块设备驱动实战基础篇一 （170行代码构建一个逻辑块设备驱动）

2014-03-17 15:47:58

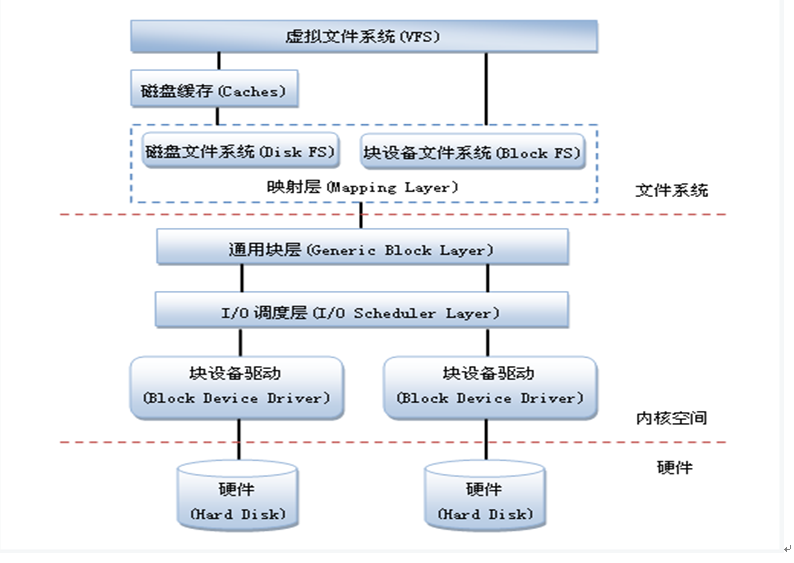
分享：

1   内核块设备驱动基础学习与实战

1.1 设备驱动IO架构初探

操作系统是如何将数据读到缓冲区的，发生了什么？我们带着这样的问题，粗略走一下read调用系统过程，希望这个初探，可以唤起大家研究操作系统内核的好奇心和兴趣，并以此为例，让我们先初步对请求在过滤块设备驱动中的处理过程有个大概印象和了解。

块设备在整个Linux中应用的总体结构图如下：



从上图可以看出，块设备的应用在Linux中是一个完整的子系统。 最上面是虚拟文件系统层，其作用是屏蔽下层具体文件系统操作的差异，为上层的操作提供一个统一的接口。有了这个层次，可以把设备抽象成文件，使得操作设备就像操作文件一样简单。在具体的文件系统层中，不同的文件系统（例如 ext2 和 NTFS）具体的操作过程也是不同的，每种文件系统定义了自己的操作集合。引入 cache 层的目的是为了提高 linux 操作系统对磁盘访问的性能，cache 层在内存中缓存了磁盘上的部分数据，当数据的请求到达时，如果在 cache 中存在该数据且是最新的，则直接将数据传递给用户程序，免除了对底层磁盘的操作，提高了性能。

接下来是通用块层，其主要工作是：接收上层发出的磁盘请求，并最终发出 IO 请求。该层隐藏了底层硬件块设备的特性，为块设备提供了一个通用的抽象视图。然后下面是IO 调度层，其功能是接收通用块层发出的 IO 请求，缓存请求并试图合并相邻的请求（如果这两个请求的数据在磁盘上是相邻的）。根据设置好的调度算法，回调驱动层提供的请求处理函数，来处理具体的 IO 请求。驱动层中的驱动程序对应具体的物理块设备，它从上层中取出 IO 请求，并根据该 IO 请求中指定的信息，通过向具体块设备的设备控制器发送命令的方式，来操纵设备传输数据。

大家都写过read读文件的数据，你想知道或者探秘read到底是如何把数据读上来的吗，操作系统是如何处理的，linux是如何处理的，从现在开始让我们逐渐养成寻根究底的学习方法，能够主动思考或者探秘操作系统行为，这样才能逐渐理解操作系统的工作原理，逐渐理解linux内核的设计艺术和实现原理，也才能够逐渐往高手的水平上迈进。让我们分析一下一个读请求从应用层到内核层的全过程，来分析read系统调用的真正工作原理，也充分理解一下请求是如何走到块设备驱动层，然后块设备做了哪些处理把请求最终提交到磁盘完成数据请求的，这样就为块设备学习开启了入门之路。

关键路径点初探：

   从应用层开始调用glibc库的read函数，经过glibc库的处理，最终通过操作系统内核提供的sys\_read函数系统调用进入内核。［再回头看看第一章中的内核系统架构图，内核klib库中系统调用是内核一个机制，系统调用中发生了什么事，cpu相关关键寄存器做了什么切换，我们现在先提出来，先不具体讲解，继续我们的主线初探过程］

