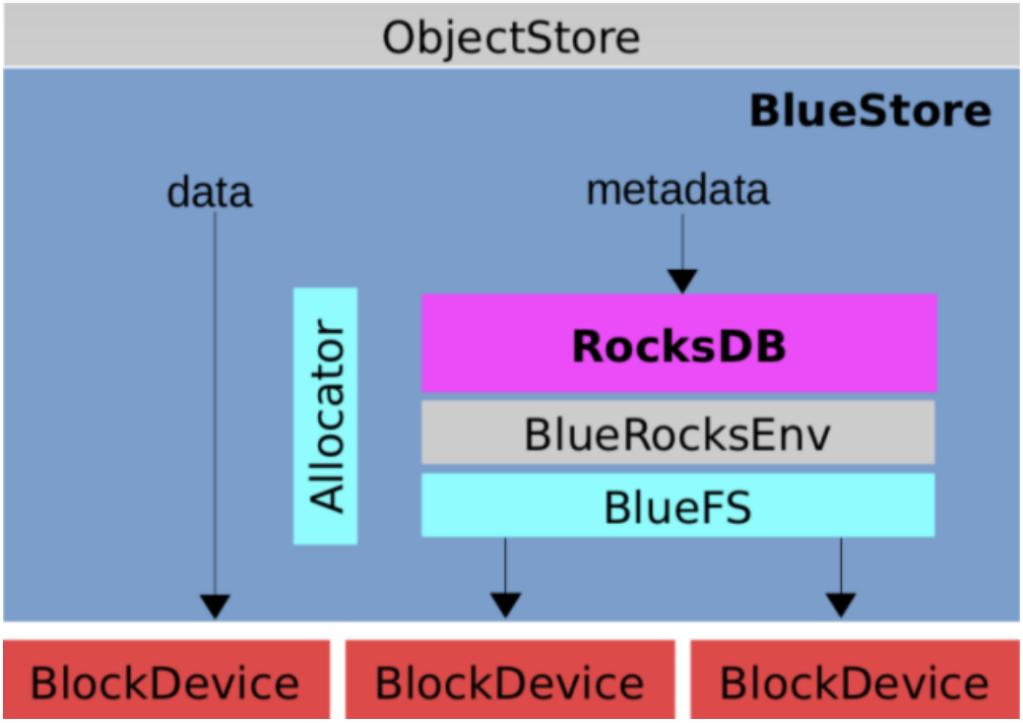
BlueStore

ceph版本：https://github.com/ceph/ceph/tree/v18.2.1

参考文章

1. Ceph存储引擎BlueStore简析
2. BlueStore-先进的用户态文件系统《一》
3. BlueStore-先进的用户态文件系统《二》-BlueFS
4. BlueStore源码分析之架构设计
5. BlueStore & BlueFS & rocksdb 关联性梳理
6. BlueStore写流程事务实现梳理
7. Ceph BlueStore
8. ceph bluestore 写操作源码分析(上)
9. 一张图讲清楚BuleStore的对象存储
10. BlueStore源码分析之对象IO
11. BlueStore 架构及原理分析
12. BlueStore源码分析之事务状态机
13. ceph bluestore源码分析:非对齐写逻辑

设计理念



BlockDevice：物理块设备，使用 Libaio 操作裸设备，AsyncIO。

RocksDB：存储 WAL 、对象元数据、对象扩展属性 Omap、磁盘分配器元数据。

BlueRocksEnv：抛弃了传统文件系统，封装 RocksDB 文件操作的接口。

BlueFS：小型的 Append 文件系统，实现了 RocksDB::Env 接口，给 RocksDB 用。

Allocator：磁盘分配器，负责高效的分配磁盘空间。

BlueStore 把元数据和对象数据分开写，对象数据直接写入硬盘，而元数据则先写入超级高速的内存数据库，后续再写入稳定的硬盘设备，这个写入过程由 BlueFS 来控制。

在所有的存储系统中，读操作一般都是同步的（前端发出读指令，后端必须返回待读取的数据后，才算读成功）。写操作则可以是异步的（前端发出写指令，后端先接收数据后，直接返回成功写，后续再慢慢把数据写到磁盘中），一般为了性能考虑，所有写操作都会先写内存缓存Page-Cache便返回客户端成功，然后由文件系统批量刷新。但是内存是易失性存储介质，掉电后数据便会丢失，所以为了数据可靠性，我们不能这么做。

通常，将数据先写入性能更好的非易失性存储介质(SSD、NVME等)充当的中间设备，然后再将数据写入内存缓存，便可以直接返回客户端成功，等到数据写入到普通磁盘的时候再释放中间设备对应的空间。

传统文件系统中将数据先写入高速盘，再同步到低速盘的过程叫做“双写”，高速盘被称为日志盘，低速盘被称为数据盘。每个写操作实际都要写两遍，这大大影响了效率，同时浪费了存储空间。

COW(Copy-On-Write)：当覆盖写发生时，不是更新磁盘对应位置已有的内容，而是新分配一块空间，写入本次更新的内容，然后更新对应的地址指针，最后释放原有数据对应的磁盘空间。这样就只需要写一次数据，并能原子性的更新，但是也带来了其他的问题：一是COW机制破坏了数据在磁盘分布的

物理连续性。经过多次COW后，读数据的顺序读将会便会随机读。二是针对小于块大小的覆盖写采用COW会得不偿失。是因为：一是将新的内容写入新的块后，原有的块仍然保留部分有效内容，不能释放无效空间，而且再次读的时候需要将两个块读出来Merge操作，才能返回最终需要的数据，将大大影响读性能。二是存储系统一般元数据越多，功能越丰富，元数据越少，功能越简单。而且任何操作必然涉及元数据，所以元数据是系统中的热点数据。COW涉及空间重分配和地址重定向，将会引入更多的元数据，进而导致系统元数据无法全部缓存在内存里面，性能会大打折扣。

