
网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

块设备驱动的引入.....	3
一， 总结“字符设备驱动程序”：.....	3
1，引入“字符设备驱动程序”：.....	3
2，这种最简单的驱动程序的写法：.....	3
3，如何知道设备有数据过来：如按键。.....	3
二，块设备驱动程序：.....	4
1，硬盘：.....	4
2，flash：.....	5
块设备驱动框架.....	7
一，层次框架：.....	7
硬件：硬盘、FLASH.....	7
二，流程：.....	7
电梯调度算法：.....	8
三，如何写块设备驱动程序：.....	10
块设备驱动编写.....	11
分配一段内存，用内存来模拟硬盘：.....	11
大致分析：.....	11
register_blkdev(XT_DISK_MAJOR, "xd").....	11
2，xd_queue = blk_init_queue(do_xd_request, &xd_lock);.....	11
3，如何使用队列 xd_queue：.....	11
4，add_disk(xd_gendisk[i]);.....	11
下面直接自己写代码：.....	12
一，入口函数：.....	12
1，分配一个 gendisk 结构体：alloack_disk().....	12
2，分配/设置队列：blk_init_queue().....	12
3，设置其他属性：.....	13
4，注册：add_disk()......	14
5，处理请求：.....	14
二，出口函数：.....	14
三，测试：.....	15
1，编译.....	15
四，最终完善的内存模拟磁盘：.....	16
1，“硬件相关操作”：分配内存。.....	16
2，在“队列请求”处理函数中操作：实现内存模拟磁盘。.....	17

3, 再编译后测试:	17
五, 开始试着分区:	21
框架:	25
硬件: 硬盘、FLASH	25



块设备驱动的引入

一， 总结“字符设备驱动程序”：

1，引入“字符设备驱动程序”：

APP: open, read, write. ----> 对应提供驱动程序的读写等函数。

驱动: drv_open, drv_read, drv_write

硬件

当应用程序的 open, read 等函数要操作“硬件”时，自然引入了“驱动程序”的概念，最简单的方式是 APP 调用 open 时，驱动程序的“drv_open”函数被调用等等。

2，这种最简单的驱动程序的写法：

代码步骤：

1, 确定主设备号：可以自己确定，也可让内核分配。

2, 要构造驱动中的“open, read, write 等”是将它们放在一个“file_operations”结构体中。

File_operations==> .open, .read, .write, .poll 等。

这里“open”函数会去配置硬件的相关引脚等，还有注册中断。

3, register_chrdev 注册字符设备构造的“file_operations”结构:使用这个 file_operations 结构体。是把这个结构放到内核的某个以此设备的“主设备号”为下标的数组中去。

Register_chrdev(主设备号, 设备号, file_operations 结构)。

4, 入口函数：调用这个“register_chrdev()”函数。内核装载某个模块时，会自动调用这个“入口函数”。

5, 出口函数：

3，如何知道设备有数据过来：如按键。

①，查询方式：

驱动程序中提供一个 读函数，直接返回某引脚状态，应用程序只能连续不断的读，比较前后两次的引脚状态是否有变化。这种查询方式“太累”。CPU 会占用很高。这样引入“休眠唤醒”机制。

②，休眠唤醒：

APP 用 读函数 进入驱动程序的读函数。在驱动程序里，若没有数据就休眠，被唤醒时，就 copy_to_user 把数据拷贝到用户空间后返回。

其中“唤醒”由“中断服务”程序来唤醒。在驱动程序中用“request_irq”注册“中断服务”程序。

但可能没有“唤醒”的时候。就想着加个“闹钟--poll”。

③，poll 机制：定时的时长内，若“中断服务”程序有来唤醒就好，若没有来，则定时超时由“闹钟”来唤醒。

但“休眠唤醒”和“poll”都得“休眠”。

④，“异步通知”：发信号

以上4种方法是写“字符驱动”的基本形式。但这4种情况有个缺点是这种驱动只有自己方便使用。写成通用的驱动程序，则是看懂内核代码，将自己的驱动代码融合进去。引出“输入子系统”。对“按键”是“输入子系统”，对LCD是“framebuffer”。

⑤，“输入子系统”：融入别人写的代码（也是上面4种方式写成）。

二，块设备驱动程序：

若块设备驱动程序也按以下字符设备驱动程序的简单思想来写：

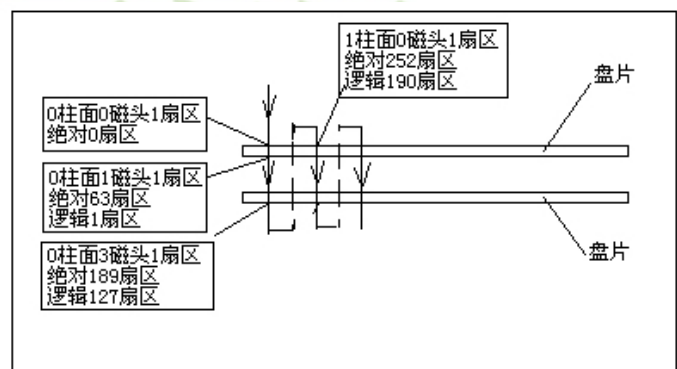
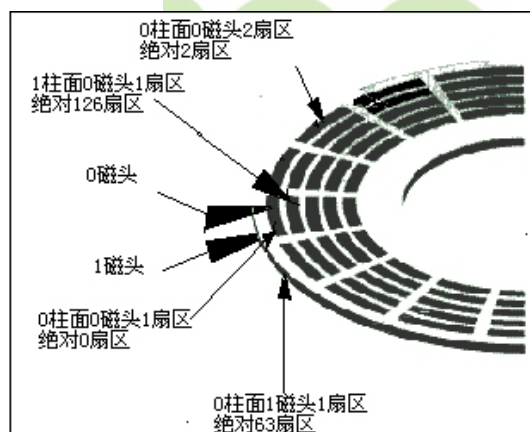
APP : open, read, write. ----> 对应提供驱动程序的读写等函数。

