Ceph 知识总结

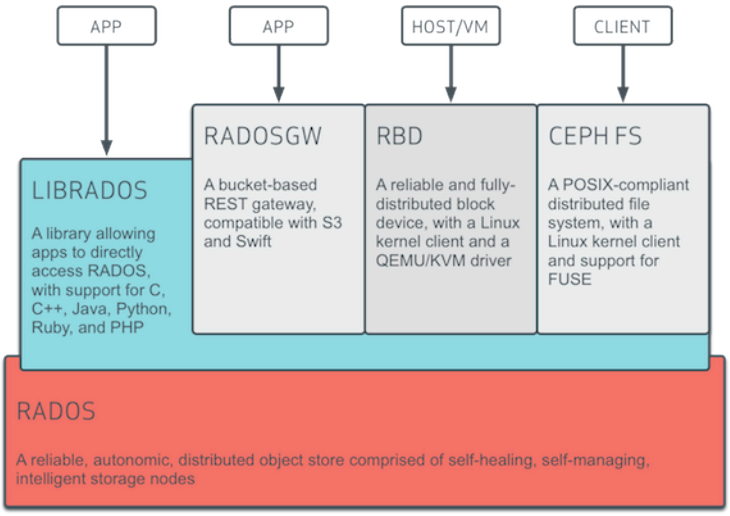

Figure 1: 集群架构

Ceph 基于 rados 提供了无限可扩展的存储集群，集群主要包括两种类型的后台进程 osd 和 mon，monitor 主要负责维护 cluster map，osd 会检测自己和邻居的状态，并上报给 monitor。

典型的 RADOS 部署架构由少量的 Monitor 监控器以及大量的 OSD 存储设备组成，它能够在动态变化的基于异质结构的存储设备集群之上提供一种稳定的、可扩展的、高性能的单一逻辑对象存储接口。RADOS 系统的架构如图所示：

Ceph 集群图：

Monitor map: mon 节点信息，包括 ceph 集群 id，监视器主机名，ip 地址和端口号，从创建 mon 至今的 epoch 信息和修改的时间

OSD map: 保存集群 id，osd 的创建和修改日期，还有跟池关联的一些信息，比如池的名称，ID，类型，副本级别和 PG。

PG map: 保存 pg 版本，时间戳，最后的 OSD map 的版本，full ratio，near full ratio。它还保存每个 pg 的 id，对象数，状态，状态时间戳，up 和 acting 的 OSD 集合，数据清洗详情。

CRUSH map: 保存集群中有关设备，桶，故障域等级和存储数据的规则集。

MDS map: 保存当前 map 的版本，map 创建和修改的时间，数据和元数据池的 ID，集群的 MDS 数量和 MDS 状态。

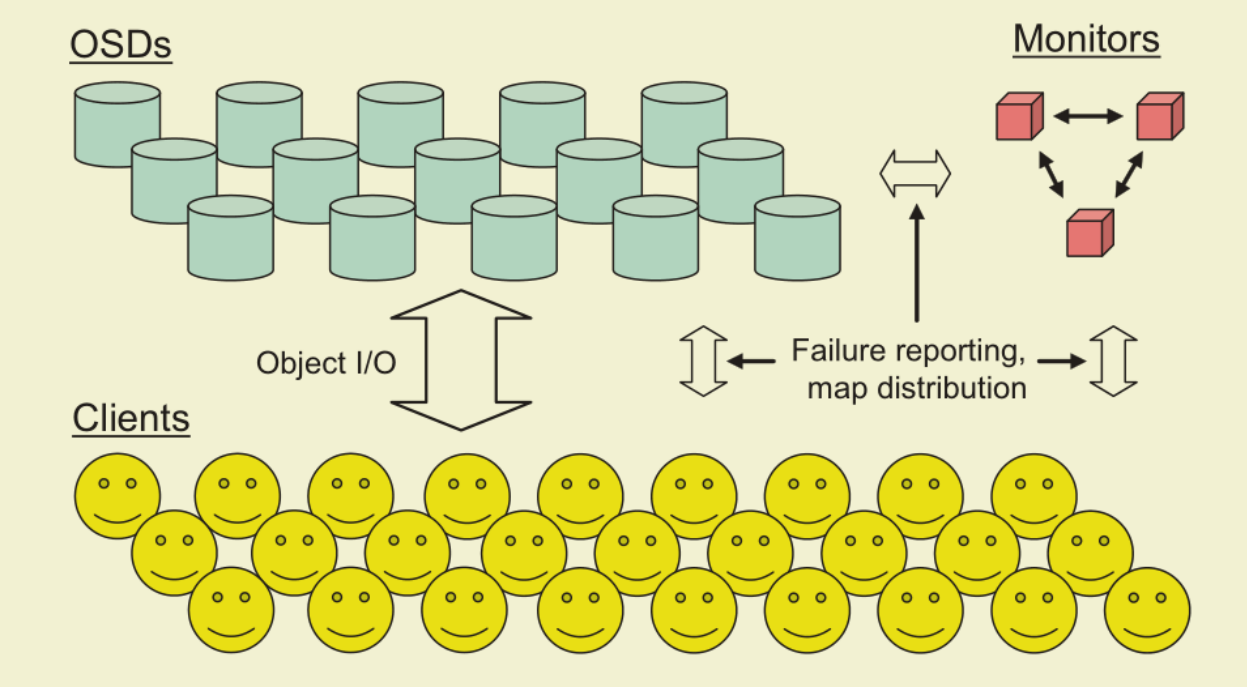

Figure 2: rados

Ceph 基础组件:

Object：Ceph 最底层的存储单元是 Object 对象，每个 Object 包含元数据和原始数据。

OSD：承担数据 IO，数据恢复工作并负责响应客户端请求返回具体数据的进程

PG：是一个逻辑概念，一个 pg 包含多个 osd，引入 pg 是为了更好的分配和定位数据

MON：用来保存 OSD 的元数据

CRUSH: Ceph 使用的数据分布算法，类似一致性哈希，让数据分配到预期的地方

RBD: RADOS block device，是 Ceph 对外提供的块设备服务

RGW: RADOS gateway，是 Ceph 对外提供的对象存储服务，接口与 S3 和 Swift 兼容

MDS: Ceph Metadata Server，是 CephFS 服务依赖的元数据服务

libradio: 上层的 RBD、RGW 和 CephFS 都是通过 librados 访问的，目前提供 PHP、Ruby、Java、Python、C 和 C++ 支持

