Redis通过MULTl，EXEC，WATCH，DISCARD等命令来实现事务（transaction）功能。

事务从MULTI命令开始，之后，该客户端发来的其他命令会被排队，客户端发来EXEC命令之后，Redis会依次执行队列中的命令。并且在执行期间，服务器不会中断事务而改去执行其他客户端的命令请求，它会将事务中的所有命令都执行完毕后，然后才去处理其他客户端的命令请求。

WATCH命令，使得客户端可以在EXEC命令执行之前，监视任意数量的数据库键，并在EXEC命令执行时，检查被监视的键是否被其他客户端修改了。如果是的话，Redis将拒绝执行事务，并向客户端返回代表事务执行失败的空回复。

DISCARD命令会清空事务队列，并退出事务模式；

**一：MULTI和EXEC命令**

1：数据结构

在表示客户端的结构体redisClient中，具有一个multiState结构的mstate属性：

**typedef** struct redisClient **{**

**...**

multiState mstate**;** */\* MULTI/EXEC state \*/*

**...**

**}** redisClient**;**

multiState结构其实就是一个事务队列，用于保存事务中的命令。multiState结构的定义如下：

/\* Client MULTI/EXEC state \*/

**typedef** struct multiCmd **{**

robj **\*\***argv**;**

int argc**;**

struct redisCommand **\***cmd**;**

**}** multiCmd**;**

**typedef** struct multiState **{**

multiCmd **\***commands**;** */\* Array of MULTI commands \*/*

int count**;** */\* Total number of MULTI commands \*/*

int minreplicas**;** */\* MINREPLICAS for synchronous replication \*/*

time\_t minreplicas\_timeout**;** */\* MINREPLICAS timeout as unixtime. \*/*

**}** multiState**;**

multiState中的commands数组属性保存每条命令，使用count标记命令条数。后两个属性没用到。

2：MULTI命令

当客户端发来MULTI命令之后，该命令的处理函数是multiCommand，代码如下：

void multiCommand**(**redisClient **\***c**)** **{**

**if** **(**c**->**flags **&** REDIS\_MULTI**)** **{**

addReplyError**(**c**,**"MULTI calls can not be nested"**);**

**return;**

**}**

c**->**flags **|=** REDIS\_MULTI**;**

addReply**(**c**,**shared**.**ok**);**

**}**

代码很简单，就是向客户端的标志位中增加REDIS\_MULTI标记，表示客户端进入事务模式。

3：事务命令排队

当客户端标志位中增加了REDIS\_MULTI标记之后，在处理命令的processCommand函数中，有这样的逻辑：

int processCommand**(**redisClient **\***c**)** **{**

**...**

*/\* Exec the command \*/*

**if** **(**c**->**flags **&** REDIS\_MULTI **&&**

c**->**cmd**->**proc **!=** execCommand **&&** c**->**cmd**->**proc **!=** discardCommand **&&**

c**->**cmd**->**proc **!=** multiCommand **&&** c**->**cmd**->**proc **!=** watchCommand**)**

**{**

queueMultiCommand**(**c**);**

addReply**(**c**,**shared**.**queued**);**

**}** **else** **{**

call**(**c**,**REDIS\_CALL\_FULL**);**

**...**

**}**

**return** REDIS\_OK**;**

**}**

因此，在事务模式下，只要客户端发来的不是EXEC、DISCARD、MULTI、WATCH这几个事务命令中之一，就会调用queueMultiCommand函数，将命令排队到c->mstate中，然后回复客户端"+QUEUED"信息。

queueMultiCommand函数的代码如下：

void queueMultiCommand**(**redisClient **\***c**)** **{**

multiCmd **\***mc**;**

int j**;**

c**->**mstate**.**commands **=** zrealloc**(**c**->**mstate**.**commands**,**

**sizeof(**multiCmd**)\*(**c**->**mstate**.**count**+**1**));**

mc **=** c**->**mstate**.**commands**+**c**->**mstate**.**count**;**

mc**->**cmd **=** c**->**cmd**;**

mc**->**argc **=** c**->**argc**;**

mc**->**argv **=** zmalloc**(sizeof(**robj**\*)\***c**->**argc**);**

memcpy**(**mc**->**argv**,**c**->**argv**,sizeof(**robj**\*)\***c**->**argc**);**

