**Redis Cluster 实现**

本文将从设计思路，功能实现几个方面介绍Redis Cluster。（下面提到的**epoch（类似于Raft算法“term”（任期）**）

假设读者已经了解Redis Cluster的使用方式,不了解可以先看之前一篇介绍Redis Cluster的博客。

**简介**

Redis Cluster作为Redis的分布式实现，主要做了两个方面的事情：

**1，数据分片**

* Redis Cluster将数据按key哈希到16384个slot上
* Cluster中的不同节点负责一部分slot

**2，故障恢复**

* Cluster中直接提供服务的节点为Master
* 每个Master可以有一个或多个Slave
* 当Master不能提供服务时，Slave会自动Failover

**设计思路**

**性能为第一目标**

* 每一次数据处理都是由负责当前slot的Master直接处理的，没有额外的网络开销

**提高可用性**

* **水平扩展**能力 ：由于slot的存在，增加机器节点时只需要**将之前由其他节点处理的一部分slot重新分配给新增节点**。slot可以看做机器节点和用户数据之间的一个抽象层。
* **故障恢复**：Slave会在需要的时候自动提升为Master

**损失一致性**

* Master与Slave之间**异步复制**，**即Master先向用户返回结果后再异步将数据同步给Slave，这就导致Master宕机后一部分已经返回用户的数据在新Master上不存在**
* **网络分区**时，由于开始Failover前的超时时间，会有一部分数据继续写到马上要失效的Master上

**功能实现**

**1，数据分片**

我们已经知道数据会按照key哈希到不同的slot，而每个节点仅负责一部分的slot，客户端根据slot将请求交给不同的节点。将slots划分给不同节点的过程称为数据分片，对应的还可以进行分片的重新分配。这部分功能依赖外部调用命令：

**分片**

* 对每个集群执行CLUSTER ADDSLOTS slot [slot ...]
* RedisCluster将命令指定的slots作为自己负责的部分

**再分配**

**再分配要做的是将一些slots从当前节点(source)迁移到其他节点**(**target**)

* 对**target**执行CLUSTER SETSLOT slot IMPORTING [node-id]，target节点将对应slots记为importing状态；
* 对**source**执行CLUSTER SETSLOT MIGRATING[node-id]，source节点将对应slots记为migrating状态，与importing状态一同在之后的请求重定向中使用
* 获取所有要迁移slot对应的keys，CLUSTER GETKEYSINSLOT slot count
* 对**source** 执行MIGRATE host port key db timeout REPLACE [KEYS key [key ...]]
* MIGRATE命令会将所有的指定的key通过RESTORE key ttl serialized-value REPLACE迁移给**target**
* 对所有节点执行CLUSTER SETSLOT slot NODE [node-id]，申明**target**对这些slots的负责，并退出importing或migrating

**2，请求重定向**

由于每个节点只负责部分slot，以及slot可能从一个节点迁移到另一节点，造成客户端有可能会向错误的节点发起请求。因此需要有一种机制来对其进行发现和修正，这就是请求重定向。有两种不同的重定向场景：

**1)，MOVE**

* ‘我’并不负责‘你’要的key，告诉’你‘正确的吧。
* 返回CLUSTER\_REDIR\_MOVED错误，和正确的节点。
* 客户端向该节点重新发起请求，注意这次依然又发生重定向的可能。

**2），ASK**

* ‘我’负责请求的key，但不巧的这个key当前在migraging状态，且‘我’这里已经取不到了。告诉‘你’importing他的‘家伙’吧，去碰碰运气。
* 返回CLUSTER\_REDIR\_ASK，和importing该key的节点。
* 客户端向新节点发送ASKING，之后再次发起请求
* 新节点对发送过ASKING，且key已经migrate过来的请求进行响应

**3），区别**

区分这两种重定向的场景是非常有必要的：

* MOVE，申明的是slot所有权的转移，收到的客户端需要更新其key-node映射关系
* ASK，申明的是一种临时的状态，所有权还并没有转移，客户端并不更新其映射关系。前面的加的ASKING命令也是申明其理解当前的这种临时状态

**3，状态检测及维护**

Cluster中的每个节点都维护一份在自己看来当前整个集群的状态，主要包括：

* 当前集群状态
* 集群中各节点所负责的slots信息，及其migrate状态
* 集群中各节点的master-slave状态
* 集群中各节点的存活状态及不可达投票

当集群状态变化时，如**新节点加入、slot迁移、节点宕机、slave提升为新Master，**我们希望这些变化尽快的被发现，传播到整个集群的所有节点并达成一致。

