Redis集群通过分片的方式来保存数据库中的键值对：一个集群中，每个键都通过哈希函数映射到一个槽位，整个集群共分16384个槽位，集群中每个主节点负责其中的一部分槽位。

当数据库中的16384个槽位都有节点在处理时，集群处于上线状态；相反，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。

所谓键的分配，实际上就是指槽位在集群节点中的分配；所谓键的迁移，实际上指槽位在集群节点间的迁移。

**一：数据结构**

在集群最主要的数据结构，记录集群状态的clusterState结构体中，与槽位相关的属性有：

clusterNode **\***slots**[**16384**];**

clusterNode **\***migrating\_slots\_to**[**16384**];**

clusterNode **\***importing\_slots\_from**[**16384**];**

zskiplist **\***slots\_to\_keys**;**

slots数组记录了16384个槽位，分别由哪个集群节点负责：比如server->cluster.slots[0] = node，这说明0号槽位由node节点负责；

migrating\_slots\_to数组记录了16384个槽位中，当前节点所负责的槽位正在迁出到哪个节点。比如server.cluster->migrating\_slots\_to[0] = node，这说明当前节点负责的0号槽位，正在迁出到node节点；

importing\_slots\_from数组记录了16384个槽位中，当前节点正在从哪个节点将某个槽位迁入到本节点中；比如server.cluster->importing\_slots\_from[0] = node，这说明当前节点正在从node节点处迁入0号槽位；

通过以上这些属性，可以快速得到某个槽位由哪个节点负责，以及该槽位正在迁出或迁入到哪个节点。

slots\_to\_keys是个跳跃表，该跳跃表中，以槽位号为分数进行排序。每个跳跃表节点保存了槽位号(分数)，以及该槽位上的某个key。通过该跳跃表，可以快速得到当前节点所负责的每一个槽位中，都有哪些key。

在表示集群节点的clusterNode结构体中，与槽位相关的属性有：

unsigned char slots**[**16384**/**8**];**

int numslots**;**

slots记录了节点负责处理哪些槽位。它是个位数组，其中每一个比特位表示一个槽位号，如果该比特位置为1，则说明该槽位由该节点负责；

numslots表示该节点负责的槽位总数；

通过以上这些属性，可以快速得到某个节点负责哪些槽位。

**二：分配槽位**

在集群刚建立时，需要手动为每个集群主节点分配其负责的槽位。这主要是通过向节点发送”CLUSTER ADDSLOTS”命令实现的。该命令的格式是：”CLUSTER ADDSLOTS <slot> [slot] ...”。

“CLUSTER”命令的处理函数是clusterCommand。在该函数中，处理” CLUSTER ADDSLOTS”部分的代码是：

**else** **if** **((!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"addslots"**)** **||**

**!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"delslots"**))** **&&** c**->**argc **>=** 3**)**

**{**

*/\* CLUSTER ADDSLOTS <slot> [slot] ... \*/*

*/\* CLUSTER DELSLOTS <slot> [slot] ... \*/*

int j**,** slot**;**

unsigned char **\***slots **=** zmalloc**(**REDIS\_CLUSTER\_SLOTS**);**

int del **=** **!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"delslots"**);**

memset**(**slots**,**0**,**REDIS\_CLUSTER\_SLOTS**);**

*/\* Check that all the arguments are parseable and that all the*

*\* slots are not already busy. \*/*

**for** **(**j **=** 2**;** j **<** c**->**argc**;** j**++)** **{**

**if** **((**slot **=** getSlotOrReply**(**c**,**c**->**argv**[**j**]))** **==** **-**1**)** **{**

zfree**(**slots**);**

**return;**

**}**

**if** **(**del **&&** server**.**cluster**->**slots**[**slot**]** **==** **NULL)** **{**

addReplyErrorFormat**(**c**,**"Slot %d is already unassigned"**,** slot**);**

zfree**(**slots**);**

**return;**

**}** **else** **if** **(!**del **&&** server**.**cluster**->**slots**[**slot**])** **{**

addReplyErrorFormat**(**c**,**"Slot %d is already busy"**,** slot**);**

zfree**(**slots**);**

**return;**

**}**

**if** **(**slots**[**slot**]++** **==** 1**)** **{**

addReplyErrorFormat**(**c**,**"Slot %d specified multiple times"**,**

**(**int**)**slot**);**

zfree**(**slots**);**

**return;**

**}**

**}**

**for** **(**j **=** 0**;** j **<** REDIS\_CLUSTER\_SLOTS**;** j**++)** **{**

**if** **(**slots**[**j**])** **{**

int retval**;**

*/\* If this slot was set as importing we can clear this*

*\* state as now we are the real owner of the slot. \*/*

**if** **(**server**.**cluster**->**importing\_slots\_from**[**j**])**

server**.**cluster**->**importing\_slots\_from**[**j**]** **=** **NULL;**

retval **=** del **?** clusterDelSlot**(**j**)** **:**

clusterAddSlot**(**myself**,**j**);**

redisAssertWithInfo**(**c**,NULL,**retval **==** REDIS\_OK**);**

**}**

**}**

zfree**(**slots**);**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_UPDATE\_STATE**|**CLUSTER\_TODO\_SAVE\_CONFIG**);**

addReply**(**c**,**shared**.**ok**);**

**}**

这里” CLUSTER ADDSLOTS”和” CLUSTER DELSLOTS”命令，采用类似的代码进行处理。ADDSLOTS和DELSLOTS，分别用于将槽位分配给节点，以及将槽位从节点中删除。ADDSLOTS命令常用于新建集群时，给每个主节点分配槽位；DELSLOTS常用于手动修改集群配置，或者用于DEBUG操作，实际中很少用到。

在代码中，首先，依次检查命令参数中的槽位号：如果是DELSLOTS操作，但是数组server.cluster->slots中，记录负责该槽位号的节点为NULL，则反馈给客户端"unassigned"错误；如果是ADDSLOTS操作，但是数组server.cluster->slots中，记录已经有节点负责该槽位号了，则反馈给客户端"busy"错误；然后将参数中的槽位号记录到数组slots中，如果slots中该槽位已经设置过了，说明发来的命令中，该槽位号出现了多次，因此反馈给客户端"multiple times"错误；

然后，依次轮训slots中记录的每一个槽位号进行处理：首先如果该槽位号在数组server.cluster->importing\_slots\_from中不为NULL，则将其置为NULL，因为该槽位已经由本节点负责了；然后根据是ADDSLOTS，还是DELSLOTS操作，调用clusterAddSlot或clusterDelSlot处理；

最后，反馈给客户端"OK";