RMW(Read-Modify-Write)：指当覆盖写发生时，如果本次改写的内容不足一个BlockSize，那么需要先将对应的块读上来，然后在内存中将原内容和待修改内容合并Merge，最后将新的块写到原来的位置。但是RMW也带来了两个问题：一是需要额外的读开销；二是RMW不是原子操作，如果磁盘中途掉电，会有数据损坏的风险。为此我们需要引入日志（Journal），先将待更新数据写入Journal，然后再更新数据。而这个过程称为 **WAL**(Write-Ahead Logging)。

BlueStore 的写策略综合运用直接写、COW 和 RMW 策略。

非覆盖写直接分配空间写入即可

块大小对齐的覆盖写采用 COW 策略；小于块大小的覆盖写采用 RMW 策略。

BlockSize：定义了底层块设备（如硬盘或SSD）的基本块大小，通常是4KB，磁盘IO操作的最小单元(原子操作)。即读写的数据就算少于 BlockSize，磁盘I/O的大小也是 BlockSize，是原子操作，要么写入成功，要么写入失败，即使掉电不会存在部分写入的情况。

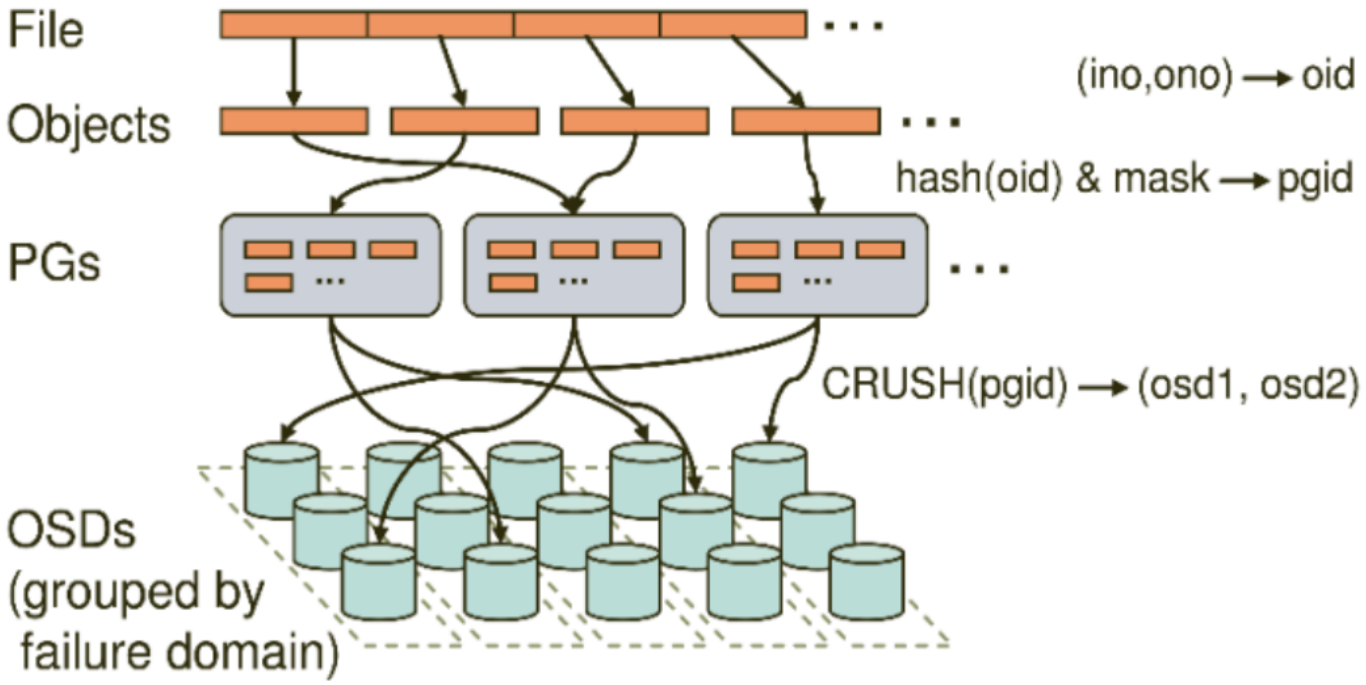
min_alloc_size：定义了BlueStore在分配空间时使用的最小分配单元的大小。这是在处理写入请求时，BlueStore会考虑的最小数据块大小。如果写入的数据小于这个值，BlueStore会尝试找到一个已经存在的数据块（blob）来复用，以避免创建新的数据块。这样可以提高存储空间的利用率，减少碎片。在SSD上，默认的 `min_alloc_size` 通常是16KB，而在HDD上，默认值可能是64KB。

稀疏写 Linux文件空洞与稀疏文件

我们知道一个文件的逻辑空间上是连续的，但是真正在磁盘上的物理空间分布并不一定是连续的。同时我们也会使用 `lseek` 系统调用，使得文件偏移量大于文件的长度，此时再对文件写入，便会在文件中形成一个空洞，这些空洞中的字节都是0。空洞是否占用磁盘空间是由文件系统决定的，不过大部分的文件系统 `ext4` 、 `xfs` 都不占磁盘空间。我们称部分数据是0同时这部分数据不占磁盘空间的文件为稀疏文件，这种写入方式为稀疏写。

