LINUX 下设备层次组织

一.sys 文件系统的基础

构建 sys 文件系统就先对 kobject 与 kset 进行分析，sys 下的目录结构也将内核中的驱动层次进行了有效的展示，kset 是 kobject 的集合，这种集合的形式也将内核中的驱动按层次结构对用户进行展示。

首先先简绍 kobject 结构：

```
struct kobject {
    const char          *name;    /*sys 目录创建名称*/
    struct list_head    entry;    /*kobject链表结构*/
    struct kobject      *parent; /*kobject的父指针*/
    struct kset         *kset;    /*对应的KSET*/
    struct kobj_type    *ktype;   /*创建等一些属性*/
    struct sysfs_dirent *sd;
    struct kref         kref;     /*引用计数*/
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```

其中 kobject 中的 ktype 结构需要重点说明：

```
struct kobj_type {
    void (*release)(struct kobject *kobj);
    const struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
    const struct kobj_ns_type_operations *(*child_ns_type)
    const void *(*namespace)(struct kobject *kobj);
};
```

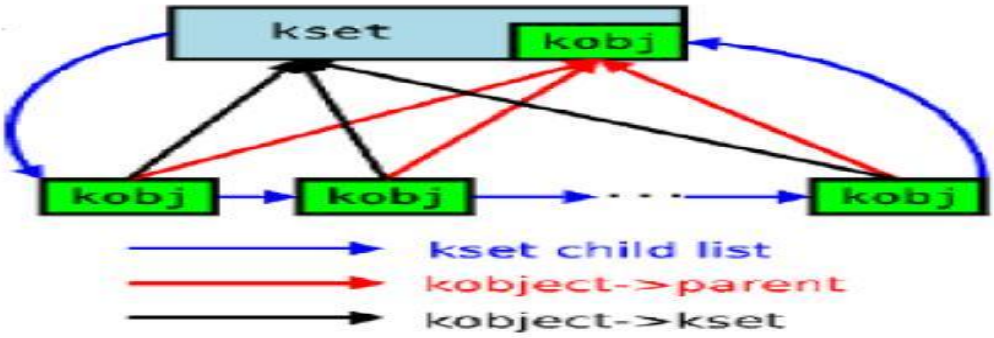
里边是不同的 Kobject 中的释放 和展示函数

查看 kset 数据结构

```
struct kset {
    struct list_head list;
    spinlock_t list_lock;
    struct kobject  kobj;
    const struct kset_uevent_ops *uevent_ops;
};
```

Kset 属性解析结构，其中包含有的数据 kobject 和 uevent_op 和其中链表。

其中 kset 和 kobject 的结构如下图所示：



由 kobject 与 sys 目录建立的关系：

```
static int kobject_add_internal(struct kobject *kobj)
{
    int error = 0;
    struct kobject *parent;

    if (!kobj)
        return -ENOENT;

    if (!kobj->name || !kobj->name[0]) {
        WARN(1, "kobject: (%p): attempted to be registered with empty
             "name!\n", kobj);
        return -EINVAL;
    }

    parent = kobject_get(kobj->parent);

    /* join kset if set, use it as parent if we do not already have o
    if (kobj->kset) {
        if (!parent)
            parent = kobject_get(&kobj->kset->kobj);
        kobj_kset_join(kobj);
        kobj->parent = parent;
    }

    pr_debug("kobject: '%s' (%p): %s: parent: '%s', set: '%s'\n",
         kobject_name(kobj), kobj, __func__,
         parent ? kobject_name(parent) : "<NULL>",
         kobj->kset ? kobject_name(&kobj->kset->kobj) : "<NULL>");

    error = create_dir(kobj);
    if (error) {
        kobj_kset_leave(kobj);
        kobject_put(parent);
        kobj->parent = NULL;
```

在该函数中 Kobject_kset_join 建立层次的 kobject 与 对应 kset 的链表连接关系， create_dir 用于在 /sys/下建立相应的设备层次目录，其中建立目录会用到 parrent 的目录结构，决定在/sys/下的某个层创建。

那么 kset 的目录创建的流程是怎样的？

Kset_register --- > Kobject_add_internal

具体的目录创建流程分析 create_dir 函数调用，最终的调用层次可以看出 /sys/目录下目录创建的基础是 kobject

二 .linux 下 sys 目录下驱动建立目录层次的结构流程

进入 linux 系统下的 /sys 目录中可以看到下图所示的目录组织结构

