# Linux　设备驱动之字符设备(一)

——编写者:草根老师(程姚根)

==========

由于水平有限,文档中难免有错误的地方，希望大家踊跃拍砖。大家可以通过邮箱:chengyaogen@163.com告诉我，以便改正,谢谢!

==========

## 一、Linux 设备分类

字符设备：以字节为单位读写的设备
块设备　：以块为单位(效率最高)读写的设备
网络设备：用于网络通讯的设备

详细解释来自http://blog.chinaunix.net/uid-23028407-id-264280.html

字符设备：

字符（char）设备是个能够像字节流（类似文件）一样被访问的设备，由字符设备驱动程序来实现这种特性。字符设备驱动程序通常至少要实现open、close、read和write的系统调用。字符终端（/dev/console）和串口（/dev/ttyS0以及类似设备）就是两个字符设备，它们能很好的说明“流”这种抽象概念。字符设备可以通过FS节点来访问，比如/dev/tty1和/dev/lp0等。这些设备文件和普通文件之间的唯一差别在于对普通文件的访问可以前后移动访问位置，而大多数字符设备是一个只能顺序访问的数据通道。然而，也存在具有数据区特性的字符设备，访问它们时可前后移动访问位置。例如framebuffer就是这样的一个设备，app可以用mmap或lseek访问抓取的整个图像。

块设备：

和字符设备类似，块设备也是通过/dev目录下的文件系统节点来访问。块设备（例如磁盘）上能够容纳filesystem。在大多数的Unix系统中，进行I/O操作时块设备每次只能传输一个或多个完整的块，而每块包含512字节（或2的更高次幂字节的数据）。Linux可以让app像字符设备一样地读写块设备，允许一次传递任意多字节的数据。因此，块设备和字符设备的区别仅仅在于内核内部管理数据的方式，也就是内核及驱动程序之间的软件接口，而这些不同对用户来讲是透明的。在内核中，和字符驱动程序相比，块驱动程序具有完全不同的接口。

网络设备：

任何网络事物都需要经过一个网络接口形成，网络接口是一个能够和其他主机交换数据的设备。接口通常是一个硬件设备，但也可能是个纯软件设备，比如回环（loopback）接口。
网络接口由内核中的网络子系统驱动，负责发送和接收数据包。许多网络连接（尤其是使用TCP协议的连接）是面向流的，但网络设备却围绕数据包的传送和接收而设计。网络驱动程序不需要知道各个连接的相关信息，它只要处理数据包即可。
由于不是面向流的设备，因此将网络接口映射到filesystem中的节点（比如/dev/tty1）比较困难。Unix访问网络接口的方法仍然是给它们分配一个唯一的名字（比如eth0），但这个名字在filesystem中不存在对应的节点。内核和网络设备驱动程序间的通信，完全不同于内核和字符以及块驱动程序之间的通信，内核调用一套和数据包相关的函数而不是read、write等。

