幂等性常用的解决方案

在进行讲解方案之前，我想先说一下什么是幂等性，下面是我用自己的理解说的，不是专业的术语，对于同样的多次请求，只会对第一次请求进行处理，多次请求返回的结果是相同的。

幂等性的专业术语如下（选自百度百科）

> 幂等（idempotent、idempotence）是一个数学与计算机学概念，常见于抽象代数中。
> 在编程中一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。幂等函数，或幂等方法，是指可以使用相同参数重复执行，并能获得相同结果的函数。这些函数不会影响系统状态，也不用担心重复执行会对系统造成改变。例如，“setTrue()”函数就是一个幂等函数,无论多次执行，其结果都是一样的.
> 更复杂的操作幂等保证是利用唯一交易号(流水号)实现。

方案一、乐观锁

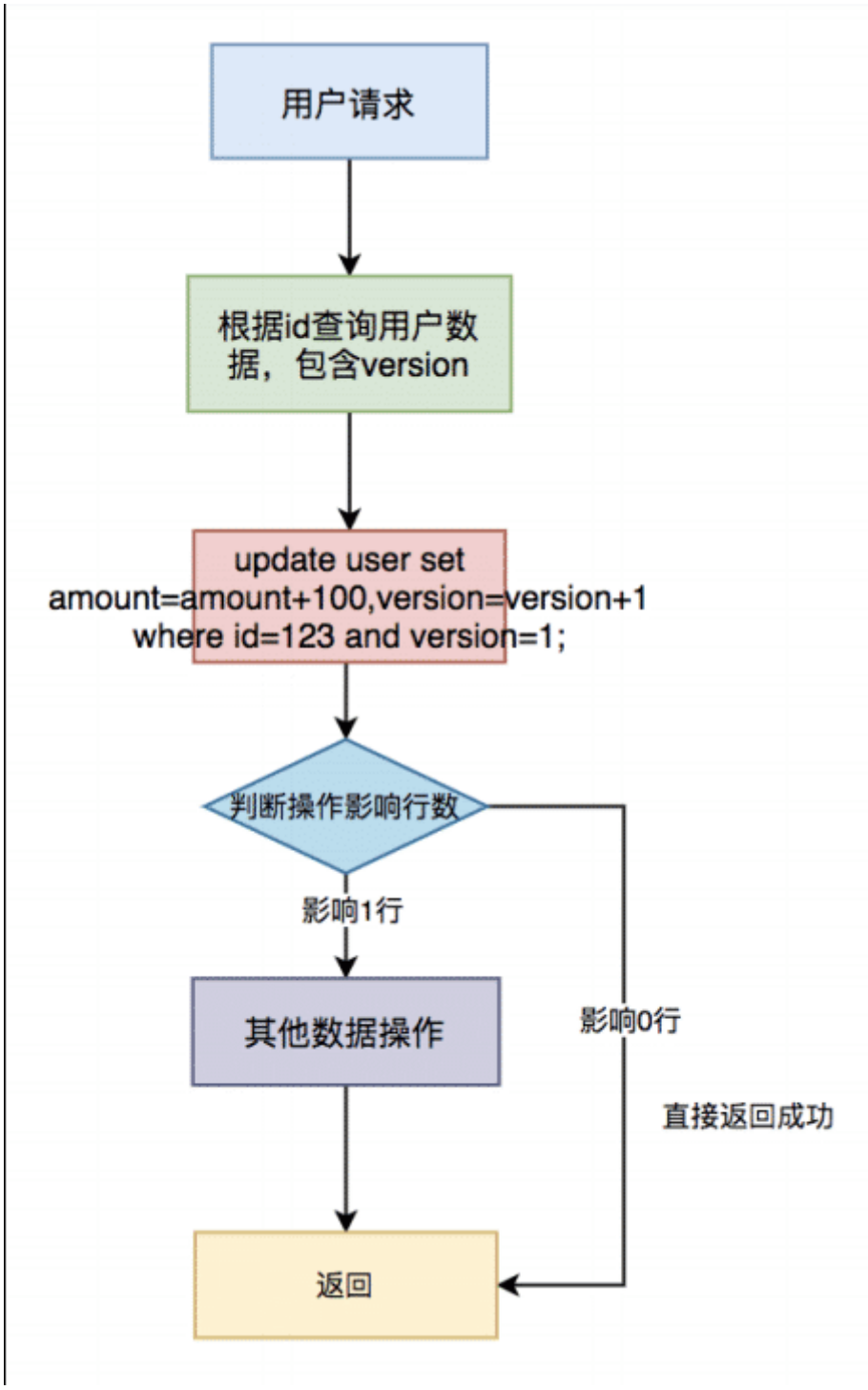
使用乐观锁的话，我们需要在数据库中设计一个用来作为版本号的字段，比如增加一个timeStamp或者version字段来作为数据库的字段，我们在每次更新操作的时候会判断这个版本号，只有版本号是一致的才可以进行更新，否则是无法进行更新的。

