分布式缓存 Redis 使用方法

作者：张小博，新炬网络技术专家。

缓存在系统中的作用：

1、少量数据存储，高速读写访问。通过数据全部 in-momery 的方式来保证高速访问，同时提供数据落地的功能，实际这正是 Redis 最主要的适用场景。

2、海量数据存储，分布式系统支持，数据一致性保证，方便的集群节点添加/删除。Redis3.0 以后开始支持集群，实现了半自动化的数据分片，不过需要 smart-client 的支持。

Redis 全角度介绍:

网络模型：Redis 使用单线程的 IO 复用模型，自己封装了一个简单的 AeEvent 事件处理框架，主要实现了 epoll、kqueue 和 select，对于单纯只有 IO 操作来说，单线程可以将速度优势发挥到最大，但是 Redis 也提供了一些简单的计算功能，比如排序、聚合等，对于这些操作，单线程模型实际会严重影响整体吞吐量，CPU 计算过程中，整个 IO 调度都是被阻塞住的。

内存管理：Redis 使用现场申请内存的方式来存储数据，并且很少使用 free-list 等方式来优化内存分配，会在一定程度上存在内存碎片，Redis 跟据存储命令参数，会把带过期时间的数据单独存放在一起，并把它们称为临时数据，非临时数据是永远不会被剔除的，即便物理内存不够，导致 swap 也不会剔除任何非临时数据（但会尝试剔除部分临时数据），这点上 Redis 更适合作为存储而不是 cache。

数据一致性问题：在一致性问题上，个人感觉 redis 没有 memcached 实现的好，Memcached 提供了 cas 命令，可以保证多个并发访问操作同一份数据的一致性问题。Redis 没有提供 cas 命令，并不能保证这点，不过 Redis 提供了事务的功能，可以保证一串 命令的原子性，中间不会被任何操作打断。

支持的 KEY 类型：Redis 除 key/value 之外，还支持 list,set,sorted set,hash 等众多数据结构，提供了 KEYS 进行枚举操作，但不能在线上使用，如果需要枚举线上数据，Redis 提供了工具可以直接扫描其 dump 文件，枚举出所有数据，Redis 还同时提供

了持久化和复制等功能。

客户端支持：redis 官方提供了丰富的客户端支持，包括了绝大多数编程语言的客户端,比如我此次测试就选择了官方推荐了 Java 客户端 Jedis.里面提供了丰富的接口、方法使得开发人员无需关系内部的数据分片、读取数据的路由等,只需简单的调用即可,非常方便。

数据复制：从 2.8 开始，Slave 会周期性（每秒一次）发起一个 Ack 确认复制流（replication stream）被处理进度， Redis 复制工作原理详细过程如下：

1. 如果设置了一个 Slave，无论是第一次连接还是重连到 Master，它都会发出一个 SYNC 命令；

2. 当 Master 收到 SYNC 命令之后，会做两件事：

a) Master 执行 BGSAVE：后台写数据到磁盘（rdb 快照）；

b) Master 同时将新收到的写入和修改数据集的命令存入缓冲区（非查询类）；

3. 当 Master 在后台把数据保存到快照文件完成之后，Master 会把这个快照文件传送给 Slave，而 Slave 则把内存清空后，加载该文件到内存中；

4. 而 Master 也会把此前收集到缓冲区中的命令，通过 Reids 命令协议形式转发给 Slave，Slave 执行这些命令，实现和 Master 的同步；

5. Master/Slave 此后会不断通过异步方式进行命令的同步，达到最终数据的同步一致；

6. 需要注意的是 Master 和 Slave 之间一旦发生重连都会引发全量同步操作。但在 2.8 之后，也可能是部分同步操作。

2.8 开始，当 Master 和 Slave 之间的连接断开之后，他们之间可以采用持续复制处理方式代替采用全量同步。

Master 端为复制流维护一个内存缓冲区（in-memory backlog），记录最近发送的复制流命令；同时，Master 和 Slave 之间都维护一个复制偏移量(replication offset)和当前 Master 服务器 ID（Master run id）。当网络断开，Slave 尝试重连时：

a. 如果 MasterID 相同（即仍是断网前的 Master 服务器），并且从断开时到当前时

刻的历史命令依然在 Master 的内存缓冲区中存在，则 Master 会将缺失的这段时间的所有命令发送给 Slave 执行，然后复制工作就可以继续执行了；

b. 否则，依然需要全量复制操作。

读写分离：redis 支持读写分离，而且使用简单，只需在配置文件中把 redis 读服务器和写服务器进行配置，多个服务器使用逗号分开如下：

<RedisConfig

WriteServerList="192.168.2.71:6379"

