Linux内核部件分析

设备驱动模型之device

[日期：2011-10-06] 来源：Linux社区 作者：qb_2008 [字体：大 中 小]

linux的设备驱动模型，是建立在sysfs和kobject之上的，由总线、设备、驱动、类所组成的关系结构。从本节开始，我们将对linux这一设备驱动模型进行深入分析。

头文件是include/linux/device.h，实现在drivers/base目录中。本节要分析的，是其中的设备，主要在core.c中。

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

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

先来分析下struct device的结构变量。首先是指向父节点的指针parent，kobj是内嵌在device中的kobject，用于把它联系到sysfs中。bus是对设备所在总线的指针，driver是对设备所用驱动的指针。还有DMA需要的数据，表示设备号的devt，表示设备资源的devres_head和保护它的devres_lock。指向类的指针class，knode_class是被连入class链表时所用的klist节点。group是设备的属性集合。release应该是设备释放时调用的函数。

```
struct device_private {
    struct klist klist_children;
    struct klist_node knode_parent;
    struct klist_node knode_driver;
    struct klist_node knode_bus;
    void *driver_data;
    struct device *device;
};
#define to_device_private_parent(obj)   \
    container_of(obj, struct device_private, knode_parent)
#define to_device_private_driver(obj)   \
    container_of(obj, struct device_private, knode_driver)
#define to_device_private_bus(obj)  \
    container_of(obj, struct device_private, knode_bus)
```

struct device中有一部分不愿意让外界看到，所以做出struct device_private结构，包括了设备驱动模型内部的链接。klist_children是子设备的链表，knode_parent是连入父设备的klist_children时所用的节点，knode_driver是连入驱动的设备链表所用的节点，knode_bus是连入总线的设备链表时所用的节点。driver_data用于在设备结构中存放相关的驱动信息，也许是驱动专门为设备建立的结构实例。device则是指向struct device_private所属的device。

下面还有一些宏，to_device_private_parent()是从父设备的klist_children上节点，获得相应的device_private。to_device_private_driver()是从驱动的设备链表上节点，获得对应的device_private。to_device_private_bus()是从总线的设备链表上节点，获得对应的device_private。

或许会奇怪，为什么knode_class没有被移入struct device_private，或许有外部模块需要用到它。

```
/*
 * The type of device, "struct device" is embedded in. A class
 * or bus can contain devices of different types
 * like "partitions" and "disks", "mouse" and "event".
 * This identifies the device type and carries type-specific
 * information, equivalent to the kobj_type of a kobject.
 * If "name" is specified, the uevent will contain it in
 * the DEVTYPE variable.
 */
struct device_type {
    const char *name;
    const struct attribute_group **groups;
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
    char *(*devnode)(struct device *dev, mode_t *mode);
    void (*release)(struct device *dev);

    const struct dev_pm_ops *pm;
};
```

device竟然有device_type，类似于与kobject相对的kobj_type，之后我们再看它怎么用。

```
/* interface for exporting device attributes */
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);
};

#define DEVICE_ATTR(_name, _mode, _show, _store) \
struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
```

这个device_attribute显然就是device对struct attribute的封装，新加的show()、store()函数都是以与设备相关的结构调用的。

至于device中其它的archdata、dma、devres，都是作为设备特有的，我们现在主要关心设备驱动模型的建立，这些会尽量忽略。

下面就来看看device的实现，这主要在core.c中。

```
int __init devices_init(void)
{
    devices_kset = kset_create_and_add("devices", &device_uevent_ops, NULL);
    if (!devices_kset)
        return -ENOMEM;
    dev_kobj = kobject_create_and_add("dev", NULL);
    if (!dev_kobj)
```

```
        goto dev_kobj_err;
    sysfs_dev_block_kobj = kobject_create_and_add("block", dev_kobj);
    if (!sysfs_dev_block_kobj)
        goto block_kobj_err;
    sysfs_dev_char_kobj = kobject_create_and_add("char", dev_kobj);
    if (!sysfs_dev_char_kobj)
        goto char_kobj_err;

    return 0;

 char_kobj_err:
    kobject_put(sysfs_dev_block_kobj);
 block_kobj_err:
    kobject_put(dev_kobj);
 dev_kobj_err:
    kset_unregister(devices_kset);
    return -ENOMEM;
}
```

这是在设备驱动模型初始化时调用的device部分初始的函数devices_init()。它干的事情我们都很熟悉，就是建立sysfs中的devices目录，和dev目录。还在dev目录下又建立了block和char两个子目录。因为dev目录只打算存放辅助的设备号，所以没必要使用kset。

```
static ssize_t dev_attr_show(struct kobject *kobj, struct attribute *attr,
                 char *buf)
{
    struct device_attribute *dev_attr = to_dev_attr(attr);
    struct device *dev = to_dev(kobj);
    ssize_t ret = -EIO;

    if (dev_attr->show)
        ret = dev_attr->show(dev, dev_attr, buf);
    if (ret >= (ssize_t)PAGE_SIZE) {
        print_symbol("dev_attr_show: %s returned bad count\n",
                (unsigned long)dev_attr->show);
    }
    return ret;
}

static ssize_t dev_attr_store(struct kobject *kobj, struct attribute *attr,
                  const char *buf, size_t count)
{
    struct device_attribute *dev_attr = to_dev_attr(attr);
    struct device *dev = to_dev(kobj);
    ssize_t ret = -EIO;

    if (dev_attr->store)
        ret = dev_attr->store(dev, dev_attr, buf, count);
    return ret;
}

static struct sysfs_ops dev_sysfs_ops = {
    .show   = dev_attr_show,
    .store  = dev_attr_store,
};
```

看到这里是不是很熟悉，dev_sysfs_ops就是device准备注册到sysfs中的操作函数。dev_attr_show()和dev_attr_store()都会再调用与属性相关的函数。

```
static void device_release(struct kobject *kobj)
{
    struct device *dev = to_dev(kobj);
    struct device_private *p = dev->p;

    if (dev->release)
        dev->release(dev);
    else if (dev->type && dev->type->release)
        dev->type->release(dev);
    else if (dev->class && dev->class->dev_release)
        dev->class->dev_release(dev);
    else
        WARN(1, KERN_ERR "Device '%s' does not have a release() "
            "function, it is broken and must be fixed.\n",
```

```
        dev_name(dev));
    kfree(p);
}

static struct kobj_type device_ktype = {
    .release    = device_release,
    .sysfs_ops  = &dev_sysfs_ops,
};
```

