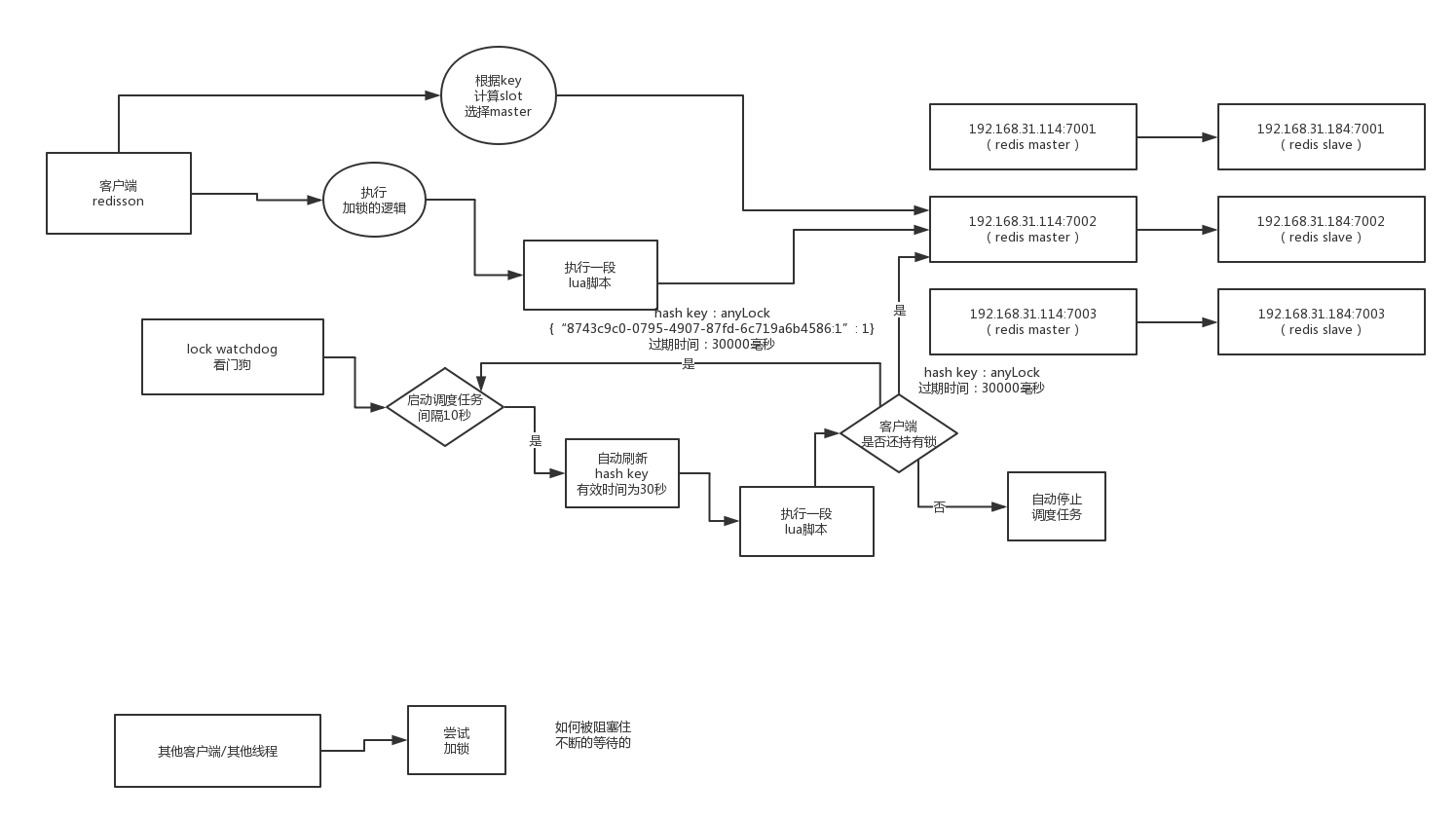
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/【高亮】附带软件下载地址/0001【高亮】软件地址.txt**

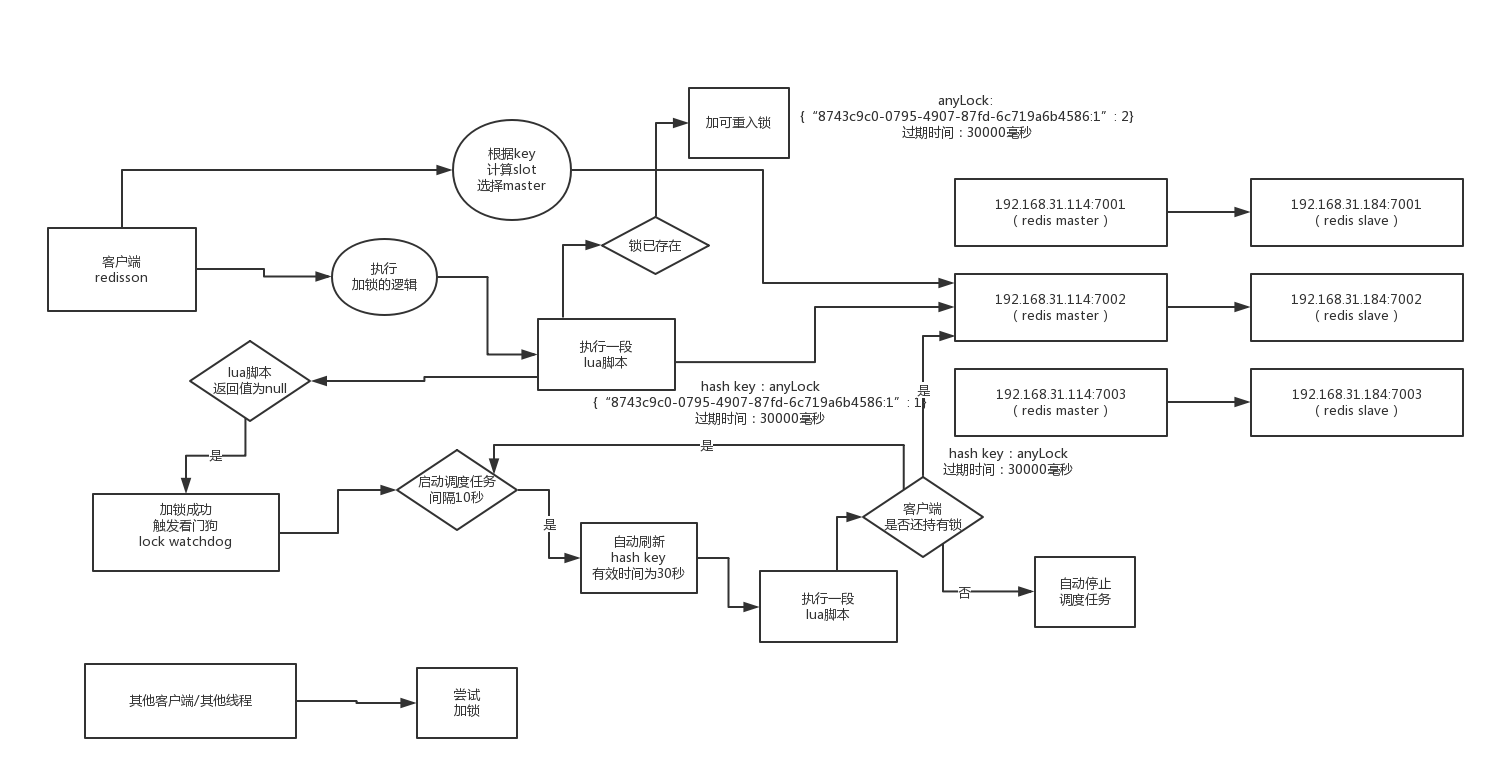
���ӣ�https://pan.baidu.com/s/1Jvyhfb-M5K4DaLWDiwTt9w

��ȡ�룺5yn9

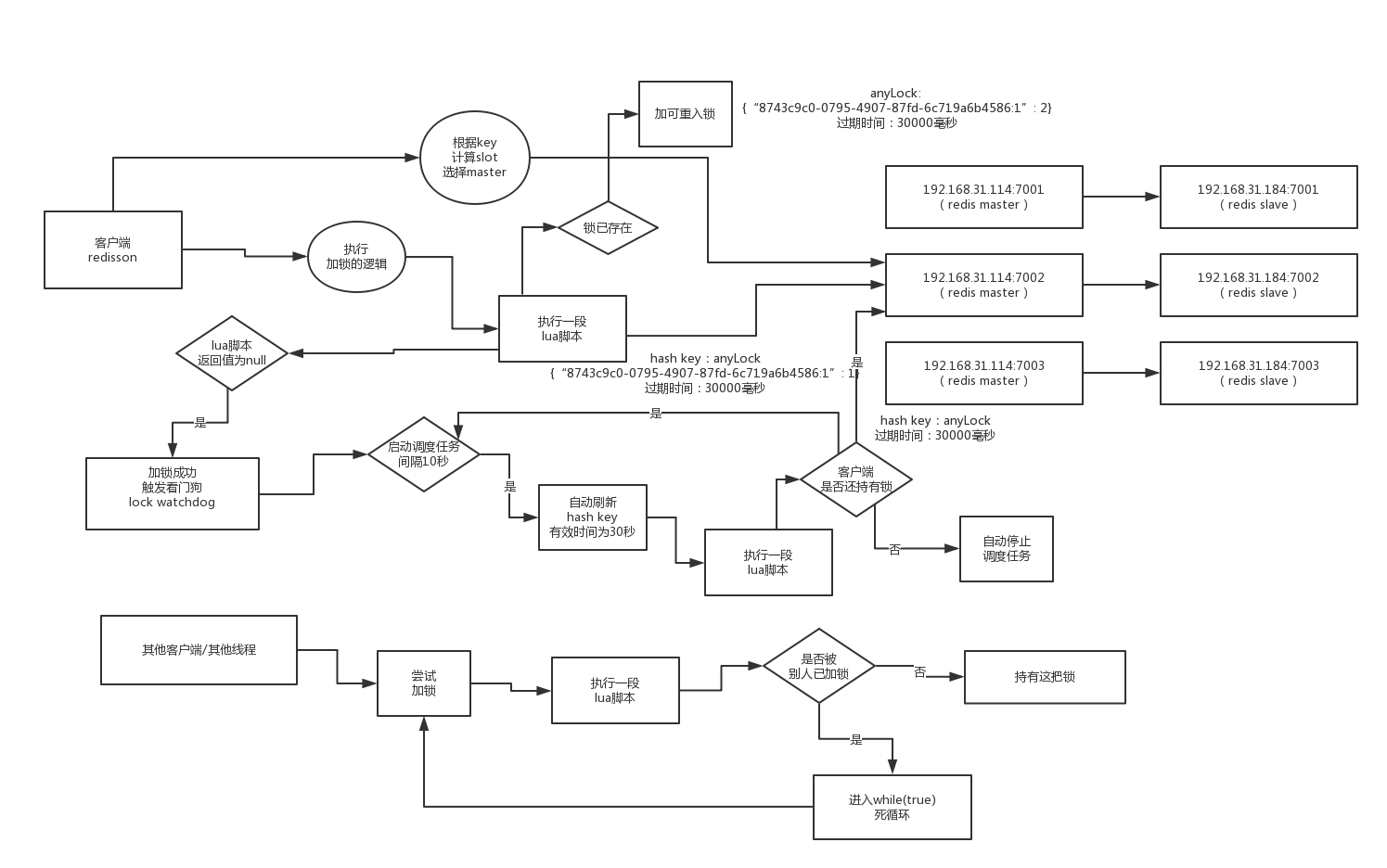
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/08\_redis分布式锁（三）：可重入锁源码剖析之watchdog维持加锁/01\_redisson ReentrantLock的原理(1).png**



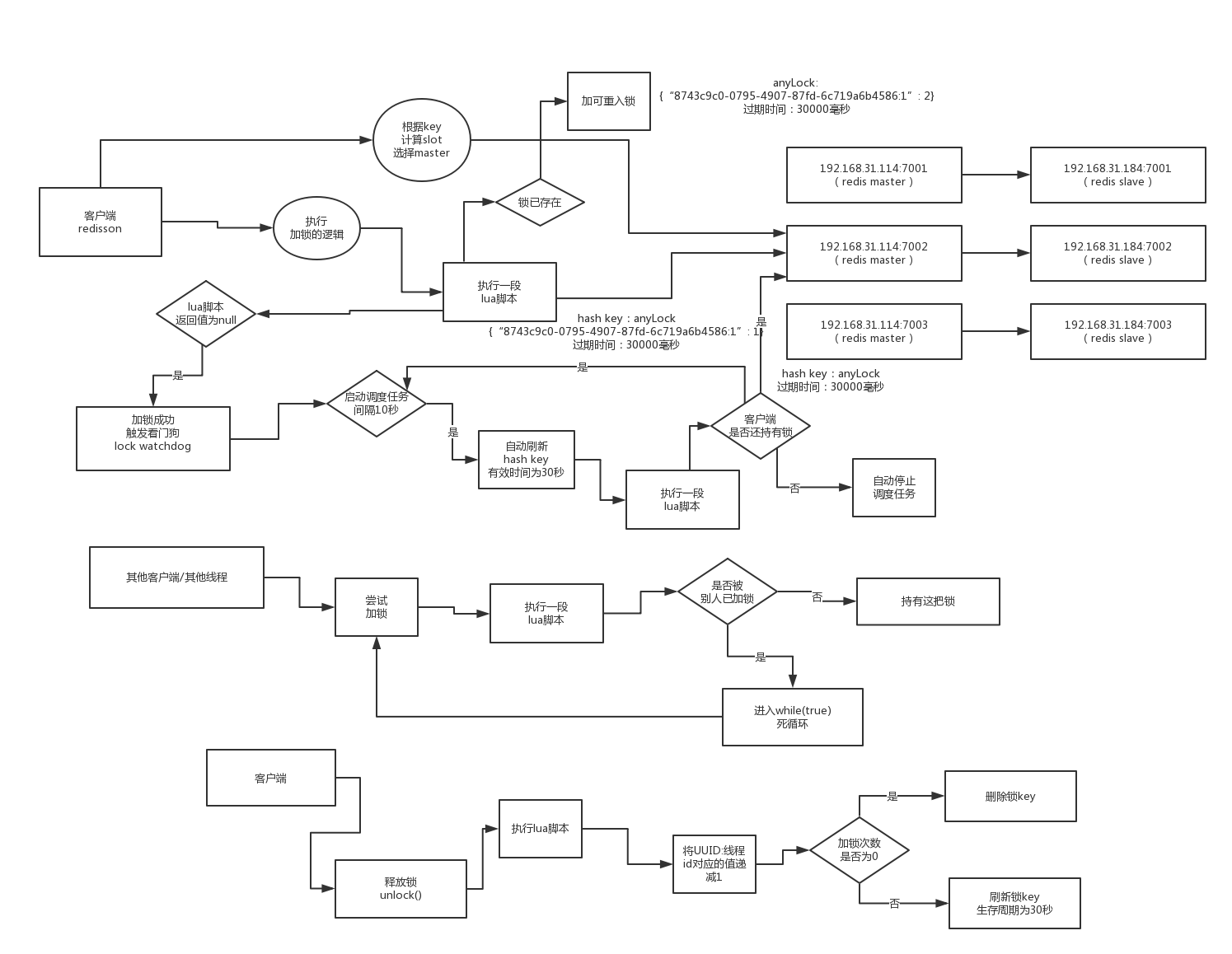
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/09\_redis分布式锁（四）：可重入锁源码剖析之可重入加锁/01\_redisson ReentrantLock的原理(2).png**



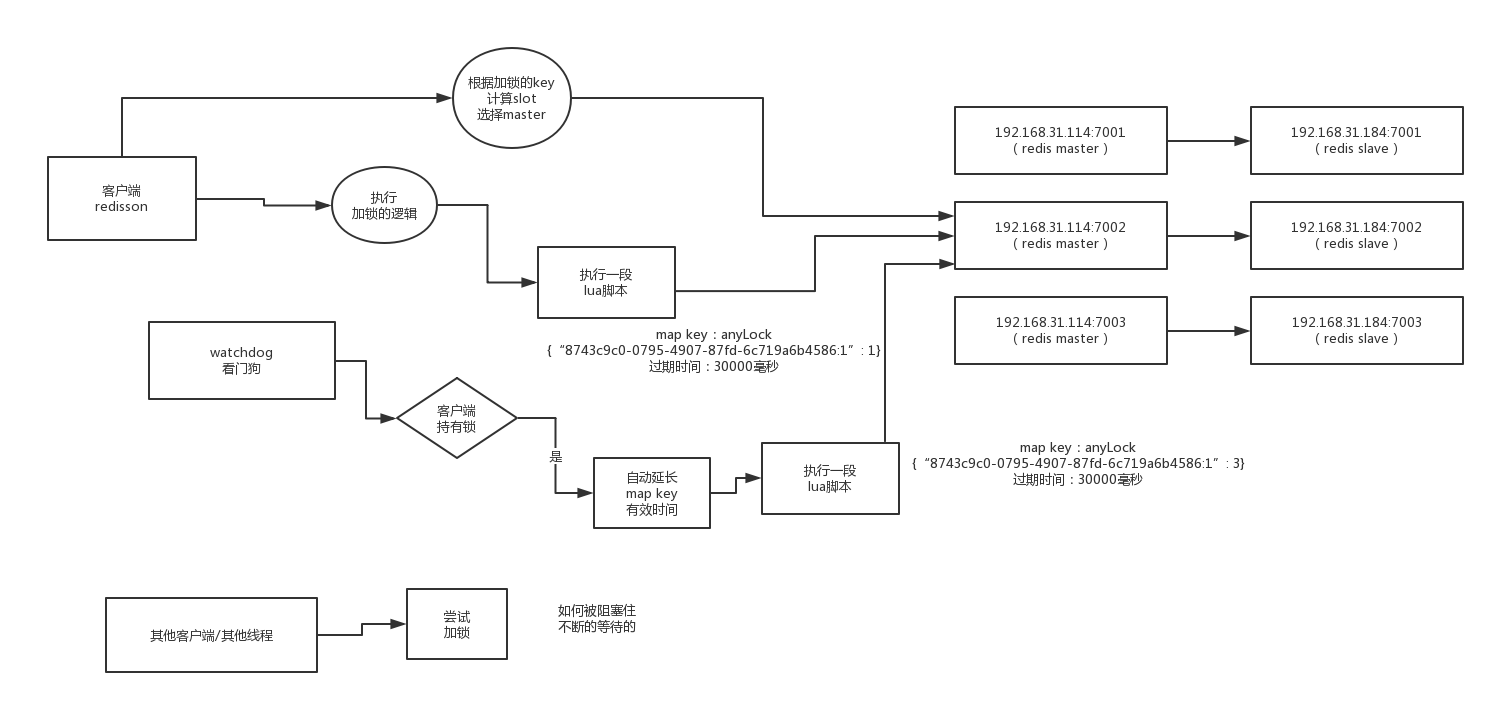
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/10\_redis分布式锁（五）：可重入锁源码剖析之锁的互斥阻塞/01\_redisson ReentrantLock的原理(3).png**



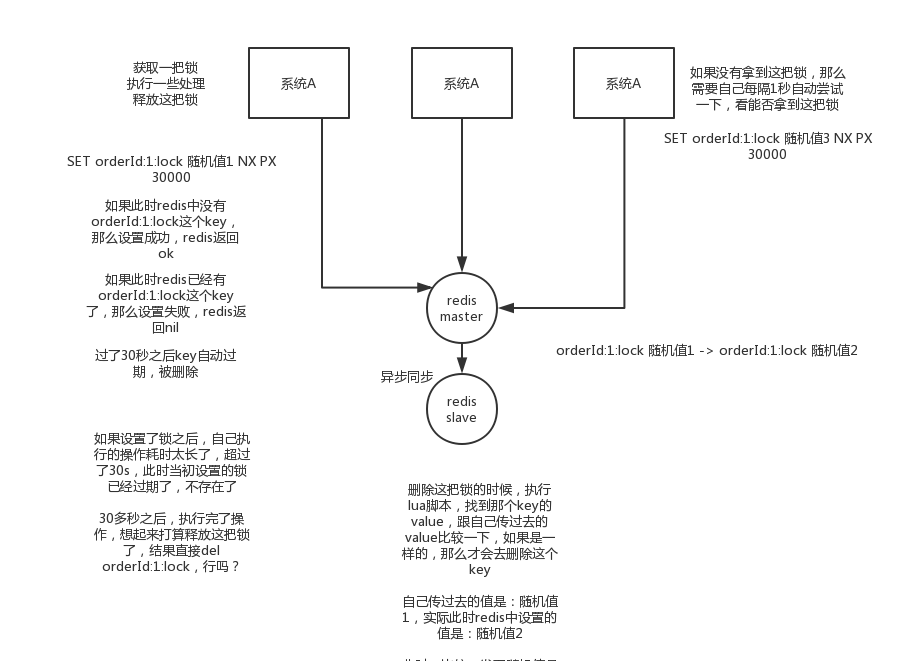
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/11\_redis分布式锁（六）：可重入锁源码剖析之释放锁/01\_redisson ReentrantLock的原理(4).png**



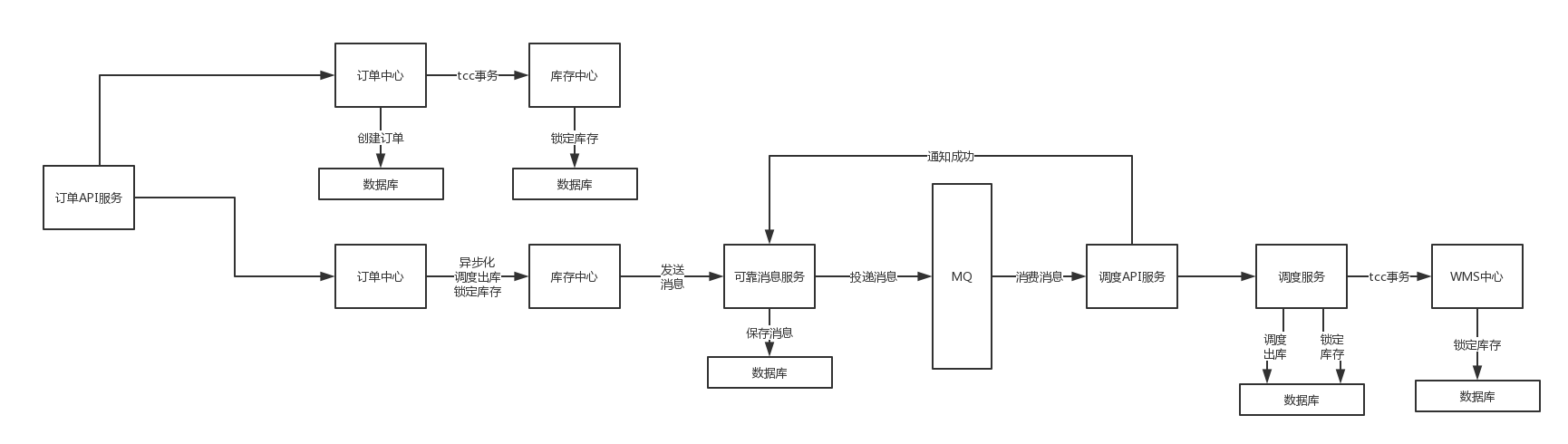
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/07\_redis分布式锁（二）：可重入锁源码剖析之lua脚本加锁逻辑/01\_redisson ReentrantLock的原理.png**



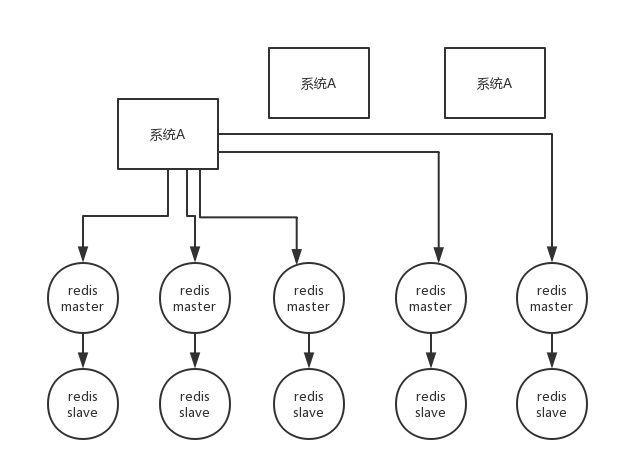
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/01\_基于redis和zookeeper的分布式锁实现原理/01\_redis分布式锁.png**



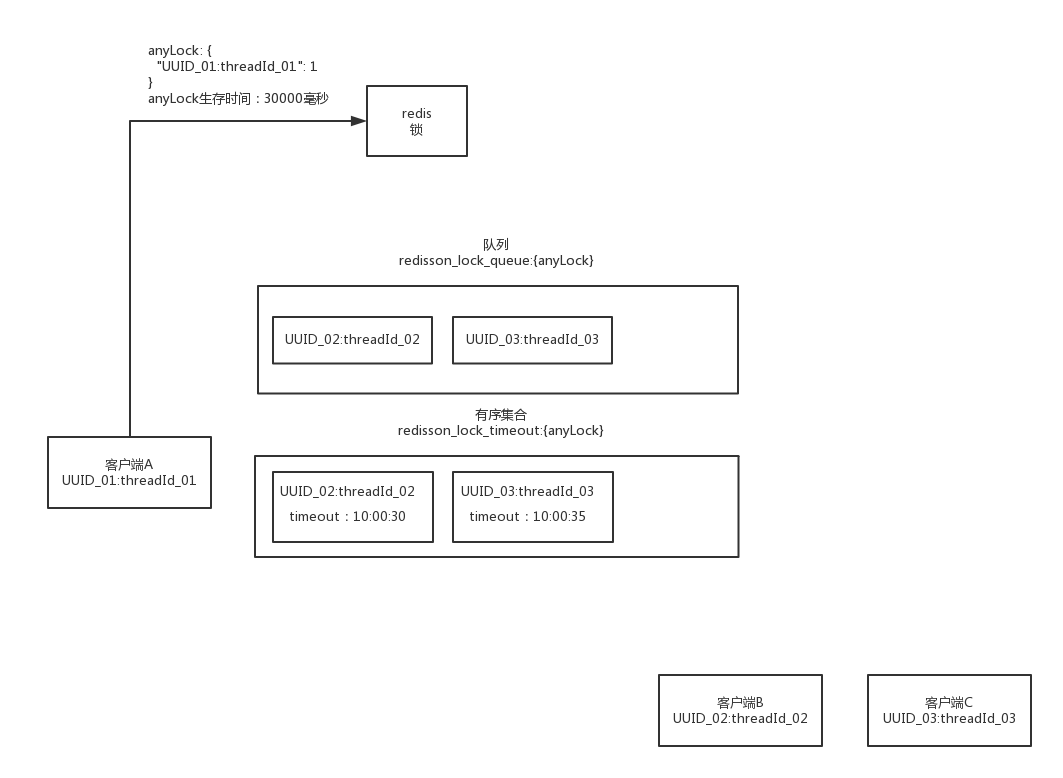
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~资料/03\_设计商品购物流程的分布式事务方案结合业务落地的各个细节/01\_创建订单子流程的分布式事务方案.png**



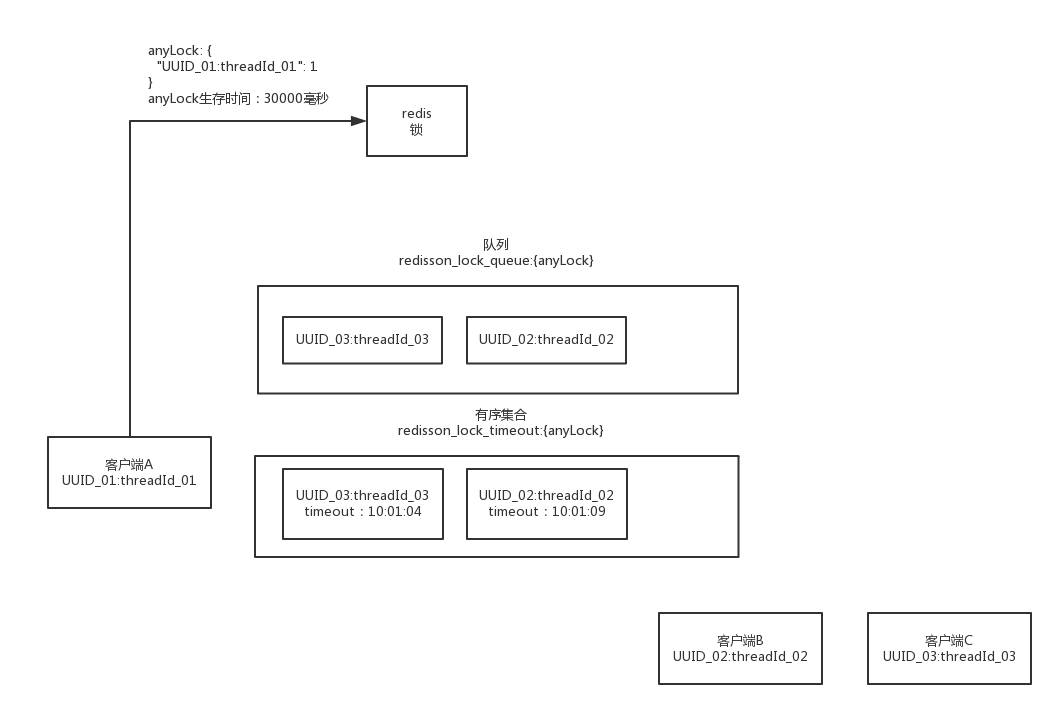
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/01\_基于redis和zookeeper的分布式锁实现原理/02\_RedLock算法.png**