节点之间相互的心跳（PING，PONG，MEET）及其携带的数据是集群状态传播最主要的途径。

**心跳时机：**

Redis节点会记录其向每一个节点上一次发出ping和收到pong的时间，心跳发送时机与这两个值有关。通过下面的方式既能保证及时更新集群状态，又不至于使心跳数过多：

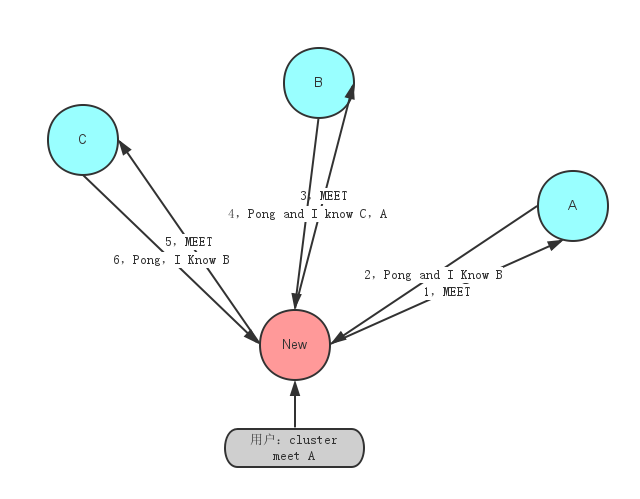
* 每次Cron向所有未建立链接的节点发送ping或meet
* 每1秒从所有已知节点中随机选取5个，向其中上次收到pong最久远的一个发送ping
* 每次Cron向收到pong超过timeout/2的节点发送ping
* 收到ping或meet，立即回复pong

**心跳数据**

* Header，发送者自己的信息
  + 所负责slots的信息
  + 主从信息
  + ip port信息
  + 状态信息
* Gossip，发送者所了解的部分其他节点的信息
  + ping\_sent, pong\_received
  + ip, port信息
  + 状态信息，比如发送者认为该节点已经不可达，会在状态信息中标记其为PFAIL或FAIL

**心跳处理**

* 1，新节点加入
  + 发送meet包加入集群
  + 从pong包中的gossip得到未知的其他节点
  + 循环上述过程，直到最终加入集群



* 2，Slots信息
  + 判断发送者声明的slots信息，跟本地记录的是否有不同
  + **如果不同，且发送者epoch较大，更新本地记录**
  + **如果不同，且发送者epoch小，发送Update信息通知发送者**
* 3，Master slave信息
  + **发现发送者的master、slave信息变化，更新本地状态**
* 4，节点Fail探测
  + **超过超时时间仍然没有收到pong包的节点会被当前节点标记为PFAIL**
  + **PFAIL标记会随着gossip传播**
  + **每次收到心跳包会检测其中对其他节点的PFAIL标记**
  + **对某个节点的PFAIL标记达到大多数时，将其变为FAIL标记并广播FAIL消息**

*注：Gossip的存在使得集群状态的改变可以更快的达到整个集群。每个心跳包中会包含多个Gossip包，那么多少个才是合适的呢，redis的选择是N/10，其中N是节点数，这样可以保证在PFAIL投票的过期时间内，****节点可以收到80%机器关于失败节点的gossip，从而使其顺利进入FAIL状态****。*

**广播**

当需要发布一些非常重要需要立即送达的信息时，上述心跳加Gossip的方式就显得捉襟见肘了，这时就需要向所有集群内机器的广播信息，使用广播发的场景：

* 节点的Fail信息：当发现某一节点不可达时，探测节点会将其标记为PFAIL状态，并通过心跳传播出去。当某一节点发现这个节点的PFAIL超过半数时修改其为FAIL并发起广播。
* Failover Request信息：slave尝试发起FailOver时广播其要求投票的信息
* 新Master信息：Failover成功的节点向整个集群广播自己的信息

**4，故障恢复（Failover）**

当slave发现自己的master变为FAIL状态时，便尝试进行Failover，以期成为新的master。由于挂掉的master可能会有多个slave。Failover的过程需要经过类Raft协议的过程在整个集群内达到一致， 其过程如下：

* **slave发现自己的master变为FAIL**
* **将自己记录的集群currentEpoch加1，并广播Failover Request信息**
* **其他节点收到该信息，只有master响应，判断请求者的合法性，并发送FAILOVER\_AUTH\_ACK，对每一个epoch只发送一次ack**
* **尝试failover的slave收集FAILOVER\_AUTH\_ACK**
* **超过半数后变成新Master**
* **广播Pong通知其他集群节点**

