努力成为 linux kernel hacker 的人李万鹏原创作品，转载请标明出处 http://blog.csdn.net/woshixingaaa/archive/2011/05/05/6396620.aspx

Linux 设备模型是由总线（bus_type），设备（device），驱动（device_driver）这三个数据结构来描述的。在设备模型中，所有的设备都通过总线来连接。即使有些设备没有连接到一根物理上的总线，Linux 为其设置了一个内部的，虚拟的 platform 总线，来维持总线，驱动，设备的关系。总线是处理器与一个或者多个设备之间的通道。比如一个 USB 控制器通常是一个 PCI 设备，设备模型展示了总线和他们所控制的设备之间的连接。

一般来说可以这么理解,整个的设备模型是一个 OO 的体系结构,总线,设备,驱动都是其中鲜活存在的对象，kobject 是他们的基类，所实现的只是一些公共的接口，kset 是同种类型的 kobject 对象的集合，也可以说是对象的容器。只是因为 C 语言里不可能会有 C++语言里类的 class 继承等概念，只有通过 kobject 嵌入到对象结构中来实现。这样，内核使用 kobject 将各个对象连接起来组成一个分层的体系结构。kobject 结构中包含了 parent 成员，指向了另一个 kobject 结构，也就是这个分层结构的上一层结点。而 kset 是通过链表来实现的。

kobject 是 Linux 在 2.6 中新引进的统一设备管理模型，目的是对 Linux 的 2.6 系统所有的设备进行统一的管理。kobject 是组成设备模型的基本结构。

kobject 是驱动程序模型中的一个核心数据结构，与 sysfs 文件系统自然的绑定在一起：——每个 kobject 对应 sysfs 文件系统中的一个目录。kobject 往往被嵌入到设备驱动程序模型中的组件中，如总线，设备和驱动程序的描述符。

kobject 的作用是，为所属“容器”提供

.引用计数器

.维持容器的层次列表或组

.为容器的属性提供一种用户态查看的视图

kset 是同类型 kobject 结构的一个集合体，通过 kset 数据结构可将 kobjects 组成一棵层次树。

```
struct bus_type {
    const char      *name;      //总线类型的名称
    struct bus_attribute    *bus_attrs;  //总线属
    struct device_attribute *dev_attrs;  //设备属
    struct driver_attribute *drv_attrs;  //驱动属性
    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 (*suspend_late)(struct device *dev, pm_message_t state);
    int (*resume_early)(struct device *dev);
    int (*resume)(struct device *dev);
    struct dev_pm_ops *pm;
    struct bus_type_private *p;
};
struct bus_type_private {
    struct kset subsys;                     //该总线的 subsystem
    struct kset *drivers_kset;              //所有与该总线相关的驱动集合
    struct kset *devices_kset;              //所有挂接在该总线上的设备集合
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type *bus;
};
struct bus_attribute {
    struct attribute    attr;
    ssize_t (*show)(struct bus_type *bus, char *buf);
    ssize_t (*store)(struct bus_type *bus, const char *buf, size_t count);
};
```

