链滴

老大吩咐的可重入分布式锁，终于完美的实现了 ~

作者：9526xu

原文链接：https://ld246.com/article/1592125595301

来源网站：链滴

重做永远比改造简单

最近在做一个项目，将一个其他公司的实现系统（下文称作旧系统），完整的整合到自己公司的系统（文称作新系统）中，这其中需要将对方实现的功能完整在自己系统也实现一遍。

旧系统还有一批存量商户，为了不影响存量商户的体验，新系统提供的对外接口，还必须得跟以前一。最后系统完整切换之后，功能只运行在新系统中，这就要求旧系统的数据还需要完整的迁移到新系中。

当然这些在做这个项目之前就有预期，想过这个过程很难，但是没想到有那么难。原本感觉排期大半，时间还是挺宽裕，现在感觉就是大坑，还不得不在坑里一点点去填。

哎，说多都是泪，不吐槽了，等到下次做完再给大家复盘下真正心得体会。

回到正文，上篇文章Redis 分布式锁，咱们基于 Redis 实现一个分布式锁。这个分布式锁基本功能没么问题，但是缺少可重入的特性，所以这篇文章小黑哥就带大家来实现一下可重入的分布式锁。

本篇文章将会涉及以下内容：

- 可重入
- 基于 ThreadLocal 实现方案
- 基于 Redis Hash 实现方案

可重入

说到可重入锁，首先我们来看看一段来自 wiki 上可重入的解释：

若一个程序或子程序可以“在任意时刻被中断然后操作系统调度执行另外一段代码，这段代码又调用该子程序不会出错”，则称其为**可重入**（reentrant或re-entrant）的。即当该子程序正在运行时，执线程可以再次进入并执行它，仍然获得符合设计时预期的结果。与多线程并发执行的线程安全不同，重入强调对单个线程执行时重新进入同一个子程序仍然是安全的。

当一个线程执行一段代码成功获取锁之后，继续执行时，又遇到加锁的代码，可重入性就就保证线程继续执行，而不可重入就是需要等待锁释放之后，再次获取锁成功，才能继续往下执行。

用一段 Java 代码解释可重入：

```
public synchronized void a() {
    b();
}

public synchronized void b() {
    // pass
}
```

假设 X 线程在 a 方法获取锁之后，继续执行 b 方法，如果此时**不可重入**，线程就必须等待锁释放，次争抢锁。

锁明明是被 X 线程拥有，却还需要等待自己释放锁，然后再去抢锁，这看起来就很奇怪，我释放我自~

可重入性就可以解决这个尴尬的问题，当线程拥有锁之后，往后再遇到加锁方法，直接将加锁次数加，然后再执行方法逻辑。退出加锁方法之后，加锁次数再减 1，当加锁次数为 0 时，锁才被真正的释。

可以看到可重入锁最大特性就是计数，计算加锁的次数。所以当可重入锁需要在分布式环境实现时，们也就需要统计加锁次数。

分布式可重入锁实现方式有两种：

- 基于 ThreadLocal 实现方案
- 基于 Redis Hash 实现方案

首先我们看下基于 ThreadLocal 实现方案。

基于 ThreadLocal 实现方案

实现方式

Java 中 ThreadLocal可以使每个线程拥有自己的实例副本，我们可以利用这个特性对线程重入次数行技术。

下面我们定义一个 ThreadLocal的全局变量 LOCKS，内存存储 Map 实例变量。

```
private static ThreadLocal<Map<String, Integer>> LOCKS = ThreadLocal.withInitial(HashMap:
new);
```

