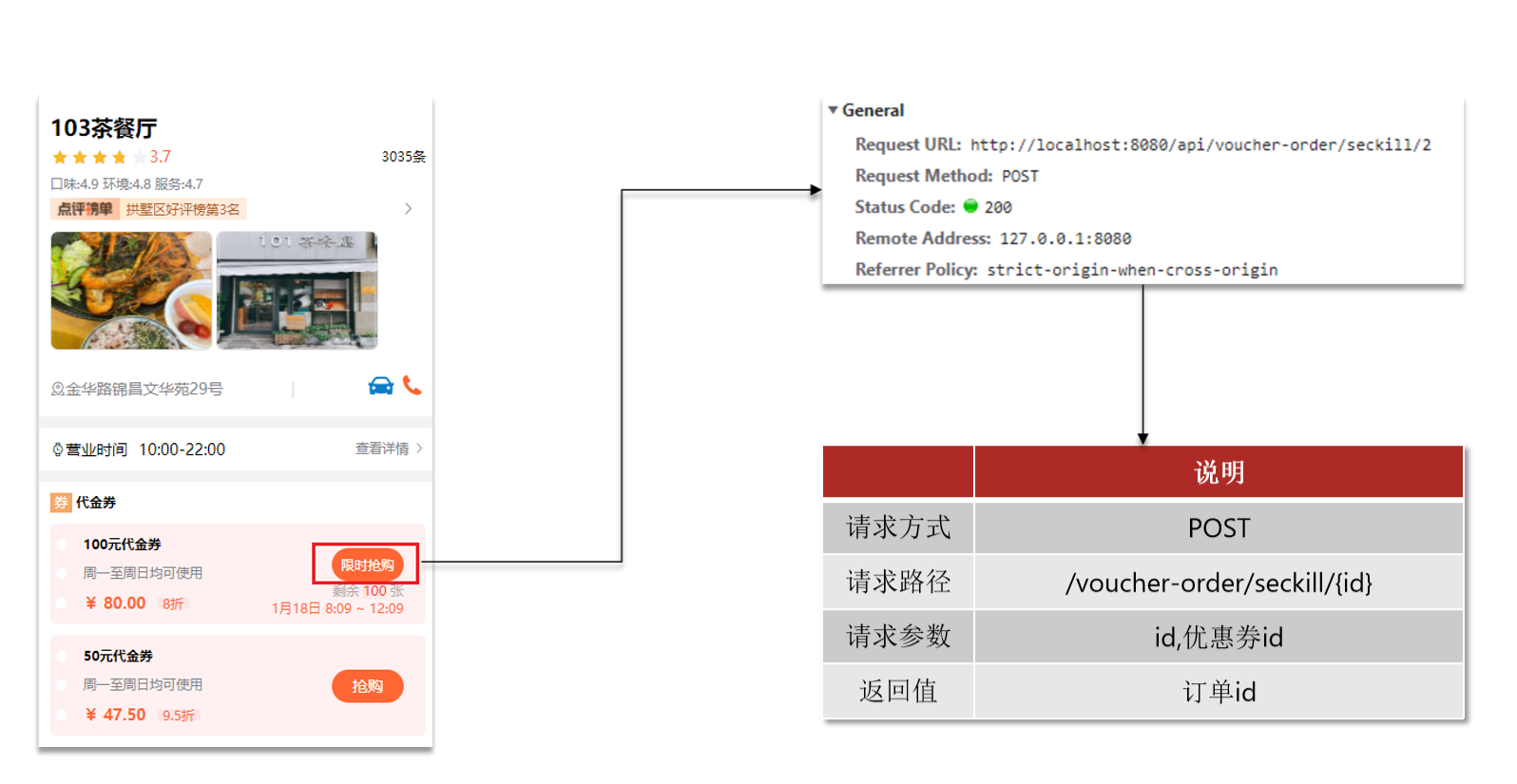
# 实现优惠卷秒杀下单

上面已经解决了下单时订单ID唯一性问题。接下来我们自然可以是新优惠券秒杀下单功能了。

每个店铺都可以发布优惠券，分为平价券和特价券。平价券可以任意购买，而特价券/秒杀券数量有限，需要秒杀抢购。



## 秒杀优惠券下单过程：



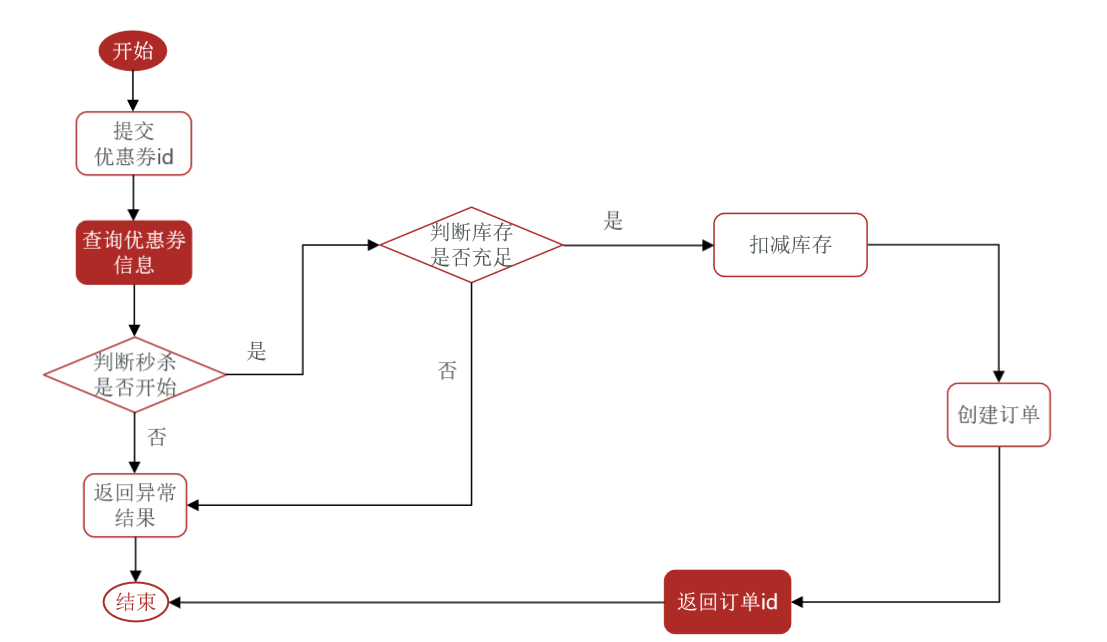
## 秒杀优惠券下单流程图

下单时需要注意三点：

1：秒杀是否开始或结束，如果尚未开始或者已经结束则无法下单；

2：库存是否充足，不足则无法下单；

3：生成订单，扣减库存。



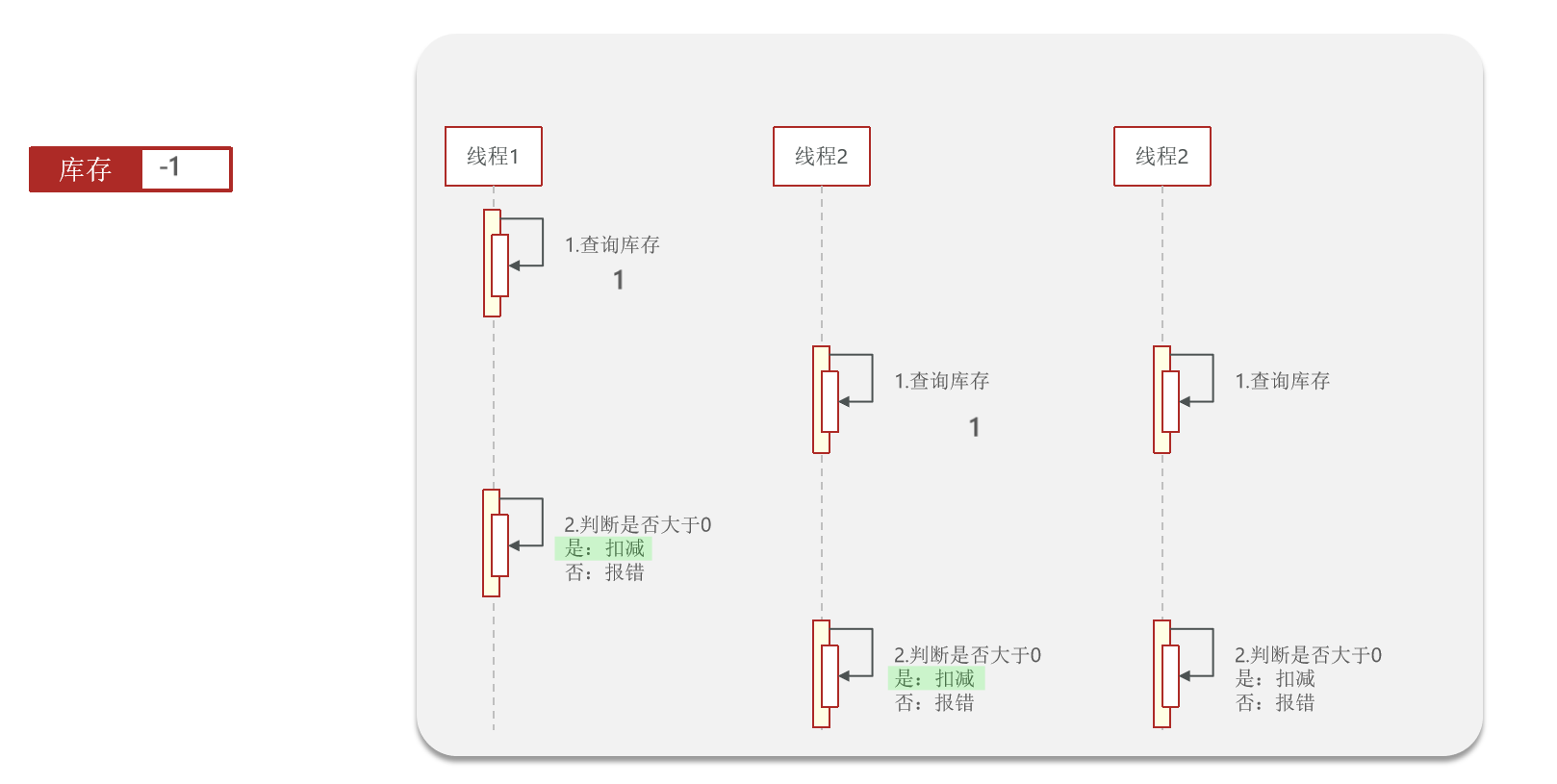
## 超卖问题

上面实现的优惠券秒杀下单功能的原始版本，并且测试发现确实可以成功下单。但是这个测试是我们在页面上自己点击来去测试的，这跟真实的秒杀场景还是有很大差异的，因为真实的秒杀场景下，肯定是有无数的用户一起来去抢购，一起下单的，就是说一瞬间的并发量可能达到每秒钟数十万数百万，甚至数千万。于是在这种情况下，我们的接口还能不能正常工作呢？

原始版本存在的问题：高并发情况下会出现超卖问题！即多线程中的并发安全问题！

### 超卖问题举例（当然是极端例子）：

秒杀卷的库存只剩下1了，这个时候有100个线程来抢购秒杀券，线程1到线程100都从数据库中查询到秒杀券，这个时候线程1到线程100都可以去修改库存，出现超卖！超卖问题说到底就是多线程并发安全问题。



