

目录

目录	1
一. 为什么我们要以 Ceph 为重点	2
1. Ceph 给 OPV-Suite 带来了什么？	2
2. 性能数据	2
3. 需解决的问题	3
二. Ceph 体系结构	3
三. Ceph 核心技术 -- “无需查表，算算就好”。	5
1. 寻址过程 -- RADIOS	5
2. 数据操作过程 -- RADIOS	5
3. 集群维护	5
4. CRUSH 算法	6
5. 集群管理	7
四. SSD 固态盘应用于 Ceph 集群的典型使用场景	8
1. 作为 OSD 的日志盘	8
2. 与 SATA、SAS 硬盘混用，但独立组成全 SSD 的 Pool	8
3. 配置 CRUSH 数据读写规则，使主备数据中的主数据落在 SSD 的 OSD 上	9
4. 作为 Ceph Cache Tiering 技术中的 Cache 层	9
5. 四种场景的优缺点以及各自的适用场景	9
五. Ceph 代码结构	10
六. RBD 寻址过程	11
1. 命令行参数	11
2. 原有卷的过程分析	11
3. RBD 的特殊点在哪里	12
4. OPV-Suite 目前使用的方式	12
5. Librbd 的实现方式	12
6. Librbd 代码	13
七. Ceph Cache Tiering 技术分解	14
1. 原理分析	14
2. 操作步骤	15
3. 一些建议	16
4. 性能到底怎么样？	16
5. 代码逻辑分析	16
5. 可单独拿出来吗 -- 单机版可用	19
6. 可扩展功能 -- 预读机制？	19

一. 为什么我们要以 Ceph 为重点

1. Ceph 给 OPV-Suite 带来了什么？

A) 相对稳定的分布式块存储解决方案。

由于是分布式的，所以由本地存储即可组成共享存储。

更进而提供 HA, Live Migration, FT 等高级操作。

B) 相对廉价的高可用性存储解决方案

相对于动辄上十万的共享存储设备，软件式的多冗余方案便宜的多。

C) 如果分布的合理，Ceph 可提供类似于本地存储的高性能

--> 虚拟机优先跑在存储着自己 image 的 osd 上

--> 需要深入研究 ceph

D) Ceph 分级存储，缓冲机制，预读机制只要利用的合适就会更加发挥 SSD 的高 IOPS，高吞吐的高性能，同时提供 SAS 的大容量特性。

--> 可认为这个一直是我们在追求的方案。

2. 性能数据

以下是 3.2.4 时本地磁盘性能数据，ST95005620AS 就是普通的 SATA 硬盘，BIWIN E9801 是 PCIE 的闪存卡，INTEL SSDSC2BA40 是 SSD 固态硬盘，KVMax 可认为是 SSD 作为 SATA 的缓冲盘。

d) 对 24 个全 Clone 的 VM，通过同时 dd 命令测试其读写速率（平均值）及并发性

	ST95005620AS	BIWIN E9801	INTEL SSDSC2BA40	KVMax (writeback)	KVMax (writethrough)
读 (MB/sec)	1.9	105.2	12.5	11.5	11.8
写 (MB/sec)	5.6	168.7	38	37.4	5.2

SSD 以及普通的 SATA 盘的 IOPS 到底是多少呢？

IOPS 一般跟块的大小有关系，一般 4k 的随机读写操作：

类别	SSD	SATA	SAS
随机读	100k	100 左右	400 左右
随机写	100k	100 左右	400 左右

从数据上看，分级存储或者缓冲机制是我们最理想的存储方案 -- 发挥 SSD 的高 IOPS，高吞吐的高性能，同时提供 SAS 的大容量特性。

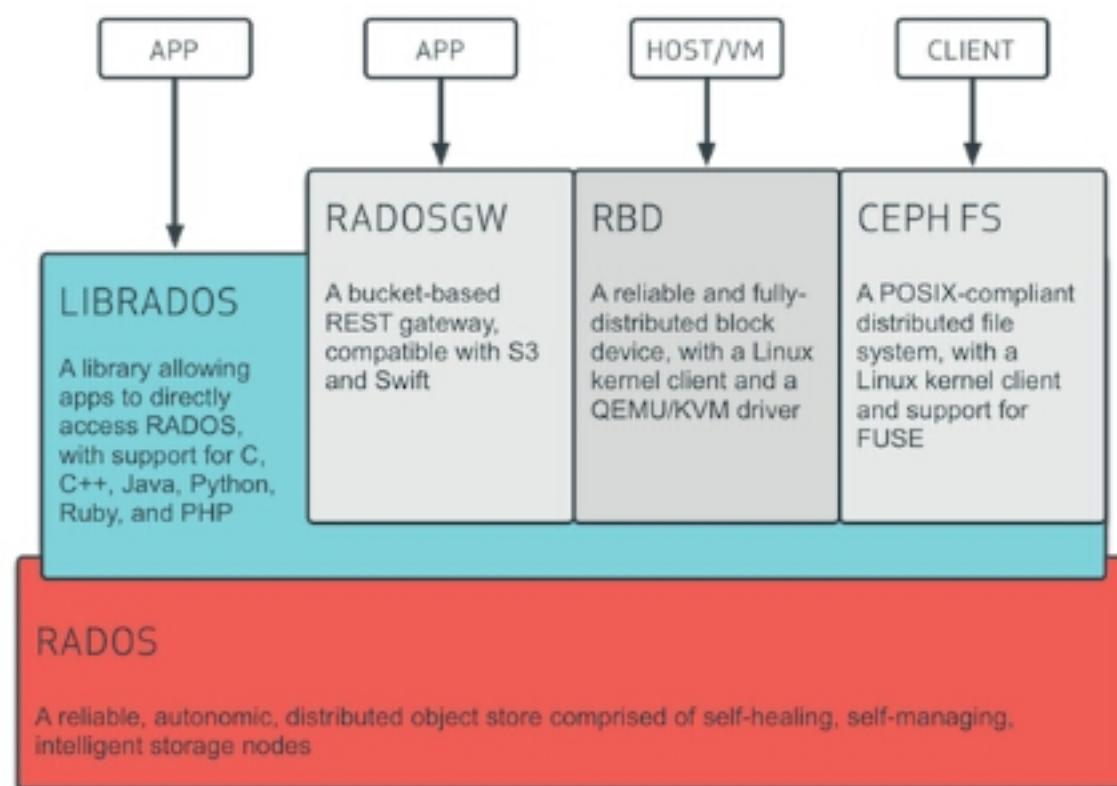
3. 需解决的问题

基于以上考虑，3.2.7 重点应该放在 ceph 的研究上来。

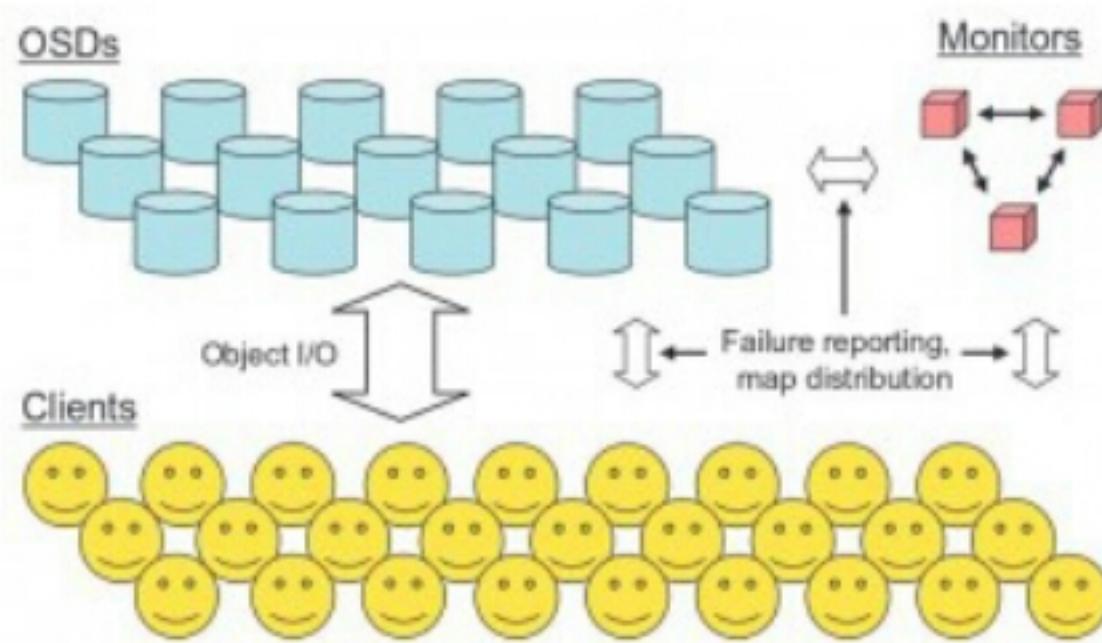
那么单机版可用 ceph 吗？可用 ceph 的机制做缓冲吗？ceph 有预读机制吗？