BlueStore中的对象也支持稀疏写，同时支持克隆等功能，所以对象的数据组织结构设计的会相对复杂一点。

相关数据结构

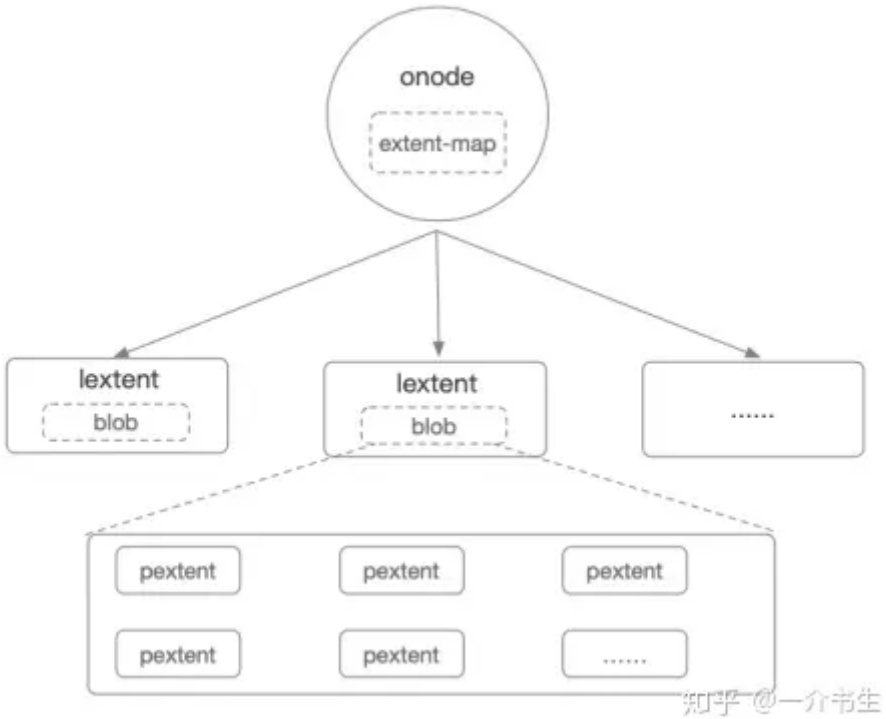


名词解释

- `Collection`：PG在内存的数据结构。
- `bluestore_cnode_t`：PG在磁盘的数据结构。
- `Onode`：对象在内存的数据结构。
- `Extent`：对象的一个逻辑空间(lextent)。
- `extent_map_t`：一个对象包含的多段逻辑空间映射。
- `bluestore_onode_t`：对象在磁盘的数据结构。
- `bluestore_pextent_t`：一段连续物理空间。
- `bluestore_blob_t` 一片不一定连续的磁盘物理空间，包含多段pextent。
- `Blob`：包含一个 `bluestore_blob_t` 、用于引用计数、共享blob等信息。

Onode

BlueStore的对象内部结构如下图：



BlueStore的每个对象对应一个Onode结构体，每个Onode包含一张extent-map，extent-map映射多个lextent，每个lextent负责管理对象内的一个逻辑段数据，并且关联一个Blob，Blob包含多个pextent，最终将对象的数据映射到磁盘上。

PExtent

每段pextent对应一段连续的磁盘物理空间，结构体为 `bluestore_pextent_t` 。

```
struct bluestore_pextent_t {
    uint64_t offset = 0; // 磁盘上的物理偏移
    uint32_t length = 0; // 数据段的长度
}
```

pextent的offset和length都是块大小对齐的。

Blob

Blob包含磁盘上物理段的集合，即 `bluestore_pextent_t` 的集合。

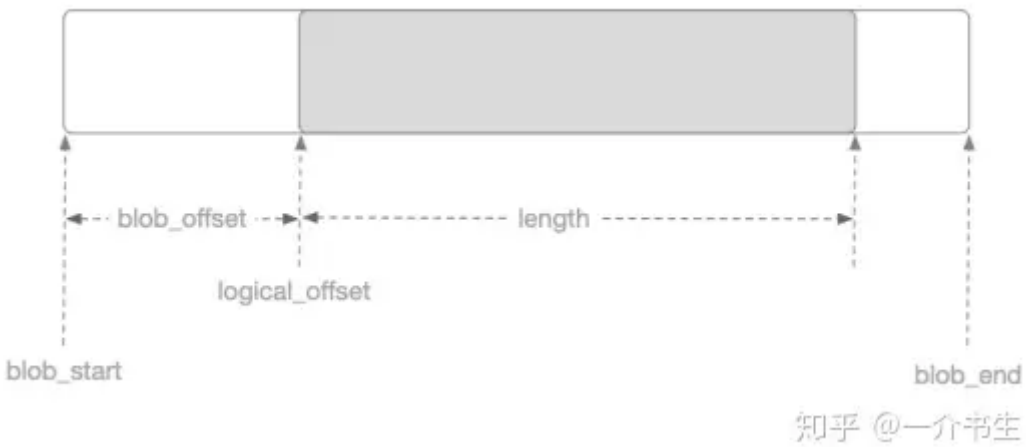
```
struct Blob {
    mutable bluestore_blob_t blob;
}
struct bluestore_blob_t {
    PExtentVector extents;
}
```

Extent

Extent是对象内的基本数据管理单元，数据压缩、数据校验、数据共享等功能都是基于Extent粒度实现的。这里的Extent是对象内的，并不是磁盘内的，所以我们称为lextent，和磁盘内的pextent以示区分。

```
struct Extent {
    uint32_t logical_offset = 0; // 对象内逻辑偏移
    uint32_t length = 0; // 逻辑段长度
    // 当logical_offset是块对齐时，blob_offset始终为0;
    // 不是块对齐时，将逻辑段内的数据通过Blob映射到磁盘物理段会产生物理段内的偏移称为blob_offset
    uint32_t blob_offset = 0;
}
```

logical_offset、length、blob_offset关系如下图：



WriteContext

bluestore 写操作上下文

```
struct WriteContext {
    ......
    // 写条目
    struct write_item {
        uint64_t logical_offset;  ///< write logical offset
        BlobRef b;
        uint64_t blob_length;
        uint64_t b_off;
        bufferlist bl;
```

```
        uint64_t b_off0;   ///< original offset in a blob prior to padding
        uint64_t length0;  ///< original data length prior to padding
    }

    vector<write_item> writes;

    // bluestore大小写都会调用，向writes这个vector里面插入一条write_item。
    void write(uint64_t loffs, BlobRef b, uint64_t blob_len, uint64_t o,
                    bufferlist &bl, uint64_t o0, uint64_t len0,
                    bool _mark_unused, bool _new_blob) {
            writes.emplace_back(loffs, b, blob_len, o, bl, o0, len0,
                                _mark_unused, _new_blob);
    }
}
```