librbd: ceph 的块存储库，利用 Rados 提供的 API 实现对块设备的管理和操作

Ceph 的寻址及数据操作流程

寻址机制

Ceph 系统中的寻址流程如下图所示：

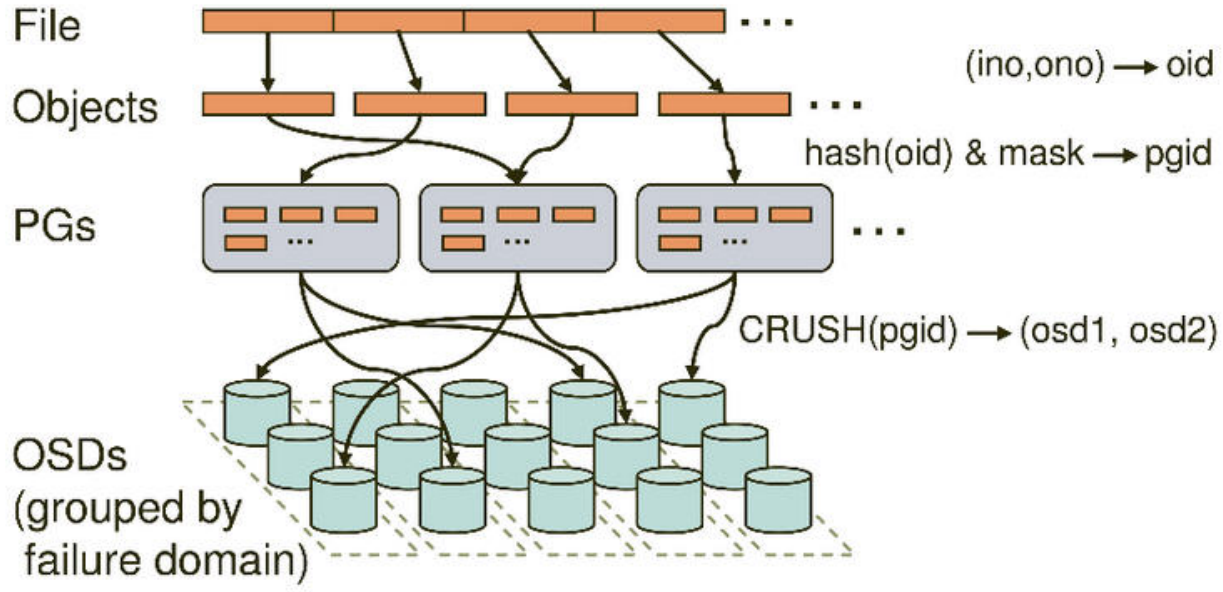

Figure 3: 寻址流程

上图中左侧的几个概念说明如下：

File：用户需要存储或者访问的文件。对于一个基于 Ceph 开发的对象存储应用而言，这个 file 也就对应于应用中的“对象”，也就是用户直接操作的“对象”。

Ojbect：此处的 object 是 RADOS 所看到的“对象”。Object 与上面提到的 file 的区别是，object 的最大 size 由 RADOS 限定（通常为 2MB 或 4MB），以便实现底层存储的组织管理。因此，当上层应用向 RADOS 存入 size 很大的 file 时，需要将 file 切分成统一大小的一系列 object（最后一个的大小可以不同）进行存储。

PG(Placement Group): 上面已经简单介绍过了 PG，PG 的用途是对 object 的存储进行组织和位置映射。具体而言，一个 PG 负责组织若干个 object（可以为数千个甚至更多），但一个 object 只能被映射到一个 PG 中，即，PG 和 object 之间是“一对多”映射关系。同时，一个 PG 会被映射到 n 个 OSD 上，而每个 OSD 上都会承载大量的 PG，即，PG 和 OSD 之间是“多对多”映射关系。在实践当中，n 至少为 2，如果用于生产环境，则至少为 3。一个 OSD 上的 PG 则可达到数百个。事实上，PG 数量的设置牵扯到数据分布的均匀性问题。

OSD：上文也已经简单提到，需要说明的是，OSD 的数量事实上也关系到系统的数据分布均匀性，因此其数量不应太少。

故障域：指任何导致不能访问一个或多个 OSD 的故障，可以是主机上停止的进程、硬盘故障、操作系统崩溃、有问题的网卡、损坏的电源、断网、断点等等。规划硬件需求时，要在多个需求间寻求平衡点，像付出很多努力减少故障域带来的成本削减、隔离每个潜在故障域增加的成本。

基于上述定义，下面介绍下寻址流程。具体而言，ceph 中的寻址至少要经历一下三次映射：

(1) File->Object

这次映射的目的是，经用户要操作的 File，映射为 Rados 能够处理的 Object。其映射十分简单，本质上就是按照 Object 的最大大小对 File 进行切分。这种切分的好处有二，一是让大小不限的 file 变成最大 size 一致、可以被 Rados 高效管理的 Object；二是让对单一 file 实施的串行处理变为对多个 object 实施的并行化处理。每一个切分后产生的 object 将获得唯一的 oid，即 object id。其产生方式也是线性映射，极其简单。图中，ino 是待操作 file 的元数据，可以简单理解为该 file 的唯一 id。ono 则是由该 file 切分产生的某个 object 的序号。而 oid 就是将这个序号简单连缀在该 file id 之后得到的。举例而言，如果一个 id 为 filename 的 file 被切分成了三个 object，则其 object 序号依次为 0、1 和 2，而最终得到的 oid 就依次为 filename0、filename1 和 filename2。

ino 的唯一性必须得到保证，后则后续的映射无法正确进行。

(2) Object->PG

在 File 被映射为一个或多个 Object 以后，就需要将每个 Object 独立地映射到一个 PG 中去。这个映射也很简单，如图中所示，其计算公式是：

```
hash(oid) & mask -> pgid
```

其计算由两步组成。首先是使用 Ceph 系统指定的一个静态哈希函数计算 oid 的哈希值，将 oid 映射成为一个近似均匀分布的伪随机值。然后，将这个伪随机值和 mask 按位相与，得到最终的 PG 序号（pgid）。根据 RADOS 的设计，给定 PG 的总数为 m（m 应该为 2 的整数幂），则 mask 的值为 m-1。因此，哈希值计算和按位与操作的整体结果事实上是从所有 m 个 PG 中近似均匀地随机选择一个。基于这一机制，当有大量 object 和大量 PG 时，RADOS 能够保证 object 和 PG 之间的近似均匀映射。又因为 object 是由 file 切分而来，大部分 object 的 size 相同，因而，这一映射最终保证了，各个 PG 中存储的 object 的总数据量近似均匀。

从介绍不难看出，这里反复强调了“大量”。只有当 object 和 PG 的数量较多时，这种伪随机关系的近似均匀性才能成立，Ceph 的数据存储均匀性才有保证。为保证“大量”的成立，一方面，object 的最大 size 应该被

合理配置，以使得同样数量的 file 能够被切分成更多的 object；另一方面，Ceph 也推荐 PG 总数应该为 OSD 总数的数百倍，以保证有足够数量的 PG 可供映射。

