# 14-从代码实现看分布式锁的原子性保证

你好，我是蒋德钧。

分布式锁是Redis在实际业务场景中的一个重要应用。当有多个客户端并发访问某个共享资源时，比如要修改数据库中的某条记录，为了避免记录修改冲突，我们可以让所有客户端从Redis上获取分布式锁，只有拿到锁的客户端才能操作共享资源。

那么，对于分布式锁来说，它实现的关键就是要保证加锁和解锁两个操作是原子操作，这样才能保证多客户端访问时锁的正确性。而通过前面课程的学习，你知道Redis能通过事件驱动框架同时捕获多个客户端的可读事件，也就是命令请求。此外，在Redis 6.0版本中，多个IO线程会被用于并发地读取或写回数据。

而既然如此，你就可以来思考一个问题：**分布式锁的原子性还能得到保证吗？**

今天这节课呢，我就带你来了解下一条命令在Redis server中的执行过程，然后结合分布式锁的要求，来带你看下命令执行的原子性是如何保证的。同时，我们再来看看在有IO多路复用和多IO线程的情况下，分布式锁的原子性是否会受到影响。

这样一来，你就既可以掌握客户端的一条命令是如何完成执行的，其原子性是如何得到保证的，而且还可以把之前学习到的知识点串接应用起来。要知道，了解客户端命令的执行过程，对于日常排查Redis问题也是非常有帮助的，你可以在命令执行的过程中加入检测点，以便分析和排查运行问题。

好，那么接下来，我们就先来了解下分布式锁的实现方法，这样就能知道分布式锁对应的实现命令，以便进行进一步分析。

## 分布式锁的实现方法

我们在第一季的课程中，有学习过分布式锁的实现，你可以再去回顾下。这里，我再来简要介绍下分布式锁的加锁和解锁实现的命令。

首先，对于分布式锁的**加锁**操作来说，我们可以使用**Redis的SET命令**。Redis SET命令提供了NX和EX选项，这两个选项的含义分别是：

- **NX**，表示当操作的key不存在时，Redis会直接创建；当操作的key已经存在了，则返回NULL值，Redis对key不做任何修改。
- **EX**，表示设置key的过期时间。

因此，我们可以让客户端发送以下命令来进行加锁。其中，lockKey是锁的名称，uid是客户端可以用来唯一标记自己的ID，expireTime是这个key所代表的锁的过期时间，当这个过期时间到了之后，这个key会被删除，相当于锁被释放了，这样就避免了锁一直无法释放的问题。

```
SET lockKey uid EX expireTime NX
```

而如果还没有客户端创建过锁，那么，假设客户端A发送了这个SET命令给Redis，如下所示：

```
SET stockLock 1033 EX 30 NX
```

这样，Redis就会创建对应的key为stockLock，而键值对的value就是这个客户端的ID 1033。此时，假设有另一个客户端B也发送了SET命令，如下所示，表示要把key为stockLock的键值对值，改为客户端B的ID 2033，也就是要加锁。

```
SET stockLock 2033 EX 30 NX
```

由于使用了NX选项，如果stockLock的key已经存在了，客户端B就无法对其进行修改了，也就无法获得锁了，这样就实现了加锁的效果。

而对于**解锁**来说，我们可以使用如下的**Lua脚本**来完成，而Lua脚本会以EVAL命令的形式在Redis server中执行。客户端会使用GET命令读取锁对应key的value，并判断value是否等于客户端自身的ID。如果等于，就表明当前客户端正拿着锁，此时可以执行DEL命令删除key，也就是释放锁；如果value不等于客户端自身ID，那么该脚本会直接返回。

```
if redis.call("get",lockKey) == uid then
   return redis.call("del",lockKey)
else
   return 0
end
```

这样一来，客户端就不会误删除别的客户端获得的锁了，从而保证了锁的安全性。

好，现在我们就了解了分布式锁的实现命令。那么在这里，我们需要搞明白的问题就是：无论是加锁的SET命令，还是解锁的Lua脚本和EVAL命令，在有IO多路复用时，会被同时执行吗？或者**当我们使用了多IO线程后，会被多个线程同时执行吗？**

这就和Redis中命令的执行过程有关了。下面，我们就来了解下，一条命令在Redis是如何完成执行的。同时，我们还会学习到，IO多路复用引入的多个并发客户端，以及多IO线程是否会破坏命令的原子性。

## 一条命令的处理过程

现在我们知道，Redis server一旦和一个客户端建立连接后，就会在事件驱动框架中注册可读事件，这就对应了客户端的命令请求。而对于整个命令处理的过程来说，我认为主要可以分成四个阶段，它们分别对应了Redis源码中的不同函数。这里，我把它们对应的入口函数，也就是它们是从哪个函数开始进行执行的，罗列如下：

