一 断连的定义

断连：表示操作系统无法识别ssd硬盘，无法读写

二 数据从文件到磁盘经历了那几个数据节点

文件--->文件系统----->操作系统内核---->驱动------>硬盘

三 故障点：

- 硬盘：ssd硬盘本身损坏，芯片组损坏，电路问题
- 驱动：ssd驱动与操作系统不匹配，驱动执行错误
- 操作系统内核：内核错误，bios参数设置，是否正确识别SSD
- 文件系统：文件系统元数据损坏，硬盘挂载错误
- 硬盘与机器的连接：过热，电源不稳定，连接松动

四 故障场景评估

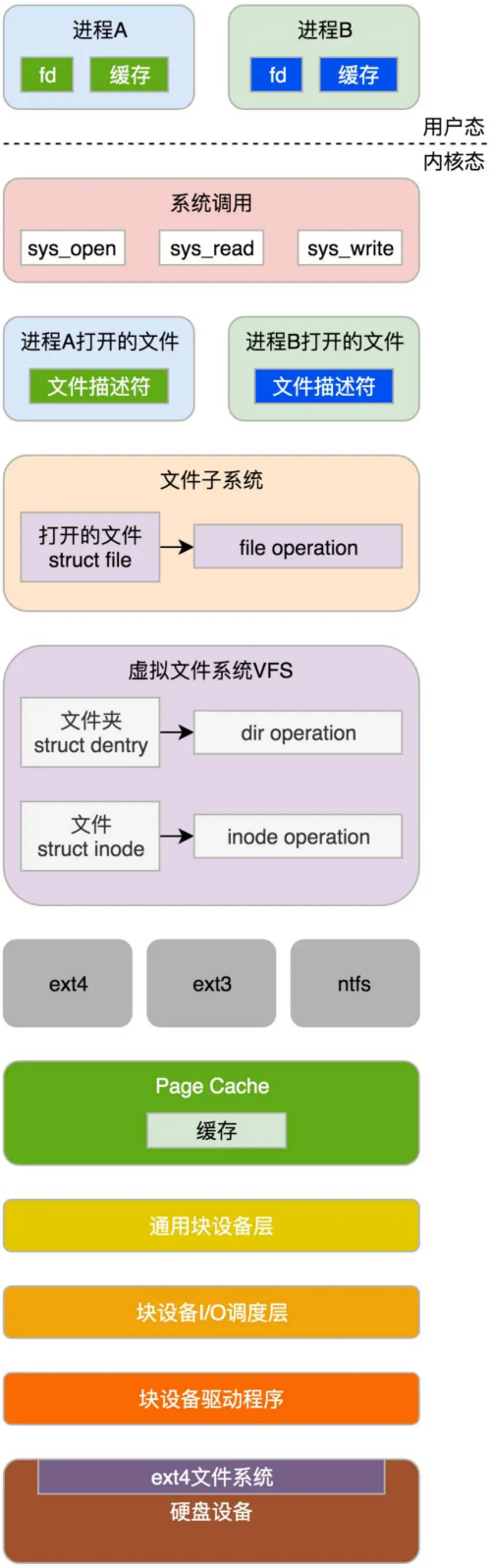
1. ssd硬盘老化，芯片组损坏，电路异常，此问题，一般发生在硬盘使用寿命快到期的时候。可检测系统日志，修复需要人工干预
2. ssd驱动与操作系统不匹配，一般发生在安装固定硬盘时，此场景可以规避，但是运行中途驱动程序运行错误也是存在的，需要厂商进行排查
3. 操作系统内核bios参数配置，是否正确识别SSD，可以提前预测并规避。
4. 文件系统元数据损坏，硬盘挂载数据被破坏，此场景发生的概率比较大，并且可以进行软件自动修复
5. 硬盘与机器的连接，过热，电源不稳定，连接松动，可能会发生在硬盘读写过程中，可检测系统日志，修复需要人工干预

五 实验思路

1. 先实现频率高，容易复现的故障场景，如下：
    a. 文件系统元数据
    b. 硬盘挂载数据
    c. 硬盘过热 -- hddtemp
    d. 硬盘连接松动
    e. 硬盘电源不稳定
2. 针对各个故障场景做根因分析以及观察系统日志特征，为故障检测提供检测依据
3. 针对以上故常场景对系统日志做定时扫描，对a,b场景进行自动修复，对c，d，e进行告警
4. 针对以下不好复现的故障或者可提前规避的场景，如下：
    a. ssd老化损坏，尽可能读写，复现（代价有点大），观察系统日志
    b. ssd驱动程序运行错误，不好复现，可以在出现时，保留系统日志
5. 以上都是发生故障后的故障检测，以及自动修复，与需要人工干预的告警

六 原理：从文件到磁盘，经历了哪些事

6.1 整体流程



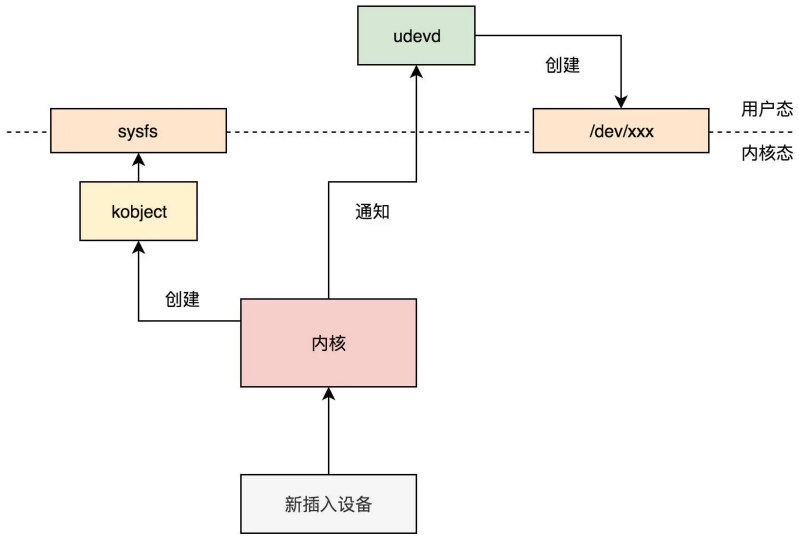
1. 用户使用高级语言函数读写文件
2. 函数进行系统调用，产生软中断，从用户态进入到内核态
3. 系统函数通过参数找到文件对应的文件描述符，经由文件系统，查找缓存中是否存在该文件，如果存在，则对缓存进行读写
4. 刷盘时，在通过通用设备层（为了向上层屏蔽不同块设备的驱动程序的差异性）调用块设备的驱动程序
5. 驱动程序通过ext4文件系统管理的磁盘，找到对应的磁盘分区，找到文件实际读写的物理位置，对块设备进行读写