   进入内核sys\_read函数处理，我们进入了内核IO路径上的第一个模块层VFS虚拟文件系统层，首先根据sys\_read函数中指定的文件描述符和要访问的文件数据起始地址，去内存上找一找，是否内存中已经缓存了我们要读取的文件数据，如果有，直接从内存拷贝一份到我们申请的buffer中即可，在这里我们知道了一个很关键的操作系统知识点，应用read函数提供的buffer 与  内核内存缓存是不一样的，内核中管理的内存数据需要拷贝给用户空间的buffer才可以，用户空间的程序是不能直接访问和使用内核中的缓存数据的；如果没有缓存数据，则sys\_read会继续往下走，首先VFS会为我们要读取的文件数据单独申请一下内核内存空间，这个内存空间是后续从磁盘真正把数据读上来时要存放的地方，我们不能直接往用户空间read函数提供的buffer中存放数据，这是linux内核架构机制决定的，我们先记住这是一个约定，是否感觉内核好麻烦，已经有内存buffer了还要申请内存，这样的事情后续还有很多，我们通过课程的逐渐加深会一一为大家解答，但是现在先不要过于心急，继续我们的初探之路；

   VFS为其分配了缓存后，sys\_read便从VFS 层进入到了具体文件系统层的IO代码处理中，为啥要进入具体文件系统，因为只有文件系统才知道文件数据在磁盘的真正位置，所以既然接下来要从磁盘上获取数据了，那一定要通过具体文件系统知道文件数据的真实磁盘位置，我们从read函数的接口输入参数上也可以看出，我们并没有指定我们从磁盘那个位置进行读取；

   具体文件系统转换请求地址后，会构造块设备请求（bio），我们由此接触到了过滤块设备驱动第一个最核心的数据结构 －块设备请求描述数据结构bio，就是block device input/output的缩写。文件系统会把bio提交给过滤块设备驱动。

   过滤块设备驱动收到请求后，会继续转发给底层真实的硬件磁盘驱动，进而由其进行数据读取操作。

整个过程的初探之路结束了，这里面没有涉及代码的细节，只是通过初探让大家先整体上粗略的走一下真实的操作系统内核处理请求的过程，接下来让我们赶紧开始实战吧，真正的挑战要开始了。

1.2 快捷实战170行代码逻辑块设备驱动

经过第一节一个简单的I/O路径的数据流动介绍，现在让我们开始真正的进入逻辑块设备驱动的学习，因为linux内核中的关键模块和术语太多了，较为抽象，但是理解和掌握后，相信大家一定会感觉事情就是这么简单，也就会觉得为什么要把它描述的那么复杂和抽象，不就是那么回事吗，所以我们首先先给I/O 路径上的各个模块简单打个比方，用一个较为形象的例子带领大家脚踏实地，完成一个简单过滤块设备驱动模块，进而以此为基础，把操作系统／国内外讲解内核的书中经常提到的自旋锁，信号量，进程，内存分配，工作队列，实践在我们的过滤块设备驱动模块中，由此我们不仅可以在内核里面开发了，同时我们开发的模块在内核中是多么的重要，同时通过我们开发的模块还能快速把一些内核机制／API使用上，一切是那么的自然，这就是我们的目标，我们不希望千篇一律再把庞大的内核翻来覆去的抽象讲解，我们就是要脚踏实地，踏踏实实的真正进入内核做开发，通过过滤块设备驱动的编写，会为我们成功打开内核学习和实战的窗口，从而为后续内核分析及修炼之路打下坚实的实战技能基础。

**［小知识：块设备与字符设备］**

*系统中能够随机读取（不需要按顺序）访问固定大小数据片（chunk）的设备被称作块设备，这些数据片就称作块。最常见的块设备是硬盘，除此之外，还有软盘驱动器、CD-ROM驱动器和闪存等许多其他设备。它们都是以安装文件系统的方式使用的——这也是块设备通常的访问方式。块设备分为物理块设备（实际的磁盘）和逻辑块设备（磁盘分区，LVM等）。块设备可以用来创建文件系统、加载卸载、存储数据，也可以用来创建分区。*

*Linux设备的每个设备都由唯一的一个设备号标识，设备号由主设备号和次设备号组成；主设备号标识设备的类型及对应的驱动程序；次设备号对应具体的设备。Linux系统负责管理全局设备号，设备驱动负责申请设备号（可以通过ll /dev/xxx命令查看设备号；cat/proc/devices可以查看系统中已使用的设备号）。*

*与字符设备的区别：*

*字符设备按照字符流的方式被有序访问，如串口和键盘就都属于字符设备，如果一个硬件设备是以字符流的方式访问的话，那就应该将它归于字符设备，反过来，如果一个设备是随机（无序的）访问的，那么它就属于块设备。根本区别是它们能否可以被随机访问，也就是说，能否在访问设备时随意的从一个位置跳转到另一个位置。块设备只能以块为单位接受输入和返回输出，而字符设备以字节为单位，只能被顺序读写。*

我们仍然以一个读请求的处理过程为例进行，首先大家先看一个例子，最近X公司耗费精力筹写了一本巨著，书内容非常庞大，该公司以512页为单位，将这本书分开存放在一个秘密的存储室里，由于这本书内容太庞大，并且就只有一本，许多读者想借阅，为了满足大家的需要，X公司规定大家把每次需要借阅的书的页码起始数和页数准备好，公司会根据读者的需要找到书后进行复印，把复印件提供给读者。