每个线程都可以通过 ThreadLocal获取自己的 Map实例，Map 中 key 存储锁的名称，而 value存储的重入次数。

加锁的代码如下：

```
/**
 * 可重入锁
 *
 * @param lockName  锁名字,代表需要争临界资源
 * @param request   唯一标识，可以使用 uuid，根据该值判断是否可以重入
 * @param leaseTime 锁释放时间
 * @param unit      锁释放时间单位
 * @return
 */
public Boolean tryLock(String lockName, String request, long leaseTime, TimeUnit unit) {
    Map<String, Integer> counts = LOCKS.get();
    if (counts.containsKey(lockName)) {
        counts.put(lockName, counts.get(lockName) + 1);
        return true;
    } else {
        if (redisLock.tryLock(lockName, request, leaseTime, unit)) {
            counts.put(lockName, 1);
            return true;
        }
    }
    return false;
}
```

ps: redisLock#tryLock 为上一篇文章实现的分布锁。

由于公号外链无法直接跳转，关注『**程序通事**』，回复**分布式锁**获取源代码。

加锁方法首先判断当前线程是否已经已经拥有该锁，若已经拥有，直接对锁的重入次数加 1。

若还没拥有该锁，则尝试去 **Redis** 加锁，加锁成功之后，再对重入次数加 1 。

释放锁的代码如下：

```
/**
 * 解锁需要判断不同线程池
 *
 * @param lockName
 * @param request
 */
public void unlock(String lockName, String request) {
    Map<String, Integer> counts = LOCKS.get();
    if (counts.getOrDefault(lockName, 0) <= 1) {
        counts.remove(lockName);
        Boolean result = redisLock.unlock(lockName, request);
        if (!result) {
            throw new IllegalMonitorStateException("attempt to unlock lock, not locked by lockN
me:+" + lockName + " with request: "
                    + request);
        }
```

```
    } else {
        counts.put(lockName, counts.get(lockName) - 1);
    }
}
```

释放锁的时首先判断重入次数，若大于 1，则代表该锁是被该线程拥有，所以直接将锁重入次数减 1 可。

若当前可重入次数小于等于 1，首先移除 Map中锁对应的 key，然后再到 Redis 释放锁。

这里需要注意的是，当锁未被该线程拥有，直接解锁，可重入次数也是小于等于 1 ，这次可能无法直解锁成功。

ThreadLocal 使用过程要记得及时清理内部存储实例变量，防止发生内存泄漏，上下文数据串用等问。

下次咱来聊聊最近使用 ThreadLocal 写的 Bug。

相关问题

使用 ThreadLocal 这种本地记录重入次数，虽然真的简单高效，但是也存在一些问题。

过期时间问题

上述加锁的代码可以看到，重入加锁时，仅仅对本地计数加 1 而已。这样可能就会导致一种情况，由业务执行过长，Redis 已经过期释放锁。

而再次重入加锁时，由于本地还存在数据，认为锁还在被持有，这就不符合实际情况。

如果要在本地增加过期时间，还需要考虑本地与 Redis 过期时间一致性的，代码就会变得很复杂。

不同线程/进程可重入问题

狭义上可重入性应该只是对于**同一线程**的可重入，但是实际业务可能需要不同的应用线程之间可以重同把锁。

而 ThreadLocal的方案仅仅只能满足同一线程重入，无法解决不同线程/进程之间重入问题。

不同线程/进程重入问题就需要使用下述方案 Redis Hash 方案解决。

基于 Redis Hash 可重入锁

实现方式

ThreadLocal 的方案中我们使用了 Map 记载锁的可重入次数，而 Redis 也同样提供了 Hash （哈希）这种可以存储键值对数据结构。所以我们可以使用 Redis Hash 存储的锁的重入次数，然后利用 lua 脚本判断逻辑。

加锁的 lua 脚本如下：

```
---- 1 代表 true
---- 0 代表 false
```

```
if (redis.call('exists', KEYS[1]) == 0) then
    redis.call('hincrby', KEYS[1], ARGV[2], 1);
    redis.call('pexpire', KEYS[1], ARGV[1]);
    return 1;
end ;
if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then
    redis.call('hincrby', KEYS[1], ARGV[2], 1);
    redis.call('pexpire', KEYS[1], ARGV[1]);
    return 1;
end ;
return 0;
```