二. Ceph 体系结构

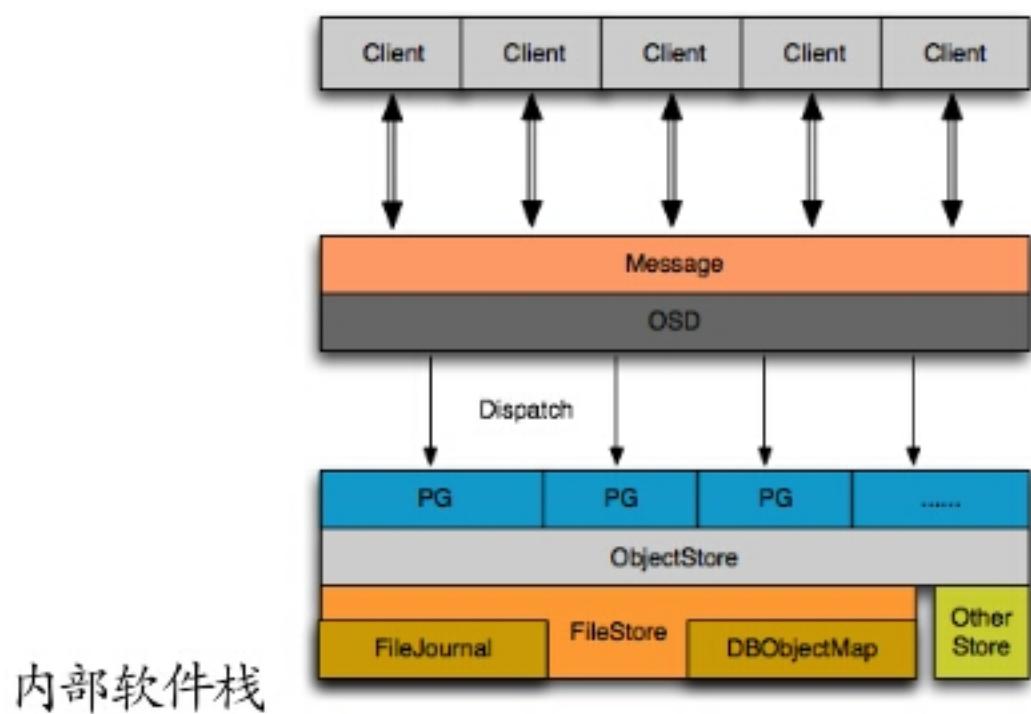
Ceph 是一种为优秀的性能、可靠性和可扩展性而设计的统一的（同时提供对象存储、块存储和文件系统存储三种功能）、分布式的存储系统（高可靠性；高度自动化；高可扩展性）。



RADOS 的系统逻辑结构如下图所示：



- 1) OSD 和 monitor 之间相互传输节点状态信息，共同得出系统的 cluster map
- 2) 客户端程序通过与 OSD 或者 monitor 的交互获取 cluster map，然后直接在本地进行计算，得出对象的存储位置后，便直接与对应的 OSD 通信，完成数据的各种操作。

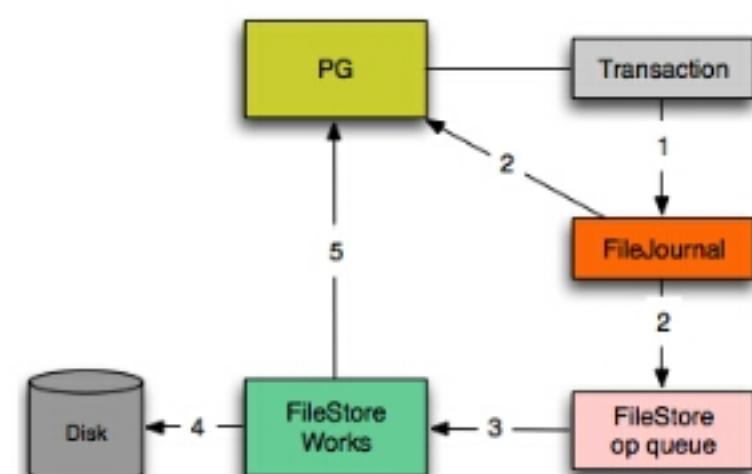


1) ObjectStore/FileStore

- i. 所有 IO 请求在 Client 端发出，在 Message 层统一解析后会被 OSD 层分发到各个 PG 中，OSD 每个 PG 都拥有一个队列
- ii. 一个线程处理队列中的 IO 请求
- i. 调用 ObjectStore 的事务 API (只采用了 Serializable 级别,保证读写的顺序性)
- ii. 目前 ObjectStore 的主要实现是 FileStore，每个 Object 在 FileStore 层会被看成是一个文件

