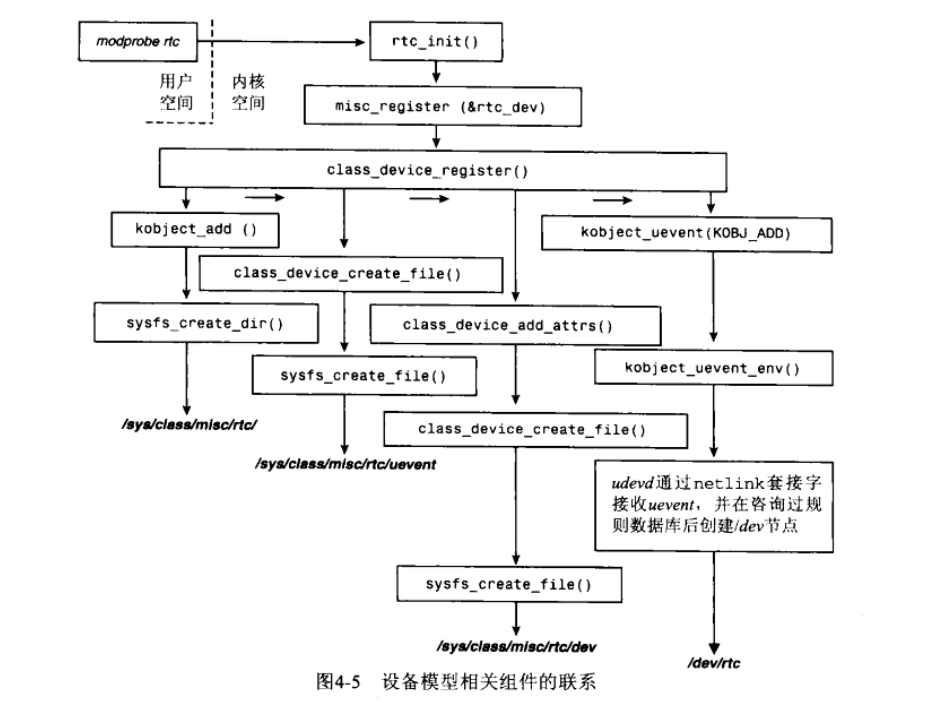
**SYS目录说明：**



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 加载的驱动module：insmod xxxx.ko可以看到系统使用编译的模块。**

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

**1．kobject, ktype, kset**

Kobject(内核对象)对应于在sysfs中的目录，用于设备对象的管理。

struct kobject {

const char \*name;

struct list\_head entry;

struct kobject \*parent;//指向kset中的kobject父对象。

struct kset \*kset;

struct kobj\_type \*ktype;

struct sysfs\_dirent \*sd;

struct kref kref;

unsigned int state\_initialized:1;

unsigned int state\_in\_sysfs:1;

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

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

unsigned int uevent\_suppress:1;

};

ktype代表kobject的类型，主要包含release函数和attr的读写函数。比如，所有的bus都有同一个bus\_type；所有的class都有同一个class\_type。

struct kobj\_type {

void (\*release)(struct kobject \*kobj); //释放kobject占用资源函数

const struct sysfs\_ops \*sysfs\_ops; //指向sysfs操作的指针，属性的操作

struct attribute \*\*default\_attrs; //sysfs文件系统属性默认列表，默认属性

。。。。。。。

};

sysfs\_ops：

包含两个成员函数：store（）和show（），用于实现对于属性的读写。

属性在sysfs中为文件，kobject为目录。

比如：

在/sys/xx/目录下有一属性A，对于属性A的读写操作，cat /echo关联到kobject下的kobj\_type下的sysfs\_ops类型指针所对应的show和store方法。

===========================================================================

例程操作：

#include <linux/device.h>   
#include <linux/module.h>   
#include <linux/kernel.h>   
#include <linux/init.h>   
#include <linux/string.h>   
#include <linux/sysfs.h>   
#include <linux/stat.h>   
 MODULE\_AUTHOR("David Xie");   
MODULE\_LICENSE("Dual BSD/GPL");

