Table of Contents

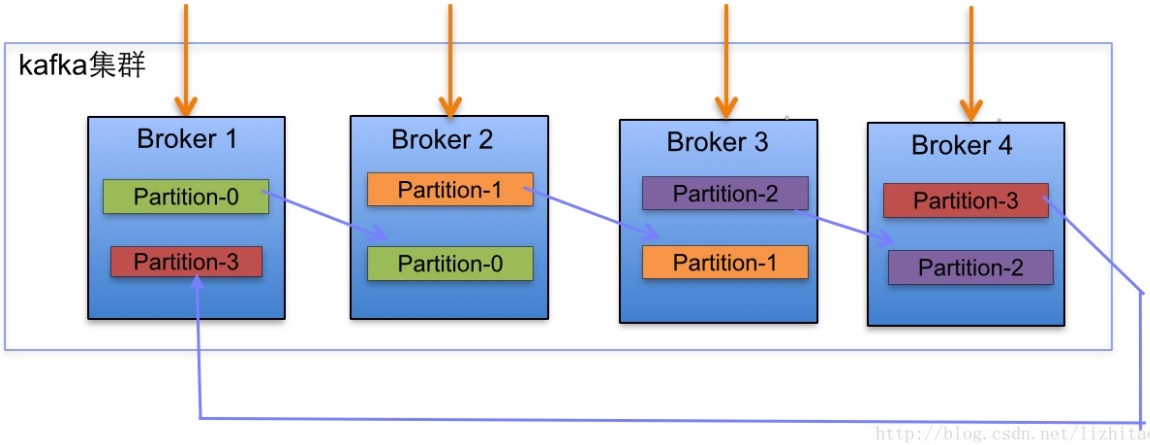
简介

分布式系统的学习笔记

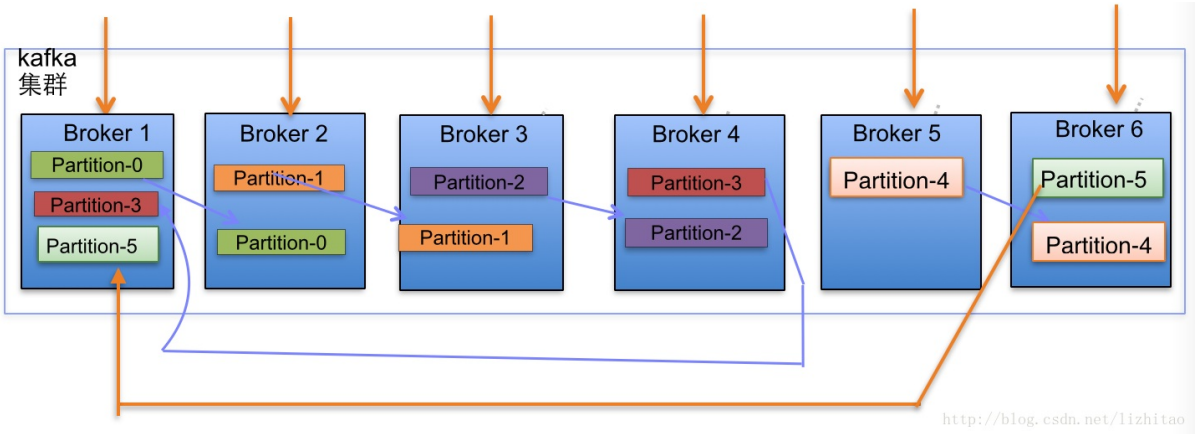
kafka

角色

- producer
- consumer
- broker 多个partition分布在多个broker上
- partition 一个partition存在所有的副本上 示例为2个副本，4个partition，4个broker



示例为2个副本，6个partition，6个broker



- replication 一个副本有所有的partition
    - 副本分配逻辑规则如下： 在Kafka集群中，每个Broker都有均等分配Partition的Leader机会。 上述图Broker Partition中，箭头指向为副本，以Partition-0为例:broker1中parition-0为Leader，Broker2中Partition-0为副本。 上述图种每个Broker(按照BrokerId有序)依次分配主Partition,下一个Broker为副本，如此循环迭代分配，多副本都遵循此规则。
    - 副本分配算法如下： 将所有N Broker和待分配的i个Partition排序. 将第i个Partition分配到第(i mod n)个Broker上. 将第i个Partition的第j个副本分配到第((i + j) mod n)个

Broker上.

- topic 对应多个partition
- offset
- leader
- ISR

副本同步

Producer在发布消息到某个Partition时，先通过Zookeeper找到该Partition的Leader， 然后无论该Topic的Replication Factor为多少（也即该Partition有多少个Replica）， Producer只将该消息发送到该Partition的Leader。

Leader会将该消息写入其本地Log。每个Follower都从Leader pull数据。这种方式上，Follower存储的数据顺序与Leader保持一致。Follower在收到该消息并写入其Log后，向Leader发送ACK。为了性能考虑,每个**follower**当消息被写到内存时就发送**ack**(而不是要完全地刷写到磁盘上才ack). 一旦Leader收到了ISR中的所有Replica的ACK，该消息就被认为已经commit了，Leader将增加HW并且向Producer发送ACK。为了提高性能，每个Follower在接收到数据后就立马向Leader发送ACK，而非等到数据写入Log中。因此，对于已经commit的消息，Kafka只能保证它被存于多个Replica的内存中，而不能保证它们被持久化到磁盘中，也就不能完全保证异常发生后该条消息一定能被Consumer消费。但考虑到这种场景非常少见，可以认为这种方式在性能和数据持久化上做了一个比较好的平衡。在将来的版本中，Kafka会考虑提供更高的持久性。 Consumer读消息也是从Leader读取，只有被commit过的消息（offset低于HW的消息）才会暴露给Consumer。Kafka的复制机制既不是完全的同步复制，也不是单纯的异步复制。事实上，同步复制要求所有能工作的Follower都复制完，这条消息才会被认为commit，这种复制方式极大的影响了吞吐率。而异步复制方式下，Follower异步的从Leader复制数据，数据只要被Leader写入log就被认为已经commit，这种情况下如果Follower都复制完都落后于Leader，而如果Leader突然宕机，则会丢失数据。而Kafka的这种使用ISR的方式则很好的均衡了确保数据不丢失以及吞吐率。Follower可以批量的从Leader复制数据，这样极大的提高复制性能（批量写磁盘），极大减少了Follower与Leader的差距。