ReadServerList="192.168.2.71:6379,192.168.2.71:6380"

MaxWritePoolSize="60"

MaxReadPoolSize="60"

AutoStart="true"

LocalCacheTime="180"

RecordeLog="false">

</RedisConfig>

水平动态扩展：历时三年之久，终于等来了期待已由的 Redis 3.0。新版本主要是实现了 Cluster 的功能，增删集群节点后会自动的进行数据迁移。实现 Redis 集群在线重配置的核心就是将槽从一个节点移动到另一个节点的能力。 因为一个哈希槽实际上就是一些键的集合， 所以 Redis 集群在重哈希（rehash）时真正要做的， 就是将一些键从一个节点移动到另一个节点。

数据淘汰策略：redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。redis 提供 6 种数据淘汰策略：

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-enviction（驱逐）：禁止驱逐数据

集群（分布式）

下面详细介绍一下 redis 的集群功能，从 3.0 以后的版本开始支持集群功能，也就是正真意义上实现了分布式。

Redis 集群是一个分布式（distributed）、容错（fault-tolerant）的 Redis 实现， 集群可以使用的功能是普通单机 Redis 所能使用的功能的一个子集（subset）。

Redis 集群中不存在中心（central）节点或者代理（proxy）节点， 集群的其中一个主要设计目标是达到线性可扩展性（linear scalability）。

Redis 集群为了保证一致性（consistency）而牺牲了一部分容错性： 系统会在保证对网络断线（net split）和节点失效（node failure）具有有限（limited）抵抗力的前提下， 尽可能地保持数据的一致性。

集群特性：

(1)所有的 redis 节点彼此互联(PING-PONG 机制),内部使用二进制协议优化传输速度和带宽.

(2)节点的 fail 是通过集群中超过半数的节点检测失效时才生效.

(3)客户端与 redis 节点直连,不需要中间 proxy 层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

(4)redis-cluster 把所有的物理节点映射到[0-16383]slot 上,cluster 负责维护 node<->slot<->value

Redis 集群实现的功能子集：

Redis 集群实现了单机 Redis 中， 所有处理单个数据库键的命令。针对多个数据库键的复杂计算操作， 比如集合的并集操作、合集操作没有被实现， 那些理论上需要使用多个节点的多个数据库键才能完成的命令也没有被实现。在将来， 用户也许可以通过 MIGRATE COPY 命令， 在集群的计算节点（computation node）中执行针对多个数据库键的只读操作， 但集群本身不会去实现那些需要将多个数据库键在多个节点中移来移去的复杂多键命令。

Redis 集群不像单机 Redis 那样支持多数据库功能， 集群只使用默认的 0 号数据库， 并且不能使用 SELECT 命令。

Redis 集群协议中的客户端和服务器：

Redis 集群中的节点有以下责任：

1、持有键值对数据。

2、记录集群的状态，包括键到正确节点的映射（mapping keys to right nodes）。

3、自动发现其他节点，识别工作不正常的节点，并在有需要时，在从节点中选举出新的主节点。

为了执行以上列出的任务， 集群中的每个节点都与其他节点建立起了“集群连接（cluster bus）”， 该连接是一个 TCP 连接， 使用二进制协议进行通讯。

节点之间使用 Gossip 协议 来进行以下工作：

1、传播（propagate）关于集群的信息，以此来发现新的节点。

2、向其他节点发送 PING 数据包，以此来检查目标节点是否正常运作。

3、在特定事件发生时，发送集群信息。

4、除此之外， 集群连接还用于在集群中发布或订阅信息。

因为集群节点不能代理（proxy）命令请求， 所以客户端应该在节点返回 -MOVED 或者 -ASK 转向（redirection）错误时， 自行将命令请求转发至其他节点。因为客户端可以自由地向集群中的任何一个节点发送命令请求， 并可以在有需要时， 根据转向错误所提供的信息， 将命令转发至正确的节点， 所以在理论上来说， 客户端是无须保存集群状态信息的。不过， 如果客户端可以将键和节点之间的映射信息保存起来，

