https://mp.weixin.qq.com/s/dHV1dPrrs59dJBgznLZr2Q

**实战，实现幂等的8种方案！**

## 前言

大家好，我是程序员田螺。今天我们一起来聊聊幂等设计。

1. 什么是幂等
2. 为什么需要幂等
3. 接口超时，如何处理呢？
4. 如何设计幂等？
5. 实现幂等的8种方案
6. HTTP的幂等

## 1. 什么是幂等?

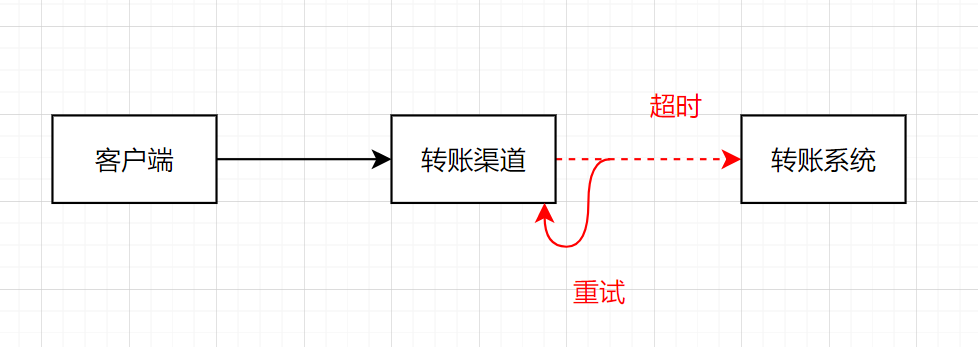
幂等是一个数学与计算机科学概念。

* 在数学中，幂等用函数表达式就是：f(x) = f(f(x))。比如求绝对值的函数，就是幂等的，abs(x) = abs(abs(x))。
* 计算机科学中，幂等表示一次和多次请求某一个资源应该具有同样的副作用，或者说，多次请求所产生的影响与一次请求执行的影响效果相同。

## 2. 为什么需要幂等

举个例子：

我们开发一个转账功能，假设我们调用下游接口**超时**了。一般情况下，**超时**可能是**网络传输丢包**的问题，也可能是请求时没送到，还有可能是请求到了，**返回结果却丢**了。这时候我们是否可以重试呢？如果**重试**的话，是否会多转了一笔钱呢？



当前互联网的系统几乎都是解耦隔离后，会存在各个不同系统的相互远程调用。调用远程服务会有三个状态：成功，失败，或者超时。前两者都是明确的状态，而超时则是**未知状态**。我们转账**超时**的时候，如果下游转账系统做好**幂等**控制，我们发起**重试**，那即可以**保证转账正常进行，又可以保证不会多转一笔**。

其实除了转账这个例子，日常开发中，还有**很多很多例子需要考虑幂等**。比如：

* MQ（消息中间件）消费者读取消息时，有可能会读取到重复消息。（**重复消费**）
* 比如提交form表单时，如果快速点击提交按钮，可能产生了两条一样的数据（**前端重复提交**）

