

28-RedisCluster数据迁移会阻塞吗？

你好，我是蒋德钧。

上节课，我给你介绍了Redis Cluster节点处理命令的过程。现在你知道，在这个过程中，节点会调用**getNodeByQuery函数**检查访问的key所属的节点，如果收到命令的节点并不是key所属的节点，那么当前节点就会生成CLUSTER_REDIR_MOVED或者CLUSTER_REDIR_ASK的报错信息，并给客户端返回MOVED或ASK命令。

其实，这两个报错信息就对应了Redis Cluster的数据迁移。数据迁移是分布式存储集群经常会遇到的一个问题，当集群节点承担的负载压力不均衡时，或者有新节点加入或是已有节点下线时，那么，数据就需要在不同的节点间进行迁移。所以，**如何设计和实现数据迁移也是在集群开发过程中，我们需要考虑的地方。**

那么今天这节课，我就来介绍下Redis Cluster是如何实现数据迁移的。从源码层面掌握这部分内容，可以帮助你了解数据迁移对集群节点正常处理命令的影响，这样你就可以选择合适时机进行迁移。而且，掌握Redis的数据迁移实现，也能为你自己开发集群提供一个不错的参考示例。

好了，接下来，我们就先来看下和数据迁移相关的主要数据结构。这些数据结构比较重要，它们记录了数据迁移的状态信息。

记录数据迁移的数据结构

首先你要知道，Redis Cluster是先把键值对映射到哈希槽（slots）中，然后通过给不同集群节点分配slots这样的方法，来完成数据在集群节点间的分配的。关于这部分的知识，你也可以去看看第一季的[第9讲](#)。

那么，在源码实现层面，Redis Cluster的每个集群节点都对应了一个**clusterNode的结构体**（在[cluster.h](#)文件中）。这个结构体中包含了一个char类型的数组，用来记录当前节点在负责哪些slots。

这个数组的定义如下所示，它的长度是宏定义CLUSTER_SLOTS除以8，而CLUSTER_SLOTS宏定义的值是16384，表示的是Redis Cluster的slots总个数。这个值除以8之后，就意味着数组每个元素的每一位表示1个slot。如果数组元素某一位的值是1，那么就表明当前节点负责这一位对应的slot。

```
typedef struct clusterNode {  
    ...  
    unsigned char slots[CLUSTER_SLOTS/8]  
    ...  
}
```

但是，如果只是用clusterNodes中的slots数组，并不能记录数据迁入迁出的情况，所以，Redis Cluster针对整个集群设计了**clusterState结构体**（在cluster.h文件中）。这个结构体中包含了三个clusterNode类型的数组和一个rax类型的字典树。这三个数组的大小，都是集群slots的总个数16384，如下所示：

```
typedef struct clusterState {  
    ...  
    clusterNode *migrating_slots_to[CLUSTER_SLOTS];  
    clusterNode *importing_slots_from[CLUSTER_SLOTS];  
    ...  
}
```

```
clusterNode *slots[CLUSTER_SLOTS];
rax *slots_to_keys;
...
}
```

这几个结构主要是被用来记录数据迁入迁出的情况，它们的含义如下。

- **migrating_slots_to数组**：表示当前节点负责的slot正在迁往哪个节点。比如，migrating_slots_to[K] = node1，这就表示当前节点负责的slot K，正在迁往node1。
- **importing_slots_from数组**：表示当前节点正在从哪个节点迁入某个slot。比如，importing_slots_from[L] = node3，这就表示当前节点正从node3迁入slot L。
- **slots数组**：表示16384个slot分别是由哪个节点负责的。比如，slots[M] = node2，这就表示slot M是由node2负责的。
- **slots_to_keys字典树**：用来记录slot和key的对应关系，可以通过它快速找到slot上有哪些keys。

好了，知道了用来记录数据迁入迁出情况的数据结构之后，我们就来学习数据迁移的具体过程。

数据迁移过程的设计与实现

Redis Cluster迁移数据的整个过程可以分成五个大步骤，分别是：

- 标记迁入、迁出节点；
- 获取迁出的keys；
- 源节点实际迁移数据；
- 目的节点处理迁移数据；
- 标记迁移结果。

下面，我们就分别来看下这五个步骤的源码实现。

标记迁入、迁出节点

在Redis Cluster中迁移数据时，我们需要先使用CLUSTER SETSLOT命令，在待迁入数据的目的节点上标记待迁出数据的源节点，使用的命令如下所示：

```
CLUSTER SETSLOT <slot> IMPORTING <node> //<slot>表示要迁入的哈希槽，<node>表示当前负责<slot>的节点
```