### 超卖问题解决方案：

超卖问题是典型的多线程安全问题，相信每个程序员能可以想到的解决方案就是加锁。没错，解决方案就是加锁。

但是加锁还是有很多不同的诀窍和姿势的。我们知道锁分为两类，一种是悲观锁，一种是乐观锁。悲观锁和乐观锁并不是一种真正的锁，只是一种锁设计的理念。

#### 悲观锁

悲观锁认为线程安全问题一定会发生，因此在操作数据之前先获取锁，确保线程串行执行，同学们，如果多个线程是串行执行，是不是就不会出现线程安全问题了呀，没错，是这样的！悲观锁的设计理念就是既然多线程并发有安全问题，那你就不让多线程并发了呗。

正因为如此，悲观锁的性能就很差！因为你不管有多少线程都只能是一个一个地去执行，所以在高并发的场景下，解决线程安全用悲观锁并不适合。

悲观锁例如：Synchronized、Lock都属于悲观锁。

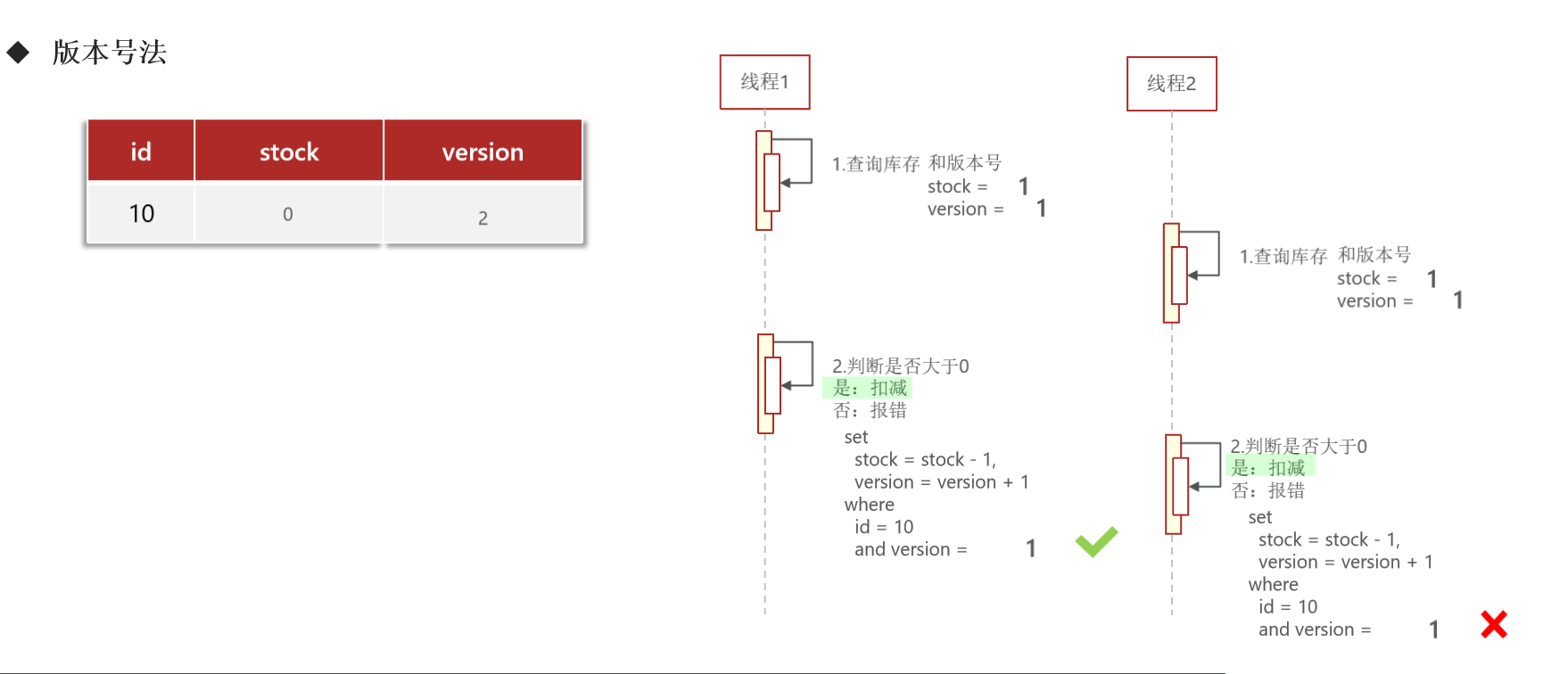
#### 乐观锁

乐观锁认为线程安全问题不一定会发生，因此不加锁，你可能会说了，我不加锁怎么保证线程安全呢？答：乐观锁认为线程安全问题发生可能性是比较低的，多数情况下是不会发生的，你没有必要上来就加锁。你为了极少数不安全情况而把所有的线程都串行了，显然是得不偿失的。保证数据安全只是在更新数据的时候去判断有没有其他线程对数据做了修改。

