

35 | 块设备（下）：如何建立代理商销售模式？

2019-06-17 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 19:05 大小 15.30M



在[文件系统](#)那一节，我们讲了文件的写入，到了设备驱动这一层，就没有再往下分析。上一节我们又讲了 mount 一个块设备，将 `block_device` 信息放到了 ext4 文件系统的

super_block 里面，有了这些基础，是时候把整个写入的故事串起来了。

还记得咱们在文件系统那一节分析写入流程的时候，对于 ext4 文件系统，最后调用的是 ext4_file_write_iter，它将 I/O 的调用分成两种情况：

第一是**直接 I/O**。最终我们调用的是 generic_file_direct_write，这里调用的是 mapping->a_ops->direct_IO，实际调用的是 ext4_direct_IO，往设备层写入数据。


第二种是**缓存 I/O**。最终我们会将数据从应用拷贝到内存缓存中，但是这个时候，并不执行真正的 I/O 操作。它们只将整个页或其中部分标记为脏。写操作由一个 timer 触发，那个时候，才调用 wb_workfn 往硬盘写入页面。

接下来的调用链为：wb_workfn->wb_do_writeback->wb_writeback->writeback_sb_inodes->__writeback_single_inode->do_writepages。在 do_writepages 中，我们要调用 mapping->a_ops->writepages，但实际调用的是 ext4_writepages，往设备层写入数据。

这一节，我们就沿着这两种情况分析下去。

直接 I/O 如何访问块设备？

我们先来看第一种情况，直接 I/O 调用到 `ext4_direct_IO`。

 复制代码


```
1 static ssize_t ext4_direct_IO(struct kiocb *iocb, struc
2 {
3     struct file *file = iocb->ki_filp;
4     struct inode *inode = file->f_mapping->host;
5     size_t count = iov_iter_count(iter);
6     loff_t offset = iocb->ki_pos;
7     ssize_t ret;
8     .....
9     ret = ext4_direct_IO_write(iocb, iter);
10    .....
11 }
12
13
14 static ssize_t ext4_direct_IO_write(struct kiocb *iocb,
15 {
16     struct file *file = iocb->ki_filp;
17     struct inode *inode = file->f_mapping->host;
18     struct ext4_inode_info *ei = EXT4_I(inode);
19     ssize_t ret;
20     loff_t offset = iocb->ki_pos;
21     size_t count = iov_iter_count(iter);
22     .....
23     ret = __blockdev_direct_IO(iocb, inode, inode->
24                               get_block_func, ext4
25                               dio_flags);
```

```
26
27
28 .....
29 }
```



在 `ext4_direct_IO_write` 调用 `__blockdev_direct_IO`，有个参数你需要特别注意一下，那就是 `inode->i_sb->s_bdev`。通过当前文件的 `inode`，我们可以得到 `super_block`。这个 `super_block` 中的 `s_bdev`，就是咱们上一节填进去的那个 `block_device`。

`__blockdev_direct_IO` 会调用 `do_blockdev_direct_IO`，在这里面我们要准备一个 `struct dio` 结构和 `struct dio_submit` 结构，用来描述将要发生的写入请求。

 复制代码

```
1 static inline ssize_t
2 do_blockdev_direct_IO(struct kiocb *iocb, struct inode
3                       struct block_device *bdev, struct
4                       get_block_t get_block, dio_iodone
5                       dio_submit_t submit_io, int flags
6 {
7     unsigned i_blkbits = ACCESS_ONCE(inode->i_blkbi
8     unsigned blkbits = i_blkbits;
9     unsigned blocksize_mask = (1 << blkbits) - 1;
10    ssize_t retval = -EINVAL;
11    size_t count = iov_iter_count(iter);
```

```
12         loff_t offset = iocb->ki_pos;
13         loff_t end = offset + count;
14         struct dio *dio;
15         struct dio_submit sdio = { 0, };
16         struct buffer_head map_bh = { 0, };
17         .....
18         dio = kmem_cache_alloc(dio_cache, GFP_KERNEL);
19         dio->flags = flags;
20         dio->i_size = i_size_read(inode);
21         dio->inode = inode;
22         if (iov_iter_rw(iter) == WRITE) {
23             dio->op = REQ_OP_WRITE;
24             dio->op_flags = REQ_SYNC | REQ_IDLE;
25             if (iocb->ki_flags & IOCB_NOWAIT)
26                 dio->op_flags |= REQ_NOWAIT;
27         } else {
28             dio->op = REQ_OP_READ;
29         }
30         sdio.blkbits = blkbits;
31         sdio.blkfactor = i_blkbits - blkbits;
32         sdio.block_in_file = offset >> blkbits;
33
34
35         sdio.get_block = get_block;
36         dio->end_io = end_io;
37         sdio.submit_io = submit_io;
38         sdio.final_block_in_bio = -1;
39         sdio.next_block_for_io = -1;
40
41
42         dio->iocb = iocb;
43         dio->refcount = 1;
44
45
46         sdio.iter = iter;
```


```

47         sdio.final_block_in_request =
48             (offset + iov_iter_count(iter)) >> blk_t
49         .....
50         sdio.pages_in_io += iov_iter_npages(iter, INT_M
51
52
53         retval = do_direct_IO(dio, &sdio, &map_bh);
54         .....
55     }