因此，clusterAddSlot才是是实际用于分配槽位的函数，该函数的实现如下：

int clusterAddSlot**(**clusterNode **\***n**,** int slot**)** **{**

**if** **(**server**.**cluster**->**slots**[**slot**])** **return** REDIS\_ERR**;**

clusterNodeSetSlotBit**(**n**,**slot**);**

server**.**cluster**->**slots**[**slot**]** **=** n**;**

**return** REDIS\_OK**;**

**}**

该函数的实现很简单，就是要设置位数组n->slots中的相应位，以及server.cluster->slots[slot]。

首先，根据server.cluster->slots[slot]的值，判断该槽位是否已经分配给其他节点了，若是，则直接返回REDIS\_ERR；

然后调用clusterNodeSetSlotBit，在位数组n->slots中设置相应的位；

最后，将server.cluster->slots[slot]置为n；

以上，就相当于把slot槽位分配给了节点n。

顺便看一下删除槽位的函数clusterDelSlot的实现：

int clusterDelSlot**(**int slot**)** **{**

clusterNode **\***n **=** server**.**cluster**->**slots**[**slot**];**

**if** **(!**n**)** **return** REDIS\_ERR**;**

redisAssert**(**clusterNodeClearSlotBit**(**n**,**slot**)** **==** 1**);**

server**.**cluster**->**slots**[**slot**]** **=** **NULL;**

**return** REDIS\_OK**;**

**}**

该函数清除slot槽位的信息，将其置为未分配的。成功返回REDIS\_OK；否则若该槽位已经被置为未分配的了，则返回REDIS\_ERR；

该函数的实现很简单，就是清除位数组n->slots中的相应位，以及将server.cluster->slots[slot]置为NULL。

首先从server.cluster->slots[slot]取得当前负责该槽位的节点n；如果n为NULL，则返回REDIS\_ERR；

然后调用clusterNodeClearSlotBit，将该槽位从位数组n->slots中清除；

最后置server.cluster->slots[slot]为NULL；

以上，就相当于把slot槽位置为未分配状态了。

集群节点在发送心跳包时，会附带自己当前记录的槽位信息（clusterNode结构中的位数组slots），这样，最终集群中的每个节点都会知道所有槽位的分配情况。

**三：槽位迁移(重新分片)**

在集群稳定一段时间之后，如果有新的集群节点加入，或者某个集群节点下线了。此时就涉及到将某个节点上的槽位迁移到另一个节点上的问题。

该过程也是需要手动完成的，Redis提供了辅助脚本redis-trib.rb，以”reshard”参数调用该脚本就可以实现重新分片的操作。但是本质上，该脚本就是通过向迁入节点和迁出节点发送一些命令实现的。

槽位迁移的步骤是：

1：向迁入节点发送” [CLUSTER SETSLOT](http://redis.io/commands/cluster-setslot)  <slot> IMPORTING <node>”命令

其中<slot>是要迁入的槽位号，<node>是当前负责该槽位的节点。在函数clusterCommand中，处理该命令的代码如下：

**else** **if** **(!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"setslot"**)** **&&** c**->**argc **>=** 4**)** **{**

*/\* SETSLOT 10 MIGRATING <node ID> \*/*

*/\* SETSLOT 10 IMPORTING <node ID> \*/*

*/\* SETSLOT 10 STABLE \*/*

*/\* SETSLOT 10 NODE <node ID> \*/*

int slot**;**

clusterNode **\***n**;**

**if** **((**slot **=** getSlotOrReply**(**c**,**c**->**argv**[**2**]))** **==** **-**1**)** **return;**

**if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"migrating"**)** **&&** c**->**argc **==** 5**)** **{**

**...**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"importing"**)** **&&** c**->**argc **==** 5**)** **{**

**if** **(**server**.**cluster**->**slots**[**slot**]** **==** myself**)** **{**

addReplyErrorFormat**(**c**,**

"I'm already the owner of hash slot %u"**,**slot**);**

**return;**

**}**

**if** **((**n **=** clusterLookupNode**(**c**->**argv**[**4**]->**ptr**))** **==** **NULL)** **{**

addReplyErrorFormat**(**c**,**"I don't know about node %s"**,**

**(**char**\*)**c**->**argv**[**3**]->**ptr**);**

**return;**

**}**

server**.**cluster**->**importing\_slots\_from**[**slot**]** **=** n**;**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"stable"**)** **&&** c**->**argc **==** 4**)** **{**

**...**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"node"**)** **&&** c**->**argc **==** 5**)** **{**

**...**

**}** **else** **{**

addReplyError**(**c**,**

"Invalid CLUSTER SETSLOT action or number of arguments"**);**

**return;**

**}**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**|**CLUSTER\_TODO\_UPDATE\_STATE**);**

addReply**(**c**,**shared**.**ok**);**

**}**

针对"CLUSTER SETSLOT"命令，首先从命令参数中取得槽位号slot，如果解析错误，则回复给客户端错误信息，然后直接返回；

如果收到的是" CLUSTER SETSLOT <SLOT> IMPORTING <node>"命令，说明本节点需要迁入槽位。

因此，首先判断server.cluster->slots[slot]是否等于myself，若是，说明slot槽位已由本节点负责，因此回复客户端错误信息后直接返回；然后根据参数<node>在字典server.cluster->nodes中查询迁入槽位的源节点n，若找不到，则回复客户端错误信息后返回；最后，置server.cluster->importing\_slots\_from[slot]为n；

2：向迁出节点发送” [CLUSTER SETSLOT](http://redis.io/commands/cluster-setslot)  <slot> MIGRATING <node>”命令

其中<slot>是要迁出的槽位号，<node>是迁出槽位的目的地节点。在函数clusterCommand中，处理该命令的代码如下：

**else** **if** **(!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"setslot"**)** **&&** c**->**argc **>=** 4**)** **{**

*/\* SETSLOT 10 MIGRATING <node ID> \*/*

*/\* SETSLOT 10 IMPORTING <node ID> \*/*

*/\* SETSLOT 10 STABLE \*/*

*/\* SETSLOT 10 NODE <node ID> \*/*

int slot**;**

clusterNode **\***n**;**

**if** **((**slot **=** getSlotOrReply**(**c**,**c**->**argv**[**2**]))** **==** **-**1**)** **return;**

**if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"migrating"**)** **&&** c**->**argc **==** 5**)** **{**

**if** **(**server**.**cluster**->**slots**[**slot**]** **!=** myself**)** **{**

addReplyErrorFormat**(**c**,**"I'm not the owner of hash slot %u"**,**slot**);**

**return;**

**}**

**if** **((**n **=** clusterLookupNode**(**c**->**argv**[**4**]->**ptr**))** **==** **NULL)** **{**

addReplyErrorFormat**(**c**,**"I don't know about node %s"**,**

**(**char**\*)**c**->**argv**[**4**]->**ptr**);**

**return;**

**}**

server**.**cluster**->**migrating\_slots\_to**[**slot**]** **=** n**;**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"importing"**)** **&&** c**->**argc **==** 5**)** **{**

**...**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"stable"**)** **&&** c**->**argc **==** 4**)** **{**

**...**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"node"**)** **&&** c**->**argc **==** 5**)** **{**

**...**

**}** **else** **{**

addReplyError**(**c**,**

"Invalid CLUSTER SETSLOT action or number of arguments"**);**

**return;**

**}**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**|**CLUSTER\_TODO\_UPDATE\_STATE**);**

addReply**(**c**,**shared**.**ok**);**

**}**

如果收到的是"CLUSTER SETSLOT <SLOT> MIGRATING <node>"命令，说明本节点需要迁出槽位。

因此，首先判断server.cluster->slots[slot]是否等于myself，若不是，说明slot槽位不由本节点负责，因此回复客户端错误信息后直接返回；然后根据参数<node>在字典server.cluster->nodes中查询迁出的目的地节点n，若找不到，则回复客户端错误信息后返回；最后，置server.cluster->migrating\_slots\_to[slot]为n；

3：向迁出节点发送”CLUSTER GETKEYSINSLOT <slot> <count>”命令

该命令主要用于获得迁出槽位<slot>中的<count>个key，以便下一步能够执行key的迁移操作。该命令以及下一步的key迁移操作需要执行多次，直到槽位<slot>中没有剩余的key为止。

这里就需要用到之前介绍过的，clusterState结构体中的slots\_to\_keys跳跃表，该跳跃表中，以槽位号为分数进行排序。每个跳跃表节点保存了槽位号(分数)，以及该槽位上的某个key。通过该跳跃表，可以快速得到当前节点所负责的每一个槽位中，都有哪些key。

每当向数据库中添加或删除key时，同时也会向该跳跃表中添加和删除节点：当调用dbAdd函数向数据库添加key时，在dbAdd中，判断如果当前处于集群模式下，就会调用slotToKeyAdd函数，向slots\_to\_keys跳跃表中添加节点。slotToKeyAdd函数的代码如下：

void slotToKeyAdd**(**robj **\***key**)** **{**

unsigned int hashslot **=** keyHashSlot**(**key**->**ptr**,**sdslen**(**key**->**ptr**));**

zslInsert**(**server**.**cluster**->**slots\_to\_keys**,**hashslot**,**key**);**

incrRefCount**(**key**);**

**}**

该函数很简单，首先计算该key对应的槽位号hashslot；然后以槽位号hashslot为分数，将hashslot和key插入到跳跃表server.cluster->slots\_to\_keys中；

