Linux 2.6 Device Model

v1.2

By semiyd

2007/8/1

semiyd@gmail.com

Linux2.6 的设备模型是一个复杂的数据结构体系，不能用一种简单形象的结构来进行整体的描述。本文试图用一些易于理解的方式，解释这个体系的构造。总体上，会偏向与 driver 的相关内容，与 driver 不太相干的东东会相应的忽略，并且侧重对体系结构的理解，对 device model 中的很多技术细节就不去一一解释了，具体可以看《linux 设备驱动程序》中的相关章节。

全文分为 2 大部分。第一部分是解释整个体系结构。第二部分是结合具体的驱动的内核注册过程，看一下体系的工作流程。

Part One： The Device Model

在介绍这个体系之前，先要说明一下下列一些重要的概念。

Kobject
Kset
Subsystem
Sysfs

Bus
Device
Class
Device_driver

KOBJECT

kobject 是设备模型中的基本的结构。从代码上看，是一个结构体。但是 linux 中引入了面向对象的思想，从某些角度，也可以看成是一个类。

```
struct kobject {
        const char              * k_name; //name of the kobject
        char                    name[KOBJ_NAME_LEN];
        struct kref             kref; //reference counting(引用计数)
        struct list_head        entry;
        struct kobject          * parent; //父类
        struct kset             * kset;
        struct kobj_type        * ktype;
        struct dentry           * dentry;
        wait_queue_head_t       poll;
};
```

kobject 对象通常被嵌入到其他的结构中。从面向对象的观点看，kobject 可以看成是基类，而其他类都是派生的产物。这一点在后面的图表中会有具体的说明。不过这里还是举一个派生的例子：device_driver。

```
struct device_driver {
        const char              * name;
        struct bus_type                 * bus;
        struct completion       unloaded;
        struct kobject                  kobj;
        struct klist            klist_devices;
        struct klist_node       knode_bus;
        struct module           * owner;
        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);
};
```

各种操作：

```
void kobject_init(struct kobject *);
```
初始化

```
int kobject_add(struct kobject *);
```
加入 kset 并且在 sysfs 中显示 kobject。**kobject 不一定会在 sysfs 里显示，除非你调用了这个函数。**

```
int kobject_register(struct kobject *);
```
kobject_init 和 kobject_add 的简单结合

```
void kobject_del(struct kobject *);

void kobject_unregister(struct kobject *);


struct kobject * kobject_get(struct kobject *);
```
增加引用计数

```
void kobject_put(struct kobject *);
```
减少引用计数

kobject 的一个重要函数是为他的结构设置引用计数(reference counting)。只要对象的引用计数存在，对象（以及支持他的代码）就必须存在。

KSET

一个 kset 是相同类型的 kobject 的一个集合。这里所说的类型，是有一个叫 kobj_type 类型的指针来描述的。**kset 一定会在 sysfs 里显示出来。**我们可以认为，他是 kobject 的顶层容器(container)类。在每个 kset 的内部，包含了自己的 kobject，并且可以用多种处理 kobject 的方法处理 kset。

```
struct kset {
        struct subsystem        * subsys;
        struct kobj_type        * ktype;
        struct list_head        list;
        spinlock_t              list_lock;
        struct kobject          kobj;
        struct kset_uevent_ops        * uevent_ops;
};


struct kobj_type {
        void (*release)(struct kobject *);
        struct sysfs_ops        * sysfs_ops;
        struct attribute        ** default_attrs;
};
```

不过这里的 kset 还不一定是顶部的，也就是说，在同属于相同的 kobj_type 类型下，他有可能是另外一个 kset 的一部分。

操作：

```
void kset_init(struct kset * k);

int kset_add(struct kset * k);

int kset_register(struct kset * k);

void kset_unregister(struct kset * k);

static inline struct kset * kset_get(struct kset * k) //改动引用计数

static inline void kset_put(struct kset * k) //改动引用计数
```

SUBSYSTEM

子系统通常就是整个体系的顶层结构。在 sysfs 中，显示为/sys 下面的第一级目录。下面就来看看，通常/sys 下面的目录结构：

block/

bus/

class/

devices/

firmware/

net/

fs/

这里，就可以看见诸如 class（相当于 struct class）之类的子系统。

```
struct subsystem {
        struct kset             kset;
        struct rw_semaphore rwsem; //用于串行访问 kset 内部的链表的信号量
};
```

在看刚刚的 class 例子

```
struct class {
        const char              * name;
        struct module           * owner;
        struct subsystem        subsys;
        struct list_head        children;
        struct list_head        devices;
        struct list_head        interfaces;
        struct semaphore        sem;    /* locks both the children and interfaces lists */
        struct class_attribute          * class_attrs;
        struct class_device_attribute * class_dev_attrs;
        int     (*uevent)(struct class_device *dev, char **envp, int num_envp, char *buffer, int buffer_size);
        void    (*release)(struct class_device *dev);
        void    (*class_release)(struct class *class);
};
```

其实从定义上可以看出，子系统就是对一个 kset 和一个信号量的封装。

操作：

```
extern void subsystem_init(struct subsystem *);

extern int subsystem_register(struct subsystem *);

extern void subsystem_unregister(struct subsystem *);
```