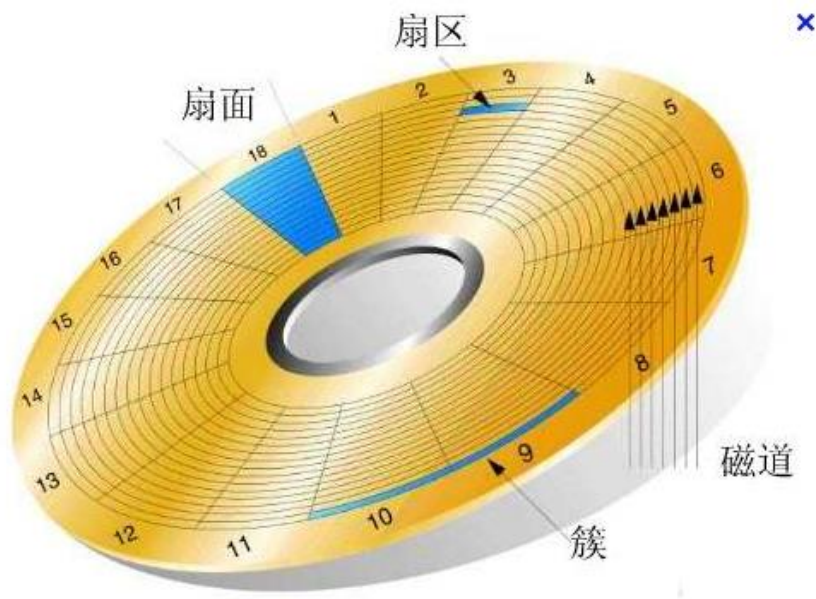
块设备驱动：drv_open, drv_read, drv_write

硬件：块设备。

举例：

1，硬盘：





磁盘上的磁道、扇区和簇

磁盘的读写其实非常快，慢在机械结构读写装置的定位上面，从一个“磁头”的某“柱面”某“扇区”读到数据后（步骤 R0），跳到另一个“磁头”的某“柱面”的某“扇区”去写（步骤 W），接着再跳回原“磁头”相同柱面的下一个“扇区”去读（步骤 R1）。慢就慢在读写扇区的跳转过程中。若按“字符设备”中的“open”, “read”, “write”方式，则总体效率在硬盘的读写上会非常低。上面过程是“R0”→“W”→“R1”，这个步骤跳转 2 次。若优化这个步骤为：R0→R1→W。这个步骤跳转 1 次。这样效率会高些。

总结：先不执行而是放入队列，优化后再执行（对硬盘有这种要求）。用“字符设备驱动”程序那样读写时就会在硬盘上跳来跳去，整体效率会非常低。所以有必要引入“优化过程”。就是读写先不执行，先放到某个“队列”中去。（调整顺序）

2, flash:

是“块”里有一个一个的扇区。

假若现在要先写“扇区 0”和“扇区 1”。FLASH 要先擦除再写，现在用字符设备驱动的读写方式来读写：

对 FLASH 上的擦除是整块整块的进行的。故：

写扇区 0 的过程：

- ①，要写时，先把这整块读到一个 buf 中。
- ②，然后修改 buf 中扇区 0 的数据。
- ③，这时再擦除整块。
- ④，再把修改过扇区 0 的数据的 buf 烧写到整块。

写扇区 1 的过程：

- ①，要写时，先把这整块读到一个 buf 中。
- ②，然后修改 buf 中扇区 1 的数据。
- ③，这时再擦除整块。
- ④，再把修改过扇区 1 的数据的 buf 烧写到整块。

则那么要修改多个扇区时，会擦除烧写多次。总体效率也会低。

优化：

①，先不执行。

②，优化 - 合并后执行。

合并：合并后只需要一次。

a, 读出整块到 buf 中。

b, 在 buf 中修改扇区 0 和扇区 1。

c, 擦除。

d, 烧写。

故，块设备不能像字符设备那样提供读写函数。

①，先把读写放入队列，先不执行。

②，优化后再执行。



块设备驱动框架

一，层次框架：

```
APP:      open, read, write "1.txt"
----- 文件读写
文件系统: vfat, ext2, ext3, yaffs      (把文件的读写转换成对扇区的读写)
----- ll_rw_block ----- 扇区读写
      块设备驱动程序
-----
```

硬件：硬盘、FLASH

对普通文件 1.txt 的读写会转成对块设备的读写，要读写哪个扇区。从文件的读写转成对扇区的读写，中间会涉及到“文件系统”。

应用程序读写一个变通的文件，最终会转换成操作硬件，由“块设备驱动程序”来操作硬件。

普通的文件转换成对扇区的读写，是由“文件系统”转换。

ll_rw_block 是通用的入口，读写请求会放入队列，优化后再来执行。ll_rw_block 会把“读写”放入队列，调用队列的处理函数去优化（调顺序、合并）执行。

如何知道是“ll_rw_block”可以看《LINUX 内核源代码情景分析》。文件系统不是我们关心的重点，分析块设备驱动程序，就是“ll_rw_block”这个函数开始分析。

二，流程：

分析“ll_rw_block”：1，把“读写”放入队列。2，调用队列的处理函数。

文件系统把一个普通文件的读写转换成块设备扇区的读写，最终就会调用这个底层的函数“ll_rw_block”。

这个函数在内核“fs”目录下，这个“fs”目录下有各种各样的文件系统（文件的组织格式-纯软件的概念。文件如何排，目录等如何表示），“fs”目录下还有很多通用的文件，ll_rw_block 这个函数在“buffer.c”是这个“fs”目录下的通用文件。

```
void ll_rw_block(int rw, int nr, struct buffer_head *bhs[])
```