- 命令读取，对应readQueryFromClient函数；
- 命令解析，对应processInputBufferAndReplicate函数；

- 命令执行，对应processCommand函数；
- 结果返回，对应addReply函数；

那么下面，我们就来分别看下这四个入口函数的基本流程，以及为了完成命令执行，它们内部的主要调用关系都是怎样的。

### 命令读取阶段：readQueryFromClient函数

首先，我们来了解下readQueryFromClient函数的基本流程。

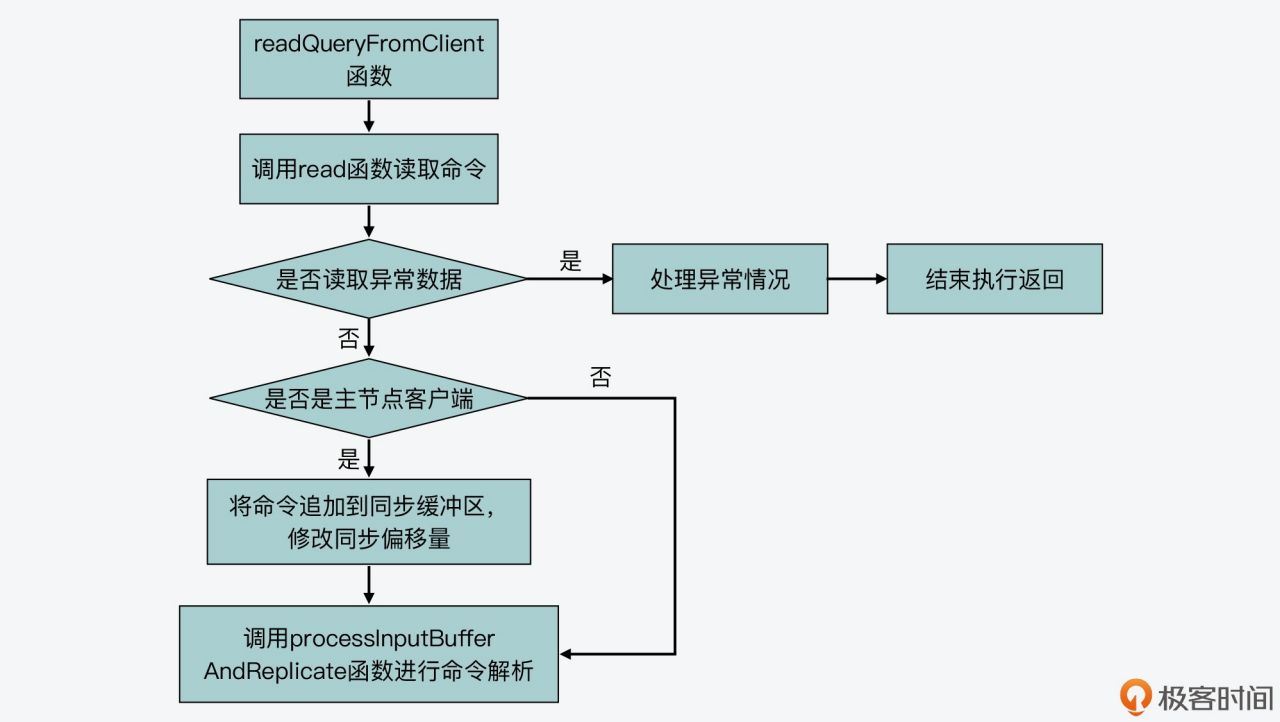
readQueryFromClient函数会从客户端连接的socket中，读取最大为readlen长度的数据，readlen值大小是宏定义PROTO_IOBUF_LEN。该宏定义是在server.h文件中定义的，默认值为16KB。

紧接着，readQueryFromClient函数会根据读取数据的情况，进行一些异常处理，比如数据读取失败或是客户端连接关闭等。此外，如果当前客户端是主从复制中的主节点，readQueryFromClient函数还会把读取的数据，追加到用于主从节点命令同步的缓冲区中。

最后，readQueryFromClient函数会调用processInputBufferAndReplicate函数，这就进入到了命令处理的下一个阶段，也就是命令解析阶段。

```
void readQueryFromClient(aeEventLoop *el, int fd, void *privdata, int mask) {
   ...
   readlen = PROTO_IOBUF_LEN;  //从客户端socket中读取的数据长度，默认为16KB
   ...
   c->querybuf = sdsMakeRoomFor(c->querybuf, readlen);  //给缓冲区分配空间
   nread = read(fd, c->querybuf+qblen, readlen);  //调用read从描述符为fd的客户端socket中读取数据
    ...
    processInputBufferAndReplicate(c);  //调用processInputBufferAndReplicate进一步处理读取内容
}
```

我在下面画了张图，展示了readQueryFromClient函数的基本流程，你可以看下。




### 命令解析阶段：processInputBufferAndReplicate函数

processInputBufferAndReplicate函数（在networking.c文件中）会根据当前客户端是否有CLIENT_MASTER标记，来执行两个分支。

