1. 为什么要用事务

我们知道Redis的单个命令是原子性的，比如get、set、mget、mset等指令。

原子性是指操作是不可分割的，在执行完毕之前不会被任何其它任务或事件中断，也就不会有并发的安全性问题

在涉及到多个命令的时候，如果需要把多个命令设置为一个不可分割的处理序列，就需要用到事务了。

比如，招财和陀螺各有100元，招财给陀螺转了10元，这时候需要在Redis中把招财的金额总数-10，同时需要把陀螺的金额总数+10。这两个操作要么同时成功，要么同时失败，这时候就需要事务了。

实际上，Redis连这个简单的需求都没办法完美做到，至于为啥，接着往下看吧

2. 事务的用法

2.1 5个基本指令

Redis提供了以下5个基本指令，先混个眼熟就行，接下来在案例中进行实操，想记不住都难

- MULTI
- EXEC
- DISCARD
- WATCH
- UNWATCH

命令	格式	作用	返回值
MULTI	MULTI	显式开启Redis事务，后续命令将排队，等候使用EXEC进行原子执行	always OK.
EXEC	EXEC	执行事务中的commands队列，恢复连接状态。如果WATCH在之前被调用，只有监测中的Keys没有被修改，命令才会被执行，否则停止执行（详见下文，CAS机制）	**成功：** 返回数组 —— 每个元素对应着原子事务中一个 command的返回结果; **失败：** 返回NULL (Ruby 返回nil)；
DISCARD	DISCARD	清除事务中的commands队列，恢复连接状态。如果WATCH在之前被调用，释放监测中的Keys	always OK.
WATCH	WATCH key [key ...]	将给出的Keys标记为监测态，作为事务执行的条件	always OK.
UNWATCH	UNWATCH	清除事务中Keys的监测态，如果调用了EXEC或者 DISCARD，则没有必要再手动调用UNWATCH	always OK.

2.2 案例演示

案例场景：招财和陀螺各有100元，招财给陀螺转了10元，这时候需要在Redis中把招财的金额-10，同时需要把陀螺的金额+10。

2.2.1 事务提交

我们首先为陀螺和招财初始化自己的金额；然后使用MULTI命令显式开启Redis事务。 该命令总是直接返回OK。此时用户可以发送多个指令，Redis不会立刻执行这些命令，而是将这些指令依次放入当前事务的指令队列中；EXEC被调用后，所有的命令才会被依次执行。

```
# 给陀螺初始化100元
127.0.0.1:6379> set tuoluo 100
OK
# 给招财初始化100元
127.0.0.1:6379> set zhaocai 100
OK
# 显式开启事务
127.0.0.1:6379> MULTI
OK
# 给陀螺增加10元
127.0.0.1:6379(TX)> INCRBY tuoluo 10
QUEUED
# 给招财减少10元
127.0.0.1:6379(TX)> DECRBY zhaocai 10
QUEUED
# 执行事务中的所有指令（提交事务）
127.0.0.1:6379(TX)> EXEC
1) (integer) 110
2) (integer) 90
```

2.2.2 嵌套事务

Redis不支持嵌套事务，多个MULTI命令和单个MULTI命令效果相同。

```
# 第一次开启事务
127.0.0.1:6379> MULTI
OK
# 尝试嵌套事务
127.0.0.1:6379(TX)> MULTI
(error) ERR MULTI calls can not be nested
# 仍然处于第一个事务当中
127.0.0.1:6379(TX)>
```

2.2.3 放弃事务

如果开启事务之后，中途后悔了怎么办？调用DISCARD可以清空事务中的指令队列，退出事务。

```
127.0.0.1:6379> MULTI
OK
# 在事务中调用DISCARD指令
127.0.0.1:6379(TX)> DISCARD
OK
# 会退出当前事务
127.0.0.1:6379>
```

2.2.4 watch指令

假如我们在一个客户端连接中开启了事务，另一个客户端连接修改了这个事务涉及的变量值，将会怎样？

client1	client2
127.0.0.1:6379> set tuoluo 100 OK 127.0.0.1:6379> set zhaocai 100 OK 127.0.0.1:6379> MULTI OK 127.0.0.1:6379(TX)> INCRBY tuoluo 10 QUEUED 127.0.0.1:6379(TX)> DECRBY zhaocai 10 QUEUED	
	127.0.0.1:6379> INCRBY tuoluo 10 (integer) 110
127.0.0.1:6379(TX)> EXEC 1) (integer) 120 2) (integer) 90	

client1开启了一个转账的事务，事务开始时招财和陀螺各自拥有100元，在执行EXEC指令之前，client2将陀螺的余额添加了10元，此时执行EXEC之后，陀螺最终的金额为120元，招财为90元。

很明显，这种情况下存在数据安全问题。

为此Redis提供了WATCH的指令，该指令可以为Redis事务提供CAS乐观锁行为，即多个连接同时更新变量的时候，会和变量的初始值进行比较，只在这个变量的值没有被修改的情况下才会更新成新的值。