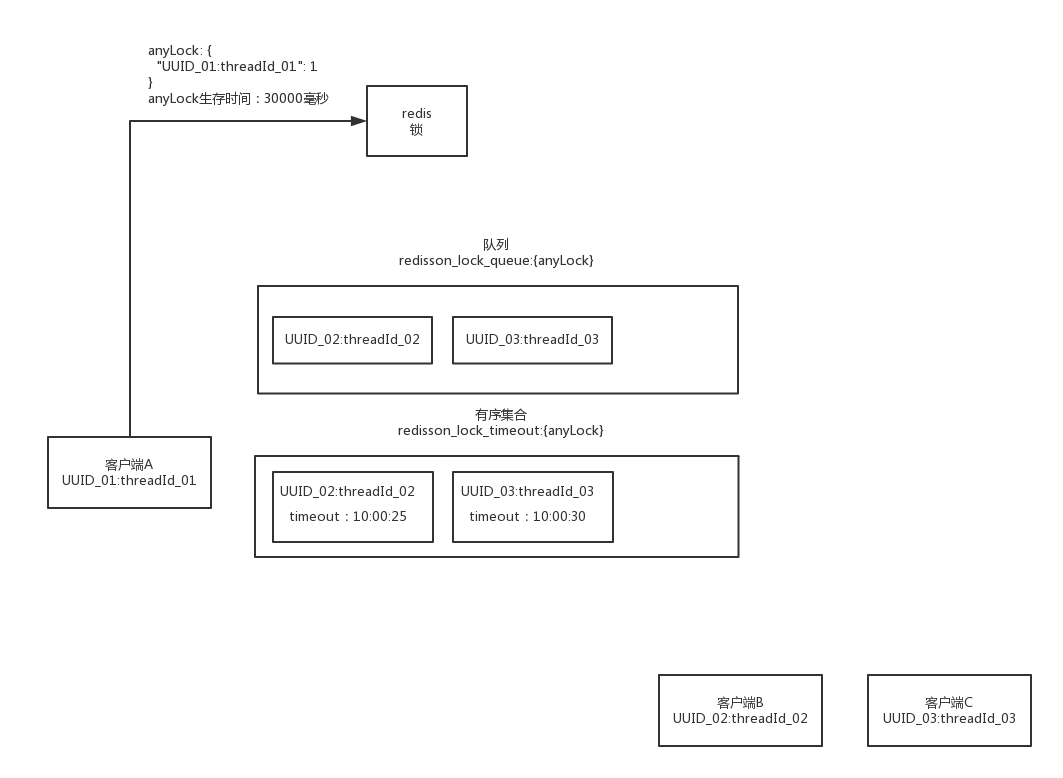
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/17\_redis分布式锁（十二）：公平锁源码剖析之排队分数刷新/02\_公平锁原理(1).png**



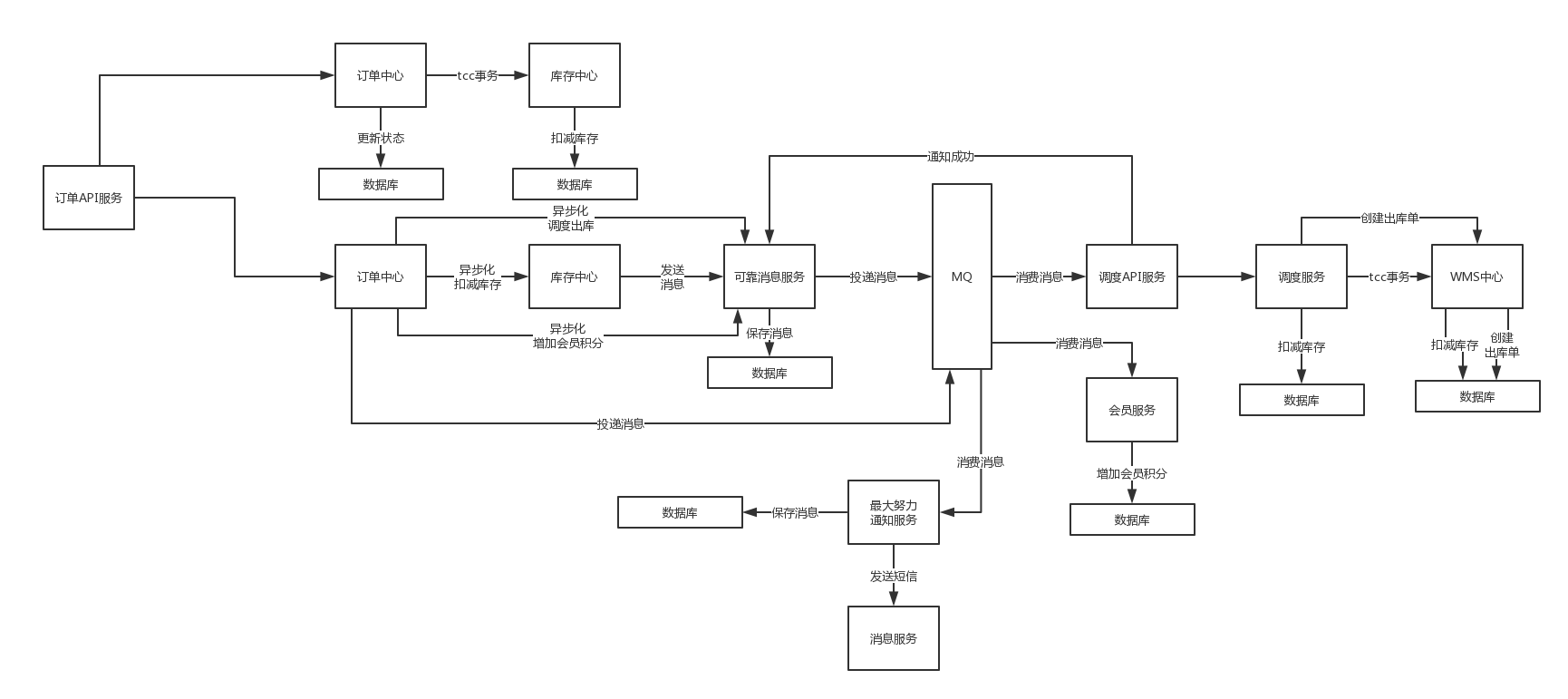
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/18\_redis分布式锁（十三）：公平锁源码剖析之队列重排/02\_公平锁原理(2).png**



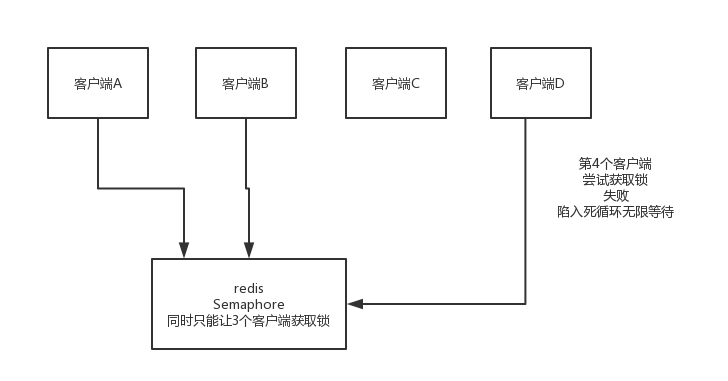
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/15\_redis分布式锁（十）：公平锁源码剖析之排队加锁/02\_公平锁原理.png**



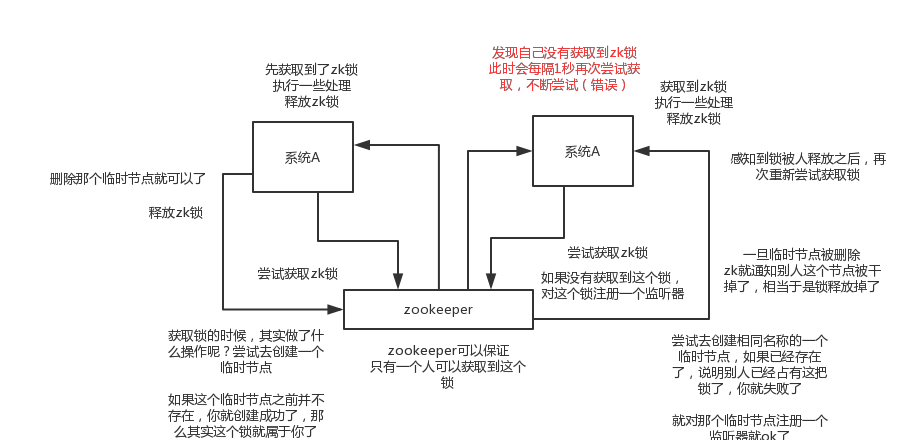
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~资料/03\_设计商品购物流程的分布式事务方案结合业务落地的各个细节/02\_支付订单子流程的分布式事务方案.png**



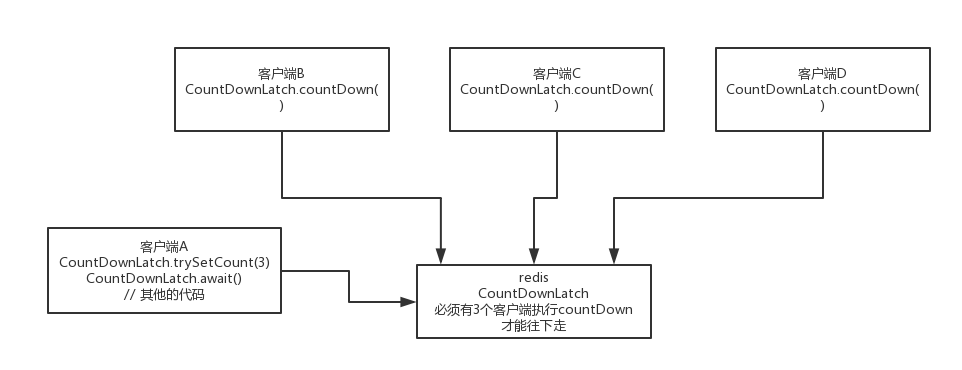
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/33\_redis分布式锁（二十八）：Semaphore原理图解/03\_semaphore原理.png**



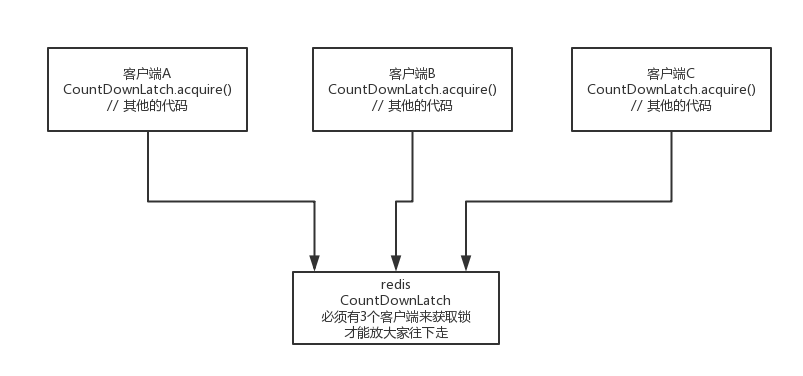
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/01\_基于redis和zookeeper的分布式锁实现原理/03\_zookeeper分布式锁.png**



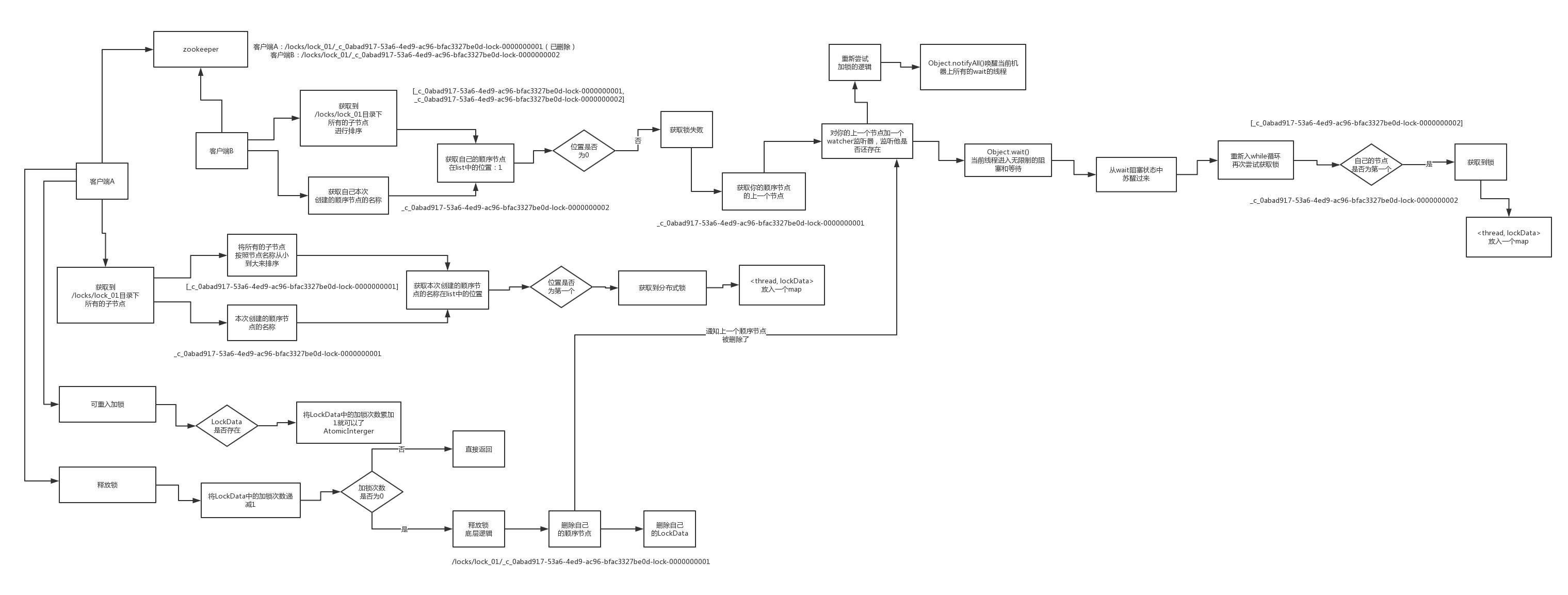
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/37\_redis分布式锁（三十二）：CountDownLatch使用演示/04\_CountDownLatch原理(1).png**



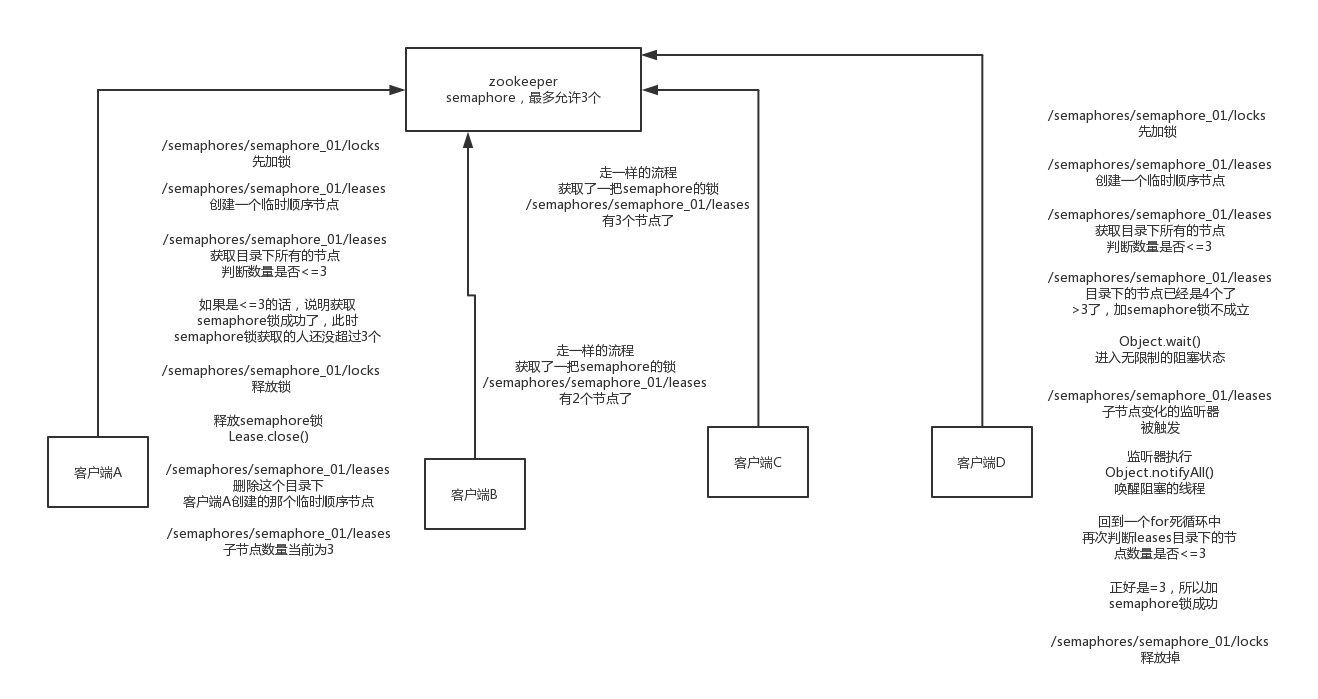
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/36\_redis分布式锁（三十一）：CountDownLatch原理图解/04\_CountDownLatch原理.png**



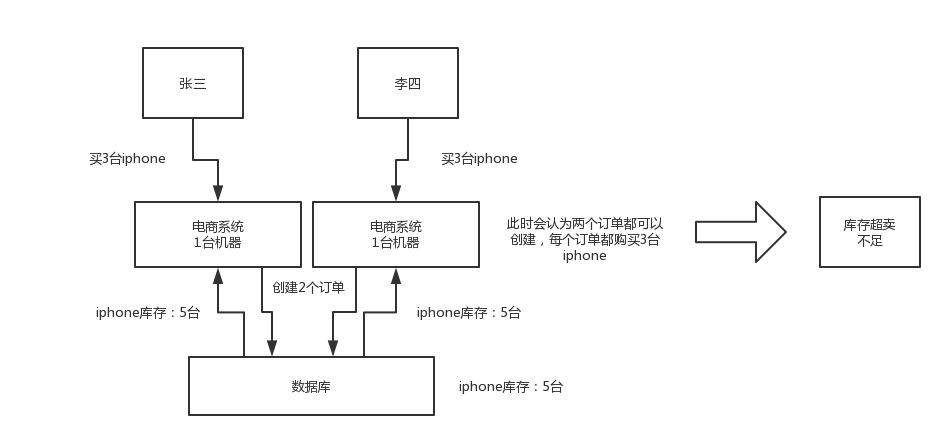
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/42\_zookeeper分布式锁（一）：可重入锁源码剖析/05\_zookeeper可重入锁的原理.png**



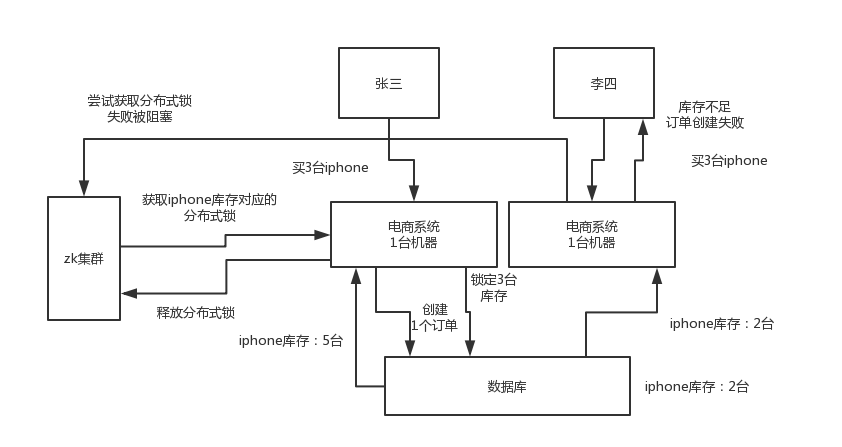
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/43\_zookeeper分布式锁（二）：Semaphore源码剖析/06\_zookeeper semaphore实现原理.png**



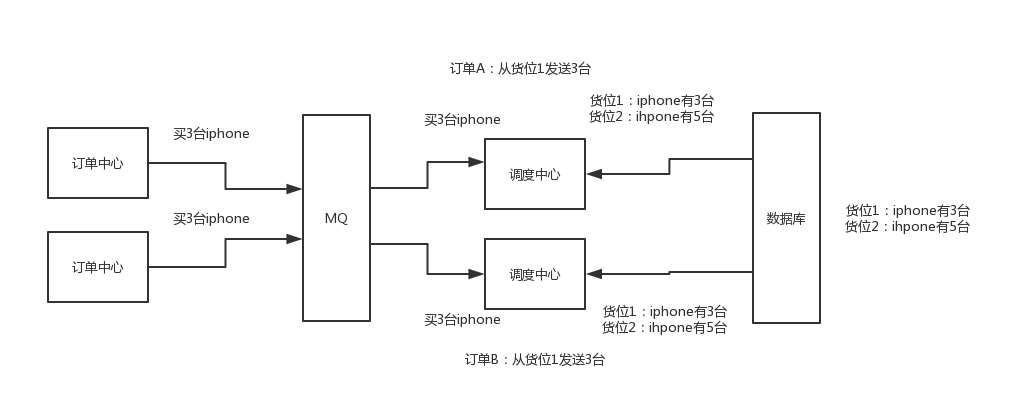
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/47~51资料/47\_电商系统中的库存超卖场景以及分布式锁解决方案/07\_库存超卖问题.png**



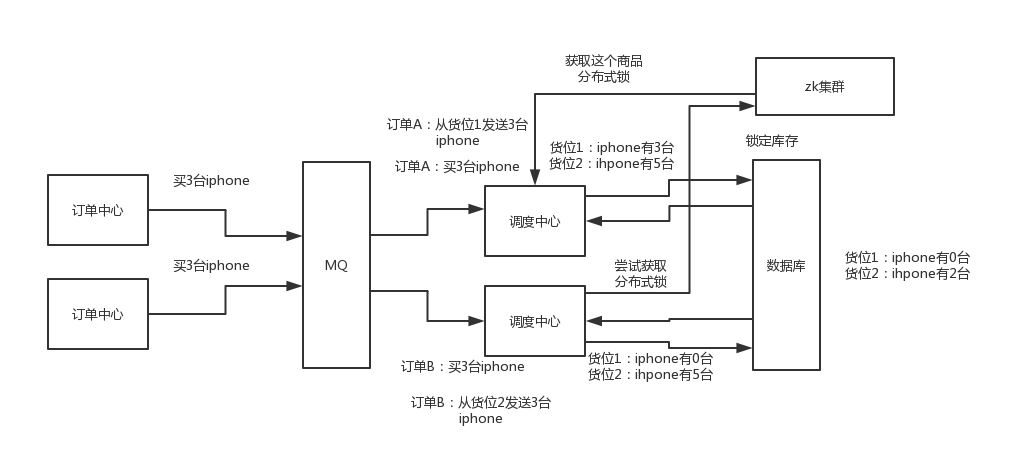
# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/47~51资料/47\_电商系统中的库存超卖场景以及分布式锁解决方案/08\_库存超卖的分布式锁解决方案.png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/47~51资料/48\_电商系统中的调度销售出库场景以及分布式锁解决方案/09\_调度销售出库问题.png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/47~51资料/48\_电商系统中的调度销售出库场景以及分布式锁解决方案/10\_调度销售出库问题的分布式锁解决方案.png**



# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/08\_电商系统的退货流程的接口幂等性方案以及开发（一）/笔记.doc**

退货，订单里面点击一个按钮，申请退货，要经过客服中心的审核，退货工单

就跟采购单，就算是万分之一的概率发生了，不小心创建了重复的退货工单，也不要紧的，客服会审核你的退货申请，发现重复的退货工单，可以删除掉一个

把退货寄回来，填写一个物流单号，会从订单中心同步到客服中心去

收到退货之后，会转到仓库里去，然后会点击确认收到退货

调度中心纯内存的操作，哪怕是被重复调用几次也是不要紧的，然后就会找wms中心，创建退货入库单

仓库收到退货，仓库管理员，完成退货入库的上架，然后提交领导审核退货入库单

到这里为止，整个电商系统，三大核心流程，采购流程、购物流程、退货流程，都已经搞定了，我们都已经将里面的核心接口，通过状态机、唯一索引、去重表，实现了各个接口的一个幂等性

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/05\_电商系统的购物流程的接口幂等性方案以及开发（一）/笔记.doc**

在做分布式事务方案的时候，基本上就已经从代码层面，梳理过购物流程的代码，创建订单子流程，支付订单子流程

我呢之前给大家留了一个作业，当时我带着大家把创建订单子流程给改造为了分布式事务的机制，支付订单子流程，让大家自己根据我给的思路改造成分布式事务，我自己把这个代码给写好了

我这一讲先给大家简单提一下，当时我写的那个支付订单子流程的分布式事务

（1）更新订单状态、库存中心扣减库存、会员中心增加积分，绑定为了一个tcc事务

（2）异步化通知调度中心和wms中心扣减库存，他们俩绑定了一个tcc事务，通过可靠消息最终一致性方案去通知了他们

（3）异步化通知调度中心进行调度销售出库，通过可靠消息最终一致性方案通知了他们

（4）异步化通知消息服务发送短信给用户说商品准备出库，最大努力通知方案

创建订单子流程、支付订单子流程，接口的幂等性如何来保障

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/11\_电商系统的退货流程的接口幂等性的测试/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/06\_电商系统的购物流程的接口幂等性方案以及开发（二）/笔记.doc**

创建订单子流程

创建订单的这个接口，是否要保证幂等性，订单不要重复被创建，这个事儿其实是比较关键的一个事情，因为这个接口，可能是被网站的前端代码调用的，也可能是直接给手机APP上的发送的请求来调用的

如果说前端代码、手机APP的代码，因为网络超时，他发起了重试，导致订单可能会被重复创建几次，这个事情其实是无法容忍的

这个事情的方案，不是纯后端可以实现的，要网站前端或者是手机APP要配合一下的

对每个订单发起的这个请求，都必须附加另外一个参数，必须是在网站前端或者是手机APP，对用户的每一次创建订单的请求，都加一个类似于UUID的这么一个唯一的请求ID，一个请求ID唯一标识一个请求

如果说网站前端或者是手机APP因为网络超时，对同一个创建订单的请求进行了重试的话，那么没关系，那样的话呢，就是可以在后端接口里，根据那个唯一的请求ID进行去重操作，避免说重复创建订单

也可以是在后端提供另外一个基础服务，基于snowflake唯一ID的算法，对所有的端都提供全局唯一的ID，网站前端或者手机APP发起一个创建订单的请求之前，都找后端的基础服务先获取一个全局唯一的ID，作为请求ID

自己瞬时生成一个唯一的ID，用户名\_时间戳\_随机值，来生成当前这个用户这个瞬间下一个订单的唯一的请求ID

我跟大家说一下对于这个订单不重复的保证，我们之前用过的其中一个方案是，将那个requestId作为一个字段保存在数据库里，对requestId建立一个唯一索引，这个方案非常不错的

手机APP，还有几个不同的终端，localId，就是在终端设备上生成的一个唯一ID，网络不可靠的，所以每天在几十亿次，几百亿次的请求量下，肯定会出现一些重试的，当时就是因为我们在表里对localId建立了一个唯一索引，所以订单一定不会出现重复的

订单中心 -> 库存中心，去锁定库存，这边是服务之间的调用，很有可能因为网络问题，也有可能会导致订单中心重复调用几次库存中心的接口，导致库存被多扣减了

库存服务消费到了消息，调用调度服务+wms服务组成的tcc事务，可能会重复调用

就已经将创建订单子流程的接口都保证了幂等性了

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/10\_电商系统的购物流程的接口幂等性的测试/笔记.doc**

{

"userAccountId": 1,

"username": "zhangsan",

"consignee": "张三",

"deliveryAddress": "测试地址",

"consigneeCellPhoneNumber": "测试号码",

"freight": 4.6,

"payType": 2,

"totalAmount": 100,

"discountAmount": 0,

"couponAmount": 0,

"payableAmount": 100,

"invoiceTitle": "测试发票抬头",

"taxpayerId": "测试纳税人识别号",

"orderComment": "测试订单",

"orderItems": [

{

"goodsId": 1,

"goodsSkuId": 1,

"goodsSkuCode": "TEST\_SKU\_01",

"goodsName": "测试商品01",

"saleProperties": "测试销售属性",

"goodsGrossWeight": 50,

"purchaseQuantity": 3,

"purchasePrice": 700,

"goodsLength": 10,

"goodsWidth": 20,

"goodsHeight": 10

}

]

}

我们长期规划里面，是要去做dubbo rpc框架的封装，会把分布式系统服务框架，给封装的非常的漂亮，服务上下线的感知，必须到秒级别，问题太多了

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/02\_保证分布式系统的接口幂等性的几种常见方案/笔记.doc**

接口幂等性的话，一般的常见方案

1、业务表内唯一索引

好比说电商系统里面，不是说有那个销售出库单表，wms中心

如果你要对创建销售出库单的接口保证幂等性，也就是说人家网络超时，重复调用的时候，保证一个订单只能有一个对应的销售出库单

销售出库单表里面，其实都是有这个unique key唯一索引的，比如说可以针对销售出库单的表的订单id，创建一个唯一索引，你如果接口被重试，同一个订单创建一个销售出库单的话，一定会违反唯一索引，那么此时会报错

保证说你在数据库里，一个订单只能有一个销售出单

常见于插入操作，创建一些关键的数据的时候，而且这个数据在库里只能有一条的时候

2、业务表内状态机

修改订单状态，比如说将订单状态修改为待发货的时候

update order set status = “待发货” where status = “待付款” and id = 1，订单的状态其实就变为了“待发货”。假如说id = 1的订单接口重复调用，又要执行一次这个操作的话，就不会生效了，就不会再次修改了

在订单模块里，我们不是用了状态模式，在进行状态流转的时候，其实都会去判断一下的，当前是否处于某个状态，然后才能流转到下一个状态。

3、基于版本号的更新

id name age version

1 张三 20 1

如果要调用人家的这个接口，更新他的这个年龄，先得查一下他的版本号是多少

version = 1

调用人家的接口修改他的年龄，要changeAge(1, 21, 1)

在你的接口里为了保证分布式接口的幂等性

update user set age=21, version=version+1 where id=1 and version=1

如果这条SQL执行成功过后

id name age version

1 张三 21 2

如果changeAge(1, 21, 1)，被重复调用了，此时会如何？

update user set age=21, version=version+1 where id=1 and version=1

我们一般不常用，对于接口调用方来说，要多做一些事情，他要先查出来数举的version，调用修改接口的时候，传过去这个version

4、基于mysql的去重表 / 基于redis的去重

就是说这个方案是很常见的一个方案

比如说你的接口的入参，参数，是changeAge(1, 21, 1)

将所有的参数拼接成一个字符串，或者是从这些入参里选择一些参数，可以唯一标识这一次请求的一些参数id和version，id和version每次请求都不一样的，应该是可以唯一的标识这一次请求

1\_21\_1，这样的一个字符串

如果基于mysql，单独搞一个表出来，就一个字段，建一个唯一索引，插入这个1\_21\_1到表里去。如果这个接口被重复调用的话，1\_21\_1，再次插入一个表的话，唯一索引会报一个冲突出来，这次插入就会失败

这个方案还是不错的，尤其是并发不是特别高的话，接口被调用的并发不是特别高的话，每秒的并发请求量在1000左右，1000以内的话，用mysql的去重表也没什么问题

但是如果接口调用量很大，并发很高，一秒请求量达到了几十万，选择使用redis，拼接一个串出来，直接set设置到redis里去，如果下一次人家请求再过来了，此时会发现这个key已经存在了，那么这个时候就不能执行了，因为已经出现重复调用了

