第 10 章 块设备驱动程序

块设备是通常以数据块大小（如 512 字节）为单位，能随机访问的设备。典型的块设备是系统中的存储设备，例如：硬盘、闪存、U 盘等。块设备内部按块进行划分，数据块大小由具体设备决定，通常为扇区（512 字节）的整数倍，这称为物理数据块，每个块按顺序编号。块设备驱动程序通过编号寻址到物理块，按块访问块设备，对连续块的访问可合并成一次访问，即一次可访问多个连续的物理数据块。

块设备驱动与字符设备驱动类似，驱动程序需要创建块设备驱动数据结构 gendisk 实例，并注册到块设备驱动数据库。但是，块设备驱动比字符设备驱动又要复杂许多。因为对字符设备而言，通常可以直接进行读写操作，数据量较小，可立即获得结果，不需要缓存，而块设备的读写数据量比较大，速度较慢，通常不能立即获得读写结果，读写操作需要缓存和延时。为此，块设备驱动中构建了请求队列用于缓存块设备访问操作，内核将对块设备的读写操作封装成请求，提交到请求队列，由驱动程序依次处理请求（执行数据传输）。请求队列是块设备驱动程序中最重要、最复杂的部分。

本章主要介绍块设备驱动框架、驱动程序实现流程、请求队列的构建及请求的处理、VFS 层对块设备的操作和管理等，最后介绍几种常见块设备驱动程序的实现。

块设备驱动通用层代码位于/block 目录下，具体块设备驱动程序位于/drivers/block 目录下，当然还有块设备驱动程序位于其它目录下。

10.1 块设备驱动框架

10.1.1 概述

请读者先看下图，图中示意了块设备驱动框架及 VFS（虚拟文件系统）层对块设备的管理和操作流程，虚线框内为块设备驱动内容，其它为 VFS 层内容。

系统中每个块设备（物理磁盘）在块设备驱动数据库中由 gendisk 结构体实例表示，gendisk 结构体包含磁盘的设备号、名称、分区表等信息。块设备驱动程序需要向块设备驱动数据库添加 gendisk 实例。

磁盘的分区信息由 hd_struct 结构体表示（gendisk 中包含此结构体的指针数组），在添加 gendisk 实例时，将读取磁盘上保存的分区信息（格式化磁盘时写入），填充至 hd_struct 结构体实例。分区 0 对应的 hd_struct 实例表示的是整个磁盘，分区 1 表示第 1 个分区，依此类推。

gendisk 关联的 block_device_operations 结构体用于实现对块设备的底层操作，包括打开设备、设备控制等（读写操作由请求队列执行），此实例需要具体块设备驱动程序定义。。

VFS 层对块设备的读写访问最终将转换成对块设备中数据块的读写操作。由于块设备的访问速度比较慢，数据量又比较大，对块设备的读写操作不能立即得到响应，获得结果。因此，块设备驱动程序中构建了请求队列（request_queue），用于缓存和管理 VFS 层提交的访问请求（request），由驱动程序依次处理队列中的请求，完成数据传输。

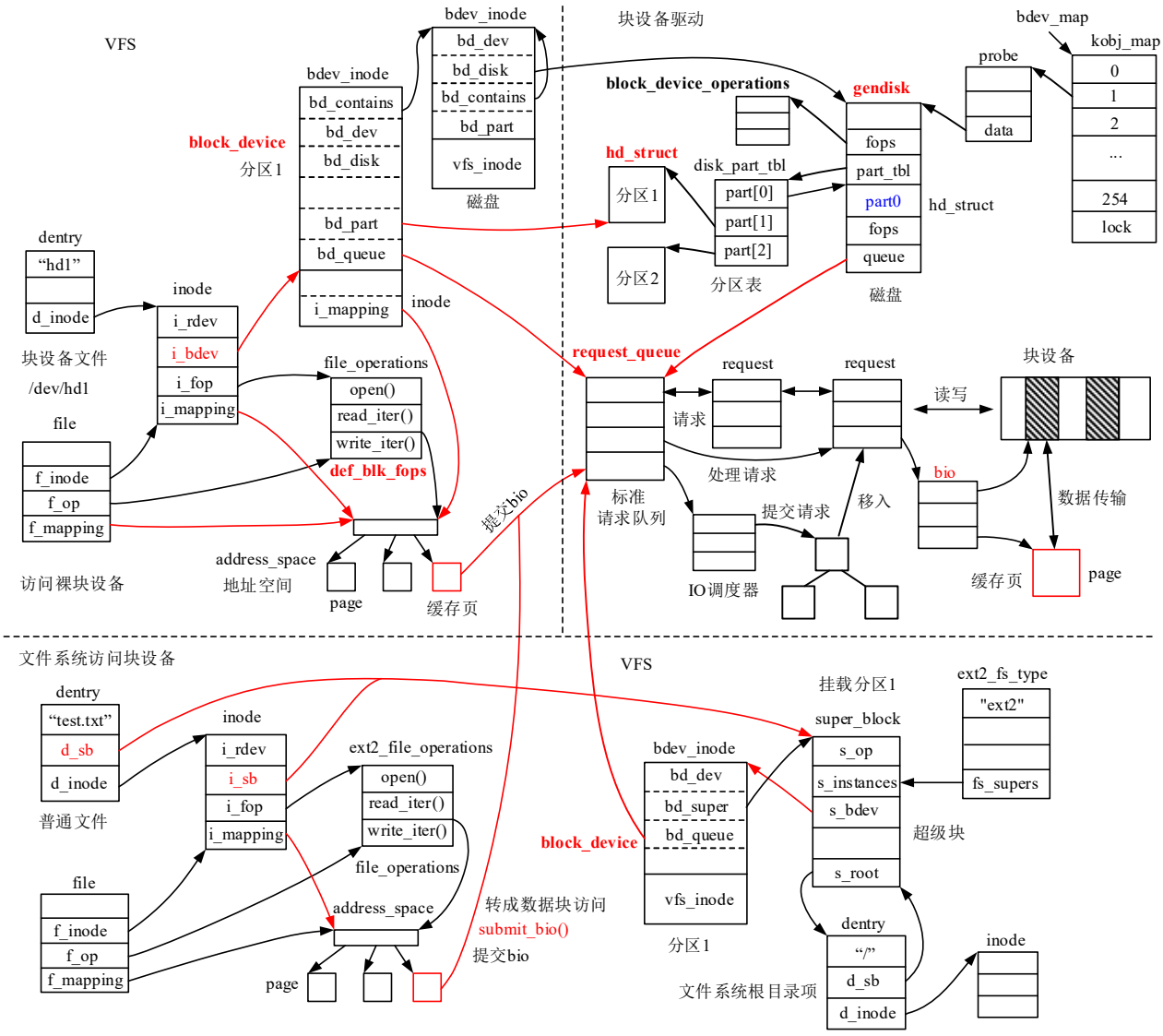
请求队列是块设备驱动程序中最重要和最复杂的部分，需由驱动程序创建并在添加 gendisk 实例之前赋予 gendisk 实例。请求队列主要分两种类型，一是标准请求队列，二是 Multi queue 请求队列。

标准请求队列中只包含一个请求组成的双链表，链表需由锁保护，在多核多进程系统中对锁的竞争将比较激烈，影响性能。Multi queue 请求队列中包含软件队列和硬件队列，每个 CPU 核对应一个软件队列，请求插入各自的软件队列时不会与其它 CPU 核竞争，软件队列中请求随后移入硬件队列进行处理。内核提供了创建标准请求队列和 Multi queue 请求队列的接口函数，块设备驱动程序可选择其中之一作为请求队列。下文中所说的请求队列将以标准请求队列为例进行说明，Multi queue 请求队列本章有专门的一节进行介绍。

驱动程序对队列中请求的处理可实施调度，区别轻重缓急，对访问相邻数据块的请求可以合并成一个更大的请求，以提高效率，因此请求队列中定义了 IO 调度器（类似于内核中对进程的调度器），用于管

理、调度对块设备的读写请求。

请求由 request 结构体表示，其中包含一个由 bio 结构体组成的链表，bio 结构体中包含了要读写块设备中哪些数据块的数据，以及读写数据保存在内存中哪个位置。驱动程序依此读写块设备中指定数据块，与指定内存进行数据传输。



块设备，包括磁盘和各个分区在 VFS 中由 bdev_inode 结构体表示。bdev_inode 结构体是 block_device 和 inode 结构体的包装器，block_device 结构体是磁盘和分区在 VFS 层的表示，包含 VFS 对块设备的读写操作信息。gendisk 和 hd_struct 结构体可理解为磁盘和分区在设备驱动层中的表示，包含设备的物理信息。

bdev_inode 实例在内核中由 bdev 伪文件系统管理，bdev_inode 结构体中的 inode 结构体成员是 bdev 伪文件系统中的 inode。在添加 gendisk 实例时，将创建表示整个磁盘的 bdev_inode 实例，并通过内嵌的 block_device 实例建立与 gendisk、hd_struct 和 **request_queue** 实例之间的关联。表示磁盘的 block_device 实例将作为其下分区 block_device 实例的容器。

在挂载分区或在打开块设备文件时，内核将为分区查找（或创建）对应的 bdev_inode 实例，通过内嵌的 block_device 实例关联 gendisk、hd_struct 和 request_queue 实例，并建立与块设备文件 inode 实例之间的关联。也就是说 block_device 实例是 VFS 层（块设备文件 inode）与块设备驱动程序之间的桥梁。

块设备（分区）必须先挂载到内核根文件系统，内核才能识别分区中文件系统内的文件和目录。在挂载分区文件系统时，将为文件系统创建超级块 super_block、根目录项 dentry 和节点 inode 结构体实例，在 bdev 伪文件系统中查找（或创建）分区对应的 block_device 实例，并建立与 super_block 实例之间的关联。

进程通过分区中文件访问块设备时，通过文件 inode 关联的 super_block 实例，获取 block_device 实例，进而获取块设备的请求队列 request_queue 实例。通常内核在内存中为打开文件建立了页缓存（地址空间）用于缓存文件内容，由页缓存与块设备进行数据传输。页缓存与块设备之间的数据传输由文件系统类型封装成 bio 实例，提交到请求队列，进而转换成请求添加到请求队列，由驱动程序处理。

进程通过块设备文件对磁盘或分区进行访问时，称为访问裸块设备。块设备中连续数据块的内容就是块设备文件的内容，文件内容与块设备中的数据是直接映射的关系。打开块设备文件时，将根据设备号查找 block_device 实例，进而获取请求队列。内核在内存中也为块设备文件内容建立了缓存，读写访问也封装成 bio 实例，提交到请求队列，由其进行处理。

综上所述，块设备驱动程序的主要工作就是创建表示磁盘的 gendisk 实例，创建 block_device_operations 实例和请求队列 request_queue 实例并赋予 gendisk 实例，将 gendisk 实例添加到块设备驱动数据库中。表示分区的 hd_struct 实例，将会在添加 gendisk 实例的操作中按实际分区数量动态创建并填充数据（从磁盘中读取分区信息）。

在 VFS 中表示分区的 block_device 实例在打开块设备文件或挂载分区文件系统时创建，并建立与磁盘 gendisk、分区 hd_struct 和请求队列 request_queue 实例的关联。block_device 实例是文件 inode 与块设备驱程序之间的桥梁，VFS 依此获取请求队列，将对块设备的访问操作提交到请求队列，由驱动程序完成数据传输。

10.1.2 数据结构

块设备驱动程序中主要的数据结构有：表示通用磁盘的 gendisk，表示分区的 hd_struct，表示请求队列的 request_queue，实现底层操作的 block_device_operations。

本小节主要介绍 gendisk、hd_struct 和 block_device_operations 结构体的定义，请求队列 request_queue 结构体后面再做介绍，VFS 层表示块设备（磁盘和分区）的 block_device 结构体也在后面介绍。

1 通用磁盘

物理上的块设备，如硬盘、U 盘、光盘等，在块设备驱动程序中统一由 gendisk 结构体表示，称之为通用磁盘，本书中简称磁盘。块设备驱动程序的主要工作就是创建和设置 gendisk 实例，并将其添加到块设备驱动数据库。

gendisk 结构体定义在/include/linux/genhd.h 头文件内：

```
struct gendisk {
    int    major;              /*磁盘主设备号*/
    int    first_minor;        /*起始从设备号*/
    int    minors;             /*从设备号数量，磁盘分区数为 minors-1，1 表示磁盘不分区*/

    char   disk_name[DISK_NAME_LEN];      /*磁盘名称，表示整个磁盘的块设备文件名称*/
    char   *(*devnode) (struct gendisk *gd, umode_t *mode);
                                          /*获取块设备文件名称的函数指针，优先级最高*/
    unsigned int    events;               /*支持的事件*/
    unsigned int    async_events;         /*异步事件*/

    struct disk_part_tbl    __rcu    *part_tbl;    /*指向磁盘分区表*/
    struct hd_struct part0;               /*表示整个磁盘的分区结构 hd_struct 实例，*/
```

```
    const struct block_device_operations    *fops;    /*底层块设备操作结构指针*/
    struct request_queue    *queue;                   /*指向请求队列*/
    void *private_data;                               /*私有数据，指向驱动程序定义的数据结构*/

    int    flags;                                     /*标记，用来描述驱动器状态的标志*/
    struct device    *driverfs_dev;                   /*指向表示磁盘父设备的 device 实例*/
    struct kobject *slave_dir;                        /*跟踪管理 gendisk 实例的 kobject 实例指针*/

    struct timer_rand_state    *random;
    atomic_t    sync_io;                  /* RAID */
    struct disk_events    *ev;            /*指向磁盘事件数据结构指针，/block/genhd.c*/
#ifdef    CONFIG_BLK_DEV_INTEGRITY
    struct blk_integrity    *integrity;   /*指向数据完整性数据结构，/include/linux/blkdev.h*/
#endif
    int    node_id;                       /*磁盘关联的内存结点编号*/
};
```

下面对 gendisk 结构体中几个重要的成员做简要介绍，请求队列到后面再单独介绍：

●**major**：磁盘的主设备号。

●**first_minor**：起始从设备号，通常为 0。

●**minors**：从设备号数量，从设备号 0 表示整个磁盘，minors 为 1 表示磁盘不分区，只有从设备号 0。minors 大于 1 时，磁盘分区的数量为 minors-1，从设备号 1 表示第一个分区，依此类推。

●**disk_name[]**：磁盘名称。

●**devnode()**：函数指针，在添加 hd_struct 实例创建设备文件时，用于获取设备文件名称。

●**part_tbl**：指向分区表 disk_part_tbl 结构体，disk_part_tbl 结构体内主要包含指向磁盘分区结构体的指针数组。disk_part_tbl 结构体定义在/include/linux/genhd.h 头文件：

```
struct disk_part_tbl {
    struct rcu_head    rcu_head;
    int    len;                                  /*part[]指针数组项数（分区数量加 1）*/
    struct hd_struct    __rcu    *last_lookup;  /*最近查找的分区实例指针*/
    struct hd_struct    __rcu    *part[];       /*分区 hd_struct 结构体指针数组*/
                                                /*part[0]指向 gendisk.part0*/
};
```

disk_part_tbl 实例在分配 gendisk 实例时，根据参数指定的分区数量动态创建，在添加 gendisk 实例时将扫描磁盘分区信息，根据磁盘真实的分区数确定是否需要扩展 disk_part_tbl 实例。

●**part0**：分区 hd_struct 结构体实例，表示整个磁盘。part_tbl 成员指向分区表实例中 part[0]指向此实例。

●**fops**：底层块设备操作结构 block_device_operations 指针，结构体中包含对磁盘执行底层操作的函数指针，如激活、释放磁盘等，block_device_operations 结构体定义见下文，block_device_operations 实例需由具体块设备驱动程序实现。

●**queue**：请求队列 request_queue 结构体指针，请求队列用于管理和处理 VFS 层对块设备发起的访问操作，是块设备驱动程序中至关重要的部分，请读者勿必牢记此成员，后面将详细介绍。

●**flags**：标记成员，取值定义在/include/linux/genhd.h 头文件，用于表示磁盘特性：

```
#define    GENHD_FL_REMOVABLE                    1    /*磁盘为可移动介质*/
```

```
#define   GENHD_FL_MEDIA_CHANGE_NOTIFY      4
#define   GENHD_FL_CD                       8    /*CD-ROM 设备*/
#define   GENHD_FL_UP                       16
#define   GENHD_FL_SUPPRESS_PARTITION_INFO  32
#define   GENHD_FL_EXT_DEVT                 64
                          /*允许扩展从设备号，最大分区数 DISK_MAX_PARTS（256）*/
#define   GENHD_FL_NATIVE_CAPACITY                  128
#define   GENHD_FL_BLOCK_EVENTS_ON_EXCL_WRITE       256
#define   GENHD_FL_NO_PART_SCAN                     512     /*不允许扫描磁盘分区*/
```

●**ev**：disk_events 结构体指针，结构体定义在/block/genhd.c 文件内，表示磁盘事件：

```
struct disk_events {
    struct list_head    node;          /*将实例添加到全局 disk_events 双链表*/
    struct gendisk      *disk;         /* the associated disk */
    spinlock_t          lock;

    struct mutex        block_mutex;  /* protects blocking */
    int             block;          /* event blocking depth */
    unsigned int        pending;  /* events already sent out */
    unsigned int        clearing;  /* events being cleared */

    long            poll_msecs;    /* interval, -1 for default */
    struct delayed_work     dwork;
};
```

内核定义了全局双链表 disk_events 用于管理 disk_events 实例。

2 分区

hd_struct 结构体表示磁盘分区的物理信息，gendisk 结构体中的 part0 成员（hd_struct 实例）表示整个磁盘的物理信息，如磁盘容量等。

hd_struct 结构体定义在/include/linux/genhd.h 头文件：

```
struct hd_struct {
    sector_t    start_sect;          /*起始扇区号，512 字节为一扇区*/
    sector_t    nr_sects;            /*扇区数量*/
    seqcount_t    nr_sects_seq;
    sector_t    alignment_offset;
    unsigned int    discard_alignment;
    struct device    __dev;                 /*在通用驱动模型中表示磁盘/分区的 device 实例*/
    struct kobject    *holder_dir;           /**/
    int    policy, partno;                  /*partno：分区编号，0 表示整个磁盘，1 表示第一个分区...*/
    struct partition_meta_info    *info;    /*分区元信息，如名称字符串等*/
#ifdef CONFIG_FAIL_MAKE_REQUEST
    int make_it_fail;
#endif
```

```
    unsigned long    stamp;
    atomic_t    in_flight[2];
#ifdef CONFIG_SMP
    struct disk_stats __percpu    *dkstats;        /*磁盘状态信息，percpu 变量*/
#else
    struct disk_stats    dkstats;        /*磁盘状态信息*/
#endif
    atomic_t    ref;                /*引用计数*/
    struct rcu_head    rcu_head;
};
```

hd_struct 结构体主要成员简介如下：

●**start_sect**：以 512 字节为扇区，分区在磁盘中的起始扇区号。

●**nr_sects**：分区包含的以 512 字节为扇区的扇区数量。

●**__dev**：内嵌 device 结构体成员，表示磁盘或分区设备，用于导出到通用驱动模型。在扫描分区信息时会调用 device_add()函数将 device 实例添加到驱动模型中，并自动创建设备文件。

●**partno**：分区编号，0 表示整个磁盘，1 表示第一个分区，依此类推。

●**info**：指向 partition_meta_info 结构体，表示分区元信息，结构体定义在/include/linux/genhd.h 头文件：

```
struct partition_meta_info {
    char uuid[PARTITION_META_INFO_UUIDLTH];            /*UUID，全局唯一 ID，挂载时用到*/
    u8 volname[PARTITION_META_INFO_VOLNAMELTH];        /*分区名称*/
};
```

●**dkstats**：disk_stats 结构体实例（或指针），表示磁盘状态信息，定义如下（/include/linux/genhd.h）：

```
struct disk_stats {
    unsigned long    sectors[2];    /* READs and WRITEs */
    unsigned long    ios[2];
    unsigned long    merges[2];
    unsigned long    ticks[2];
    unsigned long    io_ticks;
    unsigned long    time_in_queue;
};
```

在添加磁盘等操作中，扫描磁盘分区信息时将创建 hd_struct 结构体实例，关联到磁盘分区表结构中，扫描分区信息操作中会将 **hd_struct.__dev** 成员添加到通用驱动模型中，并自动创建分区设备文件。

3 块设备操作结构

block_device_operations 结构体包含对磁盘底层操作的函数指针，例如：激活设备、发送控制命令、按页读写块设备等。block_device_operations 结构体实例需由具体块设备驱动程序实现。

block_device_operations 结构体定义在/include/linux/blkdev.h 头文件：

```
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);    /*底层打开磁盘操作函数指针*/
    void (*release) (struct gendisk *, fmode_t);
    int (*rw_page)(struct block_device *, sector_t, struct page *, int rw);    /*整页读写操作函数*/
    int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);    /*发送控制命令*/
```

```
    int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
    long (*direct_access)(struct block_device *, sector_t,void **, unsigned long *pfn, long size);
    unsigned int (*check_events) (struct gendisk *disk,unsigned int clearing);
    int (*media_changed) (struct gendisk *);     /*弃用，用 check_events()代替*/
    void (*unlock_native_capacity) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo)(struct block_device *, struct hd_geometry *);     /*获取磁盘物理信息*/
    void (*swap_slot_free_notify) (struct block_device *, unsigned long);
    struct module *owner;
};
```

block_device_operations 结构体主要函数指针成员简介如下：

●**open**：打开块设备文件 open()系统调用中，块设备文件的文件操作结构中 open()函数（blkdev_open()函数）将调用此函数，完成磁盘的底层激活操作。

●**release**：释放磁盘的底层操作。

●**rw_page**：按页读写磁盘的函数。

●**ioctl**：ioctl()系统调用向块设备发送命令的操作中调用此函数。

●**check_events**：检查磁盘事件。

●**revalidate_disk**：使磁盘重新有效。

●**getgeo**：获取磁盘几何信息，如磁头、扇区、柱面（机械硬盘中的概念）等，填充至 hd_geometry 结构体实例。

10.1.3 初始化

内核在启动阶段需要为管理块设备驱动程序做一些准备工作，主要是创建块设备驱动数据库，即创建 kobj_map 结构体实例，为块设备驱动程序中的各数据结构创建 slab 缓存等。

块设备驱动初始化函数 **genhd_device_init()**定义在/block/genhd.c 文件内：

```
static int __init genhd_device_init(void)
{
    int error;

    block_class.dev_kobj = sysfs_dev_block_kobj;      /*块设备类 block_class，/block/genhd.c*/
                                /*sysfs_dev_block_kobj 对应/sys/dev/block 目录，见 devices_init()*/
    error = class_register(&block_class);                /*注册块设备类 block_class 实例*/
    if (unlikely(error))
        return error;
    bdev_map = kobj_map_init(base_probe, &block_class_lock);   /*创建 kobj_map 实例*/
    blk_dev_init();              /*创建数据结构 slab 缓存等，见下文，/block/blk-core.c*/

    register_blkdev(BLOCK_EXT_MAJOR, "blkext");      /*注册块设备号，/block/genhd.c*/

    if (!sysfs_deprecated)        /*sysfs_deprecated 通常为 0*/
        /*没有选择 SYSFS_DEPRECATED 和 SYSFS_DEPRECATED_V2，sysfs_deprecated 为 0*/
        block_depr = kobject_create_and_add("block", NULL);     /*sysfs 内创建/sys/block/目录*/
```

```
    return 0;
}
subsys_initcall(genhd_device_init);      /*内核初始化子系统时调用此函数*/
```

genhd_device_init()函数在内核初始化子系统时被调用，完成的主要工作如下：

（1）注册块设备类 block_class 实例，实例定义在/block/genhd.c 文件内。将在 sysfs 中创建表示块设备类的/sys/class/**block**/目录。添加 block_class 设备类设备时，将在/sys/class/**block**/目录下创建到设备目录的符号链接。

（2）创建和初始化块设备驱动数据库 kobj_map 实例。

（3）调用 blk_dev_init()函数为块设备驱动程序数据结构创建 slab 缓存等，见下文。

（4）调用 register_blkdev()函数注册 BLOCK_EXT_MAJOR 块设备主设备号（用于扩展分区）。

（5）在 sysfs 文件系统中创建/sys/block/目录。

下面简要介绍一下 blk_dev_init()函数和 register_blkdev()函数的实现。

1 创建数据结构缓存

blk_dev_init()函数在/block/blk-core.c 文件内实现，函数定义如下：

```
int __init blk_dev_init(void)
{
  BUILD_BUG_ON(__REQ_NR_BITS > 8 *FIELD_SIZEOF(struct request, cmd_flags));

  kblockd_workqueue = alloc_workqueue("kblockd", WQ_MEM_RECLAIM | WQ_HIGHPRI, 0);
                                                    /*创建工作队列*/
  if (!kblockd_workqueue)
        panic("Failed to create kblockd\n");

    request_cachep = kmem_cache_create("blkdev_requests",   \
                sizeof(struct request), 0, SLAB_PANIC, NULL);   /*创建请求 request 结构体 slab 缓存*/

    blk_requestq_cachep = kmem_cache_create("blkdev_queue",   sizeof(struct request_queue), 0, \
                            SLAB_PANIC, NULL);     /*创建请求队列 request_queue 结构体 slab 缓存*/
    return 0;
}
```

blk_dev_init()函数内创建了一个工作队列，为 request 和 request_queue 结构体创建了 slab 缓存。

2 注册块设备号

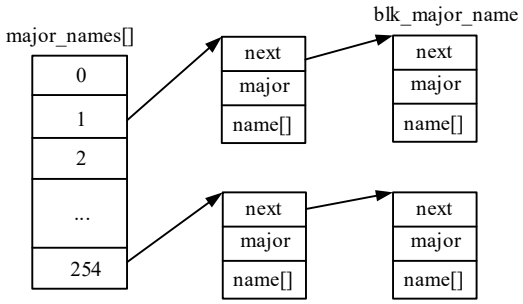
register_blkdev()函数用于向内核注册块设备号（主设备号），函数成功返回主设备号，否则返回错误码。在块设备驱动程序中也需要调用此函数注册主设备号。

内核定义了全局散列表，用于管理已使用的块设备主设备号，散列表定义在/block/genhd.c 文件内：

```
static struct blk_major_name {
    struct blk_major_name *next;     /*指向下一实例*/
    int major;                       /*主设备号*/
    char name[16];                   /*名称字符串*/
```

} *major_names[BLKDEV_MAJOR_HASH_SIZE];
/*BLKDEV_MAJOR_HASH_SIZE=255，/include/linux/fs.h*/

全局散列表结构如下图所示：



blk_major_name 实例依据（major%BLKDEV_MAJOR_HASH_SIZE）计算的散列值确定添加到的数组major_names[]中哪一项的链表中。

内核在/include/uapi/linux/major.h 头文件内定义了表示字符设备/块设备主设备号的宏，如：

```
#define   UNNAMED_MAJOR          0
#define   MEM_MAJOR              1
#define   RAMDISK_MAJOR          1
#define   FLOPPY_MAJOR           2
#define   PTY_MASTER_MAJOR       2
...
#define   BLOCK_EXT_MAJOR        259
#define   SCSI_OSD_MAJOR         260  /* open-osd's OSD scsi device */
```

register_blkdev()函数定义在/block/genhd.c 文件内，代码如下：

```
int register_blkdev(unsigned int major, const char *name)
/*major：注册的主设备号，0 表示由内核动态分配主设备号，name：名称字符串指针*/
{    struct blk_major_name **n, *p;
     int index, ret = 0;

     mutex_lock(&block_class_lock);     /*互斥锁*/

     if (major == 0) {       /*主设备号为 0，则动态分配主设备号，从大号往下搜索未使用的主设备号*/
          for (index = ARRAY_SIZE(major_names)-1; index > 0; index--) {
               if (major_names[index] == NULL)
                    break;
          }

          if (index == 0) {       /*没有未使用的设备号*/
               ...            /*如果[1，254]主设备号都被使用了，则不能动态分配主设备号*/
          }
          major = index;       /*动态分配的未使用的主设备号*/
          ret = major;         /*返回值*/
```

```
        }

        p = kmalloc(sizeof(struct blk_major_name), GFP_KERNEL);   /*创建 blk_major_name 实例*/
        ...

        p->major = major;          /*主设备号*/
        strlcpy(p->name, name, sizeof(p->name));
        p->next = NULL;
        index = major_to_index(major);       /*major%BLKDEV_MAJOR_HASH_SIZE*/

        for (n = &major_names[index]; *n; n = &(*n)->next) {
            if ((*n)->major == major)        /*判断主设备号 major 是否已经被注册*/
                break;
        }
        if (!*n)
            *n = p;       /*blk_major_name 实例添加到散列表，major 未被注册*/
        else
            ret = -EBUSY;       /*major 已经被注册，返回错误码*/

        ...
    out:
        mutex_unlock(&block_class_lock);
        return ret;
    }
```

register_blkdev()函数比较简单，函数内分两种情况进行处理：

（1）major 参数为 0：从[1，254]中最大主设备号开始往下搜索，查找未使用的主设备号，成功则创建并添加 blk_major_name 实例，返回主设备号，否则返回-EBUSY 错误码。

（2）major 参数不为 0：查找全局散列表判断 major 主设备号是否已被注册，如果没有则创建并添加 blk_major_name 实例，返回 0，否则返回-EBUSY 错误码。

在 genhd_device_init(void)函数内调用了 register_blkdev(BLOCK_EXT_MAJOR, "blkext")函数，向内核注册了 BLOCK_EXT_MAJOR（259）块设备主设备号，设备名称为“blkext”。

10.2 驱动程序实现

本节简要介绍块设备驱动程序的主要流程和接口函数，本章最后将会介绍具体块设备驱动程序的实现。

10.2.1 驱动程序流程

块设备驱动程序的主要流程是：

（1）调用 register_blkdev(major,name)函数注册（或动态分配）块设备主设备号。

（2）调用 **alloc_disk(minors)**函数分配通用磁盘 gendisk 结构体实例，定义 block_device_operations 结构体实例，并赋予 gendisk 实例，设置 gendisk（gendisk.part0）实例中部分成员值，如主设备号、磁盘容量

等。

（3）创建请求队列 request_queue 实例，并赋予 gendisk 实例，这是非常重要的一步，后面将详细讲。

（4）调用 **add_disk(disk)**函数向块设备驱动数据库添加 gendisk 实例。

在添加磁盘的 add_disk()函数内将创建表示磁盘的 block_device 结构体实例，激活磁盘，扫描磁盘中分区表信息，填充至分区 hd_struct 实例（动态创建实例），并添加 hd_struct 实例（创建设备文件）。

注册块设备主设备号的 register_blkdev()函数前面介绍过了，请求队列后面将专门介绍。本节介绍磁盘的分配和添加，需要注意的是添加磁盘是最后一步，在这之前要对其赋予 block_device_operations 实例和请求队列等。

10.2.2 分配磁盘

通用磁盘 gendisk 实例不能静态创建，只能由 alloc_disk(int minors)函数动态创建，参数 minors 表示从设备号的数量。minors 最小值为 1，表示磁盘不分区，大于 1 表示磁盘分区数量为 minors-1。从设备号 0 表示整个磁盘，1 表示第一个分区，依此类推。

alloc_disk(int minors)函数定义在/block/genhd.c 文件内，成功返回 gendisk 实例指针，否则返回 NULL。

```
struct gendisk *alloc_disk(int minors)
{
    return alloc_disk_node(minors, NUMA_NO_NODE);    /*/block/genhd.c*/
}
```

NUMA_NO_NODE 宏定义在/include/linux/numa.h 头文件：

```
#define   NUMA_NO_NODE      (-1)
```

alloc_disk_node(int minors, int node_id)函数定义在/block/genhd.c 文件内，代码如下：

```
struct gendisk *alloc_disk_node(int minors, int node_id)
{
    struct gendisk *disk;

    disk = kzalloc_node(sizeof(struct gendisk), GFP_KERNEL, node_id);
                                                    /*从通用缓存中分配 gendisk 实例*/
    if (disk) {
        if (!init_part_stats(&disk->part0)) {     /*单核系统直接返回 1*/
                /*SMP 系统为 disk->part0->dkstats 分配 percpu 实例，返回 1，/include/linux/genhd.h*/
            kfree(disk);
            return NULL;
        }
        disk->node_id = node_id;
        if (disk_expand_part_tbl(disk, 0)) {      /*创建（扩展）分区表，成功返回 0，/block/genhd.c*/
            ...
        }
        disk->part_tbl->part[0] = &disk->part0;     /*第一个分区指针指向 disk->part0*/
        seqcount_init(&disk->part0.nr_sects_seq);   /*赋值为 0*/
        hd_ref_init(&disk->part0);
```

```
        disk->minors = minors;          /*从设备号数量，分区数为 minors-1*/
        rand_initialize_disk(disk);     /*为 disk->random 赋值，/drivers/char/random.c*/
        disk_to_dev(disk)->class = &block_class;     /*part0->device->class，/include/linux/genhd.h*/
        disk_to_dev(disk)->type = &disk_type;        /*part0->device 设备类型，/block/genhd.c*/
        device_initialize(disk_to_dev(disk));        /*初始化表示整个磁盘的 device 实例，part0.__dev*/
    }
    return disk;
}
```

alloc_disk_node()函数不难理解，函数首先从通用缓存中分配 gendisk 实例，init_part_stats()在 SMP 系统中为磁盘（part0）分配 percpu 的 dkstats 实例，disk_expand_part_tbl()函数用于创建磁盘分区表，第一个分区指针指向表示整个磁盘的 gendisk->part0 实例，最后设置 **gendisk->part0.__dev** 成员。

1 创建分区表

下面看一下创建（或扩展）磁盘分区表 disk_part_tbl 实例的函数 disk_expand_part_tbl()，函数定义在/block/genhd.c 文件内，成功返回 0，否则返回错误码：

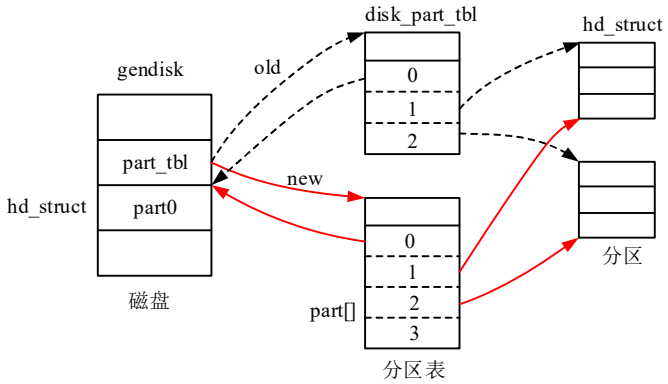
```
int disk_expand_part_tbl(struct gendisk *disk, int partno)
/*disk：指向 gendisk 实例，partno：表示磁盘分区的数量，0 表示不分区，这里为 0 默认不分区*/
{
    struct disk_part_tbl *old_ptbl = disk->part_tbl;     /*指向旧分区表实例*/
    struct disk_part_tbl *new_ptbl;                      /*新分区表指针*/
    int len = old_ptbl ? old_ptbl->len : 0;              /*旧分区数量，初始化时为 0*/
    int i, target;
    size_t size;

    target = partno + 1;      /*新分区表中分区指针数组项数，第一项表示整个磁盘，所以要加 1*/

    if (target < 0)
        return -EINVAL;

    if (disk_max_parts(disk) && target > disk_max_parts(disk))   /*分区数量不能超过最大值*/
        return -EINVAL;                 /*disk 设置了 GENHD_FL_EXT_DEVT 标记，则不能超过 256*/
                                        /*否则不能超过 disk->minors，/include/linux/genhd.h*/

    if (target <= len)       /*新分区数量小于等于旧分区数量，直接返回，不需要扩展分区表*/
        return 0;

    size = sizeof(*new_ptbl) + target * sizeof(new_ptbl->part[0]);      /*新分区表实例大小*/
    new_ptbl = kzalloc_node(size, GFP_KERNEL, disk->node_id);     /*为新分区表分配空间*/
    if (!new_ptbl)
        return -ENOMEM;

    new_ptbl->len = target;        /*分区数量加 1，part[]指针数组项数*/
```

```
    for (i = 0; i < len; i++)       /*len 为旧分区数量，新分区表重用旧分区 hd_struct 实例*/
        rcu_assign_pointer(new_ptbl->part[i], old_ptbl->part[i]); /*新分区表分区指针指向旧分区实例*/

    disk_replace_part_tbl(disk, new_ptbl);  /*指向新分区表实例，并释放旧分区表，/block/genhd.c*/
    return 0;
}
```

disk_expand_part_tbl()函数比较简单，参数 partno 表示新分区数量，新分区表中 hd_struct 实例指针数组项数为 partno+1，因为第一项表示整个磁盘。

如果 disk 已存在分区表，且旧分区表中分区数量不会少于新分区表中分区数量（分区数量没有增加），则不需要扩展分区表，否则要创建新的分区表替换旧的分区表，新分区表中分区指针依然指向原分区表中指向的 hd_struct 实例，新分区表赋予 disk 后，原分区表将被释放。

disk_expand_part_tbl()函数执行结果如下图所示：



2 磁盘设备类型

gendisk 实例中 part0 成员是表示整个磁盘的 hd_struct 实例，其__dev 成员为表示整个磁盘的 device 实例，用于将 gendisk 实例导出到通用驱动模型和 sysfs 文件系统中，并作为所有磁盘分区 hd_struct.__dev 成员的父设备。

gendisk.part0.__dev 成员（device 实例）所属设备类 block_class 实例定义在/block/genhd.c 文件内，并在 genhd_device_init()函数内注册。

gendisk.part0.__dev 成员所属设备类型为 disk_type，实例定义在/block/genhd.c 文件内：

```
static struct device_type disk_type = {         /*磁盘设备类型*/
    .name       = "disk",
    .groups     = disk_attr_groups,     /*磁盘默认设备属性组*/
    .release    = disk_release,
    .devnode = block_devnode,       /*获取设备名称函数，/block/genhd.c*/
};
```

内核在在/block/genhd.c 文件内定义了磁盘 device 实例的默认设备属性数组，请读者自行查阅源代码。

block_devnode()函数用于在自动创建设备文件时获取设备文件名（具有最高优先权），函数定义如下：

```
static char *block_devnode(struct device *dev, umode_t *mode,kuid_t *uid, kgid_t *gid)
{
    struct gendisk *disk = dev_to_disk(dev);
    if (disk->devnode)
```

```
        return disk->devnode(disk, mode);       /*调用 gendisk 实例中 devnode()函数*/
    return NULL;
}
```

在注册分区 hd_struct 实例为其创建设备文件时，从 gendisk.devnode()函数获取设备文件名称具有最高的优先权。

10.2.3 添加磁盘

块设备驱动程序在分配 gendisk 实例后，还需要对实例进行设置，如设置主设备号、赋予请求队列指针、赋予 block_device_operations 实例指针、设置磁盘容量等，最后调用 add_disk()函数将 gendisk 实例添加到块设备驱动数据库。

add_disk(struct gendisk *disk)函数定义在/block/genhd.c 文件内，代码如下：

```
void add_disk(struct gendisk *disk)
{
    struct   backing_dev_info *bdi;     /*后备存储设备信息，/include/linux/backing-dev-defs.h*/
    dev_t   devt;       /*磁盘的设备号（从设备号为 0）*/
    int   retval;       /*保存返回结果*/

    WARN_ON(disk->minors && !(disk->major || disk->first_minor));
    WARN_ON(!disk->minors && !(disk->flags & GENHD_FL_EXT_DEVT));

    disk->flags |= GENHD_FL_UP;                     /*设置标记位*/

    retval = blk_alloc_devt(&disk->part0, &devt);       /*MKDEV(disk->major, 0)*/
                /*为 hd_struct 实例 part0 生成设备号，即表示整个磁盘的设备号，/block/genhd.c*/
                /*如果 part->partno >=disk->minors，主设备号改为 BLOCK_EXT_MAJOR*/
    ...         /*错误处理*/

    disk_to_dev(disk)->devt = devt;         /*disk->part0.__dev->devt=devt，保存设备号到 device 实例*/
    disk->major = MAJOR(devt);              /*主设备号*/
    disk->first_minor = MINOR(devt);        /*给 gendisk 赋予起始从设备号，通常为 0*/

    disk_alloc_events(disk);                /*创建 disk_events 实例赋予 disk->ev 成员，/block/genhd.c*/

    bdi = &disk->queue->backing_dev_info;     /*指向请求队列中内嵌的 backing_dev_info 实例*/
    bdi_register_dev(bdi, disk_devt(disk));     /*/mm/backing-dev.c*/
                        /*注册 backing_dev_info 实例，将其添加到 bdi_list 全局双链表末尾等*/
    blk_register_region(disk_devt(disk), disk->minors, NULL,exact_match, exact_lock, disk);
                            /*调用 kobj_map()函数向块设备数据库添加 gendisk 实例，/block/genhd.c*/
    register_disk(disk);                    /*注册磁盘，/block/genhd.c*/
    blk_register_queue(disk);           /*注册请求队列（在 sysfs 中创建目录和文件），/block/blk-sysfs.c*/

    WARN_ON_ONCE(!blk_get_queue(disk->queue));
```

```
    retval = sysfs_create_link(&disk_to_dev(disk)->kobj, &bdi->dev->kobj,  "bdi");  /*创建符号链接*/
                                    /*在表示磁盘的目录下创建到 backing_dev_info 的符号链接“bdi”*/
    WARN_ON(retval);

    disk_add_events(disk);          /*disk_events 实例添加到 disk_events 双链表等，/block/genhd.c*/
}
```

add_disk()函数完成的主要工作如下：

（1）调用 blk_alloc_devt()函数为 part0 生成设备号，正常是 MKDEV(disk->major, 0)，并赋予 gendisk 实例。

（2）为 gendisk 实例分配并初始化 disk_events 实例，赋予 disk->ev 成员，在函数最后还要添加实例，主要是将实例添加到 disk_events 全局双链表，并在 sysfs 中表示磁盘的目录下创建属性文件等。

（3）注册 gendisk 实例关联请求队列 request_queue 实例中内嵌的 backing_dev_info 结构体成员，见第 11 章。

（4）将 gendisk 实例添加到块设备驱动数据库。

（5）调用 **register_disk(disk)**函数注册 gendisk 实例。

（6）完成 gendisk 实例关联请求队列在 sysfs 中的注册工作，后面再介绍。

下面主要看一下注册磁盘的 register_disk(disk)函数所做的工作。

1 注册磁盘

注册磁盘 register_disk(disk)函数定义如下（/block/genhd.c）：

```
static void register_disk(struct gendisk *disk)
{
    struct device *ddev = disk_to_dev(disk);        /*表示磁盘的 device 实例 disk->part0.__dev*/
    struct block_device    *bdev;        /*指向在 VFS 中表示磁盘的 block_device 实例*/
    struct disk_part_iter    piter;        /*分区迭代器，/include/linux/genhd.h*/
    struct hd_struct    *part;
    int err;

    ddev->parent = disk->driverfs_dev;    /*磁盘 device 实例的父设备，由驱动程序设置*/

    dev_set_name(ddev, "%s", disk->disk_name); /*设置磁盘 device 实例名称为 disk->disk_name*/

    dev_set_uevent_suppress(ddev, 1);       /*不向用户空间传递 uevents，/include/linux/device.h*/

    if (device_add(ddev))     /*添加表示磁盘的 device 实例，创建表示磁盘的设备文件等*/
        return;
    if (!sysfs_deprecated) {
        err = sysfs_create_link(block_depr, &ddev->kobj,kobject_name(&ddev->kobj));
                                /*在/sys/block/目录下创建到磁盘的符号链接（目录），名称同磁盘目录*/
        ...
    }

    pm_runtime_set_memalloc_noio(ddev, true);
```

```
        /*在 sysfs 中磁盘目录下创建子目录*/
        disk->part0.holder_dir = kobject_create_and_add("holders", &ddev->kobj);
        disk->slave_dir = kobject_create_and_add("slaves", &ddev->kobj);

        if (!disk_part_scan_enabled(disk))     /*磁盘是否分区且可扫描，minors>1，/include/linux/genhd.h*/
            goto exit;

        if (!get_capacity(disk))   /*获取磁盘容量（返回 disk->part0.nr_sects），/include/linux/genhd.h*/
            goto exit;                  /*disk->part0.nr_sects 值在添加磁盘前赋值*/

        bdev = bdget_disk(disk, 0);   /*创建表示磁盘的 block_device 实例，见本章下文，/block/genhd.c*/
        if (!bdev)
            goto exit;

        bdev->bd_invalidated = 1;     /*设置分区信息无效*/
        err = blkdev_get(bdev, FMODE_READ, NULL);   /*打开块设备，/fs/block_dev.c*/
                                /*激活块设备，扫描分区信息，添加分区等，见本章下文*/
        if (err < 0)
            goto exit;
        blkdev_put(bdev, FMODE_READ);     /*/fs/block_dev.c*/

    exit:
        dev_set_uevent_suppress(ddev, 0);          /*可以向用户空间传递 uevents 了*/
        kobject_uevent(&ddev->kobj, KOBJ_ADD);       /*触发磁盘的 uevent 事件*/

        disk_part_iter_init(&piter, disk, 0);         /*初始化分区迭代器，/block/genhd.c*/
        while ((part = disk_part_iter_next(&piter)))         /*迭代分区*/
            kobject_uevent(&part_to_dev(part)->kobj, KOBJ_ADD);
                                                /*向用户空间发送添加分区的 uevent 事件*/
        disk_part_iter_exit(&piter);     /*结束分区迭代器*/
    }
```

注册磁盘函数主要工作如下：

（1）设置表示整个磁盘的 device 实例（disk->part0.__dev）的名称为 disk->disk_name，并调用函数 device_add(ddev)添加实例，添加过程中将自动创建表示磁盘的设备文件，在 sysfs 中也将创建表示磁盘的目录，名称为 disk_name。

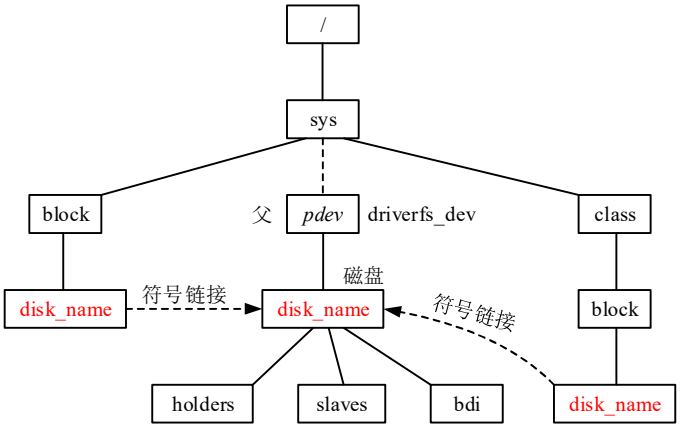
（2）调用 bdget_disk(disk, 0)函数（内部调用 **bdget()**函数）在 bdev 伪文件系统中创建（或以设备号查找）表示整个磁盘的 block_device 实例。每个磁盘和分区在 VFS 中有一个对应的 block_device 实例（内嵌在 bdev_inode 实例中），由 bdev 伪文件系统管理，详见本章下文。

（3）调用 blkdev_get()函数打开磁盘（块设备）。

blkdev_get()函数将调用__blkdev_get()函数，在这里的主要工作是调用 block_device_operations 实例中的 open()函数执行底层打开磁盘操作，再调用 **rescan_partitions(disk, bdev)**函数从磁盘中获取分区信息，填充并添加 hd_struct 实例，建立表示磁盘 block_device 实例与 gendisk、hd_struct（gendisk->part0）和 request_queue 等实例之间的关联。blkdev_get()函数实现详见本章下文。

（4）向用户空间发送添加磁盘和分区的 uevent 事件。

注册磁盘将在 sysfs 中为磁盘创建目录和文件，如下图所示：

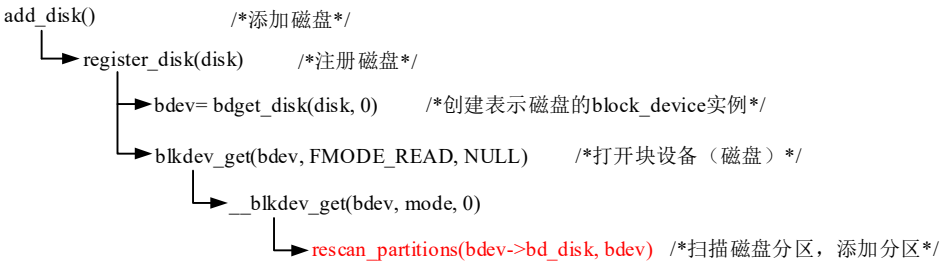


磁盘的父设备 driverfs_dev 由驱动程序指定，在注册磁盘时，将在表示父设备的目录下创建表示磁盘的子目录，磁盘目录下有 holders、slaves、bdi 等子目录，还在磁盘设备属性文件（未画出）。

在/sys/block/和/sys/class/block/目录下将创建到磁盘目录的符号链接，名称与磁盘目录名称相同。

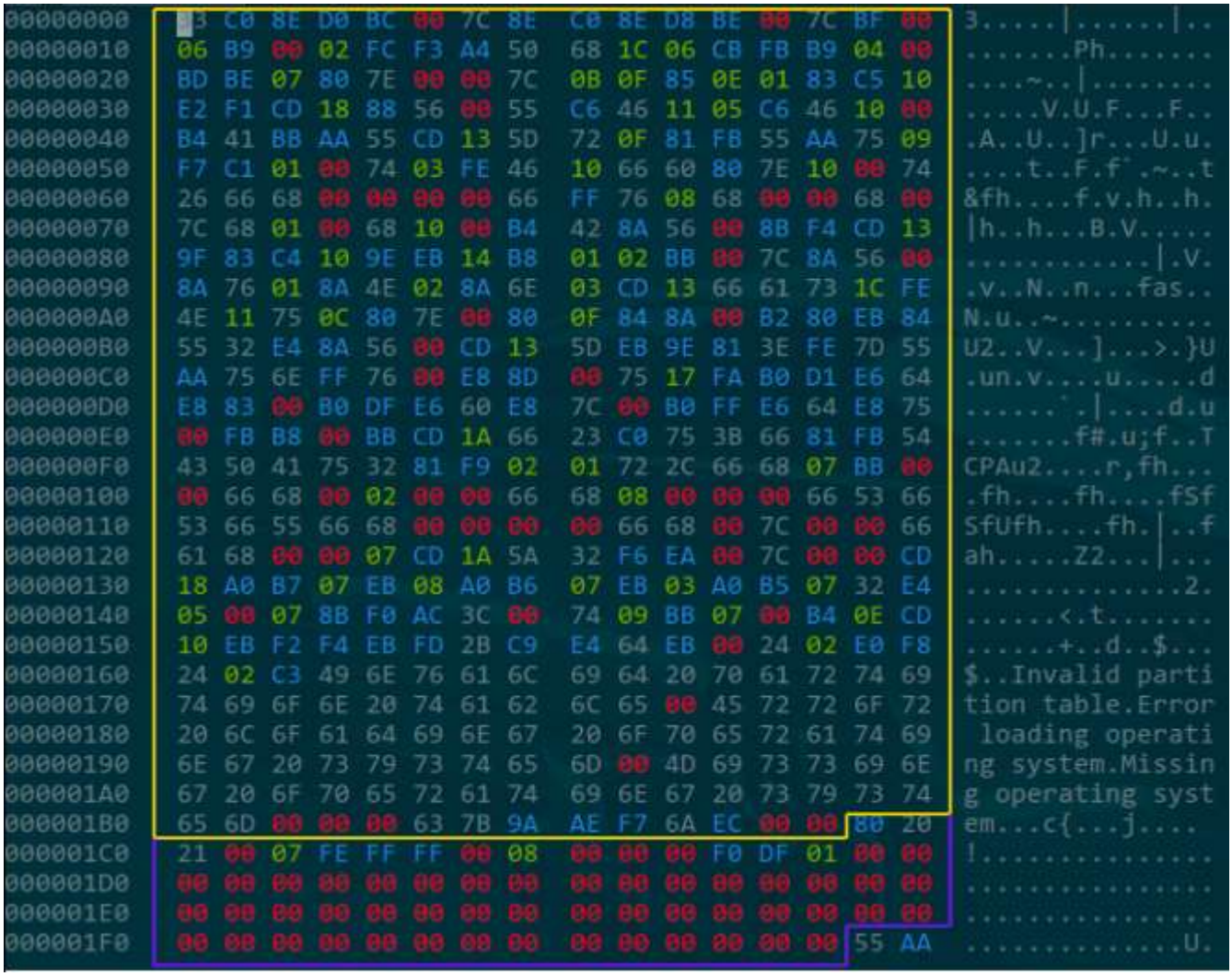
2 扫描磁盘分区

下面再看一下在添加磁盘时是如何扫描磁盘分区，添加分区的，函数调用关系如下：



blkdev_get()函数主要在打开块设备文件时调用，用于获取 block_device 对应的 gendisk、hd_struct 实例，如果分区的信息无效将会扫描磁盘中的分区信息，填充至 hd_struct 实例，并添加分区实例。在后面介绍块设备文件操作时再详见介绍 blkdev_get()函数的实现。这里主要关注的是 **rescan_partitions()**函数，它用于扫描磁盘中的分区信息，并填充、添加分区 hd_struct 实例。

在格式化磁盘时，会将磁盘的分区信息写入磁盘中规定位置，这样就可以从中读取磁盘的分区信息。磁盘的分区类型有很多种，Linux 内核支持多达十几种的分区类型，这里我们以最常见的 Windows 分区类型为例（MBR 分区），说明磁盘分区表的结构。Windows 分区类型中磁盘分区表保存在磁盘第一个扇区中（开头 512 字节），第一个扇区称之为 MBR 扇区（主引导扇区），其内容如下图所示（图片来自网络）。



图中黄色框内数据，即前 446 字节，为引导代码，用于引导操作系统，这里我们不管它。图中紫色框内数据即是我们需要的分区表，起始偏移量为 0x01BE，共 64 个字节。其中每 16 个字节表示一个分区信息，所以我们在 Window 系统中对磁盘分区只能分 4 个主分区（最后一个分区可以是扩展分区）。扇区最后两个字节为 0x55 和 0xAA，这是 Windows 分区类型的标记，用于识别 Windows 类型的分区。

16 个字节的一个分区信息语义如下（以第一条分区表信息为例）。

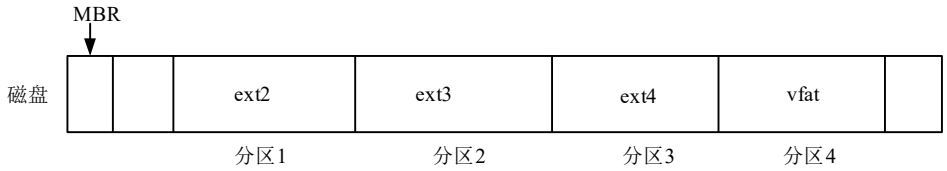
字节偏移量	长度	描述
0x01BE	1 字节	引导指示符，指明分区是否是活动分区，活动分区为 0x80（分区是否包含系统引导扇区，用于引导操作系统），其它为 0x00。
0x01BF	1 字节	开始磁头
0x01C0	6 位	开始扇区，占用 6 位
0x01C1	10 位	开始柱面，占用 10 位
0x01C2	1 字节	分区文件系统类型，0x07 为 NTFS
0x01C3	1 字节	结束磁头
0x01C4	6 位	结束扇区，占用 6 位
0x01C5	10 位	结束柱面，占用 10 位
0x01C6	4 字节	相对扇区数，从磁盘开始处到该分区开始的扇区偏移量，以扇区为单位。

0x01CA	4 字节	该分区扇区总数

表中的磁头、柱面等是机械磁盘中的概念，不必太在意，分区类型代码会对其进行识别，我们重点关注的是最后 8 个字节，从中可获取**分区起始扇区号**和**扇区数量**的信息。

现代计算机磁盘主要是 GPT 分区（efi）类型了，以支持 UEFI 固件，有兴趣的读者可以研究一下。

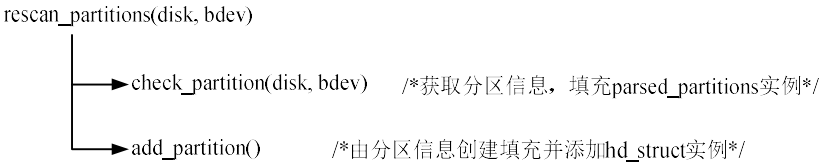
注意：磁盘分区类型与文件系统类型是不同的概念。磁盘分区类型是描述如何将磁盘划分成多个分区，并规定分区信息的格式和保存位置等，而文件系统类型是描述如何对分区进行划分，以保存目录和文件。如下图所示，MBR 分区类型将分区信息保存在第一个扇区（其它分区类型可以不一样），而各个分区可以格式化成不同的文件系统类型，如 ext2、ext3、ext4 等。



■获取磁盘分区信息

由前面的介绍可知，**rescan_partitions()**函数用于扫描磁盘分区信息，以填充和添加分区 hd_struct 实例。

rescan_partitions()函数调用关系如下所示（/block/partition-generic.c）：



check_partition()函数用于读取磁盘中的分区信息，缓存到临时数据结构中，add_partition()函数用于从临时数据结构中获取分区信息填充、添加分区 hd_struct 实例。

内核在/block/partitions/check.h 头文件内定义了 parsed_partitions 结构体，用于缓存从磁盘读取的分区信息。

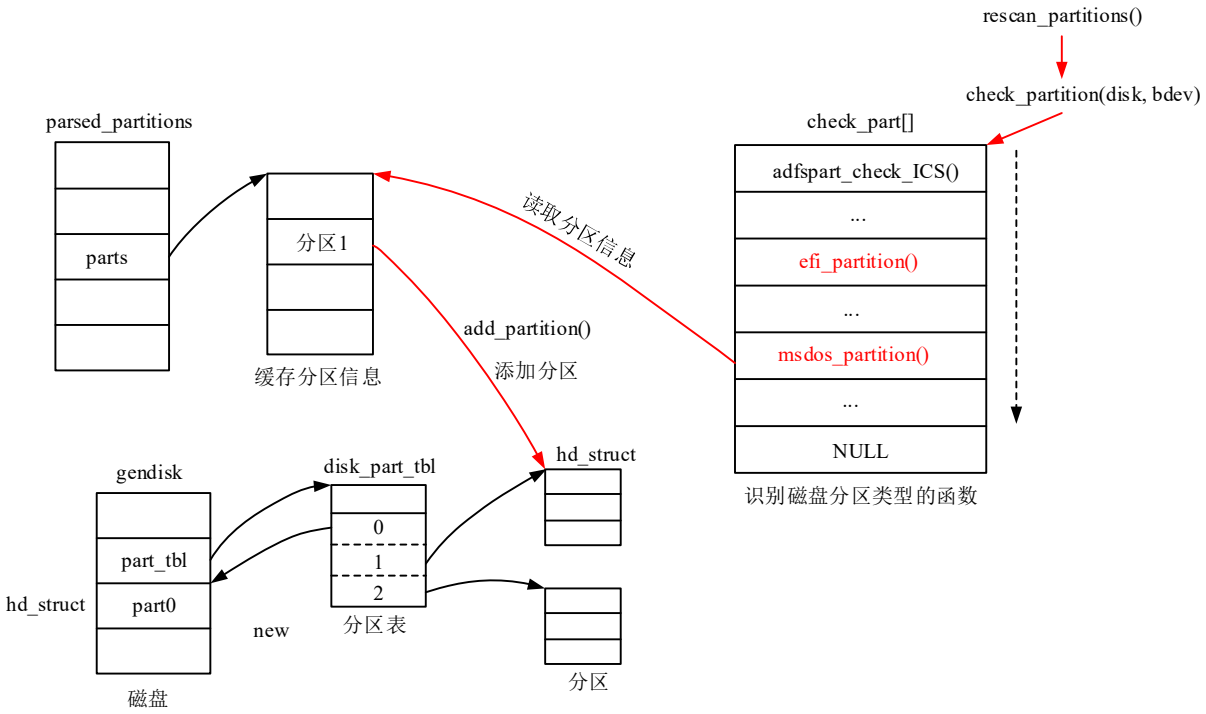
```
struct parsed_partitions {
    struct block_device *bdev;          /*指向表示整个磁盘的 block_device 实例*/
    char name[BDEVNAME_SIZE];       /*磁盘名称，复制于 gendik.disk_name*/
    struct {
        sector_t from;              /*起始扇区号*/
        sector_t size;              /*分区大小，扇区数*/
        int flags;
        bool has_info;          /*是否具有分区元信息，即 info 成员不为 NULL*/
        struct partition_meta_info info;        /*分区元信息，/include/linux/genhd.h*/
    } *parts;       /*指向结构体数组，数组项用于缓存分区信息*/
    int next;
    int limit;      /*支持的最大分区数量，通常为 gendisk.minors*/
    bool access_beyond_eod;
    char *pp_buf;       /*保存磁盘名称*/
};
```

每种分区类型需要定义一个函数，用于从磁盘中读取分区信息并填充至 parsed_partitions 实例。分区类型相关代码位于/block/partitions/目录下，例如，Window 分区类型相关代码位于/block/partitions/msdos.c 文件内，读取分区信息函数为 int msdos_partition(struct parsed_partitions *state)，有兴趣的读者可阅读源代码。

内核在/block/partitions/check.c 文件内定义了 check_part[]数组，这是一个函数指针数组，保存各分区类型代码中定义函数的指针。

check_partition()函数用于逐个调用 check_part[]数组中的函数（如下图所示），检查磁盘是不是这一类型的分区，如果是则读取函数会将分区信息填充至 parsed_partitions 实例，函数返回，如果不是则调用数组中下一个函数，直至到达数组末尾。

rescan_partitions()函数在调用 check_partition()函数读取磁盘分区信息后，随后调用 add_partition()函数依据 parsed_partitions 实例中缓存的分区信息填充并添加分区 hd_struct 实例，如下图所示。



check_partition()函数定义在/block/partitions/check.c 文件内，用于识别磁盘分区类型，读取分区信息，创建并填充至 parsed_partitions 结构体实例，函数返回 parsed_partitions 实例指针，函数代码如下：

```
struct parsed_partitions *check_partition(struct gendisk *hd, struct block_device *bdev)
/*hd：指向表示磁盘的 gendisk 实例，bdev：指向表示整个磁盘的 block_device 实例*/
{
    struct parsed_partitions *state;
    int i, res, err;

    state = allocate_partitions(hd);   /*分配并初始化 parsed_partitions 实例，/block/partitions/check.c*/
    if (!state)
        return NULL;
    state->pp_buf = (char *)__get_free_page(GFP_KERNEL);   /*分配一页内存*/
    ...
    state->pp_buf[0] = '\0';      /*设置分配内存末尾字符*/

    state->bdev = bdev;       /*指向表示磁盘的 block_device 实例*/
```

```
        disk_name(hd, 0, state->name);
                                /*复制 gendik.disk_name 至 state->name，/block/partition-generic.c*/
        snprintf(state->pp_buf, PAGE_SIZE, " %s:", state->name);      /*复制 state->name 至 state->pp_buf*/
        if (isdigit(state->name[strlen(state->name)-1]))    /*名称字符串最后一个为数字*/
            sprintf(state->name, "p");

        i = res = err = 0;

        /*依次调用 check_part[]中函数，识别分区类型，读取分区信息，填充 parsed_partitions 实例*/
        while (!res && check_part[i]) {
            memset(state->parts, 0, state->limit * sizeof(state->parts[0]));      /*state->parts 指向数组清零*/
            res = check_part[i++](state);      /*成功返回正值，出错返回错误码，否则返回 0*/
            ...
        }
        if (res > 0) {          /*读取磁盘分区信息成功*/
            printk(KERN_INFO "%s", state->pp_buf);
            free_page((unsigned long)state->pp_buf);
            return state;       /*返回 parsed_partitions 实例指针*/
        }
        ...
    }
```

■扫描分区函数

下面来看一下 **rescan_partitions()**函数的定义，代码如下（/block/partition-generic.c）：

```
int rescan_partitions(struct gendisk *disk, struct block_device *bdev)
/*disk：指向磁盘 gendisk 实例，bdev：指向表示整个磁盘的 block_device 实例*/
{
    struct parsed_partitions *state = NULL;
    struct hd_struct *part;
    int p, highest, res;
rescan:
    if (state && !IS_ERR(state)) {
        free_partitions(state);
        state = NULL;
    }
    res = drop_partitions(disk, bdev);    /*释放磁盘现有分区实例，/block/partition-generic.c*/
    if (res)
        return res;

    if (disk->fops->revalidate_disk)          /*block_device_operations 实例中函数*/
        disk->fops->revalidate_disk(disk);    /*使磁盘重新有效*/
    check_disk_size_change(disk, bdev);
```

```
                    /*检查磁盘容量是否改变，重设至 bdev->bd_inode，/fs/block_dev.c*/
    bdev->bd_invalidated = 0;                   /*设置分区信息有效*/
    if (!get_capacity(disk) || !(state = check_partition(disk, bdev)))      /*读取磁盘分区信息*/
        return 0;

    ...     /*错误处理*/

    kobject_uevent(&disk_to_dev(disk)->kobj, KOBJ_CHANGE);    /*触发 uevent 事件*/

    for (p = 1, highest = 0; p < state->limit; p++)    /*统计实际的分区数量保存至变量 highest*/
      if (state->parts[p].size)
          highest = p;

    disk_expand_part_tbl(disk, highest);      /*检查是否需要扩展 gendisk 分区表*/

    /*扫描 parsed_partitions 实例中缓存的分区信息，创建并添加分区 hd_struct 实例*/
    for (p = 1; p < state->limit; p++) {          /*从第二个分区信息开始*/
        sector_t size, from;
        struct partition_meta_info *info = NULL;
        size = state->parts[p].size;      /*分区大小，扇区数*/
        if (!size)
            continue;

        from = state->parts[p].from;      /*分区起始扇区号*/
        ...             /*分区大小出错处理*/

        if (state->parts[p].has_info)
            info = &state->parts[p].info;
        part = add_partition(disk, p, from, size,state->parts[p].flags,&state->parts[p].info);
                                    /*创建并添加 hd_struct 实例，/block/partition-generic.c*/
        ...    /*错误处理*/
      #ifdef CONFIG_BLK_DEV_MD
        if (state->parts[p].flags & ADDPART_FLAG_RAID)
            md_autodetect_dev(part_to_dev(part)->devt);
      #endif
    }   /*扫描缓存分区信息结束*/

    free_partitions(state);    /*释放 parsed_partitions 实例*/
    return 0;       /*成功返回 0*/
}
```

rescan_partitions()函数比较好理解，就是调用 check_partition(disk, bdev)函数读取磁盘中的分区信息，缓存起来，然后对每个缓存的分区信息调用 **add_partition()**函数创建并添加 hd_struct 实例。下面看一下 add_partition()函数的实现。

●**创建/添加分区**

add_partition()函数定义如下（/block/partition-generic.c）：

```
struct hd_struct *add_partition(struct gendisk *disk, int partno,sector_t start, sector_t len, int flags, \
                                                        struct partition_meta_info *info)
/*
*disk：磁盘 gendisk 实例指针，partno：从设备号，0 表示整个磁盘，1 表示第一个分区，
*start：起始扇区号，len：长度，扇区数量，flag：标记，info：分区元信息实例指针。
*/
{
    struct hd_struct *p;
    dev_t devt = MKDEV(0, 0);
    struct device *ddev = disk_to_dev(disk);      /*指向表示整个磁盘的 device 实例*/
    struct device *pdev;            /*表示分区的 device 实例指针*/
    struct disk_part_tbl *ptbl;
    const char *dname;
    int err;

    err = disk_expand_part_tbl(disk, partno);      /*如有需要则扩展磁盘分区表*/
    ...
    ptbl = disk->part_tbl;        /*分区表指针*/
    ...
    p = kzalloc(sizeof(*p), GFP_KERNEL);      /*分配 hd_struct 实例*/
    ...
    if (!init_part_stats(p)) {
        err = -ENOMEM;
        goto out_free;
    }

    seqcount_init(&p->nr_sects_seq);
    pdev = part_to_dev(p);            /*分区 device 实例指针，hd_struct.__dev*/

    p->start_sect = start;          /*起始扇区号*/
    p->alignment_offset =queue_limit_alignment_offset(&disk->queue->limits, start);
    p->discard_alignment =queue_limit_discard_alignment(&disk->queue->limits, start);
    p->nr_sects = len;              /*长度*/
    p->partno = partno;             /*从设备号（分区编号）*/
    p->policy = get_disk_ro(disk);      /*返回 disk->part0.policy*/

    if (info) {        /*如果有分区元信息，则复制到 hd_struct 实例*/
        struct partition_meta_info *pinfo = alloc_part_info(disk);
        if (!pinfo)
            goto out_free_stats;
        memcpy(pinfo, info, sizeof(*info));
```

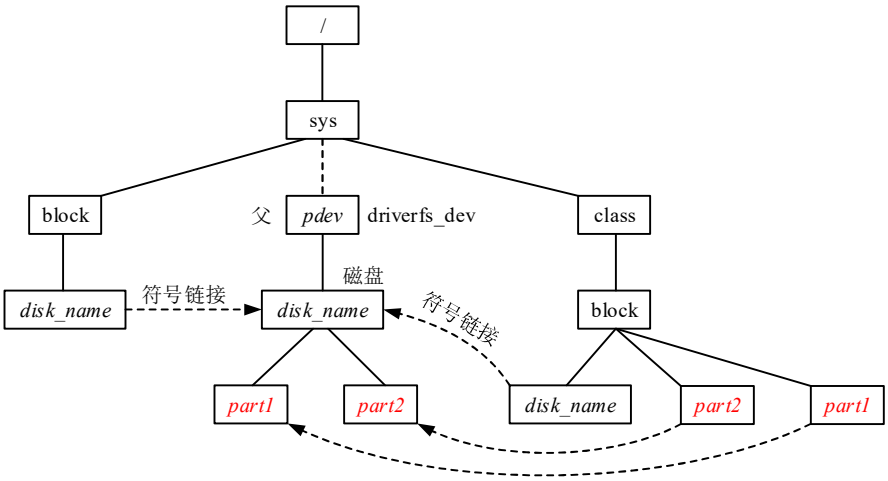
```
        p->info = pinfo;
}

dname = dev_name(ddev);        /*注意是表示整个磁盘的 device 实例的名称*/
if (isdigit(dname[strlen(dname) - 1]))
    dev_set_name(pdev, "%sp%d", dname, partno);
                /*磁盘名称最后为数字，则分区为磁盘名称后加 p 字符再加分区号*/
else
    dev_set_name(pdev, "%s%d", dname, partno);
                        /*磁盘名称最后不是数字，则分区名称为磁盘名称后加分区号*/

 device_initialize(pdev);        /*初始化分区 device 实例*/
 pdev->class = &block_class;        /*块设备类*/
 pdev->type = &part_type;     /*分区设备类型，注意不是磁盘设备类型，/block/partition-generic.c*/
 pdev->parent = ddev;          /*父 device 为表示磁盘的 device 实例（disk->part0.__dev）*/

 err = blk_alloc_devt(p, &devt);        /*为分区生成设备号*/
 ...
 pdev->devt = devt;          /*设备号赋予 device 实例*/

 dev_set_uevent_suppress(pdev, 1);
 err = device_add(pdev);        /*添加分区 device 实例，将自动创建块设备文件*/
 ...
 err = -ENOMEM;
 p->holder_dir = kobject_create_and_add("holders", &pdev->kobj);
                                        /*分区 device 实例下创建 holders 目录*/
 if (!p->holder_dir)
        goto out_del;

 dev_set_uevent_suppress(pdev, 0);
 if (flags & ADDPART_FLAG_WHOLEDISK) {
        err = device_create_file(pdev, &dev_attr_whole_disk);
        if (err)
            goto out_del;
 }
 rcu_assign_pointer(ptbl->part[partno], p);      /*分区表项指向 hd_struct 实例*/

 if (!dev_get_uevent_suppress(ddev))
        kobject_uevent(&pdev->kobj, KOBJ_ADD);      /*向用户空间发送 uevent 事件*/

 hd_ref_init(p);
 return p;          /*返回 hd_struct 实例指针*/
 ...
```

}

add_partition()函数比较好理解，主要是创建 hd_struct 实例，将参数提供的分区信息赋予 hd_struct 实例，并设置实例，最后添加实例。

分区 device 实例的父设备设为表示磁盘的 device 实例。如果磁盘的名称字符串最后一位为数字，则分区的名称为磁盘名称后加 p 字符再加分区号，例如：磁盘名称为 hd0，则第一个分区名称为 hd0p1。如果磁盘名称字符串最后一位不是数字，则分区名称为磁盘名称加分区号，例如：磁盘名称为 hd，分区 1 的名称为 hd1，分区名称将做为自动创建的块设备文件名称。

添加分区时，会在 sysfs 中表示磁盘的目录下，创建表示分区的子目录，子目录名称同分区设备文件名称，子目录下包含分区属性文件等。在/sys/class/block/目录下将创建到分区目录的符号链接，名称同分区设备文件名称，但是不会在/sys/block/目录下创建符号链接，如下图所示。



10.3 请求与请求队列

前面介绍了块设备驱动程序中分配和添加磁盘的接口函数（含添加分区），但是还有一个重要的步骤没有介绍。那就是为磁盘创建请求队列，请求队列需在添加磁盘前赋予 gendisk 实例。