写到这，大家应该能够明白X公司相当于文件系统，它知道读者需要的页码对应的内容在存储室的哪个位置存放，存储室就相当于我们的磁盘。Ok，我们继续把这个事情打比喻，请大家继续耐心。

X公司为了竭力保护好存储室，他们把存储室的位置放置在城市的郊区，同时为了更好的服务读者，他们在市区租用了一个小型的临时存储室，预先存放书籍的部分复印件，主要是考虑到大部分读者都在市区工作，郊区太远，不方便，如果临时存储室有读者需要的书籍，则直接复印给读者，如果没有则去郊区的存储室找到文件进行复印。同时为了更加安全的考虑，X公司在市区和郊区之间构建了三个虚拟的存储室站点A,B,C，样子看上去象是一个很大的实体存储室，其实里面空空的，什么都没有，只是个样子而已，空间都是虚拟的，之所以这么做，是X公司想更好的保护图书，以免出现安全问题。这样市区到郊区的路程就变为，市区－>A站点－>B站点－>C站点－>郊区。同时规定市区的人员只知道书存放在A站点，A站点的人员会知道书其实存放在B站点，B站点的人员其实只知道书存放在C站点，而C站点才真正知道书存放在郊区的地点。

写到这，大家又会进一步明白，市区的临时存储室相当于VFS缓存层，而A/B/C就是我们要给大家介绍的过滤块设备驱动模块，大家可以看到过滤块设备驱动可以层叠很多，我们现在是三个过滤块设备叠加。

       我们继续再定义几个概念，读者在市区借阅，如果市区的临时存储室没有复印件，则X公司会准备一个快递包，这个快递包就是个空盒子，里面什么也没有，但是要注意空盒子虽然是空的，但是不是谁都可以申请到的，空盒子有数量限制，如果申请太多了，读者就会被告知现在资源比较紧张，请稍等。好了，如果空盒子申请到了，那么是否就可以开始发送了，稍等，我们还要在盒子上做点标记，起码我们要写上读者要读的书的页码和页数，当然还有最重要的一个标记就是这个空盒子的目的地，即这个盒子要发到A这个站点，A站点收到盒子后，会继续发给B，此时它要把盒子的目的地标记修改为B, B收到后修改目的地，会继续发给C, C会继续发到X公司的郊区站点。

       好了，以上提到的空盒子就是大名鼎鼎的BIO, 它就是描述一个请求的。当这个请求也就是空盒子被发到郊区站点时，注意此时还有一道关卡，这到关卡准备了一些大箱子，内容页码连续的空盒子被放在同样的大箱子中，关卡会暂时缓存一下这些大箱子，等空盒子差不多积攒的够多了，然后统一送到最终的存储室，最后书籍会按照需求复印出来。上面的大箱子就是request, 而关卡就是request\_queue请求队列。

此时我们再介绍三个概念：gendisk，hd\_struct和block\_device, 不管是临时站点A/B/C, 还是郊区的存储点，都会对应一个gendisk描述结构，该结构描述了临时站点或者存储点的门牌号（major,minor号码），存储容量大小等信息，虽然A/B/C都是虚拟的站点，但是也被写上了一个虚拟容量，让大家看上去象那么回事，感觉就是一个真实的物理磁盘块设备。然后是hd\_struct，大家都知道对磁盘进行分区，一个分区就会用一个hd\_struct结构进行描述，记录分区的大小，起始位置等信息。最后至于block\_device结构，这个结构其实也是描述设备信息的，同时它也要描述文件系统相关的部分信息，具体的我们现在先不介绍，我们仍然以数据流动的过程为主，暂时先不跟进具体的细节信息，我们只要明确一件事情，一个gendisk会对应的一个block\_device,  如果gendisk有分区，则每个分区hd\_struct也会对应一个block\_device。

至此我们介绍了六个主要的数据结构，下面让我们继续描述如何构建块设备过滤驱动。还是接着借书的例子，A/B/C要建立虚拟仓库，首先我们要向操作系统申请注册并申请门牌号，这个申请的接口就是（register\_blk\_device）, 申请了门牌号，接着需要申请仓库了，也就是我们的gendisk, 通过alloc\_disk完成，接着我们要申请仓库的关卡－请求队列，通过alloc\_queue完成，当然我们要申明一下，并不是所有的虚拟仓库都会用到这个关卡即请求队列，我们举的例子中是没有使用的，后面我们会举例使用关卡的情况，在我们这个例子中，只有郊区的存储室使用了请求队列，如下图，我们再描述一下看看，A/B/C三个虚拟仓库都有自己的请求队列，但是都没有实际使用，这当然可以，操作系统并没有强制规定，必须使用请求队列，这是没问题的。这此我们介绍了三个很重要的虚拟仓库申请函数，接下来我们让重点描述一个函数－ make\_request, 我们继续举例，A/B/C每个临时仓库，接收到请求后要进行加工处理，这就是make\_request函数，这也是过滤块设备驱动最核心的地方，那么我们如何把这个函数注册到仓库里呢，通过blk\_queue\_make\_request这个函数注册的，是不是非常简单。到现在为止，我们已经把过滤块设备驱动要构建的操作全部描述完了，是的，事情就是那么回事，从此请求会经过每层过滤驱动的make\_request函数依次传递下去，如何传递，我们会继续介绍，但现在，让我们先离开一下这个例子，开始分析一下代码，从代码上充分体会一下我们如何构建一个最简单的过滤块设备驱动。

