链滴

关于 spring-integration-redis 是个不完善的 redis 分布式锁这件事

作者：zhengliwei

原文链接：https://ld246.com/article/1601537403320

来源网站：链滴

hello，大家好，欢迎来到银之庭。我是Z，一个普通的程序员。最近在工作中遇到了需要用分布式锁场景，我就直接拷贝了其他项目里前人写的一个redis分布式锁拿来用了，大体看了一眼也没啥毛病。过工作之余，我还是搜索了一下有没有开源的redis分布式锁实现，如果有的话就研究一下，看能不能入到工作项目中来。在搜到spring-integration-redis项目并仔细研究了它的源码后，我惊讶地发现个spring下属的开源项目，虽然代码质量挺高，但redis分布式锁的实现逻辑居然是有缺陷的，具体地就是**spring-integration-redis在极端场景下，会直接删除其他线程占用的redis锁**。下面我们就来细研究下spring-integration-redis的代码，看看这个问题是如何产生的。

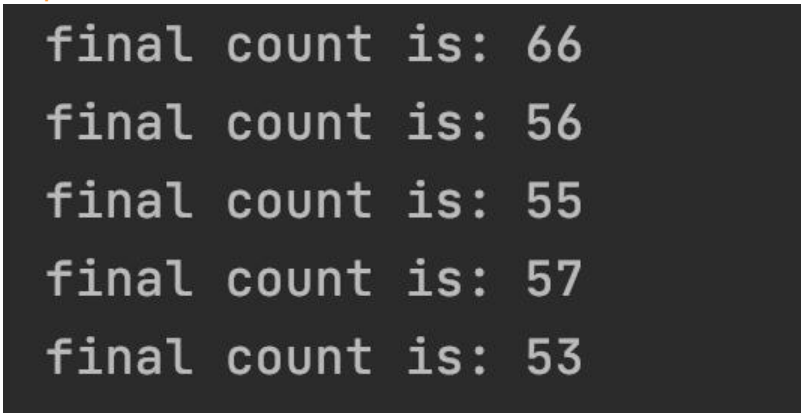
PS：以下实验用到了两个项目，可以从我的github上下载到，分别是demo项目和demo2项目。

1. 基本使用

我们先来试用一下spring-integration-redis这个组件。打开上面提到的两个项目，先运行起demo项，并查看demo项目的com.example.demo.controller.TestNoLock文件，代码如下：

```
package com.example.demo.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.CountDownLatch;

@RestController
public class TestNoLock {
    private int count;

    @RequestMapping("/testNoLock/")
    public String test() throws InterruptedException {

        CountDownLatch countDownLatch = new CountDownLatch(1000);
        count = 0;
        for (int i = 0; i < 1000; i++) {
            new Thread(() -> {
                try {
                    int tempCount = count;
                    tempCount += 1;
                    try {
                        Thread.sleep(1);
                    } catch (InterruptedException e) {
                    }
                    count = tempCount;
                }finally {
                    countDownLatch.countDown();
                }
            }).start();
        }

        countDownLatch.await();
        System.out.println("final count is: " + count);

        return "ok";
    }
}
```

这部分代码模拟的是在不加锁的情况下1000个线程并发修改一个共享变量，为了让效果更明显，并发改的代码拆分了count++的内部逻辑，并休眠了1ms（不知道是不是因为我电脑性能太高，如果不休的话，这段没加锁的代码运行结果大概率也是count=1000，没有产生并发修改的问题）。效果也很显，多次访问http://127.0.0.1:9982/testNoLock/URL，可以在终端看到最终的count的值，我的运结果如下图：



可以看到产生了明显的并发修改问题。下面我们来使用spring-integration-redis提供的分布式锁避并发修改问题。

使用spring-integration-redis比较简单，先引入依赖包：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>2.3.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-integration</artifactId>
    <version>2.3.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-redis</artifactId>
    <version>5.3.2.RELEASE</version>
</dependency>
```

然后新建一个配置类往spring容器里添加一个org.springframework.integration.redis.util.RedisLocRegistry的对象，代码如下：

```
package com.example.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.integration.redis.util.RedisLockRegistry;

