集群

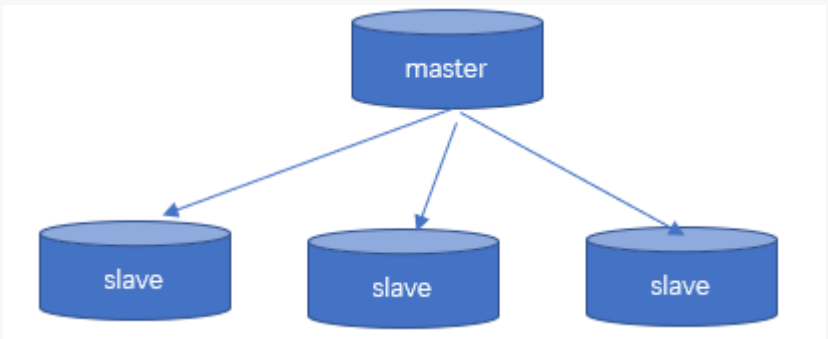
先来简单了解下redis中提供的集群策略, 虽然redis有持久化功能能够保障redis服务器宕机也能恢复并且只有少量的数据损失，但是由于所有数据在一台服务器上，如果这台服务器出现硬盘故障，那就算是有备份也仍然不可避免数据丢失的问题。

在实际生产环境中，我们不可能只使用一台redis服务器作为我们的缓存服务器，必须要多台实现集群，避免出现单点故障；

主从复制

复制的作用是把redis的数据库复制多个副本部署在不同的服务器上，如果其中一台服务器出现故障，也能快速迁移到其他服务器上提供服务。 复制功能可以实现当一台redis服务器的数据更新后，自动将新的数据同步到其他服务器上

主从复制就是我们常见的master/slave模式， 主数据库可以进行读写操作，当写操作导致数据发生变化时会自动将数据同步给从数据库。而一般情况下，从数据库是只读的，并接收主数据库同步过来的数据。 一个主数据库可以有多个从数据库



配置

在redis中配置master/slave是非常容易的，只需要在从数据库的配置文件中加入slaveof 主数据库地址 端口。 而master 数据库不需要做任何改变

```
准备两台服务器，分别安装redis ， server1  server2

\1.    在server2的redis.conf文件中增加 slaveof server1-ip 6379 、 同时将bindip注释掉，允许所有ip访问

\2.  启动server2

\3.  访问server2的redis客户端，输入 INFO replication
\4.  通过在master机器上输入命令，比如set foo bar 、 在slave服务器就能看到该值已经同步过来了
```

原理

全量复制

Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。具体步骤

完成上面几个步骤后就完成了slave服务器数据初始化的所有操作，savle服务器此时可以接收来自用户的读请求。

master/slave 复制策略是采用乐观复制，也就是说可以容忍在一定时间内master/slave数据的内容是不同的，但是两者的数据会最终同步。具体来说，redis的主从同步过程本身是异步的，意味着master执行完客户端请求的命令后会立即返回结果给客户端，然后异步的方式把命令同步给slave。

这一特征保证启用master/slave后 master的性能不会受到影响。

但是另一方面，如果在这个数据不一致的窗口期间，master/slave因为网络问题断开连接，而这个时候，master是无法得知某个命令最终同步给了多少个slave数据库。不过redis提供了一个配置项来限制只有数据至少同步给多少个slave的时候，master才是可写的：

`min-slaves-to-write 3` 表示只有当3个或以上的slave连接到master，master才是可写的

`min-slaves-max-lag 10` 表示允许slave最长失去连接的时间，如果10秒还没收到slave的响应，则master认为该slave以断开

增量复制

从redis 2.8开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份

master node会在内存中创建一个backlog，master和slave都会保存一个replica offset还有一个master id，offset就是保存在backlog中的。如果master和slave网络连接断掉了，slave会让master从上次的replica offset开始继续复制

但是如果没有找到对应的offset，那么就会执行一次全量同步

无硬盘复制

前面我们说过，Redis复制的工作原理基于RDB方式的持久化实现的，也就是master在后台保存RDB快照，slave接收到rdb文件并载入，但是这种方式会存在一些问题

\1. 当master禁用RDB时，如果执行了复制初始化操作，Redis依然会生成RDB快照，当master下次启动时执行该RDB文件的恢复，但是因为复制发生的时间点不确定，所以恢复的数据可能是任何时间点的。就会造成数据出现问题