但驱动程序的编写人员是不会用到 subsys 级别的初始化之类的调用的。

SYSFS

sysfs 是一种基于内存的文件结构。简单的说，就是为内核的数据结构，一些属性，符号链接提供一个在用户空间中显示和交互的方式。在系统里，显示为/sys 文件夹。

mount -t sysfs /sys　　就可以在文件系统里看见/sys 的目录了。

每一个在 kernel 里面注册的 kobject，在/sys 里面都会产生一个相应的目录，名字就 kobject 的 name。那么如果这个 kobject 有父类的话，那它对应的目录就会成为父类的子目录，这样就可以反映 kobject 的继承关系。

下面是/sys 下一级目录的结构。

block/

bus/

class/

devices/

firmware/

net/

fs/

这里要提一个叫 attribute 的概念，也就是属性。attribute 是 kobject 里面的东西。在 sysfs 中，属性能够以常规的文件的形式，输出到 sysfs 的目录当中。并且，当 kobject 的属性提供了读写的方法，sysfs 中就可以对相应的文件进行读写，从而访问 kobject 里面的一些信息。

具体在 kobject 里面，有 kobj_type 结构：

```
struct kobj_type {
        void (*release)(struct kobject *);
        struct sysfs_ops     * sysfs_ops;
        struct attribute     ** default_attrs;
};
struct attribute {
        const char              * name;
        struct module                   * owner;
        mode_t                          mode;
};
```

attribute 用于保存属性列表。*name 就是属性名了，也就是 sysfs 里面的文件名。mode 是保护位，如果是只读，就要设置成 S_IRUGO，读写就是 S_IWUSR。

```
struct sysfs_ops {
        ssize_t (*show)(struct kobject *, struct attribute *,char *); //读
        ssize_t (*store)(struct kobject *,struct attribute *,const char *, size_t); //写
};
```

sysfs_ops 就是具体怎么实现在用户空间/sys 下面的读写操作了。

另外，如果希望在 kobject 的 sysfs 目录中添加新的属性，只需要填写一个 attribute 结构体，然后调用下面的函数：

sysfs_create_file(struct kobject * kobj, const struct attribute * attr)

如果把 attribute 看成一个基类的，那这里有一个衍生的类，叫 device_attribute：

```
struct device_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device * dev, char * buf);
        ssize_t (*store)(struct device * dev, const char * buf);
};
```

相应的，也有函数来添加新属性：

```
int device_create_file(struct device *, struct device_attribute *);
void device_remove_file(struct device *, struct device_attribute *);
```

同理有 driver_attribute。

最后要说的就是符号链接。sysfs 具有树形的结构，来反映 kobject 之间的组织关系。但内核中的对象之间的关系远比这个复杂。比如，/sys/devices 的树表示了所有的设备，/sys/bus 的树可以表示设备的驱动。但驱动和设备之间的关系又如何表示呢？这里就需要一个指针，叫符号链接（symlink）来表示这种联系。

```
int sysfs_create_link(struct kobject * kobj, struct kobject * target, const char * name)
```

函数创建了一个叫 name 的链接，指向 target 的 sysfs 入口，并作为 kobj 的一个属性。

到目前为止，介绍了 device model 的底层部分。
下面，将讲述设备模型的上层部分。

BUS

在设备模型中，所有的 device 都通过 bus 相连。这里的 bus 包括通常意义的总线，也包括虚拟的 platform 总线，下面会有具体的说明。总线可以互相插入，比如，usb host 可以是 pci device。bus_type 结构表示了总线：

```
struct bus_type {
        const char              * name;
        struct subsystem        subsys;
        struct kset             drivers;
        struct kset             devices;
        struct klist            klist_devices;
        struct klist            klist_drivers;
        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, char **envp,
                                  int num_envp, char *buffer, int buffer_size);
        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);
};
```

name 是总线的名字。从上面的定义可以看到，每个总线都有自己的子系统，这些子系统并不是指 sysfs 顶层的那个 subsystem。另外包含 2 个 kset。分别代表总线的 device 和 driver。后面可以看到，sysfs 中，每种总线下面，总有 device 和 driver 2 个文件夹。

下面举个例子，说明一下 pci 总线的声明。

```
struct bus_type pci_bus_type = {

        .name = "pci",

        .match          = pci_bus_match,

};
```

值得注意的是，只有很少的 bus_type 成员需要初始化；大部分都可以交给 kernel 处理。

有关总线的操作函数，这里仅仅介绍一个常用的：

```
int bus_for_each_dev(struct bus_type * bus, struct device * start,
                     void * data, int (*fn)(struct device *, void *))
{
        struct klist_iter i;
        struct device * dev;
        int error = 0;

        if (!bus)
                return -EINVAL;

        klist_iter_init_node(&bus->klist_devices, &i,
                             (start ? &start->knode_bus : NULL));
        while ((dev = next_device(&i)) && !error)
                error = fn(dev, data);
        klist_iter_exit(&i);
        return error;
}
```

他的作用是遍历 bus 上的 device，并进行 fn 操作。

通常的用途是：绑定设备和驱动：

```
void driver_attach(struct device_driver * drv)
{
        bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
}
static int __driver_attach(struct device * dev, void * data)
{
        struct device_driver * drv = data;

        /*
         * Lock device and try to bind to it. We drop the error
         * here and always return 0, because we need to keep trying
         * to bind to devices and some drivers will return an error
         * simply if it didn't support the device.
         *
         * driver_probe_device() will spit a warning if there
         * is an error.
         */
```

```
        if (dev->parent)        /* Needed for USB */
                down(&dev->parent->sem);
        down(&dev->sem);
        if (!dev->driver)
                driver_probe_device(drv, dev);
        up(&dev->sem);
        if (dev->parent)
                up(&dev->parent->sem);

        return 0;
}
```

