[**理解 OpenStack + Ceph （2）：Ceph 的物理和逻辑结构 [Ceph Architecture]**](http://www.cnblogs.com/sammyliu/p/4836014.html)

 本系列文章会深入研究 Ceph 以及 Ceph 和 OpenStack 的集成：

（1）[安装和部署](http://www.cnblogs.com/sammyliu/p/4804037.html%20)

（2）[Ceph RBD 接口和工具](http://www.cnblogs.com/sammyliu/p/4838139.html%20)

（3）[Ceph 物理和逻辑结构](http://www.cnblogs.com/sammyliu/p/4836014.html%20)

（4）[Ceph 的基础数据结构](http://www.cnblogs.com/sammyliu/p/4843812.html%20)

（5）[Ceph 与 OpenStack 集成的实现](http://www.cnblogs.com/sammyliu/p/4838138.html)

（6）[QEMU-KVM 和 Ceph RBD 的 缓存机制总结](http://www.cnblogs.com/sammyliu/p/5066895.html)

（7）[Ceph 的基本操作和常见故障排除方法](http://www.cnblogs.com/sammyliu/p/5555218.html%20)

**1. Ceph 集群的物理结构**

**1.1 Ceph 内部集群**

从[前一篇文章](http://www.cnblogs.com/sammyliu/p/4804037.html) 我们知道，从物理上来讲，一个 Ceph 集群内部其实有几个子集群存在：

（1）MON（Montior）集群：MON 集群有由少量的、数目为奇数个的 Monitor 守护进程（Daemon）组成，它们负责通过维护 Ceph Cluster map 的一个主拷贝（master copy of Cluster map）来维护整 Ceph 集群的全局状态。理论上来讲，一个 MON 就可以完成这个任务，之所以需要一个多个守护进程组成的集群的原因是保证高可靠性。每个 Ceph node 上最多只能有一个 Monitor Daemon。

root@ceph1:~# ps -ef | grep ceph-mon

root 964 1 0 Sep18 ? 00:36:33 /usr/bin/ceph-mon --cluster=ceph -i ceph1 -f

实际上，除了维护 Cluster map 以外，MON 还承担一些别的任务，比如用户校验、日志等。详细的配置可以参考 [MON 配置](http://docs.ceph.com/docs/master/rados/configuration/mon-config-ref/)。

（2）OSD （Object Storage Device）集群：OSD 集群由一定数目的（从几十个到几万个） OSD Daemon 组成，负责数据存储和复制，向 Ceph client 提供存储资源。每个 OSD 守护进程监视它自己的状态，以及别的 OSD 的状态，并且报告给 Monitor；而且，OSD 进程负责在数据盘上的文件读写操作；它还负责数据拷贝和恢复。在一个服务器上，一个数据盘有一个 OSD Daemon。

root@ceph1:~# ps -ef | grep ceph-osd

root 1204 1 0 Sep18 ? 00:24:39 /usr/bin/ceph-osd --cluster=ceph -i 3 -f

root 2254 1 0 Sep18 ? 00:20:52 /usr/bin/ceph-osd --cluster=ceph -i 6 -f

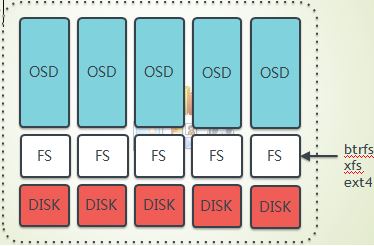
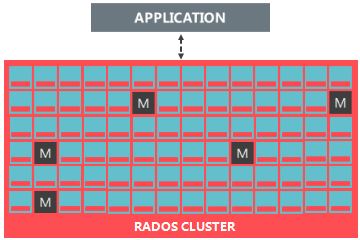
（3）若干个数据盘：一个Ceph 存储节点上可以有一个或者多个数据盘，每个数据盘上部署有特定的文件系统，比如 xfs，ext4 或者 btrfs，由一个 OSD Daemon 负责照顾其状态以及向其读写数据。

Disk /dev/vda: 21.5 GB, 21474836480 bytes

/dev/vda1 1 41943039 20971519+ ee GPT

Disk /dev/vdb: 32.2 GB, 32212254720 bytes

/dev/vdb1 1 62914559 31457279+ ee GPT

    （MON 和 OSD 可以共同在一个节点上，也可以分开）

    关于Ceph 支持的数据盘上的 xfs、ext4 和 btrfs 文件系统，它们都是[日志文件系统](http://en.wikipedia.org/wiki/Journaling_file_system)（其特点是文件系统将没提交的数据变化保存到日志文件，以便在系统崩溃或者掉电时恢复数据），三者各有优势和劣势：

* btrfs （B-tree 文件系统） 是个很新的文件系统（Oracel 在2014年8月发布第一个稳定版），它将会支持许多非常高大上的功能，比如 透明压缩（ transparent compression）、可写的COW 快照（writable copy-on-write snapshots）、去重（deduplication ）和加密（encryption ）。因此，Ceph 建议用户在非关键应用上使用该文件系统。 更多的参考包括 [（1）](https://www.ibm.com/developerworks/cn/linux/l-cn-btrfs/)[（2）](https://zh.wikipedia.org/wiki/Btrfs)[（3）](https://oss.oracle.com/projects/btrfs/)。
* xfs 和 btrfs 相比较ext3/4而言，在高伸缩性数据存储方面具有优势。
* Ceph 的[这篇文章](http://docs.ceph.com/docs/master/rados/configuration/filesystem-recommendations/) 明确推荐在生产环境中使用 XFS，在开发、测试、非关键应用上使用 btrfs。
* 网上有很多的文章比较这几种文件系统，包括：
  + [ext3，ext4，xfs和btrfs文件系统性能对比](http://www.cnblogs.com/tommyli/p/3201047.html)
  + [固态硬盘上 Ext4 和 xfs 性能比较](http://www.oschina.net/translate/ext4-vs-xfs-on-ssd)
  + [哇，让你的DB再快一倍：ext4 vs xfs对比测试](http://imysql.com/2010/08/30/make_your_db_faster_ext4_vs_xfs.html)
  + [XFS --if it's more robust, why are we using ext4 instead?](http://arstechnica.com/civis/viewtopic.php?t=1169535)

（4）要使用 CephFS，还需要 MDS 集群，用于保存 CephFS 的元数据

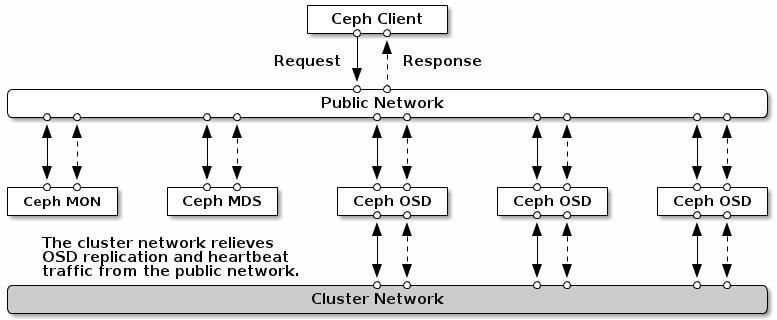
（5）要使用对象存储接口，还需要 RADOS Gateway， 它对外提供REST接口，兼容S3和Swift的API。

**1.2 Ceph 网络结构**

    Ceph 使用以太网连接内部各存储节点以及连接 client 和集群。Ceph 推荐使用两个网络：

* 前端（北向）网络（ a public (front-side) network）连接客户端和集群
* 后端/东西向网络 （a cluster (back-side) network）来连接 Ceph 各存储节点

下图（[来源](http://docs.ceph.com/docs/master/rados/configuration/network-config-ref/)）显示了这种网络拓扑结构：