 /\*声明release、show、store函数\*/

void obj\_test\_release(struct kobject \*kobject);   
ssize\_t kobj\_test\_show(struct kobject \*kobject, struct attribute \*attr,char \*buf);  
ssize\_t kobj\_test\_store(struct kobject \*kobject,struct attribute \*attr,const char \*buf, size\_t count);

/\*对应于kobject的目录下的一个文件,Name成员就是文件名\*/    
struct attribute test\_attr = {   
        .name = "kobj\_config",   
        .mode = S\_IRWXUGO,   
};   
static struct attribute \*def\_attrs[] = {   
        &test\_attr,   
        NULL,   
};   
/kobject对象的操作   
struct sysfs\_ops obj\_test\_sysops =   
{   
        .show = kobj\_test\_show,   
        .store = kobj\_test\_store,   
}; //读写操作调用。

操作：

====================================

Cat kobj\_config

执行kobj\_test\_show函数：

Echo 1 kobj\_config

执行kobj\_test\_store函数。

===========================================

/\*定义kobject对象的一些属性及对应的操作\*/   
struct kobj\_type ktype =    
{   
        .release = obj\_test\_release,   
        .sysfs\_ops=&obj\_test\_sysops,   
        .default\_attrs=def\_attrs,   
};

/\*release方法释放该kobject对象\*/    
void obj\_test\_release(struct kobject \*kobject)   
{   
        printk("eric\_test: release .\n");   
}

/\*当读文件时执行的操作\*/   
ssize\_t kobj\_test\_show(struct kobject \*kobject, struct attribute \*attr,char \*buf)  
{   
        printk("have show.\n");   
        printk("attrname:%s.\n", attr->name);   
        sprintf(buf,"%s\n",attr->name);   
        return strlen(attr->name)+2;   
}

/\*当写文件时执行的操作\*/    
ssize\_t kobj\_test\_store(struct kobject \*kobject,struct attribute \*attr,const char \*buf, size\_t count)  
{   
        printk("havestore\n");   
        printk("write: %s\n",buf);   
        return count;   
}   
struct kobject kobj;//声明kobject对象  
static int kobj\_test\_init(void)   
{   
        printk("kboject test init.\n");   
        kobject\_init\_and\_add(&kobj,&ktype,NULL,"kobject\_test");//初始化kobject对象kobj,并将其注册到linux系统

在/sys/目录下生成kobject\_test目录,并在该目录下生成kobj\_config属性文件。

        return 0;   
}   
static void kobj\_test\_exit(void)   
{   
        printk("kobject test exit.\n");   
        kobject\_del(&kobj);   
}   
module\_init(kobj\_test\_init);

module\_exit(kobj\_test\_exit);

===========================================================================

Kset（kobject内核对象的集合）包含了subsystem概念，在sysfs中体现成一个目录，kset本身也是一个kobject，所以里面包含了一个kobject对象。

包含在kset中的所有kobject被组织成一个双向循环链表，list即是该链表的头。

Kset与kobject的关系：

struct kset {

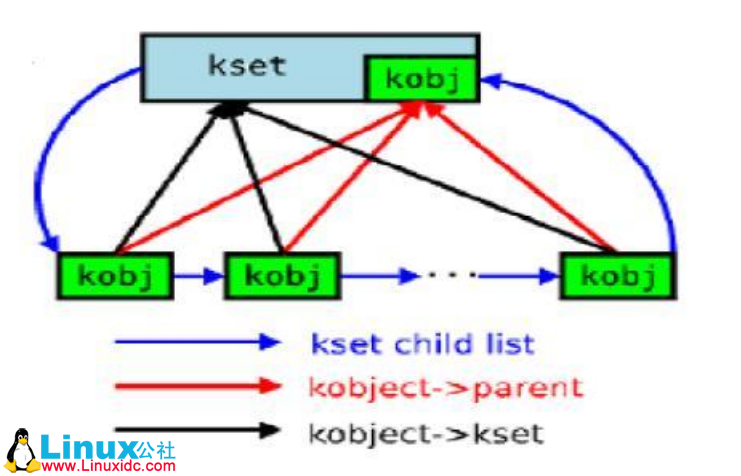
struct list\_head list;

spinlock\_t list\_lock;

struct kobject kobj;//内嵌kobject对象，kset中所有kobject对象的父对象

const struct kset\_uevent\_ops \*uevent\_ops;

