快照和克隆

基本概念

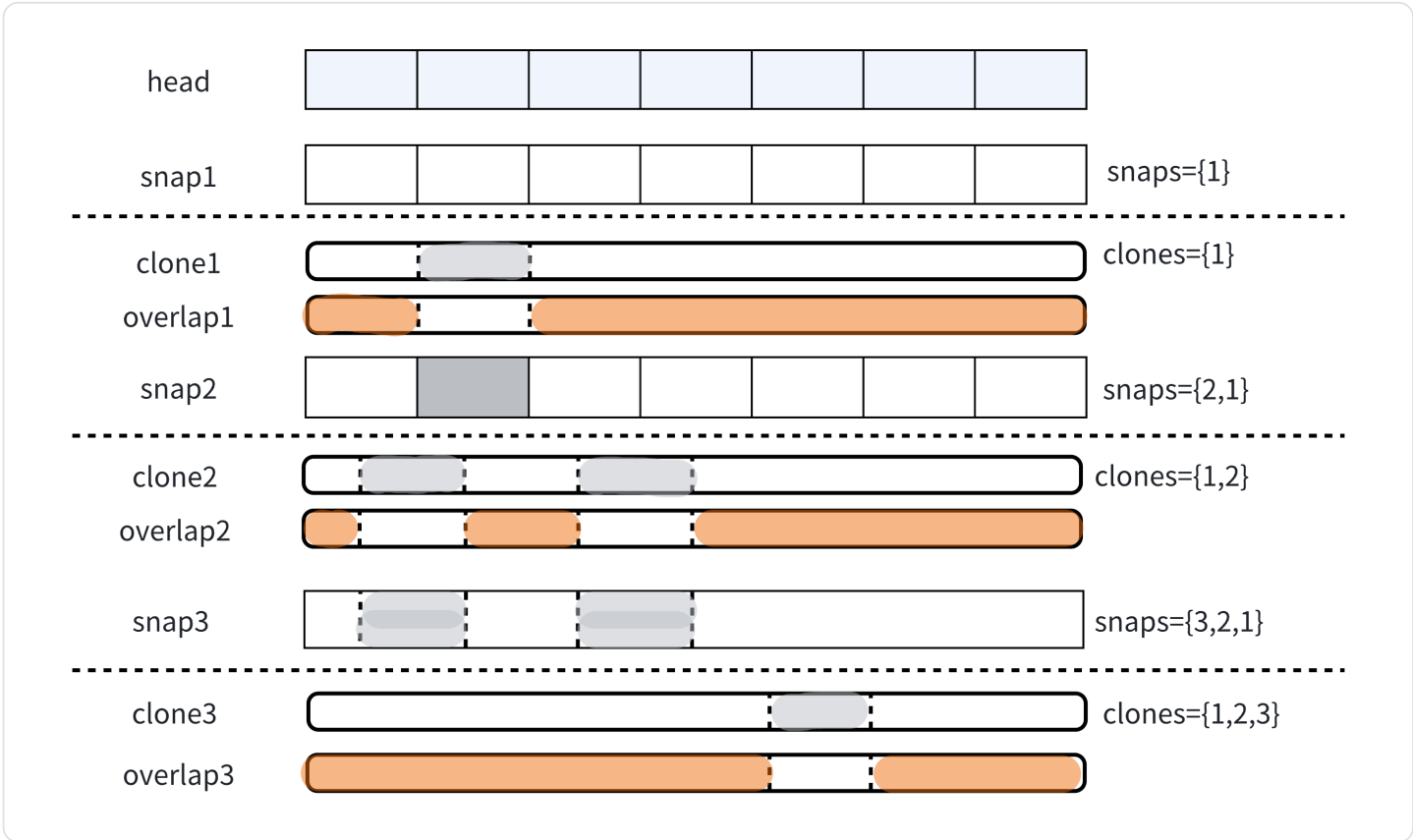
快照和克隆

快照是一个RBD(Rados Block Device)在某一时刻的全部可读镜像，克隆是在某一时刻全部数据的可写镜像，两者都是某一时间点的镜像，区别在于快照只能读，克隆可写。

快照结构

一个常见的快照对象如下图：其中head是原始对象，对原始对象做快照，此时只能读，如果对快照进行写操作，会自动创建一个clone对象用来写，此时会采用COW(Copy On Write)机制，除了数据本身带的元数据外，还有overlap用来记录与上一个对象的重叠区间(未更改区间)。