几乎在 linux 设备模型的每一层都提供了添加属性的函数，总线也不例外。

```
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);
};
```

下面说明一下 bus 在 sysfs 里面的结构。刚刚已经提到了，由于有 2 个 kset，那么每个 bus 都会有 driver 和 device2 个文件夹。

具体的说：

每个在总线上注册过的驱动都会得到一个文件夹，以 pci 总线上的驱动为例：

```
        /sys/bus/pci/
        |-- devices
        `-- drivers
            |-- Intel ICH
            |-- Intel ICH Joystick
            |-- agpgart
            `-- e100
```

而任何在总线/sys/bus/xxx/上发现的设备会得到一个 symlink(符号链接)，也就是/sys/bus/xxx/devices，指向/sys/devices/xxx 下面的文件夹

```
        /sys/bus/pci/
        |-- devices
        |    |-- 00:00.0 -> ../../../root/pci0/00:00.0
        |    |-- 00:01.0 -> ../../../root/pci0/00:01.0
        |    `-- 00:02.0 -> ../../../root/pci0/00:02.0
        `-- drivers
```

symlink 在 sysfs 中，可以理解为简单的文件夹快捷方式，你会发现在 gnome 之类的 GUI 下，查看这些文件夹，图标就是快捷方式。

DEVICE

先看下 device 的结构体：

```
struct device {
        struct klist            klist_children; // List of child devices.
        struct klist_node       knode_parent;           /* node in sibling list */
        struct klist_node       knode_driver; // Node in device's driver's devices list.
        struct klist_node       knode_bus; //Node in device's bus's devices list.
        struct device   * parent; /*  该设备所属的设备  如果 parent 是 NULL，表示是顶层设备  */

        struct kobject kobj;/*  表示该设备并把它连接到体系结构中的 kobject    */
        char    bus_id[BUS_ID_SIZE];        /* position on parent bus 在总线上唯一标识该设备的字符串。比如 PCI 设备，就是 PCI ID */
        struct device_attribute uevent_attr;
        struct device_attribute *devt_attr;

        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 is manipulating this
                                          device */
        void            *driver_data;  /* data private to and used by the device driver */
        void            *platform_data;         /* Platform specific data, device
                                          core doesn't touch it */
        void            *firmware_data; /* Firmware specific data (e.g. ACPI,
                                          BIOS data),reserved for device core*/
        struct dev_pm_info    power;

        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 list_head        dma_pools;     /* dma pools (if dma'ble) */

        struct dma_coherent_mem    *dma_mem; /* internal for coherent mem
                                             override */

        /* class_device migration path */
        struct list_head        node;
        struct class            *class;         /* optional*/
        dev_t                   devt;           /* dev_t, creates the sysfs "dev" */

        void    (*release)(struct device * dev);
};
```

基本上每个成员的作用都作了详细的注释。

对于基于特定总线的设备，会有衍生类，诸如：

```
struct platform_device {
        const char      * name;
        u32             id;
        struct device           dev;
        u32             num_resources;
        struct resource         * resource;
};
```

一个用来发现设备的总线驱动需要用下面的函数注册设备：

```
int device_register(struct device *dev)
{
        device_initialize(dev);
        return device_add(dev);
}
```

总线要初始化下面的几个成员：

- parent
- name
- bus_id
- bus

完成以后，会在/sys/devices 下面看到。

下面是与属性相关的内容：

```
struct device_attribute {
          struct attribute          attr;
          ssize_t (*show)(struct device * dev, char * buf, size_t count, loff_t off);
          ssize_t (*store)(struct device * dev, const char * buf, size_t count, loff_t off);
};
int device_create_file(struct device * dev, struct device_attribute * attr)
void device_remove_file(struct device * dev, struct device_attribute * attr)
```

CLASS

类的概念，实际上就是一个比较高层次上的视图。也就是说，它略去了一些低层的细节，而着重与从功能上去把设备分类，并不关心设备具体的连接方式。比如说，驱动程序看到的是 SCSI 磁盘和 ATA 磁盘，并且属于块设备，但在类的意义上，他们都属于磁盘类而已。所以说类的概念更 user-friendly。

```
struct class {
        const char              * name;
        struct module           * owner;

        struct subsystem        subsys;
        struct list_head        children;
        struct list_head        devices;
        struct list_head        interfaces;
        struct semaphore        sem;    /* locks both the children and interfaces lists */

        struct class_attribute          * class_attrs;
        struct class_device_attribute * class_dev_attrs;

        int     (*uevent)(struct class_device *dev, char **envp,
                          int num_envp, char *buffer, int buffer_size);

        void    (*release)(struct class_device *dev);
        void    (*class_release)(struct class *class);
};
```

有关类的就不多介绍了。

DEVICE_DRIVER

```
struct device_driver {
        const char              * name; /* The name of the driver.Will be shown in the sysfs */
        struct bus_type                 * bus; /* the specific bus this driver is manipulating */

        struct completion       unloaded;
        struct kobject                  kobj;
        struct klist            klist_devices; /* driver 操作的设备链表*/
        struct klist_node       knode_bus;

        struct module           * owner;

        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);
};
```

