redis 分布式锁的 5个坑，真是又大又深

Original 程序员内点事 程序员内点事 2020-04-22

引言

最近项目上线的频率颇高，连着几天加班熬夜，身体有点吃不消精神也有些萎靡，无奈业务方催的紧，工期就在眼前只能硬着头皮上了。脑子浑浑噩噩的时候，写的就不能叫代码，可以直接叫做 Bug 。我就熬夜写了一个 bug 被骂惨了。

由于是做商城业务，要频繁的对商品库存进行扣减，应用是集群部署，为避免并发造成库存 超买超卖 等问题，采用 redis 分布式锁加以控制。本以为给扣库存的代码加上锁 lock.tryLock 就万事大吉了

```
 /**
  * @author xiaofu
  * @description 扣减库存
  * @date 2020/4/21 12:10
  */
public String stockLock() {
    RLock lock = redissonClient.getLock("stockLock");
    try {
        /**
         * 获取锁
         */
        if (lock.tryLock(10, TimeUnit.SECONDS)) {
            /**
             * 查询库存数
             */
            Integer stock = Integer.valueOf(stringRedisTemplate.opsForValue().get(
            /**
             * 扣减库存
             */
            if (stock > 0) {
                stock = stock - 1;
                stringRedisTemplate.opsForValue().set("stockCount", stock.toString
                LOGGER.info("库存扣减成功，剩余库存数量：{}", stock);
            } else {
                LOGGER.info("库存不足~");
            }
```

```
            } else {
                LOGGER.info("未获取到锁业务结束..");
            }
        } catch (Exception e) {
            LOGGER.info("处理异常", e);
        } finally {
            lock.unlock();
        }
        return "ok";
    }
```

结果业务代码执行完以后我忘了释放锁 lock.unlock() ，导致 redis 线程池被打满， redis 服务大面积故障，造成库存数据扣减混乱，被领导一顿臭骂，这个月绩效~ 哎·~。

随着 使用 redis 锁的时间越长，我发现 redis 锁的坑远比想象中要多。就算在面试题当中 redis 分布式锁的出镜率也比较高，比如：“用锁遇到过哪些问题？”，“又是如何解决的？”基本都是一套连招问出来的。

今天就分享一下我用 redis 分布式锁的踩坑日记，以及一些解决方案，和大家一起共勉。

一、锁未被释放

这种情况是一种低级错误，就是我上边犯的错，由于当前线程 获取到 redis 锁，处理完业务后未及时释放锁，导致其它线程会一直尝试获取锁阻塞，例如：用 Jedis 客户端会报如下的错误信息

```
redis.clients.jedis.exceptions.JedisConnectionException: Could not get a resource from
```

redis线程池 已经没有空闲线程来处理客户端命令。

解决的方法也很简单，只要我们细心一点，拿到锁的线程处理完业务及时释放锁，如果是重入锁未拿到锁后，线程可以释放当前连接并且 sleep 一段时间。

```
    public void lock() {
        while (true) {
            boolean flag = this.getLock(key);
            if (flag) {
                  TODO .........
            } else {
                  // 释放当前redis连接
```

```
                redis.close();
                // 休眠1000毫秒
                sleep(1000);
            }
        }
    }
```

二、B的锁被A给释放了

我们知道 Redis 实现锁的原理在于 SETNX 命令。当 key 不存在时将 key 的值设为 value ，返回值为 1 ；若给定的 key 已经存在，则 SETNX 不做任何动作，返回值为 0 。

```
SETNX key value
```

我们来设想一下这个场景： A 、 B 两个线程来尝试给 key myLock 加锁， A线程 先拿到锁（假如锁 3秒 后过期）， B线程 就在等待尝试获取锁，到这一点毛病没有。

那如果此时业务逻辑比较耗时，执行时间已经超过 redis 锁过期时间，这时 A线程 的锁自动释放（删除 key ）， B线程 检测到 myLock 这个 key 不存在，执行 SETNX 命令也拿到了锁。

