



基于Redis的分布式锁实现

📅 2019-04-25 📁 REDIS 🏷️ #REDIS, 分布式

Catalogue

1. 前言
2. 分布式锁概览
 - 2.1. 分布式锁的特点
 - 2.2. 分布式锁的实现方式
3. Redis的分布式锁实现
 - 3.1. 1. 利用setnx+expire命令 (错误的做法)
 - 3.2. 2. 使用Lua脚本 (包含setnx和expire两条指令)
 - 3.3. 3. 使用 set key value [EX seconds][PX milliseconds][NX|XX] 命令 (正确做法)
 - 3.3.1. 关于释放锁的问题
 - 3.4. 4. Redlock算法 与 Redisson 实现
 - 3.4.1. Redisson实现简单分布式锁
4. Redis实现的分布式锁轮子
 - 4.1. 1. 自定义注解
 - 4.2. 2. AOP拦截器实现
 - 4.3. 3. Redis实现分布式锁核心类
 - 4.4. 4. Controller层控制
5. 小结
6. 参考资料 & 鸣谢

前言

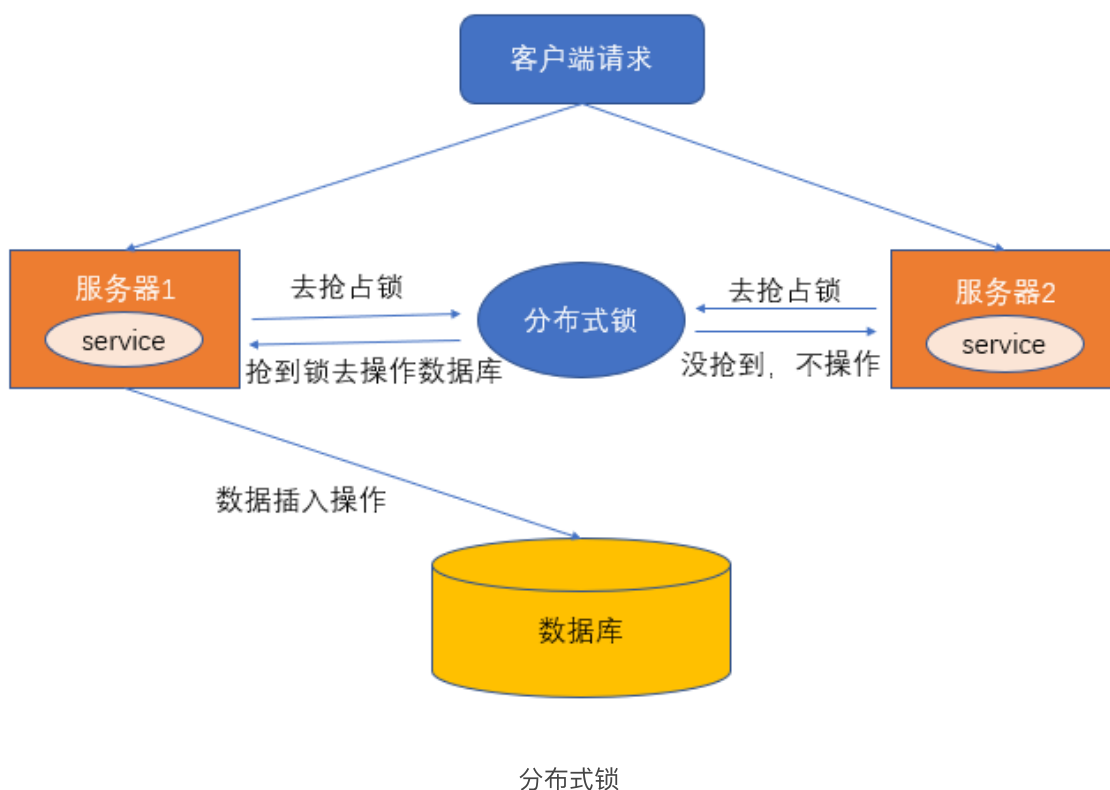
本篇文章主要介绍基于Redis的分布式锁实现到底是怎么回事，其中参考了许多大佬写的文章，算是对分布式锁做一个总结

