journal block device (jbd)源代码分析

——ext3 日志机制分析

一、 前言——为什么要写这篇文章

在阅读 ext3 源代码的时候，才对什么是日志型文件系统有了更深刻的了解。内核里单独抽象了一个层次，称之为 journal block device，简称为 jbd，位于 fs/jbd/目录，专门用于块设备的日志管理。细数其源代码，不到万行，但是相关的分析资料，少之又少，有两篇介绍 jbd 概念的，说得比较清楚，但是对着源代码看，仍然不得要领。于是痛下苦功，反复阅读代码，力求得到正解，今感觉略有小成，写出来供大家学习、批评指正，以期去伪存真。

本文分析的内核代码版本 2.6.35，主要是 fs/jbd 和 fs/ext3 两个目录。本文假设物理磁盘块大小 512B，文件系统块大小 1KB。文件系统组织物理磁盘块时是以格式化时设定的块大小为单位的，称谓文件系统块，下文有时会为了行文方便，称文件系统块为磁盘块。

读者在阅读本文前，最好对 Linux VFS 和 ext2 的相关概念比较熟悉，比如 inode、磁盘块位图、三级磁盘块索引、块组等等。

“纸上得来终觉浅，绝知此事要躬行”。希望本文能够起到一个抛砖引玉的作用，要想获得更多的收获，还是得靠自己分析源代码。源码在前，了无秘密。

还有一点说明，我想将本文写成一篇技术文章，并不想写成论文，所以在参考文献的引用方面就不严格按照写论文的方式了，那样太费时间。我把主要的参考文献集中放在文章的末尾。建议读者可以先读读参考文献，再阅读本文。

限于水平有限，代码中有些逻辑我还是没有搞清楚的，有些英文的翻译也欠斟酌，文章排版、详略、章节布局等也有很多待改进的地方。希望大家提出宝贵修改建议。

作者：潘卫平 邮件：panweiping3@163.com 博客：pwp.cublog.cn

二、 提出问题——jbd 要解决什么问题

本章主要是说明 jbd 要解决什么问题，或者说 ext2 的缺点在哪里，因为 ext3 与 ext2 的主要差别就在于 ext3 在 ext2 的基础上增加了日志功能。

假设你正在运行一个 Linux 系统，运行一个程序，在一个 ext2 分区上不断地读写磁盘文件。突然断电了，或者系统崩溃了，你的心里肯定会咯噔一下：“磁盘分区没坏吧？文件还完整么？”告诉你一个不幸的消息，文件可能不完整了，文件可能已经损坏了，甚至该分区不能再被挂载了。也就是说，意外的系统崩溃，可能会使 ext2 文件系统处于一个不一致的状态。

假设你的运气好一点，分区仍能被识别，但是重新挂载时，如果发现分区处于不一致状态，那么系统会自动调用 fsck 程序，尝试将文件系统恢复到一致的状态。那将是一个非常漫长的过程，并且随着磁盘容量的增大，花费的时间也越长，有时需要长达几个小时。这样会极大地影响系统的可用性。

总之，jbd 的主要目的不是减少系统崩溃的概率，而是系统正常运行时，尽量使文件系统处于一个一致的状态，以及系统崩溃后，尽可能减少使文件系统重新处于一致性状态的时间。通过减少维护时间，增加系统的可用性。

三、 解决问题——jbd 是如何解决的

本章从总体上说明 ext3 采用什么方式解决第二章提出的问题。

提到一致性，大家可能会联想到数据库里面的事务的概念，因为事务有四个基本属性：

1) 原子性

事务必须是原子工作单元；对于其数据修改，要么全都执行，要么全都不执行。

2) 一致性

事务在完成时，必须使所有的数据都保持一致状态。

3) 隔离性

由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务识别数据时数据所处的状态，要么是另一并发事务修改它之前的状态，要么是第二个事务修改它之后的状态，事务不会识别中间状态的数据。

4) 持久性

事务完成之后，它对于系统的影响是永久性的。该修改即使出现系统故障也将一直保持。

文件系统的开发者借用了数据库中事务的思想，将其应用于文件系统上，以期保证对文件系统操作的原子性、隔离性，尽量使文件系统处于一致性。

1．将对文件系统的某些操作抽象成原子操作

所谓原子操作，就是内部不再分割的操作，该操作要么完全完成，要么根本没有执行，不存在部分完成的状态。

那么，什么样的操作可以看成对文件系统的原子操作呢？往一个磁盘文件中追加写入 1MB 字节可以看成一个原子操作么？这个操作其实比较大，因为要写 1MB 的数据，要为文件分配 1024 个磁盘块，同时还要分配若干个索引块，也会涉及到很多的磁盘块位图、块组块的读写，非常复杂，时间也会比较长，中间出问题的机会就比较多，所以不适宜看做一个原子操作。

那么，什么样的操作可以看成对文件系统的原子操作呢？比如说为文件分配一个磁盘块，就看成一个原子操作就比较合适。分配一个磁盘块，可能需要修改一个 inode 块、一个磁盘块位图、最多三个间接索引块、块组块、超级块，一共最多 7 个磁盘块。将分配一个磁盘块看成一个原子操作，意味着上述修改 7 个磁盘块的操作要么都成功，要么都失败，不可能有第三种状态。

2．将若干个原子操作组合成一个事务

实现日志文件系统时，可以将一个原子操作就作为一个事务来处理，但是这样实现的效率比较低。于是 ext3 中将若干个原子操作组合成一个事务，对磁盘日志以事务为单位进行管理，以提高读写日志的效率。

3．在磁盘上单独划分一个日志空间

日志，在这里指的是磁盘上存储事务数据的那个地方，即若干磁盘块。它可以以一个单独的文件形式存在，也可以由文件系统预留一个 inode 和一些磁盘块，也可以是一个单独的磁盘分区。总之，就是磁盘上存储事务数据的那个地方。

提到日志时，可能还有另外一种含义，就是指它是一种机制，用于管理内存中的缓冲区、事务、磁盘日志数据读写等等所有这一切，统称为日志。读者注意根据上下文进行区分。

4．将内存中事务的数据写到日志中

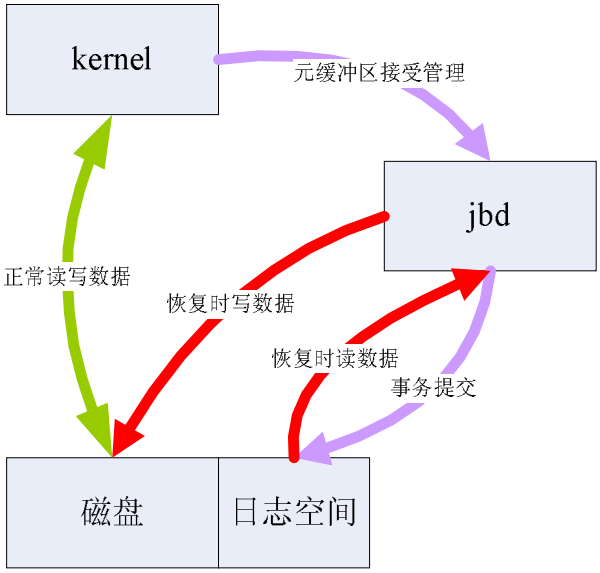
文件系统可以选择定期（每隔 5 秒，或用户指定的时间间隔）或者立即将内存中的事务数据写到磁盘日志上，以备发生系统崩溃后可以利用日志中的数据恢复，重新使文件系统保持一致的状态。

这个间隔时间的选取，要注意性能的平衡。时间间隔越短，文件系统丢失的数据可能性就越少，一致性的时间点就越新，但是 IO 负担就越重，很可能就会影响系统的性能。反过来，时间间隔越大，文件系统丢失的数据可能就越多，一致性的时间点就越旧，但是 IO 负担就比较轻，不太会影响系统的性能。

5．崩溃吧，然后我们从日志中恢复数据

我们不期望崩溃，但是崩溃总会发生的，如果发生了，那就直面惨淡的人生吧！重新挂载分区时，会根据日志中记录的数据，逐一将其写回到磁盘原始位置上，以保证文件系统的一致性。起死回生的奇迹发生了。

jbd 的思想就是原来内核读写磁盘的逻辑保持不变，但是对于影响文件系统一致性的数据块（即元数据块，第四章会详细解释），及时地写到磁盘上的日志空间中去。这样，即使系统崩溃了，也能从日志中恢复数据，确保文件系统的一致性。如**错误！未找到引用源。**，其中绿色的箭头表示正常的磁盘读写，紫色的箭头表示由 jbd 将元数据块额外写一份到磁盘日志中，红色箭头表示恢复时，由 jbd 将日志中的数据写回磁盘的原始位置。



图表 1　jbd 数据流图

四、 介绍几个概念——名正然后言顺

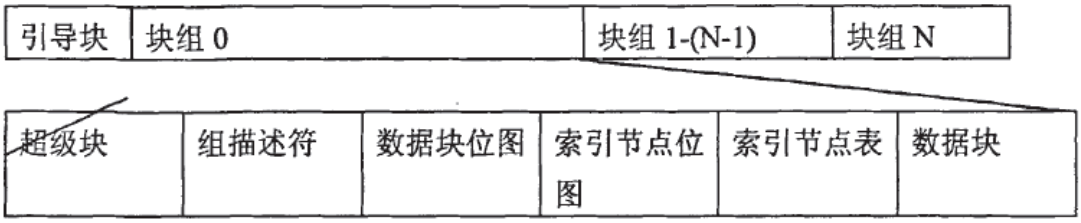
1. buffer_head

buffer_head 是内核一个用于管理磁盘缓冲区的数据结构。根据局部性原理，磁盘上的数据进入内存后一般都是存放在磁盘缓冲区中的，以备将来重复读写。所以说，一个 buffer_head 就会对应一个文件系统块，即对应一个磁盘块。

2. 元数据块

笼统地，可以将一个文件系统内的块分为两种，一种是对文件系统的一致性有重要影响的、用于文件系统管理的磁盘块，称之为元数据块，包括超级块、磁盘位图块、inode 位图块、索引块、块组描述符块等等；另一种是存放文件数据的，称之为数据块。

因为元数据块对文件系统的一致性有至关重要的影响，故 jbd 主要处理元数据块。当然，ext3 的日志可以设置为三种模式，不同的模式中 jbd 处理的数据也是不一样的，这个下文会详述。ext3 磁盘物理布局参考图表 2　ext3 磁盘物理布局



图表 2　ext3 磁盘物理布局

3. handle

第三章中提到的原子操作，jbd 中用 handle 来表示。一个 handle 代表针对文件系统的一次原子操作。这个原子操作要么成功，要么失败，不会出现中间状态。在一个 handle 中，可能会修改若干个缓冲区，即 buffer_head。

4. transaction

第三章中也提到，jbd 为了提高效率，将若干个 handle 组成一个事务，用 transaction 来表示。对日志读写来说，都是以 transaction 为单位的。在处理日志数据时，transaction 具有原子性，即恢复时，如果一个 transaction 是完整的，其中包含的数据就可用于文件系统的恢复，否则，忽略不完整的 transaction。

5. journal

journal 在英文中有“日志”之意，在 jbd 中 journal 既是磁盘上日志空间的代表，又起到管理内存中为日志机制而创建的 handle、transaction 等数据结构的作用，可以说是整个日志机制的代表。

6. commit

所谓提交，就是把内存中 transaction 中的磁盘缓冲区中的数据写到磁盘的日志空间上。注意，jbd 是将缓冲区中的数据另外写一份，写到日志上，原来的 kernel 将缓冲区写回磁盘

的过程并没有改变。

在内存中，transaction 是可以有若干个的，而不是只有一个。transaction 可分为三种，一种是已经 commit 到磁盘日志中的，它们正在进行 checkpoint 操作（详见 7）；第二种是正在将数据提交到日志的 transaction；第三种是正在运行的 transaction。正在运行的 transaction 管理随后发生的 handle，并在适当时间 commit 到磁盘日志中。注意正在运行的 transaction 最多只可能有一个，也可能没有，如果没有，则 handle 提出请求时，则会按需要创建一个正在运行的 transaction。

7. checkpoint

当一个 transaction 已经 commit，那么，是不是在内存中它就没有用了呢？好像是这样，因为其中的数据已经写到磁盘日志中了。但是实际上不是这样的。主要原因是磁盘日志是个有限的空间，比如说 100MB，如果一直提交 transaction，很快就会占满，所以日志空间必须复用。

其实与日志提交的同时，kernel 也在按照自己以前的方式将数据写回磁盘。试想，如果一个 transaction 中包含的所有磁盘缓冲区的数据都已写回到磁盘的原来的位置上（不是日志中，而是在磁盘的原来的物理块上），那么，该 transaction 就没有用了，可以被删除了，该 transaction 在磁盘日志中的空间就可以被回收，进而重复利用了。

8. revoke

假设有一个缓冲区，对应着一个磁盘块，内核多次修改该缓冲区，于是磁盘日志中就会有该缓冲区的若干个版本的数据。假设此时要从文件中删除该磁盘块，那么，一旦包含该删除操作的 transaction 提交，那么，再恢复时，已经存放在磁盘日志中的该磁盘块的若干个版本的数据就不必再恢复了，因为到头来还是要删除的。revoke 就是这样一种加速恢复速度的方法。当本 transaction 包含删除磁盘块操作时，就会在磁盘日志中写一个 revoke 块，该块中包含<被 revoked 的块号 blocknr，提交的 transaction 的 ID>，表示恢复时，凡是 transaction ID 小于等于 ID 的所有写磁盘块 blocknr 的操作都可以取消了，不必进行了。

9. recover

加入日志机制后，一旦系统崩溃，重新挂载分区时，就会检查该分区上的日志是否需要恢复。如果需要，则依次将日志空间的数据写回磁盘原始位置，则文件系统又重新处于一致状态了。

10. kjournald

日志的提交操作是由一个内核线程实现的，该线程称为 kjournald。该内核线程平时一直在睡眠，直到有进程主动唤醒它，或者是定时器时间到了（一般为每隔 5 秒）。被唤醒后它就进行事务的提交操作。

五、 介绍几个数据结构——珍珠

以下数据结构的定义在 include/linux/jbd.h 和 include/linux/journal_head.h 中。

1. handle_t

表示一个原子操作。

```
struct handle_s
{
    transaction_t   *h_transaction;     // 本原子操作属于哪个 transaction
    int             h_buffer_credits;   // 本原子操作的额度，即可以包含的磁盘块数
    int             h_ref;              // 引用计数
    int             h_err;

    unsigned int    h_sync:         1;  /* sync-on-close */
    unsigned int    h_jdata:    1;  /* force data journaling */
    unsigned int    h_aborted:      1;  /* fatal error on handle */
    // 以上是三个标志
    // h_sync 表示同步，意思是处理完该原子操作后，立即将所属的 transaction 提交。
    // 其余两个好像没有用到。
};
    typedef struct handle_s handle_t;   /* Atomic operation type */
```

注意 jbd 中数据结构定义一般都采用这种方式，即定义结构时用 XXX_s，然后用 typedef 定义 XXX_t。下面不再特别指出了。

乍看这个表示原子操作的结构有些奇怪，它怎么不包含缓冲区呢？其实 handle_t 的主要目的是顺着它能找到对应的 transaction。如果你想把一些缓冲区纳入日志管理，需要另外的步骤。

2. transaction_t

表示一个事务。

```
struct transaction_s
{
    journal_t       *t_journal;     // 指向所属的 jounal
    tid_t           t_tid;          // 本事务的序号

    /*
     * Transaction's current state
     * [no locking - only kjournald alters this]
     * [j_list_lock] guards transition of a transaction into T_FINISHED
     * state and subsequent call of __journal_drop_transaction()
     * FIXME: needs barriers
     * KLUDGE: [use j_state_lock]
     */
    enum {
        T_RUNNING,
```

```
        T_LOCKED,
        T_FLUSH,
        T_COMMIT,
        T_COMMIT_RECORD,
        T_FINISHED
} t_state;   // 事务的状态

unsigned int            t_log_start;
// log 中本 transaction_t 从日志中哪个块开始
int                     t_nr_buffers;
// 本 transaction_t 中缓冲区的个数

struct journal_head *t_reserved_list;
// 被本 transaction 保留，但是并未修改的缓冲区组成的双向循环队列。
struct journal_head *t_locked_list;
// 由提交时所有正在被写出的、被锁住的数据缓冲区组成的双向循环链表。
struct journal_head *t_buffers;
// 元数据块缓冲区链表
// 这里面可都是宝贵的元数据啊，对文件系统的一致性至关重要！
struct journal_head *t_sync_datalist;
// 本 transaction_t 被提交之前，
// 需要被刷新到磁盘上的数据块（非元数据块）组成的双向链表。
// 因为在 ordered 模式，我们要保证先刷新数据块，再刷新元数据块。
struct journal_head *t_forget;
// 被遗忘的缓冲区的链表。
// 当本 transaction 提交后，可以 un-checkpointed 的缓冲区。
// 这种情况是这样：
// 一个缓冲区正在被 checkpointed，但是后来又调用 journal_forget(),
// 此时以前的 checkpointed 项就没有用了。
// 此时需要在这里记录下来这个缓冲区，
// 然后 un-checkpointed 这个缓冲区。
struct journal_head *t_checkpoint_list;
// 本 transaction_t 可被 checkpointed 之前，
// 需要被刷新到磁盘上的所有缓冲区组成的双向链表。
// 这里面应该只包括元数据缓冲区。
struct journal_head *t_checkpoint_io_list;
// checkpointing 时，已提交进行 IO 操作的所有缓冲区组成的链表。
struct journal_head *t_iobuf_list;
// 进行临时性 IO 的元数据缓冲区的双向链表。
struct journal_head *t_shadow_list;
// 被日志 IO 复制（拷贝）过的元数据缓冲区组成的双向循环链表。
// t_iobuf_list 上的缓冲区始终与 t_shadow_list 上的缓冲区一一对应。
// 实际上，当一个元数据块缓冲区要被写到日志中时，数据会被复制一份，
// 放到新的缓冲区中。
```

```
    // 新缓冲区会进入 t_iobuf_list 队列，
    // 而原来的缓冲区会进入 t_shadow_list 队列。
    struct journal_head *t_log_list;
    // 正在写入 log 的起控制作用的缓冲区组成的链表。
    spinlock_t          t_handle_lock;
    // 保护 handle 的锁
    int                 t_updates;
    // 与本 transaction 相关联的外部更新的次数
    // 实际上是正在使用本 transaction 的 handle 的数量
    // 当 journal_start 时，t_updates++
    // 当 journal_stop 时，t_updates--
    // t_updates == 0，表示没有 handle 正在使用该 transaction，
    // 此时 transaction 处于一种可提交状态！
    int                 t_outstanding_credits;
    // 本事务预留的额度
    transaction_t       *t_cpnext, *t_cpprev;
    // 用于在 checkpoint 队列上组成链表
    unsigned long       t_expires;
    ktime_t             t_start_time;
    int t_handle_count;
    // 本 transaction_t 有多少个 handle_t
    unsigned int t_synchronous_commit:1;
    // 本 transaction 已被逼迫了，有进程在等待它的完成。
};
```

