内核模块

内核模块的编译

编写一个测试模块，只包含了最简单的模块加载和卸载

```
#include <linux/module.h>        /* Needed by all modules */
#include <linux/kernel.h>          /* Needed for KERN_INFO */

int init_module(void)
{
    printk(KERN_INFO "Hello World!\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye!\n");
}

MODULE_LICENSE("GPL");
```

接下来通过makefile规则，使用make编译

```
TARGET = helloworld

KDIR = /usr/src/linux-headers-5.4.0-150-generic

PWD = $(shell pwd)

obj-m += $(TARGET).o

default:
	make -C $(KDIR) M=$(PWD) modules
```

在编译好后会生成.ko文件，这个文件就是要被加载的内核模块文件，是elf格式

接下来使用insmod helloworld.ko加载到内核中

如果需要移除模块，rmmod helloworld.ko即可

内核模块传参数

linux内核中，可以借助module_param函数完成参数传递

编译好之后，可以通过insmod传参

```
insmod ./simple.ko irq=44 devname=simpdev debug=1
```

```
[ 9320.513377] Disabling lock debugging due to kernel taint
[ 9406.889494] Bye!
[10895.101762] hello... irq=10 name=simpdev debug=0
[10915.558126] bye... irq=10 name=simpdev debug=0
[10939.182191] hello... irq=44 name=simpdev debug=1
maliang@ubuntu:~/study/module_para$
```

此时，模块中的参数就会变成我们传入的参数

同时也可以在不重新加载模块的情况下，更改模块的参数，在sys/module/模块名/parameters下有我们创建的所有参数名，我们可以在模块运行时，通过echo向参数重新写入

```
maliang@ubuntu:/sys/module/para/parameters$ sudo sh -c 'echo "0" > /sys/module/para/parameters/debug'
```

```
[11761.652023] hello... irq=44 name=simpdev debug=1
[11793.517418] bye... irq=44 name=simpdev debug=0
```

内核模块符号导出

在内核模块中，a模块想让b模块使用自己的函数，就可以将自己的函数符号导出，让b使用,使用EXPORT_SYMBOL就可将符号导出

```
int add(int a,int b)
{
        return a+b;
}

EXPORT_SYMBOL(add);
```

当a模块的函数被b模块调用时，我们就可以说b模块依赖a模块，在两个模块同时被加载进内核时，a模块不可以被卸载，因为此时b模块还在依赖a模块，如果要卸载a模块就要先卸载b模块。

符号导出时建立在内核空间是所有进程共享一个内核空间上的。

字符设备驱动

字符设备驱动抽象

在LINUX中，内核将字符设备抽象成一个具体的数据结构（`struct cdev`），就是字符设备对象，这个对象记录了字符设备的设备号、内核对象，字符设备的open，read，write，close等（file_operations）。当我们要添加一个字符设备时，创建一个文件（设备节点）绑定对象的cdev。

