Tekkaman Ninja

tekkamanninja.blog.chinaunix.net

tekkamanninj

博客访问： 75931
博文数量： 263
博客积分： 15936
博客等级： 上将
技术积分： 13951
用 户 组： 普通用户
注册时间： 2007-03-27 11:22

加关注 短消息
论坛 加好友

个人简介

Fedora-ARM

文章分类

文章存档

Linux设备驱动程序学习（12）-Linux设备模型（底层原理简介）

2007-12-19 11:57:31

分类： LINUX

Linux设备驱动程序学习（12）
-Linux设备模型（底层原理简介）

以《LDD3》的说法：Linux设备模型这部分内容可以认为是高级教材，对于多数程序作者来说是不必要的。但是我个人认为：对于一个嵌入式Linux的底层程序员来说，这部分内容是很重要的。以我学习的ARM9为例，有很多总线（如SPI、IIC、IIS等等）在Linux下已经被编写成了子系统，无需自己写驱动；而这些总线又不像PCI、USB等在《LDD3》上有教程，有时还要自己研究它的子系统构架，甚至要自己添加一个新的总线类型。

对于这方面的学习，我推荐几个网页，这些也是我这部分文章的参考资料：

（1）《 Linux那些事儿 之 我是Sysfs》来源于复旦和交大三个牛人的Linux技术博客：http://blog.csdn.net/fudan_abc （复旦_abc）他们还分析了很多Linux的驱动，值得珍藏！

（2）《linux设备模型详解》也是一个牛人的博客文章，博客网址：http://hi.baidu.com/csdeny/blog

（3）《s3c2410设备的注册》是一篇关于2410中linux内核实现设备模型的不可多得的好资料。网址：http://blog.chinaunix.net/u1/41638/showart_438078.html

（4）luofuchong的博客 ，此人分析了一些2410中的Linux子系统（如SPI，input等），实力不凡，值得关注。网址：http://www.cnitblog.com/luofuchong/

在这部分的学习中，将会先研究linux设备模型的每个元素，最后将其一步一步整合，至底向上地分析。一开始会比较摸不着头脑，到了整合阶段就柳暗花明了。我之所以没有先介绍整体，再分析每个部分是因为如果不对每个元素做认真分析，看了整体也会云里雾里（我试过了，恕小生愚钝）。所以一开始要耐着性子看，到整合阶段就会豁然开朗。

Linux设备模型的目的是：为内核建立起一个统一的设备模型，从而有一个对系统结构的一般性抽象描述。

现在内核使用设备模型支持多种不同的任务：

电源管理和系统关机 ：这些需要对系统结构的理解，设备模型使OS能以正确顺序遍历系统硬件。

与用户空间的通讯 ：sysfs 虚拟文件系统的实现与设备模型的紧密相关，并向外界展示它所表述的结构。向用户空间提供系统信息、改变操作参数的接口正越来越多地通过 sysfs ， 也就是设备模型来完成。

热插拔设备

设备类型：设备模型包括了将设备分类的机制，在一个更高的功能层上描述这些设备，并使设备对用户空间可见。

对象生命周期：设备模型的实现需要创建一系列机制来处理对象的生命周期、对象间的关系和对象在用户空间的表示。

Linux 设备模型是一个复杂的数据结构。但对模型的大部分来说，Linux 设备模型代码会处理好这些关系，而不是把他们强加于驱动作者。模型隐藏于交互的背后，与设备模型的直接交互通常由总线级的逻辑和其他的内核子系统处理。所以许多驱动作者可完全忽略设备模型，并相信设备模型能处理好他所负责的事。

2013年（3）
2012年（61）
2011年（66）
2010年（27）
2009年（30）
2008年（23）
2007年（52）

我的朋友

小蜗牛快 cfm5538 jikaishi
shizhenc pxy05215 李怀远
yan19900 wkm81018 xiousi

最近访客

apang199 appcount zaicichu
lhxzui 小蜗牛快 小尾巴鱼
erain_30 hushup wilfred_

订阅

推荐博文

- linux 3.x的 通用时钟架构 ...
- SCN的相关解析
- Flash驱动学习
- 浅谈nagios之state type和 no...
- DB2（Linux 64位）安装教程...
- insert语句造成latch:library...
- 2014.06.13 网络公开课《让我...
- MySQL Slave异常关机的处理（...
- 巧用shell脚本分析数据库用户...
- 查询linux,HP-UX的cpu信息...

