答疑5-第25~32讲课后思考题答案及常见问题答疑

你好，我是蒋德钧。今天这节课，我们来继续解答第25讲到32讲的课后思考题。

今天讲解的这些思考题，主要是围绕哨兵命令实现、Redis Cluster实现，以及常用开发技巧提出来的。你可以根据这些思考题的解答思路，进一步了解下哨兵实例命令和普通实例命令的区别、Redis Cluster对事务执行的支持情况，以及函数式编程方法在Redis测试中的应用等内容。

第25讲

问题：如果我们在哨兵实例上执行publish命令，那么，这条命令是不是就是由pubsub.c文件中的publishCommand函数来处理的呢?

这道题目主要是希望你能了解，哨兵实例会使用到哨兵自身实现的命令，而不是普通Redis实例使用的命令。这一点我们从哨兵初始化的过程中就可以看到。

哨兵初始化时，会调用 **initSentinel函数**。而initSentinel函数会先把server.commands对应的命令表清空，然后执行一个循环，把哨兵自身的命令添加到命令表中。哨兵自身的命令是使用 **sentinelcmds数组**保存的。

那么从sentinelcmds数组中，我们可以看到publish命令对应的实现函数，其实是**sentinelPublishCommand**。所以，我们在哨兵实例上执行publish命令，执行的并不是pubsub.c文件中的publishCommand函数。

下面的代码展示了initSentinel 函数先清空、再填充命令表的基本过程，以及sentinelcmds数组的部分内容，你可以看下。

```
void initSentinel(void) {
    ...
    dictEmpty(server.commands,NULL);  //清空现有的命令表
    // 将sentinelcmds数组中的命令添加到命令表中
    for (j = 0; j < sizeof(sentinelcmds)/sizeof(sentinelcmds[0]); j++) {
        int retval;
        struct redisCommand *cmd = sentinelcmds+j;
        retval = dictAdd(server.commands, sdsnew(cmd->name), cmd);
        …
    }
    ...}

//sentinelcmds数组的部分命令定义
struct redisCommand sentinelcmds[] = {
    ...
    {"subscribe",subscribeCommand,-2,"",0,NULL,0,0,0,0,0},
    {"publish",sentinelPublishCommand,3,"",0,NULL,0,0,0,0,0}, //publish命令对应哨兵自身实现的sentinelPublishCom
    {"info",sentinelInfoCommand,-1,"",0,NULL,0,0,0,0,0},
    ...
};
```

第26讲

问题：在今天课程介绍的源码中，你知道为什么clusterSendPing函数计算wanted值时，是用的集群节点个数的十分之一吗？

Redis Cluster在使用clusterSendPing函数，检测其他节点的运行状态时，**既需要及时获得节点状态，又不能给集群的正常运行带来过大的额外通信负担。**

因此，clusterSendPing函数发送的Ping消息，其中包含的节点个数不能过多，否则会导致Ping消息体较大，给集群通信带来额外的负担，影响正常的请求通信。而如果Ping消息包含的节点个数过少，又会导致节点无法及时获知较多其他节点的状态。

所以，wanted默认设置为集群节点个数的十分之一，主要是为了避免上述两种情况的发生。

第27讲

问题：processCommand函数在调用完getNodeByQuery函数后，实际调用clusterRedirectClient函数进行请求重定向前，会根据当前命令是否是EXEC，分别调用discardTransaction和flagTransaction两个函数。

那么，你能通过阅读源码，知道这里调用discardTransaction和flagTransaction的目的是什么吗？

```
int processCommand(client *c) {
…
clusterNode *n = getNodeByQuery(c,c->cmd,c->argv,c->argc,
                                        &hashslot,&error_code);
if (n == NULL || n != server.cluster->myself) {
   if (c->cmd->proc == execCommand) {
      discardTransaction(c);
   } else {
      flagTransaction (c);
   }
   clusterRedirectClient(c,n,hashslot,error_code);
   return C_OK;
 }
 …
 }
```

这道题目，像@Kaito、@曾轼麟等同学都给了较为详细的解释，我完善了下他们的答案，分享给你。

首先你要知道，当Redis Cluster运行时，它并不支持跨节点的事务执行。那么，我们从题目中的代码中可以看到，当getNodeByQuery函数返回null结果，或者查询的key不在当前实例时，discardTransaction或flagTransaction函数会被调用。