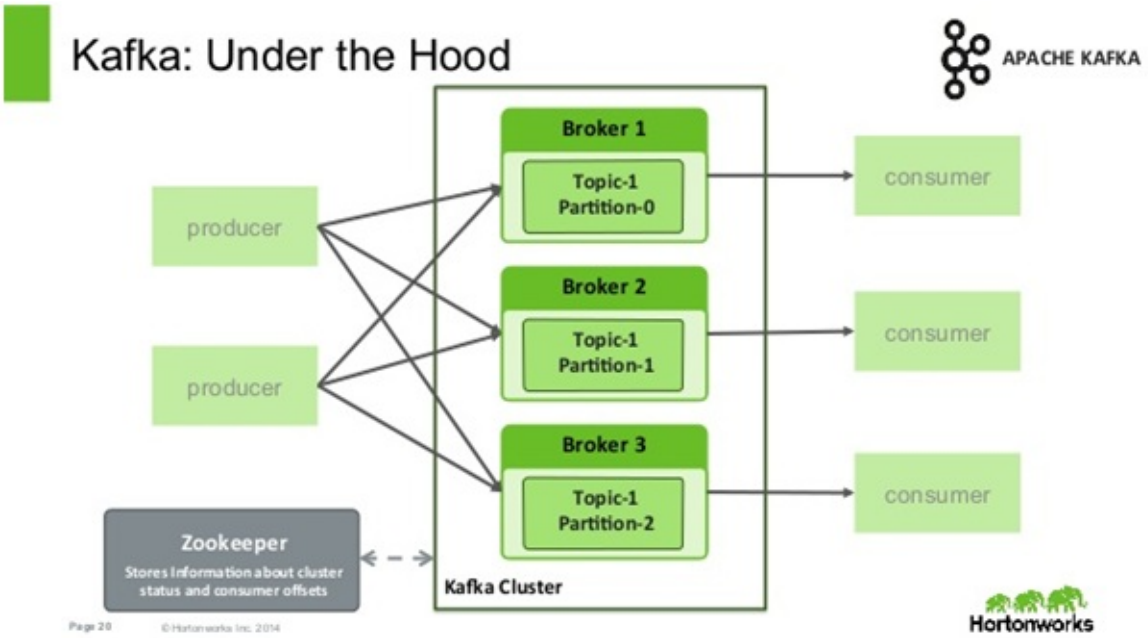
为了简单起见,只有leader可以提供读消息的服务.并且最多只到hw位置的消息才会暴露给客户端.

- ISR

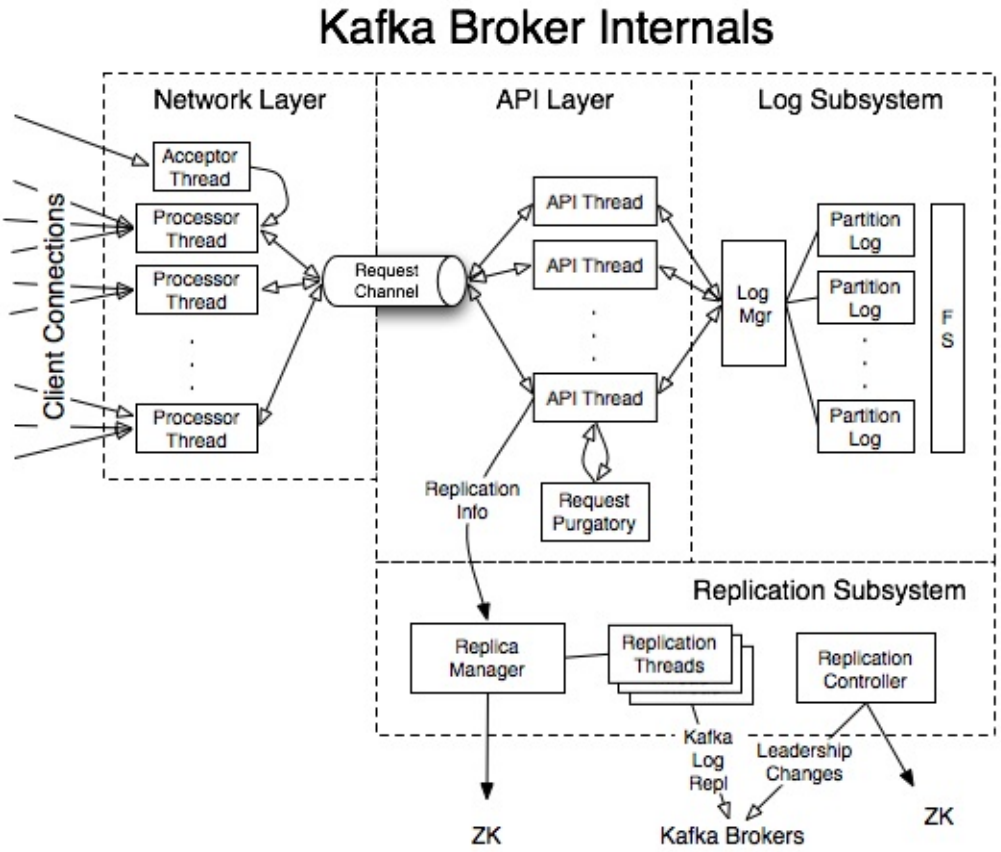
  如果失败的follower恢复过来,它首先将自己的日志截断到上次checkpointed时刻的HW.