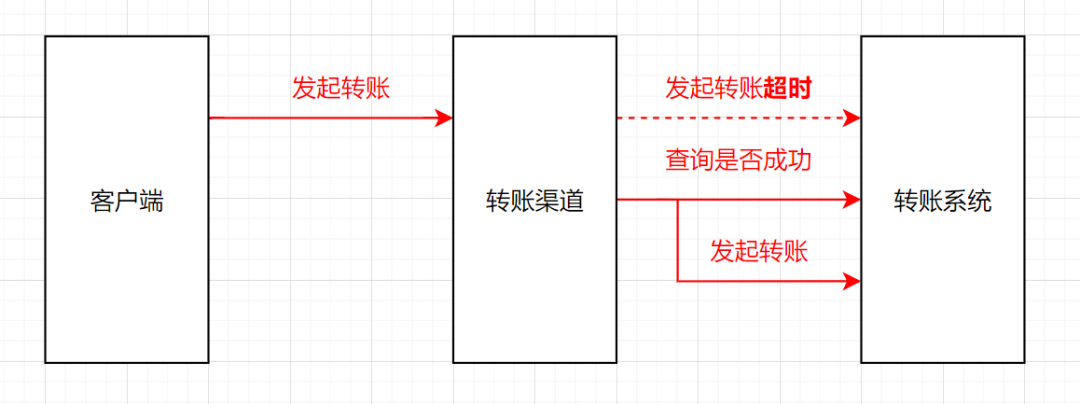
## 3. 接口超时了，到底如何处理？

如果我们调用下游接口超时了，我们应该怎么处理呢？

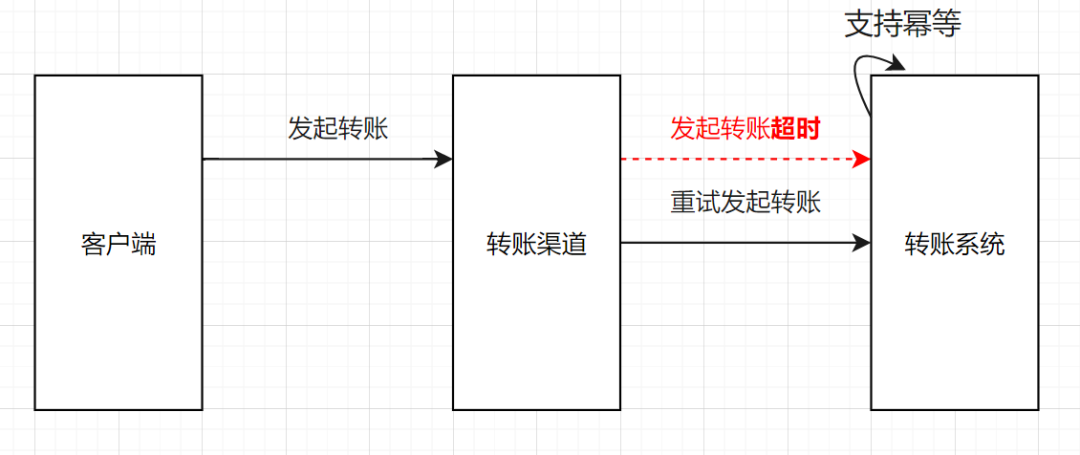
有**两种方案**处理：

* 方案一：就是下游系统提供一个对应的查询接口。如果接口超时了，先查下对应的记录，如果查到是成功，就走成功流程，如果是失败，就按失败处理。

拿我们的转账例子来说，转账系统提供一个查询**转账记录**的接口，如果**渠道系统**调用**转账系统**超时时，**渠道系统**先去查询一下这笔记录，看下这笔转账记录成功还是失败，如果成功就走成功流程，失败再重试发起转账。



方案二：下游接口**支持幂等**，上游系统如果**调用超时**，发起重试即可。



两种方案都是挺不错的，但是如果是**MQ重复消费的场景**，方案一处理并不是很妥，所以，我们还是要求下游系统**对外接口支持幂等**。

## 4. 如何设计幂等

既然这么多场景需要考虑幂等，那我们如何设计幂等呢？

幂等意味着一条请求的唯一性。不管是你哪个方案去设计幂等，都需要一个**全局唯一的ID**，去标记这个请求是独一无二的。

* 如果你是利用唯一索引控制幂等，那唯一索引是唯一的
* 如果你是利用数据库主键控制幂等，那主键是唯一的
* 如果你是悲观锁的方式，底层标记还是**全局唯一的ID**