使用的release函数是device_release。在释放device时，会依次调用device结构中定义的release函数，device_type中定义的release函数，device所属的class中所定义的release函数，最后会吧device_private结构释放掉。

```
static int dev_uevent_filter(struct kset *kset, struct kobject *kobj)
{
    struct kobj_type *ktype = get_ktype(kobj);

    if (ktype == &device_ktype) {
        struct device *dev = to_dev(kobj);
        if (dev->bus)
            return 1;
        if (dev->class)
            return 1;
    }
    return 0;
}

static const char *dev_uevent_name(struct kset *kset, struct kobject *kobj)
{
    struct device *dev = to_dev(kobj);

    if (dev->bus)
        return dev->bus->name;
    if (dev->class)
        return dev->class->name;
    return NULL;
}

static int dev_uevent(struct kset *kset, struct kobject *kobj,
             struct kobj_uevent_env *env)
{
    struct device *dev = to_dev(kobj);
    int retval = 0;

    /* add device node properties if present */
    if (MAJOR(dev->devt)) {
        const char *tmp;
        const char *name;
        mode_t mode = 0;

        add_uevent_var(env, "MAJOR=%u", MAJOR(dev->devt));
        add_uevent_var(env, "MINOR=%u", MINOR(dev->devt));
        name = device_get_devnode(dev, &mode, &tmp);
        if (name) {
            add_uevent_var(env, "DEVNAME=%s", name);
            kfree(tmp);
            if (mode)
                add_uevent_var(env, "DEVMODE=%#o", mode & 0777);
        }
    }

    if (dev->type && dev->type->name)
        add_uevent_var(env, "DEVTYPE=%s", dev->type->name);

    if (dev->driver)
        add_uevent_var(env, "DRIVER=%s", dev->driver->name);

#ifdef CONFIG_SYSFS_DEPRECATED
    if (dev->class) {
        struct device *parent = dev->parent;

```

```
        /* find first bus device in parent chain */
        while (parent && !parent->bus)
            parent = parent->parent;
        if (parent && parent->bus) {
            const char *path;

            path = kobject_get_path(&parent->kobj, GFP_KERNEL);
            if (path) {
                add_uevent_var(env, "PHYSDEVPATH=%s", path);
                kfree(path);
            }

            add_uevent_var(env, "PHYSDEVBUS=%s", parent->bus->name);

            if (parent->driver)
                add_uevent_var(env, "PHYSDEVDRIVER=%s",
                           parent->driver->name);
        }
    } else if (dev->bus) {
        add_uevent_var(env, "PHYSDEVBUS=%s", dev->bus->name);

        if (dev->driver)
            add_uevent_var(env, "PHYSDEVDRIVER=%s",
                       dev->driver->name);
    }
#endif

    /* have the bus specific function add its stuff */
    if (dev->bus && dev->bus->uevent) {
        retval = dev->bus->uevent(dev, env);
        if (retval)
            pr_debug("device: '%s': %s: bus uevent() returned %d\n",
                 dev_name(dev), __func__, retval);
    }

    /* have the class specific function add its stuff */
    if (dev->class && dev->class->dev_uevent) {
        retval = dev->class->dev_uevent(dev, env);
        if (retval)
            pr_debug("device: '%s': %s: class uevent() "
                 "returned %d\n", dev_name(dev),
                 __func__, retval);
    }

    /* have the device type specific fuction add its stuff */
    if (dev->type && dev->type->uevent) {
        retval = dev->type->uevent(dev, env);
        if (retval)
            pr_debug("device: '%s': %s: dev_type uevent() "
                 "returned %d\n", dev_name(dev),
                 __func__, retval);
    }

    return retval;
}

static struct kset_uevent_ops device_uevent_ops = {
    .filter =   dev_uevent_filter,
    .name =     dev_uevent_name,
    .uevent =   dev_uevent,
};
```

前面在讲到kset时，我们并未关注其中的kset_event_ops结构变量。但这里device既然用到了，我们就对其中的三个函数做简单介绍。kset_uevent_ops中的函数是用于管理kset内部kobject的uevent操作。其中filter函数用于阻止一个kobject向用户空间发送uevent，返回值为0表示阻止。这里dev_uevent_filter()检查device所属的bus或者class是否存在，如果都不存在，也就没有发送uevent的必要了。name函数是用于覆盖kset发送给用户空间的名称。这里dev_uevent_name()选择使用bus或者class的名称。uevent()函数是在uevent将被发送到用户空间之前调用的，用于向uevent中增加新的环境变量。dev_uevent()的实现很热闹，向uevent中添加了各种环境变量。

```
static ssize_t show_uevent(struct device *dev, struct device_attribute *attr,
               char *buf)
```

```
{
    struct kobject *top_kobj;
    struct kset *kset;
    struct kobj_uevent_env *env = NULL;
    int i;
    size_t count = 0;
    int retval;

    /* search the kset, the device belongs to */
    top_kobj = &dev->kobj;
    while (!top_kobj->kset && top_kobj->parent)
        top_kobj = top_kobj->parent;
    if (!top_kobj->kset)
        goto out;

    kset = top_kobj->kset;
    if (!kset->uevent_ops || !kset->uevent_ops->uevent)
        goto out;

    /* respect filter */
    if (kset->uevent_ops && kset->uevent_ops->filter)
        if (!kset->uevent_ops->filter(kset, &dev->kobj))
            goto out;

    env = kzalloc(sizeof(struct kobj_uevent_env), GFP_KERNEL);
    if (!env)
        return -ENOMEM;

    /* let the kset specific function add its keys */
    retval = kset->uevent_ops->uevent(kset, &dev->kobj, env);
    if (retval)
        goto out;

    /* copy keys to file */
    for (i = 0; i < env->envp_idx; i++)
        count += sprintf(&buf[count], "%s\n", env->envp[i]);
out:
    kfree(env);
    return count;
}

static ssize_t store_uevent(struct device *dev, struct device_attribute *attr,
            const char *buf, size_t count)
{
    enum kobject_action action;

    if (kobject_action_type(buf, count, &action) == 0) {
        kobject_uevent(&dev->kobj, action);
        goto out;
    }

    dev_err(dev, "uevent: unsupported action-string; this will "
          "be ignored in a future kernel version\n");
    kobject_uevent(&dev->kobj, KOBJ_ADD);
out:
    return count;
}

static struct device_attribute uevent_attr =
    __ATTR(uevent, S_IRUGO | S_IWUSR, show_uevent, store_uevent);
```

device不仅在kset中添加了对uevent的管理，而且还把uevent信息做成设备的一个属性uevent。其中show_event()是显示uevent中环境变量的，store_uevent()是发送uevent的。

```
static int device_add_attributes(struct device *dev,
            struct device_attribute *attrs)
{
    int error = 0;
    int i;

