数据修复

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

PG日志记录了PG内对象的所有操作，包括创建、修改和删除等。每个日志条目（log entry）包含了执行操作时的元数据，例如对象的版本号（eversion）、操作类型、对象的元数据（如大小、时间戳等）,这些日志条目按按照操作发生的顺序排列，形成了一个操作历史记录。

当Ceph执行数据恢复或同步操作时，它会参考PG日志来确定对象的当前状态。例如，在Recovery阶段，Ceph会使用PG日志来恢复缺失的对象副本，确保所有副本都是最新的。在这个过程中，PG日志充当了元数据变更的记录器，帮助Ceph维护数据的一致性和完整性。

Recovery是依据PG日志中的缺失记录来修复不一致的对象。Backfill是PG通过重新扫描所有的对象，对比发现缺失的对象，通过整体拷贝来修复。当一个OSD失效时间过长导致无法根据PG日志来修复，或者新加入的OSD导致数据迁移时，就会启动Backfill过程。

资源预约

如果有大量pg同时进行修复，那可能会影响到客户端读写的性能，需要对pg同时修复的数量进行限制，于是修复前需要进行资源的预约。

```
void request_reservation(
    T item,                   ///< [in] reservation key
    Context *on_reserved,     ///< [in] callback to be called on reservation
    unsigned prio,            ///< [in] priority
    Context *on_preempt = 0 ///< [in] callback to be called if we are preempted (optional)
) {
    std::lock_guard l(lock);
    // 资源关键字、优先级、预留成功时的回调、可选的抢占回调
    Reservation r(item, prio, on_reserved, on_preempt);

    // 资源优先队列
    queues[prio].push_back(r);
    // item -> 资源位置
    queue_pointers.insert(std::make_pair(item, std::make_pair(prio, --(queues[prio]).end())));
    // 资源分配
    do_queues();
}
```

取消预约

```
void cancel_reservation(
    T item ///< [in] key for reservation to cancel
) {
    std::lock_guard l(lock);
    auto i = queue_pointers.find(item);
    if (i != queue_pointers.end()) {
        unsigned prio = i->second.first;
        const Reservation &r = *i->second.second;
        delete r.grant;
        delete r.preempt;
        queues[prio].erase(i->second.second);
        if (queues[prio].empty()) {
            queues.erase(prio);
        }
        queue_pointers.erase(i);
    } else {
        auto p = in_progress.find(item);
        if (p != in_progress.end()) {
            if (p->second.preempt) {
                preempt_by_prio.erase(std::make_pair(p->second.prio, p->second.item));
                delete p->second.preempt;
            }
            in_progress.erase(p);
        }
    }
    // 继续给队列中其他的请求分配
    do_queues();
}
```

do_queues

做资源分配的函数

```
void do_queues() {
    // 抢占或回收资源
    while (!preempt_by_prio.empty() &&
           (in_progress.size() > max_allowed ||
            preempt_by_prio.begin()->first < min_priority)) {
        preempt_one();
    }

    while (!queues.empty()) {
        // choose highest priority queue
        auto it = queues.end();
        --it;
        ceph_assert(!it->second.empty());
        // 如果优先级过低
        if (it->first < min_priority) {
            break;
        }

        // 1. 正在进行的资源分配数量大于阈值
        // 2. 存在可以抢占的资源
        // 3. 优先级高于可抢占资源
        if (in_progress.size() >= max_allowed &&
            !preempt_by_prio.empty() &&
            it->first > preempt_by_prio.begin()->first) {
            preempt_one();
        }
        // 正在进行的资源分配数量大于阈值
        if (in_progress.size() >= max_allowed) {
            break; // no room
        }
        // 给资源
        Reservation p = it->second.front();
        // 从分配队列中取出待分配对象
        queue_pointers.erase(p.item);
        it->second.pop_front();
        if (it->second.empty()) {
            queues.erase(it);
        }
        // 分配资源
        f->queue(p.grant);
        p.grant = nullptr;
        in_progress[p.item] = p;
        if (p.preempt) {
            s.insert(std::make_pair(p.prio, p.item));
        }
    }
}
```

数据修复

数据修复有两个过程，一个是Recovery（修复）一个是Backfill（回填），当数据无法通过日志修复时（Recovery），就通过向其它OSD寻找完整的数据，拷贝到自己这（Backfill），一般前者作为临时故障的修复，后者作为长时间故障的修复或用于集群变更时PG的迁移。

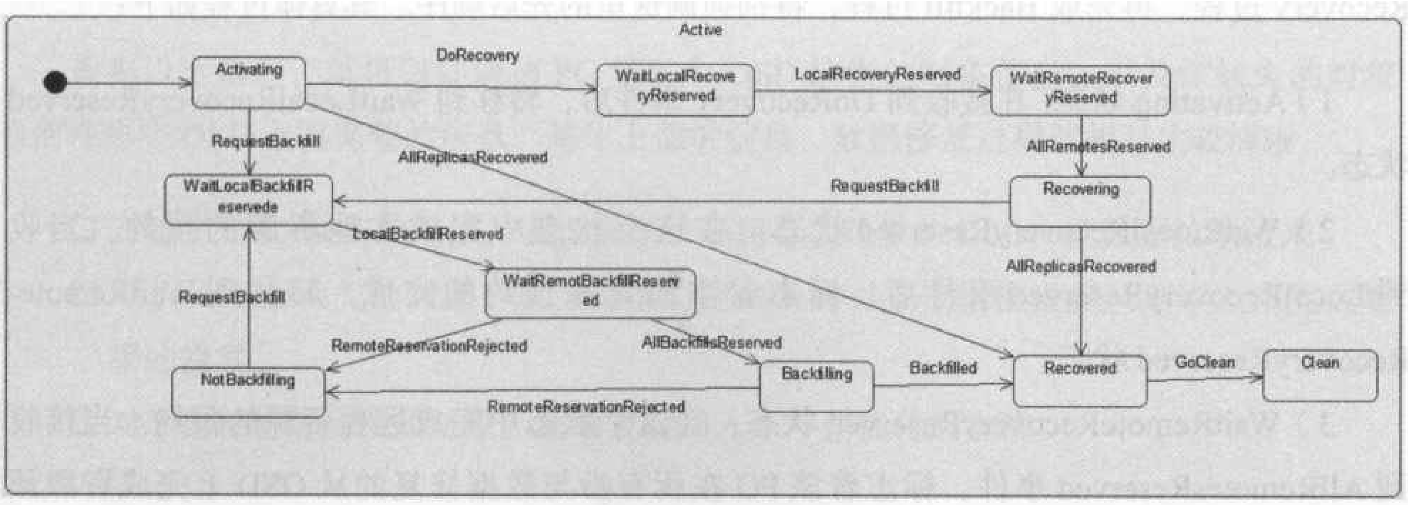


图 11-1　修复过程状态转换图

情况1：Activating状态，如果副本完整不需要修复，直接转换到Recovered状态，再到Clean状态

情况2：Activating状态，不需要Recovery时，就进行Backfill过程：

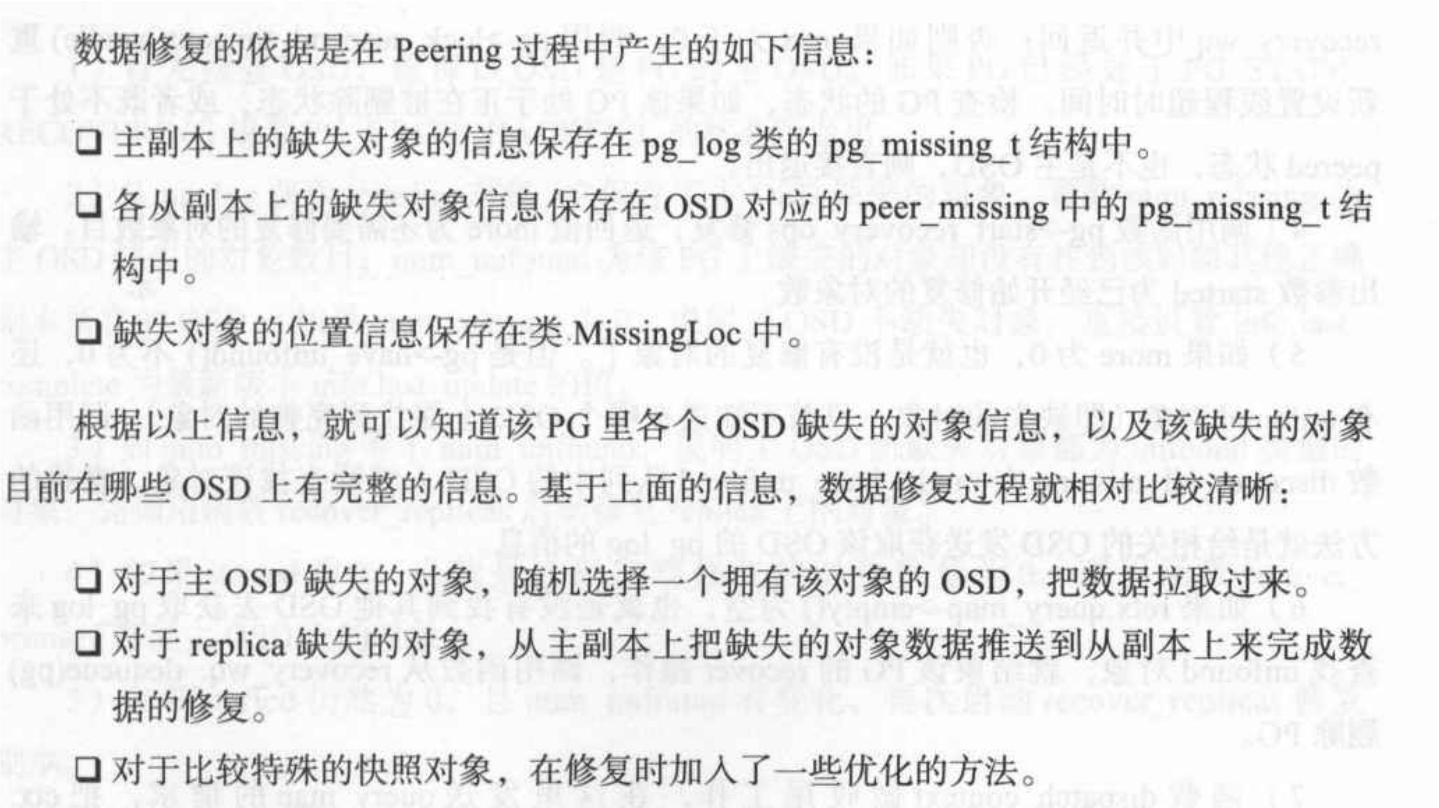
WaitLocalBackfillReserved状态申请资源，成功后，进入WaitRemoteBackfillReserved状态，**所有**副本资源预约成功后，主OSD进入Backfilling状态，完成修复进入Recovered状态。

异常：资源预约失败后进入NotBackfill状态，等待事件重新发起Backfill过程

情况3：Activating状态，进行Backfill过程：

WaitLocalRecoveryReserved状态申请资源，成功后转到WaitRemoteRecoveryReserved状态，当**所有**参与数据修复的OSD都预约到资源后，转到Recovering状态，该状态完成之后，会根据是否需要Backfill决定转入哪一个状态（见上图）

Recovery过程



数据修复的依据是在 Peering 过程中产生的如下信息：

- ❑ 主副本上的缺失对象的信息保存在 pg_log 类的 pg_missing_t 结构中。
- ❑ 各从副本上的缺失对象信息保存在 OSD 对应的 peer_missing 中的 pg_missing_t 结构中。
- ❑ 缺失对象的位置信息保存在类 MissingLoc 中。

根据以上信息，就可以知道该 PG 里各个 OSD 缺失的对象信息，以及该缺失的对象目前在哪些 OSD 上有完整的信息。基于上面的信息，数据修复过程就相对比较清晰：

- ❑ 对于主 OSD 缺失的对象，随机选择一个拥有该对象的 OSD，把数据拉取过来。
- ❑ 对于 replica 缺失的对象，从主副本上把缺失的对象数据推送到从副本上来完成数据的修复。
- ❑ 对于比较特殊的快照对象，在修复时加入了一些优化的方法。

先修复主OSD上缺失或不一致的对象，然后修复从OSD上的对象。

当前Recovery一共有两种方式

1. **Pull** 主OSD自身存在待修复对象，由主OSD按照missing_loc选择合适的副本去拉取待修复的对象目标版本至本地，完成修复方式。
2. **Push** 主OSD知道从OSD存在待修复对象，主动推送每个待修复对象目标版本到相应从OSD，然后由其本地完成修复的方式。