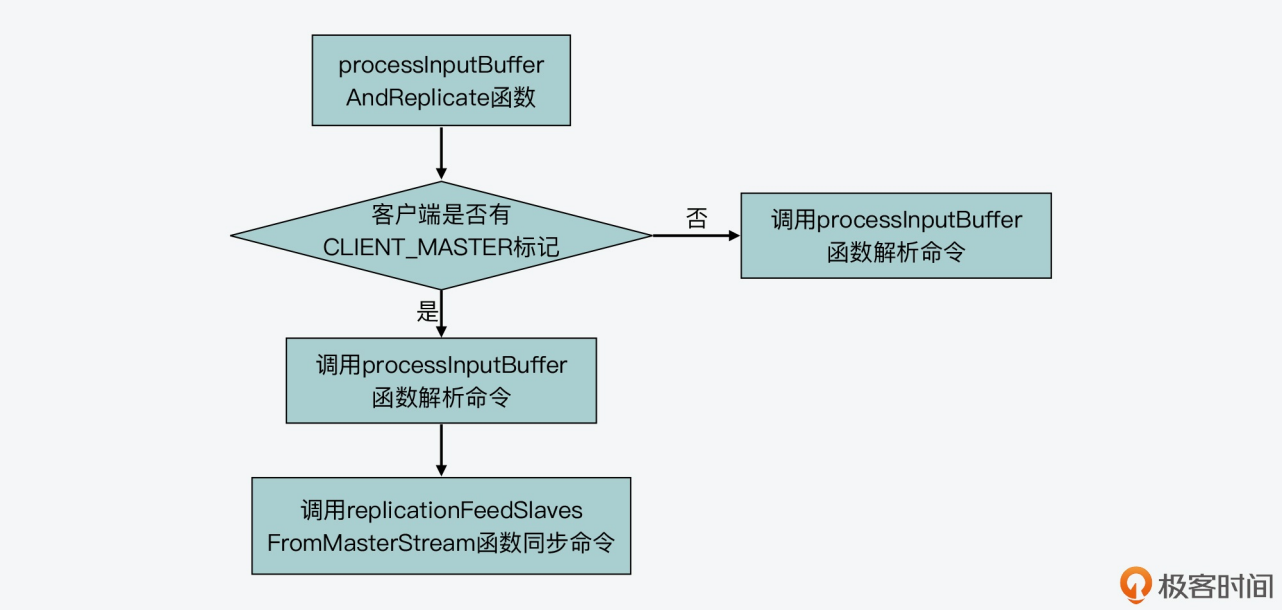
- **分支一**

这个分支对应了**客户端没有CLIENT_MASTER标记**，也就是说当前客户端不属于主从复制中的主节点。那么，processInputBufferAndReplicate函数会直接调用processInputBuffer（在networking.c文件中）函数，对客户端输入缓冲区中的命令和参数进行解析。所以在这里，实际执行命令解析的函数就是processInputBuffer函数。我们一会儿来具体看下这个函数。

- **分支二**

这个分支对应了**客户端有CLIENT_MASTER标记**，也就是说当前客户端属于主从复制中的主节点。那么，processInputBufferAndReplicate函数除了调用processInputBuffer函数，解析客户端命令以外，它还会调用replicationFeedSlavesFromMasterStream函数（在replication.c文件中），将主节点接收到的命令同步给从节点。

下图就展示了processInputBufferAndReplicate函数的基本执行逻辑，你可以看下。




好了，我们刚才了解了，**命令解析实际是在processInputBuffer函数中执行的**，所以下面，我们还需要清楚这个函数的基本流程是什么样的。

首先，processInputBuffer函数会执行一个while循环，不断地从客户端的输入缓冲区中读取数据。然后，它会**判断读取到的命令格式，是否以“*”开头**。

如果命令是以“*”开头，那就表明这个命令是PROTO_REQ_MULTIBULK类型的命令请求，也就是符合RESP协议（Redis客户端与服务器端的标准通信协议）的请求。那么，processInputBuffer函数就会进一步调用processMultibulkBuffer（在networking.c文件中）函数，来解析读取到的命令。

而如果命令不是以“*”开头，那则表明这个命令是PROTO_REQ_INLINE类型的命令请求，并不是RESP协议请求。这类命令也被称为**管道命令**，命令和命令之间是使用换行符“\r\n”分隔开来的。比如，我们使用Telnet发送给Redis的命令，就是属于PROTO_REQ_INLINE类型的命令。在这种情况下，processInputBuffer函数会调用processInlineBuffer（在networking.c文件中）函数，来实际解析命令。

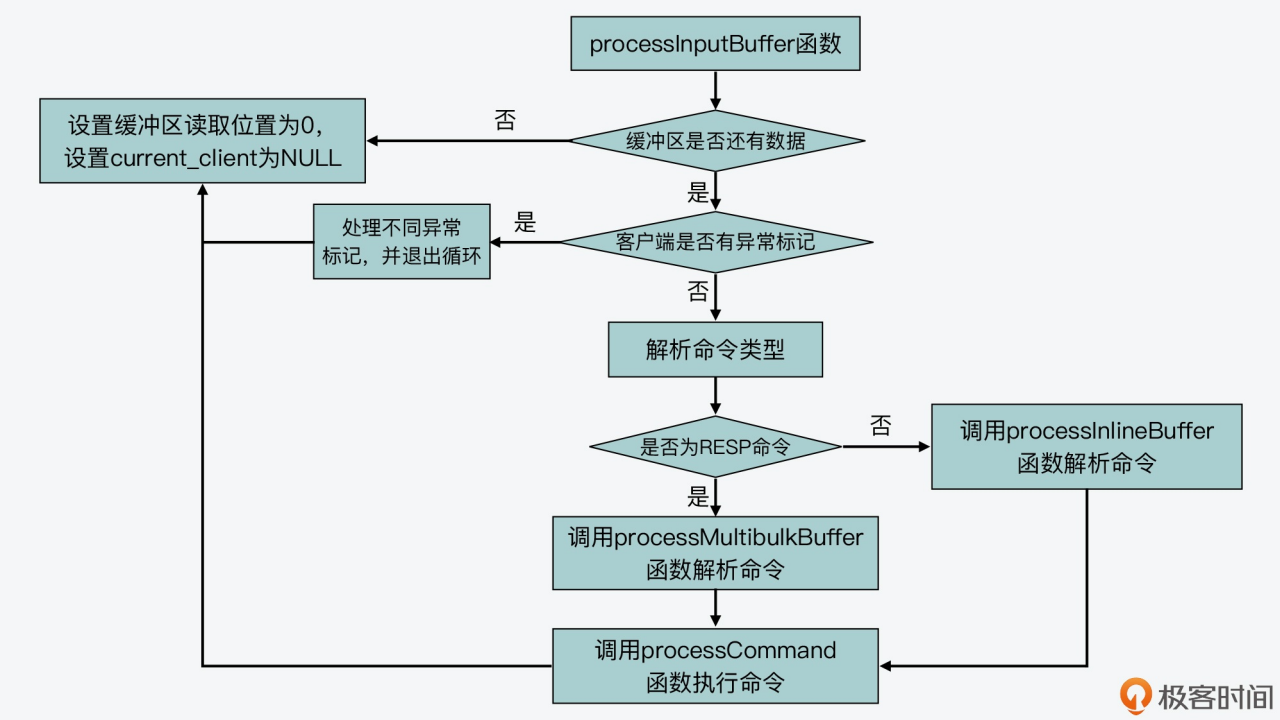
这样，等命令解析完成后，processInputBuffer函数就会调用processCommand函数，开始进入命令处理的第三个阶段，也就是命令执行阶段。