**for** **(**j **=** 0**;** j **<** c**->**argc**;** j**++)**

incrRefCount**(**mc**->**argv**[**j**]);**

c**->**mstate**.**count**++;**

**}**

代码很简单，就是将命令排队到c->mstate中，不再赘述。

4：EXEC命令

如果客户端发来了EXEC命令，则调用execCommand函数进行处理。该函数的代码如下：

void execCommand**(**redisClient **\***c**)** **{**

int j**;**

robj **\*\***orig\_argv**;**

int orig\_argc**;**

struct redisCommand **\***orig\_cmd**;**

int must\_propagate **=** 0**;** */\* Need to propagate MULTI/EXEC to AOF / slaves? \*/*

**if** **(!(**c**->**flags **&** REDIS\_MULTI**))** **{**

addReplyError**(**c**,**"EXEC without MULTI"**);**

**return;**

**}**

*/\* Check if we need to abort the EXEC because:*

*\* 1) Some WATCHed key was touched.*

*\* 2) There was a previous error while queueing commands.*

*\* A failed EXEC in the first case returns a multi bulk nil object*

*\* (technically it is not an error but a special behavior), while*

*\* in the second an EXECABORT error is returned. \*/*

**if** **(**c**->**flags **&** **(**REDIS\_DIRTY\_CAS**|**REDIS\_DIRTY\_EXEC**))** **{**

addReply**(**c**,** c**->**flags **&** REDIS\_DIRTY\_EXEC **?** shared**.**execaborterr **:**

shared**.**nullmultibulk**);**

discardTransaction**(**c**);**

**goto** handle\_monitor**;**

**}**

*/\* Exec all the queued commands \*/*

unwatchAllKeys**(**c**);** */\* Unwatch ASAP otherwise we'll waste CPU cycles \*/*

orig\_argv **=** c**->**argv**;**

orig\_argc **=** c**->**argc**;**

orig\_cmd **=** c**->**cmd**;**

addReplyMultiBulkLen**(**c**,**c**->**mstate**.**count**);**

**for** **(**j **=** 0**;** j **<** c**->**mstate**.**count**;** j**++)** **{**

c**->**argc **=** c**->**mstate**.**commands**[**j**].**argc**;**

c**->**argv **=** c**->**mstate**.**commands**[**j**].**argv**;**

c**->**cmd **=** c**->**mstate**.**commands**[**j**].**cmd**;**

*/\* Propagate a MULTI request once we encounter the first write op.*

*\* This way we'll deliver the MULTI/..../EXEC block as a whole and*

*\* both the AOF and the replication link will have the same consistency*

*\* and atomicity guarantees. \*/*

**if** **(!**must\_propagate **&&** **!(**c**->**cmd**->**flags **&** REDIS\_CMD\_READONLY**))** **{**

execCommandPropagateMulti**(**c**);**

must\_propagate **=** 1**;**

**}**

call**(**c**,**REDIS\_CALL\_FULL**);**

*/\* Commands may alter argc/argv, restore mstate. \*/*

c**->**mstate**.**commands**[**j**].**argc **=** c**->**argc**;**

c**->**mstate**.**commands**[**j**].**argv **=** c**->**argv**;**

c**->**mstate**.**commands**[**j**].**cmd **=** c**->**cmd**;**

**}**

c**->**argv **=** orig\_argv**;**

c**->**argc **=** orig\_argc**;**

c**->**cmd **=** orig\_cmd**;**

discardTransaction**(**c**);**

*/\* Make sure the EXEC command will be propagated as well if MULTI*

*\* was already propagated. \*/*

**if** **(**must\_propagate**)** server**.**dirty**++;**

handle\_monitor**:**

*/\* Send EXEC to clients waiting data from MONITOR. We do it here*

*\* since the natural order of commands execution is actually:*

*\* MUTLI, EXEC, ... commands inside transaction ...*

*\* Instead EXEC is flagged as REDIS\_CMD\_SKIP\_MONITOR in the command*

*\* table, and we do it here with correct ordering. \*/*

**if** **(**listLength**(**server**.**monitors**)** **&&** **!**server**.**loading**)**

replicationFeedMonitors**(**c**,**server**.**monitors**,**c**->**db**->**id**,**c**->**argv**,**c**->**argc**);**