```



do_direct_IO 里面有两层循环，第一层循环是依次处理这次要写入的所有块。对于每一块，取出对应的内存中的页 page，在这一块中，有写入的起始地址 from 和终止地址 to，所以，第二层循环就是依次处理 from 到 to 的数据，调用 submit_page_section，提交到块设备层进行写入。

 复制代码

```

1 static int do_direct_IO(struct dio *dio, struct dio_sub
2                         struct buffer_head *map_bh)
3 {
4     const unsigned blkbits = sdio->blkbits;
5     const unsigned i_blkbits = blkbits + sdio->blkf
6     int ret = 0;
7
8
9     while (sdio->block_in_file < sdio->final_block_
10           struct page *page;
11           size_t from, to;
12

```

```


13
14         page = dio_get_page(dio, sdio);
15     from = sdio->head ? 0 : sdio->from;
16         to = (sdio->head == sdio->tail - 1) ? s
17         sdio->head++;
18
19
20         while (from < to) {
21             unsigned this_chunk_bytes;
22             unsigned this_chunk_blocks;
23     .....
24         ret = submit_page_section(dio, sdio, page,
25                                   from,
26                                   this_
27                                   sdio-
28                                   map_t
29     .....
30             sdio->next_block_for_io += this
31             sdio->block_in_file += this_chu
32             from += this_chunk_bytes;
33             dio->result += this_chunk_bytes
34             sdio->blocks_available -= this_
35             if (sdio->block_in_file == sdic
36                 break;
37     .....
38         }
39     }
40 }

```



submit_page_section 会调用 dio_bio_submit, 进而调用 submit_bio 向块设备层提交数据。其中, 参数 struct bio


是将数据传给块设备的通用传输对象。定义如下：

 复制代码

```
1 /**
2  * submit_bio - submit a bio to the block device layer
3  * @bio: The &struct bio which describes the I/O
4  */
5 blk_qc_t submit_bio(struct bio *bio)
6 {
7     .....
8     return generic_make_request(bio);
9 }
```

缓存 I/O 如何访问块设备？

我们再来看第二种情况，缓存 I/O 调用到 `ext4_writepages`。这个函数比较长，我们这里只截取最重要的部分来讲解。

 复制代码

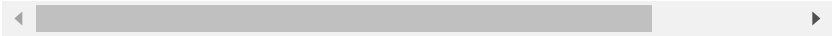
```
1 static int ext4_writepages(struct address_space *mapping
2                             struct writeback_control *wt
3 {
4     .....
5     struct mpage_da_data mpd;
6     struct inode *inode = mapping->host;
7     struct ext4_sb_info *sbi = EXT4_SB(mapping->hos
8     .....
```




```

9         mpd.do_map = 0;
10        mpd.io_submit.io_end = ext4_init_io_end(inode,
11        ret = mpage_prepare_extent_to_map(&mpd);
12        /* Submit prepared bio */
13        ext4_io_submit(&mpd.io_submit);
14        .....
15    }

```



这里比较重要的一个数据结构是 struct mpage_da_data。这里面有文件的 inode、要写入的页的偏移量，还有一个重要的 struct ext4_io_submit，里面有通用传输对象 bio。

 复制代码

```

1 struct mpage_da_data {
2     struct inode *inode;
3     .....
4     pgoff_t first_page;      /* The first page to wr
5     pgoff_t next_page;      /* Current page to exam
6     pgoff_t last_page;      /* Last page to examine
7     struct ext4_map_blocks map;
8     struct ext4_io_submit io_submit;          /* IO s
9     unsigned int do_map:1;
10 };
11
12
13 struct ext4_io_submit {
14     .....
15     struct bio          *io_bio;
16     ext4_io_end_t      *io_end;
17     sector_t            io_next_block;


```

```
18 };
```



在 `ext4_writepages` 中,
`mpage_prepare_extent_to_map` 用于初始化这个 `struct mpage_da_data` 结构。接下来的调用链为:
`mpage_prepare_extent_to_map`-
>`mpage_process_page_bufs`->`mpage_submit_page`-
>`ext4_bio_write_page`->`io_submit_add_bh`。


在 `io_submit_add_bh` 中, 此时的 `bio` 还是空的, 因而我们要调用 `io_submit_init_bio`, 初始化 `bio`。

 复制代码

```
1 static int io_submit_init_bio(struct ext4_io_submit *ic
2                               struct buffer_head *bh)
3 {
4     struct bio *bio;
5
6
7     bio = bio_alloc(GFP_NOIO, BIO_MAX_PAGES);
8     if (!bio)
9         return -ENOMEM;
10    wbc_init_bio(io->io_wbc, bio);
11    bio->bi_iter.bi_sector = bh->b_blocknr * (bh->t
12    bio->bi_bdev = bh->b_bdev;
13    bio->bi_end_io = ext4_end_bio;
14    bio->bi_private = ext4_get_io_end(io->io_end);
```

```
15         io->io_bio = bio;
16         io->io_next_block = bh->b_blocknr;
17         return 0;
18     }
19
```

我们再回到 `ext4_writepages` 中。在 `bio` 初始化完之后，我们要调用 `ext4_io_submit`，提交 I/O。在这里我们又是调用 `submit_bio`，向块设备层传输数据。`ext4_io_submit` 的实现如下：