对齐写

磁盘的最小分配单元是 `min_alloc_size` ，HDD默认64K，SSD默认16K。

对齐到 `min_alloc_size` 的写我们称为大写(big-write)，在处理中会根据实际大小生成lextent、blob，lextent包含的区域是 `min_alloc_size` 的整数倍。

非对齐写

落在 `min_alloc_size` 区间内的写我们称为小写(small-write)。因为最小分配单元 `min_alloc_size` 的限制，如果一个IO那么只会占用到blob的一部分，剩余的空间还可以存放其他的数据。所以小写会先根据offset查找有没有可复用的blob，如果没有则生成新的blob。

AioContext

封装了libaio，写设备都是通过libaio，首先需要了解回调函数的执行流程。

AioContext派生了两种context，TransContext和DeferredBatch，前者对应simple write，简称为txc，后者对应deferred write，简称为dbh。txc处理COW场景的写，dbh处理RMW场景的写。

创建块设备的时候，会设置好回调函数，由块设备的aio thread线程执行回调:

```
struct AioContext {
        virtual void aio_finish(BlueStore *store) = 0;
        virtual ~AioContext() {}
};

struct TransContext : public AioContext {
        ......
```

```
        void aio_finish(BlueStore *store) override {
                store->txc_aio_finish(this); // txc的回调
        }
}

struct DeferredBatch : public AioContext {
        ......
        void aio_finish(BlueStore *store) override {
                store->_deferred_aio_finish(osr); // dbh的回调
        }
}
```

回调函数在创建设备的时候，会提前设置好:

```
int BlueStore::_open_bdev(bool create)
{
        assert(bdev == NULL);
        string p = path + "/block";
        bdev = BlockDevice::create(cct, p, aio_cb, static_cast<void*>(this));
// 传入回调函数
        ......
}

static void aio_cb(void *priv, void *priv2)
{
        BlueStore *store = static_cast<BlueStore*>(priv);
        BlueStore::AioContext *c = static_cast<BlueStore::AioContext*>(priv2);
        c->aio_finish(store); // 执行回调函数
}
```

准备数据

```
int BlueStore::queue_transactions(CollectionHandle &ch, vector &tls,
TrackedOpRef op, ThreadPool::TPHandle *handle) {
    // 获取Collection对象和OpSequencer对象
    Collection *c = static_cast(ch.get());
    OpSequencer *osr = c->osr.get();

    // 创建事务上下文
    TransContext *txc = _txc_create(c, osr, nullptr, op);

    // 将OSD层的事务转为BlueStore层的事务
```

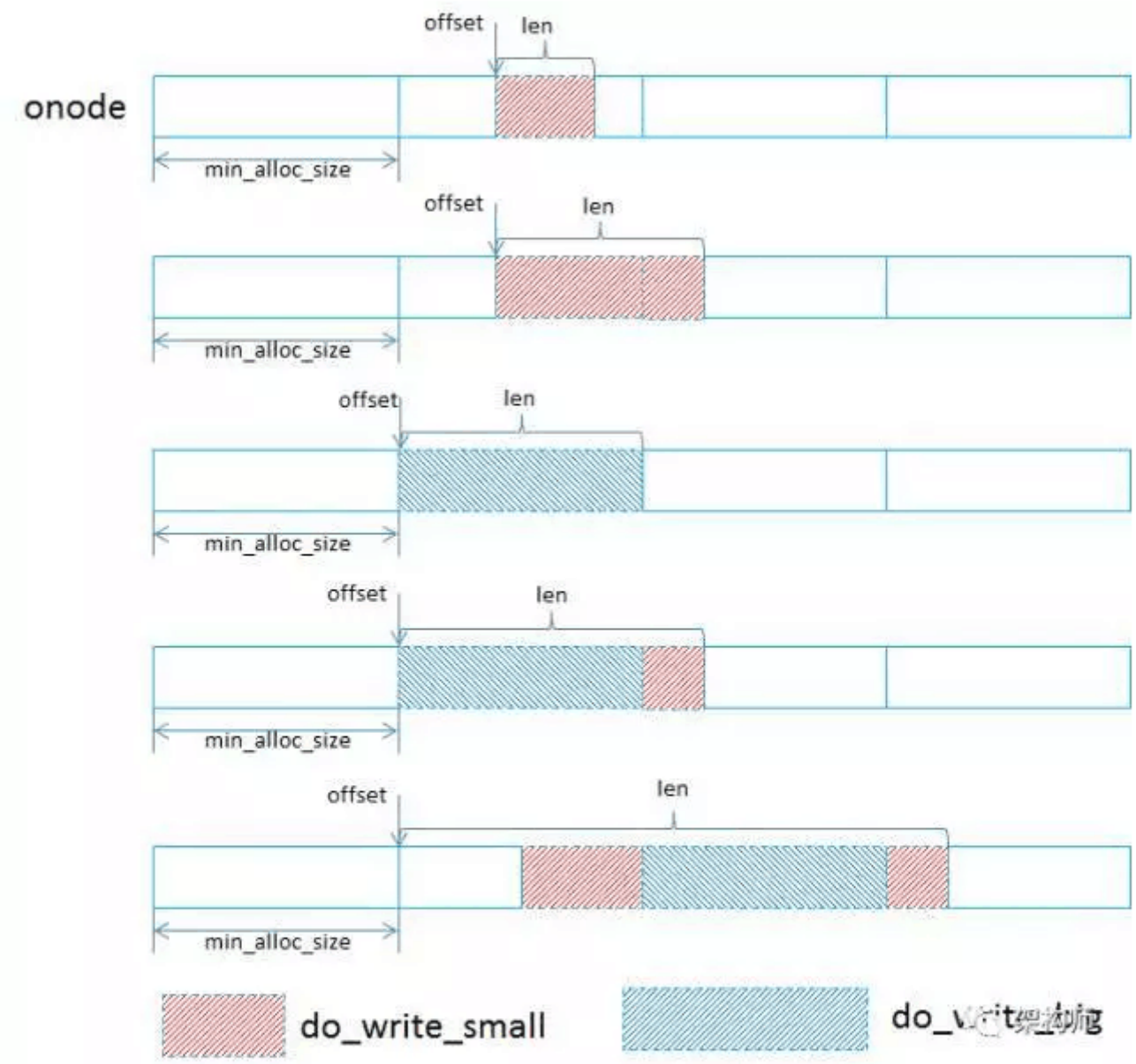
```
    for (auto &tx : tls) {
        txc->bytes += tx.get_num_bytes();
        _txc_add_transaction(txc, &tx);
    }
    _txc_calc_cost(txc); // 计算事务的“花费”

    // 最终化KV事务
    _txc_finalize_kv(txc, txc->t);

    // 尝试开始事务
    auto tstart = mono_clock::now();
    throttle.try_start_transaction(*db, *txc, tstart);

    // 执行事务
    _txc_state_proc(txc);

    // 处理同步事务
    for (auto c : on_applied_sync) {
        c->complete(0);
    }
    return 0;
}
```

