

分布式锁的原理及实现方案

JackHu

开课吧

<https://www.kaikeba.com/>



章节内容

1. 线程&CPU&锁的关系
2. 分布式锁在业务中的应用场景
3. 分布式锁基本概念及基本特性
4. 锁和事务的根本区别
5. 使用DB实现分布式锁
6. Redis实现分布式锁的原理?
7. Zookeeper实现分布式的原理?
8. 接口幂等性设计



/01

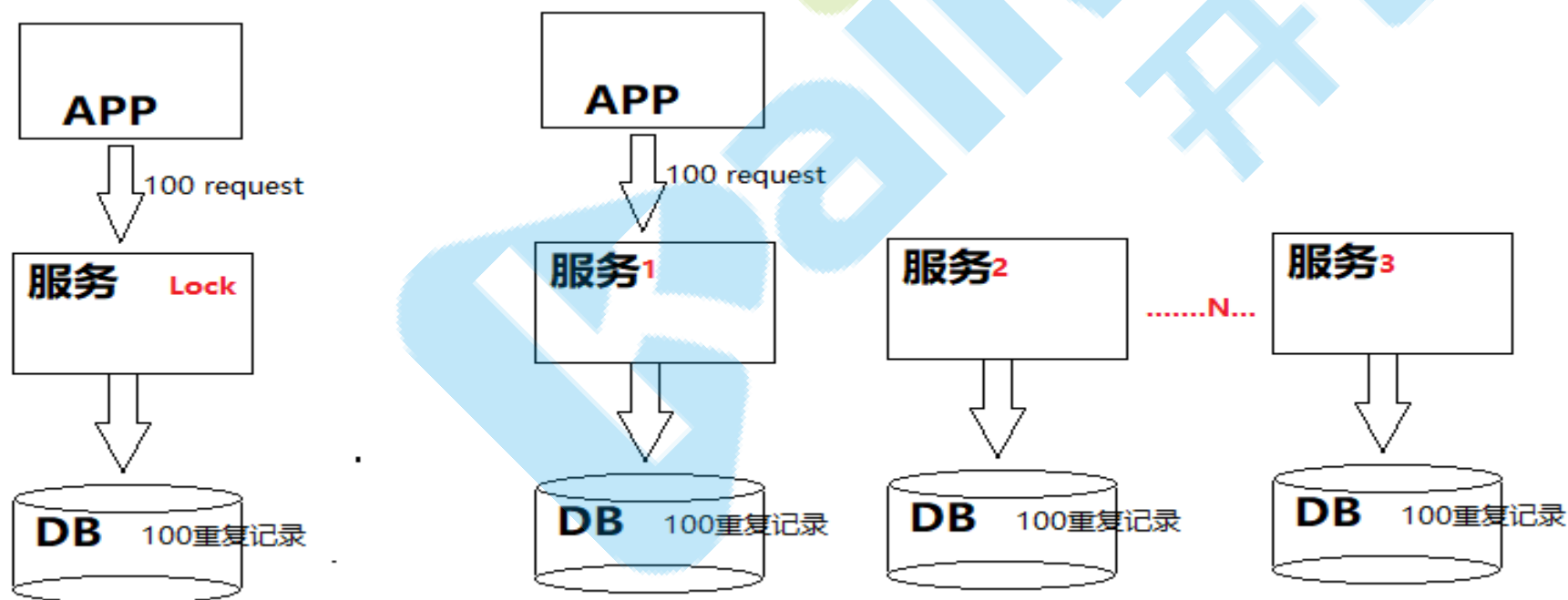
认识分布式锁的使用场景



为什么要用分布式锁? -- 业务场景-1

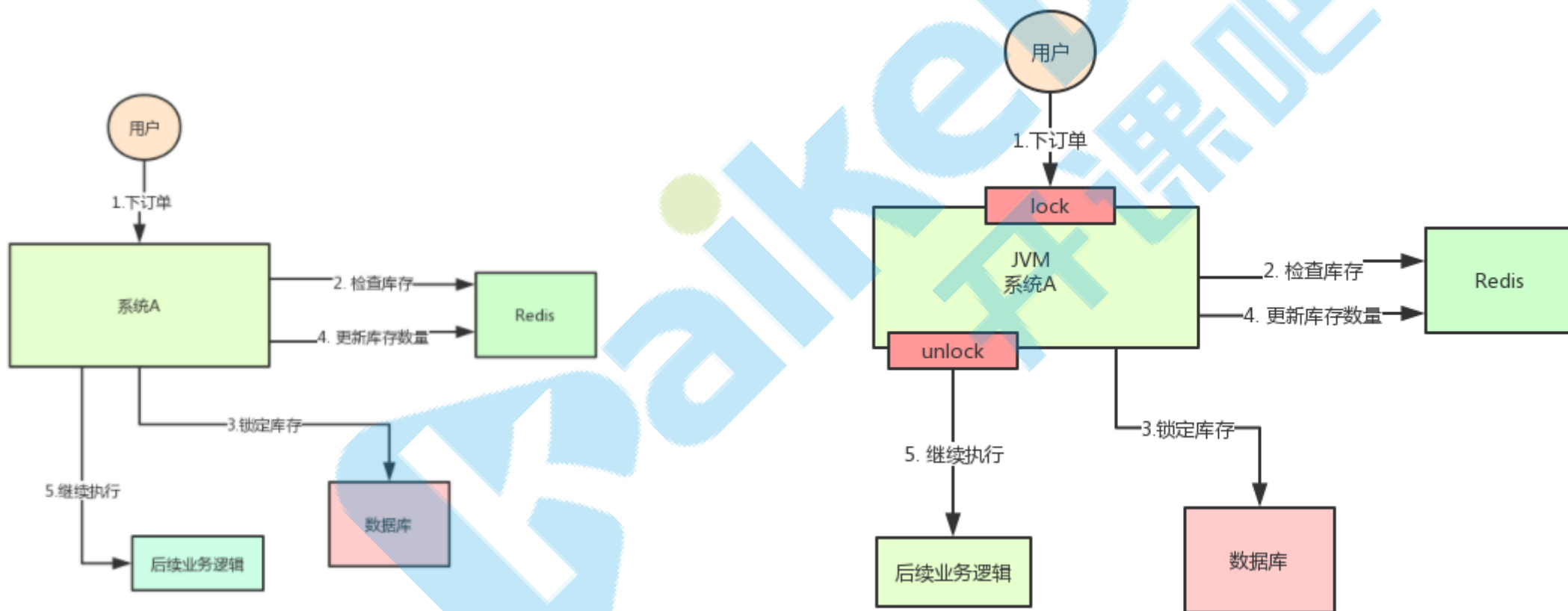
APP快速连续点击会向服务器连续发起请求，导致数据库出现重复数据（非阻塞锁）

- 表单重复提交
- 重复刷单
- APP重复请求



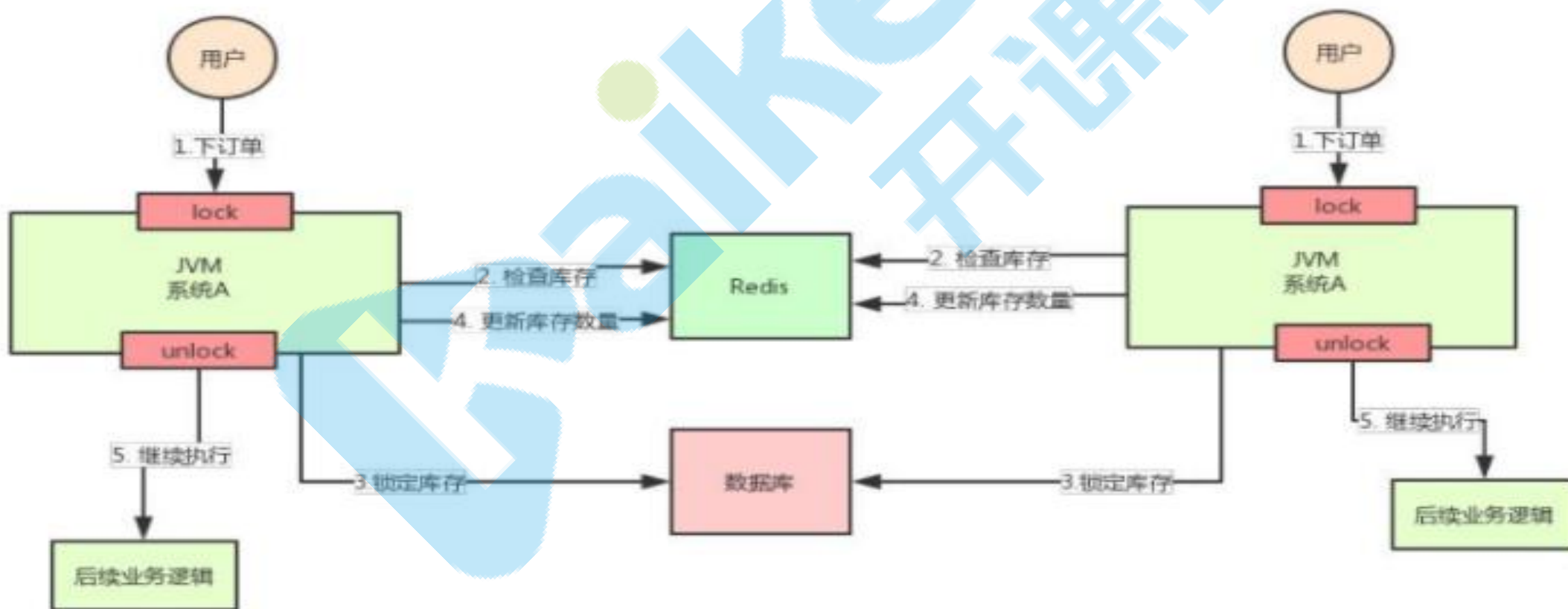
为什么要用分布式锁? -- 业务场景-2

用户下单库存超卖问题



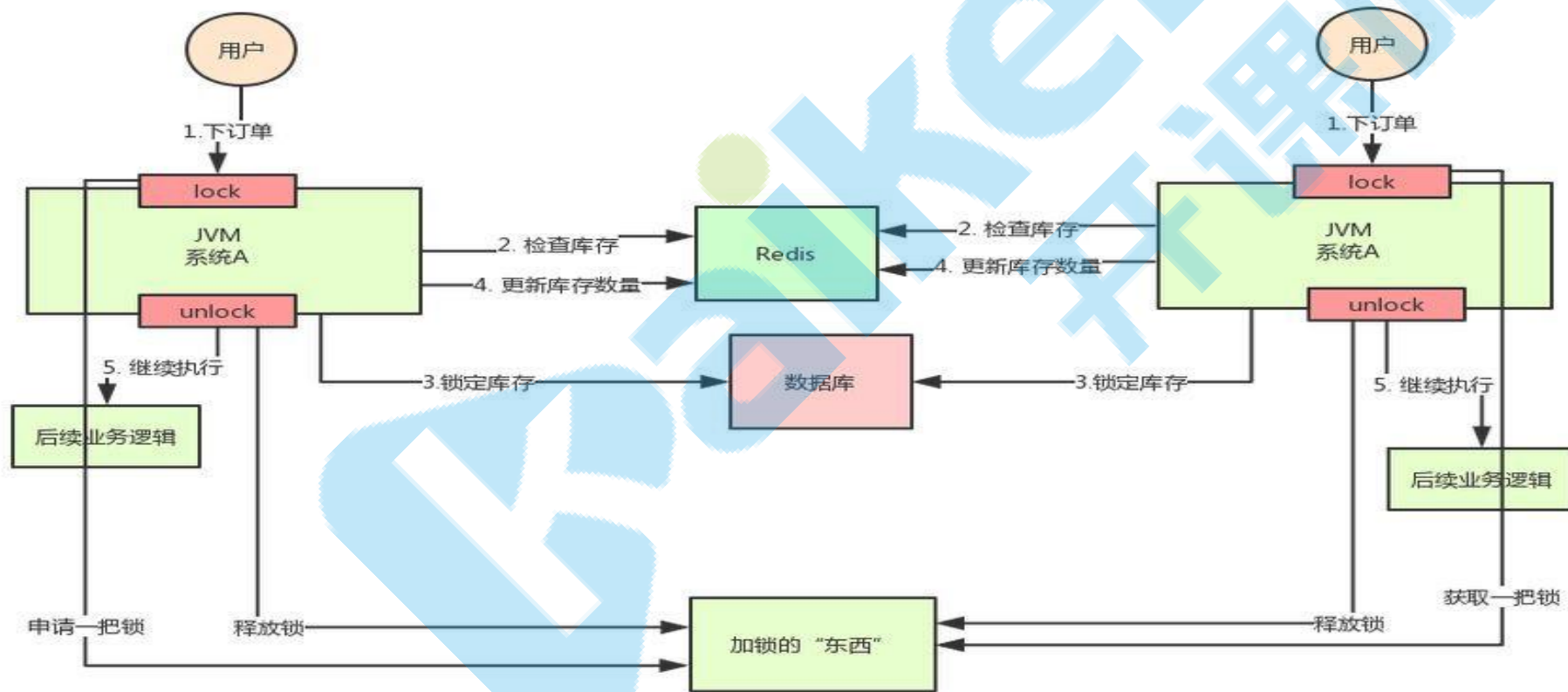
为什么要用分布式锁? -- 业务场景-2

用户下单库存超卖问题



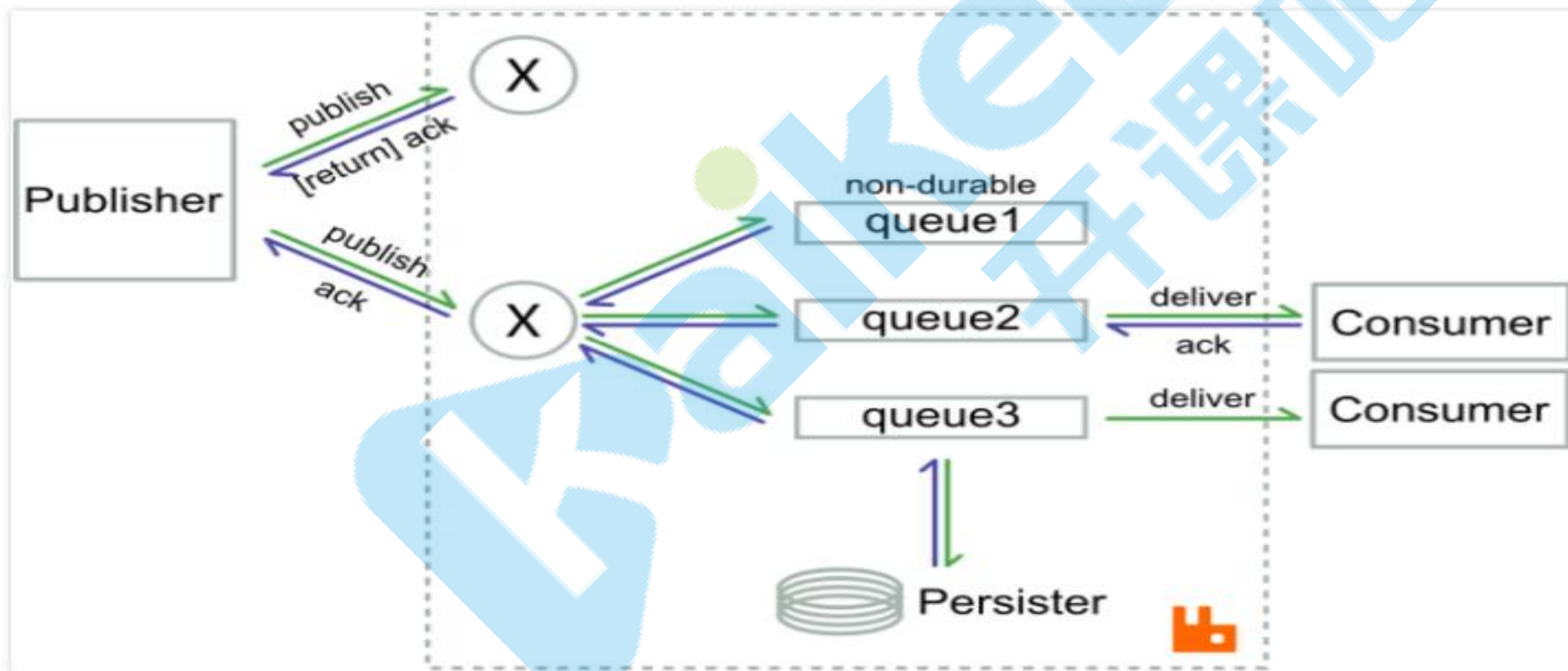
为什么要用分布式锁? -- 业务场景-2

用户下单库存超卖问题



为什么要用分布式锁? -- 业务场景-3

MQ消息去重



为什么要用分布式锁？ -- 其他业务

经典场景案例

- 秒杀
- 车票
- 退款
- 订单

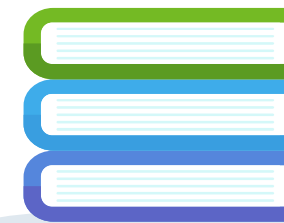
.....