6.2 块设备安装

6.2.1设备注册

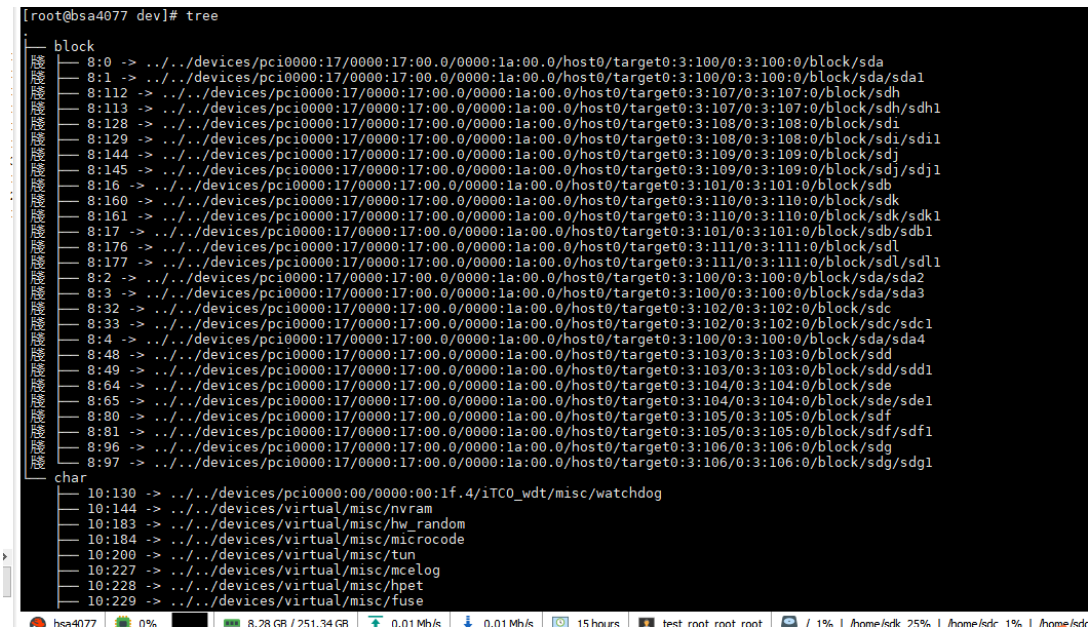
当操作系统插入一个设备：

对于用户态展示：内核会检测到这个设备，通知到守护进程udevd，udevd会创建对应的设备文件，并在/sys 路径下面的 sysfs 文件系统。把实际连接到系统上的设备和总线组成了一个分层的文件系统



- /sys/devices 是内核对系统中所有设备的分层次的表示；
- /sys/dev 目录下一个 char 文件夹，一个 block 文件夹，分别维护一个按字符设备和块设备的主次号码 (major:minor) 链接到真实的设备 (/sys/devices 下) 的符号链接文件；
- /sys/block 是系统中当前所有的块设备；
- /sys/module 有系统中所有模块的信息。

在对应的目录下面执行tree命令，可以看到对应的分层信息



对于内核：

核心思想：跟所有的程序代码一样，管理设备，其实就是创建一个设备的数据结构，以及驱动程序的数据结构，再将设备与对应的驱动程序进行绑定，就是对设备进行管理以及操作

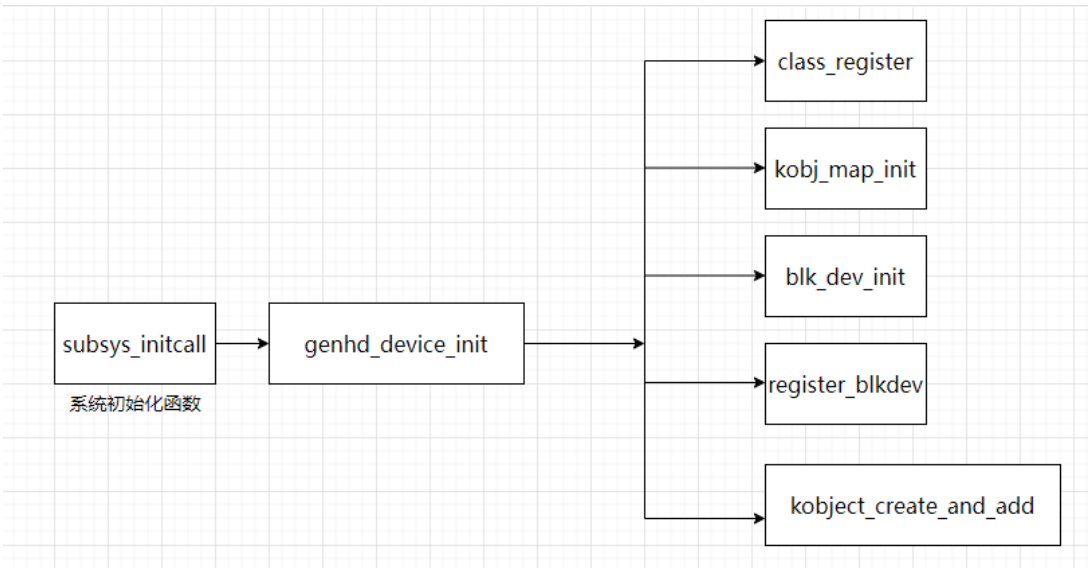
具体流程：

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

   block_class.dev_kobj = sysfs_dev_block_kobj;
   error = class_register(&block_class);
   if (unlikely(error))
      return error;
   bdev_map = kobj_map_init(base_probe, &block_class_lock);
   blk_dev_init();

   register_blkdev(BLOCK_EXT_MAJOR, "blkext");

