Linux 那些事儿

系列丛书

之

我是 Sysfs

[1] 原文为blog.csdn.net/fudan_abc 上的《linux 那些事儿之我是Sysfs》，有闲情逸致的或者有批评建议的可以到上面做客，也可以email 到ilttv.cn@gmail.com

目录

引子

看到复旦人甲热火朝天的写作，我心底不禁暗暗敬佩。话说我进入 linux 领域也有 2 年了，我学习 linux 完全是兴趣。因为我觉得用 linux 比较酷，比较吊。当年看过一篇捧 linux 大骂 windows 的文章，看了以后我是热血沸腾，当天就把 windows 给删了。后来发现实在不习惯，因为没法斗地主了。无奈之下，只好又装上了 windows。当时我就有一个愿望，我一定要打入腾讯内部，响应毛主席号召，让全世界 linux 用户也能斗上地主。大学生活，丰富多彩，我只有一个爱好，打篮球，时间充裕，一有时间就看看 linux 相关书籍。但总是静不下心来，每次看到一定阶段就放下了。等隔了一段时间，不看不用，就忘了，只是大概知道有这么一回事。现在总结学习 linux 的关键是，多用多看，持之以恒。我也把自己看 linux 内核的心得写下来响应复旦人甲的号召，与其去打游戏，看片，灌 BBS，还不如静下心来，踏踏实实看点书，写点文章。现在社会风气浮躁，就缺肯踏踏实实干事情的人。

我主要写一些文件系统相关的，结合 ldd3(linux device driver 3)中的示例代码(主要是 lddbus 和 sculld 文件夹)，分析虚拟文件系统 sysfs。使用的内核是 2.6.10。

目标：理解设备模型和 sysfs 文件系统

欢迎各位朋友发邮件指正，讨论，交流。

linux.fans.afu@gmail.com

知行合一，王阳明如是说。

Sysfs 初探

"sysfs is a ram-based filesystem initially based on ramfs. It provides a means
to export kernel data structures, their attributes, and the linkages between them to
userspace." --- documentation/filesystems/sysfs.txt
可以先把 documentation/filesystems/sysfs.txt 读一遍。文档这种东西，真正读起来就嫌少了。Sysfs 文件系统是一个类似于 proc 文件系统的特殊文件系统，

用于将系统中的设备组织成层次结构，并向用户模式程序提供详细的内核数据结构信息。

去/sys 看一看，
localhost:/sys#ls /sys/
block/ bus/ class/ devices/ firmware/ kernel/ module/ power/
Block 目录：包含所有的块设备
Devices 目录：包含系统所有的设备，并根据设备挂接的总线类型组织成层次结构
Bus 目录：包含系统中所有的总线类型
Drivers 目录：包括内核中所有已注册的设备驱动程序
Class 目录：系统中的设备类型（如网卡设备，声卡设备等）

sys 下面的目录和文件反映了整台机器的系统状况。比如 bus，
localhost:/sys/bus#ls
i2c/ ide/ pci/ pci express/ platform/ pnp/ scsi/ serio/ usb/
里面就包含了系统用到的一系列总线，比如 pci, ide, scsi, usb 等等。比如你可以在 usb 文件夹中发现你使用的 U 盘，USB 鼠标的信息。

我们要讨论一个文件系统，首先要知道这个文件系统的信息来源在哪里。所谓信息来源是指文件组织存放的地点。比如，我们挂载一个分区，

mount -t vfat /dev/hda2 /mnt/C

我们就知道挂载在/mnt/C 下的是一个 vfat 类型的文件系统，它的信息来源是在第一块硬盘的第 2 个分区。

但是，你可能根本没有去关心过 sysfs 的挂载过程，她是这样被挂载的。

mount -t sysfs sysfs /sys

ms 看不出她的信息来源在哪。sysfs 是一个特殊文件系统，并没有一个实际存放文件的介质。断电后就玩完了。简而言之，sysfs 的信息来源是 kobject 层次结构，读一个 sysfs 文件，就是动态的从 kobject 结构提取信息，生成文件。

所以，首先，我要先讲一讲 sysfs 文件系统的信息来源 -- kobject 层次结构。kobject 层次结构就是 linux 的设备模型。

莫愁前路无知己，天下谁人不识君。 唐·高适· 别董大

Linus 设备底层模型

关于 linux 设备模型网上有一些论述，有些东西我就用了拿来主义，进行了修改和整理。

§1 Kobject

Kobject 是 Linux 2.6 引入的新的设备管理机制，在内核中由 struct kobject 表示。通过这个数据结构使所有设备在底层都具有统一的接口，kobject 提供基本的对象管理，是构成 Linux2.6 设备模型的核心结构，它与 sysfs 文件系统紧密关联，每个在内核中注册的 kobject 对象都对应于 sysfs 文件系统中的一个目录。Kobject 是组成设备模型的基本结构。类似于 C++中的基类，它嵌入于更大的对象的对象中--所谓的容器--用来描述设备模型的组件。如 bus,devices, drivers 都是典型的容器。这些容器就是通过 kobject 连接起来了，形成了一个树状结构。这个树状结构就与/sys 向对应。

kobject 结构为一些大的数据结构和子系统提供了基本的对象管理，避免了类似机能的重复实现。这些机能包括

- 对象引用计数.
- 维护对象链表(集合).
- 对象上锁.
- 在用户空间的表示.

Kobject 结构定义为：

```
struct kobject {
char * k name; 指向设备名称的指针
char name[KOBJ NAME LEN]; 设备名称
struct kref kref; 对象引用计数
struct list head entry; 挂接到所在 kset 中去的单元
struct kobject * parent; 指向父对象的指针
struct kset * kset; 所属 kset 的指针
struct kobj type * ktype; 指向其对象类型描述符的指针
struct dentry * dentry; sysfs 文件系统中与该对象对应的文件节点路径指针
};
```

其中的 kref 域表示该对象引用的计数，内核通过 kref 实现对象引用计数管理，内核提供两个函数 kobject_get()、kobject_put()分别用于增加和减少引用计数，当引用计数为 0 时，所有该对象使用的资源释放。Ktype 域是一个指向 kobj type 结构的指针，表示该对象的类型。

相关函数

void kobject_init(struct kobject * kobj); kobject 初始化函数。

int kobject_set_name(struct kobject *kobj, const char *format, ...); 设置指定 kobject 的名称。

struct kobject *kobject_get(struct kobject *kobj); 将 kobj 对象的引用计数加 1，同时返回该对象的指针。

void kobject_put(struct kobject * kobj); 将 kobj 对象的引用计数减 1，

如果引用计数降为 0，则调用 kobject release()释放该 kobject 对象。
int kobject_add(struct kobject * kobj)；将 kobj 对象加入 Linux 设备层次。挂接该 kobject 对象到 kset 的 list 链中,增加父目录各级 kobject 的引用计数，在其 parent 指向的目录下创建文件节点，并启动该类型内核对象的 hotplug 函数。
int kobject_register(struct kobject * kobj)；kobject 注册函数。通过调用 kobject init()初始化 kobj，再调用 kobject_add()完成该内核对象的注册。
void kobject_del(struct kobject * kobj)；从 Linux 设备层次(hierarchy)中删除 kobj 对象。
void kobject_unregister(struct kobject * kobj)；kobject 注销函数。与 kobject register()相反，它首先调用 kobject del 从设备层次中删除该对象，再调用 kobject put()减少该对象的引用计数，如果引用计数降为 0，则释放 kobject 对象。

§2 Kobj type

```
struct kobj_type {
void (*release)(struct kobject *);
struct sysfs_ops * sysfs_ops;
struct attribute ** default_attrs;
};
```

Kobj type 数据结构包含三个域：一个 release 方法用于释放 kobject 占用的资源;一个 sysfs ops 指针指向 sysfs 操作表和一个 sysfs 文件系统缺省属性列表。Sysfs 操作表包括两个函数 store()和 show()。当用户态读取属性时，show()函数被调用，该函数编码指定属性值存入 buffer 中返回给用户态；而 store()函数用于存储用户态传入的属性值。
attribute

```
struct attribute {
char * name;
struct module * owner;
mode_t mode;
};
```

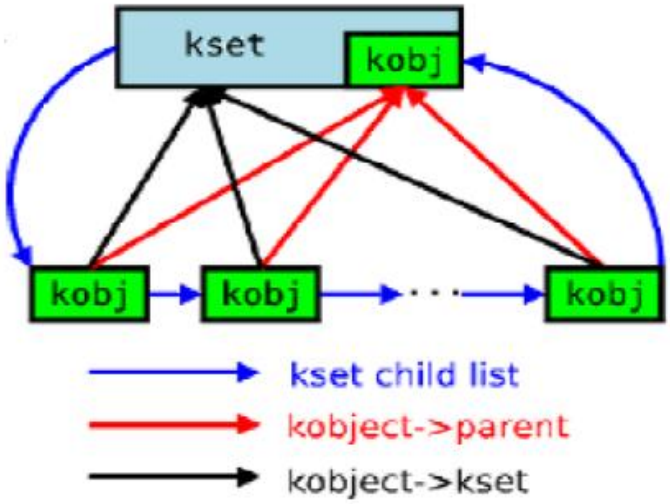
attribute, 属性。它以文件的形式输出到 sysfs 的目录当中。在 kobject 对应的目录下面。文件
名就是 name。文件读写的方法对应于 kobj type 中的 sysfs ops。

§3. kset

kset 最重要的是建立上层(sub-system)和下层的(kobject)的关联性。kobject 也会利用它了分辨自已是属于那一個类型,然後在/sys 下建立正确的目录位置。而 kset 的优先权比较高，kobject 会利用自已的*kset 找到自已所属的 kset，並把*ktype 指定成該 kset 下的 ktype，除非沒有定义 kset，才会用 ktype 來建立关系。Kobject 通过 kset 组织成层次化的结构，kset 是具有相同类型的 kobject 的集合，在内核中用 kset 数据结构表示，定义为：

```
struct kset {
```

struct subsystem * subsys; 所在的 subsystem 的指针
struct kobj type * ktype; 指向该 kset 对象类型描述符的指针
struct list head list; 用于连接该 kset 中所有 kobject 的链表头
struct kobject kobj; 嵌入的 kobject
struct kset hotplug ops * hotplug ops; 指向热插拔操作表的指针
};
包含在 kset 中的所有 kobject 被组织成一个双向循环链表，list 域正是该链表的头。Ktype 域指向一个 kobj type 结构，被该 kset 中的所有 kobject 共享，表示这些对象的类型。Kset 数据结构还内嵌了一个 kobject 对象（由 kobj 域表示），所有属于这个 kset 的 kobject 对象的 parent 域均指向这个内嵌的对象。此外，kset 还依赖于 kobj 维护引用计数：kset 的引用计数实际上就是内嵌的 kobject 对象的引用计数。
见图 1，kset 与 kobject 的关系图



