

# 分布式锁的实现之 redis 篇

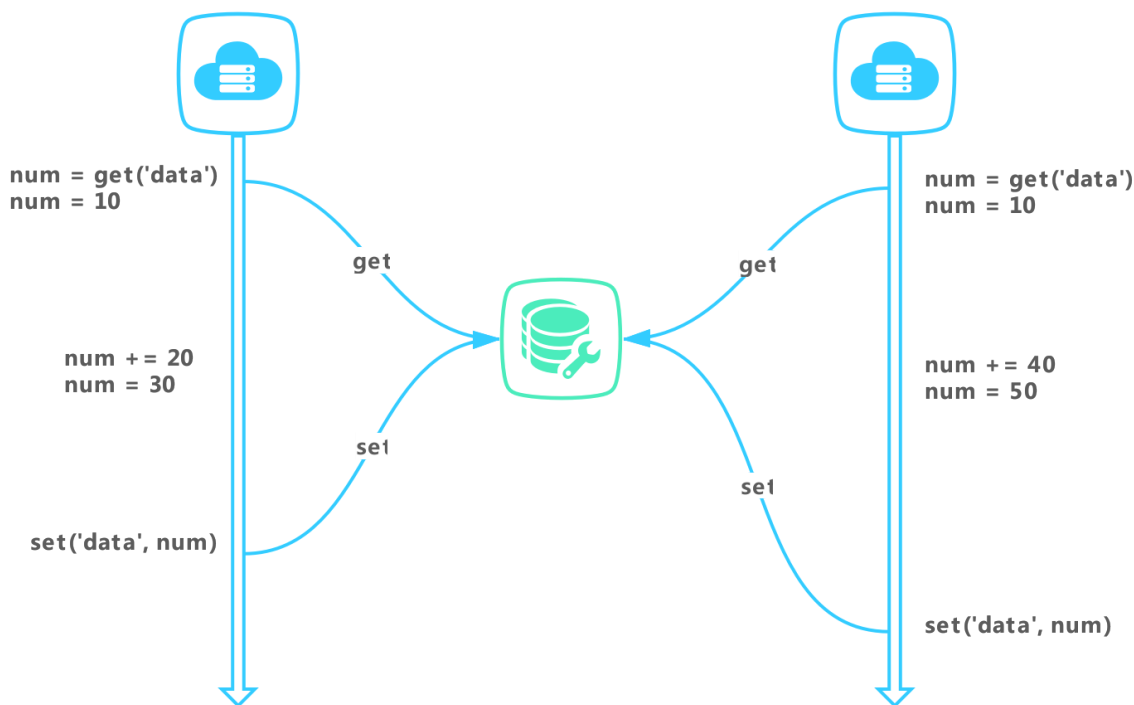
2019-12-17

## 分布式锁的实现之 redis 篇

[作者简介] 钟梦浩，信息部订单组研发工程师，目前主要负责小米订单中台业务。

### 一、引言

我们在系统中修改已有数据时，需要先读取，然后进行修改保存，此时很容易遇到并发问题。由于修改和保存不是原子操作，在并发场景下，部分对数据的操作可能会丢失。在单服务器系统我们常用本地锁来避免并发带来的问题，然而，当服务采用集群方式部署时，本地锁无法在多个服务器之间生效，这时候保证数据的一致性就需要分布式锁来实现。



### 二、实现

Redis 锁主要利用 Redis 的 `setnx` 命令。

- 加锁命令：SETNX key value，当键不存在时，对键进行设置操作并返回成功，否则返回失败。KEY 是锁的唯一标识，一般按业务来决定命名。
- 解锁命令：DEL key，通过删除键值对释放锁，以便其他线程可以通过 SETNX 命令来获取锁。
- 锁超时：EXPIRE key timeout，设置 key 的超时时间，以保证即使锁没有被显式释放，锁也可以在一定时间后自动释放，避免资源被永远锁住。