架构

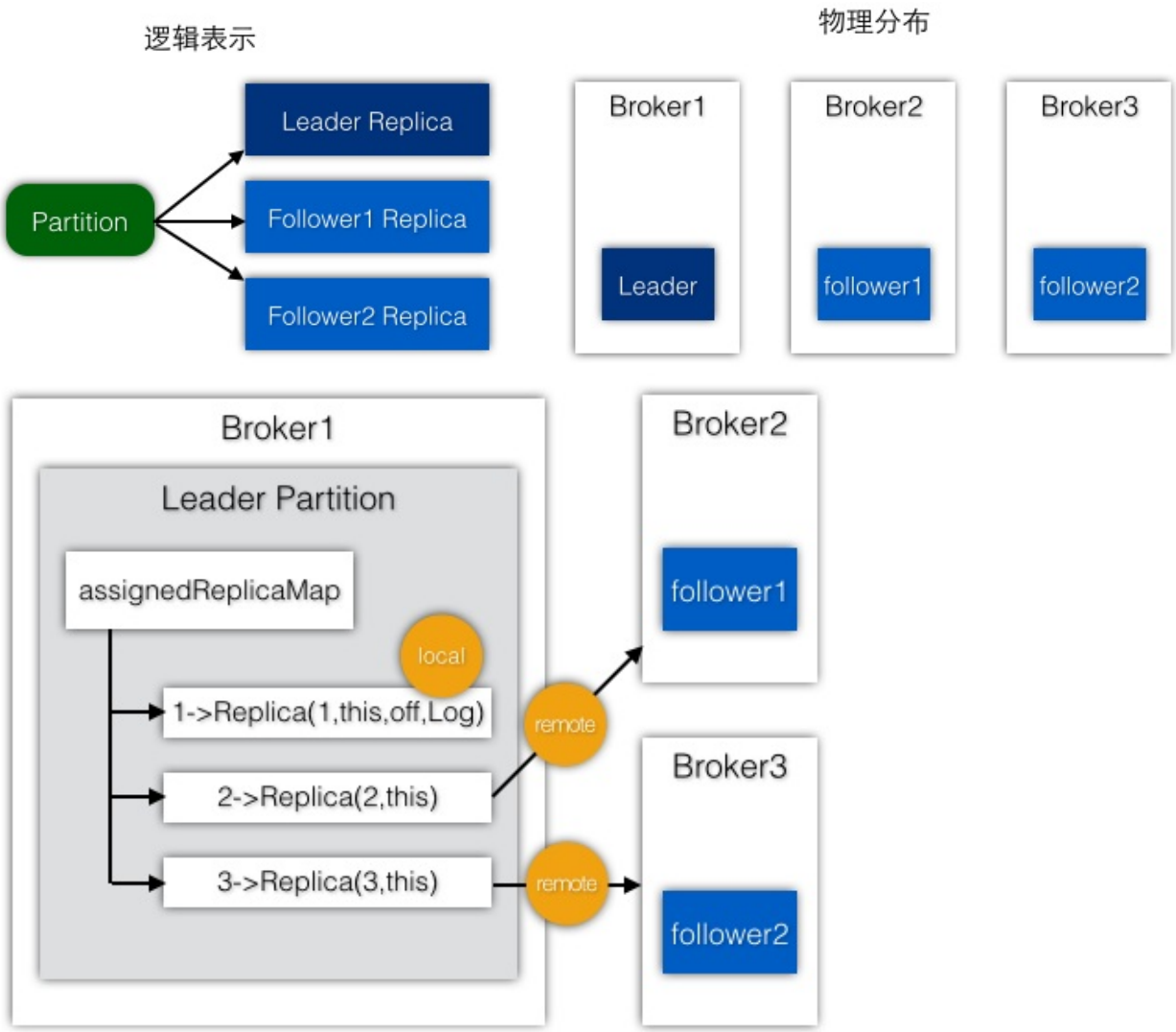
producer-consumer-cluster



broker



replication



总结

- Partition副本由Leader和follower组成,只有ISR列表中的副本是仅仅跟着Leader的
- Leader管理了ISR列表,只有ISR列表中的所有副本都复制了消息,才能认为这条消息是提交的
- Leader和follower副本都叫做Replica,同一个Partition的不同副本分布在不同Broker上
- Replica很重要的两个信息是HighWatermark(HW)和LogEndOffset(LEO)
- 只有Leader Partition负责客户端的读写,follower从Leader同步数据。类似**mongodb**
- 所有Replica都会对HW做checkpoint,Leader会在follower的拉取请求时广播HW给follower

rabbitmq

概念

基础技术

架构

riak

概念

基础技术

架构

rocksdb

针对leveldb的改进

- rocksdb支持在key的sub-part上设置Bloomfilter，这使得范围查询成为可能

问题

- **compaction**不可控，当L0文件达到12个, 而compaction来不及的时候, 写入完全阻塞, 这个阻塞时间可能长达10s。LevelDB实现上是L0达到4个时开始触发compaction, 8个时开始减慢写入, 12个时完全停止写入. 具体配置是写死的, 不过可以在编译时修改。但是
  1. 一旦写入速度>compaction速度, 不论这几个阈值设置多大, L0都迟早会满的.
  2. 阈值调大会导致数据都堆积在L0, 而L0的每个文件key范围是重叠的, 意味着一次查询要到L0的每个文件中都查一下, 如果L0文件有100个的话，这大约就是100次IO, 读性能会急剧降低.

     实际上, RocksDB的 Universal Style 就是把所有的数据都放在L0, 不再做compaction, 这样显然没有写放大了,但是读的速度就更慢了, 所以限制单个DB大小小于100G, 而且最好在内存.
- 写放大。基准数据100G的情况下, 50K的value, 用200qps写入, 磁盘带宽达到100MB\/s以上。眞实写入数据大约只有50K*200=10MB\/s, 但是磁盘上看到的写大约是10-20倍, 这些写都是compaction在写,此时的性能瓶颈已经不是CPU或者是LevelDB代码层，而是磁盘带宽了, 所以这个性能很难提上去,而且HDD和SSD在顺序写上性能差别不大, 所以换SSD后性能依然很差

参考资料

- 对LevelDB的“升级版”存储引擎RocksDB的调研成果
- leveldb和rocksdb在大value场景下的一些问题
- rocksdb 架构
- rocksdb blog

