本章主要讨论与linux的设备驱动和设备管理的相关的4个内核成分，设备类型，模块，内核对象，sysfs。

**主要内容：**

* 设备类型
* 内核模块
* 内核对象
* sysfs
* 总结

### ****1. 设备类型****

linux中主要由3种类型的设备，分别是：

|  |  |  |  |
| --- | --- | --- | --- |
| **设备类型** | **代表设备** | **特点** | **访问方式** |
| 块设备 | 硬盘，光盘 | 随机访问设备中的内容 | 一般都是把设备挂载为文件系统后再访问 |
| 字符设备 | 键盘，打印机 | 只能顺序访问(一个一个字符或者一个一个字节) | 一般不挂载，直接和设备交互 |
| 网络设备 | 网卡 | 打破了Unix "所有东西都是文件" 的设计原则 | 通过套接字API来访问 |

除了以上3种典型的设备之外，其实Linux中还有一些其他的设备类型，其中见的较多的应该算是"伪设备"。

所谓"伪设备"，其实就是一些虚拟的设备，仅提供访问内核功能而已，没有物理设备与之关联。

典型的"伪设备"就是 /dev/random(内核随机数发生器)， /dev/null(空设备)， /dev/zero(零设备)， /dev/full(满设备)

### ****2. 内核模块****

Linux内核是模块化组成的，内核中的模块可以按需加载，从而保证内核启动时不用加载所有的模块，即减少了内核的大小，也提高了效率。

通过编写内核模块来给内核增加功能或者接口是个很好的方式（既不用重新编译内核，也方便调试和删除）。

#### **2.1 内核模块示例**

内核模块可以带参数也可以不带参数，不带参数的内核模块比较简单。

我之前的几篇随笔中用于测试的例子都是用不带参数的内核模块来实验的。

**2.1.2. 带参数的内核模块**

构造带参数的内核模块其实也不难，内核中已经提供了简单的框架来给我们声明参数。

1. module\_param(name, type, perm) : 定义一个模块参数

+ 参数 name :: 既是用户可见的参数名，也是模块中存放模块参数的变量名

+ 参数 type :: 参数的类型(byte, short, int, uint, long, ulong, charp, bool...) byte型存放在char变量中，bool型存放在int变量中

+ 参数 perm :: 指定模块在 sysfs 文件系统中对应的文件权限(关于 sysfs 的内容后面介绍)

static int stu\_id = 0; // 默认id

module\_param(stu\_id, int, 0644);

2. module\_param\_named(name, variable, type, perm) : 定义一个模块参数，并且参数对内对外的名称不一样

+ 参数 name :: 用户可见的参数名

+ 参数 variable :: 模块中存放模块参数的变量名

+ 参数 type和perm :: 同 module\_param 中的 type 和 perm

static char\* stu\_name\_in = "default name"; // 默认名字

module\_param\_named(stu\_name\_out, stu\_name\_in ,charp, 0644);/\* stu\_name\_out 是对用户开放的名称

\* stu\_name\_in 是内核模块内部使用的名称

\*/

3. module\_param\_string(name, string, len, perm) : 拷贝字符串到指定的字符数组

+ 参数 name :: 用户可见的参数名

+ 参数 string :: 模块中存放模块参数的变量名

+ 参数 len :: string 参数的缓冲区长度

+ 参数 perm :: 同 module\_param 中的 perm

static char str\_in[BUF\_LEN];

module\_param\_string(str\_out, str\_in, BUF\_LEN, 0);/\* perm=0 表示完全禁止 sysfs 项 \*/

4. module\_param\_array(name, type, nump, perm) : 定义数组类型的模块参数

+ 参数 name :: 同 module\_param 中的 name

+ 参数 type :: 同 module\_param 中的 type

+ 参数 nump :: 整型指针，存放数组的长度

+ 参数 perm :: 同 module\_param 中的 perm

#define MAX\_ARR\_LEN 5static int arr\_len;static int arr\_in[MAX\_ARR\_LEN];

module\_param\_array(arr\_in, int, &arr\_len, 0644);

5. module\_param\_array\_named(name, array, type, nump, perm) : 定义数组类型的模块参数,并且数组参数对内对外的名称不一样

+ 参数 name :: 数组参数对外的名称

+ 参数 array :: 数组参数对内的名称

+ 参数 type，nump，perm :: 同 module\_param\_array 中的 type，nump，perm

#define MAX\_ARR\_LEN 5static int arr\_len;static int arr\_in[MAX\_ARR\_LEN];

module\_param\_array\_named(arr\_out, arr\_in, int, &arr\_len, 0644);

6. 参数描述宏

可以通过 MODULE\_PARM\_DESC() 来给内核模块的参数添加一些描述信息。

这些描述信息在编译完内核模块后，可以通过 modinfo  命令查看。

static int stu\_id = 0; // 默认id

module\_param(stu\_id, int, 0644);

MODULE\_PARM\_DESC(stu\_id, "学生ID，默认为 0"); // 这句就是描述内核模块参数 stu\_id 的语句

7. 带参数的内核模块的示例

示例代码：test\_paramed\_km.c

定义了3个内核模块参数，分别是 int型，char\*型，数组型的。

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

MODULE\_LICENSE("Dual BSD/GPL");

struct student

{

int id;

char\* name;

};

static void print\_student(struct student\*);

static int stu\_id = 0; // 默认id

module\_param(stu\_id, int, 0644);

MODULE\_PARM\_DESC(stu\_id, "学生ID，默认为 0");

static char\* stu\_name\_in = "default name"; // 默认名字

module\_param\_named(stu\_name\_out, stu\_name\_in ,charp, 0644);

