

官方技术交流群QQ: 806452875

块设备驱动开发

**基于Linux 5.14内核**

白皮书

# 第一章 绪论

# 1.1 从存储技术说起

不管是在何种年代，数据的记录永远是最基础的技术也是最重要的技术之一。早在1877年，爱迪生发明了世界上第一台留声机，在今天看来，其原理非常简单，但在当时算得上是技术上的一大跨越，从此，人类有能力将声音记录下来。随着技术的不断发展，特别是半导体技术的突飞猛进，半导体存储技术成为信息存储的主流已经成为事实。

早在计算机发展初期，程序都是存储在卡带上的



图1-1 早期计算机使用纸带记录程序

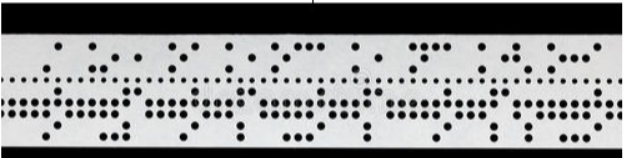


图1-2 早期计算机程序纸带

## 1.2 半导体存储技术

半导体技术的发展同时推动着存储器的发展，人类从最开始的磁带存储大容量信息，随后发展到密度更高的磁盘（软盘）存储信息。之后人类发明了巨磁电阻效应之后，硬盘（Hard Disk Drive）便开始占领计算机存储介质的垄断地位，在此期间，虽然出现过Flash存储器，但由于无法做到大容量导致其市场占用率低。随后NAND Flash的出现打破的这一局面，开始人们不太看好NAND存储器，主要原因是其寿命太短，同时其稳定性较差，但各种硬件纠错算法解决了这一问题，随后的MLC和QLC的出现使得NAND Flash的容量呈几何增长，其价格也变得与硬盘相当。从此NAND Flash开始成为主流存储介质，其代表有SSD（Solid State Disk）、eMMC、UFS、NVMe[[1]](#footnote-1)。



图1-3 存储器发展过程

### 1.2.1 机械硬盘（HDD）

也许再过十几年，人们都不曾听过这个名字，机械硬盘顾名思义即具有机械装置的硬盘，其基本原理就是利用巨磁电阻效应来实现的一种磁存储设备。磁盘是记录数据的主体，所有的数据都记录在磁盘上。读取和写入到过程正好相反，正因为由于巨磁电阻效应的发现，才使得如今的HDD容量呈几何增长。机械硬盘的结构如下图所示，主要有盘面、读写头、电源接口、数据线接口、驱动配置接口、电路控制板。

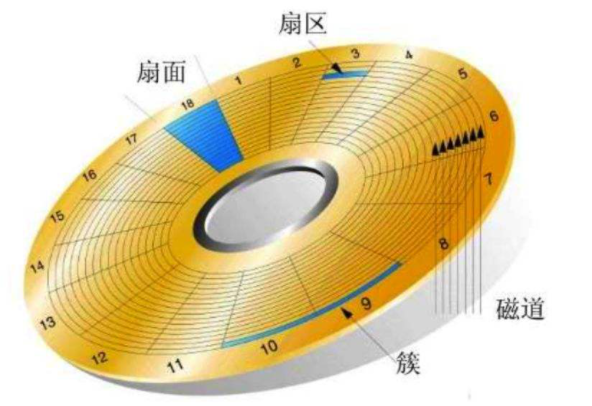
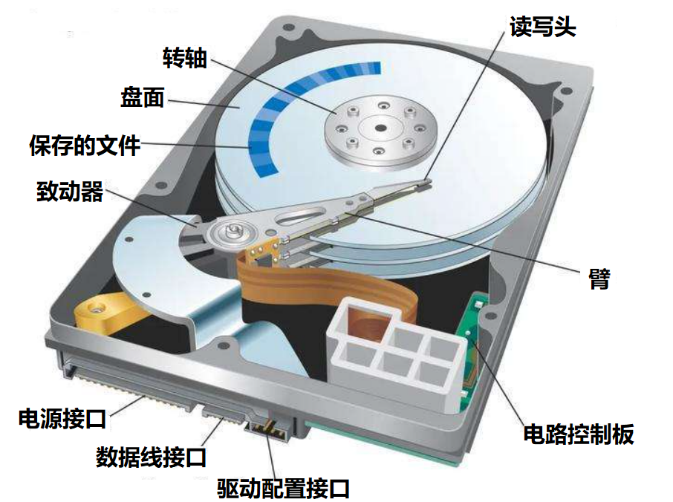


图1-4机械硬盘结构体

由于一般所有的硬盘都有控制器，在写数据时不会立刻写到硬盘上，而是先缓存在RAM中，当数据达到一定数量时才会写入到硬盘中，这样做的目的是尽可能减少磁头的读写次数，从而延长硬盘的使用寿命，因此每次写入的数据量都是一块一块的。对于HDD而言，有一个最小的写入数据量，我们称为最小写入粒度，这个大小由磁盘的扇区决定。什么是扇区呢？在HDD中，磁盘的盘面被分为一圈一圈的磁道，而每个磁道又被分割为一个一个的扇区，一个扇区就是磁盘的最小写入数据量，大部分的HDD的扇区大小一般是512Byte。那什么是蔟呢？在很多操作系统中，读写的数据量最小单位并不是一个扇区，例如DOS认为一个扇区的读写粒度太大，读写次数频繁了，因此DOS规定16KB为一个最小的读写数据量，我们将这个16KB称为一个蔟，也就是蔟的大小由操作系统决定，一般DOS分区表中为对其进行说明。那什么是柱面呢？实际的HDD一般不会是单个盘，而是由多个盘叠加在一起组合而成的，我们将不同的盘面的相同的磁道称为一个柱面。

但随着半导体技术的飞速发展，机械硬盘逐渐被固态硬盘所取代，然而很多术语却保留了下来，很多书中，NAND Flash的最小读写数据仍称为扇区，实际上对于Flash而言，根本不存在扇区，这个只是历史的术语延续下来了。

### 1.2.2 掩膜ROM

在电脑的初期，很多BIOS芯片所使用的正是掩膜ROM，那什么是掩膜ROM呢？在很早之前，还没有出现Flash，甚至连EPROM都没有出现，这个时候一些关键代码如何存储呢？存储在硬盘中？这个显然不行，因为硬盘是不可寻址的，所谓不可寻址就是说CPU中的地址和数据总线无法访问该区域，因为硬盘是需要控制器的，所有的数据都需要控制器来读写，然而控制器是需要协议通信的，因此CPU是不可能在硬盘中执行最初的代码。那么这段代码一定是存放在能够直接寻址的存储器中，早期的解决办法很简单，就是利用晶体管来实现数据存储阵列。

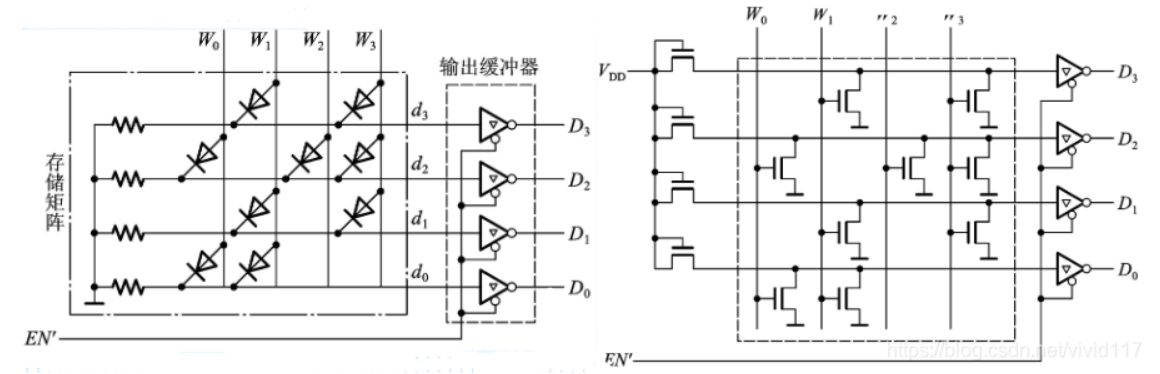


图1-5 利用晶体管实现ROM

上图左边是利用二极管来实现存储整列的，右边是利用MOS管来实现存储整列的，这些晶体管通过掩膜工艺集成在特定的芯片中，因此这种芯片中的数据是不可再修改的。

### 1.2.3 EPROM和EEPROM

掩膜ROM只能在生产的时候将预先需要写的数据直接制作出来，但在大多数时候，人们总是希望数据能够二次修改，随后人们就发明了一种能够通过紫外线来擦除ROM中的数据，这也就是EPROM[[2]](#footnote-2)。



图1-6 EPROM存储器

在这种EPROM的原理是基于浮栅晶体管的特性，这总行浮栅晶体管是在MOS管的基础上改进而来。在MOS的中间层添加一个浮空的可以存储电荷的栅极，这个栅极是悬浮在中间的，因此称为浮栅。在浮栅上面会有一个透明电极，当需要写入数据时，只需要在透明电极上施加高压，这样由于隧穿效应，衬底中的电子会进入浮栅中。即使此时透明电极的电压消失，此时浮删中的电子仍然存在，只要源极和漏极存在电势差，那么由于浮栅中存在大量电子，此沟道会积累很多的正电荷，这样就形成了导电沟道。当浮栅被强烈的紫外线照射时，此时浮栅中的电子就会流入源极或者漏极中（电子吸收紫外线后，变得非常活跃，隧穿效应加剧）。 这样就实现了擦除。