主OSD自我修复过程中，可能有多个从OSD拥有待修复对象目标版本，处于负载均衡的目的可以随机选择副本，完成修复之后，开始修复各个从OSD损坏的对象，依靠peering阶段生成的missing列表，通过push的方式逐个完成从OSD的修复。

流程总览

```
主OSD:
OSD::do_recovery
>> pg->start_recovery_ops

PrimaryLogPG::start_recovery_ops
>> 1 recover_primary
>> 2 recover_replicas
>> 状态转变，见状态图，完成修复或进入backfill流程

1
PrimaryLogPG::recover_primary
>> 获取缺失对象的最后一条日志
>> 1.1 处理”版本回退“操作，执行修复 recover_missing
>> 1.2 run_recovery_op 修复其它的对象

1.1
PrimaryLogPG::recover_missing
>> 修复已经删除的对象
>> 修复snap对象前，需要先递归修复head
>> 修复其它类型对象 pgbackend->recover_object

ReplicatedBackend::recover_object
>> prepare_pull 准备PullOp

1.2
ReplicatedBackend::run_recovery_op
>> send_pulls

2
PrimaryLogPG::recover_replicas
>> 遍历所有需要修复的OSD，准备PushOp
>> run_recovery_op

ReplicatedBackend::run_recovery_op
>> send_pushes

从OSD:
当主OSD send_pulls，从OSD收到
ReplicatedBackend::handle_pull
>> 封装对象数据，PushOp推送

当主OSD send_pushes，从OSD收到
ReplicatedBackend::handle_push
>> submit_push_data
>>>> submit_push_complete
>>>>>> ObjectStore::Transaction->clone_range // 将要修改的操作提交给事务，具体运行机制未分析
```

启动

```
void PGRecovery::run(
  OSD *osd,
  OSDShard *sdata,
  PGRef& pg,
  ThreadPool::TPHandle &handle)
{
  osd->logger->tinc(
    l_osd_recovery_queue_lat,
    time_queued - ceph_clock_now());
  osd->do_recovery(pg.get(), epoch_queued, reserved_pushes, priority, handle);
  pg->unlock();
}
```

do_recovery

```
void OSD::do_recovery(
    PG *pg, epoch_t queued, uint64_t reserved_pushes, int priority,
    ThreadPool::TPHandle &handle) {
    uint64_t started = 0;

    float recovery_sleep = get_osd_recovery_sleep();
    {
        // 休眠一定的时间再开始修复
        return;
    }
    // 开始修复
    bool do_unfound = pg->start_recovery_ops(reserved_pushes, handle, &started);

    if (do_unfound) { // 有些对象不知道能在哪个OSD上找到
        PeeringCtx rctx;
        rctx.handle = &handle;
        pg->find_unfound(queued, rctx); // 去找那部分对象
        dispatch_context(rctx, pg, pg->get_osdmap()); // 给其它OSD发送信息
    }
}
```

start_recovery_ops

```
bool PrimaryLogPG::start_recovery_ops(
    uint64_t max,
    ThreadPool::TPHandle &handle,
    uint64_t *ops_started) {
    uint64_t &started = *ops_started;
    started = 0;
    bool work_in_progress = false;
    bool recovery_started = false;
// 1 状态检查
    ceph_assert(is_primary());
    ceph_assert(is_peered());
    ceph_assert(!recovery_state.is_deleting());

    ceph_assert(recovery_queued);
    recovery_queued = false;

    if (!state_test(PG_STATE_RECOVERING) &&
        !state_test(PG_STATE_BACKFILLING)) {
        return have_unfound();
    }

// 2 获取missing , missing是缺失的对象
    const auto &missing = recovery_state.get_pg_log().get_missing();

// 2.1 unfound 是缺失但没有找到正确副本所在位置的对象
    uint64_t num_unfound = get_num_unfound();

// 2.2 如果不缺失对象，设置 info.last_complete = info.last_update;
    if (!recovery_state.have_missing()) {
        recovery_state.local_recovery_complete();
    }

// 3 主OSD没有missing，或者所有的missing都是unfound对象，就先修复副本
    if (!missing.have_missing() ||
        recovery_state.all_missing_unfound()) {
        // 修复副本
        started = recover_replicas(max, handle, &recovery_started);
    }

// 3.1 表示启动修复的对象数量为0，修复主OSD上的对象
    if (!started) {
        started += recover_primary(max, handle);
    }

// 3.2 如果仍然为0，且num_unfound发生变化，那么再次启动修复
    if (!started && num_unfound != get_num_unfound()) {
        // 修复副本
        started = recover_replicas(max, handle, &recovery_started);
    }

    if (started || recovery_started)
        work_in_progress = true;

    bool deferred_backfill = false; // 推迟backfill

// 4 决定推迟backfill还是立即开始
    if (recovering.empty() &&
        state_test(PG_STATE_BACKFILLING) &&
        !get_backfill_targets().empty() && started < max &&
        missing.num_missing() == 0 &&
        waiting_on_backfill.empty()) {
        if (get_osdmap()->test_flag(CEPH_OSDMAP_NOBACKFILL)) {
            deferred_backfill = true;
        } else if (get_osdmap()->test_flag(CEPH_OSDMAP_NOREBALANCE) && !is_degraded()) {
            deferred_backfill = true;
        } else if (!recovery_state.is_backfill_reserved()) {
            // 如果backfill_reserved没有设置
            /* DNMNOTE I think this branch is dead */
            if (!backfill_reserving) {
                backfill_reserving = true;
                queue_peering_event(
                    PGPeeringEventRef(
                        std::make_shared<PGPeeringEvent>(
                            get_osdmap_epoch(),
                            get_osdmap_epoch(),
                            PeeringState::RequestBackfill())));
            }
            deferred_backfill = true;
        } else {
            // 开始backfill过程
            started += recover_backfill(max - started, handle, &work_in_progress);
        }
    }

// 5 if: 是否有正在进行的数据恢复操作，是否有正在进行的工作，是否有活跃的恢复操作，
// 是否有延迟的数据回填操作
// 返回值为true: 表示需要继续进行数据修复，此时没有正在运行的恢复工作，且有未找到的对象
    if (!recovering.empty() ||
        work_in_progress || recovery_ops_active > 0 || deferred_backfill)
        return !work_in_progress && have_unfound();

    ceph_assert(recovering.empty());
    ceph_assert(recovery_ops_active == 0);

    int unfound = get_num_unfound();
    if (unfound) {
        return true;
    }

    if (missing.num_missing() > 0) {
    // 这不应该发生
```

```
        osd->clog->error() << info.pgid << " Unexpected Error: recovery ending with "
                       << missing.num_missing() << ": " << missing.get_items();
        return false;
    }

    if (needs_recovery()) {
    // 这不应该发生
        osd->clog->error() << info.pgid
                       << " Unexpected Error: recovery ending with missing replicas";
        return false;
    }

// 6 状态转换
    if (state_test(PG_STATE_RECOVERING)) {
        state_clear(PG_STATE_RECOVERING);
        state_clear(PG_STATE_FORCED_RECOVERY);
        if (needs_backfill()) {
            queue_peering_event(
                PGPeeringEventRef(
                    std::make_shared<PGPeeringEvent>(
                        get_osdmap_epoch(),
                        get_osdmap_epoch(),
                        PeeringState::RequestBackfill())));
        } else {
            state_clear(PG_STATE_FORCED_BACKFILL);
            queue_peering_event(
                PGPeeringEventRef(
                    std::make_shared<PGPeeringEvent>(
                        get_osdmap_epoch(),
                        get_osdmap_epoch(),
                        PeeringState::AllReplicasRecovered())));
        }
    } else { // backfilling
        state_clear(PG_STATE_BACKFILLING);
        state_clear(PG_STATE_FORCED_BACKFILL);
        state_clear(PG_STATE_FORCED_RECOVERY);
        queue_peering_event(
            PGPeeringEventRef(
                std::make_shared<PGPeeringEvent>(
                    get_osdmap_epoch(),
                    get_osdmap_epoch(),
                    PeeringState::Backfilled())));
    }

    return false;
```

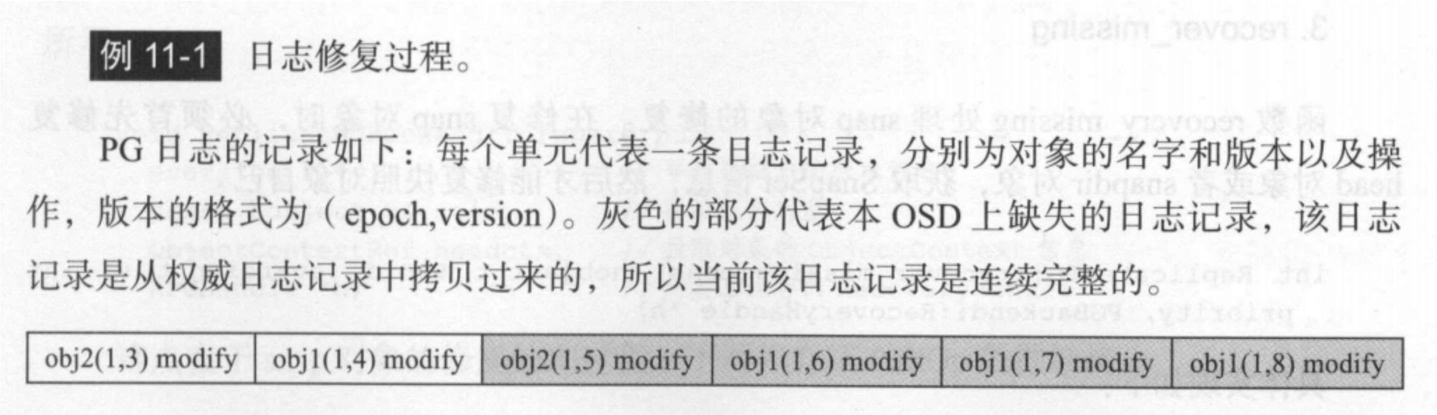
```
}

```

recover_primary

```
uint64_t PrimaryLogPG::recover_primary(uint64_t max, ThreadPool::TPHandle &handle) {
    ceph_assert(is_primary());

    const auto &missing = recovery_state.get_pg_log().get_missing();

    // look at log!
    pg_log_entry_t *latest = 0;
    unsigned started = 0;
    int skipped = 0;

// 1 获取一个Handle，用于Push和Pull
    PGBackend::RecoveryHandle *h = pgbackend->open_recovery_op();
    map<version_t, hobject_t>::const_iterator p = missing.get_rmissing().lower_bound(recovery_state.get_pg_log().get_log().last_requested);
    while (p != missing.get_rmissing().end()) { // 遍历未被修复的对象
        handle.reset_tp_timeout();
        hobject_t soid;
        version_t v = p->first;

// 2 lastest 是日志记录中保存的该缺失对象的最后的一条日志，soid为缺失的对象
        auto it_objects = recovery_state.get_pg_log().get_log().objects.find(p->second);
        if (it_objects != recovery_state.get_pg_log().get_log().objects.end()) {
            latest = it_objects->second;
            ceph_assert(latest->is_update() || latest->is_delete());
            soid = latest->soid;
        } else {
            latest = 0;
            soid = p->second;
        }
        const pg_missing_item &item = missing.get_items().find(p->second)->second;
        ++p;

        hobject_t head = soid.get_head();
        eversion_t need = item.need;