可以有效地减少可能出现的转向次数， 籍此提升命令执行的效率。

键分布模型：

Redis 集群的键空间被分割为 16384 个槽（slot）， 集群的最大节点数量也是 16384 个。

推荐的最大节点数量为 1000 个左右。每个主节点都负责处理 16384 个哈希槽的其中一部分。

当我们说一个集群处于“稳定”（stable）状态时， 指的是集群没有在执行重配（reconfiguration）操作， 每个哈希槽都只由一个节点进行处理。重配置指的是将某个/某些槽从一个节点移动到另一个节点。一个主节点可以有任意多个从节点， 这些从节点用于在主节点发生网络断线或者节点失效时， 对主节点进行替换。

集群节点属性：

每个节点在集群中都有一个独一无二的 ID ， 该 ID 是一个十六进制表示的 160 位随机数， 在节点第一次启动时由 /dev/urandom 生成。

节点会将它的 ID 保存到配置文件， 只要这个配置文件不被删除， 节点就会一直沿用这个 ID 。节点 ID 用于标识集群中的每个节点。 一个节点可以改变它的 IP 和端口号， 而不改变节点 ID 。 集群可以自动识别出 IP/端口号的变化， 并将这一信息通过 Gossip 协议广播给其他节点知道。

以下是每个节点都有的关联信息， 并且节点会将这些信息发送给其他节点：

1、节点所使用的 IP 地址和 TCP 端口号。

2、节点的标志（flags）。

3、节点负责处理的哈希槽。

4、节点最近一次使用集群连接发送 PING 数据包（packet）的时间。

5、节点最近一次在回复中接收到 PONG 数据包的时间。

6、集群将该节点标记为下线的时间。

7、该节点的从节点数量。

8、如果该节点是从节点的话，那么它会记录主节点的节点 ID 。 如果这是一个主节点的话，那么主节点 ID 这一栏的值为 0000000 。

以上信息的其中一部分可以通过向集群中的任意节点（主节点或者从节点都可以）发送 CLUSTER NODES 命令来获得。

节点握手：

节点总是应答（accept）来自集群连接端口的连接请求， 并对接收到的 PING 数据包进行回复， 即使这个 PING 数据包来自不可信的节点。然而， 除了 PING 之外，节点会拒绝其他所有并非来自集群节点的数据包。要让一个节点承认另一个节点同属于一个集群， 只有以下两种方法：

1、一个节点可以通过向另一个节点发送 MEET 信息， 来强制让接收信息的节点承认发送信息的节点为集群中的一份子。 一个节点仅在管理员显式地向它发送 CLUSTER MEET ip port 命令时， 才会向另一个节点发送 MEET 信息。

2、如果一个可信节点向另一个节点传播第三者节点的信息， 那么接收信息的那个节点也会将第三者节点识别为集群中的一份子。 也即是说， 如果 A 认识 B ， B 认识 C ， 并且 B 向 A 传播关于 C 的信息， 那么 A 也会将 C 识别为集群中的一份子， 并尝试连接 C 。

这意味着如果我们将一个/一些新节点添加到一个集群中， 那么这个/这些新节点最终会和集群中已有的其他所有节点连接起来。

这说明只要管理员使用 CLUSTER MEET 命令显式地指定了可信关系， 集群就可以自动发现其他节点。这种节点识别机制通过防止不同的 Redis 集群因为 IP 地址变更或者其他网络事件的发生而产生意料之外的联合（mix）， 从而使得集群更具健壮性。当节点的网络连接断开时， 它会主动连接其他已知的节点。

MOVED 转向：

一个 Redis 客户端可以向集群中的任意节点（包括从节点）发送命令请求。 节点会对命令请求进行分析， 如果该命令是集群可以执行的命令， 那么节点会查找这个命令所要处理的键所在的槽。如果要查找的哈希槽正好就由接收到命令的节点负责处理，那么节点就直接执行这个命令。另一方面， 如果所查找的槽不是由该节点处理的话，节点将查看自身内部所保存的哈希槽到节点 ID 的映射记录， 并向客户端回复一个