无论是超卖，还是重复退款，都是没有对需要保护的资源或业务进行完善的保护而造成的，从设计方面一定要避免这种情况的发生



/02

分布式锁基本概念及基本特性



什么是分布式锁

- **单机锁（线程锁）**

synchronized、Lock

- **分布式锁（多服务共享锁）**

在分布式的部署环境下，通过锁机制来让多客户端互斥的对共享资源进行访问

分布式锁的基本概念

- **基本概念**

- * 多任务环境中才需要
- * 任务都需要对同一共享资源进行写操作;
- * 对资源的访问是互斥的

- **状态**

- * 任务通过竞争获取锁才能对该资源进行操作(①竞争锁);
- * 当有一个任务在对资源进行更新时 (②占有锁) ,
- * 其他任务都不可以对这个资源进行操作 (③任务阻塞) ,
- * 直到该任务完成更新(④释放锁);

- **特点**

- * 排他性: 在同一时间只会有一个客户端能获取到锁, 其它客户端无法同时获取
- * 避免死锁: 这把锁在一段有限的时间之后, 一定会被释放 (正常释放或异常释放)
- * 高可用: 获取或释放锁的机制必须高可用且性能佳

锁和事务的区别？

1) 锁:

单进程的系统中，存在多线程同时操作一个公共变量，此时需要加锁对变量进行同步操作，保证多线程的操作线性执行消除并发修改。解决的是单进程中的多线程并发问题。

2) 分布式锁:

只要的应用场景是在集群模式的多个相同服务，可能会部署在不同机器上，解决进程间安全问题，防止多进程同时操作一个变量或者数据库。解决的是多进程的并发问题

3) 事务

解决一个会话过程中，上下文的修改对所有**数据库**表的操作要么全部成功，要不全部失败。所以应用在service层。解决的是一个会话中的操作的数据一致性。

4) 分布式事务

