目录

[前言 2](#_Toc428091743)

[缘起 2](#_Toc428091744)

[版权申明 2](#_Toc428091745)

[上篇：Ceph模块开发篇 3](#_Toc428091746)

[背景介绍 3](#_Toc428091747)

[Ceph平台初探 3](#_Toc428091748)

[初探Ceph架构 3](#_Toc428091749)

[Ceph基础概念 3](#_Toc428091750)

[Ceph基础数据结构 3](#_Toc428091751)

[Ceph的配置系统 3](#_Toc428091752)

[Ceph的模块化体系结构 3](#_Toc428091753)

[Ceph的请求处理 3](#_Toc428091754)

[ObjectStore模块 3](#_Toc428091755)

[模块简介 3](#_Toc428091756)

[模块的基本结构 3](#_Toc428091757)

[模块代码分析 3](#_Toc428091758)

[rgw模块 3](#_Toc428091759)

[其他模块 4](#_Toc428091760)

[下篇：Ceph原理解析篇 4](#_Toc428091761)

[Ceph架构详解 4](#_Toc428091762)

[Ceph的源码目录结构 4](#_Toc428091763)

[Ceph的进程机制 6](#_Toc428091764)

[Ceph基础设施 6](#_Toc428091765)

[Ceph的启动阶段 6](#_Toc428091766)

[Ceph的请求处理解析 6](#_Toc428091767)

[Ceph monitor原理解析 6](#_Toc428091768)

[附录 6](#_Toc428091769)

# 前言

# 缘起

# 版权申明

本书的著作权归作者华三统一存储服务器平台组成员所有。你可以：

* 下载、保存以及打印本书
* 网络链接、转载本书的部分或者全部内容，但是必须在明显处提供读者访问本书发布网站的链接
* 在你的程序中任意使用本书所附的程序代码，但是由本书的程序所引起的任何问题，作者不承担任何责任

你不可以：

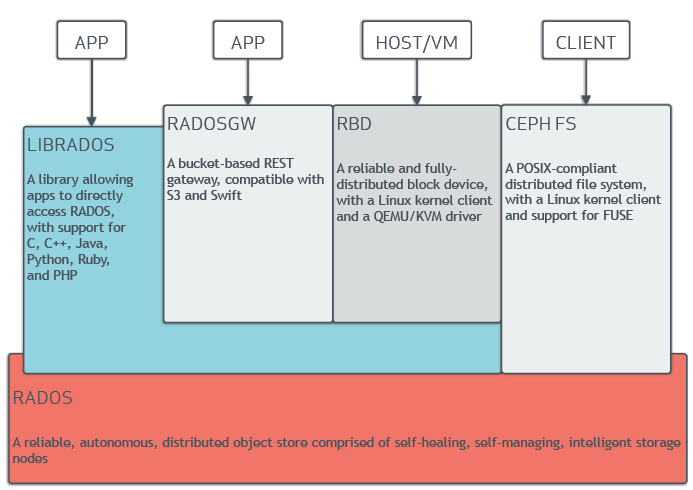
* 以任何形式出售本书的电子版或者打印版
* 擅自印刷、出版本书
* 以纸媒出版为目的，改写、改编以及摘抄本书的内容

# 上篇：Ceph模块入门篇

## 背景介绍

## Ceph平台初探

### 初探Ceph架构



#### 存储集群

Ceph是统一存储系统，支持三种接口。

1. Object：有原生的API，而且也兼容Swift和S3的API
2. Block：支持精简配置、快照、克隆
3. File：Posix接口，支持快照

Ceph也是分布式存储系统，它的特点是：

1. 高扩展性：使用普通x86服务器，支持10~1000台服务器，支持TB到PB级的扩展。
2. 高可靠性：没有单点故障，多数据副本，自动管理，自动修复。
3. 高性能：数据分布均衡，并行化度高。对于objects storage和block storage,不需要元数据服务器。

Ceph基于rados提供了无限可扩展的存储集群，集群主要包含两种类型的后台程序：



Ceph monitor主要负责维护cluster map，ceph osd会检测自己和邻居的状态，并上报给monitors。存储集群客户端和ceph osd并非通过中心节点查表而是通过crush算法高效的计算出数据的location，这样可以避免中心节点带来的热点问题。

#### 数据存储

Ceph集群将数据存储为对象，每一个对象对应一个文件系统中的文件，存放在osd上。OSD负责处理读写数据到磁盘介质。

### Ceph的组件架构



Todo:补充

### Ceph基础概念

### Ceph基础数据结构

BitVector

### Ceph的配置系统

#### librados配置参数

[global]

rados\_mon\_op\_timeout

rados\_osd\_op\_timeout

[mon]

mon\_client\_hunt\_interval // try new mon every N seconds until we connect

mon\_client\_ping\_interval // ping every N seconds

mon\_client\_ping\_timeout, OPT\_DOUBLE, 30.0) // fail if we don't hear back

mon\_client\_hunt\_interval\_backoff // each time we reconnect to a monitor, double our timeout

mon\_client\_hunt\_interval\_max\_multiple // up to a max of 10\*default (30 seconds)

mon\_client\_max\_log\_entries\_per\_message

[objecter]

#### messenger配置参数

#### objectstore配置参数

#### OSD配置参数

filestore\_op\_threads 磁盘写数据线程

//以下两个参数为journal一次写入的最大MOSDOP个数和字节数

journal\_max\_write\_entries = 10000

journal\_max\_write\_bytes = 20971520

//以下两个参数为client与osd交互时的最大并发请求数和字节数

osd\_client\_message\_cap

osd\_client\_message\_size\_cap

//以下两个参数为FileStore的线程数和队列的最大长度

filestore\_op\_threads = 10

filestore\_queue\_max\_ops = 500

//以下两个参数为FileJournal的线程队列最大长度和字节数

journal\_queue\_max\_ops

journal\_queue\_max\_bytes

//SimpleMessenger同时调度的message最大字节数

ms\_dispatch\_throttle\_bytes

因client，cluster，hb\_back\_server，ms\_objecter用的都是该参数，所以这其实是一个设计的问题；

//以下两个参数为objecter的最大发出的MOSDOP和字节数（正在处理中）

objecter\_inflight\_op\_bytes

objecter\_inflight\_ops

//

filestore\_omap\_header\_cache\_size

filestore\_fd\_cache\_size

filestore\_fiemap

ms\_type

ms\_async\_op\_threads

#### TCMALLOC配置参数

TCMALLOC\_MAX\_TOTAL\_THREAD\_CACHE\_BYTES 默认为16777216，可以通过调整这个环境变量来提高因内存分配导致的锁竞争问题；

### Ceph的模块化体系结构

## RGW模块

### 模块简介

全名：rados gateway, ceph对象网关，对外提供对象存储的restful接口；主要提供S3兼容接口，Swift兼容接口，Admin管理接口，继承keystone，以及异地灾备机制；