但是，此时 A线程 执行完业务逻辑之后，还是会去释放锁（删除 key ），这就导致 B线程 的锁被 A线程 给释放了。

为避免上边的情况，一般我们在每个线程加锁时要带上自己独有的 value 值来标识，只释放指定 value 的 key ，否则就会出现释放锁混乱的场景。

三、数据库事务超时

emm~ 聊 redis 锁咋还扯到数据库事务上来了？别着急往下看，看下边这段代码：

```
    @Transaction
    public void lock() {

        while (true) {
            boolean flag = this.getLock(key);
            if (flag) {
```

```
                insert();
            }
        }
    }
```

给这个方法添加一个 @Transaction 注解开启事务，如代码中抛出异常进行回滚，要知道数据库事务可是有超时时间限制的，并不会无条件的一直等一个耗时的数据库操作。

比如：我们解析一个大文件，再将数据存入到数据库，如果执行时间太长，就会导致事务超时自动回滚。

一旦你的 key 长时间获取不到锁，获取锁 等待的时间 远超过数据库事务 超时时间 ，程序就会报异常。

一般为解决这种问题，我们就需要将数据库事务改为手动提交、回滚事务。

```
    @Autowired
    DataSourceTransactionManager dataSourceTransactionManager;

    @Transaction
    public void lock() {
        //手动开启事务
        TransactionStatus transactionStatus = dataSourceTransactionManager.getTransact
        try {
            while (true) {
                boolean flag = this.getLock(key);
                if (flag) {
                    insert();
                    //手动提交事务
                    dataSourceTransactionManager.commit(transactionStatus);
                }
            }
        } catch (Exception e) {
            //手动回滚事务
            dataSourceTransactionManager.rollback(transactionStatus);
        }
    }
```

四、锁过期了，业务还没执行完

这种情况和我们上边提到的第二种比较类似，但解决思路上略有不同。

同样是 redis 分布式锁过期，而业务逻辑没执行完的场景，不过，这里换一种思路想问题，**把 redis 锁的过期时间再弄长点不就解决了吗?**

那还是有问题，我们可以在加锁的时候，手动调长 redis 锁的过期时间，可这个时间多长合适？业务逻辑的执行时间是不可控的，调的过长又会影响操作性能。

要是 redis 锁的过期时间能够自动续期就好了。

为了解决这个问题我们使用 redis 客户端 redisson ， redisson 很好的解决了 redis 在分布式环境下的一些棘手问题，它的宗旨就是让使用者减少对 Redis 的关注，将更多精力用在处理业务逻辑上。

redisson 对分布式锁做了很好封装，只需调用 API 即可。

```
1    RLock lock = redissonClient.getLock("stockLock");
```

redisson 在加锁成功后，会注册一个定时任务监听这个锁，每隔10秒就去查看这个锁，如果还持有锁，就对 过期时间 进行续期。默认过期时间30秒。这个机制也被叫做：“ 看门狗 ”，这名字。。。

