OSD读取流程

进队列流程

```
OSD::ms_fast_dispatch
  OSD::dispatch_session_waiting
    OSD::enqueue_op
      op_shardedwq.queue
```

- OSD::ms_fast_dispatch主要检查service服务 、把Message封装为OpRequest类型、设置并检查版本信息，获取session。分为两种处理方式：如果消息的connection有CEPH_FEATUREMASK_RESEND_ON_SPLIT的特性，则直接调用enqueue_op；否则加入session->waiting_on_map，最后dispatch_session_waiting。
- OSD::dispatch_session_waiting 主要是循环处理队列waiting_on_map中的元素，获取它们的pgid，最后调用enqueue_op处理。
- OSD::enqueue_op 的主要工作是将请求加入到op_shardedwq队列中，最终会hash到shard_list对应的ShardData中。

OSD构建时（ShardedThreadPool osd_op_tp）：

```
op_shardedwq(
      get_num_op_shards(),
      this,
      cct->_conf->osd_op_thread_timeout,
      cct->_conf->osd_op_thread_suicide_timeout,
      &osd_op_tp),
```

在OSD::init中，会调用osd_op_tp.start()创建多个WorkThreadSharded线程，WorkThreadSharded继承自Thread类，其中包括一个线程入口函数entry()会调用shardedthreadpool_worker，其中会执行wq->_process(thread_index, hb);

```
struct WorkThreadSharded : public Thread {
  ShardedThreadPool *pool;
  uint32_t thread_index;
  WorkThreadSharded(ShardedThreadPool *p, uint32_t pthread_index): pool(p),
    thread_index(pthread_index) {}
  void *entry() override {
    pool->shardedthreadpool_worker(thread_index);
    return 0;
  }
};
```

```
void run(OSD *osd, PGRef &pg, ThreadPool::TPHandle &handle) {
  RunVis v(osd, pg, handle);
  boost::apply_visitor(v, qvariant);
}

void PGOpItem::run(
  OSD *osd,
  OSDShard *sdata,
  PGRef& pg,
  ThreadPool::TPHandle &handle)
{
  osd->dequeue_op(pg, op, handle);
  pg->unlock();
}
```

出队列处理调用流程：

```
OSD::ShardedOpWQ::_process
  void PGOpItem::run
    void OSD::dequeue_op
      void PrimaryLogPG::do_request
        void PrimaryLogPG::handle_backoff
        void PrimaryLogPG::do_op
          m->finish_decode()
          int PrimaryLogPG::find_object_context
            ObjectContextRef PrimaryLogPG::get_object_context
          void PrimaryLogPG::execute_ctx
            obc->ondisk_read_lock()
            int PrimaryLogPG::prepare_transaction
              int PrimaryLogPG::do_osd_ops
                int PrimaryLogPG::do_read
                  int ReplicatedBackend::objects_read_sync
                    int BlueStore::read
            obc->ondisk_read_unlock()
            start_async_reads/complete_read_ctx
              osd->send_message_osd_client(reply, m->get_connection()

        void PrimaryLogPG::do_sub_op
        void PrimaryLogPG::do_scan
        void PrimaryLogPG::do_backfill
        void PrimaryLogPG::do_backfill_remove
        ...
```

- OSD::ShardedOpWQ::_process会处理thread_index % num_shards的ShardData，并且从ShardData中取出item有加锁操作。
- OSD::dequeue_op 调用函数进行osdmap的更新，调用do_request进入PG处理流程
- PrimaryLogPG::do_request主要检查PG和OP的状态，以及根据消息类型进行不同处理，对于read请求，调用为do_op。
- void PrimaryLogPG::do_op该函数十分复杂，只分析读数据相关部分：首先对消息解码，对读操作打标签，对读操作的错误情况进行拦截（正常情况只从Primary OSD读取信息，设置了设置CEPH_OSD_FLAG_BALANCE_READS或者CEPH_OSD_FLAG_LOCALIZE_READS都可以进行读取）。判断op中是否includes_pg_op操作，调用do_pg_op处理pg相关的操作。对操作的个中参数进行检查，检查相关对象的状态，以及该对象的head、snap、clone对象的状态等，并调用函数获取对象的上下文、操作的上下文（ObjectContext、OPContext）
- void PrimaryLogPG::execute_ctx在读流程中主要完成了4项任务：加锁，prepare_transaction，解锁，回应。
- int PrimaryLogPG::prepare_transaction将主要的工作委托给了do_osd_ops函数
- int PrimaryLogPG::do_osd_ops根据不同的类型进行不同的处理，针对读取请求调用do_read

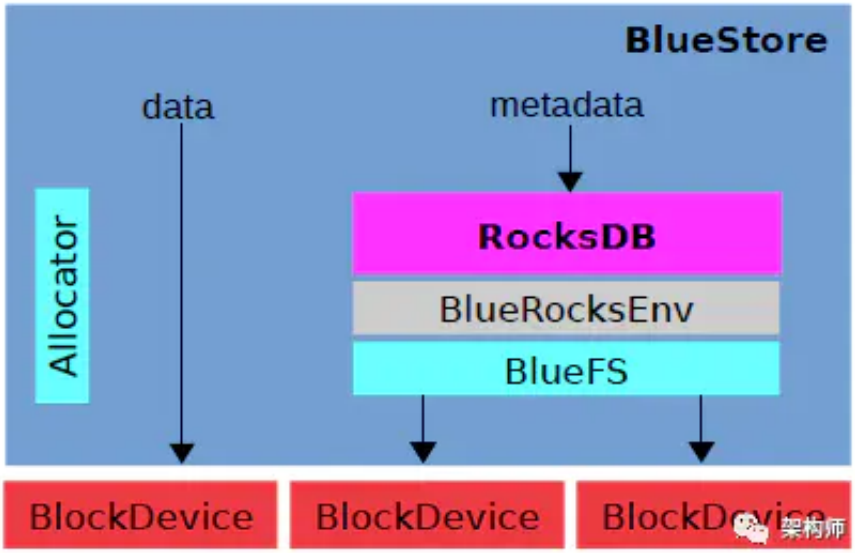
BlueStore读取流程

```
int BlueStore::read
  RWLock::RLocker l(c->lock)
  BlueStore::OnodeRef BlueStore::Collection::get_onode
    onode_map.lookup(oid)
    store->db->get(PREFIX_OBJ, key.c_str(), key.size(), &v)
  int BlueStore::_do_read
    BlueStore::extent_map_t::iterator BlueStore::ExtentMap::seek_lextent
    bdev->aio_read/bdev->read
    bdev->aio_submit(&ioc)
    ioc.aio_wait()
```

- BlueStore::Collection::get_onode首先采用oid在onode_map进行查找，如果找到直接返回即可；否则生成对象的key从rocksdb中获取对象的onode信息并加入到onode_map中。
- int BlueStore::_do_read首先会调用seek_lextent生成读取数据范围的lextent迭代器，即logical_offset <= offset <= logical_end()，然后遍历每个lextent确定其中需要读取的范围并加入到blobs2read中。遍历blobs2read中的每个blob如果数据是compressed，则将所有数据读取出来，解压后进行处理；否则在blob中按pieces进行读取。如果有多个region则采用bdev->aio_read异步读取，否则采用bdev->read，提交并等待所有操作完成。最后将读取的数据进行组合。

BlueStore结构

结构



元数据