};



// 热插拔事件：

当系统配置发生变化时,如:添加kset到系统；移动kobject, 一个通知会从内核空间发送到用户空间,这就是热插拔事件。

kobject被创建或删除时会产生事件（event），kobject 所属的 kset将有机会过滤事

件或为用户空间添加信息。每个kset能支持一些特定的事件变量，在热插拔事件发生

时，kset 的成员函数可以设置一些事件变量，这些变量将被导出到用户空间。kset 的

uevent\_ops成员是执行该 kset事件操作集kset\_uevent\_ops的指针

kset中包含kset\_uevent\_ops，里面主要定义了三个函数

struct kset\_uevent\_ops {

int (\* const filter)(struct kset \*kset, struct kobject \*kobj);//

用于过滤掉不需要导出到用户空间的事件, 判断uevent是否要发出去

const char \*(\* const name)(struct kset \*kset, struct kobject \*kobj);//

name用于得到subsystem的名字。

int (\* const uevent)(struct kset \*kset, struct kobject \*kobj,

struct kobj\_uevent\_env \*env);

用于导出一些环境变量给用户的热插拔处理程序, uevent用于填充env变量

};这三个函数都与uevent相关。

subsystem 是一系列 kset 的集合，它描述系统中某一类设备子系统，如

block\_subsys 表示所有的块设备，对应于 sysfs 文件系统中的 block 目录。

===========================================================================

5、 kset实例分析

    #include <linux/device.h>  
    #include <linux/module.h>  
    #include <linux/kernel.h>  
    #include <linux/init.h>  
    #include <linux/string.h>  
    #include <linux/sysfs.h>  
    #include <linux/stat.h>  
    #include <linux/kobject.h>

    MODULE\_AUTHOR("yinjiabin");  
    MODULE\_LICENSE("GPL");

    struct kset \*kset\_p;  
    struct kset kset\_c;

    /\* 函数声明 \*/  
    void obj\_test\_release(struct kobject \*);  
    ssize\_t kobj\_test\_show(struct kobject \*,struct attribute \*,char \*);  
    ssize\_t kobj\_test\_store(struct kobject \*,struct attribute \*,const char \*,size\_t);

    static struct attribute test\_attr =  
    {  
            .name = "kobj\_config",  
            .mode = S\_IRWXUGO,  
    };

    static struct attribute \*def\_attrs[] =  
    {  
            &test\_attr,  
            NULL,  
    };

    static struct sysfs\_ops obj\_test\_sysops =  
    {  
            .show = kobj\_test\_show,  
            .store = kobj\_test\_store,  
    };

    static struct kobj\_type ktype =  
    {  
            .release = obj\_test\_release,  
            .sysfs\_ops = &obj\_test\_sysops,  
            .default\_attrs = def\_attrs,  
};

    void obj\_test\_release(struct kobject \*kobject)  
    {  
            printk("[kobj\_test: release!]\n");  
    }

    ssize\_t kobj\_test\_show(struct kobject \*kobject,struct attribute \*attr,char \*buf)  
    {  
            printk("Have show -->\n");  
            printk("attrname: %s.\n",attr->name);  
            sprintf("buf,%s\n",attr->name);  
            return strlen(attr->name) + 2;  
    }

    ssize\_t kobj\_test\_store(struct kobject \*kobject,struct attribute \*attr, const char \*buf,size\_t size)  
    {  
            printk("Have store -->\n");  
            printk("write: %s.\n",buf);  
            return size;  
    }

    static int kset\_filter(struct kset \*kset,struct kobject \*kobj)  
    {  
        printk("Filter: kobj %s.\n",kobj->name);  
        return 1;  
    }

    static const char \*kset\_name(struct kset \*kset,struct kobject \*kobj)  
    {  
        static char buf[20];  
        printk("Name kobj %s.\n",kobj->name);  
        sprintf(buf,"%s","kset\_name");  
        return buf;  
    }

    static int kset\_uevent(struct kset \*kset,struct kobject \*kobj, struct kobj\_uevent\_env \*env)

