阅读新闻 背景：□□□□□□□□

Linux内核部件分析

设备驱动模型之device-driver

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

前面我们分析了device、driver、bus三种类型，主要是三者的注册与注销，在sysfs中的目录与属性文件创建等内容。本节就来详细分析下，在设备注册到总线上时，总线是如何为其寻找对应的驱动的；在驱动注册到总线上时，总线又是如何为其寻找对应的设备的。

本节的实现代码集中在drivers/base/bus.c和drivers/base/dd.c中。

先来回忆下，在device_register()->device_add()中，先是调用bus_add_device()添加device与bus间的联系，并添加bus为device定义的属性，然后会调用bus_probe_device()。bus_probe_device()会试图为已挂在总线上的该设备寻找对应的驱动。我们的故事就从这里开始。

```
1. /**
2.  * bus_probe_device - probe drivers for a new device
3.  * @dev: device to probe
4.  *
5.  * - Automatically probe for a driver if the bus allows it.
6.  */
7. void bus_probe_device(struct device *dev)
8. {
9.     struct bus_type *bus = dev->bus;
10.    int ret;
11.
12.    if (bus && bus->p->drivers_autoprobe) {
13.        ret = device_attach(dev);
14.        WARN_ON(ret < 0);
15.    }
16. }
```

bus_probe_device()为总线上的设备寻找驱动。它先是检查bus->p->drivers_autoprobe，看是否允许自动探测。允许了才会调用device_attach()进行实际的寻找工作。

说到bus->p->drivers_autoprobe这个变量，它是在bus_type_private中的，在调用bus_register()前都初始化不了，在bus_register()中自动定为1。所以，除非是用户空间通过drivers_autoprobe属性文件主动禁止，bus总是允许自动探测的，所有的bus都是如此。

```
1. /**
2.  * device_attach - try to attach device to a driver.
3.  * @dev: device.
4.  *
5.  * Walk the list of drivers that the bus has and call
6.  * driver_probe_device() for each pair. If a compatible
7.  * pair is found, break out and return.
8.  *
9.  * Returns 1 if the device was bound to a driver;
10.  * 0 if no matching driver was found;
11.  * -ENODEV if the device is not registered.
12.  *
13.  * When called for a USB interface, @dev->parent->sem must be held.
14.  */
15. int device_attach(struct device *dev)
16. {
17.     int ret = 0;
18.
19.     down(&dev->sem);
20.     if (dev->driver) {
21.         ret = device_bind_driver(dev);
22.         if (ret == 0)
23.             ret = 1;
24.         else {
25.             dev->driver = NULL;
26.             ret = 0;
```

```
27.         }
28.     } else {
29.         pm_runtime_get_noresume(dev);
30.         ret = bus_for_each_drv(dev->bus, NULL, dev, __device_attach);
31.         pm_runtime_put_sync(dev);
32.     }
33.     up(&dev->sem);
34.     return ret;
35. }
```

device_attach()在实际绑定之前，会用dev->sem进行加锁。不错，dev->sem几乎就是为了在设备与驱动绑定或者解除绑定时加锁用的。还没有看到它在其它地方被调用。