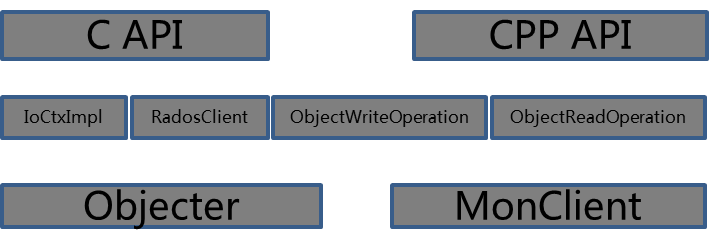
### 模块基本结构

### 模块代码分析

## Librados模块

### 模块简介

### 模块基本结构



### 模块代码分析

#### objecter业务消息

锁

rwlock

timer\_lock

linger\_callback\_lock

#### MonClient业务消息

|  |  |  |  |
| --- | --- | --- | --- |
| 消息类型 | 消息结构 | 消息作用 | 消息处理 |
| CEPH\_MSG\_MON\_MAP | MMonMap | 处理获取MonMap请求； | handle\_monmap |
| CEPH\_MSG\_AUTH\_REPLY | MAuthReply | 处理认证返回请求 | handle\_auth |
| CEPH\_MSG\_MON\_SUBSCRIBE\_ACK | MMonSubscribeAck | 订阅MonMap更新； | handle\_subscribe\_ack |
| CEPH\_MSG\_MON\_GET\_VERSION\_REPLY | MMonGetVersionReply | 获取map版本信息 | handle\_get\_version\_reply |
| MSG\_MON\_COMMAND\_ACK | MMonCommandAck | 命令行请求返回处理 | handle\_mon\_command\_ack |
| MSG\_LOGACK | MLogAck | 向monitor发送Log后处理Log响应请求 | handle\_log\_ack |

MonClient会在authenticate内部发送MMonSubscribe请求订阅MonMap，默认采用一次订阅，定时起会在mon\_subscribe\_interval/2之后重新发送订阅消息，如果MonMap有变化Monitor就会重新发送MonMap给MonClient；

在初始化时调用schedule\_tick，定时起每隔mon\_client\_ping\_interval会send\_keepalive并且查看认证的ticket是否过期，如果keepalive距目前时间超过mon\_client\_ping\_timeout，那么会调用\_reopen\_session重新auth建立session，如果ticket过期，那么重新申请ticket；

## Monitor模块

### 模块简介

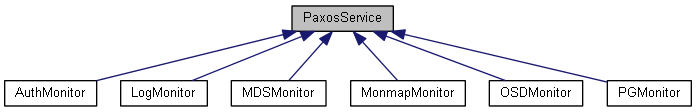
Monitor 作为Ceph的 Metada Server 维护了集群的信息，它包括了6个 Map，分别是 MONMap，OSDMap，PGMap，LogMap，AuthMap，MDSMap。其中 PGMap 和 OSDMap 是最重要的两张Map。

### 模块的基本结构



Todo：修改

### 模块代码分析



Monitor内部使用一套Paxos来实现各种数据的更新，所以所有继承自PaxosService的Monitor实现数据更新时需要通过Paxos达成一致后才能进行；PaxosService的dispatch内部调用子类的preprocess\_query进行查询相关操作，如果非查询类处理，再调用子类的prepare\_update接口实现数据的更新，所以子类Monitor实现两个接口来处理相关的业务消息；

#### Monitor业务消息

##### Monitor自身业务处理

|  |  |  |  |
| --- | --- | --- | --- |
| 消息类型 | 消息结构体 | 消息作用 | 处理接口 |
| CEPH\_MSG\_PING | MPing | 定期Ping Monitor确认Monitor的存在 | handle\_ping |
| CEPH\_MSG\_MON\_GET\_MAP | MMonGetMap | 认证前获取MonMap | handle\_mon\_get\_map |
| CEPH\_MSG\_MON\_METADATA | MMonMetadata | 处理保存某个Monitor的系统信息（cpu，内存等）； | handle\_mon\_metadata |
| MSG\_MON\_COMMAND | MMonCommand | 传递命令行消息给Monitor，Monitor再分发给相应的XXXMonitor进行处理 | handle\_command |
| CEPH\_MSG\_MON\_GET\_VERSION | MMonGetVersion | 获取cluster map的版本信息 | handle\_get\_version |
| CEPH\_MSG\_MON\_SUBSCRIBE | MMonSubscribe | Cluster map订阅更新 | handle\_subscribe |
| MSG\_ROUTE | MRoute | 路由请求转发（待确认） | handle\_route |
| MSG\_MON\_PROBE | MMonProbe | 启动加入时需要向其他Monitor发送Probe请求 | handle\_probe |
| MSG\_MON\_SYNC | MMonSync | 同步Paxos状态数据 | handle\_sync |
| MSG\_MON\_SCRUB | MMonScrub | MonitorDBStore数据一致性检测 | handle\_scrub |
| MSG\_LOGACK |  |  |  |
| MSG\_MON\_JOIN | MMonJoin | 如果不在MonMap中申请加入到MonMap | MonmapMonitor::prepare\_join |
| MSG\_MON\_PAXOS | MMonPaxos | 选举完成后，leader会触发Paxos::leader\_init，状态置为STATE\_RECOVERING，并发起该消息的OP\_COLLECT流程； | Paxos::dispatch |
| MSG\_MON\_ELECTION | MMonElection | 发起选举流程 | Elector::dispatch |
| MSG\_FORWARD | MForward | 将请求转发到leader | handle\_forward |
| MSG\_TIMECHECK | MTimeCheck | Monitor每隔mon\_timecheck\_interval检测所有Monitor的系统时间来检测节点之间的时间差 | handle\_timecheck |
| MSG\_MON\_HEALTH | MMonHealth | 每隔mon\_health\_data\_update\_interval检测存放Monitor上面使用的leveldb数据的状态； | HealthMonitor::service\_dispatch |

##### AuthMonitor业务消息处理

|  |  |  |  |
| --- | --- | --- | --- |
| 消息类型 | 消息结构体 | 消息作用 | 处理接口 |
| MSG\_MON\_COMMAND | MMonCommand | 处理ceph auth xxx命令行相关处理 | preprocess\_command处理ceph auth get/export/list等  prepare\_command处理ceph auth import/add/get-or-create/caps等 |
| CEPH\_MSG\_AUTH | MAuth | 实现认证和授权消息处理 | prep\_auth |
|  |  |  |  |
|  |  |  |  |

##### OSDMonitor业务消息处理

|  |  |  |  |
| --- | --- | --- | --- |
| 消息类型 | 消息结构体 | 消息作用 | 处理接口 |
| CEPH\_MSG\_MON\_GET\_OSDMAP | MMonGetOSDMap | 获取OSDMap | preprocess\_get\_osdmap |
| MSG\_OSD\_MARK\_ME\_DOWN | MOSDMarkMeDown | OSD shutdown之前通知Monitor发送该消息； | preprocess\_mark\_me\_down  prepare\_mark\_me\_down |
| MSG\_OSD\_FAILURE | MOSDFailure | 1. OSD每隔OSD\_TICK\_INTERVAL检测心跳无响应的OSD，并将失败的OSD report给Monitor；  2. Monitor判断上报次数>=mon\_osd\_min\_down\_reports，那么就将target\_osd标识为down； | preprocess\_failure |
| MSG\_OSD\_BOOT | MOSDBoot | 新OSD加入时发送请求到Monitor，参考新OSD的加入流程； | preprocess\_bootprepare\_boot |
| MSG\_OSD\_ALIVE | MOSDAlive | OSD判断up\_thru\_wanted决定是否发送请求给Monitor，Monitor发送Incremental OSDMap返回给OSD； | preprocess\_alive  prepare\_alive |
| MSG\_OSD\_PGTEMP | MOSDPGTemp | Primary OSD处于backfilling状态无法提供读取服务时，会发送该消息到Monitor，将PG临时映射到其他的OSD上提供去服务； | preprocess\_pgtemp  prepare\_pgtemp |
| MSG\_REMOVE\_SNAPS | MRemoveSnaps | 删除快照信息 | prepare\_remove\_snaps |
| CEPH\_MSG\_POOLOP | MPoolOp | 删除/创建Pool，创建/删除pool快照等 | prepare\_pool\_op |