 复制代码

```
1 void ext4_io_submit(struct ext4_io_submit *io)
2 {
3     struct bio *bio = io->io_bio;
4
5
6     if (bio) {
7         int io_op_flags = io->io_wbc->sync_mode
8                             REQ_SYNC : 0;
9         io->io_bio->bi_write_hint = io->io_end-
10         bio_set_op_attrs(io->io_bio, REQ_OP_WRI
11         submit_bio(io->io_bio);
12     }
13     io->io_bio = NULL;
14 }
15
```

如何向块设备层提交请求？

既然无论是直接 I/O，还是缓存 I/O，最后都到了 submit_bio 里面，我们就来重点分析一下它。

submit_bio 会调用 generic_make_request。代码如下：

 复制代码

```
1 blk_qc_t generic_make_request(struct bio *bio)
2 {
3     /*
4      * bio_list_on_stack[0] contains bios submitted
5      * make_request_fn.
6      * bio_list_on_stack[1] contains bios that were
7      * the current make_request_fn, but that haven'
8      * yet.
9      */
10    struct bio_list bio_list_on_stack[2];
11    blk_qc_t ret = BLK_QC_T_NONE;
12    .....
13    if (current->bio_list) {
14        bio_list_add(&current->bio_list[0], bio);
15        goto out;
16    }
17
18
19    bio_list_init(&bio_list_on_stack[0]);
20    current->bio_list = bio_list_on_stack;
21    do {
22        struct request_queue *q = bdev_get_queue
23
24
```

```

25         if (likely(blk_queue_enter(q, bio->bi_c
26                 struct bio_list lower, same;
27
28
29                 /* Create a fresh bio_list for
30                 bio_list_on_stack[1] = bio_list
31                 bio_list_init(&bio_list_on_stac
32                 ret = q->make_request_fn(q, bic
33
34
35                 blk_queue_exit(q);
36
37
38                 /* sort new bios into those for
39                 * and those for the same level
40                 */
41                 bio_list_init(&lower);
42                 bio_list_init(&same);
43                 while ((bio = bio_list_pop(&bic
44                         if (q == bdev_get_queue
45                             bio_list_add(&s
46                         else
47                             bio_list_add(&l
48                 /* now assemble so we handle th
49                 bio_list_merge(&bio_list_on_sta
50                 bio_list_merge(&bio_list_on_sta
51                 bio_list_merge(&bio_list_on_sta
52         }
53     .....
54         bio = bio_list_pop(&bio_list_on_stack[0
55     } while (bio);
56     current->bio_list = NULL; /* deactivate */
57 out:
58     return ret;
59 }


```

这里的逻辑有点复杂，我们先来看大的逻辑。在 do-while 中，我们先是获取一个请求队列 request_queue，然后调用这个队列的 make_request_fn 函数。

块设备队列结构


如果再来看 struct block_device 结构和 struct gendisk 结构，我们会发现，每个块设备都有一个请求队列 struct request_queue，用于处理上层发来的请求。

在每个块设备的驱动程序初始化的时候，会生成一个 request_queue。

 复制代码


```
1 struct request_queue {
2     /*
3      * Together with queue_head for cacheline shari
4      */
5     struct list_head    queue_head;
6     struct request      *last_merge;
7     struct elevator_queue *elevator;
8     .....
9     request_fn_proc     *request_fn;
10    make_request_fn      *make_request_fn;
11    .....
12 }
```

在请求队列 request_queue 上，首先是有个链表 list_head，保存请求 request。

 复制代码

```
1 struct request {
2     struct list_head queuelist;
3     .....
4     struct request_queue *q;
5     .....
6     struct bio *bio;
7     struct bio *biotail;
8     .....
9 }
```

