**BlueStore源码分析**

[源码编译安装指北](https://juejin.im/post/5ea9ac08f265da7bfa190547)

**时序图**

//进阶搞

**启动时序图**

**写数据时序图**

**读数据时序图**

**事务状态机流转图**

**核心数据结构源码分析**

**Bluestore**

**rocksdbEnv**

**BlueFS**

**BlockDev**

**UML图**

**相关事务状态说明**

**初始状态**

**描述**

\_txc\_state\_proc里就是状态机的处理逻辑，根据所处的状态进行不同阶段的处理。起始状态是STATE\_PREPARE，在这个状态下会检查是否还有未完成的aio，如果有就将状态置为STATE\_AIO\_WAIT，并调用\_txc\_aio\_submit进行处理，否则就直接进入到下一个状态STATE\_AIO\_WAIT的处理。

在\_txc\_aio\_submit里就是就是调用bdev->aio\_submit –> KernelDevice::aio\_submit –> io\_submit将aio提交到内核进行处理（注：目前支持KernelDevice和NVMEDevice，这里以KernelDevice为例）。

使用linux Aio时，将I/O提交后，当内核处理完成后会通知到用户态，一种方式是使用eventfd，将其注册到epoll里，当内核处理I/O完成后会触发eventfd的事件从而进行处理；另外一种方式是用一个单独的线程，轮询调用io\_getevents去获取完成的事件。在bluestore里采用的是第二种方式，对应的线程是KernelDevice::\_aio\_thread，当I/O处理完后会调用回调函数aio\_cb进行处理（这个回调函数是在bluestore启动时创建KernelDevice设置的），在aio\_cb中调用\_txc\_aio\_finish –> \_txc\_state\_proc，从而进入到下一个状态STATE\_AIO\_WAIT的处理。

**函数调用**

queue\_transactions->\_write()->\_do\_write()->aio\_write()->\_tx\_add\_transaction()->\_txc\_state\_proc()->STATE\_PREPARE->\_txc\_aio\_submit(),设置STATE\_AIO\_WAIT->等待设备写数据异步完成的回调执行,执行\_txc\_state\_proc()为STATE\_AIO\_WAIT。

**异步IO执行结果等待状态**

**描述**

在这个状态里是调用\_txc\_finish\_io进行处理，会将状态设置成STATE\_IO\_DONE。因为aio的完成可能是乱序的，有可能后提交的I/O先完成，但是需要保证kv事务的顺序性。bluestore里通过OpSequencer来保证kv事务的顺序性（在\_txc\_create里会将新的txc放到osr->q里，即q.push\_back。在\_osr\_reap\_done里从osr->q里挨个剔除完成的。），在\_txc\_finish\_io里就是实现通过OpSequencer来保证每个事务在处于某个状态时，这个事务之前的事务也必须在这个状态，即使某个事务的I/O先完成，也得等到它之前的事务的I/O也完成后才能进入到下个状态的处理。

做了这个保证后，再按序对STATE\_IO\_DONE状态的事务调用\_txc\_state\_proc进入下一个状态的处理。

**函数调用**

\_txc\_finish\_io()

**IO执行完成**

**描述**

进入这个状态后，会设置下个状态为STATE\_KV\_QUEUED，然后会根据bluestore\_sync\_transaction和bluestore\_sync\_submit\_transaction这两个配置参数的组合作不同的处理：

1）bluestore\_sync\_transaction为true：表示同步提交kv到rocksdb并持久化，对应调用\_txc\_finalize\_kv后再调用db->submit\_transaction，即rocksdb::Write并设置rocksdb::WriteOptions.sync=true；

2）bluestore\_sync\_transaction为false，bluestore\_sync\_submit\_transaction为true：表示将kv提交到rocksdb，但是不sync，也就是没有落盘，对应调用\_txc\_finalize\_kv后再调用db->submit\_transaction\_sync，即rocksdb::Write，但rocksdb::WriteOptions.sync=false；

不管采用何种处理方式，最后都会将事务放到kv\_queue里，然后通过kv\_cond通知\_kv\_sync\_thread。

**函数调用**

db->submit\_transaction()

kv\_queue.push\_back()

kv\_cond.notify\_one()

kv\_queue\_unsubmitted.push\_back()