分布式锁概览

在多线程的环境下，为了保证一个代码块在同一时间只能由一个线程访问，Java中我们一般可以使用synchronized语法和ReentrantLock去保证，这实际上是本地锁的方式。但是现在公司都是流行分布式架构，在分布式环境下，如何保证不同节点的线程同步执行呢？

实际上，对于分布式场景，我们可以使用分布式锁，它是控制分布式系统之间互斥访问共享资源的一种方式。

比如说在一个分布式系统中，多台机器上部署了多个服务，当客户端一个用户发起一个数据插入请求时，如果没有分布式锁机制保证，那么那多台机器上的多个服务可能进行并发插入操作，导致数据重复插入，对于某些不允许有多余数据的业务来说，这就会造成问题。而分布式锁机制就是为了解决这类问题，保证多个服务之间互斥的访问共享资源，如果一个服务抢占了分布式锁，其他服务没获取到锁，就不进行后续操作。大致意思如下图所示（不一定准确）：



分布式锁的特点

分布式锁一般有如下的特点：

- 互斥性：同一时刻只能有一个线程持有锁
- 可重入性：同一节点上的同一个线程如果获取了锁之后能够再次获取锁
- 锁超时：和J.U.C中的锁一样支持锁超时，防止死锁
- 高性能和高可用：加锁和解锁需要高效，同时也需要保证高可用，防止分布式锁失效
- 具备阻塞和非阻塞性：能够及时从阻塞状态中被唤醒

分布式锁的实现方式

我们一般实现分布式锁有以下几种方式：

- 基于数据库
- 基于Redis
- 基于zookeeper

本篇文章主要介绍基于Redis如何实现分布式锁

Redis的分布式锁实现

1. 利用setnx+expire命令 (错误的做法)

Redis的SETNX命令，setnx key value，将key设置为value，当键不存在时，才能成功，若键存在，什么也不做，成功返回1，失败返回0。SETNX实际上就是SET IF NOT Exists的缩写

因为分布式锁还需要超时机制，所以我们利用expire命令来设置，所以利用setnx+expire命令的核心代码如下：

```
1  public boolean tryLock(String key,String request,int timeout) {
2      Long result = jedis.setnx(key, request);
3      // result = 1时，设置成功，否则设置失败
4      if (result == 1L) {
5          return jedis.expire(key, timeout) == 1L;
6      } else {
7          return false;
8      }
9  }
```

实际上上面的步骤是有问题的，setnx和expire是分开的两步操作，不具有原子性，如果执行完第一条指令应用异常或者重启了，锁将无法过期。

一种改善方案就是使用Lua脚本来保证原子性（包含setnx和expire两条指令）

2. 使用Lua脚本（包含setnx和expire两条指令）

代码如下

```
1  public boolean tryLock_with_lua(String key, String UniqueId, int seconds) {
2      String lua_scripts = "if redis.call('setnx',KEYS[1],ARGV[1]) == 1
3                          'redis.call('expire',KEYS[1],ARGV[2]) return 1 else return 0";
4      List<String> keys = new ArrayList<>();
5      List<String> values = new ArrayList<>();
6      keys.add(key);
7      values.add(UniqueId);
8      values.add(String.valueOf(seconds));
```

```
9      Object result = jedis.eval(lua_scripts, keys, values);
10      //判断是否成功
11      return result.equals(1L);
12  }
```

3. 使用 set key value [EX seconds][PX milliseconds][NX|XX] 命令 (正确做法)

Redis在 2.6.12 版本开始, 为 SET 命令增加一系列选项:

```
1  SET key value[EX seconds][PX milliseconds][NX|XX]
```

- EX seconds: 设定过期时间, 单位为秒
- PX milliseconds: 设定过期时间, 单位为毫秒
- NX: 仅当key不存在时设置值
- XX: 仅当key存在时设置值

set命令的nx选项, 就等同于setnx命令, 代码过程如下:

```
1  public boolean tryLock_with_set(String key, String UniqueId, int seconds) {
2      return "OK".equals(jedis.set(key, UniqueId, "NX", "EX", seconds));
3  }
```

value必须要具有唯一性, 比如UUID, 至于为什么要保证唯一性? 我的理解是当我们释放锁时需要验证value值, 如果value不唯一, 有多个相同的值, 释放锁的过程就可能会出现问題。

