**Redis Cluster的实现机制和原理**

**Redis Cluster主要特性和设计**

**集群目标**

    1）高性能和线性扩展，最大可以支撑到1000个节点，Cluster架构中无Proxy层，Master与slave之间使用异步replication，且不存在操作的merge（即操作不能跨多个nodes，不存在merge层）。

    2）一定程度上保证writes的安全性，需要客户端容忍一定程度的数据丢失。集群将会尽可能（best-effort）保存客户端write操作的数据，通常在failover期间，会有短暂时间内的数据丢失（因为异步replication引起），当客户端与少数节点处于网络分区时（network partition），丢失数据的可能性会更高，因节点有效性检测，failover需要更长的时间。

   3）只要集群中大多数master可达、且失效的master至少有一个slave可达时，集群都可以继续提供服务。同时replicas migration可以将那些拥有多个slaves的master的某个slave，迁移到没有slave的master下，即将slaves的分布在整个集群相对平衡，尽力确保每个master都有一定数量的slave备份。Redis Cluster集群由多个shard组成，每个shard可以有一个master和多个slaves构成，数据根据hash slots配额分布在多个shard节点上，节点之间建立双向TCP链接用于有效性检测、Failover等，Client直接与shard节点进行通讯，Cluster集群没有Proxy层，也没有中央式的Master用于协调集群状态或者state存储，集群暂不提供动态reblance策略。

**性能**

Redis Cluster并没有提供Proxy层，而是告知客户端将key的请求转发给合适的node。Client保存集群中nodes与keys的映射关系（slots），并保持此数据的更新，所以通常Client总能够将请求直接发送到正确的node上。因为采用异步replication，所以master不会等待slaves保存成功后才向客户端反馈结果，除非显式的指定了WAIT指令。multi-key指令仅限于单个节点内，除了resharding操作外，节点的数据不会在节点间迁移。每个操作只会在特定的一个节点上执行，所以集群的性能为master节点的线性扩展。同时Clients与每个node保持链接，所以请求的延迟等同于单个节点，即请求的延迟并不会因为Cluster的规模增大而受到影响。高性能和扩展性，同时保持合理的数据安全性，是Redis Cluster的设计目标。

Redis Cluster没有Proxy层，Client请求的数据也无法在nodes间merge。因为Redis核心就是K-V数据存储，没有scan类型（sort，limit，group by）的操作，因此merge操作并不被Redis Cluster所接受，而且这种特性会极大增加了Cluster的设计复杂度（类比于mongodb）。

**hash tags**

在计算hash slots时有一个意外的情况，用于支持“hash tags”，hash tags用于确保多个keys能够被分配在同一个hash slot中，每个key都可以包含一个自定义的“tags”，那么在存储时将根据tags计算此key应该分布在哪个node上（而不是使用key计算，但是存储层面仍然是key）。此特性可以强制某些keys被保存在同一个节点上，用于支持multi-key操作。hash tags的实现比较简单，key中“{}”之间的字符串就是当前key的hash tags，如果存在多个“{}”，首个符合规则的字符串作为hash tags，如果“{}”存在多级嵌套，那最内层首个完整的字符串作为hash tags，比如“{foo}.student”，那么“foo”是hash tags。如果key中存在合法的hash tags，那么在计算hash slots时，将使用hash tags，而不再使用原始的key，即“foo”与“{foo}.student”将得到相同的slot值，不过“{foo}.student”仍作为key来保存数据，即redis中数据的key仍为“{foo}.student”。

**Mutli-key操作**

Redis单实例支持的命令，Cluster也都支持，但是对于“multi-key”操作（即一次RPC调用需要进行多个key的操作）比如Set类型的交集、并集等，则要求这些key必须属于同一个node。Cluster不能进行跨Nodes操作，也没有nodes提供merge层代理。在人工对slots进行resharding期间，multi-key操作可能不可用。比如这些keys不存在于同一个slot（迁移会导致keys被分离）。比如Multikeys逻辑上属于同一个slot，但是因为resharding，它们可能暂时不处于同一个node，有些可能在迁移的目标节点上（比如Multikeys包含a、b、c三个keys，逻辑上它们都属于slot 8，但是其中c在迁移期间创建，它被存储在节点B上，a、b仍然在节点A），此时将会向客户端返回“-TRYAGAIN”错误，那么客户端此后将需要重试一次，或者直接返回错误（如果迁移操作被中断），无论如何最终Multikeys的访问逻辑是一致的，slots的状态也是最终确定的。

**Cluster Bus**

     集群中nodes负责存储数据，保持集群的状态，包括keys与nodes的对应关系（内部其实为slots与nodes对应关系）。nodes也能够自动发现其他的nodes，检测失效的节点，当某个master失效时还能将合适的slave提升为master。为了达成这些行为，集群中的每个节点都通过TCP与其他所有nodes建立连接，它们之间的通信协议和方式称为“Redis Cluster Bus”。 每个Node都有一个特定的TCP端口，用来接收其他nodes的链接，此端口号为面向Client的端口号 + 10000，比如果客户端端口号为6379，那么次node的BUS端口号为16379，客户端端口号可以在配置文件中声明。由此可见，nodes之间的交互通讯是通过Bus端口进行，使用gossip协议向其他node传播集群信息，以达到自动发现的特性，通过发送ping来确认其他nodes工作正常，也会在合适的时机发送集群的信息。当然在Failover时（包括人为failover）也会使用Bus来传播消息。

      gossip最终一致性，分布式服务数据同步算法，node首先需要知道（可以读取配置）集群中至少一个seed node，此node向seed发送ping请求，此时seed节点pong返回自己已知的所有nodes列表，然后node解析nodes列表并与它们都建立tcp连接，同时也会向每个nodes发送ping，并从它们的pong结果中merge出全局nodes列表，并逐步与所有的nodes建立连接，数据传输的方式也是类似，网络拓扑结构为full mesh。

**安全写入（write safety）**

    在Master-slaves之间使用异步replication机制，在failover之后，新的Master将会最终替代旧的master。在出现网络分区时（network partition），总会有个窗口期（node timeout）可能会导致数据丢失。不过，Client与多数派的Master、少数派Master处于一个分区（网络分区，因为网络阻断问题，导致Clients与Nodes被隔离成2部分）时，这两种情况下影响并不相同。

    1）write提交到master，master执行完毕后向Client反馈“OK”，不过此时可能数据还没有传播给slaves（异步replication），如果此时master不可达的时间超过阀值（node timeout），那么将触发slave被选举为新的Master（即Failover），这意味着那些没有replication到slaves的writes将永远丢失了。

    2）还有一种情况导致数据丢失：

        A）因为网络分区，此时master不可达，且Master与Client处于一个分区，且是少数派分区；

        B）Failover机制，将其中一个slave提升为新Master；

        C）此后网络分区消除，旧的Master再次可达，此时它将被切换成slave；

        D）那么在网络分区期间，处于少数派分区的Client仍然将write提交到旧的Master，因为它们觉得Master仍然有效，当旧的Master再次加入集群，切换成slave之后，这些数据将永远丢失。

在第二种情况下，如果Master无法与其他大多数Masters通讯的时间超过阀值后，此Master也将不再接收Writes，自动切换为readonly状态。当网络分区消除后，仍然会有一小段时间，客户端的write请求被拒绝，因为此时旧的Master需要更新本地的集群状态、与其他节点建立连接、角色切换为slave等等，同时Client端的路由信息也需要更新。只有当此master被大多数其他master不可达的时间达到阀值时，才会触发Failover，这个时间称为NODE\_TIMEOUT，可以通过配置设定。所以当网络分区在此时间内被消除的话，writes不会有任何丢失。反之，如果网络分区持续时间超过此值，处于“小分区”（minority）端的Master将会切换为readonly状态，拒绝客户端继续提交writes请求，那么“大分区”端将会进行failover，这意味着NODE\_TIMEOUT期间发生在“小分区”端的writes操作将丢失。

**集群节点的属性**

    集群中每个节点都有唯一的名字，称之为node ID，一个160位随机数字的16进制表示，在每个节点首次启动时创建。每个节点都将各自的ID保存在实例的配置文件中，此后将一直使用此ID，或者说只要配置文件不被删除，或者没有使用“CLUSTER RESET”指令重置集群，那么此ID将永不会修改。集群通过node ID来标识节点，而不是使用IP + port，因为node可以修改它的IP和port，如果ID不变，仍然认定它是集群中合法一员。集群可以在cluster Bus中通过gossip协议来探测IP、port的变更，并重新配置。

    node ID并不是与node相关的唯一信息，不过是唯一一个全局一致的。每个node还持有如下相关的信息，有些信息是关系集群配置的，其他的信息比如最后ping时间等。每个node也保存其他节点的IP、Port、flags（比如flags表示它是master还是slave）、最近ping的时间、最近pong接收时间、当前配置的epoch、链接的状态，最重要的是还包含此node上持有的hash slots。这些信息均可通过“CLUSTER NODES”指令开查看。

**集群拓扑**

    Redis Cluster中每个node都与其他node的Bus端口建立TCP链接（full mesh，全网）。比如在有N个节点的集群中，每个node有N-1个向外发出的TCP链接，以及N-1个其他node发过来的TCP链接。这些TCP链接总是keepalive，不是按需创建的。如果ping发出之后，node在足够长的时间内仍然没有pong响应，那么次node将会被标记为“不可达”，那么与此node的链接将会被刷新或者重建。Nodes之间通过gossip协议和配置更新的机制，来避免每次都交互大量的消息，最终确保在nodes之间的信息传送量是可控的。

**节点间handshake**

    Nodes通过Bus端口发送ping、pong，如果一个节点不属于集群，那么它的消息将会被其他nodes全部丢弃。一个节点被认为是集群成员的方式有2种：

    1）如果此node在“Cluster meet”指令中引入，此命令的主要意义就是将指定node加入集群。那么对于当前节点，将认为指定的node为“可信任的”（此后将会通过gossip协议传播给其他nodes）。