**\_kv\_sync\_thread处理kv\_queue**

**描述**

\_kv\_sync\_thread线程里就是处理kv\_queue和kv\_submiting的事务，这一步会对这些操作进行同步刷盘操作，写入rocksdb并同步到持久化存储。然后更新事务状态为STATE\_KV\_SUBMITTED。

**函数调用**

\_txc\_apply\_kv()

submit\_transaction\_sync()

**STATE\_KV\_QUEUED**

该状态没有在状态机中处理，在\_kv\_sync\_thread中用该状态判断是否应该进行同步事务操作。

**STATE\_KV\_SUBMITTED**

进行了\_txc\_committed\_kv调用，主要是设置状态为STATE\_KV\_DONE，并完成finisher队列入队。

**STATE\_KV\_DONE**

如果有需要进行deferred的操作将状态转为STATE\_KV\_DEFERRED,否则转为finisher、

**STATE\_KV\_DEFERRED**

该状态是提交哪些wal日志的，主要会把这些数据提交到持久化存储介质中。

**STATE\_KV\_FINISHING**

调用\_txc\_finish进行一些结束状态的处理。

从上面的流程分析可以知晓，一个I/O在bluestore里经历了多个线程和队列才最终完成，对于非WAL的写，比如对齐写、写到新的blob里等，I/O先写到块设备上，然后元数据提交到rocksdb并sync了，才返回客户端写完成（在STATE\_KV\_QUEUED状态的处理）；对于WAL（即覆盖写），没有先把数据写块设备，而是将数据和元数据作为wal一起提交到rocksdb并sync后，这样就可以返回客户端写成功了，然后在后面的动作就是将wal里的数据再写到块设备的过程，对这个object的读请求要等到把数据写到块设备完成整个wal写I/O的流程后才行，代码里对应的是\_do\_read里先o->flush()的操作，所以bluestore里的wal就类似filestore里的journal的作用。

**BlueStore读数据流程**

**问题**

ceph的集群管理，数据一致性特性

**代码分析**

Draft：这些数据结构和函数操作需要细化下，把allocate，bluefs,rocksdbEnv,blockdebvice都串起来

数据结构

class ObjectStore

1. //ObjectStore.h
2. //代码注释里有关于object的定义描述:object={data, xattr,omap\_header, omap}
3. class ObjectStore
4. {
5. ...
6. //多个Collection可以并行，但是一个Collection里的事务需要保持入队时候的一致性。
7. struct CollectionImpl
8. {
9. //定义对事务的一些操作
10. }
12. ...
13. }

class Transaction

关于其中的几个代表不同操作行为的成员，代码注释中已经进行了关键意义及区别描述，可以看下代码，在Transaction.h 中声明class Transaction的开头。

1. //Transaction.h
2. class Transaction
3. {
4. struct Op
5. {
6. //定义一些支持的事务操作
7. }
8. struct TransactionData
9. {
10. ...
11. private:
12. std::list<Context \*> on\_applied;//ObjectStore异步提交的事务
13. std::list<Context \*> on\_commit;//在Finisher线程把互斥事务都提交到stable 存储中后对事务进行收尾
14. std::list<Context \*> on\_applied\_sync;//ObjectStore同步提交的事务
15. ...
16. }
17. };

Onode

SharedBlob

struct Op

struct TransactionData

**BlueStore写数据流程**

函数：

queue\_transaction->queue\_transactions:

ObjectStore::Transaction::collect\_contexts(tls, &on\_applied, &on\_commit, &on\_applied\_sync);//收集transaction相关的提交信息

\_txc\_add\_transaction()//添加transaction

\_write()->\_do\_write()->:\_do\_write\_data(),\_wctx\_finish()//进行写操作，这里面是一些写事务相关的状态机的操作。

write\_onode()//将onode insert到txc中，等待record\_onode的操作，不过，在这之前，其实已经将数据先写到磁盘了，这样，如果上层逻辑更新后那就是真的更新成功了。

//上层逻辑更新，这里涉及到的就是关于meta信息的一些更新，主要是Onode的操作。

\_txc\_write\_nodes

\_record\_onode//操作一些onode的offset

**O&A**

1. 文件锁原理
2. ceph的数据迁移怎么实现的