关于释放锁的问题

释放锁时需要验证value值, 也就是说我们在获取锁的时候需要设置一个value, 不能直接用del key这种粗暴的方式, 因为直接del key任何客户端都可以进行解锁了, 所以解锁时, 我们需要判断锁是否是自己的, 基于value值来判断, 代码如下:

```
1  public boolean releaseLock_with_lua(String key,String value) {
2      String luaScript = "if redis.call('get',KEYS[1]) == ARGV[1] then "
3          "return redis.call('del',KEYS[1]) else return 0 end";
4      return jedis.eval(luaScript, Collections.singletonList(key), Collections.singletonList(value));
5  }
```

这里使用Lua脚本的方式，尽量保证原子性。

使用 `set key value [EX seconds][PX milliseconds][NX|XX]` 命令 看上去很OK，实际上在Redis集群的时候也会出现问题，比如说A客户端在Redis的master节点上拿到了锁，但是这个加锁的key还没有同步到slave节点，master故障，发生故障转移，一个slave节点升级为master节点，B客户端也可以获取同个key的锁，但客户端A也已经拿到锁了，这就导致多个客户端都拿到锁。

所以针对Redis集群这种情况，还有其他方案

4. Redlock算法 与 Redisson 实现

Redis作者 antirez基于分布式环境下提出了一种更高级的分布式锁的实现Redlock，原理如下：

“

下面参考文章[Redlock: Redis分布式锁最牛逼的实现](#) 和 <https://redis.io/topics/distlock>

假设有5个独立的Redis节点（注意这里的节点可以是5个Redis单master实例，也可以是5个Redis Cluster集群，但并不是有5个主节点的cluster集群）：

- 获取当前Unix时间，以毫秒为单位
- 依次尝试从5个实例，使用相同的key和具有唯一性的value(例如UUID)获取锁，当向Redis请求获取锁时，客户端应该设置一个网络连接和响应超时时间，这个超时时间应用小于锁的失效时间，例如你的锁自动失效时间为10s，则超时时间应该在5~50毫秒之间，这样可以避免服务器端Redis已经挂掉的情况下，客户端还在死死地等待响应结果。如果服务端没有在规定时间内响应，客户端应该尽快尝试去另外一个Redis实例请求获取锁
- 客户端使用当前时间减去开始获取锁时间（步骤1记录的时间）就得到获取锁使用的时间，当且仅当从大多数($N/2+1$ ，这里是3个节点)的Redis节点都取到锁，并且使用的时间小于锁失效时间时，锁才算获取成功。
- 如果取到了锁，key的真正有效时间等于有效时间减去获取锁所使用的时间（步骤3计算的结果）
- 如果某些原因，获取锁失败（没有在至少 $N/2+1$ 个Redis实例取到锁或者取锁时间已经超过了有效时间），客户端应该在所有的Redis实例上进行解锁（即便某些Redis实例根本就没有加锁成功，防止某些节点获取到锁但是客户端没有得到响应而导致接下来的一段时间不能被重新获取锁）

Redisson实现简单分布式锁

对于Java用户而言，我们经常使用Jedis，Jedis是Redis的Java客户端，除了Jedis之外，Redisson也是Java的客户端，Jedis是阻塞式I/O，而Redisson底层使用Netty可以实现非阻塞I/O，该客户端封装了锁的，继承了J.U.C的Lock接口，所以我们可以像使用ReentrantLock一样使用Redisson，具体使用过程如下。

1. 首先加入POM依赖

```
1 <dependency>
2     <groupId>org.redisson</groupId>
3     <artifactId>redisson</artifactId>
4     <version>3.10.6</version>
5 </dependency>
```

