# Redis实现分布式锁

## 初级版本：

**public class SimpleRedisLock implements ILock {**

**// key的统一前缀**

**private static final String KEY\_PREFIX = "lock:";**

**private StringRedisTemplate stringRedisTemplate;**

**// 业务名**

**private String name;**

**public SimpleRedisLock(StringRedisTemplate stringRedisTemplate, String name) {**

**this.stringRedisTemplate = stringRedisTemplate;**

**this.name = name;**

**}**

**@Override**

**public boolean tryLock(long timeoutSec) {**

**// 获取锁**

**String key = KEY\_PREFIX + name;**

**/\*\***

**\* value值最好不要随便设，因为后面要判断哦，但是一般设置成一个有意义的值比较好！**

**\* 同一个JVM内，线程ID是唯一的**

**\*/**

**long threadId = Thread.currentThread().getId();**

**String value = threadId + "";**

**// 获取锁**

**Boolean isSuccess = stringRedisTemplate.opsForValue()**

**.setIfAbsent(key, value, timeoutSec, TimeUnit.SECONDS);**

**// 返回获取锁是否成功结果**

**return BooleanUtil.isTrue(isSuccess);**

**}**

**@Override**

**public void unlock() {**

**stringRedisTemplate.delete(KEY\_PREFIX + name);**

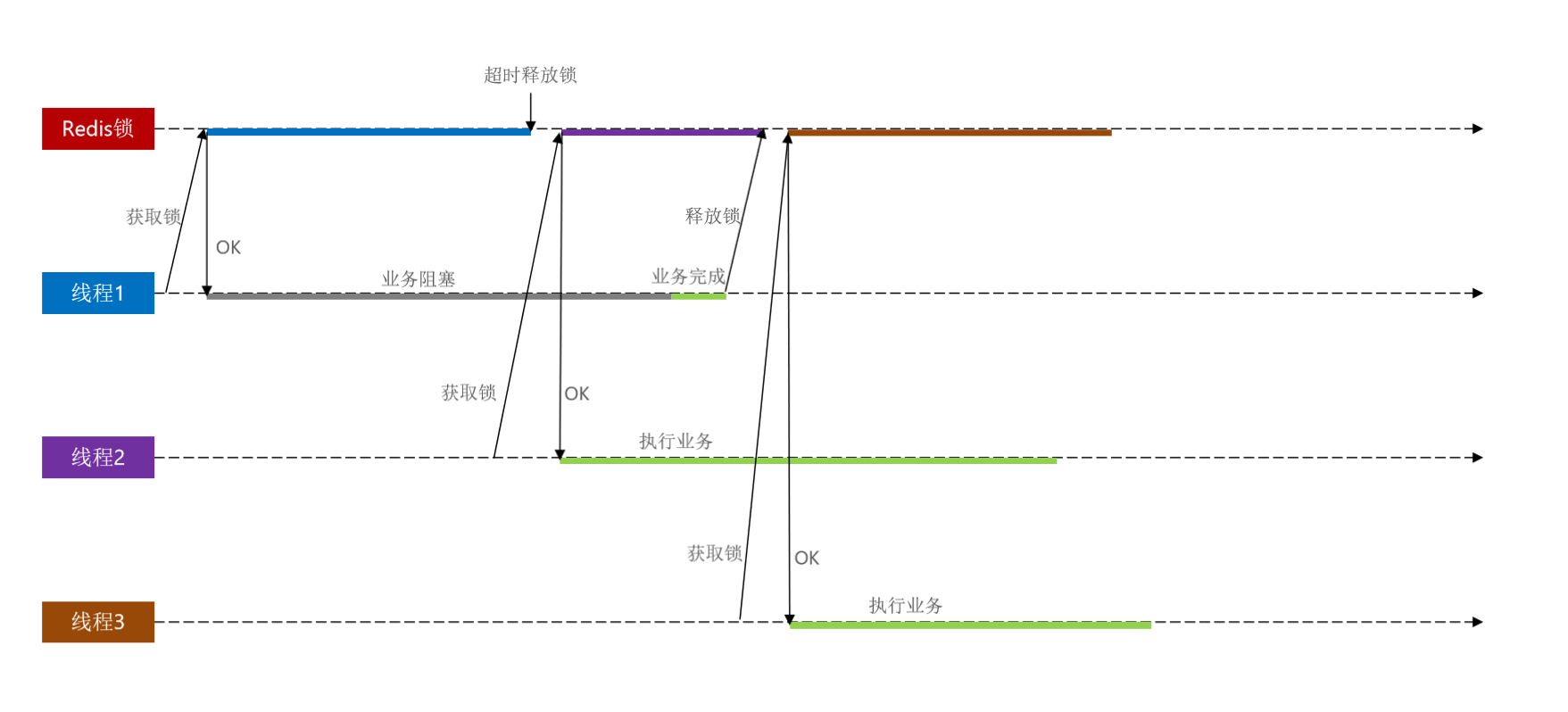
**}**

**}**

## 初步版本问题：

### 问题原因

初步版本获取锁采用的是nx与ex的方式实现互斥，释放锁的时候直接delete把锁删掉，其他线程就能获取锁了，实现方式非常简单。在大多数情况下这个锁都能正常工作，即当前版本的Redis分布式锁在多数情况下都是可以正常使用的，但是在一些极端情况下仍然会存在问题！比如获取锁的线程一因为某种原因阻塞了，如果这个线程一阻塞的时间太长了，甚至可能超过锁的超时释放时间，那么线程二就能乘虚而入获取这个释放掉的锁，就当线程二获取锁后，线程一醒来执行业务并在业务执行完后把锁释放掉，此时是线程一把线程二的锁释放了！就在这时，线程三来了，那么线程三又能获取锁.......于是就又发生线程安全的问题了！！！



详解上面极端情况：

1：线程1获取锁成功后，执行业务发生阻塞，当阻塞时间超过锁的超时时间，锁触发了超时释放，那么此时其他线程就能成功获取锁；

2：比如线程2获取锁成功，而此时线程1阻塞结束，继续执行业务，线程1执行完之后释放锁，此时线程1释放的是线程2的锁，此时的线程2还在执行自己的业务，线程2是不知道自己的锁被释放了的；