解决一个联动操作，比如一个商品的买卖分为:

(1) 添加商品到购物车

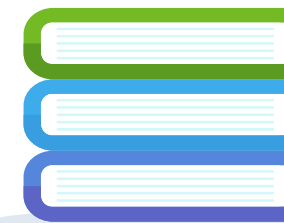
(2) 修改商品库存-1

此时购物车服务和商品库存服务可能部署在两台电脑，这时候需要保证对两个服务的操作都全部成功或者全部回退。解决的是组合服务的数据操作的一致性问题



/03

DB实现分布式锁方案



分布式锁的解决方案?

目前主流的有三种，从实现的复杂度上来看，从上往下难度依次增加：

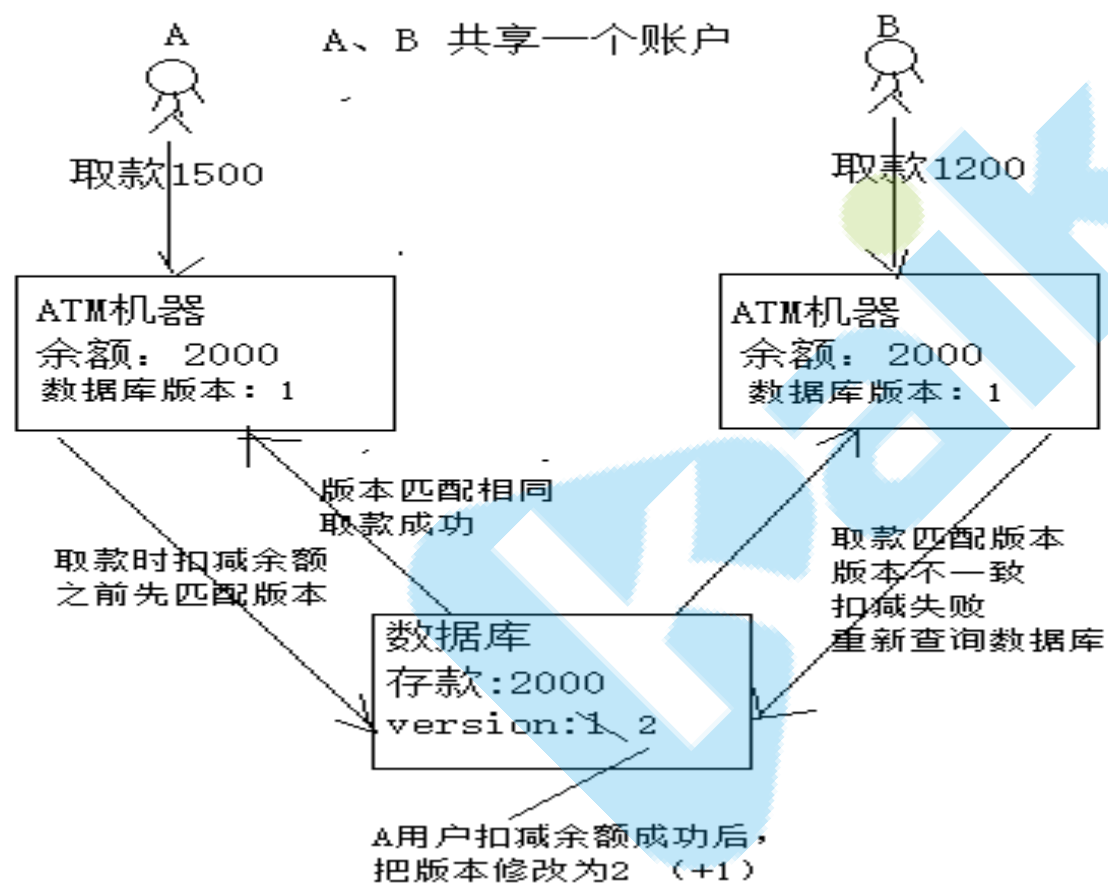
- 基于数据库实现
- 基于Redis实现
- 基于ZooKeeper实现

数据库(database)实现分布式锁

乐观锁

数据库实现分布式锁：乐观锁-----实现分布式锁

A、B 共享一个账户



数据库(database)实现分布式锁

悲观锁

```
//锁定的方法-伪代码
public boolean lock(){
    connection.setAutoCommit(false)
    for(){
        result =
        select * from user where
        id = 100 for update;
        if(result){
            //结果不为空，
            //则说明获取到了锁
            return true;
        }
        //没有获取到锁，继续获取
        sleep(1000);
    }
    return false;
}

//释放锁-伪代码
connection.commit();
```

