**Redis分布式锁**

## 什么是分布式锁？

## 分布式CAP原则告诉我们，Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。

在单机（单进程）环境中，JAVA提供了很多并发相关API，但在多机（多进程）环境中就无能为力了。有些场景需要加锁处理，比如：秒杀，全局递增ID，楼层生成等等

## 分布式的锁实现有基于数据库实现分布式锁 基于缓存实现分布式锁 基于zookeeper实现分布式锁

一．对一个redis节点的加锁

redis能用的的加锁命令分表是INCR、SETNX、SET

**1.1. 第一种锁命令INCR**

这种加锁的思路是， key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCR 操作进行加一。执行完删除锁  
然后其它用户在执行 INCR 操作进行加一时，如果返回的数大于 1 ，说明这个锁正在被使用当中。

  $redis->incr($key);

1. $redis->expire($key, $ttl); //设置生成时间为1秒

**1.2. 第二种锁SETNX 和expire**

1. 利用setnx和expire命令实现加锁。当一个线程执行setnx返回1，说明key不存在，该线程获得锁；当一个线程执行setnx返回0，说明key已经存在，则获取锁失败。expire就是给锁加一个过期时间。伪代码如下：
2. if(setnx(key,value)==1){
3. expire(key,expireTime)
4. try{
5. //业务处理
6. }finally{
7. del(key)
8. }
9. }

该方案有一个致命问题，由于setnx和expire是两条Redis命令，不具备原子性，如果一个线程在执行完setnx()之后突然崩溃，导致锁没有设置过期时间，那么将会发生死锁。

**1.3第三种锁SET**

Set key value ex|px nx|xx (key，值，秒|毫秒 不存在|存在)

Redis2.6.12以上版本为set命令增加了可选参数，伪代码如下：

if(redis.set(key,value,"ex 180","nx")){

//业务处理

do something;

//释放锁

redis.delete(key);

}

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。但是这里会出现一个问题，如果a客户端操作时间超过了过期时间，这时b客户端就能获取到锁，等a处理完之后又把锁释放了，这就出问题了。

处理方式：1.把value设置成过期时间，删除的时候和当前时间判断

String expireValue = System.currentTime()+expireTime;

if(redis.set(key, expireValue,"px”, expireTime,"nx")){

//业务处理

do something;

//释放锁

If (expireValue!=null&& System.currentTime()>=expireValue){

redis.delete(key);}

}

2.使用lua脚本删除锁

String script = “if redis.call('get', KEYS[1]) == ARGV[1]  
 then  
 return redis.call('del', KEYS[1])  
 else  
 return 0  
 end”

try {  
 boolean result = redisUtil.setnx(key,value,100);  
 if(!result){  
 System.out.println("系统繁忙中");  
 } else {  
 System.out.println("这里是你业务代码");  
 }  
  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally {  
 Object result = redisUtil.execute(script,Collections.singletonList(key),value);  
 System.out.println(result);

}

为了防止业务时间超过有效期，可以在获取锁之后开一个线程重新设置锁的有效期。

**二. Redisson**

 客户端1在Redis的master节点上拿到了锁

1. Master宕机了，存储锁的key还没有来得及同步到Slave上
2. master故障，发生故障转移，slave节点升级为master节点
3. 客户端2从新的Master获取到了对应同一个资源的锁

　　于是，客户端1和客户端2同时持有了同一个资源的锁。锁的安全性被打破了。针对这个问题。Redis作者antirez提出了RedLock算法来解决这个问题

## 2、RedLock算法的实现思路

　　antirez提出的redlock算法实现思路大概是这样的。

　　客户端按照下面的步骤来获取锁：

1. 获取当前时间的毫秒数T1。
2. 按顺序依次向N个Redis节点执行获取锁的操作。这个获取锁的操作和上一篇中基于单Redis节点获取锁的过程相同。包括唯一UUID作为Value以及锁的过期时间(expireTime)。为了保证在某个在某个Redis节点不可用的时候算法能够继续运行，这个获取锁的操作还需要一个超时时间。它应该远小于锁的过期时间。客户端向某个Redis节点获取锁失败后，应立即尝试下一个Redis节点。这里失败包括Redis节点不可用或者该Redis节点上的锁已经被其他客户端持有。
3. 计算整个获取锁过程的总耗时。即当前时间减去第一步记录的时间。计算公司为T2=now()- T1。如果客户端从大多数Redis节点(>N/2 +1)成功获取到锁。并且获取锁总共消耗的时间小于锁的过期时间（即T2<expireTime）。则认为客户端获取锁成功，否则，认为获取锁失败
4. 如果获取锁成功，需要重新计算锁的过期时间。它等于最初锁的有效时间减去第三步计算出来获取锁消耗的时间，即expireTime - T2
5. 如果最终获取锁失败，那么客户端立即向所有Redis系欸但发起释放锁的操作（防止某个masterset了命令，获取到锁，但相应出问题，以至于释放不了该锁）。执行lua脚本