每个 request 包括一个链表的 struct bio，有指针指向一头一尾。

 复制代码

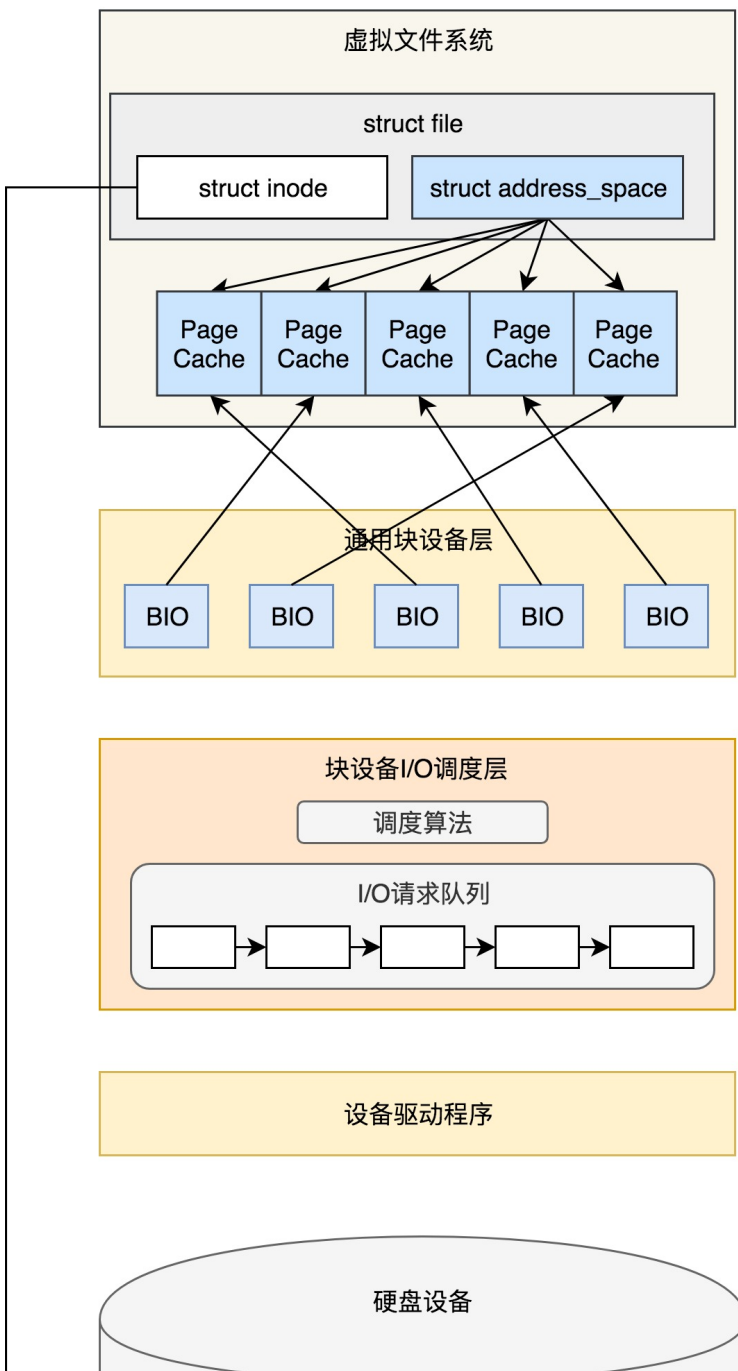
```
1 struct bio {
2     struct bio          *bi_next;          /* request
3     struct block_device *bi_bdev;
4     blk_status_t        bi_status;
5     .....
6     struct bvec_iter    bi_iter;
7     unsigned short      bi_vcnt;          /* how
```

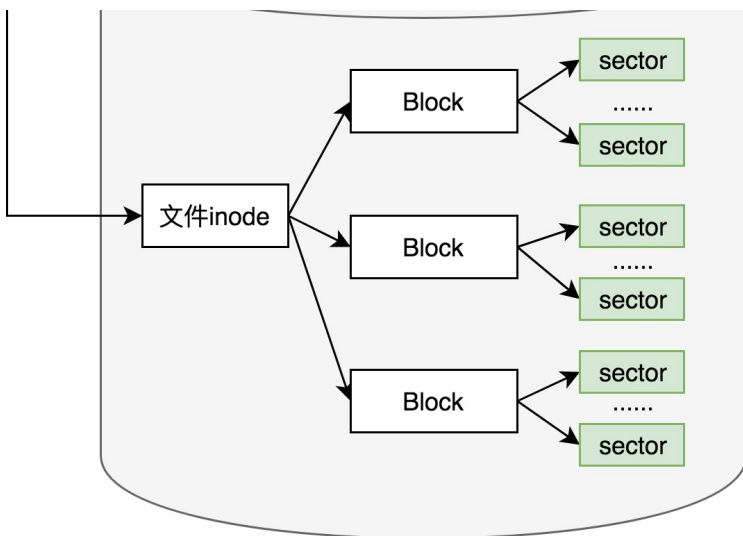
```

 8      unsigned short      bi_max_vecs;    /* max
 9      atomic_t            __bi_cnt;       /* pin
10      struct bio_vec       *bi_io_vec;     /* the
11      .....
12  };
13
14
15  struct bio_vec {
16      struct page          *bv_page;
17      unsigned int         bv_len;
18      unsigned int         bv_offset;
19  }
```



在 bio 中，bi_next 是链表中的下一项，struct bio_vec 指向一组页面。





在请求队列 `request_queue` 上，还有两个重要的函数，一个是 `make_request_fn` 函数，用于生成 request；另一个是 `request_fn` 函数，用于处理 request。

块设备的初始化

我们还是以 `scsi` 驱动为例。在初始化设备驱动的时候，我们会调用 `scsi_alloc_queue`，把 `request_fn` 设置为 `scsi_request_fn`。我们还会调用 `blk_init_allocated_queue->blk_queue_make_request`，把 `make_request_fn` 设置为 `blk_queue_bio`。

```

1  /**
2   * scsi_alloc_sdev - allocate and setup a scsi_Device
3   * @target: which target to allocate a &scsi_device for
4   * @lun: which lun
5   * @hostdata: usually NULL and set by ->slave_alloc ins
6   *
7   * Description:
8   *     Allocate, initialize for io, and return a pointer
9   *     Stores the @shost, @channel, @id, and @lun in the
10  *     adds scsi_Device to the appropriate list.
11  *
12  * Return value:
13  *     scsi_Device pointer, or NULL on failure.
14  */
15 static struct scsi_device *scsi_alloc_sdev(struct scsi_
16                                             u64 lun, void
17 {
18     struct scsi_device *sdev;
19     sdev = kzalloc(sizeof(*sdev) + shost->transport
20                   GFP_ATOMIC);
21     .....
22     sdev->request_queue = scsi_alloc_queue(sdev);
23     .....
24 }
25
26
27 struct request_queue *scsi_alloc_queue(struct scsi_device
28 {
29     struct Scsi_Host *shost = sdev->host;
30     struct request_queue *q;
31
32
33     q = blk_alloc_queue_node(GFP_KERNEL, NUMA_NO_NODE
34     if (!q)
35         return NULL;