\2. 当硬盘性能比较慢的情况下（网络硬盘），那初始化复制过程会对性能产生影响

因此2.8.18以后的版本，Redis引入了无硬盘复制选项，可以不需要通过RDB文件去同步，直接发送数据，通过以下配置来开启该功能

repl-diskless-sync yes

*master**在内存中直接创建rdb，然后发送给slave，不会在自己本地落地磁盘了*

哨兵机制

在前面讲的master/slave模式，在一个典型的一主多从的系统中，slave在整个体系中起到了数据冗余备份和读写分离的作用。当master遇到异常终端后，需要从slave中选举一个新的master继续对外提供服务，这种机制在前面提到过N次，比如在zk中通过leader选举、kafka中可以基于zk的节点实现master选举。所以在redis中也需要一种机制去实现master的决策，redis并没有提供自动master选举功能，而是需要借助一个哨兵来进行监控

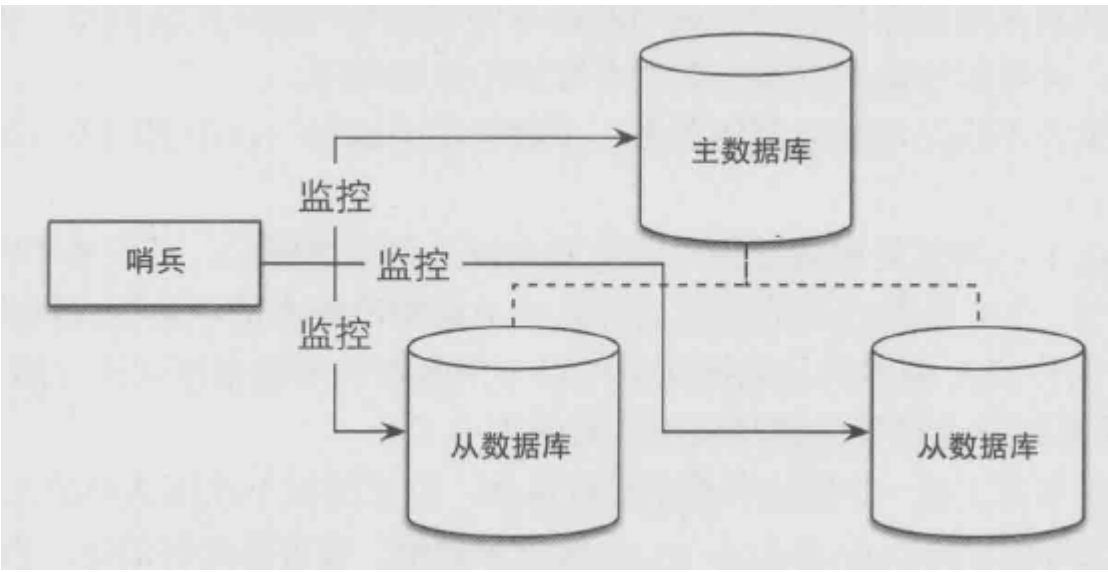
什么是哨兵

顾名思义，哨兵的作用就是监控Redis系统的运行状况，它的功能包括两个

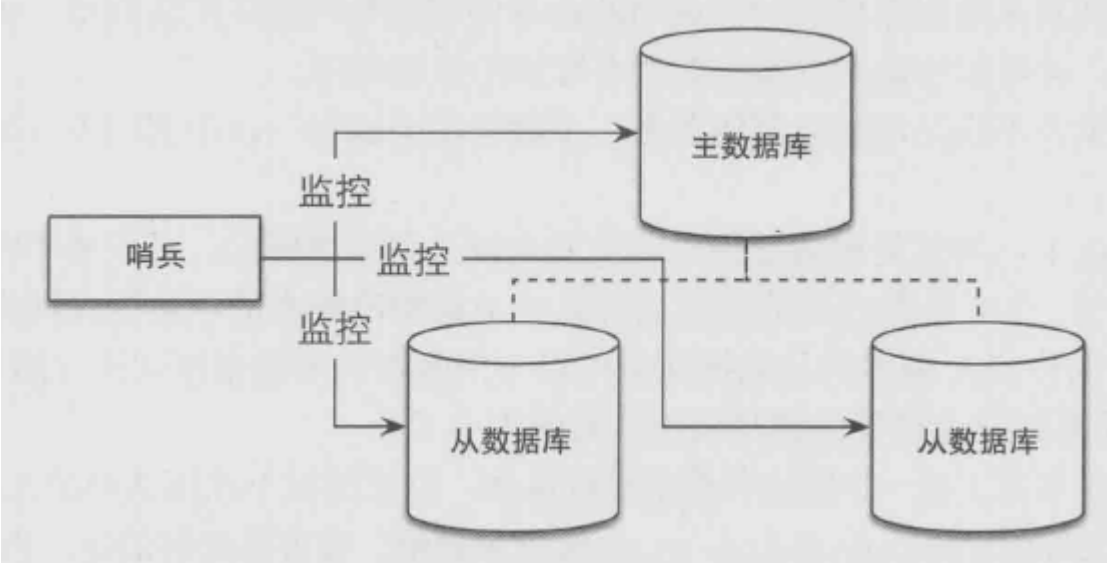
\1. 监控master和slave是否正常运行

\2. master出现故障时自动将slave数据库升级为master

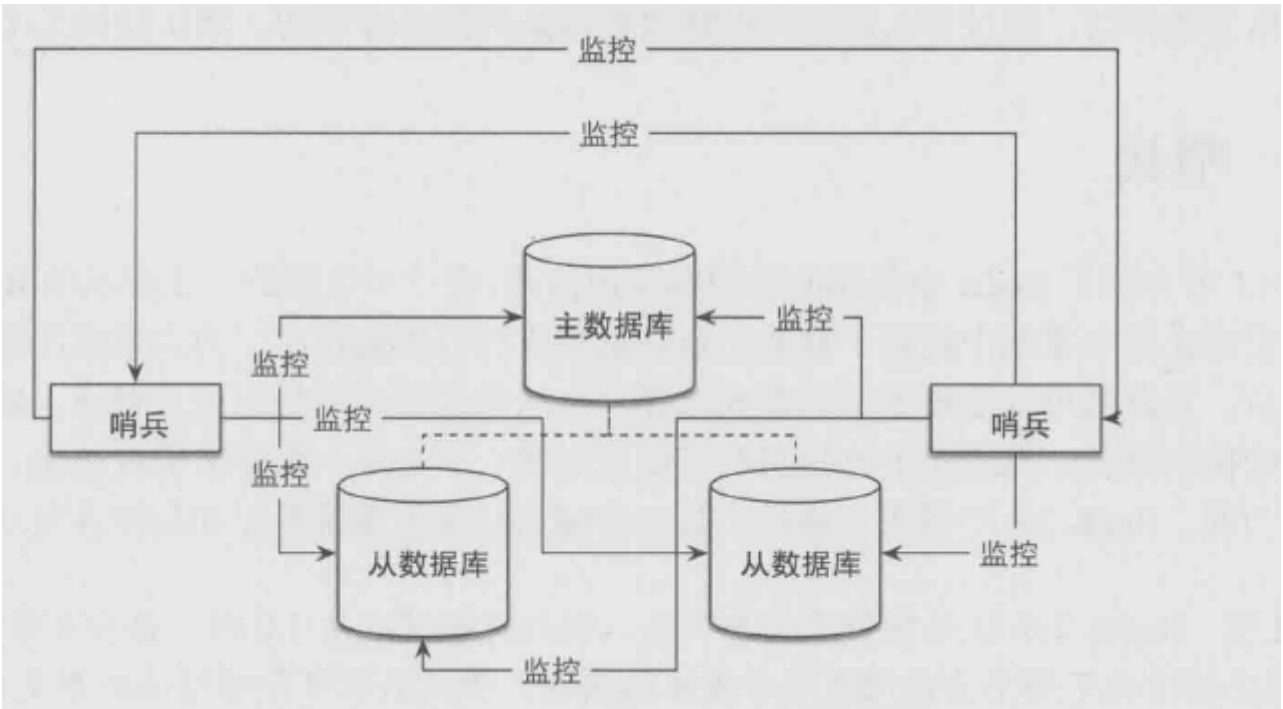
哨兵是一个独立的进程，使用哨兵后的架构图



为了解决master选举问题，又引出了一个单点问题，也就是哨兵的可用性如何解决，在一个一主多从的Redis系统中，可以使用多个哨兵进行监控任务以保证系统足够稳定。此时哨兵不仅会监控master和slave，同时还会互相监控；这种方式称为哨兵集群，哨兵集群需要解决故障发现、和master决策的协商机制问题



