OSD 读写源码分析

1 日志

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

```
$ git log --pretty=oneline
7fe91d5d5842e04be3b4f514d6dd990c54b29c76 (HEAD, tag: v18.2.1) 18.2.1
```

ceph中日志机制很高级，比如源码中是这样描述的：

```
  dout(20) << __func__ << " alloc reply " << ctx->reply
       << " result " << result << dendl;
```

实际输出却是：

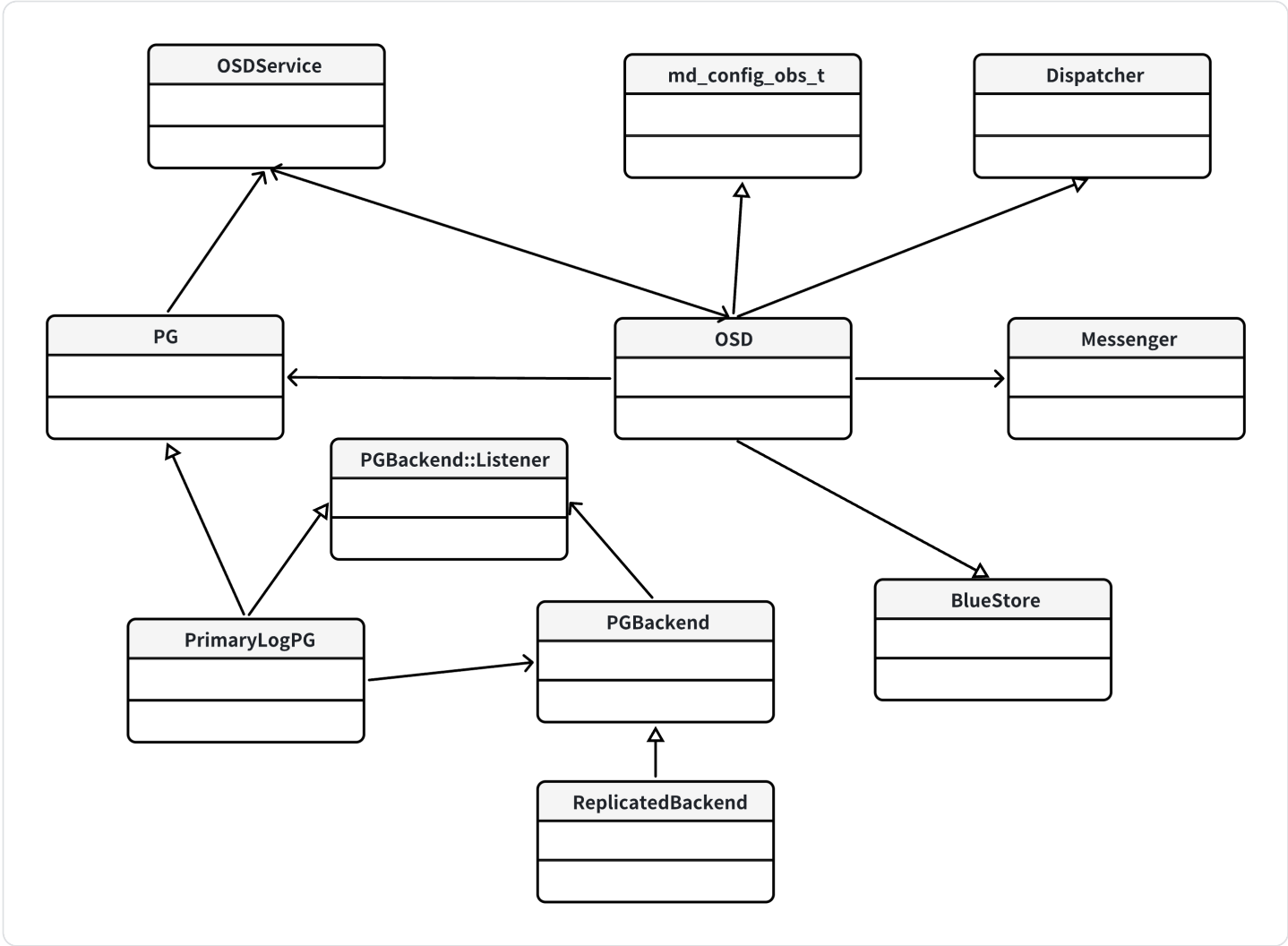
```
       2024-01-17T05:39:56.991+0000 7f2018753700 20 osd.0 pg_epoch: 130
pg[4.16( v 130'48755 (129'46400,130'48755] local-lis/les=107/108 n=401
ec=45/24 lis/c=107/107 les/c/f=108/108/0 sis=107) [0,1,2] r=0 lpr=107
crt=130'48755 lcod 130'48754 mlcod 130'48754 active+clean] execute_ctx alloc
reply 0x55f568510000 result 0
```

在__func__前面，时间戳、线程号、日志级别、pg等，都是动态添加进去的。

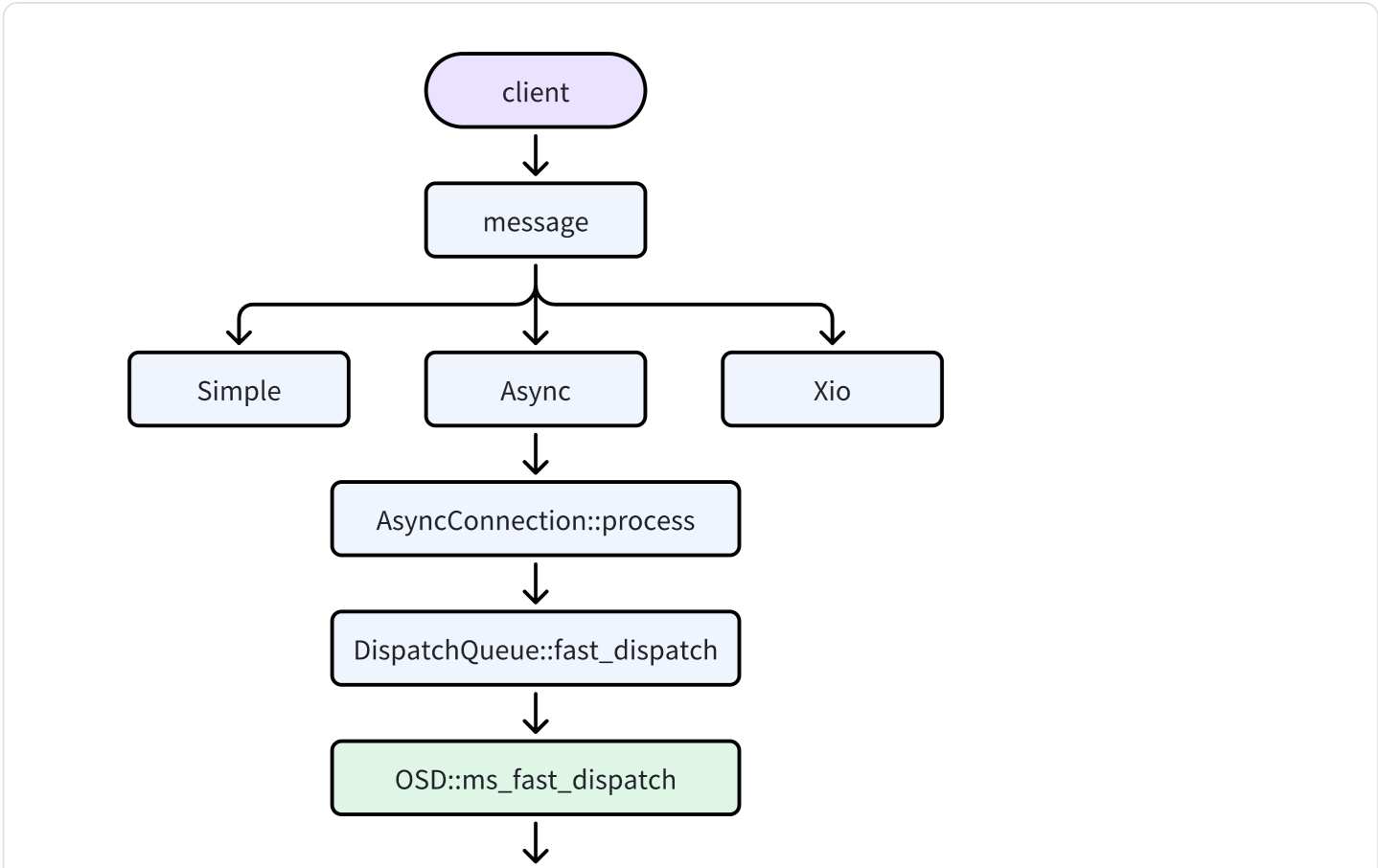
```
# 这个函数就添加了pg信息的日志前缀，具体是怎么做到的以后再看
template <typename T>
static ostream& _prefix(std::ostream *_dout, T *pg) {
  return pg->gen_prefix(*_dout);
}
```

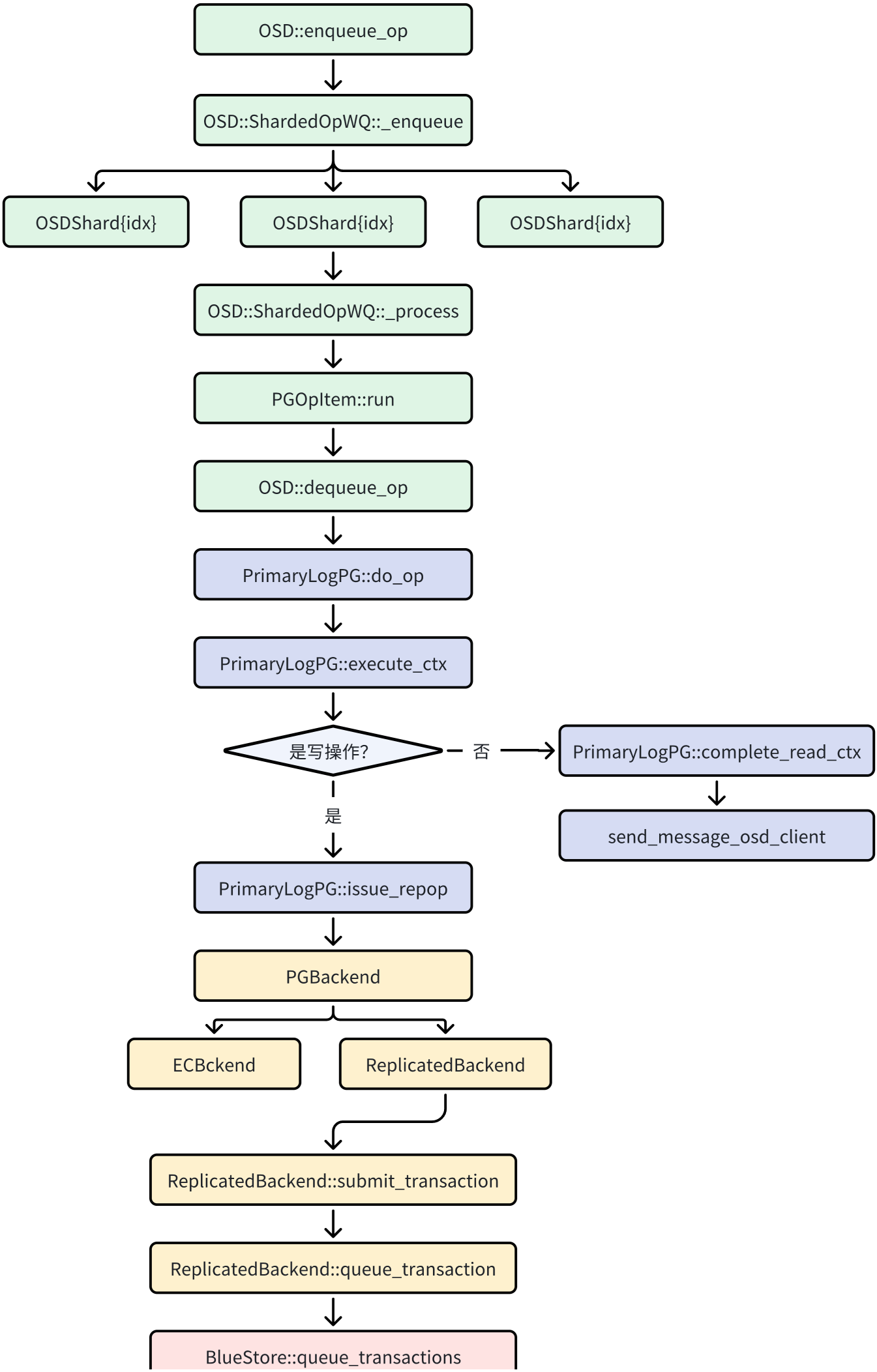
2 总览

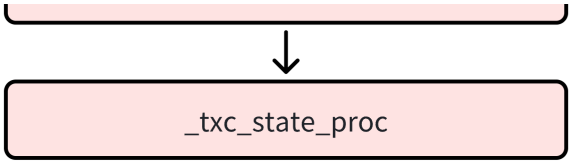
2.1 OSD读写类图


OSDService
md_config_obs_t
Dispatcher
PG
OSD
Messenger
PGBackend::Listener
BlueStore
PrimaryLogPG
PGBackend
ReplicatedBackend