{  
        int i = 0;  
        printk("uevent: kobj %s.\n",kobj->name);

        while(i < env->envp\_idx)  
        {  
            printk("%s.\n",env->envp[i]);  
            i ++;  
        }

        return 0;  
    }

    static struct kset\_uevent\_ops uevent\_ops =  
    {  
        .filter = kset\_filter,  
        .name = kset\_name,  
        .uevent = kset\_uevent,  
    };

    static int \_\_init kset\_test\_init(void)  
    {  
        int ret = 0;

        printk("kset test init!\n");

        /\* 创建并注册 kset\_p \*/  
        kset\_p = kset\_create\_and\_add("kset\_p",&uevent\_ops,NULL);

        /\* 添加 kset\_c 到 kset\_p \*/  
        kobject\_set\_name(&kset\_c.kobj,"kset\_c");  
        kset\_c.kobj.kset = kset\_p;

        /\* 对于较新版本的内核，在注册 kset 之前，需要    
             \* 填充 kset.kobj 的 ktype 成员，否则注册不会成功 \*/  
        kset\_c.kobj.ktype = &ktype;  
        ret = kset\_register(&kset\_c);

        if(ret)  
            kset\_unregister(kset\_p);

        return 0;  
    }

    static void \_\_exit kset\_test\_exit(void)  
    {  
        printk("kset test exit!\n");  
        kset\_unregister(kset\_p);  
        kset\_unregister(&kset\_c);

    }

module\_init(kset\_test\_init);  
    module\_exit(kset\_test\_exit);

运行：

Insmod xxx.ko

执行kset\_test\_init/ kset\_register,kset\_register🡪 kobject\_uevent\_env之后依次执行uevent\_ops下的filter , name ,uevent方法。

===========================================================================

2．uevent内核部分

驱动中创建设备节点：

device\_create

device\_register

device\_add

dev\_set\_name //初始化设备内部的kobject的名字

kobj = get\_device\_parent(dev, parent);

dev->kobj.parent = kobj; //在kobject层实现设备父子关系

kobject\_add(&dev->kobj, dev->kobj.parent, NULL) //将设备加入到kobject模型中，创建sys相关目录

device\_create\_file(dev, &uevent\_attr); //创建sys目录下设备的uevent属性文件，通过它可以查看设备的uevent事件

device\_create\_file(dev, &devt\_attr); //创建sys目录下设备的设备号属性，即major和minor

device\_add\_attrs(dev); //创建sys目录下设备其他属性文件

blocking\_notifier\_call\_chain(&dev->bus->p->bus\_notifier,BUS\_NOTIFY\_ADD\_DEVICE, dev);//通知注册监听该总线的设备，有新设备加 入

kobject\_uevent(&dev->kobj, KOBJ\_ADD); //产生一个内核uevent事件，该事件可以被内核以及应用层捕获，属于linux设备模型 中热插拔机制. 调用kobject\_uevent(&dev->kobj, KOBJ\_ADD); 这里kobj是发消息的kobj，KOBJ\_ADD是发出的事件。uevent的事件 在kobject\_action中定义：

+++++++++++++++++++++++++++++++++++++++++++++++++++++++

enum kobject\_action {

       KOBJ\_ADD,

        KOBJ\_REMOVE,

      KOBJ\_CHANGE,

        KOBJ\_MOVE,

        KOBJ\_ONLINE,

        KOBJ\_OFFLINE,

        KOBJ\_MAX

};

1）获取action字符串

\*action\_string = kobject\_actions[action];

Action为KOBJ\_ADD等，kobject\_actions的定义如下：

static const char \*kobject\_actions[] = {

       [KOBJ\_ADD] =            "add",

       [KOBJ\_REMOVE] =           "remove",

       [KOBJ\_CHANGE] =            "change",

       [KOBJ\_MOVE] =         "move",

       [KOBJ\_ONLINE] =             "online",

       [KOBJ\_OFFLINE] =     "offline",

};