如果 KEYS:[lock],ARGV[1000,uuid]

不熟悉 lua 语言同学也不要怕，上述逻辑还是比较简单的。

加锁代码首先使用 Redis exists 命令判断当前 lock 这个锁是否存在。

如果锁不存在的话，直接使用 hincrby创建一个键为 lock hash 表，并且为 Hash 表中键为 uuid 初化为 0，然后再次加 1，最后再设置过期时间。

如果当前锁存在，则使用 hexists判断当前 lock 对应的 hash 表中是否存在 uuid 这个键，如果存在,次使用 hincrby 加 1，最后再次设置过期时间。

最后如果上述两个逻辑都不符合，直接返回。

加锁代码如下:

```
// 初始化代码

String lockLuaScript = IOUtils.toString(ResourceUtils.getURL("classpath:lock.lua").openStream(
, Charsets.UTF_8);
lockScript = new DefaultRedisScript<>(lockLuaScript, Boolean.class);

/**
 * 可重入锁
 *
 * @param lockName  锁名字,代表需要争临界资源
 * @param request   唯一标识，可以使用 uuid，根据该值判断是否可以重入
 * @param leaseTime 锁释放时间
 * @param unit      锁释放时间单位
 * @return
 */
public Boolean tryLock(String lockName, String request, long leaseTime, TimeUnit unit) {
    long internalLockLeaseTime = unit.toMillis(leaseTime);
    return stringRedisTemplate.execute(lockScript, Lists.newArrayList(lockName), String.valueOf
internalLockLeaseTime), request);
}
```

Spring-Boot 2.2.7.RELEASE

只要搞懂 Lua 脚本加锁逻辑，Java 代码实现还是挺简单的，直接使用 SpringBoot 提供的 StringRedi Template 即可。

解锁的 Lua 脚本如下：

```
-- 判断 hash set 可重入 key 的值是否等于 0
-- 如果为 0 代表 该可重入 key 不存在
if (redis.call('hexists', KEYS[1], ARGV[1]) == 0) then
    return nil;
end ;
-- 计算当前可重入次数
local counter = redis.call('hincrby', KEYS[1], ARGV[1], -1);
-- 小于等于 0 代表可以解锁
if (counter > 0) then
    return 0;
else
    redis.call('del', KEYS[1]);
    return 1;
end ;
return nil;
```

首先使用 hexists 判断 Redis Hash 表是否存给定的域。

如果 lock 对应 Hash 表不存在，或者 Hash 表不存在 uuid 这个 key，直接返回 nil。

若存在的情况下，代表当前锁被其持有，首先使用 hincrby使可重入次数减 1 ，然后判断计算之后可入次数，若小于等于 0，则使用 del 删除这把锁。

解锁的 Java 代码如下：

```
// 初始化代码：


String unlockLuaScript = IOUtils.toString(ResourceUtils.getURL("classpath:unlock.lua").openSteam(), Charsets.UTF_8);
unlockScript = new DefaultRedisScript<>(unlockLuaScript, Long.class);

/**
 * 解锁
 * 若可重入 key 次数大于 1，将可重入 key 次数减 1 <br>
 * 解锁 lua 脚本返回含义：<br>
 * 1:代表解锁成功 <br>
 * 0:代表锁未释放，可重入次数减 1 <br>
 * nil：代表其他线程尝试解锁 <br>
 * <p>
 * 如果使用 DefaultRedisScript<Boolean>，由于 Spring-data-redis eval 类型转化，<br>
 * 当 Redis 返回  Nil bulk, 默认将会转化为 false，将会影响解锁语义，所以下述使用：<br>
 * DefaultRedisScript<Long>
 * <p>
 * 具体转化代码请查看：<br>
 * JedisScriptReturnConverter<br>
 *
 * @param lockName 锁名称
 * @param request  唯一标识，可以使用 uuid
 * @throws IllegalMonitorStateException 解锁之前，请先加锁。若为加锁，解锁将会抛出该错误
 */
public void unlock(String lockName, String request) {
    Long result = stringRedisTemplate.execute(unlockScript, Lists.newArrayList(lockName), request);
    // 如果未返回值，代表其他线程尝试解锁
```

```
    if (result == null) {
        throw new IllegalMonitorStateException("attempt to unlock lock, not locked by lockNam
:+" + lockName + " with request: "
                + request);
    }
}
```