2.2 主要流程

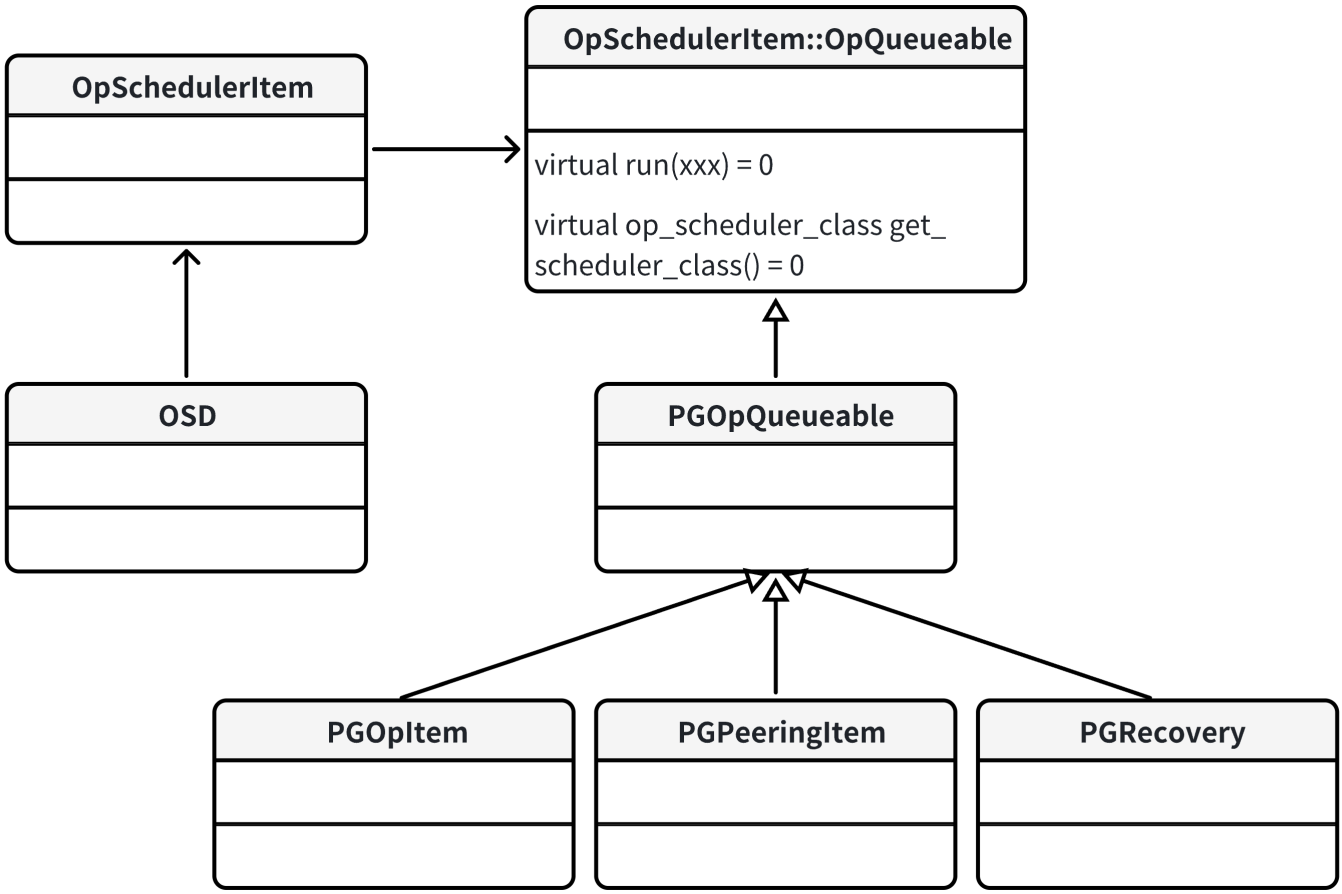

client
message
Simple
Async
Xio
AsyncConnection::process
DispatchQueue::fast_dispatch
OSD::ms_fast_dispatch


OSD::enqueue_op
OSD::ShardedOpWQ::_enqueue
OSDShard{idx}
OSDShard{idx}
OSDShard{idx}
OSD::ShardedOpWQ::_process
PGOpItem::run
OSD::dequeue_op
PrimaryLogPG::do_op
PrimaryLogPG::execute_ctx
是写操作?
否
PrimaryLogPG::complete_read_ctx
send_message_osd_client
是
PrimaryLogPG::issue_repop
PGBackend
ECBckend
ReplicatedBackend
ReplicatedBackend::submit_transaction
ReplicatedBackend::queue_transaction
BlueStore::queue_transactions



3 相关数据结构

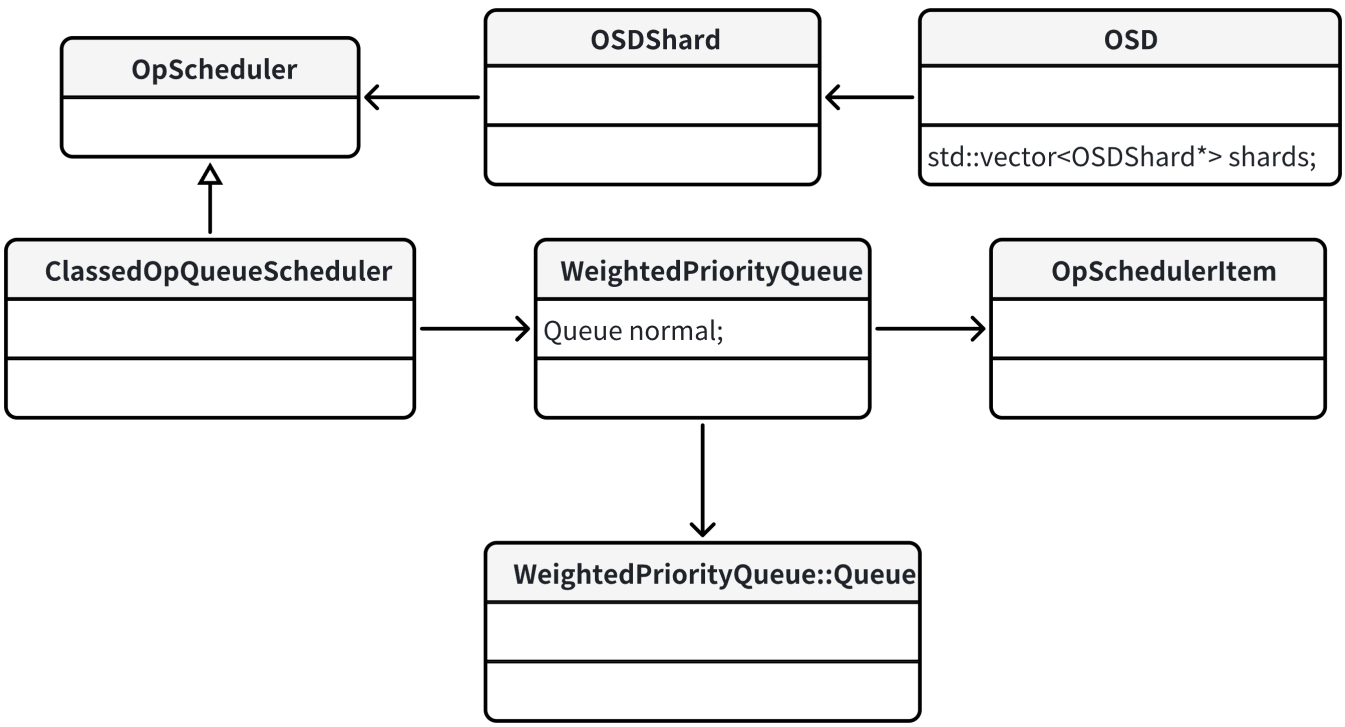
OpSchedulerItem




```
class OpSchedulerItem {
  OpQueueable::Ref qitem;
  int cost;
  unsigned priority;
  utime_t start_time;
  uint64_t owner;
  epoch_t map_epoch;

    OpSchedulerItem(
    OpQueueable::Ref &&item,
    int cost,
    unsigned priority,
```

```
    utime_t start_time,
    uint64_t owner,
    epoch_t e)
    : qitem(std::move(item)),
      cost(cost),
      priority(priority),
      start_time(start_time),
      owner(owner),
      map_epoch(e) {}
};
```

OSDShard




```
struct OSDShard {
  /// priority queue
  ceph::osd::scheduler::OpSchedulerRef scheduler;
    OSDShard(
    int id,
    CephContext *cct,
    OSD *osd);
};
```

```
class OpScheduler {
```

```
public:
  // Enqueue op for scheduling
  virtual void enqueue(OpSchedulerItem &&item) = 0;
  ...
};

template <typename T>
class ClassedOpQueueScheduler final : public OpScheduler {
  T queue;
...
};
```

```
template <typename T, typename K>
class WeightedPriorityQueue :  public OpQueue <T, K> {
...
class Queue{};
Queue normal;
}
```

