk上接收消息，与skb\_dequeue的不同指出是，如果socket的接收队列上没有消息，它将导致调用进程睡眠在等待队列nl\_sk->sk\_sleep，因此它必须在进程上下文使用，刚才讲的内核线程就可以采用这种方式来接收消息。

下面的函数input就是这种使用的示例：

|  |
| --- |
| void input (struct sock \*sk, int len) { wake\_up\_interruptible(sk->sk\_sleep); } |

当内核中发送netlink消息时，也需要设置目标地址与源地址，而且内核中消息是通过struct sk\_buff来管理的， linux/netlink.h中定义了一个宏：

|  |
| --- |
| #define NETLINK\_CB(skb) (\*(struct netlink\_skb\_parms\*)&((skb)->cb)) |

来方便消息的地址设置。下面是一个消息地址设置的例子：

|  |
| --- |
| NETLINK\_CB(skb).pid = 0; NETLINK\_CB(skb).dst\_pid = 0; NETLINK\_CB(skb).dst\_group = 1; |

字段pid表示消息发送者进程ID，也即源地址，对于内核，它为 0， dst\_pid 表示消息接收者进程 ID，也即目标地址，如果目标为组或内核，它设置为 0，否则 dst\_group 表示目标组地址，如果它目标为某一进程或内核，dst\_group 应当设置为 0。

在内核中，模块调用函数 netlink\_unicast 来发送单播消息：

|  |
| --- |
| int netlink\_unicast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, int nonblock); |

参数sk为函数netlink\_kernel\_create()返回的socket，参数skb存放消息，它的data字段指向要发送的netlink消息结构，而skb的控制块保存了消息的地址信息，前面的宏NETLINK\_CB(skb)就用于方便设置该控制块，参数pid为接收消息进程的pid，参数nonblock表示该函数是否为非阻塞，如果为1，该函数将在没有接收缓存可利用时立即返回，而如果为0，该函数在没有接收缓存可利用时睡眠。

内核模块或子系统也可以使用函数netlink\_broadcast来发送广播消息：

|  |
| --- |
| void netlink\_broadcast(struct sock \*sk, struct sk\_buff \*skb, u32 pid, u32 group, int allocation); |

前面的三个参数与netlink\_unicast相同，参数group为接收消息的多播组，该参数的每一个代表一个多播组，因此如果发送给多个多播组，就把该参数设置为多个多播组组ID的位或。参数allocation为内核内存分配类型，一般地为GFP\_ATOMIC或GFP\_KERNEL，GFP\_ATOMIC用于原子的上下文（即不可以睡眠），而GFP\_KERNEL用于非原子上下文。

在内核中使用函数sock\_release来释放函数netlink\_kernel\_create()创建的netlink socket：

|  |
| --- |
| void sock\_release(struct socket \* sock); |

注意函数netlink\_kernel\_create()返回的类型为struct sock，因此函数sock\_release应该这种调用：

|  |
| --- |
| sock\_release(sk->sk\_socket); |

sk为函数netlink\_kernel\_create()的返回值。

sk为函数netlink\_kernel\_create()的返回值。在源代码包中给出了一个使用 netlink 的示例，它包括一个内核模块 netlink-exam-kern.c 和两个应用程序 netlink-exam-user-recv.c, netlink-exam-user-send.c。内核模块必须先插入到内核，然后在一个终端上运行用户态接收程序，在另一个终端上运行用户态发送程序，发送程序读取参数指定的文本文件并把它作为 netlink 消息的内容发送给内核模块，内核模块接受该消息保存到内核缓存中，它也通过proc接口出口到 procfs，因此用户也能够通过 /proc/netlink\_exam\_buffer 看到全部的内容，同时内核也把该消息发送给用户态接收程序，用户态接收程序将把接收到的内容输出到屏幕上。

## 参考资料

<http://blog.chinaunix.net/uid-23069658-id-3405954.html>

<http://blog.sina.com.cn/s/blog_62ec291601010qgh.html>

<http://blog.chinaunix.net/uid-20788470-id-1841640.html>

<http://blog.csdn.net/hguisu/article/details/7445768/>

# Linux工作队列

1. 什么是workqueue       Linux中的Workqueue机制就是为了简化内核线程的创建。通过调用workqueue的接口就能创建内核线程。并且可以根据当前系统CPU的个数创建线程的数量，使得线程处理的事务能够并行化。workqueue是内核中实现简单而有效的机制，他显然简化了内核daemon的创建，方便了用户的编程.

      工作队列**（workqueue）**是另外一种将工作推后执行的形式.工作队列可以把工作推后，交由一个内核线程去执行，也就是说，这个下半部分可以在进程上下文中执行。最重要的就是工作队列允许被重新调度甚至是睡眠。

**workqueue与tasklet类似，都是允许内核代码请求某个函数在将来的时间被调用（抄《ldd3》上的）每个workqueue就是一个内核进程。**

2. 数据结构     我们把推后执行的任务叫做工作（work），描述它的数据结构为work\_struct:

1. **struct work\_struct {**
2. atomic\_long\_t data;       /\*工作处理函数func的参数\*/
3. #define WORK\_STRUCT\_PENDING 0        /\* T if work item pending execution \*/
4. #define WORK\_STRUCT\_STATIC 1        /\* static initializer (debugobjects) \*/
5. #define WORK\_STRUCT\_FLAG\_MASK (3UL)
6. #define WORK\_STRUCT\_WQ\_DATA\_MASK (~WORK\_STRUCT\_FLAG\_MASK)
7. **struct list\_head entry;        /\*连接工作的指针\*/**
8. work\_func\_t func;              /\*工作处理函数\*/
9. #ifdef CONFIG\_LOCKDEP
10. **struct lockdep\_map lockdep\_map;**
11. #endif
12. };

      这些工作以队列结构组织成工作队列（workqueue），其数据结构为workqueue\_struct:

1. **struct workqueue\_struct {**
2. **struct cpu\_workqueue\_struct \*cpu\_wq;**
3. **struct list\_head list;**
4. **const char \*name;   /\*workqueue name\*/**
5. int singlethread;   /\*是不是单线程 - 单线程我们首选第一个CPU -0表示采用默认的工作者线程event\*/
6. int freezeable;  /\* Freeze threads during suspend \*/
7. int rt;
8. };

     如果是多线程，Linux根据当前系统CPU的个数创建cpu\_workqueue\_struct 其结构体就是:

1. truct cpu\_workqueue\_struct {
2. spinlock\_t lock;/\*因为工作者线程需要频繁的处理连接到其上的工作，所以需要枷锁保护\*/
3. **struct list\_head worklist;**
4. wait\_queue\_head\_t more\_work;
5. **struct work\_struct \*current\_work; /\*当前的work\*/**
6. **struct workqueue\_struct \*wq;   /\*所属的workqueue\*/**
7. **struct task\_struct \*thread; /\*任务的上下文\*/**
8. } \_\_\_\_cacheline\_aligned;

       在该结构主要维护了一个任务队列，以及内核线程需要睡眠的等待队列，另外还维护了一个任务上下文，即task\_struct。  
       三者之间的关系如下：



3. 创建工作 **3.1 创建工作queue  
a. create\_singlethread\_workqueue(name)**        该函数的实现机制如下图所示，函数返回一个类型为struct workqueue\_struct的指针变量，该指针变量所指向的内存地址在函数内部调用kzalloc动态生成。所以driver在不再使用该work queue的情况下调用:

        void destroy\_workqueue(struct workqueue\_struct \*wq)来释放此处的内存地址。



        图中的cwq是一per-CPU类型的地址空间。对于create\_singlethread\_workqueue而言，即使是对于多CPU系统，内核也只负责创建一个worker\_thread内核进程。该内核进程被创建之后，会先定义一个图中的wait节点，然后在一循环体中检查cwq中的worklist，如果该队列为空，那么就会把wait节点加入到cwq中的more\_work中，然后休眠在该等待队列中。

        Driver调用queue\_work（struct workqueue\_struct \*wq, struct work\_struct \*work）向wq中加入工作节点。work会依次加在cwq->worklist所指向的链表中。queue\_work向cwq->worklist中加入一个work节点，同时会调用wake\_up来唤醒休眠在cwq->more\_work上的worker\_thread进程。wake\_up会先调用wait节点上的autoremove\_wake\_function函数，然后将wait节点从cwq->more\_work中移走。

        worker\_thread再次被调度，开始处理cwq->worklist中的所有work节点...当所有work节点处理完毕，worker\_thread重新将wait节点加入到cwq->more\_work，然后再次休眠在该等待队列中直到Driver调用queue\_work...

**b. create\_workqueue**



       相对于create\_singlethread\_workqueue, create\_workqueue同样会分配一个wq的工作队列，但是不同之处在于，对于多CPU系统而言，对每一个CPU，都会为之创建一个per-CPU的cwq结构，对应每一个cwq，都会生成一个新的worker\_thread进程。但是当用queue\_work向cwq上提交work节点时，是哪个CPU调用该函数，那么便向该CPU对应的cwq上的worklist上增加work节点。

**c.小结**       当用户调用workqueue的初始化接口create\_workqueue或者create\_singlethread\_workqueue对workqueue队列进行初始化时，内核就开始为用户分配一个workqueue对象，并且将其链到一个全局的workqueue队列中。然后Linux根据当前CPU的情况，为workqueue对象分配与CPU个数相同的cpu\_workqueue\_struct对象，每个cpu\_workqueue\_struct对象都会存在一条任务队列。紧接着，Linux为每个cpu\_workqueue\_struct对象分配一个内核thread，即内核daemon去处理每个队列中的任务。至此，用户调用初始化接口将workqueue初始化完毕，返回workqueue的指针。

        workqueue初始化完毕之后，将任务运行的上下文环境构建起来了，但是具体还没有可执行的任务，所以，需要定义具体的work\_struct对象。然后将work\_struct加入到任务队列中，Linux会唤醒daemon去处理任务。

       上述描述的workqueue内核实现原理可以描述如下：



**3.2  创建工作**       要使用工作队列，首先要做的是创建一些需要推后完成的工作。可以通过DECLARE\_WORK在编译时静态地建该结构：  
       DECLARE\_WORK(name,void (\*func) (void \*), void \*data);  
      这样就会静态地创建一个名为name，待执行函数为func，参数为data的work\_struct结构。  
      同样，也可以在运行时通过指针创建一个工作：  
      INIT\_WORK(structwork\_struct \*work, void(\*func) (void \*));

4. 调度 **a. schedule\_work**

       在大多数情况下, 并不需要自己建立工作队列，而是只定义工作, 将工作结构挂接到内核预定义的事件工作队列中调度, 在kernel/workqueue.c中定义了一个静态全局量的工作队列static struct workqueue\_struct \*keventd\_wq;默认的工作者线程叫做events/n，这里n是处理器的编号，每个处理器对应一个线程。比如，单处理器的系统只有events/0这样一个线程。而双处理器的系统就会多一个events/1线程。  
       调度工作结构, 将工作结构添加到全局的事件工作队列keventd\_wq，调用了queue\_work通用模块。对外屏蔽了keventd\_wq的接口，用户无需知道此参数，相当于使用了默认参数。keventd\_wq由内核自己维护，创建，销毁。这样work马上就会被调度，一旦其所在的处理器上的工作者线程被唤醒，它就会被执行。

**b. schedule\_delayed\_work(&work,delay);**      有时候并不希望工作马上就被执行，而是希望它经过一段延迟以后再执行。在这种情况下，同时也可以利用timer来进行延时调度，到期后才由默认的定时器回调函数进行工作注册。延迟delay后，被定时器唤醒，将work添加到工作队列wq中。

      工作队列是没有优先级的，基本按照FIFO的方式进行处理。

## ****5. 示例****

1. #include <linux/module.h>
2. #include <linux/init.h>
3. #include <linux/workqueue.h>
5. **static struct workqueue\_struct \*queue=NULL;**
6. **static struct work\_struct   work;**
8. staticvoid work\_handler(**struct work\_struct \*data)**
9. {
10. printk(KERN\_ALERT"work handler function.\n");
11. }
13. **static int \_\_init test\_init(void)**
14. {
15. queue=create\_singlethread\_workqueue("hello world");/\*创建一个单线程的工作队列\*/
16. **if (!queue)**
17. **goto err;**
19. INIT\_WORK(&work,work\_handler);
20. schedule\_work(&work);
22. return0;
23. err:
24. **return-1;**
25. }
27. **static   void \_\_exit test\_exit(void)**
28. {
29. destroy\_workqueue(queue);
30. }
31. MODULE\_LICENSE("GPL");
32. module\_init(test\_init);
33. module\_exit(test\_exit);

6. workqueue的API  
workqueue的API自2.6.20后发生了变化

1. #include <linux/workqueue.h>
2. struct workqueue\_struct;
3. struct work\_struct;
4. struct workqueue\_struct \***create\_workqueue**(const char \*name);
5. void **destroy\_workqueue**(struct workqueue\_struct \*queue);
6. **INIT\_WORK**(\_work, \_func);
7. **INIT\_DELAYED\_WORK**(\_work, \_func);
8. int **queue\_work**(struct workqueue\_struct \*wq, struct work\_struct \*work);
9. int **queue\_delayed\_work**(struct workqueue\_struct \*wq,struct delayed\_work \*dwork, unsigned long delay);
10. int **queue\_delayed\_work\_on**(int cpu, struct workqueue\_struct \*wq,

struct delayed\_work \*dwork, unsigned long delay);

1. int **cancel\_work\_sync**(struct work\_struct \*work);
2. int **cancel\_delayed\_work\_sync**(struct delayed\_work \*dwork);
3. void **flush\_workqueue**(struct workqueue\_struct \*wq);

Workqueue编程接口

|  |  |  |
| --- | --- | --- |
| 序号 | 接口函数 | 说明 |
| 1 | create\_workqueue | 用于创建一个workqueue队列，为系统中的每个CPU都创建一个内核线程。输入参数：  @name：workqueue的名称 |
| 2 | create\_singlethread\_workqueue | 用于创建workqueue，只创建一个内核线程。输入参数：  @name：workqueue名称 |
| 3 | destroy\_workqueue | 释放workqueue队列。输入参数：  @ workqueue\_struct：需要释放的workqueue队列指针 |
| 4 | schedule\_work | 调度执行一个具体的任务，执行的任务将会被挂入Linux系统提供的workqueue——keventd\_wq输入参数：  @ work\_struct：具体任务对象指针 |
| 5 | schedule\_delayed\_work | 延迟一定时间去执行一个具体的任务，功能与schedule\_work类似，多了一个延迟时间，输入参数：  @work\_struct：具体任务对象指针  @delay：延迟时间 |
| 6 | queue\_work | 调度执行一个指定workqueue中的任务。输入参数：  @ workqueue\_struct：指定的workqueue指针  @work\_struct：具体任务对象指针 |
| 7 | queue\_delayed\_work | 延迟调度执行一个指定workqueue中的任务，功能与queue\_work类似，输入参数多了一个delay。 |