在PaxosService中调用完prepare\_update后都会调用propose\_pending发起决议，paxos结束后，更新clustermap到db中；

##### PGMonitor业务消息处理

|  |  |  |  |
| --- | --- | --- | --- |
| 消息类型 | 消息结构体 | 消息作用 | 处理接口 |
| CEPH\_MSG\_STATFS | MStatfs | 返回文件系统osd占用的kb容量 | handle\_statfs |
| MSG\_PGSTATS | MPGStats | 查询或者更新pg状态 | preprocess\_pg\_stats  prepare\_pg\_stats |
| MSG\_GETPOOLSTATS | MGetPoolStats | 获取pool汇总状态信息 | preprocess\_getpoolstats |
| MSG\_MON\_COMMAND | MMonCommand | 处理ceph pg xxx相关命令行 | preprocess\_command |

##### MonMapMonitor业务消息处理

|  |  |  |  |
| --- | --- | --- | --- |
| 消息类型 | 消息结构体 | 消息作用 | 处理接口 |
| MSG\_MON\_JOIN | MMonJoin | 更新MonMap | preprocess\_join  prepare\_join |
| MSG\_MON\_COMMAND | MMonCommand | 处理ceph mon xxx相关命令行 | preprocess\_command  prepare\_command |

##### MDSMonitor业务消息处理

|  |  |  |  |
| --- | --- | --- | --- |
| 消息类型 | 消息结构体 | 消息作用 | 处理接口 |
| MSG\_MDS\_BEACON |  |  |  |
| MSG\_MDS\_OFFLOAD\_TARGETS |  |  |  |
|  |  |  |  |

##### LogMonitor业务消息处理

|  |  |  |  |
| --- | --- | --- | --- |
| 消息类型 | 消息结构体 | 消息作用 | 处理接口 |
| MSG\_LOG |  |  |  |
|  |  |  |  |
|  |  |  |  |

所有业务消息类型解析的源码为Message \*decode\_message（switch优化）；

Monitor处理业务消息接口实现为Monitor::dispatch

### 模块调试

Ceph-mon进程依赖libec\_isa.so libec\_jerasure.so libec\_jerasure\_sse4.so libec\_lrc.so，调试时需要将这些库拷贝到相应的目录下；

## OSD模块

### 模块简介

OSD全名object storage device，作为ceph中的数据存储节点，为集群提供了对象，对象属性操作，以及对omap的读写能力；

### 模块基本结构

//todo：

### 模块代码分析

#### OSD业务消息

##### objecter与OSD交互对象操作业务消息

###### 消息描述

|  |  |  |
| --- | --- | --- |
| Op类型 | Op作用 | 处理函数 |
| CEPH\_OSD\_OP\_READ | 读对象 | ReplicatedPG::do\_osd\_ops |
| CEPH\_OSD\_OP\_STAT | 获取对象信息 | ReplicatedPG::do\_osd\_ops |
| CEPH\_OSD\_OP\_WRITE | 写对象数据 | ReplicatedPG::do\_osd\_ops |
| CEPH\_OSD\_OP\_TRUNCATE | 截断对象 | ReplicatedPG::do\_osd\_ops |
| CEPH\_OSD\_OP\_APPEND | 附加对象数据 | ReplicatedPG::do\_osd\_ops |
| CEPH\_OSD\_OP\_CREATE | 创建对象 | ReplicatedPG::do\_osd\_ops |
| CEPH\_OSD\_OP\_GETXATTR | 获取对象附加属性 | ReplicatedPG::do\_osd\_ops |
| CEPH\_OSD\_OP\_SETXATTR | 设置对象附加属性 | ReplicatedPG::do\_osd\_ops |

以上CEPH\_OSD\_OP\_xxx定义在rados.h中，无法直接通过编辑器或IDE找到定义，而是通过宏\_\_CEPH\_FORALL\_OSD\_OPS(GENERATE\_ENUM\_ENTRY)内部连接符定义。

###### MOSDOp 代码处理流程简述



1、handle\_op 判断op\_is\_discardable，设置OpRequest请求caps标识

2、OSD::enqueue\_op 调用PG::queue\_op，PG负责调用waiting\_for\_map 并进入PG处理队列ShardedOpWQ。

3、dequeue\_op最终会被调用，先为PG加锁，然后调用PG::do\_request，PG在内部进行如下检测：

1）、PG::op\_has\_sufficient\_caps

2）、PG::can\_discard\_request

3)、the PG is active (PG::flushed boolean)