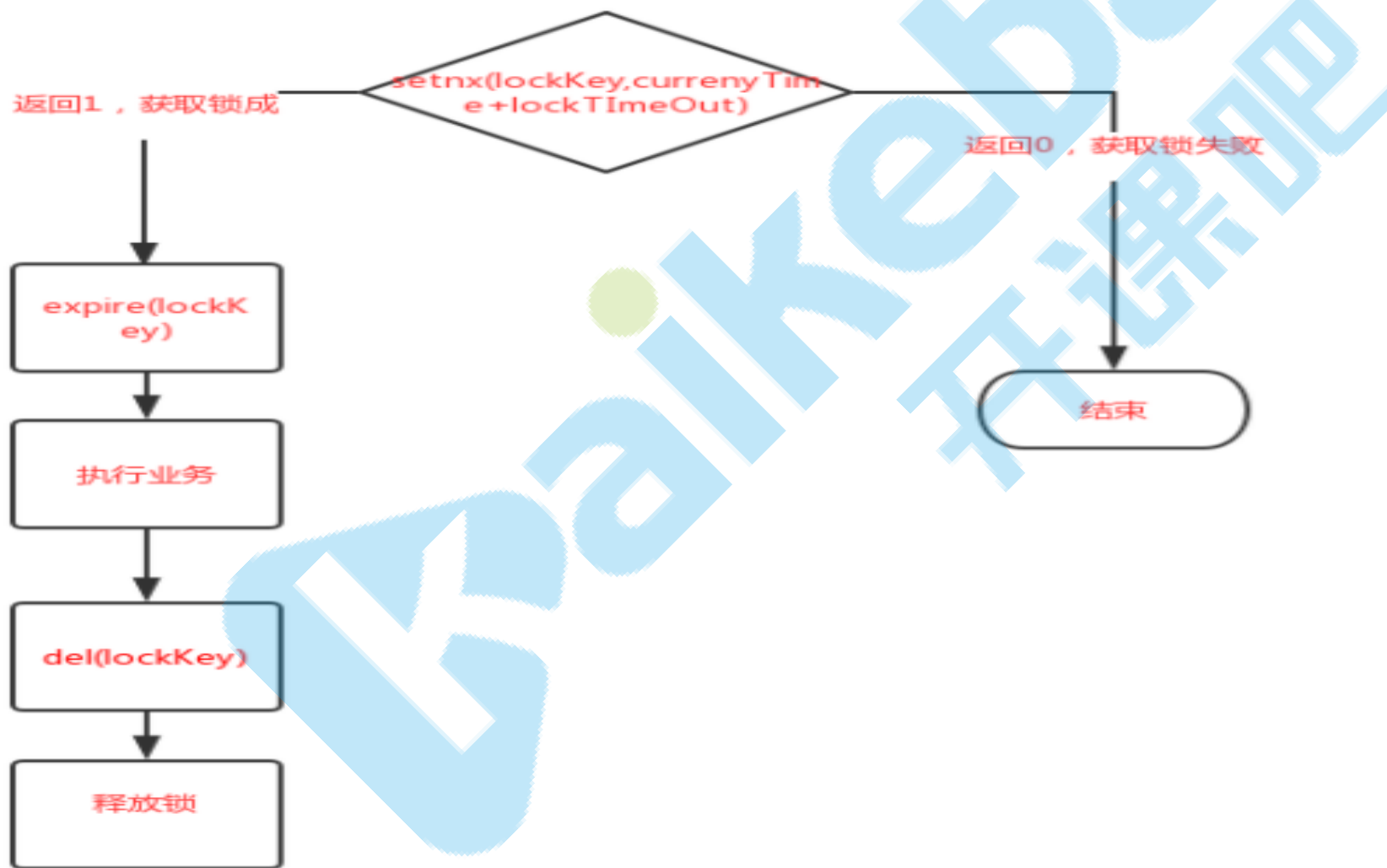


/04

Redis实现分布式锁方案



Redis方式实现分布式锁-获取锁



Redis方式实现分布式锁-获取锁

根据以上图示及思考，可的以下加锁代码：↵

```
↵
...↵
public static void wrongGetLock(Jedis jedis, String lockKey, String requestId, int expireTime) {↵
↵
    Long result = jedis.setnx(lockKey, requestId);↵
    if (result == 1) {↵
        // 若在这里程序突然崩溃，则无法设置过期时间，将发生死锁↵
        jedis.expire(lockKey, expireTime);↵
    }↵
}↵
...↵
↵
```

Redis方式实现分布式锁-获取锁-非原子操作

setnx和expire的非原子性

节点1的JVM进程



节点2的JVM进程



setnx



同步代码块



节点1的JVM进程



节点2的JVM进程



同步代码块



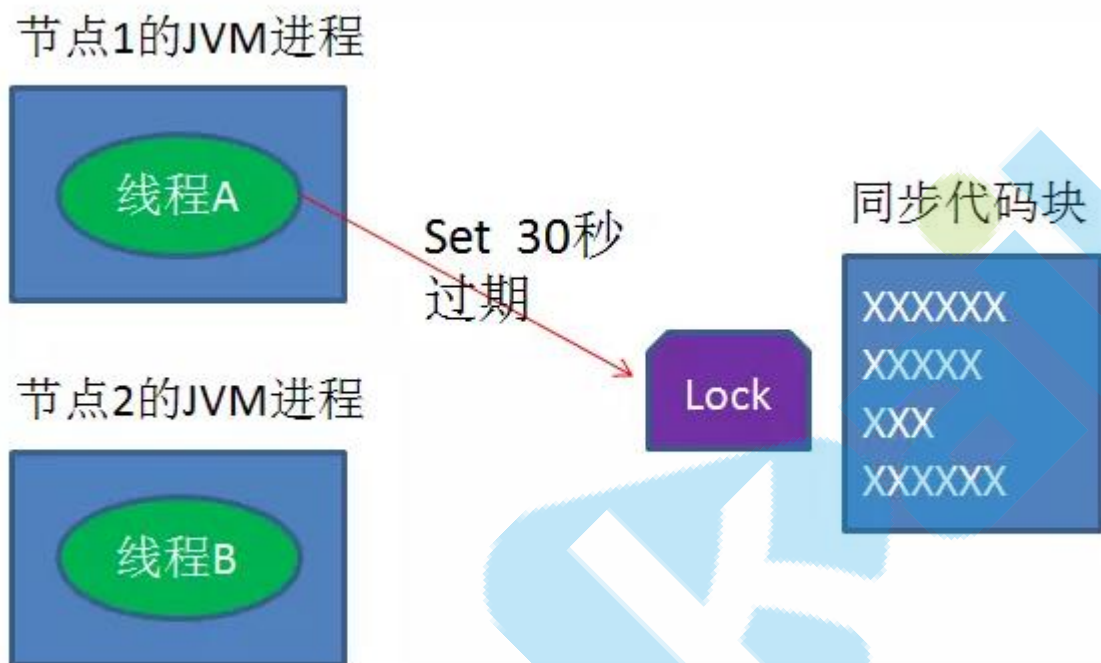
Redis方式实现分布式锁-获取锁-解决方案

- SET my_key my_value NX PX milliseconds (加锁)

