sysfs是一个基于RAM的文件系统，可以将内核子系统、硬件设备、驱动程序的信息以文件目录的形式导出到用户空间。sysfs的每个文件只有一个可读、可写或者可读写的值，可以从内核中获取信息，当然也可以向内核发送信息。

# sysfs下目录的创建

每个sysfs下的目录通过kobject结构描述：

struct kobject {

const char \*name;

struct list\_head entry;

struct kobject \*parent;

struct kset \*kset;

struct kobj\_type \*ktype;

struct kernfs\_node \*sd; /\* sysfs directory entry \*/

struct kref kref;

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;

};

.name：该kobject的名称，也就是sysfs下目录的名称，kobject注册到sysfs后再想修改名称，则需要使用kobject\_rename，其会处理sysfs相关事宜。

.entry：用于将kobject添加到kset的list\_head。

.parent：指向parent kobject，会在父object目录下创建当前kobject目录，正是因为此节点，各object才联系到了一起，形成层次结构，也就是我们在sysfs下看到的目录结构。

.kset:该object属于的kset，可为NULL，如存在且未指定parent，则kset就会作为parent。

.ktype：该kobject属于的kobj\_type，每个kobject必须有一个ktype。

.sd：该kobject在sysfs中的表示。

.kref：kobject的引用计数，当任何内核模块引用此kobject时，引用次数就会增加，引用被释放则引用次数减少，当引用次数为0，与kobject相关的内存就会被释放。

.state\_initialized：表示kobject是否已初始化。

.state\_in\_sysfs：表示该kobject是否已在sysfs中呈现，以便在自动注销时从sysfs中移除。

.state\_add\_uevent\_sent/state\_remove\_uevent\_sent：记录是否已经向用户空间发送add uevent，如果已发送但未发送remove uevent，则在自动注销时，补发remove uevent，以便让用户空间正确处理。

.uevent\_suppress：如果为1，则表示忽略所有上报的uevent事件。

sysfs中用于目录相关操作接口有：

int sysfs\_create\_dir\_ns(struct kobject \*kobj);//目录创建

void sysfs\_remove\_dir(struct kobject \*kobj);//目录移除

int sysfs\_rename\_dir\_ns(struct kobject \*kobj, const char \*new\_name);//目录重命名

我们在编写驱动时并不会直接使用上面这些方法，而是使用如下接口创建目录：

struct kobject \* kobject\_create\_and\_add(const char \*name, struct kobject \*parent);

name：创建的目录名称。

parent：指向父目录的kobject，如果为NULL，则为在sysfs下创建。

目录删除：

void kobject\_del(struct kobject \*kobj);

目录重命名：

int kobject\_rename(struct kobject \*kobj, const char \*new\_name);

kobject引用计数的维护：

struct kobject\* kobject\_get(struct kobject \*kobj);

struct kobject\* kobject\_put(struct kobject \*kobj);

# sysfs下文件的创建

每个sysfs目录下的文件通过attribute(属性)来描述，其指的其实是kobject的属性，属性最终是以文件的形式在sysfs下展现。attribute分为普通attribute和二进制attribute：

struct attribute {

const char \*name;//文件名称

umode\_t mode;//文件权限

#ifdef CONFIG\_DEBUG\_LOCK\_ALLOC

bool ignore\_lockdep:1;

struct lock\_class\_key \*key;

struct lock\_class\_key skey;

#endif

};

struct bin\_attribute {

struct attribute attr;

size\_t size;

void \*private;

ssize\_t (\*read)(struct file \*, struct kobject \*, struct bin\_attribute \*, char \*, loff\_t, size\_t);

ssize\_t (\*write)(struct file \*, struct kobject \*, struct bin\_attribute \*, char \*, loff\_t, size\_t);

int (\*mmap)(struct file \*, struct kobject \*, struct bin\_attribute \*attr, struct vm\_area\_struct \*vma);

};

使用普通的attribute生成的sysfs文件，只能用字符串的形式读写，而二进制的attribute因为在普通attribute的基础上增加了read/write/mmap，所以就可以通过任何方式读写了。