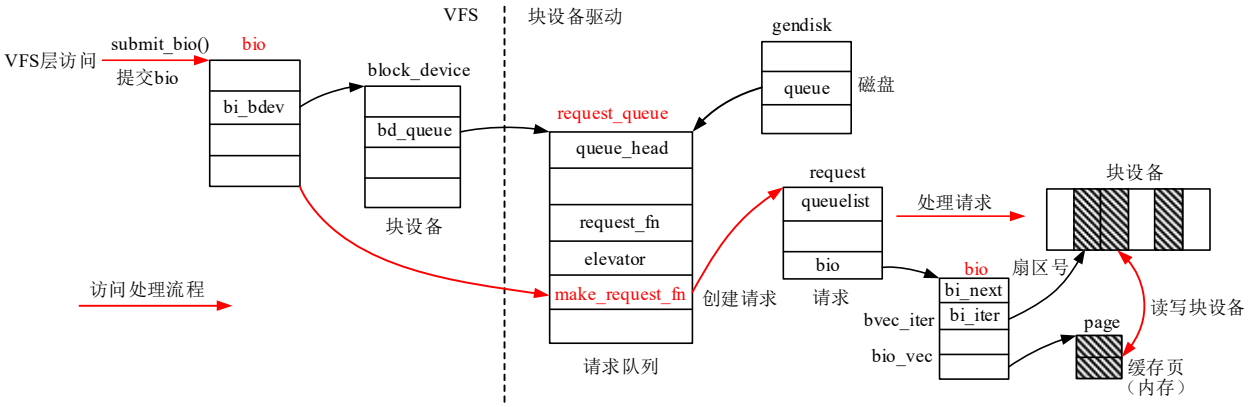
VFS 层将块设备读写访问等操作提交到请求队列，请求队列将其转换成请求，添加到请求队列，然后由块设备驱动程序执行请求中的数据传输或命令。

本节主要介绍请求和请求队列的定义，以及 VFS 层创建、提交访问操作的接口函数，后面两节将具体介绍请求队列的创建以及请求队列对请求的处理。

10.3.1 概述

1 框架

VFS 层对块设备的访问由 bio 结构体封装，提交给请求队列，依 bio 创建请求，插入到请求队列。驱动程序从队列中取下请求执行数据传输，如下图所示：



VFS 层对块设备的访问由 bio 结构体封装，其中包含访问块设备中数据块和保存数据内存块的信息，这里的访问可以是读写操作，也可以是发送控制命令。内核建立了管理 bio 实例的结构，提供了分配、提交 bio 的接口函数。VFS 层创建 bio 实例后，需关联表示被访问块设备的 block_device 实例，以便寻找到请求队列。

submit_bio(rw, bio)接口函数用于向请求队列提交访问操作，函数内通过 block_device 实例获取被访问块设备的请求队列，调用其中的 make_request_fn()函数处理 bio。make_request_fn()函数通常是先检查 bio 是否能与现有的请求合并，如果可以则合并；如果不可以合并，则为 bio 创建新请求，新请求添加到请求队列中的管理结构，并激活请求的处理。请求由 request 结构体表示，一个请求中可包含一个或多个 bio 实例。

块设备驱动程序需定义处理队列中请求的函数（或内核线程），其大致流程是从请求队列中取出请求，执行请求 bio 实例中指定的数据传输，最后执行请求处理后的工作。

请求队列由 request_queue 结构体表示，创建之后需赋予 gendisk 实例。在添加磁盘或打开块设备文件时，内核在 VFS 中将为每个磁盘和分区创建对应的 block_device 实例（表示块设备），实例由 bdev 伪文件系统管理。block_device 实例将关联对应磁盘的请求队列 request_queue 实例。block_device 实例的管理和创建本章后面将介绍。

内核支持的请求队列类型大致分为标准请求队列（单队列）、Multi queue 请求队列（多队列模型）和特殊请求队列等。

标准请求队列中只有一个由请求组成的链表，特殊请求队列用于基于内存盘的块设备，不需要构建请求，提交 bio 实例时直接进行访问操作。Multi queue 队列是为了缓解多核多进程系统中提交、获取请求行时机对队列保护锁的竞争，以及提高 SSD 磁盘的访问效率。

标准请求队列中所有请求在一个链表中，对队列进行操作时需持有队列保护锁，如果有多个 CPU 核或进程向队列提交请求，对锁的竞争将比较激烈，影响系统性能。

Multi queue 队列的目的就是为了缓存对队列锁的竞争，为每个 CPU 核定义了软件队列，还定义了硬件队列用于向驱动程序发送请求（处理请求）。VFS 提交 bio 时，若构建的请求不能提交到硬件队列，则提交到各自 CPU 核的软件队列（不需要与其它 CPU 核竞争锁），之后再移动到硬件队列进行处理。

不同类型的请求队列其对 bio 实例的处理函数 **make_request_fn()**各不相同，队列对请求的管理和处理流程也有所不同。

内核提供了创建各类型请求队列的接口函数，本节介绍请求、请求队列相关的数据结构，以及 bio 实例的管理和相关接口函数，后面两节将介绍请求队列的创建及其对请求的处理。

2 数据结构

这里介绍的数据结构有 VFS 中封装块设备访问操作的 bio 结构体、表示请求的 request 结构体，以及

表示请求队列的 request_queue 结构体。

■bio

bio 结构体封装了 VFS 层对块设备访问所需的信息，例如访问类型（读或写）、内存中保存数据的内存块、访问块设备的起始扇区号及长度等。一个 bio 实例表示对块设备中**某段连续数据块**的读/写操作或控制命令，保存数据的内存块可以是离散的。

bio 结构体定义在/include/linux/blk_types.h 头文件：

```
struct bio {
    struct bio              *bi_next;     /*指向同一 request 内下一个 bio 实例*/
    struct block_device     *bi_bdev;     /*块设备的 block_device 实例指针（见本章下文）*/
    unsigned long           bi_flags ;    /*标记成员，表示状态、命令等*/
    unsigned long           bi_rw;/*低 16 位取值同 request 结构体 cmd_flags 成员，高 16 位表示优先级*/
    struct bvec_iter        bi_iter;   /*bvec_iter 结构体，表示访问块设备信息，如起始扇区号、长度等*/

    unsigned int    bi_phys_segments;   /*传输数据的段数*/
    unsigned int    bi_seg_front_size;
    unsigned int    bi_seg_back_size;
    atomic_t    __bi_remaining;
    bio_end_io_t  *bi_end_io;       /*bio 处理完成后的回调函数*/
    void   *bi_private;             /*私有数据指针*/
#ifdef CONFIG_BLK_CGROUP
    struct io_context     *bi_ioc;
    struct cgroup_subsys_state *bi_css;
#endif
    union {
    #if defined(CONFIG_BLK_DEV_INTEGRITY)
      struct bio_integrity_payload *bi_integrity;   /*数据完整性结构体指针，/include/linux/bio.h*/
    #endif
    };

    unsigned short      bi_vcnt;            /*bi_io_vec 成员指向 bio_vec 数组项数*/
    unsigned short      bi_max_vecs;        /*最大 bio_vec 数组项数*/
    atomic_t            __bi_cnt;           /*pin count */

    struct bio_vec      *bi_io_vec;         /*指向 bio_vec 结构体数组，表示保存数据的内存块信息*/

    struct bio_set      *bi_pool;       /*指向分配 bio 和 bio_vec 实例的 bit_set 实例*/
    struct bio_vec      bi_inline_vecs[0];
                        /*bio_vec 结构体数组，bio_vec 结构体表示保存数据的内存块信息*/
};
```

bio 结构体主要成员简介如下：

●**bi_next：**指向同一请求 request 实例中的下一个 bio 实例。

●**bi_bdev**：指向访问块设备的 block_device 实例（块设备在 VFS 中的表示），用于查找请求队列，详见本章下文。

●**bi_iter**：bvec_iter 结构体成员，表示设备中被访问数据块信息，定义如下（/include/linux/blk_types.h）：

```
struct bvec_iter {
    sector_t        bi_sector;      /*读写块设备的起始扇区号（扇区 512 字节）*/
    unsigned int    bi_size;        /*数据长度，字节数*/
    unsigned int    bi_idx;         /*当前操作关联 bio_vec 实例的数组项索引*/
    unsigned int    bi_bvec_done;   /*当前 bio_vec 实例完成的字节数*/
};
```

●**bi_io_vec**：指向 bio_vec 结构体数组，bio_vec 结构体表示用于保存块设备中数据的内存信息，结构体定义在/include/linux/blk_types.h 头文件：

```
struct bio_vec {
    struct page     *bv_page;       /*保存数据的内存页 page 实例指针*/
    unsigned int    bv_len;         /*数据长度，字节数*/
    unsigned int    bv_offset;      /*页内的偏移量（起始位置）*/
};
```

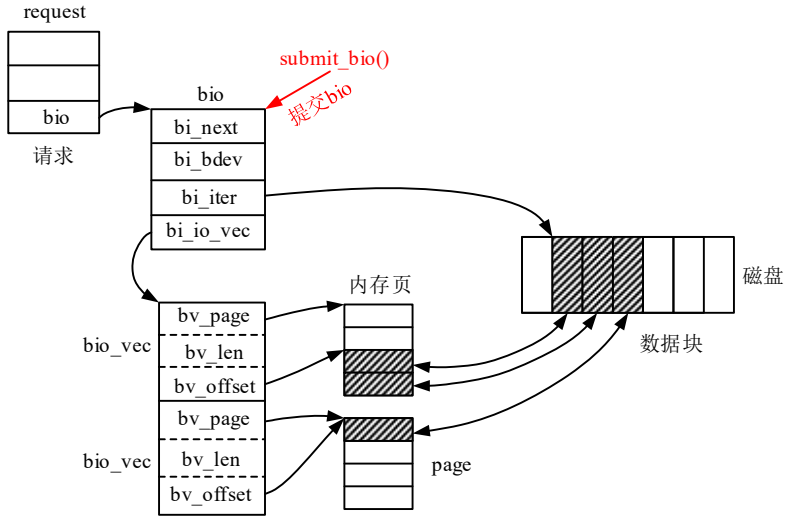
驱动程序通过由 page 实例可得出内存页的起始物理地址，加上页内偏移量 bv_offset 即可获得保存块设备数据的起始物理地址。

●**bi_inline_vecs[0]**：bio 结构体内嵌的 bio_vec 结构体数组。当 bio 实例所需的 bio_vec 实例较少时，在创建 bio 实例时，同时在其尾部附加 bio_vec 结构体数组，并令 bi_io_vec 成员指向 bi_inline_vecs[0]成员，而不必另行分配 bio_vec 数组，以提高效率。

●**bi_pool**：bio_set 结构体指针，bio_set 结构体管理着分配 bio 和 bio_vec 实例的内存池，在创建 bio 和 bio_vec 实例时，从内存池中分配实例，详见下文。

bvec_iter 结构体表示的是被访问块设备的数据块信息，是块设备中连续的数据块，bio_vec 结构体表示保存块设备数据的内存块信息。读写数据在块设备中必须是连续的，但是在内存中可以是离散保存的。因此，bio 结构体中只包含一个 bvec_iter 结构体成员，却包含一个 bio_vec 结构体数组。

bio 结构体中 bi_io_vec 和 bi_iter 成员关系如下图所示（阴影部分表示访问的数据块，一个内存页含四个数据块）：



●**bi_end_io**：bio_end_io_t 类型函数指针，bio 执行完的回调函数，函数原型定义如下：

typedef void (bio_end_io_t) (struct bio *, int);　　/*/include/linux/blk_types.h*/

●**bi_rw**：无符号整数，低 16 位取值同 request 结构体 cmd_flags 成员（见下文），高 16 位表示优先级。
●**bi_flags**：标记成员，各标记位定义如下（/include/linux/blk_types.h）：

```
#define   BIO_UPTODATE      0    /*I/O 数据操作完成，bio 指向的内存数据有效*/
#define   BIO_SEG_VALID     1    /*bi_phys_segments 有效*/
#define   BIO_CLONED        2    /*doesn't own data */
#define   BIO_BOUNCED       3    /*bio is a bounce bio */
#define   BIO_USER_MAPPED   4    /*包含用户页面*/
#define   BIO_NULL_MAPPED   5    /*包含无效用户页面*/
#define   BIO_QUIET         6    /*Make BIO Quiet */
#define   BIO_SNAP_STABLE   7    /*bio data must be snapshotted during write */
#define   BIO_CHAIN         8    /* chained bio, ->bi_remaining in effect */
#define   BIO_REFFED        9    /* bio has elevated ->bi_cnt */
#define   BIO_RESET_BITS    13
#define   BIO_OWNS_VEC      13   /* bio_free() should free bvec */

#define   bio_flagged(bio, flag)   ((bio)->bi_flags & (1 << (flag)))      /*检测 flag 位置标记位*/

/*bi_flags 成员高 4 位表示 bio_vec 实例数组来自 bvec_slabs[]数组哪个 slab 缓存，见下文*/
#define   BIO_POOL_BITS          (4)
#define   BIO_POOL_NONE          ((1UL << BIO_POOL_BITS) - 1)
#define   BIO_POOL_OFFSET        (BITS_PER_LONG - BIO_POOL_BITS)
#define   BIO_POOL_MASK          (1UL << BIO_POOL_OFFSET)
#define   BIO_POOL_IDX(bio)      ((bio)->bi_flags >> BIO_POOL_OFFSET) /*slab 缓存索引值*/
```

■请求

请求由 request 结构体表示，VFS 层提交的 bio 实例，可能合并到现有请求 request 实例中，也可能需为其创建新的请求，并添加到请求队列中。

request 结构体定义如下（/include/linux/blkdev.h）：

```
struct request {
    struct list_head    queuelist;     /*双链表元素，将实例添加到 request_queue.queue 双链表等*/
    union {
        struct call_single_data csd;
        unsigned long fifo_time;
    };

    struct request_queue    *q;          /*指向所属请求队列*/
    struct blk_mq_ctx    *mq_ctx;       /*指向软件队列*/

    u64    cmd_flags;          /*命令标记，/include/linux/blk_types.h*/
    unsigned    cmd_type;      /*命令类型，/include/linux/blkdev.h*/
```

```
    unsigned long    atomic_flags;        /*/block/blk.h*/
                        /*原子标记，如 REQ_ATOM_COMPLETE、REQ_ATOM_STARTED*/
    int cpu;
    unsigned int    __data_len;     /*请求中传输数据的总长度，不可直接访问*/
    sector_t    __sector;           /*读写操作起始扇区号*/

    struct bio    *bio;             /*bio 实例单链表*/
    struct bio    *biotail;         /*指向 bio 单链表最末尾成员*/

    /*IO 调度器可能建立了数据结构用于管理 request 实例，以下成员用于将请求加入管理结构*/
    union {                     /*用于 IO 调度器管理请求实例*/
        struct hlist_node    hash;    /*用于将请求添加到 IO 调度器的散列表*/
        struct list_head    ipi_list;
    };

    union {
        struct    rb_node rb_node;    /*红黑树节点成员，由 IO 调度器管理请求时使用*/
        void    *completion_data;
    };

    union {      /*联合体开始*/
        struct {                            /*用于 IO 调度器的指针*/
            struct io_cq          *icq;
            void              *priv[2];
        } elv;

        struct {    /*FLUSH/FUA 请求使用的成员*/
            unsigned int          seq;
            struct list_head      list;
            rq_end_io_fn          *saved_end_io;
        } flush;
    };       /*联合体结束*/

    struct gendisk    *rq_disk;     /*指向请求对应磁盘 gendisk 实例*/
    struct hd_struct    *part;      /*指向请求对应分区 hd_struct 实例*/
    unsigned long    start_time;

#ifdef CONFIG_BLK_CGROUP
    struct request_list    *rl;                    /* rl this rq is alloced from */
    unsigned long long    start_time_ns;
    unsigned long long    io_start_time_ns;       /* when passed to hardware */
#endif
```

```
    unsigned short    nr_phys_segments;       /*传输的数据段数*/

#if defined(CONFIG_BLK_DEV_INTEGRITY)
    unsigned short    nr_integrity_segments;
#endif

    unsigned short    ioprio;     /*IO 优先级*/
    void *special;        /* opaque pointer available for LLD use */
    int    tag;             /*请求标签值，整数*/
    int    errors;

    unsigned char __cmd[BLK_MAX_CDB];     /*传递的命令*/
    unsigned char    *cmd;
    unsigned short    md_len;

    unsigned int    extra_len;  /* length of alignment and padding */
    unsigned int    sense_len;
    unsigned int    resid_len;  /* residual count */
    void *sense;
    unsigned long    deadline;
    struct list_head    timeout_list;          /*添加队列 q->timeout_list 双链表*/
    unsigned int    timeout;
    int retries;

    rq_end_io_fn    *end_io;       /*请求执行完的回调函数指针，/include/linux/blkdev.h*/
    void    *end_io_data;          /*指向回调函数的参数*/

    struct request    *next_rq;     /*指向下一个请求，用于双向请求（bidi）*/
};
```

request 结构体主要成员简介如下：

●**bio**：bio 实例指针，指向 bio 实例单链表。

●**biotail**：指向 bio 实例单链表中最末尾实例。

●**end_io**：rq_end_io_fn 类型函数指针，请求处理完的回调函数，定义如下（/include/linux/blkdev.h）：

```
typedef void    (rq_end_io_fn)(struct request *, int);
```

●**end_io_data**：end_io()函数成员参数指针。

●**cmd_type**：请求的类型，由枚举类型 rq_cmd_type_bits 表示，定义如下（/include/linux/blkdev.h）：

```
enum rq_cmd_type_bits {
    REQ_TYPE_FS     = 1,        /*文件系统发送的请求，普通读写操作*/
    REQ_TYPE_BLOCK_PC,          /* scsi 命令  */
    REQ_TYPE_DRV_PRIV,          /*从此处开始往下是驱动程序定义的命令*/
};
```

●**__cmd[]**：传递的命令名称。

●**cmd_flags**：命令标记成员，成员各比特位语义由 rq_flag_bits 枚举类型表示，bio 结构体 bi_rw 成员低位使用了同样的标记位。

rq_flag_bits 枚举类型定义在/include/linux/blk_types.h 头文件：

```
enum rq_flag_bits {        /*标记位比特位位置*/
    /*普通标记*/
    __REQ_WRITE,                  /*cmd_flags 第 0 位（后面类推），0 表示读，1 表示写*/
    __REQ_FAILFAST_DEV,       /* no driver retries of device errors，bit1*/
    __REQ_FAILFAST_TRANSPORT, /* no driver retries of transport errors */
    __REQ_FAILFAST_DRIVER,        /* no driver retries of driver errors */

    __REQ_SYNC,           /*同步请求（读或写）*/
    __REQ_META,           /* metadata io request，元数据 IO 请求*/
    __REQ_PRIO,           /* boost priority in cfq */
    __REQ_DISCARD,        /* request to discard sectors */
    __REQ_SECURE,          /* secure discard (used with __REQ_DISCARD) */
    __REQ_WRITE_SAME,      /* write same block many times */

    __REQ_NOIDLE,         /* don't anticipate more IO after this one */
    __REQ_INTEGRITY,   /* I/O includes block integrity payload */
    __REQ_FUA,            /* forced unit access ，FUA 请求*/
    __REQ_FLUSH,          /* request for cache flush，FLUSH 请求，刷出缓存*/

    /* bio 结构体 bi_rw 成员专用标记位*/
    __REQ_RAHEAD,         /*预读操作*/
    __REQ_THROTTLED, /* This bio has already been subjected to throttling rules. Don't do it again. */
                          /*bit15*/
    /* request 结构体 cmd_flags 专用标记*/
    __REQ_SORTED,         /* elevator knows about this request，bit16*/
    __REQ_SOFTBARRIER,     /* may not be passed by ioscheduler */
    __REQ_NOMERGE,            /*不允许 bio 与本请求合并*/
    __REQ_STARTED,            /* drive already may have started this one */
    __REQ_DONTPREP,           /* don't call prep for this one */
    __REQ_QUEUED,         /* uses queueing */
    __REQ_ELVPRIV,        /*关联了 IO 调度器私有数据*/
    __REQ_FAILED,         /* set if the request failed */
    __REQ_QUIET,          /* don't worry about errors */
    __REQ_PREEMPT,            /* set for "ide_preempt" requests and also
                       for requests for which the SCSI "quiesce"
                       state must be ignored. */
    __REQ_ALLOCED,            /* request came from our alloc pool，标准队列分配的请求*/
    __REQ_COPY_USER,          /* contains copies of user pages */
    __REQ_FLUSH_SEQ,      /* request for flush sequence，FLUSH/FUA 请求的 FLUSH 步骤*/
    __REQ_IO_STAT,        /* account I/O stat ，bit29*/
```

```
    __REQ_MIXED_MERGE,    /* merge of different types, fail separately */
    __REQ_PM,             /* runtime pm request */
    __REQ_HASHED,         /* on IO scheduler merge hash */
    __REQ_MQ_INFLIGHT,    /* track inflight for MQ */
    __REQ_NO_TIMEOUT,     /* requests may never expire */
    __REQ_NR_BITS,        /* stops here */
};
#define   REQ_WRITE        (1ULL << __REQ_WRITE)          /*标记位标识*/
#define   REQ_FAILFAST_DEV       (1ULL << __REQ_FAILFAST_DEV)
...
```

■请求队列

请求队列由 request_queue 结构体表示，定义在/include/linux/blkdev.h 头文件，结构体中有些成员适用于所有类型队列，有些是标准请求队列专用的，有的只用于 Multi queue 请求队列：

```
struct request_queue {
    struct list_head      queue_head;          /*请求 request 实例双链表头*/
    struct request        *last_merge;         /*指向请求队列中首先可能合并的请求*/
    struct elevator_queue    *elevator;    /*IO 调度器（电梯算法，标准队列），/include/linux/elevator.h*/
    int                   nr_rqs[2];           /*队列中异步请求和同步请求的数量*/
    int                   nr_rqs_elvpriv;      /* # allocated rqs w/ elvpriv */

    struct request_list   root_rl;         /*request_list 实例，用于分配请求（标准队列）*/

    /*以下是函数指针成员*/
    request_fn_proc          *request_fn;     /*处理请求函数，标准队列，/include/linux/blkdev.h*/
    make_request_fn          *make_request_fn;    /*提交 bio 时调用，用于合并或创建请求*/
    prep_rq_fn               *prep_rq_fn;         /*请求预备函数，一般不使用*/
    unprep_rq_fn             *unprep_rq_fn;       /*拔出块设备时调用*/
    merge_bvec_fn            *merge_bvec_fn;      /*确定是否允许向一个现存的请求增加更多数据*/
    softirq_done_fn          *softirq_done_fn;    /*请求处理完，由 BLOCK_SOFTIRQ 软中断回调*/
    rq_timed_out_fn          *rq_timed_out_fn;
    dma_drain_needed_fn      *dma_drain_needed;
    lld_busy_fn           *lld_busy_fn;

    /*以下软、硬件队列相关成员是 Multi queue 队列专用的*/
    struct blk_mq_ops    *mq_ops;          /*多队列模型操作结构，/include/inux/blk-mq.h*/
    unsigned int          *mq_map;       /*软件队列与硬件队列的映射数组*/
    /*软件队列*/
    struct blk_mq_ctx    __percpu*queue_ctx;       /*软件队列），percpu 指针*/
    unsigned int          nr_queues;
    /*硬件发送队列*/
    struct blk_mq_hw_ctx    **queue_hw_ctx;
```

```
    unsigned int    nr_hw_queues;       /*硬件队列数量*/

    sector_t    end_sector;
    struct request  *boundary_rq;
    struct delayed_work    delay_work;   /*延时工作，用于处理请求*/

    struct backing_dev_info    backing_dev_info;
                                /*后备存储设备信息，用于数据回写，/include/linux/backing-dev-defs.h*/
    void            *queuedata;
    unsigned long       queue_flags;        /*请求队列标记*/
    int    id;                              /*由 ida 结构管理的请求队列 ID，用于标识队列*/
    gfp_t    bounce_gfp;
    spinlock_t      __queue_lock;      /*保护自旋锁*/
    spinlock_t      *queue_lock;       /*指向保护请求队列的自旋锁，标准队列*/

    struct kobject    kobj;             /*跟踪请求队列的 kobject 实例，导出到 sysfs*/
    struct kobject    mq_kobj;          /*跟踪 Multi queue 请求队列的 kobject 实例，导出到 sysfs*/

#ifdef CONFIG_PM
    struct device       *dev;
    int             rpm_status;
    unsigned int        nr_pending;
#endif
    /*请求队列参数*/
    unsigned long       nr_requests;            /*队列中最大请求数量*/
    unsigned int        nr_congestion_on;       /*请求队列拥塞阀值，请求数量大于该值表示拥塞*/
    unsigned int        nr_congestion_off;
    unsigned int        nr_batching;
    unsigned int        dma_drain_size;
    void               *dma_drain_buffer;
    unsigned int        dma_pad_mask;
    unsigned int        dma_alignment;
    struct blk_queue_tag    *queue_tags;        /*队列标签（标准请求队列），/include/linux/blkdev.h*/
    struct list_head    tag_busy_list;
    unsigned int        nr_sorted;
    unsigned int        in_flight[2];
    unsigned int        request_fn_active;
    unsigned int        rq_timeout;
    struct timer_list   timeout;
    struct list_head    timeout_list;
    struct list_head    icq_list;
#ifdef CONFIG_BLK_CGROUP
    DECLARE_BITMAP      (blkcg_pols, BLKCG_MAX_POLS);
```

```
        struct blkcg_gq         *root_blkg;
        struct list_head        blkg_list;
#endif
        struct queue_limits     limits;     /*请求队列限制参数*/
        unsigned int    sg_timeout;
        unsigned int    sg_reserved_size;
        int             node;
#ifdef CONFIG_BLK_DEV_IO_TRACE
        struct blk_trace        *blk_trace;
#endif

        unsigned int            flush_flags;    /*刷出设备缓存能力，如 REQ_FLUSH，是否支持 FLUSH/FUA*/
        unsigned int            flush_not_queueable:1;
        struct blk_flush_queue  *fq;    /*处理 FLUSH/FUA 请求的请求队列指针（标准队列），/block/blk.h*/

        struct list_head        requeue_list;
        spinlock_t              requeue_lock;
        struct work_struct      requeue_work;   /*工作将 queue_head 中请求移出，再插入队列（重新入队）*/
        struct mutex            sysfs_lock;
        int                     bypass_depth;
        atomic_t                mq_freeze_depth;

#if defined(CONFIG_BLK_DEV_BSG)
        bsg_job_fn              *bsg_job_fn;
        int                     bsg_job_size;
        struct bsg_class_device bsg_dev;
#endif

#ifdef CONFIG_BLK_DEV_THROTTLING        /*控制进程提交请求阀值*/
        struct throtl_data *td;
#endif
        struct rcu_head         rcu_head;
        wait_queue_head_t       mq_freeze_wq;
        struct percpu_ref       mq_usage_counter;
        struct list_head        all_q_node;         /*用于将实例添加到管理双链表，Multi queue 队列*/

        struct blk_mq_tag_set   *tag_set;
        struct list_head        tag_set_list;
};
```

request_queue 结构体主要成员简介如下：

●**queue_head**：双链表成员，链接请求队列中的请求 request 实例；

●**elevator**：elevator_queue 结构体指针，结构体定义在/include/linux/elevator.h 头文件内，表示 IO 调度器的实现，见下文（只用于标准请求队列）。

●**root_rl**：request_list 结构体成员，结构体定义在/include/linux/blkdev.h 头文件，用于分配请求实例等，此成员只用于标准请求队列：

```
struct request_list {
    struct request_queue    *q;         /*所属请求队列*/
  #ifdef  CONFIG_BLK_CGROUP
    struct blkcg_gq         *blkg;      /* blkg this request pool belongs to */
  #endif
    int         count[2];    /*请求队列中异步和同步请求的数量*/
    int         starved[2];
    mempool_t   *rq_pool;    /*分配请求 request 实例的内存池*/
    wait_queue_head_t wait[2];
    unsigned int    flags;   /*标记成员，BLK_RL_SYNCFULL 和 BLK_RL_ASYNCFULL 标记*/
};
```

●**函数指针**：内核在/include/linux/blkdev.h 头文件定义了请求队列结构中函数指针的原型，例如：

```
typedef   void (request_fn_proc) (struct request_queue *q);        /*只用于标准请求队列*/
                                                    /*块设备处理请求的函数，必须由具体驱动程序实现*/
typedef   void (make_request_fn) (struct request_queue *q, struct bio *bio);
            /*提交 bio 实例时调用此函数，与现有请求合并或创建新请求，并插入队列*/
```

●**fq**：blk_flush_queue 结构体指针，表示管理 FLUSH/FUA 请求的请求队列，定义在/block/blk.h 头文件（fq 成员只用于标准请求队列）：

```
struct blk_flush_queue {
    unsigned int        flush_queue_delayed:1;
    unsigned int        flush_pending_idx:1;
    unsigned int        flush_running_idx:1;
    unsigned long       flush_pending_since;
    struct list_head    flush_queue[2];         /*两个双链表*/
    struct list_head    flush_data_in_flight;
    struct request      *flush_rq;          /*指向请求实例*/
    struct request      *orig_rq;
    spinlock_t          mq_flush_lock;
};
```

●**backing_dev_info**：backing_dev_info 结构体成员（/include/linux/backing-dev-defs.h），用于 VFS 层控制对块设备的数据回写。在分配请求队列时，backing_dev_info 结构体成员将被初始化。backing_dev_info 结构体的定义和初始化详见第 11 章。

●**limits**：queue_limits 结构体成员，表示请求队列的限制参数，定义在/include/linux/blkdev.h 头文件：

```
struct queue_limits {
    unsigned long       bounce_pfn;
    unsigned long       seg_boundary_mask;

    unsigned int        max_hw_sectors;
    unsigned int        chunk_sectors;
```

```
    unsigned int        max_sectors;
    unsigned int        max_segment_size;
    unsigned int        physical_block_size;   /*物理块大小，初始化值为 512 字节*/
    unsigned int        alignment_offset;
    unsigned int        io_min;
    unsigned int        io_opt;
    unsigned int        max_discard_sectors;
    unsigned int        max_write_same_sectors;
    unsigned int        discard_granularity;
    unsigned int        discard_alignment;

    unsigned short      logical_block_size;   /*逻辑块大小限制值，初始化值为 512 字节*/
    unsigned short      max_segments;
    unsigned short      max_integrity_segments;

    unsigned char       misaligned;
    unsigned char       discard_misaligned;
    unsigned char       cluster;
    unsigned char       discard_zeroes_data;
    unsigned char       raid_partial_stripes_expensive;
};
```

在设置请求队列的blk_queue_make_request()函数中将调用**blk_set_default_limits()**函数对limits实例各成员进行初始化（/block/blk-settings.c）。其中io_min、physical_block_size、logical_block_size成员初始值都设为512字节。

在添加磁盘前，可调用blk_queue_physical_block_size()函数重置physical_block_size成员值，调用函数blk_queue_logical_block_size()重置logical_block_size成员值。这两个函数定义在/block/blk-settings.c文件内，代码如下：

```
void blk_queue_physical_block_size(struct request_queue *q, unsigned int size)
{
    q->limits.physical_block_size = size;     /*物理数据块大小*/

    if (q->limits.physical_block_size < q->limits.logical_block_size)
        q->limits.physical_block_size = q->limits.logical_block_size;   /*物理块不能比逻辑块小*/

    if (q->limits.io_min < q->limits.physical_block_size)      /*io_min 不能比物理块小*/
        q->limits.io_min = q->limits.physical_block_size;
}

void blk_queue_logical_block_size(struct request_queue *q, unsigned short size)
{
    q->limits.logical_block_size = size;      /*逻辑数据块大小*/

    if (q->limits.physical_block_size < size)     /*如果原物理块比 size 小，则重置为 size*/
```

```
        q->limits.physical_block_size = size;

    if (q->limits.io_min < q->limits.physical_block_size)        /*io_min 不能比物理块小*/
        q->limits.io_min = q->limits.physical_block_size;
}
```

由以上函数可知，io_min、physical_block_size、logical_block_size 三个成员的大小关系如下：

io_min≧physical_block_size≧logical_block_size

在/block/blk-settings.c 文件内还定义了其它设置请求队列 request_queue 成员值的接口函数，请读者自行查阅。

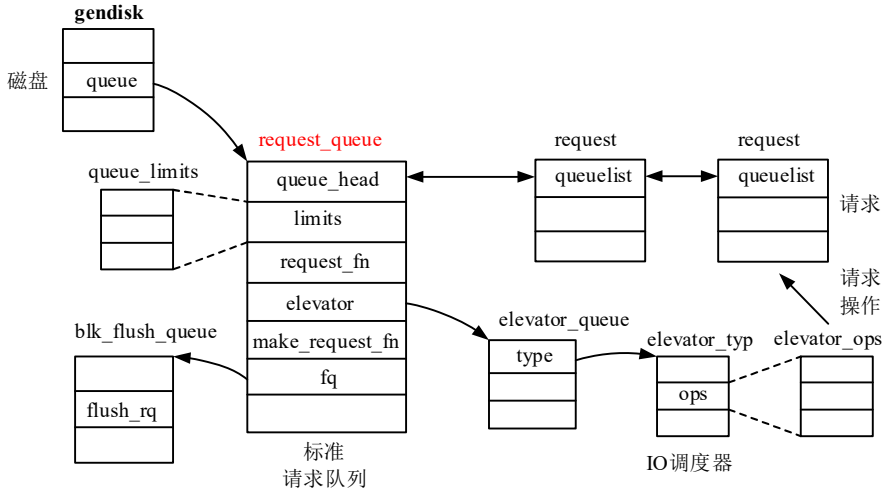
●**queue_flags**：请求队列标记，标记位语义如下（/include/linux/blkdev.h）：

```
#define   QUEUE_FLAG_QUEUED       1    /* uses generic tag queueing，使用通用队列标签*/
#define   QUEUE_FLAG_STOPPED      2    /*请求队列停止工作，不处理请求*/
#define   QUEUE_FLAG_SYNCFULL     3    /*读队列已经满了*/
#define   QUEUE_FLAG_ASYNCFULL    4    /*写队列已经满了*/
#define   QUEUE_FLAG_DYING        5    /* queue being torn down */
#define   QUEUE_FLAG_BYPASS       6    /*不使用 IO 调度器，简单的先入先出队列*/
#define   QUEUE_FLAG_BIDI         7    /* queue supports bidi requests，支持双向请求*/
#define   QUEUE_FLAG_NOMERGES     8   /*不能合并请求*/
#define   QUEUE_FLAG_SAME_COMP    9   /* complete on same CPU-group */
#define   QUEUE_FLAG_FAIL_IO      10   /* fake timeout */
#define   QUEUE_FLAG_STACKABLE    11  /* supports request stacking */
#define   QUEUE_FLAG_NONROT       12  /* non-rotational device (SSD) ，非机械磁盘*/
#define   QUEUE_FLAG_VIRT       QUEUE_FLAG_NONROT     /* paravirt device */
#define   QUEUE_FLAG_IO_STAT      13       /* do IO stats */
#define   QUEUE_FLAG_DISCARD      14       /* supports DISCARD */
#define   QUEUE_FLAG_NOXMERGES    15      / * No extended merges */
#define   QUEUE_FLAG_ADD_RANDOM   16       /* Contributes to random pool */
#define   QUEUE_FLAG_SECDISCARD   17       /* supports SECDISCARD */
#define   QUEUE_FLAG_SAME_FORCE   18     /* force complete on same CPU */
#define   QUEUE_FLAG_DEAD         19     /* queue tear-down finished，队列死了*/
#define   QUEUE_FLAG_INIT_DONE    20     /*队列初始化已经完成*/
#define   QUEUE_FLAG_NO_SG_MERGE  21      /* don't attempt to merge SG segments*/
#define   QUEUE_FLAG_SG_GAPS      22      /* queue doesn't support SG gaps */

#define   QUEUE_FLAG_DEFAULT  ((1 << QUEUE_FLAG_IO_STAT) |    \
                (1 << QUEUE_FLAG_STACKABLE)     |     \
                (1 << QUEUE_FLAG_SAME_COMP)     |     \
                (1 << QUEUE_FLAG_ADD_RANDOM))          /*标准请求队列默认标记*/

#define QUEUE_FLAG_MQ_DEFAULT    ((1 << QUEUE_FLAG_IO_STAT) |    \
                (1 << QUEUE_FLAG_STACKABLE)     |     \
```

(1 << QUEUE_FLAG_SAME_COMP))　　/*Multi queue 队列默认标记*/

在/include/linux/blkdev.h 头文件定义了测试、设置、清零 queue_flags 成员标记位的接口函数，例如：
void queue_flag_set(unsigned int flag, struct request_queue *q)：设置 queue_flags 中 flag 标记位。
void queue_flag_clear(unsigned int flag, struct request_queue *q)：清零 queue_flags 中 flag 标记位。

下图简要画出了标准请求队列的结构，Multi queue 队列结构后面再介绍：



10.3.2 VFS 层接口函数

VFS 层将块设备的访问封装成 bio 实例，提交至请求队列。内核建立了 bio 和 bio_vec 结构体的 slab 缓存，接口函数 bio_alloc()/bio_alloc_bioset()等用于分配 bio 实例（含 bio_vec 数组）。VFS 层设置 bio 实例后，最后需要调用 submit_bio(rw, bio)函数将 bio 实例提交到块设备请求队列，由队列进行处理。

1 bio 管理结构

请读者先看下图，了解内核对 bio 和 bio_vec 实例的管理。bio 实例需要若干个 bio_vec 实例来表示保存数据的内存信息。

bio_slabs 指向的是 bio_slab 结构体数组，称为 bio 结构体的缓存池。bio_slab 结构体中包含一个 slab 缓存，缓存对象是一个 bio 实例后附若干个 bio_vec 实例（默认是 4 个）。bio_slabs 指向数组关联的 slab 缓存其对象都是在 bio 实例后附 4 个 bio_vec 实例，不同之处是在 bio 实例之前附加的私有数据不同。这里的缓存是在创建 bio_set 实例时创建的，见下文。

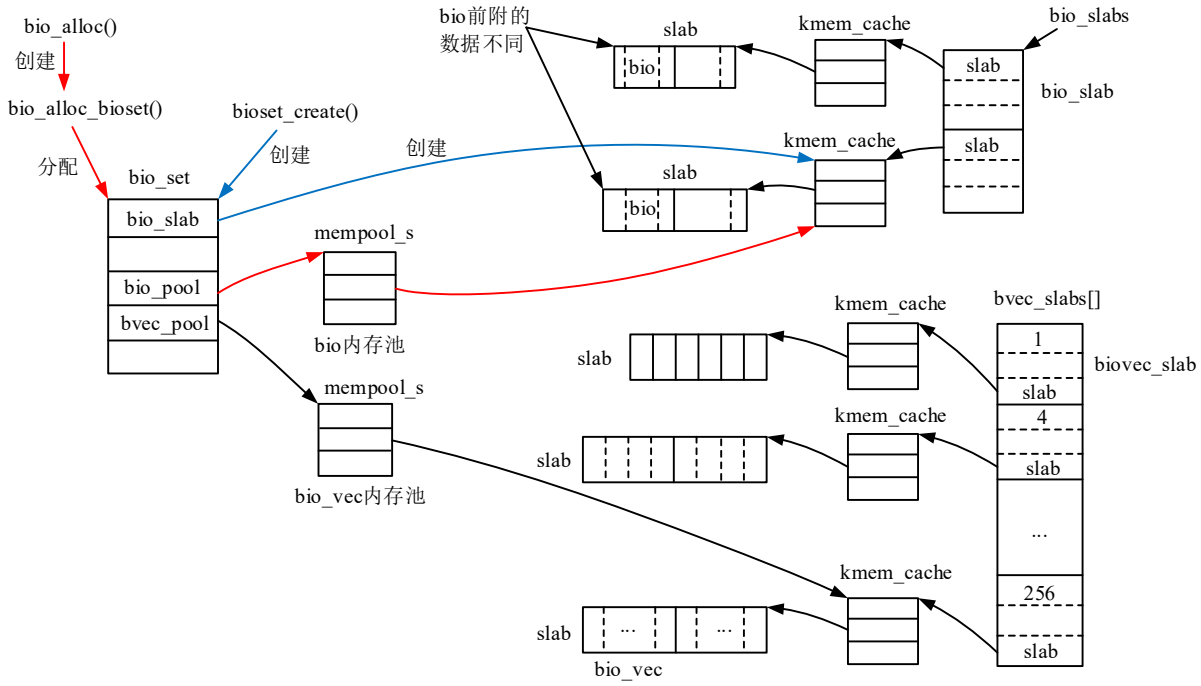
bvec_slabs[]是 biovec_slab 结构体数组，每个数组项包含一个 slab 缓存，缓存对象是若干个 bio_vec 实例（即 bio_vec 实例数组）。bvec_slabs[]数组项关联的 slab 缓存在内核初始化时创建。bvec_slabs[]数组称为 bio_vec 结构体的缓存池。

bio_set 结构体是分配 bio 实例的接口，结构体包含一个 bio 实例内存池，关联到 bio 结构体的 slab 缓存，即 bio_slabs 指向数组项关联的 slab 缓存。bio_set 结构体还包含一个 bio_vec 实例内存池，关联到一个 bio_vec 实例数组的 slab 缓存，即 bvec_slabs[]数组项关联的 slab 缓存。（内存池相关内容见第 3 章）

创建 bio_set 实例时，需要指定 bio 实例前附加的私有数据大小，在创建 bio_set 实例时会在 bio_slabs 指向数组中依附加私有数据大小，查找或创建 bio 实例的 slab 缓存（bio 后都附 4 个 bio_vec 实例）。

bio_slabs 指向数组大小是动态变化的，就是说创建的 bio_set 实例越多（如果 bio 前附加数据大小不同），bio_slabs 指向数组项也将越多（初始值为 2）。

bio_set 实例中 bio_vec 实例内存池默认关联到 bvec_slabs[]数组最末尾项的 slab 缓存。当通过 bio_set 实例分配 bio 实例时，如果所需的 bio_vec 实例不大于 4，则直接从关联的 bio 实例内存池中分配即可，如果 bio_vec 实例大于 4，则需要从 bvec_slabs[]数组关联的 slab 缓存中另行分配 bio_vec 实例数组并关联到 bio 实例。



bio_slab 结构体用于管理 bio 结构体的 slab 缓存，结构体定义在/block/bio.c 文件内：

```
struct bio_slab {                /*/block/bio.c*/
    struct kmem_cache *slab;      /*指向 slab 缓存结构*/
    unsigned int    slab_ref;     /*引用计数*/
    unsigned int    slab_size;    /*slab 中对象大小*/
    char name[8];
};
static struct bio_slab *bio_slabs;    /*指向 bio_slab 实例数组*/
```

biovec_slab 结构体用于管理 bio_vec 实例数组的 slab 缓存，结构体定义如下（/include/linux/bio.h）：

```
struct biovec_slab {
    int nr_vecs;              /*slab 对象中 bio_vec 实例数量（数组项数）*/
    char *name;               /*名称*/
    struct kmem_cache *slab;  /*指向 slab 缓存结构*/
};
```

内核在/block/bio.c 文件内静态定义了 biovec_slab 实例数组：

```
#define BV(x) { .nr_vecs = x, .name = "biovec-"__stringify(x) }
static struct biovec_slab bvec_slabs[BIOVEC_NR_POOLS] __read_mostly = {
    BV(1), BV(4), BV(16), BV(64), BV(128), BV(BIO_MAX_PAGES),
};
#undef BV
```

数组项数 BIOVEC_NR_POOLS 为 6，BIO_MAX_PAGES 宏取值为 256（/inclue/linux/bio.h）。

bvec_slabs[]数组项管理的 slab 缓存中，其对象分别是 1 个 bio_vec 实例，4 个 bio_vec 实例，16 个 bio_vec 实例，128 个 bio_vec 实例，256 个 bio_vec 实例。

bio 结构体中 bi_flags 成员的高 4 位表示实例关联的 bio_vec 实例数组来自哪个 bvec_slabs[]数组项中的 slab 缓存。

bio_set 结构体定义在/include/linux/bio.h 头文件：

```
#define   BIO_POOL_SIZE   2
#define   BIOVEC_NR_POOLS   6
#define   BIOVEC_MAX_IDX   (BIOVEC_NR_POOLS - 1)
struct bio_set {
      struct kmem_cache *bio_slab;        /*bio 结构体（尾接若干个 bio_vec 实例）slab 缓存*/
      unsigned int front_pad;             /*slab 对象中 bio 前附加字节数*/

      mempool_t *bio_pool;     /*bio 实例内存池，内存池从 bio_slab 指向 slab 缓存中分配对象*/
      mempool_t *bvec_pool;    /*bio_vec 实例内存池，从 biovec_slab[]数组中的 slab 缓存中分配对象*/
#if defined(CONFIG_BLK_DEV_INTEGRITY)
      mempool_t *bio_integrity_pool;      /*数据完整性所需数据结构的内存池*/
      mempool_t *bvec_integrity_pool;
#endif

      spinlock_t          rescue_lock;
      struct bio_list     rescue_list;
      struct work_struct  rescue_work;
      struct workqueue_struct *rescue_workqueue;        /*工作队列*/
};
```

■初始化

下面看一下内核对以上数据结构的初始化，初始化函数 init_bio()定义在/block/bio.c 文件内：

```
static int __init init_bio(void)
{
    bio_slab_max = 2;        /*全局变量，bio_slab 实例数组项数，/block/bio.c*/
    bio_slab_nr = 0;         /*全局变量，bio_slabs 指向数组实际使用的项数*/
    bio_slabs = kzalloc(bio_slab_max * sizeof(struct bio_slab), GFP_KERNEL);
                                        /*创建 2 个成员的 bio_slab 结构体数组，初始值*/
    if (!bio_slabs)
            panic("bio: can't allocate bios\n");

    bio_integrity_init();     /*创建数据完整性数据结构 slab 缓存，创建工作队列，/block/bio-integrity.c*/
    biovec_init_slabs();      /*初始化 bvec_slabs[]数组，创建对应 slab 缓存，/block/bio.c*/

    fs_bio_set = bioset_create(BIO_POOL_SIZE, 0);        /*fs_bio_set 为全局指针*/
```

```
                /*创建 bio_set 实例，缓存 bio 实例前附数据为 0，BIO_POOL_SIZE=2，/block/bio.c*/
    if (!fs_bio_set)
            panic("bio: can't allocate bios\n");

    if (bioset_integrity_create(fs_bio_set, BIO_POOL_SIZE))
                    /*为 fs_bio_set 实例创建数据完整性数据结构的内存池，/block/bio-integrity.c*/
        panic("bio: can't create integrity pool\n");
    return 0;
}
subsys_initcall(init_bio);     /*内核初始化子系统时调用*/
```

init_bio()函数完成的主要工作如下：

（1）创建初始的 bio_slab 结构体数组，数组项数为 2，这里还没有创建 slab 缓存。将在 bioset_create()函数中为其创建 slab 缓存，见下文。

（2）为 bvec_slabs[]数组创建对应的 slab 缓存。

（3）创建 bio_set 实例 **fs_bio_set**，用于分配前面不附加私有数据的 bio 实例，见下面的分配函数。

（4）创建数据完整性数据结构所需的 slab 缓存，为 fs_bio_set 创建数据完整性所需的内存池。

■创建 **bio_set 实例**

bioset_create()函数用于创建一个 bio_set 实例，它是用于分配 bio 实例的桥梁，将作为 bio_alloc_bioset()分配函数的参数，不过也可以不通过 bio_set 实例分配。

初始化函数创建的 fs_bio_set 实例用于分配不附加私有数据的 bio 实例，如果要分配带私有数据的 bio 实例，可以为其创建 bio_set 实例，然后通过这个实例来分配。

bioset_create()函数定义在/block/bio.c 文件内：

```
struct bio_set *bioset_create(unsigned int pool_size, unsigned int front_pad)
/*pool_size=2：内存池中暂存对象的数量，front_pad：slab 对象中 bio 实例前附私有数据的字节数*/
{
    return    __bioset_create(pool_size, front_pad, true);     /*/block/bio.c*/
}

static struct bio_set *__bioset_create(unsigned int pool_size,unsigned int front_pad,bool create_bvec_pool)
/*create_bvec_pool：是否为 bio_vec 实例在 bio_set 实例中创建内存池，这里为 true*/
{
    unsigned int back_pad = BIO_INLINE_VECS * sizeof(struct bio_vec);
                    /*BIO_INLINE_VECS=4，slab 中对象在 bio 实例后附 4 个 bio_vec 实例*/
    struct bio_set *bs;

    bs = kzalloc(sizeof(*bs), GFP_KERNEL);     /*分配 bio_set 实例并清零*/
    ...
    bs->front_pad = front_pad;        /*io 实例前附数据字节数*/

    spin_lock_init(&bs->rescue_lock);
```

```
    bio_list_init(&bs->rescue_list);
    INIT_WORK(&bs->rescue_work, bio_alloc_rescue);

    bs->bio_slab = bio_find_or_create_slab(front_pad + back_pad);
                    /*在 bio_slabs 指向数组项中创建或查找 slab 缓存，可能要扩展数组，/block/bio.c*/
    ...

    bs->bio_pool = mempool_create_slab_pool(pool_size, bs->bio_slab);
                        /*创建 bio 实例内存池，内存池关联 bs->bio_slab  指向 slab 缓存*/
    ...

    if (create_bvec_pool) {        /*是否为 bio_vec 实例创建内存池*/
        bs->bvec_pool = biovec_create_pool(pool_size);
                /*创建 bio_vec 实例内存池，初始关联 bvec_slabs[]数组最后一项中的 slab 缓存*/
        if (!bs->bvec_pool)
            goto bad;
    }

    bs->rescue_workqueue = alloc_workqueue("bioset", WQ_MEM_RECLAIM, 0);       /*工作队列*/
    if (!bs->rescue_workqueue)
        goto bad;

    return bs;      /*返回创建 bio_set 实例指针*/
bad:
    bioset_free(bs);
    return NULL;
}
```

__bioset_create()函数将创建 bio_set 实例，为其创建 bio 实例内存池，内存池关联到 bio_slabs 指向数组项中的 slab 缓存（不存在就创建）；调用 biovec_create_pool()函数为 bio_vec 实例创建内存池，内存池初始关联 bvec_slabs[]数组最后一项指示的 slab 缓存。

biovec_create_pool()函数定义如下：

```
mempool_t *biovec_create_pool(int pool_entries)
{
    struct biovec_slab *bp = bvec_slabs + BIOVEC_MAX_IDX;   /*bvec_slabs[]数组最后一项*/

    return mempool_create_slab_pool(pool_entries, bp->slab);      /*创建内存池*/
}
```

__bioset_create()函数最后将为 bio_set 实例创建工作队列，返回 bio_set 实例指针。

另外，struct bio_set ***bioset_create_nobvec**(unsigned int, unsigned int)函数用于创建不带 bio_vec 实例内存池的 bio_set 实例，源代码请读者自行阅读。

2 分配 bio

bio_alloc()函数用于分配不带前附私有数据的 bio 实例。

bio_alloc_bioset()函数用于分配带前附私有数据的 bio 实例，不过调用此函数前需创建对应的 bio_set 实例。

bio_alloc()函数定义在/include/linux/bio.h 头文件：

```
static inline struct bio *bio_alloc(gfp_t gfp_mask, unsigned int nr_iovecs)
/*gfp_mask：物理内存分配掩码，nr_iovecs：bio 实例所需 bio_vec 实例的数量*/
{
    return bio_alloc_bioset(gfp_mask, nr_iovecs, fs_bio_set);
                    /*从 fs_bio_set 实例分配 bio 实例（不带私有数据），/block/bio.c*/
}
```

bio_alloc_bioset()函数定义在/block/bio.c 文件内，用于从指定 bio_set 实例中分配 bio 实例：

```
struct bio *bio_alloc_bioset(gfp_t gfp_mask, int nr_iovecs, struct bio_set *bs)
{
    gfp_t saved_gfp = gfp_mask;
    unsigned front_pad;
    unsigned inline_vecs;        /*分配 bio 实例中内嵌的 bio_vec 数组项数*/
    unsigned long idx = BIO_POOL_NONE;    /*bi_flags 标记中高 4 位掩码*/
    struct bio_vec *bvl = NULL;
    struct bio *bio;
    void *p;

    if (!bs) {      /*参数没有指定 bio_set 实例，则从通用缓存中分配 bio 实例及 bio_vec 实例数组*/
        if (nr_iovecs > UIO_MAXIOV)
            return NULL;

        p = kmalloc(sizeof(struct bio) +nr_iovecs * sizeof(struct bio_vec),gfp_mask);
                                        /*为 bio 及 bio_vec 实例分配内存空间*/
        front_pad = 0;
        inline_vecs = nr_iovecs;
    }
    else {                  /*从 bio_set 实例中分配 bio 实例及 bio_vec 实例数组*/
        if (WARN_ON_ONCE(!bs->bvec_pool && nr_iovecs > 0))
            return NULL;

        if (current->bio_list && !bio_list_empty(current->bio_list))      /*进程 bio_list 链表不为空*/
            gfp_mask &= ~__GFP_WAIT;        /*内存分配路径不能进入睡眠*/

        p = mempool_alloc(bs->bio_pool, gfp_mask);      /*从 bs 指示的 bio 内存池中分配 bio 实例*/
        ...
```

```
        front_pad = bs->front_pad;
        inline_vecs = BIO_INLINE_VECS;
                        /*分配的 bio 实例中内嵌 4 个 bio_vec 实例，/block/bio.c*/
    }   /*分配 bio 实例结束*/

    if (unlikely(!p))
        return NULL;

    bio = p + front_pad;        /*bio 实例指针*/
    bio_init(bio);    /*bio 实例初始化，初始化少数成员，/block/bio.c*/

    if (nr_iovecs > inline_vecs) {      /*如果所需的 bio_vec 实例数量比 bio 实例中内嵌的数量多*/
        bvl = bvec_alloc(gfp_mask, nr_iovecs, &idx, bs->bvec_pool);      /*/block/bio.c*/
        /*从 bvec_slabs[]指定的合适的 slab 缓存中重新分配 bio_vec 实例数组，idx 保存数组项索引*/
        ...
        bio->bi_flags |= 1 << BIO_OWNS_VEC;      /*bio 实例具有 bio_vec 数组实例*/
    } else if (nr_iovecs) {       /*使用 bio 中内嵌的 bio_vec 数组实例，包括从通用缓存分配的情况*/
        bvl = bio->bi_inline_vecs;
    }

    bio->bi_pool = bs;        /*指向 bit_set 实例*/
    bio->bi_flags |= idx << BIO_POOL_OFFSET;
                        /*高 4 位保存 bio_vec 实例数组来 bvec_slabs[]数组中哪一项的 slab 缓存*/
    bio->bi_max_vecs = nr_iovecs;
    bio->bi_io_vec = bvl;          /*指向 bio_vec 实例数组*/
    return bio;       /*返回 bio 实例指针*/
    ...
}
```

bio_alloc_bioset()函数参数若指定了分配 bio 实例的 bio_set 实例，则从中分配 bio 和 bio_vec 实例数组，若 bio 中内嵌的 bio_vec 数组够用，则无需另外分配 bio_vec 数组，不够用则选择 bvec_slabs[]数组中合适的 slab 缓存，并从中分配 bio_vec 实例数组，最后对 bio 实例成员进行初始化。若函数中没有指定从哪个 bio_set 实例中分配，则直接从内核通用缓存中分配 bio 和 bio_vec 数组实例。

另外，内核还定义了其它几个 bio 操作函数，简介如下（/include/linux/bio.h）：

●struct bio ***bio_kmalloc**(gfp_t gfp_mask, unsigned int nr_iovecs)：直接从通用缓存中分配 bio 和 bio_vec 实例数组。

●struct bio ***bio_clone**(struct bio *bio, gfp_t gfp_mask)：克隆 bio 实例，新实例与旧实例共用 bio_vec 实例数组。

●struct bio ***bio_clone_kmalloc**(struct bio *bio, gfp_t gfp_mask)：克隆 bio 实例，新实例与旧实例共用 bio_vec 实例数组，新实例从通用缓存中分配。

3 提交 bio

VFS 层在分配 bio 实例后，还需要对其进行设置，如设置关联的 block_device 实例（以找到块设备请

求队列），设置 bvec_iter 结构体成员，设置各 bio_vec 实例数组项等，最后调用 **submit_bio()**函数将 bio 实例提交到块设备请求队列。

submit_bio()函数定义在/block/blk-core.c 文件内，代码如下：

```
void submit_bio(int rw, struct bio *bio)
/*rw：指示是读、写或预读操作，bio：bio 实例指针*/
{
    bio->bi_rw |= rw;                    /*0 表示读操作，1 表示写操作*/
    if (bio_has_data(bio)) {      /*bio->bi_iter 成员中指示了读写的数据长度*/
                    /*bio 是否包含有效数据信息，是则进行统计操作，/include/linux/bio.h*/
        unsigned int count;

        if (unlikely(rw & REQ_WRITE_SAME))
            count = bdev_logical_block_size(bio->bi_bdev) >> 9;   /*文件系统数据块中包含的扇区数*/
        else
            count = bio_sectors(bio);     /*bio->bi_iter.bi_size >> 9，/include/linux/bio.h*/
                                          /*字节数转扇区数*/
        if (rw & WRITE) {
            count_vm_events(PGPGOUT, count);
        } else {
            task_io_account_read(bio->bi_iter.bi_size);
            count_vm_events(PGPGIN, count);
        }

        if (unlikely(block_dump)) {
            ...          /*输出信息*/
        }
    }
    generic_make_request(bio);          /*调用通用的创建请求函数，/block/blk-core.c*/
}
```

submit_bio()函数在判断 bio 实例中数据的有效性后，进行常规的统计操作，然后调用通用的创建请求函数 generic_make_request(bio)，函数内将对本次提交的 bio 实例进行处理。对于每个 bio 实例将调用请求队列的 make_request_fn()函数进行处理。

generic_make_request()函数在有些块设备驱动程序中可能会被递归调用，也就是说进程在提交当前 bio 实例（处理请求）的过程中可能还会再次触发另一个 bio 实例的提交。因此，内核为进程创建了提交 bio 实例的缓存单链表，用于缓存在处理当前 bio 时再次提交的 bio 实例。

进程 task_struct 结构体中 bio_list 成员指向的链表用于缓存进程提交的 bio 实例，结构体成员定义如下：

```
task_struct{
    ...
    struct bio_list *bio_list;   /*bio 实例单链表，由 bi_next 成员构成单链表*/
    ...
}
```

bio_list 成员指向 bio_list 结构体实例，表示 bio 实例单链表，结构体定义在/include/linux/bio.h 头文件：

```
struct bio_list {
    struct bio    *head;      /*指向 bio 单链表第一个成员，由 bi_next 成员构成单链表*/
    struct bio    *tail;      /*指向 bio 单链表最后一个成员*/
};
```

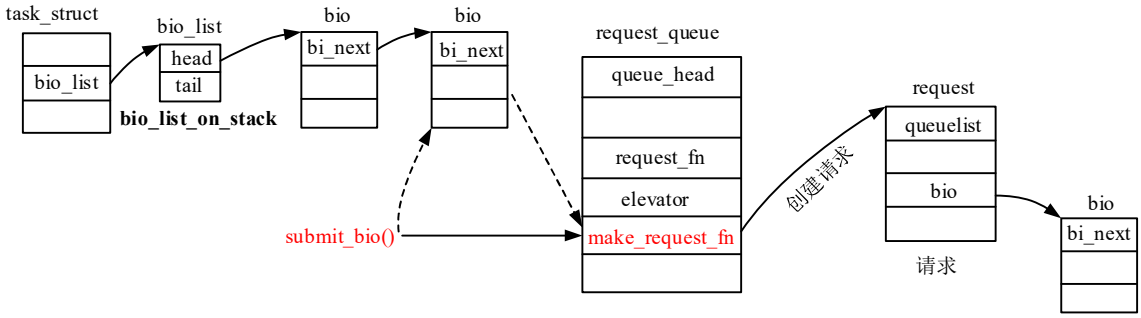
bio_list 链表用于缓存正在处理 bio 时又提交进来的 bio，第一个提交的 bio 立即处理，处理过程中提交的 bio 缓存到 bio_list 链表，第一个 bio 处理完毕后再处理 bio_list 链表中的 bio，直至链表中的 bio 处理完毕。

generic_make_request()函数定义如下（/block/blk-core.c）：

```
void generic_make_request(struct bio *bio)
{
    struct bio_list    bio_list_on_stack;      /*bio_list 结构体实例，局部变量，用于缓存之后提交的 bio*/

    if (!generic_make_request_checks(bio))     /*检查 bio 有效性，/block/blk-core.c*/
        return;
    if (current->bio_list) {     /*进程 bio_list 实例指针不为 NULL（正在提交/处理 bio 实例）*/
        bio_list_add(current->bio_list, bio);   /*插入 bio 链表末尾，/include/linux/bio.h*/
        return;          /*函数返回*/
    }

    /*以下是进程 bio_list 实例指针为 NULL 的情形*/
    BUG_ON(bio->bi_next);                /*只能提交一个 bio 实例*/
    bio_list_init(&bio_list_on_stack); /*初始化局部变量 bio_list_on_stack 实例，两个成员指向 NULL*/
    current->bio_list = &bio_list_on_stack;        /*对 current->bio_list 成员赋值*/
    do {
        struct request_queue *q = bdev_get_queue(bio->bi_bdev);   /*获取块设备请求队列*/
        q->make_request_fn(q, bio);
                    /*调用请求队列中的 make_request_fn()函数处理 bio 实例*/
        bio = bio_list_pop(current->bio_list);        /*处理完 bio 后，处理进程缓存的 bio*/
              /*获取 current->bio_list 链表中下一个 bio 实例，进行处理，/include/linux/bio.h*/
    } while (bio);          /*current->bio_list 链表为空，循环跳出*/
    current->bio_list = NULL;       /*bio_list 成员重置为 NULL，本次提交 bio 操作完成*/
}
```

generic_make_request()函数内检查进程 current->bio_list 成员是否为 NULL，如果不为 NULL 则将 bio 实例插入到 current->bio_list 链表末尾，因为当前正在处理其它 bio，函数返回。

如果 current->bio_list 成员为 NULL，则对 current->bio_list 成员赋值，用于缓存随后提交的 bio，对当前提交的 bio 实例（不插入到 current->bio_list 链表）调用 q->**make_request_fn**(q, bio)函数由请求队列处理，处理完成后，然后从 current->bio_list 链表取下一个 bio 实例进行同样的处理，直至 bio_list 链表为空，最后将 current->bio_list 重置为 NULL（如下图所示）。



由函数可知最早提交的 bio 实例将会立即进行处理，而嵌套提交的 bio 实例将插入到 current->bio_list 链表末尾，链表中 bio 实例将按提交的先后顺序进行处理，对每个 bio 实例调用 q->make_request_fn(q, bio)函数交由请求队列处理。generic_make_request()函数返回时，所有嵌套提交的 bio 实例都已经处理完毕，进程 current->bio_list 成员重置为 NULL。

不同类型请求队列其处理提交 bio 实例的 make_request_fn(q, bio)函数不同，后面介绍具体请求队列时再详细介绍。

10.4 标准请求队列

块设备驱动程序中最重要、最复杂的部分就是请求队列。请求队列按类型分为标准请求队列（单队列）、Multi queue 队列（多队列模型）和特殊请求队列等。这里说的特请求队列是指基于内存盘的块设备，数据访问不需要排队，可以直接进行。

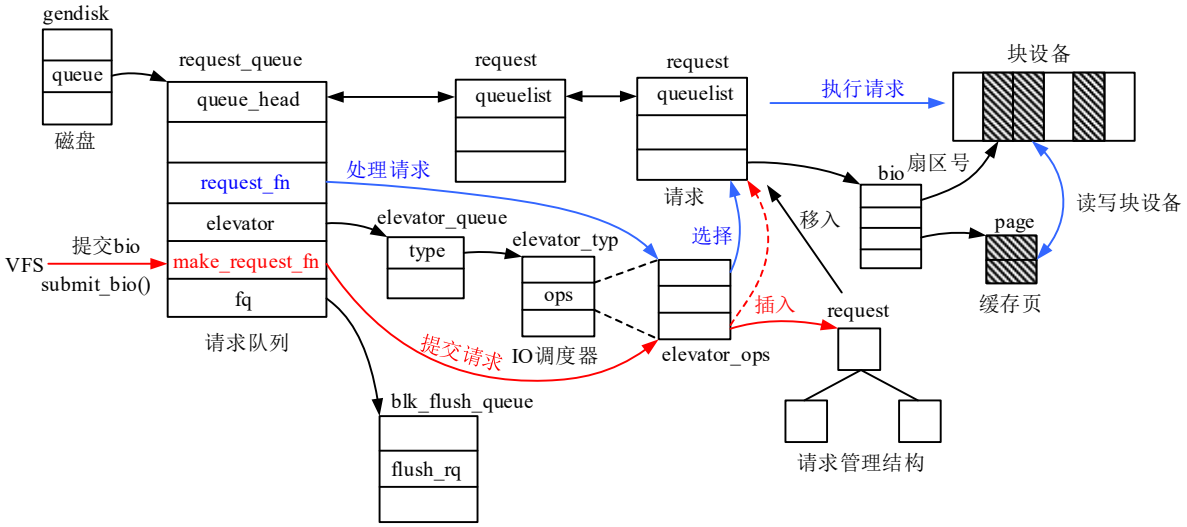
目前，多数块设备驱动程序使用的是标准请求队列，标准请求队列中只有一个由请求组成的队列。也有几个块设备驱动程序使用 Multi queue 请求队列。

本节主要介绍标准请求队列的创建以及对 bio 的处理，并简单介绍一下内存盘块设备请求队列的创建，下一节将介绍 Multi queue 请求队列。

10.4.1 概述

1 队列结构

标准请求队列结构简列如下图所示：



请求队列由 request_queue 结构体表示，请求由 request 结构体表示，VFS 层的数据传输（或命令）由 bio 结构体表示。VFS 提交的 bio 由请求队列的 make_request_fn()函数处理，请求队列将为 bio 创建请求。

标准请求队列绑定一个 IO 调度器，IO 调度器由 elevator_type 结构体表示，其中主要包含 bio 和 request 实例的操作函数，用于管理、操作、调度请求队列中的请求。IO 调度器通常也通过某种数据结构，如红黑树，来管理请求。

request_queue 中 queue_head 双链表按优先顺序排列了要处理的请求，新插入的请求一般先插入 IO 调度器的管理结构中。处理请求时从 queue_head 双链表头部开始依次取请求处理，链表为空时，则从 IO 调度器管理的请求中移入请求到 queue_head 双链表，再从中取请求进行处理。新插入的请求也可以直接插入请求队列中 queue_head 双链表。

VFS 层将块设备的访问封装成 bio 实例，接口函数 submit_bio(int rw, struct bio *bio)用于将 bio 实例提交到请求队列，函数内将调用请求队列的 make_request_fn()函数，用于为 bio 创建并添加（或合并）请求，并激活请求的处理。

标准请求队列中 make_request_fn()函数被设为 blk_queue_bio()，此函数先检查 bio 实例能否与现有请求合并，能合并则合并，不能合并则为 bio 实例创建新的请求 request 实例，并调用 IO 调度器中的函数将请求插入到请求队列中双链表或 IO 调度器中请求管理结构。最后，调用请求队列中的 **request_fn()**函数，激活块设备驱动程序对请求的处理。request_fn()函数必须由块设备驱动程序实现，用于处理队列中的请求。

request_fn()函数通常是唤醒一个驱动程序创建的内核线程，内核线程执行的大致流程是调用 IO 调度器中的函数，从队列中取出一个最高优先级的请求，执行其中 bio 实例表示的数据传输，执行请求（bio）完成后的工作，然后再取出下一个请求，如此循环。

struct request_queue ***blk_init_queue**(request_fn_proc ***rfn**, spinlock_t *lock)接口函数是创建标准请求队列的接口函数，rfn 是块设备驱动定义的请求队列 request_fn()函数指针，lock 是保护请求队列的自旋锁，函数返回请求队列 request_queue 实例指针。

本节主要介绍创建标准请求队列接口函数的实现，以及提交 bio 的处理流程，最后简要介绍一下基于内存盘的块设备请求队列的创建。

2 IO 调度器

IO 调度器用于管理和调度标准请求队列中的请求（或 bio），类似于进程调度中的调度类。VFS 层提交的 bio 实例由调度器处理，判断其能否与现有请求合并或需要创建新请求。IO 调度器以某种结构管理着请求队列中的请求，并调度请求的执行。

request_queue 结构体 elevator 成员指向的 elevator_queue 结构体用于管理队列绑定的 IO 调度器，结构体定义在/include/linux/elevator.h 头文件：

```
struct elevator_queue
{
    struct elevator_type *type;   /*指向 IO 调度器（电梯算法）*/
    void *elevator_data;          /*调度器数据*/
    struct kobject kobj;          /*跟踪绑定 IO 调度器的 kobject 实例*/
    struct mutex sysfs_lock;
    unsigned int registered:1;
    DECLARE_HASHTABLE(hash, ELV_HASH_BITS);    /*IO 调度器管理请求的散列表*/
};
```

type 成员指向 elevator_type 结构体表示 IO 调度器，结构体定义在/include/linux/elevator.h 头文件：

```
struct elevator_type
{
    struct kmem_cache *icq_cache;      /*调度器所需 icq 数据结构的 slab 缓存*/
```

```
    struct elevator_ops ops;        /*内嵌 elevator_ops 结构体，包含调度器各种操作函数指针*/
    size_t icq_size;                /*icq 数据结构大小*/
    size_t icq_align;               /*icq 数据结构对齐要求*/
    struct elv_fs_entry *elevator_attrs;        /*IO 调度器属性，导出到 sysfs*/
    char elevator_name[ELV_NAME_MAX];    /*IO 调度器名称*/
    struct module *elevator_owner;

    char icq_cache_name[ELV_NAME_MAX + 5];
                                    /* icq 数据结构的 slab 缓存名称：elvname + "_io_cq" */
    struct list_head list;          /*双链表成员，将 elevator_type 实例添加到全局链表*/
};
```

elevator_type 结构体中比较重要的成员是 elevator_ops 结构体成员，它是一组函数指针，用于实现对请求和 bio 的各种操作。

elevator_ops 结构体定义如下（/include/linux/elevator.h，函数原型也定义在同一头文件）：

```
struct elevator_ops
{
    elevator_merge_fn *elevator_merge_fn;      /*bio 能否合并到指定请求*/
    elevator_merged_fn *elevator_merged_fn;    /*请求合并后回调*/
    elevator_merge_req_fn *elevator_merge_req_fn;   /**/
    elevator_allow_merge_fn *elevator_allow_merge_fn;
    elevator_bio_merged_fn *elevator_bio_merged_fn;        /*bio 合并到指定请求*/

    elevator_dispatch_fn *elevator_dispatch_fn;      /*从 IO 调度器管理请求移入请求队列双链表*/
    elevator_add_req_fn *elevator_add_req_fn;      /*向 IO 调度器管理结构添加请求*/
    elevator_activate_req_fn *elevator_activate_req_fn;        /*激活请求*/
    elevator_deactivate_req_fn *elevator_deactivate_req_fn;    /*失活请求*/

    elevator_completed_req_fn *elevator_completed_req_fn;       /*释放请求前调用的函数*/

    elevator_request_list_fn *elevator_former_req_fn;      /*查找给定请求的前一个请求*/
    elevator_request_list_fn *elevator_latter_req_fn;        /*查找给定请求的后一个请求*/

    elevator_init_icq_fn *elevator_init_icq_fn;  /* see iocontext.h */
    elevator_exit_icq_fn *elevator_exit_icq_fn; /* ditto */

    elevator_set_req_fn *elevator_set_req_fn;    /*创建新请求后调用，设置请求*/
    elevator_put_req_fn *elevator_put_req_fn;   /*释放请求时调用*/

    elevator_may_queue_fn *elevator_may_queue_fn;      /*分配请求前调用，能否向队列插入请求*/

    elevator_init_fn *elevator_init_fn;       /*IO 调度器初始化函数，请求队列绑定调度器时调用*/
    elevator_exit_fn *elevator_exit_fn;      /*解绑调度器时调用*/
```

```
    elevator_registered_fn *elevator_registered_fn;
                                    /*由 elv_register_queue()函数调用，在添加磁盘时调用*/
};
```

内核在/block/elevator.c 文件内定义了全局双链表头 **elv_list** 用于管理所有的 elevator_type 实例。

int **elv_register**(struct elevator_type *e)函数（/block/elevator.c）用于向内核注册 IO 调度器实例，主要工作是创建调度器所需数据结构 slab 缓存以及将 elevator_type 实例添加到 **elv_list** 全局双链表末尾。内核定义了多个 IO 调度器实例供块设备驱动程序选用，详见下文。

10.4.2 创建标准请求队列

blk_init_queue()接口函数用于创建标准请求队列，块设备驱动程序须定义处理请求的 request_fn()函数，并作为 blk_init_queue()函数的参数。创建的标准请求队列需赋予磁盘 gendisk 实例。

