CCeph系统设计

袁冬 2015年12月

# 1 概述

CCeph是一个参考Ceph设计的分布式存储系统，主要针对Ceph做以下改进：

* 采用NWR一致性模型，允许客户端根据场景定义可靠性/可用性的偏好
* 避免单点故障而导致的超时等待，IO在设计范围内不存在Hang
* 严格分离RADOS和具体存储协议（CephFS、RBD、RGW）的实现
* 面向SSD进行优化
* 支持实时纠删
* 采用纯C实现

## 1.1 系统接口

CCeph以对象为数据单位，支持对于对象的读、写和删除操作。其中读写操作可以指定偏移量和长度，即支持随机访问。暂时不考虑严格意义上的Append操作。

三个接口定义为：

write(char\* oid, int64 offset, int64 length, char\* data)

read(char\* oid, int64 offset, int64 length, char\* data)

delete(char\* oid)

ceph提供的OMap、cls接口暂不支持。

## 1.2 设计目标

系统的核心设计目标是：**所有IO请求不等待任何可用性设计范围内的超时。**

即假设数据采用三副本的容灾设计，可用性为允许一个副本宕机，则对于任何单点故障，请求的延迟（Latency）不受影响。

# 2 设计

## 2.1 系统架构

系统架构仿照ceph，目前主要关心RADOS，包括两个主要的子系统：mon和osd。两者在实际环境中是两个子集群，如下图所示：



图 Cceph系统架构（同ceph）

系统设计到的几个概念与Ceph相同，包括但不限于mon、osd、client、rados、pg、pg\_log、crush等。

## 2.2 关键设计

CCeph大部分核心设计与cceph相同，但是在涉及数据版本的一致性认证和恢复操作时，则采用完全不同的设计。

**【一致性】**

系统采用NWR一致性模型，其中N为存储池级别的参数，WR为请求级别的参数。为了保证数据一致性，系统规定W必须>N/2。

当W+R > N时，系统表现为强一致性 ，即如果系统向客户端报告写成功，则该客户端（或者采用相同NWR规则的其它客户端）可以读到最新的数据。典型的情况是W=3，N=R=2的情况。

当W+R <= N时，系统表现为最终一致性，即可能存在脏读，读取到数据可能是旧版本或临时版本。考虑以下场景：N=2，W=2，R=1的规则下，假设三个数据副本分别是a，b，c，数据x初始版本为v1。客户端c1执行写操作将x更新为v2，假设写操作在a，b上完成，而尚未在c上完成，客户端c1认为x的当前版本已经v2。但是此时假设客户端c1，或者其他客户端c2执行R=1的读操作，而恰好读取的是c副本，则仍然可能读取到脏数据，即x的v1版本。

**【数据分布】**

集群状态（OSDMap）是数据分布的依据。系统的参与者（即osd和client）通过osdmap计算数据的分布。每一个数据副本基于当前版本的osdmap，被一致性哈希（CRUSH）到一组osd上，这组osd是有序的。

**【数据分布的变更】**

系统任意osd状态的变化都通过更新或发布新的osdmap生效。包括osd的加入、离开、权重的调整等。

一种等效的描述方式是：一切可能导致数据分布发生变化的情况，都通过发布osdmap通知到所有osd和client。反之，如果osd和client没有接收到任何新的osdmap，则可以认为数据分布没有任何变化。

数据变更是系统容灾的核心设计，其主要机制是PG状态的转换，详细描述参考后续章节《变更》

【PG】

数据分布的单位是PG（而非对象），PG的概念等同于ceph中PG的概念。（但在工程实现上CCeph的PG与Ceph的PG完全不同）。

数据分布的变更表现为PG所对应的osd向量产生了变化，即PG的迁移。PG迁移由该PG的最新osd负责。

**【PG与请求】**

每一个对象的请求会交给其对应的PG进行处理，PG包含一个数据的版本号pg\_version/pg\_ver/pg\_v。PG内部任意一个对象的变更都会导致pg\_v加一，pg\_v认为是无限大的（生产中采用int64）。PG每处理一个请求，都会对该请求生成一个pg\_v，代表这一次修改。

一个PG内部的对象的处理是串行的，PG之间的对象的处理是并行的。

# 3 流程

## 3.1 写入流程

写流程采用投票机制和二段式提交，即各个（1）副本收到请求后写入本地Prepare日志，（2）然后发起投票，报告自己要提交的版本及对应的数据，（3）当副本拿到半数以上的投票时，说明该写请求已经生效。则（4）写入Commit日志，并（5）回复ACK。