```

```
// 3 开始修复
        if (latest) {
            switch (latest->op) {
            case pg_log_entry_t::CLONE:
            /*
            暂时取消了对这种特殊情况的处理，直到我们能够从旧的SnapSet中正确地构建一个准确的SnapSet。
            */
                break;
// 4 该记录类型为LOST_REVERT：该revert操作为数据不一致时，管理员通过命令强行回退到指定版本
// reverting记录了回退的版本号
            case pg_log_entry_t::LOST_REVERT: {

// 4.1 此条件表示：日志记录显示当前已经拥有回退的版本，
                if (item.have == latest->reverting_to) {
                // 获取该对象的ObjectContext
                    ObjectContextRef obc = get_object_context(soid, true);

// 4.1.1 如果检查对象当前的版本 obc->obs.oi.version 等于 latest->version，说明回退操作已完成
                    if (obc->obs.oi.version == latest->version) {
                        // I''m already reverting
                    } else {
// 4.1.2 说明没有执行回退操作，直接修改对象的版本号为latest->version，关于为什么如此，见下文
                        obc->obs.oi.version = latest->version;

                        ObjectStore::Transaction t;
                        bufferlist b2;
                        obc->obs.oi.encode(
                            b2,
                            get_osdmap()->get_features(CEPH_ENTITY_TYPE_OSD, nullptr));
                        ceph_assert(!pool.info.require_rollback());
                        t.setattr(coll, ghobject_t(soid), OI_ATTR, b2);
                        // 更新恢复状态
                        recovery_state.recover_got(
                            soid,
                            latest->version,
                            false,
                            t);

                        ++active_pushes;

                        t.register_on_applied(new C_OSD_AppliedRecoveredObject(this, obc));
                        t.register_on_commit(new C_OSD_CommittedPushedObject(
                            this,
                            get_osdmap_epoch(),
                            info.last_complete));
                        osd->store->queue_transaction(ch, std::move(t));
                        continue;
                    }
                } else {
// 4.2 需要拉取该reverting_to版本的对象，这里不做特殊处理，只是检查所有OSD是否拥有该版本的对象，如果有就加入到missing_loc记录该版本的位置信息，由后续修复继续来完成
                    eversion_t alternate_need = latest->reverting_to;

                    set<pg_shard_t> good_peers;
                    for (auto p = recovery_state.get_peer_missing().begin();
                         p != recovery_state.get_peer_missing().end();
                         ++p) {
                        if (p->second.is_missing(soid, need) &&
                            p->second.get_items().at(soid).have == alternate_need) {
                            good_peers.insert(p->first);
                        }
                    }
                    recovery_state.set_revert_with_targets(soid, good_peers);
                    /*
                    void PeeringState::set_revert_with_targets(
                          const hobject_t &soid,
                          const set<pg_shard_t> &good_peers) {
                        for (auto &&peer: good_peers) {
                            missing_loc.add_location(soid, peer);
                        }
                    }
                    */
                }
            } break;
            }
        }

// 5 如果当前soid没有在修复
        if (!recovering.count(soid)) {
            if (recovering.count(head)) { // 或者head在修复
                ++skipped;
            } else {
            // 修复
                int r = recover_missing(soid, need, recovery_state.get_recovery_op_priority(), h);
                switch (r) {
                case PULL_YES:
                    ++started;
                    break;
                case PULL_HEAD:
                    ++started;
                case PULL_NONE:
                    ++skipped;
                    break;
                default:
                    ceph_abort();
                }
                if (started >= max)
                    break;
            }
        }

        if (!skipped)
            recovery_state.set_last_requested(v);
    }

// 6 把PullOp或PushOp封装的消息发送出去
    pgbackend->run_recovery_op(h, recovery_state.get_recovery_op_priority());
    return started;
}

```

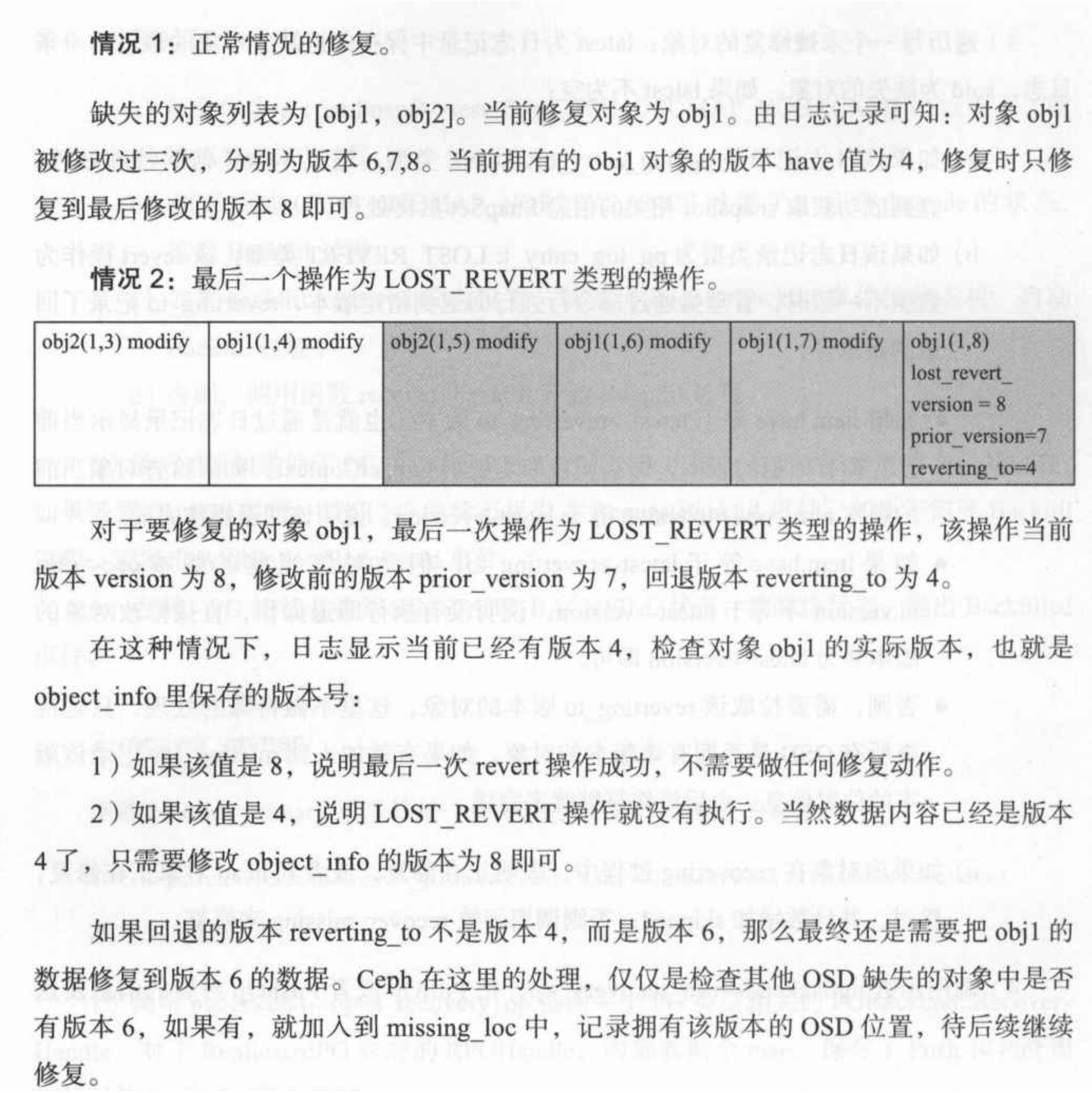
书上的例子：



例 11-1 日志修复过程。

PG 日志的记录如下：每个单元代表一条日志记录，分别为对象的名字和版本以及操作，版本的格式为（epoch,version）。灰色的部分代表本 OSD 上缺失的日志记录，该日志记录是从权威日志记录中拷贝过来的，所以当前该日志记录是连续完整的。

obj2(1,3) modify	obj1(1,4) modify	obj2(1,5) modify	obj1(1,6) modify	obj1(1,7) modify	obj1(1,8) modify



情况 1：正常情况的修复。

缺失的对象列表为 [obj1，obj2]。当前修复对象为 obj1。由日志记录可知：对象 obj1 被修改过三次，分别为版本 6,7,8。当前拥有的 obj1 对象的版本 have 值为 4，修复时只修复到最后修改的版本 8 即可。

情况 2：最后一个操作为 LOST_REVERT 类型的操作。

obj2(1,3) modify	obj1(1,4) modify	obj2(1,5) modify	obj1(1,6) modify	obj1(1,7) modify	obj1(1,8) lost_revert_ version = 8 prior_version=7 reverting_to=4

对于要修复的对象 obj1，最后一次操作为 LOST_REVERT 类型的操作，该操作当前版本 version 为 8，修改前的版本 prior_version 为 7，回退版本 reverting_to 为 4。

在这种情况下，日志显示当前已经有版本 4，检查对象 obj1 的实际版本，也就是 object_info 里保存的版本号：

1）如果该值是 8，说明最后一次 revert 操作成功，不需要做任何修复动作。

2）如果该值是 4，说明 LOST_REVERT 操作就没有执行。当然数据内容已经是版本 4 了，只需要修改 object_ info 的版本为 8 即可。

如果回退的版本 reverting_to 不是版本 4，而是版本 6，那么最终还是需要把 obj1 的数据修复到版本 6 的数据。Ceph 在这里的处理，仅仅是检查其他 OSD 缺失的对象中是否有版本 6，如果有，就加入到 missing_loc 中，记录拥有该版本的 OSD 位置，待后续继续修复。

最后一种情况，如果是版本6，对象中并没有版本6的数据，将数据修复为版本6，当前版本设置为8。

recover_missing

```
int PrimaryLogPG::recover_missing(
    const hobject_t &soid, eversion_t v,
    int priority,
    PGBackend::RecoveryHandle *h) {

// 1 如果是unfound对象，无法修复
    if (recovery_state.get_missing_loc().is_unfound(soid)) {
        return PULL_NONE;
    }

// 2 已删除的对象
    if (recovery_state.get_missing_loc().is_deleted(soid)) {
// 2.1 开始修复操作
        start_recovery_op(soid);
        // 确保不在修复中
        ceph_assert(!recovering.count(soid));
        // 加入修复队列
        recovering.insert(make_pair(soid, ObjectContextRef()));
        epoch_t cur_epoch = get_osdmap_epoch();
        remove_missing_object(soid, v, new LambdaContext([=, this](int) {
            std::scoped_lock locker{*this};
            if (!pg_has_reset_since(cur_epoch)) {
              bool object_missing = false;
              for (const auto &shard : get_acting_recovery_backfill()) {
                  if (shard == pg_whoami)
                      continue;
                  if (recovery_state.get_peer_missing(shard).is_missing(soid)) {
                      object_missing = true;
                      break;
                  }
              }
              if (!object_missing) {
                  object_stat_sum_t stat_diff;
                  stat_diff.num_objects_recovered = 1;
                  if (scrub_after_recovery)
                      stat_diff.num_objects_repaired = 1;
                  on_global_recover(soid, stat_diff, true);
              } else {
                  auto recovery_handle = pgbackend->open_recovery_op();
                  pgbackend->recover_delete_object(soid, v, recovery_handle);
                  pgbackend->run_recovery_op(recovery_handle, priority);
              }
            }
            }));
            return PULL_YES;
    }

// is this a snapped object? if so, consult the snapset.. we may not need the entire object!
// 3 快照对象
    ObjectContextRef obc;
    ObjectContextRef head_obc;
    if (soid.snap && soid.snap < CEPH_NOSNAP) {
        // do we have the head?
        hobject_t head = soid.get_head();
        if (recovery_state.get_pg_log().get_missing().is_missing(head)) {
            if (recovering.count(head)) {
                return PULL_NONE;
            } else {
                int r = recover_missing(
                    head,
recovery_state.get_pg_log().get_missing().get_items().find(head)->second.need,
priority,
                    h);
                if (r != PULL_NONE)
                    return PULL_HEAD;
                return PULL_NONE;
            }
        }
        head_obc = get_object_context(
            head,
            false,
            0);
        ceph_assert(head_obc);
    }
    start_recovery_op(soid);
    ceph_assert(!recovering.count(soid));
    recovering.insert(make_pair(soid, obc));

// 4 修复对象
    int r = pgbackend->recover_object(soid, v, head_obc, obc, h);
    ceph_assert(r >= 0);
    return PULL_YES;
}
```

recover_object

pgbackend封装了不同类型的Pool的实现。ReplicatedBackend实现了replicate类型的PG相关的底层功能，ECbackend实现了Erasure code类型的PG相关的底层功能。

我们讨论基于副本的修复。

```
int ReplicatedBackend::recover_object(
    const hobject_t &hoid,
    eversion_t v,
    ObjectContextRef head,
    ObjectContextRef obc,
    RecoveryHandle *_h) {
```

```
    RPGHandle *h = static_cast<RPGHandle *>(_h);
    if (get_parent()->get_local_missing().is_missing(hoid)) {
        ceph_assert(!obc);
        // 把请求封装为PullOp
        prepare_pull(v, hoid, head, h);
    } else {
        ceph_assert(obc);
        // 把请求封装为PushOp
        int started = start_pushes(hoid, obc, h);
        if (started < 0) {
            pushing[hoid].clear();
            return started;
        }
    }
    return 0;
}
```