在一个分布式系统里，尤其是一个微服务的系统，实现的接口保证幂等性应该是一个工程师最基础的素养

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/09\_电商系统的采购流程的接口幂等性的测试/笔记.doc**

幂等性方案是否生效的，我们的spring cloud服务刚启动的时候访问，很容易出现重试，如果重试就可以验证，幂等方案是否可以保证数据的准确性，做分布式系统架构的设计的时候，spring cloud里面很多问题，适合一些中小型公司，我觉得还ok

大型互联网公司，我觉得spring cloud里面问题还是挺多的

也都是用的是rpc服务框架，基于dubbo自己封装的，dubbo也是我们的一个重点

完成上架，然后编辑一些上架的信息

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/07\_电商系统的购物流程的接口幂等性方案以及开发（三）/笔记.doc**

支付订单子流程的幂等性

我们基本上就已经将购物流程的核心接口都保证了幂等性了，不要重复创建订单，不要胡乱扣减库存，保证销售出库单的唯一性

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/03\_电商系统的采购流程的接口幂等性方案以及开发（一）/笔记.doc**

结合我们阶段一的电商业务流程来做实战，将各种技术以及方案落地，结合业务细节来开发、测试，让大家感受到，在复杂的业务系统里，各种技术，架构，方案，是如何落地的，这样对你的技术学习才是最有好处的

采购流程

大家跟着我学技术，技术项目实战的，时候，你才能感觉到，就跟你在公司里做一个项目，手头你有一个大的系统，有自己的复杂的业务，如何落地技术，分析的过程，结合业务的过程

（1）创建采购单

创建采购单开始的，定好了我要从哪个供应商那里，采购哪些商品，每个商品的进货价是多少，每个商品要采购多少件。实际上是要由采购部门的领导来审批，只有领导审批通过了，这个采购单才会发出去给供应商，而且会找调度服务，去进行调度采购入库的事情

创建采购单的接口，是否有必要实现幂等性呢？

我的一个结论，是不需要，创建完了采购单之后，领导来审核，人为介入审核的过程，假如说真的是这个接口不幸的被重复调用了，创建出来了多个采购单，重复的，领导审核的时候会发现这个问题

大不了偶发性的出现这个问题的时候，就让领导给把采购单给删除，或者是审批不通过就可以了

创建采购单是人为的通过浏览器的页面上的表单来提交的，这个出现重试的概率还是比较低的

采购员在创建，如果真要是重复了，采购员跟他领导说一声就可以了

（2）审核采购单

更新采购单的状态：这个操作有没有必要实现幂等性的保证？没必要的，就算是接口重复调用了，update purchase\_order set status = ??? where id=1，都是没关系的，不涉及到幂等性的问题

（3）创建采购入库单

调度服务里面，没有涉及到数据库的操作的，直接都是纯内存的操作，所以这块也是没有幂等性保障的必要的

wms中心里面，涉及到了采购入库的第一个核心环节，创建采购入库单，这里就有一个问题了，如果wms中心的接口被重复调用了，会怎么样呢？就会很尴尬的创建多个重复的采购入库单

涉及到spring cloud的服务之间的重试调用，服务之间的调用，其实可能确实还是会遇到一些重复性的一些问题的，可能服务间的网络超时了，重复调用，重复采购入库单，重复了，那就不是那么回事了

这个对应的工作人员是仓库里的库管，他看到重复的采购入库单就懵逼了，他可能以为就是有好几个采购入库单，要采购好几批商品入库，重复是系统导致的，库管是分辨不出来的，所以说这个事情是无法容忍的

对于wms中心创建采购入库单的这个接口，是必须要保证说是ok的，不能有重复的情况发生，要保证这个接口的幂等性，一个采购单一定是只能对应着一个采购入库单的，是不是可以在采购入库单表的采购单id的字段上建一个唯一索引啊？

业务语义，采购入库单id + 商品sku id，联合唯一索引，一个采购入库单中，对一个商品sku的采购条目，从业务上来说，必须保证只能有一条

iphone 7这个sku采购120件

通过两个表的唯一索引，就可以保证说采购入库单的数据是一定不会出错了，哪怕是接口再怎么被重复调用也是不要紧的

（4）审核采购入库单

审核通过了以后，会干不少的事情，主要是更新库存

在更新库存的时候，如果说接口被重复调用的话，可能会导致库存就会出现问题，累加库存，如果被重复调用了，可能库存会被重复累加好几次

wms -> 调度中心 -> 库存中心，如果说这些接口被重复调用，就会导致库存出错，这个是非常要命的一个事情

wms中心的接口而言，是controller的一个接口，请求是从页面上传递过来的

对某一个采购入库单执行了一次审批，要么是通过，要么是不通过，这个采购入库单对这个接口他其实就只能调用一次

我们是否可以使用基于mysql去重表的这么一个逻辑，我觉得还真行，是库管领导在审核， 这个并发是很低很低的，既有mysql去重表，这边本质是在修改数据的状态，如果第一次数据状态修改成功了，那么下一次再次执行这个接口根据状态判断一下，如果不是合理的状态，这个接口就不让执行了

wms中心本地的那个更新库存的逻辑，就一定只会执行一遍

wms中心 -> 调度中心，接口调用，如果这里网络超时，导致了重试，就会导致调度接口重复执行，调度中心内部的库存会累加多次，导致库存中心的库存会累加多次

下一讲继续来分析和改造接口为幂等性，调度中心+库存中心的累加库存的接口，要保证幂等性，不能重复的累加库存

wms中心 -> 财务中心，创建采购结算单，也要保证幂等性，不能创建多个重复的采购结算单出的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/04\_电商系统的采购流程的接口幂等性方案以及开发（二）/笔记.doc**

结合业务来分析一下，一个采购入库单，只可能调用一次这个接口

从这里拿出来一个采购入库单的id，插入到一张mysql的去重表里面，这个接口只能调用一次，如果被重复调用，采购入库单id再次重复插入去重表，会唯一索引报错，不让你走后面的逻辑

我们在这里先不要考虑通用的幂等性服务，很多服务都有通过方法入参保证接口幂等性的需求，可以单独拉一个幂等性保证的服务出来，但是在这之前，我们可以先通过每个服务都自己做的一个方式来搞一下，体验一下在各个业务里做幂等性的时候

采购流程的接口幂等性的问题都分析清楚了，每个接口都做了对应的接口幂等性的保障方案了

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~资料/07\_对电商系统的创建订单流程进行分布式事务改造（四）/笔记.doc**

你可以认为说我们刚才做的那套tcc事务，就相当于是这里的本地事务

库存服务发消息到MQ，让调度服务来消费

库存服务异步消费消息，同步调用调度服务，开启一个tcc事务

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~资料/03\_设计商品购物流程的分布式事务方案结合业务落地的各个细节/笔记.doc**

XA（排除掉，微服务架构里面，我们大型互联网公司的实践，建议一个服务一个库，不要一个服务多个库，多个服务一个库，多个服务建议是同一个子系统）

TCC、可靠消息最终一致性方案、最大努力通知方案

（1）将商品加入购物车

（2）购物车界面里，都会提供一个按钮，购物车结算

（3）进入一个订单确认的界面，此时就会综合优惠券、促销活动、运费，给你计算出来订单的各种价格，包括了减免的价格，总价格。比如说收货地址、支付方式、期望的到货时间、发票抬头、是否使用积分抵扣一定的金额、是否购买一个退货险

（4）如果你感觉差不多了，你就点击提交订单的按钮

（5）创建订单

5-1）订单中心，将订单以及订单条目，插入到数据库中

5-2）订单中心，更新订单的状态为：待支付，1

5-3）订单中心，插入一条订单操作日志：创建订单

5-4）库存中心，锁定本地库存，将每个购买的商品的库存都锁定了指定的数量

5-5）库存中心，异步发送一个消息到MQ，异步通知调度中心和WMS中心同时进行库存的锁定

为什么都可以是MQ异步化的呢？调度中心以及WMS中心的库存，不是被前台的用户实时的使用的，只不过是在后台异步化的进行调度商品出库入库的时候，才会用到，所以这块数据不一定说要那么实时的去更新的

订单中心的创建订单的操作，以及库存中心的锁定库存的操作，完全可以用TCC分布式事务给绑定在一起，要么一起成功，要么一起失败

如果订单中心报错，那么库存中心根本不会执行

如果库存中心报错，那么库存中心本地事务回滚，同时让订单中心之前的操作也要进行cancel

只有当地订单中心和库存中心都成功的时候，才会发送一条消息到到MQ，通知调度以及WMS中心异步化的进行出库调度以及库存的锁定

5-6）调度中心，对订单中要购买的商品，每个商品都完成发货的调度，持久化保存起来，调度好了，每个商品从哪个仓库哪个货位发货几件

5-7）调度中心，锁定本地的库存

5-8）WMS中心，锁定本地的库存

上面的3个异步化的操作，是不是完全可以使用可靠消息最终一致性的方案，就是说得发送待确认的消息给可靠消息服务，可靠消息服务来投递消息到MQ里面去，让调度中心来消费到务必要执行成功

（6）找支付中心获取一个支付二维码，显示在页面上，提示用户尽快支付

（7）用户扫描二维码支付成功了以后，微信/支付宝就会收到支付成功的通知，反过来会回调我们的支付中心的接口

（8）支付中心，收到支付成功的通知以后，就会回调订单中心说支付成功了

（9）订单中心，支付订单的流程

9-1）更新订单状态：待发货，3

9-2）订单中心，插入一条订单操作日志：支付订单

9-3）库存中心，扣减库存，发送异步消息到MQ

9-4）调度中心，扣减库存

9-5）WMS中心，扣减库存

9-6）调度中心，调度销售出库

9-7）WMS中心，创建销售出库单、发货单、物流单

9-8）会员中心，增加积分和成长值

9-9）发送一条短信，通知用户说已经完成支付，仓库正在尽力发货

9-10）WMS中心，不在业务流程，后面随时可以进行，审核销售出库单，通知财务中心打款，通知订单中心已发货

9-11）订单中心，确认收货，订单状态为：已完成

分布式事务主要就是在这个购物流程的两个子流程里面来运用，保证用户在购物的时候，数据一定是准确的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~资料/04\_对电商系统的创建订单流程进行分布式事务改造（一）/笔记.doc**

就是说在spring cloud的系统中，引入bytetcc框架，就可以实现tcc事务了

主要是在订单中心和库存中心中引入bytetcc框架

1、服务提供者如何改造

（1）pom.xml

<dependency>

<groupId>org.bytesoft</groupId>

<artifactId>bytetcc-supports-springcloud</artifactId>

<version>0.4.17</version>

<exclusions>

<exclusion>

<groupId>asm</groupId>

<artifactId>asm</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-dbcp2</artifactId>

</dependency>

（2）配置文件

几乎不用修改

（3）数据库表

对每个他操作的数据库，你都要去专门搞一个表

CREATE TABLE `bytejta` (

`xid` VARCHAR(32) NOT NULL,

`gxid` VARCHAR(40) DEFAULT NULL,

`bxid` VARCHAR(40) DEFAULT NULL,

`ctime` BIGINT(20) DEFAULT NULL,

PRIMARY KEY (`xid`)

) ENGINE=INNODB DEFAULT CHARSET=utf8

（4）日志文件

他其实通过文件和数据库，都会记录分布式事务执行过程中的一些日志或者记录，状态，进度，这些东西其实是用来在分布式事务执行到一半，服务挂了，服务重启，需要根据之前记录的日志和数据库里的记录，恢复这个分布式事务，继续执行

（5）数据源配置config

@Import(SpringCloudConfiguration.class)

相当于是导入了bytetcc框架为了整合spring cloud他自己的一个SpringCloudConfiguration，里面肯定会配置一大堆bytetcc框架实现的spring cloud相关的组件

@Bean(name = "dataSource")

这个是给DataSource起一个固定的名字，有没有作用，现在看不出来，但是难保人家会用这个名字去获取DataSource bean

LocalXADataSource dataSource = new LocalXADataSource();

LocalXADataSource，其实是bytetcc框架自己封装的一个东西，一看就是对数据源做了一层包装，这样的话呢在执行SQL语句的时候，这个bytetcc可以做一些事情，比如说根据SQL的执行去记录一些数据库里的状态

我们其实从这个框架的引入中，都可以发现一些端倪，bytetecc这种框架如果要实现一整天的分布式事务，肯定会侵入到哪几个点去实现自己的组件呢？

1）spring cloud的feign、ribbon相关的组件，很有可能会被人家做一些定制化，从@import导入一个SpringCloudConfiguration，一看就是人家框架自定义了一些spring cloud的一些组件

2）DataSource肯定被包装了，LocalXaDataSource，这块就可以让人家感知到我们在分布式事务里各个地方做的一些SQL操作

3）TransactionManager，屁股想想，肯定会是bytetcc自己实现了一些TransactionManager相关的东西，所以@Transactional注解启动的事务都是bytetcc在管理

（6）改造服务接口

在服务Controller上一般要加一个注解

@Compensable(interfaceClass = AccountAmountApi.class,

confirmableKey = "accountAmountConfirmService",

cancellableKey = "accountAmountCancelService")

在这个controller中的接口，其实就是try接口，负责锁定资源，或者尝试预分配一些资源，预处理，如果能正常执行的话，那么后面应该就是可以正常进行confirm的

在这个controller中的每个接口的方法，都需要加一个@Transactional注解，这个是人家强制要求的，规约

然后呢就是基于这个接口，再实现2个service bean，confirm bean和cancel bean，都实现了同一个接口，然后在里面实现confirm和cancel的逻辑

cancel，就是说如果try失败了，要用cancel来回滚；如果try成功了，要用confirm来完成操作；如果cancel或者是confirm失败了，那么就会被bytetcc框架不停的重试调用，直到你执行成功为止

（7）改造main类

@ImportResource({ "classpath:bytetcc-supports-springcloud.xml" })

尽量就是挑选最最核心的业务逻辑去上分布式事务，一旦在业务里引入了分布式事务之后，代码就会很复杂，改造的成本其实很高

按照正常的这么一个逻辑，我们应该是将库存表都要去加一个字段，冻结字段，frozen\_stock\_cnt这么一个字段，就是先将可销售的库存扣减，冻结库存给他增加

confirm，就是将冻结库存扣减，给锁定库存增加

cancel，扣减冻结库存，增加销售库存

基于咱们现有的代码尽量低成本的改造一下，主要是带着大家体验一下，在复杂的业务系统中如何实践各个方案

购物流程，创建订单子流程，支付订单子流程

我打算就改造一个创建订单子流程，TCC事务、可靠消息最终一致性方案、最大努力通知方案（发一条短信，通知一下用户尽快付款）

支付订单子流程，我们就放着别动，不打算动手改造了，留给大家当地作业，我的方案的图都已经给大家画好了，如果你能跟着我把创建订单子流程改造分布式事务做出来，支付订单子流程的分布式事务，肯定自己会做

2、服务消费者如何改造

（1）pom.xml

<dependency>

<groupId>org.bytesoft</groupId>

<artifactId>bytetcc-supports-springcloud</artifactId>

<version>0.4.17</version>

<exclusions>

<exclusion>

<groupId>asm</groupId>

<artifactId>asm</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-dbcp2</artifactId>

</dependency>

（2）配置文件

（3）数据库表

CREATE TABLE `bytejta` (?   `xid` varchar(32) NOT NULL,?   `gxid` varchar(40) DEFAULT NULL,?   `bxid` varchar(40) DEFAULT NULL,?   `ctime` bigint(20) DEFAULT NULL,?   PRIMARY KEY (`xid`)? ) ENGINE=InnoDB DEFAULT CHARSET=utf8

（4）数据源配置bean

@Import(SpringCloudConfiguration.class)

@Bean(name = "mybatisDataSource")

LocalXADataSource dataSource = new LocalXADataSource();

（5）接口调用改造

controller上也要加一个注解

@Compensable(interfaceClass = ITransferService.class, confirmableKey = "transferServiceConfirm", cancellableKey = "transferServiceCancel")

接口方法：@Transactional

在接口里，可能会执行本地数据库操作，也会同时基于feign调用远程的其他服务

此时本地数据库操作以及feign调用远程服务，就被包裹在一个分布式tcc事务里了，这个时候就会由bytetcc框架来托管管理这个事务的一个执行

1）本地的controller里面的接口，其实就是属于try接口，调用的其他服务的feign接口，也是try接口，无论有多少个其他的服务，都会去依次调用try接口

2）如果某个服务的try失败了，就会调用之前try成功的那些服务的cancel接扩来回滚，那个try失败的不用管他

3）如果所有服务的try成功了，那么就会依次执行本地和远程其他各个服务的confirm接口，来完成分布式事务操作

在本地需要实现confirm接口和cancel接口

（6）改造main类

@ImportResource({ "classpath:bytetcc-supports-springcloud.xml" })

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~资料/06\_对电商系统的创建订单流程进行分布式事务改造（三）/笔记.doc**

1、问题1

eshop-common里面，本来就定义了数据源config bean

在库存服务和订单服务里面，我们是自己定义了基于bytetcc的数据源config bean，他会跟eshop-common里面定义的数据源config会冲突

2、问题2

还有一个bug，在订单服务的InventoryTccService，里面的服务名称，inventory-service，是错的，eshop-inventory，服务名

3、测试JSON

{

"userAccountId": 1,

"username": "zhangsan",

"consignee": "张三",

"deliveryAddress": "测试地址",

"consigneeCellPhoneNumber": "测试号码",

"freight": 4.6,

"payType": 2,

"totalAmount": 100,

"discountAmount": 0,

"couponAmount": 0,

"payableAmount": 100,

"invoiceTitle": "测试发票抬头",

"taxpayerId": "测试纳税人识别号",

"orderComment": "测试订单",

"orderItems": [

{

"goodsId": 1,

"goodsSkuId": 1,

"goodsSkuCode": "TEST\_SKU\_01",

"goodsName": "测试商品01",

"saleProperties": "测试销售属性",

"goodsGrossWeight": 50,

"purchaseQuantity": 3,

"purchasePrice": 700,

"goodsLength": 10,

"goodsWidth": 20,

"goodsHeight": 10

}

]

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~资料/01\_分析一下电商系统3个黄金链路是否要上分布式事务方案/笔记.doc**

分布式事务的各种技术方案，都已经学过了，而且深入的学习了源码，方案的原理，或者结合一个流量充值中心的案例给实战了一下3616118202

项目实战，项目阶段一里面做了一个10万代码的有一定复杂度的业务的电商系统，电商系统v1.0，我们要将分布式事务的各种技术方案来运用到电商系统里面去

梳理一下电商系统的几个核心黄金链路是什么

（1）采购流程

（2）购物流程

（3）退货流程

在一个系统里面，建议是尽量只是在那种针对有金钱的，交易的地方，使用分布式事务

分布式事务本身，真的挺复杂的，所以如果对各种各样复杂的业务流程，你都用分布式事务的话，会导致你的系统的代码过度复杂，不好维护

（1）采购流程

采购部门 -> 采购单 -> 供应商 -> 采购入库单 -> 采购入库 -> 更新库存 -> 商品的库存会增加，就可以售卖了 -> 采购结算单 -> 财务系统定期打款给供应商（周结算、月结算、季度结算）

这个流程有没有必要说上分布式事务？

按照我的经验和个人建议而言，这种业务流程，涉及大量的复杂业务，但是不涉及到实时的资金交易、订单等特别敏感的不允许出错的数据

（2）购物流程

购物车 -> 提交订单 -> 创建订单（待支付） -> 二维码（支付宝/微信） -> 订单状态（待发货） -> 扣减库存 -> 调度出库 -> 增加积分 -> 发送短信

这个流程还是可以上一下的，扫描二维码支付，一旦支付成功，后续执行的一系列的操作都必须同步完成，如果说人家用户支付成功了，支付宝/微信回调了你的一个接口，通知你了，说谁谁谁支付成功了

结果你的订单状态不正确，还是显示的是：待支付

或者是库存没有成功扣减，或者是商品没有调度出库，或者是积分没有给人家增加上去，或者是迟迟没有发送短信给用户

对于一个电商系统而言，这个事情是非常非常的要命的

直接会导致你的公司崩溃和破产，只要用户发生过一次这种事情，失去信任，口碑变坏，从此以后不在你这里再买东西了

购物流程里面，尤其是一旦支付订单成功以后，后续的一系列的事情，都必须上分布式事务，必须得执行成功，不允许执行失败

在支付之前，创建订单的时候，会有一系列的操作，叫做库存的锁定，创建订单 -> 锁定库存，这块有没有必要上分布式事务呢？还是有必要的，就是说订单一旦创建之后，库存的更新必须是跟订单是同步的

库存是很关键的一块数据

你要是说，如果是商品调度出库的话，到是可以慢一些，可以在后台慢慢的去调度出库

订单创建 -> 库存的锁定，是不是最好是绑定为一个分布式事务，要么一起成功，要么就一起失败，如果库存锁定失败的话，有任何的报错，就要导致订单创建也失败

在购物流程里面，其实是充满了各种各样的需要分布式事务的场景

（3）退货流程

提交退货申请 -> 客服审核通过 -> 寄送商品回去 -> 确认收到退货 -> 退货入库单 -> 退货入库 -> 领导审核 -> 退款给用户

这个流程，说实话也还好，因为他不是涉及到一些特别实时性的，特别敏感性的一些操作，就是说过程中如果有一些少数的错误，你只要及时发现，然后在一两天之内解决和修复数据，给人家处理好，基本上还好

（4）分析一下

采购、退货，一定是频率比较低的行为，不经常发生的，你平时经常在网上购物的话，购买10次商品，有多少次是会退货的，我太太，重度淘宝剁手党，她可能买100次东西，才可能会退货1次

采购，每周采购一次，每个月采购一次

对于低频行为，你不用分布式事务也没事，如果出错了以后，大不了就修复一下就可以了

采购如果出错，导致有些库存没有增加上去，导致商品没有足够的库存售卖，一两天之内库存数量为0，商品暂时下架一两天

退货，如果中间出了一点问题的话，可能导致库存数据有少量的不对，给人家退款的时候，稍微慢了一两天，都还好

低频行为，而且出错的概率没那么大，偶尔出错手工修复一下数据，解决一下系统的代码bug为什么会报错

反而比你硬上分布式事务，成本来的低得多的多，分布式事务威力很大，很复杂，XA、TCC、可靠消息最终一致性、最大努力通知，都要引入很多复杂的MQ、额外的服务，开发、维护、测试，很麻烦

（5）下个结论

我们就以目前的电商系统来举例，3大黄金链路里，其实就是购物流程，加入分布式事务，就可以了，购物流程是比较频繁，不希望他出错的，而且很影响用户体验的这么一个电商公司里非常核心的这么一个环节

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~资料/05\_对电商系统的创建订单流程进行分布式事务改造（二）/笔记.doc**

在try阶段

（1）订单中心，就是创建一个订单，但是状态是：UNKOWN

（2）库存中心，扣减掉可销售库存，冻结资源

在confirm阶段

（1）订单中心，就是修改订单状态：待支付

（2）库存中心，增加锁定库存的数量

在cancel阶段（try阶段的订单中心/库存中心，报错）

（1）订单中心，更新订单状态：已取消

（2）库存中心，增加可销售库存

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~资料/02\_结合代码来细细梳理一下购物流程的各个环节/笔记.doc**

特别具体和细化的一些环节，步骤是什么

梳理出来这些东西以后，才可以决定说如何设计一整套的分布式事务方案

TCC、可靠消息最终一致性、最大努力通知，这种方案都是要结合你的业务特点来进行设计的，所以在设计之前先梳理清楚业务

你要考虑清楚，哪些环节你是要进行分布式事务的保护的，各种方案落地到复杂的业务系统里之后，应该是如何综合运用的

流量充值中心呢，相对来说还是比较简单的

（1）将商品加入购物车

（2）购物车界面里，都会提供一个按钮，购物车结算

（3）进入一个订单确认的界面，此时就会综合优惠券、促销活动、运费，给你计算出来订单的各种价格，包括了减免的价格，总价格。比如说收货地址、支付方式、期望的到货时间、发票抬头、是否使用积分抵扣一定的金额、是否购买一个退货险

（4）如果你感觉差不多了，你就点击提交订单的按钮

（5）创建订单

5-1）订单中心，将订单以及订单条目，插入到数据库中

5-2）订单中心，更新订单的状态为：待支付，1

5-3）订单中心，插入一条订单操作日志：创建订单

5-4）库存中心，锁定本地库存，将每个购买的商品的库存都锁定了指定的数量

5-5）库存中心，异步发送一个消息到MQ，异步通知调度中心和WMS中心同时进行库存的锁定

为什么都可以是MQ异步化的呢？调度中心以及WMS中心的库存，不是被前台的用户实时的使用的，只不过是在后台异步化的进行调度商品出库入库的时候，才会用到，所以这块数据不一定说要那么实时的去更新的

5-6）调度中心，对订单中要购买的商品，每个商品都完成发货的调度，持久化保存起来，调度好了，每个商品从哪个仓库哪个货位发货几件

5-7）调度中心，锁定本地的库存

5-8）WMS中心，锁定本地的库存

（6）找支付中心获取一个支付二维码，显示在页面上，提示用户尽快支付

（7）用户扫描二维码支付成功了以后，微信/支付宝就会收到支付成功的通知，反过来会回调我们的支付中心的接口

（8）支付中心，收到支付成功的通知以后，就会回调订单中心说支付成功了

（9）订单中心，支付订单的流程

9-1）更新订单状态：待发货，3

9-2）订单中心，插入一条订单操作日志：支付订单

9-3）库存中心，扣减库存，发送异步消息到MQ

9-4）调度中心，扣减库存

9-5）WMS中心，扣减库存

9-6）调度中心，调度销售出库

9-7）WMS中心，创建销售出库单、发货单、物流单

9-8）会员中心，增加积分和成长值

9-9）发送一条短信，通知用户说已经完成支付，仓库正在尽力发货

9-10）WMS中心，不在业务流程，后面随时可以进行，审核销售出库单，通知财务中心打款，通知订单中心已发货

9-11）订单中心，确认收货，订单状态为：已完成

分布式事务主要就是在这个购物流程的两个子流程里面来运用，保证用户在购物的时候，数据一定是准确的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/47~51资料/47\_电商系统中的库存超卖场景以及分布式锁解决方案/笔记.doc**

分布式锁的各种源码，分布式这个东西，在实际的业务系统开发里面，一般是在什么样的场景下可以进行实战

这个库存超卖的场景，主要是在我们下订单购买的时候，比如iphone 7我要买3台。不是会检查一下库存么？那个检查库存那里，就是有问题的。默认情况下，库存一定可能会存在超卖的可能性

我们来画图说明一下

iphone 7的库存还剩下5台，张三和李四每个人想要买3台，理论上来说，肯定是不让人家买的，张三先下了一个订单以后，可以成功，说要买3台iphone 7，但是订单下成功的同时，必须保证说锁定库存

此时iphone 7的库存必须变成2台

然后如果李四来购买这个iphone 7要下单买3台的话，应该是会提示人家说库存不足

所以说如果用分布式锁怎么来解决这个问题呢？在图里体现一下

加了分布式锁之后，对同一个商品的下订单 -> 检查库存 -> 创建订单 -> 锁定库存，这个流程，所有的请求会串行化

但是这种分布式锁解决库存超卖的方案，有什么使用的局限性呢？QPS太低了，基本上对于同一个商品的下单操作出现了串行化的问题，在高并发系统里是不可取的。对热门商品进行秒杀的场景，肯定不能这么玩儿，死定了。

但是，对于那些低并发的系统，电商系统，或者不存在热门商品瞬时多人下单的场景里，就无所谓了，这个方案还挺合适的，可以保证数据不出错

库存超卖的场景，给大家来讲解一下，分布式锁结合业务一般可以怎么玩儿，他可以如何解决业务的一些问题

库存超卖的问题，现在先用分布式锁这个方案来解决一下，后面，我们在中期规划的高并发环节里，会优化这些东西的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/47~51资料/48\_电商系统中的调度销售出库场景以及分布式锁解决方案/笔记.doc**

下订单的时候，其实有一个潜在的问题场景，库存超卖，多个人同时下单的时候，都是查询库存比较一下，然后再下单，此时如果不加分布式锁，或者不用一些有效的技术方案，有可能会导致就5个库存，结果卖出去了10个，20个，30个

下完订单之后，锁定库存中心的库存，然后会异步化发送消息（可靠消息最终一致性的方案）给到调度中心，锁定调度中心的库存，同时就会走一个计算，此时就会算出来如何调度销售出库的步骤

这个调度销售出库的场景，主要是在我们下订单之后，后台不是在异步化的进行调度销售出库么，这个过程需要完成商品如何发货的调度操作，这里也有库存数据的错乱问题

调度中心，他会查询出来当前这个商品的各种库存的数据，然后根据这个库存数据，有一个简单的调度销售出库的算法，会计算出来这个订单要买的这几件商品从哪个仓库、哪个货位发送几件出去

假如说针对某个商品的多个订单都完成了，都在异步的计算如何调度销售出库，有一个订单他先加载出来库存数据，根据这份库存数据在计算说如何发货；但是在与此同时，另外一个订单已经完成了调度的计算，而且还会锁定掉调度中心的这个库存，此时就会导致第一个订单的计算结果可能是错的

我们来画图说明一下

所以说如果用分布式锁怎么来解决这个问题呢？在图里体现一下

但是这种方案的局限性和之前说的是一样的，在高并发场景下都需要优化，但是低并发的场景，无所谓了

所以大家现在看到了吧，通过实际的业务场景分析，可以看到，其实分布式锁一个主要的使用场景之一，就是保证分布式系统下，多个机器上的服务，可能同时查询+修改一个共享数据，此时如果不加分布式锁，会导致数据改错，比如库存出错

那么就必须要在分布式系统中加分布式锁

分布式锁在业务系统里的使用场景是什么呢？从数据库里查询出来一份数据，根据这份数据做一些计算，在数据库中修改这份数据，像这个过程，特别适合用分布式锁，分布式系统里用分布式锁，来保证说，查询 -> 计算 -> 修改，这个过程是保证数据一定不会出错的

针对一个数据的复杂业务流程，会出现多个请求的串行化排队的这个问题，所以在高并发的系统里尽量避免，在低并发，有分布式系统潜在的并发冲突问题的时候，可以用这个分布式锁的解决

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/47~51资料/49\_在电商系统中基于分布式锁开发库存超卖解决方案/笔记.doc**