**Ps：4.5与6.7基本一致，区别在与4.5执行的工作队列是内核提供的system\_wq;具体功能就是：提交工作给一个工作队列**

PS：在许多情况下, 设备驱动不需要它自己的工作队列。如果你只偶尔提交任务给队列, 简单地使用内核提供的共享的默认的队列可能更有效。若使用共享队列，就必须明白将和其他人共享它，这意味着不应当长时间独占队列(不能长时间睡眠), 并且可能要更长时间才能获得处理器。

共享队列的使用的顺序：  
（1） 建立work\_struct或delayed\_work

|  |
| --- |
| **static struct work\_struct jiq\_work; static struct delayed\_work jiq\_work\_delay;      /\* this line is in jiq\_init() \*/ INIT\_WORK(&jiq\_work, jiq\_print\_wq); INIT\_DELAYED\_WORK(&jiq\_work\_delay, jiq\_print\_wq);** |

（2）提交工作

|  |
| --- |
| **int schedule\_work(&jiq\_work);/\*对于work\_struct结构\*/ int schedule\_delayed\_work(&jiq\_work\_delay, delay);/\*对于delayed\_work结构\*/**  **/\*返回值的定义和 queue\_work 一样\*/** |

若需取消一个已提交给工作队列入口项, 可以使用 cancel\_delayed\_work和cancel\_work\_sync， 但刷新共享队列需要一个特殊的函数:

|  |
| --- |
| **void flush\_scheduled\_work(void);** |

因为不知道谁可能使用这个队列，因此不可能知道 flush\_schduled\_work 返回需要多长时间。

## 参考资料

<http://www.cnblogs.com/sky-heaven/p/5847519.html>

<http://bgutech.blog.163.com/blog/static/18261124320116181119889/>

<http://kernel.meizu.com/linux-workqueue.html>

http://blog.chinaunix.net/uid-7332782-id-3206831.html

# [同步/异步与阻塞/非阻塞的区别消息](http://www.cnblogs.com/Ray-chen/archive/2011/12/27/2303115.html)

同步/异步与阻塞/非阻塞的区别.

同步和异步的概念,与**消息的通知机制**有关.  
举个例子,比如我去银行办理业务,可能选择排队等候,也可能取一个小纸条上面有我的号码,等到排到我这一号时由柜台的人通知我轮到我去办理业务了.  
前者(排队等候)就是同步等待消息,而后者(等待别人通知)就是异步等待消息.

在异步消息处理中,等待消息者(在这个例子中就是等待办理业务的人)往往注册一个回调机制,在所等待的事件被触发时由触发机制(在这里是柜台的人)通过某种机制(在这里是写在小纸条上的号码)找到等待该事件的人.[**异步通知机制：fasync-helper,kill-fasync**]

在实际的程序中,同步消息处理就好比简单的read/write操作,它们需要等待这两个操作成功才能返回;

而异步处理机制就是类似于select/poll之类的多路复用IO操作,当所关注的消息被触发时,由消息触发机制通知触发对消息的处理.  
  
阻塞和非阻塞,这两个概念与**程序等待消息(无所谓同步或者异步)时的状态**有关.  
继续上面的那个例子,不论是排队还是使用号码等待通知,如果在这个等待的过程中,等待者除了等待消息之外不能做其它的事情,那么该机制就是阻塞的,表现在程序中,也就是该程序一直阻塞在该函数调用处不能继续往下执行.[**通常来说进程会睡眠，把CPU等资源让给其他进程**]

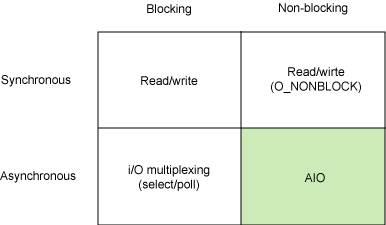
相反,有的人喜欢在银行办理这些业务的时候一边打打电话发发短信一边等待,这样的状态就是非阻塞的,因为他(等待者)没有阻塞在这个消息通知上,而是一边做自己的事情一边等待.**[一般来说进程会立刻返回一个值，可能是EAGAIN等用来说明资源不可用]**

**同步非阻塞形式实际上是效率低下的**,想象一下你一边打着电话一边还需要抬头看到底队伍排到你了没有,如果把打电话和观察排队的位置看成是程序的两个操作的话,这个程序需要在这两种不同的行为之间来回的切换,效率可想而知是低下的;而后者,异步非阻塞形式却没有这样的问题,因为打电话是你(等待者)的事情,而通知你则是柜台(消息触发机制)的事情,程序没有在两种不同的操作中来回切换.  
  
很多人会把同步和阻塞混淆,我想是因为很多时候同步操作会以阻塞的形式表现出来,比如很多人会写阻塞的read/write操作,但是别忘了可以对fd设置O\_NONBLOCK标志位,这样就可以将同步操作变成非阻塞的了;同样的,很多人也会把异步和非阻塞混淆,因为异步操作一般都不会在真正的IO操作处被阻塞,比如如果用select函数,当select返回可读时再去read一般都不会被阻塞,就好比当你的号码排到时一般都是在你之前已经没有人了,所以你再去柜台办理业务就不会被阻塞.  
  
可见,同步/异步与阻塞/非阻塞是两组不同的概念,它们可以共存组合,也可以参见这里:  
http://www.ibm.com/developerworks/cn/linux/l-async/  
  
同步和异步:上面提到过,**同步和异步仅仅是关于所关注的消息如何通知的机制,而不是处理消息的机制**.也就是说,**同步的情况下,是由处理消息者自己去等待消息是否被触发,而异步的情况下是由触发机制来通知处理消息者**,所以在异步机制中,处理消息者和触发机制之间就需要一个连接的桥梁,在我们举的例子中这个桥梁就是小纸条上面的号码,而在select/poll等IO多路复用机制中就是fd,当消息被触发时,触发机制通过fd找到处理该fd的处理函数.[异步通知机制则更加完美]  
  
请注意理解**消息通知和处理消息**这两个概念,这是理解这个问题的关键所在.还是回到上面的例子,轮到你办理业务这个就是你关注的消息,而去办理业务就是对这个消息的处理,两者是有区别的. 而在真实的IO操作时,所关注的消息就是该fd是否可读写,而对消息的处理就是对这个fd进行读写.同步/异步仅仅关注的是如何通知消息,它们对如何处理消息并不关心,好比说,银行的人仅仅通知你轮到你办理业务了,而如何办理业务他们是不知道的.  
  
而很多人之所以把同步和阻塞混淆,我想也是因为没有区分这两个概念,比如阻塞的read/write操作中,其实是把消息通知和处理消息结合在了一起,在这里所关注的消息就是fd是否可读/写,而处理消息则是对fd读/写.当我们将这个fd设置为非阻塞的时候,read/write操作就不会在等待消息通知这里阻塞,如果fd不可读/写则操作立即返回.  
  
很多人又会问了,异步操作不会是阻塞的吧?已经通知了有消息可以处理了就一定不是阻塞的了吧?  
其实**异步操作是可以被阻塞住的,只不过通常不是在处理消息时阻塞,而是在等待消息被触发时被阻塞**. 比如select函数,假如传入的最后一个timeout参数为NULL,那么如果所关注的事件没有一个被触发,程序就会一直阻塞在这个select调用处.而如果使用异步非阻塞的情况,比如aio\_\*组的操作,当我发起一个aio\_read操作时,函数会马上返回不会被阻塞,当所关注的事件被触发时会调用之前注册的回调函数进行处理,具体可以参见我上面的连接给出的那篇文章.回到上面的例子中,如果在银行等待办理业务的人采用的是异步的方式去等待消息被触发,也就是领了一张小纸条,假如在这段时间里他不能离开银行做其它的事情,那么很显然,这个人被阻塞在了这个等待的操作上面;但是呢,这个人突然发觉自己烟瘾犯了,需要出去抽根烟,于是他告诉大堂经理说,排到我这个号码的时候麻烦到外面通知我一下(注册一个回调函数),那么他就没有被阻塞在这个等待的操作上面,自然这个就是异步+非阻塞的方式了.

Linux aio是Linux下的异步读写模型。Linux 异步 I/O 是 Linux 内核中提供的一个相当新的增强。它是 2.6 版本内核的一个标准特性。对于文件的读写，即使以O\_NONBLOCK方式来打开一个文件，也会处于"阻塞"状态。因为文件时时刻刻处于可读状态。而从磁盘到内存所等待的时间是惊人的。为了充份发挥把数据从磁盘复制到内存的时间，引入了aio模型。AIO 背后的基本思想是允许进程发起很多 I/O 操作，而不用阻塞或等待任何操作完成。稍后或在接收到 I/O 操作完成的通知时，进程就可以检索 I/O 操作的结果。

## I/O 模型

在深入介绍 AIO API 之前，让我们先来探索一下 Linux 上可以使用的不同 I/O 模型。这并不是一个详尽的介绍，但是我们将试图介绍最常用的一些模型来解释它们与异步 I/O 之间的区别。图 1 给出了同步和异步模型，以及阻塞和非阻塞的模型。  
  
  
图 1. 基本 Linux I/O 模型的简单矩阵  
   
每个 I/O 模型都有自己的使用模式，它们对于特定的应用程序都有自己的优点。本节将简要对其一一进行介绍。

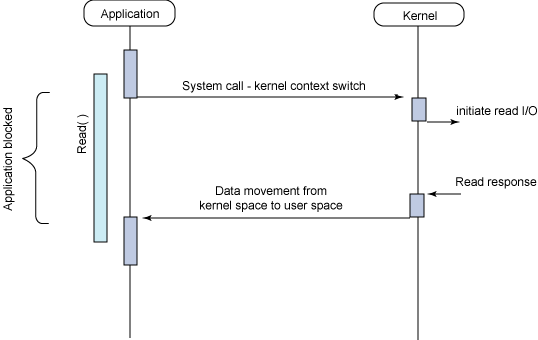


## 同步阻塞 I/O

最常用的一个模型是同步阻塞 I/O 模型。在这个模型中，用户空间的应用程序执行一个系统调用，这会导致应用程序阻塞。这意味着应用程序会一直阻塞，直到系统调用完成为止（数据传输完成或发生错误）。调用应用程序处于一种不再消费 CPU 而只是简单等待响应的状态，因此从处理的角度来看，这是非常有效的。

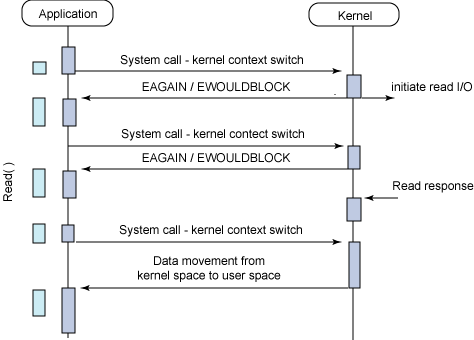
图 2 给出了传统的阻塞 I/O 模型，这也是目前应用程序中最为常用的一种模型。其行为非常容易理解，其用法对于典型的应用程序来说都非常有效。在调用 read 系统调用时，应用程序会阻塞并对内核进行上下文切换。然后会触发读操作，当响应返回时（从我们正在从中读取的设备中返回），数据就被移动到用户空间的缓冲区中。然后应用程序就会解除阻塞（read 调用返回）。

图 2. 同步阻塞 I/O 模型的典型流程  
   
从应用程序的角度来说，read 调用会延续很长时间。实际上，在内核执行读操作和其他工作时，应用程序的确会被阻塞。



## 同步非阻塞 I/O

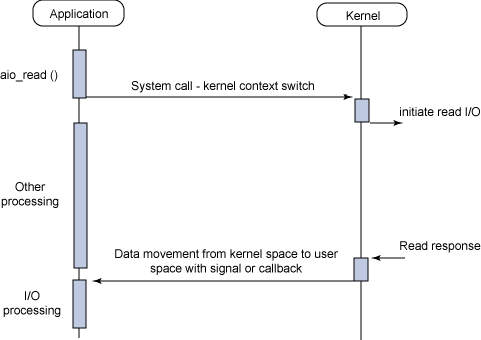
同步阻塞 I/O 的一种效率稍低的变种是同步非阻塞 I/O。在这种模型中，设备是以非阻塞的形式打开的。这意味着 I/O 操作不会立即完成，read 操作可能会返回一个错误代码，说明这个命令不能立即满足（EAGAIN 或 EWOULDBLOCK），如图 3 所示。  
  
  
图 3. 同步非阻塞 I/O 模型的典型流程  
   
非阻塞的实现是 I/O 命令可能并不会立即满足，需要应用程序调用许多次来等待操作完成。这可能效率不高，因为在很多情况下，当内核执行这个命令时，应用程序必须要进行忙碌等待，直到数据可用为止，或者试图执行其他工作。正如图 3 所示的一样，这个方法可以引入 I/O 操作的延时，因为数据在内核中变为可用到用户调用 read 返回数据之间存在一定的间隔，这会导致整体数据吞吐量的降低。



## 异步阻塞 I/O

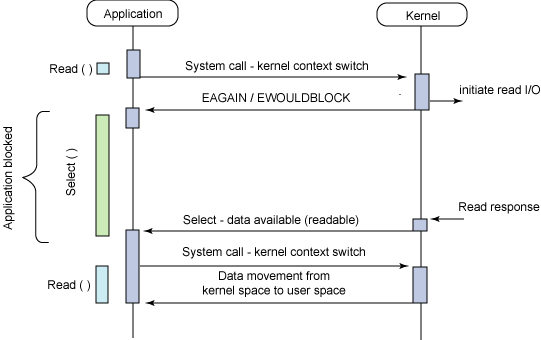
另外一个阻塞解决方案是带有阻塞通知的非阻塞 I/O。在这种模型中，配置的是非阻塞 I/O，然后使用阻塞 select 系统调用来确定一个 I/O 描述符何时有操作。使 select 调用非常有趣的是它可以用来为多个描述符提供通知，而不仅仅为一个描述符提供通知。对于每个提示符来说，我们可以请求这个描述符可以写数据、有读数据可用以及是否发生错误的通知。

图 4. 异步阻塞 I/O 模型的典型流程 (select)  
   
select 调用的主要问题是它的效率不是非常高。尽管这是异步通知使用的一种方便模型，但是对于高性能的 I/O 操作来说不建议使用。



## 异步非阻塞 I/O（AIO）

最后，异步非阻塞 I/O 模型是一种处理与 I/O 重叠进行的模型。读请求会立即返回，说明 read 请求已经成功发起了。在后台完成读操作时，应用程序然后会执行其他处理操作。当 read 的响应到达时，就会产生一个信号或执行一个基于线程的回调函数来完成这次 I/O 处理过程。  
  