参 1，表示是读或是写。

参 3，数据传输三要素（源，目的，长度）放到参 3 buffer_head 结构的数组中。这个数组有参 2nr 个数组项。

参 2，有 nr 个参 3 buffer_head 结构的数组项。

```
for (i = 0; i < nr; i++) //开始就 for 循环。
```

```

-->struct buffer_head *bh = bhs[i]; bh 等于这个参 3 数组的某一项。
-->submit_bh(WRITE, bh); 提交 buffer_head 结构的 bh。
    -->struct bio *bio;使用 buffer_head 来构造 bio (block input/output)。
下面就是用 buffer_head 结构体 bh 来构造 bio 的情况：

bio->bi_sector = bh->b_blocknr * (bh->b_size >> 9);
bio->bi_bdev = bh->b_bdev;
bio->bi_io_vec[0].bv_page = bh->b_page;
bio->bi_io_vec[0].bv_len = bh->b_size;
bio->bi_io_vec[0].bv_offset = bh_offset(bh);

bio->bi_vcnt = 1;
bio->bi_idx = 0;
bio->bi_size = bh->b_size;

bio->bi_end_io = end_bio_bh_io_sync;
bio->bi_private = bh;
    -->submit_bio(rw, bio); 提交 bio。
        -->generic_make_request(bio); 使用 bio(“通用的构造请求”)构造请求。把请求
        放入队列。
            -->__generic_make_request(bio);
                -->q = bdev_get_queue(bio->bi_bdev); 找到队列。request_queue_t *q;q
                是请求队列。
                    -->ret = q->make_request_fn(q, bio); 调用队列中一个函数“make 构造
                    request 请求的 fn 函数”。在如下地址设置“q->make_request_fn = mfn;” mfn 在下面这
                    个函数的形参里。
void blk_queue_make_request(request_queue_t *q, make_request_fn *mfn)
request_queue_t *blk_init_queue_node(request_fn_proc *rfn, spinlock_t *lock,
int node_id)
-->blk_queue_make_request(q, __make_request);
最终“make_request_fn()”的默认函数是“__make_request()”

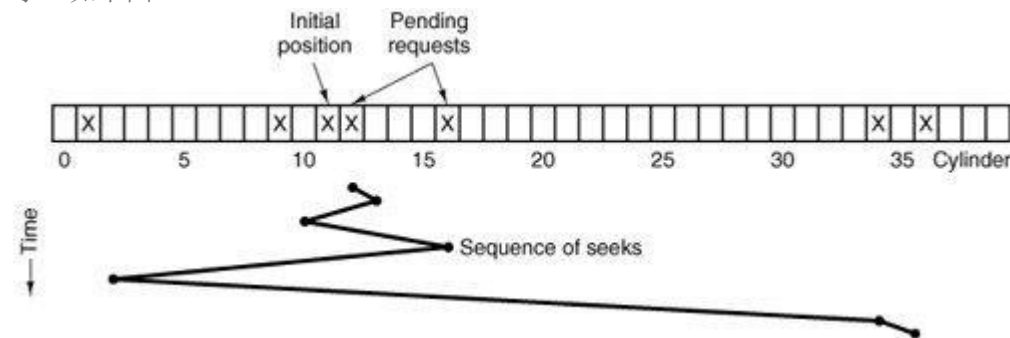
int __make_request(request_queue_t *q, struct bio *bio)
-->el_ret = elv_merge(q, &req, bio); merge 合并。这里是先尝试合并。elv 是电梯调
试算法。以 elv 算法把 bio 合并到请求队列。假若 1 楼 A 要上, 2 楼 B 要下, 三楼 C 要
上, 则不会让 A 上了再让 B 下了最后再跑到 3 楼让 C 上, 而是先让电梯从 1 楼上到头, 再
下来。一次传送中输运同样方向的人。(在一个方向上合并起来运输)。
-->init_request_from_bio(req, bio); 若合并不成功, 就用 bio 构造请求。
-->add_request(q, req); 把请求放到队列中去。
//下面接着就有一个执行队列的函数: _generic_unplug_device(q)
-->_generic_unplug_device(q); 执行队列, 但并不一定是在这个函数里执行。是 if
(sync)若同步时才执行。
    -->q->request_fn(q);其实执行队列的方式就是调用队列的处理函数(request_fn)。

```

电梯调度算法:

电梯算法主要用于磁盘寻道的优化。

第一种是我们最为原始的先到先服务(first come first served)的算法，这个对于我们去下馆子撮一顿比较合适，先来就先吃，不然顾客有意见。不过对于磁盘寻道就不太合适了。如下图：

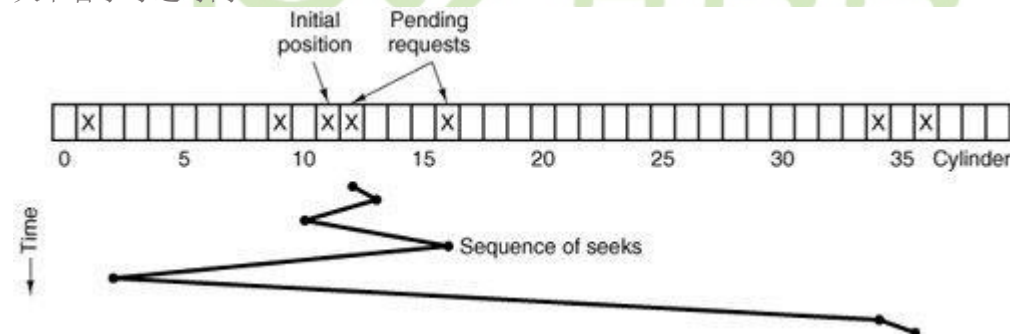


注意这张图并不是解释的先到先服务算法，我们只是借用下而已 :)

假设此时我们正在第 11 道读取数据，然后陆陆续续有其他进程来要求我们提供磁盘内容给他们。这里我们把要读取的柱面（如果你并不是研究磁盘寻道，那么这个词你可以理解为数据块，就是上面的小方块）按照进程提出要求的顺序记录下来的是 1, 36, 16, 34, 9, 12, 那么严格按照先到先服务原则，我们下一个要去的柱面是 1 号，中间要经历 10 个柱面，然后是 36 号..... 等全部读下来，我们统计下，一共要“跑过”111 个柱面。很明显的，这个算法效率太低，我们要来改善下算法。