　　虽然说RedLock算法可以解决单点Redis分布式锁的安全性问题，但如果集群中有节点发生崩溃重启，还是会锁的安全性有影响的。具体出现问题的场景如下：

　　假设一共有5个Redis节点：A, B, C, D, E。设想发生了如下的事件序列：

1. 客户端1成功锁住了A, B, C，**获取锁**成功（但D和E没有锁住）
2. 节点C崩溃重启了，但客户端1在C上加的锁没有持久化下来，丢失了
3. 节点C重启后，客户端2锁住了C, D, E，**获取锁**成功

　　这样，客户端1和客户端2同时获得了锁（针对同一资源）。针对这样场景，解决方式也很简单，也就是让Redis崩溃后延迟重启，并且这个延迟时间大于锁的过期时间就好。这样等节点重启后，所有节点上的锁都已经失效了。也不存在以上出现2个客户端获取同一个资源的情况了。有slave的情况就让slave类似别那么快接替master

　　相比之下，RedLock安全性和稳定性都比前一篇文章中介绍的实现要好很多，但要说完全没有问题不是。例如，如果客户端获取锁成功后，如果访问共享资源操作执行时间过长，导致锁过期了，后续客户端获取锁成功了，这样在同一个时刻又出现了2个客户端获得了锁的情况。所以针对分布式锁的应用的时候需要多测试。服务器台数越多，出现不可预期的情况也越多。如果客户端获取锁之后，在上面第三步发生了GC得情况导致GC完成后，锁失效了，这样同时也使得同一时间有2个客户端获得了锁。如果系统对共享资源有非常严格要求得情况下，还是建议需要做数据库锁得得方案来补充。如飞机票或火车票座位得情况。对于一些抢购获取，针对偶尔出现超卖，后续可以人为沟通置换得方式采用分布式锁得方式没什么问题。因为可以绝大部分保证分布式锁的安全性。

## 3、分布式场景下基于Redis实现分布式锁的正确姿势

## Github地址：单机，哨兵，集群，主从等多个模式。https://github.com/redisson/redisson/wiki/2.-%E9%85%8D%E7%BD%AE%E6%96%B9%E6%B3%95

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，还提供了许多分布式服务

如果负责储存这个分布式锁的Redisson节点宕机以后，而且这个锁正好处于锁住的状态时，这个锁会出现锁死的状态。为了避免这种情况的发生，Redisson内部提供了一个监控锁的看门狗，它的作用是在Redisson实例被关闭前，不断的延长锁的有效期。默认情况下，看门狗的检查锁的超时时间是30秒钟，也可以通过修改[Config.lockWatchdogTimeout](https://github.com/redisson/redisson/wiki/2.-%E9%85%8D%E7%BD%AE%E6%96%B9%E6%B3%95#lockwatchdogtimeout%E7%9B%91%E6%8E%A7%E9%94%81%E7%9A%84%E7%9C%8B%E9%97%A8%E7%8B%97%E8%B6%85%E6%97%B6%E5%8D%95%E4%BD%8D%E6%AF%AB%E7%A7%92)来另行指定。

另外Redisson还通过加锁的方法提供了leaseTime的参数来指定加锁的时间。超过这个时间后锁便自动解开了。

**Redisson功能**

1、支持同步/异步/异步流/管道流方式连接

2、多样化数据序列化

3、集合数据分片

4、分布式对象

5、分布式集合

6、分布式锁和同步器

7、分布式服务（远程服务，实时对象，分布式任务）

8、独立节点模式

9、三方框架整合

目前redisson包已经有对redlock算法封装，接下来就具体看看使用redisson包来实现分布式锁的正确姿势。Redission有单机，哨兵，集群，主从等多个模式。