比如下面的sql流程

```
#1.查询我们要操作的数据,获得当前这行数据的版本号
select id,money,version from user where id=1;
#2.根据第一步获得的版本号来进行更新操作。
update user set money=money-100,version=version+1 where id=1 and version=1;
```

如果此时返回的结果是1说明操作成功了，如果返回的数据是0的话，说明当前这个请求没有进行处理。通过这个乐观锁保证了多个请求同时过来的时候只有一个请求可以执行。同时为了保证幂等性，如果返回的结果是0的话，我们也要返回成功的结果，因为幂等性要保证多次请求返回的结果是相同的。

流程图如下：



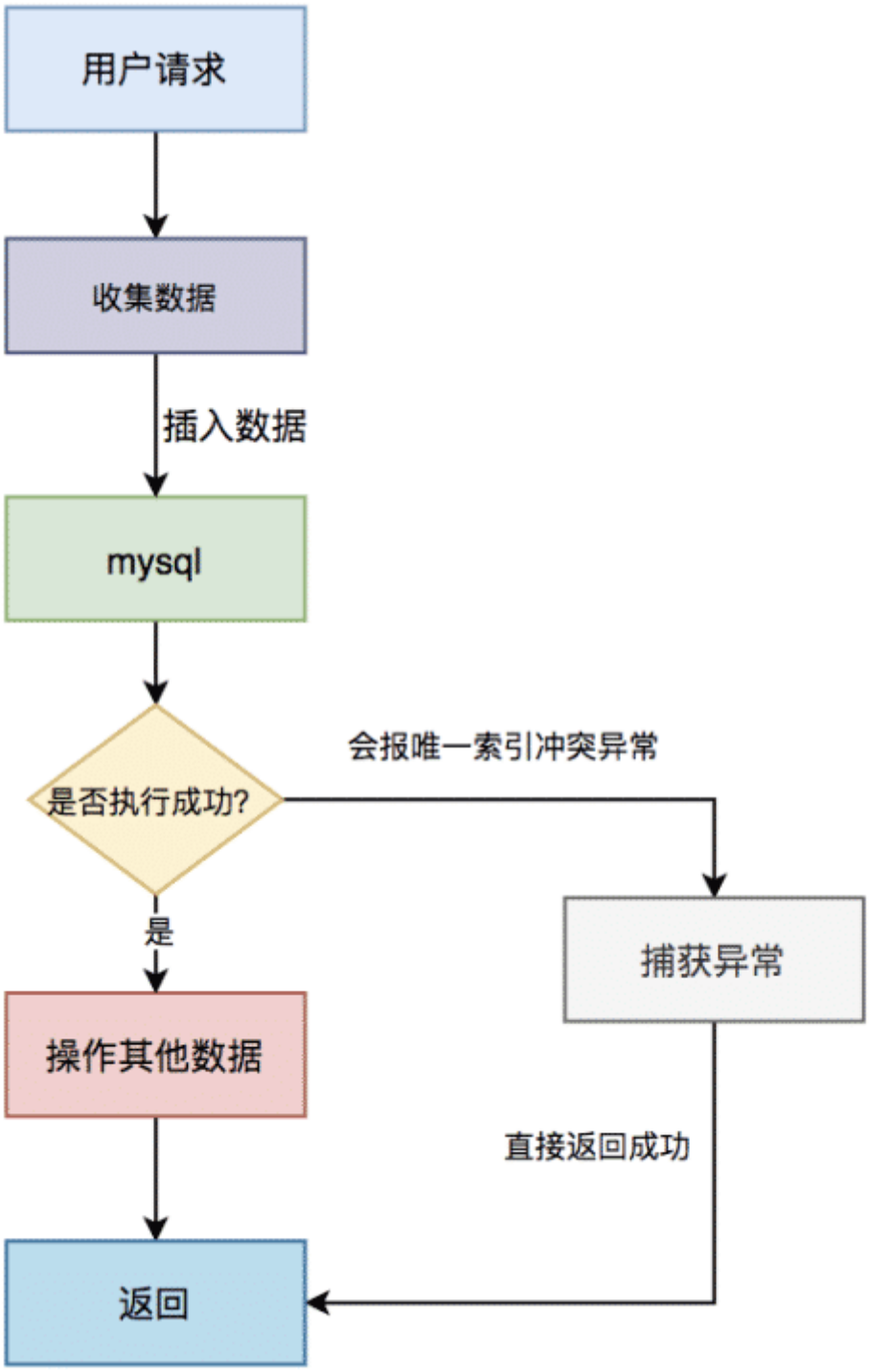
方案二、加唯一索引

加唯一索引可以保证同时有多个请求过来的时候只有一个请求可以执行插入操作，因为当一个请求已经插入了数据之后，其他请求再去插入数据的时候因为有唯一索引所以就会报错。为了保证幂等性，如果一个请求得到的结果是报错的话，我们后端要对这种异常进行处理来保证给前端返回的结果是请求成功，因为要保证多次请求返回的结果是一样的。

但是唯一索引的添加也是一个技术活，如果我们对一个单一的字段设置为唯一索引的话，如果你的这个业务场景种可以进行数据的真正删除的话那么这么设计是没有问题的，但是如果你的业务中要求你的数据不能真正的删除而是需要进行留档，用户的删除只是进行逻辑删除的话，我们设计的单一的字段为唯一索引的话，那么我们插入新数据的时候就会出现唯一索引冲突的问题了。

那么对于这种业务场景我们就要采用符合索引了，也就是将多个 字段作为唯一索引，比如我们将order_id和is_delete字段作为一个唯一索引。但是这种设计也是有问题的，比如如果我们的系统中的逻辑删除字段（is_delete）的值只有0和1，0代表没有删除，1代表删除了，那么这样的话，我们再插入新的数据的时候还是会出现唯一索引冲突的。那么解决这个问题的方案是什么呢？那就是我们可以扩大is_delete字段的值，比如如果这个字段已经删除了我们将is_delete的值设置为这一行数据的唯一主键id,如果没有删除的话就是0.那么这样的话就不会出现唯一索引冲突了。

