**Linux设备模型(4)\_sysfs**

**1. 前言**

sysfs是一个基于RAM的文件系统，它和Kobject一起，可以将Kernel的数据结构导出到用户空间，以文件目录结构的形式，提供对这些数据结构（以及数据结构的属性）的访问支持。

sysfs具备文件系统的所有属性，而本文主要侧重其设备模型的特性,因此不会涉及过多的文件系统实现细节，而只介绍sysfs在Linux设备模型中的作用和使用方法。具体包括：

* sysfs和Kobject的关系
* attribute的概念
* sysfs的文件系统操作接口

**2. sysfs和Kobject的关系**

在"[Linux设备模型\_Kobject](http://www.wowotech.net/linux_kenrel/kobject.html)”文章中，有提到过，每一个Kobject，都会对应sysfs中的一个目录。因此在将Kobject添加到Kernel时，create\_dir接口会调用sysfs文件系统的创建目录接口，创建和Kobject对应的目录，相关的代码如下：

1: /\* lib/kobject.c, line 47 \*/

2: static int create\_dir(struct kobject \*kobj)

3: {

4:     int error = 0;

5:     error = sysfs\_create\_dir(kobj);

6:     if (!error) {

7:         error = populate\_dir(kobj);

8:     if (error)

9:         sysfs\_remove\_dir(kobj);

10:     }

11:     return error;

12: }

13:

14: /\* fs/sysfs/dir.c, line 736 \*/

15: \*\*

16: \* sysfs\_create\_dir - create a directory for an object.

17: \* @kobj: object we're creating directory for.

18: \*/

19: int sysfs\_create\_dir(struct kobject \* kobj)

20: {

21:     enum kobj\_ns\_type type;

22:     struct sysfs\_dirent \*parent\_sd, \*sd;

23:     const void \*ns = NULL;

24:     int error = 0;

25:     ...

26: }

**3. attribute**

**3.1 attribute的功能概述**

在sysfs中，为什么会有attribute的概念呢？其实它是对应kobject而言的，指的是kobject的“属性”。我们知道，

sysfs中的目录描述了kobject，而kobject是特定数据类型变量（如struct device）的体现。因此kobject的属性，就是这些变量的属性。它可以是任何东西，名称、一个内部变量、一个字符串等等。而attribute，在sysfs文件系统中是以文件的形式提供的，即：kobject的所有属性，都在它对应的sysfs目录下以文件的形式呈现。这些文件一般是可读、写的，而kernel中定义了这些属性的模块，会根据用户空间的读写操作，记录和返回这些attribute的值。

总结一下：所谓的attibute，就是内核空间和用户空间进行信息交互的一种方法。例如某个driver定义了一个变量，却希望用户空间程序可以修改该变量，以控制driver的运行行为，那么就可以将该变量以sysfs attribute的形式开放出来。

Linux内核中，attribute分为普通的attribute和二进制attribute，如下：

1: /\* include/linux/sysfs.h, line 26 \*/

2: struct attribute {

3:     const char \*name;

4:     umode\_t mode;

5: #ifdef CONFIG\_DEBUG\_LOCK\_ALLOC

6:     bool ignore\_lockdep:1;

7:     struct lock\_class\_key \*key;

8:     struct lock\_class\_key skey;

9: #endif

10: };

11:

12: /\* include/linux/sysfs.h, line 100 \*/

13: struct bin\_attribute {

14:     struct attribute attr;

15:     size\_t size;

16:     void \*private;

17:     ssize\_t (\*read)(struct file \*, struct kobject \*, struct bin\_attribute \*,

18:                     char \*, loff\_t, size\_t);

19:     ssize\_t (\*write)(struct file \*,struct kobject \*, struct bin\_attribute \*,

20:                     char \*, loff\_t, size\_t);

21:     int (\*mmap)(struct file \*, struct kobject \*, struct bin\_attribute \*attr,

22:                     struct vm\_area\_struct \*vma);

23: };

struct attribute为普通的attribute，使用该attribute生成的sysfs文件，只能用字符串的形式读写（后面会说为什么）。而struct bin\_attribute在struct attribute的基础上，增加了read、write等函数，因此它所生成的sysfs文件可以用任何方式读写。

说完基本概念，我们要问两个问题：

Kernel怎么把attribute变成sysfs中的文件呢？

用户空间对sysfs的文件进行的读写操作，怎么传递给Kernel呢？

下面来看看这个过程。

**3.2 attibute文件的创建**

在linux内核中，attibute文件的创建是由fs/sysfs/file.c中sysfs\_create\_file接口完成的，该接口的实现没有什么特殊之处，大多是文件系统相关的操作，和设备模型没有太多的关系，这里先略过不提。