1 pull

```
struct PullOp {
    hobject_t soid;                                   // 需要拉取的对象
    ObjectRecoveryInfo recovery_info;                 // 对象修复的信息
    ObjectRecoveryProgress recovery_progress;         // 对象修复进度信息
};

struct ObjectRecoveryInfo {
    hobject_t soid;
    eversion_t version;
    uint64_t size;
    object_info_t oi;
    SnapSet ss;                              // 修复对象的快照信息

    interval_set<uint64_t> copy_subset;    // 修复快照时，需要从其它OSD拷贝到本地的对象的区段集合
    std::map<hobject_t, interval_set<uint64_t>> clone_subset;
    // clone对象修复时，需要从本地拷贝来修复的区间
    bool object_exist;
}

struct ObjectRecoveryProgress {
    uint64_t data_recovered_to;       // data已修复的位置指针
    std::string omap_recovered_to;    // omap已修复的位置指针
    bool first;               // 是否是首次修复
    bool data_complete;       // data是否修复完成
    bool omap_complete;       // omap是否修复完成
    bool error = false;
}
```

```
void ReplicatedBackend::prepare_pull(
    eversion_t v,              // 要拉取对象的版本信息
    const hobject_t& soid,     // 要拉取的对象
    ObjectContextRef headctx,  // 拉取对象的ObjectContext信息
    RPGHandle *h){             // 封装后保存的RecveryHandle
// 1 获取PG对象
    const auto missing_iter = get_parent()->get_local_missing().get_items().find(soid);
    ceph_assert(missing_iter != get_parent()->get_local_missing().get_items().end());
    eversion_t _v = missing_iter->second.need;
    ceph_assert(_v == v);

// 2 missing_loc 包含缺失对象的位置，peering_missing包含其它节点缺失对象的信息
    const map<hobject_t, set<pg_shard_t>> &missing_loc(
        get_parent()->get_missing_loc_shards());
    const map<pg_shard_t, pg_missing_t> &peer_missing(
        get_parent()->get_shard_missing());

// 3 查找soid所在的OSD集合
    map<hobject_t, set<pg_shard_t>>::const_iterator q = missing_loc.find(soid);
    ceph_assert(q != missing_loc.end());
    ceph_assert(!q->second.empty());

// 4 选择一个特定的分片（OSD）作为拉取操作的目标
    auto p = q->second.end();
    if (cct->_conf->osd_debug_feed_pullee >= 0) {
        for (auto it = q->second.begin(); it != q->second.end(); it++) {
            if (it->osd == cct->_conf->osd_debug_feed_pullee) {
                p = it;
                break;
            }
        }
    }
// 4.1 如果没有找到特定的pullee，可能是用户输入了错误的信息，随机选择一个目标
    if (p == q->second.end()) {
        // probably because user feed a wrong pullee
        p = q->second.begin();
        std::advance(p,ceph::util::generate_random_number<int>(0,
               q->second.size() - 1));
    }
    ceph_assert(get_osdmap()->is_up(p->osd));
    pg_shard_t fromshard = *p;

// 5 确保选中的OSD上确实需要的对象
    ceph_assert(peer_missing.count(fromshard));
    const pg_missing_t &pmissing = peer_missing.find(fromshard)->second;
    if (pmissing.is_missing(soid, v)) {
        ceph_assert(pmissing.get_items().find(soid)->second.have != v);
        v = pmissing.get_items().find(soid)->second.have;
        ceph_assert(get_parent()->get_log().get_log().objects.count(soid) &&
                    (get_parent()->get_log().get_log().objects.find(soid)->second->op ==
                     pg_log_entry_t::LOST_REVERT) &&
                    (get_parent()->get_log().get_log().objects.find(soid)
                         ->second->reverting_to ==v));
    }

    ObjectRecoveryInfo recovery_info;
    ObcLockManager lock_manager;
// 6 如果是快照
    if (soid.is_snap()) {
        ceph_assert(!get_parent()->get_local_missing().is_missing(soid.get_head()));
        ceph_assert(headctx);
        // check snapset
        SnapSetContext *ssc = headctx->ssc;
        ceph_assert(ssc);

        recovery_info.ss = ssc->snapset;
        calc_clone_subsets(
            ssc->snapset, soid, get_parent()->get_local_missing(),
            get_info().last_backfill,
            recovery_info.copy_subset,
            recovery_info.clone_subset,
            lock_manager);
        // FIXME: this may overestimate if we are pulling multiple clones in parallel...

        ceph_assert(ssc->snapset.clone_size.count(soid.snap));
        recovery_info.size = ssc->snapset.clone_size[soid.snap];
        recovery_info.object_exist = missing_iter->second.clean_regions.object_is_exist();
    } else {
// 6.1 如果是head对象，拉取全部
        // pulling head or unversioned object.
        // always pull the whole thing.
        recovery_info.copy_subset.insert(0, (uint64_t)-1);
        assert(HAVE_FEATURE(parent->min_peer_features(), SERVER_OCTOPUS));
        recovery_info.copy_subset.intersection_of(missing_iter->second.clean_regions.get_dirty_regions());
        recovery_info.size = ((uint64_t)-1);
        recovery_info.object_exist = missing_iter->second.clean_regions.object_is_exist();
    }

// 7 创建PullOp对象，设置拉取操作的相关信息
    h->pulls[fromshard].push_back(PullOp());
    PullOp &op = h->pulls[fromshard].back();
    op.soid = soid;

    op.recovery_info = recovery_info;
    op.recovery_info.soid = soid;
    op.recovery_info.version = v;
    op.recovery_progress.data_complete = false;
    op.recovery_progress.omap_complete = !missing_iter->second.clean_regions.omap_is_dirty();
    op.recovery_progress.data_recovered_to = 0;
    op.recovery_progress.first = true;

    ceph_assert(!pulling.count(soid));
    pull_from_peer[fromshard].insert(soid);
    PullInfo &pi = pulling[soid];
    pi.from = fromshard;
    pi.soid = soid;
    pi.head_ctx = headctx;
    pi.recovery_info = op.recovery_info;
    pi.recovery_progress = op.recovery_progress;
    pi.cache_dont_need = h->cache_dont_need;
    pi.lock_manager = std::move(lock_manager);
}

```

1.1 calc_clone_subsetes

用于修复快照对象，在此之前介绍两个概念

1.在SnapSet结构中，字段clone_overlap保存了clone对象和上一次clone对象的重叠部分（没有冲突部分）

```
struct SnapSet {
    snapid_t seq;
    std::vector<snapid_t> snaps;                              // descending
    std::vector<snapid_t> clones;                             // ascending
    std::map<snapid_t, interval_set<uint64_t>> clone_overlap; // overlap w/
next newest
    std::map<snapid_t, uint64_t> clone_size;
    std::map<snapid_t, std::vector<snapid_t>> clone_snaps; // descending
}
```

2.clone_overlap

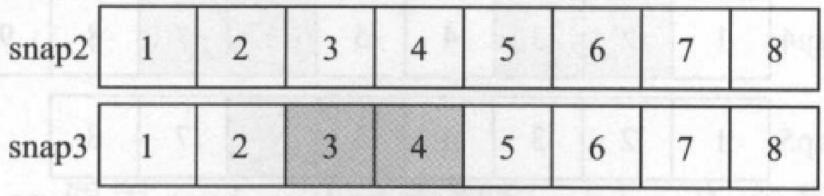


图 11-2 clone_overlap 示意图

snap3从snap2克隆过来，然后修改了区间3，4，其在对象中范围的offset和length为（4，8），（8，12）那么记录就为：

clone_overlap[3] = {(0, 4), (12, len(区间8-区间4))}

然后是calc_clone_subsets函数

```
void ReplicatedBackend::calc_clone_subsets(
    SnapSet &snapset, const hobject_t &soid,
    const pg_missing_t &missing,
    const hobject_t &last_backfill,
    interval_set<uint64_t> &data_subset,
    map<hobject_t, interval_set<uint64_t>> &clone_subsets,
    ObcLockManager &manager) {

// 1 获取快照大小，加入到data_subset中（虽然不知道这是在干嘛）
    uint64_t size = snapset.clone_size[soid.snap];
    if (size) data_subset.insert(0, size);

    // any overlap with next older clone?
    interval_set<uint64_t> cloning;
    interval_set<uint64_t> prev;

// 2 往前查找完整的快照对象区间，添加到clone_subsets和cloning
    if (size)
        prev.insert(0, size);
    for (int j = i - 1; j >= 0; j--) {
        hobject_t c = soid;
        c.snap = snapset.clones[j];
        // 计算重叠区间
        prev.intersection_of(snapset.clone_overlap[snapset.clones[j]]);
        if (!missing.is_missing(c) &&
            c < last_backfill &&
            get_parent()->try_lock_for_read(c, manager)) {
            clone_subsets[c] = prev;
            cloning.union_of(prev);
            break;
        }
    }

// 同上，往后查找
    // overlap with next newest?
    interval_set<uint64_t> next;
    if (size)
        next.insert(0, size);
    for (unsigned j = i + 1; j < snapset.clones.size(); j++) {
        hobject_t c = soid;
        c.snap = snapset.clones[j];
        next.intersection_of(snapset.clone_overlap[snapset.clones[j - 1]]);
        if (!missing.is_missing(c) &&
            c < last_backfill &&
            get_parent()->try_lock_for_read(c, manager)) {
            clone_subsets[c] = next;
            cloning.union_of(next);
            break;
        }
    }
    // 去重
    data_subset.subtract(cloning);
}
```

光看代码看不出是在做什么，书上给了一个例子：

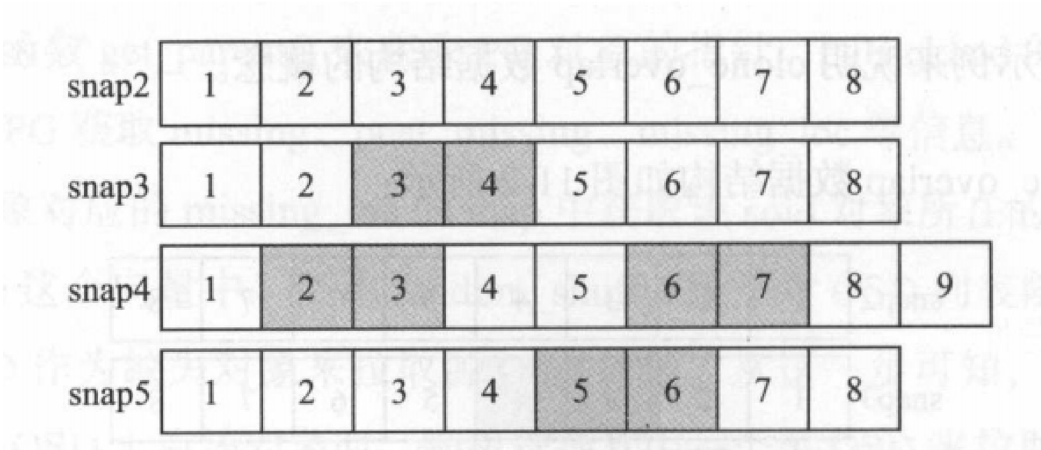


图 11-3　快照对象修复计算示例图

snap3是克隆的snap2, snap4 克隆的snap3，snap5克隆的snap4，灰色区间表示clone后修改的区间，snap2、3、5都是完整的对象，要修复的对象是snap4，不同长度代表各个clone对象的size是不同的。