配对的 blk_cleanup_queue()函数用于清理请求队列，在关闭设备（移除模块）时调用。

blk_init_queue()函数定义在/block/blk-core.c 头文件，返回新创建请求队列 request_queue 实例指针：

```
struct request_queue *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
/*
*rfn：处理请求的 request_fn_proc()类型函数指针。
*lock：自旋锁指针，操作请求队列中 request 实例时需要持有该锁，队列 q->queue_lock 指向该锁，
*       如果 lock 为 NULL，q->queue_lock 指向 q->__queue_lock。
*/
{
    return blk_init_queue_node(rfn, lock, NUMA_NO_NODE);
                                /*UMA 系统 NUMA_NO_NODE 为-1，/block/blk-core.c*/
}
```

blk_init_queue()函数内直接调用 blk_init_queue_node()函数，该函数可用于 UMA 和 NUMA 系统，函数定义如下（/block/blk-core.c）：

```
struct request_queue *blk_init_queue_node(request_fn_proc *rfn, spinlock_t *lock, int node_id)
{
    struct request_queue *uninit_q, *q;

    uninit_q = blk_alloc_queue_node(GFP_KERNEL, node_id);
                            /*分配请求队列 request_queue 实例，并做简单初始化，/block/blk-core.c*/
    if (!uninit_q)
        return NULL;

    q = blk_init_allocated_queue(uninit_q, rfn, lock);      /*初始化请求队列，/block/blk-core.c*/
    if (!q)
        blk_cleanup_queue(uninit_q);  /*不成功则释放之前创建的请求队列*/

    return q;
}
```

blk_init_queue()函数主要工作分成两部分：

（1）调用 blk_alloc_queue_node()函数分配 request_queue 实例并做简单初始化。

（2）调用 blk_init_allocated_queue()函数对分配的实例进行进一步的初始化，下文将详细介绍这两个函数的实现。

1 分配请求队列

blk_alloc_queue_node()函数定义在/block/blk-core.c 文件内，代码如下。

```
struct request_queue *blk_alloc_queue_node(gfp_t gfp_mask, int node_id)
/*gfp_mask：分配掩码，node_id：分配 request_queue 实例的内存节点*/
{
    struct request_queue *q;
    int err;

    q = kmem_cache_alloc_node(blk_requestq_cachep,gfp_mask | __GFP_ZERO, node_id);
                                                    /*从 slab 缓存分配全零 request_queue 实例*/
    if (!q)
        return NULL;

    q->id = ida_simple_get(&blk_queue_ida, 0, 0, gfp_mask);
                                /*从 ida 实例中为请求队列分配 id 号，/block/blk-core.c*/
    if (q->id < 0)
        goto fail_q;

    /*初始化 backing_dev_info 实例成员*/
    q->backing_dev_info.ra_pages =(VM_MAX_READAHEAD * 1024) / PAGE_CACHE_SIZE;
                                                    /*内存页大小为 4KB 时，ra_pages 为 32 页*/
    q->backing_dev_info.capabilities = BDI_CAP_CGROUP_WRITEBACK;
    q->backing_dev_info.name = "block";
    q->node = node_id;

    err = bdi_init(&q->backing_dev_info);
                            /*初始化 backing_dev_info 成员，见第 11 章，/mm/backing-dev.c*/
    if (err)
      goto fail_id;

    setup_timer(&q->backing_dev_info.laptop_mode_wb_timer, laptop_mode_timer_fn, (unsigned long) q);
    setup_timer(&q->timeout, blk_rq_timed_out_timer, (unsigned long) q);
    INIT_LIST_HEAD(&q->queue_head);
    INIT_LIST_HEAD(&q->timeout_list);
    INIT_LIST_HEAD(&q->icq_list);
#ifdef CONFIG_BLK_CGROUP
    INIT_LIST_HEAD(&q->blkg_list);
#endif
```

```
    INIT_DELAYED_WORK(&q->delay_work, blk_delay_work);
                    /*延时工作，blk_delay_work()调用__blk_run_queue()函数，处理请求*/

    kobject_init(&q->kobj, &blk_queue_ktype);
                    /*初始化跟踪请求队列的 kobject 实例，blk_queue_ktype 定义在/block/blk-sysfs.c*/

    mutex_init(&q->sysfs_lock);
    spin_lock_init(&q->__queue_lock);       /*初始化请求队列自旋锁*/
    q->queue_lock = &q->__queue_lock;       /*设置初始值，后面将指向 lock*/

    q->bypass_depth = 1;
    __set_bit(QUEUE_FLAG_BYPASS, &q->queue_flags);     /*设置旁路标记位（不使用 IO 调度器）*/

    init_waitqueue_head(&q->mq_freeze_wq);        /*初始化等待队列头*/

    if (blkcg_init_queue(q))     /*组控制初始化，需选择 BLK_CGROUP 选项，/block/blk-cgroup.c*/
        goto fail_bdi;
    return q;          /*返回 request_queue 实例指针*/
    ...
}
```

blk_alloc_queue_node()函数主要工作是从 slab 缓存中分配全零 request_queue 实例，并对其部分成员进行初始化，最后返回实例指针。

其中 bdi_init()函数初始化请求队列 backing_dev_info 结构体成员（见第 11 章），延时工作 delay_work 成员的执行函数 blk_delay_work()将调用__blk_run_queue()函数处理队列中请求，见本章下文。

2 初始化请求队列

blk_init_queue_node()函数在分配请求队列 request_queue 实例后，调用 blk_init_allocated_queue()函数对请求队列进行进一步的初始化，函数定义如下（/block/blk-core.c）：

```
struct request_queue *blk_init_allocated_queue(struct request_queue *q, request_fn_proc *rfn,   \
                                                                        spinlock_t *lock)
/*rfn：处理请求函数*/
{
    if (!q)
        return NULL;

    q->fq = blk_alloc_flush_queue(q, NUMA_NO_NODE, 0);
                            /*创建并初始化 blk_flush_queue 结构体实例，/block/blk-flush.c*/
    if (!q->fq)
        return NULL;

    if (blk_init_rl(&q->root_rl, q, GFP_KERNEL))
        /*初始化 request_list 结构体成员 root_rl，创建分配请求实例的内存池等，/block/blk-core.c*/
        goto fail;
```

```
    q->request_fn      = rfn;        /*请求处理函数*/
    q->prep_rq_fn      = NULL;
    q->unprep_rq_fn    = NULL;
    q->queue_flags     |= QUEUE_FLAG_DEFAULT;      /*设置默认标记*/

    if (lock)
        q->queue_lock      = lock;        /*参数 lock 非 NULL，则指向它*/

    blk_queue_make_request(q, blk_queue_bio);     /*设置请求队列，/block/blk-settings.c*/
                                /*请求队列 make_request_fn 函数指针赋值为 blk_queue_bio()*/

    q->sg_reserved_size = INT_MAX;
    mutex_lock(&q->sysfs_lock);

    if (elevator_init(q, NULL)) {          /*设置 IO 调度器，默认为 CFQ 调度器，/block/elevator.c*/
        mutex_unlock(&q->sysfs_lock);
        goto fail;
    }
    mutex_unlock(&q->sysfs_lock);
    return q;          /*返回请求队列指针*/
    ...
}
```

blk_init_allocated_queue()函数用于初始化已分配的请求队列，主要完成工作如下：

（1）调用 blk_alloc_flush_queue()函数创建处理 FLUSH/FUA 请求的 Flush 队列。

（2）调用 blk_init_rl()初始化 request_list 结构体成员 root_rl，其中包含分配请求的内存池。

（3）将参数传递的请求处理函数 rfn()指针赋予请求队列 request_fn 函数指针成员。

（4）调用 blk_queue_make_request()函数设置请求队列 make_request_fn 函数指针为 **blk_queue_bio()**，设置请求队列限制值等。

（5）调用 elevator_init()函数初始化请求队列绑定的 IO 调度器等。

下面主要看一下 blk_alloc_flush_queue()、blk_init_rl()、blk_queue_make_request()和 elevator_init()函数的实现。

■创建 FLUSH 队列

blk_alloc_flush_queue()函数用于为请求队列创建 FLUSH 队列，定义如下（/block/blk-flush.c）：

```
struct blk_flush_queue *blk_alloc_flush_queue(struct request_queue *q,int node, int cmd_size)
{
    struct blk_flush_queue *fq;          /*blk_flush_queue 结构体指针*/
    int rq_sz = sizeof(struct request);     /*请求大小*/

    fq = kzalloc_node(sizeof(*fq), GFP_KERNEL, node);     /*分配 blk_flush_queue 实例*/
```

```
    ...

    if (q->mq_ops) {        /*适用于 Multi queue 队列*/
        spin_lock_init(&fq->mq_flush_lock);
        rq_sz = round_up(rq_sz + cmd_size, cache_line_size());
    }

    fq->flush_rq = kzalloc_node(rq_sz, GFP_KERNEL, node);          /*分配请求 request 实例*/
    ...
    /*初始化双链表成员*/
    INIT_LIST_HEAD(&fq->flush_queue[0]);
    INIT_LIST_HEAD(&fq->flush_queue[1]);
    INIT_LIST_HEAD(&fq->flush_data_in_flight);

    return fq;          /*返回 blk_flush_queue 实例指针*/
    ...
}
```

■创建请求内存池

blk_init_rl()函数用于初始化请求队列 request_list 结构体成员 root_rl，主要是创建用于分配请求的内存池，队列创建新请求时从这里分配，函数定义如下（/block/blk-core.c）：

```
int blk_init_rl(struct request_list *rl, struct request_queue *q,gfp_t gfp_mask)
{
    if (unlikely(rl->rq_pool))
        return 0;

    rl->q = q;        /*指向请求队列*/
    rl->count[BLK_RW_SYNC] = rl->count[BLK_RW_ASYNC] = 0;
    rl->starved[BLK_RW_SYNC] = rl->starved[BLK_RW_ASYNC] = 0;
    init_waitqueue_head(&rl->wait[BLK_RW_SYNC]);
    init_waitqueue_head(&rl->wait[BLK_RW_ASYNC]);

    rl->rq_pool = mempool_create_node(BLKDEV_MIN_RQ, alloc_request_struct,
                        free_request_struct, (void *)(long)q->node, gfp_mask,q->node);
                            /*创建内存池，从 request 结构体 slab 缓存中分配实例*/
    if (!rl->rq_pool)
        return -ENOMEM;
    return 0;
}
```

■设置请求队列

blk_queue_make_request()函数也用于设置请求队列，函数定义在/block/blk-settings.c 文件内：

```
void blk_queue_make_request(struct request_queue *q, make_request_fn *mfn)
/*mfn：处理 bio 的函数，这里为 blk_queue_bio()*/
{
    /*设置默认值*/
    q->nr_requests = BLKDEV_MAX_RQ;      /*设置请求最大值，128，/include/linux/blkdev.h*/

    q->make_request_fn = mfn;              /*设置 make_request_fn 函数指针成员*/
    blk_queue_dma_alignment(q, 511);
    blk_queue_congestion_threshold(q);    /*设置请求队列拥塞阀值，/block/blk-core.c*/
    q->nr_batching = BLK_BATCH_REQ;     /*32，/block/blk.h*/
    blk_set_default_limits(&q->limits);    /*设置请求队列限制值*/
                        /*初始化请求队列 queue_limits 结构体成员，/block/blk-settings.c*/

    blk_queue_bounce_limit(q, BLK_BOUNCE_HIGH);     /*/block/blk-settings.c*/
}
```

mfn 函数指针参数将赋予请求队列 make_request_fn 成员，这里为 **blk_queue_bio()**函数指针，请求队列 make_request_fn 成员函数在 VFS 向请求队列提交 bio 时调用，用于确定是否为 bio 创建新的请求，还是合并到现有请求，并将新请求插入管理结构，激活请求的处理。

内核在/block/blk-settings.c 文件内还定义其它一些设置请求队列的接口函数，请读者自行查阅。

■设置 IO 调度器

blk_init_allocated_queue()函数调用 **elevator_init**(q, NULL)函数设置请求队列关联的 IO 调度器。

内核定义了 NOOP、DEADLINE、CFQ 等 IO 调度器实例，调度器实现及注册代码位于/block/*-iosched.c 文件内，这些文件永久编译或以模块的形式编译入内核，在内核初始化或加载模块时将向内核注册文件中定义的 IO 调度器实例。

NOOP 调度器：它是一个非常简单的 IO 调度器，将新来的请求按“先来先服务”的原则依次添加到请求队列，以便进行处理。请求会进行合并但无法重排，比较适合基于 Flash 的存储设备。

DEADLINE 调度器：它试图最小化磁盘寻道的次数，并尽可能确保请求在一定时间内处理完成，适用于读取数据较多的系统。

CFQ 调度器：完全公平 IO 调度器，内核默认使用的调度器。它围绕几个请求队列展开，所有的请求都在这些队列中排序。同一给定进程的请求，总是在同一队列中处理。时间片会分配到每个队列，内核使用一个轮转算法来处理各个队列。这确保了 IO 带宽以公平的方式在不同队列之间共享。

用户可通过"elevator=***"命令行参数（优先采用）或配置菜单设置标准请求队列默认采用的 IO 调度器实例。

elevator_init(struct request_queue *q, char *name)函数用于设置标准请求队列使用的 IO 调度器，函数代码如下（/block/elevator.c）：

```
int elevator_init(struct request_queue *q, char *name)
/*q：请求队列 request_queue 实例指针，name：IO 调度器名称，NULL 表示采用默认的 IO 调度器*/
{
```

```
    struct elevator_type *e = NULL;
    int err;

    lockdep_assert_held(&q->sysfs_lock);

    if (unlikely(q->elevator))
        return 0;

    INIT_LIST_HEAD(&q->queue_head);
    q->last_merge = NULL;
    q->end_sector = 0;
    q->boundary_rq = NULL;

    if (name) {              /*如果 name 不为 NULL，由名称查找 IO 调度器实例*/
        e = elevator_get(name, true);
        if (!e)
            return -EINVAL;
    }

    if (!e && *chosen_elevator) {     /*name 为 NULL 且 chosen_elevator 不为 NULL*/
        e = elevator_get(chosen_elevator, false);     /*采用 chosen_elevator 指定的 IO 调度器*/
        if (!e)
            printk(KERN_ERR "I/O scheduler %s not found\n",chosen_elevator);
    }

    if (!e) {          /*如果 chosen_elevator 为 NULL，则采用配置选项选择的 IO 调度器*/
        e = elevator_get(CONFIG_DEFAULT_IOSCHED, false);
        if (!e) {
            printk(KERN_ERR"Default I/O scheduler not found. " \"Using noop.\n");
            e = elevator_get("noop", false);   /*最后仍无法找到指定的 IO 调度器，则采用 NOOP*/
        }
    }

    err = e->ops.elevator_init_fn(q, e);       /*调用 IO 调度器初始化函数*/
    if (err)
        elevator_put(e);
    return err;
}
```

elevator_init()函数为请求队列选择 IO 调度器的顺序如下：

（1）参数 name 指定的 IO 调度器具有最高的优先级，这里 name 为 NULL。

（2）全局变量 chosen_elevator 指向 IO 调度器名称字符串，初始值为 NULL，命令行参数"elevator=*"用于设置 chosen_elevator 变量。命令行参数指定 IO 调度器的优先级次之。

（3）配置选项选择的 IO 调度器，具有第三优先级，它由 DEFAULT_IOSCHED 配置选项确定，配置

选项定义在/block/Konfig.iosched 配置文件中，默认选择 DEFAULT_CFQ（CFQ 调度器，可更改）。

（4）最后，如果 DEFAULT_IOSCHED 配置选项指定的 IO 调度器都找不到则采用 NOOP 调度器。

elevator_init()函数最后调用所选 IO 调度器内定义的初始化函数。

至此，创建标准请求队列的接口函数就介绍完了。但是，到这里并不代表请求队列就创建完了，驱动程序还需要对请求队列中成员进行设置后才能赋予 gendisk 实例，在/block/blk-settings.c 文件内定义了许多设置请求队列成员的接口函数。下面介绍的队列标签属于队列提供的一个附加功能，即用一个整数的标签值来标识请求。队列标签的创建也是在创建请求队列之后进行的。

3 队列标签

队列标签提供一种为请求分配标签值的机制，标签值是一个整数，用于标识请求，可能通过标签值来查找请求。标签可认为是赋予请求的一种属性。

标准请求队列中的标签值由 blk_queue_tag 结构体管理，请求队列 request_queue 中 **queue_tags** 成员指向 blk_queue_tag 实例。

blk_queue_tag 结构体定义如下（/include/linux/blkdev.h）：

```
struct blk_queue_tag {
    struct request **tag_index;      /*指向请求指针数组，索引值即标签值*/
    unsigned long *tag_map;          /*标签位图，位图中比特位位置即标签值*/
    int busy;                        /*当前已分配标签数量*/
    int max_depth;                   /*最大标签数量*/
    int real_max_depth;              /* what the array can hold */
    atomic_t refcnt;                 /* map can be shared */
    int alloc_policy;                /*标签分配策略*/
    int next_tag;                    /*下一个分配的标签值*/
};
```

blk_queue_tag 结构体中主要成员简介如下：

●**tag_map**：指向整数数组，用于表示标签位图，位图中比特位位置就是标签值。比特位置位表示标签值已分配给请求，清零表示尚未分配。

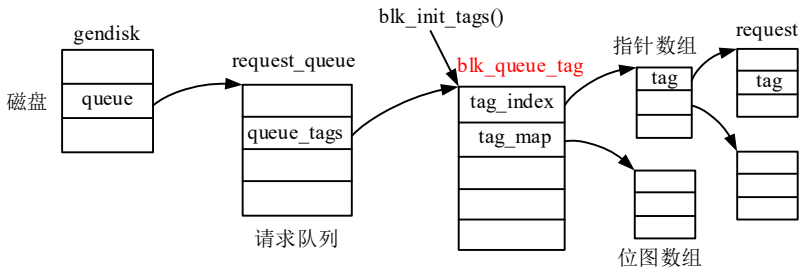
●**max_depth**：最大标签数量，即位图中有效位的数量。

●**tag_index**：指向请求指针数组，数组项数同最大标签数量，数组索引值就是标签值，数组项指向的请求表示此标签值分配给了这个请求，如果没有分配给请求则指针数组项为 NULL。

●**alloc_policy**：标签分配策略，取值定义如下（/include/linux/blkdev.h）：

```
#define   BLK_TAG_ALLOC_FIFO   0   /*从 0 开始分配*/
#define   BLK_TAG_ALLOC_RR     1   /*从上一次分配的标签值开始*/
```

队列标签结构如下图所示：



■创建队列标签

驱动程序可调用 blk_init_tags()函数创建队列标签并赋予 gendisk，函数定义如下（/block/blk-tag.c）：

```
struct blk_queue_tag *blk_init_tags(int depth, int alloc_policy)
{
    return __blk_queue_init_tags(NULL, depth, alloc_policy);
}

static struct blk_queue_tag *__blk_queue_init_tags(struct request_queue *q,int depth, int alloc_policy)
{
    struct blk_queue_tag *tags;

    tags = kmalloc(sizeof(struct blk_queue_tag), GFP_ATOMIC);      /*分配 blk_queue_tag 实例*/
    ...

    if (init_tag_map(q, tags, depth))        /*初始化 blk_queue_tag 实例*/
        goto fail;

    atomic_set(&tags->refcnt, 1);
    tags->alloc_policy = alloc_policy;      /*分配策略*/
    tags->next_tag = 0;
    return tags;            /*返回 blk_queue_tag 实例指针*/
    ...
}
```

__blk_queue_init_tags()函数调用 init_tag_map()函数初始化 blk_queue_tag 实例，如下（/block/blk-tag.c）：

```
static int init_tag_map(struct request_queue *q, struct blk_queue_tag *tags, int depth)
{
    struct request **tag_index;
    unsigned long *tag_map;
    int nr_ulongs;

    if (q && depth > q->nr_requests * 2) {        /*确定深度，最大标签数量*/
        depth = q->nr_requests * 2;
        printk(KERN_ERR "%s: adjusted depth to %d\n",__func__, depth);
    }

    tag_index = kzalloc(depth * sizeof(struct request *), GFP_ATOMIC);        /*分配指针数组*/
    ...
    nr_ulongs = ALIGN(depth, BITS_PER_LONG) / BITS_PER_LONG;
    tag_map = kzalloc(nr_ulongs * sizeof(unsigned long), GFP_ATOMIC);     /*分配整数数组位图*/
    ...
    tags->real_max_depth = depth;
```

```
        tags->max_depth = depth;
        tags->tag_index = tag_index;
        tags->tag_map = tag_map;
        return 0;       /*成功返回 0*/
        ...
}
```

blk_init_tags()函数创建的队列标签 blk_queue_tag 实例需要赋予请求队列，从而就可以使用请求的标签功能了。

int **blk_queue_init_tags**(struct request_queue *q, int depth,struct blk_queue_tag *tags, int alloc_policy)函数用于创建 blk_queue_tag 实例，并赋予请求队列（如果参数 tags 和 q->queue_tags 都为 NULL），或者为请求队列重新创建队列标签 blk_queue_tag 实例（q->queue_tags 非 NULL），或者将 tags 赋予请求队列（tags 非 NULL，q->queue_tags 为 NULL），函数源代码请读者自行阅读。

■接口函数

下面简要介绍一下使用队列标签的接口函数，这些函数都定义在/block/blk-tag.c 文件内，例如：

●int **blk_queue_start_tag**(struct request_queue *q, struct request *rq)：为请求 rq 分配标签值（查找位图，并置位比特位），标签值赋予 rq->tag 成员，请求关联到 q->queue_tags.tag_index[**tag**]指针数组项等，成功返回 0，否则返回 1。此函数可以在请求队列中的 prep_rq_fn()函数内调用。

●void **blk_queue_end_tag**(struct request_queue *q, struct request *rq)：结束请求对标签值的使用，清零位图中比特位，清 q->queue_tags.tag_index[**tag**]指针数组项等。

●struct request ***blk_queue_find_tag**(struct request_queue *q, int tag)：由标签值查找请求。

10.4.3 处理 bio

前一小节介绍了标准请求队列的创建，其 make_request_fn()函数指针被赋予 **blk_queue_bio()**函数指针。VFS 层通过 submit_bio()函数提交 bio 实例时，将调用此函数处理 bio。

bio 实例由进程通过 VFS 层提交，进程 task_struct 实例可以缓存一些进程向请求队列提交的请求，请求达到一定数量之后再提交到请求队列。这是为了增加请求合并的概率，以提高效率，也可以不缓存直接提交到请求队列。

task_struct 结构体缓存请求 request 实例的成员如下：

```
task_struct{
        ...
    #ifdef CONFIG_BLOCK
        struct blk_plug *plug;          /*进程提交请求 request 实例缓存链表*/
    #endif
        ...
}
```

blk_plug 结构体定义在/include/linux/blkdev.h 头文件：

```
struct blk_plug {
        struct list_head list;          /*请求 request 链表，适用于标准请求队列*/
        struct list_head mq_list;       /*请求 request 链表，适用于 Multi queue 队列*/
        struct list_head cb_list;       /* md requires an unplug callback */
```

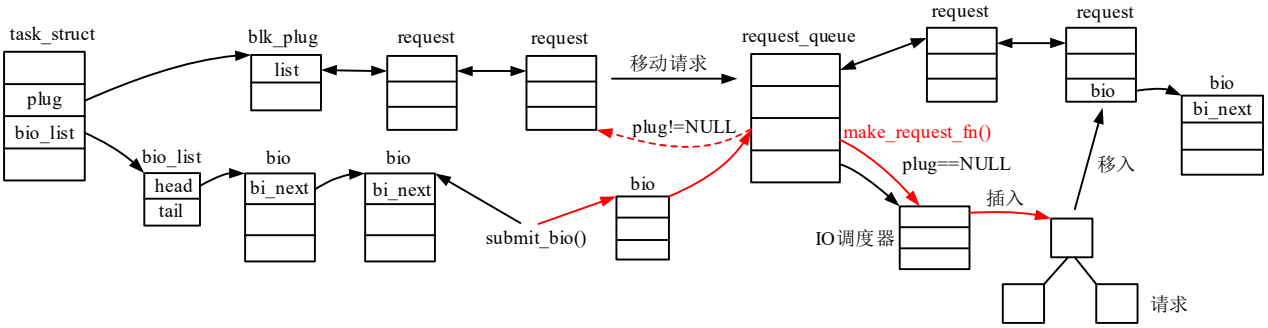
};

#define BLK_MAX_REQUEST_COUNT 16 /*最大缓存请求数量*/

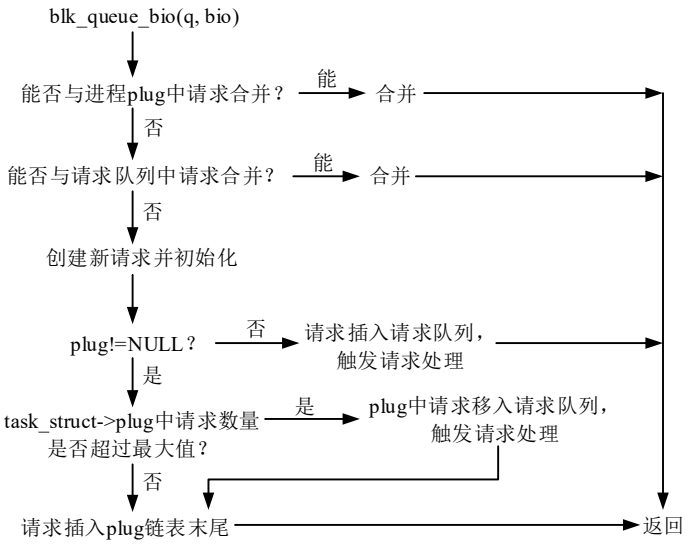
进程 task_struct 实例 plug 成员在创建进程时默认设置为 NULL，VFS 层发起对块设备的操作前可对 plug 成员赋值，本次操作结束后清除。也就是说，blk_plug 用于缓存一小段时间内请求，因为这段时间内的请求有相关性，可提高合并的概率。

进程提交 bio 的流程如下图所示，bio 由请求队列 make_request_fn()函数处理，标准请求队列此函数为**blk_queue_bio()**。

blk_queue_bio()函数先判断提交 bio 实例能否与进程 task_struct 实例中的暂存请求合并，如果可以则合并，如果不能合并则再判断能否与请求队列中现有请求合并，可以则合并，不能则创建新请求。如果进程 current->plug 不为 NULL，则将新请求添加到此链表末尾，否则将请求插入请求队列。current->plug 链表中的请求达到一定数量时，会将缓存请求移入请求队列。



blk_queue_bio()函数的执行流程简列如下图所示：



blk_queue_bio()函数定义在/block/blk-core.c 文件内，代码如下：

```
static void blk_queue_bio(struct request_queue *q, struct bio *bio)
/*q：请求队列 request_queue 实例指针，bio：提交的 bio 实例指针*/
{
    const bool sync = !!(bio->bi_rw & REQ_SYNC);    /*同步读或写操作*/
    struct blk_plug *plug;        /*/include/linux/blkdev.h*/
    int el_ret, rw_flags, where = ELEVATOR_INSERT_SORT;    /*请求插入 IO 调度器管理结构*/
    struct request *req;
    unsigned int request_count = 0;    /*扫描 task_struct->plug 链表中请求的数量*/
```

```
    blk_queue_bounce(q, &bio);     /*/block/bounce.c*/
                /*驱动程序可要求内存缓存位于某一区域之上，一般此函数不需要做什么工作*/

    if (bio_integrity_enabled(bio) && bio_integrity_prep(bio)) {
                                        /*数据完整性检查，/block/bio-integrity.c*/
        bio_endio(bio, -EIO);          /*/block/bio.c*/
        return;
    }

    if (bio->bi_rw & (REQ_FLUSH | REQ_FUA)) {     /*Flush 或 FUA 类型 bio，写操作*/
        spin_lock_irq(q->queue_lock);
        where = ELEVATOR_INSERT_FLUSH;     /*指示将新创建请求插入到 FLUSH 队列*/
        goto get_rq;       /*跳转至 get_rq 处获取请求*/
    }

    if (!blk_queue_nomerges(q) &&blk_attempt_plug_merge(q, bio, &request_count, NULL))
        return;                          /*/block/blk-core.c*/
       /*如果请求队列允许请求合并，且提交 bio 能与进程 plug 中请求合并，则合并，函数返回*/

    spin_lock_irq(q->queue_lock);        /*持有自旋锁*/

    el_ret = elv_merge(q, &req, bio);
                /*调用 IO 调度器函数，检测 bio 能否与请求队列中请求合并，/block/elevator.c*/
    if (el_ret == ELEVATOR_BACK_MERGE) {   /*能与 IO 调度器中管理请求的后项（bio）合并*/
        if (bio_attempt_back_merge(q, req, bio)) {
            elv_bio_merged(q, req, bio);              /*合并 bio 至指定请求*/
            if (!attempt_back_merge(q, req))        /*req 能否与后面的请求合并，可以则合并*/
                elv_merged_request(q, req, el_ret);       /*合并请求后操作*/
            goto out_unlock;                /*函数返回*/
        }
    } else if (el_ret == ELEVATOR_FRONT_MERGE) {     /*能与请求前项（bio）合并*/
        if (bio_attempt_front_merge(q, req, bio)) {            /*执行前向合并，函数返回*/
            elv_bio_merged(q, req, bio);
            if (!attempt_front_merge(q, req))
                elv_merged_request(q, req, el_ret);
            goto out_unlock;                /*函数返回*/
        }
    }

    /*以下是需要为 bio 创建新请求的情形*/
get_rq:
    rw_flags = bio_data_dir(bio);   /*读还是写操作，((bio)->bi_rw & 1)，/include/linux/fs.h*/
```

```
    if (sync)
        rw_flags |= REQ_SYNC;       /*是否是同步操作*/
    req = get_request(q, rw_flags, bio, GFP_NOIO); /*获取请求 request 实例，/block/blk-core.c*/
    ...
    init_request_from_bio(req, bio);        /*初始化请求 request 实例，/block/blk-core.c*/

    if (test_bit(QUEUE_FLAG_SAME_COMP, &q->queue_flags))
        req->cpu = raw_smp_processor_id();

    plug = current->plug;
    if (plug) {     /*plug!=NULL*/
        if (!request_count)         /*如果 task_struct->plug 中请求数量为 0，直接将请求插入链表末尾*/
            trace_block_plug(q);
        else {
            if (request_count >= BLK_MAX_REQUEST_COUNT) {     /*链表中请求数达到最大值*/
                blk_flush_plug_list(plug, false);
                 /*将 plug 链表中请求刷出到请求队列，激活请求处理，/block/blk-core.c*/
                trace_block_plug(q);
            }
        }
        list_add_tail(&req->queuelist, &plug->list);     /*请求添加到进程 plug->list 链表末尾*/
        blk_account_io_start(req, true);       /*更新统计量，/block/blk-core.c*/
    }

    else {        /*plug==NULL*/
        spin_lock_irq(q->queue_lock);           /*持有队列锁*/
        add_acct_request(q, req, where);     /*将请求插入到请求队列，/block/blk-core.c*/
        __blk_run_queue(q);                 /*调用 request_fn()函数处理请求，/block/blk-core.c*/
    out_unlock:     /*bio 能与 IO 调度器现有请求合并，合并后直接跳至此处，不触发请求处理*/
        spin_unlock_irq(q->queue_lock);         /*释放队列锁*/
    }
}
```

下面简要介绍 blk_queue_bio()函数中几个比较重要的步骤：

1.与现有请求合并

bio 实例与现有请求的合并要考虑两种情况，一是判断能否与进程 plug 链表中的请求合并，二是判断能否与 IO 调度器管理的现有请求合并（请求队列中请求）。

（1）与进程 plug 链表中请求合并

请求队列 queue_flags 标记成员 QUEUE_FLAG_NOMERGES 标记位，指示了请求队列是否允许将 bio 实例与现有请求合并，置位表示不允许合并，清零表示允许合并。

blk_queue_nomerges(q)函数用于提取此标记位的值，如果标记位为 0，再调用 blk_attempt_plug_merge()函数尝试将 bio 实例与 plug 链表中请求合并，如果可以合并则合并，函数返回 true，不能合并返回 false，参数 request_count 保存的是合并请求在 plug 链表中的位置值，从 1 开始，如果不能合并 request_count 保存 plug 链表中请求的数量（只计算提交到同一请求队列中的请求）。能合并的条件有很多，例如 bio 访问

的块设备数据与请求访问的块设备数据接续上，且访问属性相同等，有兴趣的读者可自行阅读源代码。

（2）与请求队列中请求合并

如果不能与进程 plug 链表中请求合并，接着将调用 elv_merge()函数判断 bio 能否与 IO 调度器管理的请求合并，能合并则由 IO 调度器执行合并操作，有兴趣的读者可以研究一下，这里作者暂且略过。

2.获取请求并初始化

如果前面的合并都不成功，则调用 **get_request()**函数为 bio 创建新请求，调用 **init_request_from_bio()**函数初始化请求实例，这两个函数在下文中将做介绍。

3.将新请求插入队列

（1）如果 plug!=NULL，则将请求插入 plug 链表末尾。若链表中请求已经达到了最大值，则先调用函数 blk_flush_plug_list()将 plug 链表中请求移至请求队列，然后再将请求插入 plug 链表末尾。

（2）如果 plug==NULL，调用 add_acct_request(q, req, where)函数将请求插入到请求队列，然后调用函数__blk_run_queue()触发请求的处理。

虚拟文件系统层发起对块设备的操作时，如果需要使用 plug 链表，在发起读写操作前先调用接口函数 blk_start_plug(plug)对进程 plug 成员赋值（实例由调用者创建）。在提交完所有 bio 实例后，调用接口函数 blk_finish_plug(plug)，函数内调用 blk_flush_plug_list()函数将 plug 中请求移至请求队列并触发请求的处理，最后对进程 plug 成员赋予 NULL。

1 获取新请求

如果提交的 bio 实例不能与现有请求合并，则需要为其创建新的请求 request 实例。get_request()函数用于分配新 request 实例并初始化，成功返回请求实例指针，否则返回错误码。

get_request()函数代码如下（/block/blk-core.c）：

```
static struct request *get_request(struct request_queue *q, int rw_flags,struct bio *bio, gfp_t gfp_mask)
/*q：请求队列指针，rw_flag：读写标记，bio：bio 实例指针，可为 NULL（传输命令）*/
{
    const bool is_sync = rw_is_sync(rw_flags) != 0;
    DEFINE_WAIT(wait);
    struct request_list *rl;
    struct request *rq;

    rl = blk_get_rl(q, bio);   /*没有选择 BLK_CGROUP 配置选项时，返回 q->root_rl*/
retry:
    rq = __get_request(rl, rw_flags, bio, gfp_mask);   /*分配请求实例，见下文，/block/blk-core.c*/
    if (!IS_ERR(rq))
        return rq;          /*成功，返回请求实例指针*/
    ...                     /*分配失败的处理，重试*/
    goto retry;
}
```

get_request()函数主要是调用__get_request()函数从请求队列 request_list 结构体成员指示的内存池中分配请求实例，函数定义如下（/block/blk-core.c）：

```
static struct request *__get_request(struct request_list *rl, int rw_flags,struct bio *bio, gfp_t gfp_mask)
{
```

```
struct request_queue *q = rl->q;        /*请求队列*/
struct request *rq;
struct elevator_type *et = q->elevator->type;        /*IO 调度器*/
struct io_context *ioc = rq_ioc(bio);    /*没有选择 BLK_CGROUP，返回 current->io_context*/
struct io_cq *icq = NULL;
const bool is_sync = rw_is_sync(rw_flags) != 0;    /*是否是同步操作*/
int may_queue;

if (unlikely(blk_queue_dying(q)))    /*检测 q->queue_flags 的 QUEUE_FLAG_DYING 标记位*/
    return ERR_PTR(-ENODEV);      /*请求队列是否死亡*/

may_queue = elv_may_queue(q, rw_flags);
                                /*调用调度器 elevator_may_queue_fn()函数，/block/elevator.c*/
if (may_queue == ELV_MQUEUE_NO)
    goto rq_starved;

/*如果已分配请求数量超过队列阀值*/
if (rl->count[is_sync]+1 >= queue_congestion_on_threshold(q)) {
                                    /*返回 q->nr_congestion_on 拥塞阀值，/block/blk.h*/
    if (rl->count[is_sync]+1 >= q->nr_requests) {
        if (!blk_rl_full(rl, is_sync)) {
            /*rl->flag 成员是否设置 BLK_RL_SYNCFULL 或 BLK_RL_SYNCFULL 标记位*/
            ioc_set_batching(q, ioc);
            blk_set_rl_full(rl, is_sync);    /*设置标记位*/
        } else {
            if (may_queue != ELV_MQUEUE_MUST&& !ioc_batching(q, ioc)) {
                return ERR_PTR(-ENOMEM);
            }
        }
    }
    blk_set_congested(rl, is_sync);    /*/block/blk-core.c*/
            /*设置 q->backing_dev_info.wb.congested.state 成员拥塞标记*/
}  /*超过拥塞阀值处理完毕*/

if (rl->count[is_sync] >= (3 * q->nr_requests / 2))    /*分配请求数量大于队列最大请求的 1.5 倍*/
    return ERR_PTR(-ENOMEM);      /*返回错误码*/

/*更新统计值*/
q->nr_rqs[is_sync]++;
rl->count[is_sync]++;
rl->starved[is_sync] = 0;

if (blk_rq_should_init_elevator(bio) && !blk_queue_bypass(q)) {    /*/block/blk-core.c*/
```

```
                    /*是否需要初始化 IO 调度器数据，REQ_FLUSH 或 REQ_FUA 请求不需要*/
            rw_flags |= REQ_ELVPRIV;
            q->nr_rqs_elvpriv++;
            if (et->icq_cache && ioc)
                icq = ioc_lookup_icq(ioc, q);
        }

        if (blk_queue_io_stat(q))       /*检测 q->queue_flags 的 QUEUE_FLAG_IO_STAT 标记位*/
            rw_flags |= REQ_IO_STAT;
        spin_unlock_irq(q->queue_lock);          /*释放自旋锁*/

        rq = mempool_alloc(rl->rq_pool, gfp_mask);       /*从内存池分配请求实例*/
        if (!rq)
            goto fail_alloc;

        blk_rq_init(q, rq);       /*初始化请求实例，/block/blk-core.c*/
        blk_rq_set_rl(rq, rl);     /*设置 rq->rl = rl*/
        rq->cmd_flags = rw_flags | REQ_ALLOCED;
        /*初始化 IO 调度器数据*/
        if (rw_flags & REQ_ELVPRIV) {
            if (unlikely(et->icq_cache && !icq)) {
                if (ioc)
                    icq = ioc_create_icq(ioc, q, gfp_mask);
                if (!icq)
                    goto fail_elvpriv;
            }
            rq->elv.icq = icq;
            if (unlikely(elv_set_request(q, rq, bio, gfp_mask))) /*调用 IO 调度器 elevator_set_req_fn()函数*/
                goto fail_elvpriv;

            if (icq)
                get_io_context(icq->ioc);
        }
    out:
        if (ioc_batching(q, ioc))
            ioc->nr_batch_requests--;

        trace_block_getrq(q, bio, rw_flags & 1);
        return rq;          /*返回请求实例指针*/
        ...
    }
```

在分配请求实例前，需要判断分配数量是否超过阀值，__get_request()函数内判断请求队列 q->root_rl 成员中统计的异步或同步请求数量是否超过请求队列拥塞的阀值，即 q->nr_congestion_on 成员值（初始化

请求队列时设置），如果是则需要设置对应后备存储设备信息结构中的拥塞标记位。然后，从 q->root_rl 成员指向的内存池中分配请求实例（最终从 slab 缓存中分配）并初始化，如果需要初始化请求对应的 IO 调度器的数据，则调用 IO 调度器的 elevator_set_req_fn()函数执行，最后函数返回请求 request 实例指针。

2 初始化请求

前面新分配的 request 实例还没有建立与 bio 实例之间的关联，blk_queue_bio()函数创建请求实例后，还需要调用 init_request_from_bio()函数对请求进一步进行设置。

init_request_from_bio()函数定义如下（/block/blk-core.c）：

```
void init_request_from_bio(struct request *req, struct bio *bio)
{
    req->cmd_type = REQ_TYPE_FS;        /*文件系统提交的命令*/

    req->cmd_flags |= bio->bi_rw & REQ_COMMON_MASK;     /*设置读写标记位*/
    if (bio->bi_rw & REQ_RAHEAD)        /*预读操作*/
        req->cmd_flags |= REQ_FAILFAST_MASK;

    req->errors = 0;
    req->__sector = bio->bi_iter.bi_sector;     /*起始扇区号*/
    req->ioprio = bio_prio(bio);
                        /*IO 优先级，保存在 bio->bi_rw 成员高 16 位，/include/linux/bio.h*/
    blk_rq_bio_prep(req->q, req, bio);      /*完成准备工作，/block/blk-core.c*/
}
```

初始化函数内调用 blk_rq_bio_prep()函数，完成请求在插入请求队列前的准备工作，函数代码如下：

```
void blk_rq_bio_prep(struct request_queue *q, struct request *rq,struct bio *bio)
{
    /*设置读写标记位（第 0 位），0 表示读，1 表示写*/
    rq->cmd_flags |= bio->bi_rw & REQ_WRITE;

    if (bio_has_data(bio))      /*包括有效的长度的读写数据，/include/linux/bio.h*/
        rq->nr_phys_segments = bio_phys_segments(q, bio);

    rq->__data_len = bio->bi_iter.bi_size;     /*读写数据长度*/
    rq->bio = rq->biotail = bio;        /*请求 bio 和 biotail 成员都指向提交的 bio 实例*/

    if (bio->bi_bdev)       /*块设备 block_device 实例指针*/
        rq->rq_disk = bio->bi_bdev->bd_disk;       /*指向表示磁盘的 gendisk 实例*/
}
```

3 插入请求

blk_queue_bio()函数在创建请求 request 实例并关联 bio 实例后，下一步的工作就是将请求插入到队列中。这里要分两种情况（相关代码段见下文）：

（1）当进程结构 plug 指针成员不为 NULL 时，请求将被插入到进程 plug 链表中，若此链表中请求超过最大值，则先将 plug 链表中请求移入请求队列，然后再将请求插入 plug 链表末尾。

（2）当进程结构 plug 指针成员为 NULL 时，请求将会插入到请求队列，并触发对请求的处理。

```
static void blk_queue_bio(struct request_queue *q, struct bio *bio)
{
    ...
    plug = current->plug;
    if (plug) {     /*plug!=NULL*/
        if (!request_count)        /*如果 task_struct->plug 中请求数量为 0，直接将请求插入链表末尾*/
            trace_block_plug(q);
        else {
            if (request_count >= BLK_MAX_REQUEST_COUNT) { /*链表中请求数量超过最大值*/
                blk_flush_plug_list(plug, false);
                 /*将 plug 链表中请求刷出到请求队列，激活请求处理，/block/blk-core.c*/
                trace_block_plug(q);
            }
        }
        list_add_tail(&req->queuelist, &plug->list);     /*请求添加到进程 plug->list 链表末尾*/
        blk_account_io_start(req, true);       /*更新统计量，/block/blk-core.c*/
    }

    else {        /*plug==NULL*/
        spin_lock_irq(q->queue_lock);
        add_acct_request(q, req, where);   /*将请求插入请求队列，/block/blk-core.c*/
        __blk_run_queue(q);                /*激活请求处理，/block/blk-core.c*/
        ...
     }
}
```

下面先介绍将请求插入请求队列时调用的 add_acct_request(q, req, where)，然后再介绍将 plug 链表中请求移出到请求队列的 blk_flush_plug_list(plug, false)函数和激活请求处理的__blk_run_queue(q)函数实现。

■插入请求队列

add_acct_request()函数用于将请求插入请求队列，插入请求队列主要又分三种情况如下所示：

（1）插入 FLUSH 请求队列，本节后面再介绍。

（2）插入请求队列 queue_head 双链表。

（3）插入 IO 调度器的管理结构中。

add_acct_request()函数定义如下（/block/blk-core.c）：

```
static void add_acct_request(struct request_queue *q, struct request *rq,int where)
/*where：指示将请求插入请求队列中什么位置*/
```

```
{
    blk_account_io_start(rq, true);        /*主要是更新分区统计值，/block/blk-core.c*/
    __elv_add_request(q, rq, where);       /*将请求插入到请求队列，/block/elevator.c*/
}
```

__elv_add_request(q, rq, where)函数根据参数 where 的不同值选择不同的操作方法，函数代码如下：

```
void __elv_add_request(struct request_queue *q, struct request *rq, int where)
/*where：请求插入位置*/
{
    trace_block_rq_insert(q, rq);
    blk_pm_add_request(q, rq);
    rq->q = q;          /*赋值请求队列*/

    if (rq->cmd_flags & REQ_SOFTBARRIER) {
        /* barriers are scheduling boundary, update end_sector */
        if (rq->cmd_type == REQ_TYPE_FS) {
            q->end_sector = rq_end_sector(rq);
            q->boundary_rq = rq;
        }
    } else if (!(rq->cmd_flags & REQ_ELVPRIV) &&
            (where == ELEVATOR_INSERT_SORT ||where == ELEVATOR_INSERT_SORT_MERGE))
        where = ELEVATOR_INSERT_BACK;

    switch (where) {    /*选择如何执行操作*/
    case ELEVATOR_INSERT_REQUEUE:       /*请求重新插入队列*/
    case ELEVATOR_INSERT_FRONT:         /*请求插入 q->queue_head 双链表头部*/
        rq->cmd_flags |= REQ_SOFTBARRIER;
        list_add(&rq->queuelist, &q->queue_head);    /*插入双链表头部*/
        break;

    case ELEVATOR_INSERT_BACK:                  /*请求插入 q->queue_head 双链表末尾*/
        rq->cmd_flags |= REQ_SOFTBARRIER;
        elv_drain_elevator(q); /*调用 IO 调度器 elevator_dispatch_fn()函数，刷出 IO 调度器管理请求*/
        list_add_tail(&rq->queuelist, &q->queue_head);       /*插入双链表末尾*/
        __blk_run_queue(q);     /*处理请求，见下文*/
        break;

    case ELEVATOR_INSERT_SORT_MERGE:     /*由 IO 调度器尝试合并，不成功再插入请求*/
        if (elv_attempt_insert_merge(q, rq))        /*合并成功返回 true，否则返回 false*/
            break;                                  /*合并不成功，继续执行后面代码*/
    case ELEVATOR_INSERT_SORT:      /*插入 IO 调度器管理结构中，初始值*/
        BUG_ON(rq->cmd_type != REQ_TYPE_FS);
        rq->cmd_flags |= REQ_SORTED;        /*由 IO 调度器排序*/
```

```
            q->nr_sorted++;
            if (rq_mergeable(rq)) {        /*请求是否允许合并，/include/linux/blkdev.h*/
                elv_rqhash_add(q, rq);      /*插入 IO 调度器管理的散列表*/
                if (!q->last_merge)
                    q->last_merge = rq;
            }
            q->elevator->type->ops.elevator_add_req_fn(q, rq);   /*调用 IO 调度器的插入函数*/
            break;

        case ELEVATOR_INSERT_FLUSH:            /*请求插入 FLUSH 请求队列*/
            rq->cmd_flags |= REQ_SOFTBARRIER;
            blk_insert_flush(rq);     /*将请求插入到 FLUSH 请求队列，见下文，/block/blk-flush.c*/
            break;
        default:
            printk(KERN_ERR "%s: bad insertion point %d\n",__func__, where);
            BUG();
        }
    }
```

blk_queue_bio()函数中 where 参数的初始值为 ELEVATOR_INSERT_SORT（后可能改变），即调用 IO 调度器中的函数将请求插入到 IO 调度器的管理结构中。

blk_queue_bio()函数在调用 add_acct_request()函数将请求插入请求队列后，将调用__blk_run_queue(q)函数激活请求的处理，详见下文。

■移出进程 plug 链表请求

移出进程 plug 链表中请求的 **blk_flush_plug_list()**函数定义在/block/blk-core.c 文件内，代码如下：

```
void blk_flush_plug_list(struct blk_plug *plug, bool from_schedule)
/*from_schedule：是不是由进程调度时调用的，这里为 false*/
{
    struct request_queue *q;
    unsigned long flags;
    struct request *rq;
    LIST_HEAD(list);     /*临时链表*/
    unsigned int depth;

    flush_plug_callbacks(plug, from_schedule);
            /*调用 plug->cb_list 链接 blk_plug_cb 实例中的回调函数，/block/blk-core.c*/

    if (!list_empty(&plug->mq_list))          /*Multi queue 队列，见下文*/
        blk_mq_flush_plug_list(plug, from_schedule);

    if (list_empty(&plug->list))          /*若链表为空，返回*/
```

```
        return;

    list_splice_init(&plug->list, &list);       /*将 plug->list 链表成员移动到临时链表 list*/
    list_sort(NULL, &list, plug_rq_cmp);     /*链表成员排序*/

    q = NULL;
    depth = 0;

    local_irq_save(flags);
    while (!list_empty(&list)) {       /*遍历临时链表成员*/
        rq = list_entry_rq(list.next);        /*请求指针*/
        list_del_init(&rq->queuelist);
        BUG_ON(!rq->q);
        if (rq->q != q) {     /*q 初始为 NULL*/
            if (q)        /*当前请求与前面请求不属于同一个请求队列*/
                queue_unplugged(q, depth, from_schedule);       /*激活前一队列的请求处理*/
            q = rq->q;        /*q 设为当前请求所属的队列*/
            depth = 0;
            spin_lock(q->queue_lock);
        }

        if (unlikely(blk_queue_dying(q))) {          /*请求队列死了*/
            __blk_end_request_all(rq, -ENODEV);
            continue;
        }

        if (rq->cmd_flags & (REQ_FLUSH | REQ_FUA))     /*执行插入操作，见上文*/
            __elv_add_request(q, rq, ELEVATOR_INSERT_FLUSH);
        else
            __elv_add_request(q, rq, ELEVATOR_INSERT_SORT_MERGE); /*IO 调度器管理请求*/

        depth++;       /*移入请求队列中请求的数量*/
    }       /*遍历链表成员结束*/

    if (q)
        queue_unplugged(q, depth, from_schedule);     /*触发请求的处理，/block/blk-core.c*/

    local_irq_restore(flags);
}
```

blk_flush_plug_list()函数将 plug 链表中所有请求调用前面介绍的**__elv_add_request**()函数插入到 IO 调度器的管理结构中，最后调用 queue_unplugged()函数激活对请求的处理。

若 plug 链表中请求可能属于不同的请求队列，在遍历请求时，当前请求与前一请求不属于同一请求队列时，对前一请求队列也将调用 queue_unplugged()函数激活对请求队列请求的处理。

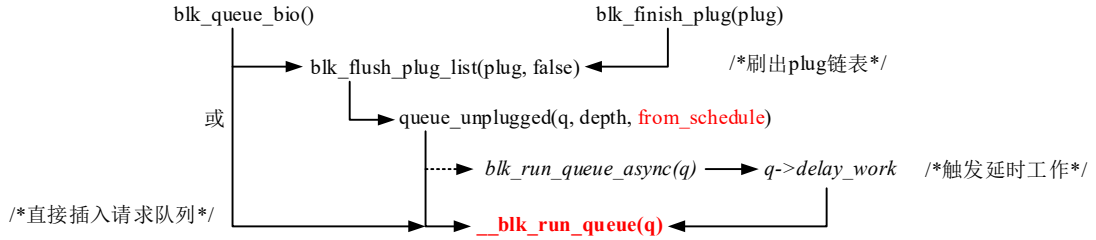
queue_unplugged()函数定义在/block/blk-core.c 文件内，代码如下：

```
static void queue_unplugged(struct request_queue *q, unsigned int depth,bool from_schedule) \
                                                        __releases(q->queue_lock)
/*form_schedule：这里为 false*/
{
    trace_block_unplug(q, depth, !from_schedule);

    if (from_schedule)
        blk_run_queue_async(q);     /*激活延时工作 q->delay_work*/
    else
        __blk_run_queue(q);     /*激活请求处理，/block/blk-core.c*/
    spin_unlock(q->queue_lock);
}
```

下面介绍__blk_run_queue(q)函数的实现。

■处理请求

在将进程 plug 链表中请求移入请求队列，以及将请求直接插入请求队时，都将调用__blk_run_queue(q)函数激活请求的处理，函数调用关系如下图所示：



__blk_run_queue(q)函数定义在/block/blk-core.c 文件内，注意参数 q 为请求队列指针，而不是请求实例指针，也就是说函数只是触发对请求的处理，而不一定就是处理刚刚提交的请求（请求处理的顺序由 IO 调度器决定）。

```
void __blk_run_queue(struct request_queue *q)
{
    if (unlikely(blk_queue_stopped(q)))     /*测试 QUEUE_FLAG_STOPPED 标记位*/
        return;

    __blk_run_queue_uncond(q);     /*无条件处理请求，/block/blk-core.c*/
}
```

__blk_run_queue_uncond()函数定义如下（/block/blk-core.c）：

```
inline void __blk_run_queue_uncond(struct request_queue *q)
{
    if (unlikely(blk_queue_dead(q)))
        return;
```

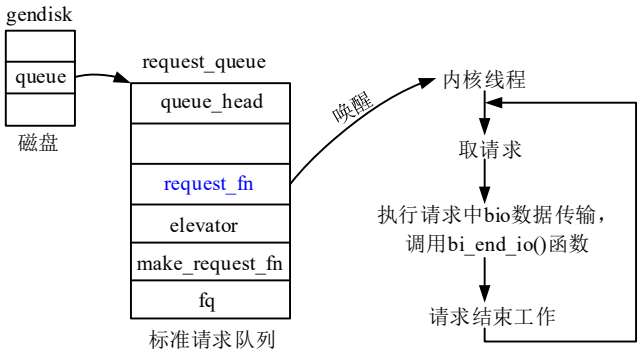
```
        q->request_fn_active++;
        q->request_fn(q);       /*调用请求队列定义的处理请求函数*/
        q->request_fn_active--;
    }
```

__blk_run_queue()函数最终调用请求队列中的 **request_fn()**函数处理请求，这是由具体块设备驱动程序定义的函数，下文将简要介绍驱动程序处理请求的大致流程。

10.4.4 处理请求

由前面介绍可知，提交 bio 创建请求后，请求被插入到请求队列，并最终调用请求队列的 request_fn()函数激活请求的处理。

request_fn()函数通常是唤醒一个内核线程，线程执行流程简列如下图所示：



内核线程是一个无限循环，循环体内从请求队列中取请求（优先级最高请求），执行请求中 bio 实例的数据传输，调用 bio 中 bi_end_io()回调函数，请求中 bio 都处理完成后，执行请求结束工作。

1 获取请求

块设备驱动的通用层提供了从请求队列中取请求的接口函数，定义在/block/blk-core.c 文件内，例如：blk_fetch_request(struct request_queue *q)函数用于从请求队列中取出最优先处理的请求。

```
struct request *blk_fetch_request(struct request_queue *q)
{
    struct request *rq;
    rq = blk_peek_request(q);          /*从队列取请求，/block/blk-core.c*/
    if (rq)
        blk_start_request(rq);      /*开始请求，从请求队列移除，设置定时器等，/block/blk-core.c*/
    return rq;          /*返回请求指针*/
}
```

blk_fetch_request(q)函数调用 blk_peek_request(q)函数，从请求队列中取出最优先处理的请求，然后调用 blk_start_request(rq)用于开始请求的处理，下面介绍这两个函数的定义。

■从队列取请求

blk_peek_request(q)函数从 q->queue_head 双链表中取出第一个请求返回，如果双链表为空则将 IO 调度器管理的请求移入到 q->queue_head 双链表后，再取第一个请求返回，如果还是没有请求，返回 NULL。

blk_peek_request(q)函数定义在/block/blk-core.c 文件内，代码如下：

```
struct request *blk_peek_request(struct request_queue *q)
{
    struct request *rq;
    int ret;

    while ((rq = __elv_next_request(q)) != NULL) {       /*取请求*/
        /*以下对请求进行初步的处理和检测*/
        rq = blk_pm_peek_request(q, rq);
        if (!rq)       /*rq 若为 NULL 跳出循环*/
            break;

        if (!(rq->cmd_flags & REQ_STARTED)) {       /*如果没有定义 REQ_STARTED 标记*/
            if (rq->cmd_flags & REQ_SORTED)
                elv_activate_rq(q, rq);         /*调用 IO 调度器 elevator_activate_req_fn()函数，激活*/

            rq->cmd_flags |= REQ_STARTED;       /*设置标记位*/
            trace_block_rq_issue(q, rq);
        }

        if (!q->boundary_rq || q->boundary_rq == rq) {
            q->end_sector = rq_end_sector(rq);
            q->boundary_rq = NULL;
        }

        if (rq->cmd_flags & REQ_DONTPREP)
            break;

        if (q->dma_drain_size && blk_rq_bytes(rq)) {
            rq->nr_phys_segments++;
        }

        if (!q->prep_rq_fn)     /*若 q->prep_rq_fn()为 NULL，跳出循环*/
            break;
        ret = q->prep_rq_fn(q, rq);          /*调用 q->prep_rq_fn()函数*/

        /*以下是根据 q->prep_rq_fn()函数返回值处理请求*/
        if (ret == BLKPREP_OK) {
            break;
        } else if (ret == BLKPREP_DEFER) {
            if (q->dma_drain_size && blk_rq_bytes(rq) &&!(rq->cmd_flags & REQ_DONTPREP)) {
                --rq->nr_phys_segments;
            }
```

```
                rq = NULL;
                break;
            } else if (ret == BLKPREP_KILL) {
                rq->cmd_flags |= REQ_QUIET;
                blk_start_request(rq);
                __blk_end_request_all(rq, -EIO);        /*结束请求*/
            } else {
                printk(KERN_ERR "%s: bad return=%d\n", __func__, ret);
                break;
            }
        }

        return rq;        /*返回请求指针*/
    }
```

blk_peek_request(q)函数调用**__elv_next_request()**函数从队列取出一个请求，然后在 while 循环体内对请求进行一些检查和处理，具体就不再解释了，最后返回请求实例指针，可能为 NULL。

下面主要看一下**__elv_next_request()**函数如何从队列中取出一个请求，定义如下（/block/blk.h）：

```
static inline struct request *__elv_next_request(struct request_queue *q)
{
    struct request *rq;
    struct blk_flush_queue *fq = blk_get_flush_queue(q, NULL);       /*FLUSH 队列*/

    while (1) {          /*无限循环*/
        if (!list_empty(&q->queue_head)) {          /*取 q->queue_head 链表第一个请求*/
            rq = list_entry_rq(q->queue_head.next);
            return rq;
        }

        /*q->queue_head 链表为 NULL 才执行下面的代码*/
        if (fq->flush_pending_idx != fq->flush_running_idx &&!queue_flush_queueable(q)) {
            fq->flush_queue_delayed = 1;        /*有 FLUSH 请求在处理，且关闭了 FLUSH 队列*/
            return NULL;
        }
        if (unlikely(blk_queue_bypass(q)) ||!q->elevator->type->ops.elevator_dispatch_fn(q, 0))
            return NULL;                        /*IO 调度器管理请求移入 q->queue_head 链表*/
    }       /*无限循环结束*/
}
```

__elv_next_request()函数比较好理解，如果 q->queue_head 双链表不为空，则取链表第一个请求返回即可；如果双链表为空，则调用 IO 调度器的 elevator_dispatch_fn()函数将 IO 调度器管理的请求移入双链表后，再取出第一个请求。FLUSH 请求的处理后面再单独介绍。

■开始请求

blk_fetch_request(q)函数在取出请求后，还要调用 blk_start_request(rq)函数开始请求，定义如下：

```
void blk_start_request(struct request *req)            /*/block/blk-core.c*/
{
    blk_dequeue_request(req);          /*请求从双链表中移出等，/block/blk-core.c*/

    req->resid_len = blk_rq_bytes(req);
    if (unlikely(blk_bidi_rq(req)))          /*双向请求*/
        req->next_rq->resid_len = blk_rq_bytes(req->next_rq);

    BUG_ON(test_bit(REQ_ATOM_COMPLETE, &req->atomic_flags));
    blk_add_timer(req);              /*/block/blk-timeout.c*/
        /*请求添加到 q->timeout_list 双链表（req->timeout_list），修改队列定时器超时时间*/
}
```

blk_fetch_request(q)函数主要是将请求从请求队列双链表中移，依请求修改队列超时定时器时间等。

2 执行数据传输

取出请求后，需遍历请求关联的 bio 实例，执行其中的数据传输。数据的传输是特定于设备的，必须由具体设备驱动程序实现。执行完 bio 包含的数据传输后，需要调用 bio_endio()函数结束此 bio。

bio_endio()函数定义如下（/block/bio.c）：

```
void bio_endio(struct bio *bio, int error)
{
    while (bio) {
        if (error)
            clear_bit(BIO_UPTODATE, &bio->bi_flags);
        else if (!test_bit(BIO_UPTODATE, &bio->bi_flags))
            error = -EIO;

        if (unlikely(!bio_remaining_done(bio)))        /*/block/bio.c*/
            break;
        if (bio->bi_end_io == bio_chain_endio) {       /*链表 bio，还有父 bio*/
            struct bio *parent = bio->bi_private;
            bio_put(bio);
            bio = parent;       /*父 bio*/
        } else {
            if (bio->bi_end_io)
                bio->bi_end_io(bio, error);          /*调用 bi_end_io()函数*/
            bio = NULL;
        }
    }
}
```

3 结束工作

请求包含的 bio 都执行完数据传输后，需要执行一些结束请求的工作，如释放请求占用的资源、释放请求实例等，下面介绍请求结束工作的接口函数。

■结束请求

常用的执行请求结束工作的函数有__blk_end_request_all()、blk_end_request_all()等，下面以这两个函数为例介绍其实现。

●__blk_end_request_all()

__blk_end_request_all()函数定义在/block/blk-core.c 文件内（调用此函数需持队列锁），代码如下：

```
void __blk_end_request_all(struct request *rq, int error)
{
    bool pending;
    unsigned int bidi_bytes = 0;

    if (unlikely(blk_bidi_rq(rq)))      /*返回 rq->next_rq != NULL，双向请求*/
        bidi_bytes = blk_rq_bytes(rq->next_rq);    /*返回下一请求的数据长度，rq->__data_len*/

    pending = __blk_end_bidi_request(rq, error, blk_rq_bytes(rq), bidi_bytes);
                                                    /*成功返回 0，/block/blk-core.c*/
    BUG_ON(pending);
}
```

__blk_end_bidi_request()函数定义在/block/blk-core.c 文件内：

```
bool __blk_end_bidi_request(struct request *rq, int error,unsigned int nr_bytes, unsigned int bidi_bytes)
/*
*rq：请求实例指针，error：请求执行的返回码，0 表示成功，<0 表示出错，
*nr_bytes：当前请求完成的数据长度，bidi_bytes：双向请求中，下一请求完成的数据长度。
*/
{
    if (blk_update_bidi_request(rq, error, nr_bytes, bidi_bytes))       /*会结束完成的 bio，调用回调函数*/
        /*调用 blk_update_request()检查是否还有数据没有完成传输，有返回 true，没有返回 false*/
        return true;

    blk_finish_request(rq, error);       /*终结请求，/block/blk-core.c*/
    return false;
}
```

blk_finish_request()函数定义在/block/blk-core.c 文件内，代码如下：

```
void blk_finish_request(struct request *req, int error)
{
```

```
    if (req->cmd_flags & REQ_QUEUED)
        blk_queue_end_tag(req->q, req);       /*释放请求标签，/block/blk-tag.c*/

    BUG_ON(blk_queued_rq(req));

    if (unlikely(laptop_mode) && req->cmd_type == REQ_TYPE_FS)
                                                    /*laptop_mode 是一个系统控制参数*/
        laptop_io_completion(&req->q->backing_dev_info);

    blk_delete_timer(req);       /*将请求从队列 q->timeout_list 双链表移出（req->timeout_list）*/

    if (req->cmd_flags & REQ_DONTPREP)
        blk_unprep_request(req);

    blk_account_io_done(req);

    if (req->end_io)     /*如果请求定义了回调函数*/
        req->end_io(req, error);       /*执行请求实例定义的回调函数*/
    else {                /*如果请求实例没有定义回调函数，默认没有定义*/
        if (blk_bidi_rq(req))       /*双向请求*/
            __blk_put_request(req->next_rq->q, req->next_rq);

        __blk_put_request(req->q, req);   /*释放请求实例，/block/blk-core.c*/
    }
}
```

__blk_put_request()函数用于释放请求实例，代码如下：

```
void __blk_put_request(struct request_queue *q, struct request *req)
{
    if (unlikely(!q))
        return;

    if (q->mq_ops) {        /*Multi queue 队列请求*/
        blk_mq_free_request(req);
        return;
    }

    blk_pm_put_request(req);

    elv_completed_request(q, req);     /*调用 IO 调度器中完成请求的函数，/block/elevator.c*/
                                       /*elevator_completed_req_fn()*/
    WARN_ON(req->bio != NULL);
```

```
    if (req->cmd_flags & REQ_ALLOCED) {        /*分配请求时设置 REQ_ALLOCED 标记位*/
        unsigned int flags = req->cmd_flags;
        struct request_list *rl = blk_rq_rl(req);

        BUG_ON(!list_empty(&req->queuelist));
        BUG_ON(ELV_ON_HASH(req));

        blk_free_request(rl, req);     /*释放请求实例，/block/blk-core.c*/
        freed_request(rl, flags);     /*修改请求队列中统计值，/block/blk-core.c*/
        blk_put_rl(rl);
    }
}
```

blk_free_request(rl, req)函数最后释放请求实例，放回 slab 缓存。

●blk_end_request_all()

blk_end_request_all()函数定义如下（/block/blk-core.c），调用此函数前不需要持有队列锁：

```
void blk_end_request_all(struct request *rq, int error)
{
    bool pending;
    unsigned int bidi_bytes = 0;

    if (unlikely(blk_bidi_rq(rq)))
        bidi_bytes = blk_rq_bytes(rq->next_rq);

    pending = blk_end_bidi_request(rq, error, blk_rq_bytes(rq), bidi_bytes);
    BUG_ON(pending);
}
```

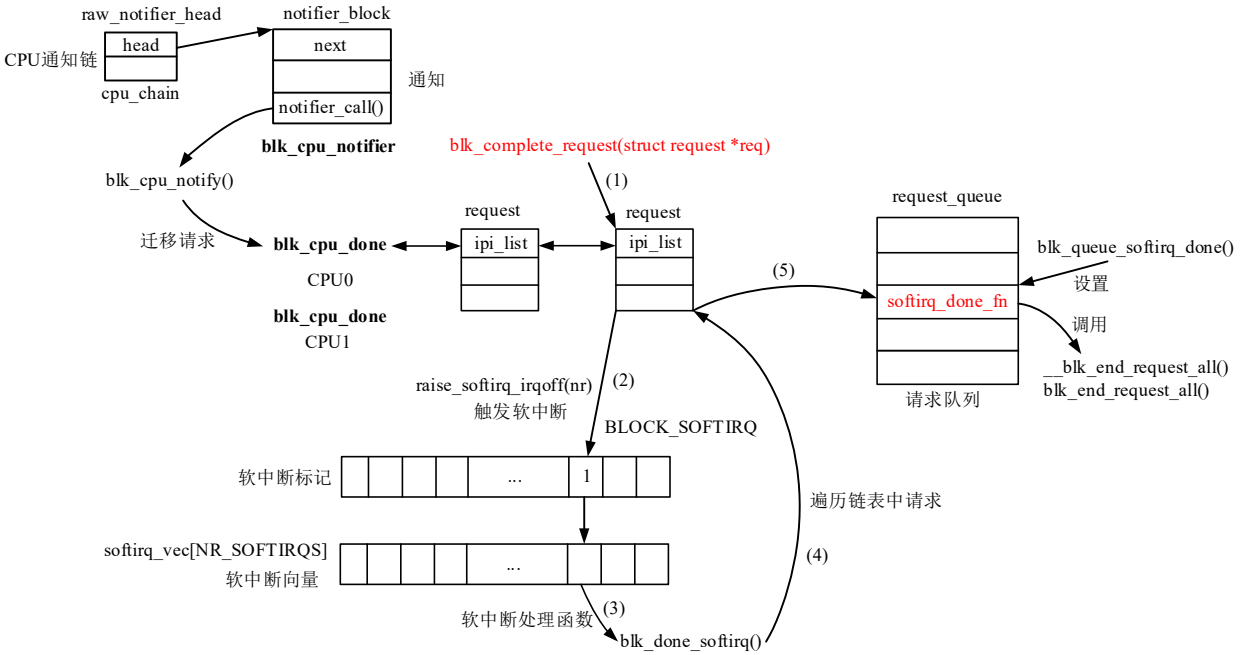
blk_end_bidi_request()函数定义如下，与__blk_end_bidi_request()函数类似，但函数内会持有锁：

```
static bool blk_end_bidi_request(struct request *rq, int error,unsigned int nr_bytes, unsigned int bidi_bytes)
{
    struct request_queue *q = rq->q;
    unsigned long flags;

    if (blk_update_bidi_request(rq, error, nr_bytes, bidi_bytes))
        return true;

    spin_lock_irqsave(q->queue_lock, flags);          /*持有队列锁*/
    blk_finish_request(rq, error);                    /*终结请求*/
    spin_unlock_irqrestore(q->queue_lock, flags);     /*释放队列锁*/

    return false;
}
```

■完成请求

前面介绍的__blk_end_request_all()/blk_end_request_all()等函数，在函数内即执行结束请求的工作，可认为是同步执行的。内核还提供了一种异步执行结束请求工作的机制，即在 BLOCK_SOFTIRQ 软中断中执行结束工作。

异步执行请求结束工作的机制称为完成请求，由 blk_complete_request(struct request *req)函数执行，如下图所示，结束请求的工作在队列 **softirq_done_fn()**函数中完成：



内核在/block/blk-softirq.c 文件内为每个 CPU 核定义了双链表 blk_cpu_done：

static DEFINE_PER_CPU(struct list_head, **blk_cpu_done**);

blk_complete_request(req)函数将请求插入到当前 CPU 核 blk_cpu_done 双链表末尾，如果链表中只有一个请求，则触发 BLOCK_SOFTIRQ 软中断，软中断处理函数为 **blk_done_softirq()**。

BLOCK_SOFTIRQ 软中断初始化函数定义如下（/block/blk-softirq.c）：

```
static __init int blk_softirq_init(void)
{
    int i;

    for_each_possible_cpu(i)
        INIT_LIST_HEAD(&per_cpu(blk_cpu_done, i));      /*初始化双链表*/

    open_softirq(BLOCK_SOFTIRQ, blk_done_softirq);      /*注册软中断*/
    register_hotcpu_notifier(&blk_cpu_notifier);        /*注册 CPU 通知*/
    return 0;
}
subsys_initcall(blk_softirq_init);
```

BLOCK_SOFTIRQ 软中断处理函数为 blk_done_softirq()，主要是扫描 CPU 核 blk_cpu_done 链表中的

请求，对每个请求从 blk_cpu_done 双链表移除，调用请求队列中的 softirq_done_fn(rq)函数处理请求，源代码请读者自行阅读。

void blk_queue_softirq_done(struct request_queue *q, softirq_done_fn ***fn**)接口函数用于设置请求队列中 softirq_done_fn()函数为 fn()。在 fn()函数中可调用__blk_end_request_all()/blk_end_request_all()等函数执行结束请求的工作，当然也可以执行其它工作。

BLOCK_SOFTIRQ 软中断初始化函数中还向 CPU 通知链注册了 blk_cpu_notifier 通知，其执行函数为 blk_cpu_notify()，主要是在 CPU 下线等时机，将 CPU 核对应 blk_cpu_done 双链表中的请求迁移到其它链表，并触发软中断（如果需要的话），源代码请读者自行阅读。

总之，完成请求的 blk_complete_request(rq)函数就是在软中断中调用请求队列的 **softirq_done_fn(rq)** 函数处理请求，在此函数中执行结束请求的工作。

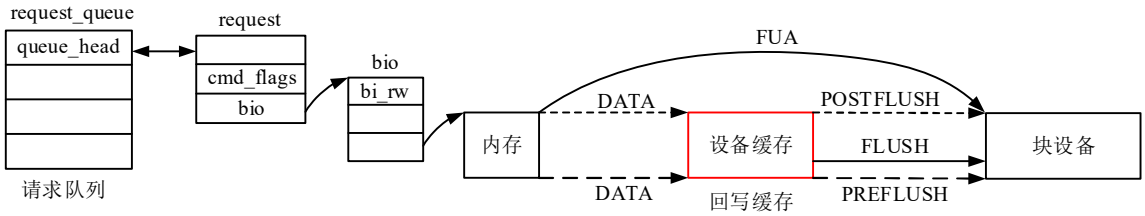
10.4.5 FLUSH/FUA 请求处理

前面介绍了标准请求队列对 bio 的处理流程，其中的 FLUSH/FUA 请求是比较特殊的请求，还没有详细介绍，下面讲解 FLUSH/FUA 请求及其处理。

1 概述

很多存储设备，特别是消费级市场，都自带板载 cache，称它为设备缓存或回写缓存。意思是在数据实际写入非易失性的存储介质前，存储设备就发送 IO 完成信号给 OS，也就是说还存在一个板级的块设备缓存，驱动程序只将数据传输到设备缓存。这样的好处是能更快地响应主机的 IO 请求，但为了确保数据落盘，就需要 OS 发送额外的命令以确保数据的完整。

如下图所示，FLUSH 和 FUA 可认为是两个动作，FLUSH 表示将设备缓存中数据写入块设备（同步），而 FUA 表示跳过设备缓存直接将请求中数据写入块设备。FLUSH 和 FUA 操作都需要硬件的支持，请求队列 **flush_flags** 成员标记了硬件（队列）是否支持 FLUSH（REQ_FLUSH）和 FUA（REQ_FUA），支持 FUA 时必须支持 FLUSH。



VFS 层需要执行 FLUSH 或 FUA 操作时，可设置 bio 实例 bi_rw 成员的 **REQ_FLUSH** 或 **REQ_FUA** 标记位，以此 bio 实例构建的请求称为 FLUSH/FUA 请求，bi_rw 成员相关标记位会传递给请求 cmd_flags 成员。FLUSH/FUA 请求只能包含一个或不包含 bio 实例，不能有多个 bio 实例。

FLUSH/FUA 请求的处理根据不同的情况要分多个步骤来进行，如下所示：

（1）如果不存在设备缓存，或者存在设备缓存但是硬件不支持 FLUSH 和 FUA（设备缓存对用户透明，队列 flush_flags 成员值为 0），FLUSH 和 FUA 没有区别：