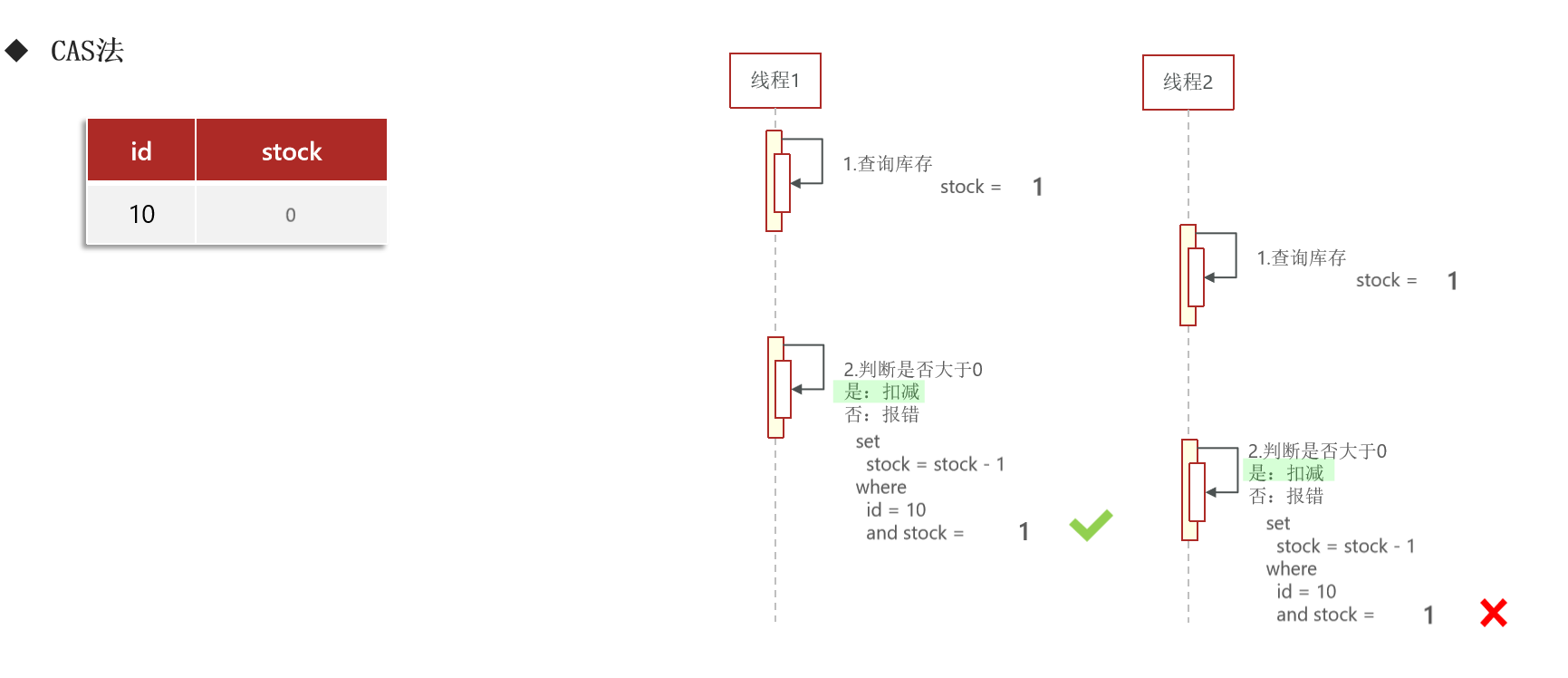
如果没有修改则认为是安全的，自己才更新数据；如果已经被其他线程修改说明发生了安全问题，此时可以重试或异常。比方说下单时要去更新库存，那么更新库存时判断一下有没有别人修改库存！如果和我之前查到的库存不一样，说明有人修改了。

因为乐观锁这个方案不用去加锁，而是在执行时再去做判断，因此性能比悲观锁要好很多。但是乐观锁的关键是判断之前查询得到的数据是否有被修改过，常见的方式有两种：

##### 版本号法：增加一个版本version字段



##### CAS法（比较并交换）：直接判断数据



超卖问题：乐观锁解决，即修改共享数据时判断共享数据是否被其他线程修改过。有版本号法，或者CAS（比较并交换 OR 比较并设置）法。

##### 乐观锁弊端：

失败率太高，因为以为共享数据只要有其他线程改了，就会有线程安全问题，实际上，有些情况下就算产生并发修改，但在业务上也没有安全问题，比如此案例中的就是这种情况，stock > 0 就可以了。

比如刚开始卖的时候库存还剩100，无数个请求都涌入进来下单，因为我们是乐观锁，也就是没加锁，比如50个并发下单请求都查到库存100，这个时候只有一个线程能去执行扣减库存的动作，那么剩下的49个库存就都无效失败了，于是下单失败率大大提高。但是从业务角度看剩下的49个请求完全没问题的，因为库存还有啊！因为业务上来讲只要库存>0就都可以下单！

### 总结：超卖这样的线程安全问题，解决方案有哪些？

#### 悲观锁：添加同步锁，让线程串行执行

优点：简单粗暴

缺点：性能一般

#### 乐观锁：不加锁，在更新时判断是否有其它线程在修改

优点：性能好

缺点：存在成功率低的问题

## 一人一单

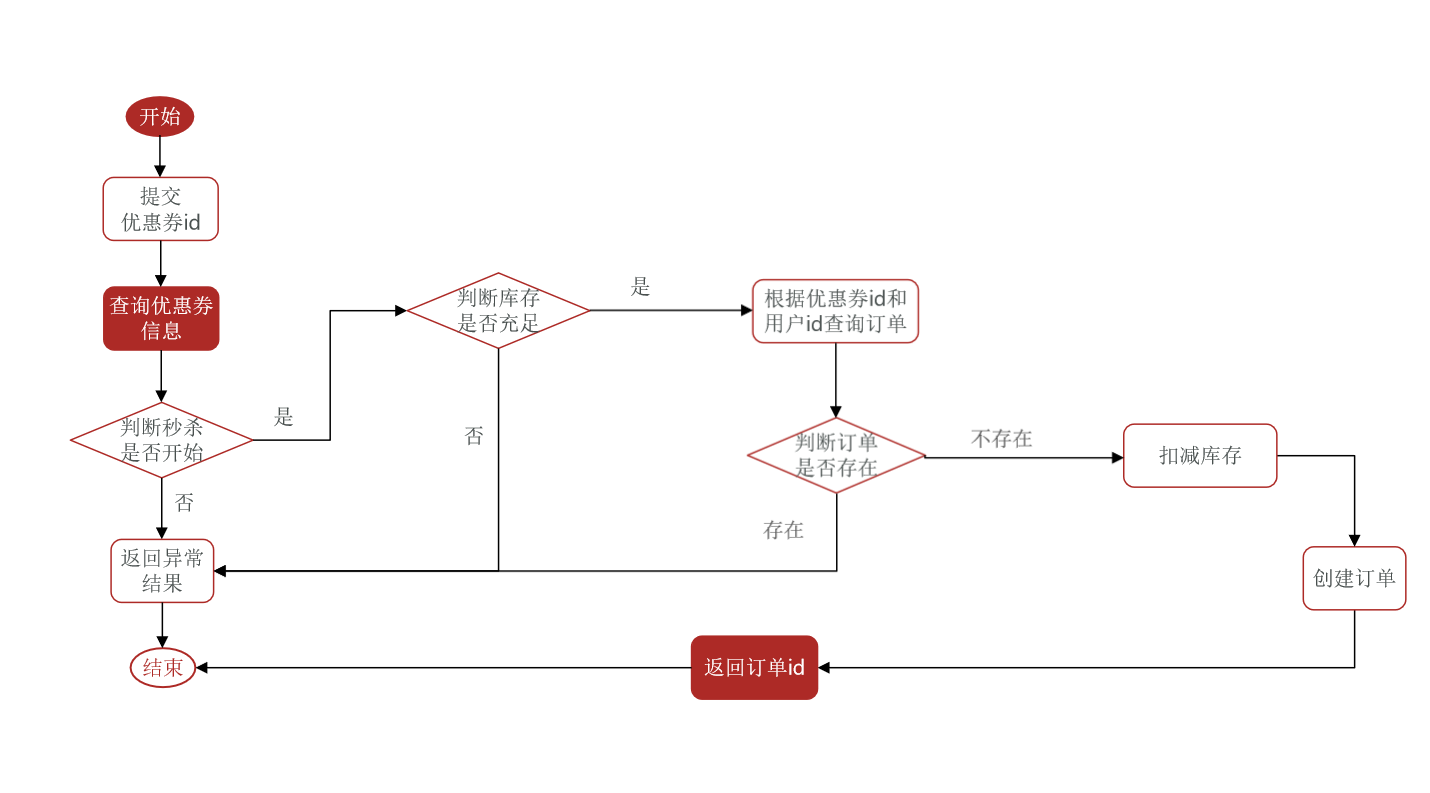
需求：修改秒杀业务，要求同一个优惠券，一个用户只能下一单。

为什么有这样的需求？像这样的特价券、秒杀券，优惠力度特别大，商家明明很可能是赔本的，这种券的目的是什么？目的只有一个，就是利用这种券吸引更多的用户来店铺体验，体验了以后口碑好就能传播出去吸引更多的用户来！即拉新!

