命令	描述
MULTI	用于标记事务的开始，其后执行的命令都将被存入命令队列，直到执行EXEC时，这些命令才会被原子的执行。
EXEC	执行在一个事务内命令队列中的所有命令，同时将当前连接的状态恢复为正常状态，即非事务状态。如果在事务中执行了WATCH命令，那么只有当WATCH所监控的Keys没有被修改的前提下，EXEC命令才能执行事务队列中的所有命令，否则EXEC将放弃当前事务中的所有命令。
DISCARD	回滚事务队列中的所有命令，同时再将当前连接的状态恢复为正常状态，即非事务状态。如果WATCH命令被使用，该命令将UNWATCH所有的Keys。
WATCH *key [key ...]*	在MULTI命令执行之前，可以指定待监控的Keys，然而在执行EXEC之前，如果被监控的Keys发生修改，EXEC将放弃执行该事务队列中的所有命令。
UNWATCH	取消当前事务中指定监控的Keys，如果执行了EXEC或DISCARD命令，则无需再手工执行该命令了，因为在此之后，事务中所有被监控的Keys都将自动取消。

剖析！Redis事务实现原理

AiChinaT...
Hi，来了？我懂~ /

作者 | Video++极链科后端Team刘聪

整理 | 包包

所谓事务(Transaction)，是指作为单个逻辑工作单元执行的一系列操作。事务必须满足ACID原则(原子性、一致性、隔离性和持久性)。简单来说事务其实就是打包一组操作（或者命令）作为一个整体，在事务处理时将顺序执行这些操作，并返回结果，如果其中任何一个环节出错，所有的操作将被回滚。

在Redis中实现事务主要依靠以下几个命令来实现：

MULTI	用于标记事务的开始，其后执行的命令都将被存入命令队列，直到执行EXEC时，这些命令才会被原子的执行。
EXEC	执行在一个事务内命令队列中的所有命令，同时将当前连接的状态恢复为正常状态，即非事务状态。如果在事务中执行了WATCH命令，那么只有当WATCH所监控的Keys没有被修改的前提下，EXEC命令才能执行事务队列中的所有命令，否则EXEC将放弃当前事务中的所有命令。
DISCARD	回滚事务队列中的所有命令，同时再将当前连接的状态恢复为正常状态，即非事务状态。如果WATCH命令被使用，该命令将UNWATCH所有的Keys。
WATCH *key [key ...]*	在MULTI命令执行之前，可以指定待监控的Keys，然而在执行EXEC之前，如果被监控的Keys发生修改，EXEC将放弃执行该事务队列中的所有命令。
UNWATCH	取消当前事务中指定监控的Keys，如果执行了EXEC或DISCARD命令，则无需再手工执行该命令了，因为在此之后，事务中所有被监控的Keys都将自动取消

Redis事务从开始到结束通常会通过三个阶段:

1.事务开始

2.命令入队

3.事务执行

以下是一个最简单的Redis事务流程：

```
redis > MULTI
OK

redis > SET "username" "bugall"
QUEUED

redis > SET "password" 161616
QUEUED

redis > GET "username"

redis > EXEC
1) ok
2) "bugall"
3) "bugall"
```

```
redis > MULTI
```

Redis中使用MULTI命令标记事务的开始，可以理解为在传统关系型数据库中的BEGIN TRANCATION语句，Redis将执行该命令的客户端从非事务状态切换成事务状态，这一切换是通过在客户端状态的flags属性中打开REDIS_MULTI标识完成, 我们看下Redis中对应部分的源码实现：

```
void multiCommand(client *c) {
    if (c->flags & CLIENT_MULTI) {
        addReplyError(c,"MULTI calls can not be nested");
        return;
    }
    c->flags |= CLIENT_MULTI;    //打开事务标识
    addReply(c,shared.ok);
}
```


在打开事务标识的客户端里，这些命令都会被暂存到一个命令队列里，不会因为用户会的输入而立即执行。

第二步就是执行事务内路基，即真正的业务逻辑：

```
redis > SET "username" "test"
redis > SET "password" 161616
redis > GET "username"
```