加唯一索引方案的流程图



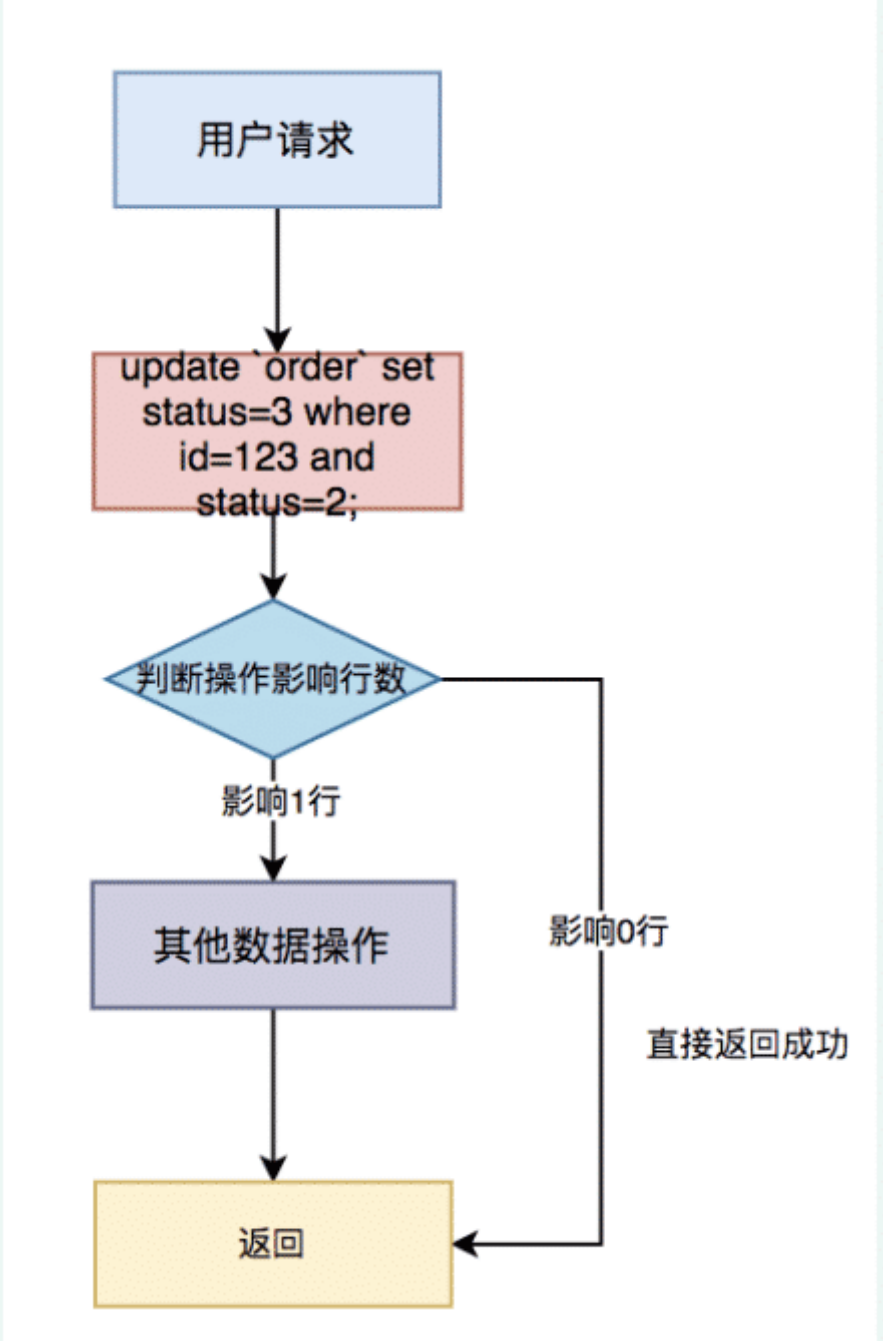
方案三、利用状态机

状态机可能看起来和乐观锁有点一样，但是有一个关键的区别，乐观锁的版本号是无界的，而状态机一般都是有边界的。而且如果你想要利用状态机来实现幂等性的话，一般是你的数据表中有状态这个字段，而且状态的变化是递增的，比如0是下单 1是已支付 2是完成支付。那么你的请求可以根据你数据表中的这个状态字段来进行更新操作。比如你想要将订单由已支付变成完成支付，你可以使用下面的sql语句

```
update user set status=2 where id=1 and status=1
```

第一次请求的时候你的状态是1是可以正常执行，将状态变成2的，后面再过来的请求再执行相同的sql的时候status就不是1了，就无法执行这个语句了。通过这种状态机的机制，就可以保证多个请求过来的时候只有第一个请求可以成功执行，后面过来的请求是会报错的，但是为了保证幂等性你需要将后面的请求返回给前端执行成功的结果。