在声明一个 device_driver 的时候，通常至少要声明下面一些成员：举个例子：

```
static struct device_driver eepro100_driver = {
        .name           = "eepro100",
        .bus            = &pci_bus_type,

        .probe          = eepro100_probe,
        .remove                 = eepro100_remove,
        .suspend                = eepro100_suspend,
        .resume                 = eepro100_resume,
};
```

这是一个很简单的例子。不过在现实当中，大多数的驱动会要求带有自己所特有的针对某种特定总线的信息。这些信息不是上面的模型可以全部包含的。比较典型的例子是：带有 device ID 的基于 PCI 总线的驱动：

```
struct pci_driver {
        const struct pci_device_id *id_table;
        struct device_driver            driver;
};
```

所以就有了上面这个从 device_driver 衍生出来的 pci_driver 类。注意，这时候的 device_driver 变成了 pci_driver 的一个成员。

同样的道理，还会有 platform_driver 等等。那么声明一个 pci_driver 的过程就变成：

```
static struct pci_driver eepro100_driver = {
        .id_table           = eepro100_pci_tbl,
        .driver           = {
                .name           = "eepro100",
                .bus            = &pci_bus_type,
                .probe          = eepro100_probe,
                .remove                 = eepro100_remove,
                .suspend        = eepro100_suspend,
                .resume                 = eepro100_resume,
        },
};
```

驱动的注册：

```
/**
 *      driver_register - register driver with bus
 *      @drv:  driver to register
 *
 *      We pass off most of the work to the bus_add_driver() call,
 *      since most of the things we have to do deal with the bus
 *      structures.
 *
 *      The one interesting aspect is that we setup @drv->unloaded
 *      as a completion that gets complete when the driver reference
 *      count reaches 0.
 */
int driver_register(struct device_driver * drv)
{
        if ((drv->bus->probe && drv->probe) ||
            (drv->bus->remove && drv->remove) ||
            (drv->bus->shutdown && drv->shutdown)) {
                printk(KERN_WARNING "Driver '%s' needs updating - please use bus_type methods\n", drv->name);
        }
        klist_init(&drv->klist_devices, klist_devices_get, klist_devices_put);/*其中包括初始化锁的内容*/
        init_completion(&drv->unloaded);
```

```
        return bus_add_driver(drv);
}
```

不过正如刚才提到的，其实大多数驱动会调用针对特定总线的诸如 pci_driver_register，platform_driver_register 之类的函数去注册。

和 BUS 一样，device_driver 也有一个函数用来遍历。

```
int driver_for_each_dev(struct device_driver * drv, void * data,int (*callback)(struct device * dev, void * data));
```

所有的 device_driver 在完成注册后，都会在所属的 bus 目录下，产生一个新的目录。

下面分析一下 device_driver 的一个 callback 函数，也就是刚刚提到的：

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

Probe 的入口是在 bus 的 rwsem 信号量（之前提到的一个 subsystem 的信号量）锁定，并且驱动部分绑定到设备的时候被调用的。它将检测这个设备是否可以支持，是否可以初始化。probe 经常在里面调用 dev_set_drvdata()来保存指向驱动特有信息的指针。当 probe 把它做要做的 bind 工作完成好的时候，返回一个 0，这样设备模型就会完成余下的绑定工作。

也许有人会问，一般驱动注册过程中，probe 函数到底在什么时候执行呢？好像不是很直接能看出。

这个问题先暂时搁在那，在 part two 讲了驱动注册流程后，再加以说明。

最后说下属性的内容。

```
struct driver_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device_driver *, char * buf, size_t count, loff_t off);
        ssize_t (*store)(struct device_driver *, const char * buf, size_t count, loff_t off);
};

int driver_create_file(struct device_driver *, struct driver_attribute *);
void driver_remove_file(struct device_driver *, struct driver_attribute *);
```

上面说了这么多，现在就要理一理 kobject,kset,subsys,sysfs,bus.....等等东西之间，到底是什么样的一个关系。
在弄清楚关系之前，先要看懂下面的这张图：

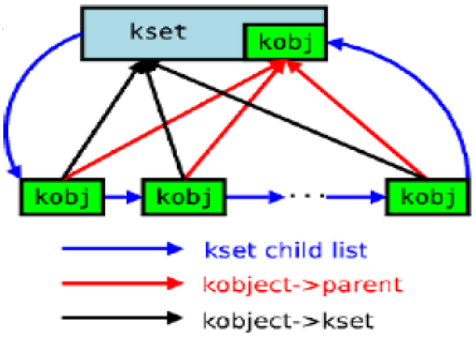


图-1

这张图反映的，是 kobject 继承体系的一个基本的结构。首先，有必要重新再说一下 kset 的功能：kset 是一组相同的 kobject 的集合，所以 kernel 可以通过跟踪 kset，来跟踪“所用的 pci 设备”，或者所有的块设备之类。kset 起到了连接的作用，它把设备模型和 sysfs 目录结合在了一起。每个 kset 自身，都含有一个 kobject，这个 kobject 将作为很多其他的 kobject 的父类。从 sysfs 上看，某个 kobject 的父类是某个目录，那它就是那个目录的子目录。parent 指针可以代表目录层次。这样，典型的设备模型的层次就建立起来了。从面向对象的观点看，kset 是顶层的容器类。kset 继承他们自己的 kobject，并且可以当作 kobject 来处理。