3. journal_t

journal_t 是整个日志机制的代表，既管理者内存中的各种日志相关的数据结构，又管理着磁盘上的日志空间。

```
struct journal_s
{
    unsigned long       j_flags;    // journal 的状态
    int                 j_errno;
    struct buffer_head  *j_sb_buffer;  // 指向日志超级块缓冲区
    journal_superblock_t    *j_superblock;
    int                 j_format_version;
    spinlock_t          j_state_lock;
    int                 j_barrier_count;
    // 有多少个进程正在等待创建一个 barrier lock
    // 这个变量是由 j_state_lock 来保护的。
    struct mutex        j_barrier;
    // 互斥锁
    transaction_t       *j_running_transaction;
    // 指向正在运行的 transaction
    transaction_t       *j_committing_transaction;
```

```
// 指向正在提交的 transaction
transaction_t        *j_checkpoint_transactions;
// 仍在等待进行 checkpoint 操作的所有事务组成的循环队列
// 一旦一个 transaction 执行 checkpoint 完成，则从此队列删除。
// 第一项是最旧的 transaction，以此类推。
wait_queue_head_t j_wait_transaction_locked;
// 等待一个已上锁的 transaction_t 开始提交，
// 或者一个 barrier 锁被释放。
wait_queue_head_t j_wait_logspace;
// 等待 checkpointing 完成以释放日志空间的等待队列。
wait_queue_head_t j_wait_done_commit;
//等待提交完成的等待队列
wait_queue_head_t j_wait_checkpoint;

wait_queue_head_t j_wait_commit;
// 等待进行提交的的等待队列
wait_queue_head_t j_wait_updates;
// 等待 handle 完成的等待队列
struct mutex         j_checkpoint_mutex;
// 保护 checkpoint 队列的互斥锁。
unsigned int         j_head;
// journal 中第一个未使用的块
unsigned int         j_tail;
// journal 中仍在使用的最旧的块号
// 这个值为 0，则整个 journal 是空的。
unsigned int         j_free;

unsigned int         j_first;
unsigned int         j_last;
// 这两个是文件系统格式化以后就保存到超级块中的不变的量。
// 日志块的范围[j_first, j_last)
// 来自于 journal_superblock_t

struct block_device *j_dev;
int                  j_blocksize;
unsigned int         j_blk_offset;
// 本 journal 相对与设备的块偏移量
struct block_device *j_fs_dev;

unsigned int         j_maxlen;
// 磁盘上 journal 的最大块数

spinlock_t           j_list_lock;
```

```
struct inode            *j_inode;

tid_t               j_tail_sequence;
// 日志中最旧的事务的序号

tid_t               j_transaction_sequence;
// 下一个授权的事务的顺序号

tid_t               j_commit_sequence;
// 最近提交的 transaction 的顺序号

tid_t               j_commit_request;
// 最近相申请提交的 transaction 的编号。
// 如果一个 transaction 想提交，则把自己的编号赋值给 j_commit_request，
// 然后 kjournald 会择机进行处理。

__u8                    j_uuid[16];

struct task_struct    *j_task;
// 本 journal 指向的内核线程

int                 j_max_transaction_buffers;
// 一次提交允许的最多的元数据缓冲区块数

unsigned long       j_commit_interval;

struct timer_list       j_commit_timer;
// 用于唤醒提交日志的内核线程的定时器

spinlock_t              j_revoke_lock;
// 保护 revoke  哈希表
struct jbd_revoke_table_s *j_revoke;
// 指向 journal 正在使用的 revoke hash table
struct jbd_revoke_table_s *j_revoke_table[2];

struct buffer_head   **j_wbuf;
// 指向描述符块页面

int                 j_wbufsize;
// 一个描述符块中可以记录的块数

pid_t                   j_last_sync_writer;

u64                 j_average_commit_time;
```

```
    void *j_private;
    // 指向 ext3 的 superblock
};
```

4. journal_superblock_t

这个结构是日志超级块在内存中的表现。

```
/*
 * The journal superblock.    All fields are in big-endian byte order.
 */
typedef struct journal_superblock_s
{
    journal_header_t s_header;    // 用于表示本块是一个超级块
    __be32    s_blocksize;        /* journal device blocksize */
                                  // journal 所在设备的块大小
    __be32    s_maxlen;           /* total blocks in journal file */
                                  // 日志的长度，即包含多少个块
    __be32    s_first;          /* first block of log information */
                                  // 日志中的开始块号，
                                  // 注意日志相当于一个文件，
                                  // 这里提到的开始块号是文件中的逻辑块号，
                                  // 而不是磁盘的物理块号。
                                  // 初始化时置为 1，因为超级块本身占用了逻辑块 0。
                                  // 注意 s_maxlen 和 s_first 是在格式化时确定的，
                                  // 以后就不会改变了。
    __be32    s_sequence;         /* first commit ID expected in log */
                                  // 日志中第一个期待的 commit ID
                                  // 就是指该值应该是日志中最旧的一个事务的 ID
    __be32    s_start;          /* blocknr of start of log */
                                  // 日志开始的块号
                                  // s_start 为 0 表示不需要恢复
                                  // 因为日志空间需要重复使用，相当于一个环形结构，
                                  // s_start 表示本次有效日志块的起点
    __be32    s_errno;

    // 注意：下列各域只有在 superblock v2 中才有效
    /* Remaining fields are only valid in a version-2 superblock */
    __be32    s_feature_compat;  /* compatible feature set */
    __be32    s_feature_incompat;       /* incompatible feature set */
    __be32    s_feature_ro_compat;     /* readonly-compatible feature set */
    __u8      s_uuid[16];         /* 128-bit uuid for journal */
    __be32    s_nr_users;         /* Nr of filesystems sharing log */
    __be32    s_dynsuper;         /* Blocknr of dynamic superblock copy*/
    __be32    s_max_transaction; /* Limit of journal blocks per trans.*/
```

```
    __be32    s_max_trans_data;  /* Limit of data blocks per trans. */
    __u32     s_padding[44];
    __u8      s_users[16*48];            /* ids of all fs'es sharing the log */
} journal_superblock_t;
```

5. journal_head

一个 buffer_head 对应一个磁盘块，而一个 journal_head 对应一个 buffer_head。日志通过 journal_head 对缓冲区进行管理。

```
struct journal_head {
    struct buffer_head *b_bh;
    int b_jcount;
    unsigned b_jlist;
    // 本 journal_head 在 transaction_t 的哪个链表上
    unsigned b_modified;
    // 标志该缓冲区是否以被当前正在运行的 transaction 修改过
    char *b_frozen_data;
    // 当 jbd 遇到需要转义的块时，
    // 将 buffer_head 指向的缓冲区数据拷贝出来，冻结起来，供写入日志使用。
    char *b_committed_data;
    // 目的是防止重新写未提交的删除操作
    // 含有未提交的删除信息的元数据块（磁盘块位图）的一份拷贝，
    // 因此随后的分配操作可以避免覆盖未提交的删除信息。
    // 也就是说随后的分配操作使用的时 b_committed_data 中的数据，
    // 因此不会影响到写入日志中的数据。
    transaction_t *b_transaction;
    // 指向所属的 transaction
    transaction_t *b_next_transaction;
    // 当有一个 transaction 正在提交本缓冲区，
    // 但是另一个 transaction 要修改本元数据缓冲区的数据，
    // 该指针就指向第二个缓冲区。

    /*
     * Doubly-linked list of buffers on a transaction's data, metadata or
     * forget queue. [t_list_lock] [jbd_lock_bh_state()]
     */
    struct journal_head *b_tnext, *b_tprev;

    transaction_t *b_cp_transaction;
    // 指向 checkpoint 本缓冲区的 transaction。
    // 只有脏的缓冲区可以被 checkpointed。

    struct journal_head *b_cpnext, *b_cpprev;
    // 在旧的 transaction_t 被 checkpointed 之前必须被刷新的缓冲区双向链表。
```

```
	/* Trigger type */
	struct jbd2_buffer_trigger_type *b_triggers;
	struct jbd2_buffer_trigger_type *b_frozen_triggers;
};
```

6. journal_header_t

每个块的开头，都有一个起描述作用的结构，定义如下：

```
/*
 * Standard header for all descriptor blocks:
 */
typedef struct journal_header_s
{
	__be32		h_magic;
	__be32		h_blocktype;
	__be32		h_sequence;
} journal_header_t;
```

其中，h_magic 是一个幻数，如果是一个日志块的描述块，则为 JFS_MAGIC_NUMBER，#define JFS_MAGIC_NUMBER 0xc03b3998U，否则该块就不是一个日志描述块。

h_blocktype 表示该块的类型，即上述五种块之一。

h_sequence 表示本描述块对应的 transaction 的序号。

六、 日志在磁盘上的布局——一图胜千言

日志中是以文件系统块为单位组织的，总体上可分为两种，一种是数据块，另一种是描述块。

注意这里的划分与第四章第 2 节的含义是不同的。这里所谓的数据块，即表示块里存放的是文件系统数据的块，既可以是文件系统的元数据块，又可以是文件系统的数据块。而这里的描述块，则是日志中特有的、为日志机制特别设置的、起组织管理日志数据作用的块。

存储一共有五种块，定义如下

```
#define JFS_DESCRIPTOR_BLOCK   1
#define JFS_COMMIT_BLOCK   2
#define JFS_SUPERBLOCK_V1  3
#define JFS_SUPERBLOCK_V2  4
#define JFS_REVOKE_BLOCK   5
```

1．超级块 JFS_SUPERBLOCK_V1、JFS_SUPERBLOCK_V2

超级块一共有两种格式，以前使用第一种，现在都使用第二种了。

journal_superblock_t 结构就是超级块在内存中的表示。如表格 1 超级块示意图

日志中超级块起的作用与文件系统中超级块的作用是类似的，都是用于组织管理一段磁盘空间。所以日志从一方面看，是一个文件，但是从另一个方面看，又可看做一个小的文件系统。

数据类型			含义
journal_superblock_t	journal_header_t 表示本块是一个描述块 （准确地说是超级块）	h_magic	一个幻数，表示一个日志描述块 JFS_MAGIC_NUMBER
		h_blocktype	日志描述块的类型 JFS_SUPERBLOCK_V2
		h_sequence	本描述块对应的 transaction 的序号
	s_blocksize		journal 所在设备的块大小
	s_maxlen		日志的长度，即包含多少个块
	s_first		日志中的开始块号， 注意日志相当于一个文件， 这里提到的开始块号是文件中的逻辑块号， 而不是磁盘的物理块号。 初始化时置为 1，因为超级块本身占用了逻辑块 0。 注意 s_maxlen 和 s_first 是在格式化时确定的， 以后就不会改变了。
	s_sequence		日志中第一个期待的 commit ID 就是指该值应该是日志中最旧的一个事务的 ID
	s_start		日志开始的块号 s_start 为 0 表示不需要恢复 因为日志空间需要重复使用，相当于一个环形结构， s_start 表示本次有效日志块的起点
	s_errno		jbd 出错标志
	s_feature_compat		兼容特性的位图
	s_feature_incompat		不兼容特性的位图
	s_feature_ro_compat		只读特性的位图
	s_uuid[16]		UUID，复制自文件系统的 UUID
	s_nr_users		共享使用本日志的用户数
	s_dynsuper		未使用
	s_max_transaction		未使用
	s_max_trans_data		未使用
	s_padding[44]		未使用
	s_users[16*48]		未使用

表格 1 超级块示意图

2．描述符块 JFS_DESCRIPTOR_BLOCK

日志存放的块的内容，来自于原来的文件系统，并且最终也要写回到原来的文件系统。比如说，原来一个 ext3 文件系统的第 1078 块是一个元数据块，进入内核，被修改了，然后先写到日志中的第 37 块中。假设此时系统崩溃了，那么恢复时，我们要把日志中的第 37 块中的数据，写回到文件系统的第 1078 块中。也就是说，37 与 1078 之间，有一种对应关系，用于指示恢复时，要把日志中的哪一块写回到文件系统中的哪一块。这种对应关系，就记录在描述符块中。

```
/*
 * The block tag: used to describe a single buffer in the journal
 */
typedef struct journal_block_tag_s
{
    __be32          t_blocknr;      /* The on-disk block number */
    __be32          t_flags;    /* See below */
} journal_block_tag_t;
```

其中，t_blocknr 表示日志中的本块对应磁盘原始位置的块号。

t_flags 是下列 4 中类型之一，

```
/* Definitions for the journal tag flags word: */
#define JFS_FLAG_ESCAPE             1   /* on-disk block is escaped */
#define JFS_FLAG_SAME_UUID      2   /* block has same uuid as previous */
#define JFS_FLAG_DELETED    4   /* block deleted by this transaction */
#define JFS_FLAG_LAST_TAG   8   /* last tag in this descriptor block */
```

JFS_FLAG_ESCAPE 表示该块的数据被转义了。

JFS_FLAG_SAME_UUID 表示这一项的 UUID 与上一项的相同。

JFS_FLAG_DELETED 在 jbd 中没有使用。

JFS_FLAG_LAST_TAG 表示是最后一项。

在描述符块中，使用一个 journal_block_tag_t 结构来描述日志中的一个块与磁盘上的一个块的对应关系的。

一个 transaction 在日志中会对应三部分，第一部分是一个描述符块，第二部分是若干个数据块，第三部分是一个提交块。描述符块的开头是一个 journal_header_t 结构，用于表示本块是一个描述块。然后是若干个索引项，用于表示描述符块之后的数据块与磁盘原始块的对应关系。

注意：

1) UUID

第一个索引项包含一个 128 位的 UUID，来自于该块所属的文件系统。为了节省空间，从第二个索引项开始，如果所属的文件系统没有变化，则在 journal_block_tag_t-> t_flags 中设置一个标志 JFS_FLAG_SAME_UUID，这样本索引项中就不用再保存 UUID 了。

2) 如何表示日志块

描述符块的主要作用是描述一个日志块对应哪个磁盘块，其中，目标磁盘块存在 journal_block_tag_t-> t_blocknr 中，而日志块，则是根据索引项的序号确定的。即，第一个索引项（编号为 0），描述的是日志中在本描述符块之后的第一个数据块，即表格 2 中的数据块 0。

3) 最后一个索引项

一个 transaction 对应一个描述符块，但是 transaction 中的缓冲区的个数是不定的，不一定能正好占满一个描述符块。JFS_FLAG_LAST_TAG 标志表示这是本描述符块中的最后一个有效的索引项。

4) 被转义的数据块

描述符块之后就是数据块了。数据块中的内容是任意的，可能某个数据块的开头的 4 个字节正好是 0xc03b3998U ，即 JFS_MAGIC_NUMBER，那怎么办呢？这个块明明是个普通的数据块，但是因为开头的偶然的 4 个字节，就会被 jbd 误认为是一个具有特殊含义的描述块了。

解决方法类似于 shell 中的转义字符的概念。当要往日志中写一块普通的数据块时，如果发现其开头的 4 个字节恰好是 0xc03b3998U，则将该 4 个字节改写成 0，并且在描述符块的索引项中设置 JFS_FLAG_ESCAPE，用于表示该日志块中的数据是被转义过的。恢复时，如果在索引项中发现了 JFS_FLAG_ESCAPE 标志，则重新把该块的前 4 个字节设置成 0xc03b3998U，然后写回到磁盘的原始位置上。

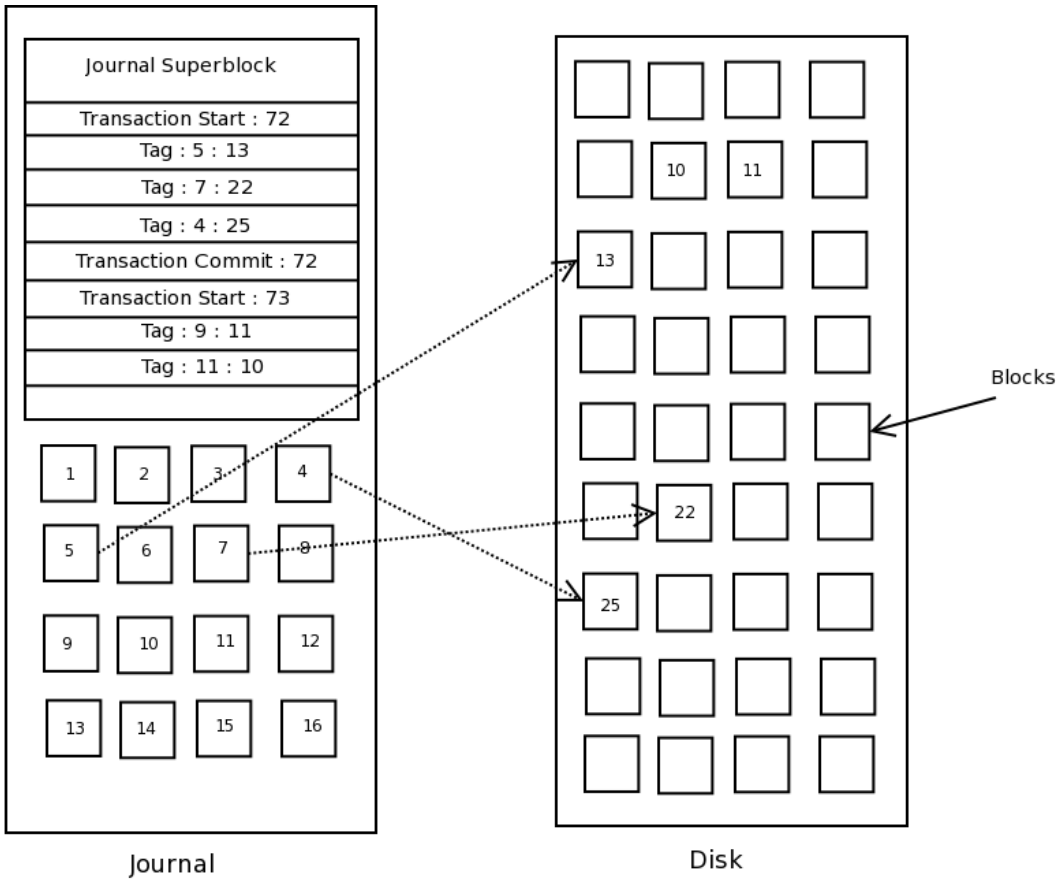
日志中一个描述符块的内容，可参见表格 2 描述符块示意图

<table>
<tr><td rowspan="8">描述符块</td><td colspan="3">journal_header_t
表示本块是一个描述块</td></tr>
<tr><td rowspan="2">0</td><td>journal_block_tag_t</td><td>t_blocknr = 124</td></tr>
<tr><td>UUID</td><td>t_flags</td></tr>
<tr><td>1</td><td>journal_block_tag_t</td><td>t_blocknr = 234
t_flags = JFS_FLAG_SAME_UUID</td></tr>
<tr><td>2</td><td>journal_block_tag_t</td><td>t_blocknr = 284
t_flags = JFS_FLAG_SAME_UUID|JFS_FLAG_ESCAP</td></tr>
<tr><td>……</td><td></td><td></td></tr>
<tr><td rowspan="2">n</td><td>journal_block_tag_t</td><td>t_blocknr = 789</td></tr>
<tr><td></td><td>t_flags = JFS_FLAG_SAME_UUID|JFS_FLAG_LAST_TAG</td></tr>
<tr><td colspan="4">数据块 0</td></tr>
<tr><td colspan="4">数据块 1</td></tr>
<tr><td colspan="4">数据块 2</td></tr>
<tr><td colspan="4">数据块 3</td></tr>
<tr><td colspan="4">……</td></tr>
<tr><td colspan="4">数据块 n</td></tr>
<tr><td colspan="4">提交块</td></tr>
</table>

表格 2 描述符块示意图

注意，表格 2 描述符块示意图
中的 t_flags 域应该根据实际情况设置，这里只是示意。

日志中一块真实描述符块示意图如图表 3 日志中描述符块内容的示意图



图表 3 日志中描述符块内容的示意图

3. 提交块 JFS_COMMIT_BLOCK

提交块用于表明本 transaction 是一个完整的 transaction，恢复时可以使用。

每个 transaction 提交时，当把所有的数据都写完了之后，就会写入一个提交块，表示我们已经写入了一个完整的 transaction。其实提交块中就只有一个 journal_header_t 结构。

数据类型		含义
journal_header_t 表示本块是一个描述块 （准确地说是提交块）	h_magic	一个幻数，表示一个日志描述块 JFS_MAGIC_NUMBER
	h_blocktype	日志描述块的类型 JFS_COMMIT_BLOCK
	h_sequence	本描述块对应的 transaction 的序号
未使用		

