前任开发在代码里下毒，支付下单居然没加幂等

原创 程序员老猫 程序员老猫 2024年01月16日 08:30 上海

> 分享是最有效的学习方式。

故事

又是一个风和日丽美好的一天，小猫戴着耳机，安逸地听着音乐，撸着代码，这种没有会议的日子真的是巴适得板。

不料祸从天降，组长火急火燎地跑过来找到了小猫。“快排查一下，目前有A公司用户反馈积分被多扣了”。

小猫回忆了一下“不对啊，这接口我也没动过啊，前几天对外平台的老六直接找我要个支付接口，我就给他了的，以前的代码，我都没有动过的.....”。

于是小猫一边疑惑一边翻看着以前的代码，越看脸色越差.....

小猫做的是一个标准的积分兑换商城，以前和客户合作的时候，客户直接用的是小猫单位自己定制的h5页面。这次合作了一家公司有点特殊，由于公司想要定制化自己个性化的H5，加上本身A公司自己有开发能力，所以经过讨论就以接口的方式直接将相关接口给出去，A客户H5开发完成之后自己来对接。

慢慢地，原因也水落石出，之前好好的业务一直没有问题是因为商城的本身H5页面做了防重复提交，由于量小，并且一般对接方式用的都是纯H5，所以都没有什么问题，然后这次是直接将接口给出去了，完了接口居然没有加幂等.....

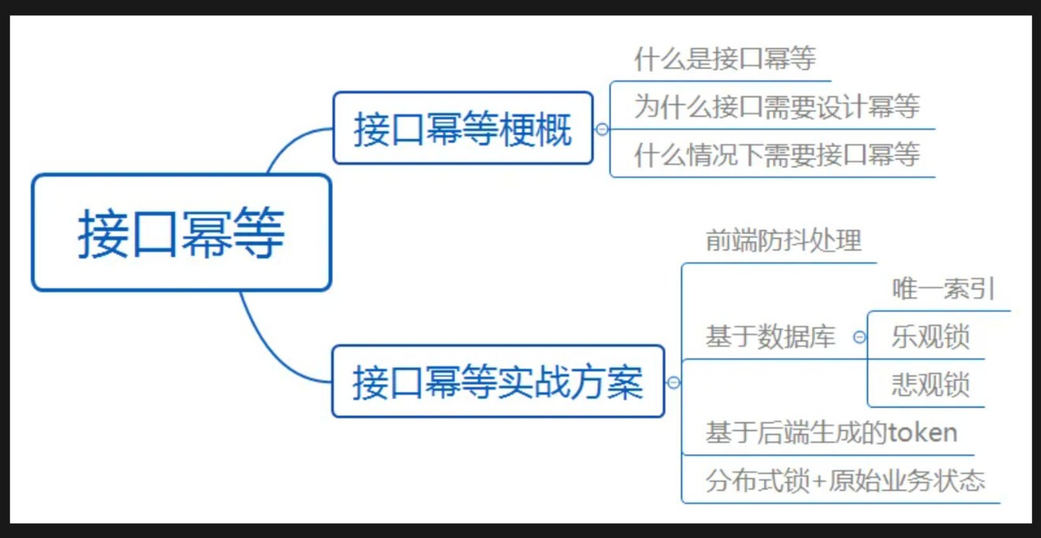
小猫躺枪，数据订正当然是少不了了，事故报告当然也少不了了。

正所谓前人挖坑，后人遭殃，前人锅后人背。

聊聊幂等

接口幂等梗概

这个案例其实就是一个典型的接口幂等案例。那么老猫就和大家从以下几个方面好好剖析一下接口幂等吧。



什么是接口幂等

比较专业的术语：其任意多次执行所产生的影响均与第一次执行的影响相同。大白话：多次调用的情况下，接口最终得到的结果是一致的。

那么为什么需要幂等呢？

1. 用户进行提交动作的时候，由于网络波动等原因导致后端同步响应不及时，这样用户就会一直点点点，这样机会发生重复提交的情况。

2. 分布式系统之间调用的情况下，例如RPC调用，为了防止网络波动超时等造成的请求失败，都会添加重试机制，导致一个请求提交多次。

3. 分布式系统经常会用到消息中间件，当由于网络原因，mq没有收到ack的情况下，就会导致消息的重复投递，从而就会导致重复提交行为。

4. 还有就是恶意攻击了，有些业务接口做的比较粗糙，黑客找到漏洞之后会发起重复提交，这样就会导致业务出现问题。打个比方，老猫曾经干过，邻居小孩报名了一个画画比赛，估计是机构培训发起的，功能做的也差，需要靠投票赢得某些礼品，然后老猫抓到接口信