1.向前查找和snap4相同的区间，可以看到区间1，5，8和snap4相同，那么snap4就直接从本地拿到这三个区间。

2.向后查找，可以看到1，2，3，4，7，8未被修改，那么直接拿过来就行

3.去重，最后只有区间6需要从其他OSD上拷贝数据来修复。

2 push

获取actingbackfill的OSD列表，通过peering_missing查找缺失该对象的OSD，并发送信息给对方

```
int ReplicatedBackend::start_pushes(
    const hobject_t &soid,
    ObjectContextRef obc,
    RPGHandle *h) {

// 1 用于存储需要数据的OSD
    list<map<pg_shard_t, pg_missing_t>::const_iterator> shards;

// 遍历actingbackfill列表，统计需要信息的OSD
    ceph_assert(get_parent()->get_acting_recovery_backfill_shards().size() > 0);
    for (set<pg_shard_t>::iterator i =
             get_parent()->get_acting_recovery_backfill_shards().begin();
         i != get_parent()->get_acting_recovery_backfill_shards().end();
         ++i) {
        if (*i == get_parent()->whoami_shard())
            continue;
        pg_shard_t peer = *i;
        map<pg_shard_t, pg_missing_t>::const_iterator j =
            get_parent()->get_shard_missing().find(peer);
        ceph_assert(j != get_parent()->get_shard_missing().end());
        if (j->second.is_missing(soid)) {
            shards.push_back(j);
        }
    }

    // If more than 1 read will occur ignore possible request to not cache
    bool cache = shards.size() == 1 ? h->cache_dont_need : false;

// 发送信息
    for (auto j : shards) {
        pg_shard_t peer = j->first;
        h->pushes[peer].push_back(PushOp());
        int r = prep_push_to_replica(obc, soid, peer,
                                     &(h->pushes[peer].back()), cache);
        if (r < 0) {
            // Back out all failed reads
            for (auto k : shards) {
                pg_shard_t p = k->first;
                h->pushes[p].pop_back();
                if (p == peer)
                    break;
            }
            return r;
        }
    }
    return shards.size();
}
```

```
int ReplicatedBackend::prep_push_to_replica(
    ObjectContextRef obc, const hobject_t &soid, pg_shard_t peer,
    PushOp *pop, bool cache_dont_need) {
    const object_info_t &oi = obc->obs.oi;
    uint64_t size = obc->obs.oi.size;

    map<hobject_t, interval_set<uint64_t>> clone_subsets;
    interval_set<uint64_t> data_subset;

    ObcLockManager lock_manager;
// 1 如果对象是一个快照
    if (soid.snap && soid.snap < CEPH_NOSNAP) {
        hobject_t head = soid;
        head.snap = CEPH_NOSNAP;

// 1.1 尝试基于成功或先于当前快照的克隆来推送数据。这需要在本地获取对象的头部（head）和当前的快照集（SnapSet）
        if (get_parent()->get_local_missing().is_missing(head)) {
            return prep_push(obc, soid, peer, pop, cache_dont_need);
        }
// 1.2 如果head存在，获取克隆数据子集和数据子集，使用这些信息来准备推送操作
        SnapSetContext *ssc = obc->ssc;
        ceph_assert(ssc);

        pop->recovery_info.ss = ssc->snapset;
        map<pg_shard_t, pg_missing_t>::const_iterator pm =
            get_parent()->get_shard_missing().find(peer);
        ceph_assert(pm != get_parent()->get_shard_missing().end());
        map<pg_shard_t, pg_info_t>::const_iterator pi =
            get_parent()->get_shard_info().find(peer);
        ceph_assert(pi != get_parent()->get_shard_info().end());
        calc_clone_subsets(
            ssc->snapset, soid,
            pm->second,
            pi->second.last_backfill,
            data_subset, clone_subsets,
            lock_manager);
    } else if (soid.snap == CEPH_NOSNAP) {
// 2 如果不是快照，同样计算数据集，并使用这些信息来准备推送操作
        SnapSetContext *ssc = obc->ssc;
        ceph_assert(ssc);

        calc_head_subsets(
            obc,
            ssc->snapset, soid, get_parent()->get_shard_missing().find(peer)->second,
            get_parent()->get_shard_info().find(peer)->second.last_backfill,
            data_subset, clone_subsets,
            lock_manager);
    }
// 3 构造PushOp
    return prep_push(
        obc,            // 对象上下文
        soid,           // 对象ID
        peer,           // 目标
        oi.version,     // 对象版本
        data_subset,    // 数据子集
        clone_subsets,  // 克隆数据子集
        pop,            // 推送操作指针
        cache_dont_need,// 是否需要缓存
        std::move(lock_manager)); // 锁管理器
}

prep_push:
>> int r = build_push_op(pi.recovery_info, pi.recovery_progress, &new_progress,
        pop, &(pi.stat), cache_dont_need);

```

```
int ReplicatedBackend::build_push_op(const ObjectRecoveryInfo &recovery_info,
                                     const ObjectRecoveryProgress &progress,
                                     ObjectRecoveryProgress *out_progress,
                                     PushOp *out_op,
                                     object_stat_sum_t *stat,
                                     bool cache_dont_need) {
    ObjectRecoveryProgress _new_progress;
    if (!out_progress)
        out_progress = &_new_progress;
    ObjectRecoveryProgress &new_progress = *out_progress;
    new_progress = progress;
    eversion_t v = recovery_info.version;
    object_info_t oi;

// 1 是第一次恢复，需要获取元数据信息
    if (progress.first) {
        int r = store->omap_get_header(ch, ghobject_t(recovery_info.soid), &out_op->omap_header);
        r = store->getattrs(ch, ghobject_t(recovery_info.soid), out_op->attrset);

// 2 解码对象信息，并检查本地版本号与请求的版本号是否一致。如果不一致，或者请求的版本号未知，则返回错误。
        // Debug
        try {
            oi.decode(out_op->attrset[OI_ATTR]);
        } catch (...) {
            return -EINVAL;
        }

        // If requestor didn''t know the version, use ours
        if (v == eversion_t()) {
            v = oi.version;
        } else if (oi.version != v) {
            return -EINVAL;
        }

// 3 标志为false，表示后续请求将包含版本信息
        new_progress.first = false;
    }
    // 一旦我们提供了版本，随后的请求就会有它，所以在这一点上它必须是已知的。
    ceph_assert(v != eversion_t());

// 4 上一步只获取了header，没有omap(key, value)信息，这一步获取
// 首先获取可用的数据块大小
    uint64_t available = cct->_conf->osd_recovery_max_chunk;
    if (!progress.omap_complete) {
        ObjectMap::ObjectMapIterator iter =
            store->get_omap_iterator(ch,
                                     ghobject_t(recovery_info.soid));
        ceph_assert(iter);

// 4.1 获取omap信息，添加到omap_entries，
// 一次获取的信息大小不能超过osd_recovery_max_chunk(available )
        for (iter->lower_bound(progress.omap_recovered_to);
             iter->valid();
             iter->next()) {
            if (!out_op->omap_entries.empty() &&
                ((cct->_conf->osd_recovery_max_omap_entries_per_chunk > 0 &&
                  out_op->omap_entries.size() >= cct->_conf->osd_recovery_max_omap_entries_per_chunk) ||
                 available <= iter->key().size() + iter->value().length()))
                break;
            out_op->omap_entries.insert(make_pair(iter->key(), iter->value()));

            if ((iter->key().size() + iter->value().length()) <= available)
                available -= (iter->key().size() + iter->value().length());
            else
                available = 0;
        }
        if (!iter->valid())
            new_progress.omap_complete = true;
        else
            new_progress.omap_recovered_to = iter->key();
    }

// 5 获取omap之后，现在要获取数据
    if (available > 0) {
        if (!recovery_info.copy_subset.empty()) {
            interval_set<uint64_t> copy_subset = recovery_info.copy_subset;
            map<uint64_t, uint64_t> m;
// 5.1 获取文件的映射信息
            int r = store->fiemap(ch, ghobject_t(recovery_info.soid), 0,
                                  copy_subset.range_end(), m);
            if (r >= 0) {
// 5.2 保存下需要推送的数据区间
                interval_set<uint64_t> fiemap_included(std::move(m));
                copy_subset.intersection_of(fiemap_included);
            } else {
            // copy_subset和empty interval_set的交集无论如何都是空的
                copy_subset.clear();
            }
// 5.3 将数据更新到data_included
            out_op->data_included.span_of(copy_subset, progress.data_recovered_to,
                                          available);

            if (out_op->data_included.empty() ||
                out_op->data_included.range_end() == copy_subset.range_end())
                new_progress.data_recovered_to = recovery_info.copy_subset.range_end();
            else
                new_progress.data_recovered_to = out_op->data_included.range_end();
        }
    } else {
        out_op->data_included.clear();
    }

    auto origin_size = out_op->data_included.size();
    bufferlist bit;
// 6 将数据读到缓冲区bit，说实话不知道为什么又要把数据转一遍
    int r = store->readv(ch, ghobject_t(recovery_info.soid),
                         out_op->data_included, bit,
                         cache_dont_need ? CEPH_OSD_OP_FLAG_FADVISE_DONTNEED : 0);

    if (cct->_conf->osd_debug_random_push_read_error &&
        (rand() % (int)(cct->_conf->osd_debug_random_push_read_error * 100.0)) == 0) {
        r = -EIO;
    }
// 7 数据块大小发生变化
    if (out_op->data_included.size() != origin_size) {
        new_progress.data_complete = true;
    }
// 8 将数据追加到out_op->data，是data_included不能直接追加吗
    out_op->data.claim_append(bit);
    // 检验数据完整性，数据校验和等
    if (progress.first && !out_op->data_included.empty() &&
        out_op->data_included.begin().get_start() == 0 &&
        out_op->data.length() == oi.size && oi.is_data_digest()) {
        uint32_t crc = out_op->data.crc32c(-1);
        if (oi.data_digest != crc) {
            return -EIO;
        }
    }

// 9 恢复进度完成
    if (new_progress.is_complete(recovery_info)) {
        new_progress.data_complete = true;
        if (stat) {
            stat->num_objects_recovered++;
            if (get_parent()->pg_is_repair())
```

```
                stat->num_objects_repaired++;
        }
    } else if (progress.first && progress.omap_complete) {
        new_progress.omap_complete = false;
    }

    if (stat) {
        stat->num_keys_recovered += out_op->omap_entries.size();
        stat->num_bytes_recovered += out_op->data.length();
        get_parent()->get_logger()->inc(l_osd_rbytes, out_op->omap_entries.size() + out_op->data.length());
    }

    get_parent()->get_logger()->inc(l_osd_push);
    get_parent()->get_logger()->inc(l_osd_push_outb, out_op->data.length());

// 10 更新out_op信息，注意out_op其实就是PushOp的引用
    // send
    out_op->version = v;
    out_op->soid = recovery_info.soid;
    out_op->recovery_info = recovery_info;
    out_op->after_progress = new_progress;
    out_op->before_progress = progress;
    return 0;
}
```

run_recovery_op

```
void ReplicatedBackend::run_recovery_op(
    PGBackend::RecoveryHandle *_h,
    int priority) {
    RPGHandle *h = static_cast<RPGHandle *>(_h);
// 这里的pushes和pulls就是上文中的PushOp和PullOp集合
    send_pushes(priority, h->pushes);
    send_pulls(priority, h->pulls);
    send_recovery_deletes(priority, h->deletes);
    delete h;
}
```

handle_pull

当主OSD将对象推送给从OSD后，从OSD需要调用handle_push来实现数据的写入；同样，当主OSD给从OSD发起拉取请求后，需要handle_pull处理对应的请求。

```
void ReplicatedBackend::handle_pull(pg_shard_t peer, PullOp &op, PushOp *reply) {
    const hobject_t &soid = op.soid;
    struct stat st;
// 1 验证对象是否存在
    int r = store->stat(ch, ghobject_t(soid), &st);

    if (r != 0) {
// 不存在返回空值
        prep_push_op_blank(soid, reply);
    } else {
        ObjectRecoveryInfo &recovery_info = op.recovery_info;
        ObjectRecoveryProgress &progress = op.recovery_progress;

// 2 第一次修复，且全部拷贝
        if (progress.first && recovery_info.size == ((uint64_t)-1)) {
            recovery_info.size = st.st_size;
            if (st.st_size) {
                interval_set<uint64_t> object_range;
                object_range.insert(0, st.st_size);
                // 添加数据
                recovery_info.copy_subset.intersection_of(object_range);
            } else {
                recovery_info.copy_subset.clear();
            }
            assert(recovery_info.clone_subset.empty());
        }
// 3 构建PushOp操作
        r = build_push_op(recovery_info, progress, 0, reply);
        if (r < 0) prep_push_op_blank(soid, reply);
    }
}
```