**}**

函数中，如果该客户端当前没有处于事务模式下，则回复客户端错误信息；

如果客户端标志位中设置了REDIS\_DIRTY\_CAS或REDIS\_DIRTY\_EXEC标记，则回复客户端相应的错误信息，然后调用discardTransaction终止客户端的事务模式，最后给monitor发送消息后直接返回。REDIS\_DIRTY\_CAS标记表示该客户端WATCH的某个key被修改了；REDIS\_DIRTY\_EXEC标记表示事务中，某条命令在经过processCommand函数检查时出现了错误，比如找不到该命令，或者命令参数个数出错等。

接下来开始执行事务中的命令：

首先调用unwatchAllKeys，设置客户端c不再WATCH任何key；然后记录客户端当前的命令属性到orig\_\*中，以便后续恢复；

接下来，轮训c->mstate中排队的每个命令，依次执行命令。注意：在遇到第一个写操作时，需要调用execCommandPropagateMulti函数，先向从节点和AOF文件追加一个MULTI命令；c->mstate中的命令依次执行，某个命令执行失败，不影响后续命令的执行；

最后，将orig\_\*中记录的命令属性恢复，并调用discardTransaction终止客户端的事务模式；给monitor发送消息后直接返回；

5：错误处理

事务期间，可能会遇到两种类型的错误。

一种错误是语法错误，比如参数错误，命令名字出错等。这些错误是在processCommand函数中，调用命令处理函数之前检查出来的。一旦检查出错，则调用flagTransaction函数，向客户端标志位中增加REDIS\_DIRTY\_EXEC标记：

void flagTransaction**(**redisClient **\***c**)** **{**

**if** **(**c**->**flags **&** REDIS\_MULTI**)**

c**->**flags **|=** REDIS\_DIRTY\_EXEC**;**

**}**

这种情况下，该命令不会入队，在最终的EXEC命令执行函数中，直接回复客户端错误信息。

还有一种错误是运行错误，这种错误只有真正调用命令执行函数时才能检查出来。比如向一个字符串类型的键，执行RPUSH这样只针对列表的命令。

在EXEC命令处理函数execCommand中，针对排队的命令是依次调用call函数的。因此这种情况下，出错的命令，不会影响后续命令的执行。

**二：WATCH命令**

客户端可以在EXEC命令执行之前，监视任意数量的数据库键，并在EXEC命令执行时，检查被监视的键是否被其他客户端修改了。如果是的话，Redis将拒绝执行事务，并向客户端返回代表事务执行失败的空回复。

