一致性检查

概念

现实中，我们的计算机组件都是不完美的，都有可能产生静默数据错误，即便通过数据修复，每个副本仍然可能存在本地数据错误，副本之间也仍然可能会有数据一致性问题。

Scrub 通过对比各个对象副本的数据和元数据，实现副本的一致性检查。它不是一种实时的纠错机制，而是一种事后的检查机制。

按照扫描的内容分为两种方式：

浅扫描：Scrub，通过检查元数据来检查数据的一致性，如果元数据一致就通过检查

深扫描：deep-Scrub，进一步检查对象的数据内容是否一致，要扫描磁盘上的所有数据，比较耗时，占用的资源也更多。

Scrub有在线扫描（不影响系统正常运行）和离线扫描（需要系统暂停业务）两种方式，一般情况下我们使用在线扫描，这个过程不会中断系统，但是被扫描的对象会被锁定暂停访问，直到对象完成Scrub操作后才能解锁。

扫描粒度

如何确定Scrub的扫描粒度，

1. 以对象为粒度，集群中往往存在大量的对象，逐个对象执行scrub的效率太低了，且对象的生命周期不固定，对象可能对是被删除和创建。
2. 以PG为粒度， 集群中PG的数量比较固定，生命周期也更稳定，OSD的读写等都围绕PG进行组织，所以使用PG作为粒度是比较合理的
3. 2仍然有问题，如果一个PG中含有大量的对象，这种锁粒度会导致PG的写请求被长时间阻塞，于是有了新的方案：针对当前PG，一次只扫描部分对象，分批次扫描完整个PG。这个方案的可取之处在于：PG内的对象顺序基本保持一致，新对象的加入不会打乱旧对象之间的相对顺序，也就是说即使有对象创建或删除，也能保证同一个对象不被重复扫描，而未被扫描的对象，会被安排在下一轮的扫描中被检查。

ScrubMap

保存校验对象以及相应校验数据

```
struct ScrubMap {
    struct object {
```

```
        std::map<std::string, ceph::buffer::ptr, std::less<>> attrs; // 对象的属性
        uint64_t size;                                  // 对象的大小
        __u32 omap_digest;                              // 对象映射的CRC32C校验和
        __u32 digest;                                   // 数据的CRC32C校验和
        bool digest_present : 1;                        // 表示数据校验和是否存在
        bool omap_digest_present : 1;                   // 表示omap校验和是否存在
        ...
    };
    ...
    std::map<hobject_t, object> objects; // 需要校验的对象 -> 校验信息的映射
};
```

调度

调度总流程

```
# 注册定时任务，C_Tick_WithoutOSDLock回调函数是tick_without_osd_lock
OSD::init
>> tick_timer_without_osd_lock.add_event_after(get_tick_interval(),
          new C_Tick_WithoutOSDLock(this));

OSD::tick_without_osd_lock
>> OSD::sched_scrub

OSD::sched_scrub
>> Scrub::schedule_result_t ScrubQueue::select_pg_and_scrub
>>>> Scrub::schedule_result_t ScrubQueue::select_from_group
>>>>>> Scrub::schedule_result_t OSDService::initiate_a_scrub
>>>>>>>> pg::sched_scrub

pg::sched_scrub
>> osd->queue_for_scrub(this, Scrub::scrub_prio_t::low_priority);
```

OSD::sched_scrub

控制启动时机

```
void OSD::sched_scrub() {
// 1 获取调度器
    auto &scrub_scheduler = service.get_scrub_services();
// 2 有被锁定的对象
```

```
    if (auto blocked_pgs = scrub_scheduler.get_blocked_pgs_count();
        blocked_pgs > 0) {
        // 此OSD管理的某些PG在清除过程中被锁定的对象阻止。
        // 这意味着我们现在可能没有清理所需的资源。
    }

// 3 没有可用资源进行Scrub
    if (!scrub_scheduler.can_inc_scrubs()) {
        return;
    }
    /*
    scrub_scheduler.can_inc_scrubs():
      if (scrubs_local + scrubs_remote < conf()->osd_max_scrubs) {
          return true;
      }
    */

    // 如果有一个PG正在尝试保留清理副本资源
    // 我们应该等待，不要开始新的Scrub
    // bool is_reserving_now() const { return a_pg_is_reserving; }
    if (scrub_scheduler.is_reserving_now()) {
        return;
    }

    Scrub::ScrubPreconds env_conditions;
// 4 如果处于恢复期间，判断是否允许在回复期间进行Scrub
    if (service.is_recovery_active() && !cct->_conf->osd_scrub_during_recovery) {
        if (!cct->_conf->osd_repair_during_recovery) {
            return;
        }
        env_conditions.allow_requested_repair_only = true;
    }

    if (g_conf()->subsys.should_gather<ceph_subsys_osd, 20>()) {
        auto all_jobs = scrub_scheduler.list_registered_jobs();
    }
// 5 选择一个PG开始检查
    auto was_started = scrub_scheduler.select_pg_and_scrub(env_conditions);
}

```

```
Scrub::schedule_result_t ScrubQueue::select_pg_and_scrub(
  Scrub::ScrubPreconds& preconds) {

```

```
  utime_t now_is = time_now();
// 1 检查是否在允许的时间段内，
// 检查now是否在配置中的区间：[begin_time, end_time)或[end_time, begin_time)
  preconds.time_permit = scrub_time_permit(now_is);
// 2 检查负载是否允许
  preconds.load_is_low = scrub_load_below_threshold();
  preconds.only_deadlined = !preconds.time_permit || !preconds.load_is_low;

  std::unique_lock lck{jobs_lock};

// 3 检查处罚列表，如果某个作业已到截止时间，或已被更新，就移除
  scan_penalized(restore_penalized, now_is);
  restore_penalized = false;

// 4 remove the 'updated' flag from all entries
  std::for_each(to_scrub.begin(),
        to_scrub.end(),
        [](const auto& jobref) -> void { jobref->updated = false; });

// 5 将那些在之前的Scrub尝试中失败的PG移动到处罚列表中
  move_failed_pgs(now_is);

  // 从两个列表中收集所有有效且成熟的作业副本
  auto to_scrub_copy = collect_ripe_jobs(to_scrub, now_is);
  auto penalized_copy = collect_ripe_jobs(penalized, now_is);
  lck.unlock(); // 解锁互斥锁

// 6 首先尝试从常规队列中选择一个 PG 进行Scrub
  auto res = select_from_group(to_scrub_copy, preconds, now_is);

// 7 如果常规队列中没有准备好的作业，且处罚队列不为空，则尝试从处罚队列中选择
  if (res == Scrub::schedule_result_t::none_ready && !penalized_copy.empty()) {
    res = select_from_group(penalized_copy, preconds, now_is);
    restore_penalized = true; // 设置恢复处罚作业的标志为 true
  }

  return res;
}
```

