

Redis 笔记九（分布式锁、事务）

作者：[matthewhan](#)

原文链接：<https://ld246.com/article/1626860609302>

来源网站：[链滴](#)

许可协议：[署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

Q: 单机上的锁和分布式锁的联系与区别

对于在单机上运行的多线程程序来说，锁本身可以用一个变量表示。

- 变量值为 0 时，表示没有线程获取锁；
- 变量值为 1 时，表示已经有线程获取到锁了。

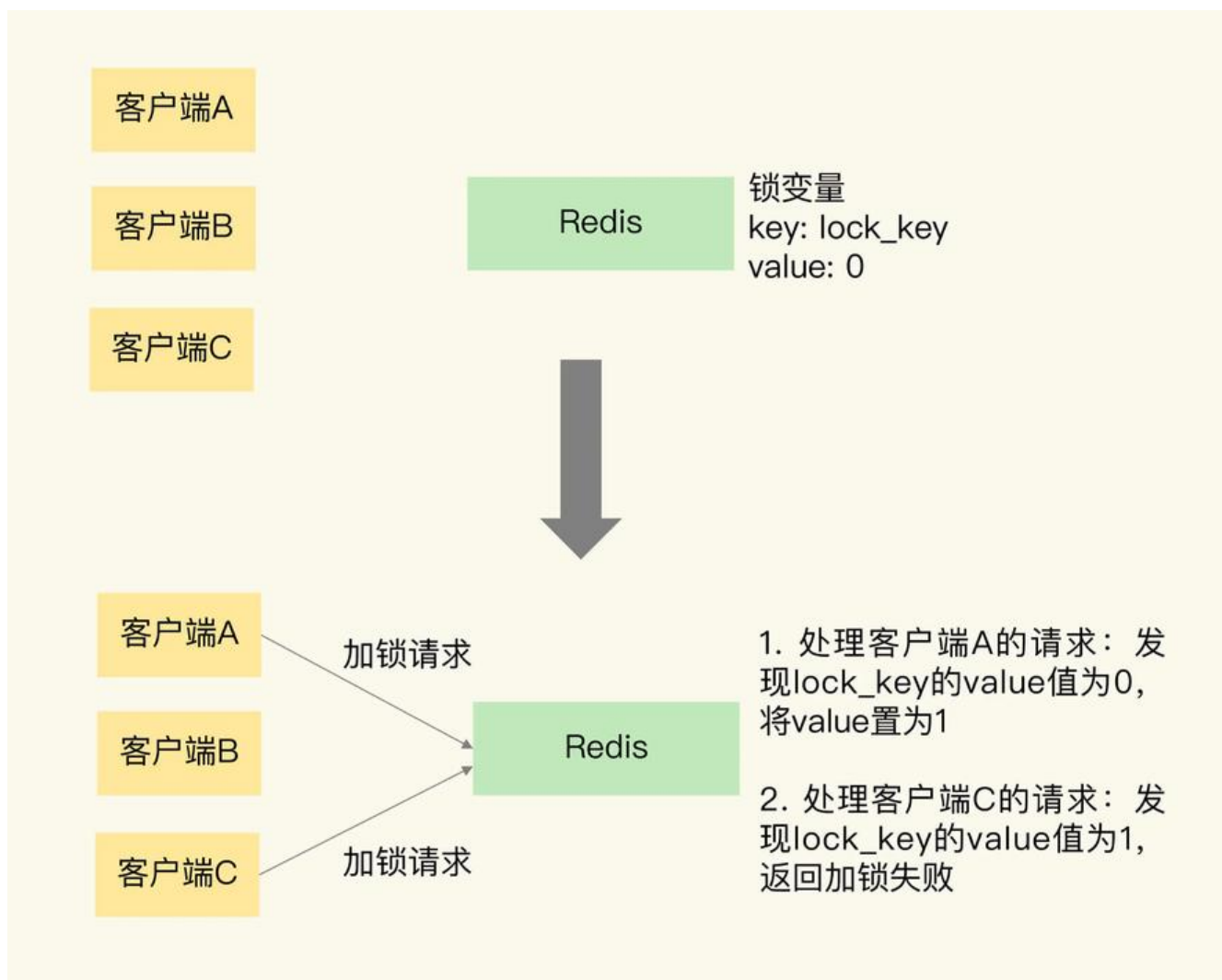
和单机上的锁类似，分布式锁同样可以用一个变量来实现。客户端加锁和释放锁的操作逻辑，也和单上的加锁和释放锁操作逻辑一致：加锁时同样需要判断锁变量的值，根据锁变量值来判断能否加锁成；释放锁时需要把锁变量值设置为 0，表明客户端不再持有锁。

