Linux内核部件分析

设备驱动模型之driver

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

上节我们分析设备驱动模型中的device，主要是drivers/base/core.c，可以说是代码量最大的一个文件。本节要分析的驱动driver，就要相对简单很多。原因也很简单，对于driver，我们能定义的公共部分实在不多，能再sysfs中表达的也很少。本节的分析将围绕drivers/base/driver.c，但头文件仍然是include/linux/device.h和drivers/base/base.h。

先让我们来看看driver的结构。

```
1. struct device_driver {
2.     const char      *name;
3.     struct bus_type     *bus;
4.
5.     struct module       *owner;
6.     const char      *mod_name;  /* used for built-in modules */
7.
8.     bool suppress_bind_attrs;   /* disables bind/unbind via sysfs */
9.
10.    int (*probe) (struct device *dev);
11.    int (*remove) (struct device *dev);
12.    void (*shutdown) (struct device *dev);
13.    int (*suspend) (struct device *dev, pm_message_t state);
14.    int (*resume) (struct device *dev);
15.    const struct attribute_group **groups;
16.
17.    const struct dev_pm_ops *pm;
18.
19.    struct driver_private *p;
20. };
```

struct device_driver就是模型定义的通用驱动结构。name是驱动名称，但这个name也只是在静态定义的初始名称，实际使用的名称还是由kobject中保管的。bus执行驱动所在的总线，owner是驱动所在的模块，还有一个所在模块名称mod_name，suppress_bind_attrs定义是否允许驱动通过sysfs决定挂载还是卸载设备。下面是一系列函数指针，probe是在驱动刚与设备挂接时调用的，remove是在设备卸载时调用的，shutdown是在设备关闭时调用的(说实话我现在还不知道remove和shutdown的区别），suspend是设备休眠时调用的，resume是设备恢复时调用的。group是属性集合，pm是电源管理的函数集合，p是指向driver_private的指针。

```
1. struct driver_private {
2.     struct kobject kobj;
3.     struct klist klist_devices;
4.     struct klist_node knode_bus;
5.     struct module_kobject *mkobj;
6.     struct device_driver *driver;
7. };
8. #define to_driver(obj) container_of(obj, struct driver_private, kobj)
```

与device类似，device_driver把与其它组件联系的大部分结构变量移到struct driver_private中来。首先是kobj，在sysfs中代表driver目录本身。klist_devices是驱动下的设备链表，knode_bus是要挂载在总线的驱动链表上的节点。mkobj是driver与相关module的联系，之前在device_driver结构中已经有指向module的指针，但这还不够，在/sys下你能发现一个module目录，所以驱动所属的模块在sysfs中也有显示，具体留到代码中再看。driver指针自然是从driver_private指回struct device_driver的。

```
1. struct driver_attribute {
2.     struct attribute attr;
3.     ssize_t (*show)(struct device_driver *driver, char *buf);
4.     ssize_t (*store)(struct device_driver *driver, const char *buf,
5.             size_t count);
6. };
7.
8. #define DRIVER_ATTR(_name, _mode, _show, _store)    \
9. struct driver_attribute driver_attr_##_name =     \
10.     __ATTR(_name, _mode, _show, _store)
```

除了以上两个结构，还有struct driver_attribute。driver_attribute是driver对struct attribute的封装，添加了两个特用于device_driver的读写函数。这种封装看似简单重复，工作量很小，但在使用时却会造成巨大的便利。