初始化

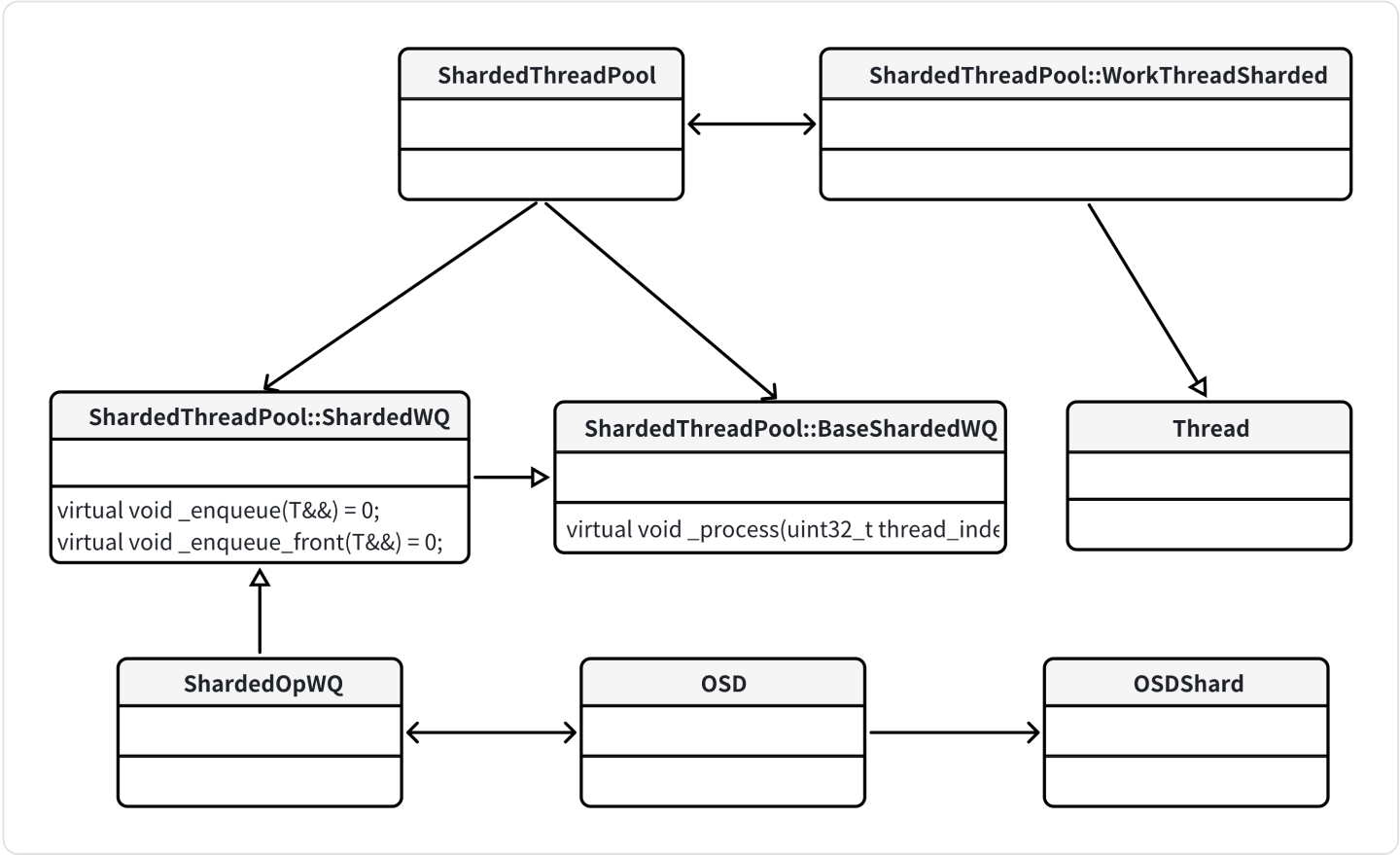
```
for (uint32_t i = 0; i < num_shards; i++) {
  OSDShard *one_shard = new OSDShard(
    i,
    cct,
    this);
  shards.push_back(one_shard);
}
============
OSDShard::OSDShard(
  int id,
  CephContext *cct,
  OSD *osd)
  : shard_id(id),
    cct(cct),
    osd(osd),
    shard_name(string("OSDShard.") + stringify(id)),
    sdata_wait_lock_name(shard_name + "::sdata_wait_lock"),
    sdata_wait_lock{make_mutex(sdata_wait_lock_name)},
    osdmap_lock{make_mutex(shard_name + "::osdmap_lock")},
    shard_lock_name(shard_name + "::shard_lock"),
    shard_lock{make_mutex(shard_lock_name)},
    scheduler(ceph::osd::scheduler::make_scheduler(
```

```
      cct, osd->whoami, osd->num_shards, id, osd->store->is_rotational(),
      osd->store->get_type(), osd->monc)),
    context_queue(sdata_wait_lock, sdata_cond){}
============
OpSchedulerRef make_scheduler() {
    return std::make_unique<
      ClassedOpQueueScheduler<WeightedPriorityQueue<OpSchedulerItem, client>>>(
        cct,
        cct->_conf->osd_op_pq_max_tokens_per_priority,
        cct->_conf->osd_op_pq_min_cost
    );
}
```

添加

```
==========调用==========
OSDShard* sdata = osd->shards[shard_index];
sdata->scheduler->enqueue(std::move(item));
==========ClassedOpQueueScheduler========
 void enqueue(OpSchedulerItem &&item) final {
    unsigned priority = item.get_priority();
    unsigned cost = item.get_cost();
    queue.enqueue(item.get_owner(), priority, cost, std::move(item));
  }
========WeightedPriorityQueue=======
void enqueue(K cl, unsigned p, unsigned cost, T&& item) final {
  normal.insert(p, cl, cost, std::move(item));
}
// normal是其内部类Queue的实例，就分析到这了，相应的，取出的操作：
T dequeue() override {
  return normal.pop();
}
```

ShardedOpWQ



ShardedThreadPool

在Ceph中，ShardedThreadPool是一个特殊类型的线程池。普通的线程池中，如果任务之间有互斥性，那么正在处理该任务的两个线程有一个必须等待另一个处理完成后才能处理，从而导致线程的阻塞，性能下降。为了解决这个问题，ShardedThreadPool引入了"Shard"的概念。每个线程对应一个任务队列，所有需要顺序执行的任务都放在同一个线程的任务队列里，全部由该线程执行。这样，就可以避免因任务之间的互斥性导致的线程阻塞，从而提高性能。

ShardedOpWQ

```
1   class ShardedOpWQ : public ShardedThreadPool::ShardedWQ<OpSchedulerItem>
2   {
3     OSD *osd;
4     bool m_fast_shutdown = false;
5   public:
6     ShardedOpWQ(OSD *o,
7         ceph::timespan ti,
8         ceph::timespan si,
9         ShardedThreadPool* tp)
10      : ShardedThreadPool::ShardedWQ<OpSchedulerItem>(ti, si, tp),
11        osd(o) {}
12  void _process(uint32_t thread_index, ceph::heartbeat_handle_d *hb) override;
13  void _enqueue(OpSchedulerItem&& item) override;
14  void _enqueue_front(OpSchedulerItem&& item) override;
15  } op_shardedwq;
```

入队

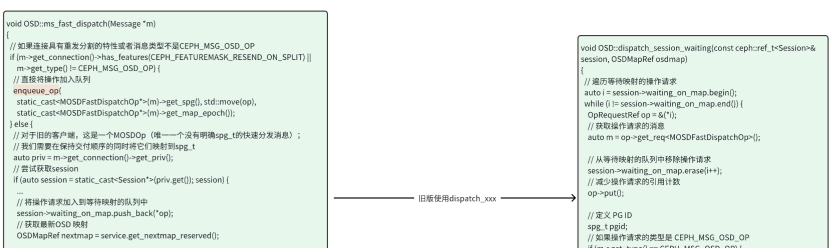
```
op_shardedwq.queue(
  OpSchedulerItem(
    unique_ptr(new PGOpItem(pg, std::move(op))),
    cost, priority, stamp, owner, epoch));
============
void OSD::ShardedOpWQ::_enqueue(OpSchedulerItem&& item) {
  uint32_t shard_index =
    item.get_ordering_token().hash_to_shard(osd->shards.size());

  OSDShard* sdata = osd->shards[shard_index];

  bool empty = true;
  {
    std::lock_guard l{sdata->shard_lock};
    empty = sdata->scheduler->empty();
    sdata->scheduler->enqueue(std::move(item));
  }

  {
    std::lock_guard l{sdata->sdata_wait_lock};
    if (empty) {
      sdata->sdata_cond.notify_all();
    } else if (sdata->waiting_threads) {
      sdata->sdata_cond.notify_one();
    }
  }
}
```

4 读写流程

4.1 从分发请求到完成io

首先要清楚的一些事情：

通信方式采用异步，存储模式使用主备复制，存储方式使用BlueStore



```
  dispatch_session_waiting(session, nextmap);
  ...
```

新版直接调用enqueue_op

```
// 创建一个操作请求并将其加入队列
void OSD::enqueue_op(spg_t pg, OpRequestRef&& op, epoch_t epoch)
{
 if (PGRecoveryMsg::is_recovery_msg(op)) {
  // 是恢复消息
  op_shardedwq.queue(
   OpSchedulerItem(
    unique_ptr<OpSchedulerItem::OpQueueable>(new PGRecoveryMsg(pg, std::move(op))),
    cost, priority, stamp, owner, epoch));
 } else {
  // 不是恢复消息
  op_shardedwq.queue(
   OpSchedulerItem(
    unique_ptr<OpSchedulerItem::OpQueueable>(new PGOpItem(pg, std::move(op))),
    cost, priority, stamp, owner, epoch));
 }
}
```

```
 if (m->get_type() == CEPH_MSG_OSD_OP) {
   pg_t actual_pgid = osdmap->raw_pg_to_pg(
static_cast<const MOSDOp*>(m)->get_pg());
   osdmap->get_primary_shard(actual_pgid, &pgid)
  } else {
   pgid = m->get_spg();
  }
  // 将操作请求加入队列
  enqueue_op(pgid, std::move(op), m->get_map_epoch());
}
```

```
void ShardedThreadPool::queue(T&& item) {
 _enqueue(std::move(item));
}
_enqueue 是纯虚函数
virtual void _enqueue(T&&) = 0;
```

```
// 将操作请求加入到分片操作队列
void OSD::ShardedOpWQ::_enqueue(OpSchedulerItem&& item) {

 // 计算分片索引
 uint32_t shard_index =
  item.get_ordering_token().hash_to_shard(osd->shards.size());

 // 获取 OSD 分片数据
 OSDShard* sdata = osd->shards[shard_index];

 bool empty = true;
 {
  std::lock_guard l(sdata->shard_lock);
  // 检查调度器是否为空
  empty = sdata->scheduler->empty();
  // 将操作请求加入到调度器
  sdata->scheduler->enqueue(std::move(item));
 }
 {
  std::lock_guard l(sdata->sdata_wait_lock);
  if (empty) { // 如果调度器为空，则通知所有等待的线程
   sdata->sdata_cond.notify_all();
  } else if (sdata->waiting_threads) { // 否则，只通知一个等待的线程
   sdata->sdata_cond.notify_one();
  }
 }
}
```

```
void OSD::ShardedOpWQ::_process(uint32_t thread_index, heartbeat_handle_d *hb) {
...
// OSD *osd;
// auto& sdata = osd->shards[shard_index];
// PGRef pg = slot->pg;
// ThreadPool::TPHandle tp_handle(osd->cct, hb, timeout_interval, suicide_interval);
// OSDShardPGSlot *slot;
// auto qi = std::move(slot->to_process.front());

qi.run(osd, sdata, pg, tp_handle);

// qi是OpSchedulerItem
...
}
```

OpSchedulerItem

```
 void run(OSD *osd, OSDShard *sdata,PGRef& pg, ThreadPool::TPHandle &handle) {
  qitem->run(osd, sdata, pg, handle);
  // qitem 是 OpQueueable，而OpQueueable是一个接口，当前场景中，其实现是PGOpItem
 }
```