这么做，主要是从性能（OSD 节点之间会有大量的数据拷贝操作）和安全性（两网分离）考虑。你可以在 Ceph 配置文件的 [global] 部分配置两个网络：

public network = {public-network/netmask}

cluster network = {cluster-network/netmask}

具体可以参考：

* [Deploying Ceph with High Performance Networks, Architectures and benchmarks for Block Storage Solutions](http://www.mellanox.com/related-docs/whitepapers/WP_Deploying_Ceph_over_High_Performance_Networks.pdf)
* [Deploying Ceph with High Performance Networks](http://www.snia.org/sites/default/files/JohnKim_CephWithHighPerformanceNetworks_V2.pdf)
* [CHAPTER 2. NETWORKING RECOMMENDATIONS](https://access.redhat.com/documentation/en/red-hat-ceph-storage/version-1.2.3/red-hat-ceph-storage-123-hardware-guide/chapter-2-networking-recommendations)
* [Ceph with a cluster and public network on IPv6](http://blog.widodh.nl/2014/11/ceph-with-a-cluster-and-public-network-on-ipv6/) 谈到了 IPV6 的支持。

**1.3 RDB Cache （缓存）**

**1.3.1 常见的 Write Cache 种类**

|  |  |  |  |
| --- | --- | --- | --- |
| 缓存种类 | 说明 | 优劣势 | 适合场景 |
| Write-through（直写） | 这种缓存方式在写 I/O 时把数据放入缓存，同时直接写入底层的持久存储，然后再向主机确认写入操作完成。 | 安全地保存数据，从缓存读，减少了读操作的延迟，但是写操作 的延迟没得到优化 | 适合于写较少，但是频繁度的应用 |
| Write-back （回写） | 数据直接写入缓存，然后向主机返回写入完成。 | 对频繁写应用减少了写的延迟，但是有数据丢失风险 | 对读写混合型应用有优势，但是需要考虑数据保护 |
|  |  |  |  |

缓存的通常位置分类：

* 服务器（主机）上：RAID 卡或者 HBA 卡上做缓存。
* VMM 内：在 Hypervisor 上做缓存。
* 客户机操作系统内：以 Windows 2012 为例，它提供 write-back 缓存机制。

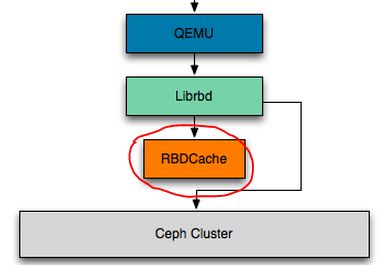
更多资料，可以参考 [Cache is vital for application deployment, but which one to choose](http://www.computerweekly.com/feature/Write-through-write-around-write-back-Cache-explained)。

**1.3.2 Ceph RBD 缓存**

默认情况下，Ceph RBD 是不使用缓存的，读和写直接到 Ceph 集群中的存储，写只有在所有 replica 上写都完成后才给客户端返回写完成。Ceph 在较新的版本上陆续添加了 RBD 缓存支持：

* 从 0.46 版本开始，Ceph 支持 write-back 缓存，你可以在 ceph.conf 文件的 [client] 部分添加 rbd cache = true 来使得 write-back 缓存生效。这时候，写几乎是立即返回，但是数据只有在被 flushed 后才写入到实际存储。
* 从 0.47 版本开始，Ceph 支持 write-through 缓存机制。你只需要再添加配置项 rbd cache max dirty = 0 即可。
* 从 0.60 版本开始，Ceph 支持 rbd cache writethrough until flush 配置项。设置它为 true 时，会使得 write-through 机制变得更加安全，因为老的客户机操作系统（2.6.32 内核版本之前）可能不支持 flush 操作。因此，在设置了该配置项为 true 时，即使用户设置了使用 write-through 机制，Ceph 也会自动使用 write-back 机制，直到它收到第一个 flush 指令后才真正使用 write-through。

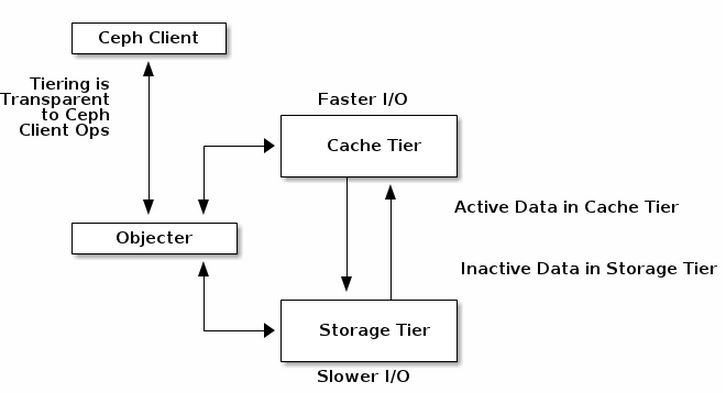
可见 RBD 缓存是在客户端做的，见如下图示：



更多信息，可以参考我的另一篇文章：[QEMU-KVM 和 Ceph RBD 的 缓存机制总结](http://www.cnblogs.com/sammyliu/p/5066895.html)

**1.3.3 Cache tiering （缓存分层）**

    Ceph 还支持在集群段做缓存分层。其原理是，在较快的磁盘比如 SSD 上建立一个 cache pool，在建立存储池（storage pool）和它之间的 cache 关系，设置一定的缓存策略，实现类似于在客户端缓存同样的效果。



更多的信息及详细配置，参见 [CACHE TIERING](http://docs.ceph.com/docs/master/rados/operations/cache-tiering/)  和 [Intel 的文章](https://software.intel.com/en-us/blogs/2015/03/03/ceph-cache-tiering-introduction)。

**1.3.4 RBD Cache 和 Cache Tiering 的区别**

  从上面的分析可以看出来，两者的区别在于缓存的位置不同：

* Cache tiering 是 RADOS 层在 OSD 端进行数据缓存，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
* RBD Cache是 rbd 层在客户端的缓存，也就是只支持块存储。

  Rbd cache是 客户端的缓存，当多个客户端使用同个块设备时，存在**客户端数据不一致**的问题。举个例子，用户A向块设备写入数据后，数据停留在客户自己的缓存中，没有立即刷新到磁盘，所以其它用户读取不到A写入的数据。但是tier不存在这个问题，因为所有用户的数据都直接写入到 ssd，用户读取数据也是在ssd中读取的，所以不存在客户端数据不一致问题。

  一般地，Tier 使用 SSD 做缓存，而 Rbd cache 只能使用内存做缓存。SSD和内存有两个方面的差别，一个是读写速度、另一个是掉电保护。掉电后内存中的数据就丢失了，而ssd中的数据不会丢失。

**2. Ceph 集群的逻辑结构（以RBD为例）**

 Ceph 集群的逻辑结构由 Pool 和 PG （Placement Group）来定义。

**2.1 Pool**

一个 Pool 是 Ceph 中的一些对象的逻辑分组，它并不表示一个连续的分区，而只是一个逻辑概念，类似于将二进制数据打了tag一样然后根据tag归类一样。它类似于 LVM 中的 Volume Group，类似于一个命名空间。RBD Image 类似于 LVM 中的 Logical Volume。RBD Image 必须且只能在一个 Pool 中。Pool 由若干个PG组成。其属性包括：

* 所有性和访问权限
* 对象副本数目
* PG 数目
* CRUSH 规则集合

Ceph Pool 有两种类型：

1. Replicated pool：拷贝型 pool，通过生成对象的多份拷贝来确保在部分 OSD 丢失的情况下数据不丢失。这种类型的 pool 需要更多的裸存储空间，但是它支持所有的 pool 操作。
2. Erasure-coded pool：纠错码型 pool（类似于 Software RAID）。在这种 pool 中，每个数据对象都被存放在 K+M 个数据块中：对象被分成 K 个数据块和 M 个编码块；pool 的大小被定义成 K+M 块，每个块存储在一个 OSD 中；块的顺序号作为 object 的属性保存在对象中。可见，这种 pool 用更少的空间实现存储，即节约空间；纠删码实现了高速的计算，但有2个缺点，一个是速度慢，一个是只支持对象的部分操作（比如：不支持局部写）。[这篇文章](http://blog.csdn.net/niuanxins/article/details/42239431) 详细介绍了其原理和细节。