3：就在这时，又有线程3进来获取锁，因为锁已经被线程1释放，此时线程3是能成功获取锁的

4：显而易见，此时此刻，有两个线程都拿到了锁，都在执行业务，所以，又一次出现了并行执行的情况，线程安全的问题有一次发生！

总结：极端情况发生的原因就是：

线程1持有的锁由于线程1在执行业务时发生阻塞，触发超时释放，导致其他线程可以成功获取锁；

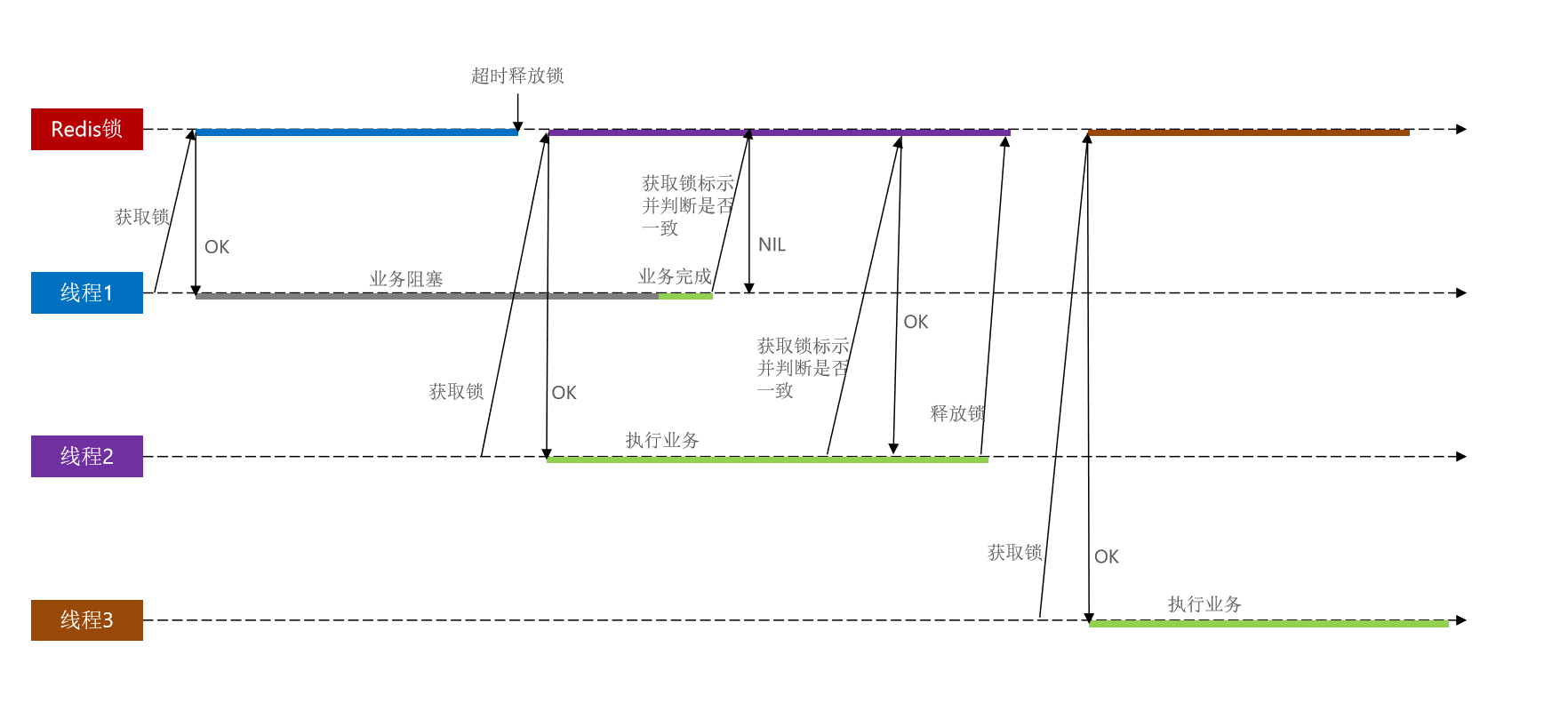
当线程1阻塞结束执行完业务去释放锁时，释放的锁是不是自己的锁，而是其他线程的锁；