利用状态机方案的流程图



方案四、加分布式锁

现在比较常用的分布式锁的框架就是Redis和zookeeper了。下面我来说一下redis中的方法。

1.setnx命令

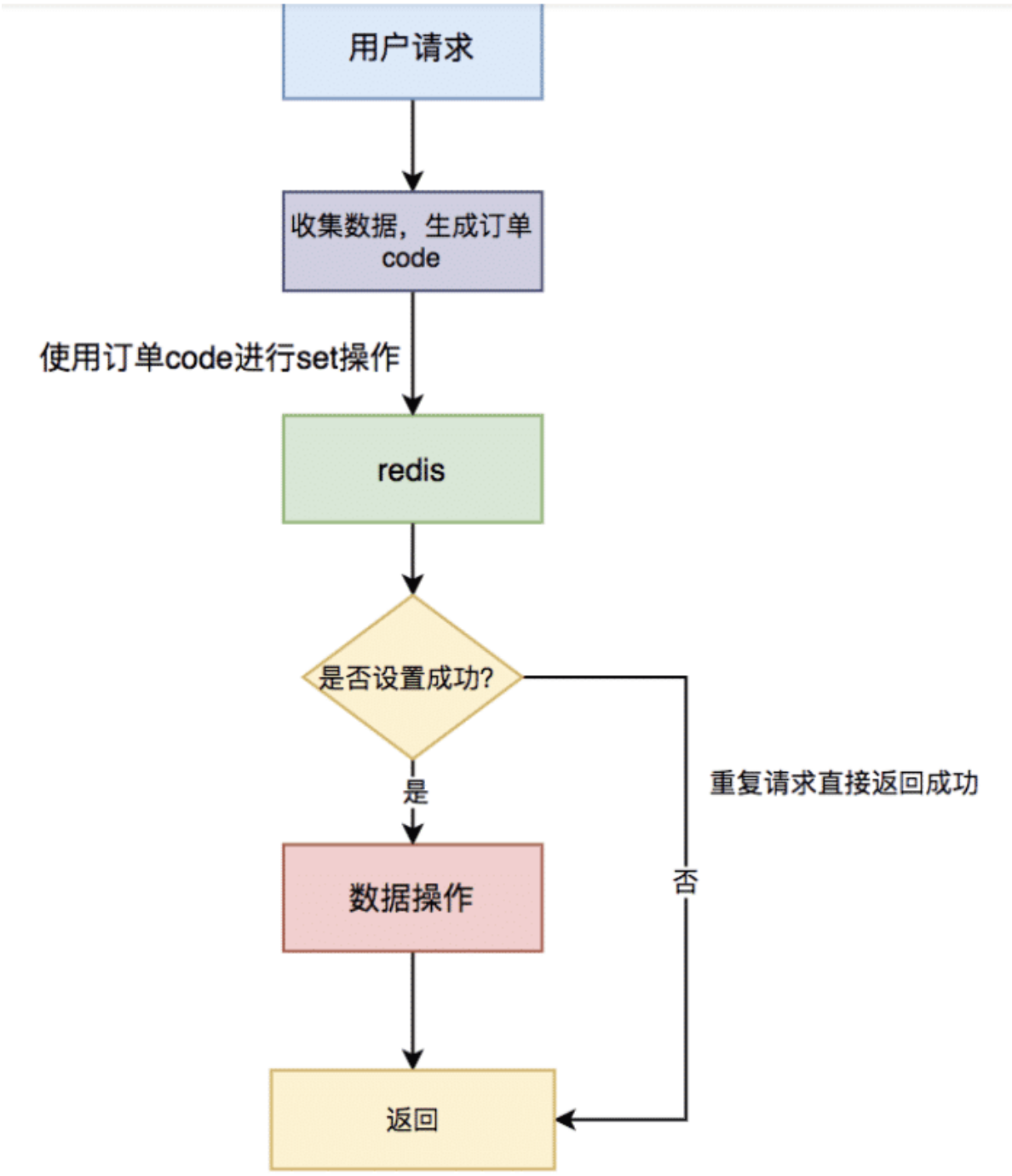
2.set命令

3.redisson

多个请求过来的时候我们只让第一个请求可以设置key,并且给这个key设置一个过期时间，后面再过来的请求发现已经有这个key了就无法设置key了。所以成功设置了key的请求就是成功执行的那个请求，后面没有成功设置key的请求就是没有成功执行的请求，但是我们还是要给前端返回成功的结果，因为要保证幂等性嘛。