下面的代码展示了processInputBuffer函数解析命令时的主要流程，你可以看下。

```
void processInputBuffer(client *c) {
   while(c->qb_pos < sdslen(c->querybuf)) {
      ...
       if (!c->reqtype) {
            //根据客户端输入缓冲区的命令开头字符判断命令类型
            if (c->querybuf[c->qb_pos] == '*') {
                c->reqtype = PROTO_REQ_MULTIBULK; //符合RESP协议的命令
            } else {
                c->reqtype = PROTO_REQ_INLINE; //管道类型命令
            }
        }
        if (c->reqtype == PROTO_REQ_INLINE) {
            if (processInlineBuffer(c) != C_OK) break;  //对于管道类型命令，调用processInlineBuffer函数解析
        } else if (c->reqtype == PROTO_REQ_MULTIBULK) {
            if (processMultibulkBuffer(c) != C_OK) break; //对于RESP协议命令，调用processMultibulkBuffer函数解析
        }
```

```
        ...
       if (c->argc == 0) {
            resetClient(c);
        } else {
            //调用processCommand函数，开始执行命令
            if (processCommand(c) == C_OK) {
               ...   }
            ... }
        }
        ...
}
```

下图展示了processInputBuffer函数的基本执行流程，你可以再回顾下。




好，那么下面，我们接着来看第三个阶段，也就是命令执行阶段的processCommand函数的基本处理流程。

### 命令执行阶段：processCommand函数

首先，我们要知道，processCommand函数是在server.c文件中实现的。它在实际执行命令前的主要逻辑可以分成三步：

- 第一步，processCommand函数会调用moduleCallCommandFilters函数（在module.c文件），将Redis命令替换成module中想要替换的命令。
- 第二步，processCommand函数会判断当前命令是否为quit命令，并进行相应处理。
- 第三步，processCommand函数会调用lookupCommand函数，在全局变量server的commands成员变量中查找相关的命令。

这里，你需要注意下，全局变量server的**commands成员变量是一个哈希表**，它的定义是在server.h文件中的redisServer结构体里面，如下所示：

```
struct redisServer {
   ...
   dict *commands;
   ...
}
```

另外，commands成员变量的初始化是在initServerConfig函数中，通过调用dictCreate函数完成哈希表创建，再通过调用populateCommandTable函数，将Redis提供的命令名称和对应的实现函数，插入到哈希表中的。

```
void initServerConfig(void) {
...
server.commands = dictCreate(&commandTableDictType,NULL);
...
populateCommandTable();
...
}
```

而这其中的populateCommandTable函数，实际上是使用到了redisCommand结构体数组redisCommandTable。

**redisCommandTable数组**是在server.c文件中定义的，它的每一个元素是一个redisCommand结构体类型的记录，对应了Redis实现的一条命令。也就是说，redisCommand结构体中就记录了当前命令所对应的实现函数是什么。

比如，以下代码展示了GET和SET这两条命令的信息，它们各自的实现函数分别是getCommand和setCommand。当然，如果你想进一步了解redisCommand结构体，也可以去看下它的定义，在server.h文件当中。

```
struct redisCommand redisCommandTable[] = {
    ...
    {"get",getCommand,2,"rF",0,NULL,1,1,1,0,0},
    {"set",setCommand,-3,"wm",0,NULL,1,1,1,0,0},
    ...
}
```

好了，到这里，你就了解了lookupCommand函数会根据解析的命令名称，在commands对应的哈希表中查找相应的命令。

那么，一旦查到对应命令后，processCommand函数就会进行多种检查，比如命令的参数是否有效、发送命令的用户是否进行过验证、当前内存的使用情况，等等。这部分的处理逻辑比较多，你可以进一步阅读processCommand函数来了解下。

这样，等到processCommand函数对命令做完各种检查后，它就开始执行命令了。**它会判断当前客户端是**

**否有CLIENT_MULTI标记**，如果有的话，就表明要处理的是Redis事务的相关命令，所以它会按照事务的要求，调用queueMultiCommand函数将命令入队保存，等待后续一起处理。

而如果没有，processCommand函数就会调用call函数来实际执行命令了。以下代码展示了这部分的逻辑，你可以看下。

```
//如果客户端有CLIENT_MULTI标记，并且当前不是exec、discard、multi和watch命令
if (c->flags & CLIENT_MULTI &&
        c->cmd->proc != execCommand && c->cmd->proc != discardCommand &&
        c->cmd->proc != multiCommand && c->cmd->proc != watchCommand)
    {
        queueMultiCommand(c);  //将命令入队保存，等待后续一起处理
        addReply(c,shared.queued);
    } else {
        call(c,CMD_CALL_FULL);  //调用call函数执行命令
        ...
    }
```