BlueStore::_txc_add_transaction ---> BlueStore::_write ---> BlueStore::_do_write

```
_do_write_data(txc, c, o, offset, length, bl, &wctx);
r = _do_alloc_write(txc, c, o, &wctx);
```

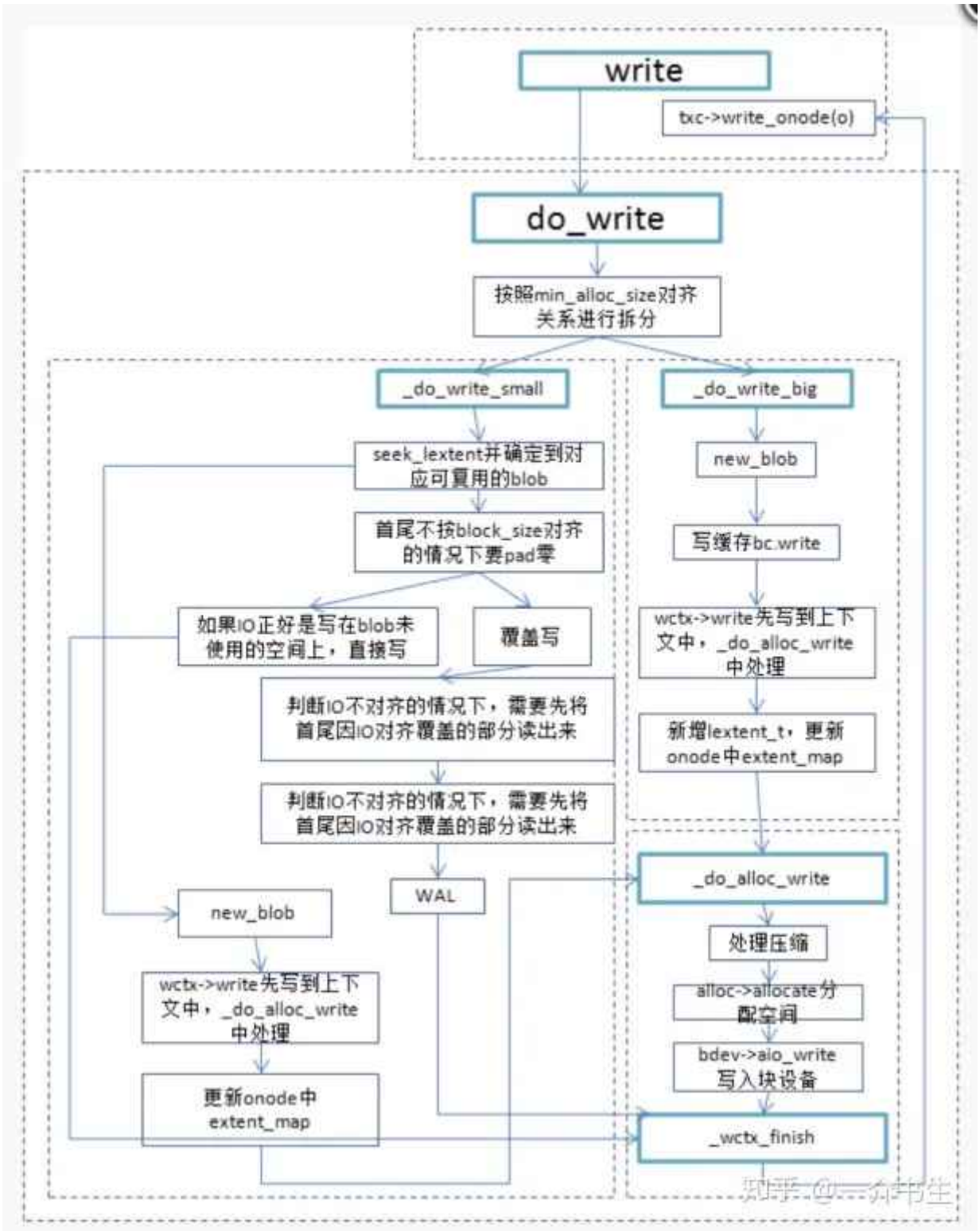
_do_write_data

min_alloc_size为block size大小的整数倍，如果要写的数据跨越一个min_alloc_size，则会把数据按照min_alloc_size划分，如下图所示



当一个写请求按照min_alloc_size进行拆分后，就会分为对齐写，对应到do_write_big，非对齐写（即落到某一个min_alloc_size区间的写I/O（对应到do_write_small）。

_do_write_data 负责安排Blob，并将数据写入Blob，如下图



ceph bluestore 写操作源码分析(上)