热词专题

- linux系统权限修复——学生误...
- Modbus协议使用
- linux
- busybox原理
- php环境搭建教程

在此之前请先了解一下sysfs，请看 Linux那些事儿之我是Sysfs(1)sysfs初探 我就不在这里废话了！这里还建议先看看 sysfs 的内核文档\Documentation\filesystems\sysfs.txt，我将其翻译好做成PDF，下载地址：http://blogimg.chinaunix.net/blog/upfile2/071229162826.pdf

如有错误欢迎指正！

Kobject、Kset 和 Subsystem

Kobjects

kobject是一种数据结构，定义在 <linux/kobject.h> 。

```
struct kobject {
    const char     * k_name;/*kobject 的名字数组（sysfs 入口使用的名字）指针；如果名字数组大小小于KOBJ_NAME_LEN，它指向本数组的name,否则指向另外分配的一个名字数组空间 */
    char            name[KOBJ_NAME_LEN];/*kobject 的名字数组，若名字数组大小不小于KOBJ_NAME_LEN，只储存前KOBJ_NAME_LEN个字符*/
    struct kref        kref;/*kobject 的引用计数*/
    struct list_head     entry;/*kobject 之间的双向链表，与所属的kset形成环形链表*/
    struct kobject        * parent;/*在sysfs分层结构中定位对象，指向上一级kset中的struct kobject kobj*/
    struct kset        * kset;/*指向所属的kset*/
    struct kobj_type     * ktype;/*负责对该kobject类型进行跟踪的struct kobj_type的指针*/
    struct dentry         * dentry;/*sysfs文件系统中与该对象对应的文件节点路径指针*/
    wait_queue_head_t     poll;/*等待队列头*/
};
```

kobject 是组成设备模型的基本结构，初始它只被作为一个简单的引用计数，但随时间的推移,其任务越来越多。现在kobject 所处理的任务和支持代码包括:

对象的引用计数 ：跟踪对象生命周期的一种方法是使用引用计数。当没有内核代码持有该对象的引用时，该对象将结束自己的有效生命期并可被删除。

sysfs 表述：在 sysfs 中出现的每个对象都对应一个 kobject，它和内核交互来创建它的可见表述。

数据结构关联：整体来看，设备模型是一个极端复杂的数据结构，通过其间的大量链接而构成一个多层次的体系结构。kobject 实现了该结构并将其聚合在一起。

热插拔事件处理 ：kobject 子系统将产生的热插拔事件通知用户空间。

一个kobject对自身并不感兴趣，它存在的意义在于把高级对象连接到设备模型上。因此内核代码很少(甚至不知道)创建一个单独的 kobject；而kobject 被用来控制对大型域（domain）相关对象的访问，所以 kobject 被嵌入到其他结构中。kobject 可被看作一个最顶层的基类，其他类都它的派生产物。 kobject 实现了一系列方法，对自身并没有特殊作用，而对其他对象却非常有效。

对于给定的kobject指针，可使用container_of宏得到包含它的结构体的指针。

kobject 初始化

kobject的初始化较为复杂，但是必须的步骤如下：

（1）将整个kobject清零，通常使用memset函数。

（2）调用kobject_init()函数，设置结构内部一些成员。所做的一件事情是设置kobject的引用计数为1。具体的源码如下：

```
void kobject_init(struct kobject * kobj)/*in kobject.c*/
{
    if (!kobj)
        return;
    kref_init(&kobj->kref);/*设置引用计数为1*/
    INIT_LIST_HEAD(&kobj->entry);/*初始化kobject 之间的双向链表*/
    init_waitqueue_head(&kobj->poll);/*初始化等待队列头*/
    kobj->kset = kset_get(kobj->kset);/*增加所属kset的引用计数（若没有所属的kset，则返回NULL）*/
}
```

```
void kref_init(struct kref *kref)/*in kobject.c*/
{
    atomic_set(&kref->refcount,1);
    smp_mb();
}

static inline struct kset * to_kset(struct kobject * kobj)/*in kobject.h*/
{
    return kobj ? container_of(kobj,struct kset,kobj) : NULL;
}

static inline struct kset * kset_get(struct kset * k)/*in kobject.h*/
{
    return k ? to_kset(kobject_get(&k->kobj)) : NULL;/*增加引用计数*/
}
```

（3）设置kobject的名字

```
int kobject_set_name(struct kobject * kobj, const char * fmt, ...);
```