Pool 提供如下的能力：

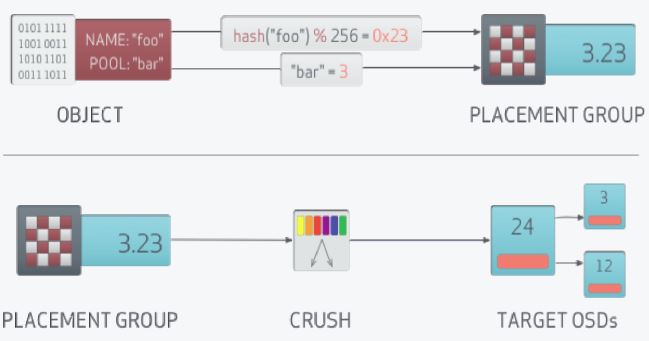
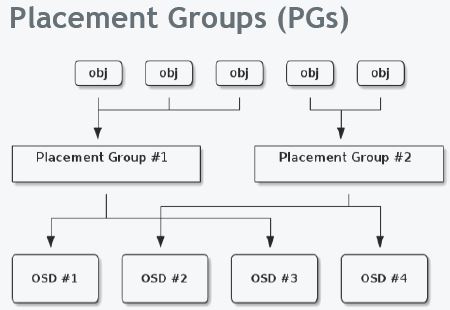
1. Resilience（弹力）：即在确保数据不丢失的情况允许一定的 OSD 失败，这个数目取决于对象的拷贝（copy/replica）份数。对拷贝型 pool 来说，Ceph 中默认的拷贝份数是2，这意味着除了对象自身外，它还有一个另外的备份。你可以自己决定一个 Pool 中的对象的拷贝份数。
2. Placement Groups（放置组）：Ceph 使用 PG 来组织对象，这是因为对象可能成千上万，因此一个一个对象来组织的成本是非常高的。PG 的值会影响 Ceph 集群的行为和数据的持久性。你可以设置 pool 的 PG 数目。推荐的配置是，每个 OSD 大概 100 个 PG。
3. CRUSH Rules （CRUSH 规则）：数据映射的策略。系统默认提供 “replicated\_ruleset"。用户可以自定义策略来灵活地设置 object 存放的区域。比如可以指定 pool1中所有objecst放置在机架1上，所有objects的第1个副本放置在机架1上的服务器A上，第2个副本分布在机架1上的服务器B上。 pool2中所有的object分布在机架2、3、4上，所有Object的第1个副本分布在机架2的服务器上，第2个副本分布在机架3的服 器上，第3个副本分布在机架4的服务器上。详细的信息可以参考这些文档 [（1）](http://docs.openfans.org/ceph/ceph4e2d658765876863/ceph-1/ceph-storage-cluster3010ceph5b5850a896c67fa43011/operations301064cd4f5c3011/crush-maps3010crush66205c043011)[（2）](http://dachary.org/?p=3189)[（3）](http://docs.ceph.com/docs/master/rados/operations/crush-map/)[（4）](http://wenku.baidu.com/view/4d246317a5e9856a5612604e.html)。
4. Snapshots（快照）：你可以对 pool 做快照。
5. Set Ownership：设置 pool 的 owner 的用户 ID。
6. Ceph 集群创建后，默认创建了 data，metadata 和 rbd 三个存储池。

**2.2 （Placement Group）PG**

**2.2.1 概念**

 PG 概念非常复杂，主要有如下几点：

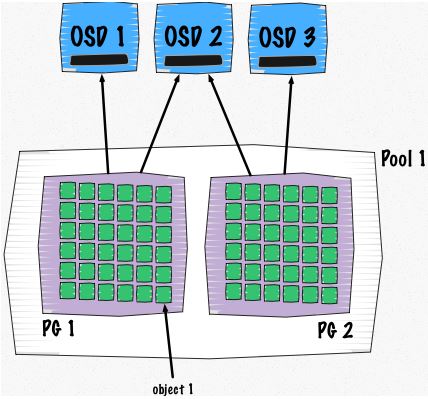
* PG 也是对象的逻辑集合。同一个PG 中的所有对象在相同的 OSD 上被复制。
* PG 聚合一部分对象成为一个组（group），这个组被放在某些OSD上（place），合起来就是 Placemeng Group （放置组）了。
* Epoch：PG map 的版本号，它是一个单调递增的序列。
* Peering：见下文的状态（8）描述。详细过程请参阅 [Ceph：pg peering过程分析](https://www.ustack.com/blog/ceph%EF%BC%8Dpg-peering/)。
* Acting set：支持一个 PG 的所有 OSD 的有序列表，其中第一个 OSD 是主OSD，其余为次。acting set 是 CRUSH 算法分配的，但是不一定已经生效了。
* Up set：某一个 PG map 历史版本的 acting set。在大多数情况下，acting set 和 up set 是一致的，除非出现了 pg\_temp。
* Current Interval or Past Interval：若干个连续的版本号，这些版本中 acting 和 up set 保持不变。
* PG temp：在Ceph 正在往主 OSD 回填数据时，这个主OSD是不能提供数据服务的，这时候，它会向 MON 申请一个临时的 acting set，这就是 PG temp。举个例子，现在 acting set 是[0,1,2]，出现了一点事情后，它变为 [3,1,2]，此时 osd.3 还是空的因此它无法提供数据服务因此它还需要等待backfilling过程结束，因此，它会向 MON 申请一个临时的 set 比如 [1,2,3]，此时将由 osd.1 提供数据服务。回填过程结束后，该临时 set 会被丢弃，重新由 osd.3 提供服务。
* 主 （primary） OSD：在 acting set 中的首个 OSD，负责接收客户端写入数据；默认情况下，提供数据读服务，但是该行为可以被修改。它还负责 peering 过程，以及在需要的时候申请 PG temp。
* 次 （replica）OSD：在 acting set 中的除了第一个以外的其余 OSD。
* 流浪 （stray） OSD：已经不是 acting set 中了，但是还没有被告知去删除数据 的 OSD。
* PG 的 acting set 是由 CRUSH 算法根据 CRUSH Rules 动态地计算得出的。

**2.2.2 特点**

其主要特点如下：

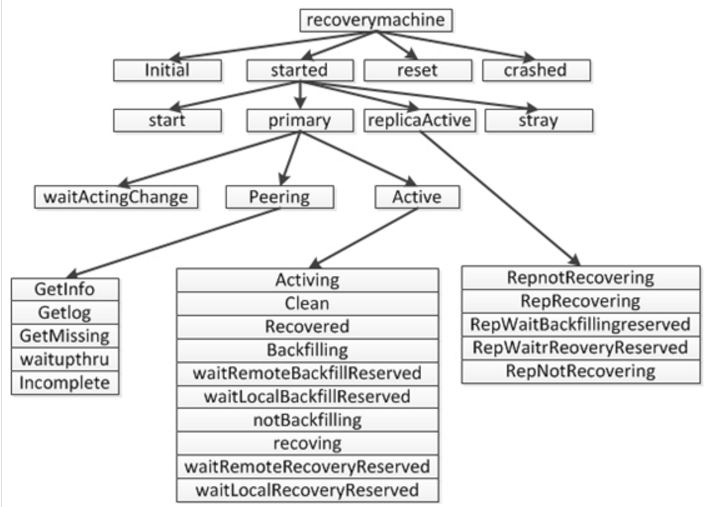
* 基本特点
  + PG 确定了 pool 中的对象和 OSD 之间的映射关系。一个 object 只会存在于一个 PG 中，但是多个 object 可以在同一个 PG 内。
  + Pool 的 PG 数目是创建 pool 时候指定的，Ceph 官方有推荐的计算方法。其值与 OSD 的总数的关系密切。当Ceph 集群扩展 OSD 增多时，根据需要，可以增加 pool 的 PG 数目。
  + 对象的副本数目，也就是被拷贝的次数，是在创建 Pool 时指定的。该分数决定了每个 PG 会在几个 OSD 上保存对象。如果一个拷贝型 Pool 的size（拷贝份数）为 2，它会包含指定数目的 PG，每个 PG 使用两个 OSD，其中，第一个为主 OSD （primary），其它的为从 OSD （secondary）。不同的 PG 可能会共享一个 OSD。
  + Ceph 引入 PG 的目的主要是为了减少直接将对象映射到 OSD 的复杂度。
  + PG 也是Ceph 集群做清理（scrubbing）的基本单位，也就是说数据清理是一个一个PG来做的。
  + PG 和 OSD 之间的映射关系由 CRUSH 决定，而它做决定的依据是 CRUSH 规则（rules）。CRUSH 将所有的存储设备（OSD）组织成一个分层结构，该结构能区分故障域（failure domain），该结构中每个节点都是一个 CRUSH bucket。详细情况请阅读 CRUSH 相关的文档。
* PG 和 OSD 的关系是动态的：
  + 一开始在 PG 被创建的时候，MON 根据 CRUSH 算法计算出 PG 所在的 OSD。这是它们之间的初始关系。
  + Ceph 集群中 OSD 的状态是不断变化的，它会在如下状态之间做切换：
    - up：守护进程运行中，能够提供IO服务；
    - down：守护进程不在运行，无法提供IO服务；
    - in：包含数据；
    - out：不包含数据
  + 部分 PG 和 OSD 的关系会随着 OSD 状态的变化而发生变化。
    - 当新的 OSD 被加入集群后，已有OSD上部分PG将可能被挪到新OSD上；此时PG 和 OSD 的关系会发生改变。
    - 当已有的某 OSD down 了并变为 out 后，其上的 PG 会被挪到其它已有的 OSD 上。
    - 但是大部分的 PG 和 OSD 的关系将会保持不变，在状态变化时，Ceph 尽可能只挪动最少的数据。
  + 客户端根据 Cluster map 以及 CRUSH Ruleset 使用 CRUSH 算法查找出某个 PG 所在的 OSD 列表（其实是 up set）。
  + PG-Object-OSD 的关系如下图所示：