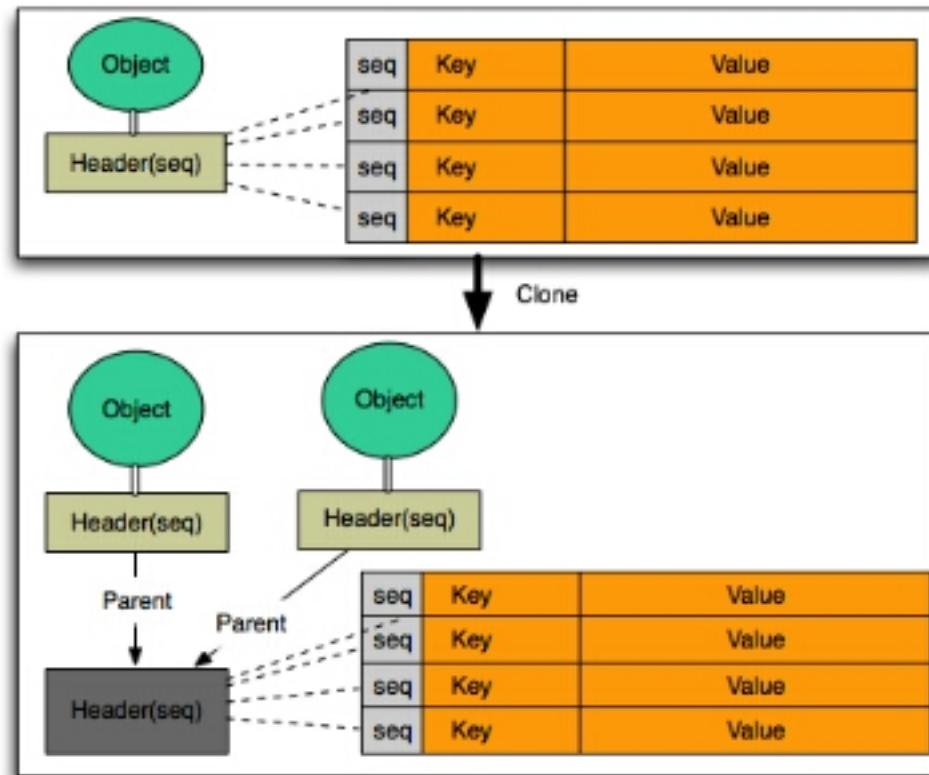
3) FileJournal

为了缩小写事务的处理时间，提高写事务的处理能力并且实现事务的原子性，FileStore 引入了 FileJournal



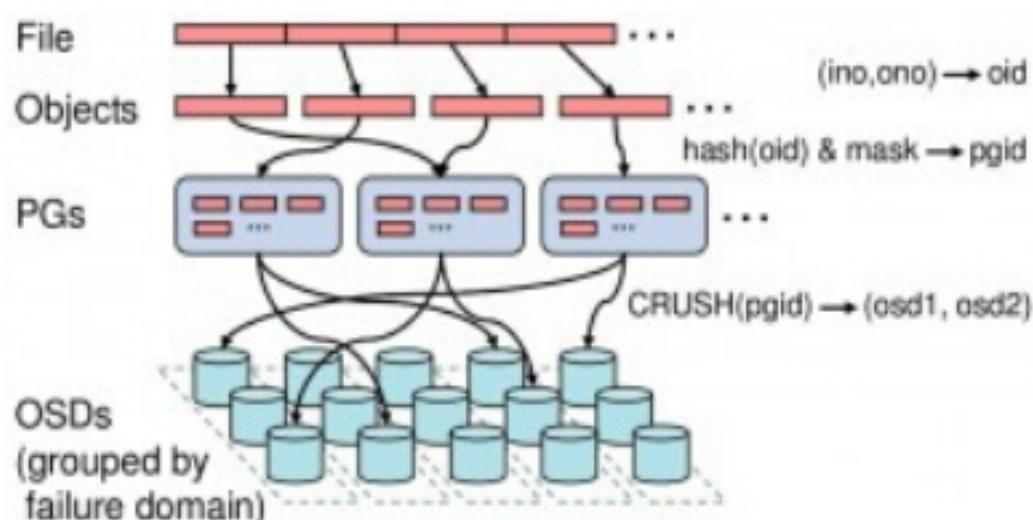
4) DBObjectMap

利用 KeyValue 数据库维护 Object 的属性和 omap



三. Ceph 核心技术 -- “无需查表，算算就好”。

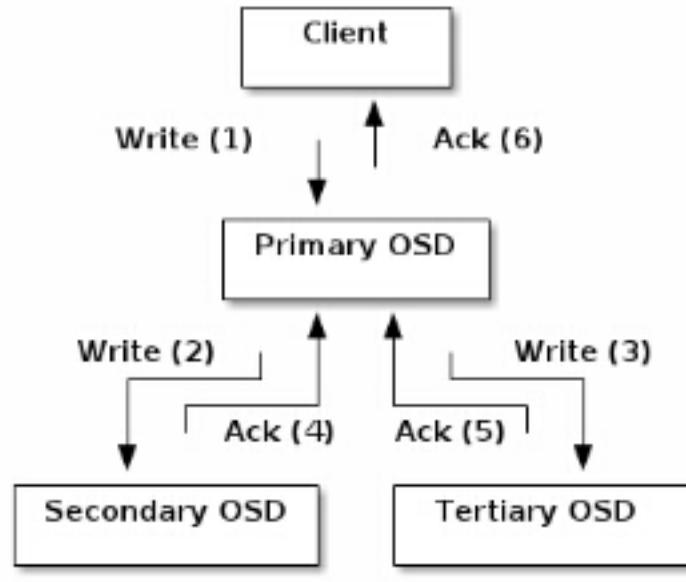
1. 寻址过程 -- RADIOS



PG (Placement Group) —— 顾名思义，PG 的用途是对 object 的存储进行组织和位置映射。Object 近似随机的分配到不同的 PG 上。

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

2. 数据操作过程 -- RADIOS



安全第一， 默认是都写完才算完：

3. 集群维护

由 monitor 集群负责整个 Ceph 集群中所有 OSD 状态的记录，并形成 cluster map。

版本号: epoch
各个 OSD 的网络地址
OSD 的状态: (up, down), (in, out)
CRUSH 算法信息: cluster hierarchy, placement rules

- A) Cluster map 以增量方式在各个 OSD 之间传递
- B) OSD 状态变化触发 cluster map 版本和内容的变化
 - a) 在任意时刻，cluster map 信息在任意一个 PG 内部必须一致
 - b) 在全局范围内 cluster map 可以不一致，但在有限时间内收敛
- C) Cluster map 的变化将触发数据的维护操作
 - a) Replication
 - b) Recovery
- D) 集群维护自动完成，无需人工干预

4. CRUSH 算法

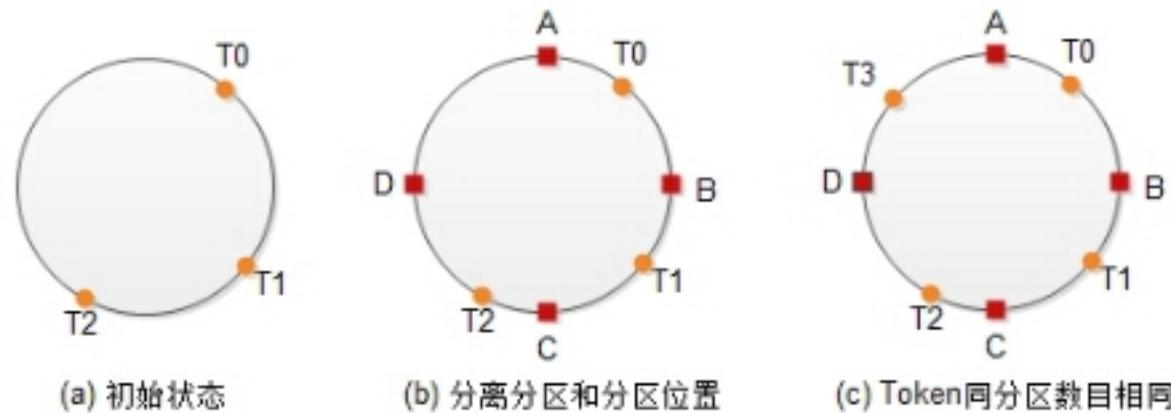
i. CRUSH 算法是干什么的

CRUSH 算法的目的是，为给定的 PG(即分区)分配一组存储数据的 OSD 节点。

$$(\text{osd}0, \text{osd}1, \text{osd}2 \dots \text{osd}n) = \text{CRUSH}(x)$$

ii. 最简单的原理

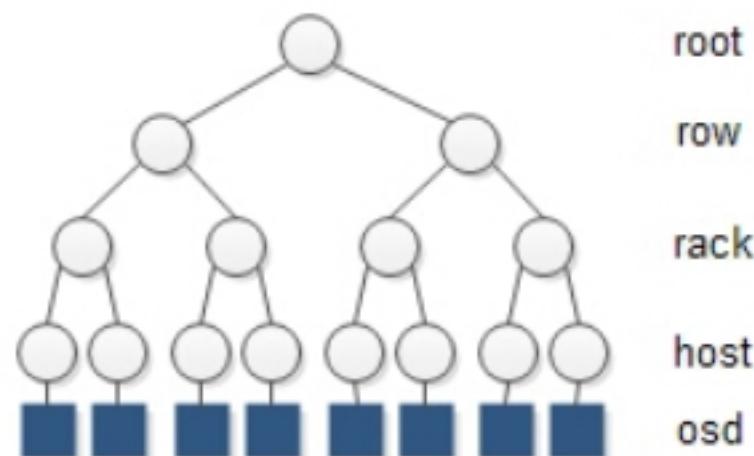
最核心的问题就是把 PG 分配到不同的 Node 上：参考一致性 Hash 算法，就是把分区（也就是 PG）放到不同的 Node 上。最简单的做法就是把分区（比如 PG A）顺时针放入 Node（比如 T0）上，有任何 Node 加入或者退出时仅影响某些分区（PG）-- 比如 T0 死掉的话，只需要把 PG A 从 T0 上转移到 T1 上即可。



iii. 往复杂进阶 -- 多副本位置

在多副本（假设副本数目为 N）的情况下，分区的数据会存储到连续或者非连续的 N 个 Node 中。

考虑因素：1) Node 或者说 OSD 的容量；2) 多个 OSD 最好分布在不同的故障域中。



CRUSH 通过重复执行 Take(bucketID) 和 Select(n, bucketType) 两个操作选取副本位置。

Take(bucketID)指定从给定的 bucketID 中选取副本位置，例如可以指定从某台机架上选取副本位置，以实现将不同的副本隔离在不同的故障域；

Select(n, bucketType)则在给定的 Bucket 下选取 n 个类型为 bucketType 的 Bucket，它选取 Bucket 主要考虑层级结构中节点的容量，以及当节点离线或者加入时的数据迁移量。

2) XXXX

5. 集群管理

A. OSD 管理

OSDMap 是 Ceph 集群中所有 OSD 的信息，所有 OSD 节点的改变如进程退出，节点的加入和退出或者节点权重的变化都会反映到这张 Map 上。

OSDMap 在 OSD, Monitor, Client 间相互传递，通过 epoch 标识其版本。

OSD 在启动时：OSD 会向 Monitor 申请加入 → IN 状态 → Monitor 会发起一个 Proposal 将 IN 变为 UP → 达成一致后，OSD 开始与其他 OSD 通信，并接受 Monitor 分配的 PG。