recover_replicas

```
uint64_t PrimaryLogPG::recover_replicas(uint64_t max, ThreadPool::TPHandle &handle,
                                        bool *work_started) {
    uint64_t started = 0;

// 1 获取一个Handle，用于Push和Pull
    PGBackend::RecoveryHandle *h = pgbackend->open_recovery_op();

    ceph_assert(!get_acting_recovery_backfill().empty());
    std::vector<std::pair<unsigned int, pg_shard_t>> replicas_by_num_missing,
        async_by_num_missing;
    replicas_by_num_missing.reserve(get_acting_recovery_backfill().size() - 1);
// 2 遍历所有正在修复的OSD
    for (auto &p : get_acting_recovery_backfill()) {
        if (p == get_primary()) {
            continue;
        }
        auto pm = recovery_state.get_peer_missing().find(p);
        ceph_assert(pm != recovery_state.get_peer_missing().end());
        auto nm = pm->second.num_missing();
        if (nm != 0) {
            if (is_async_recovery_target(p)) {
                async_by_num_missing.push_back(make_pair(nm, p));
            } else {
                replicas_by_num_missing.push_back(make_pair(nm, p));
            }
        }
    }
    // 排序函数
    auto func = [](const std::pair<unsigned int, pg_shard_t> &lhs,
                   const std::pair<unsigned int, pg_shard_t> &rhs) {
        return lhs.first < rhs.first;
    };

// 3 按缺失数量排序副本，从小到大
    std::sort(replicas_by_num_missing.begin(), replicas_by_num_missing.end(), func);
    std::sort(async_by_num_missing.begin(), async_by_num_missing.end(), func);
    replicas_by_num_missing.insert(replicas_by_num_missing.end(),
                                   async_by_num_missing.begin(), async_by_num_missing.end());

// 4 遍历副本
    for (auto &replica : replicas_by_num_missing) {
        pg_shard_t &peer = replica.second;
        ceph_assert(peer != get_primary());
        auto pm = recovery_state.get_peer_missing().find(peer);
        ceph_assert(pm != recovery_state.get_peer_missing().end());
        size_t m_sz = pm->second.num_missing();

        // oldest first!
        const pg_missing_t &m(pm->second);

// 4.1 遍历副本的缺失对象列表
        for (map<version_t, hobject_t>::const_iterator p = m.get_rmissing().begin();
             p != m.get_rmissing().end() && started < max;
             ++p) {
            handle.reset_tp_timeout();
            const hobject_t soid(p->second);

            if (recovery_state.get_missing_loc().is_unfound(soid)) {
                continue;
            }

// 4.2 对象版本比副本的last_backfill新，说明数据已经存在副本OSD上了
            const pg_info_t &pi = recovery_state.get_peer_info(peer);
            if (soid > pi.last_backfill) {
                if (!recovering.count(soid)) {
                    ceph_abort();
                }
                continue;
            }

            if (recovering.count(soid)) {
                continue;
            }

// 4.3 如果对象已被删除，准备删除操作
            if (recovery_state.get_missing_loc().is_deleted(soid)) {
                map<hobject_t, pg_missing_item>::const_iterator r = m.get_items().find(soid);
                started += prep_object_replica_deletes(soid, r->second.need, h, work_started);
                continue;
            }

// 4.4 如果对象是快照，并且快照的头部对象也在缺失列表中
            if (soid.is_snap() &&
                recovery_state.get_pg_log().get_missing().is_missing(
                    soid.get_head())) {
                continue;
            }

            if (recovery_state.get_pg_log().get_missing().is_missing(soid)) {
                continue;
            }
// 4.5 准备推送
            map<hobject_t, pg_missing_item>::const_iterator r = m.get_items().find(soid);
            started += prep_object_replica_pushes(soid, r->second.need, h, work_started);
        }
    }
```

```

// 5 推送
    pgbackend->run_recovery_op(h, recovery_state.get_recovery_op_priority());
    return started;
}
```

handle_push

```
void ReplicatedBackend::handle_push(
    pg_shard_t from, const PushOp &pop, PushReplyOp *response,
    ObjectStore::Transaction *t, bool is_repair) {
/*
从 PushOp 结构体中提取数据和进度信息。
first 表示是否是恢复过程的开始。
complete 表示数据和对象映射（omap）是否已经完全恢复。
clear_omap 表示是否需要清除对象映射。
data_zeros 是一个区间集合，用于记录数据中的零区间。
z_offset 和 z_length 分别表示数据恢复的起始偏移量和长度。
*/
    bufferlist data;
    data = pop.data;
    bool first = pop.before_progress.first;
    bool complete = pop.after_progress.data_complete &&
                    pop.after_progress.omap_complete;

    bool clear_omap = !pop.before_progress.omap_complete;
    interval_set<uint64_t> data_zeros;
    uint64_t z_offset = pop.before_progress.data_recovered_to;
    uint64_t z_length = pop.after_progress.data_recovered_to - pop.before_progress.data_recovered_to;
    if (z_length)
        data_zeros.insert(z_offset, z_length);

// 1 构造响应对象
    response->soid = pop.recovery_info.soid;

// 2 将数据提交给对象存储，应用更新
    submit_push_data(pop.recovery_info, first, complete,
        clear_omap, true, // must be replicate
        data_zeros, pop.data_included, data, pop.omap_header,
        pop.attrset, pop.omap_entries, t);

    if (complete) {
        if (is_repair) {
            get_parent()->inc_osd_stat_repaired();
        }
// 3 通知父对象本地修复完成
        get_parent()->on_local_recover(
            pop.recovery_info.soid,
            pop.recovery_info,
            ObjectContextRef(), // ok, is replica
            false,
            t);
    }
}
```

Backfill过程

Recovery过程通过日志修复对象，日志无法修复就通过Backfill过程直接拷贝数据。

数据结构

last_backfill 是backfill过程中，修复进程的指针。last_backfill初始化为 MIN对象，用来记录Backfiil过程中已修复的对象，last_backfill随着修复的进程不断推进，如果对象小于等于last_backfill，就是已经完成修复的对象，如果对象大于last_backfill，需要进一步判断是否需要修复。

如下，last_backfill从MIN推进，此时obj2对象以及obj1已经修复完成，之后的对象待修复。

Bacakfill对象列表	MIN	obj1(1,0)	obj2(1,1)	obj3(1,4)	obj4(1,5)	obj5(1,
			last_backfill			

```
struct BackfillInterval {
    eversion_t version;
    std::map<hobject_t,eversion_t> objects;
    hobject_t begin;
    hobject_t end;
}
BackfillInterval backfill_info;
std::map<pg_shard_t, BackfillInterval> peer_backfill_info;
```

backfill_info在primary上记录backfill的进度，peer_backfill_info（replica 的 backfill_info的集合）记录其余副本的进度。

earliset_peer_backfill() 和 earlist_backfill()

这两个函数非常像，但是返回结果的含义完全不同：

```
hobject_t PrimaryLogPG::earliest_peer_backfill() const {
    hobject_t e = hobject_t::get_max();
    for (const pg_shard_t &peer : get_backfill_targets()) {
        const auto iter = peer_backfill_info.find(peer);
        ceph_assert(iter != peer_backfill_info.end());
        e = std::min(e, iter->second.begin);
    }
    return e;
}

hobject_t PeeringState::earliest_backfill() const {
    hobject_t e = hobject_t::get_max();
    for (const pg_shard_t &bt : get_backfill_targets()) {
        const pg_info_t &pi = get_peer_info(bt);
        e = std::min(pi.last_backfill, e);
    }
    return e;
}
```

带peer的返回的是参与backfill的OSD中，最小的begin值。

而earliest_backfill返回的是参与backfill的OSD中，最小的last_backfill值。

流程

1. 首先，check指向所有副本的earliest_backfill()，然后判断check与primary上backfill_info.begin的大小关系。

1.1 check < backfill_info.begin 判定为多余对象，加入to_move中。

1.2 check >= backfill_info.begin 继续判断是否需要backfill

2. 如果check == backfill_info.begin 且 backfill_info.begin == peer_backfill_info.begin （replica上的对象和primary一致），进一步判断：如果版本号一致则不用backfill；不一致则加入need_ver_targs，表示需要修复。

3. 如果2不成立，表示当前check对象和primary不一致，进一步判断：

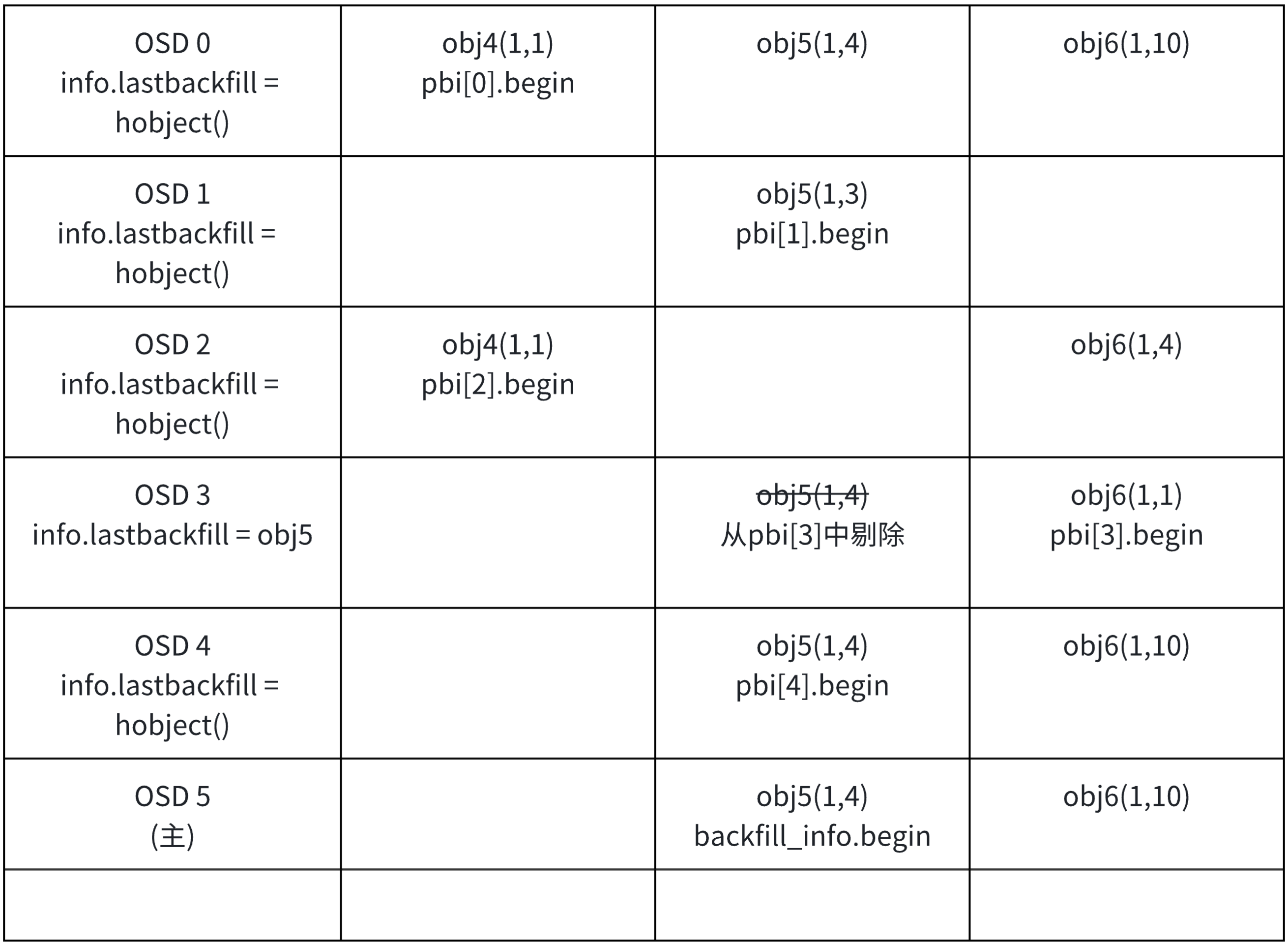
3.1 如果replica的last_backfill < backfill_info.begin 表示对象缺失，加入missing_targs；

3.2 如果replica的last_backfill >= backfill_info.begin 表示replica上的该对象已经做过backfill，加入skip_targs跳过对象

4. 更新指针 last_backfill_started = backfill_info.begin; backfill_info.pop_front(); 情况2中涉及到的replica全部更新peer_backfill_info.pop_front();

示例

如下图：每一行是一个BackfillInterval，所有replica的BackfillInterval集合被称为peer_backfill_info。



OSD 0 info.lastbackfill = hobject()	obj4(1,1) pbi[0].begin	obj5(1,4)	obj6(1,10)
OSD 1 info.lastbackfill = hobject()		obj5(1,3) pbi[1].begin	
OSD 2 info.lastbackfill = hobject()	obj4(1,1) pbi[2].begin		obj6(1,4)
OSD 3 info.lastbackfill = obj5		~~obj5(1,4)~~ 从pbi[3]中剔除	obj6(1,1) pbi[3].begin
OSD 4 info.lastbackfill = hobject()		obj5(1,4) pbi[4].begin	obj6(1,10)
OSD 5 (主)		obj5(1,4) backfill_info.begin	obj6(1,10)

初始时，lastbackfill初始化为hobject()，是个空对象，而last_backfill_started = earliest_backfill()也是一个空对象

上图中有一个特殊的OSD 3，它可能是之前做过backfill（听起来很怪，但不管出于什么原因，姑且当它backfill的进度目前超过了Primary），此时它的last_backfill并不是MIN对象，而是obj5。
第一次recover_backfill：

check指向earliest_peer_backfill()

首先，发现check（obj4）< backfill_info.begin，说明有冗余对象，查找和check相同的对象：OSD 0和OSD 2，加入to_move队列中准备删除。一直循环查找，直到冗余对象全部加入to_move队列，上图的情况，循环一次即可。

同时，被删除的对象会出队

```
peer_backfill_info.pop_front();  // 也就是peer_backfill_info.begin前移
last_backfill_started = check
```

OSD 0 info.lastbackfill = hobject()	~~obj4(1,1)~~	obj5(1,4) pbi[0].begin	obj6(1,10)
OSD 1 info.lastbackfill = hobject()		obj5(1,3) pbi[1].begin	
OSD 2 info.lastbackfill = hobject()	~~obj4(1,1)~~		obj6(1,4) pbi[2].begin
OSD 3 info.lastbackfill = obj5			obj6(1,1) pbi[3].begin
OSD 4 info.lastbackfill = hobject()		obj5(1,4) pbi[4].begin	obj6(1,10)
OSD 5 (主)		obj5(1,4) backfill_info.begin	obj6(1,10)
	last_backfill_started		