解锁代码执行方式与加锁类似，只不过解锁的执行结果返回类型使用 Long。这里之所以没有跟加锁样使用 Boolean ,这是因为解锁 lua 脚本中，三个返回值含义如下：

- 1 代表解锁成功，锁被释放
- 0 代表可重入次数被减 1
- null 代表其他线程尝试解锁，解锁失败

如果返回值使用 Boolean，**Spring-data-redis** 进行类型转换时将会把 null 转为 false，这就会影响们逻辑判断，所以返回类型只好使用 Long。

以下代码来自 JedisScriptReturnConverter：

```java
    public Object convert(Object result) {
        if (result instanceof String) {
            // evalsha converts byte[] to String. Convert back for consistency
            return SafeEncoder.encode((String) result);
        }
        if (returnType == ReturnType.STATUS) {
            return JedisConverters.toString((byte[]) result);
        }
        if (returnType == ReturnType.BOOLEAN) {
            // Lua false comes back as a null bulk reply
            if (result == null) {
                return Boolean.FALSE;
            }
            return ((Long) result == 1);
        }
        if (returnType == ReturnType.MULTI) {
            List<Object> resultList = (List<Object>) result;
            List<Object> convertedResults = new ArrayList<>();
            for (Object res : resultList) {
                if (res instanceof String) {
                    // evalsha converts byte[] to String. Convert back for
                    // consistency
                    convertedResults.add(SafeEncoder.encode((String) res));
                } else {
                    convertedResults.add(res);
                }
            }
            return convertedResults;
        }
        return result;
    }
}
```

相关问题

spring-data-redis 低版本问题

如果 Spring-Boot 使用 Jedis 作为连接客户端,并且使用Redis Cluster 集群模式，需要使用 **2.1.9** 上版本的**spring-boot-starter-data-redis**,不然执行过程中将会抛出：

org.springframework.dao.InvalidDataAccessApiUsageException: EvalSha is not supported in cl ster environment.

如果当前应用无法升级 spring-data-redis也没关系，可以使用如下方式，直接使用原生 Jedis 连接行 lua 脚本。

以加锁代码为例：

```
public boolean tryLock(String lockName, String reentrantKey, long leaseTime, TimeUnit unit) {
    long internalLockLeaseTime = unit.toMillis(leaseTime);
    Boolean result = stringRedisTemplate.execute((RedisCallback<Boolean>) connection -> {
        Object innerResult = eval(connection.getNativeConnection(), lockScript, Lists.newArrayLis
(lockName), Lists.newArrayList(String.valueOf(internalLockLeaseTime), reentrantKey));
        return convert(innerResult);
    });
    return result;
}

private Object eval(Object nativeConnection, RedisScript redisScript, final List<String> keys, fi
al List<String> args) {

    Object innerResult = null;
    // 集群模式和单点模式虽然执行脚本的方法一样，但是没有共同的接口，所以只能分开执行
    // 集群
    if (nativeConnection instanceof JedisCluster) {
        innerResult = evalByCluster((JedisCluster) nativeConnection, redisScript, keys, args);
    }
    // 单点
    else if (nativeConnection instanceof Jedis) {
        innerResult = evalBySingle((Jedis) nativeConnection, redisScript, keys, args);
    }
    return innerResult;
}
```

数据类型转化问题

如果使用 Jedis 原生连接执行 Lua 脚本，那么可能又会碰到数据类型的转换坑。

```
@Override
public Object eval(final String script, final List<String> keys, final List<String> args) {
  return eval(script, keys.size(), getParams(keys, args));
}
```