- lsm和fractaltree的对比
- yc上rocksdb的讨论
- leveldb代码分析

redis

源码剖析

redis设计与实现

- skiplist
- dict
- 压缩列表
- 对象

nosqlfans系列文章

> Redis 的源码只有2万来行，个人觉得是一个非常合适的学习Unix 环境下C语言编程的实例教材。而读源码，也对了解Redis内部结构很有帮助。
>
> 下面推荐的几篇文章，来自阿里巴巴云计算运维部的 hoterran 同学的个人博客，分别对Redis几个重要流程的源码进行了分析研究，对了解Redis内部结构很有帮助。
>
> 1.REDIS源代码分析 – HASH TABLE
>
> Redis的Hash Table 在源码里对应的是其dict结构（字典结构），本文内容介绍了Redis在hash table的结构，产生hash冲突的解决方法，以及非常Redis非常重要的rehash操作过程。
>
> 2.REDIS源代码分析 – EVENT LIBRARY
>
> 本篇文章主要介绍了Redis的异步网络事件驱动库，主要介绍了Redis使用它来实现非阻塞的网络事件处理的过程。包括了采用此库实现的Redis中各种定时器的原理。
>
> 3.REDIS源代码分析- REPLICATION
>
> 本文介绍了Redis的主从同步策略及原理，介绍了Redis在主从同步时的一些内部命令和内部状态切换。
>
> 4.REDIS源代码分析 – PERSISTENCE
>
> 此文介绍了Redis的 dump.rdb 定时镜像及 aof 日志型备份的实现原理。
>
> 5.REDIS源代码分析 – PROTOCOL
>
> 本文介绍了Redis在处理网络请求的过程中对Redis协议的分析，介绍了Redis Client对象对客户端命令的解析过程及处理流程。

Redis核心解读系列

> Redis是知名的键值数据库，它广泛用于缓存系统。关于Redis的信息已经不用我多介绍了。这个系统的Redis文章主要从另外一个角度关注，Redis作为一个开源项目，短短2W行代码包含了一个健壮的服务器端软件的必需，我们从Redis中可以学习C语言项目的编程风格、范式，学习类Unix下的系统编程，还有对于一个常驻服务的健壮性考虑等等。
>
> 对于一个C语言的初学者来说，学习一个类似Redis这样不大不小的项目是非常好的选择。Redis既没有Nginx深入性能细节的晦涩编码方式，又具备了一个性能敏感应用的C项目编程方式，是一个非常适合入门的项目。
>
> Redis核心解读系统来自于本人对于Redis的学习和总结，不同于Redis 设计与实现(对于这本书的作者表示非常佩服，能写出如此漂亮，详细的Redis解读)这个Redis代码注释方式 详细解读，本系列主要是选取精彩代码和关键路径进行解读，带领进入Redis的核心内容。并且会着重介绍Redis实现上的Hack写法。另外，本人对于Redis的某些设计也有独特见解，特别是对Redis的集群分发管理上，见相关文章。
>
> Redis核心解读系列主要有以下内容:
>
> 解读ae事件驱动库
>
> 解读Redis dict核心数据结构
>
> 解读Redis中ziplist、zipmap、intset实现细节
>
> 解读Redis运行核心循环过程
>
> Redis核心解读-从Master到Slave的Replicantion
>
> Redis核心解读–类型系统解构
>
> Redis核心解读–数据持久化过程与RDB文件
>
> Redis核心解读–AOF与REWRITE机制
>
> Redis核心解读–Slow Log
>
> Redis核心解读–事务(Multi和CAS)的实现
>
> Redis核心解读–pubsub(发布者-订阅者模式)的实现
>
> Redis核心解读–集群管理工具(Redis-sentinel)
>
> Redis集群的讨论及WheatRedis说明

核心数据结构

```
typedef struct dictEntry {
    void *key;
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
    } v;
    struct dictEntry *next;
} dictEntry;

/* This is our hash table structure. Every dictionary has two of this as we
* implement incremental rehashing, for the old to the new table. */
typedef struct dictht {
    dictEntry **table;
    unsigned long size; //dictEntry数组大小
    unsigned long sizemask;
    unsigned long used; //所有元素的数量，包含链表元素
} dictht;

typedef struct dict {
    dictType *type;
    void *privdata;
    dictht ht[2];
    int rehashidx; /* rehashing not in progress if rehashidx == -1 */
    int iterators; /* number of iterators currently running */
} dict;
```

cluster

Compared with Twemproxy and Redis Cluster

	Codis	Twemproxy	Redis Cluster
resharding without restarting cluster	Yes	No	Yes
pipeline	Yes	Yes	No
hash tags for multi-key operations	Yes	Yes	Yes
multi-key operations while resharding	Yes	-	No(details)
Redis clients supporting	Any clients	Any clients	Clients have to support cluster protocol

"Resharding" means migrating the data in one slot from one redis server to another, usually happens while increasing/decreasing the number of redis servers.

