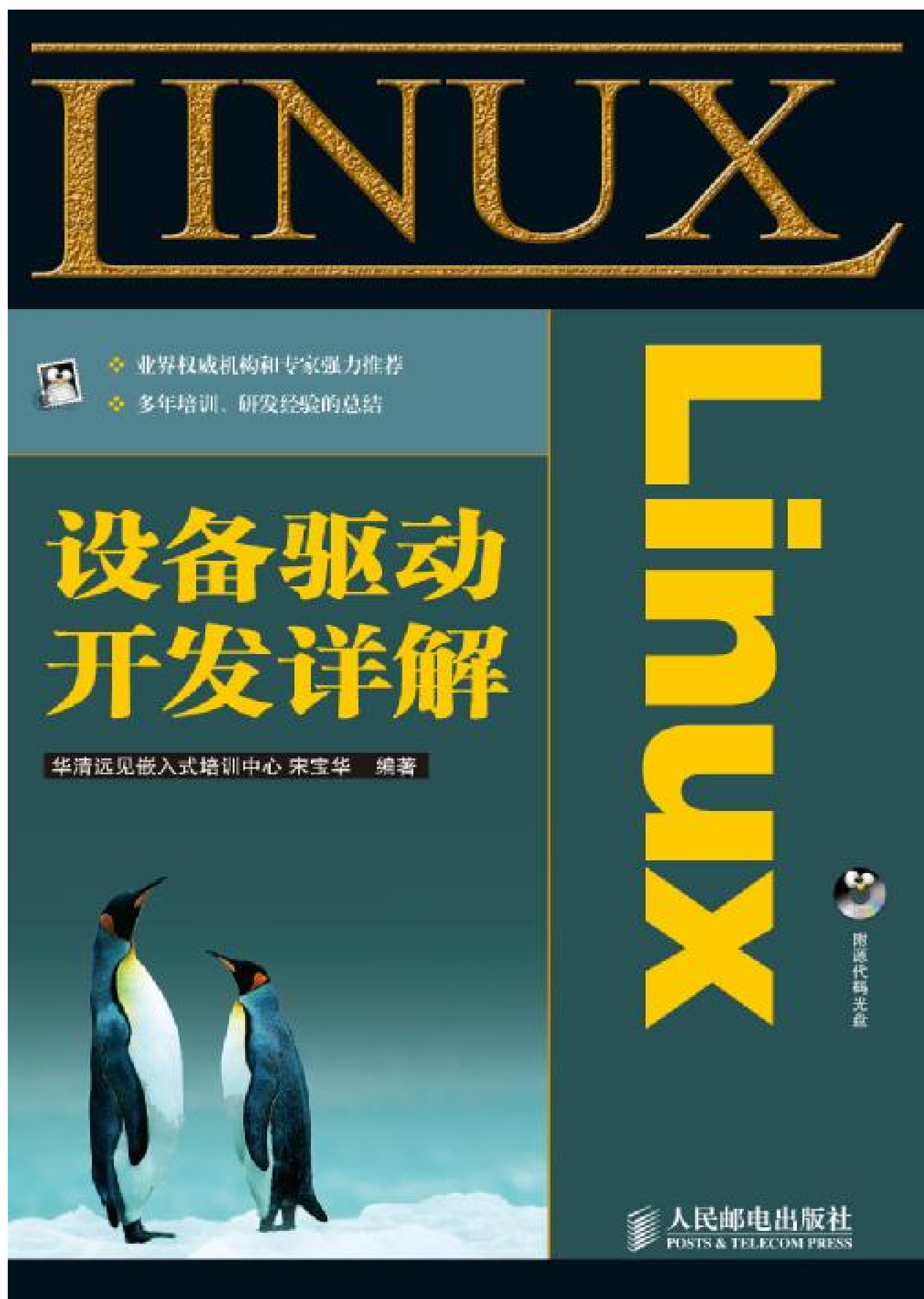

LINUX
业界权威机构和专家强力推荐
多年培训、研发经验的总结
设备驱动开发详解
Linux
华清远见嵌入式培训中心 宋宝华 编著
附源代码光盘
人民邮电出版社
POSTS & TELECOM PRESS

第 13 章　Linux 块设备驱动

华清远见

块设备是与字符设备并列的概念，这两类设备在 Linux 中驱动的结构有较大差异，总体而言，块设备驱动比字符设备驱动要复杂得多，在 I/O 操作上表现出极大的不同，缓冲、I/O 调度、请求队列等都是与块设备驱动相关的概念。本章将详细讲解 Linux 块设备驱动的编程方法。

13.1 节讲解块设备 I/O 操作的特点，分析字符设备与块设备在 I/O 操作上的差异。

13.2 节从整体上描述 Linux 块设备驱动的结构，分析主要的数据结构、函数及其关系。

13.3～13.5 节分别讲解块设备驱动模块加载与卸载、打开与释放和 ioctl()函数。

13.6 节非常重要，讲述了块设备 I/O 操作所依赖的请求队列的概念及用法。

13.2 节与 13.3～13.6 节是整体与部分的关系，13.2～13.6 节与 13.7 节是迭代递进的关系。

13.7 节在 13.1～13.6 节讲解内容的基础上，总结 Linux 下块设备的读写流程，而 13.7 节则讲解了块设备驱动的具体实例，即 RamDisk 的驱动。

13.1 块设备的 I/O 操作特点

字符设备与块设备 I/O 操作的不同如下。

（1）块设备只能以块为单位接受输入和返回输出，而字符设备则以字节为单位。大多数设备是字符设备，因为它们不需要缓冲而且不以固定块大小进行操作。

（2）块设备对于 I/O 请求有对应的缓冲区，因此它们可以选择以什么顺序进行响应，字符设备无须缓冲且被直接读写。对于存储设备而言调整读写的顺序作用巨大，因为在读写连续的扇区比分离的扇区更快。

（3）字符设备只能被顺序读写，而块设备可以随机访问。虽然块设备可随机访问，但是对于磁盘这类机械设备而言，顺序地组织块设备的访问可以提高性能，如图 13.1 所示，对扇区 1、10、3、2 的请求被调整为对扇区 1、2、3、10 的请求。而对 SD 卡、RamDisk 等块设备而言，不存在机械上的原因，进行这样的调整没有必要。

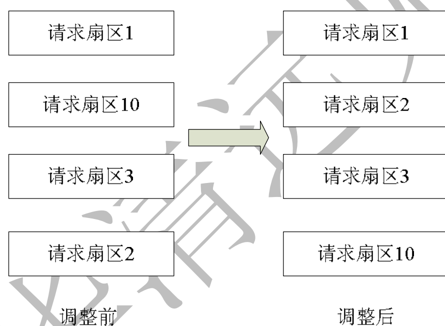


图 13.1　调整块设备 I/O 操作的顺序

13.2 Linux 块设备驱动结构

13.2.1　block_device_operations 结构体

在块设备驱动中，有 1 个类似于字符设备驱动中 file_operations 结构体的 block_device_operations 结构体，它是对块设备操作的集合，定义如代码清单 13.1 所示。

代码清单 13.1　block_device_operations 结构体

```
1  struct block_device_operations
2  {
3    int(*open)(struct inode *, struct file*);  //打开
```

```
    4   int(*release)(struct inode *, struct file*);  //释放
    5   int(*ioctl)(struct inode *, struct file *, unsigned, unsigned long);
//ioctl
    6   long(*unlocked_ioctl)(struct file *, unsigned, unsigned long);
    7   long(*compat_ioctl)(struct file *, unsigned, unsigned long);
    8    int(*direct_access)(struct block_device *, sector_t, unsigned
long*);
    9   int(*media_changed)(struct gendisk*);  //介质被改变?
    10  int(*revalidate_disk)(struct gendisk*);  //使介质有效
    11  int(*getgeo)(struct block_device *, struct hd_geometry*);//填充驱
动器信息
    12  struct module *owner; //模块拥有者
    13 };
```

下面对其主要的成员函数进行分析。

1. 打开和释放

```
int (*open)(struct inode *inode, struct file *filp);
int (*release)(struct inode *inode, struct file *filp);
```

与字符设备驱动类似，当设备被打开和关闭时将调用它们。

2. IO 控制

```
int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,
unsigned long arg);
```

上述函数是 ioctl()系统调用的实现，块设备包含大量的标准请求，这些标准请求由 Linux 块设备层处理，因此大部分块设备驱动的 ioctl()函数相当短。

3. 介质改变

```
int (*media_changed) (struct gendisk *gd);
```

被内核调用来检查是否驱动器中的介质已经改变，如果是，则返回一个非 0 值，否则返回 0。这个函数仅适用于支持可移动介质的驱动器，通常需要在驱动中增加一个表示介质状态是否改变的标志变量，非可移动设备的驱动不需要实现这个方法。

4. 使介质有效

```
int (*revalidate_disk) (struct gendisk *gd);
```

revalidate_disk()函数被调用来响应一个介质改变，它给驱动一个机会来进行必要的工作以使新介质准备好。

5. 获得驱动器信息

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

该函数根据驱动器的几何信息填充一个 hd_geometry 结构体，hd_geometry 结构体

包含磁头、扇区、柱面等信息。

6．模块指针

```
struct module *owner;
```

一个指向拥有这个结构体的模块的指针，它通常被初始化为 THIS_MODULE。

13.2.2　gendisk 结构体

在 Linux 内核中，使用 gendisk（通用磁盘）结构体来表示 1 个独立的磁盘设备（或分区），这个结构体的定义如代码清单 13.2 所示。

代码清单 13.2　gendisk 结构体

```
1  struct gendisk
2  {
3    int major; /* 主设备号 */
4    int first_minor;  /*第 1 个次设备号*/
5    int minors; /* 最大的次设备数，如果不能分区，则为 1*/
6    char disk_name[32]; /* 设备名称 */
7    struct hd_struct **part; /* 磁盘上的分区信息 */
8    struct block_device_operations *fops; /*块设备操作结构体*/
9    struct request_queue *queue;  /*请求队列*/
10   void *private_data;  /*私有数据*/
11   sector_t capacity; /*扇区数，512 字节为 1 个扇区*/
12
13   int flags;
14   char devfs_name[64];
15   int number;
16   struct device *driverfs_dev;
17   struct kobject kobj;
18
19   struct timer_rand_state *random;
20   int policy;
21
22   atomic_t sync_io; /* RAID */
23   unsigned long stamp;
24   int in_flight;
25   #ifdef CONFIG_SMP
26     struct disk_stats *dkstats;
27   #else
28     struct disk_stats dkstats;
29   #endif
30 };
```

major、first_minor 和 minors 共同表征了磁盘的主、次设备号，同一个磁盘的各个分区共享一个主设备号，而次设备号则不同。fops 为 block_device_operations，即上节描述的块设备操作集合。queue 是内核用来管理这个设备的 I/O 请求队列的指针。capacity 表明设备的容量，以 512 个字节为单位。private_data 可用于指向磁盘的任何私有数据，用法与字符设备驱动 file 结构体的 private_data 类似。

Linux 内核提供了一组函数来操作 gendisk，如下所示。

1．分配 gendisk

gendisk 结构体是一个动态分配的结构体，它需要特别的内核操作来初始化，驱动不能自己分配这个结构体，而应该使用下列函数来分配 gendisk：

```
struct gendisk *alloc_disk(int minors);
```

minors 参数是这个磁盘使用的次设备号的数量，一般也就是磁盘分区的数量，此后 minors 不能被修改。

2．增加 gendisk

gendisk 结构体被分配之后，系统还不能使用这个磁盘，需要调用如下函数来注册这个磁盘设备。

```
void add_disk(struct gendisk *gd);
```

特别要注意的是对 add_disk()的调用必须发生在驱动程序的初始化工作完成并能响应磁盘的请求之后。

3．释放 gendisk

当不再需要一个磁盘时，应当使用如下函数释放 gendisk。

```
void del_gendisk(struct gendisk *gd);
```

4．gendisk 引用计数

gendisk 中包含一个 kobject 成员，因此，它是一个可被引用计数的结构体。通过 get_disk()和 put_disk()函数可用来操作引用计数，这个工作一般不需要驱动亲自做。通常对 del_gendisk()的调用会去掉 gendisk 的最终引用计数，但是这一点并不是必须的。因此，在 del_gendisk()被调用后，这个结构体可能继续存在。