```

```
    if (attrs) {
        for (i = 0; attr_name(attrs[i]); i++) {
            error = device_create_file(dev, &attrs[i]);
            if (error)
                break;
        }
        if (error)
            while (--i >= 0)
                device_remove_file(dev, &attrs[i]);
    }
    return error;
}

static void device_remove_attributes(struct device *dev,
                     struct device_attribute *attrs)
{
    int i;

    if (attrs)
        for (i = 0; attr_name(attrs[i]); i++)
            device_remove_file(dev, &attrs[i]);
}

static int device_add_groups(struct device *dev,
                 const struct attribute_group **groups)
{
    int error = 0;
    int i;

    if (groups) {
        for (i = 0; groups[i]; i++) {
            error = sysfs_create_group(&dev->kobj, groups[i]);
            if (error) {
                while (--i >= 0)
                    sysfs_remove_group(&dev->kobj,
                               groups[i]);
                break;
            }
        }
    }
    return error;
}

static void device_remove_groups(struct device *dev,
                 const struct attribute_group **groups)
{
    int i;

    if (groups)
        for (i = 0; groups[i]; i++)
            sysfs_remove_group(&dev->kobj, groups[i]);
}
```

以上四个内部函数是用来向device中添加或删除属性与属性集合的。

device_add_attributes、device_remove_attributes、device_add_groups、device_remove_groups，都是直接通过sysfs提供的API实现。

```
static int device_add_attrs(struct device *dev)
{
    struct class *class = dev->class;
    struct device_type *type = dev->type;
    int error;

    if (class) {
        error = device_add_attributes(dev, class->dev_attrs);
        if (error)
            return error;
    }

    if (type) {
        error = device_add_groups(dev, type->groups);
```

```
        if (error)
            goto err_remove_class_attrs;
    }

    error = device_add_groups(dev, dev->groups);
    if (error)
        goto err_remove_type_groups;

    return 0;

 err_remove_type_groups:
    if (type)
        device_remove_groups(dev, type->groups);
 err_remove_class_attrs:
    if (class)
        device_remove_attributes(dev, class->dev_attrs);

    return error;
}

static void device_remove_attrs(struct device *dev)
{
    struct class *class = dev->class;
    struct device_type *type = dev->type;

    device_remove_groups(dev, dev->groups);

    if (type)
        device_remove_groups(dev, type->groups);

    if (class)
        device_remove_attributes(dev, class->dev_attrs);
}
```

device_add_attrs()实际负责device中的属性添加。也是几个部分的集合，包括class中的dev_attrs，device_type中的groups，还有device本身的groups。

device_remove_attrs()则负责对应的device属性删除工作。

```
#define print_dev_t(buffer, dev)                    \
    sprintf((buffer), "%u:%u\n", MAJOR(dev), MINOR(dev))

static ssize_t show_dev(struct device *dev, struct device_attribute *attr,
            char *buf)
{
    return print_dev_t(buf, dev->devt);
}

static struct device_attribute devt_attr =
    __ATTR(dev, S_IRUGO, show_dev, NULL);
```

这里又定义了一个名为dev的属性，就是显示设备的设备号。

```
/**
 * device_create_file - create sysfs attribute file for device.
 * @dev: device.
 * @attr: device attribute descriptor.
 */
int device_create_file(struct device *dev, struct device_attribute *attr)
{
    int error = 0;
    if (dev)
        error = sysfs_create_file(&dev->kobj, &attr->attr);
    return error;
}

/**
 * device_remove_file - remove sysfs attribute file.
 * @dev: device.
 * @attr: device attribute descriptor.
 */
void device_remove_file(struct device *dev, struct device_attribute *attr)
```

```
{
    if (dev)
        sysfs_remove_file(&dev->kobj, &attr->attr);
}

/**
 * device_create_bin_file - create sysfs binary attribute file for device.
 * @dev: device.
 * @attr: device binary attribute descriptor.
 */
int device_create_bin_file(struct device *dev, struct bin_attribute *attr)
{
    int error = -EINVAL;
    if (dev)
        error = sysfs_create_bin_file(&dev->kobj, attr);
    return error;
}

/**
 * device_remove_bin_file - remove sysfs binary attribute file
 * @dev: device.
 * @attr: device binary attribute descriptor.
 */
void device_remove_bin_file(struct device *dev, struct bin_attribute *attr)
{
    if (dev)
        sysfs_remove_bin_file(&dev->kobj, attr);
}

int device_schedule_callback_owner(struct device *dev,
        void (*func)(struct device *), struct module *owner)
{
    return sysfs_schedule_callback(&dev->kobj,
            (void (*)(void *)) func, dev, owner);
}
```

这里的五个函数，也是对sysfs提供的API的简单封装。

device_create_file()和device_remove_file()提供直接的属性文件管理方法。

device_create_bin_file()和device_remove_bin_file()则是提供设备管理二进制文件的方法。

device_schedule_callback_owner()也是简单地将func加入工作队列。

```
static void klist_children_get(struct klist_node *n)
{
    struct device_private *p = to_device_private_parent(n);
    struct device *dev = p->device;

    get_device(dev);
}

static void klist_children_put(struct klist_node *n)
{
    struct device_private *p = to_device_private_parent(n);
    struct device *dev = p->device;

    put_device(dev);
}
```