表格 3 提交块示意图

4. 取消块 JFS_REVOKE_BLOCK

第四章第 8 节中提到，取消块是为了加快恢复速度而设置的，里面保存着一个 transaction ID 和一些块号，用于表示恢复时这些块不用被恢复。

取消块中只记录块号。

数据类型		含义
journal_header_t 表示本块是一个描述块 （准确地说是提交块）	h_magic	一个幻数，表示一个日志描述块 JFS_MAGIC_NUMBER
	h_blocktype	日志描述块的类型 JFS_ REVOKE_BLOCK
	h_sequence	本描述块对应的 transaction 的序号,假设为 67
	r_count	这个是取消块中一个比较特殊的域， 表示取消块中实际使用的字节数，<=块的长度
0	18976	
1	2323	
2	2324545	
……	……	
n	343	

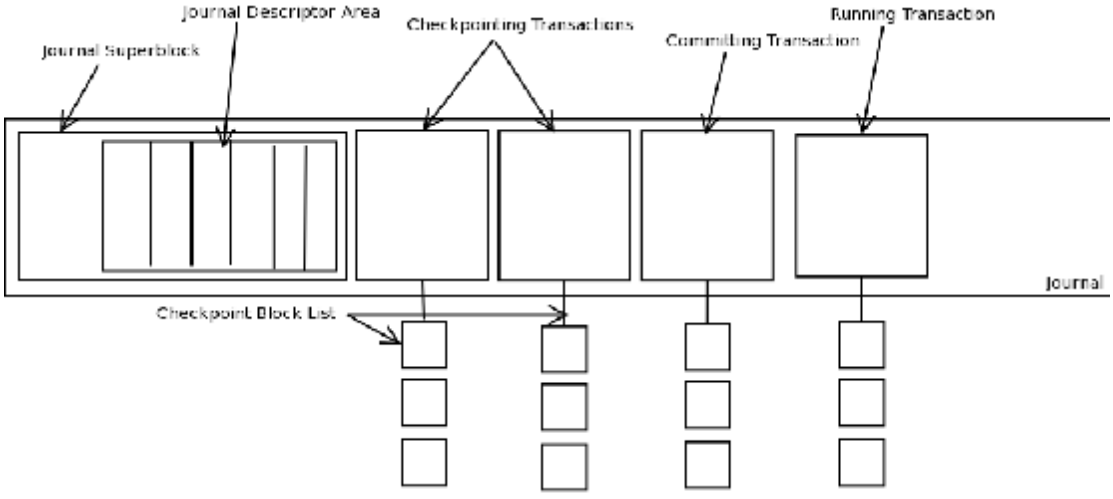
表格 4 取消块示意图

表格 4 表示一个取消块，其中 transaction ID = h_sequence = 67，然后依次是若干个 磁盘块号。这里表示，如果恢复时遇到要恢复的目标块（即磁盘原始位置）为 18976,2323,2324545,343,等块时，如果当前的 transaction ID <= 67，则不必恢复了（因为 transaction 67 中已经将这些块删除了）。

5．日志布局

从内存中看，一个日志由日志超级块和一系列事务组成。参见图表 4 内存中日志结构示意图

。



图表 4 内存中日志结构示意图

从物理角度来看，日志在磁盘上就是由一些磁盘块组成的，这些磁盘块从日志的角度分析看，可能是超级块、描述符块、数据块、提交块、取消块等。参见表格 5 日志块布局示意图

。

超级块
描述符块 1
数据块 1
数据块 2
……
数据块 n
提交块 1
取消块 1（可以没有）
描述符块 2
数据块 1
数据块 2
……
数据块 n
提交块 2
……

表格 5 日志块布局示意图

其中红色背景的块表示日志的描述块，白色背景的块表示日志的数据块。

七、 三种日志模式

日志机制的基本原理就是选择与文件系统一致性相关的缓冲区，在适当的时机“偷偷地”写入日志。但是选择什么样的缓冲区，以及在什么时间写入日志，就是所谓的日志模式。日志模式与系统性能息息相关，是个值得仔细研究、平衡的大问题。

ext3 支持三种日志模式，划分的依据是选择元数据块还是数据块写入日志，以及何时写入日志。

1. 日志（journal ）

文件系统所有数据块和元数据块的改变都记入日志。这种模式减少了丢失每个文件所作修改的机会，但是它需要很多额外的磁盘访问。例如，当一个新文件被创建时，它的所有数据块都必须复制一份作为日志记录。这是最安全和最慢的 ext3 日志模式。

2. 预定（ordered ）

只对文件系统元数据块的改变才记入日志，这样可以确保文件系统的一致性，但是不能保证文件内容的一致性。然而，ext3 文件系统把元数据块和相关的数据块进行分组，以便在元数据块写入日志之前写入数据块。这样，就可以减少文件内数据损坏的机会；例如，确保增大文件的任何写访问都完全受日志的保护。这是缺省的 ext3 日志模式。

3. 写回（writeback ）

只有对文件系统元数据的改变才记入日志，不对数据块进行任何特殊处理。这是在其他日志文件系统发现的方法，也是最快的模式。

在挂载 ext3 文件系统时，可通过 data=journal 等修改日志模式。如果不做特殊说明，下文中将默认介绍预定（ordered）模式，这是 ext3 默认的日志模式，也是实现最复杂的一种模式。

八、 jbd 基本操作

日志机制的核心是处理磁盘块缓冲区。它是在不影响系统原有的处理缓冲区的逻辑的基础上进行了。

日志的基本操作包括对原子操作的操作，对缓冲区的操作，对日志空间的操作，对日志提交的操作等等。

1. journal_start

journal_start 的主要作用是取得一个原子操作描述符 handle_t，如果当前进程已经有一个，则直接返回；否则，需要新创建一个。

同样，该 handle_t 也必须与一个正在运行的 transaction 相关联，如果没有正在运行的 transaction，则创建一个新的 transaction，并将其设置为正在运行的。

参数 nblocks 是向日志申请 nblocks 个缓冲区的空间，也表明该原子操作预期将修改 nblocks 个缓冲区。

```
271 handle_t *journal_start(journal_t *journal, int nblocks)
272 {
273         handle_t *handle = journal_current_handle();
274         int err;
279         if (handle) {
                    // 如果当前进程已经有 handle_t，则直接返回。
280                 J_ASSERT(handle->h_transaction->t_journal == journal);
281                 handle->h_ref++;
282                 return handle;
283         }
284
            // 否则，创建一个新的 handle_t
285         handle = new_handle(nblocks);
289         current->journal_info = handle;
// start_this_handle 的主要作用是将该 handle 与当前正在运行的 transaction 相关联，
// 如果没有正在运行的 transaction，则创建一个新的，并将其设置为正在运行的。
291         err = start_this_handle(journal, handle);
299         return handle;
300 }
```

2. journal_stop

该函数的主要作用是将该 handle 与 transaction 断开链接，调整所属 transaction 的额度。如果该原子操作是同步的，则设置事务的 t_synchronous_commit 标志。在事务提交时，会根据该标志决定缓冲区的写的方式。

```
1363 int journal_stop(handle_t *handle)
1364 {
1365         transaction_t *transaction = handle->h_transaction;
1366         journal_t *journal = transaction->t_journal;
```

```
// 如果该原子操作时同步的，则设置相应 transaction 的 t_synchronous_commit 标志，
// 则在提交事务时同步写出缓冲区。
1436        if (handle->h_sync)
1437                transaction->t_synchronous_commit = 1;

1438        current->journal_info = NULL;
1441        transaction->t_outstanding_credits -= handle->h_buffer_credits;
1442        transaction->t_updates--;
1455        if (handle->h_sync ||
1456                    transaction->t_outstanding_credits >
1457                            journal->j_max_transaction_buffers ||
1458                    time_after_eq(jiffies, transaction->t_expires)) {
1462                tid_t tid = transaction->t_tid;

                    // 这里的提交，只是设置 journal 中的 j_commit_request，
                    // 并唤醒等待的内核线程。
1468                __log_start_commit(journal, transaction->t_tid);

                    // 如果该原子操作时是同步的，则我们等待事务提交完成
1475                if (handle->h_sync && !(current->flags & PF_MEMALLOC))
1476                        err = log_wait_commit(journal, tid);
1480        }
1484        jbd_free_handle(handle);
1485        return err;
1486 }
```

journal_stop()函数调用了__log_start_commit()函数，该函数的主要作用是设置 journal 的 j_commit_request 域，标明申请进行提交的事务 ID 的最大值，并唤醒等待在 j_wait_commit 等待队列上的内核线程 kjournald。

```
435 int __log_start_commit(journal_t *journal, tid_t target)
436 {
440         if (!tid_geq(journal->j_commit_request, target)) {
446                 journal->j_commit_request = target;
450                 wake_up(&journal->j_wait_commit);
451                 return 1;
452         }
453         return 0;
454 }
```

但是 log_wait_commit()函数，就不这么简单了，它需要一直等待 ID 为 tid 的 transaction 结束。

```
535 int log_wait_commit(journal_t *journal, tid_t tid)
536 {
549         while (tid_gt(tid, journal->j_commit_sequence)) {
                    // journal->j_commit_sequence 保存的是上一次已提交的 transaction 的 ID
                    // 循环等待 ID 为 tid 的 transaction 结束。
```

```
552             wake_up(&journal->j_wait_commit);
553             spin_unlock(&journal->j_state_lock);
554             wait_event(journal->j_wait_done_commit,
555                     !tid_gt(tid, journal->j_commit_sequence));
556             spin_lock(&journal->j_state_lock);
557         }
565 }
```

3. journal_get_create_access

取得通过 journal_start()获得原子操作描述符后，在修改缓冲区前，我们应该在 jbd 中先获得该缓冲区的写权限。journal_get_create_access()、journal_get_write_access()和 journal_get_undo_access()这三个函数的作用就是在 jbd 中取得该缓冲区的写权限。注意，这三个函数都是将缓冲区加入 transaction 的 BJ_Reserved 队列上，表示这些缓冲区已被该 transaction 管理，但是并未修改。

元数据缓冲区的内容可能来自磁盘块，也可能是先在内存中产生数据，然后写回到磁盘块上的。典型的是文件的索引块。假设我们在扩展文件，则有可能新分配索引块。那么，对这种新创建的元数据块，必须通过 journal_get_create_access()函数取得该缓冲区的写权限。

```
781 int journal_get_create_access(handle_t *handle, struct buffer_head *bh)
782 {
783         transaction_t *transaction = handle->h_transaction;
784         journal_t *journal = transaction->t_journal;

                // 将一个新的 jh 与 bh 相关联，即将 bh 纳入 jbd 管理。
785         struct journal_head *jh = journal_add_journal_head(bh);

802         jbd_lock_bh_state(bh);
803         spin_lock(&journal->j_list_lock);
812         if (jh->b_transaction == NULL) {
                    // 如果该 jh 以前没有受 jbd 管理
821             clear_buffer_dirty(jh2bh(jh));
822             jh->b_transaction = transaction;
825             jh->b_modified = 0;
                    // 将 jh 加入 transaction 的 BJ_Reserved 队列，供 jbd 管理使用
828             __journal_file_buffer(jh, transaction, BJ_Reserved);
829         } else if (jh->b_transaction == journal->j_committing_transaction) {
                    // 如果该 jh 由已经由正在提交的事务管理，
                    // 则将 jh->b_next_transaction 设置为 handle 的 transaction，
                    // 意思是 jh 由当前所属的 transaction 处理完之后，
                    // 还要由 handle 的 transnation 继续处理。
831             jh->b_modified = 0;
834             jh->b_next_transaction = transaction;
835         }
836         spin_unlock(&journal->j_list_lock);
837         jbd_unlock_bh_state(bh);
```

```
                // 取消“取消”操作。
                // 这个有点奇怪吧。第四章第 8 节提到过 revoke 的概念。
                // 现在很明显该 jh 是文件系统需要的了，不应该再被 revoke。
847             journal_cancel_revoke(handle, jh);
848             journal_put_journal_head(jh);
849 out:
850             return err;
851 }
```

journal_add_journal_head()函数的作用是将缓冲区 bh 纳入 jbd 管理，即给每个 bh 关联一个 journal_head，一般简称 jh，然后用 jh 指向 bh。这样，事务中就可以通过管理 jh，来管理对应的缓冲区。

```
1802 struct journal_head *journal_add_journal_head(struct buffer_head *bh)
1803 {
1804             struct journal_head *jh;
1805             struct journal_head *new_jh = NULL;
1806
1807 repeat:
1808             if (!buffer_jbd(bh)) {
1809                     new_jh = journal_alloc_journal_head();
1810                     memset(new_jh, 0, sizeof(*new_jh));
1811             }
1812
1813             jbd_lock_bh_journal_head(bh);
1814             if (buffer_jbd(bh)) {
1815                     jh = bh2jh(bh);
1816             } else {
1821                     if (!new_jh) {
1822                             jbd_unlock_bh_journal_head(bh);
1823                             goto repeat;
1824                     }
1825
1826                     jh = new_jh;
1827                     new_jh = NULL;          /* We consumed it */
1828                     set_buffer_jbd(bh);
1829                     bh->b_private = jh;

                        // 本函数的主要作用在这里，将 jh 指向一个 bh。
1830                     jh->b_bh = bh;
1831                     get_bh(bh);
1833             }
1834             jh->b_jcount++;
1835             jbd_unlock_bh_journal_head(bh);
1836             if (new_jh)
```

```
1837                journal_free_journal_head(new_jh);
1838        return bh->b_private;
1839 }
```

4. journal_get_write_access

journal_get_write_access()函数的作用是使 jbd 取得对 bh 的写权限。

```
748 int journal_get_write_access(handle_t *handle, struct buffer_head *bh)
749 {
            // 将一个新的 jh 与 bh 相关联，即将 bh 纳入 jbd 管理。
750         struct journal_head *jh = journal_add_journal_head(bh);
751         int rc;

            // 将 jh 加入 transaction 的 BJ_Reserved 队列，供管理使用
756         rc = do_get_write_access(handle, jh, 0);

757         journal_put_journal_head(jh);
758         return rc;
759 }
```

do_get_write_access()函数的主要作用是将 jh 加入 transaction 的相关队列中。

```
514 static int
515 do_get_write_access(handle_t *handle, struct journal_head *jh,
516                     int force_copy)
517 {
518         struct buffer_head *bh;
519         transaction_t *transaction;
520         journal_t *journal;
521         int error;
522         char *frozen_buffer = NULL;
523         int need_copy = 0;
528         transaction = handle->h_transaction;
529         journal = transaction->t_journal;

534 repeat:
535         bh = jh2bh(jh);
539         lock_buffer(bh);
540         jbd_lock_bh_state(bh);
555         if (buffer_dirty(bh)) {
                // 从数据是否与磁盘一致的角度看，
                // 缓冲区可分为 dirty 与 update 两种状态。
                // 纳入 jbd 管理后，缓冲区的脏状态将由 jbd 管理。
560             if (jh->b_transaction) {
568                 warn_dirty_buffer(bh);
569             }
576             clear_buffer_dirty(bh);
```

```
577             set_buffer_jbddirty(bh);
578         }
579
580         unlock_buffer(bh);
593         if (jh->b_transaction == transaction ||
594             jh->b_next_transaction == transaction)
                // 如果该 jh 已经由当前 transaction 管理了，
                // 则我们不需要做什么了，直接退出即可。
595             goto done;

601         jh->b_modified = 0;
607         if (jh->b_frozen_data) {
                // 否则，该 jh 尚未由当前的 transaction 管理，
                // 这样，该 jh 已经被旧的某个 transaction 管理了，
                // 设置 b_next_transaction 的值，
                // 表示旧的 transaction 处理完本 jh 之后，
                // 本 transaction 会继续处理该 jh。
610             jh->b_next_transaction = transaction;
                // 因为 b_frozen_data 已经存在了，则不需要再拷贝了。
611             goto done;
612         }
615
616         if (jh->b_transaction && jh->b_transaction != transaction) {
                // 该缓冲区属于旧的 transaction
631             if (jh->b_jlist == BJ_Shadow) {
                    // 如果该 jh 在旧的 transaction 的 BJ_Shadow 队列上，
                    // 则表示该 jh 正在被写入到日志中，
                    // 我们要等待写操作的完成。
632                 DEFINE_WAIT_BIT(wait, &bh->b_state, BH_Unshadow);
633                 wait_queue_head_t *wqh;
635                 wqh = bit_waitqueue(&bh->b_state, BH_Unshadow);
638                 jbd_unlock_bh_state(bh);
640                 for ( ; ; ) {
641                     prepare_to_wait(wqh, &wait.wait,
642                                 TASK_UNINTERRUPTIBLE);
643                     if (jh->b_jlist != BJ_Shadow)
644                         break;
645                     schedule();
646                 }
647                 finish_wait(wqh, &wait.wait);
648                 goto repeat;
649             }
650
665             if (jh->b_jlist != BJ_Forget || force_copy) {
```

```
                    // 该缓冲区属于旧的 transaction，并且不在 BJ_Forget 队列上，
                    // 或者 force_copy == 1
                    // 则我们需要将缓冲区当前的数据冻结起来，备份起来，
                    // 供旧的 transaction 使用。
667                 if (!frozen_buffer) {
669                     jbd_unlock_bh_state(bh);
670                     frozen_buffer =
671                         jbd_alloc(jh2bh(jh)->b_size,
672                                   GFP_NOFS);
673                     if (!frozen_buffer) {
678                         error = -ENOMEM;
679                         jbd_lock_bh_state(bh);
680                         goto done;
681                     }
682                     goto repeat;
683                 }
684                 jh->b_frozen_data = frozen_buffer;
685                 frozen_buffer = NULL;
                    // 必须要拷贝，不能含糊
686                 need_copy = 1;
687             }
                // 旧的 transaction 处理完本 jh 之后，
                // 本 transaction 会继续处理该 jh。
688             jh->b_next_transaction = transaction;
689         }
691
697         if (!jh->b_transaction) {
                // 最后，如果该缓冲区目前没有被 journald，
                // 我们需要确保在调用者写回磁盘之前 journal 该缓冲区。
700             jh->b_transaction = transaction;
702             spin_lock(&journal->j_list_lock);
703             __journal_file_buffer(jh, transaction, BJ_Reserved);
704             spin_unlock(&journal->j_list_lock);
705         }
706
707 done:
708         if (need_copy) {
                // 如果需要将缓冲区目前的数据冻结起来，
                // 就复制一份，放到 jh->b_frozen_data 中。
709             struct page *page;
710             int offset;
711             char *source;
715             page = jh2bh(jh)->b_page;
716             offset = ((unsigned long) jh2bh(jh)->b_data) & ~PAGE_MASK;
```

```
717             source = kmap_atomic(page, KM_USER0);
718             memcpy(jh->b_frozen_data, source+offset, jh2bh(jh)->b_size);
719             kunmap_atomic(source, KM_USER0);
720         }
721         jbd_unlock_bh_state(bh);
            // 同理，取消“取消”操作。
727         journal_cancel_revoke(handle, jh);
734         return error;
735 }
```

5. journal_get_undo_access

这个函数是处理一种特殊的元数据块的——磁盘块位图。

磁盘块位图是文件系统用于记录磁盘块使用情况的一种结构，块中的每一个位表示相应的磁盘块是否被占用了。如果空闲，则为 0，否则为 1。磁盘块位图之所以特殊，在于一个磁盘块不能被两个文件同时占用，它要么是空闲的，要么在同一时刻只能被一个文件占用。对这种元数据块的修改，要取得 undo 权限。为什么呢？

假设 handle1 中，删除了一个数据块 b1，则对应 bitmap1 中的位被清掉，这个操作属于 transaction1。此时，再进行磁盘块的分配和释放，则我们必须要知道 bitmap1 是否已被提交到日志中了。因为，如果 bitmap1 已经被提交到日志中了，则表示 handle1 已经确实完成了，即使现在发生崩溃，删除 b1 的操作也是可以重现的。但是如果 bitmap1 没有被提交到日志中，则表示 handle1 并没有完成，那么，你说此时数据块 b1 是已被删除了还是没有被删除呢？其实从物理的角度来看，b1 并没有被删除，因为实际上磁盘块位图并没有被改变。