MODULE\_PARM\_DESC(stu\_name, "学生姓名，默认为 default name");

#define MAX\_ARR\_LEN 5static int arr\_len;static int arr\_in[MAX\_ARR\_LEN];

module\_param\_array\_named(arr\_out, arr\_in, int, &arr\_len, 0644);

MODULE\_PARM\_DESC(arr\_in, "数组参数，默认为空");

static int test\_paramed\_km\_init(void)

{

struct student\* stu1;

int i;

/\* 进入内核模块 \*/

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "test\_paramed\_km is inited!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

// 根据参数生成 struct student 信息

// 如果没有参数就用默认参数

printk(KERN\_ALERT "alloc one student....\n");

stu1 = kmalloc(sizeof(\*stu1), GFP\_KERNEL);

stu1->id = stu\_id;

stu1->name = stu\_name\_in;

print\_student(stu1);

// 模块数组

for (i = 0; i < arr\_len; ++i) {

printk(KERN\_ALERT "arr\_value[%d]: %d\n", i, arr\_in[i]);

}

return 0;

}

static void test\_paramed\_km\_exit(void)

{

/\* 退出内核模块 \*/

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "test\_paramed\_km is exited!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "\n\n\n\n\n");

}

static void print\_student(struct student \*stu)

{

if (stu != NULL)

{

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*student info\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "student id is: %d\n", stu->id);

printk(KERN\_ALERT "student name is: %s\n", stu->name);

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

else

printk(KERN\_ALERT "the student info is null!!\n");

}

module\_init(test\_paramed\_km\_init);

module\_exit(test\_paramed\_km\_exit);

内核模块运行方法：(我的运行环境是 CentOS 6.3 x86\_64)

[root@vbox chap17]# insmod paramed\_km.ko

[root@vbox chap17]# rmmod paramed\_km.ko

[root@vbox chap17]# dmesg | tail -16 <-- 结果中显示2个默认参数，第3个数组参数默认为空，所以不显示\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_paramed\_km is inited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

alloc one student....\*\*\*\*\*\*\*\*\*\*student info\*\*\*\*\*\*\*\*\*\*\*

student id is: 0

student name is: default name\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_paramed\_km is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<-- 3 个参数都被设置

[root@vbox chap17]# insmod paramed\_km.ko stu\_id=100 stu\_name\_out=myname arr\_out=1,2,3,4,5

[root@vbox chap17]# rmmod paramed\_km.ko

[root@vbox chap17]# dmesg | tail -21

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_paramed\_km is inited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

alloc one student....\*\*\*\*\*\*\*\*\*\*student info\*\*\*\*\*\*\*\*\*\*\*

student id is: 100

student name is: myname\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

arr\_value[0]: 1

arr\_value[1]: 2

arr\_value[2]: 3

arr\_value[3]: 4

arr\_value[4]: 5

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_paramed\_km is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#### **2.2 内核模块的位置**

2.2.1.  内核代码外

上面的例子，以及之前博客中内核模块的例子都是把模块代码放在内核之外来运行的。

2.2.2. 内核代码中

内核模块的代码也可以直接放到内核代码树中。

如果你开发了一种驱动，并且希望被加入到内核中，那么，可以在编写驱动的时候就将完成此驱动功能的内核模块加到内核代码树中 driver 的相应位置。

将内核模块加入内核代码树中之后，不需要另外写 Makefile，修改内核代码树中的已有的 Makefile 就行。

比如，写了一个某种字符设备相关的驱动，可以把它加到内核代码的 /drivers/char 下，

同时修改 /drivers/char下的Makefie，仿照里面已有的内容，增加新驱动的编译相关内容即可。

之后，在编译内核的时候会将新的驱动以内核模块的方式编译出来。

#### **2.3 内核模块相关操作**

2.3.1. 模块安装

make modules\_install <-- 把随内核编译出来的模块安装到合适的目录中( /lib/modules/version/kernel )

2.3.2. 模块依赖性

linux中自动生产模块依赖性的命令：

depmod <-- 产生内核依赖关系信息

depmod -A <-- 只为新模块生成依赖信息(速度更快)

2.3.3. 模块的载入

内核模块实验时已经用过：

insmod module.ko

<-- 推荐使用以下的命令， 自动加载依赖的模块modprobe module [module parameters]

2.3.4. 模块的卸载

内核模块实验时已经用过：

rmmod module.ko

<-- 推荐使用以下的命令， 自动卸载依赖的模块modprobe -r module

2.3.5. 模块导出符号表

内核模块被载入后，就动态的加载到内核中，为了能让其他内核模块使用其功能，需要将其中函数导出。

内核模块中导出函数的方法：

EXPORT\_SYMBOL(函数名) <-- 接在要导出的函数后面即可

EXPORT\_SYMBOL\_GPL(函数名) <-- 和EXPORT\_SYMBOL一样，区别在于只对标记为GPL协议的模块可见

内核模块导出符号表 **示例**：

+ 首先编写一个导出函数的模块 module\_A: test\_module\_A.c

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

MODULE\_LICENSE("Dual BSD/GPL");

static int test\_export\_A\_init(void)

{

/\* 进入内核模块 \*/

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "ENTRY test\_export\_A!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "\n\n\n\n\n");

return 0;

}

static void test\_export\_A\_exit(void)

{

/\* 退出内核模块 \*/

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "EXIT test\_export\_A!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "\n\n\n\n\n");

}

/\* 要导出的函数 \*/

int export\_add10(int param)

{

printk(KERN\_ALERT "param from other module is : %d\n", param);

return param + 10;

