链滴

redis 分布式锁

作者：AshShawn

原文链接：https://ld246.com/article/1650706623701

来源网站：链滴

1.简单锁

redis的 setnx命令可以提供互斥,可以实现一个简单分布式锁

1.1 加锁

```
127.0.0.1:6379> SETNX lock 1
(integer) 1     // 客户端1，加锁成功

127.0.0.1:6379> SETNX lock 1
(integer) 0     // 客户端2，加锁失败
```

1.2 释放锁

del命令可以释放锁,

```
127.0.0.1:6379> DEL lock // 释放锁
(integer) 1
```

1.3 存在问题

死锁问题: 线程挂掉或程序异常,锁无法释放

2.简单锁+超时

setnx+expire 可以为分布式锁加一个超时时间,这样死锁问题可以解决

2.1 实现

```
127.0.0.1:6379> SETNX lock 1    // 加锁
(integer) 1
127.0.0.1:6379> EXPIRE lock 10  // 10s后自动过期
(integer) 1
```

2.2 问题

1. 锁过期: 线程1持有锁时,由于程序执行时间过长,导致锁过期,此时如果被其他线程获取到锁,易产生并问题

2. 释放其他线程的锁: 在1的情况下, 当线程1执行完程序后,很可能会把线程二的锁释放掉

3 简单锁+超时+线程id

3.1 实现

在加锁的时候 key为加锁资源, value为线程id, 解锁时校验线程,可以保证锁不会被其他线程释放

```
// 锁的VALUE设置为UUID
```

```
127.0.0.1:6379> SET lock $uuid EX 20 NX
OK

// 锁是自己的，才释放
if redis.get("lock") == $uuid:
    redis.del("lock")
```

3.2 问题

1. 释放锁时非原子操作, 先get再del, (问题不大,一般程序中并发请求都是setnx，如果用set则会产生题)

2. 锁过期问题依旧存在

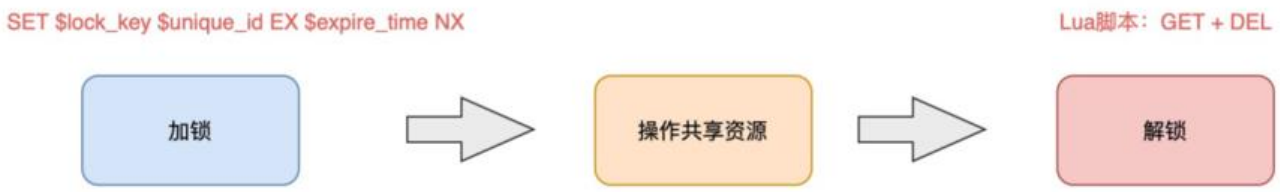
3.3 优化

使用lua脚本释放锁可以实现原子化

```
// 判断锁是自己的，才释放
if redis.call("GET",KEYS[1]) == ARGV[1]
then
    return redis.call("DEL",KEYS[1])
else
    return 0
end
```