①如果请求中不含需要传输的数据，FLUSH/FUA 请求什么也不用做，请求立即结束。

②如果请求中含需要传输的数据，FLUSH/FUA 请求被当成普通的请求插入队列处理，即只进行数据的传输（DATA 操作）。

（2）如果存在设备缓存，且硬件支持 FLUSH，但不支持 FUA（队列 flush_flags 成员值为 REQ_FLUSH）：

FLUSH 请求转换成 PREFLUSH 和 DATA 操作，PREFLUSH 操作其实就是 FLUSH 操作的别名。只不过在 DATA 操作之前进行，DATA 操作就是把请求当成普通的请求执行数据传输，即先 FLUSH 再进行数据传输（没有数据则不需要 DATA 操作）。

FUA 请求转换成 DATA 和 POSTFLUSH 操作，POSTFLUSH 操作也是 FLUSH 操作的别名，只不过是在 DATA 操作之后进行（没有数据则不需要 DATA 操作）。因为硬件不支持直接处理 REQ_FUA 请求，所以先把数据写入设备缓存，然后执行 FLUSH 来模拟 FUA 操作。

上面提到的 PREFLUSH 和 POSTFLUSH 其实都是 FLUSH 操作，FLUSH 操作通过向队列提交一个带 FLUSH 命令的请求来实现，后面再解释。

（3）如果存在设备缓存，且硬件支持 FLUSH 和 FUA（flush_flags 成员值为 REQ_FLUSH|REQ_FUA）：

FLUSH 请求转换成 PREFLUSH 和 DATA 操作（没有数据则不需要 DATA 操作）。

FUA 请求不需要转换，把它当成普通请求直接提交给请求队列，由驱动程序处理，相当于只执行 DATA 操作。

请求队列 request_queue 中 **flush_flags** 成员，表示硬件支持的 FLUSH 能力，取值为 0、REQ_FLUSH 或 REQ_FLUSH | REQ_FUA。如果支持 FLUSH，要设置 REQ_FLUSH 标记位。如果支持 FUA，必须支持 FLUSH，需设置 REQ_FLUSH 和 REQ_FUA 标记位。

void blk_queue_flush(struct request_queue *q, unsigned int flush)接口函数用于设置 **flush_flags** 成员值。

FLUSH/ FUA 请求处理分解出的步骤用位图中的比特位来标识，如下所示（/block/blk-flush.c）：

```
enum {
    REQ_FSEQ_PREFLUSH     = (1 << 0),     /*PREFLUSH 步骤，bit0*/
    REQ_FSEQ_DATA         = (1 << 1),     /*DATA 步骤，bit1*/
    REQ_FSEQ_POSTFLUSH    = (1 << 2),     /*POSTFLUSH 步骤，bit2*/
    REQ_FSEQ_DONE         = (1 << 3),     /*DONE 步骤执，可以结束请求了，bit3*/

    REQ_FSEQ_ACTIONS = REQ_FSEQ_PREFLUSH | REQ_FSEQ_DATA |
                       REQ_FSEQ_POSTFLUSH,          /*三个步骤的掩码*/

    /*挂起链表非空超时时长，若超时则强行执行 FLUSH，有请求在执行 DATA 时，会暂停 FLUSH*/
    FLUSH_PENDING_TIMEOUT    = 5 * HZ,
};
```

内核通过一个位图来标识请求需要执行的步骤，bit0 表示 PREFLUSH 步骤，bit1 表示 DATA 步骤，bit2 表示 POSTFLUSH 步骤，bit3 表示可以结束请求了。每个请求的处理以上步骤必须按顺序执行，但是并不一定需要执行所有的步骤。

blk_flush_policy()函数用于检查 FLUSH/FUA 请求需要执行哪几个步骤，也称它为请求处理的策略，函数定义如下（/block/blk-flush.c）：

```
static unsigned int blk_flush_policy(unsigned int fflags, struct request *rq)
/*fflags：request_queue 中 flush_flags 成员值，rq：请求指针*/
{
    unsigned int policy = 0;
```

```
    if (blk_rq_sectors(rq))
        policy |= REQ_FSEQ_DATA;        /*请求带数据，需 DATA 步骤*/

    if (fflags & REQ_FLUSH) {       /*如果队列支持 FLUSH*/
        if (rq->cmd_flags & REQ_FLUSH)      /*FLUSH 请求*/
            policy |= REQ_FSEQ_PREFLUSH;        /*需 PREFLUSH 步骤*/
        if (!(fflags & REQ_FUA) && (rq->cmd_flags & REQ_FUA))       /*队列不支持 FUA*/
            policy |= REQ_FSEQ_POSTFLUSH;       /*FUA 请求，需 POSTFLUSH 步骤*/
    }
    return policy;      /*返回需执行步骤的位图*/
}
```

由以上函数可知，函数返回值 policy 中置 1 的标记位表示请求处理时需要执行的步骤。如果队列不支持 FLUSH，则 FLUSH 和 FUA 请求只执行 DATA 步骤（如果有数据）；如果队列支持 FLUSH，FLUSH 请求需要 PREFLUSH 和 DATA 步骤；如果队列不支持 FUA，FUA 请求需 POSTFLUSH 和 DATA 步骤，若支持 FUA 则不需要 POSTFLUSH，当成普通请求处理（只需要 DATA 步骤）。

下面简要看一下 FLUSH/FUA 请求的处理流程，先看相关数据结构的定义：

请求 request 中与 FLUSH/FUA 请求管理相关的成员如下：

```
struct request {
    struct list_head    queuelist;     /*双链表元素，将实例添加到 request_queue.queue 双链表*/
    ...
    union {      /*联合体开始*/
        ...
        struct {            /*FLUSH/FUA 请求使用的成员*/
            unsigned int        seq;      /*请求处理需执行步骤的位图，为 0 表示要执行此步骤*/
            struct list_head    list;     /*将请求添加到 FLUSH 队列中的管理结构*/
            rq_end_io_fn        *saved_end_io;      /*保存请求原 end_io()函数指针*/
        } flush;
    };      /*联合体结束*/
    ...
}
```

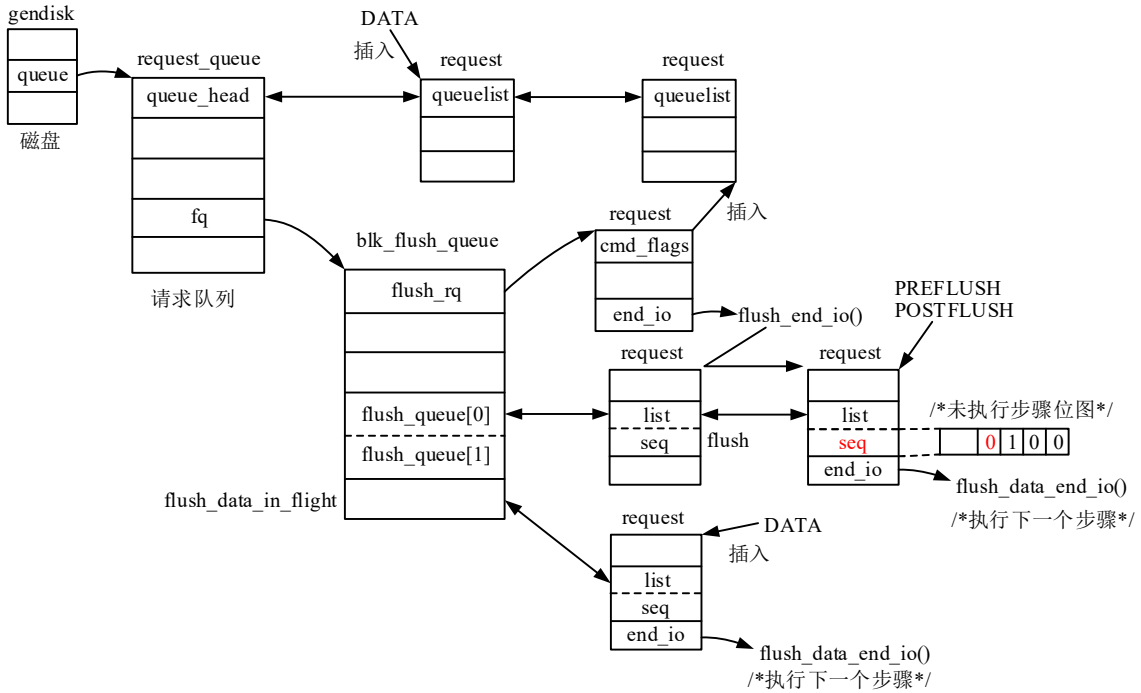
需要注意的是 seq 位图成员中比特位为 0 表示需要执行但还没有执行的步骤，比特位为 1 表示不需要执行或已经执行完了此步骤，每执行一个步骤就会置位 seq 中相应的位。处理请求时从低位往高位遍历位图，执行值为 0 比特位标识的步骤，位图中 bit3 始终为 0，因此 DONE（结束请求）总是最后一步。

FLUSH 队列用于管理 FLUSH/FUA 请求，由 blk_flush_queue 结构体表示，在创建标准请求队列时将创建 FLUSH 队列实例，并赋予请求队列 q->fq 成员，结构体定义如下（/block/blk.h）：

```
struct blk_flush_queue {
    unsigned int        flush_queue_delayed:1;
    unsigned int        flush_pending_idx:1;      /*当前挂起链表索引值，flush_queue[]*/
    unsigned int        flush_running_idx:1;      /*当前运行链表索引值，flush_queue[]*/
    unsigned long       flush_pending_since;      /*向空挂起链表插入第一个成员时的时间值*/
    struct list_head    flush_queue[2];         /*两个双链表，交替为挂起链表和运行链表*/
```

```
    struct list_head    flush_data_in_flight;     /*管理正在执行 DATA 步骤的请求*/
    struct request      *flush_rq;          /*指向用于执行 FLUSH 操作的请求，创建队列时创建*/
    struct request      *orig_rq;
    spinlock_t          mq_flush_lock;      /*保护 FLUSH 队列的自旋锁*/
};
```

下面结合下图 FLUSH/FUA 请求处理流程来解释 blk_flush_queue 结构体中各成员的用途：



内核首先会判断 FLUSH/FUA 请求处理需要执行哪些步骤，设置到 rq->flush.seq 成员中，对请求的处理就是扫描 rq->flush.seq 位图中比特位，按顺序执行各步骤（比特位为 0 表示要执行此步骤）。

blk_flush_queue 结构体中 flush_queue[2]成员是 2 个双链表，分别表示运行链表和挂起链表。当在执行 FLUSH 时，运行链表和挂起链表是交替的，也就是说 flush_queue[0]是运行链表时，flush_queue[1]就是挂起链表，flush_queue[0]是挂起链表时，flush_queue[1]就是运行链表。flush_running_idx 和 flush_pending_idx 成员表示运行链表和挂起链表的索引值，初始值都为 0。

请求在插入 FLUSH 队列前，end_io()函数重设为 flush_data_end_io()，并保存原函数指针，以此函数用于执行下一个步骤。

执行 PREFLUSH 或 POSTFLUSH 步骤时，将请求通过 rq->flush.list 成员插入到 FLUSH 队列的挂起链表，如果当前没有提交 flush_rq 请求（FLUSH 队列中 flush_rq 指向的请求），则转换挂起链表，设置 FLUSH 队列中 flush_rq 指向的请求，请求中包括 FLUSH 命令，end_io()函数设为 flush_end_io()函数，将 flush_rq 请求插入队列 q->queue_head 双链表末尾。块设备驱动程序处理 flush_rq 请求时，将执行 FLUSH 操作，请求处理完后调用 flush_end_io()函数。

flush_end_io()函数先获取原运行链表，然后转换运行链表，遍历原运行链表中需执行 PREFLUSH 或 POSTFLUSH 步骤的请求，执行其下一个步骤，也就是说原运行链表中所有请求的 FLUSH 步骤已经都由 flush_rq 请求代为执行了。

由于 flush_rq 请求只有一个，通过反复的提交来不断执行 FLUSH 操作。flush_rq 请求提交之前处于 PREFLUSH 或 POSTFLUSH 步骤的请求放入挂起链表，并作为请求结束时的运行链表，请求提交后转换挂起链表。请求结束时，执行请求提交前挂起链表，即运行链表中请求的下一个步骤，并转换运行链表。

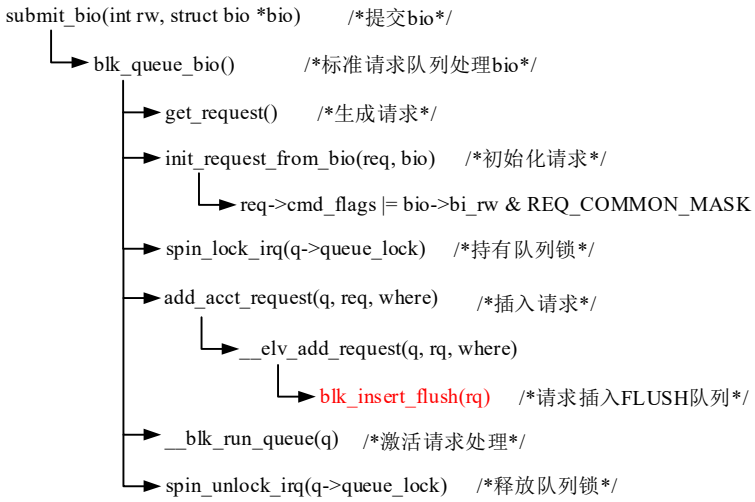
执行 DATA 步骤时，将请求插入队列 q->queue_head 双链表头部，同时通过 rq->flush.list 成员插入到 FLUSH 队列的 **fq->flush_data_in_flight** 双链表末尾，请求 end_io()函数修改为 flush_data_end_io()函数，在 DATA 步骤执行完数据传输时将调用此函数，此函数执行请求的下一个步骤（POSTFLUSH 或 DONE）。

当 FLUSH/FUA 请求执行到最后的 DONE 步骤时，执行结束请求的工作。

总之，对 FLUSH/FUA 请求依次执行各步骤，执行 PREFLUSH 或 POSTFLUSH 步骤时，请求插入当前挂起链表，设置并提交 flush_rq 请求，由其执行 FLUSH 操作，flush_rq 请求执行完后执行原挂起链表（当前运行链表）中请求的下一个步骤；执行 DATA 步骤时，请求插入 flush_data_in_flight 双链表，执行完 DATA 步骤后，执行后续的步骤；直至执行到 DONE 步骤，请求结束。

2 插入请求

下图示意了从 submit_bio()函数开始至请求插入 FLUSH 请求队列的函数调用关系：



blk_insert_flush()函数用于将 FLUSH/FUA 请求插入 FLUSH 队列，定义如下（/block/blk-flush.c）：

```
void blk_insert_flush(struct request *rq)
{
    struct request_queue *q = rq->q;
    unsigned int fflags = q->flush_flags;    /*请求队列是否支持 FLUSH 和 FUA*/
    unsigned int policy = blk_flush_policy(fflags, rq);        /*FLUSH 策略，需要执行哪些步骤，位图*/
    struct blk_flush_queue *fq = blk_get_flush_queue(q, rq->mq_ctx);        /*FLUSH 队列*/

    rq->cmd_flags &= ~REQ_FLUSH;        /*清零 REQ_FLUSH 标记位，所需步骤由 policy 表示*/
    if (!(fflags & REQ_FUA))                /*队列（硬件）不支持 FUA*/
        rq->cmd_flags &= ~REQ_FUA;        /*清零 REQ_FUA 标记位*/

    if (!policy) {                /*策略为空，结束请求*/
        if (q->mq_ops)
            blk_mq_end_request(rq, 0);
        else
            __blk_end_bidi_request(rq, 0, 0, 0);
        return;
    }
```

```
    BUG_ON(rq->bio != rq->biotail);    /*rq->bio 链表只能为空或只有一个 bio */

    /*请求只有 DATA 步骤，添加到请求队列中双链表，当作普通请求处理，函数返回*/
    if ((policy & REQ_FSEQ_DATA) &&
        !(policy & (REQ_FSEQ_PREFLUSH | REQ_FSEQ_POSTFLUSH))) {
        if (q->mq_ops) {          /*Multi queue 请求队列*/
            blk_mq_insert_request(rq, false, false, true);
        } else        /*标准请求队列*/
            list_add_tail(&rq->queuelist, &q->queue_head);   /*请求插入 q->queue_head 双链表末尾*/
        return;       /*函数返回*/
    }

    /*请求包含 PREFLUSH 或 POSTFLUSH 步骤*/
    memset(&rq->flush, 0, sizeof(rq->flush));       /*清零请求内嵌 flush 结构体成员（联合体成员）*/
    INIT_LIST_HEAD(&rq->flush.list);            /*初始化双链表成员*/
    rq->cmd_flags |= REQ_FLUSH_SEQ;         /*标记需 FLUSH 操作，由驱动程序处理*/
    rq->flush.saved_end_io = rq->end_io;    /*保存原 end_io()函数，通常为 NULL，结束请求时恢复*/
    if (q->mq_ops) {                /*Multi queue 请求队列*/
        rq->end_io = mq_flush_data_end_io;

        spin_lock_irq(&fq->mq_flush_lock);
        blk_flush_complete_seq(rq, fq, REQ_FSEQ_ACTIONS & ~policy, 0);
        spin_unlock_irq(&fq->mq_flush_lock);
        return;
    }
    /*以下适用于标准请求队列*/
    rq->end_io = flush_data_end_io;    /*请求结束回调函数中执行下一个步骤，/block/blk-flush.c*/
    blk_flush_complete_seq(rq, fq, REQ_FSEQ_ACTIONS & ~policy, 0);
                                        /*执行请求中第一个需要执行的步骤*/
}
```

blk_insert_flush()函数首先判断请求需要执行的步骤，保存至 **policy** 变量（位图），如果 policy 值为 0，立即结束请求，如果只包含 DATA 步骤，则将请求当作普通请求处理。

如果包含 PREFLUSH 或 POSTFLUSH 步骤，则需要设置请求，如 end_io 函数设为 flush_data_end_io()，最后调用 blk_flush_complete_seq()函数执行请求中第一个需要执行的步骤，第一个步骤执行完后将触发下一个步骤的执行，直至 DONE 步骤，详见下文。

3 处理请求

blk_flush_complete_seq()函数用于执行 FLUSH/FUA 请求中第一个需要执行的步骤，参数 seq 中为 1 的标记位表示要屏蔽的步骤（或者说标记已经完成的步骤），函数代码如下（/block/blk-flush.c）：

```
static bool blk_flush_complete_seq(struct request *rq,struct blk_flush_queue *fq,unsigned int seq, int error)
/*error：是否发生错误*/
{
```

```
    struct request_queue *q = rq->q;
    struct list_head *pending = &fq->flush_queue[fq->flush_pending_idx];      /*挂起链表*/
    bool queued = false, kicked;

    BUG_ON(rq->flush.seq & seq);
    rq->flush.seq |= seq;        /*标记 seq 中的步骤已经完成，为 0 的比特位表示尚未执行的步骤*/
                                 /*rq->flush.seq 初始为 0，DONE（bit3）始终为 0*/
    if (likely(!error))
        seq = blk_flush_cur_seq(rq);          /*第一个要执行的步骤（为 0 的位），/block/blk-flush.c*/
    else
        seq = REQ_FSEQ_DONE;                  /*error 非 0，表示出错，结束请求*/

    switch (seq) {
    case REQ_FSEQ_PREFLUSH:
    case REQ_FSEQ_POSTFLUSH:
        if (list_empty(pending))          /*挂起双链表为空*/
            fq->flush_pending_since = jiffies;          /*设置挂起双链表等待时间起点为当前时间*/
        list_move_tail(&rq->flush.list, pending);       /*插入挂起双链表末尾*/
        break;

    case REQ_FSEQ_DATA:         /*DATA*/
        list_move_tail(&rq->flush.list, &fq->flush_data_in_flight);     /*插入 flush_data_in_flight 链表*/
        queued = blk_flush_queue_rq(rq, true);      /*请求插入请求队列 q->queue_head 双链表头部*/
        break;

    case REQ_FSEQ_DONE:           /*结束请求*/
        BUG_ON(!list_empty(&rq->queuelist));
        list_del_init(&rq->flush.list);          /*从 FLUSH 队列移出*/
        blk_flush_restore_request(rq);       /*恢复请求中成员值，如 end_io()函数，/block/blk-flush.c*/
        if (q->mq_ops)                       /*结束请求*/
            blk_mq_end_request(rq, error);
        else
            __blk_end_request_all(rq, error);
        break;

    default:
        BUG();
    }

    kicked = blk_kick_flush(q, fq);       /*是否要设置并提交 flush_rq 请求，/block/blk-flush.c*/
    return kicked | queued;         /*返回是否向 q->queue_head 双链表插入了请求*/
}
```

blk_flush_complete_seq()函数通过 rq->flush.seq 成员获取第一个需要执行的步骤，然后执行以下操作：

（1）如果是 PREFLUSH 或 POSTFLUSH 步骤，则将请求插入 FLUSH 队列挂起双链表。

（2）如果是 DATA 步骤，则将请求插入 FLUSH 队列 flush_data_in_flight 双链表末尾和标准请求队列 q->queue_head 双链表头部。

（3）如果是 DONE 步骤，则结束请求。

（4）调用 blk_kick_flush(q, fq)函数判断是否要设置并提交 flush_rq 请求。

blk_flush_complete_seq()函数最后返回是否向 q->queue_head 双链表插入了请求，flush_rq 请求也算。

■flush_rq 请求

下面看一下设置并提交 flush_rq 请求的条件，以及 flush_rq 请求的处理。flush_rq 请求在创建 FLUSH 请求队列时创建，用于向驱动程序发送一个包含 FLUSH 命令的请求，以执行 FLUSH 操作。

blk_kick_flush()函数定义如下（/block/blk-flush.c）：

```
static bool blk_kick_flush(struct request_queue *q, struct blk_flush_queue *fq)
{
    struct list_head *pending = &fq->flush_queue[fq->flush_pending_idx];        /*挂起链表*/
    struct request *first_rq =list_first_entry(pending, struct request, flush.list);    /*第一个挂起请求*/
    struct request *flush_rq = fq->flush_rq;            /*flush_rq 请求*/

    /*挂起链表为空或（挂起与运行链表索引值不同，表示已提交了 flush_rq 请求）*/
    /*不提交 flush_rq 请求，函数返回*/
    if (fq->flush_pending_idx != fq->flush_running_idx || list_empty(pending))
        return false;

    /*flush_data_in_flight 非空，表示有请求在执行 DATA，则等其执行完，不提交 flush_rq 请求*/
    /*以便在随后的 FLUSH 操作中将新写入数据刷出设备*/
    /*但也设置了一个等待时长，如果挂起链表非空时长超时，也将提交 flush_rq 请求，执行 FLUSH*/
    if (!list_empty(&fq->flush_data_in_flight) &&time_before(jiffies,
                fq->flush_pending_since + FLUSH_PENDING_TIMEOUT))
        return false;

    /*需要提交 flush_rq 请求*/
    fq->flush_pending_idx ^= 1;      /*挂起链表转换*/

    blk_rq_init(q, flush_rq);            /*初始化 flush_rq 请求*/

    if (q->mq_ops) {                /*Multi queue 请求队列*/
        struct blk_mq_hw_ctx *hctx;

        flush_rq->mq_ctx = first_rq->mq_ctx;
        flush_rq->tag = first_rq->tag;
        fq->orig_rq = first_rq;

        hctx = q->mq_ops->map_queue(q, first_rq->mq_ctx->cpu);
```

```
        blk_mq_tag_set_rq(hctx, first_rq->tag, flush_rq);
    }

    flush_rq->cmd_type = REQ_TYPE_FS;
    flush_rq->cmd_flags = WRITE_FLUSH | REQ_FLUSH_SEQ;      /*设置 FLUSH 命令*/
    flush_rq->rq_disk = first_rq->rq_disk;
    flush_rq->end_io = flush_end_io;          /*结束请求回调函数*/

    return blk_flush_queue_rq(flush_rq, false);
                                /*flush_rq 请求插入请求队列 q->queue_head 双链表末尾，返回 true*/
}
```

blk_kick_flush()函数如果设置并提交了 flush_rq 请求，块设备驱动程序在处理 flush_rq 请求时，将执行 FLUSH 操作，因为 flush_rq 请求 cmd_flags 成员中设置了相应的命令标记。

flush_rq 请求处理完成后，将调用其中的 end_io()函数，即 flush_end_io()函数，定义如下：

```
static void flush_end_io(struct request *flush_rq, int error)
{
    struct request_queue *q = flush_rq->q;
    struct list_head *running;
    bool queued = false;
    struct request *rq, *n;
    unsigned long flags = 0;
    struct blk_flush_queue *fq = blk_get_flush_queue(q, flush_rq->mq_ctx);        /*FLUSH 队列*/

    if (q->mq_ops) {          /*Multi queue 请求队列*/
        struct blk_mq_hw_ctx *hctx;
        spin_lock_irqsave(&fq->mq_flush_lock, flags);
        hctx = q->mq_ops->map_queue(q, flush_rq->mq_ctx->cpu);
        blk_mq_tag_set_rq(hctx, flush_rq->tag, fq->orig_rq);
        flush_rq->tag = -1;
    }

    running = &fq->flush_queue[fq->flush_running_idx];        /*获取当前运行链表*/
    BUG_ON(fq->flush_pending_idx == fq->flush_running_idx);

    fq->flush_running_idx ^= 1;          /*运行链表转换*/

    if (!q->mq_ops)
        elv_completed_request(q, flush_rq);      /*调用 IO 调度器的完成请求函数，/block/elevator.c*/

    /*遍历原运行队列中请求，执行请求中下一个步骤*/
    list_for_each_entry_safe(rq, n, running, flush.list) {
        unsigned int seq = blk_flush_cur_seq(rq);
```

```
                                        /*第一个要执行的步骤，PREFLUSH 或 POSTFLUSH*/

            BUG_ON(seq != REQ_FSEQ_PREFLUSH && seq != REQ_FSEQ_POSTFLUSH);
            queued |= blk_flush_complete_seq(rq, fq, seq, error);
                                        /*屏蔽 PREFLUSH 或 POSTFLUSH 步骤，执行下一个步骤*/
        }
        /*有请求插入了请求队列*/
        if (queued || fq->flush_queue_delayed) {
            WARN_ON(q->mq_ops);
            blk_run_queue_async(q);          /*异步触发请求处理，/block/blk-core.c*/
        }
        fq->flush_queue_delayed = 0;
        if (q->mq_ops)
            spin_unlock_irqrestore(&fq->mq_flush_lock, flags);
    }
```

flush_rq 请求处理完成后代表已经执行了 FLUSH 操作，然后遍历原运行队列中请求，这些请求是要执行 PREFLUSH 或 POSTFLUSH 步骤的，因为已经由 flush_rq 请求代为执行了，因此可以执行这些请求的下一个步骤了，这也由 blk_flush_complete_seq()函数执行。如果在执行下一个步骤中也向请求队列插入了请求，将异步触发请求的处理。

■DATA 步骤

请求执行 DATA 步骤时，将插入 FLUSH 队列 flush_data_in_flight 双链表末尾和请求队列 q->queue_head 双链表头部，数据传输完成后，将调用请求的 end_io()回调函数，此时为 **flush_data_end_io()**函数。

flush_data_end_io()回调函数代码如下（/block/blk-flush.c）：

```
static void flush_data_end_io(struct request *rq, int error)
{
    struct request_queue *q = rq->q;
    struct blk_flush_queue *fq = blk_get_flush_queue(q, NULL);

    if (blk_flush_complete_seq(rq, fq, REQ_FSEQ_DATA, error))      /*执行 DATA 之后的步骤*/
        blk_run_queue_async(q);          /*如果有请求插入请求队列，异步触发请求处理*/
}
```

flush_data_end_io()函数调用 blk_flush_complete_seq()函数执行请求 DATA 步骤之后的下一个步骤，如果有请求插入了请求队列将调用 blk_run_queue_async(q)异步触发请求处理。

blk_run_queue_async()函数定义如下（/block/blk-core.c）：

```
void blk_run_queue_async(struct request_queue *q)
{
    if (likely(!blk_queue_stopped(q) && !blk_queue_dead(q)))
        mod_delayed_work(kblockd_workqueue, &q->delay_work, 0);
                                    /*延时工作中调用__blk_run_queue(q)函数处理请求*/
}
```

至此，FLUSH/FUA 请求的处理就介绍完了。简单地说就是将请求的处理划分成 4 个步骤，依次执行这 4 个步骤，每个步骤都由 blk_flush_complete_seq()函数执行。对于 PREFLUSH 或 POSTFLUSH 步骤，是将请求缓存，由 flush_rq 请求代为执行 FLUSH 操作，执行完后再执行请求的下一个步骤。

执行 DATA 步骤就是将请求插入 q->queue_head 双链表，请求处理完后，在请求结束的回调函数中执行下一个步骤。DONE 步骤就是结束请求。

请求 rq->flush.seq 成员是一个位图，依次标记 PREFLUSH、DATA、POSTFLUSH 和 DONE 步骤，比特位为 0 表示需要执行此步骤，为 1 表示不需要执行此步骤或者已经执行完了此步骤。处理请求时从低到高位扫描 rq->flush.seq 位图，对为 0 的比特位执行相应的步骤，执行完后置位。由于 DONE 步骤标记位（bit3）始终为 0，因此最后一个步骤必定是 DONE，即结束请求。

10.4.6 内存盘请求队列

基于外部存储介质的块设备，由于访问速度较慢，因此需要将访问（由 bio 封装）用队列进行缓存（请求），随后逐一进行处理。

然而，对于基于内存盘的虚拟块设备（把内存当成块设备），因为对内存的访问可直接进行，不需要等待，因此提交的 bio 可以直接立即处理，而不需要缓存，也就不需要创建请求了。

VFS 层向请求队列提交 bio 时，会调用请求队列的 make_request_fn()函数，通常用于创建请求（关联 bio），将请求插入请求队列，并激活请求的处理。对于基于内存盘的块设备，在 make_request_fn()函数中就可以直接进行数据的传输，不需要创建请求，更不需要插入请求队列。

基于内存盘块设备的请求队列创建流程简列如下：

（1）调用 struct request_queue *blk_alloc_queue_node(gfp_t gfp_mask, int node_id)函数创建请求队列。

（2）调用 void blk_queue_make_request(struct request_queue *q, make_request_fn ***mfn**)函数设置请求队列，在 mfn()函数中实现数据的传输。

10.5 Multi queue 请求队列

VFS 层在向标准请求队列提交 bio 实例创建请求，从队列分配和插入请求等时机，都需要持有标准请求队列的自旋锁（q->queue_lock）。多核多进程系统执行 IO 传输时，对 q->queue_lock 锁的竞争将会比较激烈，在锁上自旋等待将浪费不少时间，为此内核引入 Multi queue 请求队列来解决这个问题（主要用于提高多核系统 SSD 磁盘的访问效率）。

Multi queue 请求队列中每个 CPU 核有自己的软件队列，另外还有发送（处理）请求的硬件队列，提交请求时先尝试添加到硬件队列，如果不成功则添加到软件队列，随后由驱动程序移入硬件队列处理，这样进程在提交请求时就不需要与其它 CPU 核竞争队列锁了。

10.5.1 概述

1 队列结构

在 Multi queue 请求队列中，每个 CPU 核对应一个软件队列（percpu 变量），进程在操作当前 CPU 核的软件队列时，不需要与其它 CPU 核竞争锁。队列中还包含至少一个硬件队列（用于实现硬件上的并行操作？？？），每个软件队列（CPU 核）映射唯一的硬件队列，软件队列中的请求将移入映射的硬件队列，由它进行处理。不同软件队列可映射同一个硬件队列，也就是说一个硬件队列可接受多个软件队列移入的请求。

进程提交请求时，先尝试提交到当前 CPU 核映射的硬件队列，如果不成功则提交到当前 CPU 核对应

的软件队列，以避免多核之间竞争锁，提高传输效率，在软件队列中可以合并请求。

软件队列由 blk_mq_ctx 结构体表示，每个 CPU 核对应一个实例（percpu 变量）。硬件队列由结构体 blk_mq_hw_ctx 表示，至少需要一个硬件队列。每个硬件队列关联一个 blk_mq_tag 结构体实例，表示队列标签，用于管理和分配请求。

Multi queue 请求队列也由 request_queue 结构体表示，相关成员如下：

```
struct request_queue {
    ...
    /*Multi-queue 相关成员*/
    struct blk_mq_ops    *mq_ops;          /*Multi queue 队列操作结构，/include/inux/blk-mq.h*/
    unsigned int          *mq_map;        /*软件队列与硬件队列的映射关系，整数数组*/
    /*软件队列*/
    struct blk_mq_ctx    __percpu*queue_ctx;        /*软件队列 percpu 指针*/
    unsigned int          nr_queues;                 /*软件队列数量*/
    /*硬件（发送）队列*/
    struct blk_mq_hw_ctx    **queue_hw_ctx;       /*指向硬件队列指针数组*/
    unsigned int    nr_hw_queues;          /*硬件队列数量*/
    ...
    struct kobject    mq_kobj;          /*跟踪 mq queue 请求队列的 kobject 实例*/
    ...
    struct rcu_head           rcu_head;
    wait_queue_head_t         mq_freeze_wq;
    struct percpu_ref         mq_usage_counter;
    struct list_head          all_q_node;         /*用于将实例添加到全局 all_q_list 双链表*/

    struct blk_mq_tag_set    *tag_set;          /*指向 blk_mq_tag_set 实例*/
    struct list_head      tag_set_list;          /*添加到 blk_mq_tag_set.tag_list 双链表*/

}
```

request_queue 结构体中相关成员简介如下：

■**queue_ctx**：percpu 变量，指向 blk_mq_ctx 结构体，表示软件队列，每个 CPU 核对应一个实例。

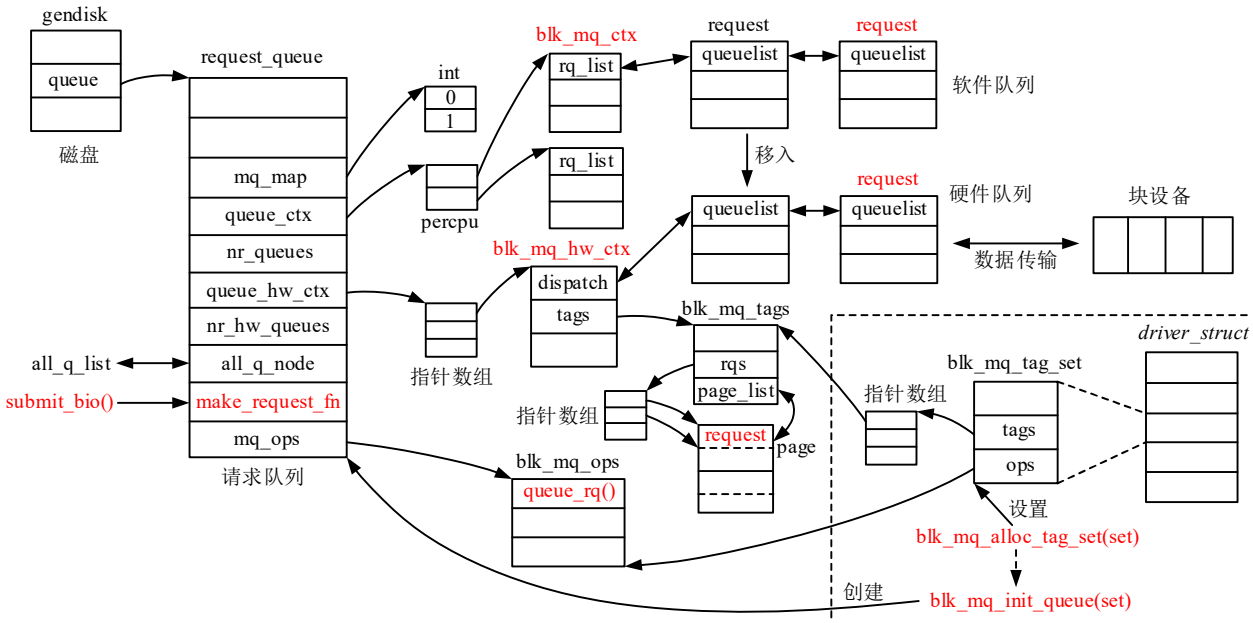
■**queue_hw_ctx**：指向指针数组，数组项指向 blk_mq_hw_ctx 结构体实例，表示硬件队列，至少需要有一个硬件队列。数组项索引值就是硬件队列的编号。

■**mq_map**：指向整数数组，数组项索引值（项数）表示软件队列编号（CPU 核编号），数组项值表示软件队列（CPU 核）映射的硬件队列编号，例如：mq_map[0]值表示 0 号软件队列（CPU0）映射的硬件队列编号。

■**mq_ops**：指向 blk_mq_ops 结构体，它是 Multi-queue 队列操作结构，执行请求的入队（硬件队列）、初始化硬件队列等操作。Multi queue 队列中没有使用 IO 调度器，也没有使用队列中的 request_fn()函数，请求的入队和处理都由 blk_mq_ops 结构体中函数完成。

■**all_q_node**：将 Multi queue 队列 request_queue 实例添加到全局双链表 all_q_list。

Multi queue 请求队列结构如下图所示：



硬件队列 blk_mq_hw_ctx 结构体中 tags 成员指向 **blk_mq_tags** 结构体，表示队列标签，主要用于管理和分配 request 实例。blk_mq_tags 结构体中 page_list 双链表链接的是内存块首页对应的 page 实例，内存块用于保存 request 实例，rqs 指向的指针数组指向这些请求，也就是说在创建硬件队列时就已经为其创建了请求实例，只是没有分配出去，没有关联到 bio 实例，也没有插入队列。blk_mq_tags 结构体中还包含一个位图，每个比特位对应一个 request 实例，表示实例是否已分配出去（分配则置 1），比特位在位图中的位置表示 request 实例的标签值（整数）。

blk_mq_hw_ctx 结构体中 dispatch 双链表管理插入硬件队列的请求 request 实例。

软件队列由 queue_ctx 结构体表示，其中 rq_list 双链表管理插入软件队列的请求，软件队列中请求将移入到映射的硬件队列中，再进行处理。mq_map 指向的整型数组表示软件队列映射的硬件队列编号。

Multi queue 队列的 make_request_fn()函数为 **blk_mq_make_request**()或 blk_sq_make_request()，接口函数 submit_bio()函数向提交的 bio，将由此函数处理。make_request_fn()函数将在当前 CPU 核映射硬件队列标签中分配 request 实例并关联 bio，请求实例将由 blk_mq_ops 实例中 **queue_rq()**函数尝试插入硬件队列，并触发请求的处理。若插入硬件队列不成功则插入软件队列，软件队列中请求随后由驱动程序移入映射的硬件队列，并执行请求。

在创建 Multi queue 队列前，需要为其下硬件队列创建队列标签 blk_mq_tags 实例。创建队列标签的函数需要一个 blk_mq_tag_set 结构体实例作为参数，用于传递队列参数以及块设备的硬件配置信息。

blk_mq_tag_set 结构体将管理所有创建的队列标签，并作为创建 Multi queue 队列函数的参数，也将管理依此实例创建的 Multi queue 队列。

blk_mq_tag_set 结构体关联的 blk_mq_ops 实例，也将传递给创建的 Multi queue 队列，blk_mq_ops 实例必须由具体块设备驱动程序定义。

块设备驱动程序中创建 Multi queue 请求队列的流程如下（请求队列需赋予 gendisk 实例）：

（1）定义 blk_mq_ops 结构体实例，定义并设置 blk_mq_tag_set 结构体实例（关联 blk_mq_ops 实例），用于传递请求队列以及块设备的硬件配置信息。

（2）调用 **blk_mq_alloc_tag_set**(struct blk_mq_tag_set *set)函数，参数是设置好的 blk_mq_tag_set 实例指针，为硬件队列创建队列标签 blk_mq_tags 实例。

（3）调用 **blk_mq_init_queue**(struct blk_mq_tag_set ***set**)函数（参数同（2）），创建 Multi queue 请求

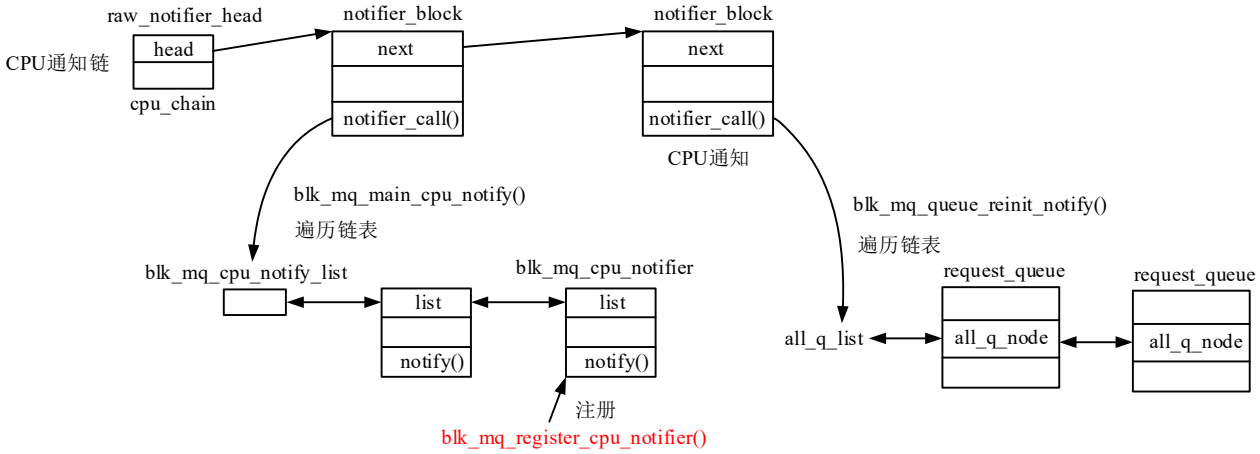
队列，函数返回 request_queue 实例指针。

本节先介绍相关数据结构的定义，然后介绍创建 Multi queue 队列相关函数的实现，以及队列对提交 bio 的处理。

2 注册 CPU 通知

Multi queue 队列与 CPU 核相关，因此在 CPU 核热插拔时有些工作需要做，例如，在 CPU 核下线时，需要将对应软件队列中的请求移到其它 CPU 核的软件队列中，并最终迁入硬件队列。

Multi queue 队列公共层代码在 CPU 通知链中注册了通知，如下图所示：



blk_mq_main_cpu_notify 通知执行函数扫描 blk_mq_cpu_notify_list 双链表中 blk_mq_cpu_notifier 实例，调用其 notify()函数。blk_mq_queue_reinit_notify 通知执行函数扫描 all_q_list 双链表中 Multi queue 队列，用于冻结、重新初始化队列等（按需进行）。

内核在/block/blk-mq.c 文件内定义了 Multi queue 队列 CPU 通知的初始化函数，如下所示：

```
static int __init blk_mq_init(void)
{
    blk_mq_cpu_init();      /*注册 blk_mq_main_cpu_notify 通知，见下文*/
    hotcpu_notifier(blk_mq_queue_reinit_notify, 0);   /*注册 blk_mq_queue_reinit_notify 通知*/
    return 0;
}
subsys_initcall(blk_mq_init);
```

blk_mq_queue_reinit_notify 通知的执行函数请读者自行阅读源代码。blk_mq_main_cpu_notify 通知执行函数中处理的 blk_mq_cpu_notifier 实例定义如下（ /include/linux/blk-mq.h）：

```
struct blk_mq_cpu_notifier {
    struct list_head list;
    void *data;
    int (*notify)(void *data, unsigned long action, unsigned int cpu);      /*处理函数*/
};
```

内核在/block/blk-mq-cpu.c 文件内定义了全局双链表 blk_mq_cpu_notify_list 管理 blk_mq_cpu_notifier 实例。

blk_mq_init_cpu_notifier()函数用于设置 blk_mq_cpu_notifier 实例，**blk_mq_register_cpu_notifier()**函数用于将实例添加到 blk_mq_cpu_notify_list 双链表末尾。

初始化函数 blk_mq_cpu_init()调用 hotcpu_notifier()函数向 CPU 通知链注册通知，通知执行函数将遍历

blk_mq_cpu_notify_list 双链表中 blk_mq_cpu_notifier 实例，调用其 notify()函数。

3 数据结构

Multi queue 队列中主要的数据结构有 blk_mq_hw_ctx、blk_mq_ctx 和 blk_mq_ops 等，下面分别介绍这些数据结构的定义。

■blk_mq_hw_ctx

硬件队列由 blk_mq_hw_ctx 结构体表示，定义如下（/include/linux/blk-mq.h）：

```
struct blk_mq_hw_ctx {
    struct {
        spinlock_t          lock;          /*保护本硬件队列的自旋锁*/
        struct list_head    dispatch;      /*双链表头，管理请求 request 实例*/
    } ____cacheline_aligned_in_smp;

    unsigned long       state;              /*状态，BLK_MQ_S_* flags */
    struct delayed_work     run_work;       /*延时工作*/
    struct delayed_work     delay_work;
    cpumask_var_t           cpumask;        /*CPU 核位图，支持的 CPU 核（软件队列）*/
    int             next_cpu;
    int             next_cpu_batch;
    unsigned long       flags;              /*标记，BLK_MQ_F_* flags */

    struct request_queue    *queue;         /*指向所属请求队列*/
    struct blk_flush_queue  *fq;            /*指向 FLUSH 队列*/
    void            *driver_data;           /*驱动私有数据*/
    struct blk_mq_ctxmap    ctx_map;        /*关联软件队列的位图，/include/linux/blk-mq.h*/

    unsigned int        nr_ctx;             /*关联的软件队列的数量*/
    struct blk_mq_ctx   **ctxs;             /*指向关联软件队列指针数组*/

    atomic_t        wait_index;
    struct blk_mq_tags  *tags;      /*blk_mq_tags 结构体指针，队列标签*/

    unsigned long       queued;     /*队列中入队请求数量（需要处理的请求数量）*/
    unsigned long       run;
#define BLK_MQ_MAX_DISPATCH_ORDER  10
    unsigned long       dispatched[BLK_MQ_MAX_DISPATCH_ORDER];      /*请求统计量*/
    unsigned int        numa_node;          /*关联的内存节点*/
    unsigned int        queue_num;          /*硬件队列编号*/
    atomic_t        nr_active;
```

```
    struct blk_mq_cpu_notifier    cpu_notifier;          /*在 CPU 通知中执行的函数，迁移请求*/
    struct kobject        kobj;        /*跟踪硬件队列的 kobject 实例，队列 queue->mq_kobj 为父节点*/
};
```

blk_mq_hw_ctx 结构体主要成员简介如下：

●**dispatch**：硬件队列请求双链表，要处理的请求。

●**ctxs**：指向指针数组，数组项指向关联的软件队列 blk_mq_ctx 实例。

●**ctx_map**：blk_mq_ctxmap 结构体成员，主要包含一个关联软件队列的位图，标记软件队列是否有请求需要移入本硬件队列。

●**fq**：指向 blk_flush_queue 结构体，表示 FLUSH 队列。

●**tags**：指向 blk_mq_tags 结构体，队列标签，用于管理请求，每个请求赋予一个由整型数表示的标签值，下一小节将详介绍此结构体。

●**kobj**：kobject 结构体成员，跟踪硬件队列，导出到 sysfs，队列 queue->mq_kobj 为其父节点。

●**state、flags**：状态、标记成员，取值如下（/include/linux/blk-mq.h）：

```
enum {
    BLK_MQ_RQ_QUEUE_OK = 0,        /*队列 OK*/
    BLK_MQ_RQ_QUEUE_BUSY    = 1,   /* requeue IO for later */
    BLK_MQ_RQ_QUEUE_ERROR = 2,   /* end IO with error */

    BLK_MQ_F_SHOULD_MERGE = 1 << 0,      /*flags 标记成员值*/
    BLK_MQ_F_TAG_SHARED      = 1 << 1,
    BLK_MQ_F_SG_MERGE    = 1 << 2,
    BLK_MQ_F_SYSFS_UP      = 1 << 3,
    BLK_MQ_F_DEFER_ISSUE      = 1 << 4,
    BLK_MQ_F_ALLOC_POLICY_START_BIT = 8,
    BLK_MQ_F_ALLOC_POLICY_BITS = 1,

    BLK_MQ_S_STOPPED      = 0,       /*state 状态成员值，队列停止了*/
    BLK_MQ_S_TAG_ACTIVE= 1,

    BLK_MQ_MAX_DEPTH     = 10240,          /*Multi queue 队列最大队列深度*/
    BLK_MQ_CPU_WORK_BATCH      = 8,
};
```

■blk_mq_ctx

软件队列由 blk_mq_ctx 结构体表示，定义如下（/block/blk-mq.h）：

```
struct blk_mq_ctx {
    struct {
        spinlock_t          lock;          /*保护自旋锁*/
        struct list_head    rq_list;       /*双链表头，管理请求 request 实例*/
    }    ____cacheline_aligned_in_smp;

    unsigned int        cpu;            /*软件队列对应的 CPU 核编号*/
```

```
    unsigned int        index_hw;       /*映射硬件队列编号*/

    unsigned int        last_tag    ____cacheline_aligned_in_smp;

    /* incremented at dispatch time */
    unsigned long       rq_dispatched[2];
    unsigned long       rq_merged;

    /* incremented at completion time */
    unsigned long       ____cacheline_aligned_in_smp rq_completed[2];

    struct request_queue    *queue;         /*指向所属请求队列*/
    struct kobject      kobj;               /*跟踪软件队列的 kobject 实例，添加到硬件队列为父节点*/
} ____cacheline_aligned_in_smp;
```

blk_mq_ctx 结构体主要成员简介如下：

●**rq_list**：双链表头，管理请求实例。

●**index_hw**：映射硬件队列编号。

●**kobj**：跟踪软件队列的 kobject 结构体成员，导出到 sysfs，其父节点为映射硬件队列的 kobject 结构体成员。

■blk_mq_ops

blk_mq_ops 表示 Multi queue 队列操作结构，由具体块设备驱动程序实现，定义如下：

```
struct blk_mq_ops {                     /*/include/linux/blk-mq.h*/
    queue_rq_fn     *queue_rq;      /*将请求插入硬件队列（处理请求），必须定义的函数*/
    map_queue_fn    *map_queue;     /*获取 CPU 核映射的硬件队列，必须定义的函数 */
    timeout_fn      *timeout;       /*请求超时时调用*/
    softirq_done_fn *complete;    /*完成请求时，由软中断调用的函数，赋予 request_queue 实例*/
    init_hctx_fn    *init_hctx;     /*初始化硬件队列函数*/
    exit_hctx_fn    *exit_hctx;     /*驱动移除时调用*/
    init_request_fn     *init_request;      /*分配请求时调用的初始化函数*/
    exit_request_fn     *exit_request;
};
```

blk_mq_ops 结构体中主要包含一些硬件队列和请求操作函数指针，原型定义在/include/linux/blk-mq.h 头文件，如下所示：

```
typedef int (queue_rq_fn)(struct blk_mq_hw_ctx *, const struct blk_mq_queue_data *);    /*入队*/
typedef struct blk_mq_hw_ctx *(map_queue_fn)(struct request_queue *, const int);    /*映射的硬件队列*/
typedef enum blk_eh_timer_return (timeout_fn)(struct request *, bool);
typedef int (init_hctx_fn)(struct blk_mq_hw_ctx *, void *, unsigned int);       /*初始化硬件队列*/
typedef void (exit_hctx_fn)(struct blk_mq_hw_ctx *, unsigned int);
typedef int (init_request_fn)(void *, struct request *, unsigned int,unsigned int, unsigned int);
typedef void (exit_request_fn)(void *, struct request *, unsigned int,unsigned int);
```

queue_rq_fn()函数中第二个参数为 blk_mq_queue_data 结构体指针，结构体定义如下：

```
struct blk_mq_queue_data {      /*/include/linux/blk-mq.h*/
    struct request *rq;         /*入队的请求*/
    struct list_head *list;
    bool last;
};
```

10.5.2 创建 Multi queue 队列

块设备驱动程序中创建 Multi queue 请求队列的流程如下（请求队列赋予 gendisk 实例）：

（1）定义 blk_mq_ops 结构体实例，定义并设置 blk_mq_tag_set 结构体实例（关联 blk_mq_ops 实例），用于传递请求队列以及块设备的硬件配置信息。

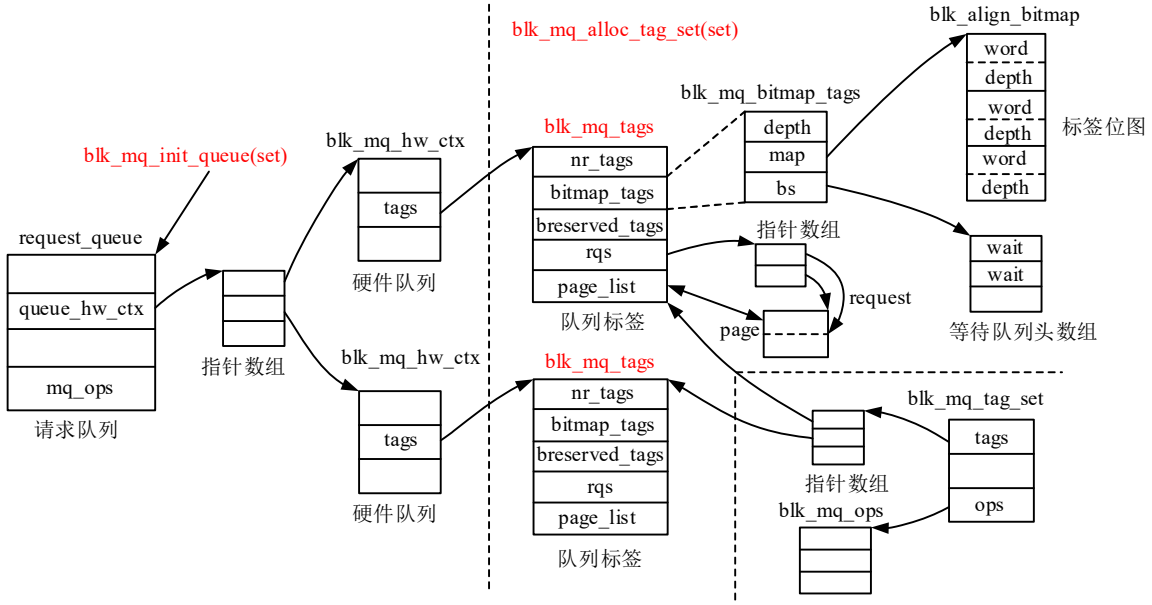
（2）调用 **blk_mq_alloc_tag_set**(struct blk_mq_tag_set *set)函数（参数是前面的 blk_mq_tag_set 实例），为硬件队列创建 blk_mq_tags 实例，用于管理请求。

（3）调用 **blk_mq_init_queue**(struct blk_mq_tag_set ***set**)函数（参数同（2）），创建 Multi queue 请求队列，函数返回 request_queue 实例指针。

下面主要介绍（2）（3）步骤中接口函数的实现。

1 硬件队列标签

在介绍以上接口函数的实现前，先来了解一下什么是队列标签。硬件队列中队列标签由 blk_mq_tags 结构体表示，如下图所示：



队列标签 blk_mq_tags 中 page_list 双链表管理的是一批内存块(按分配阶从伙伴系统分配的)首页 page 实例，在这些内存块中保存了请求实例（总数量为 nr_tags），rqs 指向的指针数组关联到这些请求实例。为 bio 实例分配请求实例时就从这些实例中分配。

队列标签中还包含一个由多个整数表示的位图，前面的每个请求对应一个比特位，置位表示请求已经被分配，清零表示请求未分配，比特位的位置值（整数），就是请求的标签值（tag）。标签值也是请求关联到 rqs 指向的指针数组项的索引值，简单地说 rqs[0]关联请求的标签值为 0，rqs[1]关联请求标签值为 1，依此类推，依标签值就可以查找到请求实例。标签数量 nr_tags 就是 rqs 指向的指针数组关联请求的总数量。

blk_mq_tags 结构体定义如下（/block/blk-mq-tag.h）：

```
struct blk_mq_tags {
    unsigned int nr_tags;            /*总的标签数量，含下面的保留标签数量*/
    unsigned int nr_reserved_tags;       /*保留标签数量*/
    atomic_t active_queues;

    struct blk_mq_bitmap_tags   bitmap_tags;      /*正常标签，/block/blk-mq-tag.h*/
    struct blk_mq_bitmap_tags   breserved_tags;      /*保留标签*/

    struct request **rqs;           /*指向指针数组，数组项数同 nr_tags 值*/
                                    /*指针数组项索引值就是关联请求的标签值*/
    struct list_head   page_list;    /*管理内存块首页 page 实例，内存块用于保存 request 实例*/
    int   alloc_policy;
    cpumask_var_t cpumask;      /*CPU 核位图*/
};
```

blk_mq_tags 中 blk_mq_bitmap_tags 结构体成员用于表示请求位图，定义如下（/block/blk-mq-tag.h）：

```
struct blk_mq_bitmap_tags {
    unsigned int depth;             /*深度，结构体管理标签（请求）数量*/
    unsigned int wake_cnt;
    unsigned int bits_per_word;      /*表示整型数位宽（位宽=2^bits_per_word)*/

    unsigned int map_nr;              /*map 成员指向数组项数*/
    struct blk_align_bitmap *map;       /*指向 blk_align_bitmap 结构体数组，
                                /*结构体中包括 word 和 depth 整数成员，表示位图，/block/blk-mq.h*/

    atomic_t wake_index;
    struct bt_wait_state *bs;         /*等待队列头，/block/blk-mq-tag.h*/
};
```

blk_mq_bitmap_tags 结构体中 map 指向 blk_align_bitmap 结构体数组，结构体定义如下：

```
struct blk_align_bitmap {           /*/block/blk-mq.h*/
    unsigned long   word;         /*整数，比特位用于表示位图*/
    unsigned long   depth;       /*word 成员中使用比特位的数量*/
} ____cacheline_aligned_in_smp;
```

word 无符号整数表示标签位图，每个比特位对应一个标签（请求），depth 表示 word 中使用的比特位的数量（从低到高，多余的位不用）。

blk_align_bitmap 结构体数组项中各 depth 值之和应当与 blk_mq_bitmap_tags 实例 depth 值相等，即等于管理标签总数。

blk_mq_tags 结构体中包含两个 blk_mq_bitmap_tags 结构体成员，分别是 bitmap_tags 和 breserved_tags，也就是说 blk_mq_tags 结构体中的 nr_tags 个标签被分成两部分，一部分是 bitmap_tags 中的正常标签，另一部分是 breserved_tags 中的保留标签，数量为 nr_reserved_tags。正常标签数为 nr_tags-nr_reserved_tags，也即以下等式成立：**nr_tags**=bitmap_tags.**depth**+breserved_tags.**depth**。

blk_mq_tags 结构体 bitmap_tags 和 breserved_tags 成员中的标签是统一编值的，保留标签值在前，正常标签值在后。例如：breserved_tags 成员中 bit0 标签值为 0，bit1 标签值为 1，假设保留标签数量为 8，则 bitmap_tags 成员中 bit0 标签值为 0+8，bit1 标签值为 1+8=9，依此类推。标签值 tag 对应 blk_mq_tags->rqs[**tag**]指向的请求。

2 创建硬件队列标签

在创建 Multi queue 队列前，需要为其下硬件队列创建队列标签，即 blk_mq_tags 实例，每个硬件队列对应一个实例。

在创建 Multi queue 队列时，内核将 blk_mq_tag_set 结构体作为参数，用来传递队列标签 blk_mq_tags 实例，以及块设备的硬件配置信息等，比如支持的硬件队列数 nr_hw_queues、队列深度 queue_depth 等。

块设备驱动程序需定义并设置 blk_mq_tag_set 实例，调用 blk_mq_alloc_tag_set(set)函数创建队列标签后（关联到 set 指向 blk_mq_tag_set 实例），才能以 blk_mq_tag_set 实例为参数创建 Multi queue 请求队列。

blk_mq_tag_set 结构体定义如下（/include/linux/blk-mq.h）：

```
struct blk_mq_tag_set {
    struct blk_mq_ops  *ops;        /*blk_mq_ops 实例指针*/
    unsigned int       nr_hw_queues;     /*硬件队列数量*/
    unsigned int       queue_depth;   /*深度，用于设置 blk_mq_tags.nr_tags 成员，即总标签数*/
    unsigned int       reserved_tags;    /*保留标签数*/
    unsigned int       cmd_size;        /*请求实例后附的额外数据大小*/
    int                numa_node;           /*内存节点*/
    unsigned int       timeout;
    unsigned int       flags;           /*标记，BLK_MQ_F_*，/include/linux/blk-mq.h*/
    void               *driver_data;       /*指向驱动数据结构*/

    struct blk_mq_tags **tags;     /*指向指针数组，数组项指向队列标签 blk_mq_tags 实例*/

    struct mutex       tag_list_lock;
    struct list_head   tag_list;          /*管理依本实例创建的 request_queue 实例*/
};
```

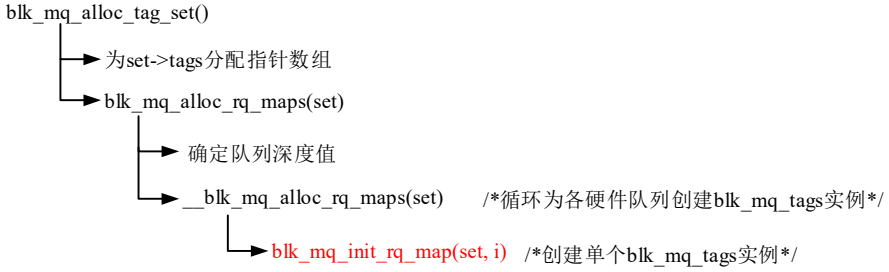
块设备驱动程序定义 blk_mq_tag_set 实例后，需要对其 ops、nr_hw_queues、queue_depth、reserved_tags 成员等进行设置，并调用 **blk_mq_alloc_tag_set()**函数创建队列标签 blk_mq_tags 实例（赋予 tags 成员指向的指针数组项）。

blk_mq_alloc_tag_set()函数定义如下（/block/blk-mq.c）：

```
int blk_mq_alloc_tag_set(struct blk_mq_tag_set *set)
{
    BUILD_BUG_ON(BLK_MQ_MAX_DEPTH > 1 << BLK_MQ_UNIQUE_TAG_BITS);

    if (!set->nr_hw_queues)        /*必须设置硬件队列数量*/
        return -EINVAL;
    if (!set->queue_depth)         /*必须设置队列深度（总标签数）*/
        return -EINVAL;
```

```
    if (set->queue_depth < set->reserved_tags + BLK_MQ_TAG_MIN)
        return -EINVAL;                /*BLK_MQ_TAG_MIN 为 1，/block/blk-mq-tag.h*/

    if (!set->ops->queue_rq || !set->ops->map_queue)       /*必须定义这两个函数*/
        return -EINVAL;

    if (set->queue_depth > BLK_MQ_MAX_DEPTH) {             /*深度不能超过最大值*/
        pr_info("blk-mq: reduced tag depth to %u\n",BLK_MQ_MAX_DEPTH);
        set->queue_depth = BLK_MQ_MAX_DEPTH;        /*10240，/include/linux/blk-mq.h*/
    }

    if (is_kdump_kernel()) {
        set->nr_hw_queues = 1;
        set->queue_depth = min(64U, set->queue_depth);
    }

    set->tags = kmalloc_node(set->nr_hw_queues *sizeof(struct blk_mq_tags *),
                                                    GFP_KERNEL, set->numa_node);
                        /*分配 blk_mq_tags 结构体指针数组，数组项数与硬件队列数量相同*/
    if (!set->tags)
        return -ENOMEM;

    if (blk_mq_alloc_rq_maps(set))       /*创建队列标签 blk_mq_tags 实例，/block/blk-mq.c*/
        goto enomem;

    mutex_init(&set->tag_list_lock);
    INIT_LIST_HEAD(&set->tag_list);

    return 0;
    ...
}
```

blk_mq_alloc_tag_set()函数为 set->tags 分配指针数组，调用 blk_mq_alloc_rq_maps()函数创建队列标签 blk_mq_tags 实例，函数调用关系如下图所示：



blk_mq_alloc_rq_maps()函数定义如下（/block/blk-mq.c）：

```
static int blk_mq_alloc_rq_maps(struct blk_mq_tag_set *set)
```

```
{
    unsigned int depth;
    int err;

    depth = set->queue_depth;       /*队列深度*/
    do {
        err = __blk_mq_alloc_rq_maps(set);   /*创建 blk_mq_tags 实例，成功返回 0，/block/blk-mq.c*/
        if (!err)
            break;          /*创建成功，跳出循环*/
        ...                 /*不成功则队列深度减半后再试*/
    } while (set->queue_depth);
    ...
    return 0;           /*成功返回 0*/
}
```

blk_mq_alloc_rq_maps()函数调用__blk_mq_alloc_rq_maps()函数为各个硬件队列创建 blk_mq_tags 实例，创建成功返回 0，如果创建不成功则将队列深度减半，再尝试，直至成功。

__blk_mq_alloc_rq_maps()函数定义如下（/block/blk-mq.c）：

```
static int __blk_mq_alloc_rq_maps(struct blk_mq_tag_set *set)
{
    int i;
    for (i = 0; i < set->nr_hw_queues; i++) {       /*为每个硬件队列创建标签*/
        set->tags[i] = blk_mq_init_rq_map(set, i);      /*创建单个 blk_mq_tags 实例，/block/blk-mq.c*/
        if (!set->tags[i])
            goto out_unwind;
    }
    return 0;
    ...
}
```

前面介绍了这么多，最终硬件队列对应的 blk_mq_tags 实例由 blk_mq_init_rq_map()函数创建，为每个硬件队列创建的 blk_mq_tags 实例都是一样的，参数都来自同一个 blk_mq_tag_set 实例，下面详细介绍此函数的实现。

■创建单个队列标签

blk_mq_init_rq_map()函数定义如下，用于创建单个 blk_mq_tags 实例（/block/blk-mq.c）：

```
static struct blk_mq_tags *blk_mq_init_rq_map(struct blk_mq_tag_set *set,unsigned int hctx_idx)
/*hctx_idx：硬件队列编号，即 set->tags 指针数组项索引值*/
{
    struct blk_mq_tags *tags;
    unsigned int i, j, entries_per_page, max_order = 4;   /*默认内存块分配阶，可能需要多个内存块*/
    size_t rq_size, left;
```

```
tags = blk_mq_init_tags(set->queue_depth, set->reserved_tags,
                        set->numa_node,BLK_MQ_FLAG_TO_ALLOC_POLICY(set->flags));
                          /*创建 blk_mq_tags 实例，包含其中位图等，/block/blk-mq-tag.c*/
...

INIT_LIST_HEAD(&tags->page_list);      /*初始化双链表成员*/

tags->rqs = kzalloc_node(set->queue_depth * sizeof(struct request *),
                    GFP_KERNEL | __GFP_NOWARN | __GFP_NORETRY,set->numa_node);
                              /*分配 request 指针数组，项数为 set->queue_depth*/
...

rq_size = round_up(sizeof(struct request) + set->cmd_size,cache_line_size());
                                                        /*请求大小，后附有额外数据*/
left = rq_size * set->queue_depth;        /*所有缓存请求的大小*/

for (i = 0; i < set->queue_depth; ) {
    int this_order = max_order;
    struct page *page;
    int to_do;
    void *p;

    while (left < order_to_size(this_order - 1) && this_order)
        this_order--;

    do {
        page = alloc_pages_node(set->numa_node,
            GFP_KERNEL | __GFP_NOWARN | __GFP_NORETRY | __GFP_ZERO,this_order);
                                                        /*分配内存页*/
        if (page)
            break;              /*分配成功跳出循环*/
        if (!this_order--)      /*分配不成功，阶数减 1，再分配*/
            break;
        if (order_to_size(this_order) < rq_size)
            break;
    } while (1);

    if (!page)
        goto fail;

    page->private = this_order;
    list_add_tail(&page->lru, &tags->page_list);
                                    /*首页 page 实例，添加到 tags->page_list 双链表*/
```

```
            p = page_address(page);            /*页物理地址*/
            entries_per_page = order_to_size(this_order) / rq_size;        /*每页请求数量*/
            to_do = min(entries_per_page, set->queue_depth - i);
            left -= to_do * rq_size;
            for (j = 0; j < to_do; j++) {
                tags->rqs[i] = p;          /*关联请求*/
                if (set->ops->init_request) {              /*初始化请求*/
                    if (set->ops->init_request(set->driver_data,tags->rqs[i], hctx_idx, i,set->numa_node)) {
                        ...
                    }
                }
                p += rq_size;
                i++;
            }
        }
        return tags;          /*返回 blk_mq_tags 实例指针*/
        ...
    }
```

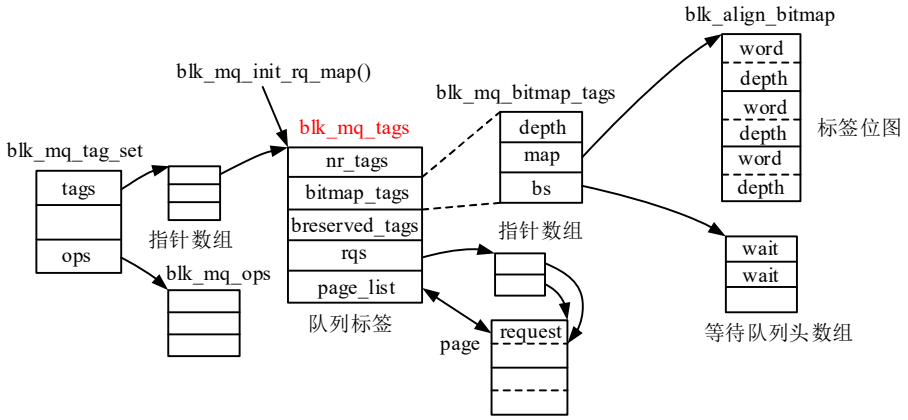
blk_mq_init_rq_map()函数完成的主要工作如下：

（1）调用 blk_mq_init_tags()函数创建 blk_mq_tags 实例，包含其中位图和等待队列头数组等，函数源代码请读者自行阅读。

（2）计算请求 request 实例大小 rq_size，请求后可能附有额外数据。

（3）分配物理内存块，用于保存 request 实例，并初始化 request 实例，关联到 rqs 指向的指针数组项。

blk_mq_init_rq_map()函数最终执行的结果如下图所示，此时请求、位图都已经分配好了：



blk_mq_tags 实例中 page_list 成员管理的是表示保存请求实例内存块首页的 page 实例，请求实例可能保存在一个内存块中（内存块默认分配阶为4，实际可能会更小），也可能需要多个内存块，也就是说 page_list 链表中可能有多个成员。

3 创建 Multi queue 队列函数

块设备驱动程序在调用 blk_mq_alloc_tag_set()函数为 blk_mq_tag_set 实例创建关联的 blk_mq_tags 实例后，就可以调用 **blk_mq_init_queue()**函数依 blk_mq_tag_set 实例创建 Multi queue 请求队列了。

blk_mq_init_queue()函数定义如下（/block/blk-mq.c）：

```
struct request_queue *blk_mq_init_queue(struct blk_mq_tag_set *set)
```

```
/*set：已经创建队列标签的 blk_mq_tag_set 实例指针*/
{
    struct request_queue *uninit_q, *q;

    uninit_q = blk_alloc_queue_node(GFP_KERNEL, set->numa_node);      /*分配请求队列，见上文*/
    if (!uninit_q)
        return ERR_PTR(-ENOMEM);

    q = blk_mq_init_allocated_queue(set, uninit_q);      /*初始化 Multi queue 队列，见下文*/
    if (IS_ERR(q))
        blk_cleanup_queue(uninit_q);

    return q;
}
```

blk_mq_init_queue()函数中调用 blk_mq_init_queue()函数分配 request_queue 实例并做简单的初始化（同标准请求队列），然后调用 blk_mq_init_allocated_queue()函数设置 Multi queue 请求队列。

blk_mq_init_queue()函数前面介绍过了，下面主要介绍 blk_mq_init_allocated_queue()函数的实现。

■初始化 Multi queue 队列

blk_mq_init_allocated_queue()函数用于初始化 Multi queue 请求队列，定义如下（/block/blk-mq.c）：

```
struct request_queue *blk_mq_init_allocated_queue(struct blk_mq_tag_set *set,struct request_queue *q)
{
    struct blk_mq_hw_ctx **hctxs;
    struct blk_mq_ctx __percpu *ctx;
    unsigned int *map;
    int i;

    ctx = alloc_percpu(struct blk_mq_ctx);          /*分配软件队列 blk_mq_ctx 实例*/
    ...
    hctxs = kmalloc_node(set->nr_hw_queues * sizeof(*hctxs), GFP_KERNEL,set->numa_node);
                                                    /*分配硬件队列指针数组*/
    ...

    map = blk_mq_make_queue_map(set);        /*创建软件队列到硬件队列的映射数组*/
    ...                                      /*/block/blk-mq-cpumap.c*/

    for (i = 0; i < set->nr_hw_queues; i++) {        /*创建硬件队列，关联到 hctxs 指向的指针数组*/
        int node = blk_mq_hw_queue_to_node(map, i);

        hctxs[i] = kzalloc_node(sizeof(struct blk_mq_hw_ctx),GFP_KERNEL, node);
        ...                                          /*分配硬件队列*/
```

```
        if (!zalloc_cpumask_var_node(&hctxs[i]->cpumask, GFP_KERNEL,node))
            goto err_hctxs;

        atomic_set(&hctxs[i]->nr_active, 0);
        hctxs[i]->numa_node = node;
        hctxs[i]->queue_num = i;                /*硬件队列编号，从 0 开始*/
}

if (percpu_ref_init(&q->mq_usage_counter, blk_mq_usage_counter_release,
                    PERCPU_REF_INIT_ATOMIC, GFP_KERNEL))
        goto err_hctxs;

setup_timer(&q->timeout, blk_mq_rq_timer, (unsigned long) q);
blk_queue_rq_timeout(q, set->timeout ? set->timeout : 30 * HZ);
/*设置请求队列成员*/
q->nr_queues = nr_cpu_ids;          /*软件队列（CPU 核）数量*/
q->nr_hw_queues = set->nr_hw_queues;        /*硬件队列数量*/
q->mq_map = map;                /*指向软件队列到硬件队列的映射数组*/

q->queue_ctx = ctx;                 /*指向软件队列*/
q->queue_hw_ctx = hctxs;            /*指向硬件队列指针数组*/

q->mq_ops = set->ops;               /*指向 blk_mq_ops 实例*/
q->queue_flags |= QUEUE_FLAG_MQ_DEFAULT;            /*Multi queue 请求队列默认标记*/

if (!(set->flags & BLK_MQ_F_SG_MERGE))
        q->queue_flags |= 1 << QUEUE_FLAG_NO_SG_MERGE;

q->sg_reserved_size = INT_MAX;

INIT_WORK(&q->requeue_work, blk_mq_requeue_work);       /*/block/blk-mq.c*/
                                /*工作执行函数将 q->requeue_list 中请求取出，再插入（硬件）队列*/
INIT_LIST_HEAD(&q->requeue_list);       /*初始化请求双链表头*/
spin_lock_init(&q->requeue_lock);

if (q->nr_hw_queues > 1)                /*设置请求队列*/
        blk_queue_make_request(q, blk_mq_make_request);         /*多硬件队列情形*/
else
        blk_queue_make_request(q, blk_sq_make_request);         /*单硬件队列情形*/

q->nr_requests = set->queue_depth;          /*深度，标签（请求）总数量*/

if (set->ops->complete)         /*set->ops->complete()赋予请求队列 q.softirq_done_fn()成员*/
```

```
        blk_queue_softirq_done(q, set->ops->complete);        /*/block/blk-settings.c*/

    blk_mq_init_cpu_queues(q, set->nr_hw_queues);        /*初始化软件队列，/block/blk-mq.c*/

    if (blk_mq_init_hw_queues(q, set))          /*初始化硬件队列，/block/blk-mq.c*/
        goto err_hctxs;

    mutex_lock(&all_q_mutex);
    list_add_tail(&q->all_q_node, &all_q_list);      /*request_queue 添加到 all_q_list 双链表末尾*/
    mutex_unlock(&all_q_mutex);

    blk_mq_add_queue_tag_set(set, q);        /*建立 blk_mq_tag_set 实例与 request_queue 实例关联*/
                                             /*/block/blk-mq.c*/
    blk_mq_map_swqueue(q);        /*设置软硬件队列，/block/blk-mq.c*/

    return q;          /*返回队列指针*/
    ...
}
```

blk_mq_init_allocated_queue()函数将分配好的 request_queue 实例设置成 Multi queue 队列，函数完成的主要工作简列如下：

