**BlueStore源码分析-写流程**

本篇我们分析写流程.

**核心数据结构**

**OpSequencer**

是一个FIFO的队列，用来保证操作的有序性，在bluestore中有两个该实例，分别用于普通写和带日志的写，前者没什么特别说明的，就是通过FIFO的特性保证顺序性，后者因为只有当写日志完成之后，才可以开始覆盖写数据，而且由于后一个阶段是异步的，所以需要一个额外的队列用于将所有已经进入覆盖写阶段，带日志的写请求在次进行排序。

1. *class* OpSequencer : *public* Sequencer\_impl {
2. *public:*
3. std::mutex qlock;
4. std::condition\_variable qcond;
5. typedef boost::intrusive::list<
6. TransContext,
7. boost::intrusive::member\_hook<
8. TransContext,
9. boost::intrusive::list\_member\_hook<>,
10. &TransContext::sequencer\_item> > *q\_list\_t*;
11. *q\_list\_t* q; ///< transactions
12. boost::intrusive::list\_member\_hook<> deferred\_osr\_queue\_item;
13. DeferredBatch \*deferred\_running = nullptr;
14. DeferredBatch \*deferred\_pending = nullptr;
15. Sequencer \*parent;
16. BlueStore \*store;
17. *uint64\_t* last\_seq = 0;
18. std::atomic\_int txc\_with\_unstable\_io = {0}; ///< num txcs with unstable io
19. std::atomic\_int kv\_committing\_serially = {0};
20. std::atomic\_int kv\_submitted\_waiters = {0};
21. std::atomic\_bool registered = {true}; ///< registered in BlueStore's osr\_set
22. std::atomic\_bool zombie = {false}; ///< owning Sequencer has gone away
23. ...
24. };

**流程**

写操作大致如下,生成一个tranaction实例,作为本次写的事务实例,然后把要操作的object对象写到这个transaction中,具体写入的信息如代码:

1. auto t = new ObjectStore::Transaction;
2. t->write(cid, oid, offset, count, data);//生成一个写事务
3. tls.push\_back(std::move(\*t));//保存生成的事务

cid代表的是该object所属的collection对应的id,collection可以看做是object的集合,是内存数据结构，oid是本次要写入的object的id,offset是object内的写入偏移,count就是写入的长度,data就是长度为count字节的待写入数据.

接下来看下transaction::write都做了什么:

