答疑4-第19~24讲课后思考题答案及常见问题答疑

你好，我是蒋德钧。这节课，我们继续来解答第19到24讲的课后思考题。

注意，今天讲解的这些思考题，一方面会涉及Redis哨兵实例的代码细节，以及管道机制在Redis中的应用；另一方面，这些思考题也是考查常用的开发知识，比如状态机、子进程使用等进程考查。希望你通过这节课的内容，可以再回顾下Redis哨兵实例的代码，并进一步了解题目解答中涉及的开发知识和技术。

第19讲

问题：RDB文件的创建是由一个子进程来完成的，而AOF重写也是由一个子进程完成的，这两个子进程可以各自单独运行。那么请你思考一下，为什么Redis源码中在有RDB子进程运行时，不会启动AOF重写子进程呢？

我设计这道题的目的，是希望你能了解和掌握RDB文件创建和AOF重写这两个操作本身，涉及到的资源消耗。我们在开发系统软件时，对于使用子进程或是线程来进行并发处理，有时会存在**一个误区：只要使用了多子进程或是多线程就可以加速并行执行的任务。**

但是，执行多子进程能够获得的收益还是要看这些子进程，对资源竞争的情况。就像这道题目提出的，虽然RDB创建和AOF重写可以会用两个子进程单独运行，但是从它们使用的资源角度来看，它们之间会存在竞争。

那么，一个最明显的资源竞争就是**对磁盘的写竞争**。创建RDB文件和重写AOF，都需要把数据写入磁盘，如果同时让这两个子进程写盘，就会给磁盘带来较大的压力。而除了磁盘资源竞争以外，RDB文件创建和AOF重写还需要读取Redis数据库中的所有键值对，如果这两个子进程同时执行，也会消耗CPU资源。

第20讲

问题：这节课，我给你介绍了重写子进程和主进程间进行操作命令传输、ACK信息传递用的三个管道。那么，你在Redis源码中还能找到其他使用管道的地方吗？

这道题目，是希望你能更多地了解下管道在Redis中的应用。有不少同学都找到了多个使用管道的地方，我在这里总结下。

- **首先，创建RDB、AOF重写和主从复制时会用到管道。**

在RDB文件的创建函数rdbSaveBackground、AOF重写的函数rewriteAppendOnlyFileBackground，以及把RDB通过socket传给从节点的函数rdbSaveToSlavesSockets中，它们都会调用openChildInfoPipe函数，创建一个管道**child_info_pipe**，这个管道的描述符数组，保存在了全局变量server中。

当RDB创建结束或是AOF文件重写结束后，这两个函数会调用sendChildInfo函数，通过刚才创建的管道child_info_pipe，把子进程写时复制的实际数据量发送给父进程。

下面的代码展示了rdbSaveBackground、rewriteAppendOnlyFileBackground、rdbSaveToSlavesSockets这三个函数使用管道的主要代码，你可以看下。

```
int rdbSaveBackground(char *filename, rdbSaveInfo *rsi) {
…
openChildInfoPipe();
if ((childpid = fork()) == 0) {
…
server.child_info_data.cow_size = private_dirty; //记录实际的写时复制数据量
 sendChildInfo(CHILD_INFO_TYPE_RDB); //将写时复制数据量发送给父进程
 …} …}

int rdbSaveToSlavesSockets(rdbSaveInfo *rsi) {
…
openChildInfoPipe();
if ((childpid = fork()) == 0) {
…
server.child_info_data.cow_size = private_dirty; //记录实际的写时复制数据量
sendChildInfo(CHILD_INFO_TYPE_RDB); //将写时复制数据量发送给父进程
…} …}


int rewriteAppendOnlyFileBackground(void) {
…
openChildInfoPipe();  //创建管道
…
if ((childpid = fork()) == 0) {
…
if (rewriteAppendOnlyFile(tmpfile) == C_OK) {
…
server.child_info_data.cow_size = private_dirty; //记录实际写时复制的数据量
sendChildInfo(CHILD_INFO_TYPE_AOF); //将写时复制的数据量发送给父进程
…} …}
…}
```