但是，和线程在单机上操作锁不同的是，在分布式场景下，锁变量需要由一个共享存储系统来维护，有这样，多个客户端才可以通过访问共享存储系统来访问锁变量。相应的，加锁和释放锁的操作就变了读取、判断和设置共享存储系统中的锁变量值。

这样一来，我们就可以得出实现分布式锁的两个要求。

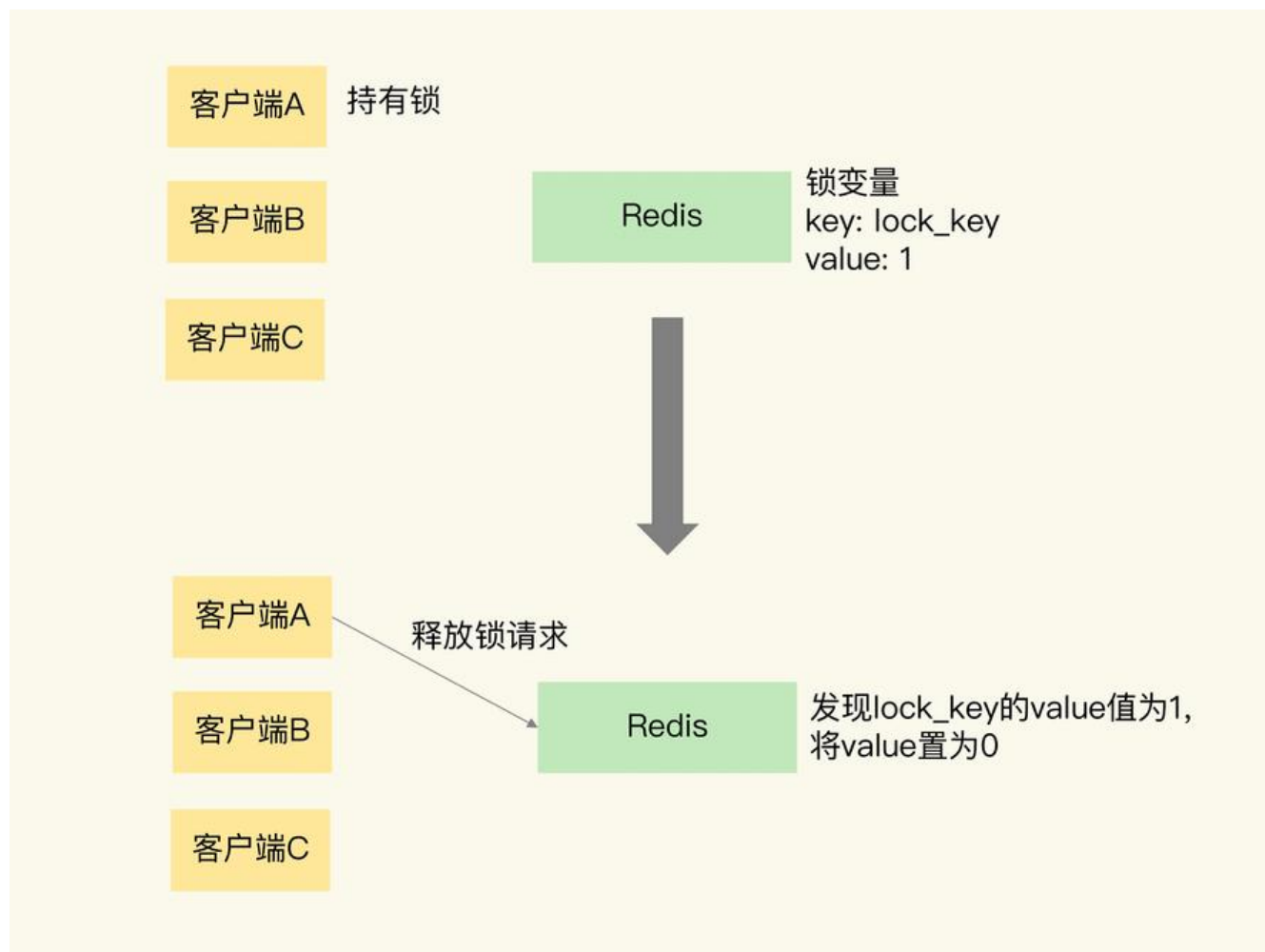
- 要求一：分布式锁的加锁和释放锁的过程，涉及多个操作。所以，在实现分布式锁时，我们需要保证这些锁操作的原子性；
- 要求二：共享存储系统保存了锁变量，如果共享存储系统发生故障或宕机，那么客户端也就无法进锁操作了。在实现分布式锁时，我们需要考虑保证共享存储系统的可靠性，进而保证锁的可靠性。