* PG 的创建过程（详细过程请参考 [PG 的创建过程](http://my.oschina.net/u/2460844/blog/535007)）：
  1. MON 节点上有PGMonitotor，它发现有 pool 被创建后，判断该 pool 是否有 PG。如果有PG，则一一判断这些 PG 是否已经存在，如果不存在，则开始下面的创建 PG 的过程。
  2. 创建过程的开始，设置PG 状态为 Creating，并将它加入待创建PG队列 creating\_pgs，等待被处理。
  3. 开始处理后，使用 CRUSH 算法根据当前的 OSD map 找出来 up/acting set，加入 PG map 中以这个 set 中 OSD 为索引的队列 creating\_pgs\_by\_osd。（看起来只会加入到主OSD的队列中）。
  4. 队列处理函数将该 OSD 上需要创建的 PG 合并，生成消息MOSDPGCreate，通过消息通道发给 OSD。
  5. OSD 收到消息字为 MSG\_OSD\_PG\_CREATE 的消息，得到消息中待创建的 PG 信息，判断类型，并获取该PG的其它OSD，加入队列 creating\_pgs （似乎是由主 OSD 负责发起创建次 OSD 上的PG），再创建具体的 PG。
  6. PG 被创建出来以后，开始 Peering 过程。
* PG 值的确定：创建 pool 时需要确定其 PG 的数目，在 pool 被创建后也可以调整该数字，该数目会影响到：
  + 数据的持久性：考虑pool 的 size 为 3，表明每个 PG 会将数据存放在 3 个 OSD 上。当一个 OSD down 了后，一定间隔后将开始 recovery 过程，recovery结束前，有部分 PG 的数据将只有两个副本。这时候和需要被恢复的数据的数量有关系，如果该 OSD 上的 PG 过多，则花的时间将越长，风险将越大。如果此时再有一个 OSD down 了，那么将有一部分 PG 的数据只有一个副本，recovery 过程继续。如果再出现第三个 OSD down 了，那么可能会出现部分数据丢失。可见，每个 OSD 上的PG数目不宜过大，否则，会降低数据的持久性。这也就要求在添加 OSD 后，PG 的数目在需要的时候也需要相应增加。
  + 数据的均匀分布性：CRUSH 算法会伪随机地保证 PG 被选中来存放客户端的数据，它还会尽可能地保证所有的 PG 均匀分布在所有的 OSD 上。比方说，有10个OSD，但是只有一个 size 为 3 的 pool，它只有一个 PG，那么10个 OSD 中将只有三个 OSD 被用到。但是 CURSH 算法在计算的时候不会考虑到OSD上已有数据的大小。比方说，100万个4K对象共4G均匀地分布在10个OSD上的1000个PG内，那么每个 OSD 上大概有400M 数据。再加进来一个400M的对象（假设它不会被分割），那么有三块 OSD 上将有 400M + 400M = 800 M 的数据，而其它七块 OSD 上只有 400M 数据。
  + 资源消耗：PG 作为一个逻辑实体，它需要消耗一定的资源，包括内存，CPU 和带宽。太多 PG 的话，则占用资源会过多。
  + 清理时间：Ceph 的清理工作是以 PG 为单位进行的。如果一个 PG 内的数据太多，则其清理时间会很长。

    那如何确定一个 Pool 中有多少 PG？Ceph 不会自己计算，而是给出了一些参考原则，让 Ceph 用户自己计算：

* + 少于 5 个 OSD， 建议设为  128
  + 5 到 10 个 OSD，建议设为 512
  + 10 到 50 个 OSD，建议设为 4096
  + 50 个 OSD 以上，就需要有更多的权衡来确定 PG 数目
  + 你可以使用 [pgcalc](http://ceph.com/pgcalc/) 工具
* PG 的状态也是不断变化的，其主要状态包括：
  + Creating 创建中：PG 正在被创建。
  + Peering 对等互联：表示一个过程，该过程中一个 PG 的所有 OSD 都需要互相通信来就PG 的对象及其元数据的状态达成一致。处于该状态的PG不能响应IO请求。Peering的过程其实就是pg状态从初始状态然后到active+clean的变化过程。一个 OSD 启动之后，上面的pg开始工作，状态为initial，这时进行比对所有osd上的pglog和pg\_info，对pg的所有信息进行同步，选举primary osd和replica osd，peering过程结束，然后把peering的结果交给recovering，由recovering过程进行数据的恢复工作。
  + Active 活动的：Peering 过程完成后，PG 的状态就是 active 的。此状态下，在主次OSD 上的PG 数据都是可用的。
  + Clean 洁净的：此状态下，主次 OSD 都已经被 peered 了，每个副本都就绪了。
  + Down：PG 掉线了，因为存放其某些关键数据（比如 pglog 和 pginfo，它们也是保存在OSD上）的副本 down 了。
  + Degraded 降级的：某个 OSD 被发现停止服务 （down）了后，Ceph MON 将该 OSD 上的所有 PG 的状态设置为 degraded，此时该 OSD 的 peer OSD 会继续提供数据服务。这时会有两种结果：一是它会重新起来（比如重启机器时），需要再经过 peering 过程再到clean 状态，而且 Ceph 会发起 recovery （恢复）过程，使该 OSD 上过期的数据被恢复到最新状态；二是 OSD 的 down 状态持续 300 秒后其状态被设置为 out，Ceph 会选择其它的 OSD 加入 acting set，并启动回填（backfilling）数据到新 OSD 的过程，使 PG 副本数恢复到规定的数目。详情可以参考 [PG 的数据恢复过程](http://my.oschina.net/u/2460844/blog/596895)。
  + Recovering 恢复中：一个 OSD down 后，其上面的 PG 的内容的版本会比其它OSD上的 PG 副本的版本落后。在它重启之后（比如重启机器时），Ceph 会启动 recovery 过程来使其数据得到更新。
  + Backfilling 回填中：一个新 OSD 加入集群后，Ceph 会尝试级将部分其它 OSD 上的 PG 挪到该新 OSD 上，此过程被称为回填。与 recovery 相比，回填（backfill）是在零数据的情况下做全量拷贝，而恢复（recovery）是在已有数据的基础上做增量恢复。
  + Remapped 重映射：每当 PG 的 acting set 改变后，就会发生从旧  acting set 到新 acting set 的数据迁移。此过程结束前，旧 acting set 中的主 OSD 将继续提供服务。一旦该过程结束，Ceph 将使用新 acting set 中的主 OSD 来提供服务。
  + Stale 过期的：OSD 每隔 0.5 秒向 MON 报告其状态。如果因为任何原因，主 OSD 报告状态失败了，或者其它OSD已经报告其主 OSD down 了，Ceph MON 将会将它们的 PG 标记为 stale 状态。
  + PG 的所有的状态是一个类似树形的结构，每个状态可能存在子状态，子状态还可能存在子状态，如下图所示：