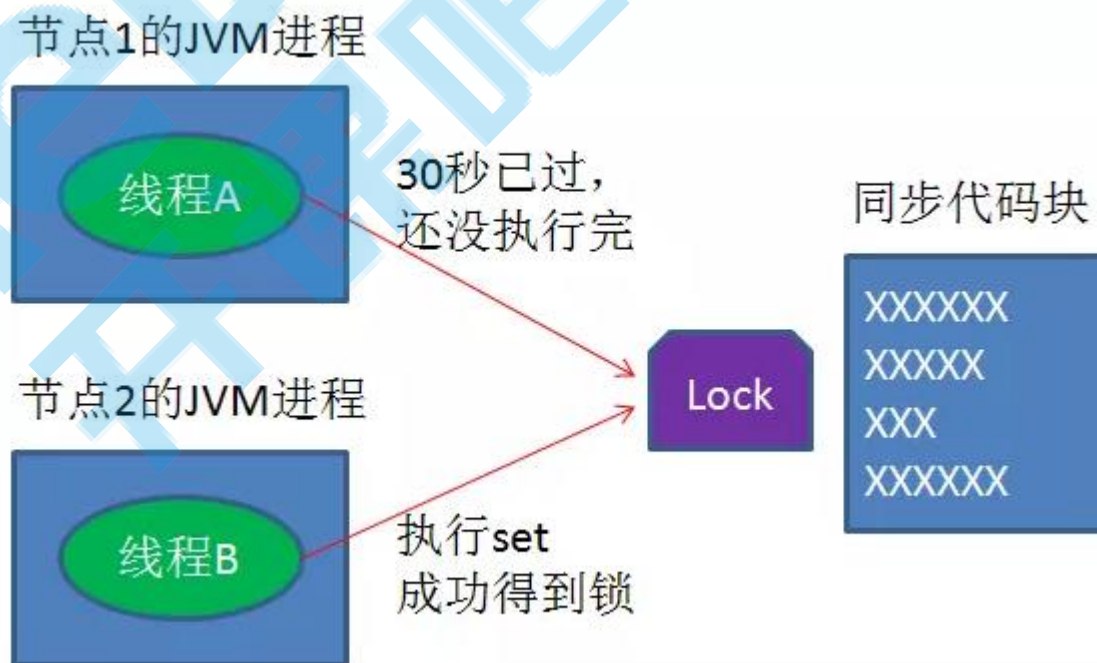
```
/**  
 * 尝试获取分布式锁  
 * @param jedis Redis 客户端  
 * @param lockKey 锁  
 * @param requestId 请求标识  
 * @param expireTime 超期时间  
 * @return 是否获取成功  
 */  
public boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {  
    String result = jedis.set(lockKey, requestId, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, expireTime);  
    if (LOCK_SUCCESS.equals(result)) {  
        return true;  
    }  
    return false;  
}
```

Redis方式实现分布式锁-释放锁-错误删除锁

线程成功得到了锁，并且设置的超时时间是30秒



线程A执行的很慢很慢，过了30秒都没执行完，这时候锁过期自动释放，线程B得到了锁。



Redis方式实现分布式锁-释放锁-错误删除锁

线程A执行完了任务，线程A接着执行del指令来释放锁。但这时候线程B还没执行完，**线程A实际上删除的是线程B加的锁**

节点1的JVM进程

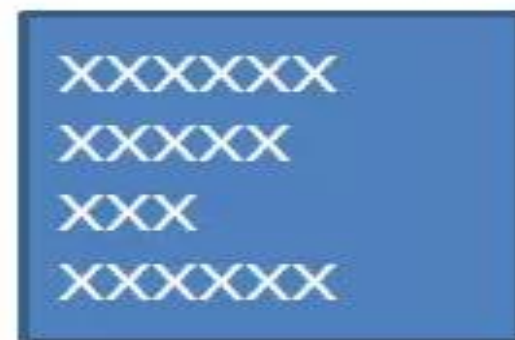


执行完了，
Del!

节点2的JVM进程



同步代码块



Redis方式实现分布式锁-释放锁-错误删除锁

线程A执行完了任务，线程A接着执行del指令来释放锁。但这时候线程B还没执行完，**线程A实际上删除的是线程B加的锁**

节点1的JVM进程



执行完了，
Del!

节点2的JVM进程



同步代码块



Redis方式实现分布式锁-释放锁-错误删除锁-解决方案

加锁的时候把当前的线程ID当做value，并在删除之前验证key对应的value是不是自己线程的ID。

加锁：

```
String threadId = Thread.currentThread().getId()
```

```
set ( key , threadId , 30 , NX )
```

解锁：

```
if ( threadId .equals(redisClient.get(key)) ) {
```

```
del(key)
```

```
}
```

Redis方式实现分布式锁-释放锁-错误删除锁-解决方案

- Lua脚本释放锁，保证释放锁的方法的原子性

```
/**  
 * 释放分布式锁  
 * @param jedis Redis 客户端  
 * @param lockKey 锁  
 * @param requestId 请求标识  
 * @return 是否释放成功  
 */  
public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {  
  
    String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";  
    Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));  
  
    if (RELEASE_SUCCESS.equals(result)) {  
        return true;  
    }  
    return false;  
}
```

Redis方式实现分布式锁-释放锁-错误删除锁-解决方案

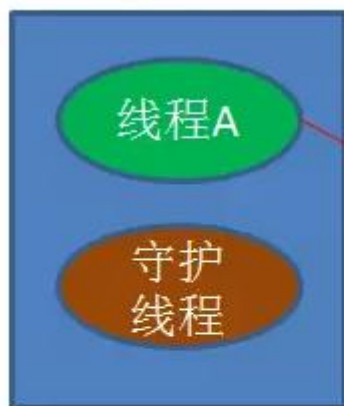
- Lua脚本释放锁，保证释放锁的方法的原子性

```
/**  
 * 释放分布式锁  
 * @param jedis Redis 客户端  
 * @param lockKey 锁  
 * @param requestId 请求标识  
 * @return 是否释放成功  
 */  
public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {  
  
    String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";  
    Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));  
  
    if (RELEASE_SUCCESS.equals(result)) {  
        return true;  
    }  
    return false;  
}
```

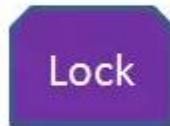
Redis方式实现分布式锁-释放锁-锁续航问题

获得锁的线程开启一个**守护线程**，用来给快要过期的锁“续航”

