Linux 内核学习笔记-磁盘篇

本文将分三部分来记录 Linux 内核磁盘相关的知识，分别是虚拟文件系统 VFS、块设备层以及文件系统。

三者的简要关系如下，如图所示，文件系统位于磁盘上，对磁盘上的文件进行组织和管理，块设备层可以理解为块设备的抽象，而虚拟文件系统 VFS 是对文件系统的一层抽象，下面先从底层的文件系统说起。

1 文件系统

Linux 支持的文件系统有几十种，但是 ext 文件系统使用的最为广泛，目前 ext 文件系统族有 ext2、ext3 和 ext4，而 ext2 又是 ext 文件系统的基础，所以本文将以 ext2 为例来讲解 ext 文件系统族。

ext2 文件系统是基于块设备的文件系统，它将硬盘划分为若干个块，每个块的长度都相同，一个文件占用的存储空间是块长度的整数倍，即一个块不能用来存储两个文件。

ext2 由大量的块组组成，块组的结构如下图：

而整个硬盘的结构可以用下图表示：

启动块是系统在启动时，由 BIOS 自动加载并执行，它包含一个启动装载程序，通常位于硬盘的起始处，文件系统是从启动块之后开始的。下面来了解块组中各个部分的构成。

1.1 超级块

超级块存储的信息包括空闲和已使用的块的数目、块长度、当前文件系统的状态、各种时间戳，标识文件系统类型的魔数，每个块组中存储的超级块内容都是相同的，这样做是为了在系统崩溃损坏超级块的情况下，有其他副本可以用来恢复数据。

```
1.  struct ext2_super_block{
2.          __le32 s_inodes_count;    /*inode 数据*/
3.          __le32 s_blocks_count;    /*块数目*/
4.          __le32 s_r_blocks_count; /* 已分配块的数目*/
```

```
5.        __le32 s_free_blocks_count; /*空闲块数目*/
6.        __le32 s_free_inodes_count; /*空闲 inode 数目*/
7.        __le32 s_first_data_block;  /*第一个数据块*/
8.        __le32 s_log_block_size;    /*块长度*/
9.        __le32 s_log_frag_size;     /*碎片长度*/
10.       __le32 s_blocks_per_group; /*每个块组包含的块数*/
11.       __le32 s_frags_per_group;   /*每个块组包含的碎片*/
12.       __le32 s_inodes_per_group;  /*每个块组的 inode 数目*/
13.       __le32 s_mtime;                    /*装载时间*/
14.       __le32 s_wtime;                    /*写入时间*/
15.       __le16 s_mnt_count;                /*装载计数*/
16.       __le16 s_max_mnt_count;           /*最大装载计数*/
17.       __le16 s_magic;                    /*魔数，标记文件系统类型
    */
18.       ..........
19. }
```

超级块的存储结构主要包括以上部分，下面解释关键字段。

- s_log_block_size：用来表示块的长度，取值为 0、1 和 2，分别对应的块长度为 1024、2048 和 4096，块长度是由 mke2fs 创建文件系统期间指定，创建后，就不能修改
- s_block_per_group 和 s_inodes_per_group：每个块组中块和 inode 的数量，创建文件系统时确定。
- s_magic：存储的是 0XEF53，用来标识 ext2 文件系统

1.2 组描述符

组描述符反映了文件系统中各个块组的状态，例如块组中的空闲块和 inode 数目，每个块组都包含了文件系统中所有块组的组描述信息，其数据结构如下：

```
1.  struct ext2_group_desc{
2.        __le32 bg_block_bitmap;              /*块位图块*/
3.        __le32 bg_inode_bitmap;              /*inode 位图块*/
4.        __le32 bg_inode_table;               /*inode 表块*/
5.        __le16 bg_free_blocks_count;       /*空闲块数目*/
6.        __le16 bg_free_inodes_count;       /*空闲 inode 数目*/
7.        __le16 bg_used_dirs_count;          /*目录数目*/
8.        __le16 bg_pad;
9.        __le32 bg_reserved[3];
10. }
```