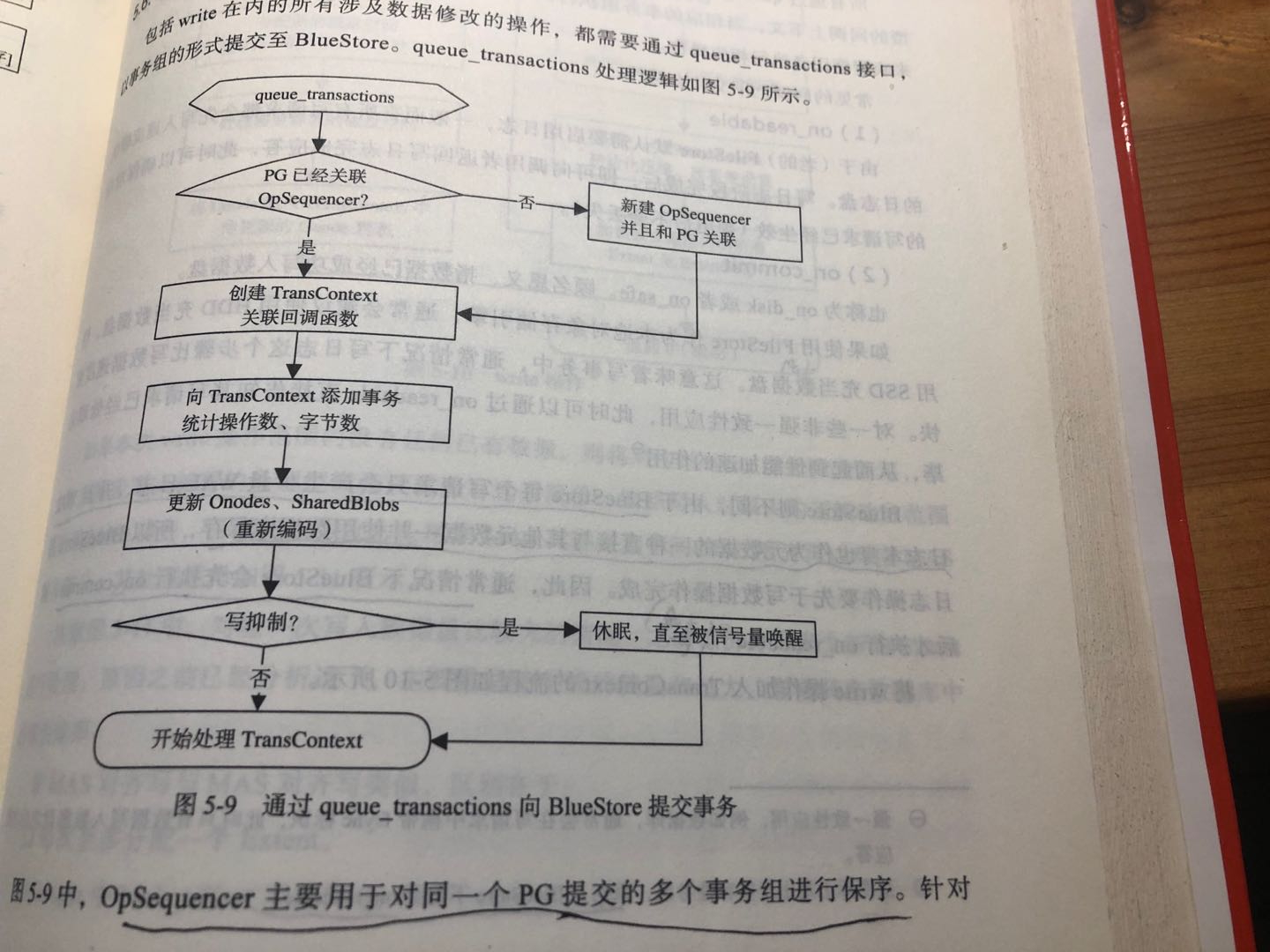
1. void write(const coll\_t& cid, const ghobject\_t& oid, uint64\_t off, uint64\_t len, const bufferlist& write\_data, uint32\_t flags = 0)
2. {
3. uint32\_t orig\_len = data\_bl.length();
4. //op就是代表这些上层的增删改查操作,这里通过op统一进行标记方便内部识别.然后将数据进行编码,其实就是head+body的形式,类似自定义二进制格式,这个我目前理解是方便内部使用更小的数据进行异步通信.
5. Op\* \_op = \_get\_next\_op();
6. \_op->op = OP\_WRITE;
7. \_op->cid = \_get\_coll\_id(cid);
8. \_op->oid = \_get\_object\_id(oid);
9. \_op->off = off;
10. \_op->len = len;
11. ::encode(write\_data, data\_bl);//二进制编码到data\_bl中
13. assert(len == write\_data.length());
14. //在data字段上记录并更新这个transaction对应的要操作的数据大小及偏移信息.
15. data.fadvise\_flags = data.fadvise\_flags | flags;
16. if (write\_data.length() > data.largest\_data\_len) {
17. data.largest\_data\_len = write\_data.length();
18. data.largest\_data\_off = off;
19. data.largest\_data\_off\_in\_data\_bl = orig\_len + sizeof(\_\_u32); // we are about to
20. }
21. data.ops++;
22. }

transaction写入object的信息完成后接下来就进行事务的提交了:

1. os->queue\_transactions(&sequencer, tls, nullptr, new C\_NotifyCond(&mutex, &cond, &done));//事务提交

然后使用条件变量进行结果等待.