（3）PG->OSD

第三次映射就是将作为 object 的逻辑组织单元的 PG 映射到数据的实际存储单元 OSD。如图所示，RADOS 采用一个名为 CRUSH 的算法，将 pgid 代入其中，然后得到一组共 n 个 OSD。这 n 个 OSD 即共同负责存储和维护一个 PG 中的所有 object。前已述及，n 的数值可以根据实际应用中对于可靠性的需求而配置，在生产环境下通常为 3。具体到每个 OSD，则由其上运行的 OSD deamon 负责执行映射到本地的 object 在本地文件系统中的存储、访问、元数据维护等操作。

和 “object -> PG” 映射中采用的哈希算法不同，这个 CRUSH 算法的结果不是绝对不变的，而是受到其他因素的影响。其影响因素主要有二：

一是当前系统状态，也就是上文逻辑结构中曾经提及的 cluster map。当系统中的 OSD 状态、数量发生变化时，cluster map 可能发生变化，而这种变化将会影响到 PG 与 OSD 之间的映射。

二是存储策略配置。这里的策略主要与安全相关。利用策略配置，系统管理员可以指定承载同一个 PG 的 3 个 OSD 分别位于数据中心的不同服务器乃至机架上，从而进一步改善存储的可靠性。

因此，只有在系统状态（cluster map）和存储策略都不发生变化的时候，PG 和 OSD 之间的映射关系才是固定不变的。在实际使用当中，策略一经配置通常不会改变。而系统状态的改变或者是由于设备损坏，或者是因为存储集群规模扩大。好在 Ceph 本身提供了对于这种变化的自动化支持，因而，即便 PG 与 OSD 之间的映射关系发生了变化，也并不会对应用造成困扰。事实上，Ceph 正是需要有目的的利用这种动态映射关系。正是利用了 CRUSH 的动态特性，Ceph 可以将一个 PG 根据需要动态迁移到不同的 OSD 组合上，从而自动化地实现高可靠性、数据分布 re-blancing 等特性。

之所以在此次映射中使用 CRUSH 算法，而不是其他哈希算法，原因之一正是 CRUSH 具有上述可配置特性，可以根据管理员的配置参数决定 OSD 的物理位置映射策略；另一方面是因为 CRUSH 具有特殊的 “稳定性”，也即，当系统中加入新的 OSD，导致系统规模增大时，大部分 PG 与 OSD 之间的映射关系不会发生改变，只有少部分 PG 的映射关系会发生变化并引发数据迁移。这种可配置性和稳定性都不是普通哈希算法所能提供的。因此，CRUSH 算法的设计也是 Ceph 的核心内容之一，具体介绍可以参考。

至此为止，Ceph 通过三次映射，完成了从 file 到 object、PG 和 OSD 整个映射过程。通观整个过程，可以看到，这里没有任何的全局性查表操作需求。至于唯一的全局性数据结构 cluster map，它的维护和操作都是轻量级的，不会对系统的可扩展性、性能等因素造成不良影响。

从上可以看出，引入 PG 的好处至少有二：一方面实现了 object 和 OSD 之间的动态映射，从而为 Ceph 的可靠性、自动化等特性的实现留下了空间；另一方面也有效简化了数据的存储组织，大大降低了系统的维护管理开销。理解这一点，对于彻底理解 Ceph 的对象寻址机制，是十分重要的。

数据操作流程

此处将首先以 file 写入过程为例，对数据操作流程进行说明。

为简化说明，便于理解，此处进行若干假定。这里假定待写入的 file 较小，无需切分，仅被映射为一个 object。其次，假定系统中一个 PG 被映射到 3 个 OSD 上。

基于上述假定，则 file 写入流程可以被下图表示：

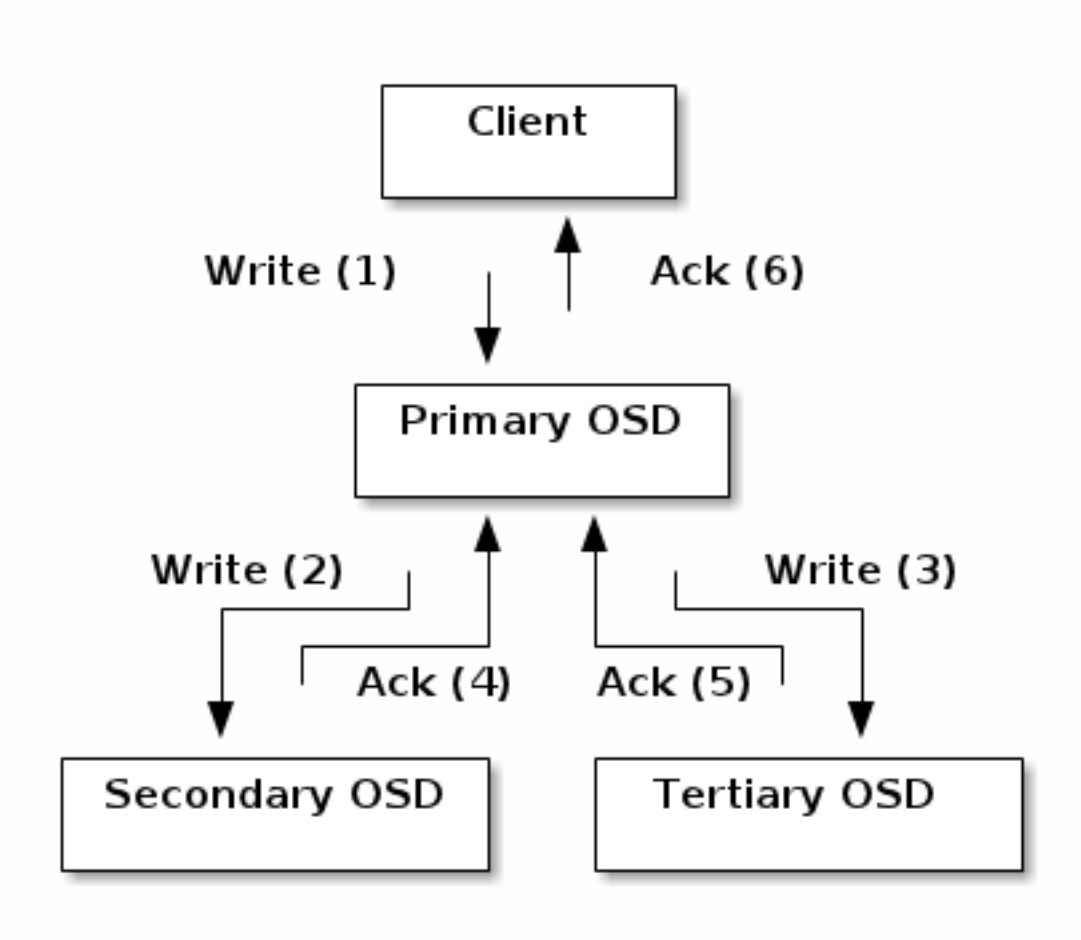

Figure 4: 数据操作流程

如图所示，当某个 client 需要向 Ceph 集群写入一个 file 时，首先需要在本地完成上述的寻址流程，将 file 变为一个 object，然后找出存储该 object 的一组三个 OSD。这三个 OSD 具有各自不同的序号，序号最靠前的那个 OSD 就是这一组中的 Primary OSD，而后两个则依次是 Secondary OSD 和 Tertiary OSD。

