CoolCeph系统设计

袁冬 2017年11月

# 1 系统概述

CoolCeph是一个参考Ceph RADOS实现的分布式存储系统，旨在提供一种基于Object语义的通用的后端存储类型，以便在之上构建NAS、Block、S3等不同类型的业务存储系统。

CoolCeph主要针对Ceph做以下改进：

* 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

系统的关键设计选型包括：

* 采用MultiPAXOS作为数据一致性算法
* 采用PG作为数据分布单位，支持PG数目的按需增长（PG分裂）
* 支持不同类型的数据分布算法
* 支持ROW形式的对象级快照

## 1.1 数据模型

CoolCeph以对象为数据的基本单位，每一个对象包含以下内容

* 对象ID（oid）：采用字符串表示，用于表示一个对象。
* 对象内容（data）：二进制内容，支持随机读写
* 对象属性（attr）：对象的键值对集合，类似文件的扩展属性

每一个对象的ID必须是唯一的，对象和对象之间没有任何关系。该数据模型可以理解为下图：



系统后续将支持存储池，不同存储池，除了共享底层物理存储空间，其数据是完全对立的。

## 1.2 访问接口

系统支持对于对象创建、删除、读写、KV等操作，还支持对象级事务，即对于单个对象的多次修改（包括对内容和KV的修改），可以通过一次事务完成。

具体来说，系统对外提供的访问接口包括：

（1）对象操作接口

* 创建对象：touch(char\* oid)
* 删除对象：delete(char\* oid)

（2）对象内容接口

* 对象内容写：write(char\* oid, int64 offset, int64 length, char\* data)
* 对象内容读：read(char\* oid, int64 offset, int64 length, char\* buffer)

（3）对象attr接口

* 添加/更新attr：put\_attr(char\* oid, char\* key, char\* value)
* 读取attr：get\_attr(char\* oid, char\* key, char\* buffer)

（4）对象事务接口

对于单个对象的多个修改，可以通过一个事务完成，代码示例如下：

1. transaction t = begin\_new\_trasaction(oid)
2. t.write(offset, length, data)
3. t.put\_atrr(key, value)
4. submit\_transaction(t);

## 1.3 性能

系统面向的场景主要是大量随机IO的场景，因此性能采用4K随机写作为主要衡量指标。

系统在预期硬件（SSD+万兆网络）环境下的预期性能为：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 50分位 | 99分位 | 最大延迟 |
| 正常场景 | 1ms | 3ms | 10ms |
| 单点故障场景 | 1ms | 5ms | 50ms |
| 多点故障场景 | 10ms | 100ms | 1s |

表格内为再次读/再次写性能，单位是毫秒，理论上初次读写性能与此相同。通常读性能要优于写性能，但考虑到缓存影响，并不将读性能作为衡量指标，但要求读性能不能低于写性能。

以上指标中，平均延迟和99分位延迟为核心指标。前者表示系统的直观性能，后者则表示系统最差情况下的表现。对于整个系统的性能监控和统计中，采样周期不大于1分钟，理想情况是10s。

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。

对于吞吐量来说，通过理解为对于大于1MB以上的请求，可以达到硬件（主要是网络带宽）的极限。

## 1.4 一致性

系统对于对象的读写，提供强一致性保证。

即对于对象的读写操作，假设数据x的当前版本是v0，则系统保证以下读写一致性：

1. 对于修改请求v1，如果系统已经给出ack，则系统确保后续所有对x的读操作返回的至少是v1之后的版本。
2. 对于修改请求v1，如果系统没有给出ack，则此时有可能读到v0版本，也可能读到v1版本。但是系统保证，如果返回了v1版本，则后续所有请求都返回至少v1版本，即不会存在脏读。