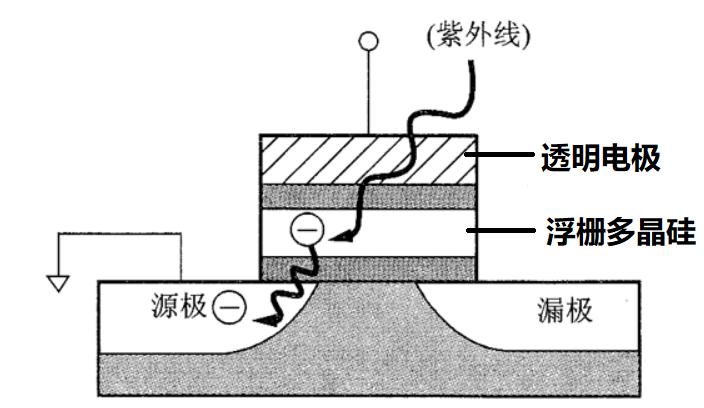


图1-7 EPROM原理

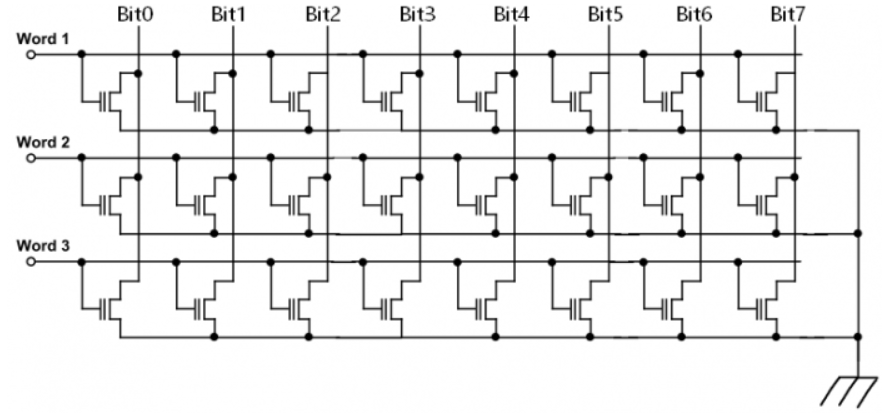
然而每次利用紫外线擦除非常不方便，科学家又再此基础上进行了改进，发明了EEPROM[[3]](#footnote-3)，即电可擦除只读存储器。顾名思义，该存储器可以利用电来擦除，不需要强烈的紫外线了。其基本原理和EEPROM很相似，将浮栅中的电子消除，我们需要用紫外线，能否有其他办法呢？答案是有。不就是将浮栅中的电子拉出来嘛，我们施加反向的电场不就行了么，没错，在透明电极和存底之间施加强电场，就可以将浮栅中的电子拉出来。这样EEPROM的透明电极也就不用做成透明的了，可以直接密封在芯片内部了。

### 1.2.4 Flash存储器

虽然EEPROM可以实现电可擦除了，但是其速度比较慢，为了能够加快EEPROM的读写速度，人们发明了Flash存储器。我们的目的就是希望加快EEPROM，因此这就需要缩短浮栅中的电子的写入和拉出的时间。最简单的办法便是减小浮栅的厚度以及浮栅到沟道的厚度，但是减小厚度这样就造成了寿命短不稳定问题。方法总比问题多，实际上写入和擦除方式有很多种，EEPROM的擦除方式是利用隧穿效应，这种方式速度一般在ms级别。而还有一种方式是us级别的，这就是热电子注入方式。这里我们不去深究这种方式，我们直需要直到这种方式速度比隧穿效应快，Flash正是利用这种方式来实现编程。

但是EEPROM的擦除电路较为复杂，如果Flash仍然采用这种擦除的电路，那么面积会占用很大，为此，Flash简化了擦除电路，只能支持一个块为最小单位擦除，这样电路就可以做的比较简单，集成度就可以增加。 同时Flash的擦除方式为热电子注入方式，这种方式有个缺陷，即只能写入0，不能写入1，因此Flash在写入数据之前必须对所在的块进行擦除（全部写入1）[[4]](#footnote-4)，然后才能下入数据。因此Flash的容量一般比EEPROM要大得多。

半导体存储中最常见的存储器便是NOR Flash（NOT OR，即或非）和NAND Flash（NOT AND，即与非）。其结构如下图所示：



# 第二章 块设备驱动

为了规范块设备驱动，Linux内核对存储设备进行了合理的管理，Linux下的所有存储设备都被抽象为块设备，最常见的便是我们的文件读写。下面我们来简单认识下并讲解块设备的基本框架。

## 2.1 文件系统

什么是文件系统？所谓文件系统就是将文件以规定的格式存放在硬盘上，这个存储格式就是文件系统。如果我们的数据不以规定的格式存放，那么我们再去读取数据时就无法知道数据是啥了，而是一堆二进制数据，因此所有的数据必须按规定的格式存放，这就出现了文件系统。我们大多数人接触的最常见的文件系统便是FAT32文件系统，该文件系统是由DOS文件系统发展而来，我们都知道，DOS文件系统是DOS操作系统中的文件系统。在字符设备驱动中，我们讲过FAT32的分区表，该文件系统在Windows中非常常见，其结构如下图所示：

表格

描述已自动生成

图2-1 FAT32文件系统结构

## 2.2 虚拟文件系统

在字符设备驱动中提到过虚拟文件系统，我们都知道根文件系统是Linux挂载的第一个文件系统，然而实际上虚拟文件系统（Virtual File System）才是Linux挂载的第一个文件系统。为何会存在虚拟文件系统呢？这个就是Linux的强大之处，由于Linux操作系统可以支持非常多的文件系统，例如ext、ext2、ext4、minix、umsdos，msdes、fat32 、ntfs、proc、stub、ncp、hpfs、affs、sysfs等等。这些文件系统都有各自的格式，为了将这些文件系统统一起来，Linux内核实现了一种中间层的文件系统接口，这就是虚拟文件系统。虚拟文件系统让驱动开发者不需要去关心此时使用的是何种文件系统，只需要调用统一的文件操作接口即可。

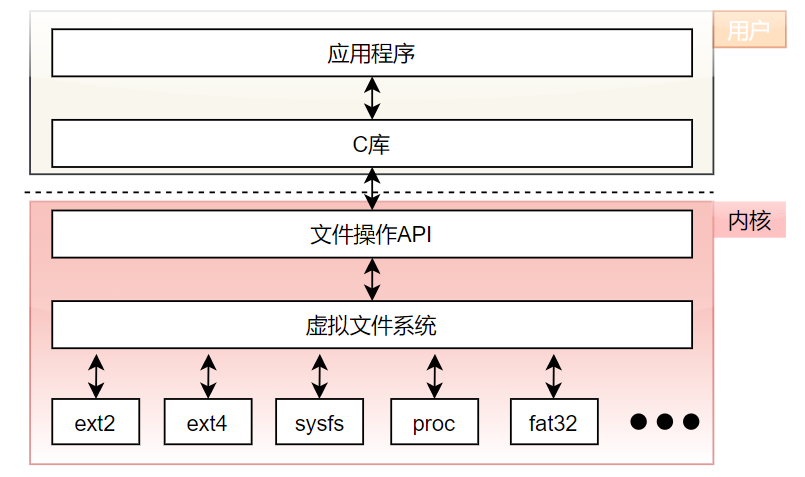


图2-2 虚拟文件系统

从上图可以看到，虚拟文件系统并不是真正的文件系统，应用程序通过系统调用之后会进入内核态，C库操作的是虚拟文件系统，而虚拟文件系统需要经过真实的文件系统才能对实际的存储介质进行读写。

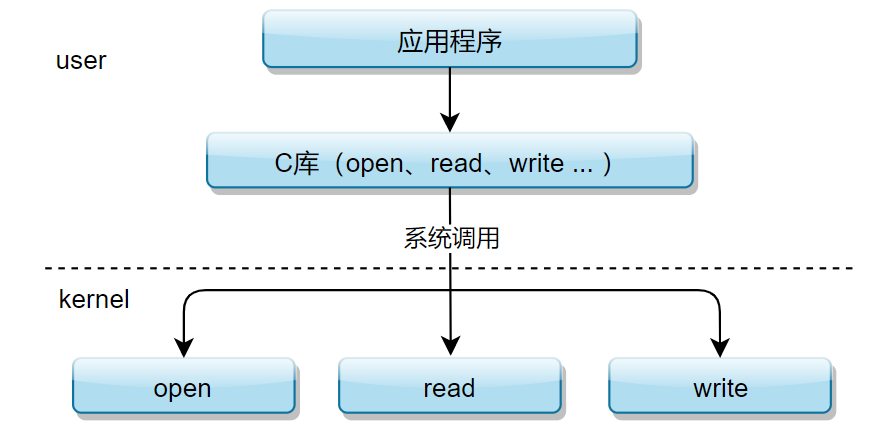


图2-3 C库通过系统调用来实现调用内核中的库函数接口

上图给出了系统调用的关系图，可以看到系统应用程序调用C库来编写各个的任务，而C库则通过系统调用来实现对内核中的接口访问，那为何要存在C库呢？应用程序直接系统调用岂不更快？在实际的过程中，每个应用程序的开发都是完全不同的，如果每个人都去直接操作系统调用，那么内核将需要提供无穷多个接口来满足不同应用场景，显然这是不现实的。这也是POSIX规范诞生的主要原因。为了让应用程序调用统一的接口，C库就诞生了，C库封装了基本的文件读写操作函数，同时还封装了大量的通用接口，这样内核只要能够为这些接口提供对应的操作即可。可以看到，C库实际上是应用程序调用的接口，当然现在几乎所有的操作系统都集成了C库。

## 2.3 块设备驱动框架

由于操作系统和块设备紧密相关，因此Linux内核对块设备做了非常多的工作，例如对块设备进行了分层，对上层提供统一的块设备读写接口，对下层提供标准的实现对硬件的读写接口，当然中间还有各个不同设备的协议层，这些标准的协议被封装为不同的子系统，例如我们常见的SCSI、MMC、MTD等，我们先有一个简单的认识，后面再深入理解。

