分布式锁

当多个进程不在同一个系统中，用分布式锁控制多个进程对资源的访问。当多个操作节点操作同一个资源节点时，如果其中一个操作节点获取到资源节点的锁，则其他操作节点无法操作资源节点。

为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

- **互斥性**：任意时刻，只能有一个客户端获取锁，不能同时有两个客户端获取到锁。
- **安全性**：锁只能被持有该锁的客户端删除，不能由其它客户端删除。
- **死锁**：获取锁的客户端因为某些原因（如 down 机等）而未能释放锁，其它客户端再也无法获取到该锁。
- **容错**：当部分节点（redis 节点等）down 机时，客户端仍然能够获取锁和释放锁。

分布式锁的几种解决方案：

分类	方案	实现原理	优点	缺点
基于数据库	基于mysql 表唯一索引	1.表增加唯一索引 2.加锁：执行insert语句，若报错，则表明加锁失败 3.解锁：执行delete语句	完全利用DB现有能力，实现简单	1.锁无超时自动失效机制，有死锁风险 2.不支持锁重入，不支持阻塞等待 3.操作数据库开销大，性能不高
基于MongoDB findAndModify原子操作	1.加锁：执行findAndModify原子命令查找document，若不存在则新增 2.解锁：删除document	实现也很容易，较基于MySQL唯一索引的方案，性能要好很多	1.大部分公司数据库用MySQL，可能缺乏相应的MongoDB运维、开发人员 2.锁无超时自动失效机制	
基于分布式协调系统	基于ZooKeeper	1.加锁：在/lock目录下创建临时有序节点，判断创建的节点序号是否最小。若是，则表示获取到锁；否，则则watch /lock目录下序号比自身小的前一个节点 2.解锁：删除节点	1.由zk保障系统高可用 2.Curator框架已原生支持系列分布式锁命令，使用简单	需单独维护一套zk集群，维保成本高
基于缓存	基于redis命令	1. 加锁：执行setnx，若成功再执行expire添加过期时间 2. 解锁：执行delete命令	实现简单，相比数据库和分布式系统的实现，该方案最轻，性能最好	1.setnx和expire分2步执行，非原子操作；若setnx执行成功，但expire执行失败，就可能出现死锁 2.delete命令存在误删除非当前线程持有的锁的可能 3.不支持阻塞等待、不可重入

基于redis Lua脚本能力	1. 加锁：执行SET lock_name random_value EX seconds NX 命令
	2. 解锁：执行Lua脚本，释放锁时验证random_value
	-- ARGV[1]为random_value, KEYS[1]为lock_name
	
	if redis.call("get", KEYS[1]) == ARGV[1] then
	  return redis.call("del",KEYS[1])
	else
	  return 0
	end
		同上；实现逻辑上也更严谨，除了单点问题，生产环境采用用这种方案，问题也不大。	不支持锁重入，不支持阻塞等待	