Q: 基于单个 Redis 节点实现分布式锁



在图中，客户端 A 和 C 同时请求加锁。因为 Redis 使用单线程处理请求，所以，即使客户端 A 和 C 时把加锁请求发给了 Redis，Redis 也会串行处理它们的请求。

我们假设 Redis 先处理客户端 A 的请求，读取 lock_key 的值，发现 lock_key 为 0，所以，Redis 把 lock_key 的 value 置为 1，表示已经加锁了。紧接着，Redis 处理客户端 C 的请求，此时，Redis 会发现 lock_key 的值已经为 1 了，所以就返回加锁失败的信息。



当客户端 A 持有锁时，锁变量 lock_key 的值为 1。客户端 A 执行释放锁操作后，Redis 将 lock_key 的值置为 0，表明已经没有客户端持有锁了。

要想保证操作的原子性，有两种通用的方法，分别是使用 Redis 的单命令操作和使用 Lua 脚本。

首先是 SETNX 命令，它用于设置键值对的值。具体来说，就是这个命令在执行时会判断键值对是否存在，如果不存在，就设置键值对的值，如果存在，就不做任何设置。

```
// 加锁
SETNX lock_key 1
// 业务逻辑
DO THINGS
// 释放锁
DEL lock_key
```

对于释放锁操作来说，我们可以在执行完业务逻辑后，使用 DEL 命令删除锁变量。不过，你不用担心锁变量被删除后，其他客户端无法请求加锁了。因为 SETNX 命令在执行时，如果要设置的键值对（就是锁变量）不存在，SETNX 命令会先创建键值对，然后设置它的值。所以，释放锁之后，再有客

端请求加锁时，SETNX 命令会创建保存锁变量的键值对，并设置锁变量的值，完成加锁。

总结来说，我们就可以用 SETNX 和 DEL 命令组合来实现加锁和释放锁操作。下面的伪代码示例显示锁操作的过程，你可以看下。

使用 SETNX 和 DEL 命令组合实现分布锁，存在两个潜在的风险。

风险一：假如某个客户端在执行了 SETNX 命令、加锁之后，紧接着却在操作共享数据时发生了异常，结果一直没有执行最后的 DEL 命令释放锁。因此，锁就一直被这个客户端持有，其它客户端无法拿锁，也无法访问共享数据和执行后续操作，这会给业务应用带来影响。

解决：针对这个问题，一个有效的解决方法是，给锁变量设置一个过期时间。这样一来，即使持有锁客户端发生了异常，无法主动地释放锁，Redis 也会根据锁变量的过期时间，在锁变量过期后，把它除。其它客户端在锁变量过期后，就可以重新请求加锁，这就不会出现无法加锁的问题了。

风险二：如果客户端 A 执行了 SETNX 命令加锁后，假设客户端 B 执行了 DEL 命令释放锁，此时，客户端 A 的锁就被误释放了。如果客户端 C 正好也在申请加锁，就可以成功获得锁，进而开始操作共享数据。这样一来，客户端 A 和 C 同时在对共享数据进行操作，数据就会被修改错误，这也是业务层能接受的。

解决：要能区分来自不同客户端的锁操作，可以在锁变量的值上想想办法。在使用 SETNX 命令进行锁的方法中，我们通过把锁变量值设置为 1 或 0，表示是否加锁成功。1 和 0 只有两种状态，无法表究竟是哪个客户端进行的锁操作。所以，我们在加锁操作时，可以让每个客户端给锁变量设置一个唯值，这里的唯一值就可以用来标识当前操作的客户端。在释放锁操作时，客户端需要判断，当前锁变的值是否和自己的唯一标识相等，只有在相等的情况下，才能释放锁。这样一来，就不会出现误释放的问题了。SET 命令在执行时还可以带上 EX 或 PX 选项，用来设置键值对的过期时间。