所以归根结底，发生线程安全问题最重要的原因就是因为线程1因为阻塞在释放锁的时候，把别人的锁释放掉了！

### 解决方法：

在释放锁的时候，先判断释放的锁是不是自己的！即在释放锁时，判断锁标识是否一致！避免误删别人的锁！

注意这种解决方法是解决锁误删问题，但是很明显线程1和线程2的执行导致线程安全问题，即同一个用户线程1可以下单成功，线程2也可以下单成功！

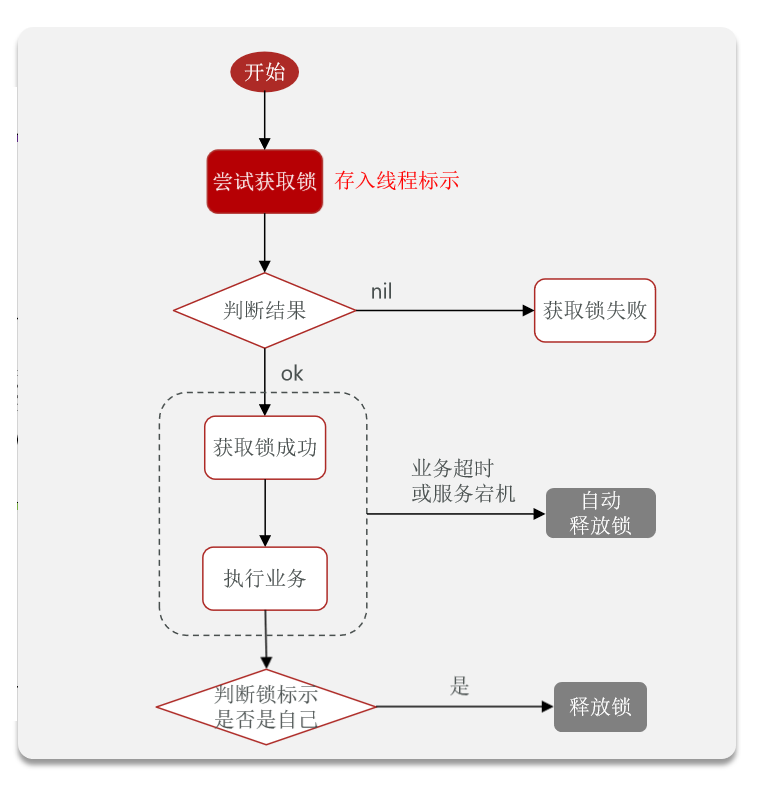


## 改正初步版本

在获取锁时存入线程标示，可以用UUID表示线程标识，为什么不用**Thread.currentThread().getId()**获取到的线程表示，因为不同JVM获取到的线程id可能一样，但是UUID却可以保证跨JVM唯一，你不放心甚至可以线程ID凭接UUID作为线程标识，即UUID标识不同JVM，线程ID标识JVM中不同线程；在释放锁时先获取锁中的线程标示，判断是否与当前线程标示一致，如果一致则释放锁，如果不一致则不释放锁。