（[来源](http://my.oschina.net/u/2460844/blog/537511)）

更多的状态请参考 <http://docs.ceph.com/docs/master/rados/operations/pg-states/>。实际上 PG 的状态可以是以上这些状态的组合，比如：

[复制代码](javascript:void(0);)

[root@ceph-mon ~]# ceph -s

cluster c5476875-2a04-41b7-a4e8-421133c69ac8

health HEALTH\_WARN

28 pgs backfill #回填，有新的 OSD 被加入了？

79 pgs degraded #降级，有 OSD down 了？

10 pgs recovering #恢复中

42 pgs recovery\_wait #等待恢复

80 pgs stuck unclean #有 80个 PG 一直处于 unclean 状态

27 pgs undersized #GP 的副本数小于pool size

recovery 4814/27835 objects degraded (17.295%)

recovery 2047/27835 objects misplaced (7.354%)

[复制代码](javascript:void(0);)

注意，只有当所有的 PG 都是 active + clean 状态时，集群的状态才是 HEALTH\_OK 的。

* 清理 scrubbing：Ceph 以 PG 为单位进行数据清理，以保证数据的完整性，它的作用类似于文件系统的 fsck 工具。
  + 有两种比较方式：（1）light scrubbing：比较对象的size和属性，一般每天进行 （2）deep scrubbing：读取对象的数据，比较检验码，一般每周进行。
  + Ceph 的 OSD 定期启动 scrub 线程来扫描部分对象，通过与其他副本比对来发现是否一致，如果存在不一致，抛出异常提示用户手动解决。管理员也可以手工发起。
  + Scrub 以 PG 为单位，对于每一个PG，Ceph 分析该 PG 下所有的对象, 产生一个类似于元数据信息摘要的数据结构，如对象大小，属性等，叫scrubmap, 比较主与副scrubmap，来保证是不是有object 丢失或者不匹配。
  + Scrub 方式分成两种， classic vs. chunky。Scrub 流程需要提取对象的校验信息然后跟其他副本的校验信息对比，这期间被校验对象的数据是不能被修改的,所以 write 请求会被 block. 由于 PG 可能包含成千上万 objects,   chunk 每一次的比较只取其中一部分 objects 来比较，这样只 block一小部分object的write请求。这是在ceph的Bobtail(v0.56  Jan 1 2013)引入的feature,称为chunky scrub。Classic scrub 没有引入chunk， 会block所有的write请求。
  + 该机制对保证数据的完整性非常重要，但是也会消耗大量的集群资源，block 住一部分对象的写入操作，降低集群的性能，特别是当一个OSD服务器上多个OSD同时进行深度清理的时候。这篇文章 [Ceph Deep-Scrubbing Impact Study](http://blog.simon.leinen.ch/2015/02/ceph-deep-scrubbing-impact.html) 说当有三个深度清理线程发生时，性能有明显的下降。

**2.3 Ceph 结构和状态地图 Cluster map**

    Ceph 要求 ceph 客户端和 OSD 守护进程需要知晓整个集群的拓扑结构，它们可以通过 Monitor 获取 cluster map 来达到这一点。Cluster map 包括：

（1）Monitor Map：MON 集群的状态（包括 the cluster fsid, the position, name address and port of each monitor, 创建时间，最后的更新时间等）。

[复制代码](javascript:void(0);)

root@ceph1:/osd/data# ceph mon dump

dumped monmap epoch 1

epoch 1

fsid 4387471a-ae2b-47c4-b67e-9004860d0fd0

last\_changed 0.000000

created 0.000000

0: 9.115.251.194:6789/0 mon.ceph1

1: 9.115.251.195:6789/0 mon.ceph2

2: 9.115.251.218:6789/0 mon.ceph3

[复制代码](javascript:void(0);)

（2）OSD Map：当前所有 Pool 的状态和所有 OSD 的状态 （包括 the cluster fsid, map 创建和最后修改时间, pool 列表, replica sizes, PG numbers, a list of OSDs and their status (e.g., up, in) 等）。通过运行 ceph osd dump 获取。

[复制代码](javascript:void(0);)

root@ceph1:~# ceph osd dump

epoch 76

fsid 4387471a-ae2b-47c4-b67e-9004860d0fd0

created 2015-09-18 02:16:19.504735

modified 2015-09-21 07:58:55.305221

flags

pool 0 'data' replicated size 3 min\_size 2 crush\_ruleset 0 object\_hash rjenkins pg\_num 64 pgp\_num 64 last\_change 1 flags hashpspool crash\_replay\_interval 45 stripe\_width 0

osd.3 up in weight 1 up\_from 26 up\_thru 64 down\_at 25 last\_clean\_interval [7,23) 9.115.251.194:6801/1204 9.115.251.194:6802/1204 9.115.251.194:6803/1204 9.115.251.194:6804/1204 exists,up d55567da-4e2a-40ca-b7c9-5a30240c895a  
......

[复制代码](javascript:void(0);)

（3）PG Map：包含PG 版本（version）、时间戳、最新的 OSD map epoch, full ratios, and 每个 PG 的详细信息比如 PG ID, Up Set, Acting Set, 状态 (e.g., active + clean), pool 的空间使用统计。可以使用命令 ceph pg dump 来获取 PG Map。

[这里](http://cephnotes.ksperis.com/blog/2015/02/23/get-the-number-of-placement-groups-per-osd) 有段代码可以以表格形式显示这些映射关系：

[复制代码](javascript:void(0);)

ceph pg dump | awk '

/^pg\_stat/ { col=1; while($col!="up") {col++}; col++ }

/^[0-9a-f]+\.[0-9a-f]+/ { match($0,/^[0-9a-f]+/); pool=substr($0, RSTART, RLENGTH); poollist[pool]=0;

up=$col; i=0; RSTART=0; RLENGTH=0; delete osds; while(match(up,/[0-9]+/)>0) { osds[++i]=substr(up,RSTART,RLENGTH); up = substr(up, RSTART+RLENGTH) }

for(i in osds) {array[osds[i],pool]++; osdlist[osds[i]];}

}

END {

printf("\n");

printf("pool :\t"); for (i in poollist) printf("%s\t",i); printf("| SUM \n");

for (i in poollist) printf("--------"); printf("----------------\n");

for (i in osdlist) { printf("osd.%i\t", i); sum=0;

for (j in poollist) { printf("%i\t", array[i,j]); sum+=array[i,j]; poollist[j]+=array[i,j] }; printf("| %i\n",sum) }

for (i in poollist) printf("--------"); printf("----------------\n");

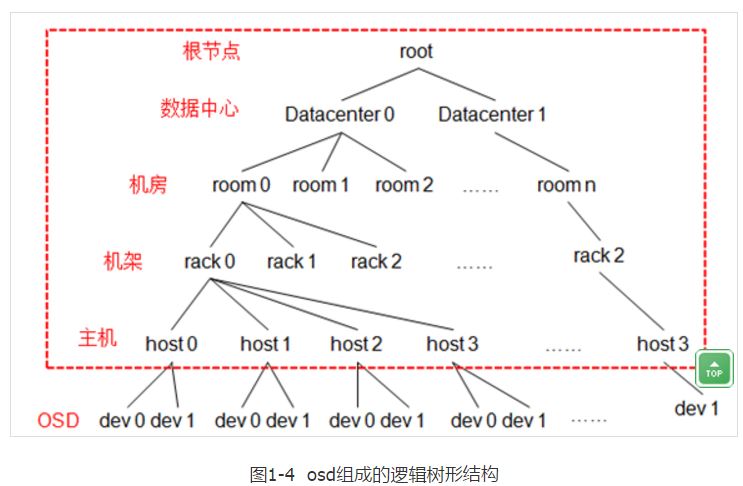
printf("SUM :\t"); for (i in poollist) printf("%s\t",poollist[i]); printf("|\n");