第二次循环 check = earliest_peer_backfill() ,此时一定大于等于backfill_info.begin，此时按OSD一个个对比：

1. 对于begin对象等于check对象的OSD: 0, 1, 4。判断版本是否正确

1.1 如果不正确就加入修复队列need_ver_targs

1.2 正确则加入keep_ver_targs，不需要修复

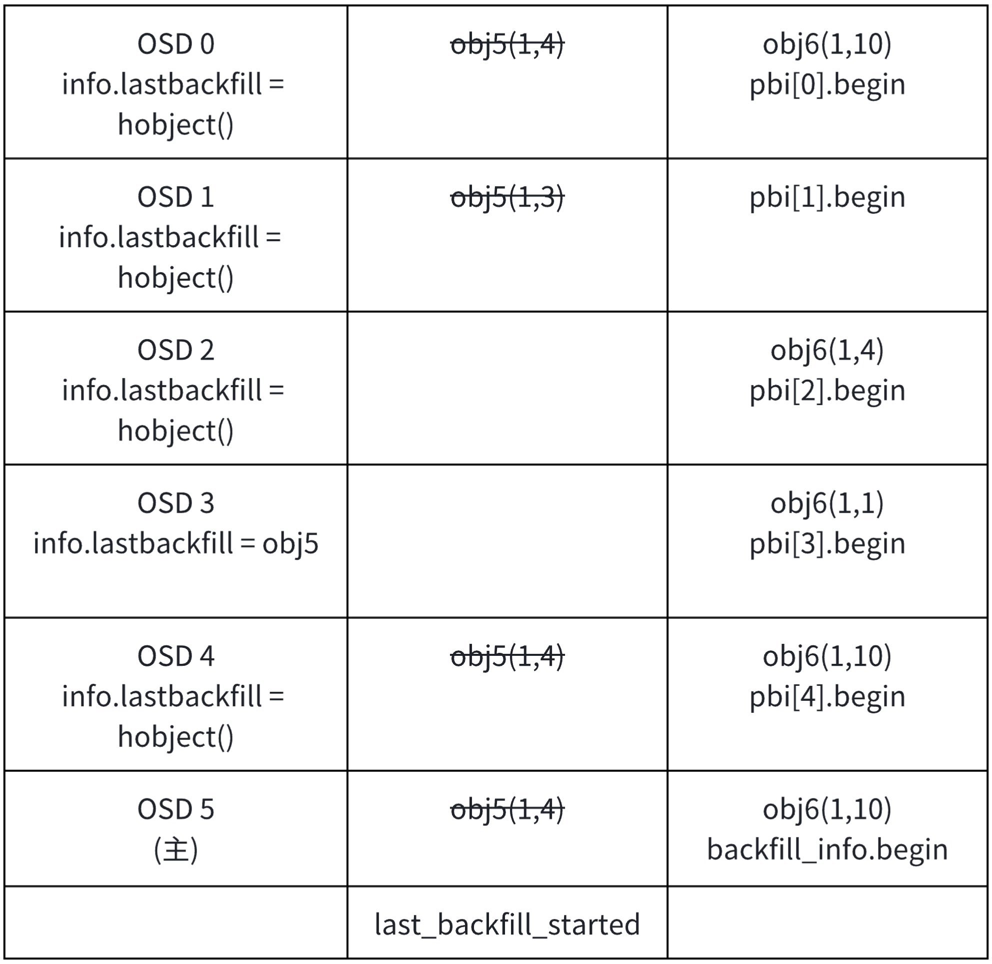
2. 对于begin对象不等于check对象的OSD: 2, 3。根据last_backfill判断其修复进度

2.1 对于OSD 2而言，其last_backfill小于backfill_info.begin，判定为缺失了obj5对象，加入missing列表

2.1 对于OSD 3而言，其last_backfill等于backfill_info.begin，判断其已经做过backfill，加入skip队列

下一步，need_ver_targs和keep_ver_targs中的begin全部前移

```
peer_backfill_info.pop_front();
last_backfill_started = backfill_info.begin
```



OSD 0 info.lastbackfill = hobject()	~~obj5(1,4)~~	obj6(1,10) pbi[0].begin
OSD 1 info.lastbackfill = hobject()	~~obj5(1,3)~~	pbi[1].begin
OSD 2 info.lastbackfill = hobject()		obj6(1,4) pbi[2].begin
OSD 3 info.lastbackfill = obj5		obj6(1,1) pbi[3].begin
OSD 4 info.lastbackfill = hobject()	~~obj5(1,4)~~	obj6(1,10) pbi[4].begin
OSD 5 (主)	~~obj5(1,4)~~	obj6(1,10) backfill_info.begin
	last_backfill_started	

假设最大操作数是1，而刚刚我们循环了两次，第一次找了一列to_move对象，第二次处理了一列obj5对象。

1. 对于to_move对象的处理，不算一次”操作“，因为删除是异步的，且消耗资源较少。
2. 对于obj5，整理出了四种操作类型，分别对应队列：keep_ver_targs, need_ver_targs, skip_targs, missing_targs

每一次循环中，如果本次循环处理的不是to_move，那么会处理需要修复的对象：

```
// 构造PushOp做backfill
if (!need_ver_targs.empty() || !missing_targs.empty()) {
    ObjectContextRef obc = get_object_context(backfill_info.begin, false);
    prep_backfill_object_push(backfill_info.begin, obj_v, obc, all_push, h);
    ops ++; // 操作数
}
```

如果达到最大操作数后退出循环，并首先处理to_move队列，通知replica删除对象

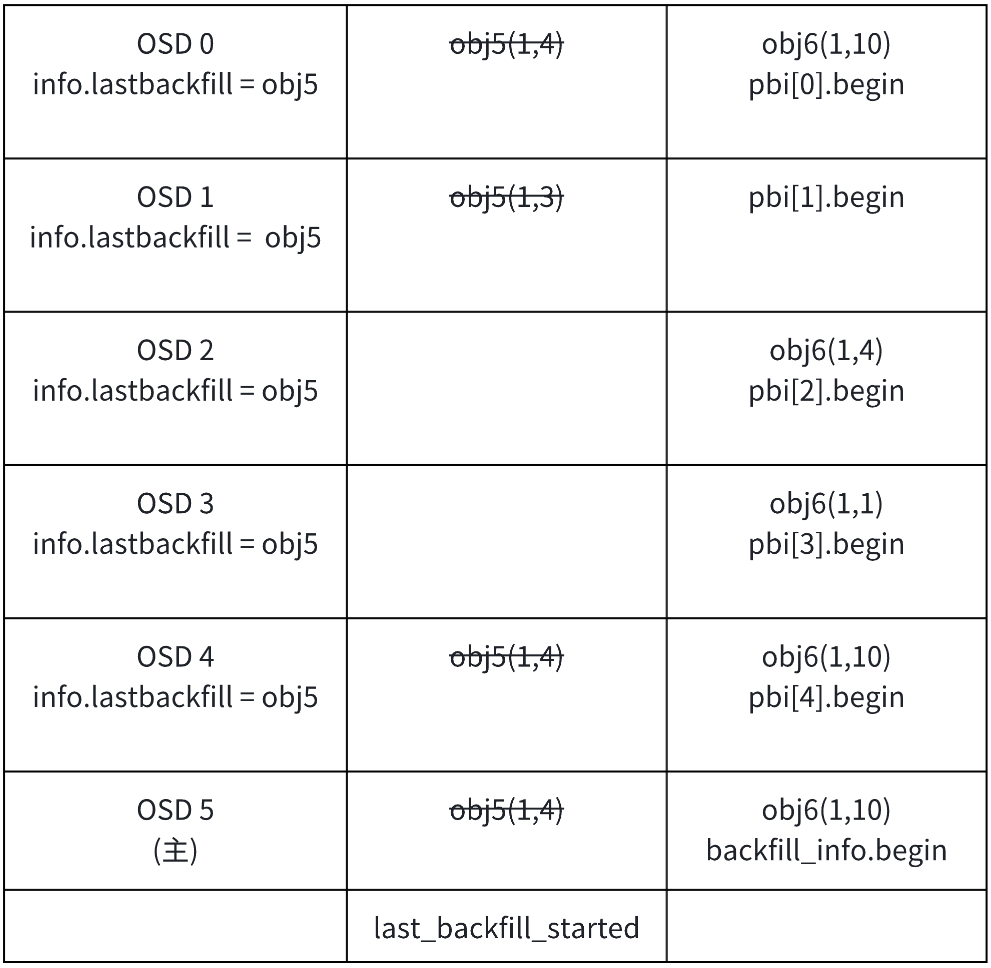
```
pg_shard_t peer = to_remove[i].get<2>();
```

```
reqs[peer] = new MOSDPGBackfillRemove(spg_t(info.pgid.pgid, peer.shard),
get_osdmap_epoch());
for (auto p : reqs) {
    osd->send_message_osd_cluster(p.first.osd, p.second, get_osdmap_epoch());
}
// 通知过后，紧接着将准备好的PushOp发出去
pgbackend->run_recovery_op(h, recovery_state.get_recovery_op_priority());
```

完成backfill后，更新replica的last_backfill：

```
hobject_t new_last_backfill = recovery_state.earliest_backfill();
recovery_state.update_peer_last_backfill(bt, new_last_backfill);
```

第二次recover_backfill



OSD 0 info.lastbackfill = obj5	~~obj5(1,4)~~	obj6(1,10) pbi[0].begin
OSD 1 info.lastbackfill = obj5	~~obj5(1,3)~~	pbi[1].begin
OSD 2 info.lastbackfill = obj5		obj6(1,4) pbi[2].begin
OSD 3 info.lastbackfill = obj5		obj6(1,1) pbi[3].begin
OSD 4 info.lastbackfill = obj5	~~obj5(1,4)~~	obj6(1,10) pbi[4].begin
OSD 5 (主)	~~obj5(1,4)~~	obj6(1,10) backfill_info.begin
	last_backfill_started	

此时check = earliest_peer_backfill()等于obj6。和obj5一样的处理思路：

1. 对于和check相等的OSD：0 2 3 4：0，4版本正确，不用修复，2，3版本错误，需要修复

2. 和check不相等的OSD 1（此时它已经是空对象了），判断它的last_backfill < backfill_info.begin，判定为缺失对象，加入missing中。