1：数据结构

WATCH命令相关的数据结构有：

在表示数据库的redisDb结构中，具有watched\_keys字典属性：

**typedef** struct redisDb **{**

**...**

dict **\***watched\_keys**;** */\* WATCHED keys for MULTI/EXEC CAS \*/*

**...**

**}** redisDb**;**

该字典以数据库键为key，以列表clients为value；列表clients中记录了WATCH该数据库键的所有客户端。

通过这种结构，可以快速得到某个数据库键当前有哪些客户端在WATCH；

在表示客户端的redisClient结构中，具有watched\_keys列表属性：

**typedef** struct redisClient **{**

**...**

multiState mstate**;** */\* MULTI/EXEC state \*/*

**...**

list **\***watched\_keys**;** */\* Keys WATCHED for MULTI/EXEC CAS \*/*

**...**

**}** redisClient**;**

每个列表元素是一个watchedKey结构，该结构具有key和db属性；列表记录了当前客户端WATCH了那些数据库键：

**typedef** struct watchedKey **{**

robj **\***key**;**

redisDb **\***db**;**

**}** watchedKey**;**

2：WATCH命令

客户端发来”WATCH <key1> [<key2> …]”后，Redis调用watchCommand函数进行处理。相关代码如下：

void watchCommand**(**redisClient **\***c**)** **{**

int j**;**

**if** **(**c**->**flags **&** REDIS\_MULTI**)** **{**

addReplyError**(**c**,**"WATCH inside MULTI is not allowed"**);**

**return;**

**}**

**for** **(**j **=** 1**;** j **<** c**->**argc**;** j**++)**

watchForKey**(**c**,**c**->**argv**[**j**]);**

addReply**(**c**,**shared**.**ok**);**

**}**

*/\* Watch for the specified key \*/*

void watchForKey**(**redisClient **\***c**,** robj **\***key**)** **{**

list **\***clients **=** **NULL;**

listIter li**;**

listNode **\***ln**;**

watchedKey **\***wk**;**

*/\* Check if we are already watching for this key \*/*

listRewind**(**c**->**watched\_keys**,&**li**);**

**while((**ln **=** listNext**(&**li**)))** **{**

wk **=** listNodeValue**(**ln**);**

**if** **(**wk**->**db **==** c**->**db **&&** equalStringObjects**(**key**,**wk**->**key**))**

**return;** */\* Key already watched \*/*

**}**

*/\* This key is not already watched in this DB. Let's add it \*/*

clients **=** dictFetchValue**(**c**->**db**->**watched\_keys**,**key**);**

**if** **(!**clients**)** **{**

clients **=** listCreate**();**

dictAdd**(**c**->**db**->**watched\_keys**,**key**,**clients**);**

incrRefCount**(**key**);**

**}**

listAddNodeTail**(**clients**,**c**);**

*/\* Add the new key to the list of keys watched by this client \*/*

wk **=** zmalloc**(sizeof(\***wk**));**

wk**->**key **=** key**;**

wk**->**db **=** c**->**db**;**

incrRefCount**(**key**);**

listAddNodeTail**(**c**->**watched\_keys**,**wk**);**

**}**

在WATCH命令处理函数watchCommand中，如果当前客户端已处于事务模式下，则回复客户端错误信息，然后直接返回；

接下来，针对命令参数中每一个key，调用watchForKey函数，设置客户端WATCH该key；

最后回复客户端"ok"；

watchForKey函数中，首先轮训列表c->watched\_keys，判断当前客户端是否已经WATCH该key了，若是，则直接返回；

然后通过key，在字典c->db->watched\_keys中查找WATCH该key的客户端列表clients。若找不到，说明还没有客户端WATCH该key，因此创建列表clients，并将key和clients添加到字典c->db->watched\_keys中；若找到了，则将当前客户端c追加到列表clients中；

然后创建一个watchedKey结构的wk，其中记录了该key及其所属的数据库db；然后将wk追加到列表c->watched\_keys中；

当某个客户端发来的命令修改了某个key后，都会调用到signalModifiedKey函数。该函数仅仅是调用touchWatchedKey而已。touchWatchedKey函数的作用，就是当数据库db中的key被改变时，增加REDIS\_DIRTY\_CAS标记到所有WATCH该key的客户端标志位中，这样这些客户端执行EXEC命令时，就会直接回复客户端错误信息。

touchWatchedKey函数的代码如下：

void touchWatchedKey**(**redisDb **\***db**,** robj **\***key**)** **{**

list **\***clients**;**

listIter li**;**

listNode **\***ln**;**

**if** **(**dictSize**(**db**->**watched\_keys**)** **==** 0**)** **return;**

clients **=** dictFetchValue**(**db**->**watched\_keys**,** key**);**

**if** **(!**clients**)** **return;**

*/\* Mark all the clients watching this key as REDIS\_DIRTY\_CAS \*/*

*/\* Check if we are already watching for this key \*/*

listRewind**(**clients**,&**li**);**

**while((**ln **=** listNext**(&**li**)))** **{**

redisClient **\***c **=** listNodeValue**(**ln**);**

c**->**flags **|=** REDIS\_DIRTY\_CAS**;**

**}**

**}**

函数中，如果字典db->watched\_keys为空，则直接返回；

然后通过key，在字典db->watched\_keys中查找WATCH该key的客户端列表clients，若找不到clients，说明没有客户端WATCH该key，直接返回；

接下来就是轮训列表clients，向其中每一个客户端的标志位中，增加REDIS\_DIRTY\_CAS标记。

这样，当WATCH该key的客户端执行EXEC时，发现客户端标志位中设置了REDIS\_DIRTY\_CAS标记后，就会回复客户端错误信息。