    2）当其他nodes通过gossip引入了新的nodes，这些nodes也是被认为是“可信任的”。

    只要将一个节点加入集群，最终此节点将会与其他节点建立链接，即cluster可以通过信息交换来自动发现新的节点，链接拓扑仍然是full mesh。

**重定向与resharding**

**MOVED重定向**

  因为redis并不提供Proxy机制，当Client将请求发给错误的node时（此node上不存在此key所属的slot），node将会反馈“MOVED”或者“ASK”错误信息，以便Client重新定向到合适的node。理论上，Client可以将请求随意发给任何一个node，包括slaves，此node解析query，如果可以执行（比如语法正确，multiple keys都应该在一个node slots上），它会查看key应该属于哪个slot、以及此slot所在的node，如果当前node持有此slot，那么query直接执行即可，否则当前node将会向Client反馈“MOVED”错误。错误信息中包括此key对应的slot（3999），以及此slot所在node的ip和port，对于Client 而言，收到MOVED信息后，它需要将请求重新发给指定的node。不过，当node向Client返回MOVED之前，集群的配置也在变更（节点调整、resharding、failover等，可能会导致slot的位置发生变更），此时Client可能需要等待更长的时间，不过最终node会反馈MOVED信息，且信息中包含指定的新的node位置。虽然Cluster使用ID标识node，但是在MOVED信息中尽可能的暴露给客户端便于使用的ip + port。

当Client遇到“MOVED”错误时，将会使用“CLUSTER NODES”或者“CLUSTER SLOTS”指令获取集群的最新信息，主要是nodes与slots的映射关系。因为遇到MOVED，一般也不会仅仅一个slot发生的变更，通常是一个或者多个节点的slots发生了变化，所以进行一次全局刷新是有必要的。Client将会把集群的这些信息缓存，以便提高query的性能。还有一个错误信息“ASK”，它与“MOVED”都属于重定向错误，客户端的处理机制基本相同，只是ASK不会触发Client刷新本地的集群信息。

**ASK重定向**

MOVED重定向与ASK非常相似。在resharding期间，为什么不能用MOVED？MOVED意思为hash slots已经永久被另一个node接管，接下来相应的查询应该与它交互，ASK的意思是当前query暂时与指定的node交互。在迁移期间，slot 8的keys有可能仍在A上，所以Client的请求仍然需要首先经由A，对于A上不存在的，才需要到B上进行尝试。迁移期间，Redis Cluster并没有粗暴的将slot 8的请求全部阻塞直到迁移结束，这种方式尽管不再需要ASK，但是会影响集群的可用性。

    1）当Client接收到ASK重定向，它仅仅将当前query重定向到指定的node，此后的请求仍然交付给旧的节点。

    2）客户端并不会更新本地的slots映射，仍然保持slot 8与A的映射，直到集群迁移完毕，且遇到MOVED重定向。一旦slot 8迁移完毕之后（集群的映射信息也已更新），如果Client再次在A上访问slot 8时，将会得到MOVED重定向信息，此后客户端也更新本地的集群映射信息。

**客户端首次链接以及重定向处理**

    可能有些Cluster客户端的实现，不会在内存中保存slots映射关系（即nodes与slots的关系），每次请求都从声明的、已知的nodes中，随机访问一个node，并根据重定向（MOVED）信息来寻找合适的node，这种访问模式，通常是非常低效的。当然，Client应该尽可能的将slots配置信息缓存在本地，不过配置信息也不需要绝对的实时更新，因为在请求时偶尔出现“重定向”，Client也能兼容此次请求的正确转发，此时再更新slots配置，所以Client通常不需要间歇性的检测Cluster中配置信息是否已经更新，客户端通常是全量更新slots配置：

    1）首次链接到集群的某个节点；

    2）当遇到MOVED重定向消息时。

    遇到MOVED时，客户端仅仅更新特定的slot是不够的，因为集群中的reshard通常会影响到多个slots。客户端通过向任意一个node发送“CLUSTER NODES”或者“CLUSTER SLOTS”指令均可以获得当前集群最新的slots映射信息，“CLUSTER SLOTS”指令返回的信息更易于Client解析。如果集群处于broken状态，即某些slots尚未被任何nodes覆盖，指令返回的结果可能是不完整的。

**集群运行时重新配置（live reconfiguration）**

    在Cluster运行时增加、删除node，这两种操作都会导致slots在nodes的迁移，当然这种机制也可用来集群数据的rebalance等等。

    1）集群中新增一个node，需要将其他node上的部分slots迁移到此node上，以实现数据负载的均衡分配。

    2）集群中移除一个node，那么在移除节点之前，必须将此节点上（如果此节点没有任何slaves）的slots迁移到其他nodes。

    3）如果数据负载不均衡，比如某些slots数据集较大、负载较大时，需要把它们迁移到负载较小的nodes上（即手动resharding），以实现集群的负载平衡。

  Cluster支持slots在nodes间移动，从实际的角度来看，一个slot只是一序列keys的逻辑标识，所以Cluster中slots的迁移，其实就是一序列keys的迁移，不过resharding操作只能以slot为单位（而不能仅仅迁移某些keys）。Redis提供了如下几个操作：

    1）[CLUSTER ADDSLOTS](http://redis.io/commands/cluster-addslots) [slot] ....

    2）[CLUSTER DELSLOTS](http://redis.io/commands/cluster-delslots) [slot] ...

    3）CLUSTER SETSLOT [slot] NODE [node]

    4）CLUSTER SETSLOT [slot] MIGRATING [destination-node]

    5）CLUSTER SETSLOT [slot] IMPORTING [source-node]

前两个指令ADDSLOTS和DELSLOTS，用于向当前node分配或者移除slots，指令可以接受多个slot值。分配slots的意思是告知指定的master（即此指令需要在某个master节点执行）此后由它接管相应slots的服务。slots分配后，这些信息将会通过gossip发给集群的其他nodes。

    ADDSLOTS指令通常在创建一个新的Cluster时使用，一个新的Cluster有多个空的Masters构成，此后管理员需要手动为每个master分配slots，并将16384个slots分配完毕，集群才能正常服务。简而言之，ADDSLOTS只能操作那些尚未分配的（即不被任何nodes持有）slots，通常在创建新的集群或者修复一个broken的集群（集群中某些slots因为nodes的永久失效而丢失）时使用。为了避免出错，Redis Cluster提供了一个redis-trib辅助工具。

    DELSLOTS就是将指定的slots删除，前提是这些slots必须在当前node上，被删除的slots处于“未分配”状态（当然其对应的keys数据也被clear），即尚未被任何nodes覆盖，这种情况可能导致集群处于不可用状态，此指令通常用于debug，在实际环境中很少使用。那些被删除的slots，可以通过ADDSLOTS重新分配。

  SETSLOT是个很重要的指令，对集群slots进行reshard的最重要手段，它用来将单个slot在两个node间迁移。根据slot的操作方式，它有两种状态“MIGRATING”、“IMPORTING”（或者说迁移的方式）。

    1）MIGRATING：将slot的状态设置为“MIGRATING”，并迁移到destination-node上，需要注意当前node必须是slot的持有者。在迁移期间，Client的查询操作仍在当前node上执行，如果key不存在，则会向Client反馈“-ASK”重定向信息，此后Client将会把请求重新提交给迁移的目标node。

    2）IMPORTING：将slot的状态设置为“IMPORTING”，并将其从source-node迁移到当前node上，前提是source-node必须是slot的持有者。Client交互机制同上。

  假如有两个节点A、B，其中slot 8在A上，要将8从A迁移到B，可以使用如下方式：

    1）在B上：CLUSTER SETSLOT 8 IMPORTING A

    2）在A上：CLUSTER SETSLOT 8 MIGRATING B

    在迁移期间，集群中其他的nodes的集群信息不会改变，即slot 8仍对应A，即此期间，Client查询仍在A上：

    1）如果key在A上存在，则由A执行；

    2）否则，将向客户端返回ASK，客户端将请求重定向到B。

    这种方式下，新key的创建就不会在A上执行，而是在B上执行，这也就是ASK重定向的原因（迁移之前的keys在A，迁移期间created的keys在B上）。当上述SETSLOT执行完毕后，slot的状态也会被自动清除，同时将slot迁移信息传播给其他nodes，至此集群中slot的映射关系将会变更，此后slot 8的数据请求将会直接提交到B上。

**slave扩展read请求**

    通常情况下，read、write请求都将由持有slots的master节点处理。redis的slaves同样可以支持read操作（前提是application能够容忍stale数据），所以客户端可以使用“READONLY”指令来扩展read请求。

    “READONLY”表明其可以访问集群的slaves节点，能够容忍stale数据，而且此次链接不会执行writes操作。当链接设定为readonly模式后，Cluster只有当keys不被slave的master节点持有时才会发送重定向消息（即Client的read请求总是发给slave，只有当此slave的master不持有slots时才会重定向），此时Client更新本地的slot配置信息，同上文所述（目前很多Client实现均基于连接池，所以不能非常便捷的设置READLONLY选项，非常遗憾）。

**容错（Fault Tolerance）**

**心跳与gossip消息**

    集群中的nodes持续的交换ping、pong数据，这两种数据包的结构一样，同样都能携带集群的配置信息，唯一不同的就是message中的type字段。通常一个node发送ping消息，那么接收者将会反馈pong消息，不过有时候并非如此，或许接收者将pong信息发给其他的nodes，而不是直接反馈给发送者，比如当集群中添加新的node时。通常一个node每秒都会随机向几个nodes发送ping，所以无论集群规模多大，每个nodes发送的ping数据包的总量是恒定的。每个node都确保尽可能的向那些在半个NODE\_TIMEOUT时间内，尚未发送过ping或者接收到它们的pong消息的nodes发送ping。在NODE\_TIMEOUT逾期之前，nodes也会尝试与那些通讯异常的nodes重新建立TCP链接，确保不能仅仅因为当前链接异常而认为它们就是不可达的。

 当NODE\_TIMEOUT值较小、集群中nodes规模较大时，那么全局交换的信息量也会非常庞大，因为每个node都尽力在半个NODE\_TIMEOUT时间内，向其他nodes发送ping。比如有100个nodes，NODE\_TIMEOUT为60秒，那么每个node在30秒内向其他99各nodes发送ping，平均每秒3.3个消息，那么整个集群全局就是每秒330个消息。这些消息量，并不会对集群的带宽带来不良问题。ping和pong数据包中也包含gossip部分，这部分信息包含sender持有的集群视图，不过它只包含sender已知的随机几个nodes，nodes的数量根据集群规模的大小按比例计算。gossip部分包含了nodes的ID、ip+port、flags，那么接收者将根据sender的视图，来判定节点的状态，这对故障检测、节点自动发现非常有用。

**心跳数据包的内容**

    1）node ID；

    2）currentEpoch和configEpoch；

    3）node flags：比如表示此node是maste、slave等；

    4）hash slots：发送者持有的slots；

    5）如果发送者是slave，那么其master的ID；

    6）其他..