5．设置 gendisk 容量

```
void set_capacity(struct gendisk *disk, sector_t size);
```

块设备中最小的可寻址单元是扇区，扇区大小一般是 2 的整数倍，最常见的大小是 512 字节。扇区的大小是设备的物理属性，扇区是所有块设备的基本单元，块设备无法对比它还小的单元进行寻址和操作，不过许多块设备能够一次就传输多个扇区。虽然大多数块设备的扇区大小都是 512 字节，不过其他大小的扇区也很常见，比如，很多 CD-ROM 盘的扇区都是 2KB。

不管物理设备的真实扇区大小是多少，内核与块设备驱动交互的扇区都以 512 字节为单位。因此，set_capacity()函数也以 512 字节为单位。

13.2.3　request 与 bio 结构体

1．请求

在 Linux 块设备驱动中，使用 request 结构体来表征等待进行的 I/O 请求，这个结构体的定义如代码清单 13.3 所示。

代码清单 13.3　request 结构体

```
1  struct request
2  {
3    struct list_head queuelist; /*链表结构*/
4    unsigned long flags; /* REQ_ */
5
6    sector_t sector; /* 要传送输的下一个扇区 */
7    unsigned long nr_sectors; /*要传送的扇区数目*/
8    /*当前要传送的扇区数目*/
9    unsigned int current_nr_sectors;
10
11   sector_t hard_sector; /*要完成的下一个扇区*/
12   unsigned long hard_nr_sectors; /*要被完成的扇区数目*/
13   /*当前要被完成的扇区数目*/
14   unsigned int hard_cur_sectors;
15
16   struct bio *bio; /*请求的 bio 结构体的链表*/
17   struct bio *biotail; /*请求的 bio 结构体的链表尾*/
18
19   void *elevator_private;
20
21   unsigned short ioprio;
22
23   int rq_status;
24   struct gendisk *rq_disk;
25   int errors;
26   unsigned long start_time;
27
28   /*请求在物理内存中占据的不连续的段的数目，scatter/gather 列表的尺寸*/
29   unsigned short nr_phys_segments;
30
31   /*与 nr_phys_segments 相同，但考虑了系统 I/O MMU 的 remap */
32   unsigned short nr_hw_segments;
33
34   int tag;
35   char *buffer; /*传送的缓冲，内核虚拟地址*/
36
37   int ref_count; /* 引用计数 */
38   ...
39 };
```

request 结构体的主要成员包括：

```
sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;
```

上述 3 个成员标识还未完成的扇区，hard_sector 是第一个尚未传输的扇区，hard_nr_sectors 是尚待完成的扇区数，hard_cur_sectors 是当前 I/O 操作中待完成的扇区数。这些成员只用于内核块设备层，驱动不应当使用它们，如下所示：

```
sector_t sector;
unsigned long nr_sectors;
```

```
unsigned int current_nr_sectors;
```

驱动中会经常与这 3 个成员打交道，这 3 个成员在内核和驱动交互中发挥着重大作用。它们以 512 字节大小为一个扇区，如果硬件的扇区大小不是 512 字节，则需要进行相应的调整。例如，如果硬件的扇区大小是 2048 字节，则在进行硬件操作之前，需要用 4 来除起始扇区号。

hard_sector 、 hard_nr_sectors 、 hard_cur_sectors 与 sector 、 nr_sectors 、 current_nr_sectors 之间可认为是“副本”关系。

```
struct bio *bio;
```

bio 是这个请求中包含的 bio 结构体的链表，驱动中不宜直接存取这个成员，而应该使用后文将介绍的 rq_for_each_bio()。

```
char *buffer;
```

指向缓冲区的指针，数据应当被传送到或者来自这个缓冲区，这个指针是一个内核虚拟地址，可被驱动直接引用。

```
unsigned short nr_phys_segments;
```

该值表示相邻的页被合并后，这个请求在物理内存中占据的段的数目。

如果设备支持分散/聚集（SG，scatter/gather）操作，可依据此字段申请 sizeof(scatterlist)* nr_phys_segments 的内存，并使用下列函数进行 DMA 映射：

```
int blk_rq_map_sg(request_queue_t) *q, struct request *req,
            struct scatterlist *sglist;
```

该函数与 dma_map_sg()类似，它返回 scatterlist 列表入口的数量。

```
struct list_head queuelist;
```

用于链接这个请求到请求队列的链表结构，blkdev_dequeue_request()可用于从队列中移除请求。

使用如下宏可以从 request 获得数据传送的方向。

```
rq_data_dir(struct request *req);
```

0 返回值表示从设备中读，非 0 返回值表示向设备写。

2．请求队列

一个块请求队列是一个块 I/O 请求的队列，其定义如代码清单 13.4。

代码清单 13.4　request 队列结构体

```
1  struct request_queue
2  {
3    ...
4    /* 保护队列结构体的自旋锁 */
5    spinlock_t __queue_lock;
6    spinlock_t *queue_lock;
7
8    /* 队列 kobject */
9    struct kobject kobj;
10
```

```
11   /* 队列设置 */
12   unsigned long nr_requests; /* 最大的请求数量 */
13   unsigned int nr_congestion_on;
14   unsigned int nr_congestion_off;
15   unsigned int nr_batching;
16
17   unsigned short max_sectors;  /* 最大的扇区数 */
18   unsigned short max_hw_sectors;
19   unsigned short max_phys_segments; /* 最大的段数 */
20   unsigned short max_hw_segments;
21   unsigned short hardsect_size;  /* 硬件扇区尺寸 */
22   unsigned int max_segment_size;  /* 最大的段尺寸 */
23
24   unsigned long seg_boundary_mask; /* 段边界掩码 */
25   unsigned int dma_alignment;  /* DMA 传送的内存对齐限制 */
26
27   struct blk_queue_tag *queue_tags;
28
29   atomic_t refcnt; /* 引用计数 */
30
31   unsigned int in_flight;
32
33   unsigned int sg_timeout;
34   unsigned int sg_reserved_size;
35   int node;
36
37   struct list_head drain_list;
38
39   struct request *flush_rq;
40   unsigned char ordered;
41 };
```

请求队列跟踪等候的块 I/O 请求，它存储用于描述这个设备能够支持的请求的类型信息、它们的最大大小、多少不同的段可进入一个请求、硬件扇区大小、对齐要求等参数，其结果是：如果请求队列被配置正确了，它不会交给该设备一个不能处理的请求。

请求队列还实现一个插入接口，这个接口允许使用多个 I/O 调度器，I/O 调度器（也称电梯）的工作是以最优性能的方式向驱动提交 I/O 请求。大部分 I/O 调度器累积批量的 I/O 请求，并将它们排列为递增（或递减）的块索引顺序后提交给驱动。进行这些工作的原因在于，对于磁头而言，当给定顺序排列的请求时，可以使得磁盘顺序地从一头到另一头工作，非常像一个满载的电梯，在一个方向移动直到所有它的“请求”被满足。

另外，I/O 调度器还负责合并邻近的请求，当一个新 I/O 请求被提交给调度器后，它会在队列里搜寻包含邻近扇区的请求。如果找到一个，并且如果结果的请求不是太大，调度器将合并这两个请求。

对磁盘等块设备进行 I/O 操作顺序的调度类似于电梯的原理，先服务完上楼的乘客，再服务下楼的乘客效率会更高，而顺序响应用户的请求则电梯会无序地忙乱。

Linux 2.6 内核包含 4 个 I/O 调度器，它们分别是 No-op I/O scheduler、Anticipatory I/O scheduler、Deadline I/O scheduler 与 CFQ I/O scheduler。

Noop I/O scheduler 是一个简化的调度程序，它只作最基本的合并与排序。

Anticipatory I/O scheduler 是当前内核中默认的 I/O 调度器，它拥有非常好的性能，在 Linux 2.5 内核中它就相当引人注意。在与 Linux 2.4 内核进行的对比测试中，在 Linux 2.4 内核中多项以分钟为单位完成的任务，它则是以秒为单位来完成的，正因为如此它成为目前 Linux 2.6 内核中默认的 I/O 调度器。Anticipatory I/O scheduler 的缺点是比较庞大与复杂，在一些特殊的情况下，特别是在数据吞吐量非常大的数据库系统中它会变得比较缓慢。

Deadline I/O scheduler 是针对 Anticipatory I/O scheduler 的缺点进行改善而来的，表现出的性能几乎与 Anticipatory I/O scheduler 一样好，但是比 Anticipatory 小巧。

CFQ I/O scheduler 为系统内的所有任务分配相同的带宽，提供一个公平的工作环境，它比较适合桌面环境。事实上在测试中它也有不错的表现，mplayer、xmms 等多媒体播放器与它配合的相当好，回放平滑，几乎没有因访问磁盘而出现的跳帧现象。

内核 block 目录中的 noop-iosched.c、as-iosched.c、deadline-iosched.c 和 cfq-iosched.c 文件分别实现了上述调度算法。

可以通过给 kernel 添加启动参数，选择使用的 IO 调度算法，如：

```
    kernel elevator=deadline
```

（1）初始化请求队列。

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

该函数的第一个参数是请求处理函数的指针，第二个参数是控制访问队列权限的自旋锁，这个函数会发生内存分配的行为，它可能会失败，因此一定要检查它的返回值。这个函数一般在块设备驱动的模块加载函数中调用。

（2）清除请求队列。

```
    void blk_cleanup_queue(request_queue_t * q);
```

这个函数完成将请求队列返回给系统的任务，一般在块设备驱动模块卸载函数中调用。

而 blk_put_queue()宏则定义为：

```
    #define blk_put_queue(q) blk_cleanup_queue((q))
```

（3）分配“请求队列”。

```
    request_queue_t *blk_alloc_queue(int gfp_mask);
```

对于 Flash、RAM 盘等完全随机访问的非机械设备，并不需要进行复杂的 I/O 调度，这个时候，应该使用上述函数分配一个“请求队列”，并使用如下函数来绑定请求队列和“制造请求”函数。

```
    void blk_queue_make_request(request_queue_t  *  q,  make_request_fn  *
mfn);
```

在 13.6.2 节我们会看到，这种方式分配的“请求队列”实际上不包含任何请求，所以给其加上引号。

（4）提取请求。

```
struct request *elv_next_request(request_queue_t *queue);
```

上述函数用于返回下一个要处理的请求（由 I/O 调度器决定），如果没有请求则返回 NULL。elv_next_request()不会清除请求，它仍然将这个请求保留在队列上，但是标识它为活动的，这个标识将阻止 I/O 调度器合并其他的请求到已开始执行的请求。因为 elv_next_request()不从队列里清除请求，因此连续调用它两次，两次会返回同一个请求结构体。

（5）去除请求。

```
void blkdev_dequeue_request(struct request *req);
```

上述函数从队列中去除一个请求。如果驱动中同时从同一个队列中操作了多个请求，它必须以这样的方式将它们从队列中去除。

如果需要将一个已经出列的请求归还到队列中，可以进行以下调用：

```
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

另外，块设备层还提供了一套函数，这些函数可被驱动用来控制一个请求队列的操作，主要包括以下操作。

（6）启停请求队列。

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

如果块设备到达不能处理等候的命令的状态，应调用 blk_stop_queue()来告知块设备层。之后，请求函数将不被调用，除非再次调用 blk_start_queue()将设备恢复到可处理请求的状态。

（7）参数设置。

```
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
```

这些函数用于设置描述块设备可处理的请求的参数。blk_queue_max_sectors()描述任一请求可包含的最大扇区数，默认值为 255；blk_queue_max_phys_segments()和 blk_queue_max_hw_segments()都控制一个请求中可包含的最大物理段（系统内存中不相邻的区），blk_queue_max_hw_segments()考虑了系统 I/O 内存管理单元的重映射，这两个参数缺省都是 128。blk_queue_max_segment_size 告知内核请求段的最大字节数，缺省值为 65,536。

（8）通告内核。

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
```

该函数用于告知内核块设备执行 DMA 时可使用的最高物理地址 dma_addr，如果一个请求包含超出这个限制的内存引用，系统将会给这个操作分配一个“反弹”缓冲

区。这种方式的代价昂贵，因此应尽量避免使用。

可以给 dma_addr 参数提供任何可能的值或使用预先定义的宏，如 BLK_BOUNCE_HIGH（对高端内存页使用反弹缓冲区）、BLK_BOUNCE_ISA（驱动只可在 16MB 的 ISA 区执行 DMA）或者 BLK_BOUCE_ANY（驱动可在任何地址执行 DMA），缺省值是 BLK_BOUNCE_HIGH。

```
    blk_queue_segment_boundary(request_queue_t  *queue,  unsigned  long mask);
```