这幅图很经典，她反映了整个 kobject 的连接情况。

相关函数
与 kobject 相似,kset_init()完成指定 kset 的初始化,kset_get()和 kset_put()分别增加和减少 kset 对象的引用计数。Kset_add()和 kset_del()函数分别实现将指定 keset 对象加入设备层次和从其中删除；kset_register()函数完成 kset 的注册而 kset_unregister()函数则完成 kset 的注销。

§4 subsystem

如果說 kset 是管理 kobject 的集合，同理，subsystem 就是管理 kset 的集合。它描述系统中某一类设备子系统，如 block subsys 表示所有的块设备，对应于 sysfs 文件系统中的 block 目录。类似的，devices subsys 对应于 sysfs 中的 devices 目录，描述系统中所有的设备。Subsystem 由 struct subsystem 数据结构描述，定义为：
struct subsystem {

```
struct kset kset; 内嵌的 kset 对象
struct rw semaphore rwsem; 互斥访问信号量
};
```

可以看出，subsystem 与 kset 的区别就是多了一个信号量，所以在后来的代码中，subsystem 已经完全被 kset 取缔了。

每个 kset 属于某个 subsystem，通过设置 kset 结构中的 subsys 域指向指定的 subsystem 可以将一个 kset 加入到该 subsystem。所有挂接到同一 subsystem 的 kset 共享同一个 rwsem 信号量，用于同步访问 kset 中的链表。
相关函数
subsystem 有一组类似的函数，分别是：

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

关于那些函数的用法，会在后面的举例中详细讲。这里仅仅是一个介绍。

设备模型上层容器

§1 bus

系统中总线由struct bus_type描述，定义为：

```
struct bus_type {
char * name; 总线类型的名称
struct subsystem subsys; 与该总线相关的subsystem
struct kset drivers; 所有与该总线相关的驱动程序集合
struct kset devices; 所有挂接在该总线上的设备集合
struct bus attribute * bus_attrs; 总线属性
struct device attribute * dev_attrs; 设备属性
struct driver attribute * drv_attrs; 驱动程序属性
int (*match)(struct device * dev, struct device_driver * drv);
int (*hotplug) (struct device *dev, char **envp, int num_envp, char
*buffer, int buffer_size);
int (*suspend)(struct device * dev, u32 state);
int (*resume)(struct device * dev);
};
```

每个bus_type对象都内嵌一个subsystem对象，bus_subsys对象管理系统中所有总线类型的subsystem对象。每个bus_type对象都对应/sys/bus目录下的一个子目录，如PCI总线类型对应于/sys/bus/pci。在每个这样的目录下都存在两个子目录：devices和drivers（分别对应于bus type结构中的devices和drivers域）。其中devices子目录描述连接在该总线上的所有设备，而drivers目录则描述与该

总线关联的所有驱动程序。与device_driver对象类似，bus_type结构还包含几个函数（match()、hotplug()等）处理相应的热插拔、即插即拔和电源管理事件。

§2 device

系统中的任一设备在设备模型中都由一个device对象描述，其对应的数据结构struct device
定义为：

```
struct device {
struct list_head g_list;
struct list_head node;
struct list_head bus_list;
struct list_head driver_list;
struct list_head children;
struct device *parent;
struct kobject kobj;
char bus_id[BUS_ID_SIZE];
struct bus_type *bus;
struct device_driver *driver;
void *driver_data;
/* Several fields omitted */
};
```

g_list 将该device对象挂接到全局设备链表中，所有的device对象都包含在devices subsys中，并组织成层次结构。Node域将该对象挂接到其兄弟对象的链表中，而bus list则用于将连接到相同总线上的设备组织成链表，driver list则将同一驱动程序管理的所有设备组织为链表。此外，children域指向该device对象子对象链表头，parent域则指向父对象。Device对象还内嵌一个kobject对象，用于引用计数管理并通过它实现设备层次结构。Driver域指向管理该设备的驱动程序对象，而driver data则是提供给驱动程序的数据。Bus域描述设备所连接的
总线类型。

内核提供了相应的函数用于操作 device 对象。其中 device_register()函数将一个新的 device 对象插入设备模型，并自动在/sys/devices 下创建一个对应的目录。device_unregister()完成相反的操作，注销设备对象。get_device()和 put_device()分别增加与减少设备对象的引用计数。通常 device 结构不单独使用，而是包含在更大的结构中作为一个子结构使用，比如描述 PCI 设备的 struct pci_dev，还有我们 ldd_dev,其中的 dev 域就是一个 device 对象。

§3. driver

系统中的每个驱动程序由一个device_driver对象描述，对应的数据结构定义为：

```
struct device_driver {
char *name; 设备驱动程序的名称
```

struct bus_type *bus; 该驱动所管理的设备挂接的总线类型
struct kobject kobj; 内嵌kobject对象
struct list_head devices; 该驱动所管理的设备链表头
int (*probe)(struct device *dev); 指向设备探测函数，用于探测设备是否可以被该驱动程序管理
int (*remove)(struct device *dev); 用于删除设备的函数
/* some fields omitted*/
};
与device 结构类似，device_driver对象依靠内嵌的kobject对象实现引用计数管理和层次结构组织。内核提供类似的函数用于操作device_driver对象，如get_driver()增加引用计数，driver_register()用于向设备模型插入新的driver对象，同时在sysfs文件系统中创建对应的目录。device_driver()结构还包括几个函数，用于处理热拔插、即插即用和电源管理事件。

可能你面对刚刚列举出来的一些列数据结构，感到很苦恼，很莫名其妙。没关系，我接下来讲个例子您就明白了。

举例一 lddbus

对了，你得把 ldd3 的 examples 代码下下来。不然没法继续了。

接下来我们从例子着手，
localhost:/home/XX/examples/lddbus#insmod lddbus.ko
此时再看/sys/bus/ 这时就多了一个文件夹 ldd。里面的文件构成是这样的
/sys/bus/ldd/
|--device
|--driver
`--version
localhost:/sys/bus/ldd#cat version
$Revision: 1.9$

这表示系统中多了一种名叫 ldd 的总线类型。同时再看/sys/device/，也多出来一个 ldd0 的文件夹。这表示系统中多了一个名叫 ldd0 的硬件。
在 lddbus.c 中， 定义了一个总线和硬件类型

```
struct bus_type ldd_bus_type = {
        .name = "ldd",
        .match = ldd_match,
        .hotplug  = ldd_hotplug,
};

struct device ldd_bus = {
        .bus_id   = "ldd0",
```

```
        .release  = ldd_bus_release
};
```

lddbus 模块初始化时调用这个函数

```
static int __init ldd_bus_init(void)
{
        int ret;

        ret = bus_register(&ldd_bus_type);
        if (ret)
                return ret;
        if (bus_create_file(&ldd_bus_type, &bus_attr_version))
                printk(KERN_NOTICE "Unable to create version attribute\n");
        ret = device_register(&ldd_bus);
        if (ret)
                printk(KERN_NOTICE "Unable to register ldd0\n");
        return ret;
}
```

其实就是调用了两个注册函数，bus_register(), device_register()。bus_create_file()是在 sysfs 下创建一个文件夹。

bus_register(),向系统注册 ldd_bus_type 这个总线类型。bus_create_file()这个就是向 sysfs 中创建一个文件。device_register()系统注册 ldd_bus 这个硬件类型。
注册好了之后，我们就可以在 sysfs 下看到相应的信息。

我们深入下去，仔细看看 bus_register 的代码。

```
688 int bus_register(struct bus_type * bus)
689 {
690         int retval;
691
692         retval = kobject_set_name(&bus->subsys.kset.kobj, "%s", bus->name);
693         if (retval)
694                 goto out;
695
696         subsys_set_kset(bus, bus_subsys);
697         retval = subsystem_register(&bus->subsys);
698         if (retval)
699                 goto out;
700
701         kobject_set_name(&bus->devices.kobj, "devices");
702         bus->devices.subsys = &bus->subsys;
703         retval = kset_register(&bus->devices);
704         if (retval)
705                 goto bus_devices_fail;
706
```

```
707         kobject_set_name(&bus->drivers.kobj, "drivers");
708         bus->drivers.subsys = &bus->subsys;
709         bus->drivers.ktype = &ktype_driver;
710         retval = kset_register(&bus->drivers);
711         if (retval)
712                 goto bus_drivers_fail;
713         bus_add_attrs(bus);
714
715         pr_debug("bus type '%s' registered\n", bus->name);
716         return 0;
717
718 bus_drivers_fail:
719         kset_unregister(&bus->devices);
720 bus_devices_fail:
721         subsystem_unregister(&bus->subsys);
722 out:
723         return retval;
724 }
```

692-700 是对 bus->subsys 的操作。701-705 是操作 bus->devices。706-710 是操作 bus->drivers。

692 kobject_set_name()设置 bus->subsys.kset.kobj 的名字。此函数很简单，就是调用 vsnprintf()。此不列出。

696 subsys_set_kset(bus, bus subsys)

```
#define subsys_set_kset(obj,_subsys)   (obj)->subsys.kset.kobj.kset = &(_subsys).kset
```

我们先看看 bus_subsys 的定义，它是一个 subsystem 类型的全局变量。在 driver/base/bus.c 中，decl subsys(bus, &ktype bus, NULL); 在/include/linux/kobject.h 中有，decl subsys 的原型，

```
#define decl_subsys(_name,_type,_hotplug_ops) \
struct subsystem _name##_subsys = { \
.kset = { \
.kobj = { .name = __stringify(_name) }, \
.ktype = _type, \
.hotplug_ops =_hotplug_ops, \
} \
}
```

就相当于

```
struct subsystem bus_subsys = { \
.kset = { \
.kobj = { .name = “bus” }, \
.ktype = ktype_bus, \
.hotplug_ops =NULL, \
} \
}
```

其中 ktype bus 定义如下，

```
static struct kobj_type ktype_bus = {
.sysfs_ops = &bus_sysfs_ops,
};
```