PGOpItem

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

```
void OSD::dequeue_op(
 PGRef pg,
 OpRequestRef op,
 ThreadPool::TPHandle &handle)
{
 // 获取操作请求的消息
 const Message *m = op->get_req();

 // 获取当前时间
 utime_t now = ceph_clock_now();

 // 设置操作的出队时间
 op->set_dequeued_time(now);

 // 计算操作的延迟
 utime_t latency = now - m->get_recv_stamp();

 // 更新操作的延迟统计
 logger->tinc(l_osd_op_before_dequeue_op_lat, latency);

 // 标记操作已经到达 pg
 op->mark_reached_pg();

 // 执行操作请求
 pg->do_request(op, handle);
}
```

```
void PrimaryLogPG::do_request( OpRequestRef& op,
 ThreadPool::TPHandle &handle)
{
 switch (msg_type) {
 ...
  case CEPH_MSG_OSD_OP:
   do_op(op);
   break;
  ...
 }
}
```

prepare_transaction

```
int PrimaryLogPG::prepare_transaction(OpContext *ctx) {
// 准备事务，将相关的操作封装为事务，存到ctx->op_t之中
 int result = do_osd_ops(ctx, *ctx->ops);
 ...
 if (soid.snap == CEPH_NOSNAP) make_writeable(ctx);

finish_ctx(ctx,
   ctx->new_obs.exists ? pg_log_entry_t::MODIFY :
    pg_log_entry_t::DELETE,
   result);
}
```

```
int PrimaryLogPG::do_osd_ops(OpContext *ctx, vector<OSDOp>& ops) {
 PGTransaction* t = ctx->op_t.get(); // 取出事务类
case CEPH_OSD_OP_WRITEFULL: // 这是覆盖写，还有一个局部写(CEPH_OSD_OP_WRITE)，唯一区别就是多了些偏移量的检查
 ++ctx->num_write;
 result = 0;
 {
  result = check_offset_and_length( // 检查偏移量和长度
   0, op.extent.length, // 偏移量为0，长度为操作的长度
   static_cast<Option::size_t>(osd->osd_max_object_size), get_dpp()); // 对象的最大大小为osd的最大对象大小
  if (result < 0)
   break;

  if (pool.info.has_flag(pg_pool_t::FLAG_WRITE_FADVISE_DONTNEED))
   op.flags = op.flags | CEPH_OSD_OP_FLAG_FADVISE_DONTNEED;

  maybe_create_new_object(ctx);
  if (pool.info.is_erasure()) { // 如果池信息是擦除的
   t->truncate(soid, 0); // 则截断对象到0长度
  } else if (obs.exists && op.extent.length < oi.size) { // 否则，如果对象存在且操作长度小于对象信息的大小
   t->truncate(soid, op.extent.length); // 则截断对象到操作长度
  }
  if (op.extent.length) {
   t->write(soid, 0, op.extent.length, osd_op.indata, op.flags); // 写入数据到缓冲区
  }
  ctx->clean_regions.mark_data_region_dirty(0, // 标记数据区域为脏
   std::max((uint64_t)op.extent.length, oi.size)); // 数据区域的长度为操作长度和对象信息大小的较大者
  write_update_size_and_usage(ctx->delta_stats, oi, ctx->modified_ranges, // 更新大小和使用情况
   0, op.extent.length, true);
 }
 break;
}
```

该case

```
// 除此之外，如果是读操作，就是这条分支、
  case CEPH_OSD_OP_SYNC_READ: // 纠删码不支持同步读，纠删码异步读在后面提到
   if (pool.info.is_erasure()) {
result = -EOPNOTSUPP;
break;
   }
   // fall through
  case CEPH_OSD_OP_READ:
   ++ctx->num_read;
   tracepoint(osd, do_osd_op_pre_read, soid.oid.name.c_str(),
 soid.snap.val, oi.size, oi.truncate_seq, op.extent.offset,
 op.extent.length, op.extent.truncate_size,
 op.extent.truncate_seq);
   if (op_finisher == nullptr) {
 if (!ctx->data_off) {
  ctx->data_off = op.extent.offset;
 }
 result = do_read(ctx, osd_op);
   } else {
 result = op_finisher->execute();
   }
   break;
```

```
void PrimaryLogPG::do_op(OpRequestRef& op)
{
 execute_ctx(ctx);
}
```

```
void PrimaryLogPG::execute_ctx(OpContext *ctx){
ctx->op_t.reset(new PGTransaction); // 创建一个新的事务

int result = prepare_transaction(ctx);
// 如果是异步读
 bool pending_async_reads = !ctx->pending_async_reads.empty();
 if (result == -EINPROGRESS || pending_async_reads) {
  // come back later.
  if (pending_async_reads) {
   ceph_assert(pool.info.is_erasure());
   in_progress_async_reads.push_back(make_pair(op, ctx));
   ctx->start_async_reads(this);
  }
  return;
 }

// 创建回复
ctx->reply = new MOSDOpReply(m, result, get_osdmap_epoch(), 0, ignore_out_data);

// 同步读
 if ((ctx->op_t->empty() || result < 0) && !ctx->update_log_only) {
  if (result >= 0)
   do_osd_op_effects(ctx, m->get_connection());
  complete_read_ctx(result, ctx);
  return;
 }
```

complete_read_ctx

```
//这个函数整理读取的结果，打包好发送给客户端
void PrimaryLogPG::complete_read_ctx(int result, OpContext *ctx)
{
 auto m = ctx->op->get_req<MOSDOp>();
 ceph_assert(ctx->async_reads_complete());

 for (auto p = ctx->ops->begin();
   p != ctx->ops->end() && result >= 0; ++p) {
  if (p->rval < 0 && !(p->op.flags & CEPH_OSD_OP_FLAG_FAILOK)) {
   result = p->rval;
   break;
  }
  ctx->bytes_read += p->outdata.length();
 }
 ctx->reply->get_header().data_off = (ctx->data_off ? *ctx->data_off : 0);

 MOSDOpReply *reply = ctx->reply;
 ctx->reply = nullptr;

 if (result >= 0) {
  if (!ctx->ignore_log_op_stats) {
   log_op_stats(*ctx->op, ctx->bytes_written, ctx->bytes_read);

   publish_stats_to_osd();
  }

  if (ctx->obs) {
   reply->set_reply_versions(eversion_t(), ctx->obs->oi.user_version);
  } else {
   reply->set_reply_versions(eversion_t(), ctx->user_at_version);
  }
 } else if (result == -ENOENT) {
  reply->set_enoent_reply_versions(info.last_update, info.last_user_version);
 }

 reply->set_result(result);
 reply->add_flags(CEPH_OSD_FLAG_ACK | CEPH_OSD_FLAG_ONDISK);
 osd->send_message_osd_client(reply, m->get_connection());
 close_op_ctx(ctx);
}
```

start_async_reads

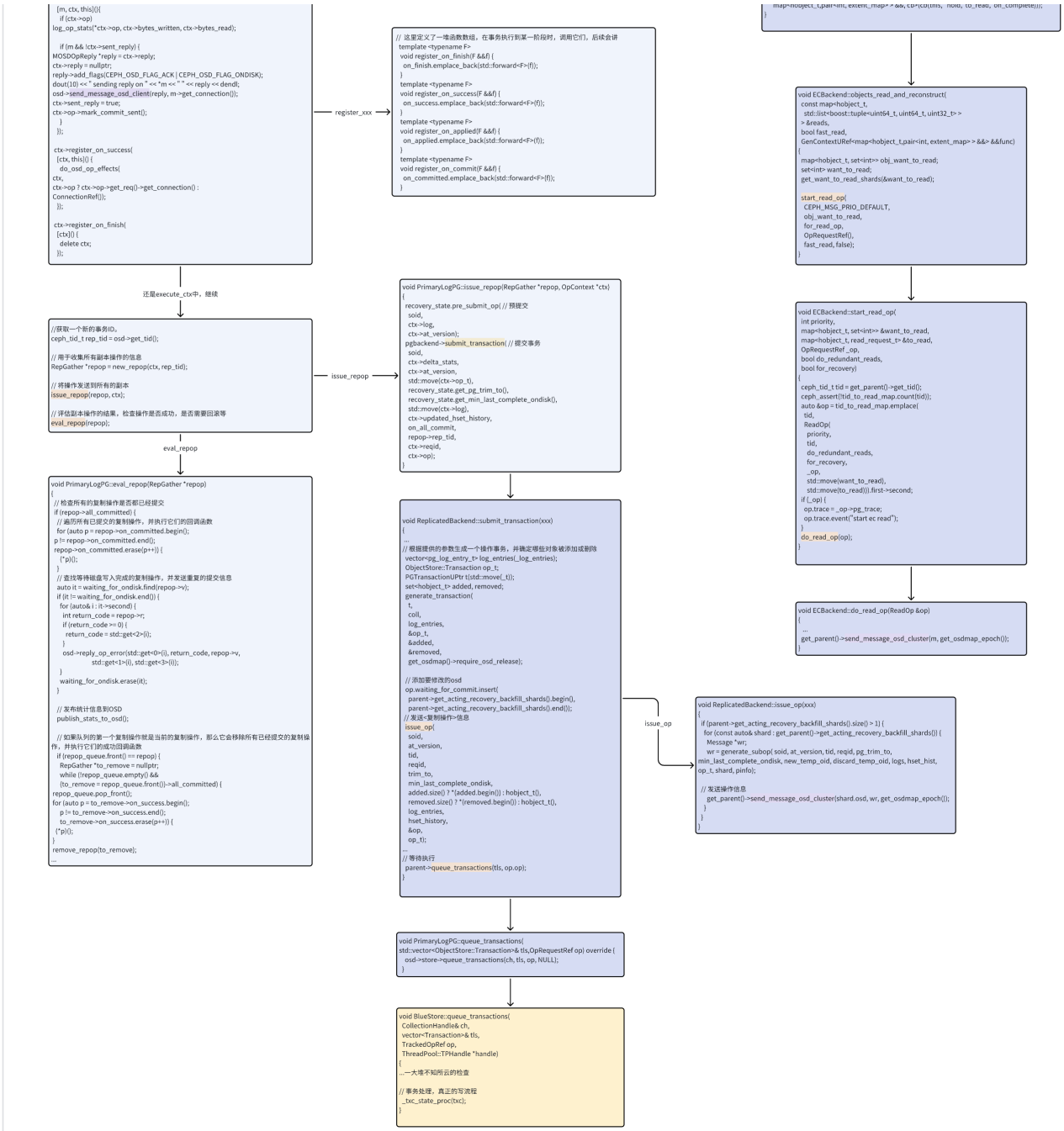
默认情况下，使用的是主备复制，不会走到这条分支

```
void PrimaryLogPG::OpContext::start_async_reads(PrimaryLogPG *pg)
{
 inflightreads = 1;
 list<pair<boost::tuple<uint64_t, uint64_t, unsigned>,
    pair<bufferlist*, Context*> > > in;
 in.swap(pending_async_reads);
 pg->pgbackend->objects_read_async(
  obc->obs.oi.soid,
  in,
  new OnReadComplete(pg, this), pg->get_pool().fast_read);
}
```

```
// 采用主备复制策略的不支持异步读
void ReplicatedBackend::objects_read_async(...)
{
 ceph_abort_msg("async read is not used by replica pool");
}
// 采用纠删码的可以
void ECBackend::objects_read_async(
 const hobject_t &hoid,
 const list<pair<boost::tuple<uint64_t, uint64_t, uint32_t>,
        pair<bufferlist*, Context*> > > &to_read,
 Context *on_complete,
 bool fast_read)
{
 ...
 objects_read_and_reconstruct(
  reads,
  fast_read,
  make_gen_lambda_context<
```

一个框装不下，继续

```
recovery_state.update_trim_to(); // 更改需要修改的快照的状态


ctx->register_on_commit(
```



