我们知道go-redis在github上比较火的基于golang的redis客户端，是在用到这个包的过程中发生了一些非预期现象，导致让人十分疑惑，花了一些时间，探索出了几个自己认为和总结出的结论。关于结论，可以自行验证，测试，相信你在测试验证的过程中也一定会总结出属于你自己的思考。

基于go-redis,假设有如下代码(演示使用)：

```
func SpeedLimit(k string, times int, t time.Duration) (e error) {
   r := RedisInstance()
   key := "speed_limit_" + k
   r.Do("MULTI")
   r.Incr(key)
   r.Expire(key, t)
   e = r.Do("EXEC").Err()
   return
}
```

这个代码第一眼看去，可以看出启动了事务，给key原子性的+1，后面设置这个key的过期时间，然后提交事务。如果你运行这个代码，你会发现始终得到: err: ERR EXEC without MULTI 的错误提示。为什么会出现这个错误?

执行事务失败了，关键字:transaction在go-redis的close issue列表中看到有如下的对话：

ClusterClient: no way to execute transaction without WATCH/UNWATCH #435

Closed flyingmutant opened this issue on 5 Dec 2016 · 11 comments

flyingmutant commented on 5 Dec 2016

Regular `Client` has `Watch()`, which is doing `MULTI` / `EXEC` (no `WATCH` / `UNWATCH`) when no keys are specified. 小哥提出问题

This behaviour is impossible to obtain using `ClusterClient`: you have to specify at least 1 key (so that master node can be selected), but you have no way to avoid passing it to `Client.Watch`. Thus, every transaction is wrapped in `WATCH` / `UNWATCH` pair.

Workaround (use a bogus random key with the hash slot you want) is not very elegant.

Assignees: No one assigned
Labels: None yet
Projects: None yet
Milestone: No milestone
Notifications: Subscribe
3 participants

vmihailenco commented on 5 Dec 2016 • edited (Contributor)

Why you are not using Pipeline? AFAIK it is the same as Tx without Watch.

作者质问为啥不用pipeline形式，开始甩锅，估计也是没明白这个提问题的小哥到底要啥

flyingmutant commented on 5 Dec 2016 (Author)

I haven't tried it — but looking at `ClusterClient.pipelineExec()` / `ClusterClient.execClusterCmds()`, I don't see `MULTI` / `EXEC` wrapper anywhere. Am I missing anything? 这个哥们看的是ClusterClient，最后说没有看到 执行事务的 包装实现

vmihailenco commented on 5 Dec 2016 • edited (Contributor)

It does not, but AFAIK it does not matter much. Redis is single threaded so commands batched together with Pipeline will be executed without interruptions, which is the same as executing them with multi/exec. 这个作者回复说redis是单线程的，用pipeline的形式执行问题不大，效果跟multi/exec事务差不多

flyingmutant commented on 5 Dec 2016 (Author)

Can you please point me to the place in the code where `MULTI` / `EXEC` is applied, for non-transactional pipelines? I can only see it in `tx.go`. 提问题的小哥继续发出质疑，最后还举例说假设tcp包被split发过来了咋办?

I think you are wrong about pipelines guaranteeing atomic execution. There are no «start pipeline»/«end pipeline» markers, so redis can not provide such guarantee (imagine what will happen if your pipeline is split between many TCP packets?). Pipelining is a network optimisation, and nothing more. 并且认为作者关于pipeline 和 原子执行的关系都没有搞清楚。

vmihailenco commented on 5 Dec 2016 (Contributor)

There was a typo in my previous my message "it does" -> "it does not" :) Anyway I will try to find a way to not require `keys`, but I don't see any atm.

> imagine what will happen if your pipeline is split between many TCP packets

I don't insist, but I believe that in practice it is not possible that Redis Server can receive and process batched commands in parts. I don't know much about TCP, but I think different packets are merged together before data arrives to Redis.

作者大致意思我理解是说这些碎包在发给redis server之前应该会合并，大概要表达的意思是说os的接收缓冲会处理这个问题！

flyingmutant commented on 5 Dec 2016 (Author)

Would you accept a PR which adds something like `ClusterClient.MasterNodeForKey(key string)`?

一言不合，小哥突然要提PR

vmihailenco commented on 6 Dec 2016 (Contributor)

I am not sure. Why instead of `cluster.MasterNodeForKey(key string).Watch(fn)` not to use `cluster.Watch(fn, key)`? How the former is better? 作者回复对比下有啥子优点啊你这个？就瞎折腾提PR了

flyingmutant commented on 6 Dec 2016 (Author)

`cluster.Watch(fn, key)` always passes `key` to `slotMasterNode.Watch(fn, key)`, which is what I want to avoid. It can be possible to add another `cluster.Watch()` variant, but it does not look very elegant.

