# ### redis 实现分布式锁

## #####1. 实现锁的要求

互斥性。在任何时刻，保证只有一个客户端持有锁。

不能出现死锁。如果在一个客户端持有锁的期间，这个客户端崩溃了，也要保证后续的其他客户端可以上锁。

保证上锁和解锁都是同一个客户端。

#####2. 使用Redis，基于setnx命令。

## 1. 实现方式1

Redis实现分布式锁主要利用Redis的setnx命令。setnx是SET if not exists(如果不存在，则 SET)的简写。

```shell

127.0.0.1:6379> setnx lock value1 #在键lock不存在的情况下，将键key的值设置为value1

(integer) 1

127.0.0.1:6379> setnx lock value2 #试图覆盖lock的值，返回0表示失败

(integer) 0

127.0.0.1:6379> get lock #获取lock的值，验证没有被覆盖

"value1"

127.0.0.1:6379> del lock #删除lock的值，删除成功

(integer) 1

127.0.0.1:6379> setnx lock value2 #再使用setnx命令设置，返回0表示成功

(integer) 1

127.0.0.1:6379> get lock #获取lock的值，验证设置成功

"value2"

```

```java

那么接下来就可以写一段很简单的加锁代码：private static Jedis jedis = new Jedis("127.0.0.1");

private static final Long SUCCESS = 1L;

/\*\*

\* 加锁

\*/

public boolean tryLock(String key, String requestId) {

//使用setnx命令。

//不存在则保存返回1，加锁成功。如果已经存在则返回0，加锁失败。

return SUCCESS.equals(jedis.setnx(key, requestId));

}

//删除key的lua脚本，先比较requestId是否相等，相等则删除

private static final String DEL\_SCRIPT = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

/\*\*

\* 解锁

\*/

public boolean unLock(String key, String requestId) {

//删除成功表示解锁成功

Long result = (Long) jedis.eval(DEL\_SCRIPT, Collections.singletonList(key), Collections.singletonList(requestId));

return SUCCESS.equals(result);

}

```

\*\*存在的问题：\*\*

这仅仅满足上述的第一个条件和第三个条件，保证上锁和解锁都是同一个客户端，也保证只有一个客户端持有锁。

但是第二点没法保证，因为如果一个客户端持有锁的期间突然崩溃了，就会导致无法解锁，最后导致出现死锁的现象。

\*\*解决方式：\*\*

所以要有个超时的机制，在设置key的值时，需要加上有效时间，如果有效时间过期了，就会自动失效，就不会出现死锁。然后加锁的代码就会变成这样。

2. 实现方式2

但是聪明的同学肯定会问，有效时间设置多长，假如我的业务操作比有效时间长，我的业务代码还没执行完就自动给我解锁了，不就完蛋了吗。这个问题就有点棘手了，在网上也有很多讨论，第一种解决方法就是靠程序员自己去把握，预估一下业务代码需要执行的时间，然后设置有效期时间比执行时间长一些，保证不会因为自动解锁影响到客户端业务代码的执行。但是这并不是万全之策，比如网络抖动这种情况是无法预测的，也有可能导致业务代码执行的时间变长，所以并不安全。有一种方法比较靠谱一点，就是给锁续期。在Redisson框架实现分布式锁的思路，就使用watchDog机制实现锁的续期。当加锁成功后，同时开启守护线程，默认有效期是30秒，每隔10秒就会给锁续期到30秒，只要持有锁的客户端没有宕机，就能保证一直持有锁，直到业务代码执行完毕由客户端自己解锁，如果宕机了自然就在有效期失效后自动解锁。

\*\*存在的问题：\*\*

但是聪明的同学可能又会问，你这个锁只能加一次，不可重入。可重入锁意思是在外层使用锁之后，内层仍然可以使用，那么可重入锁的实现思路又是怎么样的呢？

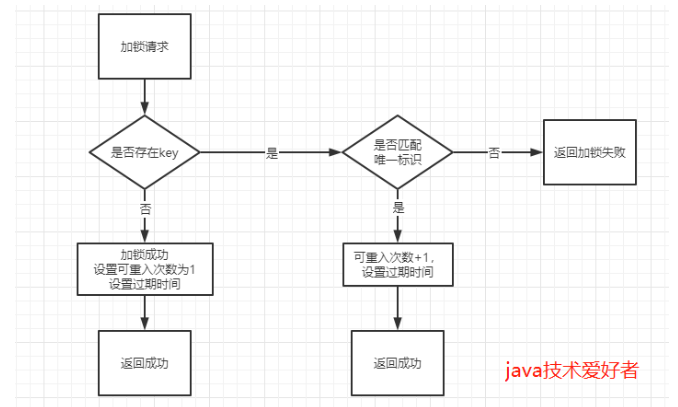
3.方式三

在Redisson实现可重入锁的思路，使用Redis的哈希表存储可重入次数，当加锁成功后，使用hset命令，value(重入次数)则是1。

"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; "

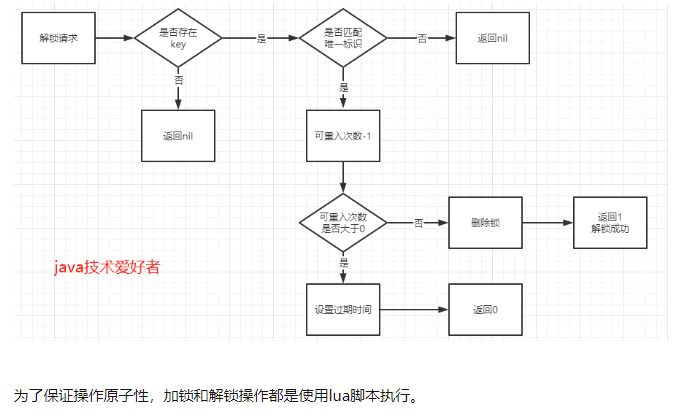
如果同一个客户端再次加锁成功，则使用hincrby自增加一。

"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]);"



解锁时，先判断可重复次数是否大于0，大于0则减一，否则删除键值，释放锁资源。

protected RFuture<Boolean> unlockInnerAsync(long threadId) { return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands.EVAL\_BOOLEAN, "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " + "[return nil](https://www.zhihu.com/search?q=return+nil&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A1749442787%7D);" + "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](https://www.zhihu.com/search?q=redis.call&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A1749442787%7D)('del', KEYS[1]); " + "redis.call('publish', KEYS[2], ARGV[1]); " + "return 1; "+ "end; " + "return nil;", Arrays.<Object>asList(getName(), getChannelName()), LockPubSub.UNLOCK\_MESSAGE, internalLockLeaseTime, getLockName(threadId)); }



问题：

上面的加锁方法是加锁后立即返回加锁结果，如果加锁失败的情况下，总不可能一直轮询尝试加锁，直到加锁成功为止，这样太过耗费性能。所以需要利用发布订阅的机制进行优化。

方式4：

步骤如下：

当加锁失败后，订阅锁释放的消息，自身进入阻塞状态。

当持有锁的客户端释放锁的时候，发布锁释放的消息。

当进入阻塞等待的其他客户端收到锁释放的消息后，解除阻塞等待状态，再次尝试加锁。