**失效检测**

    集群失效检测就是当某个master或者slave不能被大多数nodes可达时，用于故障迁移并将合适的slave提升为master。当slave提升未能有效实施时，集群将处于error状态且停止接收Client端查询。如上所述，每个node持有其已知nodes的列表包括flags，有2个flag状态：PFAIL和FAIL。PFAIL表示“可能失效”，是一种尚未完全确认的失效状态（即某个节点或者少数masters认为其不可达）。FAIL表示此node已经被集群大多数masters判定为失效（大多数master已认定为不可达，且不可达时间已达到设定值，需要failover）。

**PFAIL**

    一个被标记为PFAIL的节点，表示此node不可达的时间超过NODE\_TIMEOUT，master和slave有可能被标记为PFAIL。所谓不可达，就是当“active ping”（发送ping且能收到pong）尚未成功的时间超过NODE\_TIMEOUT，因此设定的NODE\_TIMEOUT的值应该比网络交互往返的时间延迟要大一些（通常要大的多，以至于交互往返时间可以忽略）。为了避免误判，当一个node在半个NODE\_TIMEOUT时间内仍未能接收pong，那么当前node将会尽力尝试重新建立连接进行重试，以排除pong未能接收是因为当前链接故障的问题。

**FAIL**

    PFAIL只是当前node有关于其他nodes的本地视图，可能每个node对其他nodes的本地视图都不一样，所以PFAIL还不足以触发Failover。处于PFAIL状态下的node可能会被提升到FAIL状态。如上所述，每个node在向其他nodes发送gossip消息时，都会包含本地视图中几个随机nodes的状态信息，每个node最终都会从其他nodes发送的消息中获得一组nodes的flags。因此，每个node都可以通过这种机制来通知其他nodes它检测到的故障情况。

    PFAIL被上升为FAIL的集中情况：

    1）比如A节点，认为B为PFAIL；

    2）那么A通过gossip信息，收集集群中大多数masters关于B的状态视图；

    3）多数master都认为B为PFAIL，或者PFAIL情况持续时间为NODE\_TIMEOUT \* FAIL\_REPORT\_VALIDITY\_MULT（此值当前为2）。

  如果上述条件成立，那么A将会：

    1）将B节点设定为FAIL；

    2）将FAIL信息发送给其所有能到达的所有节点。

  每个接收到FAIL消息的节点都会强制将此node标记为FAIL状态，不管此节点在本地视图中是否为PFAIL。FAIL状态是单向的，即PFAIL可以转换为FAIL，但是FAIL状态只能清除，不能回转为PFAIL：

    1）当此node已经变的可达，且为slave，这种情况下FAIL状态将会被清除，因为没有发生failover。

    2）此node已经可达，且是一个没有服务任何slots的master（空的master）。这种情况下，FAIL将会被清除，因为master没有持有slots，所以它并没有真正参与到集群中，需要等到重新配置以便它加入集群。

    3）此node已经可达，且是master，且在较长时间内（N倍的NODE\_TIMEOUT）没有检测到slave的提升，即没有slave发生failover（比如此master下没有slave），那么它只能重新加入集群且仍为master。需要注意的是PFAIL->FAIL的转变，使用了“协议”（agreement）的形式：

    1）nodes会间歇性的收集其他nodes的视图，即使大多数masters都“agree”，事实上这个状态仅仅是从不同的nodes、不同的时间收集到的，无法确认（也不需要）在特定时刻大多数masters是否“agree”。丢弃较旧的故障报告，所以此故障（FAIL）是有大多数masters在一段时间内的信号。

    2）虽然每个node在检测到FAIL情况时，都会通过FAIL消息发送给其他nodes，但是无法保证消息一定会到达所有的nodes，比如可能当前节点（发送消息的node）因为网络分区与其他部分nodes隔离了，如果只有少数master认为某个node为FAIL，并不会触发相应的slave提升，即failover。FAIL标记只是用来触发slave 提升，在原理上，当master不可达时将会触发slave提升，不过当master仍然被大多数可达时，它会拒绝提供相应的确认。

**Failover相关的配置**

**集群currentEpoch**

    Redis Cluster使用了类似于Raft算法term（任期）的概念，那么在redis Cluster中term称为epoch，用来给events增量版本号。当多个nodes提供的信息有冲突时，它可以作为node知道哪个状态是最新的。currentEpoch为一个64位无签名数字。

    在集群node创建时，master和slave都会将各自的currentEpoch设置为0，每次从其他node接收到数据包时，如果发现发送者的epoch值比自己的大，那么当前node将自己的currentEpoch设置为发送者的epoch。由此，最终所有的nodes都会认同集群中最大的epoch值。当集群状态变更或者为了执行某个行为需求agreement时，都将需要epoch（传递或者比较）。当前来说，只有在slave提升期间发生。currentEpoch为集群的逻辑时钟（logical clock），指使持有较大值的获胜（currentEpoch，当前集群已达成认同的epoch值，通常所有的nodes应该一样）。

**configEpoch**

    每个master总会在ping、pong数据包中携带自己的configEpoch以及它持有的slots列表。新创建的node，其configEpoch为0，slaves通过递增它们的configEpoch来替代失效的master，并尝试获得其他大多数master的授权（认同）。当slave被授权，一个新的configEpoch被生成，slave提升为master且使用此configEpoch。接下来介绍configEpoch帮助解决冲突，当不同的nodes宣称有分歧的配置时，slaves在ping、pong数据包中也会携带自己的configEpoch信息，不过这个epoch为它与master在最近一次数据交换时master的configEpoch。每当节点发现configEpoch值变更时，都会将新值写入nodes.conf文件，当然currentEpoch也是如此。这两个变量在写入文件后会伴随磁盘的fsync持久写入。严格来说，集群中所有的master都持有唯一的configEpoch值。同一组master-slaves持有相同的configEpoch。

**slave选举与提升**

    在slaves节点中进行选举，在其他masters的帮助下进行投票，选举出一个slave并提升为master。当master处于FAIL状态时，将会触发slave的选举。slaves都希望将自己提升为master，此master的所有slaves都可以开启选举，不过最终只有一个slave获胜。当如下情况满足时，slave将会开始选举：

    1）当此slave的master处于FAIL状态；

    2）此master持有非零个slots；

    3）此slave的replication链接与master断开时间没有超过设定值，为了确保此被提升的slave的数据是新鲜的，这个时间用户可以配置。

为了选举，第一步就是slave自增currentEpoch值，然后向其他masters请求投票（需求支持votes）。slave通过向其他masters传播“FAILOVER\_AUTH\_REQUEST”数据包，然后最长等待2倍的NODE\_TIMEOUT时间来接收反馈。一旦一个master向此slave投票，将会响应“FAILOVER\_AUTH\_ACK”，此后在2 \* NODE\_TIMOUT时间内，它将不会向同一个master的slave投票。虽然这对保证安全上没有必要，但是对避免多个slaves同时选举时有帮助。slave将会丢弃那些epoch值小于自己的currentEpoch的AUTH\_ACK反馈，即不会对上一次选举的投票计数（只对当前轮次的投票计数）。一旦此slave获取了大多数master的ACKs，它将在此次选举中获胜。否则如果大多数master不可达（在2 \* NODE\_TIMEOUT）或者投票额不足，那么它的选举将会被中断，那么其他的slave将会继续尝试。

**slave rank（次序）**

    当master处于FAIL状态时，slave将会随机等待一段时间，然后才尝试选举，等待的时间：

DELAY = 500ms + random(0 ~ 500ms) + SLAVE\_RANK \* 1000ms。一定的延迟确保等待FAIL状态在集群中传播，否则slave立即尝试选举（不进行等待的话），不顾此时其他masters尚未意识到FAIL状态，可能会拒绝投票。

延迟的时间是随机的，这用来“去同步”（desynchronize），避免slaves同时开始选举。SLAVE\_RANK表示此slave已经从master复制数据的总量的rank。当master失效时，slaves之间交换消息以尽可能的构建rank，持有replication offset最新的rank为0，第二最新的为1，依次轮推。这种方式下，持有最新数据的slave将会首先发起选举（理论上）。当然rank顺序也不是严格执行的，如果一个持有较小rank的slave选举失败，其他slaves将会稍后继续。一旦slave选举成功，它将获取一个新的、唯一的、自增的configEpoch值，此值比集群中任何masters持有的都要大，它开始宣称自己是master，并通过ping、pong数据包传播，并提供自己的新configEpoch以及持有的slots列表。为了加快其他nodes的重新配置，pong数据包将会在集群中广播。当前node不可达的那些节点，它们可以从其他节点的ping或者pong中获知信息（gossip），并重新配置。其他节点也会检测到这个新的master和旧master持有相同的slots，且持有更高的configEpoch，此时也会更新自己的配置（epoch以及master），旧master的slaves不仅仅更新配置信息，也会重新配置并与新的master跟进（slave of）。