此时，如果要重新分配磁盘块 b1，我们必须等待，直到 t1 提交完成，以保证 handle1 的可恢复性。

因此，我们从磁盘块位图中分配磁盘块时，只可以分配在缓冲区中和日志中该位都为 0 的磁盘块。为此，jbd 在取得磁盘块位图缓冲区的写权限时，必须将缓冲区当前的内容拷贝一份，以备分配磁盘块时使用。

journal_get_undo_access()与 journal_get_write_access()函数基本类似，但是注意在调用 do_get_write_access()函数时最后一个参数是 1，表示 force_copy 为真，表示一定要将缓冲区当前的数据冻结起来。

```
878 int journal_get_undo_access(handle_t *handle, struct buffer_head *bh)
879 {
880         int err;
881         struct journal_head *jh = journal_add_journal_head(bh);
882         char *committed_data = NULL;
883
891         err = do_get_write_access(handle, jh, 1);
            // 将缓冲区当前的数据拷贝一份，放到 jh->b_committed_data 中，
            // 作为将来分配磁盘块的依据。
895 repeat:
896         if (!jh->b_committed_data) {
897             committed_data = jbd_alloc(jh2bh(jh)->b_size, GFP_NOFS);
898             if (!committed_data) {
901                 err = -ENOMEM;
```

```
902                     goto out;
903             }
904     }
905
906     jbd_lock_bh_state(bh);
907     if (!jh->b_committed_data) {
911             if (!committed_data) {
912                     jbd_unlock_bh_state(bh);
913                     goto repeat;
914             }
915
916             jh->b_committed_data = committed_data;
917             committed_data = NULL;
918             memcpy(jh->b_committed_data, bh->b_data, bh->b_size);
919     }
920     jbd_unlock_bh_state(bh);
921 out:
922     journal_put_journal_head(jh);
925     return err;
926 }
```

我们接着看看 ext3 中是如何判断一个磁盘块是否可以被分配的。

fs/ext3/balloc.c。

判断一个磁盘块是否可以被分配的逻辑是这样的：

1) 先看内存中磁盘块位图中该块对应位的值，如果为 1，表示不可分配；
   （注意，此时虽然内存中该位为 1，可能磁盘上该位为 0，但是，没说的，无论如何不分配这样的块。）
2) 如果内存中磁盘块位图中该块对应位的值为 0，再检查 b_committed_data 中该块对应位的值。如果为 0，表示可以分配；如果为 1，表示不可分配！

ext3_test_allocatable()函数返回值：

返回 0：第 nr 位已经被置为 1，表示不可分配。

返回 1：第 nr 位没有被置为 1，表示可以分配。

```
704 static int ext3_test_allocatable(ext3_grpblk_t nr, struct buffer_head *bh)
705 {
706     int ret;
707     struct journal_head *jh = bh2jh(bh);
708
709     if (ext3_test_bit(nr, bh->b_data))
710             return 0;
711
712     jbd_lock_bh_state(bh);
713     if (!jh->b_committed_data)
714             ret = 1;
715     else
716             ret = !ext3_test_bit(nr, jh->b_committed_data);
```

```
717         jbd_unlock_bh_state(bh);
718         return ret;
719 }
```

6. journal_dirty_data

jbd 在取得缓冲区的写权限之后，文件系统就可以修改改缓冲区的内容了。修改完毕，文件系统需要调用一个函数，来通知 jbd 该缓冲区的修改已经完成。journal_dirty_data()和 journal_dirty_metadata()函数就是作通知作用的。

journal_dirty_data()的作用是通知 jbd 一个数据块缓冲区的修改已经完成。这样在 ordered 模式中，可以确保在提交 transaction 之前，将相关联的数据块缓冲区都写回磁盘原始位置。

```
945 int journal_dirty_data(handle_t *handle, struct buffer_head *bh)
946 {
947         journal_t *journal = handle->h_transaction->t_journal;
949         struct journal_head *jh;
            // 将一个新的 jh 与 bh 相关联，即将 bh 纳入 jbd 管理。
955         jh = journal_add_journal_head(bh);
985         jbd_lock_bh_state(bh);
986         spin_lock(&journal->j_list_lock);
987
994         if (jh->b_transaction) {
                // 该 jh 已经由某个 transaction 管理了
996             if (jh->b_transaction != handle->h_transaction) {
                    // 如果该 jh 是由旧的 transaction 管理的
1034                if (jh->b_jlist != BJ_None &&
1035                        jh->b_jlist != BJ_SyncData &&
1036                        jh->b_jlist != BJ_Locked) {
                        // 一个数据块缓冲区，在 jbd 控制下正常写回磁盘时，
                        // 会依次经历 BJ_SyncData、BJ_Locked、BJ_None 三种状态。
                        // 如果该 jh 不在上述三个队列上，则表示 jbd 没有管理该 jh。
1038                    goto no_journal;
1039                }
1040
1047                if (buffer_dirty(bh)) {
                        // 如果该缓冲区已经是脏的了，则我们需要同步地写回磁盘。
1048                    get_bh(bh);
1049                    spin_unlock(&journal->j_list_lock);
1050                    jbd_unlock_bh_state(bh);
1051                    need_brelse = 1;
1052                    sync_dirty_buffer(bh);
1053                    jbd_lock_bh_state(bh);
1054                    spin_lock(&journal->j_list_lock);
1062                }
1074                if (jh->b_transaction != NULL) {
                    // 此时，经过 1047-1062 行的代码，该缓冲区已经是 update 的了。
```

```
                    // 故可以脱离原来的 transaction，由 handle->h_transaction 进行管理了。
1076                    __journal_temp_unlink_buffer(jh);
1081                    jh->b_transaction = handle->h_transaction;
1082                }
1085            }
1092            if (jh->b_jlist != BJ_SyncData && jh->b_jlist != BJ_Locked) {
                    // 数据块缓冲区只能有三种状态 BJ_SyncData、BJ_Locked、BJ_None，
                    // 这里只能是 BJ_None 了，
                    // 而 journal_dirty_data()需要将缓冲区链入 BJ_SyncData 队列。
1095                __journal_temp_unlink_buffer(jh);
1096                jh->b_transaction = handle->h_transaction;
1098                __journal_file_buffer(jh, handle->h_transaction,
1099                                BJ_SyncData);
1100            }
1101        } else {
                // 这个 else 对应的是 994 行的 if
1103            __journal_file_buffer(jh, handle->h_transaction, BJ_SyncData);
1104        }
1105 no_journal:
1106        spin_unlock(&journal->j_list_lock);
1107        jbd_unlock_bh_state(bh);
1108        if (need_brelse) {
1110            __brelse(bh);
1111        }
1113        journal_put_journal_head(jh);
1114        return ret;
1115 }
```

7. journal_dirty_metadata

journal_dirty_metadata()的作用是通知 jbd 一个元数据块缓冲区的修改已经完成。

```
1136 int journal_dirty_metadata(handle_t *handle, struct buffer_head *bh)
1137 {
1138        transaction_t *transaction = handle->h_transaction;
1139        journal_t *journal = transaction->t_journal;
1140        struct journal_head *jh = bh2jh(bh);
1147        jbd_lock_bh_state(bh);
1159
1167        if (jh->b_transaction == transaction && jh->b_jlist == BJ_Metadata) {
1169            J_ASSERT_JH(jh, jh->b_transaction ==
1170                                journal->j_running_transaction);
                // 如果该 jh 已经由当前正在运行的 transaction 管理了，
                // 则本函数不需要做什么了。
1171            goto out_unlock_bh;
1172        }
```

```
1173
1174        set_buffer_jbddirty(bh);
1175
1182        if (jh->b_transaction != transaction) {
                // 该 jh 在正在提交的 transaction 上
1184            J_ASSERT_JH(jh, jh->b_transaction ==
1185                                journal->j_committing_transaction);
1186            J_ASSERT_JH(jh, jh->b_next_transaction == transaction);
                // 从这两个断言来看，jh->b_next_transaction 已经设置好了，
                // 则正在提交的 transaction 完成后，本 transaction 会继续处理的。
1189            goto out_unlock_bh;
1190        }
1196        spin_lock(&journal->j_list_lock);
            // 将 jh 链入 BJ_Metadata 队列。
            // 注意，本缓冲区以前可能通过 journal_get_XXX_access()加入了 BJ_Reserved 队列，这里要从原队列上移除，然后加入 BJ_Metadata 队列。
1197        __journal_file_buffer(jh, handle->h_transaction, BJ_Metadata);
1198        spin_unlock(&journal->j_list_lock);
1199 out_unlock_bh:
1200        jbd_unlock_bh_state(bh);
1201 out:
1202        JBUFFER_TRACE(jh, "exit");
1203        return 0;
1204 }
```

8. journal_forget

以索引块缓冲区为例。如果一个原子操作在运行过程中，要分配一个新的索引块，则它要先调用 journal_get_write_access()函数取得写权限，然后修改这个索引块缓冲区，然后调用 journal_dirty_metadata()将该缓冲区设为脏。但是，如果不幸，该原子操作后边运行出错了，需要将之前的修改全部取消，则需要调用 journal_forget()函数使 jbd“忘记”该缓冲区。

```
1234 int journal_forget (handle_t *handle, struct buffer_head *bh)
1235 {
1236        transaction_t *transaction = handle->h_transaction;
1237        journal_t *journal = transaction->t_journal;
1238        struct journal_head *jh;
1239        int drop_reserve = 0;
1240        int err = 0;
1241        int was_modified = 0;
1242
1245        jbd_lock_bh_state(bh);
1246        spin_lock(&journal->j_list_lock);
1250        jh = bh2jh(bh);
1261        was_modified = jh->b_modified;
1267        jh->b_modified = 0;
```

```
1268
1269        if (jh->b_transaction == handle->h_transaction) {
                // 如果该缓冲区属于该 handle 所属的 transaction
                // 立即从本 transaction 中删除即可。
1275            clear_buffer_dirty(bh);
1276            clear_buffer_jbddirty(bh);
1284            if (was_modified)
1285                drop_reserve = 1;
1299            if (jh->b_cp_transaction) {
                    // 如果该缓冲区在 transaction 的 checkpoint 队列上，
                    // 则重新将缓冲区加入 BJ_Forget 队列，
                    // 这样，本 transaction 提交时，
                    // 才会根据该 bh 是否为脏将其链入本 transaction 的 checkpoint 队列上。
1300                __journal_temp_unlink_buffer(jh);
1301                __journal_file_buffer(jh, transaction, BJ_Forget);
1302            } else {
                    // 如果该缓冲区不在 transaction 的 checkpoint 队列上，
                    // 则直接删除
1303                __journal_unfile_buffer(jh);
1304                journal_remove_journal_head(bh);
1312            }
1313        } else if (jh->b_transaction) {
                // 如果该缓冲区属于以前的 transaction，
                // 则我们不能丢掉它
1314            J_ASSERT_JH(jh, (jh->b_transaction ==
1315                        journal->j_committing_transaction));
1322            if (jh->b_next_transaction) {
1323                J_ASSERT(jh->b_next_transaction == transaction);
                    // jh->b_next_transaction 设置为 NULL，
                    // 表示旧的 transaction 处理完该 jh 之后，
                    // 没有 transaction 要处理它了。
1324                jh->b_next_transaction = NULL;
1330                if (was_modified)
1331                    drop_reserve = 1;
1332            }
1333        }
1339 drop:
1340        if (drop_reserve) {
                // 既然要使 jbd“忘记”该缓冲区，
                // 那么该缓冲区在 handle 中占用的额度可以回收了
1342            handle->h_buffer_credits++;
1343        }
1344        return err;
1345 }
```

9. journal_revoke

revoke 的原理第四章第 8 节已经提到过了，这里看看如何实现。

jbd 在内存中设置了两个 hash 表用于管理被 revoked 的缓冲区，

```
struct journal_s
{
    ……
    spinlock_t          j_revoke_lock;      // 保护 revoke 哈希表的自旋锁
    struct jbd_revoke_table_s *j_revoke;    // 指向 journal 正在使用的 revoke hash table
    struct jbd_revoke_table_s *j_revoke_table[2]; // 指向两个 revoke hash table
}
```

jbd 初始化时，要为创建两个 hash 表。

```
272 /* Initialise the revoke table for a given journal to a given size. */
273 int journal_init_revoke(journal_t *journal, int hash_size)
274 {
        // 创建第一个 hash 表
278     journal->j_revoke_table[0] = journal_init_revoke_table(hash_size);
279     if (!journal->j_revoke_table[0])
280         goto fail0;
281
        // 创建第二个 hash 表
282     journal->j_revoke_table[1] = journal_init_revoke_table(hash_size);
283     if (!journal->j_revoke_table[1])
284         goto fail1;
285
        // 将 journal 当前使用的 revoke hash 表设置为第二个。
286     journal->j_revoke = journal->j_revoke_table[1];
287
288     spin_lock_init(&journal->j_revoke_lock);
289
290     return 0;
291
292 fail1:
293     journal_destroy_revoke_table(journal->j_revoke_table[0]);
294 fail0:
295     return -ENOMEM;
296 }
```

journal_init_revoke_table()函数会创建给定大小的一个 hash 表，每个 hash 表在内存中都由一个 jbd_revoke_table_s 结构表示。

```
/* The revoke table is just a simple hash table of revoke records. */
struct jbd_revoke_table_s
{
```

```
    int             hash_size;
    int             hash_shift;
    struct list_head *hash_table;
};
```

其中 hash_size 表示该 hash 表的大小，而 2hash_shift = hash_size。而 hash_table 则指向一个 list_head 结构数组，即 hash 表。从逻辑上看，这个 hash 表中保存的数据是 jbd_revoke_record_s 结构，但实际上，只是 hash 表中保存的数据是 jbd_revoke_record_s. hash。

```
struct jbd_revoke_record_s
{
    struct list_head    hash;
    tid_t               sequence; /* Used for recovery only */
    unsigned int        blocknr;
};
```

其中 hash 是用于将本结构链入 revoke hash 表的，sequence 是调用 revoke 的 transaction 的 ID，blocknr 是磁盘块号。

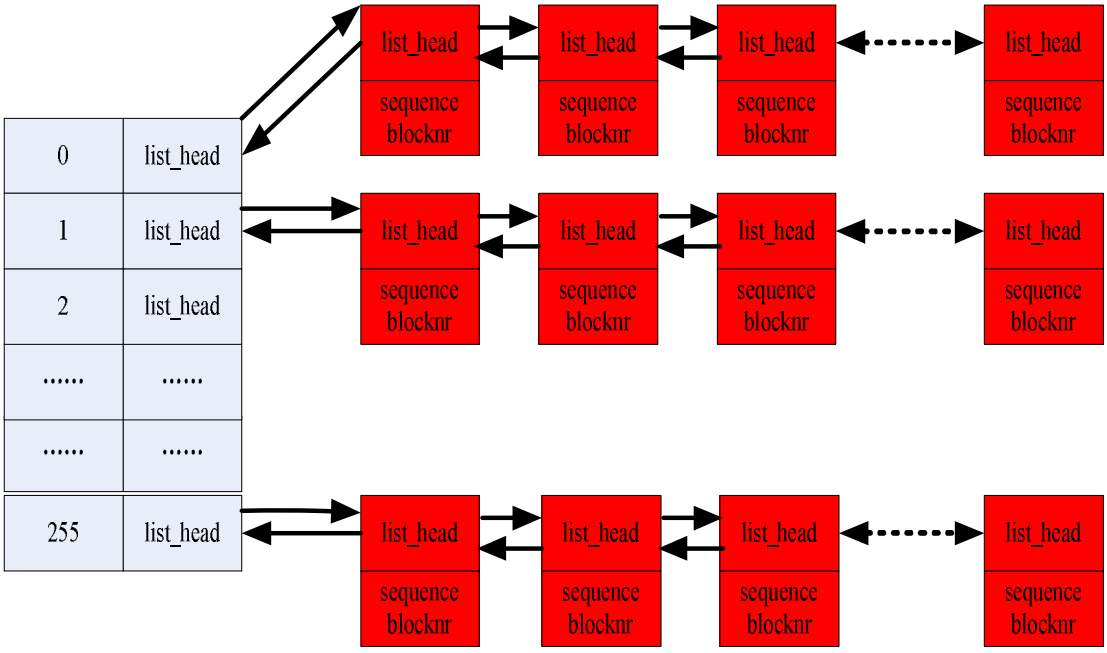
```
228 static struct jbd_revoke_table_s *journal_init_revoke_table(int hash_size)
229 {
230     int shift = 0;
231     int tmp = hash_size;
232     struct jbd_revoke_table_s *table;
233
234     table = kmem_cache_alloc(revoke_table_cache, GFP_KERNEL);
235     if (!table)
236         goto out;
237
        // 计算 hash_shift
238     while((tmp >>= 1UL) != 0UL)
239         shift++;
240
241     table->hash_size = hash_size;
242     table->hash_shift = shift;
        // 创建 hash 表
243     table->hash_table =
244         kmalloc(hash_size * sizeof(struct list_head), GFP_KERNEL);
245     if (!table->hash_table) {
246         kmem_cache_free(revoke_table_cache, table);
247         table = NULL;
248         goto out;
249     }
250
        // 初始化 hash 表
251     for (tmp = 0; tmp < hash_size; tmp++)
252         INIT_LIST_HEAD(&table->hash_table[tmp]);
253
```

```
254 out:
255         return table;
256 }
```

在 include/linux/jbd.h 中，

962 #define JOURNAL_REVOKE_DEFAULT_HASH 256

所以每个 hash 表都有 256 项，每项都是一个用 struct list_head 链起来的双向链表。



图表 5 revoke hash table 示意图

其中，每个红色部分都是一个 jbd_revoke_record_s 结构。

```
335 int journal_revoke(handle_t *handle, unsigned int blocknr,
336                     struct buffer_head *bh_in)
337 {
338         struct buffer_head *bh = NULL;
339         journal_t *journal;
340         struct block_device *bdev;
346
347         journal = handle->h_transaction->t_journal;
352
353         bdev = journal->j_fs_dev;
354         bh = bh_in;
355
356         if (!bh) {
                // 如果 bh==NULL，则根据 blocknr 查找对应的缓冲区
357             bh = __find_get_block(bdev, blocknr, journal->j_blocksize);
360         }
386         if (bh) {
```

```
387             if (!J_EXPECT_BH(bh, !buffer_revoked(bh),
388                              "inconsistent data on disk")) {
389                     if (!bh_in)
390                             brelse(bh);
391                     return -EIO;
392             }
393             set_buffer_revoked(bh);
394             set_buffer_revokevalid(bh);
395             if (bh_in) {
397                     journal_forget(handle, bh_in);
398             } else {
400                     __brelse(bh);
401             }
402     }
403     // 将<blocknr, tid>加入 hash 表。
405     err = insert_revoke_hash(journal, blocknr,
406                              handle->h_transaction->t_tid);
408     return err;
409 }
```

为什么 jbd 要设置两个 hash 表呢？这要从提交 revoke 记录说起。当一个正在运行的 transaction 要提交时，与之相对应的 revoke hash 表也要提交。要提交 revoke hash 表，必须把其中的数据冻结起来，不再被改动。此时，为了能使 jbd 能够继续接收 revoke 记录，则需为 journal 设置另一个 hash 表。所以，jbd 设置了两个 hash 表，供 journal 交替使用。

fs/jbd/revoke.c 中还包含以下若干与 revoke 机制相关的函数，都比较简单，读者可以自行阅读。

insert_revoke_hash
find_revoke_record
journal_destroy_revoke_caches
journal_init_revoke_caches
journal_init_revoke_table
journal_destroy_revoke_table
journal_init_revoke
journal_destroy_revoke
journal_cancel_revoke
journal_switch_revoke_table
journal_write_revoke_records
write_one_revoke_record
flush_descriptor
journal_set_revoke
journal_test_revoke
journal_clear_revoke