++++++++++++++++++++++++++++++++++++++++++++++++

………

list\_for\_each\_entry(class\_intf,&dev->class->p->interfaces, node)

             if (class\_intf->add\_dev)

                 class\_intf->add\_dev(dev, class\_intf);//通知有新设备加入

==========================

 kobject\_uevent调用流程：

kobject\_uevent\_env://发送用户事件和环境数据

1. 由kobject的parent向上查找，直到找到一个kobject包含kset。
2. set中有filter函数，调用filter函数，看看是否需要过滤uevent消息。
3. set中有name函数，调用name函数得到subsystem的名字；否则，subsystem的名字是kset中kobject的名字。

具体实现：

获取subsystem字符串

subsystem = kobject\_name(&kset->kobj);

static inline const char \*kobject\_name(const struct kobject \*kobj)

{

       return kobj->name;

}

这里主要获取kobj的名字。

Kobj的名字在创建时传递，power\_supply

power\_supply\_class = class\_create(THIS\_MODULE, "power\_supply");

将调用以下函数class\_create---> \_\_class\_register ----> kobject\_set\_name:

最终设置为power\_supply。

1. 分配一个kobj\_uevent\_env，并开始填充env环境变量：kzalloc(sizeof(struct kobj\_uevent\_env), GFP\_KERNEL);
2. 增加环境变量ACTION=<action name>

add\_uevent\_var(env, "ACTION=%s", action\_string);

1. 增加环境变量DEVPATH=<kobj’s path>

add\_uevent\_var(env, "DEVPATH=%s", devpath);

1. 增加环境变量SUBSYSTEM=<subsystem name>

add\_uevent\_var(env, "SUBSYSTEM=%s", subsystem);

        4）增加环境变量kobject\_uevent\_env中参数envp\_ext指定的环境变量。

接着加入不同class的附加的字符串

retval = add\_uevent\_var(env, "%s", envp\_ext[i]);

………………….

1. 调用kset的uevent函数，这个函数会继续填充环境变量。

uevent\_ops->uevent(kset, kobj, env);

1. 增加环境变量SEQNUM=<seq>，这里seq是静态变量，每次累加。

add\_uevent\_var(env, "SEQNUM=%llu", (unsigned long long)++uevent\_seqnum);

1. 调用netlink发送uevent消息。

具体实现：

字符串准备完毕，就要准备发送了，由于Android的CONFIG\_NET选项是选上的，因此可以通过socket发送：

首先分配一个skb用于存储网络发送的数据

scratch = skb\_put(skb, len);

sprintf(scratch, "%s@%s", action\_string, devpath);

此时scratch中就增加了change@/devices/platform/msm-battery/power\_supply/usb的字符，然后将之前准备好的各个字符传加在后面

for (i = 0; i < env->envp\_idx; i++) {

       len = strlen(env->envp[i]) + 1;

scratch = skb\_put(skb, len);

       strcpy(scratch, env->envp[i]);

}

最后发送调用retval = netlink\_broadcast\_filtered

1. 调用uevent\_helper，最终转换成对用户空间sbin/mdev的调用。

kobj\_usermode\_filter

**sys中的bus总线。**

设备模型元素  
        总线  
        驱动  
        设备  
总线  
       总线是处理器和设备之间的通道,在设备模型中, 所有的设备都通过总线相连, 甚至是内部的虚拟“platform”总线。 在 Linux 设备模型中, 总线由 bus\_type 结构表示。

struct bus\_type {

const char \*name;

const char \*dev\_name;

struct device \*dev\_root;

struct bus\_attribute \*bus\_attrs;

struct device\_attribute \*dev\_attrs;

struct driver\_attribute \*drv\_attrs;

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

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

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

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

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 iommu\_ops \*iommu\_ops;

struct subsys\_private \*p;

struct lock\_class\_key lock\_key;};

**总线的操作：**

总线的注册使用:bus\_register(struct bus\_type \* bus)  
             若成功,新的总线将被添加进系统,并可在sysfs 的 /sys/bus 下看到。

 1）int (\*match)(struct device \* dev, struct device\_driver \* drv)  
        当一个新设备或者驱动被添加到这个总线时,该方法被调用。用于判断指定的驱动程序是否能处理指定的设备。若可以,则返回非零值。  
 2）int (\*uevent)(struct device \*dev, char \*\*envp, int num\_envp, char \*buffer, int buffer\_size)  
        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。