  
图 5. 异步非阻塞 I/O 模型的典型流程  
 



在一个进程中为了执行多个 I/O 请求而对计算操作和 I/O 处理进行重叠处理的能力利用了处理速度与 I/O 速度之间的差异。当一个或多个 I/O 请求挂起时，CPU 可以执行其他任务；或者更为常见的是，在发起其他 I/O 的同时对已经完成的 I/O 进行操作。

从前面 I/O 模型的分类中，我们可以看出 AIO 的动机。这种阻塞模型需要在 I/O 操作开始时阻塞应用程序。这意味着不可能同时重叠进行处理和 I/O 操作。同步非阻塞模型允许处理和 I/O 操作重叠进行，但是这需要应用程序根据重现的规则来检查 I/O 操作的状态。这样就剩下异步非阻塞 I/O 了，它允许处理和 I/O 操作重叠进行，包括 I/O 操作完成的通知。除了需要阻塞之外，select 函数所提供的功能（异步阻塞 I/O）与 AIO 类似。不过，它是对通知事件进行阻塞，而不是对 I/O 调用进行阻塞。

## 异步通知范例

阻塞式I/O是一直等待直到设备可以访问，非阻塞式I/O是定期轮询设备是否可以访问。异步通知则是当设备可以访问时才主动通知应用程序，有点像设备的硬中断。

并不是所有的设备都支持异步通知，应用程序通常假设只有套接字和终端才有异步通知的能力。

异步通知存在一个问题，当进程收到SIGIO信号时，它并不知道是哪个文件有了新的输入，如果**有多于一个的文件可以异步通知同一个进程，那么应用进程还需要借助于poll或select来确定输入的来源。**

### 应用层

1. #include <stdio.h>
2. #include <unistd.h>
3. #include <stdlib.h>
4. #include <signal.h>
5. #include <fcntl.h>
7. #define MAX\_LEN 100
9. **void** my\_handler(**int** signum)
10. {
11. **char** data[MAX\_LEN];
12. **int** len;
14. len = read(STDIN\_FILENO, &data, MAX\_LEN);
15. data[len] = 0;
17. printf("Input message: %s\n", data);
18. exit(0);
19. }
21. **int** main(**void**)
22. {
23. **int** oflags;
25. /\* set new SIGIO handler \*/
26. signal(SIGIO, my\_handler);
27. /\* set fd's owner process \*/
28. fcntl(STDIN\_FILENO, F\_SETOWN, getpid());
29. /\* get old fd flags \*/
30. oflags = fcntl(STDIN\_FILENO, F\_GETFL);
31. /\* set new fd flags \*/
32. fcntl(STDIN\_FILENO, F\_SETFL, oflags | O\_ASYNC);
34. /\* infinitely wait until recv SIGIO \*/
35. **while**(1);
36. **return** 0;
37. }

为了能处理设备发出的SIGIO信号，用户程序需要做：

1. 设置设备文件的拥有者为本进程，这样一来才能收到设备驱动发出的SIGIO信号。

2. 使设备文件支持异步通知，即设置O\_ASYNC标志。

3. 通过signal()指定SIGIO的处理函数。

### 设备驱动

1. struct fasync\_struct {
2. spinlock\_t fa\_lock;
3. **int** magic;
4. **int** fa\_fd; /\* 文件描述符 \*/
5. struct fasync\_struct \*fa\_next; /\* 用于链入单向链表 \*/
6. struct file \*fa\_file; /\* fa\_file->f\_owner记录接收信号的进程 \*/
7. struct rcu\_head fa\_rcu;
8. };

(1) 设备的异步通知链

1. struct xxx\_dev {
2. struct cdev cdev;
3. ...
4. struct fasync\_struct \*async\_queue; /\* 异步通知链 \*/
5. };

(2) 插入到异步通知链

1. **static** **int** xxx\_fasync(**int** fd, struct file \*filp, **int** on)
2. {
3. struct xxx\_dev \*dev = filp->private\_data;
4. **return** fasync\_helper(fd, filp, on, &dev->async\_queue);
5. }

(3) 发送信号通知进程

设备使用kill\_fasync()来发送信号给用户进程，一般sig为SIGIO，可读时band为POLL\_IN，

可写时band为POLL\_OUT。

1. **static** ssize\_t xxx\_write(struct file \*filp, **const** **char** \_\_user \*buf, size\_t count, loff\_t \*f\_ops)
2. {
3. struct xxx\_dev \*dev = filp->private\_data;
4. ...
5. **if** (dev->async\_queue) {
6. kill\_fasync(&dev->async\_queue, SIGIO, POLL\_IN);
7. }
8. ...
9. }

(4) 从异步通知链中删除

在关闭文件时，需要把对应的异步通知结构从链表中删除。

1. **static** xxx\_release(struct inode \*inode, struct file \*filp)
2. {
3. struct xxx\_dev \*dev = filp->private\_data;
4. ...
5. fasync\_helper(-1, filp, 0, &dev->async\_queue);
6. ...
7. }

### 内核API

异步通知结构体的插入和删除，失败返回负值，没有变化返回0，有变化返回正值。

1. /\* fasync\_helper() is used by almost all character device drivers to set up the
2. \* fasync queue, and for regular files by the file lease code. It returns negative
3. \* on error, 0 if it did no changes and positive if it added/deleted the entry.
4. \*/
6. **int** fasync\_helper(**int** fd, struct file \*filp, **int** on, struct fasync\_struct \*\*fapp)
7. {
8. **if** (!on) /\* 删除 \*/
9. **return** fasync\_remove\_entry(filp, fapp);
10. **return** fasync\_add\_entry(fd, filp, fapp); /\* 插入 \*/
11. }

发送信号，通知用户进程，sig一般设为SIGIO，可读时band为POLL\_IN，可写时band为POLL\_OUT。

1. **void** kill\_fasync(struct fasync\_struct \*\*fp, **int** sig, **int** band)
2. {
3. /\* First a quick test without locking: usually the list is empty. \*/
4. **if** (\*fp) {
5. rcu\_read\_lock();
6. kill\_fasync\_rcu(rcu\_dereference(\*fp), sig, band);
7. rcu\_read\_unlock();
8. }
9. }

## 参考资料

<http://www.cnblogs.com/Ray-chen/archive/2011/12/27/2303115.html>

<http://www.ibm.com/developerworks/cn/linux/l-async/>

<http://blog.csdn.net/zhangskd/article/details/45938267> 异步通知的使用方法

ps:

阻塞：在执行设备操作的时候，若不能获得资源，则挂起进程（睡眠等待）直到满足操作条件（唤醒，通过等待队列将当前进程加入等待队列，未来某一时刻即可唤醒）

阻塞io:进程阻塞在io操作上（读写一个文件，读写一个串口，读写单一设备的时候用法哦）

非阻塞io：防止进程阻塞在io操作上（在单一线程里面服务多个io）

io多路复用：允许同时对多个io进行控制

信号驱动io：一种异步通信模型

一个驱动应该同时支持上面四种io方式

# ****Linux内核的动态电压和电流控制接口****

　　前面已经提到半导体器件的功耗是两个部分组成，一是静态功耗，一是动态功耗。静态功耗主要来自待机状态的泄漏电流，相比而言动态功耗更大，例如，音视频播放中频率和电压的增加会让电量将成线形增长，动态功耗也是电源管理要解决的主要问题，解决动态功耗的方法有几种，如IBM和Montavista合作开发DPM项目（现用在Montavista Mobilinux 5.0 产品中）和TI OMAP[3430](http://www.dzsc.com/stock-ic/3430.html)的Linux电源管理，自[Wolfson](http://www.dzsc.com/stock/W/WOLFSON.html)微电子的Liam Girdwood最近介绍了一种称为校准器（regulator）的动态电压和电流控制的方法，很有参考意义和实际使用价值。

1 校准器的基本概念

　　所谓校准器实际是在软件控制下把输入的电源调节精心输出。例如电压的控制，输入时5V 输出是[1.8V](http://www.dzsc.com/stock/1/1-8V.html)；电流的限制，最大20mA；简单的切换和电源的[开关](http://www.dzsc.com/product/file411.html)等，如图1所示。



　　电源域是一组校准器，设备组成、输入可能是校准器，开关也许是电源域，电源域可以级联，电源约束可以和电源域配合以保护硬件。例如一个Internet Tablet/PMP，它由CPU、NOR [Flash](http://www.dzsc.com/stock-ic/FLASH.html)、音频编解码器、[触摸屏](http://www.dzsc.com/product/file825.html)、LCD控制器、USB、WiFi 等其他外设组成，如图2所示。



　　为了实现上面的构想，需要在内核里建立一个校准器构架，目的就是设计一个可以控制电压和电流的标准内核接口以节省电能，从而尽可能的延长电池的供应。这个内核的架构分为四个部分：**针对设备驱动的消费接口（consumer）、校准器驱动的接口、系统配置的接口和面向应用sysfs的userspace接口。**

## ****2 Consumer的API****

regulator = regulator\_get(dev, “Vcc”)；

　　其中，dev 是设备“Vcc”一个字符串代表，校准器（regulator）然后返回一个指针，也是regulator\_put(regulator)使用的。

　　打开和关闭校准器（regulator）API如下。

int regulator\_enable(regulator);

int regulator\_disable(regulator);

## ****3 电压的API****

　　消费者可以申请提供给它们的电压，如下所示。

int regulator\_set\_voltage(regulator, int min\_uV, int max\_uV);

　　在改变电压前要检查约束，如下所示。

regulator\_set\_voltage(regulator,[10000](http://www.dzsc.com/stock-ic/10000.html)0,[150000](http://www.dzsc.com/stock/1/150000.html))

　　电压值下面的设置改变如下所示。

int regulator\_get\_voltage)struct regulator \*regulator);

## ****4 电流的API****

　　电流的API也是类似，需要指出的是，校准器的方法并不一定是最高的效率，效率和加载(如加载10mA电流)、操作模式都有关系，通过下面的API可以改变模式设置。

regulator\_set\_optimum\_mode(requlator,10000);//10mA

## ****5 校准器的驱动和系统配置****

　　在实际使用校准器之前，需要按照下面的结构写校准器的驱动程序，然后注册后通知给消费者使用。

struct regulator\_ops {

/\* get/set regulator voltage \*/

int (\*set\_voltage)(struct regulator\_cdev \*, int uV);

int (\*get\_voltage)(struct regulator\_cdev \*);

/\* get/set regulator current \*/

int (\*set\_current)(struct regulator\_cdev \*, int uA);

int (\*get\_current)(struct regulator\_cdev \*);

/\* enable/disable regulator \*/

int (\*enable)(struct regulator\_cdev \*);

int (\*disable)(struct regulator\_cdev \*);

int (\*is\_enabled)(struct regulator\_cdev \*);

/\* get/set regulator operating mode (defined in regulator.h) \*/

int (\*set\_mode)(struct regulator\_cdev \*, unsigned int mode);

unsigned int (\*get\_mode)(struct regulator\_cdev \*);

/\* get most efficient regulator operating mode for load \*/

unsigned int (\*get\_optimum\_mode)(struct regulator\_cdev \*, int input\_uV,

int output\_uV, int load\_uA);

};

　　完成了校准器驱动程序之后，下一步就是系统设置（machine specific），即匹配如电压、LDO1和NAND等关系。

regulator\_set\_supply(“LDO1”,dev, “Vcc”)

　　对于userspace，校准器的使用是通过sysfs，但是目前所有的包括电压、电流、操作模式、限制等信息多只是只读信息，应该是非常适合象powerTop这样工具的使用。

## ****6 应用****

　　校准器的典型的应用包括如下：CPUfreq——CPU频率的调节；CPU idle——CPU空闲模式控制；LCD[背光](http://www.dzsc.com/product/searchfile/124.html)调节——通过电流控制LED灯的亮度达到控制LCD背光的目的；音频单元——如FM收音机在MP3使用的时候应该是关闭的，麦克风使用的时候，[扬声器](http://www.dzsc.com/product/searchfile/856.html)的放大器应该是关闭的；NAND/NOR[存储器](http://www.dzsc.com/product/searchfile/310.html)是耗电大户，根据不同操作方式（读/写、擦除等）优化操作模式（控制电流）达到节省电量的要求。同其他电源管理的方法比较，校准器方法具有一定的硬件独立和抽象性，简单实用，原理上可以适合任何有电源管理芯片支持嵌入式系统电源管理，目前已经移植到[Freescale](http://www.dzsc.com/stock-ic/FREESCALE.html) [MC13783](http://www.dzsc.com/stock_MC1/MC13783.html)、Wolfson WM8350/8400等几个集成度很高的电源管理器件上了.