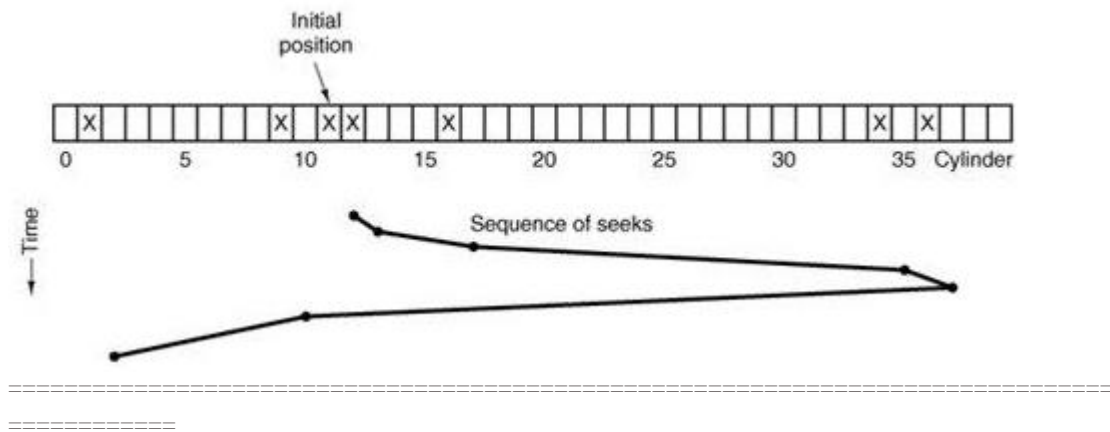
第二种是最短寻道算法(shortest seek first)

这种算法有点类似贪心，即是每次我们选择距离我们现在所处的点最近的一个点(柱面)。如下图，若当前我们正好执行完对于 11 号块的读取，下一个最近的是 12 号块，那么我们读取 12 号块的数据，接着读取 16 号块..... 我们看到如果用这种算法的话，我们经过的方块号码 12, 9, 16, 1, 34, 36 这样我们总共的经历的柱面数为 61 块，这样我们大大节省了寻道时间。



这个算法本来已经很好了，不过我们不得不面临这样一个问题： 现在我们正在读取 16 号块，马上要读取 1 号块了，这是一个进程闯进来要求我们为他提供 20 号块的信息，20 号距离 16 号比较近，那我们就去二十号吧，然后 我们又接到通知要 23 号数据..... 这样一直做下去，呃，1 号信息呢？天晓得要等到什么时候去读取它内容！

所以这里我们需要一种算法来平衡效率和公平性(我们也不希望歧视了 1 号小方块)。所以我们引进了电梯算法 。我们需要做一个标记，标记现在是向数字大的方向读，还是方向小的。如果现在是向前(数字大)读，那么我们就需要一直读下去，一直到最尾一个。同理向后读。这个算法如下图所示：



三，如何写块设备驱动程序：

LINUX 驱动程序中老种驱动都构造了一个结构体。

1，以面向对象的思想分配 gendisk 结构体。用 alloc_disk 函数。

2，设置 gendisk 结构体。

①，分配/设置一个队列：request_queue_t. （提供读写能力）用 blk_init_queue 函数。

②，设置 gendisk 其他信息。（提供磁盘属性：磁盘容量，扇区大小等）

3，注册 gendisk 结构体。用 add_disk 函数。

块设备驱动编写

分配一段内存，用内存来模拟硬盘：

搜索“blk_init_queue()”参考“drivers/block/xd.c”和“drivers/block/z2ram.c”两个文件。

大致分析：

看一个驱动程序从“入口函数”开始看。

Xd.c

```
register_blkdev(XT_DISK_MAJOR, "xd")
```

与之前的字符设备驱动的注册 register_chrdev() 相比，形参中少了一个“file_operations”结构体。其实这个注册块设备已经退化了。

只是在 Cat /proc/devices 可以让你看到一些信息，若者这个“XT_DISK_MAJOR”主设备号写为 0 时，register_blkdev(0, “xd”) 可以返回一个主设备号给你。

```
2, xd_queue = blk_init_queue(do_xd_request, &xd_lock);
```

初始化一个队列。

```
request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
```

形参 1 是在队列里传入一个处理函数“request_fn_proc”函数。

3, 如何使用队列 xd_queue:

-->struct gendisk *disk = alloc_disk(64); 分配一个 gendisk 结构体变量 disk

-->disk->queue = xd_queue; 将此分配的 gendisk 结构 disk 的成员“queue”等于之前初始化的队列“xd_queue”。

过程中还有主设备号，容量等等属性的设置。

```
4, add_disk(xd_gendisk[i]);
```

下面直接自己写代码：

一，入口函数：

```
//入口函数
static int ramblock_init(void)
{
    //1, 分配一个 gendisk 结构体


    //2, 设置
    //2.1, 分配/设置队列:提供读写能力

    //2.2, 设置其他属性:比如容量等等.

    //3, 注册
}
```

1, 分配一个 gendisk 结构体: alloc_disk()

alloc_disk (int minors)需要参数“minor+s”是指次设备号个数。即“分区个数+0”。0是指整个磁盘。为1时，就是把整个磁盘当成一个分区，则在上面不能再创建分区了。如写成16，则最多可以创建15个分区。若是说，这个主设备号之下，从哪个次设备号开始都是对应这个块设备。



```
book@book-desktop:~$ ls /dev/sd* -l
brw-rw---- 1 root disk 8, 0 2011-12-26 12:52 /dev/sda
brw-rw---- 1 root disk 8, 1 2011-12-26 12:52 /dev/sda1
brw-rw---- 1 root disk 8, 2 2011-12-26 12:52 /dev/sda2
brw-rw---- 1 root disk 8, 5 2011-12-26 12:52 /dev/sda5
brw-rw---- 1 root disk 8, 6 2011-12-26 12:52 /dev/sda6
brw-rw---- 1 root disk 8, 16 2011-12-26 12:52 /dev/sdb
brw-rw---- 1 root disk 8, 17 2011-12-26 12:52 /dev/sdb1
brw-rw---- 1 root disk 8, 32 2011-12-26 12:52 /dev/sdc
book@book-desktop:~$
```

对于一个块设备，次设备号为“0”时，表示整个磁盘。如“/dev/sda”。次设备号“1，2，、”表示是磁盘的第几个主分区。次设备号从5开始是“扩展分区”。

2, 分配/设置队列: blk_init_queue()