## 二、上层应用程序是如何访问到底层的驱动程序?

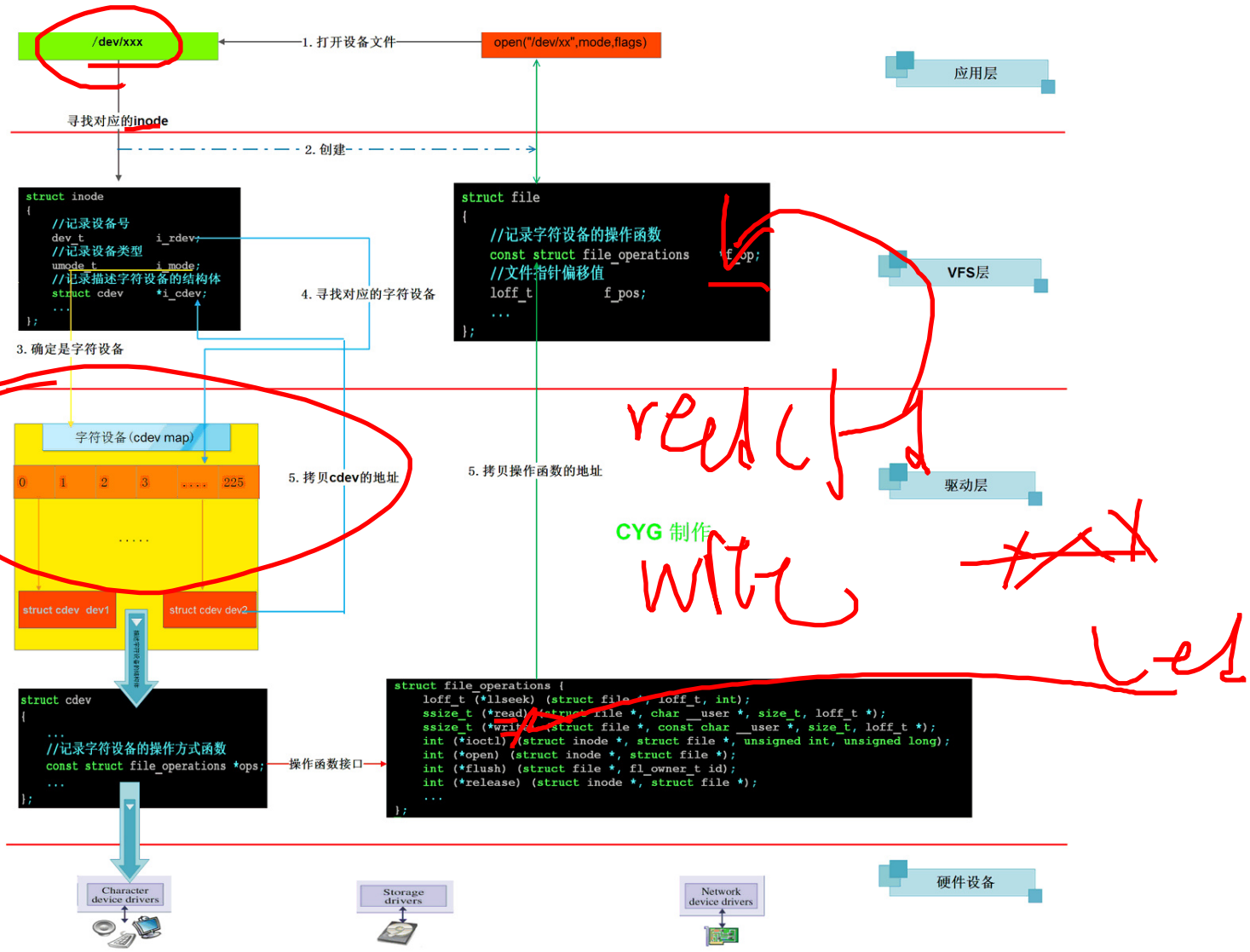
在Linux 的世界里面一切皆文件，所有的硬件设备操作到应用层都会被抽象成文件的操作。我们知道如果应用层要访问硬件设备，它必定要调用到硬件对应的驱动程序。Linux 内核中有那么多驱动程序，应用层怎么才能精确的调用到底层的驱动程序呢?

在这里我们字符设备为例，来看一下应用程序是如何和底层驱动程序关联起来的。

必须知道的知识:

(1)在 Linux 文件系统中，每个文件都用一个struct inode结构体来描述，这个结构体里面记录了这个文件的所有信息，例如:文件类型，访问权限等。
(2)在Linux操作系统中，每个驱动程序在应用层的/dev目录下都会一个设备文件和它对应。
(3)在Linux操作系统中，每个驱动程序都有一个设备号。
(4)在Linux操作系统中，每打开一次文件，Linux 操作系统在VFS层都会分配一个struct file结构体来描述打开的这个文件。

注意:常常我们认为struct inode描述的是文件的静态信息，即这些信息很少会改变。而struct file描述的是动态信息，即在对文件的操作的时候，struct file里面的信息经常会发生变化。典型的是struct file结构体里面的f_pos(记录当前文件的位移量),每次读写一个普通文件时f_ops的值都会发生改变。