如果在调用device_attach()前就已经有了dev->driver()，就调用device_bind_driver()进行绑定，不然还要调用bus_for_each_drv()进行依次匹配。至于pm_runtime_get_noresume之类的函数，属于电源管理部分，我们现在先忽略。

```
1. static void driver_bound(struct device *dev)
2. {
3.     if (klist_node_attached(&dev->p->knode_driver)) {
4.         printk(KERN_WARNING "%s: device %s already bound\n",
5.             __func__, kobject_name(&dev->kobj));
6.         return;
7.     }
8.
9.     pr_debug("driver: '%s': %s: bound to device '%s'\n", dev_name(dev),
10.         __func__, dev->driver->name);
11.
12.     if (dev->bus)
13.         blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
14.                         BUS_NOTIFY_BOUND_DRIVER, dev);
15.
16.     klist_add_tail(&dev->p->knode_driver, &dev->driver->p->klist_devices);
17. }
18.
19. static int driver_sysfs_add(struct device *dev)
20. {
21.     int ret;
22.
23.     ret = sysfs_create_link(&dev->driver->p->kobj, &dev->kobj,
24.             kobject_name(&dev->kobj));
25.     if (ret == 0) {
26.         ret = sysfs_create_link(&dev->kobj, &dev->driver->p->kobj,
27.                     "driver");
28.         if (ret)
29.             sysfs_remove_link(&dev->driver->p->kobj,
30.                     kobject_name(&dev->kobj));
31.     }
32.     return ret;
33. }
34.
35. static void driver_sysfs_remove(struct device *dev)
36. {
37.     struct device_driver *drv = dev->driver;
38.
39.     if (drv) {
40.         sysfs_remove_link(&drv->p->kobj, kobject_name(&dev->kobj));
41.         sysfs_remove_link(&dev->kobj, "driver");
42.     }
43. }
44.
45. /**
46.  * device_bind_driver - bind a driver to one device.
47.  * @dev: device.
48.  *
49.  * Allow manual attachment of a driver to a device.
50.  * Caller must have already set @dev->driver.
51.  *
52.  * Note that this does not modify the bus reference count
53.  * nor take the bus's rwsem. Please verify those are accounted
54.  * for before calling this. (It is ok to call with no other effort
55.  * from a driver's probe() method.)
```

```
 *
 * This function must be called with @dev->sem held.
 */
int device_bind_driver(struct device *dev)
{
    int ret;

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

device_bind_driver()将device与driver绑定。它调用了两个内部函数。

其中drivers_sysfs_add()负责创建sysfs中driver和device指向对方的软链接。还有一个与它相对的函数drivers_sysfs_remove()。

driver_bound()则实际将device加入驱动的设备链表。

因为在调用device_bind_driver()之前就已经设置过dev->driver了，所以这样就将device和driver绑定了。

只是这样好像还缺少了什么，不错，之前看到driver时曾定义了drv->probe函数，bus->probe也有类似的功能，这里只是绑定，却没有调用probe函数。

让我们回过头来，继续看如果device_attach()中没有定义dev->driver会怎么样，是用bus_for_each_drv()对bus的驱动链表进行遍历，遍历函数使用__device_attach。

```
static int __device_attach(struct device_driver *drv, void *data)
{
    struct device *dev = data;

    if (!driver_match_device(drv, dev))
        return 0;

    return driver_probe_device(drv, dev);
}
```

不要小看了__device_attach()，就是在__device_attach()中既完成了匹配工作，又完成了绑定工作。bus_for_each_drv()在遍历中，如果遍历函数返回值不为0，则遍历结束。所以在__device_attach()找到并绑定了适合的驱动，就会返回1停止遍历，否则继续遍历剩余的驱动。

先来看匹配工作，这是在driver_match_device()中完成的。

```
static inline int driver_match_device(struct device_driver *drv,
                      struct device *dev)
{
    return drv->bus->match ? drv->bus->match(dev, drv) : 1;
}
```

原来driver_match_device()实际是调用drv->bus->match()来完成设备和驱动的匹配的。其实这也是理所当然。因为总线不同，总线规范设备、厂商、类设备等定义的规格都不同，也只有bus亲自主持匹配工作。再具体的就只能等分析具体总线的时候了。

```
int driver_probe_device(struct device_driver *drv, struct device *dev)
{
    int ret = 0;

    if (!device_is_registered(dev))
        return -ENODEV;

    pr_debug("bus: '%s': %s: matched device %s with driver %s\n",
         drv->bus->name, __func__, dev_name(dev), drv->name);

    pm_runtime_get_noresume(dev);
    pm_runtime_barrier(dev);
    ret = really_probe(dev, drv);
    pm_runtime_put_sync(dev);

    return ret;
}
```

如果driver_match_device()匹配成功了，__device_attach()就会继续调用driver_probe_devices()完成绑定。但driver_probe_devices()又是调用really_probe()完成的。