首先我们继续假设一种情况，假设读者去市区借书，没有复印件，我们继续去A那拿，不幸的是，A 压根儿不理睬我们，直接拒绝了我们，好悲哀啊。接下来，我们就是用170行代码，把这个过程的程序代码呈现给大家。

让我们再记住这几个步骤：

1.                                                                                                注册并申请门牌号： register\_blkdev

2.                                                                                                申请仓库：alloc\_disk

3.                                                                                                申请仓库的关卡：alloc\_queue

4.                                                                                                注册仓库的加工处理函数：blk\_queue\_make\_request

让我们看一下内核代码是如何写的。首先我们先给出全部的170行代码，然后我们会从module\_init函数开始阅读理解，把我们上面提到的步骤一步一步验证一下，下面是全部的代码。

**fbd\_driver.h**

  1#ifndef  \_FBD\_DRIVER\_H

  2#define  \_FBD\_DRIVER\_H

  3#include <linux/init.h>

  4#include <linux/module.h>

  5#include <linux/blkdev.h>

  6#include <linux/bio.h>

  7#include <linux/genhd.h>

  8

  9#define SECTOR\_BITS             (9)

 10#define DEV\_NAME\_LEN            32

 11#define DEV\_SIZE                (512UL<< 20)   /\* 512M Bytes \*/

 12

 13#define DRIVER\_NAME            "filter driver"

 14

 15#define DEVICE1\_NAME           "fbd1\_dev"

 16#define DEVICE1\_MINOR           0

 17#define DEVICE2\_NAME           "fbd2\_dev"

 18#define DEVICE2\_MINOR           1

 19

 20struct fbd\_dev {

 21        struct request\_queue \*queue;

 22        struct gendisk \*disk;

 23        sector\_t size;          /\* devicesize in Bytes \*/

 24};

 25#endif

**fbd\_driver.c**

  1/\*\*

 2  \*  fbd-driver - filter block device driver

 3  \*  Author: Talk@studio

  4\*\*/

  5#include "fbd\_driver.h"

  6

  7static int fbd\_driver\_major = 0;

  8

  9static struct fbd\_dev fbd\_dev1 = {NULL};

 10static struct fbd\_dev fbd\_dev2 = {NULL};

 11

 12static int fbddev\_open(struct inode \*inode, struct file \*file);

 13 staticint fbddev\_close(struct inode \*inode, struct file \*file);

 14

 15static struct block\_device\_operations disk\_fops = {

 16        .open = fbddev\_open,

 17        .release = fbddev\_close,

 18        .owner = THIS\_MODULE,

 19};

 20

 21static int fbddev\_open(struct inode \*inode, struct file \*file)

 22 {

 23        printk("device is opened by:[%s]\n", current->comm);

 24        return 0;

 25 }

 26

 27static int fbddev\_close(struct inode \*inode, struct file \*file)

 28 {

 29        printk("device is closed by:[%s]\n", current->comm);

 30        return 0;

 31 }

 32

 33static int make\_request(struct request\_queue \*q, struct bio \*bio)

 34 {

 35        struct fbd\_dev \*dev = (struct fbd\_dev \*)q->queuedata;

 36        printk("device [%s] recevied [%s] io request, "

 37                 "access on dev sector[%llu], length is [%u] sectors.\n",

 38                 dev->disk->disk\_name,

 39                 bio\_data\_dir(bio) == READ ?"read" : "write",

 40                 bio->bi\_sector,

 41                bio\_sectors(bio));

 42

 43        bio\_endio(bio, bio->bi\_size, 0);

 44        return 0;

 45 }

 46

 47static int dev\_create(struct fbd\_dev \*dev, char \*dev\_name, int major, intmi    nor)

 48 {

 49        int ret = 0;

 50

 51        /\* init fbd\_dev \*/

 52        dev->size = DEV\_SIZE;

 53         dev->disk = alloc\_disk(1);

 54        if (!dev->disk) {

 55                 printk("alloc diskerror");

 56                 ret = -ENOMEM;

 57                 goto err\_out1;

 58        }

 59

 60         dev->queue = blk\_alloc\_queue(GFP\_KERNEL);

 61        if (!dev->queue) {

 62                 printk("alloc queueerror");

 63                 ret = -ENOMEM;

 64                 goto err\_out2;

 65        }

 66

 67        /\* init queue \*/

 68        blk\_queue\_make\_request(dev->queue, make\_request);

 69        dev->queue->queuedata = dev;

 70

 71        /\* init gendisk \*/

 72        strncpy(dev->disk->disk\_name, dev\_name, DEV\_NAME\_LEN);

 73        dev->disk->major = major;