通过上图我们可以知道，如果想访问底层设备，就必须打开对应的设备文件。也就是在这个打开的过程中，Linux 内核将应用层和对应的驱动程序关联起来。

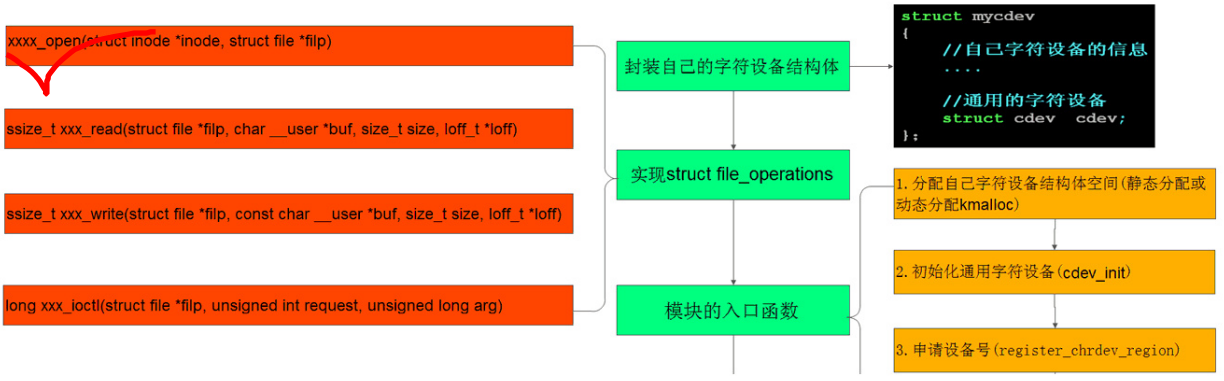
(1)当open函数打开设备文件时，可以根据设备文件对应的struct inode结构体描述的信息，可以知道接下来要操作的设备类型(字符设备还是块设备) 。还会分配一个struct file结构体哦。

(2)根据struct inode结构体里面记录的设备号，可以找到对应的驱动程序。这里以字符设备为例。在 Linux 操作系统中每个字符设备有一个struct cdev结构体。此结构体描述了字符设备所有的信息，其中最重要一项的就是字符设备的操作函数接口。

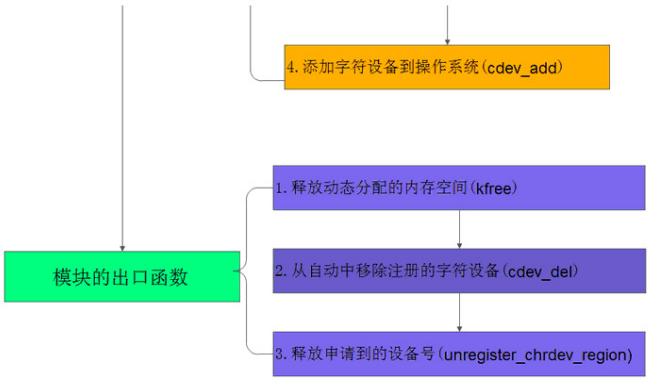
(3)找到struct cdev结构体后，Linux 内核就会将struct cdev结构体所在的内存空间首地记录在struct inode结构体的i_cdev成员中。将struct cdev结构体的中记录的函数操作接口地址记录在struct file 结构体的f_op成员中。

(4)任务完成，VFS层会给应用层返回一个文件描述符(fd)。这个fd是和struct file结构体对应的。接下来上层的应用程序就可以通过fd来找到strut file ,然后在由struct file找到操作字符设备的函数接口了。

## 三、如何编写字符设备驱动



CYG 制作




## 四、字符驱动相关函数分析

```
/**
 * cdev_init() - initialize a cdev structure
 * @cdev: the structure to initialize
 * @fops: the file_operations for this device
 *
 * Initializes @cdev, remembering @fops, making it ready to add to the
 * system with cdev_add().
 */
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
```

功能:初始化cdev 结构体
参数:
@cdev  cdev结构体地址
@fops  操作字符设备的函数接口地址
返回值:
无

```
/**
 * register_chrdev_region() - register a range of device numbers
 * @from: the first in the desired range of device numbers; must include
 *        the major number.
 * @count: the number of consecutive device numbers required
 * @name: the name of the device or driver.
 *
 * Return value is zero on success, a negative error code on failure.
 */
int register_chrdev_region(dev_t from, unsigned count, const char *name)
```

功能: 注册一个范围()的设备号
参数:
@from   设备号
@count  注册的设备个数
@name   设备的名字
返回值:
成功返回0,失败返回错误码(负数)

```
/**
 * cdev_add() - add a char device to the system
 * @p: the cdev structure for the device
 * @dev: the first device number for which this device is responsible
 * @count: the number of consecutive minor numbers corresponding to this
 *         device
 *
 * cdev_add() adds the device represented by @p to the system, making it
 * live immediately.  A negative error code is returned on failure.
 */
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
```