如果之前认真看过klist的实现，应该知道，klist_children_get()和klist_children_put()就是在设备挂入和删除父设备的klist_children链表时调用的函数。在父设备klist_children链表上的指针，相当于对device的一个引用计数。

```
struct device *get_device(struct device *dev)
{
    return dev ? to_dev(kobject_get(&dev->kobj)) : NULL;
}

/**
 * put_device - decrement reference count.
 * @dev: device in question.
 */
```

```
void put_device(struct device *dev)
{
    /* might_sleep(); */
    if (dev)
        kobject_put(&dev->kobj);
}
```

device中的引用计数，完全交给内嵌的kobject来做。如果引用计数降为零，自然是调用之前说到的包含甚广的device_release函数。

```
void device_initialize(struct device *dev)
{
    dev->kobj.kset = devices_kset;
    kobject_init(&dev->kobj, &device_ktype);
    INIT_LIST_HEAD(&dev->dma_pools);
    init_MUTEX(&dev->sem);
    spin_lock_init(&dev->devres_lock);
    INIT_LIST_HEAD(&dev->devres_head);
    device_init_wakeup(dev, 0);
    device_pm_init(dev);
    set_dev_node(dev, -1);
}
```

device_initialize()就是device结构的初始化函数，它把device中能初始化的部分全初始化。它的界限在其中kobj的位置与device在设备驱动模型中的位置，这些必须由外部设置。可以看到，调用kobject_init()时，object的kobj_type选择了device_ktype，其中主要是sysops的两个函数，还有device_release函数。

```
static struct kobject *virtual_device_parent(struct device *dev)
{
    static struct kobject *virtual_dir = NULL;

    if (!virtual_dir)
        virtual_dir = kobject_create_and_add("virtual",
                             &devices_kset->kobj);

    return virtual_dir;
}

static struct kobject *get_device_parent(struct device *dev,
                 struct device *parent)
{
    int retval;

    if (dev->class) {
        struct kobject *kobj = NULL;
        struct kobject *parent_kobj;
        struct kobject *k;

        /*
         * If we have no parent, we live in "virtual".
         * Class-devices with a non class-device as parent, live
         * in a "glue" directory to prevent namespace collisions.
         */
        if (parent == NULL)
            parent_kobj = virtual_device_parent(dev);
        else if (parent->class)
            return &parent->kobj;
        else
            parent_kobj = &parent->kobj;

        /* find our class-directory at the parent and reference it */
        spin_lock(&dev->class->p->class_dirs.list_lock);
        list_for_each_entry(k, &dev->class->p->class_dirs.list, entry)
            if (k->parent == parent_kobj) {
                kobj = kobject_get(k);
                break;
            }
        spin_unlock(&dev->class->p->class_dirs.list_lock);
        if (kobj)
            return kobj;

        /* or create a new class-directory at the parent device */
        k = kobject_create();
```

```
        if (!k)
            return NULL;
        k->kset = &dev->class->p->class_dirs;
        retval = kobject_add(k, parent_kobj, "%s", dev->class->name);
        if (retval < 0) {
            kobject_put(k);
            return NULL;
        }
        /* do not emit an uevent for this simple "glue" directory */
        return k;
    }

    if (parent)
        return &parent->kobj;
    return NULL;
}
```

这里的get_device_parent()就是获取父节点的kobject，但也并非就如此简单。get_device_parent()的返回值直接决定了device将被挂在哪个目录下。到底该挂在哪，是由dev->class、dev->parent、dev->parent->class等因素综合决定的。我们看get_device_parent()中是如何判断的。如果dev->class为空，表示一切随父设备，有parent则返回parent->kobj，没有则返回NULL。如果有dev->class呢，情况就比较复杂了，也许device有着与parent不同的class，也许device还没有一个parent，等等。我们看具体的情况。如果parent不为空，而且存在parent->class，则还放在parent目录下。不然，要么parent不存在，要么parent没有class，很难直接将有class的device放在parent下面。目前的解决方法很简单，在parent与device之间，再加一层表示class的目录。如果parent都没有，那就把/sys/devices/virtual当做parent。class->p->class_dirs就是专门存放这种中间kobject的kset。思路理清后，再结合实际的sysfs，代码就很容易看懂了。

```
static void cleanup_glue_dir(struct device *dev, struct kobject *glue_dir)
{
    /* see if we live in a "glue" directory */
    if (!glue_dir || !dev->class ||
        glue_dir->kset != &dev->class->p->class_dirs)
        return;

    kobject_put(glue_dir);
}

static void cleanup_device_parent(struct device *dev)
{
    cleanup_glue_dir(dev, dev->kobj.parent);
}
```

cleanup_device_parent()是取消对parent引用时调用的函数，看起来只针对这种glue形式的目录起作用。

```
static void setup_parent(struct device *dev, struct device *parent)
{
    struct kobject *kobj;
    kobj = get_device_parent(dev, parent);
    if (kobj)
        dev->kobj.parent = kobj;
}
```

setup_parent()就是调用get_device_parent()获得应该存放的父目录kobj，并把dev->kobj.parent设为它。

```
static int device_add_class_symlinks(struct device *dev)
{
    int error;

    if (!dev->class)
        return 0;

    error = sysfs_create_link(&dev->kobj,
                  &dev->class->p->class_subsys.kobj,
                  "subsystem");
    if (error)
        goto out;
    /* link in the class directory pointing to the device */
    error = sysfs_create_link(&dev->class->p->class_subsys.kobj,
                  &dev->kobj, dev_name(dev));
    if (error)
        goto out_subsys;

    if (dev->parent && device_is_not_partition(dev)) {
        error = sysfs_create_link(&dev->kobj, &dev->parent->kobj,
                      "device");
```

```
        if (error)
            goto out_busid;
    }
    return 0;

out_busid:
    sysfs_remove_link(&dev->class->p->class_subsys.kobj, dev_name(dev));
out_subsys:
    sysfs_remove_link(&dev->kobj, "subsystem");
out:
    return error;
}
```

device_add_class_symlinks()在device和class直接添加一些软链接。在device目录下创建指向class的subsystem文件，在class目录下创建指向device的同名文件。如果device有父设备，而且device不是块设备分区，则在device目录下建立一个指向父设备的device链接文件。这一点在usb设备和usb接口间很常见。

```
static void device_remove_class_symlinks(struct device *dev)
{
    if (!dev->class)
        return;

#ifdef CONFIG_SYSFS_DEPRECATED
    if (dev->parent && device_is_not_partition(dev)) {
        char *class_name;

        class_name = make_class_name(dev->class->name, &dev->kobj);
        if (class_name) {
            sysfs_remove_link(&dev->parent->kobj, class_name);
            kfree(class_name);
        }
        sysfs_remove_link(&dev->kobj, "device");
    }

    if (dev->kobj.parent != &dev->class->p->class_subsys.kobj &&
        device_is_not_partition(dev))
        sysfs_remove_link(&dev->class->p->class_subsys.kobj,
                  dev_name(dev));
#else
    if (dev->parent && device_is_not_partition(dev))
        sysfs_remove_link(&dev->kobj, "device");

    sysfs_remove_link(&dev->class->p->class_subsys.kobj, dev_name(dev));
#endif

    sysfs_remove_link(&dev->kobj, "subsystem");
}
```