```
liu@ubuntu:/sys$ ls
block  bus  class  dev  devices  firmware  fs  kernel  module  power
```

当然这些也正是系统启动的时候根据相应的 KSET 进行建立的，现在从 内核中代码进行分析，查看 /sys/目录下的结构是在什么时候进行建立的。

```
int __init devices_init(void)
{
    devices_kset = kset_create_and_add("devices", &device_uevent_ops, NULL);
    if (!devices_kset)
        return -ENOMEM;
    dev_kobj = kobject_create_and_add("dev", NULL);
    if (!dev_kobj)
        goto dev_kobj_err;
    sysfs_dev_block_kobj = kobject_create_and_add("block", dev_kobj);
    if (!sysfs_dev_block_kobj)
        goto block_kobj_err;
    sysfs_dev_char_kobj = kobject_create_and_add("char", dev_kobj);
    if (!sysfs_dev_char_kobj)
        goto char_kobj_err;

    return 0;
```

在系统启动的时候，通过创建和初始化这些 kset 结构首先在 sys 目录建立下 device dev block char 目录。同时也建立了一些 kset 用于以后的 kobject 进行建立的时候进行各种其他的层次目录建立。

查看 Bus 目录的建立，便于以下说明 platform 建立的说明

```
int __init buses_init(void)
{
    bus_kset = kset_create_and_add("bus", &bus_uevent_ops, NULL);
    if (!bus_kset)
        return -ENOMEM;
    return 0;
}
```

系统启动后在 /sys/根目录下创建那 bus 目录

三 .paltform 设备驱动总线的建立流程分析：

首先先分析 bus_type 结构体的变量及说明：

```
struct bus_type {
    const char      *name;
    struct bus_attribute    *bus_attrs;
    struct device_attribute *dev_attrs;
    struct driver_attribute *drv_attrs;

    int (*match)(struct device *dev,struct device_driver *drv); /*总线提供的匹配函数*/
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);/*总线提供的udev处理*/
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);

    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    const struct dev_pm_ops *pm;

    struct bus_type_private *p;                    /*所有的kobject链表都在这*/
};
```

其中的数据结构 bus_type_paivate 如下图所示：driver_kset ,device_kset 存放着链连接着设备中所有的 driver 和 device 的 kobject 。

```
struct bus_type_private {
    struct kset subsys;
    struct kset *drivers_kset;
    struct kset *devices_kset;
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type *bus;
};
```

以下是 platform 顶层结构注册生成：
Device_regster 注册生成在 /sys/device/platform 设备节点
重点分析 bus_register ();

```
struct bus_type platform_bus_type = {
    .name       = "platform",
    .dev_attrs  = platform_dev_attrs,
    .match      = platform_match,
    .uevent     = platform_uevent,
    .pm     =   &platform_dev_pm_ops,
};
```

上图所示的为 plateform 的设备匹配函数和设备添加时的 uevent 通知函数

```
int __init platform_bus_init(void)
{
    int error;

    early_platform_cleanup();

    error = device_register(&platform_bus);
    if (error)
        return error;
    error =  bus_register(&platform_bus_type);
    if (error)
        device_unregister(&platform_bus);
    return error;
}
```

Bus_register

通过对一下 bus_register 的分析可知，在注册 plateform 后，首先在 bus 目录下创建一个 Platform 目录，创建的目录结构 /sys/bus/platform/ ，另外在前目录下创建 driver 和 device 目录结构 。