@Configuration
public class RedisLock {

    @Bean
```

```
    public RedisLockRegistry redisLockRegistry(RedisConnectionFactory redisConnectionFactor
) {
        return new RedisLockRegistry(redisConnectionFactory, "registry_key");
    }
}
```

最后在需要的地方使用RedisLockRegistry新建锁对象即可，示例在demo项目的com.example.demo.controller.TestLock文件中，代码如下：

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.integration.redis.util.RedisLockRegistry;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.locks.Lock;

@RestController
public class TestLock {

    private int count;

    @Autowired
    private RedisLockRegistry redisLockRegistry; // 自动注入我们创建的redisLockRegistry对象

    @GetMapping("/testLock/")
    public String testLock() throws InterruptedException {

        // 创建锁对象
        Lock lock = redisLockRegistry.obtain("lock_key");

        CountDownLatch countDownLatch = new CountDownLatch(1000);
        count = 0;
        for (int i = 0; i < 1000; i++) {
            new Thread(() -> {
                // 加锁
                lock.lock();
                try {
                    int tempCount = count;
                    tempCount += 1;
                    try {
                        Thread.sleep(1);
                    } catch (InterruptedException e) {
                    }
                    count = tempCount;
                } finally {
                    countDownLatch.countDown();
                    // finally中确保释放锁
                    lock.unlock();
                }
            }).start();
```

```
        }

        countDownLatch.await();
        System.out.println("final count is: " + count);

        return "ok";
    }

}
```

这个接口是起1000个线程并发尝试去获取锁，然后修改共享变量count。这个接口依赖在本地的637端口启动的redis服务，启动redis比较简单，这里不再赘述。

访问http://127.0.0.1:9982/testLock/，可以在终端看到count的值都是1000，确实防止了并发修改问题，如下图：

```
final count is: 1000
final count is: 1000
final count is: 1000
final count is: 1000
final count is: 1000
```

到目前为止这个组件的功能看起来好像是没问题的，但实际上，我们现在只测试了一台实例（一个进）的情况，这种情况实际上使用的是我称之为“进程锁”的特性，即保证一个进程内不会有多个线程时进入某个临界区，而没有使用到“分布式锁”的特性。网上很多介绍spring-integration-redis的章中的例子都只到了这一步，而没有验证到“分布式锁”的特性。可能是混淆了“进程锁“和“分布锁“的概念，也可能是因为模拟多实例比较麻烦，所以用验证“进程锁”的特性糊弄过去了。而sprin-integration-redis实现的“分布式锁”特性是有缺陷的。我们先来研究下它的代码，再来看看缺陷哪里。

2. 源码分析

点击demo项目代码中的redisLockRegistry.obtain或lock.lock或lock.unlock都可以进入spring-integation-redis的主文件org.springframework.integration.redis.util.RedisLockRegistry。我们主要分析ock.lock(), lock.tryLock(), lock.tryLock(long time, TimeUnit unit), lock.unlock()方法。这几个方也是实现分布式锁语义的关键方法。

2.1 lock()

首先看lock.lock()。代码截取如下：

```
@Override
public void lock() {
    this.localLock.lock();
    while (true) {
```

```
        try {
            while (!obtainLock()) {
                Thread.sleep(100); //NOSONAR
            }
            break;
        }
        catch (InterruptedException e) {
            /***/
        }
        catch (Exception e) {
            this.localLock.unlock();
            rethrowAsLockException(e);
        }
    }
}

private boolean obtainLock() {
    Boolean success = RedisLockRegistry.this.redisTemplate.execute(
                    RedisLockRegistry.this.obtainLockScript,
            Collections.singletonList(this.lockKey),
            RedisLockRegistry.this.clientId,
            String.valueOf(RedisLockRegistry.this.expireAfter));

    boolean result = Boolean.TRUE.equals(success);

    if (result) {
        this.lockedAt = System.currentTimeMillis();
    }
    return result;
}
```

方法中会先尝试调用localLock.lock()方法，localLock是个本地的重入锁ReentrantLock，这个本地有两个作用，一是直接在本地拦截其他线程的请求，避免每次都请求redis，提高锁性能，二是借助重锁实现了分布式锁的可重入特性的一部分（另一部分是在发给redis执行的脚本中实现的，如果是重入取锁，则刷新过期时间），这个localLock的使用还是比较巧妙的，值得学习。接下来是个死循环，断调用obtainLock()方法尝试获取锁，如果失败就休眠100ms重试，并忽略了中断异常，遇到其他异时会释放本地锁，并重新抛出异常。下面主要看下obtainLock()方法，这个方法就是直接向redis发请，执行一个lua脚本，防止执行多条redis命令中间被其他实例的线程打断，导致锁处于一个不一致的态。脚本的意思比较简单，先查一下锁的key，如果存在且value是本线程的，就刷新过期时间，证明次是个重入获取锁的过程，如果锁的key不存在，则新建锁的key，写入本线程的value，其他情况直返回失败。

2.2 tryLock()