同样的，它们的begin会再次前移，直到backfill_info为空或者达到最大并发处理数

源码

```
uint64_t PrimaryLogPG::recover_backfill(
    uint64_t max,
    ThreadPool::TPHandle &handle, bool *work_started) {
    ceph_assert(!get_backfill_targets().empty());

// 1 初始化backfill区间
    // Initialize from prior backfill state
    if (new_backfill) {
        // on_activate() was called prior to getting here
        ceph_assert(last_backfill_started == recovery_state.earliest_backfill());
        new_backfill = false;

        // initialize BackfillIntervals
        for (set<pg_shard_t>::const_iterator i = get_backfill_targets().begin();
             i != get_backfill_targets().end();
             ++i) {
            peer_backfill_info[*i].reset(
                recovery_state.get_peer_info(*i).last_backfill);
        }
        backfill_info.reset(last_backfill_started);

        backfills_in_flight.clear();
        // 保存着需要删除的对象
        pending_backfill_updates.clear();
    }

// 2 更新backfill_info.begin, update_range更新需要进行Backfill操作的对象列表
    backfill_info.begin = last_backfill_started;
    update_range(&backfill_info, handle);

    unsigned ops = 0;
    vector<boost::tuple<hobject_t, eversion_t, pg_shard_t>> to_remove;
    set<hobject_t> add_to_stat;

// 3 去掉不需要backfill的区间
    for (set<pg_shard_t>::const_iterator i = get_backfill_targets().begin();
         i != get_backfill_targets().end();
         ++i) {
        peer_backfill_info[*i].trim_to(
            std::max(
                recovery_state.get_peer_info(*i).last_backfill,
                last_backfill_started));
    }
    backfill_info.trim_to(last_backfill_started);

    PGBackend::RecoveryHandle *h = pgbackend->open_recovery_op();
    while (ops < max) {
// 4 当前区间完成backfill后，添加新的backfill区间
        if (backfill_info.begin <= earliest_peer_backfill() &&
            !backfill_info.extends_to_end() && backfill_info.empty()) {
            hobject_t next = backfill_info.end;
            backfill_info.reset(next);
            backfill_info.end = hobject_t::get_max();
            update_range(&backfill_info, handle);
            backfill_info.trim();
        }

        bool sent_scan = false;
        for (set<pg_shard_t>::const_iterator i = get_backfill_targets().begin();
             i != get_backfill_targets().end();
             ++i) {
            pg_shard_t bt = *i;
            BackfillInterval &pbi = peer_backfill_info[bt];

// 5 当前backfillInterval没有需要回填的区间，对相应副本发起扫描操作，更新数据
            if (pbi.begin <= backfill_info.begin &&
                !pbi.extends_to_end() && pbi.empty()) {
                epoch_t e = get_osdmap_epoch();
                MOSDPGScan *m = new MOSDPGScan(
                    MOSDPGScan::OP_SCAN_GET_DIGEST, pg_whoami, e, get_last_peering_reset(),
                    spg_t(info.pgid.pgid, bt.shard),
                    pbi.end, hobject_t());

                if (cct->_conf->osd_op_queue == "mclock_scheduler") {
                    /* This guard preserves legacy WeightedPriorityQueue behavior for
                     * now, but should be removed after Reef */
                    m->set_priority(recovery_state.get_recovery_op_priority());
                }
                osd->send_message_osd_cluster(bt.osd, m, get_osdmap_epoch());
                ceph_assert(waiting_on_backfill.find(bt) == waiting_on_backfill.end());
                waiting_on_backfill.insert(bt);
                sent_scan = true;
            }
        }

        // Count simultaneous scans as a single op and let those complete
// 6 获取OSd的对象列表后，对比当前主OSD的对象列表来进行修复
        if (sent_scan) {
            ops++;
            start_recovery_op(hobject_t::get_max()); // XXX: was pbi.end
            break;
        }

        if (backfill_info.empty() && all_peer_done()) {
            break;
        }

        // Get object within set of peers to operate on and
        // the set of targets for which that object applies.
        hobject_t check = earliest_peer_backfill();
// 7 check指向当前OSD中最小的需要进行Backfill操作的对象
        if (check < backfill_info.begin) {
// 7.1 冗余对象，加入到to_remove队列中
            set<pg_shard_t> check_targets;
            for (set<pg_shard_t>::const_iterator i =
get_backfill_targets().begin();
                 i != get_backfill_targets().end();
                 ++i) {
                pg_shard_t bt = *i;
                BackfillInterval &pbi = peer_backfill_info[bt];
                if (pbi.begin == check)
                    check_targets.insert(bt);
            }
            ceph_assert(!check_targets.empty());

            for (set<pg_shard_t>::iterator i = check_targets.begin();
                 i != check_targets.end();
                 ++i) {
                pg_shard_t bt = *i;
                BackfillInterval &pbi = peer_backfill_info[bt];
                ceph_assert(pbi.begin == check);

                to_remove.push_back(boost::make_tuple(check,
pbi.objects.begin()->second, bt));
                pbi.pop_front();
            }

            last_backfill_started = check;
        } else {
// 7.2 check等于backfill_info_begin，判断对象是否需要backfill
            eversion_t &obj_v = backfill_info.objects.begin()->second;
            vector<pg_shard_t> need_ver_targs, missing_targs, keep_ver_targs, skip_targs;
            for (set<pg_shard_t>::const_iterator i = get_backfill_targets().begin();
                 i != get_backfill_targets().end();
                 ++i) {
                pg_shard_t bt = *i;
                BackfillInterval &pbi = peer_backfill_info[bt];
                // Find all check peers that have the wrong version
                if (check == backfill_info.begin && check == pbi.begin) {
                    if (pbi.objects.begin()->second != obj_v) {
                        need_ver_targs.push_back(bt);
                    } else {
                        keep_ver_targs.push_back(bt);
                    }
                } else {
                    const pg_info_t &pinfo = recovery_state.get_peer_info(bt);
                    if (backfill_info.begin > pinfo.last_backfill)
                        missing_targs.push_back(bt);
                    else
                        skip_targs.push_back(bt);
                }
            }

            if (!keep_ver_targs.empty()) {
                // These peers have version obj_v
                // assert(!waiting_for_degraded_object.count(check));
            }
// 8 对于keep_ver_targs中的OSD，不需要进行Backfill操作
// 对于need_ver_targs和missing_targs中的OSD，需要进行Backfill操作，准备PushOp
            if (!need_ver_targs.empty() || !missing_targs.empty()) {
                ObjectContextRef obc = get_object_context(backfill_info.begin, false);
                ceph_assert(obc);
                if (obc->get_recovery_read()) {
                    vector<pg_shard_t> all_push = need_ver_targs;
                    all_push.insert(all_push.end(), missing_targs.begin(), missing_targs.end());

                    handle.reset_tp_timeout();
                    int r = prep_backfill_object_push(backfill_info.begin, obj_v, obc, all_push, h);
                    if (r < 0) {
                        *work_started = true;
                        break;
                    }
                    ops++;
                } else {
                    *work_started = true;
                    break;
                }
            }

            last_backfill_started = backfill_info.begin;
            add_to_stat.insert(backfill_info.begin);
            backfill_info.pop_front();
            vector<pg_shard_t> check_targets = need_ver_targs;
            check_targets.insert(check_targets.end(), keep_ver_targs.begin(), keep_ver_targs.end());
            for (vector<pg_shard_t>::iterator i = check_targets.begin();
                 i != check_targets.end();
                 ++i) {
                pg_s     hard_t bt = *i;
                BackfillInterval &pbi = peer_backfill_info[bt];
                pbi.pop_front();
            }
        }
    }

    for (set<hobject_t>::iterator i = add_to_stat.begin();
         i != add_to_stat.end();
         ++i) {
        ObjectContextRef obc = get_object_context(*i, false);
        ceph_assert(obc);
        pg_stat_t stat;
        add_object_context_to_pg_stat(obc, &stat);
        pending_backfill_updates[*i] = stat;
    }

// 9 冗余对象
    map<pg_shard_t, MOSDPGBackfillRemove *> reqs;
    for (unsigned i = 0; i < to_remove.size(); ++i) {
        handle.reset_tp_timeout();
        const hobject_t &oid = to_remove[i].get<0>();
        eversion_t v = to_remove[i].get<1>();
        pg_shard_t peer = to_remove[i].get<2>();
        MOSDPGBackfillRemove *m;
        auto it = reqs.find(peer);
        if (it != reqs.end()) {
            m = it->second;
        } else {
            m = reqs[peer] = new MOSDPGBackfillRemove(
                spg_t(info.pgid.pgid, peer.shard),
                get_osdmap_epoch());
            if (cct->_conf->osd_op_queue == "mclock_scheduler") {
                m->set_priority(recovery_state.get_recovery_op_priority());
            }
        }
        m->ls.push_back(make_pair(oid, v));

        if (oid <= last_backfill_started)
            pending_backfill_updates[oid]; // add empty stat!
    }
    for (auto p : reqs) {
        osd->send_message_osd_cluster(p.first.osd, p.second, get_osdmap_epoch());
    }

// 10 发送PushOp
    pgbackend->run_recovery_op(h, recovery_state.get_recovery_op_priority());

// 11 处理已完成backfill的对象，更新backfill位置和状态
    hobject_t next_backfill_to_complete = backfills_in_flight.empty() ? backfill_pos : *(backfills_in_flight.begin());
    hobject_t new_last_backfill = recovery_state.earliest_backfill();
    for (map<hobject_t, pg_stat_t>::iterator i =
             pending_backfill_updates.begin();
         i != pending_backfill_updates.end() &&
         i->first < next_backfill_to_complete;
         pending_backfill_updates.erase(i++)) {
        ceph_assert(i->first > new_last_backfill);
        recovery_state.update_complete_backfill_object_stats(
            i->first,
            i->second);
        new_last_backfill = i->first;
    }

    ceph_assert(!pending_backfill_updates.empty() ||
                new_last_backfill == last_backfill_started);
    if (pending_backfill_updates.empty() &&
        backfill_pos.is_max()) {
        ceph_assert(backfills_in_flight.empty());
        new_last_backfill = backfill_pos;
        last_backfill_started = backfill_pos;
    }

// 12 更新各个OSD的backfill位置
```

```
    for (set<pg_shard_t>::const_iterator i = get_backfill_targets().begin();
         i != get_backfill_targets().end();
         ++i) {
        pg_shard_t bt = *i;
        const pg_info_t &pinfo = recovery_state.get_peer_info(bt);

        if (new_last_backfill > pinfo.last_backfill) {
            recovery_state.update_peer_last_backfill(bt, new_last_backfill);
            epoch_t e = get_osdmap_epoch();
            MOSDPGBackfill *m = NULL;
// 12.1 如果值为MAX，说明backfill完成，发送OP_BACKFILL_FINISH
            if (pinfo.last_backfill.is_max()) {
                m = new MOSDPGBackfill(
                    MOSDPGBackfill::OP_BACKFILL_FINISH,
                    e,
                    get_last_peering_reset(),
                    spg_t(info.pgid.pgid, bt.shard));
                // Use default priority here, must match sub_op priority
                start_recovery_op(hobject_t::get_max());
            } else {
// 12.2 否则发送OP_BACKFILL_PROGRESS
                m = new MOSDPGBackfill(
                    MOSDPGBackfill::OP_BACKFILL_PROGRESS,
                    e,
                    get_last_peering_reset(),
                    spg_t(info.pgid.pgid, bt.shard));
                // Use default priority here, must match sub_op priority
            }
            m->last_backfill = pinfo.last_backfill;
            m->stats = pinfo.stats;

            if (cct->_conf->osd_op_queue == "mclock_scheduler") {
                /* This guard preserves legacy WeightedPriorityQueue behavior for
                 * now, but should be removed after Reef */
                m->set_priority(recovery_state.get_recovery_op_priority());
            }
// 12.3 发送消息
            osd->send_message_osd_cluster(bt.osd, m, get_osdmap_epoch());
        }
    }

    if (ops)
        *work_started = true;
    return ops;
}
```

对象排序方法

做backfill时，即使各个OSD中相同版本号不同，也能保证各个OSD内对象的相对顺序一致。

```
// 首先对对象计算出对象的唯一hash值
// 传入参数中一共就两个成员变量：对象名，快照序号
  explicit hobject_t(const sobject_t &o) :
    oid(o.oid), snap(o.snap), max(false), pool(POOL_META) {
    set_hash(std::hash<sobject_t>()(o));
  }

  namespace std {
template<> struct hash<hobject_t> {
  size_t operator()(const hobject_t &r) const {
    static rjhash<uint64_t> RJ;
    return RJ(r.get_hash() ^ r.snap);
  }
};
}

inline uint64_t rjhash64(uint64_t key) {
  key = (~key) + (key << 21); // key = (key << 21) - key - 1;
  key = key ^ (key >> 24);
  key = (key + (key << 3)) + (key << 8); // key * 265
  key = key ^ (key >> 14);
  key = (key + (key << 2)) + (key << 4); // key * 21
  key = key ^ (key >> 28);
  key = key + (key << 31);
  return key;
}

// 然后根据hash值，计算出用于排序的序号
  void build_hash_cache() {
    nibblewise_key_cache = _reverse_nibbles(hash);
    hash_reverse_bits = _reverse_bits(hash);
  }

uint32_t reverse_bits(uint32_t v) {
  if (v == 0)
    return v;

  /* reverse bits
   * swap odd and even bits
   */
  v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);
  /* swap consecutive pairs */
  v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);
  /* swap nibbles ... */
  v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);
  /* swap bytes */
  v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);
  /* swap 2-byte long pairs */
  v = ( v >> 16             ) | ( v               << 16);

  return v;
}

// 对象放入backfill的map中后，根据以下规则排列，一般到reverse_bits就能计算出结果
  auto operator<=>(const hobject_t &rhs) const noexcept {
    auto cmp = max <=> rhs.max;            // 比较max与rhs.max，相等返回0，不相等返回比较结果
    if (cmp != 0) return cmp;                     // 不相等返回比较结果
    cmp = pool <=> rhs.pool;                  // 比较pool
    if (cmp != 0) return cmp;
    cmp = get_bitwise_key() <=> rhs.get_bitwise_key();    // reverse_bits
    if (cmp != 0) return cmp;
    cmp = nspace <=> rhs.nspace;           // 命名空间
    if (cmp != 0) return cmp;
    if (!(get_key().empty() && rhs.get_key().empty())) {
      cmp = get_effective_key() <=> rhs.get_effective_key(); // 自定义key和对象名
      if (cmp != 0) return cmp;
    }
    cmp = oid <=> rhs.oid;                     // 对象id
    if (cmp != 0) return cmp;
    return snap <=> rhs.snap;                // 快照信息
  }

```