当调用dbDelete函数从数据库删除key时，在dbDelete中，判断如果当前处于集群模式下，就会调用slotToKeyDel函数，从slots\_to\_keys跳跃表中删除节点。slotToKeyDel函数的代码如下：

void slotToKeyDel**(**robj **\***key**)** **{**

unsigned int hashslot **=** keyHashSlot**(**key**->**ptr**,**sdslen**(**key**->**ptr**));**

zslDelete**(**server**.**cluster**->**slots\_to\_keys**,**hashslot**,**key**);**

**}**

该函数很简单，首先计算该key对应的槽位号hashslot；然后将该key，及其对应的槽位号，从跳跃表server.cluster->slots\_to\_keys中删除。

回到”CLUSTER GETKEYSINSLOT“命令，在函数clusterCommand中，处理该命令的代码如下：

**else** **if** **(!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"getkeysinslot"**)** **&&** c**->**argc **==** 4**)** **{**

*/\* CLUSTER GETKEYSINSLOT <slot> <count> \*/*

long long maxkeys**,** slot**;**

unsigned int numkeys**,** j**;**

robj **\*\***keys**;**

**if** **(**getLongLongFromObjectOrReply**(**c**,**c**->**argv**[**2**],&**slot**,NULL)** **!=** REDIS\_OK**)**

**return;**

**if** **(**getLongLongFromObjectOrReply**(**c**,**c**->**argv**[**3**],&**maxkeys**,NULL)**

**!=** REDIS\_OK**)**

**return;**

**if** **(**slot **<** 0 **||** slot **>=** REDIS\_CLUSTER\_SLOTS **||** maxkeys **<** 0**)** **{**

addReplyError**(**c**,**"Invalid slot or number of keys"**);**

**return;**

**}**

keys **=** zmalloc**(sizeof(**robj**\*)\***maxkeys**);**

numkeys **=** getKeysInSlot**(**slot**,** keys**,** maxkeys**);**

addReplyMultiBulkLen**(**c**,**numkeys**);**

**for** **(**j **=** 0**;** j **<** numkeys**;** j**++)** addReplyBulk**(**c**,**keys**[**j**]);**

zfree**(**keys**);**

**}**

首先从命令参数中解析出槽位号slot，以及要获取的key的数量maxkeys。如果解析失败，或者得到的值不正常，则回复客户端错误信息后直接返回；

然后调用getKeysInSlot，从跳跃表server.cluster->slots\_to\_keys中取出slot槽位中最多maxkeys个key，取出的key存入数组keys中；getKeysInSlot函数返回实际取得的key的数量；

最后，将取得的所有key及数量回复给客户端；

getKeysInSlot函数的代码如下：

unsigned int getKeysInSlot**(**unsigned int hashslot**,** robj **\*\***keys**,** unsigned int count**)** **{**

zskiplistNode **\***n**;**

zrangespec range**;**

int j **=** 0**;**

range**.**min **=** range**.**max **=** hashslot**;**

range**.**minex **=** range**.**maxex **=** 0**;**

n **=** zslFirstInRange**(**server**.**cluster**->**slots\_to\_keys**,** **&**range**);**

**while(**n **&&** n**->**score **==** hashslot **&&** count**--)** **{**

keys**[**j**++]** **=** n**->**obj**;**

n **=** n**->**level**[**0**].**forward**;**

**}**

**return** j**;**

**}**

根据槽位号，得到要查找的范围是[hashslot, hashslot]，首先调用zslFirstInRange，在跳跃表中找到第一个处于该范围的节点；然后依次轮训该节点及其在level0上的后继节点，只要节点的分数为hashslot，就将该节点的key填充到keys中；

最后返回实际获取的key的个数。

4：向迁出节点发送”MIGRATE <target\_host> <target\_port> <key> <target\_database> <timeout>”命令

针对上一步得到的每一个key，向迁出节点发送该命令，用于将<key>迁出到目标节点的<target\_database>数据库中，迁出过程的超时时间为<timeout>，一旦超时，则回复客户端错误信息。

该命令不仅可以用于集群节点间的key迁移，还能用于普通节点间的key迁移。如果是在集群模式下，则<target\_database>固定为0。

该命令是原子性的将key从A迁移到B，迁移过程中，节点A和节点B都会阻塞（很小的时间），从而避免了竞争的发生。

4.1、缓存连接

因为一般情况下，是需要将多个key从A迁移到B中，为了避免A和B之间需要多次TCP建链，这里采用了缓存连接的实现方法。具体而言，当迁移第一个key时，节点A向节点B建链，并将该TCP链接缓存起来，一定时间内，当需要迁移下一个key时，可以直接使用缓存的链接，而无需重复建链。缓存的链接如果长时间不用，则会自动释放。

源码中使用migrateCachedSocket结构体表示缓存的TCP连接，该结构体的定义如下：

**typedef** struct migrateCachedSocket **{**

int fd**;**

long last\_dbid**;**

time\_t last\_use\_time**;**

**}** migrateCachedSocket**;**

该结构中保存了socket描述符fd，上一次使用的目的节点的数据库ID，以及该链接上一次被使用的时间。

migrateGetSocket就是用于建链并缓存的函数，该函数的代码如下：

migrateCachedSocket**\*** migrateGetSocket**(**redisClient **\***c**,** robj **\***host**,** robj **\***port**,** long timeout**)** **{**

int fd**;**

sds name **=** sdsempty**();**

migrateCachedSocket **\***cs**;**

*/\* Check if we have an already cached socket for this ip:port pair. \*/*

name **=** sdscatlen**(**name**,**host**->**ptr**,**sdslen**(**host**->**ptr**));**

name **=** sdscatlen**(**name**,**":"**,**1**);**

name **=** sdscatlen**(**name**,**port**->**ptr**,**sdslen**(**port**->**ptr**));**

cs **=** dictFetchValue**(**server**.**migrate\_cached\_sockets**,**name**);**

**if** **(**cs**)** **{**

sdsfree**(**name**);**

cs**->**last\_use\_time **=** server**.**unixtime**;**

**return** cs**;**

**}**

*/\* No cached socket, create one. \*/*

**if** **(**dictSize**(**server**.**migrate\_cached\_sockets**)** **==** MIGRATE\_SOCKET\_CACHE\_ITEMS**)** **{**

*/\* Too many items, drop one at random. \*/*

dictEntry **\***de **=** dictGetRandomKey**(**server**.**migrate\_cached\_sockets**);**

cs **=** dictGetVal**(**de**);**

close**(**cs**->**fd**);**

zfree**(**cs**);**

dictDelete**(**server**.**migrate\_cached\_sockets**,**dictGetKey**(**de**));**

**}**

*/\* Create the socket \*/*

fd **=** anetTcpNonBlockConnect**(**server**.**neterr**,**c**->**argv**[**1**]->**ptr**,**

atoi**(**c**->**argv**[**2**]->**ptr**));**

**if** **(**fd **==** **-**1**)** **{**

sdsfree**(**name**);**

addReplyErrorFormat**(**c**,**"Can't connect to target node: %s"**,**

server**.**neterr**);**

**return** **NULL;**

**}**

anetEnableTcpNoDelay**(**server**.**neterr**,**fd**);**

*/\* Check if it connects within the specified timeout. \*/*

**if** **((**aeWait**(**fd**,**AE\_WRITABLE**,**timeout**)** **&** AE\_WRITABLE**)** **==** 0**)** **{**

sdsfree**(**name**);**

addReplySds**(**c**,**

sdsnew**(**"-IOERR error or timeout connecting to the client\r\n"**));**

close**(**fd**);**

**return** **NULL;**

**}**

*/\* Add to the cache and return it to the caller. \*/*

cs **=** zmalloc**(sizeof(\***cs**));**

cs**->**fd **=** fd**;**

cs**->**last\_dbid **=** **-**1**;**

cs**->**last\_use\_time **=** server**.**unixtime**;**

dictAdd**(**server**.**migrate\_cached\_sockets**,**name**,**cs**);**

**return** cs**;**

**}**

字典server.migrate\_cached\_sockets表示一个缓存连接池，该字典以目的节点的"<ip>:<port>"为key，以migrateCachedSocket结构为value。该字典中就保存了当前节点所有已经建链的TCP连接；