SET key value [EX seconds | PX milliseconds] [NX]

// 例子

// 加锁, unique_value 作为客户端唯一性的标识

SET lock_key unique_value NX PX 10000

NX：类似 SETNX，当不存在才 set 一个 kv

虽然使用 SET 命令和 Lua 脚本在 Redis 单节点上实现分布式锁。但是现在只用了一个 Redis 实例来存锁变量，如果这个 Redis 实例发生故障宕机了，那么锁变量就没有了。此时，客户端也无法进行锁作了，这就会影响到业务的正常执行。所以，我们在实现分布式锁时，还需要保证锁的可靠性。那怎提高呢？这就要提到基于多个 Redis 节点实现分布式锁的方式了。

基于多个 Redis 节点实现高可靠的分布式锁

Redlock 算法的基本思路，是让客户端和多个独立的 Redis 实例依次请求加锁，如果客户端能够和半以上的实例成功地完成加锁操作，那么我们就认为，客户端成功地获得分布式锁了，否则加锁失败。样一来，即使有单个 Redis 实例发生故障，因为锁变量在其它实例上也有保存，所以，客户端仍然可正常地进行锁操作，锁变量并不会丢失。

第一步是，客户端获取当前时间。

第二步是，客户端按顺序依次向 N 个 Redis 实例执行加锁操作。

这里的加锁操作和在单实例上执行的加锁操作一样，使用 SET 命令，带上 NX, EX/PX 选项，以及带客户端的唯一标识。当然，如果某个 Redis 实例发生故障了，为了保证在这种情况下，Redlock 算法够继续运行，我们需要给加锁操作设置一个超时时间。

如果客户端在和 Redis 实例请求加锁时，一直到超时都没有成功，那么此时，客户端会和下一个 Redis 实例继续请求加锁。加锁操作的超时时间需要远远地小于锁的有效时间，一般也就是设置为几十秒。

第三步是，一旦客户端完成了和所有 Redis 实例的加锁操作，客户端就要计算整个加锁过程的总耗时。

客户端只有在满足下面的这两个条件时，才能认为是加锁成功。

- 条件一：客户端从超过半数（大于等于 $N/2+1$ ）的 Redis 实例上成功获取到了锁；
- 条件二：客户端获取锁的总耗时没有超过锁的有效时间。

在满足了这两个条件后，我们需要重新计算这把锁的有效时间，计算的结果是锁的最初有效时间减去客户端为获取锁的总耗时。

Q: Redis 与事务

事务的执行过程包含三个步骤，Redis 提供了 MULTI、EXEC 两个命令来完成这三个步骤。

第一步，客户端要使用一个命令显式地表示一个事务的开启。在 Redis 中，这个命令就是 MULTI。

第二步，客户端把事务中本身要执行的具体操作（例如增删改数据）发送给服务器端。这些操作就是 Redis 本身提供的读写命令，例如 GET、SET 等。不过，这些命令虽然被客户端发送到了服务器，但 Redis 实例只是把这些命令暂存到一个命令队列中，并不会立即执行。

第三步，客户端向服务器端发送提交事务的命令，让数据库实际执行第二步中发送的具体操作。Redis 提供的 EXEC 命令就是执行事务提交的。当服务器端收到 EXEC 命令后，才会实际执行命令队列中的有命令。

```
#开启事务
127.0.0.1:6379> MULTI
OK
#将a:stock减1,
127.0.0.1:6379> DECR a:stock
QUEUED
#将b:stock减1
127.0.0.1:6379> DECR b:stock
QUEUED
#实际执行事务
127.0.0.1:6379> EXEC
1) (integer) 4
2) (integer) 9
```

Q: Redis 与原子性

先说个结论：虽然在这门课中声称是事务，但是 Redis 有个锤子的事务，原子性说白了也没法保障的拉的一批。

- 情况一：在执行 EXEC 命令前，客户端发送的操作命令本身就有错误（比如语法错误，使用了不存在的命令），在命令入队时就被 Redis 实例判断出来了。**那么整个事务就会被放弃，这个勉强算半个事务。**
- 情况二：事务操作入队时，命令和操作的数据类型不匹配，但 Redis 实例没有检查出错误。**例如现了一条命令成功，一条失败的情况，那么并不会回滚，而是成功的命令就成功的执行了。**