客户端收到了W份Rep的请求成功的响应，则客户端都可以认为请求已完成。

**第一步：请求发送**



客户端向每一个副本发送完全相同的请求，包含以下内容：

* client\_id：每个客户端的唯一ID，客户端连接到集群时，从mon获取。
* request\_id：客户端每个请求的ID，由客户端编号，不重复。
* osdmap\_epoch：客户端所使用的osdmap的版本号
* oid：要写入到对象ID
* offset、length、data、crc：请求内容。

每个请求的ClientID+RequestID可以唯一的区分这个请求。

在实现中，为了最大程度减少并发导致的版本冲突，降低请求延迟，所有客户端都将根据副本的序号依次发送请求。这样在理想状态下，请求到达各个副本都是按照相同的顺序，避免乱序导致的版本冲突。这是一个理想情况，协议并不要求此条件必须满足。

**第二步：副本投票**



投票的目的是确认数据修改操作得到了半数以上的Rep的认可。具体的流程又可细分为：（1）Preapre、（2）Vote和（3）Apply

（1）Preapre

当每一个Rep收到请求时，都需要基于要修改的对象的版本，生成新的数据版本号。

假设，客户端c1的请求q1，要修改o1，该请求可以描述为：

(client\_id=c1, req\_id=q1, oid=o1)

则对于Rep，该请求还会有一个oid对应的新版本号，因此请求被标示为：

(client\_id=c1, req\_id=q1, oid=o1，v=1)

版本号是基于oid的，是一个递增的整数。实现中采用int64，视为无穷大，考虑到性能，所有对象的当前版本号全部需要缓存到内存中（占内存很小）。

当编号完成后，Rep会将请求，包含其数据版本号和请求的所有内容，记录到本地的日志中，这是一条Prepare日志。

此时数据并不能被访问到。

（2）Vote

当prepare日志被成功的记录到本地后，Rep会发起投票。即向其它Rep发送请求的情况，包括（ClientID、RequestID、oid、v）。注意Vote并不需要互相发送数据，最主要的信息是该Rep对某一个请求的版本号v，投票的主要目的是在Rep之间达成对数据变更的一致的认知。

当任意Rep收到半数（包含本身）以上的相同的Vote消息后，则可以确认请求已经的状态已经达成一致，则可以进入Apply过程。

（3）Apply

Apply流程会向磁盘写入一条Commit的日志，然后修改内存缓存以反映数据的变化。

**第三步：回复响应**



当一个Rep收到半数以上的投票并完成本地的应用操作（Apply）后，即可向客户端返回请求完成。

对于客户端来说，写请求的返回值满足以下约束：

* 如果客户端收到了>=W个Rep的ACK，则客户端认为请求已完成
* 如果客户端收到了任意Rep的请求失败的响应，则该请求必然已经失败。
* 如果客户端没有收到任何响应，则数据的状态对于该客户端未知。

如果一个请求成功，则满足：

* 该版本永久生效（除非数据丢失）
* 之后任意客户端都不会读到该版本之前的数据（强一致性）

另外，副本不仅会向客户端发送ACK，还向所有参与该操作的副本发送ACK。每一个副本对于一个已经Apply的请求，会维护一个ACK数目的计数器ack\_count。每当副本收到其他副本发来的一个ACK，则将ack\_count++。当ack\_count>=W时，意味着该副本请求在全局上都是成功的，即对于所有客户端来说是成功的。

当该副本收到一个对于对象o的读请求时，副本会根据ack\_count返回o的最新版本的状态，当ack\_count>=W时，版本状态是应用/Apply，否则是提交/Commit。

“应用/Apply”状态保证：在此之后，该客户端以及其他客户端都不会读到更早版本的数据（只要系统没有发生导致数据丢失的异常）。而“提交/Commit”状态只意味着该请求对于该副本是成功的，但不保证是全局成功的。

客户端在收到副本报告的失败请求时，其原因是该投票失败，即存在并发的请求，使得当前的请求无法成功。则客户端需要重试。

## 3.2 读取流程

客户端在读取数据时，可以采用以下三种策略：

* 弱一致性读：读取任一副本
* 强一致性读：读取所有副本

弱一致性较为简单，不做分析，以下主要分析强一致性设计：

当客户端需要强一致性的读取某个对象时，客户端向其所有副本发送读请求，请求中包括以下数据：

* ClientID：客户端的ID
* RequestID：ClientID+RequestID唯一确定一个请求
* OSDMapEpoch：请求所对应的OSDMap版本号
* ObjectID/oid：对象ID
* Offset+Length：请求内容