697 subsystem_register(&bus->subsys)作用是向全局的 bus_subsys”登记”，把自己加入到 bus_subsys 的链表中去。

subsystem_register() -> kset_add() -> kobject_add()

```
155 int kobject_add(struct kobject * kobj)
156 {
157         int error = 0;
158         struct kobject * parent;
159
160         if (!(kobj = kobject_get(kobj)))
161                 return -ENOENT;
162         if (!kobj->k_name)
163                 kobj->k_name = kobj->name;
164         parent = kobject_get(kobj->parent);
165
166         pr_debug("kobject %s: registering. parent: %s, set: %s\n",
167                  kobject_name(kobj), parent ? kobject_name(parent) : "<NULL>",
168                  kobj->kset ? kobj->kset->kobj.name : "<NULL>" );
169
170         if (kobj->kset) {
171                 down_write(&kobj->kset->subsys->rwsem);
172
173                 if (!parent)
174                         parent = kobject_get(&kobj->kset->kobj);
175
176                 list_add_tail(&kobj->entry,&kobj->kset->list);
177                 up_write(&kobj->kset->subsys->rwsem);
178         }
179         kobj->parent = parent;
180
181         error = create_dir(kobj);
182         if (error) {
183                 /* unlink does the kobject_put() for us */
184                 unlink(kobj);
185                 if (parent)
186                         kobject_put(parent);
187         } else {
188                 kobject_hotplug(kobj, KOBJ_ADD);
189         }
190
```

```
191         return error;
192 }
```

代码的 170-178 就是把自己连入到父辈上级 kset 中。我们注意到在 kobject_add()函数中 181 行调用了 create_dir(kobj),这个函数作用是在 sysfs 下创建一个文件夹。可见 kobject 和 sysfs 是同时更新的。

kset_register(&bus->devices) 和 kset_register(&bus->drivers)作用类似,把 bus->devices 这个 kset 加入到 bus->subsys 这个 subsystem 中去。最后形成图 1 的层次结构。

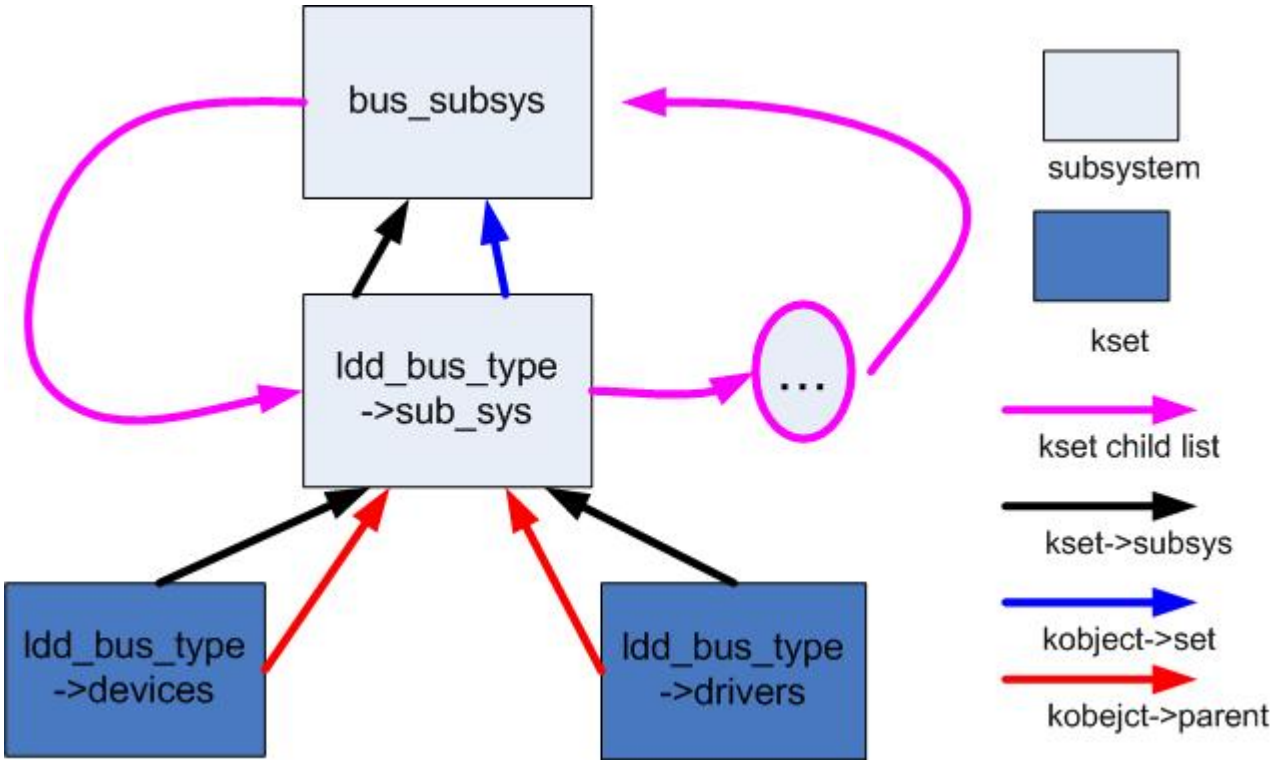


图 1：lddbus kobject 层次结构

同理，我们可以看看 device_register()的代码，它也是向 devices_subsys 这个 subsystem 注册，最后形成这样的结构与图 1 类似。

目前为止，我们知道了所谓的 xx_register 函数，就是通过其内嵌的 kobject 链入对应的 subsystem，或是 kset 的层次结构中去。这样就可以通过一些全局的变量找到它们了。

举例二 sculld

不妨再把 sculld 的代码也分析一下，先看初始函数
sculld_init()

```
-> register_ldd_driver()
        ->driver_register()
                ->bus_add_driver()
-> register_ldd_device()
        ->device_register()
                ->device_add()
                        ->kobject_add()
                        ->bus_add_device()
```

首先注册驱动，看 bus_add_driver()

```
532 int bus_add_driver(struct device_driver * drv)
533 {
534       struct bus_type * bus = get_bus(drv->bus);
535       int error = 0;
536
537       if (bus) {
538             pr_debug("bus %s: add driver %s\n", bus->name, drv->name);
539             error = kobject_set_name(&drv->kobj, "%s", drv->name);
540             if (error) {
541                   put_bus(bus);
542                   return error;
543             }
544             drv->kobj.kset = &bus->drivers;
545             if ((error = kobject_register(&drv->kobj))) {
546                   put_bus(bus);
547                   return error;
548             }
549
550             down_write(&bus->subsys.rwsem);
551             driver_attach(drv);
552             up_write(&bus->subsys.rwsem);
553             module_add_driver(drv->owner, drv);
554
555             driver_add_attrs(bus, drv);
556       }
557       return error;
558 }
559
```

545 行 kobject_register()与 kobject_add()差不多，进行注册，把自己 kobject 链接到内核中去。

551，driver_attach(drv)；在总线中寻找，有没有设备可以让这个 driver 驱动。

```
353 void driver_attach(struct device_driver * drv)
354 {
```

```
355       struct bus_type * bus = drv->bus;
356       struct list_head * entry;
357       int error;
358
359       if (!bus->match)
360             return;
361
362       list_for_each(entry, &bus->devices.list) {
363             struct device * dev = container_of(entry, struct device, bus_list);
364             if (!dev->driver) {
365                   error = driver_probe_device(drv, dev);
366                   if (error && (error != -ENODEV))
367                         /* driver matched but the probe failed */
368                         printk(KERN_WARNING
369                            "%s: probe of %s failed with error %d\n",
370                            drv->name, dev->bus_id, error);
371             }
372       }
373 }
```

然后注册设备，

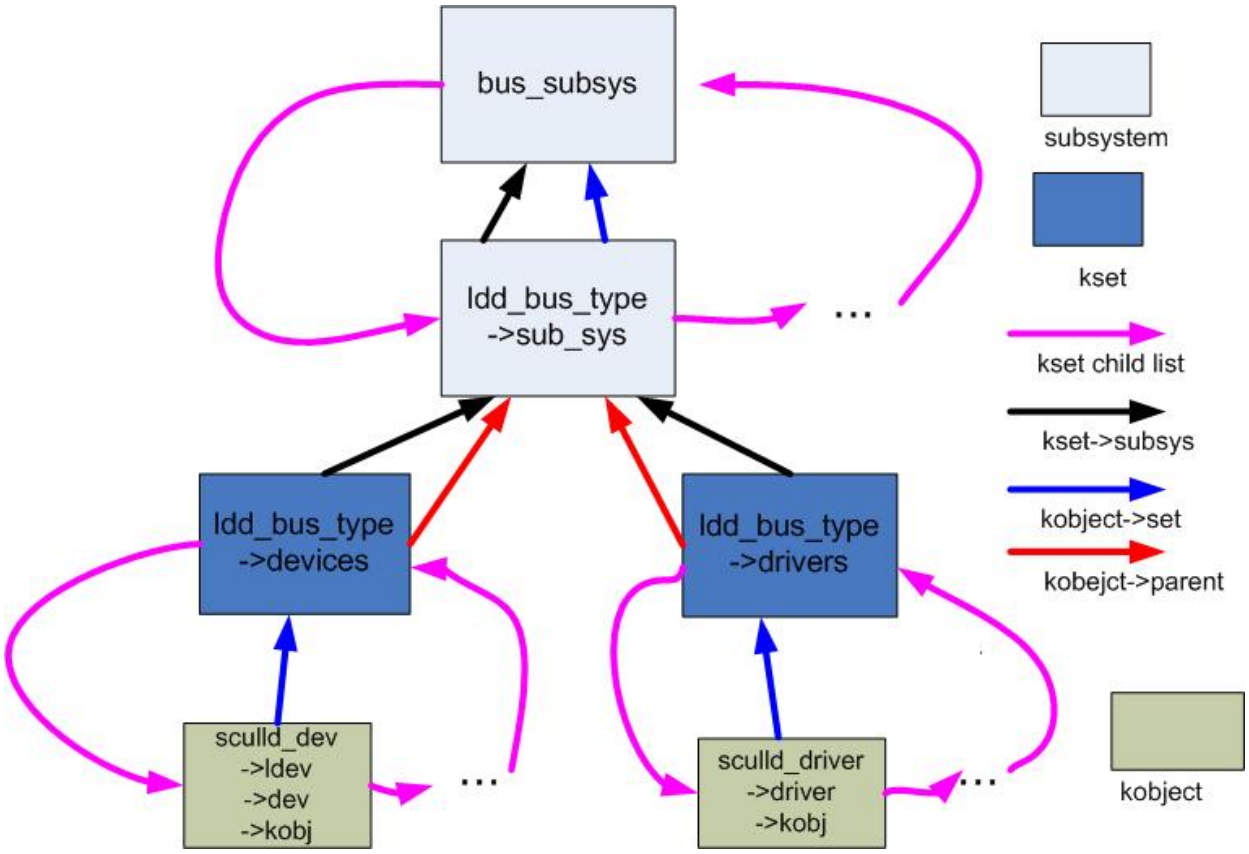
```
455 int bus_add_device(struct device * dev)
456 {
457       struct bus_type * bus = get_bus(dev->bus);
458       int error = 0;
459
460       if (bus) {
461             down_write(&dev->bus->subsys.rwsem);
462             pr_debug("bus %s: add device %s\n", bus->name,
dev->bus_id);
463             list_add_tail(&dev->bus_list, &dev->bus->devices.list);
464               465             up_write(&dev->bus->subsys.rwsem);
466             device_add_attrs(bus, dev);
467             sysfs_create_link(&bus->devices.kobj, &dev->kobj,
dev->bus_id);
468       }
469       return error;
470 }
```