如果我们正在驱动编写的设备无法处理跨越一个特殊大小内存边界的请求，应该使用这个函数来告知内核这个边界。例如，如果设备处理跨 4MB 边界的请求有困难，应该传递一个 0x3fffff 掩码，缺省的掩码是 0xffffffff（对应 4GB 边界）。

```
    void blk_queue_dma_alignment(request_queue_t *queue, int mask);
```

告知内核块设备施加于 DMA 传送的内存对齐限制，所有请求都匹配这个对齐，缺省的屏蔽是 0x1ff，它导致所有的请求被对齐到 512 字节边界。

```
    void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);
```

该函数告知内核块设备硬件扇区的大小，所有由内核产生的请求都是这个大小的倍数并且被正确对界。但是，内核块设备层和驱动之间的通信还是以 512 字节扇区为单位进行。

3．块 I/O

通常一个 bio 对应一个 I/O 请求，代码清单 13.5 给出了 bio 结构体的定义。I/O 调度算法可将连续的 bio 合并成一个请求。所以，一个请求可以包含多个 bio。

代码清单 13.5　bio 结构体

```
1  struct bio
2  {
3    sector_t bi_sector; /* 要传输的第一个扇区 */
4    struct bio *bi_next; /* 下一个 bio */
5    struct block_device    *bi_bdev;
6    unsigned long bi_flags; /* 状态、命令等 */
7    unsigned long bi_rw; /* 低位表示 READ/WRITE，高位表示优先级*/
8
9    unsigned short bi_vcnt; /* bio_vec 数量 */
10   unsigned short bi_idx; /* 当前 bvl_vec 索引 */
11
12   /*不相邻的物理段的数目*/
13   unsigned short bi_phys_segments;
14
15   /*物理合并和 DMA remap 合并后不相邻的物理段的数目*/
16   unsigned short bi_hw_segments;
17
18   unsigned int bi_size; /* 以字节为单位所需传输的数据大小 */
19
20   /* 为了明了最大的 hw 尺寸，我们考虑这个 bio 中第一个和最后一个
21    虚拟的可合并的段的尺寸 */
22   unsigned int bi_hw_front_size;
23   unsigned int bi_hw_back_size;
24
25   unsigned int bi_max_vecs; /* 我们能持有的最大 bvl_vecs 数 */
26
27   struct bio_vec *bi_io_vec; /* 实际的 vec 列表 */
28
```

```
    29  bio_end_io_t *bi_end_io;
    30  atomic_t bi_cnt;
    31
    32  void *bi_private;
    33
    34  bio_destructor_t *bi_destructor; /* destructor */
    35 };
```

下面我们对其中的核心成员进行分析：

```
sector_t bi_sector;
```

标识这个 bio 要传送的第一个（512 字节）扇区。

```
unsigned int bi_size;
```

被传送的数据大小，以字节为单位，驱动中可以使用 bio_sectors(bio)宏获得以扇区为单位的大小。

```
unsigned long bi_flags;
```

一组描述 bio 的标志，如果这是一个写请求，最低有效位被置位，可以使用 bio_data_dir(bio)宏来获得读写方向。

```
unsigned short bio_phys_segments;
unsigned short bio_hw_segments;
```

分别表示包含在这个 BIO 中要处理的不连续的物理内存段的数目和考虑 DMA 重映像后的不连续的内存段的数目。

bio 的核心是一个称为 bi_io_vec 的数组，它由 bio_vec 结构体组成，bio_vec 结构体的定义如代码清单 13.6 所示。

代码清单 13.6 bio_vec 结构体

```
1 struct bio_vec
2 {
3  struct page *bv_page;   /* 页指针 */
4  unsigned int bv_len;  /* 传输的字节数 */
5  unsigned int bv_offset; /* 偏移位置 */
6 };
```

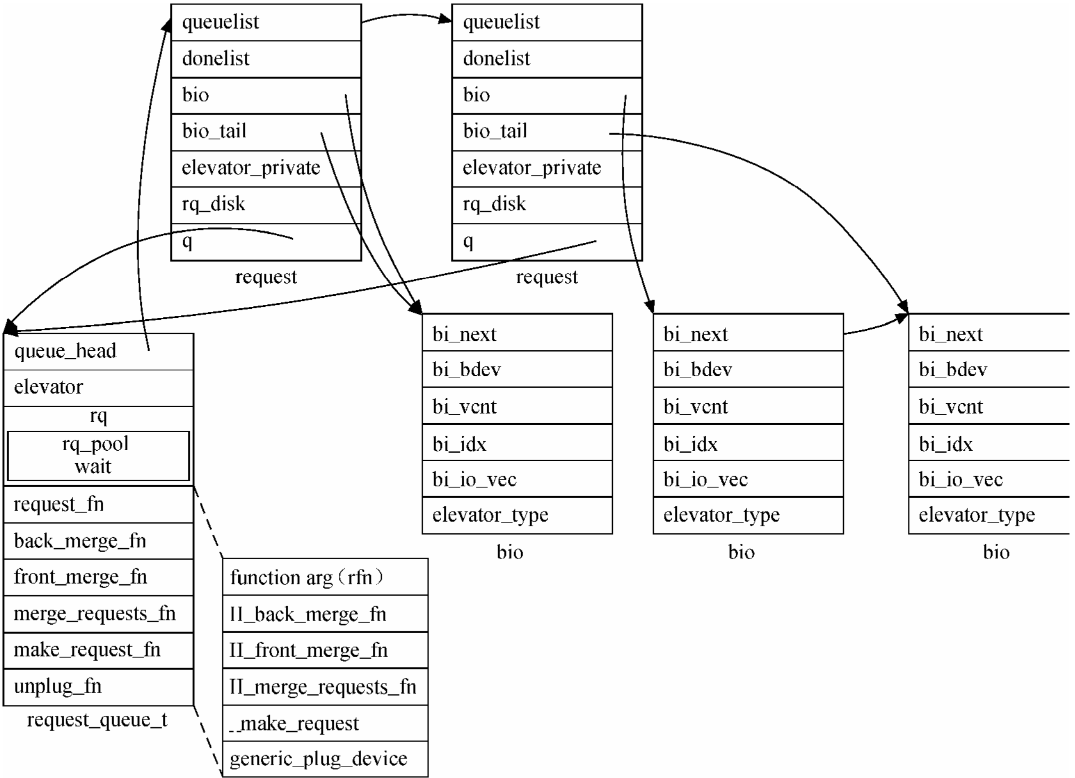
我们不应该直接访问 bio 的 bio_vec 成员，而应该使用 bio_for_each_segment()宏来进行这项工作，可以用这个宏循环遍历整个 bio 中的每个段，这个宏的定义如代码清单 13.7 所示。

代码清单 13.7 bio_for_each_segment 宏

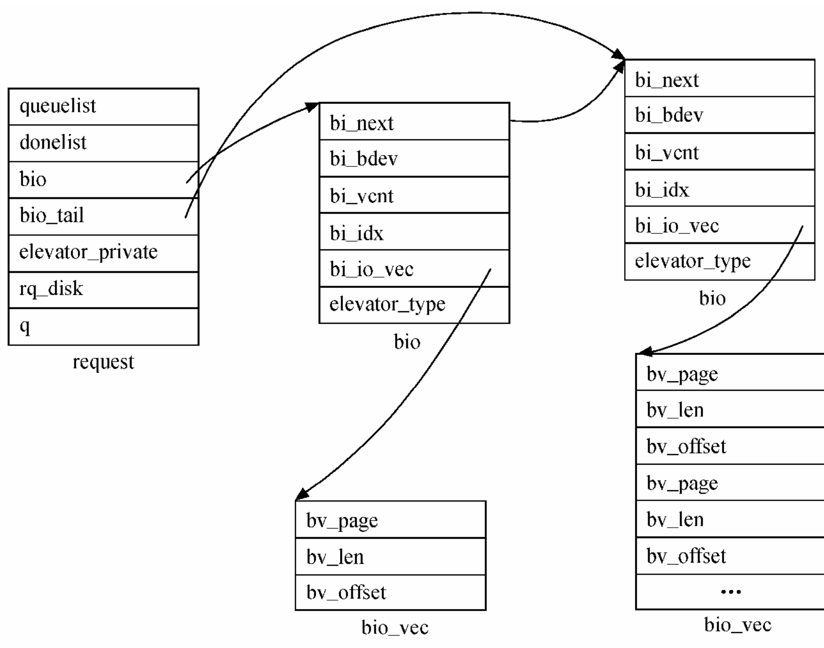
```
1  #define  _  _bio_for_each_segment(bvl,  bio,  i,  start_idx)
\
2      for (bvl = bio_iovec_idx((bio), (start_idx)), i = (start_idx);
\
3          i < (bio)->bi_vcnt; \
4          bvl++, i++)
5
6 #define bio_for_each_segment(bvl, bio, i) \
```

```
7             __bio_for_each_segment(bvl, bio, i, (bio)->bi_idx
```

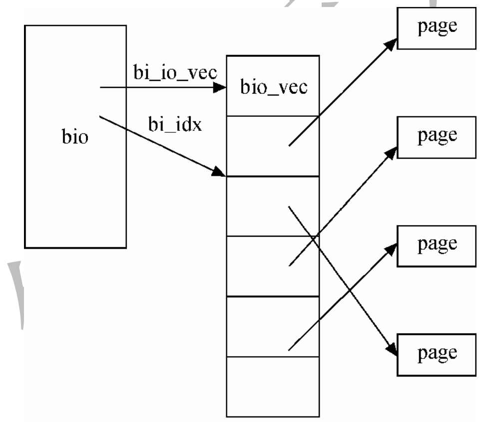
图 13.2（a）所示为 request 队列、request 与 bio 数据结构之间的关系，13.2（b）所示为 request、bio 和 bio_vec 数据结构之间的关系，13.2（c）所示为 bio 与 bio_vec 数据结构之间的关系，因此整个图 13.2 递归地呈现了 request 队列、request、bio 和 bio_vec 这 4 个结构体之间的关系。



（a）request 与 bio



（b）request、bio 和 bio_vec



（c） bio 与 bio_vec

图 13.2 request 队列、request、bio 和 bio_vec 结构体之间的关系

内核还提供了一组函数（宏）用于操作 bio：

```
int bio_data_dir(struct bio *bio);
```

这个函数可用于获得数据传输的方向是 READ 还是 WRITE。

```
struct page *bio_page(struct bio *bio) ;
```

这个函数可用于获得目前的页指针。

```
int bio_offset(struct bio *bio) ;
```

这个函数返回操作对应的当前页内的偏移，通常块 I/O 操作本身就是页对齐的。

```
int bio_cur_sectors(struct bio *bio) ;
```

这个函数返回当前 bio_vec 要传输的扇区数。

```
char *bio_data(struct bio *bio) ;
```

这个函数返回数据缓冲区的内核虚拟地址。

```
char *bvec_kmap_irq(struct bio_vec *bvec, unsigned long *flags) ;
```

这个函数返回一个内核虚拟地址，这个地址可用于存取被给定的 bio_vec 入口指向的数据缓冲区。它也会屏蔽中断并返回一个原子 kmap，因此，在 bvec_kunmap_irq()被调用以前，驱动不应该睡眠。

```
void bvec_kunmap_irq(char *buffer, unsigned long *flags);
```

这个函数是 bvec_kmap_irq()函数的“反函数”，它撤销 bvec_kmap_irq()创建的映射。

```
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
```

这个函数是对 bvec_kmap_irq()的包装，它返回给定的 bio 的当前 bio_vec 入口的映射。

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
```

这个函数通过 kmap_atomic()获得返回给定 bio 的第 i 个缓冲区的虚拟地址。

```
void __bio_kunmap_atomic(char *addr, enum km_type type);
```

这个函数返还由__bio_kmap_atomic()获得的内核虚拟地址。

另外，对 bio 的引用计数通过如下函数完成：

```
void bio_get(struct bio *bio);  //引用 bio
void bio_put(struct bio *bio);  //释放对 bio 的引用
```

13.2.4 块设备驱动注册与注销

块设备驱动中的第一个工作通常是注册它们自己到内核，完成这个任务的函数是 register_ blkdev()，其原型为：