> penalized 直译为“处罚”
>
> ceph中指的是某些 PG 因为特定的问题或错误而被标记为需要特别处理的状态。这些问题可能包括但不限于数据损坏、副本不一致、或者在之前的Scrub过程中发现的错误。处罚是一种机制，用于确保这些有问题的 PGs能够被优先处理，以便修复或恢复它们的状态。

> 处罚的 PG 可能会被暂时从正常的Scrub队列中移除，并放入一个特殊的处罚列表中。这样，系统就可以在处理正常Scrub任务的同时，特别关注这些需要额外注意的 PG。处罚列表中的 PGs 会在一定条件下被重新评估，例如在负载较低或者在特定的时间窗口内，以便进行必要的Scrub或修复操作。
>
> 在函数中，处罚的 PGs 会在特定条件下被考虑进行Scrub，例如当常规的Scrub队列中没有其他 PG 准备好进行Scrub时。这种机制有助于确保 Ceph 集群中的数据完整性和可靠性。

```
Scrub::schedule_result_t ScrubQueue::select_from_group(
  ScrubQContainer& group,
  const Scrub::ScrubPreconds& preconds,
  utime_t now_is)
{
  for (auto& candidate : group) {
    if (preconds.only_deadlined && (candidate->schedule.deadline.is_zero() ||
                    candidate->schedule.deadline >= now_is)) {
      continue;
    }
    switch (
      osd_service.initiate_a_scrub(candidate->pgid,
                   preconds.allow_requested_repair_only)) {
      case Scrub::schedule_result_t::scrub_initiated:
    return Scrub::schedule_result_t::scrub_initiated;
    // 其它的case都是return none_ready，省略不看了
    }
  }
  return Scrub::schedule_result_t::none_ready;
}

```

```
Scrub::schedule_result_t OSDService::initiate_a_scrub(spg_t pgid,
                bool allow_requested_repair_only) {

    // 我们有一个候选人要淘汰。我们需要一些PG信息来了解是否允许Scrub

    PGRef pg = osd->lookup_lock_pg(pgid);
    if (!pg) {
        return Scrub::schedule_result_t::no_such_pg;
    }

    // This has already started, so go on to the next scrub job
    if (pg->is_scrub_queued_or_active()) {
        pg->unlock();
```

```
        return Scrub::schedule_result_t::already_started;
    }
    // Skip other kinds of scrubbing if only explicitly requested repairing is allowed
    if (allow_requested_repair_only && !pg->get_planned_scrub().must_repair) {
        pg->unlock();
        return Scrub::schedule_result_t::preconditions;
    }

    auto scrub_attempt = pg->sched_scrub();
    pg->unlock();
    return scrub_attempt;
}
```

pg::sched_scrub

设置Scrub任务参数，并完成**本地**资源预约

```
Scrub::schedule_result_t PG::sched_scrub() {
    using Scrub::schedule_result_t;
    ceph_assert(ceph_mutex_is_locked(_lock));
    ceph_assert(m_scrubber);
// 1 状态检测
    if (is_scrub_queued_or_active()) {
        return schedule_result_t::already_started;
    }
    if (!is_primary() || !is_active() || !is_clean()) {
        return schedule_result_t::bad_pg_state;
    }
    if (state_test(PG_STATE_SNAPTRIM) || state_test(PG_STATE_SNAPTRIM_WAIT)) {
        return schedule_result_t::bad_pg_state;
    }

// 2 决定Scrub的深浅
    auto updated_flags = validate_scrub_mode();
    if (!updated_flags) { // 无法开始Scrub
        return schedule_result_t::preconditions;
    }
// 3 本地资源预约
    if (!m_scrubber->reserve_local()) {
        return schedule_result_t::no_local_resources;
    }

    m_planned_scrub = *updated_flags;

```

```
    state_clear(PG_STATE_REPAIR);

    m_scrubber->set_op_parameters(m_planned_scrub);

// 4 加入Scrub队列，等待调度
    osd->queue_for_scrub(this, Scrub::scrub_prio_t::low_priority);
    return schedule_result_t::scrub_initiated;
}
```

```
std::optional<requested_scrub_t> PG::validate_scrub_mode() const {
// 1 是否允许Scrub和深层Scrub，检查是否有深层错误，是否尝试自动修复
    const bool allow_shallow_scrub =
        !(get_osdmap()->test_flag(CEPH_OSDMAP_NOSCRUB) ||
          pool.info.has_flag(pg_pool_t::FLAG_NOSCRUB));
    const bool allow_deep_scrub =
        !(get_osdmap()->test_flag(CEPH_OSDMAP_NODEEP_SCRUB) ||
          pool.info.has_flag(pg_pool_t::FLAG_NODEEP_SCRUB));
    const bool has_deep_errors = (info.stats.stats.sum.num_deep_scrub_errors > 0);
    const bool try_to_auto_repair = (cct->_conf->osd_scrub_auto_repair &&
                                     get_pgbackend()->auto_repair_supported());

// 2 PG正在恢复中，且不允许在此期间进行Scrub
    const bool prevented_by_recovery =
        osd->is_recovery_active() && !cct->_conf->osd_scrub_during_recovery &&
        (!cct->_conf->osd_repair_during_recovery || !m_planned_scrub.must_repair);
    if (prevented_by_recovery) {
        return std::nullopt;
    }

// 3 检查是否到了时间，内部逻辑：只要计划自动修复或时间已到或有深层错误
// 或使用随机值达到配置范围，都返回true
    const bool time_for_deep = is_time_for_deep(allow_deep_scrub,
                                                allow_shallow_scrub,
                                                has_deep_errors,
                                                m_planned_scrub);
    std::optional<requested_scrub_t> upd_flags;

// 4 验证计划Scrub
    if (m_planned_scrub.must_scrub) {
        upd_flags = validate_initiated_scrub(allow_deep_scrub,
                                             try_to_auto_repair,
```

```
                                              time_for_deep,
                                              has_deep_errors,
                                              m_planned_scrub);
    } else {
// 5 验证周期性Scrub
        ceph_assert(!m_planned_scrub.must_deep_scrub);
        upd_flags = validate_periodic_mode(allow_deep_scrub,
                                           try_to_auto_repair,
                                           allow_shallow_scrub,
                                           time_for_deep,
                                           has_deep_errors,
                                           m_planned_scrub);
        if (!upd_flags) { // 取消周期性scrub
            return std::nullopt;
        }
    }
    upd_flags->need_auto = false;
    return upd_flags;
}
```