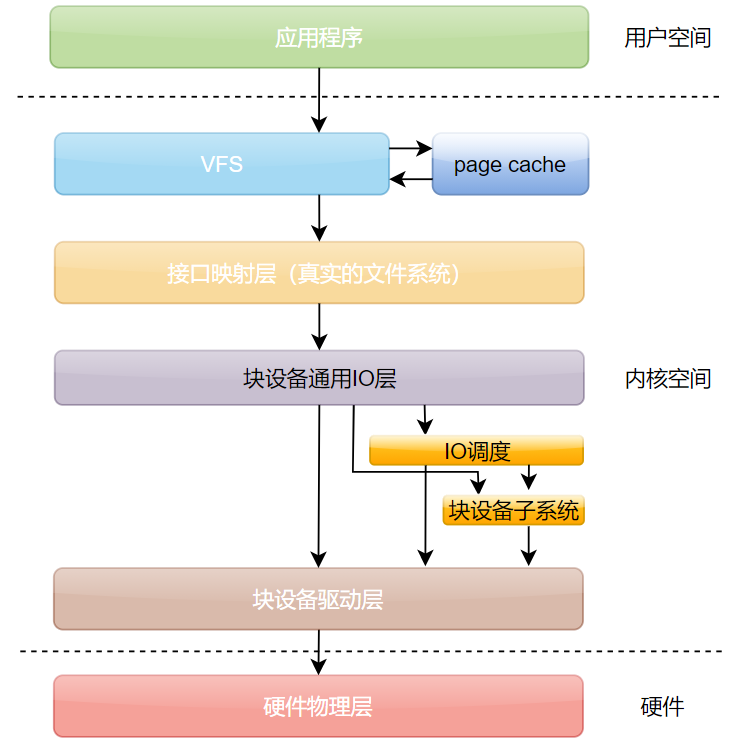


图2-4 Linux中块设备驱动结构

可以看到，就驱动开发而言，我们并不需要过多的了解上层的机制，我们只需要去实现块设备驱动层的接口，这些接口是需要实际操作各个硬件的。Linux内核的这种层次化设备很大程度上降低了驱动开发者的难度，同时也提高了代码的可重用性。

对于机械硬盘、光盘等设备，块设备驱动会走IO调度，这是因为此类设备的读写都是机械的，顺序读写的性能较好，而随机读写的性能非常差，因此，为了提高读写性能，我们可以将多个读写命令先按读写的位置顺序排列后，再发送给磁盘控制器，这样就可以减少随机读写的概率。

而对于SSD、UFS、NVMe、MMC等Flash存储设备，由于其结构并不是机械结构，因此可以实现快读的随机读写（较机械硬盘而言）。我们的IO调度中的排序对读写性能并没有改善，相反由于引入了一个IO调度层导致其路径加长反而下降，因此对于此类块设备，Linux中没有走IO调度层，而是直接都块设备驱动层。

page cache的作用是能够缓存一部分的数据，当VFS读取数据时，其最先会查找page cache中有无数据，如果没有数据才会去往下层走，这样做的目的是减少磁盘的读写次数。

在Linux中，所有的块设备读写都被抽象为一个request，该结构中包含了最重要的bio结构体，这个结构体描述了读写的所有信息。如下图所示，在字符设备中，我们的ops包含了设备的open、release、read、write等，其中read、write对设备进行读写操作。在最新的Linux内核中，其读写被抽象为submit\_bio操作，该操作来完成硬盘的读写。为何要引入这个呢？这个是由于历史原因，在最较老版本的内核中，使用的是make\_request\_fn函数来实现读写，该部分是独立出去的。

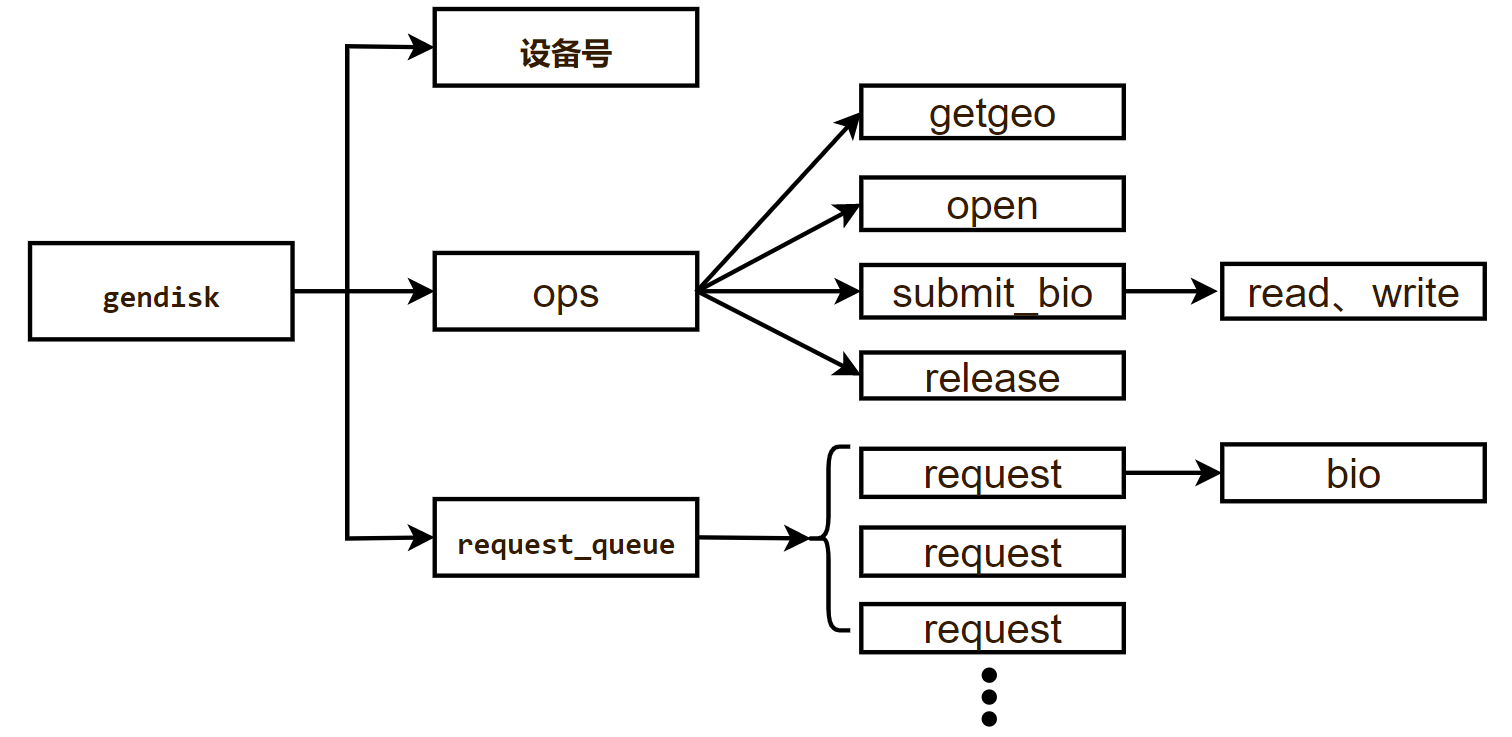


图2-5 块设备驱动结构体关系图

下面我们将详细对各个结构体进行介绍，同时通过一个简单的块设备驱动来讲解简单的块设备驱动开发流程。

## 2.4 块设备相关结构体

上面我们简单描述了下块设备驱动的整体结构，下面我们将对块设备驱动中涉及到了结构体进行说明。对于机械硬盘的块驱动而言，IO层发送的request都会被放在IO调度层中进行排序，完毕之后再发送给硬盘驱动程序来将数据封装成命令对磁盘设备进行读写。而对于固态硬盘而言，其流程中就不需要进行IO调度，而是直接发送命令，当然，在Linux中也有很多子系统，大部分时候我们的块设备驱动都会经过子系统来实现。首先我们先来了解下块设备驱动相关的基本结构体，然后在利用两个简单的块设备驱动说明其驱动的编写步骤。

### 2.4.1 gendisk

该结构体类似于字符设备驱动中的cdev结构体，其对所有的块设备进行了抽象，具体如下所示：

struct gendisk {

    /\* major, first\_minor and minors are input parameters only,

     \* don't use directly.  Use disk\_devt() and disk\_max\_parts().

     \*/

    int major;          /\* major number of driver \*/

    int first\_minor;

    int minors;                     /\* maximum number of minors, =1 for

                                         \* disks that can't be partitioned. \*/

    char disk\_name[DISK\_NAME\_LEN];  /\* name of major driver \*/

    unsigned short events;      /\* supported events \*/

    unsigned short event\_flags; /\* flags related to event processing \*/

    struct xarray part\_tbl;

    struct block\_device \*part0;

    const struct block\_device\_operations \*fops;

    struct request\_queue \*queue;

    void \*private\_data;

    int flags;

    unsigned long state;

#define GD\_NEED\_PART\_SCAN       0

#define GD\_READ\_ONLY            1

#define GD\_QUEUE\_REF            2

    struct mutex open\_mutex;    /\* open/close mutex \*/

    unsigned open\_partitions;   /\* number of open partitions \*/

    struct kobject \*slave\_dir;

    struct timer\_rand\_state \*random;

    atomic\_t sync\_io;       /\* RAID \*/

    struct disk\_events \*ev;

#ifdef  CONFIG\_BLK\_DEV\_INTEGRITY

    struct kobject integrity\_kobj;

#endif  /\* CONFIG\_BLK\_DEV\_INTEGRITY \*/

#if IS\_ENABLED(CONFIG\_CDROM)

    struct cdrom\_device\_info \*cdi;

#endif

    int node\_id;

    struct badblocks \*bb;

    struct lockdep\_map lockdep\_map;

};

可以看到，块设备的结构体比cdev结构体复杂的多，其主要成员如下：

* major：主设备号
* first\_minor：次设备号，对于多个分区的设备，每个分区都应该有一个次设备号
* minors：次设备号个数（分区个数），只有一个分区而言，此值为1
* disk\_name：设备名称
* fops：操作集合
* queue：请求队列，该指针指向磁盘的读写操作

主设备号和次设备号与字符设备类似；这里的fops后面会讲到，该指针为块设备的一些操作集合，例如open、release、ioctl等，但是该操作集合中没有write和read，对于块设备驱动而言，其读写操作单独用request\_queue来表示；queue便是指向request\_queue的指针。