如图：kset 把它的子类（children）放在一个 kernel 链表里（list）。kset 子类链表里面的那些 kobject 的 kset 指针指向上面的 kset，parent 指向父类。大致就是上面画的这个结构。有 2 个注意点：1.上面的图里的 kobject 通常是已经被嵌入到其他结构中去了，有可能，已经嵌入到某个 kset 中了。2.kobject 的 parent 可以是单纯的 kobject，而不一定要是顶层的容器 kset 里的 kobject。

```
struct kobject {
        const char              * k_name; //name of the kobject
        char                    name[KOBJ_NAME_LEN];
        struct kref             kref; //reference counting(引用计数)
        struct list_head        entry;
        struct kobject          * parent; //父类
        struct kset             * kset;
        struct kobj_type        * ktype;
        struct dentry           * dentry;
        wait_queue_head_t       poll;
};
struct kset {
        struct subsystem        * subsys;
        struct kobj_type        * ktype;
        struct list_head        list;
        spinlock_t              list_lock;
        struct kobject          kobj;
        struct kset_uevent_ops          * uevent_ops;
};
```

看懂了图-1，下面就可以用类似的图来描述一下 subsystem,bus 还有 device,class,device_driver. *注：这里 **kset** 图中的 **kset** 是指它自己特有的一些成员，如不重要恕不列出。绿色代表可以代入。*

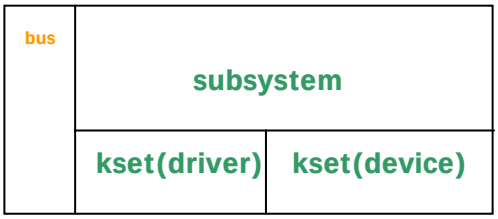
Kset

kset	kobject

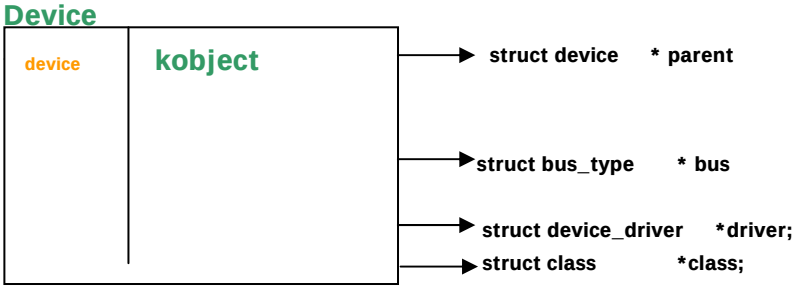
Subsystem

subsystem	kset

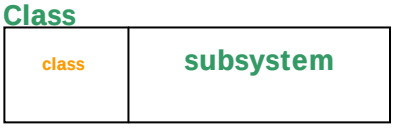
BUS



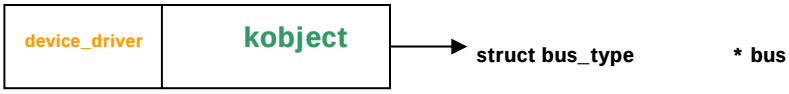
Device



Class

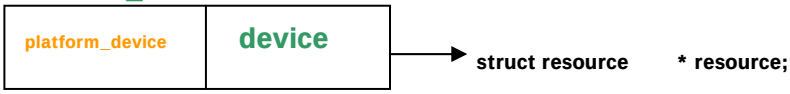


Device_driver



有了这些基本的单元，就可以衍生出 N 多其他的新的类别，如 platform_device:

Platform_device



这样我们就可以在 sysfs 下看一下实际的结构是怎么组织的。我们以 LCD 设备作为一个例子。

在这之前，先简要地提一下一个概念：platform。

platform 是一个虚拟的总线。相比 PCI,USB，它主要用于描述 SOC 上的一些资源，比如说，s3c2410 上集成的控制器。platform 所描述的资源有一个共同点，就是在 cpu 的总线上直接取址。LCD 控制器，就是这样一个 platform device。下面列举一些 platform 相关的数据结构：

下面的这些结构都是上述的基本的单元的衍生类，读者可以自己画格子来表示这些衍生的新类。

```
struct bus_type platform_bus_type = {
        .name           = "platform",
        .dev_attrs      = platform_dev_attrs,
        .match          = platform_match,
        .uevent                 = platform_uevent,
        .suspend        = platform_suspend,
        .resume                 = platform_resume,
};
struct platform_device {
        const char      * name;
        u32             id;
        struct device   dev;
        u32             num_resources;
        struct resource         * resource;
};
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*resume)(struct platform_device *);
        struct device_driver driver;
};
```

这些数据结构，会是下面的图表的基本单元之一。

首先看一下命令行下面，sysfs 的结构：
（这里只看 class,devices,bus,其他忽略。）

注：绿色：symlink，也就是符号链接。后面括号里写上了链接的指向。

CLASS

/sys/class

graphics mem ppdev tty vc
hwmon misc printer usb_device vtconsole
i2c-adapter mtd scsi_device usb_endpoint
input net scsi_host usb_host

/sys/class/graphics

fb0 fbcon

/sys/class/graphics/fb0

bits_per_pixel dev name stride
blank device（/sys/devices/platform/s3c2410-lcd） pan subsystem（/sys/class/graphics）
console mode rotate uevent
cursor modes state virtual_size