**总线属性**  
       总线属性由结构bus\_attribute 描述,定义如下:

struct bus\_attribute {  
struct attribute  
attr;  
ssize\_t (\*show)(struct bus\_type \*, char \* buf);  
ssize\_t (\*store)(struct bus\_type \*, const char \*  
buf, size\_t count);  
}

       1）int bus\_create\_file(struct bus\_type \*bus, struct bus\_attribute \*attr)  
            创建属性  
       2）void bus\_remove\_file(struct bus\_type \*bus, struct bus\_attribute \*attr)  
            删除属性

实例：

Iic总线的添加：

Iic总线I2c-core.c下的i2c\_init函数调用：bus\_register

bus\_register

kobject\_set\_name

kset\_register //创建sys/bus下的iic目录

kset\_create\_and\_add //在iic目录下的添加device和driver

实例：

===============================================================================

 Bus\_basic.c源码

#include <linux/device.h>  
#include <linux/module.h>  
#include <linux/kernel.h>  
#include <linux/init.h>  
#include <linux/string.h>

MODULE\_AUTHOR("yinjiabin");  
MODULE\_LICENSE("Dual BSD/GPL");

static char \*Version = "$Revision: 1.0 $";

/\*当一个新设备或者驱动被添加到这个总线时,该方法被调用。用于判断指定的驱动程序是否能处理指定的设备。若可以,则返回非零值。\*/

static int my\_match(struct device \*dev, struct device\_driver \*driver)  
{  
        return !strncmp(dev->kobj.name, driver->name, strlen(driver->name));  
}

/\*声明总线\*/  
struct bus\_type my\_bus\_type = {  
        .name = "my\_bus",  //总线名字  
        .match = my\_match, //总线match函数指针  
};

static ssize\_t show\_bus\_version(struct bus\_type \*bus, char \*buf)  
{  
        return snprintf(buf, PAGE\_SIZE, "%s\n", Version);  
}

/\*内核代码中如此定义：#define BUS\_ATTR(\_name, \_mode, \_show, \_store) \  
struct bus\_attribute bus\_attr\_##\_name = \_\_ATTR(\_name, \_mode, \_show, \_store),

它将bus\_attr\_作为给定的name的前缀来创建总线的真正名称。对应下面的是bus\_attr\_version\*/  
static BUS\_ATTR(version, S\_IRUGO, show\_bus\_version, NULL);

/\*模块加载函数\*/  
static int \_\_init my\_bus\_init(void)  
{  
        int ret;

        /\*注册总线\*/  
        ret = bus\_register(&my\_bus\_type);  
        if (ret)  
                return ret;

        /\*创建属性文件\*/  
        if (bus\_create\_file(&my\_bus\_type, &bus\_attr\_version))  
                printk(KERN\_NOTICE "Fail to create version attribute!\n");

        return ret;  
}

/\*模块卸载函数\*/  
static void my\_bus\_exit(void)  
{  
        bus\_unregister(&my\_bus\_type);  
}

module\_init(my\_bus\_init);

module\_exit(my\_bus\_exit);

运行：

在sys/bus下创建my\_bus的目录，并创建device和driver目录，在bus目录下添加version属性文件。

===============================================================================

Sys/device:

设备描述  
        Linux 系统中的每个设备由一个 struct device 描述:

struct device {

...... ...... ...... ...... ...... ......  
struct kobject kobj;  
char bus\_id[BUS\_ID\_SIZE]; /\*在总线上唯一标识该设备的字符串 \*/  
struct bus\_type /\* 设备所在总线 \*/  
\*bus;  
struct device\_driver \*driver; /\*管理该设备的驱动\*/  
void \*driver\_data;  
/\*该设备驱动使用的私有数据成员 \*  
struct klist\_node knode\_class;  
struct class \*class;  
struct attribute\_group  
\*\*groups;  
void (\*release)(struct device \*dev);

}

2、 设备注册  
       1）int device\_register(struct device \*dev)  
            注册设备  
      2）void device\_unregister(struct device \*dev)  
           注销设备  
      \*\*一条总线也是个设备,也必须按设备注册\*\*  
3、 设备属性  
       设备属性由struct device\_attribute 描述:

struct device\_attribute  
{

struct attribute attr;  
ssize\_t (\*show)(struct device \*dev, struct device\_attribute  
\*attr,char \*buf);  
ssize\_t (\*store)(struct device \*dev, struct device\_attribute \*attr,  
const char \*buf, size\_t count);