函数中，首先根据参数host和port，组成key，使用该key查询字典server.migrate\_cached\_sockets中是否已经缓存了到该地址的连接cs，若找到了缓存的cs，则更新cs->last\_use\_time为当前时间，然后直接返回cs即可；

若找不到相应的连接cs，则判断字典当前的大小是否已经达到了阈值64，若是，则从字典中随机选择一个字典项de，取出其中的连接cs，关闭cs->fd，释放cs结构，并将de从字典中删除；

接下来，调用anetTcpNonBlockConnect，根据地址信息，向远端Redis发起TCP建链，如果anetTcpNonBlockConnect返回-1，则回复给客户端错误信息后，直接返回NULL；

然后设置socket描述符fd的NO\_DELAY选项；然后调用aeWait，等待可写事件的触发，等待时间为timeout，如果在该时间段内没有触发可写事件，则建链超时，因此回复给客户端错误信息，关闭socket描述符，返回NULL；否则，表示建链成功（实际上并没有检查建链是否真的成功，若建链失败，后续调用者在写消息时会发生错误，从而释放连接）；

接下来，构建一个migrateCachedSocket结构的cs，保存socket描述符，置其中的last\_dbid为-1，置last\_use\_time属性为当前时间；然后将cs插入到字典server.migrate\_cached\_sockets中。

当某个连接长时间不用时，需要断开连接，删除缓存的migrateCachedSocket结构。这是通过migrateCloseTimedoutSockets函数实现的。该函数每隔1秒在定时器函数serverCron中调用一次。该函数的代码如下：

void migrateCloseTimedoutSockets**(**void**)** **{**

dictIterator **\***di **=** dictGetSafeIterator**(**server**.**migrate\_cached\_sockets**);**

dictEntry **\***de**;**

**while((**de **=** dictNext**(**di**))** **!=** **NULL)** **{**

migrateCachedSocket **\***cs **=** dictGetVal**(**de**);**

**if** **((**server**.**unixtime **-** cs**->**last\_use\_time**)** **>** MIGRATE\_SOCKET\_CACHE\_TTL**)** **{**

close**(**cs**->**fd**);**

zfree**(**cs**);**

dictDelete**(**server**.**migrate\_cached\_sockets**,**dictGetKey**(**de**));**

**}**

**}**

dictReleaseIterator**(**di**);**

**}**

轮训字典server.migrate\_cached\_sockets，针对其中的每一个migrateCachedSocket结构的cs，如果该cs的最后一次使用时间，距离当前时间已经超过10s，则关闭socket描述符，释放cs结构，并将其从字典中删除。

4.2、MIGRATE命令

MIGRATE命令的格式是：”MIGRATE <target\_host> <target\_port> <key> <target\_database> <timeout> [COPY | REPLACE]"，如果最后一个参数是REPLACE，则发送成功之后，还要在当前实例中删除该key；如果是COPY，则无需删除key；默认参数就是REPLACE。

MIGRATE命令的处理函数是migrateCommand，该函数的代码如下：

void migrateCommand**(**redisClient **\***c**)** **{**

migrateCachedSocket **\***cs**;**

int copy**,** replace**,** j**;**

long timeout**;**

long dbid**;**

long long ttl**,** expireat**;**

robj **\***o**;**

rio cmd**,** payload**;**

int retry\_num **=** 0**;**

try\_again**:**

*/\* Initialization \*/*

copy **=** 0**;**

replace **=** 0**;**

ttl **=** 0**;**

*/\* Parse additional options \*/*

**for** **(**j **=** 6**;** j **<** c**->**argc**;** j**++)** **{**

**if** **(!**strcasecmp**(**c**->**argv**[**j**]->**ptr**,**"copy"**))** **{**

copy **=** 1**;**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**j**]->**ptr**,**"replace"**))** **{**

replace **=** 1**;**

**}** **else** **{**

addReply**(**c**,**shared**.**syntaxerr**);**

**return;**

**}**

**}**

*/\* Sanity check \*/*

**if** **(**getLongFromObjectOrReply**(**c**,**c**->**argv**[**5**],&**timeout**,NULL)** **!=** REDIS\_OK**)**

**return;**

**if** **(**getLongFromObjectOrReply**(**c**,**c**->**argv**[**4**],&**dbid**,NULL)** **!=** REDIS\_OK**)**

**return;**

**if** **(**timeout **<=** 0**)** timeout **=** 1000**;**

*/\* Check if the key is here. If not we reply with success as there is*

*\* nothing to migrate (for instance the key expired in the meantime), but*

*\* we include such information in the reply string. \*/*

**if** **((**o **=** lookupKeyRead**(**c**->**db**,**c**->**argv**[**3**]))** **==** **NULL)** **{**

addReplySds**(**c**,**sdsnew**(**"+NOKEY\r\n"**));**

**return;**

**}**

*/\* Connect \*/*

cs **=** migrateGetSocket**(**c**,**c**->**argv**[**1**],**c**->**argv**[**2**],**timeout**);**

**if** **(**cs **==** **NULL)** **return;** */\* error sent to the client by migrateGetSocket() \*/*

rioInitWithBuffer**(&**cmd**,**sdsempty**());**

*/\* Send the SELECT command if the current DB is not already selected. \*/*

int select **=** cs**->**last\_dbid **!=** dbid**;** */\* Should we emit SELECT? \*/*

**if** **(**select**)** **{**

redisAssertWithInfo**(**c**,NULL,**rioWriteBulkCount**(&**cmd**,**'\*'**,**2**));**

redisAssertWithInfo**(**c**,NULL,**rioWriteBulkString**(&**cmd**,**"SELECT"**,**6**));**

redisAssertWithInfo**(**c**,NULL,**rioWriteBulkLongLong**(&**cmd**,**dbid**));**

**}**

*/\* Create RESTORE payload and generate the protocol to call the command. \*/*

expireat **=** getExpire**(**c**->**db**,**c**->**argv**[**3**]);**

**if** **(**expireat **!=** **-**1**)** **{**

ttl **=** expireat**-**mstime**();**

**if** **(**ttl **<** 1**)** ttl **=** 1**;**

**}**

redisAssertWithInfo**(**c**,NULL,**rioWriteBulkCount**(&**cmd**,**'\*'**,**replace **?** 5 **:** 4**));**

**if** **(**server**.**cluster\_enabled**)**

redisAssertWithInfo**(**c**,NULL,**

rioWriteBulkString**(&**cmd**,**"RESTORE-ASKING"**,**14**));**

**else**

redisAssertWithInfo**(**c**,NULL,**rioWriteBulkString**(&**cmd**,**"RESTORE"**,**7**));**

redisAssertWithInfo**(**c**,NULL,**sdsEncodedObject**(**c**->**argv**[**3**]));**

redisAssertWithInfo**(**c**,NULL,**rioWriteBulkString**(&**cmd**,**c**->**argv**[**3**]->**ptr**,**

sdslen**(**c**->**argv**[**3**]->**ptr**)));**

redisAssertWithInfo**(**c**,NULL,**rioWriteBulkLongLong**(&**cmd**,**ttl**));**

*/\* Emit the payload argument, that is the serialized object using*

*\* the DUMP format. \*/*

createDumpPayload**(&**payload**,**o**);**

redisAssertWithInfo**(**c**,NULL,**rioWriteBulkString**(&**cmd**,**payload**.**io**.**buffer**.**ptr**,**

sdslen**(**payload**.**io**.**buffer**.**ptr**)));**

sdsfree**(**payload**.**io**.**buffer**.**ptr**);**

*/\* Add the REPLACE option to the RESTORE command if it was specified*

*\* as a MIGRATE option. \*/*

**if** **(**replace**)**

redisAssertWithInfo**(**c**,NULL,**rioWriteBulkString**(&**cmd**,**"REPLACE"**,**7**));**