当副本收到读请求时，每一个副本基于其所持有的对象o的情况，返回以下信息：

（1）所持有的o的最新版本的版本号v

（2）所持有的o的最新版本的数据data\_v

（3）所持有的o的最新版本的状态stat\_v，值是“提交/Commit”或“应用/Apply”

对于副本来说，以上内容的确认规则为：

（1）最新版本v：如果对象o不存在v+1版本，或v+1版本处于Prepare状态，则其最新版本为v，否则其最新版本为v+1。

（2）版本v的状态stat\_v：如果版本v已经写入了commit日志，但是未收到超过N/2个ACK，则其状态为提交Commit；否则，其状态为应用/Apply。

对于客户端来说，至少获取R个副本的消息才能确认数据的版本，其对象版本和数据的确认逻辑为：

【情况-1】在R个回复中，如果有一个响应的版本号最大，且其版本状态为“应用/Apply”时，则该数据即为该次读取操作的结果。

【情况-2】如果R个回复中，最大版本号vmax有W个“提交/Commit”状态的响应，且其数据一致，则可以采用该数据作为结果。这种情景可以等同于该客户端发起该版本的写入请求，并且请求成功的情况。

【情况-3】如果R个回复中，，最大版本号vmax没有W个“提交/Commit”状态的版本，或其数据不一致，则需要等待更多副本的返回，直到满足【情况-1】或【情况-2】。

基于以上逻辑，可以设计一个额外特性：渐近读，即客户端在开始执行读操作时只尝试读取R个副本，如果没有收到全部的响应，或者R个响应的情况为【情况-3】时，再向其余的副本发送请求，直到满足【情况-1】或【情况-2】。

渐近读特性对于类似于缓存机制，如果读请求能够命中【情况-1】则可以降低集群的负担，否则的话则会造成请求延迟的增加。

## 3.3 删除流程

删除被视作一条特殊的写操作，其流程与写操作完全相同。

# 4 故障

对于整个分布式系统，可能的异常包括以下情况以及组合：数据不完整、数据位反转、数据包重复、数据包丢失、数据包超时、网络中断、并发请求/乱序、节点宕机、慢磁盘。

## 4.1 数据异常

**【数据不完整】**

数据不完整指的是在数据写入磁盘过程中发生异常，导致部分数据写入、部分数据没有完整写入的情况。

为了应对数据不完整的情况，系统要求本地存储支持事务接口，保证写入操作的原子性，类似ceph中的OS的设计。

**【数据位反转】**

数据位反转指的是由于磁盘的错误导致已经保存在磁盘上的数据的位发生了变化。

为了识别数据的位反转，每个对象的数据都包含其对应的CRC，当客户端读取数据的时候，可以基于其CRC进行校检，从多个副本中选择正确的数据。

另外，系统设计实现定时的检查机制，用于周期性的核对数据与其CRC。如果发现了位反转的数据，则通过其它有效副本进行修复。

## 4.2 网络异常

**【数据包重复】**

为了避免由于网络等原因导致的数据包重复，所有消息都设计为幂等的。

在对消息进行处理时，必须考虑重复消息的出现。而由于消息是幂等的，因此服务端对于重复的消息，只需要重新发送响应即可（通常需要记录WARN日志）

对于请求的发起端来说，由于所有消息都是幂等的，发起端可以在认定有必要的情况下，任意多次重试。

**【数据包丢失】**

由于所有消息都采用幂等的设计，因此任意网络消息可以重复多次发送。因此，当数据包发生丢失时，请求端可以在等待响应超时后重发请求。则服务端在收到重复的请求后，将再次返回响应。

**【数据包超时】**

数据包超时类似数据包重复，也可以通过重新发送请求来获取响应。不同的时候，超时的数据包最终达到接收端时，接收端需要对齐进行识别，并做抛弃处理。

实际上对于以上三种网络异常，都可以通过“消息幂等+重试”机制进行处理。

## 4.3 网络中断

一个osd的网络是否仍然处于连通状态由该osd与mon之间的心跳信息作为判断的依据。当一个osd仍然能够与mon通讯时，则认为该osd的网络是连通的，否则认为其网络已经中断。

当mon认定一个osd的网络中断后，将会触发OSDMap的变更，将osdmap中该osd的状态标记为down；并将根据配置决定是否在一段时间后将该osd的状态变更为out，即剔除集群。

为了防止osd的网络不稳定导致的短时间内osd多次上线/下线，只有当osd连接到mon并保持一段稳定的时间后（60s~5min），mon才会认为该osd的网络连接是良好的，进而触发OSDMap的变更。