 74        dev->disk->first\_minor = minor;

 75        dev->disk->fops = &disk\_fops;

 76        set\_capacity(dev->disk, (dev->size >> SECTOR\_BITS));

 77

 78         /\* bind queue to disk \*/

 79         dev->disk->queue =dev->queue;

 80

 81        /\* add disk to kernel \*/

 82         add\_disk(dev->disk);

 83        return 0;

 84err\_out2:

 85        put\_disk(dev->disk);

 86err\_out1:

 87        return ret;

 88 }

 89

 90static void dev\_delete(struct fbd\_dev \*dev, char \*name)

 91 {

 92        printk("delete the device [%s]!\n", name);

 93        blk\_cleanup\_queue(dev->queue);

 94        del\_gendisk(dev->disk);

 95        put\_disk(dev->disk);

 96 }

 97

 98static int \_\_init fbd\_driver\_init(void)

 99 {

100        int ret;

101

102        /\* register fbd driver, get the driver major number\*/

103     fbd\_driver\_major =register\_blkdev(fbd\_driver\_major, DRIVER\_NAME);

104        if (fbd\_driver\_major < 0) {

105                 printk("get majorfail");

106                 ret = -EIO;

107                 goto err\_out1;

108         }

109

110        /\* create the first device \*/

111        ret = dev\_create(&fbd\_dev1, DEVICE1\_NAME, fbd\_driver\_major,DEVICE1\_MINOR);

112        if (ret) {

113                 printk("create device[%s] failed!\n", DEVICE1\_NAME);

114                 goto err\_out2;

115        }

116

117        /\* create the second device \*/

118        ret = dev\_create(&fbd\_dev2, DEVICE2\_NAME, fbd\_driver\_major,DEVICE2\_MINOR);

119        if (ret) {

120                 printk("create device[%s] failed!\n", DEVICE2\_NAME);

121                 goto err\_out3;

122        }

123        return ret;

124 err\_out3:

125        dev\_delete(&fbd\_dev1, DEVICE1\_NAME);

126 err\_out2:

127        unregister\_blkdev(fbd\_driver\_major, DRIVER\_NAME);

128 err\_out1:

129        return ret;

130 }

131

132 static void \_\_exitfbd\_driver\_exit(void)

133 {

134        /\* delete the two devices \*/

135        dev\_delete(&fbd\_dev2, DEVICE2\_NAME);

136        dev\_delete(&fbd\_dev1, DEVICE1\_NAME);

137

138        /\* unregister fbd driver \*/

139         unregister\_blkdev(fbd\_driver\_major,DRIVER\_NAME);

140        printk("block device driver exit successfuly!\n");

141 }

142

143 module\_init(fbd\_driver\_init);

144 module\_exit(fbd\_driver\_exit);

145 MODULE\_LICENSE("GPL");

**Makefile**

  1 obj-m := fbd\_driver.o

  2 KDIR := /lib/modules/$(shell uname-r)/build

  3 PWD := $(shell pwd)

  4 default:

  5        $(MAKE) -C $(KDIR) M=$(PWD) modules

  6 clean:

  7        $(MAKE) -C $(KDIR) M=$(PWD) clean

  8        rm -rf Module.markers modules.order Module.symvers

一共三个文件，fbd\_driver.c和fbd\_driver.h两个文件是源码文件，Makefile文件是我们的编译规则文件，大家可以回忆下这个Makefile文件是否与我们上册一开始写的简单内核模块中的Makefile文件非常类似，是的，内核模块的编译规则和方法就是这么简单，我们不需要在这上面的花太多的精力，会读懂和修改编译规则即可。有兴趣深入研究的同学，我们附了一个专门讲解Makefile规则语法的书，大家有选择性查阅，更多的是掌握好基本的规则和当成工具书方便查阅即可。

先看一下fbd\_driver.h头文件的内容，首先看1－2行，这个非常有意思，是一个C语言语法中的条件编译关键字，#ifndef \_FBD\_DRIVER\_H 意思就是说如果“\_FBD\_DRIVER\_H”该宏没有定义，则第2行用#define定义一下这个宏，然后再看第25行“#endif”，#ifndef 与 #endif是一对条件编译关键字语法，作为头文件中这么用的作用非常强大，它能够防止我们在.c文件中对头文件重复包含，避免代码冗余，大家体会一下这个用法。接下来3－7行共包含了5个头文件，这5个头文件是我们这个过滤块设备驱动实现需要引用的头文件，它们中有我们需要的一些函数API接口声明和数据结构的定义，其中一个我们一定不陌生就是module.h，任何一个内核模块不管它是块设备驱动，还是其它内核驱动模块，这个头文件是一定要包含的。然后bio.h/blkdev.h/genhd.h是内核块设备驱动必须要包含的三个头文件，内核的头文件命名上很有意义，基本上相关的内核API调用会放在相应的头文件中，这里我们先不具体介绍这三个头文件，在下一章节我们具体分析块设备驱动核心数据结构及API接口声明时再详细分析，我们继续往下走。第9行是定义了扇区比特数是9，对于块设备，扇区是其最小的传输和存储单位，是按扇区来划分的，默认扇区大小是512字节，这里的9代表512如果换算为二进制需要多少位描述，我们一定很快算出来就是2＾9 = 512，后面我们会经常遇到这样的二进制转换描述，在内核中是经常遇到的。第10行，我们定义的宏叫DISK\_NAME\_LEN，表示我们要写的过滤块设备的名字最大是32个字节，第11行定义了我们要创建的过滤块设备大小是512M，1左移20位是1M，再乘以扇区大小即是512M。