*/\* Transfer the query to the other node in 64K chunks. \*/*

errno **=** 0**;**

**{**

sds buf **=** cmd**.**io**.**buffer**.**ptr**;**

size\_t pos **=** 0**,** towrite**;**

int nwritten **=** 0**;**

**while** **((**towrite **=** sdslen**(**buf**)-**pos**)** **>** 0**)** **{**

towrite **=** **(**towrite **>** **(**64**\***1024**)** **?** **(**64**\***1024**)** **:** towrite**);**

nwritten **=** syncWrite**(**cs**->**fd**,**buf**+**pos**,**towrite**,**timeout**);**

**if** **(**nwritten **!=** **(**signed**)**towrite**)** **goto** socket\_wr\_err**;**

pos **+=** nwritten**;**

**}**

**}**

*/\* Read back the reply. \*/*

**{**

char buf1**[**1024**];**

char buf2**[**1024**];**

*/\* Read the two replies \*/*

**if** **(**select **&&** syncReadLine**(**cs**->**fd**,** buf1**,** **sizeof(**buf1**),** timeout**)** **<=** 0**)**

**goto** socket\_rd\_err**;**

**if** **(**syncReadLine**(**cs**->**fd**,** buf2**,** **sizeof(**buf2**),** timeout**)** **<=** 0**)**

**goto** socket\_rd\_err**;**

**if** **((**select **&&** buf1**[**0**]** **==** '-'**)** **||** buf2**[**0**]** **==** '-'**)** **{**

*/\* On error assume that last\_dbid is no longer valid. \*/*

cs**->**last\_dbid **=** **-**1**;**

addReplyErrorFormat**(**c**,**"Target instance replied with error: %s"**,**

**(**select **&&** buf1**[**0**]** **==** '-'**)** **?** buf1**+**1 **:** buf2**+**1**);**

**}** **else** **{**

*/\* Update the last\_dbid in migrateCachedSocket \*/*

cs**->**last\_dbid **=** dbid**;**

robj **\***aux**;**

addReply**(**c**,**shared**.**ok**);**

**if** **(!**copy**)** **{**

*/\* No COPY option: remove the local key, signal the change. \*/*

dbDelete**(**c**->**db**,**c**->**argv**[**3**]);**

signalModifiedKey**(**c**->**db**,**c**->**argv**[**3**]);**

server**.**dirty**++;**

*/\* Translate MIGRATE as DEL for replication/AOF. \*/*

aux **=** createStringObject**(**"DEL"**,**3**);**

rewriteClientCommandVector**(**c**,**2**,**aux**,**c**->**argv**[**3**]);**

decrRefCount**(**aux**);**

**}**

**}**

**}**

sdsfree**(**cmd**.**io**.**buffer**.**ptr**);**

**return;**

socket\_wr\_err**:**

sdsfree**(**cmd**.**io**.**buffer**.**ptr**);**

migrateCloseSocket**(**c**->**argv**[**1**],**c**->**argv**[**2**]);**

**if** **(**errno **!=** ETIMEDOUT **&&** retry\_num**++** **==** 0**)** **goto** try\_again**;**

addReplySds**(**c**,**

sdsnew**(**"-IOERR error or timeout writing to target instance\r\n"**));**

**return;**

socket\_rd\_err**:**

sdsfree**(**cmd**.**io**.**buffer**.**ptr**);**

migrateCloseSocket**(**c**->**argv**[**1**],**c**->**argv**[**2**]);**

**if** **(**errno **!=** ETIMEDOUT **&&** retry\_num**++** **==** 0**)** **goto** try\_again**;**

addReplySds**(**c**,**

sdsnew**(**"-IOERR error or timeout reading from target node\r\n"**));**

**return;**

**}**

首先检查最后一个命令参数，如果该参数既不是COPY，也不是REPLACE，则直接回复给客户端语法错误信息；然后从命令中解析出timeout和dbid，若解析错误，则直接回复给客户端错误信息。如果解析得到的timeout小于等于0，则将其置为1000，也就是1秒；

然后从客户端当前连接的数据库中，查找key，得到其值对象o。如果找不到key，则回复给客户端"+NOKEY"，这不算是错误，因为可能该key刚好超时被删除了；

接下来，根据参数中的host和port，调用migrateGetSocket函数，得到与远端Redis的连接。如果之前已经与该Redis建链了，则该函数会返回之前缓存的连接，否则，直接向该Redis发起TCP同步建链，建链超时时间为timeout。如果建链失败，则在migrateGetSocket中回复给客户端错误信息后，直接返回；

接下来，开始构建要发送给远端Redis的RESTORE命令：首先初始化rio结构的cmd，该结构中记录要发送的命令；如果命令参数中的dbid，与上次迁移时的dbid不同，则需要首先向cmd中填充"SELECT <dbid>"命令；然后取得该key的超时时间expireat，将其转换为相对时间ttl；如果当前处于集群模式下，则向cmd中填充"RESTORE-ASKING"命令，否则填充"RESTORE"命令；然后向cmd中填充key，以及ttl；然后调用createDumpPayload函数，将值对象o，按照DUMP的格式填充到payload中，然后再将payload填充到cmd中；如果最后一个命令参数是REPLACE，则还需要填充"REPLACE"到cmd中；

接下来，开始向远端Redis发送命令：循环调用syncWrite函数，向远端Redis同步发送cmd中的内容，每次最多发送64k个字节；

发送完成后，开始读取远端Redis的回复：如果之前发送了"SELECT"命令，则首先读取"SELECT"命令的回复到buf1中；然后读取"RESTORE"命令的回复到buf2中。读取超时时间为timeout；

如果buf1或buf2首字符为'-'，说明远端Redis回复了错误信息，则先设置cs->last\_dbid为-1，这样下次迁移时会强制发送"SELECT"命令，然后回复错误信息给客户端；否则，说明迁移成功了，先设置cs->last\_dbid为dbid，然后回复客户端"OK"信息。

如果客户端命令最后一个参数不是"COPY"，则先将该key从数据库中删除，然后调用rewriteClientCommandVector函数，将当前客户端的命令修改为"DEL <key>"，这样接下来在propagate函数中，会将该DEL命令传递给AOF文件或从节点；

如果写命令或者读回复发生错误，则调用migrateCloseSocket关闭与远端Redis的连接，如果不是超时错误的话，则重试一次，否则回复给客户端相应的错误信息；

注意：为了避免出现竞争条件（同一个key出现在两个节点中），在本函数中，涉及到向其他Redis服务器建链、发送命令和等待回复的过程，以上过程都是同步的，因此如果网络异常，并且超时时间又设置的比较大，则该函数有可能会阻塞Redis对于其他事件的处理，导致其他客户端无法操作当前Redis服务器（亲测）!!!

