# 集群

Re dis 集群是Redis 提供的分布式数据库方案，集群通过分片（ sharding ）来进行数据共、事，并提供复制和故障转移功能。

## 节点

一个Redis 集群通常由多个节点（ node ）组成，在刚开始的时候，每个节点都是相互独立的，它们都处于一个只包含自己的集群当中，要组建一个真正可工作的集群，我们必须将各个独立的节点连接起来，构成一个包含多个节点的集群。

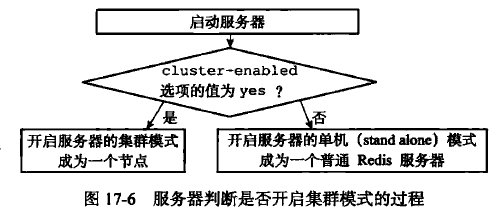
连接各个节点的工作可以使用CLUSTER MEET 命令来完成，该命令的格式如下：

CLUSTER MEET <ip> <port>

向一个节点node 发送CLUSTER MEET 命令，可以让node 节点与ip 和port 所指定的节点进行握手（ handshake ），当握手成功时， node 节点就会将ip 和port所指定的节点添加到node 节点当前所在的集群中。

### 启动节点

一个节点就是一个运行在集群模式下的Redis 服务器， Redis 服务器在启动时会根据cluster-enabled配置选项是否为yes 来决定是否开启服务器的集群模式，如图17-6 所示。



节点（运行在集群模式下的Redis 服务器）会继续使用所有在单机模式中使用的服务器组件，比如说：

* 节点会继续使用文件事件处理器来处理命令请求和返回命令回复。
* 节点会继续使用时间事件处理器来执行serverCron 函数，而serverCron 函数又会调用集群模式特有的clusterCron 函数。clusterCron 函数负责执行在集群模式下需要执行的常规操作，例如向集群中的其他节点发送Gossip 消息，检查节点是否断线，或者检查是否需要对下线节点进行自动故障转移等。
* 节点会继续使用数据库来保存键值对数据，键值对依然会是各种不间类型的对象。
* 节点会继续使用RDB 持久化模块和AOF 持久化模块来执行持久化工作。
* 节点会继续使用发布与订阅模块来执行PUBLISH、SUBSCRIBE 等命令。
* 节点会继续使用复制模块来进行节点的复制工作。
* 节点会继续使用Lua 脚本环境来执行客户端输人的Lua 脚本。

除此之外，节点会继续使用redisServer 结构来保存服务器的状态，使用redisClient 结构来保存客户端的状态，至于那些只有在集群模式下才会用到的数据，节点将它们保存到了cluster.h/clusterNode结构、cluster.h/clusterLink 结构，以及cluster.h/clusterState 结构里面。

### 集群数据结构

clusterNode 结构保存了一个节点的当前状态，比如节点的创建时间、节点的名字、节点当前的配置纪元、节点的IP 地址和端口号等等。

每个节点都会使用一个clusterNode 结构来记录自己的状态，并为集群中的所有其他节点（包括主节点和从节点）都创建一个相应的clusterNode 结构，以此来记录其他节点的状态：