1. 使用Redisson, 代码如下(与使用ReentrantLock类似)

```
1 // 1. 配置文件
2 Config config = new Config();
3 config.useSingleServer()
4     .setAddress("redis://127.0.0.1:6379")
5     .setPassword(RedisConfig.PASSWORD)
6     .setDatabase(0);
7 //2. 构造RedissonClient
8 RedissonClient redissonClient = Redisson.create(config);
9
10 //3. 设置锁定资源名称
11 RLock lock = redissonClient.getLock("redlock");
12 lock.lock();
13 try {
14     System.out.println("获取锁成功, 实现业务逻辑");
15     Thread.sleep(10000);
16 } catch (InterruptedException e) {
17     e.printStackTrace();
18 } finally {
19     lock.unlock();
20 }
```

关于Redlock算法的实现, 在Redisson中我们可以使用RedissonRedLock来完成, 具体使用细节可以参考大佬的文章: https://mp.weixin.qq.com/s/8uhYult2h_YUHT7q7YCKYQ

Redis实现的分布式锁轮子

下面利用SpringBoot + Jedis + AOP的组合来实现一个简易的分布式锁。

1. 自定义注解

自定义一个注解, 被注解的方法会执行获取分布式锁的逻辑

```
1 @Target(ElementType.METHOD)
2 @Retention(RetentionPolicy.RUNTIME)
```

```

3  @Documented
4  @Inherited
5  public @interface RedisLock {
6      /**
7       * 业务键
8       *
9       * @return
10      */
11      String key();
12      /**
13       * 锁的过期秒数,默认是5秒
14       *
15       * @return
16       */
17      int expire() default 5;
18
19      /**
20       * 尝试加锁, 最多等待时间
21       *
22       * @return
23       */
24      long waitTime() default Long.MIN_VALUE;
25      /**
26       * 锁的超时时间单位
27       *
28       * @return
29       */
30      TimeUnit timeUnit() default TimeUnit.SECONDS;
31  }

```

2. AOP拦截器实现

在AOP中我们去执行获取分布式锁和释放分布式锁的逻辑, 代码如下:

```

1  @Aspect
2  @Component
3  public class LockMethodAspect {
4      @Autowired
5      private RedisLockHelper redisLockHelper;
6      @Autowired
7      private JedisUtil jedisUtil;
8      private Logger logger = LoggerFactory.getLogger(LockMethodAspect.class);
9
10     @Around("@annotation(com.redis.lock.annotation.RedisLock)")
11     public Object around(ProceedingJoinPoint joinPoint) {

```

```

12     Jedis jedis = jedisUtil.getJedis();
13     MethodSignature signature = (MethodSignature) joinPoint.getSignature();
14     Method method = signature.getMethod();
15
16     RedisLock redisLock = method.getAnnotation(RedisLock.class);
17     String value = UUID.randomUUID().toString();
18     String key = redisLock.key();
19     try {
20         final boolean islock = redisLockHelper.lock(jedis, key, value);
21         logger.info("isLock : {}", islock);
22         if (!islock) {
23             logger.error("获取锁失败");
24             throw new RuntimeException("获取锁失败");
25         }
26         try {
27             return joinPoint.proceed();
28         } catch (Throwable throwable) {
29             throw new RuntimeException("系统异常");
30         }
31     } finally {
32         logger.info("释放锁");
33         redisLockHelper.unlock(jedis, key, value);
34         jedis.close();
35     }
36 }
37 }

```

3. Redis实现分布式锁核心类

```

1  @Component
2  public class RedisLockHelper {
3      private long sleepTime = 100;
4      /**
5       * 直接使用setnx + expire方式获取分布式锁
6       * 非原子性
7       *
8       * @param key
9       * @param value
10      * @param timeout
11      * @return
12      */
13      public boolean lock_setnx(Jedis jedis, String key, String value,
14                               Long result = jedis.setnx(key, value);
15      // result = 1时, 设置成功, 否则设置失败
16      if (result == 1) {