4.3、RESTORE-ASKING（或RESTORE）命令

key迁移的目的节点收到源节点发来的RESTORE-ASKING或RESTORE命令后，将命令中的key和value保存到本地数据库中。命令格式是："RESTORE <key> <ttl> <serialized-value> [REPLACE]"或" RESTORE-ASKING <key> <ttl> <serialized-value> [REPLACE]"

这两个命令的区别是：RESTORE-ASKING命令用于集群节点间的key迁移，RESTORE命令用于普通节点间的key迁移。RESTORE-ASKING命令对应的redisCommand结构标志位中带有'k'标记，这样在键迁移时，就不会返回ASK重定向错误；

这两个命令都通过调用restoreCommand函数处理。该函数的代码如下：

void restoreCommand**(**redisClient **\***c**)** **{**

long long ttl**;**

rio payload**;**

int j**,** type**,** replace **=** 0**;**

robj **\***obj**;**

*/\* Parse additional options \*/*

**for** **(**j **=** 4**;** j **<** c**->**argc**;** j**++)** **{**

**if** **(!**strcasecmp**(**c**->**argv**[**j**]->**ptr**,**"replace"**))** **{**

replace **=** 1**;**

**}** **else** **{**

addReply**(**c**,**shared**.**syntaxerr**);**

**return;**

**}**

**}**

*/\* Make sure this key does not already exist here... \*/*

**if** **(!**replace **&&** lookupKeyWrite**(**c**->**db**,**c**->**argv**[**1**])** **!=** **NULL)** **{**

addReply**(**c**,**shared**.**busykeyerr**);**

**return;**

**}**

*/\* Check if the TTL value makes sense \*/*

**if** **(**getLongLongFromObjectOrReply**(**c**,**c**->**argv**[**2**],&**ttl**,NULL)** **!=** REDIS\_OK**)** **{**

**return;**

**}** **else** **if** **(**ttl **<** 0**)** **{**

addReplyError**(**c**,**"Invalid TTL value, must be >= 0"**);**

**return;**

**}**

*/\* Verify RDB version and data checksum. \*/*

**if** **(**verifyDumpPayload**(**c**->**argv**[**3**]->**ptr**,**sdslen**(**c**->**argv**[**3**]->**ptr**))** **==** REDIS\_ERR**)**

**{**

addReplyError**(**c**,**"DUMP payload version or checksum are wrong"**);**

**return;**

**}**

rioInitWithBuffer**(&**payload**,**c**->**argv**[**3**]->**ptr**);**

**if** **(((**type **=** rdbLoadObjectType**(&**payload**))** **==** **-**1**)** **||**

**((**obj **=** rdbLoadObject**(**type**,&**payload**))** **==** **NULL))**

**{**

addReplyError**(**c**,**"Bad data format"**);**

**return;**

**}**

*/\* Remove the old key if needed. \*/*

**if** **(**replace**)** dbDelete**(**c**->**db**,**c**->**argv**[**1**]);**

*/\* Create the key and set the TTL if any \*/*

dbAdd**(**c**->**db**,**c**->**argv**[**1**],**obj**);**

**if** **(**ttl**)** setExpire**(**c**->**db**,**c**->**argv**[**1**],**mstime**()+**ttl**);**

signalModifiedKey**(**c**->**db**,**c**->**argv**[**1**]);**

addReply**(**c**,**shared**.**ok**);**

server**.**dirty**++;**

**}**

首先，解析命令中第四个参数是否为"REPLACE"，若是则置replace为1，否则，直接回复客户端语法错误信息；

如果replace为1，则从数据库中查找相应的key，如果查不到，则直接回复客户端错误信息；

然后从命令中解析ttl参数，如果解析错误，或者解析出的ttl小于0，则直接回复客户端错误信息；

然后调用verifyDumpPayload函数，验证远端Redis发来的命令参数中，DUMP格式的值对象参数中的验证码是否正确，验证失败则回复客户端错误信息；

接下来，从命令参数中解析出值对象的类型和值对象本身，将值对象保存在obj中，如果解析错误，则回复客户端错误信息；

如果replace为1，则将该key从数据库中删除；然后将key和obj添加到数据库中；

如果ttl不为0，则设置该key的超时时间；最后，回复客户端"OK"信息；

以上，就完成了一个key的迁移过程。

5：向所有节点发送”CLUSTER SETSLOT  <slot>  NODE  <nodeid>”命令

当槽位中的所有key都迁移完成之后，需要向集群中所有节点，包括迁移的源节点以及目的节点，发送”CLUSTER SETSLOT  <slot>  NODE  <nodeid>”命令，以便通知所有节点，更新槽位<slot> 新的负责节点为<nodeid>。

在函数clusterCommand中，处理该命令的代码如下：

**else** **if** **(!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"setslot"**)** **&&** c**->**argc **>=** 4**)** **{**

*/\* SETSLOT 10 MIGRATING <node ID> \*/*

*/\* SETSLOT 10 IMPORTING <node ID> \*/*

*/\* SETSLOT 10 STABLE \*/*

*/\* SETSLOT 10 NODE <node ID> \*/*

int slot**;**

clusterNode **\***n**;**

**if** **((**slot **=** getSlotOrReply**(**c**,**c**->**argv**[**2**]))** **==** **-**1**)** **return;**

**if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"migrating"**)** **&&** c**->**argc **==** 5**)** **{**

**...**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"importing"**)** **&&** c**->**argc **==** 5**)** **{**

**...**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"stable"**)** **&&** c**->**argc **==** 4**)** **{**

**...**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**3**]->**ptr**,**"node"**)** **&&** c**->**argc **==** 5**)** **{**

*/\* CLUSTER SETSLOT <SLOT> NODE <NODE ID> \*/*

clusterNode **\***n **=** clusterLookupNode**(**c**->**argv**[**4**]->**ptr**);**

**if** **(!**n**)** **{**

addReplyErrorFormat**(**c**,**"Unknown node %s"**,**

**(**char**\*)**c**->**argv**[**4**]->**ptr**);**

**return;**

**}**

*/\* If this hash slot was served by 'myself' before to switch*

*\* make sure there are no longer local keys for this hash slot. \*/*

**if** **(**server**.**cluster**->**slots**[**slot**]** **==** myself **&&** n **!=** myself**)** **{**

**if** **(**countKeysInSlot**(**slot**)** **!=** 0**)** **{**

addReplyErrorFormat**(**c**,**

"Can't assign hashslot %d to a different node "

"while I still hold keys for this hash slot."**,** slot**);**

**return;**

**}**

**}**

*/\* If this slot is in migrating status but we have no keys*

*\* for it assigning the slot to another node will clear*

*\* the migratig status. \*/*

**if** **(**countKeysInSlot**(**slot**)** **==** 0 **&&**

server**.**cluster**->**migrating\_slots\_to**[**slot**])**

server**.**cluster**->**migrating\_slots\_to**[**slot**]** **=** **NULL;**

*/\* If this node was importing this slot, assigning the slot to*

*\* itself also clears the importing status. \*/*

**if** **(**n **==** myself **&&**

server**.**cluster**->**importing\_slots\_from**[**slot**])**

**{**

*/\* This slot was manually migrated, set this node configEpoch*

*\* to a new epoch so that the new version can be propagated*

*\* by the cluster.*

*\**

*\* Note that if this ever results in a collision with another*

*\* node getting the same configEpoch, for example because a*

*\* failover happens at the same time we close the slot, the*

*\* configEpoch collision resolution will fix it assigning*

*\* a different epoch to each node. \*/*

**if** **(**clusterBumpConfigEpochWithoutConsensus**()** **==** REDIS\_OK**)** **{**

redisLog**(**REDIS\_WARNING**,**

"configEpoch updated after importing slot %d"**,** slot**);**

**}**

server**.**cluster**->**importing\_slots\_from**[**slot**]** **=** **NULL;**

**}**

clusterDelSlot**(**slot**);**

clusterAddSlot**(**n**,**slot**);**

**}** **else** **{**

addReplyError**(**c**,**

"Invalid CLUSTER SETSLOT action or number of arguments"**);**

**return;**

**}**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**|**CLUSTER\_TODO\_UPDATE\_STATE**);**

addReply**(**c**,**shared**.**ok**);**

**}**

如果收到的是"CLUSTER SETSLOT <SLOT> NODE <node ID>"命令，说明需要更新负责相应槽位的节点。

首先根据参数<node ID>在字典server.cluster->nodes中查询新的负责该槽位的节点n，若找不到，则回复客户端错误信息后返回；

如果目前负责该槽位的节点为当前节点myself，并且myself不等于n，说明当前节点正在将该槽位迁出到节点n中，调用countKeysInSlot函数计算该槽位中尚存多少个key，如果该函数返回值不为0，说明该槽位中还有未迁出的key，因此回复客户端错误信息后返回；

如果当前节点正在迁出该槽位，并且该槽位中所有的key都已经迁出，则置server.cluster->migrating\_slots\_to[slot]为NULL；