为了解决master选举问题，又引出了一个单点问题，也就是哨兵的可用性如何解决，在一个一主多从的Redis系统中，可以使用多个哨兵进行监控任务以保证系统足够稳定。此时哨兵不仅会监控master和slave，同时还会互相监控；这种方式称为哨兵集群，哨兵集群需要解决故障发现、和master决策的协商机制问题



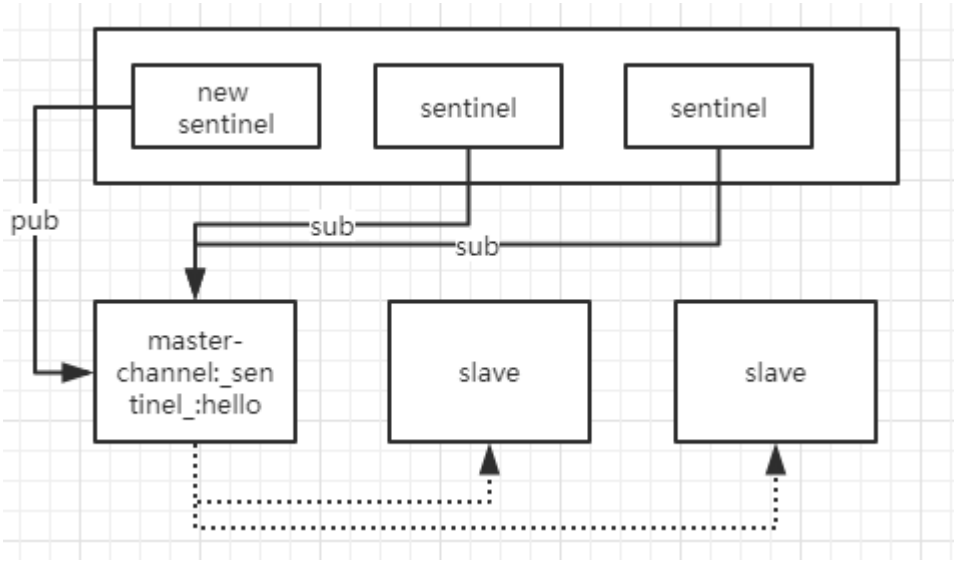
sentinel之间的相互感知

sentinel节点之间会因为共同监视同一个master从而产生了关联，一个新加入的sentinel节点需要和其他监视相同master节点的sentinel相互感知，首先

\1. 需要相互感知的sentinel都向他们共同监视的master节点订阅channel:*sentinel*:hello

\2. 新加入的sentinel节点向这个channel发布一条消息，包含自己本身的信息，这样订阅了这个channel的sentinel就可以发现这个新的sentinel

\3. 新加入得sentinel和其他sentinel节点建立长连接



master的故障发现

sentinel节点会定期向master节点发送心跳包来判断存活状态，一旦master节点没有正确响应，sentinel会把master设置为“主观不可用状态”，然后它会把“主观不可用”发送给其他所有的sentinel节点去确认，当确认的sentinel节点数大于>quorum时，则会认为master是“客观不可用”，接着就开始进入选举新的master流程；但是这里又会遇到一个问题，就是sentinel中，本身是一个集群，如果多个节点同时发现master节点达到客观不可用状态，那谁来决策选择哪个节点作为maste呢？这个时候就需要从sentinel集群中选择一个leader来做决策。而这里用到了一致性算法Raft算法、它和Paxos算法类似，都是分布式一致性算法。但是它比Paxos算法要更容易理解；Raft和Paxos算法一样，也是基于投票算法，只要保证过半数节点通过提议即可;

动画演示地址：http://thesecretlivesofdata.com/raft/

配置实现

通过在这个配置的基础上增加哨兵机制。在其中任意一台服务器上创建一个sentinel.conf文件，文件内容

sentinel monitor name ip port quorum

`其中name表示要监控的master的名字，这个名字是自己定义。 ip和port表示master的ip和端口号。 最后一个1表示最低通过票数，也就是说至少需要几个哨兵节点统一才可以，后面会具体讲解`

port 6040

sentinel monitor mymaster 192.168.11.131 6379 1

sentinel down-after-milliseconds mymaster 5000 --表示如果5s内mymaster没响应，就认为SDOWN

sentinel failover-timeout mymaster 15000 --表示如果15秒后,mysater仍没活过来，则启动failover，从剩下的slave中选一个升级为master