   /* create top-level block dir */
   if (!sysfs_deprecated)
      block_depr = kobject_create_and_add("block", NULL);
   return 0;
}
```



1. class_register注册新的设备类，设备类用于将相关设备组织在一起，并提供一些共享的操作和接口。通过注册设备类，可以将设备加入到设备模型中，并由内核进行管理。

eg:存储设备：不管是**机械硬盘还是TF卡**，或者是一个**设备控制代码模块**，向操作系统内核表明的都是它是**存储设备**

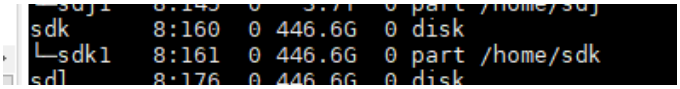
具体步骤：

a. 首先函数会检查struct class中的几个重要字段是否为空，如name表示设备类的名称，dev_uevent表示设备事件回调函数等。如果这些字段为空，函数将返回一个错误码。

b. 最后，class_register()函数会将设备类添加到全局的设备类列表中，以便后续可以通过class_find()函数根据设备类名称查找对应的设备类。

2. bdev_map = kobj_map_init(base_probe, &block_class_lock)块设备的管理初始化，bdev_map是一个长度为256的散列链表，主要使用此散列表完成每个主设备号下面维护着若干个链表，每个链表每个节点下注册若干个块次设备号

eg：这里的/dev/sdk是主设备，/dev/sdk1是主设备下的次设备



3. blk_dev_init初始化块设备
    1. 调用blk_init_queue函数初始化请求队列。请求队列用于管理块设备上的I/O请求。
    2. 调用register_blkdev函数注册字符设备主设备号。这是将块设备与字符设备绑定在一起的步骤，通过字符设备文件可以访问块设备。
    3. 调用blk_alloc_queue_node函数为块设备分配一个请求队列（struct request_queue）结构体，并使用GFP_KERNEL标志进行内存分配。
    4. 调用blk_register_queue函数将请求队列添加到系统的全局请求队列列表中。
    5. 调用blk_register_region函数将块设备注册到全局块设备列表中，使之能够被系统访问
4. 注册设备register_blkdev
    1. 检查传入的主设备号是否已经被使用，如果已经被使用，则返回一个负的错误码。
    2. 如果传入的主设备号为0，则调用blk_alloc_devt()分配一个未使用的主设备号。
    3. 调用register_chrdev_region()注册主设备号和块设备次设备号的范围。
    4. 调用cdev_init()初始化字符设备结构struct cdev。
    5. 将初始化好的字符设备结构struct cdev添加到内核字符设备列表中，调用cdev_add()。
5. kobject_create_and_add，创建一个内核对象（struct kobject）并将其加入到设备模型的层次结构中。通过这个内核对象，可以使用sysfs文件系统提供的接口与设备类进行交互。

Tips:kobject 是内核中用来表示一个对象的结构体，它包含了该对象的一些属性和操作方法，这些属性和方法可以被其他部分的代码使用和访问。kobject 对象可以用来表示各种不同的内核对象，比如设备、总线、驱动程序等。

6.2.2 驱动初始化

在硬盘在进行分区挂载，内核对那部分分区进行驱动初始化

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

    if (register_blkdev(simdisk_major, "simdisk") < 0) {
        pr_err("SIMDISK: register_blkdev: %d\n", simdisk_major);
        return -EIO;
    }
    pr_info("SIMDISK: major: %d\n", simdisk_major);
      // simdisk单元数
    if (n_files > simdisk_count)
        simdisk_count = n_files;
    if (simdisk_count > MAX_SIMDISK_COUNT)
        simdisk_count = MAX_SIMDISK_COUNT;

    sddev = kmalloc(simdisk_count * sizeof(struct simdisk),
            GFP_KERNEL);
```

```
    if (sddev == NULL)
        goto out_unregister;

    //第一个参数传入"simdisk"表示要创建一个名为"simdisk"的目录，
    //第二个参数传入0表示将其添加到proc文件系统的根目录。
    simdisk_procdir = proc_mkdir("simdisk", 0);
    if (simdisk_procdir == NULL)
        goto out_free_unregister;

    for (i = 0; i < simdisk_count; ++i) {
        if (simdisk_setup(sddev + i, i, simdisk_procdir) == 0) {
            if (filename[i] != NULL && filename[i][0] != 0 &&
                  (n_files == 0 || i < n_files))
                simdisk_attach(sddev + i, filename[i]);
        }
    }

    return 0;

out_free_unregister:
    kfree(sddev);
out_unregister:
    unregister_blkdev(simdisk_major, "simdisk");
    return -ENOMEM;
}

struct simdisk {
    const char *filename;
    spinlock_t lock;
    struct request_queue *queue; //用于存储请求队列的变量，使用blk_alloc_queue函数为其分配请求队列的内存。
    struct gendisk *gd;
    struct proc_dir_entry *procfile;
    int users;
    unsigned long size;
    int fd;
};

static int __init simdisk_setup(struct simdisk *dev, int which,
        struct proc_dir_entry *procdir)
{
    char tmp[2] = { '0' + which, 0 };

    dev->fd = -1;
    dev->filename = NULL;
    spin_lock_init(&dev->lock);
    dev->users = 0;

```

```
65    dev->queue = blk_alloc_queue(GFP_KERNEL);
66    if (dev->queue == NULL) {
67       pr_err("blk_alloc_queue failed\n");
68       goto out_alloc_queue;
69    }
70
71    blk_queue_make_request(dev->queue, simdisk_make_request);
72    dev->queue->queuedata = dev;
73
74    dev->gd = alloc_disk(SIMDISK_MINORS);
75    if (dev->gd == NULL) {
76       pr_err("alloc_disk failed\n");
77       goto out_alloc_disk;
78    }
79    dev->gd->major = simdisk_major;
80    dev->gd->first_minor = which;
81    dev->gd->fops = &simdisk_ops;
82    dev->gd->queue = dev->queue;
83    dev->gd->private_data = dev;
84    snprintf(dev->gd->disk_name, 32, "simdisk%d", which);
85    set_capacity(dev->gd, 0);
86    add_disk(dev->gd);
87
88    dev->procfile = proc_create_data(tmp, 0644, procdir, &fops, dev);
89    return 0;
90
91 out_alloc_disk:
92    blk_cleanup_queue(dev->queue);
93    dev->queue = NULL;
94 out_alloc_queue:
95    simc_close(dev->fd);
96    return -EIO;
97 }
98 struct gendisk {
99    /* major, first_minor and minors are input parameters only,
100     * don't use directly.  Use disk_devt() and disk_max_parts().
101     */
102    int major;       /* major number of driver */
103    int first_minor;
104    int minors;                     /* maximum number of minors, =1 for
105                                         * disks that can't be partitioned. */
106
107    char disk_name[DISK_NAME_LEN]; /* name of major driver */
108    char *(*devnode)(struct gendisk *gd, umode_t *mode);
109
110    unsigned int events;      /* supported events */
111    unsigned int async_events; /* async events, subset of all */
```