463，把设备连入其总线的 devices.list 链表中。

464，device_attach(dev)与 driver_attach()相对应，它在总线的驱动中寻找，看有没有一个 driver 能驱动这个设备。

467，创建了一个链接。

最后形成的 kobject 层次结构如图所示。



变化

计划赶不上变化，当前的内核版本已经是 2.6.22 了，其中不少数据结构作了变动，而且 subsystem 这个数据结构已经没有了，完全被 kset 取缔了。但是原理上并没有变，我认为，学习知识是一方面，更重要的是学习方法。只要懂了方法，我们才可"以不变应万变"。

17 大马上要召开了，刘翔又夺冠了，奥尔默特与阿巴斯也会面了，明年就奥运了。和谐的社会里充满着希望与绝望。不管怎样，终于把设备模型介绍完毕，接下来进入 sysfs 部分。

人面不知何处去，桃花依旧笑春风。 唐·崔护·题都城南庄

文件系统

接下来，我们进入 sysfs 部分。看看

kobject_add()->create_dir()->sysfs_create_dir()

bus create file->sysfs create file()

. . .

这些 sysfs 函数的内幕。

说白了，sysfs 就是利用 VFS 的接口去读写 kobject 的层次结构，建立起来的文件系统。关于

sysfs 的内容就在 fs/sysfs/下。 kobject 的层次结构的更新与删除就是那些乱七八糟的 XX_register()们干的事情。

在 kobject_add()里面，调用了 sysfs_create_dir()。让我们看看它究竟是如何 create 的。

```
135 int sysfs_create_dir(struct kobject * kobj)
136 {
137         struct dentry * dentry = NULL;
138         struct dentry * parent;
139         int error = 0;
140
141         BUG_ON(!kobj);
142
143         if (kobj->parent)
144                 parent = kobj->parent->dentry;
145         else if (sysfs_mount && sysfs_mount->mnt_sb)
146                 parent = sysfs_mount->mnt_sb->s_root;
147         else
148                 return -EFAULT;
149
150         error = create_dir(kobj,parent,kobject_name(kobj),&dentry);
151         if (!error)
152                 kobj->dentry = dentry;
153         return error;
154 }
```

当你看见这么些新东西，如 dentry 出现的时候，你一定感到很困惑。诚然，我一度为代码中突然出现的事物感到恐慌，人类对未知的恐惧是与生俱来的，面对死亡，面对怪力乱神，我们抱着一颗敬畏的心灵就可以了。而面对 linux，我们始终坚信，未知肯定是可以被探索出来的。妖是妖他妈生的，代码是人他妈写出来的，既然写得出来，那就肯定看得懂。对不起，扯远了....我还是介绍介绍文件系统的基本知识先。

文件系统

文件系统是个很模糊广泛的概念，"文件"狭义地说，是指磁盘文件，广义理解，可以是有组织有次序地存储与任何介质(包括内存)的一组信息。linux 把所有的资源都看成是文件，让用户通过一个统一的文件系统操作界面，也就是同一组系统调用，对属于不同文件系统的文件进行操作。这样，就可以对用户程序隐藏各种不同文件系统的实现细节，为用户程序提供了一个统一的，抽象的，虚拟的文件系统界面，这就是所谓"VFS(Virtual Filesystem Switch)"。这个抽象出来的接口就是一组函数操作。

我们要实现一种文件系统就是要实现 VFS 所定义的一系列接口，file_operations, dentry_operations, inode_operations 等，供上层调用。file_operations 是对每个具体文件的读写操作，dentry_operations, inode_operations 则是对文件的属性，如改名字，建立或删除的操作。

```
struct file_operations {
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
int (*open) (struct inode *, struct file *);
...
};
struct dentry_operations {
...
};
struct inode_operations {
int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
int (*link) (struct dentry *,struct inode *,struct dentry *);
int (*mkdir) (struct inode *,struct dentry *,int);
int (*rmdir) (struct inode *,struct dentry *);
...}
```

举个例子，我们写 C 程序，open(“hello.c”, O_RDONLY)，它通过系统调用的流程是这样的

```
open() ->      /*用户空间*/
-> 系统调用->
sys_open() -> filp_open()-> dentry_open() ->  file_operations->open()        /*
内核空间*/
```

不同的文件系统，调用不同的 file_operations->open()，在 sysfs 下就是 sysfs_open_file()。

Dentry 与 inode

我们在进程中要怎样去描述一个文件呢？我们用目录项(dentry)和索引节点(inode)。它们的定义如下:

```
struct dentry {
struct inode                                *d_inode; /* Where the name belongs to -
NULL is
struct dentry                                *d_parent; /* parent directory */
struct list_head                           d_child; /* child of parent list */
struct dentry_operations          *d_op;
struct super_block                     *d_sb; /* The root of the dentry tree */
void                                          *d_fsdata; /* fs-specific data */
unsigned char                   d_iname[DNAME_INLINE_LEN_MIN]; /* small names */
......
};
```

```
struct inode {
unsigned long                    i_ino;
atomic_t                          i_count;
umode_t                          i_mode;
unsigned int                     i_nlink;
uid_t                                 i_uid;
gid_t                                 i_gid;
dev_t                                i_rdev;
loff_t                                 i_size;
struct timespec                  i_atime;
unsigned long                    i_blocks;
unsigned short                   i_bytes;
unsigned char                     _sock;
12
struct inode_operations *i_op;
struct file_operations *i_fop; /* former ->i_op->default_file_ops */
struct super_block *i_sb;
......
};
```

所谓"文件", 就是按一定的形式存储在介质上的信息,所以一个文件其实包含了两方面的信息，一是存储的数据本身，二是有关该文件的组织和管理的信息。在内存中, 每个文件都有一个dentry(目录项)和inode(索引节点)结构，dentry记录着文件名，上级目录等信息，正是它形成了我们所看到的树状结构；而有关该文件的组织和管理的信息主要存放inode里面，它记录着文件在存储介质上的位置与分布。同时dentry->d_inode指向相应的inode结构。dentry与inode是多对一的关系，因为有可能一个文件有好几个文件名(硬链接, hard link, 可以参考这个网页 http://www.ugrad.cs.ubc.ca/~cs219/CourseNotes/Unix/commands-links.html)。

所有的 dentry 用 d_parent 和 d_child 连接起来，就形成了我们熟悉的树状结构。

inode 代表的是物理意义上的文件，通过 inode 可以得到一个数组，这个数组记录了文件内容的位置，如该文件位于硬盘的第 3，8，10 块，那么这个数组的内容就是 3,8,10。其索引节点号 inode->i_ino，在同一个文件系统中是唯一的，内核只要根据 i_ino，就可以计算出它对应的 inode 在介质上的位置。就硬盘来说，根据 i_ino 就可以计算出它对应的 inode 属于哪个块(block)，从而找到相应的 inode 结构。但仅仅用 inode 还是无法描述出所有的文件系统，对于某一种特定的文件系统而言,比如 ext3,在内存中用 ext3_inode_info 描述。他是一个包含 inode 的"容器"。

```
struct ext3_inode_info {
        __le32 i_data[15];
        ......
        struct inode vfs_inode;
};
```

le32 i data[15]这个数组就是上一段中所提到的那个数组。

注意，在遥远的 2.4 的古代，不同文件系统索引节点的内存映像(ext3_inode_info，reiserfs_inode_info，msdos_inode_info ...)都是用一个 union 内嵌在 inode 数据结构中的. 但 inode 作为一种非常基本的数据结构而言，这样搞太大了，不利于快速的分配和回收。但是后来发明了 container_of(...)这种方法后，就把 union 移到了外部，我们可以用类似 container of(inode, struct ext3_inode_info, vfs_inode)，从 inode 出发，得到其的"容器"。

dentry 和 inode 终究都是在内存中的，它们的原始信息必须要有一个载体。否则断电之后岂不是玩完了？且听我慢慢道来。

文件可以分为磁盘文件，设备文件，和特殊文件三种。设备文件暂且不表。

磁盘文件

就磁盘文件而言，dentry和inode的载体在存储介质(磁盘)上。对于像ext3 这样的磁盘文件来说，存储介质中的目录项和索引节点载体如下，

```
struct ext3_inode {
__le16 i_mode; /* File mode */
__le16 i_uid; /* Low 16 bits of Owner Uid */
__le32 i_size; /* Size in bytes */
__le32 i_atime; /* Access time */
__le32 i_ctime; /* Creation time */
__le32 i_mtime; /* Modification time */

__le32 i_dtime; /* Deletion Time */
__le16 i_gid; /* Low 16 bits of Group Id */
__le16 i_links_count; /* Links count */
......
__le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
......
}
struct ext3_dir_entry_2 {
__u32 inode; /* Inode number */
__u16 rec_len; /* Directory entry length */
__u8 name_len; /* Name length */
__u8 file_type;
char name[EXT3_NAME_LEN]; /* File name */
};

le32 i block[EXT2 N BLOCKS];/* Pointers to blocks */
```

i_block 数组指示了文件的内容所存放的地点(在硬盘上的位置)。

ext3_inode 是放在索引节点区，而 ext3_dir_entry_2 是以文件内容的形式存放在数据区。我们只要知道了 ino，由于 ext3_inode 大小已知，我们就可以计算出 ext3_inode 在索引节点区的位置(ino * sizeof(ext3_inode))，而得到了 ext3_inode，我们根据 i_block 就可以知道这个文件的数据存放的地点。将磁盘上 ext3_inode 的内容读入到 ext3_inode_info 中的函数是