### 4.1 全局的唯一性ID

全局唯一性ID，我们怎么去生成呢？你可以回想下，数据库主键Id怎么生成的呢？

是的，我们可以使用UUID，但是UUID的缺点比较明显，它字符串占用的空间比较大，生成的ID过于随机，可读性差，而且没有递增。

我们还可以使用雪花算法（Snowflake） 生成唯一性ID。

雪花算法是一种生成分布式全局唯一ID的算法，生成的ID称为Snowflake IDs。这种算法由Twitter创建，并用于推文的ID。

一个Snowflake ID有64位。

* 第1位：Java中long的最高位是符号位代表正负，正数是0，负数是1，一般生成ID都为正数，所以默认为0。
* 接下来前41位是时间戳，表示了自选定的时期以来的毫秒数。
* 接下来的10位代表计算机ID，防止冲突。
* 其余12位代表每台机器上生成ID的序列号，这允许在同一毫秒内创建多个Snowflake ID。

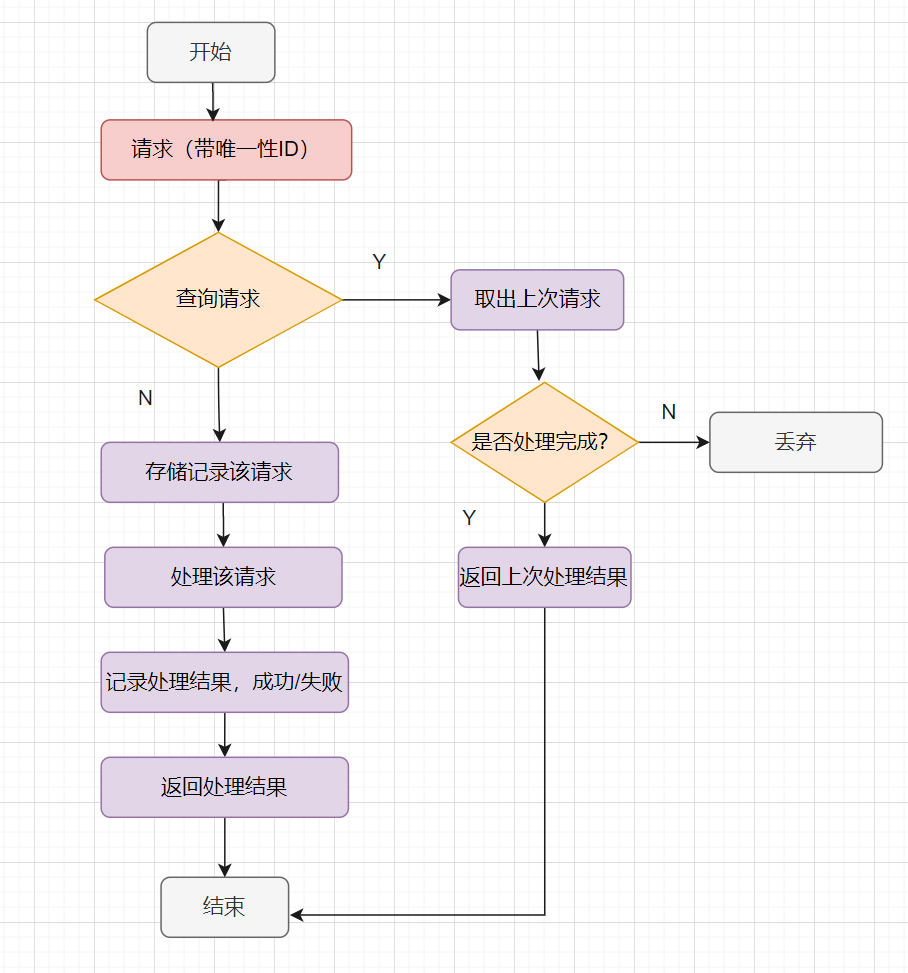
雪花算法

当然，全局唯一性的ID，还可以使用百度的Uidgenerator，或者美团的Leaf。

### 4.2 幂等设计的基本流程

幂等处理的过程，说到底其实就是过滤一下已经收到的请求，当然，请求一定要有一个全局唯一的ID标记哈。然后，怎么判断请求是否之前收到过呢？把请求储存起来，收到请求时，先查下存储记录，记录存在就返回上次的结果，不存在就处理请求。