10. journal_extend

每个原子操作 handle 在创建时，会为之分配若干额度。但是如果后便的运行中发现还需要更多的额度，则可通过 journal_extend()函数增加某个 handle 的额度。

```
322 int journal_extend(handle_t *handle, int nblocks)
323 {
324         transaction_t *transaction = handle->h_transaction;
325         journal_t *journal = transaction->t_journal;
326         int result;
327         int wanted;
334
335         spin_lock(&journal->j_state_lock);
336
344         spin_lock(&transaction->t_handle_lock);
345         wanted = transaction->t_outstanding_credits + nblocks;
346
347         if (wanted > journal->j_max_transaction_buffers) {
// j_max_transaction_buffers 表示在一次 transnation 提交中允许提交的最多个缓冲区个数。
//  如果 wanted > journal->j_max_transaction_buffers，表示要求的太多了，
//  本 transaction 不能满足。
350                 goto unlock;
351         }
352
353         if (wanted > __log_space_left(journal)) {
                    //  如果请求的额度大于日志剩余的空间，则出错。
356                 goto unlock;
357         }
358
            //  给 handle 和相应的 transaction 增加额度。
359         handle->h_buffer_credits += nblocks;
360         transaction->t_outstanding_credits += nblocks;
361         result = 0;
362
364 unlock:
365         spin_unlock(&transaction->t_handle_lock);
366 error_out:
367         spin_unlock(&journal->j_state_lock);
368 out:
369         return result;
370 }
```

11. 元数据缓冲区处理流程

当文件系统要把一个元数据块纳入 jbd 管理时，处理流程需要 5 步：

1) 获取一个原子操作描述符
   调用 journal_start()函数。
2) 取得在 jbd 中的写权限
   这个针对不同的元数据缓冲区，调用的函数是不同的：
   新创建的：journal_get_create_access()、
   文件系统中已经存在的：journal_get_write_access()
   磁盘块位图：journal_get_undo_access()
3) 修改缓冲区的数据
   这里文件系统可以根据自己的需要随意修改了。
4) 将缓冲区设置为脏
   调用 journal_dirty_metadata()函数。
5) 关闭原子操作符
   调用 journal_stop()函数。

我们举一个 ext3 中的实际的例子来说明一下文件系统是如何处理元数据缓冲区的。注意，ext3 对 jbd 的各个函数都进行了一定的封装，如 journal_dirty_metadata()封装成了 ext3_journal_dirty_metadata()，这里不再特殊解释了。

以在一个目录中创建一个新文件为例吧。

ext3 中关于目录节点的操作由 ext3_dir_inode_operations 结构来描述。

fs/ext3/namei.c

```
2451 const struct inode_operations ext3_dir_inode_operations = {
2452        .create         = ext3_create,
2453        .lookup         = ext3_lookup,
2454        .link           = ext3_link,
2455        .unlink         = ext3_unlink,
2456        .symlink        = ext3_symlink,
2457        .mkdir          = ext3_mkdir,
2458        .rmdir          = ext3_rmdir,
2459        .mknod          = ext3_mknod,
2460        .rename         = ext3_rename,
2461        .setattr        = ext3_setattr,
2468        .check_acl      = ext3_check_acl,
2469 };
```

其中 ext3_create()是负责创建一个新文件的。所谓创建一个新文件，实现时主要做三件事情：

1) 在文件系统中创建一个代表该文件的 inode；
2) 在父目录中分配一个新的目录项，用于指向新创建的文件。
3) 将目录中的新目录项与 inode 相关联。

```
1692 static int ext3_create (struct inode * dir, struct dentry * dentry, int mode,
1693                 struct nameidata *nd)
1694 {
          // 参数说明：
          // dir 是父目录的 inode，
          // dentry 是新创建的目录项，但是尚未与一个磁盘上的 inode 相关联。
1695       handle_t *handle;
1696       struct inode * inode;
1697       int err, retries = 0;
1700
1701 retry:
          // 取得原子操作描述符,对应第 1 步
1702       handle = ext3_journal_start(dir, EXT3_DATA_TRANS_BLOCKS(dir->i_sb) +
1703                                   EXT3_INDEX_EXTRA_TRANS_BLOCKS + 3 +
1704                                   EXT3_MAXQUOTAS_INIT_BLOCKS(dir->i_sb));
          // 在目录 dir 所在的块组中分配一个新的 inode，
          // 即在块组的 inode 位图中寻找一个为 0 的位，然后标记为 1
1711      inode = ext3_new_inode (handle, dir, mode);
1712      err = PTR_ERR(inode);
1713      if (!IS_ERR(inode)) {
              // 新分配的 indoe 是一个普通文件，设置文件操作。
1714          inode->i_op = &ext3_file_inode_operations;
1715          inode->i_fop = &ext3_file_operations;
1716          ext3_set_aops(inode);
              // 将新分配的 inode 与目录项关联起来，这样通过目录就可查找到文件。
1717          err = ext3_add_nondir(handle, dentry, inode);
1718      }
          // 关闭原子操作描述符，对应第 5 步
1719      ext3_journal_stop(handle);
1722      return err;
1723 }
```

ext3_new_inode()函数比较复杂，这里只把与 jbd 相关的主干部分列出来。

```
419 struct inode *ext3_new_inode(handle_t *handle, struct inode * dir, int mode)
420 {
          ……
          // 先分配一个内存 inode
439       sb = dir->i_sb;
440       inode = new_inode(sb);
443       ei = EXT3_I(inode);
444
          // 445-454 行，找到父目录所在的块组
          ……

          // 尝试在块组的 inode 位图中分配一个未使用的磁盘 inode
```

```
459         for (i = 0; i < sbi->s_groups_count; i++) {
460                 err = -EIO;
461
                    // 获取块组描述符
462                 gdp = ext3_get_group_desc(sb, group, &bh2);
463                 if (!gdp)
464                         goto fail;

                    // 获取块组 inode 位图
467                 bitmap_bh = read_inode_bitmap(sb, group);
471                 ino = 0;
472
473 repeat_in_this_group:
                    // 在块组 inode 位图中查找第一个未使用的磁盘 inode。
474                 ino = ext3_find_next_zero_bit((unsigned long *)
475                                 bitmap_bh->b_data, EXT3_INODES_PER_GROUP(sb), ino);
476                 if (ino < EXT3_INODES_PER_GROUP(sb)) {
                        // 找到了，我们要修改块组磁盘块位图了，
                        // 先取得写权限，对应第 2 步
479                     err = ext3_journal_get_write_access(handle, bitmap_bh);

                        // 将块组磁盘块位图相应位置 1，表示分配该磁盘 inode。对应第 3 步
483                     if (!ext3_set_bit_atomic(sb_bgl_lock(sbi, group),
484                                     ino, bitmap_bh->b_data)) {

                            // 既然我们修改了元数据块缓冲区，则我们要把它置为脏。
                            // 对应第 4 步
488                         err = ext3_journal_dirty_metadata(handle,
489                                                 bitmap_bh);
492                         goto got;
493                     }
499                 }
510         }
514 got:
515         ino += group * EXT3_INODES_PER_GROUP(sb) + 1;

            // 下面要修改块组描述符了，这也是一个元数据块缓冲区，
            // 先取得写权限。
525         err = ext3_journal_get_write_access(handle, bh2);
526         if (err) goto fail;
527         spin_lock(sb_bgl_lock(sbi, group));
528         le16_add_cpu(&gdp->bg_free_inodes_count, -1);
529         if (S_ISDIR(mode)) {
530                 le16_add_cpu(&gdp->bg_used_dirs_count, 1);
```

```
531         }
532         spin_unlock(sb_bgl_lock(sbi, group));

            // 既然我们修改了元数据块缓冲区，则我们要把它置为脏。
534         err = ext3_journal_dirty_metadata(handle, bh2);

            // 下面是一些内存 inode 初始化的代码，我们简单略过
            ……

            // 既然我们修改了磁盘 inode（创建当然也是一种修改），
            // 我们将磁盘 inode 对应的内存缓冲区设置为脏，以便在适当时机写回磁盘。
            // 注意这里也将原子操作描述符 hanlde 传入了，因为磁盘 inode 也是一个元数据块，
            // 故仍然会进行上述 5 步操作中的 2-4 步。
            // 因为过程与本函数前面的修改磁盘块位图和块组描述符很类似，这里略过。
603         err = ext3_mark_inode_dirty(handle, inode);
631 }
```

12. 数据缓冲区处理流程

第七章介绍了 ext3 支持的三种日志模式。在 data=journal 模式中，将数据块也视作元数据块即可。而在 data=writeback 模式中，是不处理数据块的。所以这里我们只介绍在 data=ordered 模式中的数据块缓冲区的处理。

在 data=ordered 模式中，ext3 保证在元数据块写入日志之前写入数据块。这样，transaction 也需要对数据块缓冲区进行管理。对数据块缓冲区的管理比较简单，不需要在修改缓冲区之前先取得写权限，直接调用 journal_dirty_data()函数即可。

我们也是通过一个 ext3 中的例子，看看文件系统时如何处理数据块缓冲区的。

内核对文件内容进行读写时，是以页面为单位的，每个页面会对应若干个磁盘块缓冲区，也即对应若干磁盘块。ext3_ordered_write_end()函数是在内核修改完文件内容之后调用的，用以进行写操作的最后的处理。

fs/ext3/inode.c

```
1288 static int ext3_ordered_write_end(struct file *file,
1289                                    struct address_space *mapping,
1290                                    loff_t pos, unsigned len, unsigned copied,
1291                                    struct page *page, void *fsdata)
1292 {
            // 取得一个原子操作描述符
1293        handle_t *handle = ext3_journal_current_handle();
1294        struct inode *inode = file->f_mapping->host;
1295        unsigned from, to;
1296        int ret = 0, ret2;
1297
1298        copied = block_write_end(file, mapping, pos, len, copied, page, fsdata);

            // 对该页的每个页面，调用一次 journal_dirty_data_fn()函数。
1302        ret = walk_page_buffers(handle, page_buffers(page),
```

```
1303                from, to, NULL, journal_dirty_data_fn);
1304
            // 关闭原子操作描述符
1313        ret2 = ext3_journal_stop(handle);
1322 }
```

walk_page_buffers()函数也在 fs/ext3/inode.c。

```
1092 static int walk_page_buffers(      handle_t *handle,
1093                            struct buffer_head *head,
1094                            unsigned from,
1095                            unsigned to,
1096                            int *partial,
1097                            int (*fn)(    handle_t *handle,
1098                                        struct buffer_head *bh))
1099 {
1100        struct buffer_head *bh;
1101        unsigned block_start, block_end;
1102        unsigned blocksize = head->b_size;
1103        int err, ret = 0;
1104        struct buffer_head *next;
1105
1106        for (      bh = head, block_start = 0;
1107             ret == 0 && (bh != head || !block_start);
1108             block_start = block_end, bh = next)
1109        {
1110             next = bh->b_this_page;
1111             block_end = block_start + blocksize;
1112             if (block_end <= from || block_start >= to) {
1113                  if (partial && !buffer_uptodate(bh))
1114                       *partial = 1;
1115                  continue;
1116             }
1117             err = (*fn)(handle, bh);
1118             if (!ret)
1119                  ret = err;
1120        }
1121        return ret;
1122 }
```

journal_dirty_data_fn()函数也在 fs/ext3/inode.c。

```
1244 static int journal_dirty_data_fn(handle_t *handle, struct buffer_head *bh)
1245 {
1250        if (buffer_mapped(bh) && buffer_uptodate(bh))
1251             return ext3_journal_dirty_data(handle, bh);
1252        return 0;
1253 }
```

```
1234 int ext3_journal_dirty_data(handle_t *handle, struct buffer_head *bh)
1235 {
1236         int err = journal_dirty_data(handle, bh);
1240         return err;
1241 }
```

九、 等待提交事务 kjournald——我们时刻准备着

对文件系统来说，对元数据块缓冲区和数据库块缓冲区完成第八章第 11、12 节的操作就算它已经完成任务了，剩下的逻辑文件系统就不用操心了，jbd 会全权负责。

有一个内核线程，它的惟一作用就是将事务的数据提交到日志中。每隔固定间隔（一般是 5 秒）它都会醒来，选择合适的事务进行提交。我们也可以主动唤醒该线程进行事务提交。

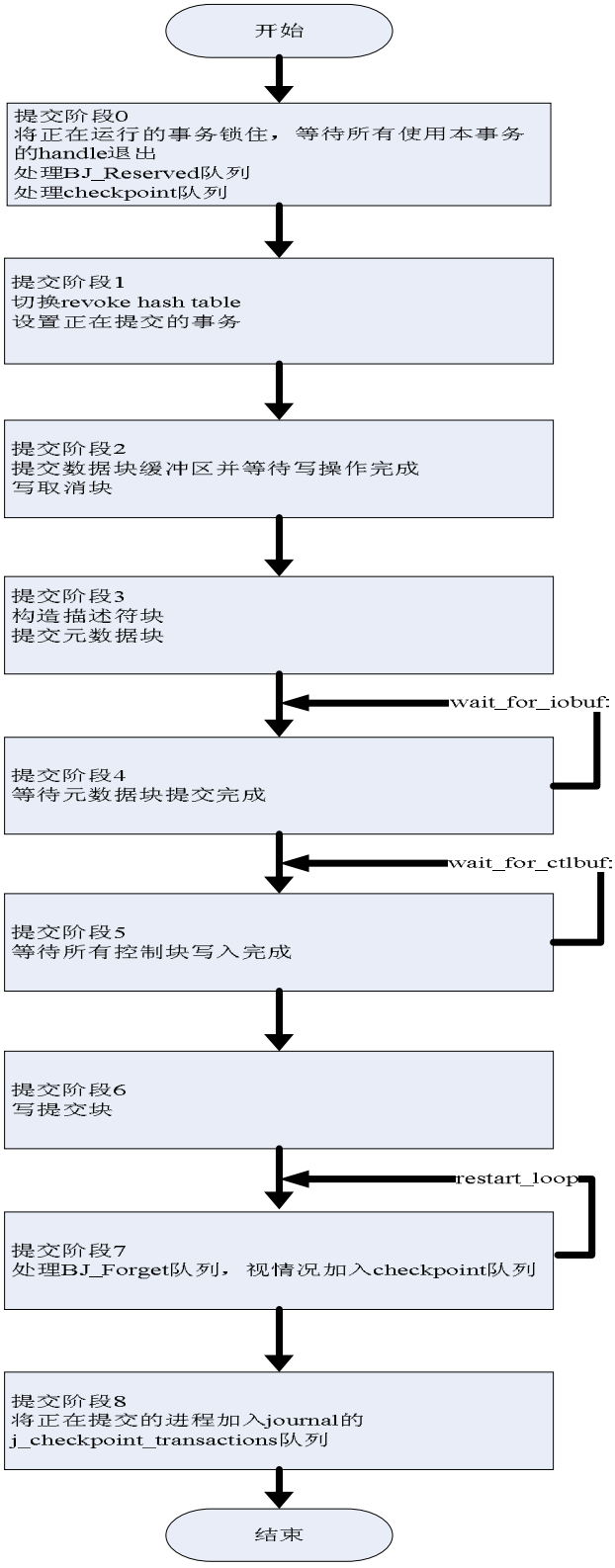
我们将 kjournald()中与睡眠相关的代码删掉，就可以很清楚地看到它的逻辑。

fs/jbd/journal.c

```
115 static int kjournald(void *arg)
116 {
117         journal_t *journal = arg;
118         transaction_t *transaction;
124         setup_timer(&journal->j_commit_timer, commit_timeout,
125                         (unsigned long)current);
126
127         /* Record that the journal thread is running */
128         journal->j_task = current;
129         wake_up(&journal->j_wait_done_commit);
130
137         spin_lock(&journal->j_state_lock);
138
139 loop:
146         if (journal->j_commit_sequence != journal->j_commit_request) {
                    // j_commit_sequence 表示已经提交的最新的 transaction 的 ID，
                    // j_commit_request 表示申请进行提交的 transaction 的 ID，
                    // 如果两者不等，我们进行一次提交。
148                 spin_unlock(&journal->j_state_lock);
149                 del_timer_sync(&journal->j_commit_timer);
                    // 进行事务的提交，详见第十章。
150                 journal_commit_transaction(journal);
151                 spin_lock(&journal->j_state_lock);
152                 goto loop;
153         }
154
204 end_loop:
205         spin_unlock(&journal->j_state_lock);
206         del_timer_sync(&journal->j_commit_timer);
207         journal->j_task = NULL;
208         wake_up(&journal->j_wait_done_commit);
209         jbd_debug(1, "Journal thread exiting.\n");
210         return 0;
211 }
```