节点1的JVM进程



Set 30秒
过期



同步代码块



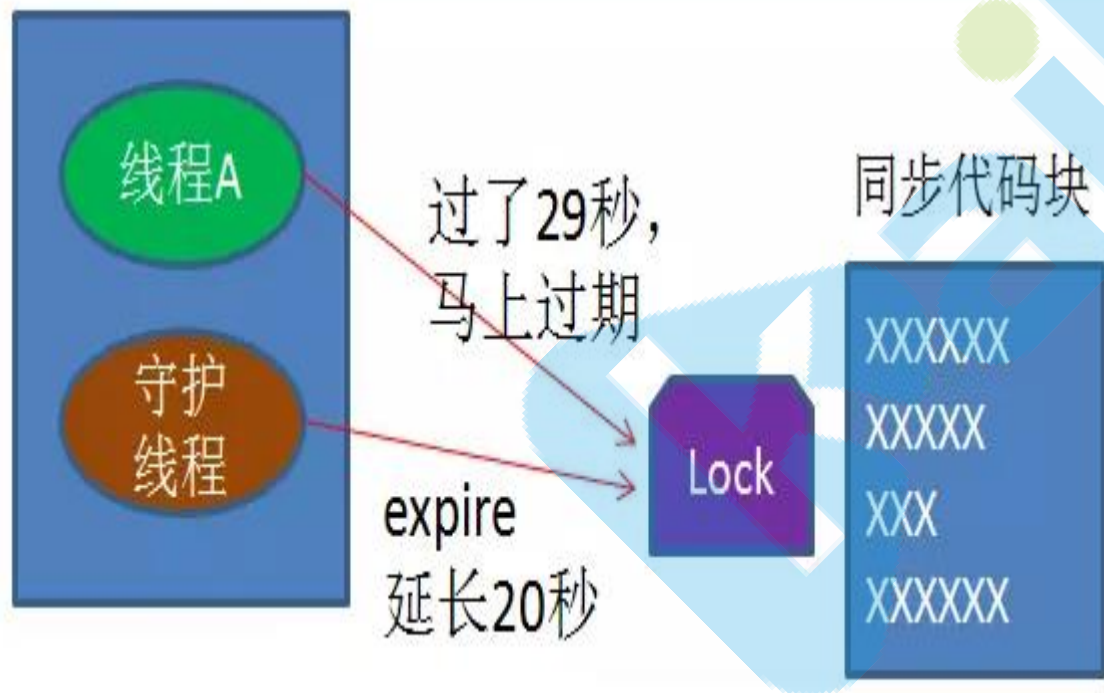
节点2的JVM进程



Redis方式实现分布式锁-释放锁-锁续航问题

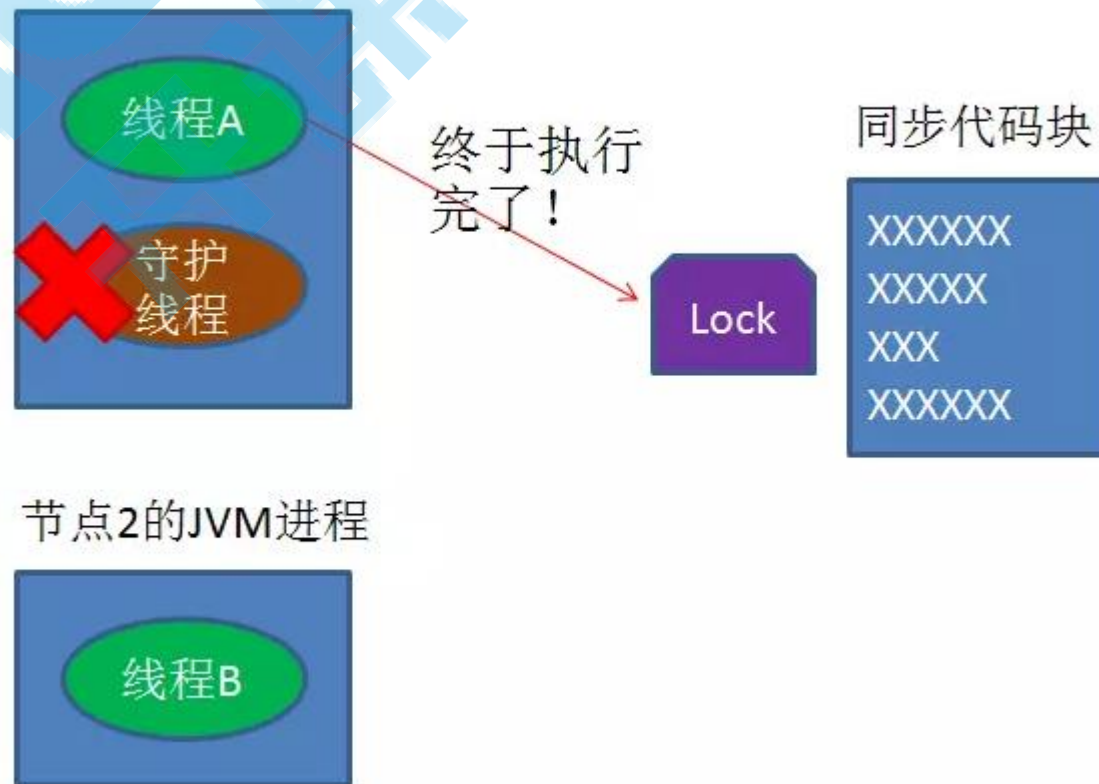
过去了29秒，线程A还没执行完，这时候守护线程会执行expire指令，为这把锁“续命”20秒。守护线程从第29秒开始执行，每20秒执行一次