一般的幂等处理就是这样啦，如下：



## 5. 实现幂等的8种方案

幂等设计的基本流程都是类似的，我们简简单单来过一下幂等实现的8中方案哈

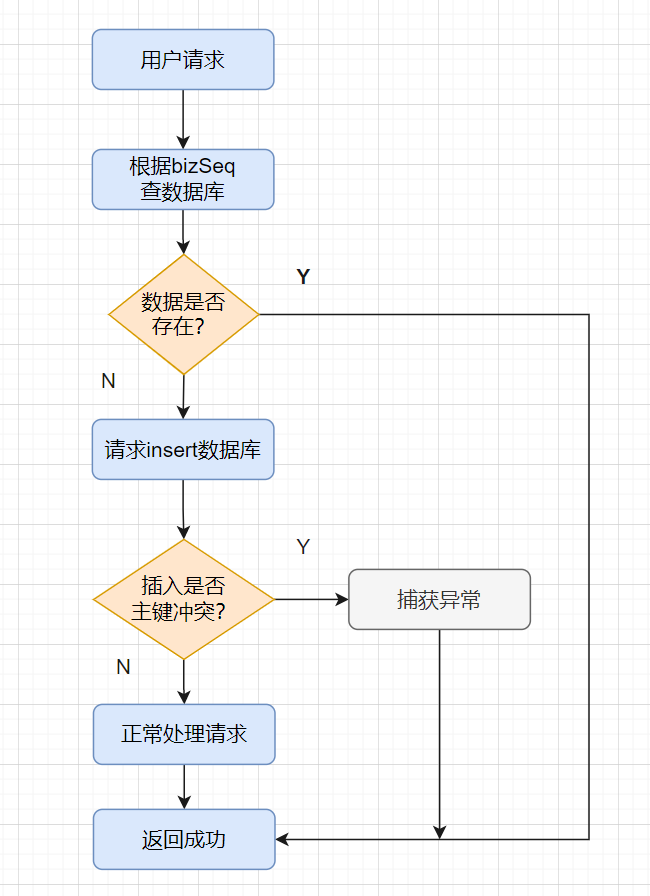
### 5.1 select+insert+主键/唯一索引冲突

日常开发中，为了实现交易接口幂等，我是这样实现的：

交易请求过来，我会先根据请求的**唯一流水号** bizSeq字段，先select一下**数据库的流水表**

* 如果数据已经存在，就拦截是重复请求，直接返回成功；
* 如果数据不存在，就执行insert插入，如果insert成功，则直接返回成功，如果insert产生主键冲突异常，则捕获异常，接着直接返回成功。

流程图如下



为什么前面已经select查询了，还需要try...catch...捕获重复异常呢？

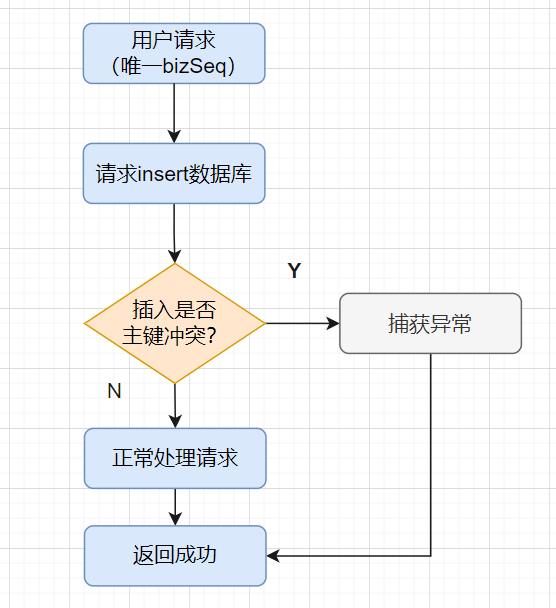
是因为高并发场景下，两个请求去select的时候，可能都没查到，然后都走到insert的地方啦。