举例子：假如加锁的时间是30秒，过10秒检查一次，一旦加锁的业务没有执行完，就会进行一次续期，把锁的过期时间再次重置成30秒。

通过分析下边 redisson 的源码实现可以发现，不管是 加锁 、 解锁 、 续约 都是客户端把一些复杂的业务逻辑，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 。

```
1  @Slf4j
2  @Service
3  public class RedisDistributionLockPlus {
4
5      /**
6       * 加锁超时时间，单位毫秒， 即：加锁时间内执行完操作，如果未完成会有并发现象
7       */
8      private static final long DEFAULT_LOCK_TIMEOUT = 30;
9
10     private static final long TIME_SECONDS_FIVE = 5 ;
11
12     /**
13      * 每个key的过期时间 {@link LockContent}
14      */
15     private Map<String, LockContent> lockContentMap = new ConcurrentHashMap<>(512);
16
17     /**
18      * redis执行成功的返回
19      */
```

```
20      private static final Long EXEC_SUCCESS = 1L;
21
22      /**
23       * 获取锁lua脚本， k1：获锁key，k2：续约耗时key，arg1:requestId，arg2：超时时间
24       */
25      private static final String LOCK_SCRIPT = "if redis.call('exists', KEYS[2]) == 1
26              "if redis.call('exists', KEYS[1]) == 0 then " +
27                 "local t = redis.call('set', KEYS[1], ARGV[1], 'EX', ARGV[2]) " +
28                 "for k, v in pairs(t) do " +
29                   "if v == 'OK' then return tonumber(ARGV[2]) end " +
30                 "end " +
31              "return 0 end";
32
33      /**
34       * 释放锁lua脚本，k1：获锁key，k2：续约耗时key，arg1:requestId，arg2：业务耗时 arg3:
35       */
36      private static final String UNLOCK_SCRIPT = "if redis.call('get', KEYS[1]) == ARG
37              "local ctime = tonumber(ARGV[2]) " +
38              "local biz_timeout = tonumber(ARGV[3]) " +
39              "if ctime > 0 then  " +
40                 "if redis.call('exists', KEYS[2]) == 1 then " +
41                     "local avg_time = redis.call('get', KEYS[2]) " +
42                     "avg_time = (tonumber(avg_time) * 8 + ctime * 2)/10 " +
43                     "if avg_time >= biz_timeout - 5 then redis.call('set', KEYS[2], av
44                     "else redis.call('del', KEYS[2]) end " +
45                 "elseif ctime > biz_timeout -5 then redis.call('set', KEYS[2], ARGV[2]
46              "end " +
47              "return redis.call('del', KEYS[1]) " +
48              "else return 0 end";
49      /**
50       * 续约lua脚本
51       */
52      private static final String RENEW_SCRIPT = "if redis.call('get', KEYS[1]) == ARGV
53
54
55      private final StringRedisTemplate redisTemplate;
56
57      public RedisDistributionLockPlus(StringRedisTemplate redisTemplate) {
58          this.redisTemplate = redisTemplate;
59          ScheduleTask task = new ScheduleTask(this, lockContentMap);
60          // 启动定时任务
61          ScheduleExecutor.schedule(task, 1, 1, TimeUnit.SECONDS);
62      }
63
64      /**
65       * 加锁
66       * 取到锁加锁，取不到锁一直等待知道获得锁
67       *
68       * @param lockKey
69       * @param requestId 全局唯一
70       * @param expire    锁过期时间，单位秒
71       * @return
72       */
73      public boolean lock(String lockKey, String requestId, long expire) {
```

```
74          log.info("开始执行加锁，lockKey ={}，requestId={}", lockKey, requestId);
75          for (; ; ) {
76              // 判断是否已经有线程持有锁，减少redis的压力
77              LockContent lockContentOld = lockContentMap.get(lockKey);
78              boolean unLocked = null == lockContentOld;
79              // 如果没有被锁，就获取锁
80              if (unLocked) {
81                  long startTime = System.currentTimeMillis();
82                  // 计算超时时间
83                  long bizExpire = expire == 0L ? DEFAULT_LOCK_TIMEOUT : expire;
84                  String lockKeyRenew = lockKey + "_renew";
85
86                  RedisScript<Long> script = RedisScript.of(LOCK_SCRIPT, Long.class);
87                  List<String> keys = new ArrayList<>();
88                  keys.add(lockKey);
89                  keys.add(lockKeyRenew);
90                  Long lockExpire = redisTemplate.execute(script, keys, requestId, Long
91                  if (null != lockExpire && lockExpire > 0) {
92                      // 将锁放入map
93                      LockContent lockContent = new LockContent();
94                      lockContent.setStartTime(startTime);
95                      lockContent.setLockExpire(lockExpire);
96                      lockContent.setExpireTime(startTime + lockExpire * 1000);
97                      lockContent.setRequestId(requestId);
98                      lockContent.setThread(Thread.currentThread());
99                      lockContent.setBizExpire(bizExpire);
100                     lockContent.setLockCount(1);
101                     lockContentMap.put(lockKey, lockContent);
102                     log.info("加锁成功，lockKey ={}，requestId={}", lockKey, requestId
103                     return true;
104                 }
105             }
106             // 重复获取锁，在线程池中由于线程复用，线程相等并不能确定是该线程的锁
107             if (Thread.currentThread() == lockContentOld.getThread()
108                     && requestId.equals(lockContentOld.getRequestId())){
109                 // 计数 +1
110                 lockContentOld.setLockCount(lockContentOld.getLockCount()+1);
111                 return true;
112             }
113
114             // 如果被锁或获取锁失败，则等待100毫秒
115             try {
116                 TimeUnit.MILLISECONDS.sleep(100);
117             } catch (InterruptedException e) {
