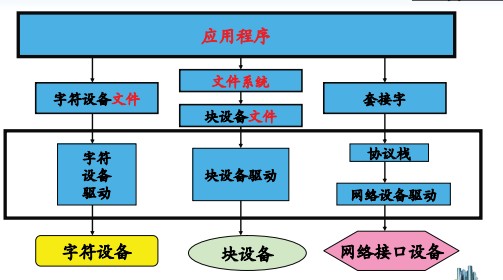
**一、linux系统将设备**分为3类：字符设备、块设备、网络设备。使用驱动程序：



1、**字符设备**：是指只能一个字节一个字节读写的设备，不能随机读取设备内存中的某一数据，读取数据需要按照先后数据。字符设备是面向流的设备，常见的字符设备有鼠标、键盘、串口、控制台和LED设备等。

2、**块设备**：是指可以从设备的任意位置读取一定长度数据的设备。块设备包括硬盘、磁盘、U盘和SD卡等。

　　每一个字符设备或块设备都在/dev目录下对应一个设备文件。linux用户程序通过设备文件（或称设备节点）来使用驱动程序操作字符设备和块设备。

**二、字符设备驱动程序基础:**  
1、**主设备号和次设备号**（二者一起为设备号）：

一个字符设备或块设备都有一个主设备号和一个次设备号。主设备号用来标识与设备文件相连的驱动程序，用来反映设备类型。次设备号被驱动程序用来辨别操作的是哪个设备，用来区分同类型的设备。

linux内核中，设备号用dev\_t来描述，2.6.28中定义如下：

typedef u\_long dev\_t;

在32位机中是4个字节，高12位表示主设备号，低12位表示次设备号。

可以使用下列宏从dev\_t中获得主次设备号,也可以使用下列宏通过主次设备号生成dev\_t: MAJOR(dev\_t dev);

MKDEV(int major,int minor);

MINOR(dev\_t dev);

//宏定义：

#define MINORBITS 20

#define MINORMASK ((1U << MINORBITS) - 1)

#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))

#define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))

#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))

2、**分配设备号**（两种方法）：

（1）静态申请：

int register\_chrdev\_region(dev\_t from, unsigned count, const char \*name)；

（2）动态分配：

int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count, const char \*name)；

**注销设备号**：

void unregister\_chrdev\_region(dev\_t from, unsigned count)；

**创建设备文件**：

利用cat /proc/devices查看申请到的设备名，设备号。

（1）使用mknod手工创建：mknod filename type major minor

（2）自动创建；

　　利用udev（mdev）来实现设备文件的自动创建，首先应保证支持udev（mdev），由busybox配置。在驱动初始化代码里调用class\_create为该设备创建一个class，再为每个设备调用device\_create创建对应的设备。

3、**字符设备驱动程序重要的数据结构**：

（1）struct file：代表一个打开的文件描述符，系统中每一个打开的文件在内核中都有一个关联的struct file。它由内核在open时创建，并传递给在文件上操作的任何函数，直到最后关闭。当文件的所有实例都关闭之后，内核释放这个数据结构。

//重要成员：

const struct file\_operations \*f\_op; //该操作是定义文件关联的操作的。内核在执行open时对这个指针赋值。

off\_t f\_pos; //该文件读写位置。

void \*private\_data;//该成员是系统调用时保存状态信息非常有用的资源

（2）struct inode:用来记录文件的物理信息。它和代表打开的file结构是不同的。一个文件可以对应多个file结构，但只有一个inode结构。inode一般作为file\_operations结构中函数的参数传递过来。

　　inode译成中文就是索引节点。每个存储设备或存储设备的分区（存储设备是硬盘、软盘、U盘 ... ... ）被格式化为文件系统后，应该有两部份，一部份是inode，另一部份是Block，Block是用来存储数据用的。而inode呢，就是用来存储这些数据的信息，这些信息包括文件大小、属主、归属的用户组、读写权限等。inode为每个文件进行信息索引，所以就有了inode的数值。操作系统根据指令，能通过inode值最快的找到相对应的文件。

dev\_t i\_rdev; //对表示设备文件的inode结构，该字段包含了真正的设备编号。

struct cdev \*i\_cdev; //是表示字符设备的内核的内部结构。当inode指向一个字符设备文件时，该字段包含了指向struct cdev结构的指针。

//我们也可以使用下边两个宏从inode中获得主设备号和此设备号：

unsigned int iminor(struct inode \*inode);

unsigned int imajor(struct inode \*inode);