注意线程标识不要简单理解成**Thread.currentThread()**，就是只要可以标识出当前执行方法的线程都可以做线程标识！

此时的实现分布式锁的流程图变成下面：



### 代码实现：

**public class SimpleRedisLock implements ILock {**

**/\*\***

**\* key的统一前缀**

**\*/**

**private static final String KEY\_PREFIX = "lock:";**

**/\*\***

**\* 获取机器的UUID**

**\*/**

**private static final String ID\_PREFIX = UUID.randomUUID().toString(true) + "-";**

**private StringRedisTemplate stringRedisTemplate;**

**/\*\***

**\* 锁名**

**\*/**

**private String name;**

**public SimpleRedisLock(StringRedisTemplate stringRedisTemplate, String name) {**

**this.stringRedisTemplate = stringRedisTemplate;**

**this.name = name;**

**}**

**@Override**

**public boolean tryLock(long timeoutSec) {**

**// 获取锁**

**String key = KEY\_PREFIX + name;**

**/\*\***

**\* 线程标识：UUID + 线程ID**

**\*/**

**String threadId = ID\_PREFIX + Thread.currentThread().getId();**

**// 获取锁**

**Boolean isSuccess = stringRedisTemplate.opsForValue()**

**.setIfAbsent(key, threadId, timeoutSec, TimeUnit.SECONDS);**

**// 返回获取锁是否成功结果**

**return