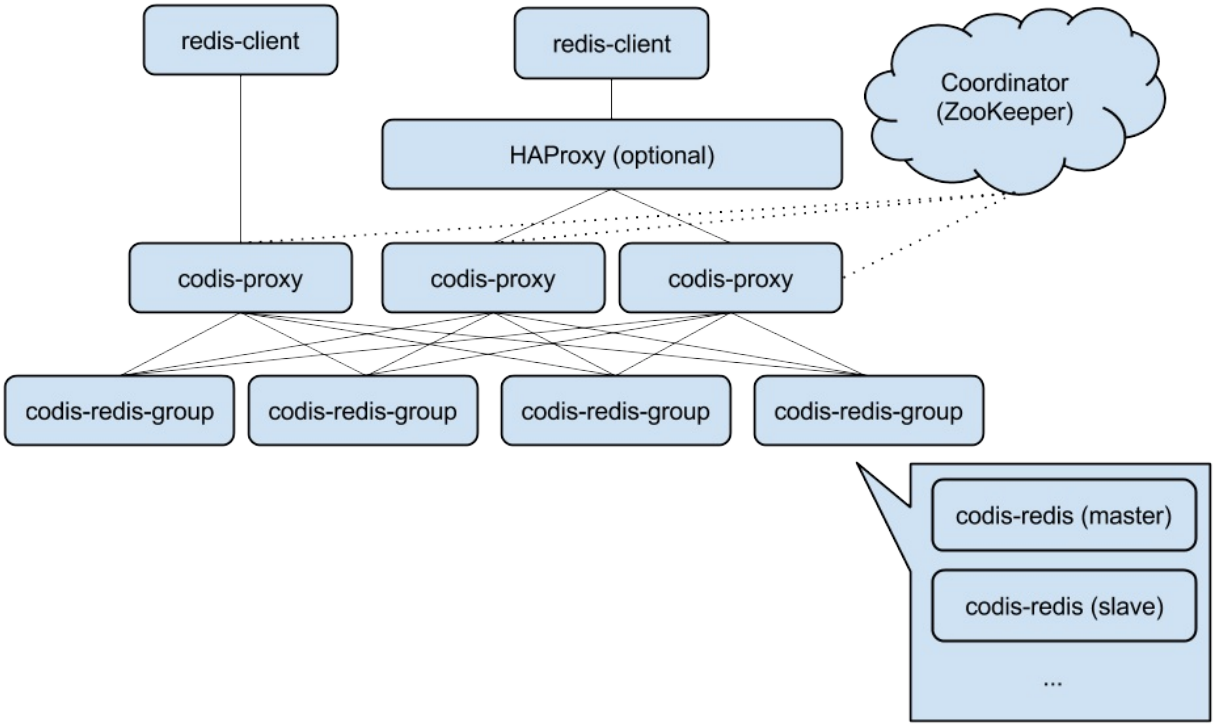
Twemproxy

Twemproxy 代码

Twitter使用redis的经验

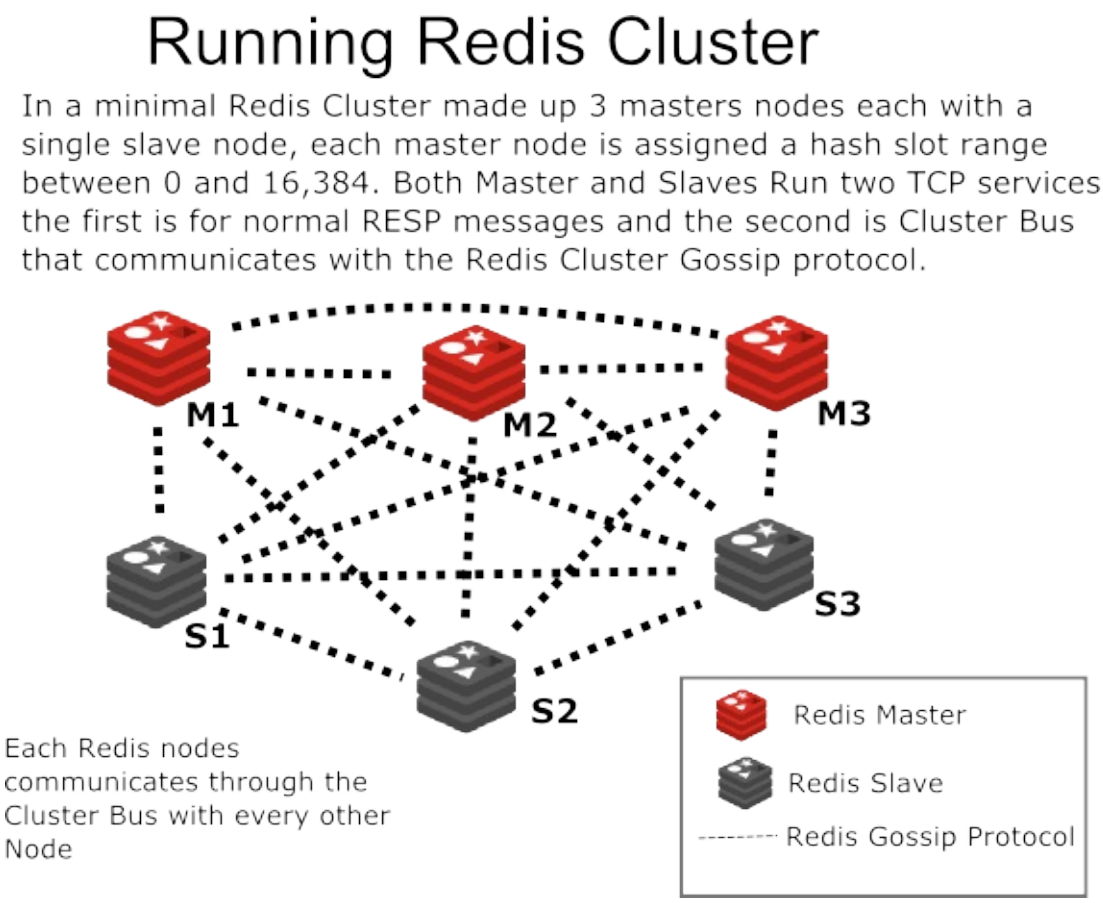
codis

architecture



redis cluster

Redis Cluster原理



Redis Cluster 是Redis的集群实现，内置数据自动分片机制，集群内部将所有的key映射到16384个Slot中，集群中的每个Redis Instance负责其中的一部分的Slot的读写。集群客户端连接集群中任一Redis Instance即可发送命令，当Redis Instance收到自己不负责的Slot的请求时，会将负责请求Key所在Slot的Redis Instance地址返回给客户端，客户端收到后自动将原请求重新发往这个地址，对外部透明。一个Key到底属于哪个Slot由crc16(key) % 16384 决定。

关于负载均衡，集群的Redis Instance之间可以迁移数据，以Slot为单位，但不是自动的，需要外部命令触发。

关于集群成员管理，集群的节点(Redis Instance)和节点之间两两定期交换集群内节点信息并且更新，从发送节点的角度看，这些信息包括：集群内有哪些节点，IP和PORT是什么，节点名字是什么，节点的状态(比如OK，PFAIL，FAIL，后面详述)是什么，包括节点角色(master或者 slave)等。

关于可用性，集群由N组主从Redis Instance组成。主可以没有从，但是没有从 意味着主宕机后主负责的Slot读写服务不可用。一个主可以有多个从，主宕机时，某个从会被提升为主，具体哪个从被提升为主，协议类似于Raft，参见这里。如何检测主宕机？Redis Cluster采用quorum+心跳的机制。从节点的角度看，节点会定期给其他所有的节点发送Ping，cluster-node-timeout(可配置，秒级)时间内没有收到对方的回复，则单方面认为对端节点宕机，将该节点标为PFAIL状态。通过节点之间交换信息收集到quorum个节点都认为这个节点为PFAIL，

则将该节点标记为FAIL，并且将其发送给其他所有节点，其他所有节点收到后立即认为该节点宕机。从这里可以看出，主宕机后，至少cluster-node-timeout时间内该主所负责的Slot的读写服务不可用。

redis cluster turorial

Redis Cluster supports multiple key operations as long as all the keys involved into a single command execution (or whole transaction, or Lua script execution) **all belong to the same hash slot**. The user can force multiple keys to be part of the same hash slot by using a concept called **hash tags**.