（1）分配软件队列 blk_mq_ctx 实例，percpu 变量，每个 CPU 核对应一个实例。

（2）分配指向硬件队列的指针数组（hctxs 指向此数组）。

（3）创建软件队列到硬件队列的映射数组（map 指向此数组）。

（4）分配硬件队列 blk_mq_hw_ctx 实例，并关联到 hctxs 指向的指针数组项。

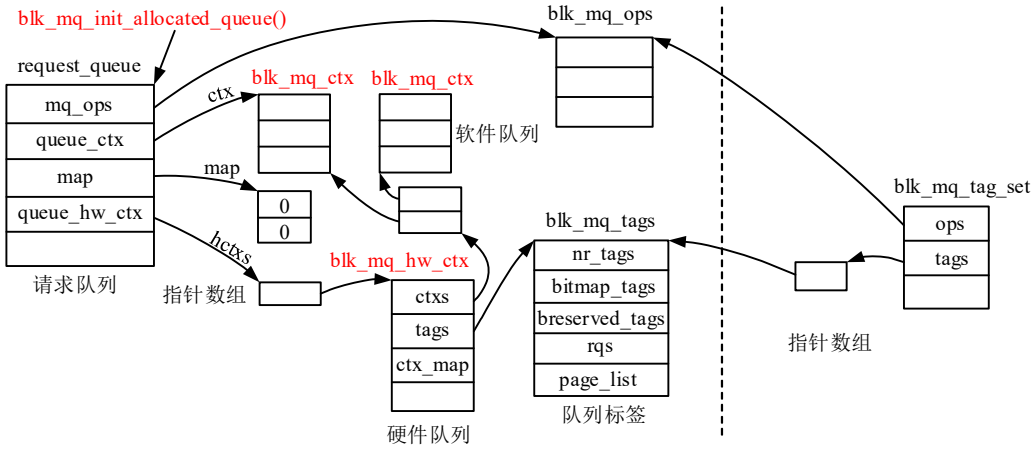
（5）设置 request_queue 实例，如设置 bio 处理函数等等。

（6）初始化硬件队列，关联队列标签 blk_mq_tags 实例等。

（7）建立 blk_mq_tag_set 实例与 request_queue 实例之间关联。

（8）设置软硬件队列。

blk_mq_init_allocated_queue()函数设置的 Multi queue 队列简列如下图所示（假设有 2 个软件队列，1 个硬件队列）：



队列标签 blk_mq_tags 实例在前面介绍过了，这里就不再重复了。下面简单介绍一下以上代码中调用

的 blk_mq_init_hw_queues(q, set)和 blk_mq_map_swqueue(q)函数，它们用来初始化/设置软硬件队列。

●初始化硬件队列

blk_mq_init_hw_queues()函数用于初始化硬件队列，函数定义如下（/block/blk-mq.c）：

```
static int blk_mq_init_hw_queues(struct request_queue *q,struct blk_mq_tag_set *set)
{
    struct blk_mq_hw_ctx *hctx;
    unsigned int i;

    queue_for_each_hw_ctx(q, hctx, i) {          /*遍历硬件队列，/include/linux/blk-mq.h*/
        if (blk_mq_init_hctx(q, set, hctx, i))          /*初始化硬件队列，/block/blk-mq.c*/
            break;
    }

    if (i == q->nr_hw_queues)        /*所有硬件队列都已成功初始化，返回 0*/
        return 0;
    ...
}
```

blk_mq_init_hctx()函数用于初始化单个硬件队列，定义如下（/block/blk-mq.c）：

```
static int blk_mq_init_hctx(struct request_queue *q,struct blk_mq_tag_set *set,
                                        struct blk_mq_hw_ctx *hctx, unsigned hctx_idx)
/*hctx_idx：硬件队列编号*/
{
    int node;
    unsigned flush_start_tag = set->queue_depth;

    node = hctx->numa_node;
    if (node == NUMA_NO_NODE)
        node = hctx->numa_node = set->numa_node;

    INIT_DELAYED_WORK(&hctx->run_work, blk_mq_run_work_fn);     /*/block/blk-mq.c*/
                                        /*工作将软件队列中请求移入硬件队列*/
    INIT_DELAYED_WORK(&hctx->delay_work, blk_mq_delay_work_fn);   /*/block/blk-mq.c*/
                                    /*如果硬件队列停止了，则激活它，迁入软件队列中请求*/
    spin_lock_init(&hctx->lock);
    INIT_LIST_HEAD(&hctx->dispatch);
    hctx->queue = q;                        /*指向请求队列*/
    hctx->queue_num = hctx_idx;             /*硬件队列编号*/
    hctx->flags = set->flags;               /*标记*/
    blk_mq_init_cpu_notifier(&hctx->cpu_notifier,blk_mq_hctx_notify, hctx);
                            /*blk_mq_hctx_notify()函数用于迁移请求，/block/blk-mq.c*/
    blk_mq_register_cpu_notifier(&hctx->cpu_notifier);      /*注册 blk_mq_cpu_notifier 实例*/
```

```
    hctx->tags = set->tags[hctx_idx];          /*硬件队列关联 blk_mq_tags 实例*/
    hctx->ctxs = kmalloc_node(nr_cpu_ids * sizeof(void *),GFP_KERNEL, node);
                                               /*分配指针数组，指向软件队列*/
    ...          /*分配失败的处理*/

    if (blk_mq_alloc_bitmap(&hctx->ctx_map, node))        /*为 hctx->ctx_map 成员分配位图*/
        goto free_ctxs;

    hctx->nr_ctx = 0;

    if (set->ops->init_hctx &&set->ops->init_hctx(hctx, set->driver_data, hctx_idx))       /*初始化函数*/
        goto free_bitmap;

    hctx->fq = blk_alloc_flush_queue(q, hctx->numa_node, set->cmd_size);
                                                  /*分配 FLUSH 队列，见创建标准请求队列*/
    ...

    /*初始化 FLUSH 队列 fluh_rq 请求*/
    if (set->ops->init_request &&set->ops->init_request(set->driver_data,
                    hctx->fq->flush_rq, hctx_idx, flush_start_tag + hctx_idx, node))
        goto free_fq;

    return 0;
    ...
}
```

blk_mq_init_hctx()函数用于设置硬件队列成员，设置延时工作成员，关联 blk_mq_tags 实例，分配指针数组用于关联映射到本硬件队列的软件队列，为 hctx->ctx_map 成员分配位图，调用 blk_mq_ops 实例中定义的硬件队列初始化函数，分配 FLUSH 队列（同标准请求队列）等。

这里需要说明一下的是硬件队列 hctx->ctx_map 成员，它是 blk_mq_ctxmap 结构体实例，定义如下：

```
struct blk_mq_ctxmap {              /*/include/linux/blk-mq.h*/
    unsigned int size;
    unsigned int bits_per_word;
    struct blk_align_bitmap *map;       /*关联软件队列（CPU 核）位图*/
};
```

blk_mq_ctxmap 与队列标签中的 blk_mq_bitmap_tags 结构体有几分相似，它也包含一个由 map 成员指向的位图，不过这个位图不是标记请求，而是标记软件队列（或者说是 CPU 核）。每个软件队列对应位图中一个比特位，置位表示软件队列有请求需要移入本硬件队列。在运行硬件队列将软件队列中请求移入硬件队列后清零相应标记位。

blk_mq_alloc_bitmap()函数用于设置 ctx_map 成员，并为其分配位图，即结构体 blk_align_bitmap 数组，位图中标签数量通常为 CPU 核数量，源代码请读者自行阅读。

●设置软硬件队列

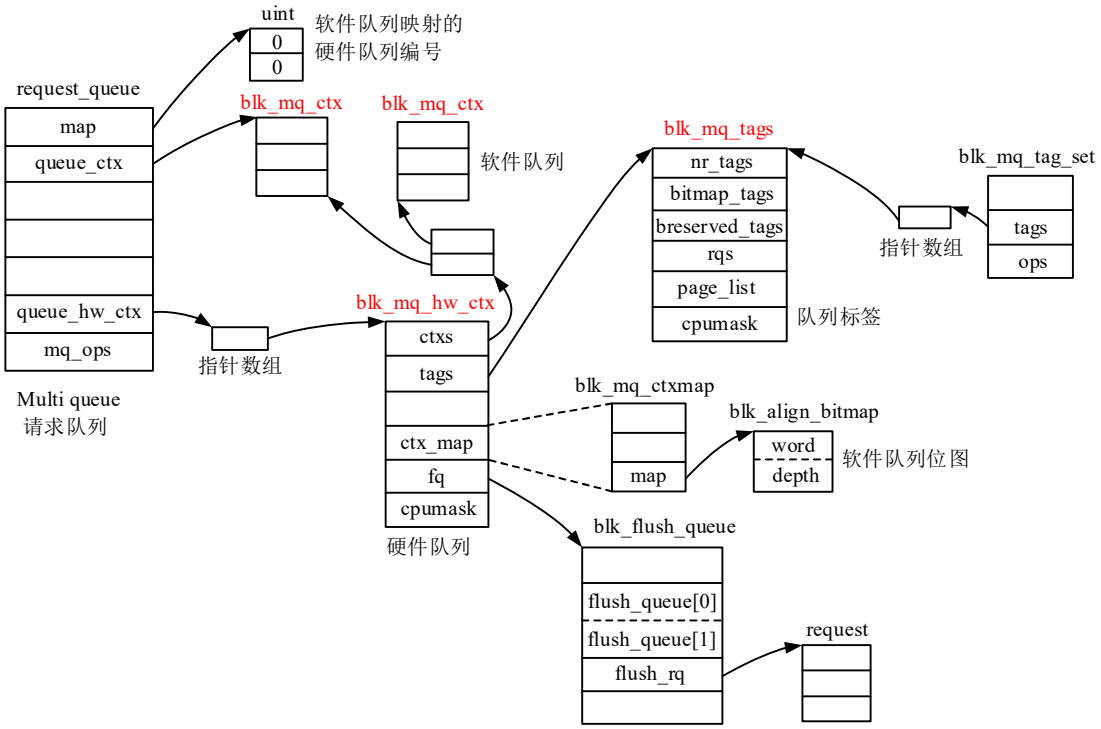
blk_mq_map_swqueue()函数用于设置 Multi queue 队列中的软件和硬件队列（/block/blk-mq.c）：

```
static void blk_mq_map_swqueue(struct request_queue *q)
{
    unsigned int i;
    struct blk_mq_hw_ctx *hctx;
    struct blk_mq_ctx *ctx;
    struct blk_mq_tag_set *set = q->tag_set;

    queue_for_each_hw_ctx(q, hctx, i) {        /*清零硬件队列 CPU 位图*/
        cpumask_clear(hctx->cpumask);
        hctx->nr_ctx = 0;
    }

    queue_for_each_ctx(q, ctx, i) {        /*遍历软件队列*/
        if (!cpu_online(i))
            continue;

        hctx = q->mq_ops->map_queue(q, i);    /*软件队列（CPU 核）映射的硬件队列*/
        cpumask_set_cpu(i, hctx->cpumask);      /*设置硬件队列中 CPU 位图*/
        ctx->index_hw = hctx->nr_ctx;        /*软件队列关联到 hctx->ctxs[]数组项数*/
        hctx->ctxs[hctx->nr_ctx++] = ctx;          /*关联到软件队列*/
    }

    queue_for_each_hw_ctx(q, hctx, i) {        /*遍历硬件队列*/
        struct blk_mq_ctxmap *map = &hctx->ctx_map;        /*硬件队列内嵌的 blk_mq_ctxmap 实例*/

        if (!hctx->nr_ctx) {    /*如果硬件队列不关联到软件队列，则释放关联的 blk_mq_tags 实例*/
            if (set->tags[i]) {
                blk_mq_free_rq_map(set, set->tags[i], i);
                set->tags[i] = NULL;
            }
            hctx->tags = NULL;
            continue;
        }

        if (!set->tags[i])      /*如果为 NULL，则需要创建 blk_mq_tags 实例*/
            set->tags[i] = blk_mq_init_rq_map(set, i);
        hctx->tags = set->tags[i];          /*关联到 blk_mq_tags 实例*/
        WARN_ON(!hctx->tags);

        map->size = DIV_ROUND_UP(hctx->nr_ctx, map->bits_per_word);
        hctx->next_cpu = cpumask_first(hctx->cpumask);          /*位图中下一个 CPU 编号*/
```

```
            hctx->next_cpu_batch = BLK_MQ_CPU_WORK_BATCH;
        }       /*遍历硬件队列结束*/

        queue_for_each_ctx(q, ctx, i) {         /*遍历软件队列*/
            if (!cpu_online(i))
                continue;

            hctx = q->mq_ops->map_queue(q, i);          /*软件队列映射的硬件队列*/
            cpumask_set_cpu(i, hctx->tags->cpumask);    /*设置 blk_mq_tags 实例中 CPU 位图*/
        }
    }
```

blk_mq_map_swqueue()函数将建立硬件队列与关联软件队列之间的联系，建立硬件队列与 blk_mq_tags 实例之间的关联，设置硬件队列和队列标签中的 CPU 核位图等。

下图示意了 blk_mq_init_hw_queues()和 blk_mq_map_swqueue()函数初始化后的软硬件队列，假设有 2 个软件队列，只有一个硬件队列，2 个软件队列映射到同一个硬件队列（编号为 0）。



10.5.3 处理 bio

VFS 层调用 submit_bio()函数向请求队列提交 bio 实例时，将由请求队列的 make_request_fn()函数处理。

在前面介绍的初始化 Multi queue 队列的 blk_mq_init_allocated_queue()函数中，将设置队列处理 bio 的 make_request_fn()函数为 **blk_mq_make_request**()或 blk_sq_make_request()，前者用于多个硬件队列的情形，后者用于单个硬件队列的情形。

下面以 blk_mq_make_request()函数为例，介绍其实现，函数定义如下（/block/blk-mq.c）：

```
static void blk_mq_make_request(struct request_queue *q, struct bio *bio)
{
    const int is_sync = rw_is_sync(bio->bi_rw);         /*是不是同步读写*/
```

```
const int is_flush_fua = bio->bi_rw & (REQ_FLUSH | REQ_FUA);      /*是否是 FLUSH/FUA 请求*/
struct blk_map_ctx data;          /*软硬件队列集合，/block/blk-mq.c*/
struct request *rq;
unsigned int request_count = 0;
struct blk_plug *plug;
struct request *same_queue_rq = NULL;        /*请求指针*/

blk_queue_bounce(q, &bio);

if (bio_integrity_enabled(bio) && bio_integrity_prep(bio)) {        /*数据完整性检查*/
      bio_endio(bio, -EIO);
      return;
}

if (!is_flush_fua && !blk_queue_nomerges(q) &&            /*非 FLUS/FUA 请求*/
            blk_attempt_plug_merge(q, bio, &request_count, &same_queue_rq))
                                    /*能否与进程 plug 链表中请求合并，/block/blk-core.c*/
                                    /*same_queue_rq 指向可合并本 bio 的请求，但可能合并不成功*/
      return;

rq = blk_mq_map_request(q, bio, &data);        /*分配请求，/block/blk-mq.c*/
...

if (unlikely(is_flush_fua)) {          /*处理 FLUSH/FUA 请求，下一小节将专门介绍*/
      blk_mq_bio_to_request(rq, bio);        /*设置请求，关联 bio 等，/block/blk-mq.c*/
      blk_insert_flush(rq);      /*请求插入硬件队列中的 FLUSH 队列，/block/blk-flush.c*/
      goto run_queue;            /*跳转至运行硬件队列*/
}

plug = current->plug;
if (((plug && !blk_queue_nomerges(q)) || is_sync) &&                    /*处理同步操作*/
      !(data.hctx->flags & BLK_MQ_F_DEFER_ISSUE)) {
      struct request *old_rq = NULL;

      blk_mq_bio_to_request(rq, bio);          /*设置请求，关联 bio 等*/
      if (plug) {       /*plug!=NULL*/
            if (same_queue_rq && !list_empty(&plug->mq_list)) { /*有能合并的请求，但合并不成功*/
                  old_rq = same_queue_rq;
                  list_del_init(&old_rq->queuelist);
            }
            list_add_tail(&rq->queuelist, &plug->mq_list);        /*添加到 plug->mq_list 链表末尾*/
      } else            /*plug==NULL，同步操作（尝试插入到硬件队列）*/
            old_rq = rq;
```

```
        blk_mq_put_ctx(data.ctx);
        if (!old_rq)        /*old_rq 为 NULL（请求已添加到 plug->mq_list 链表），函数返回*/
            return;

        if (!blk_mq_direct_issue_request(old_rq))
                    /*尝试将请求插入硬件队列，成功返回 0，不成功返回-1，/block/blk-mq.c*/
            return;
        blk_mq_insert_request(old_rq, false, true, true);    /*插入硬件队列失败，调用此函数*/
                                /*请求插入软件队列，并运行硬件队列，/block/blk-mq.c*/
        return;
    }
    /*current->plug 为 NULL，且是异步操作，执行以下代码*/
    if (!blk_mq_merge_queue_io(data.hctx, data.ctx, rq, bio)) {   /*请求插入软件队列（允许合并）*/
run_queue:
        blk_mq_run_hw_queue(data.hctx, !is_sync || is_flush_fua); /*运行硬件队列，/block/blk-mq.c*/
    }
    blk_mq_put_ctx(data.ctx);
}
```

Multi queue 队列的 blk_mq_make_request()函数与标准请求队列中的函数有些类似，但又有不同，其主要完成的工作简述如下：

（1）判断 bio 能否与进程 plug 链表（plug->mq_list）中请求合并，能则合并，函数返回，不能合并继续往下执行。Multi queue 队列没有使用 IO 调度器，因此不需要再调用 IO 调度器中的合并请求函数。

（2）调用 blk_mq_map_request()函数为 bio 分配请求。

（3）如果是 FLUSH/FUA 请求，则插入到硬件队列中的 FLUSH 队列，并运行硬件队列，函数返回。若不是 FLUSH/FUA 请求则继续往下执行。

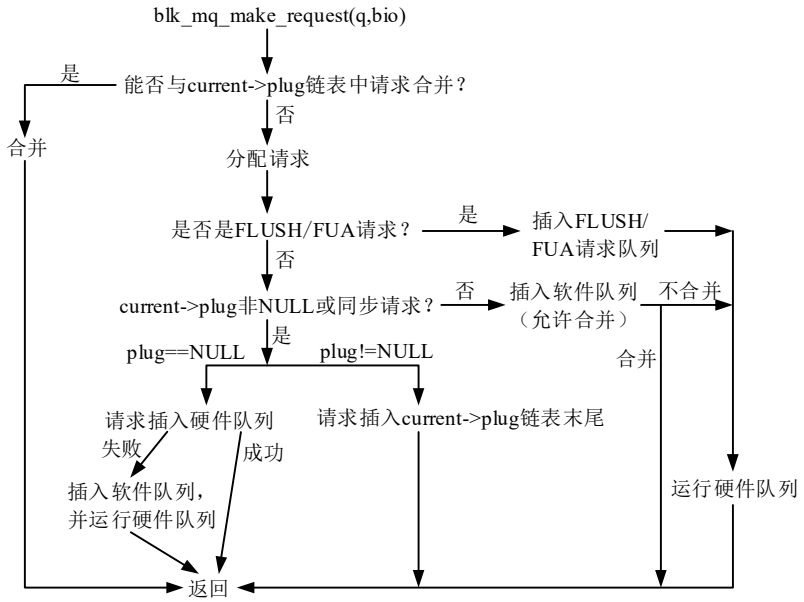
（4）如果是同步请求或 current->plug 非 NULL，则执行下列操作，否则跳至步骤（5）：

①current->plug 非 NULL，将请求插入 current->plug 链表末尾，函数返回，否则执行②。

②current->plug 为 NULL，调用 blk_mq_direct_issue_request()函数尝试将请求插入硬件队列，若成功函数返回；若不成功再调用 blk_mq_insert_request()函数将请求插入软件队列，并运行硬件队列，函数返回。

（5）如果是异步请求且 current->plug 为 NULL，则调用 blk_mq_merge_queue_io()函数尝试将 bio 与软件队列中请求合并，能合并则合并，不能合并则将请求插入软件队列。若 bio 不能与软件队列中请求合并，则继续调用 blk_mq_run_hw_queue()函数运行硬件队列，函数返回。若 bio 能与软件队列中请求合并，合并后即返回，不运行硬件队列。

下图示意了 blk_mq_make_request()函数的执行流程：



下面介绍以上流程中几个比较重要的函数。

1 分配请求

blk_mq_map_request()函数用于为 bio 分配请求 request 实例，在介绍函数实现前，先介绍 blk_map_ctx 结构体定义（/block/blk-mq.c）：

```
struct blk_map_ctx {
    struct blk_mq_hw_ctx *hctx;          /*硬件队列指针*/
    struct blk_mq_ctx *ctx;              /*软件队列指针*/
};
```

blk_map_ctx 结构体是硬件队列和软件队列指针的集合。

blk_mq_map_request()函数定义如下（/block/blk-mq.c）：

```
static struct request *blk_mq_map_request(struct request_queue *q,struct bio *bio,struct blk_map_ctx *data)
/*q：请求队列，bio：bio 指针，data：指向 blk_map_ctx 结构体实例*/
{
    struct blk_mq_hw_ctx *hctx;
    struct blk_mq_ctx *ctx;
    struct request *rq;
    int rw = bio_data_dir(bio);
    struct blk_mq_alloc_data alloc_data;          /*请求队列、硬软件队列集合，/block/blk-mq.h*/

    if (unlikely(blk_mq_queue_enter(q, GFP_KERNEL))) {
        bio_endio(bio, -EIO);
        return NULL;
    }

    ctx = blk_mq_get_ctx(q);              /*当前 CPU 核对应的软件队列，/block/blk-mq.h*/
    hctx = q->mq_ops->map_queue(q, ctx->cpu);        /*软件队列对应的硬件队列*/
```

```
    if (rw_is_sync(bio->bi_rw))          /*是否是同步读写*/
        rw |= REQ_SYNC;

    trace_block_getrq(q, bio, rw);
    blk_mq_set_alloc_data(&alloc_data, q, GFP_ATOMIC, false, ctx,hctx);     /*不使用保留标签*/
                                                                    /*设置 blk_mq_alloc_data 结构体实例*/
    rq = __blk_mq_alloc_request(&alloc_data, rw);          /*分配请求，/block/blk-mq.c*/
    ...      /*分配不成功，则运行硬件队列后再分配*/

    hctx->queued++;          /*分配请求数加 1*/
    data->hctx = hctx;       /*data 参数返回对应的软硬件队列*/
    data->ctx = ctx;
    return rq;               /*返回请求实例指针*/
}
```

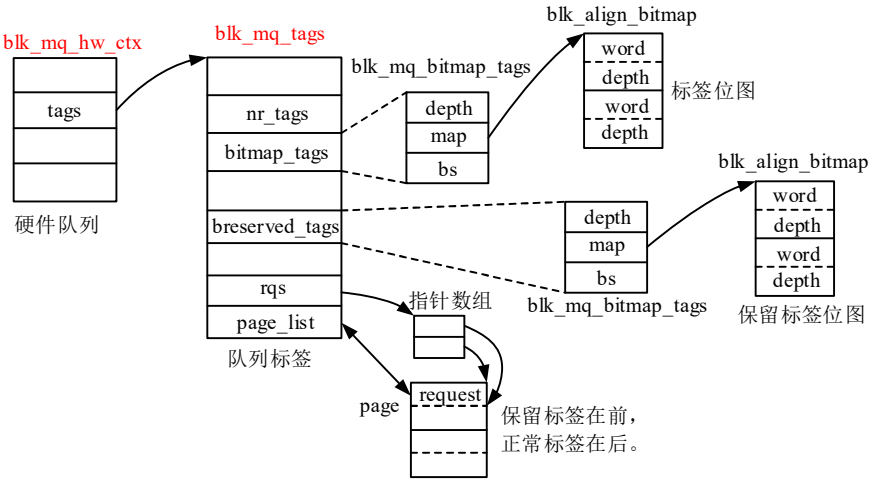
blk_mq_map_request()函数确定软、硬件队列后，调用**__blk_mq_alloc_request**(&alloc_data, rw)函数分配请求实例。这里的分配请求是在硬件队列关联队列标签 blk_mq_tags 实例的 rqs 指针数组中选择一个现成的空闲请求，请求在创建标签时就已经创建好了。

__blk_mq_alloc_request()函数定义如下（/block/blk-mq.c）：

```
static struct request *__blk_mq_alloc_request(struct blk_mq_alloc_data *data, int rw)
{
    struct request *rq;
    unsigned int tag;

    tag = blk_mq_get_tag(data);      /*获取空闲请求的标签值，/block/blk-mq-tag.c*/

    if (tag != BLK_MQ_TAG_FAIL) {
        rq = data->hctx->tags->rqs[tag];        /*请求实例指针*/

        if (blk_mq_tag_busy(data->hctx)) {
            rq->cmd_flags = REQ_MQ_INFLIGHT;
            atomic_inc(&data->hctx->nr_active);
        }

        rq->tag = tag;          /*标签值赋予请求*/
        blk_mq_rq_ctx_init(data->q, data->ctx, rq, rw);          /*初始化请求，/block/blk-mq.c*/
        return rq;
    }

    return NULL;
}
```

__blk_mq_alloc_request()函数调用 blk_mq_get_tag(data)函数分配请求标签值，假设为 tag，则分配的请求为 **hctx->tags->rqs[tag]**，然后调用 blk_mq_rq_ctx_init()函数初始化请求，具体函数源代码请读者自行阅读。

下面用图示的方式简单说明分配标签值的流程，如下图所示。



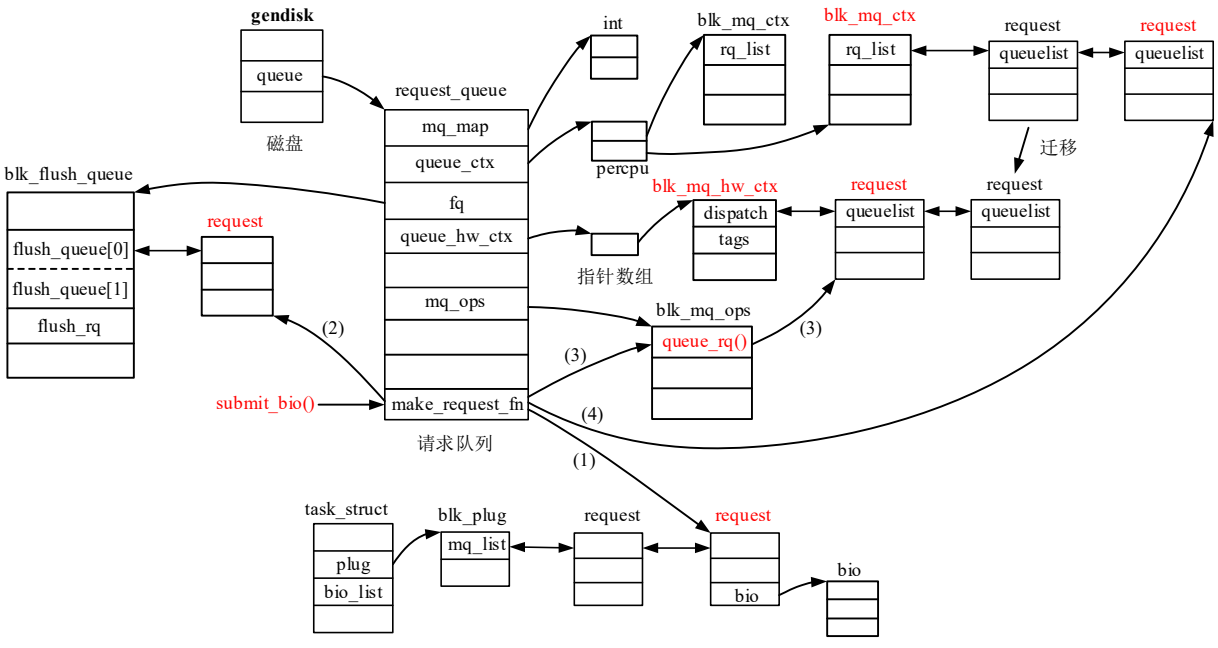
由前面介绍的硬件队列标签可知，标签中包含正常标签和保留标签，两种标签都是由位图来表示，位图中的比特位位置用于计算标签值，分配出去的标签值置位对应比特位，空闲标签值对应比特位清零。正常标签值和保留标签值是统一编值的，保留标签值在前，正常标签值在后。例如，保留标签位图 bit0 对应的标签值为 0，bit1 标签值为 1，若保留标签值数量为 8，则正常标签位图 bit0 对应的标签值为 0+8，bit1 对应的标签值为 9，依此类推。

blk_mq_get_tag(data)函数中 data->reserved 参数值表示是否是分配保留标签值，不是的话就分配正常标签值，分配标签值就是查找位图中第一个为 0 的比特位，置位比特位，再计算标签值。假设计算的标签值为 tag，则硬件队列 **hctx->tags->rqs[tag]**指向的请求就是本次分配的请求，也就是说 rqs[]数组项索引值就是标签值。分配标签值时可能进入睡眠等待。

blk_mq_map_request()函数分配请求时，并没有关联 bio 实例，随后需要调用 blk_mq_bio_to_request(rq, bio)函数将 bio 实例关联到请求后，才能将请求插入软/硬件队列。

2 插入请求

由前面 Multi queue 队列 make_request_fn()函数 **blk_mq_make_request()**的分析可知，关联 bio 实例的请求可能插入以下队列（链表）：（1）current->plug、（2）FLUSH 队列、（3）硬件队列、（4）软件队列，如下图所示。



下面将介绍将请求插入以上队列函数的实现，插入 current->plug 链表只是简单地将请求添加到双链表末尾，就不需要过多解释了，插入 FLUSH 队列的情形下一节再专门介绍。

■插入硬件队列

blk_mq_direct_issue_request()函数用于尝试将请求插入硬件队列，函数定义如下（/block/blk-mq.c）：

```
static int blk_mq_direct_issue_request(struct request *rq)
{
    int ret;
    struct request_queue *q = rq->q;
    struct blk_mq_hw_ctx *hctx = q->mq_ops->map_queue(q,rq->mq_ctx->cpu);
                                                        /*CPU 核映射的硬件队列*/
    struct blk_mq_queue_data bd = {
        .rq = rq,
        .list = NULL,
        .last = 1
    };

    ret = q->mq_ops->queue_rq(hctx, &bd);          /*调用 blk_mq_ops 实例中函数*/
    if (ret == BLK_MQ_RQ_QUEUE_OK)
        return 0;
    else {              /*插入硬件队列失败*/
        __blk_mq_requeue_request(rq);        /*只是修改请求成员值，没有插入队列*/
        if (ret == BLK_MQ_RQ_QUEUE_ERROR) {
            rq->errors = -EIO;
            blk_mq_end_request(rq, rq->errors);        /*出错，结束请求*/
            return 0;
        }
        return -1;
    }
}
```

blk_mq_direct_issue_request()函数比较简单就是调用 blk_mq_ops 实例中 **queue_rq()**函数将请求插入硬件队列，并激活队列请求的处理，成功返回 0，失败返回-1。

■插入软件队列

blk_mq_insert_request()函数用于将请求插入软件队列，函数定义如下（/block/blk-mq.c）：

```
void blk_mq_insert_request(struct request *rq, bool at_head, bool run_queue,bool async)
/*rq：请求，at_head：是否插入队列头部，run_queue：是否运行硬件队列（true），async：异步?*/
{
    struct request_queue *q = rq->q;
    struct blk_mq_hw_ctx *hctx;
    struct blk_mq_ctx *ctx = rq->mq_ctx, *current_ctx;
```

```
    current_ctx = blk_mq_get_ctx(q);            /*当前 CPU 核对应的软件队列*/
    if (!cpu_online(ctx->cpu))          /*若原关联软件队列对应 CPU 核不在线*/
        rq->mq_ctx = ctx = current_ctx;         /*关联当前 CPU 核软件队列*/

    hctx = q->mq_ops->map_queue(q, ctx->cpu);               /*映射的硬件队列*/

    spin_lock(&ctx->lock);          /*持有软件队列的自旋锁*/
    __blk_mq_insert_request(hctx, rq, at_head);         /*将请求插入软件队列，/block/blk-mq.c*/
    spin_unlock(&ctx->lock);

    if (run_queue)
        blk_mq_run_hw_queue(hctx, async);       /*运行硬件队列，见下文*/

    blk_mq_put_ctx(current_ctx);
}
```

blk_mq_insert_request()函数需要确定请求添加到哪个软件队列，以及软件队列映射的硬件队列，然后调用__blk_mq_insert_request()函数将请求添加到软件队列 **ctx->rq_list** 双链表头部或末尾，并设置软件队列在硬件队列 hctx->ctx_map 位图中的标记位。注意这里没有考虑与软件队列中现有请求的合并。

如果需要运行硬件队列则调用 blk_mq_run_hw_queue()函数运行硬件队列，即将软件队列中请求移入硬件队列。

在处理 bio 实例的 blk_mq_make_request()函数中对于 current->plug 为 NULL 的异步请求，将调用函数 blk_mq_merge_queue_io()将请求插入软件队列，此函数将判断 bio 能否与 ctx->rq_list 双链表中请求合并，如果可以则合并，函数返回；如果不能合并，再调用__blk_mq_insert_request()函数将请求插入 ctx->rq_list 队列，函数源代码请读者自行阅读。

3 运行硬件队列

在分析 blk_mq_make_request()函数的执行流程中，有多处涉及到运行硬件队列，其含义就是将软件队列中请求移入硬件队列。

运行硬件队列的接口函数为 blk_mq_run_hw_queue()，定义如下（/block/blk-mq.c）：

```
void blk_mq_run_hw_queue(struct blk_mq_hw_ctx *hctx, bool async)
{
    if (unlikely(test_bit(BLK_MQ_S_STOPPED, &hctx->state) ||!blk_mq_hw_queue_mapped(hctx)))
        return;

    if (!async) {           /*async 为 0 表示是同步请求*/
        int cpu = get_cpu();            /*当前 CPU 核*/
        if (cpumask_test_cpu(cpu, hctx->cpumask)) {
            __blk_mq_run_hw_queue(hctx);        /*移动软件队列请求至硬件队列，/block/blk-mq.c*/
            put_cpu();
            return;
        }
```

```
        put_cpu();
    }
    /*async 为 1 或当前 CPU 核不在硬件队列的位图中*/
    kblockd_schedule_delayed_work_on(blk_mq_hctx_next_cpu(hctx),&hctx->run_work, 0);
        /*初始化硬件队列时，run_work 执行函数为 blk_mq_run_work_fn()，其工作也是迁移请求*/
}
```

__blk_mq_run_hw_queue(hctx)函数将硬件队列 hctx 关联的软件队列中的请求迁移到硬件队列，函数代码简列如下（/block/blk-mq.c）：

```
static void __blk_mq_run_hw_queue(struct blk_mq_hw_ctx *hctx)
{
    struct request_queue *q = hctx->queue;        /*请求队列*/
    struct request *rq;
    LIST_HEAD(rq_list);          /*临时双链表*/
    LIST_HEAD(driver_list);
    struct list_head *dptr;
    int queued;

    WARN_ON(!cpumask_test_cpu(raw_smp_processor_id(), hctx->cpumask));

    if (unlikely(test_bit(BLK_MQ_S_STOPPED, &hctx->state)))        /*硬件队列停止*/
        return;

    hctx->run++;          /*运行次数加 1*/
    flush_busy_ctxs(hctx, &rq_list);      /*迁移软件队列中请求至临时双链表，/block/blk-mq.c*/
                    /*将 hctx->ctx_map 位图中置 1 位对应的软件队列中请求迁入临时双链表*/

    if (!list_empty_careful(&hctx->dispatch)) {
        spin_lock(&hctx->lock);
        if (!list_empty(&hctx->dispatch))
            list_splice_init(&hctx->dispatch, &rq_list); /*hctx->dispatch 双链表成员也迁入临时链表*/
        spin_unlock(&hctx->lock);
    }

    dptr = NULL;
    queued = 0;
    while (!list_empty(&rq_list)) {              /*遍历临时双链表中请求*/
        struct blk_mq_queue_data bd;
        int ret;

        rq = list_first_entry(&rq_list, struct request, queuelist);      /*临时链表中第一个请求*/
        list_del_init(&rq->queuelist);       /*请求从链表中移出*/
```

```
            bd.rq = rq;                 /*初始化 blk_mq_queue_data 实例*/
            bd.list = dptr;
            bd.last = list_empty(&rq_list);

            ret = q->mq_ops->queue_rq(hctx, &bd);       /*将请求插入硬件队列*/
            switch (ret) {
            case BLK_MQ_RQ_QUEUE_OK:        /*插入硬件队列成功*/
                queued++;                   /*插入硬件队列的请求数量加 1*/
                continue;
            case BLK_MQ_RQ_QUEUE_BUSY:
                list_add(&rq->queuelist, &rq_list);         /*重新放回临时双链表*/
                __blk_mq_requeue_request(rq);
                break;
            default:
                pr_err("blk-mq: bad return on queue: %d\n", ret);
            case BLK_MQ_RQ_QUEUE_ERROR:
                rq->errors = -EIO;
                blk_mq_end_request(rq, rq->errors);       /*结束请求*/
                break;
            }

            if (ret == BLK_MQ_RQ_QUEUE_BUSY)
                break;

            if (!dptr && rq_list.next != rq_list.prev)
                dptr = &driver_list;
        }       /*遍历临时双链表结束*/

        if (!queued)
            hctx->dispatched[0]++;
        else if (queued < (1 << (BLK_MQ_MAX_DISPATCH_ORDER - 1)))
            hctx->dispatched[ilog2(queued) + 1]++;

        if (!list_empty(&rq_list)) {            /*如果临时双链表中还有请求，再次运行硬件队列*/
            spin_lock(&hctx->lock);
            list_splice(&rq_list, &hctx->dispatch);   /*临时双链表移入 hctx->dispatch 双链表*/
            spin_unlock(&hctx->lock);
            blk_mq_run_hw_queue(hctx, true);          /*运行硬件队列*/
        }
    }
```

简单地说，运行硬件队列就是将关联软件队列中的请求从软件队列移出，调用 mq_ops->queue_rq()函数将请求插入硬件队列。

4 入队函数

由前面的分析可知，blk_mq_ops 实例的 **queue_rq()**函数用于将请求队列插入硬件队列，即将请求添加到 hctx->dispatch 双链表，同时应当激活对硬件队列请求的处理。

queue_rq()函数原型定义如下（/include/linux/blk-mq.h）：

```
typedef int (queue_rq_fn)(struct blk_mq_hw_ctx *, const struct blk_mq_queue_data *);
```

queue_rq_fn()函数中第二个参数为 blk_mq_queue_data 结构体指针，结构体定义如下：

```
struct blk_mq_queue_data {      /*/include/linux/blk-mq.h*/
    struct request *rq;         /*入队的请求*/
    struct list_head *list;     /*双链表节点指针*/
    bool last;
};
```

请求的处理一般也是在一个内核线程中完成，内核线程从硬件队列中取请求，执行数据传输，最后结束请求。

结束请求的接口函数有 blk_mq_end_request()、__blk_mq_end_request()等，下面以 blk_mq_end_request()为例介绍其实现。

blk_mq_end_request()函数定义如下（/block/blk-mq.c）：

```
void blk_mq_end_request(struct request *rq, int error)
{
    if (blk_update_request(rq, error, blk_rq_bytes(rq)))      /*更新请求，/block/blk-core.c*/
        BUG();
    __blk_mq_end_request(rq, error);      /*/block/blk-core.c*/
}
```

__blk_mq_end_request()函数定义如下，用于释放请求：

```
inline void __blk_mq_end_request(struct request *rq, int error)
{
    blk_account_io_done(rq);

    if (rq->end_io) {
        rq->end_io(rq, error);      /*请求结束函数，需调用 blk_mq_free_request(rq)函数*/
    } else {                        /*没有定义 end_io()函数*/
        if (unlikely(blk_bidi_rq(rq)))      /*双向请求*/
            blk_mq_free_request(rq->next_rq);
        blk_mq_free_request(rq);        /*/block/blk-core.c*/
    }
}
```

如果请求没有定义 end_io()函数，则调用 blk_mq_free_request(rq)函数释放请求，函数定义如下：

```
void blk_mq_free_request(struct request *rq)
{
    struct blk_mq_hw_ctx *hctx;
```

```
    struct request_queue *q = rq->q;

    hctx = q->mq_ops->map_queue(q, rq->mq_ctx->cpu);          /*硬件队列*/
    blk_mq_free_hctx_request(hctx, rq);          /*硬件队列释放请求*/
}
```

请求是由硬件队列关联的队列标签管理的，blk_mq_free_hctx_request(hctx, rq)函数用于硬件队列释放请求，这里的释放只是清零队列标签中请求的标记位，标记其为可用，并没有释放请求实例，函数源代码请读者自行阅读。

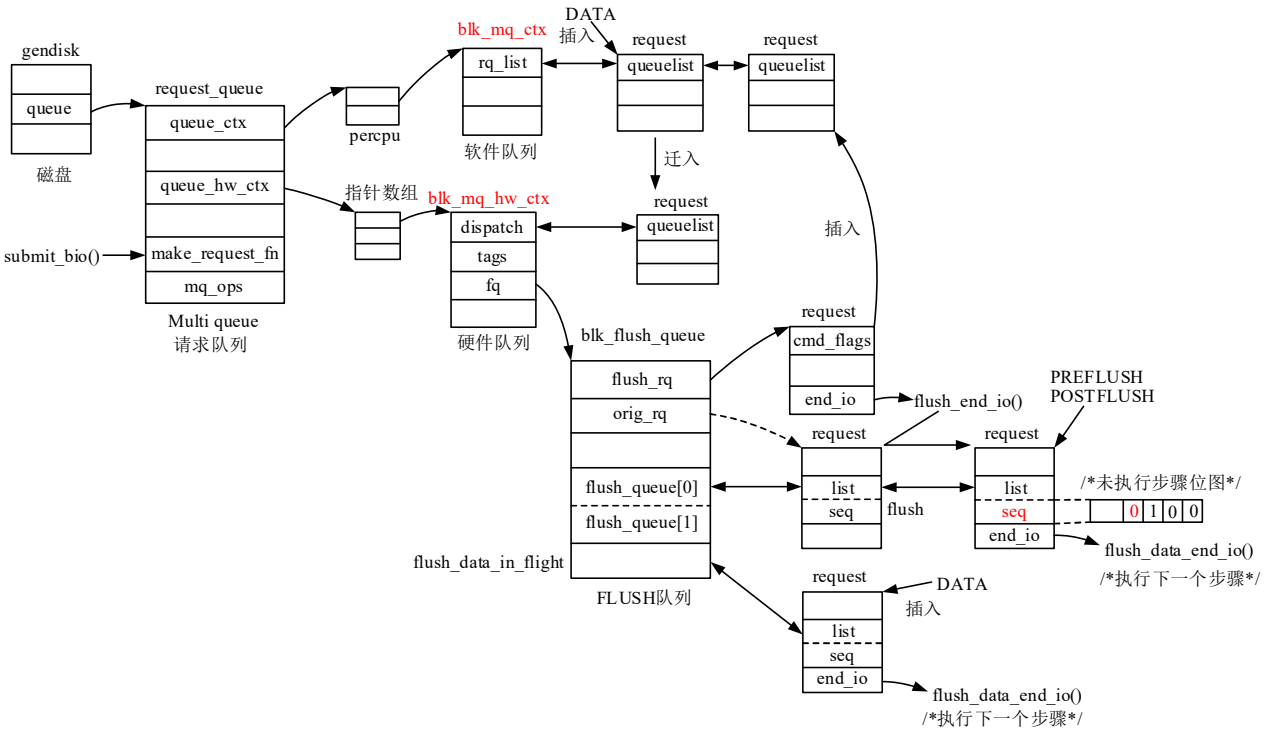
10.5.4 FLUSH/FUA 请求处理

FLUSH 队列和 FLUSH/FUA 请求在前面介绍标准请求队列时已经介绍过了，Multi queue 队列中也有 FLUSH 队列，只不过是由硬件队列管理的，每个硬件队列都有一个 FLUSH 队列。

本小节主要介绍 Multi queue 队列中 FLUSH 队列和 FLUSH/FUA 请求处理与标准请求队列的不同之处。

1 概述

Multi queue 队列中 FLUSH 队列和 FLUSH/FUA 请求的处理如下图所示，注意与标准请求队列的不同之处：



在创建 Multi queue 队列中创建单个硬件队列时，将为硬件队列创建 FLUSH 队列，函数调用关系如下：

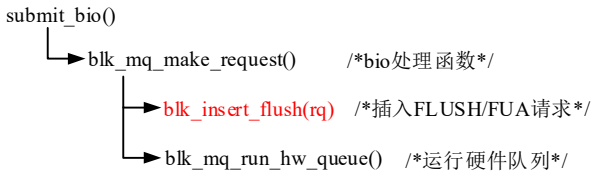
```
blk_mq_init_queue()        /*创建Multi queue请求队列*/
    blk_alloc_queue_node()      /*创建请求队列*/
    blk_mq_init_allocated_queue()  /*初始化Multi queue请求队列*/
        blk_mq_init_hw_queues()  /*创建所有硬件队列*/
            blk_mq_init_hctx()   /*创建单个硬件队列*/
                blk_alloc_flush_queue()  /*创建FLUSH队列*/
```

创建 FLUSH 队列的 blk_alloc_flush_queue()函数前面介绍过了，这里不解释了。

FLUSH/FUA 请求的处理也是一样的，分步骤进行，DATA 步骤是将请求插入软件队列，并运行硬件队列；执行 FLUSH 操作的请求借用了挂起链表中第一个请求的标签（挂起链表第一个请求由 orig_rq 暂管），执行完 FLUSH 操作后标签重新恢复至原挂起链表中第一个请求。

2 插入请求

Multi queue 队列插入 FLUSH/FUA 请求的函数调用关系如下图所示：



blk_insert_flush()函数在前面介绍过了，下面主要看一下与 Multi queue 队列相关的代码：

```
void blk_insert_flush(struct request *rq)            /*/block/blk-flush.c*/
{
    struct request_queue *q = rq->q;
    unsigned int fflags = q->flush_flags;    /*请求队列是否支持 FLUSH 和 FUA*/
    unsigned int policy = blk_flush_policy(fflags, rq);        /*FLUSH 策略，需要执行哪些步骤，位图*/
    struct blk_flush_queue *fq = blk_get_flush_queue(q, rq->mq_ctx);        /*硬件队列中 FLUSH 队列*/
    ...

    /*请求只有 DATA 步骤，添加到软件队列，当作普通请求处理，函数返回*/
    if ((policy & REQ_FSEQ_DATA) &&
        !(policy & (REQ_FSEQ_PREFLUSH | REQ_FSEQ_POSTFLUSH))) {
        if (q->mq_ops) {          /*Multi queue 请求队列*/
            blk_mq_insert_request(rq, false, false, true);        /*插入软件队列，运行硬件队列*/
        } else        /*标准请求队列*/
            list_add_tail(&rq->queuelist, &q->queue_head);   /*请求插入 q->queue_head 双链表末尾*/
        return;       /*函数返回*/
    }

    /*请求包含 PREFLUSH 或 POSTFLUSH 步骤*/
    memset(&rq->flush, 0, sizeof(rq->flush));        /*清零请求内嵌 flush 结构体成员（联合体成员）*/
    INIT_LIST_HEAD(&rq->flush.list);              /*初始化双链表成员*/
    rq->cmd_flags |= REQ_FLUSH_SEQ;         /*标记需 FLUSH 操作，由驱动程序处理*/
    rq->flush.saved_end_io = rq->end_io;     /*保存原 end_io()函数，通常为 NULL，结束请求时恢复*/
    if (q->mq_ops) {                  /*Multi queue 请求队列*/
        rq->end_io = mq_flush_data_end_io;

        spin_lock_irq(&fq->mq_flush_lock);
        blk_flush_complete_seq(rq, fq, REQ_FSEQ_ACTIONS & ~policy, 0);       /*处理请求*/
        spin_unlock_irq(&fq->mq_flush_lock);
        return;
```

```
    }
    /*以下适用于标准请求队列*/
    ...
}
```

Multi queue 队列中也是调用 blk_flush_complete_seq()函数依次执行请求中各个步骤，直至 DONE 步骤，详见下文。

3 处理请求

blk_flush_complete_seq()函数代码简列如下（/block/blk-flush.c）：

```
static bool blk_flush_complete_seq(struct request *rq,struct blk_flush_queue *fq,unsigned int seq, int error)
/*error：是否发生错误*/
{
    struct request_queue *q = rq->q;
    struct list_head *pending = &fq->flush_queue[fq->flush_pending_idx];      /*挂起链表*/
    bool queued = false, kicked;

    BUG_ON(rq->flush.seq & seq);
    rq->flush.seq |= seq;        /*标记 seq 中的步骤已经完成，为 0 的比特位表示尚未执行的步骤*/
                                 /*rq->flush.seq 初始为 0，DONE（bit3）始终为 0*/
    if (likely(!error))
        seq = blk_flush_cur_seq(rq);        /*第一个要执行的步骤（为 0 的位），/block/blk-flush.c*/
    else
        seq = REQ_FSEQ_DONE;            /*error 非 0，表示出错，结束请求*/

    switch (seq) {
    case REQ_FSEQ_PREFLUSH:
    case REQ_FSEQ_POSTFLUSH:
        if (list_empty(pending))        /*挂起双链表为空*/
            fq->flush_pending_since = jiffies;          /*设置挂起双链表等待时间起点为当前时间*/
        list_move_tail(&rq->flush.list, pending);      /*插入挂起双链表末尾*/
        break;

    case REQ_FSEQ_DATA:        /*DATA*/
        list_move_tail(&rq->flush.list, &fq->flush_data_in_flight);    /*插入 flush_data_in_flight 链表*/
        queued = blk_flush_queue_rq(rq, true);
                                /*请求先插入 q->requeue_list 双链表，再插入软件队列，运行硬件队列*/
        break;

    case REQ_FSEQ_DONE:         /*结束请求*/
        BUG_ON(!list_empty(&rq->queuelist));
        list_del_init(&rq->flush.list);          /*从 FLUSH 队列移出*/
        blk_flush_restore_request(rq);      /*恢复请求中成员值，如 end_io()函数，/block/blk-flush.c*/
        if (q->mq_ops)                          /*结束请求*/
```

```
                blk_mq_end_request(rq, error);
            else
                __blk_end_request_all(rq, error);
            break;

        default:
            BUG();
        }

        kicked = blk_kick_flush(q, fq);       /*是否要设置并提交 flush_rq 请求，/block/blk-flush.c*/
        return kicked | queued;       /*返回是否向 q->queue_head 双链表插入了请求*/
    }
```

blk_kick_flush(q, fq)函数用于确定是否要设置 flush_rq 请求并提交，以执行 FLUSH 操作，函数代码如下。

■flush_rq 请求

blk_kick_flush()函数定义如下（/block/blk-flush.c）：

```
static bool blk_kick_flush(struct request_queue *q, struct blk_flush_queue *fq)
{
    struct list_head *pending = &fq->flush_queue[fq->flush_pending_idx];       /*挂起链表*/
    struct request *first_rq =list_first_entry(pending, struct request, flush.list);     /*第一个挂起请求*/
    struct request *flush_rq = fq->flush_rq;            /*flush_rq 请求*/
    ...

    /*需要提交 flush_rq 请求*/
    fq->flush_pending_idx ^= 1;       /*挂起链表转换*/

    blk_rq_init(q, flush_rq);             /*初始化 flush_rq 请求*/

    if (q->mq_ops) {                  /*Multi queue 请求队列*/
        struct blk_mq_hw_ctx *hctx;

        flush_rq->mq_ctx = first_rq->mq_ctx;       /*挂起链表中第一个请求所在软件队列*/
        flush_rq->tag = first_rq->tag;          /*挂起链表中第一个请求标签值*/
        fq->orig_rq = first_rq;                 /*暂存挂起链表中第一个请求*/

        hctx = q->mq_ops->map_queue(q, first_rq->mq_ctx->cpu);        /*硬件队列*/
        blk_mq_tag_set_rq(hctx, first_rq->tag, flush_rq);        /*第一个挂起请求的标签赋予 flush_rq*/
    }

    flush_rq->cmd_type = REQ_TYPE_FS;
    flush_rq->cmd_flags = WRITE_FLUSH | REQ_FLUSH_SEQ;       /*设置 FLUSH 命令*/
```

```
    flush_rq->rq_disk = first_rq->rq_disk;
    flush_rq->end_io = flush_end_io;          /*结束请求回调函数*/

    return blk_flush_queue_rq(flush_rq, false);
                /*请求先插入 q->requeue_list 双链表，再插入软件队列，运行硬件队列，返回 false*/
}
```

blk_kick_flush()函数中提交的 flush_rq 请求，将借用挂起请求链表中第一个请求的标签值，第一个请求由 orig_rq 暂管，在 flush_rq 请求处理完成后再恢复原标签。

下面看一下 flush_rq 请求结束时的回调函数 flush_end_io()，代码如下（/block/blk-flush.c）：

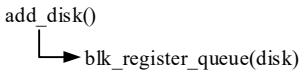
```
static void flush_end_io(struct request *flush_rq, int error)
{
    struct request_queue *q = flush_rq->q;
    struct list_head *running;
    bool queued = false;
    struct request *rq, *n;
    unsigned long flags = 0;
    struct blk_flush_queue *fq = blk_get_flush_queue(q, flush_rq->mq_ctx);      /*FLUSH 队列*/

    if (q->mq_ops) {          /*Multi queue 请求队列*/
        struct blk_mq_hw_ctx *hctx;
        spin_lock_irqsave(&fq->mq_flush_lock, flags);
        hctx = q->mq_ops->map_queue(q, flush_rq->mq_ctx->cpu);    /*硬件队列*/
        blk_mq_tag_set_rq(hctx, flush_rq->tag, fq->orig_rq);  /*恢复挂起链表第一个请求标签*/
        flush_rq->tag = -1;
    }

    running = &fq->flush_queue[fq->flush_running_idx];      /*获取当前运行链表*/
    BUG_ON(fq->flush_pending_idx == fq->flush_running_idx);

    fq->flush_running_idx ^= 1;          /*运行链表转换*/

    if (!q->mq_ops)
        elv_completed_request(q, flush_rq);      /*调用 IO 调度器的完成请求函数，/block/elevator.c*/

    /*遍历原运行队列中请求，执行请求中下一个步骤*/
    list_for_each_entry_safe(rq, n, running, flush.list) {
        unsigned int seq = blk_flush_cur_seq(rq);
                                    /*第一个要执行的步骤，PREFLUSH 或 POSTFLUSH*/

        BUG_ON(seq != REQ_FSEQ_PREFLUSH && seq != REQ_FSEQ_POSTFLUSH);
        queued |= blk_flush_complete_seq(rq, fq, seq, error);
                                    /*屏蔽 PREFLUSH 或 POSTFLUSH 步骤，执行下一个步骤*/
```

```
    }
    ...             /*不触发异常处理请求*/
    fq->flush_queue_delayed = 0;
    if (q->mq_ops)
        spin_unlock_irqrestore(&fq->mq_flush_lock, flags);
}
```

以上代码中除了恢复第一个挂起请求的标签值外，其它操作与标准请求队都是一样的，这里就不再解释了，可参考前一节。

10.5.5 注册请求队列

前面介绍了标准请求队列和 Multi queue 请求队列的创建，创建的请求队列需赋予 gendisk 实例，并调用 add_disk()函数添加 gendisk 实例。在 add_disk()函数中将调用 **blk_register_queue(disk)**函数注册磁盘关联的请求队列，函数调用关系如下图所示。



注册的请求队列可能是标准请求队列也可能是 Multi queue 请求队列，注册操作主要是在 sysfs 中创建表示队列、IO 调度器等的目录和属性文件。

本小节主要介绍注册请求队列 blk_register_queue(disk)函数的实现。

blk_register_queue(disk)函数定义如下（/block/blk-sysfs.c）：

```
int blk_register_queue(struct gendisk *disk)
{
    int ret;
    struct device *dev = disk_to_dev(disk);        /*disk->part0.__dev*/
    struct request_queue *q = disk->queue;         /*请求队列*/
    ...

    if (!blk_queue_init_done(q)) {        /*QUEUE_FLAG_INIT_DONE 标记位是否为 0*/
        queue_flag_set_unlocked(QUEUE_FLAG_INIT_DONE, q);    /*设置标记位*/
        blk_queue_bypass_end(q);        /*离开旁路模式，清 QUEUE_FLAG_BYPASS 标记位*/
                                        /*/block/blk-core.c*/
        if (q->mq_ops)          /*Multi queue 请求队列*/
            blk_mq_finish_init(q);      /*处理 q->mq_usage_counter 成员，/block/blk-mq-sysfs.c*/
    }

    ret = blk_trace_init_sysfs(dev);    /*若没有选择 BLK_DEV_IO_TRACE，返回 0*/
    ...

    ret = kobject_add(&q->kobj, kobject_get(&dev->kobj), "%s", "queue");      /*添加 kobject 实例*/
            /*q->kobj 成员在分配请求队列的 blk_alloc_queue_node()函数中初始化，磁盘为父节点*/
    ...
    kobject_uevent(&q->kobj, KOBJ_ADD);        /*触发 uevent 事件*/
```

```
    if (q->mq_ops)        /*Multi queue 请求队列*/
        blk_mq_register_disk(disk);       /*/block/blk-mq-sysfs.c*/

    if (!q->request_fn)
        return 0;         /*注册 Multi queue 请求队列到此结束*/

    /*以下代码只有注册标准请求队列才会执行*/
    ret = elv_register_queue(q);
    ...       /*错误处理*/
    return 0;
}
```

请求队列 q->kobj 成员（kobject 结构体）在分配请求队列的 blk_alloc_queue_node()函数中初始化，在这里将添加 q->kobj 成员。表示磁盘的 dev->kobj 为其父节点，请求队列在 sysfs 中的目录名称为“queue”，其下包含请求队列属性文件。内核在/block/blk-sysfs.c 文件内定义了请求队列的默认属性及读写属性函数。

对于 Multi queue 请求队列还需要添加 q->mq_kobj 和跟踪软硬件队列的 kobject 结构体成员，对于标准请求队列还需要注册 IO 调度器，下面分别给予介绍。

1 注册 Multi queue 请求队列

blk_mq_register_disk(disk)函数用于注册 Multi queue 请求队列，函数定义如下（/block/blk-mq-sysfs.c）：

```
int blk_mq_register_disk(struct gendisk *disk)
{
    struct device *dev = disk_to_dev(disk);          /*表示磁盘的 device 实例*/
    struct request_queue *q = disk->queue;
    struct blk_mq_hw_ctx *hctx;
    int ret, i;

    blk_mq_sysfs_init(q);        /*初始化 q->mq_kobj 和跟踪软硬件队列的 kobject 结构体成员*/
                                 /*为 mq 队列、软件队列、硬件队列都定义了默认属性*/
    ret = kobject_add(&q->mq_kobj, kobject_get(&dev->kobj), "%s", "mq");
                                    /*在表示磁盘的目录下创建“mq”目录*/
    ...

    kobject_uevent(&q->mq_kobj, KOBJ_ADD);

    queue_for_each_hw_ctx(q, hctx, i) {
        hctx->flags |= BLK_MQ_F_SYSFS_UP;
        ret = blk_mq_register_hctx(hctx);          /*添加跟踪硬件队列的 kobject 成员*/
  /*在“mq”目录下创建子目录，名称为硬件队列编号，在其下添加对应软件队列的 kobject 成员*/
                /*创建子目录，目录名称为“cpuX”，X 为 cpu 核编号*/
        if (ret)
            break;
    }
```

```
    ...
    return 0;
}
```

简要解释一下，blk_mq_register_disk(disk)函数在 sysfs 中表示磁盘的目录下创建“mq”目录，在“mq”目录下创建表示硬件队列的子目录，名称为硬件队列编号，如“mq/0”，在硬件队列子目录下为映射到本队列的软件队列创建子目录，名称为“cpuX”，X 表示软件队列（CPU 核）编号，如“mq/0/cpu0”表示 cpu0 映射到硬件队列 0。在以上目录下都包括属性文件，请读者查阅/block/blk-mq-sysfs.c 文件。

2 注册 IO 调度器

对于标准请求队列需要调用 elv_register_queue()函数注册绑定的 IO 调度器，代码如下：

```
int elv_register_queue(struct request_queue *q)          /*/block/elevator.c*/
{
    struct elevator_queue *e = q->elevator;
    int error;

    error = kobject_add(&e->kobj, &q->kobj, "%s", "iosched");
                                    /*在队列“queue”目录下创建"iosched"目录*/
    if (!error) {
        struct elv_fs_entry *attr = e->type->elevator_attrs;          /*IO 调度器属性*/
        if (attr) {
            while (attr->attr.name) {
                if (sysfs_create_file(&e->kobj, &attr->attr))        /*创建属性文件*/
                    break;
                attr++;
            }
        }
        kobject_uevent(&e->kobj, KOBJ_ADD);
        e->registered = 1;
        if (e->type->ops.elevator_registered_fn)
            e->type->ops.elevator_registered_fn(q);        /*注册函数*/
    }
    return error;
}
```

elv_register_queue()函数主要在是表示队列的“queue”目录下创建表示 IO 调度器的“iosched”目录，并创建 IO 调度器属性文件。

10.6 VFS 层块设备操作

前面介绍了块设备驱动程序的实现，以及驱动程序如何通过请求队列来实现块设备的读写操作。块设备读写操作被封装成 bio 实例，提交到请求队列，而提交 bio 是来自于 VFS 的。VFS 如何建立与块设备之间的联系呢？尤其是通过普通文件访问块设备时，用户其实并不知道文件保存在哪个块设备（分区）上。本节将解决 VFS 与块设备之间的对接问题。

10.6.1 概述

请读者先看下图。

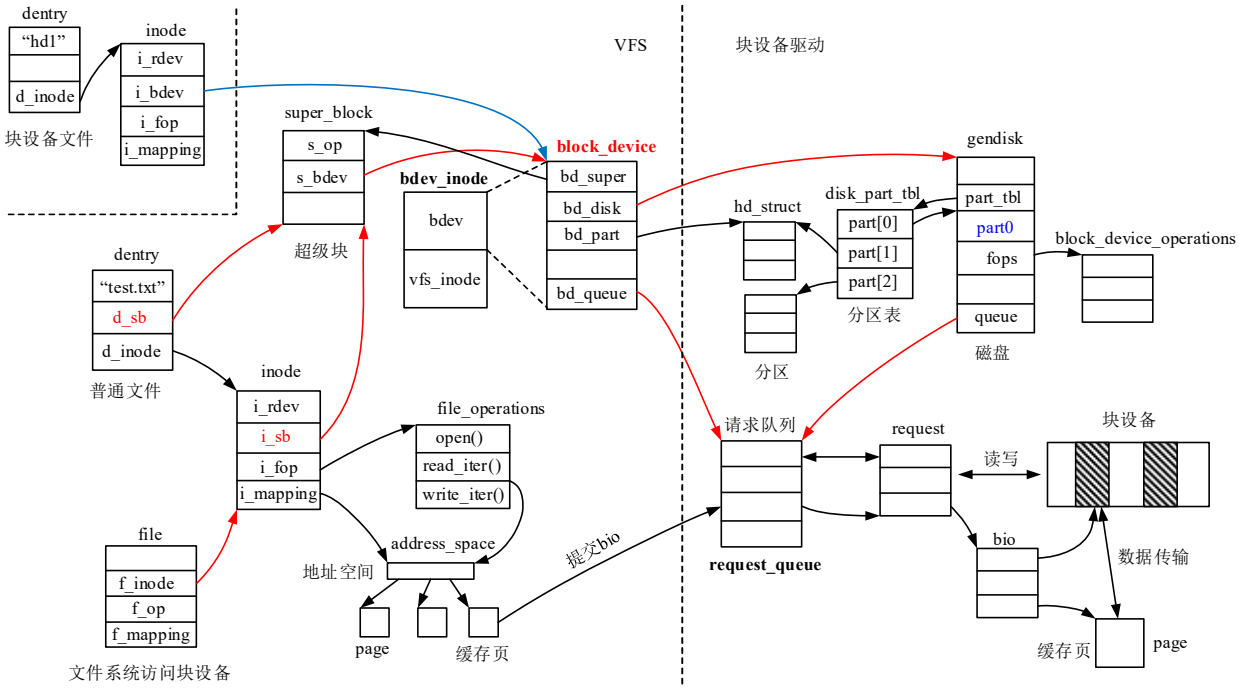
在 VFS 中每个磁盘和分区由一个 block_device 结构体实例表示，它与块设备驱动中的 hd_struct 实例一一对应。

block_device 结构体内嵌在 bdev_inode 结构体中，bdev_inode 结构体中还包含一个 inode 结构体成员。inode 结构体成员将作为 bdev 伪文件系统中的节点 inode 实例，bdev 伪文件系统通过此 inode 实例来管理 bdev_inode 实例，进而管理 block_device 实例。内核通过 bdev 伪文件系统来查找、创建 block_device 实例。

在前面介绍的添加磁盘的操作中将为磁盘创建 block_device 实例，block_device 实例关联到 gendisk、hd_struct 和 request_queue 实例（来自 gendisk 实例）。block_device 实例是 VFS 与块设备驱动之间的桥梁。

磁盘分区要挂载到内核根文件系统中，才能被内核识别。在挂载分区的操作中，将打开分区设备文件，查找（创建）分区对应的 block_device 实例，并关联到分区设备文件 inode 实例。挂载操作中为分区创建的超级块 super_block 实例也将关联到 block_device 实例。

当打开挂载分区中的文件时，其 dentry 和 inode 实例将关联到分区超级块 super_block 实例，进而关联上表示分区的 block_device 实例，通过 block_device 实例建立文件与块设备驱动（请求队列）之间的关联。分区中文件的读写操作通过地址空间（页缓存）转换成对分区中数据块的访问，从而构建 bio 实例，提交到 block_device 实例关联的请求队列。



用户通过块设备文件对裸块设备进行访问或控制时，也需要先打开块设备文件，此时块设备文件 inode 将关联表示块设备的 block_device 实例，如果实例不存在则创建。对裸块设备的读写访问也将转换成对块设备数据块的访问，从而构建 bio 实例，提交到 block_device 实例关联的请求队列。对块设备的控制操作，将调用 block_device 实例对应磁盘 gendisk 实例关联的 block_device_operations 实例中的函数完成。

本将先介绍管理 block_device 实例的 bdev 伪文件系统，然后介绍挂载分区操作中 block_device 实例的创建最后介绍裸块设备的操作，普通文件的读写操作下一章再介绍。

10.6.2 bdev 伪文件系统

内核定义了 bdev 伪文件系统用于管理在 VFS 中表示块设备的 block_device 实例。bdev 伪文件系统实

际上并不是完整意义上的文件系统，内核只是借用其 inode 管理功能，用于跟踪管理 bdev_inode 结构体实例。

1 block_device

bdev_inode 结构体定义在/fs/block_dev.c 文件内：

```
struct bdev_inode {
    struct block_device    bdev;          /*内嵌 block_device 结构体*/
    struct inode    vfs_inode;            /*bdev 伪文件系统中 inode 实例*/
};
```

bdev_inode 结构体中 vfs_inode 成员是 inode 结构体实例，即 bdev 伪文件系统中的 inode 实例，bdev 成员为 block_device 结构体实例，用于用于表示块设备。

block_device 结构体定义在/include/linux/fs.h 头文件内：

```
struct block_device {
    dev_t               bd_dev;          /*块设备号，用于唯一标识 block_device 实例*/
    int                 bd_openers;      /*块设备被打开的次数*/
    struct inode *      bd_inode;        /*指向 bdev_inode.vfs_inode，未来将删除*/
    struct super_block *    bd_super;    /*指向挂载分区文件系统时创建的超级块实例*/
    struct mutex        bd_mutex;        /*打开/关闭互斥量*/
    struct list_head    bd_inodes;       /*对应设备文件 inode 链表，链表元素为 inode.i_devices*/
    void *          bd_claiming;         /*独占打开设备的持有者*/
    void *          bd_holder;           /*实例持有者*/
    int             bd_holders;
    bool            bd_write_holder;
#ifdef CONFIG_SYSFS
    struct list_head    bd_holder_disks;
#endif
    struct block_device *    bd_contains;   /*所属磁盘的 block_device 实例（表示整个磁盘）*/
    unsigned        bd_block_size;          /*内核看到的文件系统中数据块大小*/
    struct hd_struct *    bd_part;          /*指向分区 hd_struct 结构体实例*/
    unsigned        bd_part_count;          /*使用计数，计算内核中引用该设备内分区的次数*/
    int             bd_invalidated;         /*1 表示分区信息无效，要重新获取分区信息*/
    struct gendisk *    bd_disk;            /*指向通用磁盘 gendisk 实例*/
    struct request_queue *    bd_queue;     /*指向 gendisk 中请求队列*/
    struct list_head    bd_list;      /*将实例添加到全局 all_bdevs 双链表*/
    unsigned long       bd_private;   /*存储特定于持有者的信息*/

    int                 bd_fsfreeze_count;
    struct mutex        bd_fsfreeze_mutex;
};
```

block_device 结构体主要成员简介如下：

●**bd_dev**：块设备号，在 bdev 伪文件系统中用于唯一标识 block_device 实例，查找实例时以它为键值。

●**bd_super**：在挂载文件系统的操作（mount_bdev()）中设置为指向超级块 super_block 实例。

●**bd_inodes**：链接块设备文件 inode，链表元素为 inode.i_devices。一个块设备可以在根文件系统中由

多个设备文件同时表示，也就是说可以多个块设备文件对应同一个块设备，只要其中的设备类型和设备号相同即可。

●**bd_contains**：指向表示整个磁盘的 block_device 实例。

●**bd_block_size**：格式化文件系统时设置的数据块大小，即字节数，VFS 读写块设备的单位。

●**bd_part**：指向对应的分区 hd_struct 结构体实例。

●**bd_invalidated**：置 1 表示 hd_struct 实例中分区信息无效，需要扫描磁盘中的分区信息。

●**bd_disk**：指向磁盘 gendisk 结构体实例。

●**bd_queue**：请求队列指针，非常重要的成员，指向 gendisk 实例中的请求队列。

●**bd_list**：双链表元素，用于将 block_device 实例链接到全局 block_device 实例链表，表头为 all_bdevs，定义在/fs/block_dev.c 文件内。

2 挂载 bdev 伪文件系统

内核在/fs/block_dev.c 文件内定义了 bdev 伪文件系统类型实例:

```
static struct file_system_type bd_type = {
        .name     = "bdev",
        .mount    = bd_mount,       /*文件系统类型挂载函数*/
        .kill_sb  = kill_anon_super,
};
```

