

REDIS分布式锁

分布式锁一般有三种实现方式：

1. 数据库乐观锁；
2. 基于Redis的分布式锁；
3. 基于ZooKeeper的分布式锁

本篇博客将介绍第二种方式，基于Redis实现分布式锁。

虽然网上已经有各种介绍Redis分布式锁实现的博客，然而他们的实现却有着各种各样的问题，为了避免误人子弟，本篇博客将详细介绍如何正确地实现Redis分布式锁。

可靠性

首先，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**。在任意时刻，只有一个客户端能持有锁。
- **不会发生死锁**。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
- **具有容错性**。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
- **解铃还须系铃人**。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

代码实现

组件依赖

首先我们要通过Maven引入Jedis开源组件，在pom.xml文件加入下面的代码：

```
<dependency>
  <groupId>redis.client</groupId>
  <artifactId>jedis</artifactId>
  <version>2.9.0</version>
</dependency>
```

加锁代码

正确姿势

Talk is cheap, show me the code。先展示代码，再带大家慢慢解释为什么这样实现：

```
public class RedisTool {

    private static final String LOCK_SUCCESS = "OK";
    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";

    /**
     * 尝试获取分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @param expireTime 超期时间
     * @return 是否获取成功
     */
    public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {

        String result = jedis.set(lockKey, requestId, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, expireTime);

        if (LOCK_SUCCESS.equals(result)) {
            return true;
        }
        return false;
    }

}
```

可以看到，我们加锁就一行代码：

```
jedis.set(String key, String value, String nxxx, String expx, int time)
```

这个set()方法一共有五个形参：

第一个为key，我们使用key来当锁，因为key是唯一的。

第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？

原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。

requestId可以使用UUID.randomUUID().toString()方法生成。

第三个为nxxx，这个参数我们填的是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；

第四个为expx，这个参数我们传的是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。

第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：

1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。

2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件。

首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。

其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。

最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。

由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

错误示例1

比较常见的错误示例就是使用jedis.setnx()和jedis.expire()组合实现加锁

代码如下：

```
public static void wrongGetLock1(Jedis jedis, String lockKey, String requestId, int expireTime) {  
  
    Long result = jedis.setnx(lockKey, requestId);  
    if (result == 1) {  
        // 若在这里程序突然崩溃，则无法设置过期时间，将发生死锁  
        jedis.expire(lockKey, expireTime);  
    }  
  
}
```

setnx()方法作用就是SET IF NOT EXIST，expire()方法就是给锁加一个过期时间。

乍一看好像和前面的set()方法结果一样，然而由于这是两条Redis命令，不具有原子性，如果程序在执行完setnx()之后突然崩溃，导致锁没有设置过期时间。那么将会发生死锁。

网上之所以有人这样实现，是因为低版本的jedis并不支持多参数的set()方法。

错误示例2

这一种错误示例就比较难以发现问题，而且实现也比较复杂。实现思路：使用jedis.setnx()命令实现加锁，其中key是锁，value是锁的过期时间。

执行过程：

1. 通过setnx()方法尝试加锁，如果当前锁不存在，返回加锁成功。
2. 如果锁已经存在则获取锁的过期时间，和当前时间比较，如果锁已经过期，则设置新的过期时间，返回加锁成功。

代码如下：

```
public static boolean wrongGetLock2(Jedis jedis, String lockKey, int expireTime) {
```

```

long expires = System.currentTimeMillis() + expireTime;
String expiresStr = String.valueOf(expires);

// 如果当前锁不存在，返回加锁成功
if (jedis.setnx(lockKey, expiresStr) == 1) {
    return true;
}

// 如果锁存在，获取锁的过期时间
String currentValueStr = jedis.get(lockKey);
if (currentValueStr != null && Long.parseLong(currentValueStr) < System.currentTimeMillis()) {
    // 锁已过期，获取上一个锁的过期时间，并设置现在锁的过期时间
    // 这里是问题的根本点，当AB两个线程同时执行到这里，AB两个线程都会执行getset方法，
    // 假设A先执行，此时过期时间存储的是A的，B再执行会将B的存储时间覆盖掉A的，
    // 在下面的If条件中，虽然B无法拿到锁，但B成功的将自己的过期时间覆盖了A的过期时间。
    String oldValueStr = jedis.getSet(lockKey, expiresStr);
    if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {
        // 考虑多线程并发的情况，只有一个线程的设置值和当前值相同，它才有权利加锁
        return true;
    }
}

// 其他情况，一律返回加锁失败
return false;
}