然后，我们需要使用CLUSTER SETSLOT命令，在待迁出数据的源节点上标记将要迁入数据的目的节点，使用的命令如下所示：

```
CLUSTER SETSLOT <slot> MIGRATING <node> //<slot>表示要迁出的哈希槽，<node>表示<slot>要迁往的目的节点
```

为了便于你理解，我来举个例子。假设slot 3在节点A上，现在我们要把slot 3从节点A上迁移到节点B上，那么，此时节点A就是待迁出数据的源节点，而节点B就是待迁入数据的目的节点。我们要先在节点B上执行如下命令，用来标记源节点。

```
CLUSTER SETSLOT slot3 IMPORTING nodeA
```

然后，我们在节点A上执行如下命令，用来标记目的节点。

```
CLUSTER SETSLOT slot3 MIGRATING nodeB
```

对于CLUSTER命令来说，它的处理函数是**clusterCommand**（在[cluster.c](#)文件中）。在这个函数中，它会根据CLUSTER命令携带的不同选项，执行不同的代码分支。因此，对于刚才介绍的标记slot迁入、迁出的SETSLOT选项，它们在clusterCommand函数中对应的代码分支如下所示：

```
void clusterCommand(client *c) {
...
//处理SETSLOT选项
else if (!strcasecmp(c->argv[1]->ptr,"setslot") && c->argc >= 4) {
...
//处理migrating标记
if (!strcasecmp(c->argv[3]->ptr,"migrating") && c->argc == 5) {
...
} //处理importing标记
else if (!strcasecmp(c->argv[3]->ptr,"importing") && c->argc == 5) {
...
}
}
}
```

这里，我们来看一下处理migrating和importing标记的具体逻辑。其实，clusterCommand函数对这两个标记的处理逻辑基本都是分成三步。

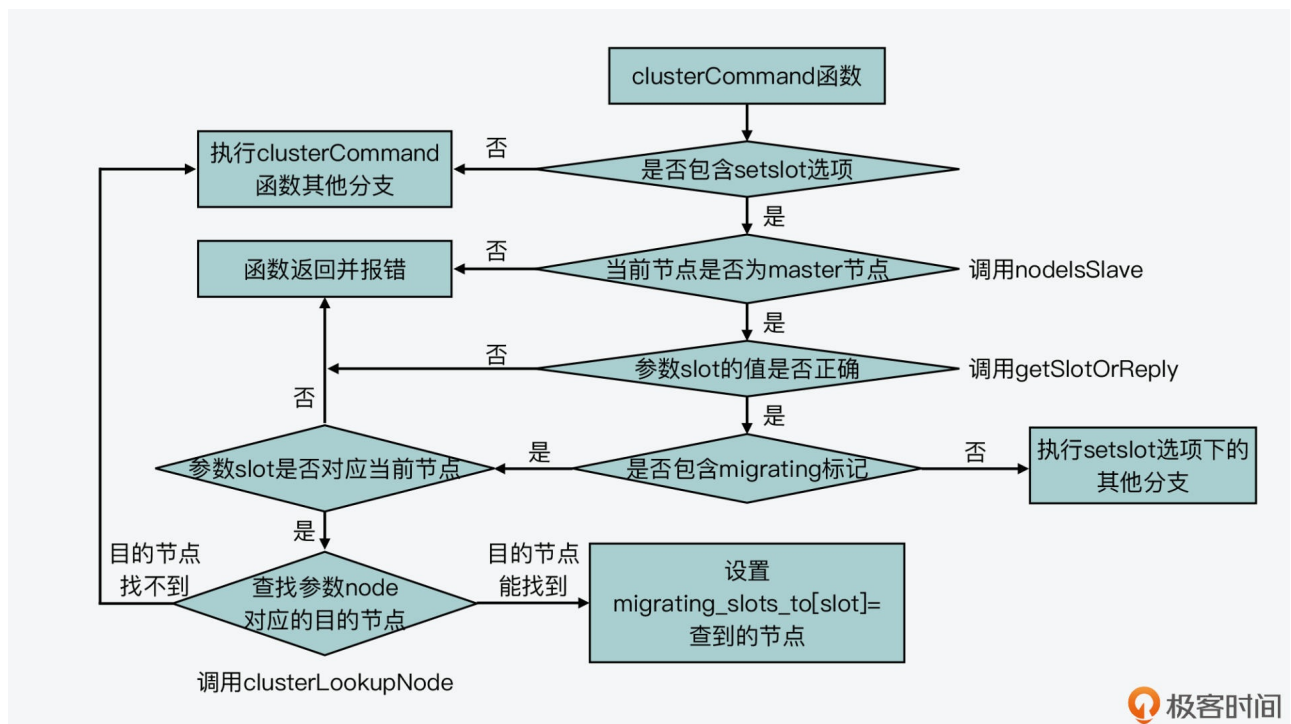
第一步，对于数据迁出来说，该函数会判断迁出的slot是否在当前节点；而对于数据迁入来说，该函数会判断迁入的slot是否在当前节点。如果迁出的slot不在当前节点，或者迁入的slot已在当前节点，那么clusterCommand函数就**返回报错信息**了。这是因为，在这两种情况下节点无法执行slot迁移。

