**接口幂等性**

# 什么是接口幂等性

接口幂等性就是用户对于同一操作发起一次请求或者多次请求结果都是一致的，不会因为多次点击而产生了不同的结果，eg：支付场景，由于网络原因用户多次点击了结算按钮，导致多次扣款

# 哪些情况需要防止

1. 用户多次点击按钮
2. 用户页面回退再次提交
3. 微服务互相调用，由于网络原因，导致接口调用失败而重试，eg：feign触发重试机制

## 以sql为例

有些操作时天然幂等的

Select \* from table where id = ? 无论执行多少次都不会改变状态，是天然的幂等

Update table set column = 1 where id = ? 无论执行多少次都是将column改为1，都不会改变状态，是天然的幂等

Delete from table where id = ? 多次操作，结果都一样，是天然的幂等

Insert into table(c1,c2) values(1,2) 如果主键c1为唯一，即重复上面的业务，只会插入一条主键为1的数据，具备幂等性

Update table set column = column + 1 where id = ? 每次执行结果都会发生变化，不是幂等性

Insert into table(c1,c2) values(1,2) 如果主键c1不是唯一，即重复上面的业务，会插入多条c1为1的数据，不具备幂等性

# 幂等性解决方案

## token机制(也可以是验证码)

### 概述

1. 服务端提供了发送token的接口，我们在分析业务的时候，哪些业务是存在幂等问题的，就必须在执行业务前，先去获取token(或者是可以在给前端页面返回之前生成一个token给返回到前端，前端页面在提交请求时，携带刚刚返回的token)，服务器会把token保存到redis中
2. 然后调用业务接口请求时，携带token，一般将token放在请求头
3. 服务器判断token是否存在redis中，存在表示第一次请求，然后删除token，继续执行业务
4. 如果判断redis中不存在此token，就表示是重复操作，不会继续执行业务逻辑，这样就保证了业务代码不会被重复执行

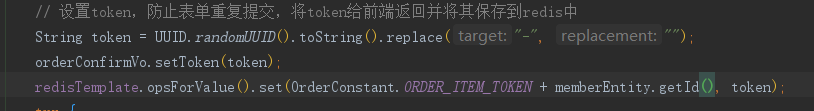
### Token机制的注意事项

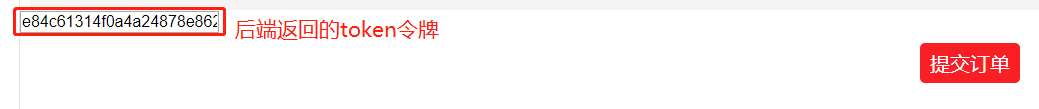
1. 删除token的时机
2. 先删除token可能导致，业务执行失败，但是token已经被删除，重试执行的时候携带之前的token ，由于防重设计，导致业务还是不能执行
3. 后删除token可能导致，业务处理成功，但是服务闪断，出现超时，没有删除token，或者是还没有删除token(因为token如果存在redis中的话，需要进行网络交互)，第二次请求进来，判断token还在，重复执行了业务代码
4. 最好是先删除token，如果业务调用失败，就重新获取token再次请求
5. Token获取、比较、删除必须是原子性
6. 如果在高并发的场景下，token获取、比较、删除这三步操作不是原子性的话，可能导致两个请求同时进来同时获取到token，判断都成功而重复执行了业务逻辑
7. 可以使用redis的lua脚本实现

### 项目实战

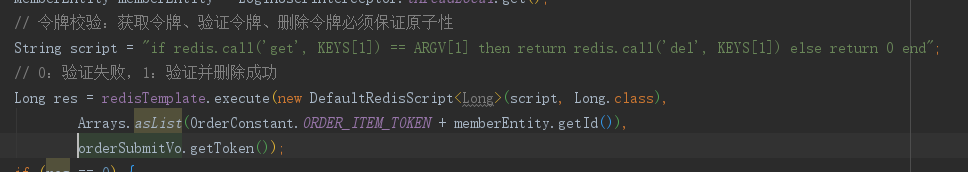
在电商系统中的提交订单功能

1. 在返回提交订单(订单确认页)的时候携带token

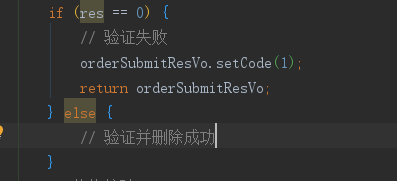




1. 在点击提交订单按钮时，将token带上与redis进行比较



1. 如果比较成功，说明是第一次提交，则删除redis里面的令牌



## 锁机制

### 数据库悲观锁

Select \* from table where id = ? for update

for update在查询的时候会锁定这条数据，其它操作不能操作这条数据。悲观锁使用时一般伴随着事务一起使用，数据锁定时间可能会很长，需要根据实际情况使用，另外要注意的是，id字段一定是主键或者唯一索引，不然可能造成锁表的结果，处理起来会非常麻烦

### 数据库乐观锁

这种方法适合在更新的业务场景中，主要用于读多写少的问题

Update table set column = column + 1,version = version + 1 where id = ? and version = ?

根据version版本，也就是在操作库存前先获取当前商品的版本号version，然后操作的时候带上此version版本号。eg：我们第一次操作库存时，version为1，调用库存服务时version变成了2，但返回给订单服务出现了问题，订单服务又一次发起库存服务，但version传的还是1，再次执行上面的sql语句时就不会执行，因为此时数据库版本号已经变成2，where条件不成立，这样就保证不管调用几次，只会真正的执行一次

### 业务层分布式锁

多个服务器可能在同一时间处理同一条数据，eg：多台机器定时任务都拿到了相同的数据进行处理，可以加分布式锁来锁定此数据，处理完成之后释放锁，获取到锁之后必须先判断此数据是否被处理

## 其它一些方法

1. 数据库唯一约束
2. Redis set 防重
3. 防重表
4. 全局请求唯一id