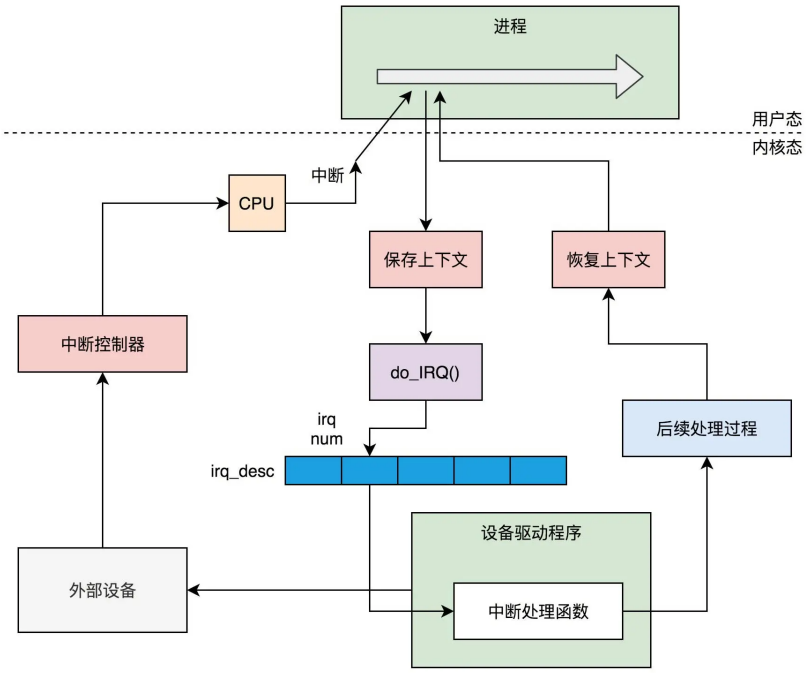
```
112
113     /* Array of pointers to partitions indexed by partno.
114      * Protected with matching bdev lock but stat and other
115      * non-critical accesses use RCU.  Always access through
116      * helpers.
117      */
118     struct disk_part_tbl __rcu *part_tbl;
119     struct hd_struct part0;
120
121     const struct block_device_operations *fops; // 块设备驱动程序提供的一系列回调函数（例如读、写、转换请求等）。
122     struct request_queue *queue; // 是请求队列，负责将IO请求排队，然后交给设备驱动进行处理。
123     void *private_data;
124
125     int flags;
126     struct device *driverfs_dev;  // FIXME: remove
127     struct kobject *slave_dir;
128
129     struct timer_rand_state *random;
130     atomic_t sync_io;     /* RAID */
131     struct disk_events *ev;
132 #ifdef  CONFIG_BLK_DEV_INTEGRITY
133     struct blk_integrity *integrity;
134 #endif
135     int node_id;
136     RH_KABI_EXTEND(struct badblocks *bb)
137 };
```

1. 调用register_blkdev函数注册虚拟块设备，设备号是240，名称是simdisk
2. 首先，在该函数中定义了一个名为simdisk的结构体指针，该结构体定义了一些用于虚拟块设备的参数和操作函数。
3. 然后，使用kmalloc函数为simdisk结构体分配内存。该结构体存储了虚拟块设备的信息，例如扇区大小、容量等。
4. 创建了一个名为"simdisk"的目录，作为proc文件系统的一个节点。proc文件系统是一个虚拟文件系统，它提供了内核和用户空间之间的接口，允许用户空间程序通过读写文件的方式来与内核进行通信
5. 循环遍历每个虚拟磁盘单元，初始化和注册虚拟磁盘驱动
    a. 初始化simdisk,并为simdisk存储请求队列queue分配内存
    b. 使用blk_queue_make_request函数设置请求队列的make_request_fn回调函数，这个回调函数负责处理块设备的读写请求，当用户程序发起对 simdisk 设备的读写请求时，dev->queue 会将请求传递给 simdisk_make_request 函数进行处理。simdisk_make_request 函数根据请求的类型和参数，调用底层驱动程序或设备模拟器来完成实际的读写操作
    c. 初始化gendisk，genddisk的主要两个字段是fops（驱动回调函数）和queue（IO请求队列）
    d. add_disk将虚拟磁盘也就是partiton磁盘分区的相关参数注册到内核中
    e. proc_create_data会返回一个新创建的proc_dir_entry结构体的指针。你可以使用该指针来引用和操作该proc文件。

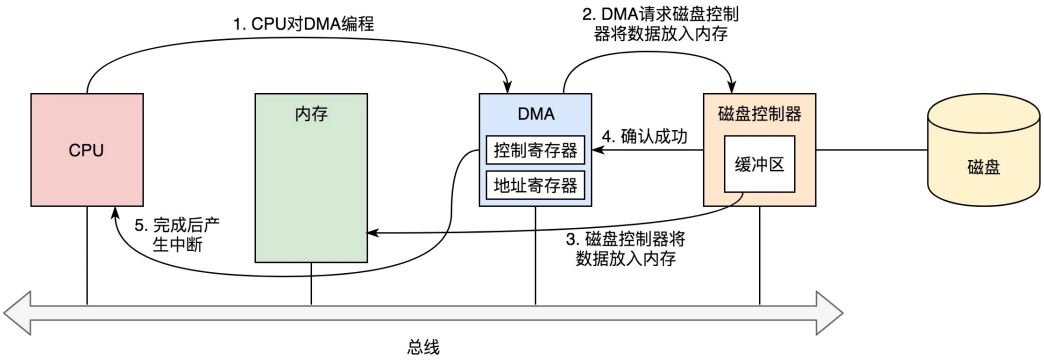
6.2.3 块设备读写

磁盘读写有两种形式：

1. 经历文件系统，对文件进行读写



2．直接对块设备进行读写，eg:DMA

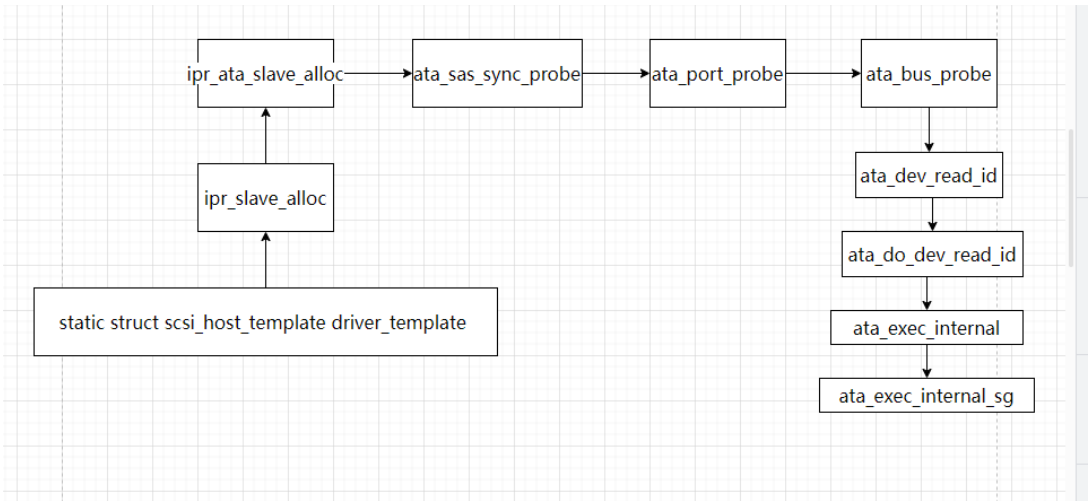


但是最终原理都是一样：设备以及驱动在内核中以及注册并且绑定好了之后，读写无非就是调用设备的驱动程序函数，把完成任务的细节用参数的形式传递给设备的驱动程序。

比如我们进程说的发送给磁盘的IO包：并不是数据本身，，而且将操作所需要的各种参数封装在一个数据结构中，成为IO包，这样就可以统一驱动程序功能函数的形式了。

6.3 qc timeount故障根因分析

整体调度流程：



重点说一下**ata_bus_probe**

ata_bus_probe()是在用户态程序中用于探测ATA总线的函数。该函数通常用于初始化和检测ATA总线上连接的设备，并为它们分配相应的驱动程序。

当调用ata_bus_probe()函数时，它会执行ATA总线的探测过程。这个过程通常包括以下几个步骤：

- 检测总线状态：函数会检测ATA总线的状态，包括检查总线是否连接正常、检测总线上的设备是否处于可用状态。
- 识别设备：函数会向总线上的设备发送IDENTIFY或者类似的命令，用于获取设备的信息，例如设备类型、容量等。
- 分配驱动程序：根据设备的类型和其他属性，函数会为每个设备分配合适的驱动程序，并将其与设备进行关联。

