

# 浅析混杂设备

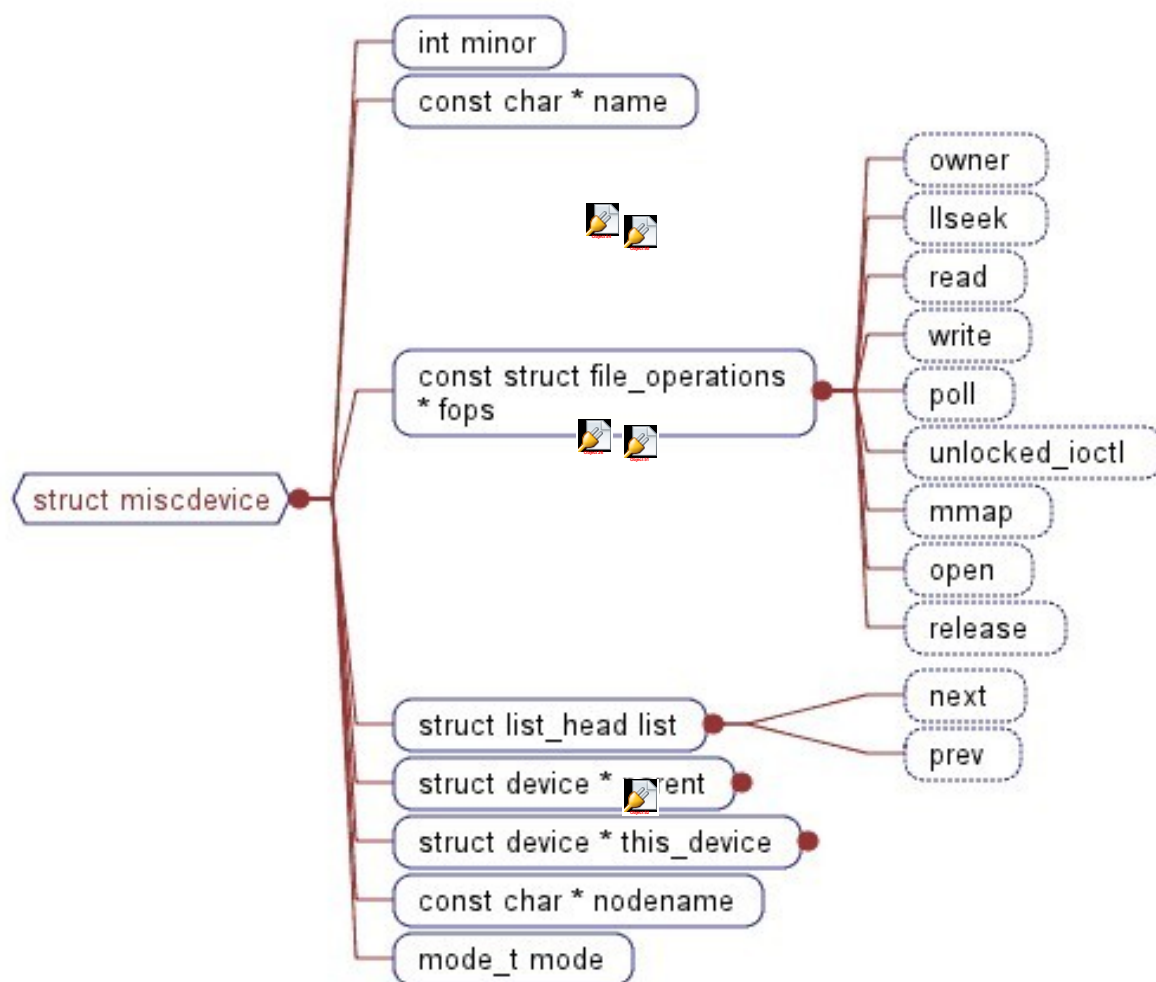
作者：李万鹏

混杂设备是在字符设备之上的一层抽象，它们拥有一些相同的特性。内核将这些共同性抽象至一个 API 中，这简化了这些驱动程序初始化的方式。字符设备驱动完成初始化的顺序如下：

- 通过 alloc\_chrdev\_region 及相关函数分配主/次设备号
- 使用 device\_create() 创建 /dev 和 /sys 节点
- 使用 cdev\_init() 和 cdev\_add() 将自身注册为字符驱动程序

混杂设备只要调用 misc\_register 即可完成上述步骤。

miscdevice 共享一个主设备号 MISC\_MAJOR(即 10)，但次设备号不同。所有的 miscdevice 设备形成了一个链表，对设备访问时内核根据次设备号查找对应的 miscdevice 设备，然后调用其 file\_operations 结构中注册的文件操作接口进行操作。在内核中用 struct miscdevice 表示 miscdevice 设备，然后调用其 file\_operations 结构中注册的文件操作接口进行操作。miscdevice 的 API 实现在 drivers/char/misc.c 中。



