Linux内核部件分析

设备驱动模型之bus

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

前面我们分析了设备驱动模型中的device和driver，device和driver本来是不相关的东西，只因为bus的存在，才被联系到了一起。本节就来看看设备驱动模型中起枢纽作用的bus。本节的头文件在include/linux/device.h和drivers/base/base.h，实现代码主要在bus.c中。因为在bus中有很多代码时为了device找到driver或者driver找到device而定义的，本节先尽量忽略这部分，专注于bus的注册和注销，属性定义等内容。剩下的留到讨论device和driver关系时在分析。

先来看看bus的数据结构。

```
1. struct bus_type {
2.     const char      *name;
3.     struct bus_attribute    *bus_attrs;
4.     struct device_attribute *dev_attrs;
5.     struct driver_attribute *drv_attrs;
6.
7.     int (*match)(struct device *dev, struct device_driver *drv);
8.     int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
9.     int (*probe)(struct device *dev);
10.    int (*remove)(struct device *dev);
11.    void (*shutdown)(struct device *dev);
12.
13.    int (*suspend)(struct device *dev, pm_message_t state);
14.    int (*resume)(struct device *dev);
15.
16.    const struct dev_pm_ops *pm;
17.
18.    struct bus_type_private *p;
19. };
```

struct bus_type是bus的通用数据结构。

name是bus的名称，注意到这里也是const char类型的，在sysfs中使用的还是kobj中动态创建的名称，这里的name只是初始名。

bus_attrs是bus为自己定义的一系列属性，dev_attrs是bus为旗下的device定义的一系列属性，drv_attrs是bus为旗下的driver定义的一系列属性。其中dev_attrs在bus_add_device()->device_add_attrs()中被加入dev目录下，drv_attrs在bus_add_driver()->driver_add_attrs()中被加入driver目录下。

match函数匹配总线中的dev和driver，返回值为1代表匹配成功，为0则失败。

uevent函数用于总线对uevent的环境变量添加，但在总线下设备的dev_uevent处理函数也有对它的调用。

probe函数是总线在匹配成功时调用的函数，bus->probe和drv->probe中只会有一个起效，同时存在时使用bus->probe。

remove函数在总线上设备或者驱动要删除时调用，bus->remove和drv->remove中同样只会有一个起效。

shutdown函数在所有设备都关闭时调用，即在core.c中的device_shutdown()函数中调用，bus->shutdown和drv->shutdown同样只会有一个起效。

suspend函数是在总线上设备休眠时调用。

resume函数是在总线上设备恢复时调用。

pm是struct dev_pm_ops类型，其中定义了一系列电源管理的函数。

p是指向bus_type_private的指针，其中定义了将bus同其它组件联系起来的变量。

```
1. struct bus_type_private {
2.     struct kset subsys;
3.     struct kset *drivers_kset;
4.     struct kset *devices_kset;
5.     struct klist klist_devices;
6.     struct klist klist_drivers;
7.     struct blocking_notifier_head bus_notifier;
8.     unsigned int drivers_autoprobe:1;
9.     struct bus_type *bus;
10. };
```

```
11.
12. #define to_bus(obj) container_of(obj, struct bus_type_private, subsys.kobj)
```

struct bus_type_private是将bus同device、driver、sysfs联系起来的结构。

subsys是kset类型，代表bus在sysfs中的类型。

drivers_kset代表bus目录下的drivers子目录。
devices_kset代表bus目录下地devices子目录。

klist_devices是bus的设备链表，klist_drivers是bus的驱动链表。

bus_notifier用于在总线上内容发送变化时调用特定的函数，这里略过。

driver_autoprobe标志定义是否允许device和driver自动匹配，如果允许会在device或者driver注册时就进行匹配工作。

bus指针指向struct bus_type类型。

使用struct bus_type_private可以将struct bus_type中的部分细节屏蔽掉，利于外界使用bus_type。struct driver_private和struct device_private都有类似的功能。

```
1. struct bus_attribute {
2.     struct attribute    attr;
3.     ssize_t (*show)(struct bus_type *bus, char *buf);
4.     ssize_t (*store)(struct bus_type *bus, const char *buf, size_t count);
5. };
6.
7. #define BUS_ATTR(_name, _mode, _show, _store)   \
8. struct bus_attribute bus_attr_##_name = __ATTR(_name, _mode, _show, _store)
9.
10. #define to_bus_attr(_attr) container_of(_attr, struct bus_attribute, attr)
```