报错位置ata_exec_internal_sg函数解读：根据前面说得，内核要求设备完成任务，会调用设备的驱动程序函数，ata_exec_internal_sg就是这样一个函数，负责将一个或多个SG元素（即散步元素）传递给底层硬件，并等待命令完成。每一步的操作都已经在注释中标出

```
/**
 * ata_exec_internal_sg - execute libata internal command
 * @dev: Device to which the command is sent  执行命令的设备
 * @tf: Taskfile registers for the command and the result 执行的命令以及命令执行的结果的一个数据结构
 * @cdb: CDB for packet command 表示要执行的命令块描述符的指针。
 * @dma_dir: Data tranfer direction of the command 表示数据传输的方向，可以是 DMA_NONE（无数据传输）、DMA_TO_DEVICE（从主机到设备）或DMA_FROM_DEVICE（从设备到主机）。
 * @sgl: sg list for the data buffer of the command 一个指向散列列表的指针，用于指定数据传输的缓冲区。
 * @n_elem: Number of sg entries 散列列表中的元素数量。
 * @timeout: Timeout in msecs (0 for default) 操作的超时值，以毫秒为单位
 *
 * Executes libata internal command with timeout.  @tf contains
 * command on entry and result on return.  Timeout and error
 * conditions are reported via return value.  No recovery action
 * is taken after a command times out.  It's caller's duty to
 * clean up after timeout.
 * 执行带有超时的libata内部命令@tf包含输入时的命令和返回时的结果。超时和错误情况通过返回值报告。命令超时后不执行恢复操作。它的调用者会在超时后进行清理。
 * LOCKING:
 * None.  Should be called with kernel context, might sleep.
 *在没有内核上下文调用的情况下，会进行休眠
 * RETURNS:
 * Zero on success, AC_ERR_* mask on failure 0表示成功， AC_ERR_* mask表示失败
 */
unsigned ata_exec_internal_sg(struct ata_device *dev,
                              struct ata_taskfile *tf, const u8 *cdb,
                              int dma_dir, struct scatterlist *sgl,
                              unsigned int n_elem, unsigned long timeout)
{
   /* 1 函数会获取ATA设备的scsi地址 */
   struct ata_link *link = dev->link; #用于表示设备之间连接的数据结构。每个 ATA 接口可以连接多个 ATA 设备，而每个设备都会有一个对应的 ata_link 结构
   struct ata_port *ap = link->ap;# 指向设备所在的 ATA 端口的指针。一个 ATA 端口可以连接多个设备，但是一个 ata_link 只能属于一个端口。
```

```
31    u8 command = tf->command; # 要执行的命令
32    int auto_timeout = 0;
33    struct ata_queued_cmd *qc; # ATA 命令的队列命令。
34    unsigned int tag, preempted_tag;
35    u32 preempted_sactive, preempted_qc_active;
36    int preempted_nr_active_links;
37    /* 开始同步加锁  */
38    DECLARE_COMPLETION_ONSTACK(wait);
39    unsigned long flags;
40    unsigned int err_mask;
41    int rc;
42    // 获取自旋锁并且禁止中断
43    spin_lock_irqsave(ap->lock, flags);
44
45    /* no internal command while frozen */
46    if (ap->pflags & ATA_PFLAG_FROZEN) {
47       spin_unlock_irqrestore(ap->lock, flags); // 如果有命令在执行了，则释放自旋锁
48       return AC_ERR_SYSTEM;
49    }
50
51    /* initialize internal qc */
52    /* 2 初始化qc */
53    // internal qc 是指"internal queued command"，即 ATA/SATA 子系统中的内部队列命令。
54    /* XXX: Tag 0 is used for drivers with legacy EH as some
55     * drivers choke if any other tag is given.  This breaks
56     * ata_tag_internal() test for those drivers.  Don't use new
57     * EH stuff without converting to it.
58     */
59     // ap->ops 指向一个结构体，提供了操作这个 ata_port 的函数指针
60     // 如果存在错误处理函数，则将tag设置为ATA_TAG_INTERNAL,ATA_TAG_INTERNAL ,当 tag 被设置
为 ATA_TAG_INTERNAL 时，表示这是一个内部命令，一般用于处理错误或其他特殊情况。
61    if (ap->ops->error_handler)
62       tag = ATA_TAG_INTERNAL;
63    else
64       tag = 0;
65    //根据给定的控制器端口 ap 和命令标签 tag 获取相应的队列命令（queued command，简称QC）结构体 qc。
66    qc = __ata_qc_from_tag(ap, tag);
67    qc->tag = tag;
68    qc->scsicmd = NULL; #对应的scsi命令
69    qc->ap = ap;
70    qc->dev = dev;
71    ata_qc_reinit(qc); // 重新初始化 ATA 队列命令
72
73    preempted_tag = link->active_tag;
74    preempted_sactive = link->sactive;
75    preempted_qc_active = ap->qc_active;
76    preempted_nr_active_links = ap->nr_active_links;
```

```
77    link->active_tag = ATA_TAG_POISON;
78    link->sactive = 0;
79    ap->qc_active = 0;
80    ap->nr_active_links = 0;
81
82    /* prepare & issue qc */
83    /* 3 开始预备发布qc */
84    qc->tf = *tf;
85      // 如果cdb不为空，则把cdb复制到qc中
86    if (cdb)
87       memcpy(qc->cdb, cdb, ATAPI_CDB_LEN);
88
89    /* some SATA bridges need us to indicate data xfer direction */
90    // 指定数据传输方向DMA_NONE（无数据传输）、DMA_TO_DEVICE（从主机到设备）或DMA_FROM_DEVICE（从设备到主机）。
91    if (tf->protocol == ATAPI_PROT_DMA && (dev->flags & ATA_DFLAG_DMADIR) &&
92        dma_dir == DMA_FROM_DEVICE)
93       qc->tf.feature |= ATAPI_DMADIR;
94
95    qc->flags |= ATA_QCFLAG_RESULT_TF;
96    qc->dma_dir = dma_dir;
97    if (dma_dir != DMA_NONE) {
98       unsigned int i, buflen = 0;
99       struct scatterlist *sg;
100
101       for_each_sg(sgl, sg, n_elem, i)
102          buflen += sg->length;
103
104       ata_sg_init(qc, sgl, n_elem);
105       qc->nbytes = buflen;
106    }
107
108    qc->private_data = &wait;
109    qc->complete_fn = ata_qc_complete_internal;
110    // 发布qc，此处是真正将命令放入ATA命令队列，并提交给适配器以执行的地方
111    ata_qc_issue(qc);
112    // 释放自旋锁
113    spin_unlock_irqrestore(ap->lock, flags);
114    // 设置超时时间
115    if (!timeout) {
116       if (ata_probe_timeout) // ata_probe_timeout是一个用于控制ATA磁盘驱动探测的超时时间的参数。
117       //ata_probe_timeout只影响驱动程序在探测阶段的超时时间，与磁盘分区和挂载过程没有直接的关系。磁盘分区和挂载是在设备探测成功后进行的操作。
118          timeout = ata_probe_timeout * 1000;
119       else {
120          timeout = ata_internal_cmd_timeout(dev, command);
121          auto_timeout = 1;
122       }
```

```
123     }
124
125     if (ap->ops->error_handler)
126         ata_eh_release(ap);// 释放ATA端口的资源，初始化状态
127
128     //等待一个完成事件在超时时间内完成,如果返回0，则说明超时了，如果大于0则说明事件执行成功，这个函数通常用于等待异步操作的完成
129     rc = wait_for_completion_timeout(&wait, msecs_to_jiffies(timeout));
130
131     if (ap->ops->error_handler)
132      //获取并持有ATA端口（ap）的资源
133         ata_eh_acquire(ap);
134      // 刷新SATA设备上的PIO数据传输任务。
135     ata_sff_flush_pio_task(ap);
136
137     if (!rc) {
138         // 获取自旋锁并且禁止中断
139         spin_lock_irqsave(ap->lock, flags);
140
141         /* We're racing with irq here.  If we lose, the
142          * following test prevents us from completing the qc
143          * twice.  If we win, the port is frozen and will be
144          * cleaned up by ->post_internal_cmd().
145          */
146         // ATA_QCFLAG_ACTIVE是一个ATA任务队列标志，用于表示任务队列中的命令是激活状态，如果磁盘qc的状态还是激活状态，则说明命令执行超时了
147         if (qc->flags & ATA_QCFLAG_ACTIVE) {
148             qc->err_mask |= AC_ERR_TIMEOUT;
149
150             if (ap->ops->error_handler)
151                 // 如何错误处理函数存在，则停止与该设备端口的通信。这将导致所有正在进行的ATA命令都被暂停，并且该设备端口将不会接收到新的命令，错误处理函数会自动处理
152                 ata_port_freeze(ap);
153             else
154                 // 否则直接标记传输命令已完成
155                 ata_qc_complete(qc);
156
157             if (ata_msg_warn(ap))
158                 ata_dev_warn(dev, "qc timeout (cmd 0x%x)\n",
159                              command);
160         }
161
162         spin_unlock_irqrestore(ap->lock, flags);
163     }
164
165     /* do post_internal_cmd */
166     if (ap->ops->post_internal_cmd)