2, 设置

```
//2.1, 分配/设置队列:提供读写能力
ramblock_queue = blk_init_queue (do_rambloc_request, &ramblock_lock);
```

blk_init_queue (request_fn_proc * rfn, spinlock_t * lock) 分配/设置一个队列。

参1，执行处理队列的函数。

参2，一个“自旋锁”。 DEFINE_SPINLOCK (beep_lock)

之前分析把“文件读写”转成“扇区读写”，对“扇区的读写”会放入个队列里面：

把“buffer_head”构造为“bio”，把“bio”放入到队列里面，调用队列里面的“q->

make_request_fn”函数，这个“make_request_fn”构造请求的函数有默认的函数“__make_request”。

当我们初始化队列时，给我们提供了一个默认构造请求的函数“__make_request”。当它把这个“请求”放入到“队列”之后，以后会用这个队列里的“q->request_fn()”来处理。这个“request_fn()”就等于“blk_init_queue_node()”函数传进来的参数“形参1: request_fn_proc *rfn”。这个形参1会赋给“request_fn”，最后就是我们要定义的一个处理请求的函数。

```
request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
-->blk_init_queue_node(rfn, lock, -1);
    -->blk_queue_make_request(q, __make_request);提供了默认构造请求的函数。
```

对于我们这里这个把内存模拟成硬盘的驱动来说，最终会是我们这里定义的“do_rambloc_request”来处理请求。

3，设置其他属性：

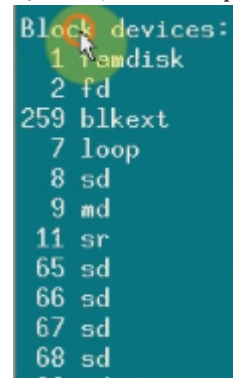
①，alloc_disk()：

②，主设备号：

```
static int major;
major = register_blkdev(0, "ramblock"); //自动分配主设备号
ramblock_disk->major = major; //属性:主设备号
```

主设备号可以自己定义，也可以让系统自动分配。Register_blkdev()函数退化了，用此函数当参1为“0”时可以让系统自动分配一个主设备号：

```
Static int major;
Major = register_blkdev(0, "ramblock");
Register_blkdev()相比“register_chrdev()”少了一个形参3“file_operations”结构体。只不过当形参1为“0”时，register_blkdev()可以自动为块设备分配一个主设备号，且在“cat /proc/device”时可以看到有哪些块设备：
```



A terminal window showing the output of the 'cat /proc/device' command. The output lists various block devices with their major numbers and names. The first few are 1 ramdisk, 2 fd, 259 blkext, 7 loop, 8 sd, 9 md, 11 sr, 65 sd, 66 sd, 67 sd, 68 sd.

③，第一个次设备号是什么 和 块设备的名字。

```
ramblock_disk->first_minor = 0; //第一个次设备号写为0，则从0~16都对应这个块设备。
printf(ramblock_disk->disk_name, "ramblock"); //块设备的名字
```

④, fops : 操作函数。即使是空的操作函数, 这个 fops 也要提供。

```
static struct block_device_operations ramblock_fops = {
    .owner          = THIS_MODULE,
    .ioctl          = ramblock_ioctl,
    .getgeo         = ramblock_getgeo,
};
```

经过实验, 即使这个函数里什么也没有 (如没有“.ioctl”和“.getgeo”), 也要提供这个结构体。不然会出错。

```
ramblock_disk->fops = &ramblock_fops; //必要的操作函数
```

```
(ramblock_disk->private_data = p;私有数据, 这里我们不需要。)
```

⑤, 设置队列:

把设置队列放在“分配队列”那一步骤代码紧跟的之下。就是把队列使用起来, 即放到 request_queue_t 结构变量 ramblock_queue 里中去。

//2.1, 分配/设置队列:提供读写能力

```
ramblock_queue = blk_init_queue (do_rambloc_request, &ramblock_lock);
ramblock_disk->queue = ramblock_queue; //设置队列
```

⑥, 容量: 设置容量时, 是以扇区为单位。

块设备容量 1M 字节

```
#define RAMBLOCK_SIZE (1024*1024)
set_capacity(ramblock_disk, RAMBLOCK_SIZE / 512);
```

单位是“扇区”, 在内核里, 对于文件系统那一层, 永远认为扇区是 512 字节。

4, 注册: add_disk().

//3, 注册

```
add_disk(ramblock_disk);
```

5, 处理请求:

在“分配/设置队列 blk_init_queue()”函数中, 形参 1 “request_fn_proc *rfn”是用来“处理”队列中的请求的函数。

//定义“处理队列请求”的函数

```
static void do_rambloc_request (request_queue_t * q)
{
    static int cnt = 0;
    printk("do_rambloc_request", ++cnt);
}
```

这里什么也没做, 以后慢慢完善。以上一个驱动程序基本上写完了。

二, 出口函数:

//出口函数


```
static void ramblock_exit(void)
{
    unregister_blkdev(ramblock_disk); //卸载块设备
    del_gendisk(ramblock_disk); //清除gendisk结构
    put_disk(ramblock_disk); //释放块设备结构空间
    blk_cleanup_queue (ramblock_queue); //清除队列。
}
```

这里是想用内存分模拟磁盘，但还未分配内存。这里先做实验。

三，测试：

1，编译

```
"Makefile" 10L, 177C written
book@book-desktop:/work/drivers_and_test/13th_ramblock/1th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M] /work/drivers_and_test/13th_ramblock/1th/ramblock.o
/work/drivers_and_test/13th_ramblock/1th/ramblock.c:70: warning: function declaration isn't a prototype
/work/drivers_and_test/13th_ramblock/1th/ramblock.c: In function `ramblock_init':
/work/drivers_and_test/13th_ramblock/1th/ramblock.c:67: warning: control reaches end of non-void function
  Building modules, stage 2.
  MODPOST 1 modules
  CC /work/drivers_and_test/13th_ramblock/1th/ramblock.mod.o
  LD [M] /work/drivers_and_test/13th_ramblock/1th/ramblock.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/13th_ramblock/1th$
```