_do_alloc_write

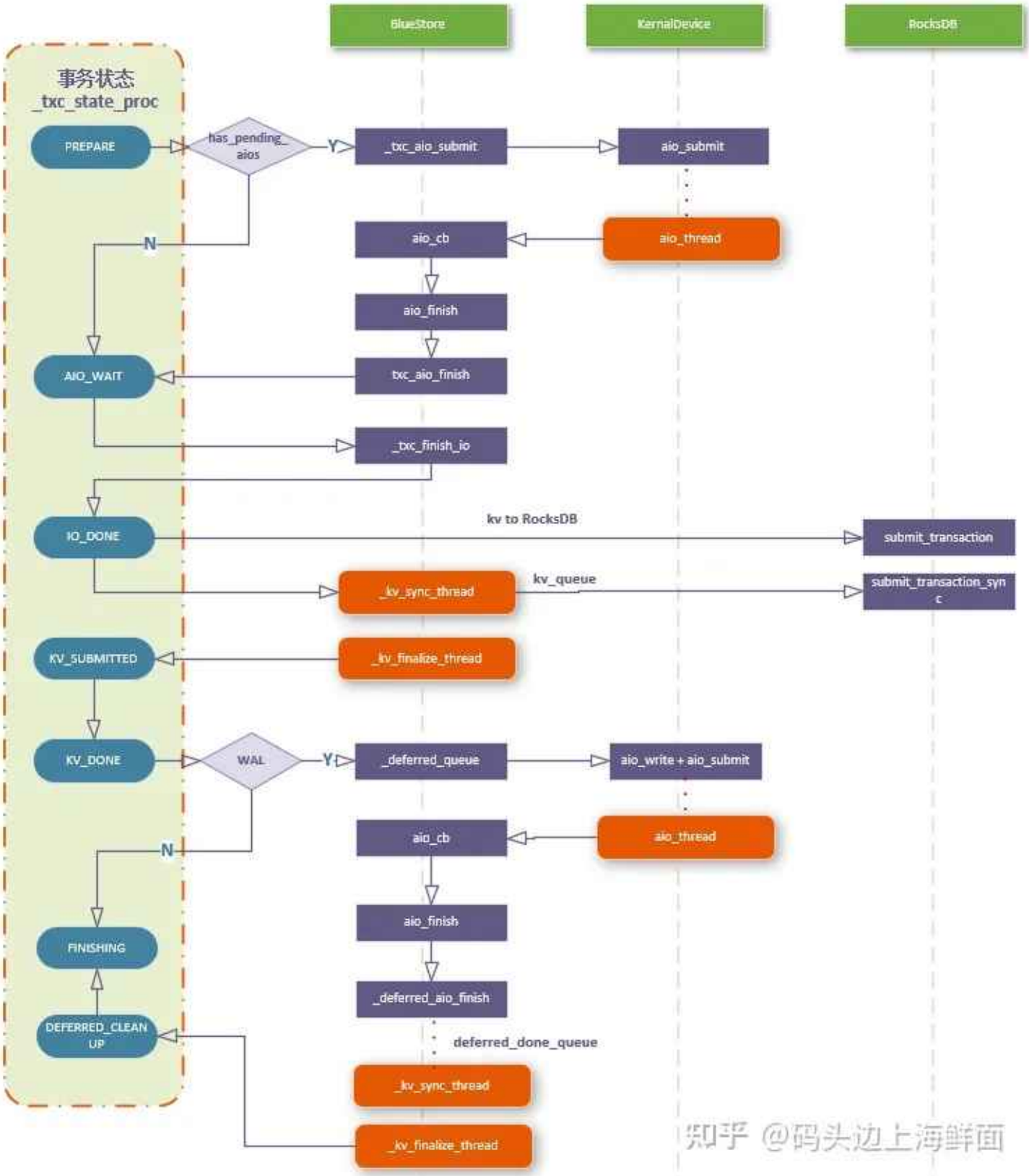
负责分配Blob的空间，_do_write_data选出了Blob，现在为Blob分配实际的物理空间。

并将要写入的数据放在pending_aios中，等待写入磁盘

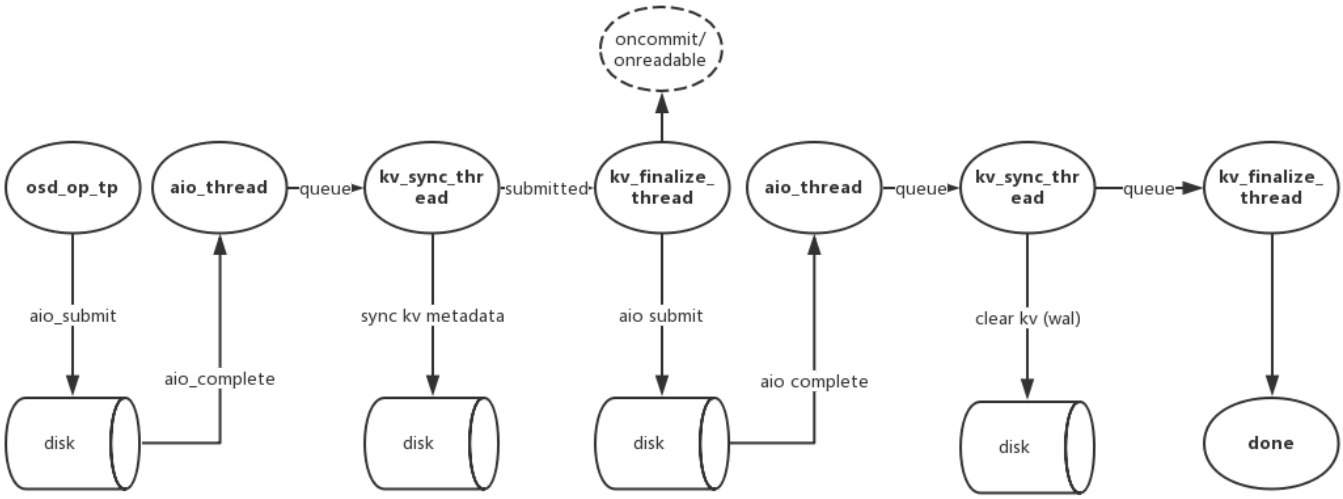
```
1 BlueStore::_do_alloc_write
2 >> bdev->aio_write(offset, t, &txc->ioc, false);
3
```

```
4 KernelDevice::aio_write
5 >> ioc->pending_aios.push_back(aio_t(ioc, choose_fd(false, write_hint)));
```

