06_设备模型

一、概述（设备模型的意义）

过去linux系统下的设备驱动都是独立的，驱动与驱动之间没有什么实质的联系。当随着linux的发展，面对庞大的系统和丰富的硬件设备，过去的设备驱动机制遇到了挑战，例如有的设备不是单纯的一个驱动实现，u盘这种常见的设备就是典型，实际上u盘是连接在一个usb
hub上，而usb hub又是连接在usb 2.0 host controller（ECHI）上,而ECHI又是一个接在pci bus上的设备。这里就有一个这样的层级关系:

pci->ECHI->usb_hub->usb_disk

当系统需要休眠时，系统需要逐层的通知所有的外设进入休眠模式（强制退出是非常不可取的），最后整个系统才能休眠。这就要求要在驱动层面建立完善的电源管理机制，如果能把找一根“线”，这有联系的设备串起来，当要逐层通知休眠时，只需要顺着这根“线”逐个通知就能很简单的完成这个要求。即建立一个组织所有设备和驱动的树状结构，用户就可以通过这棵树去遍历所有的设备，建立设备和驱动程序之间的联系，根据类型不同也可以对设备进行归类，这样就可以更清晰的去“看”这颗枝繁叶茂的大树。怎么让外界能“看”到这课大树完整的结构呢？这就需要通过sysfs文件系统对外展示出它的各个“枝干”。通过sysfs文件系统的目录层次结构，我们就能清晰的看到设备驱动之间的互联层次关系，而且能通过文件接口的方式实现与外界的沟通与互动。其实早期的linux系统是通过proc文件系统完成对设备控制（即完成驱动的基本功能），本来完全是可以在proc文件系统上展示这棵大树，但是专家认为proc文件系统非常混乱落后，所以就推出全新sysfs文件系统，让sysfs成为承载设备驱动这课大树的土壤。我们对sysfs的先了解到这儿，至于怎么用，和设备模型又有怎么的实质联系，后面会有详细的讲解,在此只做一个全局性的讲解，方便之后对更复杂的内容的理解。

那么sysfs文件系统的作用是什么呢。概括的说有三点：

1、建立系统中总线、驱动、设备三者之间的桥梁

2、向用户空间展示内核中各种设备的拓扑图

3、提供给用户空间对设备获取信息和操作的接口，部分取代ioctl功能。

二、内核对象数据结构分析

从2.6版本的内核开始，采用设备模型机制来统一管理设备，思路是简单的，但是设备模型是一个非常复杂的数据结构。那么学习的一个重点就是分析这个复杂的数据结构。虽然复杂，但是细心的分析过来也不是很难理解的。在讲设备模型，最低层的数据结构前，我们先补充点基础知识，OO思想（面向对象）。c语言是面向过程语言，c++和java是面向对象语言。其实面向对象主要就是多了一个“类”的概念，即class，c++也称为“带类的c语言”。通过“类”来实现继承和派生的特征。c语言其实是能通过结构体来实现面向对象，我们的gtk+就是这样做的。就举gtk+的例子，控件有个非常基础的结构体struct widget，其他的控件都是要包含他，在构建数据结构的时候再适当加一些别的属性和方法来创造出新的控件。简单说就是通过包含一个基础的结构体来建立新的结构体。那么我们就先来学习几个基础且重要的结构体。

内核对像strcut kobject

构建设备模型的最底层的结构体，如果学过面向对象，那么就可以把它理解成“基类“，其他更高级的对象都是由他派生出来的。内核很少单独的使用这样一个结构体，它存在的意义是在于把更高级的对象（数据结构）连接到设备模型上。一般都是构建一个具体的对象时包含他，这就是前面我们提到过的通过结构体实现面向对象思想，我们可以看看cdev结构体，他是包含kobject的。包含kobject还有很多，我们会以platform总线为例，来具体讲解怎么用kobject构建设备模型的。

```
struct kobject {
    const char      *name;
    struct list_head    entry;
    struct kobject      *parent;
    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;
};
```

name　用来表示该内核对象的名称。将来会出现在sys文件系统中，表现形式就是一个新的目录名。
entry　用来将一系列内核对象构成列表（内核链表结构体，双向的）。
parenet　指向该内核链表的上层节点。通过该成员实现内核对象之间的层次化关系。
kset　当前内核对象所属的内核对象集合的指针。
ktype　定义了该内核对象的一组sysfs文件系统相关的操作函数和属性。显然不同类型的内核对象会有不同的ktype。也是通过这个结构体实现了
c++中class类型的某些特点，体现出基于c的面向对象设计思想。
sd　用来表示内核对象在sysfs文件系统中对于的目录项的实例