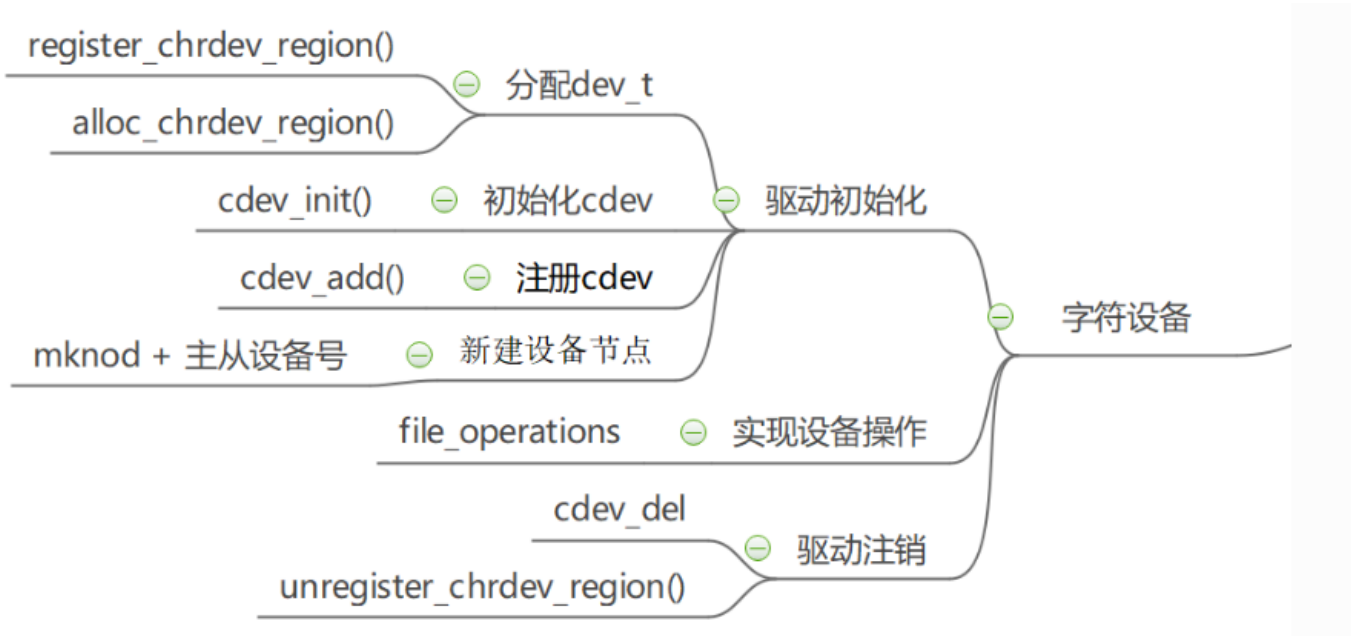
```
<include/linux/cdev.h>

struct cdev {
    struct kobject kobj;                    //内嵌的内核对象.
    struct module *owner;                   //该字符设备所在的内核模块的对象指针.
    const struct file_operations *ops;      //该结构描述了字符设备所能实现的方法，是极为关
键的一个结构体.
    struct list_head list;                  //用来将已经向内核注册的所有字符设备形成链表.
    dev_t dev;                              //字符设备的设备号，由主设备号和次设备号构成.
    unsigned int count;                     //隶属于同一主设备号的次设备号的个数.
};
```

设备号

字符设备或者块设备都会有一个主设备号和此设备号，主设备号用来表示一个特定的的驱动程序，此设备号用来标识使用该驱动程序的其他设备。

在linux中，设备号用 dev_t 类型的变量进行标识，这是一个 32位 无符号整数，dev_t的高12位表示主设备号，低20位表示次设备号。

misc机制



在linux驱动中，驱动会被分为几类，例如input，tty等，当无法确定驱动是说明类型时，可以使用杂项设备-misc。misc设备的主设备号是10，不同的杂项设备通过次设备号区分

misc设备的结构体

```
struct miscdevice  {

    int minor;  //指定次设备号
    const char *name;   //名字
    const struct file_operations *fops; //文件操作
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const struct attribute_group **groups;
    const char *nodename;
    umode_t mode;
};
```

在创建一个miscdevice结构体时，需要我们指定minor，name，fops这三个成员

```
static struct miscdevice misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = CHAR_NAME,
    .fops = &drv_fops,
};
```

fops就是文件的具体操作

```
static const struct file_operations drv_fops = {
    .owner = THIS_MODULE,
    .open = drv_open,
    .release = drv_release,
    .read = drv_read,
    .write = drv_write
};
```

file_operations结构体：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    ...............
} __randomize_layout;
```

read、write中：

```
copy_from_user(void *to, const void __user *from, unsigned long n)
copy_to_user(void __user *to, const void *from, unsigned long n)
```

在向内核中注册设备时只需要使用misc_register(&misc_device)，将我们自己定义的misc_device传入即可，在卸载设备时使用misc_deregister(&misc_device);

补充：

inode结构体

```
struct inode {
    ......
    dev_t                   i_rdev;
    struct file_operations  *i_fop;
    ......
    union {
        struct pipe_inode_info *i_pipe;  /* linux内核管道 */
        struct block_device    *i_bdev;  /* 如果这是块设备，则设置并使用 */
        struct cdev            *i_cdev;  /* 如果这是字符设备，则设置并使用 */
        char                   *i_link;
        unsigned               i_dir_seq;
    };
    ......
} __randomize_layout;
```

IO

kfifo

kfifo是内核里面的一个First In First Out数据结构，它采用环形循环队列的数据结构来实现；它提供一个无边界的字节流服务，最重要的一点是，它使用并行无锁编程技术，即当它用于只有一个入队线程和一个出队线程的场情时，两个线程可以并发操作，而不需要任何加锁行为，就可以保证kfifo的线程安全，在后面的io中我们是kfifo来写数据和读数据

无阻塞IO

无阻塞io，就是读数据时没有数据直接返回，不等待数据。在内核层里，我们可以先获取kfifo中是否有数据，如果没有数据返回-1，如果有数据就从kfifo中拷贝数据到用户层。

在写数据时从用户层向kfifo中写入，在读取时从kfifo中读出，在读出时，先判断一下kfifo里有没有数据，没有数据的话通过返回值告诉用户层没有数据

阻塞IO

阻塞IO通过wait_queue实现

wait_queue的定义和初始化