Redis Cluster does not use consistent hashing, but a different form of sharding where every key is conceptually part of what we call an hash slot.可以对比一下一致性哈希，确实不完全一样 There are 16384 hash slots in Redis Cluster, and to compute what is the hash slot of a given key, we simply take the CRC16 of the key modulo 16384.

Every node in a Redis Cluster is responsible for a subset of the hash slots, so for example you may have a cluster with 3 nodes, where:

Node A contains hash slots from 0 to 5500. Node B contains hash slots from 5501 to 11000. Node C contains hash slots from 11001 to 16383.

redis cluster spec

Implemented subset

Redis Cluster implements all the single key commands available in the non-distributed version of Redis. Commands performing complex multi-key operations like Set type unions or intersections are implemented as well as long as **the keys all belong to the same node**.

Redis Cluster implements a concept called **hash tags** that can be used in order to force certain keys to be stored in the same node. However during manual **reshardings**, multi-key operations may become unavailable for some time while single key operations are always available.

Redis Cluster **does not support** multiple databases like the stand alone version of Redis. There is just database 0 and the SELECT command is not allowed.

write safety 数据丢失的可能性

1. A write may reach a master, but while the master may be able to reply to the client, the write may not be propagated to slaves via the **asynchronous replication** used between master and slave nodes. If the master dies without the write reaching the slaves, the write is lost forever if the master is unreachable for a long enough period

that one of its slaves is promoted. This is usually hard to observe in the case of a total, sudden failure of a master node since masters try to reply to clients (with the acknowledge of the write) and slaves (propagating the write) at about the same time. However it is a real world failure mode.异步方式同步还是不可靠，应该参考kafka

2. Another theoretically possible failure mode where writes are lost is the following:
    - A master is unreachable because of a partition.
    - It gets failed over by one of its slaves.
    - After some time it may be reachable again.
    - A client with an out-of-date routing table may write to the old master before it is converted into a slave (of the new master) by the cluster.客户端不知道这时这台机器已经是slave还以为他是master去写，而且写成功了。

kudu

项目资源

apache kudu overview

apach kudu document

github kudu code

技术要点

- OLAP
- 列存储

Twitter Firehose Table

tweet_id	user_name	created_at	text
INT64	STRING	TIMESTAMP	STRING
23059873	newsycbot	1442865158	Visual Explanation of the Raft Consensus Algorithm http://bit.ly/1DOUac0 (cmts http://bit.ly/1HKmjfc)
22309487	RideImpala	1442828307	Introducing the Ibis project: for the Python experience at Hadoop Scale
23059861	fastly	1442865156	Missed July's SF @papers_we_love? You can now watch @el_bhs talk about @google's globally-distributed database: http://fastly.us/1eVz8MM
23010982	llvmorg	1442865155	LLVM 3.7 is out! Get it while it's HOT! http://llvm.org/releases/download.html#3.7.0

- libpmem
- Persistent Memory wiki
- kudu paper
- Raft

参考

https://www.wikiwand.com/en/Persistent_memory

cassandra

概念

基础技术

gossip

每个节点根据一个特定的集群状态计算出的某个节点的sharding信息应该是完全一样的。

首先集群中的每个节点获得的集群状态应该是一致的，其次根据这个一致的状态应该用同样的逻辑来进行计算

Gossip协议是一种无中心节点的分布式系统中常用的通信协议，akka等系统也用。简单来说就是每个节点周期性地随机找一个节点互相同步彼此的信息，P2P通信，很像人类社会的“gossip”因而得名。Cassandra为了防止集群被拆成两半，主要通过两个手段。第一个手段是gossip协议需要在起始阶段人工指定“种子节点”来获取初步的信息，并且一般来说需要每个节点填相同的若干个种子节点以保证在初期肯定能连到同一个集群中，此外每个gossip周期随机选其他节点同步消息时，如果随机到的不是种子节点则额外选随机一个种子节点再进行同步。这样既可以保证每个节点都能连到同一个集群中，也可以保证每个已存在的节点可以第一时间知道有新的节点加入（因为种子节点是最先知道新节点加入的）。

总的来说对gossip消息的处理就是在消息中封装自己知道的集群中每个节点的当前状态的version给对方，对方与本地信息判断，如果比本地新就更新本地，如果比本地旧就把更新的信息给对方。