找出三个 OSD 后，client 将直接和 Primary OSD 通信，发起写入操作（步骤 1）。Primary OSD 收到请求后，分别向 Secondary OSD 和 Tertiary OSD 发起写入操作（步骤 2、3）。当 Secondary OSD 和 Tertiary OSD 各

自完成写入操作后，将分别向 Primary OSD 发送确认信息（步骤 4、5）。当 Primary OSD 确信其他两个 OSD 的写入完成后，则自己也完成数据写入，并向 client 确认 object 写入操作完成（步骤 6）。

之所以采用这样的写入流程，本质上是为了保证写入过程中的可靠性，尽可能避免造成数据丢失。同时，由于 client 只需要向 Primary OSD 发送数据，因此，在 Internet 使用场景下的外网带宽和整体访问延迟又得到了一定程度的优化。

当然，这种可靠性机制必然导致较长的延迟，特别是，如果等到所有的 OSD 都将数据写入磁盘后再向 client 发送确认信号，则整体延迟可能难以忍受。因此，Ceph 可以分两次向 client 进行确认。当各个 OSD 都将数据写入内存缓冲区后，就先向 client 发送一次确认，此时 client 即可以向下执行。待各个 OSD 都将数据写入磁盘后，会向 client 发送一个最终确认信号，此时 client 可以根据需要删除本地数据。

分析上述流程可以看出，在正常情况下，client 可以独立完成 OSD 寻址操作，而不必依赖于其他系统模块。因此，大量的 client 可以同时和大量的 OSD 进行并行操作。同时，如果一个 file 被切分成多个 object，这多个 object 也可被并行发送至多个 OSD。

从 OSD 的角度来看，由于同一个 OSD 在不同的 PG 中的角色不同，因此，其工作压力也可以被尽可能均匀地分担，从而避免单个 OSD 变成性能瓶颈。

如果需要读取数据，client 只需完成同样的寻址过程，并直接和 Primary OSD 联系。目前的 Ceph 设计中，被读取的数据仅由 Primary OSD 提供。但目前也有分散读取压力以提高性能的讨论。

集群维护

前面的介绍中已经提到，由若干个 monitor 共同负责整个 Ceph 集群中所有 OSD 状态的发现与记录，并且共同形成 cluster map 的 master 版本，然后扩散至全体 OSD 以及 client。OSD 使用 cluster map 进行数据的维护，而 client 使用 cluster map 进行数据的寻址。

在集群中，各个 monitor 的功能总体上是一样的，其相互间的关系可以被简单理解为主从备份关系。因此，在下面的讨论中不对各个 monitor 加以区分。

略显出乎意料的是，monitor 并不主动轮询各个 OSD 的当前状态。正相反，OSD 需要向 monitor 上报状态信息。常见的上报有两种情况：一是新的 OSD 被加入集群，二是某个 OSD 发现自身或者其他 OSD 发生异常。在收到这些上报信息后，monitor 将更新 cluster map 信息并加以扩散。其细节将在下文中加以介绍。

Cluster map 的实际内容包括：

(1) Epoch，即版本号。Cluster map 的 epoch 是一个单调递增序列。Epoch 越大，则 cluster map 版本越新。因此，持有不同版本 cluster map 的 OSD 或 client 可以简单地通过比较 epoch 决定应该遵从谁手

中的版本。而 monitor 手中必定有 epoch 最大、版本最新的 cluster map。当任意两方在通信时发现彼此 epoch 值不同时，将默认先将 cluster map 同步至高版本一方的状态，再进行后续操作。

(2) 各个 OSD 的网络地址。

(3) 各个 OSD 的状态。OSD 状态的描述分为两个维度：up 或者 down（表明 OSD 是否正常工作），in 或者 out（表明 OSD 是否在至少一个 PG 中）。因此，对于任意一个 OSD，共有四种可能的状态：

Up 且 in：说明该 OSD 正常运行，且已经承载至少一个 PG 的数据。这是一个 OSD 的标准工作状态；

Up 且 out：说明该 OSD 正常运行，但并未承载任何 PG，其中也没有数据。一个新的 OSD 刚刚被加入 Ceph 集群后，便会处于这一状态。而一个出现故障的 OSD 被修复后，重新加入 Ceph 集群时，也是处于这一状态；

Down 且 in：说明该 OSD 发生异常，但仍然承载着至少一个 PG，其中仍然存储着数据。这种状态下的 OSD 刚刚被发现存在异常，可能仍能恢复正常，也可能会彻底无法工作；

Down 且 out：说明该 OSD 已经彻底发生故障，且已经不再承载任何 PG。

(4) CRUSH 算法配置参数。表明了 Ceph 集群的物理层级关系（cluster hierarchy），位置映射规则（placement rules）。

根据 cluster map 的定义可以看出，其版本变化通常只会由（3）和（4）两项信息的变化触发。而这两者相比，（3）发生变化的概率更高一些。这可以通过下面对 OSD 工作状态变化过程的介绍加以反映。

一个新的 OSD 上线后，首先根据配置信息与 monitor 通信。Monitor 将其加入 cluster map，并设置为 up 且 out 状态，再将最新版本的 cluster map 发给这个新 OSD。

收到 monitor 发来的 cluster map 之后，这个新 OSD 计算出自己所承载的 PG（为简化讨论，此处我们假定这个新的 OSD 开始只承载一个 PG），以及和自己承载同一个 PG 的其他 OSD。然后，新 OSD 将与这些 OSD 取得联系。如果这个 PG 目前处于降级状态（即承载该 PG 的 OSD 个数少于正常值，如正常应该是 3 个，此时只有 2 个或 1 个。这种情况通常是 OSD 故障所致），则其他 OSD 将把这个 PG 内的所有对象和元数据复制给新 OSD。数据复制完成后，新 OSD 被置为 up 且 in 状态。而 cluster map 内容也将据此更新。这事实上是一个自动化的 failure recovery 过程。当然，即便没有新的 OSD 加入，降级的 PG 也将计算出其他 OSD 实现 failure recovery。

如果该 PG 目前一切正常，则这个新 OSD 将替换掉现有 OSD 中的一个（PG 内将重新选出 Primary OSD），并承担其数据。在数据复制完成后，新 OSD 被置为 up 且 in 状态，而被替换的 OSD 将退出该 PG（但状态通常仍然为 up 且 in，因为还要承载其他 PG）。而 cluster map 内容也将据此更新。这事实上是一个自动化的数据 re-balancing 过程。