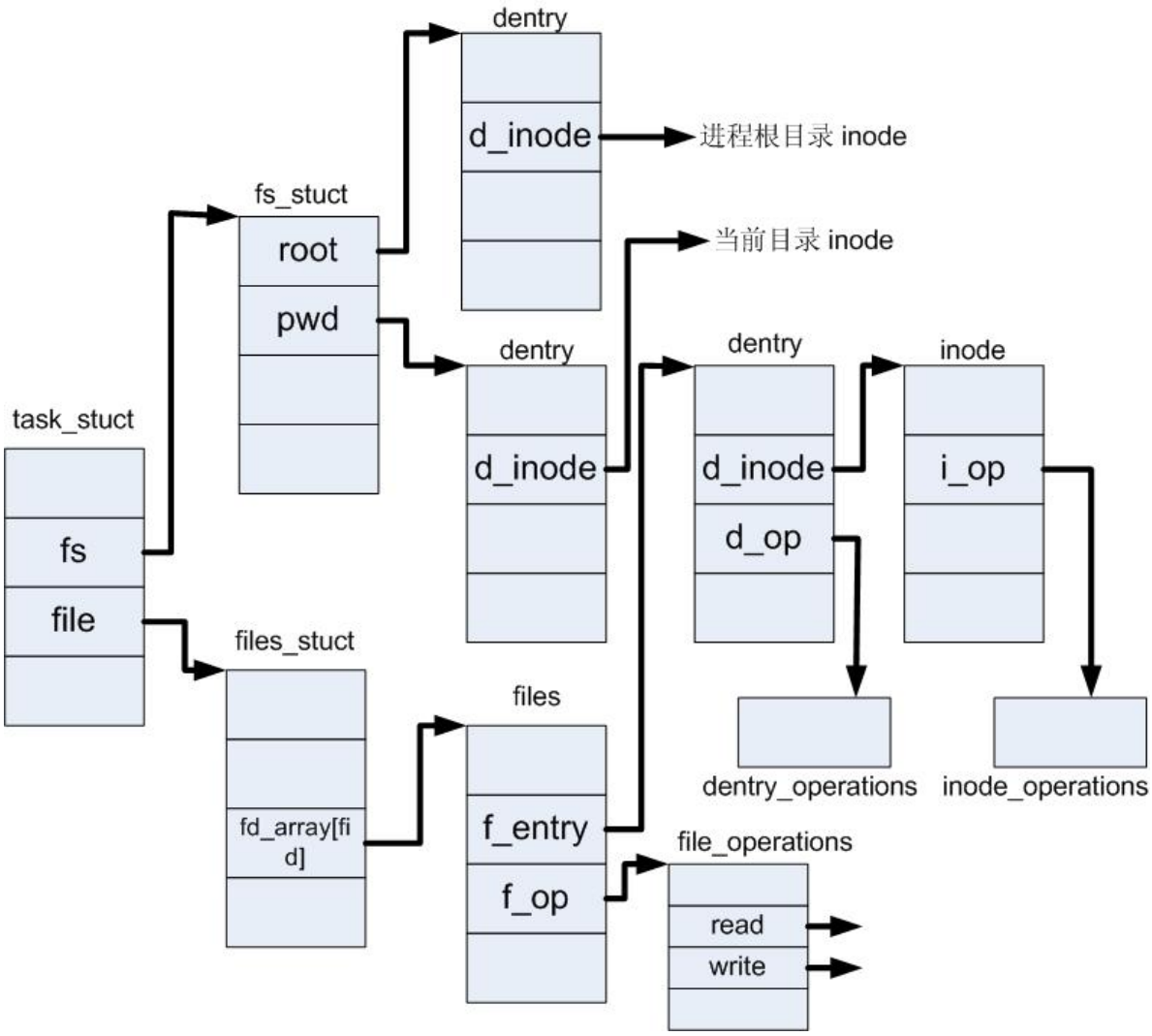
ext3_read_inode()。以一个有 100 block 的硬盘为例，一个文件系统的组织布局大致如下图。位图区中的每一位表示每一个相应的对象有没有被使用。

特殊文件

特殊文件在内存中有inode和dentry数据结构，但是不一定在存储介质上有"索引节点"，它断电之后的确就玩完了，所以不需要什么载体。当从一个特殊文件读时，所读出的数据是由系统内部按一定的规则临时生成的，或从内存中收集，加工出来的。sysfs里面就是典型的特殊文件。它存储的信息都是由系统动态的生成的，它动态的包含了整个机器的硬件资源情况。从sysfs读写就相当于向kobject层次结构提取数据。

还请注意，我们谈到目录项和索引节点时，有两种含义。一种是在存储介质(硬盘)中的(如ext3_inode)，一种是在内存中的，后者是根据在前者生成的。内存中的表示就是 dentry 和 inode，它是 VFS 中的一层，不管什么样的文件系统，最后在内存中描述它的都是 dentry 和 inode 结构。我们使用不同的文件系统，就是将它们各自的文件信息都抽象到 dentry 和 inode 中去。这样对于高层来说，我们就可以不关心底层的实现，我们使用的都是一系列标准的函数调用。这就是 VFS 的精髓，实际上就是面向对象。

我们在进程中打开一个文件 F，实际上就是要在内存中建立 F 的 dentry,和 inode 结构，并让它们与进程结构联系来，把 VFS 中定义的接口给接起来。我们来看一看这个经典的图。这张图之于文件系统，就像每天爱你多一些之于张学友，番茄炒蛋之于复旦南区食堂，刻骨铭心。



一起散散步——pathwalk

前面说过，只要知道文件的索引节点号，就可以得到那个文件。但是我们在操作文件时，从没听说谁会拿着索引节点号来操作文件，我们只知道文件名而已。它们是如何"和谐"起来的呢？linux 把目录也看成一种文件，里面记录着文件名与索引节点号的对应关系。比如在 ext3 文件系统中，如果文件是一个目录，那么它的内容就是一系列 ext3_dir_entry_2 的结构

```
struct ext3_dir_entry_2 {
__u32 inode; /* Inode number */
__u16 rec_len; /* Directory entry length */
__u8 name_len; /* Name length */
__u8 file_type;
```

```
char name[EXT3_NAME_LEN]; /* File name */
};
```

举个例子，比如要打开/home/test/hello.c。首先，找到‘/’，读入其内容，找到名为"home"的文件的索引节点号，打开/home 这个"文件"，读入内容，找到名为 "test" 的的文件的索引节点号，同理，再打开文件"/home/test"，找到找到名为"hello.c"的文件的索引节点号，最后就得到/home/test/hello.c 了。这就是 path_walk()函数的原理。

其中，根据一个文件夹的 inode，和一个文件名来获取该文件的 inode 结构的函数，就叫 lookup，它是 inode_operations 里面的函数。

```
struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
```

lookup，顾名思义，就是查找，比如查查在 test 这个文件夹下，有没有叫 hello.c 的文件，有的话，就从存储介质中读取其 inode 结构。并用 dentry->d_inode 指向它。所以，我们只要知道了文件的路径和名字，总可以从根目录开始，一层一层的往下走，定位到某一个文件。

superblock 与 vfsmount

接下来还要介绍两个数据结构，superblock 和 vfsmount。super_block 结构是从所有具体的文件系统所抽象出来的一个结构，每一个文件系统实例都会有一对应 super_block 结构。比如每一个 ext2 的分区就有一个 super_block 结构，它记录了该文件系统实例(分区)的某些描述性的信息，比如该文件系统实例的文件系统类型，有多大，磁盘上每一块的大小，还有就是 super_operations。它与 inode，dentry 一样，只是某些内容在内存中的映像。就 ext2 文件系统而言，设备上的超级块为 ext2_super_block。由于 sysfs 是虚拟的文件系统，独一无二，并且只能被 mount 一次，sysfs 的 super_block 结构是 sysfs_sb。sysfs_sb 也是动态的从内存中生成的。

还有要提一下 super_operations，它也算是 VFS 的一个接口。实现一个文件系统 file_operations, dentry_operations, inode_operations, super_operations 这四个结构都要实现。

把一个设备安装到一个目录节点时要用一个 vfsmount 的作为连接件。vfsmount 结构定义如下：

```
struct vfsmount {
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent; /* fs we are mounted on */
        struct dentry *mnt_mountpoint; /* dentry of mountpoint */
        struct dentry *mnt_root; /* root of the mounted tree */
        struct super_block *mnt_sb; /* pointer to superblock */
        ..........
}
```

对于某个文件系统实例，内存中 super_block 和 vfsmount 都是唯一的。比如，我们将某个挂载硬盘分区 mount -t vfat /dev/hda2 /mnt/d。实际上就是新建一个 vfsmount 结构作为连接件，vfsmount->mnt_sb = /dev/hda2 的超级块结构；vfsmount->mntroot = /dev/hda2 的"根"目录的 dentry；vfsmount->mnt_mountpoint = /mnt/d 的

dentry； vfsmount->mnt_parent = /mnt/d 所属的文件系统的 vfsmount。并且把这个新建的 vfsmount 连入一个全局的 hash 表 mount_hashtable 中。

从而我们就可以从总根'/'开始，沿着 dentry 往下找。假如碰到一个某个目录的 dentry 是被 mount 了的，那么我们就从 mount_hashtable 表中去寻找相应的 vfsmount 结构 (函数是 lookup_mnt())。然后我们得到 vfsmount ->mnt_root，就可以找到 mount 在该目录的文件系统的"根"dentry 结构。然后又继续往下走，就可以畅通无阻了。

关于 path_walk()的代码我就不贴了，太长了。其实懂了原理后再去看，很简单，跟看故事会差不多。我当年就是看完这个函数后，信心倍增阿。pathwalk，不管前面是高速公路，或是泥泞的乡间小路，我们都要走到底。

Sysfs 文件系统模型

最近 Linus 炮轰 C++，“C++是一种糟糕的（horrible）语言。而且因为有大量不够标准的程序员在使用而使许多真正懂得底层问题，而不会折腾那些白痴'对象模型'"。牛人就是牛气冲天阿。

在 fs/sysfs/下面，除去 makefile，还有 8 个文件。其中， bin.c, file.c, dir.c, symblink.c 分别代表了在 sysfs 文件系统中当文件类型为二进制文件，普通文件，目录，符号连接时的各自的 file operations 结构体的实现。inode.c 则是 inode oprations 的实现，还有创建和删除 inode。mount.c 包括了 sysfs 的初始化函数。sysfs.h 就是头文件，里面有函数的原形，并将其 extern 出去。

sysfs 的文件系统的所读写的信息是存放在 kobject 当中，那么 dentry 是如何与 kobject 联系起来的呢？是通过 sysfs_dirent。

sysfs_dirent

sysfs 文件系统有自己的 dirent 结构，dirent = directory entry (目录实体)。sysfs 中，每一个 dentry 对应了一个 dirent 结构，dentry->d _fsdata 是一个 void 的指针，它指向 sysfs_dirent 结构。

```
struct sysfs_dirent {
        atomic_t            s_count;
        struct list_head  s_sibling;
        struct list_head  s_children;
        void *                s_element;
        int                    s_type;
        umode_t           s_mode;
        struct dentry *     s_dentry;
        struct iattr *       s_iattr;
        atomic_t           s_event;
};
```

s_count 是引用计数，s_sibling，s_children 指针是这些 sysfs_dirent 连成一个树状结构。s_type 则说明了这个 dirent 具体的类型:

```
#define SYSFS_ROOT 0x0001
#define SYSFS_DIR 0x0002
#define SYSFS_KOBJ_ATTR 0x0004
#define SYSFS_KOBJ_BIN_ATTR 0x0008
#define SYSFS_KOBJ_LINK 0x0020
```

s_element 就是指向相应与 s_type 类型的数据结构。如 DIR(就是 kobject，一个 kobject 对应一个 DIR)，KOBJ_ATTR(attribute 属性，代表一个文件)。sysfs_dirent 是 kobject 和 sysfs 联系的一个中间连接结构。它通过 s_sibling，s_children 连接成一个层次结构。而且它的层次结构与 sysfs 完全一致的，它就是一个连接 kobject 和 dentry 结构的连接件。