167         ap->ops->post_internal_cmd(qc);
```

```
168
169     /* perform minimal error analysis */
170     if (qc->flags & ATA_QCFLAG_FAILED) {
171         if (qc->result_tf.command & (ATA_ERR | ATA_DF))
172             qc->err_mask |= AC_ERR_DEV;
173
174         if (!qc->err_mask)
175             qc->err_mask |= AC_ERR_OTHER;
176
177         if (qc->err_mask & ~AC_ERR_OTHER)
178             qc->err_mask &= ~AC_ERR_OTHER;
179     }
180
181     /* finish up */
182     spin_lock_irqsave(ap->lock, flags);
183
184     *tf = qc->result_tf;
185     err_mask = qc->err_mask;
186
187     ata_qc_free(qc);
188     link->active_tag = preempted_tag;
189     link->sactive = preempted_sactive;
190     ap->qc_active = preempted_qc_active;
191     ap->nr_active_links = preempted_nr_active_links;
192
193     spin_unlock_irqrestore(ap->lock, flags);
194
195     if ((err_mask & AC_ERR_TIMEOUT) && auto_timeout)
196         ata_internal_cmd_timed_out(dev, command);
197
198     return err_mask;
199 }
```

```
1  /* 在libata.h中定义
2  struct ata_device {
3      struct ata_link       *link; # 用于表示设备之间连接的数据结构。每个 ATA 接口可以连接多个 ATA 设备，而每个设备都会有一个对应的 ata_link 结构
4      unsigned int      devno;    /* 0 or 1 表示设备的编号，从 0 开始。对于 ATA 接口来说，0 表示主设备，1 表示从设备 */
5      unsigned int      horkage;   /* List of broken features */
6      unsigned long     flags;    /* ATA_DFLAG_xxx */
7      struct scsi_device *sdev;    /* attached SCSI device */
8      void          *private_data;
9  #ifdef CONFIG_ATA_ACPI
10     union acpi_object  *gtf_cache;
```

```
    unsigned int      gtf_filter;
#endif
#ifdef CONFIG_SATA_ZPODD
    void          *zpodd;
#endif
    struct device     tdev;
    /* n_sector is CLEAR_BEGIN, read comment above CLEAR_BEGIN */
    u64           n_sectors; /* size of device, if ATA */
    u64           n_native_sectors; /* native size, if ATA */
    unsigned int      class;    /* ATA_DEV_xxx */
    unsigned long     unpark_deadline;

    u8        pio_mode;
    u8        dma_mode;
    u8        xfer_mode;
    unsigned int      xfer_shift;    /* ATA_SHIFT_xxx */

    unsigned int      multi_count;   /* sectors count for
                      READ/WRITE MULTIPLE */
    unsigned int      max_sectors;   /* per-device max sectors */
    unsigned int      cdb_len;

    /* per-dev xfer mask */
    unsigned long     pio_mask;
    unsigned long     mwdma_mask;
    unsigned long     udma_mask;

    /* for CHS addressing */
    u16           cylinders; /* Number of cylinders */
    u16           heads;    /* Number of heads */
    u16           sectors;   /* Number of sectors per track */

    union {
        u16       id[ATA_ID_WORDS]; /* IDENTIFY xxx DEVICE data */
        u32       gscr[SATA_PMP_GSCR_DWORDS]; /* PMP GSCR block */
    } ____cacheline_aligned;

    /* DEVSLP Timing Variables from Identify Device Data Log */
    u8        devslp_timing[ATA_LOG_DEVSLP_SIZE];

    /* NCQ send and receive log subcommand support */
    u8        ncq_send_recv_cmds[ATA_LOG_NCQ_SEND_RECV_SIZE];

    /* error history */
    int           spdn_cnt;
    /* ering is CLEAR_END, read comment above CLEAR_END */
    struct ata_ering    ering;
```

```
};


struct ata_link {
   struct ata_port       *ap; # 指向设备所在的 ATA 端口的指针。一个 ATA 端口可以连接多个设备，但是一个 ata_link 只能属于一个端口。
   int          pmp;      /* port multiplier port # */

   struct device     tdev;
   unsigned int      active_tag;    /* active tag on this link */
   u32          sactive;   /* active NCQ commands */

   unsigned int      flags;    /* ATA_LFLAG_xxx */

   u32          saved_scontrol;    /* SControl on probe */
   unsigned int      hw_sata_spd_limit;
   unsigned int      sata_spd_limit;
   unsigned int      sata_spd;  /* current SATA PHY speed */
   enum ata_lpm_policy    lpm_policy;

   /* record runtime error info, protected by host_set lock */
   struct ata_eh_info eh_info;
   /* EH context */
   struct ata_eh_context  eh_context;

   struct ata_device  device[ATA_MAX_DEVICES];

   unsigned long     last_lpm_change; /* when last LPM change happened */
};

struct ata_port {
   struct Scsi_Host   *scsi_host; /* our co-allocated scsi host  指向此 ata_port 所属的 ATA/SATA 控制器。 */
   struct ata_port_operations *ops; // 指向一个结构体，提供了操作这个 ata_port 的函数指针
   spinlock_t    *lock;
   /* Flags owned by the EH context. Only EH should touch these once the
      port is active */
   unsigned long     flags; /* ATA_FLAG_xxx  用于标识和控制此 ata_port 的各种状态和功能。*/
   /* Flags that change dynamically, protected by ap->lock */
   unsigned int      pflags; /* ATA_PFLAG_xxx 用于标识和控制 ATA/SATA 设备的不同状态和功能。 */
   unsigned int      print_id; /* user visible unique port ID */
   unsigned int            local_port_no; /* host local port num */
   unsigned int      port_no; /* 0 based port no. inside the host */

#ifdef CONFIG_ATA_SFF
   struct ata_ioports ioaddr;    /* ATA cmd/ctl/dma register blocks */
   u8       ctl;   /* cache of ATA control register */
```

```
    u8      last_ctl;  /* Cache last written value */
    struct ata_link*   sff_pio_task_link; /* link currently used */
    struct delayed_work    sff_pio_task;
#ifdef CONFIG_ATA_BMDMA
    struct ata_bmdma_prd   *bmdma_prd;    /* BMDMA SG list */
    dma_addr_t    bmdma_prd_dma; /* and its DMA mapping */
#endif /* CONFIG_ATA_BMDMA */
#endif /* CONFIG_ATA_SFF */

    unsigned int      pio_mask;
    unsigned int      mwdma_mask;
    unsigned int      udma_mask;
    unsigned int      cbl;   /* cable type; ATA_CBL_xxx */

    struct ata_queued_cmd  qcmd[ATA_MAX_QUEUE];
    unsigned long     sas_tag_allocated; /* for sas tag allocation only */
    unsigned int      qc_active;
    int          nr_active_links; /* #links with active qcs */
    unsigned int      sas_last_tag;  /* track next tag hw expects */

    struct ata_link       link;     /* host default link */
    struct ata_link       *slave_link;   /* see ata_slave_link_init() */

    int          nr_pmp_links;  /* nr of available PMP links */
    struct ata_link       *pmp_link; /* array of PMP links */
    struct ata_link       *excl_link;    /* for PMP qc exclusion */

    struct ata_port_stats  stats;
    struct ata_host       *host;
    struct device     *dev;
    struct device     tdev;

    struct mutex      scsi_scan_mutex;
    struct delayed_work    hotplug_task;
    struct work_struct scsi_rescan_task;

    unsigned int      hsm_task_state;

    u32          msg_enable;
    struct list_head   eh_done_q;
    wait_queue_head_t  eh_wait_q;
    int          eh_tries;
    struct completion  park_req_pending;

    pm_message_t      pm_mesg;
    enum ata_lpm_policy    target_lpm_policy;