}

       1）int device\_create\_file(struct device \*device, struct device\_attribute \* entry)  
             创建属性  
       2）void device\_remove\_file(struct device \* dev, struct device\_attribute \* attr)  
            删除属性  
**4、 实例分析**

**===============================================================================**

      1）Bus.c源码

#include <linux/device.h>  
#include <linux/module.h>  
#include <linux/kernel.h>  
#include <linux/init.h>  
#include <linux/string.h>

MODULE\_AUTHOR("yinjiabin");  
MODULE\_LICENSE("Dual BSD/GPL");

static char \*Version = "$Revision: 1.0 $";

/\*当一个新设备或者驱动被添加到这个总线时,该方法被调用。用于判断指定的驱动程序是否能处理指定的设备。若可以,则返回非零值。\*/

static int my\_match(struct device \*dev, struct device\_driver \*driver)  
{  
        return !strncmp(dev->kobj.name, driver->name, strlen(driver->name));  
}

static void my\_bus\_release(struct device \*dev)  
{  
        printk(KERN\_DEBUG "my bus release\n");  
}

struct device my\_bus = {  
        .kobj.name   = "my\_bus0",  
        .release  = my\_bus\_release  
};

/\*声明总线\*/  
struct bus\_type my\_bus\_type = {  
        .name = "my\_bus",  //总线名字  
        .match = my\_match, //总线match函数指针  
};

EXPORT\_SYMBOL(my\_bus);      //导出my\_bus  
EXPORT\_SYMBOL(my\_bus\_type); //导出my\_bus\_type

/\*  
 \* Export a simple attribute.  
 \*/  
static ssize\_t show\_bus\_version(struct bus\_type \*bus, char \*buf)  
{  
        return snprintf(buf, PAGE\_SIZE, "%s\n", Version);  
}

/\*内核代码中如此定义：#define BUS\_ATTR(\_name, \_mode, \_show, \_store) \  
struct bus\_attribute bus\_attr\_##\_name = \_\_ATTR(\_name, \_mode, \_show, \_store),

它将bus\_attr\_作为给定的name的前缀来创建总线的真正名称。对应下面的是bus\_attr\_version\*/  
static BUS\_ATTR(version, S\_IRUGO, show\_bus\_version, NULL);

/\*模块加载函数\*/  
static int \_\_init my\_bus\_init(void)  
{  
        int ret;

        /\*注册总线\*/  
        ret = bus\_register(&my\_bus\_type);  
        if (ret)  
                return ret;

        /\*创建属性文件\*/  
        if (bus\_create\_file(&my\_bus\_type, &bus\_attr\_version))  
                printk(KERN\_NOTICE "Fail to create version attribute!\n");

        /\*注册总线设备\*/  
        ret = device\_register(&my\_bus);  
        if (ret)  
                printk(KERN\_NOTICE "Fail to register device:my\_bus!\n");

        return ret;  
}

/\*模块卸载函数\*/  
static void my\_bus\_exit(void)  
{  
        device\_unregister(&my\_bus);  
        bus\_unregister(&my\_bus\_type);  
}

module\_init(my\_bus\_init);  
module\_exit(my\_bus\_exit);

**======================================**

**device实例：**

2）Device.c源码

#include <linux/device.h>  
#include <linux/module.h>  
#include <linux/kernel.h>  
#include <linux/init.h>  
#include <linux/string.h>

MODULE\_AUTHOR("yinjiabin");  
MODULE\_LICENSE("Dual BSD/GPL");

extern struct device my\_bus;  
extern struct bus\_type my\_bus\_type;