下边是描述这个设备的结构体：



```

1. struct miscdevice {
2.     int minor;           //次设备号
3.     const char *name;    //设备的名称
4.     const struct file_operations *fops; //文件操作
5.     struct list_head list; //misc_list 的链表头
6.     struct device *parent; //父设备(Linux 设备模型中的东东了, 哈哈)
7.     struct device *this_device; //当前设备, 是 device_create 的返回值, 下边会看到
8. };

```

然后来看看 misc 子系统的初始化函数:

```

1. static int __init misc_init(void)
2. {
3.     int err;
4.
5.     #ifdef CONFIG_PROC_FS
6.         /*创建一个 proc 入口项*/
7.         proc_create("misc", 0, NULL, &misc_proc_fops);
8.     #endif
9.     /*在/sys/class/目录下创建一个名为 misc 的类*/
10.    misc_class = class_create(THIS_MODULE, "misc");
11.    err = PTR_ERR(misc_class);
12.    if (IS_ERR(misc_class))
13.        goto fail_remove;
14.
15.    err = -EIO;
16.    /*注册设备, 其中设备的主设备号为 MISC_MAJOR, 为 10。设备名为 misc, misc_fops 是操作函数的集合*/
17.    if (register_chrdev(MISC_MAJOR, "misc", &misc_fops))
18.        goto fail_printk;
19.    return 0;
20.
21.fail_printk:
22.    printk("unable to get major %d for misc devices/n", MISC_MAJOR);
23.    class_destroy(misc_class);
24.fail_remove:
25.    remove_proc_entry("misc", NULL);
26.    return err;
27.}
28./*misc 作为一个子系统被注册到 linux 内核中*/
29.subsys_initcall(misc_init);

```

下边是 register\_chrdev 函数的实现:

```

1. int register_chrdev(unsigned int major, const char *name,
2.     const struct file_operations *fops)
3. {
4.     struct char_device_struct *cd;

```

```

5.  struct cdev *cdev;
6.  char *s;
7.  int err = -ENOMEM;
8.  /*主设备号是 10，次设备号为从 0 开始，分配 256 个设备*/
9.  cd = __register_chrdev_region(major, 0, 256, name);
10. if (IS_ERR(cd))
11.     return PTR_ERR(cd);
12. /*分配字符设备*/
13. cdev = cdev_alloc();
14. if (!cdev)
15.     goto out2;
16.
17. cdev->owner = fops->owner;
18. cdev->ops = fops;
19. /*Linux 设备模型中的,设置 kobject 的名字*/
20. kobject_set_name(&cdev->kobj, "%s", name);
21. for (s = strchr(kobject_name(&cdev->kobj), '/'); s; s = strchr(s, '/'))
22.     *s = '!';
23. /*把这个字符设备注册到系统中*/
24. err = cdev_add(cdev, MKDEV(cd->major, 0), 256);
25. if (err)
26.     goto out;
27.
28. cd->cdev = cdev;
29.
30. return major ? 0 : cd->major;
31.out:
32. kobject_put(&cdev->kobj);
33.out2:
34. kfree(__unregister_chrdev_region(cd->major, 0, 256));
35. return err;
36.}

```

来看看这个设备的操作函数的集合：

```

1. static const struct file_operations misc_fops = {
2.     .owner    = THIS_MODULE,
3.     .open     = misc_open,
4. };

```

可以看到这里只有一个打开函数，用户打开 miscdevice 设备是通过主设备号对应的打开函数，在这个函数中找到次设备号对应的相应的具体设备的 open 函数。它的实现如下：

```

1. static int misc_open(struct inode * inode, struct file * file)
2. {
3.     int minor = iminor(inode);
4.     struct miscdevice *c;
5.     int err = -ENODEV;

```

```

6.  const struct file_operations *old_fops, *new_fops = NULL;
7.
8.  lock_kernel();
9.  mutex_lock(&misc_mtx);
10. /*找到次设备号对应的操作函数集合，让 new_fops 指向这个具体设备的操作函数集合*/
11. list_for_each_entry(c, &misc_list, list) {
12.     if (c->minor == minor) {
13.         new_fops = fops_get(c->fops);
14.         break;
15.     }
16. }
17.
18. if (!new_fops) {
19.     mutex_unlock(&misc_mtx);
20.     /*如果没有找到，则请求加载这个次设备号对应的模块*/
21.     request_module("char-major-%d-%d", MISC_MAJOR, minor);
22.     mutex_lock(&misc_mtx);
23.     /*重新遍历 misc_list 链表，如果没有找到就退出，否则让 new_fops 指向这个具体设备的操作函数集合*/
24.     list_for_each_entry(c, &misc_list, list) {
25.         if (c->minor == minor) {
26.             new_fops = fops_get(c->fops);
27.             break;
28.         }
29.     }
30.     if (!new_fops)
31.         goto fail;
32. }
33.
34. err = 0;
35. /*保存旧打开函数的地址*/
36. old_fops = file->f_op;
37. /*让主设备号的操作函数集合指针指向具体设备的操作函数集合*/
38. file->f_op = new_fops;
39. if (file->f_op->open) {
40.     /*使用具体设备的打开函数打开设备*/
41.     err=file->f_op->open(inode,file);
42.     if (err) {
43.         fops_put(file->f_op);
44.         file->f_op = fops_get(old_fops);
45.     }
46. }
47. fops_put(old_fops);
48.fail:
49. mutex_unlock(&misc_mtx);
50. unlock_kernel();