|  |
| --- |
| // 节点状态  struct clusterNode {  // 创建节点的时间  mstime\_t ctime; /\* Node object creation time. \*/  // 节点的名字，由 40 个十六进制字符组成  // 例如 68eef66df23420a5862208ef5b1a7005b806f2ff  char name[REDIS\_CLUSTER\_NAMELEN]; /\* Node name, hex string, sha1-size \*/  // 节点标识  // 使用各种不同的标识值记录节点的角色（比如主节点或者从节点），  // 以及节点目前所处的状态（比如在线或者下线）。  int flags; /\* REDIS\_NODE\_... \*/  // 节点当前的配置纪元，用于实现故障转移  uint64\_t configEpoch; /\* Last configEpoch observed for this node \*/  // 由这个节点负责处理的槽  // 一共有 REDIS\_CLUSTER\_SLOTS / 8 个字节长  // 每个字节的每个位记录了一个槽的保存状态  // 位的值为 1 表示槽正由本节点处理，值为 0 则表示槽并非本节点处理  // 比如 slots[0] 的第一个位保存了槽 0 的保存情况  // slots[0] 的第二个位保存了槽 1 的保存情况，以此类推  unsigned char slots[REDIS\_CLUSTER\_SLOTS/8]; /\* slots handled by this node \*/  // 该节点负责处理的槽数量  int numslots; /\* Number of slots handled by this node \*/  // 如果本节点是主节点，那么用这个属性记录从节点的数量  int numslaves; /\* Number of slave nodes, if this is a master \*/  // 指针数组，指向各个从节点  struct clusterNode \*\*slaves; /\* pointers to slave nodes \*/  // 如果这是一个从节点，那么指向主节点  struct clusterNode \*slaveof; /\* pointer to the master node \*/  // 最后一次发送 PING 命令的时间  mstime\_t ping\_sent; /\* Unix time we sent latest ping \*/  // 最后一次接收 PONG 回复的时间戳  mstime\_t pong\_received; /\* Unix time we received the pong \*/  // 最后一次被设置为 FAIL 状态的时间  mstime\_t fail\_time; /\* Unix time when FAIL flag was set \*/  // 最后一次给某个从节点投票的时间  mstime\_t voted\_time; /\* Last time we voted for a slave of this master \*/  // 最后一次从这个节点接收到复制偏移量的时间  mstime\_t repl\_offset\_time; /\* Unix time we received offset for this node \*/  // 这个节点的复制偏移量  long long repl\_offset; /\* Last known repl offset for this node. \*/  // 节点的 IP 地址  char ip[REDIS\_IP\_STR\_LEN]; /\* Latest known IP address of this node \*/  // 节点的端口号  int port; /\* Latest known port of this node \*/  // 保存连接节点所需的有关信息  clusterLink \*link; /\* TCP/IP link with this node \*/  // 一个链表，记录了所有其他节点对该节点的下线报告  list \*fail\_reports; /\* List of nodes signaling this as failing \*/  };  typedef struct clusterNode clusterNode; |

clusterNode 结构的link 属性是一个clusterLink 结构，该结构保存了连接节点所需的有关信息，比如套接宇描述符，输入缓冲区和输出缓冲区：

|  |
| --- |
| typedef struct clusterLink {  // 连接的创建时间  mstime\_t ctime; /\* Link creation time \*/  // TCP 套接字描述符  int fd; /\* TCP socket file descriptor \*/  // 输出缓冲区，保存着等待发送给其他节点的消息（message）。  sds sndbuf; /\* Packet send buffer \*/  // 输入缓冲区，保存着从其他节点接收到的消息。  sds rcvbuf; /\* Packet reception buffer \*/  // 与这个连接相关联的节点，如果没有的话就为 NULL  struct clusterNode \*node; /\* Node related to this link if any, or NULL \*/  } clusterLink; |

**redisClient 结构和clusterlink 结构的相同和不同之处：**redisClient 结构和clusterLink 结构都有自己的套接字描述符和输入、输出缓冲区，这两个结构的区别在于， redisClient 结构中的套接字和缓冲区是用于连接客户端的，而clusterLink 结构中的套接字和缓冲区则是用于连接节点的。

最后，每个节点都保存着一个clusterState 结构，这个结构记录了在当前节点的视角下，集群目前所处的状态，例如集群是在线还是下线，集群包含多少个节点，集群当前的配置纪元，诸如此类：