第二步，如果迁出的slot在当前节点，或者迁入的slot不在当前节点，那么，clusterCommand函数就会**调用clusterLookupNode函数**（在cluster.c文件中），来查询CLUSTER SETSLOT命令中包含的。这主要是依赖于clusterLookupNode函数根据输入的节点ID，在全局变量server的cluster->nodes数组中，查找并返回对应节点。

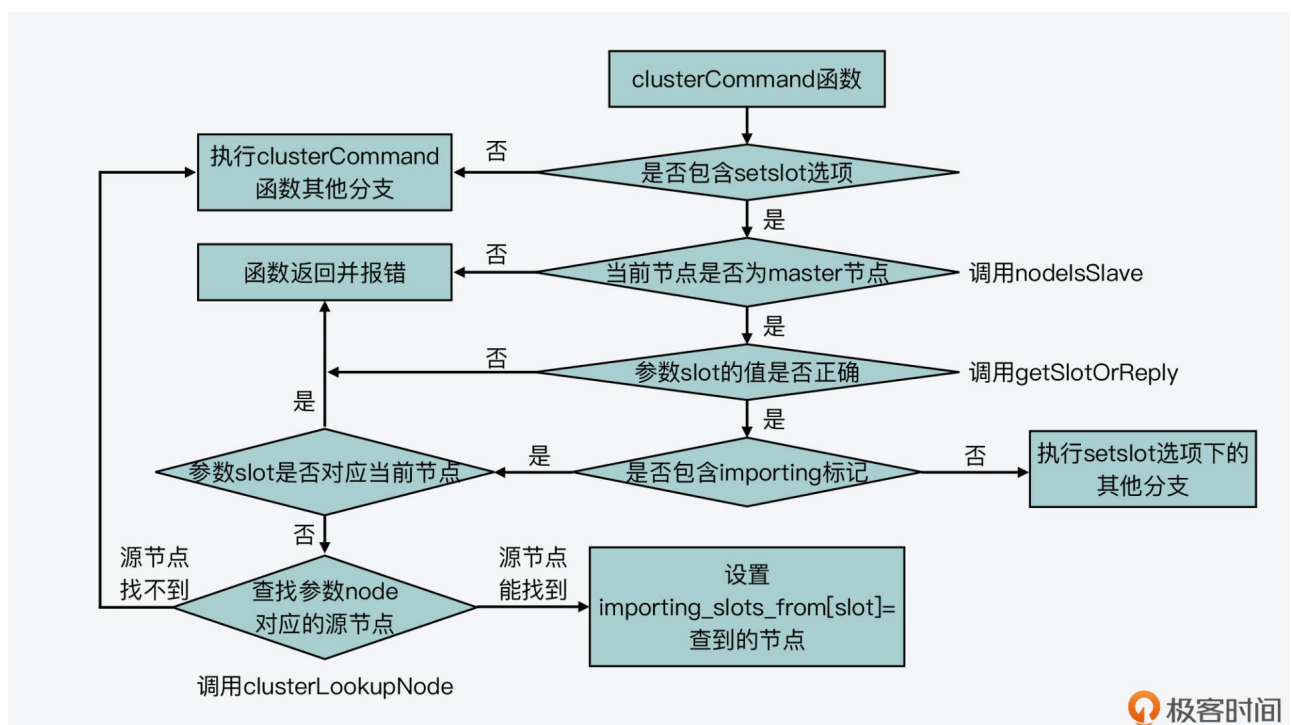
第三步，clusterCommand函数会把migrating_slots_to数组中迁出slot，或者importing_slots_from数组

中迁入slot对应的节点，设置为clusterLookupNode函数查找的结果。

我也画了两张图，分别展示了clusterCommand函数处理CLUSTER SETSLOT命令的migrating和importing标记的基本逻辑，你可以再看下。



极客时间



极客时间

这样一来，当在Redis Cluster中标记完迁入和迁出的节点后，我们就可以使用CLUSTER GETKEYSINSLOT命令，来获取要迁出的keys了。下面我们来看下这步操作的实现。