好，结构介绍完毕，下面看driver.c中的实现。

```
1. static struct device *next_device(struct klist_iter *i)
2. {
3.     struct klist_node *n = klist_next(i);
4.     struct device *dev = NULL;
5.     struct device_private *dev_prv;
6.
7.     if (n) {
8.         dev_prv = to_device_private_driver(n);
9.         dev = dev_prv->device;
10.    }
11.    return dev;
12. }
13.
14. int driver_for_each_device(struct device_driver *drv, struct device *start,
15.                void *data, int (*fn)(struct device *, void *))
16. {
17.    struct klist_iter i;
18.    struct device *dev;
19.    int error = 0;
20.
21.    if (!drv)
22.        return -EINVAL;
23.
24.    klist_iter_init_node(&drv->p->klist_devices, &i,
25.                 start ? &start->p->knode_driver : NULL);
26.    while ((dev = next_device(&i)) && !error)
27.        error = fn(dev, data);
28.    klist_iter_exit(&i);
29.    return error;
30. }
31. struct device *driver_find_device(struct device_driver *drv,
32.                 struct device *start, void *data,
33.                 int (*match)(struct device *dev, void *data))
34. {
35.    struct klist_iter i;
36.    struct device *dev;
37.
38.    if (!drv)
39.        return NULL;
40.
41.    klist_iter_init_node(&drv->p->klist_devices, &i,
42.                 (start ? &start->p->knode_driver : NULL));
43.    while ((dev = next_device(&i)))
44.        if (match(dev, data) && get_device(dev))
45.            break;
46.    klist_iter_exit(&i);
47.    return dev;
48. }
```

driver_for_each_device()是对drv的设备链表中的每个设备调用一次指定函数。

driver_find_device()是在drv的设备链表中寻找一个设备，寻找使用指定的匹配函数。

这两个函数都不陌生，在之前分析device的core.c中已经见到与它们很类似的函数，只不过那里是遍历设备的子设备链表，这里是遍历驱动的设备链表。next_device()同样是辅助用的内部函数。

```
1. int driver_create_file(struct device_driver *drv,
2.                struct driver_attribute *attr)
3. {
4.    int error;
5.    if (drv)
6.        error = sysfs_create_file(&drv->p->kobj, &attr->attr);
7.    else
8.        error = -EINVAL;
9.    return error;
10. }
11.
```

```
12. void driver_remove_file(struct device_driver *drv,
13.             struct driver_attribute *attr)
14. {
15.     if (drv)
16.         sysfs_remove_file(&drv->p->kobj, &attr->attr);
17. }
```

driver_create_file()创建drv下的属性文件，调用sysfs_create_file()实现。

driver_remove_file()删除drv下的属性文件，调用sysfs_remove_file()实现。

```
1. static int driver_add_groups(struct device_driver *drv,
2.                 const struct attribute_group **groups)
3. {
4.     int error = 0;
5.     int i;
6.
7.     if (groups) {
8.         for (i = 0; groups[i]; i++) {
9.             error = sysfs_create_group(&drv->p->kobj, groups[i]);
10.             if (error) {
11.                 while (--i >= 0)
12.                     sysfs_remove_group(&drv->p->kobj,
13.                                groups[i]);
14.                 break;
15.             }
16.         }
17.     }
18.     return error;
19. }
20.
21. static void driver_remove_groups(struct device_driver *drv,
22.                 const struct attribute_group **groups)
23. {
24.     int i;
25.
26.     if (groups)
27.         for (i = 0; groups[i]; i++)
28.             sysfs_remove_group(&drv->p->kobj, groups[i]);
29. }
```

driver_add_groups()在drv目录下添加属性集合，调用sysfs_create_groups()实现。

driver_remove_groups()在drv目录下删除属性集合，调用sysfs_remove_groups()实现。

发现两点问题：第一，是不是觉得driver_add_groups()不太合适，最好改为driver_create_groups()才搭调。但不只是driver用driver_add_groups()，device也使用device_add_groups()，不知一处这样做。第二，有没有发现driver_create_file()是外部函数，driver_add_groups()就是内部函数，也就是说driver只对外提供添加属性的接口，却不提供添加属性集合的接口。理由吗？在struct device_driver()已经专门定义了一个groups变量来添加属性集合，后面就不易再重复提供接口，而且创建属性集合需要的操作远比创建属性费时。在device中也是这样做的。