代码开发起来非常的简单

如果你要在一个项目中用分布式锁，你应该如何来做？

（1）你的系统对应的基础设施里面，要有redis集群，或者是zk集群

（2）在系统中引入curator / redisson框架的依赖

（3）然后就可以在代码中进行开发，基于curator / redisson框架来使用分布式锁的功能，就ok了，没什么特别的

（4）模拟一些多客户端并发对同一个商品下单的场景，做一些分布式锁的功能性的测试

VirtualBox我也不跑起来了，示范过了，都是可以的

我就直接就在需要使用分布式锁的系统里，引入依赖，写一些基础性的代码，在业务代码中加入分布式锁的使用

留一个作业，关于分布式锁在业务场景下的测试，以及进一步自己再次阅读分布式锁的源码，我会到时候给大家留一个作业

3616118202

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/47~51资料/51\_分布式锁在电商系统中实战以及源码透彻分析的作业布置/笔记.doc**

无论是redis锁，zk锁，框架的实现，都已经理解的非常的透彻了

结合电商里的两个业务场景，让大家明白了，什么用分布式锁？分布式系统里就可能要用到分布式锁，不同的机器上的进程并发的修改一些共享的数据，此时需要加锁，不然可能会出现类似于库存超卖，调度错乱之类的问题

（1）自己结合我给的那个代码，将订单中心和调度中心的分布式锁的代码，测试一下，先保证这个分布式锁的代码可以跑通

（2）测试的时候，建议大家就是说是可以结合你的测试的代码，将对应的源码流程在给读一下，你可以将订单中心、调度中心启动多个实例，让多个实例去争抢同一把分布式锁，源码层面看看他们在争抢锁的时候，底层的运行逻辑是什么样子的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/47~51资料/50\_在电商系统中基于分布式锁开发调度销售出库解决方案/笔记.doc**

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/14\_redis分布式锁（九）：公平锁源码剖析之定位加锁源码位置/笔记.doc**

我们来看一下第二种redis分布式锁

第一种锁是可重入锁，非公平可重入锁，所谓的非公平可重入锁是什么意思呢？胡乱的争抢，根本没有任何公平性和顺序性可言

第二种锁，可重入锁，公平锁

分布式锁，公平锁，如何来实现，基于redis的分布式公平锁如何实现？

通过公平锁，可以保证说，客户端获取锁的顺序，就跟他们请求获取锁的顺序，是一样的，公平锁，排队，谁先申请获取这把锁，谁就可以先获取到这把锁，这个是按照顺序来的，不是胡乱的争抢的

会把各个客户端对加锁的请求进行排队处理，保证说先申请获取锁的，就先可以得到这把锁，实现所谓的公平性

可重入非公平锁、公平锁，他们在整体的技术实现上都是一样的，只不过唯一不同的一点就是在于加锁的逻辑那里

非公平锁，加锁是比计较简单粗暴，争抢；公平锁，在加锁的逻辑里，要加入这个排队的机制，保证说各个客户端排队，按照顺序获取锁，不是胡乱争抢的

RedissonFairLock是RedissonLock的子类，整体的锁的技术框架的实现，都是跟之前讲解的RedissonLock是一样的，无非就是重载了一些方法，加锁和释放锁的lua脚本的逻辑稍微复杂了一些，别的没什么特别的

return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, command,

// remove stale threads

"while true do "

+ "local firstThreadId2 = redis.call('lindex', KEYS[2], 0);"

+ "if firstThreadId2 == false then "

+ "break;"

+ "end; "

+ "local timeout = tonumber(redis.call('zscore', KEYS[3], firstThreadId2));"

+ "if timeout <= tonumber(ARGV[4]) then "

+ "redis.call('zrem', KEYS[3], firstThreadId2); "

+ "redis.call('lpop', KEYS[2]); "

+ "else "

+ "break;"

+ "end; "

+ "end;"

+ "if (redis.call('exists', KEYS[1]) == 0) and ((redis.call('exists', KEYS[2]) == 0) "

+ "or (redis.call('lindex', KEYS[2], 0) == ARGV[2])) then " +

"redis.call('lpop', KEYS[2]); " +

"redis.call('zrem', KEYS[3], ARGV[2]); " +

"redis.call('hset', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +

"redis.call('hincrby', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

"local firstThreadId = redis.call('lindex', KEYS[2], 0); " +

"local ttl; " +

"if firstThreadId ~= false and firstThreadId ~= ARGV[2] then " +

"ttl = tonumber(redis.call('zscore', KEYS[3], firstThreadId)) - tonumber(ARGV[4]);" +

"else "

+ "ttl = redis.call('pttl', KEYS[1]);" +

"end; " +

"local timeout = ttl + tonumber(ARGV[3]);" +

"if redis.call('zadd', KEYS[3], timeout, ARGV[2]) == 1 then " +

"redis.call('rpush', KEYS[2], ARGV[2]);" +

"end; " +

"return ttl;",

Arrays.<Object>asList(getName(), threadsQueueName, timeoutSetName),

internalLockLeaseTime, getLockName(threadId), currentTime + threadWaitTime, currentTime);

既然你是研究分布式锁，那么就必须研究一整套的分布式锁的体系，redis分布式锁，可重入锁、公平锁、读写锁、RedLock锁、Semaphore、CountDownLatch，这样才能算是真的精通分布式锁了

面试突击第一季的课程，那个里面简单介绍了一下基于redis和zk的两种分布式锁的实现思路，那个基本上停留在理论和思路的角度，你如果在公司里落地企业级的分布式锁的方案和使用，必须是得跟我们这个课程一样来研究的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/46\_zookeeper分布式锁（四）：MultiLock源码剖析/笔记.doc**

MultiLock，依次遍历获取每个锁，阻塞直到获取每个锁为止，然后返回true

如果过程中有报错，依次释放已经获取到的锁，然后返回false

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/38\_redis分布式锁（三十三）：CountDownLatch源码剖析/笔记.doc**

trySetCount()，set anyCountDownLatch 3

awati()，while true死循环，不断的去判断，get anyCountDownLatch，获取到这个值，如果这个值是>0的话，就说明还没有达到指定数量的客户端去执行countDown的操作，就始终陷入一个死循环之中

不是很想给大家去讲那块逻辑，个人认为，redisson源码写的非常好，但是唯一美中不足的一点，就是在这里混入了一些PUBSUB，基于redis去做发布订阅的一些时间，redis怎么能做类似MQ的事情呢？

redis怎么能做发布订阅的事情呢？完全违背了这个技术的初衷

redis作者的喜好，redis应该发展的方向，是可以支持很多的应用场景，锁、队列、发布订阅，但是我觉得这样子搞就是适得其反

await()方法，其实就是陷入一个while true死循环，不断的get anyCountDownLatch的值，如果这个值还是大于0那么就继续死循环，否则的话呢，就退出这个死循环

countDown()，decr anyCountDownLatch，就是每次一个客户端执行countDown操作，其实就是将这个cocuntDownLatch的值递减1就可以了。如果这个值已经小于等于0，del anyCcoutnDownLatch，删除掉他就可以ile；

如果是这个值为0的时候，还会去发布一个消息，publish redisson\_countdownlatch\_\_channel\_\_{anyCountDownLatch} 0

可能要花费不少的时间去看里面的细节代码，那块东西我觉得封装的不是特别好，redisson源码别的地方都写的很清晰和漂亮，但是就是PUB/SUB这里，我觉得源码封装的特别不好，易读性不高

看源码的时候，抓大放小，既然已经把这个核心原理都给搞定了，那么其实我们就不用纠结于一些细节了

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/29\_redis分布式锁（二十四）：读写锁源码剖析之写锁与写锁互斥/笔记.doc**

客户端A加了一个写锁

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = UUID\_01:threadId\_01:write

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 1

}

客户端B也来尝试加写锁

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = UUID\_02:threadId\_02:write

hget anyLock mode，mode = write，已经有人加了一把写锁

hexists anyLock UUID\_02:threadId\_02:write，如果之前是客户端B自己加的写锁的话，才能进入下面的分支，但是实际上写锁并不是他加的，是客户端A加的，所以这个分支是不会进入的

直接返回一个ttl时间，导致加锁失败

不同的客户端之间加写锁，是会互斥的

同一个客户端同一个线程，重复加读锁会如何；读写锁是否互斥；写锁与写锁是否互斥

不同的客户端/线程之间，读锁与读锁不互斥，读锁与写锁互斥，写锁与写锁互斥

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/22\_redis分布式锁（十七）：MultiLock源码剖析之加锁与释放锁/笔记.doc**

仔细的分析一下MultiLock加锁和释放锁的源码，释放锁，一目了然，很简单

加锁，主要的逻辑，不是在lua，MultiLock是一个high level高层次的API，是基于底层的low level的API，RedissonLock来进行封装的

逻辑，主要是在java代码里，所以我们可以用debug的模式来研究一下他的原理

RedissonLock、RedissonFairLock，他们的逻辑没有办法debug，主要是在lua脚本里面

long baseWaitTime = locks.size() \* 1500;

locks是所有的底层包装的锁，locks.size()是3,3个lock，1500毫秒，baseWaitTime，基础的等待时间，4500毫秒，算出来了一个waitTime

while (true) {

if (tryLock(waitTime, leaseTime, unit)) {

return;

}

}

不停的尝试去获取到所有的锁，只有获取到所有的锁的时候，while true死循环才会退出，否则只要你的锁还没全部获取到，就会一直在while true死循环里

watTime = 4500毫秒

time = 当前时间

remainTime = 4500毫秒

lockWaitTime = 4500毫秒

failedLocksLimit = 0

awaitTime = 4500毫秒

lockAcquired = lock.tryLock(awaitTime, newLeaseTime, TimeUnit.MILLISECONDS);

lock是底层的RedissonLock，他没有使用lock.lock()，用的是tryLock()，指定了获取锁等待超时的时间，4500毫秒，必须获取到这个锁，如果获取不到这个锁，就退出，标记为获取锁失败

哪怕是获取到锁之后，这个锁在多长时间内会自动释放，newLeaseTime是-1，因为你的newLeaseTime是-1，所以说如果获取到了锁，会启动一个lock watchdog不断的刷新你的锁key的生存时间为30000毫秒

remainTime -= (System.currentTimeMillis() - time);

看不出来怎么回事的，为什么呢？因为这里是这样的，我们因为在debug导致获取一个锁的时间可能达到了几分钟，4500毫秒减去这么一个几分钟的时间，算出来的就是一个负数了

if (remainTime != -1) {

remainTime -= (System.currentTimeMillis() - time);

time = System.currentTimeMillis();

if (remainTime <= 0) {

unlockInner(acquiredLocks);

return false;

}

}

remainTime = 4500毫秒

经过了一个lock的获取，可能消耗掉了比如说20毫秒，100毫秒，500毫秒，耗费了500毫秒

remainTime = 4500毫秒 - 500毫秒 = 4000毫秒

time = 当前时间

如果remainTime <= 0，意味着什么呢？获取锁的时间已经超过了4500毫秒了，迄今为止，你获取到这些所的时间，已经超过了预设的4500毫秒了，相当于是你获取多个锁的时间，最多不能超过4500毫秒

如果一旦获取各个锁的时间超过了4500毫秒，此时就会释放掉所有已经获取的锁，然后返回一个false，再次进入while true中的一个死循环，尝试走上述一模一样的流程

获取了三把锁，耗时了1000毫秒，此时remainTime还剩下3500毫秒，4477，4500，23毫秒获取了三把锁

加锁的逻辑我们就已经全部看完了

释放锁的话，就是依次调用所有的锁的释放的逻辑，lua脚本，同步等待所有的锁释放完毕，才会返回

默认的行为之下，你包裹了几把锁，就会锁数量 \* 1500毫秒，获取所有的锁必须在多长时间之内就要结束，如果超时就会重新再次死循环尝试获取锁。使用的是各个锁的tryLock()方法，指定了说在获取每个单独的锁的时候，会有一个获取超时退出的时间

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/27\_redis分布式锁（二十二）：读写锁源码剖析之读锁与读锁非互斥/笔记.doc**

读锁与读锁是非互斥

客户端A，客户端B，两个客户端部署在不同的机器上，都要对anyLock这个锁加读锁，此时是不会互斥的，他们的加锁都可以成功

假设，客户端A（UUID\_01:threadId\_01），先加了一个读锁

hset anyLock mode read

hset anyLock UUID\_01:threadId\_01 1

set {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

pexpire {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 30000

pexpire anyLock 30000

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

然后此时客户端B在另外一台机器上的，同时也要来加读锁（UUID\_02:threadId\_02），也要加读锁

KEYS[1] = anyLock

KEYS[2] = {anyLock}:UUID\_02:threadId\_02:rwlock\_timeout

ARGV[1] = 30000毫秒

ARGV[2] = UUID\_02:threadId\_02

ARGV[3] = UUID\_02:threadId\_02:write

hget anyLock mode，获取到mode = read，此时已经有人加了一把读锁

hincrby anyLock UUID\_02:threadId\_02 1

set {anyLock}:UUID\_02:threadId\_02:rwlock\_timeout:1 1

pexpire anyLock 30000

pexpire {anyLock}:UUID\_02:threadId\_02:rwlock\_timeout:1 30000

ind其实就是累加了1之后的一个值，ind = 1，{anyLock}:UUID\_02:threadId\_02:rwlock\_timeout:1

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 1,

“UUID\_02:threadId\_02”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

{anyLock}:UUID\_02:threadId\_02:rwlock\_timeout:1 1

多个客户端，同时加读锁，读锁与读锁是不互斥的，只会让你不断的在hash里加入哪个客户端也加了一个读锁

每个客户端都会维持一个watchdog，不断的刷新anyLock的生存时间，同时也会刷新那个客户端自己对应的timeout key的生存时间

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/18\_redis分布式锁（十三）：公平锁源码剖析之队列重排/笔记.doc**

涉及到队列的重排序

恍恍惚惚的，又行进到了10:00:36秒，客户端C来进行的重新尝试进行加锁，此时客户端B他其实在这之前不知道可能因为网络原因或者是别的什么原因，可能他就是没有尝试过重新加锁

进入while true，拿到队列第一个元素的timeout时间，10:00:30 <= 10:00:36，条件成立

zrem redisson\_lock\_timeout:{anyLock} UUID\_02:threadId\_02，就从有序集合中将客户端B移除了，lpop redisson\_lock\_queue:{anyLock}，就从队列中将客户端B也移除了

此时队列的第一个元素是客户端C，他的timeout时间是10:00:35 <= 10:00:36，条件成立，从有序集合以及队列中删除掉客户端C

尝试进行加锁，不成立，因为客户端A还持有着那把锁

重新排队入队，pttl anyLock = 23000毫秒

ttl = 23000毫秒

timeout = 23000毫秒 + 10:00:36 + 5000毫秒 = 10:01:04

zadd和rpush两个指令，将客户端C压入队列和有序集合中

假设此时，在10:00:38，客户端B再次尝试来进行加锁，while true退出

尝试加锁，不成立

重新尝试入队

ttl = 10:01:04 - 10:00:38 = 26000毫秒

timeout = 26000毫秒 + 10:00:38 + 5000毫秒 = 10:01:09

zadd和rpush两个指令，重新尝试入队

从这个里面，我们可以看到，在一个客户端刚刚加锁之后，其他的客户端来争抢这把锁，刚开始在一定时间范围之内，时间不要过长，各个客户端是可以按照公平的节奏，在队列和有序集合里面进行排序

在一定时间范围内，时间不要过长，其实队列里的元素顺序是不会改变的，各个客户端重新尝试加锁，只不过是刷新有序集合中的分数（timeout），各个客户端的timeout不断加长，但是整体顺序大致还是保持一致的

但是如果客户端A持有的锁的时间过长，timeout，这个所谓的排队是有timeout，可能会在while true死循环中将一些等待时间过长的客户端从队列和有序集合中删除，一旦删除过后，就会发生各个客户端随着自己重新尝试加锁的时间次序，重新进行一个队列中的重排，也就是排队的顺序可能会发生变化

客户端跟redis通信的网络的一个问题，延迟，各种情况都可能会发生

客户端释放锁，释放锁之后队列中的排队的客户端是如何依次获取这把锁的，是按照队列里的顺序去获取锁的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/07\_redis分布式锁（二）：可重入锁源码剖析之lua脚本加锁逻辑/笔记.doc**

说一下

@Override

public RLock getLock(String name) {

return new RedissonLock(connectionManager.getCommandExecutor(), name);

}

我们关注的重点，只要知道，getLock()方法的时候，获取到的Lock对象是RedissonLock对象，就可以了，里面封装了一个ConnectionManager里获取的一个CommandExecutor，CommandExecutor是什么东西？

CommandExecutor用屁股想一下，里面一定是封装了一个跟redis之间进行通信的一个Cconnection连接对象，CommandExecutor，命令执行器，封装了一个redis连接的命令执行器，可以执行一些set、get redis的一些操作，用来执行底层的redis命令的

public RedissonLock(CommandAsyncExecutor commandExecutor, String name) {

super(commandExecutor, name);

this.commandExecutor = commandExecutor;

this.id = commandExecutor.getConnectionManager().getId();

this.internalLockLeaseTime = commandExecutor.getConnectionManager().getCfg().getLockWatchdogTimeout();

this.entryName = id + ":" + name;

}

在这个RedissonLock的构造函数里面，建议大家关注的一行代码，别的没什么，主要是一个internalLockLeaseTime的东西，跟watchdog看门狗有关系的

this.internalLockLeaseTime = commandExecutor.getConnectionManager().getCfg().getLockWatchdogTimeout();

@Override

public void lock() {

try {

lockInterruptibly();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

@Override

public void unlock() {

try {

get(unlockAsync(Thread.currentThread().getId()));

} catch (RedisException e) {

if (e.getCause() instanceof IllegalMonitorStateException) {

throw (IllegalMonitorStateException)e.getCause();

} else {

throw e;

}

}

// Future<Void> future = unlockAsync();

// future.awaitUninterruptibly();

// if (future.isSuccess()) {

// return;

// }

// if (future.cause() instanceof IllegalMonitorStateException) {

// throw (IllegalMonitorStateException)future.cause();

// }

// throw commandExecutor.convertException(future);

}

watchdog，看门狗，看起来是一个后台定时不断的执行的后台线程，我们直接去redisson源码包里找一下，尝试打一些断点看看看门狗的

interlnalLockLeaseTime：默认的一个值是30000毫秒，30秒

在默认情况下，加锁的时候，long leaseTime, TimeUnit unit，是没有的，-1和null，就代表着说，只要你加到了一把锁，就一定会永久性的持有这把锁，除非是你当前持有这把锁的机器宕机了，watchdog看门狗，是不是就会发现，然后就会释放锁，避免说永久性的一个死锁发生

"if (redis.call('exists', KEYS[1]) == 0) then " +

"redis.call('hset', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +

"redis.call('hincrby', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

"return redis.call('pttl', KEYS[1]);"

这坨东西，其实就是实际的针对redis加锁的这么一个底层的命令，这个是执行了一段针对redis的lua脚本，redis本身就是可以支持执行lua脚本，基于redis执行一些复杂得多逻辑和命令

if (redis.call('exists', KEYS[1]) == 0) then，KEYS[1]一看就是我们设置的那个锁的名字，人家先执行了redis的exists的指令，判断一下，如果“anyLock”这个key不存在，那么就进行加锁，实际加锁的指令

"redis.call('hset', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

hset，redis的一个指令，相当于是在redis的一个map数据结构里设置一个key value

hset KEYS[1] ARGV[2] 1

hset anyLock lockState 1

anyLock的一个map

{

“lockState”: 1,

“otherKey”: “otherValue”

}

pexpire KEYS[1] ARGV[1]，设置一个key的过期时间

KEYS[1]其实可以理解为就是我们设置的那个key，“anyLock”，ARGV[1]其实就是一个这个key的过期时间，可能是默认的一个值，叫做30000毫秒，30s，很有可能是说的是，这个anyLock这个key对应的过期时间就是30秒

默认情况下，到了30秒之后，他就自动过期

"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +

"redis.call('hincrby', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

hexits KEYS[1] ARGV[2]，这个意思就是说针对KEYS[1]（anyLock）这个名字的一个map，里面是否存在一个ARGV[2]的一个key（lockState），如果是存在的话，hincrby KEYS[1] ARGV[2] 1，将anyLock这个map中的lockState这个key的值累加1

anyLock为名称的一个map

{

“lockState”: 2,

“otherKey”: “otherValue”

}

又执行了一下pexpire anyLock 30000，再一次将anyLock这个key的有效期设置问了30秒

return redis.call('pttl', KEYS[1]);，这行指令，其实是pttl这个指令，就是返回anyLock这个key当前还剩下的一个有效的存活期，就是当前这个key还能存活多少毫秒，或者多少秒这样子，30000毫秒，29998毫米，2毫秒已经过去了

根据anyLock这个key，尝试去获取到一个NodeSource的这么一个东西，我们可以认为说，我们现在不是有3主3从的一个redis集群，用屁股推测和猜想一下，是不是有可能说他是通过key获取一个NodeSource

这个NodeSource有可能就是3个master实例中的一个，有可能就是说对这个key的设置，就释放在3个master的其中一个上而已

private NodeSource getNodeSource(String key) {

int slot = connectionManager.calcSlot(key);

MasterSlaveEntry entry = connectionManager.getEntry(slot);

return new NodeSource(entry);

}

redis cluster默认的slot数量，是16384，无论你创建的是多大的一个redis cluster，他统一有一个slot的一个划分，就是将集群的存储空间划分为16384个slot，这16384个slot就平均分布在各个master实例上

3个master，每个master实例有5000多slot

那么在针对redis cluster读写操作的时候，先会去基于hash算法计算这个slot，anyLock这个key，13434，算出来是这个slot，相当于是针对“anyLock”这个key计算出来一个hash值，然后将这个hash值对16384这个slot数量进行取模，int slot = connectionManager.calcSlot(key);

取模之后就可以拿到当前这个“anyLock”的key对应的是哪个slot

MasterSlaveEntry entry = connectionManager.getEntry(slot);，这行代码是什么意思？算出来了一个slot，就是根据这个slot的编号，获取到这个slot是属于哪个master实例的？？每个master实例是不是都挂载了一个slave实例

MasterSlaveEntry [masterEntry=[freeSubscribeConnectionsAmount=1, freeSubscribeConnectionsCounter=50, freeConnectionsAmount=32, freeConnectionsCounter=64, freezed=false, freezeReason=null, client=[addr=redis://192.168.31.114:7002], nodeType=MASTER, firstFail=0]]

redis://192.168.31.114:7002，编号为13434的slot所在的master是这台机器的这个端口对应的master实例

此时就是已经知道了，其实必须是将加锁的那段lua脚本，放到redis://192.168.31.114:7002这个master实例上去执行，完成加锁的操作

keys，是一个数组，[anyLock]，这个其实对应的就是上面分析的那个脚本里的KEYS[1]

params，是一个参数，[30000，8743c9c0-0795-4907-87fd-6c719a6b4586:1]，代表的其实就是ARGV[1]和ARGV[2]

"if (redis.call('exists', KEYS[1]) == 0) then " +

"redis.call('hset', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +

"redis.call('hincrby', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

"return redis.call('pttl', KEYS[1]);"

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = 8743c9c0-0795-4907-87fd-6c719a6b4586:1

8743c9c0-0795-4907-87fd-6c719a6b4586，代表的是一个UUID，其实就是这个客户端上的一个ConnectionManager的这么一个id，1是什么呢？1大家还记得吧，其实就是threadId，大体上可以认为是一个客户端的一个线程对应的唯一的标识

ARGV[2]，代表的就是一个客户端上的一个线程，对这个key进行了加锁

anyLock这个key是否存在？

hset anyLock 8743c9c0-0795-4907-87fd-6c719a6b4586:1 1

anyLock就对应一个map数据结构（hash）

{

“8743c9c0-0795-4907-87fd-6c719a6b4586:1”: 1

}

pexpire anyLock 30000：anyLock这个key只能存活30000毫秒，30秒，默认情况下，你使用redisson加锁，其实不是无限制的不停的可以拥有这把锁的，人家默认情况下给你设置的一个锁的有效期就是30秒，watchdog看门狗的代码的实现，人家一定是在不断的监控这个锁，如果到30秒，就自动释放掉这把锁，或者是自动延期这个key的有效时长

redisson实现了一个看门狗，推测，根据文档里面的东西来推测，可能这个看门狗到了30秒，就自动把这个key给延期了，在延期的时候，他可能是怎么做的呢？如果看门狗在30秒内再次执行了这个加锁的方法，此时很可能会走到下面的那段脚本

"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +

"redis.call('hincrby', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

hexists anyLock 8743c9c0-0795-4907-87fd-6c719a6b4586:1：判断一下，针对anyLock这个map，里面是否有一个8743c9c0-0795-4907-87fd-6c719a6b4586:1 key，如果有，就说明这个客户端的那个线程，对这个key之前是加过锁的，而且此时还持有锁，因为key还没过期

hincrby anyLock 8743c9c0-0795-4907-87fd-6c719a6b4586:1 1，针对anyLock这个map中的8743c9c0-0795-4907-87fd-6c719a6b4586:1这个key的值，累加1

anyLock就对应一个map数据结构（hash）

{

“8743c9c0-0795-4907-87fd-6c719a6b4586:1”: 2

}

这个值对应的数字就代表了，看门狗延长过多少次这个key的有效时长

pexpire anyLock 30000，再次将这个anyLock这个key的有效时长设置为了30000毫秒，30秒

大家好好的去体会一下这个加锁的一个指令，以及看门狗自动延长key有效期的这么一个妙用，看门狗的代码，我们现在还没看到，后面看一下，基本上那个看门狗的源码，就是干这么一件事情

如果客户端一直长期持有了一把锁，还没释放，但是redis里的key对应的有效期是30秒，看门狗会自动去延长那个key的有效期，更新一下延长的次数，只有当你的客户端手动释放unlock锁的时候，才会删除那个key

大家一定要看明白这个最最基本的分布式锁的一个妙用

加完一次锁之后，获取到的一个ttlRemainingFuture，这个东西，他是怎么回事？他其实就是说这个key未来还剩余的可以存活的时间是多少毫秒

我估计可能是因为我们在调试源码的原因，导致了他的代码的一些执行都是有问题的

纯调试，可能还是挺难的去看他的原理，debug的时候会导致他的源码运行不太正常，通过源码的一个分析，watchdog的原理，判断下客户端如果对一个锁还是持有锁的话，但是他对应的那个锁的key，会去判断他的一个当前剩余的过期时间，如果是快要过期的话， 就会去执行一段脚本，给那个key进行一个延期

我觉得我们针对一个分布式锁要去探索的是另外两个关键的东西：

（1）如果在某个机器上的某个线程，已经对key加锁了，那么这台机器上的其他线程如果尝试去对key加锁，会怎么样？肯定是会阻塞住，那么他是如何阻塞住的？

（2）如果在某个机器上的某个线程，已经对key加锁了，那么其他机器上的某个线程尝试去对那个key加锁，会怎么样？肯定也是会阻塞住的，那么他是如何阻塞住的呢？

对于这两个关键性的问题，我们都要去分析一下源码

如果设置了两个加锁的参数，如何在一定时间之后，自动释放锁？如何在尝试等待获取锁一段时间之后，如果还没获取到那把锁，就自动退出，标识为获取锁失败？

读写锁，读锁和写锁是互斥，写锁和写锁是互斥，读锁和读锁是非互斥；公平锁；MultiLock；同步组件，Semaphore、CountDownLatch

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/36\_redis分布式锁（三十一）：CountDownLatch原理图解/笔记.doc**

redisson分布式锁的组件，CountDownLatch

CountDownLatch最基本的原理，他可以要求必须有几个线程来获取这个锁，然后达到线程的数量之后，然后才能各个客户端的代码才能往下走，没有达到指定数量的客户端来获取这个锁，会导致各个客户端的代码会阻塞住

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/44\_zookeeper分布式锁（五）：非可重入锁源码剖析/笔记.doc**

非可重入锁

跟可重入锁其实是一样的，但是只不过没有提供可重入的支持

同一个时间只能有一个客户端获取到锁，其他人都是要排队的，同一个客户端是不可重入加锁的

基于semaphore来实现的，其实根本自己没有什么代码

只不过将semaphore允许获取锁的客户端的数量设置为了1，同一时间只能有一个客户端获取到锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/06\_redis分布式锁（一）：可重入锁源码剖析之使用场景介绍/笔记.doc**

redis分布式锁的实现原理

redis最知名的一个开源客户端框架，redisson，源码

redisson如何来进行redis分布式锁实现的源码，基于redis实现各种各样的分布式锁的原理

几乎不用花费多少的精力在于学习redis分布式锁的使用，看一下他的文档，看一下他对各种锁的功能的介绍

RLock lock = redisson.getLock("anyLock");

看一下上面这一行代码，分布式锁是什么东西？以及分布式锁在什么场景下使用，我之前给大家都讲解过了，分布式系统里面，如果多个机器上的服务要同时对一个共享资源（比如说修改数据库里的一份数据），此时的话，某台机器就需要先获取一个针对那个资源（数据库里的某一行数据）的分布式锁

获取到了分布式锁之后，就可以任由你查询那条数据，修改那条数据，在这个期间，没有任何其他的客户端可以来修改这条数据，获取了一个分布式锁之后，就对某个共享的数据获取了一定时间范围内的独享的操作

其他的客户端如果同时要修改那条数据，尝试去获取分布式锁，就会被卡住，他需要等待第一个客户端先操作完了之后释放锁

数据库里，有一行数据

表名：users

id name age

1 张三 21

客户端A（机器01）：

我需要先查询id=1的一条数据，然后对他进行修改，但是我要求的是，从我查询出来以后，别人不能修改这条数据。查出来的时候age=20，此时他也要把age变成21，但是客户端B已经将age变成21了。所以此时客户端A再次将age=21，数据就不对了。

客户端A先针对id=1的这条数据，获取一个基于redis的分布式锁，RLock lock = redisson.getLock("users\_1");，获取的是users表中的id=1的这条数据的一个分布式锁，获取到了锁之后，再去查询id=1的数据，将age修改为21

客户端B（机器02）：同时将id=1的数据查出来，然后修改了age=21

同时，也要去查询和修改id=1的这条数据，没关系的，尝试去获取一个分布式锁，RLock lock = redisson.getLock("users\_1");，但是此时因为别人已经基于redis对这个“users\_1”这个key加了锁了，此时客户端B会阻塞，不断的等待尝试获取这个key（users\_1）对应的锁

直到客户端A，用完了这个锁之后，会释放锁，lock.unlock();，客户端B才能获取到“users\_1”这个key对应的锁，然后才能查询id=1的数据，age=21，此时将age设置为22，这样的话呢，多个客户端同时并发的修改同一个共享数据和资源，就不要紧了

搞java的，有一定的java经验的同学，对java多线程并发以及锁、同步或多或少都该有一些接触的

RLock lock = redisson.getLock("anyLock");