/\* Why need this ?\*/  
static void my\_dev\_release(struct device \*dev)  
{

}

struct device my\_dev = {  
        .init\_name = "my\_dev",  
        .bus = &my\_bus\_type,   //指明设备所属的总线  
        .parent = &my\_bus,     //该设备的父设备，总线也是一种设备  
        .release = my\_dev\_release,  
};

static ssize\_t mydev\_show(struct device \*dev,struct device\_attribute \*attr, char \*buf)  
{  
        return sprintf(buf, "%s\n", "This is my device!");  
}

static DEVICE\_ATTR(dev, S\_IRUGO, mydev\_show, NULL);

static int \_\_init my\_device\_init(void)  
{  
        int ret = 0;

        /\* 初始化设备 \*/  
        ///strncpy(my\_dev->kobj.name, "my\_dev", strlen(my\_dev->name));

        /\*注册设备\*/  
        device\_register(&my\_dev);

        /\*创建属性文件\*/  
        device\_create\_file(&my\_dev, &dev\_attr\_dev);

        return ret;

}

static void my\_device\_exit(void)  
{  
        device\_unregister(&my\_dev);  
}

module\_init(my\_device\_init);  
module\_exit(my\_device\_exit);

运行效果：

在相关的bus目录的device下创建相关的device设备目录及属性文件。

**=====================================================================**

Bus下的driver。

驱动描述：

       驱动程序由struct device\_driver 描述 :

struct device\_driver {

const char \*name; /\*驱动程序的名字( 体现在 sysfs 中 )\*/  
struct bus\_type \*bus; /\*驱动程序所在的总线\*/  
struct module  
\*owner;  
const char  
\*mod\_name;  
int (\*probe) (struct device \*dev);  
int (\*remove) (struct device \*dev);  
void (\*shutdown) (struct device \*dev);  
int (\*suspend) (struct device \*dev, pm\_message\_t state);  
int (\*resume) (struct device \*dev);  
struct attribute\_group \*\*groups;  
struct dev\_pm\_ops \*pm;  
struct driver\_private \*p;

}

2、 驱动注册/注册

        1）int driver\_register(struct device\_driver \*drv)

            注册驱动  
       2）void driver\_unregister(struct device\_driver \*drv)

           注销驱动

3、 驱动属性  
       驱动的属性使用struct driver\_attribute 来描述:

struct driver\_attribute {

struct attribute attr;  
ssize\_t (\*show)(struct device\_driver \*drv,  
char \*buf);  
ssize\_t (\*store)(struct device\_driver \*drv,  
const char \*buf, size\_t count);

}

      1）int driver\_create\_file(struct device\_driver \* drv, struct driver\_attribute \* attr)  
            创建属性  
      2）void driver\_remove\_file(struct device\_driver \* drv, struct driver\_attribute \* attr)  
            删除属性  
4、 实例分析  
       driver.c源码

#include <linux/device.h>  
#include <linux/module.h>  
#include <linux/kernel.h>  
#include <linux/init.h>  
#include <linux/string.h>

MODULE\_AUTHOR("David Xie");  
MODULE\_LICENSE("Dual BSD/GPL");

extern struct bus\_type my\_bus\_type;

/\*当驱动找到对应的设备时会执行该函数\*/  
static int my\_probe(struct device \*dev)  
{  
    printk("Driver found device which my driver can handle!\n");  
    return 0;  
}

static int my\_remove(struct device \*dev)  
{  
    printk("Driver found device unpluged!\n");  
    return 0;  
}

struct device\_driver my\_driver = {  
        .name = "my\_dev",  
        .bus = &my\_bus\_type,  
        .probe = my\_probe,  
        .remove = my\_remove,  
};

static ssize\_t mydriver\_show(struct device\_driver \*driver, char \*buf)  
{  
        return sprintf(buf, "%s\n", "This is my driver!");  
}

static DRIVER\_ATTR(drv, S\_IRUGO, mydriver\_show, NULL);

static int \_\_init my\_driver\_init(void)  
{  
        int ret = 0;

        /\*注册驱动\*/  
        driver\_register(&my\_driver);

        /\*创建属性文件\*/  
        driver\_create\_file(&my\_driver, &driver\_attr\_drv);

        return ret;

}

static void my\_driver\_exit(void)  
{  
        driver\_unregister(&my\_driver);  
}

module\_init(my\_driver\_init);

module\_exit(my\_driver\_exit);

试验结果

Bus ,device,driver相互作用。

Kobject层次图：