MOVED 错误。

即使客户端在重新发送 GET 命令之前， 等待了非常久的时间， 以至于集群又再次更改了配置， 使得节点 127.0.0.1:6381 已经不再处理槽 3999 ， 那么当客户端向节点 127.0.0.1:6381 发送 GET 命令的时候， 节点将再次向客户端返回 MOVED 错误，指示现在负责处理槽 3999 的节点。

虽然我们用 ID 来标识集群中的节点， 但是为了让客户端的转向操作尽可能地简单， 节点在 MOVED 错误中直接返回目标节点的 IP 和端口号， 而不是目标节点的 ID 。但一个客户端应该记录（memorize）下“槽 3999 由节点 127.0.0.1:6381 负责处理“这一信息， 这样当再次有命令需要对槽 3999 执行时， 客户端就可以加快寻找正确节点的速度。

注意， 当集群处于稳定状态时， 所有客户端最终都会保存有一个哈希槽至节点的映射记录（map of hash slots to nodes）， 使得集群非常高效： 客户端可以直接向正确的节点发送命令请求， 无须转向、代理或者其他任何可能发生单点故障（single point failure）的实体（entiy）。

除了 MOVED 转向错误之外， 一个客户端还应该可以处理稍后介绍的 ASK 转向错误。

集群在线重配置：

Redis 集群支持在集群运行的过程中添加或者移除节点。

实际上， 节点的添加操作和节点的删除操作可以抽象成同一个操作， 那就是， 将哈希槽从一个节点移动到另一个节点：

添加一个新节点到集群， 等于将其他已存在节点的槽移动到一个空白的新节点里面。

从集群中移除一个节点， 等于将被移除节点的所有槽移动到集群的其他节点上面去。

因此， 实现 Redis 集群在线重配置的核心就是将槽从一个节点移动到另一个节点的能力。 因为一个哈希槽实际上就是一些键的集合， 所以 Redis 集群在重哈希（rehash）时真正要做的， 就是将一些键从一个节点移动到另一个节点。

要理解 Redis 集群如何将槽从一个节点移动到另一个节点，我们需要对 CLUSTER 命令的各个子命令进行介绍，这些命理负责管理集群节点的槽转换表（slots translation table）。

以下是 CLUSTER 命令可用的子命令：

CLUSTER ADDSLOTS slot1 [slot2] ... [slotN]

CLUSTER DELSLOTS slot1 [slot2] ... [slotN]

CLUSTER SETSLOT slot NODE node

CLUSTER SETSLOT slot MIGRATING node

CLUSTER SETSLOT slot IMPORTING node

最开头的两条命令 ADDSLOTS 和 DELSLOTS 分别用于向节点指派（assign）或者移除节点，当槽被指派或者移除之后，节点会将这一信息通过 Gossip 协议传播到整个集群。ADDSLOTS 命令通常在新创建集群时，作为一种快速地将各个槽指派给各个节点的手段来使用。

CLUSTER SETSLOT slot NODE node 子命令可以将指定的槽 slot 指派给节点 node 。

至于 CLUSTER SETSLOT slot MIGRATING node 命令和 CLUSTER SETSLOT slot IMPORTING node 命令，前者用于将给定节点 node 中的槽 slot 迁移出节点，而后者用于将给定槽 slot 导入到节点 node ：

当一个槽被设置为 MIGRATING 状态时，原来持有这个槽的节点仍然会继续接受关于这个槽的命令请求，但只有命令所处理的键仍然存在于节点时，节点才会处理这个命令请求。

如果命令所使用的键不存在与该节点，那么节点将向客户端返回一个 -ASK 转向（redirection）错误，告知客户端，要将命令请求发送到槽的迁移目标节点。

当一个槽被设置为 IMPORTING 状态时，节点仅在接收到 ASKING 命令之后，才会接受关于这个槽的命令请求。

如果客户端没有向节点发送 ASKING 命令，那么节点会使用 -MOVED 转向错误将

命令请求转向至真正负责处理这个槽的节点。

上面关于 MIGRATING 和 IMPORTING 的说明有些难懂， 让我们用一个实际的实例来说明一下。

假设现在， 我们有 A 和 B 两个节点， 并且我们想将槽 8 从节点 A 移动到节点 B ， 于是我们：

向节点 B 发送命令 CLUSTER SETSLOT 8 IMPORTING A

向节点 A 发送命令 CLUSTER SETSLOT 8 MIGRATING B

每当客户端向其他节点发送关于哈希槽 8 的命令请求时， 这些节点都会向客户端返回指向节点 A 的转向信息：

如果命令要处理的键已经存在于槽 8 里面， 那么这个命令将由节点 A 处理。

如果命令要处理的键未存在于槽 8 里面（比如说，要向槽添加一个新的键）， 那么这个命令由节点 B 处理。

这种机制将使得节点 A 不再创建关于槽 8 的任何新键。

与此同时， 一个特殊的客户端 redis-trib 以及 Redis 集群配置程序（configuration utility）会将节点 A 中槽 8 里面的键移动到节点 B 。