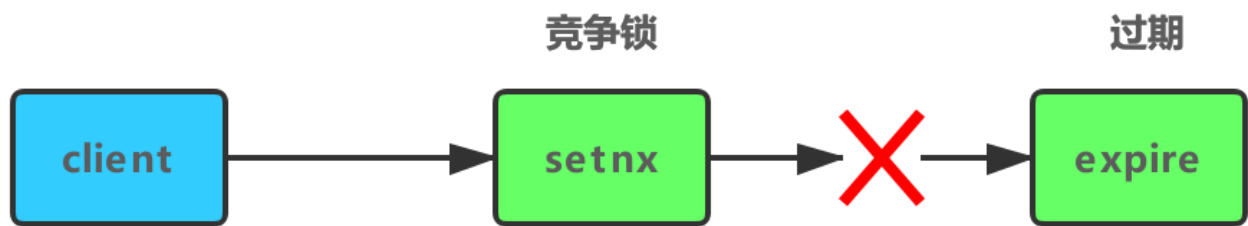
则加锁解锁伪代码如下：

```
4      //TODO 业务逻辑
5  } finally {
6      del(key)
7  }
8 }
```

上述锁实现方式存在一些问题：

## 1. SETNX 和 EXPIRE 非原子性

如果 SETNX 成功，在设置锁超时时间后，服务器挂掉、重启或网络问题等，导致 EXPIRE 命令没有执行，锁没有设置超时时间变成死锁。



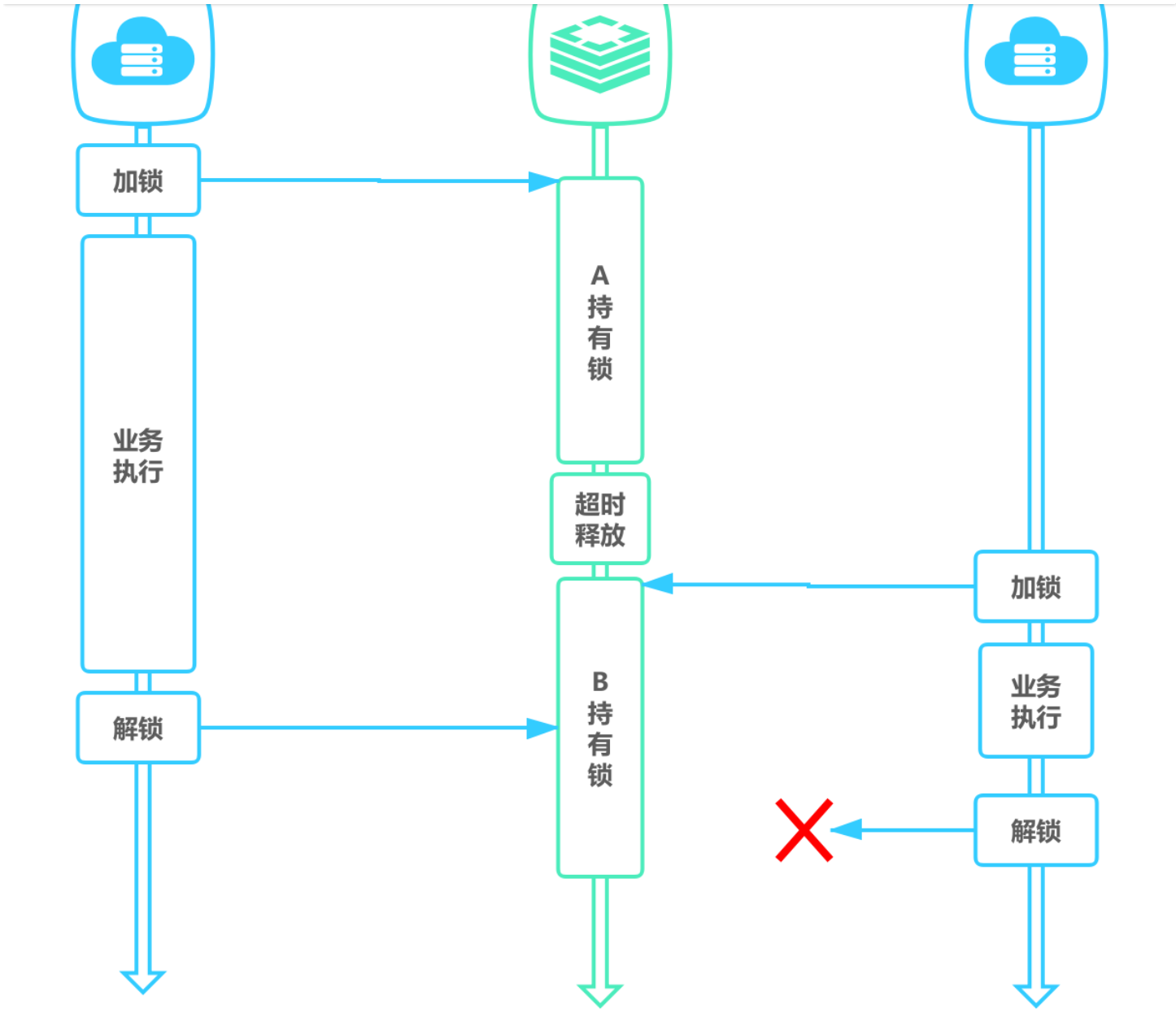
有很多开源代码来解决这个问题，比如使用 lua 脚本。示例：

```
1  if (redis.call('setnx', KEYS[1], ARGV[1]) < 1)
2  then return 0;
3  end;
4  redis.call('expire', KEYS[1], tonumber(ARGV[2]));
5  return 1;
6
7  // 使用实例
8  EVAL "if (redis.call('setnx',KEYS[1],ARGV[1]) < 1) then return 0; end; redis.call('expire',KEYS[1],tonumber(ARGV[2]));"
```