加锁过程

1. 执行lua脚本，尝试获取锁，如果未获取到，则订阅解锁消息，同时取得锁的剩余释放时间并阻塞自身，直到被唤醒或超时。

2. 锁被释放后，会广播解锁消息，解锁消息监听器释放信号，获取锁而阻塞的线程将被唤醒并尝试重新获取锁。

锁的释放机制

1. 为锁设置了到期时间，如果持锁服务宕机，到期后锁会被释放。
2. 为锁设置了到期时间，如果到期后持锁服务业务未能完成，则需要延长锁的有效期。

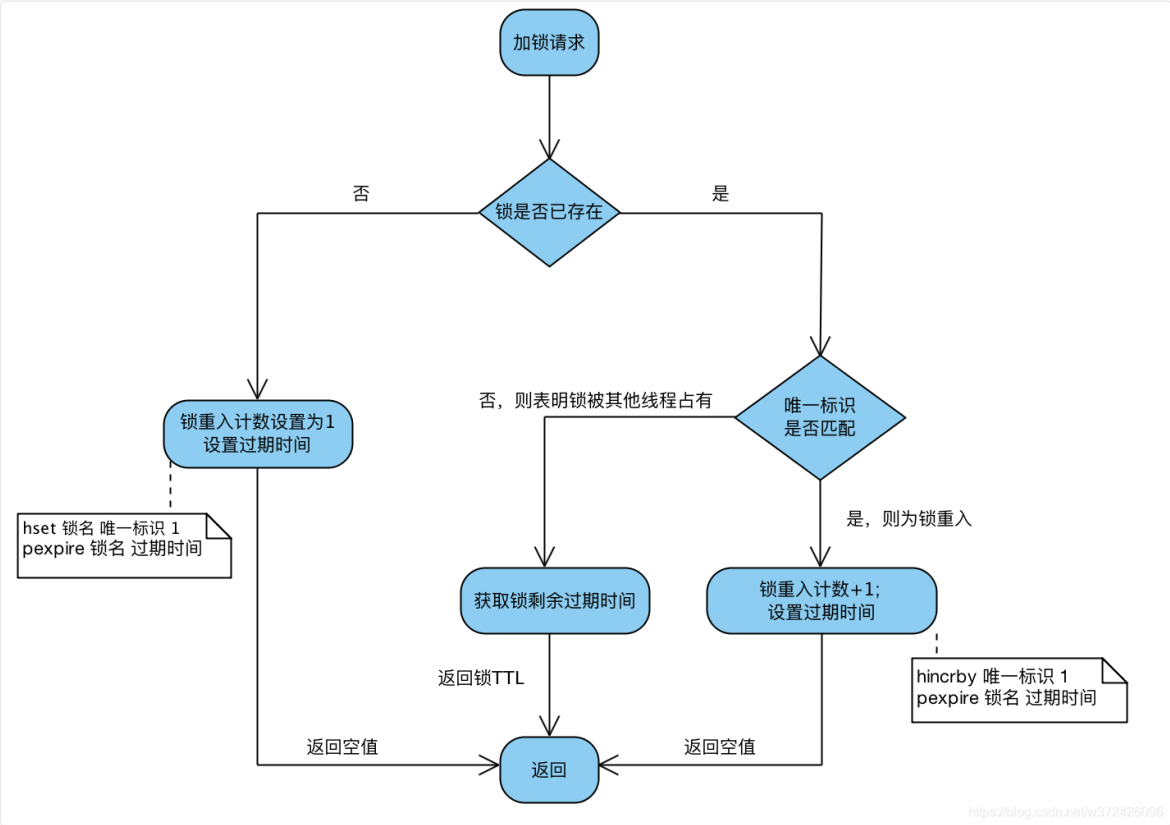
Redission 分布式锁

https://blog.csdn.net/asd051377305/article/details/108384490

Lua 加解锁

加锁

加锁流程



加锁脚本

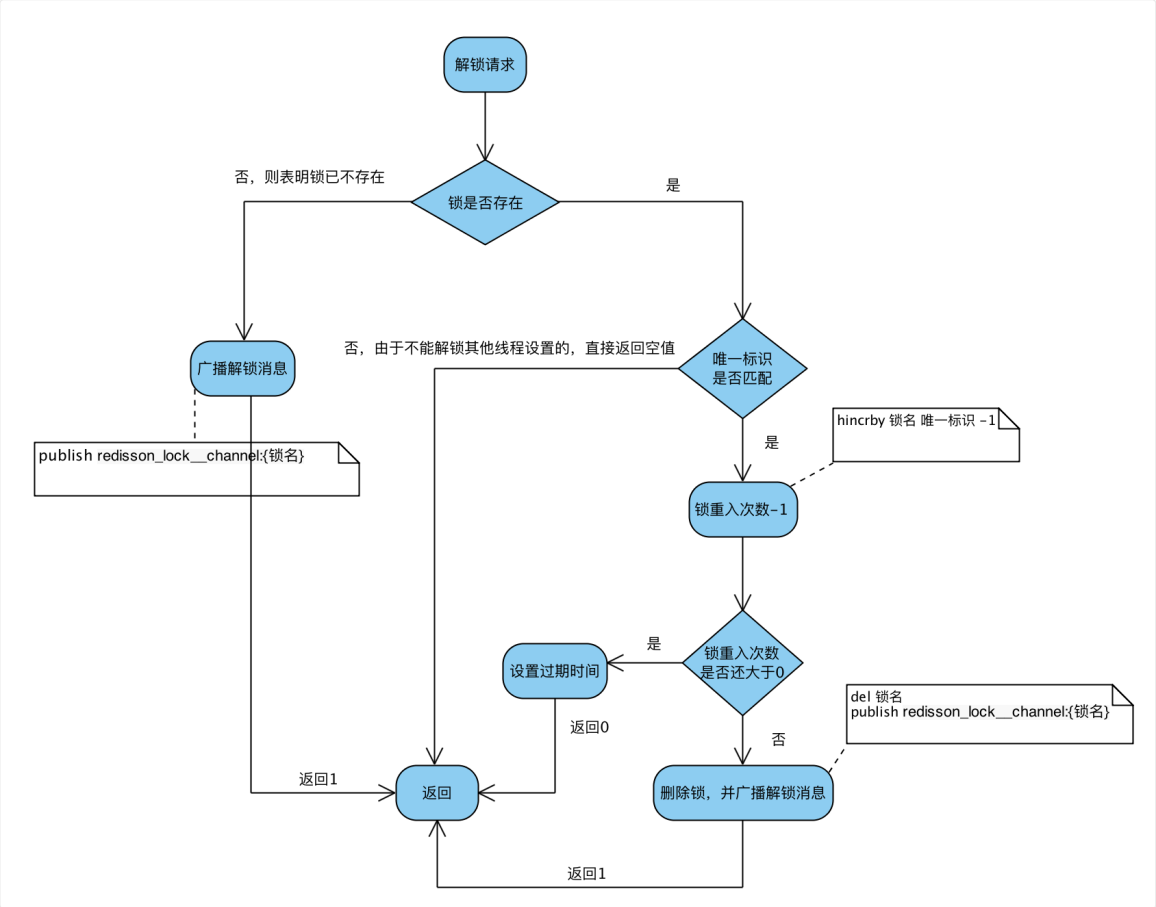
```lua
-- 若锁不存在：则新增锁，并设置锁重入计数为1、设置锁过期时间
if (redis.call('exists', KEYS[1]) == 0) then
    redis.call('hset', KEYS[1], ARGV[2], 1);
    redis.call('pexpire', KEYS[1], ARGV[1]);
    return nil;
end;

-- 若锁存在，且唯一标识也匹配：则表明当前加锁请求为锁重入请求，故锁重入计数+1，并再次设置锁过期时间
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;

-- 若锁存在，但唯一标识不匹配：表明锁是被其他线程占用，当前线程无权解他人的锁，直接返回锁剩余过期时间
return redis.call('pttl', KEYS[1]);
```