```


```

36         q->cmd_size = sizeof(struct scsi_cmnd) + shost-
37         q->rq_alloc_data = shost;
38         q->request_fn = scsi_request_fn;
39         q->init_rq_fn = scsi_init_rq;
40         q->exit_rq_fn = scsi_exit_rq;
41         q->initialize_rq_fn = scsi_initialize_rq;
42
43
44         // 调用 blk_queue_make_request(q, blk_queue_bio);
45         if (blk_init_allocated_queue(q) < 0) {
46             blk_cleanup_queue(q);
47             return NULL;
48         }
49
50
51         __scsi_init_queue(shost, q);
52         .....
53         return q
54     }

```



在 `blk_init_allocated_queue` 中，除了初始化 `make_request_fn` 函数，我们还要做一件很重要的事情，就是初始化 I/O 的电梯算法。


 复制代码

```

1 int blk_init_allocated_queue(struct request_queue *q)
2 {
3     q->fq = blk_alloc_flush_queue(q, NUMA_NO_NODE,
4     .....
5     blk_queue_make_request(q, blk_queue_bio);

```

```
6 .....
7         /* init elevator */
8         if (elevator_init(q, NULL)) {
9 .....
10        }
11 .....
12 }
```



电梯算法有很多种类型，定义为 `elevator_type`。下面我来逐一说一下。

`struct elevator_type elevator_noop`

Noop 调度算法是最简单的 IO 调度算法，它将 IO 请求放入到一个 FIFO 队列中，然后逐个执行这些 IO 请求。

`struct elevator_type iosched_deadline`

Deadline 算法要保证每个 IO 请求在一定的时间内一定要被服务到，以此来避免某个请求饥饿。为了完成这个目标，算法中引入了两类队列，一类队列用来对请求按起始扇区序号进行排序，通过红黑树来组织，我们称为 `sort_list`，按照此队列传输性能会比较高；另一类队列对请求按它们的生成时间进行排序，由链表来组织，称为 `fifo_list`，并且每一个请求都有一个期限值。


struct elevator_type iosched_cfq

又看到了熟悉的 CFQ 完全公平调度算法。所有的请求会在多个队列中排序。同一个进程的请求，总是在同一队列中处理。时间片会分配到每个队列，通过轮询算法，我们保证了 I/O 带宽，以公平的方式，在不同队列之间进行共享。

elevator_init 中会根据名称来指定电梯算法，如果没有选择，那就默认使用 iosched_cfq。

请求提交与调度

接下来，我们回到 generic_make_request 函数中。调用队列的 make_request_fn 函数，其实就是调用 blk_queue_bio。

 复制代码

```
1 static blk_qc_t blk_queue_bio(struct request_queue *q,
2 {
3     struct request *req, *free;
4     unsigned int request_count = 0;
5     .....
6     switch (elv_merge(q, &req, bio)) {
7     case ELEVATOR_BACK_MERGE:
8         if (!bio_attempt_back_merge(q, req, bio))
9             break;
10        elv_bio_merged(q, req, bio);
11        free = attempt_back_merge(q, req);
```


```

12             if (free)
13                 __blk_put_request(q, free);
14             else
15                 elv_merged_request(q, req, ELEV
16             goto out_unlock;
17         case ELEVATOR_FRONT_MERGE:
18             if (!bio_attempt_front_merge(q, req, bi
19                 break;
20             elv_bio_merged(q, req, bio);
21             free = attempt_front_merge(q, req);
22             if (free)
23                 __blk_put_request(q, free);
24             else
25                 elv_merged_request(q, req, ELEV
26             goto out_unlock;
27         default:
28             break;
29     }
30
31
32 get_rq:
33     req = get_request(q, bio->bi_opf, bio, GFP_NOIC
34     .....
35     blk_init_request_from_bio(req, bio);
36     .....
37     add_acct_request(q, req, where);
38     __blk_run_queue(q);
39 out_unlock:
40     .....
41     return BLK_QC_T_NONE;
42 }

```

blk_queue_bio 首先做的一件事情是调用 elv_merge 来判断，当前这个 bio 请求是否能够和目前已有的 request 合并起来，成为同一批 I/O 操作，从而提高读取和写入的性能。

判断标准和 struct bio 的成员 struct bvec_iter 有关，它里面有两个变量，一个是起始磁盘簇 bi_sector，另一个是大小 bi_size。