```
int register_blkdev(unsigned int major, const char *name);
```

major 参数是块设备要使用的主设备号，name 为设备名，它会在/proc/devices 中被显示。如果 major 为 0，内核会自动分配一个新的主设备号，register_blkdev()函数的返回值就是这个主设备号。如果 register_blkdev()返回一个负值，表明发生了一个错误。

与 register_blkdev()对应的注销函数是 unregister_blkdev()，其原型为：

```
int unregister_blkdev(unsigned int major, const char *name);
```

这里，传递给 register_blkdev()的参数必须与传递给 register_blkdev()的参数匹配，否则这个函数返回-EINVAL。

值得一提的是，在 Linux 2.6 内核中，对 register_blkdev()的调用完全是可选的，register_blkdev()的功能已随时间正在减少，这个调用最多只完成两件事。

① 如果需要，分配一个动态主设备号。

② 在/proc/devices 中创建一个入口。

在将来的内核中，register_blkdev()可能会被去掉。但是目前的大部分驱动仍然调用它。代码清单 13.8 给出了一个块设备驱动注册的模板。

代码清单 13.8 块设备驱动注册模板

```
1 xxx_major = register_blkdev(xxx_major, "xxx");
2 if (xxx_major <= 0) //注册失败
3 {
4   printk(KERN_WARNING "xxx: unable to get major number\n");
5   return -EBUSY;
6 }
```

13.3 Linux 块设备驱动的模块加载与卸载

在块设备驱动的模块加载函数中通常需要完成如下工作。

① 分配、初始化请求队列，绑定请求队列和请求函数。

② 分配、初始化 gendisk，给 gendisk 的 major、fops、queue 等成员赋值，最后添加 gendisk。

③ 注册块设备驱动。

代码清单 13.9 和 13.10 分别给出了使用 blk_alloc_queue()分配请求队列并使用 blk_queue_ make_request()绑定"请求队列"和"制造请求"的函数，以及使用 blk_init_queue()初始化请求队列并绑定请求队列与请求处理函数两种不同情况下的块设备驱动模块加载函数模板。

代码清单 13.9　块设备驱动的模块加载函数模板（使用 blk_alloc_queue）

```
1  static int __init xxx_init(void)
2  {
3    //分配 gendisk
4    xxx_disks = alloc_disk(1);
5    if (!xxx_disks)
6    {
7      goto out;
8    }
9
10   //块设备驱动注册
11   if (register_blkdev(XXX_MAJOR, "xxx"))
12   {
13     err = - EIO;
14     goto out;
15   }
16
17   // "请求队列" 分配
18   xxx_queue = blk_alloc_queue(GFP_KERNEL);
19   if (!xxx_queue)
20   {
21     goto out_queue;
22   }
23
24   blk_queue_make_request(xxx_queue, &xxx_make_request); //绑定"制造请求"函数
25   blk_queue_hardsect_size(xxx_queue, xxx_blocksize); //硬件扇区尺寸设置
26
27   //gendisk 初始化
```

```
28   xxx_disks->major = XXX_MAJOR;
29   xxx_disks->first_minor = 0;
30   xxx_disks->fops = &xxx_op;
31   xxx_disks->queue = xxx_queue;
32   sprintf(xxx_disks->disk_name, "xxx%d", i);
33   set_capacity(xxx_disks, xxx_size); //xxx_size 以 512bytes 为单位
34   add_disk(xxx_disks); //添加 gendisk
35
36   return 0;
37   out_queue: unregister_blkdev(XXX_MAJOR, "xxx");
38   out: put_disk(xxx_disks);
39   blk_cleanup_queue(xxx_queue);
40
41   return  - ENOMEM;
42 }
```

代码清单 13.10　块设备驱动的模块加载函数模板（使用 blk_init_queue）

```
1  static int __init xxx_init(void)
2  {
3    //块设备驱动注册
4    if (register_blkdev(XXX_MAJOR, "xxx"))
5    {
6      err =  - EIO;
7      goto out;
8    }
9
10   //请求队列初始化
11   xxx_queue = blk_init_queue(xxx_request, xxx_lock);
12   if (!xxx_queue)
13   {
14     goto out_queue;
15   }
16
17   blk_queue_hardsect_size(xxx_queue, xxx_blocksize); //硬件扇区尺寸
设置
18
19   //gendisk 初始化
20   xxx_disks->major = XXX_MAJOR;
21   xxx_disks->first_minor = 0;
22   xxx_disks->fops = &xxx_op;
23   xxx_disks->queue = xxx_queue;
24   sprintf(xxx_disks->disk_name, "xxx%d", i);
25   set_capacity(xxx_disks, xxx_size *2);
26   add_disk(xxx_disks); //添加 gendisk
27
28   return 0;
29   out_queue: unregister_blkdev(XXX_MAJOR, "xxx");
30   out: put_disk(xxx_disks);
31   blk_cleanup_queue(xxx_queue);
32
33   return  - ENOMEM;
34 }
```

在块设备驱动的模块卸载函数中完成与模块加载函数相反的工作。

① 清除请求队列。

② 删除 gendisk 和对 gendisk 的引用。

③ 删除对块设备的引用，注销块设备驱动。

代码清单 13.11 给出了块设备驱动的模块卸载函数的模板。

代码清单 13.11　块设备驱动的模块卸载函数模板

```
1  static void __exit xxx_exit(void)
2  {
3    if (bdev)
4    {
5      invalidate_bdev(xxx_bdev, 1);
6      blkdev_put(xxx_bdev);
7    }
8    del_gendisk(xxx_disks); //删除 gendisk
9    put_disk(xxx_disks);
10   blk_cleanup_queue(xxx_queue[i]); //清除请求队列
11   unregister_blkdev(XXX_MAJOR, "xxx");
12 }
```

13.4 块设备的打开与释放

块设备驱动的 open()和 release()函数并非是必须的，一个简单的块设备驱动可以不提供 open()和 release()函数。

块设备驱动的 open()函数和其字符设备驱动的对等体非常类似，都以相关的 inode 和 file 结构体指针作为参数。当一个节点引用一个块设备时，inode->i_bdev->bd_disk 包含一个指向关联 gendisk 结构体的指针。因此，类似于字符设备驱动，我们也可以将 gendisk 的 private_data 赋给 file 的 private_data，private_data 同样最好是指向描述该设备的设备结构体 xxx_dev 的指针，如代码清单 13.12 所示。

代码清单 13.12　在块设备的 open()函数中赋值 private_data

```
1 static int xxx_open(struct inode *inode, struct file *filp)
2 {
3   struct xxx_dev *dev = inode->i_bdev->bd_disk->private_data;
4   filp->private_data = dev; //赋值 file 的 private_data
5   ...
6   return 0;
7 }
```

在一个处理真实的硬件设备的驱动中，open()和 release()方法还应当设置驱动和硬件的状态，这些工作可能包括启停磁盘、加锁一个可移出设备和分配 DMA 缓冲等。

13.5 块设备驱动的 ioctl 函数

与字符设备驱动一样，块设备可以包含一个 ioctl()函数以提供对设备的 I/O 控制能力。实际上，高层的块设备层代码处理了绝大多数 ioctl()，因此，具体的块设备驱动中通常不再需要实现很多 ioctl 命令。

代码清单 13.13 给出的 ioctl()函数只实现一个命令 HDIO_GETGEO，用于获得磁盘的几何信息（geometry，指 CHS，即 Cylinder、Head、Sector/Track）。

代码清单 13.13　块设备驱动的 I/O 控制函数模板

```
    1  int xxx_ioctl(struct inode *inode, struct file *filp, unsigned int
cmd,
    2    unsigned long arg)
    3  {
    4    long size;
    5    struct hd_geometry geo;
    6    struct xxx_dev *dev = filp->private_data; //通过 file->private 获
得设备结构体
    7
    8    switch (cmd)
    9    {
    10     case HDIO_GETGEO:
    11       size = dev->size *(hardsect_size / KERNEL_SECTOR_SIZE);
    12       geo.cylinders = (size &~0x3f) >> 6;
    13       geo.heads = 4;
    14       geo.sectors = 16;
    15       geo.start = 4;
    16       if (copy_to_user((void __user*)arg, &geo, sizeof(geo)))
    17       {
    18         return  - EFAULT;
    19       }
    20       return 0;
    21   }
    22
    23   return  - ENOTTY; //不知道的命令
    24 }
```

13.6 块设备驱动的 I/O 请求处理

13.6.1　使用请求队列

块设备驱动请求函数的原型为：

```
void request(request_queue_t *queue);
```

这个函数不能由驱动自己调用，只有当内核认为是时候让驱动处理对设备的读写等操作时，它才调用这个函数。

请求函数可以在没有完成请求队列中的所有请求的情况下返回，甚至它一个请求不完成都可以返回。但是，对大部分设备而言，在请求函数中处理完所有请求后再返回通常是值得推荐的方法。代码清单 13.14 给出了一个简单的 request()函数的例子。

代码清单 13.14　块设备驱动请求函数例程

```
    1 static void xxx_request(request_queue_t *q)
    2 {
    3   struct request *req;
    4   while ((req = elv_next_request(q)) != NULL)
    5   {
    6     struct xxx_dev *dev = req->rq_disk->private_data;
    7     if (!blk_fs_request(req)) //不是文件系统请求
    8     {
    9       printk(KERN_NOTICE "Skip non-fs request\n");
    10      end_request(req, 0);//通知请求处理失败
    11      continue;
    12    }
    13        xxx_transfer(dev,  req->sector,  req->current_nr_sectors,
req->buffer,
    14      rq_data_dir(req)); //处理这个请求
    15    end_request(req, 1); //通知成功完成这个请求
    16  }
    17 }
    18
    19 //完成具体的块设备 I/O 操作
    20 static void xxx_transfer(struct xxx_dev *dev, unsigned long sector,
unsigned
    21  long nsect, char *buffer, int write)
    22 {
    23  unsigned long offset = sector * KERNEL_SECTOR_SIZE;
    24  unsigned long nbytes = nsect * KERNEL_SECTOR_SIZE;
    25  if ((offset + nbytes) > dev->size)
    26  {
    27      printk(KERN_NOTICE  "Beyond-end  write  (%ld  %ld)\n",  offset,
nbytes);
    28    return ;
    29  }
    30  if (write)
    31  {
    32    write_dev(offset, buffer, nbytes);  //向设备写 nbytes 个字节的数据
    33  }
    34  else
    35  {
    36    read_dev(offset, buffer, nbytes); //从设备读 nbytes 个字节的数据
    37  }
    38 }
```

上述代码第 4 行使用 elv_next_request()获得队列中第一个未完成的请求，

end_request()会将请求从请求队列中剥离。第 7 行判断请求是否为文件系统请求，如果不是，则直接清除，调用 end_ request()，传递给 end_request()的第二个参数为 0 意味着处理该请求失败。而第 15 行传递给 end_request()的第二个参数为 1 意味着该请求处理成功。

end_request()函数非常重要，其源代码如代码清单 13.15 所示。

代码清单 13.15　end_request()函数源代码

```
1  void end_request(struct request *req, int uptodate)
2  {
3
4               if    (!end_that_request_first(req,    uptodate,
req->hard_cur_sectors))
5    {
6      add_disk_randomness (req->rq_disk);
7      blkdev_dequeue_request (req);
8      end_that_request_last(req);
9    }
10 }
```

当设备已经完成一个 I/O 请求的部分或者全部扇区传输后，它必须通告块设备层，上述代码中的第 4 行完成这个工作。end_that_request_first()函数的原型为：

```
int  end_that_request_first(struct  request  *req,  int  success,  int
count);
```

这个函数告知块设备层，块设备驱动已经完成 count 个扇区的传送。end_that_request_first()的返回值是一个标志，指示是否这个请求中的所有扇区已经被传送。返回值为 0 表示所有的扇区已经被传送并且这个请求完成，之后，我们必须使用 blkdev_dequeue_request()来从队列中清除这个请求。最后，将这个请求传递给 end_that_request_last()函数。

```
void end_that_request_last(struct request *req);
```

end_that_request_last()通知所有正在等待这个请求完成的对象请求已经完成并回收这个请求结构体。

第 6 行的 add_disk_randomness()函数的作用是使用块 I/O 请求的定时来给系统的随机数池贡献熵，它不影响块设备驱动。但是，仅当磁盘的操作时间是真正随机的时候（大部分机械设备如此），才应该调用它。

代码清单 13.16 给出了一个更复杂的请求函数，它进行了 3 层遍历：遍历请求队列中的每个请求，遍历请求中的每个 bio，遍历 bio 中的每个段。

代码清单 13.16　请求函数遍历请求、bio 和段