　　基于构件的面向CPU的电源管理技术

　　无论是PM\_QoS、控制电压和电流的校准器方法，还是许许多多半导体公司支持自己CPU和电源管理芯片的Linux BSP电源管理部分，都还没有一个构建在更高层面的构件级嵌入式系统电源解决方案和商业产品。虽然包括CELF（消费和嵌入式Linux 论坛）和Intel主导的Mobile &Internet Linux项目都设立了专门的电源管理计划（power manager project），但是显然距离人们的要求和实际的应用还太远了。

Montavista在过去和IBM合作开发DPM（动态电源管理）技术的基础上，最近在专门针对手机、互联网移动终端、PMP/PDN等便携消费电子设备的mobilinux5.0上提出嵌入式电源管理技术的构件方法。Montavista的构件方式主要是针对以先进的多媒体应用处理器为核心的新一代嵌入式系统，比如Freescale的MX31、TI OMAP2430/3430为核心的系统级电源管理，它包含下面几个主要的部分。

## ****1 动态的电压和频率调节****

　　正如前面提到的，电压和频率的提升将会让功耗线性增加，按照设计需要和应用的指令将电压和频率调节到合适的操作点可以大大降低功耗的有效方法。要想实现动态的电压和频率调节（DVFS），在内核里CPUrefs子系统是关键的部件，如图3所示。



　　图3 CPUrefs结构

　　那么管理者（Governor）是按照什么情况改变操作点呢？性能要求、省电的要求、用户的应用以及CPU的使用效率等条件都可以让管理者改变操作模式。Mobilinux5.0提供了userspace机制充当管理者的工作，即应用可以改变操作点。

　　在TI OMAP3中有一个称为SmartReflex的技术，动态调整VDD1和VDD2操作点电压以适应芯片特性、温度和电压。SmartReflex技术有四个级别：0级——在工厂生产时优化校准后设置的操作点；1级——引导时优化后校准确定的操作点；2级——通过软件循环实时优化电压点然后由CPU的中断程序设置；3级——完全的硬件循环优化电压点，无须CPU干预，是一种硬件控制“傻瓜”操作点改变方式。无论是mobilinux5.0还是TI 3430 Linux distribution都已经支持DVFS和SmartReflex驱动。

## ****2 挂起和恢复****

　　在内核里，mobilinux5.0已经提供支持挂起和恢复的驱动程序的功能，新的驱动必须要增加回调函数以响应系统休眠中关机和再次唤醒的动作。

## ****3 支持电源管理的驱动程序****

　　每一个驱动程序必须经过重新的书写支持DVFS，即当操作点改变的时候，驱动程序通过CPUrefs的告知作出响应。驱动程序还必须正确处理系统的挂起和恢复事件。

## ****4 CPU空闲调节****

　　由一个定义的处理器特定的空闲状态点的CPUidle驱动管理、内核的一个CPUidle 框架和管理者组成，如OMAP3430定义7个空闲状态点。

## ****5 应用设计策略****

　　包括手机在内的便携式消费电子产品主要的能耗分布如图4所示。



　　图4 便携电子产品能耗分布

　　除了CPU外，其他主要的能耗大户是LCD背光、NOR/NAND Flash/RAM存储器、[DC/DC](http://www.dzsc.com/stock-ic/DC_DC.html)转换和音视频放大器等，例如，MPEG4的播放就是一个能耗集中的应用。

　　如何使用mobilinux5.0等已经具备电源管理功能的商业嵌入式操作系统以使便携电子产品能耗降低到最少？下面的方法是设计人员应该考虑的。

·实际测量的结果证明使用DVFS的方法是降低CPU运行时的能耗的关键。当然，如何让管理者设置操作点和状态转移是要设计人员全盘考虑的。

·不要忽视CPU空闲状态的能耗管理。mobilinux5.0的CPUrefs 和所有的驱动都已经支持空闲的调节（idle scaling），加上内核使用了动态滴嗒时钟（dynamic tick），改变了过去CPU无论是否运行都按照固定的时间唤醒的方法，大大节省能源。

·可延迟的定时器（deferrable timer）——它可以告诉内核某个定时器不需要在时限到的时候唤醒，这将可以降低能耗。

·PowerTop工具——前面已经提到的这个工具已经集成到mobilinux5.0 中，而且证明对于分析系统空闲状态是一个非常有用的工具

## 参考资料

<http://www.cnblogs.com/lishixian/articles/2866998.html>

# [led子系统](http://blog.csdn.net/yuanlulu/article/details/6438841)

============================================  
作者：yuanlulu  
http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/yuanlulu

版权没有，但是转载请保留此段声明  
============================================

## ****数据结构****

/include/linux/leds.h

enum led\_brightness {  
     LED\_OFF          = 0,  
     LED\_HALF     = 127,  
     LED\_FULL     = 255,  
};

led\_classdev代表**led的实例**：  
struct led\_classdev {  
     const char          \*name;               //名字  
     int               brightness;                   //当前亮度  
     int               flags;                         //标志，目前只支持 LED\_SUSPENDED  
  
#define LED\_SUSPENDED          (1 << 0)  
  
     /\*设置led的亮度，不可以睡眠，有必要的话可以使用工作队列\*/  
     void          (\*brightness\_set)(struct led\_classdev \*led\_cdev,  
                           enum led\_brightness brightness);  
     /\* 获取亮度 \*/  
     enum led\_brightness (\*brightness\_get)(struct led\_classdev \*led\_cdev);  
  
     /\* 激活硬件加速的闪烁 \*/  
     int          (\*blink\_set)(struct led\_classdev \*led\_cdev,  
                         unsigned long \*delay\_on,  
                         unsigned long \*delay\_off);  
  
     struct device          \*dev;  
     struct list\_head     node;               /\* 所有已经注册的led\_classdev使用这个节点串联起来 \*/  
     const char          \*default\_trigger;     /\* 默认触发器 \*/  
  
#ifdef CONFIG\_LEDS\_TRIGGERS               //如果配置内核时使能了触发器功能，才会编译下面一段  
     /\* 这个读写子轩锁保护下面的触发器数据 \*/  
     struct rw\_semaphore     trigger\_lock;  
  
     struct led\_trigger     \*trigger;          //触发器指针  
     struct list\_head     trig\_list;               //触发器使用的链表节点，用来连接同一触发器上的所有led\_classdev  
     void               \*trigger\_data;            //触发器使用的私有数据  
#endif  
};

## ****触发器的结构体****

#define TRIG\_NAME\_MAX 50  
struct led\_trigger {  
     const char     \*name;          //触发器名字  
     void          (\*activate)(struct led\_classdev \*led\_cdev);     //激活ledled。led\_classdev和触发器建立连接时会调用这个方法。  
     void          (\*deactivate)(struct led\_classdev \*led\_cdev);     //取消激活。led\_classdev和触发器取消连接时会调用这个方法。  
  
     /\* 本触发器控制之下的led链表 \*/  
     rwlock\_t       leddev\_list\_lock; //保护链表的锁  
     struct list\_head  led\_cdevs;    //链表头  
  
     /\* 连接下一个已注册触发器的链表节点 ，所有已注册的触发器都会被加入一个全局链表\*/  
     struct list\_head  next\_trig;  
};

## ****平台设备相关的led数据结构****

struct led\_info {  
     const char     \*name;  
     char          \*default\_trigger;  
     int          flags;  
};  
  
struct led\_platform\_data {  
     int          num\_leds;  
     struct led\_info     \*leds;  
};

## ****平台设备相关的gpio led数据结构****

struct gpio\_led {  
     const char \*name;  
     char \*default\_trigger;  
     unsigned      gpio;  
     u8           active\_low;  
};  
  
struct gpio\_led\_platform\_data {  
     int           num\_leds;  
     struct gpio\_led \*leds;  
     int          (\*gpio\_blink\_set)(unsigned gpio,  
                         unsigned long \*delay\_on,  
                         unsigned long \*delay\_off);  
};

led\_classdev接口分析/driver/rtc/led-class.c

## ****注册struct led\_classdev：****

int **led\_classdev\_register**(struct device \*parent, struct led\_classdev \*led\_cdev)  
{  
     int rc;

     /\* 创建一个struct device，他的父设备是parent，drvdata是led\_cdev，名字是led\_cdev->name，类别是 leds\_class\*/

     led\_cdev->dev = device\_create\_drvdata(leds\_class, parent, 0, led\_cdev,  
                               "%s", led\_cdev->name);  
     if (IS\_ERR(led\_cdev->dev))  
          return PTR\_ERR(led\_cdev->dev);  
  
     /\* register the attributes \*/  
     rc = device\_create\_file(led\_cdev->dev, &dev\_attr\_brightness);//在sys/class/rtc/下创建一个led的属性文件。  
     if (rc)  
          goto err\_out;  
  
     /\* add to the list of leds \*/  
     down\_write(&leds\_list\_lock);  
     list\_add\_tail(&led\_cdev->node, &leds\_list);//将新的led加入链表，全局链表是leds\_list  
     up\_write(&leds\_list\_lock);  
  
     led\_update\_brightness(led\_cdev);//获取led当前的亮度更新led\_cdev的brightness成员  
  
#ifdef CONFIG\_LEDS\_TRIGGERS  
     init\_rwsem(&led\_cdev->trigger\_lock);//初始化led\_cdev的触发器自旋锁  
  
     rc = device\_create\_file(led\_cdev->dev, &dev\_attr\_trigger);//在sys/class/led中为触发器创建属性文件  
     if (rc)  
          goto err\_out\_led\_list;  
  
     **led\_trigger\_set\_default**(led\_cdev); //为led\_cdev设置默认的触发器  
#endif  
  
     printk(KERN\_INFO "Registered led device: %s/n",  
               led\_cdev->name);  
  
     return 0;  
  
#ifdef CONFIG\_LEDS\_TRIGGERS  
err\_out\_led\_list:  
     device\_remove\_file(led\_cdev->dev, &dev\_attr\_brightness);  
     list\_del(&led\_cdev->node);  
#endif  
err\_out:  
     device\_unregister(led\_cdev->dev);  
     return rc;  
}  
EXPORT\_SYMBOL\_GPL(led\_classdev\_register);

## ****注销struct led\_classdev：****

void led\_classdev\_unregister(struct led\_classdev \*led\_cdev)；

注销所做的工作和注册相反。

**将led挂起**：将led的flag设为LED\_SUSPENDED，关闭led.

void led\_classdev\_suspend(struct led\_classdev \*led\_cdev)

**从挂起中恢复：**

void led\_classdev\_resume(struct led\_classdev \*led\_cdev)

## ****sysfs中的属性文件****：

/driver/rtc/led-class.c会首先创建一个leds类，生成/sys/class/leds目录。

在led\_classdev\_register中生成了两个sysfs属性文件，它们使用的属性参数如下：

static DEVICE\_ATTR(**brightness**, 0644, led\_brightness\_show, led\_brightness\_store);  
static DEVICE\_ATTR(**trigger**, 0644, led\_trigger\_show, led\_trigger\_store);

led\_brightness\_show和led\_brightness\_store分别负责显示和设置亮度，用户控件通过

/sys/class/leds/<device>/brightness查看和设置亮度就是和这两个函数交互的。

 led\_trigger\_show用于读取当前触发器的名字，led\_trigger\_store用于指定触发器的名字，

它会寻找所有已注册的触发器，找到同名的并设置为当前led的触发器。

/sys/class/leds/<device>/trigger用于用户空间查看和设置触发器。

## ****led\_classdev全局链表：****

led\_classdev\_register注册的struct led\_classdev会被加入leds\_list链表，这个链表定义在driver/leds/led-core.c。

**led\_trigger接口分析**/driver/leds/led-triggers.c

**注册触发器**

int led\_trigger\_register(struct led\_trigger \*trigger)；

这个函数注册的trigger会被加入全局链表**trigger\_list**，这个链表头是在/driver/leds/led-triggers.c定义的。

此外，这个函数还会遍历所有的已注册的 led\_classdev，如果有哪个led\_classdev的默认触发器和自己同名，则

调用led\_trigger\_set将自己设为那个led的触发器。

led\_classdev注册的时候也会调用led\_trigger\_set\_default来遍历所有已注册的触发器，找到和led\_classdev.default\_trigger同名的触发器则将它设为自己的触发器。

**注销触发器**

void led\_trigger\_unregister(struct led\_trigger \*trigger)；

这个函数做和注册相反的工作，并把所有和自己建立连接的led的led\_classdev.trigger设为NULL。

**设置触发器上所有的led为某个亮度**

void led\_trigger\_event(struct led\_trigger \*trigger, enum led\_brightness brightness)；

**注册触发器的简单方法**

指定一个名字就可以注册一个触发器，注册的触发器通过\*\*tp返回，但是这样注册的触发器没有active和deactivede。

void led\_trigger\_register\_simple(const char \*name, struct led\_trigger \*\*tp)；

相对应的注销函数为：

void led\_trigger\_unregister\_simple(struct led\_trigger \*trigger)；

**触发器和led的连接**

void led\_trigger\_set(struct led\_classdev \*led\_cdev, struct led\_trigger \*trigger)；//建立连接。建立连接的时候会调用触发器的**activate方法**

void led\_trigger\_remove(struct led\_classdev \*led\_cdev)；//取消连接。取消连接的时候会调用触发器的**deactivate方法**

void led\_trigger\_set\_default(struct led\_classdev \*led\_cdev)；//在所有已注册的触发器中寻找led\_cdev的默认触发器并调用 led\_trigger\_set建立连接

最后总结一下led、led\_classdev、led\_trigger的关系：



 也就是说trigger好比是控制LED类设备的算法，这个算法决定着LED什么时候亮什么时候暗。LED trigger类设备可以是现实的硬件设备，比如IDE硬盘，也可以是系统心跳等事件。

## 参考资料

<http://blog.csdn.net/yuanlulu/article/details/6438841>

# 固件子系统

request\_firmware() hotplug interface:

------------------------------------

Copyright (C) 2003 Manuel Estrada Sainz

Why:

---

Today, the most extended way to use firmware in the Linux kernel is linking

it statically in a header file. Which has political and technical issues:

1) Some firmware is not legal to redistribute.

2) The firmware occupies memory permanently, even though it often is just

used once.

3) Some people, like the Debian crowd, don't consider some firmware free

enough and remove entire drivers (e.g.: keyspan).

High level behavior (mixed):

============================

1), kernel(driver):

- calls request\_firmware(&fw\_entry, $FIRMWARE, device)

- kernel searches the firmware image with name $FIRMWARE directly

in the below search path of root filesystem:

User customized search path by module parameter 'path'[1]

"/lib/firmware/updates/" UTS\_RELEASE,

"/lib/firmware/updates",

"/lib/firmware/" UTS\_RELEASE,

"/lib/firmware"

- If found, goto 7), else goto 2)

[1], the 'path' is a string parameter which length should be less

than 256, user should pass 'firmware\_class.path=$CUSTOMIZED\_PATH'

if firmware\_class is built in kernel(the general situation)

2), userspace:

- /sys/class/firmware/xxx/{loading,data} appear.

- hotplug gets called with a firmware identifier in $FIRMWARE

and the usual hotplug environment.

- hotplug: echo 1 > /sys/class/firmware/xxx/loading

3), kernel: Discard any previous partial load.

4), userspace:

- hotplug: cat appropriate\_firmware\_image > \

/sys/class/firmware/xxx/data

5), kernel: grows a buffer in PAGE\_SIZE increments to hold the image as it

comes in.

6), userspace:

- hotplug: echo 0 > /sys/class/firmware/xxx/loading

7), kernel: request\_firmware() returns and the driver has the firmware

image in fw\_entry->{data,size}. If something went wrong

request\_firmware() returns non-zero and fw\_entry is set to

NULL.

8), kernel(driver): Driver code calls release\_firmware(fw\_entry) releasing

the firmware image and any related resource.

High level behavior (driver code):

==================================

if(request\_firmware(&fw\_entry, $FIRMWARE, device) == 0)

copy\_fw\_to\_device(fw\_entry->data, fw\_entry->size);

release\_firmware(fw\_entry);

Sample/simple hotplug script:

============================

# Both $DEVPATH and $FIRMWARE are already provided in the environment.

HOTPLUG\_FW\_DIR=/usr/lib/hotplug/firmware/

echo 1 > /sys/$DEVPATH/loading

cat $HOTPLUG\_FW\_DIR/$FIRMWARE > /sys/$DEVPATH/data

echo 0 > /sys/$DEVPATH/loading

Random notes:

============

- "echo -1 > /sys/class/firmware/xxx/loading" will cancel the load at

once and make request\_firmware() return with error.

- firmware\_data\_read() and firmware\_loading\_show() are just provided

for testing and completeness, they are not called in normal use.

- There is also /sys/class/firmware/timeout which holds a timeout in

seconds for the whole load operation.