device_remove_class_symlinks()删除device和class之间的软链接。

```
static inline const char *dev_name(const struct device *dev)
{
    return kobject_name(&dev->kobj);
}

int dev_set_name(struct device *dev, const char *fmt, ...)
{
    va_list vargs;
    int err;

    va_start(vargs, fmt);
    err = kobject_set_name_vargs(&dev->kobj, fmt, vargs);
    va_end(vargs);
    return err;
}
```

dev_name()获得设备名称，dev_set_name()设置设备名称。但这里的dev_set_name()只能在设备未注册前使用。device的名称其实是完全靠dev->kobj管理的。

```
static struct kobject *device_to_dev_kobj(struct device *dev)
{
    struct kobject *kobj;

    if (dev->class)
```

```
        kobj = dev->class->dev_kobj;
    else
        kobj = sysfs_dev_char_kobj;

    return kobj;
}
```

device_to_dev_kobj()为dev选择合适的/sys/dev下的kobject，或者是块设备，或者是字符设备，或者没有。

```
#define format_dev_t(buffer, dev)                   \
    ({                          \
        sprintf(buffer, "%u:%u", MAJOR(dev), MINOR(dev));   \
        buffer;                         \
    })

static int device_create_sys_dev_entry(struct device *dev)
{
    struct kobject *kobj = device_to_dev_kobj(dev);
    int error = 0;
    char devt_str[15];

    if (kobj) {
        format_dev_t(devt_str, dev->devt);
        error = sysfs_create_link(kobj, &dev->kobj, devt_str);
    }

    return error;
}

static void device_remove_sys_dev_entry(struct device *dev)
{
    struct kobject *kobj = device_to_dev_kobj(dev);
    char devt_str[15];

    if (kobj) {
        format_dev_t(devt_str, dev->devt);
        sysfs_remove_link(kobj, devt_str);
    }
}
```

device_create_sys_dev_entry()是在/sys/dev相应的目录下建立对设备的软链接。先是通过device_to_dev_kobj()获得父节点的kobj，然后调用sysfs_create_link()建立软链接。

device_remove_sys_dev_entry()与其操作正相反，删除在/sys/dev下建立的软链接。

```
int device_private_init(struct device *dev)
{
    dev->p = kzalloc(sizeof(*dev->p), GFP_KERNEL);
    if (!dev->p)
        return -ENOMEM;
    dev->p->device = dev;
    klist_init(&dev->p->klist_children, klist_children_get,
           klist_children_put);
    return 0;
}
```

device_private_init()分配并初始化dev->p。至于空间的释放，是等到释放设备时调用的device_release()中。

之前的函数比较散乱，或许找不出一个整体的印象。但下面马上就要看到重要的部分了，因为代码终于攒到了爆发的程度！

```
/**
 * device_register - register a device with the system.
 * @dev: pointer to the device structure
 *
 * This happens in two clean steps - initialize the device
 * and add it to the system. The two steps can be called
 * separately, but this is the easiest and most common.
 * I.e. you should only call the two helpers separately if
 * have a clearly defined need to use and refcount the device
 * before it is added to the hierarchy.
 *
 * NOTE: _Never_ directly free @dev after calling this function, even
 * if it returned an error! Always use put_device() to give up the
```

```
 * reference initialized in this function instead.
 */
int device_register(struct device *dev)
{
    device_initialize(dev);
    return device_add(dev);
}
```

device_register()是提供给外界注册设备的接口。它先是调用device_initialize()初始化dev结构，然后调用device_add()将其加入系统中。但要注意，在调用device_register()注册dev之前，有一些dev结构变量是需要自行设置的。这其中有指明设备位置的struct device *parent,struct bus_type *bus, struct class *class，有指明设备属性的 const char *init_name, struct device_type *type, const struct attribute_group **groups, void (*release)(struct device *dev), dev_t devt，等等。不同设备的使用方法不同，我们留待之后再具体分析。device_initialize()我们已经看过，下面重点看看device_add()是如何实现的。

```
int device_add(struct device *dev)
{
    struct device *parent = NULL;
    struct class_interface *class_intf;
    int error = -EINVAL;

    dev = get_device(dev);
    if (!dev)
        goto done;

    if (!dev->p) {
        error = device_private_init(dev);
        if (error)
            goto done;
    }

    /*
     * for statically allocated devices, which should all be converted
     * some day, we need to initialize the name. We prevent reading back
     * the name, and force the use of dev_name()
     */
    if (dev->init_name) {
        dev_set_name(dev, "%s", dev->init_name);
        dev->init_name = NULL;
    }

    if (!dev_name(dev))
        goto name_error;

    pr_debug("device: '%s': %s\n", dev_name(dev), __func__);

    parent = get_device(dev->parent);
    setup_parent(dev, parent);

    /* use parent numa_node */
    if (parent)
        set_dev_node(dev, dev_to_node(parent));

    /* first, register with generic layer. */
    /* we require the name to be set before, and pass NULL */
    error = kobject_add(&dev->kobj, dev->kobj.parent, NULL);
    if (error)
        goto Error;

    /* notify platform of device entry */
    if (platform_notify)
        platform_notify(dev);

    error = device_create_file(dev, &uevent_attr);
    if (error)
        goto attrError;

    if (MAJOR(dev->devt)) {
        error = device_create_file(dev, &devt_attr);
        if (error)
            goto ueventattrError;

```

```
        error = device_create_sys_dev_entry(dev);
        if (error)
            goto devtattrError;

        devtmpfs_create_node(dev);
    }

    error = device_add_class_symlinks(dev);
    if (error)
        goto SymlinkError;
    error = device_add_attrs(dev);
    if (error)
        goto AttrsError;
    error = bus_add_device(dev);
    if (error)
        goto BusError;
    error = dpm_sysfs_add(dev);
    if (error)
        goto DPMError;
    device_pm_add(dev);

    /* Notify clients of device addition.  This call must come
     * after dpm_sysf_add() and before kobject_uevent().
     */
    if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                         BUS_NOTIFY_ADD_DEVICE, dev);

    kobject_uevent(&dev->kobj, KOBJ_ADD);
    bus_probe_device(dev);
    if (parent)
        klist_add_tail(&dev->p->knode_parent,
                   &parent->p->klist_children);

    if (dev->class) {
        mutex_lock(&dev->class->p->class_mutex);
        /* tie the class to the device */
        klist_add_tail(&dev->knode_class,
                   &dev->class->p->class_devices);

        /* notify any interfaces that the device is here */
        list_for_each_entry(class_intf,
                    &dev->class->p->class_interfaces, node)
            if (class_intf->add_dev)
                class_intf->add_dev(dev, class_intf);
        mutex_unlock(&dev->class->p->class_mutex);
    }
done:
    put_device(dev);
    return error;
 DPMError:
    bus_remove_device(dev);
 BusError:
    device_remove_attrs(dev);
 AttrsError:
    device_remove_class_symlinks(dev);
 SymlinkError:
    if (MAJOR(dev->devt))
        device_remove_sys_dev_entry(dev);
 devtattrError:
    if (MAJOR(dev->devt))
        device_remove_file(dev, &devt_attr);
 ueventattrError:
    device_remove_file(dev, &uevent_attr);
 attrError:
    kobject_uevent(&dev->kobj, KOBJ_REMOVE);
    kobject_del(&dev->kobj);
 Error:
    cleanup_device_parent(dev);
    if (parent)
```

```
        put_device(parent);
name_error:
    kfree(dev->p);
    dev->p = NULL;
    goto done;
}
```