LUA

## 2. 锁误解除

如果线程 A 成功获取到了锁，并且设置了过期时间 30 秒，但线程 A 执行时间超过了 30 秒，锁过期自动释放，此时线程 B 获取到了锁；随后 A 执行完成，线程 A 使用 DEL 命令来释放锁，但此时线程 B 加的锁还没有执行完成，线程 A 实际释放的线程 B 加的锁。



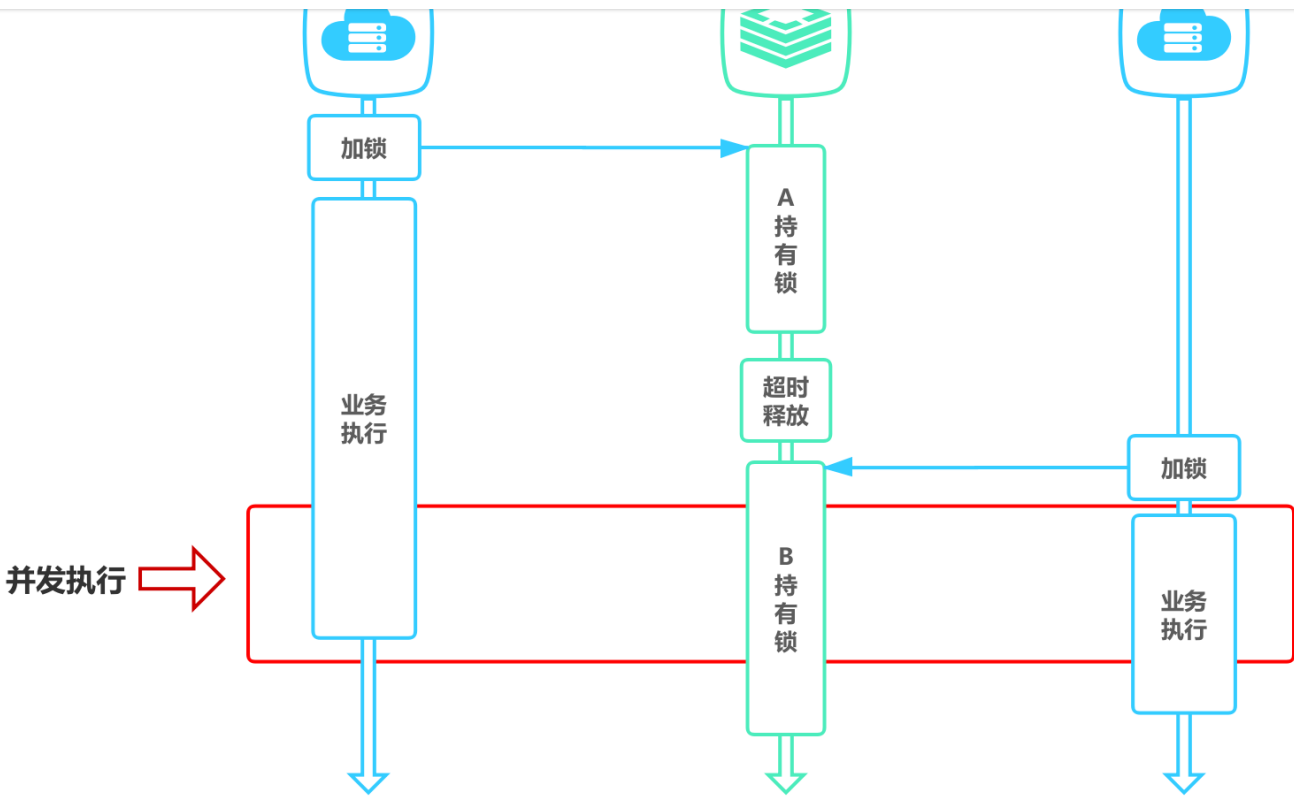
通过在 value 中设置当前线程加锁的标识，在删除之前验证 key 对应的 value 判断锁是否是当前线程持有。可生成一个 UUID 标识当前线程，使用 lua 脚本做验证标识和解锁操作。