DEVICES

/sys/devices

isa platform system

/sys/devices/platform

power s3c2410-lcd s3c2410-uart.0 s3c2410-wdt
s3c2410-i2c s3c2410-nand s3c2410-uart.1 serial8250
s3c2410-iis s3c2410-ohci s3c2410-uart.2 uevent

/sys/devices/platform/s3c2410-lcd

bus（/sys/bus/platform） driver（/sys/bus/platform/drivers/s3c2410-lcd） modalias subsystem（/sys/bus/platform）
debug graphics:fb0（/sys/class/graphics/fb0） power uevent

BUS

/sys/bus

i2c ide isa platform scsi serio usb

/sys/bus/platform

devices drivers

/sys/bus/platform/devices

s3c2410-i2c s3c2410-nand s3c2410-uart.1 serial8250
s3c2410-iis s3c2410-ohci s3c2410-uart.2
s3c2410-lcd s3c2410-uart.0 s3c2410-wdt 全是/sys/devices/platform 下面的链接

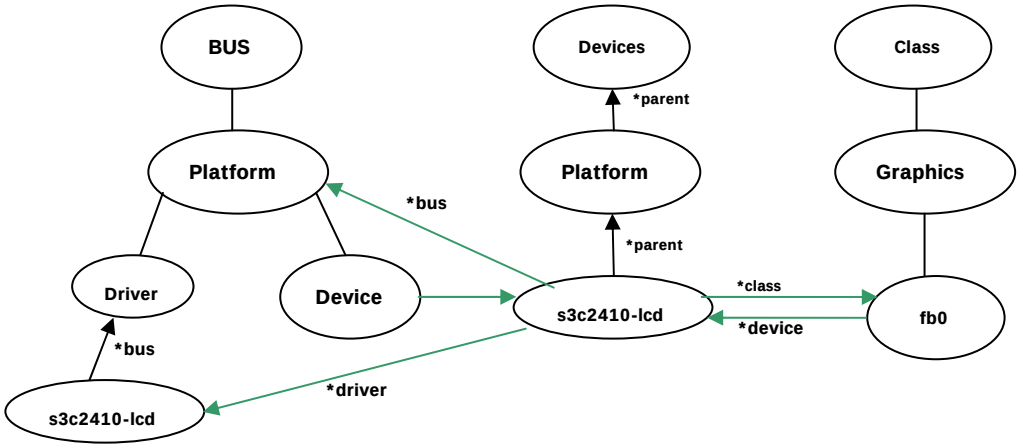
/sys/bus/platform/drivers

bast-nor s3c2410-nand s3c2410-uart s3c2412-uart s3c2440-uart
s3c2410-i2c s3c2410-ohci s3c2410-wdt s3c2440-i2c serial8250
s3c2410-lcd s3c2410-rtc s3c2412-nand s3c2440-nand

/sys/bus/platform/drivers/s3c2410-lcd

bind s3c2410-lcd(/sys/devices/platform/s3c2410-lcd) unbind

于是，LCD 设备的图就应该这么表示：



注：

1.绿色箭头代表 symlink

2.椭圆内的是 entity，也就是说，可以是 kset,kobject。

3.图中的名字是依据 sysfs 里面的命名来画的,与 device model 结构的对应关系举例如下：

比如说，bus->platform->driver->s3c2410-lcd:

s3c2410-lcd 是 name，其实是叫 s3c2410fb_driver 的 platform_driver

```
static struct platform_driver s3c2410fb_driver = {
        .probe          = s3c2410fb_probe,
        .remove                 = s3c2410fb_remove,
        .suspend        = s3c2410fb_suspend,
        .resume                 = s3c2410fb_resume,
        .driver         = {
                .name  = "s3c2410-lcd",
                .owner = THIS_MODULE,
        },
};
```

Part Two：Driver Register Flow

这一部分以platform driver注册为例，跟踪一下驱动的内核注册过程。

注册函数是：platform_driver_register()。

```
platform_driver_register(struct platform_driver *drv)
    driver_register(struct device_driver * drv)
        void klist_init(struct klist * k, void (*get)(struct klist_node *),void (*put)(struct klist_node *))
            INIT_LIST_HEAD(&k->k_list)
        init_completion(struct completion *x)
            init_waitqueue_head(wait_queue_head_t *q)
                INIT_LIST_HEAD(&k->k_list)
        bus_add_driver(struct device_driver * drv)
            kobject_set_name(struct kobject * kobj, const char * fmt, ...) //设置kobject的名字
            kobject_register(struct kobject * kobj) //注册kobject
                kobject_init(struct kobject * kobj)   //初始化kobject的成员
                    kref_init(struct kref *kref)   //set reference counting to 1
                        atomic_set(&kref->refcount,1);
                    INIT_LIST_HEAD(&kobj->entry);
                    init_waitqueue_head(&kobj->poll);
                kobject_add(struct kobject * kobj)   /把kobject添加到对应的kset体系中，也就是说，在sysfs中显示kobject
            driver_attach(struct device_driver * drv) //遍历这个bus上的所有的设备，看有没有和driver相匹配的
                bus_for_each_dev(struct bus_type * bus, struct device * start,void * data, int (*fn)(struct device *, void *)) //用来操作注册到总线上的每个设备
                    klist_iter_init_node(struct klist * k, struct klist_iter * i, struct klist_node * n)
                    klist_iter_exit(struct klist_iter * i)
            klist_add_tail(struct klist_node * n, struct klist * k)
                klist_node_init(struct klist * k, struct klist_node * n)
                    INIT_LIST_HEAD(&n->n_node);
                    init_completion(&n->n_removed);
                    kref_init(struct kref *kref)
            module_add_driver(struct module *mod, struct device_driver *drv)
                sysfs_create_link(struct kobject * kobj, struct kobject * target, const char * name) //创建符号链接name指向target的sysfs入口，并作为kobj的一个属性
            driver_add_attrs(struct bus_type * bus, struct device_driver * drv)
                driver_create_file(struct device_driver * drv, struct driver_attribute * attr)     //add driver attribute to driver dir.
                    sysfs_create_file(struct kobject * kobj, const struct attribute * attr)     //在kobject的sysfs目录中添加新属性
                        sysfs_add_file(struct dentry * dir, const struct attribute * attr, int type)
            add_bind_files(struct device_driver *drv)
                driver_create_file(struct device_driver * drv, struct driver_attribute * attr)   //add driver attribute to driver dir.
```