内核在虚拟文件系统初始化函数 vfs_cache_init()中调用 bdev_cache_init()函数，完成 bdev 伪文件系统类型的注册、内核挂载等工作。

bdev_cache_init()函数定义在/fs/block_dev.c 文件内：

```
void __init bdev_cache_init(void)
{
    int err;
    struct vfsmount *bd_mnt;

    bdev_cachep = kmem_cache_create("bdev_cache", sizeof(struct bdev_inode),
            0, (SLAB_HWCACHE_ALIGN|SLAB_RECLAIM_ACCOUNT|
            SLAB_MEM_SPREAD|SLAB_PANIC),init_once);   /*创建 bdev_inode 结构体 slab 缓存*/
    err = register_filesystem(&bd_type);          /*注册伪文件系统类型*/
    ...          /*输出信息*/
    bd_mnt = kern_mount(&bd_type);          /*内核挂载 bdev 伪文件系统*/
     ...          /*输出信息*/
     blockdev_superblock = bd_mnt->mnt_sb;   /*blockdev_superblock 指向伪文件系统超级块实例*/
}
```

初始化函数中完成 bdev_inode 结构全 slab 缓存的创建、bdev 伪文件系统类型的注册及内核挂载操作，全局变量 blockdev_superblock 指向挂载 bdev 伪文件系统时创建的超级块实例。

这里我们来看一下内核挂载函数 kern_mount(&bd_type)的执行结果，函数定义在/include/linux/fs.h 头文件内，参数 type 为文件系统类型结构指针：

```
#define kern_mount(type)     kern_mount_data(type, NULL)
```

kern_mount_data()函数定义在/fs/namespace.c 文件内：

```
struct vfsmount *kern_mount_data(struct file_system_type *type, void *data)
{
    struct vfsmount *mnt;
    mnt = vfs_kern_mount(type, MS_KERNMOUNT, type->name, data);
    if (!IS_ERR(mnt)) {
        real_mount(mnt)->mnt_ns = MNT_NS_INTERNAL;
    }
    return mnt;
}
```

kern_mount_data()函数内直接调用我们熟悉的内核挂载函数 vfs_kern_mount()，注意标记参数设置了 MS_KERNMOUNT 标记位，请读者回顾第 7 章的内容。

内核挂载函数将调用文件系统类型实例中定义的 mount()函数，创建文件系统超级块 super_block、根目录项 dentry 和 inode 结构体实例。kern_mount_data()函数并没有将 bdev 伪文件系统根目录项 dentry 实例关联到挂载点，也就是说文件系统没有导出到内核的根文件系统中。

bdev 伪文件系统类型实例中 mount()函数为 bd_mount()，定义在/fs/block_dev.c 文件内：

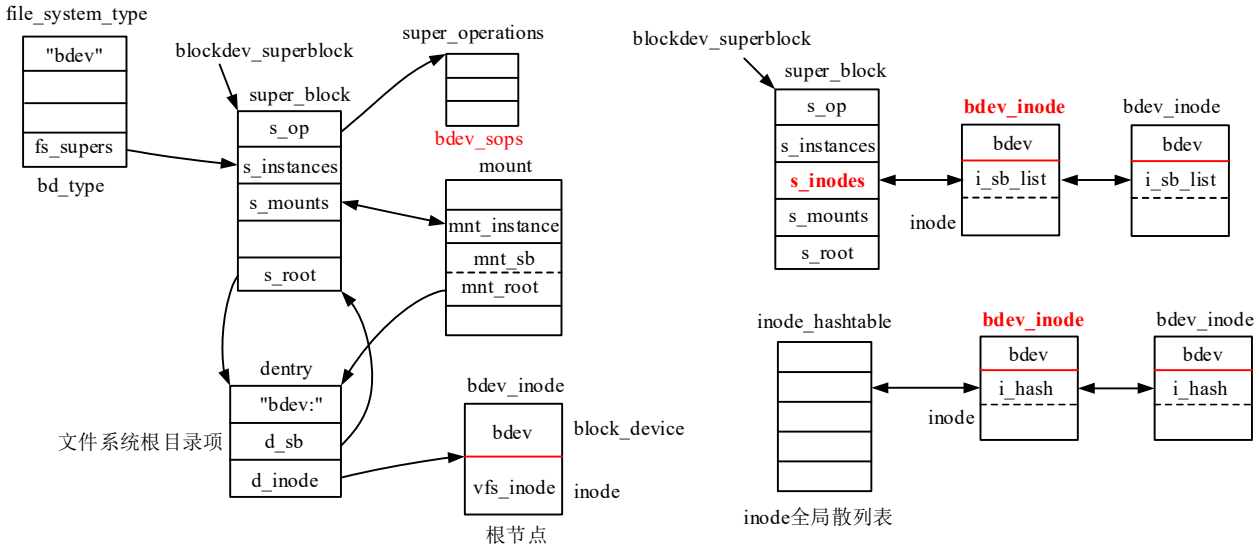
```
static struct dentry *bd_mount(struct file_system_type *fs_type,int flags, const char *dev_name, void *data)
{
    return mount_pseudo(fs_type, "bdev:", &bdev_sops, NULL, BDEVFS_MAGIC);   /*/fs/libfs.c*/
                /*bdev_sops：超级块操作结构实例指针，BDEVFS_MAGIC：bdev 伪文件系统魔数*/
}
```

mount_pseudo()函数是挂载伪文件系统的标准函数，定义在/fs/libfs.c 文件内，代码如下：

```
struct dentry *mount_pseudo(struct file_system_type *fs_type, char *name,    \
    const struct super_operations *ops,const struct dentry_operations *dops, unsigned long magic)
{
    struct super_block *s;
    struct dentry *dentry;
    struct inode *root;
    struct qstr d_name = QSTR_INIT(name, strlen(name));

    s = sget(fs_type, NULL, set_anon_super, MS_NOUSER, NULL);    /*创建超级块结构实例*/
    if (IS_ERR(s))
        return ERR_CAST(s);

    s->s_maxbytes = MAX_LFS_FILESIZE;
    s->s_blocksize = PAGE_SIZE;          /*数据块大小默认值设置为内存页大小*/
    s->s_blocksize_bits = PAGE_SHIFT;
    s->s_magic = magic;
    s->s_op = ops ? ops : &simple_super_operations;      /*参数传递了 bdev_sops 实例指针*/
    s->s_time_gran = 1;
    root = new_inode(s);          /*调用 bdev_sops 实例中创建 inode 的函数，创建 bdev_inode 实例*/
    if (!root)
```

```
        goto Enomem;
    root->i_ino = 1;
    root->i_mode = S_IFDIR | S_IRUSR | S_IWUSR;
    root->i_atime = root->i_mtime = root->i_ctime = CURRENT_TIME;
    dentry = __d_alloc(s, &d_name);      /*创建 bdev 伪文件系统根目录项 dentry 实例*/
    if (!dentry) {
        iput(root);
        goto Enomem;
    }
    d_instantiate(dentry, root);      /*关联 dentry 和 inode 实例*/
    s->s_root = dentry;
    s->s_d_op = dops;          /*传递参数为 NULL*/
    s->s_flags |= MS_ACTIVE;
    return dget(s->s_root);
    ...
}
```

挂载函数首先调用 sget()（详见第 7 章）创建超级块 super_block 实例，并对其进行初始化，然后调用**new_inode(s)**函数创建文件系统根目录项对应的 bdev_inode 实例（调用 inode_init_always(sb, inode)函数初始化其中 inode 实例并插入到超级块表），最后创建文件系统根目录项对应的 dentry 实例，并建立其与新创建 super_block 和 inode 实例的关联，如下图所示（右侧为内核对 inode 实例的管理结构）。



bdev 伪文件系统超级块操作结构实例为 **bdev_sops**（参数传递），定义如下（/fs/block_dev.c）：

```
static const struct super_operations bdev_sops = {
    .statfs = simple_statfs,
    .alloc_inode = bdev_alloc_inode,      /*创建 inode 实例函数*/
    .destroy_inode = bdev_destroy_inode,
    .drop_inode = generic_delete_inode,
    .evict_inode = bdev_evict_inode,
};
```

bdev_sops 实例中创建 inode 实例的函数定义如下，函数内直接从 bdev_inode 结构体 slab 缓存中分配实例，返回实例中 vfs_inode 成员（inode 实例）指针。

```
static struct inode *bdev_alloc_inode(struct super_block *sb)
{
    struct bdev_inode *ei = kmem_cache_alloc(bdev_cachep, GFP_KERNEL);
    if (!ei)
        return NULL;
    return &ei->vfs_inode;     /*返回 bdev_inode.vfs_inode 成员指针*/
}
```

3 查找/创建 block_device 实例

在打开块设备文件时，会将块设备号写入 inode->i_rdev 成员。bdget(inode->i_rdev)函数用于根据块设备号在 bdev 伪文件系统中为块设备查找或创建对应 bdev_inode 结构体实例，并返回其中的 block_device 结构体成员指针。

bdget()函数在/fs/block_dev.c 文件内实现，代码如下：

```
struct block_device *bdget(dev_t dev)
/*dev：块设备号，用于唯一标识块设备，查找的键值*/
{
    struct block_device *bdev;
    struct inode *inode;        /*指向 bdev_inode.vfs_inode 成员*/

    inode = iget5_locked(blockdev_superblock, hash(dev),bdev_test, bdev_set, &dev);/*/fs/inode.c*/
                                /*查找或创建 bdev_inode 实例，返回 bdev_inode.vfs_inode 指针*/
    if (!inode)
        return NULL;

    bdev = &BDEV_I(inode)->bdev;       /*block_device 实例指针*/

    if (inode->i_state & I_NEW) {          /*初始化新创建 bdev_inode 实例*/
        bdev->bd_contains = NULL;
        bdev->bd_super = NULL;
        bdev->bd_inode = inode;            /*指向 bdev_inode.vfs_inode*/
        bdev->bd_block_size = (1 << inode->i_blkbits);   /*设置数据块大小值为 inode->i_blkbits*/
                    /*inode->i_blkbits 来源于超级块 s->s_blocksize 成员，这里为页大小*/
        bdev->bd_part_count = 0;
        bdev->bd_invalidated = 0;          /*分区信息无效*/
        inode->i_mode = S_IFBLK;           /*文件类型为块设备文件，bdev_inode.vfs_inode*/
        inode->i_rdev = dev;               /*块设备号*/
        inode->i_bdev = bdev;              /*指向 block_device 实例*/
        inode->i_data.a_ops = &def_blk_aops;     /*地址空间操作结构实例，/fs/block_dev.c*/
        mapping_set_gfp_mask(&inode->i_data, GFP_USER);
        spin_lock(&bdev_lock);
        list_add(&bdev->bd_list, &all_bdevs);          /*block_device 添加到全局 all_bdevs 链表头部*/
        spin_unlock(&bdev_lock);
        unlock_new_inode(inode);
```

```
    }
    return bdev;
}
```

bdget(dev)函数调用 iget5_lock()函数在 bdev 伪文件系统中查找或创建 bdev_inode 实例，块设备号将作为查找实例的键值，bdev_test()和 bdev_set()函数分别用来测试和设置 bdev.bd_dev 成员值（块设备号）。

如果 iget5_lock()函数创建了新 bdev_inode 实例，则对其进行初始化，不是新创建的则不需要初始化，bdget(dev)函数最后返回 bdev_inode 实例中 bdev 成员指针。

注意，bdget(dev)函数中初始化的 inode 实例是 bdev_inode 实例中内嵌的 vfs_inode 成员（inode 实例），而不是打开块设备文件的 inode 实例。

10.6.2 挂载中打开块设备

块设备（分区）执行最多的就是挂载（卸载）操作，挂载系统调用中需要传递挂载分区的设备文件。在挂载操作中将打开块设备文件，为其查找或创建 block_device 实例，并建立 block_device 实例与 gendisk、hd_struct 和 request_queue 等实例之间的关联（在添加磁盘时已经创建了表示磁盘的 block_device 实例）。

挂载操作中以上工作由 **blkdev_get_by_path()**函数完成，函数调用关系如下所示：

```
mount()                 /*挂载系统调用*/
  └► do_mount()
       └► do_new_mount()        /*新挂载*/
            └► vfs_kern_mount()  /*内核挂载*/
                 └► mount_fs()
                      └► type->mount()  /*文件系统类型挂载函数*/
                           ext2_mount()   /*ext2挂载系统函数*/
                             └► mount_bdev()    /*标准挂函数*/
                                  └► bdev = blkdev_get_by_path(dev_name, mode, fs_type)
```

对于基于外部存储介质的文件系统类型，其挂载函数内通常调用 **mount_bdev()**函数完成内核挂载操作，mount_bdev()函数内调用 **blkdev_get_by_path()**函数执行对块设备文件的打开操作，并返回 block_device 实例指针。

blkdev_get_by_path()函数定义在/fs/block_dev.c 文件内，代码如下：

```
struct block_device *blkdev_get_by_path(const char *path, fmode_t mode,void *holder)
/*path：块设备文件名，mode：模式参数，holder：指向文件系统类型实例*/
{
    struct block_device *bdev;
    int err;

    bdev = lookup_bdev(path);       /*打开块设备文件，返回 block_device 实例指针，/fs/block_dev.c*/
    if (IS_ERR(bdev))
        return bdev;

    err = blkdev_get(bdev, mode, holder);     /*打开块设备，/fs/block_dev.c*/
```

```
    if (err)
        return ERR_PTR(err);

    if ((mode & FMODE_WRITE) && bdev_read_only(bdev)) {
        blkdev_put(bdev, mode);
        return ERR_PTR(-EACCES);
    }
    return bdev;        /*返回 block_device 实例*/
}
```

blkdev_get_by_path()函数执行的工作主要分两步：

（1）调用 lookup_bdev(path)函数根据块设备文件名称，打开块设备文件，为其创建 dentry 和 inode 实例，并查找或创建对应的 block_device 实例，返回 block_device 实例指针。

（2）调用 blkdev_get()函数执行打开块设备操作，主要是关联对应的 gendisk、hd_struct 和 request_queue 等实例。

下面分别介绍这两步执行的工作。

1 查找 block_device 实例

lookup_bdev(path)函数用于打开块设备文件，查找（创建）block_device 实例，定义如下（/fs/block_dev.c）：

```
struct block_device *lookup_bdev(const char *pathname)
{
    struct block_device *bdev;
    struct inode *inode;
    struct path path;
    int error;

    if (!pathname || !*pathname)
        return ERR_PTR(-EINVAL);

    error = kern_path(pathname, LOOKUP_FOLLOW, &path);
                                        /*内核发起打开块设备文件操作，见第 7 章*/
    if (error)
        return ERR_PTR(error);

    inode = d_backing_inode(path.dentry);      /*返回块设备文件 inode 实例，dentry->d_inode*/
    error = -ENOTBLK;
    if (!S_ISBLK(inode->i_mode))
        goto fail;
    error = -EACCES;
    if (path.mnt->mnt_flags & MNT_NODEV)
        goto fail;
    error = -ENOMEM;
    bdev = bd_acquire(inode);      /*获取 block_device 实例，/fs/block_dev.c*/
    if (!bdev)
```

```
            goto fail;
    out:
        path_put(&path);
        return bdev;        /*返回 block_device 实例指针*/
    fail:
        ...
    }
```

lookup_bdev(path)函数调用 kern_path()函数依路径名查找块设备文件，为其创建 dentry 和 inode 结构体实例，然后调用 **bd_acquire(inode)**函数查找或创建表示块设备 block_device 实例，并返回实例指针。

依路径名打开文件的操作在前面第 7 章已经介绍过了，这里主要看一下 bd_acquire(inode)函数的实现。

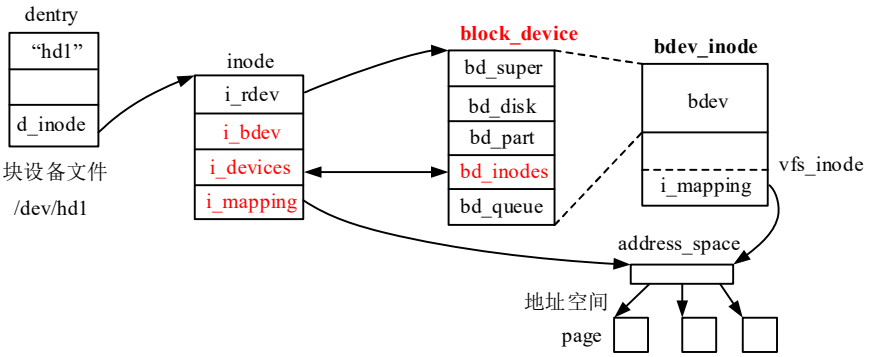
■bd_acquire()

bd_acquire(inode)定义在/fs/block_dev.c 文件内，代码如下：

```
static struct block_device *bd_acquire(struct inode *inode)
/*inode：表示块设备文件的 inode 实例指针*/
{
    struct block_device *bdev;

    spin_lock(&bdev_lock);
    bdev = inode->i_bdev;       /*块设备文件 inode 指向的 block_device 实例*/
    if (bdev) {                 /*如果 inode->i_bdev 已经存在，则增加 block_device 引用计数即可*/
        ihold(bdev->bd_inode);          /*增加 bdev_inode.vfs_inode 引用计数*/
        spin_unlock(&bdev_lock);
        return bdev;                /*返回 block_device 实例指针*/
    }
    spin_unlock(&bdev_lock);

    /*如果 inode->i_bdev 为指针为空，则需要查找或创建 block_device 实例*/
    bdev = bdget(inode->i_rdev);        /*以块设备号查找或创建 block_device 实例，见上一小节*/
    if (bdev) {                 /*查找或创建了 block_device 实例*/
        spin_lock(&bdev_lock);
        if (!inode->i_bdev) {           /*建立块设备文件 inode 与 block_device 实例之间关联*/
            ihold(bdev->bd_inode);
            inode->i_bdev = bdev;               /*指向 block_device 实例*/
            inode->i_mapping = bdev->bd_inode->i_mapping;       /*块设备文件地址空间指针*/
            list_add(&inode->i_devices, &bdev->bd_inodes);
              /*将设备文件 inode 添加到 bdev->bd_inodes 双链表，同个块设备可有多个设备文件*/
        }
        spin_unlock(&bdev_lock);
    }
    return bdev;                /*返回 block_device 实例指针*/
```

}

bd_acquire(inode)函数首先判断块设备文件 inode->i_bdev（指向 block_device 实例）是否为空，若不为空则增加 bdev_inode.vfs_inode 引用计数后，即可返回 block_device 实例指针。若为空则表示设备文件未被打开，则需要在 bdev 伪文件系统中查找或创建 bdev_inode 实例，由 bdget(inode->i_rdev)函数完成这项工作，并返回 block_device 实例指针。bd_acquire(inode)函数最后需建立块设备文件 inode 实例与 block_device 实例之间的关联，如下图所示。



2 blkdev_get()

前面介绍的 lookup_bdev(path)函数已经打开了块设备文件并查找（创建）了 block_device 实例。挂载操作调用的 blkdev_get_by_path()函数下一步就是调用 blkdev_get(bdev, filp->f_mode, filp)函数打开块设备。

blkdev_get()函数内主要工作由__blkdev_get()函数完成，在添加磁盘时也会调用 blkdev_get()函数，以打开磁盘，这个操作在挂载磁盘分区之前进行。

blkdev_get()函数定义在/fs/block_dev.c 文件内，代码如下：

```
int blkdev_get(struct block_device *bdev, fmode_t mode, void *holder)
/*bdev：block_device 实例指针，mode：访问模式，如 FMODE_EXCL，holder：块设备持有者*/
{
    struct block_device *whole = NULL;     /*指向表示磁盘的 block_device 实例*/
    int res;

    WARN_ON_ONCE((mode & FMODE_EXCL) && !holder);

    if ((mode & FMODE_EXCL) && holder) {          /*如果要求独占设备*/
        whole = bd_start_claiming(bdev, holder);
            /*请求独占块设备，返回表示整个磁盘的 block_device 实例指针，/fs/block_dev.c*/
        ...    /*错误处理*/
     }

     res = __blkdev_get(bdev, mode, 0);    /*执行打开操作，成功返回 0，详见下文，/fs/block_dev.c*/

     /*如果打开块设备文件设置了 O_EXCL 标记，继续执行以下 if 语句，没有设置返回 0*/
     if (whole) {
          struct gendisk *disk = whole->bd_disk;        /*磁盘*/
          mutex_lock(&bdev->bd_mutex);
```

```
            spin_lock(&bdev_lock);

            if (!res) {
                BUG_ON(!bd_may_claim(bdev, whole, holder));
                whole->bd_holders++;
                whole->bd_holder = bd_may_claim;    /*磁盘持有者函数指针，/fs/block_dev.c*/
                bdev->bd_holders++ ;                /*分区持有者*/
                bdev->bd_holder = holder;
            }

            /* tell others that we're done */
            BUG_ON(whole->bd_claiming != holder);
            whole->bd_claiming = NULL;
            wake_up_bit(&whole->bd_claiming, 0);

            spin_unlock(&bdev_lock);

            if (!res && (mode & FMODE_WRITE) && !bdev->bd_write_holder &&
                (disk->flags & GENHD_FL_BLOCK_EVENTS_ON_EXCL_WRITE)) {
                bdev->bd_write_holder = true;
                disk_block_events(disk);
            }
            mutex_unlock(&bdev->bd_mutex);
            bdput(whole);
        }
        return res;        /*成功返回 0*/
    }
```

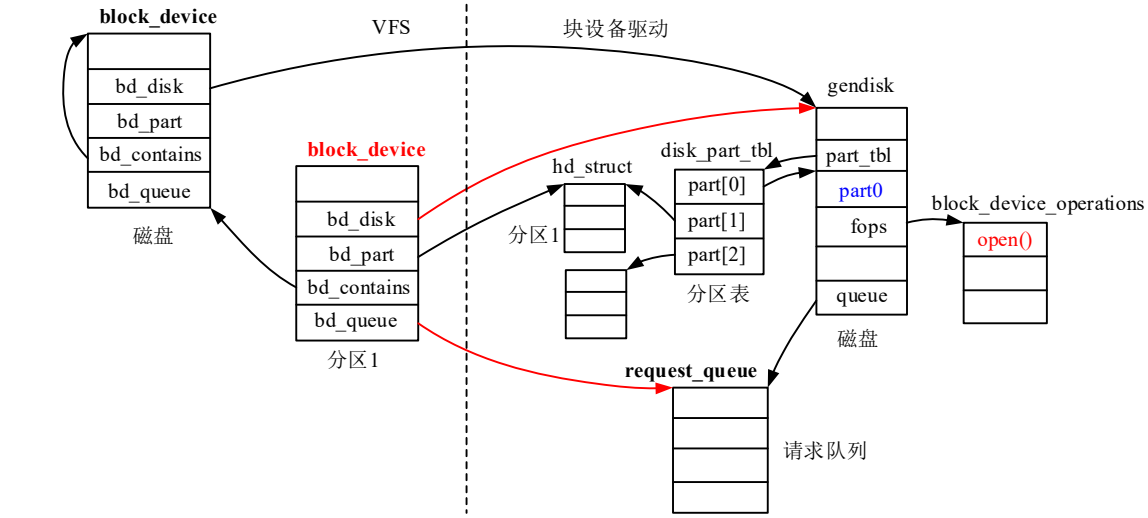
blkdev_get()函数内主要是调用**__blkdev_get**(bdev, mode, 0)函数完成打开块设备操作。打开块设备的目的是建立 block_device 实例与 gendisk、hd_struct 和 request_queue 实例之间的关联，实现 VFS 与块设备驱动的对接。下面将详细介绍**__blkdev_get**()函数的实现。

■__blkdev_get()

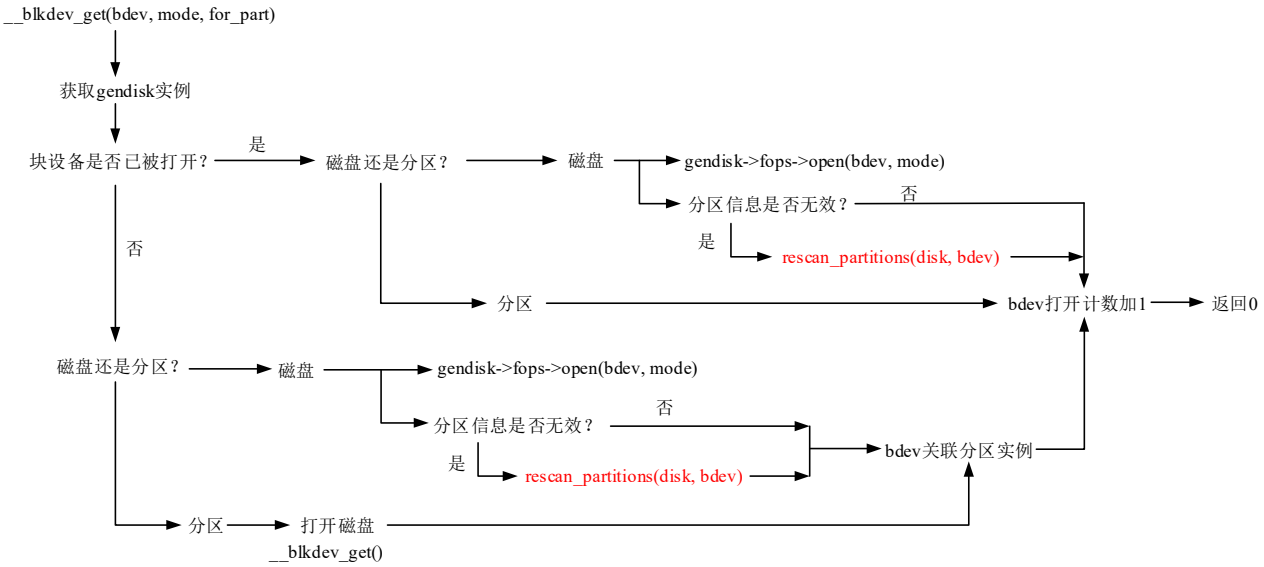
__blkdev_get()函数可用于打开磁盘和分区。打开分区时，如果磁盘还未打开将先执行打开磁盘操作，再执行打开分区操作。

打开磁盘操作主要是调用 block_device_operations.open()函数执行底层打开操作，然后扫描磁盘分区，填充和添加分区 hd_struct 实例，将表示磁盘的 block_device 实例关联到 gendisk、hd_struct（gendisk->part0）和 request_queue 实例。

打开分区操作建立在磁盘已打开的基础上，也就是分区 hd_struct 实例已经存在了。打开分区操作主要是将表示分区的 block_device 实例关联到 gendisk、hd_struct 和 request_queue 实例，将 block_device 实例的 bd_contains 成员指向表示磁盘的 block_device 实例，如下图所示。



__blkdev_get(bdev,mode, for_part)函数执行流程如下图所示：



参数 bdev 是打开块设备的 block_device 实例指针，mode 表示打开块设备文件时传递的模式参数，参数 for_part 若为 1 表示打开的是磁盘，但是在打开分区的操作中触发的。

__blkdev_get()函数首先判断块设备是否已打开，如果已打开（执行过__blkdev_get()函数），则执行流程如下：

（1）如果 bdev 表示的是磁盘，则调用 gendisk->fops->open(bdev, mode)函数执行磁盘底层打开操作，即 block_device_operations 实例中 open()函数，然后判断分区信息是否无效，如果分区信息有效，bdev 打开计数加 1 后函数返回；如果分区信息无效再调用函数 rescan_partitions(disk, bdev)从磁盘中获取分区信息，添加 hd_struct 实例。bdev 打开计数加 1 后函数返回。

（2）如果 bdev 表示的是分区，则 bdev 打开计数加 1 后即可返回。

如果块设备尚未被打开，则执行流程如下：

（1）如果 bdev 表示的是磁盘，则调用 gendisk->fops->open(bdev, mode)函数执行磁盘底层打开操作，然后判断分区信息是否无效，若有效则 bdev 打开计数加 1 后，函数返回；如果分区信息无效则调用函数 rescan_partitions(disk, bdev)获取分区信息，添加 hd_struct 实例，bdev 实例关联 hd_struct 实例，引用计数加 1，函数返回。

（2）如果 bdev 表示的是分区，则先执行打开磁盘操作，这里的打开磁盘操作是__blkdev_get()函数的

递归调用，也要区分磁盘已打开还是未打开的情形；然后查找分区对应 hd_struct 实例，bdev 实例关联到 hd_struct 实例，增加其打开计数，函数返回。

扫描磁盘分区的 rescan_partitions(disk, bdev)函数在前面介绍添加磁盘操作时已经介绍过了。

__blkdev_get()函数在/fs/block_dev.c 文件内实现，代码如下：

```
static int __blkdev_get(struct block_device *bdev, fmode_t mode, int for_part)
/*
*bdev：指向打开块设备 block_device 实例，mode：系统调用传递的模式参数，
*for_part：这里固定为 0，为 1 表示本次打开是由打开其它分区时触发的。
*/
{
    struct gendisk *disk;
    struct module *owner;
    int ret;
    int partno;       /*保存分区号，0 表示磁盘，1 表示第一个分区，用于确定是打开磁盘还是分区*/
    int perm = 0;

    if (mode & FMODE_READ)
        perm |= MAY_READ;
    if (mode & FMODE_WRITE)
        perm |= MAY_WRITE;

    if (!for_part) {          /*打开磁盘需要检查权限*/
        ret = devcgroup_inode_permission(bdev->bd_inode, perm);    /*访问权限检查*/
        ...     /*错误处理*/
    }

restart:

    ret = -ENXIO;

    /*从块设备驱动数据库查找 gendisk 实例，partno 保存了分区号，来自 block_device 实例*/
    disk = get_gendisk(bdev->bd_dev, &partno);    /*/block/genhd.c*/
    if (!disk)
        goto out;
    owner = disk->fops->owner;

    disk_block_events(disk);        /*/block/genhd.c*/
    mutex_lock_nested(&bdev->bd_mutex, for_part);

    /*下面处理块设备首次打开的情形*/
    if (!bdev->bd_openers) {        /*bdev->bd_openers 为 0，表示块设备未被打开*/
        bdev->bd_disk = disk;                    /*指向 gendisk 实例*/
```

```
    bdev->bd_queue = disk->queue;       /*指向 gendisk 请求队列*/
    bdev->bd_contains = bdev;           /*初始化指向自身 block_device 实例*/
    bdev->bd_inode->i_flags = disk->fops->direct_access ? S_DAX : 0;
    if (!partno) {                      /*partno 为 0，首次打开磁盘的情形*/
        ret = -ENXIO;
        bdev->bd_part = disk_get_part(disk, partno);        /*gendisk->part0*/
                                /*关联表示磁盘的分区 hd_struct 实例，/block/genhd.c*/
        ...     /*错误处理*/

        ret = 0;
        if (disk->fops->open) {         /*执行底层磁盘的打开（激活）操作，成功返回 0*/
            ret = disk->fops->open(bdev, mode);     /*block_device_operations.open()*/
            ...     /*错误处理*/
        }

        if (!ret)
            bd_set_size(bdev,(loff_t)get_capacity(disk)<<9);        /*扇区数转字节数*/
                                    /*设置磁盘容量至 block_device 实例，/fs/block_dev.c*/
        if (bdev->bd_invalidated) {         /*如果分区信息无效，读取磁盘中的分区信息*/
            if (!ret)
                rescan_partitions(disk, bdev);     /*扫描分区信息，/block/partition-generic.c*/
            else if (ret == -ENOMEDIUM)
                invalidate_partitions(disk, bdev);
        }
        if (ret)
            goto out_clear;
    }      /*首次打开磁盘处理结束*/

/*以下是处理首次打开分区的情形*/
else {          /*partno 不为 0，表示打开的是分区*/
    struct block_device *whole;
    whole = bdget_disk(disk, 0);   /*返回表示磁盘的 block_device 实例指针，/block/genhd.c*/
    ret = -ENOMEM;
    if (!whole)
        goto out_clear;
    BUG_ON(for_part);
    ret = __blkdev_get(whole, mode, 1);   /*先打开磁盘，递归调用*/
    if (ret)
        goto out_clear;
    bdev->bd_contains = whole;              /*指向表示磁盘的 block_device 实例*/
    bdev->bd_part = disk_get_part(disk, partno);        /*关联分区 hd_struct 实例*/
    if (!(disk->flags & GENHD_FL_UP) || !bdev->bd_part || !bdev->bd_part->nr_sects) {
        ret = -ENXIO;
```

```
                goto out_clear;
            }
            bd_set_size(bdev, (loff_t)bdev->bd_part->nr_sects << 9);     /*设置分区容量，字节数*/

            if ((bdev->bd_part->start_sect % (PAGE_SIZE / 512)) ||
                (bdev->bd_part->nr_sects % (PAGE_SIZE / 512)))
                bdev->bd_inode->i_flags &= ~S_DAX;
        }       /*首次打开分区处理结束*/
    }     /*if (!bdev->bd_openers)  首次打开块设备处理结束*/

    /*以下是处理块设备已打开的情形，bdev->bd_openers!=0*/
    else {
        if (bdev->bd_contains == bdev) {     /*磁盘已打开的情形*/
            ret = 0;
            if (bdev->bd_disk->fops->open)
                ret = bdev->bd_disk->fops->open(bdev, mode);   /*执行底层打开操作*/
            if (bdev->bd_invalidated) {          /*分区信息是否无效*/
                if (!ret)
                    rescan_partitions(bdev->bd_disk, bdev);       /*无效则扫描磁盘分区信息*/
                else if (ret == -ENOMEDIUM)
                    invalidate_partitions(bdev->bd_disk, bdev);
            }
            if (ret)
                goto out_unlock_bdev;
        }       /*磁盘已打开的情形处理结束*/
        /*分区已打开的情形不需要处理*/
        put_disk(disk);
        module_put(owner);
    }   /*块设备已打开的情形处理结束*/

    /*block_device 已关联 gendisk、hd_struct 和 request_queue 实例*/
    bdev->bd_openers++;       /*增加 block_device 实例打开次数*/
    if (for_part)
        bdev->bd_part_count++;
    mutex_unlock(&bdev->bd_mutex);
    disk_unblock_events(disk);
    return 0;        /*成功返回 0*/
    ...
}
```

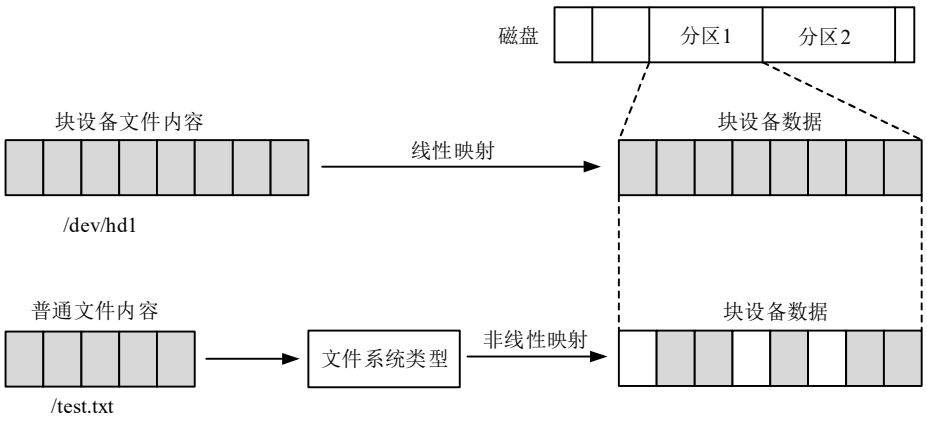
__blkdev_get()函数执行流程前面都介绍过了，请读者自行阅读以上源代码。如果是首次打开块设备时，block_device 实例先关联 gendisk 和 request_queue 实例，再打开磁盘扫描分区信息并添加 hd_struct 实例后，block_device 实例再关联 hd_struct 实例。

10.6.3 裸块设备操作

块设备在内核中都由一个块设备文件表示，对块设备的控制操作需通过块设备文件来进行，但是对块设备的读写操作有两个途径，如下图所示。

一是通过对块设备文件的读写来读写块设备。这时块设备文件的内容就是实际块设备上的数据，对块设备文件内容的读写可以线性地转换成对块设备中数据块的读写。

二是通过对块设备中保存文件的读写来读写块设备。块设备（分区）格式化成某种类型的文件系统后，可用来存储目录和文件，由文件系统类型确定文件内容保存在块设备中哪些数据块中，文件内容在块设备中是离散保存的（不在连续的数据块中）。对文件内容的读写将由文件系数类型代码转换成对块设备中离散数据块的访问，这时文件内容和块设备中数据块就不是线性映射的关系了。



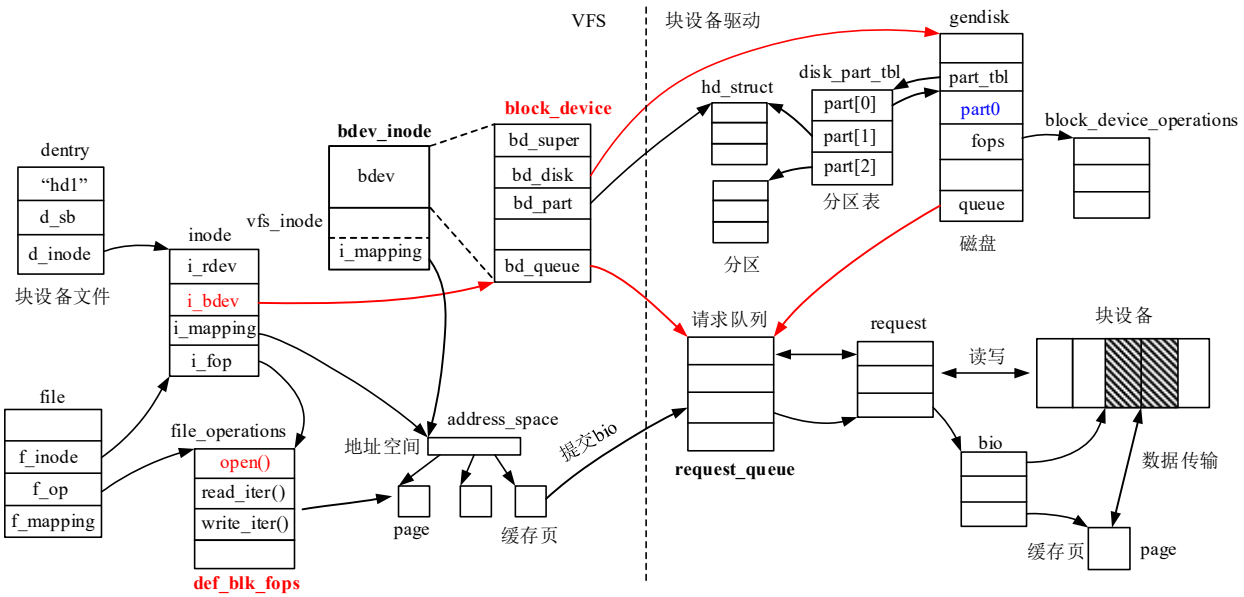
以上两种方式访问块设备最终都会转换成对块设备中数据块的访问，也就是说块设备驱动程序并不会区分这两种方式，到它这里就只是对某个或某些数据块的访问。

块设备（分区）中文件的读写操作到第 11 章再介绍，通过块设备文件访问块设备，我们称它为访问裸块设备，也就是只是将块设备视为连续的数据块，不识别其中的文件系统。

下面介绍对裸块设备的访问。

1 概述

块设备文件的访问流程如下图所示：



在 open()系统调用中打开块设备文件时，由于其为特殊文件，将调用 init_special_inode()对文件 inode 实例进行处理，inode->i_fop 赋予 **def_blk_fops** 实例指针（file_operations 实例），inode->i_rdev 赋予块设备号。

def_blk_fops 实例是块设备文件的操作接口，实例定义如下（/fs/block_dev.c）：

```
const struct file_operations    def_blk_fops = {
        .open           = blkdev_open,          /*打开块设备*/
        .release        = blkdev_close,         /*关闭块设备*/
        .llseek         = block_llseek,
        .read_iter      = blkdev_read_iter,     /*（裸）块设备读函数，/fs/block_dev.c*/
        .write_iter     = blkdev_write_iter,    /*（裸）块设备写操作，/fs/block_dev.c*/
        .mmap           = generic_file_mmap,    /*内存映射，通用函数*/
        .fsync          = blkdev_fsync,         /*同步块设备，/fs/block_dev.c(L340)*/
        .unlocked_ioctl     = block_ioctl,      /*发送控制命令，/block/ioctl.c*/
#ifdef CONFIG_COMPAT
        .compat_ioctl   = compat_blkdev_ioctl,  /*/block/compat_ioctl.c*/
#endif
        .splice_read    = generic_file_splice_read,
        .splice_write   = iter_file_splice_write,
};
```

def_blk_fops 实例中 open()函数 blkdev_open()将执行类似于上一小节介绍的 blkdev_get_by_path()函数执行的工作，为块设备创建 block_device 实例，并关联 gendisk、hd_struct 和 request_queue 实例。

def_blk_fops 实例中读写函数调用通用的读写文件函数对块设备进行读写操作。

def_blk_fops 实例中 unlocked_ioctl()函数 block_ioctl()用于对块设备进行控制操作。

2 打开块设备文件

虚拟文件系统层在打开块设备文件的 open()系统调用中，当检查到打开的是特殊文件时，将调用通用函数 init_special_inode()对文件 inode 进行处理，函数代码简列如下：

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;       /*字符设备文件操作*/
        inode->i_rdev = rdev;
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;       /*块设备文件通用文件操作结构实例*/
        inode->i_rdev = rdev;               /*设备号*/
    } else if (S_ISFIFO(mode))
        inode->i_fop = &pipefifo_fops;
    else if (S_ISSOCK(mode))
        ;
    else
        ...
}
```

open()系统调用随后将调用 inode->i_fop->open()函数完成块设备文件的打开操作。def_blk_fops 实例中 open()函数 **blkdev_open()**在/fs/block_dev.c 文件内实现，代码如下：

```
static int blkdev_open(struct inode * inode, struct file * filp)
/*inode：指向块设备文件 inode 实例，filp：指向块设备文件 file 实例*/
{
    struct block_device *bdev;
    filp->f_flags |= O_LARGEFILE;

    if (filp->f_flags & O_NDELAY)
        filp->f_mode |= FMODE_NDELAY;
    if (filp->f_flags & O_EXCL)              /*独占打开设备*/
        filp->f_mode |= FMODE_EXCL;
    if ((filp->f_flags & O_ACCMODE) == 3)
        filp->f_mode |= FMODE_WRITE_IOCTL;       /*文件标记设置*/

     bdev = bd_acquire(inode);                /*查找或创建 bdev_inode 实例，/fs/block_dev.c*/
     if (bdev == NULL)
        return -ENOMEM;
     filp->f_mapping = bdev->bd_inode->i_mapping;     /*指向 bdev_inode.vfs_inode 中地址空间*/
     return    blkdev_get(bdev, filp->f_mode, filp);       /*打开块设备，/fs/block_dev.c*/
            /*建立 block_device 实例与 gendisk、hd_struct 和 request_queue 实例之间关联*/
}
```

读者应该很容易理解 blkdev_open()函数了，bd_acquire(inode)和 blkdev_get(bdev,filp->f_mode,filp)函数前面都介绍过了，这里就不再重复了。

3 读写操作

进程对块设备文件的发起的 read()和 write()系统调用，将会调用 def_blk_fops 实例中定义的 read_iter()和 write_iter()函数对块设备进行读写。

def_blk_fops 实例中 **blkdev_read_iter()**和 **blkdev_write_iter()**读写函数，将调用通用的文件读写函数 generic_file_read_iter()和__generic_file_write_iter()（或 generic_write_sync()），执行读写操作。所谓的通用读写函数就是建立页缓存来缓存文件内容，用户读写是对页缓存进行的，由内核实现页缓存与块设备的同步，同步操作将访问封装成 bio 实例，提交到请求队列，以上通用函数到第 11 章再介绍。

下面简要列出 blkdev_read_iter()和 blkdev_write_iter()函数的实现，读者可以学习完第 11 章后，再回过头来看。读函数代码如下（/fs/block_dev.c）：

```
ssize_t blkdev_read_iter(struct kiocb *iocb, struct iov_iter *to)
{
    struct file *file = iocb->ki_filp;
    struct inode *bd_inode = file->f_mapping->host;
    loff_t size = i_size_read(bd_inode);
    loff_t pos = iocb->ki_pos;

    if (pos >= size)
        return 0;
```

```
        size -= pos;
        iov_iter_truncate(to, size);
        return generic_file_read_iter(iocb, to);          /*通用读文件函数*/
}

块设备文件写操作函数：
ssize_t blkdev_write_iter(struct kiocb *iocb, struct iov_iter *from)
{
        struct file *file = iocb->ki_filp;
        struct inode *bd_inode = file->f_mapping->host;
        loff_t size = i_size_read(bd_inode);
        struct blk_plug plug;
        ssize_t ret;

        if (bdev_read_only(I_BDEV(bd_inode)))        /*块设备只读，不能写*/
                return -EPERM;

        if (!iov_iter_count(from))
                return 0;

        if (iocb->ki_pos >= size)
                return -ENOSPC;

        iov_iter_truncate(from, size - iocb->ki_pos);

        blk_start_plug(&plug);
        ret = __generic_file_write_iter(iocb, from);          /*通用函数*/
        if (ret > 0) {
                ssize_t err;
                err = generic_write_sync(file, iocb->ki_pos - ret, ret);       /*通用函数*/
                if (err < 0)
                        ret = err;
        }
        blk_finish_plug(&plug);
        return ret;
}
```

上面通用的块设备读写访问函数，最终会把访问操作转换成对块设备中数据块的访问，由 bio 实例封装，并提交到请求队列，bio 实例由请求队列处理。通用读写函数到后面第 11 章再介绍。

4 块设备控制

对于块设备文件，ioctl()系统调用可用于向块设备发送控制命令，命令名称定义在/include/uapi/linux/fs.h

头文件，例如：

```
#define   BLKROSET     _IO(0x12,93)     /* set device read-only (0 = read-write) */
#define   BLKROGET     _IO(0x12,94)     /* get read-only status (0 = read_write) */
#define   BLKRRPART    _IO(0x12,95)     /* re-read partition table */
#define   BLKGETSIZE   _IO(0x12,96)     /* return device size /512 (long *arg) */
#define   BLKFLSBUF    _IO(0x12,97)     /* flush buffer cache */
#define   BLKRASET     _IO(0x12,98)     /* set read ahead for block device */
....
```

ioctl()系统调用最终调用 def_blk_fops 实例中的 unlocked_ioctl()函数 **block_ioctl()**处理块设备命令，函数定义如下（/fs/block_dev.c）：

```
static long block_ioctl(struct file *file, unsigned cmd, unsigned long arg)
{
    struct block_device *bdev = I_BDEV(file->f_mapping->host);      /*block_device 实例*/
    fmode_t mode = file->f_mode;

    if (file->f_flags & O_NDELAY)
        mode |= FMODE_NDELAY;
    else
        mode &= ~FMODE_NDELAY;

    return blkdev_ioctl(bdev, mode, cmd, arg);          /*/block/ioctl.c*/
}
```

blkdev_ioctl()函数定义在/block/ioctl.c 文件内，用于处理命令，代码简列如下：

```
int blkdev_ioctl(struct block_device *bdev, fmode_t mode, unsigned cmd,unsigned long arg)
{
    struct gendisk *disk = bdev->bd_disk;
    struct backing_dev_info *bdi;
    loff_t size;
    int ret, n;
    unsigned int max_sectors;

    switch(cmd) {
    case BLKFLSBUF:
        if (!capable(CAP_SYS_ADMIN))
            return -EACCES;

        ret = __blkdev_driver_ioctl(bdev, mode, cmd, arg);
        if (!is_unrecognized_ioctl(ret))
            return ret;

        fsync_bdev(bdev);
```

```
            invalidate_bdev(bdev);
            return 0;

        case BLKROSET:
            ret = __blkdev_driver_ioctl(bdev, mode, cmd, arg);
            if (!is_unrecognized_ioctl(ret))
                return ret;
            if (!capable(CAP_SYS_ADMIN))
                return -EACCES;
            if (get_user(n, (int __user *)(arg)))
                return -EFAULT;
            set_device_ro(bdev, n);
            return 0;

        ...         /*处理块设备通用命令*/
        default:
            ret = __blkdev_driver_ioctl(bdev, mode, cmd, arg);     /*处理特定于某一设备的命令*/
        }
        return ret;
    }
```

blkdev_ioctl()函数内将处理一些块设备通用的命令，而对于需要具体块设备驱动程序处理的命令由函数__blkdev_driver_ioctl()处理，此函数内调用 block_device_operations.ioctl()函数处理命令。

10.7 LOOP 块设备驱动

从本节开始将介绍几个具体块设备驱动程序的实现框架，先易后难。本节先介绍简单的模拟块设备驱动的实现，后面两节分别介绍 SD 卡和 MTD 设备驱动框架的实现。

10.7.1 概述

LOOP 设备是用一个文件（后备文件）来模拟块设备，文件的内容就是这个模拟块设备的内容。对 LOOP 设备的读写转化成对这个文件的读写，由此文件再转换成对真正块设备的读写。

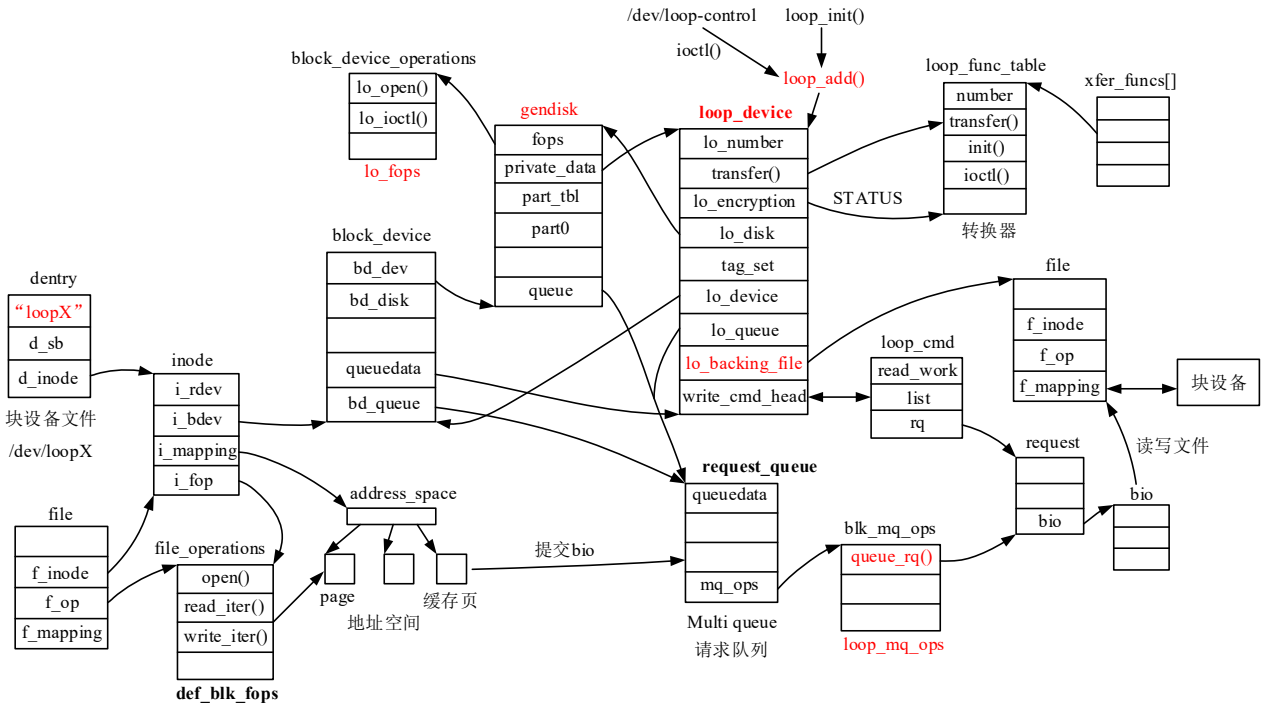
LOOP 设备的一个应用就是在虚拟机中，用主机一个文件来模拟一个磁盘，在虚拟机看来这是一个磁盘，在主机看来它其实只是个普通文件。

LOOP 设备驱动程序位于/drivers/block/loop.c 文件内。

1 框架

LOOP 设备驱动框架如下图所示，LOOP 设备由 loop_device 结构体表示。默认情况下系统中会有 8 个 LOOP 设备，设备名称为/dev/loop0~7，这个数量是由配置选项 BLK_DEV_LOOP_MIN_COUNT 确定的，用户可以修改此值。另外也可以通过模块参数 max_loop 或命令行参数"max_loop="确定，并且这具有比配置选项更高的优先权。

在初始化函数 loop_init()中将调用 loop_add()函数添加 loop_device 实例，并创建和添加对应的 gendisk 实例，每个 LOOP 设备对应一个 gendisk 实例。另外还可以对/dev/loop-control 设备文件执行 ioctl()系统调用，通过 LOOP_CTL_ADD/LOOP_CTL_REMOVE 命令来添加/删除 LOOP 设备。



由 loop_add()函数添加的 loop_device 实例是没有关联模拟块设备的文件的，用户需对 LOOP 设备文件，即/dev/loop0~7 执行 ioctl()系统调用，通过 LOOP_SET_FD/LOOP_CHANGE_FD 等命令，来设置 LOOP 设备关联的文件，这将由 lo_fops 实例中的 lo_ioctl()函数实现。loop_device 结构体 lo_backing_file 成员关联此文件的 file 实例（用户通过文件描述符设置）。

添加 loop_device 实例时，将为关联的 gendisk 创建请求队列，这是 Multi queue 请求队列，其关联的 blk_mq_ops 实例为 loop_mq_ops。对 LOOP 设备的访问封装成 bio 实例，将提交到 Multi queue 请求队列，依 bio 实例创建的请求，将由 loop_mq_ops 实例中的 queue_rq()函数插入硬件队列，此函数将为请求创建并添加 loop_cmd 实例。在工作队列的工作中将执行 loop_cmd 实例关联的请求，即将 bio 实例中的数据访问转换成对后备文件内容的访问，最终转换成对真实块设备的访问。

另外，在用户读写的 LOOP 设备数据与后备文件内容之间可以添加一个数据转换器，由 loop_func_table 结构体表示，其中的 transfer()函数用于实现数据的转换。对 LOOP 设备进行写操作时，用户数据先转换再写入后备文件，读数据时，先对后备文件内容中数据进行转换，再复制到用户空间。

2 数据结构

LOOP 设备驱动主要数据结构有 loop_device、loop_cmd 等，下面看一下它们的定义。

■loop_device

loop_device 结构体定义如下（/drivers/block/loop.h）：

```
struct loop_device {
    int         lo_number;          /*实例编号，由 idr 结构体管理，键值用于查找实例*/
    atomic_t    lo_refcnt;
    loff_t          lo_offset;
    loff_t          lo_sizelimit;
    int         lo_flags;               /*标记*/
    int         (*transfer)(struct loop_device *, int cmd,struct page *raw_page, unsigned raw_off,
```

```
                struct page *loop_page, unsigned loop_off,int size, sector_t real_block); /*数据转换函数*/
    char        lo_file_name[LO_NAME_SIZE];
    char        lo_crypt_name[LO_NAME_SIZE];
    char        lo_encrypt_key[LO_KEY_SIZE];
    int         lo_encrypt_key_size;
    struct loop_func_table *lo_encryption;          /*指向 loop_func_table 结构体，转换器*/
    __u32               lo_init[2];
    kuid_t          lo_key_owner;       /* Who set the key */
    int         (*ioctl)(struct loop_device *, int cmd, unsigned long arg);

    struct file *     lo_backing_file;          /*指向后备文件 file 实例*/
    struct block_device    *lo_device;        /*指向块设备 block_device 实例*/
    unsigned        lo_blocksize;
    void        *key_data;

    gfp_t           old_gfp_mask;

    spinlock_t          lo_lock;
    struct workqueue_struct *wq;            /*工作队列*/
    struct list_head        write_cmd_head;      /*双链表头，管理 loop_cmd 实例*/
    struct work_struct   write_work;          /*工作，执行函数为 loop_queue_write_work()*/
    bool            write_started;
    int             lo_state;
    struct mutex        lo_ctl_mutex;

    struct request_queue      *lo_queue;            /*指向请求队列*/
    struct blk_mq_tag_set    tag_set;          /*用于创建 Multi queue 请求队列的 blk_mq_tag_set 实例*/
    struct gendisk        *lo_disk;         /*指向 gendisk 实例*/
};
```

loop_device 结构体中主要成员在上面都有注释了，这里就不解释了。

■loop_cmd

loop_cmd 结构体表示 LOOP 设备命令，封装了读写请求，定义如下（/drivers/block/loop.h）：

```
struct loop_cmd {
    struct work_struct read_work;           /*工作，执行函数初始化为 loop_queue_read_work()*/
    struct request    *rq;              /*请求*/
    struct list_head    list;             /*双链表成员*/
};
```

10.7.2 添加 LOOP 设备

本小节介绍如何向内核添加 LOOP 设备。

1 初始化

LOOP 设备驱动程序初始化函数 loop_init()定义如下（/drivers/block/loop.c）：

```
static int __init loop_init(void)
{
    int i, nr;
    unsigned long range;
    struct loop_device *lo;
    int err;

    err = misc_register(&loop_misc);        /*注册 miscdevice 实例，设备文件/dev/loop-control*/
    ...

    part_shift = 0;
    if (max_part > 0) {              /*LOOP 设备最大分区数，默认为 0，可通过同名的模块参数设置*/
        part_shift = fls(max_part);
        max_part = (1UL << part_shift) - 1;
    }

    if ((1UL << part_shift) > DISK_MAX_PARTS) {
        err = -EINVAL;
        goto misc_out;
    }

    if (max_loop > 1UL << (MINORBITS - part_shift)) {
        err = -EINVAL;
        goto misc_out;
    }

    if (max_loop) {                  /*确定 LOOP 设备数量，max_loop 具有优先权*/
        nr = max_loop;
        range = max_loop << part_shift;
    } else {
        nr = CONFIG_BLK_DEV_LOOP_MIN_COUNT;
        range = 1UL << MINORBITS;
    }

    if (register_blkdev(LOOP_MAJOR, "loop")) {          /*注册块设备主设备号*/
        ...
    }

    blk_register_region(MKDEV(LOOP_MAJOR, 0), range,
                THIS_MODULE, loop_probe, NULL, NULL); /*向块设备驱动数据库添加 probe 实例*/
```

```
    mutex_lock(&loop_index_mutex);
    for (i = 0; i < nr; i++)
        loop_add(&lo, i);
    mutex_unlock(&loop_index_mutex);

    printk(KERN_INFO "loop: module loaded\n");
    return 0;
    ...
}
```

loop_init()函数中调用 misc_register(&loop_misc)函数注册了 misc 设备 miscdevice 实例 loop_misc，设备文件/dev/loop-control，通过对此文件执行 ioctl()系统调用可添加/删除 loop_device 实例，随后注册了 LOOP 设备的主设备号。

loop_init()函数最后调用 loop_add()函数添加了默认的 loop_device 实例，数量为 nr，下面将介绍此函数的实现。

2 添加函数

loop_add()函数定义如下，用于添加 loop_device 实例（/drivers/block/loop.c）：

```
static int loop_add(struct loop_device **l, int i)
/*i：loop_device 实例的编号*/
{
    struct loop_device *lo;
    struct gendisk *disk;
    int err;

    err = -ENOMEM;
    lo = kzalloc(sizeof(*lo), GFP_KERNEL);          /*分配 loop_device 实例*/
    ...

    lo->lo_state = Lo_unbound;              /*初始状态*/

    /*分配 id*/
    if (i >= 0) {
        err = idr_alloc(&loop_index_idr, lo, i, i + 1, GFP_KERNEL);
        if (err == -ENOSPC)
            err = -EEXIST;
    } else {
        err = idr_alloc(&loop_index_idr, lo, 0, 0, GFP_KERNEL);
    }
    ...
    i = err;

    err = -ENOMEM;
    /*设置 blk_mq_tag_set 结构体成员，用于创建 Multi queue 请求队列*/
```

```
    lo->tag_set.ops = &loop_mq_ops;
    lo->tag_set.nr_hw_queues = 1;        /*硬件队列数量*/
    lo->tag_set.queue_depth = 128;       /*深度*/
    lo->tag_set.numa_node = NUMA_NO_NODE;
    lo->tag_set.cmd_size = sizeof(struct loop_cmd);          /*请求实例后附 loop_cmd 结构体实例*/
    lo->tag_set.flags = BLK_MQ_F_SHOULD_MERGE | BLK_MQ_F_SG_MERGE;
    lo->tag_set.driver_data = lo;

    err = blk_mq_alloc_tag_set(&lo->tag_set);        /*分配硬件队列标签*/
    ...

    lo->lo_queue = blk_mq_init_queue(&lo->tag_set);      /*创建 Multi queue 请求队列*/
    ...
    lo->lo_queue->queuedata = lo;          /*指向 loop_device 实例*/

    INIT_LIST_HEAD(&lo->write_cmd_head);
    INIT_WORK(&lo->write_work, loop_queue_write_work);       /*工作执行函数*/

    disk = lo->lo_disk = alloc_disk(1 << part_shift);          /*分配 gendisk 实例*/
    ...

    if (!part_shift)       /*part_shift 为 0*/
        disk->flags |= GENHD_FL_NO_PART_SCAN;       /*不扫描分区*/
    disk->flags |= GENHD_FL_EXT_DEVT;
    mutex_init(&lo->lo_ctl_mutex);
    atomic_set(&lo->lo_refcnt, 0);
    lo->lo_number           = i;
    spin_lock_init(&lo->lo_lock);
    disk->major             = LOOP_MAJOR;
    disk->first_minor       = i << part_shift;
    disk->fops              = &lo_fops;          /*block_device_operations 实例*/
    disk->private_data      = lo;                /*指向 loop_device 实例*/
    disk->queue             = lo->lo_queue;          /*指向请求队列*/
    sprintf(disk->disk_name, "loop%d", i);      /*磁盘设备文件名称/dev/loopX*/
    add_disk(disk);                 /*添加磁盘*/
    *l = lo;
    return lo->lo_number;       /*返回 loop_device 实例编号*/
    ...
}
```

如果读者对前面几节介绍的块设备驱动框架比较熟悉了的话，理解以上代码应该不难，这里就不解释了。

这里需要注意的一点是 loop_device 实例中 write_work 工作成员设置的 loop_queue_write_work()执行函数，它用于处理 write_cmd_head 双链表中命令，后面将介绍。

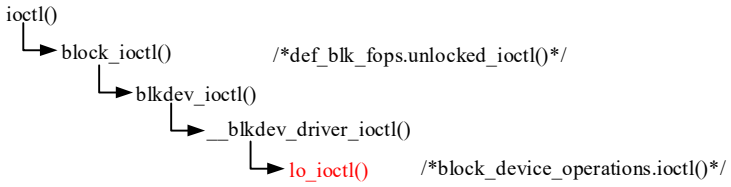
另外，用户还可以通过/dev/loop-control 设备文件的 ioctl()系统调用添加/删除 loop_device 实例，请读者自行阅读相关代码。

10.7.3 LOOP 设备操作

用户通过/dev/loop0~7 设备文件操作 LOOP 设备，操作前需要对文件执行 ioctl()系统调用，通过命令 LOOP_SET_FD 设置后备文件，然后就可以对 LOOP 设备执行其它操作了。

1 设备控制

用户通过 ioctl()系统调用设置 LOOP 设备后备文件的函数调用关系如下图所示：



LOOP 设备专有命令由 block_device_operations 结构体实例 lo_fops 中的 ioctl()函数，即 lo_ioctl()函数处理，如下所示（/drivers/block/loop.c）：

```
static int lo_ioctl(struct block_device *bdev, fmode_t mode,unsigned int cmd, unsigned long arg)
{
    struct loop_device *lo = bdev->bd_disk->private_data;        /*loop_device 实例*/
    int err;

    mutex_lock_nested(&lo->lo_ctl_mutex, 1);
    switch (cmd) {          /*命令，/include/uapi/linux/loop.h*/
    case LOOP_SET_FD:               /*设置后备文件*/
        err = loop_set_fd(lo, mode, bdev, arg);
        break;
    case LOOP_CHANGE_FD:        /*修改后备文件*/
        err = loop_change_fd(lo, bdev, arg);
        break;
    case LOOP_CLR_FD:               /*清除后备文件*/
        /* loop_clr_fd would have unlocked lo_ctl_mutex on success */
        err = loop_clr_fd(lo);
        if (!err)
            goto out_unlocked;
        break;
    case LOOP_SET_STATUS:   /*状态，由 loop_info 结构体表示（老版本），/include/uapi/linux/loop.h*/
        err = -EPERM;
        if ((mode & FMODE_WRITE) || capable(CAP_SYS_ADMIN))
            err = loop_set_status_old(lo,(struct loop_info __user *)arg);
        break;
    case LOOP_GET_STATUS:
        err = loop_get_status_old(lo, (struct loop_info __user *) arg);
```

```
            break;
        case LOOP_SET_STATUS64:   /*设置状态，loop_info64 结构体定义在/include/uapi/linux/loop.h*/
            err = -EPERM;
            if ((mode & FMODE_WRITE) || capable(CAP_SYS_ADMIN))
                err = loop_set_status64(lo,(struct loop_info64 __user *) arg);
            break;
        case LOOP_GET_STATUS64:
            err = loop_get_status64(lo, (struct loop_info64 __user *) arg);
            break;
        case LOOP_SET_CAPACITY:              /*设置能力*/
            err = -EPERM;
            if ((mode & FMODE_WRITE) || capable(CAP_SYS_ADMIN))
                err = loop_set_capacity(lo, bdev);
            break;
        default:
            err = lo->ioctl ? lo->ioctl(lo, cmd, arg) : -EINVAL;       /*loop_device 实例中函数*/
        }
        mutex_unlock(&lo->lo_ctl_mutex);

    out_unlocked:
        return err;
    }
```

LOOP 设备文件命令定义在/include/uapi/linux/loop.h 头文件，包括设置、修改后备文件，设置、获取设备状态等。下面以设置后备文件的 LOOP_SET_FD 命令为例，说明其执行结果。

■设置后备文件

LOOP_SET_FD 命令执行函数 loop_set_fd()定义如下：

```
static int loop_set_fd(struct loop_device *lo, fmode_t mode,struct block_device *bdev, unsigned int arg)
/*arg：后备文件描述符*/
{
    struct file *file, *f;
    struct inode     *inode;
    struct address_space *mapping;
    unsigned lo_blocksize;
    int          lo_flags = 0;
    int          error;
    loff_t           size;

    __module_get(THIS_MODULE);

    error = -EBADF;
    file = fget(arg);          /*后备文件 file 实例*/
```

```
if (!file)
        goto out;

error = -EBUSY;
if (lo->lo_state != Lo_unbound)
        goto out_putf;

f = file;
while (is_loop_device(f)) {          /*file 是 LOOP 设备文件的情形*/
        struct loop_device *l;

        if (f->f_mapping->host->i_bdev == bdev)
                goto out_putf;

        l = f->f_mapping->host->i_bdev->bd_disk->private_data;
        if (l->lo_state == Lo_unbound) {
                error = -EINVAL;
                goto out_putf;
        }
        f = l->lo_backing_file;
}

mapping = file->f_mapping;
inode = mapping->host;

error = -EINVAL;
if (!S_ISREG(inode->i_mode) && !S_ISBLK(inode->i_mode))
        goto out_putf;

if (!(file->f_mode & FMODE_WRITE) || !(mode & FMODE_WRITE) ||
        !file->f_op->write_iter)
        lo_flags |= LO_FLAGS_READ_ONLY;

lo_blocksize = S_ISBLK(inode->i_mode) ?inode->i_bdev->bd_block_size : PAGE_SIZE;

error = -EFBIG;
size = get_loop_size(lo, file);              /*LOOP 设备容量大小*/
...
lo->wq = alloc_workqueue("kloopd%d",
                WQ_MEM_RECLAIM | WQ_HIGHPRI | WQ_UNBOUND, 16,lo->lo_number);
                                                        /*创建工作队列*/
...
```

```
        error = 0;

        set_device_ro(bdev, (lo_flags & LO_FLAGS_READ_ONLY) != 0);
        /*设置 loop_device 实例*/
        lo->lo_blocksize = lo_blocksize;
        lo->lo_device = bdev;
        lo->lo_flags = lo_flags;
        lo->lo_backing_file = file;
        lo->transfer = NULL;
        lo->ioctl = NULL;
        lo->lo_sizelimit = 0;
        lo->old_gfp_mask = mapping_gfp_mask(mapping);
        mapping_set_gfp_mask(mapping, lo->old_gfp_mask & ~(__GFP_IO|__GFP_FS));

        if (!(lo_flags & LO_FLAGS_READ_ONLY) && file->f_op->fsync)
            blk_queue_flush(lo->lo_queue, REQ_FLUSH);

        set_capacity(lo->lo_disk, size);          /*设置磁盘容量*/
        bd_set_size(bdev, size << 9);
        loop_sysfs_init(lo);              /*LOOP 设备导出到 sysfs*/
        /* let user-space know about the new size */
        kobject_uevent(&disk_to_dev(bdev->bd_disk)->kobj, KOBJ_CHANGE);

        set_blocksize(bdev, lo_blocksize);

        lo->lo_state = Lo_bound;
        if (part_shift)
            lo->lo_flags |= LO_FLAGS_PARTSCAN;
        if (lo->lo_flags & LO_FLAGS_PARTSCAN)
            loop_reread_partitions(lo, bdev);          /*扫描磁盘分区，添加分区*/

        bdgrab(bdev);
        return 0;
        ...
    }
```

loop_set_fd()函数不难理解，就是执行一些例行性的工作，请读者自行阅读源代码。

2 读写访问

最后来看一下 LOOP 设备的读写操作如何进行。首先要明白一点，LOOP 设备从用户看来就是普通的块设备，只不过在读写数据时多了一层转换才到真正的块设备，而这层转换是由 LOOP 设备驱动程序实现的，用户可以不用关心。

LOOP 设备驱动程序中使用的是 Multi queue 请求队列，依 VFS 层提交的 bio 实例创建的请求，最终由 blk_mq_ops 实例的 **loop_mq_ops** 中的 queue_rq()函数插入到硬件队列，并进行处理。

loop_mq_ops 实例定义如下：

```
static struct blk_mq_ops loop_mq_ops = {
    .queue_rq       = loop_queue_rq,        /*入队（处理）请求函数*/
    .map_queue      = blk_mq_map_queue,
    .init_request   = loop_init_request,    /*初始化队列标签中请求实例，后附 loop_cmd 实例*/
};
```

请求入队（处理）函数 loop_queue_rq()定义如下：

```
static int loop_queue_rq(struct blk_mq_hw_ctx *hctx,const struct blk_mq_queue_data *bd)
{
    struct loop_cmd *cmd = blk_mq_rq_to_pdu(bd->rq);
    struct loop_device *lo = cmd->rq->q->queuedata;

    blk_mq_start_request(bd->rq);

    if (lo->lo_state != Lo_bound)
        return -EIO;

    if (cmd->rq->cmd_flags & REQ_WRITE) {           /*写请求*/
        struct loop_device *lo = cmd->rq->q->queuedata;
        bool need_sched = true;

        spin_lock_irq(&lo->lo_lock);
        if (lo->write_started)
            need_sched = false;
        else
            lo->write_started = true;
        list_add_tail(&cmd->list, &lo->write_cmd_head);     /*loop_cmd 实例添加到双链表末尾*/
        spin_unlock_irq(&lo->lo_lock);

        if (need_sched)
            queue_work(lo->wq, &lo->write_work);    /*触发工作*/
    } else {
        queue_work(lo->wq, &cmd->read_work);        /*触发读工作*/
    }

    return BLK_MQ_RQ_QUEUE_OK;
}
```

对于写请求就是将关联 loop_cmd 实例添加到 lo->write_cmd_head 双链表末尾，并触发 lo->write_work 工作（如果需要的话），工作执行函数为 loop_queue_write_work()（添加 LOOP 设备时设置）。

对于读请求直接触发 cmd->read_work 工作，工作执行函数为 loop_queue_read_work()（在初始化请求时设置）。读写请求的处理最后会由同一个函数来进行，因此下面以写请求的处理为例，说明其流程。

■处理写请求

处理写请求的 loop_queue_write_work()函数定义如下：

```
static void loop_queue_write_work(struct work_struct *work)
{
    struct loop_device *lo =container_of(work, struct loop_device, write_work);
    LIST_HEAD(cmd_list);          /*临时双链表*/

    spin_lock_irq(&lo->lo_lock);
 repeat:
    list_splice_init(&lo->write_cmd_head, &cmd_list);       /*写请求（命令）移入临时双链表*/
    spin_unlock_irq(&lo->lo_lock);

    while (!list_empty(&cmd_list)) {          /*遍历临时双链表，处理命令*/
        struct loop_cmd *cmd = list_first_entry(&cmd_list,struct loop_cmd, list);
        list_del_init(&cmd->list);
        loop_handle_cmd(cmd);       /*处理命令，读请求命令也由这个函数处理*/
    }

    spin_lock_irq(&lo->lo_lock);
    if (!list_empty(&lo->write_cmd_head))
        goto repeat;
    lo->write_started = false;
    spin_unlock_irq(&lo->lo_lock);
}
```

loop_queue_write_work()函数调用 loop_handle_cmd(cmd)函数依次处理各写请求命令，函数定义如下：

```
static void loop_handle_cmd(struct loop_cmd *cmd)
{
    const bool write = cmd->rq->cmd_flags & REQ_WRITE;
    struct loop_device *lo = cmd->rq->q->queuedata;
    int ret = -EIO;

    if (write && (lo->lo_flags & LO_FLAGS_READ_ONLY))
        goto failed;

    ret = do_req_filebacked(lo, cmd->rq);       /*读写后备文件*/

 failed:
    if (ret)
        cmd->rq->errors = -EIO;
    blk_mq_complete_request(cmd->rq);          /*结束请求*/
}
```

do_req_filebacked()函数用于将请求中的读写操作转换成对后备文件的读写操作，定义如下：