subsys 描述该总线的子系统，subsys 是一个 kset 结构，他连接到一个全局变量 kset bus_subsys 中。这样，每一根总线系统都会通过 bus_subsys 结构连接起来。kset *devices_kset 是指向该总线所有设备的集合的指针，kset *drivers_kset 是指向该总线所有驱动的集合的指针。该总线上的设备和驱动分别用一个链表连接在一起，分别是 klist_devices，klist_drivers。每次都用 kset *drivers_kset，kset *devices_kset 遍历所有设备/驱动很麻烦，用 klist 比较直接方便。

```
struct device {
    struct klist        klist_children;
    struct klist_node   knode_parent;   /* node in sibling list */
    struct klist_node   knode_driver;
    struct klist_node   knode_bus;
    struct device       *parent;
    struct kobject kobj;
    char    bus_id[BUS_ID_SIZE];    /* position on parent bus */
    unsigned        uevent_suppress:1;
    const char      *init_name; /* initial name of the device */
    struct device_type  *type;
    struct semaphore    sem;    /* semaphore to synchronize calls to
                 * its driver.
                 */
    struct bus_type *bus;       /* type of bus device is on */
    struct device_driver *driver;   /* which driver has allocated this
```

```
                   device */
    void        *driver_data;   /* data private to the driver */
    void        *platform_data; /* Platform specific data, device
                   core doesn't touch it */
    struct dev_pm_info  power;
#ifdef CONFIG_NUMA
    int     numa_node;  /* NUMA node this device is close to */
#endif
    u64     *dma_mask;  /* dma mask (if dma'able device) */
    u64     coherent_dma_mask;/* Like dma_mask, but for
                     alloc_coherent mappings as
                     not all hardware supports
                     64 bit addresses for consistent
                     allocations such descriptors. */
    struct device_dma_parameters *dma_parms;
    struct list_head    dma_pools;  /* dma pools (if dma'ble) */
    struct dma_coherent_mem *dma_mem; /* internal for coherent mem
                     override */
    /* arch specific additions */
    struct dev_archdata archdata;
    dev_t           devt;   /* dev_t, creates the sysfs "dev" */
    spinlock_t      devres_lock;
    struct list_head    devres_head;
    struct klist_node   knode_class;
    struct class        *class;
    struct attribute_group  **groups;   /* optional groups */
    void    (*release)(struct device *dev);
};
struct device_private {
    struct klist klist_children;
    struct klist_node knode_parent;
    struct klist_node knode_driver;
    struct klist_node knode_bus;
    struct device *device;
};
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);
};
```

需要注意的是，总线也是设备，也必须按设备注册。这里的 parent 是指该设备所属的父设备，struct kobject kobj;表示该设备并把它连接到结构体系中的 kobject。请注意，作为一个通用准则，device->kobj->parent 与&device-

>parent->kobj 是相同的。bus_id 是在总线上唯一标识该设备的字符串。struct bus_type *bus;标识了该设备连接在何种类型的总线上。struct device_driver *driver;管理该设备的驱动。void (*release)(struct device *dev);当指向设备的最后一个引用被删除时,内核调用该方法。它将从内嵌的 kobject 的 release 方法中调用。device_private 中的 knode_parent,knode_driver,knode_bus 分别是挂入 parent,驱动，总线链表中的指针。

```
struct device_driver {
    const char      *name;              //设备驱动程序的名称
    struct bus_type     *bus;             //该驱动所管理的设备挂接的总线类型
    struct module       *owner;
    const char      *mod_name;  /* used for built-in modules */
    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;
};
struct driver_private {
    struct kobject kobj;
    struct klist klist_devices;             //该驱动所管理的设备链表头
    struct klist_node knode_bus;            //挂入总线链表中的指针
    struct module_kobject *mkobj;
    struct device_driver *driver;
};
struct driver_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device_driver *driver, char *buf);
    ssize_t (*store)(struct device_driver *driver, const char *buf,
            size_t count);
};
```

name 指向驱动的名字，上边的 device 中也有一个名为 bus_id 的字符数组。查看一下，struct bus_type 中有一个 match，函数，这个是干什么用的呢。设备有了驱动才可以工作，只有驱动没有设备也是不行，驱动和设备需要关联上，这就需要这个 match 函数。驱动和设备是通过 name 来管理的，所以在 match 函数中要比较 device 的 bus_id 和 driver 中的 name 是否相等。如果相等，就说明驱动和设备互相找到了，这时 device_driver 中的 probe 函数被调用。我下边的例子中是这样实现的：

```
static ssize_t show_driver_author(struct device_driver *driver, char *buf){
    return snprintf(buf, PAGE_SIZE, "%s/n", author);
}
```

下面是一个测试程序：

BUS：

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

static char *author = "LiWanPeng";

static ssize_t show_bus_author(struct bus_type *bus, char *buf){
    return snprintf(buf, PAGE_SIZE, "%s/n", author);
}

void my_bus_release(struct device *dev){
    printk(KERN_DEBUG "my bus release/n");
}

static int virtual_bus_match(struct device *dev, struct device_driver *drv){
    return !strncmp(dev->bus_id, drv->name, strlen(drv->name));
}

struct bus_type virtual_bus = {
    .name = "my_bus",
    .match = virtual_bus_match,
};

struct device my_bus = {
    .init_name = "my_bus0",
    .release = my_bus_release,
};

