CoolCeph系统设计

袁冬 2017年11月

# 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

### 4.1.1 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。

### 4.1.2 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。

## 4.2 事务日志

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。

日志区域与数据区域的关系如下图所示：



当Collection接收到修改操作时，为了保证修改操作的原子性，都会先写到事务日志中，然后应用（Apply、又称转储）到实际的数据区域。

事务日志可以理解为一个队列，其中每一条日志存在一个编号，该编号是连续自增的。事务日志中：（1）下一条要写入的日志的编号称之为CurrentPoint；（2）已经转储完成的日志编号称之为AppliedPoint；（3）可以删除的日志编号称之为CheckPoint。通常情况下，CheckPoint等于AppliedPoint。

大多数情况下日志区域的内容不需要调用方关心，存储引擎会将日志的内容会自动应用（Apply）到数据区域，同时CheckPoint也会自做对应的更新，小于CheckPoint的日志条目会被自动清理（Trim）。

对于故障恢复等场景， 调用者可以通过日志接口直接访问其对象集合（Collection）事务日志区域，包括读写指定的日志条目、更新CheckPoint等，以及对日志的回放行为进行控制。

### 4.2.1 日志读写

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

其中buffer和data都是串行化后的日志内容，即二进制blob数据。

“写入事务日志” 只能在“日志自动回放”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。

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。

### 4.2.2 日志转储

对于对象集合的所有修改，都通过事务的方式完成，因此修改操作将先写入到集合的日志区域，然后再应用到数据区域。事务日志应用到数据区域的过程称之为转储。

事务的转储操作按照事务日志的序号进行，因此要求事务一直必须是连续递增的。

### 4.2.3 日志清理

CheckPoint表示当前可以清理的事务日志的位置，事务日志编号小于CheckPoint的日志条目可以被清理掉，以便释放空间给后续的事务日志条目。

正常情况下，所有已经Apply到数据区域的事务日志都可以被清理掉，因此CheckPoint等于AppliedPoint。

但是为了在PG的不同副本之间进行数据的同步，因此可能需要保留一部分已经转储到数据区域的日志条目。

为了实现这个目的，存储引擎提供以下接口用于操作CheckPoint：

* set\_check\_point\_update\_mode(coll\_id, auto|manual)
* get\_check\_point(coll\_id, \*tran\_log\_id)
* set\_check\_point(coll\_id, tran\_log\_id)

set\_check\_point\_update\_mode用于设置CheckPoint的更新模式。auto表示自动更新模式，处于该模式下，存储引擎会自动将CheckPoint设置为AppliedPoint。manual表示手工更新模式，处于该模式下，存储引擎不会更新CheckPoint，而需要调用方（DataNode）通过set\_check\_point来更新CheckPoint模式。默认情况下处于auto模式。

get\_check\_point和set\_check\_point两个接口用于获取和设置当前的CheckPoint，需要注意的是，set\_check\_point只有在manual模式下才能调用，且设置的tran\_log\_id不能小于当前的check\_point。

在整个系统中，CheckPoint的更新模式处于manual状态，只有当确定了PG的所有副本都已经将日志写入到磁盘后，才会更新CheckPoint。

## 4.3 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。