驱动学习整理

### 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

**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





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link

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

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如下表所示：

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)

### 从面向对象的角度来看：

- 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；

- 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link

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

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；**将控制权转交给MBR**

**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：延申  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。

         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。



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

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

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

1. 用于抽象Platform设备的数据结构----“struct platform\_device”：
2. 用于抽象Platform设备驱动的数据结构----“struct platform\_driver”：
3. 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。

参考文献

<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 1.1  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。

**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。