```

```
    struct timer_list  fastdrain_timer;
    unsigned long      fastdrain_cnt;

    int          em_message_type;
    void         *private_data;

#ifdef CONFIG_ATA_ACPI
    struct ata_acpi_gtm    __acpi_init_gtm; /* use ata_acpi_init_gtm() */
#endif
    /* owned by EH */
    u8       sector_buf[ATA_SECT_SIZE] ____cacheline_aligned;
};

/**
 *ata_queued_cmd 是 Linux 内核中 ATA/SATA 子系统的一个数据结构，用于表示一个 ATA 命令的队列命令。
 *在 ATA/SATA 接口中，所有的命令都是通过 ATA 命令寄存器和数据寄存器来进行传输的。
 * 而ata_queued_cmd 结构体是用于描述一个 ATA 命令的控制信息的，包括命令的类型、设备号、扇区号、数据缓冲区等。
 */
struct ata_queued_cmd {
    struct ata_port       *ap; # 指向设备所在的 ATA 端口的指针
    struct ata_device  *dev;  # 设备

    struct scsi_cmnd   *scsicmd; # 对应的scsi命令
    void         (*scsidone)(struct scsi_cmnd *);

    struct ata_taskfile    tf;
    u8       cdb[ATAPI_CDB_LEN];

    unsigned long      flags;    /* ATA_QCFLAG_xxx */
    unsigned int       tag; // 命令标识符
    unsigned int       n_elem;
    unsigned int       orig_n_elem;

    int          dma_dir;

    unsigned int       sect_size;

    unsigned int       nbytes;
    unsigned int       extrabytes;
    unsigned int       curbytes;

    struct scatterlist sgent;

    struct scatterlist *sg;

    struct scatterlist *cursg;
    unsigned int       cursg_ofs;
```

```
197
198     unsigned int      err_mask;
199     struct ata_taskfile    result_tf;
200     ata_qc_cb_t       complete_fn;
201
202     void          *private_data;
203     void          *lldd_task;
204 };
```