键的移动操作由以下两个命令执行：

CLUSTER GETKEYSINSLOT slot count

上面的命令会让节点返回 count 个 slot 槽中的键， 对于命令所返回的每个键，redis-trib 都会向节点 A 发送一条 MIGRATE 命令， 该命令会将所指定的键原子地（atomic）从节点 A 移动到节点 B （在移动键期间，两个节点都会处于阻塞状态，以免出现竞争条件）。

以下为 MIGRATE 命令的运作原理：

MIGRATE target_host target_port key target_database id timeout

执行 MIGRATE 命令的节点会连接到 target 节点， 并将序列化后的 key 数据发送给 target ， 一旦 target 返回 OK ， 节点就将自己的 key 从数据库中删除。

从一个外部客户端的视角来看， 在某个时间点上， 键 key 要么存在于节点 A ，要么存在于节点 B ， 但不会同时存在于节点 A 和节点 B 。

因为 Redis 集群只使用 0 号数据库， 所以当 MIGRATE 命令被用于执行集群操作时， target_database 的值总是 0 。

target_database 参数的存在是为了让 MIGRATE 命令成为一个通用命令， 从而可以作用于集群以外的其他功能。

我们对 MIGRATE 命令做了优化， 使得它即使在传输包含多个元素的列表键这样的复杂数据时， 也可以保持高效。

不过， 尽管 MIGRATE 非常高效， 对一个键非常多、并且键的数据量非常大的集群来说， 集群重配置还是会占用大量的时间， 可能会导致集群没办法适应那些对于响应时间有严格要求的应用程序。

ASK 转向：

在之前介绍 MOVED 转向的时候， 我们说除了 MOVED 转向之外， 还有另一种 ASK 转向。

当节点需要让一个客户端长期地（permanently）将针对某个槽的命令请求发送至另一个节点时， 节点向客户端返回 MOVED 转向。另一方面， 当节点需要让客户端仅仅在下一个命令请求中转向至另一个节点时， 节点向客户端返回 ASK 转向。

比如说， 在我们上一节列举的槽 8 的例子中， 因为槽 8 所包含的各个键分散在节点 A 和节点 B 中， 所以当客户端在节点 A 中没找到某个键时， 它应该转向到节点 B 中去寻找， 但是这种转向应该仅仅影响一次命令查询， 而不是让客户端每次都直接去查找节点 B ： 在节点 A 所持有的属于槽 8 的键没有全部被迁移到节点 B 之前， 客户端应该先访问节点 A ， 然后再访问节点 B 。因为这种转向只针对 16384 个槽中的其中一个槽， 所以转向对集群造成的性能损耗属于可接受的范围。

因为上述原因， 如果我们要在查找节点 A 之后， 继续查找节点 B ， 那么客户端在向节点 B 发送命令请求之前， 应该先发送一个 ASKING 命令， 否则这个针对带有 IMPORTING 状态的槽的命令请求将被节点 B 拒绝执行。接收到客户端 ASKING 命令的节点将为客户端设置一个一次性的标志（flag）， 使得客户端可以执行一次针对 IMPORTING 状态的槽的命令请求。从客户端的角度来看， ASK 转向的完整语义

（semantics）如下：

1、如果客户端接收到 ASK 转向， 那么将命令请求的发送对象调整为转向所指定的节点。

2、先发送一个 ASKING 命令，然后再发送真正的命令请求。

3、不必更新客户端所记录的槽 8 至节点的映射：槽 8 应该仍然映射到节点 A ，而不是节点 B 。

一旦节点 A 针对槽 8 的迁移工作完成， 节点 A 在再次收到针对槽 8 的命令请求时， 就会向客户端返回 MOVED 转向， 将关于槽 8 的命令请求长期地转向到节点 B 。