下面看一下tryLock()，它直接调用了tryLock(long time, TimeUnit unit)这个重载方法。我们直接看面的重载方法。tryLock实现的是可设置超时时间的尝试获取锁的语义。代码如下：

```
@Override
public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
    long now = System.currentTimeMillis();
    if (!this.localLock.tryLock(time, unit)) {
        return false;
    }
```

```
    try {
        long expire = now + TimeUnit.MILLISECONDS.convert(time, unit);
        boolean acquired;
        while (!(acquired = obtainLock()) && System.currentTimeMillis() < expire) { //NOSONAR
            Thread.sleep(100); //NOSONAR
        }
        if (!acquired) {
            this.localLock.unlock();
        }
        return acquired;
    }
    catch (Exception e) {
        this.localLock.unlock();
        rethrowAsLockException(e);
    }
    return false;
}
```

也是先尝试获取本地的重入锁，如果本地锁获取失败直接返回失败。获取成功后再调用obtainLock()法获取redis锁，并循环检查是否到超时时间，如果最终没有获取到redis锁，需要释放本地锁，如果取到了redis锁，则一直持有本地锁，直到unlock()方法中释放本地和redis的两个锁。这里有个细节，传入的时间其实用在两个地方，一是重试获取本地锁的超时时间，二是尝试获取redis锁的超时时间所以调用tryLock(time, unit)方法，实际可能等待2*time的时间。一个参数代表两个含义，我也不清这个设计是好是坏。

到目前为止都还挺正常的，本地锁的使用也值得我们学习，但后面unlock()的实现就有些糙了，我们起来看一下。

2.3 unlock()

unlock()的代码如下：

```
@Override
public void unlock() {
    if (!this.localLock.isHeldByCurrentThread()) {
        throw new IllegalStateException("You do not own lock at " + this.lockKey);
    }
    if (this.localLock.getHoldCount() > 1) {
        this.localLock.unlock();
        return;
    }
    try {
        if (!isAcquiredInThisProcess()) {
            throw new IllegalStateException("Lock was released in the store due to expiration. " +
                    "The integrity of data protected by this lock may have been compromised.");
        }

        if (Thread.currentThread().isInterrupted()) {
                RedisLockRegistry.this.executor.execute(this::removeLockKey);
        }
        else {
            removeLockKey();
        }
```

```
            if (LOGGER.isDebugEnabled()) {
                LOGGER.debug("Released lock; " + this);
            }
        }
        catch (Exception e) {
            ReflectionUtils.rethrowRuntimeException(e);
        }
        finally {
            this.localLock.unlock();
        }
    }
```

unlock()方法全程实现的语义类似：如果本地锁不是本线程持有的，直接报错，后续就以本地锁是本程持有为前提。如果本地锁重入次数大于1，则本次释放只需要把重入次数减一即可，然后直接返回如果重入次数已经等于1了，就得同时释放redis锁和本地锁了。释放redis锁时先检查redis里锁的valu是不是本线程的，如果不是就报错，这时候可能是因为本线程写入redis的锁过期释放了，然后被其他例的线程重新写入锁了。如果是，再发一条命令删除锁的key，命令可以是unlink或del，这不重要。这可能有同学已经发现问题了，我们准备redis分布式锁的面试题的时候，都会注意说“释放锁要用lu脚本，把检查锁是不是本线程持有和删除锁放到一个lua脚本中，防止高并发时误删其他线程写入的”。但这里判断锁是不是本线程持有和删除key却分成了两条redis命令，那么在极端情况下，本线程断锁是本线程持有的，但在删除命令执行前，key过期了，有其他实例的线程写入了锁，然后本线程直接删除了锁的key，就会误删其他实例线程写入的锁了（其他实例的线程真是躺着中枪了）。这种况确实比较极端，不容易复现，下面我们借助debug来拉长unlock()方法每一步的执行时间，并通过两个服务来模拟多个实例的情况（实际上是多个进程，进程在同一实例还是不同实例其实没区别）。