（4）直接或间接设置其它成员：ktype、kset和parent。 （重要）

对引用计数的操作

kobject 的一个重要函数是为包含它的结构设置引用计数。只要对这个对象的引用计数存在，这个对象(和支持它的代码）必须继续存在。底层控制 kobject 的引用计数的函数有：

```
struct kobject *kobject_get(struct kobject *kobj);/*若成功，递增 kobject 的引用计数并返回一个指向 kobject 的指针，否则返回 NULL。必须始终测试返回值以免产生竞态*/
void kobject_put(struct kobject *kobj);/*递减引用计数并在可能的情况下释放这个对象*/
```

注意：kobject _init 设置这个引用计数为 1，因此创建一个 kobject时，当这个初始化引用不再需要，应当确保采取 kobject_put 调用。同理：struct cdev 的引用计数实现如下：

```
struct kobject *cdev_get(struct cdev *p)
{
 struct module *owner = p->owner;
 struct kobject *kobj;
 if (owner && !try_module_get(owner))
 return NULL;
 kobj = kobject_get(&p->kobj);
 if (!kobj)
 module_put(owner);
 return kobj;
}
```

创建一个对 cdev 结构的引用时，还需要创建包含它的模块的引用。因此，cdev_get 使用 try_module_get 来试图递增这个模块的使引用计数。如果这个操作成功，kobject_get 被同样用来递增 kobject 的引用计数。kobject_get 可能失败，因此这个代码检查 kobject_get 的返回值，如果调用失败，则释放它的对模块的引用计数。

release 函数和 kobject 类型

引用计数不由创建 kobject 的代码直接控制，当 kobject 的最后引用计数消失时，必须异步通知，而后kobject中ktype所指向的kobj_type结构体包含的release函数会被调用。通常原型如下：

```
void my_object_release(struct kobject *kobj)
{
 struct my_object *mine = container_of(kobj, struct my_object, kobj);
/* Perform any additional cleanup on this object, then... */
 kfree(mine);
}
```

每个 kobject 必须有一个release函数，并且这个 kobject 必须在release函数被调用前保持不变(稳定

状态 ）。这样，每一个 kobject 需要有一个关联的 kobj_type 结构，指向这个结构的指针能在 2 个不同的地方找到：

（1）kobject 结构自身包含一个成员(ktype)指向kobj_type ；

（2）如果这个 kobject 是一个 kset 的成员，kset 会提供kobj_type 指针。

```
struct kset {
    struct kobj_type    * ktype; /*指向该kset对象类型的指针*/
    struct list_head    list;/*用于连接该kset中所有kobject以形成环形链表的链表头*/
    spinlock_t          list_lock;/*用于避免竞态的自旋锁*/
    struct kobject          kobj; /*嵌入的kobject*/
    struct kset_uevent_ops     * uevent_ops;
/*原有的struct kset_hotplug_ops * hotplug_ops;已经不存在，被kset_uevent_ops 结构体替换，在热插拔操作中会介绍*/
};
```

以下宏用以查找指定kobject的kobj_type 指针：

```
struct kobj_type *get_ktype(struct kobject *kobj);
```

这个函数其实就是从以上提到的这两个地方返回kobj_type指针，源码如下：

```
static inline struct kobj_type * get_ktype(struct kobject * k)
{
    if (k->kset && k->kset->ktype)
        return k->kset->ktype;
    else
        return k->ktype;
}
```

关于新版本内核的kobj_type ：

在新版本的内核中已经在struct kset中去除了 struct kobj_type * ktype;
也就是说只有struct kobject中存在 kobj_type ,所以get_ktype也相应变为了：

```
static inline struct kobj_type * get_ktype(struct kobject * kobj)
{
       return kobj->ktype;
}
```

---

kobject 层次结构、kset 和子系统

内核通常用kobject 结构将各个对象连接起来组成一个分层的结构体系，与模型化的子系统相匹配。有 2 个独立的机制用于连接：parent 指针和 kset。

parent 是指向另外一个kobject 结构（分层结构中上一层的节点）的指针，主要用途是在 sysfs 层次中定位对象.

kset

kset 象 kobj_type 结构的扩展；一个 kset 是嵌入到相同类型结构的 kobject 的集合。但 struct kobj_type 关注的是对象的类型，而struct kset 关心的是对象的聚合和集合，其主要功能是包容，可认为是kobjects 的顶层容器类。每个 kset 在内部包含自己的 kobject，并可以用多种处理kobject 的方法处理kset。 ksets 总是在 sysfs 中出现；一旦设置了 kset 并把它添加到系统中，将在 sysfs 中创建一个目录；kobjects 不必在 sysfs 中表示，但kset中的每一个 kobject 成员都要在sysfs中表述。