但是当前的业务中，，同一个用户可以无限下单，，这不就是黄牛行为吗！这肯定是不可以的！必须要求同一个用户对于同一个券只能下一单！

### 解决方案

怎么做到一人一单呢？其实思路非常简单：就是用户下单的时候，在订单表中判断一下在订单表中是否存在当前用户id和特价券id的订单！存在证明用户已经下过单了！



### 存在线程安全问题

测试发现不能完全解决一人一单问题，虽然情况没有那么严重！为什么？还是线程安全问题！举例（当然也是极端情况）：

比如当同一个没下过单的用户发送了100次下单请求，这100个请求做一人一单去查询订单表，确实都没查到订单，那么都可以下单！于是就产生一个人可以下多个订单的情况？

### 解决线程安全问题：

加锁，要加悲观锁！乐观锁必然不行，因为乐观锁方案是在更新数据时用的，而此时场景是插入/新增数据，你不能判断一下数据有没有修改过，没有办法判断。

然后悲观锁加在哪里？答：在查询、判断订单到新增订单的整块逻辑上加悲观锁！

**/\*\***

**\* 事务范围只需要更新数据库的范围，即减库存和新增订单，查询不需要事务，所以加在创建订单这里就行**

**\*/**

**@Transactional**

**public Result createVoucherOrder(Long voucherId) {**

**/\*\***

**\* 悲观锁不建议加在方法上，加在方法上意味着所有线程都是同一把锁：this，那么这个创建订单方法就变成**

**\* 串行执行了，性能就变得很差了，所谓的一人一单，是指同一个用户来了我们才去判断并发安全问题，如果不是**

**\* 同一个用户，就不需要去加锁。因此，可以得出两个结论：一是悲观锁不应该加在方法上，二是锁的范围因该限定**

**\* 为同一用户！**

**\* 因此：锁应该是当前用户，我们可以把用户ID作为锁，把锁的范围缩小，也就是说同一个用户加同一把锁，**

**\* 不同用户加不同的锁。**

**\*/**

**// 一人一单：查询订单，判断该用户是否已经抢购过该秒杀特价券**

**Long userId = UserHolder.getUser().getId();**

**/\*\***

**\* 使用Long.toString()不能保证Long类型的同数值转成的字符串是一致的，不能保证一致，那么这个锁就不是同一把锁。