```
1  static void xxx_full_request(request_queue_t *q)
2  {
3    struct request *req;
4    int sectors_xferred;
5    struct xxx_dev *dev = q->queuedata;
6    /* 遍历每个请求 */
7    while ((req = elv_next_request(q)) != NULL)
8    {
9      if (!blk_fs_request(req))
10     {
```

```
11        printk(KERN_NOTICE "Skip non-fs request\n");
12
13        end_request(req, 0);
14        continue;
15      }
16      sectors_xferred = xxx_xfer_request(dev, req);
17      if (!end_that_request_first(req, 1, sectors_xferred))
18      {
19        blkdev_dequeue_request(req);
20        end_that_request_last(req);
21      }
22    }
23 }
24 /* 请求处理 */
25 static int xxx_xfer_request(struct xxx_dev *dev, struct request *req)
26 {
27   struct bio *bio;
28   int nsect = 0;
29   /* 遍历请求中的每个 bio */
30   rq_for_each_bio(bio, req)
31   {
32     xxx_xfer_bio(dev, bio);
33     nsect += bio->bi_size / KERNEL_SECTOR_SIZE;
34   }
35   return nsect;
36 }
37 /* bio 处理 */
38 static int xxx_xfer_bio(struct xxx_dev *dev, struct bio *bio)
39 {
40   int i;
41   struct bio_vec *bvec;
42   sector_t sector = bio->bi_sector;
43
44   /* 遍历每一段 */
45   bio_for_each_segment(bvec, bio, i)
46   {
47     char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
48         xxx_transfer(dev,  sector,  bio_cur_sectors(bio),  buffer,
bio_data_dir(bio)
49        == WRITE);
50     sector += bio_cur_sectors(bio);
51     __bio_kunmap_atomic(bio, KM_USER0);
52   }
53   return 0;
54 }
```

图 13.3 所示为一个请求队列内 request、bio 以及 bio 中 segment 的层层遍历关系。

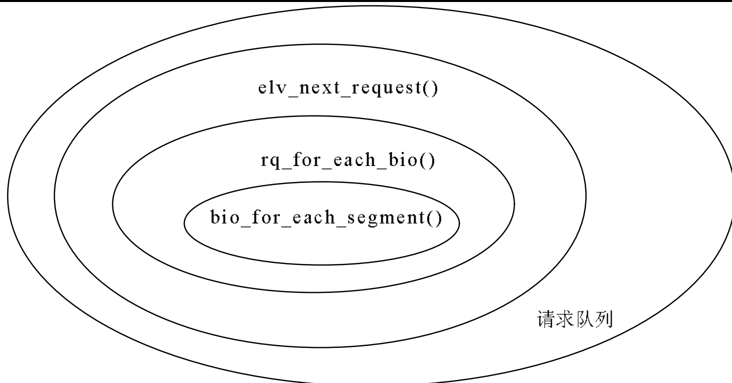


图 13.3　遍历一个请求队列

13.6.2　不使用请求队列

使用请求队列对于一个机械的磁盘设备而言的确有助于提高系统的性能，但是对于许多块设备，如数码相机的存储卡、RAM 盘等完全可真正随机访问的设备而言，无法从高级的请求队列逻辑中获益。对于这些设备，块层支持“无队列”的操作模式，为使用这个模式，驱动必须提供一个“制造请求”函数，而不是一个请求函数，“制造请求”函数的原型为：

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

上述函数的第一个参数仍然是“请求队列”，但是这个“请求队列”实际不包含任何请求。因此，“制造请求”函数的主要参数是 bio 结构体，这个 bio 结构体表示一个或多个要传送的缓冲区。“制造请求”函数或者直接进行传输，或者把请求重定向给其他设备。

在“制造请求”函数中处理 bio 的方式与 13.6.1 小节中讲解的完全一致，但是在处理完成后应该使用 bio_endio()函数通知处理结束，如下所示：

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

参数 bytes 是已经传送的字节数，它可以比这个 bio 所代表的字节数少，这意味着“部分完成”，同时 bio 结构体中的当前缓冲区指针需要更新。当设备进一步处理这个 bio 后，驱动应该再次调用 bio_endio()，如果不能完成这个请求，应指出一个错误，错误码赋值给 error 参数。

不管对应的 I/O 处理成功与否，“制造请求”函数都应该返回 0。如果“制造请求”函数返回一个非零值，bio 将被再次提交。

代码清单 13.17 所示为一个“制造请求”函数的例子。

代码清单 13.17　“制造请求”函数例程

```
1 static int xxx_make_request(request_queue_t *q, struct bio *bio)
2 {
3   struct xxx_dev *dev = q->queuedata;
4   int status;
5   status = xxx_xfer_bio(dev, bio); //处理 bio
6   bio_endio(bio, bio->bi_size, status); //通告结束
7   return 0;
8 }
```

为了使用无队列的 I/O 请求处理，驱动模块的加载函数应遵循代码清单 13.9

的模板而非代码清单 13.10 的模板，而使用请求队列时，驱动模块的加载函数应遵循代码清单 13.10 的模板。

13.7 实例 1：RamDisk 驱动

13.7.1 RamDisk 的硬件原理

RamDisk（RAM 盘）是一种模拟磁盘，其数据实际上存储在 RAM 中，它使用一部分内存空间来模拟出一个磁盘，以块设备的方式来访问这片内存，RamDisk 对应的设备文件一般为/dev/ram%d。

使用如下一组命令就可以创建并挂载 RamDisk：

```
mkdir /tmp/ramdisk0     //创建装载点
mke2fs /dev/ram0    //创建一个文件系统
mount /dev/ram0 /tmp/ramdisk0   //装载 RamDisk
```

其中，mke2f /dev/ram0 命令的执行会回馈如下信息：

```
mke2fs 1.14, 9-Jan-1999 for EXT2 FS 0.5b, 95/08/09
Linux ext2 filesystem format
Filesystem label=
1024 inodes, 4096 blocks
204 blocks (4.98%) reserved for the super user
First data block=1
Block size=1024 (log=0)
Fragment size=1024 (log=0)
1 block group
8192 blocks per group, 8192 fragments per group
1024 inodes per group
```

表明创建了 1 个 4MB 的块设备，共包含 4096 块，每块 1024 字节。

13.7.2 RamDisk 驱动模块的加载与卸载

RamDisk 驱动的模块加载函数完成的工作与 13.3 节给出的模板完全一致，由于 RAM 盘属于完全的随机设备，宜使用无队列的 I/O 处理方式，其驱动中实现了如图 13.4 所示的与块设备驱动模板对应的函数。

代码清单 13.18 给出了 RamDisk 设备驱动的模块加载与卸载函数，实现的功能与模块是一致的。

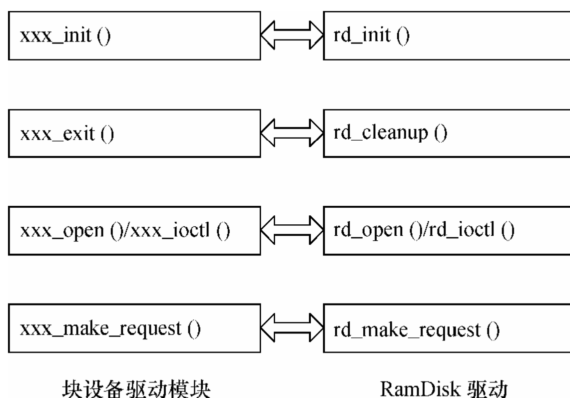


图 13.4 块设备驱动模板与 RamDisk 设备驱动的映射

代码清单 13.18 RamDisk 设备驱动的模块加载与卸载函数

```
1 static int __init rd_init(void)
2 {
3   int i;
4   int err =  - ENOMEM;
5   //调整块尺寸
6   if (rd_blocksize > PAGE_SIZE || rd_blocksize < 512 || (rd_blocksize
&
7     (rd_blocksize - 1)))
8   {
9     printk("RAMDISK: wrong blocksize %d, reverting to defaults\n",
rd_blocksize) ;
10
11    rd_blocksize = BLOCK_SIZE;
12  }
13  //分配 gendisk
14  for (i = 0; i < CONFIG_BLK_DEV_RAM_COUNT; i++)
15  {
16    rd_disks[i] = alloc_disk(1); //分配 gendisk
17    if (!rd_disks[i])
18      goto out;
19  }
20  //块设备注册
21  if (register_blkdev(RAMDISK_MAJOR, "ramdisk"))
22  //注册块设备
23  {
24    err =  - EIO;
25    goto out;
26  }
27
28  devfs_mk_dir("rd"); //创建 devfs 目录
29
30  for (i = 0; i < CONFIG_BLK_DEV_RAM_COUNT; i++)
31  {
32    struct gendisk *disk = rd_disks[i];
33    //分配并绑定请求队列与“制造请求”函数
34    rd_queue[i] = blk_alloc_queue(GFP_KERNEL);
35    if (!rd_queue[i])
```

```
36        goto out_queue;
37
38      blk_queue_make_request(rd_queue[i], &rd_make_request);   //绑定
“制造请求”函数
39      blk_queue_hardsect_size(rd_queue[i], rd_blocksize); //硬件扇区尺
寸设置
40
41      //初始化 gendisk
42      disk->major = RAMDISK_MAJOR;
43      disk->first_minor = i;
44      disk->fops = &rd_bd_op;
45      disk->queue = rd_queue[i];
46      disk->flags |= GENHD_FL_SUPPRESS_PARTITION_INFO;
47      sprintf(disk->disk_name, "ram%d", i);
48      sprintf(disk->devfs_name, "rd/%d", i);
49      set_capacity(disk, rd_size *2);
50      add_disk(rd_disks[i]); //添加 gendisk
51    }
52
53    // rd_size 以 KB 为单位
54    printk("RAMDISK driver initialized: "
55      "%d RAMDISKs of %dK size %d blocksize\n",
56      CONFIG_BLK_DEV_RAM_COUNT,rd_size, rd_blocksize);
57
58    return 0;
59    out_queue: unregister_blkdev(RAMDISK_MAJOR, "ramdisk");
60    out:
61    while (i--)
62    {
63      put_disk(rd_disks[i]);
64      blk_cleanup_queue(rd_queue[i]);
65    }
66    return err;
67  }
68
69  static void __exit rd_cleanup(void)
70  {
71    int i;
72
73    for (i = 0; i < CONFIG_BLK_DEV_RAM_COUNT; i++)
74    {
75      struct block_device *bdev = rd_bdev[i];
76      rd_bdev[i] = NULL;
77      if (bdev)
78      {
79        invalidate_bdev(bdev, 1);
80        blkdev_put(bdev);
81      }
82      del_gendisk(rd_disks[i]); //删除 gendisk
83      put_disk(rd_disks[i]); //释放对 gendisk 的引用
84      blk_cleanup_queue(rd_queue[i]); //清除请求队列
85    }
```

```
86   devfs_remove("rd");
87   unregister_blkdev(RAMDISK_MAJOR, "ramdisk"); //块设备注销
88 }
```

华清远见

13.7.3 RamDisk 设备驱动 block_device_operations 及成员函数

RamDisk 提供 block_device_operations 结构体中两个成员函数的实现：open()和 ioctl()，代码清单 13.19 给出了 RamDisk 设备驱动的 block_device_operations 结构体定义及 open()和 ioctl()结构。

代码清单 13.19 RamDisk 设备驱动 block_device_operations 结构体及成员函数