　　具体实现代码如下代码所示：

Redisson提供的分布式锁分类，都提供了看门狗，和设置有效期

### 1可重入锁（Reentrant Lock）

redisson.getLock("anyLock");

### 2公平锁（Fair Lock）

redisson.getFairLock("anyLock");

### 3联锁 （MultiLock） 所有锁上才成功

RLock lock1 = redissonInstance1.getLock("lock1");

RLock lock2 = redissonInstance2.getLock("lock2");

RLock lock3 = redissonInstance3.getLock("lock3");

RedissonMultiLock lock = new RedissonMultiLock(lock1, lock2, lock3);

### 4 红锁（RedLock） 大部分锁上就成功

RLock lock1 = redissonInstance1.getLock("lock1");

RLock lock2 = redissonInstance2.getLock("lock2");

RLock lock3 = redissonInstance3.getLock("lock3");

RedissonRedLock lock = new RedissonRedLock(lock1, lock2, lock3);

### 5. 读写锁（ReadWriteLock） 允许同时有多个读锁和一个写锁处于加锁状态。

RReadWriteLock rwlock = redisson.getReadWriteLock("anyRWLock");

// 最常见的使用方法

rwlock.readLock().lock();

// 或

rwlock.writeLock().lock();

### 6. 信号量（Semaphore）

RSemaphore semaphore = redisson.getSemaphore("semaphore");

semaphore.acquire();

### 7. 可过期性信号量（PermitExpirableSemaphore）

RPermitExpirableSemaphore semaphore = redisson.getPermitExpirableSemaphore("mySemaphore");

String permitId = semaphore.acquire();

// 获取一个信号，有效期只有2秒钟。

String permitId = semaphore.acquire(2, TimeUnit.SECONDS);

// ...

semaphore.release(permitId);

**8. 闭锁（CountDownLatch）**

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

latch.trySetCount(1);

latch.await();

// 在其他线程或其他JVM里

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

latch.countDown();

3.1单机版

Config config = new Config();  
 config.useSingleServer().setAddress("127.0.0.1:6379").setDatabase(0);  
 RedissonClient redissonClient = Redisson.*create*(config);  
 String key = "key:1";  
 RLock lock = redissonClient.getLock(key);  
 try {  
 boolean b = lock.tryLock(10, 100, TimeUnit.*SECONDS*);  
 if (b){  
 System.*out*.println("开始业务");  
 }else{  
 System.*out*.println("获取锁失败");  
 }  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally {  
 lock.unlock();  
 }  
  
}

几个加锁的方法：

void lock(long leaseTime, TimeUnit unit); 加锁和锁的有效期，到期自动解锁，无需手动unlock，会一直阻塞直到锁的时间失效  
boolean tryLock(long time, TimeUnit unit) throwsInterruptedException;

尝试获取锁，成功返回true，失败false  
boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws InterruptedException;

尝试在time时间内获取锁，成功返回true，失败false  
还有三个异步获取锁，大致方法和上面差不多

Future<Boolean> res = lock.tryLockAsync(3,100, TimeUnit.SECONDS);

# 3.2 主从redisson

Config config = new Config();  
 config.useMasterSlaveServers()  
 .setMasterAddress("redis://127.0.0.1:6379")  
 .addSlaveAddress("redis://127.0.0.1:6380", "redis://127.0.0.1:6381");  
 RedissonClient redisson = Redisson.create(config);  
  
 String key ="product:001";  
 RLock lock = redisson.getLock(key);  
 try {  
 boolean res = lock.tryLock(10,100,TimeUnit.SECONDS);  
 if (res){  
 System.out.println("这里是你的业务代码");  
 }else{  
 System.out.println("系统繁忙");  
 }  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally {  
 lock.unlock();  
 }

# 3.3 哨兵redisson

Config config = new Config();  
 config.useSentinelServers()  
 .addSentinelAddress("redis://127.0.0.1:26379")  
 .addSentinelAddress("redis://127.0.0.1:26389")  
 .addSentinelAddress("redis://127.0.0.1:26399");  
 RedissonClient redisson = Redisson.create(config);  
  
 String key ="product:001";  
 RLock lock = redisson.getLock(key);  
 try {  
 boolean res = lock.tryLock(10,100,TimeUnit.SECONDS);  
 if (res){  
 System.out.println("这里是你的业务代码");  
 }else{  
 System.out.println("系统繁忙");  
 }  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally {  
 lock.unlock();  
 }