}

EXPORT\_SYMBOL(export\_add10);

module\_init(test\_export\_A\_init);

module\_exit(test\_export\_A\_exit);

test\_module\_A.c 的 Makefile

# must complile on customize kernel

obj-m += export\_A.o

export\_A-objs := test\_export\_A.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

+ 再编写一个内核模块 module\_B，使用 module\_A 导出的函数 : test\_module\_B.c

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

MODULE\_LICENSE("Dual BSD/GPL");

extern int export\_add10(int); // 这个函数是 module\_A 中实现的

static int test\_export\_B\_init(void)

{

/\* 进入内核模块 \*/

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "ENTRY test\_export\_B!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "\n\n\n\n\n");

/\* 调用 module\_A 导出的函数 \*/

printk(KERN\_ALERT "result from test\_export\_A: %d\n", export\_add10(100));

return 0;

}

static void test\_export\_B\_exit(void)

{

/\* 退出内核模块 \*/

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "EXIT test\_export\_B!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "\n\n\n\n\n");

}

module\_init(test\_export\_B\_init);

module\_exit(test\_export\_B\_exit);

test\_module\_B.c 的 Makefile

# must complile on customize kernel

obj-m += export\_B.o

export\_B-objs := test\_export\_B.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

+ 测试方法

1. 将 test\_export\_A.c 和对应的 Makefile 拷贝到 module\_A 文件夹中

2. 将 test\_export\_B.c 和对应的 Makefile 拷贝到 module\_B 文件夹中

3. 编译 module\_A 中的 test\_export\_A.c

4. 将编译 module\_A 后生成的 Module.symvers 拷贝到 module\_B 文件夹中

5. 编译 module\_B 中的 test\_export\_B.c

6. 先安装 模块A，再安装模块B

7. dmesg 查看log

8. 用 rmmod 卸载模块B 和 模块A (注意卸载顺序，先卸载B再卸载A)

[root@vbox chap17]# ll

total 8

drwxrwxr-x 2 root root 4096 Dec 7 22:14 module\_A

drwxrwxr-x 2 root root 4096 Dec 7 22:14 module\_B

[root@vbox chap17]# ll module\_A

total 8

-rw-r--r-- 1 root root 517 Dec 7 21:58 Makefile-rw-r--r-- 1 root root 893 Dec 7 21:58 test\_export\_A.c

[root@vbox chap17]# ll module\_B

total 8

-rw-r--r-- 1 root root 532 Dec 7 21:58 Makefile-rw-r--r-- 1 root root 830 Dec 7 21:58 test\_export\_B.c

[root@vbox chap17]# cd module\_A/

[root@vbox module\_A]# ll

total 8

-rw-r--r-- 1 root root 517 Dec 7 21:58 Makefile-rw-r--r-- 1 root root 893 Dec 7 21:58 test\_export\_A.c

[root@vbox module\_A]# makemake -C /usr/src/kernels/2.6.32-279.el6.x86\_64 M=/root/chap17/module\_A modulesmake[1]: Entering directory `/usr/src/kernels/2.6.32-279.el6.x86\_64' CC [M] /root/chap17/module\_A/test\_export\_A.o

LD [M] /root/chap17/module\_A/export\_A.o

Building modules, stage 2.

MODPOST 1 modules

CC /root/chap17/module\_A/export\_A.mod.o

LD [M] /root/chap17/module\_A/export\_A.ko.unsigned

NO SIGN [M] /root/chap17/module\_A/export\_A.komake[1]: Leaving directory `/usr/src/kernels/2.6.32-279.el6.x86\_64'rm -rf modules.order .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

[root@vbox module\_A]# ll

total 120

-rw-r--r-- 1 root root 110452 Dec 7 22:31 export\_A.ko-rw-r--r-- 1 root root 517 Dec 7 21:58 Makefile-rw-r--r-- 1 root root 69 Dec 7 22:31 Module.symvers-rw-r--r-- 1 root root 893 Dec 7 21:58 test\_export\_A.c

[root@vbox module\_A]# cd ../module\_B

[root@vbox module\_B]# ll

total 8

-rw-r--r-- 1 root root 532 Dec 7 21:58 Makefile-rw-r--r-- 1 root root 830 Dec 7 21:58 test\_export\_B.c

[root@vbox module\_B]# cp ../module\_A/Module.symvers .

[root@vbox module\_B]# ll

total 12

-rw-r--r-- 1 root root 532 Dec 7 21:58 Makefile-rw-r--r-- 1 root root 69 Dec 7 22:32 Module.symvers-rw-r--r-- 1 root root 830 Dec 7 21:58 test\_export\_B.c

[root@vbox module\_B]# makemake -C /usr/src/kernels/2.6.32-279.el6.x86\_64 M=/root/chap17/module\_B modulesmake[1]: Entering directory `/usr/src/kernels/2.6.32-279.el6.x86\_64' CC [M] /root/chap17/module\_B/test\_export\_B.o

LD [M] /root/chap17/module\_B/export\_B.o

Building modules, stage 2.