两种方式启动哨兵

redis-sentinel sentinel.conf

redis-server /path/to/sentinel.conf --sentinel

哨兵监控一个系统时，只需要配置监控master即可，哨兵会自动发现所有slave；

这时候，我们把master关闭，等待指定时间后（默认是30秒），会自动进行切换，会输出如下消息

img

+sdown表示哨兵主管认为master已经停止服务了，+odown表示哨兵客观认为master停止服务了。关于主观和客观，后面会给大家讲解。接着哨兵开始进行故障恢复，挑选一个slave升级为master

+try-failover表示哨兵开始进行故障恢复

+failover-end 表示哨兵完成故障恢复

+slave表示列出新的master和slave服务器，我们仍然可以看到已经停掉的master，哨兵并没有清楚已停止的服务的实例，这是因为已经停止的服务器有可能会在某个时间进行恢复，恢复以后会以slave角色加入到整个集群中

Redis-Cluster

即使是使用哨兵，此时的Redis集群的每个数据库依然存有集群中的所有数据，从而导致集群的总数据存储量受限于可用存储内存最小的节点，形成了木桶效应。而因为Redis是基于内存存储的，所以这一个问题在redis中就显得尤为突出了

在redis3.0之前，我们是通过在客户端去做的分片，通过hash环的方式对key进行分片存储。分片虽然能够解决各个节点的存储压力，但是导致维护成本高、增加、移除节点比较繁琐。因此在redis3.0以后的版本最大的一个好处就是支持集群功能，集群的特点在于拥有和单机实例一样的性能，同时在网络分区以后能够提供一定的可访问性以及对主数据库故障恢复的支持。

哨兵和集群是两个独立的功能，当不需要对数据进行分片使用哨兵就够了，如果要进行水平扩容，集群是一个比较好的方式

拓扑结构

一个Redis Cluster由多个Redis节点构成。不同节点组服务的数据没有交集，也就是每个一节点组对应数据sharding的一个分片。节点组内部分为主备两类节点，对应master和slave节点。两者数据准实时一致，通过异步化的主备复制机制来保证。一个节点组有且只有一个master节点，同时可以有0到多个slave节点，在这个节点组中只有master节点对用户提供些服务，读服务可以由master或者slave提供

3.png

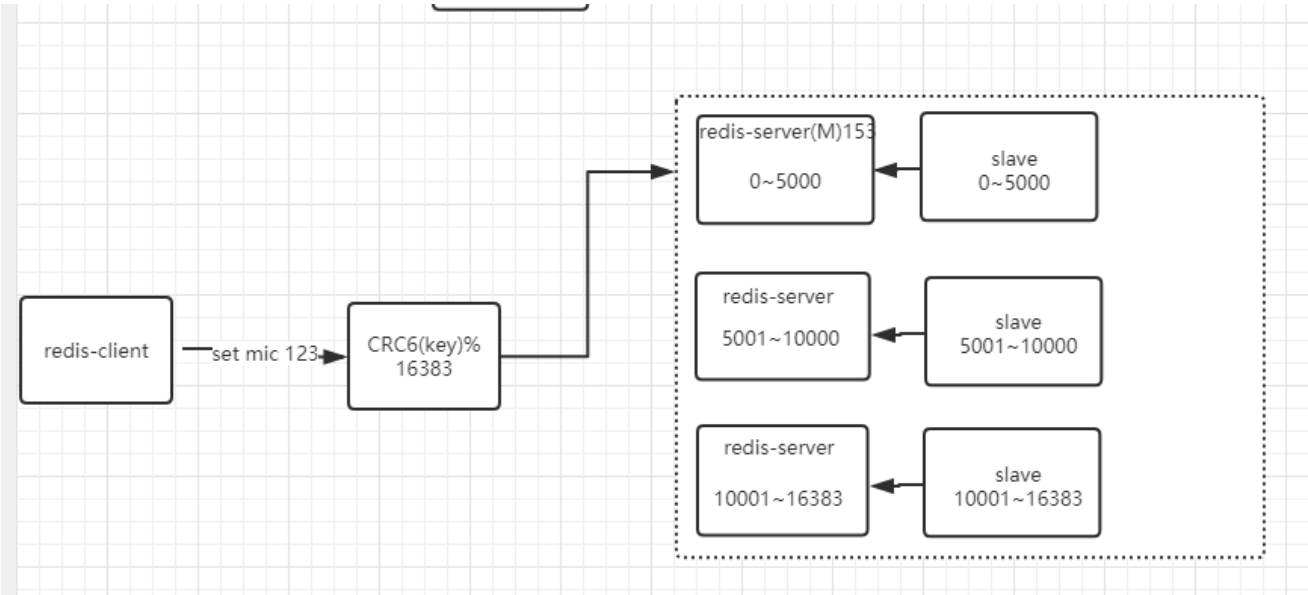
redis-cluster是基于gossip协议实现的无中心化节点的集群，因为去中心化的架构不存在统一的配置中心，各个节点对整个集群状态的认知来自于节点之间的信息交互。在Redis Cluster，这个信息交互是通过Redis Cluster Bus来完成的