### 2.4.2 block\_device\_operations

该结构体便是gendisk结构体中的fops，从结构体的名字上看，和file\_operations，block也是需要对实际的块设备进行操作，因此需要提供必要的接口供上层调用。此结构体如下：

struct block\_device\_operations {

    blk\_qc\_t (\*submit\_bio) (struct bio \*bio);

    int (\*open) (struct block\_device \*, fmode\_t);

    void (\*release) (struct gendisk \*, fmode\_t);

    int (\*rw\_page)(struct block\_device \*, sector\_t, struct page \*, unsigned int);

    int (\*ioctl) (struct block\_device \*, fmode\_t, unsigned, unsigned long);

    int (\*compat\_ioctl) (struct block\_device \*, fmode\_t, unsigned, unsigned long);

    unsigned int (\*check\_events) (struct gendisk \*disk,unsigned int clearing);

    void (\*unlock\_native\_capacity) (struct gendisk \*);

    int (\*getgeo)(struct block\_device \*, struct hd\_geometry \*);

int (\*set\_read\_only)(struct block\_device \*bdev, bool ro);

    /\* this callback is with swap\_lock and sometimes page table lock held \*/

    void (\*swap\_slot\_free\_notify) (struct block\_device \*, unsigned long);

    int (\*report\_zones)(struct gendisk \*, sector\_t sector,

            unsigned int nr\_zones, report\_zones\_cb cb, void \*data);

    char \*(\*devnode)(struct gendisk \*disk, umode\_t \*mode);

    struct module \*owner;

    const struct pr\_ops \*pr\_ops;

};

我们看几个较为重要的成员：

* submit\_bio：该成员是块设备通用IO层之上用来将bio提交到通用IO层，如下图所示，接口映射层调用submit\_bio来将bio请求转发给 generic block layer层。



图2-5 submit\_io转发bio

* open：和file\_operations中的open一样，实现设备的打开。
* release：和file\_operations中的release一样，实现设备的关闭。
* ioctl：和file\_operations中的ioctl[[5]](#footnote-5)一样，实现设备的命令操作，大部分命令内核已经实现，驱动开发人员只需要实现一小部分的命令即可。
* check\_events：检测插拔事件。
* getgeo：获取磁盘属性，例如柱头（磁头）、柱面、扇区大小等。
* owner：一般初始化为THIS\_MODULE

### 2.4.3 bio

该结构体目的是对块设备的属性进行描述，例如一个机械硬盘，其属性有：存储容量、磁盘数量、柱面、柱头、扇区大小、磁道数量等。其具体结构体如下所示：

struct bio {

    struct bio      \*bi\_next;   /\* request queue link \*/

    struct block\_device \*bi\_bdev;

    unsigned int        bi\_opf; /\* bottom bits req flags,\* top bits REQ\_OP.Use

                          \* accessors.\*/

    unsigned short      bi\_flags;  /\* BIO\_\* below \*/

    unsigned short      bi\_ioprio;

    unsigned short      bi\_write\_hint;

    blk\_status\_t        bi\_status;

    atomic\_t        \_\_bi\_remaining;

    struct bvec\_iter    bi\_iter;

    bio\_end\_io\_t        \*bi\_end\_io;

    void            \*bi\_private;

#ifdef CONFIG\_BLK\_CGROUP

    /\*

     \* Represents the association of the css and request\_queue for the bio.

     \* If a bio goes direct to device, it will not have a blkg as it will

     \* not have a request\_queue associated with it.  The reference is put

     \* on release of the bio.

     \*/

    struct blkcg\_gq     \*bi\_blkg;

    struct bio\_issue    bi\_issue;

#ifdef CONFIG\_BLK\_CGROUP\_IOCOST

    u64         bi\_iocost\_cost;

#endif

#endif

#ifdef CONFIG\_BLK\_INLINE\_ENCRYPTION

    struct bio\_crypt\_ctx    \*bi\_crypt\_context;

#endif

    union {

#if defined(CONFIG\_BLK\_DEV\_INTEGRITY)

        struct bio\_integrity\_payload \*bi\_integrity; /\* data integrity \*/

#endif

    };

    unsigned short      bi\_vcnt;    /\* how many bio\_vec's \*/

    /\*

     \* Everything starting with bi\_max\_vecs will be preserved by bio\_reset()

     \*/

    unsigned short      bi\_max\_vecs;    /\* max bvl\_vecs we can hold \*/

    atomic\_t        \_\_bi\_cnt;   /\* pin count \*/

    struct bio\_vec      \*bi\_io\_vec; /\* the actual vec list \*/

    struct bio\_set      \*bi\_pool;

    /\*

     \* We can inline a number of vecs at the end of the bio, to avoid

     \* double allocations for a small number of bio\_vecs. This member

     \* MUST obviously be kept at the very end of the bio.

     \*/

    struct bio\_vec      bi\_inline\_vecs[];

};

我们来看些这个结构体中比较重要的成员变量，如下：

* bi\_next：指向写一个struct bio结构体。
* bi\_bdev：记录该bio所关联的块设备。
* bi\_opf：操作标志，末尾bit表示读写标志，开始bit表示优先级标志。
* bi\_iter：bio迭代器，保存着要写入或者读取的信息，例如扇区地址、数据长度等。
* bi\_private：指向bio私有数据
* bi\_vcnt：bio容器的数量
* bi\_io\_vec：bio容器，记录着要写入的数据的页的物理地址、长度、偏移地址等。
* bi\_inline\_vecs：访问该结构体成员的零长度数组。

注：bi\_inline\_vecs是一个0长度的数组变量，该变量不占用内存空间，这个是gnu C的语法特性，标准的C不支持该特性，在Linux中用的比较多。例如，如果我们需要记录一个字符串，但该字符串没有指定长度，那么我们也许会这样写：

struct string

{

int length;

char data[100];

}

如果我们的字符串小于100，那势必会浪费空间，因此我们可以这样定义：

struct string

{

int length;

char \*data;

}

这样我们使用动态分配内存来实现，但是这样data就会占用4个字节的空间，但如果使用零长度数组，这样就不会浪费这个4个字节了，如下：

struct string

{

int length;

char data[0];

}

为了更加清除的看到bio的结构，笔者将该结构绘制成图的形式，如下所示：

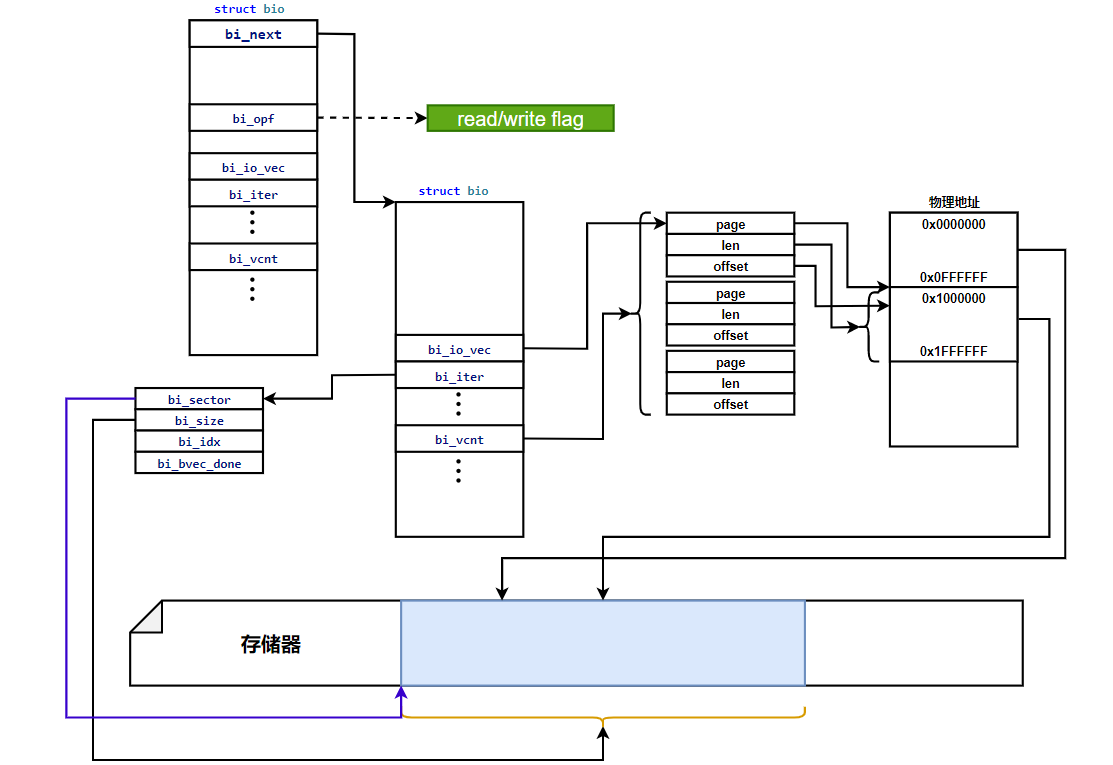


图2-6 bio结构图

### 2.4.5 request\_queue

该结构体较为庞大，这里省略其他不相关的成员，其具体结构如下所示：

struct request\_queue {

    /\* 省略无关成员 \*/

struct request       \*last\_merge;

struct elevator\_queue   \*elevator;

    struct percpu\_ref   q\_usage\_counter;

    const struct blk\_mq\_ops \*mq\_ops;

    unsigned long       nr\_requests;    /\* Max # of requests \*/

    struct blk\_flush\_queue  \*fq;

    struct list\_head    requeue\_list;

    spinlock\_t      requeue\_lock;

    /\* 省略无关成员 \*/

};

其中主要的成员说明如下：

* last\_merge：指向最先可能合并的queue
* elevator：指向电梯算法
* q\_usage\_counter：记录queue的使用情况
* mq\_ops：request的底层的操作方法
* nr\_requests：queue中request的最大个数
* fq：指向flush操作，即强制将缓存数据写入到硬盘中
* requeue\_list：指向request的链表头
* requeue\_lock：指向request的自旋锁