struct bus_attribute是bus对struct attribute类型的封装，更方便总线属性的定义。

```
1. static ssize_t bus_attr_show(struct kobject *kobj, struct attribute *attr,
2.                 char *buf)
3. {
4.     struct bus_attribute *bus_attr = to_bus_attr(attr);
5.     struct bus_type_private *bus_priv = to_bus(kobj);
6.     ssize_t ret = 0;
7.
8.     if (bus_attr->show)
9.         ret = bus_attr->show(bus_priv->bus, buf);
10.     return ret;
11. }
12.
13. static ssize_t bus_attr_store(struct kobject *kobj, struct attribute *attr,
14.                 const char *buf, size_t count)
15. {
16.     struct bus_attribute *bus_attr = to_bus_attr(attr);
17.     struct bus_type_private *bus_priv = to_bus(kobj);
18.     ssize_t ret = 0;
19.
20.     if (bus_attr->store)
21.         ret = bus_attr->store(bus_priv->bus, buf, count);
22.     return ret;
23. }
24.
25. static struct sysfs_ops bus_sysfs_ops = {
26.     .show   = bus_attr_show,
27.     .store  = bus_attr_store,
28. };
29.
30. static struct kobj_type bus_ktype = {
31.     .sysfs_ops  = &bus_sysfs_ops,
32. };
```

以上应该是我们最熟悉的部分，bus_ktype中定义了bus对应的kset应该使用的kobj_type实例。与此类似，driver使用的是自定义的driver_ktype，device使用的是自定义的device_ktype。只是这里仅仅定义了sysfs_ops，并未定义release函数，不知bus_type_private打算何时释放。

```
1. int bus_create_file(struct bus_type *bus, struct bus_attribute *attr)
2. {
3.     int error;
4.     if (bus_get(bus)) {
5.         error = sysfs_create_file(&bus->p->subsys.kobj, &attr->attr);
```

```
6.          bus_put(bus);
7.      } else
8.          error = -EINVAL;
9.      return error;
10. }
11.
12. void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr)
13. {
14.     if (bus_get(bus)) {
15.         sysfs_remove_file(&bus->p->subsys.kobj, &attr->attr);
16.         bus_put(bus);
17.     }
18. }
```

bus_create_file()在bus目录下创建属性文件，bus_remove_file()在bus目录下删除属性文件。类似的函数在driver和device中都有见到。

```
1. static int bus_uevent_filter(struct kset *kset, struct kobject *kobj)
2. {
3.     struct kobj_type *ktype = get_ktype(kobj);
4.
5.     if (ktype == &bus_ktype)
6.         return 1;
7.     return 0;
8. }
9.
10. static struct kset_uevent_ops bus_uevent_ops = {
11.     .filter = bus_uevent_filter,
12. };
13.
14. static struct kset *bus_kset;
```

可以看到这里定义了一个bus_uevent_ops变量，这是kset对uevent事件处理所用的结构，它会用在bus_kset中。

```
1. int __init buses_init(void)
2. {
3.     bus_kset = kset_create_and_add("bus", &bus_uevent_ops, NULL);
4.     if (!bus_kset)
5.         return -ENOMEM;
6.     return 0;
7. }
```

在buses_init()中创建了/sys/bus目录，这是一个kset类型，使用了bus_uevent_ops的uevent操作类型。

其实这里的操作不难想象，在devices中我们有一个类似的devices_kset，可以回顾一下。

```
1. static struct kset_uevent_ops device_uevent_ops = {
2.     .filter =   dev_uevent_filter,
3.     .name =     dev_uevent_name,
4.     .uevent =   dev_uevent,
5. };
6.
7. /* kset to create /sys/devices/  */
8. struct kset *devices_kset;
9.
10. int __init devices_init(void)
11. {
12.     devices_kset = kset_create_and_add("devices", &device_uevent_ops, NULL);
13.                 ...
14. }
15.
16. void device_initialize(struct device *dev)
17. {
18.     dev->kobj.kset = devices_kset;
19.                 ...
20. }
```

devices_kset在devices_init()中被创建，使用相应的device_uevent_ops进行uevent处理。而devices_kset又被设为每个device初始化时使用的kset。这就不难想象每个device都是以devices_kset为所属kset的，并使用device_uevent_ops中的处理函数。