Redis的数据分区

分布式数据库首要解决把整个数据集按照分区规则映射到多个节点的问题，即把数据集划分到多个节点上，每个节点负责整个数据的一个子集, Redis Cluster采用哈希分区规则,采用虚拟槽分区。

虚拟槽分区巧妙地使用了哈希空间，使用分散度良好的哈希函数把所有的数据映射到一个固定范围内的整数集合，整数定义为槽（slot）。比如Redis Cluster槽的范围是0 ~ 16383。槽是集群内数据管理和迁移的基本单位。采用大范围的槽的主要目的是为了方便数据的拆分和集群的扩展，每个节点负责一定数量的槽。

计算公式：slot = CRC16(key)%16383。每一个节点负责维护一部分槽以及槽所映射的键值数据。



HashTags

通过分片手段，可以将数据合理的划分到不同的节点上，这本来是一件好事。但是有的时候，我们希望对相关联的业务以原子方式进行操作。举个简单的例子

我们在单节点上执行MSET , 它是一个原子性的操作，所有给定的key会在同一时间内被设置，不可能出现某些指定的key被更新另一些指定的key没有改变的情况。但是在集群环境下，我们仍然可以执行MSET命令，但它的操作不在是原子操作，会存在某些指定的key被更新，而另外一些指定的key没有改变，原因是多个key可能会被分配到不同的机器上。

所以，这里就会存在一个矛盾点，及要求key尽可能的分散在不同机器，又要求某些相关联的key分配到相同机器。这个也是在面试的时候会容易被问到的内容。怎么解决呢?

从前面的分析中我们了解到，分片其实就是一个hash的过程，对key做hash取模然后划分到不同的机器上。所以为了解决这个问题，我们需要考虑如何让相关联的key得到的hash值都相同呢？如果key全部相同是不现实的，所以怎么解决呢？在redis中引入了HashTag的概念，可以使得数据分布算法可以根据key的某一个部分进行计算，然后让相关的key落到同一个数据分片

举个简单的例子，加入对于用户的信息进行存储， user:user1:id、user:user1:name/ 那么通过hashtag的方式，user:{user1}:id、user:{user1}.name; 表示

当一个key包含 {} 的时候，就不对整个key做hash，而仅对 {} 包括的字符串做hash。

重定向客户端

Redis Cluster并不会代理查询，那么如果客户端访问了一个key并不存在的节点，这个节点是怎么处理的呢？比如我想获取key为msg的值，msg计算出来的槽编号为254，当前节点正好不负责编号为254的槽，那么就会返回客户端下面信息：

-MOVED 254 127.0.0.1:6381

表示客户端想要的254槽由运行在IP为127.0.0.1，端口为6381的Master实例服务。如果根据key计算得出的槽恰好由当前节点负责，则当期节点会立即返回结果