```
static atomic_t probe_count = ATOMIC_INIT(0);
static DECLARE_WAIT_QUEUE_HEAD(probe_waitqueue);

static int really_probe(struct device *dev, struct device_driver *drv)
```

```
{
    int ret = 0;

    atomic_inc(&probe_count);
    pr_debug("bus: '%s': %s: probing driver %s with device %s\n",
         drv->bus->name, __func__, drv->name, dev_name(dev));
    WARN_ON(!list_empty(&dev->devres_head));

    dev->driver = drv;
    if (driver_sysfs_add(dev)) {
        printk(KERN_ERR "%s: driver_sysfs_add(%s) failed\n",
            __func__, dev_name(dev));
        goto probe_failed;
    }

    if (dev->bus->probe) {
        ret = dev->bus->probe(dev);
        if (ret)
            goto probe_failed;
    } else if (drv->probe) {
        ret = drv->probe(dev);
        if (ret)
            goto probe_failed;
    }

    driver_bound(dev);
    ret = 1;
    pr_debug("bus: '%s': %s: bound device %s to driver %s\n",
         drv->bus->name, __func__, dev_name(dev), drv->name);
    goto done;

probe_failed:
    devres_release_all(dev);
    driver_sysfs_remove(dev);
    dev->driver = NULL;

    if (ret != -ENODEV && ret != -ENXIO) {
        /* driver matched but the probe failed */
        printk(KERN_WARNING
               "%s: probe of %s failed with error %d\n",
               drv->name, dev_name(dev), ret);
    }
    /*
     * Ignore errors returned by ->probe so that the next driver can try
     * its luck.
     */
    ret = 0;
done:
    atomic_dec(&probe_count);
    wake_up(&probe_waitqueue);
    return ret;
}
```

really_probe()完成的绑定工作和device_bind_driver()差不多，只是它还会调用bus->probe或者drv->probe中定义的probe函数。

至于在really_probe()中使用probe_count保护，最后调用wake_up(&probe_waitqueue)，都是为了进行同步。

```
/**
 * driver_probe_done
 * Determine if the probe sequence is finished or not.
 *
 * Should somehow figure out how to use a semaphore, not an atomic variable...
 */
int driver_probe_done(void)
{
    pr_debug("%s: probe_count = %d\n", __func__,
         atomic_read(&probe_count));
    if (atomic_read(&probe_count))
        return -EBUSY;
    return 0;
}
```

```
15.
16. /**
17.  * wait_for_device_probe
18.  * Wait for device probing to be completed.
19.  */
20. void wait_for_device_probe(void)
21. {
22.     /* wait for the known devices to complete their probing */
23.     wait_event(probe_waitqueue, atomic_read(&probe_count) == 0);
24.     async_synchronize_full();
25. }
```

driver_probe_done()检查当前是否有设备正在绑定驱动。

wait_for_device_probe()会阻塞到所有的设备绑定完驱动。

关于bus_probe_device()的过程就分析到这里，下面来看下bus_add_driver()又是怎样做的。

之前我们已经知道driver_register()把绝大部分操作都移到了bus_add_driver()中来。其中只有一点和设备与驱动的绑定相关，就是对driver_attach()的调用。

```
1. int driver_attach(struct device_driver *drv)
2. {
3.     return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
4. }
```

driver_attach()一如device_attach，只是这里是对总线的设备链表进行遍历，使用的遍历函数是__driver_attach()。