}'

[复制代码](javascript:void(0);)

（4）CRUSH （[Controlled Replication under Scalable Hashing](http://www.ssrc.ucsc.edu/Papers/weil-sc06.pdf)）Map：包含当前磁盘、服务器、机架等层级结构 （Contains a list of storage devices, the failure domain hierarchy (e.g., device, host, rack, row, room, etc.), and rules for traversing the hierarchy when storing data）。 要查看该 map 的话，先运行 ceph osd getcrushmap -o {filename} 命令，然后运行 crushtool -d {comp-crushmap-filename} -o {decomp-crushmap-filename} 命令，在vi 或者 cat {decomp-crushmap-filename} 即可。

CRUSH map 使用分层结构来组织集群中的所有存储设备：



CRUSH rules 主要有三个作用：

* 指定从CRUSH Map 中的哪个节点开始查找
* 指定使用那个节点作为故障隔离域
* 指定定位副本的搜索模式（广度优先 or 深度优先）

例子：

[复制代码](javascript:void(0);)

rule replicated\_ruleset #规则集的命名，创建pool时可以指定rule集

{

ruleset 0 #rules集的编号，顺序编即可

type replicated #定义pool类型为replicated(还有esurecode模式)

min\_size 1 #pool中最小指定的副本数量不能小1

max\_size 10 #pool中最大指定的副本数量不能大于10

step take default #定义PG查找副本的入口点

step chooseleaf firstn 0 type host #选叶子节点、深度优先、隔离host

step emit #结束

}

[复制代码](javascript:void(0);)

PG 选择 OSD 的过程（详情可阅读 [PG选择osd的过程(crush 算法)](http://my.oschina.net/u/2460844/blog/531722)）：

1. 首先要知道在 rules 中指明从 CRUSH map 中哪个节点开始查找，入口点默认为 default 也就是 root 节点
2. 然后隔离域为 host 节点(也就是同一个host下面不能选择两个子节点)。由 default 到3个host的选择过程，这里由default根据节点的bucket类型选择下一个子节点，由子节点再根据本身的类型继续选择，知道选择到host，然后在host下选择一个osd。

因此，Ceph Admin 可以通过配置 CRUSH map 和 rules 来决定数据的存放方式。详细信息，可以参考  [The RADOS object store and Ceph filesystem: Part 2](http://www.admin-magazine.com/HPC/Articles/RADOS-and-Ceph-Part-2)。

（5）MDS Map：包含当前所有 MDS 的状态 （the current MDS map epoch, when the map was created, and the last time it changed. It also contains the pool for storing metadata, a list of metadata servers, and which metadata servers are up and in）。通过执行 ceph mds dump 获取。

[复制代码](javascript:void(0);)

root@ceph1:~# ceph mds dump

dumped mdsmap epoch 13

epoch 13

flags 0

created 2015-09-18 02:16:19.504427

modified 2015-09-18 08:05:55.438558

4171: 9.115.251.194:6800/962 'ceph1' mds.0.2 up:active seq 7

5673: 9.115.251.195:6800/959 'ceph2' mds.-1.0 up:standby seq 1

[复制代码](javascript:void(0);)

MDS 只用于Ceph 文件系统该，与 RDB 和对象存储无关。

**3. CRUSH 算法（以RBD为例）**

**3.1 Ceph client 是如何将数据块放到 OSD 的**

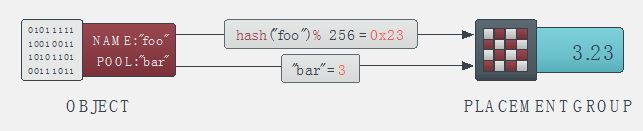
  Ceph 架构中，Ceph 客户端是直接读或者写存放在 OSD上的 RADOS 对象存储中的对象（data object）的，因此，Ceph 需要走完 **(Pool, Object) → (Pool, PG) → OSD set → OSD/Disk** 完整的链路，才能让 ceph client 知道目标数据 object的具体位置在哪里：

（1）创建 Pool 和它的 PG。根据上述的计算过程，PG 在 Pool 被创建后就会被 MON 在根据 CRUSH 算法计算出来的 PG 应该所在若干的 OSD 上被创建出来了。也就是说，在客户端写入对象的时候，PG 已经被创建好了，PG 和 OSD 的映射关系已经是确定了的。

（2）Ceph 客户端通过哈希算法计算出存放 object 的 PG 的 ID：

1. 客户端输入 pool ID 和 object ID （比如 pool = “liverpool” and object-id = “john”）
2. ceph 对 object ID 做哈希
3. ceph 对该 hash 值取 PG 总数的模，得到 PG 编号 （比如 58）（第2和第3步基本保证了一个 pool 的所有 PG 将会被均匀地使用）
4. ceph 对 pool ID 取 hash （比如 “liverpool” = 4）
5. ceph 将  pool ID 和 PG ID 组合在一起（比如 4.58）得到 PG 的完整ID。

也就是：PG-id = hash(pool-id). hash(objet-id) % PG-number

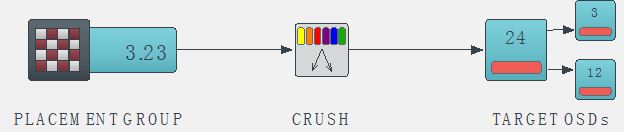


（3）客户端通过 CRUSH 算法计算出（或者说查找出） object 应该会被保存到 PG 中哪个 OSD 上。（注意：这里是说”应该“，而不是”将会“，这是因为 PG 和 OSD 之间的关系是已经确定了的，那客户端需要做的就是需要知道它所选中的这个 PG 到底将会在哪些 OSD 上创建对象。）。这步骤也叫做 CRUSH 查找。

   对 Ceph 客户端来说，只要它获得了 Cluster map，就可以使用 CRUSH 算法计算出某个 object 将要所在的 OSD 的 ID，然后直接与它通信。

1. Ceph client 从 MON 获取最新的 cluster map。
2. Ceph client 根据上面的第（2）步计算出该 object 将要在的 PG 的 ID。
3. Ceph client 再根据 CRUSH 算法计算出 PG 中目标主和次 OSD 的 ID。

也就是：OSD-ids = CURSH(PG-id, cluster-map, cursh-rules)。



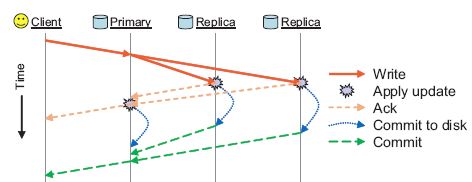
（4）客户端写入数据

在客户端使用 rbd 时一般有两种方法：

* 第一种 是 Kernel rbd。就是创建了rbd设备后，把rbd设备map到内核中，形成一个虚拟的块设备，这时这个块设备同其他通用块设备一样，一般的设备文件为/dev/rbd0，后续直接使用这个块设备文件就可以了，可以把 /dev/rbd0 格式化后 mount 到某个目录，也可以直接作为裸设备使用。这时对rbd设备的操作都通过kernel rbd操作方法进行的。
* 第二种是 librbd 方式。就是创建了rbd设备后，这时可以使用librbd、librados库进行访问管理块设备。这种方式不会map到内核，直接调用librbd提供的接口，可以实现对rbd设备的访问和管理，但是不会在客户端产生块设备文件。

应用写入rbd块设备的过程（详细步骤请参考 [rbd client 端的数据请求处理](http://my.oschina.net/u/2460844/blog/532755)）：