MODPOST 1 modules

CC /root/chap17/module\_B/export\_B.mod.o

LD [M] /root/chap17/module\_B/export\_B.ko.unsigned

NO SIGN [M] /root/chap17/module\_B/export\_B.komake[1]: Leaving directory `/usr/src/kernels/2.6.32-279.el6.x86\_64'rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

[root@vbox module\_B]# ll

total 116

-rw-r--r-- 1 root root 108596 Dec 7 22:32 export\_B.ko-rw-r--r-- 1 root root 532 Dec 7 21:58 Makefile-rw-r--r-- 1 root root 830 Dec 7 21:58 test\_export\_B.c

[root@vbox module\_B]# insmod ../module\_A/export\_A.ko

[root@vbox module\_B]# insmod export\_B.ko

[root@vbox module\_B]# dmesg | tail -18

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ENTRY test\_export\_A!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ENTRY test\_export\_B!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

param from other module is : 100

result from test\_export\_A: 110

[root@vbox module\_B]# rmmod export\_B

[root@vbox module\_B]# rmmod export\_A

**注：**1. 必须把编译模块A后生成的 Module.symvers 拷贝到module\_B 中再编译模块B，否在模块B找不到模块A导出的函数

2. 先安装模块A，再安装模块B。

3. 先卸载模块B，再卸载模块A。

4. 安装卸载如果不按照上面的顺序，会有错误提示，大家可以试试看。

5. 我实验的系统是 CentOS6.3 x86\_64

### ****3. 内核对象****

2.6内核中增加了一个引人注目的新特性--统一设备模型(device model)。

统一设备模型的最初动机是为了实现智能的电源管理，linux 内核为了实现智能电源管理，需要建立表示系统中所有设备拓扑关系的树结构，

这样在关闭电源时，可以从树的节点开始关闭。

#### **3.1 kobject 简介**

统一设备模型的核心部分就是 kobject，通过下面对kobject结构体的介绍，可以大致了解它是如何使得各个物理设备能够以树结构的形式组织起来的。

3.1.1. kobject

kobject的定义在 <linux/kobject.h> 中

struct kobject {

const char \*name; /\* kobject 名称 \*/

struct list\_head entry; /\* kobject 链表 \*/

struct kobject \*parent; /\* kobject 的父对象，说明kobject是有层次结构的 \*/

struct kset \*kset; /\* kobject 的集合，接下来有详细介绍 \*/

struct kobj\_type \*ktype; /\* kobject 的类型，接下来有详细介绍 \*/

struct sysfs\_dirent \*sd; /\* 在sysfs中，这个结构体表示kobject的一个inode结构体，sysfs之后也会介绍 \*/

struct kref kref; /\* 提供 kobject 的引用计数 \*/

/\* 一些标志位 \*/

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;

};

kobject 本身不代表什么实际的内容，一般都是嵌在其他数据结构中来发挥作用。（感觉有点像内核数据结构链表的节点）

比如 <linux/cdev.h> 中的 struct cdev （表示字符设备的struct）

struct cdev {

struct kobject kobj; /\* 嵌在 cdev 中的kobject \*/

struct module \*owner;

const struct file\_operations \*ops;

struct list\_head list;

dev\_t dev;

unsigned int count;

};

cdev中嵌入了kobject之后，就可以通过 cdev->kboj.parent 建立cdev之间的层次关系，通过 cdev->kobj.entry 获取关联的所有cdev设备等。

总之，嵌入了kobject之后，cdev设备之间就有了树结构关系，cdev设备和其他设备之间也有可层次关系。

3.1.2. ktype

ktype是为了描述一族的kobject所具有的普遍属性，也就是将这一族的kobject的属性统一定义一下，避免每个kobject分别定义。

（感觉有点像面向对象语言中的抽象类或者接口）

ktype的定义很简单，参见<linux/kobject.h>

struct kobj\_type {

void (\*release)(struct kobject \*kobj); /\* kobject的引用计数降到0时触发的析构函数，负责释放和清理内存的工作 \*/

struct sysfs\_ops \*sysfs\_ops; /\* sysfs操作相关的函数 \*/

struct attribute \*\*default\_attrs; /\* kobject 相关的默认属性 \*/

};

3.1.3. kset

kset是kobject对象的集合体，可以所有相关的kobject置于一个kset之中，比如所有“块设备”可以放在一个表示块设备的kset中。

kset的定义也不复杂，参见 <linux/kobject.h>

struct kset {

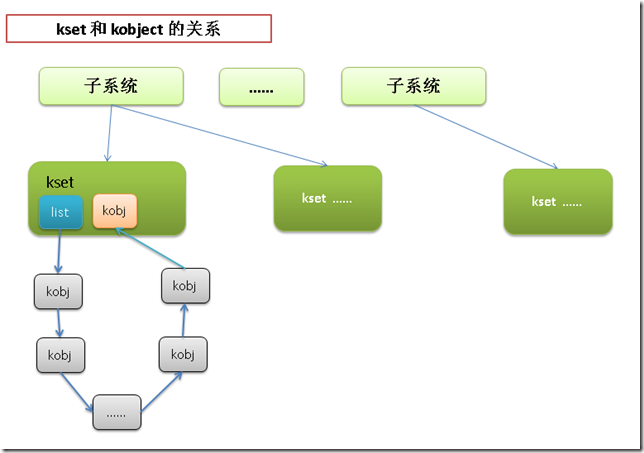
struct list\_head list; /\* 表示kset中所有kobject的链表 \*/

spinlock\_t list\_lock; /\* 用于保护 list 的自旋锁\*/

struct kobject kobj; /\* kset中嵌入的一个kobject，使得kset也可以表现的像一样kobject一样\*/

struct kset\_uevent\_ops \*uevent\_ops; /\* 处理kset中kobject的热插拔事件 提供了与用户空间热插拔进行通信的机制 \*/

};

[](https://images0.cnblogs.com/blog/83005/201312/24215225-7ae8cf55c80c4e8ebecf988b05f1424e.png)