```
1. static int __driver_attach(struct device *dev, void *data)
2. {
3.     struct device_driver *drv = data;
4.
5.     /*
6.      * Lock device and try to bind to it. We drop the error
7.      * here and always return 0, because we need to keep trying
8.      * to bind to devices and some drivers will return an error
9.      * simply if it didn't support the device.
10.     *
11.     * driver_probe_device() will spit a warning if there
12.     * is an error.
13.     */
14.
15.     if (!driver_match_device(drv, dev))
16.         return 0;
17.
18.     if (dev->parent) /* Needed for USB */
19.         down(&dev->parent->sem);
20.     down(&dev->sem);
21.     if (!dev->driver)
22.         driver_probe_device(drv, dev);
23.     up(&dev->sem);
24.     if (dev->parent)
25.         up(&dev->parent->sem);
26.
27.     return 0;
28. }
```

在__driver_attach()中，driver_match_device()就不说了，它是调到bus->match去的。

然后依然是加锁，调用driver_probe_device()函数。这就与__device_attach()的路径一致了。

不要以为就这样结束了，现在我们只是看到了把device和driver绑定到一起的方法，却没有看到解除绑定的方法。

既然绑定的方法是在设备和驱动注册的时候调用的，那解除绑定自然是在设备或驱动注销的时候。

还是先来看设备的，device_unregister()->device_del()会调用bus_remove_device()将设备从总线上删除。

bus_remove_device()是与bus_add_device()相对的，但也不仅如此，它还调用了device_release_driver()来解除与driver的绑定。

```
1. /**
2.  * device_release_driver - manually detach device from driver.
3.  * @dev: device.
4.  *
```

```
 * Manually detach device from driver.
 * When called for a USB interface, @dev->parent->sem must be held.
 */
void device_release_driver(struct device *dev)
{
    /*
     * If anyone calls device_release_driver() recursively from
     * within their ->remove callback for the same device, they
     * will deadlock right here.
     */
    down(&dev->sem);
    __device_release_driver(dev);
    up(&dev->sem);
}

/*
 * __device_release_driver() must be called with @dev->sem held.
 * When called for a USB interface, @dev->parent->sem must be held as well.
 */
static void __device_release_driver(struct device *dev)
{
    struct device_driver *drv;

    drv = dev->driver;
    if (drv) {
        pm_runtime_get_noresume(dev);
        pm_runtime_barrier(dev);

        driver_sysfs_remove(dev);

        if (dev->bus)
            blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                         BUS_NOTIFY_UNBIND_DRIVER,
                         dev);

        if (dev->bus && dev->bus->remove)
            dev->bus->remove(dev);
        else if (drv->remove)
            drv->remove(dev);
        devres_release_all(dev);
        dev->driver = NULL;
        klist_remove(&dev->p->knode_driver);
        if (dev->bus)
            blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                         BUS_NOTIFY_UNBOUND_DRIVER,
                         dev);

        pm_runtime_put_sync(dev);
    }
}
```

device_release_driver()还是负责加加锁，实际的工作由__device_release_driver()来完成。

除了sysfs和结构中解除绑定的操作，还调用了bus->remove或者driver->remove。

虽然device注销时与driver解除绑定很简单，但driver注销要与device解除绑定就要复杂一些，因为它要与设备链表上所有的设备解除绑定。

在driver_unregister()->bus_remove_driver()中，调用了driver_detach()函数。

```
/**
 * driver_detach - detach driver from all devices it controls.
 * @drv: driver.
 */
void driver_detach(struct device_driver *drv)
{
    struct device_private *dev_prv;
    struct device *dev;