十、 提交事务——我们放心了

1. journal_commit_transaction



图表 6 journal_commit_transaction 流程图

journal_commit_transaction()函数，长达 600 多行，是事务提交的主要函数。

```
302 void journal_commit_transaction(journal_t *journal)
303 {
304         transaction_t *commit_transaction;
305         struct journal_head *jh, *new_jh, *descriptor;
306         struct buffer_head **wbuf = journal->j_wbuf;
307         int bufs;
308         int flags;
309         int err;
310         unsigned int blocknr;
311         ktime_t start_time;
312         u64 commit_time;
313         char *tagp = NULL;
314         journal_header_t *header;
315         journal_block_tag_t *tag = NULL;
316         int space_left = 0;
317         int first_tag = 0;
318         int tag_flag;
319         int i;
320         int write_op = WRITE;

          // 将当前正在运行的 transaction 设置为正在提交的 transaction，然后进行提交。
344         commit_transaction = journal->j_running_transaction;
349
350         spin_lock(&journal->j_state_lock);
          // 将 transaction 的状态设置为 T_LOCKED，
          // 表示不能再向其中加入新的原子操作了。
351         commit_transaction->t_state = T_LOCKED;
358         if (commit_transaction->t_synchronous_commit)
359                 write_op = WRITE_SYNC_PLUG;
360         spin_lock(&commit_transaction->t_handle_lock);
361         while (commit_transaction->t_updates) {
                  // 因为此时，可能仍有 handle 在修改此 transaction，
                  // 故 361-374 行等所有的 handle 结束。
374         }
375         spin_unlock(&commit_transaction->t_handle_lock);
396         while (commit_transaction->t_reserved_list) {
                  // t_reserved_list 队列上的缓冲区是本 transaction 已管理的、但是未修改的缓冲区。既然未修改，则不必提交。
397                 jh = commit_transaction->t_reserved_list;
                  // 将该 jh 从本 transaction 中移除。
411                 journal_refile_buffer(journal, jh);
412         }
```

```
419     spin_lock(&journal->j_list_lock);
        // 先遍历 journal 中所有已提交的事务，依次处理其中的 checkpoint 队列
420     __journal_clean_checkpoint_list(journal);
421     spin_unlock(&journal->j_list_lock);
422
        // 提交阶段 1
423     jbd_debug (3, "JBD: commit phase 1\n");
424
        // 切换一下 revoke hash table。
        // 这样就不会发生该表以便被提交，一边被修改的情况。
428     journal_switch_revoke_table(journal);
429
        //将 transaction 的状态设置为 T_FLUSH，
430     commit_transaction->t_state = T_FLUSH;
431     journal->j_committing_transaction = commit_transaction;
432     journal->j_running_transaction = NULL;
433     start_time = ktime_get();
434     commit_transaction->t_log_start = journal->j_head;
435     wake_up(&journal->j_wait_transaction_locked);
436     spin_unlock(&journal->j_state_lock);
437
        // 提交阶段 2
        // 将与本 transaction 相关联的数据块缓冲区先写回磁盘，
        // ordered 模式就是这样实现的。
        // 这些数据块缓冲区原来在 BJ_SyncData 队列上，
        // journal_submit_data_buffers()会将它们移动到 BJ_Locked 队列上。
438     jbd_debug (3, "JBD: commit phase 2\n");
444     err = journal_submit_data_buffers(journal, commit_transaction,
445                                       write_op);
446
        // 等待数据块缓冲区写入完成。
450     spin_lock(&journal->j_list_lock);
451     while (commit_transaction->t_locked_list) {
            // 等待 bh 写回完成
            ……
            // 将 jh 从 transaction 中删除
480         if (buffer_jbd(bh) && bh2jh(bh) == jh &&
481             jh->b_transaction == commit_transaction &&
482             jh->b_jlist == BJ_Locked) {
483             __journal_unfile_buffer(jh);
484             jbd_unlock_bh_state(bh);
485             journal_remove_journal_head(bh);
486             put_bh(bh);
487         }
```

```
492         }
493         spin_unlock(&journal->j_list_lock);

505         // 将 revoke hash table 中的记录写到日志中。
            // 这就是取消块。
506         journal_write_revoke_records(journal, commit_transaction, write_op);
507
            // 提交阶段 3
            // 将元数据块缓冲区写到日志中
516         jbd_debug (3, "JBD: commit phase 3\n");

523         spin_lock(&journal->j_state_lock);
            // 将 transaction 的状态设置为 T_COMMIT，
524         commit_transaction->t_state = T_COMMIT;
525         spin_unlock(&journal->j_state_lock);

530         descriptor = NULL;
531         bufs = 0;
532         while (commit_transaction->t_buffers) {
                // 532-685 将 BJ_Metadata 队列(即 t_buffers)中的元数据缓冲区写到日志中。
                // 这里只是启动所有缓冲区的写操作，第四阶段会等待写操作完成。
536             jh = commit_transaction->t_buffers;
556
                // 元数据块缓冲区写入日志的格式是：描述符块、数据块、提交块
                // 这里的 descriptor 表示一个描述符块。
557             if (!descriptor) {
558                 struct buffer_head *bh;
                    // 从日志空间新分配一个块，用作描述符块
564                 descriptor = journal_get_descriptor_buffer(journal);
570                 bh = jh2bh(descriptor);
573                 header = (journal_header_t *)&bh->b_data[0];
                    // 设置描述符块
574                 header->h_magic     = cpu_to_be32(JFS_MAGIC_NUMBER);
575                 header->h_blocktype = cpu_to_be32(JFS_DESCRIPTOR_BLOCK);
576                 header->h_sequence  = cpu_to_be32(commit_transaction->t_tid);

                    // tagp 用于指示新的 journal_block_tag_t 的起点
578                 tagp = &bh->b_data[sizeof(journal_header_t)];
579                 space_left = bh->b_size - sizeof(journal_header_t);
580                 first_tag = 1;
581                 set_buffer_jwrite(bh);
582                 set_buffer_dirty(bh);
                    // wbuf[]数组记录要写到日志的所有缓冲区。
583                 wbuf[bufs++] = bh;
```

```
584
                // 将描述符块的缓冲区加入 BJ_LogCt 链表，下面会写出到日志。
588             journal_file_buffer(descriptor, commit_transaction,
589                                 BJ_LogCtl);
590         }
591
            // 计算日志中下一个空间块的块号，保存在 blocknr 中。
594         err = journal_next_log_block(journal, &blocknr);
608         commit_transaction->t_outstanding_credits--;
609
613         atomic_inc(&jh2bh(jh)->b_count);

            // journal_write_metadata_buffer()函数从字面意思上看是写一个元数据缓冲区，
            // 但是实际上它并未完成真正的写操作。
            // 它主要的作用是根据已存在的元数据块缓冲区（jh 对应的那个），
            // 创建一个新的缓冲区（new_jh 对应的那个），供 jbd 写入到日志时使用。
            // jh 此时会从 BJ_Metadata 队列上移动到 BJ_Shadow 队列上，
            // 而 new_jh 则会链入到 BJ_IO 队列上。
            // journal_write_metadata_buffer()函数也会处理转义问题。
627         flags = journal_write_metadata_buffer(commit_transaction,
628                                               jh, &new_jh, blocknr);
629         set_bit(BH_JWrite, &jh2bh(new_jh)->b_state);

            // 记录下来，后面一起写出。
630         wbuf[bufs++] = jh2bh(new_jh);

            // 635-652，将刚写出的缓冲区信息记录在描述符块中。
635         tag_flag = 0;
636         if (flags & 1)
637             tag_flag |= JFS_FLAG_ESCAPE;
638         if (!first_tag)
639             tag_flag |= JFS_FLAG_SAME_UUID;
640
641         tag = (journal_block_tag_t *) tagp;
            // 记录块号和标志。
642         tag->t_blocknr = cpu_to_be32(jh2bh(jh)->b_blocknr);
643         tag->t_flags = cpu_to_be32(tag_flag);
644         tagp += sizeof(journal_block_tag_t);
645         space_left -= sizeof(journal_block_tag_t);
646
647         if (first_tag) {
                // 第一个 journal_block_tag_t 会记录一个 UUID
648             memcpy (tagp, journal->j_uuid, 16);
649             tagp += 16;
```

```
650                 space_left -= 16;
651                 first_tag = 0;
652             }

                // 如果达到了 journal 一次允许写出的缓冲区个数，
                // 或者 BJ_Metadata 队列已经为空，
                // 或者描述符块已被 journal_block_tag_t 记录填满了
                // 则需进行提交
657             if (bufs == journal->j_wbufsize ||
658                 commit_transaction->t_buffers == NULL ||
659                 space_left < sizeof(journal_block_tag_t) + 16) {
                    // 最后一个 journal_block_tag_t 要设置 JFS_FLAG_LAST_TAG 标志。
667                 tag->t_flags |= cpu_to_be32(JFS_FLAG_LAST_TAG);
668
669 start_journal_io:
670                 for (i = 0; i < bufs; i++) {
                        // 启动写操作。
671                     struct buffer_head *bh = wbuf[i];
672                     lock_buffer(bh);
673                     clear_buffer_dirty(bh);
674                     set_buffer_uptodate(bh);
675                     bh->b_end_io = journal_end_buffer_io_sync;
676                     submit_bh(write_op, bh);
677                 }
678                 cond_resched();
679
                    // 如果描述符块已被 journal_block_tag_t 记录填满了，则重新分配一个。
682                 descriptor = NULL;
683                 bufs = 0;
684             }
685         }
            // 提交阶段 4
            // 等待元数据块缓冲区写入日志。
698         jbd_debug(3, "JBD: commit phase 4\n");
704 wait_for_iobuf:
705         while (commit_transaction->t_iobuf_list != NULL) {
                // 705-753 循环，等待每个元数据块缓冲区写入日志。
706             struct buffer_head *bh;
707
708             jh = commit_transaction->t_iobuf_list->b_tprev;
709             bh = jh2bh(jh);
710             if (buffer_locked(bh)) {
711                 wait_on_buffer(bh);
712                 goto wait_for_iobuf;
```

```
713             }
714             if (cond_resched())
715                     goto wait_for_iobuf;
716
720             clear_buffer_jwrite(bh);
723             journal_unfile_buffer(journal, jh);
730             journal_put_journal_head(jh);
731             __brelse(bh);
733             free_buffer_head(bh);
734
735             /* We also have to unlock and free the corresponding
736                                     shadowed buffer */
                // BJ_IO 队列上的缓冲区与 BJ_Shadow 队列上的缓冲区是一一对应的。
                // 每处理完一个 BJ_IO 队列上的缓冲区，都会将其从 transaction 中删除，
                // 并且要将在 BJ_Shadow 队列上对应的缓冲区移到 BJ_Forget 队列上。
                // (后便 checkpoint 还要处理 BJ_Forget 队列)
737             jh = commit_transaction->t_shadow_list->b_tprev;
738             bh = jh2bh(jh);
739             clear_bit(BH_JWrite, &bh->b_state);
747             journal_file_buffer(jh, commit_transaction, BJ_Forget);
750             wake_up_bit(&bh->b_state, BH_Unshadow);
752             __brelse(bh);
753     }

        // 提交阶段 5
        // 等待取消块和描述符块写入日志。
        // 取消块和描述符块都可视为 jbd 的控制块，都在 BJ_LogCtl 队列上。
        // 参考上面的 588 行。
757     jbd_debug(3, "JBD: commit phase 5\n");
760 wait_for_ctlbuf:
761     while (commit_transaction->t_log_list != NULL) {
            // 等待每一个控制块都写入日志
762             struct buffer_head *bh;
763
764             jh = commit_transaction->t_log_list->b_tprev;
765             bh = jh2bh(jh);
766             if (buffer_locked(bh)) {
767                     wait_on_buffer(bh);
768                     goto wait_for_ctlbuf;
769             }
770             if (cond_resched())
771                     goto wait_for_ctlbuf;
772
773             if (unlikely(!buffer_uptodate(bh)))
```

```
774                     err = -EIO;

                // 控制块已经写到日志中了，则可以从 transaction 中删除了。
777             clear_buffer_jwrite(bh);
778             journal_unfile_buffer(journal, jh);
779             journal_put_journal_head(jh);
780             __brelse(bh);           /* One for getblk */
782         }

            // 提交阶段 6
            // 描述符块和数据块都写到日志中了，
            // 现在我们写一个提交块。
787         jbd_debug(3, "JBD: commit phase 6\n");
788
790         spin_lock(&journal->j_state_lock);
            //将 transaction 的状态设置为 T_COMMIT_RECORD
792         commit_transaction->t_state = T_COMMIT_RECORD;
793         spin_unlock(&journal->j_state_lock);
794
            // 将提交块同步地写到日志中。
795         if (journal_write_commit_record(journal, commit_transaction))
796             err = -EIO;

            // 提交阶段 7
806         jbd_debug(3, "JBD: commit phase 7\n");
807         // 确保下列队列都为空了！
808         J_ASSERT(commit_transaction->t_sync_datalist == NULL);
809         J_ASSERT(commit_transaction->t_buffers == NULL);
810         J_ASSERT(commit_transaction->t_checkpoint_list == NULL);
811         J_ASSERT(commit_transaction->t_iobuf_list == NULL);
812         J_ASSERT(commit_transaction->t_shadow_list == NULL);
813         J_ASSERT(commit_transaction->t_log_list == NULL);
814
815 restart_loop:
820         spin_lock(&journal->j_list_lock);
821         while (commit_transaction->t_forget) {
                // 这里实现 checkpoint 机制的地方。
                // 此时，对元数据块缓冲区而言，
                // 或者它已被内核按照原有的方式写回到磁盘的原始位置(update 状态)，
                // 那么直接从 transaction 中删除即可；
                // 或者未被内核写回到磁盘的原始位置(dirty 状态)，
                // 这样我们就要把它加入到本 transaction 德 checkpoint 队列上，
                // 本函数的 420 行会处理该队列。
822             transaction_t *cp_transaction;
```

```
823             struct buffer_head *bh;
824
825             jh = commit_transaction->t_forget;
826             spin_unlock(&journal->j_list_lock);
827             bh = jh2bh(jh);
828             jbd_lock_bh_state(bh);

854             spin_lock(&journal->j_list_lock);
855             cp_transaction = jh->b_cp_transaction;
856             if (cp_transaction) {
                    // 如果该 jh 已经在旧的 transaction 的 checkpoint 队列上了，
                    // 则从旧的 transaction 的 checkpoint 队列上删除，
                    // 因为我们会把它加入到本 transaction 的 checkpoint 队列上。
858                 __journal_remove_checkpoint(jh);
859             }

880             if (buffer_jbddirty(bh)) {
                    // 如果该缓冲区仍为脏，则加入到本 transaction 的 checkpoint 队列上。
882                 __journal_insert_checkpoint(jh, commit_transaction);
                    // 该 jh 除了在本 transaction 的 checkpoint 队列上之外，
                    // 不需要在其他队列上了。
                    // 从 transaction 的队列上移除。
886                 __journal_refile_buffer(jh);
887                 jbd_unlock_bh_state(bh);
888             } else {
                    // 该缓冲区已处于 update 状态，
                    // 直接从 transaction 中删除即可。
898                 __journal_refile_buffer(jh);
899                 if (!jh->b_transaction) {
900                     jbd_unlock_bh_state(bh);
902                     journal_remove_journal_head(bh);
903                     release_buffer_page(bh);
904                 } else
905                     jbd_unlock_bh_state(bh);
906             }
907             cond_resched_lock(&journal->j_list_lock);
908         }
909         spin_unlock(&journal->j_list_lock);


916         spin_lock(&journal->j_state_lock);
917         spin_lock(&journal->j_list_lock);

        // 如果因为加锁过程中 BJ_Forget 队列上又有缓冲区了，
```

```
        // 转到 815 行继续处理
922     if (commit_transaction->t_forget) {
923             spin_unlock(&journal->j_list_lock);
924             spin_unlock(&journal->j_state_lock);
925             goto restart_loop;    // 815 行
926     }
927
        // 提交阶段 8
        // 事务提交已经完成了，
        // 将本事务加入 journal->j_checkpoint_transactions 队列
930     jbd_debug(3, "JBD: commit phase 8\n");
        //将 transaction 的状态设置为 T_FINISHED。
934     commit_transaction->t_state = T_FINISHED;
935     J_ASSERT(commit_transaction == journal->j_committing_transaction);
936     journal->j_commit_sequence = commit_transaction->t_tid;
937     journal->j_committing_transaction = NULL;
938     commit_time = ktime_to_ns(ktime_sub(ktime_get(), start_time));
939
944     if (likely(journal->j_average_commit_time))
945             journal->j_average_commit_time = (commit_time*3 +
946                             journal->j_average_commit_time) / 4;
947     else
948             journal->j_average_commit_time = commit_time;
949
950     spin_unlock(&journal->j_state_lock);
951
952     if (commit_transaction->t_checkpoint_list == NULL &&
953             commit_transaction->t_checkpoint_io_list == NULL) {
                // 本事务的 checkpoint 已经为空，则本事务可从内存中直接删除了。
954             __journal_drop_transaction(journal, commit_transaction);
955     } else {
                // 头插法
956             if (journal->j_checkpoint_transactions == NULL) {
957                     journal->j_checkpoint_transactions = commit_transaction;
958                     commit_transaction->t_cpnext = commit_transaction;
959                     commit_transaction->t_cpprev = commit_transaction;
960             } else {
961                     commit_transaction->t_cpnext =
962                             journal->j_checkpoint_transactions;
963                     commit_transaction->t_cpprev =
964                             commit_transaction->t_cpnext->t_cpprev;
965                     commit_transaction->t_cpnext->t_cpprev =
966                             commit_transaction;
967                     commit_transaction->t_cpprev->t_cpnext =
```

```
968                         commit_transaction;
969                 }
970         }
971         spin_unlock(&journal->j_list_lock);
976         wake_up(&journal->j_wait_done_commit);
977 }
```

2. __journal_clean_checkpoint_list

journal_commit_transaction()函数中调用的以下几个函数仍需要仔细分析一下。

__journal_clean_checkpoint_list()函数的作用是处理 journal->j_checkpoint_transactions 队列，如果某个 transaction 的 checkpoint 队列上的缓冲区已经处于 update 状态，则从该 transaction 的 checkpoint 队列上删除；如果某个 transaction 的 checkpoint 队列已经为空，则从 journal->j_checkpoint_transactions 队列上删除。

fs/jbd/checkpoint.c

```
579 int __journal_clean_checkpoint_list(journal_t *journal)
580 {
581         transaction_t *transaction, *last_transaction, *next_transaction;
582         int ret = 0;
583         int released;
584
585         transaction = journal->j_checkpoint_transactions;
586         if (!transaction)
587                 goto out;
588
589         last_transaction = transaction->t_cpprev;
590         next_transaction = transaction;
591         do {
                    // 循环处理 journal 中的每个事务
592                 transaction = next_transaction;
593                 next_transaction = transaction->t_cpnext;
                    // 处理一个事务的 checkpoint 队列。
594                 ret += journal_clean_one_cp_list(transaction->
595                                 t_checkpoint_list, &released);
601                 if (need_resched())
602                         goto out;
603                 if (released)
604                         continue;
610                 ret += journal_clean_one_cp_list(transaction->
611                                 t_checkpoint_io_list, &released);
612                 if (need_resched())
613                         goto out;
614         } while (transaction != last_transaction);
615 out:
```

```
616         return ret;
617 }
```

journal_clean_one_cp_list()函数的作用是处理一个 transaction 中的 checkpoint 队列。

```
531 static int journal_clean_one_cp_list(struct journal_head *jh, int *released)
532 {
533         struct journal_head *last_jh;
534         struct journal_head *next_jh = jh;
535         int ret, freed = 0;
536
537         *released = 0;
538         if (!jh)
539                 return 0;
540
541         last_jh = jh->b_cpprev;
542         do {
                    // 循环处理该 checkpoint 队列上的每一个缓冲区
543                 jh = next_jh;
544                 next_jh = jh->b_cpnext;
546                 if (jbd_trylock_bh_state(jh2bh(jh))) {
                            // 处理一个缓冲区
547                         ret = __try_to_free_cp_buf(jh);
555                 }
564         } while (jh != last_jh);
565
566         return freed;
567 }
```

```
 91 static int __try_to_free_cp_buf(struct journal_head *jh)
 92 {
 93         int ret = 0;
 94         struct buffer_head *bh = jh2bh(jh);
 96         if (jh->b_jlist == BJ_None && !buffer_locked(bh) &&
 97                 !buffer_dirty(bh) && !buffer_write_io_error(bh)) {
                    // 如果该缓冲区不是脏，即为 update 状态，则立即从 checkpoint 队列上删除。
                    // 从此，该 jh 与本 transaction 再无关系。
 99                 ret = __journal_remove_checkpoint(jh) + 1;
100                 jbd_unlock_bh_state(bh);
101                 journal_remove_journal_head(bh);
103                 __brelse(bh);
104         } else {
105                 jbd_unlock_bh_state(bh);
106         }
107         return ret;
108 }
```

3. journal_submit_data_buffers

journal_submit_data_buffers()函数将与本 transaction 相关联的数据块缓冲区先写回磁盘，ordered 模式就是这样实现的。这些数据块缓冲区原来在 BJ_SyncData 队列上， 现在会移动到 BJ_Locked 队列上。