3.1.4. kobject，ktype和kset之间的关系

这3个概念中，kobject是最基本的。kset和ktype是为了将kobject进行分类，以便将共通的处理集中处理，从而减少代码量，也增加维护性。

这里kset和ktype都是为了将kobject进行分类，为什么会有2中分类呢？

从整个内核的代码来看，其实kset的数量是多于ktype的数量的，同一种ktype的kobject可以位于不同的kset中。

做个不是很恰当的比喻，如果把kobject比作一个人的话，kset相当于一个一个国家，ktype则相当于人种(比如黄种人，白种人等等)。

人种的类型只有少数几个，但是国家确有很多，人种的目的是描述一群人的共通属性，而国家的目地则是为了管理一群人。

同样，ktype侧重于描述，kset侧重于管理。

3.1.5. kref

kref记录kobject被引用的次数，当引用计数降到0的时候，则执行release函数释放相关资源。

kref的定义参见：<linux/kref.h>

struct kref {

atomic\_t refcount; /\* 只有一个表示引用计数的属性，atomic\_t 类型表示对它的访问是原子操作 \*/

};

void kref\_set(struct kref \*kref, int num); /\* 设置引用计数的值 \*/

void kref\_init(struct kref \*kref); /\* 初始化引用计数 \*/

void kref\_get(struct kref \*kref); /\* 增加引用计数 +1 \*/

int kref\_put(struct kref \*kref, void (\*release) (struct kref \*kref)); /\* 减少引用计数 -1 当减少到0时，释放相应资源 \*/

上面这些函数的具体实现可以参考内核代码  lib/kref.c

#### **3.2 kobject 操作**

kobject的相关都在 <linux/kobject.h> 中定义了，主要由以下一些：

extern void kobject\_init(struct kobject \*kobj, struct kobj\_type \*ktype); /\* 初始化一个kobject，设置它是哪种ktype \*/

extern int \_\_must\_check kobject\_add(struct kobject \*kobj,

struct kobject \*parent,

const char \*fmt, ...); /\* 设置此kobject的parent，将此kobject加入到现有对象层次结构中 \*/

extern int \_\_must\_check kobject\_init\_and\_add(struct kobject \*kobj,

struct kobj\_type \*ktype,

struct kobject \*parent,

const char \*fmt, ...); /\* 初始化kobject，完成kobject\_add 函数的功能\*/

extern void kobject\_del(struct kobject \*kobj); /\* 将此kobject从现有对象层次结构中取消 \*/

extern struct kobject \* \_\_must\_check kobject\_create(void); /\* 创建一个kobject，比kobject\_init更常用 \*/

extern struct kobject \* \_\_must\_check kobject\_create\_and\_add(const char \*name,

struct kobject \*parent); /\* 创建一个kobject，并将其加入到现有对象层次结构中 \*/

extern int \_\_must\_check kobject\_rename(struct kobject \*, const char \*new\_name); /\* 改变kobject的名称 \*/

extern int \_\_must\_check kobject\_move(struct kobject \*, struct kobject \*); /\* 给kobject设置新的parent \*/

extern struct kobject \*kobject\_get(struct kobject \*kobj); /\* 增加kobject的引用计数 +1 \*/

extern void kobject\_put(struct kobject \*kobj); /\* 减少kobject的引用计数 -1 \*/

extern char \*kobject\_get\_path(struct kobject \*kobj, gfp\_t flag); /\* 生成并返回与给定的一个kobj和kset相关联的路径 \*/

上面这些函数的具体实现可以参考内核代码  lib/kobject.c

### ****4. sysfs****

sysfs是一个处于内存中的虚拟文件系统，它提供了kobject对象层次结构的视图。

可以用下面这个命令来查看 /sys 的结构

tree /sys # 显示所有目录和文件

或者

tree -L 1 /sys # 只显示一层目录

#### **4.1 sysfs中的kobject**

既然sysfs是kobject的视图，那么内核中肯定提供了在sysfs中操作kobject的API。

kobject结构体中与sysfs关联的字段就是 「struct sysfs\_dirent    \*sd; 」这是一个目录项结构，它表示kobject在sysfs中的位置。

关于目录项，可以参考：[《Linux内核设计与实现》读书笔记（十三）- 虚拟文件系统](http://www.cnblogs.com/wang_yb/p/3144291.html" \t "https://www.cnblogs.com/wang_yb/p/_blank)

4.1.1. sysfs中添加和删除kobject非常简单，就是上面介绍的 kobject操作中提到的

extern int \_\_must\_check kobject\_add(struct kobject \*kobj,

struct kobject \*parent,

const char \*fmt, ...); /\* 设置此kobject的parent，将此kobject加入到现有对象层次结构中 \*/extern int \_\_must\_check kobject\_init\_and\_add(struct kobject \*kobj,

struct kobj\_type \*ktype,

struct kobject \*parent,

const char \*fmt, ...); /\* 初始化kobject，完成kobject\_add 函数的功能\*/

extern void kobject\_del(struct kobject \*kobj); /\* 将此kobject从现有对象层次结构中取消 \*/

... ...等等

添加了kobject之后，只会增加文件夹，不会增加文件。

因为kobject在sysfs中就是映射成一个文件夹。

添加删除kobject的示例代码如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : test\_kobject.c

\* @author : wangyubin

\* @date : Tue Dec 24 09:49:53 2013

\*

\* @brief : 测试 kobject的创建和删除

\* history : init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/kobject.h>

MODULE\_LICENSE("Dual BSD/GPL");

struct kobject\* kobj = NULL;

static int test\_kobject\_init(void)

{

/\* 初始化kobject，并加入到sysfs中 \*/

kobj = kobject\_create\_and\_add("test\_kobject", NULL);

/\* 进入内核模块 \*/

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "test\_kobject is inited!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

return 0;

}

static void test\_kobject\_exit(void)

{

/\* 如果 kobj 不为空，则将其从sysfs中删除 \*/

if (kobj != NULL)

kobject\_del(kobj);

/\* 退出内核模块 \*/

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "test\_kobject is exited!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "\n\n\n\n\n");

}

module\_init(test\_kobject\_init);

module\_exit(test\_kobject\_exit);

对应的Makefile

# must complile on customize kernel

obj-m += mykobject.o

mykobject-objs := test\_kobject.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

测试方法：(我使用的测试系统是：Centos6.5 x86)

[root@localhost test\_kobject]# insmod mykobject.ko <-- 安装内核模块

[root@localhost test\_kobject]# ll /sys/ <-- 安装后，sysfs中多了一个文件夹 test\_kobject

....