只是这里还不知bus_kset会在哪里用到，或许是每个bus所属的kset吧，下面会有答案。

```
1. static ssize_t show_drivers_autoprobe(struct bus_type *bus, char *buf)
2. {
```

```
    return sprintf(buf, "%d\n", bus->p->drivers_autoprobe);
}

static ssize_t store_drivers_autoprobe(struct bus_type *bus,
                       const char *buf, size_t count)
{
    if (buf[0] == '0')
        bus->p->drivers_autoprobe = 0;
    else
        bus->p->drivers_autoprobe = 1;
    return count;
}

static ssize_t store_drivers_probe(struct bus_type *bus,
                   const char *buf, size_t count)
{
    struct device *dev;

    dev = bus_find_device_by_name(bus, NULL, buf);
    if (!dev)
        return -ENODEV;
    if (bus_rescan_devices_helper(dev, NULL) != 0)
        return -EINVAL;
    return count;
}

static BUS_ATTR(drivers_probe, S_IWUSR, NULL, store_drivers_probe);
static BUS_ATTR(drivers_autoprobe, S_IWUSR | S_IRUGO,
        show_drivers_autoprobe, store_drivers_autoprobe);
```

这里定义了总线下的两个属性，只写得drivers_probe，和可读写的drivers_autoprobe。至于其怎么实现的，我们现在还不关心。

```
static int add_probe_files(struct bus_type *bus)
{
    int retval;

    retval = bus_create_file(bus, &bus_attr_drivers_probe);
    if (retval)
        goto out;

    retval = bus_create_file(bus, &bus_attr_drivers_autoprobe);
    if (retval)
        bus_remove_file(bus, &bus_attr_drivers_probe);
out:
    return retval;
}

static void remove_probe_files(struct bus_type *bus)
{
    bus_remove_file(bus, &bus_attr_drivers_autoprobe);
    bus_remove_file(bus, &bus_attr_drivers_probe);
}
```

add_probe_files()在bus目录下添加drivers_probe和drivers_autoprobe文件。

remove_probe_files()在bus目录下删除drivers_probe和drivers_autoprobe文件。

这两个函数对bus的probe类型属性进行管理，就像add_bind_files/remove_bind_files对driver的bind类型属性进行管理一样。

```
static ssize_t bus_uevent_store(struct bus_type *bus,
                const char *buf, size_t count)
{
    enum kobject_action action;

    if (kobject_action_type(buf, count, &action) == 0)
        kobject_uevent(&bus->p->subsys.kobj, action);
    return count;
}
static BUS_ATTR(uevent, S_IWUSR, NULL, bus_uevent_store);
```

上面定义了bus的一个属性uevent，用于bus所在的kset节点主动发起uevent消息。

同样地uevent文件在driver目录中也有见到。device目录中也有，不过除了store_uevent之外，还增加了show_uevent的功能。

```
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_bus(n);
9.         dev = dev_prv->device;
10.    }
11.    return dev;
12. }
13.
14. int bus_for_each_dev(struct bus_type *bus, 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 (!bus)
22.        return -EINVAL;
23.
24.    klist_iter_init_node(&bus->p->klist_devices, &i,
25.                 (start ? &start->p->knode_bus : NULL));
26.    while ((dev = next_device(&i)) && !error)
27.        error = fn(dev, data);
28.    klist_iter_exit(&i);
29.    return error;
30. }
31.
32. struct device *bus_find_device(struct bus_type *bus,
33.                 struct device *start, void *data,
34.                 int (*match)(struct device *dev, void *data))
35. {
36.    struct klist_iter i;
37.    struct device *dev;
38.
39.    if (!bus)
40.        return NULL;
41.
42.    klist_iter_init_node(&bus->p->klist_devices, &i,
43.                 (start ? &start->p->knode_bus : NULL));
44.    while ((dev = next_device(&i)))
45.        if (match(dev, data) && get_device(dev))
46.            break;
47.    klist_iter_exit(&i);
48.    return dev;
49. }
```

bus_for_each_dev()是以bus的设备链表中每个设备为参数，调用指定的处理函数。

bus_find_device()是寻找bus设备链表中的某个设备，使用指定的匹配函数。

这两个函数提供遍历bus的设备链表的方法，类似于drivers_for_each_device/drivers_find_device对driver的设备链表的遍历，device_for_each_child/device_find_child对device的子设备链表的遍历。

```
1. static int match_name(struct device *dev, void *data)
2. {
3.     const char *name = data;
4.
5.     return sysfs_streq(name, dev_name(dev));
6. }
7.
8. struct device *bus_find_device_by_name(struct bus_type *bus,
9.                     struct device *start, const char *name)
10. {
11.    return bus_find_device(bus, start, (void *)name, match_name);
12. }
```