这里你要知道，call函数是在server.c文件中实现的，它执行命令是通过调用命令本身，即redisCommand结构体中定义的函数指针来完成的。而就像我刚才所说的，每个redisCommand结构体中都定义了它对应的实现函数，在redisCommandTable数组中能查找到。

因为分布式锁的加锁操作就是使用SET命令来实现的，所以这里，我就以SET命令为例来介绍下它的实际执行过程。

SET命令对应的实现函数是**setCommand**，这是在t_string.c文件中定义的。setCommand函数首先会对命令参数进行判断，比如参数是否带有NX、EX、XX、PX等这类命令选项，如果有的话，setCommand函数就会记录下这些标记。

然后，setCommand函数会调用setGenericCommand函数，这个函数也是在t_string.c文件中实现的。setGenericCommand函数会根据刚才setCommand函数记录的命令参数的标记，来进行相应处理。比如，如果命令参数中有NX选项，那么，setGenericCommand函数会调用lookupKeyWrite函数（在db.c文件中），查找要执行SET命令的key是否已经存在。

如果这个key已经存在了，那么setGenericCommand函数就会调用addReply函数，返回NULL空值，而这也正是符合分布式锁的语义的。

下面的代码就展示了这个执行逻辑，你可以看下。

```
//如果有NX选项，那么查找key是否已经存在
if ((flags & OBJ_SET_NX && lookupKeyWrite(c->db,key) != NULL) ||
        (flags & OBJ_SET_XX && lookupKeyWrite(c->db,key) == NULL))
    {
        addReply(c, abort_reply ? abort_reply : shared.nullbulk);  //如果已经存在，则返回空值
        return;
    }
```

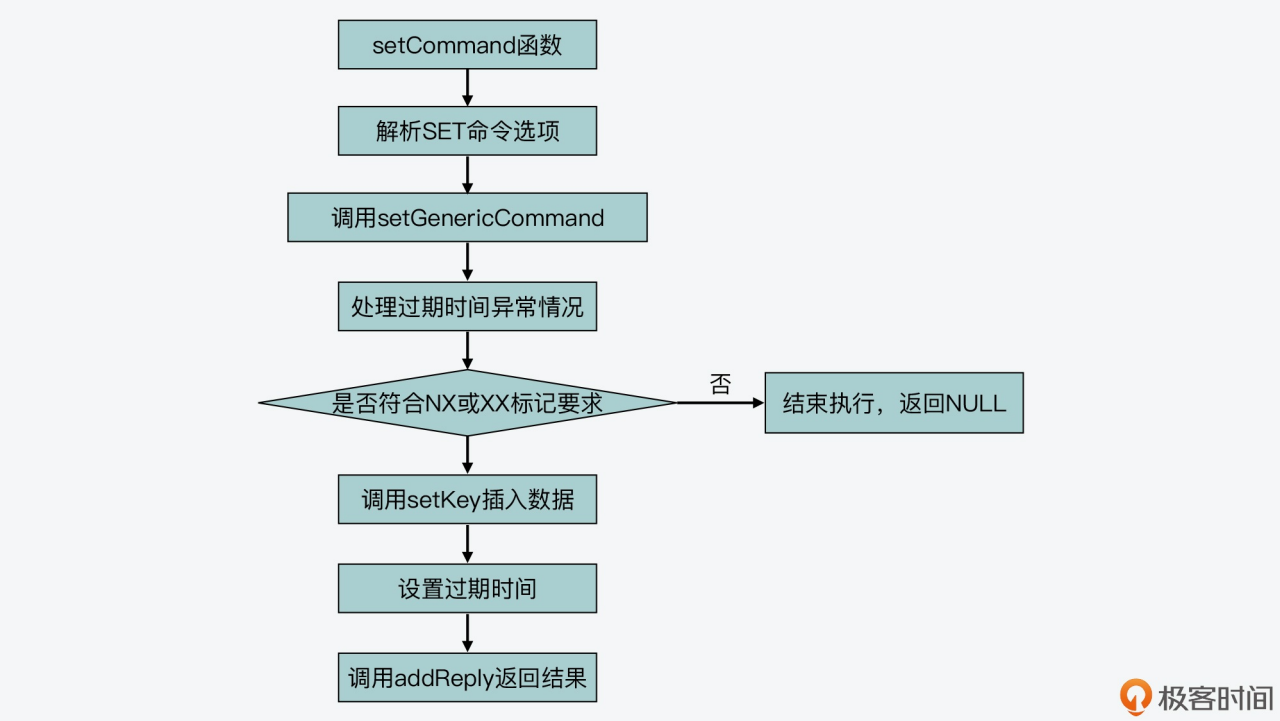
好，那么如果SET命令可以正常执行的话，也就是说命令带有NX选项但是key并不存在，或者带有XX选项但是key已经存在，这样setGenericCommand函数就会调用setKey函数（在db.c文件中）来完成键值对的实际插入，如下所示：

```
setKey(c->db,key,val);
```

然后，如果命令设置了过期时间，setGenericCommand函数还会调用setExpire函数设置过期时间。最后，setGenericCommand函数会**调用addReply函数，将结果返回给客户端**，如下所示：

```
addReply(c, ok_reply ? ok_reply : shared.ok);
```