举个例子总结一下这些数据结构的连接关系。在 sysfs 中的文件结构如下

```
/sys/bus/ldd/
|--device
|--driver
`--version
```

它对应的 dentry，dirent，kobject 的连接图如图 1，2，3

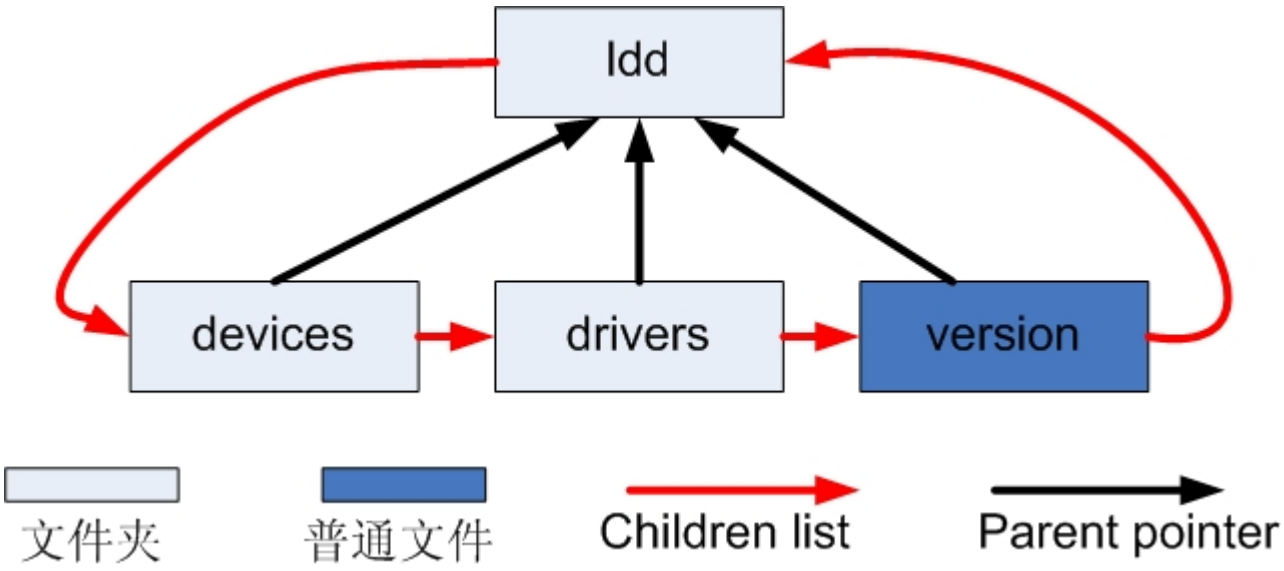


图 1: dentry 连接图

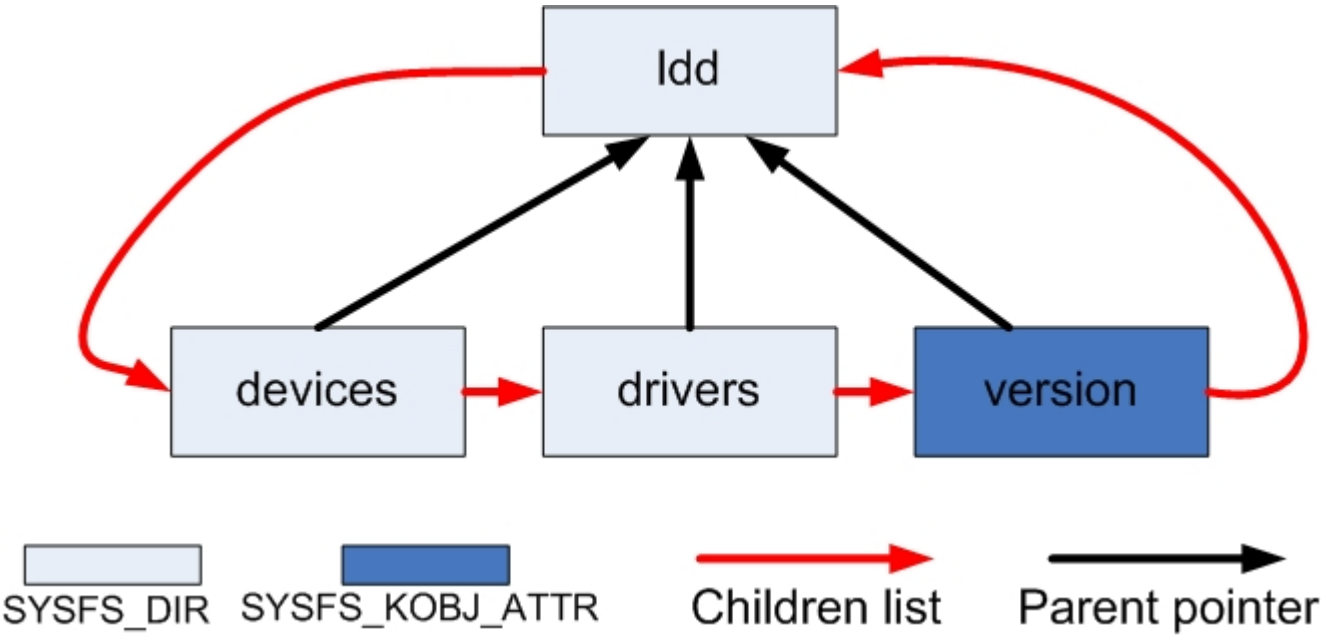


图 2: dirent 连接图

图 3: kobject 连接图

对比一下可以发现不同之处。向 version 这样用 bus_create_file()创建的文件，或曰属性，只停留在 sysfs_dirent 这一层。

对于 sysfs 下的文件夹而言，denrty, dirent, kobject 之间通过指针相互联系起来。

```
dentry->d_fsdata = &dirent;
dirent->element = &kobject;
kobject->dentry = &dentry;
```

Sysfs 创建目录

每当我们新增一个 kobject 结构的时候，同时会在/sys 下创建一个目录。

kobject_add() -> create_dir() -> sysfs_create_dir()

此时，我还想重申，kernel 代码的更新换代是很快的，我们的目的是懂得代码背后的原理，知识，或曰哲学。我不想讲的太细，因为关于 sysfs 的部分从 2.6.10 到现在 2.6.22 已经改了很多了。但其总体架构没变。写此文的目的是让您跟着我的思路走一遍，对 sysfs 有了一个总体上的认识。然后自己就可以去看最新的代码了。最新的代码肯定是效率更高，条理逻辑更清晰。

sysfs_create_dir()流程图如下:

```
            -> create_dir()
                        -> *d = sysfs_get_dentry()
                                    -> lookup_hash()
                                                -> __lookup_hash()
                                                -> cached_lookup()
                                                -> new = d_alloc(base,
name);
                                                ->
inode->i_op->lookup(inode, new, nd)
                        -> sysfs_create(*d, mode, init_dir)
                                    -> sysfs_new_inode(mode)
                                    -> init_dir(inode); \\ Call back
function
                        -> sysfs_make_dirent()
                                    -> sysfs_new_dirent()
                                    -> dentry->d_fsdata = sysfs_get(sd);
                                    -> dentry->d_op =
&sysfs_dentry_ops;
                        -> (*d)->d_op = &sysfs_dentry_ops;
```

```
135 int sysfs_create_dir(struct kobject * kobj)
136 {
137       struct dentry * dentry = NULL;
138       struct dentry * parent;
139       int error = 0;
140
141       BUG_ON(!kobj);
142
143       if (kobj->parent)
144            parent = kobj->parent->dentry;
145       else if (sysfs_mount && sysfs_mount->mnt_sb)
146            parent = sysfs_mount->mnt_sb->s_root;
147       else
148            return -EFAULT;
149
150       error = create_dir(kobj,parent,kobject_name(kobj),&dentry);
151       if (!error)
152            kobj->dentry = dentry;
153       return error;
154 }
```

143-148 就是找到父辈的 kobject，再调用 create_dir()；

```
 95 static int create_dir(struct kobject * k, struct dentry * p,
 96                 const char * n, struct dentry ** d)
```

```
 97 {
 98        int error;
 99        umode_t mode = S_IFDIR| S_IRWXU | S_IRUGO | S_IXUGO;
100
101        down(&p->d_inode->i_sem);
102        *d = sysfs_get_dentry(p,n);
103        if (!IS_ERR(*d)) {
104             error = sysfs_create(*d, mode, init_dir);
105             if (!error) {
106                  error = sysfs_make_dirent(p->d_fsdata, *d, k, mode,
107                                            SYSFS_DIR);
108                  if (!error) {
109                       p->d_inode->i_nlink++;
110                       (*d)->d_op = &sysfs_dentry_ops;
111                       d_rehash(*d);
112                  }
113             }
114             if (error && (error != -EEXIST))
115                  d_drop(*d);
116             dput(*d);
117        } else
118             error = PTR_ERR(*d);
119        up(&p->d_inode->i_sem);
120        return error;
121 }
```

99 行，设置‘文件’ 属性，101 获取信号量。

(1)sysfs_get_dentry()