The best thing would be to expose redis transactions as a first-class API, and not something hidden behind `Watch` special-case. But that is a much more invasive change.

小哥继续讨论，并且后面又来个哥们表示支持这个小哥的观点。。。

bmarini commented on 10 Dec 2016

> The best thing would be to expose redis transactions as a first-class API, and not something hidden behind Watch special-case. But that is a much more invasive change.

Agreed. Also I would prefer the regular client expose redis transactions in a more obvious way too. The fact that you must use `Watch()` with no watched keys to create a normal "MULTI ... EXEC" transaction is not clear.

vmihailenco mentioned this issue on 13 Dec 2016

Add TxPipeline. #444 Merged

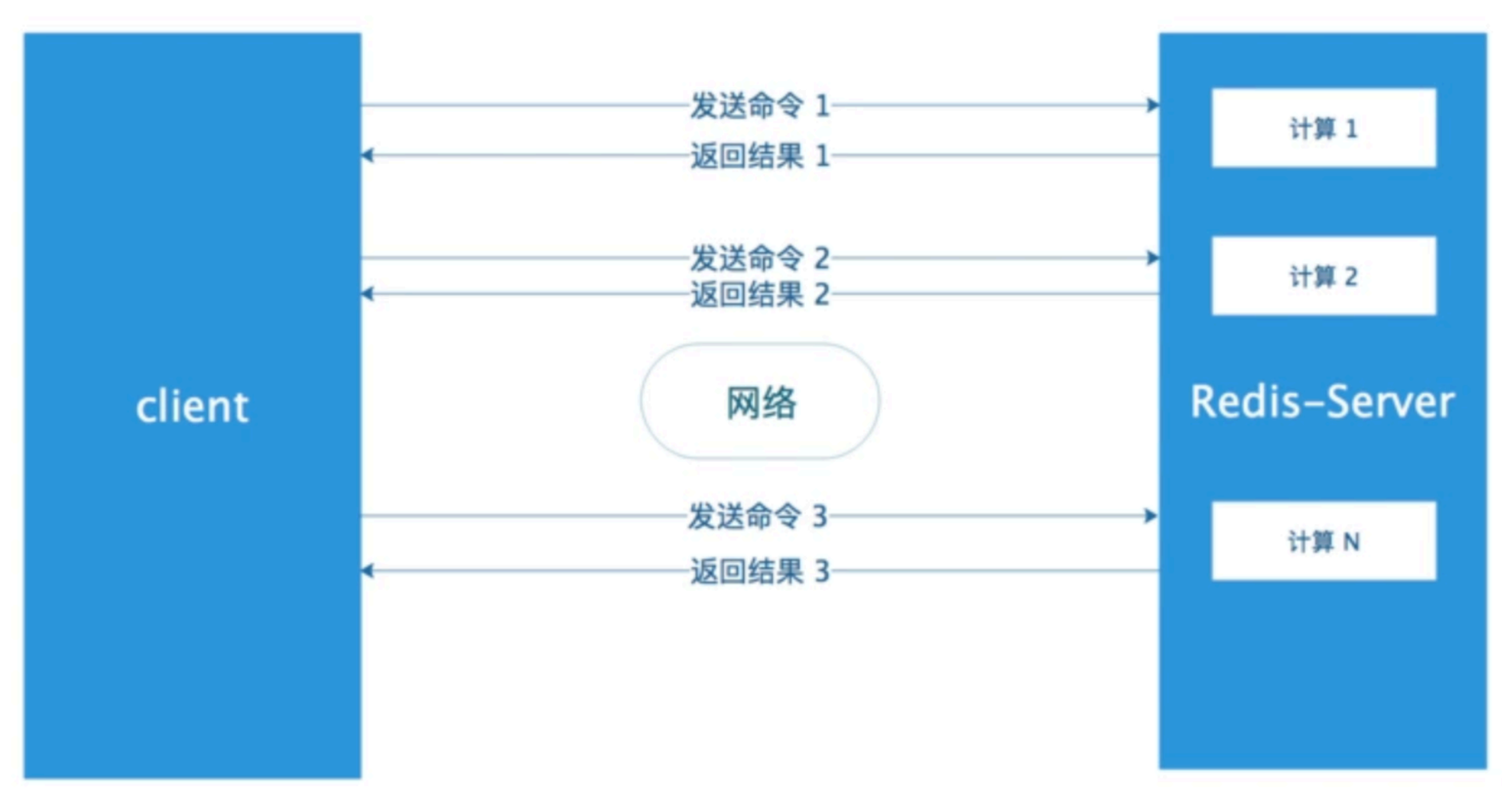
作者主动认怂，话锋一转，像是明白了什么，添加了个TxPipeline。后面也merge到主分支了

vmihailenco commented on 13 Dec 2016 (Contributor)

I like the idea. I already added `TxPipeline` to normal Client - #444. ClusterClient requires much more work though...