```
static int do_req_filebacked(struct loop_device *lo, struct request *rq)
{
    loff_t pos;
    int ret;

    pos = ((loff_t) blk_rq_pos(rq) << 9) + lo->lo_offset;

    if (rq->cmd_flags & REQ_WRITE) {            /*写*/
        if (rq->cmd_flags & REQ_FLUSH)          /*FLUSH 请求*/
            ret = lo_req_flush(lo, rq);
        else if (rq->cmd_flags & REQ_DISCARD)
            ret = lo_discard(lo, rq, pos);
        else if (lo->transfer)
            ret = lo_write_transfer(lo, rq, pos);       /*设置了转换器，先转换再写入*/
        else
            ret = lo_write_simple(lo, rq, pos);         /*直接写后备文件*/

    } else {                        /*读*/
        if (lo->transfer)
            ret = lo_read_transfer(lo, rq, pos);      /*设置了转换器，先转换再读取*/
        else
            ret = lo_read_simple(lo, rq, pos);      /*直接读后备文件*/
    }

    return ret;
}
```

对后备文件的读写可能还需要经过一个转换器（如同网络中的防火墙），如果 LOOP 设备设置的转换器，则先由转换器处理后，再执行读写操作。这个转换器位于用户内存（读写的 LOOP 设备数据）与后备文件内容（页缓存）之间，下面将做简要介绍。

■转换器

数据转换器由 loop_func_table 结构体表示（/drivers/block/loop.h）：

```
struct loop_func_table {
    int number;   /*转换器类型（编号）*/
    int (*transfer)(struct loop_device *lo, int cmd,
        struct page *raw_page, unsigned raw_off,
        struct page *loop_page, unsigned loop_off,int size, sector_t real_block);          /*转换函数*/
    int (*init)(struct loop_device *, const struct loop_info64 *);
    int (*release)(struct loop_device *);
    int (*ioctl)(struct loop_device *, int cmd, unsigned long arg);
    struct module *owner;
```

};

内核定义了 loop_func_table 结构体指针数组 xfer_funcs[]，数组最大项数及索引值标识定义如下：

```
/*转换器类型（编号），xfer_funcs[]数组项索引值，/include/uapi/linux/loop.h*/
#define   LO_CRYPT_NONE          0
#define   LO_CRYPT_XOR           1
#define   LO_CRYPT_DES           2
#define   LO_CRYPT_FISH2         3      /* Twofish encryption */
#define   LO_CRYPT_BLOW          4
#define   LO_CRYPT_CAST128       5
#define   LO_CRYPT_IDEA          6
#define   LO_CRYPT_DUMMY         9
#define   LO_CRYPT_SKIPJACK      10
#define   LO_CRYPT_CRYPTOAPI     18
#define   MAX_LO_CRYPT           20
```

用户可以在模块代码中调用以下函数注册/删除转换器：

```
int loop_register_transfer(struct loop_func_table *funcs);        /*关联到 xfer_funcs[]数组项*/
int loop_unregister_transfer(int number);
```

用户可对 LOOP 设备文件通过 ioctl()系统调用的 LOOP_SET_STATUS64/LOOP_GET_STATUS 命令设置/清除 LOOP 设备关联的转换器。

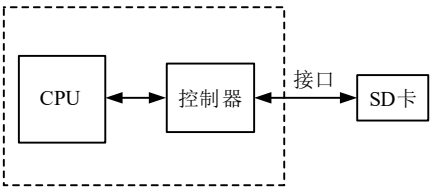
设置转换器时，转换器实例将赋予 loop_device 实例 lo_encryption 成员，转换函数 transfer()也将赋予 loop_device 实例 transfer()函数指针成员。在前面介绍的读写函数中将先调用转换函数 transfer()处理数据，再执行对后备文件内容的读写操作，请读者自行阅读相关源代码。

10.8 SD 卡驱动

SD/SDIO/MMC 卡（统称 MMC 设备）是嵌入式系统中典型的块设备，这三种存储卡的驱动程序统一由 MMC 框架实现，实现代码位于/drivers/mmc/目录下。SD 卡是目前常用的存储卡、SDIO 卡是 SD 卡的升级版本，MMC 卡是老式的多媒体存储卡。本节以常用的 SD 卡为例介绍其驱动程序的实现。

10.8.1 驱动框架

SD 卡是标准的存储设备，SD 卡协会规定了 SD 卡的标准和通讯协议。SD 卡具有两种接口模式，一种是标准的 SD 卡接口模式，另一种是 SPI 总线接口模式。SD 卡需要经过主机控制器与 CPU 相连，主机控制器对外实现 SD 卡的接口标准，对内连接在 CPU 地址总线上或板级总线上，如下图所示。主机控制器通过发送命令与 SD 卡进行数据的传输和控制，SD 卡协议规定了命令的格式和功能，这里就不详细介绍了，请读者查阅 SD 卡协会的技术手册。



MMC 驱动框架实现了 SD 卡的通信协议，具体驱动程序无需关注，具体设备驱动只需要实现控制器

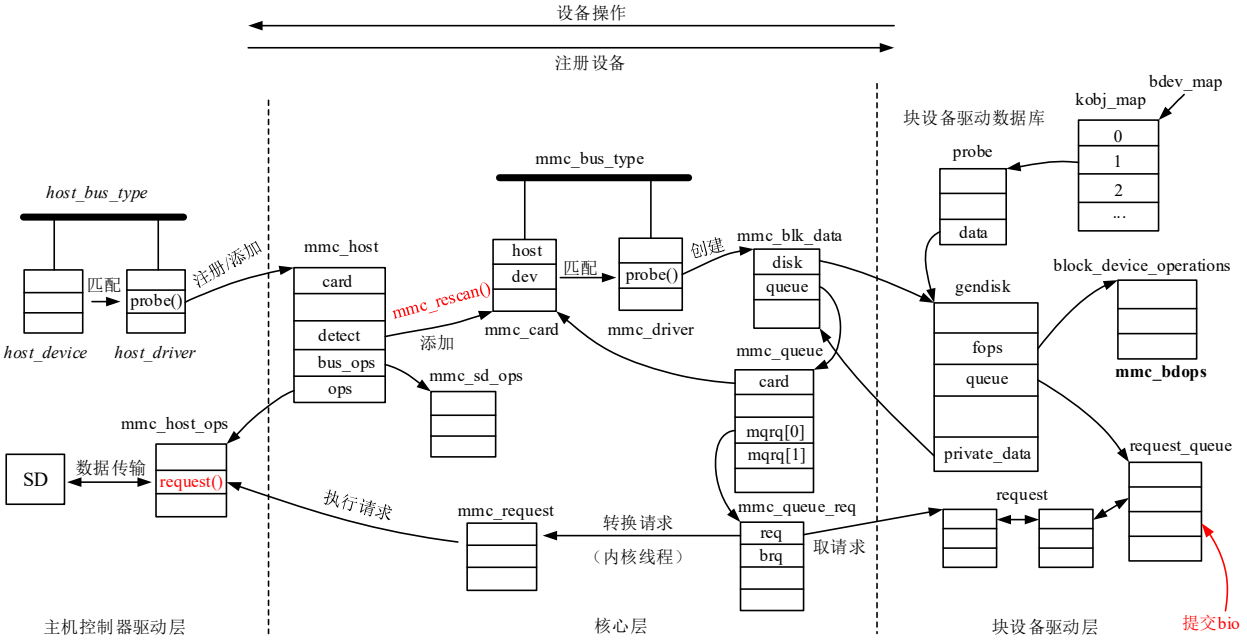
命令的产生和数据的收发。也就是说驱动框架实现了操作流程，具体驱动只需实现具体的步骤，驱动框架会组织正确的步骤以实现 SD 卡的正常工作。

MMC 驱动在/drivers/mmc/目录下，目录下包含三个文件夹，card 目录下文件实现 SD 卡块设备驱动层代码，core 目录下文件实现 MMC 设备驱动核心层代码（主要是实现 MMC 设备通信协议，工作流程），host 目录下的文件实现主机控制器驱动层代码，主要实现具体的命令和数据传输。

MMC 驱动框架如下图所示。主机控制器作为一个设备具有属于它的驱动程序，主机控制器挂接在 CPU 总线或板级总线上。向内核注册主机控制器设备时，将会调用主机控制器驱动的探测函数（probe()函数），探测函数在内核创建和注册 mmc_host 结构体实例，表示主机控制器。mmc_host 结构体最主要的成员是 mmc_host_ops 结构体指针和 detect 延时工作。**mmc_host_ops** 是一个由具体驱动程序实现的结构体，实现控制器对 SD 卡命令的产生和处理，mmc_host_ops 结构体的实现是具体控制器驱动的主要工作。detect 延时工作中调用 mmc_rescan()函数执行 SD 卡的探测和激活，并创建表示 SD 卡设备的 mmc_card 实例并向 MMC 驱动核心层注册。

MMC 驱动框架定义了 mmc_bus_type 总线实例，用于关联 SD 卡设备（mmc_card）和驱动（mmc_driver）。前面提到的 mmc_rescan()函数用于创建和注册 mmc_card 实例时，会将其添加到 mmc_bus_type 总线并匹配 mmc_driver，匹配成功后将调用 mmc_driver 中的 probe()函数。

MMC 设备驱动的 probe()函数将会在核心层创建表示 MMC 卡信息的 mmc_blk_data 结构体实例（含 gendisk 和请求队列等），在创建该实例时完成块设备驱动层 gendisk 结构体实列的创建和添加。添加 gendisk 实例前将会对其赋值块设备操作结构实例和请求队列。



在创建 mmc_blk_data 实例时，还将创建一个内核线程，用于处理队列请求。内核线程循环从标准请求队列中取请求，将其转换成 MMC 请求（mmc_request），最终调用主机控制器驱动定义的 mmc_host_ops 实例中的 **request()**函数处理 MMC 请求。MMC 请求中包含需要执行的命令、传输的数据等信息。标准请求队列的中的处理请求函数就是唤醒这个内核线程。

MMC 驱动框架中主机控制器驱动层需要具体板级相关驱动程序实现，这也是驱动移植的主要工作。核心层和块设备驱动层代码由内核实现，是公用的代码，移植时无需修改。

MMC 设备的注册从控制器的注册开始，逐级向核心层和块设备驱动层进行，而 VFS 层对 MMC 设备的访问从块设备驱动层开始，经过核心层，最后调用主机控制器驱动层的操作函数（产生命令）实现数据的传输。

10.8.2 主机控制器驱动层

MMC 设备主机控制器由 mmc_host 结构体表示，核心层代码提供了分配和 mmc_host 实例的接口函数。控制器操作结构 **mmc_host_ops** 包含控制器数据传输的实现函数（命令的实现），是具体设备驱动需要实现的最重要的数据结构。

MMC 设备主机控制器驱动程序的主要流程如下：

（1）实现 mmc_host_ops 结构体实例。

（2）调用核心层接口函数 mmc_alloc_host()分配 mmc_host 结构体实例并初始化，赋予 mmc_host_ops 结构体实例等。

（3）调用接口函数 mmc_add_host()向核心层添加 mmc_host 实例。在添加 mmc_host 实例后，其延时工作将会进入核心层，执行函数完成 SD 卡设备 mmc_card 结构体实例的创建和向 MMC 总线的添加。

1 数据结构

mmc_host 结构体定义在/include/linux/mmc/host.h 头文件内，表示 MMC 设备主机控制器，结构体成员简列如下：

```
struct mmc_host {
    struct device       *parent;        /*父设备*/
    struct device       class_dev;      /*表示控制器的 device 实例*/
    int                 index;          /*实例在内核中编号*/
    const struct mmc_host_ops *ops;     /*控制器操作结构指针*/
    ...
    /* host specific block data，块设备特有数据*/
    unsigned int        max_seg_size;  /* see blk_queue_max_segment_size */
    unsigned short      max_segs;      /* see blk_queue_max_segments */
    unsigned short      unused;
    unsigned int        max_req_size;      /* maximum number of bytes in one req */
    unsigned int        max_blk_size;      /* maximum size of one mmc block */
    unsigned int        max_blk_count;     /* maximum number of blocks in one req */
    unsigned int        max_busy_timeout; /* max busy timeout in ms */
    ...
    struct mmc_ios          ios;        /* current io bus settings */
    struct mmc_card     *card;          /*控制器连续的 MMC 设备*/

    wait_queue_head_t   wq;             /*等待队列头*/
    struct task_struct  *claimer;       /*持有控制器的进程*/
    int                 claim_cnt;       /* "claim" nesting count */

    struct delayed_work detect;     /*延时工作，完成 MMC 设备的探测和激活*/
    int    detect_change;       /* card detect flag */
    struct mmc_slot     slot;

    const struct mmc_bus_ops *bus_ops;     /*物理总线操作结构，/drivers/mmc/core/core.h*/
    unsigned int        bus_refs;       /* reference counter */
```

```
    ...
    struct mmc_async_req    *areq;          /* active async req */
    struct mmc_context_info     context_info; /* async synchronization info */
    ...
    unsigned long       private[0] ____cacheline_aligned;
};
```

mmc_host 结构体主要成员简介如下：

●**ops**：mmc_host_ops 结构体指针成员，mmc_host_ops 结构体包含控制器实现与 MMC 设备数据传输、设备控制等的操作函数。mmc_host_ops 结构体定义如下（/include/linux/mmc/host.h）：

```
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,bool is_first_req);
    void (*request)(struct mmc_host *host, struct mmc_request *req);      /*处理 MMC 请求的函数*/
    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);
    void (*init_card)(struct mmc_host *host, struct mmc_card *card);      /*初始化 MMC 设备*/
    int   (*start_signal_voltage_switch)(struct mmc_host *host, struct mmc_ios *ios);
    int   (*card_busy)(struct mmc_host *host);
    int   (*execute_tuning)(struct mmc_host *host, u32 opcode);
    int   (*prepare_hs400_tuning)(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);
    void (*hw_reset)(struct mmc_host *host);
    void (*card_event)(struct mmc_host *host);          /*有卡事件发生*/
    int   (*multi_io_quirk)(struct mmc_card *card, unsigned int direction, int blk_size);
};
```

MMC 设备块设备驱动层的请求处理函数（实际为内核线程），从块设备请求队列中取请求，然后将通用的 request 实例转换成 MMC 设备的请求 mmc_request 实例，交由 mmc_host_ops 实例的 **request()**函数处理，函数内以命令的形式完成数据的传输。mmc_request 结构体定义在/include/inux/mmc/core.h 头文件：

```
struct mmc_request {
    struct mmc_command    *sbc;          /*MMC 命令，/include/inux/mmc/core.h*/
    struct mmc_command    *cmd;
    struct mmc_data       *data;         /*命令数据，/include/inux/mmc/core.h*/
    struct mmc_command    *stop;

    struct completion     completion;    /*完成量*/
    void                  (*done)(struct mmc_request *);    /*命令完成的回调函数*/
    struct mmc_host       *host;         /*主机控制器*/
};
```

MMC 命令及数据的表示形式，有兴趣的读者可自行查阅源代码，这里不涉及过多的细节。

●**bus_ops**：指向 mmc_bus_ops 结构体，表示物理总线操作结构，如下所示（/drivers/mmc/core/core.h）：

```
struct mmc_bus_ops {
    void (*remove)(struct mmc_host *);
    void (*detect)(struct mmc_host *);
    int (*pre_suspend)(struct mmc_host *);
    int (*suspend)(struct mmc_host *);
    int (*resume)(struct mmc_host *);
    int (*runtime_suspend)(struct mmc_host *);
    int (*runtime_resume)(struct mmc_host *);
    int (*power_save)(struct mmc_host *);
    int (*power_restore)(struct mmc_host *);
    int (*alive)(struct mmc_host *);
    int (*shutdown)(struct mmc_host *);
    int (*reset)(struct mmc_host *);
};
```

前面介绍的 mmc_host_ops 结构体侧重于对主机控制器连接设备（卡）的操作，而 mmc_bus_ops 侧重于对控制器自身的操作和控制。

●**card**：指向表示 MMC 设备（卡）的 mmc_card 结构体实例，实例在注册 mmc_host 实例探测 MMC 设备时创建。

●**detect**：延时工作 delayed_work 实例，执行函数初始为 mmc_rescan()，负责探测并激活 MMC 设备。

●**context_info**：mmc_context_info 结构体实例，定义如下（/include/linux/mmc/host.h）：

```
struct mmc_context_info {
    bool            is_done_rcv;
    bool            is_new_req;
    bool            is_waiting_last_req;
    wait_queue_head_t   wait;
    spinlock_t          lock;
};
```

2 分配控制器

MMC 驱动框架核心层提供了分配和添加 mmc_host 结构体实例的接口函数。

mmc_alloc_host()函数用于分配主机控制器结构体实例，代码简列如下（/drivers/mmc/core/host.c）：

```
struct mmc_host *mmc_alloc_host(int extra, struct device *dev)
/*extra：私有数据结构长度，紧接在 mmc_host 实例之后，dev：控制器父设备 device 实例*/
{
    int err;
    struct mmc_host *host;

    host = kzalloc(sizeof(struct mmc_host) + extra, GFP_KERNEL);
                                    /*分配结构体实例，附加驱动程序私有数据*/
    if (!host)
```

```
        return NULL;
    host->rescan_disable = 1;
    idr_preload(GFP_KERNEL);
    spin_lock(&mmc_host_lock);
    err = idr_alloc(&mmc_host_idr, host, 0, 0, GFP_NOWAIT);     /*分配 idr 编号*/
    if (err >= 0)
        host->index = err;
    spin_unlock(&mmc_host_lock);
    idr_preload_end();
    if (err < 0) {
        kfree(host);
        return NULL;
    }

    dev_set_name(&host->class_dev, "mmc%d", host->index);       /*表示控制器的设备名称*/

    host->parent = dev;
    host->class_dev.parent = dev;
    host->class_dev.class = &mmc_host_class;          /*设备类*/
    device_initialize(&host->class_dev);          /*初始化表示控制器的 device 实例*/

    if (mmc_gpio_alloc(host)) {
        put_device(&host->class_dev);
        return NULL;
    }

    mmc_host_clk_init(host);
            /*时钟初始化，没有配置 MMC_CLKGATE 选项为空操作，/drivers/mmc/core/host.c*/
    spin_lock_init(&host->lock);
    init_waitqueue_head(&host->wq);
    INIT_DELAYED_WORK(&host->detect, mmc_rescan);     /*初始化延时工作，注意执行函数*/
#ifdef CONFIG_PM
    host->pm_notify.notifier_call = mmc_pm_notify;
#endif
    setup_timer(&host->retune_timer, mmc_retune_timer, (unsigned long)host);

    host->max_segs = 1;             /*初始化实例成员*/
    host->max_seg_size = PAGE_CACHE_SIZE;
    host->max_req_size = PAGE_CACHE_SIZE;
    host->max_blk_size = 512;
    host->max_blk_count = PAGE_CACHE_SIZE / 512;

    return host;            /*返回控制器实例指针*/
```

}

mmc_host 分配函数比较简单，无需多做解释，最重要的工作是设置延时工作 detect 成员的执行函数为 **mmc_rescan()**，它完成控制器中 MMC 设备的探测和激活，创建 MMC 设备数据结构实例并向 MMC 总线添加，下一小节再介绍此函数的实现。

3 添加控制器

控制器驱动 probe()探测函数在分配 mmc_host 实例后，还需要对其进行设置，例如：对 mmc_host->ops 成员赋值，然后调用接口函数 **mmc_add_host()**向驱动框架添加 mmc_host 实例。添加函数中最主要的工作就是完成对控制器中 MMC 设备的探测和激活（激活 host->detect 延时工作）。

mmc_add_host()函数定义在/drivers/mmc/core/host.c 文件内，代码简列如下：

```
int mmc_add_host(struct mmc_host *host)
{
    int err;

    WARN_ON((host->caps & MMC_CAP_SDIO_IRQ) &&!host->ops->enable_sdio_irq);

    err = device_add(&host->class_dev);        /*添加控制器设备*/
    if (err)
        return err;

    led_trigger_register_simple(dev_name(&host->class_dev), &host->led);

#ifdef CONFIG_DEBUG_FS
    mmc_add_host_debugfs(host);
#endif
    mmc_host_clk_sysfs_init(host);

    mmc_start_host(host);      /*触发延时工作，探测 MMC 设备，/drivers/mmc/core/core.c*/
    register_pm_notifier(&host->pm_notify);
    return 0;
}
```

mmc_start_host(host)函数通过激活 host->detect 延时工作完成 MMC 设备的探测和激活。延时工作执行函数 mmc_rescan()内将探测并激活控制器中的 MMC 设备，创建表示 MMC 设备的 mmc_card 结构体实例，并将其添加到 MMC 总线，调用匹配 MMC 设备驱动的探测函数，详见下文。

这里还要再说明一下，在启动内核加载主机控制器驱动程序时，由于是在延时工作中探测 MMC 设备，因此如果设置了挂载 MMC 设备作为根文件系统，要通过“rootdelay=***”命令行参数设置一个延时时间，以便 MMC 设备被探测到并添加了驱动程序。

10.8.3 核心层

MMC 设备驱动核心层定义了 MMC 设备（mmc_card）、驱动（mmc_driver）和总线（mmc_bus_type）数据结构。MMC 设备驱动和总线由内核静态定义，在分配 mmc_host 实例时，其延时工作成员执行函数完成 MMC 设备（mmc_card）的创建和向 MMC 总线的添加。设备添加到总线，将与 MMC 驱动匹配，调用

MMC 驱动的探测函数（probe()）完成 MMC 设备在块设备驱动层数据结构实例的创建和添加。

1 数据结构

下面介绍核心层相关数据结构的定义。

■MMC 设备

MMC 设备由 mmc_card 结构体表示，结构体定义在/include/linux/mmc/card.h 头文件，简列如下：

```
struct mmc_card {
    struct mmc_host    *host;          /*连接的主机控制器实例指针*/
    struct device      dev;            /*表示设备的 device 实例，挂接到 MMC 总线*/
    u32                ocr;            /* the current OCR setting */
    unsigned int       rca;            /* relative card address of device */
    unsigned int       type;           /*卡的类型，SD、SDIO 或 MMC 等*/
    ...
    unsigned int       state;          /*状态*/
    ...
    struct mmc_part    part[MMC_NUM_PHY_PARTITION];
                                       /*物理分区信息，/include/linux/mmc/card.h*/
    unsigned int       nr_parts;       /*分区数量*/
}
```

mmc_card 结构体主要成员简介如下：

●**host**：指向主机控制器 mmc_host 实例。

●**dev**：表示设备的 device 实例。

●**part[]**：mmc_part 结构体数组，表示 MMC 设备的物理分区，如启动分区、普通分区等。在探测激活 MMC 设备时将会填充该数组，在 MMC 驱动的探测函数中将会为每个物理分区创建块设备 gendisk 实例并添加到设备驱动数据库，见下文。

■MMC 设备驱动

MMC 设备驱动由 mmc_driver 结构体表示，定义在/include/linux/mmc/card.h 头文件：

```
struct mmc_driver {
    struct device_driver drv;              /*device_driver 实例，挂接到 MMC 总线*/
    int (*probe)(struct mmc_card *);       /*探测函数，添加块设备驱动层数据结构实例*/
    void (*remove)(struct mmc_card *);
    void (*shutdown)(struct mmc_card *);
};
```

mmc_driver 实例由 MMC 驱动框架定义，如下（/drivers/mmc/card/block.c）：

```
static struct mmc_driver mmc_driver = {
    .drv        = {
        .name     = "mmcblk",
```

```
        .pm  = &mmc_blk_pm_ops,
    },
    .probe          = mmc_blk_probe,   /*添加块设备驱动层数据结构实例，/drivers/mmc/card/block.c*/
    .remove         = mmc_blk_remove,
    .shutdown       = mmc_blk_shutdown,
};
```

mmc_driver 实例中最重要的是探测函数 mmc_blk_probe()，在添加控制器的延时工作中将创建 MMC 设备 mmc_card 实例并添加到 MMC 总线，此时将匹配 MMC 设备驱动 mmc_driver 实例，调用驱动探测函数，完成 MMC 设备驱动数据结构的创建和添加。

■MMC 总线

MMC 驱动框架在/drivers/mmc/core/bus.c 文件内定义了 MMC 总线实例：

```
static struct bus_type mmc_bus_type = {
    .name           = "mmc",
    .dev_groups     = mmc_dev_groups,
    .match          = mmc_bus_match,    /*直接返回 1，mmc_driver 实例与所有 MMC 设备匹配*/
    .uevent         = mmc_bus_uevent,   /*向 uevent 事件添加环境变量*/
    .probe          = mmc_bus_probe,    /*总线探测函数，直接调用 mmc_driver.probe()函数*/
    .remove         = mmc_bus_remove,
    .shutdown       = mmc_bus_shutdown,
    .pm         = &mmc_bus_pm_ops,
};
```

总线匹配函数直接返回 1，表示核心层定义的 mmc_driver 实例可匹配任意 MMC 设备，总线探测函数直接调用 mmc_driver 实例中的 probe()探测函数。

■初始化

MMC 驱动核心层初始化函数为 mmc_init()，定义在/drviers/mmc/core/core.c 文件内：

```
static int __init mmc_init(void)
{
    int ret;

    workqueue = alloc_ordered_workqueue("kmmcd", 0);      /*创建工作队列*/
    ...

    ret = mmc_register_bus();      /*注册 mmc_bus_type 总线实例，/drivers/mmc/core/bus.c*/
    ...

    ret = mmc_register_host_class();      /*注册主机控制器设备类，/drivers/mmc/core/host.c*/
    ...

    ret = sdio_register_bus();
```

```
                    /*注册 SDIO 卡专用的总线 sdio_bus_type，/drivers/mmc/core/sdio_bus.c*/
    ...
    return 0;
    ...
}
subsys_initcall(mmc_init);
```

2 探测/添加设备

前面介绍过，在添加主机控制器的 mmc_start_host(host)函数中将激活 mmc_host 实例的延时工作，执行函数 mmc_rescan()将完成 MMC 设备的探测和激活，创建 MMC 设备 mmc_card 实例并添加到 MMC 总线，下面简单看一下该函数的实现。

mmc_rescan()函数定义在/drivers/mmc/core/core.c 文件内，代码简列如下：

```
void mmc_rescan(struct work_struct *work)
{
    struct mmc_host *host =container_of(work, struct mmc_host, detect.work);        /*主机控制器*/
    int i;

    if (host->trigger_card_event && host->ops->card_event) {
        host->ops->card_event(host);
        host->trigger_card_event = false;
    }

    if (host->rescan_disable)
        return;

    /* If there is a non-removable card registered, only scan once */
    if ((host->caps & MMC_CAP_NONREMOVABLE) && host->rescan_entered)
        return;
    host->rescan_entered = 1;

    mmc_bus_get(host);      /*host->bus_refs++，/drivers/mmc/core/core.c*/

    if (host->bus_ops && !host->bus_dead&& !(host->caps & MMC_CAP_NONREMOVABLE))
        host->bus_ops->detect(host);          /*探测函数*/

    host->detect_change = 0;

    mmc_bus_put(host);     /*host->bus_refs--*/
    mmc_bus_get(host);

    if (host->bus_ops != NULL) {
        mmc_bus_put(host);
        goto out;
```

```
        }
        mmc_bus_put(host);

        if (!(host->caps & MMC_CAP_NONREMOVABLE) && host->ops->get_cd &&
                host->ops->get_cd(host) == 0) {
            mmc_claim_host(host);
            mmc_power_off(host);
            mmc_release_host(host);
            goto out;
        }

        mmc_claim_host(host);   /*当前进程持有控制器，/drivers/mmc/core/core.c*/
        for (i = 0; i < ARRAY_SIZE(freqs); i++) {   /*频率列表，/drivers/mmc/core/core.c*/
            if (!mmc_rescan_try_freq(host, max(freqs[i], host->f_min)))
                                                /*探测 MMC 设备，/drivers/mmc/core/core.c*/
                break;
            if (freqs[i] <= host->f_min)
                break;
        }
        mmc_release_host(host);

     out:
        if (host->caps & MMC_CAP_NEEDS_POLL)
            mmc_schedule_delayed_work(&host->detect, HZ);       /*调度延时工作*/
    }
```

这里我们主要看一下探测 MMC 设备的 mmc_rescan_try_freq()函数，它负责探测并激活控制器中 MMC 设备，探测的顺序为 SDIO 卡、SD 卡、MMC 卡。

mmc_rescan_try_freq()函数源代码请读者自行阅读，这里看一下其探测 SD 卡调用的 mmc_attach_sd()函数，代码简列如下（/drivers/mmc/core/sd.c）：

```
    int mmc_attach_sd(struct mmc_host *host)
    {
        ...
        mmc_attach_bus(host, &mmc_sd_ops);      /*host->bus_ops=mmc_sd_ops*/
        ...
        err = mmc_sd_init_card(host, rocr, NULL);
                        /*探测并初始化 SD 卡（激活 SD 卡），创建 mmc_card 实例，/drivers/mmc/core/sd.c*/
        ...
        err = mmc_add_card(host->card);
                /*mmc_card 实例添加至 mmc_bus_type 总线，匹配驱动，/drivers/mmc/core/bus.c*/
        ...
        return 0;
        ...
    }
```

mmc_attach_sd()函数用于探测 SD 卡。mmc_attach_bus()函数对 host->bus_ops 成员赋值，对于 SD 卡其值为 mmc_sd_ops 实例（mmc_bus_ops 结构体）指针。mmc_sd_init_card()函数用于探测激活 SD 卡（获取卡信息），创建 mmc_card 实例并初始化等。 mmc_add_card()主要用于将 mmc_card 实例添加到 MMC 总线，此时将匹配 MMC 驱动 mmc_driver 实例，并调用其中的 probe()函数，完成 MMC 设备在块设备驱动层数据结构实例的创建和添加，见下文。

10.8.4 块设备驱动层

在向 MMC 总线添加 mmc_card 实例时将匹配 MMC 驱动 mmc_driver 实例，并调用其中 probe()函数完成 MMC 设备在块设备驱动层数据结构实例的创建和添加。

mmc_driver 实例的 probe()函数为 mmc_blk_probe()。

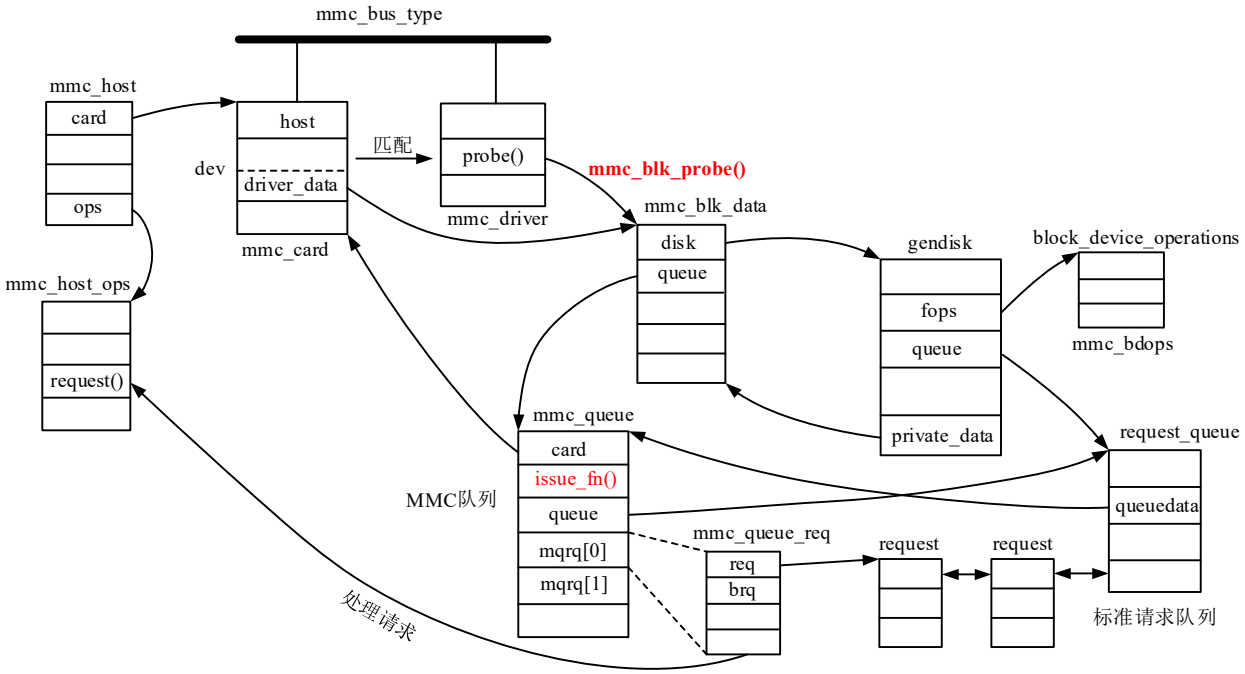
1 初始化

在介绍 MMC 驱动探测函数前，先看一下块设备驱动层的初始化函数（/drivers/mmc/card/block.c）：

```
static int __init mmc_blk_init(void)
{
    int res;

    if (perdev_minors != CONFIG_MMC_BLOCK_MINORS)
        pr_info("mmcblk: using %d minors per device\n", perdev_minors);

    max_devices = min(MAX_DEVICES, (1 << MINORBITS) / perdev_minors);

    res = register_blkdev(MMC_BLOCK_MAJOR, "mmc");     /*注册块设备设备号*/
    if (res)
        goto out;

    res = mmc_register_driver(&mmc_driver);     /*注册 mmc_driver，/drivers/mmc/core/bus.c*/
    if (res)
        goto out2;
    return 0;
    ...
}
module_init(mmc_blk_init);
```

2 探测函数

MMC 设备驱动 mmc_driver 实例探测函数为 mmc_blk_probe()，函数内创建的数据结构实例如下图所示：



mmc_blk_data 结构体表示 MMC 设备的信息，其中 disk 成员指向表示块设备的 gendisk 实例，queue 成员为内嵌 mmc_queue 结构体实例（图中画成指针了）。如果 MMC 设备中包含多个物理分区，则对每个分区创建 mmc_blk_data 结构体实例，并组成链表。

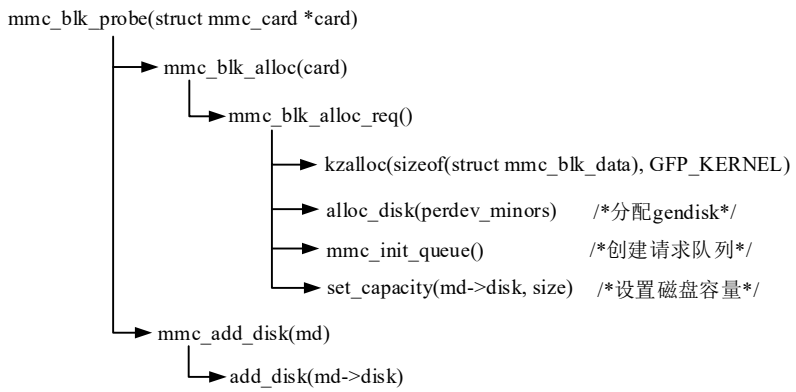
mmc_blk_data 结构体定义如下（/drivers/mmc/card/block.c）：

```
struct mmc_blk_data {
    spinlock_t      lock;
    struct gendisk  *disk;          /*指向磁盘 gendisk 实例*/
    struct mmc_queue queue;         /*MMC 队列，不是块设备驱动中的请求队列*/
    struct list_head part;          /*双链表成员*/

    unsigned int    flags;          /*标记*/
    ...
    unsigned int    usage;
    unsigned int    read_only;
    unsigned int    part_type;
    unsigned int    name_idx;
    unsigned int    reset_done;
    ...
    unsigned int    part_curr;
    struct device_attribute force_ro;
    struct device_attribute power_ro_lock;
    int    area_type;
};
```

mmc_blk_probe()函数定义在/drivers/mmc/card/block.c 文件内，代码简列如下：

```
static int mmc_blk_probe(struct mmc_card *card)
{
    struct mmc_blk_data *md, *part_md;
    ...
    md = mmc_blk_alloc(card);       /*创建 mmc_blk_data 实例，/drivers/mmc/card/block.c*/
    ...
    if (mmc_blk_alloc_parts(card, md))        /*只对 MMC 卡有效*/
                    /*为物理分区创建 mmc_blk_data 实例，card->part[idx]，/drivers/mmc/card/block.c*/
        goto out;

    dev_set_drvdata(&card->dev, md);     /*card->dev.driver_data=md，mmc_blk_data 实例*/

    if (mmc_add_disk(md))     /*添加 gendisk 实例，/drivers/mmc/card/block.c*/
        goto out;

    list_for_each_entry(part_md, &md->part, part) {
        if (mmc_add_disk(part_md))          /*添加物理分区的 gendisk 实例*/
            goto out;
    }
    pm_runtime_set_autosuspend_delay(&card->dev, 3000);
    pm_runtime_use_autosuspend(&card->dev);

    if (card->type != MMC_TYPE_SD_COMBO) {
        pm_runtime_set_active(&card->dev);
        pm_runtime_enable(&card->dev);
    }
    return 0;
    ...
}
```

mmc_blk_alloc()函数调用关系如下图所示：



mmc_blk_alloc()函数调用 mmc_blk_alloc_req()函数创建 mmc_blk_data 实例，包括块设备 gendisk 实例、MMC 队列及其关联的标准请求队列等。mmc_add_disk(md)用于将 gendisk 实例添加到块设备驱动数据库。

对于 MMC 卡，mmc_blk_alloc_parts()函数对分区创建 mmc_blk_data 实例，后面也将添加关联的 gendisk

实例。这里我们只考虑 SD 卡，因此 mmc_blk_alloc_parts()函数不会执行什么工作，可将其忽略。

■创建 mmc_blk_data 实例

mmc_blk_alloc()函数内调用 mmc_blk_alloc_req()函数创建 mmc_blk_data 实例，包括关联的 gendisk 和队列实例，函数代码简列如下（/drivers/mmc/card/block.c）：

```
static struct mmc_blk_data *mmc_blk_alloc_req(struct mmc_card *card,struct device *parent,sector_t size,
                                              bool default_ro,const char *subname,int area_type)
/*size：卡容量，扇区数，由调用者传递，在探测 SD 卡时获取（card->csd.capacity）*/
{
    struct mmc_blk_data *md;
    int devidx, ret;

    devidx = find_first_zero_bit(dev_use, max_devices);       /*内核用位图表示卡的编号*/
    if (devidx >= max_devices)
        return ERR_PTR(-ENOSPC);
    __set_bit(devidx, dev_use);

    md = kzalloc(sizeof(struct mmc_blk_data), GFP_KERNEL);     /*分配 mmc_blk_data 实例*/
    ...

    if (!subname) {
        md->name_idx = find_first_zero_bit(name_use, max_devices);
        __set_bit(md->name_idx, name_use);
    } else
        md->name_idx = ((struct mmc_blk_data *)dev_to_disk(parent)->private_data)->name_idx;

    md->area_type = area_type;

    md->read_only = mmc_blk_readonly(card);       /*卡是否只读*/

    md->disk = alloc_disk(perdev_minors);     /*分配 gendisk 实例，从设备号为全局变量*/
                                              /*perdev_minors = CONFIG_MMC_BLOCK_MINORS*/
    ...

    spin_lock_init(&md->lock);
    INIT_LIST_HEAD(&md->part);
    md->usage = 1;

    ret = mmc_init_queue(&md->queue, card, &md->lock, subname);
                                                  /*创建标准请求队列（MMC 队列），见下文*/
    if (ret)
        goto err_putdisk;
```

```
    md->queue.issue_fn = mmc_blk_issue_rq;      /*执行单个请求的函数，/drivers/mmc/card/block.c*/
    md->queue.data = md;

    md->disk->major   = MMC_BLOCK_MAJOR;
    md->disk->first_minor = devidx * perdev_minors;
    md->disk->fops = &mmc_bdops;      /*块设备操作结构实例，/drivers/mmc/card/block.c*/
    md->disk->private_data = md;
    md->disk->queue = md->queue.queue;    /*标准请求队列*/
    md->disk->driverfs_dev = parent;
    set_disk_ro(md->disk, md->read_only || default_ro);
    if (area_type & (MMC_BLK_DATA_AREA_RPMB | MMC_BLK_DATA_AREA_BOOT))
        md->disk->flags |= GENHD_FL_NO_PART_SCAN;

    snprintf(md->disk->disk_name, sizeof(md->disk->disk_name),
                "mmcblk%u%s", md->name_idx, subname ? subname : "");    /*磁盘名称*/

    if (mmc_card_mmc(card))    /*MMC 卡*/
        blk_queue_logical_block_size(md->queue.queue,card->ext_csd.data_sector_size);
    else
        blk_queue_logical_block_size(md->queue.queue, 512);      /*设置数据块大小为 512 字节*/

    set_capacity(md->disk, size);      /*设置磁盘容量*/
    ...
    return md;
    ...
}
```

mmc_blk_alloc_req()函数创建了 mmc_blk_data 实例，创建了 gendisk 实例，调用 mmc_init_queue()函数创建 MMC 队列（内含标准请求队列），最后对 gendisk 和队列进行设置。gendisk 实例的名称设为 **mmcblk*xy*** 后接系统中 MMC 设备编号 s，y 为子名称，这也是 MMC 设备文件的名称。

下面介绍一下 MMC 队列及其创建。

●请求队列

MMC 队列由 mmc_queue 结构体表示，定义如下（/drivers/mmc/card/queue.h）：

```
struct mmc_queue {
    struct mmc_card     *card;          /*MMC 设备*/
    struct task_struct  *thread;        /*内核线程*/
    struct semaphore    thread_sem;
    unsigned int        flags;          /*标记*/
    ...
    int                 (*issue_fn)(struct mmc_queue *, struct request *);
    void                *data;
    struct request_queue    *queue;     /*标准请求队列*/
```

```
    struct mmc_queue_req   mqrq[2];              /*管理 MMC 请求的数据结构数组*/
    struct mmc_queue_req   *mqrq_cur;            /*当前处理的 MMC 请求，初始指向 mqrq[0]*/
    struct mmc_queue_req   *mqrq_prev;           /*上一个请求，初始指向 mqrq[1]*/
};
```

mmc_init_queue()函数用于为 MMC 设备创建 MMC 队列，函数定义如下（/drivers/mmc/card/queue.c）：

```
int mmc_init_queue(struct mmc_queue *mq, struct mmc_card *card,spinlock_t *lock, const char *subname)
/*mq：指向 mmc_blk_data 实例内嵌 mmc_queue 实例*/
{
    struct mmc_host *host = card->host;
    u64 limit = BLK_BOUNCE_HIGH;
    int ret;
    struct mmc_queue_req *mqrq_cur = &mq->mqrq[0];
    struct mmc_queue_req *mqrq_prev = &mq->mqrq[1];

    if (mmc_dev(host)->dma_mask && *mmc_dev(host)->dma_mask)
        limit = (u64)dma_max_pfn(mmc_dev(host)) << PAGE_SHIFT;

    mq->card = card;
    mq->queue = blk_init_queue(mmc_request_fn, lock);
                                /*创建标准请求队列，请求处理函数 mmc_request_fn()*/
    if (!mq->queue)
        return -ENOMEM;

    mq->mqrq_cur = mqrq_cur;          /*指向 mq->mqrq[0]*/
    mq->mqrq_prev = mqrq_prev;        /*指向 mq->mqrq[1]*/
    mq->queue->queuedata = mq;        /*指向 mmc_queue 实例*/

    blk_queue_prep_rq(mq->queue, mmc_prep_request);
    queue_flag_set_unlocked(QUEUE_FLAG_NONROT, mq->queue);
    queue_flag_clear_unlocked(QUEUE_FLAG_ADD_RANDOM, mq->queue);
    if (mmc_can_erase(card))
        mmc_queue_setup_discard(mq->queue, card);
    ...

    mq->thread = kthread_run(mmc_queue_thread, mq, "mmcqd/%d%s",host->index,  \
                                        subname ? subname : "");      /*创建内核线程*/
    ...
    return 0;
    ...
}
```

mmc_init_queue()函数内为 mmc_queue 实例（内嵌在 mmc_blk_data 实例）创建了标准请求队列，请求处理函数为 **mmc_request_fn()**，并创建内核线程，线程执行函数为 **mmc_queue_thread()**。内核线程用于

处理请求，而请求处理函数 mmc_request_fn()只是唤醒内核线程，后面将详细介绍。

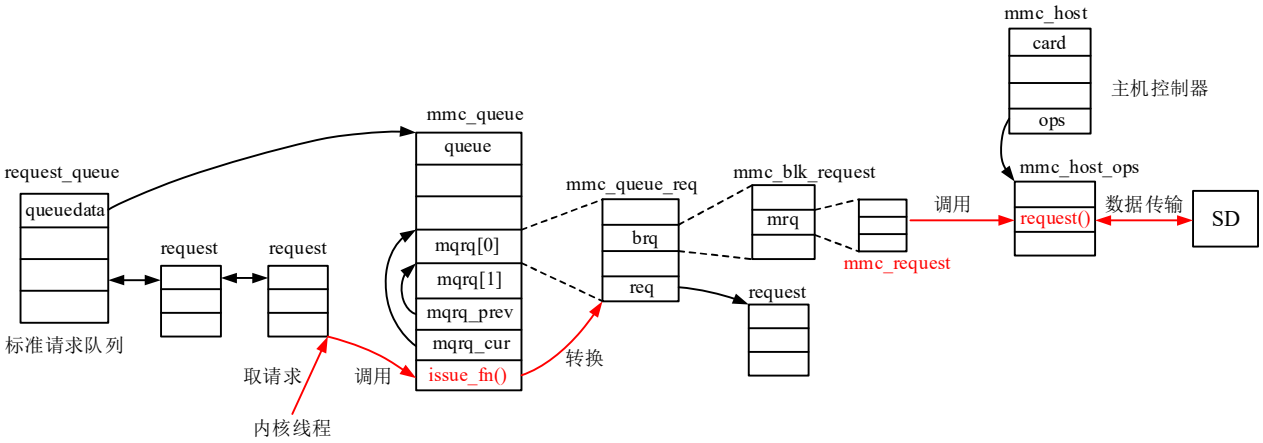
■添加磁盘

MMC 设备驱动探测函数 mmc_blk_probe()在创建 mmc_blk_data 实例后，调用 mmc_add_disk()函数添加实例关联的 gendisk 实例，函数定义如下（/drivers/mmc/card/block.c）：

```
static int mmc_add_disk(struct mmc_blk_data *md)
{
    int ret;
    struct mmc_card *card = md->queue.card;

    add_disk(md->disk);              /*添加磁盘*/
    md->force_ro.show = force_ro_show;
    md->force_ro.store = force_ro_store;
    sysfs_attr_init(&md->force_ro.attr);
    md->force_ro.attr.name = "force_ro";
    md->force_ro.attr.mode = S_IRUGO | S_IWUSR;
    ret = device_create_file(disk_to_dev(md->disk), &md->force_ro);
    if (ret)
        goto force_ro_fail;

    if ((md->area_type & MMC_BLK_DATA_AREA_BOOT) &&card->ext_csd.boot_ro_lockable) {
        umode_t mode;

        if (card->ext_csd.boot_ro_lock & EXT_CSD_BOOT_WP_B_PWR_WP_DIS)
            mode = S_IRUGO;
        else
            mode = S_IRUGO | S_IWUSR;

        md->power_ro_lock.show = power_ro_lock_show;
        md->power_ro_lock.store = power_ro_lock_store;
        sysfs_attr_init(&md->power_ro_lock.attr);
        md->power_ro_lock.attr.mode = mode;
        md->power_ro_lock.attr.name ="ro_lock_until_next_power_on";
        ret = device_create_file(disk_to_dev(md->disk),&md->power_ro_lock);
        if (ret)
            goto power_ro_lock_fail;
    }
    return ret;
    ...
}
```

4 处理请求

由前面的介绍可知，MMC 设备驱动程序中创建了一个内核线程来处理请求，内核线程执行流程简列如下图所示：



内核线程从标准请求队列中取出一个请求，调用 mmc_queue 实例中的 issue_fn()函数，将请求转换成 MMC 请求，设为当前 MMC 请求（mqrq_cur 指向的请求）。MMC 请求由 mmc_request 结构体表示，其中包含执行请求时的 SD 卡命令和数据等信息，调用主机控制器驱动定义的 mmc_host_ops 实例中的 request()函数执行 MMC 请求，通过 SD 卡命令完成数据的传输。内核线程重复以上动作处理请求。

■处理请求函数

MMC 设备标准请求队列请求处理函数为 **mmc_request_fn()**，定义如下（/drivers/mmc/card/queue.c）：

```
static void mmc_request_fn(struct request_queue *q)
{
    struct mmc_queue *mq = q->queuedata;
    struct request *req;
    unsigned long flags;
    struct mmc_context_info *cntx;

    if (!mq) {          /*如果没有关联的 mmc_queue 实例，通常是关联的，不执行 if 内代码*/
        while ((req = blk_fetch_request(q)) != NULL) {
            req->cmd_flags |= REQ_QUIET;
            __blk_end_request_all(req, -EIO);          /*结束请求*/
        }
        return;
    }

    cntx = &mq->card->host->context_info;
    if (!mq->mqrq_cur->req && mq->mqrq_prev->req) {
        spin_lock_irqsave(&cntx->lock, flags);
        if (cntx->is_waiting_last_req) {
            cntx->is_new_req = true;
```

```
            wake_up_interruptible(&cntx->wait);
        }
        spin_unlock_irqrestore(&cntx->lock, flags);
    } else if (!mq->mqrq_cur->req && !mq->mqrq_prev->req)        /*两个都为空，唤醒内核线程*/
        wake_up_process(mq->thread);        /*唤醒创建的内核线程*/
}
```

mmc_request_fn()函数的主要工作就是唤醒处理请求的内核线程。

■内核线程

下面看一下内核线程如何处理 MMC 设备请求。

内核线程执行函数为 **mmc_queue_thread()**，定义如下（/drivers/mmc/card/queue.c）：

```
static int mmc_queue_thread(void *d)
{
    struct mmc_queue *mq = d;
    struct request_queue *q = mq->queue;

    current->flags |= PF_MEMALLOC;    /*内核线程 task_struct*/

    down(&mq->thread_sem);

    do {        /*无限循环*/
        struct request *req = NULL;
        unsigned int cmd_flags = 0;

        spin_lock_irq(q->queue_lock);
        set_current_state(TASK_INTERRUPTIBLE);    /*设置内核线程为不可中断睡眠状态*/
        req = blk_fetch_request(q);      /*从请求队列取请求*/
        mq->mqrq_cur->req = req;      /*关联到当前 MMC 请求*/
        spin_unlock_irq(q->queue_lock);

        if (req || mq->mqrq_prev->req) {
            set_current_state(TASK_RUNNING);      /*内核线程设为运行状态*/
            cmd_flags = req ? req->cmd_flags : 0;
            mq->issue_fn(mq, req);      /*处理单个请求，执行函数为 mmc_blk_issue_rq()*/
            cond_resched();
            if (mq->flags & MMC_QUEUE_NEW_REQUEST) {
                mq->flags &= ~MMC_QUEUE_NEW_REQUEST;
                continue;    /* fetch again */
            }
            if (cmd_flags & MMC_REQ_SPECIAL_MASK)
                mq->mqrq_cur->req = NULL;
```

```
                mq->mqrq_prev->brq.mrq.data = NULL;
                mq->mqrq_prev->req = NULL;
                swap(mq->mqrq_prev, mq->mqrq_cur);     /*交换指针成员*/
            } else {
                if (kthread_should_stop()) {
                    set_current_state(TASK_RUNNING);
                    break;
                }
                up(&mq->thread_sem);
                schedule();          /*进程调度*/
                down(&mq->thread_sem);
            }
        } while (1);            /*无限循环结束*/
        up(&mq->thread_sem);
        return 0;
    }
```

mmc_queue_thread()函数内是一个无限循环，循环体从请求队列中取请求，然后调用 mq->issue_fn(mq, req)函数处理请求，然后再取下一个请求，如此循环。

在前面介绍的 mmc_blk_alloc_req()函数中，mmc_queue->issue_fn 成员赋值为 **mmc_blk_issue_rq()**函数指针，函数定义如下（/drivers/mmc/card/block.c）：

```
    static int mmc_blk_issue_rq(struct mmc_queue *mq, struct request *req)
    {
        int ret;
        struct mmc_blk_data *md = mq->data;
        struct mmc_card *card = md->queue.card;
        struct mmc_host *host = card->host;
        unsigned long flags;
        unsigned int cmd_flags = req ? req->cmd_flags : 0;

        if (req && !mq->mqrq_prev->req)
            mmc_get_card(card);

        ret = mmc_blk_part_switch(card, md);        /*如果不是 MMC 卡，不需要做什么工作*/
        ...

        mq->flags &= ~MMC_QUEUE_NEW_REQUEST;
        if (cmd_flags & REQ_DISCARD) {
            if (card->host->areq)
                mmc_blk_issue_rw_rq(mq, NULL);
            if (req->cmd_flags & REQ_SECURE)
                ret = mmc_blk_issue_secdiscard_rq(mq, req);
            else
```

```
            ret = mmc_blk_issue_discard_rq(mq, req);
        } else if (cmd_flags & REQ_FLUSH) {          /*FLUSH 请求*/
            if (card->host->areq)
                mmc_blk_issue_rw_rq(mq, NULL);
            ret = mmc_blk_issue_flush(mq, req);
        } else {
            if (!req && host->areq) {
                spin_lock_irqsave(&host->context_info.lock, flags);
                host->context_info.is_waiting_last_req = true;
                spin_unlock_irqrestore(&host->context_info.lock, flags);
            }
            ret = mmc_blk_issue_rw_rq(mq, req);      /*普通读写请求，/drivers/mmc/card/block.c*/
        }

    out:
        if ((!req && !(mq->flags & MMC_QUEUE_NEW_REQUEST)) ||
             (cmd_flags & MMC_REQ_SPECIAL_MASK))
            mmc_put_card(card);
        return ret;
    }
```

以上代码中 mmc_blk_issue_rw_rq()函数用于处理普通的读写请求，函数内依请求设置 mmc_request 实例，最后调用 host->ops->request(host, mrq)函数完成数据传输，请求处理完成后调用 blk_end_request()函数结束请求，有兴趣的读者可自行阅读相关源代码。

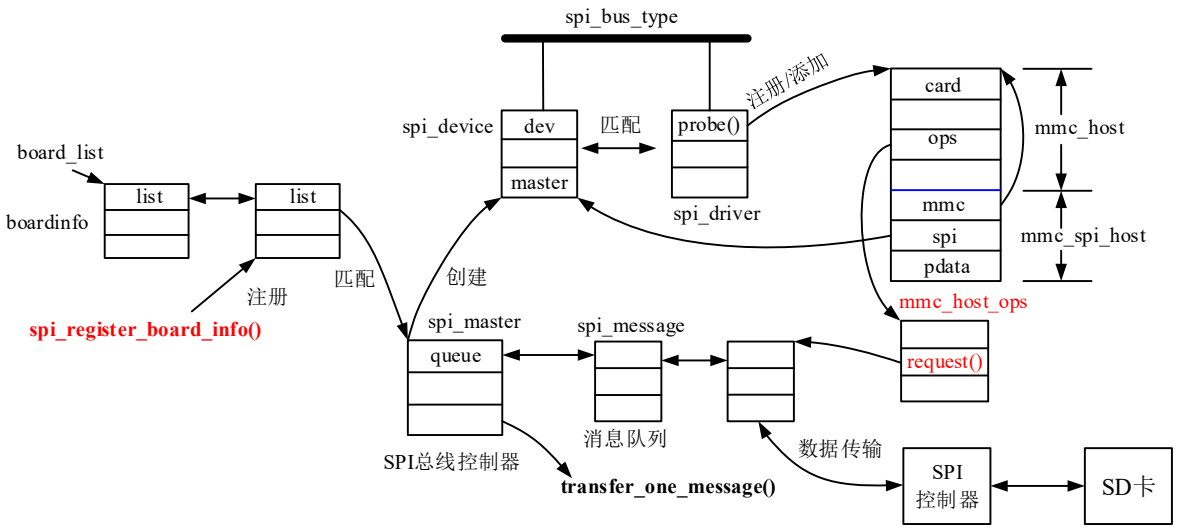
10.8.5 驱动示例

MMC 设备驱动程序其实内核已经实现了，用户真正需做的是实现 MMC 设备主机控制器驱动程序。

本小节以龙芯 1B 开发板为例，介绍其 SD 卡控制驱动程序的实现。开发板通过处理器的 SPI 总线与 SD 卡槽连接，SD 卡控制驱动可直接采用/drivers/mmc/host/mmc_spi.c 中实现的驱动程序，并不需要修改文件中源代码，但需要实现 SPI 控制器驱动并注册 SD 卡使用的 SPI 接口的板级信息。

驱动程序框架如下图所示，请读者先回顾一下第 8 章的 SPI 总线驱动。采用 SPI 接口的 SD 卡控制器驱动中定义了 SPI 设备驱动 spi_driver 实例，在其探测函数中将创建 mmc_host 实例（后接 mmc_spi_host 结构体实例）。驱动程序中需要实现 **mmc_host_ops** 实例，实例中的函数调用 SPI 总线数据传输的接口函数实现 SD 卡命令和数据传输。SPI 设备 spi_device 实例在调用 spi_register_board_info()函数注册 SPI 接口的板级信息时由内核自动创建。

在实现了 SPI 控制器的驱动时，SPI 接口 SD 卡驱动只需要在板级文件中调用 spi_register_board_info()函数注册 SPI 接口信息即可直接使用 mmc_spi.c 文件实现的 SD 卡控制驱动程序。



1 注册板级信息

龙芯 1B 开发板中使 SPI1 控制器连接 SD 卡槽（片选信号 0），在板级相关文件中定义了接口信息：

```
static struct spi_board_info ls1x_spi1_devices[] = {
    {
        .modalias       = "mmc_spi",    /*需与表示 SD 卡设备驱动的 spi_driver 实例中的名称相同*/
        .bus_num        = 1,            /*总线控制器 1*/
        .chip_select    = SPI1_CS0,     /*片选信号*/
        .max_speed_hz   = 25000000,     /*传输速率*/
        .platform_data= &mmc_spi,       /*需用户实现的结构体实例*/
                    /*mmc_spi_platform_data 结构体实例，/include/linux/spi/mmc_spi.h*/
    },
    ...
}
```

在板级相关的初始化函数需要注册此接口信息，如下所示：

```
int __init ls1b_platform_init(void)
{
    ...
    spi_register_board_info(ls1x_spi1_devices, ARRAY_SIZE(ls1x_spi1_devices));
    ...
}
arch_initcall(ls1b_platform_init);
```

2 SPI 设备驱动

在/drivers/mmc/host/mmc_spi.c 文件中定义了 SPI 设备（SD 卡主机控制器）驱动，如下：

```
static struct spi_driver mmc_spi_driver = {
    .driver = {
        .name ="mmc_spi",
```

```
            .owner =  THIS_MODULE,
            .of_match_table = mmc_spi_of_match_table,
        },
        .probe =    mmc_spi_probe,          /*探测函数*/
        .remove =  mmc_spi_remove,
    };
```

驱动探测函数 mmc_spi_probe()中需要实现 MMC 设备主机控制器实例的创建、初始化和添加，函数代码简列如下：

```
    static int mmc_spi_probe(struct spi_device *spi)
    {
        void                *ones;
        struct mmc_host         *mmc;
        struct mmc_spi_host     *host;      /*mmc_spi_host 实例指针*/
        int                 status;
        bool                has_ro = false;
        ...
        status = spi_setup(spi);
        ...
        mmc = mmc_alloc_host(sizeof(*host), &spi->dev);     /*分配 mmc_host 和 mmc_spi_host 实例*/
        ...
        mmc->ops = &mmc_spi_ops;        /*mmc_host_ops 结构体实例*/
        ...
        host->pdata = mmc_spi_get_pdata(spi);     /*指向 mmc_spi_platform_data 结构体实例*/
        ...
        dev_set_drvdata(&spi->dev, mmc);        /*device.driver_data=mmc_host*/
        ...
        status = mmc_add_host(mmc);        /*添加 MMC 主机控制器实例*/
        ...
        return 0;
        ...
    }
```

SPI 接口的 SD 卡主机控制器关联的 mmc_host_ops 实例为 **mmc_spi_ops**，定义如下：

```
    static const struct mmc_host_ops mmc_spi_ops = {
        .request    = mmc_spi_request,          /*处理 MMC 请求*/
        .set_ios    = mmc_spi_set_ios,
        .get_ro         = mmc_gpio_get_ro,
        .get_cd         = mmc_gpio_get_cd,
    };
```

mmc_spi_ops 实例中最重要的 mmc_spi_request()函数就是通过 SPI 总线来传输 SD 卡命令和数据。SD 卡命令以及数据的传输请读者参考 SD 卡协议，mmc_spi_request()函数源代码请有兴趣的读者自行阅读。

10.9 MTD 设备驱动

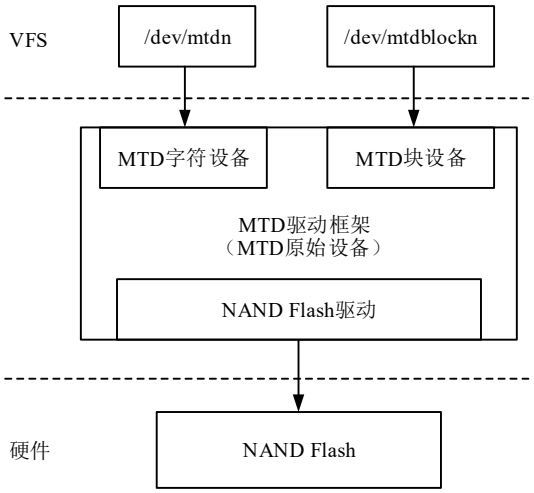
NAND Flash（闪存）是嵌入式系统中常见的存储设备，可用以存储启动代码、内核、文件系统等。内核 NAND Flash 驱动采用了 MTD（Memory Technology Device）驱动框架。MTD 驱动框架不仅适用于 NAND Flash，还适用于 NorFlash、OneNAND 等设备。

本章首先介绍 MTD 驱动框架，然后介绍 NAND Flash 硬件原理及驱动框架，最后示例说明 NAND Flash 驱动程序的实现。

MTD 框架代码位于/drivers/mtd/目录下，NAND Flash 驱动代码位于/drivers/mtd/nand/目录下。

10.9.1 概述

MTD 驱动框架为底层硬件（闪存等）和上层（文件系统）之间提供统一的抽象接口，是两层之间的桥梁，如下图所示。MTD 层为文件系统层提供访问底层硬件的字符设备文件和块设备文件接口，用于对下层硬件进行操作。



NAND Flash 驱动采用了 MTD 驱动框架，NAND Flash 驱动只需要实现底层硬件操作函数并注册相关数据结构实例即可。

1 驱动框架

MTD 驱动框架如下图所示，mtd_info 结构体表示 MTD 原始设备，其中包含底层设备操作（读写数据块）的函数指针成员。具体 MTD 设备驱动程序主要工作就是创建并注册 mtd_info 实例，同时需要通过静态数组、命令行参数等形式向内核传递 MTD 设备的分区信息。

内核将 MTD 设备既视为字符设备，又视为块设备，因此有字符设备驱动程序和块设备驱动程序。MTD 字符设备驱动程序在初始化时注册，字符设备文件的读写操作函数调用 mtd_info 实例中的底层操作函数实现数据传输。块设备驱动程序在注册 mtd_info 实例注册。

在注册 mtd_info 实例时，将为每个分区创建并添加 mtd_part 结构体实例。mtd_part 结构体中 master 成员指向表示 MTD 原始设备的 mtd_info 实例，结构体中还嵌入了 mtd_info 结构体成员，它将复制 master 指向实例中的数据。

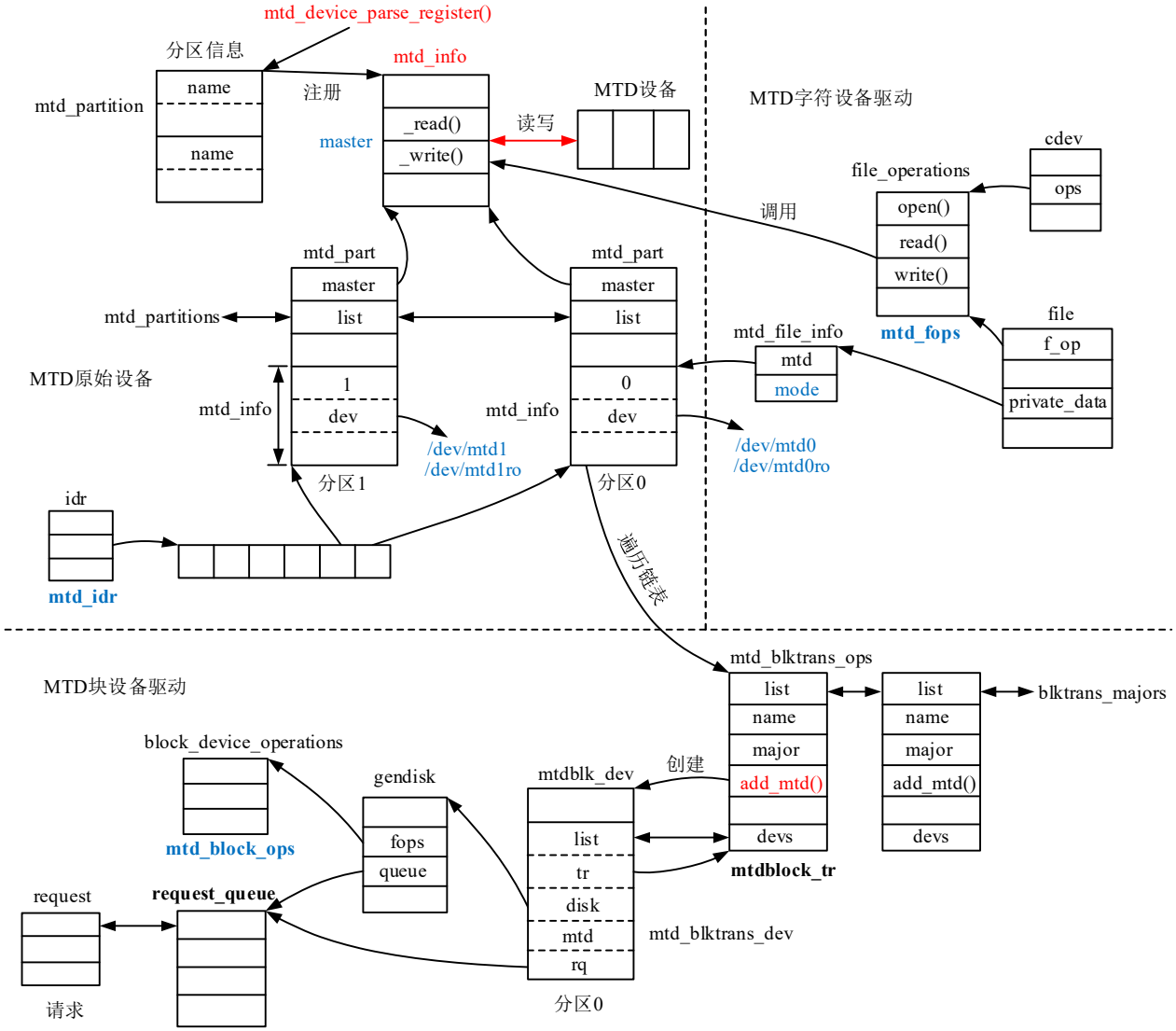
如果 MTD 设备没有分区，则将注册表示原始设备的 mtd_info 实例，如果有分区，则只注册表示分区的 mtd_info 实例，并创建字符设备文件。所有注册的 mtd_info 实例内核为其分配全局唯一的编号，也作为块设备的从设备号。

内核支持以多种传输模式访问 MTD 块设备，每种传输模式由 mtd_blktrans_ops 结构体实例。对于每种

传输模式，MTD 设备都有对应的块设备驱动程序。

在注册 mtd_info 实例时，将遍历 mtd_blktrans_ops 实例链表，调用 mtd_blktrans_ops 实例中的 **add_mtd()** 函数，为 mtd_info 实例注册块设备驱动程序（每个分区都有）。

mtd_blktrans_dev 结构体管理着 mtd_info 实例对应的块设备驱动程序。mtd_blktrans_dev 结构体中包含一个工作队列和工作成员，块设备驱动程序中标准请求队列的请求处理函数就是激活这个工作。这个工作的执行函数将循环从队列中取请求，调用 mtd_blktrans_ops 实例中的函数执行请求的中数据传输，最后结束请求，而 mtd_blktrans_ops 实例中的函数最终也将调用 mtd_info 实例中的底层操作函数实现数据传输。



2 初始化

MTD 驱动框架初始化函数 init_mtd()定义如下（/drivers/mtd/mtdcore.c）：

```
static int __init init_mtd(void)
{
    int ret;

    ret = class_register(&mtd_class);     /*注册 MTD 设备类*/
    ...
    ret = mtd_bdi_init(&mtd_bdi, "mtd");
```

```
                                /*初始化并注册 backing_dev_info 实例，/drivers/mtd/mtdcore.c*/
    ...
    proc_mtd = proc_create("mtd", 0, NULL, &mtd_proc_ops);   /*在 proc 文件系统中创建 mtd 文件*/
    ret = init_mtdchar();     /*注册 MTD 字符设备驱动程序，见下文*/
    ...
    return 0;
    ...
}
module_init(init_mtd);
```

10.9.2 MTD 原始设备

MTD 原始设备由 **mtd_info** 结构体表示，其实例由板级代码定义并向内核注册。在注册 mtd_info 实例时需要同时传递 MTD 设备的分区信息，在注册 MTD 原始设备时需要解析 MTD 原始设备的分区信息。分区信息可通过命令行参数、静态定义的分区信息数组等方式传递，注册 mtd_info 实例的操作中将为各分区创建设备文件。

MTD 设备驱动程序的主要工作就是传递设备分区信息，定义并注册原始设备 mtd_info 实例。

1 mtd_info

MTD 原始设备由 mtd_info 结构体表示，定义如下（/include/linux/mtd/mtd.h）：

```
struct mtd_info {
    u_char type;       /*设备类型*/
    uint32_t flags;     /*标记*/
    uint64_t size;       /*MTD 设备总大小，字节数*/
    uint32_t erasesize;     /*擦写单元大小*/
    uint32_t writesize;     /*写单元大小*/
    uint32_t writebufsize;     /*写缓冲大小*/

    uint32_t oobsize;     /*每个 block 中的 OOB 数据大小*/
    uint32_t oobavail;   /*每个 block 中的可用 OOB 数据大小*/
    unsigned int erasesize_shift;
    unsigned int writesize_shift;
    unsigned int erasesize_mask;
    unsigned int writesize_mask;

    unsigned int bitflip_threshold;     /*位反转阀值*/

    /*以下是仅内核使用的成员*/
    const char *name;     /*名称*/
    int index;                /*设备全局编号，在设备文件中用于生成从设备号*/

    struct nand_ecclayout *ecclayout;   /*ECC 布局*/
    unsigned int ecc_step_size;     /*ECC step 大小*/
```

```
/*每个 ECC step 允许的最大可修复位*/
unsigned int ecc_strength;

/*擦写区信息*/
int numeraseregions;
struct mtd_erase_region_info *eraseregions;

/*底层操作回调函数，不要直接调用，由 mtd_*()函数调用（字符设备、块设备操作函数）*/
int (*_erase) (struct mtd_info *mtd, struct erase_info *instr);
int (*_point) (struct mtd_info *mtd, loff_t from, size_t len,
                                        size_t *retlen, void **virt, resource_size_t *phys);
int (*_unpoint) (struct mtd_info *mtd, loff_t from, size_t len);
unsigned long (*_get_unmapped_area) (struct mtd_info *mtd,unsigned long len,
                                                    unsigned long offset,unsigned long flags);
int (*_read) (struct mtd_info *mtd, loff_t from, size_t len,size_t *retlen, u_char *buf);   /*读函数*/
int (*_write) (struct mtd_info *mtd, loff_t to, size_t len,size_t *retlen, const u_char *buf);   /*写函数*/
int (*_panic_write) (struct mtd_info *mtd, loff_t to, size_t len,size_t *retlen, const u_char *buf);
int (*_read_oob) (struct mtd_info *mtd, loff_t from,struct mtd_oob_ops *ops);
int (*_write_oob) (struct mtd_info *mtd, loff_t to,struct mtd_oob_ops *ops);
int (*_get_fact_prot_info) (struct mtd_info *mtd, size_t len,size_t *retlen, struct otp_info *buf);
int (*_read_fact_prot_reg) (struct mtd_info *mtd, loff_t from,size_t len, size_t *retlen, u_char *buf);
int (*_get_user_prot_info) (struct mtd_info *mtd, size_t len,size_t *retlen, struct otp_info *buf);
int (*_read_user_prot_reg) (struct mtd_info *mtd, loff_t from,size_t len, size_t *retlen, u_char *buf);
int (*_write_user_prot_reg) (struct mtd_info *mtd, loff_t to,size_t len, size_t *retlen, u_char *buf);
int (*_lock_user_prot_reg) (struct mtd_info *mtd, loff_t from,size_t len);
int (*_writev) (struct mtd_info *mtd, const struct kvec *vecs,unsigned long count,
                                                        loff_t to, size_t *retlen);
void (*_sync) (struct mtd_info *mtd);     /*同步操作*/
int (*_lock) (struct mtd_info *mtd, loff_t ofs, uint64_t len);
int (*_unlock) (struct mtd_info *mtd, loff_t ofs, uint64_t len);
int (*_is_locked) (struct mtd_info *mtd, loff_t ofs, uint64_t len);
int (*_block_isreserved) (struct mtd_info *mtd, loff_t ofs);
int (*_block_isbad) (struct mtd_info *mtd, loff_t ofs);
int (*_block_markbad) (struct mtd_info *mtd, loff_t ofs);
int (*_suspend) (struct mtd_info *mtd);
void (*_resume) (struct mtd_info *mtd);
void (*_reboot) (struct mtd_info *mtd);
int (*_get_device) (struct mtd_info *mtd);
void (*_put_device) (struct mtd_info *mtd);