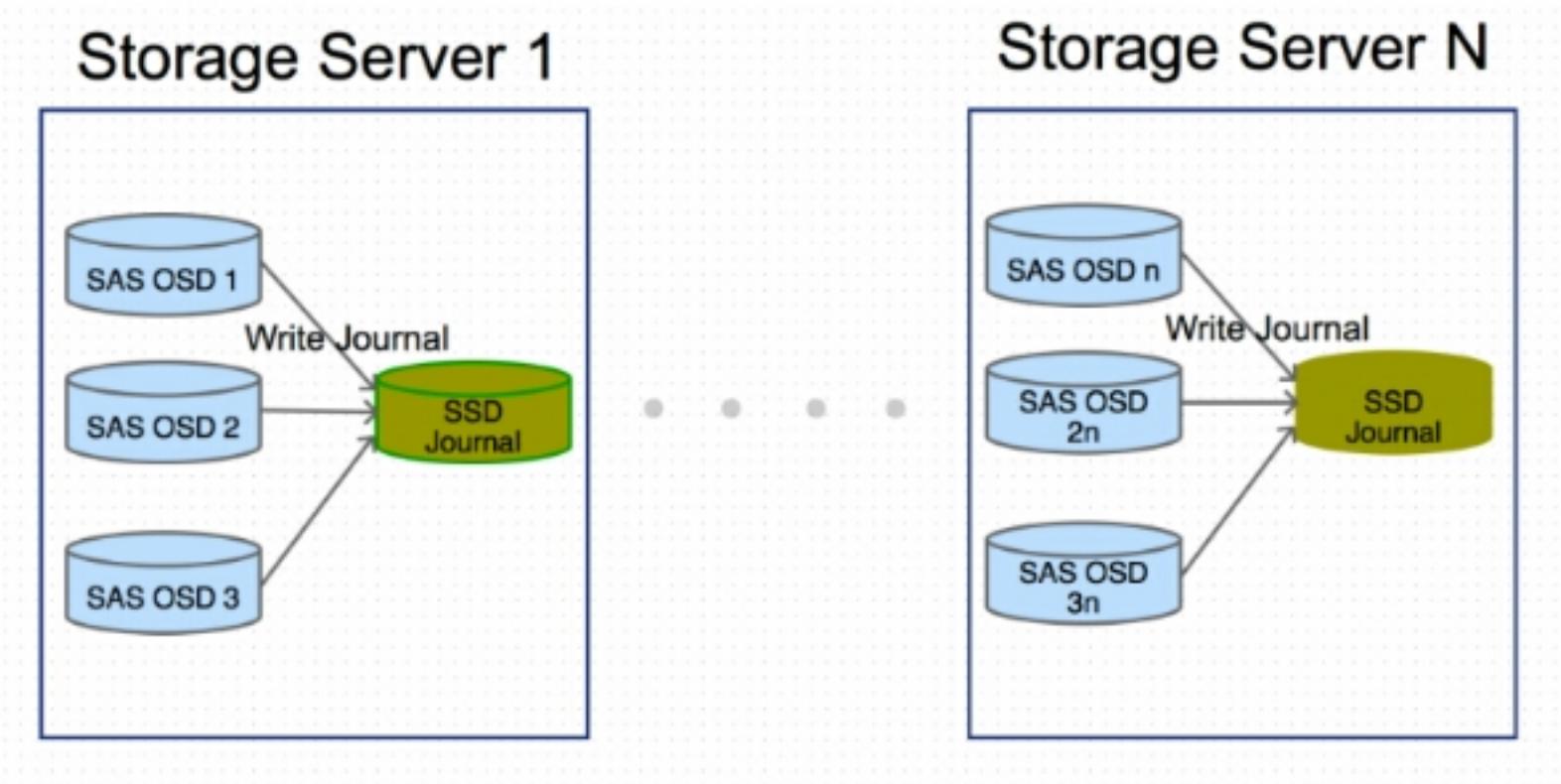
OSD 意外 crash 时：Heartbeat 连接该 OSD 的其他 OSD → 临时 OUT 状态 → 该 OSD 上的 Primary PG 都会将 Primary 角色交给其他 OSD。

B. PG 管理

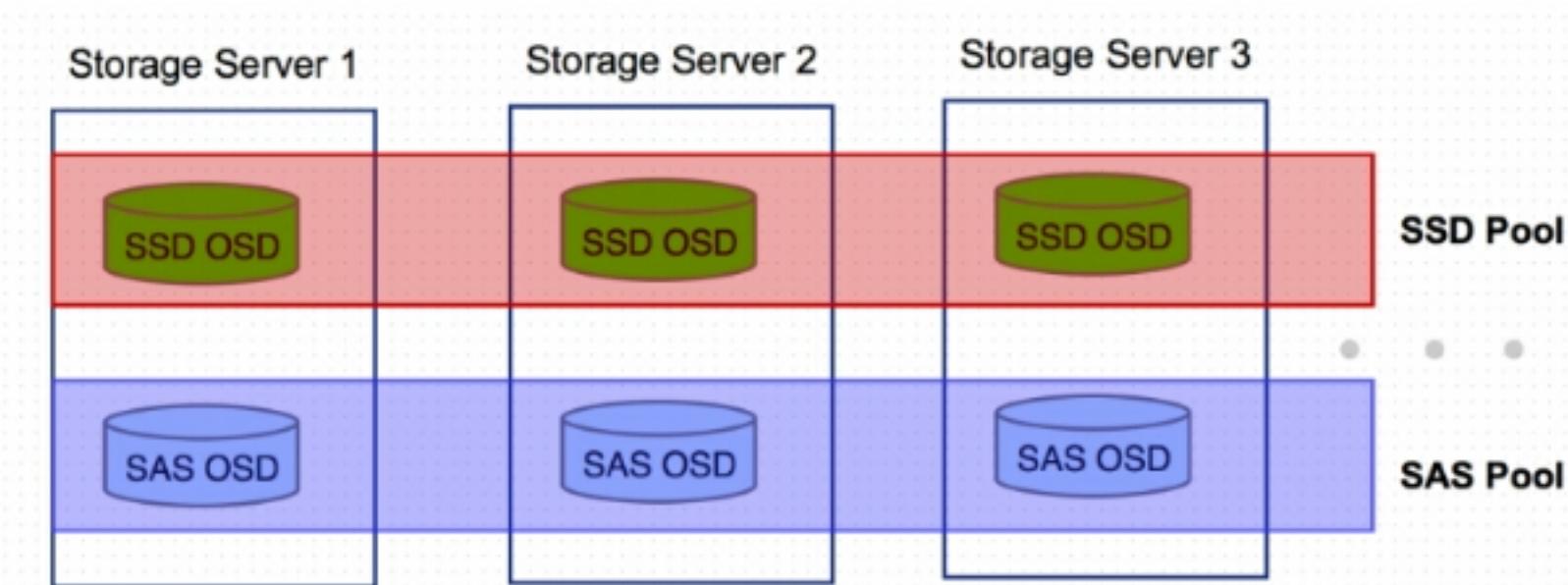
OSD 发生故障 → Monitor 会将该 OSD 上的所有角色为 Primary 的 PG 的 Replicated 角色的 OSD 提升为 Primary PG → PG 都会处于 Degraded 状态 → OSD 被踢出集群 → 这些 PG 会被 Monitor 分配到新的 OSD 上