```



```

16         ... (result == 1L) {
17             return jedis.expire(key, timeout) == 1L;
18         } else {
19             return false;
20         }
21     }
22
23     /**
24      * 使用Lua脚本，脚本中使用setnx+expire命令进行加锁操作
25      *
26      * @param jedis
27      * @param key
28      * @param UniqueId
29      * @param seconds
30      * @return
31      */
32     public boolean Lock_with_lua(Jedis jedis, String key, String UniqueId,
33         String lua_scripts = "if redis.call('setnx',KEYS[1],ARGV[1]) > 0 then
34             'redis.call('expire',KEYS[1],ARGV[2]) return 1 else return 0 end"
35         List<String> keys = new ArrayList<>();
36         List<String> values = new ArrayList<>();
37         keys.add(key);
38         values.add(UniqueId);
39         values.add(String.valueOf(seconds));
40         Object result = jedis.eval(lua_scripts, keys, values);
41         //判断是否成功
42         return result.equals(1L);
43     }
44
45     /**
46      * 在Redis的2.6.12及以后中,使用 set key value [NX] [EX] 命令
47      *
48      * @param key
49      * @param value
50      * @param timeout
51      * @return
52      */
53     public boolean lock(Jedis jedis, String key, String value, int timeout,
54         long seconds = TimeUnit.SECONDS.convert(timeout, TimeUnit.MILLISECONDS));
55         return "OK".equals(jedis.set(key, value, "NX", "EX", seconds));
56     }
57
58     /**
59      * 自定义获取锁的超时时间
60      *
61      * @param jedis
62      * @param key
63      * @param value

```

```

64      * @param timeout
65      * @param waitTime
66      * @param timeUnit
67      * @return
68      * @throws InterruptedException
69      */
70      public boolean lock_with_waitTime(Jedis jedis,String key, String
71          long seconds = timeUnit.toSeconds(timeout);
72          while (waitTime >= 0) {
73              String result = jedis.set(key, value, "nx", "ex", seconds);
74              if ("OK".equals(result)) {
75                  return true;
76              }
77              waitTime -= sleepTime;
78              Thread.sleep(sleepTime);
79          }
80          return false;
81      }
82      /**
83       * 错误的解锁方法-直接删除key
84       *
85       * @param key
86       */
87      public void unlock_with_del(Jedis jedis,String key) {
88          jedis.del(key);
89      }
90
91      /**
92       * 使用Lua脚本进行解锁操纵，解锁的时候验证value值
93       *
94       * @param jedis
95       * @param key
96       * @param value
97       * @return
98       */
99      public boolean unlock(Jedis jedis,String key,String value) {
100          String luaScript = "if redis.call('get',KEYS[1]) == ARGV[1] then
101              return redis.call('del',KEYS[1]) else return 0 end";
102          return jedis.eval(luaScript, Collections.singletonList(key),
103              value);
104      }

```

4. Controller层控制

定义一个TestController来测试我们实现的分布式锁

```
1  @RestController
2  public class TestController {
3      @RedisLock(key = "redis_lock")
4      @GetMapping("/index")
5      public String index() {
6          return "index";
7      }
8  }
```



小结

分布式锁重点在于互斥性，在任意一个时刻，只有一个客户端获取了锁。在实际的生产环境中，分布式锁的实现可能会更复杂，而我这里的讲述主要针对的是单机环境下的基于Redis的分布式锁实现，至于Redis集群环境并没有过多涉及，有兴趣的朋友可以查阅相关资料。

项目源码地址：<https://github.com/pjmike/redis-distributed-lock>

参考资料 & 鸣谢

- <https://mp.weixin.qq.com/s/eHsuEc8Dq3h1Kz1uqBWsjA>
- https://mp.weixin.qq.com/s/y2HPj2ji2KLS_eTR5nBnDA
- https://mp.weixin.qq.com/s/8uhYult2h_YUHT7q7YCKYQ
- https://mp.weixin.qq.com/s/xCe2ljuhMWD2rsstmNab_Q
- <https://crossoverjie.top/2018/03/29/distributed-lock/distributed-lock-redis/>
- <https://blog.battcn.com/2018/06/13/springboot/v2-cache-redislock/#%E5%85%B7%E4%BD%93%E4%BB%A3%E7%A0%81>
- <https://redis.io/topics/distlock>