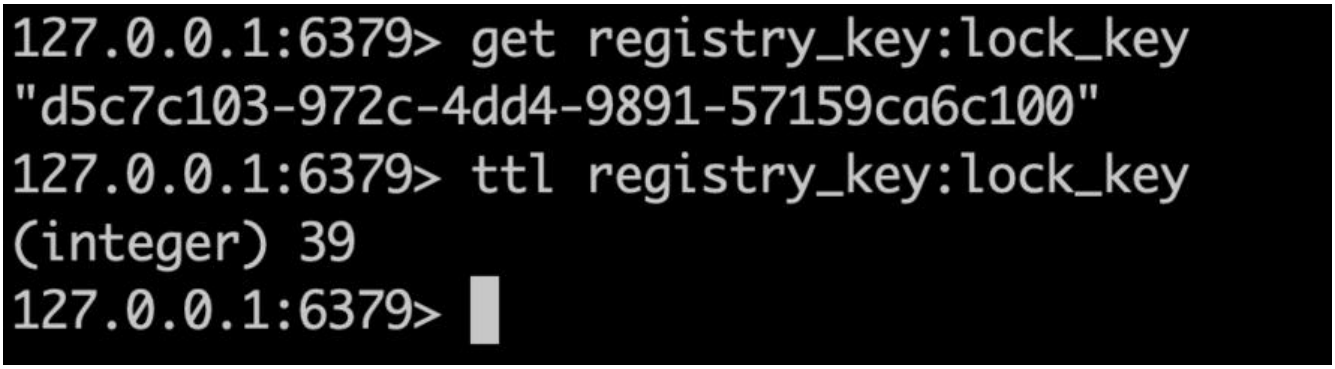
3. unlock方法缺陷验证

这时候需要用debug模式启动demo和demo2两个项目，demo2项目的代码和demo几乎一样，只配置不同。我们用两个项目的controller.TestUnlock中提供的接口来验证unlock()方法在极端情况下缺陷。代码如下：

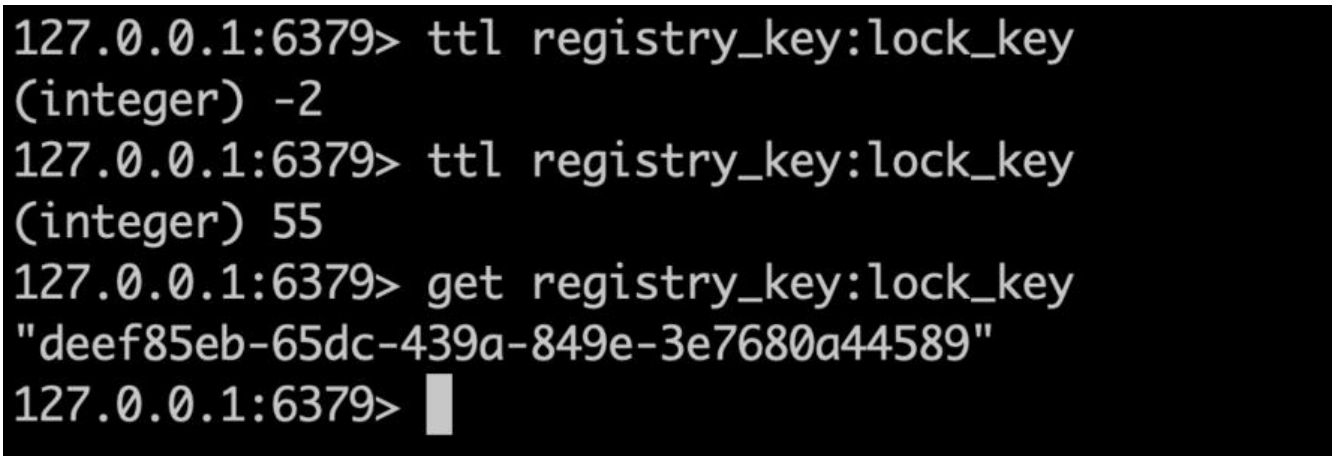
```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.integration.redis.util.RedisLockRegistry;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;

@RestController
public class TestUnlock {

    @Autowired
    private RedisLockRegistry redisLockRegistry; // 自动注入我们创建的redisLockRegistry对象

    @GetMapping("/testUnlock/")
    public String testUnlock() throws InterruptedException {
        // 创建锁对象
        Lock lock = redisLockRegistry.obtain("lock_key");
```

```
        lock.tryLock(60, TimeUnit.SECONDS);
        try {
            Thread.sleep(10);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }

        return "ok";
    }
}
```

我们把断点打到demo2的lock.unlock()方法内的removeLockKey()方法那一行，然后调用demo2的口http://127.0.0.1:9983/testUnlock/，如果执行到断点的地方，证明redis锁已经写入成功了，并且经在尝试释放锁了，我们可以去redis里验证一下，效果如下图：



现在我们什么都不做，等待redis里这个key过期，然后把断点打到demo项目的lock.unlock()方法那行，调用demo的接口http://127.0.0.1:9982/testUnlock/，尝试获取锁，这时候会获取成功。redis的锁的value已经变成demo线程的clientId了，如下图：



先不要往下执行demo项目，回到demo2，点击继续，会发现demo2直接把demo写入的锁删除掉了而demo继续执行下去，在解锁时会报Lock was released in the store due to expiration. The integrity of data protected by this lock may have been compromised.的错，但实际上，它写入的锁并有超时，而是被demo2给删除了。

以上，我们试用了spring-integration-redis提供的分布式锁功能，并阅读了它的代码，发现了它在现分布式锁时有“可能删除其他进程写入的锁”的问题。所以，redis分布式锁的实现我并不建议大家这个组件，我更推荐使用redission这个组件。这个组件也是redis官方推荐的实现。后续有机会我再

大家介绍一下redission的使用。就这样～

原文链接：关于 spring-integration-redis 是个不完善的 redis 分布式锁这件事