四. SSD 固态盘应用于 Ceph 集群的典型使用场景

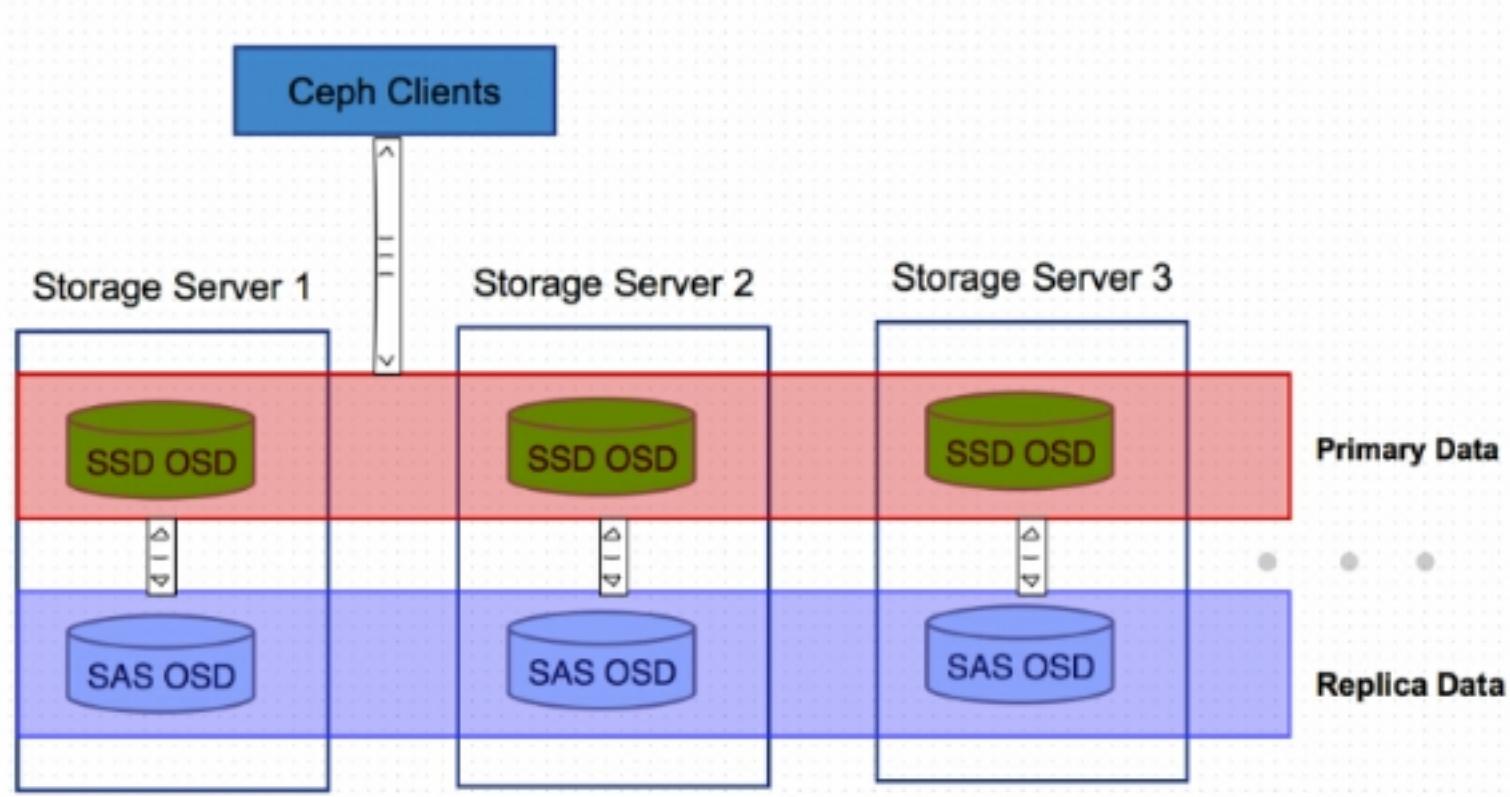
1. 作为 OSD 的日志盘



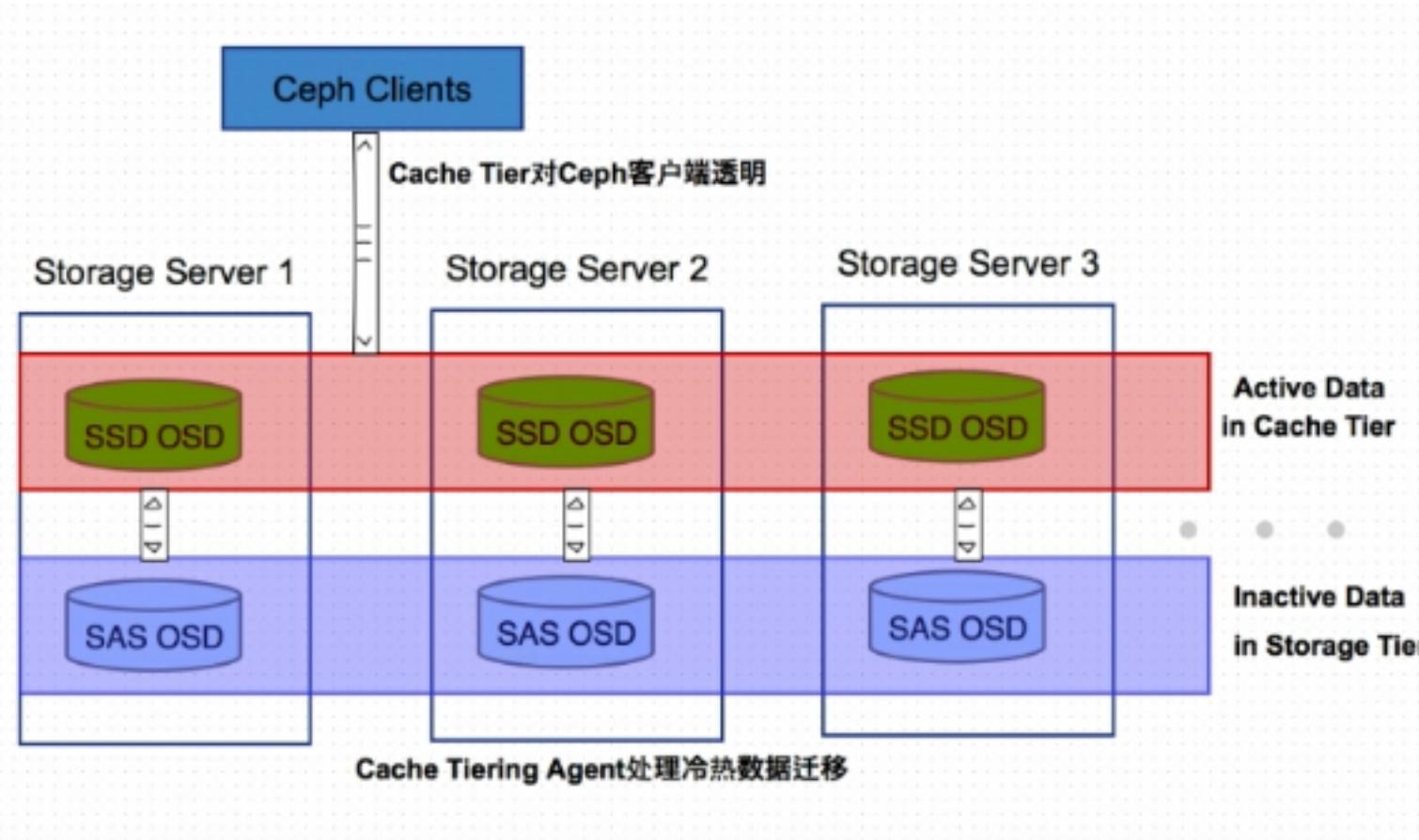
2. 与 SATA、SAS 硬盘混用，但独立组成全 SSD 的 Pool



3. 配置 CRUSH 数据读写规则，使主备数据中的主数据落在 SSD 的 OSD 上



4. 作为 Ceph Cache Tiering 技术中的 Cache 层



5. 四种场景的优缺点以及各自的适用场景

方案描述	技术特点	分析
作为 OSD 的日志盘	OSD 日志是大量小数据块、随机 IO 写操作；所以可提高 ceph 整体性能	大材小用， 排除
与 SATA、SAS 硬盘混用，但独立组成全 SSD 的 Pool	性能要求高的用 SSD Pool，普通的用 SAS Pool	业务的重要程度分的比较清楚，并且不会再改变。 简单稳定 。
配置 CRUSH 数据读写规则，使主备数据中的主数据落在 SSD	调整 ceph 参数，主备份落在 SSD OSD 上。所有的 Ceph 客户端直接读写的都是 SSD OSD 节点，既提高了性能又节约了对	应可大幅提高读的性能；SSD、SAS 的部署位置等需要固定，

的 OSD 上	OSD 容量的要求	不灵活。
作为 Ceph Cache Tiering 技术中的 Cache 层	冷热数据分离，用相对快速/昂贵的存储设备如 SSD 盘，组成一个 Pool 来作为 Cache 层，后端用相对慢速/廉价的设备来组建冷数据存储池。	最灵活的方式。性价比最高。

五. Ceph 代码结构

目前最新的代码是 ceph-0.94.1\src，最关键的代码在以下几个目录中
算法：

Cursh：该目录里包含 cursh 算法的代码，是 ceph 的核心
消息通信

Msg：网络通信模块，包括了网络传输的代码，基础模块。

Messages：定义了传输的消息格式。

Mon：包括了 Ceph Monitor 的代码

Mds：元数据服务器相关的代码，只与 Ceph FS 有关系。暂时不需要关心。

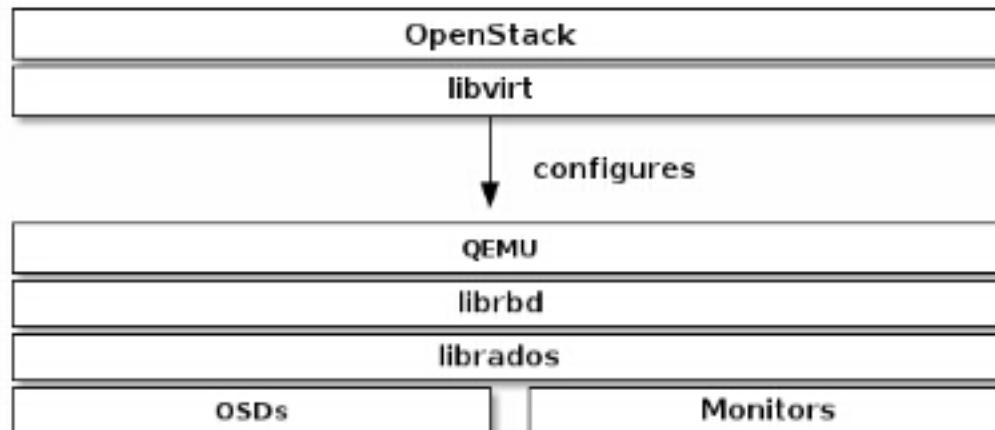
数据服务器：

Os：该目录里包含了 object store 的代码

Osd：该目录包括了 object storage device 的代码

RDB 相关：

Librbd：利用 Rados 提供的 API 实现对卷的管理和操作



RBD Replay is a set of tools for capturing and replaying Rados Block Device (RBD) workloads.