Gossip只维护集群节点状态信息，不负责维护其他信息。

一致性哈希

参考资料

一致性哈希算法

维基百科

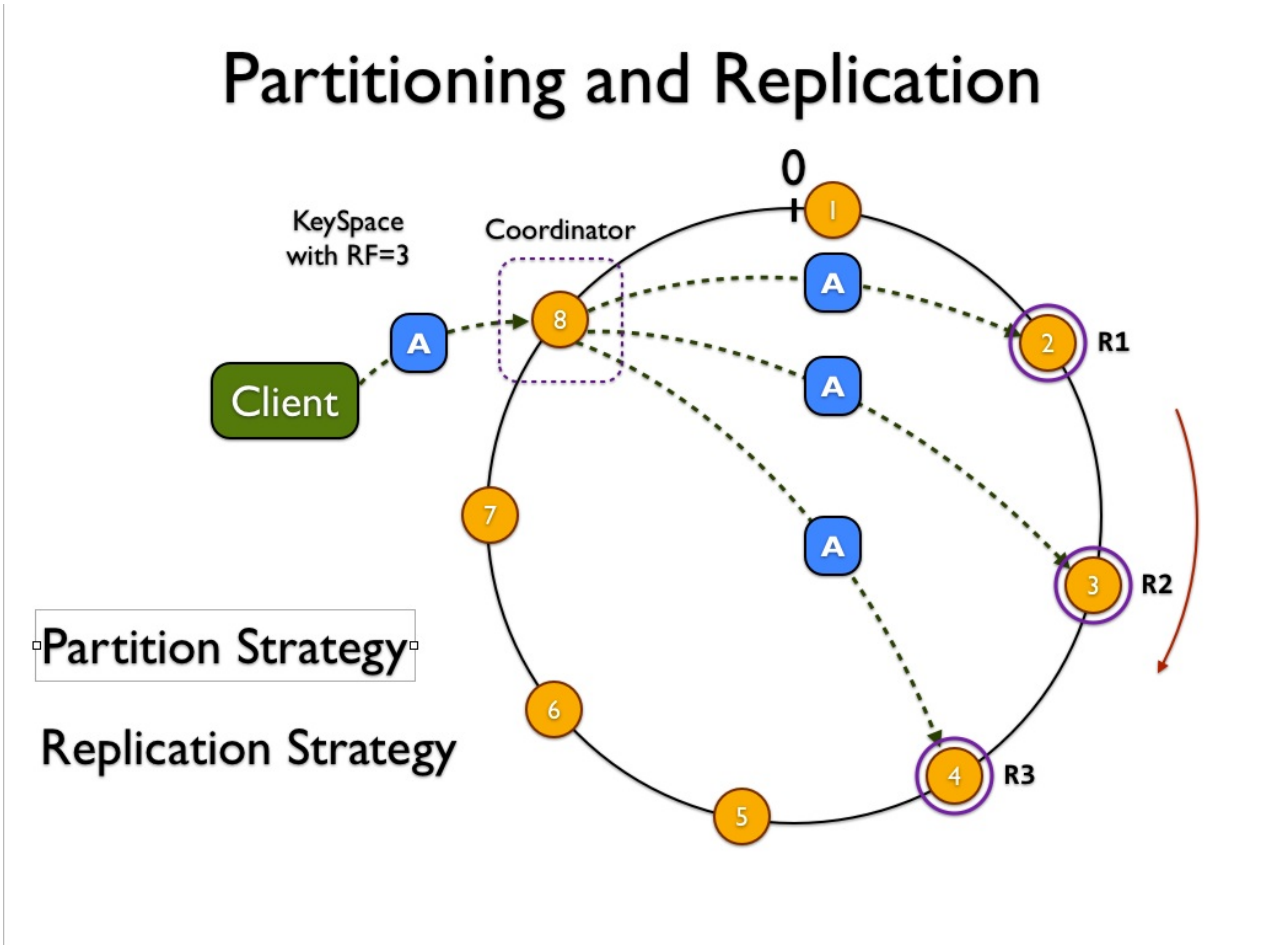
一致性哈希（consistent hashing）是分布式哈希表（distributed hash table）的一种

Kademlia算法的细节可以看维基百科等资料，Kademlia算法解决的是节点数非常多、不能把网络中的每个节点都存起来的情况。

一致性哈希是DHT的另一种算法，把哈希值的取值空间首尾连接变成一个环，每个哈希值在环上都能找到对应的位置，数据的哈希值对应位置顺时针找下一个节点对应的哈希值的位置，就是该存放这条数据的节点（如果数据存N份就是接下来的N个节点）。这种算法的优势是如果环上增减了节点，只需要迁移环上位置上下两侧对应的数据，其他的数据不用动？这会导致数据不均匀。

Cassandra从1.2开始支持虚拟节点vnode——每个Cassandra实例在环上注册多个节点（默认256个）。vnode意味着增加或者减少一个实例的时候，导数据会从几乎所有节点导，而不是只从相邻节点导，在数据总量不变的情况下，从更多的节点导可以减少导数据的时间，也可以均衡负载。

副本同步 **Replication**



问题

N=3,R=W=2 因为为了确保可用性，通常读写时W和R都会小于N，因此通常N是3然后WR都设为quorum也就是2。不考虑时间戳误差的问题的话基本上是可以确保总是能读到最新的数据的。但是可能很多数据实际上只写了两份，第三份失败了。虽然平时读起来没啥问题，对client来说是感知不到的，无论选哪两个节点读，都还是能读到数据。如果这个时候想新增一

个节点，而这个节点在导数据的时候选择从没有数据的那个节点导（为了加快速度减少初期的磁盘冗余Cassandra只会从N个节点中的一个来导某个区间的数据），那么导完数据在status从joining变成up后，可能会出三种问题

第一个问题是，假如新上来的节点在环上取代的那个节点恰好是本来有数据的节点，那么接下来这个数据对应的三个节点里只有一个节点有这个数据，这个时候依然quorum的读就有1/3的概率读不到数据了不符合预期，而且数据冗余只有1份也很容易彻底丢失。

第二个问题是，导数据的时候新写的数据还是写在原来的那三个节点，假设依然是有一个节点没写成，那么导完数据节点变成up后，还是有可能只有一个节点有数据。

第三个问题是，因为gossip的集群信息维护是最终一致性的，一个节点从joining变成up并不是马上就让所有人知道的，一旦有些client知道有些client不知道，那么读写的节点就只有两个是一样的第三个是错开的，假如写的client又是只写成两份，与他错开的client可能就只能读到一份了。

这三个问题总体上都是因为quorum的写并不能保证所有节点都写成功，而降低冗余度无论对数据安全性还是一致性都没好处。因此Cassandra提供两种方式来修复冗余度。第一种是在读取的时候进行read repair操作，强制从所有节点读数据，一旦发现数据不一致就将N个数据合并作为真正的最新数据，把最新数据与每个节点当前数据diff的部分重新写回去。第二种是后台的repair操作，一次repair一个表在当前节点的所有数据。