```
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);
```

```
priv->devices_kset = kset_create_and_add("devices", NULL,
                                         &priv->subsys.kobj);
if (!priv->devices_kset) {
    retval = -ENOMEM;
    goto bus_devices_fail;
}

priv->drivers_kset = kset_create_and_add("drivers", NULL,
                                         &priv->subsys.kobj);
```

上边已经创建好 paltform 总线，之后再进行 platform 总线 ，驱动和设备的添加过程。

Platform_device_register

接下来就是平台设备的注册了，此时首先调用 device_initialize 进行设备的初始化，注意此时的 device_initalize 已经进行了 bus 的初始化，此后在总线上的驱动匹配也就可以正常进行了。具体的初始化流程如下。

```
int platform_device_register(struct platform_device *pdev)
{
    device_initialize(&pdev->dev);
    return platform_device_add(pdev);
}
```

Device_initialize 首先进行的是 kset device_kset ..这是在注册之后 在 /sys/device 下显示的设备目录名称，初始化链表结构，接下来就要进行设备的总线的注册和匹配。

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

接下来进行平台设备的注册，此时首先设置 设备的总线 platform_bus ，注意此时的设备总线的匹配函数，将设备注册之后，就会调用总线上的匹配函数进行设备与驱动的匹配，接下来会调用 device_add 进行设备注册

```
 */
int platform_device_add(struct platform_device *pdev)
{
    int i, ret = 0;

    if (!pdev)
        return -EINVAL;

    if (!pdev->dev.parent)
        pdev->dev.parent = &platform_bus;

    pdev->dev.bus = &platform_bus_type;

    if (pdev->id != -1)
        dev_set_name(&pdev->dev, "%s.%d", pdev->name,  pdev->id);
    else
        dev_set_name(&pdev->dev, "%s", pdev->name);
```

```
 ret = device_add(&pdev->dev);
 if (ret == 0)
     return ret;
```

Device_add

进入设备注册的最重要的一个阶段，设备注册于驱动匹配，详细过程如下：

1.首先在初始化的时候已将设置过父亲指针，之后再设置了福指针之后，调用 Kobject_add 进行设备在/sys/device 下目录的创建 xx 目录，在用 device_create_file 然后在调用 device_create_file 在/sys/device/xx 下创建设备的 udev 属性文件

```
    set_dev_node(dev, dev_to_node(parent));

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

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

error = device_create_file(dev, &uevent_attr);
```

继续向下走：

```
kobject_uevent(&dev->kobj, KOBJ_ADD);
bus_probe_device(dev);
```

Kobject_uevent 进行设备插入的通知创建设备节点其中会调用总线的 platform 的 uevent 函数，进行目录的补充。

最重要的函数到来 **bus_parobe_device**

Bus_probe_device

```
void bus_probe_device(struct device *dev)
{
	struct bus_type *bus = dev->bus;
	int ret;

	if (bus && bus->p->drivers_autoprobe) {
		ret = device_attach(dev);
		WARN_ON(ret < 0);
	}
}
```

在前边的调用的 bus 自动枚举的标示呗设置，紧接着调用 device_attach

Device_detach

```
	pm_runtime_get_noresume(dev);
	ret = bus_for_each_drv(dev->bus, NULL, dev, __device_attach);
	pm_runtime_put_sync(dev);
```

Device_detach 函数最终调用 bus_for_each_drv 进行设备的匹配，调用回调函数 _device_attach 。

```
int bus_for_each_drv(struct bus_type *bus, struct device_driver *s
		     void *data, int (*fn)(struct device_driver *, void *)
{
	struct klist_iter i;
	struct device_driver *drv;
	int error = 0;

	if (!bus)
		return -EINVAL;

	klist_iter_init_node(&bus->p->klist_drivers, &i,
			     start ? &start->p->knode_bus : NULL);

	while ((drv = next_driver(&i)) && !error)
		error = fn(drv, data);

	klist_iter_exit(&i);
	return error;
}
```

将链表中的设备驱动与设备进行匹配：

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

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

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

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

调用总线的方式进行匹配，总线的匹配方式是名字的匹配，之后就要调用驱动的 probe 调用驱动的 probe 函数。

至此 plateform 驱动分析完毕