六. RBD 寻址过程

1. 命令行参数

i. Qemu-kvm

Qemu-kvm -m 1024 -drive format=rbd,file=rbd:data/squeeze,cache=writeback

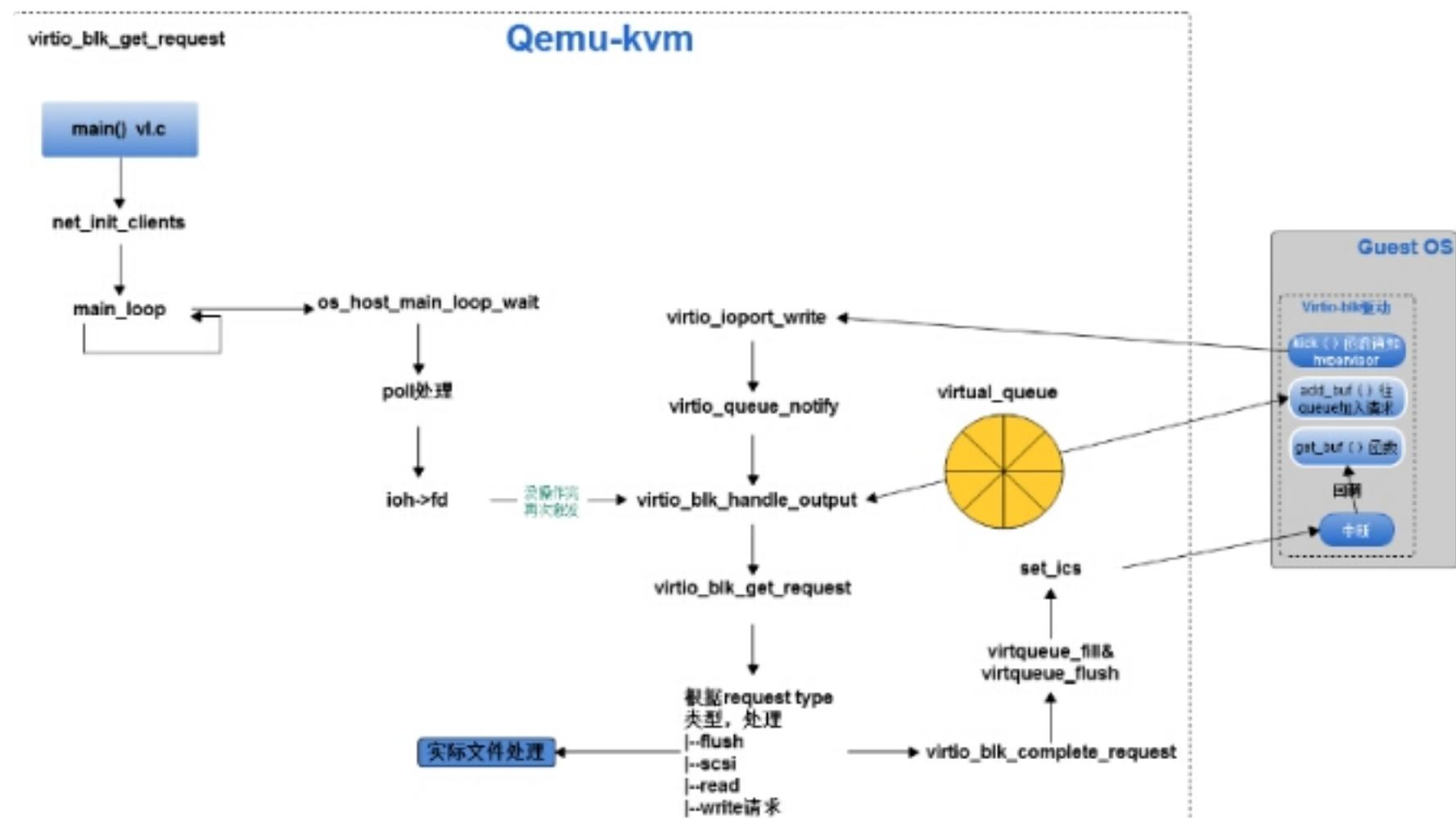
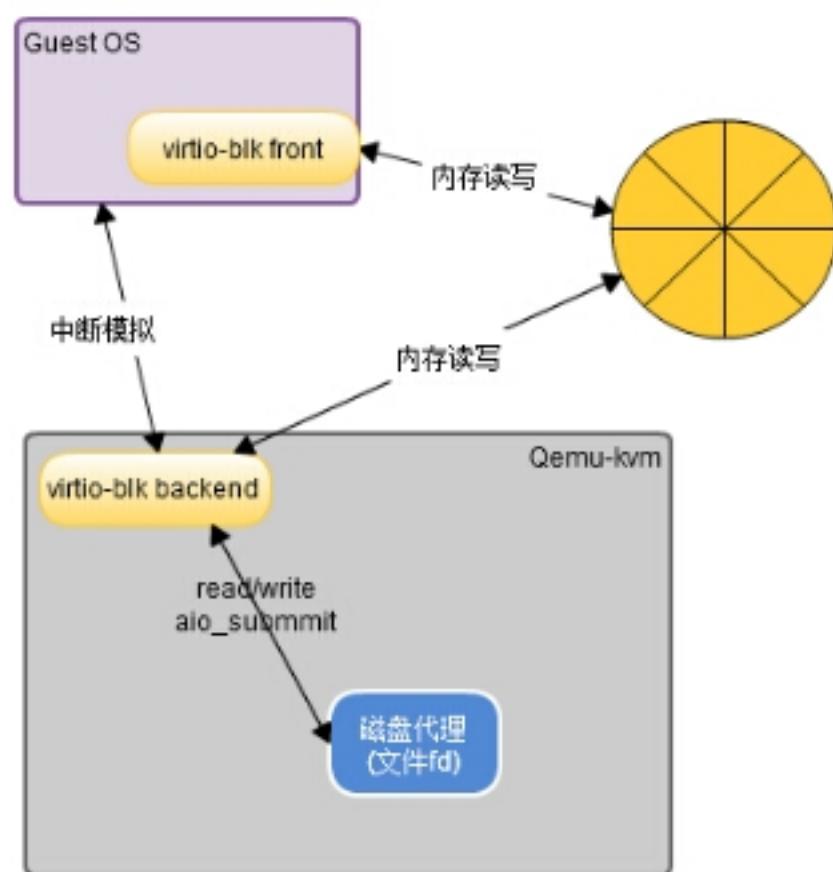
ii. Qemu-img

```
qemu-img create -f raw rbd:data/foo 10G
```

iii. Libvirt

```
<disk type='file' device='disk'>
  <driver name='qemu' type='raw'/>
  <source file='/path/to/image/recent-linux.img' />
  <target dev='vda' bus='virtio' />
  <address type='drive' controller='0' bus='0' unit='0' />
</disk>
```

2. 原有卷的过程分析



3. RBD 的特殊点在哪里

没什么特殊的地方，它就是直接调用 librbd 的接口。但是它自己带有缓冲，相当于块设备缓冲。

rbd_aio_read()/rbd_aio_write()。

4. OPV-Suite 目前使用的方式

使用 Ceph 的块存储有两种路径，一种是利用 QEMU 走 librbd 路径(上文 a 小节介绍)，另一种是使用 kernel module，走 kernel 的路径。前者主要为虚拟机提供块存储设备，后者主要为 Host 提供块设备支持，两种途径的接口实现不完全相同。就目前来说，前者是目前更稳定。

目前 OPV-Suite 在 Pool, Volume 的操作时用的 librbd 的 python 接口。

RBD image 卷直接用/usr/bin/rbd -p MAP 命令把 ceph 卷 scan 出来本地发现成设备文件，走 kernel module ([rbd.ko](#)) 的路径。