如果一个 OSD 发现和自己共同承载一个 PG 的另一个 OSD 无法联通，则会将这一情况上报 monitor。此

外，如果一个 OSD deamon 发现自身工作状态异常，也将把异常情况主动上报给 monitor。在上述情况下，monitor 将把出现问题的 OSD 的状态设为 down 且 in。如果超过某一预订时间期限，该 OSD 仍然无法恢复正常，则其状态将被设置为 down 且 out。反之，如果该 OSD 能够恢复正常，则其状态会恢复为 up 且 in。在上述这些状态变化发生之后，monitor 都将更新 cluster map 并进行扩散。这事实上是自动化的 failure detection 过程。

由之前介绍可以看出，对于一个 Ceph 集群而言，即便由数千个甚至更多 OSD 组成，cluster map 的数据结构大小也并不惊人。同时，cluster map 的状态更新并不会频繁发生。即便如此，Ceph 依然对 cluster map 信息的扩散机制进行了优化，以便减轻相关计算和通信压力。

首先，cluster map 信息是以增量形式扩散的。如果任意一次通信的双方发现其 epoch 不一致，则版本更新的一方将把二者所拥有的 cluster map 的差异发送给另外一方。

其次，cluster map 信息是以异步且 lazy 的形式扩散的。也即，monitor 并不会在每一次 cluster map 版本更新后都将新版本广播至全体 OSD，而是在有 OSD 向自己上报信息时，将更新回复给对方。类似的，各个 OSD 也是在和其他 OSD 通信时，将更新发送给版本低于自己的对方。

基于上述机制，Ceph 避免了由于 cluster map 版本更新而引起的广播风暴。这虽然是一种异步且 lazy 的机制，但根据 Sage 论文中的结论，对于一个由 n 个 OSD 组成的 Ceph 集群，任何一次版本更新能够在 O(log(n)) 时间复杂度内扩散到集群中的任何一个 OSD 上。

一个可能被问到的问题是：既然这是一种异步和 lazy 的扩散机制，则在版本扩散过程中，系统必定出现各个 OSD 看到的 cluster map 不一致的情况，这是否会导致问题？答案是：不会。事实上，如果一个 client 和它要访问的 PG 内部的各个 OSD 看到的 cluster map 状态一致，则访问操作就可以正确进行。而如果这个 client 或者 PG 中的某个 OSD 和其他几方的 cluster map 不一致，则根据 Ceph 的机制设计，这几方将首先同步 cluster map 至最新状态，并进行必要的数据 re-balancing 操作，然后即可继续正常访问。

通过上述介绍，我们可以简要了解 Ceph 究竟是如果基于 cluster map 机制，并由 monitor、OSD 和 client 共同配合完成集群状态的维护与数据访问的。特别的，基于这个机制，事实上可以自然而然的完成自动化的数据备份、数据 re-balancing、故障探测和故障恢复，并不需要复杂的特殊设计。这一点确实让人印象深刻。