    for (;;) {
        spin_lock(&drv->p->klist_devices.k_lock);
```

```
12.         if (list_empty(&drv->p->klist_devices.k_list)) {
13.             spin_unlock(&drv->p->klist_devices.k_lock);
14.             break;
15.         }
16.         dev_prv = list_entry(drv->p->klist_devices.k_list.prev,
17.                     struct device_private,
18.                     knode_driver.n_node);
19.         dev = dev_prv->device;
20.         get_device(dev);
21.         spin_unlock(&drv->p->klist_devices.k_lock);
22.
23.         if (dev->parent) /* Needed for USB */
24.             down(&dev->parent->sem);
25.         down(&dev->sem);
26.         if (dev->driver == drv)
27.             __device_release_driver(dev);
28.         up(&dev->sem);
29.         if (dev->parent)
30.             up(&dev->parent->sem);
31.         put_device(dev);
32.     }
33. }
```

可以看到，driver_detach()基本操作就是与设备链表上的设备解除绑定。等了这么久，终于有个有点意思的地方。一看这个drv的设备链表遍历，首先明明是klist，却没使用标准的循环函数，奇怪，然后发现竟然没有将设备卸下链表的地方，更奇怪。其实再一想就明白了。你看到list_entry()中，是从设备链表末尾取设备解除绑定的，这是驱动生怕前面的设备解除绑定了，后面的就不工作了。也正是因为klist遍历是逆向的，所以无法使用标准函数。至于将设备卸下链表的地方，是在__device_release_driver()中。

或许会奇怪这里为什么会有get_device()和put_device()的操作。这是为了防止设备一取下链表，就会释放最后一个引用计数，导致直接注销。那时候的情况，一定是在占用了dev->sem的同时去等待dev->sem，通俗来说就是死锁。

通过driver_attach()和driver_detach()的训练，我们已经习惯在为设备加锁时，顺便为其父设备加锁。虽然在device_attach()和device_release_driver()中只是对设备本身加锁。或许是害怕在驱动与设备解除绑定的过程中，父设备突然也要解除绑定，导致不一致状态。为至于为什么设备方主动要求时不需要对父设备加锁，或许是设备的主动申请更靠谱，不会在子设备绑定或释放的同时，父设备也申请释放。总之，在linux看来，设备恐怕比驱动还要靠谱一些，从driver和bus的引用计数，从这里的加锁情况，都可以看出一二。

```
1. void *dev_get_drvdata(const struct device *dev)
2. {
3.     if (dev && dev->p)
4.         return dev->p->driver_data;
5.     return NULL;
6. }
7.
8. void dev_set_drvdata(struct device *dev, void *data)
9. {
10.     int error;
11.
12.     if (!dev)
13.         return;
14.     if (!dev->p) {
15.         error = device_private_init(dev);
16.         if (error)
17.             return;
18.     }
19.     dev->p->driver_data = data;
20. }
```

最后的dev_set_drvdata()是在dev->p->driver_data中存放驱动定义的数据。dev_get_drvdata()是获取这个数据。

不要 小看这个device_private结构中小小的driver_data，在驱动编写中总能派上大用场。当然也不是说没有driver_data就过不下去，毕竟驱动可以定义一个自己的device结构，并把通用的struct device内嵌其中，然后想放多少数据都行。可那样太麻烦，许多驱动都要专门设置这样一个变量，索性加到通用的数据结构中。而且是直接加到device_private中，眼不见为净，方便省事。

```
1. /**
2.  * device_reprobe - remove driver for a device and probe for a new driver
3.  * @dev: the device to reprobe
4.  *
5.  * This function detaches the attached driver (if any) for the given
6.  * device and restarts the driver probing process.  It is intended
7.  * to use if probing criteria changed during a devices lifetime and
8.  * driver attachment should change accordingly.
9.  */
```

```
10. int device_reprobe(struct device *dev)
11. {
12.     if (dev->driver) {
13.         if (dev->parent)        /* Needed for USB */
14.             down(&dev->parent->sem);
15.         device_release_driver(dev);
16.         if (dev->parent)
17.             up(&dev->parent->sem);
18.     }
19.     return bus_rescan_devices_helper(dev, NULL);
20. }
```

device_reprobe()显然是dev对之前的驱动不满意，要新绑定一个。

```
1. static int __must_check bus_rescan_devices_helper(struct device *dev,
2.                         void *data)
3. {
4.     int ret = 0;
5.
6.     if (!dev->driver) {
7.         if (dev->parent) /* Needed for USB */
8.             down(&dev->parent->sem);
9.         ret = device_attach(dev);
10.         if (dev->parent)
11.             up(&dev->parent->sem);
12.     }
13.     return ret < 0 ? ret : 0;
14. }
```

bus_rescan_devices_helper()就是用来绑定新驱动的内部函数。

我们终于成功完成了对dd.c的分析，并将bus.c剩余的部分结了尾。想必大家已经充分领略了device、driver和bus的铁三角结构，下节我们将进入设备驱动模型的另一方天地。

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