4.2 写操作同步

4.1写操作的最后，issue_op使用send_message_osd_cluster发送了同步消息，这节来观察这个操作的两端是怎么工作的

这函数有四个形式的重载，最终调用的都是Connection::send_message，而send_message实际又由AsyncConnection实现

```
1 void OSDService::send_message_osd_cluster(int peer, Message *m, epoch_t
  from_epoch)
2 {
3   OSDMapRef next_map = get_nextmap_reserved();
4   // service map is always newer/newest
5   ceph_assert(from_epoch <= next_map->get_epoch());
```

```
6
7   if (next_map->is_down(peer) ||
8       next_map->get_info(peer).up_from > from_epoch) {
9     m->put();
10    release_map(next_map);
11    return;
12  }
13  ConnectionRef peer_con;
14  if (peer == whoami) {
15    peer_con = osd->cluster_messenger->get_loopback_connection();
16  } else {
17    peer_con = osd->cluster_messenger->connect_to_osd(
18    next_map->get_cluster_addrs(peer), false, true);
19  }
20  maybe_share_map(peer_con.get(), next_map);
21  peer_con->send_message(m);
22  release_map(next_map);
23 }
24
25 void OSDService::send_message_osd_cluster(std::vector<std::pair<int,
   Message*>>& messages, epoch_t from_epoch)
26 {
27  OSDMapRef next_map = get_nextmap_reserved();
28  ceph_assert(from_epoch <= next_map->get_epoch());
29
30  for (auto& iter : messages) {
31    if (next_map->is_down(iter.first) ||
32    next_map->get_info(iter.first).up_from > from_epoch) {
33      iter.second->put();
34      continue;
35    }
36    ConnectionRef peer_con;
37    if (iter.first == whoami) {
38      peer_con = osd->cluster_messenger->get_loopback_connection();
39    } else {
40      peer_con = osd->cluster_messenger->connect_to_osd(
41      next_map->get_cluster_addrs(iter.first), false, true);
42    }
43    maybe_share_map(peer_con.get(), next_map);
44    peer_con->send_message(iter.second);
45  }
46  release_map(next_map);
47 }
48
49 void send_message_osd_cluster(MessageRef m, Connection *con) {
50  con->send_message2(std::move(m));
51  /*
```

```
52      virtual int send_message2(MessageRef m)
53    {
54      return send_message(m.detach());
55    }
56    */
57  }
58  void send_message_osd_cluster(Message *m, const ConnectionRef& con) {
59    con->send_message(m);
60  }
```

send_messgae之后的流程见<5.7 发送消息>

4.3 接收消息

4.1的流程开始是ms_fast_dispatch，这是谁调用的?

在<5.5 add_fast_dispatch_tail>中，讲了add_dispatcher_tail的流程。类似的还有add_dispatcher_head

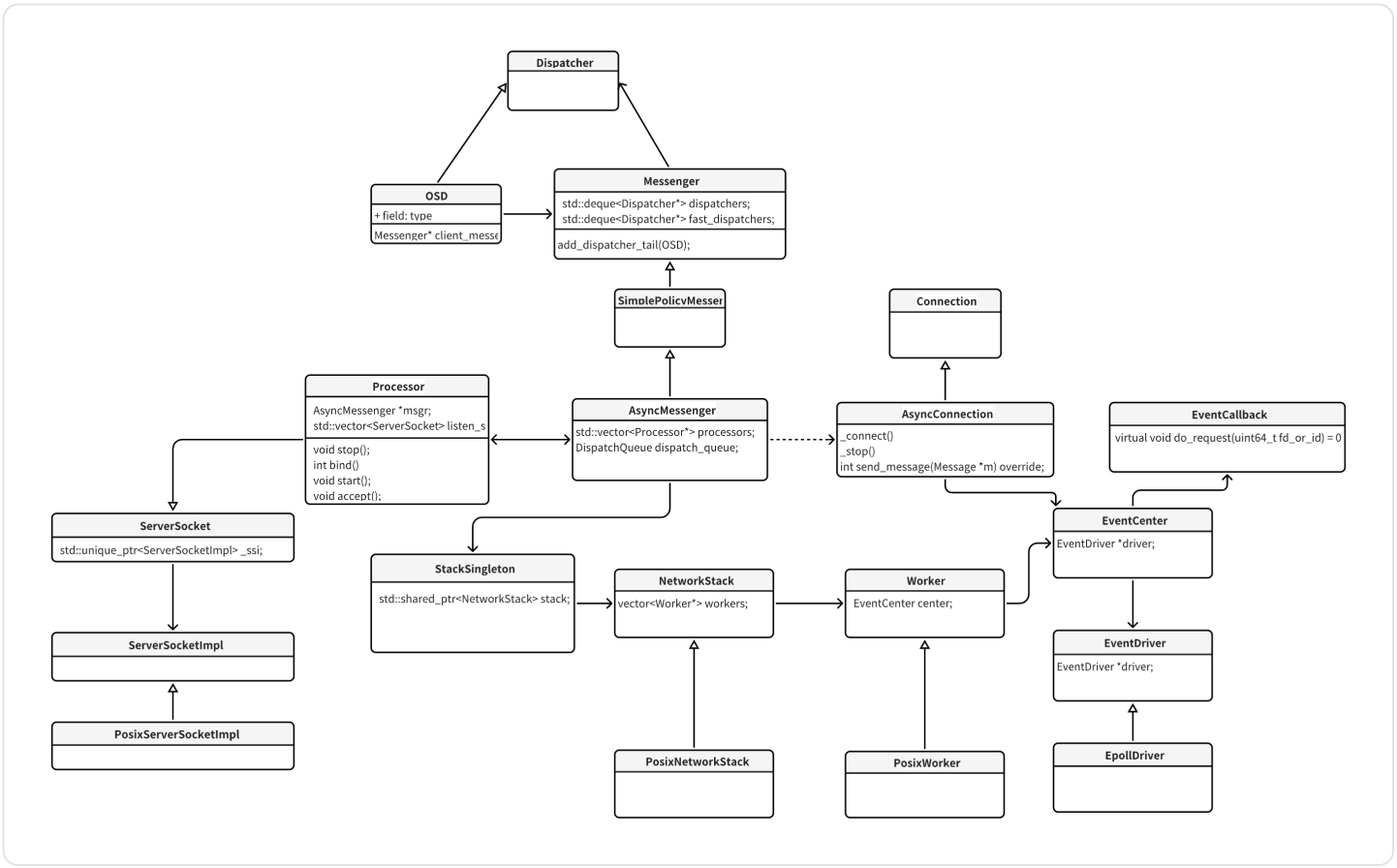
```
1   int OSD::init() {
2     cluster_messenger->add_dispatcher_head(this);
3   }
4   void add_dispatcher_head(Dispatcher *d) {
5     bool first = dispatchers.empty();
6     dispatchers.push_front(d);
7     if (d->ms_can_fast_dispatch_any())
8       fast_dispatchers.push_front(d);
9     if (first)
10      ready();
11  }
```

这里OSD将自身绑定到了一个fast_dispatchers中，然后cluster_messager就能调用osd的ms_fast_dispatch

具体过程在<5.5 add_fast_dispatch_tail>中说明了。

5 通信模块

5.1 OSD 通信类图



OSD注册的Messenger实例列表

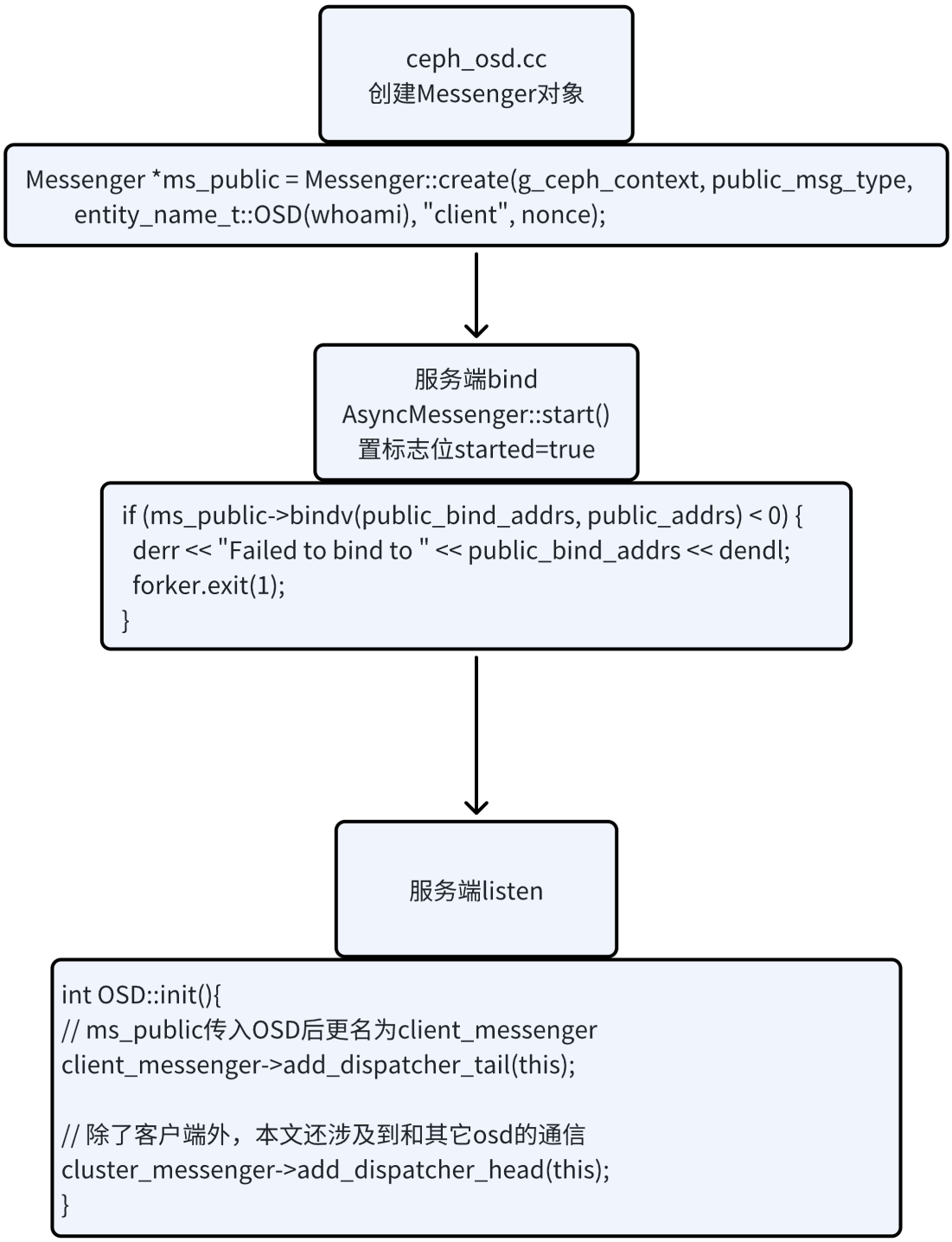
编号	Messenger实例名称	作用
1	*ms_public	用来处理OSD和Client之间的消息
2	*ms_cluster	用来处理OSD和集群之间的消息
3	*ms_hbclient	用来处理OSD和其它OSD保持心跳的消息
4	*ms_hb_back_server	用来处理OSD接收心跳消息
5	*ms_hb_front_server	用来处理OSD发送心跳消息
6	*ms_objecter	用来处理OSD和Objecter之间的消息

