## 新字符设备驱动

在上一节的“内核模块模板”源码的基础上，添加新字符设备功能，能自动创建设备，类和节点。一共可分为以下4个步骤：申请设备号，注册cdev，创建设备类，创建设备节点

### 1.申请设备号

申请设备号的方式有2种：静态和动态方式

|  |  |  |
| --- | --- | --- |
| 静态 | **int register\_chrdev\_region(dev\_t from, unsigned count, const char \*name);** | |
| **注销函数** | **void unregister\_chrdev\_region(dev\_t from, unsigned count);** | |
| 参数 | **from** | 指定的设备号 |
| **count** | 连续次设备号个数 |
| **name** | 字符设备名称 |
| 返回值 | 成功 |  |
| 失败 | 小于0 |
| 示例 | #define CHR\_MAJOR 100 //主设备号  #define CHR\_MINOR 0 //次设备号  dev\_t dev = MKDEV(CHR\_MAJOR, CHR\_MINOR);  int ret = register\_chrdev\_region(dev , 1, "chrbase");//静态注册  unregister\_chrdev\_region(dev\_t from, unsigned count);//注销注册 | |

|  |  |  |
| --- | --- | --- |
| 动态 | **int alloc\_chrdev\_region(dev\_t \*dev,unsigned int firstminor,unsigned int count,char \*name);** | |
| **注销函数** | **void unregister\_chrdev\_region(dev\_t from, unsigned count);** | |
| 参数 | **\*dev** | 存放系统申请的设备号 |
| **firstminor** | 通常为0 |
| **count** | 连续次设备号个数 |
| **name** | 字符设备名称 |
| 返回值 | 成功 |  |
| 失败 | 小于0 |
| 示例 | int ret = alloc\_chrdev\_region(&devid, 0, 1, "chrbase"); //静态注册  unregister\_chrdev\_region(dev\_t from, unsigned count);//注销注册 | |

### 2. 注册cdev到内核

内核中每生成一个字符设备都对应一个cdev结构的变量。

内核提供了2种初始化cdev的方法：静态，动态。

|  |  |  |
| --- | --- | --- |
| 静态 | **void cdev\_init(struct cdev \*cdev, const struct file\_operations \*fops)** | |
| 参数 | cdev | 设备类 |
| fops | file\_operation结构 |
| 返回值 | 无 | |
| 示例 | struct cdev chr\_cdev;  cdev\_init(chr\_cdev, &fops);  chr\_cdev.owner = THIS\_MODULE; | |

|  |  |  |
| --- | --- | --- |
| 动态 | **struct cdev \*cdev\_alloc(void)** | |
| 参数 | cdev | 系统分配的设备类 |
| 返回值 | 系统分配的设备类 | |
| 示例 | struct cdev \*m\_cdev = cdev\_alloc();  m\_cdev->ops = &fops;  m\_cdev->owner = THIS\_MODULE; | |

不管是静态/动态，都是申请到设备类号，还要添加到内核中：

|  |  |  |
| --- | --- | --- |
| 动态 | **int cdev\_add(struct cdev \*p, dev\_t dev, unsigned count)** | |
| 参数 | **\*p** | 系统分配的设备类 |
| **dev** | 设备号 |
| **count** | 申请设备的数量 |
| 返回值 | 失败： < 0 | |
| 示例 | ret = cdev\_add(&chr\_cdev, devid, 1); | |

当一个字符设备驱动不再需要的时候（比如模块卸载），就可以用 cdev\_del() 函数来释放 cdev 占用的内存。

|  |  |  |
| --- | --- | --- |
| 注销 | **void cdev\_del(struct cdev \*p)** | |
| 参数 | **\*p** | 设备类结构体 |
| 返回值 | 无 | |
| 示例 | dev\_del(&chr\_cdev); //释放 cdev | |

### 3. 创建设备类

|  |  |  |
| --- | --- | --- |
| 创建 | **struct class \*class\_create(struct module \*owner, const char \*name)** | |
| 注销 | **void class\_destroy(struct class \*cls)** | |
| 参数 | **\*owner** | 一般为THIS\_MODULE |
| **\*name** | 类名 |
| 返回值 | 成功：类结构体指针 | |
|  | 失败： | |
| 示例 | struct class\* chr\_class = class\_creat(THIS\_MODULE, "chrbase");  if(IS\_ERR(chr\_class)) goto class\_error;  class\_destroy(chr\_class); | |

### 4. 创建设备节点

|  |  |  |
| --- | --- | --- |
| 创建 | **struct device \*device\_create(struct class \*cls, struct device \*parent,dev\_t devt, void \*drvdata, const char \*fmt, ...)** | |
| 注销 | **void device\_destroy(struct class \*dev, dev\_t devt);** | |
| 参数 | **\*cls** | 类结构体指针 |
| **\*parent** | 为NULL |
|  | **devt** | 设备号 |
|  | **\*drvdata** | 为NULL |
|  | **\*fmt** | 节点名 |
| 返回值 | 成功：节点结构体指针 | |
|  | 失败： | |
| 示例 | struct device\* chr\_device=device\_create(chr\_class,NULL,devid, NULL,CHRDASE\_NAME);  if(IS\_ERR(chr\_device)) goto device\_error;  device\_destroy(chr\_device, devid); | |

### 操作file\_operation结构体流程

static int xxx\_open(struct inode \*inode, struct file \*filp)

{return 0;}

static int xxx\_release(struct inode \*inode, struct file \*filp)

{return 0;}

static ssize\_t xxx\_read(struct file \*filp, char \_\_user \*buffer, size\_t size, loff\_t \*p)

{return 0;}

static ssize\_t xxx\_write(struct file \*filp, const char \_\_user \*buffer, size\_t count, loff\_t \*ppos){return 0;}

static struct file\_operations fops =

{

.owner = THIS\_MODULE,

.open = xxx\_open,

.read = xxx\_read,

.write = xxx\_write,

.release= xxx\_release,

};