BooleanUtil.isTrue(isSuccess);**

**}**

**@Override**

**public void unlock() {**

**String key = KEY\_PREFIX + name;**

**// 获取线程ID**

**String threadId = ID\_PREFIX + Thread.currentThread().getId();**

**// 获取缓存的线程标识**

**String cacheThreadId = stringRedisTemplate.opsForValue().get(key);**

**// 判断线程标识是否一致**

**if (threadId.equals(cacheThreadId)) {**

**// 释放锁**

**stringRedisTemplate.delete(key);**

**}**

**}**

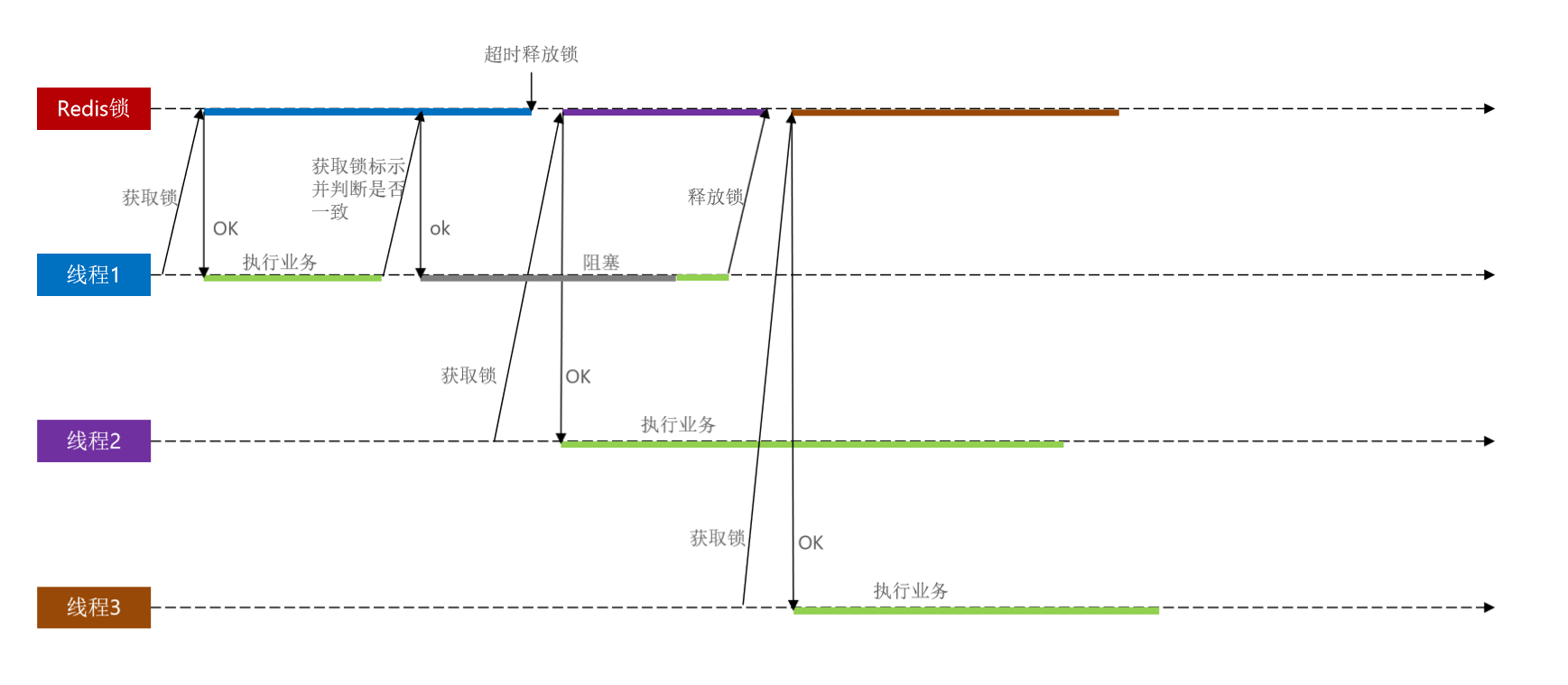
**}**

## 改进版本还是存在问题

### 问题原因

上面的改进版本是解决Redis分布式锁的误删问题，解决的思路就是在获取锁的时候存入线程标识，在释放锁的时候不能上来就删除，而是判断缓存的线程标识和当前线程的标识是否一致，如果一致，说明锁是线程自己的，再去删除。

不过改进版本在极端情况下还是有问题！因为从redis中获取锁标识去判断锁标识是否一致，与释放锁不是原子操作，在极端情况下就会产生问题，比如，线程一判断锁标识是否一致后，在执行释放锁的时候发生了阻塞，而阻塞的时候如果足够长，很有可能会触发锁的超时释放，锁一旦超时释放，那么此时别的线程又可以乘虚而入成功获取锁，而当线程二获取锁成功后，线程一阻塞结束醒来去执行释放锁的操作，于是就把线程二的锁释放掉了，此时线程三又可以乘虚而入获取锁，那么再一次发生线程安全问题！！！



上图存在问题详解：

1：线程1在执行完业务去执行释放锁动作时，因为释放锁包括两步：获取锁标识并判断是否一致 与 释放锁，如果在判断完锁标识完成后，还没释放锁时，发生了阻塞，如果这个阻塞时间足够长，很有可能触发锁的超时释放；

2：由于线程1的锁超时释放，其他线程就又可以获取锁了，如线程2获取锁成功，而此时线程1阻塞结束，去执行释放锁动作，因为线程1之前已经判断过锁标识了，所以释放的是线程2的锁，又一次发生误删；