上面的警告一个是“入口函数”中没有“return 0”，一个是“出口函数”的形参没有写“void”。

```
book@book-desktop:/work/drivers_and_test/13th_ramblock/1th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M] /work/drivers_and_test/13th_ramblock/1th/ramblock.o
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M] /work/drivers_and_test/13th_ramblock/1th/ramblock.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/13th_ramblock/1th$ cp ramblock.ko /work/nfs_root/first_fs
book@book-desktop:/work/drivers_and_test/13th_ramblock/1th$
```

```
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt/
# insmod ramblock.ko
ramblock:do_ramblock_request 1
```

```
static void do_ramblock_request (request_queue_t * q)
{
    static int cnt = 0;
    printk("do_ramblock_request", ++cnt);
}
```

加载驱动后，马上用到了“do_ramblock_request”队列请求处理函数。但到这里后就再也不能返回到 shell 了。因为这个函数对“队列 request_queue_t * q”没有任何处理。所以没有再返回。

看内核示例“drivers/block/xd.c”中的“do_xd_request(request_queue_t * q)”：

以“电梯调度算法”从“request_queue_t * q”队列里面取出下一个请求“req = elv_next_request(q)”，然后执行这个请求“req”。最后“end_request(req, res); /* wrap up, 0 = fail, 1 = success */” res 即 0 或 1。

将“队列请求处理函数”修改为下面：

//定义“处理队列请求”的函数

```
static void do_rambloc_request (request_queue_t * q)
{
    static int cnt = 0;
    struct request *req;
    printk("do_rambloc_request", ++cnt);

    while( (req = elv_next_request(q)) != NULL ){ //以电梯调度算法取出队列下一个请求,但没处理直接返回.
        end_request(req, 1);
    }
}
```

```
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt/
# insmod ramblock.ko
ramblock:do_rambloc_request 1
unknown partition table
#
```

可以返回到 SHELL，只是说了不识别的分区表。

这样便可以装载块驱动后返回。在“while”中一般是在这里做真正的操作硬件。若是真正的磁盘，则从队列里取请求开始读写操作即可。这里实验目的是“内存模拟磁盘”则一会用 memcpy 操作内存即可完成读写。

```
# ls -l /dev/ramblock
brw-rw---- 1 0 0 254 0 Jan 1 00:00 /dev/ramblock
```

```
179 mmc
254 ramblock
```

四，最终完善的内存模拟磁盘：

1，“硬件相关操作”：分配内存。

```
//声明一个块内存
static unsigned char *ramblock_buf;

//3，硬件相关操作：
ramblock_buf = kzalloc (RAMBLOCK_SIZE, GFP_KERNEL); //分配内存
```


2，在“队列请求”处理函数中操作：实现内存模拟磁盘。

①，数据传输三要素：

涉及到数据的传输，则离不开“三要素”：块设备数据传输的三要素在结构体“request”中定义：

```
struct request {
sector_t sector;          /* next sector to submit 下一个要提交的扇区。这就源或是目的*/
...
unsigned int current_nr_sectors; //当前要处理的扇区个数。这是长度。
...
char *buffer; //要写时这时目的(从扇区读数据到此)，要读时这是源。
}
```

块设备操作时是以扇区为单位，即使是写一个字节，也是先读出一个扇区，再修改这个扇区里的那个字节后，再将整个扇区写进去。

②，读写的方向：

```
while( (req = elv_next_request(q)) != NULL ){ //以电梯调度算法取出队列下一个请求,但没处理直接返回.

    //数据传输三要素:源,目的,长度 结构request中会有这三要素.
    //3.3, 源 或者 目的
    unsigned long offset = req->sector <<9; //偏移值。左移9位相当于乘以512-1个扇区大小.
    故相当于 req->sector*512.
    //3.4, 目的 或者 源 : req->buffer

    //3.5, 长度
    unsigned long len = req->current_nr_sectors <<9; //长度。当前要传输多少个扇区.

    //方向:读写。
    if(rq_data_dir(req) == READ) //若是读
    { //memcpy(目的, 源, 长度);
        memcpy(req->buffer, ramblock_buf+offset, len); //从磁盘上的源读长度len到目的内存的buffer。
    }
    else //写则倒过来
    {
        memcpy(ramblock_buf+offset, req->buffer, len); //把内存buffer中的数据写到磁盘上.
    }

    end_request(req, 1);
}
```

这样这后，一会就可以格式化，挂接内容了。

3，再编译后测试：

①. insmod ramblock.ko

```
# rmmod ramblock
# insmod ramblock.ko
ramblock: unknown partition table
# █
```

还是提示了“未识别的分区表”，这是因为分配的内存全都是清 0 的。
ramblock_buf = kzalloc (RAMBLOCK_SIZE, GFP_KERNEL); //分配内存
里面还没有分区表。

②. 格式化: mkdosfs /dev/ramblock

```
# mk
mkdir      mke2fs      mkfs.minix  mkswap
mkdosfs    mkfifo     mknod      mktemp
# mkdosfs /dev/ramblock
mkdosfs 2.11 (12 Mar 2005)
unable to get drive geometry. using default 255/63#
#
```

此时，系统里只有“mkdosfs”，就用这个。

③. 挂载: mount /dev/ramblock /tmp/

```
# cd /tmp/
# mount /dev/ramblock /tmp/
#
```

④. 读写文件: cd /tmp, 在里面 vi 文件等都可以。

⑤. cd /; umount /tmp/ 后, 重新挂载时, 里面的内容还存在。

```
# mount /dev/ramblock /tmp/
# ls /tmp/
1.txt  inittab
#
```

⑥. cat /dev/ramblock > /mnt/ramblock.bin

可以把整个磁盘 cat 成一个 bin 文件。相当于一个镜像。
上面都是在开发板上操作的。

⑦. 在 PC 上查看 ramblock.bin

```
sudo mount -o loop ramblock.bin /mnt
```

(-o loop 回还设备: 可将一个普通文件当成一个块设备文件挂载。)

```
book@book-desktop:/work/nfs_root/first_fs$ ls ramblock.bin
ramblock.bin
book@book-desktop:/work/nfs_root/first_fs$
book@book-desktop:/work/nfs_root/first_fs$ sudo mount -o loop ramblock.bin /mnt
[sudo] password for book:
book@book-desktop:/work/nfs_root/first_fs$ cd /mnt
book@book-desktop:/mnt$ ls
1.txt  inittab
book@book-desktop:/mnt$ cat 1.txt
hello
book@book-desktop:/mnt$
```