七 常见的检测工具

常见的SSD检测工具总结

工具名称	作用	平台兼容性	模块
smartctl	用于监控和分析硬盘驱动器的健康状况	LINUX	驱动
hdparm	用于设置和诊ATA/SATA硬盘的实用工具。可以使用hdparm命令来获取硬盘的参数和性能信息。	LINUX	硬盘，驱动
lshw	用于列出计算机硬件信息，包括磁盘驱动器。它可以提供有关硬件设备及其配置的详细信息。	LINUX	硬件，驱动
fsck	用于检查和修复文件系统的工具	LINUX	文件系统
FIO	灵活的、可定制的I/O性能测试工具。可以模拟各种负载类型来评估存储系统的性能，包括顺序读写、随机读写、混合工作负载等。	LINUX	性能测试
CrystalDiskMark	简单而直观的工具，可检测和显示硬盘的详细健康信息	WINDOWS	windows，硬盘

smartctl使用详解

1 安装

```
1 yum install -y smartmontools
```

2 查看磁盘是否支持smart

```
1 smartctl -i /dev/sda  #/dev/sda为对应的磁盘设备，如何是raid，则要指定物理盘编号 -d megaraid,0
```

```
[root@bsa121 sdb]# smartctl -i /dev/sdb -d megaraid,0
smartctl 6.5 2016-05-07 r4318 [x86_64-linux-3.10.0-957.el7.x86_64] (local build)
Copyright (C) 2002-16, Bruce Allen, Christian Franke, www.smartmontools.org

=== START OF INFORMATION SECTION ===
Device Model:     INTEL SSDSC2KB480GZ
Serial Number:    PHYI238500EF480BGN
LU WWN Device Id: 5 5cd2e4 156120b28
Firmware Version: 7CV10111
User Capacity:    480,103,981,056 bytes [480 GB]
Sector Sizes:     512 bytes logical, 4096 bytes physical
Rotation Rate:    Solid State Device
Form Factor:      2.5 inches
Device is:        Not in smartctl database [for details use: -P showall]
ATA Version is:   ACS-3 T13/2161-D revision 5
SATA Version is:  SATA 3.2, 6.0 Gb/s (current: 6.0 Gb/s)
Local Time is:    Wed Jul 26 09:[illegible]:11 2023 CST
SMART support is: Available - device has SMART capability.
SMART support is: Enabled
```

enabled表示支持

3 启用smart

```
smartctl --smart=on --offlineauto=on --saveauto=on /dev/sda  #/dev/sda为对应的磁盘设备，如何是raid，则要指定物理盘编号 -d megaraid,0
```

4 查看硬盘的所有SMART信息

```
smartctl -a /dev/sda
```

5 查看硬盘的健康状况

```
smartctl -H /dev/sda
```

```
[root@bsa121 sdb]# smartctl -H /dev/sdb
smartctl 6.5 2016-05-07 r4318 [x86_64-linux-3.10.0-957.el7.x86_64] (local build)
Copyright (C) 2002-16, Bruce Allen, Christian Franke, www.smartmontools.org

=== START OF READ SMART DATA SECTION ===
SMART Health Status: OK

[root@bsa121 sdb]#
```

6 查看硬盘的历史错误信息

```
smartctl -l error /dev/sda #/dev/sda为对应的磁盘设备，如何是raid，则要指定物理盘编号 -d megaraid,0
```

```
[root@bsa121 sdb]# smartctl -l error /dev/sdb -d megaraid,0
smartctl 6.5 2016-05-07 r4318 [x86_64-linux-3.10.0-957.el7.x86_64] (local build)
Copyright (C) 2002-16, Bruce Allen, Christian Franke, www.smartmontools.org

=== START OF READ SMART DATA SECTION ===
SMART Error Log Version: 1
No Errors Logged

[root@bsa121 sdb]#
```

7 后台执行smart测试

```
smartctl --test=long /dev/sda #/dev/sda为对应的磁盘设备，如何是raid，则要指定物理盘编号 -d megaraid,0
```

```
[root@bsa121 sdb]# smartctl --test=long /dev/sdb -d megaraid,0
smartctl 6.5 2016-05-07 r4318 [x86_64-linux-3.10.0-957.el7.x86_64] (local build)
Copyright (C) 2002-16, Bruce Allen, Christian Franke, www.smartmontools.org

=== START OF OFFLINE IMMEDIATE AND SELF-TEST SECTION ===
Sending command: "Execute SMART Extended self-test routine immediately in off-line mode".
Drive command "Execute SMART Extended self-test routine immediately in off-line mode" successful.
Testing has begun.
Please wait 2 minutes for test to complete.
Test will complete after Wed Jul 26 09:37:40 2023

Use smartctl -X to abort test.
[root@bsa121 sdb]#
```

8 中断smart自测

```
smartctl -X /dev/sda #/dev/sda为对应的磁盘设备，如何是raid，则要指定物理盘编号 -d megaraid,0
```

```
[root@bsa121 sdb]# smartctl -X /dev/sdb -d megaraid,0
smartctl 6.5 2016-05-07 r4318 [x86_64-linux-3.10.0-957.el7.x86_64] (local build)
Copyright (C) 2002-16, Bruce Allen, Christian Franke, www.smartmontools.org

=== START OF OFFLINE IMMEDIATE AND SELF-TEST SECTION ===
Sending command: "Abort SMART off-line mode self-test routine".
Self-testing aborted!
[root@bsa121 sdb]#
```

9 显示smart自测日志

```
smartctl -l selftest /dev/sda -d megaraid,0
```

```
[root@bsa121 sdb]# smartctl -l selftest /dev/sda -d megaraid,0
smartctl 6.5 2016-05-07 r4318 [x86_64-linux-3.10.0-957.el7.x86_64] (local build)
Copyright (C) 2002-16, Bruce Allen, Christian Franke, www.smartmontools.org

=== START OF READ SMART DATA SECTION ===
SMART Self-test log structure revision number 1
Num  Test_Description    Status                  Remaining  LifeTime(hours)  LBA_of_first_error
# 1  Extended offline    Completed without error       00%        62         -

[root@bsa121 sdb]#
```

hdparm命令使用详解

1 安装

```
yum install hdparm
```

2 具体使用参数可以查看文档

```
man hdparm
```

3 检测硬盘温度是否正常

```
hdparm -H /dev/sdb
```

```
[root@bsa121 sdb]# hdparm -H /dev/sdb

/dev/sdb:
SG_IO: bad/missing sense data, sb[]:  70 00 05 00 00 00 00 0d 00 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 drive temperature (celsius) is:  under -20
 drive temperature in range:  yes
[root@bsa121 sdb]#
```

4 显示硬盘的相关设置

```
hdparm /dev/sda
```

```
[root@bsa121 sdb]# hdparm /dev/sdb

/dev/sdb:
SG_IO: bad/missing sense data, sb[]:  70 00 05 00 00 00 00 0d 00 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 multcount     =  0 (off)
 readonly      =  0 (off)
 readahead     = 256 (on)
 geometry      = 58303/255/63, sectors = 936640512, start = 0
[root@bsa121 sdb]#
```

fsck使用详解

使用方法：

```
fsck  [-panyrcdfktvDFV] [-b 超级块] [-B 块大小] [-l|-L 坏块文件] [-C fd] [-j 外部日志] [-E 扩展选项] [-z 撤销文件] 设备
```

参数	备注
-p	自动修复（不询问）
-n	不对文件系统做任何更改
-y	对所有询问都回答“是”
-c	检查可能的坏块，并将它们加入坏块列表
-f	强制进行检查，即使文件系统被标记为“没有问题”
-b	superblock 使用备选超级块
-B	blocksize 使用指定块大小来查找超级块
-j	external_journal 指定外部日志的位置
-l	bad_blocks_file 添加到指定的坏块列表（文件）
-L	bad_blocks_file 指定坏块列表（文件）
-z	undo_file 创建一个撤销文件