另外，driver中只提供管理属性文件的方法，却不提供管理二进制属性文件的方法，这是因为驱动本身没有这种需求，只有部分设备才要求二进制文件表示。

```
1. struct device_driver *get_driver(struct device_driver *drv)
2. {
3.     if (drv) {
4.         struct driver_private *priv;
5.         struct kobject *kobj;
6.
7.         kobj = kobject_get(&drv->p->kobj);
8.         priv = to_driver(kobj);
9.         return priv->driver;
10.     }
11.     return NULL;
12. }
13.
14. void put_driver(struct device_driver *drv)
15. {
16.     kobject_put(&drv->p->kobj);
17. }
```

get_driver()增加drv的引用计数，put_driver()减少drv的引用计数。这都是通过drv->p->kobj来做的。

```
1. struct device_driver *driver_find(const char *name, struct bus_type *bus)
```

```
2. {
3.     struct kobject *k = kset_find_obj(bus->p->drivers_kset, name);
4.     struct driver_private *priv;
5.
6.     if (k) {
7.         priv = to_driver(k);
8.         return priv->driver;
9.     }
10.    return NULL;
11. }
```

driver_find()从bus的驱动链表中寻找特定名称的driver。

```
1. /**
2.  * driver_register - register driver with bus
3.  * @drv: driver to register
4.  *
5.  * We pass off most of the work to the bus_add_driver() call,
6.  * since most of the things we have to do deal with the bus
7.  * structures.
8.  */
9. int driver_register(struct device_driver *drv)
10. {
11.     int ret;
12.     struct device_driver *other;
13.
14.     BUG_ON(!drv->bus->p);
15.
16.     if ((drv->bus->probe && drv->probe) ||
17.         (drv->bus->remove && drv->remove) ||
18.         (drv->bus->shutdown && drv->shutdown))
19.         printk(KERN_WARNING "Driver '%s' needs updating - please use "
20.             "bus_type methods\n", drv->name);
21.
22.     other = driver_find(drv->name, drv->bus);
23.     if (other) {
24.         put_driver(other);
25.         printk(KERN_ERR "Error: Driver '%s' is already registered, "
26.             "aborting...\n", drv->name);
27.         return -EBUSY;
28.     }
29.
30.     ret = bus_add_driver(drv);
31.     if (ret)
32.         return ret;
33.     ret = driver_add_groups(drv, drv->groups);
34.     if (ret)
35.         bus_remove_driver(drv);
36.     return ret;
37. }
```

driver_register()将drv注册到系统中。它真是做得难以预料地简单，所有的工作几乎完全是由bus_add_driver()代为完成的。但你要注意，在调用driver_register()前，drv->bus一定要预先设置。device可以不绑定bus，但driver一定要绑定到bus上。

```
1. void driver_unregister(struct device_driver *drv)
2. {
3.     if (!drv || !drv->p) {
4.         WARN(1, "Unexpected driver unregister!\n");
5.         return;
6.     }
7.     driver_remove_groups(drv, drv->groups);
8.     bus_remove_driver(drv);
9. }
```

driver_unregister()将drv从系统中撤销。大部分工作是调用bus_remove_driver()完成的。可以看出bus_add_driver()与bus_remove_driver()相对。driver和bus的联系如此紧密，以至于driver的注册和撤销工作都可以由bus代劳了。我们需要更进一步的分析。

经过调查，我们发现很有一部分driver的代码被移动到了bus.c中。我们本节是以driver为主，所以接下来会尽量在不惊动bus的情况下，分析存在于bus.c中的driver代码。

```
1. static ssize_t drv_attr_show(struct kobject *kobj, struct attribute *attr,
2.                 char *buf)
```

```
3. {
4.     struct driver_attribute *drv_attr = to_drv_attr(attr);
5.     struct driver_private *drv_priv = to_driver(kobj);
6.     ssize_t ret = -EIO;
7.
8.     if (drv_attr->show)
9.         ret = drv_attr->show(drv_priv->driver, buf);
10.    return ret;
11. }
12.
13. static ssize_t drv_attr_store(struct kobject *kobj, struct attribute *attr,
14.                 const char *buf, size_t count)
15. {
16.    struct driver_attribute *drv_attr = to_drv_attr(attr);
17.    struct driver_private *drv_priv = to_driver(kobj);
18.    ssize_t ret = -EIO;
19.
20.    if (drv_attr->store)
21.        ret = drv_attr->store(drv_priv->driver, buf, count);
22.    return ret;
23. }
24.
25. static struct sysfs_ops driver_sysfs_ops = {
26.    .show   = drv_attr_show,
27.    .store  = drv_attr_store,
28. };
```