validate_initiated_scrub和validate_periodic_mode内部会确认Scrub类型，依据下面这张表：

```
   Periodic  type     | none    | no-scrub  | no-scrub+no-deep | no-deep
   ------------------------------------------------------------------------
   ------------------------------------------------------------------------
   periodic           | shallow | x         | x                | shallow
   ------------------------------------------------------------------------
   periodic + t.f.deep| deep    | deep      | x                | shallow
   ------------------------------------------------------------------------
   initiated          | shallow | shallow   | shallow          | shallow
   ------------------------------------------------------------------------
   init. + t.f.deep   | deep    | deep      | shallow          | shallow
   ------------------------------------------------------------------------
   initiated deep     | deep    | deep      | deep             | deep
   ------------------------------------------------------------------------
   "periodic"       - 周期性Scrub       - if !must_scrub && !must_deep_scrub;
   "initiated deep" - 手动发起的深层Scrub - if must_scrub && must_deep_scrub;
   "initiated"      - 手动发起的浅层Scrub - if must_scrub && !must_deep_scrub;
   "t.f.deep"   - triggered deep scrub 触发深层Scrub，意味着系统检测到了需要进行深层
Scrub
```

这张表展示在什么操作类型下，遇到什么PG状态/配置会发起怎样程度的Scrub，比如 `init. + t.f.deep` 操作遇到none，执行深层Scrub，而遇到no-deep状态执行浅层Scrub。

执行

通过比较对象各个副本上的元数据和数据，来完成元数据和数据的校验。

状态机

Scrub状态机的转换是这样的一个套路：

1.调用OSD层的queue_for_srcub_xxx

```
m_osds->queue_for_scrub_resched(m_pg, Scrub::scrub_prio_t::high_priority);
```

2.封装成PGScrub任务，要转什么状态，就用对应的实现，比如这里是PGScrubResched

```
void OSDService::queue_for_scrub_resched(PG *pg, Scrub::scrub_prio_t with_priority) {
    queue_scrub_event_msg<PGScrubResched>(pg, with_priority);
}
```

3.进一步封装成OPItem，并加入OSD的任务队列

```
enqueue_back(OpSchedulerItem(
        unique_ptr<OpSchedulerItem::OpQueueable>(msg), get_scrub_cost(),
        pg->scrub_requeue_priority(with_priority), ceph_clock_now(), 0, epoch));
```

4.从任务队列取出后，调用run，比如这里是PGScrubResched实现

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

5.这个scrub_send_scrub_xxx 也不知道用的什么抽象的技术，反正是看不懂的

```
1 void scrub_send_scrub_resched(epoch_t queued, ThreadPool::TPHandle& handle)
2 {
3     forward_scrub_event(&ScrubPgIF::send_scrub_resched, queued, "InternalSchedScrub");
4 }
5
6 void PG::forward_scrub_event(ScrubAPI fn, epoch_t epoch_queued, std::string_view desc) {
7     ceph_assert(m_scrubber);
8     if (is_active()) {
9         ((*m_scrubber).*fn)(epoch_queued);
10     } else {
11         // pg might be in the process of being deleted
12     }
13 }
```

6.实际调用的是send_scrub_xxx函数（看上一段代码的第三行）

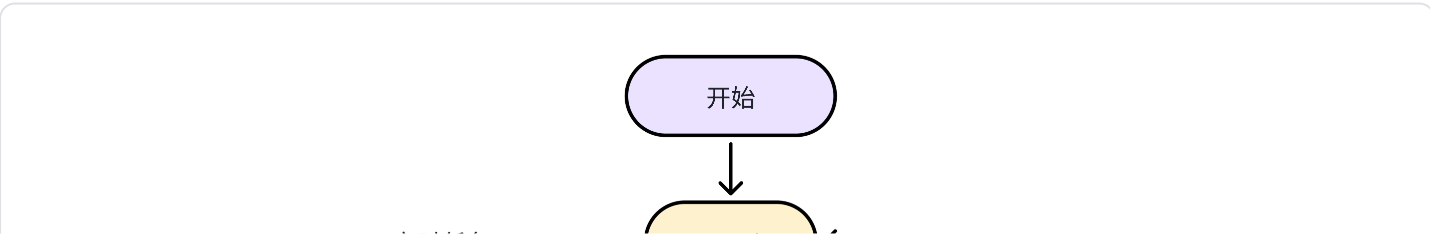
```
void PgScrubber::send_scrub_resched(epoch_t epoch_queued) {
    if (is_message_relevant(epoch_queued)) {
// 抛出InternalSchedScrub事件
        m_fsm->process_event(InternalSchedScrub{});
    }
}
```