kref　用来表示内核对象的引用计数，其核心的数据是一原子变量
state_initialized　表示该kobject所表示的内核对象初始化状态，1表示对象已被初始化，0表示尚未初始化
state_in_sysfs　表示该内核对象有没有在sysfs文件中建立以个入口点
state_add_uevent_sent　如果该内核对象
state_remove_uevent_sent
uevent_suppress　如果该内核对象隶属于某个kset，那么他的状态变化可以导致其所在的kset对象向用户控件发送event消息。成员uevent_suppress用来表示当该kobject状态发生变化是，是否让其所在的kset向用户空间发送event消息。置1表示不让kset发送这种event消息。

内核对像集合 strcut kset

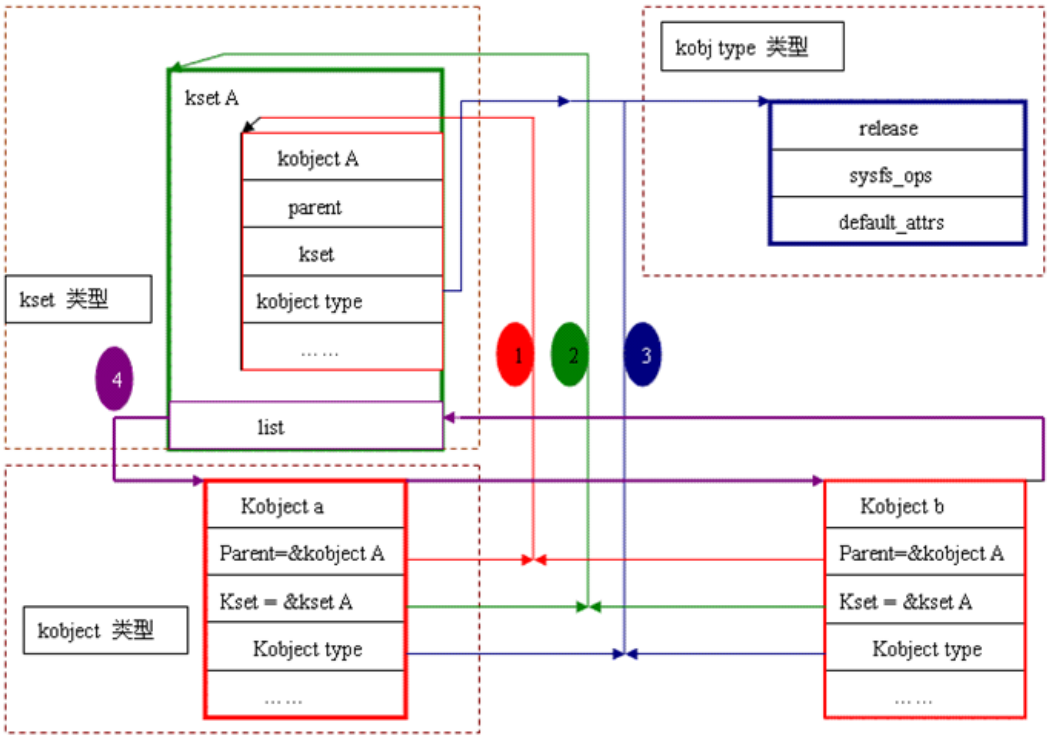
kset可以认为是一组kobject的集合，就是kobject的容器。kset本身也是以个内核对象，所以需要内嵌一个kobject对象。

```
struct kset {
    struct list_head list;
    spinlock_t list_lock;
    struct kobject kobj;
    const struct kset_uevent_ops *uevent_ops;
};
```

list　用来将其中的kobject对象构建成链表
list_lock　对kset上的list链表进行访问操作时用来作为互斥保护使用的自旋锁
kobj　代表当前kset内核对象的kobject变量
uevent_ops　定义了一组函数指针，当kset中的某些kobject对象发生状态变化需要通知用户空间时，调用其中的函数来完成。

```
struct kset_uevent_ops {
    int (* const filter)(struct kset *kset, struct kobject *kobj);
    const char *(* const name)(struct kset *kset, struct kobject *kobj);
    int (* const uevent)(struct kset *kset, struct kobject *kobj,
              struct kobj_uevent_env *env);
};
```