对于网络脑裂的情况，即每个osd都与mon相同，但osd之间或osd与client之间无法通讯的情况，存在两种备选的策略：（1）报告异常，请求人工干预；（2）依据策略标记将部分osd剔除集群。

## 4.4 请求乱序

请求乱序主要主要是m个客户端同时发起请求的情况，考虑以下情况：

* client.1发送请求的顺序是(rep.a, rep.b, rep.c)
* client.2发送请求的顺序是(rep.b, rep.c, rep.a)
* client.3发送请求的顺序是(rep.c, rep.a, rep.b)

假设请求都是同时到达，则rep.a，rep.b，rep.c无法同时就任何请求达成一致，即Vote过程无法通过。

针对这种情况，采用以下两种策略：

（1）client发送请求的顺序必须严格按照PG对应的Rep/OSD的顺序。这样在一定程度上能避免乱序的问题。

（2）如果发生Vote不一致的情况，则以Rep序号小的请求为准。Rep编号大的将放弃自身的请求，并向对应的客户端返回错误。

## 4.5 节点故障

节点故障包括断电、重启、操作系统异常、磁盘IO异常、磁盘损坏等任何导致正常流程无法继续的异常。

根据写流程，节点可能的故障点如下图所示：



图 写入流程的故障点

我们将故障点进行初步的分析：

|  |  |  |
| --- | --- | --- |
| 故障点 | 描述 | 影响 |
| F1 | 未收到客户请求 | 未参与请求 |
| F2 | 接收到客户请求  未开始写入Prepare日志  写入Prepare未成功 | 未参与请求 |
| F3 | 写入Prepare日志完成  未能够成功发送出Vote消息 | 未参与请求 |
| F4 | Vote消息发送成功  未收到足够的Vote消息 | 待分析 |
| F5 | 收到足够的Vote消息  写入Commit消息成功  但未发出ACK消息 | 待分析 |

其中F4又可以分为以下两种情况：

* F4-a：发出的Vote消息未使得请求成功
* F4-b：发出的Vote消息使得整个请求成功

其中F4-a相当于“未参与请求”的情况。

综上所述，需要对三类错误进行分析，分别是：

**【F1、F2、F3、F4-a：未参与请求】**

当一个节点宕机，且其造成的影响相当于“未参与请求时”，剩余的节点如果满足W个，可以独立的达成该请求；如果剩余的节点不足W个，则请求将一直处于挂起状态，即由于没有副本可以收到W个Vote消息，导致请求的流程无法继续，整个请求流程挂起。

如果请求流程挂起，那么最终mon会在获取存在节点宕机后，发布新的OSDMap，推动相关的PG进入故障恢复流程。整个流程参考第五章：变更。

**【F4-b：发出的Vote消息使得整个请求成功，但未写入Commit日志】**

这种情况下，当该节点重新加入集群时，会通过集群其它相关副本的状态，确认本身的状态，从而使该请求继续完成。

**【F5：写入Commit消息，但是未发出ACK消息或发出的ACK消息丢失】**

这种情况下，如果有足够的ACK到达客户端，则客户端可以认为请求完成。如果没有足够的ACK达到客户端（说明超出了可用性设计范围），则客户端需要等待OSDMap变更后才能收到ACK请求。

当该节点重新加入到集群后，Commit的请求一定会成功，因此可以继续请求的流程。

实际上，任意数目的节点，在任意状态宕机，最终都将导致OSDMap的变化，从而导致“变更”流程，这一流程对于高阶故障的处理仍然有效。

对于变更的分析，详见第五章。

## 4.6 慢磁盘

对于磁盘异常缓慢的请款，暂时采用报告人工处理的方式。

后续可以设计实现自动减低权重、自动剔除集群等自动化的处理方法。

# 5 变更

OSDMap的变更是CCeph故障处理的核心，其设计思路参考ceph的OSDMap变更流程。

变更的实质是PG在osd之间的迁移和重建。

## 5.1 数据分布的变更

对于一个PG来说，当OSDMap发生变化时，其数据分布，即其所对应的osd也（可能）随之发生变化。

例如假设整个集群当前的OSDMap版本号为3，则对于PG.a来说，其分布相对于每一个OSDMap版本来说可能是：

Crush(PG.a, OSMap.v1) = (osd.0, osd.1, osd.2)，数据版本[v1,v2,v3,v4]

Crush(PG.a, OSMap.v2) = (osd.3, osd.1, osd.2)，数据版本[v5,v6,v7]

Crush(PG.a, OSMap.v3) = (osd.0, osd.1, osd.2)，数据版本[]