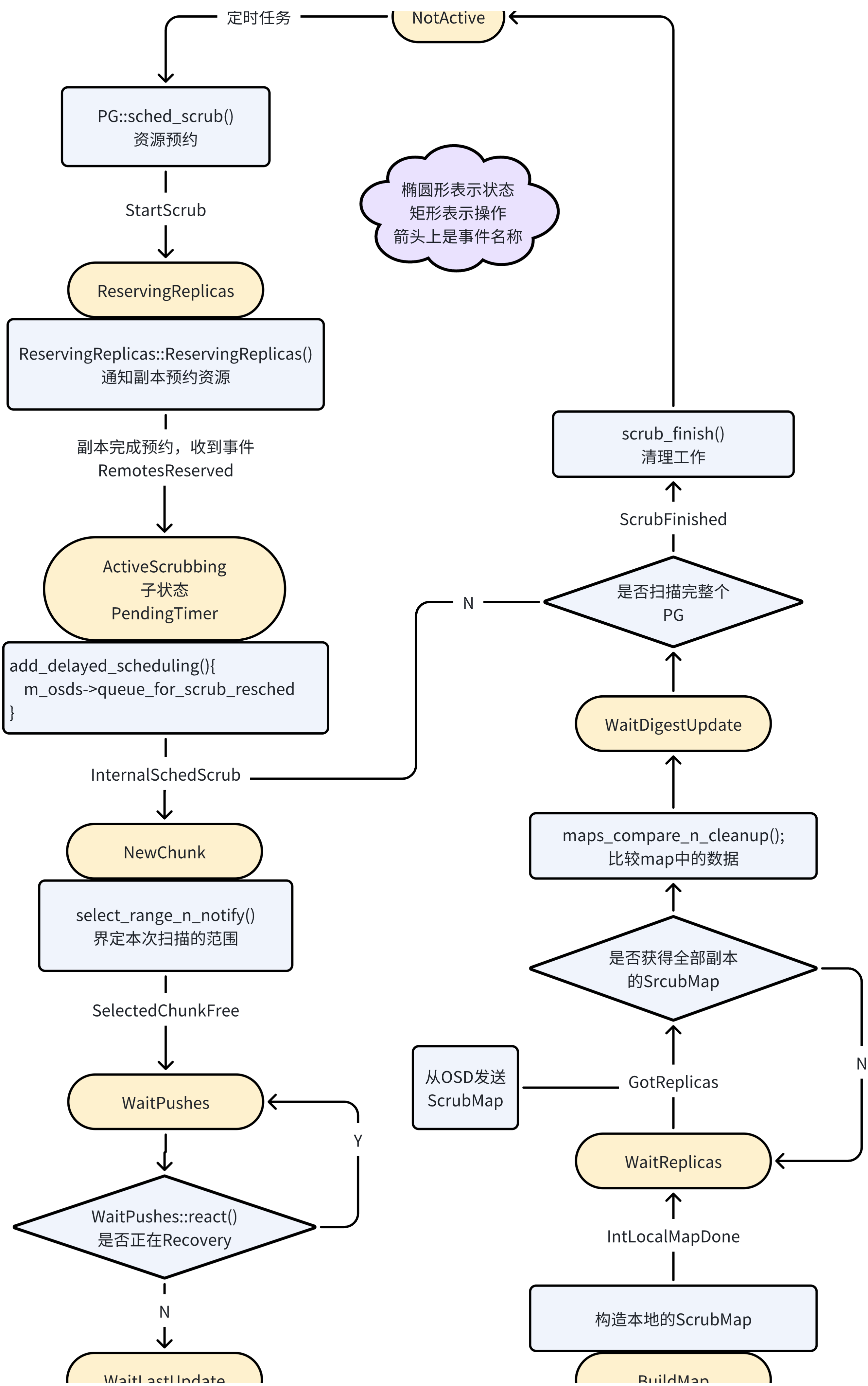
7.最后投递的事件被当前状态接收，完成状态转换

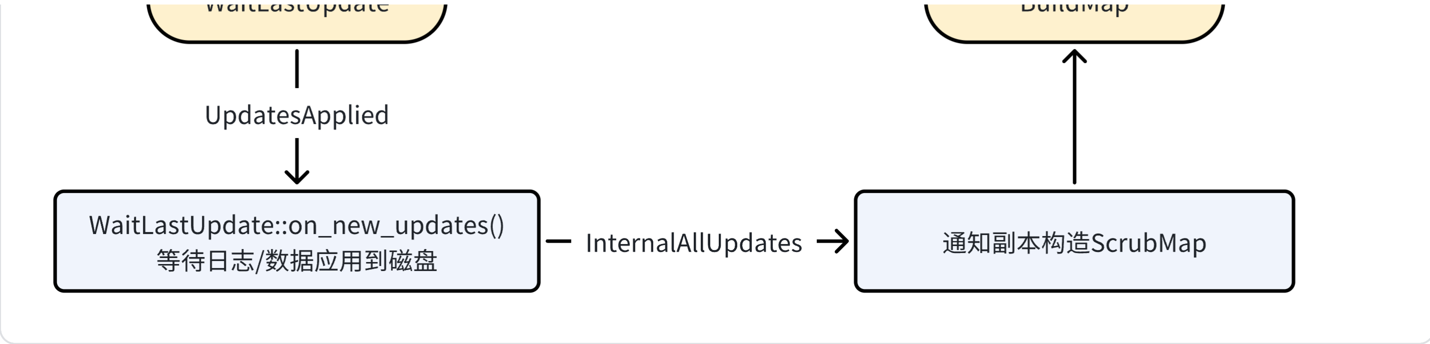
```
struct PendingTimer : sc::state<PendingTimer, ActiveScrubbing>, NamedSimply {
// 当收到InternalSchedScrub事件时，自动转到状态NewChunk
  using reactions = mpl::list<sc::transition<InternalSchedScrub, NewChunk>>;
};
```

> 说实话是理解不了的，为什么切换状态要涉及这么多层，还要重新获得线程资源，为了方便被随时抢占吗。。。

总体流程




定时任务
NotActive
PG::sched_scrub()
资源预约
椭圆形表示状态
矩形表示操作
箭头上是事件名称
StartScrub
ReservingReplicas
ReservingReplicas::ReservingReplicas()
通知副本预约资源
副本完成预约，收到事件
RemotesReserved
ActiveScrubbing
子状态
PendingTimer
add_delayed_scheduling(){
m_osds->queue_for_scrub_resched
}
InternalSchedScrub
NewChunk
select_range_n_notify()
界定本次扫描的范围
SelectedChunkFree
WaitPushes
Y
WaitPushes::react()
是否正在Recovery
N
scrub_finish()
清理工作
ScrubFinished
N
是否扫描完整个
PG
WaitDigestUpdate
maps_compare_n_cleanup();
比较map中的数据
是否获得全部副本
的SrcubMap
N
从OSD发送
ScrubMap
GotReplicas
WaitReplicas
IntLocalMapDone
构造本地的ScrubMap



资源预留

```
void PGScrub::run() ==> 抛出StartScrub事件

sc::result NotActive::react(const StartScrub&)
{
  DECLARE_LOCALS;
  scrbr->set_scrub_begin_time();
  return transit<ReservingReplicas>();
}

ReservingReplicas::ReservingReplicas(my_context ctx) : my_base(ctx)
    , NamedSimply(context<ScrubMachine>().m_scrbr, "ReservingReplicas")
{
  DECLARE_LOCALS;
  // 预留资源副本，防止 OSD 在资源预留过程中启动另一个扫描操作
  scrbr->set_reserving_now(); // a_pg_is_reserving = true;

  // 通知副本预留资源
  scrbr->reserve_replicas();
}

副本预约完成后，收到RemotesReserved，切换状态
 sc::transition<RemotesReserved, ActiveScrubbing>
```

ActiveScrubbing

```
ActiveScrubbing::ActiveScrubbing(my_context ctx)
    : my_base(ctx)
    , NamedSimply(context<ScrubMachine>().m_scrbr, "ActiveScrubbing")
{
  DECLARE_LOCALS;   // 'scrbr' & 'pg_id' aliases
  scrbr->on_init();
}