# 3.4 集群redisson

Config config = new Config();  
 config.useClusterServers()  
 *// 集群状态扫描间隔时间，单位是毫秒* .setScanInterval(2000)  
 *//cluster方式至少6个节点(3主3从，3主做sharding，3从用来保证主宕机后可以高可用)* .addNodeAddress("redis://127.0.0.1:6379" )  
 .addNodeAddress("redis://127.0.0.1:6380")  
 .addNodeAddress("redis://127.0.0.1:6381")  
 .addNodeAddress("redis://127.0.0.1:6382")  
 .addNodeAddress("redis://127.0.0.1:6383")  
 .addNodeAddress("redis://127.0.0.1:6384");  
 RedissonClient redisson = Redisson.create(config);  
  
 String key ="product:001";  
 RLock lock = redisson.getLock(key);  
 try {  
 boolean res = lock.tryLock(10,100,TimeUnit.SECONDS);  
 if (res){  
 System.out.println("这里是你的业务代码");  
 }else{  
 System.out.println("系统繁忙");  
 }  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally {  
 lock.unlock();  
 }

　　由于RedLock是针对主从和集群场景准备。上面代码采用哨兵模式。所以要让上面代码运行起来，需要先本地搭建Redis哨兵模式。本人的环境是Windows,具体Windows 哨兵环境搭建参考文章：[redis sentinel部署(Windows下实现)](https://blog.csdn.net/liuchuanhong1/article/details/53206028)。

　　具体测试代码如下所示：

[复制代码](javascript:void(0);)

复制代码

public class RedisDistributedRedLockTest {

static int n = 5;

public static void secskill() {

if(n <= 0) {

System.out.println("抢购完成");

return;

}

System.out.println(--n);

}

public static void main(String[] args) {

Config config = new Config();

//支持单机，主从，哨兵，集群等模式

//此为哨兵模式

config.useSentinelServers()

.setMasterName("mymaster")

.addSentinelAddress("127.0.0.1:26369","127.0.0.1:26379","127.0.0.1:26389")

.setDatabase(0);

Runnable runnable = () -> {

RedisDistributedRedLock redisDistributedRedLock = null;

RedissonClient redissonClient = null;

try {

redissonClient = Redisson.create(config);

redisDistributedRedLock = new RedisDistributedRedLock(redissonClient, "stock\_lock");

redisDistributedRedLock.acquire();

secskill();

System.out.println(Thread.currentThread().getName() + "正在运行");

} finally {

if (redisDistributedRedLock != null) {

redisDistributedRedLock.release(null);

}

redissonClient.shutdown();

}

};

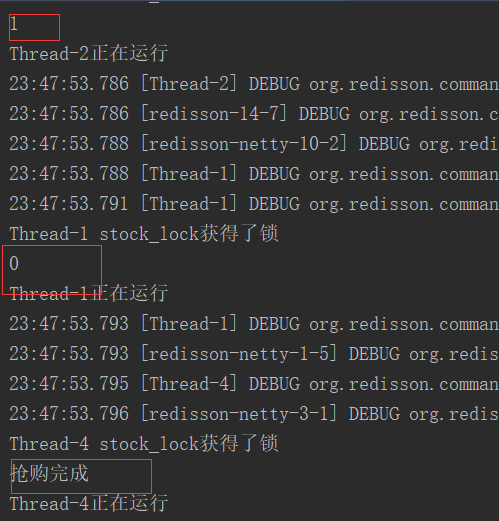
for (int i = 0; i < 10; i++) {

Thread t = new Thread(runnable);

t.start();