从上面可以看出，go-redis提供了2个方法，Pipeline，TxPipeline ，这个是解决什么的呢?

Pipeline顾名思义管道，在没有使用Pipeline的redis客户端中，交互像是这样的：



如果使用了Pipeline，那么流程会如下：
1.发送命令
2.**命令排队 (!!!)**
3.命令执行
4.返回结果

这个与上面的区别在于，把多个命令一次全部发到server，然后拿结果，减少了往返时间，效率自然高效一些。

关于Pipeline一些说法如下：
可以使用Pipeline模拟出批量操作的效果,但是在使用时需要质疑它与原生批量命令的区别,具体包含几点:

- 原生批量命令是原子性,Pipeline是非原子性的. （**这个在上面截图的兑换中也有体现**）
- 原生批量命令是一个命令对应多个key,Pipeline支持多个命令.
- 原生批量命令是Redis服务端支持实现的,而Pipeline需要服务端与客户端的共同实现.

Pipeline组装的命令个数不能没有节制,否则一次组装Pipeline数据量过大,一方面会增加客户端的等待时机,另一方面会造成一定的网络阻塞,可以将一次包含大量命令的Pipeline拆分成多次较小的Pipeline来完成.

Pipeline如上解释，那么TxPipeline表示什么呢？注意前缀Tx，这一看就是事务，猜测是支持事务的Pipeline, 所以叫TxPipeline。可以看go-redis源码来验证，如下：

```
func (c *Client) TxPipelined(fn func(Pipeliner) error) ([]Cmder, error) {
    return c.TxPipeline().Pipelined(fn)
}

// TxPipeline acts like Pipeline, but wraps queued commands with MULTI/EXEC.
func (c *Client) TxPipeline() Pipeliner {
    pipe := Pipeline{
        exec: c.processTxPipeline,
    }
    pipe.statefulCmdable.setProcessor(pipe.Process)
    return &pipe
}

func (c *Client) pubSub() *PubSub {
```

从源码的注释上也可以看出说TxPipeline的行为和Pipeline一致，但是做了一些包装使其支持了事务。

```
type baseClient struct {
    opt      *Options
    connPool pool.Pooler
    limiter  Limiter

    process           func(Cmder) error
    processPipeline   func([]Cmder) error
    processTxPipeline func([]Cmder) error

    onClose func() error // hook called when client is closed
}

func (c *baseClient) init() {
    c.process = c.defaultProcess
    c.processPipeline = c.defaultProcessPipeline
    c.processTxPipeline = c.defaultProcessTxPipeline
}

func (c *baseClient) String() string {
    return fmt.Sprintf("Redis<%s db:%d>", c.getAddr(), c.opt.DB)
}

func (c *baseClient) newConn() (*pool.Conn, error) {
    cn, err := c.connPool.NewConn()
    if err != nil {
```

init时处理的action不同：Pipeline、TxPipeline

跟进`defaultProcessTxPipeline` --> `txPipelineProcessCmds`

```
func (c *baseClient) txPipelineProcessCmds(cn *pool.Conn, cmds []Cmder) (bool, error) {
    err := cn.WithWriter(c.opt.WriteTimeout, func(wr *proto.Writer) error {
        return txPipelineWriteMulti(wr, cmds)
    })
    if err != nil {
        setCmdsErr(cmds, err)
        return true, err
    }

    err = cn.WithReader(c.opt.ReadTimeout, func(rd *proto.Reader) error {
        err := txPipelineReadQueued(rd, cmds)
        if err != nil {
            setCmdsErr(cmds, err)
            return err
        }
        return pipelineReadCmds(rd, cmds)
    })
    return false, err
}

func txPipelineWriteMulti(wr *proto.Writer, cmds []Cmder) error {
    multiExec := make([]Cmder, 0, len(cmds)+2)
    multiExec = append(multiExec, NewStatusCmd( args...: "MULTI"))
    multiExec = append(multiExec, cmds...)
    multiExec = append(multiExec, NewSliceCmd( args...: "EXEC"))
    return writeCmd(wr, multiExec...)
}
```