此外，在刚才介绍的rdbSaveToSlavesSockets函数中，它还会创建一个管道。当子进程把数据传给从节点后，子进程会使用这个管道，向父进程发送成功接收到所有数据传输的从节点ID，你可以看看下面的代码。

```
int rdbSaveToSlavesSockets(rdbSaveInfo *rsi) {
…
if (pipe(pipefds) == -1) return C_ERR;
server.rdb_pipe_read_result_from_child = pipefds[0];  //创建管道读端
server.rdb_pipe_write_result_to_parent = pipefds[1]; //创建管道写端
…
if ((childpid = fork()) == 0) {
…
//数据传输完成后，通过管道向父进程传输从节点ID
if (*len == 0 || write(server.rdb_pipe_write_result_to_parent,msg,msglen) != msglen) {…}
…} …}
```

- **其次，Redis module运行时会用到管道。**

在module的初始化函数moduleInitModulesSystem中，它会创建一个管道**module_blocked_pipe**，这个管道会用来唤醒由于处理module命令而阻塞的客户端。

下面的代码展示了管道在Redis module中的使用，你可以看下。

```
void moduleInitModulesSystem(void) {
...
if (pipe(server.module_blocked_pipe) == -1) {...} //创建管道
...}

int RM_UnblockClient(RedisModuleBlockedClient *bc, void *privdata) {
...
if (write(server.module_blocked_pipe[1],"A",1) != 1) {...} //向管道中写入“A”字符，表示唤醒被module阻塞的客户端
...}

void moduleHandleBlockedClients(void) {
...
while (read(server.module_blocked_pipe[0],buf,1) == 1); //从管道中读取字符
...}
```

- **最后，linuxMadvFreeForkBugCheck函数会用到管道。**

基于arm64架构的Linux内核有一个Bug，这个Bug可能会导致数据损坏。而Redis源码就针对这个Bug，打了一个补丁，这个补丁在main函数的执行过程中，会调用linuxMadvFreeForkBugCheck函数，这个函数会fork一个子进程来判断是否发现Bug，而子进程会使用管道来和父进程交互检查结果。你也可以具体看下修复这个Bug的补丁。

第21讲

问题：这节课我们介绍的状态机是当实例为从库时会使用的。那么，当一个实例是主库时，为什么不需要使用一个状态机，来实现主库在主从复制时的流程流转呢？

在Redis实现主从复制时，从库涉及到的状态变迁有很多，包括了发起连接、主从握手、复制类型判断、请求数据等。因此，使用状态机开发从库的复制流程，可以很好地帮助我们实现状态流转。

但是，如果你再去看下主从复制的启动，你会发现，**主从复制都是由从库执行slaveof或replicaof命令而开始**。这也就是说，主从复制的发起方是从库，而对于主库来说，它只是**被动式地响应**从库的各种请求，并根据从库的请求执行相应的操作，比如生成RDB文件或是传输数据等。

而且，从另外一个角度来说，主库可能和多个从库进行主从复制，而**不同从库的复制进度和状态很可能并不一样**，如果主库要维护状态机的话，那么，它还需要为每个从库维护一个状态机，这个既会增加开发复杂度，也会增加运行时的开销。正是因为这些原因，所以主库并不需要使用状态机进行状态流转。

除此之外，@曾轼麟同学也提到了一个原因，主库本身是可能发生故障，并要进行故障切换的。如果主库在执行主从复制时，也维护状态机，那么**一旦主库发生了故障，也还需要考虑状态机的冗余备份和故障切换**，这会给故障切换的开发和执行带来复杂度和开销。而从库维护状态机本身就已经能完成主从复制，所以没有必要让主库再维护状态机了。

第22讲

问题：哨兵实例本身是有配置文件sentinel.conf的，那么你能在哨兵实例的初始化过程中，找到解析这个配置文件的函数吗？

在前面的第8讲中，我重点给你介绍了Redis server的启动和初始化过程。因为哨兵实例本身也是一个Redis server，所以它启动后的初始化代码执行路径，和Redis server是类似的。