功能:添加一个字符设备到操作系统
参数:
@p      cdev结构体地址
@dev    设备号
@count  次设备号个数
返回值:
成功返回0,失败返回错误码(负数)

```
/**
 * cdev_del() - remove a cdev from the system
 * @p: the cdev structure to be removed
```

```
 *
 * cdev_del() removes @p from the system, possibly freeing the structure
 * itself.
 */
void cdev_del(struct cdev *p)
```

功能:从系统中删除一个字符设备
参数:
@p  cdev结构体地址
返回值:
无

```
/**
 * unregister_chrdev_region() - return a range of device numbers
 * @from: the first in the range of numbers to unregister
 * @count: the number of device numbers to unregister
 *
 * This function will unregister a range of @count device numbers,
 * starting with @from.  The caller should normally be the one who
 * allocated those numbers in the first place...
 */
void unregister_chrdev_region(dev_t from, unsigned count)
```

功能:释放申请的设备号
参数:
@from  设备号
@count 次设备号个数
返回值:
无

## 五、开始写字符设备驱动

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/cdev.h>
#include <linux/fs.h>

#define MAJOR_NUM 168

struct mycdev
{
    unsigned char buffer[50];
    struct cdev cdev;
}dev_fifo;

MODULE_LICENSE("GPL");


static int dev_fifo_open(struct inode *inode, struct file *file)
{
    printk("dev_fifo_open success!\n");

    return 0;
}

static ssize_t dev_fifo_read(struct file *file, char __user *buf, size_t size, loff_t *ppos)
{
    printk("dev_fifo_read success!\n");
    return 0;
}

static ssize_t dev_fifo_write(struct file *file, const char __user *buf, size_t size, loff_t *ppos)
{
    printk("dev_fifo_write success!\n");
    return size;
}

static const struct file_operations fifo_operations = {
 .owner = THIS_MODULE,
 .open = dev_fifo_open,
 .read = dev_fifo_read,
 .write = dev_fifo_write,
};

int __init dev_fifo_init(void)
{
    int ret;
    dev_t dev_num;
```

```
    //初始化字符设备
    cdev_init(&dev_fifo.cdev,&fifo_operations);

    //设备号：主设备号(12bit) | 次设备号(20bit)
    dev_num = MKDEV(MAJOR_NUM, 0);

    //注册设备号
    ret = register_chrdev_region(dev_num,1,"dev_fifo");
    if(ret < 0){
        printk("Fail to register_chrdev_region");
        return -EIO;
    }

    //添加设备到操作系统
    ret = cdev_add(&dev_fifo.cdev,dev_num,1);
    if (ret < 0)
    {
        printk("Fail to cdev_add");
        goto unregister_chrdev;
    }

    printk("Register dev_fifo to system,ok!\n");

    return 0;

unregister_chrdev:
    unregister_chrdev_region(MKDEV(250, 0), 1);
    return -1;
}

void __exit dev_fifo_exit(void)
{
    //从系统中删除添加的字符设备
    cdev_del(&dev_fifo.cdev);
    //释放申请的设备号
    unregister_chrdev_region(MKDEV(MAJOR_NUM, 0), 1);
    printk("Exit dev_fifo ok!\n");
    return;
}

module_init(dev_fifo_init);
module_exit(dev_fifo_exit);
```

Makefile:

```
ifeq (S(KERNELRELEASE),)
KERNEL_DIR ?=/lib/modules/S(shell uname -r)/build
PWD :=S(shell pwd)

modules:
 S(MAKE) -C S(KERNEL_DIR) M=S(PWD) modules

.PHONY:modules clean
clean:
 S(MAKE) -C S(KERNEL_DIR) M=S(PWD) clean
else
 obj-m := dev_fifo.o
endif
```

测试步骤:

(1)加载模块

```
sudo insmod dev_fifo.ko
```

(2)创建设备节点并且指定权限

```
sudo mknod  /dev/dev_fifo   c  168  0
sudo chmod  666 /dev/dev_fifo
```

(3)测试字符设备

```
cat     /dev/dev_fifo                [读设备]

echo  "test" > /dev/dev_fifo      [写设备]
```