```

```
void PgScrubber::on_init() {
    // 断言确保没有其他活跃的扫描操作正在进行
    ceph_assert(!is_scrub_active());

    // 重置 PG 中已扫描对象的计数
    m_pg->reset_objects_scrubbed();

    // 重置预处理数据
    preemption_data.reset();

    // 设置扫描操作的起始时间间隔
    m_interval_start = m_pg->get_history().same_interval_since;

    // 创建负责执行实际扫描操作的后端对象
    m_be = std::make_unique<ScrubBackend>(
        *this,  // 当前 PgScrubber 实例
        *m_pg,  // PG 实例
        m_pg_whoami,  // PG 的标识符
        m_is_repair,  // 是否执行修复操作
        m_is_deep ? scrub_level_t::deep : scrub_level_t::shallow,  // 扫描级别
        m_pg->get_actingset()  // PG 的活动副本集合
    );

    // 创建一个新的存储对象
    {
        ObjectStore::Transaction t;
        cleanup_store(&t);  // 清理现有存储
        m_store.reset(
            Scrub::Store::create(m_pg->osd->store, &t, m_pg->info.pgid, m_pg->coll)  // 创建新的存储实例
        );
        m_pg->osd->store->queue_transaction(m_pg->ch, std::move(t), nullptr);  // 将事务加入队列
    }

    // 设置扫描操作的起始对象
    m_start = m_pg->info.pgid.pgid.get_hobj_start();

    // 将扫描操作的状态设置为活跃
    m_active = true;

    // 增加会话计数器
    ++m_sessions_counter;

    // 将扫描操作的状态和会话计数器发布到 OSD
    m_pg->publish_stats_to_osd();
}
```

然后，它有一个默认子状态，决定是否立即开始Scrub

```
struct ActiveScrubbing
    : sc::state<ActiveScrubbing, ScrubMachine, PendingTimer>, NamedSimply {}

PendingTimer::PendingTimer(my_context ctx)
    : my_base(ctx)
    , NamedSimply(context<ScrubMachine>().m_scrbr, "Act/PendingTimer")
{
  scrbr->add_delayed_scheduling();
}

立即或推迟后调用:
m_osds->queue_for_scrub_resched(m_pg, Scrub::scrub_prio_t::high_priority);
==> 切换到NewChunk状态
```

NewChunk

```
NewChunk::NewChunk(my_context ctx)
    : my_base(ctx), NamedSimply(context<ScrubMachine>().m_scrbr, "Act/NewChunk") {

    DECLARE_LOCALS; // 'scrbr' & 'pg_id' aliases
    scrbr->get_preemptor().adjust_parameters();
    scrbr->select_range_n_notify();
}


void PgScrubber::select_range_n_notify() {
// 1 范围界定
    if (select_range()) {
        // 抛出SelectedChunkFree事件，转换到WaitPushes状态
        m_osds->queue_scrub_chunk_free(m_pg, Scrub::scrub_prio_t::low_priority);
    } else {
        m_osds->queue_scrub_chunk_busy(m_pg, Scrub::scrub_prio_t::low_priority);
    }
}

sc::result NewChunk::react(const SelectedChunkFree &) {
    DECLARE_LOCALS;
```

```
    scrbr->set_subset_last_update(scrbr->search_log_for_updates());
    return transit<WaitPushes>();
}
```

WaitPushes

```
WaitPushes::WaitPushes(my_context ctx)
    : my_base(ctx), NamedSimply(context<ScrubMachine>().m_scrbr, "Act/WaitPushes") {
    post_event(ActivePushesUpd{});
}

sc::result WaitPushes::react(const ActivePushesUpd &) {
    DECLARE_LOCALS;
    if (!scrbr->pending_active_pushes()) {
// 表示没有正在进行修复，跳转到下一状态
        return transit<WaitLastUpdate>();
    }
// 如果PG正在修复，忽略这次事件，保持WaitPushes
    return discard_event();
}

```

WaitLastUpdate

```
struct WaitLastUpdate : sc::state<WaitLastUpdate, ActiveScrubbing>,
            NamedSimply {
...
  using reactions =
    mpl::list<sc::in_state_reaction<UpdatesApplied, WaitLastUpdate,
                &WaitLastUpdate::on_new_updates>>;
...
};

WaitLastUpdate::WaitLastUpdate(my_context ctx)
    : my_base(ctx), NamedSimply(context<ScrubMachine>().m_scrbr, "Act/WaitLastUpdate") {
    post_event(UpdatesApplied{});
}

void WaitLastUpdate::on_new_updates(const UpdatesApplied &) {
    DECLARE_LOCALS;
```

```

// 等待日志应用完毕 return last_applied >= m_subset_last_update;
    if (scrbr->has_pg_marked_new_updates()) {
        post_event(InternalAllUpdates{});
    } else {
        // will be requeued by op_applied
    }
}

sc::result WaitLastUpdate::react(const InternalAllUpdates &) {
    DECLARE_LOCALS;
// 通知副本准备好map
    scrbr->get_replicas_maps(scrbr->get_preemptor().is_preemptable());
// 状态转换
    return transit<BuildMap>();
}
```

续

```
void PgScrubber::get_replicas_maps(bool replica_can_preempt) {

    m_primary_scrubmap_pos.reset();

    for (const auto &i : m_pg->get_actingset()) {

        if (i == m_pg_whoami)
            continue;
        m_maps_status.mark_replica_map_request(i);
        _request_scrub_map(i,
                           m_subset_last_update,
                           m_start,
                           m_end,
                           m_is_deep,
                           replica_can_preempt);
    }
}

void PgScrubber::_request_scrub_map(pg_shard_t replica,
                                     eversion_t version,
                                     hobject_t start,
                                     hobject_t end,
                                     bool deep,
                                     bool allow_preemption) {
    ceph_assert(replica != m_pg_whoami);

```

```
    auto repscrubop = new MOSDRepScrub(spg_t(m_pg->info.pgid, replica.shard),
            version, get_osdmap_epoch(), m_pg->get_last_peering_reset(),
            start, end,  deep, allow_preemption, m_flags.priority,
                                       m_pg->ops_blocked_by_scrub());
// 后续见从副本的流程
    m_osds->send_message_osd_cluster(replica.osd, repscrubop, get_osdmap_epoch());
}

```

BuildMap

```
BuildMap::BuildMap(my_context ctx)
    : my_base(ctx), NamedSimply(context<ScrubMachine>().m_scrbr, "Act/BuildMap") {
    DECLARE_LOCALS;

    if (scrbr->get_preemptor().was_preempted()) {
        // we were preempted, either directly or by a replica
        scrbr->mark_local_map_ready();
        post_event(IntBmPreempted{});
    } else {
// 构造本地的ScrubMap
        auto ret = scrbr->build_primary_map_chunk();
        if (ret == -EINPROGRESS) {
            // must wait for the backend to finish. No specific event provided.
            // build_primary_map_chunk() has already requeued us.
        } else if (ret < 0) {
            post_event(InternalError{});
        } else {
            // the local map was created
            post_event(IntLocalMapDone{});
        }
    }
}

sc::result BuildMap::react(const IntLocalMapDone &) {
    DECLARE_LOCALS;
    scrbr->mark_local_map_ready();
    return transit<WaitReplicas>();
}