最后看到确实包装了：

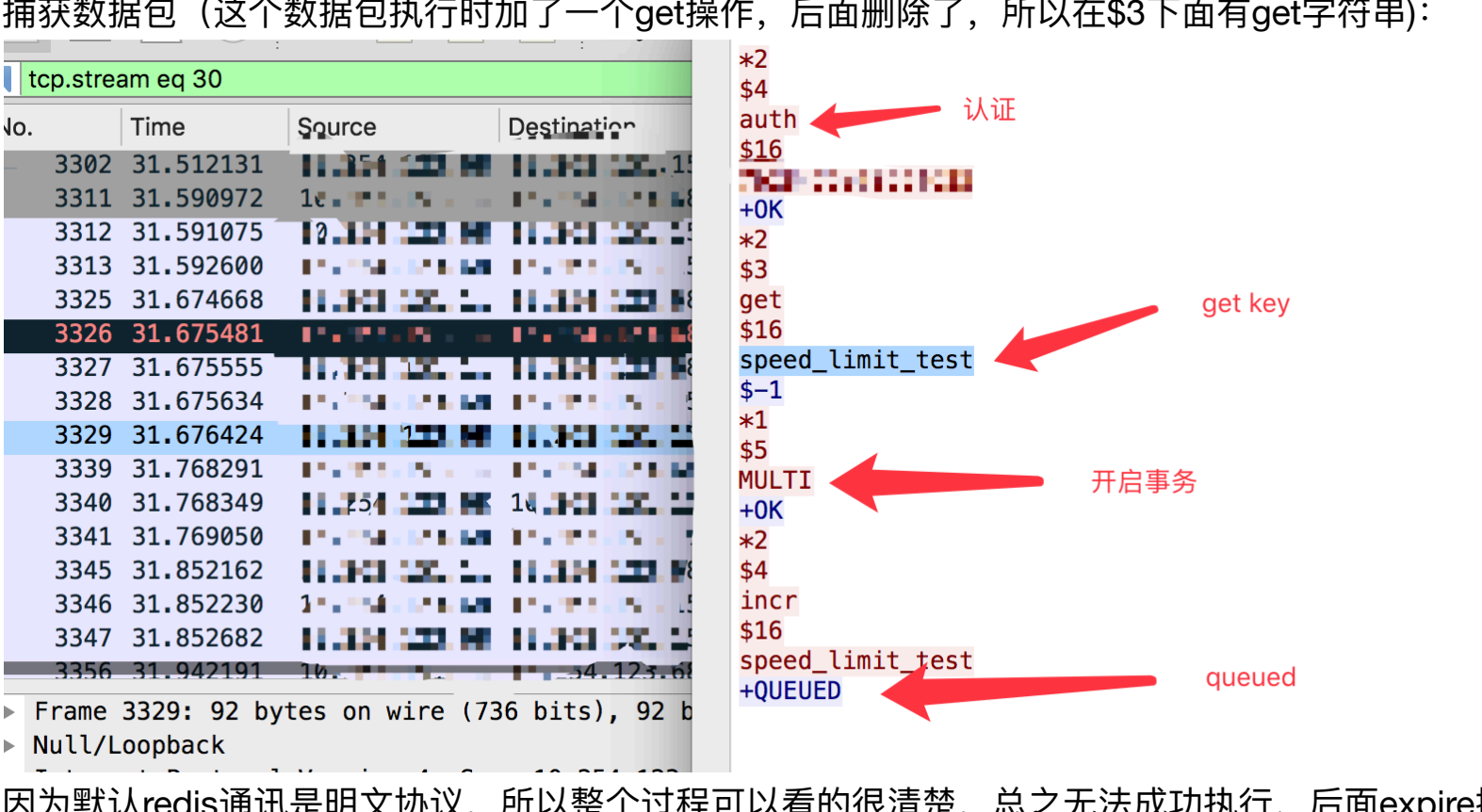
```
multiExec := make([]Cmder, 0, len(cmds)+2)
multiExec = append(multiExec, NewStatusCmd("MULTI"))
multiExec = append(multiExec, cmds...)
multiExec = append(multiExec, NewSliceCmd("EXEC"))
return writeCmd(wr, multiExec...)
```

make的时候算出len(cmds)长度并且+2，之所以+2，是因为需要增加multi和exec这两个进行事务的命令

到这里我们仅仅知道了TxPipeline说是可以支持事务，还没有解决一开始说的运行下面这段代码，始终得到: err: ERR EXEC without MULTI 的错误提示问题。

```
func SpeedLimit(k string, times int, t time.Duration) (e error) {
   r := RedisInstance()
   key := "speed_limit_" + k
   r.Do("MULTI")
   r.Incr(key)
   r.Expire(key, t)
   e = r.Do("EXEC").Err()
   return
}
```