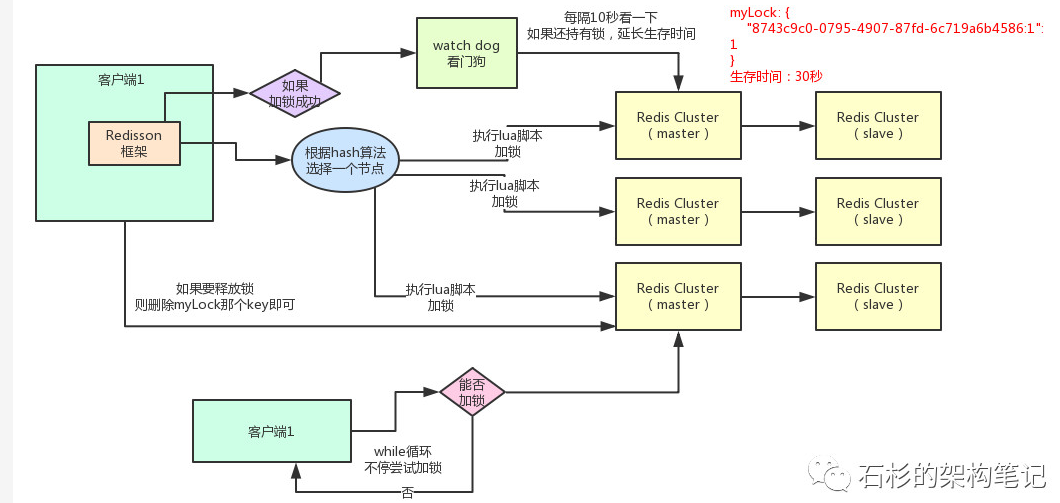
}

}



3.5其他配置见redisson的github地址

3.6**Redisson实现Redis分布式锁的底层原理**

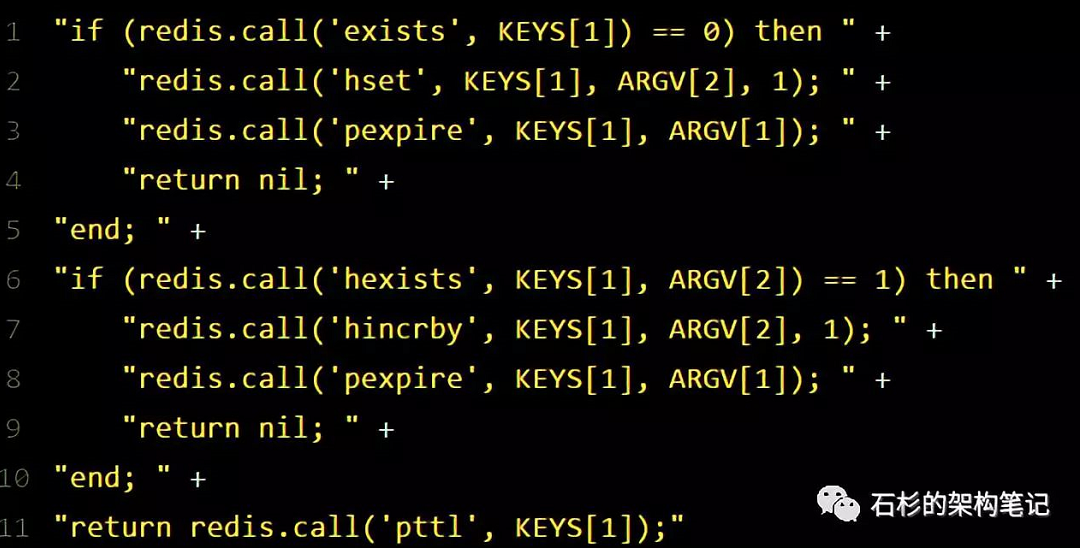


**（1）加锁机制**

咱们来看上面那张图，现在某个客户端要加锁。如果该客户端面对的是一个redis cluster集群，他首先会根据hash节点选择一台机器。

**这里注意**，仅仅只是选择一台机器！这点很关键！

紧接着，就会发送一段lua脚本到redis上，那段lua脚本如下所示：



为啥要用lua脚本呢？

因为一大坨复杂的业务逻辑，可以通过封装在lua脚本中发送给redis，保证这段复杂业务逻辑执行的**原子性**。

那么，这段lua脚本是什么意思呢？

**KEYS[1]**代表的是你加锁的那个key，比如说：

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

这里你自己设置了加锁的那个锁key就是“myLock”。

**ARGV[1]**代表的就是锁key的默认生存时间，默认30秒。

**ARGV[2]**代表的是加锁的客户端的ID，类似于下面这样：

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

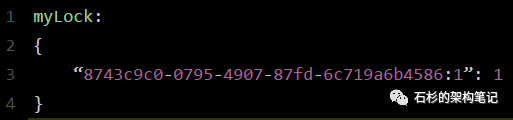
给大家解释一下，第一段if判断语句，就是用“exists myLock”命令判断一下，如果你要加锁的那个锁key不存在的话，你就进行加锁。