**Masters响应slave的投票请求**

    当Master接收到slave的“FAILOVER\_AUTH\_REQUEST”请求后，开始投票，不过需要满足如下条件：

    1）此master只会对指定的epoch投票一次，并拒绝对旧的epoch投票。每个master都持有一个

lastVoteEpoch，将会拒绝AUTH\_REQUEST中currentEpoch比lastVoteEpoch小的请求。当master响应投票

时，将会把lastVoteEpoch保存在磁盘中；

    2）此slave的master处于FAIL状态时，master才会投票；

    3）如果slave的currentEpoch比此master的currentEpoch小，AUTH\_REQUEST将被忽略。因为masters只会响应那些与自己的currentEpoch相等的请求。如果同一个slave再次请求投票，持有已经增加的

currentEpoch，它（slave）将保证旧的投票响应不能参与计票。

比如master的currentEpoch为5，lastVoteEpoch为1：

    1）slave的currentEpoch为3；

    2）slave在选举开始时，使用epoch为4（先自增），因为小于master的epoch，所以投票响应被延缓；

    3）slave在一段时间后将重新选举，使用epoch为5（4 + 1，再次自增），此时master上延缓的响应发给

slave，接收后视为有效。

**Hash Slots配置传播**

    Redis Cluster中重要的一部分就是传播集群中哪些节点上持有哪些hash slots信息。无论是启动一个新的集群，还是当master失效其slave提升后更新配置，这对它们都至关重要。有2种方式用于hash slot配置的传播：

    1）heartbeat 消息：发送者的ping、pong消息中，总是携带自己目前持有的slots信息，不管自己是master还是slave。

    2）UPDATE 消息：因为每个心跳消息中会包含发送者的configEpoch和其持有的slots，如果接收者发现发送者的信息已经stale（比如发送者的configEpoch值小于持有相同slots的master的值），它会向发送者反馈新的配置信息（UPDATE），强制stale节点更新它。

当一个新的节点加入集群，其本地的hash slots映射表将初始为NULL，即每个hash slot都没有与该节点绑定。如果此node本地视图中一个hash slot尚未分配（设置为NULL），并且有一个已知的node声明持有它，那么此node将会修改本地hash slot的映射表，将此slot与那个node关联。slave的failover操作、reshard操作都会导致hash slots映射的变更，新的配置信息将会通过心跳在集群中传播。如果此node的本地视图中一个hash slot已经分配，并且一个已知的node也声明持有它，且此node的configEpoch比当前slot关联的master的configEpoch值更大，那么此node将会把slot重新绑定到新的node上。根据此规则，最终集群中所有的nodes都赞同那个持有声明持有slot、且configEpoch最大值的node为slot的持有者。

**nodes如何重新加入集群**

    node A被告知slot 1、2现在由node B接管，假如这两个slots目前由A持有，且A只持有这两个slots，如果A将放弃这2个slots，成为空的节点。如果A被重新配置，成为其他新master的slave。这个规则可能有些复杂，A离群一段时间后重新加入集群，此时A发现此前自己持有的slots已经被其他多个nodes接管，比如slot 1被B接管，slot 2被C接管。在重新配置时，最终此节点上的slots将会被清空，那个窃取A最后一个slot的node，将成为它的新master。节点重新加入集群，通常发生在failover之后，旧的master（也可以为slave）离群，然后重新加入集群。

**Replica迁移**

    Redis Cluster实现了一个成为“Replica migration”的概念，用来提升集群的可用性。比如集群中每个master都有一个slave，当集群中有一个master或者slave失效时，而不是master与它的slave同时失效，集群仍然可以继续提供服务。

    1）master A，有一个slave A1；

    2）master A失效，A1被提升为master；

    3）一段时间后，A1也失效了，那么此时集群中没有其他的slave可以接管服务，集群将不能继续服务。

  如果masters与slaves之间的映射关系是固定的（fixed），提高集群抗灾能力的唯一方式，就是给每个master增加更多的slaves，不过这种方式开支很大，需要更多的redis实例。解决这个问题的方案，可以将集群非对称，且在运行时可以动态调整master-slave的布局（而不是固定master-slave的映射），比如集群中有三个master A、B、C，它们对应的slave为A1、B1、C1、C2，即C节点有2个slaves。Replica迁移可以自动的重新配置slave，将其迁移到某个没有slave的master下。

    1）A失效，A1被提升为master；

    2）此时A1没有任何slave，但是C仍然有2个slave，此时C2被迁移到A1下，成为A1的slave；

    3）此后某刻A1失效，那么C2将被提升为master，集群可以继续提供服务。

**Replica迁移算法**

    迁移算法并没有使用“agree”形式，而是使用一种算法来避免大规模迁移，这个算法确保最终每个master至少有一个slave即可。起初先定义哪个slave是良好的，一个良好的slave不能处于FAIL状态，触发时机为任何一个slave检测到某个master没有一个良好slave时。参与迁移的slave必须为持有最多slaves的master的其中一个slave，且不处于FAIL状态，且持有最小的node ID。比如有10个masters都持有一个slave，有2个masters各持有5个slaves，那么迁移将会发生在持有5个slaves的masters中，且node ID最小的slave node上。这个时候就不再使用“agreement”，不过也有可能当集群的配置不够稳定时，有一种竞争情况的发生，即多个slaves都认为自己的ID最小。如果这种情况发生，结果就是可能多个slaves会迁移到同一个master下，不过这并没有什么害处，但是最坏的结果是导致原来的master迁出了所有的slaves，让自己变得单一。但是迁移算法（进程）会在迁移完毕之后重新判断，如果尚未平衡，那么将会重新迁移，最终每个master最少持有一个slave。这个算法由用户配置的“cluster-migration-barrier”，此配置参数表示一个master至少保留多少个slaves，其他多余的slaves可以被迁出。此值通常为1，如果设置为2，表示一个master持有的slaves个数大于2时，多余的slaves才可以迁移到持有更少slaves的master下。

**configEpoch冲突解决算法**

    在slave failover期间，会生成新的configEpoch值，需要保证唯一性。有2种不同的event会导致configEpoch的创建是不安全的。仅仅自增本地的currentEpoch并希望它不会发生冲突。这两个事件由系统管理员触发：

    1）CLUSTER FAILOVER：这个指令，就是人为的将某个slave提升为master，而不需要大多数masters的投票参与。

    2）slots的迁移，用于平衡集群的数据分布（reshard）。此时本地的configEpoch也会修改，因为性能的考虑，这个过程也不需要“agreement”。

  在手动reshard期间，当一个hash slot从A迁移到B，resharding程序将强制B更新自己的配置信息、epoch值也修改为集群的最大值 + 1（除非B的configEpoch已经是最大值），这种变更则不需要其他nodes的agreement（注意与failover的原理不同）。通常每次resharding都会迁移多个slots，且有多个nodes参与，如果每个slots迁移都需要agreement，才能生成新的epoch，这种性能是很差的，也不可取。在首个slots迁移开始时，只会生成一个新的configEpoch，在迁移完毕后，将新的配置传播给集群即可，这种方式在生产环境中更加高效。因为上述两个情况，有可能（虽然概率极小）最终多个nodes产生了相同的configEpoch。比如管理员正在进行resharding，但是此时failover发生了...无论是failover还是resharding都是将currentEpoch自增，而且resharding不使用agreement形式（即其他nodes或许不知道，而且网络传播可能延迟），这就会发生epoch值的冲突问题。当持有不同slots的masters持有相同的configEpoch，这并不会有什么问题。比较遗憾的是，人工干预或者resharding会以不同的方式修改了集群的配置，Cluster要求所有的slots都应该被nodes覆盖，所以在任何情况下，都希望所有的master都持有不同的configEpoch。避免冲突的算法，就是用来解决当2个nodes持有相同的configEpoch：

    1）如果一个master节点发现其他master持有相同的configEpoch；

    2）并且此master逻辑上持有较小的node ID（字典顺序）；

    3）然后此master将自己的currentEpoch加1，并作为自己新的configEpoch。

  如果有多个nodes持有相同的congfigEpoch，那么除了持有最大ID的节点外，其他的nodes都将往前推进（+1，直到冲突解决），最终保证每个master都持有唯一的configEpoch（slave的configEpoch与master一样）。对于新创建的cluster也是同理，所有的nodes都初始为不同的configEpoch。

**Node resets**

    所有的nodes都可以进行软件级的reset（不需要重启、重新部署它们），reset为了重用集群（重新设定集群），必须需要将某个（些）节点重置后添加到其他集群。可以使用“CLUSTER RESET”指令：

    1）CLUSTER RESET SOFT

    2）CLUSTER RESET HARD

  指令必须直接发给需要reset的节点，如果没有指定reset类型，默认为SOFT。

    1）soft和hard：如果节点为slave，那么节点将会转换为master，并清空其持有的数据，成为一个空的master。如果此节点为master，且持有slots数据，那么reset操作将被中断；

    2）soft和hard：其上持有的slots将会被释放；

    3）soft和hard：此节点上的nodes映射表将会被清除，此node将不会知道其他节点的存在与状态；

    4）hard：currentEpoch、configEpoch、lastVoteEpoch值将被重置为0；

    5）hard：此nodeID将会重新生成。

  持有数据的（slot映射不为空的）master不能被reset（除非现将此master上的slot手动迁移到其他nodes上，或者手动failover，将其切换成slave）。在某些特定的场景下，在执行reset之前，或许需要执行FLUSHALL来清空原有的数据。

**集群中移除节点**

    将node移除集群之前，首先将其上的slots迁移到其他nodes上（reshard），然后关闭它。不过这似乎还并未结束，因为其他nodes仍然记住了它的ID，仍然不会终止与它建立连接。因此，将节点移除集群时，可以使用“CLUSTER FORGET <node-ID>”指令：