如果当前节点正在迁入该槽位，并且n就是myself，则首先调用函数clusterBumpConfigEpochWithoutConsensus增加纪元configEpoch的值，然后置server.cluster->importing\_slots\_from[slot]为NULL；

最后，调用clusterDelSlot清空该slot相关的信息，然后调用clusterAddSlot，将该槽位的负责人改为节点n；

至此，就完成了一次槽位迁移(重新分片)流程。

**四：集群节点执行命令**

在集群模式下，数据库的key分布在多个集群节点中。因此当某个集群节点收到客户端的命令时，与普通模式下稍有不同。这不同主要体现在：

a：若命令中涉及到多个key，而这些key处于不同的槽位中，则该命令不能被执行，直接返回错误；

b：某个集群节点收到客户端发来的命令后，会判断命令中的key是否由本节点负责，若是，则直接处理命令；若不是，则反馈给客户端MOVED重定向错误，错误中指明了该key真正的负责节点。客户端收到MOVED重定向错误之后，需要重新向真正的负责节点再次发送命令；

c：如果节点A正在迁出槽位，此时收到了客户端的命令，而命令中的key已经迁入到了B节点，则节点A返回给客户端ASK重定向错误，该错误中指明了该key的迁入目的地节点。客户端收到ASK错误之后，需要先向B节点发送”ASKING”命令，然后在向B节点发送该命令。

ASK错误和MOVED错误都会导致客户端转向，它们的区别在于：

a：MOVED错误代表槽位的负责权已经从一个节点转移到了另一个节点：在客户端收到 关于槽位i的MOVED错误之后，会更新槽位i及其负责节点的对应关系，这样下次遇到关于槽位i的命令请求时，就可以直接将命令请求发送新的负责节点。

b：ASK错误只是两个节点在迁移槽的过程中使用的一种临时措施：客户端收到关于槽位i的ASK错误之后，客户端只会在接下来的一次命令请求中将关于槽位i的命令请求发送至ASK错误所指示的节点，但这种重定向不会对客户端今后发送关于槽位i的命令请求产生任何影响，客户端之后仍然会将关于槽位i的命令请求发送至目前负责处理该槽位的节点，除非ASK错误再次出现。

在处理客户端命令的函数processCommand中，如果Redis服务器处于集群模式下，在实际执行命令处理函数之前，需要判断当前节点是否能处理该命令中的key，若本节点不能处理该命令，则回复给客户端重定向错误，表示该命令应由其他集群节点处理。

以下情况下，可以无需判断命令，本节点可以直接处理该命令：

a：本节点为从节点，该命令是主节点发来的消息；

b：该命令中不包含key；

c：LUA客户端发来的命令；

processCommand中的这部分代码如下：

/\* If cluster is enabled perform the cluster redirection here.

\* However we don't perform the redirection if:

\* 1) The sender of this command is our master.

\* 2) The command has no key arguments. \*/

**if** **(**server**.**cluster\_enabled **&&**

**!(**c**->**flags **&** REDIS\_MASTER**)** **&&**

**!(**c**->**flags **&** REDIS\_LUA\_CLIENT **&&**

server**.**lua\_caller**->**flags **&** REDIS\_MASTER**)** **&&**

**!(**c**->**cmd**->**getkeys\_proc **==** **NULL** **&&** c**->**cmd**->**firstkey **==** 0**))**

**{**

int hashslot**;**

**if** **(**server**.**cluster**->**state **!=** REDIS\_CLUSTER\_OK**)** **{**

flagTransaction**(**c**);**

clusterRedirectClient**(**c**,NULL,**0**,**REDIS\_CLUSTER\_REDIR\_DOWN\_STATE**);**

**return** REDIS\_OK**;**

**}** **else** **{**

int error\_code**;**

clusterNode **\***n **=** getNodeByQuery**(**c**,**c**->**cmd**,**c**->**argv**,**c**->**argc**,&**hashslot**,&**error\_code**);**

**if** **(**n **==** **NULL** **||** n **!=** server**.**cluster**->**myself**)** **{**

flagTransaction**(**c**);**

clusterRedirectClient**(**c**,**n**,**hashslot**,**error\_code**);**

**return** REDIS\_OK**;**

**}**

**}**

**}**

判断本节点是否能执行该命令的步骤是：

如果当前集群的状态不是REDIS\_CLUSTER\_OK，则直接回复给客户端REDIS\_CLUSTER\_REDIR\_DOWN\_STATE错误，直接返回；

否则，调用getNodeByQuery函数，查询能够处理该命令的节点n，如果n为NULL，或者n不是当前节点，则直接回复给客户端相应的错误，直接返回；

其他情况，说明本节点可以处理该命令；

getNodeByQuery函数是集群模式下，判断当前节点是否能处理客户端命令的函数，本函数还会查找能够处理客户端命令的节点。该函数的代码如下：

clusterNode **\***getNodeByQuery**(**redisClient **\***c**,** struct redisCommand **\***cmd**,** robj **\*\***argv**,** int argc**,** int **\***hashslot**,** int **\***error\_code**)** **{**

clusterNode **\***n **=** **NULL;**

robj **\***firstkey **=** **NULL;**

int multiple\_keys **=** 0**;**

multiState **\***ms**,** \_ms**;**

multiCmd mc**;**

int i**,** slot **=** 0**,** migrating\_slot **=** 0**,** importing\_slot **=** 0**,** missing\_keys **=** 0**;**

*/\* Set error code optimistically for the base case. \*/*

**if** **(**error\_code**)** **\***error\_code **=** REDIS\_CLUSTER\_REDIR\_NONE**;**

*/\* We handle all the cases as if they were EXEC commands, so we have*

*\* a common code path for everything \*/*

**if** **(**cmd**->**proc **==** execCommand**)** **{**

*/\* If REDIS\_MULTI flag is not set EXEC is just going to return an*

*\* error. \*/*

**if** **(!(**c**->**flags **&** REDIS\_MULTI**))** **return** myself**;**

ms **=** **&**c**->**mstate**;**

**}** **else** **{**

*/\* In order to have a single codepath create a fake Multi State*

*\* structure if the client is not in MULTI/EXEC state, this way*

*\* we have a single codepath below. \*/*

ms **=** **&**\_ms**;**

\_ms**.**commands **=** **&**mc**;**

\_ms**.**count **=** 1**;**

mc**.**argv **=** argv**;**

mc**.**argc **=** argc**;**

mc**.**cmd **=** cmd**;**

**}**

*/\* Check that all the keys are in the same hash slot, and obtain this*

*\* slot and the node associated. \*/*

**for** **(**i **=** 0**;** i **<** ms**->**count**;** i**++)** **{**

struct redisCommand **\***mcmd**;**

robj **\*\***margv**;**

int margc**,** **\***keyindex**,** numkeys**,** j**;**

mcmd **=** ms**->**commands**[**i**].**cmd**;**

margc **=** ms**->**commands**[**i**].**argc**;**

margv **=** ms**->**commands**[**i**].**argv**;**

keyindex **=** getKeysFromCommand**(**mcmd**,**margv**,**margc**,&**numkeys**);**