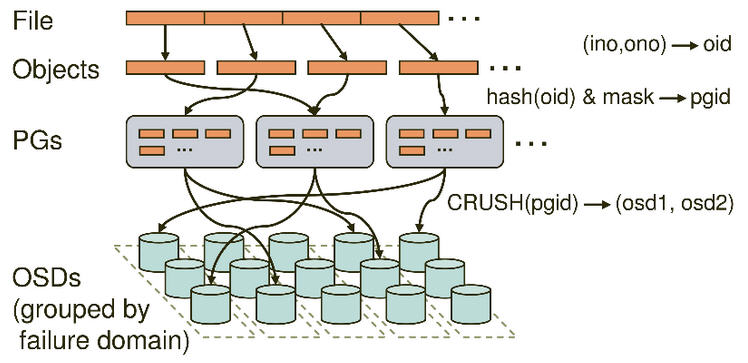
1. 应用调用 librbd 接口或者对linux 内核虚拟块设备写入二进制块。下面以 librbd 为例。
2. librbd 对二进制块进行分块，默认块大小为 4M，每一块都有名字，成为一个对象
3. librbd 调用 librados 将对象写入 Ceph 集群
4. librados 向主 OSD 写入分好块的二进制数据块 (先建立TCP/IP连接，然后发送消息给 OSD，OSD 接收后写入其磁盘)
5. 主 OSD 负责同时向一个或者多个次 OSD 写入副本。注意这里是写到日志（Journal）就返回，因此，使用SSD作为Journal的话，可以提高响应速度，做到服务器端对客户端的快速同步返回写结果（ack）。
6. 当主次OSD都写入完成后，主 OSD 向客户端返回写入成功。
7. 当一段时间（也许得几秒钟）后Journal 中的数据向磁盘写入成功后，Ceph通过事件通知客户端数据写入磁盘成功（commit），此时，客户端可以将写缓存中的数据彻底清除掉了。
8. 默认地，Ceph 客户端会缓存写入的数据直到收到集群的commit通知。如果此阶段内（在写方法返回到收到commit通知之间）OSD 出故障导致数据写入文件系统失败，Ceph 将会允许客户端重做尚未提交的操作（replay）。因此，PG 有个状态叫 replay：“The placement group is waiting for clients to replay operations after an OSD crashed.”。



也就是，文件系统负责文件处理，librdb 负责块处理，librados 负责对象处理，OSD 负责将数据写入在Journal和磁盘中。

关于 RDB 镜像在存储池中是如何被存放的，请阅读 [理解 OpenStack + Ceph （4）：Ceph 的基础数据结构 [Pool, Image, Snapshot, Clone]](http://www.cnblogs.com/sammyliu/p/4843812.html)。

（5）以存放一个文件为例，下图（[来源](https://www.ustack.com/blog/ceph_infra/)）说明了完整的计算过程：



（6）一些说明

几个比例关系：

* 文件 ：对象 = 1 ： n （由客户端实时计算）
* object ：PG = n ： 1 （有客户端使用哈希算法计算）
* PG ：OSD = m ： n （由 MON 根据 CRUSH 算法计算）

CRUSH 算法是相当复杂，快速看看的话可以参考 [官方文章](http://docs.ceph.com/docs/master/architecture/#mapping-pgs-to-osds)，或者直接读代码和作者的论文。几个简单的结论或原则：

* 一个 RBD image（比如虚机的一个镜像文件）会分成几个 data objects 保存在 Ceph 对象存储中。
* 一个 Ceph 集群含有多个 pool （使用 ceph osd pool create 命令创建pool）
* 一个 Pool 包含若干个 PG (在创建 pool 时必须指定 pg\_num，在需要的时候对已有的pool的 pg\_num 也可以进行修改)
* 一个 PG 可以包含多个对象
* 一个 object 只在一个 PG 中
* 一个 PG 映射到一组 OSD，其中第一个 OSD 是主（primary），其余的是从（secondary）
* 许多 PG 可以映射到某个 OSD，通常一个OSD上会有50到100个PG。

使用 [这里](http://cephnotes.ksperis.com/blog/2015/02/23/get-the-number-of-placement-groups-per-osd) 的脚本，可以看出在我的测试环境（pool的副本数为1）中，一共有 6 个pool，7 个 OSD，每个 pool 中有 192 个PG，每个 OSD 大概在164个 （192 \* 6\*1/7）PG 中：

[复制代码](javascript:void(0);)

pool : 4 5 0 1 2 3 | SUM

----------------------------------------------------------------

osd.4 20 19 21 23 21 24 | 128

osd.5 38 28 30 28 34 44 | 202

osd.6 30 33 33 32 34 36 | 198

osd.7 23 19 22 21 22 21 | 128

osd.8 26 36 34 36 30 20 | 182

osd.9 21 26 21 20 21 19 | 128

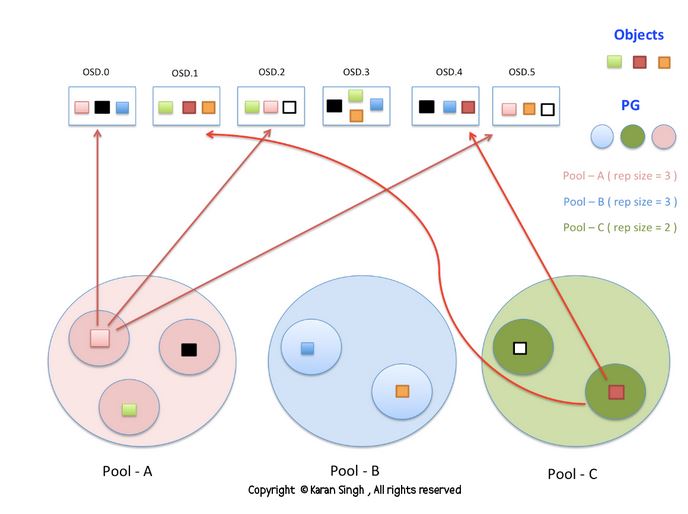
osd.3 34 31 31 32 30 28 | 186

----------------------------------------------------------------

SUM : 192 192 192 192 192 192 |

[复制代码](javascript:void(0);)

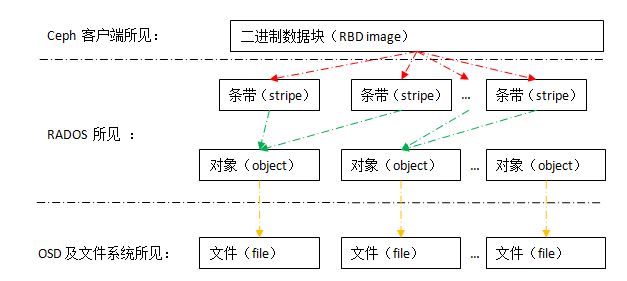
这张图（[来源](http://karan-mj.blogspot.com/2014/01/how-data-is-stored-in-ceph-cluster.html)）也有助于理清其中的关系：



总之，Ceph 采用的是通过计算找到对象应该被保存的 OSD 位置，这比通过常见的查询算法获取位置快得多。CRUSH 算法是的 ceph 客户端自己计算对象要被保存在哪里（哪个 OSD），也使得客户端可以从主 OSD 上保存或者读取数据。

**3.2 RBD  image 保存过程和形式**

如下图所示，Ceph 系统中不同层次的组件/用户所看到的数据的形式是不一样的：



* Ceph 客户端所见的是一个完整的连续的二进制数据块，其大小为创建 RBD image 是设置的大小或者 resize 的大小，客户端可以从头或者从某个位置开始写入二进制数据。
* librados 负责在 RADOS 中创建对象（object），其大小为 pool 的 order 决定，默认情况下 order = 22 此时 object 大小为 4MB；以及负责将客户端传入的二进制块条带化为若干个条带（stripe）。
* librados 控制哪个条带由哪个 OSD 写入（条带 ---写入哪个----> object ----位于哪个 ----> OSD）
* OSD 负责创建在文件系统中创建文件，并将 librados 传入的数据写入数据。