102 行 sysfs_get_dentry()。它的作用是根据父辈 dentry 和文件名得到 dentry 结构。首先在缓存中找，如果找到就返回，找不到就用 d_alloc()新建一个 dentry 结构。我们是新建文件夹，缓存中自然是没有的，所以要用 d_alloc()来新建一个。接着我们调用 lookup 函数，它定义如下。

```
struct inode_operations sysfs_dir_inode_operations = {
        .lookup = sysfs_lookup,
};
```

```
204 static struct dentry * sysfs_lookup(struct inode *dir, struct dentry *dentry,
205                                      struct nameidata *nd)
206 {
207        struct sysfs_dirent * parent_sd = dentry->d_parent->d_fsdata;
208        struct sysfs_dirent * sd;
209        int err = 0;
210
211        list_for_each_entry(sd, &parent_sd->s_children, s_sibling) {
```

```
212             if (sd->s_type & SYSFS_NOT_PINNED) {
213                     const unsigned char * name = sysfs_get_name(sd);
214
215                     if (strcmp(name, dentry->d_name.name))
216                             continue;
217
218                     if (sd->s_type & SYSFS_KOBJ_LINK)
219                             err = sysfs_attach_link(sd, dentry);
220                     else
221                             err = sysfs_attach_attr(sd, dentry);
222                     break;
223             }
224     }
225
226     return ERR_PTR(err);
227 }
```

前面讲过 lookup 函数的作用。它在 inode 代表的文件夹下查找有没有名为 dentry.d name.name 的文件。如果有，就将其对应的 inode 结构从信息的载体中读出来。由于是新建的文件夹，所以 lookup 函数在我们这个故事里根本没做事。但是我还是忍不住想分析一下 lookup 函数。

sysfs 文件系统中，文件夹的 inode 和 dentry 结构一直都是存在于内存中的，所以不用再进行读取了。而文件，链接的 inode 事先是没有的，需要从载体中读出。这就是 212 行这个判断的作用。可以看出，如果是文件夹，循环里面啥都没做。

```
#define SYSFS_NOT_PINNED \
(SYSFS_KOBJ_ATTR | SYSFS_KOBJ_BIN_ATTR | SYSFS_KOBJ_LINK)
```

但是 sysfs 的 lookup 还有它不同之处。其他文件系统像 ext3 格式中普通文件的 inode，在文件创建之时就已经创建了。但是，sysfs 不一样，它在创建普通文件时，只是先创建一个 sysfs_dirent 结构。创建 inode 的工作是推迟到 lookup 函数来完成的。在下一节 sysfs_create_file()会看到这一点。

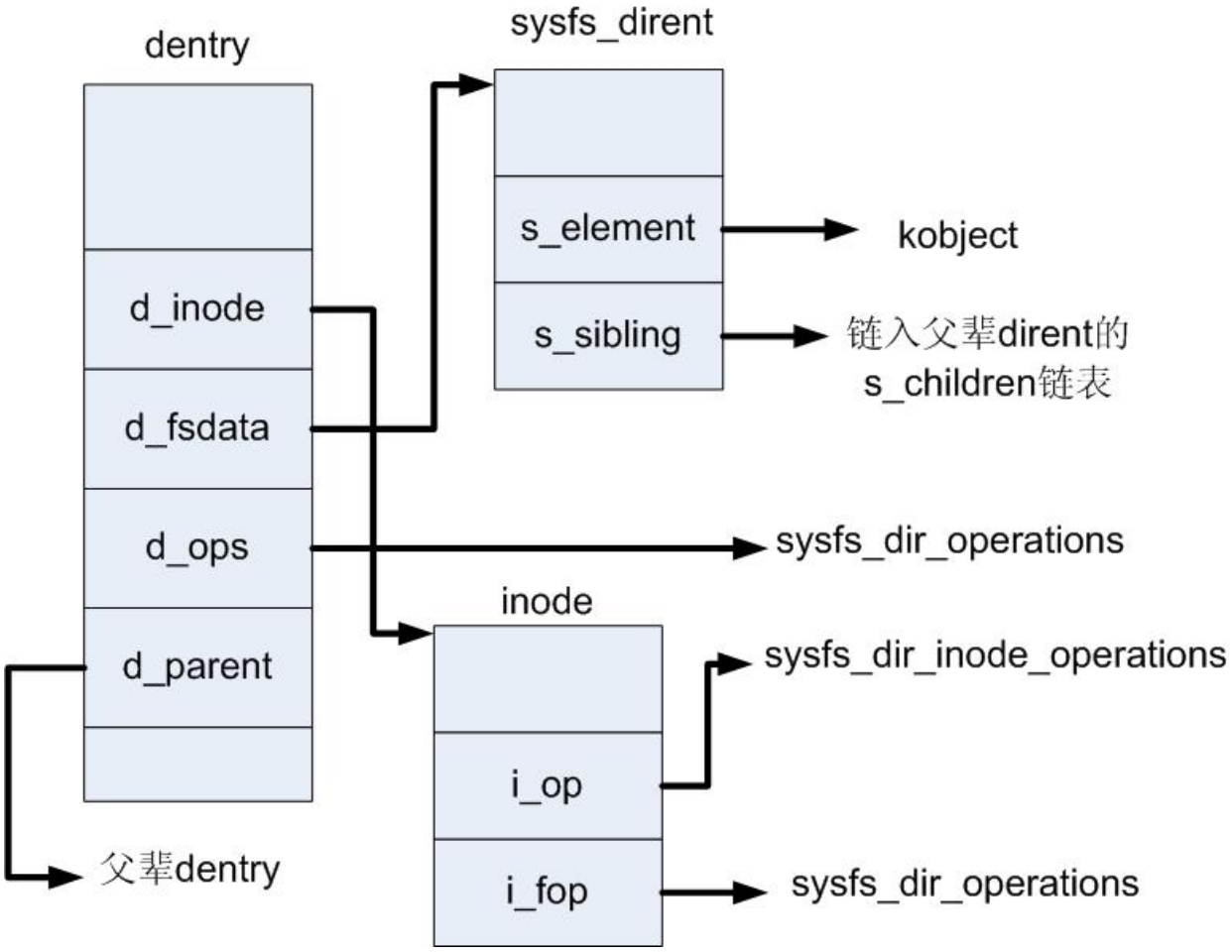
sysfs_attach_attr()和 sysfs_attach_link()的作用就是根据 dentry 和 sysfs_dirent 新建一个 inode。

总之，我们通过 sysfs_get_dentry()得到了一个新建的 dentry 结构。

(2)sysfs_create()分析 (104 行)
sysfs_create()->sysfs_new_inode(mode) -> new_inode(sysfs_sb)
创建一个新的索引节点 inode。sysfs_sb 是 sysfs 的超级块(super_block)结构。mode 则是 inode 的属性，它记录了如下信息，比如，文件类型(是文件夹，链接，还是普通文件)，inode 的所有者，创建时间等等。

(3)sysfs make dirent()分析 (104 行)
至此，我们得到了一个 dirent 结构,初始化，再把它连接到上层目录的 sysfs_dirent 的 s_children 链表里去。sysfs_make_dirent()为刚刚新建出来的 dentry 建立一个 dirent 结构。并将 dentry 和 dirent 联系起来。

(4)总结
在 sysfs 下创建一个目录，提供的函数是 sysfs_create_dir()。创建了 dentry, dirent, inode 结构，它们之间的连接关系见图
1



Sysfs 创建普通文件

最近彭宇的案件炒得沸沸扬扬，究竟这个社会怎么了？

sysfs 文件系统中，普通文件对应于 kobject 中的属性。用 sysfs_create_file()，参数如下：

sysfs_create_file(struct kobject * kobj, const struct attribute * attr)

传给它的参数是 kobj 和 attr，其中，kobject 对应的是文件夹，attribute 对应的是该文件夹下的文件。

```
int sysfs_create_file(struct kobject * kobj, const struct attribute * attr)
{
        BUG_ON(!kobj || !kobj->dentry || !attr);
        return sysfs_add_file(kobj->dentry, attr, SYSFS_KOBJ_ATTR);
}
```

它直接调用 sysfs_add_file()

```
int sysfs_add_file(struct dentry * dir, const struct attribute * attr, int type)
{
        struct sysfs_dirent * parent_sd = dir->d_fsdata;
        umode_t mode = (attr->mode & S_IALLUGO) | S_IFREG;
        int error = 0;

        down(&dir->d_inode->i_sem);
        error = sysfs_make_dirent(parent_sd, NULL, (void *) attr, mode,
type);
        up(&dir->d_inode->i_sem);
        return error;
}

int sysfs_make_dirent(struct sysfs_dirent * parent_sd, struct dentry * dentry,
  void * element, umode_t mode, int type)
{
        struct sysfs_dirent * sd;

        sd = sysfs_new_dirent(parent_sd, element);
        if (!sd)
                return -ENOMEM;

        sd->s_mode = mode;
        sd->s_type = type;
        sd->s_dentry = dentry;
        if (dentry) {
                dentry->d_fsdata = sysfs_get(sd);
                dentry->d_op = &sysfs_dentry_ops;
        }

        return 0;
}
```

sysfs_create_file()仅仅是调用了 sysfs_make_dirent()创建了一个 sysfs_dirent 结构。与 sysfs_create_dir()不同，它甚至没有在 sysfs 文件系统下创建 inode 结构。这项工作被滞后了，在 sysfs_lookup()->sysfs_attach_attr()里面完成的。

Sysfs 读入文件夹内容

上回我们说到，如何创建文件夹和文件。我们发现，在 sysfs 中，inode 并不那么重要。这是因为我们所要读写的信息已经就在内存中，并且已经形成了层次结构。我们只需有 dentry，就可以 dentry->fsdata，就能找到我们读些信息的来源 --- sysfs_dirent 结构。这也是我觉得有必要研究 sysfs 的原因之一，因为它简单，而且不涉及具体的硬件驱动，但是从这个过程中，我们可以把文件系统中的一些基本数据结构搞清楚。接下来，我以读取 sysfs 文件和文件夹的内容为例子，讲讲文件读的流程。那么关于写，还有关于 symblink 的东西完全可以以此类推了。

我们新建文件夹时，设置了

```
inode->i_op = &sysfs_dir_inode_operations;
inode->i_fop = &sysfs_dir_operations;

struct file_operations sysfs_dir_operations = {
        .open = sysfs_dir_open,
        .release = sysfs_dir_close,
        .llseek = sysfs_dir_lseek,
        .read = generic_read_dir,
        .readdir = sysfs_readdir,
};
```

用一个简短的程序来做实验。

```
#include<sys/types.h>
#include<dirent.h>
#include<unistd.h>
int main(){
        DIR * dir;
        struct dirent *ptr;
        dir = opendir("/sys/bus/");
          while((ptr = readdir(dir))!=NULL){
                printf("d_name :%s ",ptr->d_name);
        }
        closedir(dir);
        return -1;
}
```

在用户空间，用 gcc 编译执行即可。我们来看看它究竟做了什么。

(1)sysfs_dir_open()

这是个用户空间的程序。opendir()是 glibc 的函数，glibc 也就是著名的标准 c 库。至于 opendir()是如何与 sysfs dir open ()接上头的，那还得去看 glibc 的代码。我就不想分析了...glibc 可以从 gnu 的网站上自己下载源代码，编译。再用 gdb 调试，就可以看得跟清楚。