**

**\* 为了保证Long类型的同数值转成的字符串是一致的，使用要使用String.intern()方法，作用是返回字符串对象的规范表示。**

**\*/**

**synchronized (userId.toString().intern()) {**

**int count = query().eq("user\_id", userId).eq("voucher\_id", voucherId).count();**

**if (count > 0) {**

**// 用户下过单则不能再下单**

**return Result.fail("用户已经购买过一次！");**

**}**

**// 扣减库存：update table set stock = stock -1 where voucher\_id = 2 and stock > 0;**

**boolean isSuccess = seckillVoucherService.update()**

**.setSql("stock = stock -1")**

**.eq("voucher\_id", voucherId)**

**.gt("stock", 0)**

**.update();**

**if (!isSuccess) {**

**return Result.fail("扣减库存失败！");**

**}**

**// 创建订单存入数据库**

**VoucherOrder voucherOrder = new VoucherOrder();**

**long orderId = redisIdWorker.nextId(VOUCHER\_ORDER\_KEY);**

**voucherOrder.setId(orderId);**

**voucherOrder.setUserId(userId);**

**voucherOrder.setVoucherId(voucherId);**

**save(voucherOrder);**

**// 返回订单ID**

**return Result.ok(orderId);**

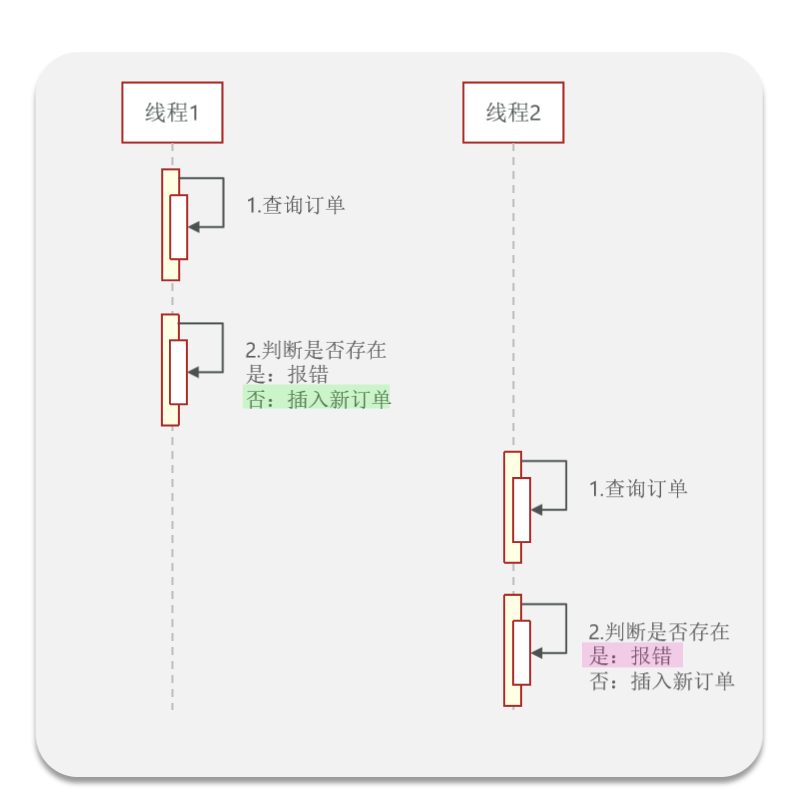