3：此时又有线程3进来，那么线程3可以获取锁成功，于是并发安全问题又发生了。

总结：此时锁误删的原因是释放锁的：获取锁标识并判断是否一致 与 释放锁 是两个动作，这两个动作之间发生阻塞，最后出现问题！

### 解决方法：

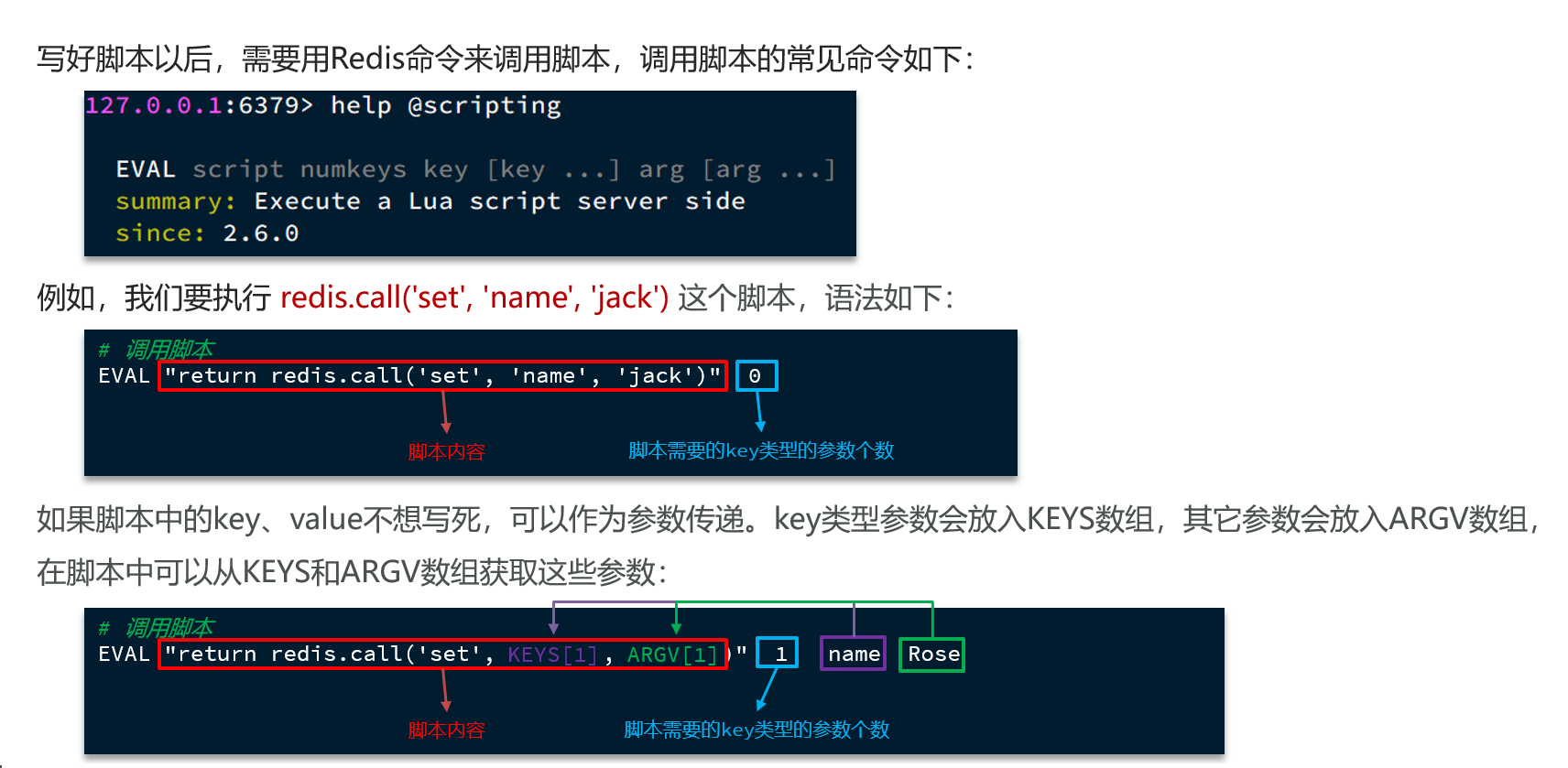
以上锁误删问题的出现是因为判断锁标识和释放锁是两个动作，这两个动作之间产生了阻塞，导致锁超时释放，从而产生线程安全问题！