可以看到 Jedis#eval返回 Object，我们需要具体根据 Lua 脚本的返回值的，再进行相关转化。这其就涉及到 Lua 数据类型转化为 Redis 数据类型。

下面主要我们来讲下 Lua 数据转化 Redis 的规则中几条比较容易踩坑：

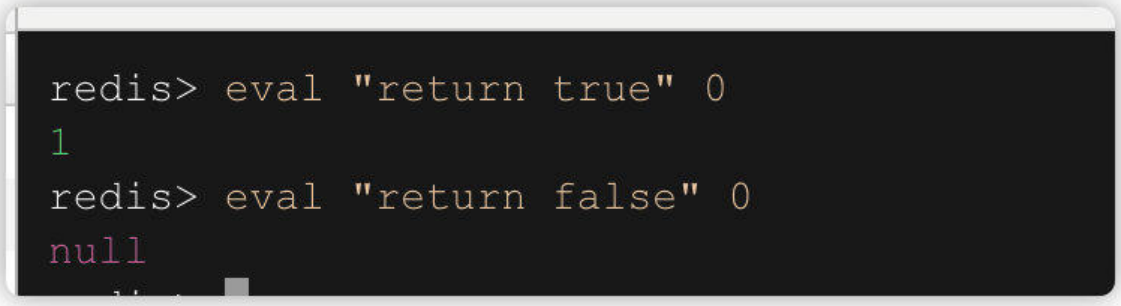
1. Lua number 与 Redis 数据类型转换

Lua 中 number 类型是一个双精度的浮点数，但是 Redis 只支持整数类型，所以这个转化过程将会弃小数位。

```
redis> eval "return 2.2" 0
2
redis> eval "return 2.5677" 0
2
```

2. Lua boolean 与 Redis 类型转换

这个转化比较容易踩坑，Redis 中是不存在 boolean 类型，所以当Lua 中 true 将会转为 Redis 整数。而 Lua 中 false 并不是转化整数，而是转化 **null** 返回给客户端。



3. Lua nil 与 Redis 类型转换

Lua nil 可以当做是一个空值，可以等同于 Java 中的 **null**。在 Lua 中如果 nil 出现在条件表达式，将当做 false 处理。

所以 Lua nil 也将会 **null** 返回给客户端。

其他转化规则比较简单，详情参考:

http://doc.redisfans.com/script/eval.html

总结

可重入分布式锁关键在于对于锁重入的计数，这篇文章主要给出两种解决方案，一种基于 ThreadLocal 实现方案，这种方案实现简单，运行也比较高效。但是若要处理锁过期的问题，代码实现就比较复杂。

另外一种采用 Redis Hash 数据结构实现方案，解决了 ThreadLocal 的缺陷，但是代码实现难度稍大需要熟悉 Lua 脚本，以及Redis 一些命令。另外使用 **spring-data-redis** 等操作 Redis 时不经意间会遇到各种问题。

帮助

https://www.sofastack.tech/blog/sofa-jraft-rheakv-distributedlock/

https://tech.meituan.com/2016/09/29/distributed-system-mutually-exclusive-idempotence-crberus-gtis.html

最后说两句（求关注）

最近大家应该发现微信公众号信息流改版了，再也不是按照时间顺序展示了。这就对我这样的迷你号，加倍打击，没有阅读量，正反馈持续减弱。

所以看完文章，哥哥姐姐们点个**在看**吧，周更真的超累，不知觉又写了两天，拒绝白嫖，来点正反馈~。

最后感谢各位的阅读，才疏学浅，难免存在纰漏，如果你发现错误的地方，可以留言指出。如果看完章还有其他不懂的地方，欢迎加我，互相学习，一起成长~

最后谢谢大家支持~

最最后，重要的事再说一篇~

快来关注我呀~
快来关注我呀~
快来关注我呀~

原文链接：老大吩咐的可重入分布式锁，终于完美的实现了 ~