- request\_firmware\_nowait() is also provided for convenience in

user contexts to request firmware asynchronously, but can't be called

in atomic contexts.

about in-kernel persistence:

---------------------------

Under some circumstances, as explained below, it would be interesting to keep

firmware images in non-swappable kernel memory or even in the kernel image

(probably within initramfs).

Note that this functionality has not been implemented.

- Why OPTIONAL in-kernel persistence may be a good idea sometimes:

- If the device that needs the firmware is needed to access the

filesystem. When upon some error the device has to be reset and the

firmware reloaded, it won't be possible to get it from userspace.

e.g.:

- A diskless client with a network card that needs firmware.

- The filesystem is stored in a disk behind an scsi device

that needs firmware.

- Replacing buggy DSDT/SSDT ACPI tables on boot.

Note: this would require the persistent objects to be included

within the kernel image, probably within initramfs.

And the same device can be needed to access the filesystem or not depending

on the setup, so I think that the choice on what firmware to make

persistent should be left to userspace.

about firmware cache:

--------------------

After firmware cache mechanism is introduced during system sleep,

request\_firmware can be called safely inside device's suspend and

resume callback, and callers needn't cache the firmware by

themselves any more for dealing with firmware loss during system

resume.

在一般的使用中，可能会需要使用到升级固件这个功能，在linux的系统中提供了固件子系统这个设备模型来帮助快速的升级固件。  
  
在这里先将linux中提供的常用的接口来说明下：  
内核的固件接口：

**[cpp]** [view plain](http://blog.csdn.net/skywalkzf/article/details/7479132) [copy](http://blog.csdn.net/skywalkzf/article/details/7479132)

1. #include <linux/firmware.h>
2. **int** request\_firmware(**const** **struct** firmware \*\*firmware\_p, **const** **char** \*name,
3. **struct** device \*device)
4. //request\_firmware()调用要求用户空间为内核定位并提供一个固件映像文件。
5. //firmware\_p：指向firmware image的指针；
6. //name：firmware文件名称；
7. //device：将要加载firmware的设备；
8. /\*\*
9. \*      @name will be used as $FIRMWARE in the uevent environment and
10. \*      should be distinctive enough not to be confused with any other
11. \*      firmware image for this or any other device.
12. \*\*/

把固件发送到设备后，需要使用下面的函数释放内核中的结构：

**[cpp]** [view plain](http://blog.csdn.net/skywalkzf/article/details/7479132) [copy](http://blog.csdn.net/skywalkzf/article/details/7479132)

1. **void** release\_firmware(**struct** firmware \*fw);

由于request\_firmware()需要用户空间的操作，因此在返回前他将保持睡眠状态。如果当驱动程序必须要使用固件，而又不能进入睡眠时，可以使用下面的异步函数：

**[cpp]** [view plain](http://blog.csdn.net/skywalkzf/article/details/7479132) [copy](http://blog.csdn.net/skywalkzf/article/details/7479132)

1. **int** request\_firmware\_nowait(**struct** module \*module, **char** \*name, **struct** device \*device, **void** \*context, **void** (\*cont)(**const** **struct** firmware \*fw, **void** \*context));

固件子系统使用sysfs和热插拔机制工作。当调用request\_firmware的时候，在/sys/class/firmware下降创建一个目录，该设备使用设备名作为它的目录名。该目录包含三个属性：  
loading：  
     该属性由负责加载固件的用户空间进程设置为1.当装载过程完毕时，他将设置为0.将loading设置为-1，将终止固件装载过程。  
data：  
    data是一个二进制属性，用来接收固件数据。在设置完loading后，用户空间进程将把固件写入该属性；  
device：  
    该属性是到/sys/devices下相应入口的符号链接；  
  
  
了解了基本的函数说明之后，这里要了解利用系统的固件子系统来进行升级的步骤：  
**1、从内核层要发送请求到应用层，这里就要用到我们所说的request\_firmware来实现这个步骤；  
2、从应用层copy固件到内核层，这里就是应用层的编写，简单的open，write；  
3、内核将接收到的固件通过固有的协议写入硬件设备中；**  
这三个步骤就可以来完成此次的升级。如果只是简单的使用的话，这里可以有一个简单的demo例子可以给你，你只要把这段代码改成你的即可。  
例子：

**[cpp]** [view plain](http://blog.csdn.net/skywalkzf/article/details/7479132) [copy](http://blog.csdn.net/skywalkzf/article/details/7479132)

1. retval = request\_firmware(&cust\_firmware, devices->fw\_fname, dev);
2. **if** (retval < 0) {
3. pr\_err("%s: %s request firmware failed(%d)\n", \_\_func\_\_,
4. devices->fw\_fname, retval);
5. } **else** {
6. /\* check and start upgrade \*/
7. devices\_upgrade\_start();//这里就是自己实现的函数，跟实际的物理硬件相关
9. release\_firmware(cust\_firmware);
10. }

如果只是想使用的话，到了这里就可以结束了，这就是内核中如何的升级固件的接口。如果想看看是如何实现的，可以向下继续讨论。当然，自己理解的也不一定对。  
要分析，这里我们就从使用的函数入口，看下内核中是如何来处理这种事情的。  
request\_firmware()  
    ---->fw\_get\_builtin\_firmware() //检查是否在\_\_start\_builtin\_fw和\_\_end\_builtin\_fw之间有没有指定的firmware  
---->fw\_create\_instance() //通过device\_create\_bin\_file来在sys下创建更新固件的接口  
---->kobject\_uevent() //通过kobject来上报uevent通知应用层需要更新firmware  
---->wait\_for\_completion //等待应用加载万firmware完成；  
  
当然看到了这里我们就可以了解到这里只是简单的将firmware的uevent事件上报给应用层来处理，这里我们来追踪下是谁在接收这个event事件。  
这里我们要看一下系统中的init.c文件。至于为什么要看这个文件，因为我们从底层的request\_firmware可以看到是通过kobject uevent来通知应用层，底层有需求，应用就需要满足需求。  
而uevent的监听的建立则是在init进程中创建的。

**[cpp]** [view plain](http://blog.csdn.net/skywalkzf/article/details/7479132) [copy](http://blog.csdn.net/skywalkzf/article/details/7479132)

1. ueventd\_main
2. ---->device\_init()
3. ----->open\_uevent\_socket()
4. -----> s = socket(PF\_NETLINK, SOCK\_DGRAM, NETLINK\_KOBJECT\_UEVENT); //创建一个socket来监听KOBJECT\_UEVENT；
5. ---->handle\_device\_fd(); //调用uevent的NETLINK\_KOBJECT\_UEVENT的socket处理函数
6. -----> parse\_event(msg, &uevent);//解析上传的uevent数据；
7. ----->handle\_firmware\_event(); //开始处理firmware的event
8. ----->process\_firmware\_event();//打开dev的attr结点
9. ---->load\_firmware();//下载firmware
10. ---->write(loading\_fd, "1", 1);  /\* start transfer \*/
11. ---->写入firmware
12. ----> **if**(!ret)
13. write(loading\_fd, "0", 1);  /\* successful end of transfer \*/
14. **else**
15. write(loading\_fd, "-1", 2); /\* abort transfer \*/

到了这里，可以看到，整个用户空间的下载的过程就结束了。  
下面来看下上面提到的在kernel中如何在sys中建立bin\_attr的过程，这个过程其实是封装在request\_firmware中的。  
当然你也可以采用下面的这个办法来自己建立，当然这个过程就需要应用来主动发起，而不是驱动主动发起。

1. **static** ssize\_t firmware\_write(**struct** kobject \*kobj,
2. **struct** bin\_attribute \*bin\_attr,
3. **char** \*buf, loff\_t pos, **size\_t** size)
4. {
5. **struct** device \*dev = container\_of(kobj, **struct** device, kobj);
6. **struct** cust\_devices \*ts = dev\_get\_drvdata(dev);
7. LOCK(ts->mutex);
8. //处理firmware的写操作，这个就涉及到相关的与具体的硬件来交互；
9. UNLOCK(ts->mutex);
10. **return** size;
11. }

14. **static** ssize\_t firmware\_read(**struct** kobject \*kobj,
15. **struct** bin\_attribute \*ba,
16. **char** \*buf, loff\_t pos, **size\_t** size)
17. {
18. **int** count = 0;
19. u8 reg\_data;
20. **struct** device \*dev = container\_of(kobj, **struct** device, kobj);
21. **struct** cust\_devices \*ts = dev\_get\_drvdata(dev);

24. LOCK(ts->mutex);
25. //读取底层的firmware mode之类的属性，可以在这里实现；
26. UNLOCK(ts->mutex);
27. **return** count;
28. }

31. **static** **struct** bin\_attribute cust\_devices\_firmware = {
32. .attr = {
33. .name = "firmware",
34. .mode = 0644,
35. },
36. .size = XXX, //是否可以超过PAGE\_SIZE？？？？
37. .read = firmware\_read,
38. .write = firmware\_write,
39. };
40. **struct** bin\_attribute {
41. **struct** attribute    attr;
42. **size\_t**          size;
43. **void**            \***private**;
44. ssize\_t (\*read)(**struct** file \*, **struct** kobject \*, **struct** bin\_attribute \*,
45. **char** \*, loff\_t, **size\_t**);
46. ssize\_t (\*write)(**struct** file \*,**struct** kobject \*, **struct** bin\_attribute \*,
47. **char** \*, loff\_t, **size\_t**);
48. **int** (\*mmap)(**struct** file \*, **struct** kobject \*, **struct** bin\_attribute \*attr,
49. **struct** vm\_area\_struct \*vma);
50. };

53. **if** (sysfs\_create\_bin\_file(&dev->kobj, &devices\_firmware))
54. printk(KERN\_ERR "%s: unable to create file\n",
55. \_\_func\_\_);

Ps：request\_firmware 其实包含两部分搜素固件，开始时去内核firmware下搜素固件，这一般用于SOC IP核固件加载，因为SOC一般要在内核启动阶段就得准备好。在内核firmware目录下找不到固件，再发uevent，等待应用加载固件，上层uevent则拥有一个默认的搜索路径，大致位置在firmware\_dirs[]里面存放。

## 参考资料

<https://www.kernel.org/doc/Documentation/firmware_class/README>

http://blog.csdn.net/skywalkzf/article/details/7479132

# [/sys目录下各个子目录的具体说明](http://blog.chinaunix.net/uid-27177626-id-3585189.html)

|  |  |
| --- | --- |
| /sys下的子目录 | 内容 |
| /sys/devices | 该目录下是全局设备结构体系，包含所有被发现的注册在各种总线上的各种物理设备。一般来说，所有的物理设备都按其在总线上的拓扑结构来显示，但有两个例外，即platform devices和system devices。platform devices一般是挂在芯片内部的高速或者低速总线上的各种控制器和外设，它们能被CPU直接寻址；system devices不是外设，而是芯片内部的核心结构，比如CPU，timer等，它们一般没有相关的驱动，但是会有一些体系结构相关的代码来配置它们。  (sys/devices是内核对系统中所有设备的分层次表达模型，也是/sys文件系统管理设备的最重要的目录结构) |
| sys/dev | 该目录下维护一个按照字符设备和块设备的主次号码(major:minor)链接到真是设备(/sys/devices)的符号链接文件。 |
| /sys/class | 该目录下包含所有注册在kernel里面的设备类型，这是按照设备功能分类的设备模型，每个设备类型表达具有一种功能的设备。每个设备类型子目录下都是这种哦哦那个设备类型的各种具体设备的符号链接，这些链接指向/sys/devices/name下的具体设备。设备类型和设备并没有一一对应的关系，一个物理设备可能具备多种设备类型；一个设备类型只表达具有一种功能的设备，比如：系统所有输入设备都会出现在/sys/class/input之下，而不论它们是以何种总线连接到系统的。(/sys/class也是构成linux统一设备模型的一部分) |
| /sys/block | 该目录下的所有子目录代表着系统中当前被发现的所有块设备。按照功能来说防止在/sys/class下会更合适，但由于历史遗留因素而一直存在于/sys/block，但从linux2.6.22内核开始这部分就已经标记为过去时，只有打开了CONFIG\_SYSFS\_DEPRECATED配置编译才会有这个目录存在，并且其中的内容在从linux2.6.26版本开始已经正式移到了/sys/class/block，旧的接口/sys/block为了向后兼容而保留存在，但其中的内容已经变为了指向它们在/sys/devices/中真实设备的符号链接文件。 |
| /sys/bus | 该目录下的每个子目录都是kernel支持并且已经注册了的总线类型。这是内核设备按照总线类型分层放置的目录结构，/sys/devices中的所有设备都是连接于某种总线之下的，bus子目录下的每种具体总线之下可以找到每个具体设备的符号链接，  一般来说每个子目录(总线类型)下包含两个子目录，一个是devices，另一个是drivers；其中devices下是这个总线类型下的所有设备，这些设备都是符号链接，它们分别指向真正的设备(/sys/devices/name/下)；而drivers下是所有注册在这个总线上的驱动，每个driver子目录下 是一些可以观察和修改的driver参数。  (它也是构成linux统一设备模型的一部分) |
| /sys/fs | 按照设计，该目录使用来描述系统中所有的文件系统，包括文件系统本身和按照文件系统分类存放的已挂载点。 |
| /sys/kernel | 这个目录下存放的是内核中所有可调整的参数 |
| /sys/firmware | 该目录下包含对固件对象(firmware object)和属性进行操作和观察的接口，即这里是系统加载固件机制的对用户空间的接口.(关于固件有专用于固件加载的一套API) |
| /sys/hypervisor | 该目录是与虚拟化Xen相关的装置。(Xen是一个开放源代码的虚拟机监视器) |
| /sys/module | 该目录下有系统中所有的模块信息，不论这些模块是以内联(inlined)方式编译到内核映像文件中还是编译为外模块(.ko文件)，都可能出现在/sys/module中。即module目录下包含了所有的被载入kernel的模块。 |
| /sys/power | 该目录是系统中的电源选项，对正在使用的power子系统的描述。这个目录下有几个属性文件可以用于控制整个机器的电源状态，如可以向其中写入控制命令让机器关机/重启等等。 |

## 参考资料

<http://blog.chinaunix.net/uid-27177626-id-3585189.html>

# U-boot如何向kernel传递参数 + kernel如何读取参数

**UBOOT版本：UBOOT2010.06**

**硬件环境：ARM**

U-boot 会给 linux Kernel 传递很多参数，例如：串口、RAM、commandline（bootargs）等。而 linux kernel 也会读取和处理这些参数。它们两者之间通过ATAG方式来传递参数。