第13－18行定义我们定义了驱动程序注册的名字“fbd\_driver”，及通过过滤块设备驱动程序创建的过滤块设备名字叫“fbd1\_dev”和”fbd2\_dev”。

接下来20－24行，我们定义了一个数据结构结构体叫fbd\_dev，这个结构体里面有三个成员，首先是一个queue指针成员，然后是disk指针，最后是设备大小，这个结构体用于描述我们创建的过滤块设备，从我们前面列举的图书馆的例子，大家应该可以对上号了，

好了准备工作一切就绪，我们开始分析源码文件fbd\_driver.c，由于内核驱动模块的特殊性，我们向系统加载一个模块时linux内核一定会首先调用module\_init所约定的函数，注意看143行代码，module\_init是内核的一个API, 我们所写的驱动模块一定要写143／144这样两行代码，告诉内核我们的模块加载时会执行module\_init的约定函数，模块卸载时会执行module\_exit的约定函数，好的，先记住这个。然后对于我们的过滤块设备驱动来说，我们要如何设计module­­\_init约定的初始化函数呢，接下来我们介绍104行代码中的这个 fbd\_driver­­\_init函数。至于module\_init的具体实现原理我们希望大家现在暂时先沉住气，不要去分析，有时候先放一放，把精力用在最主要的事情上是非常好的一个方法。现在先来看这个加载模块时就会被执行的函数fbd\_driver\_init函数。我们再把代码贴一下：

 95static int \_\_init fbd\_driver\_init(void)

 96 {

 97        int ret;

 98

 99        /\* register fbd driver, get the driver major number\*/

100           fbd\_driver\_major =register\_blkdev(fbd\_driver\_major, DRIVER\_NAME);

101        if (fbd\_driver\_major < 0) {

102                 printk("get majorfail");

103                 ret = -EIO;

104                 goto err\_out1;

105        }

106

107        /\* create the first device \*/

108        ret = dev\_create(&fbd\_dev1, DEVICE1\_NAME, fbd\_driver\_major,DEVICE1\_    MINOR);

109        if (ret) {

110                 printk("create device[%s] failed!\n", DEVICE1\_NAME);

111                 goto err\_out2;

112        }

113

114        /\* create the second device \*/

115        ret = dev\_create(&fbd\_dev2, DEVICE2\_NAME, fbd\_driver\_major,DEVICE2\_    MINOR);

116        if (ret) {

117                 printk("create device[%s] failed!\n", DEVICE2\_NAME);

118                goto err\_out3;

119        }

120

121        return ret;

122

123 err\_out3:

124        dev\_delete(&fbd\_dev1, DEVICE1\_NAME);

125 err\_out2:

126        unregister\_blkdev(fbd\_driver\_major, DRIVER\_NAME);

127 err\_out1:

128        return ret;

129 }

首先函数是一个static函数，这个是C语言的一个基本语法，表示该函数只能在当前的文件中被调用，static后面是int 表示该函数返回值是整型，然后是\_\_init， 这个需要大家注意一下，这是gcc的一个语法，gcc是编译器，这个\_\_init就是告诉gcc在编译后，在代码运行时把这个函数的代码放在特殊的内存区域，函数执行完毕，这部分内存就会被linux内核回收，因为这个函数是模块加载时就只会调用一次的函数，后面不会再有人用这个函数了，所以执行完，就可以释放出占用的内存。

说到这希望大家还没有忘记我们开头提的4个步骤，我们再啰嗦一下块设备驱动程序需要做4件非常重要的准备工作：

1.                                                                                                注册并申请门牌号： register\_blk\_device

2.                                                                                                申请仓库：alloc\_disk

3.                                                                                                申请仓库的关卡：alloc\_queue

4.                                                                                                注册仓库的加工处理函数：blk\_queue\_make\_request

我们稍等一下，再介绍一下一个我们自己定义数据结构，也就是我们的仓库的一个描述性结构，这个结构只有我们自己写这个过滤块设备驱动的作者知道，这个结构对于linux内核是不可见的，我们内部使用而已，如下：

**fbd\_driver.h**

 20struct fbd\_dev {

 21        struct request\_queue \*queue;

 22        struct gendisk \*disk;

 23        sector\_t size;          /\* devicesize in Bytes \*/

 24};