```
wait_queue_head_t wq;
DECLARE_WAIT_QUEUE_HEAD(wq);
```

在读取时阻塞等待wait_event_interruptible(wq,data_flag);将当前线程放入wq中，如果被唤醒且data_flag为真时执行下面的操作

```
static ssize_t drv_read(struct file *file, char __user *buf, size_t count, loff_t
*ppos)
{
    int ret;
    unsigned int copied_count=0;
    int data_len = kfifo_len(&my_kfifo);
    if(data_len==0)
    {
        return -1;
    }
    wait_event_interruptible(wq,data_flag);
    data_flag=0;
    ret = kfifo_to_user(&my_kfifo, buf,  count, &copied_count);
        if(ret != 0)
    {
            return -EIO;
       }
    printk("read success\r\n");
    return 0;
}
```

在写入时唤醒wake_up_interruptible(&wq)，并置标志位为1

```
static ssize_t drv_write(struct file *file, const char __user *buf, size_t count,
loff_t *f_pos)
{
    int ret;
    unsigned int copied_count=0;
    ret = kfifo_from_user(&my_kfifo, buf, count, &copied_count);
        if(ret != 0)
    {
            return -EIO;
        }
    printk("write success\r\n");
    wake_up_interruptible(&wq);
    data_flag=1;
    return count;
}
```

多路复用poll

poll是实现多路复用IO的一种方法，poll的结构体是

```
struct pollfd{
    int fd;         // 感兴趣fd
    short events;  // 监听事件
    short revents; // 就绪事件
};
```

fd是我们要监听的文件，event是我们要监听的事件，revents是内核中poll返回的事件。

在用户态：调用ret=poll(fds,1,5000);（poll的返回值：

返回值小于0, 表示出错。

返回值等于0, 表示poll函数等待超时。

返回值大于0, 表示poll由于监听的文件描述符就绪而返回。数值为就绪的文件数量）

当ret==1 并且revents为我们想要的事件时，就说明可以读取了

```
fds[0].fd=fd;
fds[0].events=POLLIN;
ret=poll(fds,1,5000);
if((ret==1) && (fds[0].revents & POLLIN))
            {
                int r = read(fd, buffer, 32);
                printf("%s\r\n",buffer);
            }
```

在内核层：

```
static unsigned int drv_poll(struct file *fp, poll_table * wait)
{
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    poll_wait(fp, &wq, wait);
    if( kfifo_len(&my_kfifo)!=0)    return POLLIN;
    else return POLLRDNORM;
}
```

poll_wait等待wq被唤醒（wq仍然是被write唤醒的），被唤醒后返回状态，在用户态poll和drv_poll中间的一层，去判断返回值和预期的事件是否一致，根据这个再将最终的结果返回给用户层（poll的返回值）。

异步通知

前面几种的方式要么是尝试读取数据要么是阻塞等待数据到来，效率都不高。异步通知可以由写入者向读出者通知：我有数据了你来取。

驱动程序在有数据时，发送SIGIO信号给应用层，应用层收到信号后，执行信号处理函数，去收取数据。

应用层：

告诉内核层自己的pid和是否使能异步通知,并注册信号处理函数，f_flags 中有一个 FASYNC 位，它被设置为 1 时表示使能异步通知功能。 当 f_flags 中的 FASYNC 位发生变化时，驱动程序的 fasync 函数被调用

```
static void sig_func(int sig)
{
    char buffer[32];
    read(fd, buffer, 32);
    printf("%s\r\n", buffer

);
signal(SIGIO, sig_func);
fcntl(fd, F_SETOWN, getpid());
int flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | FASYNC);
```

内核层：

当改变了flag的fasync位时，内核层的fasync被调用，在这个函数中，主要是根据传进的参数（fd里包括了pid和fasync位）对data_fasy进行初始化

```
static int drv_fasync(int fd,struct file *file,int on)
{
    if(fasync_helper(fd,file,on,&data_fasync)>=0)
    {
        return 0;
    }
    else
    {
        return -EIO;
    }
}
```

当有数据时，发送信号给应用层

```
kill_fasync(&data_fasync, SIGIO, POLLIN);
```