1.3 数据块位图和 inode 位图

位图是保存长的比特位串，该结构中每个比特位都对应于一个数据块或 inode，用来标识对应的数据块或 inode 是空闲还是被使用。总是占用一个数据块。

1.4 inode 表

inode 表包含了块组中所有的 inode，inode 包含了文件的属性和对应的数据块的标号，inode 的数据结构如下：

```
1.  struct ext2_inode{
2.        __le16 i_mode;       /*文件模式*/
3.        __le16 i_uid;        /*所有者 UID 的低 16 位*/
4.        __le32 i_size;       /*长度，按字节计算*/
5.        __le32 i_atime;      /*访问时间*/
6.        __le32 i_ctime;      /*创建时间*/
7.        __le32 i_mtime;      /*修改时间*/
8.        __le32 i_dtime;      /*删除时间*/
9.        __le16 i_gid;        /*组 ID 的低 16 位*/
10.       __le16 i_links_count; /*链接计数*/
11.       __le32 i_blocks;       /*块数目*/
12.       __le32 i_flags;        /*文件标志*/
13.       uion{
14.             ……
15.       }masix1;
16.       __le32 i_blocks[EXT2_N_BLOCKS]; /*块指针*/
17.       __le32 i_generation;       /*文件版本，用于 NFS*/
18.       ……
19. }
```

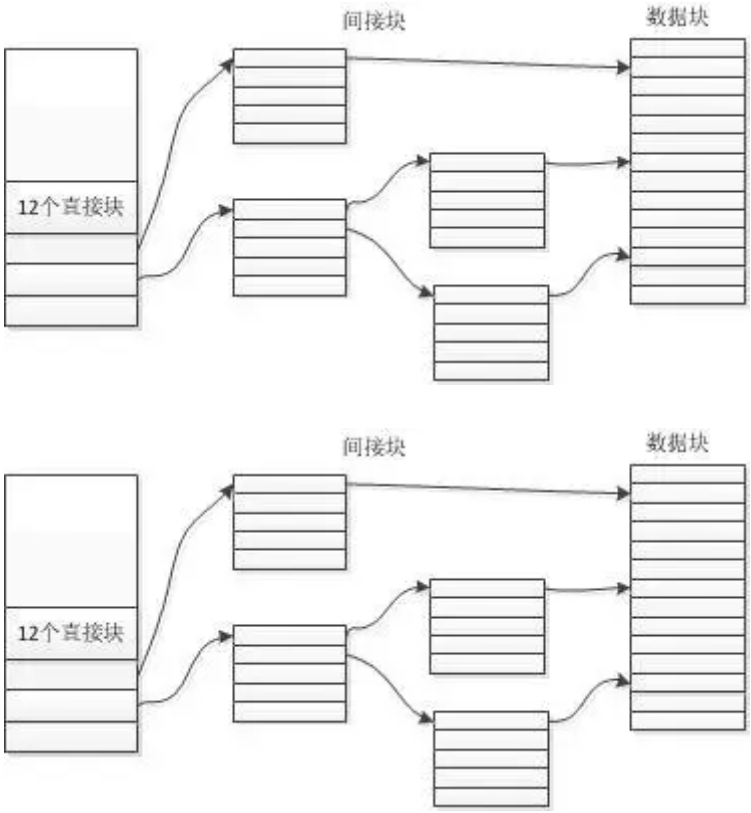
- i_mode：访问权限
- i_size 和 i_block：分别以字节和块为单位指定了文件的长度，需要注意的是，这里总是假定块的大小是 512 字节（和文件系统实际使用的块大小没有关系）
- i_blocks：存储数据块的标号的数组，数组长度为 EXT2_N_BLOCKS，其默认值是 12+3
- i_link_count：统计硬链接的计数器
- 每个块组的 inode 数量也是由文件系统创建时设定，默认的设置是 128

1.5 数据块

数据块保存的是文件的有用数据。

知识点 1：inode 中保存了文件占用的数据块的编号，那么 inode 是怎么保存数据块的编码的？

对于一个 700MB 的文件，如果数据块的长度是 4KB，那么需要 175000 个数据块，而 inode 需要用 175000*4 个字节来存储所有的块号信息，这就需要耗费大量的磁盘空间来存储 inode 信息，更重要的是大多数文件都不需要存储这么多块号。