当然，用**唯一索引**代替**数据库主键**也是可以的哈，都是**全局唯一的ID**即可。

### 5.2. 直接insert + 主键/唯一索引冲突

在5.1方案中，都会先查一下**流水表**的交易请求，判断是否存在，然后不存在再插入请求记录。如果**重复请求的概率比较低**的话，我们可以直接插入请求，利用**主键/唯一索引冲突**，去判断是**重复请求**。

流程图如下：



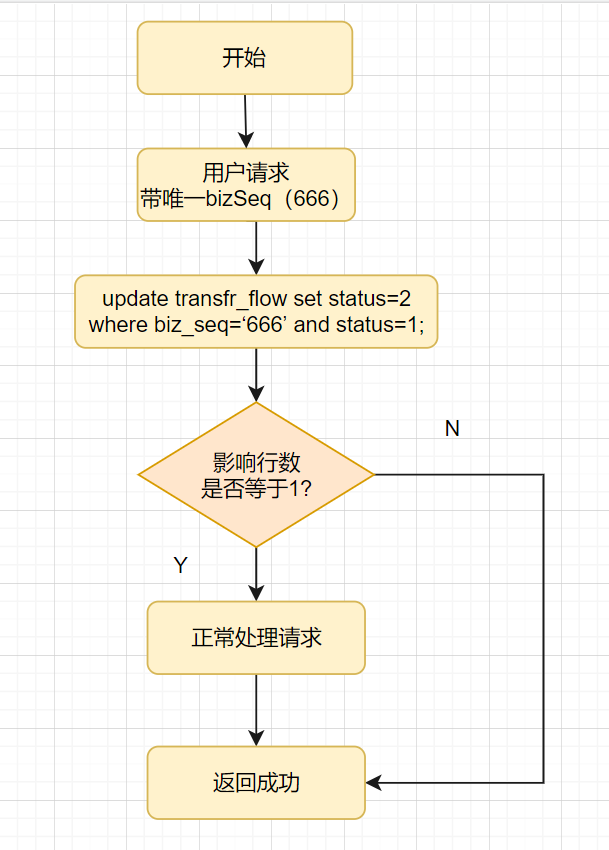
### 5.3 状态机幂等

很多业务表，都是有状态的，比如转账流水表，就会有0-待处理，1-处理中、2-成功、3-失败状态。转账流水更新的时候，都会涉及流水状态更新，即涉及**状态机 (即状态变更图)**。我们可以利用状态机实现幂等，一起来看下它是怎么实现的。

比如转账成功后，把**处理中**的转账流水更新为**成功**状态，SQL这么写：

update transfr\_flow set status=2 where biz\_seq=‘666’ and status=1;

**简要流程图**如下：



状态机是怎么**实现幂等**的呢？

* 第1次请求来时，bizSeq流水号是 666，该流水的状态是处理中，值是 1，要更新为2-成功的状态，所以该update语句可以正常更新数据，sql执行结果的影响行数是1，流水状态最后变成了2。
* 第2请求也过来了，如果它的流水号还是 666，因为该流水状态已经2-成功的状态了，所以更新结果是0，不会再处理业务逻辑，接口直接返回。