4）、最后调用表格中的处理函数进行处理；

TODO：确认CEPH\_OSD\_OP\_CALL这个消息的作用，跟cls是否有关；

TODO: 阅读do\_pg\_op中的内容

##### OSD与OSD交互业务消息

###### 消息描述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op类型 | 消息结构 | 消息类型 | Op作用 | 处理函数 |
| OSDOp/  CEPH\_OSD\_OP\_SCRUB\_MAP | MOSDSubOp | MSG\_OSD\_SUBOP | 响应ScrubMap给PrimaryPG | ReplicatedPG::do\_sub\_op |
| OSDOp/  CEPH\_OSD\_OP\_SCRUB\_RESERVE | MOSDSubOp | MSG\_OSD\_SUBOP | PrimaryPG向ReplicatedPG scrub开始时发送scrub reserve | ReplicatedPG::do\_sub\_op |
| PushOp | MOSDPGPush | MSG\_OSD\_PG\_PUSH | ReplicatedPG获取PrimaryPG的Push请求恢复最新数据 | ReplicatedBackend::handle\_message |
| PullOp | MOSDPGPull | MSG\_OSD\_PG\_PULL | PrimaryPG向ReplicatePG主动拉取最新数据 | ReplicatedBackend::handle\_message |
| \_\_u32/ OP\_BACKFILL\_PROGRESS | MOSDPGBackfill | MSG\_OSD\_PG\_BACKFILL | 在发生永久性故障或者PGlog无法起到恢复作用时，需要做全量恢复，同样也是PrimaryPg主动发起Backfill向ReplicatedPG复制数据 | ReplicatedPG::do\_backfill |
| 无 | MOSDPGQuery | MSG\_OSD\_PG\_QUERY | 命令行查询pg状态信息 | OSD:: handle\_pg\_query |
| 无 | MOSDPGCreate | MSG\_OSD\_PG\_CREATE | 创建PG | OSD:: handle\_pg\_create |
| 无 | MOSDPGNotify | MSG\_OSD\_PG\_NOTIFY | Nonprimary到primary发送 | OSD:: handle\_pg\_notify |
| 无 | MOSDPGInfo | MSG\_OSD\_PG\_INFO | 获取PG信息 | OSD:: handle\_pg\_info |
| 无 | MOSDPGLog | MSG\_OSD\_PG\_LOG | PG log | OSD:: handle\_pg\_log |
| 无 | MOSDPGRemove | MSG\_OSD\_PG\_REMOVE | 删除PG | OSD:: handle\_pg\_remove |
| 无 | MOSDPGTrim | MSG\_OSD\_PG\_TRIM | PG修剪 | OSD:: handle\_pg\_trim |
| 无 | MBackfillReserve | MSG\_OSD\_BACKFILL\_RESERVE | PG Backfill开始前资源申请 | OSD:: handle\_pg\_backfill\_reserve |
| 无 | MRecoveryReserve | MSG\_OSD\_RECOVERY\_RESERVE | PG Recovery开始前资源申请 | OSD:: handle\_pg\_recovery\_reserve |
| 无 | MOSDRepScrub | MSG\_OSD\_REP\_SCRUB | PrimaryPG向ReplicatedPG请求scrubmap | PG::replica\_scrub |
|  | MOSDRepOp | MSG\_OSD\_REPOP | 类似于MOSDSubOp | ReplicatedBackend::handle\_message |
| \_\_u32/OP\_SCAN\_GET\_DIGEST | MOSDPGScan | MSG\_OSD\_PG\_SCAN | 扫描PG内容 | ReplicatedPG::do\_scan |
|  | MOSDECSubOpRead | MSG\_OSD\_EC\_READ | 纠删码读数据 | ECBackend::handle\_message |
|  | MOSDECSubOpWrite | MSG\_OSD\_EC\_WRITE | 纠删码写数据 | ECBackend::handle\_message |

###### MOSDSubOps代码流程简述

in\_progress\_ops保存发送给副本的请求消息，在sub\_op\_modify\_reply处理请求响应，如果所有请求均有响应，调用回掉；



1、handle\_sub\_op 检查发送者是否是一个OSD

2、OSD::enqueue\_op 调用PG::queue\_op，PG负责调用waiting\_for\_map 并进入PG处理队列ShardedOpWQ。

3、dequeue\_op最终会被调用，先为PG加锁，然后调用PG::do\_request，PG在内部进行如下检测：

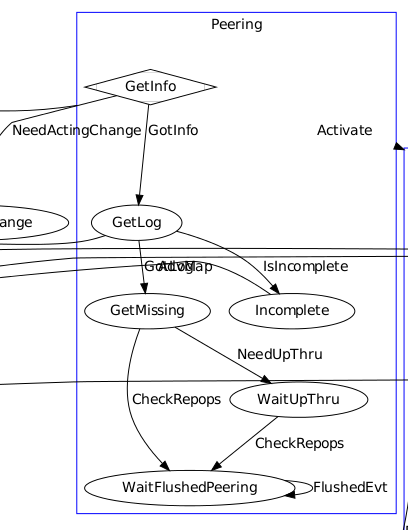
1）、PG::op\_has\_sufficient\_caps

2）、PG::can\_discard\_request

3)、the PG is active (PG::flushed boolean)

4）、最后调用表格中的处理函数进行处理；

###### Peering Message代码流程描述





1. dispatch\_op 将消息分发到handle\_pg\_xxx
2. handle\_pg\_peering\_evt将evt入PG队列；
3. OSD的peering\_wq线程池出队列，调用OSD:: process\_peering\_events, 然后调用handle\_peering\_event，内部调用状态机处理事件；
4. dispatch\_context

GetInfo

Peering的初始状态，通过send\_query(info)获取其他OSD的pg\_info\_t数据信息，当另外一个OSD响应MSG\_OSD\_PG\_NOTIFY信息时，它会被转化为一个MNotifyRec事件，由状态机处理，当所有的OSD均返回该信息时，就会转移到GetLog状态；

GetLog

进入GetLog状态后，如果acting set无法决定，将会退出peering state，并退回到Started或者转化到Incomplete状态，并经过IsIncomplete和AdvMap事件进入Reset状态；否则send\_query(log)，获取GotLog事件后，调用pg->proc\_master\_log然后进入GetMissing 状态；

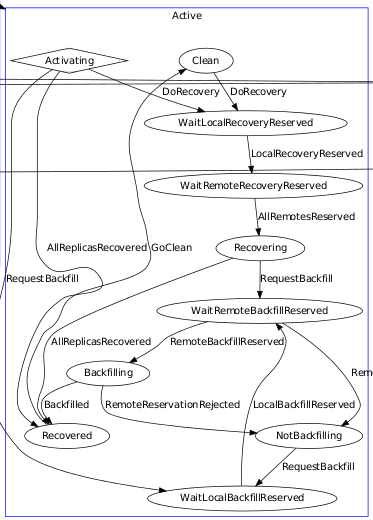
GetMissing

进入GetMissing状态后，如有必要就会请求acting set中所有OSD的PGlog。当接收完毕后，如果需要up\_thru, 先转换到WaitUpThru状态，否则会post一个Activate事件，并退出peering状态进入Active状态；

Todo：

状态机的send\_query如何与dispatch\_context关联？

###### Recovery&Backfilling代码流程描述



Ceph的状态机中PrimaryPG Peering结束后，会进入Active状态，首先决定进入log-based recovery或者backfill recovery，如果进入log-based recovery，primary首先从OSDService的local\_reserver获取local reservation, 然后发送MRemoteReservationRequest消息到所有的replica osd（中间过程PG状态的改变都会反馈给Monitor），直到授权成功才会正式进入Recover过程：



1. Recovering状态调用OSDService::queue\_for\_recovery将PG recovery入队列；
2. recovery\_wq关联的线程池执行OSD::do\_recovery；
3. ReplicatedPG::start\_recovery\_ops判断PG状态，recover\_replicas-> ReplicatedBackend::start\_pushes和recover\_primary-> ReplicatedBackend::prepare\_pull
4. run\_recovery\_op中执行网络请求的Push和Pull数据

log-based recovery结束后，primary释放掉所有的remote reservations,但仍然持有local reservations；然后决定backfill是否有必要，如果没有必要，primary释放掉local reservation，然后处在recovered state直到所有的osd反馈clean状态；

如果backfill有必要（log-based recovery不必要），那么从backfill目标请求remote reservation。

直到所有的reservation返回，正式进入backfilling状态：



1. Backfilling调用OSDService::queue\_for\_recovery将PG Backfill入队列；
2. recovery\_wq关联的线程池执行OSD::do\_recovery；
3. ReplicatedPG::start\_recovery\_ops判断PG状态，调用recover\_backfill；
4. 扫描backfill target的PG内容，并与primary本地PG作对比，然后记录不同的或者missing的Object，最后pgbackend->run\_recovery\_op执行Object的Push请求；

当一个新的OSD加入时，最终需要将新分配的PG需要全量恢复，如果所有PG的backfill同时发生，那么会导致负载过重，所以通过osd\_max\_backfills限制出或入的backfills数量，因为这个过程往往耗时较长，所以每个PG通过last\_backfill 跟踪本地PG拷贝的进度，拷贝完成时，last\_backfill的值为hobject\_t::max()。