```

那么这段代码问题在哪里？

- \1. 由于是客户端自己生成过期时间，所以需要强制要求分布式下每个客户端的时间必须同步。
- \2. 当锁过期的时候，如果多个客户端同时执行jedis.getSet()方法，那么虽然最终只有一个客户端可以加锁，但这个客户端的锁过期时间可能被其他客户端覆盖。[详见我的注释](#)
- \3. 锁不具备拥有者标识，即任何客户端都可以解锁。

解锁代码

正确姿势

还是先展示代码，再带大家慢慢解释为什么这样实现：

```

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 releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {

```

```

        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));

        if (RELEASE_SUCCESS.equals(result)) {
            return true;
        }
        return false;
    }
}

```

可以看到，我们解锁只需要两行代码就搞定了！

jedis的eval()方法，入参1：lua脚本，剩下的参数，就是对应lua脚本的参数。

第一行代码，我们写了一个简单的Lua脚本代码，上一次见到这个编程语言还是在《黑客与画家》里，没想到这次居然用上了。

第二行代码，我们将Lua代码传到jedis.eval()方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。

那么这段Lua代码的功能是什么呢？

其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。

那么为什么要使用Lua语言来实现呢？

因为要确保上述操作是原子性的。关于非原子性会带来什么问题，可以阅读【解锁代码-错误示例2】

那么为什么执行eval()方法可以确保原子性，源于Redis的特性，下面是官网对eval命令的部分解释：

简单来说，就是在eval命令执行Lua代码的时候，Lua代码将被当成一个命令去执行，并且直到eval命令执行完成，Redis才会执行其他命令。

错误示例1

最常见的解锁代码就是直接使用jedis.del()方法删除锁，这种不先判断锁的拥有者而直接解锁的方式，会导致任何客户端都可以随时进行解锁，即使这把锁不是它的。

```

public static void wrongReleaseLock1(Jedis jedis, String lockKey) {
    jedis.del(lockKey);
}

```

错误示例2

这种解锁代码乍一看也是没问题，甚至我之前也差点这样实现，与正确姿势差不多，唯一区别的是分成两条命令去执行

代码如下：

```
public static void wrongReleaseLock2(Jedis jedis, String lockKey, String requestId) {  
  
    // 判断加锁与解锁是不是同一个客户端  
    if (requestId.equals(jedis.get(lockKey))) {  
        // 若在此时，这把锁突然不是这个客户端的，则会误解锁  
        jedis.del(lockKey);  
    }  
  
}
```

如代码注释，问题在于如果调用jedis.del()方法的时候，这把锁已经不属于当前客户端的时候会解除他人加的锁。

那么是否真的有这种场景？

答案是肯定的，比如客户端A加锁，一段时间之后客户端A解锁，在执行jedis.del()之前，锁突然过期了，此时客户端B尝试加锁成功，然后客户端A再执行del()方法，则将客户端B的锁给解除了。

总结

本文主要介绍了如何使用Java代码正确实现Redis分布式锁，对于加锁和解锁也分别给出了两个比较经典的错误示例。

其实想要通过Redis实现分布式锁并不难，只要保证能满足可靠性里的四个条件。

互联网虽然给我们带来了方便，只要有问题就可以google，然而网上的答案一定是对的吗？其实不然，所以我们更应该时刻保持着质疑精神，多想多验证。

如果你的项目中Redis是多机部署的，那么可以尝试使用Redisson实现分布式锁，这是Redis官方提供的Java组件，链接在参考阅读章节已经给出。