Linux 使用间接存储的方案来解决这个问题，上图表示了简单间接和二次间接，inode 本身会存储 15 个数据块的标号，其中 12 个直接指向对应的数据块，当文件需要的数据块超过 12 个时，就需要使用间接，间接指向的数据块，用来存储数据块的标号，而不会存储文件数据，同理，当一次间接还不够用时，就需要二次间接，ext2 最高提供三次间接，这样我们就很容易算出文件系统支持的最大文件长度。

〈 如显示不全，请左右滑动 〉

块长度	最大文件长度
1024	16GB
2048	256GB
4096	2TB

知识点 2：将分区分为多个块组有什么好处？

文件系统会试图把文件储存到同一个块组中，以最小化磁头在 inode、块位图和数据块之间寻道的代价，这样可以显著提高磁盘访问速度

知识点 3：目录是怎么存储的？

目录本身也是文件，其同样是有 inode 和对应的数据块，只不过数据块上存放的是描述目录项的的结构，其定义如下。

```
1.  struct ext2_dir_entry_2{
2.              __le32   inode;
3.              __le16   rec_len;
4.              __u8      name_len;
5.              __u8      file_type;
6.              char      name[EXT2_NAME_LEN];
7.  };
```

- file_type:指定了目录的类型，常用的值有 EXT2_FT_REG_FILE 和 EXT2_FT_DIR，分别用来标识文件和目录。
- rec_len: 表示从 rec_len 字段末尾到下一个 rec_len 字段末尾的偏移量，单位是字节，对于删除的文件和目录，不用删除对应的数据，只需要修改 rec_len 的值就可以，用来有效地扫描文件目录。

对于文件系统的创建加载，以及数据块的读取和创建，以及预分配等细节，这里不再赘述。

2 块设备层

块设备有一下几个特点：

1. 可以在数据中的任何位置进行访问
2. 数据总是以固定长度块进行传输
3. 对块设备的访问有大规模的缓存

需要注意是，这里提到的块和上文 ext2 文件系统的块概念是一样。块的最大长度受内存页的长度限制。另外，我们知道磁盘还有一个概念是sector扇区，它表示磁盘读写的最小单位，通常是 512 个字节，块是扇区的整数倍。

块设备层是一个抽象层，用来提高磁盘的读写效率，使用请求队列，来缓存并重排读写数据块的请求，同时提供预读的功能，提供缓存来保存预读取的内容。因此下文将重点介绍请求队列以及调度策略。

2.1 请求队列

请求队列是一个存储了 I/O 请求的双向链表，下面来看表示 I/O 请求的数据结构。

```
1.  struct request{
2.        struct list_head queuelist;
3.        struct list_head_donelist;
4.        struct request_queue *q;
5.        unsigned int cmd_flags;
6.        enum rq_cmd_type_bits cmd_type;
7.        ...
8.        sector_t sector;       /*需要传输的下一个扇区号*/
9.        sector_t hard_sector;     /*需要传输的下一个扇区号*/
10.       unsigned long nr_sectors;     /*还需要传输的扇区数目*/
11.       unsigned long hard_nr_sectors;    /*还需要传输的扇区数目*/
12.       unsigned long current_nr_secotrs; /*当前段中还需要传输的扇区数
    目*/
13.       struct bio *bio;
14.       struct bio *biotail;
15.       ...
16.       void *elevator_private;
17.       void *elevator_private2;
```

```
18.         ...
19. };
```

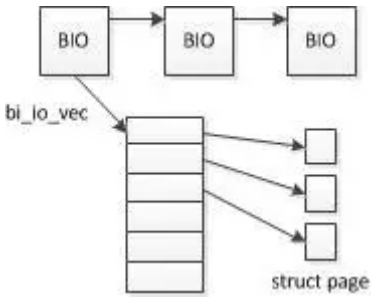
该结构有 3 个成员可以指定所需传输数据的准确位置。

- sector：指定了数据传输的起始扇区
- current_nr_sectors：当前请求在当前段中还需要传输的扇区数目
- nr_sectors：当前请求还需要传输的扇区数目