好了，准备工作一切就绪，然后揭开块设备过滤驱动的面纱，开始分析fbd\_driver\_init函数吧，大家从此刻开始要打起二十分的精力，这是构建块设备驱动最核心的部分。

fbd\_driver\_init这个函数首先调用register\_blk\_device函数，获取到了块设备驱动程序的主设备号，register\_blkdev终于浮出水面了，还记得那4个步骤不？第一步注册并申请门牌号，对就是它，我们要向系统申请和注册，第一个参数就是一个初始化的major号，第二参数是我们块设备驱动的名字，这里我们第一参数是0, 此时系统会去它自己管理的登记情况表上看看是否有不用的号码可以我们，如果有就会我们一个，这就是regiser\_blkdev的返回值，那这个第一个参数一定要是0吗？不需要的，如果你选好自己的幸运数字了比如8,你可以把8传入这个函数，但是要小心了，系统里面如果有那位仁兄已经申请过这个8了，很不幸，就会申请失败。然后调用两次dev\_create函数创建了两个块设备，fbd\_driver­\_init函数比较简单，通过register\_blk\_device申请到门牌号（主设备号）后，我们直接跟进到dev\_create函数中分析，代码如下：

 47static int dev\_create(struct fbd\_dev \*dev, char \*dev\_name, int major, intmi    nor)

 48 {

 49        int ret = 0;

 50

 51        /\* init fbd\_dev \*/

 52        dev->size = DEV\_SIZE;

 53         dev->disk = alloc\_disk(1);

 54        if (!dev->disk) {

 55                 printk("alloc diskerror");

 56                 ret = -ENOMEM;

 57                 goto err\_out1;

 58        }

 59

 60        dev->queue = blk\_alloc\_queue(GFP\_KERNEL);

 61        if (!dev->queue) {

 62                 printk("alloc queueerror");

 63                 ret = -ENOMEM;

 64                 goto err\_out2;

 65        }

 66

 67        /\* init queue \*/

 68        blk\_queue\_make\_request(dev->queue, make\_request);

 69        dev->queue->queuedata = dev;

 70

 71        /\* init gendisk \*/

 72        strncpy(dev->disk->disk\_name, dev\_name, DEV\_NAME\_LEN);

 73        dev->disk->major = major;

 74        dev->disk->first\_minor = minor;

 75        dev->disk->fops = &disk\_fops;

 76        set\_capacity(dev->disk, (dev->size >> SECTOR\_BITS));

 77

 78         /\* bind queue to disk \*/

 79         dev->disk->queue = dev->queue;

 80

 81        /\* add disk to kernel \*/

 82        add\_disk(dev->disk);

 83        return 0;

 84err\_out2:

 85        put\_disk(dev->disk);

 86err\_out1:

 87        return ret;

 88 }

首先第49行，我们定义了一个整型变量，用于记录块设备驱动初始化过程中的返回值，再看53行到57行，接下来申请我们的仓库gendisk，我们看到是通过调用alloc\_disk这个函数，我们得到了gendisk结构，我们依然不去细说gendisk中的具体字段，在第二节详细分析。

申请完仓库，我们要建立关卡了，只有经过关卡，才能进入仓库，是的，这就是入库前的规则，当前关卡申请了，可以用也可以不用。我们的过滤块设备驱动就是这样，申请了，但是没用，但是注意一定要申请的。

接着我们看60行－65行，我们看到了申请关卡的函数blk\_alloc\_queue函数，这样我们就有申请到了一个数据结构。三个步骤我们已经走完了三个，我们都没有介绍数据结构里面的具体成员，接下来我们继续做第四个步骤，注册我们的仓库加工函数－ 请求处理函数make\_request。

我们看68行代码，我们再次贴一下：

 67         /\* init queue \*/

 68        blk\_queue\_make\_request(dev->queue, make\_request);

 69        dev->queue->queuedata = dev;

调用的函数是blk\_queue\_make\_request, 第一参数是我们刚刚申请到的请求队列，第二个参数就是我们自己写好的make\_request函数名，这个函数我们待会分析它。然后注意69行，我们做了一个赋值操作，把我们的设备描述结构绑定给了request\_queue的一个成员变量queuedata。

接下来，我们要对我们申请的仓库装饰一下，代码同样再贴一下：

 71         /\* init gendisk \*/

 72        strncpy(dev->disk->disk\_name, dev\_name, DEV\_NAME\_LEN);

 73        dev->disk->major = major;

 74        dev->disk->first\_minor = minor;

 75        dev->disk->fops = &disk\_fops;

 76        set\_capacity(dev->disk, (dev->size >> SECTOR\_BITS));

 77

 78          /\* bind queue to disk \*/

 79         dev->disk->queue = dev->queue;

 80

72行代码是给gendisk的disk\_name成员赋值，就是给我们的仓库取名字。73行代码就是把我们申请到的门牌号赋值给disk的成员major，74行我们赋值了一个次设备号，75行我们为gendisk的文件操作函数赋值了一个函数指针集结构体，这个我们在稍后分析，最后76行我们设置了设备的容量大小为512M。

79行代码就是把我们申请的queue地址保存在disk中，这样仓库和关卡就绑定在一起了，同时我们也知道了disk中有个成员叫queue, 是个指针，对吧，我们没有放弃详细说明数据结构中的成员，只不过在我们遇到的时候我们一定会予以介绍，然后在第二节详细总结分析。