**}**

**}**

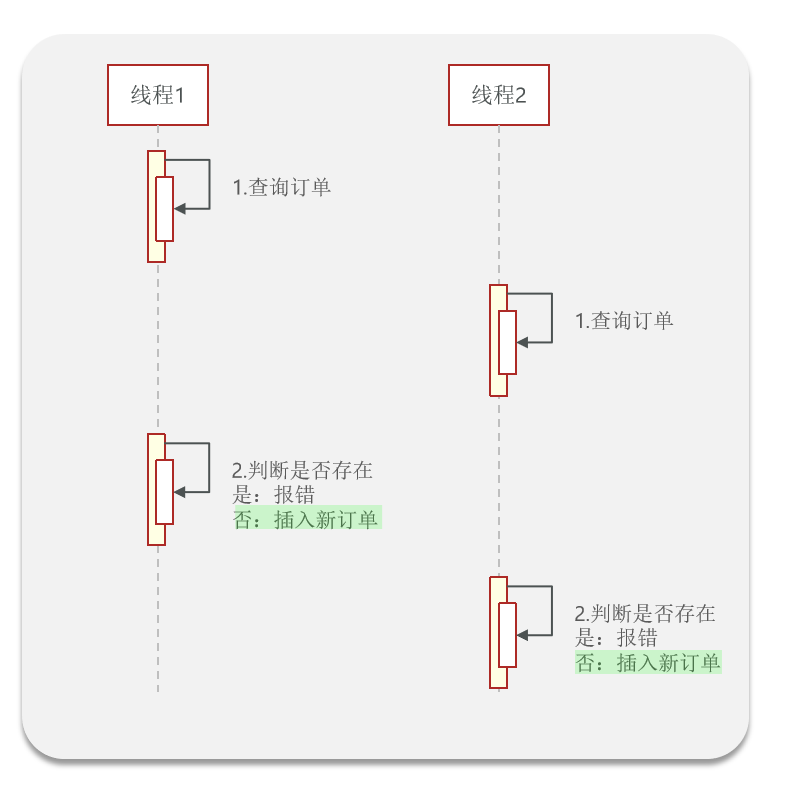
## 一人一单的并发安全问题：

### 单体模式下一人一单并发安全问题：

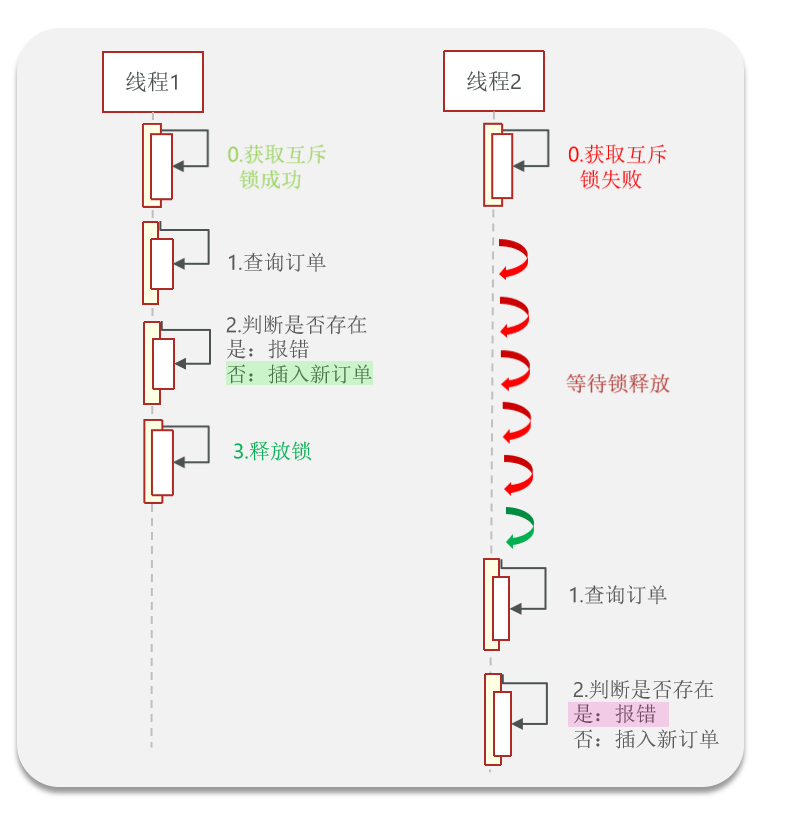
#### 正常情况：



#### 异常情况：



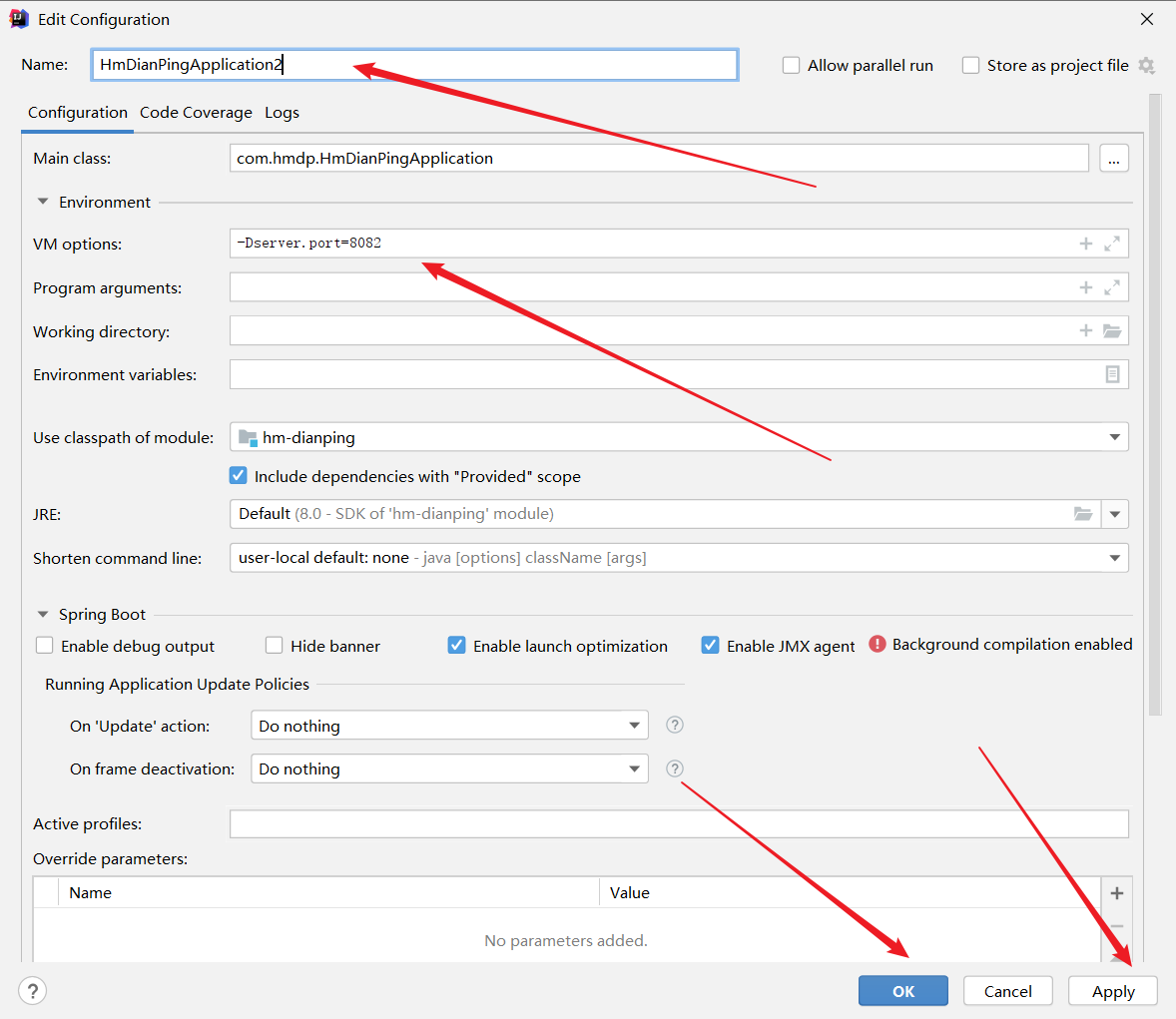
### 解决单体模式一人一单问题：



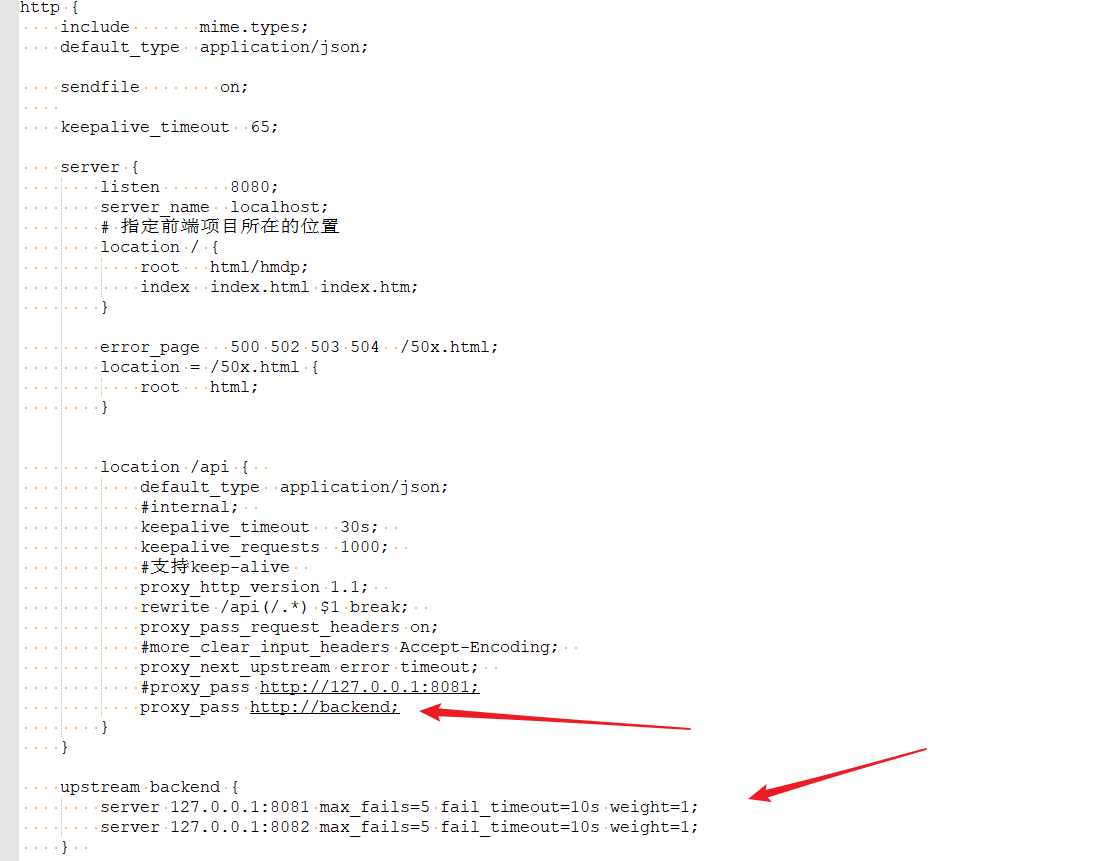
以上已经演示了秒杀下单的一人一单功能，并且基于悲观锁synchronized解决了一人一单的并发安全问题。不过这种解决方案仅仅适合在单机情况下使用，我们的项目虽然是单体项目，但是将来如果有高并发场景，我们可以做集群部署的，就是把一个项目部署到多个机器，形成负载均衡的集群。而在集群模式下，我们使用悲观锁synchronized就不能解决一人一单的并发安全问题了。

### 模拟集群模式下一人一单并发安全问题：

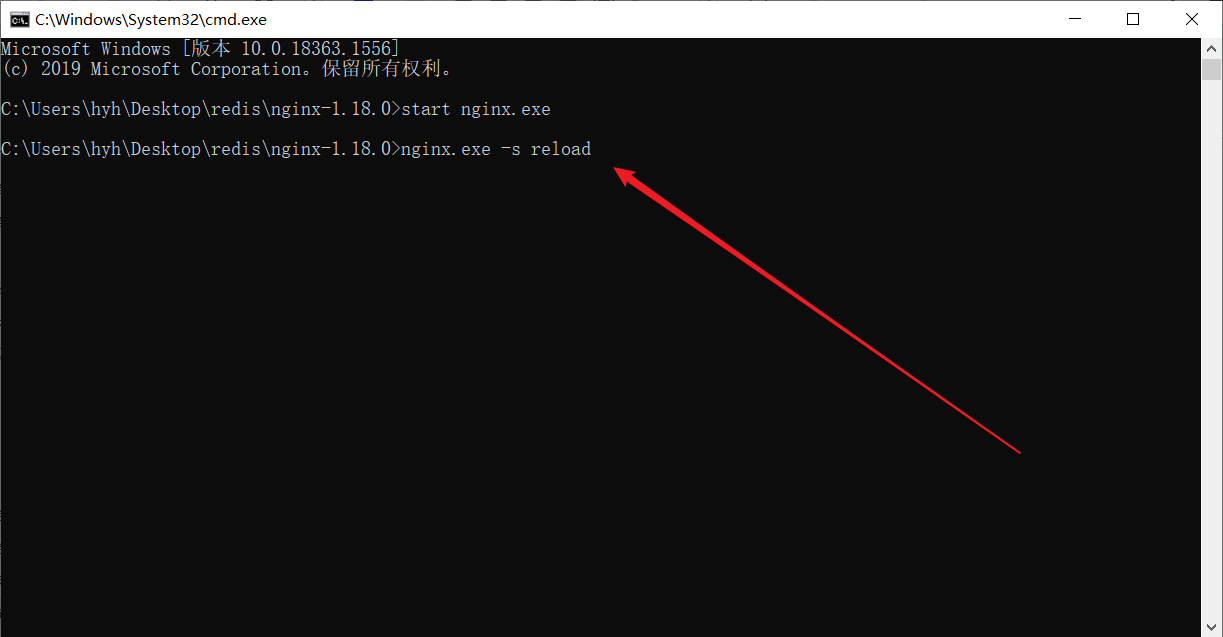
#### 1：我们将服务启动两份，端口分别为8081和8082



#### 2：然后修改nginx的conf目录下的nginx.conf文件，配置反向代理和负载均衡



#### 3：重启nginx



#### 4:：浏览器输入http://localhost:8080/api/voucher/list/1

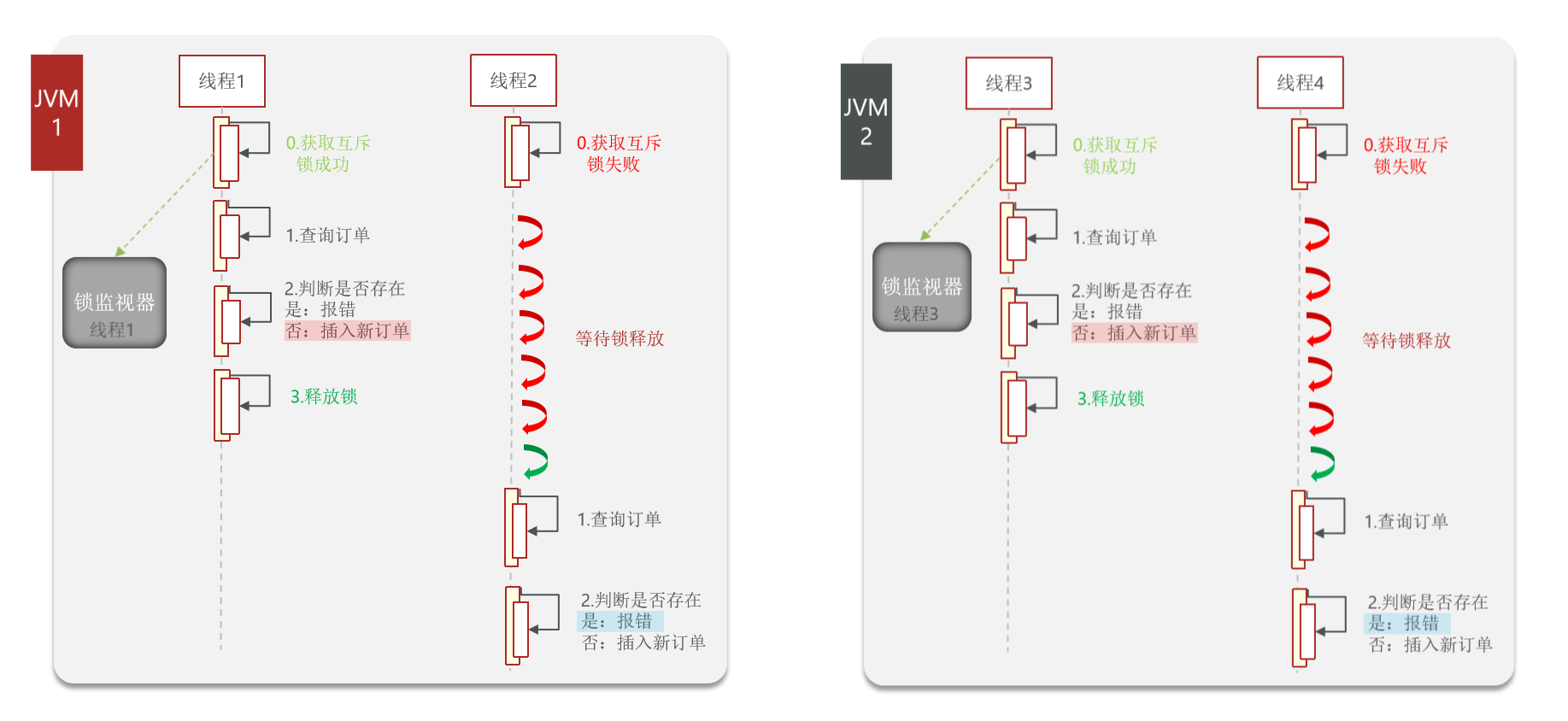
### 集群模式下的线程安全问题

#### 产生原因

一人一单问题集群模式下，synchronized锁会失效，因为synchronized只能保证在单个jvm内部的多个线程互斥！没有办法让集群下的多个jvm进程互斥！！！