kset 和kobject之间的各个成员的联系是比较复杂的，我们来通过这个图来分析。
kset和kobject都有构成链表所需的成员，类型是struct list_head。将所有的kset和kobject连在一起（双向链表），
kset是kobject的容器，可以容纳多个kobject，通过kobject的kset指针变量指向似个kset，而且他们的parent都是指向kset所包含的kobject，而且kobject_type都是和kset里的kobject指向同一kobj type。



kobject、sysfs_file、kset相关接口
int kobject_set_name(struct kobject *kobj,const char *fmt,...)
给内核对象设置名字

void kobject_init(struct kobject *kobj,struct kobj_type *ktype)->void kobject_init_internal(struct kobject *kobj)(初始化引用计数，链表指针，还有各个状态位的初始化)
初始化内核对象

int kobject_add(struct kobject *kobj,struct kobject *parent,const char *fmt,...)
在sysfs中创建目录（建立内核对象的层次关系）
int kobject_init_and_add(struct kobject *kobj,struct kobj_type *ktype,struct kobject *parent,const char *fmt,...)

struct kobject *kobject_create(void)
分配和初始化一个kobject对象,但是不能指定kobj_type，只能选择默认的。
struct kobject *kobject_create_and_add(const char *name,struct kobject *parent)

void kobject_del(struct kobject *kobj)
删除sysfs中内核对象对应的目录，如果有所属的kset，那么在kset链表中删除该内核对象

int sysfs_create_file(struct kobject *kobj,const struct attribute *attr)
创建内核对象属性文件

int sysfs_open_file(struct inodp *inode,struct file *filp)
打开内核对象属性文件，最终是在filp->private_data存放该对象的属性信息的入口（地址）

int sysfs_remove_file(struct kobject *kobj,const struct attribute *attr)
删除内核对象属性文件

void kset_init(struct kset *k)
初始化kset

int kset_regiter(struct kset *k)
初始化并向内核注册kset

void kset_unregister(struct kset *k)
从内核中注销kset

三、热插拔(hotplug)

当一个设备动态的加入系统时（u盘这样典型的设备接入到计算机上），设备驱动程序可以检查到这种设备状态的变化,通过某种机制使得用户空间找到到该设备对应的驱动程序模块并加载之。linux上有两种机制实现热插拔：udev和/sbin/hotplug。早期是使用后者来实现的，但是随着内核的发展演进，udev逐渐取代/sbin/hotplug。我们着重学习udev。开发板使用的是针对嵌入式系统的简化版udev--mdev。udev的实现是基于内核中的网络机制，他通过创建标准的socket接口来监听来自内核的网络广播包，并对接收到的包进行分析处理。其实这两种机制都是必须依赖内核空间的支持才可以工作。（简单的说就，热插拔就是在内核在设备发生变化的时候汇报给用户空间，然后用户空间的udev根据收到的状态进行操作，如驱动的加载和卸载）
我们具体来看udev，我们这里就着重的研究内核中为热插拔做的工作，刚说到内核要做的就是将设备的状态发送给用户空间（在变化的时候）。比较核心的接口就是kobject_uevent。而kobject_uevent就是调用kobject_uevent_env函数完成。那我们就来看看kobject_uevent_env。这个函数的主要工作就是把一些环境变量和“kobject_action”发送给用户空间。

```
enum kobject_action {
    KOBJ_ADD,
    KOBJ_REMOVE,
    KOBJ_CHANGE,
    KOBJ_MOVE,
    KOBJ_OFFLINE,
    KOBJ_MAX
};
```

具体的请参考内核的kobject_uevent_env函数的实现。

4、总线、设备与驱动

按照设备模型的思想，设备驱动由总线、设备和驱动这三大组件构成。这样做的优势是方便管理和移植。我们常见的一些子系统例如platform，i2c都是典型的按照设备模型的标准来构建的。