device_add()将dev加入设备驱动模型。它先是调用get_device(dev)增加dev的引用计数，然后调用device_private_init()分配和初始化dev->p，调用dev_set_name()设置dev名字。然后是准备将dev加入sysfs，先是用get_device(parent)增加对parent的引用计数(无论是直接挂在parent下还是通过一个类层挂在parent下都要增加parent的引用计数），然后调用setup_parent()找到实际要加入的父kobject，通过kobject_add()加入其下。然后是添加属性和属性集合的操作，调用device_create_file()添加uevent属性，调用device_add_attrs()添加device/type/class预定义的属性与属性集合。如果dev有被分配设备号，再用device_create_file()添加dev属性，并用device_create_sys_dev_entry()在/sys/dev下添加相应的软链接，最后调用devtmpfs_create_node()在/dev下创建相应的设备文件。然后调用device_add_class_symlinks()添加dev与class间的软链接，调用bus_add_device()添加dev与bus间的软链接，并将dev挂入bus的设备链表。调用dpm_sysfs_add()增加dev下的power属性集合，调用device_pm_add()将dev加入dpm_list链表。

调用kobject_uevent()发布KOBJ_ADD消息，调用bus_probe_device()为dev寻找合适的驱动。如果有parent节点，把dev->p->knode_parent挂入parent->p->klist_children链表。如果dev有所属的class，将dev->knode_class挂在class->p->class_devices上，并调用可能的类设备接口的add_dev()方法。可能对于直接在bus上的设备来说，自然可以调用bus_probe_device()查找驱动，而不与总线直接接触的设备，则要靠class来发现驱动，这里的class_interface中的add_dev()方法，就是一个绝好的机会。最后会调用put_device(dev)释放在函数开头增加的引用计数。

device_add()要做的事很多，但想想每件事都在情理之中。device是设备驱动模型的基本元素，在class、bus、dev、devices中都有它的身影。device_add()要适应各种类型的设备注册，自然会越来越复杂。可以说文件开头定义的内部函数，差不多都是为了这里服务的。

```
void device_unregister(struct device *dev)
{
    pr_debug("device: '%s': %s\n", dev_name(dev), __func__);
    device_del(dev);
    put_device(dev);
}
```

有注册自然又注销。device_unregister()就是用于将dev从系统中注销，并释放创建时产生的引用计数。

```
void device_del(struct device *dev)
{
    struct device *parent = dev->parent;
    struct class_interface *class_intf;

    /* Notify clients of device removal.  This call must come
     * before dpm_sysfs_remove().
     */
    if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                         BUS_NOTIFY_DEL_DEVICE, dev);
    device_pm_remove(dev);
    dpm_sysfs_remove(dev);
    if (parent)
        klist_del(&dev->p->knode_parent);
    if (MAJOR(dev->devt)) {
        devtmpfs_delete_node(dev);
        device_remove_sys_dev_entry(dev);
        device_remove_file(dev, &devt_attr);
    }
    if (dev->class) {
        device_remove_class_symlinks(dev);

        mutex_lock(&dev->class->p->class_mutex);
        /* notify any interfaces that the device is now gone */
        list_for_each_entry(class_intf,
                    &dev->class->p->class_interfaces, node)
            if (class_intf->remove_dev)
                class_intf->remove_dev(dev, class_intf);
        /* remove the device from the class list */
        klist_del(&dev->knode_class);
        mutex_unlock(&dev->class->p->class_mutex);
    }
    device_remove_file(dev, &uevent_attr);
    device_remove_attrs(dev);
    bus_remove_device(dev);

    /*
     * Some platform devices are driven without driver attached
     * and managed resources may have been acquired.  Make sure
```

```
     * all resources are released.
     */
    devres_release_all(dev);

    /* Notify the platform of the removal, in case they
     * need to do anything...
     */
    if (platform_notify_remove)
        platform_notify_remove(dev);
    kobject_uevent(&dev->kobj, KOBJ_REMOVE);
    cleanup_device_parent(dev);
    kobject_del(&dev->kobj);
    put_device(parent);
}
```

device_del()是与device_add()相对的函数，进行实际的将dev从系统中脱离的工作。这其中既有将dev从设备驱动模型各种链表中脱离的工作，又有将dev从sysfs的各个角落删除的工作。大致流程与dev_add()相对，就不一一介绍。

爆发结束，下面来看一些比较轻松的函数。

```
/**
 * device_get_devnode - path of device node file
 * @dev: device
 * @mode: returned file access mode
 * @tmp: possibly allocated string
 *
 * Return the relative path of a possible device node.
 * Non-default names may need to allocate a memory to compose
 * a name. This memory is returned in tmp and needs to be
 * freed by the caller.
 */
const char *device_get_devnode(struct device *dev,
                mode_t *mode, const char **tmp)
{
    char *s;

    *tmp = NULL;

    /* the device type may provide a specific name */
    if (dev->type && dev->type->devnode)
        *tmp = dev->type->devnode(dev, mode);
    if (*tmp)
        return *tmp;

    /* the class may provide a specific name */
    if (dev->class && dev->class->devnode)
        *tmp = dev->class->devnode(dev, mode);
    if (*tmp)
        return *tmp;

    /* return name without allocation, tmp == NULL */
    if (strchr(dev_name(dev), '!') == NULL)
        return dev_name(dev);

    /* replace '!' in the name with '/' */
    *tmp = kstrdup(dev_name(dev), GFP_KERNEL);
    if (!*tmp)
        return NULL;
    while ((s = strchr(*tmp, '!')))
        s[0] = '/';
    return *tmp;
}
```

device_get_devnode()返回设备的路径名。不过似乎可以由device_type或者class定义一些独特的返回名称。

```
static struct device *next_device(struct klist_iter *i)
{
    struct klist_node *n = klist_next(i);
    struct device *dev = NULL;
    struct device_private *p;