我们之所以要创建文件是要用来让用户空间与内核通信，但是attribute结构中并没有文件操作方法，kobject.h中定义了如下结构，包含了文件属性以及操作方法：

struct kobj\_attribute {

struct attribute attr;//属性

ssize\_t (\*show)(struct kobject \*kobj, struct kobj\_attribute \*attr, char \*buf);//读取方法

ssize\_t (\*store)(struct kobject \*kobj, struct kobj\_attribute \*attr, const char \*buf, size\_t count);//写入方法

};

include/linux/sysfs.h中提供了很多宏方便我们初始化：

#define \_\_ATTR(\_name, \_mode, \_show, \_store) { \

.attr = {.name = \_\_stringify(\_name), \

.mode = VERIFY\_OCTAL\_PERMISSIONS(\_mode) }, \

.show = \_show, \

.store = \_store, \

}

然后就可以创建文件了：

int sysfs\_create\_file(struct kobject \*kobj, const struct attribute \*attr);

kobj：文件所属目录，为NULL则为在sysfs目录下创建文件，也可指向我们自己创建的目录，当然也可以将文件创建在内核已经创建好的目录中如/sys/kernel目录的kobj为kernel\_kobj。

attr：属性。

我们可通过如下接口删除文件：

void sysfs\_remove\_file(struct kobject \*kobj, const struct attribute \*attr);

当我们在目录下需要创建很多文件时，我们可以创建一组属性，然后一起创建，属性组通过如下结构描述：

struct attribute\_group {

//属性组名称，为NULL则不会创建子目录，如果不为NULL，则会创建子目录，然后下面的属性全部创建在此子目录中。

const char \*name;

//可不实现，返回属性组中普通属性的权限。

umode\_t (\*is\_visible)(struct kobject \*,

struct attribute \*, int);

//可不实现，返回属性组中二进制属性的权限

umode\_t (\*is\_bin\_visible)(struct kobject \*,

struct bin\_attribute \*, int);

struct attribute \*\*attrs;//普通属性列表，列表以NULL为结尾。

//二进制属性列表，列表以NULL为结尾。

struct bin\_attribute \*\*bin\_attrs;

};

属性组的初始化也可通过宏ATTRIBUTE\_GROUPS(\_name)来完成：

#define \_\_ATTRIBUTE\_GROUPS(\_name) \

static const struct attribute\_group \*\_name##\_groups[] = { \

&\_name##\_group, \

NULL, \

}

#define ATTRIBUTE\_GROUPS(\_name) \

static const struct attribute\_group \_name##\_group = { \

.attrs = \_name##\_attrs, \

}; \

\_\_ATTRIBUTE\_GROUPS(\_name)

这个宏展开其实就是对struct attribute\_group的attrs进行了赋值，如果使用ATTRIBUTE\_GROUPS(test)则需要已存在名称为test\_attrs的属性列表，初始化完成后就生成了名为test\_groups的属性组。属性组中属性列表中的属性依然可以通过\_\_ATTR宏来完成。

然后我们就可以创建属性组了：

int sysfs\_create\_groups(struct kobject \*kobj, const struct attribute\_group \*\*groups);

注意这个函数中的groups是双指针，所以这个函数时用来创建多个属性组，groups指向的属性组列表，以NULL为结尾。

如果创建单个属性组，则用如下方法，此时如果还是用ATTRIBUTE\_GROUPS宏初始化属性组，就需要把属性组数组的第一个成员取出来：

int sysfs\_create\_group(struct kobject \*kobj, const struct attribute\_group \*grp);

然后可利用如下接口删除属性组或者属性组列表：

void sysfs\_remove\_groups(struct kobject \*kobj, const struct attribute\_group \*\*groups);

void sysfs\_remove\_group(struct kobject \*kobj, const struct attribute\_group \*grp);

# 示例说明

示例演示了创建目录与文件的方法，分别使用宏以及不使用宏初始化属性与属性组，不使用宏可以更方便的将相关变量组合到一起，更适合面向对象编程。