### 5.4 抽取防重表

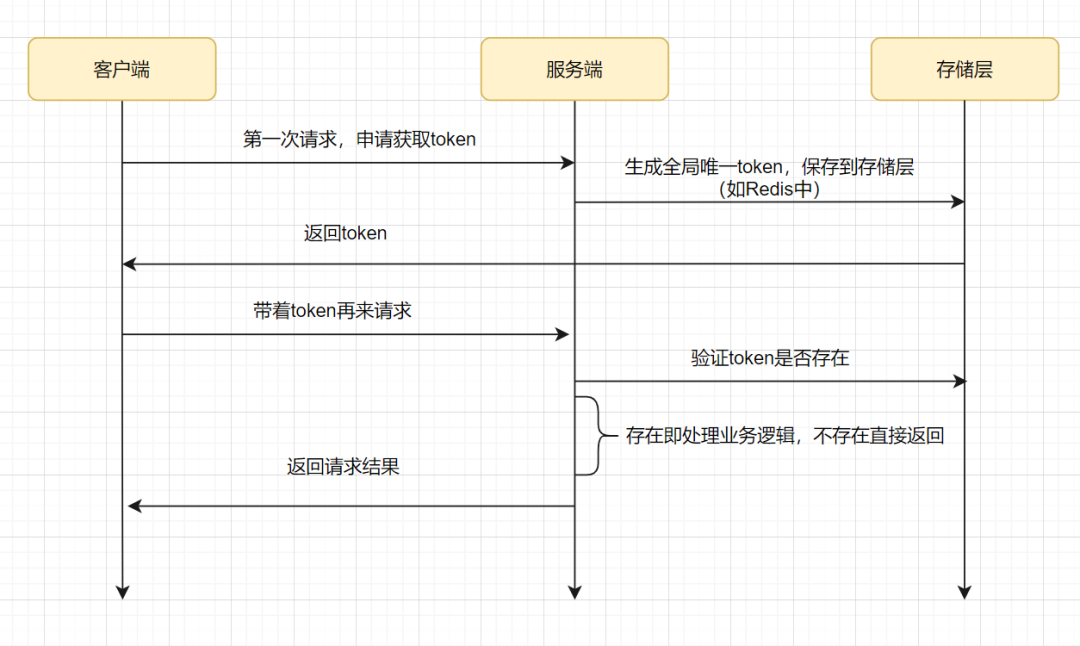
**5.1和5.2的方案**，都是建立在业务流水表上bizSeq的唯一性上。很多时候，我们**业务表唯一流水号**希望后端系统生成，又或者我们希望**防重功能与业务表分隔开**来，这时候我们可以单独搞个**防重表**。当然防重表也是利用主键/索引的唯一性，如果插入防重表冲突即直接返回成功，如果插入成功，即去处理请求。

### 5.5 token令牌

token 令牌方案一般包括两个请求阶段：

1. 客户端请求申请获取token，服务端生成token返回
2. 客户端带着token请求，服务端校验token

流程图如下：



1. 客户端发起请求，申请获取token。
2. 服务端生成全局唯一的token，保存到redis中（一般会设置一个过期时间），然后返回给客户端。
3. 客户端带着token，发起请求。
4. 服务端去redis确认token是否存在，一般用 redis.del(token)的方式，如果存在会删除成功，即处理业务逻辑，如果删除失败不处理业务逻辑，直接返回结果。

### 5.6 悲观锁(如select for update)

什么是**悲观锁**？

通俗点讲就是**很悲观**，每次去操作数据时，都觉得别人中途会修改，所以每次在拿数据的时候都会上锁。官方点讲就是，共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程。

悲观锁如何控制幂等的呢？就是**加锁**呀，一般配合事务来实现。

举个更新订单的业务场景：

假设先查出订单，如果查到的是处理中状态，就处理完业务，再然后更新订单状态为完成。如果查到订单，并且是不是处理中的状态，则直接返回

* 这里面order\_id需要是**索引**或**主键**哈，要锁住这条记录就好，如果不是**索引或者主键**，会**锁表**的！
* 悲观锁在同一事务操作过程中，锁住了一行数据。别的请求过来只能**等待**，如果当前事务耗时比较长，就很影响接口性能。所以一般不建议用悲观锁做这个事情。