**U-boot 把要传递给 kernel 的数据保存在 struct tag 数据结构中，启动内核时，把这个结构体的物理地址传给内核，然后内核通过这个地址，用 parse\_tags 分析出传递过来的参数。**

这里以 U-boot 传递 RAM 参数和Linux **kernel** 读取 RAM 参数为例进行介绍。

**1、u-boot 向kernel 传递RAM 参数**

./common/cmd\_bootm.c 文件调用 ./uboot/arch/arm/lib/bootm.c 文件中的 do\_nand\_boot函数来启动 Linux kernel。

    在do\_nand\_boot 函数中：

int do\_nand\_boot (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

       DECLARE\_GLOBAL\_DATA\_PTR;

       int ret;

       bd\_t \*bd = gd->bd;

       ulong addr, data, len, initrd\_start, initrd\_end;

**void (\*theKernel)(int zero, int arch, uint params);**

int strlen;

**char \*commandline = getenv ("bootargs");**

**setup\_start\_tag (bd);    // 初始化第一个kernel tag结构体**

**setup\_serial\_tag (&params);**

**setup\_revision\_tag (&params);**

**setup\_memory\_tags (bd);**

**theKernel** (0,

machid,

bd->bi\_boot\_params  // 传给 Kernel 的参数＝ (struct tag \*) 型的 bd->bi\_boot\_params

);

}

**setup\_start\_tag 和 setup\_memory\_tags 函数说明**

函数 setup\_start\_tag 也在此文件中定义，如下：

static void setup\_start\_tag (bd\_t \*bd)

{

       params = (struct tag \*) bd->bi\_boot\_params;

**/\* 初始化 (struct tag \*) 型的全局变量 params 为bd->bi\_boot\_params 的地址，**

**\* 之后的setup tags 相关函数如下面的 setup\_memory\_tags**

**\* 就把其它 tag 的数据放在此地址的偏移地址上。**

**\*/**

       params->hdr.tag = ATAG\_CORE;

       params->hdr.size = tag\_size (tag\_core);

       params->u.core.flags = 0;

       params->u.core.pagesize = 0;

       params->u.core.rootdev = 0;

       params = tag\_next (params);

}

RAM 相关参数在函数 setup\_memory\_tags 中初始化：

static void setup\_memory\_tags (bd\_t \*bd)

{

       int i;

       for (i = 0; i < CONFIG\_NR\_DRAM\_BANKS; i++) {

              params->hdr.tag = ATAG\_MEM;

              params->hdr.size = tag\_size (tag\_mem32);

              params->u.mem.start = bd->bi\_dram[i].start;

              params->u.mem.size = bd->bi\_dram[i].size;

              params = tag\_next (params);

       }                   // 初始化内存相关 tag

}

**2、** **Kernel 读取U-boot 传递的ATAG参数**

对于 Linux Kernel，ARM 平台启动时，先执行 arch/arm/kernel/head.S ，此文件会调用arch/arm/kernel/head-common.S 中的函数，并最后调用 start\_kernel ：

......

b     start\_kernel

......

init/main.c 中的 start\_kernel 函数中会调用**setup\_arch**函数来处理各种平台相关的动作，包括了 u-boot 传递过来参数的分析和保存：

start\_kernel()

{

......

**setup\_arch(&command\_line);**

......

}

其中， setup\_arch 函数在 arch/arm/kernel/setup.c 文件中实现，如下：

void \_\_init setup\_arch(char \*\*cmdline\_p)

{

       struct tag \*tags = (struct tag \*)&init\_tags;

       struct machine\_desc \*mdesc;

       char \*from = default\_command\_line;

       setup\_processor();

       mdesc = setup\_machine(machine\_arch\_type);

       machine\_name = mdesc->name;

       if (mdesc->soft\_reboot)

              reboot\_setup("s");

       if (\_\_atags\_pointer)

**// 指向各种 tag 起始位置的指针，定义如下：**

**// unsigned int \_\_atags\_pointer  \_\_initdata;**

**// 此指针指向 \_\_initdata 段，各种 tag 的信息保存在这个段中。**

              tags = phys\_to\_virt(\_\_atags\_pointer);

       else if (mdesc->boot\_params)

              tags = phys\_to\_virt(mdesc->boot\_params);

       if (tags->hdr.tag != ATAG\_CORE)

              convert\_to\_tag\_list(tags);

       if (tags->hdr.tag != ATAG\_CORE)

              tags = (struct tag \*)&init\_tags;

       if (mdesc->fixup)

              mdesc->fixup(mdesc, tags, &from, &meminfo);

       if (tags->hdr.tag == ATAG\_CORE) {

              if (meminfo.nr\_banks != 0)

                     squash\_mem\_tags(tags);

              save\_atags(tags);

              parse\_tags(tags);

**// 处理各种 tags ，其中包括了 RAM 参数的处理。**

**// 这个函数处理如下 tags ：**

**\_\_tagtable(ATAG\_MEM, parse\_tag\_mem32);**

\_\_tagtable(ATAG\_VIDEOTEXT, parse\_tag\_videotext);

\_\_tagtable(ATAG\_RAMDISK, parse\_tag\_ramdisk);

\_\_tagtable(ATAG\_SERIAL, parse\_tag\_serialnr);

\_\_tagtable(ATAG\_REVISION, parse\_tag\_revision);

\_\_tagtable(ATAG\_CMDLINE, parse\_tag\_cmdline);

       }

       init\_mm.start\_code = (unsigned long) &\_text;

       init\_mm.end\_code   = (unsigned long) &\_etext;

       init\_mm.end\_data   = (unsigned long) &\_edata;

       init\_mm.brk       = (unsigned long) &\_end;

       memcpy(boot\_command\_line, from, COMMAND\_LINE\_SIZE);

       boot\_command\_line[COMMAND\_LINE\_SIZE-1] = '/0';

**parse\_cmdline(cmdline\_p, from);  // 处理编译内核时指定的 cmdline 或 u-boot 传递的cmdline**

       paging\_init(&meminfo, mdesc);

       request\_standard\_resources(&meminfo, mdesc);

#ifdef CONFIG\_SMP

       smp\_init\_cpus();

#endif

……

}

对于处理 RAM 的 tag ，调用了 parse\_tag\_mem32 函数：

static int \_\_init parse\_tag\_mem32(const struct tag \*tag)

{

......

       arm\_add\_memory(tag->u.mem.start, tag->u.mem.size);

......

}

\_\_tagtable(ATAG\_MEM, parse\_tag\_mem32);

       上述的 arm\_add\_memory 函数定义如下：

static void \_\_init arm\_add\_memory(unsigned long start, unsigned long size)

{

       struct membank \*bank;

       size -= start & ~PAGE\_MASK;

       bank = &meminfo.bank[meminfo.nr\_banks++];

       bank->start = PAGE\_ALIGN(start);

       bank->size  = size & PAGE\_MASK;

       bank->node  = PHYS\_TO\_NID(start);

}

如上可见， parse\_tag\_mem32 函数调用 arm\_add\_memory 函数把 RAM 的 start 和 size等参数保存到了 meminfo 结构的 meminfo 结构体中。最后，在 setup\_arch 中执行下面语句：

    paging\_init(&meminfo, mdesc);

对有 MMU 的平台上调用 arch/arm/mm/nommu.c 中的 paging\_init ，否则调用arch/arm/mm/mmu.c 中的 paging\_init 函数。这里暂不分析 mmu.c 中的 paging\_init 函数。

**总结一下**

简单的讲，uboot利用函数指针及**传参规范**，它将

   R0: 0x0

   R1: 机器号

   R2: 参数地址

三个参数传递给内核。

其中，R2寄存器传递的是一个指针，这个指针指向一个TAG区域。

UBOOT和Linux内核之间正是通过这个扩展了的TAG区域来进行复杂参数的传递，如command line，文件系统信息等等，用户也可以扩展这个TAG来进行更多参数的传递。TAG区域的首地址，正是R2的值。

TAG区域由许多个TAG数组构成，每一个TAG传递一种特定类型的参数。各种TAG的定义可以参考：

arch/arm/include/asm/setup.h。

下面是一个TAG区的例子：

0x30000100 **00000005 54410001 00000000 00000000**

0x30000110 **00000000** **0000000F 54410009 746F6F72**

0x30000120 **65642F3D 61722F76 7220306D 6F632077**

0x30000130 **6C6F736E 74743D65 2C305379 30303639**

0x30000140 **696E6920 6C2F3D74 78756E69 EA006372**

0x30000150 **00000004 54420005 30300040 00200000**

0x30000160 00000000 00000000

我们可以看到一共有三个TAG：

   第一个TAG的长度是5个字，类型是ATAG\_CORE（54410001），有三个元素，均为全零．TAG区必须以这个TAG开头．

   第二个TAG的长度是F个字，类型是ATAG\_CMDLINE（54410009），这是一个字符串，是向内核传递的kernel command line

   第三个TAG的长度是4个字，类型是ATAG\_INITRD2（54410005），有两个元素，第一个是start:30300040（30300000+40），

第二个是size:200000（2M）

   如果说还有第四个TAG，那就是末尾的两个全零，这是TAG结束的标志。

这些TAG是在函数do\_bootm\_linux函数中建立起来的。具体建立哪些TAG，由相应的控制宏决定。具体可以参考相应代码。例子中第一个TAG是起始TAG，如果环境变量中有bootargs，则建立第二个TAG，如果bootm有两个参数（引导文件系统），则会读取文件系统头部的必要信息，建立第三个TAG．

内核启动后，将根据R2寄存器的值找到这些TAG，并根据TAG类型，调用相应的函数进行处理，从而获取内核运行的必要信息。

**附录：**

1、关于U-boot 中的bd 和gd

U-boot 中有一个用来保存很多有用信息的全局结构体——gd\_t （global data 缩写），其中包括了 bd 变量，可以说 gd\_t 结构体包括了 u-boot 中所有重要全局变量。最后传递给内核的参数，都是从 gd 和 bd 中来的，如上述的 setup\_memory\_tags 函数作用就是用 bd 中的值来初始化 RAM 相应的 tag 。

对于 ARM平台这个结构体的定义如下：

include/asm-arm/global\_data.h

typedef  struct  global\_data {

       bd\_t        \*bd;

       unsigned long  flags;

       unsigned long  baudrate;

       unsigned long  have\_console; /\* serial\_init() was called \*/

       unsigned long  reloc\_off;       /\* Relocation Offset \*/

       unsigned long  env\_addr;       /\* Address  of Environment struct \*/

       unsigned long  env\_valid;       /\* Checksum of Environment valid? \*/

       unsigned long  fb\_base;  /\* base address of frame buffer \*/

       void        \*\*jt;        /\* jump table \*/

} gd\_t;

在 U-boot 中使用 gd 结构之前要用先用宏DECLARE\_GLOBAL\_DATA\_PTR来声明。这个宏的定义如下：

include/asm-arm/global\_data.h

#define DECLARE\_GLOBAL\_DATA\_PTR register volatile **gd\_t**\*gd asm ("r8")

从这个宏的定义可以看出，**gd 是一个保存在 ARM 的 r8 寄存器中的 gd\_t 结构体的指针**。

## u-boot怎么传入一个自定义参数给内核？？然后内核又是怎样解析？

uboot里面有一个命令叫linux\_args，是一个字符串描述，uboot启动linux时将linux\_args作为参数传给linux， linux把这个启动参数叫cmdline， 在linux启动过程中，相应的模块会捕获这个cmdline里的内容，一旦发现有自身模块的相关描述， 那么就会按照这些描述来实现初始化。  
  
我的uboot里的linux\_args里有有描述了flash相关有关的内容：  
Dream 5220 # printenv   
linux\_args=mtdparts=octeon\_nor0:1m(BOOT),7m(LINUX),21m(CONFIG),3m(MD)  
  
linux启动后， /mtd/cmdlinepart.c 会捕获到"mtdparts="部分的内容， 之后按照这些描述来初始化flash分区。  
  
在linux启动阶段，start\_kernel 也会把这个cmdline 打印出来：  
Kernel command line:  bootoctlinux 0x1dd00000 coremask=0x1 mtdparts=octeon\_nor0:1m(BOOT),7m(LINUX),21m(CONFIG),3m(MD)  console=t0  
  
你在shell界面下可以查看到这个cmdline的内容：  
root@(none):~# cat /proc/cmdline   
 bootoctlinux mtdparts=octeon\_nor0:1m(BOOT),7m(LINUX),24m(CONFIG) console=ttyS0,115200  
  
你只需要做三件事：  
第一， 在uboot界面下使用linux\_args命令添加你的描述.   
第二， \_\_setup() 函数就是linux的模块里用来捕获cmdline里有关自身描述的。你查一下这个资料。  
第三， 实现你的module.  
  
另外， 你也可以将你的描述信息在uboot里面用现成的命令写到eeprom等设备里面，之后你在linux添加一个module去eeprom里面查找相关的信息，同样可以实现根据不同的参数来init 不同的hw。这个办法也是可行的。

linux\_args 是一个uboot的环境变量，用setenv设置的，比如实现上面的设置，就用：  
setenv linux\_args mtdparts=octeon\_nor0:1m(BOOT),7m(LINUX),21m(CONFIG),3m(MD)

这个args有长度限制的 1024个字节内。

Ps：[/system下的文件为只读... 怎么样remount成为可读写?](https://www.baidu.com/link?url=JY_1TA0EnEQPwNiDo9Ba-py4K0TGhJFa2tHZPNJcTzEuociGvI7-SVFBNrSkdSC-JWif0JxKLSbnUQNye7qSEa&wd=&eqid=c8ccf7a200018fe700000004583e2f5f)

mount –o remount,rw /system

## 参考资料

<http://bbs.csdn.net/topics/390616789/>

<http://zqwt.012.blog.163.com/blog/static/12044684201332535457347/>

# [linux中mdelay()与msleep()的区别](http://blog.csdn.net/cbk861110/article/details/40747139)

1.)对于模块本身

mdelay是忙等待函数，在延迟过程中无法运行其他任务．这个延迟的时间是准确的．是需要等待多少时间就会真正等待多少时间．

msleep是休眠函数，它不涉及忙等待．你如果是msleep(１０)，那实际上延迟的时间，大部分时候是要多于１０ms的，是个不定的时间值．

2.)对于系统：

mdelay() 会占用cpu资源，导致其他功能此时也无法使用cpu资源。

msleep() 则不会占住cpu资源，其他模块此时也可以使用cpu资源。

delay函数是忙则等待，占用CPU时间；而sleep函数使调用的进程进行休眠。

3.)udelay() mdelay() ndelay() 区别：

udelay(); mdelay(); ndelay();实现的原理本质上都是忙等待，ndelay和mdelay都是通过udelay衍生出来的。

我们使用这些函数的实现往往会碰到编译器的警告implicit declaration of function 'udelay'，这往往是由于头文件的使用不当造成的。

在include/asm-\*\*\*/delay.h中定义了udelay（），而在include/[**Linux**](http://lib.csdn.net/base/linux)/delay.h中定义了mdelay和ndelay.