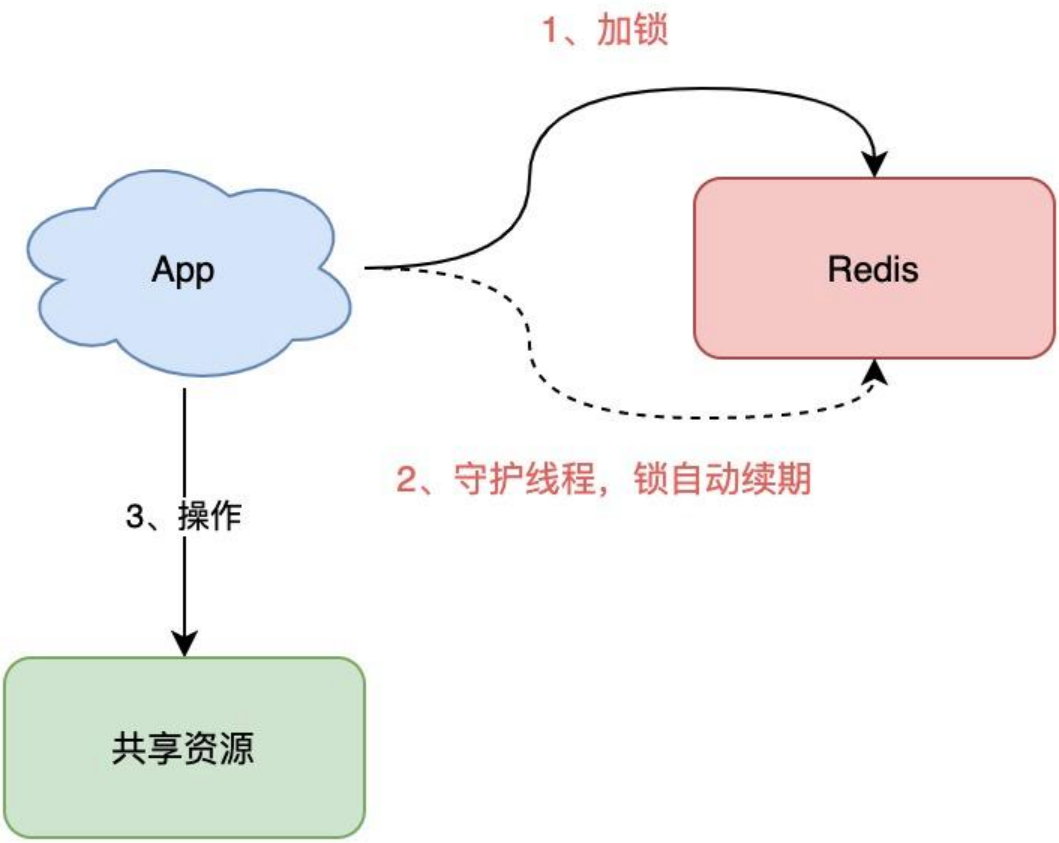
4. 锁过期问题

综上所述,使用setnx +超时+线程id+lua脚本 基本可以实现一个严谨的分布式锁



唯一问题似乎只有锁超时的问题了,锁超时问题可以这么处理:

加锁时，先设置一个过期时间，然后我们开启一个「守护线程」，定时去检测这个锁的失效时间，如锁快要过期了，操作共享资源还未完成，那么就自动对锁进行「续期」，重新设置过期时间。



业界成熟方案有redission,可以提供

- 可重入锁
- 乐观锁
- 公平锁
- 读写锁
- Redlock（红锁，下面会详细讲）

具体使用方法参考

0. 项目介绍 - 《Redisson 使用手册》 - 书栈网 · BookStack

5. redlock

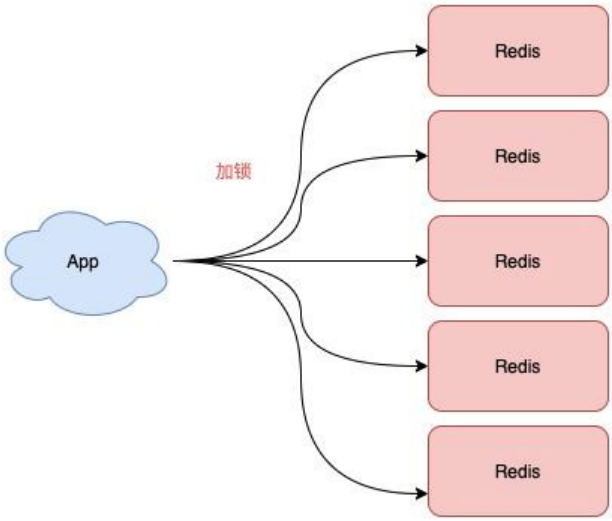
以上讨论仅限于单机redis, 一般我们在使用redis时,一般会采用主从集群+哨兵的模式,这样当发生故主从切换时, 以上分布式锁会存在安全性问题

1. 客户端在　主库上执行 SET 命令，加锁成功
2. 此时，主库异常宕机，SET 命令还未同步到从库上（主从复制是异步的）
3. 从库被哨兵提升为新主库，这个锁在新的主库上，丢失了！

5.1 redlock方案前提

redlock 前提:

1. 只部署从库
2. 至少5个实例



5.2 redlock加锁流程

1. 客户端先获取「当前时间戳T1」

2. 客户端依次向这 5 个 Redis 实例发起加锁请求，且每个请求会设置超时时间（毫秒级，要远小于的有效时间），如果某一个实例加锁失败（包括网络超时、锁被其它人持有等各种异常情况），就立向下一个 Redis 实例申请加锁