### 2.4.4 request

该结构体是块设备驱动的读写操作的抽象集合，其中包含了bio结构体，块设备驱动中的所有读写都有request来完成，可以看出该结构体的重要性。下面是该结构体的具体成员：

struct request {

/\* 省略无关成员 \*/

    struct request\_queue \*q;

    /\* the following two fields are internal, NEVER access directly \*/

    unsigned int \_\_data\_len;    /\* total data len \*/

    sector\_t \_\_sector;      /\* sector cursor \*/

    struct bio \*bio;

    struct bio \*biotail;

    struct list\_head queuelist;

    struct gendisk \*rq\_disk;

struct block\_device \*part;

/\* 省略无关成员 \*/

};

其成员说明如下：

* q：指向该request的request\_queue
* \_\_data\_len：记录整个数据的长度
* \_\_sector：记录当前扇区的位置
* bio：指向bio结构体
* biotail：指向bio结构体的
* queuelist：指向下一个request，该结构是一个链表结构
* rq\_disk：指向gendisk，即要操作的磁盘设备
* part：指向分区位置

## 2.5 请求队列的结构图

上面我们对块设备驱动中涉及到的结构体进行了简单讲解，但这些结构体都比较复杂，下面我们对这个结构体联系起来，其整体结构图如下所示：

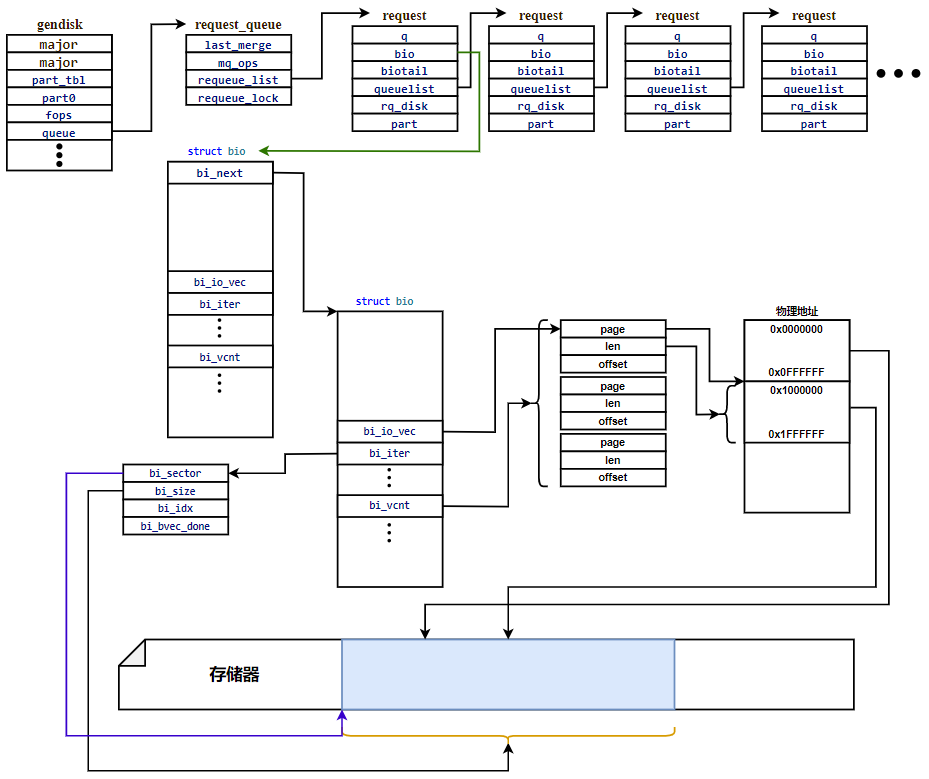


图2-7 使用请求队列的块设备驱动结构图

上面的结构中会涉及到三个概念，即扇区(sector)、块(Block)、段(Segment)，其中扇区就是在第一章节中提到的，大部分块设备的一个扇区规定为512Byte，块由多个扇区组成，而段由多个块组成，其结构如下图所示。对于大部分块设备而言，一个page的大小为4Kbyte，一个segment的大小为2Kbyte，一个block的大小为1Kbyte，一个sector的大小为512Byte。在Linux中，划分为Segment的主要原因是DMA的通道数据宽度大小与段的大小相等，这样传输数据就可以利用DMA来实现数据的读写，而不用CPU参与。在Linux中，page的大小由一个宏来表示，这个宏就是PAGE\_SIZE，该值一般为4KByte。需要注意的是，这里的page是一个虚拟的概念，与实际的介质没有关系。

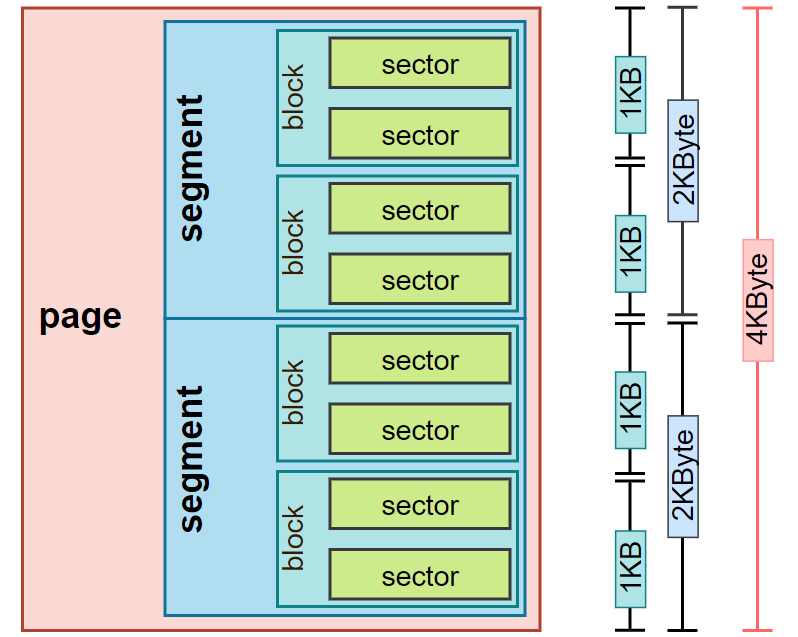


图2-8 page、segment、block、sector大小说明

## 2.5 无请求队列的结构图（续）

读者需要注意的是，由于引入请求队列的目的是使用特定算法能够将多个request合并，这样对于多个离散的地址读写时，可以将这些离散的地址按地址的大小顺序重新排列，这样的好处就是对于机械硬盘来说，其读写速度将有非常大的提高。然而技术的发展，当前固态硬盘已经开始逐渐取代机械硬盘了，这样引入请求队列的机制将对硬盘的读写没有任何改善，反而由于引入请求导致其路径变长而速度下降。因此在Linux中，也同样支持没有请求队列的驱动框架。结构图如下所示，可以看到，在不使用请求队列时，此时gendisk中的fops中的submit\_bio函数直接将bio进行处理，并没有将bio用queue保存起来。

这种没有使用request queue的块设备驱动通常在Flash存储器中用的非常多，例如SSD、U盘、UFS、NVMe等。当不适用请求队列时，我们就不需要分配request queue了，当然也就无需去实现请求处理了，我们只需要实现submit\_bio函数指针的实例。

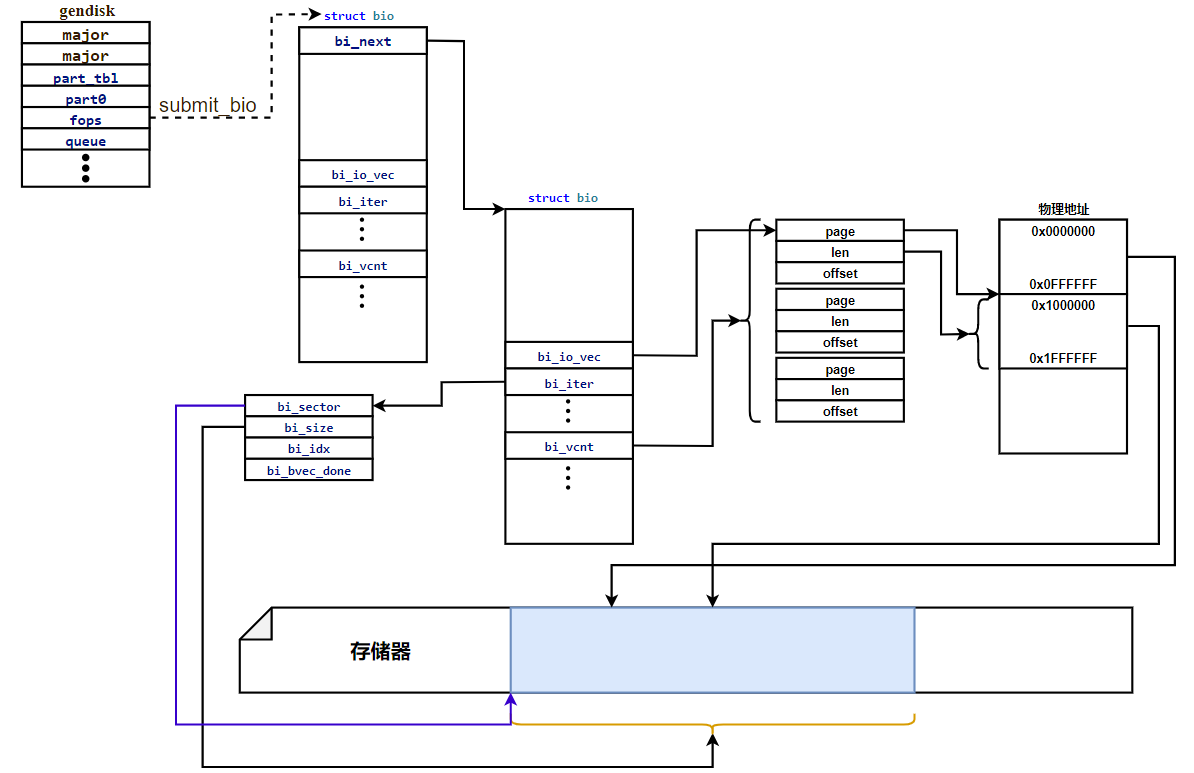


图2-8 不使用请求队列的块设备驱动结构图

## 2.6 块设备驱动注册

上面我们对各个重要的结构体进行了简单描述，现在我们对块设备驱动的注册进行说明，即如何在Linux中注册一个块设备驱动。创建一个块设备驱动的步骤如下：

1. 创建一个块设备
2. 分配一个申请队列
3. 分配一个gendisk结构体
4. 设置gendisk结构体的成员
5. 注册gendisk结构体

下面我们首先对块设备驱动中用到的常用API进行介绍。

### 2.6.1 块设备API

#### 2.6.1.1分配一个块设备结构体

在Linux中，结构体可以用定义一个结构体，由编译器来分配内存，还有一种就是使用Linux提供的动态分配方式。但对于块设备驱动而言，Linux中只能用动态方式分配，下面是Linux内核提供了块设备结构体的分配函数：