增加 kobject 到 kset 中去，通常是在kobject 创建时完成，其过程分为2步：

（1）完成kobject的初始化，特别注意mane和parent和初始化。

（2）把kobject 的 kset 成员指向目标kset。

（3）将kobject 传递给下面的函数：

```
int kobject_add(struct kobject *kobj); /*函数可能失败(返回一个负错误码),程序应作出相应地反应*/
```

内核提供了一个组合函数：

```
extern int kobject_register(struct kobject *kobj); /*仅仅是一个 kobject_init 和 kobject_add 的结合，其他成员的初始化必须在之前手动完成*/
```

当把一个kobject从kset中删除以清除引用时使用：

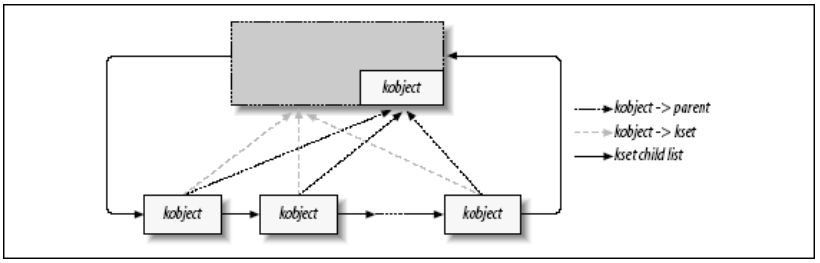
```
void kobject_unregister(struct kobject *kobj); /*是 kobject_del 和 kobject_put 的结合*/
```

但是在新的内核中已经没有"register"和"unregister"类似的函数了,取而代之的是类似

```
extern int __must_check kobject_init_and_add(struct kobject *kobj,
                     struct kobj_type *ktype,
                     struct kobject *parent,
                     const char *fmt, ...);
```

这样的函数,所以请根据你使用的内核版本自己研究了.

kset 在一个标准的内核链表中保存了它的子节点，在大部分情况下，被包含的 kobjects 在它们的 parent 成员中保存指向 kset内嵌的 kobject的指针，关系如下：



图表中所有被包含的 kobjects 实际上被嵌入在一些其他类型中， 甚至可能是其他的 kset。

kset 上的操作

ksets 有类似于kobjects初始化和设置接口:

```
void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);

/*管理 ksets 的引用计数:*/
struct kset *kset_get(struct kset *kset);
void kset_put(struct kset *kset);

/* kset 也有一个名字,存储于嵌入的 kobject,因此设置它的名字用:*/
kobject_set_name(&my_set->kobj, "The name");
```

ksets 还有一个指针指向 kobj_type 结构来描述它包含的 kobject，这个类型优先于 kobject 自身中的 ktype 。因此在典型的应用中， 在 struct kobject 中的 ktype 成员被设为 NULL， 而 kset 中的ktype 是实际被使用的。
在新的内核里， kset 不再包含一个子系统指针struct subsystem * subsys， 而且subsystem已经被kset取代。

子系统

子系统是对整个内核中一些高级部分的表述。子系统通常(但不一定)出现在 sysfs分层结构中的顶层，内核子系统包括 block_subsys(/sys/block 块设备)、 devices_subsys(/sys/devices 核心设备层)以及内核已知的用于各种总线的特定子系统。

对于新的内核已经不再有subsystem数据结构了，用kset代替了。每个 kset 必须属于一个子系统，子系统成员帮助内核在分层结构中定位 kset 。

```
/*子系统通常用以下的宏声明:*/
decl_subsys(name, struct kobj_type *type, struct kset_uevent_ops * uevent_ops);

/*子系统的操作函数:*/
void subsystem_init(struct kset *s);
int subsystem_register(struct kset *s);
void subsystem_unregister(struct kset *s);
```

```
struct subsystem *subsys_get(struct kset *s)
void subsys_put(struct kset *s);
/*这些函数基本上是kset操作函数的包装，以实现子系统的操作*/
```

在新的内核里连以上的subsystem的函数都已经被取消了,完全被kset取代了。其原因可能是因为原来的subsystem的函数根本就是kset函数的简单包装，而在Linux世界里简单就是美，多余的东西被剔出了。

底层sysfs操作

kobject 是在 sysfs 虚拟文件系统后的机制。对每个在 sysfs 中的目录，在内核中都会有一个 kobject 与之对应。每个 kobject 都输出一个或多个属性，它在 kobject 的 sysfs 目录中以文件的形式出现，其中的内容由内核产生。 <linux/sysfs.h> 包含 sysfs 的工作代码。