上面的结构是这样的：

从platform_driver_register开始，driver_register是它调用的函数，所以换一行，加一个制表符间距。后面的函数调用以此类推。给出了大多数关键函数的注释，并忽略了不重要的函数调用部分。

前面提到了probe()函数的调用位置。这里就来分析一下。

```
platform_driver_register(struct platform_driver *drv)
    driver_register(struct device_driver * drv)
        bus_add_driver(struct device_driver * drv)
            driver_attach(struct device_driver * drv)
                bus_for_each_dev(struct bus_type * bus, struct device * start,void * data, int (*fn)(struct device *, void *))
```

到这一步，根据源码，看下bus_for_each_dev的其中的*fn的具体的参数，是

```
static int __driver_attach(struct device * dev, void * data)
```

其中有driver_probe_device(drv, dev);

这里面的

```
if (dev->bus->probe) {
        ret = dev->bus->probe(dev);
        if (ret) {
            dev->driver = NULL;
            goto ProbeFailed;
        }
    } else if (drv->probe) {
        ret = drv->probe(dev);
        if (ret) {
            dev->driver = NULL;
            goto ProbeFailed;
        }
```

便是了。这里的关键是如果不知道*bus_for_each_dev*的功能，很可能找不到到底probe哪里调用了。

说到这里，顺便详细说明下*bus_for_each_dev*吧。

bus_for_each_dev()：

```
int bus_for_each_dev(struct bus_type * bus, struct device * start,
                     void * data, int (*fn)(struct device *, void *))
{
    struct klist_iter i;
    struct device * dev;
```

```
        int error = 0;

        if (!bus)
                return -EINVAL;

        klist_iter_init_node(&bus->klist_devices, &i,
                             (start ? &start->knode_bus : NULL));
        while ((dev = next_device(&i)) && !error)
                error = fn(dev, data);
        klist_iter_exit(&i);
        return error;
}
```

这个函数的作用是这样的：从start开始，遍历(iterate)bus上的所有设备，对找到的设备，调用fn(device,data)，也就是对参数device,data进行fn操作。这个过程，要临时建立一个遍历链表klist_iter用来遍历。

调用参数是bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);

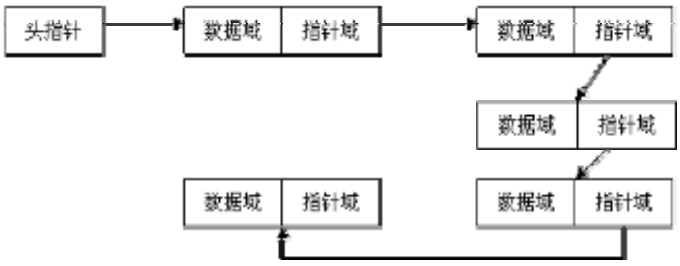
有了参数，具体的功能就是对bus下面的设备进行驱动的绑定。这里涉及的klist_iter就不细说了。

读者会发现，在上面的流程当中，除了注册kobject,sysfs,驱动等等，还有很多诸如*klist_init*（）之类的和链表操作有关的函数。下面就介绍一下与链表有关的内容。

下面的内容是这么安排的。我们一边讲解linux的链表的组织和操作，一边结合上面的platform驱动注册的流程中，设备链表的部分（链表涉及了太多的数据结构，在整个驱动注册流程中，与设备链表无关的，一律不提，以免过于复杂），进行分析。等我们把驱动注册流程中关于链表的内容走完一遍以后，读者对linux的链表操作会有一个比较清晰的印象。

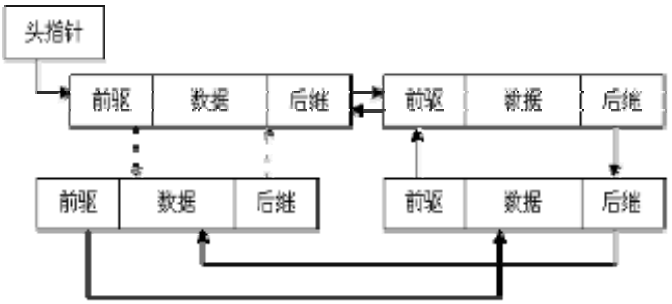
链表在教科书上一般是这么分类的：单链表和双链表：

图 1 单链表



单链表是最简单的一类链表，它的特点是仅有一个指针域指向后继节点（next），因此，对单链表的遍历只能从头至尾（通常是 NULL 空指针）顺序进行。

图 2 双链表



通过设计前驱和后继两个指针域，双链表可以从两个方向遍历，这是它区别于单链表的地方。如果打乱前驱、后继的依赖关系，就可以构成"二叉树"；如果再让首节点的前驱指向链表尾节点、尾节点的后继指向首节点（如图 2 中虚线部分），就构成了循环链表；如果设计更多的指针域，就可以构成各种复杂的树状数据结构。