// 最常见的使用方法

lock.lock();

就是尝试获取“anyLock”这个key对应的一个锁，获取到了锁之后，尝试使用lock.lock()方法去进行加锁，此时如果没有人已经加了这把锁，你就可以加锁，如果别人已经对这个“anyLock”的锁加锁了，你此时就会阻塞住

这里，我们后面看源码的时候，就要注意一下，redisson在客户端里实现了一个看门狗，watchdog，主要是监控持有一把锁的客户端是否还存活着，如果还存活着，那么看门狗会不断的延长这个锁的过期时间

可以指定一个leaseTime，你获取了一把锁之后，可能你在锁定的期间，执行的操作特别的耗时，可能长达10分钟，1个小时。你就可以在获取锁的时候指定一个leaseTime，比如说，指定好，如果我自己1分钟之内没释放这把锁，redisson自动释放这把锁，让别的客户端可以获取锁来做一些操作。

// Acquire lock and release it automatically after 10 seconds

// if unlock method hasn't been invoked

lock.lock(10, TimeUnit.SECONDS);

// Wait for 100 seconds and automatically unlock it after 10 seconds

boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);

lock.unlock();

客户端A已经获取了一把锁，此时客户端B尝试去获取这把锁，是不是要等待？默认情况下是无限制的等待，但是这里你在获取锁的时候是可以指定一个时间的，最多等待100秒的时间，如果获取不到锁直接就返回，boolean res，这个res如果是false就代表你加锁失败了，在指定时间范围内，没有获取到锁

如果获取到了锁之后，在10秒之内，没有手动释放锁，那么就自动释放锁

lock.unlock();

很简单，就是你通过lock.lock()、lock.tryLock()方法获取到了一把锁之后，在锁定期间把你该干的事儿给干完了之后，你就可以使用lock.unlock()方法释放掉这把锁，让其他的客户端获取这把锁，进行事务的处理

RLock lock = redisson.getLock("anyLock");

lock.lockAsync();

lock.lockAsync(10, TimeUnit.SECONDS);

Future<Boolean> res = lock.tryLockAsync(100, 10, TimeUnit.SECONDS);

如果是lock.lock()方法，是属于同步加锁，在这些代码执行的期间，如果等待锁什么的，都会被阻塞住，lock.lockAsync()，异步加锁，用了其他的线程去进行加锁，不会阻塞你当前主线程的执行

Future<Boolean> res，不断的去查询这个feture对象的一些状态，看看异步加锁是否成功

你用哪个线程去加一把分布式锁，就必须用那个线程来对分布式锁进行释放，否则如果用不同的线程，会导致说IllegalMonitorStateException

咱们深入研究分布式锁的第一讲，所以就讲到这里就可以了，下一讲，我们就来研究一下最普通的分布式可重入锁的实现原理，看一下redisson的源码

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/10\_redis分布式锁（五）：可重入锁源码剖析之锁的互斥阻塞/笔记.doc**

如果已经有一个客户端的线程对一个key加了锁，那么此时其他的线程或者是客户端如果也要对这个key加锁，是如何被阻塞住的呢？部署在其他机器上的服务实例，或者是部署在其他机器上的其他服务

exists anyLock，是否存在这个锁key？存在的了

hexists anyLock 另一台机器的id:另外一个thread id，存在吗？当然不存在了

return redis.call('pttl', KEYS[1]);

如果说客户端A已经上锁了，还持有着这把锁，此时客户端B尝试加锁，此时就会直接执行pttl anyLock指令，返回这个key剩余的一个存活时间

ttlRemaining不是null，说明加锁没成功，就不会启动后台的定时调度任务每隔10秒去刷新锁key的生存周期770396102

long threadId = Thread.currentThread().getId();

Long ttl = tryAcquire(leaseTime, unit, threadId);

// lock acquired

if (ttl == null) {

return;

}

tryAcquire()方法尝试加锁，获取到一个ttl，如果是一个线程第一次加锁，ttl一定是null；如果是一个线程多次加锁，可重入锁的概念，此时ttl也一定是null，lua脚本里返回的就是nil；但是如果加锁没成功，锁被其他机器占用了，你执行lua脚本直接获取到的是这个key对应的剩余生存时间

如果ttl是null，证明加锁成功，就会直接返回

如果ttl不是null，证明加锁不成功，此时就会走阻塞逻辑

while (true) {

ttl = tryAcquire(leaseTime, unit, threadId);

// lock acquired

if (ttl == null) {

break;

}

// waiting for message

if (ttl >= 0) {

getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);

} else {

getEntry(threadId).getLatch().acquire();

}

}

如果加锁不成功，直接会进入while(true)就是一个死循环内

在死循环内，再次执行这个ttl = tryAcquire(leaseTime, unit, threadId);，尝试去获取这个分布式的锁，如果获取到了锁，证明ttl是null，此时就会退出死循环，如果ttl大于等于0，说明其他的客户端还是占据着这把锁

if (ttl >= 0) {

getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);

} else {

getEntry(threadId).getLatch().acquire();

}

这两行代码的细节，不用过多的关注，里面涉及到了其他的同步组件，Semaphore，在这里的话呢，其实大家只要知道一点，如果获取锁不成功，此时就会等待一段时间，再次投入到while(true)死循环的逻辑内，尝试去获取锁

以此循环往复

看到这样的一个分布式锁的阻塞逻辑，如果一个客户端的其他线程，或者是其他客户端的线程，尝试获取一个已经被加锁的key的锁，就会在while(true)死循环里被无限制的阻塞住，无限制的等待，尝试获取这把锁

揣测一下，如果没有获取到一把分布式锁，可能就是等待那个ttl指定的时间，再次去尝试获取那把锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/20\_redis分布式锁（十五）：公平锁源码剖析之按顺序依次加锁/笔记.doc**

如果客户端A释放了锁，删除了锁key之后，客户端B和客户端C是如何按照顺序依次加锁的。假设客户端B在队列里面是排在后面的，假设锁被释放掉了之后，是客户端B先拉尝试加锁

10:00:40，锁已经被释放了，客户端B来尝试重新加锁

10:01:04 <= 10:00:40？不成立

exists anyLock = 0，当前锁不存在；exists redisson\_lock\_queue:{anyLock} = 0，要不然就是队列不存在，但是现在队列是存在的；lindex redisson\_lock\_queue:{anyLock} 0 = UUID\_02:threadId\_02，队列存在，排在队头的是客户端B也可以

所以上面整体条件不成立，无法加锁

ttl = 10:01:04 - 10:00:40 = 24000毫秒

timeout = 24000 + 10:00:40 + 5000 = 10:01:09

zadd指令，刷新一下客户端B在有序集合中的timeout分数，10:01:09

哪怕是锁释放掉了，其他各个客户端来尝试重新加锁也是不行的，因为此时排在队头的不是这个客户端也不行，此时只会重新计算timeout分数刷新一下有序集合中的timeout分数罢了

此时客户端C来尝试加锁会如何？

anyLock锁key不存在的；队列是存在的；队列的队头就是客户端C，所以此时加锁的条件成立了，进入加锁的逻辑

lpop redisson\_lock\_queue:{anyLock}，将队列中的第一个元素弹出来

zrem redisson\_lock\_timeout:{anyLock} UUID\_03:threadId\_03，将有序集合中的客户端C的线程id的元素给删除掉

hset anyLock UUID\_03:threadId\_03 1，加锁

pexpire anyLock 30000，设置生存时间为30000毫秒

完成加锁，而且客户端C从队列中出队，此时排在队头的就是客户端B了

获取锁超时，其他客户端获取不到锁，一定会在java代码里进入一个while true死循环，一定时间内没有获取到锁，就返回false标识获取锁失败，过了一段时间，只要没有刷新有序集合中的timeout分数，就会自然被lua脚本里的while true逻辑给清理掉

超时自动释放锁，不会开启lock watchdog后台定时调度的任务

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/41\_zookeeper curator的使用代码演示/笔记.doc**

搭建一个工程，引入curator的依赖，写一些他的基础的代码，演示一下他的使用

如果各位同学有谁对zookeeper是一无所知的话，如果连这个东西是什么都不知道，我觉得有点不应该，作为一个java工程师的话，或多或少都应该知道一点点zookeeper是什么东西，对不对

说实话我们公司里不用curator，但是的话呢，我可以给大家说一下curator里面分布式锁这块的源码，这块源码对我来说其实很熟悉的，因为我们平时自己封装的zk客户端框架对分布式锁的实现，几乎跟curator是差不多的

我带着大家来看一下curator这个框架在分布式锁支持这块的源码

zookeeper的版本他不是3.4.9么？curator 2.x版本可以适配zookeeper 3.4.x版本

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/37\_redis分布式锁（三十二）：CountDownLatch使用演示/笔记.doc**

RCountDownLatch latch = redisson.getCountDownLatch("anyCountDownLatch");

latch.trySetCount(3);

System.out.println(new Date() + "：线程[" + Thread.currentThread().getName() + "]设置了必须有3个线程执行countDown，进入等待中。。。");

for(int i = 0; i < 3; i++) {

new Thread(new Runnable() {

@Override

public void run() {

try {

System.out.println(new Date() + "：线程[" + Thread.currentThread().getName() + "]在做一些操作，请耐心等待。。。。。。");

Thread.sleep(3000);

RCountDownLatch localLatch = redisson.getCountDownLatch("anyCountDownLatch");

localLatch.countDown();

System.out.println(new Date() + "：线程[" + Thread.currentThread().getName() + "]执行countDown操作");

} catch (Exception e) {

e.printStackTrace();

}

}

}).start();

}

latch.await();

System.out.println(new Date() + "：线程[" + Thread.currentThread().getName() + "]收到通知，有3个线程都执行了countDown操作，可以继续往下走");

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/33\_redis分布式锁（二十八）：Semaphore原理图解/笔记.doc**

redis分布式锁，可重入锁、公平锁、MultiLock、RedLock、读写锁

可重入锁：非公平锁，最最基础的分布式锁，我们最最常用的锁

公平锁：各个客户端尝试获取锁的时候会排队，按照队列的顺序先后获取锁

MultiLock：底层就是一次性要锁住多个小lock，让你一次性锁多个资源

RedLock：相当于是一把锁，虽然底层包裹了多个小lock，但是那多个小lock并不是说对应着多个资源，而是说每个锁key对应一个redis实例，只要大多数的redis实例加锁成功，就认为这个整体的RedLock锁加锁成功，健壮性比其他的普通的锁要好一些

redis普通的锁，都是在一个redis实例上加锁，写入master实例，如果还没异步同步到slave实例上去，master实例宕机了，自动切换到slave实例上去，新的master此时没有那个锁key，其他客户端在新的master上可以对同一个锁key加锁成功

可能导致多个客户端同时持有一个资源的锁

RedLock算法，底层是对应着多个小lock，每个小lock应该是在一个redis实例上去的，他每次都要在大多数的redis master实例上加锁成功，3个master实例，2个master实例上加锁成功，才算是一把锁加成功了

有一个客户端，在3个master实例，假设成功在里面加了3个master实例的锁， 不幸的是其中一台master突然宕机，还没同步到slave实例上去，此时他的slave切换成了新的master。

另外一个客户端尝试加锁，此时只能在那个新切换过来的那个master实例上加锁，另外两个master是无法成功加锁的，这样就能保证他加锁是不会成功的

这个算法就是所谓的RedLock算法

大家可以自己去尝试分析一下，枚举一下各种情况，这个算法还是有漏洞的

5个master实例，客户端A尝试加锁，仅仅成功的在3个master实例加了锁，成功了；此时不幸的是此时3个master中的1个master突然宕机了，锁key还没同步到他的slave实例上去，此时salve切换为新的master

5个master，其中一个是新切换过来的master，其实只有2个master是有客户端A加锁的一个痕迹的，另外3个master是没有这个锁key的

然后的不幸的是，此时客户端B来加锁，他其实很有可能可以成功的在3个master上成功加锁，达到了一个大多数的数字，完成了加锁，还是会发生说多个客户端同时重复加锁，所以说也是不是完全靠谱的

redisson已经尽力了，搞了各种锁，还搞了两个同步组件，Semaphore，CountDownLatch，zk的各种分布式锁，zk的机制真的比redisson是要靠谱的多

读写锁：读锁、写锁，各种锁可以互斥，可以非互斥，可以支持更加复杂的加锁的语义

Semaphore也是redis分布式锁支持的一种，同步组件

之前给大家的讲解的锁，基本上都是同时间只能一个客户端获取这个锁，然后做一些事情，处理完了以后释放锁

Semaphore，信号量，他作为一个锁机制，可以允许多个线程同时获取一把锁，任何一个线程释放锁之后，其他等待的一个线程就可以尝试进来获取一下这个锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/35\_redis分布式锁（三十）：Semaphore源码剖析/笔记.doc**

get semaphore，获取到一个当前的值

set semaphore 3，将这个信号量同时能够允许获取锁的客户端的数量设置为3

get semaphore，获取到一个当前的值，比如说是3，3 > 1

decrby semaphore 1，将信号量允许获取锁的客户端的数量递减1，变成2

decrby semaphore 1

decrby semaphore 1

导致semaphore的值变为0，无法允许任何客户端来获取这个值了

第四个，第五个，客户端来尝试获取锁，0 >= 1，不是，所以会导致此时获取semaphore的锁会失败，导致进入一个while true的死循环，不断的尝试去获取这个semaphore的锁，如果不行就等待一下，再次死循环去获取，直到有人退出让他获取到这个锁

incrby semaphore 1，每次一个客户端释放掉这个锁的话，就会将信号量的值累加1，信号量的值就不是0了

其他的客户端此时就可以尝试去获取到这个信号量的锁，再次将这个semaphore的值递减1，变为0，就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/13\_redis分布式锁（八）：可重入锁源码剖析之总结/笔记.doc**

做一个总结，从实现原理以及源码的层面，真正剖析和了解到了redis分布式锁的企业级的实现，这个分布式锁实现的还是非常漂亮的，麻雀虽小，五脏俱全，分布式的可重入锁，总结一下redis

（1）加锁：在redis里设置hash数据结构，生存周期是30000毫秒

（2）维持加锁：代码里一直加锁，redis里的key会一直保持存活，后台每隔10秒的定时任务（watchdog）不断的检查，只要客户端还在加锁，就刷新key的生存周期为30000毫秒

（3）可重入锁：同一个线程可以多次加锁，就是在hash数据结构中将加锁次数累加1

（4）锁互斥：不同客户端，或者不同线程，尝试加锁陷入死循环等待

（5）手动释放锁：可重入锁自动递减加锁次数，全部释放锁之后删除锁key

（6）宕机自动释放锁：如果持有锁的客户端宕机了，那么此时后台的watchdog定时调度任务也没了，不会刷新锁key的生存周期，此时redis里的锁key会自动释放

（7）尝试加锁超时：在指定时间内没有成功加锁就自动退出死循环，标识本次尝试加锁失败

（8）超时锁自动释放：获取锁之后，在一定时间内没有手动释放锁，则redis里的key自动过期，自动释放锁

这8大机制，组合在一起，才是构成了一个企业级的基于redis的分布式锁的方案

redisson基于redis实现的分布式锁的核心原理给搞通透了，后续我们再看其他的锁，包括公平锁、读写锁、MultiLock、RedLock这一系列的源码的时候，就比较得心应手了，分析源码会稍微快一些，主要分析各种不同的锁之间的实现原理的区别

redis加锁，本质，还是在redis集群中挑选一个master实例来加锁，master -> slave，实现了高可用的机制，如果master宕机，slave会自动切换为master

假设客户端刚刚在master写入一个锁，此时发生了master的宕机，但是master还没来得及将那个锁key异步同步到slave，slave就切换成了新的master。此时别的客户端在新的master上也尝试获取同一个锁，会成功获取锁

此时两个客户端，都会获取同一把分布式锁，可能有的时候就会导致一些数据的问题

redisson的分布式锁，隐患主要就是在这里

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/19\_redis分布式锁（十四）：公平锁源码剖析之释放锁/笔记.doc**

也就是说在客户端A他释放锁的时候，也会走while true的脚本逻辑，看一下有序集合中的元素的timeout时间如果小于了当前时间，就认为他的那个排队就过期了，就删除他，让他后面重新尝试获取锁的时候重排序

while true的逻辑，比如说客户端B或者客户端C，他们用的是tryAcquire()方法，他们其实设置了一个获取锁超时的时间，比如说他们在队列里排队，但是尝试获取锁超过了20秒，人家就不再尝试获取锁了

此时他们还是在队列和有序集合里占了一个坑位，while true的逻辑就可以保证说剔除掉这种不再尝试获取锁的客户端，有序集合里的timeout分数就不会刷新了，随着时间的推移，肯定就会剔除掉他

如果客户端宕机了，也会导致他就不会重新尝试来获取锁，也就不会刷新有序集合中的timeout分数，不会延长timeout分数，while true的逻辑也可以剔除掉这种宕机的客户端在队列里的占用

因为网络延迟等各种因素在里面，可能会在等待锁时间过长的时候，触发各个客户端的排队的顺序的重排序，有的客户端如果在队列里等待时间过长了，那么其实是可以触发一次队列的重排序的

他在这里发布一个锁被释放的消息，肯定在他的源码中是有一些人是订阅了这个释放锁的消息的，此时他们就可以得到一个锁被释放掉的通知

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/24\_redis分布式锁（十九）：RedLock源码剖析之算法实现/笔记.doc**

RedissonRedLock锁的实现，非常的简单，他是RedissonMultiLock的一个子类，RedLock算法的实现，是依赖于MultiLock的一个机制来实现的

@Override

protected int failedLocksLimit() {

return locks.size() - minLocksAmount(locks);

}

protected int minLocksAmount(final List<RLock> locks) {

return locks.size()/2 + 1;

}

@Override

protected long calcLockWaitTime(long remainTime) {

return Math.max(remainTime / locks.size(), 1);

}

主要就是通过方法的重载，改变了MultiLock中的几个特殊的行为

failedLocksLimit，locks.size() - minLocksAmount(locks) = 1，也就是failedLocksLimit，就是能够加锁失败的一个数量

minLocksAmount(locks)里面就是用的大多数节点的一个算法，n / 2 + 1，比如你有3个lock，那么至少要成功加上3 / 2 + 1 = 2个lock

RedLock里包裹了3个小锁，最多能够加锁失败1个锁

calcLockWaitTime，这个东西算出来的时间，是说在对每个lock进行加锁的时候，有一个尝试获取锁超时的时间，原来默认的就是remainTime，4500毫秒，4500毫秒 / 3 = 1500毫秒，每个小lock获取锁超时的时间改成了1500毫秒

waitTime = 4500毫秒

remainTime = 4500毫秒

lockWaitTIme = 1500毫秒

failedLocksLimit，默认是0，就是一个锁都不能容忍加失败了，现在最新的是n - (n / 2 + 1)，3个lock，最多可以容忍1个lock加锁失败

waitTime = 1500毫秒

对每个小lock尝试加锁的时候，能够容忍的最大超时时间，就是1500毫秒，1500毫秒之内必须加成功这个小锁，否则就是加锁失败

大家先试想一下，如果是之前讲解的那个最原始的MultiLock，如果任何一个lock加锁失败了，走这里会如何呢？failedLocksLimit = 0，不可能的，因为你都已经一个锁加锁失败了，所以此处不可能等于0

如果failedLocksLimit是0，默认的一个行为，此时就会将所有已经加锁的lock都释放掉，返回一个false，标记本次尝试加锁失败

之前讲解的那个MultiLock的话，只要你有任何一个锁加锁失败，此次加这个MultiLock就会标记为失败，再重来一次

但是，现在的话呢，使用的是RedLock，faildLocksLimit（3个小lock的时候，这个值是1），可以容忍一个锁加锁失败的，此时就会将failedLockLimit--，从1变为了0，也就是说之前可以容忍一次lock加锁失败，但是此时已经失败了一次了，不能再容忍加锁失败了

如果第二个lock加锁又失败了，此时failedLocksLimit已经是0了，那么就会标记为加锁失败，RedLock加锁失败了

如果是假设是3个lock，前2个加锁成功的，最后一个加锁失败了，3 - 2 = 1 = failedLocksLimit，此时是正好需要的最少的加锁的次数都加锁成功了，剩余的锁加锁失败了，也不要紧的，本次加锁就成功了

也就是说，针对多个lock进行加锁，每个lock都有一个1500毫秒的加锁超时时间，如果在4500毫秒内，成功的对n / 2 + 1个lock加锁成功了，就可以认为这个RedLock加锁成功了，不要求所有的lock都加锁成功的

RedLock，是一个锁，只不过是在各个不同的master实例上进行加锁，但是现在说RedLock合并了多个小lock。也就是说，如果你有3个redis master实例，你就用lock1、lock2、lock3三个锁key，人家本来就是分布在3个不同的redis master实例上的

加这个RedLock，相当于是3个redis master实例上的key，有2个加成功了，就算这个RedLock加锁成功了

此时别人如果要来加锁，用一样的key，人家是无法成功加锁的，锁被你占用了，人家就会卡在那儿，死循环，不断的尝试加锁，直到你释放锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/42\_zookeeper分布式锁（一）：可重入锁源码剖析/笔记.doc**

1、获取锁的源码

关键性的加锁代码

ourPath = client.create().creatingParentsIfNeeded().withProtection().withMode(CreateMode.EPHEMERAL\_SEQUENTIAL).forPath(path);

client.create().creatingParentsIfNeeded()：这块东西就是说，他要去创建znode，但是这个node他的说明是如果有必要就自动创建他的父目录

withProtection()：不管他了

withMode(CreateMode.EPHEMERAL\_SEQUENTIAL)：使用的临时顺序节点，首先他是临时节点，如果当前这台机器如果自己宕机的话，他创建的这个临时节点就会自动消失，如果有获取锁的客户端宕机了，zk可以保证锁会自动释放的

顺序节点，在创建这个节点的时候，zk会自动按照顺序给创建的节点标上序号

path：/locks/lock\_01/lock-

顺序节点，他的意思就是说，你给他传递的一个path是类似于，传递过去的path-这样的一个名字，他zk在创建节点的时候，会按照你创建节点的顺序，搞出来类似于，lock-01，lock-02，lock-03这样子有顺序的一坨节点

/locks/lock\_01/\_c\_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000001

上面那个东西，就是在加锁的时候，自己创建出来了一个临时顺序节点，后面你不断的创建node，他会不断的给你按照顺序累加数字

/locks/lock\_01/\_c\_0abad917-53a6-4ed9-ac96-bfac3327be0d-lock-0000000002

maxLeases是什么意思呢？每次允许几个客户端同时获取一个锁，默认肯定是最多允许一个客户端获取一把锁

2、不同客户端获取锁的互斥

outIndex = 1

maxLeases = 1

ourIndex - maxLeases = 0

3、客户端可重入加锁

客户端A重复进行加锁

4、客户端释放锁

client.delete().guaranteed().forPath(ourPath);，这行代码就是说保证一定会给你删除掉指定的这个path

5、释放锁后其他客户端获取锁

之前客户端B获取不到锁，是不是针对客户端A的那个顺序节点加了一个监听器，此时如果客户端A最终释放锁，将自己的顺序节点给删除，此时客户端B会收到对应的监听的通知，然后可以尝试获取锁

private final Watcher watcher = new Watcher()

{

@Override

public void process(WatchedEvent event)

{

notifyFromWatcher();

}

};

客户端B在监听客户端A的顺序节点是否被删除，如果被删除了，就会回调他的这个监听器的方法

6、公平锁

默认情况下，就是一个公平锁，所有的客户端都会创建顺序节点，都会按照自己申请锁的顺序来排序，最后都会依次按照自己所在的排序来尝试获取到锁，所有客户端相当于都是在排队获取到锁

redission公平锁其实是一样的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/15\_redis分布式锁（十）：公平锁源码剖析之排队加锁/笔记.doc**

第一件事情，如果此时某个锁，没有人加锁，此时第一个客户端跑进来加锁，会出现什么事情呢？

进入一个while true的死循环，光是看while true死循环你其实也不知道他在干嘛

KEYS = Arrays.<Object>asList(getName(), threadsQueueName, timeoutSetName)

KEYS[1] = getName() = 锁的名字，“anyLock”

KEYS[2] = threadsQueueName = redisson\_lock\_queue:{anyLock}，基于redis的数据结构实现的一个队列

KEYS[3] = timeoutSetName = redisson\_lock\_timeout:{anyLock}，基于redis的数据结构实现的一个Set数据集合，有序集合，可以自动按照你给每个数据指定的一个分数（score）来进行排序

从这里开始，我们接触到的这个redis的一些数据结构会比较多一些，我建议大家，自己之前事先可以先补充一些redis的基础知识，redis是支持很多种这个数据结构，队列、集合、hash，等等吧，大家先自己随便百度一下redis数据结构的学习，可以先看一下

学java编程，List、Map、Set，之类的，是一样的

ARGV = internalLockLeaseTime, getLockName(threadId), currentTime + threadWaitTime, currentTime

ARGV[1] = 30000毫秒

ARGV[2] = UUID:threadId

ARGV[3] = 当前时间（10:00:00） + 5000毫秒 = 10:00:05

ARGV[4] = 当前时间（10:00:00）

lindex redisson\_lock\_queue:{anyLock} 0，就是从redisson\_lock\_queue:{anyLock}这个队列中弹出来第一个元素，刚开始，队列是空的，所以什么都获取不到，此时就会直接退出while true死循环

exists anyLock，锁不存在，也就是没人加锁，刚开始确实是没人加锁的，这个条件肯定是成立的；要么是exists redisson\_lock\_queue:{anyLock}，这个队列不存在，或者是lindex redisson\_lock\_queue:{anyLock} 0，队列的第一个元素是UUID:threadId，或者是这个队列存在，但是排在队头的第一个元素，是当前这个线程

那么这个条件整体就可以成立了

anyLock和队列，都是不存在的，所以这个条件肯定会成立

lpop redisson\_lock\_queue:{anyLock}，弹出队列的第一个元素，现在队列是空的，所以什么都不会干

zrem redisson\_lock\_timeout:{anyLock} UUID:threadId，从set集合中删除threadId对应的元素，此时因为这个set集合是空的，所以什么都不会干

hset anyLock UUID:threadId 1，加锁

anyLock: {

“UUID\_01:threadId\_01”: 1

}

pexpire anyLock 30000，将这个锁key的生存时间设置为30000毫秒

返回一个nil，在外层代码中，就会认为是加锁成功，此时就会开启一个watchdog看门狗定时调度的程序，每隔10秒判断一下，当前这个线程是否还对这个锁key持有着锁，如果是，则刷新锁key的生存时间为30000毫秒

下一步，就是已经有人加锁成功了，此时其他的客户端来尝试加锁，如何进行排队？公平锁的核心，就在于说，申请加锁的时候，各个客户端会排队，依次获取锁，实现公平性，第二个客户端端来申请加锁会怎么样呢？

进入while true死循环，lindex redisson\_lock\_queue:{anyLock} 0，获取队列的第一个元素，此时队列还是空的，所以获取到的是false，直接退出while true死循环

exists anyLock是否不存在？当然不是了 ，此时anyLock这个锁key已经存在了，说明已经有人加锁了，这个条件首先就肯定不成立了；判断一下，此时这个第二个客户端是UUID\_02，threadId\_02，此时会判断一下，hexists anyLock UUID\_02:threadId\_02，判断一下在anyLock这个map中，是否存在UUID\_02:threadId\_02这个key？这个条件也不成立

排队的关键逻辑

lindex redisson\_lock\_queue:{anyLock} 0，从队列中获取第一个元素，此时队列是空的，所以什么都不会有

ttl = pttl anyLock = 20000毫秒，获取anyLock剩余的生存时间，ttl假设这里就被设置为了20000毫秒

timeout = ttl + 当前时间 + 5000毫秒 = 20000毫秒 + 10:00:00 + 5000毫秒 = 10:00:25

zadd redisson\_lock\_timeout:{anyLock} 10:00:25 UUID\_02:threadId\_02

这行命令的意思，就是在set集合中插入一个元素，元素的值是UUID\_02:threadId\_02，他对应的分数是10:00:25（会用这个时间的long型的一个时间戳来表示这个时间，时间越靠后，时间戳就越大），sorted set，有序set集合，他会自动根据你插入的元素的分数从小到大来进行排序

rpush redisson\_lock\_queue:{anyLock} UUID\_02:theadId\_02

这个指令就是将UUID\_02:threadId\_02，插入到队列的头部去

返回的是ttl，也就是anyLock剩余的生存时间，如果拿到的返回值是ttl是一个数字的话，那么此时客户端B而言就会进入一个while true的死循环，每隔一段时间都尝试去进行加锁，重新执行这段lua脚本

如果此时有一个客户端C，也来进行加锁，此时会如何呢？

while true死循环，lindex redisson\_lock\_queue:{anyLock} 0，获取队列中的第一个元素，UUID\_02:threadId\_02，代表的是这个客户端02正在队列里排队

zscore redisson\_lock\_timeout:{anyLock} UUID\_02:threadId\_02，从有序集合中获取UUID\_02:threadId\_02对应的分数，timeout = 10:00:25

假设客户端C尝试加锁的当前时间是10:00:05，timeout <= 10:00:05？，这个条件不成立，退出死循环，我们后面给大家再来讲

判断当前是否没有人加锁？不成立；判断如果加锁了是不是客户端C自己加的锁？不成立

入队，排队

lindex redisson\_lock\_queue:{anyLock} 0，获取队列中的第一个元素，UUID\_02:threadId\_02

ARGV[2] = UUID\_03:threadId\_03，也就是说在队列里排队的第一个元素并不是当前尝试获取锁的这个客户端

zscore redisson\_lock\_timeout:{anyLock} UUID\_02:threadId\_02 = 10:00:25

ttl = 10:00:25 - 10:00:05 = 20000毫秒

timeout = 20000毫秒 + 10:00:05 + 5000毫秒 = 10:00:30

zadd redisson\_lock\_timeout:{anyLock} 10:00:30 UUID\_03:threadId\_03

rpush redisson\_lock\_queue:{anyLock} UUID\_03:theadId\_03

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/43\_zookeeper分布式锁（二）：Semaphore源码剖析/笔记.doc**

可重入锁

Semaphore信号量

redisson semaphore刚刚讲完，就是可以指定同时有多个线程获取到锁

lock = new InterProcessMutex(client, ZKPaths.makePath(path, LOCK\_PARENT));

锁的path是：/semaphores/semaphore\_01/locks

lease path：/semaphores/semaphore\_01/leases

如果一个客户端尝试去获取semaphore的锁，直接核心的逻辑，就是尝试获取自动生成的一个锁：/semaphores/semaphore\_01/locks，底层就是基于顺序节点来排队获取锁的，用的是我们之前给大家讲解的那个锁的原理