```

续

```
int PgScrubber::build_primary_map_chunk() {
    epoch_t map_building_since = m_pg->get_osdmap_epoch();
    // 见文末
    auto ret = build_scrub_map_chunk(m_be->get_primary_scrubmap(),
        m_primary_scrubmap_pos, m_start, m_end, m_is_deep);

    if (ret == -EINPROGRESS) {
        // 重新安排另一轮请求后端收集清理数据
        // 抛出InternalSchedScrub事件
        m_osds->queue_for_scrub_resched(m_pg,
Scrub::scrub_prio_t::low_priority);
    }
    return ret;
}


```

从副本准备和发送ScrubMap

主OSD在WaitLastUp阶段发送了一个MOSDRepScrub信息给从OSD，用于获取对象的校验信息。

```
void PrimaryLogPG::do_request(
  OpRequestRef& op,
  ThreadPool::TPHandle &handle)
{
  case MSG_OSD_REP_SCRUB:
    replica_scrub(op, handle);
    break;
}

void PG::replica_scrub(OpRequestRef op, ThreadPool::TPHandle &handle) {
    ceph_assert(m_scrubber);
    m_scrubber->replica_scrub_op(op);
}
```

```
void PgScrubber::replica_scrub_op(OpRequestRef op) {
    op->mark_started();
    auto msg = op->get_req<MOSDRepScrub>();
```

```

// 这是过时的请求
    if (msg->map_epoch < m_pg->info.history.same_interval_since) {
        return;
    }

// 意思是当前的清理操作尚未完成时，就收到了新的请求，重置scrub状态
    if (is_queued_or_active()) {
        scrub_clear_state(); // 转为NotActive
    }

    // 确保状态机处于非活动状态
    m_fsm->assert_not_active();

    replica_scrubmap = ScrubMap{};
    replica_scrubmap_pos = ScrubMapBuilder{};

    m_replica_min_epoch = msg->min_epoch;
    m_start = msg->start;
    m_end = msg->end;
    m_max_end = msg->end;
    m_is_deep = msg->deep;
    m_interval_start = m_pg->info.history.same_interval_since;
    m_replica_request_priority = msg->high_priority
                                      ? Scrub::scrub_prio_t::high_priority
                                      : Scrub::scrub_prio_t::low_priority;
    m_flags.priority = msg->priority ? msg->priority : m_pg->get_scrub_priority();

    preemption_data.reset();
    preemption_data.force_preemptability(msg->allow_preemption);

    replica_scrubmap_pos.reset();

    set_queued_or_active();

// 最终的行为是调用 send_start_replica
    m_osds->queue_for_rep_scrub(m_pg,
                                m_replica_request_priority,
                                m_flags.priority,
                                m_current_token);
}


void PgScrubber::send_start_replica(epoch_t epoch_queued,
                                    Scrub::act_token_t token) {
    if (is_primary()) {
```

```
        return;
    }

// 检查消息是否过时 return epoch_to_verify >= m_pg->get_same_interval_since();
    if (check_interval(epoch_queued) && is_token_current(token)) {
        if (pending_active_pushes())
            m_fsm->process_event(StartReplica{});
        else
            m_fsm->process_event(StartReplicaNoWait{});
    }
}


struct NotActive : sc::state<NotActive, ScrubMachine>, NamedSimply {
  using reactions = mpl::list<
              sc::transition<StartReplica, ReplicaWaitUpdates>,
              sc::transition<StartReplicaNoWait, ActiveReplica>>;
};

ReplicaWaitUpdates 状态会等待 pushes操作结束，然后转到 ActiveReplica
```

```
ActiveReplica::ActiveReplica(my_context ctx)
    : my_base(ctx)
    , NamedSimply(context<ScrubMachine>().m_scrbr, "ActiveReplica")
{
  DECLARE_LOCALS;
  scrbr->on_replica_init();
  post_event(SchedReplica{});
}

sc::result ActiveReplica::react(const SchedReplica&)
{
  DECLARE_LOCALS;
// 被抢占
  if (scrbr->get_preemptor().was_preempted()) {
    scrbr->send_preempted_replica();
    scrbr->replica_handling_done();
    return transit<NotActive>();
  }
// 构造ScrubMap，并发送给主OSD
  auto ret_init = scrbr->build_replica_map_chunk();
  if (ret_init != -EINPROGRESS) {
    return transit<NotActive>();
  }

```

```
  return discard_event();
}



int PgScrubber::build_replica_map_chunk() {
    ceph_assert(m_be);

// 构造ScrubMap，同主OSD流程
    auto ret = build_scrub_map_chunk(replica_scrubmap,
                                     replica_scrubmap_pos,
                                     m_start,
                                     m_end,
                                     m_is_deep);


    auto required_fixes = m_be->replica_clean_meta(replica_scrubmap,
                                                    m_end.is_max(),
                                                    m_start,
                                                    get_snap_mapper_accessor());
    auto reply = prep_replica_map_msg(PreemptionNoted::no_preemption);
    replica_handling_done();
    // 发给主OSD
    // 消息类型是MOSDRepScrubMap
    send_replica_map(reply);

    return ret;
}
```

WaitReplicas

回到主OSD，收到从OSD的消息后会抛出GotReplicas事件

```
WaitReplicas::WaitReplicas(my_context ctx)
    : my_base(ctx), NamedSimply(context<ScrubMachine>().m_scrbr,
"Act/WaitReplicas") {
    post_event(GotReplicas{});
}

sc::result WaitReplicas::react(const GotReplicas &) {
    DECLARE_LOCALS;

    if (!all_maps_already_called && scrbr->are_all_maps_available()) {

        all_maps_already_called = true;
```

```

        // were we preempted?
        if (scrbr->get_preemptor().disable_and_test()) {
            return transit<PendingTimer>();
        } else {
// 比较map数据
            scrbr->maps_compare_n_cleanup();
            return transit<WaitDigestUpdate>(); // 切状态
        }
    } else {
        return discard_event();
    }
}
```