注意， 即使客户端出现 Bug ， 过早地将槽 8 映射到了节点 B 上面， 但只要这个客户端不发送 ASKING 命令， 客户端发送命令请求的时候就会遇上 MOVED 错误，并将它转向回节点 A 。

容错：

节点失效检测，以下是节点失效检查的实现方法：

1、当一个节点向另一个节点发送 PING 命令， 但是目标节点未能在给定的时限内返回 PING 命令的回复时，那么发送命令的节点会将目标节点标记为 PFAIL（possible failure，可能已失效）。等待 PING 命令回复的时限称为“节点超时时限（node timeout)”， 是一个节点选项（node-wise setting)。

2、每次当节点对其他节点发送 PING 命令的时候， 它都会随机地广播三个它所知道的节点的信息， 这些信息里面的其中一项就是说明节点是否已经被标记为 PFAIL 或者 FAIL 。

当节点接收到其他节点发来的信息时， 它会记下那些被其他节点标记为失效的节点。 这称为失效报告（failure report)。

3、如果节点已经将某个节点标记为 PFAIL ， 并且根据节点所收到的失效报告显式， 集群中的大部分其他主节点也认为那个节点进入了失效状态， 那么节点会将那个失效节点的状态标记为 FAIL 。

4、一旦某个节点被标记为 FAIL ， 关于这个节点已失效的信息就会被广播到整个

集群， 所有接收到这条信息的节点都会将失效节点标记为 FAIL 。

简单来说， 一个节点要将另一个节点标记为失效， 必须先询问其他节点的意见，并且得到大部分主节点的同意才行。因为过期的失效报告会被移除， 所以主节点要将某个节点标记为 FAIL 的话， 必须以最近接收到的失效报告作为根据。

从节点选举：一旦某个主节点进入 FAIL 状态， 如果这个主节点有一个或多个从节点存在， 那么其中一个从节点会被升级为新的主节点， 而其他从节点则会开始对这个新的主节点进行复制。

新的主节点由已下线主节点属下的所有从节点中自行选举产生， 以下是选举的条件：

1、这个节点是已下线主节点的从节点。

2、已下线主节点负责处理的槽数量非空。

3、从节点的数据被认为是可靠的， 也即是， 主从节点之间的复制连接（replication link）的断线时长不能超过节点超时时限（node timeout）乘以 REDIS_CLUSTER_SLAVE_VALIDITY_MULT 常量得出的积。

如果一个从节点满足了以上的所有条件， 那么这个从节点将向集群中的其他主节点发送授权请求， 询问它们， 是否允许自己（从节点）升级为新的主节点。

如果发送授权请求的从节点满足以下属性， 那么主节点将向从节点返 FAILOVER_AUTH_GRANTED 授权， 同意从节点的升级要求：

1、发送授权请求的是一个从节点， 并且它所属的主节点处于 FAIL 状态。

2、在已下线主节点的所有从节点中， 这个从节点的节点 ID 在排序中是最小的。

3、这个从节点处于正常的运行状态： 它没有被标记为 FAIL 状态， 也没有被标记为 PFAIL 状态。

一旦某个从节点在给定的时限内得到大部分主节点的授权， 它就会开始执行以下故障转移操作：

1、通过 PONG 数据包（packet）告知其他节点， 这个节点现在是主节点了。

2、通过 PONG 数据包告知其他节点， 这个节点是一个已升级的从节点（promoted

slave）。

3、接管（claiming）所有由已下线主节点负责处理的哈希槽。

4、显式地向所有节点广播一个 PONG 数据包，加速其他节点识别这个节点的进度，而不是等待定时的 PING / PONG 数据包。

所有其他节点都会根据新的主节点对配置进行相应的更新：

1、所有被新的主节点接管的槽会被更新。

2、已下线主节点的所有从节点会察觉到 PROMOTED 标志， 并开始对新的主节点进行复制。

3、如果已下线的主节点重新回到上线状态， 那么它会察觉到 PROMOTED 标志， 并将自身调整为现任主节点的从节点。

在集群的生命周期中， 如果一个带有 PROMOTED 标识的主节点因为某些原因转变成了从节点， 那么该节点将丢失它所带有的 PROMOTED 标识。