在 sysfs 中创建kobject的入口是kobject_add的工作的一部分，只要调用 kobject_add 就会在sysfs 中显示，还有些知识值得记住:

（1）kobjects 的 sysfs 入口始终为目录，kobject_add 的调用将在sysfs 中创建一个目录，这个目录包含一个或多个属性（文件）；

（2）分配给 kobject 的名字(用 kobject_set_name) 是 sysfs 中的目录名，出现在 sysfs 层次的相同部分的 kobjects 必须有唯一的名字. 分配给 kobjects 的名字也应当是合法的文件名字：它们不能包含非法字符（如：斜线）且不推荐使用空白。

（3）sysfs 入口位置对应 kobject 的 parent 指针。若 parent 是 NULL ，则它被设置为嵌入到新 kobject 的 kset 中的 kobject；若 parent 和 kset 都是 NULL，则sysfs 入口目录在顶层，通常不推荐。

默认属性

当创建kobject 时，每个 kobject 都被给定一系列默认属性。这些属性保存在 kobj_type 结构中:

```
struct kobj_type {
 void (*release)(struct kobject *);
 struct sysfs_ops *sysfs_ops;/*提供实现以下属性的方法*/
 struct attribute **default_attrs; /*用于保存类型属性列表（指针的指针） */
};

struct attribute {
 char *name;/*属性的名字( 在 kobject 的 sysfs 目录中显示)*/
 struct module *owner;/*指向模块的指针(如果有)，此模块负责实现这个属性*/
 mode_t mode; /*属性的保护位,modes 的宏定义在 <linux/stat.h>:例如S_IRUGO 为只读属性等等*/
}; /*default_attrs 列表中的最后一个元素必须用 0 填充*/
```

sysfs 读写这些属性是由 kobj_type->sysfs_ops 成员中的函数完成的:

```
struct sysfs_ops {
 ssize_t (*show)(struct kobject *kobj, struct attribute *attr, char *buffer);
 ssize_t (*store)(struct kobject *kobj, struct attribute *attr, const char *buffer,
size_t size);
};
```

当用户空间读取一个属性时，内核会使用指向 kobject 的指针（kobj）和正确的属性结构（*attr）来调用show 方法，该方法将给定属性值编码进缓冲(buffer)（注意不要越界(PAGE_SIZE 字节)），并返回实际数据长度。sysfs 的约定要求每个属性应当包含一个单个人眼可读值；若返回大量信息,需将它分为多个属性.

也可对所有 kobject 关联的属性使用同一个 show 方法，用传递到函数的 attr 指针来判断所请求的属性。有的 show 方法包含对属性名字的检查。有的show 方法会将属性结构嵌入另一个结构，这个结构包含需要返回属性值的信息，这时可用container_of 获得上层结构的指针以返回属性值的信息。

store 方法将存在缓冲（buffer）的数据(size 为数据的长度，不能超过 PAGE_SIZE)解码并保存新值到属性（*attr），返回实际解码的字节数。store 方法只在拥有属性的写权限时才能被调用。此时注意：接收来自用户空间的数据一定要验证其合法性。如果到数据不匹配，返回一个负的错误值。

非默认属性

虽然 kobject 类型的 default_attrs 成员描述了所有的 kobject 会拥有的属性，倘若想添加新属性到

kobject 的 sysfs 目录属性只需简单地填充一个attribute结构并传递到以下函数:

```
int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
/*若成功，文件以attribute结构中的名字创建并返回 0；否则，返回负错误码*/
/*注意：内核会调用相同的 show() 和 store() 函数来实现对新属性的操作，所以在添加一个新非默认属性前，应采取必要的步骤确保这些函数知道如何实现这个属性*/
```

若要删除属性，调用:

```
int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
/*调用后，这个属性不再出现在 kobject 的 sysfs 入口。若一个用户空间进程可能有一个打开的那个属性的文件描述符，在这个属性已经被删除后，show 和 store 仍然可能被调用*/
```