好了，到这里，SET命令的执行就结束了，你也可以再看下下面的基本流程图。



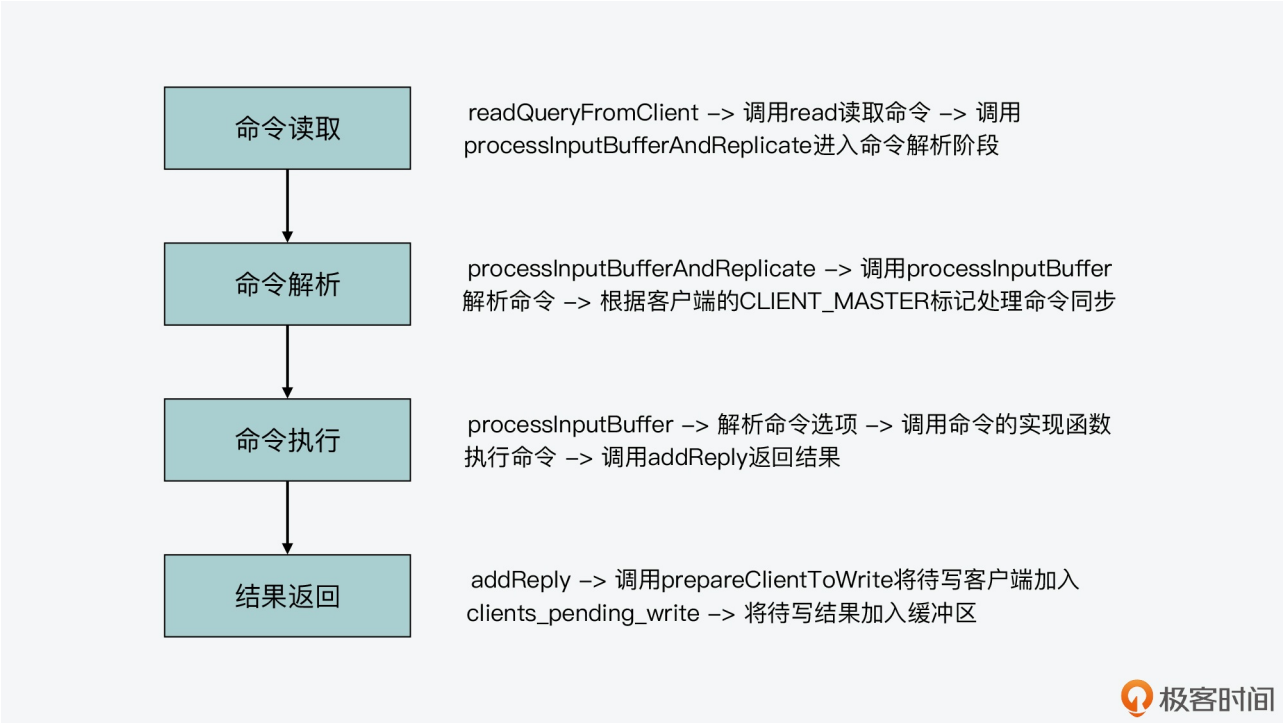

而且你也可以看到，无论是在命令执行的过程中，发现不符合命令的执行条件，或是命令能成功执行，addReply函数都会被调用，用来返回结果。所以，这就进入到我所说的命令处理过程的最后一个阶段：结果返回阶段。

## 结果返回阶段：addReply函数

addReply函数是在networking.c文件中定义的。它的执行逻辑比较简单，主要是调用prepareClientToWrite函数，并在prepareClientToWrite函数中调用clientInstallWriteHandler函数，将待写回客户端加入到全局变量server的clients_pending_write列表中。

然后，addReply函数会调用_addReplyToBuffer等函数（在networking.c中），将要返回的结果添加到客户端的输出缓冲区中。

好，现在你就了解一条命令是如何从读取，经过解析、执行等步骤，最终将结果返回给客户端的了。下图展示了这个过程以及涉及的主要函数，你可以再回顾下。




不过除此之外，你还需要注意一点，就是如果在前面的命令处理过程中，都是由IO主线程处理的，那么命令执行的原子性肯定能得到保证，分布式锁的原子性也就相应能得到保证了。

但是，如果这个处理过程配合上了我们前面介绍的IO多路复用机制和多IO线程机制，那么，这两个机制是在这个过程的什么阶段发挥作用的呢，以及会不会影响命令执行的原子性呢？

所以接下来，我们就来看下它们各自对原子性保证的影响。

## IO多路复用对命令原子性保证的影响

首先你要知道，**IO多路复用机制是在readQueryFromClient函数执行前发挥作用的**。它实际是在事件驱动框架中调用aeApiPoll函数，获取一批已经就绪的socket描述符。然后执行一个循环，针对每个就绪描述符上的读事件，触发执行readQueryFromClient函数。

这样一来，即使IO多路复用机制同时获取了多个就绪socket描述符，在实际处理时，Redis的主线程仍然是针对每个事件逐一调用回调函数进行处理的。而且对于写事件来说，IO多路复用机制也是针对每个事件逐一处理的。