3. 如果客户端从 >=3 个（大多数）以上 Redis 实例加锁成功，则再次获取「当前时间戳T2」，如果2 - T1 < 锁的过期时间，此时，认为客户端加锁成功，否则认为加锁失败

4. 加锁成功，去操作共享资源（例如修改 MySQL 某一行，或发起一个 API 请求）

5. 加锁失败，向「全部节点」发起释放锁请求（前面讲到的 Lua 脚本释放锁）

5.3 重点分析

1. 客户端在多个 Redis 实例上申请加锁

多实例可以容错,避免单点故障

2. 必须保证大多数节点加锁成功

避免单点故障,只要大部分节点正常,那么整个系统是可以运行的

3. 大多数节点加锁的总耗时，要小于锁设置的过期时间

网络请求过程中会有延时丢包等情况出现,如果总耗时超过过期时间,那么整个锁就没意义了

4. 释放锁，要向全部节点发起释放锁请求

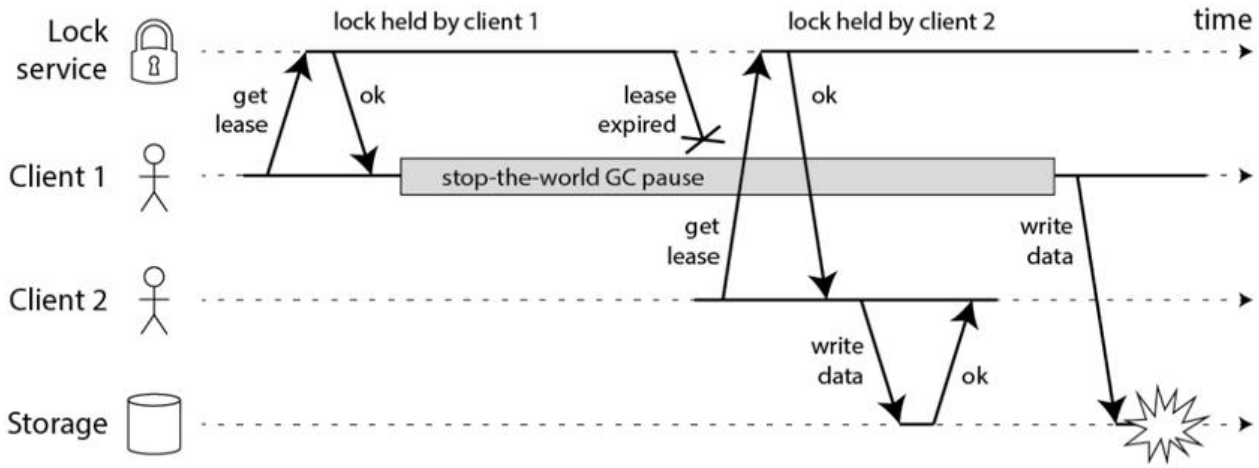
清除所有锁,确保每个节点都释放锁

5.4 redlock依旧存在问题

1. 网络延迟
2. 进程暂停(GC)
3. 时钟漂移

5.4.1 GC案例

1. 客户端 1 请求锁定节点 A、B、C、D、E
2. 客户端 1 的拿到锁后，进入 GC（时间比较久）
3. 所有 Redis 节点上的锁都过期了
4. 客户端 2 获取到了 A、B、C、D、E 上的锁
5. 客户端 1 GC 结束，认为成功获取锁
6. 客户端 2 也认为获取到了锁，发生「冲突」