息之后就模拟投票进行重复刷了投票。

那么哪些接口需要做幂等呢？

首先我们说是不是所有的接口都需要幂等？是不是加了幂等就好呢？显然不是。因为接口幂等的实现某种意义上是要消耗系统性能的，我们没有必要针对所有业务接口都加上幂等。

这个其实并不能做一个完全的定义说哪个就不用幂等，因为很多时候其实还是得结合业务逻辑一起看。但是其中也是有规律可循的。

既然我们说幂等就是多次调用，接口最终得到结果一致，那么很显然，查询接口肯定是不要加幂等的，另外一些简单删除数据的接口，无论是逻辑删除还是物理删除，看场景的情况下其实也不用加幂等。

但是大部分涉及到多表更新行为的接口，咱们最好还是得加上幂等。

接口幂等实战方案

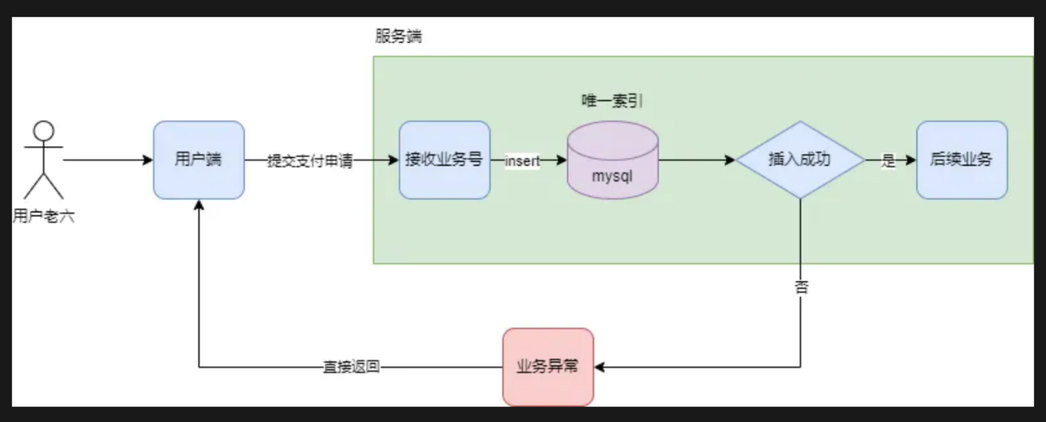
前端防抖处理

前端防抖主要可以有两种方案，一种是技术层面的，一种是产品层面的：

1. 技术层面：例如提交控制在100ms内，同一个用户最多只能做一次订单提交的操作。
2. 产品层面：当然用户点击提交之后，按钮直接置灰。

基于数据库唯一索引

1. 利用数据库唯一索引。我们具体来看一下流程，咱们就用小猫遇到的例子。如下：



过程描述：

- 建立一张去重表，其中某个字段需要建立唯一索引，例如小猫这个场景中，咱们就可以将订单提交流水单号作为唯一索引存储到我们的数据库中，就模型上而言，可以将其定义为支付请求流水表。
- 客户端携带相关流水信息到后端，如果发现编号重复，那么此时就会插入失败，报主键冲突的错误，此时我们针对该错误做一下业务报错的二次封装给到客户另一个友好的提示即可。

数据库乐观锁实现

什么是乐观锁，它假设多用户并发的事务在处理时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的那部分数据。说得直白一点乐观锁就是一个马大哈。总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，只在更新的时候会判断一下在此期间别人有没有去更新这个数据。

例如提交订单的进行支付扣款的时候，本来可能更新账户金额扣款的动作是这样的：

```
update Account set balance = balance-#{payAmount} where accountCode = #{accountCode}
```