在集群模式下还是并发安全问题，因为集群时有多个jvm的存在，每个jvm都有自己的锁监视器（synchronized就是利用jvm的锁监视器来控制线程的），导致同一个JVM中的每个锁有可以有一个线程获取。于是出现并行运行，那么就可能出现安全问题。

也就是说锁监视器在当前JVM的内部可以监视到线程实现互斥，但是如果有多个JVM，就会有多个锁监视器，因为集群模式先一个服务一个JVM。一个JVM内部维护一个锁监视器对象，即不同JVM有各自的锁监视器，集群下就会出现锁失效的情况。



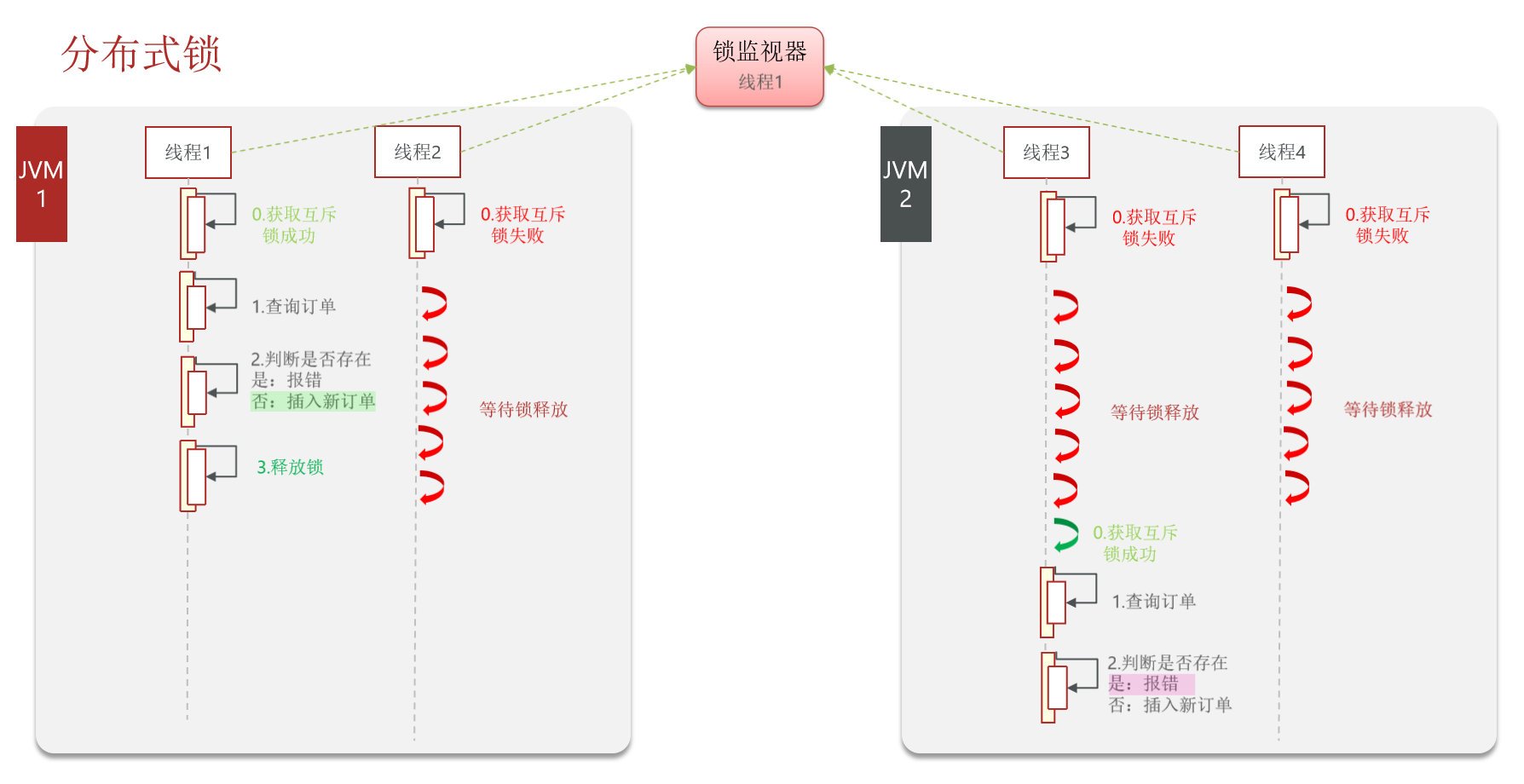
#### 解决方法：分布式锁

让多个JVM只能使用同一把锁！即需要跨JVM（跨进程）的锁！！也就是分布式锁！！！这样的锁啊可不是JDK里面有提供的，需要我们自己去实现。分布式锁就是跨JVM、跨进程的锁。

### 分布式锁

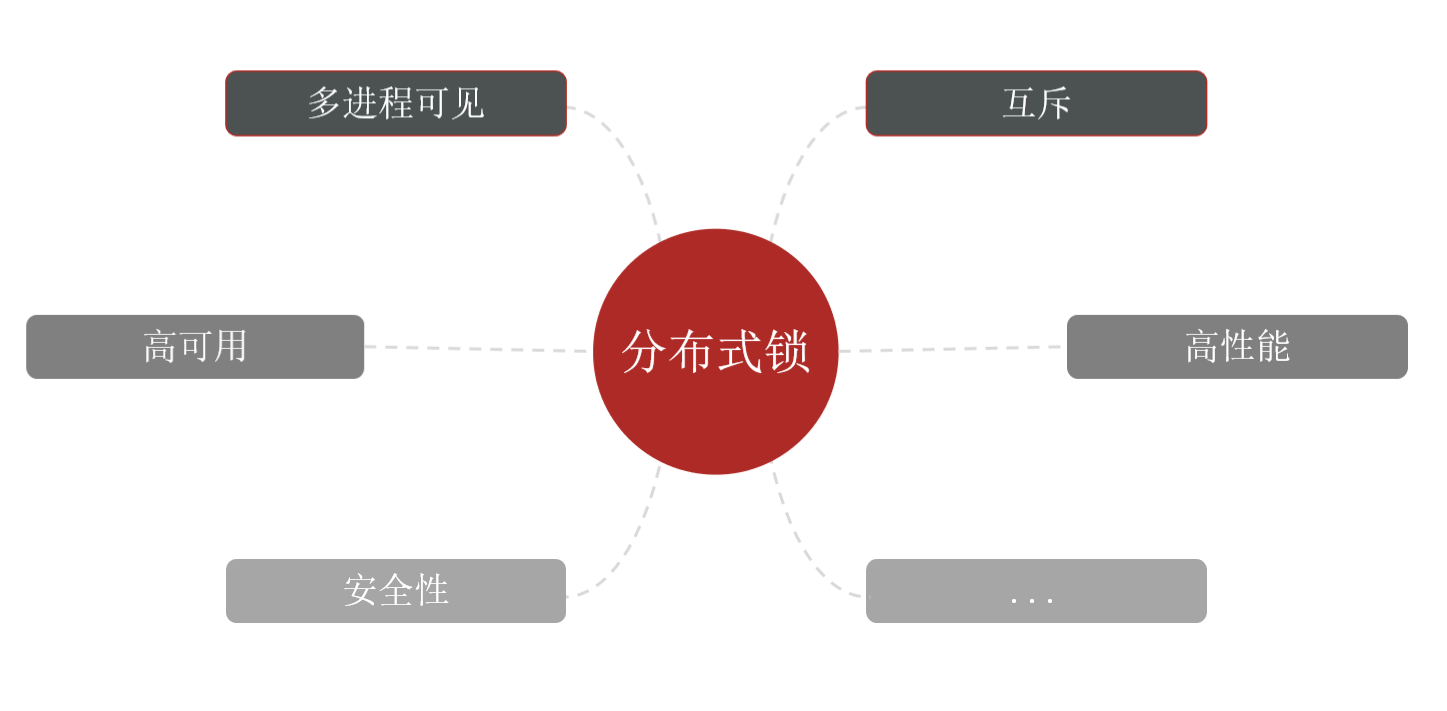
#### 什么是分布式锁

我们已经发现，在集群模式下Synchronized失效了，Synchronized只能保证单个JVM内部的多个线程之间的互斥，没有办法让集群下的多个JVM进程之间互斥！这就是集群模式线程安全问题的原因。要解决这个问题，必须使用分布式锁。