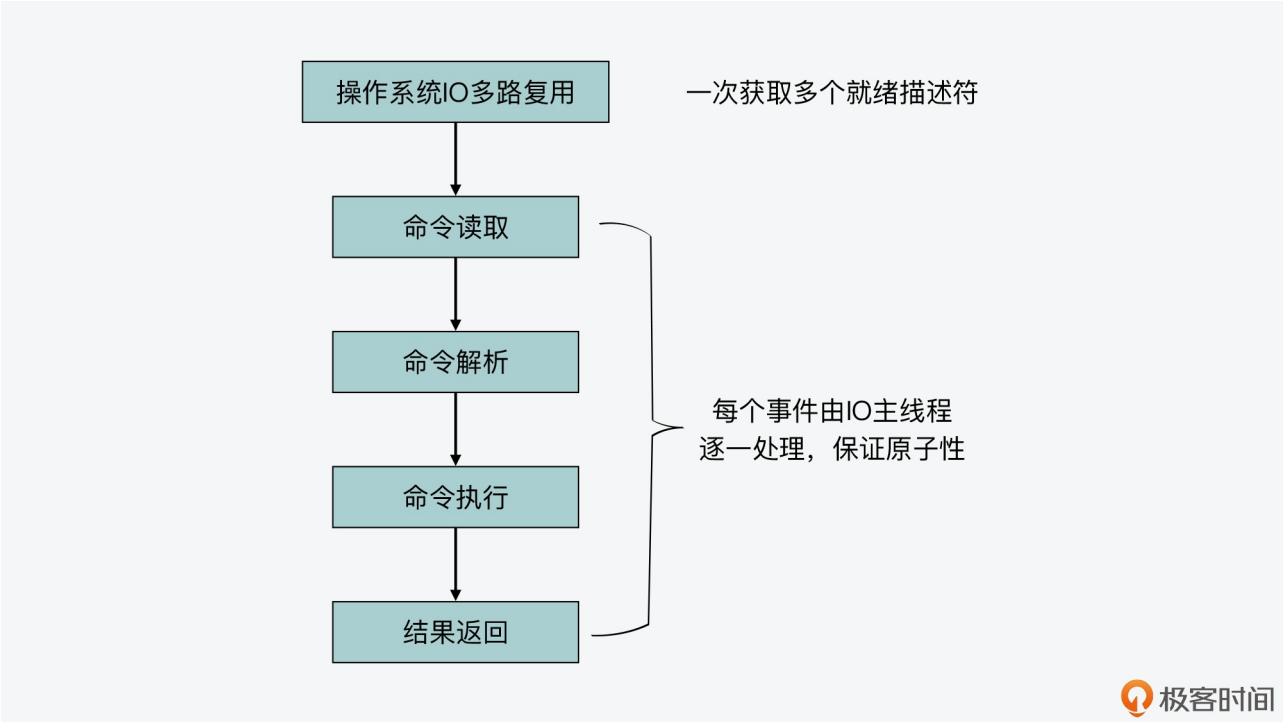
下面的代码展示了IO多路复用机制通过aeApiPoll函数获取一批事件，然后逐一处理的逻辑，你可以再看下。

```
numevents = aeApiPoll(eventLoop, tvp);

for (j = 0; j < numevents; j++) {
   aeFileEvent *fe = &eventLoop->events[eventLoop->fired[j].fd];
   if (!invert && fe->mask & mask & AE_READABLE) {
               fe->rfileProc(eventLoop,fd,fe->clientData,mask);
               fired++;
```

```
    }
```

所以这也就是说，**即使使用了IO多路复用机制，命令的整个处理过程仍然可以由IO主线程来完成，也仍然可以保证命令执行的原子性。**下图就展示了IO多路复用机制和命令处理过程的关系，你可以看下。




接下来，我们再来看下多IO线程对命令原子性保证的影响。

## 多IO线程对命令原子性保证的影响

我们知道，多IO线程可以执行读操作或是写操作。那么，对于读操作来说，readQueryFromClient函数会在执行过程中，调用postponeClient将待读客户端加入clients_pending_read等待列表。这个过程你可以再回顾下第13讲。

然后，待读客户端会被分配给多IO线程执行，每个IO线程执行的函数就是readQueryFromClient函数，readQueryFromClient函数会读取命令，并进一步调用processInputBuffer函数解析命令，这个基本过程和Redis 6.0前的代码是一样的。

不过，相比于Redis 6.0前的代码，在Redis 6.0版本中，processInputBuffer函数中**新增加了一个判断条件**，也就是当客户端标识中有CLIENT_PENDING_READ的话，那么在解析完命令后，processInputBuffer函数只会把客户端标识改为CLIENT_PENDING_COMMAND，就退出命令解析的循环流程了。

此时，processInputBuffer函数只是解析了第一个命令，也并不会实际调用processCommand函数来执行命令，如下所示：

```
void processInputBuffer(client *c) {
    /* Keep processing while there is something in the input buffer */
    while(c->qb_pos < sdslen(c->querybuf)) {
    ...
   if (c->argc == 0) {
            resetClient(c);
```

```
        } else {
            //如果客户端有CLIENT_PENDING_READ标识，将其改为CLIENT_PENDING_COMMAND，就退出循环，并不调用processComma
            if (c->flags & CLIENT_PENDING_READ) {
                c->flags |= CLIENT_PENDING_COMMAND;
                break;
            }
            if (processCommandAndResetClient(c) == C_ERR) {
                return;
            }
        }
    }
}
```