**写事务核心流程分析**



OpSequencer主要用来对同一个PG的多个事务组进行保序，同一个事务组中的多个操作，BlueStore会创建一个事务上下文TransContext,将每个写操作顺序的添加到TransContext,然后批量进行处理。所以写操作完成之后，由TransContext汇总操作数及涉及的字节数，提交至Throttle，判断是否需要进行写抑制。如果没有过载，直接执行TransContext,否则需要进行休眠等待。

写入一个事务类似这样先写入bluestore对应的事务队列中:

1. os->queue\_transactions(&sequencer, tls, nullptr, new C\_NotifyCond(&mutex, &cond, &done));//事务提交

对应操作源码如下:

1. int BlueStore::queue\_transactions(
2. Sequencer \*posr,
3. vector<Transaction>& tls,
4. TrackedOpRef op,
5. ThreadPool::TPHandle \*handle)
6. {
7. ...
8. osr = new OpSequencer(cct, this);//生成一个事务操作"顺序器"
9. osr->parent = posr;//一个链式结构
10. posr->p = osr;
12. TransContext \*txc = \_txc\_create(osr);//生成一个transaction的context并关联opSequencer
13. txc->onreadable = onreadable;
14. txc->onreadable\_sync = onreadable\_sync;
15. txc->oncommit = ondisk;
16. for (vector<Transaction>::iterator p = tls.begin(); p != tls.end(); ++p) {
17. (\*p).set\_osr(osr);//把事务操作"顺序器"记录到传进来的transaction实例中
18. txc->bytes += (\*p).get\_num\_bytes();
19. \_txc\_add\_transaction(txc, &(\*p));//把transaction添加到上一步生成的transactioncontext上.
20. \_txc\_calc\_cost(txc);
21. \_txc\_write\_nodes(txc, txc->t);//根据txc里transaction涉及到的Onode进行操作,主要是对需要flush的进行计数,对共享的blob进行持久化,对需要reshard的Extentmap进行reshard
22. // journal deferred items
23. if (txc->deferred\_txn)//这个是延迟操作,需要走WAL写入,一般在small写中会发生,主要是因为出现了覆盖写,为了保证数据的完整性,需要先写WAL,而对齐的数据,写入失败了就失败了,反正node还没更新.
24. {
25. txc->deferred\_txn->seq = ++deferred\_seq;
26. bufferlist bl;
27. ::encode(\*txc->deferred\_txn, bl);
28. string key;
29. get\_deferred\_key(txc->deferred\_txn->seq, &key);
30. txc->t->set(PREFIX\_DEFERRED, key, bl);
31. }
32. ...
33. \_txc\_finalize\_kv(txc, txc->t);//没看懂
34. ...
35. if (txc->deferred\_txn)
36. {
37. // ensure we do not block here because of deferred writes
38. if (!throttle\_deferred\_bytes.get\_or\_fail(txc->cost))
39. {
40. dout(10) << \_\_func\_\_ << " failed get throttle\_deferred\_bytes, aggressive"
41. << dendl;
42. ++deferred\_aggressive;
43. deferred\_try\_submit();
44. {
45. // wake up any previously finished deferred events
46. std::lock\_guard<std::mutex> l(kv\_lock);
47. kv\_cond.notify\_one();
48. }
49. throttle\_deferred\_bytes.get(txc->cost);
50. --deferred\_aggressive;
51. }
52. }//if (txc->deferred\_txn)
53. }
54. // execute (start)
55. \_txc\_state\_proc(txc);//开始进入状态机,初始状态是:prepare
56. }

当所有写操作添加到TransContext后开始执行事务了：

1. 等待所有在途的写I/O完成

COW的I/O，可以与事务并发执行。

1. 同步元数据到数据库

同一个事务组中受到波及而发生变化的元数据

1. 通过WAL对应的日志事务执行覆盖写

**依赖的函数分别如下**

没有细分的地方是没看懂,细分的地方目前许多也是一知半解

创建一个事务:

1. BlueStore::TransContext \*BlueStore::\_txc\_create(OpSequencer \*osr)
2. {
3. TransContext \*txc = new TransContext(cct, osr);
4. txc->t = db->get\_transaction();//从rocksdb获取一个transaction
5. osr->queue\_new(txc);//把transaction放到osr的队列中,当做要排队执行的事务
6. return txc;
7. }

添加一个事务到context中:

1. void BlueStore::\_txc\_add\_transaction(TransContext \*txc, Transaction \*t)
2. {
3. Transaction::iterator i = t->begin();
4. //取出事务开始之前生成的collection,这个是object的集合.
5. vector<CollectionRef> cvec(i.colls.size());
6. unsigned j = 0;
7. for (vector<coll\_t>::iterator p = i.colls.begin(); p != i.colls.end();
8. ++p, ++j) {
9. cvec[j] = \_get\_collection(\*p);
10. }
12. vector<OnodeRef> ovec(i.objects.size());//根据transaction要操作的object的大小创建一个Onode的集合.
13. //逐个处理之前放进去的"操作",就是要执行的对应操作,增删改查什么的.
14. for (int pos = 0; i.have\_op(); ++pos) {
15. Transaction::Op \*op = i.decode\_op();
16. ...
17. // object operations
18. RWLock::WLocker l(c->lock);
19. OnodeRef &o = ovec[op->oid];//上边那个Onode的集合,取出来
20. if (!o) {
21. ghobject\_t oid = i.get\_oid(op->oid);
22. o = c->get\_onode(oid, create);//从collection中取出一个Onode,没有就去create
23. }
24. //通过分支判断操作类型并进行相应的处理,这里我们重点分析写操作
25. switch (op->op) {
26. ...
27. case Transaction::OP\_WRITE:
28. {
29. uint64\_t off = op->off;//本次操作的偏移,偏移是相对Onode的,因为它是object的载体
30. uint64\_t len = op->len;//本次操作的长度
31. uint32\_t fadvise\_flags = i.get\_fadvise\_flags();//暂不清楚
32. bufferlist bl;
33. i.decode\_bl(bl);//把数据取出来放到buffer上
34. r = \_write(txc, c, o, off, len, bl, fadvise\_flags);//写事务
35. }
36. }
37. ...
38. }
39. }

写事务具体实现:

1. int BlueStore::\_write(TransContext \*txc,
2. CollectionRef& c,
3. OnodeRef& o,
4. uint64\_t offset, size\_t length,
5. bufferlist& bl,
6. uint32\_t fadvise\_flags)
7. {
8. ...
9. r = \_do\_write(txc, c, o, offset, length, bl, fadvise\_flags);//写设备
10. txc->write\_onode(o);//写Onode
11. ...
12. }

写设备:

1. int BlueStore::\_do\_write(
2. TransContext \*txc,
3. CollectionRef& c,
4. OnodeRef o,
5. uint64\_t offset,
6. uint64\_t length,
7. bufferlist& bl,
8. uint32\_t fadvise\_flags)
9. {
10. ...
11. //为刷盘做记录
12. uint64\_t end = offset + length;
13. auto dirty\_start = offset;
14. auto dirty\_end = end;
16. o->extent\_map.fault\_range(db, offset, length);//这个是比较核心的一些关于Onode结构的设置,这里先简单描述下,到核心数据结构篇章再详细描述实现原理,大致就是为object的Onode与blob进行映射,这期间涉及到Onode包含哪些blob,准备从哪里开始写,写多长,可能写入的时候还会出现夸多个blob的情况,也是这里需要记录处理的,Onode和blob是多对多的关系.
17. \_do\_write\_data(txc, c, o, offset, length, bl, &wctx);//这个对写进行拆分,按照min\_alloc\_size对齐区分,这个决定写的方式是是append还是RMW,对最后的刷盘方式有影响.
18. r = \_do\_alloc\_write(txc, c, o, &wctx);//给txc分配空间
19. //中间还有一些gc的东西,这里暂时先忽略
20. ...
21. \_wctx\_finish(txc, c, o, &wctx);//主要是针oldExtent的数据进行读取并从oldExtent记录中删除,完成上下文的写操作.
22. ...
23. o->extent\_map.compress\_extent\_map(dirty\_start, dirty\_end - dirty\_start);
24. o->extent\_map.dirty\_range(dirty\_start, dirty\_end - dirty\_start);
25. ...
26. }