###### Scrub代码流程描述

Ceph 的 OSD 会定时启动 Scrub 线程来扫描部分对象，通过与其他副本进行对比来发现是否一致，如果存在不一致的情况，Ceph 会抛出这个异常交给用户去解决。



1. 定时器会定时调用tick\_without\_osd\_lock接口；
2. OSD::sched\_scrub内部循环ScrubJob，调用pg->sched\_scrub
3. PG::sched\_scrub内部判断scrubber.reserved是否为真，如果不为真，发送CEPH\_OSD\_OP\_SCRUB\_RESERVE到replicas，否则入队列到op\_wq中
4. 执行scrub

Scrub描述

(1) Block writes on the chunk

(2) Request maps from replicas

(3) Wait for pushes to be applied (after recovery)

(4) Wait for writes to flush on the chunk

(5) Wait for maps from replicas

(6) Compare / repair all scrub maps

Ceph 的 PG.cc 源文件中的 ASCII 流程描述已经非常形象了，这里只简述内容和补充部分信息。

1. OSD 会以 PG 为粒度触发 Scrub 流程，触发的频率可以通过选项指定，而一个 PG 的 Scrub 启动都是由该 PG 的 Master 角色所在 OSD 启动
2. 一个 PG 在普通的环境下会包含几千个到数十万个不等的对象，因为 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。因此一个 PG 的 Scrub 流程每次会启动小部分的对象校验，Ceph 会以每个对象名的哈希值的部分作为提取因子，每次启动对象校验会找到符合本次哈希值的对象，然后进行比较。这也是 Ceph 称其为 Chunky Scrub 的原因。
3. 在找到待校验对象集后，发起者需要发出请求来锁定其他副本的这部分对象集。因为每个对象的 master 和 replicate 节点在实际写入到底层存储引擎的时间会出现一定的差异。这时候，待校验对象集的发起者会附带一个版本发送给其他副本，直到这些副本节点与主节点同步到相同版本。
4. 在确定待校验对象集在不同节点都处于相同版本后，发起者会要求所有节点都开始计算这个对象集的校验信息并反馈给发起者。
5. 该校验信息包括每个对象的元信息如大小、扩展属性的所有键和历史版本信息等等，在 Ceph 中被称为 ScrubMap。
6. 发起者会比较多个 ScrubMap并发现不一致的对象，不一致对象会被收集最后发送给 Monitor，最后用户可以通过 Monitor 了解 Scrub的结果信息。

用户在发现出现不一致的对象后，可以通过 “ceph pg repair [pg\_id]” 的方式来启动修复进程，目前的修复仅仅会将主节点的对象全量复制到副本节点，因此目前要求用户手工确认主节点的对象是“正确副本”。另外，Ceph 允许 Deep Scrub 模式来全量比较对象信息来期望发现 Ceph 本身或者文件系统问题，这通常会带来较大的 IO 负担，因此在实际生产环境中很难达到预期效果。

MSG\_OSD\_REP\_SCRUB处理接口为replica\_scrub，在此接口中，发送MOSDSubOp（CEPH\_OSD\_OP\_SCRUB\_MAP）收到此消息后进行scrub机制。

##### Monitor与OSD交互业务消息

|  |  |  |  |
| --- | --- | --- | --- |
| 消息结构 | 消息类型 | 消息作用 | 处理函数 |
| MOSDScrub | MSG\_OSD\_SCRUB | 命令行启动scrub机制 | OSD:: handle\_scrub |
| MOSDMAP | CEPH\_MSG\_OSD\_MAP | OSD map更新后发送给OSD | OSD::handle\_osd\_map |

##### OSD心跳机制

<http://blog.csdn.net/dapao123456789/article/details/14169635>

## ObjectStore模块

### 模块简介

### 模块的基本结构

### 模块代码分析

## 其他模块

## 常用工具解析

### 集群不运行时如何获取cluster map，并进行解析

**通过Monitor的DB中获取cluster map**

./ceph-monstore-tool store-path get monmap > monmap

./ceph-monstore-tool store-path get osdmap > osdmap

**解析cluster map**

./monmaptool --print monmap

./osdmaptool --print osdmap

同时还提供了查看leveldb或rocksdb的命令行工具ceph-kvstore-tool

./ceph-kvstore-tool leveldb /xxx/store.db/ list/get

### 如何获取某个PG的info，log，Object信息

**获取PGLog**

./ceph-objectstore-tool --data-path xxx/osd0 --journal-path xxx/osd0.journal --pgid=1.4 --op log

**获取PG info**

./ceph-objectstore-tool --data-path xxx/osd0 --journal-path xxx/osd0.journal --pgid=1.4 --op info

**获取Object Info**

./ceph-objectstore-tool --data-path xxx/osd0 --journal-path xxx/osd0.journal testobj1 dump-info

**查看Journal**

./ceph-objectstore-tool --data-path xxx/osd0 --journal-path xxx/osd0.journal --op dump-journal

**查看osd的super block**

./ceph-objectstore-tool --data-path xxx/osd0 --journal-path xxx/osd0.journal –op dump-super

当然利用ceph-objectstore-tool还可以设置获取osdmap，删除，导出，导入pg，对象删除，attrs/omap获取设置等

注意在xfs文件系统中可以通过attr命令行获取对象的object\_info\_t和snapset信息，在ObjectStore的中就是通过文件系统获取对象的信息，而非通过leveldb获取，在其他不支持扩展属性的文件系统中可能才会通过leveldb获取属性信息；这样的话就可能会存在PGLog与文件系统的attr不一致的问题，可能是由插拔硬盘导致的问题等；

### 通过ceph-kvstore-tool和ceph-dencoder获取和解析MonitorDB中的数据

**获取MonitorDB中的osdmap**

正确方式：

./ceph-kvstore-tool leveldb xxx/mon.a/store.db get osdmap full\_14 out osdmap14

错误方式：

./ceph-kvstore-tool leveldb xxx /mon.a/store.db get osdmap full\_14 > osdmap14

**解析osdmap**

./ceph-dencoder type OSDMap import osdmap14 decode dump\_json

Note：

osdmap:full\_14为ceph-kvstore-tool工具list的结果

Todo：pgmap解析一直异常，目前原因未知

### 通过ceph-kvstore-tool和ceph-dencoder获取和解析DBObjectMap中的数据

每个pg对应一个文件夹，在该文件夹下对应一个object，一般是\_\_head\_xxx\_xxx，该object既pg对应的object，通过该object在leveldb中映射到map object header，该header中保存了seq序号，通过该序号即可找到该object相关的所有属性信息，以系统中只有一个pg，且pgid=1.0为例解释，可以看到该pg文件夹下有唯一的一个\_\_head\_00000000\_\_1 object：