图中，灰色表示被修改过的数据区间，橙色是overlap记录的区间。



> 有些书上描述head对象和其它快照对象，也会像这里一样被分为了多个区间，但我觉得根本不存在这个区间的概念，比如clone2，可以看到它随意修改了数据，并没有受到所谓区间的限制，而overlap记录“区间”的方式仅仅是offset+length。

上面画的都是数据区间大小不作改变的情况，可以很清晰的看出overlap是怎么配合快照/克隆做读写操作的：已修改的数据通过克隆/快照本身读取，未修改的数据通过overlap的指示向上查找实际存数

据的对象。

假设一个场景：

snap1 的区间是[0, 100]，snap2在snap1的基础上，在中间[50,80]插入了一条数据，使得snap2的区间为[0, 130]。此时对于2来说，未修改的区间是：[0,50]，[80, 130]，如果overlap存(0, 50)，(80, 50)，虽然一眼能看出被修改的区域是[50, 80]，但是向上从snap1查找数据时，怎么知道(80,50)对应snap1的哪部分区间呢?

SnapSet

用于保存OSD端与快照相关的信息

```
struct SnapSet {
    snapid_t seq;                                                    // 最新的快照序号
    std::vector<snapid_t> snaps;                                     // 快照序号列表，降序
    std::vector<snapid_t> clones;                                    // clone对象序号列表，升序
    std::map<snapid_t, interval_set<uint64_t>> clone_overlap; // 和上次对象之间的overlap的部分
    std::map<snapid_t, uint64_t> clone_size;                         // clone对象的size
    std::map<snapid_t, std::vector<snapid_t>> clone_snaps;    // 快照和clone对象列表的映射
}

template<typename T, template<typename, typename, typename ...> class C = std::map>
class interval_set {
 public:
  using Map = C<T, T>;
  using value_type = typename Map::value_type;
  using offset_type = T;
  using length_type = T;
  using reference = value_type&;
  using const_reference = const value_type&;
  using size_type = typename Map::size_type;
}
```

客户端（RBD端）相应的数据结构

```
struct SnapContext {
    snapid_t seq;                // 最新的快照序号
    vector<snapid_t> snaps;      // 降序排列的快照序号
}
```

```
5
6  struct OpContext {
7      const ObjectState *obs;          // Old objectstate
8      const SnapSet *snapset;          //旧的Snapset，OSD服务端保存的快照信息
9
10     ObjectState new_obs;             //resulting ObjectState
11     SnapSet new_snapset;             //新的SnapSet
12
13     SnapContext snapc;              //写操作带的，也就是客户端的SnapContext信息
14
15 };
```

快照的写操作

当对一个image做了一次快照后，该image写入数据时，由于快照的存在需要启动copy-on-write(cow)机制。下面介绍cow机制的具体实现。

客户端的每次写操作，消息中都必须带数据结构SnapContext信息，它包含了客户端认为的最新快照序号seq，以及该对象的所有快照序号snaps的列表。在OSD端，对象的Snap相关信息保存在SnapSet数据结构中，当有写操作发生时，处理过程按照如下规则进行。

规则1 客户端seq<服务端seq

如果写操作所带的SnapContext的seq值小于SnapSet的seq值，也就是客户端最新的快照序号小于OSD端保存的最新的快照序号，那么直接返回-EOLDSNAP错误。

Ceph客户端始终保持最新的快照序号。如果客户端不是最新的快照序号，可能的情况是：在多个客户端的情形下，其他客户端有可能创建了快照，本客户端有可能没有获取到最新的快照序号。

Ceph有一套Watcher回调通知机制来实现快照序号的更新。如果其他客户端对一个卷做了快照，就会产生一个最新的快照序号。OSD端接收到最新快照序号变化后，通知相应的连接客户端更新最新的快照序号。如果有客户端没有及时更新，也没有太大的问题，OSD端会返回客户端-EOLDSNAP，客户端会主动更新为最新的快照序号，重新发起写操作。

规则2 head不存在

如果head对象不存在，创建该对象并写入数据，用SnapContext相应的信息更新SnapSet的信息。

规则3 客户端seq=服务端seq

如果写操作所带的SnapContext的seq值等于SnapSet的seq值，做正常的读写

规则4 客户端seq>服务端seq

如果写操作所带的SnapContext的seq值大于SnapSet的seq值：

1） 对当前head对象做copy操作，clone出一个新的快照对象，该快照对象的snap序号为最新的序号，并把clone操作记录在clones列表里，也就是把最新的快照序号加入到clones队列中。