这里你要**注意**，getNodeByQuery函数返回null结果，通常是表示集群不可用、key找不到对应的slot、操作的key不在同一个 slot中、key正在迁移等这些情况。

那么，当这些情况发生，或者是查询的key不在当前实例时，如果client执行的是EXEC命令，**discardTransaction函数**就会被调用，它会放弃事务的执行，清空当前client之前缓存的命令，并对事务中的key执行unWatch操作，最后重置client的事务标记。

而如果当前client执行的是事务中的普通命令，那么 **flagTransaction函数**会被调用。它会给当前client设置标记CLIENT_DIRTY_EXEC。这样一来，当client后续执行EXEC命令时，就会根据这个标记，放弃事务执行。

总结来说，就是当集群不可用、key找不到对应的slot、key不在当前实例中、操作的key不在同一个slot中，或者key正在迁移等这几种情况发生时，事务的执行都会被放弃。

第28讲

问题：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？

这道题目也有不少同学给出了正确答案，我来给你总结下。

首先，**dbAdd函数是用来将键值对插入数据库中的**。如果Redis Cluster被启用了，那么dbAdd函数会调用slotToKeyAdd函数，而slotToKeyAdd函数会调用slotToKeyUpdateKey函数。

那么在slotToKeyUpdateKey函数中，它会调用raxInsert函数更新slots_to_keys，调用链如下所示：

> dbAdd -> slotToKeyAdd -> slotToKeyUpdateKey -> raxInsert

然后，**dbAsyncDelete和dbSyncDelete是用来删除键值对的**。如果Redis Cluster被启用了，这两个函数都会调用slotToKeyUpdateKey函数。而在slotToKeyUpdateKey函数里，它会调用raxRemove函数更新slots_to_keys，调用链如下所示：

> dbAsyncDelete/dbSyncDelete -> slotToKeyDel -> slotToKeyUpdateKey -> raxRemove

另外，**empytDb函数是用来清空数据库的**。它会调用slotToKeyFlush函数，并由slotToKeyFlush函数，调用raxFree函数更新slots_to_keys，调用链如下所示：

> empytDb -> slotToKeyFlush -> raxFree

还有在 **getKeysInSlot函数**中，它会调用raxStart获得slots_to_keys的迭代器，进而查询指定slot中的keys。而在 **delKeysInSlot函数**中，它也会调用raxStart获得slots_to_keys的迭代器，并删除指定slot中的keys。

此外，@曾轼麟同学还通过查阅Redis源码的git历史提交记录，发现slots_to_keys原先是使用跳表实现的，后来才替换成字典树。而这一替换的目的，也主要是为了方便通过slot快速查找到slot中的keys。

第29讲

问题：在addReplyReplicationBacklog函数中，它会计算从节点在全局范围内要跳过的数据长度，如下所示：

```
skip = offset - server.repl_backlog_off;
```

然后，它会根据这个跳过长度计算实际要读取的数据长度，如下所示：

```
len = server.repl_backlog_histlen - skip;
```

请你阅读addReplyReplicationBacklog函数和调用它的masterTryPartialResynchronization函数，你觉得这里的skip会大于repl_backlog_histlen吗？

其实，在masterTryPartialResynchronization函数中，从节点要读取的全局位置对应了变量psync_offset，这个函数会比较psync_offset是否小于repl_backlog_off，以及psync_offset是否大于repl_backlog_off加上repl_backlog_histlen的和。

当这两种情况发生时，masterTryPartialResynchronization函数会进行**全量复制**，如下所示：

```
int masterTryPartialResynchronization(client *c) {
…
// psync_offset小于repl_backlog_off时，或者psync_offset 大于repl_backlog_off加repl_backlog_histlen的和时
if (!server.repl_backlog ||
        psync_offset < server.repl_backlog_off ||
        psync_offset > (server.repl_backlog_off + server.repl_backlog_histlen)) {
   …
   goto need_full_resync;  //进行全量复制
}
```

当psync_offset大于repl_backlog_off，并且小于repl_backlog_off加上repl_backlog_histlen的和，此时，masterTryPartialResynchronization函数会调用addReplyReplicationBacklog函数，进行**增量复制**。