总线及其注册

总线是linux设备驱动模型的核心框架，对设备和驱动的管理是通过总线。设备模型中的总线，即可以是实际物理总线（比如pci总线和i2c总线），也可以是虚拟出来的“平台”总线（比如platform）。符合linux设备模型的设备和驱动是必须挂载到一根总线上，无论它是实际存在的总线还是虚拟总线。具体总线又是怎么管理我们的设备和驱动的呢？那我们就需要对关于总线的结构体来分析--struct bus_type。

```
struct bus_type {
    const char      *name;
    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 bus_type_private *p;
};
```

name 总线的名称
bus_attrs 总线属性，包括操作这些属性的一组函数，都包含在struct bus_attribute 结构体中。
dev_attrs 挂载到该总线上的设备的属性，逻辑和总线属性相同。
drv_attrs 挂载到该总线上的驱动的属性，逻辑和总线属性相同。
match 总线用来实现挂载到他上面的驱动和设备之间的匹配操作。其他的操作函数就不一一介绍了。
pm 电源管理相关接口集
p 一个用来管理总线上的设备和驱动的结构体，定义如下：

```
struct bus_type_private {
    struct kset subsys;
    struct kset *drivers_kset;
    struct kset *devices_kset;
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type *bus;
};
```

subsys 内核中所有的bus内核对象的集合。
drivers_kset 该总线上所挂载的驱动的集合。
devices_kset 该总线上所挂载的设备的集合。
klist_devices和klist_drivers 表示该bus上所有设备和驱动的链表。
bus_notifier 内核通知链表。
drivers_autoprobe 表示当向总线上注册某一设备或驱动时，是否进行设备和驱动的绑定。
bus 指向包含这个结构体的bus_type。

总线要先于设备和驱动的注册，在内核启动时，在驱动相关的初始化中调用了buses_init函数进行初始化，起就是就调用kset_create_and_add函数在sysfs根目录下创建bus目录，该函数所在文件的路径/driver/base/bus.c。

总线相关接口：
//向系统注册一条总线
int bus_register(struct bus_type *bus);
该函数主要做了：
1，分配 bus的private成员所指向的空间，并完成成员填充
2，在该bus所在的目录下创建devices和drivers目录（kset_create_and_add），
3，创建设备和驱动与匹配相关的属性文件和总线基础的属性文件。

设备和驱动的绑定
当一个驱动或者设备注册到总线上时，就会尝试和其他的设备或者驱动进行绑定（绑定成不成功就不一定了）。当调用device_register函数向某一总线注册设备时，device_bind_driver函数就会被调用,来将该设备与它的驱动程序绑定起来。

```
int device_bind_driver(struct device *dev)
{
    int ret;

    ret = driver_sysfs_add(dev);
    if (!ret)
        driver_bound(dev);
    return ret;
}
```

设备
设备在内核中的数据结构为struct device,该类型的实例就是对具体设备的一个抽象。

```
struct device {
    struct device      *parent;

    struct device_private   *p;

    struct kobject kobj;
    const char      *init_name; /* initial name of the device */
    struct device_type *type;

    struct mutex        mutex;  /* mutex 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        *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;
#ifdef CONFIG_OF
    struct device_node  *of_node;
#endif

    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;
    const struct attribute_group **groups; /* optional groups */

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

parent　当前设备的父设备
p　　指向该设备的驱动相关的数据
kobj　代表struct device的内核对象
init_name　设备对象的名称。将来该设备加入到总线上是，内核会把这个名称设置成kobj的名称，具体表现在sysfs中的目录名
bus　　该设备所在的总线的对象指针
driver　用来表示该设备是否已经和他的driver进行了绑定，如果该值为NULL，说明还没有找到该设备对应的驱动

```
struct device_driver {
    const char      *name;
    struct bus_type     *bus;

    struct module       *owner;
    const char      *mod_name;  /* used for built-in modules */

    bool suppress_bind_attrs;  /* disables bind/unbind via sysfs */

#if defined(CONFIG_OF)
    const struct of_device_id   *of_match_table;
#endif

    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 attribute_group **groups;

    const struct dev_pm_ops *pm;

    struct driver_private *p;
} ? end device_driver ? ;
```

platform_device_register()->platform_device_add()->device_add()->bus_probe_device()->device_attach()->bus_for_each_drv()->__device_attach()->driver_match_device()->driver_probe_device()->really_probe()->driver_bound();

platform_driver_register()->driver_register()->bus_add_driver()->driver_attach()->bus_for_each_dev()->__driver_attach()->driver_match_device()->driver_probe_device()->really_probe()->driver_bound();