```
1  static struct block_device_operations rd_bd_op =
2  {
3    .owner = THIS_MODULE,
4    .open = rd_open,
5    .ioctl = rd_ioctl,
6  };
7
8  static int rd_open(struct inode *inode, struct file *filp)
9  {
10  unsigned unit = iminor(inode);//获得次设备号
11
12  if (rd_bdev[unit] == NULL) {
13      struct block_device *bdev = inode->i_bdev;//获得 block_device
结构体指针
14      struct address_space *mapping; //地址空间
15      unsigned bsize;
16      gfp_t gfp_mask;
17    /* 设置 inode 成员 */
18      inode = igrab(bdev->bd_inode);
19      rd_bdev[unit] = bdev;
20      bdev->bd_openers++;
21      bsize = bdev_hardsect_size(bdev);
22      bdev->bd_block_size = bsize;
23      inode->i_blkbits = blksize_bits(bsize);
24      inode->i_size = get_capacity(bdev->bd_disk)<<9;
25
26      mapping = inode->i_mapping;
27      mapping->a_ops = &ramdisk_aops;
28      mapping->backing_dev_info = &rd_backing_dev_info;
29      bdev->bd_inode_backing_dev_info = &rd_file_backing_dev_info;
30
31      gfp_mask = mapping_gfp_mask(mapping);
32      gfp_mask &= ~(__GFP_FS|__GFP_IO);
33      gfp_mask |= __GFP_HIGH;
34      mapping_set_gfp_mask(mapping, gfp_mask);
35  }
36
37  return 0;
38 }
39
40 static int rd_ioctl(struct inode *inode, struct file *file,
41           unsigned int cmd, unsigned long arg)
42 {
43  int error;
44  struct block_device *bdev = inode->i_bdev;
45
46  if (cmd != BLKFLSBUF) /* 不是 flush buffer cache 命令 */
47      return -ENOTTY;
```

```
48  /* 刷新 buffer cache */
49  error = -EBUSY;
50  down(&bdev->bd_sem);
51  if (bdev->bd_openers <= 2) {
52      truncate_inode_pages(bdev->bd_inode->i_mapping, 0);
53      error = 0;
54  }
55  up(&bdev->bd_sem);
56  return error;
57 }
```

13.7.4 RamDisk I/O 请求处理

鉴于 RamDisk 是一种完全随机设备，其驱动中宜使用"制造请求"函数而非请求函数，这个函数的实现如代码清单 13.20 所示。

代码清单 13.20 RamDisk 设备驱动的"制造请求"函数

```
1  static int rd_make_request(request_queue_t *q, struct bio *bio)
2  {
3    struct block_device *bdev = bio->bi_bdev;
4    struct address_space *mapping = bdev->bd_inode->i_mapping;
5    sector_t sector = bio->bi_sector;
6    unsigned long len = bio->bi_size >> 9;
7    int rw = bio_data_dir(bio);//数据传输方向: 读/写?
8    struct bio_vec *bvec;
9    int ret = 0, i;
10
11   if (sector + len > get_capacity(bdev->bd_disk))
12   //超过容量
13     goto fail;
14
15   if (rw == READA)
16     rw = READ;
17   //遍历每个段
18   bio_for_each_segment(bvec, bio, i)
19   {
20     ret |= rd_blkdev_pagecache_IO(rw, bvec, sector, mapping);
21     sector += bvec->bv_len >> 9;
22   }
23   if (ret)
24     goto fail;
25
26   bio_endio(bio, bio->bi_size, 0); //处理结束
27   return 0;
28   fail: bio_io_error(bio, bio->bi_size);
29   return 0;
30 }
```

13.8 实例 2：IDE 硬盘设备驱动

13.8.1 IDE 硬盘设备原理

IDE（Integrated Drive Electronics）接口，也就是集成驱动器电路接口，原名为 ATA（AT Attachment，AT 嵌入式）接口，其本意为将硬盘控制器与盘体集成在一起的硬盘

驱动器，经历了 ATA-1 到 ATA-7 以及 SATA-1 和 SATA-2 的发展历史。ATA-1 至 ATA-4 采用 40 芯排线缆，ATA-5 至 ATA-7 则采用 40 针 80 芯线缆，虽然线缆数量增加了，但是逻辑原理没有变，只是通过物理上的改变来达到改善 PCB 信号完整性的目的，它提供更多的地线并使信号线临近地线，从而减少电流回流的面积。SATA-1 和 SATA-2 与 ATA-1 至 ATA-7 相比，数据传输方式由并行转变为串行。

IDE 接口的硬件原理实际上非常简单，对 CPU 的外围总线进行简单扩展后就可外接 IDE 控制器，表 13.1 所示为 40 针 IDE 接口的引脚定义。

表 13.1　　　　IDE 接口的引脚定义

引脚	信号	信号描述	信号方向	引脚	信号	信号描述	信号方
1	RSET	复位	I	2	GND	地	I/O
3	DD7	数据位 7	I/O	4	DD8	数据位 8	I/O
5	DD6	数据位 6	I/O	6	DD9	数据位 9	I/O
7	DD5	数据位 5	I/O	8	DD10	数据位 10	I/O
9	DD4	数据位 4	I/O	10	DD11	数据位 11	I/O
11	DD3	数据位 3	I/O	12	DD12	数据位 12	I/O
13	DD2	数据位 2	I/O	14	DD13	数据位 13	I/O
15	DD1	数据位 1	I/O	16	DD14	数据位 14	I/O
17	DD0	数据位 0	I/O	18	DD15	数据位 15	I/O
19	GND	地		20	N.C	未用	
21	DMARQ	DMA 请求	O	22	GND	地	
23	$\overline{\text{IDOW}}$	写选通	I	24	GND	地	
25	$\overline{\text{DIOR}}$	读选通	I	26	GND	地	
27	IORDY	通道就绪	O	28	DPSYNC:CXEL	同步电缆选择	
29	$\overline{\text{DMACK}}$	DMA 应答	O	30	GND	地	
31	$\overline{\text{INTRQ}}$	中断请求	O	32	$\overline{\text{IOCS16}}$	16 位 I/O 片选	O
33	DA1	地址 1	I	34	$\overline{\text{PDIAG}}$	诊断完成	O
35	DA0	地址 0	I	36	DA2	地址 2	I
37	$\overline{\text{CS0}}$	片选 0	I	38	$\overline{\text{CS1}}$	片选 1	I
39	$\overline{\text{DASP}}$	驱动器状态指示	O	40	GND	地	

华清远见

IDE 控制器提供了一组寄存器，通过这些寄存器，主机能控制 IDE 驱动器的行为和查询其状态，表 13.2 所示 IDE 接口寄存器的定义。

表 13.2 IDE 接口寄存器定义

片选 1	片选 0	地址 2	地址 1	地址 0	读	写	位数
1	0	0	0	0	数据寄存器	数据寄存器	16
1	0	0	0	1	错误寄存器	特征寄存器	8
1	0	0	1	0	扇区数寄存器	扇区数寄存器	8
1	0	0	1	1	扇区号寄存器	扇区号寄存器	8
1	0	1	0	0	柱面号寄存器（低 8 位）	柱面号寄存器（低 8 位）	8
1	0	1	0	1	柱面号寄存器（高 8 位）	柱面号寄存器（高 8 位）	8
1	0	1	1	0	驱动器选择/磁头寄存器	驱动器选择/磁头寄存器	8
1	0	1	1	1	状态寄存器	命令寄存器	8
0	1	1	1	0	状态寄存器	设备控制器寄存器	8

IDE 硬盘的传输模式有以下 3 种。

- PIO（Programmed I/O）模式：PIO 模式是一种通过 CPU 执行 I/O 端口指令来进行数据读写的数据交换模式，是最早的硬盘数据传输模式，数据传输速率低下，CPU 占有率也很高。
- DMA（Driect Memory Access）模式：DMA 模式是一种不经过 CPU 而直接从内存存取数据的数据交换模式。PIO 模式下硬盘和内存之间的数据传输是由 CPU 来控制的；而在 DMA 模式下，CPU 只需向 DMA 控制器下达指令，让 DMA 控制器来处理数的传送，数据传送完毕再把信息反馈给 CPU，这样就在很大程度上减轻了 CPU 资源占有率。
- Ultra DMA（简称 UDMA）模式：它在包含了 DMA 模式的优点的基础上又增加了 CRC 校验技术，提高数据传输过程中的准确性，安全性得到保障。另外，在以往的硬盘数据传输模式下，一个时钟周期只传输一次数据，而在 UDMA 模式中逐渐应用了 Double Data Rate（双倍数据传输）技术，它在时钟的上升沿和下降沿各自进行一次数据传输，使数据传输速度成倍增长。

除了可以以 CHS（Cylinder、Head 和 Sector）的方式定位硬盘的扇区外，还可以用 LBA（逻辑块线性地址）的方式来定位，CHS 可以换算为 LBA。CHS 设计最多只允许 65536 个柱面、16 个磁头以及 255 个扇区/磁轨。这就将容量限制为 267386880 个扇区，即大约 137GB。

假设用 c 表示当前柱面号，h 表示当前磁头号，cs 表示起始柱面号，hs 表示起始磁头号，ss 表示起始扇区号，ps 表示每磁道有多少个扇区，ph 表示每柱面有多少个磁道（一般情况下，$cs = 0$、$hs = 0$、$ss = 1$、$ps = 63$、$ph = 255$），LBA 与 CHS 有如下对应关系：

```
lba=(c-cs)*ph*ps+(h-hs)*ps+(s-ss)
```

LBA 使得系统忽略硬盘的几何结构，交由驱动器来完成。系统不需要去查询 CHS 值，而只需要查询逻辑块地址（Logical Block Address，LBA），驱动器电子装置会找出要读或写的实际扇区。而 LBA48（48 位逻辑块地址）则可以使系统支持超过 137GB 的硬盘。

Linux 内核中，与 IDE 驱动相关的文件被放置在/drivers/ide 目录下，这个目录包含 ide.c、ide-cd.c、ide-cd.h、ide-disk.c、ide-dma.c、ide-floppy.c、ide-generic.c、ide-io.c、ide-iops.c、ide-lib.c、ide-pnp.c、ide-probe.c、ide-proc.c、ide-tape.c、ide-taskfile.c、ide-timing.h 文件以及针对 ARM、PPC、MIPS 等外围 IDE 设备驱动的目录。整个 IDE 设备驱动的体系结构很复杂，但驱动工程师要使 Linux 支持某嵌入式系统中的 IDE 硬盘，所需编写的代码量是非常少的。

华清远见

13.8.2 IDE 硬盘设备驱动的 block_device_operations 及成员函数

IDE 硬盘驱动的 block_device_operations 中包含了打开、释放、I/O 控制、获得几何信息、媒介改变和使介质有效的成员函数，这些函数的实现较简单，如代码清单 13.21 所示。

代码清单 13.21 IDE 硬盘驱动 block_device_operations 结构体及其成员函数

```
1 static struct block_device_operations idedisk_ops =
2 {
3  .owner      = THIS_MODULE,
4  .open       = idedisk_open,
5  .release    = idedisk_release,
6  .ioctl      = idedisk_ioctl,
7  .getgeo     = idedisk_getgeo, //得到几何信息
8  .media_changed  = idedisk_media_changed,  //媒介改变
9  .revalidate_disk= idedisk_revalidate_disk //使介质有效
10 };
11
12 static int idedisk_ioctl(struct inode *inode, struct file *file,
13         unsigned int cmd, unsigned long arg)
14 {
15  struct block_device *bdev = inode->i_bdev;
16  struct ide_disk_obj *idkp = ide_disk_g(bdev->bd_disk);
17  return generic_ide_ioctl(idkp->drive, file, bdev, cmd, arg);//通
用 IDE 的 I/O 控制
18 }
19
20
21  static  int  idedisk_getgeo(struct  block_device  *bdev,  struct
hd_geometry *geo)
22 {
23  struct ide_disk_obj *idkp = ide_disk_g(bdev->bd_disk);
24  ide_drive_t *drive = idkp->drive;
25  /* 得到几何信息，CHS */
26  geo->heads = drive->bios_head;
27  geo->sectors = drive->bios_sect;
28  geo->cylinders = (u16)drive->bios_cyl; /* truncate */
29  return 0;
30 }
31
32 static int idedisk_open(struct inode *inode, struct file *filp)
33 {
34  struct gendisk *disk = inode->i_bdev->bd_disk;
35  struct ide_disk_obj *idkp;
36  ide_drive_t *drive;
37
38  if (!(idkp = ide_disk_get(disk)))
39      return -ENXIO;
40
41  drive = idkp->drive;
42
43  drive->usage++;  //使用计数加 1
44  if (drive->removable && drive->usage == 1) {
45      ide_task_t args;
46      memset(&args, 0, sizeof(ide_task_t));
47      args.tfRegister[IDE_COMMAND_OFFSET] = WIN_DOORLOCK;
48      args.command_type = IDE_DRIVE_TASK_NO_DATA;
```