看到这里，你终于觉得driver开始正常了，它还要定义sysfs读写时操作的函数。

```
1. static void driver_release(struct kobject *kobj)
2. {
3.     struct driver_private *drv_priv = to_driver(kobj);
4.
5.     pr_debug("driver: '%s': %s\n", kobject_name(kobj), __func__);
6.     kfree(drv_priv);
7. }
8.
9. static struct kobj_type driver_ktype = {
10.    .sysfs_ops  = &driver_sysfs_ops,
11.    .release    = driver_release,
12. };
```

与device的释放函数device_release不同，driver_release没有提供外界代码运行的机会，只是简单地释放drv_priv函数。

```
1. /* Manually detach a device from its associated driver. */
2. static ssize_t driver_unbind(struct device_driver *drv,
3.                 const char *buf, size_t count)
4. {
5.     struct bus_type *bus = bus_get(drv->bus);
6.     struct device *dev;
7.     int err = -ENODEV;
8.
9.     dev = bus_find_device_by_name(bus, NULL, buf);
10.    if (dev && dev->driver == drv) {
11.        if (dev->parent) /* Needed for USB */
12.            down(&dev->parent->sem);
13.        device_release_driver(dev);
14.        if (dev->parent)
15.            up(&dev->parent->sem);
16.        err = count;
17.    }
18.    put_device(dev);
19.    bus_put(bus);
20.    return err;
21. }
22. static DRIVER_ATTR(unbind, S_IWUSR, NULL, driver_unbind);
23.
24. /*
25.  * Manually attach a device to a driver.
26.  * Note: the driver must want to bind to the device,
27.  * it is not possible to override the driver's id table.
```

```
28. */
29. static ssize_t driver_bind(struct device_driver *drv,
30.              const char *buf, size_t count)
31. {
32.     struct bus_type *bus = bus_get(drv->bus);
33.     struct device *dev;
34.     int err = -ENODEV;
35.
36.     dev = bus_find_device_by_name(bus, NULL, buf);
37.     if (dev && dev->driver == NULL && driver_match_device(drv, dev)) {
38.         if (dev->parent) /* Needed for USB */
39.             down(&dev->parent->sem);
40.         down(&dev->sem);
41.         err = driver_probe_device(drv, dev);
42.         up(&dev->sem);
43.         if (dev->parent)
44.             up(&dev->parent->sem);
45.
46.         if (err > 0) {
47.             /* success */
48.             err = count;
49.         } else if (err == 0) {
50.             /* driver didn't accept device */
51.             err = -ENODEV;
52.         }
53.     }
54.     put_device(dev);
55.     bus_put(bus);
56.     return err;
57. }
58. static DRIVER_ATTR(bind, S_IWUSR, NULL, driver_bind);
```

上面描述了driver下两个只写的属性文件，unbind和bind。应该是提供用户空间命令是否将设备与驱动挂接的接口。

```
1. static int driver_add_attrs(struct bus_type *bus, struct device_driver *drv)
2. {
3.     int error = 0;
4.     int i;
5.
6.     if (bus->drv_attrs) {
7.         for (i = 0; attr_name(bus->drv_attrs[i]); i++) {
8.             error = driver_create_file(drv, &bus->drv_attrs[i]);
9.             if (error)
10.                goto err;
11.        }
12.    }
13. done:
14.    return error;
15. err:
16.    while (--i >= 0)
17.        driver_remove_file(drv, &bus->drv_attrs[i]);
18.    goto done;
19. }
20.
21. static void driver_remove_attrs(struct bus_type *bus,
22.                struct device_driver *drv)
23. {
24.    int i;
25.
26.    if (bus->drv_attrs) {
27.        for (i = 0; attr_name(bus->drv_attrs[i]); i++)
28.            driver_remove_file(drv, &bus->drv_attrs[i]);
29.    }
30. }
```

driver_add_attrs()向drv目录下添加属性，只是这些属性都是在bus中定义的drv_attrs[]。

driver_remove_attrs()从drv目录中删除相应的bus->drv_attrs[]。

```
1. static int __must_check add_bind_files(struct device_driver *drv)
2. {
```

```
    int ret;

    ret = driver_create_file(drv, &driver_attr_unbind);
    if (ret == 0) {
        ret = driver_create_file(drv, &driver_attr_bind);
        if (ret)
            driver_remove_file(drv, &driver_attr_unbind);
    }
    return ret;
}

static void remove_bind_files(struct device_driver *drv)
{
    driver_remove_file(drv, &driver_attr_bind);
    driver_remove_file(drv, &driver_attr_unbind);
}
```

