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模块开发篇

## 背景介绍

## Ceph平台初探

### 初探Ceph架构

### Ceph的组件架构



Todo:补充

### Ceph基础概念

### Ceph基础数据结构

### Ceph的配置系统

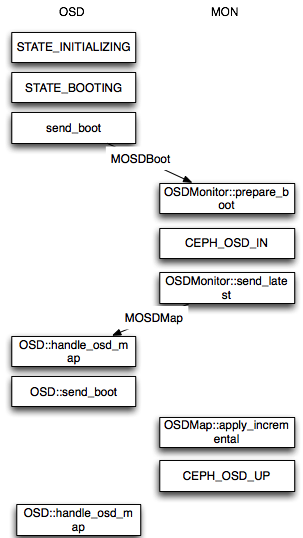
### Ceph的模块化体系结构

### Ceph的请求处理

#### 请求处理的流程

1. 新的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。



## ObjectStore模块

### 模块简介

### 模块的基本结构

### 模块代码分析

## rgw模块

## mon模块

### 模块简介

### 模块的基本结构

### 模块代码分析

1. Client->mon业务消息

|  |  |  |
| --- | --- | --- |
| 消息类型 | 消息值 | 消息结构体 |
| CEPH\_MSG\_MON\_MAP | 4 | MMonMap |
| CEPH\_MSG\_MON\_GET\_MAP | 5 | MMonGetMap |
| CEPH\_MSG\_MON\_GET\_OSDMAP | 6 | MMonGetOSDMap |
| CEPH\_MSG\_MON\_METADATA | 7 | MMonMetadata |
| CEPH\_MSG\_STATFS | 13 | MStatfs |
| CEPH\_MSG\_STATFS\_REPLY | 14 | MStatfsReply |
| CEPH\_MSG\_MON\_SUBSCRIBE | 15 | MMonSubscribe |
| CEPH\_MSG\_MON\_SUBSCRIBE\_ACK | 16 | MMonSubscribeAck |
| CEPH\_MSG\_AUTH | 17 | MAuth |
| CEPH\_MSG\_AUTH\_REPLY | 18 | MAuthReply |
| CEPH\_MSG\_MON\_GET\_VERSION | 19 | MMonGetVersion |
| CEPH\_MSG\_MON\_GET\_VERSION\_REPLY | 20 | MMonGetVersionReply |

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

### 模块基本结构

### 模块代码分析

## 其他模块

# 下篇：Ceph原理解析篇

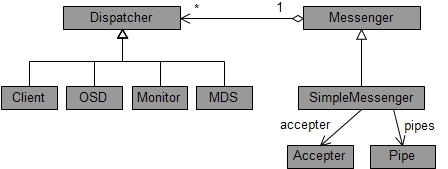
## Ceph架构详解

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

1. 根目录
   1. [src]：各功能模块的源代码
   2. [qa]：各个模块的功能测试（测试脚本和测试代码）
   3. [wireshark]：#wireshark的ceph插件
   4. [admin]：管理工具，编译生成文档工具
   5. [debian]：用于制作debian（Ubuntu）安装包的相关脚本和文件
   6. [doc]：用于生成项目文档，生成结果参考http://ceph.com/docs/master/
   7. [man]：ceph各命令行工具的man文件
   8. [examples]：一些使用示例
   9. configure.ac：用于生成configure的脚本
   10. Makefile.am：用于生成Makefile的脚本
   11. autogen.sh：负责生成configure
   12. do\_autogen.sh：生成configure的脚本，实际上通过调用autogen.sh实现
   13. ceph.spec.in：RPM包制作文件
2. src目录
   1. [include]：头文件，包含各种基本类型的定义，简单通用功能等。
   2. [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。
   3. [log]：日志模块，主要负责记录本地log信息（默认/var/log/ceph/目录）
   4. [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
   5. [auth]：授权模块，实现了三方认知机制
   6. [crush]：Crush模块，Ceph的数据分布算法
   7. [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
   8. [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
   9. [os]：对象（Object Store）模块，用于实现本地的对象存储功能
   10. [osdc]：OSD客户端（OSD Client），封装了各类访问OSD的方法
   11. [mon]：mon模块
   12. [osd]：osd部分
   13. [mds]：mds模块
   14. [rgw]：rgw模块的
   15. [librados]：rados库模块的代码
   16. [librdb]：librbd库模块的代码
   17. [client]：client模块，实现了用户态的CephFS客户端
   18. [mount]：mount模块
   19. [tools]：各类工具
   20. [test]：单元测试
   21. [perfglue]：与性能优化相关的源代码
   22. [json\_spirit]：外部项目json\_spirit
   23. [leveldb]：外部项目leveldb from google
   24. [gtest]：gtest单元测试框架
   25. [doc]：关于代码的一些说明文档
   26. [bash\_completion]：部分bash脚本的实现
   27. [pybind]：python的包装器
   28. [script]：各种python脚本
   29. [upstart]：各种配置文件
   30. ceph\_mds.cc：驻留程序mds
   31. ceph\_mon.cc：驻留程序mon
   32. ceph\_osd.cc：驻留程序osd
   33. libcephfs.cc：cephfs库
   34. librdb.cc：rdb库
   35. ceph\_authtool.cc：工具ceph\_authtool
   36. ceph\_conf.cc：工具ceph\_conf
   37. ceph\_fuse.cc：工具ceph\_fuse
   38. ceph\_syn.cc：工具ceph\_syn
   39. cephfs.cc：工具cephfs
   40. crushtool.cc：工具crushtool
   41. dupstore.cc：工具dupstore
   42. librados-config.cc：rados库配置工具
   43. monmaptool.cc：工具monmap
   44. osdmaptool.cc：工具osdmap
   45. psim.cc：工具psim
   46. rados.cc：工具rados
   47. rdb.cc：工具rdb
   48. rados\_export.cc：rados工具相关类
   49. rados\_import.cc：rados工具相关类
   50. rados\_sync.cc：rados工具相关类
   51. rados\_sync.h：rados工具相关类
   52. sample.ceph.conf：配置文件样例
   53. ceph.conf.twoosds：配置文件样例
   54. Makefile.am：makefile的源文件
   55. valgrind.supp：内存检查工具valgrind的配置文件
   56. init-ceph.in：启动和停止ceph的脚本
   57. mkcephfs.in：cephfs部署脚本