**3.3 attibute文件的read和write**

看到3.1章节struct attribute的原型时，也许我们会犯嘀咕，该结构很简单啊，name表示文件名称，mode表示文件模式，其它的字段都是内核用于debug Kernel Lock的，那文件操作的接口在哪里呢？

不着急，我们去fs/sysfs目录下看看sysfs相关的代码逻辑。

所有的文件系统，都会定义一个struct file\_operations变量，用于描述本文件系统的操作接口，sysfs也不例外：

1: /\* fs/sysfs/file.c, line 472 \*/

2: const struct file\_operations sysfs\_file\_operations = {

3:     .read = sysfs\_read\_file,

4:     .write = sysfs\_write\_file,

5:     .llseek = generic\_file\_llseek,

6:     .open = sysfs\_open\_file,

7:     .release = sysfs\_release,

8:     .poll = sysfs\_poll,

9: };

attribute文件的read操作，会由VFS转到sysfs\_file\_operations的read（也就是sysfs\_read\_file）接口上，让我们大概看一下该接口的处理逻辑。

1: /\* fs/sysfs/file.c, line 127 \*/

2: static ssize\_t

3: sysfs\_read\_file(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*ppos)

4: {

5:     struct sysfs\_buffer \* buffer = file->private\_data;

6:     ssize\_t retval = 0;

7:

8:     mutex\_lock(&buffer->mutex);

9:     if (buffer->needs\_read\_fill || \*ppos == 0) {

10:        retval = fill\_read\_buffer(file->f\_path.dentry,buffer);

11:        if (retval)

12:            goto out;

13:    }

14: ...

15: }

16: /\* fs/sysfs/file.c, line 67 \*/

17: static int fill\_read\_buffer(struct dentry \* dentry, struct sysfs\_buffer \* buffer)

18: {

19:    struct sysfs\_dirent \*attr\_sd = dentry->d\_fsdata;

20:    struct kobject \*kobj = attr\_sd->s\_parent->s\_dir.kobj;

21:    const struct sysfs\_ops \* ops = buffer->ops;

22:    ...

23:    count = ops->show(kobj, attr\_sd->s\_attr.attr, buffer->page);

24:    ...

25: }

read处理看着很简单，sysfs\_read\_file从file指针中取一个私有指针（注：大家可以稍微留一下心，私有数据的概念，在VFS中使用是非常普遍的），转换为一个struct sysfs\_buffer类型的指针，以此为参数（buffer），转身就调用fill\_read\_buffer接口。

而fill\_read\_buffer接口，直接从buffer指针中取出一个struct sysfs\_ops指针，调用该指针的show函数，即完成了文件的read操作。

那么后续呢？当然是由ops->show接口接着处理咯。而具体怎么处理，就是其它模块（例如某个driver）的事了，sysfs不再关心（其实，Linux大多的核心代码，都是只提供架构和机制，具体的实现，也就是苦力，留给那些码农吧！这就是设计的魅力）。

不过还没完，这个struct sysfs\_ops指针哪来的？好吧，我们再看看open(sysfs\_open\_file)接口吧。

1: /\* fs/sysfs/file.c, line 326 \*/

2: static int sysfs\_open\_file(struct inode \*inode, struct file \*file)

3: {

4:     struct sysfs\_dirent \*attr\_sd = file->f\_path.dentry->d\_fsdata;

5:     struct kobject \*kobj = attr\_sd->s\_parent->s\_dir.kobj;

6:     struct sysfs\_buffer \*buffer;

7:     const struct sysfs\_ops \*ops;

8:     int error = -EACCES;

9:

10:    /\* need attr\_sd for attr and ops, its parent for kobj \*/

11:    if (!sysfs\_get\_active(attr\_sd))

12:    return -ENODEV;

13:

14:    /\* every kobject with an attribute needs a ktype assigned \*/

15:    if (kobj->ktype && kobj->ktype->sysfs\_ops)

16:        ops = kobj->ktype->sysfs\_ops;

17:    else {

18:        WARN(1, KERN\_ERR "missing sysfs attribute operations for "

19:            "kobject: %s\n", kobject\_name(kobj));

20:        goto err\_out;

21:    }

22:

23:    ...

24:

25:    buffer = kzalloc(sizeof(struct sysfs\_buffer), GFP\_KERNEL);

26:    if (!buffer)

27:        goto err\_out;

28:

29:    mutex\_init(&buffer->mutex);

30:    buffer->needs\_read\_fill = 1;

31:    buffer->ops = ops;

32:    file->private\_data = buffer;

33:    ...