add_bind_files()在drv目录下增加bind和unbind属性。

remove_bind_files()从drv目录下删除bind和unbind属性。

```
static ssize_t driver_uevent_store(struct device_driver *drv,
                   const char *buf, size_t count)
{
    enum kobject_action action;

    if (kobject_action_type(buf, count, &action) == 0)
        kobject_uevent(&drv->p->kobj, action);
    return count;
}
static DRIVER_ATTR(uevent, S_IWUSR, NULL, driver_uevent_store);
```

这是drv目录下地uevent属性文件，提供了从drv发送uevent的方法。

```
/**
 * bus_add_driver - Add a driver to the bus.
 * @drv: driver.
 */
int bus_add_driver(struct device_driver *drv)
{
    struct bus_type *bus;
    struct driver_private *priv;
    int error = 0;

    bus = bus_get(drv->bus);
    if (!bus)
        return -EINVAL;

    pr_debug("bus: '%s': add driver %s\n", bus->name, drv->name);

    priv = kzalloc(sizeof(*priv), GFP_KERNEL);
    if (!priv) {
        error = -ENOMEM;
        goto out_put_bus;
    }
    klist_init(&priv->klist_devices, NULL, NULL);
    priv->driver = drv;
    drv->p = priv;
    priv->kobj.kset = bus->p->drivers_kset;
    error = kobject_init_and_add(&priv->kobj, &driver_ktype, NULL,
                     "%s", drv->name);
    if (error)
        goto out_unregister;

    if (drv->bus->p->drivers_autoprobe) {
        error = driver_attach(drv);
        if (error)
            goto out_unregister;
    }
    klist_add_tail(&priv->knode_bus, &bus->p->klist_drivers);
    module_add_driver(drv->owner, drv);
```

```
38.
39.     error = driver_create_file(drv, &driver_attr_uevent);
40.     if (error) {
41.         printk(KERN_ERR "%s: uevent attr (%s) failed\n",
42.             __func__, drv->name);
43.     }
44.     error = driver_add_attrs(bus, drv);
45.     if (error) {
46.         /* How the hell do we get out of this pickle? Give up */
47.         printk(KERN_ERR "%s: driver_add_attrs(%s) failed\n",
48.             __func__, drv->name);
49.     }
50.
51.     if (!drv->suppress_bind_attrs) {
52.         error = add_bind_files(drv);
53.         if (error) {
54.             /* Ditto */
55.             printk(KERN_ERR "%s: add_bind_files(%s) failed\n",
56.                 __func__, drv->name);
57.         }
58.     }
59.
60.     kobject_uevent(&priv->kobj, KOBJ_ADD);
61.     return 0;
62.
63. out_unregister:
64.     kfree(drv->p);
65.     drv->p = NULL;
66.     kobject_put(&priv->kobj);
67. out_put_bus:
68.     bus_put(bus);
69.     return error;
70. }
```

bus_add_driver()看似是把drv与bus联系起来，其实是完成driver加入系统的大部分操作。

首先调用bus_get(drv->bus)增加对bus的引用。

分配并初始化drv->p，即driver_private结构。

调用kobject_init_and_add()将drv加入sysfs，之前只是设置了priv->obj.kset为bus->p->drivers_kset，所以drv目录会出现在bus目录的drivers子目录中。如果总线允许自动probe，就会调用driver_attach()将驱动和总线上的设备进行匹配，这个过程先略过。

然后调用klist_add_tail()将drv挂入总线的驱动链表。

调用module_add_driver()创建driver相关的模块在sysfs中的表示。后面专门描述。

调用driver_create_file()在drv目录下创建uevent属性文件。

调用driver_add_attrs()在drv目录下添加bus->driver_attrs[]中定义的属性。

如果drv->suppress_bind_attrs为零，即允许用户空间决定驱动何时链接和卸载设备，则调用add_bind_files()添加bind和unbind属性文件。