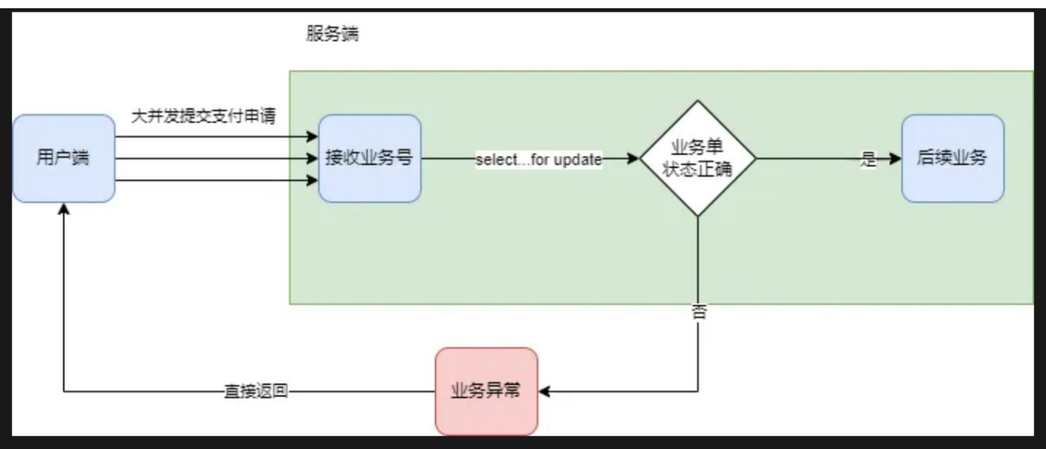
加上版本号之后，咱们的代码就是这样的。

```
update Account set balance = balance-#{payAmount},version=version +1 where accountCode =
```

这种情况下其实就要求客户端每次在请求支付下单的时候都需要上层客户端指定好当前的版本信息。不过这种幂等的处理方式，老猫用的比较少。

数据库悲观锁实现

悲观锁的话具有强烈的独占和排他特性。大白话谁都不信的主。所以我们就用select ... for update这样的语法进行行锁，当然老猫觉得单纯的select ... for update只能解决同一时刻大并发的幂等，所以要保证单号重试这样非并发的幂等请求还是得去校验当前数据的状态才行。就拿当前的小猫遇到的场景来说，流程如下：



悲观锁

```
begin;  # 1.开始事务
select * from order where order_code='666' for update # 查询订单，判断状态,锁住这条记录
if (status !=处理中) {
   //非处理中状态，直接返回；
   return ;
}
## 处理业务逻辑
update order set status='完成' where order_code='666' # 更新完成
update stock set num = num - 1 where spu='xxx' # 库存更新
commit; # 5.提交事务
```