LUA

```
1 // 加锁
2 String uuid = UUID.randomUUID().toString().replaceAll("-", "");
3 SET key uuid NX EX 30
4 // 解锁
5 if (redis.call('get', KEYS[1]) == ARGV[1])
6     then return redis.call('del', KEYS[1])
7 else return 0
8 end
```

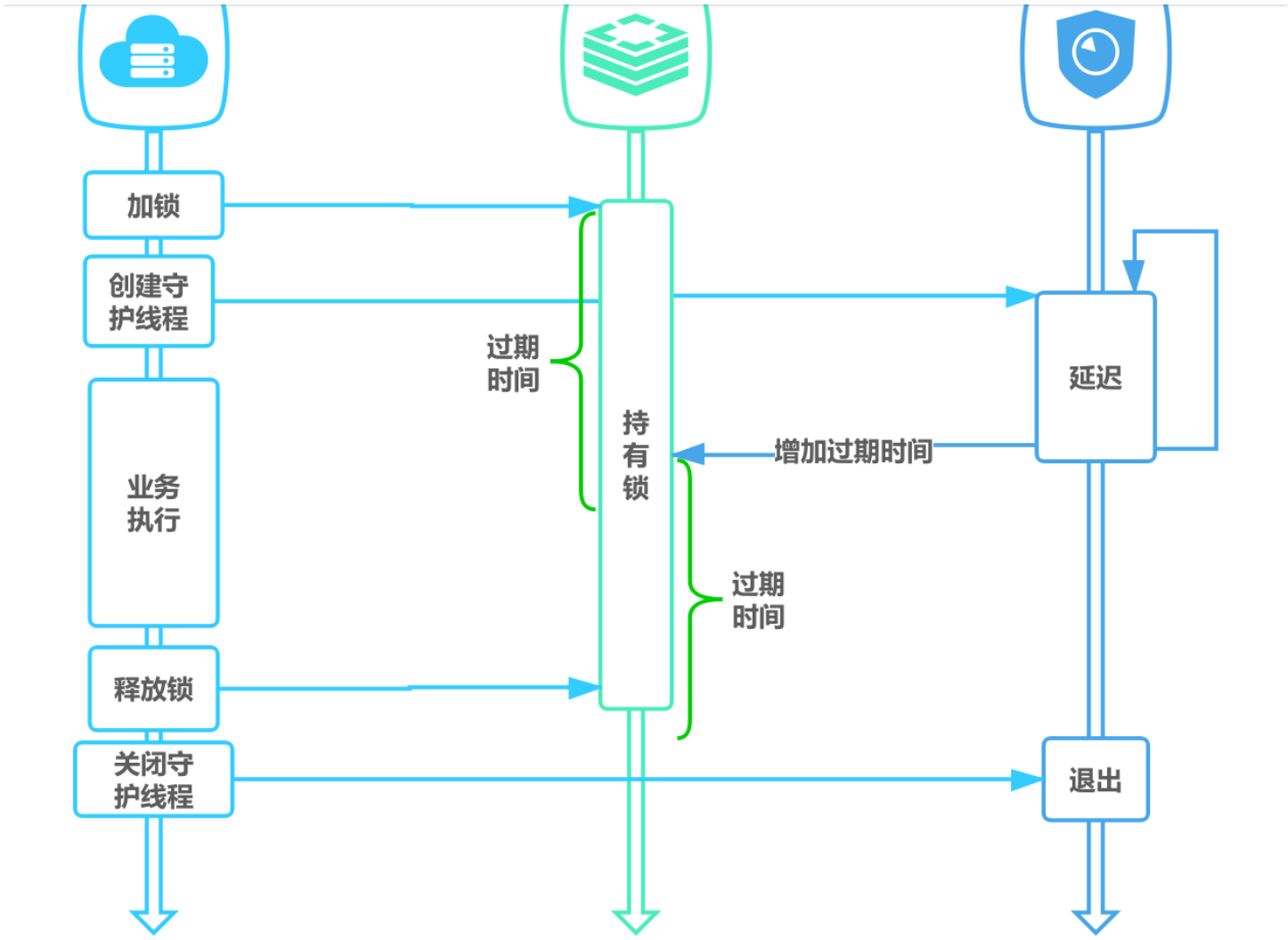
3. 超时解锁导致并发

如果线程 A 成功获取锁并设置过期时间 30 秒，但线程 A 执行时间超过了 30 秒，锁过期自动释放，此时线程 B 获取到了锁，线程 A 和线程 B 并发执行。



A、B 两个线程发生并发显然是不被允许的，一般有两种方式解决该问题：

- 将过期时间设置足够长，确保代码逻辑在锁释放之前能够执行完成。
- 为获取锁的线程增加守护线程，为将要过期但未释放的锁增加有效时间。



4. 不可重入

当线程在持有锁的情况下再次请求加锁，如果一个锁支持一个线程多次加锁，那么这个锁就是可重入的。如果一个不可重入锁被再次加锁，由于该锁已经被持有，再次加锁会失败。Redis 可通过对锁进行重入计数，加锁时加 1，解锁时减 1，当计数归 0 时释放锁。

在本地记录记录重入次数，如 Java 中使用 ThreadLocal 进行重入次数统计，简单示例代码：

JAVA

```
1 private static ThreadLocal<Map<String, Integer>> LOCKERS = ThreadLocal.withInitial(HashMap::new);
2 // 加锁
3 public boolean lock(String key) {
4     Map<String, Integer> lockers = LOCKERS.get();
5     if (lockers.containsKey(key)) {
6         lockers.put(key, lockers.get(key) + 1);
7         return true;
8     } else {
9         if (SET key uuid NX EX 30) {
10             lockers.put(key, 1);
11             return true;
12         }
13     }
14     return false;
15 }
16 // 解锁
17 public void unlock(String key) {
18     Map<String, Integer> lockers = LOCKERS.get();
19     if (lockers.getOrDefault(key, 0) <= 1) {
20         lockers.remove(key);
21         DEL key
22     } else {
23         lockers.put(key, lockers.get(key) - 1);
```

本地记录重入次数虽然高效，但如果考虑到过期时间和本地、Redis 一致性的问题，就会增加代码的复杂性。另一种方式是 Redis Map 数据结构来实现分布式锁，既存锁的标识也对重入次数进行计数。Redisson 加锁示例：