## 1.5 可靠性

提供9个9的可靠性，基于单个物理机房、三副本、多Tor数据分布。

数据丢失的定义：即一年内存在任意一个对象丢失的概率。

## 1.6 可用性

提供5个9（99.999%）的可用性。

可用性采用时间作为单位来定义，即 可用性 = 可用时间/总时间。

通常总时间采用一个月为单位，可用时间的颗粒度理想状态下采用1s为单位。出于统计的考虑，也可以采用10s或者1分钟作为单位。

则单位时间可用的标准为：

* 所有系统收到的请求都成功
* 所有系统收到的请求延迟都满足标准（按照正常状态计算）

以上两条中任意一条不满足，则该时间段视为不可用，即可用时间减少了1s。

则，按照5个9的可用性，1个月内的不可用性时间为25秒，1年的不可用时间为300秒。

## 1.7 限制

（1）集群规模

单个集群支持1000台物理机，单台物理机支持两种目标型号：

* SSD型：12 \* 8TB，集群物理容量96PB
* SATA型：36 \* 8TB，集群物理容量288PB

支持集群的平滑扩容，初始集群大小为9台。磁盘利用率大于90%。

（2）环境要求

目标软硬件环境为

|  |  |  |
| --- | --- | --- |
|  | 配置 | 备注 |
| CPU | Intel Xeon 2630V3+ | SSD机型需要较好的CPU |
| 内存 | 64GB+ | 更大的内存将有利用缓存 |
| 存储 | SSD支持 12 \* 8TB  SATA支持 36 \* 8TB | 视存储引擎的情况支持NVME和SDPK |
| 网络 | 10Gb \* 2 或 25Gb \*2 | 支持RDMA网络 |
| 其它 | Intel Optane | 视存储引擎支持情况而定 |
| OS | CentOS 7 x64 | 系统实现为用户态 |

# 2 系统架构

## 2.1 系统架构



系统以lib的形式提供SDK给业务端，业务端需要将改lib编译连接到自己的程序中，然后调用SDK的接口向集群发起IO请求。

系统有两个核心的组件：DataNode和NameNode。其中DataNode即为实际的存储数据服务器；NameNode则负责DataNode的管理，包括DataNode的加入、离开、负载均衡等。

通常情况下，SDK在启动的时候会与NameNode进行通讯，获取当前集群中所有DataNode信息，并缓存到本地，此后的所有请求都直接发往DataNode。只有当集群的DataNode发生变化时，例如扩容新的DataNode，SDK才需要再次与NameNode进行通讯，以便获取新的DataNode集群信息。

DataNode管理其所在服务器上的磁盘，并处理SDK发起的数据读写请求。DataNode也需要从NameNode获取到视图，以便确认自身仍然作为一个被认可的节点提供服务。DataNode在故障恢复等场景下只会相互之间进行通讯，以便完成数据缺失副本的重建等操作。

NameNode采用Active-Standy的高可用方式，实现中依赖于外部的Zookeeper集群（后续可以可能考虑去除对ZK的依赖）。

## 2.2 功能模块

### 2.2.1 SDK

SDK以库的方式（cceph.h, cceph.a, cceph.so）提供给业务端。业务端需要将其编译链接进业务程序的内部。

当需要访问集群时，需要调用SDK的初始化方法对其进行初始化，然后即可对集群发起IO请求。当不在需要访问集群时，需要调用销毁方法SDK所占用的相关资源。

### 2.2.2 DataNode

DataNode是提供实际处理客户端发送的IO请求的节点。每一台物理机通常运行一个DataNode服务，管理该机器上所有的磁盘资源。

集群中的对象根据数据分布算法均匀的分散到集群中所有的DataNode上，当SDK发起的请求达到DataNode的时候，DataNode会首先判断请求的对象是否属于当前DataNode负责，如果是，则负责该请求的处理，并在处理完成后，向SDK返回ACK；如果不是，则拒绝该请求。

当对集群的DataNode进行扩容或者集群中DataNode由于故障等原因退出集群时，DataNode需要根据变更后的数据分布情况进行数据恢复操作，以便完成数据副本的重建，并保证对象的分布符合集群数据分布算法给出的预期结果。

### 2.2.3 NameNode

DataNode负责集群内DataNode的管理，即根据DataNode的扩容和移除，更新集群的视图以及对应的数据分布情况。

SDK和DataNode在需要集群视图的场景，例如新加入集群或者视图更新的情况下，都需要跟NameNode进行通讯，已获取最新的集群视图。

NameNode采用Active-Standy的方式实现高可用，支持一主多从。当前Active故障的情况下，通常需要10~30秒的时间完成Standy的切换操作。

由于NameNode只负责DataNode的管理，因此NameNode不可用的情况下，只会影响DataNode的加入和离开，并不影响数据的IO操作。

### 2.2.4 Zookeeper

外部的ZK集群，主要用户实现NameNode的高可用，还用于保存集群视图等少量的元数据信息。实际生产中，通常包括三个或者五个独立的Zookeeper服务器，部署不同的TOR之下。Zookeeper可以和NameNode混布。

# 3 数据分布

## 3.1 数据分布

为了解决对象副本间的数据同步问题，系统在对象和DataNode之间引入一个中间层，称之为PG（Placement Group）。PG表示一个副本组，即处于同一个PG中的对象，其数据分布是一致的。

