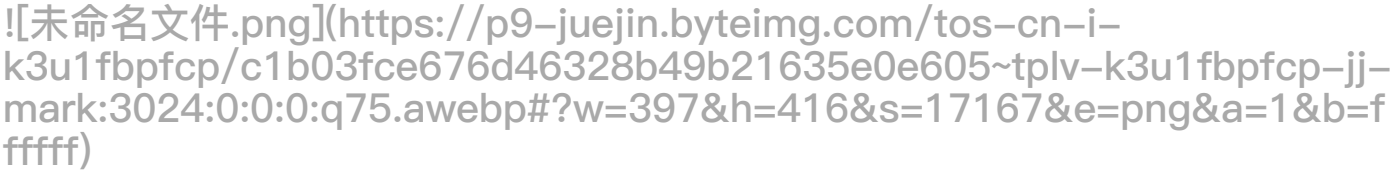
> 在ZooKeeper中，每个事务请求都会被赋予一个全局唯一的事务id，也就是zxid（ZooKeeper Transaction Id）。当主节点向从节点同步数据时，这些数据变更会作为一个事务被赋予一个zxid。
> 因此，如果一个从节点成功地从主节点那里接收并应用了这些数据变更，那么它的zxid就会更新。这就意味着，同步过数据的节点和未同步过数据的节点的zxid是不一样的。而在选举新的主节点时，ZooKeeper会选择zxid最大的节点，也就是数据最新的节点作为新的主节点。因为zxid最大的节点最有可能是已经同步过最新数据的节点。

2.Redlock

Redlock一种基于Redis实现的分布式锁算法，用来解决Redis分布式锁失效的问题，在使用多个独立Redis实例的情况下，能够提供更高的可靠性和安全性。



Redlock算法的核心思想是：使用多个独立的Redis实例作为锁服务器，当客户端获取锁和释放锁时，需要在所有的Redis实例上设置和释放。只有当超过半数以上的实例都设置或释放锁成功时，才认为操作成功，可以看得出和zookeeper的那套还是蛮像的。那么redLock怎么用呢？很简单，redisson为我们提供了封装，不需要我们动手实现，代码示例如下：

```
String lockKey = ”lock:product:001”;
//获取锁对象
RLock redissonLock1 = redisson.getLock(lockKey);
RLock redissonLock2 = redisson.getLock(lockKey);
RLock redissonLock3 = redisson.getLock(lockKey);
//将多个RLock对象关联为一个红锁
RedissonRedLock re ForValue().set(”stock”, String.valueOf(realStock));
            log.info(”扣减成功，剩余库存：” + realStock);
        } else {
```

```
            log.info("扣减失败，库存不足");
        }
    } finally {
        //释放锁
        redissonLock.unlock();
    }
    return "OK";
}
```

当然，实际业务中肯定逻辑会很复杂，所以我们需要做的就是梳理出其中不需要加锁的逻辑，把它们提出来减少锁的粒度。

2.分段锁

我们还以下单减库存这个场景为例，假如我们有100件库存，我们在库存阶段把每十件商品作为一个库存单位来设置key，当然这就是十把锁了，大家想想原本获取库存这一步从一把锁变成了十把锁，那么同一时刻处理的请求数就由一个变为十个了，性能瞬间提升十倍。分段锁的思想就是拆分业务逻辑，根据拆分情况加不同的锁从而提升并发和性能，当然提升性能的代价就是需要在找库存这步多写点逻辑进行处理了。

从场景谈分布式锁的应用
==========

1.下单重复提交

这个场景很常见，前后端都有相应的方案。首先，前端肯定要加“防抖”，后端则需要做幂等， 幂等的方案也很多，我们这里就谈谈分布式锁在该场景怎么做。很简单，**我们只需要在下单的业务逻辑前加锁，但锁的key在这里有说法的，肯定不能用订单id，因为每次下单都会产生一条新的订单，所以我们这里用用户id作为key更为合适。**

2.支付与取消订单同时发生

很多电商网站的订单如果你在一定的时间内未支付的话，会自动取消，那如果你即将在取消订单前付款，在高并发场景下，就有可能发生支付了一个已经取

消的订单，用户付款了却永远也收不到货，那势必会带来问题。我们这个场景也是可以用分布式锁来解决的，这两个业务逻辑前都加上分布式锁，就用订单id作为key，这样两个操作一次只能执行一个。

原文链接: https://juejin.cn/post/7365741765729583154