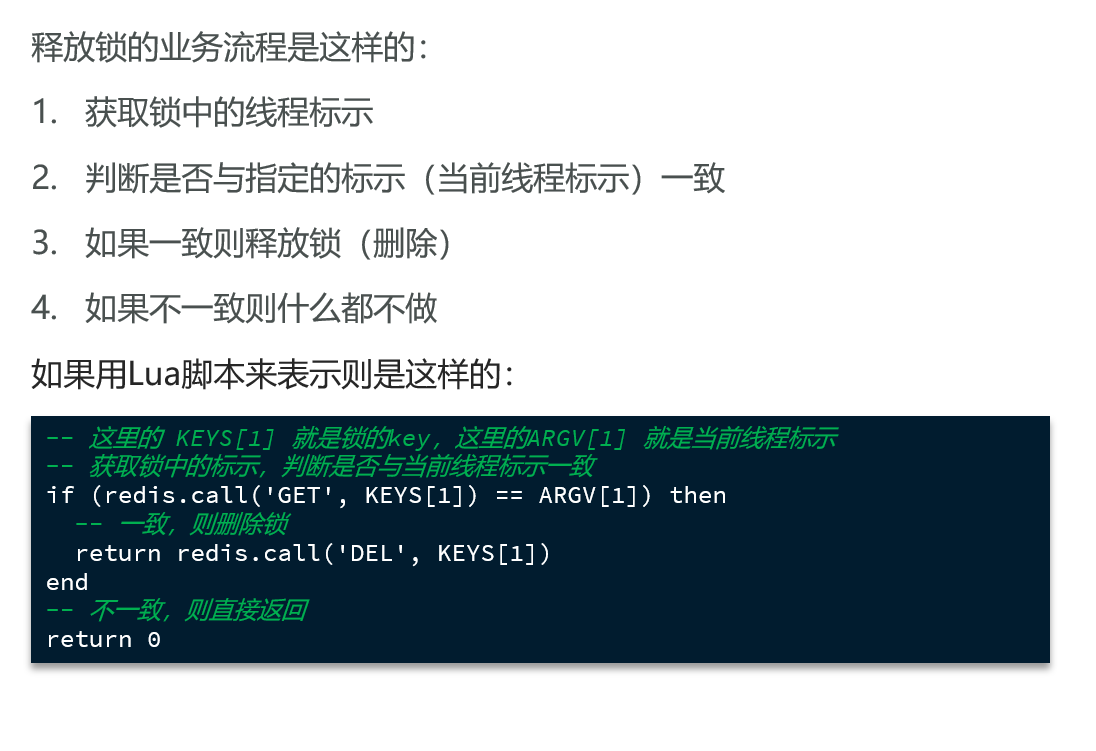
因此，要想避免这种情况，必须确保判断锁标识的和释放锁的两个动作得成原子性操作，一起执行，不能出现间隔！即释放锁方法中的中“获取锁标识并判断是否一致”与“释放锁”必须是一个原子性的操作，即一起执行，不能出现间隔！请看下面的Lua脚本实现。