2.2.4.1 WATCH用法

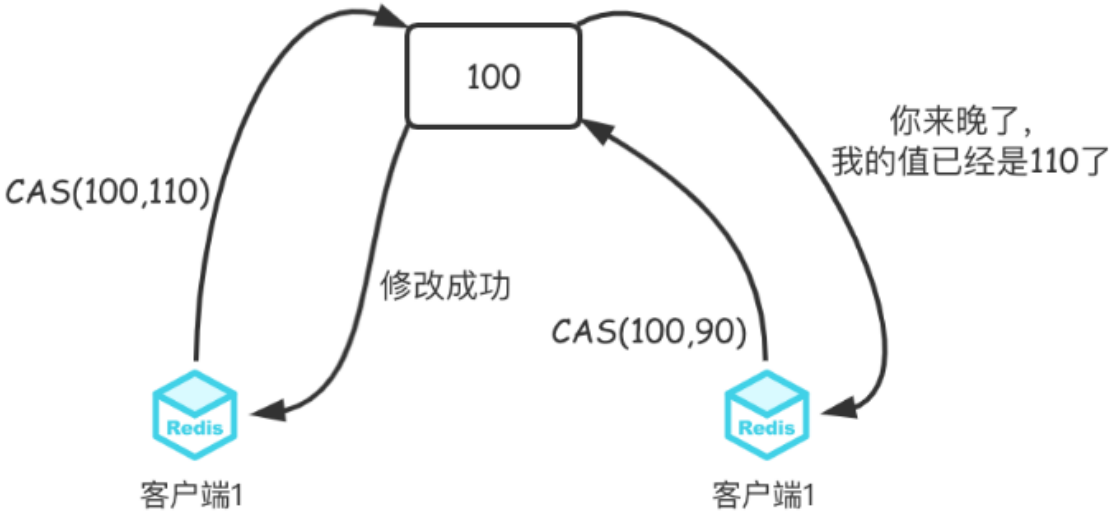
对应我们的案例，我们可以使用WATCH监听一个或多个key，如果开启事务之前，至少有一个被监视的key在EXEC执行之前被修改了，那么整个事务都会被取消，直接返回nil（见下面的案例）。UNWATCH是WATCH的反操作。

client1	client2
127.0.0.1:6379> set tuoluo 100 OK 127.0.0.1:6379> set zhaocai 100 OK # 为tuoluo添加乐观锁 127.0.0.1:6379> WATCH tuoluo OK 127.0.0.1:6379> MULTI OK 127.0.0.1:6379(TX)> INCRBY tuoluo 10 QUEUED 127.0.0.1:6379(TX)> DECRBY zhaocai 10 QUEUED	
	127.0.0.1:6379> INCRBY tuoluo 10 (integer) 110
# EXEC指令报错了，事务执行失败！！！！！ 127.0.0.1:6379(TX)> EXEC (nil)	

2.2.4.2 CAS机制

CAS（Compare And Swap）比较并替换，是多并发时常用的一种乐观锁技术

CAS需要三个变量信息，分别是内存位置(JAVA中内存地址的值，V)，旧的预期值(A)和新值(B)。CAS执行时，当且仅当V和预期值A相等时，更新V的值为新值B，否则不执行更新。



3. 事务执行出错怎么办

事务执行时可能遇到问题，按照发生的时机不同分为两种：

- 执行EXEC之前
- 执行EXEC之后

3.1 执行EXEC之前发生错误

比如指令存在语法错误（参数数量不对，指令单词拼错）导致不能进入commands队列，这一步主要是编译错误，还未到运行时。

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379(TX)> SET tuoluo
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379(TX)> EXEC
(error) EXECABORT Transaction discarded because of previous errors.
```

这种情况下事务会执行失败，队列中的所有指令都不会得到执行。

3.2 执行EXEC之后发生错误

这种错误往往是类型错误，比如对String使用了Hash的命令，这是运行时错误，编译期间不会出错

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379(TX)> SET tuoluo 100
QUEUED
127.0.0.1:6379(TX)> LPOP tuoluo
```

```
QUEUED
127.0.0.1:6379(TX)> EXEC
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
```

我们发现，SET tuoluo 100的命令居然执行成功了，也就是在发生了运行时异常的情况下，错误的指令不会被执行，但是其他的命令不会受影响。

这种方式显然不符合我们对原子性的定义，也就是Redis的事务无法实现原子性，无法保证数据一致。

针对这种缺陷，Redis官方也是做了说明的。

4. Redis事务为什么不支持回滚

引自Redis官方文档。

If you have a relational databases background, the fact that Redis commands can fail during a transaction, but still Redis will execute the rest of the transaction instead of rolling back, may look odd to you.

However there are good opinions for this behavior:

- Redis commands can fail only if called with a wrong syntax (and the problem is not detectable during the command queueing), or against keys holding the wrong data type: this means that in practical terms a failing command is the result of a programming errors, and a kind of error that is very likely to be detected during development, and not in production.
- Redis is internally simplified and faster because it does not need the ability to roll back.

An argument against Redis point of view is that bugs happen, however it should be noted that in general the roll back does not save you from programming errors. For instance if a query increments a key by 2 instead of 1, or increments the wrong key, there is no way for a rollback mechanism to help. Given that no one can save the programmer from his or her errors, and that the kind of errors required for a Redis command to fail are unlikely to enter in production, we selected the simpler and faster approach of not supporting roll backs on errors.

为了方便大家理解，我翻译一下就是：

你们程序员的锅，关我们Redis屁事儿！

Redis官方认为，只有在命令语法错误或者类型错误的时候，Redis命令才会执行失败。而且他们认为有这种错误的语法一般也不会进入到生产环境。而且不支持回滚可以使他们有更多时间玩儿Redis运行得更简单快捷。

这种说法多牛！如果出问题就是程序员的问题，写错了还让代码进入生产环境，那就是罪上加罪，你永远赖不着Redis官方。

这可能就是不推荐使用Redis事务的原因了吧，鸡肋是一方面，万一被官方打脸了呢？所以Redis事务的知识稍微了解一下就好，面试被问到能回到上来就可以了。