（3）struct file\_operations

struct file\_operations \*\*\*\_ops={

.owner = THIS\_MODULE,  
 .llseek = \*\*\*\_llseek,  
 .read = \*\*\*\_read,  
 .write = \*\*\*\_write,  
 .ioctl = \*\*\*\_ioctl,  
 .open = \*\*\*\_open,  
 .release = \*\*\*\_release,   
 。。。 。。。  
};  
  
struct module \*owner;  
 /\*第一个 file\_operations 成员根本不是一个操作; 它是一个指向拥有这个结构的模块的指针.  
 这个成员用来在它的操作还在被使用时阻止模块被卸载. 几乎所有时间中, 它被简单初始化为   
THIS\_MODULE, 一个在 <linux/module.h> 中定义的宏.这个宏比较复杂，在进行简单学习操作的时候，一般初始化为THIS\_MODULE。\*/  
  
  
loff\_t (\*llseek) (struct file \* filp , loff\_t p, int orig);  
/\*(指针参数filp为进行读取信息的目标文件结构体指针；参数 p 为文件定位的目标偏移量；参数orig为对文件定位  
的起始地址，这个值可以为文件开头（SEEK\_SET，0,当前位置(SEEK\_CUR,1)，文件末尾(SEEK\_END,2)）  
llseek 方法用作改变文件中的当前读/写位置, 并且新位置作为(正的)返回值.  
loff\_t 参数是一个"long offset", 并且就算在 32位平台上也至少 64 位宽. 错误由一个负返回值指示.  
如果这个函数指针是 NULL, seek 调用会以潜在地无法预知的方式修改 file 结构中的位置计数器( 在"file 结构" 一节中描述).\*/  
  
ssize\_t (\*read) (struct file \* filp, char \_\_user \* buffer, size\_t size , loff\_t \* p);  
/\*(指针参数 filp 为进行读取信息的目标文件，指针参数buffer 为对应放置信息的缓冲区（即用户空间内存地址），  
参数size为要读取的信息长度，参数 p 为读的位置相对于文件开头的偏移，在读取信息后，这个指针一般都会移动，移动的值为要读取信息的长度值）  
这个函数用来从设备中获取数据. 在这个位置的一个空指针导致 read 系统调用以 -EINVAL("Invalid argument") 失败.  
 一个非负返回值代表了成功读取的字节数( 返回值是一个 "signed size" 类型, 常常是目标平台本地的整数类型).\*/  
  
ssize\_t (\*aio\_read)(struct kiocb \* , char \_\_user \* buffer, size\_t size , loff\_t p);  
/\*可以看出，这个函数的第一、三个参数和本结构体中的read()函数的第一、三个参数是不同 的，  
异步读写的第三个参数直接传递值，而同步读写的第三个参数传递的是指针，因为AIO从来不需要改变文件的位置。  
异步读写的第一个参数为指向kiocb结构体的指针，而同步读写的第一参数为指向file结构体的指针，每一个I/O请求都对应一个kiocb结构体);  
初始化一个异步读 -- 可能在函数返回前不结束的读操作.如果这个方法是 NULL, 所有的操作会由 read 代替进行(同步地).  
(有关linux异步I/O，可以参考有关的资料，《linux设备驱动开发详解》中给出了详细的解答)\*/  
  
ssize\_t (\*write) (struct file \* filp, const char \_\_user \* buffer, size\_t count, loff\_t \* ppos);  
/\*(参数filp为目标文件结构体指针，buffer为要写入文件的信息缓冲区，count为要写入信息的长度，  
ppos为当前的偏移位置，这个值通常是用来判断写文件是否越界）  
发送数据给设备. 如果 NULL, -EINVAL 返回给调用 write 系统调用的程序. 如果非负, 返回值代表成功写的字节数.  
(注：这个操作和上面的对文件进行读的操作均为阻塞操作）\*/  
  
ssize\_t (\*aio\_write)(struct kiocb \*, const char \_\_user \* buffer, size\_t count, loff\_t \* ppos);  
/\*初始化设备上的一个异步写.参数类型同aio\_read()函数;\*/  
  
int (\*readdir) (struct file \* filp, void \*, filldir\_t);  
/\*对于设备文件这个成员应当为 NULL; 它用来读取目录, 并且仅对文件系统有用.\*/  
  
unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);  
/\*(这是一个设备驱动中的轮询函数，第一个参数为file结构指针，第二个为轮询表指针）  
这个函数返回设备资源的可获取状态，即POLLIN，POLLOUT，POLLPRI，POLLERR，POLLNVAL等宏的位“或”结果。  
每个宏都表明设备的一种状态，如：POLLIN（定义为0x0001）意味着设备可以无阻塞的读，POLLOUT（定义为0x0004）意味着设备可以无阻塞的写。  
(poll 方法是 3 个系统调用的后端: poll, epoll, 和 select, 都用作查询对一个或多个文件描述符的读或写是否会阻塞.  
 poll 方法应当返回一个位掩码指示是否非阻塞的读或写是可能的, 并且, 可能地, 提供给内核信息用来使调用进程睡眠直到 I/O 变为可能.   
如果一个驱动的 poll 方法为 NULL, 设备假定为不阻塞地可读可写.  
(这里通常将设备看作一个文件进行相关的操作，而轮询操作的取值直接关系到设备的响应情况，可以是阻塞操作结果，同时也可以是非阻塞操作结果）\*/  
  
int (\*ioctl) (struct inode \*inode, struct file \*filp, unsigned int cmd, unsigned long arg);  
/\*(inode 和 filp 指针是对应应用程序传递的文件描述符 fd 的值, 和传递给 open 方法的相同参数.  
cmd 参数从用户那里不改变地传下来, 并且可选的参数 arg 参数以一个 unsigned long 的形式传递, 不管它是否由用户给定为一个整数或一个指针.  
如果调用程序不传递第 3 个参数, 被驱动操作收到的 arg 值是无定义的.  
因为类型检查在这个额外参数上被关闭, 编译器不能警告你如果一个无效的参数被传递给 ioctl, 并且任何关联的错误将难以查找.）  
ioctl 系统调用提供了发出设备特定命令的方法(例如格式化软盘的一个磁道, 这不是读也不是写). 另外, 几个 ioctl 命令被内核识别而不必引用 fops 表.  
 如果设备不提供 ioctl 方法, 对于任何未事先定义的请求(-ENOTTY, "设备无这样的 ioctl"), 系统调用返回一个错误.\*/  
  
int (\*mmap) (struct file \*, struct vm\_area\_struct \*);  
/\*mmap 用来请求将设备内存映射到进程的地址空间. 如果这个方法是 NULL, mmap 系统调用返回 -ENODEV.  
(如果想对这个函数有个彻底的了解，那么请看有关“进程地址空间”介绍的书籍)\*/  
  
int (\*open) (struct inode \* inode , struct file \* filp ) ;  
/\*(inode 为文件节点,这个节点只有一个，无论用户打开多少个文件，都只是对应着一个inode结构；  
但是filp就不同，只要打开一个文件，就对应着一个file结构体，file结构体通常用来追踪文件在运行时的状态信息）  
 尽管这常常是对设备文件进行的第一个操作, 不要求驱动声明一个对应的方法. 如果这个项是 NULL, 设备打开一直成功, 但是你的驱动不会得到通知.  
与open()函数对应的是release()函数。\*/  
  
int (\*flush) (struct file \*);  
/\*flush 操作在进程关闭它的设备文件描述符的拷贝时调用; 它应当执行(并且等待)设备的任何未完成的操作.  
这个必须不要和用户查询请求的 fsync 操作混淆了. 当前, flush 在很少驱动中使用;  
 SCSI 磁带驱动使用它, 例如, 为确保所有写的数据在设备关闭前写到磁带上. 如果 flush 为 NULL, 内核简单地忽略用户应用程序的请求.\*/  
  
int (\*release) (struct inode \*, struct file \*);  
/\*release ()函数当最后一个打开设备的用户进程执行close()系统调用的时候，内核将调用驱动程序release()函数：  
void release(struct inode inode,struct file \*file),release函数的主要任务是清理未结束的输入输出操作，释放资源，用户自定义排他标志的复位等。  
 在文件结构被释放时引用这个操作. 如同 open, release 可以为 NULL.\*/  
  
int(\*synch)(struct file \*,struct dentry \*,int datasync);  
//刷新待处理的数据,允许进程把所有的脏缓冲区刷新到磁盘。  
  
  
int (\*aio\_fsync)(struct kiocb \*, int);  
 /\*这是 fsync 方法的异步版本.所谓的fsync方法是一个系统调用函数。系统调用fsync  
把文件所指定的文件的所有脏缓冲区写到磁盘中（如果需要，还包括存有索引节点的缓冲区）。  
相应的服务例程获得文件对象的地址，并随后调用fsync方法。通常这个方法以调用函数\_\_writeback\_single\_inode()结束，  
这个函数把与被选中的索引节点相关的脏页和索引节点本身都写回磁盘。\*/  
  