执行./ceph-kvstore-tool leveldb ./dev/osd0/current/omap list 结果：

\_HOBJTOSEQ\_:...head.1.00000000

\_SYS\_:HEADER

\_USER\_0000000000000009\_USER\_:0000000014.00000000000000000001

\_USER\_0000000000000009\_USER\_:\_biginfo

\_USER\_0000000000000009\_USER\_:\_epoch

\_USER\_0000000000000009\_USER\_:\_info

\_USER\_0000000000000009\_USER\_:\_infover

\_USER\_0000000000000009\_USER\_:can\_rollback\_to

\_USER\_0000000000000009\_USER\_:rollback\_info\_trimmed\_to

获取map header

./ceph-kvstore-tool leveldb ./dev/osd0/current/omap get \_HOBJTOSEQ\_ ...head.1.00000000 out header

解析header

执行 ./ceph-dencoder type DBObjectMap::\_Header import header decode dump\_json 输出

{

"seq": 9,

"parent": 0,

"num\_children": 1,

"coll": "meta",

"oid": "1\/00000000\/\/head"

}

可以看到该object header中保存的序号为9

查看9序号相关的key可以看到有如下keys：

\_USER\_0000000000000009\_USER\_:0000000014.00000000000000000001

\_USER\_0000000000000009\_USER\_:\_biginfo

\_USER\_0000000000000009\_USER\_:\_epoch

\_USER\_0000000000000009\_USER\_:\_info

\_USER\_0000000000000009\_USER\_:\_infover

\_USER\_0000000000000009\_USER\_:can\_rollback\_to

\_USER\_0000000000000009\_USER\_:rollback\_info\_trimmed\_to

那么可以肯定的说\_USER\_0000000000000009\_USER\_:\_info对应的信息即为pg对应的pg\_info\_t

获取PGInfo

./ceph-kvstore-tool leveldb ./dev/osd0/current/omap get \_USER\_0000000000000009\_USER\_ \_info out pginfo

解析PGInfo

./ceph-dencoder type pg\_info\_t import pginfo decode dump\_json 输出与执行./ceph-objectstore-tool --data-path ./dev/osd0 --journal-path ./dev/osd0.journal --pgid 1.0 --op info命令一样的信息。

# 下篇：Ceph原理解析篇

## Ceph架构详解

Ceph的下篇将会更加深入的介绍ceph的实现原理。

### Ceph的源码目录结构

#### 根目录

|  |  |
| --- | --- |
| 文件/文件夹 | 描述 |
| src | 各功能模块的源代码 |
| qa | 各个模块的功能测试（测试脚本和测试代码为teuthology提供服务 |
| wireshark | #wireshark的ceph插件 |
| admin | 管理工具，编译生成文档工具 |
| debian | 用于制作debian（Ubuntu）安装包的相关脚本和文件 |
| doc | 用于生成项目文档，生成结果参考http://ceph.com/docs/master/ |
| man | ceph各命令行工具的man文件 |
| examples | 一些使用示例 |
| configure.ac | 用于生成configure的脚本 |
| Makefile.am | 用于生成Makefile的脚本 |
| autogen.sh | 负责生成configure |
| do\_autogen.sh | 生成configure的脚本，实际上通过调用autogen.sh实现 |
| ceph.spec.in | RPM包制作文件 |

#### src目录

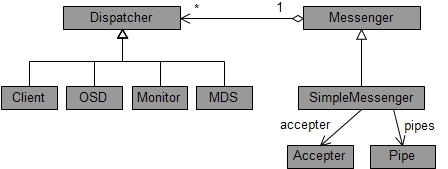
|  |  |
| --- | --- |
|  |  |
| include | 头文件，包含各种基本类型的定义，简单通用功能等 |
| common | 共有模块，包含各类共有机制的实现，例如线程池、管理端口、节流阀等 |
| log | 日志模块，主要负责记录本地log信息（默认/var/log/ceph/目录） |
| global | 全局模块，主要是声明和初始化各类全局变量（全局上下文）、构建驻留进程、信号处理等 |
| auth | 授权模块，实现了三方认知机制 |
| crush | Crush模块，Ceph的数据分布算法 |
| msg | 消息通讯模块，包括用于定义通讯功能的抽象类Messenger以及目前的实现SimpleMessager |
| messages | 消息模块，定义了Ceph各节点之间消息通讯中用到的消息类型 |
| os | 对象（Object Store）模块，用于实现本地的对象存储功能 |
| osdc | OSD客户端（OSD Client），封装了各类访问OSD的方法 |
| mon | mon模块 |
| osd | osd部分 |
| mds | mds模块 |
| rgw | rgw模块 |
| librados | rados库模块的代码 |
| librdb | librbd库模块的代码 |
| client | client模块，实现了用户态的CephFS客户端 |
| mount | mount模块 |
| tools | 各类工具 |
| test | 单元测试 |
| perfglue | 与性能优化相关的源代码 |
| json\_spirit | 外部项目json\_spirit |
| rocksdb | 外部项目 |
| gtest | gtest单元测试框架 |
| doc | 关于代码的一些说明文档 |
| bash\_completion | 部分bash脚本的实现 |
| pybind | python的包装器 |
| script | 各种脚本 |
| upstart | 各种配置文件 |
| ceph\_mds.cc | 驻留程序mds |
| ceph\_mon.cc | 驻留程序mon |
| ceph\_osd.cc | 驻留程序osd |
| libcephfs.cc | cephfs库 |
| ceph\_authtool.cc | 工具ceph\_authtool |
| ceph\_conf.cc | 工具ceph\_conf |
| ceph\_fuse.cc | 工具ceph\_fuse |
| ceph\_syn.cc | 工具ceph\_syn |
| cephfs.cc | 工具cephfs |
| librados-config.cc | rados库配置工具 |
| rdb.cc | 工具rdb |
| sample.ceph.conf | 配置文件样例 |
| ceph.conf.twoosds | 配置文件样例 |
| valgrind.supp | 内存检查工具valgrind的配置文件 |
| init-ceph.in | 启动和停止ceph的脚本 |
| vstart.sh | 单机上启动调试集群 |
| init-radosgw | radosgw启动停止脚本 |

## Ceph基础设施

### Ceph的buffer

### Ceph的线程模型

### Ceph的通信模型



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。

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。

### PerfCounter分析

Mon, OSD均通过admin socket提供了性能统计的接口，使用

ceph daemon osd.x perf dump, ceph daemon mon.x perf dump可以查看perf统计情况，通过开启optracker（默认开启），可以ceph daemon osd(mon).x ops查看运行时进程中的op所处的状态。

#### FileStore各个字段解析

queue\_transaction\_latency\_avg代表filestore throttle的等待时间，如果该值过大，可以考虑修改参数

filestore\_op\_threads(默认为2)和filestore\_queue\_max\_ops(默认为50)因在关键路径上，所以考虑增大两个值来降低 queue\_transaction\_latency\_avg的值；