获取待迁出的keys

我们用来获取待迁出的keys的具体命令如下所示，其中表示要迁移的slot，而表示要迁移的key的数量。

```
CLUSTER GETKEYSINSLOT <slot> <count>
```

因为这里我们用的还是CLUSTER命令，所以，获取待迁出keys的命令处理也还是在**clusterCommand函数**中，对应了GETKEYSINSLOT选项的代码分支，如下所示：

```
void clusterCommand(client *c) {  
    ...  
    //处理GETKEYSINSLOT选项  
    else if (!strcasecmp(c->argv[1]->ptr,"getkeysinslot") && c->argc == 4) {...}  
    ...  
}
```

这个代码分支的处理逻辑也比较简单，它主要可以分成三步。

首先，这个代码分支会**调用getLongLongFromObjectOrReply函数**（在object.c文件中），从CLUSTER GETKEYSINSLOT命令中获取和参数，这里的参数会被赋值给maxkeys变量，如下所示：

```
//解析获取slot参数  
if (getLongLongFromObjectOrReply(c,c->argv[2],&slot,NULL) != C_OK)  
    return;  
//解析获取count参数，赋值给maxkeys  
if (getLongLongFromObjectOrReply(c,c->argv[3],&maxkeys,NULL) != C_OK)  
    return;
```

然后，clusterCommand函数会**调用countKeysInSlot函数**（在db.c文件中），获取待迁移slot中实际的key的数量。如果刚才从命令中获取的key的迁移数量maxkeys，大于实际的key数量，那么maxkeys的值会被更新为实际的key数量。紧接着，clusterCommand函数会给这些key分配空间。

```
unsigned int keys_in_slot = countKeysInSlot(slot); //获取迁移slot中实际的key数量  
if (maxkeys > keys_in_slot) maxkeys = keys_in_slot; //如果实际的key数量小于maxkeys，将maxkeys更新为实际的key数量  
keys = zmalloc(sizeof(robj)*maxkeys); //给key分配空间
```

最后，这个代码分支会**调用getKeysInSlot函数**（在db.c文件中），从迁移slot中获取实际的key，并将这些key返回给客户端，如下所示：

```
numkeys = getKeysInSlot(slot, keys, maxkeys); //获取实际的key  
addReplyMultiBulkLen(c,numkeys); //将key返回给客户端  
for (j = 0; j < numkeys; j++) {  
    addReplyBulk(c,keys[j]);  
    decrRefCount(keys[j]);  
}
```

好了，到这里，客户端就通过CLUSTER GETKEYSINSLOT命令，获得了一定数量的要迁移的key。接下来，我们就要开始执行实际的迁移操作了，我们来具体看下。

源节点实际迁移数据

在实际迁移数据时，我们需要在待迁出数据的源节点上**执行MIGRATE命令**。其实，MIGRATE命令既支持迁移单个key，也支持一次迁移多个key，它们的基本处理流程是相同的，都是在**migrateCommand函数**中实现的。

这里，我以一次迁移多个key的MIGRATE命令为例，这个命令的选项中包含了目的节点的IP、端口号、数据库编号，以及要迁移的多个key、迁移超时时间，它的格式如下所示：

```
MIGRATE host port "" dbid timeout [COPY | REPLACE] KEYS key1 key2 ... keyN
```

从这个命令中，你也可以看到，它还包括了COPY或REPLACE选项，这两个选项的含义如下。

- **COPY**：如果目的节点已经存在待迁移的key，则报错；如果目的节点不存在待迁移的key，那么就正常迁移，并在迁移后，删除源节点上的key。
- **REPLACE**：无论目的节点是否存在待迁移的key，都会正常执行迁移，并覆盖已经存在的key。

好，了解了MIGRATE命令的含义后，我们就来看下migrateCommand函数的基本处理流程，这个函数的执行过程主要可以分成四步。

第一步，命令参数检查

migrateCommand函数首先会检查MIGRATE命令携带的参数，比如是否有COPY或REPLACE标记、dbid和timeout是否能正常读取等。在这一步，migrateCommand函数如果检查到timeout值小于等于0了，它就会把timeout值设置为1000毫秒，用于迁移过程中的超时判断。

第二步，读取要迁移的key和value

检查完命令参数后，migrateCommand函数会分配两个数组ov和kv，它们的初始大小等于MIGRATE命令中要迁移的key的数量。然后，migrateCommand函数会调用lookupKeyRead函数（在db.c文件中），逐一检查要迁移的key是否存在。这是因为有的key在迁移时可能正好过期了，所以就不用迁移这些key了。这一步的最后，migrateCommand函数会根据实际存在的key数量，来设置要迁移的key数量。