int (\*fasync) (int, struct file \*, int);  
//这个函数是系统支持异步通知的设备驱动，下面是这个函数的模板：  
  
static int \*\*\*\_fasync(int fd,struct file \*filp,int mode)  
{  
 struct \*\*\*\_dev \* dev=filp->private\_data;  
 return fasync\_helper(fd,filp,mode,&dev->async\_queue);//第四个参数为 fasync\_struct结构体指针的指针。  
//这个函数是用来处理FASYNC标志的函数。（FASYNC：表示兼容BSD的fcntl同步操作）当这个标志改变时，驱动程序中的fasync（）函数将得到执行。  
}  
/\*此操作用来通知设备它的 FASYNC 标志的改变. 异步通知是一个高级的主题, 在第 6 章中描述.  
这个成员可以是NULL 如果驱动不支持异步通知.\*/  
  
int (\*lock) (struct file \*, int, struct file\_lock \*);  
//lock 方法用来实现文件加锁; 加锁对常规文件是必不可少的特性, 但是设备驱动几乎从不实现它.  
  
ssize\_t (\*readv) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);  
ssize\_t (\*writev) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);  
/\*这些方法实现发散/汇聚读和写操作. 应用程序偶尔需要做一个包含多个内存区的单个读或写操作;  
 这些系统调用允许它们这样做而不必对数据进行额外拷贝. 如果这些函数指针为 NULL, read 和 write 方法被调用( 可能多于一次 ).\*/  
  
ssize\_t (\*sendfile)(struct file \*, loff\_t \*, size\_t, read\_actor\_t, void \*);  
/\*这个方法实现 sendfile 系统调用的读, 使用最少的拷贝从一个文件描述符搬移数据到另一个.  
例如, 它被一个需要发送文件内容到一个网络连接的 web 服务器使用. 设备驱动常常使 sendfile 为 NULL.\*/  
  
ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);  
/\*sendpage 是 sendfile 的另一半; 它由内核调用来发送数据, 一次一页, 到对应的文件. 设备驱动实际上不实现 sendpage.\*/

unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);

/\*这个方法的目的是在进程的地址空间找一个合适的位置来映射在底层设备上的内存段中.

这个任务通常由内存管理代码进行; 这个方法存在为了使驱动能强制特殊设备可能有的任何的对齐请求. 大部分驱动可以置这个方法为 NULL.[10]\*/

int (\*check\_flags)(int)  
//这个方法允许模块检查传递给 fnctl(F\_SETFL...) 调用的标志.

int (\*dir\_notify)(struct file \*, unsigned long);

//这个方法在应用程序使用 fcntl 来请求目录改变通知时调用. 只对文件系统有用; 驱动不需要实现 dir\_notify.

三、**字符设备驱动程序设计**：

1.**设备注册**：  
在linux2.6内核中，字符设备使用struct cdev来描述；

struct cdev  
{  
 struct kobject kobj;//内嵌的kobject对象  
 struct module \*owner;//所属模块  
 struct file\_operations \*ops;//文件操作结构体  
 struct list\_head list;  
 dev\_t dev;//设备号,长度为32位，其中高12为主设备号，低20位为此设备号  
 unsigned int count;  
};

字符设备的注册分为三个步骤：

（1）分配cdev: struct cdev \*cdev\_alloc(void);  
（2）初始化cdev： void cdev\_init(struct cdev \*cdev, const struct file\_operations \*fops)；  
（3）添加cdev： int cdev\_add(struct cdev \*p, dev\_t dev, unsigned count)

2.**设备操作的实现**：file\_operations函数集的实现（要明确某个函数什么时候被调用？调用来做什么操作？）  
**特别注意：驱动程序应用程序的数据交换:**  
　　驱动程序和应用程序的数据交换是非常重要的。file\_operations中的read()和write()函数，就是用来在驱动程序和应用程序间交换数据的。通过数据交换，驱动程序和应用程序可以彼此了解对方的情况。但是驱动程序和应用程序属于不同的地址空间。驱动程序不能直接访问应用程序的地址空间；同样应用程序也不能直接访问驱动程序的地址空间，否则会破坏彼此空间中的数据，从而造成系统崩溃，或者数据损坏。安全的方法是使用内核提供的专用函数，完成数据在应用程序空间和驱动程序空间的交换。这些函数对用户程序传过来的指针进行了严格的检查和必要的转换，从而保证用户程序与驱动程序交换数据的安全性。这些函数有：

unsigned long copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long n);   
unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long n);   
put\_user(local,user);   
get\_user(local,user);

3.**设备注销**：void cdev\_del(struct cdev \*p)；

字符设备是3大类设备（字符设备、块设备、网络设备）中较简单的一类设备，其驱动程序中完成的主要工作是初始化、添加和删除cdev结构体，申请和释放设备号，以及填充file\_operation结构体中操作函数，并实现file\_operations结构体中的read()、write()、ioctl()等重要函数。如图所示为cdev结构体、file\_operations和用户空间调用驱动的关系。