34: }

哦，原来和ktype有关系。这个指针是从该attribute所从属的kobject中拿的。再去看一下"[Linux设备模型\_Kobject](http://www.wowotech.net/linux_kenrel/kobject.html)”中ktype的定义，还真有一个struct sysfs\_ops的指针。

我们注意一下14行的注释以及其后代码逻辑，如果从属的kobject（就是attribute文件所在的目录）没有ktype，或者没有ktype->sysfs\_ops指针，是不允许它注册任何attribute的！

经过确认后，sysfs\_open\_file从ktype中取出struct sysfs\_ops指针，并在随后的代码逻辑中，分配一个struct sysfs\_buffer类型的指针（buffer），并把struct sysfs\_ops指针保存在其中，随后（注意哦），把buffer指针交给file的private\_data，随后read/write等接口便可以取出使用。嗯！惯用伎俩！

顺便看一下struct sysfs\_ops吧，我想你已经能够猜到了。

1: /\* include/linux/sysfs.h, line 124 \*/

2: struct sysfs\_ops {

3:     ssize\_t (\*show)(struct kobject \*, struct attribute \*,char \*);

4:     ssize\_t (\*store)(struct kobject \*,struct attribute \*,const char \*, size\_t);

5:     const void \*(\*namespace)(struct kobject \*, const struct attribute \*);

6: };

attribute文件的write过程和read类似，这里就不再多说。另外，上面只分析了普通attribute的逻辑，而二进制类型的呢？也类似，去看看fs/sysfs/bin.c吧，这里也不说了。

讲到这里，应该已经结束了，事实却不是如此。上面read/write的数据流，只到kobject（也就是目录）级别哦，而真正需要操作的是attribute（文件）啊！这中间一定还有一层转换！确实，不过又交给其它模块了。 下面我们通过一个例子，来说明如何转换的。

**4. sysfs在设备模型中的应用总结**

让我们通过设备模型class.c中有关sysfs的实现，来总结一下sysfs的应用方式。

首先，在class.c中，定义了Class所需的ktype以及sysfs\_ops类型的变量，如下：

1: /\* drivers/base/class.c, line 86 \*/

2: static const struct sysfs\_ops class\_sysfs\_ops = {

3:     .show = class\_attr\_show,

4:     .store = class\_attr\_store,

5:     .namespace = class\_attr\_namespace,

6: };

7:

8: static struct kobj\_type class\_ktype = {

9:     .sysfs\_ops = &class\_sysfs\_ops,

10:    .release = class\_release,

11:    .child\_ns\_type = class\_child\_ns\_type,

12: };

由前面章节的描述可知，所有class\_type的Kobject下面的attribute文件的读写操作，都会交给class\_attr\_show和class\_attr\_store两个接口处理。以class\_attr\_show为例：

1: /\* drivers/base/class.c, line 24 \*/

2: #define to\_class\_attr(\_attr) container\_of(\_attr, struct class\_attribute, attr)

3:

4: static ssize\_t class\_attr\_show(struct kobject \*kobj, struct attribute \*attr,

5: char \*buf)

6: {

7:     struct class\_attribute \*class\_attr = to\_class\_attr(attr);

8:     struct subsys\_private \*cp = to\_subsys\_private(kobj);

9:     ssize\_t ret = -EIO;

10:

11:    if (class\_attr->show)

12:    ret = class\_attr->show(cp->class, class\_attr, buf);

13:    return ret;

14: }

该接口使用container\_of从struct attribute类型的指针中取得一个class模块的自定义指针：struct class\_attribute，该指针中包含了class模块自身的show和store接口。下面是struct class\_attribute的声明：

1: /\* include/linux/device.h, line 399 \*/

2: struct class\_attribute {

3:     struct attribute attr;

4:     ssize\_t (\*show)(struct class \*class, struct class\_attribute \*attr,

5:                     char \*buf);

6:     ssize\_t (\*store)(struct class \*class, struct class\_attribute \*attr,

7:                     const char \*buf, size\_t count);

8:     const void \*(\*namespace)(struct class \*class,

9:                                 const struct class\_attribute \*attr);

10: };

因此，所有需要使用attribute的模块，都不会直接定义struct attribute变量，而是通过一个自定义的数据结构，该数据结构的一个成员是struct attribute类型的变量，并提供show和store回调函数。然后在该模块ktype所对应的struct sysfs\_ops变量中，实现该本模块整体的show和store函数，并在被调用时，转接到自定义数据结构（struct class\_attribute）中的show和store函数中。这样，每个atrribute文件，实际上对应到一个自定义数据结构变量中了。