这里本函数只是提出了写缓冲区的申请，但是并不保证缓冲区同步地写回磁盘。

fs/jbd/commit.c

```
189 static int journal_submit_data_buffers(journal_t *journal,
190                                         transaction_t *commit_transaction,
191                                         int write_op)
192 {
193         struct journal_head *jh;
194         struct buffer_head *bh;
195         int locked;
196         int bufs = 0;
197         struct buffer_head **wbuf = journal->j_wbuf;
198         int err = 0;
208 write_out_data:
209         cond_resched();
210         spin_lock(&journal->j_list_lock);
211
212         while (commit_transaction->t_sync_datalist) {
                // 依次处理每个缓冲区
213             jh = commit_transaction->t_sync_datalist;
214             bh = jh2bh(jh);
215             locked = 0;
243             if (!buffer_jbd(bh) || bh2jh(bh) != jh
244                     || jh->b_transaction != commit_transaction
245                     || jh->b_jlist != BJ_SyncData) {
                    // 该缓冲区已经由别人写回磁盘了，
                    // 那太幸运了
246                     jbd_unlock_bh_state(bh);
247                     if (locked)
248                             unlock_buffer(bh);
249                     BUFFER_TRACE(bh, "already cleaned up");
250                     release_data_buffer(bh);
251                     continue;
252             }
253             if (locked && test_clear_buffer_dirty(bh)) {
                    // 如果该缓冲区仍是脏的，则需要我们写回磁盘了
                    // 在数组中记录该缓冲区。
255                     wbuf[bufs++] = bh;
                    // 将缓冲区移动到 BJ_Locked 队列上。
256                     __journal_file_buffer(jh, commit_transaction,
```

```
257                                 BJ_Locked);
258                 jbd_unlock_bh_state(bh);
259                 if (bufs == journal->j_wbufsize) {
                        // 如果缓冲区个数已累积到 jouranl 一次写操作允许的最大值，
                        // 则提交写操作。
260                     spin_unlock(&journal->j_list_lock);
261                     journal_do_submit_data(wbuf, bufs, write_op);
262                     bufs = 0;
263                     goto write_out_data;
264                 }
265             } else if (!locked && buffer_locked(bh)) {
266                 __journal_file_buffer(jh, commit_transaction,
267                                 BJ_Locked);
268                 jbd_unlock_bh_state(bh);
269                 put_bh(bh);
270             } else {
                    // 该缓冲区已经写回磁盘了
272                 if (unlikely(!buffer_uptodate(bh)))
273                     err = -EIO;
                    // 从本 transaction 中删除即可。
274                 __journal_unfile_buffer(jh);
275                 jbd_unlock_bh_state(bh);
276                 if (locked)
277                     unlock_buffer(bh);
278                 journal_remove_journal_head(bh);
281                 put_bh(bh);
282                 release_data_buffer(bh);
283             }
289         }
290         spin_unlock(&journal->j_list_lock);
291         journal_do_submit_data(wbuf, bufs, write_op);
292
293         return err;
294 }
```

journal_do_submit_data()函数只是提交写操作而已。

```
174 static void journal_do_submit_data(struct buffer_head **wbuf, int bufs,
175                                    int write_op)
176 {
177         int i;
178
179         for (i = 0; i < bufs; i++) {
180             wbuf[i]->b_end_io = end_buffer_write_sync;
181             /* We use-up our safety reference in submit_bh() */
182             submit_bh(write_op, wbuf[i]);
```

```
183         }
184 }
185
```

4. journal_write_revoke_records

journal_write_revoke_records()函数的作用是将取消块的信息写到日志中。

fs/jbd/revoke.c

```
503 void journal_write_revoke_records(journal_t *journal,
504                                    transaction_t *transaction, int write_op)
505 {
506         struct journal_head *descriptor;
507         struct jbd_revoke_record_s *record;
508         struct jbd_revoke_table_s *revoke;
509         struct list_head *hash_list;
510         int i, offset, count;
511
512         descriptor = NULL;
513         offset = 0;
514         count = 0;

517         revoke = journal->j_revoke == journal->j_revoke_table[0] ?
518                 journal->j_revoke_table[1] : journal->j_revoke_table[0];
519
520         for (i = 0; i < revoke->hash_size; i++) {
                    // 循环处理 hash 表的每一个链表。
521                 hash_list = &revoke->hash_table[i];
522
523                 while (!list_empty(hash_list)) {
                            // 循环处理每一个记录
524                         record = (struct jbd_revoke_record_s *)
525                                 hash_list->next;
                            // “写”一个 revoke 记录。
                            // 其实不是真正的写，只是记录在取消块缓冲区中。
526                         write_one_revoke_record(journal, transaction,
527                                                 &descriptor, &offset,
528                                                 record, write_op);
529                         count++;
530                         list_del(&record->hash);
531                         kmem_cache_free(revoke_record_cache, record);
532                 }
533         }
534         if (descriptor)
535                 flush_descriptor(journal, descriptor, offset, write_op);
537 }
```

write_one_revoke_record()函数的作用是将一个 revoke 记录写到取消块缓冲区中。

```
544 static void write_one_revoke_record(journal_t *journal,
545                                       transaction_t *transaction,
546                                       struct journal_head **descriptorp,
547                                       int *offsetp,
548                                       struct jbd_revoke_record_s *record,
549                                       int write_op)
550 {
551         struct journal_head *descriptor;
552         int offset;
553         journal_header_t *header;
562         descriptor = *descriptorp;
563         offset = *offsetp;
566         if (descriptor) {
567                 if (offset == journal->j_blocksize) {
                            // 如果取消块缓冲区已经写满了，
                            // 则刷新该缓冲区。
568                         flush_descriptor(journal, descriptor, offset, write_op);
569                         descriptor = NULL;
570                 }
571         }
572
573         if (!descriptor) {
                    // 如果尚未分配取消块缓冲区，
                    // 则从日志中分配一个未使用的块作为取消块缓冲区。
574                 descriptor = journal_get_descriptor_buffer(journal);
575                 if (!descriptor)
576                         return;
                    // 设置取消块
577                 header = (journal_header_t *) &jh2bh(descriptor)->b_data[0];
578                 header->h_magic         = cpu_to_be32(JFS_MAGIC_NUMBER);
579                 header->h_blocktype = cpu_to_be32(JFS_REVOKE_BLOCK);
580                 header->h_sequence    = cpu_to_be32(transaction->t_tid);

                    // 取消块也是 jbd 的控制块。
584                 journal_file_buffer(descriptor, transaction, BJ_LogCtl);
585
586                 offset = sizeof(journal_revoke_header_t);
587                 *descriptorp = descriptor;
588         }
589
            // 取消块中记录什么？
            // 只需记录块号即可！
590         * ((__be32 *)(&jh2bh(descriptor)->b_data[offset])) =
```

```
591             cpu_to_be32(record->blocknr);
592         offset += 4;
593         *offsetp = offset;
594 }
```

flush_descriptor()函数的作用是启动该取消块的写操作。

```
603 static void flush_descriptor(journal_t *journal,
604                               struct journal_head *descriptor,
605                               int offset, int write_op)
606 {
607         journal_revoke_header_t *header;
608         struct buffer_head *bh = jh2bh(descriptor);
615         header = (journal_revoke_header_t *) jh2bh(descriptor)->b_data;
616         header->r_count = cpu_to_be32(offset);
617         set_buffer_jwrite(bh);
619         set_buffer_dirty(bh);
620         ll_rw_block((write_op == WRITE) ? SWRITE : SWRITE_SYNC_PLUG, 1, &bh);
621 }
```

5. journal_write_metadata_buffer

journal_write_metadata_buffer()函数的作用是将一个元数据块缓冲区写到日志中，采用的方式是根据输入的缓冲区 jh_in，新创建一个数据与 jh_in 中的数据相同，但是指向日志中的某一个块的缓冲区。

fs/jbd/journal.c

```
 277 int journal_write_metadata_buffer(transaction_t *transaction,
 278                                   struct journal_head    *jh_in,
 279                                   struct journal_head **jh_out,
 280                                   unsigned int blocknr)
 281 {
 282         int need_copy_out = 0;
 283         int done_copy_out = 0;
 284         int do_escape = 0;
 285         char *mapped_data;
 286         struct buffer_head *new_bh;
 287         struct journal_head *new_jh;
 288         struct page *new_page;
 289         unsigned int new_offset;
 290         struct buffer_head *bh_in = jh2bh(jh_in);
 291         journal_t *journal = transaction->t_journal;
 303
             // 新创建一个缓冲区结构 new_bh
 304         new_bh = alloc_buffer_head(GFP_NOFS|__GFP_NOFAIL);
 306         new_bh->b_state = 0;
 307         init_buffer(new_bh, NULL, NULL);
 308         atomic_set(&new_bh->b_count, 1);
```

```
309         new_jh = journal_add_journal_head(new_bh);    /* This sleeps */
315         jbd_lock_bh_state(bh_in);
316 repeat:
317         if (jh_in->b_frozen_data) {
318                 done_copy_out = 1;
319                 new_page = virt_to_page(jh_in->b_frozen_data);
320                 new_offset = offset_in_page(jh_in->b_frozen_data);
321         } else {
322                 new_page = jh2bh(jh_in)->b_page;
323                 new_offset = offset_in_page(jh2bh(jh_in)->b_data);
324         }
325
326         mapped_data = kmap_atomic(new_page, KM_USER0);

        // 判断是否需要转义
330         if (*((__be32 *)(mapped_data + new_offset)) ==
331                         cpu_to_be32(JFS_MAGIC_NUMBER)) {
332                 need_copy_out = 1;
333                 do_escape = 1;
334         }
335         kunmap_atomic(mapped_data, KM_USER0);

340         if (need_copy_out && !done_copy_out) {
341                 char *tmp;
342
343                 jbd_unlock_bh_state(bh_in);
344                 tmp = jbd_alloc(bh_in->b_size, GFP_NOFS);
345                 jbd_lock_bh_state(bh_in);
346                 if (jh_in->b_frozen_data) {
347                         jbd_free(tmp, bh_in->b_size);
348                         goto repeat;
349                 }
350
351                 jh_in->b_frozen_data = tmp;
352                 mapped_data = kmap_atomic(new_page, KM_USER0);
353                 memcpy(tmp, mapped_data + new_offset, jh2bh(jh_in)->b_size);
354                 kunmap_atomic(mapped_data, KM_USER0);
355
356                 new_page = virt_to_page(tmp);
357                 new_offset = offset_in_page(tmp);
358                 done_copy_out = 1;
359         }

365         if (do_escape) {
```

```
                // 处理转义
366             mapped_data = kmap_atomic(new_page, KM_USER0);
367             *((unsigned int *)(mapped_data + new_offset)) = 0;
368             kunmap_atomic(mapped_data, KM_USER0);
369         }
370
371         set_bh_page(new_bh, new_page, new_offset);
372         new_jh->b_transaction = NULL;
373         new_bh->b_size = jh2bh(jh_in)->b_size;
374         new_bh->b_bdev = transaction->t_journal->j_dev;
          // 注意，这个 blocknr 是从日志中分配的一个磁盘块
375         new_bh->b_blocknr = blocknr;
376         set_buffer_mapped(new_bh);
377         set_buffer_dirty(new_bh);
378
379         *jh_out = new_jh;
387         spin_lock(&journal->j_list_lock);
          // 将原来的元数据块缓冲区加入到 BJ_Shadow 队列
388         __journal_file_buffer(jh_in, transaction, BJ_Shadow);
389         spin_unlock(&journal->j_list_lock);
390         jbd_unlock_bh_state(bh_in);
391
392         // 将新创建的缓冲区加入到 BJ_IO 队列
393         journal_file_buffer(new_jh, transaction, BJ_IO);
394
395         return do_escape | (done_copy_out << 1);
396 }
```

6. journal_write_commit_record

journal_write_commit_record()函数的作用是写一个提交块。

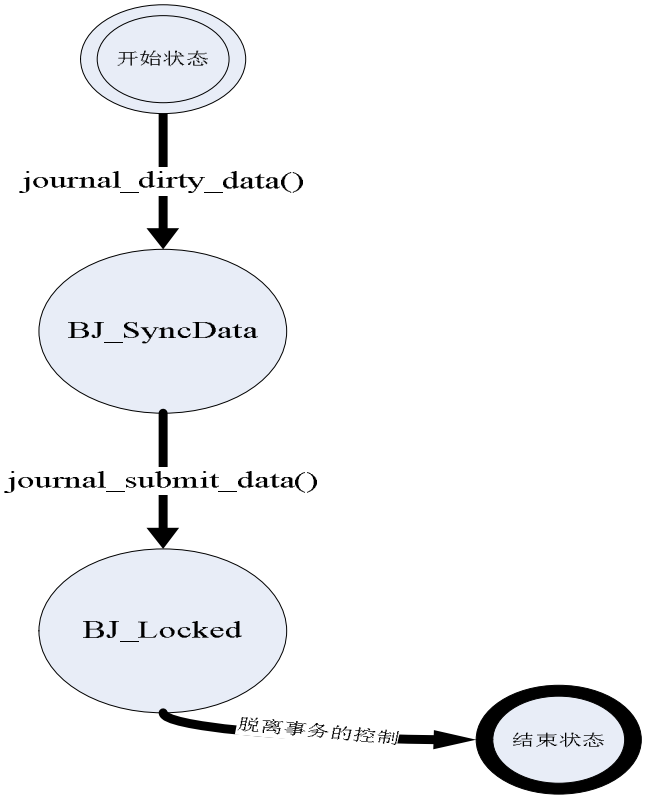
fs/jbd/commit.c

```
115 static int journal_write_commit_record(journal_t *journal,
116                                        transaction_t *commit_transaction)
117 {
118         struct journal_head *descriptor;
119         struct buffer_head *bh;
120         journal_header_t *header;
121         int ret;
122         int barrier_done = 0;

          // 从日志中分配一个块，作为提交块。
127         descriptor = journal_get_descriptor_buffer(journal);
131         bh = jh2bh(descriptor);
132
```

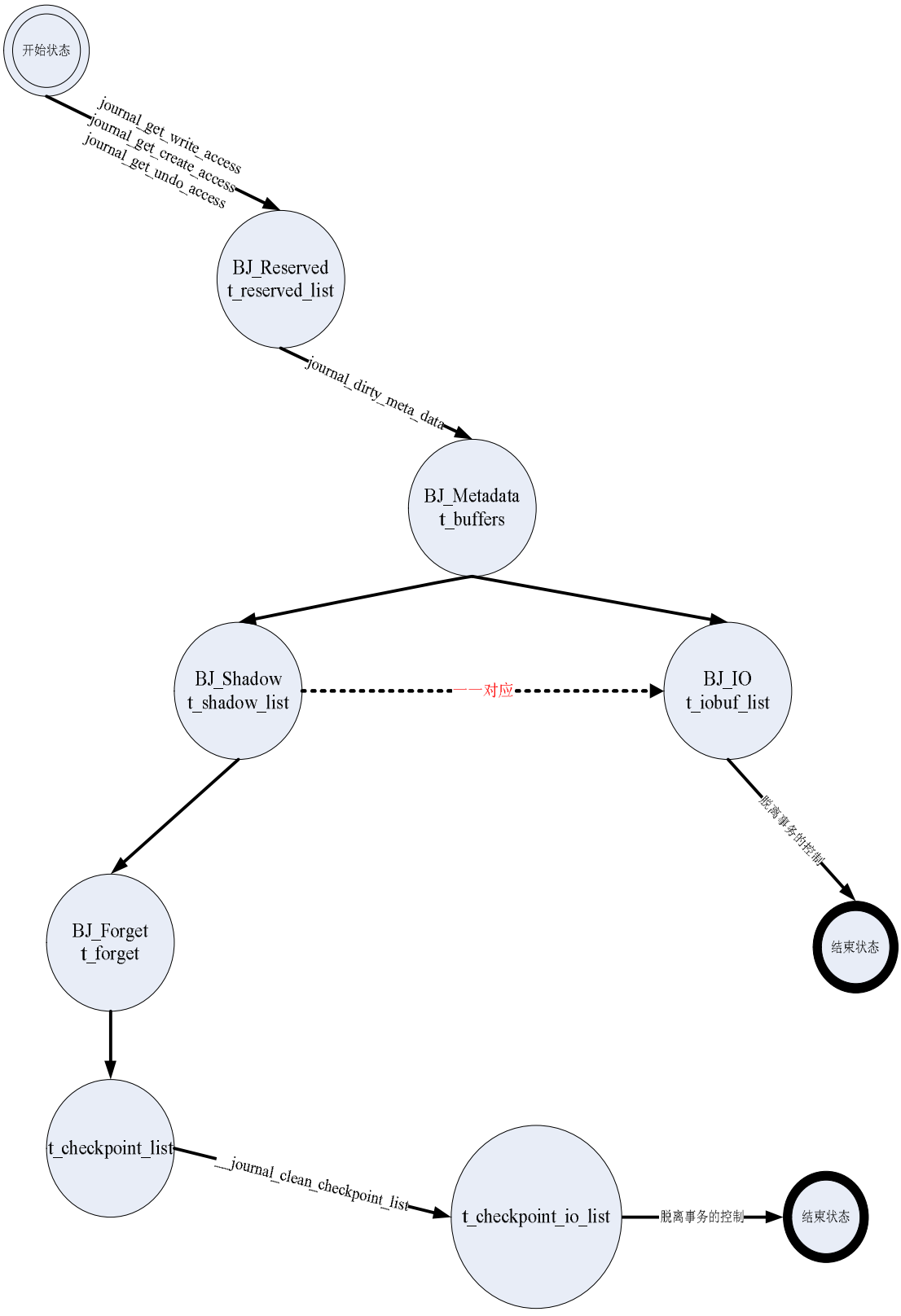
```
        // 设置提交块信息
133     header = (journal_header_t *)(bh->b_data);
134     header->h_magic = cpu_to_be32(JFS_MAGIC_NUMBER);
135     header->h_blocktype = cpu_to_be32(JFS_COMMIT_BLOCK);
136     header->h_sequence = cpu_to_be32(commit_transaction->t_tid);
137
139     set_buffer_dirty(bh);
        // 同步将提交块写到日志中
144     ret = sync_dirty_buffer(bh);
168     put_bh(bh);         /* One for getblk() */
169     journal_put_journal_head(descriptor);
170
171     return (ret == -EIO);
172 }
```

十一、 数据块缓冲区状态转移图



图表 7 数据块缓冲区状态转移图

十二、 元数据块缓冲区状态转移图



图表 8 元数据块缓冲区状态转移图

图 6 中同时标出了 transaction 中对应的队列的名字。

十三、 恢复日志——奇迹发生了

1．恢复前的准备工作

这里只解释与日志机制相关的内容。

ext3_fill_super()　读入文件系统的超级块

à ext3_load_journal()　读入对应日志的 inode 信息

à ext3_get_journal()　根据日志 inode 信息构建一个 journal_t 结构

à journal_load()　进行日志的恢复操作

à load_superblock()　读入日志的超级块

à journal_recover()　进行实际的日志恢复操作。

2．journal_recover 函数

journal_recover()函数的主要作用是分三个阶段进行日志的恢复操作。

fs/jbd/recovery.c

```
223 int journal_recover(journal_t *journal)
224 {
225         int                err, err2;
226         journal_superblock_t *    sb;
227
228         struct recovery_info       info;
229
230         memset(&info, 0, sizeof(info));
231         sb = journal->j_superblock;
238
239         if (!sb->s_start) {
                // 如果文件系统是被正常卸载的，则不需要恢复。
                // 递增 j_transaction_sequence，使整个日志无效。
242             journal->j_transaction_sequence = be32_to_cpu(sb->s_sequence) + 1;
243             return 0;
244         }
245
            // 恢复步骤 1：PASS_SCAN
            // 这个步骤主要的作用是找到日志的起点和终点，
            // 注意日志空间可看做一个环形结构。
246         err = do_one_pass(journal, &info, PASS_SCAN);

            // 恢复步骤 2：PASS_REVOKE
            // 这个步骤主要的作用找到 revoke 块，并把信息读入内存的 revoke hash table。
247         if (!err)
248             err = do_one_pass(journal, &info, PASS_REVOKE);

            // 恢复步骤 3：PASS_REPLAY
```

```
        // 这个步骤主要的作用是根据描述符块的指示，
        // 将日志中的数据块写回到磁盘的原始位置上。
249     if (!err)
250         err = do_one_pass(journal, &info, PASS_REPLAY);

        // 恢复完成，递增 j_transaction_sequence，使整个日志无效。
260     journal->j_transaction_sequence = ++info.end_transaction;
261
        // 清空内存中的 revoke hash table。
262     journal_clear_revoke(journal);
        // 同步日志所在的设备。
263     err2 = sync_blockdev(journal->j_fs_dev);
266
267     return err;
268 }
```

三个步骤实际上调用的都是一个函数 do_one_pass，只不过参数不同罢了。

首先看一个结构 recovery_info，它是在恢复过程中保留日志信息的。

fs/jbd/recovery.c

```
 29 struct recovery_info
 30 {
 31     tid_t       start_transaction;
 32     tid_t       end_transaction;
 33
 34     int     nr_replays;
 35     int     nr_revokes;
 36     int     nr_revoke_hits;
 37 };
```