下面的代码展示了这一步的基本逻辑，你可以看下。

```
ov = zrealloc(ov,sizeof(robj*)*num_keys); //分配ov数组，保存要迁移的value
kv = zrealloc(kv,sizeof(robj*)*num_keys); //分配kv数组，保存要迁移的key
...
for (j = 0; j < num_keys; j++) {
    //逐一检查要迁移的key是否存在
    if ((ov[oi] = lookupKeyRead(c->db,c->argv[first_key+j])) != NULL) {
        kv[oi] = c->argv[first_key+j]; //只记录存在的key
    }
}
```

```
        oi++;
    }
}
num_keys = oi; //要迁移的key数量等于实际存在的key数量
```

第三步，填充迁移用的命令、key和value

接下来，migrateCommand函数就开始为迁移数据做准备了。这一步骤中的操作主要包括：

- 调用migrateGetSocket函数（在cluster.c文件中），和目的节点建立连接；
- 调用rioInitWithBuffer函数初始化一块缓冲区，然后调用rioWriteBulkString、rioWriteBulkLongLong等函数（在[rio.c](#)文件中），往这个缓冲区中填充要发送给目的节点的命令、key和value。

下面的代码也展示了在这一步中主要填充的命令、key和value，你可以看下。

```
rioInitWithBuffer(&cmd,sdsempty()); //初始化buffer
... //往buffer中填充SELECT命令
//针对每一个要迁移的key，往buffer中填充命令、key和value
for (j = 0; j < num_keys; j++) {
    //在集群模式下，填充RESTORE-ASKING命令，用来发给目的节点
    if (server.cluster_enabled)
        serverAssertWithInfo(c,NULL, rioWriteBulkString(&cmd,"RESTORE-ASKING",14));
    ...
    //填充key
    serverAssertWithInfo(c,NULL,rioWriteBulkString(&cmd,kv[j]->ptr,
        sdslen(kv[j]->ptr)));
    //填充TTL
    serverAssertWithInfo(c,NULL,rioWriteBulkLongLong(&cmd,ttl));
    //调用createDumpPayload函数序列化value
    createDumpPayload(&payload,ov[j],kv[j]);
    //填充value
    serverAssertWithInfo(c,NULL, rioWriteBulkString(&cmd,payload.io.buffer.ptr,
        ...
    }
}
```

这里，你需要注意的是，migrateCommand函数会调用createDumpPayload函数（在cluster.c文件中）将迁移key的value序列化，以便于传输。在序列化的结果中，**createDumpPayload函数会增加RDB版本号**

和CRC校验和。等目的节点收到迁移数据后，也会检查这两部分内容，我稍后还会给你介绍。当在缓冲区中填充完要发送给目的节点的命令、key和value后，migrateCommand函数就开始发送这个缓冲区中的内容了。

第四步，发送迁移用的命令和数据，并读取返回结果

migrateCommand函数会调用syncWrite函数（在[syncio.c](#)文件中），把缓冲区中的内容按照64KB的粒度发送给目的节点，如下所示：

```
while ((towrite = sdslen(buf)-pos) > 0) {
    towrite = (towrite > (64*1024) ? (64*1024) : towrite);
```



```

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

```

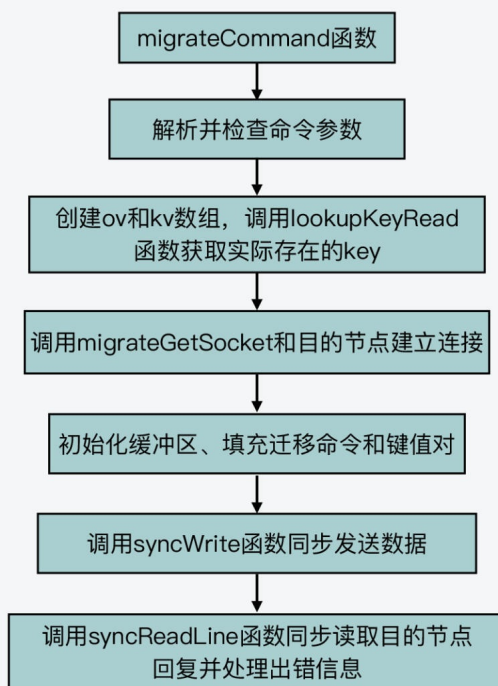
然后，针对发送给目的节点的每个键值对，migrateCommand函数会调用syncReadLine函数（在syncio.c文件中），读取目的节点的返回结果。如果返回结果中有报错信息，那么它就会进行相应的处理。这部分的逻辑并不复杂，但是针对各种出错情况的处理会比较多，你可以进一步阅读源码来进行学习。

```

//针对迁移的每个键值对，调用syncReadLine函数读取目的节点返回结果
for (j = 0; j < num_keys; j++) {
    if (syncReadLine(cs->fd, buf2, sizeof(buf2), timeout) <= 0) { ...}
    ... //处理目的节点返回的结果
}