WaitDigestUpdate

```
WaitDigestUpdate::WaitDigestUpdate(my_context ctx)
    : my_base(ctx)
    , NamedSimply(context<ScrubMachine>().m_scrbr, "Act/WaitDigestUpdate")
{
  DECLARE_LOCALS;
  post_event(DigestUpdate{});
}

sc::result WaitDigestUpdate::react(const DigestUpdate&)
{
  DECLARE_LOCALS;
  scrbr->on_digest_updates();
  return discard_event();
}

void PgScrubber::on_digest_updates() {
    if (num_digest_updates_pending > 0) {
        return;
    }

    if (m_end.is_max()) { // 扫描完毕
// 抛出ScrubFinished
        m_osds->queue_scrub_is_finished(m_pg);
    } else {
        // go get a new chunk (via "requeue")
        preemption_data.reset();
// 抛出NewChunk，继续扫描当前PG中剩下的对象
        m_osds->queue_scrub_next_chunk(m_pg, m_pg->is_scrub_blocking_ops());
    }
```

```
}

sc::result WaitDigestUpdate::react(const ScrubFinished&)
{
  DECLARE_LOCALS;

  scrbr->set_scrub_duration();
  scrbr->scrub_finish();
  return transit<NotActive>();
}

```

scrub_finish

这个函数主要做清理工作，重置清理计划，更新修复状态等，最重要的是如果发现错误，会启动数据修复。

```
void PgScrubber::scrub_finish() {
    if (has_error) {
        m_pg->queue_peering_event(PGPeeringEventRef(
            std::make_shared<PGPeeringEvent>(get_osdmap_epoch(),
                                             get_osdmap_epoch(),
                                             PeeringState::DoRecovery())));
    }
}
```

补充

更具体的细节，比如怎么构建ScrubMap，怎么根据ScrubMap做检查。

1 build_scrub_map_chunk

构建指定范围内，所有对象的校验信息，并保存在SrcubMap中

```

int PgScrubber::build_scrub_map_chunk(ScrubMap &map,
                                      ScrubMapBuilder &pos,
                                      hobject_t start,
                                      hobject_t end,
                                      bool deep) {
    // start
    while (pos.empty()) {

```

```
        pos.deep = deep;
        map.valid_through = m_pg->info.last_update;

// 1.1 获取指定范围内的对象
        vector<ghobject_t> rollback_obs;
        pos.ret = m_pg->get_pgbackend()->objects_list_range(start,
                                                            end,
                                                            &pos.ls,
                                                            &rollback_obs);
        if (pos.ret < 0) {
            return pos.ret;
        }
        if (pos.ls.empty()) {
            break;
        }
        m_pg->_scan_rollback_obs(rollback_obs);
        pos.pos = 0;
        return -EINPROGRESS;
    }

// 1.2 扫描对象，并构建map
    while (!pos.done()) {
        int r = m_pg->get_pgbackend()->be_scan_list(map, pos);
        if (r == -EINPROGRESS) {
            return r;
        }
    }

    // finish
    ceph_assert(pos.done());
    repair_oinfo_oid(map);

    return 0;
}
```

1.1 objects_list_range

```
int PGBackend::objects_list_range(
    const hobject_t &start,
    const hobject_t &end,
    vector<hobject_t> *ls,
    vector<ghobject_t> *gen_obs) {
    ceph_assert(ls);
    vector<ghobject_t> objects;
    int r;
```

```
    if (HAVE_FEATURE(parent->min_upacting_features(),
                     OSD_FIXED_COLLECTION_LIST)) {
// 这里是用BlueStore的函数，寻找指定范围中的对象
        r = store->collection_list(
            ch,
            ghobject_t(start, ghobject_t::NO_GEN, get_parent()->whoami_shard().shard),
            ghobject_t(end, ghobject_t::NO_GEN, get_parent()->whoami_shard().shard),
            INT_MAX,
            &objects,
            NULL);
    } else {
        // 旧的接口
        r = store->collection_list_legacy(
            ch,
            ghobject_t(start, ghobject_t::NO_GEN, get_parent()->whoami_shard().shard),
            ghobject_t(end, ghobject_t::NO_GEN, get_parent()->whoami_shard().shard),
            INT_MAX,
            &objects,
            NULL);
    }
    ls->reserve(objects.size());
    for (vector<ghobject_t>::iterator i = objects.begin();
         i != objects.end();
         ++i) {
         // 如果是临时对象，跳过
        if (i->is_pgmeta() || i->hobj.is_temp()) {
            continue;
        }
        // 没有生成号的加入ls，有生成号的加入gen，这个gen_obs在调用方是回滚对象
        if (i->is_no_gen()) {
            ls->push_back(i->hobj);
        } else if (gen_obs) {
            gen_obs->push_back(*i);
        }
    }
    return r;
}
```

```
void PG::_scan_rollback_obs(const vector<ghobject_t> &rollback_obs) {
```

```
    ObjectStore::Transaction t;
    eversion_t trimmed_to = recovery_state.get_last_rollback_info_trimmed_to_applied();
    for (vector<ghobject_t>::const_iterator i = rollback_obs.begin();
         i != rollback_obs.end();
         ++i) {
        if (i->generation < trimmed_to.version) {
            t.remove(coll, *i);
        }
    }
    if (!t.empty()) {
        osd->store->queue_transaction(ch, std::move(t), NULL);
    }
}
```

1.2 be_scan_list

```
int PGBackend::be_scan_list(
    ScrubMap &map,
    ScrubMapBuilder &pos) {
    ceph_assert(!pos.done());
    ceph_assert(pos.pos < pos.ls.size());
    hobject_t &poid  = pos.ls[pos.pos];

// 获取对象的状态信息
    struct stat st;
    int r = store->stat(ch,
        ghobject_t(poid, ghobject_t::NO_GEN, get_parent()->whoami_shard().shard),
        &st, true);

    if (r == 0) {
        ScrubMap::object &o = map.objects[poid];
        o.size = st.st_size;
        ceph_assert(!o.negative);
        // 获取对象的属性，用于浅扫描
        store->getattrs(ch,
            ghobject_t(poid, ghobject_t::NO_GEN, get_parent()->whoami_shard().shard),
            o.attrs);
        // 如果需要深扫描
        if (pos.deep) {
            r = be_deep_scrub(poid, map, pos, o);
        }
    }
```

```
    pos.next_object();
    return 0;
}
```

