[https://mp.weixin.qq.com/s?\_\_biz=MzIyNDU2ODA4OQ==&mid=2247483934&idx=1&sn=8b5a9dcf5f7601971464bf5e0c9944de&chksm=e80db468df7a3d7eef45c8e35e60c4aae5fa3ffa646dbec3833d5b882506fcef6336c2cac354&scene=21#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzIyNDU2ODA4OQ==&mid=2247483934&idx=1&sn=8b5a9dcf5f7601971464bf5e0c9944de&chksm=e80db468df7a3d7eef45c8e35e60c4aae5fa3ffa646dbec3833d5b882506fcef6336c2cac354&scene=21" \l "wechat_redirect)

# Redis中是如何实现分布式锁的？

分布式锁常见的三种实现方式：

1. 数据库乐观锁；
2. 基于Redis的分布式锁；
3. 基于ZooKeeper的分布式锁。

本地面试考点是，你对Redis使用熟悉吗？Redis中是如何实现分布式锁的。

## 要点

Redis要实现分布式锁，以下条件应该得到满足

**互斥性**

* 在任意时刻，只有一个客户端能持有锁。

**不能死锁**

* 客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。

**容错性**

* 只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。

## 实现

可以直接通过 set key value px milliseconds nx 命令实现加锁， 通过Lua脚本实现解锁。



**代码解释**

* set 命令要用 set key value px milliseconds nx，替代 setnx + expire 需要分两次执行命令的方式，保证了原子性，
* value 要具有唯一性，可以使用UUID.randomUUID().toString()方法生成，用来标识这把锁是属于哪个请求加的，在解锁的时候就可以有依据；
* 释放锁时要验证 value 值，防止误解锁；
* 通过 Lua 脚本来避免 Check And Set 模型的并发问题，因为在释放锁的时候因为涉及到多个Redis操作 （利用了eval命令执行Lua脚本的原子性）；

**加锁代码分析**

首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，用来标识这把锁是属于哪个请求加的，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。

**解锁代码分析**

将Lua代码传到jedis.eval()方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。在执行的时候，首先会获取锁对应的value值，检查是否与requestId相等，如果相等则解锁（删除key）。

**存在的风险**

如果存储锁对应key的那个节点挂了的话，就可能存在丢失锁的风险，导致出现多个客户端持有锁的情况，这样就不能实现资源的独享了。

1. 客户端A从master获取到锁
2. 在master将锁同步到slave之前，master宕掉了（Redis的主从同步通常是异步的）。  
   主从切换，slave节点被晋级为master节点
3. 客户端B取得了同一个资源被客户端A已经获取到的另外一个锁。导致存在同一时刻存不止一个线程获取到锁的情况。

## redlock算法出现

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，那么就依次之前建立过的锁删除；
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。
7. Redis 官方给出了以上两种基于 Redis 实现分布式锁的方法，详细说明可以查看：
8. https://redis.io/topics/distlock 。

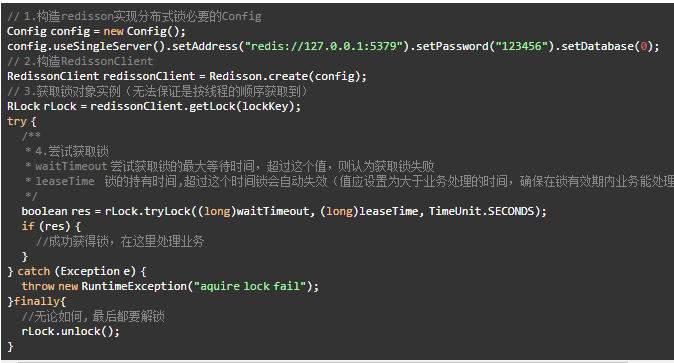
## Redisson实现

Redisson是一个在Redis的基础上实现的Java驻内存数据网格（In-Memory Data Grid）。它不仅提供了一系列的分布式的Java常用对象，还实现了可重入锁（Reentrant Lock）、公平锁（Fair Lock、联锁（MultiLock）、 红锁（RedLock）、 读写锁（ReadWriteLock）等，还提供了许多分布式服务。

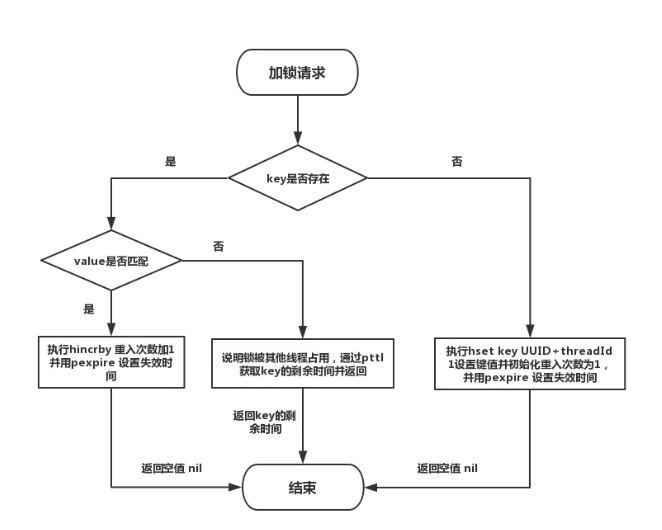
Redisson提供了使用Redis的最简单和最便捷的方法。Redisson的宗旨是促进使用者对Redis的关注分离（Separation of Concern），从而让使用者能够将精力更集中地放在处理业务逻辑上。

**Redisson 分布式重入锁用法**

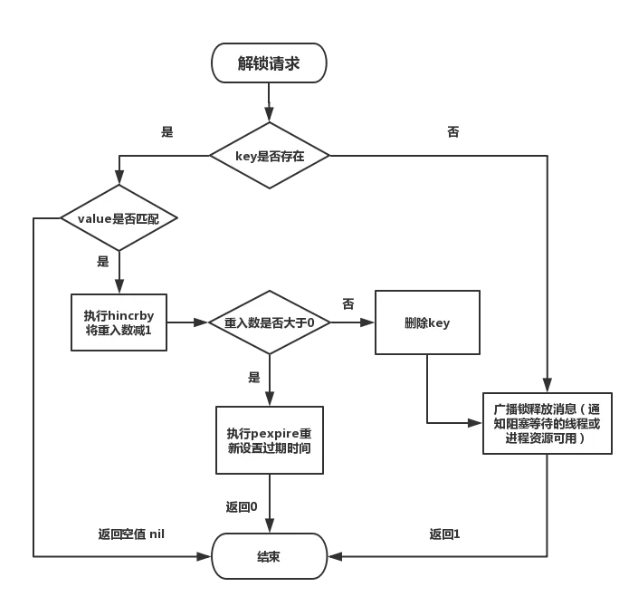
Redisson 支持单点模式、主从模式、哨兵模式、集群模式，这里以单点模式为例：



加锁流程图



解锁流程图



我们可以看到，RedissonLock是可重入的，并且考虑了失败重试，可以设置锁的最大等待时间， 在实现上也做了一些优化，减少了无效的锁申请，提升了资源的利用率。

需要特别注意的是，RedissonLock 同样没有解决 节点挂掉的时候，存在丢失锁的风险的问题。而现实情况是有一些场景无法容忍的，所以 Redisson 提供了实现了redlock算法的 RedissonRedLock，RedissonRedLock 真正解决了单点失败的问题，代价是需要额外的为 RedissonRedLock 搭建Redis环境。

所以，如果业务场景可以容忍这种小概率的错误，则推荐使用 RedissonLock， 如果无法容忍，则推荐使用 RedissonRedLock。