3．恢复步骤 1：PASS_SCAN

fs/jbd/recovery.c

```
312 static int do_one_pass(journal_t *journal,
313                         struct recovery_info *info, enum passtype pass)
314 {
315     unsigned int        first_commit_ID, next_commit_ID;
316     unsigned int        next_log_block;
317     int         err, success = 0;
318     journal_superblock_t *  sb;
319     journal_header_t *  tmp;
320     struct buffer_head *    bh;
321     unsigned int        sequence;
322     int         blocktype;
335     sb = journal->j_superblock;
336     next_commit_ID = be32_to_cpu(sb->s_sequence);
```

```
        // next_log_block 表示要在日志中读的下一个块的块号
337     next_log_block = be32_to_cpu(sb->s_start);
338
339     first_commit_ID = next_commit_ID;
340     if (pass == PASS_SCAN)
341         info->start_transaction = first_commit_ID;

352     while (1) {
            // 遍历所有的日志块
353         int             flags;
354         char *          tagp;
355         journal_block_tag_t *   tag;
356         struct buffer_head *    obh;
357         struct buffer_head *    nbh;

            // 读入日志中的下一个块
377         err = jread(&bh, journal, next_log_block);
381         next_log_block++;

            // 注意日志是个环形结构
382         wrap(journal, next_log_block);
390         tmp = (journal_header_t *)bh->b_data;
391
392         if (tmp->h_magic != cpu_to_be32(JFS_MAGIC_NUMBER)) {
                // 如果该块不是日志的描述块，则说明已经处理完了，退出
393             brelse(bh);
394             break;
395         }
396
397         blocktype = be32_to_cpu(tmp->h_blocktype);
398         sequence = be32_to_cpu(tmp->h_sequence);
402         if (sequence != next_commit_ID) {
                // 如果序号不对，退出
403             brelse(bh);
404             break;
405         }

            // 根据描述块的类型，进行相应的处理
411         switch(blocktype) {
412         case JFS_DESCRIPTOR_BLOCK:
                // 描述符块
416             if (pass != PASS_REPLAY) {
                    // 我们现在在 PASS_SCAN 中，
                    // count_tags()函数会计算该描述符块中一共描述了对少个数据块的
```

对应关系，现在直接跳过数据块即可。

```
417                         next_log_block +=
418                             count_tags(bh, journal->j_blocksize);

419                         wrap(journal, next_log_block);
420                         brelse(bh);
421                         continue;
422                     }

510                 case JFS_COMMIT_BLOCK:
                        // 提交块
514                     brelse(bh);
515                     next_commit_ID++;
516                     continue;
517
518                 case JFS_REVOKE_BLOCK:
                        // 取消块，只在取消步骤中处理。
521                     if (pass != PASS_REVOKE) {
522                         brelse(bh);
523                         continue;
524                     }

533                 default:
                        // 不是以上的四种块，完成。
536                     brelse(bh);
537                     goto done;
538                 }
539         }
540
541     done:
549         if (pass == PASS_SCAN)
550             info->end_transaction = next_commit_ID;
551         else {
554             if (info->end_transaction != next_commit_ID) {
558                 if (!success)
559                     success = -EIO;
560             }
561         }
562
563         return success;
567 }
```

4．恢复步骤 2：PASS_REVOKE

fs/jbd/recovery.c

```
312 static int do_one_pass(journal_t *journal,
313                         struct recovery_info *info, enum passtype pass)
314 {
315         unsigned int            first_commit_ID, next_commit_ID;
316         unsigned int            next_log_block;
317         int                     err, success = 0;
318         journal_superblock_t *  sb;
319         journal_header_t *      tmp;
320         struct buffer_head *    bh;
321         unsigned int            sequence;
322         int                     blocktype;
335         sb = journal->j_superblock;
336         next_commit_ID = be32_to_cpu(sb->s_sequence);

                // next_log_block 表示要在日志中读的下一个块的块号
337         next_log_block = be32_to_cpu(sb->s_start);
338
339         first_commit_ID = next_commit_ID;
352         while (1) {
                    // 遍历所有的日志块
353                 int             flags;
354                 char *          tagp;
355                 journal_block_tag_t *   tag;
356                 struct buffer_head *    obh;
357                 struct buffer_head *    nbh;

365                 if (pass != PASS_SCAN)
366                         if (tid_geq(next_commit_ID, info->end_transaction))
367                                 break;

                    // 读入日志中的下一个块
377                 err = jread(&bh, journal, next_log_block);
378                 if (err)
379                         goto failed;
380
381                 next_log_block++;
382                 wrap(journal, next_log_block);
383
390                 tmp = (journal_header_t *)bh->b_data;
391
392                 if (tmp->h_magic != cpu_to_be32(JFS_MAGIC_NUMBER)) {
                        // 如果该块不是日志的描述块，则说明已经处理完了，退出
393                         brelse(bh);
394                         break;
```

```
395             }
396
397             blocktype = be32_to_cpu(tmp->h_blocktype);
398             sequence = be32_to_cpu(tmp->h_sequence);
401
402             if (sequence != next_commit_ID) {
                    // 如果序号不对，退出
403                 brelse(bh);
404                 break;
405             }
406
410
411             switch(blocktype) {
412             case JFS_DESCRIPTOR_BLOCK:
                    // 描述符块
416                 if (pass != PASS_REPLAY) {
                        // 我们现在在 PASS_REVOKE 中，
                        // count_tags()函数会计算该描述符块中一共描述了对少个数据块的
                        对应关系，现在直接跳过数据块即可。
417                     next_log_block +=
418                         count_tags(bh, journal->j_blocksize);
419                     wrap(journal, next_log_block);
420                     brelse(bh);
421                     continue;
422                 }
509
510             case JFS_COMMIT_BLOCK:
                    // 提交块
514                 brelse(bh);
515                 next_commit_ID++;
516                 continue;
517
518             case JFS_REVOKE_BLOCK:
                    // 取消块
                    // 我们现在在 PASS_REVOKE 步骤中，
                    // 调用 scan_revoke_records 从取消块中读入数据，
                    // 构造内存中的 revoke hash table
526                 err = scan_revoke_records(journal, bh,
527                                           next_commit_ID, info);
528                 brelse(bh);
531                 continue;
532
533             default:
536                 brelse(bh);
```

```
537                     goto done;
538             }
539         }
540
541     done:
549         if (pass == PASS_SCAN)
550             info->end_transaction = next_commit_ID;
551         else {
554             if (info->end_transaction != next_commit_ID) {
558                 if (!success)
559                     success = -EIO;
560             }
561         }
562
563         return success;
567 }
```

scan_revoke_records()函数的主要作用是将一个取消块中的信息填入内存中的 revoke hash table。

fs/jbd/revoke.c

```
572 static int scan_revoke_records(journal_t *journal, struct buffer_head *bh,
573                                tid_t sequence, struct recovery_info *info)
574 {
575         journal_revoke_header_t *header;
576         int offset, max;
577
578         header = (journal_revoke_header_t *) bh->b_data;
579         offset = sizeof(journal_revoke_header_t);

            // r_count 保存的是本取消块的界限。
580         max = be32_to_cpu(header->r_count);
581
582         while (offset < max) {
583             unsigned int blocknr;
584             int err;

585             //  取得磁盘块号
586             blocknr = be32_to_cpu(* ((__be32 *) (bh->b_data+offset)));
587             offset += 4;

                //  将< blocknr, sequence >插入到 revoke hash table 表中
588             err = journal_set_revoke(journal, blocknr, sequence);
591             ++info->nr_revokes;
592         }
```

```
593         return 0;
594 }
```

journal_set_revoke()函数的逻辑是，如果 revoke hash table 中没有比< blocknr, sequence > 更新的记录，则插入到 hash 表中。

```
646 int journal_set_revoke(journal_t *journal,
647                          unsigned int blocknr,
648                          tid_t sequence)
649 {
650         struct jbd_revoke_record_s *record;
651
            // blocknr 在 hash 表中是否已经出现过？
652         record = find_revoke_record(journal, blocknr);
653         if (record) {
                // 出现过。则取最近的值，即 transaction ID 最大的。
656             if (tid_gt(sequence, record->sequence))
657                 record->sequence = sequence;
658             return 0;
659         }
            // 未出现过，插入之。
660         return insert_revoke_hash(journal, blocknr, sequence);
661 }
```

find_revoke_record()函数是在 hash 表中查找给定的块号。

```
168 static struct jbd_revoke_record_s *find_revoke_record(journal_t *journal,
169                                                        unsigned int blocknr)
170 {
171         struct list_head *hash_list;
172         struct jbd_revoke_record_s *record;
173
174         hash_list = &journal->j_revoke->hash_table[hash(journal, blocknr)];
175
176         spin_lock(&journal->j_revoke_lock);
177         record = (struct jbd_revoke_record_s *) hash_list->next;
178         while (&(record->hash) != hash_list) {
179             if (record->blocknr == blocknr) {
180                 spin_unlock(&journal->j_revoke_lock);
181                 return record;
182             }
183             record = (struct jbd_revoke_record_s *) record->hash.next;
184         }
185         spin_unlock(&journal->j_revoke_lock);
186         return NULL;
187 }
```

insert_revoke_hash()函数的作用是将<blocknr，seq>插入到 hash 表中。

```
139 static int insert_revoke_hash(journal_t *journal, unsigned int blocknr,
```

```
140                             tid_t seq)
141 {
142         struct list_head *hash_list;
143         struct jbd_revoke_record_s *record;
144
145 repeat:
            // 分配并设置一个 jbd_revoke_record_s 结构
146         record = kmem_cache_alloc(revoke_record_cache, GFP_NOFS);
147         if (!record)
148                 goto oom;
150         record->sequence = seq;
151         record->blocknr = blocknr;
            // 相应的 hash 链表头
152         hash_list = &journal->j_revoke->hash_table[hash(journal, blocknr)];
153         spin_lock(&journal->j_revoke_lock);
            // 插入链表。
154         list_add(&record->hash, hash_list);
155         spin_unlock(&journal->j_revoke_lock);
156         return 0;
157
158 oom:
163         goto repeat;
164 }
```

5．恢复步骤 3：PASS_REPLAY

开始动真格的了。日志机制终于要起作用了！前边一切一切的准备，只为了今天的 replay。

replay 是什么意思呢？就是系统意外崩溃时，有一部分元数据块已经写到日志中了，但是尚未写回到磁盘的原始位置上。replay 的作用就是将日志中的元数据块再重新写回到磁盘的原始位置上。

fs/jbd/recovery.c

```
312 static int do_one_pass(journal_t *journal,
313                         struct recovery_info *info, enum passtype pass)
314 {
315         unsigned int            first_commit_ID, next_commit_ID;
316         unsigned int            next_log_block;
317         int                     err, success = 0;
318         journal_superblock_t *  sb;
319         journal_header_t *      tmp;
320         struct buffer_head *        bh;
321         unsigned int            sequence;
322         int                     blocktype;

335         sb = journal->j_superblock;
```

```
336     next_commit_ID = be32_to_cpu(sb->s_sequence);

        // next_log_block 表示要在日志中读的下一个块的块号
337     next_log_block = be32_to_cpu(sb->s_start);
338
339     first_commit_ID = next_commit_ID;
352     while (1) {
            // 遍历所有的日志块
353         int             flags;
354         char *              tagp;
355         journal_block_tag_t *   tag;
356         struct buffer_head *    obh;
357         struct buffer_head *    nbh;
364
365         if (pass != PASS_SCAN)
366             if (tid_geq(next_commit_ID, info->end_transaction))
367                 break;

            // 读入日志中的下一个块
377         err = jread(&bh, journal, next_log_block);
381         next_log_block++;
382         wrap(journal, next_log_block);
390         tmp = (journal_header_t *)bh->b_data;
391
392         if (tmp->h_magic != cpu_to_be32(JFS_MAGIC_NUMBER)) {
                // 如果该块不是日志的描述块，则说明已经处理完了，退出
393             brelse(bh);
394             break;
395         }
396
397         blocktype = be32_to_cpu(tmp->h_blocktype);
398         sequence = be32_to_cpu(tmp->h_sequence);
402         if (sequence != next_commit_ID) {
                // 如果序号不对，退出
403             brelse(bh);
404             break;
405         }

            // 根据描述块的类型，进行相应的处理
411         switch(blocktype) {
412         case JFS_DESCRIPTOR_BLOCK:
                // 描述符块
428             tagp = &bh->b_data[sizeof(journal_header_t)];
429             while ((tagp - bh->b_data +sizeof(journal_block_tag_t))
```

```
430                 <= journal->j_blocksize) {
                    // 428-505 行，处理这个描述符块。
                    // 将其中的数据块从日志中读出来，
                    // 根据指示写回到磁盘的原始位置。
431                 unsigned int io_block;
432
                    // 从描述符块中取出一个 journal_block_tag_t 结构。
433                 tag = (journal_block_tag_t *) tagp;
434                 flags = be32_to_cpu(tag->t_flags);
435
                    // io_block 表示描述符块之后的日志中的一个数据块，
                    // 我们要把 io_block 中的数据写回到磁盘的原始位置。
436                 io_block = next_log_block++;
437                 wrap(journal, next_log_block);

                    // 读入 io_block 块的数据。
438                 err = jread(&obh, journal, io_block);
439                 if (err) {
442                     success = err;
447                 } else {
448                     unsigned int blocknr;

                        // 磁盘原始位置在哪里？
                        // 在 t_blocknr 中。
                        // 故 blocknr 表示日志中的该块对应的磁盘原始位置的块号
451                     blocknr = be32_to_cpu(tag->t_blocknr);

                        // 判断 blocknr 是否需要恢复，
                        // revoke 机制在这里起作用了！
456                     if (journal_test_revoke
457                         (journal, blocknr,
458                          next_commit_ID)) {
459                         brelse(obh);
460                         ++info->nr_revoke_hits;
461                         goto skip_write;
462                     }

                        // 读入 blocknr 对应的缓冲区
466                     nbh = __getblk(journal->j_fs_dev,
467                                    blocknr,
468                                    journal->j_blocksize);
478
479                     lock_buffer(nbh);
```

```
                    // obh 代表日志中的一块数据，
                    // nbh 代表磁盘原始位置的一块数据，
                    // 进行数据拷贝！
480                 memcpy(nbh->b_data, obh->b_data,
481                        journal->j_blocksize);
482                 if (flags & JFS_FLAG_ESCAPE) {
                        // 处理转义块。
483                     *((__be32 *)nbh->b_data) =
484                     cpu_to_be32(JFS_MAGIC_NUMBER);
485                 }
486
488                 set_buffer_uptodate(nbh);
489                 mark_buffer_dirty(nbh);
491                 ++info->nr_replays;
492                 /* ll_rw_block(WRITE, 1, &nbh); */
493                 unlock_buffer(nbh);
494                 brelse(obh);
495                 brelse(nbh);
496             }
497
498         skip_write:
                // 因为 UUID 的关系，描述符块中的记录可视为不定长的，
                // 所以这里要特殊处理一下。
499             tagp += sizeof(journal_block_tag_t);
500             if (!(flags & JFS_FLAG_SAME_UUID))
501                 tagp += 16;
502
503             if (flags & JFS_FLAG_LAST_TAG)
504                 break;
505         }
506
507         brelse(bh);
508         continue;
509
510     case JFS_COMMIT_BLOCK:
            // 提交块
514         brelse(bh);
515         next_commit_ID++;
516         continue;
517
518     case JFS_REVOKE_BLOCK:
            // 取消块
521         if (pass != PASS_REVOKE) {
522             brelse(bh);
```

```
523                         continue;
524                 }
532
533             default:
536                 brelse(bh);
537                 goto done;
538             }
539         }
540
541     done:
549         if (pass == PASS_SCAN)
550             info->end_transaction = next_commit_ID;
551         else {
554             if (info->end_transaction != next_commit_ID) {
558                 if (!success)
559                     success = -EIO;
560             }
561         }
562
563         return success;
567 }
```

恢复过后，云淡风轻！

下面解释一下一行无用的代码，即

```
492                             /* ll_rw_block(WRITE, 1, &nbh); */
```

这行代码可能以前是起作用的后来被注释掉了。我们注意到恢复时，480-489 行，磁盘原始位置上的磁盘块中的数据已经是正确的了，但是并未将缓冲区写回到磁盘中去。也就是说，磁盘中的数据和内存中缓冲区的数据仍是不一致的。这里可以像注释中那样，进行一次写操作，但是这其实也是没有必要的。因为如果文件系统后面访问该磁盘块，会从内存缓冲区中直接得到正确的也是最新的数据，至于磁盘，可以通过正常的缓冲区刷新策略择机进行写回。

6．恢复后的设置工作

journal_reset()函数是在恢复完成后调用的，主要是更新日志超级块中的数据，以及初始化 journal 中的数据。

fs/jbd/journal.c

```
881 static int journal_reset(journal_t *journal)
882 {
883         journal_superblock_t *sb = journal->j_superblock;
884         unsigned int first, last;
885
886         first = be32_to_cpu(sb->s_first);
887         last = be32_to_cpu(sb->s_maxlen);
888         if (first + JFS_MIN_JOURNAL_BLOCKS > last + 1) {
```

```
891             journal_fail_superblock(journal);
 892             return -EINVAL;
 893         }
 894
 895         journal->j_first = first;
 896         journal->j_last = last;
 897
 898         journal->j_head = first;
 899         journal->j_tail = first;
 900         journal->j_free = last - first;
 901
 902         journal->j_tail_sequence = journal->j_transaction_sequence;
 903         journal->j_commit_sequence = journal->j_transaction_sequence - 1;
 904         journal->j_commit_request = journal->j_commit_sequence;
 905
 906         journal->j_max_transaction_buffers = journal->j_maxlen / 4;
 907
             // 更新磁盘上日志中的超级块的信息
 909         journal_update_superblock(journal, 1);

             // 启动 kjournald 内核线程，我们现在可以接受原子操作了。
 910         return journal_start_thread(journal);
 911 }
```

journal_update_superblock()函数的作用是更新日志超级块中的数据，然后同步地写回到磁盘中。

```
 992 void journal_update_superblock(journal_t *journal, int wait)
 993 {
 994         journal_superblock_t *sb = journal->j_superblock;
 995         struct buffer_head *bh = journal->j_sb_buffer;
1012
1013         spin_lock(&journal->j_state_lock);
1016
             // 更新日志超级块中的数据
1017         sb->s_sequence = cpu_to_be32(journal->j_tail_sequence);
1018         sb->s_start        = cpu_to_be32(journal->j_tail);
1019         sb->s_errno        = cpu_to_be32(journal->j_errno);
1020         spin_unlock(&journal->j_state_lock);
1021
1023         mark_buffer_dirty(bh);
             // 注意此时 wait==1，同步地将日志超级块写到磁盘上。
1024         if (wait)
1025             sync_dirty_buffer(bh);
1026         else
1027             ll_rw_block(SWRITE, 1, &bh);
```

1040 }

你可能还会有一个问题，如果恰好恢复时系统又崩溃了，那么正在进行的恢复操作又被打断了，那么会产生什么问题么？答案是不会的。因为如果恢复操作被打断了，journal_reset()函数就不会被调用了，于是日志中超级块的信息与恢复前仍一样。再次挂载该文件系统时，会又一次根据日志超级块中的信息在进行一遍恢复操作。

十四、 参考资料

1. http://msdn.microsoft.com/zh-cn/library/ms190612.aspx
2. http://www.ibm.com/developerworks/cn/linux/l-jfs/
3. Stephen C. Tweedie, 《Journaling the Linux ext2fs Filesystem》
4. Stephen C. Tweedie, 《EXT3, Journaling Filesystem》
5. Mingming Cao, 《State of the Art: Where we are with Ext3 filesystem》
6. Amey Inandar, Kedar Sovani, 《Linux: The Journaling Block Device》
7. 王旭，《Linux: The Journaling Block Device》
8. Rey Card，《Design and Implementation of the Second Extended Filesystem》
9. Dave Poirier,《The Second Extended File System Internal Layout》
10. 毛德操，《Linux 内核源代码情景分析》第五章
11. Linux 内核源代码 2.6.35