对象先映射到PG，然后PG再映射到DataNode，如下图所示：



PG可以理解为一个逻辑概念，即对象的集合。PG的多个实例，即其多个副本，是完全对等的，互相之间通过MultiPAXOS协议保证一致性。

### 3.1.1 对象映射到PG

系统当前的PG数目是相对静态的，通常可以认为是不变地，例如256个，因此对象映射到PG的过程采用直接HASH，即

pg\_id = hash(oid) % pg\_num

由于对象数目远多于PG的数目，因此可以认为对象是均匀的分布在所有PG中的。

### 3.1.2 PG映射到DataNode

PG到DataNode的映射通过集群视图完成，集群视图可以理解为一个查找表，其中包含了每一个PG对应的DataNode的信息。

集群视图的相关信息参考3.2节。

### 3.1.3 PG的数目

PG的数目通常可以认为是固定的，仅当集群节点大幅度增加，以至于无法满足负载均衡要求时，才需要扩展PG的数目。

PG数目的扩展操作采用PG分裂的方式进行， 即原来的一个PG.X将分裂为两个子PG：PG.X1和PG.X2。PG分裂由于涉及PG之间的数据迁移，对集群的IO能力是有损的，因此需要尽量避免。

为了满足负载均衡和故障恢复的要求，通常要求每个磁盘分配256个PG，每台物理机12块磁盘，则为3072个PG。实际生产中需预留一部分buffer，则建议每个物理机4096个PG分配。当集群扩容导致每台物理机的PG数目小于4096时，建议进行PG的分裂操作。

另一方面，为了避免频繁的PG扩容导致集群服务能力的损失，建议采用翻倍PG数目策略，即

（1）集群初始化时指定的PG数目 = 物理机 \* 4096 \* 2；

（2）每次PG分裂时，对每一个PG都执行分裂操作，使PG数目翻倍。

这样可以保证只有当集群容量翻倍时，才需要执行PG分裂操作，也有助于更好的负载均衡。

## 3.2 集群视图

系统的集群视图描述了PG到DataNode之间的映射关系，即集群的数据分布。

当且仅当集群的数据分布（特别PG到DataNode之间的映射关系）发生变化时，集群的视图才会发生变化。反之，集群内任意数据分布的变化，包括集群扩容、节点故障、PG分裂、从新负载均衡等导致的各类数据重新分布的场景，都通过集群视图的更新完成。

集群视图采用版本（epoch）的组织方式，每一个版本的视图表示了当前版本下，集群所期望的数据分布状态，即目标状态。

集群视图不管理数据变更的过程，数据变更的过程由DataNode根据其所持有的数据的当前状态和集群视图给出的目标状态计算得出。

### 3.2.1 集群视图的内容

集群视图中包括以下主要内容：

* 集群视图版本号（epoch）：单调递增
* DataNode信息：包括每个DataNode的IP:Port、容量、状态等
* PG信息：包括每个PG的ID、数据分布（所在DataNode）等

其中DataNode的状态包括：

* Wait：DataNode已经启动，但是尚未加入到视图中，不提供服务
* Active：DataNode已经启动并且加入到Hash环中，正常服务状态
* Expired：DataNode被认定为超时，但还未进行数据分布的变更

### 3.2.2 集群视图的生成

集群视图的生成由NameNode负责，NameNode会根据集群内的PG数目、DataNode的状态和容量等信息，计算出每一个PG的实例所在的DataNode。

需要注意的时，PG实例对应的DataNode是有序的。

具体生成算法可以是基于容量的负载均衡，也可以是一致性HASH。(暂未确定)

### 3.2.3 集群视图的变更

集群视图的作用是用于表示数据分布状态，因此需要重新分布数据的场景会导致集群视图的变更，包括以下可能的场景：

* DataNode离开（物理机故障、磁盘故障）
* DataNode加入（集群扩容）
* PG分裂
* 重新负载均衡

当集群视图需要变更时，由NameNode根据目标状态的PG和DataNode信息，计算出PG的数据分布，从而得到完整的集群视图，并使得集群的epcoh +1。最后，NameNode将新生成新的集群视图分发给集群内的所有DataNode和所有连接到集群的SDK。

集群视图分发过程中，由于集群将短暂的处于视图不一致的状态，因此集群的IO将受到影响。具体影响大小根据视图分发速度和集群规模而定。后续将通过AB切换的方法尽可能的降低该影响。