函数流程如下：

opendir("/sys/bus/") -> /*用户空间*/

-> 系统调用->
sys_open() -> filp_open()-> dentry_open() -> sysfs_dir_open()/*内核空间*/

```
 static int sysfs_dir_open(struct inode *inode, struct file *file)
{
        struct dentry * dentry = file->f_dentry;
        struct sysfs_dirent * parent_sd = dentry->d_fsdata;
        down(&dentry->d_inode->i_sem);
        file->private_data = sysfs_new_dirent(parent_sd, NULL);
        up(&dentry->d_inode->i_sem);
        return file->private_data ? 0 : -ENOMEM;
}
```

内核空间:新建一个 dirent 结构,连入父辈的 dentry 中,并将它地址保存在 file->private_data 中。这个 dirent 的具体作用待会会讲。
用户空间:新建了一个 DIR 结构, DIR 结构如下。

```
#define __dirstream DIR
struct __dirstream
{
        int fd; /* File descriptor. */
        char *data; /* Directory block. */
        size_t allocation; /* Space allocated for the block. */
        size_t size; /* Total valid data in the block. */
        size_t offset; /* Current offset into the block. */
        off_t filepos; /* Position of next entry to read. */
        __libc_lock_define (, lock) /* Mutex lock for this structure. */
};
```

(2)sysfs_readdir()

流程如下:

readdir(dir) -> getdents() ->/*用户空间*/
-> 系统调用->
sys32 readdir() -> vfs readdir() -> sysfs readdir()/*内核空间*/

readdir(dir)这个函数有点复杂,虽然在 main 函数里的 while 循环中,readdir 被执行了多次,我们看看 glibc 里面的代码

```
readdir(dir){
        ......
          if (dirp->offset >= dirp->size){
        ......
        getdents()
        ......
        }
```

```
......
}
```

实际上，getdents() -> ... -> sysfs_readdir()只被调用了两次，getdents()一次就把所有的内容都读完，存在 DIR 结构当中，readdir()只是从 DIR 结构当中每次取出一个。DIR(dirstream)结构就是一个流。而回调函数 filldir 的作用就是往这个流中填充数据。第二次调用 getdents()是用户把 DIR 里面的内容读完了，所以它又调用 getdents()但是这次 getdents()回返回 NULL。

```
 static int sysfs_readdir(struct file * filp, void * dirent, filldir_t filldir)
{
        struct dentry *dentry = filp->f_dentry;
        struct sysfs_dirent * parent_sd = dentry->d_fsdata;
        struct sysfs_dirent *cursor = filp->private_data;
        struct list_head *p, *q = &cursor->s_sibling;
        ino_t ino;
        int i = filp->f_pos;
          switch (i) {
                case 0:
                ino = dentry->d_inode->i_ino;
                if (filldir(dirent, ".", 1, i, ino, DT_DIR) < 0)
                        break;
                filp->f_pos++;
                i++;
                  /* fallthrough */
                case 1:
                ino = parent_ino(dentry);
                if (filldir(dirent, "..", 2, i, ino, DT_DIR) < 0)
                        break;
                filp->f_pos++;
                i++;
                  /* fallthrough */
                default:
                  if (filp->f_pos == 2) {
                        list_del(q);
                        list_add(q, &parent_sd->s_children);
                }
                  for (p=q->next; p!= &parent_sd->s_children; p=p->next) {
                        struct sysfs_dirent *next;
                        const char * name;
                        int len;
                        next = list_entry(p, struct sysfs_dirent, s_sibling);
                        if (!next->s_element)
                                continue;
                        name = sysfs_get_name(next);
```

```
                        len = strlen(name);
                        if (next->s_dentry)
                                ino = next->s_dentry->d_inode->i_ino;
                        else
                                ino = iunique(sysfs_sb, 2);
                        if (filldir(dirent, name, len, filp->f_pos, ino,dt_type(n
ext)) < 0)
                                return 0;
                        list_del(q);
                        list_add(q, p);
                        p = q;
                        filp->f_pos++;
                }
        }
return 0;
}
```

看 sysfs_readdir()其实很简单，它就是从我们调用 sysfs_dir_open()时新建的一个 sysfs_dirent 结构开始，便利当前 dentry->dirent 下的所有子 sysfs_dirent 结构。读出名字，再回调函数 filldir()将文件名，文件类型等信息，按照一定的格式写入某个缓冲区。

一个典型的 filldir()就是 filldir64()，它的作用的按一定格式向缓冲区写数据，再把数据复制到用户空间去。

Sysfs 读入普通文件内容

跟上回一样，我用这个小程序来读

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(){
        char *name = "/sys/bus/ldd/version";
        char buf[500];
        int fd;
        int size;
        fd = open(name, O_RDONLY);
        printf("fd:%d ",fd);
        size = read(fd,buf,sizeof(buf));
        printf("size:%d ",size);
        printf("%s",buf);
        close(fd);
```

```
        return -1;
}
```

(1)sysfs_open_file()

open() ->/*用户空间*/
-> 系统调用->
sys_open() -> filp_open()-> dentry_open() -> sysfs_open_file()/*内核空间*/

```
static int sysfs_open_file(struct inode * inode, struct file * filp)
{
        return check_perm(inode,filp);
}
```

```
 static int check_perm(struct inode * inode, struct file * file)
{
    struct kobject *kobj = sysfs_get_kobject(file->f_dentry->d_parent);
    struct attribute * attr = to_attr(file->f_dentry);
    struct sysfs_buffer * buffer;
    struct sysfs_ops * ops = NULL;
    int error = 0;

    if (!kobj || !attr)
        goto Einval;

    /* Grab the module reference for this attribute if we have one */
    if (!try_module_get(attr->owner)) {
        error = -ENODEV;
        goto Done;
    }

    /* if the kobject has no ktype, then we assume that it is a subsystem
     * itself, and use ops for it.
     */
    if (kobj->kset && kobj->kset->ktype)
        ops = kobj->kset->ktype->sysfs_ops;
    else if (kobj->ktype)
        ops = kobj->ktype->sysfs_ops;
    else
        ops = &subsys_sysfs_ops;

    /* No sysfs operations, either from having no subsystem,
     * or the subsystem have no operations.
     */
```

```
    if (!ops)
        goto Eaccess;

    /* File needs write support.
     * The inode's perms must say it's ok,
     * and we must have a store method.
     */
    if (file->f_mode & FMODE_WRITE) {

        if (!(inode->i_mode & S_IWUGO) || !ops->store)
            goto Eaccess;

    }

    /* File needs read support.
     * The inode's perms must say it's ok, and we there
     * must be a show method for it.
     */
    if (file->f_mode & FMODE_READ) {
        if (!(inode->i_mode & S_IRUGO) || !ops->show)
            goto Eaccess;
    }

    /* No error? Great, allocate a buffer for the file, and store it
     * it in file->private_data for easy access.
     */
    buffer = kmalloc(sizeof(struct sysfs_buffer),GFP_KERNEL);
    if (buffer) {
        memset(buffer,0,sizeof(struct sysfs_buffer));
        init_MUTEX(&buffer->sem);
        buffer->needs_read_fill = 1;
        buffer->ops = ops;
        file->private_data = buffer;
    } else
        error = -ENOMEM;
    goto Done;

Einval:
    error = -EINVAL;
    goto Done;
Eaccess:
    error = -EACCES;
    module_put(attr->owner);
Done:
```

```
    if (error && kobj)
        kobject_put(kobj);
    return error;
}
```

check_perm()检查一下权限，创建一个 sysfs 的缓冲区 sysfs_buffer buffer，并设置其 sysfs_ops sysfs_buffer->ops。在我们这个故事里，sysfs_buffer->ops 被设置成 bus_sysfs_ops。最后让 file->private_data = buffer。

(2)sysfs read file()

流程如下：
read()->/*用户空间*/
-> 系统调用->
sys_read() -> vfs_read() -> sysfs_read_file()/*内核空间*/

看看 sysfs_read_file()函数，

```
 static ssize_t
 sysfs_read_file(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    struct sysfs_buffer * buffer = file->private_data;
    ssize_t retval = 0;

    down(&buffer->sem);
    if (buffer->needs_read_fill) {
        if ((retval = fill_read_buffer(file->f_dentry,buffer)))
            goto out;
    }
    pr_debug("%s: count = %d, ppos = %lld, buf = %s ",
        __FUNCTION__,count,*ppos,buffer->page);
    retval = flush_read_buffer(buffer,buf,count,ppos);
out:
    up(&buffer->sem);
    return retval;
}
```

顺着 sysfs_read_file()往下走：
sysfs_read_file()
        ---> fill_read_buffer()
                ---> sysfs_buffer->bus_sysfs_ops->bus_attr_show()
                        ---> bus_attribute->show_bus_version() //注意这个函数是我们在 lddbus.c 里面定义的
        ---> flush_read_buffer()

fill_read_buffer()的是真正的读，它把内容读到 sysfs 定义的缓冲区 sysfs_buffer。
flush_read_buffer()是把缓冲区 copy 到用户空间。详细内容我就不贴了。

后记

关于 sysfs 的介绍就到这里。文笔不如大哥甲幽默，希望我写的这些咚咚对大家有所启发，帮助。把东西看懂是一回事，写出来又是另一回事。写到想吐血...但是写得过程中，可以使自己的理解更深一层，更有逻辑。授人以鱼不如授人以渔，学习 linux 内核最好的方法就是 Reading the f**king source code。 推荐用 source insight 看代码，外加 KDB 调试，挺方便的。另外，除了 linux 内核代码，最好把 glibc 的代码也下下来，都自己编译一遍。这样的话，在用户空间就可以用 gdb 方便的调试。

近来无事去母校 BBS 的星座版逛了一逛，发现里面里斥着小 mm 们的"求解星盘"，"XX 座和 XX 座的性格比较"等话题，人气极旺。心想哪天要是混不下去了，改行去算命得了，这比做技术有前途多了。或者就去写写鬼故事，像什么鬼吹灯，盗墓笔记，也是一个字，火。只要不太监，写得越离奇，越诡异，看得人越多。我也很喜欢看，可是后面实在是太罗嗦了。明年就奥运了，17 大也快要召开了，事情挺多的。关键是做好自己的事情。

感谢大家的支持，只要有人看，我们就会继续下去。虽然比较辛苦，但是有意义。真的，哥儿们，读代码比打网游有意思多了。静下心来，泡杯茶，听听音乐，你会发现，读源代码就跟看故事会一样简单。

读 好书

干 实事

知行合一