/*本设备作为后备存储设备信息 */
struct backing_dev_info *backing_dev_info;
```

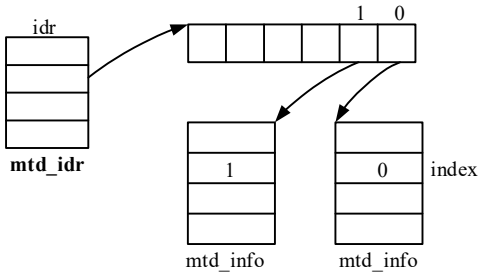
```
    struct notifier_block reboot_notifier;    /* default mode before reboot */

    /* ECC status information */
    struct mtd_ecc_stats ecc_stats;
    /* Subpage shift (NAND) */
    int subpage_sft;
    void *priv;            /*私有数据结构指针*/
    struct module *owner;
    struct device dev;     /*表示原始设备的 device 实例*/
    int usecount;
};
```

mtd_info 结构体中部分成员简介如下：

●**type**：类型，取值定义在/include/uapi/mtd/mtd-abi.h 头文件：

```
#define MTD_ABSENT          0
#define MTD_RAM             1
#define MTD_ROM             2
#define MTD_NORFLASH        3
#define MTD_NANDFLASH       4    /* SLC NAND */
#define MTD_DATAFLASH       6
#define MTD_UBIVOLUME       7
#define MTD_MLCNANDFLASH    8    /* MLC NAND (including TLC) */
```

●**index**：全局编号，内核中通过 idr 数据结构 mtd_idr 实例管理 mtd_info 实例，如下图所示，每个实例分配一个编号。



mtd_info 结构体中其它成员主要是 MTD 设备的硬件信息，以及底层操作函数指针。底层操作函数需由具体设备驱动程序实现，在 MTD 字符设备、块设备的操作函数中将调用这些底层操作函数，例如读写函数。

2 注册 MTD 原始设备

在注册 MTD 原始设备 mtd_info 实例时，需要向注册函数传递 MTD 设备的分区信息，这是与前面介绍的块设备驱动程序的不同之处。

下面先介绍如何传递和解析分区信息，然后再介绍注册 mtd_info 实例的函数。

■解析分区

内核提供了多种传递 MTD 原始设备分区信息的方式，如命令行参数、静态数组等，详见内核配置文

件/drivers/mtd/Kconfig。

MTD 原始设备分区信息由 mtd_partition 结构体表示，定义如下（/include/linux/mtd/partitions.h）：

```
struct mtd_partition {
    const char *name;       /*名称*/
    uint64_t size;          /*分区大小（字节数）*/
    uint64_t offset;        /*分区起始偏移量*/
    uint32_t mask_flags;        /*标记掩码*/
    struct nand_ecclayout *ecclayout;       /* out of band layout for this partition (NAND only) */
};
```

利用 mtd_partition 结构体数组之外的方式传递分区信息时，需要从传递的信息中解析出分区信息，并依此构建 mtd_partition 结构体数组。

对于其它传递分区信息的方式需要注册一个 **mtd_part_parser** 结构体实例，用于接收传递的信息进而解析并创建 mtd_partition 结构体数组，以便注册 mtd_info 实例时使用。

mtd_part_parser 结构体定义如下（/include/linux/mtd/partitions.h）：

```
struct mtd_part_parser {
    struct list_head list;    /*双链表成员，将实例链入全局双链表 part_parsers*/
    struct module *owner;
    const char *name;      /*名称，用于标志 mtd_part_parser 实例*/
    int (*parse_fn)(struct mtd_info *, struct mtd_partition **,struct mtd_part_parser_data *);
                                    /*解析函数，依传递信息创建 mtd_partition 结构体数组*/
};
```

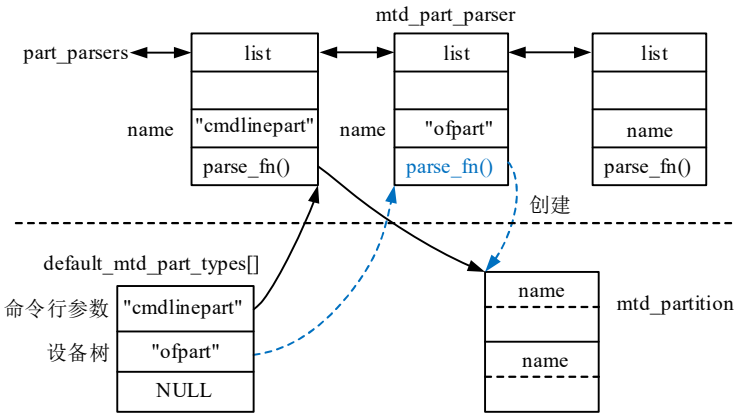
mtd_part_parser 结构体中 parse_fn()是解析信息生成 mtd_partition 结构体数组的函数指针，函数中第二个参数是 mtd_part_parser_data 结构体指针，表示传递给 parse_fn()解析函数的参数。

mtd_part_parser_data 结构体定义如下（/include/linux/mtd/partitions.h）：

```
struct mtd_part_parser_data {
    unsigned long origin;
    struct device_node *of_node;        /*设备树节点*/
};
```

用户可通过/drivers/mtd/Kconfig 文件中的配置选项，选择内核支持哪些传递分区信息的方式，每种方式对应一个/drivers/mtd/目录下的源文件，文件内将实现并注册相应的 mtd_part_parser 实例。例如：源文件/drivers/mtd/cmdlinepart.c 内定义并注册了解析命令行分区信息的 mtd_part_parser 实例。所有 mtd_part_parser 实例由全局双链表管理。

注册 MTD 原始设备时，需指定支持哪些传递分区信息的方式，如下图所示，default_mtd_part_types 字符串数组确定支持哪些传递分区信息方式（以名称标识）。在解析分区信息时将依字符串数组项，查找 mtd_part_parser 结构体实例数组，查找匹配的实例并调用解析函数创建分区 mtd_partition 数组，如果有一个实例的解析函数调用成功（创建了分区数组）将不再解析后面的字符串。



parse_mtd_partitions()函数用于解析 MTD 设备分区信息，函数定义如下（/drivers/mtd/mtdpart.c）：

```
int parse_mtd_partitions(struct mtd_info *master, const char *const *types,
                         struct mtd_partition **pparts,struct mtd_part_parser_data *data)
/*master：指向 mtd_info 实例，types：解析方式字符串数组，pparts：保存指向分区数组的指针*/
{
    struct mtd_part_parser *parser;
    int ret = 0;

    if (!types)          /*如果为 NULL*/
        types = default_mtd_part_types;       /*赋予默认的字符串数组，如上图所示*/

    for ( ; ret <= 0 && *types; types++) {        /*遍历字符串数组项*/
        parser = get_partition_parser(*types);
                    /*以名称在 part_parsers 双链表查找 mtd_part_parser 实例*/
        if (!parser && !request_module("%s", *types))
            parser = get_partition_parser(*types);
        if (!parser)
            continue;
        ret = (*parser->parse_fn)(master, pparts, data);       /*调用解析函数，返回分区数量*/
        put_partition_parser(parser);
        if (ret > 0) {       /*解析成功则跳出循环*/
            printk(KERN_NOTICE "%d %s partitions found on MTD device %s\n",
                   ret, parser->name, master->name);
            break;
        }
    }
    return ret;       /*返回分区数量*/
}
```

■注册函数

在板级代码中可以静态定义 **mtd_partition** 结构体数组，用于描述 MTD 设备分区信息，但是此种方式

的优先级最低，详见下面的注册函数。

在设备驱动程序中需要创建 MTD 原始设备 mtd_info 结构体实例，并调用 **mtd_device_parse_register()** 接口函数注册 MTD 原始设备，函数声明如下。

```
int mtd_device_parse_register(struct mtd_info *mtd,const char * const *part_probe_types,
        struct mtd_part_parser_data *parser_data,const struct mtd_partition *defparts,int defnr_parts);
```

参数 mtd 指向 mtd_info 实例，part_probe_types 指向解析分区方式字符串数组（优先级高于 defparts 参数传递的 mtd_partition 数组），parser_data 指向解析函数参数数据结构，defparts 指向静态定义的 mtd_partition 结构体数组，defnr_parts 表示 defparts 指向数组项数。

另外，内核在/include/linux/mtd/mtd.h 头文件还定义了 **mtd_device_register()**接口函数，如下：

```
#define mtd_device_register(master, parts, nr_parts) \
        mtd_device_parse_register(master, NULL, NULL, parts, nr_parts)
                /*采用默认的解析分区方式，parts 指向 mtd_partition 数组*/
```

下面看一下 mtd_device_parse_register()函数的定义，如下（/drivers/mtd/mtdcore.c）：

```
int mtd_device_parse_register(struct mtd_info *mtd, const char * const *types,
        struct mtd_part_parser_data *parser_data,const struct mtd_partition *parts,int nr_parts)
{
    int ret;
    struct mtd_partition *real_parts = NULL;

    ret = parse_mtd_partitions(mtd, types, &real_parts, parser_data);
                /*解析分区，real_prart 指向创建的分区数组，见上文，/drivers/mtd/mtdpart.c*/

    /*如果 parse_mtd_partitions()函数解析分区失败，采用 parts 传递的 mtd_partition 数组*/
    if (ret <= 0 && nr_parts && parts) {
        real_parts = kmemdup(parts, sizeof(*parts) * nr_parts,GFP_KERNEL);
                                        /*复制分区信息数组*/
        if (!real_parts)
            ret = -ENOMEM;
        else
            ret = nr_parts;     /*分区数量*/
    }

    if (ret >= 0)
        ret = mtd_add_device_partitions(mtd, real_parts, ret);
                /*创建分区字符设备文件，/drivers/mtd/mtdcore.c*/
    if (mtd->_reboot && !mtd->reboot_notifier.notifier_call) {
        mtd->reboot_notifier.notifier_call = mtd_reboot_notifier;
        register_reboot_notifier(&mtd->reboot_notifier);        /*系统重启时执行的通知*/
    }
```

```
    kfree(real_parts);
    return ret;
}
```

mtd_device_parse_register()函数的主要工作是解析 MTD 设备分区信息；创建分区对应的字符设备文件（此处还没有注册块设备驱动程序）；向重启通知链添加通知，mtd_reboot_notifier()为通知执行函数，函数内调用 mtd->_reboot(mtd)函数。

下面将介绍为分区创建字符设备文件的 mtd_add_device_partitions()函数。

●注册 MTD 设备

mtd_add_device_partitions()函数用于为分区创建字符设备文件，函数定义如下（/drivers/mtd/mtdcore.c）：

```
static int mtd_add_device_partitions(struct mtd_info *mtd,struct mtd_partition *real_parts,int nbparts)
{
    int ret;
    if (nbparts == 0 || IS_ENABLED(CONFIG_MTD_PARTITIONED_MASTER)) {     /*不支持分区*/
        ret = add_mtd_device(mtd);          /*注册 MTD 设备，见下文，/drivers/mtd/mtdcore.c*/
        if (ret)
            return ret;
    }

    if (nbparts > 0) {     /*MTD 设备进行了分区*/
        ret = add_mtd_partitions(mtd, real_parts, nbparts);     /*/drivers/mtd/mtdpart.c*/
        if (ret && IS_ENABLED(CONFIG_MTD_PARTITIONED_MASTER))
                                                    /*如果配置选择不支持分区*/
            del_mtd_device(mtd);
        return ret;
    }
    return 0;
}
```

如果支持 MTD 分区 mtd_add_device_partitions()函数将调用 **add_mtd_partitions()**函数添加分区。在 MTD 驱动框架内部分区由 mtd_part 结构体表示，定义如下（/drivers/mtd/mtdpart.c）：

```
struct mtd_part {
    struct mtd_info mtd;        /*表示分区的 mtd_info 实例*/
    struct mtd_info *master;     /*指向主 mtd_info 实例*/
    uint64_t offset;             /*分区起始偏移量*/
    struct list_head list;       /*双链表成员*/
};
static LIST_HEAD(mtd_partitions);     /*全局双链表，管理 mtd_part 实例*/
```

注意：分区 mtd_part 结构体中内嵌了 mtd_info 结构体成员，表示 MTD 原始设备。

添加 MTD 分区的 add_mtd_partitions()函数定义如下（/drivers/mtd/mtdpart.c）：

```
int add_mtd_partitions(struct mtd_info *master,const struct mtd_partition *parts,int nbparts)
```

```
/*master：指向表示整个 MTD 设备的 mtd_info 实例*/
{
    struct mtd_part *slave;
    uint64_t cur_offset = 0;
    int i;

    printk(KERN_NOTICE "Creating %d MTD partitions on \"%s\":\n", nbparts, master->name);

    for (i = 0; i < nbparts; i++) {     /*遍历 mtd_partition 数组*/
        slave = allocate_partition(master, parts + i, i, cur_offset);
                    /*创建并初始化 mtd_part 实例，复制 master 中信息，/drivers/mtd/mtdpart.c*/
        ...
        mutex_lock(&mtd_partitions_mutex);
        list_add(&slave->list, &mtd_partitions);     /*mtd_part 实例添加到全局双链表*/
        mutex_unlock(&mtd_partitions_mutex);

        add_mtd_device(&slave->mtd);
          /*注册分区内嵌 mtd_info 实例，设备文件名称为/dev/mtdi，/drivers/mtd/mtdcore.c*/
        mtd_add_partition_attrs(slave);
                        /*创建设备属性文件（slave->mtd->dev），/drivers/mtd/mtdpart.c*/

        cur_offset = slave->offset + slave->mtd.size;
    }
    return 0;
}
```

add_mtd_partitions()函数将遍历分区信息数组，为每个分区创建并初始化 mtd_part 实例，主要是初始化实例中 mtd_info 结构体成员；然后将 mtd_part 实例添加到全局双链表；调用 **add_mtd_device()**函数注册 mtd_part 实例内嵌的 mtd_info 结构体实例，并添加设备属性文件。

如果 MTD 设备没有分区或不支持分区，在 mtd_add_device_partitions()函数将调用 **add_mtd_device()** 函数注册表示整个 MTD 设备的 mtd_info 实例。如果 MTD 存在分区且支持分区，则只会注册分区中内嵌的 mtd_info 实例，而不会注册表示整个 MTD 设备的 mtd_info 实例。

下面简要看一下 add_mtd_device()函数的定义（/drivers/mtd/mtdcore.c）：

```
int add_mtd_device(struct mtd_info *mtd)
{
    struct mtd_notifier *not;
    int i, error;

    mtd->backing_dev_info = &mtd_bdi;        /*静态定义的 backing_dev_info 实例*/

    BUG_ON(mtd->writesize == 0);
    mutex_lock(&mtd_table_mutex);

    i = idr_alloc(&mtd_idr, mtd, 0, 0, GFP_KERNEL);        /*分配编号*/
```

```
    ...
    mtd->index = i;
    mtd->usecount = 0;
    ...
    /*解锁硬件设备*/
    if ((mtd->flags & MTD_WRITEABLE) && (mtd->flags & MTD_POWERUP_LOCK)) {
        error = mtd_unlock(mtd, 0, mtd->size);
        ...
        error = 0;
    }

    /*设置表示分区的 device 实例*/
    mtd->dev.type = &mtd_devtype;           /*设备类型，/drivers/mtd/mtdcore.c*/
    mtd->dev.class = &mtd_class;            /*设备类，/drivers/mtd/mtdcore.c*/
    mtd->dev.devt = MTD_DEVT(i);      /*主设备号 MTD_CHAR_MAJOR，从设备号（2*i）*/
    dev_set_name(&mtd->dev, "mtd%d", i);        /*device 名称，设备文件名称/dev/mtdi*/
    dev_set_drvdata(&mtd->dev, mtd);
    error = device_register(&mtd->dev);         /*注册 device 实例*/
    if (error)
        goto fail_added;

    device_create(&mtd_class, mtd->dev.parent, MTD_DEVT(i) + 1, NULL,"mtd%dro", i);
                        /*创建并注册 devcie 实例，设备文件名称/dev/mtdiro*/

    ...
    list_for_each_entry(not, &mtd_notifiers, list)      /*执行 mtd_notifier 通知链中通知*/
        not->add(mtd);                  /*添加 MTD 块设备驱动程序等，见下文*/

    mutex_unlock(&mtd_table_mutex);
    __module_get(THIS_MODULE);
    return 0;
    ...
}
```

add_mtd_device()函数完成的主要工作如下：

（1）为 mtd_info 实例分配全局编号 i。

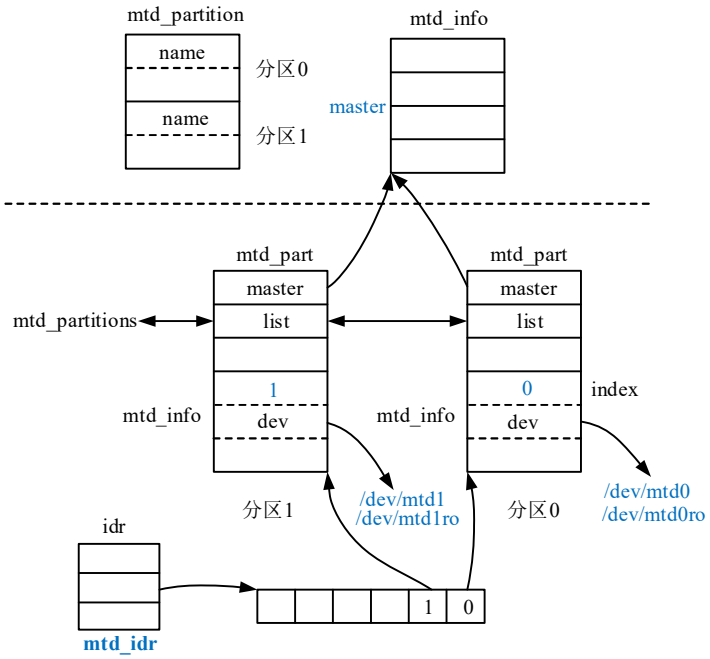
（2）设置 mtd_info 实例，包括内嵌的 device 实例。

（3）添加 mtd_info->dev 成员（devcie 实例），创建设备文件/dev/mtd*i*（字符设备文件）。

（4）创建添加 devcie 实例，创建设备文件/dev/mtd*i*ro（字符设备文件）。

（5）执行执行 mtd_notifier 通知链中通知，包括添加 MTD 块设备驱动程序等。

注册 mtd_info 实例结果如下图所示：



如果 MTD 设备不区，则注册表示 MTD 原始设备的 mtd_info 实例，否则注册每个分区内嵌的 mtd_info 实例。内核为每个 mtd_info 实例分配唯一的编号 i，并创建字符设备文件/dev/mtd*i* 和/dev/mtd*i*ro，下一小节将介绍 MTD 字符设备文件的操作函数。

MTD 块设备驱动程序的注册隐含在 mtd_notifier 通知链中，后面将介绍。

10.9.3 MTD 字符设备驱动

在前面介绍的注册 mtd_info 实例的函数中将为分区创建字符设备文件，下面将介绍此字符设备驱动程序的实现，驱动程序在/drivers/mtd/mtdchar.c 文件内实现。

MTD 字符设备驱动程序初始化函数定义如下（/drivers/mtd/mtdchar.c）：

```
int __init init_mtdchar(void)          /*由 init_mtd()函数调用*/
{
    int ret;

    ret = __register_chrdev(MTD_CHAR_MAJOR, 0, 1 << MINORBITS,"mtd", &mtd_fops);
                                /*创建并添加字符设备驱动程序 cdev 实例，适用于所有从设备*/
    ...
    return ret;
}
```

__register_chrdev()函数内将申请 MTD 字符设备号，创建并添加 cdev 实例，其关联的文件操作结构实例为 **mtd_fops**。

MTD 字符设备文件操作结构实例 mtd_fops 定义如下：

```
static const struct file_operations mtd_fops = {
    .owner     = THIS_MODULE,
    .llseek    = mtdchar_lseek,
    .read      = mtdchar_read,      /*读操作*/
    .write     = mtdchar_write,     /*写操作*/
    .unlocked_ioctl= mtdchar_unlocked_ioctl,    /*设备控制，擦除设备等*/
```

```
#ifdef CONFIG_COMPAT
    .compat_ioctl  = mtdchar_compat_ioctl,
#endif
    .open       = mtdchar_open,     /*打开函数*/
    .release    = mtdchar_close,
    .mmap     = mtdchar_mmap,
#ifndef CONFIG_MMU
    .get_unmapped_area = mtdchar_get_unmapped_area,
    .mmap_capabilities = mtdchar_mmap_capabilities,
#endif
};
```

下面简要介绍一下 MTD 字符设备文件打开操作、读/写操作和设备控制等函数的实现。

1 打开操作

MTD 字符设备文件打开操作函数 mtdchar_open()简列如下：

```
static int mtdchar_open(struct inode *inode, struct file *file)
{
    int minor = iminor(inode);     /*从设备号*/
    int devnum = minor >> 1;     /*mtd_info 编号*/
    int ret = 0;
    struct mtd_info *mtd;
    struct mtd_file_info *mfi;
    ...
    /*只读设备不能以读写模式打开设备文件*/
    if ((file->f_mode & FMODE_WRITE) && (minor & 1))
        return -EACCES;

    mutex_lock(&mtd_mutex);
    mtd = get_mtd_device(NULL, devnum);/*查找 mtd_info 实例，调用 mtd->_get_device(mtd)函数等*/
    ...
    /*不可写设备，不能以读写模式打开*/
    if ((file->f_mode & FMODE_WRITE) && !(mtd->flags & MTD_WRITEABLE)) {
        ...
    }

    mfi = kzalloc(sizeof(*mfi), GFP_KERNEL);     /*分配 mtd_file_info 结构体实例*/
    ...
    mfi->mtd = mtd;              /*指向 mtd_info 实例*/
    file->private_data = mfi;
    mutex_unlock(&mtd_mutex);
    return 0;
    ...
}
```

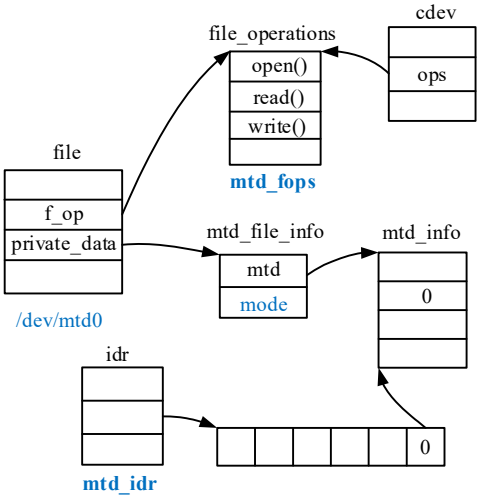
mtdchar_open()函数中将创建 mtd_file_info 结构体实例，结构体定义如下（/drivers/mtd/mtdchar.c）：

```
struct mtd_file_info {
    struct mtd_info *mtd;
    enum mtd_file_modes mode;  /*文件模式，通过 MTDFILEMODE 命令设置（ioctl()系统调用）*/
};
```

mode 成员为 mtd_file_modes 枚举类型，定义如下（/include/uapi/mtd/mtd-abi.h）：

```
enum mtd_file_modes {
    MTD_FILE_MODE_NORMAL = MTD_OTP_OFF,        /*OTP disabled, ECC enabled，初始模式*/
    MTD_FILE_MODE_OTP_FACTORY = MTD_OTP_FACTORY,    /*OTP enabled in factory mode*/
    MTD_FILE_MODE_OTP_USER = MTD_OTP_USER,        /*OTP enabled in user mode*/
    MTD_FILE_MODE_RAW,                            /*OTP disabled, ECC disabled*/
};
```

mtdchar_open()函数函数执行结果如下图所示（由编号查找 mtd_info 实例）：



2 读/写操作

MTD 字符设备文件读操作函数 mtdchar_read()定义如下：

```
static ssize_t mtdchar_read(struct file *file, char __user *buf, size_t count,loff_t *ppos)
{
    struct mtd_file_info *mfi = file->private_data;
    struct mtd_info *mtd = mfi->mtd;
    size_t retlen;
    size_t total_retlen=0;
    int ret=0;
    int len;
    size_t size = count;
    char *kbuf;

    ...
    if (*ppos + count > mtd->size)
        count = mtd->size - *ppos;
```

```
...
kbuf = mtd_kmalloc_up_to(mtd, &size);       /*分配缓存区*/
...
while (count) {
    len = min_t(size_t, count, size);

    switch (mfi->mode) {   /*不同访问模式调用不同的读函数*/
    case MTD_FILE_MODE_OTP_FACTORY:
        ret = mtd_read_fact_prot_reg(mtd, *ppos, len,&retlen, kbuf);
        break;                  /*调用 mtd->_read_fact_prot_reg()函数，/drivers/mtd/mtdcore.c*/
    case MTD_FILE_MODE_OTP_USER:
        ret = mtd_read_user_prot_reg(mtd, *ppos, len,&retlen, kbuf);
        break;                  /*调用 mtd->_read_user_prot_reg()函数，/drivers/mtd/mtdcore.c*/
    case MTD_FILE_MODE_RAW:    /*读原始设备*/
    {
        struct mtd_oob_ops ops;

        ops.mode = MTD_OPS_RAW;
        ops.datbuf = kbuf;
        ops.oobbuf = NULL;
        ops.len = len;

        ret = mtd_read_oob(mtd, *ppos, &ops);          /*调用 mtd->_read_oob()函数*/
        retlen = ops.retlen;
        break;
    }
    default:        /*初始（默认的模式）*/
        ret = mtd_read(mtd, *ppos, len, &retlen, kbuf);     /*调用 mtd_info->_read()函数*/
    }
    if (!ret || mtd_is_bitflip_or_eccerr(ret)) {
        *ppos += retlen;
        if (copy_to_user(buf, kbuf, retlen)) {
            ...
        }
        else
            total_retlen += retlen;

        count -= retlen;
        buf += retlen;
        if (retlen == 0)
            count = 0;
    }
```

```
            else {
                kfree(kbuf);
                return ret;
            }

        }

        kfree(kbuf);
        return total_retlen;
    }
```

MTD 字符设备文件读操作函数根据不同的模式调用 mtd_info 实例中的函数完成读操作。

MTD 字符设备文件写操作函数 mtdchar_write()与读操作函数类似，源代码读者自行阅读。

3 设备控制

对 MTD 字符设备文件执行 ioctl()系统调用时，对于 MTD 设备命令，将调用文件操作结构 mtd_fops 实例中的 unlocked_ioctl()函数，即 **mtdchar_unlocked_ioctl()**函数进行处理。

MTD 字符设备命令及命令参数相关数据结构定义在/include/uapi/mtd/mtd-abi.h 头文件，例如：

```
#define    MEMGETINFO          _IOR('M', 1, struct mtd_info_user)     /*获取设备信息*/
#define    MEMERASE            _IOW('M', 2, struct erase_info_user)   /*擦除设备*/
...
#define    MTDFILEMODE         _IO('M', 19)     /*设置字符设备访问模式*/
...

struct mtd_info_user {
    __u8 type;
    __u32 flags;
    __u32 size;     /* Total size of the MTD */
    __u32 erasesize;
    __u32 writesize;
    __u32 oobsize;      /* Amount of OOB data per block (e.g. 16) */
    __u64 padding;      /* Old obsolete field; do not use */
};
...
```

mtdchar_unlocked_ioctl()函数内根据不同的命令调用不同的处理函数，源代码请读者自行阅读。

10.9.4 MTD 块设备驱动

MTD 框架中将 MTD 设备视为字符设备和块设备。MTD 字符设备驱动前面介绍了，但是似乎还没有看到 MTD 块设备驱动。MTD 块设备驱动程序隐藏在注册 mtd_info 实例时执行的 MTD 通知链中，下面将详细介绍。

1 概述

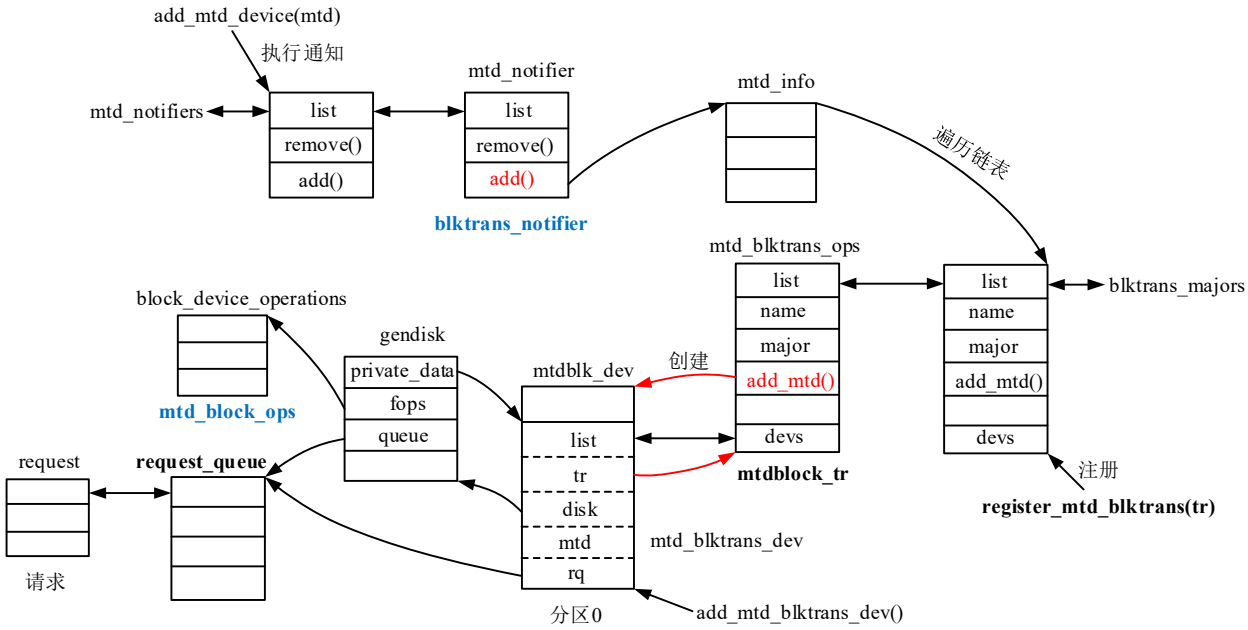
注册 MTD 块设备驱动程序的流程如下图所示。在前面介绍的注册 mtd_info 实例的函数中，将遍历双链表 mtd_notifiers，调用链表成员中的 add()函数。mtd_notifiers 双链表管理的 mtd_notifier 结构体实例，内核向双链表注册了 **blktrans_notifier** 实例。

blktrans_notifier 实例的 add()函数将对 mtd_info 实例，遍历 mtd_blktrans_ops 结构体实例双链表，对每个实例调用其中的 **add_mtd()**函数，在 add_mtd()函数中将为 mtd_info 实例创建并注册块设备驱动程序。

mtd_blktrans_ops 结构体表示访问 MTD 块设备的一种模式，这里暂称为传输模式。用户可通过配置选项选择支持的 MTD 块设备传输模式。每个传输模式的代码将定义并注册一个 mtd_blktrans_ops 实例，所有实例由全局双链表管理。

mtd_blktrans_ops 结构体中的 **add_mtd()**函数，将为 mtd_info 实例创建并注册对应的 mtd_blktrans_dev 结构体实例，以及块设备驱动的 gendisk、request_queue（标准请求队列）等实例。

请求的处理将调用 mtd_blktrans_ops 实例中的函数，进而调用 mtd_info 实例中的函数，实现底层的数据传输。



■ MTD 通知链

在上面的框图中包含一个 mtd_notifiers 双链表（MTD 通知链），链表成员为 mtd_notifier 结构体实例，表示 MTD 通知，mtd_notifier 结构体定义如下（/include/linux/mtd/mtd.h）：

```
struct mtd_notifier {
    void (*add)(struct mtd_info *mtd);          /*添加 mtd_info 实例时调用的函数*/
    void (*remove)(struct mtd_info *mtd);       /*移除 mtd_ifno 实例时调用的函数*/
    struct list_head list;          /*添加到 mtd_notifiers 双链表*/
};
```

add()表示添加 mtd_info 实例时调用的函数，remove()表示移除 mtd_info 实例时调用的函数。

mtd_notifiers 双链表定义在/drivers/mtd/mtdcore.c 文件内：

```
static LIST_HEAD(mtd_notifiers)
```

向 mtd_notifiers 双链表注册 mtd_notifier 实例的函数定义如下（/drivers/mtd/mtdcore.c）：

```
void register_mtd_user (struct mtd_notifier *new)
{
    struct mtd_info *mtd;

    mutex_lock(&mtd_table_mutex);
    list_add(&new->list, &mtd_notifiers);      /*实例添加到全局双链表头部*/
    __module_get(THIS_MODULE);

    mtd_for_each_device(mtd)      /*遍历所有 mtd_info 实例*/
        new->add(mtd);          /*对 mtd_info 实例调用 mtd_notifier.add()函数*/

    mutex_unlock(&mtd_table_mutex);
}
```

在向 mtd_notifiers 双链表注册新 mtd_notifier 实例，将遍历内核中已有 mtd_info 实例，对于调用实例中的 add()函数。

删除 mtd_notifier 实例的函数为 int unregister_mtd_user (struct mtd_notifier *old)，函数中对内核中所有 mtd_info 实例调用其中的 remove()函数，并将 mtd_info 实例从双链表中移除。

内核在/drivers/mtd/mtd_blkdevs.c 文件内定义了 mtd_notifier 实例 **blktrans_notifier**：

```
static struct mtd_notifier blktrans_notifier = {
    .add = blktrans_notify_add,              /*添加 mtd_info 实例时调用的函数*/
    .remove = blktrans_notify_remove,
};
```

blktrans_notifier 实例中 add()函数为 blktrans_notify_add()，定义如下：

```
static void blktrans_notify_add(struct mtd_info *mtd)
{
    struct mtd_blktrans_ops *tr;

    if (mtd->type == MTD_ABSENT)
        return;

    list_for_each_entry(tr, &blktrans_majors, list)     /*遍历 mtd_blktrans_ops 双链表*/
        tr->add_mtd(tr, mtd); /*对 mtd_info 实例调用 mtd_blktrans_ops 实例中的 add_mtd()函数*/
}
```

blktrans_notify_add()函数将遍历 blktrans_majors 双链表中的 mtd_blktrans_ops 实例，对 mtd_info 实例调用各实例中的 **add_mtd()**函数，在 add_mtd()函数中将为 mtd_info 实例注册块设备驱动程序。也就是说内核中有多少个 mtd_blktrans_ops 实例，每个 mtd_info 实例就对应多少个块设备驱动程序。

2 传输模式

MTD 块设备的传输模式决定了 MTD 块设备对应的块设备驱动程序。下面看一下传输模式相关数据结

构的定义，以及传输模式的注册。

■数据结构

mtd_blktrans_ops 结构体表示 MTD 块设备传输模式，结构体定义如下（/include/linux/mtd/blktrans.h）：

```
struct mtd_blktrans_ops {
    char *name;        /*名称，用于块设备文件名称*/
    int major;         /*主设备号，不同传输模式下 MTD 块设备主设备号不同*/
    int part_bits;     /**/
    int blksize;       /*块大小*/
    int blkshift;

    /*块设备访问函数，处理请求时调用这些函数，这些函数进而调用 mtd_info 实例中函数*/
    int (*readsect)(struct mtd_blktrans_dev *dev,unsigned long block, char *buffer);     /*读扇区*/
    int (*writesect)(struct mtd_blktrans_dev *dev,unsigned long block, char *buffer);    /*写扇区*/
    int (*discard)(struct mtd_blktrans_dev *dev,unsigned long block, unsigned nr_blocks);
    void (*background)(struct mtd_blktrans_dev *dev);

    /*块设备层控制函数*/
    int (*getgeo)(struct mtd_blktrans_dev *dev, struct hd_geometry *geo);
    int (*flush)(struct mtd_blktrans_dev *dev);

    int (*open)(struct mtd_blktrans_dev *dev);     /*打开操作*/
    void (*release)(struct mtd_blktrans_dev *dev);
    void (*add_mtd)(struct mtd_blktrans_ops *tr, struct mtd_info *mtd);   /*添加 mtd_info 实例*/
    void (*remove_dev)(struct mtd_blktrans_dev *dev);

    struct list_head devs;     /*双链表头，管理 mtd_blktrans_dev 实例，见下文*/
    struct list_head list;     /*将实例添加到 blktrans_majors 全局双链表*/
    struct module *owner;
};
```

mtd_blktrans_ops 结构体中的成员比较简单，上面都加了注释，就不再解释了。

mtd_blktrans_ops 结构体中 add_mtd()函数内将为 mtd_info 实例创建 mtdblk_dev 结构体实例，结构体定义如下（/drivers/mtd/mtdblock.c）：

```
struct mtdblk_dev {
    struct mtd_blktrans_dev mbd;          /*mtd_blktrans_dev 结构体成员，见下文*/
    int count;
    struct mutex cache_mutex;
    unsigned char *cache_data;     /*缓存区指针*/
    unsigned long cache_offset;    /*当前偏移量*/
    unsigned int cache_size;       /*缓存区大小*/
    enum { STATE_EMPTY, STATE_CLEAN, STATE_DIRTY } cache_state;   /*缓存区状态*/
```

```
};
```

mtdblk_dev 结构体中内嵌了 mtd_blktrans_dev 结构体成员 mbd，关联了块设备驱动程序相关的数据结构实例，结构体定义如下（/include/linux/mtd/blktrans.h）：

```
struct mtd_blktrans_dev {
    struct mtd_blktrans_ops *tr;        /*指向 mtd_blktrans_ops 实例*/
    struct list_head list;        /*双链表成员，将实例添加到 mtd_blktrans_ops.devs 双链表*/
    struct mtd_info *mtd;      /*指向 mtd_info 实例*/
    struct mutex lock;
    int devnum;                 /*mtd_info 实例编号*/
    bool bg_stop;
    unsigned long size;
    int readonly;
    int open;                   /*是否已经打开*/
    struct kref ref;
    struct gendisk *disk;      /*指向 gendisk 实例*/
    struct attribute_group *disk_attributes;
    struct workqueue_struct *wq;        /*工作队列*/
    struct work_struct work;          /*工作，用于处理队列中请求*/
    struct request_queue *rq;          /*指向标准请求队列*/
    spinlock_t queue_lock;
    void *priv;
    fmode_t file_mode;      /*块设备文件访问模式*/
};
```

■注册传输模式

MTD 块设备传输模式公共代码位于/drivers/mtd/mtd_blkdevs.c 文件内。注册 mtd_blktrans_ops 实例的接口函数 **register_mtd_blktrans()**定义如下：

```
int register_mtd_blktrans(struct mtd_blktrans_ops *tr)
{
    struct mtd_info *mtd;
    int ret;

    if (!blktrans_notifier.list.next)       /*如果 blktrans_notifier 实例（mtd_notifier 结构体）尚未注册*/
        register_mtd_user(&blktrans_notifier);          /*注册实例，/drivers/mtd/mtdcore.c*/

    mutex_lock(&mtd_table_mutex);

    ret = register_blkdev(tr->major, tr->name);   /*注册块设备主设备号*/
    ...
    if (ret)
        tr->major = ret;      /*主设备号*/
```

```
    tr->blkshift = ffs(tr->blksize) - 1;     /*块大小*/

    INIT_LIST_HEAD(&tr->devs);
    list_add(&tr->list, &blktrans_majors);   /*mtd_blktrans_ops 实例添加到全局双链表头部*/

    mtd_for_each_device(mtd)
                        /*遍历所有 mtd_info 实例调用 mtd_blktrans_ops 实例中的 add_mtd()函数*/
        if (mtd->type != MTD_ABSENT)
            tr->add_mtd(tr, mtd);

    mutex_unlock(&mtd_table_mutex);
    return 0;
}
```

register_mtd_blktrans()函数首先注册了 tr->major 中指定的块设备主设备号，然后将 mtd_blktrans_ops 实例添加到全局双链表头部，最后对现有所有 mtd_info 实例调用其中的 add_mtd()函数，注册块设备驱动程序。

3 注册块设备驱动程序

在/drivers/mtd/Kconfig 配置文件内，用户可选择支持的 MTD 块设备传输模式，从而将选择编译不同的源文件，在各源文件中将定义并注册相应的 mtd_blktrans_ops 实例。

例如：如果选择了 MTD_BLOCK 选项，对 MTD 块设备的访问将在内存中建立缓存，对应的源文件为/drivers/mtd/mtdblock.c，在此文件内将定义并注册了 mtd_blktrans_ops 结构体实例 **mtdblock_tr**，如下：

```
static struct mtd_blktrans_ops mtdblock_tr = {
    .name     = "mtdblock",        /*用于块设备文件名*/
    .major    = MTD_BLOCK_MAJOR,   /*主设备号*/
    .part_bits = 0,                /*确定块设备驱动程序支持的分区数量，为 0 表示不分区*/
    .blksize  = 512,               /*块大小，512 字节*/
    .open     = mtdblock_open,     /*打开操作*/
    .flush    = mtdblock_flush,    /*刷出缓存*/
    .release  = mtdblock_release,
    .readsect = mtdblock_readsect, /*读扇区，处理请求时调用*/
    .writesect= mtdblock_writesect,/*写扇区，处理请求时调用*/
    .add_mtd      = mtdblock_add_mtd,   /*创建 mtd_blktrans_dev 实例等*/
    .remove_dev   = mtdblock_remove_dev,
    .owner        = THIS_MODULE,
};
```

在初始化（加载模块）函数中将注册 mtdblock_tr 实例：

```
static int __init init_mtdblock(void)
{
    return register_mtd_blktrans(&mtdblock_tr);
}
```

```
module_init(init_mtdblock);
```

mtdblock_tr 实例中 **add_mtd()**函数为 mtdblock_add_mtd()，定义如下（/drivers/mtd/mtdblock.c）：

```
static void mtdblock_add_mtd(struct mtd_blktrans_ops *tr, struct mtd_info *mtd)
{
    struct mtdblk_dev *dev = kzalloc(sizeof(*dev), GFP_KERNEL);     /*创建 mtdblk_dev 实例*/
    ...

    dev->mbd.mtd = mtd;                 /*mtd_info 实例*/
    dev->mbd.devnum = mtd->index;   /*mtd_info 编号*/

    dev->mbd.size = mtd->size >> 9;     /*扇区数*/
    dev->mbd.tr = tr;   /*指向 mtd_blktrans_ops 实例*/

    if (!(mtd->flags & MTD_WRITEABLE))
        dev->mbd.readonly = 1;

    if (add_mtd_blktrans_dev(&dev->mbd))   /*注册块设备驱动程序等，通用函数，见下文*/
        kfree(dev);
}
```

mtdblock_add_mtd()函数中为 mtd_info 实例创建 mtdblk_dev 实例，调用 **add_mtd_blktrans_dev()**函数设置 mtdblk_dev 实例中的 mtd_blktrans_dev 结构体成员，注册块设备驱动程序等，见下文。

■注册驱动函数

add_mtd_blktrans_dev()是一个通用的接口函数，各传输模式的 add_mtd()函数中都会调用这个函数，为 mtd_info 实例注册块设备驱动程序，函数定义如下（/drivers/mtd/mtd_blkdevs.c）：

```
int add_mtd_blktrans_dev(struct mtd_blktrans_dev *new)
{
    struct mtd_blktrans_ops *tr = new->tr;
    struct mtd_blktrans_dev *d;
    int last_devnum = -1;
    struct gendisk *gd;
    int ret;
    ...
    mutex_lock(&blktrans_ref_mutex);
    list_for_each_entry(d, &tr->devs, list) { /*遍历 mtd_blktrans_ops 实例 devs 双链表，查找插入位置*/
        if (new->devnum == -1) {                /*如查 mtd_info 编号为-1*/
            if (d->devnum != last_devnum+1) {
                new->devnum = last_devnum+1;        /*设置编号值*/
                list_add_tail(&new->list, &d->list);     /*插入 devs 双链表末尾*/
                goto added;
```

```
            }
        } else if (d->devnum == new->devnum) {
            mutex_unlock(&blktrans_ref_mutex);
            return -EBUSY;
        } else if (d->devnum > new->devnum) {      /*实例在双链表中按 devnum 值从小到大排列*/
            list_add_tail(&new->list, &d->list);
            goto added;        /*跳转至 added 处*/
        }
        last_devnum = d->devnum;
    }      /*遍历 mtd_blktrans_ops 实例 devs 链表结束*/

    ret = -EBUSY;
    if (new->devnum == -1)
        new->devnum = last_devnum+1;

    if (new->devnum > (MINORMASK >> tr->part_bits) ||(tr->part_bits && new->devnum >= 27 * 26)) {
        mutex_unlock(&blktrans_ref_mutex);
        goto error1;
    }
    list_add_tail(&new->list, &tr->devs);
    /*以上代码为 mtd_blktrans_dev 实例赋予一个编号，并将其插入 devs 双链表*/

added:
    mutex_unlock(&blktrans_ref_mutex);
    mutex_init(&new->lock);
    kref_init(&new->ref);
    if (!tr->writesect)
        new->readonly = 1;

    ret = -ENOMEM;
    gd = alloc_disk(1 << tr->part_bits);     /*分配 gendisk 实例，不分区*/
    ...
    new->disk = gd;                  /*指向 gendisk 实例*/
    gd->private_data = new;     /*指向 mtd_blktrans_dev 实例*/
    gd->major = tr->major;     /*主设备号*/
    gd->first_minor = (new->devnum) << tr->part_bits;       /*起始从设备号，就为 devnum*/
    gd->fops = &mtd_block_ops;       /*块设备操作结构，/drivers/mtd/mtd_blkdevs.c*/

    if (tr->part_bits)     /*设置 gendisk 名称，用于块设备文件名*/
        if (new->devnum < 26)
            snprintf(gd->disk_name, sizeof(gd->disk_name),"%s%c", tr->name, 'a' + new->devnum);
        else
            snprintf(gd->disk_name, sizeof(gd->disk_name),"%s%c%c", tr->name,
```

```
                    'a' - 1 + new->devnum / 26,'a' + new->devnum % 26);
else        /*mtdblock_tr 实例 tr->part_bits 为 0*/
    snprintf(gd->disk_name, sizeof(gd->disk_name),"%s%d", tr->name, new->devnum);
                                        /*块设备文件名为/dev/tr->namen（n 为 mtd_info 编号）*/

set_capacity(gd, (new->size * tr->blksize) >> 9);     /*设置磁盘容量*/

/*创建请求队列*/
spin_lock_init(&new->queue_lock);
new->rq = blk_init_queue(mtd_blktrans_request, &new->queue_lock);       /*标准请求队列*/
...                                                 /*处理请求函数为 mtd_blktrans_request()*/

if (tr->flush)
    blk_queue_flush(new->rq, REQ_FLUSH);

new->rq->queuedata = new;         /*标准请求队列关联 mtd_blktrans_dev 实例*/
blk_queue_logical_block_size(new->rq, tr->blksize);

queue_flag_set_unlocked(QUEUE_FLAG_NONROT, new->rq);
queue_flag_clear_unlocked(QUEUE_FLAG_ADD_RANDOM, new->rq);

if (tr->discard) {
    queue_flag_set_unlocked(QUEUE_FLAG_DISCARD, new->rq);
    new->rq->limits.max_discard_sectors = UINT_MAX;
}

gd->queue = new->rq;     /*标准请求队列*/

/*创建工作队列，用于处理请求*/
new->wq = alloc_workqueue("%s%d", 0, 0,tr->name, new->mtd->index);
...
INIT_WORK(&new->work, mtd_blktrans_work);       /*初始化工作*/
                                    /*工作执行函数调用 mtd_blktrans_ops 实例中函数处理请求*/
gd->driverfs_dev = &new->mtd->dev;

if (new->readonly)
    set_disk_ro(gd, 1);

add_disk(gd);       /*添加磁盘*/

if (new->disk_attributes) {
    ret = sysfs_create_group(&disk_to_dev(gd)->kobj,new->disk_attributes);     /*创建属性文件*/
    WARN_ON(ret);
```

```
    }
    return 0;
    ...
}
```

add_mtd_blktrans_dev()函数读者理解起来应该没有什么难度了。这里要注意的是函数中创建了一个工作队列赋予 mtd_blktrans_dev 实例，mtd_blktrans_dev 实例中 work 工作的执行函数为 **mtd_blktrans_work()**。创建的标准请求队列的请求处理函数为 **mtd_blktrans_request()**。

mtd_blktrans_request()函数的主要工作就是将 work 工作添加到工作队列中，在 mtd_blktrans_work()函数中将执行请求的处理，详见下文。

■请求处理

MTD 块设备驱动程序中标准请求队列的请求处理函数为 mtd_blktrans_request()，定义如下：

```
static void mtd_blktrans_request(struct request_queue *rq)          /*/drivers/mtd/mtd_blkdevs.c*/
{
    struct mtd_blktrans_dev *dev;
    struct request *req = NULL;

    dev = rq->queuedata;          /*mtd_blktrans_dev 实例*/

    if (!dev)          /*如果没有关联 mtd_blktrans_dev 实例了，结束所有请求*/
        while ((req = blk_fetch_request(rq)) != NULL)
            __blk_end_request_all(req, -ENODEV);
    else
        queue_work(dev->wq, &dev->work);        /*正常情况下就是将工作添加到工作队列*/
}
```

mtd_blktrans_request()函数正常情况下就是将 dev->work 工作添加到 dev->wq 工作队列。

下面看一下 dev->work 工作执行函数 **mtd_blktrans_work()**的定义（/drivers/mtd/mtd_blkdevs.c）：

```
static void mtd_blktrans_work(struct work_struct *work)
{
    struct mtd_blktrans_dev *dev =
        container_of(work, struct mtd_blktrans_dev, work);
    struct mtd_blktrans_ops *tr = dev->tr;
    struct request_queue *rq = dev->rq;
    struct request *req = NULL;
    int background_done = 0;

    spin_lock_irq(rq->queue_lock);          /*持有队列锁*/

    while (1) {                 /*无限循环*/
        int res;
```

```
        dev->bg_stop = false;
        if (!req && !(req = blk_fetch_request(rq))) {          /*取请求*/
            if (tr->background && !background_done) {
                spin_unlock_irq(rq->queue_lock);
                mutex_lock(&dev->lock);
                tr->background(dev);              /*mtd_blktrans_ops 实例中函数*/
                mutex_unlock(&dev->lock);
                spin_lock_irq(rq->queue_lock);
                background_done = !dev->bg_stop;
                continue;
            }
            break;
        }

        spin_unlock_irq(rq->queue_lock);          /*释放队列锁*/

        mutex_lock(&dev->lock);
        res = do_blktrans_request(dev->tr, dev, req);        /*执行请求，/drivers/mtd/mtd_blkdevs.c*/
        mutex_unlock(&dev->lock);

        spin_lock_irq(rq->queue_lock);            /*持有队列锁*/

        if (!__blk_end_request_cur(req, res))        /*结束请求*/
            req = NULL;

        background_done = 0;
    }        /*无限循环结束*/

    spin_unlock_irq(rq->queue_lock);          /*释放队列锁*/
}
```

mtd_blktrans_work()函数内是一个无限循环，循环体内从请求队列中取请求，调用 **do_blktrans_request**()函数执行请求，最后结束请求，如此循环。

do_blktrans_request()函数内将调用 mtd_blktrans_ops 实例中的 readsect()、writesect()等函数执行请求中的数据传输，而这些函数最终将调用 mtd_info 实例中的相应函数执行底层的数据传输，相关源代码请读者自行阅读。

至此，MTD 设备驱动程序框架就介绍完了，下一小节将以 NAND Flash 为例，说明具体驱动程序的实现。

10.9.5 NAND Flash 驱动

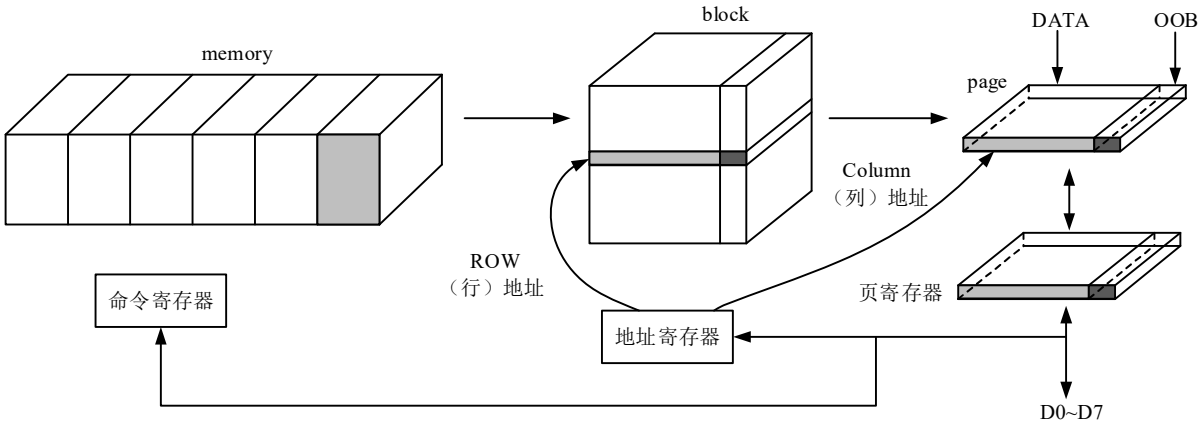
NAND Flash 是非易性的存储芯片（闪存），常在嵌入式系统性作外部存储器，用来存放固件、内核、文件系统等。本小节先简要介绍 NAND Flash 的硬件原理，然后介绍 NAND Flash 驱动框架，最后用示例的方式简要说明驱动程序的实现。

1 硬件原理

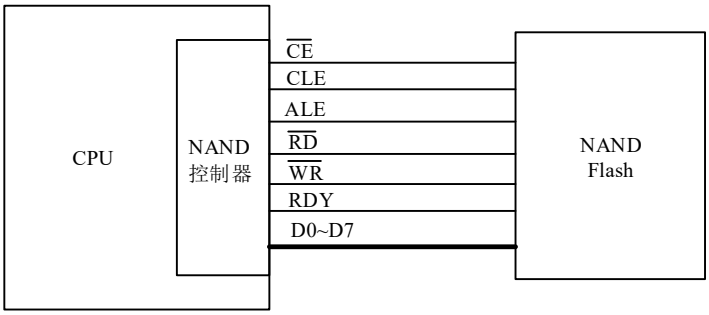
NAND Flash 由多个 block（块）组成，每个块中包含多个页，如下图所示。每个页（page）中包含数据区（DATA）和空闲区（OOB，out of band），OOB 区用于存放额外的数据，例如 ECC（数据校验值）。

NAND Flash 存储容量的计算：假设每个 page 中数据区为 2KB，OOB 区为 64B，每个块 block 有 64 页，芯片共有 1024 个 block，则存储容量为：

1 Device = (2K+64)B x 64Pages x 1,024 Blocks= 132MB=1,056 Mbits



NAND Flash 芯片与处理器连接的接口如下图所示：



NAND Flash 没有专门的地址总线和数据总线，地址和数据都通过 D0~D7 引脚传输，各引脚定义如下表所示：

引脚名称	描述
CE\	片选信号，低电平有效。
CLE	命令锁存信号，高电平时，在 WR\信号上升沿锁存 D0~D7 命令值到命令寄存器。
ALE	地址锁存信号，高电平时，在 WR\信号上升沿锁存 D0~D7 地址值到地址寄存器。
WR\	写控制信号，芯片在上升沿从 D0~D7 采集地址、命令、数据值。
RD\	读控制信号，芯片在下降沿从 D0~D7 输出数据，内部页内地址加 1。
RDY	输出信号，显示芯片状态，低电平表示忙，操作未完成，高电平表示操作完成。
D0~D7	输入输出数据，包括命令、地址、数据。

NAND Flash 内有命令寄存器、地址寄存器和页寄存器，分别用于缓存命令、地址和数据。NAND Flash 在写之前需要进行擦除操作（全写 1），擦除操作需按块（block）进行，写操作需按页进行，读操作可随机进行。

NAND Flash 通过 D0~D7 传输命令、地址和数据，其中命令和数据都是按字节传输，地址需要 4 个字节表示，如下表所示：

	D0	D1	D2	D3	D4	D5	D6	D7
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7
2nd Cycle	A8	A9	A10	A11	*L（低电平）	*L	*L	*L

3rd Cycle	A12	A13	A14	A15	A16	A17	A18	A19
4th Cycle	A20	A21	A22	A23	A24	A25	A26	A27

其中第 1 和第 2 个字节表示列（Column）地址，即页内地址（含 OOB 区），第 3 和第 4 字节表示行（Row）地址，即表示哪个块中的哪个页（寻址页）。

NAND Flash 与 SD 卡类似，通过写入命令来对其进行读写和控制，以下列举了部分命令值：

```
/*/include/linux/mtd/nand.h*/
#define   NAND_CMD_READ0        0     /*读一页的前半部分*/
#define   NAND_CMD_READ1        1     /*读一页的后半部分*/
#define   NAND_CMD_RNDOUT       5
#define   NAND_CMD_PAGEPROG     0x10
#define   NAND_CMD_READOOB      0x50
#define   NAND_CMD_ERASE1       0x60     /*擦除块*/
#define   NAND_CMD_STATUS       0x70
#define   NAND_CMD_SEQIN        0x80     /*按页写*/
#define   NAND_CMD_RNDIN        0x85
#define   NAND_CMD_READID       0x90     /*读芯片 ID 值*/
#define   NAND_CMD_ERASE2       0xd0
#define   NAND_CMD_PARAM        0xec
#define   NAND_CMD_GET_FEATURES 0xee
#define   NAND_CMD_SET_FEATURES  0xef
#define   NAND_CMD_RESET        0xff     /*复位芯片 ID*/
```

NAND Flash 操作时序请读者参考具体芯片手册。

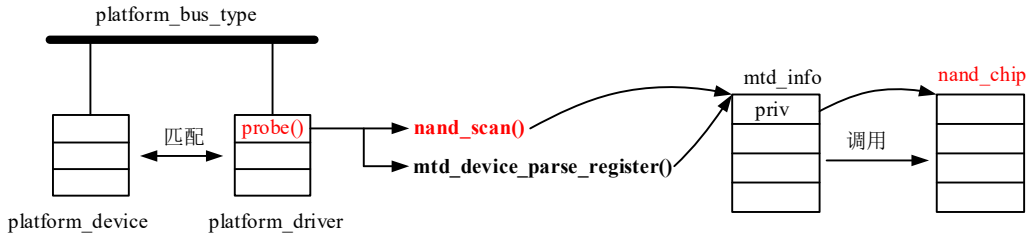
NAND Flash 驱动程序实际是 NAND Flash 控制器驱动程序，驱动程序需要按芯片要求产生接口时序，实现数据的传输。

2 驱动框架

NAND Flash 驱动采用了 MTD 驱动框架，驱动程序中 NAND Flash 由 **nand_chip** 结构体表示，结构体中主要包含底层操作函数指针成员。驱动程序中需要定义 nand_chip 实例和底层操作函数，以及 MTD 驱动框架中表示原始设备的 mtd_info 实例，**mtd_info->priv** 成员指向 nand_chip 实例。

在嵌入式系统中 NAND Flash 驱动实际是主机控制器驱动，驱动框架如下图所示。nand_chip 结构体中底层操作函数通过对控制器中寄存器的操作，实现与 NAND Flash 之间的数据/命令传输。

主机控制器驱动的 probe()函数中除了需要实现底层硬件的初始化、中断的注册等操作外，最主要的工作就是调用 **nand_scan()**函数扫描连接的 NAND Flash 芯片及初始化 mtd_info 实例，最后调用前面介绍的注册 mtd_info 实例的接口函数注册实例，实现与 MTD 驱动框架的对接。



■nand_chip

nand_chip 结构体表示 NAND Flash 芯片，结构体定义如下（/include/linux/mtd/nand.h）：

```
struct nand_chip {
    void __iomem *IO_ADDR_R;     /*读 D0~D7 数据的寄存器（控制器中寄存器）*/
    void __iomem *IO_ADDR_W;     /*写 D0~D7 数据的寄存器（控制器中寄存器）*/

    struct device_node *dn;          /*设备树节点*/

    uint8_t (*read_byte)(struct mtd_info *mtd);     /*读字节*/
    u16 (*read_word)(struct mtd_info *mtd);       /*读字*/
    void (*write_byte)(struct mtd_info *mtd, uint8_t byte);     /*写字节*/
    void (*write_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);   /*写缓存区数据至芯片*/
    void (*read_buf)(struct mtd_info *mtd, uint8_t *buf, int len);     /*从芯片读数据至缓存区*/
    void (*select_chip)(struct mtd_info *mtd, int chip);         /*片选*/
    int (*block_bad)(struct mtd_info *mtd, loff_t ofs, int getchip);   /*检查是否是坏块*/
    int (*block_markbad)(struct mtd_info *mtd, loff_t ofs);     /*标记坏块*/
    void (*cmd_ctrl)(struct mtd_info *mtd, int dat, unsigned int ctrl);   /*控制 ALE/CLE/nCE 引脚*/
    int (*init_size)(struct mtd_info *mtd, struct nand_chip *this,u8 *id_data);
                                                    /*设置 mtd->oobsize, mtd->writesize 等*/
    int (*dev_ready)(struct mtd_info *mtd);     /*访问 RDY 引脚*/
    void (*cmdfunc)(struct mtd_info *mtd, unsigned command, int column,int page_addr);
                                                                /*向芯片写入命令*/
    int(*waitfunc)(struct mtd_info *mtd, struct nand_chip *this);   /*等待芯片准备好*/
    int (*erase)(struct mtd_info *mtd, int page);       /*擦除功能*/
    int (*scan_bbt)(struct mtd_info *mtd);     /*扫描坏块列表*/
    int (*errstat)(struct mtd_info *mtd, struct nand_chip *this, int state,int status, int page);
    int (*write_page)(struct mtd_info *mtd, struct nand_chip *chip,uint32_t offset,
                            int data_len, const uint8_t *buf,int oob_required, int page, int cached, int raw);
                                                            /*按页写函数*/
    int (*onfi_set_features)(struct mtd_info *mtd, struct nand_chip *chip,
                                    int feature_addr, uint8_t *subfeature_para);
    int (*onfi_get_features)(struct mtd_info *mtd, struct nand_chip *chip,
                                        int feature_addr, uint8_t *subfeature_para);
    int (*setup_read_retry)(struct mtd_info *mtd, int retry_mode);

    int chip_delay;
    unsigned int options;   /*选项，例如：NAND_CACHEPRG，定义在/include/linux/mtd/nand.h*/
    unsigned int bbt_options;
    ...
    int read_retries;
    flstate_t state;     /*芯片状态*/
```

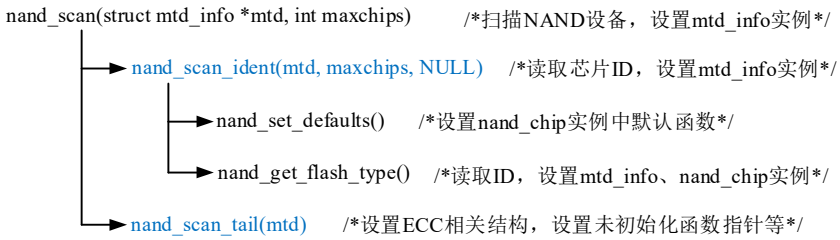
```
    uint8_t *oob_poi;
    struct nand_hw_control *controller;      /*硬件控制结构指针*/

    struct nand_ecc_ctrl ecc;
    struct nand_buffers *buffers;     /*读写缓存*/
    struct nand_hw_control hwcontrol;   /*硬件控制结构*/
    ...
    void *priv;
};
```

nand_chip 结构体中包含 NAND Flash 芯片信息，以及底层操作函数指针等，mtd_info 实例中的函数将调用这些函数。

■扫描芯片

主机控制器驱动程序在定义 nand_chip 和 mtd_info 实例后，需要扫描控制器连接的 NAND Flash 芯片，定义、设置和注册 mtd_info 实例。

扫描芯片操作主要分两步，由 **nand_scan()**接口函数完成，函数调用关系如下图所示：



第一步是调用 nand_scan()函数读取芯片 ID，设置 mtd_info 实例，第二步是调用 nand_scan_tail()函数设置 mtd_info 实例中未初始化的函数指针成员。

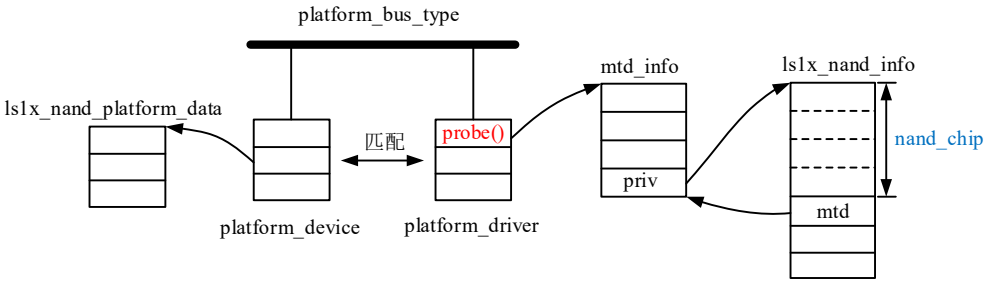
nand_scan()函数定义如下（/drivers/mtd/nand/nand_base.c）：

```
int nand_scan(struct mtd_info *mtd, int maxchips)
{
    int ret;
    ...
    ret = nand_scan_ident(mtd, maxchips, NULL);  /*读取芯片 ID，设置 mtd_info、nand_chip 实例*/
    if (!ret)
        ret = nand_scan_tail(mtd);  /*设置未初始化函数指针，扫描坏块列表等*/
    return ret;
}
```

nand_scan_ident()和 nand_scan_tail()函数源代码，有兴趣的读者可自行阅读。

3 驱动示例

下面以龙芯 1B 处理器为例简要说明 NAND Flash（主机控制器）驱动程序的实现。驱动程序框架如下图所示：



在平台相关的头文件中定义了 ls1x_nand_platform_data 结构体，用于传递 NAND Flash（主机控制器）的硬件信息，在驱动程序中定义了 ls1x_nand_info 结构体表示 NAND Flash 芯片信息（内嵌 nand_chip 结构体成员）。在主机控制器驱动的 probe()函数中将创建 mtd_info 和 ls1x_nand_info 结构体实例，并建立关联，然后调用 **ls1x_nand_scan(mtd)**函数设置 mtd_info 和 nand_chip 实例，最后调用 **mtd_device_register()**函数注册 mtd_info 实例。

ls1x_nand_platform_data 结构体表示 NAND Flash（主机控制器）的硬件信息，定义如下：

```
struct ls1x_nand_platform_data {
    int    enable_arbiter;
    int    keep_config;

    const struct mtd_partition      *parts;     /*分区信息，指向数组*/
    unsigned int                    nr_parts;
    const struct ls1x_nand_flash *      flash;
    size_t                          num_flash;
};
```

在平台相关的 platform.c 文件内定义分区数组，以及主机控制器的 platform_device 实例：

```
static struct mtd_partition ls1x_nand_partitions[] = {      /*分区信息*/
    {
        .name      = "kernel",
        .offset    = MTDPART_OFS_APPEND,
        .size = 14*1024*1024,
    }, {
        .name      = "rootfs",
        .offset    = MTDPART_OFS_APPEND,
        .size = 100*1024*1024,
    }, {
        .name      = "data",
        .offset    = MTDPART_OFS_APPEND,
        .size = MTDPART_SIZ_FULL,
    },
};

static struct ls1x_nand_platform_data ls1x_nand_parts = {
    .parts      = ls1x_nand_partitions,      /*分区信息*/
```

```
    .nr_parts  = ARRAY_SIZE(ls1x_nand_partitions),
};

static struct resource ls1x_nand_resources[] = {    /*资源*/
    ...
};

struct platform_device ls1x_nand_device = {          /*实例在初始化函数中注册*/
    .name      = "ls1x-nand",       /*名称，匹配驱动*/
    .id        = -1,
    .dev = {
        .platform_data = &ls1x_nand_parts,
    },
    .num_resources      = ARRAY_SIZE(ls1x_nand_resources),
    .resource       = ls1x_nand_resources,
};
```

在 NAND Flash 主机控制器驱动程序中定义并注册了对应的 platform_driver 实例，如下：

```
static struct platform_driver ls1x_nand_driver = {
    .driver = {
        .name     = "ls1x-nand",      /*匹配设备*/
        .owner    = THIS_MODULE,
    },
    .probe          = ls1x_nand_probe,      /*驱动 probe()函数*/
    .remove         = ls1x_nand_remove,
    .suspend  = ls1x_nand_suspend,
    .resume         = ls1x_nand_resume,
};
```

下面重点看一下主机控制器驱动 probe()函数的实现，如下所示：

```
static int ls1x_nand_probe(struct platform_device *pdev)
{
    struct ls1x_nand_platform_data *pdata;
    struct ls1x_nand_info *info;
    struct nand_chip *chip;
    struct mtd_info *mtd;
    struct resource *r;
    int ret = 0, irq;

    pdata = pdev->dev.platform_data;
    ...
    mtd = kzalloc(sizeof(struct mtd_info) + sizeof(struct ls1x_nand_info),GFP_KERNEL); /*分配实例*/
    ...
```

```
/*设置实例*/
info = (struct ls1x_nand_info *)(&mtd[1]);
chip = (struct nand_chip *)(&mtd[1]);  /*  注意指针指向  */
info->pdev = pdev;
info->mtd = mtd;
mtd->priv = info;
mtd->owner = THIS_MODULE;
/*设置底层操作函数指针*/
chip->options        = NAND_CACHEPRG;
chip->ecc.mode           = NAND_ECC_SOFT;
chip->waitfunc           = ls1x_nand_waitfunc;     /*nand_chip 操作函数主要是对寄存器的操作*/
chip->select_chip    = ls1x_nand_select_chip;
chip->dev_ready          = ls1x_nand_dev_ready;
chip->cmdfunc            = ls1x_nand_cmdfunc;
chip->read_word          = ls1x_nand_read_word;
chip->read_byte          = ls1x_nand_read_byte;
chip->read_buf           = ls1x_nand_read_buf;
chip->write_buf          = ls1x_nand_write_buf;
chip->verify_buf     = ls1x_nand_verify_buf;

info->clk = clk_get(NULL, "apb");
...
irq = platform_get_irq(pdev, 0);          /*中断编号*/
...
r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
...
r = request_mem_region(r->start, resource_size(r), pdev->name);
...
info->mmio_base = ioremap(r->start, resource_size(r));
...
ret = ls1x_nand_init_buff(info);
...
ret = request_irq(irq, ls1x_nand_irq, IRQF_DISABLED,pdev->name, info);       /*注册中断*/
...
nand_gpio_init();
ls1x_dma_init(info);
ls1x_nand_init_hw(info);
chip->cmdfunc(mtd, NAND_CMD_RESET, 0, 0);
platform_set_drvdata(pdev, info);

if (ls1x_nand_scan(mtd)) {        /*扫描 NAND Flash 芯片，与 nand_scan_ident()函数类似*/
    ...
}
```

```
    mtd->name = "ls1x-nand";

    if (mtd_has_cmdlinepart()) {     /*如果从命令行获取分区信息*/
        const char *probes[] = { "cmdlinepart", NULL };
        struct mtd_partition *parts = NULL;
        int nr_parts = 0;

        nr_parts = parse_mtd_partitions(mtd, probes, &parts, 0);
        if (nr_parts) {
            return mtd_device_register(mtd, parts, nr_parts);          /*注册 mtd_info 实例*/
        }
    }

    return mtd_device_register(mtd, pdata->parts, pdata->nr_parts);     /*注册 mtd_info 实例*/
    ...
}
```

probe()函数主要是创建并初始化 mtd_info 和 ls1x_nand_info 结构体实例；调用 ls1x_nand_scan()函数扫描 NAND Flash 芯片，最后调用 mtd_device_register()函数注册 mtd_info 实例。

NAND Flash 控制器的操作主要是对其寄存器的操作，在数据的读写操作中使用了内存缓存和 DMA，操作函数源代码请读者自行阅读。

扫描芯片的 ls1x_nand_scan()函数与 nand_scan()函数内调用的函数相同，请读者自行阅读源代码。

10.10 小结

由于块设备传输的数据量大，且相对于 CPU 来说速度较慢，对块设备的访问不能立即得到结果，因此块设备驱动程序相对于字符设备驱动程序来说复杂许多。

块设备驱动程序的主要工作是分配并设置表示磁盘的 gendisk 结构体实例，创建请求队列 request_queue 实例并赋予 gendisk 实例，最后添加 gendisk 实例。在添加 gendisk 实例时，将扫描磁盘分区，创建、设置并添加表示分区的 hd_struct 结构体实例。

用户在打开块设备文件或挂载分区时，将在内核中为分区（磁盘）创建 block_devcie 结构体实例，这个结构体主要表示分区（磁盘）的软件信息。

用户对块设备的访问封装成 bio 实例，提交到请求队列，请求队列进而将 bio 实例打包成请求 request 实例添加到请求队列。块设备驱动程序中通常创建一个内核线程，此线程不断地从队列中取请求，执行请求中的数据传输或命令，最后结束请求。请求添加到请求队列时将会唤醒这个内核线程。

请求队列是块设备驱动程序中最重要和最复杂的部分。内核支持的请求队列类型主要有标准请求队列（单队列）和 Multi queue 请求队列（多队列模型）等，后者是为了减轻多核多进程系统中对队列锁的竞争，以提高效率。

本章首先介绍了块设备驱动程序的框架，然后介绍了用户对块设备的访问操作，最后介绍了 LOOP、SD 卡和 MTD 设备驱动程序框架的实现。