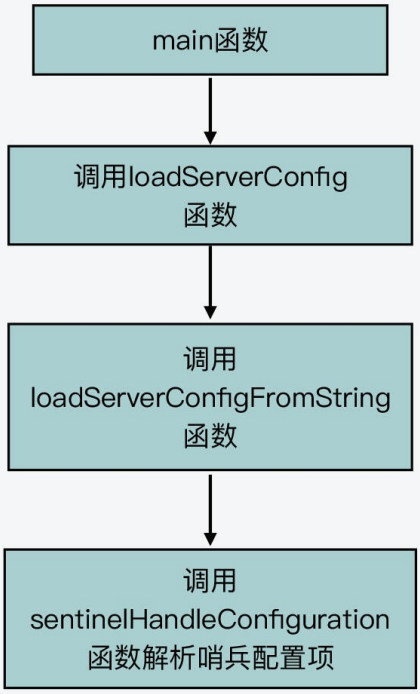
哨兵实例启动后，它的入口函数是serve.c文件中的**main函数**。然后，main函数会调用loadServerConfig函数加载配置文件。而loadServerConfig会进一步调用loadServerConfigFromString函数，解析配置文件中的具体配置项。

那么，当loadServerConfigFromString函数在解析配置项时，它会使用条件分支判断来匹配不同的配置项。当它匹配到配置项为“**sentinel**”时，它就会执行解析哨兵实例配置项的代码分支了，具体来说，它会**调用sentinelHandleConfiguration函数来进行解析**，如下所示：

```
void loadServerConfigFromString(char *config) {
else if (!strcasecmp(argv[0],"sentinel")) {
…
err = sentinelHandleConfiguration(argv+1,argc-1);
…}…}
```

sentinelHandleConfiguration函数是在sentinel.c文件中实现的，它和loadServerConfigFromString函数类似，也是匹配sentinel.conf中的不同配置项，进而执行不同的代码分支。你可以进一步阅读它的代码来进行了解。

我在这里也画了一张图，展示了哨兵实例解析配置项的函数调用关系，你可以看下。



第23讲

问题：哨兵实例执行的周期性函数sentinelTimer，它在函数执行逻辑的最后，会修改server.hz配置项，如下所示：

```
void sentinelTimer(void) {
...
server.hz = CONFIG_DEFAULT_HZ + rand() % CONFIG_DEFAULT_HZ;
}
```

那么，你知道调整server.hz的目的是什么吗？

这道题目，像是@Kaito、@曾轼麟、@可怜大灰狼等不少同学，都给出了正确答案，这里我就来总结一下。

那么，要回答这道题目，首先你要知道**server.hz表示的是定时任务函数serverCron的执行频率**，而哨兵实例执行的周期性函数sentinelTimer，也是在serverCron中被调用执行的。所以，sentinelTimer函数的运行频率会按照server.hz来执行。

我在第23讲中给你介绍过，当哨兵实例判断了主节点客观下线后，它们就要开始选举Leader节点，以便进行故障切换。但是，Leader选举时，哨兵需要获得半数以上的赞成票，如果在一轮选举中没能选出Leader，此时，哨兵实例会再次进行选举。

但是，为了避免多个哨兵同时开始进行选举，又同时都没法获得超过半数的赞成票，而导致Leader选举失败，sentinelTimer函数在执行的最后一步，对server.hz做了微调：**在默认值CONFIG_DEFAULT_HZ的基础上，增加一个随机值。**

这样一来，每个哨兵的执行频率就不会完全同步了。一轮选举失败后，哨兵再次选举时，不同哨兵的再次执行频率不一样，这就把它们发起投票的时机错开了，从而降低了它们都无法获得超过半数赞成票的概率，也就保证了Leader选举能快速完成，可以执行实际的故障切换。

所以，sentinelTimer函数修改server.hz，可以避免故障切换过程中，因为Leader节点选举不出来而导致无法完成的情况，提升了Redis的可用性。

第24讲

问题：哨兵在sentinelTimer函数中会调用sentinelHandleDictOfRedisInstances函数，对每个主节点都执行sentinelHandleRedisInstance函数，并且还会对主节点的所有从节点，也执行sentinelHandleRedisInstance函数。那么，哨兵会判断从节点的主观下线和客观下线吗？

这道题目是希望你能进一步阅读sentinelHandleRedisInstance函数的源码，对它的执行流程有个更加详细的了解。

@曾轼麟同学在留言区就给出了比较详细的分析，我在此基础上做了些完善，分享给你。

首先，在sentinelHandleDictOfRedisInstances函数中，它会执行一个循环流程，针对当前哨兵实例监听的每个主节点，都**执行sentinelHandleRedisInstance函数**。