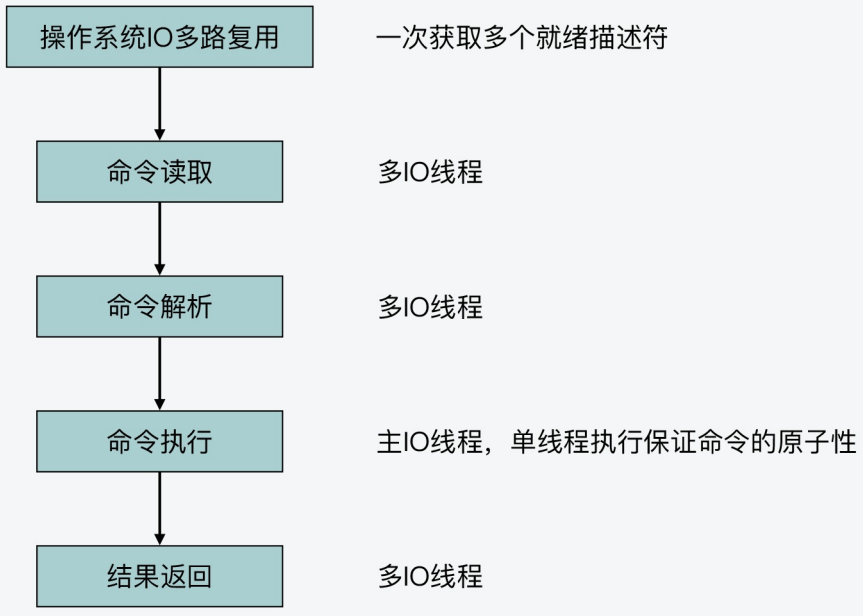
这样，等到所有的IO线程都解析完了第一个命令后，IO主线程中执行的handleClientsWithPendingReadsUsingThreads函数，会再调用processCommandAndResetClient函数执行命令，以及调用processInputBuffer函数解析剩余命令，这部分的内容你也可以再回顾下第13讲。

所以现在，你就可以知道，**即使使用了多IO线程，其实命令执行这一阶段也是由主IO线程来完成的，所有命令执行的原子性仍然可以得到保证**，也就是说分布式锁的原子性也仍然可以得到保证。

我们再来看下写回数据的流程。

在这个阶段，addReply函数是将客户端写回操作推迟执行的，而此时Redis命令已经完成执行了，所以，即使有多个IO线程在同时将客户端数据写回，也只是把结果返回给客户端，并不影响命令在Redis server中的执行结果。也就是说，**即使使用了多IO线程写回，Redis同样可以保证命令执行的原子性。**

下图展示了使用多IO线程机制后，命令处理过程各个阶段是由什么线程执行的，你可以再看下。






## 小结

今天这节课我主要结合分布式锁的原子性保证需求，带你学习了Redis处理一条命令的整个过程。其中，你需要重点关注**分布式锁实现的方法**。

我们知道，加锁和解锁操作分别可以使用SET命令和Lua脚本与EVAL命令来完成。那么，分布式锁的原子性保证，就主要依赖SET和EVAL命令在Redis server中执行时的原子性保证了。

紧接着，我还带你具体剖析了下Redis中命令处理的整个过程。我把这个过程分成了四个阶段，分别是**命令读取、命令解析、命令执行和结果返回**。所以，你还需要了解这四个阶段中所执行函数的主要流程。

这四个阶段在Redis 6.0版本前都是由主IO线程来执行完成的。虽然Redis使用了IO多路复用机制，但是该机制只是一次性获取多个就绪的socket描述符，对应了多个发送命令请求的客户端。而Redis在主IO线程中，还是逐一来处理每个客户端上的命令的，所以命令执行的原子性依然可以得到保证。

而当使用了Redis 6.0版本后，命令处理过程中的读取、解析和结果写回，就由多个IO线程来处理了。不过你也不用担心，多个IO线程只是完成解析第一个读到的命令，命令的实际执行还是由主IO线程处理。当多个IO线程在并发写回结果时，命令就已经执行完了，不存在多IO线程冲突的问题。所以，使用了多IO线程后，命令执行的原子性仍然可以得到保证。

好，最后，我也想再说下我对多IO线程的看法。从今天课程介绍的内容中，你可以看到，**多IO线程实际并不会加快命令的执行**，而是只会将读取解析命令并行化执行，以及写回结果并行化执行，并且读取解析命令还是针对收到的第一条命令。实际上，这一设计考虑还是由于网络IO需要加速处理。那么，如果命令执行本身成为Redis运行时瓶颈了，你其实可以考虑使用Redis切片集群来提升处理效率。

## 每课一问

如果将命令处理过程中的命令执行也交给多IO线程执行，你觉得除了对原子性会有影响，还会有什么好处或是其他不好的影响吗？

欢迎在留言区分享你的答案和见解。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

## 精选留言：

- Kaito 2021-08-26 00:56:27

  1、无论是 IO 多路复用，还是 Redis 6.0 的多 IO 线程，Redis 执行具体命令的主逻辑依旧是「单线程」的

  2、执行命令是单线程，本质上就保证了每个命令必定是「串行」执行的，前面请求处理完成，后面请求才能开始处理

  3、所以 Redis 在实现分布式锁时，内部不需要考虑加锁问题，直接在主线程中判断 key 是否存在即可，实现起来非常简单

  课后题：如果将命令处理过程中的命令执行也交给多 IO 线程执行，除了对原子性会有影响，还会有什么好处和坏处？

  好处：

  - 每个请求分配给不同的线程处理，一个请求处理慢，并不影响其它请求
  - 请求操作的 key 越分散，性能会变高（并行处理比串行处理性能高）
  - 可充分利用多核 CPU 资源

  坏处：

- 操作同一个 key 需加锁，加锁会影响性能，如果是热点 key，性能下降明显
- 多线程上下文切换存在性能损耗
- 多线程开发和调试不友好 [3赞]

- Milittle 2021-08-26 10:02:15
  这节课在学习之前，有80%内容是之前通过学习前面课程结合源码知晓的，老师再一串更加完整了。
  有一个问题是：单机redis，在开启多线程读写的时候，qps最高可以打到多少？多谢老师