|  |
| --- |
| // 集群状态，每个节点都保存着一个这样的状态，记录了它们眼中的集群的样子。  // 另外，虽然这个结构主要用于记录集群的属性，但是为了节约资源，  // 有些与节点有关的属性，比如 slots\_to\_keys 、 failover\_auth\_count  // 也被放到了这个结构里面。  typedef struct clusterState {  // 指向当前节点的指针  clusterNode \*myself; /\* This node \*/  // 集群当前的配置纪元，用于实现故障转移  uint64\_t currentEpoch;  // 集群当前的状态：是在线还是下线  int state; /\* REDIS\_CLUSTER\_OK, REDIS\_CLUSTER\_FAIL, ... \*/  // 集群中至少处理着一个槽的节点的数量。  int size; /\* Num of master nodes with at least one slot \*/  // 集群节点名单（包括 myself 节点）  // 字典的键为节点的名字，字典的值为 clusterNode 结构  dict \*nodes; /\* Hash table of name -> clusterNode structures \*/  // 节点黑名单，用于 CLUSTER FORGET 命令  // 防止被 FORGET 的命令重新被添加到集群里面  // （不过现在似乎没有在使用的样子，已废弃？还是尚未实现？）  dict \*nodes\_black\_list; /\* Nodes we don't re-add for a few seconds. \*/  // 记录要从当前节点迁移到目标节点的槽，以及迁移的目标节点  // migrating\_slots\_to[i] = NULL 表示槽 i 未被迁移  // migrating\_slots\_to[i] = clusterNode\_A 表示槽 i 要从本节点迁移至节点 A  clusterNode \*migrating\_slots\_to[REDIS\_CLUSTER\_SLOTS];  // 记录要从源节点迁移到本节点的槽，以及进行迁移的源节点  // importing\_slots\_from[i] = NULL 表示槽 i 未进行导入  // importing\_slots\_from[i] = clusterNode\_A 表示正从节点 A 中导入槽 i  clusterNode \*importing\_slots\_from[REDIS\_CLUSTER\_SLOTS];  // 负责处理各个槽的节点  // 例如 slots[i] = clusterNode\_A 表示槽 i 由节点 A 处理  clusterNode \*slots[REDIS\_CLUSTER\_SLOTS];  // 跳跃表，表中以槽作为分值，键作为成员，对槽进行有序排序  // 当需要对某些槽进行区间（range）操作时，这个跳跃表可以提供方便  // 具体操作定义在 db.c 里面  zskiplist \*slots\_to\_keys;  /\* The following fields are used to take the slave state on elections. \*/  // 以下这些域被用于进行故障转移选举  // 上次执行选举或者下次执行选举的时间  mstime\_t failover\_auth\_time; /\* Time of previous or next election. \*/  // 节点获得的投票数量  int failover\_auth\_count; /\* Number of votes received so far. \*/  // 如果值为 1 ，表示本节点已经向其他节点发送了投票请求  int failover\_auth\_sent; /\* True if we already asked for votes. \*/  int failover\_auth\_rank; /\* This slave rank for current auth request. \*/  uint64\_t failover\_auth\_epoch; /\* Epoch of the current election. \*/  /\* Manual failover state in common. \*/  /\* 共用的手动故障转移状态 \*/  // 手动故障转移执行的时间限制  mstime\_t mf\_end; /\* Manual failover time limit (ms unixtime).  It is zero if there is no MF in progress. \*/  /\* Manual failover state of master. \*/  /\* 主服务器的手动故障转移状态 \*/  clusterNode \*mf\_slave; /\* Slave performing the manual failover. \*/  /\* Manual failover state of slave. \*/  /\* 从服务器的手动故障转移状态 \*/  long long mf\_master\_offset; /\* Master offset the slave needs to start MF  or zero if stil not received. \*/  // 指示手动故障转移是否可以开始的标志值  // 值为非 0 时表示各个主服务器可以开始投票  int mf\_can\_start; /\* If non-zero signal that the manual failover  can start requesting masters vote. \*/  /\* The followign fields are uesd by masters to take state on elections. \*/  /\* 以下这些域由主服务器使用，用于记录选举时的状态 \*/  // 集群最后一次进行投票的纪元  uint64\_t lastVoteEpoch; /\* Epoch of the last vote granted. \*/  // 在进入下个事件循环之前要做的事情，以各个 flag 来记录  int todo\_before\_sleep; /\* Things to do in clusterBeforeSleep(). \*/  // 通过 cluster 连接发送的消息数量  long long stats\_bus\_messages\_sent; /\* Num of msg sent via cluster bus. \*/  // 通过 cluster 接收到的消息数量  long long stats\_bus\_messages\_received; /\* Num of msg rcvd via cluster bus.\*/  } clusterState; |

### CLUSTER MEET 命令的实现

通过向节点A 发送CLUSTER MEET命令，客户端可以让接收命令的节点A 将另一个节点B 添加到节点A 当前所在的集群里面。

CLUSTER MEET <ip> <port>

收到命令的节点A 将与节点B 进行握手（ handshake ），以此来确认彼此的存在，并为将来的进一步通信打好基础：

1 ）节点A 会为节点B 创建一个clusterNode 结构，并将该结构添加到自己的clusterState.nodes 字典里面。

2 ）之后，节点A 将根据CLUSTER MEET命令给定的E 地址和端口号，向节点B 发送一条MEET 消息（ message ）。

3 ）如果一切顺利，节点B 将接收到节点A 发送的MEET 消息，节点B 会为节点A 创建一个clusterNode 结构，并将该结构添加到自己的clusterState.nodes 宇典里面。