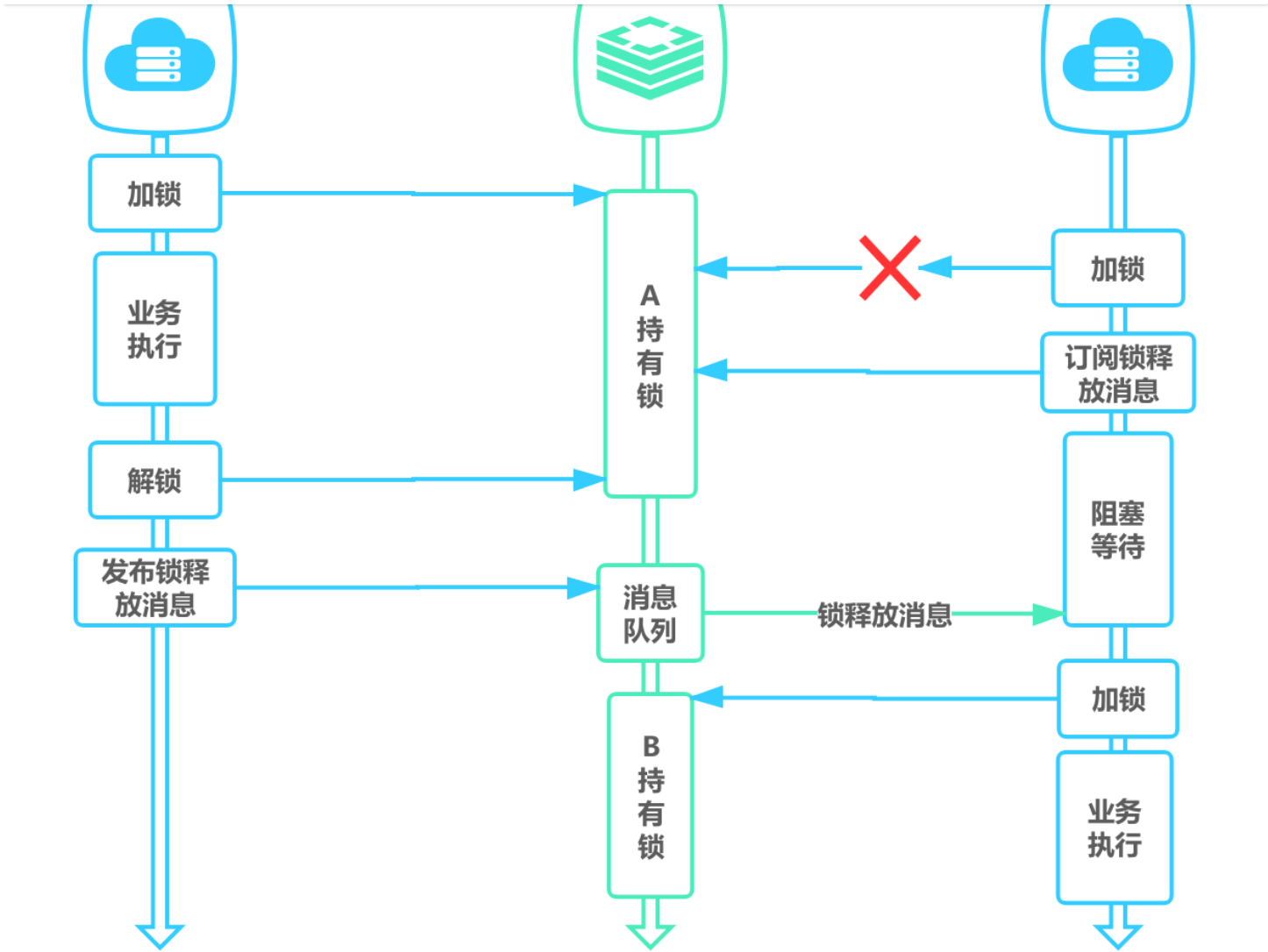
LUA

```
1 // 如果 lock_key 不存在
2 if (redis.call('exists', KEYS[1]) == 0)
3 then
4     // 设置 lock_key 线程标识 1 进行加锁
5     redis.call('hset', KEYS[1], ARGV[2], 1);
6     // 设置过期时间
7     redis.call('pexpire', KEYS[1], ARGV[1]);
8     return nil;
9 end;
10 // 如果 lock_key 存在且线程标识是当前欲加锁的线程标识
11 if (redis.call('hexists', KEYS[1], ARGV[2]) == 1)
12     // 自增
13     then redis.call('hincrby', KEYS[1], ARGV[2], 1);
14     // 重置过期时间
15     redis.call('pexpire', KEYS[1], ARGV[1]);
16     return nil;
17 end;
18 // 如果加锁失败，返回锁剩余时间
19 return redis.call('pttl', KEYS[1]);
```