```

好了，到这里，你就了解了MIGRATE命令的执行基本过程，我把它执行过程的四大步骤也画在了下面的这张图中，你可以再回顾下。



极客时间

其实在迁移数据的过程中，**目的节点对迁移命令的处理也是迁移过程的一个重要环节**。所以，下面我们就来看下，目的节点在收到RESTORE-ASKING命令后的处理过程。

目的节点处理迁移数据

目的节点在收到源节点发送的RESTORE-ASKING命令后，这个命令的实际处理函数是**restoreCommand**（在cluster.c文件中）。这个函数的处理逻辑并不复杂，主要可以分成三步。

首先，它会解析收到的命令参数，包括是否覆盖数据的标记replace、key过期时间标记ttl、key的LRU标记idleTime、key的LFU标记freq。接着，它会根据这些标记执行一系列检查。

这其中就包括，如果检测到没有replace标记的话，它会调用lookupKeyWrite函数（在db.c文件中），检查目的节点数据库中是否有迁移的key，如果已经存在待迁移key的话，它就会返回报错信息，如下所示。此外，它还会检查TTL值是否小于0。

```
//如果没有replace标记，并且数据库中存在待迁移的key
if (!replace && lookupKeyWrite(c->db,c->argv[1]) != NULL) {
    addReply(c,shared.busykeyerr); //返回报错信息
    return;
}
```

然后，restoreCommand函数会检查迁移key的value的序列化结果，就像我刚才介绍的，migrateCommand函数在实际迁移value时，会把value序列化后再传输。而序列化后的结果中包含了RDB版本和CRC校验和，restoreCommand函数会**调用verifyDumpPayload函数**（在cluster.c文件中），检测RDB版本和CRC校验和。如果这两部分内容不正确，它就会返回报错信息。

```
//检查value序列化结果中的RDB版本和CRC校验和
if (verifyDumpPayload(c->argv[3]->ptr,sdslen(c->argv[3]->ptr)) == C_ERR)
{
    addReplyError(c,"DUMP payload version or checksum are wrong");
    return;
}
```

紧接着，restoreCommand函数会调用rdbLoadObjectType函数和rdbLoadObject函数（在[rdb.c](#)文件中），从序列化结果中解析出实际的value类型和value实际值。

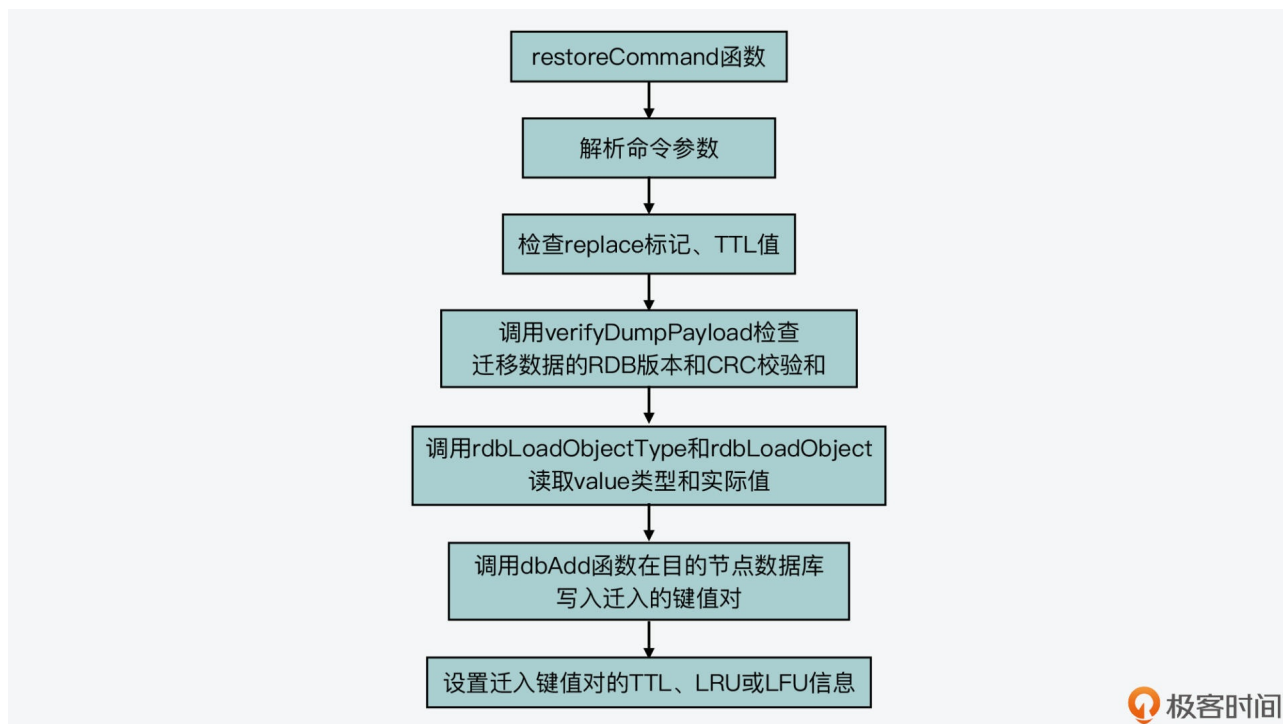
最后，restoreCommand函数会调用dbAdd函数，把解析得到key和value写入目的节点的数据库中。这里，你要注意的是，**如果迁移命令中带有REPLACE标记**，那么，restoreCommand函数会先调用dbDelete函数，删除在目的节点数据库中已经存在的迁移key，然后再调用dbAdd函数写入迁移key。此外，restoreCommand函数还会设置迁移key的过期时间，以及LRU或LFU信息，并最终返回成功信息。