分片迁移

在一个稳定的Redis cluster下，每一个slot对应的节点是确定的，但是在某些情况下，节点和分片对应的关系会发生变更

\1. 新加入master节点

\2. 某个节点宕机

也就是说当动态添加或减少node节点时，需要将16384个槽做个再分配，槽中的键值也要迁移。当然，这一过程，在目前实现中，还处于半自动状态，需要人工介入。

新增一个主节点

新增一个节点D，redis cluster的这种做法是从各个节点的前面各拿取一部分slot到D上。大致就会变成这样：

节点A覆盖1365-5460

节点B覆盖6827-10922

节点C覆盖12288-16383

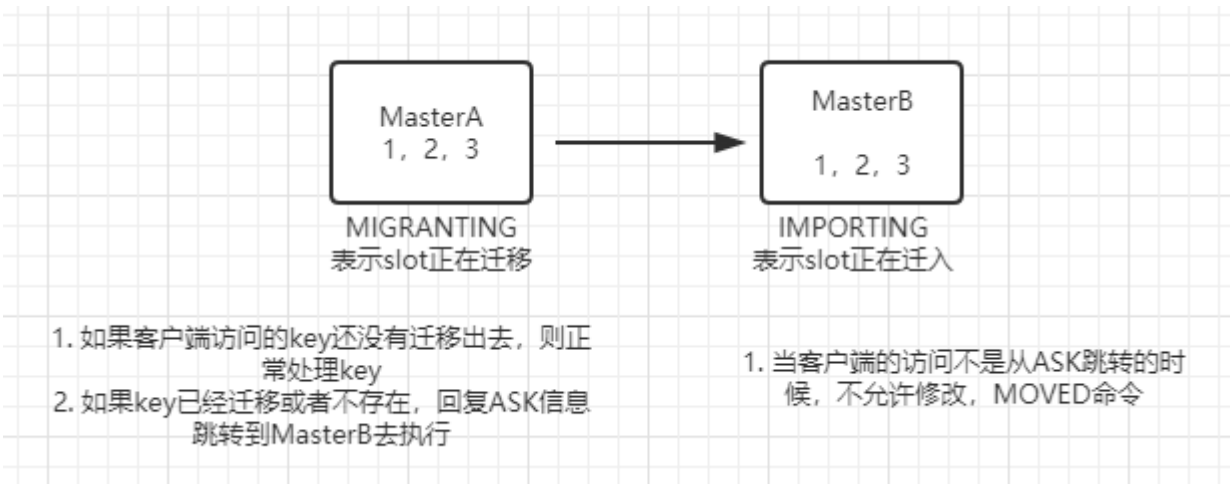
节点D覆盖0-1364,5461-6826,10923-12287

删除一个主节点

先将节点的数据移动到其他节点上，然后才能执行删除

槽迁移的过程

槽迁移的过程中有一个不稳定状态，这个不稳定状态会有一些规则，这些规则定义客户端的行为，从而使得Redis Cluster不必宕机的情况下可以执行槽的迁移。下面这张图描述了我们迁移编号为1、2、3的槽的过程中，他们在MasterA节点和MasterB节点中的状态。



简单的工作流程

\1. 向MasterB发送状态变更命令，吧Master B对应的slot状态设置为IMPORTING

\2. 向MasterA发送状态变更命令，将Master对应的slot状态设置为MIGRATING

当MasterA的状态设置为MIGRANTING后，表示对应的slot正在迁移，为了保证slot数据的一致性，MasterA此时对于slot内部数据提供读写服务的行为和通常状态下是有区别的，

MIGRATING状态

\1. 如果客户端访问的Key还没有迁移出去，则正常处理这个key

\2. 如果key已经迁移或者根本就不存在这个key，则回复客户端ASK信息让它跳转到MasterB去执行

IMPORTING状态

当MasterB的状态设置为IMPORTING后，表示对应的slot正在向MasterB迁入，及时Master仍然能对外提供该slot的读写服务，但和通常状态下也是有区别的

\1. 当来自客户端的正常访问不是从ASK跳转过来的，说明客户端还不知道迁移正在进行，很有可能操作了一个目前还没迁移完成的并且还存在于MasterA上的key，如果此时这个key在A上已经被修改了，那么B和A的修改则会发生冲突。所以对于MasterB上的slot上的所有非ASK跳转过来的操作，MasterB都不会uu出去护理，而是通过MOVED命令让客户端跳转到MasterA上去执行

这样的状态控制保证了同一个key在迁移之前总是在源节点上执行，迁移后总是在目标节点上执行，防止出现两边同时写导致的冲突问题。而且迁移过程中新增的key一定会在目标节点上执行，源节点也不会新增key，是的整个迁移过程既能对外正常提供服务，又能在一定的时间点完成slot的迁移。