118                 // 这里用lombok 有问题
119                 log.error("获取redis 锁失败，lockKey ={}，requestId={}", lockKey, requ
120                 return false;
121             }
122         }
123     }
124
125
126     /**
127      * 解锁
```

```
128         *
129         * @param lockKey
130         * @param lockValue
131         */
132        public boolean unlock(String lockKey, String lockValue) {
133            String lockKeyRenew = lockKey + "_renew";
134            LockContent lockContent = lockContentMap.get(lockKey);
135
136            long consumeTime;
137            if (null == lockContent) {
138                consumeTime = 0L;
139            } else if (lockValue.equals(lockContent.getRequestId())) {
140                int lockCount = lockContent.getLockCount();
141                // 每次释放锁， 计数 -1，减到0时删除redis上的key
142                if (--lockCount > 0) {
143                    lockContent.setLockCount(lockCount);
144                    return false;
145                }
146                consumeTime = (System.currentTimeMillis() - lockContent.getStartTime()) /
147            } else {
148                log.info("释放锁失败，不是自己的锁。");
149                return false;
150            }
151
152            // 删除已完成key，先删除本地缓存，减少redis压力，分布式锁，只有一个，所以这里不加锁
153            lockContentMap.remove(lockKey);
154
155            RedisScript<Long> script = RedisScript.of(UNLOCK_SCRIPT, Long.class);
156            List<String> keys = new ArrayList<>();
157            keys.add(lockKey);
158            keys.add(lockKeyRenew);
159
160            Long result = redisTemplate.execute(script, keys, lockValue, Long.toString(co
161                    Long.toString(lockContent.getBizExpire()));
162            return EXEC_SUCCESS.equals(result);
163
164        }
165
166        /**
167         * 续约
168         *
169         * @param lockKey
170         * @param lockContent
171         * @return true:续约成功，false:续约失败（1、续约期间执行完成，锁被释放 2、不是自己的锁
172         */
173        public boolean renew(String lockKey, LockContent lockContent) {
174
175            // 检测执行业务线程的状态
176            Thread.State state = lockContent.getThread().getState();
177            if (Thread.State.TERMINATED == state) {
178                log.info("执行业务的线程已终止,不再续约 lockKey ={}, lockContent={}", lockKe
179                return false;
180            }
181
```

```
182             String requestId = lockContent.getRequestId();
183             long timeOut = (lockContent.getExpireTime() - lockContent.getStartTime()) / 1
184 
185             RedisScript<Long> script = RedisScript.of(RENEW_SCRIPT, Long.class);
186             List<String> keys = new ArrayList<>();
187             keys.add(lockKey);
188 
189             Long result = redisTemplate.execute(script, keys, requestId, Long.toString(ti
190             log.info("续约结果，True成功，False失败 lockKey ={}, result={}", lockKey, EXEC_
191             return EXEC_SUCCESS.equals(result);
192         }
193 
194 
195         static class ScheduleExecutor {
196 
197             public static void schedule(ScheduleTask task, long initialDelay, long period
198                 long delay = unit.toMillis(initialDelay);
199                 long period_ = unit.toMillis(period);
200                 // 定时执行
201                 new Timer("Lock-Renew-Task").schedule(task, delay, period_);
202             }
203         }
204 
205         static class ScheduleTask extends TimerTask {
206 
207             private final RedisDistributionLockPlus redisDistributionLock;
208             private final Map<String, LockContent> lockContentMap;
209 
210             public ScheduleTask(RedisDistributionLockPlus redisDistributionLock, Map<Stri
211                 this.redisDistributionLock = redisDistributionLock;
212                 this.lockContentMap = lockContentMap;
213             }
214 
215             @Override
216             public void run() {
217                 if (lockContentMap.isEmpty()) {
218                     return;
219                 }
220                 Set<Map.Entry<String, LockContent>> entries = lockContentMap.entrySet();
221                 for (Map.Entry<String, LockContent> entry : entries) {
222                     String lockKey = entry.getKey();
223                     LockContent lockContent = entry.getValue();
224                     long expireTime = lockContent.getExpireTime();
225                     // 减少线程池中任务数量
226                     if ((expireTime - System.currentTimeMillis())/ 1000 < TIME_SECONDS_FI
227                         //线程池异步续约
228                         ThreadPool.submit(() -> {
229                             boolean renew = redisDistributionLock.renew(lockKey, lockCont
230                             if (renew) {
231                                 long expireTimeNew = lockContent.getStartTime() + (expire
232                                 lockContent.setExpireTime(expireTimeNew);
233                             } else {
234                                 // 续约失败，说明已经执行完 OR redis 出现问题
235                                 lockContentMap.remove(lockKey);
```

```
                        }
                    });
                }
            }
        }
    }
}
```