udelay一般适用于一个比较小的delay，如果你填的数大于2000，系统会认为你这个是一个错误的delay函数，因此如果需要2ms以上的delay需要使用mdelay函数。

4.)秒的单位

ms是毫秒=0.001秒

us是微秒=0.000001秒

ns是纳秒=0.000000001秒

## 参考资料

<http://blog.csdn.net/cbk861110/article/details/40747139>

# [用户空间内核空间ipc总结](http://bbs.chinaunix.net/thread-1940094-1-1.html)

多数的 Linux 内核态程序都需要和用户空间的进程交换数据，但 Linux 内核态无法对传统的 Linux 进程间同步和通信的方法提供足够的支持！本文就总结下常见的ipc

## getsockopt/setsockopt：

缺点最终使用**copy\_from\_user()/copy\_to\_user()**完成内核和用户的通信， 效率其实不高， 多用在传递控制 选项 信息，不适合做大量的数据传输

### 用户态函数：

发送：int setsockopt ( int sockfd, int proto, int cmd, void \*data, int datelen);

接收：int getsockopt(int sockfd, int proto, int cmd, void \*data, int datalen)

第一个参数是socket描述符；

第二个参数proto是sock协议，IP RAW的就用SOL\_SOCKET/SOL\_IP等，TCP/UDP socket的可用SOL\_SOCKET/SOL\_IP/SOL\_TCP/SOL\_UDP等，即高层的socket是都可以使用低层socket的命令字 的，IPPROTO\_IP；

第三个参数cmd是操作命令字，由自己定义；

第四个参数是数据缓冲区起始位置指针，set操作时是将缓冲区数据写入内核，get的时候是将内核中的数 据读入该缓冲区；

第五个参数数据长度

### 内核态函数：

注册：nf\_register\_sockopt(struct nf\_sockopt\_ops \*sockops)

解除：nf\_unregister\_sockopt(struct nf\_sockopt\_ops \*sockops)

结构体 nf\_sockopt\_ops test\_sockops

1. **static** **struct** nf\_sockopt\_ops nso = {
2. .pf  = PF\_INET,       // 协议族
3. .set\_optmin = 常数,    // 定义最小set命令字
4. .set\_optmax = 常数+N,  // 定义最大set命令字
5. .set  = recv\_msg,   // 定义set处理函数
6. .get\_optmin = 常数,    // 定义最小get命令字
7. .get\_optmax = 常数+N,  // 定义最大get命令字
8. .get  = send\_msg,   // 定义set处理函数
9. };

其中命令字不能和内核已有的重复，宜大不宜小。命令字很重要，是用来做标识符的。而且用户态和内核态要定义的相同

Ps:我们要做的就是把这个nf\_sockopt\_ops 结构体填充完整，写好.set以及.get即可。

## mmap /shm：

**共享内存是最快的一种进程间通信机制，mmap是其中的一种方式**

共享内存可以通过mmap()映射普通文件（特殊情况下还可以采用匿名映射）机制实现，也可以通过系统V（shm）共享内存机制实现。应用接口和原理很简单，内部机制复杂。为了实现更安全通信，往往还与信号灯等同步机制共同使用。

因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据[1]：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就 解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存 中的内容往往是在解除映射时才写回文件的。

mmap仅在第一次使用时为进程建立页表，也就是将一段物理地址映射到一段虚拟地址上，以后操作时不再检测其地址的合法性（合法性测试由CPU页保护异常来做），另一方面是内 核下直接操作mmap地址，可以不用频繁拷贝，也就是说在内核下直接可用指针向该地址操作，而不再在内核中专门开一个缓冲区，然后将缓冲区中的数据拷贝一次进来,mmap一般是将一段连续的物理地址映射成一段虚拟地址，当然，也可以将每段连续，但各段不连续的物理地址映射成一段连续的虚拟地址，无论如何，其物理地址在每段之中是连续的，这样一来，就不会造成CPU的CACHE频繁失效，从而大大节约时间。

1、mmap保存到实际硬盘，实际存储并没有反映到主存上。

mmap的机制就是在磁盘上建立一个文件，每个进程存储器里面，单独开辟一个空间来进行映射。如果多进程的话，那么不会对实际的物理存储器（主存）消耗太大。

优点：储存量可以很大（多于主存）缺点：进程间读取和写入速度要比主存的要慢。

2、shm保存到物理存储器（主存），实际的储存量直接反映到主存上。每个进程的共享内存都直接映射到实际物理存储器里面。

优点，进程间访问速度（读写）比磁盘要快；缺点，储存量不能非常大（多于主存）

使用上看：如果分配的存储量不大，那么使用shm；如果存储量大，那么使用mmap。

mmap系统调用并不是完全为了用于共享内存而设计的。它本身提供了不同于一般对普通文件的访问方式，进程可以像读写内存一样对普通文件的操作。而Posix或系统V的共享内存IPC则纯粹用于共享目的，当然mmap()实现共享内存也是其主要应用之一。

2）使用特殊文件提供匿名内存映射：适用于具有亲缘关系的进程之间；由于父子进程特殊的亲缘关系，在父进程中先调用mmap()，然后调用fork()。那么在调用fork()之后，子进程继承父进程匿名映射后的地址空间，同样也继承mmap()返回的地址，这样，父子进程就可以通过映射区域进行通信了。

### Mmap的优点

1、对文件的读取操作跨过了页缓存，减少了数据的拷贝次数，用内存读写取代I/O读写，提高了文件读取效率。

2、实现了用户空间和内核空间的高效交互方式。两空间的各自修改操作可以直接反映在映射的区域内，从而被对方空间及时捕捉。

3、提供进程间共享内存及相互通信的方式。不管是父子进程还是无亲缘关系的进程，都可以将自身用户空间映射到同一个文件或匿名映射到同一片区域。从而通过各自对映射区域的改动，达到进程间通信和进程间共享的目的。

同时，如果进程A和进程B都映射了区域C，当A第一次读取C时通过缺页从磁盘复制文件页到内存中；但当B再读C的相同页面时，虽然也会产生缺页异常，但是不再需要从磁盘中复制文件过来，而可直接使用已经保存在内存中的文件数据。

4、可用于实现高效的大规模数据传输。内存空间不足，是制约大数据操作的一个方面，解决方案往往是借助硬盘空间协助操作，补充内存的不足。但是进一步会造成大量的文件I/O操作，极大影响效率。这个问题可以通过mmap映射很好的解决。换句话说，但凡是需要用磁盘空间代替内存的时候，mmap都可以发挥其功效。

共享内存块提供了在任意数量的进程之间进行高效双向通信的机制。每个使用者都可以读取写入数据，但是所有程序之间必须达成并遵守一定的协议，以防止诸如在读取信息之前覆写内存空间等竞争状态的出现。不幸的是，Linux无法严格保证提供对共享内存块的独占访问，甚至是在您通过使用IPC\_PRIVATE创建新的共享内存块的时候也不能保证访问的独占性。 同时，多个使用共享内存块的进程之间必须协调使用同一个键值。

**PS：物理地址（Physical Address）**

**内核空间申请内存的方法：PA：**

**kmalloc()和\_\_get\_free\_pages(): < 896M 且虚拟地址和PA都连续 . 返回的是虚拟地址，-3G = PA**

**vmalloc(): 高端内存分配 > 896M 虚拟地址是连续，PA不一定连续**

## netlink/socket：

netlink socekt是一种用于在内核态和用户态进程之间进行数据传输的特殊的IPC。它通过为内核模块提供一组特殊的API，并为用户程序提供了一组标准的socket 接口的方式，实现了一种全双工的通讯连接。类似于TCP/IP中使用AF\_INET地址族一样，netlink socket使用地址族AF\_NETLINK。每一个netlinksocket在内核头文件

1. include/linux/netlink.h

中定义自己的协议类型。  
下面是netlink socket 目前的特性集合以及它支持的协议类型：

|  |
| --- |
| NETLINK\_ROUTE 用户空间的路由守护程序之间的通讯通道，比如BGP,OSPF,RIP以及内核数据转发模块。  用户态的路由守护程序通过此类型的协议来更新内核中的路由表。 NETLINK\_FIREWALL:接收IPV4防火墙代码发送的数据包。 NETLINK\_NFLOG:用户态的iptables管理工具和内核中的netfilter模块之间通讯的通道。 NETLINK\_ARPD:用来从用户空间管理内核中的ARP表。 |

   为什么以上的功能在实现用户程序和内核程序通讯时，都使用netlink方法而不是系统调用，ioctls或者proc文件系统呢？原因在于：为新的特性添加一个新的系统调用,ioctls或者一个proc文件的做法并不是很容易的一件事情，因为我们要冒着污染内核代码并且可能破坏系统稳定性的风险去完成这件事情。  
然而，netlink socket却是如此的简单，你只需要在文件netlink.h中添加一个常量来标识你的协议类型，然后，内核模块和用户程序就可以立刻使用socket风格的API进行通讯了！Netlink提供了一种异步通讯方式，与其他socket API一样，它提供了一个socket队列来缓冲或者平滑瞬时的消息高峰。发送netlink消息的系统调用在把消息加入到接收者的消息对列后，会触发接收者的接收处理函数。接收者在接收处理函数上下文中，可以决定立即处理消息还是把消息放在队列中，在以后其它上下文去处理它(因为我们希望接收处理函数执行的尽可能快)。系统调用与netlink不同，它需要一个同步的处理，因此，当我们使用一个系统调用来从用户态传递消息到内核时，如果处理这个消息的时间很长的话，内核调度的粒度就会受到影响。  
        内核中实现系统调用的代码都是在编译时静态链接到内核的，因此，在动态加载模块中去包含一个系统调用的做法是不合适的，那是大多数设备驱动的做法。使用netlink socket时,动态加载模块中的netlink程序不会和linux内核中的netlink部分产生任何编译时依赖关系。  
Netlink优于系统调用，ioctls和proc文件系统的另外一个特点就是它支持多点传送。一个进程可以把消息传输给一个netlink组地址，然后任意多个进程都可以监听那个组地址(并且接收消息)。这种机制为内核到用户态的事件分发提供了一种近乎完美的解决方案。  
系统调用和ioctl都属于单工方式的IPC，也就是说，这种IPC会话的发起者只能是用户态程序。但是，如果内核有一个紧急的消息想要通知给用户态程序时，该怎么办呢？如果直接使用这些IPC的话，是没办法做到这点的。通常情况下，应用程序会周期性的轮询内核以获取状态的改变，然而，高频度的轮询势必会增加系统的负载。Netlink 通过允许内核初始化会话的方式完美的解决了此问题，我们称之为netlink socket的双工特性。  
        最后，netlink socket提供了一组开发者熟悉的BSD风格的API函数，因此，相对于使用神秘的系统调用API或者ioctl而言，netlink开发培训的费用会更低些。

与BSD的Routing socket的关系  
在BSD TCP/IP的协议栈实现中，有一种特殊的socket叫做Routing socket.它的地址族为AF\_ROUTE, 协议族为PF\_ROUTE, socket类型为SOCK\_RAW. 这种Routing socket是用户态进程用来向内核中的路由表增加或者删除路由信息用的。在Linux系统中，netlink socket通过协议类型NETLINK\_ROUTE实现了与Routing socket相同的功能，可以说，netlink socket提供了BSD Routing socket功能的超集。

## proc/seq/sys文件系统：

具体实现就是通过对文件系统下的文件进行读写。然后执行读写函数。完成数据的传输

## copy\_from\_user/copy\_to\_user ：

copy\_to\_user 在每次拷贝时需要检测指针的合法性，也就是用户空间的指针所指向的地址的确是一段该进程本身的地址，而不是指向了不属于它的地方，而且每次都会拷贝一次数 据，频繁访问内存，由于虚拟地址连续，物理地址不一定会连续，从而造成CPU的CACHE频繁失效，从而使速度降低；

copy\_to\_user 和 copy\_from\_user都有可能引起阻塞，当包含用户数据的页被换出到硬盘上而不是在物理内存上的时候，这种情况就会发生。此时，进程就会休眠，直到缺页处理程序将该页从硬盘重新换回物理内存[执行copy前你的进程很有可能被剥夺过执行权，被剥夺后如果触发了某种交换的条件，就会把你的进程的相应数据存放到硬盘上，等你的进程重新获得执行权之后，按照linux的原则，在硬盘上的部分只有当真正用到的时候才会放到物理内存，于是拷贝时，你又睡眠了，不过这个时候的task状态跟以前不一样。

## 参考资料

http://bbs.chinaunix.net/thread-1940094-1-1.html

<http://blog.csdn.net/jk110333/article/details/8642261>

<http://www.cnblogs.com/my_life/articles/3469263.html>

<http://www.cnblogs.com/huxiao-tee/p/4660352.html>

<http://blog.chinaunix.net/uid-26669729-id-3077015.html>

<http://lxr.free-electrons.com/source/Documentation/DMA-API-HOWTO.txt>

<http://blog.csdn.net/luckywang1103/article/details/50619251>

ps:

**1. put\_user**

Name

put\_user -- Write a simple value into user space.

Synopsis

put\_user ( x, ptr);

Arguments

x

Value to copy to user space.

ptr

Destination address, in user space.

Context

User context only. This function may sleep.

Description

This macro copies a single simple value from kernel space to user space. It supports simple types like char and int, but not larger data types like structures or arrays.

ptr must have pointer-to-simple-variable type, and x must be assignable to the result of dereferencing ptr .

Returns zero on success, or -EFAULT on error.

**2. get\_user**

Name

get\_user -- Get a simple variable from user space.

Synopsis

get\_user ( x, ptr);

Arguments

x

Variable to store result.

ptr

Source address, in user space.

Context

User context only. This function may sleep.

Description

This macro copies a single simple variable from user space to kernel space. It supports simple types like char and int, but not larger data types like structures or arrays.

ptr must have pointer-to-simple-variable type, and the result of dereferencing ptr must be assignable to x without a cast.

Returns zero on success, or -EFAULT on error. On error, the variable x is set to zero.