[root@localhost test\_kobject]# rmmod mykobject.ko <-- 卸载内核模块

[root@localhost test\_kobject]# ll /sys/ <-- 卸载后，sysfs 中的文件夹 test\_kobject 也消失了

....

4.1.2. sysfs中添加文件

kobject是映射成sysfs中的目录，那sysfs中的文件是什么呢？

其实sysfs中的文件就是kobject的属性，属性的来源有2个：

+ 默认属性 :: kobject所关联的ktype中的 default\_attrs 字段

默认属性 default\_attrs 的类型是结构体 struct attribute， 定义在 <linux/sysfs.h>

struct attribute {

const char \*name; /\* sysfs文件树中的文件名 \*/

struct module \*owner; /\* x86体系结构中已经不再继续使用了，可能在其他体系结构中还会使用 \*/

mode\_t mode; /\* sysfs中该文件的权限 \*/

};

ktype中的 default\_attrs 字段(即默认属性)描述了sysfs中的文件，还有一个字段 sysfs\_ops 则描述了如何使用默认属性。

struct sysfs\_ops 的定义也在 <linux/sysfs.h>

struct sysfs\_ops {

/\* 在读sysfs文件时该方法被调用 \*/

ssize\_t (\*show)(struct kobject \*kobj, struct attribute \*attr,char \*buffer);

/\* 在写sysfs文件时该方法被调用 \*/

ssize\_t (\*store)(struct kobject \*kobj,struct attribute \*attr,const char \*buffer, size\_t size);

};

show 方法在读取sysfs中文件时调用，它会拷贝attr提供的属性到buffer指定的缓冲区

store  方法在写sysfs中文件时调用，它会从buffer中读取size字节的数据到attr提供的属性中

增加默认属性的示例代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : test\_kobject\_default\_attr.c

\* @author : wangyubin

\* @date : Tue Dec 24 10:28:09 2013

\*

\* @brief : 测试 kobject 的默认属性的创建和删除

\* history : init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include<linux/init.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/kobject.h>

#include<linux/sysfs.h>

MODULE\_LICENSE("Dual BSD/GPL");

static void myobj\_release(struct kobject\*);static ssize\_t my\_show(struct kobject \*, struct attribute \*, char \*);static ssize\_t my\_store(struct kobject \*, struct attribute \*, const char \*, size\_t);

/\* 自定义的结构体，2个属性，并且嵌入了kobject \*/

struct my\_kobj

{

int ival;

char\* cname;

struct kobject kobj;

};

static struct my\_kobj \*myobj = NULL;

/\* my\_kobj 的属性 ival 所对应的sysfs中的文件，文件名 val \*/

static struct attribute val\_attr = {

.name = "val",

.owner = NULL,

.mode = 0666,

};

/\* my\_kobj 的属性 cname 所对应的sysfs中的文件，文件名 name \*/

static struct attribute name\_attr = {

.name = "name",

.owner = NULL,

.mode = 0666,

};

static int test\_kobject\_default\_attr\_init(void)

{

struct attribute \*myattrs[] = {NULL, NULL, NULL};

struct sysfs\_ops \*myops = NULL;

struct kobj\_type \*mytype = NULL;

/\* 初始化 myobj \*/

myobj = kmalloc(sizeof(struct my\_kobj), GFP\_KERNEL);

if (myobj == NULL)

return -ENOMEM;

/\* 配置文件 val 的默认值 \*/

myobj->ival = 100;

myobj->cname = "test";

/\* 初始化 ktype \*/

mytype = kmalloc(sizeof(struct kobj\_type), GFP\_KERNEL);

if (mytype == NULL)

return -ENOMEM;

/\* 增加2个默认属性文件 \*/

myattrs[0] = &val\_attr;

myattrs[1] = &name\_attr;

/\* 初始化ktype的默认属性和析构函数 \*/

mytype->release = myobj\_release;

mytype->default\_attrs = myattrs;

/\* 初始化ktype中的 sysfs \*/

myops = kmalloc(sizeof(struct sysfs\_ops), GFP\_KERNEL);

if (myops == NULL)

return -ENOMEM;

myops->show = my\_show;

myops->store = my\_store;

mytype->sysfs\_ops = myops;

/\* 初始化kobject，并加入到sysfs中 \*/

memset(&myobj->kobj, 0, sizeof(struct kobject)); /\* 这一步非常重要，没有这一步init kobject会失败 \*/

if (kobject\_init\_and\_add(&myobj->kobj, mytype, NULL, "test\_kobj\_default\_attr"))

kobject\_put(&myobj->kobj);

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "test\_kobject\_default\_attr is inited!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

return 0;

}

static void test\_kobject\_default\_attr\_exit(void)

{

kobject\_del(&myobj->kobj);

kfree(myobj);

/\* 退出内核模块 \*/

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "test\_kobject\_default\_attr is exited!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "\n\n\n\n\n");

}

static void myobj\_release(struct kobject \*kobj)

{

printk(KERN\_ALERT, "release kobject");

kobject\_del(kobj);

}

/\* 读取属性文件 val 或者name时会执行此函数 \*/

static ssize\_t my\_show(struct kobject \*kboj, struct attribute \*attr, char \*buf)