**struct gendisk \*alloc\_disk(int minors)**

其中minors为分区个数，如果不分区，则填写1即可。

#### 2.6.1.2 删除一个块设备结构体

当我们的块设备卸载时，我们需要删除该块设备结构体，即释放内存资源，Linux内核提供了如下API供驱动开发者使用：

**void del\_gendisk(struct gendisk \*gp)**

该函数的参数为gendisk，即需要删除的块设备结构体。

#### 2.6.1.3 注册块设备驱动

**int register\_blkdev(unsigned int major, const char \*name)**

该函数实现对一个块设备的注册，其中第一个参数major为主设备号，第二个参数name为设备的名称。如果major为0，则表示动态分配一个主设备号。注册成功返回主设备号值，否则返回负数。

#### 2.6.1.4 注销块设备驱动

**void unregister\_blkdev(unsigned int major, const char \*name)**

该函数实现对一个块设备进行注销，其中第一个参数major为主设备号，第二个参数name为设备名称。

#### 2.6.1.5 将gendisk添加到内核

**inline void add\_disk(struct gendisk \*disk)**

该函数将一个gendisk添加到内核中，其中参数为gandisk结构体。需要注意的是，在添加gendisk之前必须对gendisk结构体中的成员进行初始化。

#### 2.6.1.6 将gendisk从内核中删除

**void put\_disk(struct gendisk \*disk)**

该函数将一个gendisk从内核中删除，其中参数为gandisk结构体。

#### 2.6.1.7 设置gendisk的扇区数

**void set\_capacity(struct gendisk \*disk, sector\_t sectors)**

该函数用来设置gendisk的扇区数，即硬盘容量，在分配gendisk之后，我们需要用此函数来设置gendisk的扇区数。

#### 2.6.1.8 分配一个请求队列[[6]](#footnote-6)

**struct request\_queue \*blk\_mq\_init\_queue(struct blk\_mq\_tag\_set \*set)**

该函数用来申请一个请求队列同时初始化，即request\_queue，该请求队列用来容纳request。其参数为blk\_mq\_tag\_set即用来存储器件相关的tag集合。

#### 2.6.1.9 清除请求队列中的所有请求

**void blk\_cleanup\_queue(struct request\_queue \*q)**

该函数用于清除request\_queue中的所有request，当gendisk释放后，内核中可能还存在没有处理完的请求，因此我们在删除块设备之前，必须保证内核中没有请求了。

## 2.7 一个简单的示例

上面我们讲解了块设备驱动的基本框架和步骤，现在我们以一个虚拟的块设备驱动来具体说明开发过程。

### 2.7.1 不使用请求队列

由于现在大部分硬盘都是固态硬盘，因此这里我们先以一个不使用请求队列的驱动来讲解。首先是块设备驱动的分配和注册，然后最重要的是实现submit\_bio函数指针的实例。为了方便我们讲解块设备驱动，这里以RAM作为存储介质，即使用内存来实现块设备的数据存储，这样读写操作就变得比较方便。

和字符设备驱动一样，我们这里为了让结构变得简单，使用最简单的驱动框架，而不使用platform框架，实际开发的时候最好使用platform驱动框架。在\_\_init函数需要实现块设备驱动的注册，其中包括块设备的设备号分配、设备注册、设置扇区大小等。代码如下所示：

static int \_\_init ram\_blk\_init(void)

{

    /\* 注册一个块设备，返回值为主设备号 \*/

    ram\_blk\_major = register\_blkdev(0, "ram\_blk")

    if ( ram\_blk\_major < 0) {

        printk(KERN\_ERR"ram disk register failed!\n");

        return -1;

    }

    ram\_gendisk = alloc\_disk(1);//分配一个gendisk

    ram\_gendisk->major = ram\_blk\_major;//设置主设备号

    ram\_gendisk->first\_minor = 0;//设置第一个分区的次设备号

    ram\_gendisk->minors = 1;//设置分区个数：1

    ram\_gendisk->fops = &ram\_blk\_ops;//指定块设备ops集合

    set\_capacity(ram\_gendisk, 2048);//设置扇区数量：1MiB/512B=2048

add\_disk(ram\_gendisk);//添加硬盘

spin\_lock\_init(&ram\_blk\_lock);//初始化自旋锁

    ram\_blk\_addr = (char\*)vmalloc(2048\*512);//分配1M空间作为硬盘

    if(ram\_blk\_addr == NULL)

    {

        printk(KERN\_ERR"alloc memory failed!\n");

}

    return 0；

}

上面的函数较为简单，需要注意的是设置硬盘扇区数的时候我们以512Byte作为一个扇区，这样我们的1MiB的硬盘大小就有2048个扇区了。

static const struct block\_device\_operations ram\_blk\_ops = {

    .owner      = THIS\_MODULE,

    .submit\_bio = ram\_blk\_submit\_bio,

    .open       = ram\_blk\_open,

    .release    = ram\_blk\_release,

};

我们再来上面函数中的ram\_blk\_ops，这个是块设备的操作集合，其中包含了open、release、submit\_bio等，open和release函数如下所示：

static int ram\_blk\_open(struct block\_device \*bdev, fmode\_t mode)

{

    return 0;

}

static void ram\_blk\_release(struct gendisk \*disk, fmode\_t mode)

{

}

可以看到，上面的这两个函数啥也没干，因为我们使用的是RAM来模拟硬盘的。我们再来看下ram\_blk\_submit\_bio这个函数，这个函数实现了数据的读写，因此较为关键，我们这里对这个函数详细讲解。

该函数是块设备驱动最终要执行的读写操作，首先应用层通过读写文件来实现对介质的读写，这个过程需要调用C库，然后C库会通过系统调用来进入实际的读写操作，之后会进入虚拟文件系统，然后映射到具体的文件系统上，这时内核的块设备框架会调用submit函数来实现数据的读写，实际上这个过程会有缓存，也就是数据不会立刻写入到块设备驱动，而是先放在缓存中，也就是内存中，当数据满足一定数量后，块设备驱动就会触发submit来提交bio。

static blk\_qc\_t ram\_blk\_submit\_bio(struct bio \*bio)

{

    struct bio\_vec bvec;

    struct bvec\_iter iter;

    sector\_t sector = bio->bi\_iter.bi\_sector;

    bio\_for\_each\_segment(bvec, bio, iter) {

        char \*buffer = kmap\_atomic(bvec.bv\_page) + bvec.bv\_offset;

        unsigned len = bvec.bv\_len >> SECTOR\_SHIFT;

        ram\_blk\_transfer(sector, len, buffer,

                bio\_data\_dir(bio) == WRITE);

        sector += len;

        kunmap\_atomic(buffer);

    }

    bio\_endio(bio);

    return BLK\_QC\_T\_NONE;

}

上面是一个虚拟块设备驱动的submit函数，该函数实际上做的事情就是遍历每个bio中的数据。将需要写入的数据写入到介质中（硬盘）。上面有一个宏bio\_for\_each\_segment，这个宏实际上就是一个for循环[[7]](#footnote-7)。通过bio\_data\_dir(bio)来获取BIO的读写属性，可以看到，每个bio中一定保存这读写的标志位。kmap\_atomic(bvec.bv\_page)是将保存在page中的数据映射到永久映射区中，为何这里需要将bv\_page映射到这个区域呢？这是因为bvec.bv\_page记录的数据在用户空间的地址，内核空间要想访问用户空间，则必须将用户空间映射到内核空间中。由于我们每次读写的数据都不是持续的（写入完毕就无效了），因此我们可以利用kmap\_atomic函数来实现临时的内存映射。

ram\_blk\_transfer函数将数据写入到实际的介质中，该函数的定义如下：

static void ram\_blk\_transfer(unsigned long sector,unsigned long nsect, char \*buffer, int write)

{

    unsigned long offset = sector << SECTOR\_SHIFT;

    unsigned long nbytes = nsect << SECTOR\_SHIFT;

    spin\_lock(&ram\_blk\_lock);

    unsigned long io;

    ram\_blk\_sursor = ram\_blk\_addr;

    ram\_blk\_sursor += offset;

    READ\_ONCE(\*buffer);

    if (write)

        io = memcpy(ram\_blk\_sursor,buffer,nbytes);

    else

        io = memcpy(buffer,ram\_blk\_sursor,nbytes);

    if (io == -1) {

        printk(KERN\_ERR"ram disk failed\n");

    }

    spin\_unlock(&ram\_blk\_lock);

}

上面的过程较为简单，也就是将数据写入到介质中，这里我们写入的是RAM中。最后我们需要实现出口函数，也就是exit函数，函数定义如下：

static void \_\_exit ram\_blk\_exit(void)