zookeeper的解决方案的话，使用**Curator**框架就可以了。

分布式锁方案的流程图



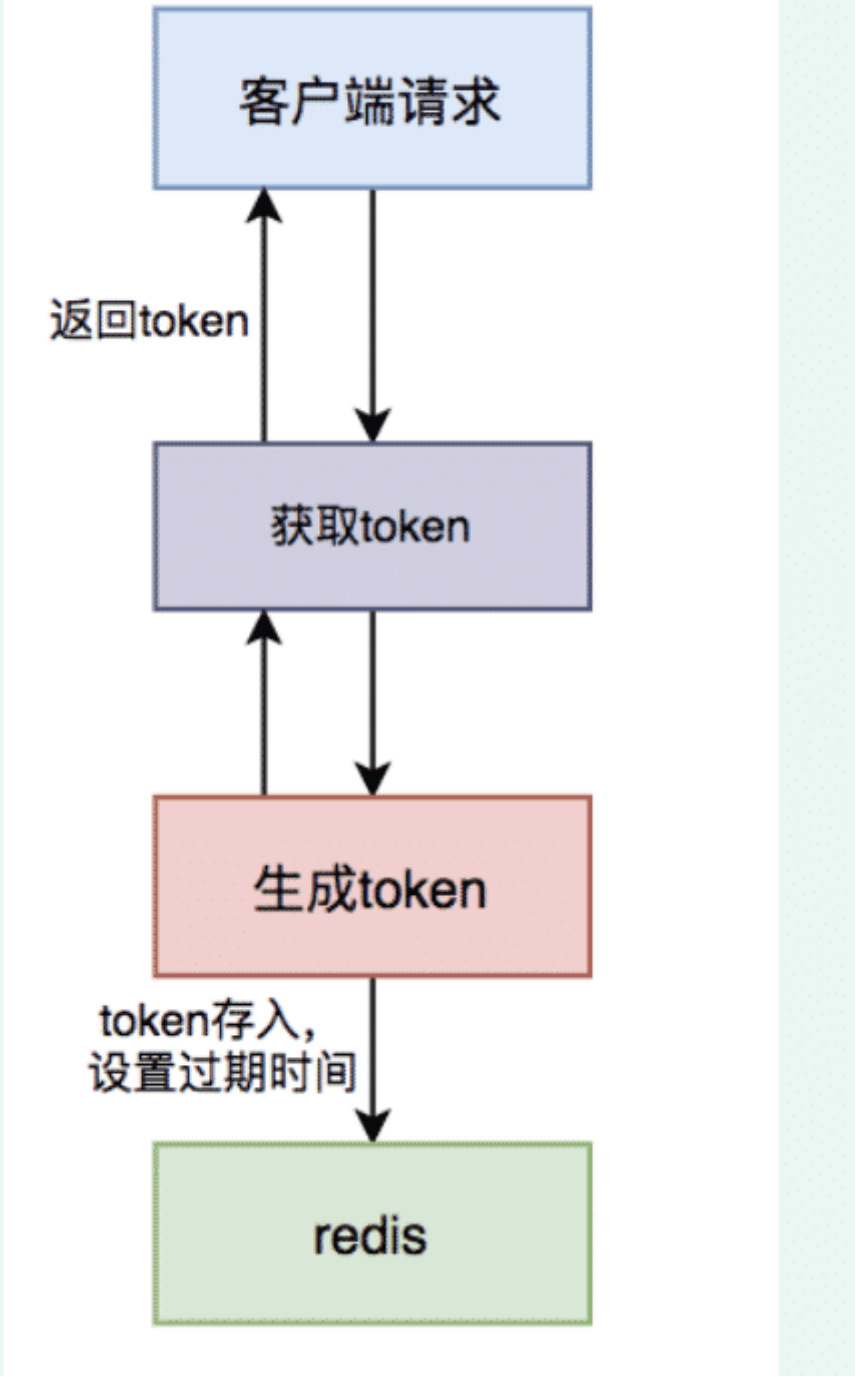
方案五、获取token

获取token的方案也是需要借助于Redis的。大体的流程是这样的，当前端发过来一个请求之后，后端会生成一个token,后端将这个token发送给前端，同时保存到redis中同时要记得设置过期时间。然后前端的每次请求都要携带这个token,如果是第一次发过来请求，那么将这个token从redis中删除。后序的请求携带了这个token过来，后端去redis中查询发现没有了这个token就意味着这不是第一次请求而是重复的请求，就不会执行，但是还是要给前端返回成功的结果，来保证幂等性。