这个函数一共四步：读数据，读header，读omap，计算digest（校验信息）

```
int ReplicatedBackend::be_deep_scrub(
    const hobject_t &poid,
    ScrubMap &map,
    ScrubMapBuilder &pos,
    ScrubMap::object &o) {
    int r;
    uint32_t fadvise_flags = CEPH_OSD_OP_FLAG_FADVISE_SEQUENTIAL |
                             CEPH_OSD_OP_FLAG_FADVISE_DONTNEED |
                             CEPH_OSD_OP_FLAG_BYPASS_CLEAN_CACHE;

    utime_t sleeptime;
    sleeptime.set_from_double(cct->_conf->osd_debug_deep_scrub_sleep);
    if (sleeptime != utime_t()) {
        sleeptime.sleep();
    }

    ceph_assert(poid == pos.ls[pos.pos]);
    if (!pos.data_done()) {
        if (pos.data_pos == 0) {
            pos.data_hash = bufferhash(-1);
        }

        const uint64_t stride = cct->_conf->osd_deep_scrub_stride;

        bufferlist bl;
// 1 一次读取stride长度的数据，如果读取除了完整的stride，说明可能还有需要读取的块
// 返回-EINPROGRESS，同时上层会再次调用到这里，循环直到读取完
        r = store->read(
            ch,
            ghobject_t(
                poid, ghobject_t::NO_GEN, get_parent()->whoami_shard().shard),
            pos.data_pos,
            stride, bl,
            fadvise_flags);
        if (r < 0) {
            o.read_error = true;
            return 0;
        }
        if (r > 0) {
```

```
            pos.data_hash << bl;
        }
        pos.data_pos += r;
        if (static_cast<uint64_t>(r) == stride) {
            return -EINPROGRESS;
        }
        // done with bytes
        pos.data_pos = -1;
        o.digest = pos.data_hash.digest();
        o.digest_present = true;
    }

// 2 omap header
    if (pos.omap_pos.empty()) {
        pos.omap_hash = bufferhash(-1);

        bufferlist hdrbl;
        r = store->omap_get_header(
            ch,
            ghobject_t(
                poid, ghobject_t::NO_GEN, get_parent()->whoami_shard().shard),
            &hdrbl, true);
        if (r == -EIO) {
            o.read_error = true;
            return 0;
        }
        if (r == 0 && hdrbl.length()) {
            bool encoded = false;
            pos.omap_hash << hdrbl;
        }
    }

// 3 omap
    ObjectMap::ObjectMapIterator iter = store->get_omap_iterator(
        ch,
        ghobject_t(
            poid, ghobject_t::NO_GEN, get_parent()->whoami_shard().shard));
    ceph_assert(iter);
    if (pos.omap_pos.length()) {
        iter->lower_bound(pos.omap_pos);
    } else {
        iter->seek_to_first();
    }
    int max = g_conf()->osd_deep_scrub_keys;
    while (iter->status() == 0 && iter->valid()) {
        pos.omap_bytes += iter->value().length();
        ++pos.omap_keys;
```

```
        --max;
        // fixme: we can do this more efficiently.
        bufferlist bl;
        encode(iter->key(), bl);
        encode(iter->value(), bl);
        pos.omap_hash << bl;

        iter->next();

        if (iter->valid() && max == 0) {
            pos.omap_pos = iter->key();
            return -EINPROGRESS;
        }
        if (iter->status() < 0) {

            o.read_error = true;
            return 0;
        }
    }

    if (pos.omap_keys > cct->_conf->osd_deep_scrub_large_omap_object_key_threshold ||
        pos.omap_bytes > cct->_conf->osd_deep_scrub_large_omap_object_value_sum_threshold) {

        o.large_omap_object_found = true;
        o.large_omap_object_key_count = pos.omap_keys;
        o.large_omap_object_value_size = pos.omap_bytes;
        map.has_large_omap_object_errors = true;
    }
// 4 计算digest
    o.omap_digest = pos.omap_hash.digest();
    o.omap_digest_present = true;

    // Sum up omap usage
    if (pos.omap_keys > 0 || pos.omap_bytes > 0) {
        map.has_omap_keys = true;
        o.object_omap_bytes = pos.omap_bytes;
        o.object_omap_keys = pos.omap_keys;
    }

    // done!
    return 0;
}
```

2 maps_compare_n_cleanup

在上文中， 等到所有的副本都计算好了digest，就调用这个函数进行比较

```
void PgScrubber::maps_compare_n_cleanup() {
    // 更新已扫描对象的数量
    m_pg->add_objects_scrubbed_count(m_be->get_primary_scrubmap().objects.size());

    // 比较map并获取需要修复的对象
    auto required_fixes = m_be->scrub_compare_maps(m_end.is_max(), get_snap_mapper_accessor());

    if (!required_fixes.inconsistent_objs.empty()) { // 有不一致的对象
        if (state_test(PG_STATE_REPAIR)) {
            // 如果PG正在修复中，丢弃
        } else {
            // 如果不是修复状态，执行修复操作，处理不一致的对象
            persist_scrub_results(std::move(required_fixes.inconsistent_objs));
        }
    }

    // 应用快照映射器的修复
    apply_snap_mapper_fixes(required_fixes.snap_fix_list);

    auto chunk_err_counts = m_be->get_error_counts();
    m_shallow_errors += chunk_err_counts.shallow_errors;
    m_deep_errors += chunk_err_counts.deep_errors;

// 更新起始位置，为下一轮清理做准备
    m_start = m_end;

    // 运行回调，可能用于通知其他组件清理完成
    run_callbacks();

    requeue_waiting();
}
```

```
objs_fix_list_t ScrubBackend::scrub_compare_maps(
    bool max_reached,
    SnapMapReaderI &snaps_getter) {

    ceph_assert(m_scrubber.is_primary());

```

```
// 1 构建权威map
// insert(this_chunk->received_maps[m_pg_whoami])
    m_cleaned_meta_map.insert(my_map());
// 1.1 合并所有的map
// 将this_chunk->received_maps中的对象ID全添加到authoritative_set
    merge_to_authoritative_set();
// 1.2 收集统计信息
    omap_checks();
// 1.3 更新权威map，通过各个map间的共同信息，计算出真正的权威map
    update_authoritative();
// 5 返回一个包含不一致对象的列表
    auto for_meta_scrub = clean_meta_map(m_cleaned_meta_map, max_reached);
// 6 检查快照元数据
    scrub_snapshot_metadata(for_meta_scrub);
// 7 返回一个包含不一致对象列表，和快照扫描结果的对象
    return objs_fix_list_t{std::move(this_chunk->m_inconsistent_objs),
                           scan_snaps(for_meta_scrub, snaps_getter)};
}
```

计算权威对象的过程很乱，但是遵循以下原则：

1. 没有状态错误，静默数据错误，OI属性正确，对象大小正确等
2. 如果各项指标正确，优先选择校验信息更多的副本
3. 如果存在满足上述条件的多个副本，优先选择主OSD对应的副本，否则随机选择