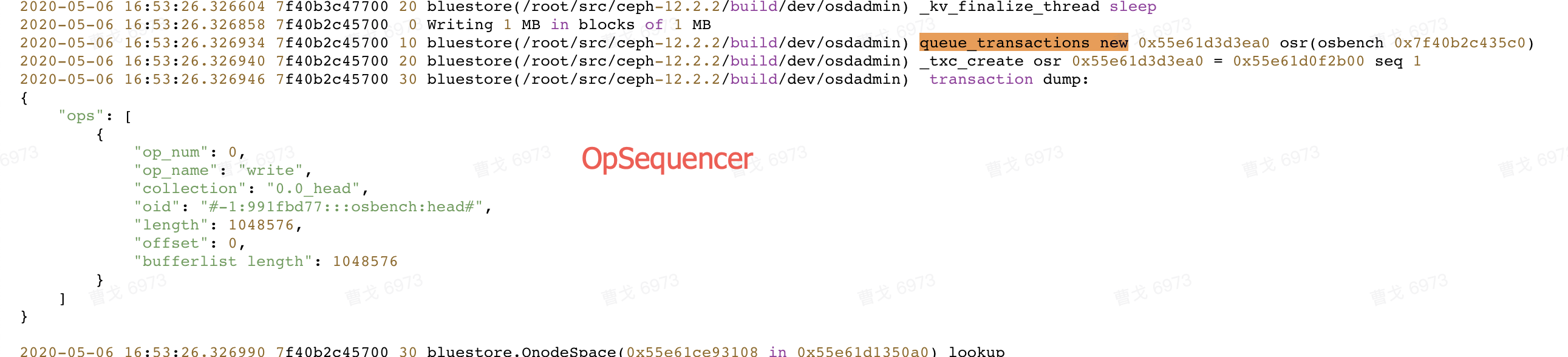


图-1

代码中的OpSequencer是用来关联相关事务的,这些事务需要有FIFO的特性,先来先执行,不能并发,不同OpSequencer之间是可以并发的.

osr会把自己注册到上边的this中,即当前这个bluestore实例中:

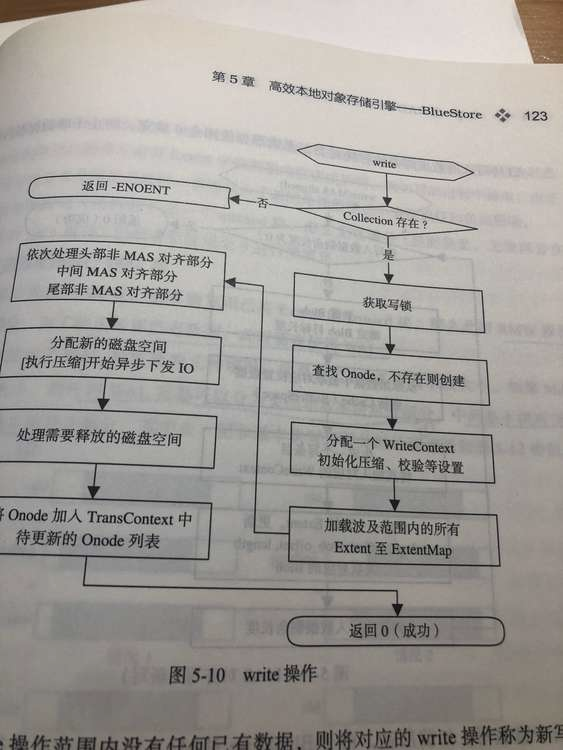
1. OpSequencer(CephContext\* cct, BlueStore \*store): Sequencer\_impl(cct), parent(NULL), store(store)
2. {
3. store->register\_osr(this);
4. }

所有通过queue\_tranactions提交的事务组都是异步执行的，因此需要指定若干种类型的回调上下文，当相应的事务组执行到某个特定阶段后，通过执行对应的回调上下文来唤醒调用者执行相应操作。