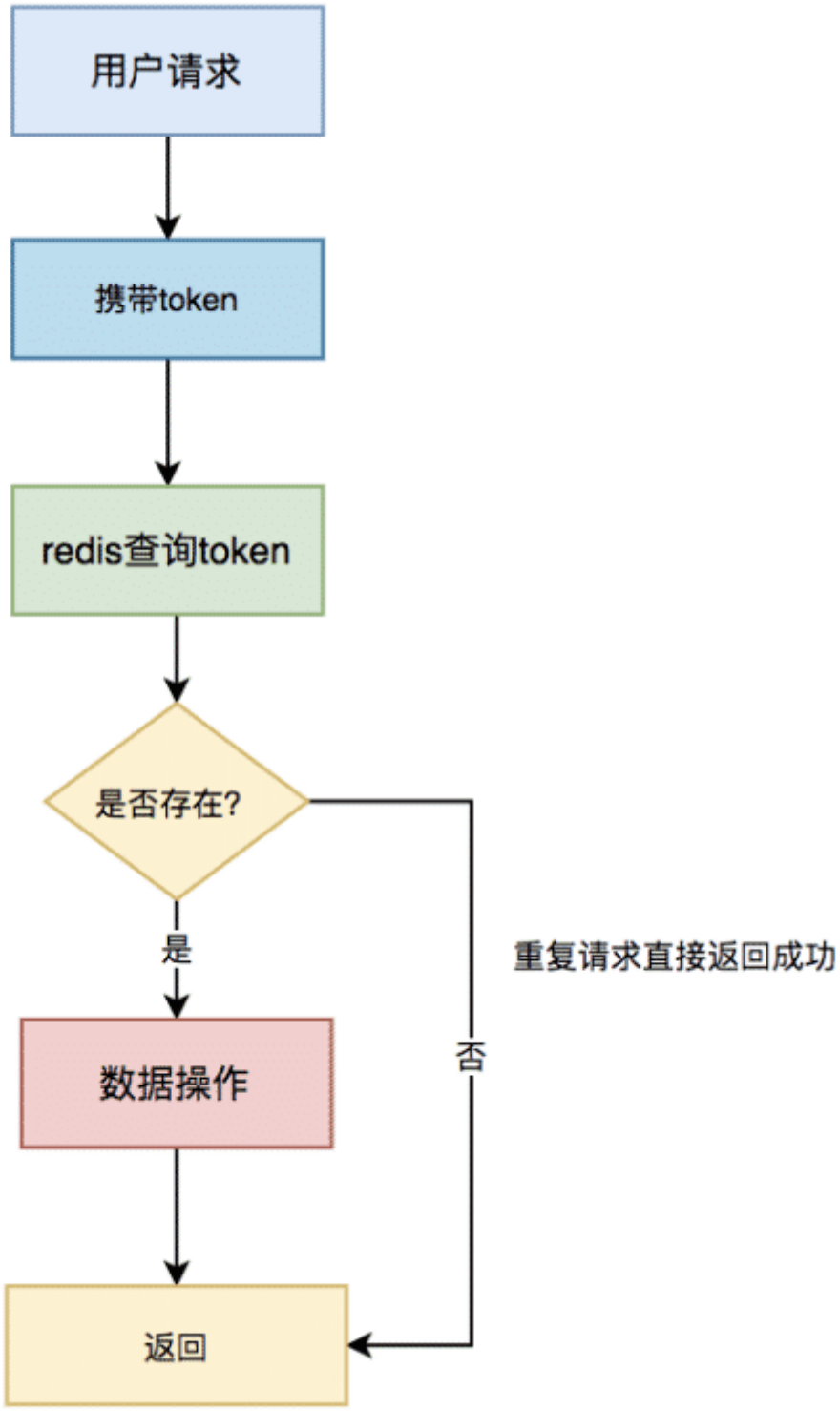
同时需要注意的一点就是，第一个请求来的时候查询redis中的token和删除这个token我们要使用lua脚本来封装成一个原子操作，这样才能保证只有第一个请求能够处理。

使用获取token方案的流程图

第一步，先获取token。



第二步，做具体业务操作


用户请求
携带token
redis查询token
是否存在?
是
否
重复请求直接返回成功
数据操作
返回