下面的代码展示了restoreCommand函数最后一步的处理逻辑，你可以看下。

```
//如果有REPLACE标记，在目的节点数据库中删除已存在的迁移key
if (replace) dbDelete(c->db,c->argv[1]);

//将迁移key及value写入目的节点数据库
dbAdd(c->db,c->argv[1],obj);
if (ttl) { //设置TTL时间
    if (!absttl) ttl+=mstime();
    setExpire(c,c->db,c->argv[1],ttl);
}
objectSetLRUorLFU(obj,lfu_freq,lru_idle,lru_clock); //设置LRU或LFU信息
...
addReply(c,shared.ok); //返回成功信息
```

我在这里也画了一张图，展示了目的节点处理迁移数据的基本过程，你可以再整体看下。



好了，到这里，你就了解了源节点发送迁移数据，以及目的节点接收迁移数据的基本过程实现了。最后，当迁移slot中的key全部完成迁移后，我们还需要执行CLUSTER SETSLOT命令，来标记迁移的最终结果，下面我们来看下。

标记迁移结果

在数据迁移完成后，我们需要先在目的节点上**执行CLUSTER SETSLOT命令**，向目的节点标记迁移slot的最终所属节点，如下所示。然后，我们需要在源节点上执行相同的命令，用来向源节点标记迁移slot的最终所属节点。

```
CLUSTER SETSLOT <slot> NODE <node>
```

因为这个命令还是CLUSTER命令，所以它的处理仍然在**clusterCommand函数**中实现的。这个命令的选项是SETSLOT，并带有NODE标记，所以它对应的代码分支如下所示：

```
void clusterCommand(client *c) {
    ...
    //处理SETSLOT选项
    else if (!strcasecmp(c->argv[1]->ptr,"setslot") && c->argc >= 4) {
        ...
        //处理NODE标记
        else if (!strcasecmp(c->argv[3]->ptr,"node") && c->argc == 5) { ...}
        ...
    }
    ...
}
```

在刚才介绍的处理NODE标记的代码分支中，主要的工作是清除节点上migrating_slots_to数组和importing_slots_from数组中的标记。

对于migrating_slots_to数组来说，在源节点上，这个数组中迁移slot所对应的元素，记录了目的节点。那么，在源节点上执行迁移结果标记命令时，处理NODE标记的代码分支，就会调用**countKeysInSlot函数**（在db.c文件中）检查迁移slot中是否还有key。如果没有key了，那么migrating_slots_to数组中迁移slot所对应的元素会被置为NULL，也就是取消了源节点上的迁出标记。

```
if (countKeysInSlot(slot) == 0 && server.cluster->migrating_slots_to[slot]) //如果有迁出标记， 并且迁移slot中已无key
    server.cluster->migrating_slots_to[slot] = NULL; //将迁出标记置为NULL
```