 复制代码

```
1 enum elv_merge elv_merge(struct request_queue *q, struct
2     struct bio *bio)
3 {
4     struct elevator_queue *e = q->elevator;
5     struct request *__rq;
6     .....
7     if (q->last_merge && elv_bio_merge_ok(q->last_m
8         enum elv_merge ret = blk_try_merge(q->l
9
10
11         if (ret != ELEVATOR_NO_MERGE) {
12             *req = q->last_merge;
13             return ret;
14         }
15     }
16     .....
17     __rq = elv_rqhash_find(q, bio->bi_iter.bi_sectc
18     if (__rq && elv_bio_merge_ok(__rq, bio)) {
19         *req = __rq;
20         return ELEVATOR_BACK_MERGE;
21     }
```



```
22
23
24         if (e->uses_mq && e->type->ops.mq.request_merge
25             return e->type->ops.mq.request_merge(q,
26         else if (!e->uses_mq && e->type->ops.sq.elevatc
27             return e->type->ops.sq.elevator_merge_f
28
29
30         return ELEVATOR_NO_MERGE;
31 }
```




elv_merge 尝试了三次合并。

第一次，它先判断和上一次合并的 request 能不能再次合并，看看能不能赶上马上要走的这部电梯。在 blk_try_merge 主要做了这样的判断：如果 $\text{blk_rq_pos}(\text{rq}) + \text{blk_rq_sectors}(\text{rq}) == \text{bio->bi_iter.bi_sector}$ ，也就是说这个 request 的起始地址加上它的大小（其实是这个 request 的结束地址），如果和 bio 的起始地址能接得上，那就把 bio 放在 request 的最后，我们称为 ELEVATOR_BACK_MERGE。

如果 $\text{blk_rq_pos}(\text{rq}) - \text{bio_sectors}(\text{bio}) == \text{bio->bi_iter.bi_sector}$ ，也就是说，这个 request 的起始地址减去 bio 的大小等于 bio 的起始地址，这说明 bio 放在 request 的最前面能够接得上，那就把 bio 放在 request 的

最前面，我们称为 ELEVATOR_FRONT_MERGE。否则，那就不合并，我们称为 ELEVATOR_NO_MERGE。


 复制代码

```
1 enum elv_merge blk_try_merge(struct request *rq, struct
2 {
3     .....
4     if (blk_rq_pos(rq) + blk_rq_sectors(rq) == bio->bi_
5         return ELEVATOR_BACK_MERGE;
6     else if (blk_rq_pos(rq) - bio_sectors(bio) == 0
7         return ELEVATOR_FRONT_MERGE;
8     return ELEVATOR_NO_MERGE;
9 }
```

第二次，如果和上一个合并过的 request 无法合并，我们就调用 `elv_rqhash_find`。然后按照 bio 的起始地址查找 request，看有没有能够合并的。如果有的话，因为是按照起始地址找的，应该接在人家的后面，所以是 ELEVATOR_BACK_MERGE。

第三次，调用 `elevator_merge_fn` 试图合并。对于 `iosched_cfq`，调用的是 `cfq_merge`。在这里面，`cfq_find_rq_fmerge` 会调用 `elv_rb_find` 函数，里面的参数是 bio 的结束地址。我们还是要看，能不能找到可以合并

的。如果有的话，因为是按照结束地址找的，应该接在人家前面，所以是 ELEVATOR_FRONT_MERGE。

 复制代码

```
1 static enum elv_merge cfq_merge(struct request_queue *q,
2                                 struct bio *bio)
3 {
4     struct cfq_data *cfqd = q->elevator->elevator_c
5     struct request *__rq;
6
7
8     __rq = cfq_find_rq_fmerge(cfqd, bio);
9     if (__rq && elv_bio_merge_ok(__rq, bio)) {
10         *req = __rq;
11         return ELEVATOR_FRONT_MERGE;
12     }
13
14
15     return ELEVATOR_NO_MERGE;
16 }
17
18
19 static struct request *
20 cfq_find_rq_fmerge(struct cfq_data *cfqd, struct bio *b
21 {
22     struct task_struct *tsk = current;
23     struct cfq_io_cq *cic;
24     struct cfq_queue *cfqq;
25
26
27     cic = cfq_cic_lookup(cfqd, tsk->io_context);
28     if (!cic)
29         return NULL;
```

```
30
31
32     cfqq = cic_to_cfqq(cic, op_is_sync(bio->bi_opf)
33     if (cfqq)
34         return elv_rb_find(&cfqq->sort_list, bi
35
36
37     return NUL
38 }
```



等从 `elv_merge` 返回 `blk_queue_bio` 的时候，我们就知道，应该做哪种类型的合并，接着就要进行真的合并。如果没有办法合并，那就调用 `get_request`，创建一个新的 `request`，调用 `blk_init_request_from_bio`，将 `bio` 放到新的 `request` 里面，然后调用 `add_acct_request`，把新的 `request` 加到 `request_queue` 队列中。

至此，我们解析完了 `generic_make_request` 中最重要的两大逻辑：获取一个请求队列 `request_queue` 和调用这个队列的 `make_request_fn` 函数。

其实，`generic_make_request` 其他部分也很令人困惑。感觉里面有特别多的 `struct bio_list`，倒腾过来，倒腾过去的。这是因为，很多块设备是有层次的。

比如，我们用两块硬盘组成 RAID，两个 RAID 盘组成 LVM，然后我们就可以在 LVM 上创建一个块设备给用户用，我们称接近用户的块设备为**高层次的块设备**，接近底层的块设备为**低层次 (lower) 的块设备**。这样，`generic_make_request` 把 I/O 请求发送给高层次的块设备的时候，会调用高层块设备的 `make_request_fn`，高层块设备又要调用 `generic_make_request`，将请求发送给低层次的块设备。虽然块设备的层次不会太多，但是对于代码 `generic_make_request` 来讲，这可是递归的调用，一不小心，就会递归过深，无法正常退出，而且内核栈的大小又非常有限，所以要比较小心。