### 5.7 乐观锁

悲观锁有性能问题，可以试下**乐观锁**。

什么是**乐观锁**？

乐观锁在操作数据时,则非常乐观，认为别人不会同时在修改数据，因此乐观锁不会上锁。只是在执行更新的时候判断一下，在此期间别人是否修改了数据。

**怎样实现乐观锁呢？**

就是给表的加多一列version版本号，每次更新记录version都升级一下（version=version+1）。具体流程就是先查出当前的版本号version，然后去更新修改数据时，确认下是不是刚刚查出的版本号，如果是才执行更新

比如，我们更新前，先查下数据，查出的版本号是version =1

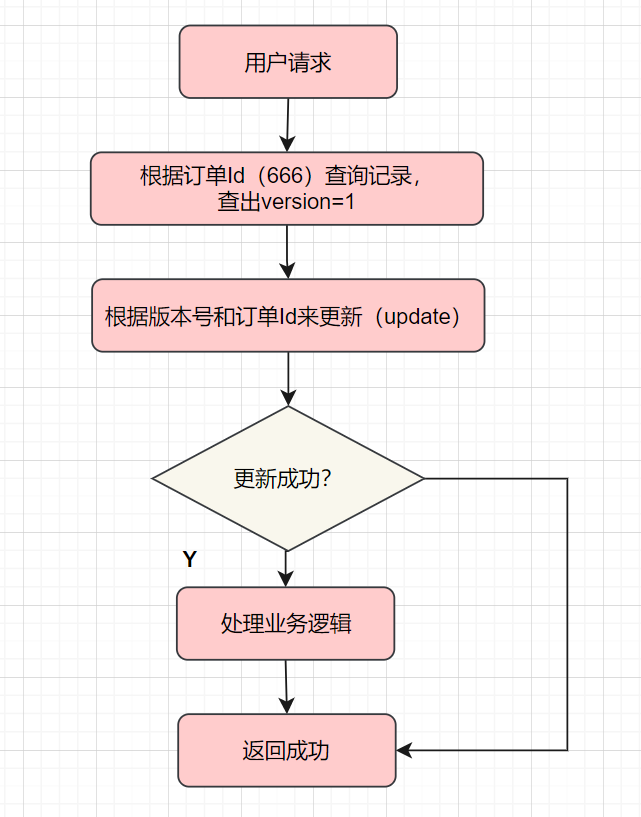
select order\_id，version from order where order\_id='666'；

然后使用version =1和订单Id一起作为条件，再去更新

update order set version = version +1，status='P' where  order\_id='666' and version =1