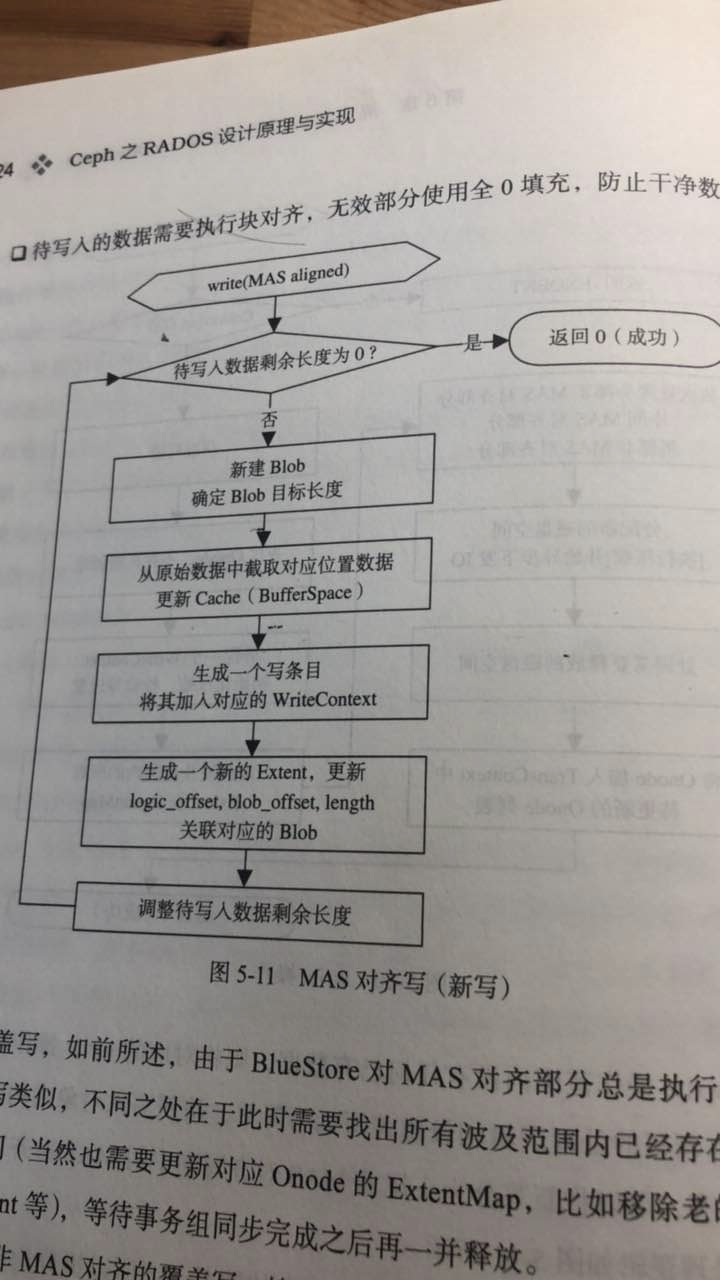
常见的回调上下文有一种：

1. on\_commit

on\_disk/on\_safe。数据写入磁盘成功。

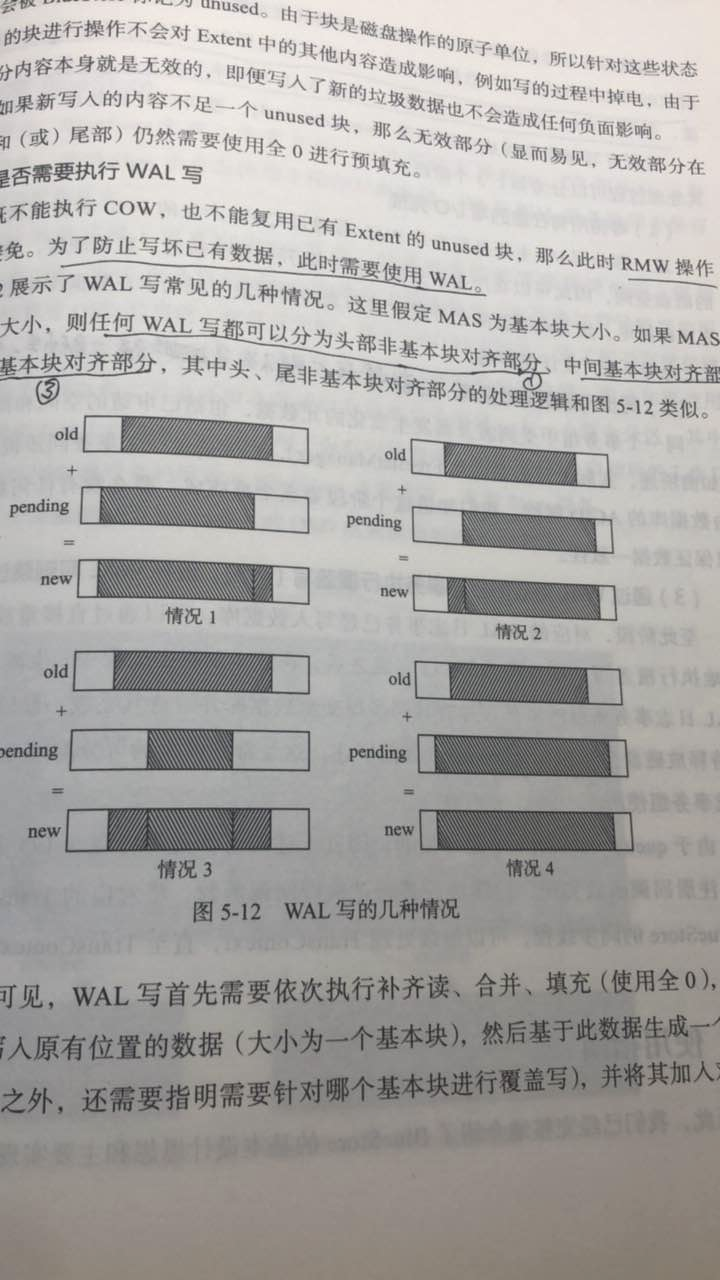


如果是对齐写，如下图：



如果是不是对齐写，但是可以使用unused的块，也是可以直接写入的。

但是如果要执行RAW，需要先写WAL.如下图:



写入分为三种情况

1. 新写

Minimal Allocable Size简称MAS,如果本次write操作范围内没有任何已有的数据，则将对应的write操作成为新写，否则就是覆盖写，按照写入对象内的逻辑地址范围是否是MAS对齐，新写可以分为头尾非MAS对齐写和中间MAS对齐写。对应MAS对齐部分如上图5-11，而对于非MAS对齐的新写，区别在于：

1. 每次至多分配一个Extent
2. Extent中blob\_offset不是0
3. 待写入的数据需要执行块对齐，无效部分使用全0填充，防止干净数据被污染
4. MAS对齐覆盖写

和上边MAS对齐写相同，不同的地方是需要找出覆盖写范围内已经存在的Extent和它们占有的空间，等事务组同步完成后一并释放

1. 非MAS对齐覆盖写

是最复杂的一种写，既要考虑不对齐的问题又要考虑覆盖数据的问题，一般需要重点考虑三个因素

1. 能否直接跳过，执行COW

压缩过的Extent因为考虑性能的问题不COW

1. 能否直接复用已有Extent的unused块
2. 是否要执行WAL写

如果必须要覆盖已有的数据，而且不能COW，就需要先使用WAL记录RMW的操作。如上图5-12，需要依次执行补齐读，合并，填充0，得到一个新的，待写入原有位置的一个基本块，并形成一个日志事务，指名需要针对哪个基本块进行覆盖写，并加入到TransContext。日志事务和其它元数据一样使用数据库保存，只有WAL成功返回后才能开始对原有区域执行覆盖写。

\_do\_write\_data的代码细节:

1. void BlueStore::\_do\_write\_data(
2. TransContext \*txc,
3. CollectionRef& c,
4. OnodeRef o,
5. uint64\_t offset,
6. uint64\_t length,
7. bufferlist& bl,
8. WriteContext \*wctx)
9. {
10. ...
11. uint64\_t end = offset + length;//计算写数据的end位置,这里的offset是Onode的,ExtentMap会根据这个位置和min\_alloc\_size计算所在的extent位置,进而计算出大概在哪个blob上或者哪些blob上.