写数据



知乎 @码头边上海鲜面



1. **线程池（osd_op_tp）：**
   - 设置事务状态为 `STATE_PREPARE`，表示事务准备阶段。
   - 设置状态为 `STATE_AIO_WAIT`，表示事务正在等待异步I/O操作完成。
2. **异步I/O线程（aio_thread）：**
   - 在异步I/O操作完成后，回调处理将事务状态设置为 `STATE_IO_DONE`，表示I/O操作已完成。
   - 然后设置状态为 `STATE_KV_QUEUED`，表示事务已准备好进入键值对（KV）队列等待进一步处理。
3. **KV同步线程（kv_sync_thread）：**
   - 提交KV日志和部分元信息，将事务状态设置为 `STATE_KV_SUBMITTED`，表示事务已提交到KV存储。
   - 然后，事务状态被设置为 `STATE_KV_DONE`，表示KV操作已完成。
4. **KV最终化线程（kv_finalize_thread）：**
   - 生成写入数据的数据库句柄（dbh），并提交IO请求，等待回调。
   - 设置状态为 `STATE_DEFERRED_QUEUE`，表示事务已排队等待最终处理。
   - 在IO请求完成后，设置状态为 `STATE_FINISHING`，表示事务正在完成阶段。
   - 最后，设置状态为 `STATE_DONE`，表示事务已完成。
5. **异步I/O线程（aio_thread）回调处理：**
   - 在IO请求完成后，事务状态被设置为 `STATE_DEFERRED_CLEANUP`，表示事务正在清理阶段。
6. **KV同步线程（kv_sync_thread）清理：**
   - 在事务状态为 `STATE_KV_DONE` 后，清理延迟KV的键值对。
7. **kv_finalize_thread：**

    - 设置状态 `STATE_FINISHING` 和 `STATE_DONE`
- OSD::osd_op_tp: 通过libaio的方式，提交io请求给KernelDevice
- KernelDevice::aio_thread: 执行libaio完成后的回调
- BlueStore::kv_sync_thread: 同步kv数据，包括对象的meta信息和磁盘空间使用信息；以及wal日志的清理
- BlueStore::kv_finalize_thread: 完成时回调的处理以及其他清理工作。wal情况生成dbh以及提交io请求
- BlueStore::deferred_finisher: 通过libaio的方式，提交deferred io的请求
- BlueStore::finishers: finisher线程的sharding，用来回调通知用户请求完成

开始事务

```
BlueStore::_txc_state_proc
>> case TransContext::STATE_PREPARE:
      txc->set_state(TransContext::STATE_AIO_WAIT);
      txc->had_ios = true;
      _txc_aio_submit(txc);
      return;

BlueStore::_txc_aio_submit
>> bdev->aio_submit(&txc->ioc);


KernelDevice::aio_submit
>> ioc->running_aios.splice(e, ioc->pending_aios);// 移动操作
>> r = io_queue->submit_batch(ioc->running_aios.begin(), e, pending, priv,
&retries);

// 函数原型:
// int aio_queue_t::submit_batch(aio_iter begin, aio_iter end,
//               uint16_t aios_size, void *priv, int *retries)
// 至此数据已经提交给内核了，接下来等待数据写入完成
```

aio_thread

负责处理完成数据写入后的回调

```
struct AioCompletionThread : public Thread {
```

```
    KernelDevice *bdev;
    explicit AioCompletionThread(KernelDevice *b) : bdev(b) {}
    void *entry() override {
        bdev->_aio_thread();
        return NULL;
    }
} aio_thread;

void KernelDevice::_aio_thread() {
    while (!aio_stop) {
        // 获取已完成的IO
        int r = io_queue->get_next_completed(cct->_conf->bdev_aio_poll_ms,
aio, max);
        // 调用回调函数
        aio_callback(aio_callback_priv, ioc->priv);
    }
}

static void aio_cb(void *priv, void *priv2) {
    BlueStore *store = static_cast(priv);
    BlueStore::AioContext *c = static_cast(priv2);
    c->aio_finish(store);
}

void aio_finish(BlueStore *store) override {
    store->txc_aio_finish(this);
}

void txc_aio_finish(void *p) {
    _txc_state_proc(static_castntext*>(p));
}
```

_txc_finish_io

控制事务执行顺序

```
case TransContext::STATE_AIO_WAIT:
    _txc_finish_io(txc);
    return;

void BlueStore::_txc_finish_io(TransContext *txc) {
    /*
     *我们需要保持kv事务的顺序，即使aio将以任何顺序完成。
```

```
     */

    OpSequencer *osr = txc->osr.get();
    std::lock_guard l(osr->qlock);
    txc->set_state(TransContext::STATE_IO_DONE);

    txc->ioc.release_running_aios();
    // 定位到当前txc在队列中的位置
    OpSequencer::q_list_t::iterator p = osr->q.iterator_to(*txc);
    while (p != osr->q.begin()) {
        --p;
        // 前面还有未完成io操作的txc，这个txc不能继续进行下去，等待前面的完成，所以直接
return
        if (p->get_state() < TransContext::STATE_IO_DONE) {
            return;
        }
        // 前面的txc进入下一个状态了，递增p并退出循环，下面处理当前的txc
        if (p->get_state() > TransContext::STATE_IO_DONE) {
            ++p;
            break;
        }
    }
    // 依次处理状态为STATE_IO_DONE的txc
    // 会将txc放入kv_sync_thread的队列kv_queue和kv_queue_unsubmitted
    do {
        _txc_state_proc(&*p++);
    } while (p != osr->q.end() && p->get_state() ==
TransContext::STATE_IO_DONE);

    if (osr->kv_submitted_waiters) {
        osr->qcond.notify_all();
    }
}
```