- Redis 中并没有提供回滚机制。虽然 Redis 提供了 DISCARD 命令，但是，这个命令只能用来主动弃事务执行，把暂存的命令队列清空，起不到回滚的效果。

- 可以像如下这样主动地终结事务，反正就是没什么卵用。

```
#读取a:stock的值4
127.0.0.1:6379> GET a:stock
"4"
#开启事务
127.0.0.1:6379> MULTI
OK
#发送事务的第一个操作，对a:stock减1
127.0.0.1:6379> DECR a:stock
QUEUED
#执行DISCARD命令，主动放弃事务
127.0.0.1:6379> DISCARD
OK
#再次读取a:stock的值，值没有被修改
127.0.0.1:6379> GET a:stock
"4"
```

- 情况三：在执行事务的 EXEC 命令时，Redis 实例发生了故障，导致事务执行失败。在这种情况下如果 Redis 开启了 AOF 日志，那么，只会有部分的事务操作被记录到 AOF 日志中。****我们需要使用 edis-check-aof 工具检查 AOF 日志文件，这个工具可以把未完成的事务操作从 AOF 文件中去除。***使用 AOF 恢复实例后，事务操作不会再被执行，从而保证了原子性。

总结：你看这个 Redis 啊？才搞几个命令就不回滚了，真的太逊了。这个 Redis 就是逊啦。

Q: Redis 与一致性

可以理解一致性就是，应用系统从一个正确的状态到另一个正确的状态，而 ACID 就是说事务能够通过 AID 来保证这个 C 的过程。C 是目的，AID 都是手段。

所以个人感觉一致性拉胯

Q: Redis 与隔离性

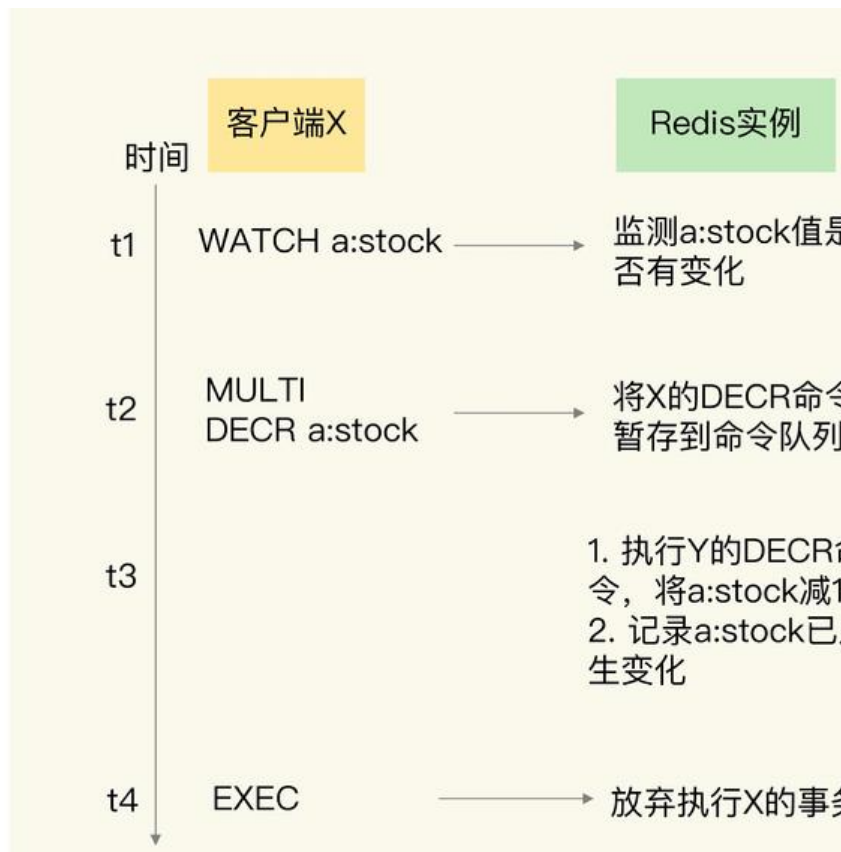
事务的隔离性保证，会受到和事务一起执行的并发操作的影响。而事务执行又可以分成命令入队（EX C 命令执行前）和命令实际执行（EXEC 命令执行后）两个阶段，所以，我们就针对这两个阶段，分两种情况来分析：