bus_find_device_by_name()给出了如何使用遍历函数的例子，寻找bus设备链表中指定名称的设备。

```
1. static struct device_driver *next_driver(struct klist_iter *i)
2. {
3.     struct klist_node *n = klist_next(i);
4.     struct driver_private *drv_priv;
5.
6.     if (n) {
7.         drv_priv = container_of(n, struct driver_private, knode_bus);
8.         return drv_priv->driver;
9.     }
10.    return NULL;
11. }
12.
13. int bus_for_each_drv(struct bus_type *bus, struct device_driver *start,
14.                void *data, int (*fn)(struct device_driver *, void *))
15. {
16.    struct klist_iter i;
17.    struct device_driver *drv;
18.    int error = 0;
19.
20.    if (!bus)
21.        return -EINVAL;
22.
23.    klist_iter_init_node(&bus->p->klist_drivers, &i,
24.                 start ? &start->p->knode_bus : NULL);
25.    while ((drv = next_driver(&i)) && !error)
26.        error = fn(drv, data);
27.    klist_iter_exit(&i);
28.    return error;
29. }
```

bus_for_each_drv()对bus的驱动链表中的每个驱动调用指定的函数。

这和前面的bus_for_each_dev/bus_find_dev什么都是类似的，只是你可能怀疑为什么会没有bus_find_drv。是没有它的用武之地吗？

请看driver.c中的driver_find()函数。

```
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()函数是在bus的驱动链表中寻找指定名称的驱动，它的存在证明bus_find_drv()完全是用得上的。可linux却偏偏没有实现bus_find_drv。driver_find()的实现也因此一直走内层路线，它直接用kset_find_obj()进行kobect的名称匹配，调用to_driver()等内容将kobj转化为drv。首先这完全不同于bus_for_each_drv()等一系列遍历函数，它们走的都是在klist中寻找的路线，这里确实走的sysfs中kset内部链表。其次，这里其实也是获得了drv的一个引用计数，在kset_find_obj()中会增加匹配的kobj的引用计数，driver_find()并没有释放，就相当于获取了drv的一个引用计数。这样虽然也可以，但代码写得很不优雅。可见人无完人，linux代码还有许多可改进之处。当然，也可能在最新的linux版本中已经改正了。

```
1. static int bus_add_attrs(struct bus_type *bus)
2. {
3.     int error = 0;
4.     int i;
5.
6.     if (bus->bus_attrs) {
7.         for (i = 0; attr_name(bus->bus_attrs[i]); i++) {
8.             error = bus_create_file(bus, &bus->bus_attrs[i]);
9.             if (error)
10.                goto err;
11.        }
12.    }
13. done:
14.    return error;
15. err:
16.    while (--i >= 0)
17.        bus_remove_file(bus, &bus->bus_attrs[i]);
18.    goto done;
```

```
}

static void bus_remove_attrs(struct bus_type *bus)
{
    int i;

    if (bus->bus_attrs) {
        for (i = 0; attr_name(bus->bus_attrs[i]); i++)
            bus_remove_file(bus, &bus->bus_attrs[i]);
    }
}
```

bus_add_attrs()将bus->bus_attrs中定义的属性加入bus目录。

bus_remove_attrs()将bus->bus_attrs中定义的属性删除。

开始看struct bus_type时我们说到结构中的bus_attrs、dev_attrs、drv_attrs三种属性，后两者分别在device_add_attrs()和driver_add_attrs()中添加，最后的bus_attrs也终于在bus_add_attrs()中得到添加。只是它们虽然都定义在bus_type中，确实添加在完全不同的三个地方。

```
static void klist_devices_get(struct klist_node *n)
{
    struct device_private *dev_prv = to_device_private_bus(n);
    struct device *dev = dev_prv->device;

    get_device(dev);
}

static void klist_devices_put(struct klist_node *n)
{
    struct device_private *dev_prv = to_device_private_bus(n);
    struct device *dev = dev_prv->device;

    put_device(dev);
}
```

klist_devices_get()用于bus设备链表上添加节点时增加对相应设备的引用。

klist_devices_put()用于bus设备链表上删除节点时减少对相应设备的引用。

相似的函数是device中的klist_children_get/klist_children_put，这是device的子设备链表。除此之外，bus的驱动链表和driver的设备链表，都没有这种引用计数的保护。原因还未知，也许是linux觉得驱动不太靠谱，万一突然当掉，也不至于影响device的正常管理。