**for** **(**j **=** 0**;** j **<** numkeys**;** j**++)** **{**

robj **\***thiskey **=** margv**[**keyindex**[**j**]];**

int thisslot **=** keyHashSlot**((**char**\*)**thiskey**->**ptr**,**

sdslen**(**thiskey**->**ptr**));**

**if** **(**firstkey **==** **NULL)** **{**

*/\* This is the first key we see. Check what is the slot*

*\* and node. \*/*

firstkey **=** thiskey**;**

slot **=** thisslot**;**

n **=** server**.**cluster**->**slots**[**slot**];**

*/\* Error: If a slot is not served, we are in "cluster down"*

*\* state. However the state is yet to be updated, so this was*

*\* not trapped earlier in processCommand(). Report the same*

*\* error to the client. \*/*

**if** **(**n **==** **NULL)** **{**

getKeysFreeResult**(**keyindex**);**

**if** **(**error\_code**)**

**\***error\_code **=** REDIS\_CLUSTER\_REDIR\_DOWN\_UNBOUND**;**

**return** **NULL;**

**}**

*/\* If we are migrating or importing this slot, we need to check*

*\* if we have all the keys in the request (the only way we*

*\* can safely serve the request, otherwise we return a TRYAGAIN*

*\* error). To do so we set the importing/migrating state and*

*\* increment a counter for every missing key. \*/*

**if** **(**n **==** myself **&&**

server**.**cluster**->**migrating\_slots\_to**[**slot**]** **!=** **NULL)**

**{**

migrating\_slot **=** 1**;**

**}** **else** **if** **(**server**.**cluster**->**importing\_slots\_from**[**slot**]** **!=** **NULL)** **{**

importing\_slot **=** 1**;**

**}**

**}** **else** **{**

*/\* If it is not the first key, make sure it is exactly*

*\* the same key as the first we saw. \*/*

**if** **(!**equalStringObjects**(**firstkey**,**thiskey**))** **{**

**if** **(**slot **!=** thisslot**)** **{**

*/\* Error: multiple keys from different slots. \*/*

getKeysFreeResult**(**keyindex**);**

**if** **(**error\_code**)**

**\***error\_code **=** REDIS\_CLUSTER\_REDIR\_CROSS\_SLOT**;**

**return** **NULL;**

**}** **else** **{**

*/\* Flag this request as one with multiple different*

*\* keys. \*/*

multiple\_keys **=** 1**;**

**}**

**}**

**}**

*/\* Migarting / Improrting slot? Count keys we don't have. \*/*

**if** **((**migrating\_slot **||** importing\_slot**)** **&&**

lookupKeyRead**(&**server**.**db**[**0**],**thiskey**)** **==** **NULL)**

**{**

missing\_keys**++;**

**}**

**}**

getKeysFreeResult**(**keyindex**);**

**}**

*/\* No key at all in command? then we can serve the request*

*\* without redirections or errors. \*/*

**if** **(**n **==** **NULL)** **return** myself**;**

*/\* Return the hashslot by reference. \*/*

**if** **(**hashslot**)** **\***hashslot **=** slot**;**

*/\* This request is about a slot we are migrating into another instance?*

*\* Then if we have all the keys. \*/*

*/\* If we don't have all the keys and we are migrating the slot, send*

*\* an ASK redirection. \*/*

**if** **(**migrating\_slot **&&** missing\_keys**)** **{**

**if** **(**error\_code**)** **\***error\_code **=** REDIS\_CLUSTER\_REDIR\_ASK**;**

**return** server**.**cluster**->**migrating\_slots\_to**[**slot**];**

**}**

*/\* If we are receiving the slot, and the client correctly flagged the*

*\* request as "ASKING", we can serve the request. However if the request*

*\* involves multiple keys and we don't have them all, the only option is*

*\* to send a TRYAGAIN error. \*/*

**if** **(**importing\_slot **&&**

**(**c**->**flags **&** REDIS\_ASKING **||** cmd**->**flags **&** REDIS\_CMD\_ASKING**))**

**{**

**if** **(**multiple\_keys **&&** missing\_keys**)** **{**

**if** **(**error\_code**)** **\***error\_code **=** REDIS\_CLUSTER\_REDIR\_UNSTABLE**;**

**return** **NULL;**

**}** **else** **{**

**return** myself**;**

**}**

**}**

*/\* Handle the read-only client case reading from a slave: if this*

*\* node is a slave and the request is about an hash slot our master*

*\* is serving, we can reply without redirection. \*/*

**if** **(**c**->**flags **&** REDIS\_READONLY **&&**

cmd**->**flags **&** REDIS\_CMD\_READONLY **&&**

nodeIsSlave**(**myself**)** **&&**

myself**->**slaveof **==** n**)**

**{**

**return** myself**;**

**}**

*/\* Base case: just return the right node. However if this node is not*

*\* myself, set error\_code to MOVED since we need to issue a rediretion. \*/*

**if** **(**n **!=** myself **&&** error\_code**)** **\***error\_code **=** REDIS\_CLUSTER\_REDIR\_MOVED**;**

**return** n**;**

**}**

参数c、cmd、argv和argc表示客户端及其发来的命令；参数hashslot为出参，返回命令中key所属的槽位号；参数error\_code为出参，出错时设置为相应错误码，成功时设置为REDIS\_CLUSTER\_REDIR\_NONE。该函数返回能够处理该命令的节点，若返回NULL，说明该命令目前无法在集群中执行。

需要注意的是，如果当前处于事务模式下，则事务中的所有命令中的所有key，需要一起进行判断。对于非事务模式下的命令，也按照事务的方式进行处理，只不过本事务只包含当前一条命令；

首先，如果命令执行函数为execCommand，则说明当前处于事务模式下，并且本条命令是事务中的最后一条命令"EXEC"。事务模式下，在c->mstate中保存了事务中之前的所有命令，因此将ms指向c->mstate。如果客户端没有设置REDIS\_MULTI标志，则直接返回myself，表示当前节点能够处理该命令，但是实际上这种情况下，在命令处理函数execCommand中，会直接反馈给客户端"EXEC without MULTI"错误；

如果命令处理函数不是execCommand，则构造伪事务结构ms，其中只包含当前命令这一条命令；

接下来，针对ms中的每一条命令进行判断：调用getKeysFromCommand函数，从命令中得到所有key的索引，保存在数组keyindex中，以及key的个数numkeys；

接下来就循环处理本条命令中的所有key：

首先调用keyHashSlot函数，计算该key所属的槽位号thisslot；

如果该key是命令中的第一个key，则用firstkey记录该key，用slot记录该key所属的槽位号；然后从server.cluster->slots中取得负责该槽位的节点n，如果n为NULL，则说明该槽位没有节点负责，集群目前处于下线状态，因此设置error\_code为REDIS\_CLUSTER\_REDIR\_DOWN\_UNBOUND，并且返回NULL；如果节点n就是当前节点，并且当前节点正在迁出该槽位，则设置migrating\_slot为1；否则如果当前节点正在迁入该槽位，则设置importing\_slot为1；

如果该key不是命令中的第一个key，则只要该key与第一个key内容不同，就比较该key所属的槽位是否与第一个key的槽位一致，若不一致，则设置错误码为REDIS\_CLUSTER\_REDIR\_CROSS\_SLOT，并返回NULL；若一致，则置multiple\_keys为1；

如果当前节点正在迁入或者迁出该槽位，并且在0号数据库中找不到该key，则增加missing\_keys的值；

遍历完所有命令的所有key后，走到现在，能保证所有key都属于同一个槽位slot，该槽位由节点n负责处理。接下来接着进行判断：

如果n为NULL，说明所有命令中都不包含任何key，因此返回myself，表示当前节点可以处理该命令；

将slot保存到出参hashslot中；

如果当前节点正在迁出槽位，并且命令中的key有的已经不再当前节点中了，则设置错误码为REDIS\_CLUSTER\_REDIR\_ASK，并返回该槽位所迁出的目的地节点；

如果当前节点正在迁入槽位，并且客户端具有ASKING标记（客户端之前发来过”ASKING”命令）或者该命令本身就具有ASKING标记（”RESTORE-ASKING”命令），则只有在涉及多个key，并且有的key不在当前节点中的情况下，才设置错误码为REDIS\_CLUSTER\_REDIR\_UNSTABLE，并返回NULL；否则，返回当前节点；

以上两条判断条件，保证了当命令中只有一个key时，写（新增key）命令需直接写入到迁入节点中，读命令需在具有key的节点中读取；当涉及多个key时，写（新增key）命令既无法在迁出节点中执行，也无法在迁入节点中执行，读命令需在具有所有key的节点中读取；（亲测）

如果当前节点正好为n节点的从节点，而且客户端是只读客户端，并且该命令是只读命令，则返回当前节点；

其他情况下，如果当前节点不是n节点，则设置错误码为REDIS\_CLUSTER\_REDIR\_MOVED，并返回节点n。