- 并发操作在 EXEC 命令前执行，此时，隔离性的保证要使用 WATCH 机制来实现，否则隔离性无法证；
- 并发操作在 EXEC 命令后执行，此时，隔离性可以保证。

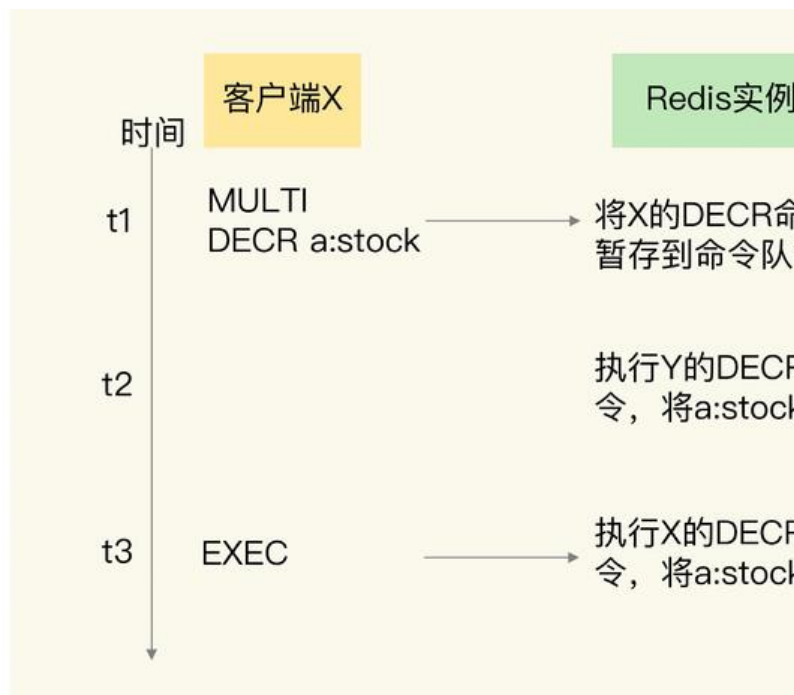
WATCH 机制：在事务执行前，监控一个或多个键的值变化情况，当事务调用 EXEC 命令执行时，WATCH 机制会先检查监控的键是否被其它客户端修改了。如果修改了，就放弃事务执行，避免事务的隔性被破坏。然后，客户端可以再次执行事务，此时，如果没有并发修改事务数据的操作了，事务就能常执行，隔离性也得到了保证。

- 情况一：一个事务的 EXEC 命令还没有执行时，事务的命令操作是暂存在命令队列中的。此时，如有其它的并发操作，我们就需要看事务是否使用了 WATCH 机制。

- 开启了 WATCH 机制（可以保证隔离性）：



- 未开启 WATCH 机制（无法保证隔离性）：



- 情况二：并发操作在 `EXEC` 命令之后被服务器端接收并执行。因为 Redis 是用单线程执行命令，而 `EXEC` 命令执行后，Redis 会保证先把命令队列中的所有命令执行完。所以，在这种情况下，并发操作不会破坏事务的隔离性。

Q: Redis 与持久性

如果 Redis 没有使用 RDB 或 AOF，那么事务的持久化属性肯定得不到保证。如果 Redis 使用了 RDB 模式，那么，在一个事务执行后，而下一次的 RDB 快照还未执行前，如果发生了实例宕机，这种情况下，事务修改的数据也是不能保证持久化的。

如果 Redis 采用了 AOF 模式，因为 AOF 模式的三种配置选项 no、everysec 和 always 都会存在数据丢失的情况，所以，事务的持久性属性也还是得不到保证。

所以，不管 Redis 采用什么持久化模式，事务的持久性属性是得不到保证的。

小结：

Redis 通过 MULTI、EXEC、DISCARD 和 WATCH 四个命令来支持事务机制。

命令	作用
MULTI	开启一个事务
EXEC	提交事务，从命令队列中取出提交的操作命令，进行实际执行
DISCARD	放弃一个事务，清空命令队列
WATCH	检测一个或多个键的值在事务执行期间是否发生变化，如果发生变化，那么当前事务放弃执行

小问题

Q：在执行事务时，如果 Redis 实例发生故障，而 Redis 使用了 RDB 机制，那么，事务的原子性还得到保证吗？

A：可能能，RDB一般没这么快生成，所以理论上可以回滚到上一个 RDB 的版本。