```
/**
 * bus_register - register a bus with the system.
 * @bus: bus.
 *
 * Once we have that, we registered the bus with the kobject
 * infrastructure, then register the children subsystems it has:
 * the devices and drivers that belong to the bus.
 */
int bus_register(struct bus_type *bus)
{
    int retval;
    struct bus_type_private *priv;

    priv = kzalloc(sizeof(struct bus_type_private), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;

    priv->bus = bus;
    bus->p = priv;

    BLOCKING_INIT_NOTIFIER_HEAD(&priv->bus_notifier);

    retval = kobject_set_name(&priv->subsys.kobj, "%s", bus->name);
    if (retval)
        goto out;

    priv->subsys.kobj.kset = bus_kset;
    priv->subsys.kobj.ktype = &bus_ktype;
    priv->drivers_autoprobe = 1;

    retval = kset_register(&priv->subsys);
```

```
32.     if (retval)
33.         goto out;
34.
35.     retval = bus_create_file(bus, &bus_attr_uevent);
36.     if (retval)
37.         goto bus_uevent_fail;
38.
39.     priv->devices_kset = kset_create_and_add("devices", NULL,
40.                          &priv->subsys.kobj);
41.     if (!priv->devices_kset) {
42.         retval = -ENOMEM;
43.         goto bus_devices_fail;
44.     }
45.
46.     priv->drivers_kset = kset_create_and_add("drivers", NULL,
47.                          &priv->subsys.kobj);
48.     if (!priv->drivers_kset) {
49.         retval = -ENOMEM;
50.         goto bus_drivers_fail;
51.     }
52.
53.     klist_init(&priv->klist_devices, klist_devices_get, klist_devices_put);
54.     klist_init(&priv->klist_drivers, NULL, NULL);
55.
56.     retval = add_probe_files(bus);
57.     if (retval)
58.         goto bus_probe_files_fail;
59.
60.     retval = bus_add_attrs(bus);
61.     if (retval)
62.         goto bus_attrs_fail;
63.
64.     pr_debug("bus: '%s': registered\n", bus->name);
65.     return 0;
66.
67. bus_attrs_fail:
68.     remove_probe_files(bus);
69. bus_probe_files_fail:
70.     kset_unregister(bus->p->drivers_kset);
71. bus_drivers_fail:
72.     kset_unregister(bus->p->devices_kset);
73. bus_devices_fail:
74.     bus_remove_file(bus, &bus_attr_uevent);
75. bus_uevent_fail:
76.     kset_unregister(&bus->p->subsys);
77.     kfree(bus->p);
78. out:
79.     bus->p = NULL;
80.     return retval;
81. }
```

bus_register()将bus注册到系统中。

先分配并初始化bus->p，名称使用bus->name，所属的kset使用bus_kset（果然不出所料），类型使用bus_ktype。bus_ktype的使用同driver中的driver_ktype，和device中的device_ktype一样，都是自定义的kobj_type，要知道kobj_type的使用关系到release函数，和自定义属性类型能否正常发挥。

调用kset_register()将bus加入sysfs，因为只是设置了kset，所以会被加入/sys/bus目录下。与driver直接加入相关总线的drivers目录类似，却是与device复杂的寻找父节点过程相去甚远。

在bus目录下添加uevent属性。

在bus目录下创建devices子目录。它是一个kset类型的，目的是展示bus下的设备链表。

在bus目录下创建drivers子目录。它也是一个kset类型的，目的是展示bus下的驱动链表。

或许在最开始有设备驱动模型时，还需要kset来表达这种链表关系，但随着klist等结构的加入，kset的作用也越来越少，现在更多的作用是用来处理uevent消息。

之后初始化bus的设备链表和驱动链表，其中设备链表会占用设备的引用计数。

调用add_probe_files()在bus目录下添加probe相关的两个属性文件。

调用bus_add_attrs添加bus结构中添加的属性。

bus_register()中的操作出乎意料的简单。bus既不需要在哪里添加软链接，也不需要主动向谁报道，从来都是device和driver到bus这里报道的。所以bus_register()中只需要初始一下结构，添加到sysfs中，添加相关的子目录和属性文件，就行了。