12. if (offset / min\_alloc\_size == (end - 1) / min\_alloc\_size &&
13. (length != min\_alloc\_size)) {
14. //如果本次要写入的数据大小不超过min\_alloc\_size,就认为是small写,调用small写操作.
15. // we fall within the same block
16. \_do\_write\_small(txc, c, o, offset, length, p, wctx);
17. } else {
18. //下边这些是把写分成head,middle,tail三个部分,其中middle是min\_alloc\_size的整数倍写,head和tail都不足一个min\_alloc\_size,middle\_offset~middle\_offset+middle\_length区间走的是大块写,head和tail走的写小块写,下边我们细看下大块写和小块写的细节,然后总结下区别.
19. uint64\_t head\_offset, head\_length;
20. uint64\_t middle\_offset, middle\_length;
21. uint64\_t tail\_offset, tail\_length;
23. head\_offset = offset;
24. head\_length = P2NPHASE(offset, min\_alloc\_size);
26. tail\_offset = P2ALIGN(end, min\_alloc\_size);
27. tail\_length = P2PHASE(end, min\_alloc\_size);
29. middle\_offset = head\_offset + head\_length;
30. middle\_length = length - head\_length - tail\_length;
32. if (head\_length) {
33. \_do\_write\_small(txc, c, o, head\_offset, head\_length, p, wctx);
34. }
36. if (middle\_length) {
37. \_do\_write\_big(txc, c, o, middle\_offset, middle\_length, p, wctx);
38. }
40. if (tail\_length) {
41. \_do\_write\_small(txc, c, o, tail\_offset, tail\_length, p, wctx);
42. }
43. }
44. ...
45. }