EXPORT_SYMBOL(my_bus);
EXPORT_SYMBOL(virtual_bus);

static BUS_ATTR(author, S_IRUGO, show_bus_author, NULL);

static int __init bus_init(void){
    int ret;
    ret = bus_register(&virtual_bus);
    if(ret)
        return ret;
    if(bus_create_file(&virtual_bus, &bus_attr_author))
        printk(KERN_NOTICE "Unable to create author attribute/n");
    ret = device_register(&my_bus);
    if(ret)
```

```
        printk(KERN_NOTICE "Fail to register device/n");
    printk("bus regiter success/n");
    return ret;
}

static void __exit bus_exit(void){
    bus_unregister(&virtual_bus);
    device_unregister(&my_bus);
}

module_init(bus_init);
module_exit(bus_exit);
MODULE_LICENSE("GPL");
```

DEVICE：

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

char *author = "LiWanPeng";
extern struct bus_type virtual_bus;
extern struct device my_bus;

static ssize_t show_device_author(struct device *dev, struct device_attribute *attr, char *buf){
    return snprintf(buf, PAGE_SIZE, "%s/n", author);
}

void virtual_device_release(struct device *dev){
    printk("virtual_device is released/n");
}

struct device virtual_device ={
    .bus_id = "my_dev",
    .bus = &virtual_bus,
    .parent = &my_bus,
    .release = virtual_device_release,
};

static DEVICE_ATTR(author, S_IRUGO, show_device_author, NULL);

static int __init device_init(void){
    int ret;
    ret = device_register(&virtual_device);
    if(ret)
        return ret;
    if(device_create_file(&virtual_device, &dev_attr_author))
        printk(KERN_NOTICE "Unable to create author attribute/n");
    printk("device register success/n");
    return ret;
}

static void __exit device_exit(void){
    device_unregister(&virtual_device);
}

module_init(device_init);
module_exit(device_exit);
MODULE_AUTHOR("liwanpeng");
MODULE_LICENSE("GPL");
```

DRIVER：

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

extern struct bus_type virtual_bus;
char *author = "LiWanPeng";

static ssize_t show_driver_author(struct device_driver *driver, char *buf){
    return snprintf(buf, PAGE_SIZE, "%s/n", author);
}

int my_driver_remove(struct device *dev){
    printk("driver is removed/n");
    return 0;
}

int my_driver_probe(struct device *dev){
    printk("driver can handle the device/n");
    return 0;
}

struct device_driver virtual_driver = {
    .name = "my_dev",
    .bus = &virtual_bus,
    .probe = my_driver_probe,
    .remove = my_driver_remove,
};

static DRIVER_ATTR(author, S_IRUGO, show_driver_author, NULL);

static int __init my_driver_init(void){
    int ret;
    ret = driver_register(&virtual_driver);
    if(ret)
        return ret;
    if(driver_create_file(&virtual_driver, &driver_attr_author))
        printk(KERN_NOTICE "Unable to create author attribute/n");
    printk("driver register success/n");
    return ret;
}

static void __exit my_driver_exit(void){
    driver_unregister(&virtual_driver);
}

module_init(my_driver_init);
module_exit(my_driver_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("liwanpeng");
```

Makefile:

```
ifneq ($(KERNELRELEASE),)
    obj-m:= driver.o bus.o device.o
else
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)
modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clear:
    rm -rf *.o
endif
```

测试：

```
root@hacker:/home/hacker/program# dmesg
[  500.120888] bus regiter success
[  503.635832] device register success
[  515.237701] driver can handle the device
[  515.237772] driver register success


root@hacker:/home/hacker/program# dmesg
[  627.552494] bus regiter success
[  631.652273] driver register success
[  641.867854] driver can handle the device
[  641.867861] device register success

root@hacker:/sys/bus/my_bus/drivers/my_dev# ls -l
total 0
-r--r--r-- 1 root root 4096 2011-05-06 22:46 author
--w------- 1 root root 4096 2011-05-06 22:46 bind
lrwxrwxrwx 1 root root    0 2011-05-06 22:46 my_dev -> ../../../../devices/my_bus0/my_dev
--w------- 1 root root 4096 2011-05-06 22:46 uevent
```

```
--w------- 1 root root 4096 2011-05-06 22:46 unbind
root@hacker:/sys/bus/my_bus/drivers/my_dev# cat author
LiWanPeng
```