 Komentari  Bagikan

LEBIH BARU

CSAPP笔记之存储器层次结构

LEBIH LAMA

浅析 Synchronized的底层实现及锁升级

0 Comments

pjMike

🔒

Disqus' Privacy Policy

1


Login

❤️ Recommend

🐦 Tweet

📌 Share

Sort by Best



Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

📧 Subscribe

🔗 Add Disqus to your siteAdd DisqusAdd

⚠️ Do Not Sell My Data

KATEGORI

- ▶ Java (13)
 - ▶ 并发 (1)
- ▶ Java, 并发 (1)
- ▶ MyBatis (1)
- ▶ MySQL (7)
- ▶ NIO (1)
- ▶ Netty (6)
 - ▶ RPC (1)
- ▶ Nginx (1)
- ▶ RabbitMQ (2)
- ▶ Redis (3)
- ▶ Spring (1)

- Spring Security (2)
- maven (1)
- spring (1)
- springboot (14)
- web (1)
- 代理 (1)
- 工具 (1)
- 技术杂谈 (1)
- 操作系统 (1)
- 数据结构 (2)
- 数据结构与算法 (3)
- 文件系统 (1)
- 日常翻译 (1)
 - Java (1)
- 笔记 (3)
- 随笔 (1)

ARSIP

- February 2020 (1)
- January 2020 (1)
- December 2019 (2)
- October 2019 (2)
- June 2019 (1)
- May 2019 (1)
- April 2019 (6)

- ▶ January 2019 (3)
- ▶ December 2018 (1)
- ▶ November 2018 (6)
- ▶ October 2018 (4)
- ▶ September 2018 (17)
- ▶ August 2018 (22)
- ▶ July 2018 (2)
- ▶ June 2018 (1)
- ▶ March 2018 (3)

TAG

- ▶ BeanUtils (1)
- ▶ CSAPP (2)
- ▶ CSAPP, 笔记 (1)
- ▶ DNS (1)
- ▶ HTTP (1)
- ▶ JVM (4)
- ▶ Java (16)
- ▶ MyBatis (1)
- ▶ MySQL (7)
- ▶ NIO (2)
- ▶ NIO.2 (1)
- ▶ Netty (7)
- ▶ Nginx (1)
- ▶ RPC (1)

- RabbitMQ (2)
- Reactor (1)
- Redis (2)
- Redis, 分布式 (1)
- Socket (1)
- Spring (1)
- Spring Security (2)
- TCP (1)
- leetcode (2)
- maven (1)
- mybatis (1)
- redis (1)
- spring (1)
- spring data jpa (1)
- springboot (16)
- websocket (1)
- 二叉搜索树 (1)
- 代理 (1)
- 平衡二叉树 (1)
- 排序算法 (1)
- 操作系统 (2)
- 文件I/O (1)
- 文件系统 (1)
- 日常翻译 (1)
- 笔记 (2)

- ▶ [虚拟内存](#) (1)
- ▶ [限流](#) (1)
- ▶ [随笔](#) (1)

AWAN TAG

[BeanUtils](#) [CSAPP](#) [CSAPP, 笔记](#) [DNS](#) [HTTP](#) [JVM](#) [Java](#) [MyBatis](#) [MySQL](#) [NIO](#) [NIO.2](#) [Netty](#) [Nginx](#) [RPC](#) [RabbitMQ](#) [Reactor](#) [Redis](#) [Redis, 分布式](#) [Socket](#) [Spring](#) [Spring Security](#) [TCP](#) [leetcode](#) [maven](#) [mybatis](#) [redis](#) [spring](#) [spring data jpa](#) [springboot](#) [websocket](#) [二叉搜索树](#) [代理](#) [平衡二叉树](#) [排序算法](#) [操作系统](#) [文件I/O](#) [文件系统](#) [日常翻译](#) [笔记](#) [虚拟内存](#) [限流](#) [随笔](#)

TAUTAN

- ▶ [Roger](#)

© 2020 pjMike

Powered by [Hexo](#). Theme by [PPOffice](#)

[本站总访问量次](#) [本站访客数人次](#)