其中的 bio 和 biotail 字段涉及到另外一个概念 BIO，下面来详述。

2.2 BIO

BIO 用于在系统和设备之间传输数据，其结构如下，主要关联到了一个数组上，数组项则指向一个内存页。



BIO 的数据结构如下：

```
1.  struct bio{
2.    sector_t bi_sector;
3.    struct bio *bi_next; /*将与请求关联的几个 BIO 组织到一个单链表中*/
4.    ...
5.    unsigned short bi_vcnt;    /*bio_vec 的数目*/
6.    unsigned short bi_idx;     /*bi_io_vec 数组中，当前处理数组项的索引
    */
7.    unsigned short bi_phys_segments;
8.    unsigned short bi_hw_segments;
9.    unsigned int bi_size; /*剩余 IO 数据量*/
10.   struct bio_vec *bi_io_vec;     /*实际的 bio_vec 数组*/
11.   bio_end_io_t *bi_end_io;
12.   void   *bi_private;
13. };
```

大体上，内核在提交请求时，可以分两步：

1. 首先创建一个 BIO 实例以描述请求，然后请求的 bio 字段指向创建的 BIO 实例，并把请求置于请求队列上
2. 内核处理请求队列并执行 BIO 中的操作。

2.3 请求插入队列

内核使用队列插入机制，来有意阻止请求的处理，请求队列可能处于空闲状态或者插入状态，如果队列处于空闲状态，队列中等待的请求将会被处理，否则，新的请求只是添加到队列，但并不处理。

2.4 I/O 调度

调度和重排 I/O 操作的算法，称之为 I/O 调度器，也称为电梯（elevator），

目前常用的调度算法有：

- noop：按照先来先服务的原则一次添加到队列，请求会进行合并当无法重排
- deadline：试图最小化磁盘寻道的次数，并尽可能确保请求在一定时间内完成
- as：预测调度器，会尽可能预测进程的行为。
- cfq：完全公平排队，时间片会分配到每个队列，内核使用一个轮转算法来处理各个队列，确保了 I/O 带宽以公平的方式在不同队列之间共享。

3 虚拟文件系统

VFS 是对文件系统的一层抽象，来屏蔽各种文件系统的差异，对于 VFS 来说，其主要操作的对象依然是 inode，这里需要注意的是，内存中的 inode 结构和磁盘上文件系统中的 inode 结构稍有不同。其包含了一些磁盘上 inode 没有的成员。

3.1 inode

VFS 中 inode 结构如下：

```
1. struct inode{
```

```
        struct hlist_node ihash;
        struct list_head i_list;
        struct list_head i_sb_list;
        struct list_head i_dentry;
        ...
        loff_t i_size;
        ...
        unsigned int i_blkbits;
        blkcnt_t i_blocks;
        umode_t i_mode;
        ...
};
```

inode 中没有保存文件名，而文件名保存在目录项 dentry 中，因此，如果应用层需要打开一个给定的文件名，就需要先查找其 dentry，找其对应的 inode，从而找到它在磁盘上的存储位置。

3.2 dentry

```
struct dentry{
    atomic_t d_count;
    unsigned int d_flags;     /*由 d_lock 保护*/
    spinlock_t d_lock;    /*每个 dentry 的锁*/
    struct inode *d_inode;    /*文件名所属的 inode，如果为 NULL，则标识不存在的文件名*/
    struct hlist_node d_hash;     /*用于查找的散列表*
    struct dentry *d_parent;       /*父目录的 dentry 实例*/
    struct qstr d_name;
    ...
    unsigned char d_iname[DNAME_INLINE_LEN_MIN]     /*短文件名存储在这里*/
};
```