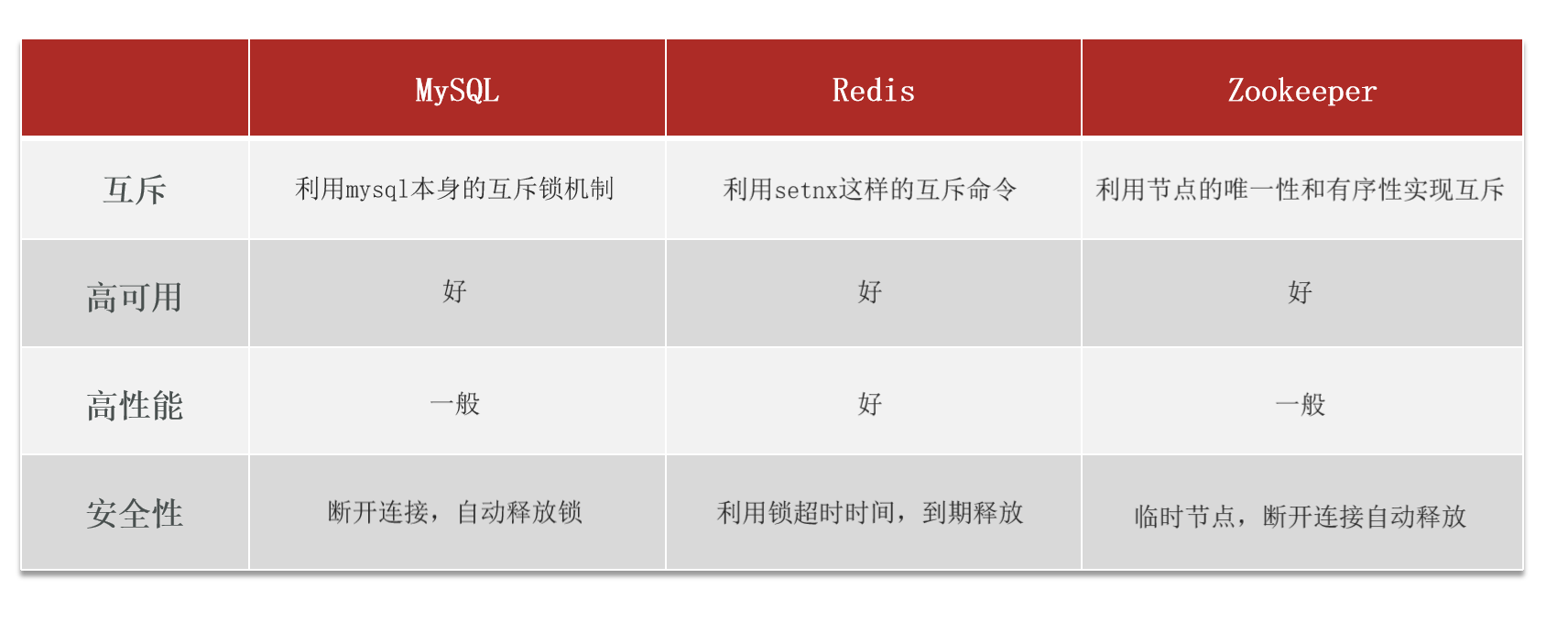


分布式锁，在分布式系统或者集群模式下多进程可见并互斥的锁，满足：多进程可见、互斥、高可用、高性能/高并发、安全。

#### 分布式锁要求：



#### 分布式锁实现方式：

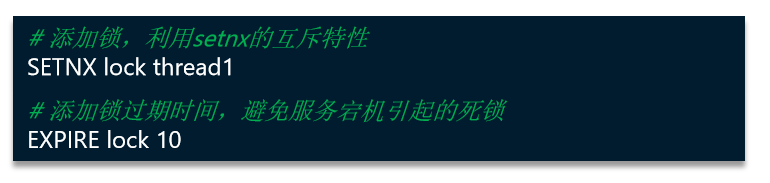


### 基于Redis实现分布式锁：

实现分布式锁时需要实现两个基本方法，就是获取锁和释放锁！

#### 1：获取锁：

互斥：确保只能有一个线程获取锁，可以**利用redis命令setnx的互斥特性；但是要对锁添加超时时间**，避免服务宕机引起的死锁，即还没来及去释放锁，web服务宕机了,这样一来释放锁的动作永远的得不到执行，也就意味着这个锁永远存在，这样其他线程也就进不来，这样整个业务就处于死锁状态。



如果获取锁分成以上两个命令是有问题的：我们加过期时间的母的就是为了避免因为服务宕机导致锁无法示范，但是存在这种情况：当执行SETNX成功了，还没来得及执行EXPIRE，此时服务宕机了，即服务宕机发生在这两个命令之间，那么锁依然不会释放！所以说必须保证SETNX和EXPIRE要么都成功、要么都失败，即必须具备原子性！于是，上面的两条命令可以写成下面的一条：



到这已经可以去获取一个分布式锁了，不过还有一个小小的问题值得我们讨论：就是在获取锁的时，结果要么成功，要么失败，成功了返回ok，失败了返回nil，但是失败了以后我们怎么做呢？在JDK里提供的锁有两种机制，一种是获取锁失败了会阻塞等待，等到有人释放锁为止，即是阻塞等待式地获取；一种是非阻塞式的获取锁，也就是我来尝试获取锁，如果我获取失败了，我会立即结束，返回一个结果，而不是说一直去尝试，一直等待。

我们实现非阻塞式的分布式锁，因为阻塞式一对CPU有一定的浪费，因为一直等待重试啊，二实现起来会相对麻烦！所以我们会用非阻塞式的方式获取分布式锁，也就是说**set key value nx ex seconds**执操作执行一次，根据返回结果，如果成功返回true，失败返回false！

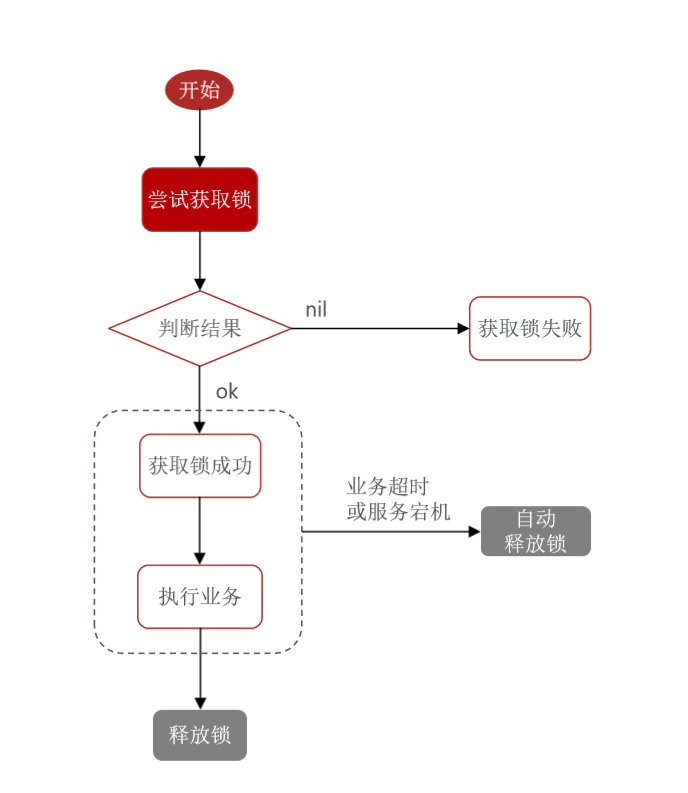
#### 2：释放锁：

手动释放；

超时释放：获取锁时添加一个超时时间；



#### 实现分布式锁流程图如下：



2 5 9 11 12 13 14