    1）将此node从nodes映射表中移除；

    2）然后设定一个60秒的隔离时间，阻止持有相同ID的node再次加入集群。

  之所以出现2规则，因为FORGET指令将会通过gossip协议传播给其他nodes，集群中所有的节点都收到消息是需要一定的时间延迟。

**集群架构**

每个节点都会跟其他节点保持连接，用来交换彼此的信息。节点组成集群的方式使用cluster meet命令，meet命令可以让两个节点相互握手，然后通过gossip协议交换信息。如果一个节点r1在集群中，新节点r2加入的时候与r1节点握手，r1节点会把集群内的其他节点信息通过gossip协议发送给r2，r2会一一与这些节点完成握手，从而加入到集群中。节点在启动的时候会生成一个全局的标识符，并持久化到配置文件，在节点与其他节点握手后，这些信息也都持久化下来。节点与其他节点通信，标识符是它唯一的标识，而不是IP、PORT地址。如果一个节点移动位置导致IP、PORT地址发生变更，集群内的其他节点能把该节点的IP、PORT地址纠正过来。

**集群数据分布**

集群数据以数据分布表的方式保存在各个slot上。默认的数据分布表含有16384个slot。key与slot映射使用的CRC16算法，即slot = CRC16(key) mod 16384。集群只有在16384个slot都有对应的节点才能正常工作。slot可以动态的分配、删除和迁移。每个节点会保存一份数据分布表，节点会将自己的slot信息发送给其他节点，发送的方式使用一个unsigned char的数组，数组长度为16384/8。每个bit标识为0或者1来标识某个slot是否是它负责的。使用这样的方式传输，就能大大减少数据分布表的字节。这种方式使用的字节为2048，如果纯粹的传递数据分布表，那边一个slot至少需要2字节的slot值+2字节port+4字节ip，共8\*16384=131072字节，或者以ip、port为键，把节点对应的slot放在后面，那么也至少需要2\*16384加上节点的数量，也远远大于2048字节。由于节点间不停的在传递数据分布表，所以为了节省带宽，redis选择了只传递自己的分布数据。但这样的方式也会带来管理方面的麻烦，如果一个节点删除了自己负责的某个slot，这样该节点传递给其他节点数据分布表的slot标识为0，而redis采用了bitmapTestBit方法，只处理slot为1的节点，而并未把每个slot与收到的数据分布表对比，从而产生了节点间数据分布表视图的不一致。这种问题目前只能通过使用者来避免。  
**集群访问**

客户端在初始化的时候只需要知道一个节点的地址即可，客户端会先尝试向这个节点执行命令，比如“get key”，如果key所在的slot刚好在该节点上，则能够直接执行成功。如果slot不在该节点，则节点会返回MOVED错误，同时把该slot对应的节点告诉客户端。客户端可以去该节点执行命令。目前客户端有两种做法获取数据分布表，一种就是客户端每次根据返回的MOVED信息缓存一个slot对应的节点，但是这种做法在初期会经常造成访问两次集群。还有一种做法是在节点返回MOVED信息后，通过cluster nodes命令获取整个数据分布表，这样就能每次请求到正确的节点，一旦数据分布表发生变化，请求到错误的节点，返回MOVED信息后，重新执行cluster nodes命令更新数据分布表。

订阅在使用上与单机版没有任何区别。订阅功能通过集群间共享publish消息实现的，客户端可以向任意一个节点（包括slave节点）订阅消息，然后在一个节点上执行publish命令，该节点会把该命令传播给集群内的每个节点，这样订阅该消息的客户端就能收到该命令。  
 redis集群只使用db0，select命令虽然能够支持select 0，其他的db都会返回错误。  
127.0.0.1:6379> select 0  
OK  
127.0.0.1:6379> select 1  
(error) ERR SELECT is not allowed in cluster mode  
 redis集群对多key命令的支持，只支持多key都在同一个slot上，即使多个slot在一个节点上也不行。  
127.0.0.1:6379> mget key7 key28  
(error) CROSSSLOT Keys in request don't hash to the same slot  
 事务的支持只能在一个slot上完成。MULTI命令之后的命令的key必须都在同一个slot上，如果某条命令的key对应不在相同的slot上，则事务直接回滚。迁移的时候，在迁移源节点执行命令的key必须在迁移原节点上存在，否则事务就会回滚。在迁移目的节点执行的时候需要先执行ASKING命令再执行MULTI命令，这样接下来该slot的命令都能被执行。可以看出，对于单key和相同hash tags的事务，集群还是能很好的支持。在迁移的时候有个地方需要注意，对于多key命令在迁移目的节点执行时，如果多个key全在该节点上，则命令无法执行。如下所示，key和key14939对应的slot为12539，执行命令的节点是迁移目的节点：  
127.0.0.1:6379> asking  
OK  
127.0.0.1:6379> mget key key14939  
(error) TRYAGAIN Multiple keys request during rehashing of slot  
**集群消息**

集群间互相发送消息，使用另外的端口，所有的消息在该端口上完成，可以成为消息总线，这样可以做到不影响客户端访问redis，可见redis对于性能的追求。目前集群有如下几种消息：  
CLUSTERMSG\_TYPE\_PING：gossip协议的ping消息。  
CLUSTERMSG\_TYPE\_PONG：gossip协议的pong消息。  
CLUSTERMSG\_TYPE\_MEET：握手消息。  
CLUSTERMSG\_TYPE\_FAIL：master节点检测到超过半数master认为某master离线，则发送fail消息。  
CLUSTERMSG\_TYPE\_PUBLISH：publish消息，向其他节点推送消息。  
CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_REQUEST：故障转移时，slave发送向其他master投票请求。  
CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK：故障转移时，其他master回应slave的请求。  
CLUSTERMSG\_TYPE\_UPDATE：通知某节点，它负责的某些slot被另一个节点替换。  
CLUSTERMSG\_TYPE\_MFSTART：手动故障转移时，slave请求master停止访问，从而对比两者的数据偏移量，可以达到一致。  
 集群节点间相互通信使用了gossip协议的push/pull方式，ping和pong消息，节点会把自己的详细信息和已经和自己完成握手的3个节点地址发送给对方，详细信息包括消息类型，集群当前的epoch，节点自己的epoch，节点复制偏移量，节点名称，节点数据分布表，节点master的名称，节点地址，节点flag，节点所处的集群状态。节点根据自己的epoch和对方的epoch来决定哪些数据需要更新，哪些数据需要告诉对方更新。然后根据对方发送的其他地址信息，来发现新节点的加入，从而和新节点完成握手。  
 节点默认每秒在集群中的其他节点选择一个节点，发送ping消息。选择节点步骤是：  
 1、随机 5 个节点；  
 2、跳过断开连接和已经在ping还没收到pong响应的节点；  
 3、从筛选的节点中选择最近一次接收pong回复距离现在最近的节点；  
 除了常规的选择节点外，对于那些一直未随机到的节点，redis也有所支持。当有节点距离上一次接收到pong消息超过节点超时配置的一半，节点就会给这些节点发送ping消息。ping消息会带上其他节点的信息，选择其他节点步骤是：  
 1、最多选择3个节点；  
 2、最多随机遍历6个节点，如果因为一些条件不能被选出，可能会不满3个；  
 3、忽略自己；  
 4、忽略正在握手的节点；  
 5、忽略带有NOADDR标识的节点；  
 6、忽略连接断开而且没有负责任何slot的节点。  
 ping消息会把发送节点的ping\_sent改成当前时间，直到接收到pong消息，才更新ping\_sent为0。当ping消息发送后，超过节点超时配置的一半，就会把发送节点的连接断开。超过节点超时配置，就会认为该节点已经下线。接收到某个节点发来的ping或者pong消息，节点会更新对接收节点的认识。比如该节点主从角色是否变化，该节点负责的slot是否变化，然后获取消息带上的节点信息，处理新节点的加入。  
 slot更新这里要细说下。假设这里是节点A接收到节点B的消息，A节点会取出保存的B节点的分布表与消息的分布表进行对比，B节点如果是slave，则比较的是A节点保存的B的master的分布表和消息中的分布表。比较只是使用memcmp简单的比较两份数据是否一样，一样则无需处理，不一样就处理不一致的slot。更新的时候，这里只是处理消息中分布表为1的slot。如果和A节点保持一致或者该slot正准备迁移到A节点，则继续处理。如果slot产生了冲突，则以epoch大的为准。如果冲突的slot中有A自己负责的节点，而且B比A的epoch大导致需要更新slot为B负责，此时A负责的slot为0的时候，可以认为B是A的slave。这种情况经常发生在A由原来的master变成slave，B提升为master的场景下。如果B比A的epoch大，则A更新对slot的认识。如果A比B的epoch大， 在redis接下来的逻辑会再次处理，A会给B发送update消息，B收到A发送的update消息，执行slot更新方法。这种情况也经常发生在主从切换的时候。第一种情况发生在新master把数据分布表推给旧master。第二种情况发生在旧master给新master发消息的时候，新master给旧master发送update消息。  
**slot迁移**

redis cluster支持slot的动态迁移，迁移需按照指定步骤进行，不然可能破坏当前的集群数据分布表。cluster setslot <slot> IMPORTING <node ID>命令在迁移目的节点执行，表示需要把该slot迁移到本节点。redis的cluster setslot命令提供了对迁移的支持。cluster setslot <slot> MIGRATING <node ID>命令在迁移源节点执行，表示需要把该slot迁出。cluster setslot <slot> NODE <node ID>在迁移完成后在迁移源节点和迁移目的节点执行后，表示迁移完成，数据分布表恢复稳定。如果需要取消迁移操作，在迁移源节点和迁移目的节点上执行cluster setslot <slot> STABLE。  
下面先来看一下完整的迁移流程：  
 1、在迁移目的节点执行cluster setslot <slot> IMPORTING <node ID>命令，指明需要迁移的slot和迁移源