需要我们自己对比下这两者在性能，稳定性上的差异。

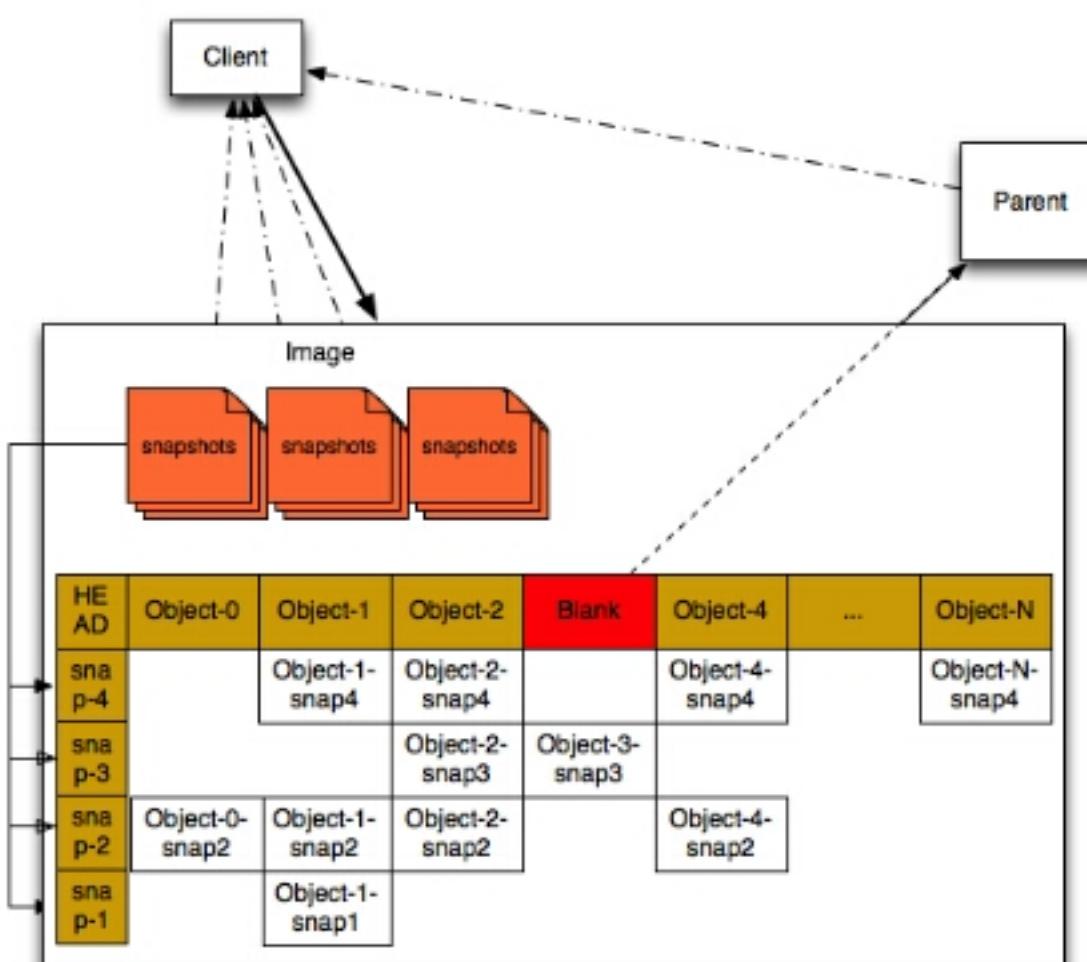
5. Librbd 的实现方式

Librbd 中最基本和常用的概念是 Volume，相当于 LVM 的 Logical Volume，是能被 attach/detach 到 VM 的载体。它也有快照，克隆，快照回滚等概念。

创建卷：使用 Rados API 创建一个 [Header Object](#) (就是 OSD FileObject 里的一个文件)，将这个卷的元数据如 id, size, snaps, name, seq 等信息写入。

写入数据：通常卷被 QEMU 或者 Kernel driver 管理，因此卷元信息会被一直持有，在写入数据时得到数据写入卷的 offset, length，然后根据卷元信息得到这些数据分别归属的 objects，可能会跨越多个。这样，一个 op 请求会转为多个 object op 分别发送到对应的 OSD。<-- 网络操作

读取数据：与写入数据的逻辑类似，读取请求也可能跨越多个 object，或者存在 parent 卷。



6. Librbd 代码

a) 接口

```
rbd_aio_read --> 接口
|-- librbd::aio_read           src\librbd\internal.c
|  |-- readahead <- 预读一部分数据
|  |-- req = new AioRead() <- 封装成 aio 请求
|  |-- ictx->aio_read_from_cache() <- 从 cache 中读取
|  |  |-- ObjectCacher::_readx
|  |  |  |-- o = get Object cache
|  |  |  |-- o->map_read() --> 从缓冲中读取
|  |  |  |-- 激发 cache 读取那些 missing 的数据
|  |-- or
|  |-- req->send()
|  |  |-- 很多 op.read() <- 把请求用 librods 接口通过网络请求 osd
```

```
rbd_aio_write
|-- librbd::aio_write           src\librbd\internal.c
|  |-- write_to_cache
|  |  |-- ObjectCacher::writex()
|  |  |  |-- o = get Object cache
|  |  |  |-- o->map_write() --> 写到缓冲里
|  |  |  |-- mark_dirty(bh) --> 标识这个数据需要刷到 storage 中
|  |  |  |-- try_merge_bh --> merge 多次请求到一次
|  |-- or
|  |-- req->send()
|  |  |-- 很多 op.write() <- 把请求用 librods 接口通过网络请求 osd
```

b) Cache

RBDCache 目前在 Librbd 中主要以 Object 为基本单位进行缓存，一个 RBD 块设备在 Librbd 层会以固定大小分为若干个对象，而读写请求通常会有不同的 IO 大小。

Cache 是如何被激发去读取的呢？

```
ObjectCacher::_readx
|-- o->map_read() 操作后
|-- read missing 发生
|-- bh_read() --> 激发 cache 读取哪些 missing 的数据
|  |-- writeback_handler.read() --> 读取到缓冲中
```

c) cache 什么时候 flush 呢？

```
ObjectCacher 有个 FlusherThread 一直在执行
ObjectCacher::flusher_entry           src\librbd\internal.c
|-- flush(actual - target_dirty)      --> flush some dirty pages
|  |-- bh_write
```

- ```

| | |-- writeback_handler.write() --> 写入到 storage 中

d) ReadHead
 librbd::readahead src\librbd\internal.c
 |--计算 total_bytes_read
 |-- aio_read_from_cache <-- 仅仅从 cache 中读取
 | |-- object_cacher->readx()
 | | |-- o->map_read() --> 从缓冲中读取，激发 cache 读取哪些 missing 的数据
e) Rbd 命令行
 待补充
f) xxx

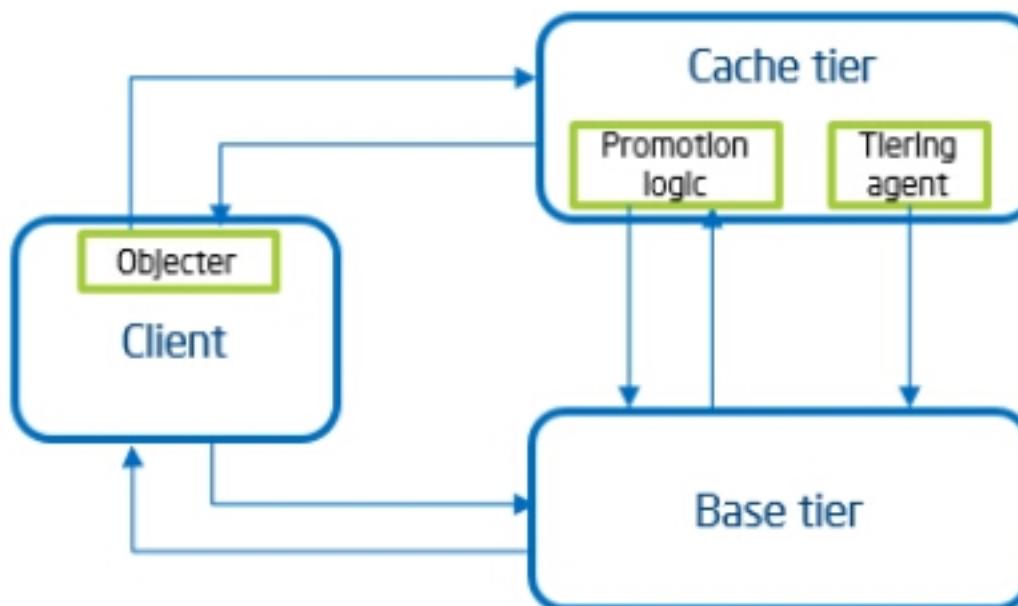
```

## 七. Ceph Cache Tiering 技术分解

### 1. 原理分析

Tier 工作在 rados 层，也就是说不论是块存储、对象存储还是文件存储都可以使用 tier 来提高读写速度。

实际上还是有两个不同属性的 pool (cache-pool, base-pool)，不过是组成了 tier，并且 Ceph 提供接口让设定 cache 策略，在外面看来相当于只有一个 pool，也就是 tier 以后的 base-pool。