```

```
51. return err;
52.}
```

再来看看 misc 子系统对外提供的两个重要的 API, misc\_register, misc\_deregister:

```
1. int misc_register(struct miscdevice * misc)
2. {
3.     struct miscdevice *c;
4.     dev_t dev;
5.     int err = 0;
6.     /*初始化 misc_list 链表*/
7.     INIT_LIST_HEAD(&misc->list);
8.     mutex_lock(&misc_mtx);
9.     /*遍历 misc_list 链表, 看这个次设备号以前有没有被用过, 如果次设备号已被占有则退出*/
10.    list_for_each_entry(c, &misc_list, list) {
11.        if (c->minor == misc->minor) {
12.            mutex_unlock(&misc_mtx);
13.            return -EBUSY;
14.        }
15.    }
16.    /*看是否是动态分配次设备号*/
17.    if (misc->minor == MISC_DYNAMIC_MINOR) {
18.        /*
19.         *#define DYNAMIC_MINORS 64 /* like dynamic majors */
20.         *static unsigned char misc_minors[DYNAMIC_MINORS / 8];
21.         *这里存在一个次设备号的位图, 一共 64 位。下边是遍历每一位,
22.         *如果这位为 0, 表示没有被占有, 可以使用, 为 1 表示被占用。
23.         */
24.        int i = DYNAMIC_MINORS;
25.        while (--i >= 0)
26.            if ( (misc_minors[i>>3] & (1 << (i&7))) == 0)
27.                break;
28.        if (i<0) {
29.            mutex_unlock(&misc_mtx);
30.            return -EBUSY;
31.        }
32.        /*得到这个次设备号*/
33.        misc->minor = i;
34.    }
35.    /*设置位图中相应位为 1*/
36.    if (misc->minor < DYNAMIC_MINORS)
37.        misc_minors[misc->minor >> 3] |= 1 << (misc->minor & 7);
38.    /*计算出设备号*/
39.    dev = MKDEV(MISC_MAJOR, misc->minor);
40.    /*在/dev 下创建设备节点, 这就是有些驱动程序没有显式调用 device_create, 却出现了设备节点的原因*/
```

```

41. misc->this_device = device_create(misc_class, misc->parent, dev, NULL,
42.     "%s", misc->name);
43. if (IS_ERR(misc->this_device)) {
44.     err = PTR_ERR(misc->this_device);
45.     goto out;
46. }
47.
48. /*
49.  * Add it to the front, so that later devices can "override"
50.  * earlier defaults
51.  */
52. /*将这个 miscdevice 添加到 misc_list 链表中*/
53. list_add(&misc->list, &misc_list);
54. out:
55. mutex_unlock(&misc_mtx);
56. return err;
57.}

```

这个是 miscdevice 的卸载函数：

```

1. int misc_deregister(struct miscdevice *misc)
2. {
3.     int i = misc->minor;
4.
5.     if (list_empty(&misc->list))
6.         return -EINVAL;
7.
8.     mutex_lock(&misc_mtx);
9.     /*在 misc_list 链表中删除 miscdevice 设备*/
10.    list_del(&misc->list);
11.    /*删除设备节点*/
12.    device_destroy(misc_class, MKDEV(MISC_MAJOR, misc->minor));
13.    if (i < DYNAMIC_MINORS && i > 0) {
14.        /*释放位图相应位*/
15.        misc_minors[i >> 3] &= ~(1 << (misc->minor & 7));
16.    }
17.    mutex_unlock(&misc_mtx);
18.    return 0;
19.}

```

总结一下 miscdevice 驱动的注册和卸载流程：

misc\_register: 匹配次设备号->找到一个没有占用的次设备号(如果需要动态分配的话)->计算设备号->创建设备文件->miscdevice 结构体添加到 misc\_list 链表中。

misc\_deregister: 从 misc\_list 中删除 miscdevice->删除设备文件->位图位清零。