在 P C 机上也可以看到之前在开发板上创建的两个文件。证明块设备实验成功了。

用内存来模拟, 忽略了硬件的复杂操作。这里只用了“m e m c p y ()”就实现了硬件操作。

进一步做实验，把“队列请求”处理函数中读写打印出来。

```
static void do_ambloc_request (request_queue_t * q)
{
    static int r_cnt = 0;
    static int w_cnt = 0;
    struct request *req;
    // printk("do_ambloc_request", ++cnt);

    while( (req = elv_next_request(q)) != NULL ) { //以电梯调度算法取出队列下一个请求, 但没处
理直接返回.
        //数据传输三要素:源, 目的, 长度 结构request中会有这三要素.
        //3.3, 源 或者 目的
        unsigned long offset = req->sector <<9; //偏移值. 左移9位相当于乘以512-1个扇区大小.
        故相当于 req->sector*512.
        //3.4, 目的 或者 源 : req->buffer

        //3.5, 长度
        unsigned long len    = req->current_nr_sectors <<9; //长度. 当前要传输多少个扇区.

        //方向:读写.
        if(rq_data_dir(req) == READ) //若是读
        { //memcpy(目的, 源, 长度);
            printk("do_ambloc_request read %d\n", ++r_cnt);
            memcpy(req->buffer, ramblock_buf+offset, len); //从磁盘上的源读长度len到目的内存
的buffer.
        }
        else //写则倒过来
        {
            printk("do_ambloc_request write %d\n", ++w_cnt);
            memcpy(ramblock_buf+offset, req->buffer, len); //把内存buffer中的数据写到磁盘上.
        }

        end_request(req, 1);
    }
}
```

编译测试:

```
# rmmod ramblock
# insmod ramblock.ko
ramblock:do_ambloc_request read 1
unknown partition table
# mbl
```

加载驱动时读了一次。

```
# mkdosfs /dev/ramblock
mkdosfs 2.11 (12 Mar 2005)
do_ramblock_request read 2
do_ramblock_request read 3
unable to get drive geomdo_ramblock_request write 1
do_ramblock_request write 2
do_ramblock_request write 3
do_ramblock_request write 4
do_ramblock_request write 5
etry, using default 255/63#
#
```

格式化时读写次数。

```
# mount /dev/ramblock /tmp
do_ramblock_request read 4
do_ramblock_request read 5
do_ramblock_request read 6
do_ramblock_request read 7
do_ramblock_request read 8
do_ramblock_request read 9
do_ramblock_request read 10
do_ramblock_request read 11
do_ramblock_request read 12
do_ramblock_request read 13
do_ramblock_request read 14
do_ramblock_request read 15
```

```
do_ramblock_request read 39
do_ramblock_request read 40
do_ramblock_request read 41
do_ramblock_request read 42
#
```

挂接时从， r e a d 4 一直到了 r e a d 4 2

```
# cp /etc/inittab /tmp
do_ramblock_request read 43
#
```

拷贝一个文件到磁盘上，发现也是“读”。

这是并没有立即写。块设备的读写操作会先放到队列里面，并不会立即执行。

```
# do_ramblock_request write 6
do_ramblock_request write 7
do_ramblock_request write 8
do_ramblock_request write 9
```

到了“read 43”后，等了好些时候才出现上面的写。

有时候在 WINDOWS 上可发现 拷贝文件到 U 盘的“进度条”已完成，这时去卸载 U 盘时，会提示说“设备忙”，U 盘的灯也在闪。这表示在后台还要写。

但是要想让写到块设备上的文件立即写入块设备：sync（同步）

```
# cp /etc/init.d/rcS /tmp
# sync
do_ramblock_request write 11
do_ramblock_request write 12
do_ramblock_request write 13
do_ramblock_request write 14
do_ramblock_request write 15
#
```

sync 是一个系统调用，同步的意思。

```
# cp /mnt/ramblock.ko /tmp
# cd /
# umdo_ramblock_request write 16
do_ramblock_request write 17
do_ramblock_request write 18
# umount /tmp/
do_ramblock_request write 19
do_ramblock_request write 20
do_ramblock_request write 21
do_ramblock_request write 22
do_ramblock_request write 23
do_ramblock_request write 24
do_ramblock_request write 25
do_ramblock_request write 26
do_ramblock_request write 27
do_ramblock_request write 28
do_ramblock_request write 29
do_ramblock_request write 30
do_ramblock_request write 31
do_ramblock_request write 32
do_ramblock_request write 33
do_ramblock_request write 34
do_ramblock_request write 35
do_ramblock_request write 36
do_ramblock_request write 37
do_ramblock_request write 38
do_ramblock_request write 39
do_ramblock_request write 40
#
```

再写一个数据到块设备，发现拷贝命令结束时，并没有立即去写，这时“umount”这个块设备。就立即开始去写了。

上面的现象都是要么读，要么写，就像电梯一样，一路先上到顶，再下来。

五，开始试着分区：

```
# fd
fdflush  fdformat  fdisk
# fdisk /dev/ramblock
do_ramblock_request read 44
do_ramblock_request read 45
do_ramblock_request read 46
do_ramblock_request read 47
Unknown value(s) for: cylinders (settable in the extra functions menu)
Command (m for help):
```

不知道“柱面”数。现在可能有的磁盘已经没有这种结构了，但是为了兼容这些“fdisk”老工具，要假装说自己有多少个“磁头”，多少个“柱面”。这些信息就是由“block_device_operations（块设备的操作）”

```

struct block_device_operations {
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned, unsigned long);
    int (*direct_access) (struct block_device *, sector_t, unsigned long *);
    int (*media_changed) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo) (struct block_device *, struct hd_geometry *);
    struct module *owner;
};

```

其中:

```
int (*getgeo)(struct block_device *, struct hd_geometry *);
```

geometryr 几何的意思。获得几何属性。

//定义一个操作函数，即使是空的操作函数也一定要。

```

static struct block_device_operations ramblock_fops = {
    .owner = THIS_MODULE,
    //.ioctl = ramblock_ioctl,
    .getgeo = ramblock_getgeo,
};