脚本参数

参数	示例值	含义
KEYS[1]	my_first_lock_name	锁名
ARGV[1]	60000	持有锁的有效时间：毫秒
ARGV[2]	58c62432-bb74-4d14-8a00-9908cc8b828f:1	唯一标识：获取锁时 set 的唯一值，实现上为 redisson 客户端 ID(UUID)+ 线程 ID

解锁

解锁流程



解锁脚本

```lua
-- 若锁不存在：则直接广播解锁消息，并返回1（广播是为了通知其他争抢锁阻塞住的线程，从阻塞中解除，并再次去争抢锁。）
if (redis.call('exists', KEYS[1]) == 0) then
    redis.call('publish', KEYS[2], ARGV[1]);
    return 1;
end;

-- 若锁存在，但唯一标识不匹配：则表明锁被其他线程占用，当前线程不允许解锁其他线程持有的锁
if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then
    return nil;
end;

-- 若锁存在，且唯一标识匹配：则先将锁重入计数减1
local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1);
if (counter > 0) then
    -- 锁重入计数减1后还大于0：表明当前线程持有的锁还有重入，不能进行锁删除操作，但可以友好地帮忙设置下过期时期
    redis.call('pexpire', KEYS[1], ARGV[2]);
    return 0;
else
    -- 锁重入计数已为0：间接表明锁已释放了。直接删除掉锁，并广播解锁消息，去唤醒那些争抢过锁但还处于阻塞中的线程
    redis.call('del', KEYS[1]);
    redis.call('publish', KEYS[2], ARGV[1]);
    return 1;
end;

```

```
25
    return nil;
```

脚本参数

参数	示例值	含义
KEYS[1]	my_first_lock_name	锁名
KEYS[2]	redisson_lock__channel:{my_first_lock_name}	解锁消息 PubSub 频道
ARGV[1]	0	redisson 定义 0 表示解锁消息
ARGV[2]	30000	设置锁的过期时间；默认值 30 秒
ARGV[3]	58c62432-bb74-4d14-8a00-9908cc8b828f:1	唯一标识；同加锁流程

基于 Redis Lua 脚本

加锁

Java

```java
public class RedisTool {

  /**
  * 尝试获取分布式锁
  * @param jedis Redis客户端
  * @param lockKey 锁
  * @param requestId 请求标识,用于判断是哪个请求加的锁
  * @param expireTime 超期时间
  * @return 是否获取成功
  */
  public static boolean getLock(Jedis jedis, String lockKey, String requestId, int expireTime) {
    // NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
    // 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
    String result = jedis.set(lockKey, requestId, "NX", "PX", expireTime);
    return "OK".equals(result);
  }

}
```

解锁：使用 lua 脚本

Java

```java
public class RedisTool {
  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 是否释放成功
  */
  public static boolean deleteLock(Jedis jedis, String lockKey, String requestId) {
    // 脚本用途：获取锁对应的value值，检查是否与requestId相等，如果相等则解锁。
    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));
    return 1L.equals(result)
}
```