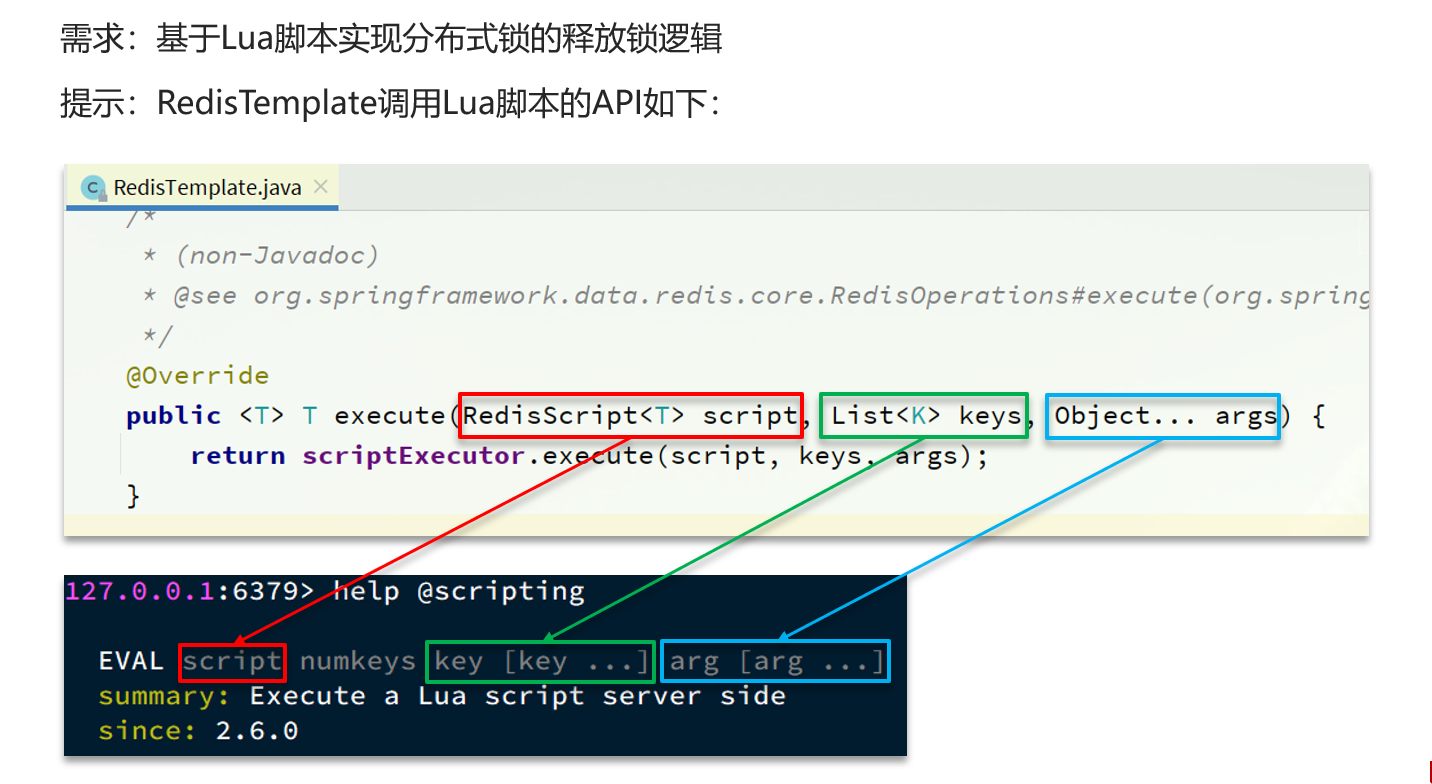
## Redis的Lua脚本

Redis提供了Lua脚本功能，在一个脚本中编写多条redis命令，确保多条命令执行时的原子性。









基于Redis的分布式锁实现思路：

获取锁时，利用set nx ex获取锁，设置过期时间，保存线程标识；

释放锁时，先判断县城表示是否与自己一致，一致则删除锁

目前基于Redis的分布式锁的版本具有了以下特性：

1：利用set nx满足互斥性；

2：利用set ex保证故障时锁依然能释放，避免死锁，提高安全性；

3：利用redis集群保证高可用和高并发特性。

## 以上改进的总结

说到底就一句话；就是让锁误删的难度越来越大：

一是让线程在释放锁操作的时候，判断当前线程的线程标识与缓存的线程标识是否一致，一致才删除锁。

二是避免锁释放时间太久，即让释放锁方法中的中“获取锁标识并判断是否一致”与“释放锁”变成一个原子性的操作，避免阻塞。

以上问题的起源都是因为阻塞的原因触发锁的超时释放，然后导致锁的误删。

但是你可以看到，以上都是改进让阻塞线程1醒来后不去误删别的线程的锁，但是以上锁误删的改进也不是完美解决一人一单线程安全问题的，还是有可能存在线程安全问题哦。即线程1和线程2可能都下单成功！