{

printk(KERN\_ALERT "SHOW -- attr-name: [%s]\n", attr->name);

if (strcmp(attr->name, "val") == 0)

return sprintf(buf, "%d\n", myobj->ival);

else

return sprintf(buf, "%s\n", myobj->cname);

}

/\* 写入属性文件 val 或者name时会执行此函数 \*/

static ssize\_t my\_store(struct kobject \*kobj, struct attribute \*attr, const char \*buf, size\_t len)

{

printk(KERN\_ALERT "STORE -- attr-name: [%s]\n", attr->name);

if (strcmp(attr->name, "val") == 0)

sscanf(buf, "%d\n", &myobj->ival);

else

sscanf(buf, "%s\n", myobj->cname);

return len;

}

module\_init(test\_kobject\_default\_attr\_init);

module\_exit(test\_kobject\_default\_attr\_exit);

对应的Makefile如下：

# must complile on customize kernel

obj-m += mykobject\_with\_default\_attr.o

mykobject\_with\_default\_attr-objs := test\_kobject\_default\_attr.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

测试方法：(我使用的测试系统是：Centos6.5 x86)

############################ 编译 ########################################################

[root@localhost test\_kobject\_defalt\_attr]# ll

total 8

-rw-r--r-- 1 root root 582 Dec 24 15:02 Makefile-rw-r--r-- 1 root root 4032 Dec 24 16:58 test\_kobject\_default\_attr.c

[root@localhost test\_kobject\_defalt\_attr]# makemake -C /usr/src/kernels/2.6.32-431.el6.i686 M=/home/wyb/chap17/test\_kobject\_defalt\_attr modulesmake[1]: Entering directory `/usr/src/kernels/2.6.32-431.el6.i686' CC [M] /home/wyb/chap17/test\_kobject\_defalt\_attr/test\_kobject\_default\_attr.o/home/wyb/chap17/test\_kobject\_defalt\_attr/test\_kobject\_default\_attr.c: In function ‘myobj\_release’:/home/wyb/chap17/test\_kobject\_defalt\_attr/test\_kobject\_default\_attr.c:109: warning: too many arguments for format

LD [M] /home/wyb/chap17/test\_kobject\_defalt\_attr/mykobject\_with\_default\_attr.o

Building modules, stage 2.

MODPOST 1 modules

CC /home/wyb/chap17/test\_kobject\_defalt\_attr/mykobject\_with\_default\_attr.mod.o

LD [M] /home/wyb/chap17/test\_kobject\_defalt\_attr/mykobject\_with\_default\_attr.ko.unsigned

NO SIGN [M] /home/wyb/chap17/test\_kobject\_defalt\_attr/mykobject\_with\_default\_attr.komake[1]: Leaving directory `/usr/src/kernels/2.6.32-431.el6.i686'rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

[root@localhost test\_kobject\_defalt\_attr]# ll

total 104

-rw-r--r-- 1 root root 582 Dec 24 15:02 Makefile-rw-r--r-- 1 root root 96805 Dec 24 16:58 mykobject\_with\_default\_attr.ko-rw-r--r-- 1 root root 4032 Dec 24 16:58 test\_kobject\_default\_attr.c

############################ 安装 ########################################################

[root@localhost test\_kobject\_defalt\_attr]# insmod mykobject\_with\_default\_attr.ko

[root@localhost test\_kobject\_defalt\_attr]# ll /sys/ <-- kobject对应的文件夹

total 0

drwxr-xr-x 2 root root 0 Dec 24 15:50 block

drwxr-xr-x 17 root root 0 Dec 24 15:50 bus

drwxr-xr-x 40 root root 0 Dec 24 15:50 class

drwxr-xr-x 4 root root 0 Dec 24 15:50 dev

drwxr-xr-x 12 root root 0 Dec 24 15:50 devices

drwxr-xr-x 4 root root 0 Dec 24 15:50 firmware

drwxr-xr-x 3 root root 0 Dec 24 15:50 fs

drwxr-xr-x 2 root root 0 Dec 24 16:06 hypervisor

drwxr-xr-x 5 root root 0 Dec 24 15:50 kernel

drwxr-xr-x 85 root root 0 Dec 24 16:59 module

drwxr-xr-x 2 root root 0 Dec 24 16:06 power

drwxr-xr-x 2 root root 0 Dec 24 16:59 test\_kobj\_default\_attr

[root@localhost test\_kobject\_defalt\_attr]# ll /sys/test\_kobj\_default\_attr/ <-- kobject的2个属性文件

total 0

-rw-rw-rw- 1 root root 4096 Dec 24 16:59 name-rw-rw-rw- 1 root root 4096 Dec 24 16:59 val

[root@localhost test\_kobject\_defalt\_attr]# dmesg <-- dmesg 中只有初始化的信息\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_kobject\_default\_attr is inited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

############################ 读取属性文件 ###############################################

[root@localhost test\_kobject\_defalt\_attr]# cat /sys/test\_kobj\_default\_attr/val <-- 属性值就是我们在测试代码中输入的值

100

[root@localhost test\_kobject\_defalt\_attr]# cat /sys/test\_kobj\_default\_attr/name <-- 属性值就是我们在测试代码中输入的值

test

[root@localhost test\_kobject\_defalt\_attr]# dmesg <-- dmesg 中多了2条读取属性文件的log

SHOW -- attr-name: [val]

SHOW -- attr-name: [name]

############################ 写入属性文件 ################################################

[root@localhost test\_kobject\_defalt\_attr]# echo "200" > /sys/test\_kobj\_default\_attr/val <-- val文件中写入 200

[root@localhost test\_kobject\_defalt\_attr]# echo "abcdefg" > /sys/test\_kobj\_default\_attr/name <-- name文件中写入 adcdefg