Ceph 模块间的关系

MSG

上图中的 MSG 指的是 Messenger 类的实例，在 OSD 中使用的是其子类 SimpleMessenger，其继承关系如下：

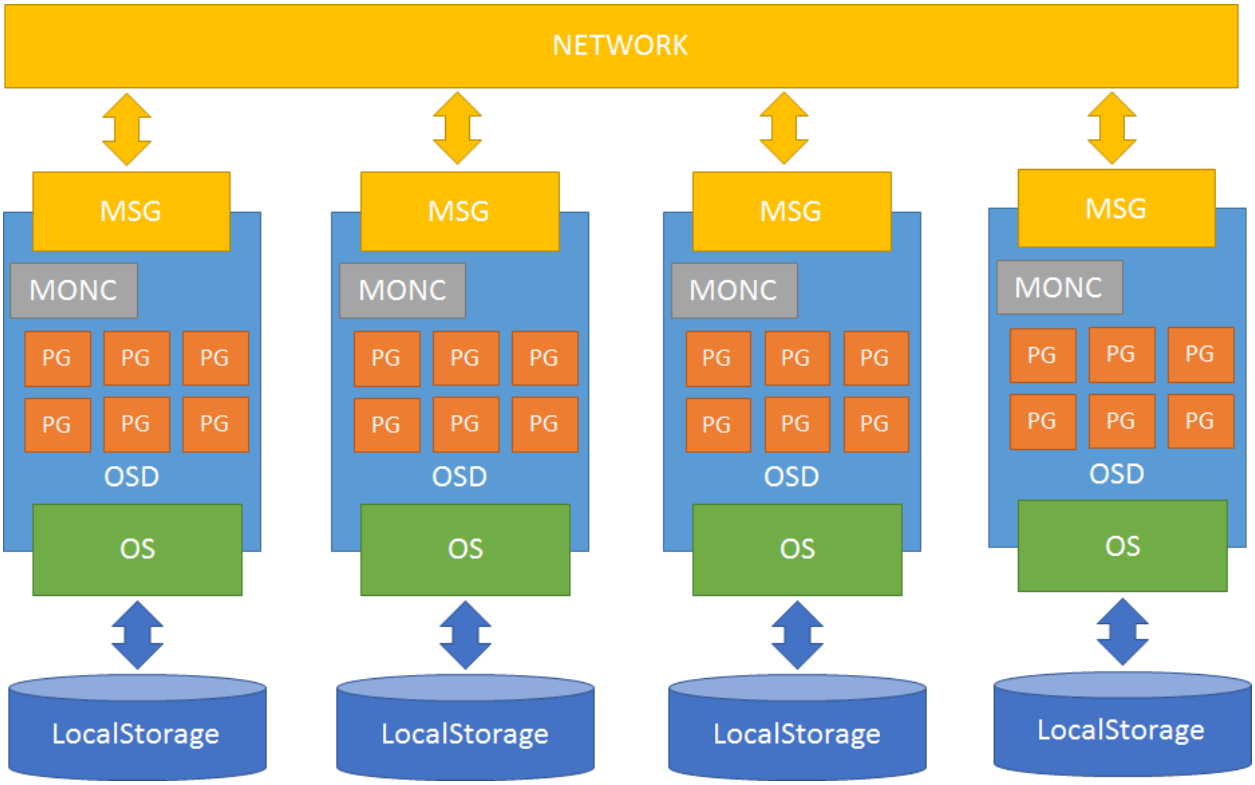

Figure 5: 模块关系图

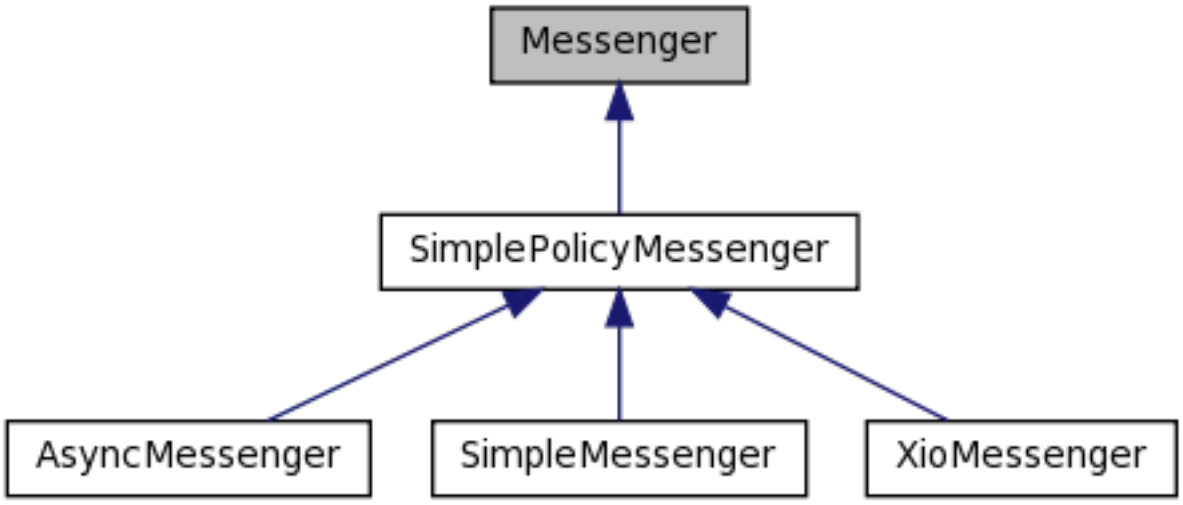

Figure 6: Messenger 类图

启动 Messenger 这里以 ceph 的 osd 服务为例来介绍 Messenger，源码为 `ceph_osd.cc`

(1) 启动 Messenger 对象，总共有如下几个 Messenger 会被创建:

- ms_public
- ms_cluster
- ms_hbclient
- `ms_hb_back_server`
- `ms_hb_front_server`
- ms_objecter

创建这些实例的时候，基类 Messenger 调用 create 方法，根据配置的 `ms_type` 类型来实例化具体的 Messenger 对象，目前的 Messenger 有三种不同的实现：SimpleMessenger, AsyncMessenger 和 XioMessenger

SimpleMessenger: 相对比较简单，目前可以在生产环境中使用的模式。它最大的特点是，每一个连接，都创建两个线程，一个专门用于接收，一个专门用于发送

AsyncMessenger: 使用了基于事件 IO 的多路复用模式，这是比较通用的方式，没有用第三方库，实现起来比较复杂，目前还处于试验阶段，ceph 源码中基于 epoll 来实现，有助于减少集群中网络通信所需要的线程数，目前虽然不是默认的通信组件，但是以后一定会取代 SimpleMessenger

XioMessenger: 使用了开源的网络通信模块 accelio 来实现，依赖第三方库，实现起来较简单，但需要熟悉 accelio 的使用方式，目前也处于试验阶段

默认的 `ms_type` 类型为 “simple”

(2) 设置 Messenger 的协议为 CEPH_OSD_PROTOCOL

```
set_cluster_protocol(CEPH_OSD_PROTOCOL);
```

(3) 设置两个 throttler(流量控制器)，分别用于限制 OSD 在网络层接受 client 请求的消息大小和消息数量，默认为 500M 和 100

```
boost::scoped_ptr<Throttle> client_byte_throttler(new Throttle(g_ceph_context, "osd_client_
boost::scoped_ptr<Throttle> client_msg_throttler( new Throttle(g_ceph_context, "osd_client_
```

(4) 设置 OSD 服务支持的特性

(5) 设置每个 Messenger 的 policy

(6) 绑定地址

`ms_public` 绑定到 `g_conf->public_addr` 上

`ms_cluster` 绑定到 `g_cong->cluster_addr` 上

`ms_hb_back_server` 绑定到 `g_conf->hb_back_addr` 上

`ms_hb_front_server` 绑定到 `g_conf->hb_front_addr` 上

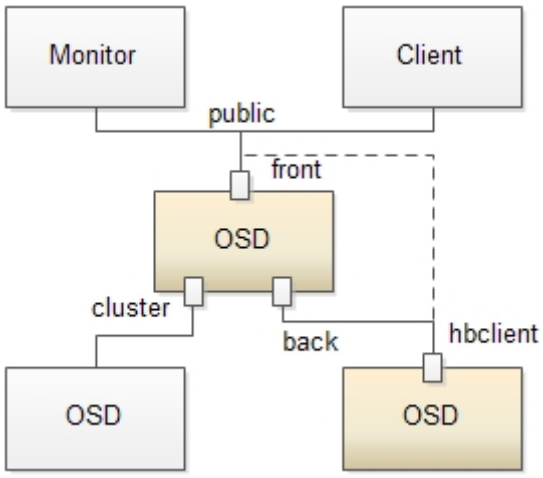

Figure 7: OSD,Monitor 和 Client 之间的连接

Messenger 的作用 从上可以看出，对于 OSD 服务而言，启动了多个 Messenger 监听器，每个监听器的作用如下，其中 OSD 节点会监听 public、cluster、front 和 back 四个端口。

- public 监听来自 Monitor 和 Client 的连接
- cluster 监听处理来自 OSD peer 的连接
- 另外，OSD 单独创建了一个名为 hbclient 的 Messenger，作为心跳的客户端，单独用来建立连接发送心跳报文，心跳优先发送给 back 连接

消息分发方式 总体上，Ceph 的消息处理框架是发布者订阅者的设计结构。Messenger 担当发布者的角色，Dispatcher 担当订阅者的角色。Messenger 将接收到的消息通知给已注册的 Dispatcher，由 Dispatcher 完成具体的消息处理。

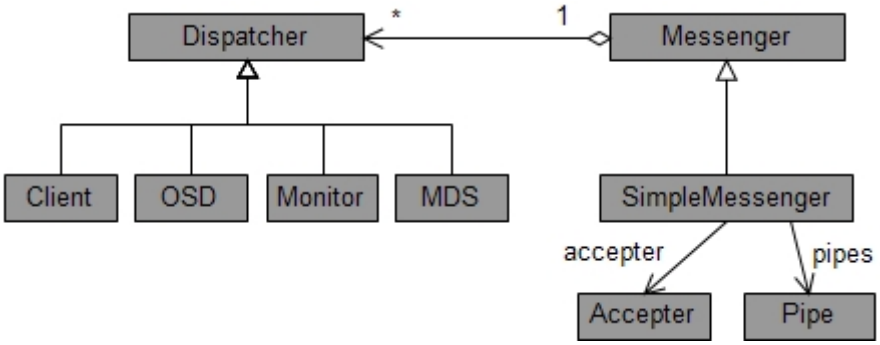

Figure 8: MSG 通信模型

在服务端，SimpleMessenger 通过 Accepter 实例监听端口，接收来自客户端的连接。Accepter 接受客户端的连接后，为该连接创建一个 Pipe 实例。