```
    49        args.handler      = &task_no_data_intr;
    50        check_disk_change(inode->i_bdev);
    51
    52        if  (drive->doorlocking  &&  ide_raw_taskfile(drive,  &args,
NULL))
    53            drive->doorlocking = 0;
    54   }
    55   return 0;
    56  }
    57
    58 static int idedisk_release(struct inode *inode, struct file *filp)
    59  {
    60   struct gendisk *disk = inode->i_bdev->bd_disk;
    61   struct ide_disk_obj *idkp = ide_disk_g(disk);
    62   ide_drive_t *drive = idkp->drive;
    63
    64   if (drive->usage == 1)
    65       ide_cacheflush_p(drive);
    66   if (drive->removable && drive->usage == 1) {
    67       ide_task_t args;
    68       memset(&args, 0, sizeof(ide_task_t));
    69       args.tfRegister[IDE_COMMAND_OFFSET] = WIN_DOORUNLOCK;
    70       args.command_type = IDE_DRIVE_TASK_NO_DATA;
    71       args.handler      = &task_no_data_intr;
    72       if  (drive->doorlocking  &&  ide_raw_taskfile(drive,  &args,
NULL))
    73           drive->doorlocking = 0;
    74   }
    75   drive->usage--;   //使用计数减 1
    76
    77   ide_disk_put(idkp);
    78   return 0;
    79  }
```

13.8.3 IDE 硬盘设备驱动的 I/O 请求处理

Linux 系统对 IDE 驱动进行了再封装，定义了 ide_driver_t 结构体，这个结构体容纳了 IDE 硬盘的探测、移除、请求处理和结束请求处理等函数指针。结束请求处理函数 ide_end_request()是对 end_request()函数针对 IDE 的修改。代码清单 13.21 所示为 ide_driver_t 结构体的定义。

代码清单 13.22　ide_driver_t 结构体

```
    1  static ide_driver_t idedisk_driver = {
    2   .gen_driver = {
    3       .owner       = THIS_MODULE,
    4       .name        = "ide-disk",
    5       .bus         = &ide_bus_type,
    6   },
    7   .probe           = ide_disk_probe,  //探测
    8   .remove          = ide_disk_remove,  //移除
    9   .shutdown        = ide_device_shutdown, //关闭
    10  .version         = IDEDISK_VERSION,
    11  .media           = ide_disk,  //媒介类型
```

```
    12  .supports_dsc_overlap   = 0,
    13  .do_request     = ide_do_rw_disk,  //请求处理函数
    14  .end_request        = ide_end_request,  //请求处理结束
    15  .error          = __ide_error,
    16  .abort          = __ide_abort,
    17  .proc           = idedisk_proc,
    18 };
```

代码清单 13.22 第 13 行的 ide_do_rw_disk()函数完成硬盘 I/O 操作请求的处理，如代码清单 13.23 所示。

代码清单 13.23　IDE 硬盘驱动 I/O 请求处理

```
    1   static ide_startstop_t ide_do_rw_disk(ide_drive_t *drive, struct
request *rq,
    2     sector_t block)
    3   {
    4     ide_hwif_t *hwif = HWIF(drive);
    5
    6     BUG_ON(drive->blocked);
    7
    8     if (!blk_fs_request(rq)) //不是文件系统请求
    9     {
    10      blk_dump_rq_flags(rq, "ide_do_rw_disk - bad command");
    11      ide_end_request(drive, 0, 0);  //以失败结束该请求
    12      return ide_stopped;
    13    }
    14
    15    pr_debug("%s: %sing: block=%llu, sectors=%lu, buffer=0x%08lx\n",
    16      drive->name,rq_data_dir(rq) == READ ? "read" : "writ", (unsigned
long
    17      long)block, rq->nr_sectors, (unsigned long)rq->buffer);
    18
    19    if (hwif->rw_disk)
    20      hwif->rw_disk(drive, rq);
    21
    22    return __ide_do_rw_disk(drive, rq, block);  //具体的请求处理
    23  }
    24
    25  static ide_startstop_t __ide_do_rw_disk(ide_drive_t *drive, struct
request *rq,
    26    sector_t block)
    27  {
    28    ide_hwif_t *hwif = HWIF(drive);
    29    unsigned int dma = drive->using_dma;
    30    u8 lba48 = (drive->addressing == 1) ? 1 : 0;
    31    task_ioreg_t command = WIN_NOP;
    32    ata_nsector_t nsectors;
    33
    34    nsectors.all = (u16)rq->nr_sectors; //要传送的扇区数
    35
    36    if (hwif->no_lba48_dma && lba48 && dma)
    37    {
    38      if (block + rq->nr_sectors > 1ULL << 28)
    39        dma = 0;
    40      else
    41        lba48 = 0;
    42    }
    43
```

```
44    if (!dma)
45    {
46      ide_init_sg_cmd(drive, rq);
47      ide_map_sg(drive, rq);
48    }
49
50    if (IDE_CONTROL_REG)
51      hwif->OUTB(drive->ctl, IDE_CONTROL_REG);
52
53    if (drive->select.b.lba)
54    {
55      if (lba48) //48 位 LBA
56      {
57        ...
58      }
59      else
60      {
61        //LBA 方式，写入要读写的位置信息到 IDE 寄存器
62        hwif->OUTB(0x00, IDE_FEATURE_REG);
63        hwif->OUTB(nsectors.b.low, IDE_NSECTOR_REG);
64        hwif->OUTB(block, IDE_SECTOR_REG);
65        hwif->OUTB(block >>= 8, IDE_LCYL_REG);
66        hwif->OUTB(block >>= 8, IDE_HCYL_REG);
67                          hwif->OUTB(((block    >>    8)    &0x0f)    |
drive->select.all,IDE_SELECT_REG);
68      }
69    }
70    else
71    {
72      unsigned int sect, head, cyl, track;
73      track = (int)block / drive->sect;
74      sect = (int)block % drive->sect + 1;
75      hwif->OUTB(sect, IDE_SECTOR_REG);
76      head = track % drive->head;
77      cyl = track / drive->head;
78
79      pr_debug("%s: CHS=%u/%u/%u\n", drive->name, cyl, head, sect);
80      //CHS 方式，写入要读写的位置信息到 IDE 寄存器
81      hwif->OUTB(0x00, IDE_FEATURE_REG);
82      hwif->OUTB(nsectors.b.low, IDE_NSECTOR_REG);
83      hwif->OUTB(cyl, IDE_LCYL_REG);
84      hwif->OUTB(cyl >> 8, IDE_HCYL_REG);
85      hwif->OUTB(head | drive->select.all, IDE_SELECT_REG);
86    }
87
88    if (dma)  //DMA 方式
89    {
90      if (!hwif->dma_setup(drive))   //设置 DMA 成功
91      {
92        if (rq_data_dir(rq))
93        {
94          command = lba48 ? WIN_WRITEDMA_EXT : WIN_WRITEDMA;
95          if (drive->vdma)
96            command = lba48 ? WIN_WRITE_EXT : WIN_WRITE;
97        }
```

```
98       else
99       {
100        command = lba48 ? WIN_READDMA_EXT : WIN_READDMA;
101        if (drive->vdma)
102          command = lba48 ? WIN_READ_EXT : WIN_READ;
103      }
104      hwif->dma_exec_cmd(drive, command);
105      hwif->dma_start(drive);
106      return ide_started;
107    }
108    /* 回到 PIO 模式 */
109    ide_init_sg_cmd(drive, rq);
110  }
111
112  if (rq_data_dir(rq) == READ) //数据传输方向是读
113  {
114    if (drive->mult_count)
115    {
116      hwif->data_phase = TASKFILE_MULTI_IN;
117      command = lba48 ? WIN_MULTREAD_EXT : WIN_MULTREAD;
118    }
119    else
120    {
121      hwif->data_phase = TASKFILE_IN;
122      command = lba48 ? WIN_READ_EXT : WIN_READ;
123    }
124    //执行读命令
125     ide_execute_command(drive, command, &task_in_intr, WAIT_CMD,
NULL);
126    return ide_started;
127  }
128  else  //数据传输方向是写
129  {
130    if (drive->mult_count)
131    {
132      hwif->data_phase = TASKFILE_MULTI_OUT;
133      command = lba48 ? WIN_MULTWRITE_EXT : WIN_MULTWRITE;
134    }
135    else
136    {
137      hwif->data_phase = TASKFILE_OUT;
138      command = lba48 ? WIN_WRITE_EXT : WIN_WRITE;
139    }
140
141    //写 IDE 命令寄存器/写入写命令
142    hwif->OUTB(command, IDE_COMMAND_REG);
143
144    return pre_task_out_intr(drive, rq);
145  }
146 }
```

从代码清单 13.23 可知，真正开始执行 I/O 操作的是其 22 行引用的__ide_do_rw_disk()函数。这个函数会根据不同的操作模式，将要读写的 LBA 或 CHS 信息写入 IDE 寄存器内，并给其命令寄存器写入读、写命令。

为了进行硬盘读写操作，第 61～67 行和第 80～85 行将参数写入地址寄存器和特性寄存器，如果是读，第 125 行调用的 ide_execute_command()会将读命令写入命令寄

存器；如果是写，第 142 行将写命令写入 IDE 命令寄存器 IDE_COMMAND_REG。

真正调用 ide_driver_t 结构体中 do_request()成员函数即 ide_do_rw_disk()的是 ide-io.c 文件中的 start_request()函数，这个函数会过滤掉一些请求，最终将读写 I/O 操作请求传递给 ide_do_rw_ disk()函数，如代码清单 13.24 所示。

代码清单 13.24　开始执行一个 IDE 请求的 start_request()函数

```
1  static ide_startstop_t start_request(ide_drive_t *drive, struct request *rq)
2  {
3    ide_startstop_t startstop;
4    sector_t block;
5
6    BUG_ON(!(rq->flags &REQ_STARTED));
7
8    /* 超过了最大失败次数 */
9    if (drive->max_failures && (drive->failures > drive->max_failures))
10   {
11     goto kill_rq;
12   }
13
14   block = rq->sector; //要传输的下一个扇区
15   if (blk_fs_request(rq) && (drive->media == ide_disk || drive->media ==
16     ide_floppy)) //是文件系统请求，是 IDE 盘
17   {
18     block += drive->sect0;
19   }
20   /* 如果将 0 扇区重映射到一扇区 */
21   if (block == 0 && drive->remap_0_to_1 == 1)
22     block = 1;
23
24   if (blk_pm_suspend_request(rq) && rq->pm->pm_step ==
25     ide_pm_state_start_suspend)
26     drive->blocked = 1;
27   else if (blk_pm_resume_request(rq) && rq->pm->pm_step ==
28     ide_pm_state_start_resume)
29   {
30     /* 醒来后的第一件事就是等待 BSY 位不忙 */
31     int rc;
32
33     rc = ide_wait_not_busy(HWIF(drive), 35000);
34     if (rc)
35       printk(KERN_WARNING "%s: bus not ready on wakeup\n", drive->name);
36     SELECT_DRIVE(drive);
37     HWIF(drive)->OUTB(8, HWIF(drive)->io_ports[IDE_CONTROL_OFFSET]);
38     rc = ide_wait_not_busy(HWIF(drive), 10000);
39     if (rc)
40       printk(KERN_WARNING "%s: drive not ready on wakeup\n", drive->name);
41   }
```

```
    42
    43   SELECT_DRIVE(drive);
    44   if (ide_wait_stat(&startstop, drive, drive->ready_stat, BUSY_STAT
| DRQ_STAT,
    45     WAIT_READY))  //等待驱动器 READY
    46   {
    47         printk(KERN_ERR  "%s:  drive  not  ready  for  command\n",
drive->name);
    48     return startstop;
    49   }
    50   if (!drive->special.all)
    51   {
    52     ide_driver_t *drv;
    53
    54     //其他非读写请求
    55     if (rq->flags &(REQ_DRIVE_CMD | REQ_DRIVE_TASK))
    56       return execute_drive_cmd(drive, rq);
    57     else if (rq->flags &REQ_DRIVE_TASKFILE)
    58       return execute_drive_cmd(drive, rq);
    59     else if (blk_pm_request(rq))
    60     {
    61       startstop = ide_start_power_step(drive, rq);
    62             if  (startstop  ==  ide_stopped  &&  rq->pm->pm_step  ==
ide_pm_state_completed)
    63         ide_complete_pm_request(drive, rq);
    64       return startstop;
    65     }
    66
    67     drv = *(ide_driver_t **)rq->rq_disk->private_data;
    68     return drv->do_request(drive, rq, block); //处理 I/O 操作请求
    69   }
    70   return do_special(drive);
    71   kill_rq: ide_kill_rq(drive, rq);
    72   return ide_stopped;
    73 }
```

IDE 硬盘驱动对 I/O 请求结束处理进行了针对 IDE 的整理，并填充在 ide_driver_t 结构体的 end_request 成员中，对应的函数为 ide_end_request()，如代码清单 13.25 所示。

代码清单 13.25　IDE I/O 请求结束处理