  Ceph client 向一个 RBD image 写入二进制数据（假设 pool 的拷贝份数为 3）：

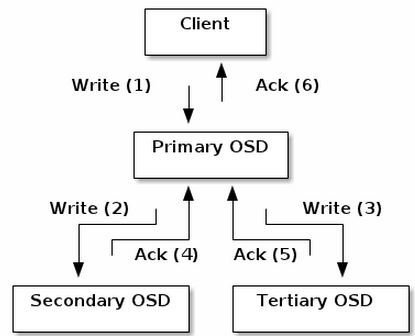
（1）Ceph client 调用 librados 创建一个 RBD image，这时候不会做存储空间分配，而是创建若干元数据对象来保存元数据信息。

（2）Ceph client 调用 librados 开始写数据。librados 计算条带、object 等，然后开始写第一个 stripe 到特定的目标 object。

（3）librados 根据 CRUSH 算法，计算出 object 所对应的主 OSD ID，并将二进制数据发给它。

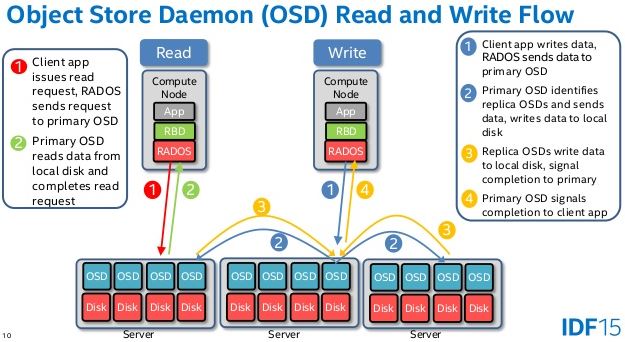
（4）主 OSD 负责调用文件系统接口将二进制数据写入磁盘上的文件（每个 object 对应一个 file，file 的内容是一个或者多个 stripe）。

（5）主 ODS 完成数据写入后，它使用 CRUSH 算啊计算出第二个OSD（secondary OSD）和第三个OSD（tertiary OSD）的位置，然后向这两个 OSD 拷贝对象。都完成后，它向 ceph client 反馈该 object 保存完毕。



（6）然后写第二个条带，直到全部写入完成。全部完成后，librados 还应该会做元数据更新，比如写入新的 size 等。

完整的过程（[来源](https://www.google.com.hk/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0CAUQjhxqFQoTCOfz4ZCQmcgCFcMHLAodD3sDdw&url=%68%74%74%70%3a%2f%2f%77%77%77%2e%73%6c%69%64%65%73%68%61%72%65%2e%6e%65%74%2f%4c%61%72%72%79%43%6f%76%65%72%2f%63%65%70%68%2d%6f%70%65%6e%2d%73%6f%75%72%63%65%2d%73%74%6f%72%61%67%65%2d%73%6f%66%74%77%61%72%65%2d%6f%70%74%69%6d%69%7a%61%74%69%6f%6e%73%2d%6f%6e%2d%69%6e%74%65%6c%2d%61%72%63%68%69%74%65%63%74%75%72%65%2d%66%6f%72%2d%63%6c%6f%75%64%2d%77%6f%72%6b%6c%6f%61%64%73&psig=AFQjCNFqMBOmR1MCHDcx4E8NrIHRdDKN8Q&ust=1443508949557117)）：



该过程具有强一致性的特点：

* Ceph 的读写操作采用 Primary-Replica 模型，Client 只向 Object 所对应 OSD set 的 Primary 发起读写请求，这保证了数据的强一致性。
* 由于每个 Object 都只有一个 Primary OSD，因此对 Object 的更新都是顺序的，不存在同步问题。
* 当 Primary 收到 Object 的写请求时，它负责把数据发送给其他 Replicas，只要这个数据被保存在所有的OSD上时，Primary 才应答Object的写请求，这保证了副本的一致性。这也带来一些副作用。相比那些只实现了最终一致性的存储系统比如 Swift，Ceph 只有三份拷贝都写入完成后才算写入完成，这在出现磁盘损坏时会出现写延迟增加。
* 在 OSD 上，在收到数据存放指令后，它会产生2~3个磁盘seek操作：
  + 把写操作记录到 OSD 的 Journal 文件上(Journal是为了保证写操作的原子性)。
  + 把写操作更新到 Object 对应的文件上。
  + 把写操作记录到 PG Log 文件上。

**3.3 条带化（striping）**

 在 RADOS 层，Ceph 本身没有条带的概念，因为一个object 是作为一个 文件整体性保存的。但是，RBD 可以控制向一个 object 的写入方式，默认是将一个 object 写满再去写下一个object；还可以通过指定 stripe\_unit 和 stripe\_count，来将 object 分成若干个条带即 strip。

  一个 RDB image 会被分为多个 object 来保存，从而使得对一个 image 的多个读写可以分在多个 object 进行，从而可以防止某个 image 非常大或者非常忙时单个节点称为性能瓶颈。还可以将 object 进一步条带化为多个条带（stripe unit）。条带（stripe）是 librados 通过 ODS 写入数据的基本单位。这么做的好处是在保持对象数目的同时，进一步减少可以同步读写的粒度（从 object 粒度减少到 stripe 粒度），从而提高读写效率。

Ceph 的条带化行为（如果划分条带和如何写入条带）受三个参数控制：

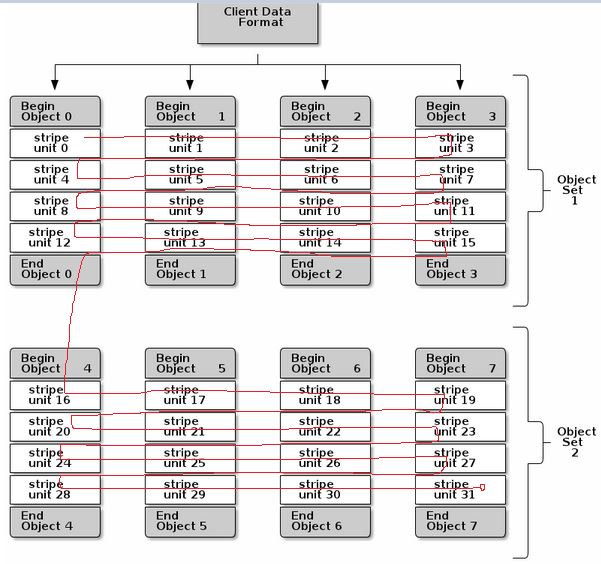
* order：RADOS Object 的大小为 2^[*order*] bytes。默认的 oder 为 22，这时候对象大小为4MB。最小 4k，最大 32M，默认 4M.
* stripe\_unit：条带（stripe unit）的大小。每个 [stripe\_unit] 的连续字节会被连续地保存到同一个对象中，client 写满 stripe unit 大小的数据后，接着去下一个 object 中写下一个 stripe unit 大小的数据。默认为 1，此时一个 stripe 就是一个 object。
* stripe\_count：在分别写入了 [stripe\_unit] 个字节到 [stripe\_count] 个对象后，ceph 又重新从一个新的对象开始写下一个条带，直到该对象达到了它的最大大小。这时候，ceph 转移到下 [stripe\_unit] 字节。默认为 object site。

以下图为例：

（1）RBD image 会被保存在总共 8 个 RADOS object （计算方式为 client data size 除以 2^[*order*]）中。

（2）stripe\_unit 为 object size 的四分之一，也就是说每个 object 包含 4 个 stripe。

（3）stripe\_count 为 4，即每个 object set 包含四个 object。这样，client 以 4 为一个循环，向一个 object set 中的每个 object 依次写入 stripe，写到第 16 个 stripe 后，按照同样的方式写第二个 object set。



   默认的情况下，[stripe\_unit] 等于 object size；stripe\_count 为1。意味着 ceph client 在将第一个 object 写满后再去写下一个 object。要设置其他的 [*stripe\_unit*] 值，需要Ceph v0.53 版本及以后版本对 STRIPINGV2 的支持以及使用 format 2 image 格式。

参考链接：

<http://www.wzxue.com/ceph-librbd-block-library/>

<http://docs.ceph.com/docs/master/architecture/>

<https://www.ustack.com/blog/ceph_infra/>

<https://hustcat.github.io/rbd-image-internal-in-ceph/>

<http://www.wzxue.com/category/ceph-2/>