好了我们已经申请注册了门牌号，申请了仓库，申请了关卡，给仓库安装了加工函数，对我们的仓库进行了装饰，就可以了吗？还差最后一个关键步骤，非常的重要，就是告诉内核我们的仓库需要审核一下，如果通过，那恭喜你，你的仓库建好了，那这个步骤就是82行代码：

 81        /\* add disk to kernel \*/

 82        add\_disk(dev->disk);

好了我们终于建好自己的仓库了，稍等，我们还有一个没有分析，就是我们仓库的加工函数make\_request，让我们赶紧看看前面图书馆例子中的请求处理函数的功能是什么，我们说过我们的过滤块设备驱动在接受到请求后不做任何处理，直接结束请求，我们看看到底是如何实现的。

 33static int make\_request(struct request\_queue \*q, struct bio \*bio)

 34 {

 35        struct fbd\_dev \*dev = (struct fbd\_dev \*)q->queuedata;

 36        printk("device [%s] recevied [%s] io request, "

 37                 "access on dev sector[%llu], length is [%u] sectors.\n",

 38                 dev->disk->disk\_name,

 39                 bio\_data\_dir(bio) == READ ?"read" : "write",

 40                 bio->bi\_sector,

 41                 bio\_sectors(bio));

 42

 43        bio\_endio(bio, bio->bi\_size, 0);

 44        return 0;

 45 }

这个函数输入参数就是我们的关卡请求队列，第二个参数就是上层准备好的盒子bio请求描述结构指针，我们的函数就调了个bio\_endio就完事了，是的，这个函数就是用于结束一个请求bio的，这样我们就知道了为什么请求到我们的仓库，就会结束，是因为我们的加工函数就是通过调用bio\_endio做到的。

至此我们终于完成了一个最简单的过滤块设备驱动的开发，赶紧试试吧，在自己的虚拟上，执行make，得到fbd\_driver.ko后，加载你的驱动insmod fbd\_driver.ko，在/dev/下面是否可以看到我们的过滤设备/dev/fbd1\_dev和/dev/fbd2\_dev，对其进行dd操作然后看dmesg信息，从dmesg命令显示的信息中我们会看到如下信息，比如执行：

[root@localhost fbd\_driver\_stage1]# ddif=/dev/zero of=/dev/fbd1\_dev bs=1M oflag=direct count=1

1+0 records in

1+0 records out

1048576 bytes(1.0 MB) copied, 0.038983 seconds, 26.9 MB/s

[root@localhost fbd\_driver\_stage1]# dmesg

device is openedby:[dd]

device[fbd1\_dev] recevied [write] io request, access on dev sector [0], length is[248] sectors.

device[fbd1\_dev] recevied [write] io request, access on dev sector [248], length is[248] sectors.

device[fbd1\_dev] recevied [write] io request, access on dev sector [496], length is[248] sectors.

device[fbd1\_dev] recevied [write] io request, access on dev sector [744], length is[248] sectors.

device[fbd1\_dev] recevied [write] io request, access on dev sector [992], length is[248] sectors.

device [fbd1\_dev]recevied [write] io request, access on dev sector [1240], length is [248]sectors.

device[fbd1\_dev] recevied [write] io request, access on dev sector [1488], length is[248] sectors.

device[fbd1\_dev] recevied [write] io request, access on dev sector [1736], length is[248] sectors.

device[fbd1\_dev] recevied [write] io request, access on dev sector [1984], length is[64] sectors.

device is closedby:[dd]

[root@localhostfbd\_driver\_stage1]#

75行我们为gendisk的文件操作函数赋值了一个函数指针集结构体，我们继续通过分析dmesg的信息完成这个解读，我们做dd写了一个1M的数据，看到了

“device is openedby:[dd]” 和 “device is closed by:[dd]” 这两行信息，这就是下面fbddev\_open和fbddev\_close函数打出的，这样我们应该能够理解了，这两个函数指针就是我们创建的块设备被打开时和关闭时会调用到，我们都写过这样的简单程序open/read/close，对吧，只不过我们跑的dd命令包含了这三个函数调用。

 15static struct block\_device\_operations disk\_fops = {

 16        .open = fbddev\_open,

 17        .release = fbddev\_close,

 18        .owner = THIS\_MODULE,

 19};

 20

 21static int fbddev\_open(struct inode \*inode, struct file \*file)

 22 {

 23        printk("device is opened by:[%s]\n", current->comm);

 24        return 0;

 25 }

 26

 27static int fbddev\_close(struct inode \*inode, struct file \*file)

 28 {

 29        printk("device is closed by:[%s]\n", current->comm);

 30        return 0;

 31 }

这就是我们这个最简单的过滤块设备驱动，由于在make\_request函数中我们直接调用bio\_endio，这个函数的作用是直接返回收到的请求，不做任何处理，这样我们写入的1M数据实际并没有处理，而是直接返回了。在第三节我们会继续完善这个170行代码的驱动，让我们的驱动能够真正的过滤请求，而不是直接退出请求处理。