上文提到，dentry 的主要用途是建立文件名和 inode 的关联，其中有 3 个重要字段：

- d_inode：指向相关 inode 实例的指针，d_entry 还可以为不存在的文件名建立，这时 d_inode 为 NULL 指针，这有助于查找不存在的文件名
- d_name：指定了文件的名称

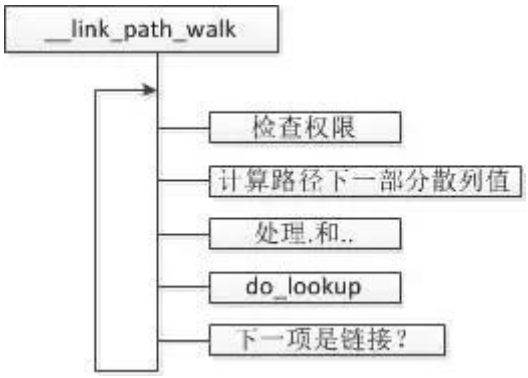
- d_iname：如果文件名有少量字符组成，则存储在该字段，以加速访问

由于块设备的访问速度较慢，为了加速 dentry 的查找，内核使用 dentry 缓存来加速其查找。而 dentry 缓存在内存中的组织形式如下：

1. 一个散列表：包含了所有的 dentry 对象
2. 一个 LRU 链表：不再使用的对象将授予一个最后宽限期，宽限期过后才从内存移除。

对于 VFS 来说，主要的一个工作是查找 inode，下面就介绍 inode 的查找流程。

首先调用 __link_path_walk 来进行权限检查然后主要的逻辑在 do_lookup 中实现。

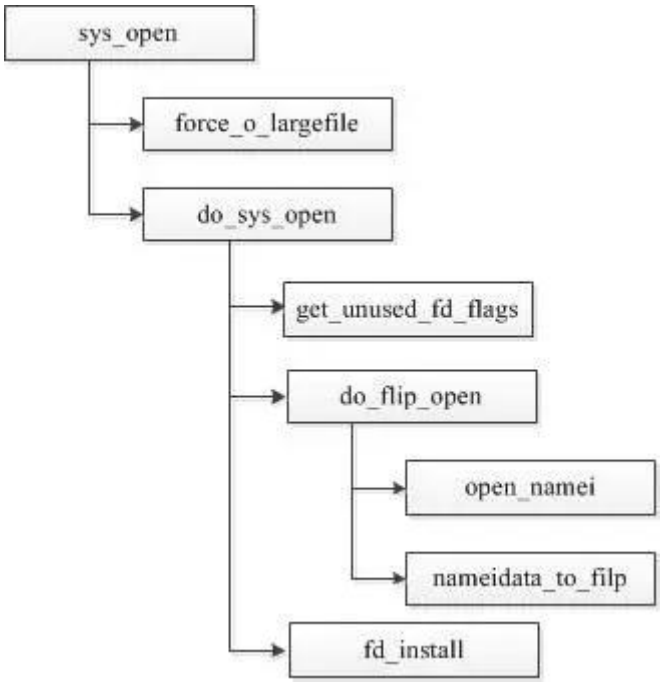


do_lookup 始于一个路径分量，最终返回一个和带查找文件名相关的 inode。

1. 去 dentry 缓存中查找 inode，如果查找到，仍会调用文件系统的 d_revalidate 函数来检查缓存是否有效
2. 调用 read_lookup 执行特定于文件系统的查找操作

3.3 打开文件

对于应用程序来说，通常会调用 open 系统函数来打开一个文件，下面就看下内核处理 open 的流程。



1. 首先检查 force_o_largefile 设置
2. 找到一个可用的文件描述符
3. open_namei 调用 path_lookup 函数查找 inode
4. 将创建的 file 实例放置到 s_files 链表上
5. 将 file 实例放置到进程 task_struct 的 file->fd 数组中
6. 返回到用户层

总结

内核有关磁盘的实现错综复杂，本文试图通过介绍一些关键的概念：ext 文件系统、inode、dentry、BIO 等，让大家了解内核实现磁盘 I/O 的主要原理。本文属于读书笔记，如果有疑问的地方 ，可以直接参阅《深入 Linux 内核架构》。