5.2 主流程

最简单的服务器是这样的模式：

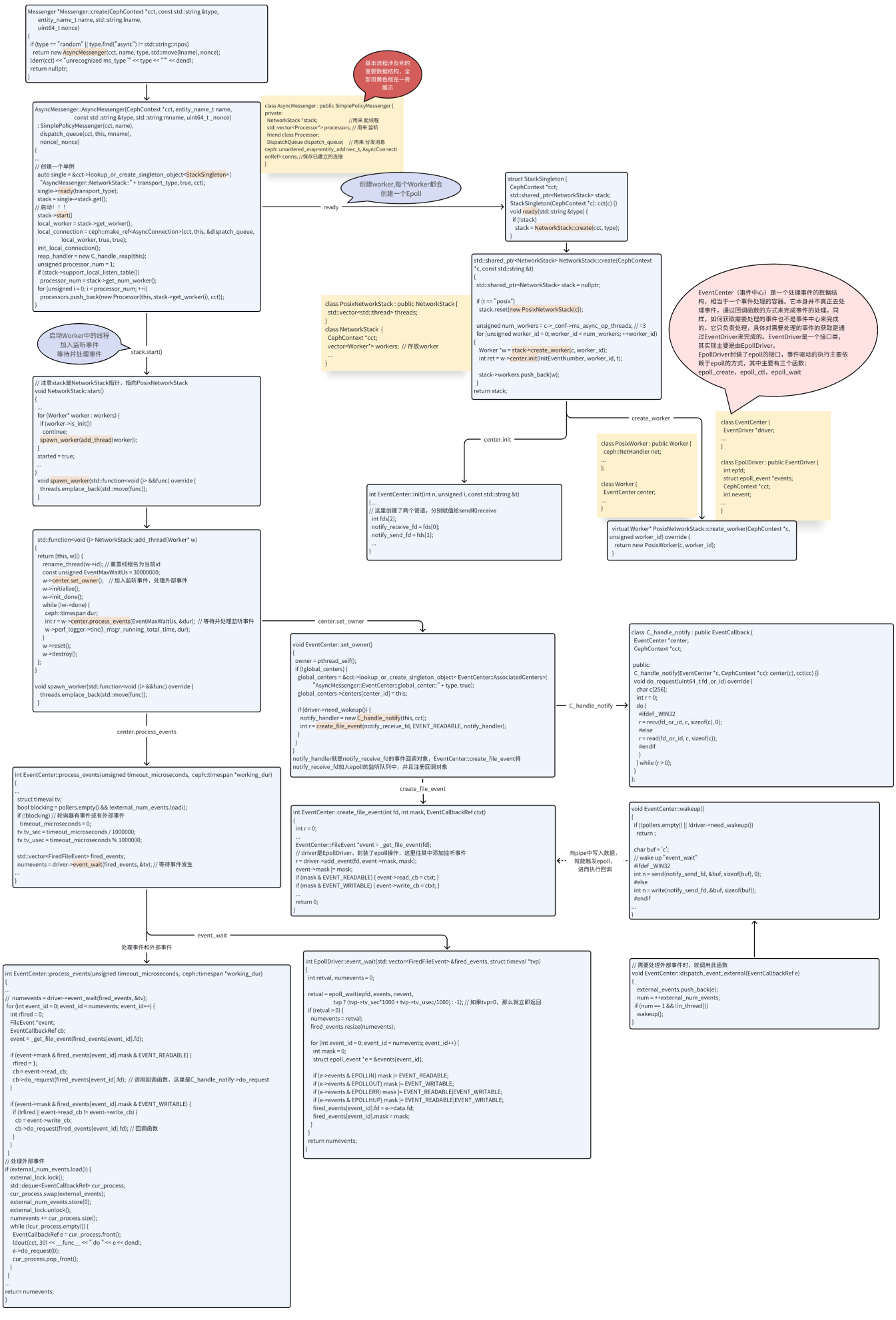
1. **创建监听套接字**：使用 `socket` 函数创建一个套接字。
2. **绑定本机IP和端口**：使用 `bind` 函数将套接字绑定到特定的IP地址和端口上。
3. **监听**：使用 `listen` 函数开始监听这个套接字。
4. **创建epoll实例**：使用 `epoll_create` 函数创建一个epoll实例。
5. **注册文件描述符**：使用 `epoll_ctl` 函数将监听套接字注册到epoll实例中，监听 `EPOLLIN` 事件（新的连接请求）。
6. **等待事件**：使用 `epoll_wait` 函数等待注册在epoll实例上的文件描述符上的事件发生。
7. **处理事件**：当 `epoll_wait` 返回时，遍历所有就绪的事件，对于每一个事件：
   - 如果是监听套接字上的事件，那么使用 `accept` 函数接受新的连接，然后使用 `epoll_ctl` 将新的连接套接字注册到epoll实例中，监听 `EPOLLIN` 和 `EPOLLOUT` 事件。
   - 如果是连接套接字上的 `EPOLLIN` 事件，那么读取数据，处理请求。
   - 如果是连接套接字上的 `EPOLLOUT` 事件，那么写入数据，发送响应。
8. **关闭连接**：当连接结束后，使用 `close` 函数关闭连接套接字，并使用 `epoll_ctl` 将其从epoll实例中删除。