另外还有个循环链表。而链表的节点，经典的定义是（以单链表为例）：

```
struct list_node {
```

```
        struct list_node *next;
        ElemType        data;
};
```

换句话说，就是有个指针，一个数据项。现在看看linux的定义（list.h）：

```
struct list_head {
        struct list_head *next, *prev;
};
```

一看，发现是双链表的节点，可是怎么就没了数据项？

理由是这样的：在linux当中，不是在链表中包含数据，而是在数据结构体中包含链表节点来实现链接。

现在可以结合platform driver分析一下了。流程中有这么一步：

*void klist_init(struct klist * k, void (*get)(struct klist_node *),void (*put)(struct klist_node *))*

初始化klist。首先看看，什么是klist:

```
struct klist {
        spinlock_t              k_lock;
        struct list_head        k_list;
        void                    (*get)(struct klist_node *);
        void                    (*put)(struct klist_node *);
};
```

klist是一个包含链表的数据结构。他是device,kobject,等诸多结构体的成员。这里就体现了linux的设计思想：在数据结构体中包含list_head/klist及其衍生的链表结构以实现关联。这种层层嵌入，由一些基本的单元完成链接的数据组织方式，和kobject的体系如出一辙。可以看出，包含了list_head链表，spinlock_t自旋锁，还有2个函数，get和put。再来看klist_init()：

```
void klist_init(struct klist * k, void (*get)(struct klist_node *),void (*put)(struct klist_node *))
{
        INIT_LIST_HEAD(&k->k_list);
        spin_lock_init(&k->k_lock);
        k->get = get;
        k->put = put;
}
```

INIT_LIST_HEAD(&k->k_list);就是初始化链表头，把prev和next指针都指向自己。

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
        list->next = list;
        list->prev = list;

}
```

调用的参数是：klist_init(&drv->klist_devices, klist_devices_get, klist_devices_put);

klist_devices_get和 klist_devices_put分别是增加和减少klist所属的device结构体的reference count。

这样klist_init(&drv->klist_devices, klist_devices_get, klist_devices_put)的作用就是：初始化自旋锁和引用计数的操作，初始化device_driver结构体里面，**klist_devices**这个klist设备链表。

```
struct device_driver {
        const char              * name; /* The name of the driver.Will be shown in the sysfs */
        struct bus_type                 * bus; /* the specific bus this driver is manipulating */
        struct completion       unloaded;
        struct kobject          kobj;
        struct klist            klist_devices; /* driver 操作的设备链表*/
        struct klist_node       knode_bus;
        ......
};
```

完成了klist_init()以后，就要执行init_completion()，与device_driver里面的unloaded完成标志有关，表示链表初始化完成。这个函数的细节这里暂不讨论，和设备链表关系不大。

下一个要关注的是这个函数，klist_add_tail。

```
void klist_add_tail(struct klist_node * n, struct klist * k)
{
        klist_node_init(k, n);
        add_tail(k, n);
}
```

功能是，初始化一个klist_node节点并加到klist屁股后面去。这里介绍一下klist_node:

```
struct klist_node {
        struct klist            * n_klist;
        struct list_head        n_node;
        struct kref             n_ref;
        struct completion       n_removed;
};
```

前面介绍了klist，可以看成是头节点，那么这个klist_node可以看成是被头节点链接的普通的节点。

具体的调用是：

klist_add_tail(&drv->knode_bus, &bus->klist_drivers);

现在大家就清楚了，原来是初始化device_driver里面的knode_bus节点，然后把这个节点添加到device_driver所属的bus里面的klist_drivers链表后面去。简单点说就是驱动向总线注册。至此，**knode_bus**搞定了。

现在可以总结一下了，刚刚讲了那么多，一共做了什么呢？

一共做了2件事：初始化了device_driver里面的2个与klist有关的成员：

```
struct device_driver {
        ......
        struct klist            klist_devices;
        struct klist_node       knode_bus;
        ......
};
```

具体的说，就是初始化klist_devices；将knode_bus注册到所属的bus的klist_driver里面去。

这里有人要问，klist_devices只是完成了初始化，是个空的链表，没有包含任何设备信息啊。

其实是这样的，完成初始化后，在刚刚提到过的__driver_attach()函数里，有个driver_probe_device()，里面有device_bind_driver()：

```
void device_bind_driver(struct device * dev)
{
        if (klist_node_attached(&dev->knode_driver))
                return;

        pr_debug("bound device '%s' to driver '%s'\n",
                 dev->bus_id, dev->driver->name);
        klist_add_tail(&dev->knode_driver, &dev->driver->klist_devices);
        sysfs_create_link(&dev->driver->kobj, &dev->kobj,
                          kobject_name(&dev->kobj));
        sysfs_create_link(&dev->kobj, &dev->driver->kobj, "driver");
}
```

里面的黑体的这句话，就是关键了：把device结构体的knode_driver初始化，然后加到device_driver里面的klist_devices后面去。

至此，有关设备链表的内容就讲完了。