# 电源管理

### 手机关机的做法

在手机的做法一般如下   
1，pm\_power\_off()这个函数最终machine 都会实现的。   
2，machine的实现方式大都一样。即控制PMIC。让PMIC给soc断电   
3，下完如上第二点的命令后。虽然控制pmic掉电。但是一般pmic的输出电源都有滤波电容。所以电源是慢慢掉下去的。即下电后cpu还能挣扎执行一些命令   
4，如上第3点。所以一般加一些delay   
5.有的做法还会做一些判断。比如我插入充电器。执行了关机命令。一般充电器插入pmic的时候，即使cpu发命令让pmic下电。pmic也不会掉电。所以如果延时过后会check是否插入充电器。如果插入则重启手机，以便进入关机充电模式。

于是新版本的内核，就将这些Callbacks统一封装为一个数据结构----struct dev\_pm\_ops，上层的数据结构只需要包含这个结构即可。这样如果需要增加或者修改PM callbacks，就不用改动上层结构了（这就是软件设计中抽象和封装的生动体现，像艺术一样优雅）。当然，内核为了兼容旧的设计，也保留了上述的suspend/resume类型的callbacks，只是已不建议使用，本文就不再介绍它们了。

相信每一个熟悉了旧版本内核的Linux工程师，看到struct dev\_pm\_ops时都会虎躯一震，这玩意也太复杂了吧！不信您请看：

1: /\* include/linux/pm.h, line 276 in linux-3.10.29 \*/

2: struct dev\_pm\_ops {

3: int (\*prepare)(struct device \*dev);

4: void (\*complete)(struct device \*dev);

5: int (\*suspend)(struct device \*dev);

6: int (\*resume)(struct device \*dev);

7: int (\*freeze)(struct device \*dev);

8: int (\*thaw)(struct device \*dev);

9: int (\*poweroff)(struct device \*dev);

10: int (\*restore)(struct device \*dev);

11: int (\*suspend\_late)(struct device \*dev);

12: int (\*resume\_early)(struct device \*dev);

13: int (\*freeze\_late)(struct device \*dev);

14: int (\*thaw\_early)(struct device \*dev);

15: int (\*poweroff\_late)(struct device \*dev);

16: int (\*restore\_early)(struct device \*dev);

17: int (\*suspend\_noirq)(struct device \*dev);

18: int (\*resume\_noirq)(struct device \*dev);

19: int (\*freeze\_noirq)(struct device \*dev);

20: int (\*thaw\_noirq)(struct device \*dev);

21: int (\*poweroff\_noirq)(struct device \*dev);

22: int (\*restore\_noirq)(struct device \*dev);

23: int (\*runtime\_suspend)(struct device \*dev);

24: int (\*runtime\_resume)(struct device \*dev);

25: int (\*runtime\_idle)(struct device \*dev);

26: };

从Linux PM Core的角度来说，这些callbacks并不复杂，因为PM Core要做的就是在特定的电源管理阶段，调用相应的callbacks，例如在suspend/resume的过程中，PM Core会依次调用“prepare—>suspend—>suspend\_late—>suspend\_noirq-------wakeup--------->resume\_noirq—>resume\_early—>resume-->complete”。

但由于这些callbacks需要由具体的设备Driver实现，这就要求驱动工程师在设计每个Driver时，清晰的知道这些callbacks的使用场景、是否需要实现、怎么实现，这才是struct dev\_pm\_ops的复杂之处。

Linux kernel对struct dev\_pm\_ops的注释已经非常详细了，但要弄清楚每个callback的使用场景、背后的思考，并不是一件容易的事情。

### ▆ Hibernate（冬眠）和Sleep（睡眠）

是Linux电源管理在用户角度的抽象，是用户可以看到的实实在在的东西。它们的共同点，是保存系统运行的上下文后挂起（suspend）系统，并在系统恢复后接着运行，就像什么事情都没有发生一样。它们的不同点，是上下文保存的位置、系统恢复的触发方式以及具体的实现机制。

### ▆ Suspend

有两个层次的含义。一是Hibernate和Sleep功能在底层实现上的统称，都是指挂起（Suspend）系统，根据上下文的保存位置，可以分为Suspend to Disk（STD，即Hibernate，上下文保存在硬盘/磁盘中）和Suspend to RAM（STR，为Sleep的一种，上下文保存在RAM中）；二是Sleep功能在代码级的实现，表现为“kernel/power/suspend.c”文件。

▆ Standby，是Sleep功能的一个特例，可以翻译为“**打盹**”。

正常的Sleep（STR），会在处理完上下文后，由arch-dependent代码将CPU置为低功耗状态（通常为Sleep）。而现实中，根据对功耗和睡眠唤醒时间的不同需求，CPU可能会提供多种低功耗状态，如除Sleep之外，会提供Standby状态，该状态下，CPU处于浅睡眠模式，有任何的风吹草动，就会立即醒来。

### ▆ Wakeup

这是我们第一次正式的提出Wakeup的概念。我们多次提到恢复系统，其实在内核中称为Wakeup。表面上，wakeup很简单，无论是冬眠、睡眠还是打盹，总得有一个刺激让我们回到正常状态。但复杂的就是，什么样的刺激才能让我们醒来？

动物界，温度回升可能是唯一可以让动物从冬眠状态醒来的刺激。而踢一脚、闹钟响等刺激，则可以让我们从睡眠状态唤醒。对于打盹来说，则任何的风吹草动，都可以唤醒。

而在计算机界，冬眠（Hibernate）时，会关闭整个系统的供电，因此想醒来，唯有Power按钮可用。而睡眠时，为了缩短Wakeup时间，并不会关闭所有的供电，另外，为了较好的用户体验，通常会保留某些重要设备的供电（如键盘），那样这些设备就可以唤醒系统。

这些刻意保留下来的、可以唤醒系统的设备，统称为唤醒源（Wakeup source）。而Wakeup source的选择，则是PM设计工作（特别是Sleep、Standby等功能）的重点。

Linux内核提供了三种Suspend: Freeze、Standby和STR(Suspend to RAM)，在用户空间向”/sys/power/state”文件分别写入”freeze”、”standby”和”mem”，即可触发它们。

## 参考资料

<http://www.wowotech.net/content/uploadfile/201412/bd2e1419231198.pdf>

# linux内核中的GPIO系统

# [自旋锁和互斥锁区别](http://blog.csdn.net/sunmenggmail/article/details/8105279)

从 实现原理上来讲，Mutex属于sleep-waiting类型的锁。例如在一个双核的机器上有两个线程(线程A和线程B)，它们分别运行在Core0和 Core1上。假设线程A想要通过pthread\_mutex\_lock操作去得到一个临界区的锁，而此时这个锁正被线程B所持有，那么线程A就会被阻塞 (blocking)，Core0 会在此时进行上下文切换(Context Switch)将线程A置于等待队列中，此时Core0就可以运行其他的任务(例如另一个线程C)而不必进行忙等待。而Spin lock则不然，它属于busy-waiting类型的锁，如果线程A是使用pthread\_spin\_lock操作去请求锁，那么线程A就会一直在 Core0上进行忙等待并不停的进行锁请求，直到得到这个锁为止。

## 自旋锁(Spin lock)

自旋锁与互斥锁有点类似，只是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是 否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。其作用是为了解决某项资源的互斥使用。因为自旋锁不会引起调用者睡眠，所以自旋锁的效率远 高于互斥锁。虽然它的效率比互斥锁高，但是它也有些不足之处：

1、自旋锁一直占用CPU，他在未获得锁的情况下，一直运行－－自旋，所以占用着CPU，如果不能在很短的时 间内获得锁，这无疑会使CPU效率降低。

2、在用自旋锁时有可能造成死锁，当递归调用时有可能造成死锁，调用有些其他函数也可能造成死锁，如 copy\_to\_user()、copy\_from\_user()、kmalloc()等。

因此我们要慎重使用自旋锁，自旋锁只有在内核可抢占式或SMP的情况下才真正需要，在单CPU且不可抢占式的内核下，自旋锁的操作为空操作。自旋锁适用于锁使用者保持锁时间比较短的情况下。

自旋锁的用法如下：

首先定义：spinlock\_t x;

然后初始化：spin\_lock\_init(spinlock\_t \*x); //自旋锁在真正使用前必须先初始化

在2.6.11内核中将定义和初始化合并为一个宏：DEFINE\_SPINLOCK(x)

获得自旋锁：spin\_lock(x); //只有在获得锁的情况下才返回，否则一直“自旋”

spin\_trylock(x); //如立即获得锁则返回真，否则立即返回假

释放锁：spin\_unlock(x);

结合以上有以下代码段：

spinlock\_t lock; //定义一个自旋锁

spin\_lock\_init(&lock);

spin\_lock(&lock);

....... //临界区

spin\_unlock(&lock); //释放锁

还有一些其他用法：

spin\_is\_locked(x)

//　　该宏用于判断自旋锁x是否已经被某执行单元保持（即被锁），如果是， 返回真，否则返回假。

spin\_unlock\_wait(x)

//　　该宏用于等待自旋锁x变得没有被任何执行单元保持，如果没有任何执行单元保持该自旋锁，该宏立即返回，否

//将循环 在那里，直到该自旋锁被保持者释放。

spin\_lock\_irqsave(lock, flags)

//　　该宏获得自旋锁的同时把标志寄存器的值保存到变量flags中并失效本地中//断。相当于：spin\_lock()+local\_irq\_save()

spin\_unlock\_irqrestore(lock, flags)

//　　该宏释放自旋锁lock的同时，也恢复标志寄存器的值为变量flags保存的//值。它与spin\_lock\_irqsave配对使用。

//相当于：spin\_unlock()+local\_irq\_restore()

spin\_lock\_irq(lock)

　 //该宏类似于spin\_lock\_irqsave，只是该宏不保存标志寄存器的值。相当 //于：spin\_lock()+local\_irq\_disable()

spin\_unlock\_irq(lock)

//该宏释放自旋锁lock的同时，也使能本地中断。它与spin\_lock\_irq配对应用。相当于: spin\_unlock()+local\_irq+enable()

spin\_lock\_bh(lock)

//　　该宏在得到自旋锁的同时失效本地软中断。相当于： //spin\_lock()+local\_bh\_disable()

spin\_unlock\_bh(lock)

//该宏释放自旋锁lock的同时，也使能本地的软中断。它与spin\_lock\_bh配对//使用。相当于：spin\_unlock()+local\_bh\_enable()

spin\_trylock\_irqsave(lock, flags)

//该宏如果获得自旋锁lock，它也将保存标志寄存器的值到变量flags中，并且失//效本地中断，如果没有获得锁，它什么也不做。因此如果能够立即 获得锁，它等//同于spin\_lock\_irqsave，如果不能获得锁，它等同于spin\_trylock。如果该宏//获得自旋锁lock，那需要 使用spin\_unlock\_irqrestore来释放。

spin\_trylock\_irq(lock)

//该宏类似于spin\_trylock\_irqsave，只是该宏不保存标志寄存器。如果该宏获得自旋锁lock，需要使用spin\_unlock\_irq来释放。

spin\_trylock\_bh(lock)

//　　该宏如果获得了自旋锁，它也将失效本地软中断。如果得不到锁，它什么//也不做。因此，如果得到了锁，它等同于spin\_lock\_bh，如果得 不到锁，它等同//于spin\_trylock。如果该宏得到了自旋锁，需要使用spin\_unlock\_bh来释放。

spin\_can\_lock(lock)

//　　该宏用于判断自旋锁lock是否能够被锁，它实际是spin\_is\_locked取反。//如果lock没有被锁，它返回真，否则，返回 假。该宏在2.6.11中第一次被定义，在//先前的内核中并没有该宏。

## 参考资料

http://blog.csdn.net/sunmenggmail/article/details/8105279

# [kmalloc、vmalloc、malloc的区别](http://blog.csdn.net/macrossdzh/article/details/5958368)

**简单的说：**

1. kmalloc和vmalloc是分配的是内核的内存,malloc分配的是用户的内存
2. kmalloc保证分配的内存在物理上是连续的,vmalloc保证的是在虚拟地址空间上的连续,***malloc不保证任何东西(这点是自己猜测的,不一定正确)***
3. kmalloc能分配的大小有限,vmalloc和malloc能分配的大小相对较大
4. 内存只有在要被DMA访问的时候才需要物理上连续
5. vmalloc比kmalloc要慢

## kmalloc函数详解

#include <[**Linux**](http://lib.csdn.net/base/linux)/slab.h> void \*kmalloc(size\_t size, int flags);

最一般使用的标志,

**GFP\_KERNEL**, 意思是这个分配((内部最终通过调用 \_\_get\_free\_pages 来进行, 它是 GFP\_ 前缀的来源) 代表运行在内核空间的进程而进行的. 换句话说, 这意味着调用函数是代表一个进程在执行一个系统调用. 使用 GFP\_KENRL 意味着 kmalloc 能够使当前进程在少内存的情况下睡眠来等待一页. 一个使用 GFP\_KERNEL 来分配内存的函数必须, 因此, 是可重入的并且不能在原子上下文中运行. 当当前进程睡眠, 内核采取正确的动作来定位一些空闲内存, 或者通过刷新缓存到磁盘或者交换出去一个用户进程的内存.

GFP\_KERNEL 不一直是使用的正确分配标志; 有时 kmalloc 从一个进程的上下文的外部调用. 例如, **这类的调用可能发生在中断处理, tasklet, 和内核定时器中.**

**在这个情况下, 当前进程不应当被置为睡眠, 并且驱动应当使用一个 GFP\_ATOMIC 标志来代替.**

内核正常地试图保持一些空闲页以便来满足原子的分配. 当使用 GFP\_ATOMIC 时, kmalloc 能够使用甚至最后一个空闲页. 如果这最后一个空闲页不存在, 但是, 分配失败.

其他用来代替或者增添 GFP\_KERNEL 和 GFP\_ATOMIC 的标志, 尽管它们 2 个涵盖大部分设备驱动的需要. 所有的标志定义在 <linux/gfp.h>

**kmalloc 能够分配的内存块的大小有一个上限. 这个限制随着体系和内核配置选项而变化. 如果你的代码是要完全可移植, 它不能指望可以分配任何大于 128 KB.**

alloc\_page( )可以分配的最大连续页面是4K

static inline struct page \* alloc\_pages(unsigned int gfp\_mask, unsigned int order)

{

/\*

\* Gets optimized away by the compiler.

\*/

if (order >= MAX\_ORDER)

return NULL;

return \_alloc\_pages(gfp\_mask, order);

}

**alloc\_pages最大分配页面数为512个，则可用内存数最大为2^9\*4K=2M**

## kzalloc与kmalloc区别

这个函数就是原来的两个函数的整合 , 即原来我们每次申请内存的时候都会这么做 , 先是用 kmalloc() 申请空间 , 然后用 memset() 来初始化 , 而现在省事了 , 一步到位 , 直接调用 kzalloc(), 效果等同于原来那两个函数 , 所有申请的元素都被初始化为 0. 其实对写驱动的来说 , 知道现在应该用 kzalloc() 代替原来的 kmalloc() 和 memset() 就可以了 , 这是内核中内存管理部分做出的改变 , 确切的说是改进 , 负责内存管理那部分的兄弟们的目标无非就是让内核跑起来更快一些 , 而从 kmalloc/memset 到 kzalloc 的改变确实也是为了实现这方面的优化

## 参考资料

<http://blog.csdn.net/macrossdzh/article/details/5958368>

http://blog.csdn.net/xujianqun/article/details/6715243