

答疑5 | 第25~32讲课后思考题答案及常见问题答疑

2021-11-04 蒋德钧

《Redis源码剖析与实战》

[课程介绍 >](#)



讲述：蒋德钧

时长 13:03 大小 11.96M



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。

 复制代码

```
1 void initSentinel(void) {
2     ...
3     dictEmpty(server.commands,NULL); //清空现有的命令表
4     // 将sentinelcmds数组中的命令添加到命令表中
5     for (j = 0; j < sizeof(sentinelcmds)/sizeof(sentinelcmds[0]); j++) {
6         int retval;
7         struct redisCommand *cmd = sentinelcmds+j;
8         retval = dictAdd(server.commands, sdsnew(cmd->name), cmd);
9         ...
10    }
11    ...}
12
13 //sentinelcmds数组的部分命令定义
14 struct redisCommand sentinelcmds[] = {
15     ...
16     {"subscribe",subscribeCommand,-2,"",0,NULL,0,0,0,0,0},
17     {"publish",sentinelPublishCommand,3,"",0,NULL,0,0,0,0,0}, //publish命令对应哨兵
18     {"info",sentinelInfoCommand,-1,"",0,NULL,0,0,0,0,0},
19     ...
20 };
```

🔗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？

 复制代码

```
1 int processCommand(client *c) {
2     ...
3     clusterNode *n = getNodeByQuery(c,c->cmd,c->argv,c->argc,
4                                     &hashslot,&error_code);
5     if (n == NULL || n != server.cluster->myself) {
6         if (c->cmd->proc == execCommand) {
7             discardTransaction(c);
8         } else {
9             flagTransaction (c);
10        }
11        clusterRedirectClient(c,n,hashslot,error_code);
12        return C_OK;
13    }
14    ...
15 }
```

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，**emptyDb 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：

```
emptyDb -> 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，@曾轶麟同学还通过查阅 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：

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

 复制代码

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：

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

 复制代码

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：

 复制代码

```
1 int masterTryPartialResynchronization(client *c) {
2     ...
3     // psync_offset小于repl_backlog_off时, 或者psync_offset 大于repl_backlog_off加repl_b
4     if (!server.repl_backlog ||
5         psync_offset < server.repl_backlog_off ||
6         psync_offset > (server.repl_backlog_off + server.repl_backlog_histlen)) {
7         ...
8         goto need_full_resync; //进行全量复制
9     }
```

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。

 复制代码

```
1 int __failed_tests = 0; //失败的测试项的数目
2 int __test_num = 0; //已测试项的数目
3 #define test_cond(descr,_c) do { \
4     __test_num++; printf("%d - %s: ", __test_num, descr); \
```

```
5     if(_c) printf("PASSED\n"); else {printf("FAILED\n"); __failed_tests++;} \ //  
6 } 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。

 复制代码

```
1 int sdsTest(void) {  
2     {  
3         sds x = sdsnew("foo"); //调用sdsnew创建一个sds变量x  
4         test_cond("Create a string and obtain the length",  
5         sdslen(x) == 3 && memcmp(x,"foo\0",4) == 0) //调用test_cond测试sdsnew是否成功执行  
6  
7         ...  
8         x = sdscat(x,"bar"); //调用sdscat向sds变量x追加字符串  
9         test_cond("Strings concatenation",  
10        sdslen(x) == 5 && memcmp(x,"fobar\0",6) == 0); //调用test_cond测试sdscat是否成功执  
11        ...}
```


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。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 3  提建议

上一篇 答疑4 | 第19~24讲课后思考题答案及常见问题答疑

下一篇 结束语 | Redis源码阅读，让我们从新开始

更多学习推荐

最新 Java 面试加油包 重磅上线！限时免费



6 家大厂面试常考题

4 位资深专家视频课

15 个 Java 核心技术点

100 道算法必会题+详细解析

免费去领 

686 道 Java 面试高频题+详细解析

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。