Objecter 功能：

决定把 object 放到哪里去，从哪里去取数据

Tiering Agent 功能：

决定数据如何在 cache-pool，具体由两个动作完成的：

Flushing --> 把 cache-pool 上的脏数据刷到 base-pool 上

Evicting --> 把最不经常用的数据从 cache-pool 上剔除

Cache 策略有两种模式：

### 1) Write-back 模式

Ceph 客户端直接往 cache-pool 里写数据，写完立即返回，Tiering Agent 再及时把数据 flush 到 base-pool。

当客户端要读的数据不在 cache-pool 时，Tiering Agent 负责把数据从 base-pool 迁移到 cache-pool。

### 2) read-only 模式

Ceph 客户端在写操作时往 base-pool 上直接写。读数据时先从 cache-pool 上读取，如果不在则从 base-pool 上读取，同时把数据缓冲在 cache-pool。

## 2. 操作步骤

### A) 创建

1. ceph osd tier add base-pool cache-pool
2. ceph osd tier cache-mode cache-pool writeback
3. ceph osd tier set-overlay base-pool cache-pool

### B) 配置

1. ceph osd pool set cache-pool hit\_set\_type bloom
2. ceph osd pool set cache-pool cache\_target\_dirty\_ratio 0.4
3. ceph osd pool set cache-pool cache\_target\_full\_ratio 0.8
4. ceph osd pool set cache-pool cache\_min\_flush\_age 600
5. ceph osd pool set cache-pool cache\_min\_evict\_age 1800

### C) 删除

1. ceph osd tier cache-mode cache-pool forward
2. rados -p cache-pool cache-flush-evict-all
3. ceph osd tier remove-overlay base-pool
4. ceph osd tier remove base-pool cache-pool

用户 (Client) 仅仅跟 base-pool 打交道，可忽视 cache-pool 的存在

### 3. 一些建议

- A) Cache-pool 的 SSD 最好是专用的
- B) Write-back、read-only 模式之间切换的话先 delete tier 再重新配置
- C) Cache Tier 貌似不能横向扩展
  - > 不应该啊，pool 是支持动态自动扩展的
  - > 需要实测下
  - > 如果真没有，看下社区什么时候有--> tier 用户设定的，并告之不可变更
- D) Tier-Pool 暂不支持 snapshot
  - > rbd image 上面的我们利用的 qcow2 去支持即可，不需要用它的 snapshot。  
Link 克隆也用我们自己的。  
Full 克隆类似于 VAAI，如果它的 Full 克隆速度更快，调用它本身的 Full 克隆。

### 4. 性能到底怎么样？

理论分析应该是不错的，但实际呢，需要分析下：四种 SSD 的使用类型到底哪种效率更高。

### 5. 代码逻辑分析

#### A) 参数解析 -- 命令开始的地方

##### a) Add tier

```
OSDMonitor::preprocess_command src\mon\OSDMonitor.c
|-- pool_id = lookup_pg_pool_name() --> 从名字得到 pool id
|-- tierpool_id = lookup_pg_pool_name() --> 从名字得到 pool id
|-- p = get_pg_pool(pool_id) --> 从 id 得到 pool 的数据结构
|-- tp = get_pg_pool(pool_id) --> 从 id 得到 pool 的数据结构
|-- 检查 p 和 tp 的 tier 是否为空：目前仅支持 1 对 1
|-- p->tiers.insert(tierpool_id) <-- 数据结构操作
|-- tp->tier_of = pool_id <-- 数据结构操作
|-- wait_for_finished_proposal() --> 等待 monitor 提交该请求审批
```

##### b) Set cache mode

```
OSDMonitor::preprocess_command src\mon\OSDMonitor.c
|-- pool_id = lookup_pg_pool_name() --> 从名字得到 pool id
|-- p = get_pg_pool(pool_id) --> 从 id 得到 pool 的数据结构
|-- 检查以前设置的 mode 是否支持进阶成当前要设置的 mode
|-- p->cache_mode = mode
|-- wait_for_finished_proposal() --> 等待 monitor 提交该请求审批
```

\* none: No cache-mode defined

\* forward: Forward all reads and writes to base pool

- \* writeback: Cache writes, promote reads from base pool
- \* readonly: Forward writes to base pool
- \* readforward: Writes are in writeback mode, Reads are in forward mode
- \* readproxy: Writes are in writeback mode, Reads are in proxy mode

### c) Set overlay

```
OSDMonitor::preprocess_command src\mon\OSDMonitor.c
|-- 参数解析 p 和 overlaypool_id
|-- p->read_tier = overlaypool_id
|-- p->write_tier = overlaypool_id
|-- wait_for_finished_proposal --> 等待 monitor 提交该请求审批
```

## B) Objecter

```
Objecter::op_submit --> Client 操作时会执行 Objecter::op_submit 操作
|-- _op_submit_with_budget
| |-- _op_submit
| | |-- _calc_target <-- 检查命令应该发给谁
| | | |-- if (is_read && pi->has_read_tier())
| | | |-- t->target_oloc.pool = pi->read_tier <-- 计算 target, 有 tier 就发给
tier
| | | |-- if (is_write && pi->has_write_tier())
| | | |-- t->target_oloc.pool = pi->write_tier <-- 计算 target, 有 tier 就发给
tier
| | |-- 其他操作
```

## C) OSD 的操作

OSD 接到请求，WorkQueue 得到响应，调用\_process 处理具体请求，其中会 handle cache

```
OSD::ShardedOpWQ::_process
|-- osd->dequeue_op()
| |-- pg->do_request()
| | |-- switch (op->get_req()->get_type()){
| | | |-- case CEPH_MSG OSD_OP:
| | | | |-- do_op()
| | | | |-- agent_choose_mode()
| | | | |-- maybe_handle_cache()
| | | | | |-- switch (pool.info.cache_mode)
| | | | | | |-- case pg_pool_t::CACHEMODE_WRITEBACK:
| | | | | | | |-- if(cache full){
| | | | | | | | |-- if(read)
| | | | | | | | |-- do_proxy_read
| | | | | | | | |-- else
| | | | | | | | |-- waiting_for_cache_not_full
| | | | | }else{
```

```

| | | | |-- if(write)
| | | | |-- promote_object --> waiting_for_active.push_back
| | | | |-- else
| | | | |-- do_proxy_read
| | | |-- execute_ctx()
| | | |-- prepare_transaction
| | | | |-- do_osd_ops
| | | | | |-- case CEPH_OSD_OP_CACHE_FLUSH: <-- 也有可接受
flush 指令
| | | | | |-- if(oi.is_dirty())
| | | | |-- start_flush()

```

## D) Tiering Agent

### a) 如何启动的？

在 PG (Pool) 启动(on\_activate)时，会调用 agent\_setup

ReplicatedPG::agent\_setup src\osd\ReplicatedPG.c

```

|-- agent_state.reset()
|-- agent_choose_mode() <-- 设置 model
| |-- agent_state->flush_mode = flush_mode
| |-- agent_state->evict_mode = evict_mode
| |-- agent_state->evict_effort = evict_effort

```

### b) Agent 如何运行

OSDService 会一直运行 pg->agent\_work

ReplicatedPG::agent\_work

```

|-- agent_maybe_flush
|-- agent_maybe_evict
|-- agent_choose_mode --> 重新设置 Mode

```

### c) Flushing

agent\_maybe\_flush

|-- 计算 ob\_local\_mtime

|-- if(ob\_local\_mtime + age > now) --> skip (too young)

|-- if 正在 flushing --> skip (flushing)

|-- start\_flush()

| |-- base\_oloc.pool = pool.info.tier\_of <-- 操作对象设成 base\_pool

| |-- osd->objecter->mutate(base\_pool) --> 把请求对象发给 base\_pool

### d) Evicting

agent\_maybe\_evict

|-- if(obc->obs.oi.is\_dirty()) --> skip (dirty)

|-- if (atime\_upper + agent\_state->evict\_effort <= 1000000) --> skip

|-- \_delete\_oid() --> 就是简单从 pool 上删除它，因为脏数据已经被 skip 掉了

|-- 如果 EVICT\_MODE 设置成了 FULL，除了 dirty 的都删除

```
EVICT_MODE_IDLE, // no need to evict anything
EVICT_MODE_SOME, // evict some things as we are near the target
EVICT_MODE_FULL, // evict anything
```

## 5. 可单独拿出来吗 -- 单机版可用

不好拿，可行的是借鉴它的思想和部分代码，结合 flashcache + librbd 部分的 readhead。看看能否找到一种性能高效的存储方式。

## 6. 可扩展功能 -- 预读机制？

Tiering 部分没有，librbd 部分有。