ceph的通信模块采用事件驱动的模式，先创建事件中心，事件中心接收自定义的各类事件，触发事件响应时调用相应的回调函数，松散耦合。



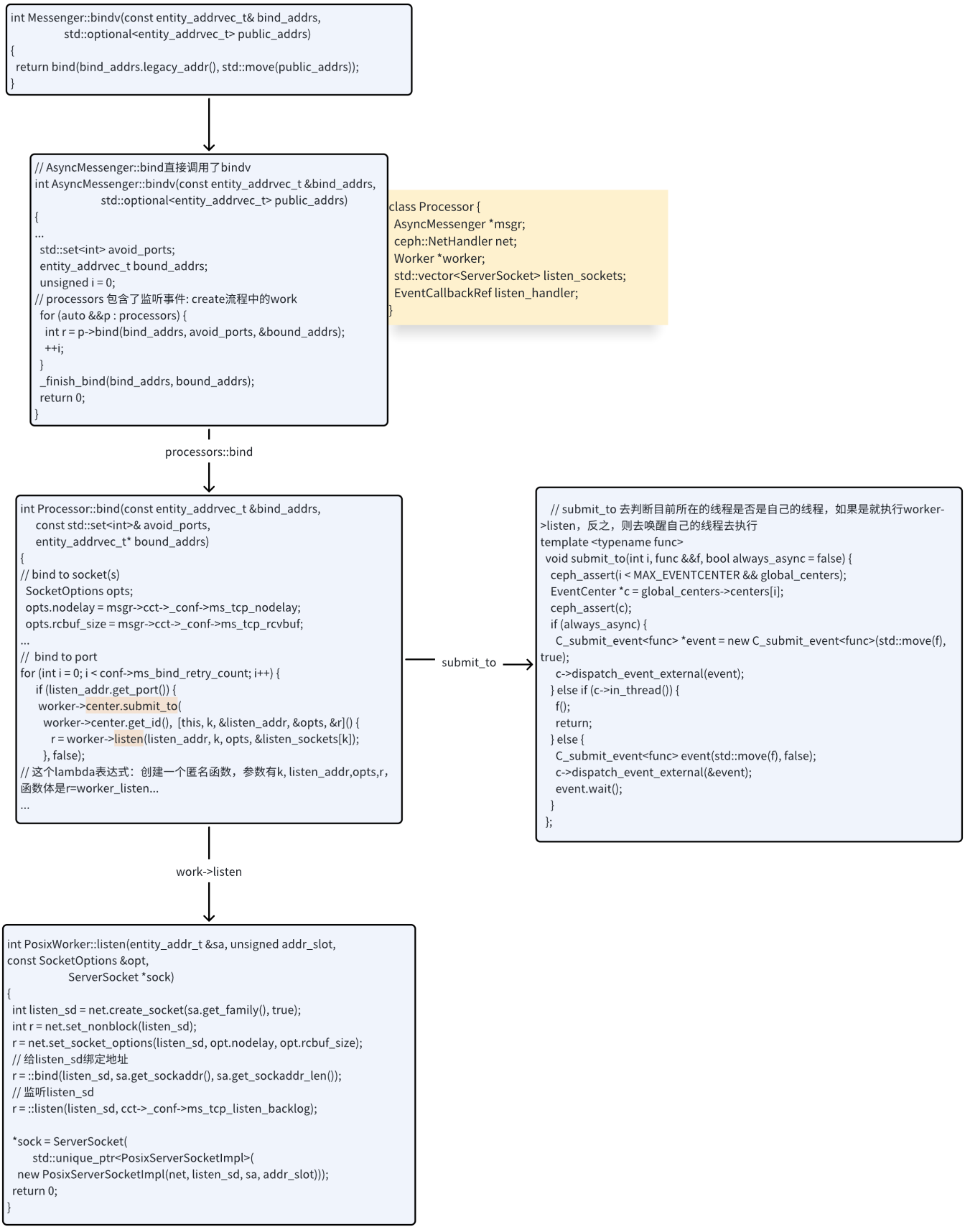
5.3 create 流程

创建消息模块、初始化、启动事件监听，但是此时还没有事件加入。

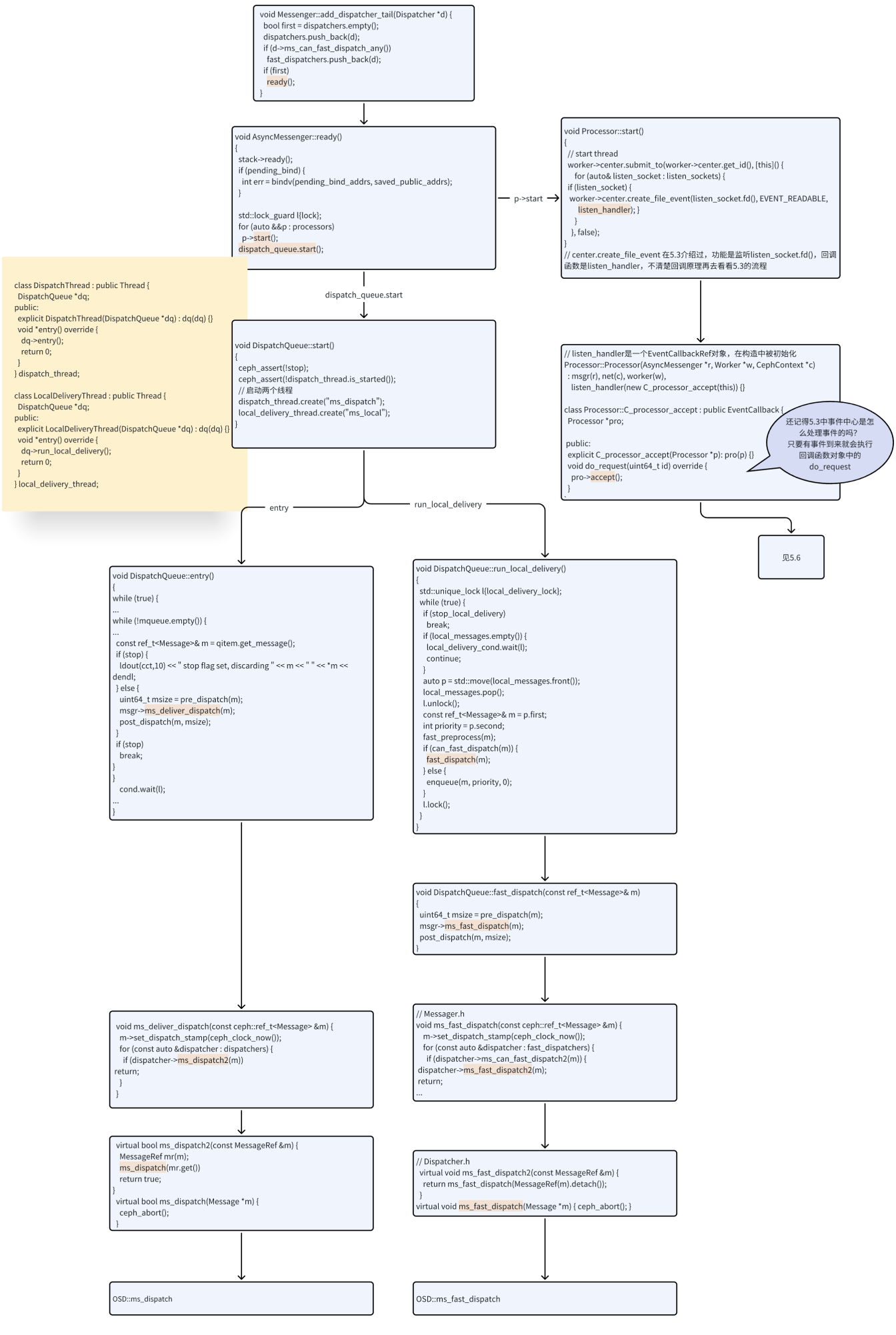

Messenger *Messenger::create(CephContext *cct, const std::string &type,
entity_name_t name, std::string lname,
uint64_t nonce)
{
if (type == "random" || type.find("async") != std::string::npos)
return new AsyncMessenger(cct, name, type, std::move(lname), nonce);
lderr(cct) << " unrecognized ms_type '" << type << "'" << dendl;
return nullptr;
}
基本流程涉及到的重要数据结构，全部用黄色框在一旁展示
class AsyncMessenger : public SimplePolicyMessenger {
private:
NetworkStack *stack; //用来 起线程
std::vector<Processor*> processors; // 用来 监听
friend class Processor;
DispatchQueue dispatch_queue; // 用来 分发消息
ceph::unordered_map<entity_addrvec_t, AsyncConnectionRef> conns; //保存已建立的连接
}
AsyncMessenger::AsyncMessenger(CephContext *cct, entity_name_t name,
const std::string &type, std::string mname, uint64_t _nonce)
: SimplePolicyMessenger(cct, name),
dispatch_queue(cct, this, mname),
nonce(_nonce)
{
...
// 创建一个单例
auto single = &cct->lookup_or_create_singleton_object<StackSingleton>(
"AsyncMessenger::NetworkStack::" + transport_type, true, cct);
single->ready(transport_type);
stack = single->stack.get();
// 启动！！！
stack->start()
local_worker = stack->get_worker();
local_connection = ceph::make_ref<AsyncConnection>(cct, this, &dispatch_queue,
local_worker, true, true);
init_local_connection();
reap_handler = new C_handle_reap(this);
unsigned processor_num = 1;
if (stack->support_local_listen_table())
processor_num = stack->get_num_worker();
for (unsigned i = 0; i < processor_num; ++i)
processors.push_back(new Processor(this, stack->get_worker(i), cct));
}
ready
创建worker,每个Worker都会创建一个Epoll
struct StackSingleton {
CephContext *cct;
std::shared_ptr<NetworkStack> stack;
StackSingleton(CephContext *c): cct(c) {}
void ready(std::string &type) {
if (!stack)
stack = NetworkStack::create(cct, type);
}
std::shared_ptr<NetworkStack> NetworkStack::create(CephContext *c, const std::string &t)
{
std::shared_ptr<NetworkStack> stack = nullptr;
if (t == "posix")
stack.reset(new PosixNetworkStack(c));
unsigned num_workers = c->_conf->ms_async_op_threads; // =3
for (unsigned worker_id = 0; worker_id < num_workers; ++worker_id)
{
Worker *w = stack->create_worker(c, worker_id);
int ret = w->center.init(InitEventNumber, worker_id, t);
stack->workers.push_back(w);
}
return stack;
}
class PosixNetworkStack : public NetworkStack {
std::vector<std::thread> threads;
}
class NetworkStack {
CephContext *cct;
vector<Worker*> workers; // 存放worker
...
}
EventCenter（事件中心）是一个处理事件的数据结构，相当于一个事件处理的容器。它本身并不真正去处理事件，通过回调函数的方式来完成事件的处理。同样，如何获取需要处理的事件也不是事件中心来完成的，它只负责处理，具体对需要处理的事件的获取是通过EventDriver来完成的。EventDriver是一个接口类，其实现主要是由EpollDriver。
EpollDriver封装了epoll的接口，事件驱动的执行主要依赖于epoll的方式，其中主要有三个函数：
epoll_create，epoll_ctl，epoll_wait
class EventCenter {
EventDriver *driver;
...
}
class EpollDriver : public EventDriver {
int epfd;
struct epoll_event *events;
CephContext *cct;
int nevent;
...
}
create_worker
center.init
class PosixWorker : public Worker {
ceph::NetHandler net;
...
};
class Worker {
EventCenter center;
...
}
virtual Worker* PosixNetworkStack::create_worker(CephContext *c, unsigned worker_id) override {
return new PosixWorker(c, worker_id);
}
int EventCenter::init(int n, unsigned i, const std::string &t)
{ ...
// 这里创建了两个管道，分别赋值给send和receive
int fds[2];
notify_receive_fd = fds[0];
notify_send_fd = fds[1];
...
}
启动Worker中的线程
加入监听事件
等待并处理事件
stack.start()
// 注意stack是NetworkStack指针，指向PosixNetworkStack
void NetworkStack::start()
{
...
for (Worker* worker : workers) {
if (worker->is_init())
continue;
spawn_worker(add_thread(worker));
}
started = true;
...
}
void spawn_worker(std::function<void ()> &&func) override {
threads.emplace_back(std::move(func));
}
std::function<void ()> NetworkStack::add_thread(Worker* w)
{
return [this, w]() {
rename_thread(w->id); // 重置线程名为当前id
const unsigned EventMaxWaitUs = 30000000;
w->center.set_owner(); // 加入监听事件，处理外部事件
w->initialize();
w->init_done();
while (!w->done) {
ceph::timespan dur;
int r = w->center.process_events(EventMaxWaitUs, &dur); // 等待并处理监听事件
w->perf_logger->tinc(l_msgr_running_total_time, dur);
}
w->reset();
w->destroy();
};
}
void spawn_worker(std::function<void ()> &&func) override {
threads.emplace_back(std::move(func));
}
center.set_owner
void EventCenter::set_owner()
{
owner = pthread_self();
if (!global_centers) {
global_centers = &cct->lookup_or_create_singleton_object< EventCenter::AssociatedCenters>(
"AsyncMessenger::EventCenter::global_center::" + type, true);
global_centers->centers[center_id] = this;
if (driver->need_wakeup()) {
notify_handler = new C_handle_notify(this, cct);
int r = create_file_event(notify_receive_fd, EVENT_READABLE, notify_handler);
}
}
notify_handler就是notify_receive_fd的事件回调对象，EventCenter::create_file_event将notify_receive_fd加入epoll的监听队列中，并且注册回调对象
C_handle_notify
class C_handle_notify : public EventCallback {
EventCenter *center;
CephContext *cct;
public:
C_handle_notify(EventCenter *c, CephContext *cc): center(c), cct(cc) {}
void do_request(uint64_t fd_or_id) override {
char c[256];
int r = 0;
do {
#ifdef _WIN32
r = recv(fd_or_id, c, sizeof(c), 0);
#else
r = read(fd_or_id, c, sizeof(c));
#endif
} while (r > 0);
}
};
center.process_events
int EventCenter::process_events(unsigned timeout_microseconds, ceph::timespan *working_dur)
{
...
struct timeval tv;
bool blocking = pollers.empty() && !external_num_events.load();
if (!blocking) // 轮询器有事件或有外部事件
timeout_microseconds = 0;
tv.tv_sec = timeout_microseconds / 1000000;
tv.tv_usec = timeout_microseconds % 1000000;
std::vector<FiredFileEvent> fired_events;
numevents = driver->event_wait(fired_events, &tv); // 等待事件发生
...
}
create_file_event
int EventCenter::create_file_event(int fd, int mask, EventCallbackRef ctxt)
{
int r = 0;
...
EventCenter::FileEvent *event = _get_file_event(fd);
// driver是EpollDriver，封装了epoll操作，这里往其中添加监听事件
r = driver->add_event(fd, event->mask, mask);
event->mask |= mask;
if (mask & EVENT_READABLE) { event->read_cb = ctxt; }
if (mask & EVENT_WRITABLE) { event->write_cb = ctxt; }
...
return 0;
}
向pipe中写入数据，就能触发epoll，进而执行回调
void EventCenter::wakeup()
{
if (!pollers.empty() || !driver->need_wakeup())
return ;
char buf = 'c';
// wake up "event_wait"
#ifdef _WIN32
int n = send(notify_send_fd, &buf, sizeof(buf), 0);
#else
int n = write(notify_send_fd, &buf, sizeof(buf));
#endif
...
}
event_wait
处理事件和外部事件
int EventCenter::process_events(unsigned timeout_microseconds, ceph::timespan *working_dur)
{
...
// numevents = driver->event_wait(fired_events, &tv);
for (int event_id = 0; event_id < numevents; event_id++) {
int rfired = 0;
FileEvent *event;
EventCallbackRef cb;
event = _get_file_event(fired_events[event_id].fd);
if (event->mask & fired_events[event_id].mask & EVENT_READABLE) {
rfired = 1;
cb = event->read_cb;
cb->do_request(fired_events[event_id].fd); // 调用回调函数，这里是C_handle_notify->do_request
}
if (event->mask & fired_events[event_id].mask & EVENT_WRITABLE) {
if (!rfired || event->read_cb != event->write_cb) {
cb = event->write_cb;
cb->do_request(fired_events[event_id].fd); // 回调函数
}
}
}
// 处理外部事件
if (external_num_events.load()) {
external_lock.lock();
std::deque<EventCallbackRef> cur_process;
cur_process.swap(external_events);
external_num_events.store(0);
external_lock.unlock();
numevents += cur_process.size();
while (!cur_process.empty()) {
EventCallbackRef e = cur_process.front();
ldout(cct, 30) << __func__ << " do " << e << dendl;
e->do_request(0);
cur_process.pop_front();
}
}
...
return numevents;
}
int EpollDriver::event_wait(std::vector<FiredFileEvent> &fired_events, struct timeval *tvp)
{
int retval, numevents = 0;
retval = epoll_wait(epfd, events, nevent,
tvp ? (tvp->tv_sec*1000 + tvp->tv_usec/1000) : -1); // 如果tvp=0，那么就立即返回
if (retval > 0) {
numevents = retval;
fired_events.resize(numevents);
for (int event_id = 0; event_id < numevents; event_id++) {
int mask = 0;
struct epoll_event *e = &events[event_id];
if (e->events & EPOLLIN) mask |= EVENT_READABLE;
if (e->events & EPOLLOUT) mask |= EVENT_WRITABLE;
if (e->events & EPOLLERR) mask |= EVENT_READABLE|EVENT_WRITABLE;
if (e->events & EPOLLHUP) mask |= EVENT_READABLE|EVENT_WRITABLE;
fired_events[event_id].fd = e->data.fd;
fired_events[event_id].mask = mask;
}
}
return numevents;
}
// 需要处理外部事件时，就调用此函数
void EventCenter::dispatch_event_external(EventCallbackRef e)
{
external_events.push_back(e);
num = ++external_num_events;
if (num == 1 && !in_thread())
wakeup();
}