节点；  
2、在迁移源节点执行cluster setslot <slot> MIGRATING <node ID>命令，指明需要迁移的slot和迁移目的节点；

3、在迁移源节点执行cluster getkeysinslot获取该slot的key列表；

4、在迁移源节点对每个key执行migrate命令，该命令会同步把该key迁移到目的节点；

5、在迁移源节点反复执行cluster getkeysinslot命令，直到该slot的列表为空；

6、在迁移源节点和目的节点执行cluster setslot <slot> NODE <node ID>完成迁移操作。

迁移过程中该slot允许继续有流量进来，redis保证了迁移过程中slot的正常访问。在迁移过程中，对于该slot的请求，如果key在源节点，则表示该key还没有迁移到目的节点。源节点返回ASK错误，告诉客户端去迁移目的节点请求。这样新的key就直接写入到迁移目的节点了。客户端写入目的节点前需要发送ASKING命令，告诉迁移目的节点是写入增量数据，没有ASKING命令，迁移目的节点会不认这次请求，返回MOVED错误，告诉客户端去迁移源节点请求。凭借ASK机制和migrate命令，redis能保证slot的全量数据和增量数据都能导入目的节点。因为对于源节点返回了ASK错误，就能保证该key不在源节点上，那么它只会出现在目的节点或者不存在。所以客户端获取ASK错误到向目的节点无需保证原子性。然后migrate命令是个原子操作，它会等待目的节点写入成功才在源节点返回，保证了迁移期间不接受其他请求，从一个外部客户端的视角来看，在某个时间点上，迁移的键要么存在于源节点，要么存在于目的节点，但不会同时存在于源节点和目的节点。迁移过程中几乎不影响用户使用，除了在多key的命令在迁移目的节点无法执行，这个在集群访问已经说明。不过由于migrate命令是同步阻塞执行的，所以如果key对应的value很大，会增加阻塞时间，特别对于list、set、zset等结构如果value很大的话，redis并不关心这些结构的长度，而是直接以key为单位一次性迁移。同时迁移过程中大量执行migrate命令，会增加客户端的响应时间。迁移的时候在master出现异常的时候，迁移工作需要做些处理。  
 如果在迁移过程中，源节点宕机，此时需做如下调整：目的节点执行cluster setslot <slot> IMPORTING <node ID>命令，node ID为group内的新master源节点，group内的新master执行cluster setslot <slot> MIGRATING <node ID>命令，迁移该slot到目的节点。这样可以继续完成迁移操作。  
 当槽x从Node A向Node B迁移时，Node A和Node B都会有这个槽x，Node A上槽x的状态设置为MIGRATING，Node B上槽x的状态被设置为IMPORTING。

**MIGRATING状态**

如果key存在则成功处理，如果key不存在，则返回客户端ASK，客户端根据ASK首先发送ASKING命令到目标节点，然后发送请求的命令到目标节点。当key包含多个命令，如果都存在则成功处理；如果都不存在则返回客户端ASK；如果一部分存在，则返回客户端TRYAGAIN，通知客户端稍后重试，这样当所有的key都迁移完的时候客户端重试请求的时候回得到ASK，然后经过一次重定向就可以获取这批键，此时不刷新客户端中node的映射关系。

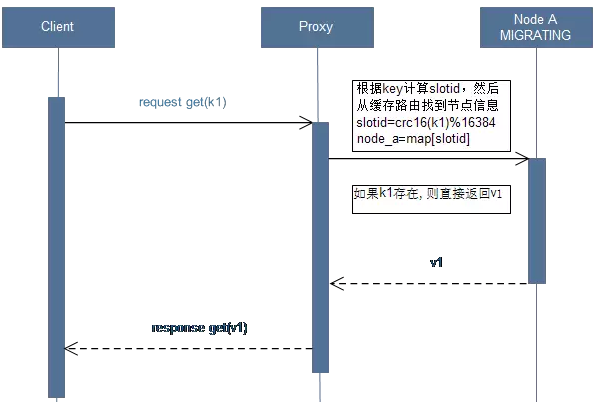
**IMPORTING状态**

如果key不在该节点上，会被MOVED重定向，刷新客户端中node的映射关系；

如果是ASKING命令则会被执行，key不在迁移的源节点就在已经被迁移到目标节点；

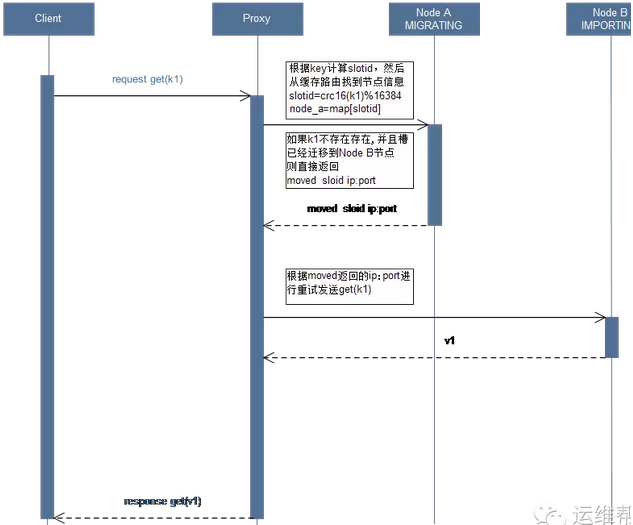
Key不存在则新建。

### ****读写请求****

槽里面key还未迁移，并且槽属于迁移中，假如k1属于槽x，并且k1还在Node A。  


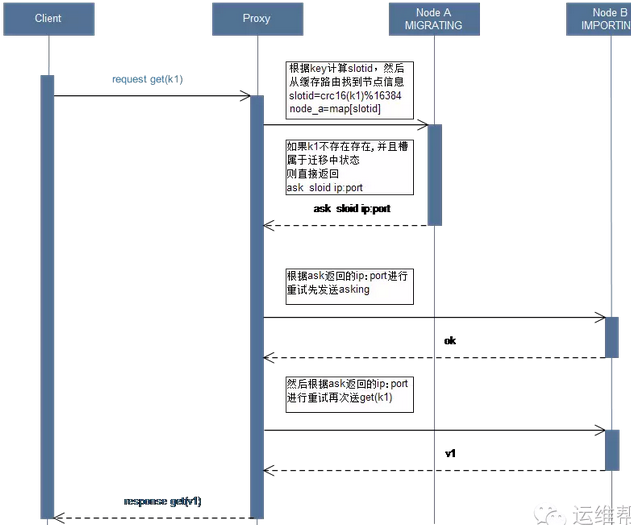
**MOVED请求**

槽里面的key已经迁移过去，并且槽属于迁移完，假如k1属于槽x，并且k1不在Node A，而且槽x已经迁移到Node B。



### ****ASK请求****

槽里面的key已经迁移完，并且槽属于迁移中，假如k1属于槽x，并且k1不在Node A，而且槽x还是MIGRATING状态。



**主从复制**

集群间节点支持主从关系，复制的逻辑基本复用了单机版的实现。不过还是有些地方需要注意。首先集群间节点建立主从关系不再使用原有的SLAVEOF命令和SLAVEOF配置，而是通过cluster replicate命令，这保证了主从节点需要先完成握手，才能建立主从关系。集群是不能组成链式主从关系的，也就是说从节点不能有自己的从节点。不过对于集群外的没开启集群功能的节点，redis并不干预这些节点去复制集群内的节点，但是在集群故障转移时，这些集群外的节点，集群不会处理。集群内节点想要复制另一个节点，需要保证本节点不再负责任何slot，不然redis是不允许的。集群内的从节点在与其他节点通信的时候，传递的消息中数据分布表和epoch是master的值。  
**故障转移**

集群主节点出现故障，发生故障转移时，其他主节点会把故障主节点的从节点自动提为主节点，原来的主节点恢复后，自动成为新主节点的从节点。这里先说明，把一个master和它的全部slave描述为一个group，故障转移是以group为单位的，集群故障转移的方式跟sentinel的实现很类似。某个节点一段时间没收到心跳响应，则集群内的master会把该节点标记为pfail，类似sentinel的sdown。集群间的节点会交换相互的认识，超过一半master认为该异常master宕机，则这些master把异常master标记为fail，类似sentinel的odown，fail消息会被master广播出来。group的slave收到fail消息后开始竞选成为master。竞选的方式跟sentinel选主的方式类似，都是使用了raft协议，slave会从其他的master拉取选票，票数最多的slave被选为新的master，新master会马上给集群内的其他节点发送pong消息，告知自己角色的提升。其他slave接着开始复制新master。等旧master上线后，发现新master的epoch高于自己，通过gossip消息交互，把自己变成了slave。自动故障转移的方式跟sentinel很像，redis还支持手动的故障转移，即通过在slave上执行cluster failover命令，可以让slave提升为master。failover命令支持传入FORCE和TAKEOVER参数。  
 FORCE：使用FORCE参数与sentinel的手动故障转移流程基本类似，强制开始一次故障转移。不传入额外参数。如果主节点异常，则不能进行failover，主节点正常的情况下需要先比较从节点和主节点的偏移量，此时会让主节点停止客户端请求，直到超时或者故障转移完成。主从偏移量相同后开始手动故障转移流程。  
 TAKEOVER：这种手动故障转移的方式比较暴力，slave直接提升自己的epoch为最大的epoch，并把自己变成master。这样在消息交互过程中，旧master能发现自己的epoch小于该slave，同时两者负责的slot一致，它会把自己降级为slave。  
**网络分区说明**

redis的集群模式下，客户端需要和全部的节点保持连接，这样可能出现网络分区问题，客户端和一些节点在一个网络分区，另一部分节点在另一个网络分区。在分区期间，客户端仍然能执行命令，直到集群经过cluster-node-timeout发现分区情况，节点探测到有slot无法提供服务，才开始禁止客户端执行命令。这时候会出现一种现象，假设客户端和一个master在小分区，其他节点在大分区。超时后，其他节点共同投票把group内的一个slave提为master，等分区恢复。旧的master会成为新master的slave。这样在cluster-node-timeout期间对旧master的写入数据都会丢失。这个问题可以通过设置cluster-node-timeout来减少不一致。如果对一致性要求高的应用还可以通过min-slaves-to-write配置来提高写入的要求。

