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。

考点：Redis是如何实现分布式锁的。

#### 1、要点

Redis实现分布式锁，要满足以下条件：

* 互斥性：在任意时刻，只有一个客户端能持有锁。
* 不能锁死：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。
* 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

#### 2、实现

**Java实现：**

**1）加锁**

private static final String *LOCK\_SUCCESS* = "OK";

private static final String *SET\_IF\_NOT\_EXIST* = "NX";  
 private static final String *SET\_WITH\_EXPIRE\_TIME* = "PX";  
  
 */\*\*  
 \* 尝试获取分布式锁  
 \** ***@param*** *jedis Redis客户端  
 \** ***@param*** *lockKey 锁  
 \** ***@param*** *requestId 请求标识  
 \** ***@param*** *expireTime 超期时间  
 \** ***@return*** *\*/* public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {  
 String result = jedis.set(lockKey, requestId, *SET\_IF\_NOT\_EXIST*, *SET\_WITH\_EXPIRE\_TIME*, expireTime);  
 if (*LOCK\_SUCCESS*.equals(result)) {  
 return true;  
 }  
 return false;  
 }

加锁就一行代码：jedis.set(final String key, final String value, final String nxxx, final String expx, final int time);这个set()方法有5个形参：

* key，使用key来当锁，因为key是唯一的；
* value，传的是requestId，通过给value复制为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID.toString()方法生成。
* nxxx，这个参数填的是NX，意思是当key不存在时，我们进行set操作；若key已经存在，则不作任何操作；
* expx，这个参数填的PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。
* time，与第四个参数相呼应，代表key的过期时间。

以上代码满足了互斥性（任意时刻，只有一个客户端能持有锁）、不能锁死（设置过期时间）、加锁和解锁必须是同一个客户端。

**2）解锁**

private static final Long *RELEASE\_SUCCESS* = 1L;

*/\*\**

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

两行代码实现了释放分布式锁。第一行写了一个Lua脚本代码；第二行代码，将脚本传到jedis.eval()方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。

那么这段Lua代码的功能是什么呢？其实这很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。那么为什么要使用Lua语言实现呢？因为要确保上述操作是原子性的。执行eval()方法可以确保原子性源于Redis的特性。

**脚本实现：**

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。

//获取锁（unique\_value可以是UUID等）

SET resource\_name unique\_value NX PX 30000

//释放锁（lua脚本中，一定要比较value，防止误解锁）

if redis.call("get",KEYS[1]) == ARGV[1] then

return redis.call("del",KEYS[1])

else

return 0

end

* 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到（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，用来标识这把锁时属于哪个请求加的，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，首先户获取锁对应的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将锁同步打破slave之前，master宕掉了（Redis的主从同步通常是异步的）。主从切换，slave节点被晋升为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，如果建立锁的事件小于超时时间，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。



Redis官方给出了以上两种基于Redis实现分布式锁的方法。

1. **Redisson实现**

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：

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。

<https://github.com/javazhiyin/advanced-java/>

<https://crazyfzw.github.io/2019/04/15/distributed-locks-with-redis/>