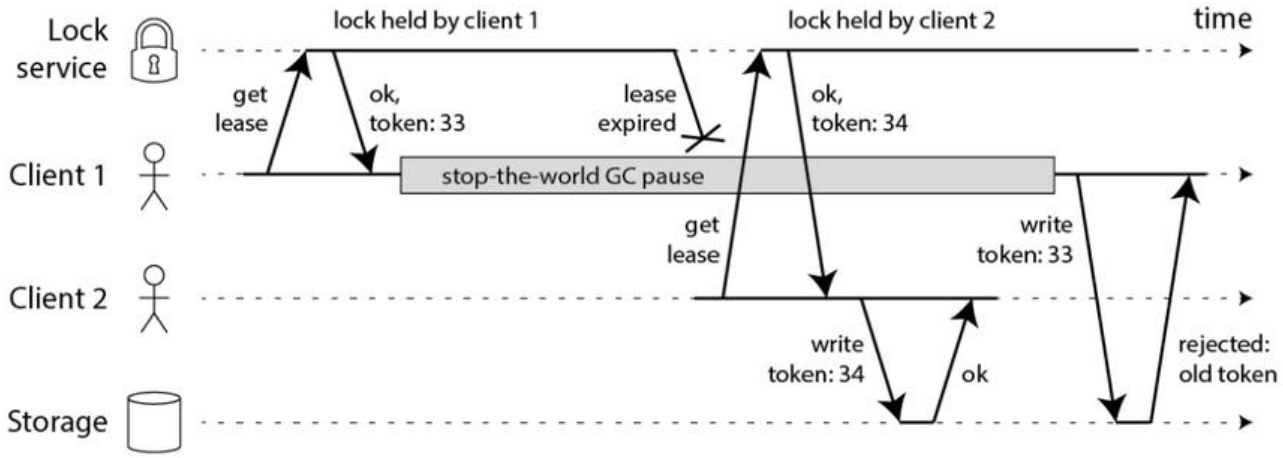


5.4.2 时钟漂移案例

1. 客户端 1 获取节点 A、B、C 上的锁，但由于网络问题，无法访问 D 和 E
2. 节点 C 上的时钟「向前跳跃」，导致锁到期
3. 客户端 2 获取节点 C、D、E 上的锁，由于网络问题，无法访问 A 和 B
4. 客户端 1 和 2 现在都相信它们持有了锁（冲突）

5.4.3 解决方案

1. 客户端在获取锁时，锁服务可以提供一个「递增」的 token
2. 客户端拿着这个 token 去操作共享资源
3. 共享资源可以根据 token 拒绝「后来者」的请求



6. zookeeper分布式锁安全吗

6.1 zookeeper分布式锁流程

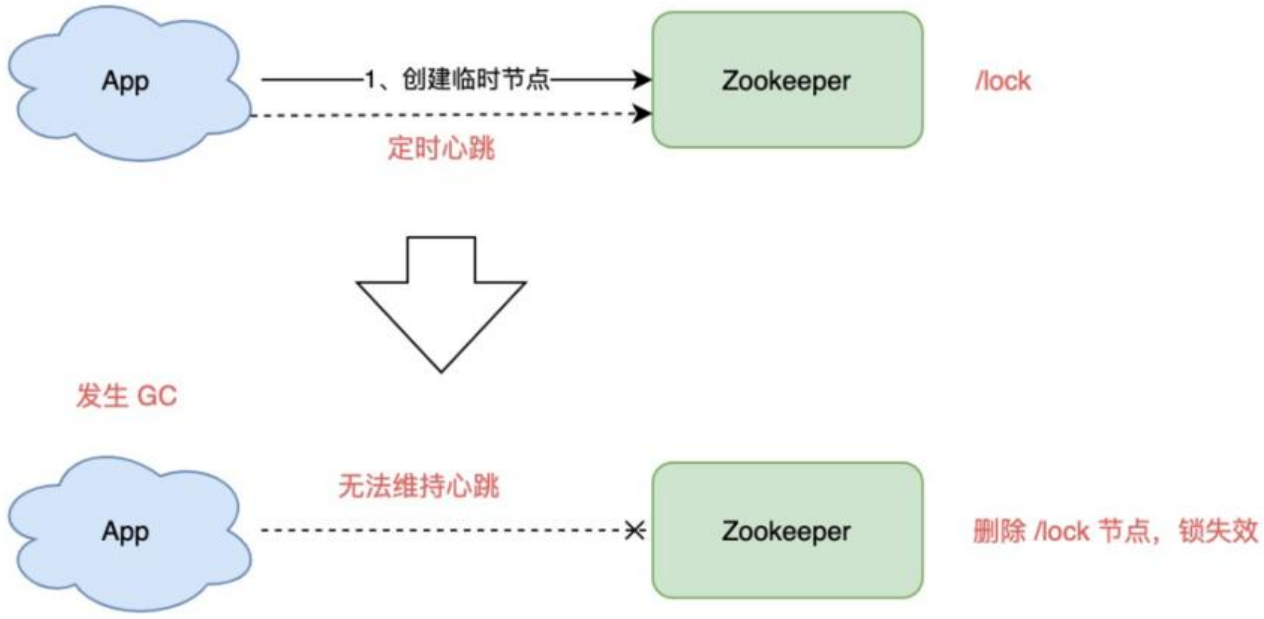
1. 客户端 1 和 2 都尝试创建「临时节点」，例如 /lock
2. 假设客户端 1 先到达，则加锁成功，客户端 2 加锁失败
3. 客户端 1 操作共享资源
4. 客户端 1 删除 /lock 节点，释放锁