```

```
    if (n) {
        p = to_device_private_parent(n);
        dev = p->device;
    }
    return dev;
}

int device_for_each_child(struct device *parent, void *data,
             int (*fn)(struct device *dev, void *data))
{
    struct klist_iter i;
    struct device *child;
    int error = 0;

    if (!parent->p)
        return 0;

    klist_iter_init(&parent->p->klist_children, &i);
    while ((child = next_device(&i)) && !error)
        error = fn(child, data);
    klist_iter_exit(&i);
    return error;
}

struct device *device_find_child(struct device *parent, void *data,
                 int (*match)(struct device *dev, void *data))
{
    struct klist_iter i;
    struct device *child;

    if (!parent)
        return NULL;

    klist_iter_init(&parent->p->klist_children, &i);
    while ((child = next_device(&i)))
        if (match(child, data) && get_device(child))
            break;
    klist_iter_exit(&i);
    return child;
}
```

device_for_each_child()对dev下的每个子device，都调用一遍特定的处理函数。

device_find_child()则是查找dev下特点的子device，查找使用特定的match函数。

这两个遍历过程都使用了klist特有的遍历函数，支持遍历过程中的节点删除等功能。next_device()则是为了遍历方便封装的一个内部函数。

下面本该是root_device注册相关的代码。但经过检查，linux内核中使用到的root_device很少见，而且在sysfs中也未能找到一个实际的例子。所以root_device即使还未被弃用，也并非主流，我们将其跳过。

与kobject和kset类似，device也为我们提供了快速device创建方法，下面就看看吧。

```
static void device_create_release(struct device *dev)
{
    pr_debug("device: '%s': %s\n", dev_name(dev), __func__);
    kfree(dev);
}

struct device *device_create_vargs(struct class *class, struct device *parent,
                   dev_t devt, void *drvdata, const char *fmt,
                   va_list args)
{
    struct device *dev = NULL;
    int retval = -ENODEV;

    if (class == NULL || IS_ERR(class))
        goto error;

    dev = kzalloc(sizeof(*dev), GFP_KERNEL);
    if (!dev) {
        retval = -ENOMEM;
        goto error;
```

```
    }

    dev->devt = devt;
    dev->class = class;
    dev->parent = parent;
    dev->release = device_create_release;
    dev_set_drvdata(dev, drvdata);

    retval = kobject_set_name_vargs(&dev->kobj, fmt, args);
    if (retval)
        goto error;

    retval = device_register(dev);
    if (retval)
        goto error;

    return dev;

error:
    put_device(dev);
    return ERR_PTR(retval);
}

struct device *device_create(struct class *class, struct device *parent,
                 dev_t devt, void *drvdata, const char *fmt, ...)
{
    va_list vargs;
    struct device *dev;

    va_start(vargs, fmt);
    dev = device_create_vargs(class, parent, devt, drvdata, fmt, vargs);
    va_end(vargs);
    return dev;
}
```

这里的device_create()提供了一个快速的dev创建注册方法。只是中间没有提供设置device_type的方法，或许是这样的device已经够特立独行了，不需要搞出一类来。

```
static int __match_devt(struct device *dev, void *data)
{
    dev_t *devt = data;

    return dev->devt == *devt;
}

void device_destroy(struct class *class, dev_t devt)
{
    struct device *dev;

    dev = class_find_device(class, NULL, &devt, __match_devt);
    if (dev) {
        put_device(dev);
        device_unregister(dev);
    }
}
```

device_destroy()就是与device_create()相对的注销函数。至于这里为什么会多一个put_device(dev)，也很简单，因为在class_find_device()找到dev时，调用了get_device()。

```
struct device *class_find_device(struct class *class, struct device *start,
                 void *data,
                 int (*match)(struct device *, void *))
{
    struct class_dev_iter iter;
    struct device *dev;

    if (!class)
        return NULL;
    if (!class->p) {
        WARN(1, "%s called for class '%s' before it was initialized",
             __func__, class->name);
```

```
        return NULL;
    }

    class_dev_iter_init(&iter, class, start, NULL);
    while ((dev = class_dev_iter_next(&iter))) {
        if (match(dev, data)) {
            get_device(dev);
            break;
        }
    }
    class_dev_iter_exit(&iter);

    return dev;
}
```

class_find_device()本来是class.c中的内容，其实现也于之前将的遍历dev->p->klist_children类似，无非是在klist提供的遍历方法上加以封装。但我们这里列出class_find_device()的实现与使用它的device_destroy()，却是为了更好地分析这个调用流程中dev是如何被保护的。它实际上是经历了三个保护手段：首先在class_dev_iter_next()->klist_next()中，是受到struct klist中 spinlock_t k_lock保护的。在找到下一点并解锁之前，就增加了struct klist_node中的struct kref n_ref引用计数。在当前的next()调用完，到下一个next()调用之前，都是受这个增加的引用计数保护的。再看class_find_device()中，使用get_device(dev)增加了dev本身的引用计数保护(当然也要追溯到kobj->kref中)，这是第三种保护。知道device_destroy()中主动调用put_device(dev)才去除了这种保护。