## 5. 无法等待锁释放

上述命令执行都是立即返回的，如果客户端可以等待锁释放就无法使用。

- 可以通过客户端轮询的方式解决该问题，当未获取到锁时，等待一段时间重新获取锁，直到成功获取锁或等待超时。这种方式比较消耗服务器资源，当并发量比较大时，会影响服务器的效率。
- 另一种方式是使用 Redis 的发布订阅功能，当获取锁失败时，订阅锁释放消息，获取锁成功后释放时，发送锁释放消息。如下：

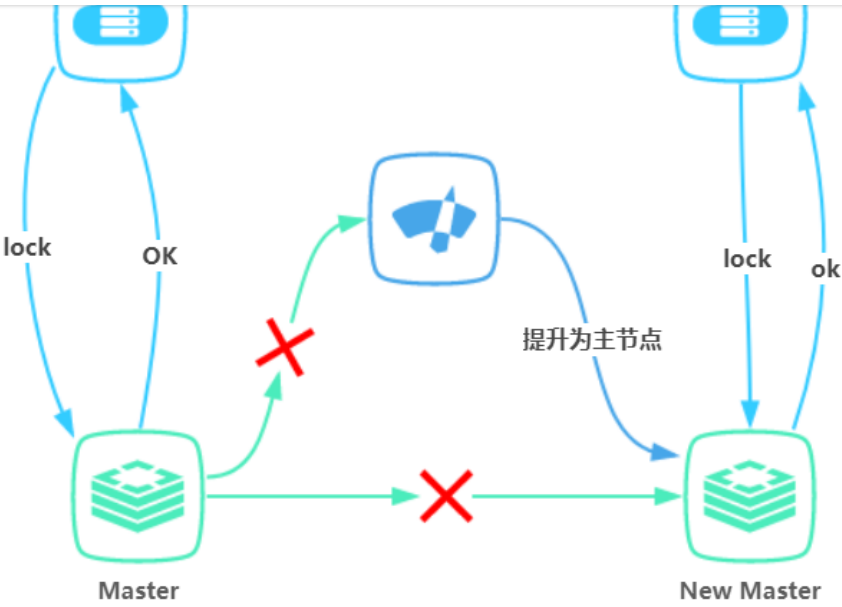


### 三、集群

#### 1. 主备切换

为了保证 Redis 的可用性，一般采用主从方式部署。主从数据同步有异步和同步两种方式，Redis 将指令记录在本地内存 buffer 中，然后异步将 buffer 中的指令同步到从节点，从节点一边执行同步的指令流来达到和主节点一致的状态，一边向主节点反馈同步情况。