zookeeper采用临时节点,当链接一直存在时,就可以持有锁,当客户端崩溃后,这个临时节点会自动删除,保证了锁一定会被释放

参考: (40条消息) Zookeeper原理--分布式锁_IT利刃出鞘的博客-CSDN博客_zookeeper分布式锁

6.2 问题

连接其实是依赖于心跳实现的,当客户端处于GC或者网络延迟,无法维持心跳也可能被zookeeper删除锁



原文链接：redis 分布式锁

1. 客户端 1 创建临时节点 /lock 成功，拿到了锁
2. 客户端 1 发生长时间 GC
3. 客户端 1 无法给 Zookeeper 发送心跳，Zookeeper 把临时节点「删除」
4. 客户端 2 创建临时节点 /lock 成功，拿到了锁
5. 客户端 1 GC 结束，它仍然认为自己持有锁（冲突）

6.3 zookeerper与redis分布式锁对比

Zookeeper 的优点：

1. 不需要考虑锁的过期时间
2. watch 机制，加锁失败，可以 watch 等待锁释放，实现乐观锁

但它的劣势是：

1. 性能不如 Redis
2. 部署和运维成本高
3. 客户端与 Zookeeper 的长时间失联，锁被释放问题

7. mysql分布式锁

7.1 悲观锁实现

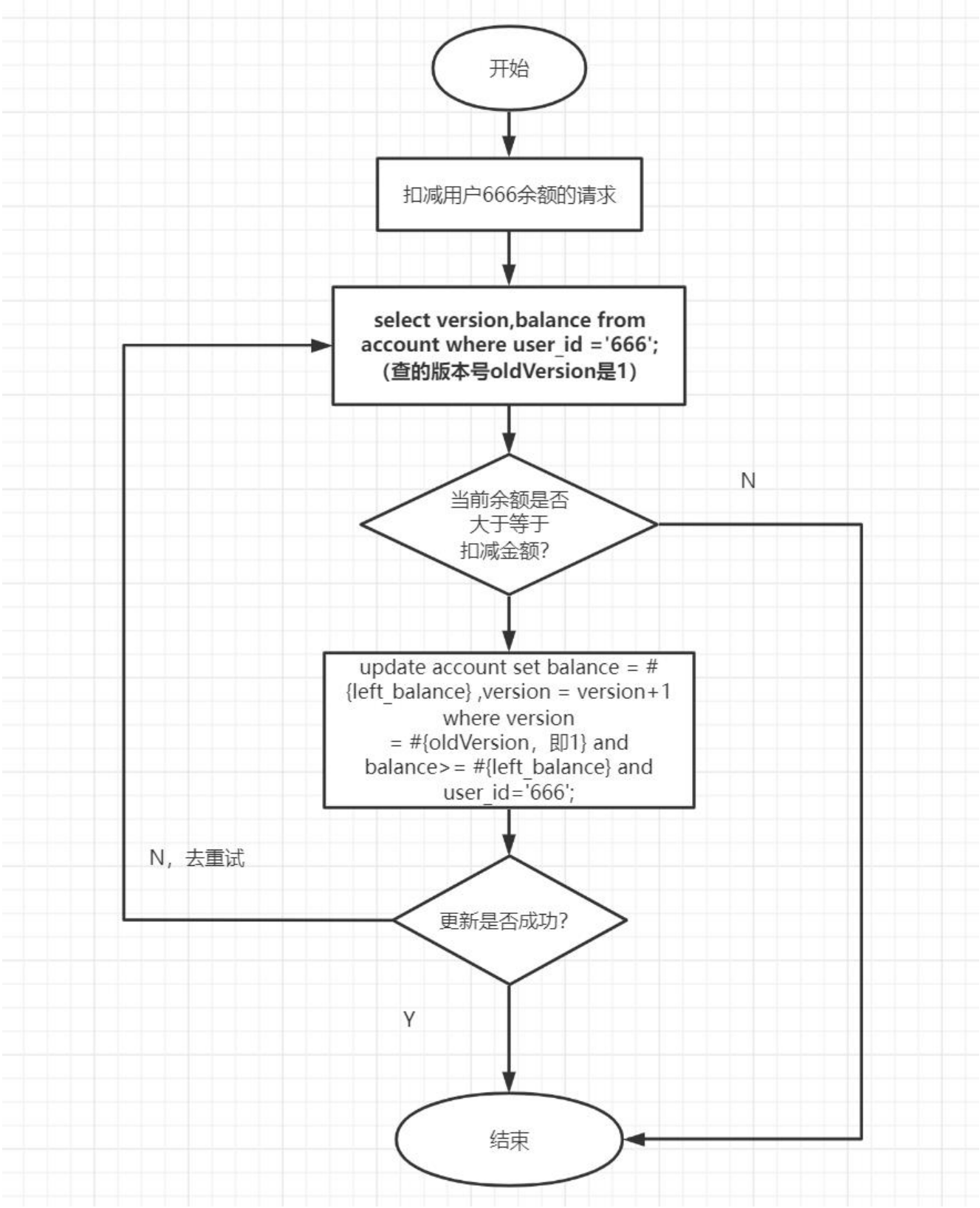
基于select ... for update实现

```
CREATE TABLE `t_resource_lock` (
  `key_resource` varchar(45) COLLATE utf8_bin NOT NULL DEFAULT '资源主键',
  `lock_flag` int(10) unsigned NOT NULL DEFAULT '0' COMMENT '1是已经锁 0是未锁',
  `begin_time` datetime DEFAULT NULL COMMENT '开始时间',
  `client_ip` varchar(45) COLLATE utf8_bin NOT NULL DEFAULT '抢到锁的IP',
  `time` int(10) unsigned NOT NULL DEFAULT '60' COMMENT '方法生命周期内只允许一个结点
取一次锁，单位：分钟',
  PRIMARY KEY (`key_resource`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin

@Transcational //一定要加事务
public boolean lock(String keyResource，int time){
   resourceLock = 'select * from t_resource_lock where key_resource ='#{keySource}' for updat
';

   try{
    if(resourceLock==null){
      //插入锁的数据
      resourceLock = new ResourceLock();