同步KV数据

```
case TransContext::STATE_IO_DONE:

    txc->set_state(TransContext::STATE_KV_QUEUED);

    _txc_apply_kv(txc, true);
    >> txc->set_state(TransContext::STATE_KV_SUBMITTED);

```

```
    kv_queue.push_back(txc);
    kv_cond.notify_one();

    return;
```

```
struct KVSyncThread : public Thread {
    BlueStore *store;
    explicit KVSyncThread(BlueStore *s) : store(s) {}
    void *entry() override {
        store->_kv_sync_thread();
        return NULL;
    }
};

void BlueStore::_kv_sync_thread() {
    kv_committing_to_finalize.swap(kv_committing);

    deferred_stable_to_finalize.swap(deferred_stable);

    kv_finalize_cond.notify_one();
}
```

将KV数据持久化

```
struct KVFinalizeThread : public Thread {
    BlueStore *store;
    explicit KVFinalizeThread(BlueStore *s) : store(s) {}
    void *entry() override {
        store->_kv_finalize_thread();
        return NULL;
    }
};

void BlueStore::_kv_finalize_thread(){
    kv_committed.swap(kv_committing_to_finalize);
    while (!kv_committed.empty()) {
        TransContext *txc = kv_committed.front();
        _txc_state_proc(txc); // 处理提交的事务
        kv_committed.pop_front();
    }

```

```
    for (auto b : deferred_stable) {
        auto p = b->txcs.begin();
        while (p != b->txcs.end()) {
            TransContext *txc = &*p;
            p = b->txcs.erase(p); // unlink here because
            _txc_state_proc(txc); // 处理延迟事务
        }
        delete b;
    }
}
```

第一个_txc_state_proc，提交一个回调

```
case TransContext::STATE_KV_SUBMITTED:
    _txc_committed_kv(txc);

void BlueStore::_txc_committed_kv(TransContext *txc) {
    txc->set_state(TransContext::STATE_KV_DONE);
    finisher.queue(txc->oncommits);
}

void queue(std::list<Context *> &ls) {
    {
        std::unique_lock ul(finisher_lock);
        if (finisher_queue.empty()) {
            finisher_cond.notify_all();
        }
        for (auto i : ls) {
            finisher_queue.push_back(std::make_pair(i, 0));
        }
        if (logger)
            logger->inc(l_finisher_queue_len, ls.size());
    }
    ls.clear();
}

struct FinisherThread : public Thread {
    Finisher *fin;
    explicit FinisherThread(Finisher *f) : fin(f) {}
    void *entry() override { return fin->finisher_thread_entry(); }
} finisher_thread;

void *Finisher::finisher_thread_entry()
{
    in_progress_queue.swap(finisher_queue);
```

```
    for (auto p : in_progress_queue) {
        p.first->complete(p.second); // 执行回调返回OSD层
    }
}


```

```
case TransContext::STATE_KV_DONE:
    throttle.log_state_latency(*txc, logger, l_bluestore_state_kv_done_lat);
    if (txc->deferred_txn) {
        txc->set_state(TransContext::STATE_DEFERRED_QUEUED);
        _deferred_queue(txc);
        return;
    }
    // 如果没有延迟事务，流程就基本结束了
    txc->set_state(TransContext::STATE_FINISHING);
    break;

void BlueStore::_deferred_queue(TransContext *txc) {
    _deferred_submit_unlock(txc->osr.get());
}

void BlueStore::_deferred_submit_unlock(OpSequencer *osr) {
    int r = bdev->aio_write(start, bl, &b->ioc, false);
    bdev->aio_submit(&b->ioc);
}

void KernelDevice::aio_submit(IOContext *ioc) {
    r = io_queue->submit_batch(ioc->running_aios.begin(), e, pending, priv,
&retries);
}
```

aio_thread

上面处理完延迟写，写完之后又由aio_thread处理回调，过程和之前一致，不同的是回调函数变了

```
void aio_finish(BlueStore *store) override {
    store->_deferred_aio_finish(osr);
}
```

```
4
5  void BlueStore::_deferred_aio_finish(OpSequencer *osr) {
6      DeferredBatch *b = osr->deferred_running;
7      for (auto &i : b->txcs) {
8          txc->set_state(TransContext::STATE_DEFERRED_CLEANUP);
9      }
10     deferred_done_queue.emplace_back(b);
11     kv_cond.notify_one();
12 }
13
14 void BlueStore::_kv_sync_thread()
15 >> 同上文，通过notify线程kv_finalize_thread，进而调用_txc_state_proc
16
17 case TransContext::STATE_DEFERRED_CLEANUP:
18     txc->set_state(TransContext::STATE_FINISHING);
19
```

清理资源

```
1  case TransContext::STATE_FINISHING:
2      throttle.log_state_latency(*txc, logger, l_bluestore_state_finishing_lat);
3      _txc_finish(txc);
4      return;
5
6  void BlueStore::_txc_finish(TransContext *txc) {
7      OpSequencerRef osr = txc->osr;
8      bool empty = false;
9      bool submit_deferred = false;
10     OpSequencer::q_list_t releasing_txc;
11     {
12         std::lock_guard l(osr->qlock);
13         txc->set_state(TransContext::STATE_DONE);
14         bool notify = false;
15         while (!osr->q.empty()) {
16             osr->q.pop_front();
17             releasing_txc.push_back(*txc);
18         }
19     }
20     while (!releasing_txc.empty()) {
21         auto txc = &releasing_txc.front();
22         _txc_release_alloc(txc);
23         releasing_txc.pop_front();
24         throttle.log_state_latency(*txc, logger, l_bluestore_state_done_lat);
```

```
        throttle.complete(*txc);
        delete txc;
    }
}

```

读数据

```
BlueStore::read()
>> BlueStore::_do_read()

```