[root@localhost test\_kobject\_defalt\_attr]# dmesg <-- dmesg 中又多了2条写入属性文件的log

STORE -- attr-name: [val]

STORE -- attr-name: [name]

[root@localhost test\_kobject\_defalt\_attr]# cat /sys/test\_kobj\_default\_attr/val <-- 再次查看 val文件中的值，已变为200200

[root@localhost test\_kobject\_defalt\_attr]# cat /sys/test\_kobj\_default\_attr/name <-- 再次查看 name文件中的值，已变为abcdefg

abcdefg

############################ 卸载 ########################################################

[root@localhost test\_kobject\_defalt\_attr]# rmmod mykobject\_with\_default\_attr.ko

4.1.3. sysfs相关约定

为了保持sysfs的干净和直观，在内核开发中涉及到sysfs相关内容时，需要注意以下几点：

+ sysfs属性保证每个文件只导出一个值，该值为文本形式并且可以映射为简单的C类型

+ sysfs中要以一个清晰的层次组织数据

+ sysfs提供内核到用户空间的服务

#### **4.2 基于sysfs的内核事件**

内核事件层也是利用kobject和sysfs来实现的，用户空间通过监控sysfs中kobject的属性的变化来异步的捕获内核中kobject发出的信号。

用户空间可以通过一种netlink的机制来获取内核事件。

内核空间向用户空间发送信号使用 kobject\_uevent() 函数，具体参见： <linux/kobject.h>

int kobject\_uevent(struct kobject \*kobj, enum kobject\_action action);

下面用个小例子演示一些内核事件的实现原理：

4.2.1. 内核模块安装或者删除时，会发送 KOBJ\_ADD 或者 KOBJ\_REMOVE 的消息

内核模块的代码就用上面最简单的那个例子 test\_kobject.c 的代码即可

4.2.2. 用户态程序: 通过 netlink机制来接收 kobject 的事件通知

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : test\_netlink\_client.c

\* @author : wangyubin

\* @date : Tue Dec 24 19:48:54 2013

\*

\* @brief : 通过 netlink机制接收kobject发出的信号

\* history : init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <sys/types.h>

#include <asm/types.h>

#include <sys/socket.h>

#include <linux/netlink.h>

void MonitorNetlinkUevent()

{

int sockfd;

struct sockaddr\_nl sa;

int len;

char buf[4096];

struct iovec iov;

struct msghdr msg;

int i;

memset(&sa,0,sizeof(sa));

sa.nl\_family = AF\_NETLINK;

sa.nl\_groups = NETLINK\_KOBJECT\_UEVENT;

sa.nl\_pid = 0;//getpid(); both is ok

memset(&msg,0,sizeof(msg));

iov.iov\_base = (void \*)buf;

iov.iov\_len = sizeof(buf);

msg.msg\_name = (void \*)&sa;

msg.msg\_namelen = sizeof(sa);

msg.msg\_iov = &iov;

msg.msg\_iovlen = 1;

sockfd = socket(AF\_NETLINK, SOCK\_RAW, NETLINK\_KOBJECT\_UEVENT);

if(sockfd == -1)

printf("socket creating failed:%s\n",strerror(errno));

if(bind(sockfd,(struct sockaddr \*)&sa,sizeof(sa)) == -1)

printf("bind error:%s\n", strerror(errno));

while(1) {

memset(buf, 0, sizeof(buf));

len=recvmsg(sockfd, &msg, 0);

if(len < 0){}

//printf("receive error\n");

else if(len < 32||len > sizeof(buf))

printf("invalid message");

for(i=0; i<len; i++)

if(\*(buf+i) == '\0')

buf[i] = '\n';

printf("received %d bytes\n%s\n", len, buf);

}

}

int main(int argc, char \*argv[])

{

MonitorNetlinkUevent();

return 0;

}

**注**：代码是拷贝的 [《Linux设备节点创建》内核kobject上报uevent过滤规则](http://blog.csdn.net/tankai19880619/article/details/11776589" \t "https://www.cnblogs.com/wang_yb/p/_blank) 中的用户态程序部分

4.2.3. 测试方法：(我使用的测试系统是：Centos6.5 x86)

############################ 编译并启动用户态程序 （窗口1）###################################

[root@localhost test\_kobject\_event]# ll

total 4

-rw-r--r-- 1 root root 1846 Dec 24 20:36 test\_netlink\_client.c

[root@localhost test\_kobject\_event]# gcc -o test\_netlink\_client test\_netlink\_client.c <-- 编译用户态程序

[root@localhost test\_kobject\_event]# ./test\_netlink\_client <-- 启动后等待内核kobject的事件到来

############################ 安装内核模块并查看用户态程序输出 （窗口2）#######################

[root@localhost test\_kobject]# insmod mykobject.ko <-- 在窗口2中安装内核模块，窗口1中会接收到 KOBJ\_ADD 信号

############################ 卸载内核模块并查看用户态程序输出 （窗口2）#######################

[root@localhost test\_kobject]# rmmod mykobject.ko <-- 在窗口2中安装内核模块，窗口1中会接收到 KOBJ\_REMOVE 信号

### ****5. 总结****

kobject加sysfs给内核带来的好处不是三言两语能够说得清楚的，上面的例子也只是通过简单的使用来直观的感受一下kobject，例子本身没有什么实际意义。

最后一个例子中使用了 netlink机制，我在之前的项目中使用过netlink来监控系统进程的I/O信息，对netlink有一些初步的了解。

但是例子中只是用来获取一下 kobject的事件而已，这里主要为了说明kobject，netlink的相关内容以后有机会再补充。