#### OSD perf字段解析

op\_latency 从接收到请求到请求到落到journal后的时间和

op\_process\_latency 从请求出队列到请求到journal后的时间和

根据分析op\_process\_latency在op\_latency中占主要部分，所以是否可以考虑增加op的线程池；

#### Monitor perf字段解析

## Ceph的启动阶段

## Ceph的请求处理解析

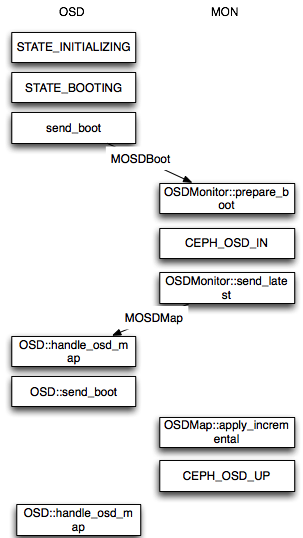
### Mon cmd请求流程

Todo：

1. 新的osd加入流程

### 新OSD加入流程

当一个新的 OSD 启动时，这时 Monitor 所掌握的最新 OSDMap 并没有该 OSD 的情况，因此该 OSD 会向 Monitor 申请加入，Monitor 再验证其信息后会将其加入 OSDMap 并标记为IN，并且将其放在 Pending Proposal 中会在下一次 Monitor “讨论”中提出，OSD 在得到 Monitor 的回复信息后发现自己仍然没在 OSDMap 中会继续尝试申请加入，接下来 Monitor 会发起一个 Proposal ，申请将这个 OSD 加入 OSDMap 并且标记为 UP 。然后按照 Paxos 的流程，从 proposal->accept->commit 到最后达成一致，OSD 最后成功加入 OSDMap 。当新的 OSD 获得最新 OSDMap 发现它已经在其中时。这时，OSD 才真正开始建立与其他OSD的连接，Monitor 接下来会开始给他分配PG。

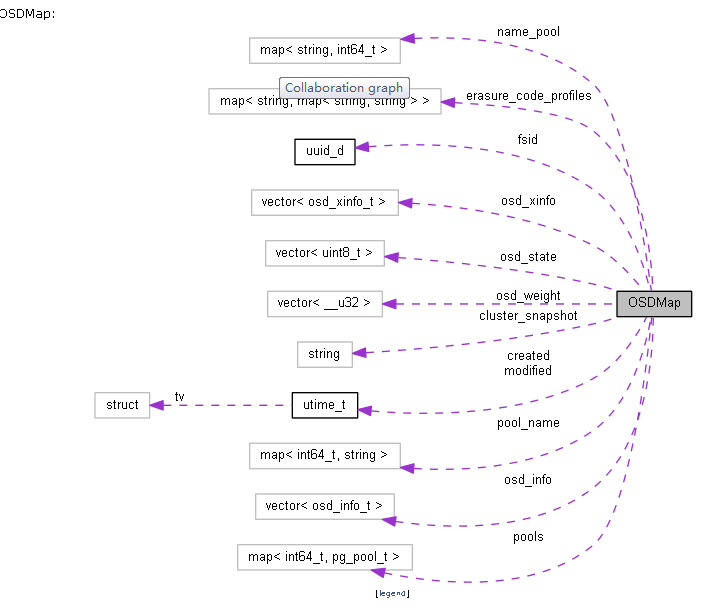


## Ceph monitor原理解析

### OSDMonitor

#### 关键点分析

##### OSDMap



pool\_name可以根据pool名称获取pool Id，pools成员变量存放pool id与pool所有信息的关系映射；

##### 以ceph osd map <poolname> <objectname>为例讲解oid->pgid, pgid->osds映射关系

int64\_t pool **=** osdmap**.**lookup\_pg\_pool\_name**(**poolstr**.**c\_str**()); 获取Pool id**

**if** **(**pool **<** 0**)** **{**

ss **<<** "pool " **<<** poolstr **<<** " does not exist"**;**

r **=** **-**ENOENT**;**

**goto** reply**;**

**}**

object\_locator\_t oloc**(**pool**,** namespacestr**);**

object\_t oid**(**objstr**);**

pg\_t pgid **=** osdmap**.**object\_locator\_to\_pg**(**oid**,** oloc**); oid做hash取得fake pid**

pg\_t mpgid **=** osdmap**.**raw\_pg\_to\_pg**(**pgid**); 对fake pid取模获取real pid**

vector**<int>** up**,** acting**;**

**int** up\_p**,** acting\_p**;**

osdmap**.**pg\_to\_up\_acting\_osds**(**mpgid**,** **&**up**,** **&**up\_p**,** **&**acting**,** **&**acting\_p**); 根据crush算法计算出pgid映射到那些acting 状态的osd**

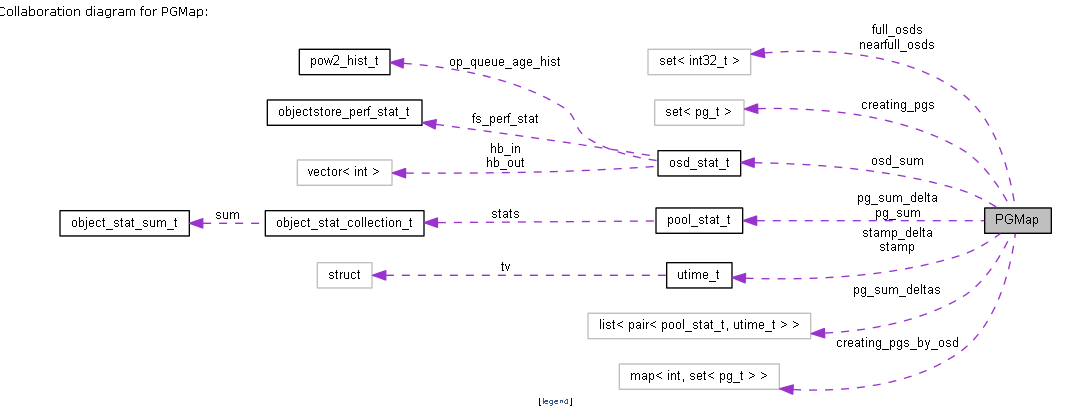
##### ceph pool create 命令触发PG创建的流程

在调用ceph pool create命令行时，调用的是OSDMonitor中的prepare\_command，然而如何触发PGMap的改变呢，关键点是在paxos结束后在OSDMonitor::update\_from\_paxos中调用mon->pgmon()->check\_osd\_map(osdmap.epoch);通过此调用PGMonitor内部检测是否需要有新的PG创建，如果有重新发起paxos，并发送创建pool请求到各个OSD；

### PGMonitor

#### 关键点分析

##### PGMap



其中pg\_stat包含了所有的pg状态信息，osd\_stat包含了所有的osd状态信息；

##### 剖析ceph pg map <pgid>

参数pgid实际上组成为poolid.fakepid，解析出pool id和fake pid后，剩下的步骤与之前的ceph osd map命令行一致；

### Ceph Paxos算法分析

Reference： http://blog.csdn.net/skdkjzz/article/details/41979665

### Crush算法

<http://way4ever.com/?p=123>

设置pool使用哪个rule：

ceph osd pool set <poolname> crush\_ruleset 4

## Ceph OSD原理解析

### 数据结构解析

### 关键点分析

#### OSD心跳故障自我检测

心跳是用于OSD节点间检测对方是否故障的，以便及时发现故障节点进入相应的故障处理流程。故障检测需要在故障的发现时间和心跳带来的负载之间做权衡，如果心跳频率太高则过多的心跳报文会影响系统性能，如果心跳频率过低则会延长发现故障节点的时间，从而影响系统的可用性。