# 4 存储引擎

存储存储引擎负责单机物理磁盘资源的管理，包括数据的组织和IO服务。

在系统中，通常一个物理磁盘（卷）对应一个存储引擎，该引擎管理了所在磁盘的所有物理空间，并能够充分发挥磁盘的性能。

DataNode通过存储引擎接口访问存储引擎，而不依赖于引擎的具体实现。不同的物理硬件可能需要不同的存储引擎实现。

存储引擎本身相当于在将磁盘转化为一种单机对象存储系统（类似文件系统），可以作为一个独立的组件使用。

## 4.1 数据模型

存储引擎同样采用对象语义模型，同时支持集合的概念，即每个对象隶属于一个本地集合（Collection）。 该数据模型可以理解为一个支持一层目录的文件系统，对象相当于文件系统中的文件。

引擎对象类似于系统对象，包括对象ID、对象内容和对象属性。通常不严格区分系统对象和引擎对象。集合是对象的容器，每一个集合包括集合名称和集合属性（KV对集合）。对应到整个系统设计中，一个PG即对应存储引擎的一个集合。

## 4.2 访问接口

存储引擎接口主要有三类操作组成，分别是集合类操作、对象类操作和日志类操作。

（1）集合操作

包括集合列表、创建集合、删除集合、集合属性读写等。

（2）对象操作

包括对象创建删除、对象内容读写、对象属性读写等。

（3）日志操作

包括读取事务日志、写入事务日志、控制日志自动回放、更新CheckPoint等操作。

（4）控制类操作

主要是引擎的格式化（Format）和挂载（Mount）操作。

存储引擎对于同一个集合内的对象的修改，包括对同一个对象不同部分（内容、属性）的修改通过事务保证其原子性。实际上，存储引擎的对象类操作通过事务的方式对外提供修改操作接口。

集合类操作和日志类操作通常只服务于集群的故障恢复、PG分裂等数据发生重新分布的场景。正常IO情况下，引擎的调用方只会通过对象类的操作进行数据的读写。

## 4.3 集合日志

为了支持分布式层面的故障恢复机制，存储引擎除了提供基于集合和对象的事务接口之外，还支持直接更底层的事务接口，即访问接口中的日志类操作。

对于每一个对象集合（Collection），调用者可以通过日志接口直接访问其事务日志区域，包括读写指定的日志条目、更新CheckPoint的位置等。

日志区域与数据区域的关系如下图所示：



通常情况下日志区域的内容不需要调用方关心，日志的内容会自动应用（Apply）到数据区域，同时CheckPoint也会自动更新，小于CheckPoint的日志条目会被自动清理（Trim）。

对于故障恢复等场景，DataNode可以通过日志类操作对以上行为进行控制，具体包括以下访问接口。

### 4.3.1 事务日志的读写

存储引擎提供结构允许直接读写 PG的某条事务日志，对应的接口为：

* read\_transaction\_log(pg\_id, tran\_log\_id, buffer)
* write\_transaction\_log(pg\_id, tran\_log\_id, data)

其中buffer和data都是串行化后的日志内容（char\*）。

通过事务日志的读写接口，PG实例之间可以直接进行日志的同步操作，即从某个PG实例中读取一条日志，然后写入到另一个PG实例中。

“写入事务日志” 只能在“日志自动回放”开关处于关闭状态下才能够调用，当“日志自动回放”处于打开状态下，调用该方法写入该PG的事务日志将返回失败。

### 4.3.2 CheckPoint

存储引擎提供

### 4.3.3 日志回放控制

日志类结构接口的具体使用方法参考故障恢复和PG分裂相关章节。

## 4.4 引擎实现

由于DataNode只依赖于存储引擎的接口访问，因此引擎的实现可以独立演化。

目前计划存储引擎分为两期实现，第一版实现基于文件系统的引擎（FileBasedEngine），后续演化为直接基于磁盘的引擎（RawDiskEngine），后者将支持SPDK等较为先进的特性。

### 4.3.1 FileBasedEngine

基于文件系统的存储引用类似Ceph的FileStore，采用文件系统作为数据组织方式，一个集合对应一个本地目录，而对象就是存储在目录中的文件。

FileBasedEngine通过额外的文件或者裸块设备实现日志机制，具体设计参考对应的详细设计文档。

### 4.3.2 RawDiskEngine

直接基于裸磁盘实现存储引擎的接口，以便绕过文件系统，提供更好的性能。

具体设计待定。