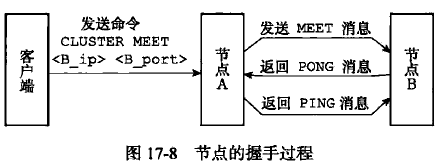
4 ）之后，节点B 将向节点A 返回一条PONG 消息。

5 ）如果一切顺利，节点A 将接收到节点B 返回的PONG 消息，通过这条PONG 消息节点A 可以知道节点B 已经成功地接收到了自己发送的MEET 消息。

6 ）之后，节点A 将向节点B 返回一条PING 捎息。

7 ）如果一切顺利，节点B 将接收到节点A 返回的P 工NG 消息，通过这条P 工NG 消息节点B 可以知道节点A 已经成功地接收到了自己返回的PONG 消息，握手完成。

图17-8 展示了以上步骤描述的握手过程。



之后，节点A 会将节点B 的信息通过Gossip 协议传播给集群中的其他节点，让其他节点也与节点B 进行握手，最终，经过一段时间之后，节点B 会被集群中的所有节点认识。

### 槽指派

Re dis 集群通过分片的方式来保存数据库中的键值对：集群的整个数据库被分为16384个槽（ slot ），数据库中的每个键都属于这16384 个槽的其中一个，集群中的每个节点可以处理0 个或最多16384 个槽。

当数据库中的16384 个槽都有节点在处理时，集群处于上线状态（ ok ）；相反地，如果数据库中有任何一个槽没有得到处理，那么集群处于下线状态（ fail)。

通过向节点发送CLUSTER ADDSLOTS 命令，我们可以将一个或多个槽指派（ assign)给节点负责：

CLUSTER ADDSLOTS <slot> [slot ... ]

### 记录节点的槽指派信息

slots 属性是一个二进制位数组（ bit array ），这个数组的长度为16384/8=2048 个字节，共包含16384 个二进制位。

Redis 以0 为起始索引， 16383 为终止索引，对s 工ots 数组中的16384 个二进制位进行编号，并根据索引土上的二进制位的值来判断节点是否负责处理槽i：

* 如果slots 数组在索引i 上的二进制位的值为1 ，那么表示节点负责处理槽i。
* 如果slots 数组在索引i 上的二进制位的值为0 ，那么表示节点不负责处理槽i 。

图17-9 展示了一个slots 数组示例：这个数组索引0 至索引7 上的二进制位的值都为1 ，其余所有二进制位的值都为0 ，这表示节点负责处理槽0 至槽7。

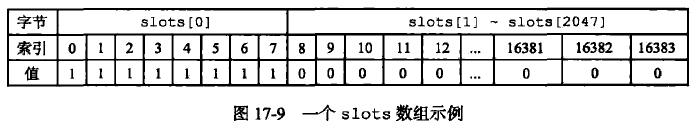


图17-10 展示了另一个slots 数组示例：这个数组索引1 、3 、5 、8、民10 上的二进制位的值都为1 ，而其余所有二进制位的值都为0 ，这表示节点负责处理槽1 、3、5、8 、9、10。

因为取出和设置slots 数组中的任意一个二进制位的值的复杂度仅为0(1 ），所以对于一个给定节点的slots 数组来说，程序检查节点是否负责处理某个槽，又或者将某个槽指派给节点负责，这两个动作的复杂度都是O(1)。

至于numslots 属性则记录节点负责处理的槽的数量，也即是slots 数组中值为l 的二进制位的数量。

比如说，对于图17-9 所示的slots 数组来说，节点处理的槽数量为8 ，而对于图17-10 所示的slots 数组来说，节点处理的槽数量为6。

### 传播节点的槽指派信息

一个节点除了会将自己负责处理的槽记录在clusterNode 结构的slots 属性和numslots 属性之外，它还会将自己的slots 数组通过消息发送给集群中的其他节点，以此来告知其他节点自己目前负责处理哪些槽。

因为集群中的每个节点都会将自己的slots 数组通过消息发送给集群中的其他节点，并且每个接收到slots 数组的节点都会将数组保存到相应节点的clusterNode 结构里面，因此，集群中的每个节点都会知道数据库中的16384 个槽分别被指派给了集群中的哪些节点。