这个普通的锁同一时间只能有一个客户端获取到这个锁，其他的客户端都会通过顺序节点排队等候

PathAndBytesable<String> createBuilder = client.create().creatingParentsIfNeeded().withProtection().withMode(CreateMode.EPHEMERAL\_SEQUENTIAL);

/semaphores/semaphore\_01/leases/\_c\_a9302e20-de9c-4356-923a-274664d7676c-lease-0000000000

在lease path下，创建了一个顺序节点

创建了一个Lease，这个Lease对象其实就代表了客户端A成功加的一个semaphore的这个锁，他先尝试成功加锁，如果成功加锁了以后，他就可以创建一个这个锁对应的一个lease，比如semaphore一共可以允许最多3个客户端同时加锁

最多是允许不超过3个客户端获取锁，他在这里会判断一下逻辑，就是说当前/semaphores/semaphore\_01/leases目录下的节点的数量是否<=3

如果是<=3的话，此时就会realse释放掉之前加的那个/semaphores/semaphore\_01/locks锁

return new Lease()

{

@Override

public void close() throws IOException

{

try

{

client.delete().guaranteed().forPath(path);

}

catch ( KeeperException.NoNodeException e )

{

log.warn("Lease already released", e);

}

catch ( Exception e )

{

throw new IOException(e);

}

}

@Override

public byte[] getData() throws Exception

{

return client.getData().forPath(path);

}

};

在释放semaphore锁的时候，会调用Lease的close()方法

内部借助了一个普通的锁，保证说所有的客户端都是按照顺序在尝试加semaphore锁的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/31\_redis分布式锁（二十六）：读写锁源码剖析之释放读锁/笔记.doc**

假设现在已经有各种锁各种加锁了之后，释放读锁，逻辑是如何的呢？

（1）不同客户端加了读锁 / 同一个客户端/线程多次可重入加了读锁

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 2,

“UUID\_02:threadId\_02”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:2 1

{anyLock}:UUID\_02:threadId\_02:rwlock\_timeout:1 1

（2）同一个客户端/线程先加写锁再加读锁

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 1,

“UUID\_01:threadId\_01”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

我们现在来看一下，针对上述三种情况，读锁是如何释放的呢？

KEYS[1] = anyLock

KEYS[2] = redisson\_rwlock:{anyLock}

KEYS[3] = {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout

KEYS[4] = {anyLock}

ARGV[1] = 0

ARGV[2] = UUID\_01:threadId\_01

hget anyLock mode，mode = read

hexists anyLock UUID\_01:threadId\_01，肯定是存在的，因为这个客户端A加过读锁

hincrby anyLock UUID\_01:threadId\_01 -1，将这个客户端对应的加锁次数递减1，现在就是变成1，counter = 1

del {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:2，删除了一个timeout key

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 1,

“UUID\_02:threadId\_02”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

{anyLock}:UUID\_02:threadId\_02:rwlock\_timeout:1 1

hlen anyLock > 1，就是hash里面的元素超过1个

pttl {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1，此时获取那个timeout key的剩余生存时间还有多少毫秒，比如说此时这个key的剩余生存时间是20000毫秒

其实是获取到了所有的timeout key的最大的一个剩余生存时间，假设最大的剩余生存时间是25000毫秒

pexpire anyLock 25000

客户端A再来释放一次读锁

hincrby anyLock UUID\_01:threadId\_01 -1，将这个客户端对应的加锁次数递减1，现在就是变成1，counter = 0

hdel anyLock UUID\_01:threadId\_01，此时就是从hash数据结构中删除客户端A这个加锁的记录

del {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1，删除了一个timeout key

anyLock: {

“mode”: “read”,

“UUID\_02:threadId\_02”: 1

}

{anyLock}:UUID\_02:threadId\_02:rwlock\_timeout:1 1

会用timeout key的剩余生存时间刷新一下anyLock的生存时间，pexpire

客户端B再来释放一次读锁

hincrby anyLock UUID\_02:threadId\_02 -1，将这个客户端对应的加锁次数递减1，现在就是变成1，counter = 0

hdel anyLock UUID\_02:threadId\_02，此时就是从hash数据结构中删除客户端A这个加锁的记录

del {anyLock}:UUID\_02:threadId\_02:rwlock\_timeout:1，删除了一个timeout key

anyLock: {

“mode”: “read”

}

hlen anyLock = 1

del anyLock，当没有人再持有这个锁的读锁的时候，此时会识别出来，就会彻底删除这个读锁，整个读锁释放的全过程，给大家就分析清楚了

针对第二种情况，同一个客户端先加写锁，然后加读锁的情况

hincrby anyLock UUID\_01:threadId\_01 -1，将这个客户端对应的加锁次数递减1，现在就是变成1，counter = 0

hdel anyLock UUID\_01:threadId\_01，此时就是从hash数据结构中删除客户端A这个加锁的记录

del {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1，删除了一个timeout key

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 1

}

mode是write的话，返回一个值是0就可以了

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/26\_redis分布式锁（二十一）：读写锁源码剖析之加写锁的逻辑/笔记.doc**

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = UUID\_01:threadId\_01:write

hget anyLock mode，此时肯定是没有的，因为根本没这个锁

hset anyLock mode write

hset anyLock UUID\_01:threadId\_01:write 1

pexpire anyLock 30000

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 1

}

加写锁就已经成功了

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/03\_搭建几台linux虚拟机为redis集群部署做准备/笔记.doc**

1、准备5台linux虚拟机

都是CentOS 7操作系统，64位，每台机器都是1个cpu，1G内存，安装JDK 1.8

（1）使用课程提供的CentOS 7镜像即可，CentOS-7-x86\_64-Minimal-1611.iso。

（2）创建虚拟机：打开Virtual Box，点击“新建”按钮，点击“下一步”，输入虚拟机名称为hadoop01，选择操作系统为Linux，选择版本为Red Hat-64bit，分配1024MB内存，后面的选项全部用默认，在Virtual Disk File location and size中，一定要自己选择一个目录来存放虚拟机文件，最后点击“create”按钮，开始创建虚拟机。

（3）设置虚拟机网卡：选择创建好的虚拟机，点击“设置”按钮，在网络一栏中，连接方式中，选择“Bridged Adapter”。（搭建虚拟机还真不是我专门要讲的一块内容，我一般常用的搭建虚拟机做实验的方式就是桥接网络）

如果你的笔记本电脑在家里用wifi的ip地址配置了一下虚拟机的网络，然后你的电脑拿到外面咖啡馆里去，或者公司里去，ip地址变化了之后，就会导致你的之前搭建的那套虚拟机环境就没法用了

如果你希望无论到哪里，ip地址变化了以后，虚拟机都可以正常工作，不需要重新配置网络，那么你就去百度一下，虚拟机如何配置网络，可以更换环境的时候，不需要重新配置，NAT如何配置

桥接，换了一个网络环境，ip地址都换了，5台虚拟机重新配置一下里面的网卡的一些ip地址什么的，就ok了

（4）安装虚拟机中的CentOS 7操作系统：选择创建好的虚拟机，点击“开始”按钮，选择安装介质（即本地的CentOS 7镜像文件），按照课程选择后自动安装即可

（5）安装完以后，CentOS会提醒你要重启一下，就是reboot，你就reboot就可以了。

（6）配置网络

（如果重启网卡没看到人家给你分配的ip地址，你自己应该是要去连一个路由器就可以了）

vi /etc/sysconfig/network-scripts/ifcfg-enp0s3

先让它动态分配一个ip地址

ONBOOT=yes

service network restart

ip addr

再设置静态ip地址

BOOTPROTO=static

IPADDR=192.168.31.250

NETMASK=255.255.255.0

GATEWAY=192.168.31.1

service network restart

ip addr

配置DNS

检查NetManager的状态：systemctl status NetworkManager.service

检查NetManager管理的网络接口：nmcli dev status

检查NetManager管理的网络连接：nmcli connection show

设置dns：nmcli con mod enp0s3 ipv4.dns "114.114.114.114 8.8.8.8"

让dns配置生效：nmcli con up enp0s3

（7）配置hosts

vi /etc/hosts

配置本机的hostname到ip地址的映射

（8）配置SecureCRT

此时就可以使用SecureCRT从本机连接到虚拟机进行操作了

一般来说，虚拟机管理软件，virtual box，可以用来创建和管理虚拟机，但是一般不会直接在virtualbox里面去操作，因为比较麻烦，没有办法复制粘贴

SecureCRT，在windows宿主机中，去连接virtual box中的虚拟机

收费的，我这里有完美破解版，跟着课程一起给大家，破解

（9）关闭防火墙

systemctl stop firewalld.service

systemctl disable firewalld.service

关闭windows的防火墙

后面要搭建集群，有的大数据技术的集群之间，在本地你给了防火墙的话，可能会没有办法互相连接，会导致搭建失败

（10）配置yum

yum clean all

yum makecache

yum install -y wget

（11）安装JDK

1、将jdk-8u131-linux-x64.rpm通过WinSCP上传到虚拟机中

2、安装JDK：rpm -ivh jdk-8u131-linux-x64.rpm

3、配置jdk相关的环境变量

vi ~/ .bashrc

export JAVA\_HOME=/usr/java/latest

export PATH=$PATH:$JAVA\_HOME/bin

source .bashrc

4、测试jdk安装是否成功：java -version

（12）在另外4个虚拟机中安装CentOS集群

按照上述步骤，再安装三台一模一样环境的linux机器

另外三台机器的hostname分别设置为hadoop02，hadoop03，hadoop04，hadoop05

安装好之后，在每台机器的hosts文件里面，配置好所有的机器的ip地址到hostname的映射关系

比如说，在elasticsearch01的hosts里面

192.168.31.250 hadoop0101

192.168.31.xxx hadoop02

192.168.31.xxx hadoop03

192.168.31.xxx hadoop04

（13）配置5台CentOS为ssh免密码互相通信

首先在三台机器上配置对本机的ssh免密码登录

ssh-keygen -t rsa，生成本机的公钥，过程中不断敲回车即可，ssh-keygen命令默认会将公钥放在/root/.ssh目录下

cd /root/.ssh

cp id\_rsa.pub authorized\_keys

将公钥复制为authorized\_keys文件，此时使用ssh连接本机就不需要输入密码了

接着配置三台机器互相之间的ssh免密码登录

使用ssh-copy-id -i hostname命令将本机的公钥拷贝到指定机器的authorized\_keys文件中

2、hdfs HA集群部署规划

namenode，就相当于是在eclipse中新建一个工程，取个名字叫做namenode，在工程里面开发代码，写好代码之后，就可以打包，在linux服务器上，用java -jar给他启动，namenode就是这个意思

namenode就是一个java工程，写完里面的代码，打包，在linux上部署，java -jar启动，出来一个进程，namenode其实就是一个系统，启动的时候就是一个进程

hadoop01：active namenode，ZKFC

hadoop02：standby namenode，ZKFC

hadoop03：datanode，journal node，QuorumPeerMain

hadoop04：datanode，journal node，QuorumPeerMain

hadoop05：datanode，journal node，QuorumPeerMain

3、部署zookeeper集群

在hadoop03、hadoop04、hadoop05三台机器上部署zookeeper

下载zookeeper 3.4.9：? HYPERLINK "http://archive.apache.org/dist/zookeeper/zookeeper-3.4.9/" \t "https://blog.csdn.net/zz657114506/article/details/\_blank" ?http://archive.apache.org/dist/zookeeper/zookeeper-3.4.9/?，放到/home/apps目录下去

tar -zxf zookeeper-3.4.9.tar.gz

vi /etc/profile

export ZOOKEEPER\_HOME=/home/apps/zookeeper-3.4.9 ?export PATH=$PATH:$ZOOKEEPER\_HOME/bin

source /etc/profile

vi zoo.cfg （$ZOOKEEPER\_HOME/conf）

dataDir=/home/data/zkdata ?dataLogDir=/home/log/zk ?server.1=hadoop03:2888:3888

server.2=hadoop04:2888:3888 ?server.3=hadoop05:2888:3888

mkdir /home/data/zkdata

mkdir /home/log/zk

cd /home/data/zkdata

echo 1 > myid

scp -r /home/apps/zookeeper-3.4.9 hadoop04:/home/apps/ ，然后干一样的事儿?scp -r /home/apps/zookeeper-3.4.9 hadoop05:/home/apps/，然后干一样的事儿

在hadoop04将myid的内容改为2 （echo 2 > myid） ?在hadoop05将myid的内容改为3 （echo 3 > myid）

三台机器上执行：zkServer.sh start

查看集群状态 ?1、jps（查看进程） ?2、zkServer.sh status（查看集群状态，主从信息）

4、部署hadoop集群

先在hadoop01上操作

tar -zxf hadoop-2.9.1.tar.gz -C /home/apps/

vim /etc/profile

export JAVA\_HOME=/home/apps/jdk1.8.0\_111

export HADOOP\_HOME=/home/apps/hadoop-2.9.1

export PATH=$PATH:$JAVA\_HOME/bin:$HADOOP\_HOME/bin

source /etc/profile

vi $HADOOP\_HOME/etc/hadoop/hadoop-env.sh

export JAVA\_HOME=/home/apps/jdk1.8.0\_111

vi core-site.xml

<configuration>

<property>

<name>fs.defaultFS</name>

<value>hdfs://ns1/</value>

</property>

<property>

<name>hadoop.tmp.dir</name>

<value>/home/apps/hadoop-2.9.1/tmp</value>

</property>

<property>

<name>ha.zookeeper.quorum</name> <value>hadoop01:2181,hadoop02:2181,hadoop03:2181</value>

</property>

</configuration>

<configuration>

<property>

<name>dfs.nameservices</name>

<value>ns1</value>

</property>

<property>

<name>dfs.ha.namenodes.ns1</name>

<value>nn1,nn2</value>

</property>

<property>

<name>dfs.namenode.rpc-address.ns1.nn1</name>

<value>hadoop01:9000</value>

</property>

<property>

<name>dfs.namenode.http-address.ns1.nn1</name>

<value>hadoop01:50070</value>

</property>

<property>

<name>dfs.namenode.rpc-address.ns1.nn2</name>

<value>hadoop02:9000</value>

</property>

<property>

<name>dfs.namenode.http-address.ns1.nn2</name>

<value>hadoop02:50070</value>

</property>

<property>

<name>dfs.namenode.shared.edits.dir</name>

<value>qjournal://hadoop03:8485;hadoop04:8485;hadoop05:8485/ns1</value>

</property>

<property>

<name>dfs.journalnode.edits.dir</name>

<value>/home/apps/hadoop-2.9.1/journaldata</value>

</property>

<property>

<name>dfs.ha.automatic-failover.enabled</name>

<value>true</value>

</property>

<property>

<name>dfs.client.failover.proxy.provider.ns1</name>

<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>

</property>

<property>

<name>dfs.ha.fencing.methods</name>

<value>

sshfence

shell(/bin/true)

</value>

</property>

<property>

<name>dfs.ha.fencing.ssh.private-key-files</name>

<value>/root/.ssh/id\_rsa</value>

</property>

<property>

<name>dfs.ha.fencing.ssh.connect-timeout</name>

<value>30000</value>

</property>

</configuration>

vi slaves

hadoop03

hadoop04

hadoop05

上面其实就已经将active namenode给配置好了

然后将这个hadoop拷贝到其他4台机器上去：scp -r /home/apps/ hadoop02:/home/apps/

5、启动hadoop集群

（1）在hadoop03、hadoop04、hadoop05上启动journal nodes集群：

sbin/hadoop-daemon.sh start journalnode

（2）格式化namenode（hadoop01上执行）

hdfs namenode -format

hdfs namenode -bootstrapStandby

（3）格式化ZKFC（hadoop01上执行）

hdfs zkfc -formatZK

（4）启动hdfs集群（hadoop01上执行）

sbin/start-dfs.sh，这个脚本会自动在hadoop01和hadoop02上启动一个namenode，同时启动一个ZKFC，然后会自动在hadoop03、hadoop04、hadoop05上分别启动一个datanode

6、查看hadoop集群

在hadoop01和hadoop02访问50070端口即可

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/11\_redis分布式锁（六）：可重入锁源码剖析之释放锁/笔记.doc**

比较关键的一块，释放锁，主动释放锁，宕机自动释放锁

宕机自动释放锁，这个锁对应的就是redis里的一个key，如果这个机器宕机了，对这个key不断的刷新其生存周期的后台定时调度的任务就没了，redis里的key，自动就会在最多30秒内就过期删除

其他的客户端就可以成功加锁了

get(unlockAsync(Thread.currentThread().getId()));，里面的那个unlockAsync()是异步化执行的一个方法，释放锁的操作是异步执行的，get(unlockAnsync())，get()包裹了之后就是会同步的等待异步执行的结果

"if (redis.call('exists', KEYS[1]) == 0) then " +

"redis.call('publish', KEYS[2], ARGV[1]); " +

"return 1; " +

"end;" +

"if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " +

"return nil;" +

"end; " +

"local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1); " +

"if (counter > 0) then " +

"redis.call('pexpire', KEYS[1], ARGV[2]); " +

"return 0; " +

"else " +

"redis.call('del', KEYS[1]); " +

"redis.call('publish', KEYS[2], ARGV[1]); " +

"return 1; "+

"end; " +

"return nil;",

如果anyLock这个key不存在，publish redisson\_lock\_\_channel\_anyLock 0，redis是支持发布/订阅模型的，就是说可以对里面的某个channel key进行订阅，订阅他的消息，如果别人发布了消息在这个channel key里，别人就可以监听到

如果key不存在，他发布一个消息在干什么事情，我就不想讲了

hexists anyLock 26cebeaa-e3b0-4097-8192-d62d0d0214b8:1，也就是说判断一下当前这个锁key对应的hash数据结构中，是否存在当前线程加的这个锁？释放锁的这个线程，是否之前加过锁？

否则如果当前这个线程确实对这个锁进行了加锁，此时就会做一个处理

hincrby anyLock 26cebeaa-e3b0-4097-8192-d62d0d0214b8:1，给递减1，递减完了以后的值，是一个counter，计数器，可重入锁，一个线程可以多次加锁，线程id对应的counter，加锁的次数，先给加锁的次数减1,

如果此时线程只加过一次锁，此时26cebeaa-e3b0-4097-8192-d62d0d0214b8:1对应的值本来是1，此时递减1，此时就是对应的counter = 0，就会删除这个锁key，del anyLock，删除这个锁key，释放掉这个锁，返回的值是1

如果此时线程加过多次锁，此时26cebeaa-e3b0-4097-8192-d62d0d0214b8:1对应的值是2，说明这个线程加过两次锁，可重复锁，递减1之后，还是1，大于0，pexpire anyLock 30000，刷新一下这个锁key的生存周期是30秒，返回的值是0

加锁、持续加锁、可重入加锁、锁互斥、主动释放锁、客户端宕机自动释放锁、尝试加锁超时、锁超时自动释放

才是一个企业级的可以在生产环境使用的一个分布式锁

boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);，可以指定说尝试获取一把锁，如果超过100秒获取不到的话，就自动放弃获取锁，不要永久性的阻塞；获取到锁之后，如果在10秒之内没有主动释放锁，那么就自动释放锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/30\_redis分布式锁（二十五）：读写锁源码剖析之可重入加锁/笔记.doc**

同一个客户端同一个线程，先加读锁，再加一次读锁；先加读锁，再加一次写锁；先加写锁，再加一次读锁；先加写锁，再加一次写锁

1、读锁+读锁

先加了一次读锁

KEYS[1] = anyLock

KEYS[2] = {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout

ARGV[1] = 30000毫秒

ARGV[2] = UUID\_01:threadId\_01

ARGV[3] = UUID\_01:threadId\_01:write

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

再来加一次读锁

hget anyLock mode，mode = read，此时已经加过读锁

hincrby anyLock UUID\_01:threadId\_01 1

set {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:2 1

pexpire anyLock 30000

pexpire {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:2 30000

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 2

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:2 1

2、读锁+写锁

先加了一次读锁

KEYS[1] = anyLock

KEYS[2] = {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout

ARGV[1] = 30000毫秒

ARGV[2] = UUID\_01:threadId\_01

ARGV[3] = UUID\_01:threadId\_01:write

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

再来加一次写锁

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = UUID\_01:threadId\_01:write

hget anyLock mode，mode = read，不符合条件

hexists anyLock UUID\_01:threadId\_01:write，判断一下之前是否是自己加过一次写锁

此时同一个客户端同一个线程，先读锁再写锁，是互斥的，会导致加锁失败

3、写锁+读锁

先加一次写锁

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = UUID\_01:threadId\_01:write

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 1

}

再来加一次读锁

KEYS[1] = anyLock

KEYS[2] = {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout

ARGV[1] = 30000毫秒

ARGV[2] = UUID\_01:threadId\_01

ARGV[3] = UUID\_01:threadId\_01:write

mode = write

hexists anyLock UUID\_01:threadId\_01:write，判断一下之前是否是自己加过一次写锁，此时是成立的，也就是之前加过一次写锁的，同一个客户端同一个线程

hincrby anyLock UUID\_01:threadId\_01 1，也就是说此时，加了一个读锁

set {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1,

pexpire anyLock 30000

pexpire {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 30000

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 1,

“UUID\_01:threadId\_01”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

lua脚本体现出来的逻辑，我们可以看到，如果是同一个客户端同一个线程，先加了一次写锁，然后加读锁，居然是可以加的，默认是在同一个线程写锁的期间，可以多次加读锁

4、写锁+写锁

先加一次写锁

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = UUID\_01:threadId\_01:write

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 1

}

再来加一次写锁

hexists anyLock UUID\_01:threadId\_01:write，是否存在？之前是他自己加的写锁

hincrby anyLock UUID\_01:threadId\_01:write 1

pexpire anyLock 50000

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 2

}

所以大家通过源码可以看到，同一个客户端同一个线程，多次加写锁，是可以重入加锁的

读写锁其实也是一种可重入锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/21\_redis分布式锁（十六）：MultiLock源码剖析之理念初探/笔记.doc**

redis分布式锁的两个锁，非公平可重入锁，公平可重入锁（加锁要排队）

MultiLock，redisson分布式锁这块是支持MultiLock这个机制的，可以将多个锁合并为一个大锁，对一个大锁进行统一的申请加锁以及释放锁

一次性锁定多个资源，再去处理一些事情，然后事后一次性释放所有的资源对应的锁

在项目里使用的时候，很多时候一次性要锁定多个资源，比如说锁掉一个库存，锁掉一个订单，锁掉一个积分，一次性锁掉多个资源，多个资源都不让别人随意修改，然后你再一次性更新多个资源，释放多个锁

MultiLock的源码，我们初步看一下，其实也不过是没什么特别的，就是包裹了多个RedissonLock，底层就是尝试依次对每一个锁都要成功加锁，如果所有的锁都成功加锁了之后，那么就认为MultiLock就成功加锁了

释放锁，依次去释放每一把锁就可以

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/23\_redis分布式锁（十八）：RedLock源码剖析之算法原理回顾/笔记.doc**

RedLock算法原理的回顾

这个场景是假设有一个redis cluster，有3个redis master实例

然后执行如下步骤获取一把分布式锁：

1）获取当前时间戳，单位是毫秒

2）跟上面类似，轮流尝试在每个master节点上创建锁，过期时间较短，一般就几十毫秒，在每个节点上创建锁的过程中，需要加一个超时时间，一般来说比如几十毫秒如果没有获取到锁就超时了，标识为获取锁失败

3）尝试在大多数节点上建立一个锁，比如3个节点就要求是2个节点（n / 2 +1）

4）客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了

5）要是锁建立失败了，那么就依次删除已经创建的锁

6）只要别人创建了一把分布式锁，你就得不断轮询去尝试获取锁

他这里最最核心的一个点，普通的redis分布式锁，其实是在redis集群中根据hash算法选择一台redis实例创建一个锁就可以了

RedLock算法思想，不能只在一个redis实例上创建锁，应该是在多个redis实例上创建锁，n / 2 + 1，必须在大多数redis节点上都成功创建锁，才能算这个整体的RedLock加锁成功，避免说仅仅在一个redis实例上加锁

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/04\_基于两台虚拟机搭建一套3主3从的redis cluster/笔记.doc**

（1）将redis-4.0.1.tar.gz包上传到一台机器上去，/usr/local目录下，解压缩，重命名为redis

（2）安装redis需要的一些依赖：yum install -y tcl gcc zlib-devel openssl-devel

（3）进入redis录内，执行make MALLOC=libc命令

（4）把redis的一些命令脚本拷贝到PATH变量所在的目录

cp -a src/redis-server src/redis-cli src/redis-sentinel src/redis-trib.rb src/redis-check-aof src/redis-check-rdb src/redis-benchmark /usr/local/bin/

那个/usr/local/bin，就是PATH变量的目录，这样就可以直接执行redis的一些命令了

（5）创建redis集群的目录

cd /opt

mkdir redis-cluster

mkdir redis-cluster/nodes-{7001,7002,7003}

（6）编辑redis.conf配置文件（以7001来举个例子，将7002和7003都编辑一下配置文件）

cd redis-cluster/nodes-7001

vi redis.conf

bind 127.0.0.1 192.168.9.216

port 7001

pidfile redis\_7001.pid

loglevel notice

logfile "/opt/redis-cluster/nodes-7001/redis\_7001.log"

dir /opt/redis-cluster/nodes-7001/

cluster-config-file nodes-7001.conf

daemonize yes

supervised no

appendonly yes

cluster-enabled yes

cluster-node-timeout 15000

save 900 1

save 300 10

save 60 10000

dbfilename dump.rdb

appendfilename "appendonly.aof"

appendfsync everysec

（7）在另外一台机器上，重复上面的步骤，但是弄成7001、7002、7003

（8）依次启动各个redis实例（以7001作为参考）

在两台机器上都依次执行下面的命令就可以了：

cd /usr/local/redis/src

./redis-server /opt/redis-cluster/nodes-7001/redis.conf

./redis-server /opt/redis-cluster/nodes-7002/redis.conf

./redis-server /opt/redis-cluster/nodes-7003/redis.conf

ps -ef | grep redis

因为我们现在的重点不是讲redis这个技术，其实我们是在讲基于redis的分布式锁，大家现在对redis不用太过于的纠结，我们就不去管这些东西了，后面都会详细讲解redis技术的，我们再看

这个时候看nodes-7001里面，会自动生成一些文件，包括nodes-7001.conf（自动生成的集群配置文件），appendonly.aof，redis\_7001.log，redis\_7001.pid

ps -ef | grep redis

（9）安装ruby

我们的这个centos 7操作系统默认没带ruby

ruby -v（默认是老版本ruby，卸载掉）

rpm -qa | grep ruby

yum erase ruby

将ruby-2.3.4.tar.gz上传到/usr/local下，解压缩，重命名为ruby

cd ruby

./configure

make && make install

ruby -v

按照上面的步骤，两台机器都装一下ruby

（10）安装ruby-redis.gem

（如果之前没装过，那么：yum install -y zlib-devel openssl-devel）

在/usr/local/redis下，gem install redis

如果报错：

ERROR: Loading command: install (LoadError)

cannot load such file -- zlib

ERROR: While executing gem ... (NoMethodError)

undefined method `invoke\_with\_build\_args' for nil:NilClass

cd /usr/local/ruby/ext/zlib

ruby extconf.rb

make && make install

再次gem install redis，如果再次报错：

ERROR: While executing gem ... (Gem::Exception)

Unable to require openssl, install OpenSSL and rebuild ruby (preferred) or use non-HTTPS sources

cd /usr/local/ruby/ext/openssl

ruby extconf.rb

将上个步骤生成的MakeFile文件中的${top\_srcdir}都替换为../..

make && make install

基本就是上面两个问题，都解决了之后，就可以正常的：gem install redis

（11）创建redis cluster集群

redis-trib.rb create --replicas 1 192.168.31.114:7001 192.168.31.114:7002 192.168.31.114:7003 192.168.31.184:7001 192.168.31.184:7002 192.168.31.184:7003

跟上所有的redis实例，一共是6个

redis cluster自动会将6个redis实例做成3主3从的

此时显示一直在等待，其实是因为redis.conf中的bind出了问题，不能bind 127.0.0.1

在两台机器上，都执行：ps -aux | grep redis，看到3个redis进程

kill -9 杀死所有的redis进程

对/opt/redis-cluster目录下的nodes-\*\*\*几个文件夹内的文件，都删除，保留一个redis.conf

然后修改6个redis实例的redis.conf，bind里面都去掉127.0.0.1

重启启动6个redis实例

再次执行redis-trib.rb create --replicas 1命令

（12）测试redis cluster

在/usr/local/redis/src目录下：

redis-cli -c -h 192.168.31.114 -p 7001

127.0.0.1:7001> CLUSTER info

127.0.0.1:7001> CLUSTER nodes

127.0.0.1:7001> set foo bar

127.0.0.1:7003> get foo

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/39\_zookeeper curator框架介绍以及分布式锁支持/笔记.doc**

如果你使用一个中间件的话，比如说你使用redis这个技术，那么你就必须要使用人家提供的对应的编程语言的客户端框架，就比如redis提供的就是jedis，可以用jedis对redis进行各种操作

当然也有一些非官方的知名的客户端框架，就比如说对redis而言，就是redisson，提供了一大堆的高阶的实用的一些技术，redisson + jedis，结合起来使用，甚至可以将redis当成是一个纯内存的NoSQL数据库来操作了

zookeeper分布式锁，也是一样的道理

你可以使用官方提供的zookeeper的java client api，来操作zookeeper

也有一些非官方的、开源的一些客户端框架，可以实现各种高阶的操作，比如zookeeper非官方的知名的开源的客户端框架，curator

犯了一个错误，跟大家说的zk锁的劣势，其实是没有特别好的开源的框架封装了分布式锁的支持，其实说法是不对的，过时的

在很多年以前确实是这样子的，我们的zk客户端框架都是自己封装的，基于zk的分布式锁的实现，都是自己做的，curator这种框架我们是不用的，但是我正好最近讲课，又去看了一下人家curator框架的官网，人家是支持完整的分布式锁的功能的

几乎和redisson支持的分布式锁是差不多的，各种类型的都有

颠覆掉我之前说的一个结论，redis优势是redisson锁支持很好，劣势是很多技术问题；zk优势是锁模型健壮，劣势是没有良好的封装锁的客户端框架的支持；zk锁模型很健壮，而且还有良好的客户端框架curator对各种类型的分布式锁的支持

你在公司里考虑分布式锁的技术选型的时候，我建议优先选择zk锁，redis这个技术就不要去用来做分布式系统的基础技术组件，分布式锁，队列，发布/订阅，做纯内存的kv存储，纯内存的NoSQL数据库都可以