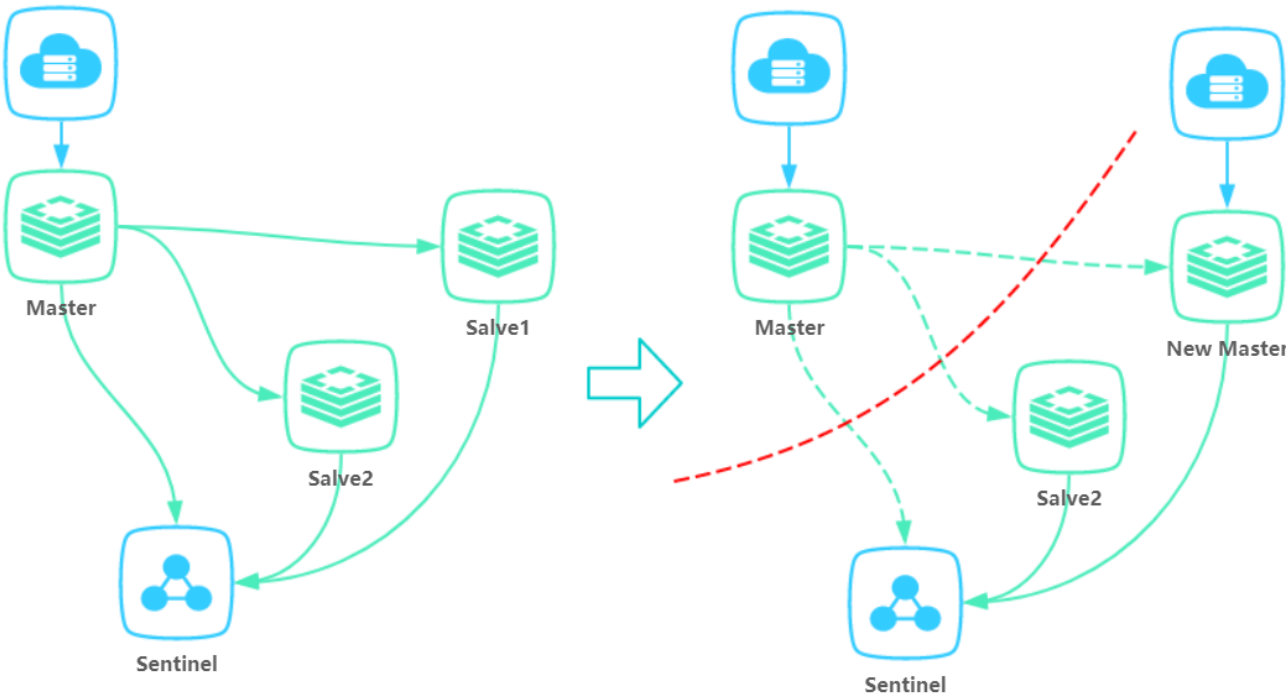
在包含主从模式的集群部署方式中，当主节点挂掉时，从节点会取而代之，但客户端无明显感知。当客户端 A 成功加锁，指令还未同步，此时主节点挂掉，从节点提升为主节点，新的主节点没有锁的数据，当客户端 B 加锁时就会成功。



2. 集群脑裂

集群脑裂指因为网络问题，导致 Redis master 节点跟 slave 节点和 sentinel 集群处于不同的网络分区，因为 sentinel 集群无法感知到 master 的存在，所以将 slave 节点提升为 master 节点，此时存在两个不同的 master 节点。Redis Cluster 集群部署方式同理。

当不同的客户端连接不同的 master 节点时，两个客户端可以同时拥有同一把锁。如下：



四、结语

Redis 以其高性能著称，但使用其实现分布式锁来解决并发仍存在一些困难。Redis 分布式锁只能作为一种缓解并发的手段，如果要完全解决并发问题，仍需要数据库的防并发手段。

参考：

- 1.“Redis 分布式锁的正确实现方式（Java 版）” <https://mp.weixin.qq.com/s/qJK61ew0kCEvvXrqb7-RSg>
- 2.“漫画：什么是分布式锁？” [https://mp.weixin.qq.com/s/8fdBKAYHZrfHmSajXT\\_dnA](https://mp.weixin.qq.com/s/8fdBKAYHZrfHmSajXT_dnA)



5. 《逆流而上·阿里巴巴技术成长之路》

作者

钟梦浩，小米信息部订单组

招聘

信息部是小米公司整体系统规划建设的核心部门，支撑公司国内外的线上线下销售服务体系、供应链体系、ERP 体系、内网 OA 体系、数据决策体系等精细化管控的执行落地工作，服务小米内部所有的业务部门以及 40 家生态链公司。

同时部门承担大数据基础平台研发和微服务体系建设落，语言涉及 Java、Go，长年虚位以待对大数据处理、大型电商后端系统、微服务落地有深入理解和实践的各路英雄。

欢迎投递简历：jin.zhang(a)xiaomi.com（武汉）

Tags: redis 分布式锁

← [走进 NSQ 源码细节](#)

[一次线上线程池任务问题处理历程](#) →



扫描二维码，分享此文章

♥ Like

Issue Page

Write

Preview

Login with GitHub

Leave a comment

Comment

Error: Comments Not Initialized

Powered by [Gitment](#)