```

要加上 “.getgeo = ramblock_getgeo” 这个函数。用内存模拟磁盘，没有磁头，柱面，扇区的概念，但为了用这个 “fdisk” 老工具，也得假装有这些属性。

```

static int ramblock_getgeo(struct block_device *bdev, struct hd_geometry *geo)
{ //容量 = heads * sectors * cylinders * 512
    geo->heads = 2; //假设有 2 磁头
    geo->sectors = 32; //假设有 32 柱面
    geo->cylinders = RAMBLOCK_SIZE/2/32/512; |
    return 0;
}

```

容量 = 磁头 * 柱面 * 扇区 * 512 (容量 = heads * sectors * cylinders * 512)

磁头: 即有多少面。这里假设有 2 面。

柱面: 有多少环, 这里假设有 32 环。

扇区: 一环里有多少个扇区, 这个得通过 “公式” 计算出来。

重新加载有 “磁头、柱面、扇区” 信息的驱动。

```

# rmmod ramblock
# cd /mnt/
# insmod ramblock.ko
ramblock: unknown partition table
#

```

上面说没有识别的分区表，还是因为分配空间时，里面都是 0，并没有分区信息。

```
ramblock_buf = kzalloc (RAMBLOCK_SIZE, GFP_KERNEL); //分配内存
```

```

# ls /dev/ramblock* -l
brw-rw---- 1 0 0 254, 0 Jan 1 00:22 /dev/ramblock
#

```

上面次设备号为 “0” 表示整个磁盘。


```
# fdisk /dev/ramblock
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.

Warning: invalid flag 0x00,0x00 of partition table 4 will be corrected by w(rite)

Command (m for help):
```

给这个 1M 的空间分区。

```
Command (m for help): m
Command Action
a      toggle a bootable flag
b      edit bsd disklabel
c      toggle the dos compatibility flag
d      delete a partition
l      list known partition types
n      add a new partition 添加一个新分区
o      create a new empty DOS partition table
p      print the partition table
q      quit without saving changes
s      create a new empty Sun disklabel
t      change a partition's system id
u      change display/entry units
v      verify the partition table
w      write table to disk and exit
x      extra functionality (experts only)

Command (m for help):
```

“n” 添加一个新分区。

```
Command (m for help): n
Command action
e      extended
p      primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-32, default 1):
```

```
geo->sectors = 32; //假设有 32 柱面
```

之前代码中构造几何信息时，柱面定的是 32 个。

添加主分区。第一个“柱面 1-32”。Default 1:默认柱面为 1. 这里保持默认。

```
Partition number (1-4): 1
First cylinder (1-32, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-32, default 32): 5
```

接着是结束柱面，这里定为“5”。

```
Command (m for help): p

Disk /dev/ramblock: 1 MB, 1048576 bytes
2 heads, 32 sectors/track, 32 cylinders
Units = cylinders of 64 * 512 = 32768 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/ramblock1    1             5        144    83  Linux

Command (m for help):
```

这时可以查看，有一个分区出来了。

下面再增加一个分区：

```
Command (m for help): n 新建一个分区
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 2 主分区2
First cylinder (6-32, default 6): 6 这个主分区2的默认柱面从6开始
Last cylinder or +size or +sizeM or +sizeK (6-32, default 32): Using default value 32
                                          最后一个柱面直接设置成32
Command (m for help): p 查看分区信息

Disk /dev/ramblock: 1 MB, 1048576 bytes
2 heads, 32 sectors/track, 32 cylinders
Units = cylinders of 64 * 512 = 32768 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/ramblock1    1           5        144    83  Linux
/dev/ramblock2    6          32       864    83  Linux 有了两个分区

Command (m for help):
```

最后写入硬盘：

```
Command (m for help): w
The partition table has been altered!
ramblock: ramblock1 ramblock2

Calling ioctl() to re-read partition table
# █
```

所谓分区表就是 磁盘 里的第一个扇区。

```
calling ioctl() to re-read partition table
# ls /dev/ramblock* -l
brw-rw----  1 0      0      254, 0 Jan  1 00:25 /dev/ramblock
brw-rw----  1 0      0      254, 1 Jan  1 00:25 /dev/ramblock1
brw-rw----  1 0      0      254, 2 Jan  1 00:25 /dev/ramblock2
#
```

次设备号为“0”表示整个磁盘。

次设备号为“1”表示磁盘上的第一个主分区。

次设备号为“2”表示磁盘上的第二个主分区。

```
# mkdosfs /dev/ramblock1
mkdosfs 2.11 (12 Mar 2005)
# mkdosfs /dev/ramblock2
mkdosfs 2.11 (12 Mar 2005)
# mount /dev/ramblock2 /mnt
# mount /dev/ramblock1 /tmp
# █
```

这时可以分别格式化，分别挂载。

若以后，硬盘也有像内存一样直接用“memcpy()”直接像内存一样访问时，那块设备就非常的简单了。但现在这只是美好的愿望。现在还是要把“memcpy()”替换成其他方法。这将是块设备复杂的地方。但块设备的框架如上所述，却是并不复杂的。

框架：

LINUX 驱动程序中老种驱动都构造了一个结构体。

```
APP:      open, read, write "1.txt"
----- 文件读写
文件系统: vfat, ext2, ext3, yaffs      (把文件的读写转换成对扇区的读写)
----- ll_rw_block ----- 扇区读写
      块设备驱动程序
-----
```

硬件：硬盘、FLASH

1，以面向对象的思想分配 gendisk 结构体。用 alloc_disk 函数。

2，设置 gendisk 结构体。

①，分配/设置一个队列：request_queue_t. （提供读写能力）用 blk_init_queue 函数。

②，设置 gendisk 其他信息。（提供磁盘属性：磁盘容量，扇区大小等）

3，注册 gendisk 结构体。用 add_disk 函数。

操作不用我们关心，格式化、读写文件等都是由“文件系统”这一层将文件的读写转换成对扇区的读写的。调用“ll_rw_block”会把读写放到你的队列中去。会调用你“队列请求处理函数”来处理。只要你写好“队列请求处理”函数即可。