处于“小分区”的集群节点是不可用的，“大分区”端必须持有大多数Masters，同时每个不可达的Master至少有一个slave也在“大分区”端，当NODE\_TIMEOUT时，触发failover，此后集群才是可用的。Redis Cluster在小部分nodes失效后仍然可以恢复有效性，如果application希望大面积节点失效仍然有效，那么Cluster不适合这种情况。

  比如集群有N个Master，且每个Master都有一个slave，那么集群的有效性只能容忍一个节点（master）被分区隔离（即一个master处于小分区端，其他处于大分区端），当第二个节点被分区隔离之前仍保持可用性的概率为1 - (1/(N \* 2 - 1))（解释：当第一个节点失效后，剩余N \* 2 -1个节点，此时没有slave的Master失效的概率为1/(N \* 2 -1)）。比如有5个Master，每个Master有一个slave，当2个nodes被隔离出去（或者失效）后，集群可用性的概率只有1/(5 \* 2 - 1) = 11.11%，因此集群不再可用。幸好Redis Cluster提供了“replicas migration”机制，在实际应用方面，可以有效的提高集群的可用性，当每次failover发生后，集群都会重新配置、平衡slaves的分布，以更好的抵御下一次失效情况的发生。

**slot保存key列表**

redis提供了cluster countkeysinslot和cluster getkeysinslot命令，可以获得某个slot的全部key列表。通过该列表，可以实现slot的迁移，该功能是通过skiplist实现的。skiplist是redis内部用来实现zset的数据结构，在slot保存key的时候也派上了用场。redis所有在db层对hash表的操作，也会在skiplist上执行相应的操作。比如往hash表增加数据，redis也会往skiplist也写一份数据，该skiplist的score就是slot的值，value对应了具体的key。这等于是redis在数据分布表上冗余了所有的key。不过相比skiplist所带来迁移的方便，冗余的结果是可以接受的，这也期望客户端，不要使用过长的key，从而增加内存的消耗。

**keys分布模型**

    集群将key分成16384个slots（hash槽），slot是数据映射的单位，言外之意，Redis Cluster最多支持16384个nodes（每个nodes持有一个slot）。集群中的每个master持有16384个slots中的一部分，处于“stable”状态时，集群中没有任何slots在节点间迁移，即任意一个hash slot只会被单个node所服务（master，当然可以有多个slave用于replicas，slave也可以用来扩展read请求）。keys与slot的映射关系，是按照如下算法计算的：HASH\_SLOT = CRC16(key) mod 16384。其中CRC16是一种冗余码校验和，可以将字符串转换成16位的数字。

## 集群相关命令

### cluster meet ip:port  #集群间相互握手，加入彼此所在的集群（将指定节点加入到集群）。

**cluster nodes #**获取集群内节点信息的列表，格式为<node ID> <node IP:PORT> <node role> [master node ID|-] <node ping\_sent> <node pong\_received> <node epoch> <node status>。

### cluster myid #返回节点的id。 127.0.0.1:6379> cluster myid "9b35a393fa6623887215023b761d531dde452d3c"

### cluster slots #返回集群间各节点负责的数据分布表。 127.0.0.1:6379> cluster slots 1) 1) (integer) 10923    2) (integer) 16383    3) 1) "10.180.157.201"       2) (integer) 6379    4) 1) "10.180.157.208"       2) (integer) 6379 2) 1) (integer) 5461    2) (integer) 10922    3) 1) "10.180.157.200"       2) (integer) 6379    4) 1) "10.180.157.205"       2) (integer) 6379 3) 1) (integer) 0    2) (integer) 5460    3) 1) "10.180.157.199"       2) (integer) 6379    4) 1) "10.180.157.202"       2) (integer) 6379

### cluster flushslots #清空该节点负责slots，必须在节点负责的这些slot都没有数据的情况下才能执行，该命令需要谨慎使用，由于之前说的bitmapTestBit方法，redis只比较负责的节点，清空的slots信息无法被其他节点同步。

### cluster addslots [slot] #在当前节点上增加slot（将指定的一个或多个slot 指派给当前节点）。

### cluster delslots [slot] #在节点上取消slot的负责。这也会导致前面说的slot信息无法同步，而且一旦集群有slot不负责，配置cluster-require-full-coverage为yes的话，该节点就无法提供服务了，所以使用也需谨慎。

### cluster setslot <slot> MIGRATING <nodeid> #把本节点负责的某个slot设置为迁移到目的节点(即将本节点的slot指派给或叫迁移到指定的节点)。

### cluster setslot <slot> IMPORTING <nodeid> #设置某个slot为从迁移源节点迁移标志(即将指定节点的slot指派给或叫迁移到本节点)。

### cluster setslot <slot> STABLE #设置某个slot为从迁移状态恢复为正常状态(取消slot的导入（importing）或迁移（migrating）)。

### cluster setslot <slot> NODE <nodeid> #设置某个slot为某节点负责。该命令使用也需要注意，cluster setslot的四个命令需要配置迁移工具使用，单独使用容易引起集群混乱。该命令在集群出现异常时，需要指定某个slot为某个节点负责时，最好在每个节点上都执行一遍，至少要在迁移的节点和最高epoch的节点上执行成功(将指定的slot指派给指定的节点，如果该slot已经指派给另一个节点，则要另一个节点先删除该slot)。

### cluster info #集群的一些info信息。

### cluster saveconfig #保存集群的配置文件，集群默认在配置修改的时候会自动保存配置文件，该方法也能手动执行命令保存。

### cluster keyslot <key> #可以查询某个key对应的slot地址。 127.0.0.1:6379> cluster keyslot key (integer) 12539

### cluster countkeysinslot <slot-id> #可以查询该节点负责的某个slot内部key的数量。 127.0.0.1:6379> cluster countkeysinslot 13252 (integer) 2

### cluster getkeysinslot <slot> <count> #可以查询该节点负责的某个slot内部指定数量的key列表。 127.0.0.1:6379> cluster getkeysinslot 13252 10 1) "key0" 2) "key2298"

### cluster forget #把某个节点加入黑名单，这样就无法完成握手。黑名单的过期时为60s，60s后两节点又会继续完成握手。

### cluster replicate <nodeid> #负责某个节点，成为它的slave（将当前节点设置为指定nodeid节点的从节点）。

### cluster slaves #列出某个节点slave列表。 127.0.0.1:6379> cluster slaves 2b5603326d0fca28031467727fae4558115a99d8 1) "a15705fdb7cac60e07ff699bf4c514e80f245a2c 10.180.157.205:6379 slave 2b5603326d0fca28031467727fae4558115a99d8 0 1450854932667 11 connected"

### cluster count-failure-reports #列出某个节点的故障转移记录的长度。

### cluster failover [FORCE|TAKEOVER] #手动执行故障转移。 cluster set-config-epoch #设置节点epoch，只有在节点加入集群前才能设置。 cluster reset [SOFT|HARD] #重置集群信息，soft是清空其他节点的信息，但不修改自己的id。hard还会修改自己的id。不传该参数则使用soft方式。

### readonly #在slave上执行，执行该命令后，可以在slave上执行只读命令。

### readwrite #在slave上执行，执行该命令后，取消在slave上执行写命令。

## 集群相关配置

### cluster-enabled

说明：集群开关，默认是不开启集群模式。  
默认值:no。  
是否可以动态修改:no。  
值的范围:yes|no。

### cluster-config-file

说明：集群配置文件的名称，每个节点都有一个集群相关的配置文件，持久化保存集群的信息。  
默认值:”nodes.conf”。  
是否可以动态修改:no。  
值的范围:文件路径。  
**cluster-node-timeout**

说明：节点的超时时间，单位是毫秒。  
默认值:15000。  
是否可以动态修改:yes。  
值的范围:大于0。

### cluster-slave-validity-factor

说明：在进行故障转移的时候，group的全部slave都会请求申请为master，但是有些slave可能与master断开连接一段时间了，导致数据过于陈旧，这样的slave不应该被提升为master。该参数就是用来判断slave节点与master断线的时间是否过长。判断方法是比较slave断开连接的时间和(node-timeout \* slave-validity-factor) + repl-ping-slave-period。  
默认值:10。  
是否可以动态修改:yes。  
值的范围:大于等于0。

### cluster-migration-barrier

说明：master的slave数量大于该值，slave才能迁移到其他孤儿master上。  
默认值:1。  
是否可以动态修改:yes。  
值的范围:大于等于0。

### cluster-require-full-coverage

说明：默认情况下，集群全部的slot有节点负责，集群状态才为ok，才能提供服务。设置为no，可以在slot没有全部分配的时候提供服务。不建议打开该配置，这样会造成分区的时候，小分区的master一直在接受写请求，而造成很长时间数据不一致。  
默认值:yes。  
是否可以动态修改:yes。  
值的范围:yes|no。

# 应用案例收集

## 有道：Redis Cluster使用经验

“redis cluster的设计在这块有点奇葩，跟集群相关的操作需要一个外部的ruby脚本来协助（当然可能是为了让主程序的代码足够简洁？），然后那个脚本还只支持填实例的ip不支持host，还不告诉你不支持让你用host之后各种莫名其妙。”“第一个缺点就是严格依赖客户端driver的成熟度。如果把redis cluster设计成类似Cassandra，请求集群中任何一个节点都可以负责转发请求，client会好写一些。”