最后一个阶段是提交事务(或者回滚事务)：

```
redis > EXEC //DISCARD
```

这里需要注意的是，在客户端打开了事务标识后，只有命令：EXEC，DISCARD，WATCH，MULTI命令会被立即执行，其它命令服务器不会立即执行，而是将这些命令放入到一个事务队列里面，然后向客户端返回一个QUEUED回复；Redis客户端有自己的事务状态，这个状态保存在客户端状态mstate属性中，mstate的结构体类型是multiState，我们看下multiState的定义：

```
typedef struct multiState {
    multiCmd *commands;     //存放MULTI commands的数组
    int count;              //命令数量
} multiState;
```


我们再看下结构体类型multiCmd的结构：

```
typedef struct multiCmd {
    robj **argv;     //参数
    int argc;    //参数数量
    struct redisCommand *cmd; //命令指针
} multiCmd;
```


事务队列以先进先出的保存方法，较先入队的命令会被放到数组的前面，而较后入队的命令则会被放到数组的后面。

当开启事务标识的客户端发送EXEC命令的时候，服务器就会执行，客户端对应的事务队列里的命令，我们来看下EXEC 的实现细节：

```
    int orig_argc;
    struct redisCommand *orig_cmd;
    int must_propagate = 0; //同步持久化,同步主从节点

    //如果客户端没有开启事务标识
    if (!(c->flags & CLIENT_MULTI)) {
        addReplyError(c,"EXEC without MULTI");
        return;
    }
    //检查是否需要放弃EXEC
    //如果某些被watch的key被修改了就放弃执行
    if (c->flags & (CLIENT_DIRTY_CAS|CLIENT_DIRTY_EXEC)) {
        addReply(c, c->flags & CLIENT_DIRTY_EXEC ? shared.execaborterr
:

 shared.nullmultibulk);
        discardTransaction(c);
        goto handle_monitor;
    }

    //执行事务队列里的命令
    unwatchAllKeys(c); //因为redis是单线程的所以这里,当检测watch的key没有被修
改后就统一clear掉所有的watch
    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;

        //同步主从节点,和持久化
        if (!must_propagate && !(c->cmd->flags & CMD_READONLY)) {
            execCommandPropagateMulti(c);
            must_propagate = 1;
        }
        //执行命令
        call(c,CMD_CALL_FULL);
        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;
    //取消客户端的事务标识
    discardTransaction(c);
    if (must_propagate) server.dirty++;

handle_monitor:
    if (listLength(server.monitors) && !server.loading)
        replicationFeedMonitors(c,server.monitors,c->db->id,c->argv,c-
>argc);
}
```


最后我们再回顾一下事务本身的特性， 在传统关系型数据库中的事务必须依靠ACID来保证事务的可靠性和安全性，在Redis中事务总是具有一致性(Consistency)和隔离性(Isolation)，并且当Redis运行在某种特定的持久化模式下，事务也具有耐久性(Durability)； 但是并不总是能够保证原子性(Atomicity)，在正常状态下一个事务的所有命令是能按照原子性的原则执行的，但是执行的中途遇到错误，不会回滚，而是继续执行后续命令, 如下：

```
redis 127.0.0.1:7000> set k1 v1
QUEUED
redis 127.0.0.1:7000> set k2 v2
QUEUED
redis 127.0.0.1:7000> set k3 v3
QUEUED
redis 127.0.0.1:7000> exec
1) OK
2) OK
3) OK
```


如果在set k2 v2处失败，set k1已成功不会回滚，set k3还会继续执行；Redis的事务和传统的关系型数据库事务的最大区别在于，Redis不支持事务的回滚机制，即使事务队列中的某个命令在执行期间出现错误，整个事务也会继续执行下去，直到将事务队列中的所有命令都执行完毕为止，我们看下面的例子：

```
redis > SET username "bug"
OK

redis > MULTI
OK

redis > SADD member "bug" "li" "yang"
QUEUED

redis > RPUSH  username "b" "l" "y" //错误对键username使用列表键命令
QUEUED

redis > SADD password "123456" "123456" "123456"
QUEUED

redis > EXEC
1) (integer) 3
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
3) (integer) 3
```


Redis的作者在事务功能的文档中解释说，不支持事务回滚是因为这种复杂的功能和Redis追求的简单高效的设计主旨不符合，并且他认为，Redis事务的执行时，错误通常都是编程错误造成的，这种错误通常只会出现在开发环境中，