捕获数据包（这个数据包执行时加了一个get操作，后面删除了，所以在$3下面有get字符串）：



因为默认redis通讯是明文协议，所以整个过程可以看的很清楚，总之无法成功执行，后面expire操作在另一个tcp流里，整个过程进行了不止一次握手。

在从命令行sniffer看看：

```
2. go-sniffer lo0 redis
➜ ~ go-sniffer lo0 redis
tcp and port 6379
# Start new stream: 127.0.0.1->127.0.0.1 58162->6379
# Start new stream: 127.0.0.1->127.0.0.1 6379->58162
 MULTI
 incr speed_limit_test
# Start new stream: 127.0.0.1->127.0.0.1 58163->6379
# Start new stream: 127.0.0.1->127.0.0.1 6379->58163
 expire speed_limit_test 60
 EXEC
```

第一次握手

发出新的握手 new stream

可以看出来，整个过程被打断，执行的事务也在两个会话中，无论如何都是不会成功执行的。

既然如此，我们使用Pipeline的形式改写一下代码如下，使其以Pipeline的形式执行，上面也说了Pipelin的作用：

```
func SpeedLimit(k string, times int, t time.Duration) (e error) {
   r := RedisInstance()
   key := "speed_limit_" + k
   pipe := r.Pipeline()
   pipe.Do("MULTI")
   pipe.Incr(key)
   pipe.Expire(key, t)
   pipe.Do("EXEC")
   _, e = pipe.Exec()
   return
}
```

捕获数据包：

```
2. go-sniffer lo0 redis
➜ ~ go-sniffer lo0 redis
tcp and port 6379
# Start new stream: 127.0.0.1->127.0.0.1 56043->6379
# Start new stream: 127.0.0.1->127.0.0.1 6379->56043
 MULTI
 incr speed_limit_test
 expire speed_limit_test 60
 EXEC
```

会话没有被中断，成功执行

成功执行，但是`_, e = pipe.Exec()`中e != nil ,最后返回了一个redis: can't parse int reply: "+QUEUED"的错误信息，但是命令成功执行。

我们继续改写成**TxPipeline**的形式如下：

```
func SpeedLimit(k string, times int, t time.Duration) (e error) {
   r := RedisInstance()
   key := "speed_limit_" + k
   pipe := r.TxPipeline()
   pipe.Incr(key)
   pipe.Expire(key, t)
   _, e = pipe.Exec()
   return
}
```

通过上面的分析我们知道**TxPipeline**中默认就是加了multi/exec事务标志，所以我们不需要显示的设置Do.("multi"),Do.("exec")，捕获数据包如下：

```
# Start new stream: 127.0.0.1->127.0.0.1 63422->6379
# Start new stream: 127.0.0.1->127.0.0.1 6379->63422
 MULTI
 incr speed_limit_test
 expire speed_limit_test 60
 EXEC
```

TxPipeline形式成功执行，包在了一个会话中，并且返回也没有任何报错，很优雅

比起**Pipeline，TxPipeline**显然更符合预期，**但是这里还存在一个疑问（为什么Pipeline形式的执行事务成功但是最后返回了一个redis: can't parse int reply: "+QUEUED"的错误信息提示！）**

Pipeline本身就不支持事务，官方也说了，再实际**Pipeline**模式的测试中会自己显示的增加(multi/exec)事务命令，如下其中的任意包在事务中的任意一个原子操作，在最后成功执行的返回都会返回一个parsing类型错误，虽然都执行成功了，这个感觉可以单独提个issue问问go-redis作者在**Pipeline**模式下自己手动开启事务为什么又这种问题。

```
pipe := r.Pipeline()
pipe.Do("multi")
pipe.Decr(key)
pipe.Incr(key)
pipe.IncrBy(key,100)
pipe.Decr(key)
pipe.DecrBy(key,100)
pipe.IncrByFloat(key,0.1)
pipe.Do("exec")
_, e = pipe.Exec()
```

根据我的测试，如果你要执行事务，事务之中又包含以上的原子操作，那么建议你使用**TxPipeline**，一些原子操作在非**Pipeline，TxPipeline**模式下根据抓包发现，都会在原子操作后关闭当前tcp链接，后续的命令在执行操作前会重新进行tcp三次握手。

至于一开始提出的问题：err: ERR EXEC without MULTI 的错误提示我的猜测是在普通模式下进行事务操作，事务之中又包含原子操作，触发了二次握手，导致没有构成一个完整会话，最后触发了 ERR EXEC without MULTI 的错误。关于这一块还有一些分析有空可能会在后续的文章中写出。

以上。