```
    1  /*   ide_end_request: 完成了一个 IDE 请求
    2   参数:  nr_sectors 被完成的扇区数量
    3  */
    4   int  ide_end_request  (ide_drive_t  *drive,  int  uptodate,  int
nr_sectors)
    5  {
    6   struct request *rq;
    7   unsigned long flags;
    8   int ret = 1;
    9
    10  spin_lock_irqsave(&ide_lock, flags);  //获得自旋锁
    11  rq = HWGROUP(drive)->rq;
    12
    13  if (!nr_sectors)
    14      nr_sectors = rq->hard_cur_sectors;
    15
    16  ret = __ide_end_request(drive, rq, uptodate, nr_sectors); //具体
```

```
的结束请求处理
    17
    18  spin_unlock_irqrestore(&ide_lock, flags);  //释放获得自旋锁
    19  return ret;
    20 }
    21
    22 static int __ide_end_request(ide_drive_t *drive, struct request *rq,
int
    23   uptodate, int nr_sectors)
    24 {
    25   int ret = 1;
    26
    27   BUG_ON(!(rq->flags &REQ_STARTED));
    28
    29   /*  如果对请求设置了 failfast，立即完成整个请求 */
    30   if (blk_noretry_request(rq) && end_io_error(uptodate))
    31     nr_sectors = rq->hard_nr_sectors;
    32
    33   if (!blk_fs_request(rq) && end_io_error(uptodate) && !rq->errors)
    34     rq->errors =  - EIO;
    35
    36   /* 决定是否再使能 DMA，如果 DMA 超过 3 次，采用 PIO 模式 */
    37   if (drive->state == DMA_PIO_RETRY && drive->retry_pio <= 3)
    38   {
    39     drive->state = 0;
    40     HWGROUP(drive)->hwif->ide_dma_on(drive);
    41   }
    42
    43   //结束请求
    44   if (!end_that_request_first(rq, uptodate, nr_sectors))
    45   {
    46     add_disk_randomness(rq->rq_disk);   //给系统的随机数池贡献熵
    47     blkdev_dequeue_request(rq);
    48     HWGROUP(drive)->rq = NULL;
    49     end_that_request_last(rq, uptodate);
    50     ret = 0;
    51   }
    52
    53   return ret;
    54 }
```

13.8.4 在内核中增加对新系统 IDE 设备的支持

尽管 IDE 设备的驱动非常复杂，但是由于其访问方式符合 ATA 标准，因而内核提供的 I/O 操作的代码是通用的，为了使内核能找到新系统中的 IDE 硬盘，工程师只需编写少量的针对特定硬件平台的底层代码。

使用 ide_register_hw()函数可注册 IDE 硬件接口，其原型为：

```
int ide_register_hw(hw_regs_t *hw, ide_hwif_t **hwifp);
```

这个函数接收的两个参数对应的数据结构为 hw_regs_s 和 ide_hwif_t， 其定义如代码清单 13.26 所示。

代码清单 13.26　hw_regs_s 和 ide_hwif_t 结构体的定义

```
1  typedef struct hw_regs_s
2  {
3    unsigned long io_ports[IDE_NR_PORTS]; /* task file 寄存器 */
4    int irq; /* 中断号 */
5    int dma; /* DMA 入口 */
6    ide_ack_intr_t *ack_intr; /* 确认中断 */
7    hwif_chipset_t chipset;
8    struct device *dev;
9  } hw_regs_t;
10
11 typedef struct hwif_s
12 {
13   ...
14
15   char name[6]; /* 接口名，如"ide0" */
16
17   hw_regs_t hw; /* 硬件信息 */
18   ide_drive_t drives[MAX_DRIVES]; /* 驱动器信息 */
19
20   u8 major; /* 主设备号 */
21   u8 index; /* 索引，如果为 0，则对应 ide0，如果为 1，则对应 ide1 */
22   ...
23   //DMA 操作
24   int(*dma_setup)(ide_drive_t*);
25   void(*dma_exec_cmd)(ide_drive_t *, u8);
26   void(*dma_start)(ide_drive_t*);
27   int(*ide_dma_end)(ide_drive_t *drive);
28   int(*ide_dma_check)(ide_drive_t *drive);
29   int(*ide_dma_on)(ide_drive_t *drive);
30   int(*ide_dma_off_quietly)(ide_drive_t *drive);
31   int(*ide_dma_test_irq)(ide_drive_t *drive);
32   int(*ide_dma_host_on)(ide_drive_t *drive);
33   int(*ide_dma_host_off)(ide_drive_t *drive);
34   int(*ide_dma_lostirq)(ide_drive_t *drive);
35   int(*ide_dma_timeout)(ide_drive_t *drive);
36   //寄存器访问
37   void(*OUTB)(u8 addr, unsigned long port);
38   void(*OUTBSYNC)(ide_drive_t *drive, u8 addr, unsigned long port);
39   void(*OUTW)(u16 addr, unsigned long port);
40   void(*OUTL)(u32 addr, unsigned long port);
41   void(*OUTSW)(unsigned long port, void *addr, u32 count);
42   void(*OUTSL)(unsigned long port, void *addr, u32 count);
43
44   u8(*INB)(unsigned long port);
45   u16(*INW)(unsigned long port);
46   u32(*INL)(unsigned long port);
47   void(*INSW)(unsigned long port, void *addr, u32 count);
48   void(*INSL)(unsigned long port, void *addr, u32 count);
49
50   ...
51 } ____cacheline_internodealigned_in_smp ide_hwif_t;
```

hw_regs_s 结构体描述了 IDE 接口的寄存器、用到的中断号和 DMA 入口，是对寄存器和硬件资源的描述，而 ide_hwif_t 是对 IDE 接口硬件访问方法的描述。

因此，为使得新系统支持 IDE 并被内核侦测到，工程师只需要初始化 hw_regs_s 和 ide_hwif_t 这两个结构体并使用 ide_register_hw()注册 IDE 接口即可。代码清单 13.27 所示为 H8/300 系列单片机 IDE 驱动的适配器注册代码。

代码清单 13.27　H8/300 系列单片机 IDE 接口注册

```
1   /* 寄存器操作函数 */
2   static void mm_outw(u16 d, unsigned long a)
3   {
4   __asm__("mov.b %w0,r2h\n\t"
5        "mov.b %x0,r2l\n\t"
6        "mov.w r2,@%1"
7        :
8        :"r"(d),"r"(a)
9        :"er2");
10  }
11
12  static u16 mm_inw(unsigned long a)
13  {
14  register u16 r __asm__("er0");
15  __asm__("mov.w @%1,r2\n\t"
16       "mov.b r2l,%x0\n\t"
17       "mov.b r2h,%w0"
18       :"=r"(r)
19       :"r"(a)
20       :"er2");
21  return r;
22  }
23
24  static void mm_outsw(unsigned long addr, void *buf, u32 len)
25  {
26  unsigned short *bp = (unsigned short *)buf;
27  for (; len > 0; len--, bp++)
28       *(volatile u16 *)addr = bswap(*bp);
29  }
30
31  static void mm_insw(unsigned long addr, void *buf, u32 len)
32  {
33  unsigned short *bp = (unsigned short *)buf;
34  for (; len > 0; len--, bp++)
35       *bp = bswap(*(volatile u16 *)addr);
36  }
37
38  #define H8300_IDE_GAP (2)
39
40  /* hw_regs_t 结构体初始化 */
41  static inline void hw_setup(hw_regs_t *hw)
```

```
42 {
43 int i;
44
45 memset(hw, 0, sizeof(hw_regs_t));
46 for (i = 0; i <= IDE_STATUS_OFFSET; i++)
47     hw->io_ports[i] = CONFIG_H8300_IDE_BASE + H8300_IDE_GAP*i;
48 hw->io_ports[IDE_CONTROL_OFFSET] = CONFIG_H8300_IDE_ALT;
49 hw->irq = EXT_IRQ0 + CONFIG_H8300_IDE_IRQ;
50 hw->dma = NO_DMA;
51 hw->chipset = ide_generic;
52 }
53
54 /* ide_hwif_t 结构体初始化 */
55 static inline void hwif_setup(ide_hwif_t *hwif)
56 {
57 default_hwif_iops(hwif);
58
59 hwif->mmio  = 2;
60 hwif->OUTW  = mm_outw;
61 hwif->OUTSW = mm_outsw;
62 hwif->INW   = mm_inw;
63 hwif->INSW  = mm_insw;
64 hwif->OUTL  = NULL;
65 hwif->INL   = NULL;
66 hwif->OUTSL = NULL;
67 hwif->INSL  = NULL;
68 }
69
70 /* 注册 IDE 适配器 */
71 void __init h8300_ide_init(void)
72 {
73 hw_regs_t hw;
74 ide_hwif_t *hwif;
75 int idx;
76
77   /* 申请内存区域 */
78 if   (!request_region(CONFIG_H8300_IDE_BASE,   H8300_IDE_GAP*8,
"ide-h8300"))
79     goto out_busy;
80 if    (!request_region(CONFIG_H8300_IDE_ALT,     H8300_IDE_GAP,
"ide-h8300")) {
81     release_region(CONFIG_H8300_IDE_BASE, H8300_IDE_GAP*8);
82     goto out_busy;
83 }
84
```

```
85  hw_setup(&hw); //初始化 hw_regs_t
86
87  /* 注册 IDE 接口 */
88  idx = ide_register_hw(&hw, &hwif);
89  if (idx == -1) {
90      printk(KERN_ERR "ide-h8300: IDE I/F register failed\n");
91      return;
92  }
93
94  hwif_setup(hwif);   //设置 ide_hwif_t
95  printk(KERN_INFO "ide%d: H8/300 generic IDE interface\n", idx);
96  return;
97
98  out_busy:
99  printk(KERN_ERR "ide-h8300: IDE I/F resource already used.\n");
100 }
```

第 1～36 行定义了寄存器读写函数，第 41 行的 hw_setup()函数用于初始化 hw_regs_t 结构体，第 55 行的 hwif_setup()函数用于初始化 ide_hwif_t 结构体，第 71 行的 h8300_ide_init()函数中使用 ide_register_hw()注册了这个接口。

13.9 总结

块设备的 I/O 操作方式与字符设备存在较大的不同，因而引入了 request_queue、request、bio 等一系列数据结构。在整个块设备的 I/O 操作中，贯穿于始终的就是“请求”，字符设备的 I/O 操作则是直接进行不绕弯，块设备的 I/O 操作会排队和整合。

驱动的任务是处理请求，对请求的排队和整合由 I/O 调度算法解决，因此，块设备驱动的核心就是请求处理函数或“制造请求”函数。

尽管在块设备驱动中仍然存在 block_device_operations 结构体及其成员函数，但其不再包含读写一类的成员函数，而只是包含打开、释放及 I/O 控制等与具体读写无关的函数。

块设备驱动的结构相当复杂的，但幸运的是，块设备不像字符设备那么包罗万象，它通常就是存储设备，而且驱动的主体已经由 Linux 内核提供，针对一个特定的硬件系统，驱动工程师所涉及的工作往往只是编写少量的与硬件直接交互的代码。

推荐课程： **嵌入式学院-嵌入式 Linux 长期就业班**

· **招生简章：** http://www.embedu.org/courses/index.htm

· 课程内容： http://www.embedu.org/courses/course1.htm

· 项目实战： http://www.embedu.org/courses/project.htm

· 出版教材： http://www.embedu.org/courses/course3.htm

· 实验设备： http://www.embedu.org/courses/course5.htm

推荐课程： 华清远见-嵌入式 Linux 短期高端培训班

· 嵌入式 Linux 应用开发班：

http://www.farsight.com.cn/courses/TS-LinuxApp.htm

· 嵌入式 Linux 系统开发班：

http://www.farsight.com.cn/courses/TS-LinuxEMB.htm

· 嵌入式 Linux 驱动开发班：

http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见