节点1的JVM进程



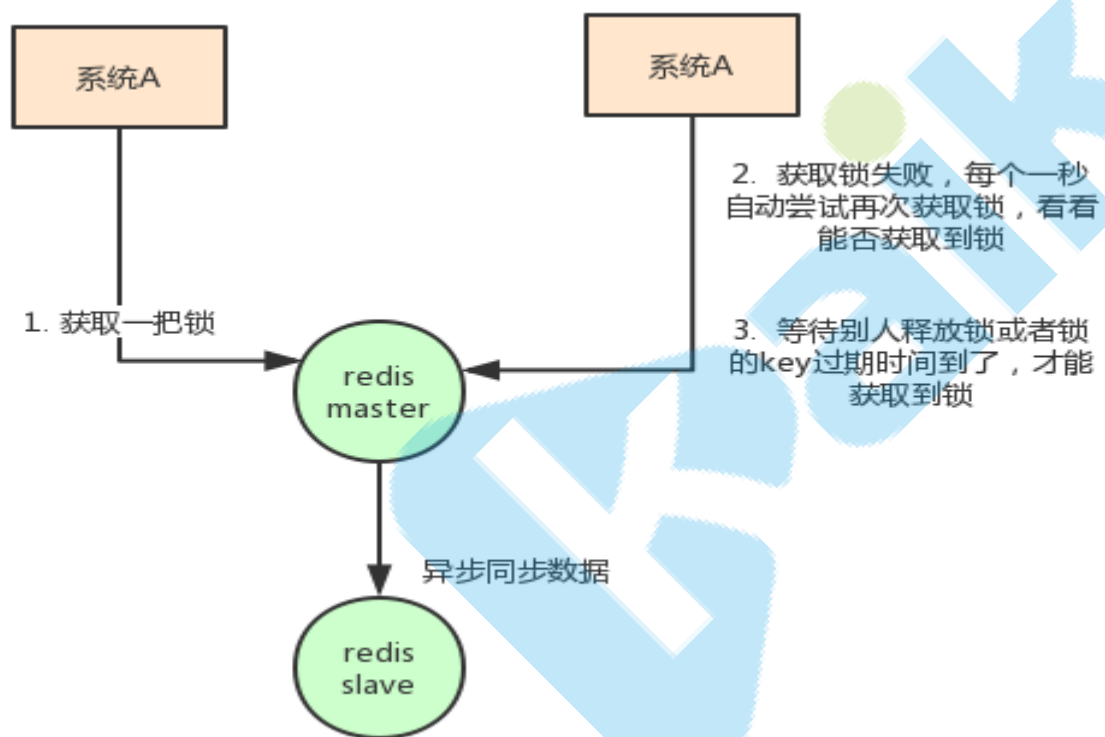
当线程A执行完任务，会显式关掉守护线程。

节点1的JVM进程



Redis方式实现分布式锁-要点回顾

- 一定要用SET key value NX PX milliseconds 命令
- value要具有唯一性
- 释放锁一定要使用lua脚本



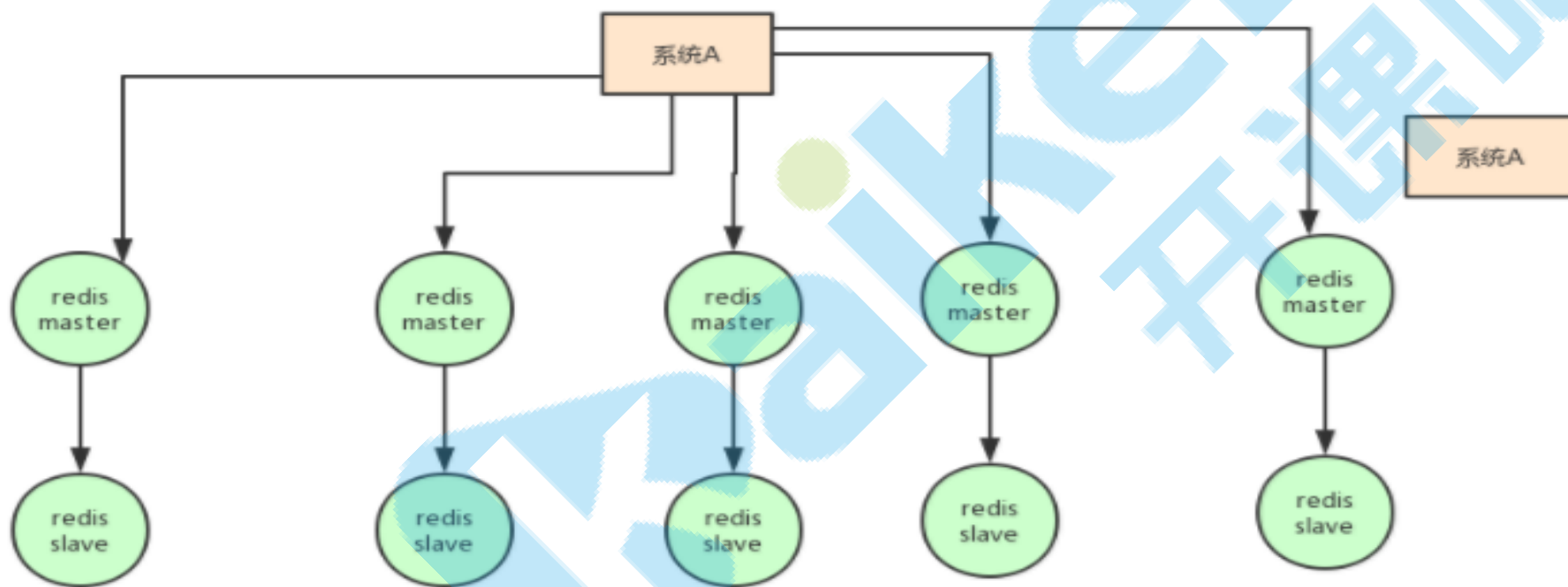
Redis分布式锁的可靠性思考

redis有3种部署方式:

- 单机模式
- master-slave + sentinel选举模式
- redis cluster模式

Redis分布式锁的可靠性思考-RedLock

分布式缓存锁—Redlock

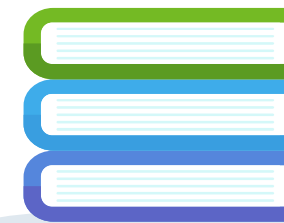


系统A同时向5个master设置一个key，如果超过3个设置成功，则认为加锁成功



/05

Zookeeper实现分布式锁方案

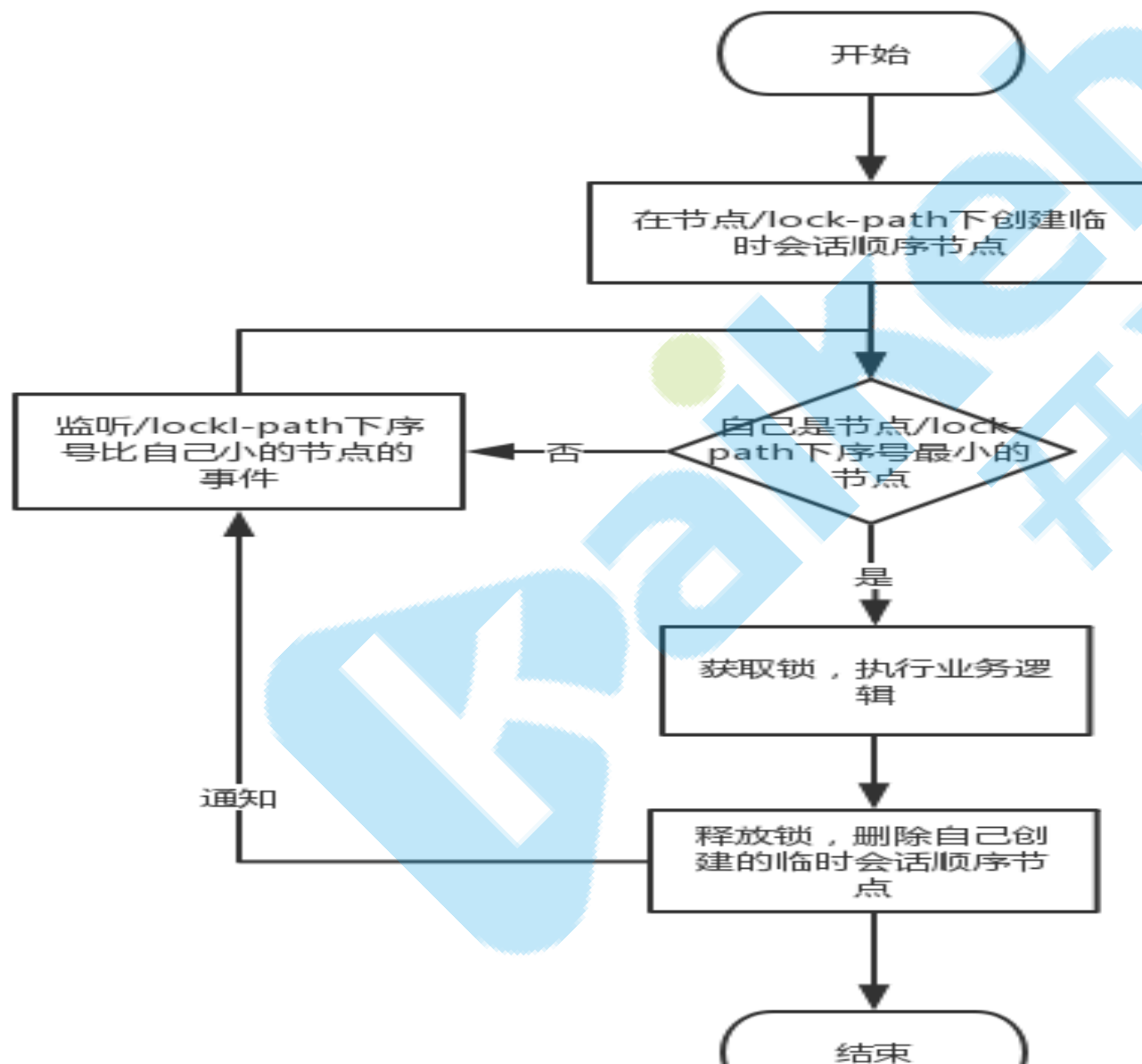


Zookeeper实现分布式锁

Zookeeper节点类型:

- 持久节点 (PERSISTENT)
- 持久节点顺序节点 (PERSISTENT_SEQUENTIAL)
- 临时节点 (EPHEMERAL)
- 临时顺序节点 (EPHEMERAL_SEQUENTIAL)