二进制属性

sysfs 通常要求所有属性都只包含一个可读文本格式的值，很少需要创建能够处理大量二进制数据的属性。但当在用户空间和设备间传递不可改变的数据时（如上传固件到设备）就需要这个特性。二进制属性使用一个 bin_attribute 结构来描述:

```
struct bin_attribute {
    struct attribute    attr;/*属性结构体*/
    size_t              size;/*这个二进制属性的最大大小(若无最大值则为0)*/
    void                *private;
    ssize_t (*read)(struct kobject *, char *, loff_t, size_t);
    ssize_t (*write)(struct kobject *, char *, loff_t, size_t);
/*read 和 write 方法类似字符驱动的读写方法;，在一次加载中可被多次调用,每次调用最大操作一页数据，且必须能以其他方式判断操作数据的末尾*/
    int (*mmap)(struct kobject *, struct bin_attribute *attr,
         struct vm_area_struct *vma);
};

/*二进制属性必须显式创建，不能以默认属性被创建，创建一个二进制属性调用:*/
int sysfs_create_bin_file(struct kobject *kobj, struct bin_attribute *attr);

/*删除二进制属性调用:*/
int sysfs_remove_bin_file(struct kobject *kobj, struct bin_attribute *attr);
```

符号链接

sysfs 文件系统具有树型结构，反映 kobject之间的组织层次关系。为了表示驱动程序和所管理的设备间的关系，需要额外的指针，其在 sysfs 中通过符号链接实现。

```
/*在 sysfs 创建一个符号链接:*/
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char *name);
/*函数创建一个链接(name)指向target的 sysfs 入口作为 kobj 的一个属性，是一个相对连接，与它在sysfs 系统中的位置无关*/

/*删除符号连接调用:*/
void sysfs_remove_link(struct kobject *kobj, char *name);
```

热插拔事件产生

一个热插拔事件是一个从内核空间发送到用户空间的通知，表明系统配置已经改变. 无论 kobject 被创建或删除，都会产生这种事件。热插拔事件会导致对 /sbin/hotplug 的调用，它通过加载驱动程序，创建设备节点，挂载分区或其他正确动作响应事件。

热插拔事件的实际控制是通过一套存储于 kset_uevent_ops （《LDD3》中介绍的struct kset_hotplug_ops * hotplug_ops;在2.6.22.2中已经被kset_uevent_ops 结构体替换）结构的方法完成:

```
struct kset_uevent_ops {
    int (*filter)(struct kset *kset, struct kobject *kobj);
    const char *(*name)(struct kset *kset, struct kobject *kobj);
    int (*uevent)(struct kset *kset, struct kobject *kobj, char **envp,
            int num_envp, char *buffer, int buffer_size);
};
```

可以在 kset 结构的uevent_ops 成员中找到指向kset_uevent_ops结构的指针。

若在 kobject 中不包含指定的 kset ， 内核将通过 parent 指针在分层结构中进行搜索，直到发现一个包含有kset的 kobject ； 接着使用这个 kset 的热插拔操作。

kset_uevent_ops 结构中的三个方法作用如下：

（1） filter 函数让 kset 代码决定是否将事件传递给用户空间。如果 filter 返回 0,将不产生事件。以磁盘的 filter 函数为例，它只允许kobject产生磁盘和分区的事件，源码如下:

```
static int block_hotplug_filter(struct kset *kset, struct kobject *kobj)
{
 struct kobj_type *ktype = get_ktype(kobj);
    return ((ktype == &ktype_block) || (ktype == &ktype_part));
}
```

（2） 当调用用户空间的热插拔程序时，相关子系统的名字将作为唯一的参数传递给它。name 函数负责返回合适的字符串传递给用户空间的热插拔程序。

（3）热插拔脚本想得到的任何其他参数都通过环境变量传递。uevent 函数的作用是在调用热插拔脚本之前将参数添加到环境变量中。函数原型：

```
int (*uevent)(struct kset *kset, struct kobject *kobj, /*产生事件的目标对象*/
 char **envp,/*一个保存其他环境变量定义(通常为 NAME=value 的格式)的数组*/
 int num_envp, /*环境变量数组中包含的变量个数（数组大小）*/
 char *buffer, int buffer_size/*环境变量被编码后放入的缓冲区的指针和字节数（大小）*/
/*若需要添加任何环境变量到 envp, 必须在最后的添加项后加一个 NULL 入口，使内核知道数组的结尾*/
        );
/*返回值正常应当是 0，若返回非零值将终止热插拔事件的产生*/
```

热插拔事件的产生通常是由在总线驱动程序层的逻辑所控制。

以上是Linux设备模型的底层原理简介，具体的细节应该参阅内核源码和《ULK3》。

上一篇：Linux设备驱动程序学习（3-补）-Linux中的循环缓冲区
下一篇：Linux设备驱动程序学习（13）-Linux设备模型（总线、设备、驱动程序和类）