2） 用SnapContext的seq和snaps值更新SnapSet的seq和snaps值；

3） 写入最新的数据到head对象中

在OSD写操作的流程中，在函数PrimaryLogPG::execute_ctx()中，把消息带的SnapContext信息保存在了OpContext的snapc中：

```
void PrimaryLogPG::execute_ctx(OpContext *ctx)
{
    ...
    ctx->snapc.seq = m->get_snap_seq();
    ctx->snapc.snaps = m->get_snaps();
    ...
}
```

在OpContext的构造函数里，用结构snapset字段初始化了结构new_snapset的相关字段。当前new_snapset保存的就是OSD服务端的快照信息

```
if (obc->ssc) {
    new_snapset = obc->ssc->snapset;
    snapset = &obc->ssc->snapset;
}
```

make_writeable

处理快照相关的写操作，这个函数的目的只在于将不可写的对象克隆，然后到克隆对象执行上写操作

```
void PrimaryLogPG::make_writeable(OpContext *ctx) {
    const hobject_t &soid = ctx->obs->oi.soid;
    SnapContext &snapc = ctx->snapc;

    // clone?
    ceph_assert(soid.snap == CEPH_NOSNAP);
// 1 设置脏标记
    bool was_dirty = ctx->obc->obs.oi.is_dirty();
    if (ctx->new_obs.exists) {
        // we will mark the object dirty
        if (ctx->undirty && was_dirty) {
// 1.1 对象的数据已经被持久化了，清除脏标记
            ceph_assert(ctx->new_obs.oi.is_dirty());
            ctx->new_obs.oi.clear_flag(object_info_t::FLAG_DIRTY);
            --ctx->delta_stats.num_objects_dirty;
            osd->logger->inc(l_osd_tier_clean);
```

```
        } else if (!was_dirty && !ctx->undirty) {
// 1.2 对象被修改后，未持久化，设置脏标记
            ctx->new_obs.oi.set_flag(object_info_t::FLAG_DIRTY);
            ++ctx->delta_stats.num_objects_dirty;
            osd->logger->inc(l_osd_tier_dirty);
        }
    } else {
// 1.3 对象已被删除，清除脏标记
        if (was_dirty) {
            ctx->new_obs.oi.clear_flag(object_info_t::FLAG_DIRTY);
            --ctx->delta_stats.num_objects_dirty;
        }
    }

// 2 过时的快照序号
    if (ctx->new_snapset.seq > snapc.seq) {
        dout(10) << " op snapset is old" << dendl;
    }

    if ((ctx->obs->exists && !ctx->obs->oi.is_whiteout()) && // 对象存在
        snapc.snaps.size() &&           // 有快照
        !ctx->cache_operation &&        // 不是缓存操作
        snapc.snaps[0] > ctx->new_snapset.seq) { // 序号比当前的新
// 3.1 构造clone对象，snap设置为最新的客户端的seq值
        hobject_t coid = soid;
        coid.snap = snapc.seq;

// 3.2 计算本次克隆对象对应的所有快照
        const auto snaps = [&] {
            auto last = find_if_not(
                begin(snapc.snaps), end(snapc.snaps),
                [&](snapid_t snap_id) { return snap_id > ctx->new_snapset.seq;
});
            return vector<snapid_t>{begin(snapc.snaps), last};
        }();

        object_info_t static_snap_oi(coid);
        object_info_t *snap_oi;
        if (is_primary()) {
// 3.3 构造克隆对象的上下文
            ctx->clone_obc =
object_contexts.lookup_or_create(static_snap_oi.soid);
            ctx->clone_obc->destructor_callback =
                new C_PG_ObjectContext(this, ctx->clone_obc.get());
            ctx->clone_obc->obs.oi = static_snap_oi;
            ctx->clone_obc->obs.exists = true;
            ctx->clone_obc->ssc = ctx->obc->ssc;
```

```
62              ctx->clone_obc->ssc->ref++;
63              ...
64          } else {
65          // 这意味着克隆对象的元数据将不会在对象上下文映射中创建
66              snap_oi = &static_snap_oi;
67          }
68          snap_oi->version = ctx->at_version;
69          snap_oi->prior_version = ctx->obs->oi.version;
70          snap_oi->copy_user_bits(ctx->obs->oi);
71
72  // 3.4 创建实际的克隆对象
73          _make_clone(ctx, ctx->op_t.get(), ctx->clone_obc, soid, coid, snap_oi);
74
75  // 4 更新信息
76          ctx->delta_stats.num_objects++;
77          if (snap_oi->is_dirty()) {
78              ctx->delta_stats.num_objects_dirty++;
79              osd->logger->inc(l_osd_tier_dirty);
80          }
81          if (snap_oi->is_omap())
82              ctx->delta_stats.num_objects_omap++;
83          if (snap_oi->is_cache_pinned())
84              ctx->delta_stats.num_objects_pinned++;
85          if (snap_oi->has_manifest())
86              ctx->delta_stats.num_objects_manifest++;
87
88          ctx->delta_stats.num_object_clones++;
89          ctx->new_snapset.clones.push_back(coid.snap);
90          ctx->new_snapset.clone_size[coid.snap] = ctx->obs->oi.size;
91          ctx->new_snapset.clone_snaps[coid.snap] = snaps;
92
93  // 4.1 更新clone_overlap
94          ctx->new_snapset.clone_overlap[coid.snap];
95          if (ctx->obs->oi.size) {
96              ctx->new_snapset.clone_overlap[coid.snap].insert(0, ctx->obs-
    >oi.size);
97          }
98
99          // log clone
100         ctx->log.push_back(pg_log_entry_t(
101             pg_log_entry_t::CLONE, coid, ctx->at_version,
102             ctx->obs->oi.version,
103             ctx->obs->oi.user_version,
104             osd_reqid_t(), ctx->new_obs.oi.mtime, 0));
105         encode(snaps, ctx->log.back().snaps);
106
107         ctx->at_version.version++;
```

```
    }
// 更新clone_overlap
    // update most recent clone_overlap and usage stats
    if (ctx->new_snapset.clones.size() > 0) {
        hobject_t last_clone_oid = soid;
        last_clone_oid.snap = ctx->new_snapset.clone_overlap.rbegin()->first;
        interval_set<uint64_t> &newest_overlap =
            ctx->new_snapset.clone_overlap.rbegin()->second;
        ctx->modified_ranges.intersection_of(newest_overlap);
        if (is_present_clone(last_clone_oid)) {
            // modified_ranges is still in use by the clone
            ctx->delta_stats.num_bytes += ctx->modified_ranges.size();
        }
        newest_overlap.subtract(ctx->modified_ranges);
    }

// 5 更新服务端的快照信息为客户端的快照记录信息
    if (snapc.seq > ctx->new_snapset.seq) {
        // update snapset with latest snap context
        ctx->new_snapset.seq = snapc.seq;
        if (get_osdmap()->require_osd_release < ceph_release_t::octopus) {
            ctx->new_snapset.snaps = snapc.snaps;
        } else {
            ctx->new_snapset.snaps.clear();
        }
    }
}
```