在这个处理过程中，存在一个**递归调用**，也就是说，如果当前处理的节点就是主节点，那么sentinelHandleDictOfRedisInstances函数，会进一步针对这个主节点的从节点，再次调用

sentinelHandleDictOfRedisInstances函数，从而对每个从节点执行sentinelHandleRedisInstance函数。

这部分的代码逻辑如下所示：

```
void sentinelHandleDictOfRedisInstances(dict *instances) {
…
di = dictGetIterator(instances);
while((de = dictNext(di)) != NULL) {
   sentinelRedisInstance *ri = dictGetVal(de);  //获取哨兵实例监听的每个主节点
   sentinelHandleRedisInstance(ri);  //调用sentinelHandleRedisInstance
   if (ri->flags & SRI_MASTER) {  //如果当前节点是主节点，那么调用sentinelHandleDictOfRedisInstances对它的所有从节
 sentinelHandleDictOfRedisInstances(ri->slaves);
 …}
…}…}
```

然后，在sentinelHandleRedisInstance函数执行时，它会**调用sentinelCheckSubjectivelyDown函数**，来判断当前处理的实例是否主观下线。这步操作没有任何额外的条件约束，也就是说，无论当前是主节点还是从节点，都会被判断是否主观下线的。这部分代码如下所示：

```
void sentinelHandleRedisInstance(sentinelRedisInstance *ri) {
…
sentinelCheckSubjectivelyDown(ri);  //无论是主节点还是从节点，都会检查是否主观下线
…}
```

但是要注意，sentinelHandleRedisInstance函数在调用sentinelCheckObjectivelyDown函数，判断实例客观下线状态时，它会检查当前实例是否有主节点标记，如下所示：

```
void sentinelHandleRedisInstance(sentinelRedisInstance *ri) {
…
  if (ri->flags & SRI_MASTER) {  //只有当前是主节点，才检查是否客观下线
        sentinelCheckObjectivelyDown(ri);
   …}
…}
```

那么总结来说，对于主节点和从节点，它们的sentinelHandleRedisInstance函数调用路径就如下所示：

> 主节点：sentinelHandleRedisInstance -> sentinelCheckSubjectivelyDown -> sentinelCheckObjectivelyDown
> 从节点：sentinelHandleRedisInstance -> sentinelCheckSubjectivelyDown

所以，回到这道题目的答案上来说，哨兵会判断从节点的主观下线，但不会判断其是否客观下线。

此外，@曾轼麟同学还通过分析代码，看到了**从节点被判断为主观下线后，是不能被选举为新主节点的**。这个过程是在sentinelSelectSlave函数中执行的，这个函数会遍历当前的从节点，依次检查它们的标记，如果

一个从节点有主观下线标记，那么这个从节点就会被直接跳过，不会被选为新主节点。

下面的代码展示了sentinelSelectSlave函数这部分的逻辑，你可以看下。

```
sentinelRedisInstance *sentinelSelectSlave(sentinelRedisInstance *master) {
…
di = dictGetIterator(master->slaves);
    while((de = dictNext(di)) != NULL) { //遍历主节点的每一个从节点
        sentinelRedisInstance *slave = dictGetVal(de);
        …
        if (slave->flags & (SRI_S_DOWN|SRI_O_DOWN)) continue; //如果从节点主观下线，那么直接跳过该节点，不能被选为新
    …} …}
```

小结

好了，今天这节课就到这里了，我们来总结下。

今天这节课，我主要给你解答了第19讲到24讲的课后思考题。这些题目有些是涉及Redis源码的细节，比如哨兵实例的初始化操作、周期性任务对主从节点执行的操作等。对于这部分内容，我希望你能结合代码做进一步的阅读，并掌握好它。

而有些题目，则是和通用的开发知识和技巧相关。比如，管道在子进程和父进程间提供的通信机制、状态机在复杂流程开发中的使用、分布式共识开发中的投票频率调整等。关于这部分内容，我希望你能结合它们在Redis代码实现中的应用，掌握它们的使用方法，并应用到你自己的系统开发中。

精选留言：

- 曾轼麟 2021-11-02 10:25:26
  感谢老师的答疑，针对这些经典的问题，相信我还会重新回来复阅的