建立连接

在大规模部署的场景中，如果任意两个OSD节点间都建立心跳连接将带来巨大的负担。尤其，当新加入一个OSD节点时这个负担就会几倍地增加。Ceph中每个OSD只和以下两类节点建立心跳连接：一类是同一个PG下的OSD节点之间，因为属于同个PG的OSD节点会保存同份数据的副本，如若出现故障则会直接影响数据的可用性。另一类是OSD的左右两个相邻的节点，这两个节点同自己物理上存在比较紧密的联系，例如可能连接在同台交换机。另外，如果建立心跳的Peer数目少于osd\_heartbeat\_min\_peers，那么OSD会继续同离他较近的几个OSD建立心跳连接。

OSD节点会监听public、cluster、front和back四个端口，其中front和back两个端口都是用于心跳的，cluster端口用来监听来自OSD Peer的连接，public用来监听来自Monitor和Client的连接。如果启动OSD时没有提供back的IP地址，则back使用cluster的IP地址；而front不单独提供IP地址，直接使用public的IP地址。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。心跳报文优先发送给back连接。

代码注释

// ceph-osd.cc 启动osd时创建Messengers

OSD::maybe\_update\_heartbeat\_peers() 确定同哪些peer建立心跳连接，剔除已经down掉的节点的心跳连接

OSD::\_add\_heartbeat\_peer() 同给定的peer建立心跳连接

OSDServeice::get\_con\_osd\_hb() 获取peer的front和back连接

配置

OPTION(public\_network, OPT\_STR, "")

OPTION(cluster\_network, OPT\_STR, "")

OPTION(osd\_heartbeat\_min\_peers, OPT\_INT, 10) // minimum number of peers

检测故障

OSD使用T\_Heartbeat线程定时向Peer OSDs发送心跳报文，发送报文的时间间隔在0.5~6.5之间，由osd\_heartbeat\_interval配置选项决定。心跳报文会同时向Peer OSD的front和back端口发送。心跳报文分两种类型一种是Ping类型，另一种是Reply类型。Ping类型的报文是OSD主动发送给Peer OSD的报文，而Reply是Peer OSD回应给自己的报文。两种类型的心跳报文都携带时间戳，但它们的时间戳代表的含义不一样。Ping类型报文的时间戳是发送报文时的时间，而Reply类型报文的时间戳是从Ping报文中读取出来的，不是代表它自己的发送时间而是代表它对应的Ping报文的发送时间。OSD接收到Reply报文时将记录报文的时间戳，并以此来判断是否超时。

对每个Peer节点，如果其最近的应答的时间(最近的Reply报文的时间戳)位于cutoff之前(即超时grace秒)，则将其加入到failure\_queue队列。OSD会定时向Monitor汇报自己的状态，在汇报状态时将failure\_queue队列中Peer发送给Monitor，由Monitor将其标记为down状态。Monitor在接收到OSD对Peer的故障报告后，通过PAXOS算法决定是否将Peer OSD标记为Down状态。如果将Peer OSD标记为Down状态，那么将更新OSD MAP，OSD接收到OSD Map更新的消息后，断开和Peer OSD的心跳连接。

如果在向Monitor报告故障之后但在接收到OSD Down消息之前，再次接收到Peer OSD对心跳报文的回应，则将Peer OSD从failure\_queue队列中移除，并通知Monitor该节点依旧存活着。

代码注释

void OSD::heartbeat\_entry() // T\_Heartbeat线程入口函数，定时向心跳Peers发送心跳报文

void OSD::heartbeat()

map<int,utime\_t> failure\_queue; // 检测到peer长时间没心跳时，将peer加入到failure\_queue队列

map<int,entity\_inst\_t> failure\_pending; // 故障报告给Monitor的Peer OSD

void send\_failures();

void send\_still\_alive(epoch\_t epoch, const entity\_inst\_t &i);

void OSD::note\_down\_osd(int peer)

void OSD::handle\_osd\_ping(MOSDPing \*m) // 处理MOSDPing消息

配置

OPTION(osd\_heartbeat\_interval, OPT\_INT, 6) // (seconds) how often we ping peers

OPTION(osd\_heartbeat\_grace, OPT\_INT, 20) // (seconds) how long before we decide a peer has failed

Reference: <http://blog.csdn.net/skdkjzz/article/details/41980885>

#### 新增osd或者weight变化时PG的迁移原理

无论进行上面什么样的操作都会导致osdmap变化，然后新的OSD上面如果分配了主PG，那么主PG何时创建的呢，那么先看下osd中创建PG的代码：

OSD::handle\_pg\_peering\_evt中：

PG::\_create(\*rctx.transaction, pgid, pgid.get\_split\_bits(pp->get\_pg\_num()));

PG::\_init(\*rctx.transaction, pgid, pp);

PG \*pg = \_create\_lock\_pg(

get\_map(epoch),

pgid, create, false, result == RES\_SELF,

role,

up, up\_primary,

acting, acting\_primary,

history, pi,

\*rctx.transaction);

那么什么时候会调用handle\_pg\_peering\_evt呢？

我们来看下PG的Reset状态：

无论是Primary还是replicated，Started状态都会处理AdvMap，并转移到Reset状态，

Reset状态判断是否需要重启Peering，并判断如果不是primary，就置send\_notify为true，然后Reset状态响应ActMap，调用send\_notify向主PG发送notify并带有info信息！主PG所在的OSD收到消息调用handle\_pg\_notify进行处理，并创建PG；

同时主PG进行Peering过程中， proc\_replica\_info会发现之前主PG所在的osd有冗余数据，加入stray\_set，最终发送MOSDPGRemove消息将该osd上的PG删除；

# 系统问题以及解决方案

## Scrub 问题

正如流程所述，目前的 Scrub 有以下问题:

1. 在发现不一致对象后，缺少策略来自动矫正错误，比如如果多数副本达成一致，那么少数副本对象会被同化
2. Scrub 机制并不能及时解决存储系统端到端正确的问题，很有可能上层应用早已经读到错误数据

对于第一个问题，目前 Ceph 已经有 Blueprint 来加强 Scrub 的修复能力，用户启动 Repair 时会启动多数副本一致的策略来替代目前的主副本同步策略。

对于第二个问题，传统端到端解决方案会更多采用固定数据块附加校验数据的“端到端校验”方案，但是 Ceph 因为并不是存储设备空间实际的管理和分配者，它依赖于文件系统来实现存储空间的管理，如果采用对象校验的方式会严重损耗性能。因此在从文件系统到设备的校验需要依赖于文件系统，而 Ceph 包括客户端和服务器端的对象正确性校验只能更多的依赖于 Read Verify 机制，在涉及数据迁移时需要同步的比较不同副本对象的信息来保证正确性。目前的异步方式会允许期间发生错误数据返回的可能性。

## ThreadPool滥用，系统线程过多，高并发时系统资源过耗

## 条带功能没有放在librados中，导致所有的客户端模块均需实现自己的条带模块

## Crushmap编辑时，step choose firstn 0 type rack如果不修改为step chooseleaf firstn 0 type rack会导致./crushtool': double free or corruption (out)错误，导致无法导入crushmap

## 修改系统时间导致集群崩溃（OSD进程退）

## MonitorClient问题

start\_mon\_command有三个接口，只有一个接口有超时机制；

wait\_for\_latest\_osdmap 无超时机制；

# QA

##### Peering过程中发生OSD故障后，集群恢复可用性的时延

# 附录