调用kobject_uevent()向用户空间发布KOBJ_ADD消息。

从bus_add_driver()的处理过程来看，driver只在bus的drivers目录下出现，没什么软链接，需要的属性也不多。

```
1. /**
2.  * bus_remove_driver - delete driver from bus's knowledge.
3.  * @drv: driver.
4.  *
5.  * Detach the driver from the devices it controls, and remove
6.  * it from its bus's list of drivers. Finally, we drop the reference
7.  * to the bus we took in bus_add_driver().
8.  */
9. void bus_remove_driver(struct device_driver *drv)
10. {
11.     if (!drv->bus)
12.         return;
13.
14.     if (!drv->suppress_bind_attrs)
15.         remove_bind_files(drv);
16.     driver_remove_attrs(drv->bus, drv);
17.     driver_remove_file(drv, &driver_attr_uevent);
```

```
    klist_remove(&drv->p->knode_bus);
    pr_debug("bus: '%s': remove driver %s\n", drv->bus->name, drv->name);
    driver_detach(drv);
    module_remove_driver(drv);
    kobject_put(&drv->p->kobj);
    bus_put(drv->bus);
}
```

bus_remove_driver()将drv从系统中撤销，与bus_add_driver()相对应。

driver真正精彩的地方在于probe函数，对设备的操作，对用户空间提供的接口，可惜这些都是特定的。这里只能将driver与bus联系起来，并在以后与device联系起来。

不过不必失望，下面我们分析下drivers/base/module.c，它显示了与驱动有关的module，在sysfs中的表现情况。

首先介绍使用到的结构。应该说module.c的代码实现很简单，但使用到的结构不简单。

```
struct module_attribute {
        struct attribute attr;
        ssize_t (*show)(struct module_attribute *, struct module *, char *);
        ssize_t (*store)(struct module_attribute *, struct module *,
             const char *, size_t count);
    void (*setup)(struct module *, const char *);
    int (*test)(struct module *);
    void (*free)(struct module *);
};

struct param_attribute
{
    struct module_attribute mattr;
    struct kernel_param *param;
};

struct module_param_attrs
{
    unsigned int num;
    struct attribute_group grp;
    struct param_attribute attrs[0];
};

struct module_kobject
{
    struct kobject kobj;
    struct module *mod;
    struct kobject *drivers_dir;
    struct module_param_attrs *mp;
};
```

可以看到module_attribute结构除了包含struct attribute，还多增加了好几条函数指针。而这只是最简单的，struct param_attribute除了包含module_attribute，还有一个指向kernel_param的指针param。这个kernel_param就太复杂了，是外界向module提供参数用的窗口，这里忽略。后面还有struct module_param_attrs和struct module_kobject。

```
static char *make_driver_name(struct device_driver *drv)
{
    char *driver_name;

    driver_name = kmalloc(strlen(drv->name) + strlen(drv->bus->name) + 2,
                  GFP_KERNEL);
    if (!driver_name)
        return NULL;

    sprintf(driver_name, "%s:%s", drv->bus->name, drv->name);
    return driver_name;
}
```

make_driver_name()将drv的名字和drv->bus的名字合起来，不过这是一个内部函数，具体使用还要看后面。

```
static void module_create_drivers_dir(struct module_kobject *mk)
{
    if (!mk || mk->drivers_dir)
        return;

    mk->drivers_dir = kobject_create_and_add("drivers", &mk->kobj);
```

```
7. }
```

module_create_drivers_dir()在mk所在的目录下创建一个drivers的目录。不过因为是使用kobject_create_and_add()，所以这个kobject使用默认的dynamic_kobj_ktype。

```
1. void module_add_driver(struct module *mod, struct device_driver *drv)
2. {
3.     char *driver_name;
4.     int no_warn;
5.     struct module_kobject *mk = NULL;
6.
7.     if (!drv)
8.         return;
9.
10.    if (mod)
11.        mk = &mod->mkobj;
12.    else if (drv->mod_name) {
13.        struct kobject *mkobj;
14.
15.        /* Lookup built-in module entry in /sys/modules */
16.        mkobj = kset_find_obj(module_kset, drv->mod_name);
17.        if (mkobj) {
18.            mk = container_of(mkobj, struct module_kobject, kobj);
19.            /* remember our module structure */
20.            drv->p->mkobj = mk;
21.            /* kset_find_obj took a reference */
22.            kobject_put(mkobj);
23.        }
24.    }
25.
26.    if (!mk)
27.        return;
28.
29.    /* Don't check return codes; these calls are idempotent */
30.    no_warn = sysfs_create_link(&drv->p->kobj, &mk->kobj, "module");
31.    driver_name = make_driver_name(drv);
32.    if (driver_name) {
33.        module_create_drivers_dir(mk);
34.        no_warn = sysfs_create_link(mk->drivers_dir, &drv->p->kobj,
35.                    driver_name);
36.        kfree(driver_name);
37.    }
38. }
```