这里你是否理解了 `struct bio_list bio_list_on_stack[2]` 的名字为什么叫 `stack` 呢？其实，将栈的操作变成对于队列的操作，队列不在栈里面，会大很多。每次 `generic_make_request` 被当前任务调用的时候，将 `current->bio_list` 设置为 `bio_list_on_stack`，并在 `generic_make_request` 的一开始就判断 `current->bio_list` 是否为空。如果不为空，说明已经在 `generic_make_request` 的调用里面了，就不必调用 `make_request_fn` 进行递归了，直接把请求加入到 `bio_list` 里面就可以了，这就实现了递归的及时退出。

如果 `current->bio_list` 为空，那我们就将 `current->bio_list` 设置为 `bio_list_on_stack` 后，进入 do-while 循环，做咱们分析过的 `generic_make_request` 的两大逻辑。但是，当前的队列调用 `make_request_fn` 的时候，在 `make_request_fn` 的具体实现中，会生成新的 `bio`。调用更底层的块设备，也会生成新的 `bio`，都会放在 `bio_list_on_stack` 的队列中，是一个边处理还边创建的过程。

`bio_list_on_stack[1] = bio_list_on_stack[0]` 这一句在 `make_request_fn` 之前，将之前队列里面遗留没有处理的保存下来，接着 `bio_list_init` 将 `bio_list_on_stack[0]` 设置为空，然后调用 `make_request_fn`，在 `make_request_fn` 里面如果有新的 `bio` 生成，都会加到 `bio_list_on_stack[0]` 这个队列里面来。


`make_request_fn` 执行完毕后，可以想象 `bio_list_on_stack[0]` 可能又多了一些 `bio` 了，接下来的循环中调用 `bio_list_pop` 将 `bio_list_on_stack[0]` 积攒的 `bio` 拿出来，分别放在两个队列 `lower` 和 `same` 中，顾名思义，`lower` 就是更低层次的块设备的 `bio`，`same` 是同层次的块设备的 `bio`。

接下来我们能将 lower、same 以及 bio_list_on_stack[1] 都取出来，放在 bio_list_on_stack[0] 统一进行处理。当然应该 lower 优先了，因为只有底层的块设备的 I/O 做完了，上层的块设备的 I/O 才能做完。

到这里，generic_make_request 的逻辑才算解析完毕。对于写入的数据来讲，其实仅仅到将 bio 请求放在请求队列上，设备驱动程序还没往设备里面写呢。

请求的处理

设备驱动程序往设备里面写，调用的是请求队列 request_queue 的另外一个函数 request_fn。对于 scsi 设备来讲，调用的是 scsi_request_fn。

 复制代码

```
1 static void scsi_request_fn(struct request_queue *q)
2     __releases(q->queue_lock)
3     __acquires(q->queue_lock)
4 {
5     struct scsi_device *sdev = q->queuedata;
6     struct Scsi_Host *shost;
7     struct scsi_cmnd *cmd;
8     struct request *req;
9
10
11     /*
12      * To start with, we keep looping until the que
```

```

13      * the host is no longer able to accept any mor
14      */
15      shost = sdev->host;
16      for (;;) {
17          int rtn;
18          /*
19           * get next queueable request. We do t
20           * that the request is fully prepared €
21           * accept it.
22           */
23          req = blk_peek_request(q);
24      .....
25          /*
26           * Remove the request from the request
27           */
28          if (!(blk_queue_tagged(q) && !blk_queue
29              blk_start_request(req);
30      .....
31          cmd = req->special;
32      .....
33          /*
34           * Dispatch the command to the low-level
35           */
36          cmd->scsi_done = scsi_done;
37          rtn = scsi_dispatch_cmd(cmd);
38      .....
39      }
40      return;
41      .....
42  }