5.4 bindv 流程

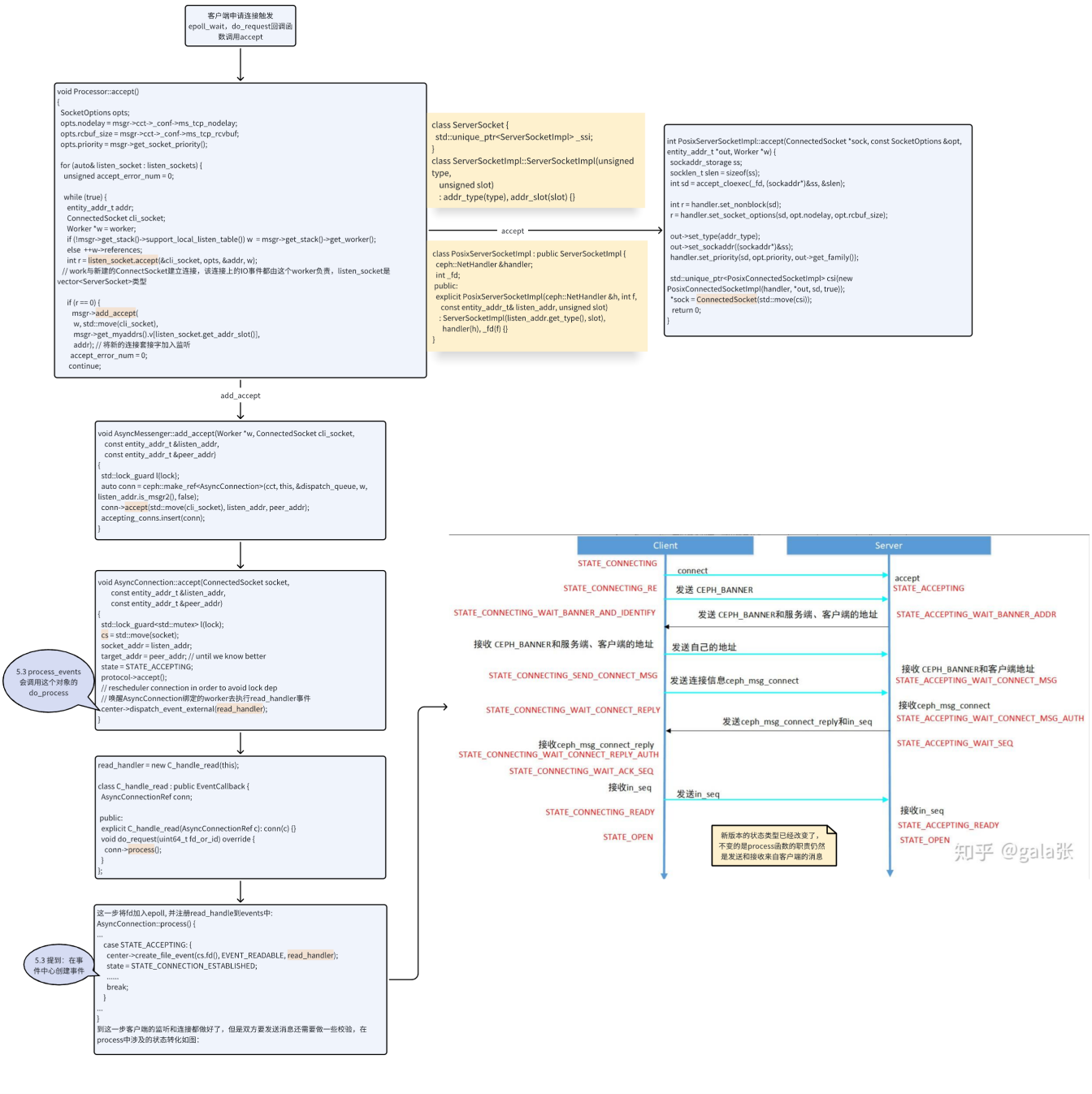
绑定本机ip



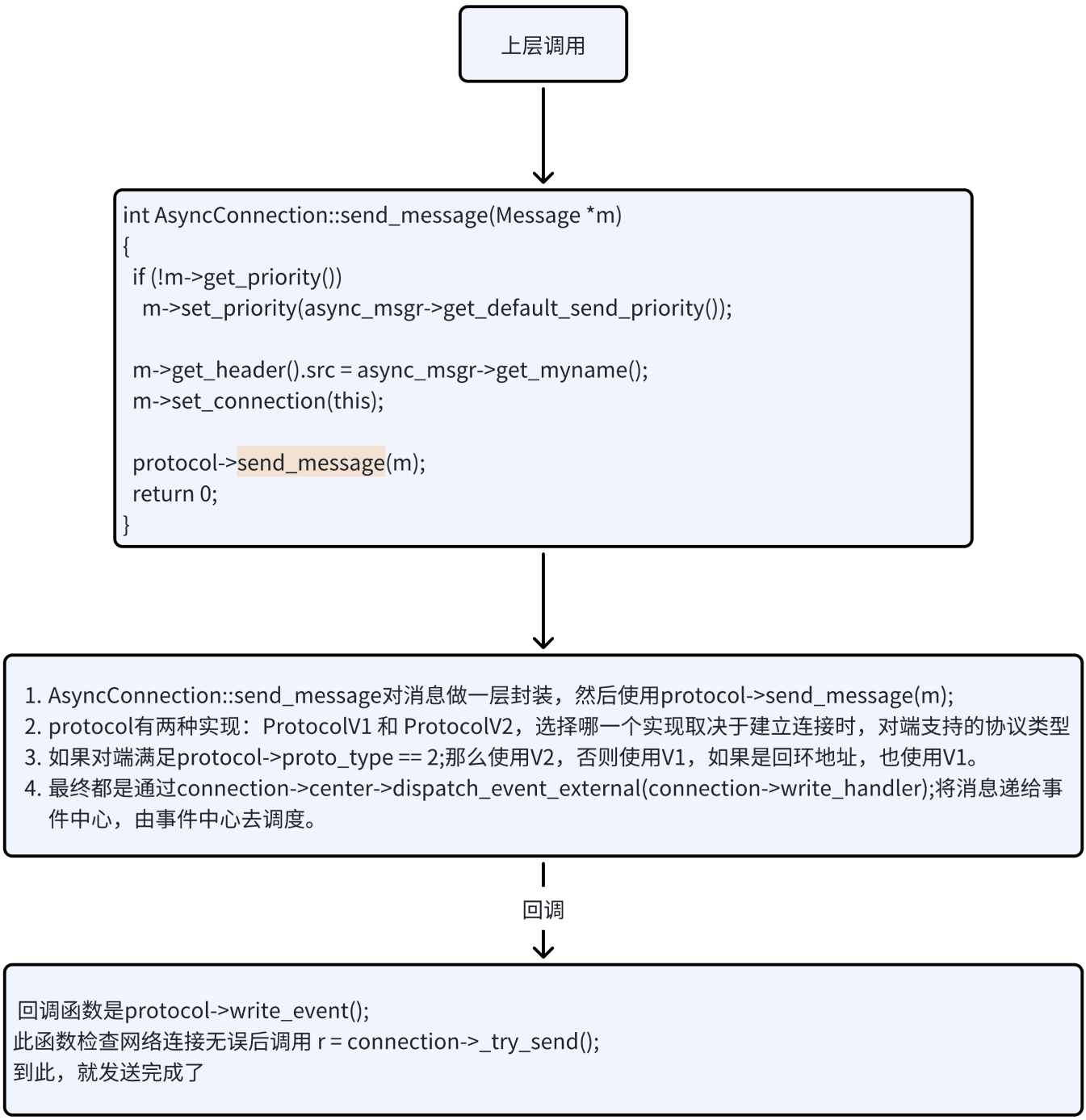
5.5 add_dispatcher_tail 流程


void Messenger::add_dispatcher_tail(Dispatcher *d) {
bool first = dispatchers.empty();
dispatchers.push_back(d);
if (d->ms_can_fast_dispatch_any())
fast_dispatchers.push_back(d);
if (first)
ready();
}
void AsyncMessenger::ready()
{
stack->ready();
if (pending_bind) {
int err = bindv(pending_bind_addrs, saved_public_addrs);
}
std::lock_guard l{lock};
for (auto &&p : processors)
p->start();
dispatch_queue.start();
p->start
void Processor::start()
{
// start thread
worker->center.submit_to(worker->center.get_id(), [this]() {
for (auto& listen_socket : listen_sockets) {
if (listen_socket) {
worker->center.create_file_event(listen_socket.fd(), EVENT_READABLE,
listen_handler); }
}
}, false);
}
// center.create_file_event 在5.3介绍过，功能是监听listen_socket.fd()，回调函数是listen_handler，不清楚回调原理再去看看5.3的流程
class DispatchThread : public Thread {
DispatchQueue *dq;
public:
explicit DispatchThread(DispatchQueue *dq) : dq(dq) {}
void *entry() override {
dq->entry();
return 0;
}
} dispatch_thread;
class LocalDeliveryThread : public Thread {
DispatchQueue *dq;
public:
explicit LocalDeliveryThread(DispatchQueue *dq) : dq(dq) {}
void *entry() override {
dq->run_local_delivery();
return 0;
}
} local_delivery_thread;
dispatch_queue.start
void DispatchQueue::start()
{
ceph_assert(!stop);
ceph_assert(!dispatch_thread.is_started());
// 启动两个线程
dispatch_thread.create("ms_dispatch");
local_delivery_thread.create("ms_local");
}
// listen_handler是一个EventCallbackRef对象，在构造中被初始化
Processor::Processor(AsyncMessenger *r, Worker *w, CephContext *c)
: msgr(r), net(c), worker(w),
listen_handler(new C_processor_accept(this)) {}
class Processor::C_processor_accept : public EventCallback {
Processor *pro;
public:
explicit C_processor_accept(Processor *p): pro(p) {}
void do_request(uint64_t id) override {
pro->accept();
}
还记得5.3中事件中心是怎么处理事件的吗？只要有事件到来就会执行回调函数对象中的do_request
见5.6
entry
run_local_delivery
void DispatchQueue::entry()
{
while (true) {
...
while (!mqueue.empty()) {
...
const ref_t<Message>& m = qitem.get_message();
if (stop) {
ldout(cct,10) << " stop flag set, discarding " << m << " " << *m <<
dendl;
} else {
uint64_t msize = pre_dispatch(m);
msgr->ms_deliver_dispatch(m);
post_dispatch(m, msize);
}
if (stop)
break;
}
}
cond.wait(l);
...
}
void DispatchQueue::run_local_delivery()
{
std::unique_lock l{local_delivery_lock};
while (true) {
if (stop_local_delivery)
break;
if (local_messages.empty()) {
local_delivery_cond.wait(l);
continue;
}
auto p = std::move(local_messages.front());
local_messages.pop();
l.unlock();
const ref_t<Message>& m = p.first;
int priority = p.second;
fast_preprocess(m);
if (can_fast_dispatch(m)) {
fast_dispatch(m);
} else {
enqueue(m, priority, 0);
}
l.lock();
}
}
void DispatchQueue::fast_dispatch(const ref_t<Message>& m)
{
uint64_t msize = pre_dispatch(m);
msgr->ms_fast_dispatch(m);
post_dispatch(m, msize);
}
void ms_deliver_dispatch(const ceph::ref_t<Message> &m) {
m->set_dispatch_stamp(ceph_clock_now());
for (const auto &dispatcher : dispatchers) {
if (dispatcher->ms_dispatch2(m))
return;
}
}
// Messager.h
void ms_fast_dispatch(const ceph::ref_t<Message> &m) {
m->set_dispatch_stamp(ceph_clock_now());
for (const auto &dispatcher : fast_dispatchers) {
if (dispatcher->ms_can_fast_dispatch2(m)) {
dispatcher->ms_fast_dispatch2(m);
return;
...
virtual bool ms_dispatch2(const MessageRef &m) {
MessageRef mr(m);
ms_dispatch(mr.get())
return true;
}
virtual bool ms_dispatch(Message *m) {
ceph_abort();
}
// Dispatcher.h
virtual void ms_fast_dispatch2(const MessageRef &m) {
return ms_fast_dispatch(MessageRef(m).detach());
}
virtual void ms_fast_dispatch(Message *m) { ceph_abort(); }
OSD::ms_dispatch
OSD::ms_fast_dispatch

5.6 接受新的连接



5.7 发送消息



参考

【ceph】AsyncMessenger 网络模块总结--编辑中 - bdy - 博客园

【ceph】CEPH源码解析:读写流程_ceph源码分析-CSDN博客

ceph的数据存储之路(5) -----osd数据处理 - 一只小江的个人空间 - OSCHINA - 中文开源技术交流社区