module_add_drivers()在module下添加与driver的联系。

开始调用kset_find_obj()从module_kset下寻找drv所属的module对应的kobj。说明每个module在加载时都会在/sys/module中创建一个kobject目录。这里找到后只是将其赋给drv->p->kmobj，并调用kobject_put()释放找到时加上的引用计数。至于为什么driver不保留对module的引用计数，或许是不需要，或许是已经存在了。

接下来调用sysfs_create_link()在驱动目录中添加指向module目录的软链接，名称就是module。

调用module_create_drivers_dir()在module目录下建立drivers子目录。

调用sysfs_create_link()在drivers子目录下建立指向驱动目录的软链接，名称使用make_driver_name()的返回结果。

```
1. void module_remove_driver(struct device_driver *drv)
2. {
3.     struct module_kobject *mk = NULL;
4.     char *driver_name;
5.
6.     if (!drv)
7.         return;
8.
9.     sysfs_remove_link(&drv->p->kobj, "module");
10.
11.    if (drv->owner)
12.        mk = &drv->owner->mkobj;
13.    else if (drv->p->mkobj)
14.        mk = drv->p->mkobj;
15.    if (mk && mk->drivers_dir) {
16.        driver_name = make_driver_name(drv);
17.        if (driver_name) {
```

```
18.             sysfs_remove_link(mk->drivers_dir, driver_name);
19.             kfree(driver_name);
20.         }
21.     }
22. }
```

module_remove_driver()消除driver与相应module之间的软链接关系。

对于module，应该是另一个议题了，这里只是简单涉及，下节我们将涉及到总线bus，并深入分析device和driver的关系。

上一页 1 2 3 4 5 6 7 8 9 下一页 8

Linux内核的学习方法 Linux根目录下主要目录功能说明及常用分区方案

相关资讯 Linux内核

Linux内核Git源码树中的代码已达（今 20:48）
Linux 5.4.7 / 4.19.92 / 4.14.161 (01月01日)
Linux内核将用Rust编程语言编写？ (09/03/2019 12:06:17)
Linux内核将很快默认情况启用"- (05/11/2019 13:43:07)
Linux内核正在努力实现快速高效的I (02/15/2019 14:51:33)
Linux内核的冷热缓存 (01/27/2019 19:10:52)

本文评论 查看全部评论 (5)

表情： 姓名： 匿名 匿名 字数 0

同意评论声明 请登录

评论声明

- 尊重网上道德，遵守中华人民共和国的各项有关法律法规
- 承担一切因您的行为而直接或间接导致的民事或刑事法律责任
- 本站管理人员有权保留或删除其管辖留言中的任意内容
- 本站有权在网站内转载或引用您的评论
- 参与本评论即表明您已经阅读并接受上述条款

AlexXue 发表于 2018/6/22 11:29:47 第 5 楼

好吧，没有问题，当我没说

回复 支持 (0) 反对 (0)

AlexXue 发表于 2018/6/22 9:05:51 第 4 楼

作者，我认为你的__list_add函数有问题。麻烦画图分析一下。

回复 支持 (0) 反对 (0)

AlexXue 发表于 2018/6/20 10:07:20 第 3 楼

作者你好关于__list_add这个函数，我画图分析之后发现存在问题，烦请您贴图分析一下，感谢。

回复 支持 (0) 反对 (0)

lzxname 发表于 2014/11/4 9:48:24 第 2 楼

好东西。留一笔。

回复 支持 (14) 反对 (11)

lzxname 发表于 2014/11/4 9:43:13 第 1 楼

好东西啊啊。。。

回复 支持 (8) 反对 (13)