“第二个缺点完全是设计问题了，就是一个redis进程既负责读写数据又负责集群交互，虽然设计者已经尽可能简化了代码和逻辑，但还是让redis从一个内存NoSQL变成了一个分布式NoSQL。分布式系统很容易有坑，一旦有坑必须升级redis。”

### 去中心化 vs. Proxy

“关于redis cluster的设计，Gossip/P2P的去中心化架构本身不是问题，但一旦有了中心节点，能做的事情就多了，比如sharding不均匀是很容易自动rebalance的，而无中心的只能靠外界来搞。然后redis cluster又是slot的形式而非C\*式的一致性哈希，新节点分slot又不自动，依赖外界（ruby脚本）来分配显得不方便更不优美和谐。而且因为是master-slave的系统而非W+R>N的那种，master挂掉之后尽快发现是比较重要的，gossip对于节点挂掉的发现终究没有中心节点/zookeeper方便快速。基于proxy做转发意味着屏蔽了下层存储，完全可以根据前缀/tag/冷热程度，来把部分甚至大多数数据放在磁盘从而节约成本又保证一致性，这都是有中心节点所带来的好处。

**奇虎360：Redis Cluster浅析和Bada对比**

### 负载均衡问题

“redis cluster的主备是以节点为单位，而bada则是以partition为单位，这样同样是3个节点，1024个partition的情况下，redis cluster的主节点负责整个1024个partition的服务，**而两个从节点则只负责异步备份，导致集群负载不均，**再看bada，将1024个partition的主均分到3个节点中，每个节点各有主备，主对外提供服务，这样均分了访问压力，有效的利用了资源。”

### 一致性的保证

“**redis cluster与bada一样，最终一致性**，读写都只请求主节点，当一条写请求在对应的主节点写成功后，会立刻返回给客户端成功，然后主节点通过异步的方式将新的数据同步到对应的从节点，这样的方式减少了客户端多个节点写成功等待的时间，不过在某些情况下会造成写丢失：

1）当主节点接受一条写请求，写入并返回给客户端成功后不幸宕掉，此时刚才的写还未同步给其对应的从节点，而从节点在发现主节点挂掉并重新选主后，新的主节点则永久丢失了之前老的主节点向用户确认的写。

2）当网络发生割裂，将集群分裂成少数派与多数派，这样在客户端不知情的情况下，会将写继续写入到少数派中的某些主节点中，而当割裂超过一定时长后，集群感知到异常，此时少数派中的所有主节点会停止响应所有的写请求，多数派其对应的从节点则会发起选举成为新的主节点，假设过了一会后割裂恢复，老的主节点发现有更新的主存在，自动变成其从节点，而新的主节点中则会永久丢失掉网络割裂至集群感知异常进行切主这个阶段老主节点确认的所有写。

**相对于redis cluster的永久丢失，bada通过binlog merge有效的解决了这一问题**。所有partition的主节点在响应客户端的写请求时，都会在本地记录binlog，binlog实质就是带有时间戳的KV对。当老主以从节点的身份重新加入集群时，会触发binlog merge操作，新主会比较并且合并二者的binlog，这样就可以将之前丢失掉得写再补回来。”

**请求重定向问题**

“bada服务端节点在收到本不该由自己负责的Partition请求后，不会向客户端返回重定向信息，而是通过代理的方式，直接在集群内部向正确节点转发客户端的请求，并将结果同meta信息再转发回客户端。”“再看multi key操作，**redis cluster为了追求高性能，支持multi key的前提是所有的key必须在同一个节点中**, 不过这样的处理需要交给用户，对需要进行multi key操作的所有key，在写入前人为的加上hash tags。当redis cluster进行resharding的时候，也就是将某些slot从一个节点迁移到另一个节点时，此时的multi key操作可能会失败，因为在迁移的slot中的key此时存在于两个节点。bada怎么做呢？用户如果对multi key操作性能很在乎时，可以采用与redis cluster同样的方式，给这些key加上hash tags来让它们落在同一个节点，如果可以接受性能的稍微损耗而解放用户的处理逻辑，则可以像single key操作一样，**请求任一bada节点，它会代理所有的key请求并将结果返回给用户，并且multi key操作在任何时候都可以，即使在进行partition的迁移，**bada也会提前进行切主，保证服务的正常提供。”

## 芒果TV：Redis服务解决方案

芒果TV在Redis Cluster基础上进行开发，主要增加了两个组件：

**监控管理**：以Python为主要开发框架的Web应用程序[Redis-ctl](https://github.com/HunanTV/redis-ctl)。

**请求代理**：以C++11为开发语言的轻量数据代理程序[cerberus](https://github.com/HunanTV/redis-cerberus)。

**其作用和优点为：**   
集群代理程序的自动请求分发/重试机制使得应用不必修改自身代码或更新Redis库，代理节点为所有Redis节点加上统一管理和状态监测, 可以查阅历史数据, 或在发生任何问题之后快速响应修复，代理进程的无状态性使之可在故障后快速恢复, 不影响后端集群数据完整性。

关于Redis Cluster带来的种种优势就不说了，在这里主要是“鸡蛋里挑骨头”，总结一下现阶段集群功能的欠缺之处和可能的“坑”。

## 无中心化架构

### Gossip消息

Gossip消息的网络开销和延时是决定Redis Cluster能够线性扩展的因素之一。

### 节点粒度备份

Redis Cluster也许是为了简化设计采用了Master-Slave复制的数据备份方案，并没有采取如Cassandra或IMDG等对等分布式系统中常见的Slot粒度（或叫Partition/Bucket等）的自动冗余和指派。这种设计虽然避免比较复杂的分布式技术，但也带来了一些问题：

**Slave完全闲置**：即便是读请求也不会被重定向到Slave节点上，Slave属于“冷备”；

**写压力无法分摊**：Slave闲置导致的另一个问题就是写压力也都在Master上。

**客户端的挑战**

由于Redis Cluster的设计，客户端要担负起一部分责任：

**Cluster协议支持**：不管Dummy还是Smart模式，都要具备解析Cluster协议的能力；

**网络开销**：Dummy客户端不断重定向的网络开销；

**连接维护**：Smart客户端对连接到集群中每个节点Socket的维护；

**缓存路由表**：Smart客户端Slot路由表的缓存和更新；

**内存消耗**：Smart客户端上述维护的信息都是有内存消耗的；

**MultiOp有限支持**：对于MultiOp，由客户端通过KeyTag保证所有Key都在同一Slot。而即便如此，迁移时也会导致MultiOp失败。同理，对Pipeline和Transaction的支持也受限于必须操作同一Slot内的Key。

## Redis实现问题

尽管属于无中心化架构一类的分布式系统，但不同产品的细节实现和代码质量还是有不少差异的，就比如Redis Cluster有些地方的设计看起来就有一些“奇葩”和简陋：

**不能自动发现**：无Auto Discovery功能。集群建立时以及运行中新增结点时，都要通过手动执行MEET命令或redis-trib.rb脚本添加到集群中。

**不能自动Resharding**：不仅不自动，连Resharding算法都没有，要自己计算从哪些结点上迁移多少Slot，然后还是得通过redis-trib.rb操作。

**严重依赖外部redis-trib**：如上所述，像集群健康状况检查、结点加入、Resharding等等功能全都抽离到一个Ruby脚本中了。还不清楚上面提到的缺失功能未来是要继续加到这个脚本里还是会集成到集群结点中？redis-trib也许要变成Codis中Dashboard的角色。

**无监控管理UI**：即便未来加了UI，像迁移进度这种信息在无中心化设计中很难得到。

**只保证最终一致性**：写Master成功后立即返回，**如需强一致性，自行通过WAIT命令实现**。但对于“脑裂”问题，目前Redis没提供网络恢复后的Merge功能，“脑裂”期间的更新可能丢失。

## 性能损耗：由于之前手头没有空闲的物理机资源，所以只在虚拟机上做了简单的单机测试，在单独的一台压力机使用YCSB测试框架向虚拟机产生读写负载。虚拟机的配置为8核Intel Xeon CPU X5650@2.67GHz，16GB内存，分别搭建了4结点的单机版Redis和集群版Redis，测试一下Redis Cluster的性能损耗。由于不是最近做的测试，所以Jedis用的2.6.2版本。注：当然Redis Cluster可以通过多机部署获得水平扩展带来的性能提升，这里只是由于环境有限所以做的简单单机测试。由于YCSB本身仅支持Redis单机版，所以需要我们自己增加扩展插件，具体方法请参照[《YCSB性能测试工具使用》](http://blog.csdn.net/dc_726/article/details/43991871)。通过YCSB产生2000w随机数据，Value大约100Byte左右。然后通过YCSB测试Read-Mostly(90% Read)和Read-Write-Mixed(50% Read)两种情况：

**数据加载**：吞吐量上有约18%的下降。

**Read-Mostly**：吞吐量上有约3.5%~7.9%的下降。

**Read-Write-Mixed**：吞吐量上有约3.3%~5.5%下降。

**内存占用**：Jedis客户端多占用380MB内存。

## 最后的总结

现阶段看来，相比Sentinel或Codis等方案，Redis Cluster的优势还真是有限，最大的优点有两个：

1、官方提供的Slot实现而不用像Codis那样去改源码了；

2、不用额外的Sentinel集群或类似的代码实现了。

同其他分布式系统，如Cassandra，或内存型的IMDG，如Hazelcast和GridGain，除了性能方面外，从功能上Redis Cluster简直被爆得体无完肤… 看GridGain：

1、节点自动发现和Rebalance；

2、分区粒度的备份；

3、故障时分区角色自动调整；

4、结果聚合（不会重定向客户端）；

5、“脑裂”恢复后的Merge（Hazelcast支持多种合并策略）；

6、多Primary分区写操作（见Replicated模式）。

**这些都是Redis Cluster没有或者要手动完成的。当然这也不足为奇，因为这与Redis的设计初衷有关**，毕竟作者都已经说了，最核心的设计目标就是性能、水平伸缩和可用性。