      resourceLock.setTime(time);
      resourceLock.setLockFlag(1);  //上锁
      resourceLock.setBeginTime(new Date());
      int count = "insert into resourceLock";
      if(count==1){
```

```
        //获取锁成功
        return true;
      }
      return false;
    }
    }catch(Exception x){
      return false;
    }

    //没上锁或者锁已经超时，即可以获取锁成功
    if(resourceLock.getLockFlag=='0'|| new Date()>=resourceLock.addDateTime(resourceLock.g
tBeginTime(),time)){
      resourceLock.setLockFlag(1);  //上锁
      resourceLock.setBeginTime(new Date());
      //update resourceLock;
      return true;
    }else{
      return false;
    }
}


public void unlock(String v，status){
      resourceLock.setLockFlag(0);  //解锁
      //update resourceLock;
      return ;
}
```

7.2乐观锁实现

```
//查询版本号和余额
select version,balance from account where user_id ='666';

//逻辑处理
if(balance<扣减金额){
   return;
}
left_balance = balance - 扣减金额;

//带版本号更新入库
update account set balance = #{left_balance} ,version = version+1 where version
= #{oldVersion} and balance>= #{left_balance} and user_id ='666';
```



8. 总结

redlock只有在时钟正确的情况下才能使用,而时钟漂移是无法避免的

1. 硬件因素,时间漂移无法避免
2. 运维因素,修改时间也会造成影响

原文链接：redis 分布式锁

那么如何正确使用redlock呢?可以基于fencing token方案

1. **使用分布式锁，在上层完成「互斥」目的，虽然极端情况下锁会失效，但它可以最大程度把并发请阻挡在最上层，减轻操作资源层的压力。**

2. **但对于要求数据绝对正确的业务，在资源层一定要做好「兜底」，设计思路可以借鉴 fecing token 的方案来做。**