如何加锁呢？很简单，用下面的命令：

hset myLock

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

通过这个命令设置一个hash数据结构，这行命令执行后，会出现一个类似下面的数据结构：



上述就代表“8743c9c0-0795-4907-87fd-6c719a6b4586:1”这个客户端对“myLock”这个锁key完成了加锁。

接着会执行“pexpire myLock 30000”命令，设置myLock这个锁key的生存时间是30秒。

好了，到此为止，ok，加锁完成了。

**（2）锁互斥机制**

那么在这个时候，如果客户端2来尝试加锁，执行了同样的一段lua脚本，会咋样呢？

很简单，第一个if判断会执行“exists myLock”，发现myLock这个锁key已经存在了。

接着第二个if判断，判断一下，myLock锁key的hash数据结构中，是否包含客户端2的ID，但是明显不是的，因为那里包含的是客户端1的ID。

所以，客户端2会获取到pttl myLock返回的一个数字，这个数字代表了myLock这个锁key的**剩余生存时间。**比如还剩15000毫秒的生存时间。

此时客户端2会进入一个while循环，不停的尝试加锁。

**（3）watch dog自动延期机制**

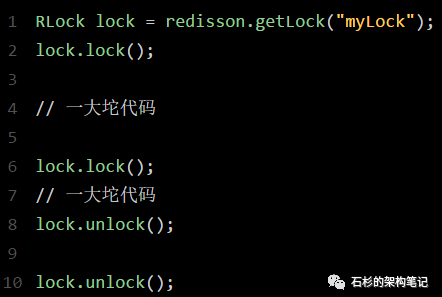
客户端1加锁的锁key默认生存时间才30秒，如果超过了30秒，客户端1还想一直持有这把锁，怎么办呢？

简单！只要客户端1一旦加锁成功，就会启动一个watch dog看门狗，**他是一个后台线程，会每隔10秒检查一下**，如果客户端1还持有锁key，那么就会不断的延长锁key的生存时间。

**（4）可重入加锁机制**

那如果客户端1都已经持有了这把锁了，结果可重入的加锁会怎么样呢？

比如下面这种代码：



这时我们来分析一下上面那段lua脚本。

第一个if判断肯定不成立，“exists myLock”会显示锁key已经存在了。

第二个if判断会成立，因为myLock的hash数据结构中包含的那个ID，就是客户端1的那个ID，也就是“8743c9c0-0795-4907-87fd-6c719a6b4586:1”

此时就会执行可重入加锁的逻辑，他会用：

incrby myLock

 8743c9c0-0795-4907-87fd-6c71a6b4586:1 1

通过这个命令，对客户端1的加锁次数，累加1。

此时myLock数据结构变为下面这样：

大家看到了吧，那个myLock的hash数据结构中的那个客户端ID，就对应着加锁的次数

**（5）释放锁机制**

如果执行lock.unlock()，就可以释放分布式锁，此时的业务逻辑也是非常简单的。

其实说白了，就是每次都对myLock数据结构中的那个加锁次数减1。

如果发现加锁次数是0了，说明这个客户端已经不再持有锁了，此时就会用：

“del myLock”命令，从redis里删除这个key。

然后呢，另外的客户端2就可以尝试完成加锁了。

这就是所谓的**分布式锁的开源Redisson框架的实现机制。**

一般我们在生产系统中，可以用Redisson框架提供的这个类库来基于redis进行分布式锁的加锁与释放锁。

**（6）上述Redis分布式锁的缺点**

其实上面那种方案最大的问题，就是如果你对某个redis master实例，写入了myLock这种锁key的value，此时会异步复制给对应的master slave实例。

但是这个过程中一旦发生redis master宕机，主备切换，redis slave变为了redis master。

接着就会导致，客户端2来尝试加锁的时候，在新的redis master上完成了加锁，而客户端1也以为自己成功加了锁。

此时就会导致多个客户端对一个分布式锁完成了加锁。

这时系统在业务语义上一定会出现问题，**导致各种脏数据的产生**。

所以这个就是redis cluster，或者是redis master-slave架构的**主从异步复制**导致的redis分布式锁的最大缺陷：在redis master实例宕机的时候，可能导致多个客户端同时完成加锁。