\_do\_wirte\_small代码细节:

1. void BlueStore::\_do\_write\_small(
2. TransContext \*txc,
3. CollectionRef &c,
4. OnodeRef o,
5. uint64\_t offset, uint64\_t length,
6. bufferlist::iterator& blp,
7. WriteContext \*wctx)
8. {
9. ...
10. //WIP,再写个不对齐的测试下,size=1000
11. ...
12. }

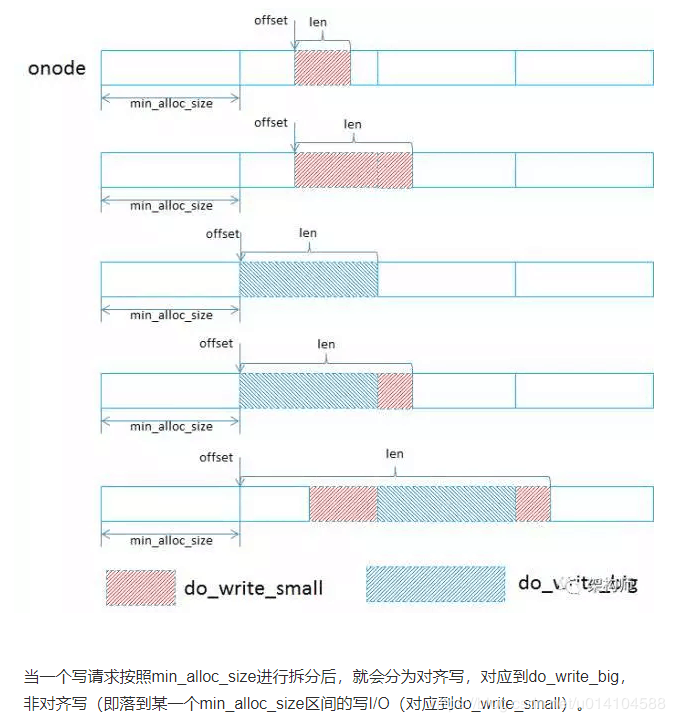
先写日志,再覆盖数据.日志中主要包括meta数据和需要覆盖写入的数据,等覆盖完成后再删除日志中的数据.

\_do\_write\_big代码细节:

1. void BlueStore::\_do\_write\_big(
2. TransContext \*txc,
3. CollectionRef &c,
4. OnodeRef o,
5. uint64\_t offset, uint64\_t length,
6. bufferlist::iterator &blp,
7. WriteContext \*wctx)
8. {
9. ...
10. //根据要写入的数据在Extent的位置情况进行Extent拆分
11. o->extent\_map.punch\_hole(c, offset, length, &wctx->old\_extents);
12. //下边就是blob的复用逻辑,然后生成一个write\_item记录到wctx上.
13. ...
14. }

我们用一张图描述下small写和big写:这张图是我从这里取的,侵权请联系删除( https://gist.github.com/baymaxium/41689786a4648d88ffc0c11d86a6eb98, https://gist.github.com/cgCodeLife/f8558909f8f8d4fd16e14376f3f56668 先fork到自己的Git上了)

这里其实是写I/O的核心,它处理了Onode的各种对齐情况.small比较复杂:https://gist.github.com/cgCodeLife/f8558909f8f8d4fd16e14376f3f56668#do\_write\_small



**写I/O流程图**

虽然函数许多地方细节没有看的太清楚,不过,大概知道在进入状态机之前,都是先构造一系列的transaction然后再写Onode,包括下边的logical extent,blob, pextent.

这里放一张网上别人写的图,虽然代码暂时没太看太明白,不过看过代码后这张图我还是基本看懂了,之前一脸懵逼.

