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。

## 1.3 一致性

读写采用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。

# 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】**

为了保证数据一致性（不一定是强一致性），强制规定W必须大于N/2，即写操作必须完成半数以上的冗余，才能视为完成。相应的，客户端在发起一个请求时，必须收到W个Ack，才能认为才写操作完成。

**【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，（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。这是正常状态下的写入流程

**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。

**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、（3）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。

如果某一个Rep发现o的版本号并不相同，怎么处理？

（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。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。

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）

客户端在收到失败请求时，其原因应该是需要重试。完整的客户端逻辑是什么样子的？请求失败是什么流程？

## 3.2 读流程

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内容，返回以下信息：

（1）所持有的oid最新版本的版本号v

（2）所持有的oid最新版本的数据data\_v

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

对于副本来说，对以上内容的确认规则为：

（1）副本对于对象o的最新版本v的认定规则为：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）对于版本v的状态stat\_v的认定规则为：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，任意返回值的版本状态为“应用/Apply”时，可以立刻采用该数据作为该次读操作的结果。“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）。

【情况-2】如果R个回复中，有W个“提交/Commit”状态的版本，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操作情况。

【情况-3】如果R个回复中，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

## 4.1 数据不完整

**【消息的幂等】**

所有消息必须设计为幂等的，以防止由于网络故障产生重复的数据包，从而导致系统失败。

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，只需要简单抛弃即可（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。

消息中的OSDMap版本号

# 5 变更

**【PG ToActive】**

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。

# 4 性能