五、redis主从复制的坑

redis 高可用最常见的方案就是 主从复制 （master-slave），这种模式也给 redis分布式锁 挖了一坑。

redis cluster 集群环境下，假如现在 A客户端 想要加锁，它会根据路由规则选择一台 master 节点写入 key mylock ，在加锁成功后， master 节点会把 key 异步复制给对应的 slave 节点。

如果此时 redis master 节点宕机，为保证集群可用性，会进行 主备切换 ， slave 变为了 redis master 。 B客户端 在新的 master 节点上加锁成功，而 A客户端 也以为自己还是成功加了锁的。

此时就会导致同一时间内多个客户端对一个分布式锁完成了加锁，导致各种脏数据的产生。

至于解决办法嘛，目前看还没有什么根治的方法，只能尽量保证机器的稳定性，减少发生此事件的概率。

总结

上面就是我在使用 Redis 分布式锁时遇到的一些坑，有点小感慨，经常用一个方法填上这个坑，没多久就发现另一个坑又出来了，其实根本没有什么十全十美的解决方案，哪有什么银弹，只不过是在权衡利弊后，选一个在接受范围内的折中方案而已。

如有不严谨、错误之处还望温柔指正，共同进步！

END

和一些志同道合的小伙伴们，共同建了一个技术交流群，一起探讨技术、分享技术资料，旨在共同学习进步，如果感兴趣就扫码加入我们吧！

小福利：

获取到一些课 ， 嘘~， **免费** 送给小伙伴们。关注公众号

往期精彩回顾

为了不复制粘贴，我被逼着学会了JAVA爬虫
一口气说出 9种 分布式ID生成方式，面试官有点懵了
面试总被问分库分表怎么办？这些知识点你要懂
基于 Java 实现的人脸识别功能（附源码）
面试被问分布式ID怎么办？滴滴（Tinyid）甩给他
9种分布式ID生成之美团（Leaf）实战

Modified on 2020/04/22

People who liked this content also liked

拿捏！隔离级别、幻读、Gap Lock、Next-Key Lock
艾小仙