而对于importing_slots_from数组来说，在目的节点上，这个数组中迁移slot所对应的元素记录了源节点。那么，在目的节点上执行迁移结果标记命令时，处理NODE标记的代码分支会**检查命令参数中的节点是否是目的节点自身**。如果是的话，importing_slots_from数组中迁移slot所对应的元素会被置为NULL，这就是取消了目的节点上的迁入标记。

```
//如果命令参数中的节点是当前节点，并且有迁入标记
if (n == myself && server.cluster->importing_slots_from[slot]) {
    ...
    server.cluster->importing_slots_from[slot] = NULL; //取消迁入标记
}
```

最后，处理NODE标记的代码分支，会调用clusterDelSlot和clusterAddSlot函数（在cluster.c文件中），分别更新slot迁移前和迁移后所属节点的slots数组，你可以去进一步阅读这两个函数的代码进行了解。

到这里，Redis Cluster中数据迁移的整个过程也就完成了。

小结

在今天的课程中，我给你介绍了Redis Cluster数据迁移过程的代码实现，你要掌握以下两个要点。

首先是记录集群状态的数据结构clusterState。这个结构中是使用了migrating_slots_to和importing_slots_from两个数组，来记录数据迁出迁入情况，使用了slots数组记录每个slot所属的节点，以及使用slots_to_keys字典树记录slots中的keys。你需要掌握这几个数据结构的含义，因为在你阅读集群源码时，这几个结构是会频繁使用到的。

然后是数据迁移过程的五大步骤。分别是：

- 标记迁入、迁出节点；
- 获取待迁出的keys；
- 源节点实际迁移数据；
- 目的节点处理迁移数据；

- 标记迁移结果。

这五个步骤对应了CLUSTER命令的不同选项、MIGRATE命令以及RESTORE命令，所以，它们的实现逻辑就主要对应在clusterCommand、migrateCommand和restoreCommand函数中。如果你了解数据迁移的更多细节，你可以从这几个函数入手进一步学习。

最后，我也想再**提醒你两个关键点**。

一是，Redis Cluster在执行数据迁移时，会调用syncWrite和syncReadLine函数，向目的节点同步发送迁移数据，以及同步读取回复结果。而这个同步写和同步读的过程，会阻塞源节点正常处理请求。所以，你在迁移数据时要**控制迁移的key数量和key大小**，避免一次性迁移过多的key或是过大的key，而导致Redis阻塞。

二是，我们在实际应用中，会用到**redis-cli工具**，或者是Ruby开发的Redis Cluster集群**运维工具redis-trib**，来执行数据迁移。这些工具最终也会调用这节课中，我们介绍的命令来完成数据的实际迁移。所以，学习今天课程的内容，对于你在实际应用中，从代码层面排查redis-cli、redis-trib这些工具的问题也是有所帮助的。

每课一问

在维护Redis Cluster集群状态的数据结构clusterState中，有一个字典树slots_to_keys。当在数据库中插入key时它会被更新，你能在Redis源码文件db.c中，找到更新slots_to_keys字典树的相关函数调用吗？

精选留言：

- 曾轼麟 2021-10-14 11:08:44

回答老师的问题：

clusterState 中的 slots_to_keys 在查阅git历史提交记录中发现，之前是使用跳跃表（zskiplist）在后来才替换成了-字典树，其最主要的目的是为了通过 slot 快速查找到slot下的keys，getKeysInSlot 函数正是使用这种方式来获取待迁出的 keys。

【获取key调用路径】:clusterCommand -> getKeysInSlot -> raxStart(迭代器)

【插入key调用路径】:dbAdd -> slotToKeyAdd -> slotToKeyUpdateKey -> raxInsert

【删除key调用路径】:dbAsyncDelete/dbSyncDelete -> slotToKeyDel -> slotToKeyUpdateKey -> raxRemove

总结：

本篇文章老师介绍了Redis Cluster 数据迁移过程的实现思路，其中数据迁移整体可以归纳为5个步骤：

- 1、标记迁入、迁出节点；
- 2、获取待迁出的 keys；
- 3、源节点实际迁移数据；
- 4、目的节点处理迁移数据；
- 5、标记迁移结果；

其中迁移涉及的函数有clusterCommand，migrateCommand，restoreCommand，其中老师有个点提醒的很好，syncWrite 和 syncReadLine是进行同步调用的，需要对key数量进行控制。

此外Redis为了保证传输效率和传输的安全性，分别使用了自定义的序列化方案（这里复用了DUMP命令

，文件序列化的方案） 和 CRC64验证其中结构大致如下所示：



理论上来说RESP协议也能实现类似的效果，但是在传输效率上来说自定义的序列化方案效率更高。