### Ceph的消息处理机制



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。

## Ceph基础设施

## Ceph的启动阶段

## Ceph的请求处理解析

### Mon cmd请求流程

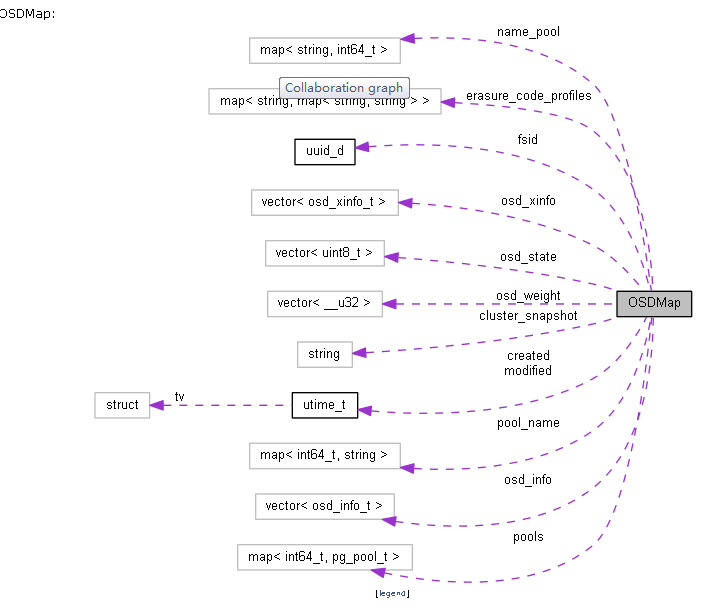
Todo：

## 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**;**

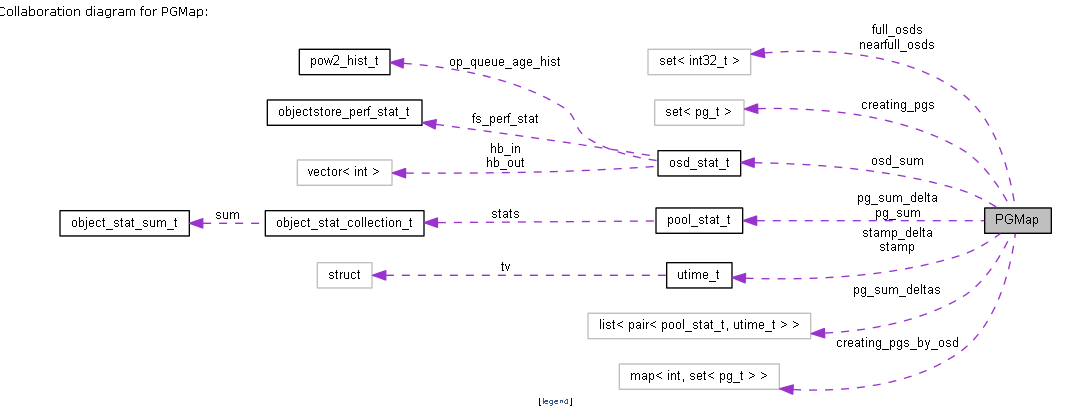
**int** up\_p**,** acting\_p**;**

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**

### 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

### 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

# 附录