分布式系统协调的一些事情，比如说经典的分布式锁，就不适合用redis这种技术来做，redis从最一开始设计架构的时候，就不是作为分布式协调类的基础系统来设计的，跟zk恰恰相反，zk的设计初衷就是作为分布式系统的基础技术

讲这个课之前，脑子里还残留着我们平时在公司里zk客户端是自己封装的这个事情，居然忽略了大名鼎鼎的curator框架也是支持分布式锁的，纠正一下，curator框架对分布式锁的支持也是非常的好的

除非是你的公司、或者是项目的环境，不支持用zk作为基础设施来实现分布式锁，或者是你的项目，目前引入的依赖主要就是redis，没有依赖zk，虽然你公司有zk作为基础设施，但是作为一个架构师来从全局考虑，你想要尽量减少系统对外部各种基础设施的依赖，不想依赖太多的这个东西，你只想依赖 redis就可以了

而且可以接受redis分布式锁可能存在的一些极端情况下的隐患

那么其实在项目里用redis锁也是可以的

我们平时在项目里，很多项目里，redis锁、zk锁都用，没有特别严格的一个要求

curator框架实现了大量的zookeeper的企业级使用场景，比如说分布式锁，服务注册与发现，多实例的master选举，主备切换

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/28\_redis分布式锁（二十三）：读写锁源码剖析之读锁与写锁互斥/笔记.doc**

1、先读锁后写锁如何互斥

客户端A（UUID\_01:threadId\_01）和客户端B（UUID\_02:threadId\_02），先后加了一个读锁，此时的数据如下

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 1,

“UUID\_02:threadId\_02”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

{anyLock}:UUID\_02:threadId\_02:rwlock\_timeout:1 1

客户端C（UUID\_03:threadId\_03），来加写锁

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = UUID\_03:threadId\_03:write

hget anyLock mode，mode = read，已经有人加了读锁，不是写锁，此时会直接执行：pttl anyLock，返回一个anyLock的剩余生存时间

会导致客户端C加锁失败，就会不断的尝试重试去加锁，陷入一个死循环

除非是你原先加读锁的人释放了读锁，他这个写锁才能够重构加上去

2、如果先有写锁再有人加读锁是如何互斥的

假设客户端A先加了一个写锁

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 1

}

假设此时客户端B来加读锁

KEYS[1] = anyLock

KEYS[2] = {anyLock}:UUID\_02:threadId\_02:rwlock\_timeout

ARGV[1] = 30000毫秒

ARGV[2] = UUID\_02:threadId\_02

ARGV[3] = UUID\_02:threadId\_02:write

hget anyLock mode，mode = write，已经有人加了一个写锁了

hexists anyLock UUID\_02:threadId\_02:write，存在的话，如果客户端B自己之前加过写锁的话，此时才能进入这个分支，但是不幸的是，加写锁的是客户端A，所以这里的条件会全部失败，不会成立

返回pttl anyLock，导致加锁失败，不断的陷入死循环不断的重试

导致写锁和读锁，无论如何都会是失败的，互斥的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/17\_redis分布式锁（十二）：公平锁源码剖析之排队分数刷新/笔记.doc**

客户端B和客户端C都是处于排队的状态，他们在java源码中，会进入while ture循环，每次等待一段时间之后，就会重新来尝试进行加锁，客户端B和客户端C，模拟他们等待一段时间之后，来尝试进行重新加锁

客户端B，10:00:00，10:00:15，他来再次尝试进行加锁，但是客户端A此时还是持有着这把锁的，此时会出现什么情况呢？

while true，获取队列第一个元素，同时获取这个元素的timeout时间，10:00:25 <= 10:00:15？不成立，while死循环退出

判断一下是否没人加锁，以及判断是否是客户端B进行的加锁，都不成立

ttl = 10:00:25 - 10:00:15 = 10000毫秒

timeout =10000毫秒 + 10:00:15 + 5000毫秒 = 10:00:30

zadd redisson\_lock\_timeout:{anyLock} 10:00:30 UUID\_02:threadId\_02，刷新一下有序集合中的元素的分数，10:00:30

大致可以理解为，你的某个客户端不是在不断的重试尝试加锁么？每次重试尝试加锁的时候，就判定为是这个客户端一次新的尝试加锁的行为，此时会将这个客户端对应的timeout时间往后延长，10:00:25，这次他重试加锁之后，timeout时间算出来就变成10:00:30，把这个客户端的timeout时间延长了，有序集合中的分数变大了

zadd指令的返回值，如果是第一次往里面怼入一个元素，返回值是什么？如果是第二次刷新这个元素的分数，返回值是什么？如果是第一次插入一个元素在有序集合中，此时就会返回值是1，同时会入队；但是后续不断的尝试加锁的时候，其实是会不断的刷新这个有序集合中的元素的分数，越来越大，但是每次刷新分数，返回值是0

所以不会重复的往队列中插入这个元素的

客户端C，10:00:20，再次尝试加锁

10:00:30 < 10:00:20？不成立；尝试加锁，都不成立

再次刷新他的timeout分数

ttl = 10:00:30 - 10:00:20 = 10000毫秒

timeout = 10000毫秒 + 10:00:20 + 5000毫秒 = 10:00:35

zadd redisson\_lock\_timeout:{anyLock} 10:00:35 UUID\_03:threadId\_03

还是维持着跟之前一样的这么一个顺序，客户端B的分数会刷大，客户端C的分数其实也在不断的刷大，所以在有序集合中的分数的大小的顺序，基本上还是按照的是最初他们是如何排队的，此时也会如何排队

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/32\_redis分布式锁（二十七）：读写锁源码剖析之释放写锁/笔记.doc**

同一个客户端多次可重入加写锁 / 同一个客户端先加写锁再加读锁

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01:write”: 2,

“UUID\_01:threadId\_01”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

释放写锁

KEYS[1] = anyLock

KEYS[2] = redisson\_rwlock:{anyLock}

ARGV[1] = 0

ARGV[2] = 30000

ARGV[3] = UUID\_01:threadId\_01:write

mode = write

hdel anyLock UUID\_01:threadId\_01:write

anyLock: {

“mode”: “write”,

“UUID\_01:threadId\_01”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

hset anyLock mode read，将写锁转换为读锁

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/08\_redis分布式锁（三）：可重入锁源码剖析之watchdog维持加锁/笔记.doc**

如果你的某个客户端上锁了之后，一直比如过了5分钟，10分钟，都没释放掉这个锁，那么你觉得会怎么样呢？锁对应的key刚开始的生存周期其实就是30秒而已，难道是默认情况下30秒后这个锁就自动释放？？？

lock watchdog，你对anyLock这个key加锁了以后，然后后台有一个定时调度的一个watchdog任务，这个任务的话呢其实就是每隔一定时间，10秒左右，延长一下anyLock这个key的生存周期，重新延长到30秒

只要你的这个lock没有释放掉，那么这个后台的watchdog他就会不断的去重复延长这个key的生存周期，对应的就是你的客户端长期的5分钟，10分钟，30分钟，一直持有了某一个key对应的分布式锁

1、首先从源码的层面，带着大家来看一下，如果你成功的对某个key加锁了之后，后台的定时调度任务（lock watchdog）是如何每隔10秒钟去延长一下那个key的生存时间的？

RFuture<Long> ttlRemainingFuture = tryLockInnerAsync()

这里的RFuture，ttlRemainingFuture，这里封装的其实是当前这个key对应的一个剩余的存活时间，单位是毫秒，pttl anyLock这个命令所返回的

ttlRemainingFuture.addListener(new FutureListener<Long>() {

@Override

public void operationComplete(Future<Long> future) throws Exception {

if (!future.isSuccess()) {

return;

}

Long ttlRemaining = future.getNow();

// lock acquired

if (ttlRemaining == null) {

scheduleExpirationRenewal(threadId);

}

}

});

这里给那个RFuture加了一个监听器，也就是说只要这个lua脚本执行完成，返回了pttl anyLock那个指令返回的一个剩余存活的时间之后，这个RFuture的监听器就会被触发执行的

if (!future.isSuccess()) {

return;

}

如果那段加锁的lua脚本执行失败的话，那么这里就不是success，相当于是基于redis加锁失败了，正常情况下，ttlRemaining，也就是pttl那个指令返回的值，在这里呢其实是一个null，很多同学可能会搞不清楚，说为什么这里是ttlRemaining必须是null呢？

这里面底层牵扯到一些lua脚本的执行，大家不要过多的关注这个细节

// lock acquired

if (ttlRemaining == null) {

scheduleExpirationRenewal(threadId);

}

如果ttlRemaining是null的话，就说明，锁已经获取了，lock acquired，如果锁是成功获取的话，接下来他就会执行一个逻辑，也就是后台开启一个定时调度的任务，只要这个锁还被客户端持有着，那么就不会不断的去延长那个key的生存周期

scheduleExpirationRenewal(threadId);

执行这行代码，后台定时调度任务不断的延长key的生存周期，只要客户端还持有那把锁就可以了

Timeout task = commandExecutor.getConnectionManager().newTimeout(new TimerTask() {

@Override

public void run(Timeout timeout) throws Exception {

RFuture<Boolean> future = renewExpirationAsync(threadId);

future.addListener(new FutureListener<Boolean>() {

@Override

public void operationComplete(Future<Boolean> future) throws Exception {

expirationRenewalMap.remove(getEntryName());

if (!future.isSuccess()) {

log.error("Can't update lock " + getName() + " expiration", future.cause());

return;

}

if (future.getNow()) {

// reschedule itself

scheduleExpirationRenewal(threadId);

}

}

});

}

}, internalLockLeaseTime / 3, TimeUnit.MILLISECONDS);

internalLockLeaseTime，默认是30000毫秒，除以3之后，就是10000毫秒，也就是10秒左右，当你成功加锁之后，这边一定会走这段代码，开启定时调度的任务，初次执行是在10秒以后

RFuture<Boolean> future = renewExpirationAsync(threadId);

这行代码，其实renewExpirationAsync()方法，更新过期时间

"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return 1; " +

"end; " +

"return 0;",

hexists anyLock 8e6b27a7-5346-483a-b9b5-0957c690c27f:1，执行这行指令，看下anyLock这个map中，是否存在8e6b27a7-5346-483a-b9b5-0957c690c27f:1这个key

如果存在的话，那么就说明人家锁还是持有的，还没过期，此时就需要去执行一行命令：pexpire anyLock 30000，也就是说将anyLock的生存时间重新设置为30000毫秒，也就是30秒

也就是说此时anyLock这个key已经存在了超过10秒了，30秒，生存时间就剩下20秒了

但是因为他里面的那个8e6b27a7-5346-483a-b9b5-0957c690c27f:1还存在，说明客户端还持有着这把锁，所以他就会延长一下生存时间，anyLock这个key的生存时间重新变为30秒，再次开始

只要你的anyLock这个锁还被当前的这个客户端的这个线程持有了锁，redis里的那个数据还存在，那么就靠这个定时调度的任务，就可以不断的刷新anyLock的生存时间，保证说，你的客户端只要一直持有这把锁，那么他对应的redis里的key，也会一直保持存在，不会过期的

2、然后我们程序模拟出来长期持有一把锁，但是不释放，在redis-cli命令行上看一下那个锁对应的key的生存时间的变化

通过观察后台的那个锁对应的key的生存时间都知道，源码的执行逻辑其实是完全正确的，就是说那个key的生存时间不断的减少，但是每次减少了10秒，到剩余20秒生存周期的时候，就会由代码里重新调度一个任务，刷新一下这个key的生存时间，重新变为30秒

3、如果持有锁的那台机器宕机了呢？

其实很简单，那台机器如果宕机了以后，就会导致那台机器上的lock wathdog，就是那个每隔10秒执行一次的定时任务，那个任务就不会执行了，不执行以后，那个anyLock那个锁的key自动就会在30秒以内自动过期，释放掉这把锁

此时其他的客户端最多就是再等待30秒就可以获取到这把锁了

4、我们看一下如果释放锁之后，那么后台的定时调度的任务是如何自动取消的，就不需要你后台的watchdog程序不断的延长那个key的声明周期了

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/05\_redis开源客户端框架redisson的初步介绍以及使用/笔记.doc**

redisson，说白了就是一个功能非常强大的开源的redis客户端框架，我们其实就是看一下他的官网的介绍，大概就知道他是干嘛的了

然后这里就是先做一个demo工程出来，基于spring boot技术，就最简单的就ok了，整合redisson进去，然后用他的一些分布式锁的功能，看下源码

建一个普通的工程

在pom.xml里引入依赖

<dependency>

<groupId>org.redisson</groupId>

<artifactId>redisson</artifactId>

<version>3.8.1</version>

</dependency>

（3）参照官网构建RedissonClient，同时看看对应的配置

Config config = new Config();

config.useClusterServers()

.addNodeAddress("redis://192.168.31.114:7001")

.addNodeAddress("redis://192.168.31.114:7002")

.addNodeAddress("redis://192.168.31.114:7003")

.addNodeAddress("redis://192.168.31.184:7001")

.addNodeAddress("redis://192.168.31.184:7002")

.addNodeAddress("redis://192.168.31.184:7003");

RedissonClient redisson = Redisson.create(config);

（4）简单用一下分布式锁的功能

RLock lock = redisson.getLock("anyLock");

lock.lock();

lock.unlock();

RMap<String, Object> map = redisson.getMap("anyMap");

map.put("foo", "bar");

map = redisson.getMap("anyMap");

System.out.println(map.get("foo"));

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/25\_redis分布式锁（二十）：读写锁源码剖析之加读锁的逻辑/笔记.doc**

读写锁的意义

没什么可画图的，redis分布式锁，主要就是在理解他里面的lua脚本的逻辑，逻辑全部都在lua脚本里，你只能枚举清楚各种情况下，lua脚本会执行什么东西？你其实就知道了这个分布式锁的实现原理

读写锁

多个客户端同时加读锁，是不会互斥的，多个客户端可以同时加这个读锁，读锁和读锁是不互斥的

如果有人加了读锁，此时就不能加写锁，任何人都不能加写锁了，读锁和写锁是互斥的

如果有人加了写锁，此时任何人都不能加写锁了，写锁和写锁也是互斥的

RedissonReadLock是RedissonLock的子类

关注要几块东西，第一个是加读锁的lua脚本的逻辑；第二个是读锁的释放的lua脚本的逻辑；第三个是读锁的wathdog刷新锁key的生存时间的逻辑

研究分布式锁的源码，一般来说不需要画图，就是之前最初的那个图就够了，后面的锁都是基于之前的最早最基础的那个锁的技术框架来实现的，后面的直接分析lua脚本

客户端A（UUID\_01:threadId\_01）来加读锁

KEYS[1] = anyLock

KEYS[2] = {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout

ARGV[1] = 30000毫秒

ARGV[2] = UUID\_01:threadId\_01

ARGV[3] = UUID\_01:threadId\_01:write

anyLock: {

“mode”: “xxxx”

}

hget anyLock mode，相当于是从anyLock这个hash里面获取一个mode作为key的值。但是此时刚开始你都没有加锁呢，所以这里的mode肯定是空的，此时就是直接加一个读锁

hset anyLock mode read

hset anyLock UUID\_01:threadId\_01 1

set {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

pexpire {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 30000

pexpire anyLock 30000

anyLock: {

“mode”: “read”,

“UUID\_01:threadId\_01”: 1

}

{anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 1

技术框架，都是基于RedissonLock去实现的，包括各种特性，各种不同的锁的逻辑都封装在lua脚本里了

开启一个watchdog看门狗，会每隔10秒钟去执行一段lua脚本，判断一下当前这个线程是否还持有着这个锁，如果还持有锁，更新一下锁key的生存时间为30000毫秒，保持redis的锁key和java代码中持有的锁是保持同步的

读锁的watchdog的逻辑

读锁，加锁、watchdog的逻辑一次性都给大家讲完得了

客户端A已经成功加了一个读锁了，此时watchdog运转的逻辑

KEYS[1] = anyLock

KEYS[2] = {anyLock}

ARGV[1] = 30000毫秒

ARGV[2] = UUID\_01:threadId\_01

hget anyLock UUID\_01:threadId\_01，获取一下当前这个线程是否对这个锁加了一个读锁，这里返回的应该是1，此时可以判定是当前这个线程加的读锁

pexpire anyLock 30000，刷新一下anyLock锁key的生存时间为30000毫秒

hlen anyLock = 2 > 1，就是说，如果你的读锁，anyLock hash内部的key-value对超过了1个，这里肯定是成立的

for i = 1, 1, -1 do

加读锁的时候，其实是每个线程都可以加多次这个读锁，读锁也是可重入的，每次同一个线程加多次读锁的时候，他的加锁次数就会加1，counter = 1 / 10 /20

就是遍历counter -> 1，每次递减1，假设counter = 10，10,9,8,7,6,5,4,3,2,1,

pexpire {anyLock}:UUID\_01:threadId\_01:rwlock\_timeout:1 30000

watchdog在刷新生存周期的时候，一方面是说，如果这个线程对这个key还加着锁的话，那么此时就会刷新锁key的生存周期，anyLock，30000毫秒；同时还会遍历加锁次数，对那个锁key的每次加锁对应的一个timeout key也把生存周期刷新为30000毫秒

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/16\_redis分布式锁（十一）：公平锁源码剖析之可重入加锁/笔记.doc**

分析一下，在公平锁的这个加锁机制下，客户端A如果此时重复进行lock.lock()，是如何进行可重入加锁的呢？

while true，lindex redisson\_lock\_queue:{anyLock} 0，获取队列第一个元素，判断一下他的timeout时间是否大于当前时间了，假设当前时间是10:00:08，此时肯定是是不成立的，假设就直接跳出死循环

当前锁是否不存在，不成立；hexists anyLock UUID\_01:threadId\_01，在anyLock hash中，是否存在一个UUID\_01:threadId\_02的key，当前这个锁是否是客户端A加锁，成立，hincrby给加锁次数累加1，重新pexpire刷新锁key的生存时间为30000毫秒，返回一个nil

此时可重入加锁成功，无非就是累加了1次加锁的次数

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/02\_再次梳理一下redis分布式锁的问题以及使用建议/笔记.doc**

redis分布式锁的问题

第一种方案，基于redis单实例，不靠谱，因为redis单点故障，会导致系统全盘崩溃，做不到高可用，除非你是那种不太核心的小系统，随便用一下分布式锁，你可以这么弄，应该还好，毕竟对高可用要求没那么高

第二种方案，基于redis主从架构+哨兵，保证高可用，master宕机，slave接替，但是有隐患，master宕机的一瞬间，还没异步复制锁到slave，导致重复加锁的问题，高可用是高可用了，但是锁的实现有漏洞，可能导致系统异常

客户端刚在master实例加了一个锁，但是master->slave的复制数据（锁复制过去）是异步的，导致master突然宕机，此时锁还没复制到slave，然后master->slave主备切换（哨兵），客户端B此时也对同一个key上锁，此时就会成功的在切换为master的slave实例上加锁，客户端A和客户端B同时对一个key完成了上锁

只要多个客户端同时对某个key上锁，就会导致数据肯定会出错

第三种方案，基于redis多master集群（redis-cluster，或者redis部署多机器，twitter开源的twemproxy做客户端集群分片，豌豆荚开源的codis做集群分片，都行），redlock算法，个人不推荐，实现过程太复杂繁琐，很脆弱，多节点同时设置分布式锁，但是失效时间都不一样，随着不同的linux机器的时间不同步，以及各种你无法考虑到的问题，很可能出现重复加锁

举个例子，给5个redis master都设置了一把锁key，失效时间是10s，但是因为设置key的时间可能因为网络等各种情况不同，会导致，也许客户端A还加着锁呢，因为处理耗时特别长，然后结果过了几秒钟，其中3台机器的锁key都过期失效了，因为这3台机器加锁的时间都比较靠前，已经10秒过期了，此时分布式锁自然失效，客户端B成功加锁，出现两个客户端同时加锁的问题

所以redlock算法，有两个问题，第一是实现过程和步骤太复杂，上锁的过程和机制很重，很复杂，导致很脆弱，各种意想不到的情况都可能发生；第二是并不健壮，不一定能完全实现健壮的分布式锁的语义

http://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

这是国外一个牛人写的redlock算法的分析小论文，大家可以看一下，里面的结论就是我时候的那两个，第一是太复杂，第二是不健壮可能加锁会失败

当然redis作者不赞同他，但是同样redis作者自己也没法证明他的算法是健壮的，所以这个方案不建议大家采用

因为redis RedLock分布式锁的算法，涉及过程比较复杂，涉及到很多不同情况下的时间问题，就会导致说，你如果要证明这个算法是ok的，到最后其实会成为一个数学问题，你要从数学和逻辑的角度，大量的推算，证明说，这个算法，无论在任何极端的情况下都是成功的

综上所述，redis分布式锁实际上目前没有100%完美的方案，或多或少有点问题，实际生产系统中，我设计的系统有时候用zk分布式锁，但是有的时候也会用redis分布式锁。

redis锁又有一个优点，redisson，知名的、优秀的开源的redis客户端类库，他封装了大量的基于redis的复杂的一些操作，数据集合（map、set、list）的分布式的存储，还有多种复杂的分布式锁，分布式执行操作以及对象，甚至就是说基于redis+redisson，甚至都可以将redis作为一个轻量级的NoSQL数据库、数据存储来使用

redisson：redis分布式锁的支持，非常不错，可重入锁、读写锁、公平锁、信号量、CountDownLatch，很多种复杂的锁的语义，可以支持我们将分布式锁玩儿的非常的好

但是zk分布式锁呢？其实有优点也有缺点，优点是锁模型健壮、稳定、可用性高，缺点是目前没有太好的开源的zk分布式锁的类库，封装多种不同的锁类型，因为有可重入锁、读写锁、公平锁等多种锁类型，但是zk的话，目前常见的方案，就是自己动手封装一个基于顺序节点的普通的悲观锁，没有复杂的锁机制的支持

你对redis分布式锁的使用，用是可以用的，但是你要知道他的优势和劣势，如果你可以容忍一些劣势，以及做一些对应的措施和预案，后台可以容忍做补数据的操作，而且你对分布式锁的需求很旺盛，需要人家各种高级分布式锁的一些支持

但是如果有zookeeper的环境，而且你的分布式锁的需求很简单，就是普通的悲观锁模型就可以了，不涉及到什么公平锁、读写锁之类的东西，那么可以考虑基于zk的分布式锁模型，健壮，稳定，zk的临时顺序节点实现的分布式锁，其实那套模型挺健壮的，zk本身就是集群多节点分布式高可用，分布式系统协调、处理的，支持分布式锁的时候，基于zk的一些语义，监听节点的变化

zk锁，优点就是健壮、稳定、简单、易懂；缺点就是没有太好的开源的客户端类库，对zk分布式锁的封装和支持，可以支持那么多种不同的高阶的锁类型，公平锁、可重入锁、读写锁、信号量、CountDownLatch，Curator客户端主要还是针对zk的一些基础的语义做的一些封装，redis的Jedis + Redisson结合起来，Jedis封装redis的一些基础的语义，一些操作，都是不错的

目前行业里，基本redis和zk做分布式锁，都有，具体看架构师自己对这个东西怎么理解和考量了，redis锁没有特别好的100%完美和健壮的锁模型，或多或少会导致一些问题，如果你在一些业务场景下，能容忍这些问题，同时需要redission的复杂锁类型的支持，可以用redis锁；但是如果你的业务场景要求锁的语义绝对没问题，很健壮，很稳定，绝对不会导致说多个客户端同时加到一把锁，100%不会发生，而且对锁的功能没有特别的需求，那么可以用zk锁

但是我说下我个人的角度，和经验，以及对待技术的一些态度，来理解的。设计架构为什么偏好用zk分布式锁，原因两个，第一是redis锁的算法模型目前业内方案都有隐患，心里不安，zk锁的机制更加健壮和稳定；第二是我设计架构，喜欢考虑到技术的本质问题，redis本质是什么？其实说白了，还是缓存！redis现在越来越往什么可以实现队列的功能、分布式锁的功能、发布订阅的功能，如果我是redis作者，我个人觉得本末倒置了

要敢于质疑一下那些技术权威，如果我是作者开源的发起人，我会回归他的本质，他的本质是一个kv类的缓存，如果要发展，应该是纵向发展，比如支持磁盘存储，做成可以支持磁盘+内存的大规模，海量数据的kv存储，这是他可以的发展方向

而且作为开源项目，rabbitmq，就考虑到一些集群的管理和运维的一些操作，提供了漂亮的可视化的界面，让人可以管理。

完全可以提供更加方便的全可视化的界面管理工作台，方便运维和管理，包括一键部署集群，集群上下线节点，不同的集群模式的一键转换（单实例模式 -> 主从模式 -> 哨兵模式 -> 集群模式），数据迁移，数据备份，一键集群版本滚动升级，子集群模式与业务隔离，热key大value的自动发现与报警，集群资源的监控与报警，多机房集群部署容灾，集群访问量监控与扩容预警，等等吧。

这些东西，现在有一个CacheCloud（搜狐开源的，自己弄的）在搞，还有各大公司，都是基础架构团队自己在搞

以我设计大量的系统的经验而言，我真的觉得redis还支持什么lua脚本，但是做一个轻量级的支持就可以了，大部分情况下肯定都是不需要使用这么重的操作的，hyperloglog

但是让我失望的是，redis cluster，居然刚搞出来的时候，还不支持天然的读写分离，slave纯就是做高可用自动切换的，简直是在搞笑，而且极其繁琐的运维，还搞什么乱七八糟的RedLock分布式锁算法，队列支持，发布订阅支持，我觉得简直是在瞎搞

redis天然就不是为了分布式锁这种分布式系统的基础组件来设计的，zk才是最适合做各种分布式系统的基础设施依赖的，而且业界基本各大开源项目，都是依赖的zk做各种分布式系统的基础设施

所以国内的工程师们，要开始有点自己的技术判断力了，不要人云亦云，老外做什么就认可什么

下一讲我们还是给大家来演示一下redis的分布式锁的使用，其实一般当然不会自己傻呵呵写一堆代码去实现redis分布式锁的客户端代码了，那太繁琐了，一般就是用redisson这个客户端框架，他有完整的一套redis分布式锁的实现

我们出于技术完整性的考虑，还是给大家仔细说一说，先是部署几台虚拟机，然后部署个redis集群，使用redis cluster就好了，然后基于目前比较好的redis开源客户端框架，redisson来做一个分布式锁的学习，重点是读一下人家客户端框架的源码

毕竟在行业里，redis分布式锁其实用的也是很多的

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/40\_基于3台linux虚拟机搭建一个基本的zookeeper集群/笔记.doc**

1、准备5台linux虚拟机

都是CentOS 7操作系统，64位，每台机器都是1个cpu，1G内存，安装JDK 1.8

（1）使用课程提供的CentOS 7镜像即可，CentOS-7-x86\_64-Minimal-1611.iso。

（2）创建虚拟机：打开Virtual Box，点击“新建”按钮，点击“下一步”，输入虚拟机名称为hadoop01，选择操作系统为Linux，选择版本为Red Hat-64bit，分配1024MB内存，后面的选项全部用默认，在Virtual Disk File location and size中，一定要自己选择一个目录来存放虚拟机文件，最后点击“create”按钮，开始创建虚拟机。

（3）设置虚拟机网卡：选择创建好的虚拟机，点击“设置”按钮，在网络一栏中，连接方式中，选择“Bridged Adapter”。（搭建虚拟机还真不是我专门要讲的一块内容，我一般常用的搭建虚拟机做实验的方式就是桥接网络）

如果你的笔记本电脑在家里用wifi的ip地址配置了一下虚拟机的网络，然后你的电脑拿到外面咖啡馆里去，或者公司里去，ip地址变化了之后，就会导致你的之前搭建的那套虚拟机环境就没法用了

如果你希望无论到哪里，ip地址变化了以后，虚拟机都可以正常工作，不需要重新配置网络，那么你就去百度一下，虚拟机如何配置网络，可以更换环境的时候，不需要重新配置，NAT如何配置

桥接，换了一个网络环境，ip地址都换了，5台虚拟机重新配置一下里面的网卡的一些ip地址什么的，就ok了

（4）安装虚拟机中的CentOS 7操作系统：选择创建好的虚拟机，点击“开始”按钮，选择安装介质（即本地的CentOS 7镜像文件），按照课程选择后自动安装即可

（5）安装完以后，CentOS会提醒你要重启一下，就是reboot，你就reboot就可以了。

（6）配置网络

（如果重启网卡没看到人家给你分配的ip地址，你自己应该是要去连一个路由器就可以了）

vi /etc/sysconfig/network-scripts/ifcfg-enp0s3

先让它动态分配一个ip地址

ONBOOT=yes

service network restart

ip addr

再设置静态ip地址

BOOTPROTO=static

IPADDR=192.168.31.250

NETMASK=255.255.255.0

GATEWAY=192.168.31.1

service network restart

ip addr

配置DNS

检查NetManager的状态：systemctl status NetworkManager.service

检查NetManager管理的网络接口：nmcli dev status

检查NetManager管理的网络连接：nmcli connection show

设置dns：nmcli con mod enp0s3 ipv4.dns "114.114.114.114 8.8.8.8"