本来对dev的保护，应该完全是由dev中的引用计数完成的。但实际上这种保护很多时候是间接完成的。例如这里的klist中的自旋锁，klist_node中的引用计数，都不过是为了保持class的设备链表中对dev的引用计数不消失，这是一种间接保护的手段，保证了这中间即使外界主动释放class设备链表对dev的引用计数，dev仍然不会被实际注销。这种曲折的联系，才真正发挥了引用计数的作用，构成设备驱动模型独特的魅力。

```
int device_rename(struct device *dev, char *new_name)
{
    char *old_device_name = NULL;
    int error;

    dev = get_device(dev);
    if (!dev)
        return -EINVAL;

    pr_debug("device: '%s': %s: renaming to '%s'\n", dev_name(dev),
          __func__, new_name);
    old_device_name = kstrdup(dev_name(dev), GFP_KERNEL);
    if (!old_device_name) {
        error = -ENOMEM;
        goto out;
    }

    error = kobject_rename(&dev->kobj, new_name);
    if (error)
        goto out;
    if (dev->class) {
        error = sysfs_create_link_nowarn(&dev->class->p->class_subsys.kobj,
                          &dev->kobj, dev_name(dev));
        if (error)
            goto out;
        sysfs_remove_link(&dev->class->p->class_subsys.kobj,
                  old_device_name);
    }
out:
    put_device(dev);

    kfree(old_device_name);

    return error;
}
```

device_rename()是供设备注册后改变名称用的，除了改变/sys/devices下地名称，还改变了/sys/class下地软链接名称。前者很自然，但后者却很难想到。即使简单的地方，经过重重调试，我们也会惊讶于linux的心细如发。

```
static int device_move_class_links(struct device *dev,
                struct device *old_parent,
                struct device *new_parent)
{
    int error = 0;
    if (old_parent)
        sysfs_remove_link(&dev->kobj, "device");
```

```
    if (new_parent)
        error = sysfs_create_link(&dev->kobj, &new_parent->kobj,
                      "device");
    return error;
#endif
}
```

device_move_class_links()只是一个内部函数，后面还有操纵它的那只手。这里的device_move_class_links显得很名不副实，并没用操作class中软链接的举动。这很正常，因为在sysfs中软链接是针对kobject来说的，所以即使位置变掉了，软链接还是很准确地定位。

```
/**
 * device_move - moves a device to a new parent
 * @dev: the pointer to the struct device to be moved
 * @new_parent: the new parent of the device (can by NULL)
 * @dpm_order: how to reorder the dpm_list
 */
int device_move(struct device *dev, struct device *new_parent,
        enum dpm_order dpm_order)
{
    int error;
    struct device *old_parent;
    struct kobject *new_parent_kobj;

    dev = get_device(dev);
    if (!dev)
        return -EINVAL;

    device_pm_lock();
    new_parent = get_device(new_parent);
    new_parent_kobj = get_device_parent(dev, new_parent);

    pr_debug("device: '%s': %s: moving to '%s'\n", dev_name(dev),
         __func__, new_parent ? dev_name(new_parent) : "<NULL>");
    error = kobject_move(&dev->kobj, new_parent_kobj);
    if (error) {
        cleanup_glue_dir(dev, new_parent_kobj);
        put_device(new_parent);
        goto out;
    }
    old_parent = dev->parent;
    dev->parent = new_parent;
    if (old_parent)
        klist_remove(&dev->p->knode_parent);
    if (new_parent) {
        klist_add_tail(&dev->p->knode_parent,
                   &new_parent->p->klist_children);
        set_dev_node(dev, dev_to_node(new_parent));
    }

    if (!dev->class)
        goto out_put;
    error = device_move_class_links(dev, old_parent, new_parent);
    if (error) {
        /* We ignore errors on cleanup since we're hosed anyway... */
        device_move_class_links(dev, new_parent, old_parent);
        if (!kobject_move(&dev->kobj, &old_parent->kobj)) {
            if (new_parent)
                klist_remove(&dev->p->knode_parent);
            dev->parent = old_parent;
            if (old_parent) {
                klist_add_tail(&dev->p->knode_parent,
                           &old_parent->p->klist_children);
                set_dev_node(dev, dev_to_node(old_parent));
            }
        }
        cleanup_glue_dir(dev, new_parent_kobj);
        put_device(new_parent);
        goto out;
    }
    switch (dpm_order) {
```

```
    case DPM_ORDER_NONE:
        break;
    case DPM_ORDER_DEV_AFTER_PARENT:
        device_pm_move_after(dev, new_parent);
        break;
    case DPM_ORDER_PARENT_BEFORE_DEV:
        device_pm_move_before(new_parent, dev);
        break;
    case DPM_ORDER_DEV_LAST:
        device_pm_move_last(dev);
        break;
    }
out_put:
    put_device(old_parent);
out:
    device_pm_unlock();
    put_device(dev);
    return error;
}
```

device_move()就是将dev移到一个新的parent下。但也有可能这个parent是空的。大部分操作围绕在引用计数上，get_device()，put_device()。而且换了新的parent，到底要加到sysfs中哪个目录下，还要再调用get_device_parent()研究一下。主要的操作就是kobject_move()和device_move_class_links()。因为在sysfs中软链接是针对kobject来说的，所以即使位置变掉了，软链接还是很很准确地定位，所以在/sys/dev、/sys/bus、/sys/class中的软链接都不用变，这实在是sysfs的一大优势。除此之外，device_move()还涉及到电源管理的问题，device移动影响到dev在dpm_list上的位置，我们对此不了解，先忽略之。

```
void device_shutdown(void)
{
    struct device *dev, *devn;

    list_for_each_entry_safe_reverse(dev, devn, &devices_kset->list,
                kobj.entry) {
        if (dev->bus && dev->bus->shutdown) {
            dev_dbg(dev, "shutdown\n");
            dev->bus->shutdown(dev);
        } else if (dev->driver && dev->driver->shutdown) {
            dev_dbg(dev, "shutdown\n");
            dev->driver->shutdown(dev);
        }
    }
    kobject_put(sysfs_dev_char_kobj);
    kobject_put(sysfs_dev_block_kobj);
    kobject_put(dev_kobj);
    async_synchronize_full();
}
```

这个device_shutdown()是在系统关闭时才调用的。它动用了很少使用的devices_kset，从而可以遍历到每个注册到sysfs上的设备，调用相应的总线或驱动定义的shutdown()函数。提起这个，还是在device_initialize()中将dev->kobj->kset统一设为devices_kset的。原来设备虽然有不同的parent，但kset还是一样的。这样我们就能理解/sys/devices下的顶层设备目录是怎么来的，因为没用parent，就在调用kobject_add()时将kset->kobj当成了parent，所以会直接挂在顶层目录下。这样的目录大致有pci0000:00、virtual等等。

看完了core.c，我有种明白机器人也是由零件组成的的感觉。linux设备驱动模型的大门已经打开了四分之一。���着分析的深入，我们大概也会越来越明白linux的良苦用心。

上一页 1 2 3 4 5 6 7 8 9 下一页 7

【内容导航】

第1页：连通世界的list
第2页：原子性操作atomic_t
第3页：记录生命周期的kref
第4页：更强的链表klist
第5页：设备驱动模型的基石kobject
第6页：设备驱动模型之device
第7页：设备驱动模型之driver
第8页：设备驱动模型之bus
第9页：设备驱动模型之device-driver