Zookeeper实现分布式锁逻辑



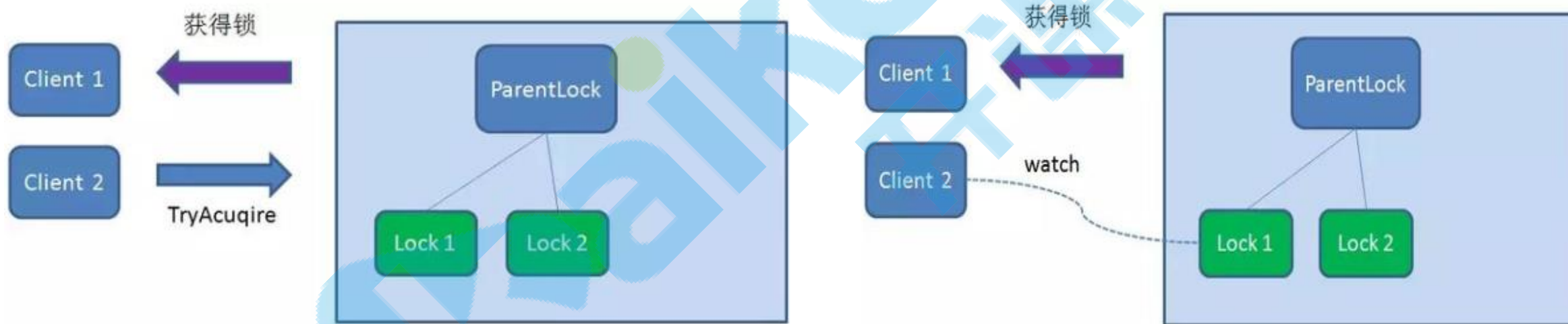
Zookeeper实现分布式锁的实现流程

client1获取锁:



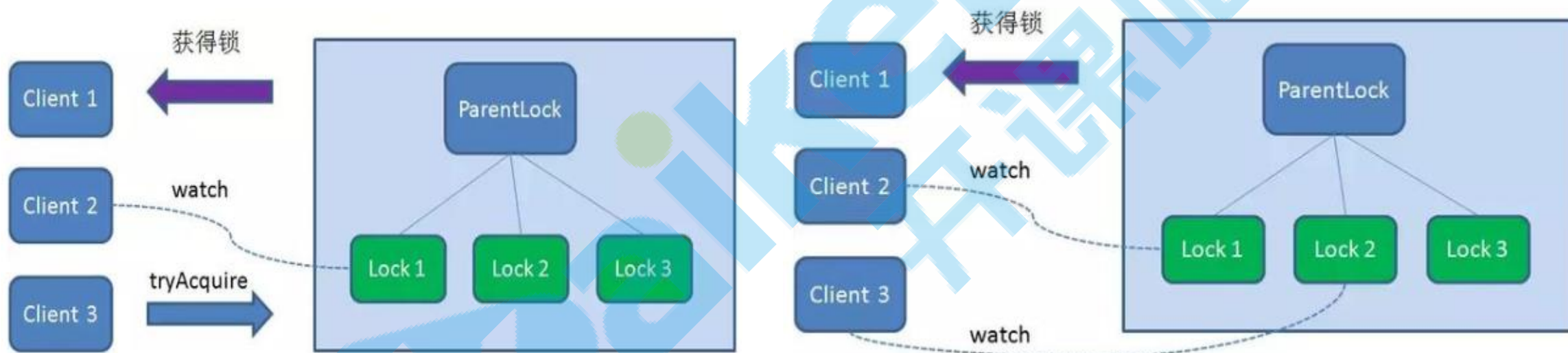
Zookeeper实现分布式锁的实现流程

client2获取锁:



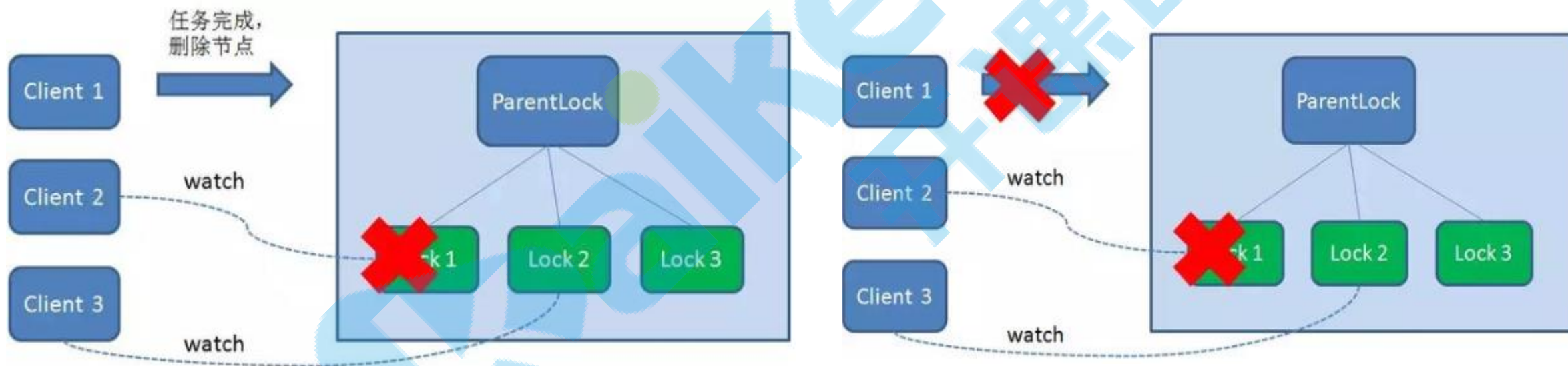
Zookeeper实现分布式锁的实现流程

client3获取锁:



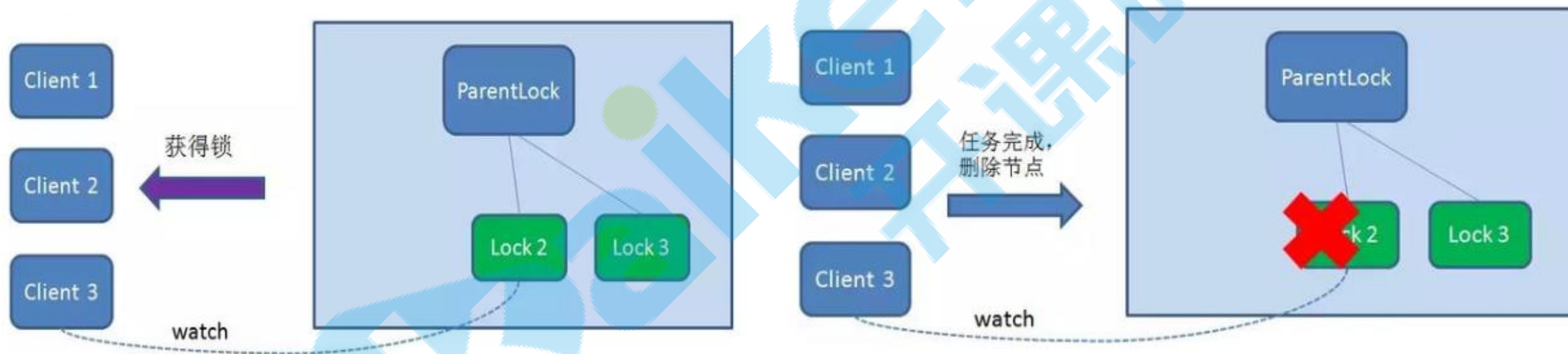
Zookeeper实现分布式锁的实现流程

client1释放锁:



Zookeeper实现分布式锁的实现流程

client2获取锁及释放锁:



Zookeeper实现分布式锁的实现流程

- 性能上可能并没有缓存服务那么高，因为每次在创建锁和释放锁的过程中，都要动态创建、销毁临时节点来实现锁功能
- ZK 中创建和删除节点只能通过 Leader 服务器来执行，然后将数据同步到所有的 Follower 机器上
- 取舍

Zookeeper分布式锁可靠性思考?

Caikēba
开课吧

三种分布式锁方案小结

上面几种方式，哪种方式都无法做到完美。就像CAP一样，在复杂性、可靠性、性能等方面无法同时满足。所以，根据不同的应用场景选择最适合自己的才是王道。

从理解的难易程度角度（从低到高）

数据库 > 缓存 > Zookeeper

从实现的复杂性角度（从低到高）

Zookeeper >= 缓存 > 数据库

从性能角度（从高到低）

缓存 > Zookeeper >= 数据库

从可靠性角度（从高到低）

Zookeeper > 缓存 > 数据库



/06

接口幂等性设计实践