对第二个问题和第三个问题为了避免不得不开read repair，Cassandra有个特殊处理，如果发现当前写入的数据想写W份而目前正有一个节点在导数据而未来这个数据也会放在这个新节点上，那么就把这个新节点也当做写入的目标，并且必须额外再多写成功一份才算全局成功，也就是从四个节点里写成三个才可以。这样就不用开read repair了。

compaction

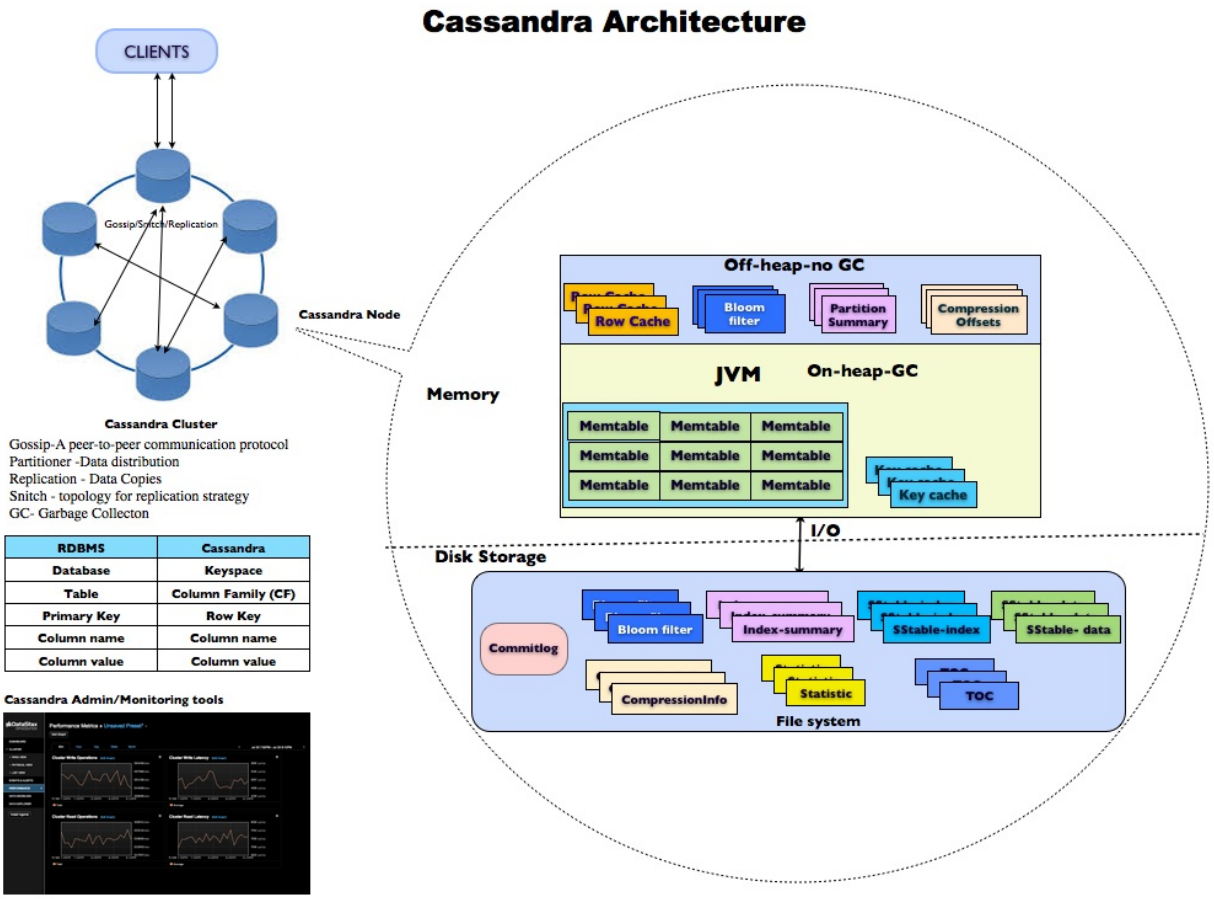
Cassandra现在提供三种compaction的策略。Size-tiered、Leveled、Date-tiered。Size就是把大小相近的SSTable合成大的，是最常用的；Leveled就是跟leveldb一样，适合读>>写而且插入多于修改和删除的；Date-tiered是最近刚加的，适合类似发微博之类的对每个partition key插入的column key永远单增（比如当前时间戳）而且不修改旧数据的。第三个没用过，只有一个表用过leveled，是词典的发音数据。词典发音数据是一堆真人发音+例句的TTS发音文件，当数据库没有数据的时候，去请求TTS服务器来合成语音并存入Cassandra，所以这个表的写入量非常小，几乎没有，因此很适合leveled compaction。除了这个表还用了另一个表存“海量发音”，就是词典去年新出的能看一个词全世界各国人民的当地发音的数据，这个表只用了Size-tiered，因为要不断从合作方抓数据，更新的频率比前一个表大不少。对比来看的话，使用leveled compaction的表的平均读取延迟少了一半。

删除产生tombstone

LSM类的数据库都是把删除替换为写入一个标记，叫tombstone。HBase也差不多的原理，删除数据的话把tombstone写到log里，compaction的时候如果遇到tombstone就可以不用保留已经被删的数据，读数据的时候遇到tombstone就屏蔽掉数据并且跳过，最后返回所有没删的数据。但是，C*因为并不是靠单个region server来读写数据，而是N个节点同时读写，所以跟HBase最大的区别，就是每个节点维护tombstone的时候不能只考虑自己。于是某单个节点读数据的时候读到tombstone后，不能屏蔽掉data然后跳过，而是要把tombstone也返回给接受client请求的那个coordinator，因为你不知道其他节点是否写入了这个tombstone（因为写入W个就算成功）。而且在compaction的时候不能判断说当前节点跟这个tombstone有关的数据都删完了就把自己也删了，因为你不确定别的节点有没有这个tombstone——如果别的节点在写tombstone的时候没写成（毕竟写W个节点就算成功，最多N-W个节点没有这个tombstone），自己又在compaction的时候把tombstone删了，那么再读数据的时候有节点有这行数据，又没有节点有这行数据的tombstone，于是这个数据就复活了。

Cassandra通过写一条“tombstone”来标记一个数据被删除了。被标记的数据默认要10天(配置文件中的gc_grace_seconds)后且被compaction或cleanup执行到对应的SSTable时才会被真正从磁盘删除，因为如果当时这个delete操作只在3个节点中的2个执行成功，那么一旦2个有tombstone的节点把数据删了，集群上只剩下没tombstone的那个节点，下次读这个key的时候就又返回对应的数据，从而导致被删除的数据复活。Repair操作可以同步所有节点的数据从而保证tombstone在3个节点中都存在，因此如果想确保删除100%成功不会复活需要以小于gc_grace_seconds的周期定期执行repair操作(所以官方建议”weekly”)。

架构



参考

Gossip、DHT和传说中的W+R>N

Cassandra删除数据的坑

greenplum

概念

基础技术

架构

代码

https://github.com/greenplum-db/gpdb