让dns配置生效：nmcli con up enp0s3

（7）配置hosts

vi /etc/hosts

配置本机的hostname到ip地址的映射

（8）配置SecureCRT

此时就可以使用SecureCRT从本机连接到虚拟机进行操作了

一般来说，虚拟机管理软件，virtual box，可以用来创建和管理虚拟机，但是一般不会直接在virtualbox里面去操作，因为比较麻烦，没有办法复制粘贴

SecureCRT，在windows宿主机中，去连接virtual box中的虚拟机

收费的，我这里有完美破解版，跟着课程一起给大家，破解

（9）关闭防火墙

systemctl stop firewalld.service

systemctl disable firewalld.service

关闭windows的防火墙

后面要搭建集群，有的大数据技术的集群之间，在本地你给了防火墙的话，可能会没有办法互相连接，会导致搭建失败

（10）配置yum

yum clean all

yum makecache

yum install -y wget

（11）安装JDK

1、将jdk-8u131-linux-x64.rpm通过WinSCP上传到虚拟机中

2、安装JDK：rpm -ivh jdk-8u131-linux-x64.rpm

3、配置jdk相关的环境变量

vi ~/ .bashrc

export JAVA\_HOME=/usr/java/latest

export PATH=$PATH:$JAVA\_HOME/bin

source .bashrc

4、测试jdk安装是否成功：java -version

（12）在另外4个虚拟机中安装CentOS集群

按照上述步骤，再安装三台一模一样环境的linux机器

另外三台机器的hostname分别设置为hadoop02，hadoop03，hadoop04，hadoop05

安装好之后，在每台机器的hosts文件里面，配置好所有的机器的ip地址到hostname的映射关系

比如说，在elasticsearch01的hosts里面

192.168.31.250 hadoop0101

192.168.31.xxx hadoop02

192.168.31.xxx hadoop03

192.168.31.xxx hadoop04

（13）配置5台CentOS为ssh免密码互相通信

首先在三台机器上配置对本机的ssh免密码登录

ssh-keygen -t rsa，生成本机的公钥，过程中不断敲回车即可，ssh-keygen命令默认会将公钥放在/root/.ssh目录下

cd /root/.ssh

cp id\_rsa.pub authorized\_keys

将公钥复制为authorized\_keys文件，此时使用ssh连接本机就不需要输入密码了

接着配置三台机器互相之间的ssh免密码登录

使用ssh-copy-id -i hostname命令将本机的公钥拷贝到指定机器的authorized\_keys文件中

2、hdfs HA集群部署规划

namenode，就相当于是在eclipse中新建一个工程，取个名字叫做namenode，在工程里面开发代码，写好代码之后，就可以打包，在linux服务器上，用java -jar给他启动，namenode就是这个意思

namenode就是一个java工程，写完里面的代码，打包，在linux上部署，java -jar启动，出来一个进程，namenode其实就是一个系统，启动的时候就是一个进程

hadoop01：active namenode，ZKFC

hadoop02：standby namenode，ZKFC

hadoop03：datanode，journal node，QuorumPeerMain

hadoop04：datanode，journal node，QuorumPeerMain

hadoop05：datanode，journal node，QuorumPeerMain

3、部署zookeeper集群

在hadoop03、hadoop04、hadoop05三台机器上部署zookeeper

tar -zxvf zookeeper-3.4.9.tar.gz

mv zookeeper-3.4.9 zookeeper

vi ~/.bashrc

export ZOOKEEPER\_HOME=/usr/local/zookeeper ?export PATH=$PATH:$ZOOKEEPER\_HOME/bin

source ~/.bashrc

vi zoo.cfg （$ZOOKEEPER\_HOME/conf）

dataDir=/home/data/zookeeper ?dataLogDir=/home/log/zookeeper?server.1=hadoop03:2888:3888

server.2=hadoop04:2888:3888 ?server.3=hadoop05:2888:3888

mkdir -p /home/data/zookeeper

mkdir -p /home/log/zookeeper

cd /home/data/zookeeper

echo 1 > myid

scp -r /usr/local/zookeeper hadoop04:/usr/local ，修改环境变量，在hadoop04将myid的内容改为2 （echo 2 > myid）

scp -r /usr/local/zookeeper hadoop05:/usr/local，修改环境变量，在hadoop05将myid的内容改为3 （echo 3 > myid）

三台机器上执行：zkServer.sh start

查看集群状态 ?1、jps（查看进程） ?2、zkServer.sh status（查看集群状态，主从信息）

4、部署hadoop集群

先在hadoop01上操作

tar -zxf hadoop-2.9.1.tar.gz -C /home/apps/

vim /etc/profile

export JAVA\_HOME=/home/apps/jdk1.8.0\_111

export HADOOP\_HOME=/home/apps/hadoop-2.9.1

export PATH=$PATH:$JAVA\_HOME/bin:$HADOOP\_HOME/bin

source /etc/profile

vi $HADOOP\_HOME/etc/hadoop/hadoop-env.sh

export JAVA\_HOME=/home/apps/jdk1.8.0\_111

vi core-site.xml

<configuration>

<property>

<name>fs.defaultFS</name>

<value>hdfs://ns1/</value>

</property>

<property>

<name>hadoop.tmp.dir</name>

<value>/home/apps/hadoop-2.9.1/tmp</value>

</property>

<property>

<name>ha.zookeeper.quorum</name> <value>hadoop01:2181,hadoop02:2181,hadoop03:2181</value>

</property>

</configuration>

<configuration>

<property>

<name>dfs.nameservices</name>

<value>ns1</value>

</property>

<property>

<name>dfs.ha.namenodes.ns1</name>

<value>nn1,nn2</value>

</property>

<property>

<name>dfs.namenode.rpc-address.ns1.nn1</name>

<value>hadoop01:9000</value>

</property>

<property>

<name>dfs.namenode.http-address.ns1.nn1</name>

<value>hadoop01:50070</value>

</property>

<property>

<name>dfs.namenode.rpc-address.ns1.nn2</name>

<value>hadoop02:9000</value>

</property>

<property>

<name>dfs.namenode.http-address.ns1.nn2</name>

<value>hadoop02:50070</value>

</property>

<property>

<name>dfs.namenode.shared.edits.dir</name>

<value>qjournal://hadoop03:8485;hadoop04:8485;hadoop05:8485/ns1</value>

</property>

<property>

<name>dfs.journalnode.edits.dir</name>

<value>/home/apps/hadoop-2.9.1/journaldata</value>

</property>

<property>

<name>dfs.ha.automatic-failover.enabled</name>

<value>true</value>

</property>

<property>

<name>dfs.client.failover.proxy.provider.ns1</name>

<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>

</property>

<property>

<name>dfs.ha.fencing.methods</name>

<value>

sshfence

shell(/bin/true)

</value>

</property>

<property>

<name>dfs.ha.fencing.ssh.private-key-files</name>

<value>/root/.ssh/id\_rsa</value>

</property>

<property>

<name>dfs.ha.fencing.ssh.connect-timeout</name>

<value>30000</value>

</property>

</configuration>

vi slaves

hadoop03

hadoop04

hadoop05

上面其实就已经将active namenode给配置好了

然后将这个hadoop拷贝到其他4台机器上去：scp -r /home/apps/ hadoop02:/home/apps/

5、启动hadoop集群

（1）在hadoop03、hadoop04、hadoop05上启动journal nodes集群：

sbin/hadoop-daemon.sh start journalnode

（2）格式化namenode（hadoop01上执行）

hdfs namenode -format

hdfs namenode -bootstrapStandby

（3）格式化ZKFC（hadoop01上执行）

hdfs zkfc -formatZK

（4）启动hdfs集群（hadoop01上执行）

sbin/start-dfs.sh，这个脚本会自动在hadoop01和hadoop02上启动一个namenode，同时启动一个ZKFC，然后会自动在hadoop03、hadoop04、hadoop05上分别启动一个datanode

6、查看hadoop集群

在hadoop01和hadoop02访问50070端口即可

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/45\_zookeeper分布式锁（三）：可重入读写锁源码剖析/笔记.doc**

（1）读锁+读锁

firstWriteIndex：2147483647，Integer.MAX\_VALUE

加读锁的逻辑非常简单，如果N多个客户端同时加读锁，有没有问题？一定没问题，他这里每次去加一个读锁的时候

都是在/locks/lock\_01目录下创建一个顺序节点

然后的话呢，获取一个顺序节点在/locks/locks\_01目录下的位置，索引，判断这个位置只要< Integer.MAX\_VALUE这个值就可以了

2147483647客户端同时加读锁都是没问题的

N多个客户端同时加读锁，肯定是不会互斥的，满足读写锁的逻辑

（2）读锁+写锁

/locks/lock\_01目录下，此时已经有了一个顺序节点，有了N个读锁的顺序节点

/locks/lock\_01/\_c\_0548a389-3307-4134-9551-088d305b86c7-\_\_READ\_\_0000000003

就代表了说一个客户端加了读锁

/locks/lock\_01/\_c\_73b60882-9361-4fb7-8420-a8d4911d2c99-\_\_WRIT\_\_0000000005

加写锁的时候，上来会不分青红皂白，直接在/locks/locks\_01目录下创建一个\_\_WRITE\_\_的写锁的顺序节点

[\_c\_13bf63d6-43f3-4c2f-ba98-07a641d351f2-\_\_READ\_\_0000000004, \_c\_73b60882-9361-4fb7-8420-a8d4911d2c99-\_\_WRIT\_\_0000000005]

此时/locks/lock\_01目录下，之前已经有人加过一个读锁了，此时又往里面写了一个写锁的顺序节点

写锁节点在children里是排在第二位的，index是1

写锁的maxLeases是1，所以说如果你要加一个写锁成功的话，你必须是在/locks/lock\_01目录里，是处于第一个位置的，index = 0，才能小于maxLeases，写锁才能够加成功

但是此时children中，第一个的是别人加的读锁，所以此时你的写锁一定是失败的

index = 1 < maxLeases = 1，条件是不成立的

他会给他的前一个节点加一个监听器，如果前面一个节点释放了锁，他就会被唤醒，再次尝试判断，他是不是处于处于当前这个children列表中的第一个，如果是第一个的话，才能是加写锁成功

除非是前面加的读锁先释放掉了，写锁才能被唤醒然后尝试加写锁成功

（3）写锁+读锁

/locks/lock\_01/\_c\_d4842035-5ba2-488f-93a4-f85fafd5cc32-\_\_READ\_\_0000000007

[\_c\_39258b5d-4383-466e-9b86-fda522ea061a-\_\_WRIT\_\_0000000006, \_c\_d4842035-5ba2-488f-93a4-f85fafd5cc32-\_\_READ\_\_0000000007]

如果是同一个客户端，先加写锁，再加读锁，是可以成功的，同一个客户端的写锁+读锁是不互斥的

如果是不同的客户端先加写锁，再加读锁呢？

if ( node.contains(WRITE\_LOCK\_NAME) )

{

firstWriteIndex = Math.min(index, firstWriteIndex);

}

遍历children列表，如果在children列表中发现了一个写锁，此时会干什么呢？

firstWriteIndex = 0

ourIndex = 1

boolean getTheLock = ourIndex（1） < firstWriteIndex（0）

这样的话呢，只要你是先加了一个写锁，写锁之后要加读锁的话，此时都会一律不让加读锁，会卡住，加锁就失败了，wait等待

不同的客户端，如果先加写锁，再加读锁，一定是互斥的

只有等到前面的写锁释放掉了，这个读锁他才能加锁成功，后面一大堆的人跟着加了N个读锁，也是一样的排队等待

（4）写锁+写锁

如果有一个人先加了写锁，然后后面又有一个人来加了这个写锁，此时会发现第二个写锁的node是第二位，不是第一位，所以会导致写锁也会等待，加锁失败，只有第一个写锁先成功了，第二个写锁才能成功

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/34\_redis分布式锁（二十九）：Semaphore使用演示/笔记.doc**

RSemaphore semaphore = redisson.getSemaphore("semaphore");

semaphore.trySetPermits(3);

for(int i = 0; i < 10; i++) {

new Thread(new Runnable() {

@Override

public void run() {

try {

System.out.println(new Date() + "：线程[" + Thread.currentThread().getName() + "]尝试获取Semaphore锁");

semaphore.acquire();

System.out.println(new Date() + "：线程[" + Thread.currentThread().getName() + "]成功获取到了Semaphore锁，开始工作");

Thread.sleep(3000);

semaphore.release();

System.out.println(new Date() + "：线程[" + Thread.currentThread().getName() + "]释放Semaphore锁");

} catch (Exception e) {

e.printStackTrace();

}

}

}).start();

}

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/12\_redis分布式锁（七）：可重入锁源码剖析之获取锁超时与自动释放/笔记.doc**

尝试获取锁超时，超时锁自动释放

（1）尝试获取锁超时

time = waitTime，是我们指定的最大的等待获取锁的时间，比如是100秒

current = 第一次尝试获取锁之前的一个时间戳

当前时间减去current = 第一次获取锁耗费的时间，假设是1秒

time -= 第一次获取锁耗费的时间 = 100秒 - 1秒 = 99秒，最多再等待99秒尝试获取这个锁，如果第一次获取锁的时间直接超过了waitTime等待最大超时时间，就会直接标记为获取锁失败

current = 当前时间

final RFuture<RedissonLockEntry> subscribeFuture = subscribe(threadId);

if (!await(subscribeFuture, time, TimeUnit.MILLISECONDS)) {

if (!subscribeFuture.cancel(false)) {

subscribeFuture.addListener(new FutureListener<RedissonLockEntry>() {

@Override

public void operationComplete(Future<RedissonLockEntry> future) throws Exception {

if (subscribeFuture.isSuccess()) {

unsubscribe(subscribeFuture, threadId);

}

}

});

}

acquireFailed(threadId);

return false;

}

他肯定是做了一些事情，看起来是订阅了什么东西，基于redis的PUB/SUB发布订阅模型，订阅了什么东西，执行了一些操作，所以在下面，将可以等待的时间又减去了这段时间的耗时，比如说是1秒

此时time = 98秒

接下来进入死循环，不断的尝试获取锁、等待，每次time都不断的减去尝试获取锁的耗时，以及等待的耗时，然后如果说在time范围内，获取到了锁，就会返回true，如果始终无法获取到锁的话，那么就会在time指定的最大时间之后，就返回一个false

（2）超时锁自动释放

就是我们自己定义了一个leaseTime和TimeUnit

if (leaseTime != -1) {

return tryLockInnerAsync(leaseTime, unit, threadId, RedisCommands.EVAL\_LONG);

}

如果你没传递这个leaseTime的话，这个if分支是不会走的

RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(commandExecutor.getConnectionManager().getCfg().getLockWatchdogTimeout(), TimeUnit.MILLISECONDS, threadId, RedisCommands.EVAL\_LONG);

用默认的LockWatchdogTimeout()时间，30000毫秒，30秒

如果你自己指定了一个leaseTime，就会直接执行lua脚本去加锁，加完锁的结果就直接返回了，并不会对那个future加一个监听器以及执行定时调度任务去刷新key的生存周期，因为你已经指定了leaseTime以后，就意味着你需要的是这个key最多存在10秒钟，必须被删除

"if (redis.call('exists', KEYS[1]) == 0) then " +

"redis.call('hset', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

hset anyLock 7124f827-475a-4e66-9f3c-fc00e8e8712e:1 1

anyLock: {

“7124f827-475a-4e66-9f3c-fc00e8e8712e:1”: 1

}

pexpire anyLock 10000

也就是说，人家在加锁的时候就设定好了，我们的锁key最多就只能存活10秒钟，而且后台没有定时调度的任务不断的去刷新锁key的生存周期

我们的那个锁到了10秒钟，就会自动被redis给删除，生存时间只能是10秒钟，然后就会自动释放掉了，别的客户端就可以加锁了，但是在10秒之内，其实你也可以自己去手动释放锁

无非就是我们上一讲讲的原理，递减加锁次数，删除锁key

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/09\_redis分布式锁（四）：可重入锁源码剖析之可重入加锁/笔记.doc**

如果客户端一直持有锁，没有释放，redis中的锁key是如何一直保持着存活不要过期的，这个事情也已经搞清楚了，无非就是靠的是后台每隔10秒的定时调度任务，如果持有锁的客户端宕机了，锁没有了定时调度任务更新生存周期，30秒过后自然就释放掉了

如果客户端一直持有锁，没有释放掉，加锁的时候，那个加锁的lua脚本，会返回什么东西？看一下这个细节

（顺带一提，redisson的源码写的真漂亮，bytetcc，这种才是应该有的开源项目的代码质量，抽象设计，代码的思路和顺序，读人家的源码，真是一种享受。spring cloud的eureka和hystrix，这两个开源项目，如此大名鼎鼎的spring cloud集成的核心组件，源码质量只差，真的是读源码的时候，感觉那个源码就跟实习生写的一样）

（1）如果一开始这个锁是没有的，第一次来加锁，这段lua脚本会如何执行？

"if (redis.call('exists', KEYS[1]) == 0) then " +

"redis.call('hset', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

一开始这个锁如果没有，第一次加锁，会进这个if then分支，hset设置一个hash的数据结构，pexpire设置这个key的生存时间，直接返回nil，也就是已给null，这个lua脚本后面的内容其实就不会执行了

如果Future拿到了那个lua脚本执行成功后的返回值之后，就会触发一个监听器

吐一个小槽，从代码的可读性的层面，这个代码不应该这么写，ttlRemaing为Null，如果拿到了一个RFuture.getNow()，获取到了一个Long result，如果if(result == null) acquiredLock = true

（2）如果是在一个客户端的一个线程内，先对一个lock进行了加锁，然后后面又加了一次锁，形成了一个叫做可重入锁的概念，就同一个线程对一个lock可以反复的重复加锁多次，每次加锁和一次释放锁必须是配对的

anyLock:

{

“8e6b27a7-5346-483a-b9b5-0957c690c27f:1”: 1

}

exists anyLock是不是为0呢？此时这个anyLock这个key已经存在了

hexists anyLock 8e6b27a7-5346-483a-b9b5-0957c690c27f:1，此时就是判断一下anyLock对应的hash数据结构中，是否存在一个8e6b27a7-5346-483a-b9b5-0957c690c27f:1一个key，此时肯定是存在的

说明这个客户端的这个线程，已经对这个锁加过一次锁了

"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +

"redis.call('hincrby', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

hincrby anyLock 8e6b27a7-5346-483a-b9b5-0957c690c27f:1，此时就会将8e6b27a7-5346-483a-b9b5-0957c690c27f:1这个key对应的值（本来是1），累加1，变为2，此时这个hash的数据变成：

anyLock:

{

“8e6b27a7-5346-483a-b9b5-0957c690c27f:1”: 2

}

整个anyLock锁的生存周期重新刷新为30000毫秒，pexpire anyLock 30000

返回值还是null，说明加锁成功

我之前给大家讲源码的时候，当时不好细说，做了一些比喻，但是其实这里，这个数字，1、2、3、4，代表的是一个可重入锁的概念，也就是在一个线程内可以对一个锁，多次加锁，可重入，一次加锁跟一次释放锁，是配对的

无论加多少次锁，其实anyLock这个东西一直存在，还没有被完全释放掉，那么此时就是后台的watchdog调度任务就会每隔10秒不断的刷新anyLock的生存后期，保证他对应的锁key不要过期

通过代码演示一下刚才说的可重入锁的概念

（3）当已经对一个锁加锁之后，同一个客户端的其他线程，或者是其他客户端的线程，尝试对这个key进行加锁会怎么样？

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~11资料/01\_分布式系统的接口幂等性到底指的是什么？/笔记.docx**

1、面试题

分布式服务接口的幂等性如何设计（比如不能重复扣款）？

2、面试官心里分析

从这个问题开始，面试官就已经进入了实际的生产问题的面试了

一个分布式系统中的某个接口，要保证幂等性，该如何保证？这个事儿其实是你做分布式系统的时候必须要考虑的一个生产环境的技术问题。啥意思呢？

你看，假如你有个服务提供一个接口，结果这服务部署在了5台机器上，接着有个接口就是付款接口。然后人家用户在前端上操作的时候，不知道为啥，总之就是一个订单不小心发起了两次支付请求，然后这俩请求分散在了这个服务部署的不同的机器上，好了，结果一个订单扣款扣两次？尴尬了。。。

或者是订单系统调用支付系统进行支付，结果不小心因为网络超时了，然后订单系统走了前面我们看到的那个重试机制，咔嚓给你重试了一把，好，支付系统收到一个支付请求两次，而且因为负载均衡算法落在了不同的机器上，尴尬了。。。

所以你肯定得知道这事儿，否则你做出来的分布式系统恐怕容易埋坑

网络问题很常见，100次请求，都ok；1万次，可能1次是超时会重试；10万，10次；100万，100次；如果有100个请求重复了，你没处理，导致订单扣款2次，100个订单都扣错了；每天被100个用户投诉；一个月被3000个用户投诉

我们之前生产就遇到过，是往数据库里写入数据，重复的请求，就导致我们的数据经常会错，出现一些重复数据，就会导致一些问题

3、面试题剖析

这个不是技术问题，这个没有通用的一个方法，这个是结合业务来看应该如何保证幂等性的，你的经验。

所谓幂等性，就是说一个接口，多次发起同一个请求，你这个接口得保证结果是准确的，比如不能多扣款，不能多插入一条数据，不能将统计值多加了1。这就是幂等性，不给大家来学术性词语了。

其实保证幂等性主要是三点：

（1）对于每个请求必须有一个唯一的标识，举个例子：订单支付请求，肯定得包含订单id，一个订单id最多支付一次，对吧

（2）每次处理完请求之后，必须有一个记录标识这个请求处理过了，比如说常见的方案是在mysql中记录个状态啥的，比如支付之前记录一条这个订单的支付流水，而且支付流水采

（3）每次接收请求需要进行判断之前是否处理过的逻辑处理，比如说，如果有一个订单已经支付了，就已经有了一条支付流水，那么如果重复发送这个请求，则此时先插入支付流水，orderId已经存在了，唯一键约束生效，报错插入不进去的。然后你就不用再扣款了。

（4）上面只是给大家举个例子，实际运作过程中，你要结合自己的业务来，比如说用redis用orderId作为唯一键。只有成功插入这个支付流水，才可以执行实际的支付扣款。

要求是支付一个订单，必须插入一条支付流水，order\_id建一个唯一键，unique key

所以你在支付一个订单之前，先插入一条支付流水，order\_id就已经进去了

你就可以写一个标识到redis里面去，set order\_id payed，下一次重复请求过来了，先查redis的order\_id对应的value，如果是payed就说明已经支付过了，你就别重复支付了

然后呢，你再重复支付这个订单的时候，你写尝试插入一条支付流水，数据库给你报错了，说unique key冲突了，整个事务回滚就可以了

来保存一个是否处理过的标识也可以，服务的不同实例可以一起操作redis。

# **File Path: /Users/xiaotingting/Downloads/学习资料/07\_分布式系统架构实战unzip/01~46资料/01\_基于redis和zookeeper的分布式锁实现原理/笔记.docx**

1、面试题

一般实现分布式锁都有哪些方式？使用redis如何设计分布式锁？使用zk来设计分布式锁可以吗？这两种分布式锁的实现方式哪种效率比较高？3616118202

2、面试官心里分析

其实一般问问题，都是这么问的，先问问你zk，然后其实是要过度的zk关联的一些问题里去，比如分布式锁。因为在分布式系统开发中，分布式锁的使用场景还是很常见的。

3、面试题剖析

（1）redis分布式锁

官方叫做RedLock算法，是redis官方支持的分布式锁算法。

这个分布式锁有3个重要的考量点，互斥（只能有一个客户端获取锁），不能死锁，容错（大部分redis节点或者这个锁就可以加可以释放）

第一个最普通的实现方式，如果就是在redis里创建一个key算加锁

SET my:lock 随机值 NX PX 30000，这个命令就ok，这个的NX的意思就是只有key不存在的时候才会设置成功，PX 30000的意思是30秒后锁自动释放。别人创建的时候如果发现已经有了就不能加锁了。

释放锁就是删除key，但是一般可以用lua脚本删除，判断value一样才删除：

关于redis如何执行lua脚本，自行百度

if redis.call("get",KEYS[1]) == ARGV[1] then

return redis.call("del",KEYS[1])

else

return 0

end

为啥要用随机值呢？因为如果某个客户端获取到了锁，但是阻塞了很长时间才执行完，此时可能已经自动释放锁了，此时可能别的客户端已经获取到了这个锁，要是你这个时候直接删除key的话会有问题，所以得用随机值加上面的lua脚本来释放锁。

但是这样是肯定不行的。因为如果是普通的redis单实例，那就是单点故障。或者是redis普通主从，那redis主从异步复制，如果主节点挂了，key还没同步到从节点，此时从节点切换为主节点，别人就会拿到锁。

第二个问题，RedLock算法

这个场景是假设有一个redis cluster，有5个redis master实例。然后执行如下步骤获取一把锁：

1）获取当前时间戳，单位是毫秒

2）跟上面类似，轮流尝试在每个master节点上创建锁，过期时间较短，一般就几十毫秒

3）尝试在大多数节点上建立一个锁，比如5个节点就要求是3个节点（n / 2 +1）

4）客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了

5）要是锁建立失败了，那么就依次删除这个锁

6）只要别人建立了一把分布式锁，你就得不断轮询去尝试获取锁

（2）zk分布式锁

zk分布式锁，其实可以做的比较简单，就是某个节点尝试创建临时znode，此时创建成功了就获取了这个锁；这个时候别的客户端来创建锁会失败，只能注册个监听器监听这个锁。释放锁就是删除这个znode，一旦释放掉就会通知客户端，然后有一个等待着的客户端就可以再次重新枷锁。

/\*\*

\* ZooKeeperSession

\* @author Administrator

\*

\*/

public class ZooKeeperSession {

private static CountDownLatch connectedSemaphore = new CountDownLatch(1);

private ZooKeeper zookeeper;

private CountDownLatch latch;

public ZooKeeperSession() {

try {

this.zookeeper = new ZooKeeper(

"192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181",

50000,

new ZooKeeperWatcher());

try {

connectedSemaphore.await();

} catch(InterruptedException e) {

e.printStackTrace();

}

System.out.println("ZooKeeper session established......");

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*

\* 获取分布式锁

\* @param productId

\*/

public Boolean acquireDistributedLock(Long productId) {

String path = "/product-lock-" + productId;

try {

zookeeper.create(path, "".getBytes(),

Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

return true;

} catch (Exception e) {

while(true) {

try {

Stat stat = zk.exists(path, true); // 相当于是给node注册一个监听器，去看看这个监听器是否存在

if(stat != null) {

this.latch = new CountDownLatch(1);

this.latch.await(waitTime, TimeUnit.MILLISECONDS);

this.latch = null;

}

zookeeper.create(path, "".getBytes(),

Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

return true;

} catch(Exception e) {

continue;

}

}

// 很不优雅，我呢就是给大家来演示这么一个思路

// 比较通用的，我们公司里我们自己封装的基于zookeeper的分布式锁，我们基于zookeeper的临时顺序节点去实现的，比较优雅的

}

return true;

}

/\*\*

\* 释放掉一个分布式锁

\* @param productId

\*/

public void releaseDistributedLock(Long productId) {

String path = "/product-lock-" + productId;

try {

zookeeper.delete(path, -1);

System.out.println("release the lock for product[id=" + productId + "]......");

} catch (Exception e) {

e.printStackTrace();

}

}

/\*\*

\* 建立zk session的watcher

\* @author Administrator

\*

\*/

private class ZooKeeperWatcher implements Watcher {

public void process(WatchedEvent event) {

System.out.println("Receive watched event: " + event.getState());

if(KeeperState.SyncConnected == event.getState()) {

connectedSemaphore.countDown();

}

if(this.latch != null) {

this.latch.countDown();

}

}

}

/\*\*

\* 封装单例的静态内部类

\* @author Administrator

\*

\*/

private static class Singleton {

private static ZooKeeperSession instance;

static {

instance = new ZooKeeperSession();

}

public static ZooKeeperSession getInstance() {

return instance;

}

}

/\*\*

\* 获取单例

\* @return

\*/

public static ZooKeeperSession getInstance() {

return Singleton.getInstance();

}

/\*\*

\* 初始化单例的便捷方法

\*/

public static void init() {

getInstance();

}

}

（3）redis分布式锁和zk分布式锁的对比

redis分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能

zk分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小

另外一点就是，如果是redis获取锁的那个客户端bug了或者挂了，那么只能等待超时时间之后才能释放锁；而zk的话，因为创建的是临时znode，只要客户端挂了，znode就没了，此时就自动释放锁

redis分布式锁大家每发现好麻烦吗？遍历上锁，计算时间等等。。。zk的分布式锁语义清晰实现简单

所以先不分析太多的东西，就说这两点，我个人实践认为zk的分布式锁比redis的分布式锁牢靠、而且模型简单易用

（4）zookeeper分布式锁实现代码

public class ZooKeeperDistributedLock implements Watcher{

private ZooKeeper zk;

private String locksRoot= "/locks";

private String productId;

private String waitNode;

private String lockNode;

private CountDownLatch latch;

private CountDownLatch connectedLatch = new CountDownLatch(1);

private int sessionTimeout = 30000;

public ZooKeeperDistributedLock(String productId){

this.productId = productId;

try {

String address = "192.168.31.187:2181,192.168.31.19:2181,192.168.31.227:2181";

zk = new ZooKeeper(address, sessionTimeout, this);

connectedLatch.await();

} catch (IOException e) {

throw new LockException(e);

} catch (KeeperException e) {

throw new LockException(e);

} catch (InterruptedException e) {

throw new LockException(e);

}

}

public void process(WatchedEvent event) {

if(event.getState()==KeeperState.SyncConnected){

connectedLatch.countDown();

return;

}

if(this.latch != null) {

this.latch.countDown();

}

}

public void acquireDistributedLock() {

try {

if(this.tryLock()){

return;

}

else{

waitForLock(waitNode, sessionTimeout);

}

} catch (KeeperException e) {

throw new LockException(e);

} catch (InterruptedException e) {

throw new LockException(e);

}

}

public boolean tryLock() {

try {

// 传入进去的locksRoot + “/” + productId

// 假设productId代表了一个商品id，比如说1

// locksRoot = locks

// /locks/10000000000，/locks/10000000001，/locks/10000000002

lockNode = zk.create(locksRoot + "/" + productId, new byte[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL);

// 看看刚创建的节点是不是最小的节点

// locks：10000000000，10000000001，10000000002

List<String> locks = zk.getChildren(locksRoot, false);

Collections.sort(locks);

if(lockNode.equals(locksRoot+"/"+ locks.get(0))){

//如果是最小的节点,则表示取得锁

return true;

}

//如果不是最小的节点，找到比自己小1的节点

int previousLockIndex = -1;

for(int i = 0; i < locks.size(); i++) {

if(lockNode.equals(locksRoot + “/” + locks.get(i))) {

previousLockIndex = i - 1;

break;

}

}

this.waitNode = locks.get(previousLockIndex);

} catch (KeeperException e) {

throw new LockException(e);

} catch (InterruptedException e) {

throw new LockException(e);

}

return false;

}

private boolean waitForLock(String waitNode, long waitTime) throws InterruptedException, KeeperException {

Stat stat = zk.exists(locksRoot + "/" + waitNode, true);

if(stat != null){

this.latch = new CountDownLatch(1);

this.latch.await(waitTime, TimeUnit.MILLISECONDS); this.latch = null;

}

return true;

}

public void unlock() {

try {

// 删除/locks/10000000000节点

// 删除/locks/10000000001节点

System.out.println("unlock " + lockNode);

zk.delete(lockNode,-1);

lockNode = null;

zk.close();

} catch (InterruptedException e) {

e.printStackTrace();

} catch (KeeperException e) {

e.printStackTrace();

}

}

public class LockException extends RuntimeException {

private static final long serialVersionUID = 1L;

public LockException(String e){

super(e);

}

public LockException(Exception e){

super(e);

}

}

// 如果有一把锁，被多个人给竞争，此时多个人会排队，第一个拿到锁的人会执行，然后释放锁，后面的每个人都会去监听排在自己前面的那个人创建的node上，一旦某个人释放了锁，排在自己后面的人就会被zookeeper给通知，一旦被通知了之后，就ok了，自己就获取到了锁，就可以执行代码了

}