这里老猫一再想要强调的是在校验的时候还是得带上本身的业务状态去做校验，select ... for update并非万能幂等。

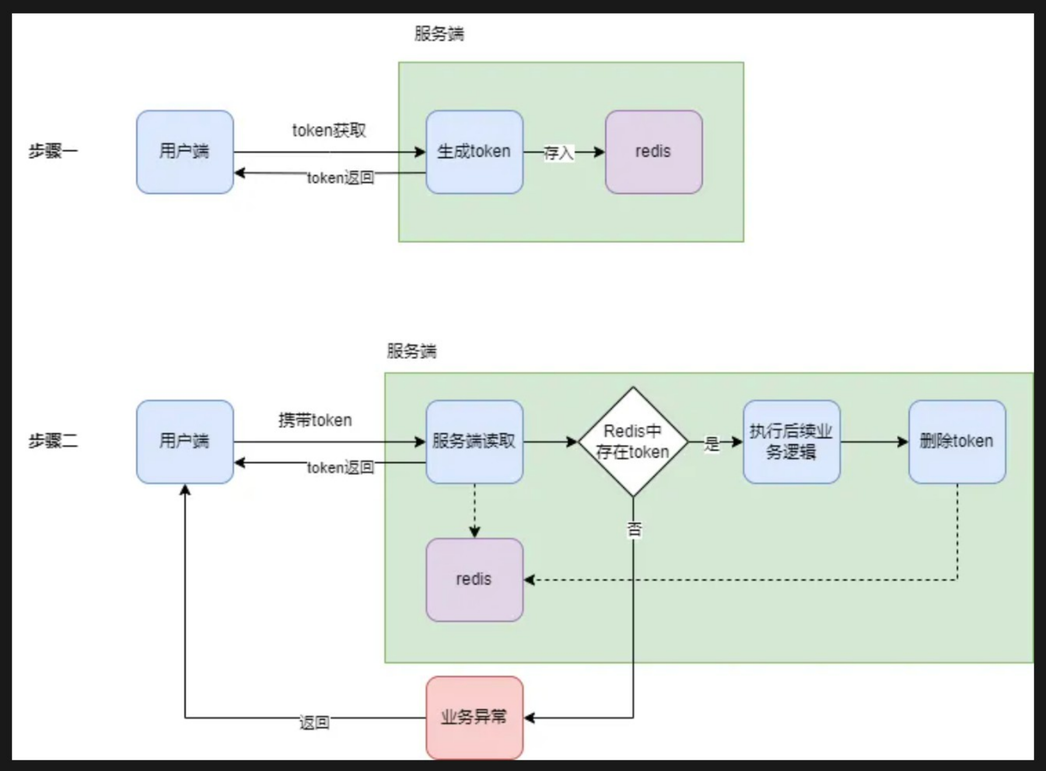
后端生成token

这个方案的本质其实是引入了令牌桶的机制，当提交订单的时候，前端优先会调用后端接口获取一个token，token是由后端发放的。当然token的生成方式有很多种，例如定时刷新令牌桶，或者定时生成令牌并放到令牌池中，当然目的只有一个就是保住token的唯一性即可。

生成token之后将token放到redis中，当然需要给token设置一个失效时间，超时的token也会被删除。

当后端接收到订单提交的请求的时候，会先判断token在缓存中是否存在，第一次请求的时候，token一定存在，也会正常返回结果，但是第二次携带同一个token的时候被拒绝了。

流程如下：



token机制

有个注意点大家可以思考一下：如果用户用程序恶意刷单，同一个token发起了多次请求怎么办？想要实现这个功能，就需要借助分布式锁以及Lua脚本了，分布式锁可以保证同一个token不能有多个请求同时过来访问，lua脚本保证从redis中获取令牌->比对令牌->生成单号->删除令牌这一系列行为的原子性。

分布式锁+状态机（订单状态）

现在很多的业务服务都是分布式系统，所以就拿分布式锁来说，关于分布式锁，老猫在此不做赘述，之前老猫写过redis的分布式锁和实现，还有zk锁和实现，具体可见链接：

1. **锁的演化**
2. **手撕redis分布式锁**
3. **手撸ZK锁**

当然和上述的数据库悲观锁类似，咱们的分布式锁也只能保证同一个订单在同一时间的处理。其次也是要去校订单的状态，防止其重复支付的，也就是说，只要支付的订单进入后端，都要将原先的订单修改为支付中，防止后续支付中断之后的重复支付。

在上述小猫的流程中还没有涉及到现金补充，如果涉及到现金补充的话，例如对接了微信或者支付宝的情况，还需要根据最终的支付回调结果来最终将订单状态进行流转成支付完成或者是支付失败。

总结

在我们日常的开发中，一些重要的接口还是需要大家谨慎对待，即使是前任开发留下的接口，没有任何改动，当有人咨询的时候，其实就要好好去了解一下里面的实现，看看方案有没有问题，看看技术实现有没有问题，这应该也是每一个程序员的基本素养。

另外的，在一些重要的接口上，尤其是资金相关的接口上，幂等真的是相当的重要。小伙伴们，你们觉得呢？如果大家还有好的解决方案，或者有其他思考或者意见也欢迎大家的留言。

> 我是老猫，资深研发老鸟，让我们一起聊聊技术，聊聊人生。
>
> 都看到这了，求个点赞、关注、在看三连呗，感谢支持。

小猫 17　# 幂等 1

小猫 · 目录 ≡

下一篇 · 拓展了个新业务枚举类型，资损了 >

修改于2024年01月16日