

谷粒商城

接口幂等性



一、什么是幂等性

接口幂等性就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用；比如说支付场景，用户购买了商品支付扣款成功，但是返回结果的时候网络异常，此时钱已经扣了，用户再次点击按钮，此时会进行第二次扣款，返回结果成功，用户查询余额发现多扣钱了，流水记录也变成了两条... ,这就没有保证接口的幂等性。

二、哪些情况需要防止

用户多次点击按钮

用户页面回退再次提交

微服务互相调用，由于网络问题，导致请求失败。feign 触发重试机制

其他业务情况

三、什么情况下需要幂等

以 SQL 为例，有些操作是天然幂等的。

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 tab1 SET col1=1 WHERE col2=2, 无论执行成功多少次状态都是一致的，也是幂等操作。

delete from user where userid=1, 多次操作，结果一样，具备幂等性

insert into user(userid,name) values(1,'a') 如 userid 为唯一主键，即重复操作上面的业务，只会插入一条用户数据，具备幂等性。

UPDATE tab1 SET col1=col1+1 WHERE col2=2, 每次执行的结果都会发生变化，不是幂等的。

insert into user(userid,name) values(1,'a') 如 userid 不是主键，可以重复，那上面业务多次操作，数据都会新增多条，不具备幂等性。

四、幂等解决方案

1、token 机制

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，服务器会把 token 保存到 redis 中。

- 2、然后调用业务接口请求时，把 token 携带过去，一般放在请求头部。
- 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 token 不存在 redis 中，就表示是重复操作，直接返回重复标记给 client，这样就保证了业务代码，不被重复执行。

危险性:

1、先删除 token 还是后删除 token:

- (1) 先删除可能导致，业务确实没有执行，重试还带上之前 token，由于防重设计导致，请求还是不能执行。
- (2) 后删除可能导致，业务处理成功，但是服务闪断，出现超时，没有删除 token，别人继续重试，导致业务被执行两边
- (3) 我们最好设计为先删除 token，如果业务调用失败，就重新获取 token 再次请求。

2、Token 获取、比较和删除必须是原子性

- (1) redis.get(token) 、token.equals、redis.del(token)如果这两个操作不是原子，可能导致，高并发下，都 get 到同样的数据，判断都成功，继续业务并发执行
- (2) 可以在 redis 使用 lua 脚本完成这个操作

```
if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end
```

2、各种锁机制

1、数据库悲观锁

```
select * from xxxx where id = 1 for update;
```

悲观锁使用时一般伴随事务一起使用，数据锁定时间可能会很长，需要根据实际情况选用。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。

2、数据库乐观锁

这种方法适合在更新的场景中，

```
update t_goods set count = count - 1, version = version + 1 where good_id=2 and version = 1
```

根据 version 版本，也就是在操作库存前先获取当前商品的 version 版本号，然后操作的时候带上此 version 号。我们梳理下，我们第一次操作库存时，得到 version 为 1，调用库存服务 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调用库存服务，当订单服务传如的 version 还是 1，再执行上面的 sql 语句时，就不会执行；因为 version 已经变为 2 了，where 条件就不成立。这样就保证了不管调用几次，只会真正的处理一次。

乐观锁主要使用于处理读多写少的问题

3、业务层分布式锁

如果多个机器可能在同一时间同时处理相同的数据，比如多台机器定时任务都拿到了相同数据，我们就可以加分布式锁，锁定此数据，处理完成后释放锁。获取到锁的必须先判断这个数据是否被处理过。

3、各种唯一约束

1、数据库唯一约束

插入数据，应该按照唯一索引进行插入，比如订单号，相同的订单就不可能有两条记录插入。我们在数据库层面防止重复。

这个机制是利用了数据库的主键唯一约束的特性，解决了在 insert 场景时幂等问题。但主键的要求不是自增的主键，这样就需要业务生成全局唯一的主键。

如果是分库分表场景下，路由规则要保证相同请求下，落地在同一个数据库和同一表中，要不然数据库主键约束就不起作用了，因为是不同的数据库和表主键不相关。

2、redis set 防重

很多数据需要处理，只能被处理一次，比如我们可以计算数据的 MD5 将其放入 redis 的 set，每次处理数据，先看这个 MD5 是否已经存在，存在就不处理。

4、防重表

使用订单号 orderNo 做为去重表的唯一索引，把唯一索引插入去重表，再进行业务操作，且他们在同一个事务中。这个保证了重复请求时，因为去重表有唯一约束，导致请求失败，避免了幂等问题。这里要注意的是，去重表和业务表应该在同一库中，这样就保证了在同一个事务，即使业务操作失败了，也会把去重表的数据回滚。这个很好的保证了数据一致性。

之前说的 redis 防重也算

5、全局请求唯一 id

调用接口时，生成一个唯一 id，redis 将数据保存到集合中（去重），存在即处理过。

可以使用 nginx 设置每一个请求的唯一 id；

```
proxy_set_header X-Request-Id $request_id;
```