```


在这里面是一个 for 无限循环，从 request_queue 中读取 request，然后封装更加底层的指令，给设备控制器下指令，实施真正的 I/O 操作。

总结时刻

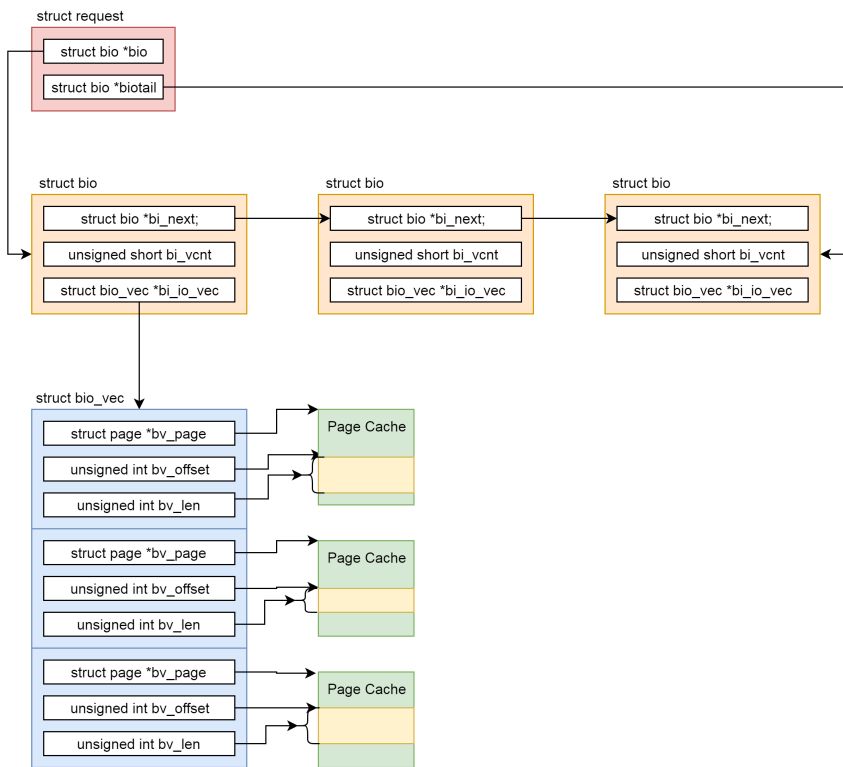
这一节我们讲了如何将块设备 I/O 请求送达到外部设备。

对于块设备的 I/O 操作分为两种，一种是直接 I/O，另一种是缓存 I/O。无论是哪种 I/O，最终都会调用 submit_bio 提交块设备 I/O 请求。

对于每一种块设备，都有一个 gendisk 表示这个设备，它有一个请求队列，这个队列是一系列的 request 对象。每个 request 对象里面包含多个 BIO 对象，指向 page cache。所谓的写入块设备，I/O 就是将 page cache 里面的数据写入硬盘。

对于请求队列来讲，还有两个函数，一个函数叫 make_request_fn 函数，用于将请求放入队列。submit_bio 会调用 generic_make_request，然后调用这个函数。

另一个函数往往在设备驱动程序里实现，我们叫 request_fn 函数，它用于从队列里面取出请求来，写入外部设备。



至此，整个写入文件的过程才完整结束。这真是个复杂的过程，涉及系统调用、内存管理、文件系统和输入输出。这足以说明，操作系统真的是一个非常复杂的体系，环环相扣，需要分层次层层展开来学习。

到这里，专栏已经过半了，你应该能发现，很多我之前说“后面会细讲”的东西，现在正在一点一点解释清楚，而文中越来越多出现“前面我们讲过”的字眼，你是否当时学习前面知识的时候，没有在意，导致学习后面的知识产生困惑了呢？没关系，及时倒回去复习，再回过头去看，当初学过的很多知识会变得清晰很多。

课堂练习

你知道如何查看磁盘调度算法、修改磁盘调度算法以及 I/O 队列的长度吗？

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「👤 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 34 | 块设备（上）：如何建立代理商销售模式？

下一篇 36 | 进程间通信：遇到大项目需要项目组之间的合作才...

精选留言 (11)

写留言



匿名

2019-06-17

一堆代码分析，感觉脱离了所谓的趣谈😂



12



安排

2019-06-17

看了一点儿其它的资料，大概了解了一下，有以下几种情况：1、request_fn是在unplug泄流的时候调用（也就是队列里面的请求达到一定的数量）2、或者是由定时器触发，也就是即使队列里面的请求很少，但是也不能无限期的不执行它们，所以在定时器超时后会调用request_fn...



👍 2



小美

2019-06-19

会不会断电丢失数据呢？



👍 1



Leon 📷

2019-06-17

那个队列，队头和队尾的合并都是为了所谓的顺序写，减少机械硬盘寻址消耗吧，能赶上就上同一辆车，赶不上自己叫一辆车等满了再走，但是也是有超时等待时间的，可以这么理解吧



👍 1



安排

2019-06-17

往设备里面写的时候调用的是request_fn，这个函数是谁来调用的呢？触发这个调用的时机是什么呢？还是说请求一旦进入队列就会立即写入？



Geek_54edc1

2019-06-24

/sys/block/xvda/queue/scheduler 磁盘的调度算法，
临时修改：echo noop >

/sys/block/xvda/queue/scheduler，永久修改就需要修改内核参数，然后重启。

iostat -d -x 中的avgqu-sz是平均I/O队列长度。



Leon

2019-06-22

文件系统中的page_cache对应逻辑上的块的概念，将page_cache打包成bio递交给通用块设备层，通用块设备层将多个bio打包成一个请求request，尽量把bio对应的sector临近的数据合并，提交给块设备调度层，块设备调度层就是把各个request当成段提交给设备驱动程序，...





安排

2019-06-19

老师，希望在答疑篇能讲一讲request_fn取出请求之后的具体执行过程，具体的执行是不是和block_device有关，磁盘的最底层的操作是不是都在block_device中？



安排

2019-06-18

直接读写裸设备不会走文件系统，那还会走通用块层吗？



安排

2019-06-18

make_request_fn被初始化成了blk_queue_bio函数。submit_bio会调用到generic_make_request，然后进一步调用到make_request_fn，也就是blk_queue_bio。在blk_queue_bio里面其实是先尝试将bio合并到当前进程的plug_list里面的request，如果可以合并，则合并后...



石维康

2019-06-17

```
bio_list_merge(&bio_list_on_stack[0], &lower);  
bio_list_merge(&bio_list_on_stack[0], &same);
```

```
bio_list_merge(&bio_list_on_stack[0],  
&bio_list_on_stack[1]);
```

...