最后更新成功，才可以处理业务逻辑，如果更新失败，默认为重复请求，直接返回。

**流程图如下：**

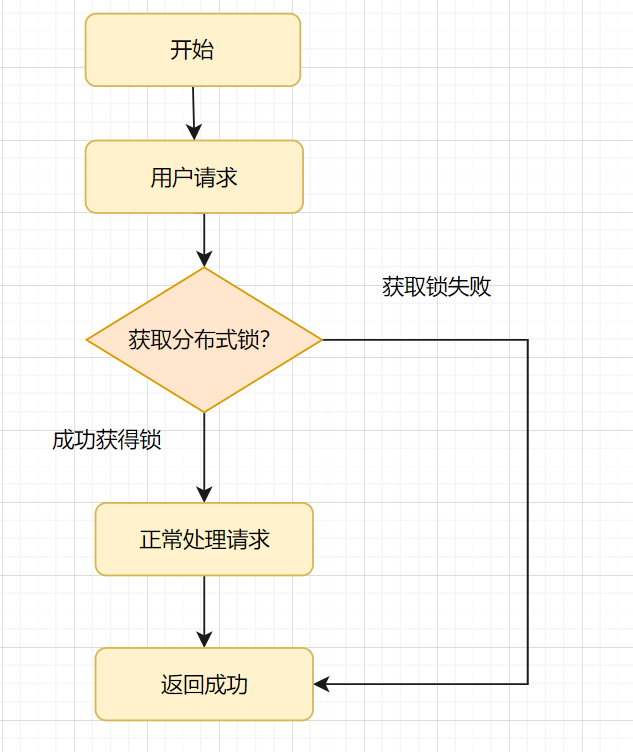


**为什么版本号建议自增的呢？**

因为乐观锁存在ABA的问题，如果version版本一直是自增的就不会出现ABA的情况啦。

### 5.8 分布式锁

分布式锁实现幂等性的逻辑就是，请求过来时，先去尝试获得分布式锁，如果获得成功，就执行业务逻辑，反之获取失败的话，就舍弃请求直接返回成功。执行流程如下图所示：



* 分布式锁可以使用Redis，也可以使用ZooKeeper，不过还是Redis相对好点，因为较轻量级。
* Redis分布式锁，可以使用命令SET EX PX NX + 唯一流水号实现，分布式锁的key必须为业务的唯一标识哈
* Redis执行设置key的动作时，要设置过期时间哈，这个过期时间不能太短，太短拦截不了重复请求，也不能设置太长，会占存储空间。

## 6. HTTP的幂等

我们的接口，一般都是基于http的，所以我们再来聊聊Http的幂等吧。HTTP 请求方法主要有以下这几种，我们看下各个接口是否都是幂等的。

* GET方法
* HEAD方法
* OPTIONS方法
* DELETE方法
* POST 方法
* PUT方法

我们的接口，一般都是基于http的，所以我们再来聊聊Http的幂等吧。HTTP 请求方法主要有以下这几种，我们看下各个接口是否都是幂等的。

* GET方法
* HEAD方法
* OPTIONS方法
* DELETE方法
* POST 方法
* PUT方法

### 6.1 GET 方法

HTTP 的GET方法用于获取资源，可以**类比**于数据库的select查询，不应该有副作用，所以是幂等的。它不会改变资源的状态，不论你调用一次还是调用多次，效果一样的，都没有副作用。

如果你的GET方法是获取最近最新的新闻，不同时间点调用，返回的资源内容虽然不一样，但是最终对资源本质是没有影响的哈，所以还是幂等的。

### 6.2 HEAD 方法

HTTP HEAD和GET有点像，主要区别是HEAD不含有呈现数据，而仅仅是HTTP的头信息，所以它也是**幂等**的。如果想判断某个资源是否存在，很多人会使用GET，实际上用HEAD则更加恰当。即HEAD方法通常用来做探活使用。

### 6.3 OPTIONS方法

HTTP OPTIONS 主要用于获取当前URL所支持的方法，也是有点像查询，因此也是幂等的。

### 6.4 DELETE方法

HTTP DELETE 方法用于删除资源，它是的**幂等的**。比如我们要删除id=666的帖子，一次执行和多次执行，影响的效果是一样的呢。

### 6.5 POST 方法

HTTP POST 方法用于创建资源，可以类比于提交信息，显然一次和多次提交是有副作用，执行效果是不一样的，**不满足幂等性**。

比如：POST http://www.tianluo.com/articles的语义是在http://www.tianluo.com/articles下创建一篇帖子，HTTP 响应中应包含帖子的创建状态以及帖子的 URI。两次相同的POST请求会在服务器端创建两份资源，它们具有不同的 URI；所以，**POST方法不具备幂等性**。

### 6.6 PUT 方法

HTTP PUT 方法用于创建或更新操作，所对应的URI是要创建或更新的资源本身，有副作用，它应该满足幂等性。

比如：PUT http://www.tianluo.com/articles/666的语义是创建或更新 ID 为666的帖子。对同一 URI 进行多次 PUT 的副作用和一次 PUT 是相同的；因此，PUT 方法具有幂等性。