而psync_offset会作为参数offset，传给addReplyReplicationBacklog函数。因此，在addReplyReplicationBacklog函数中计算skip时，就不会发生skip会大于repl_backlog_histlen的情况了，这种情况已经在masterTryPartialResynchronization函数中处理了。

第30讲

问题：Redis在命令执行的call函数中，为什么不会针对EXEC命令，调用slowlogPushEntryIfNeeded函数来记录慢命令呢？

我设计这道题的主要目的，是希望你能理解EXEC命令的使用场景和事务执行的过程。

EXEC命令是用来执行属于同一个事务的所有命令的。当程序要执行事务时，会先执行MULTI命令，紧接着，执行的命令并不会立即执行，而是被放到一个队列中缓存起来。等到EXEC命令执行时，在它之前被缓存起来等待执行的事务命令，才会实际执行。

因此，EXEC命令执行时，实际上会执行多条事务命令。此时，如果调用slowlogPushEntryIfNeeded函数记

录了慢命令的话，并不能表示EXEC本身就是一个慢命令。而实际可能会耗时长的命令是事务中的命令，并不是EXEC命令自身，所以，这里不会针对EXEC命令，来调用slowlogPushEntryIfNeeded函数。

第31讲

问题：你使用过哪些Redis的扩展模块，或者自行开发过扩展模块吗？欢迎分享一些你的经验。

我自己有使用过Redis的**TimeSeries扩展模块**，用来在一个物联网应用的场景中保存一些时间序列数据。TimeSeries模块的功能特点是可以使用标签来对不同的数据集合进行过滤，通过集合标签筛选应用需要的集合数据。而且这个模块还支持对集合数据做聚合计算，比如直接求最大值、最小值等。

此外，我还使用过**RedisGraph扩展模块**。这个模块支持把图结构的数据保存到Redis中，并充分利用了Redis使用内存读写数据的性能优势，提供对图数据进行快速创建、查询和条件匹配。你要是感兴趣，可以看下RedisGraph的官网。

第32讲

问题：Redis源码中还有一个针对SDS的小型测试框架，你知道这个测试框架是在哪个代码文件中吗？

这个小型测试框架是在testhelp.h文件中实现的。它定义了一个**宏test_cond**，而这个宏实际是一段测试代码，它的参数包括了测试项描述descr，以及具体的测试函数_c。

这里，你需要注意的是，在这个小框架中，测试函数是作为test_cond参数传递的，这体现了函数式编程的思想，而且这种开发方式使用起来也很简洁。

下面的代码展示了这个小测试框架的主要部分，你可以看下。

```
int __failed_tests = 0;  //失败的测试项的数目
int __test_num = 0;    //已测试项的数目
#define test_cond(descr,_c) do { \
    __test_num++; printf("%d - %s: ", __test_num, descr); \
    if(_c) printf("PASSED\n"); else {printf("FAILED\n"); __failed_tests++;} \  //运行测试函数_c，如果能通过，则
} while(0);
```

那么，基于这个测试框架，在sds.c文件的sdsTest函数中，我就调用了test_cond宏，对SDS相关的多种操作进行了测试，你可以看看下面的示例代码。

```
int sdsTest(void) {
    {
        sds x = sdsnew("foo");  //调用sdsnew创建一个sds变量x
        test_cond("Create a string and obtain the length",
 sdslen(x) == 3 && memcmp(x,"foo\0",4) == 0)  //调用test_cond测试sdsnew是否成功执行

        …
        x = sdscat(x,"bar");  //调用sdscat向sds变量x追求字符串
        test_cond("Strings concatenation",
 sdslen(x) == 5 && memcmp(x,"fobar\0",6) == 0); //调用test_cond测试sdscat是否成功执行
         …}
```

小结

今天这节课，也是我们最后一节答疑课，希望通过这5节答疑课程，解答了你对咱们课后思考题的疑问。同时也希望，你能通过这些课后思考题，去进一步扩展自己对Redis源码的了解，以及掌握Redis实现中的设计思想。

当然，如果你在看了答疑后，仍然有疑惑不解的话，也欢迎你在留言区写下你的疑问，我会和你继续探讨。