{

    del\_gendisk(ram\_gendisk);//删除硬盘

    blk\_cleanup\_disk(ram\_gendisk);//删除所有未完成的请求

    unregister\_blkdev(ram\_blk\_major, "ram\_blk");//注销块设备

}

上面的操作较为简单，硬盘卸载后需要删除硬盘设备号，同时删除未完成了bio，然后注销块设备即可。

#### 2.7.1.1 Linux5.9版本之前与之后区别

Linux 5.9中提交了一个patch，该patch将blk\_init\_queue函数删除，同时将在block\_device\_operations结构体中添加了submit\_bio函数指针，此后用户将不需要使用blk\_init\_queue函数来预先分配一个请求队列，然后再实现一个make\_request函数。现在我们只需要实现submit\_bio函数即可，也就是我们上面例子中讲到的对request进行遍历所有的bio。

#### 2.7.1.2 源码

#include <linux/module.h>

#include <linux/moduleparam.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/string.h>

#include <linux/blkdev.h>

#include <linux/bio.h>

#include <linux/proc\_fs.h>

#include <linux/uaccess.h>

#include <linux/hdreg.h>

int ram\_blk\_major;

struct gendisk \* ram\_gendisk;

spinlock\_t ram\_blk\_lock;

char \*ram\_blk\_addr = NULL;

char \*ram\_blk\_sursor=NULL;//定义一个写入游标

static void ram\_blk\_transfer(unsigned long sector,

        unsigned long nsect, char \*buffer, int write)

{

    unsigned long offset = sector << SECTOR\_SHIFT;

    unsigned long nbytes = nsect << SECTOR\_SHIFT;

    spin\_lock(&ram\_blk\_lock);

    unsigned long io;

    ram\_blk\_sursor = ram\_blk\_addr;

    ram\_blk\_sursor += offset;

    READ\_ONCE(\*buffer);

    if (write)

        io = memcpy(ram\_blk\_sursor,buffer,nbytes);

    else

        io = memcpy(buffer,ram\_blk\_sursor,nbytes);

    if (io == -1) {

        printk(KERN\_ERR"ram disk failed\n");

    }

    spin\_unlock(&ram\_blk\_lock);

}

static blk\_qc\_t ram\_blk\_submit\_bio(struct bio \*bio)

{

    struct bio\_vec bvec;

    struct bvec\_iter iter;

    sector\_t sector = bio->bi\_iter.bi\_sector;

    bio\_for\_each\_segment(bvec, bio, iter) {

        char \*buffer = kmap\_atomic(bvec.bv\_page) + bvec.bv\_offset;

        unsigned len = bvec.bv\_len >> SECTOR\_SHIFT;

        ram\_blk\_transfer(sector, len, buffer,

                bio\_data\_dir(bio) == WRITE);

        sector += len;

        kunmap\_atomic(buffer);

    }

    bio\_endio(bio);

    return BLK\_QC\_T\_NONE;

}

static int ram\_blk\_open(struct block\_device \*bdev, fmode\_t mode)

{

    return 0;

}

static void ram\_blk\_release(struct gendisk \*disk, fmode\_t mode)

{

}

static int ram\_blk\_getgeo(struct block\_device \*bdev, struct hd\_geometry \*geo)

{

    geo->heads = 1;

    geo->sectors = get\_capacity(ram\_gendisk);

    geo->cylinders = 1;

    return 0;

}

static int ram\_blk\_ioctl(struct block\_device \*bdev,fmode\_t mode,unsigned int cmd,unsigned long arg)

{

    return 0;

}

static const struct block\_device\_operations ram\_blk\_ops = {

    .owner      = THIS\_MODULE,

    .submit\_bio = ram\_blk\_submit\_bio,

    .open       = ram\_blk\_open,

    .release    = ram\_blk\_release,

    .getgeo     = ram\_blk\_getgeo,

    .ioctl      = ram\_blk\_ioctl,

};

static int \_\_init ram\_blk\_init(void)

{

    /\* 注册一个块设备，返回值为主设备号 \*/

    ram\_blk\_major = register\_blkdev(0, "ram\_blk");

    if ( ram\_blk\_major < 0)

    {

        printk(KERN\_ERR"ram disk register failed!\n");

        return -1;

    }

    ram\_gendisk = blk\_alloc\_disk(NUMA\_NO\_NODE);//分配一个gendisk

    if(!ram\_gendisk)

    {

        printk(KERN\_ERR"alloc\_disk failed!\n");

        return -1;

    }

    strcpy(ram\_gendisk->disk\_name, "ram\_blk");

    ram\_gendisk->major = ram\_blk\_major;//设置主设备号

    ram\_gendisk->first\_minor = 0;//设置第一个分区的次设备号

    ram\_gendisk->minors = 1;//设置分区个数：1

    ram\_gendisk->fops = &ram\_blk\_ops;//指定块设备ops集合

    set\_capacity(ram\_gendisk, 20480); //设置扇区数量：10MiB/512B=20480

    add\_disk(ram\_gendisk);//添加硬盘

    spin\_lock\_init(&ram\_blk\_lock);  //初始化自旋锁

    ram\_blk\_addr = (char\*)vmalloc(10\*2048\*512); //分配10M空间作为硬盘

    if(ram\_blk\_addr == NULL)

    {

        printk(KERN\_ERR"alloc memory failed!\n");

        return -1;

    }

    ram\_blk\_sursor = ram\_blk\_addr;

    return 0;

}

static void \_\_exit ram\_blk\_exit(void)

{

    del\_gendisk(ram\_gendisk);//删除硬盘

    blk\_cleanup\_disk(ram\_gendisk);//删除所有未完成的请求

    unregister\_blkdev(ram\_blk\_major, "ram\_blk");//注销块设备

}

module\_init(ram\_blk\_init);

module\_exit(ram\_blk\_exit);

MODULE\_LICENSE("GPL");

#### 2.7.1.3 测试

将上面的源码编译为.ko文件，然后拷贝到开发板上的根文件系统中，使用insmod命令挂载驱动，此时可以看到在/dev目录下生成一个ram\_blk设备文件节点。现在我们使用mkfs.ext4工具对该文件进行文件系统格式化，然后使用mount命令挂载该文件系统。

### 2.7.2 使用请求队列

在Linux中，块设备驱动的写入分为两种，一种就是上面的不使用请求队列的方式，这种方式每发起一个bio请求就会直接写入到介质中。另一种就是下面我们将要举例说明的使用请求方式，为何要使用队列呢？这是由于在Flash存储器广泛使用之前，HDD（固态硬盘）占据绝对统治地位，但固态硬盘有一个特点，就是顺序写入和读取的效率最高（因为读写靠磁头临近磁盘，磁盘旋转）。这样每次bio请求不能直接写入到介质中，而是先按地址的先后顺序排列之后，再写入，这样就可以大大提到读写速度（这便是赫赫有名的电梯算法）。当然，目前HDD已经开始退出观众视野，大部分的嵌入式设备也不会出现HHD，但作为一种Linux的重要的框架，还是有必要对其简单了解下。

## 2.8 总结

可以看到，块设备驱动要比字符设备驱动复杂的多，这主要是因为块设备驱动牵扯到的东西非常多且杂。而字符设备驱动相对要简单很多，在字符设备中，整个框架相对简单且薄，这是因为字符设备驱动都是单个单个写入的，而且由于字符设备文件太杂乱，不可能将很多东西都抽象出来，大部分需要用户来自己完成。但对于块设备驱动而言，其种类要少的多，当前无非就是HD，SSD，EMMC等，这些设备都是有严格的协议规范，因此Linux内核为了让块设备驱动开发变得更加简单，引入了大量的子系统，这也就是我们下面将会介绍的各种块设备子系统。

同样，我们以一个RAM存储方式来实现一个虚拟的块设备驱动，这次我们使用请求队列的方式实现。

# 第三章 MMC子系统

Linux内核为块设备驱动做了大量了工作，驱动工程师几乎可以不用关心如何去实现的，但对于一些芯片厂家而言，他们需要知道MMC的框架，因为他们的BSP工程师需要为他们的产品做底层主机的驱动开发。