所以对于每一轮OSDMap来说，PG.a可能生活在不同的osd上。

以上是的osd.0宕机后又重新加入集群的典型场景（例1）。

## 5.2 PG的状态

**【Initial状态】**

初始状态，当集群的变化导致PG的分布发生变化时，PG都从此状态可出发。

PG会试图进入Active和Clean状态。

**【Active状态】**

PG可以接受请求的状态称之为Active状态。相对的，一个PG的状态机从Initial状态到Active的过程称之为ToActive过程。

PG在进入Active前，必须验证该PG中所有数据都是确定的。所谓“确定”，指的是对该PG内的对象的的数据是无歧义且可以获取的，否则该PG的数据即为丢失，PG不能自动进入Active状态。如果PG数据丢失，则需要人工干预。在手工认可数据的丢失后，PG可以进入Active状态。

简单来讲ToActive过程是确认PG数据状态的过程，但并不一定得到了获取了所有的数据。

**【Clean状态】**

Clean状态意味着该PG对应的OSD已经获取了该PG的所有数据，在今后的PG恢复过程中，只需回溯到当前这一轮的OSDMap即可。

Clean状态也意味着除了当前osd意外，其它osd可以删除该PG的数据。

整个PG的状态迁移如下图所示：



图 PG的状态迁移

## 5.3 PG状态的确定

一个PG能够进入Active，需要满足以下两个条件：

（1）该PG所有的数据状态都是确定的。

（2）该PG内的所有数据都是可以获取的。

即对于上例来说，如果PG.a在OSDMap.v3中能够进入Active状态，则v1-v7的版本都必须是确定的，并且可以访问到的。

一个数据版本是“确定的”，意味着：基于目前可以访问的副本，能够得知该版本是成功的或者不成功的，而不是一个不确定的状态。如果一个版本的状态是不确认的，则意味着发生了数据的丢失。

对于一个确定的数据版本来说，可以读取其任意Commit/Apply状态的副本获取其数据内容。

## 5.4 数据版本的确定

对于给定的OSDMap的版本e，和要修改的PG p，假设其副本数目是N，写操作要求的的数目为W，且W > N/2。

则对于p的版本v，其N个副本的可能的状态为：

（1）Commit状态：已经成功向磁盘写入Commit日志

（3）Prepare状态：已经成功写入Prepare日志，但未写入Commit日志

（4）UnPartIn状态：未参与，即没有持有任何关于v的信息

（5）UnKnown/None状态：未知，副本不可达，宕机等。

以上状态标记为SC，SP，SU和SN。

考虑到请求可能来来自于多个并发的客户端，所以SC和SP都与客户端相关。记对于来源于客户端i的请求对应的的版本为vi，则其相关状态为SCi和SPi；SU和SN与客户端无关，不做区分。

基于以上状态，设PG p的N个副本中，状态为SCi的副本数目为qi，状态为SPi的副本数目为pi，状态为SU的副本数目为x，状态为SN的客户端的副本数目为y，则有

则对于p的版本v，其所有可能的状态包括：

**【情景1】：存在Commit状态**

如果存在任意qi≠0，则根据写入流程，必然有：对于任意j ≠i，qj=0，即只可能有一个请求vi进入到Commit状态。对于这种情景，数据版本可以确认，即v=vi。

但此时并不能确认客户端是否获取了ACK消息，需要重新向客户端发送ACK。

**【情景2】不存在任意副本的状态为Commit**

对于这这种情况，满足。不妨假设max(pi) = m。

【情景2-1】 m ≥W

即存在≥W个副本为SPi，则版本vi被确认。

但版本vi并未执行成功，需要继续执行写入操作。

【情景2-2】m<W，m+q<W

没有任何vi可以成功, p的版本仍然为v-1。

同时可以确认所有SN状态的副本的状态不可能是SC。

【情景2-3】m<W，m+q≥W，q<W

没有任何vi可以成功, p的版本仍然为v-1。

最多有q个副本进入SC状态，客户端最多收到q个ACK，因此客户端不会认为Vi成功。

状态为SN的q个副本的vi版本的数据无效，重入集群时需要放弃。

【情景2-4】m<W，m+q≥W，q>W，等价于q>W

则p的版本v状态未知

处于SN状态的q个副本可以独立达成任何请求。

另外，根据数据的写入流程，如果版本v+1是确定的，则v必然是确定的。因此，对于任意一个OSDMap来说，PG的状态是否是确定的，取决于该PG相对于该OSDMap的边界数据版本是否是可以确认的。所谓边界数据版本，指的是当OSDMap发生变更时，该PG所处的数据版本。对于例1来说，边界版本是v4和v7。