Pipe 实例负责具体消息的接收和发送，一个 Pipe 实例包含一个读线程和一个写线程。读线程读取到消息后，有三种分发消息的方法：

(1) 快速分发，直接在 Pipe 的读线程中处理掉消息。可快速分发的消息在 Dispatcher 的 ms_can_fast_dispatch 中注册。

(2) 正常分发，将消息放入 DispatchQueue，由单独的线程按照消息的优先级从高到低进行分发处理。需要注意的是，属于同一个 SimpleMessenger 实例的 Pipe 间使用同个 DispatchQueue。

(3) 延迟分发，为消息随机设置延迟时间，定时时间到时由单独的线程走快速分发或正常分发的流程分发消息。Pipe 的写线程将消息放入 out_q 队列，按照消息的优先级从高到低发送消息。另外，消息 (Message) 中携带了 seq 序列号，Pipe 使用 `in_seq` 和 out_seq 记录它接收到和发送出去的消息的序列号。发送消息时，Pipe 用 out_seq 设置消息的序列号；接收消息时，通过比较消息的序列号和 `in_seq` 来确定消息是否为旧消息，如果为旧消息则丢弃，否则使用消息的序列号更新 in_seq。

RBD

Ceph 的块存储有两种使用途径，一种是利用 librbd，另一种是使用内核模块。第一种主要为虚拟机提供块存储设备，第二种主要为 Host 提供块设备支持，这两种途径的接口实现完全不同。librbd 在 ceph 源码里已经提供，而且更稳定，也是 ceph 应用场景最广泛的。

kvm 虚拟机使用 rbd 设备：

定义如下 xml 文件：

rbd_device.xml

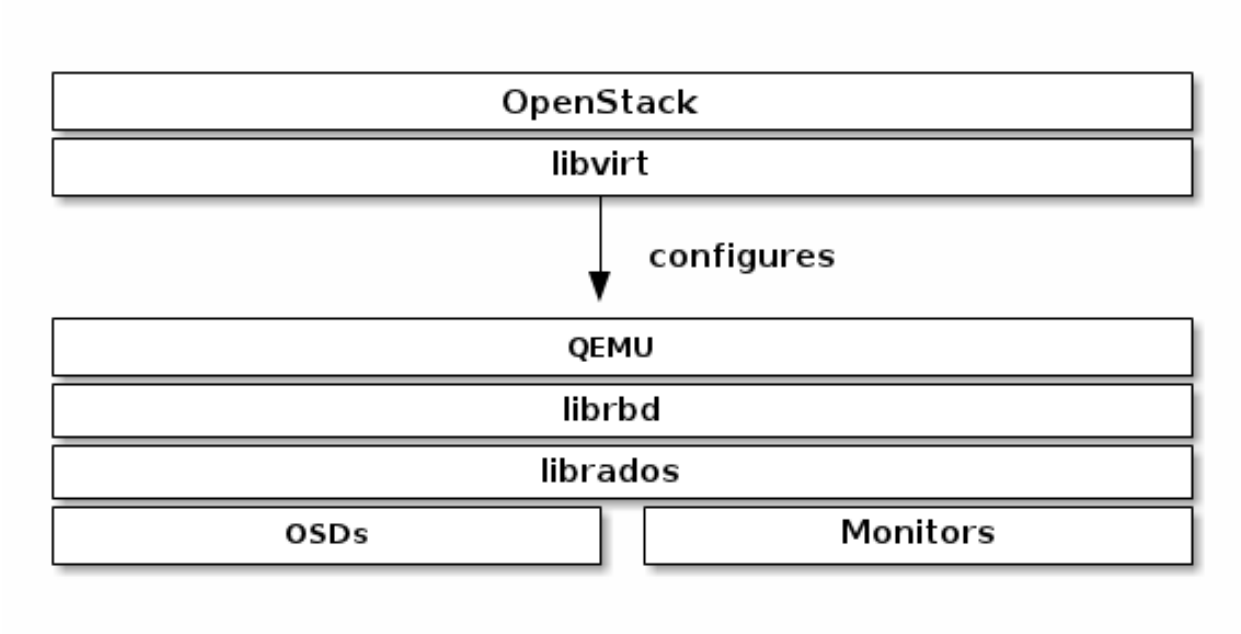

Figure 9: qemu_rbd

```
<disk type='network' device='disk'>
  <driver name='qemu' type='raw' cache='writeback'/>
  <auth username='admin'>
    <secret type='ceph' uuid='e403fe43-3e9e-473b-9853-44ea881489d2'/>
  </auth>
  <source protocol='rbd' name='sata/test'>
    <host name='192.168.104.11' port='6789'/>
    <host name='192.168.104.12' port='6789'/>
    <host name='192.168.104.13' port='6789'/>
    <!--<snapshot name='snapname'/>-->
    <!--<config file='/path/to/file'/>-->
  </source>
  <target dev='vdc' bus='virtio'/>
  <!--<iotune>-->
    <!--<read_bytes_sec>209715200</read_bytes_sec>-->
    <!--<write_bytes_sec>167772160</write_bytes_sec>-->
    <!--<read_iops_sec>80</read_iops_sec>-->
    <!--<write_iops_sec>40</write_iops_sec>-->
    <!--</iotune>-->
  <alias name='virtio-disk0'/>
  <address type='pci' domain='0x0000' but='0x00' slot='0x07' function='0x0'/>
</disk>
```

使用 `virsh attah-device <domain_id> rbd_device.xml` 将 ceph 的 rbd 镜像附加到虚拟机里，在虚拟机里对应的设备为/dev/vdx

PG

PG 状态及变迁

Creating:

创建存储池时, 它会创建指定数量的归置组。ceph 在创建一或多个归置组时会显示 creating; 创建完后, 在其归置组的 Acting Set 里的 OSD 将建立互联; 一旦互联完成, 归置组状态应该变为 active+clean, 意思是 ceph 客户端可以向归置组写入数据了。

Peering:

ceph 为归置组建立互联时, 会让存储归置组副本的 OSD 之间就其中的对象和元数据状态达成一致。ceph 完成了互联, 也就意味着存储着归置组的 OSD 就其当前状态达成了一致。然而, 互联过程的完成并不能表明各副本都有了数据的最新版本。

Active:

ceph 完成互联进程后, 一归置组就可变为 active。active 状态通常意味着在主归置组和副本中的数据都可以读写。

Clean:

某一归置组处于 clean 状态时, 主 OSD 和副本 OSD 已成功互联, 并且没有偏离的归置组。ceph 已把归置组中的对象复制了规定次数。

Degraded:

当客户端向主 OSD 写入数据时, 由主 OSD 负责把副本写入其余复制 OSD。主 OSD 把对象写入复制 OSD 后, 在没收到成功完成的确认前, 主 OSD 会一直停留在 degraded 状态。归置组状态可以是 active+degraded 状态, 原因在于一 OSD 即使没所有对象也可以处于 active 状态。如果一 OSD 挂了,ceph 会把相关的归置组都标记为 degraded; 那个 OSD 重生后, 它们必须重新互联。然而, 如果归置组仍处于 active 状态, 即便它处于 degraded 状态, 客户端还可以向其写入新对象。如果一 OSD 挂了, 且 degraded 状态持续,ceph 会把 down 的 OSD 标记为在集群外 (out)、并把那些 down 掉的 OSD 上的数据重映射到其它 OSD。从标记为 down 到 out 的时间间隔由 mon osd down out interval 控制, 默认是 300 秒。归置组也会被降级 (degraded), 因为归置组

找不到本应存在于归置组中的一或多个对象, 这时, 你不能读或写找不到的对象, 但仍能访问其它位于降级归置组中的对象。

Recovering:

ceph 被设计为可容错, 可抵御一定规模的软、硬件问题。当某 OSD 挂了 (down) 时, 其内容版本会落后于归置组内的其它副本; 它重生 (up) 时, 归置组内容必须更新, 以反映当前状态; 在此期间,OSD 在 recovering 状态。恢复并非总是这些小事, 因为一次硬件失败可能牵连多个 OSD。比如一个机柜的网络交换机失败了, 这会导致多个主机落后于集群的当前状态, 问题解决后每一个 OSD 都必须恢复。ceph 提供了很多选项来均衡资源竞争, 如新服务请求、恢复数据对象和恢复归置组到当前状态。osd recovery delay start 选项允许一 OSD 在开始恢复进程前, 先重启、重建互联、甚至处理一些重放请求;osd recovery threads 选项限制恢复进程的线程数, 默认为 1 线程; osd recovery thread timeout 设置线程超时, 因为多个 OSD 可能交替失败、重启和重建互联; osd recovery max active 选项限制一 OSD 最多同时接受多少请求, 以防它压力过大而不能正常服务; osd recovery max chunk 选项限制恢复数据块尺寸, 以防网络拥塞。

Back Filling:

有新 OSD 加入集群时,CRUSH 会把现有集群内的归置组重分配给它。强制新 OSD 立即接受重分配的归置组会使之过载, 用归置组回填可使这个过程在后台开始。回填完成后, 新 OSD 准备好时就可以对外服务了。

Remapped:

某一 PG 的 Acting Set 变更时, 数据要从旧集合迁移到新的。主 OSD 要花费一些时间才能提供服务, 所以它可以让老的主 OSD 持续服务，直到归置组迁移完。数据迁移完后, 主 OSD 会映射到新的 acting set。

Stale:

虽然 ceph 用心跳来保证主机和守护进程在运行, 但是 ceph-osd 仍有可能进入 stuck 状态, 它们没有按时报告其状态 (如网络瞬断)。默认,OSD 守护进程每半秒 (0.5) 会一次报告其归置组、出流量、引导和失败统计状态, 此频率高于心跳阀值。如果一归置组的主 OSD 所在的 acting set 没能向监视器报告、或者其它监视器已经报告了那个主 OSD 已 down, 监视器们就会把此归置组标记为 stale。启动集群时, 会经常看到 stale 状态, 直到互联完成。集群运行一阵后, 如果还能看到有归置组位于 stale 状态, 就说明那些归置组的主 OSD 挂了 (down)、或没在向监视器报告统计信息。

Peering 过程

Peering 的作用　Ceph 用多副本来保证数据可靠性，一般设置为 2 或 3 个副本，每个 pg，通过 peering 来使它的多个副本达到一致的状态。

一些相关的概念

(1) Acting set：负责这个 pg 的所有 osd 的集合

(2) Epoch：osdmap 的版本号，单调递增，osdmap 每变化一次，版本号加 1

(3) Past interval：一个 epoch 序列，在这个序列内，这个 pg 的 acting set 没有变化过

(4) Last epoch start：上一次 peering 完成的 epoch

(5) up_thru：一个 past interval 内，第一次完成 peering 的 epoch

Peering 的触发时机 OSD 启动时，osd 负责的所有 pg，会触发 peering；负责 pg 的 osd 状态发生变化时，会触发 peering。

Peering 的流程 pg 的副本有主从的角色，主负责协调整个 peering 过程，大致流程如下：

(1) 生成 past interval 序列

在每次 osd map 发生变化时，如果本 pg 的 acting set 有变化，则生成新的 past interval，同时记录下上次的 past interval

(2) 选择需要参与 peering 的 osd

(3) 选取权威 osd

(4) Merge 权威 osd 的 log 到 primary

- 将缺失的 log entry merge 到本地的 pg log
- 将 merge 的 log entry 对应的 oid，填充到 missing 结构中

(5) 对比修复各个副本的 pg log

- 确定缺失的 log 区间
- 在稍后的 activate 函数中，将缺失的 log 发送到副本，同时将发送的 log 对应的 oid 填充到 peer_missing 结构中

recovery 过程

recovery 的启动时机 recovery 是对已知的副本不一致或者副本数不足进行修复。以 pg 为单位进行操作。大概有三个启动时机：

- peering 完成后
- scrub 完成后
- 读写操作时，操作的 oid 处于 missing 状态，则先 recovery 这个 object。

OSD 维护了一个 recovery_wq 的线程池，用于执行所有 pg 的 recovery。

peering 后的 recovery peering 在某个 pg 由 degrade 状态重新恢复到 active 状态后启动。通过对比 pg 的每个 osd 的 pglog，得到 osd 缺少的 oid。Primary 缺少的的放到 pg_log 对象的 missing 结构中。Replica 缺少的放到 peer_missing 结构中。Recovery 就是修复 missing 和 peer_missing 中的 oid。

TODO Recovery 流程图

Replica 修复和 Primary 修复流程图

TODO Replica 修复和 Primary 修复流程图

scrub 后的 recovery scrub 依据配置的时间间隔定时启动。通过对比 pg 的每个 osd 的所有 object 信息，得到 osd 缺少的 oid。如果需要修复，则把 primary 缺少 oid 的放到 pg_log 对象的 missing 结构中。Replica 缺少的放到 peer_missing 结构中。

Scrub 后，会判断是否需要 recovery。日常定时启动的 scrub 是不会进行 recovery 的。只有通过 osd command 执行 repair 命令，才会先进行 scrub，完成后进行 recovery。

如果进行 recovery，则流程同上。

这样不太好，应该 scrub 后立即修复。假设 2 副本，其中一副本坏了一块儿盘，scrub 检测到了这些丢失的副本，但没有修复。这些副本如果不被读写，或者很久没有 peering，那这些副本将长期处于丢失状态。万一这段时间内，另外一份副本也出问题了，就会数据丢失。

读写操作前的 Recovery TODO 读写操作前的 Recovery 流程图

参考

librbd-块存储库