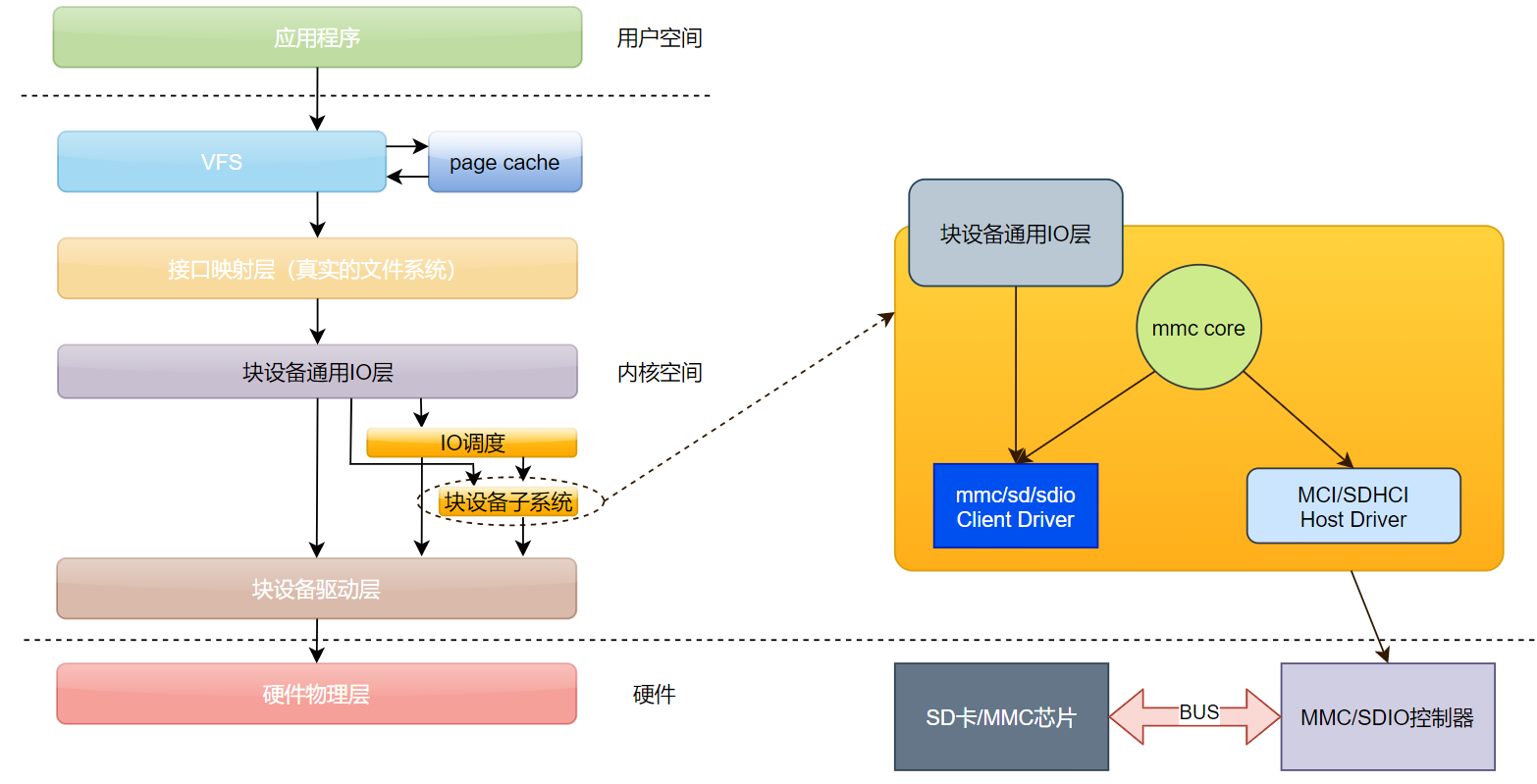


图3-1 mmc子系统框架图

上图是MMC子系统的框架图，MMC子系统属于块设备驱动子系统之一，从图中可以看到，块设备驱动子系统分为三个部分，分别是：Client Driver、MMC Core、Host Driver。其中Client Driver表示卡的驱动，该驱动也称为Card Driver层，该层就是我们所说的MMC驱动层。Host Driver表示控制器驱动，每个芯片所使用的MMC控制器都不一样，这主要表现为其寄存器、总线以及中断号都不相同，因此主机控制器驱动一般由芯片原厂的BSP工程师来完成驱动的开发，我们也称其为MMC的BSP驱动。MMC Core是Linux中mmc子系统的核心，该部分由Linux内核开发这来完成，这部分是不需要驱动工程师开发的。对于MMC而言，全世界的厂家就那么多，而且都遵循严格的协议，因此Linux内核已经实现了几乎所有的MMC驱动，对于驱动工程师只需要做非常小的改动（修改一些时序参数）甚至不需要做任何修改就可以直接使用。MMC子系统中唯一需要开发的就是Host Driver驱动，该部分也是驱动工程师修改最多的地方。

## 3.1 MMC Host驱动

主机驱动也就是我们所说的控制器驱动，该部分是提供给MMC Core层回调的，也就是MMC Core实现了MMC的所有的协议操作，但最终的读写只是一个钩子函数，具体的需要主机控制器驱动来实现。对于SoC厂家来说，Host Driver是需要BSP驱动开发工程师编写的，该部分，需要做的事情就是将一些读写、中断、时钟等一些操作实现并注册到驱动框架中即可。下面我们就来详细分析下主机控制器驱动。

### 3.1.1 mmc\_host结构体

struct mmc\_host {

    struct device       \*parent;

    struct device       class\_dev;

    int         index;

    const struct mmc\_host\_ops \*ops;

    struct mmc\_pwrseq   \*pwrseq;

    unsigned int        f\_min;

    unsigned int        f\_max;

    unsigned int        f\_init;

    u32         ocr\_avail;

    u32         ocr\_avail\_sdio; /\* SDIO-specific OCR \*/

    u32         ocr\_avail\_sd;   /\* SD-specific OCR \*/

    u32         ocr\_avail\_mmc;  /\* MMC-specific OCR \*/

    struct wakeup\_source    \*ws;        /\* Enable consume of uevents \*/

    u32         max\_current\_330;

    u32         max\_current\_300;

u32         max\_current\_180;

...

};

该结构体非常庞大，这里仅仅列举一部分，从该结构体可以看到，其中有一个struct mmc\_host\_ops \*ops成员，该成员是mmc\_host结构体中最重要的成员之一，也就是上层对底层的操作接口。下面我们来重点看这个结构体。

### 3.1.1 mmc\_host\_ops结构体

MMC子系统将主机驱动中的读写抽象为request，其具体的结构体如下所示：

struct mmc\_host\_ops {

    void    (\*post\_req)(struct mmc\_host \*host, struct mmc\_request \*req,int err);

    void    (\*pre\_req)(struct mmc\_host \*host, struct mmc\_request \*req);

    void    (\*request)(struct mmc\_host \*host, struct mmc\_request \*req);

    /\* Submit one request to host in atomic context. \*/

    int (\*request\_atomic)(struct mmc\_host \*host,struct mmc\_request \*req);

    void    (\*set\_ios)(struct mmc\_host \*host, struct mmc\_ios \*ios);

    int (\*get\_ro)(struct mmc\_host \*host);

    int (\*get\_cd)(struct mmc\_host \*host);

    void    (\*enable\_sdio\_irq)(struct mmc\_host \*host, int enable);

    /\* Mandatory callback when using MMC\_CAP2\_SDIO\_IRQ\_NOTHREAD. \*/

    void    (\*ack\_sdio\_irq)(struct mmc\_host \*host);

    /\* optional callback for HC quirks \*/

    void    (\*init\_card)(struct mmc\_host \*host, struct mmc\_card \*card);

    int (\*start\_signal\_voltage\_switch)(struct mmc\_host \*host, struct mmc\_ios \*ios);

    /\* Check if the card is pulling dat[0:3] low \*/

    int (\*card\_busy)(struct mmc\_host \*host);

    /\* The tuning command opcode value is different for SD and eMMC cards \*/

    int (\*execute\_tuning)(struct mmc\_host \*host, u32 opcode);

    /\* Prepare HS400 target operating frequency depending host driver \*/

    int (\*prepare\_hs400\_tuning)(struct mmc\_host \*host, struct mmc\_ios \*ios);

    /\* Prepare switch to DDR during the HS400 init sequence \*/

    int (\*hs400\_prepare\_ddr)(struct mmc\_host \*host);

    /\* Prepare for switching from HS400 to HS200 \*/

    void    (\*hs400\_downgrade)(struct mmc\_host \*host);

    /\* Complete selection of HS400 \*/

    void    (\*hs400\_complete)(struct mmc\_host \*host);

    /\* Prepare enhanced strobe depending host driver \*/

    void    (\*hs400\_enhanced\_strobe)(struct mmc\_host \*host,struct mmc\_ios \*ios);

    int (\*select\_drive\_strength)(struct mmc\_card \*card,unsigned int max\_dtr, int host\_drv,int card\_drv, int \*drv\_type);

    /\* Reset the eMMC card via RST\_n \*/

    void    (\*hw\_reset)(struct mmc\_host \*host);

    void    (\*card\_event)(struct mmc\_host \*host);

    int (\*multi\_io\_quirk)(struct mmc\_card \*card,

unsigned int direction, int blk\_size);

    /\* Initialize an SD express card, mandatory for MMC\_CAP2\_SD\_EXP. \*/

    int (\*init\_sd\_express)(struct mmc\_host \*host, struct mmc\_ios \*ios);

};

下面我们简单介绍下其中常用的成员。

* post\_req，pre\_req：这两个成员用来提供Linux中request的双缓冲机制，是可选项。
* request：该成员实现了mmc的读写操作请求，是ops中最重要的成员之一。
* request\_atomic：原子操作请求。
* set\_ios：该函数是core层调用寄存器的时候会最开始执行的函数，在该函数中一般需要实现mmc的识别、电源开启、时钟频率、总线宽度等必要的操作。
* get\_ro：返回卡的读写属性，可以使用内核默认的mmc\_gpio\_get\_ro来检测卡的读写属性。
* get\_cd：返回卡是否存在，可以使用内核默认的mmc\_gpio\_get\_cd函数。
* enable\_sdio\_irq：开启mmc或者SD卡的中断
* start\_signal\_voltage\_switch：切换卡的电压，大部分卡都默认3.3V，因此直接返回3.3V电压即可。
* hw\_reset：硬件复位，复位SD卡或者MMC卡。
* card\_busy：检测SD卡或者MMC卡是否为忙状态。

上面的函数中，request和set\_ios是必须要实现的，因此我们在写控制器驱动的时候必须实现这两个函数。

## 3.2 SDHCI 驱动

SDHCI（SDIO Host Controller Interface）,即SDIO主机控制器接口。SD卡对读者来说可能并不陌生，在大部分的数码相机中，存储数据所用的介质正是SD卡，其总线协议就是SDIO协议（当然，SD卡也支持SPI协议）。随着SD卡的普及以及该协议的优越性，越来越多的外围芯片采用SDIO协议来传输数据，例如ESP8089 WIFI芯片所用的接口便是SDIO协议。因此SDHCI驱动并不仅仅针对SD卡，同时还针对所有采用SDIO协议的设备。

# 第四章 SCSI子系统

# 第五章 UFS设备

1. 实际上，这些都是SSD，只是其协议不同，此处是为了更细化区分。 [↑](#footnote-ref-1)
2. EPROM：Erasable Programmable Read-Only Memory [↑](#footnote-ref-2)
3. EEPROM：Electrically Erasable Programmable read only memory [↑](#footnote-ref-3)
4. 向浮栅中注入电荷表示写入了'0'，没有注入电荷表示'1' [↑](#footnote-ref-4)
5. 在Linux 2.6版本之后，file\_operations中的ioctl已经删除不再使用，而是使用locked ioctl替代。 [↑](#footnote-ref-5)
6. 在之前的内核版本中，使用的是blk\_init\_queue函数，但在较新的内核中已经废除了该函数，这是因为之前的函数都是single queue，之后由于全部使用multi queue，之前的单队列就被废除了。 [↑](#footnote-ref-6)
7. 具体可参考文件：include/linux/bio.h [↑](#footnote-ref-7)