```
1. void bus_unregister(struct bus_type *bus)
2. {
3.     pr_debug("bus: '%s': unregistering\n", bus->name);
4.     bus_remove_attrs(bus);
5.     remove_probe_files(bus);
6.     kset_unregister(bus->p->drivers_kset);
7.     kset_unregister(bus->p->devices_kset);
8.     bus_remove_file(bus, &bus_attr_uevent);
9.     kset_unregister(&bus->p->subsys);
10.    kfree(bus->p);
11.    bus->p = NULL;
12. }
```

bus_unregister()与bus_register()相对，将bus从系统中注销。不过要把bus注销也不是那么简单的，bus中的driver和device都对bus保有一份引用计数。或许正是如此，bus把释放bus->p的动作放在了bus_unregister()中，这至少能保证较早地释放不需要的内存空间。而且在bus引用计数用完时，也不会有任何操作，bus的容错性还是很高的。

```
1. static struct bus_type *bus_get(struct bus_type *bus)
2. {
3.     if (bus) {
4.         kset_get(&bus->p->subsys);
5.         return bus;
6.     }
7.     return NULL;
8. }
9.
10. static void bus_put(struct bus_type *bus)
11. {
12.     if (bus)
13.         kset_put(&bus->p->subsys);
14. }
```

bus_get()增加对bus的引用计数，bus_put()减少对bus的引用计数。实际上这里bus的引用计数降为零时，只是将sysfs中bus对应的目录删除。

无论是bus，还是device，还是driver，都是将主要的注销工作放在相关的unregister中。至于在引用计数降为零时的操作，大概只在device_release()中可见。这主要是因为引用计数，虽然是广泛用在设备驱动模型中，但实际支持的，绝大部分是设备的热插拔，而不是总线或者驱动的热插拔。当然，桥设备的热插拔也可能附带总线的热插拔。

```
1. /*
2.  * Yes, this forcably breaks the klist abstraction temporarily.  It
3.  * just wants to sort the klist, not change reference counts and
4.  * take/drop locks rapidly in the process.  It does all this while
5.  * holding the lock for the list, so objects can't otherwise be
6.  * added/removed while we're swizzling.
7.  */
8. static void device_insertion_sort_klist(struct device *a, struct list_head *list,
9.                  int (*compare)(const struct device *a,
10.                       const struct device *b))
11. {
12.     struct list_head *pos;
13.     struct klist_node *n;
14.     struct device_private *dev_prv;
15.     struct device *b;
16.
17.     list_for_each(pos, list) {
18.         n = container_of(pos, struct klist_node, n_node);
19.         dev_prv = to_device_private_bus(n);
20.         b = dev_prv->device;
21.         if (compare(a, b) <= 0) {
22.             list_move_tail(&a->p->knode_bus.n_node,
23.                        &b->p->knode_bus.n_node);
24.             return;
25.         }
26.     }
27.     list_move_tail(&a->p->knode_bus.n_node, list);
28. }
29.
30. void bus_sort_breadthfirst(struct bus_type *bus,
31.              int (*compare)(const struct device *a,
```

```
32.                 const struct device *b))
33. {
34.     LIST_HEAD(sorted_devices);
35.     struct list_head *pos, *tmp;
36.     struct klist_node *n;
37.     struct device_private *dev_prv;
38.     struct device *dev;
39.     struct klist *device_klist;
40.
41.     device_klist = bus_get_device_klist(bus);
42.
43.     spin_lock(&device_klist->k_lock);
44.     list_for_each_safe(pos, tmp, &device_klist->k_list) {
45.         n = container_of(pos, struct klist_node, n_node);
46.         dev_prv = to_device_private_bus(n);
47.         dev = dev_prv->device;
48.         device_insertion_sort_klist(dev, &sorted_devices, compare);
49.     }
50.     list_splice(&sorted_devices, &device_klist->k_list);
51.     spin_unlock(&device_klist->k_lock);
52. }
```

bus_sort_breadthfirst()是将bus的设备链表进行排序，使用指定的比较函数，排成降序。

本节主要分析了bus的注册注销过程，下节我们将深入分析device和driver的绑定过程，了解bus在这其中到底起了什么作用。随着我们了解的逐渐深入，未知的东西也在逐渐增多。但饭要一口一口吃，我们的分析也要一点一点来，急不得。

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