快照的读操作

读操作的核心流程在find_object_context实现，原理是根据读对象的快照序号，查找实际对应的克隆对象的ObjectContext，基本的步骤如下：

```
int PrimaryLogPG::find_object_context(const hobject_t& oid,
                                      ObjectContextRef *pobc,
                                      bool can_create,
                                      bool map_snapid_to_clone,
                                      hobject_t *pmissing)
{
// 1 验证对象ID：确保请求的对象ID与当前PG的池ID匹配。
// 2 处理头部对象：如果请求的是头部对象（没有快照ID），则获取该对象的上下文。
    if (oid.snap == CEPH_NOSNAP) {
        *pobc = get_object_context(oid, can_create);
        return 0;
```

```
    }

// 3 如果请求的是快照对象，首先获取快照集的上下文。
    hobject_t head = oid.get_head();
    SnapSetContext *ssc = get_snapset_context(oid, can_create);

// 3.1 如果请求的是头部对象
    if (oid.snap > ssc->snapset.seq) {
        *pobc = get_object_context(head, false);
        return 0;
    }

// 3.2 如果请求的是克隆对象，找到对应的克隆对象ID，并获取其上下文
    unsigned k = 0;
    while (k < ssc->snapset.clones.size() &&
           ssc->snapset.clones[k] < oid.snap) k++;
    hobject_t soid(oid.oid, oid.get_key(), ssc->snapset.clones[k],
oid.get_hash(),
                   info.pgid.pool(), oid.get_namespace());

// 4 错误处理：如果在任何步骤中找不到对象，或者对象处于不可用状态（如正在恢复），则返回相
应的错误码。
// 5 返回结果：如果成功找到对象上下文，则将其赋值给输出参数并返回成功。
}
```