**跨JVM的线程安全问题**

在单体的应用开发场景中，在多线程的环境下，涉及并发同步的时候，为了保证一个代码块在同一时间只能由一个线程访问，我们一般可以使用synchronized语法和ReetrantLock去保证，这实际上是本地锁的方式。

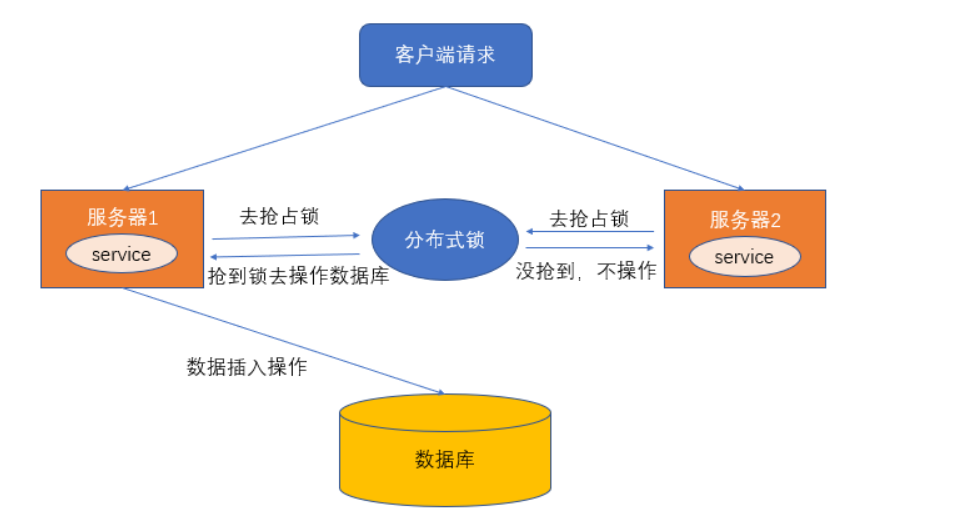
也就是说，在同一个JVM内部，大家往往采用synchronized或者Lock的方式来解决多线程间的安全问题。但在分布式集群工作的开发场景中，在JVM之间，那么就需要一种更加高级的锁机制，来处理种跨JVM进程之间的线程安全问题.

解决方案是：使用分布式锁

总之，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类似这类问题，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）：



### 何为分布式锁？

**何为分布式锁？**

* 当在分布式模型下，数据只有一份（或有限制），此时需要利用锁的技术控制某一时刻修改数据的进程数。
* 用一个状态值表示锁，对锁的占用和释放通过状态值来标识。

**分布式锁的条件：**

* 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。

**分布式锁的实现：**

分布式锁的实现由很多种，文件锁、数据库、redis等等，比较多；分布式锁常见的多种实现方式：

1. 数据库悲观锁、
2. 数据库乐观锁；
3. 基于Redis的分布式锁；
4. 基于ZooKeeper的分布式锁。

在实践中，还是redis做分布式锁性能会高一些

### 数据库悲观锁

所谓悲观锁，悲观锁是对数据被的修改持悲观态度（认为数据在被修改的时候一定会存在并发问题），因此在整个数据处理过程中将数据锁定。

悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在应用层中实现了加锁机制，也无法保证外部系统不会修改数据）。

数据库的行锁、表锁、排他锁等都是悲观锁，这里以行锁为例，进行介绍。以我们常用的MySQL为例，我们通过使用select...for update语句, 执行该语句后，会在表上加持行锁，一直到事务提交，解除行锁。

使用场景举例:

在秒杀案例中，生成订单和扣减库存的操作，可以通过商品记录的行锁，进行保护。们通过使用select...for update语句，在查询商品表库存时将该条记录加锁，待下单减库存完成后，再释放锁。

**示例的SQL如下：**

/0.开始事务

begin;

//1.查询出商品信息

select stockCount from seckill\_good where id=1 for update;

//2.根据商品信息生成订单

insert into seckill\_order (id,good\_id) values (null,1);

//3.修改商品stockCount减一

update seckill\_good set stockCount=stockCount-1 where id=1;

//4.提交事务

commit;

以上，在对id = 1的记录修改前，先通过for update的方式进行加锁，然后再进行修改。这就是比较典型的悲观锁策略。

如果以上修改库存的代码发生并发，同一时间只有一个线程可以开启事务并获得id=1的锁，其它的事务必须等本次事务提交之后才能执行。这样我们可以保证当前的数据不会被其它事务修改。

我们使用select\_for\_update，另外一定要写在事务中.

注意：要使用悲观锁，我们必须关闭mysql数据库中自动提交的属性，命令set autocommit=0;即可关闭，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。

悲观锁的实现，往往依靠数据库提供的锁机制。在数据库中，悲观锁的流程如下：

* 在对记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。
* 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。具体响应方式由开发者根据实际需要决定。
* 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。
* 其间如果有其他事务对该记录做加锁的操作，都要等待当前事务解锁或直接抛出异常。

### 数据库乐观锁

使用乐观锁就不需要借助数据库的锁机制了。

乐观锁的概念中其实已经阐述了他的具体实现细节：主要就是两个步骤：冲突检测和数据更新。其实现方式有一种比较典型的就是**Compare and Swap(CAS)技术**。

CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，**失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。**

CAS的实现中，在表中增加一个version字段，操作前先查询version信息，在数据提交时检查version字段是否被修改，如果没有被修改则进行提交，否则认为是过期数据。

比如前面的扣减库存问题，通过乐观锁可以实现如下：

//1.查询出商品信息

select stockCount, version from seckill\_good where id=1;

//2.根据商品信息生成订单

insert into seckill\_order (id,good\_id) values (null,1);

//3.修改商品库存

update seckill\_good set stockCount=stockCount-1, version = version+1 where id=1, version=version;

以上，我们在更新之前，先查询一下库存表中当前版本（version），然后在做update的时候，以version 作为一个修改条件。

当我们提交更新的时候，判断数据库表对应记录的当前version与第一次取出来的version进行比对，如果数据库表当前version与第一次取出来的version相等，则予以更新，否则认为是过期数据。

CAS 乐观锁有两个问题：

(1) CAS 存在一个比较重要的问题，即**ABA问题**. 解决的办法是version字段顺序递增。

(2) 乐观锁的方式，在高并发时，只有一个线程能执行成功，会造成大量的失败，这给用户的体验显然是很不好的。

### Zookeeper分布式锁

除了在数据库层面加分布式锁，通常还可以使用以下更高性能、更高可用的分布式锁：

* 分布式缓存（如redis）锁
* 分布式协调（如zookeeper）锁

有关zookeeper分布式锁的原理和实现，具体请参见下面的博客：  
[Zookeeper 分布式锁 （图解+秒懂+史上最全）](https://www.cnblogs.com/crazymakercircle/p/14504520.html)

### Redis分布式锁

本文重点介绍Redis分布式锁，分为两个维度进行介绍：

（1）基于Jedis手工造轮子分布式锁

（2）介绍Redission 分布式锁的使用和原理。

### 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

## 手工造轮子：基于Jedis 的API实现分布式锁

我们首先讲解 Jedis 普通分布式锁实现，并且是纯手工的模式，从最为基础的Redis命令开始。

只有充分了解与分布式锁相关的普通Redis命令，才能更好的了解高级的Redis分布式锁的实现，因为高级的分布式锁的实现完全基于普通Redis命令。

### Redis几种架构

Redis发展到现在，几种常见的部署架构有：

* 单机模式；
* 主从模式；
* 哨兵模式；
* 集群模式；

从分布式锁的角度来说， 无论是单机模式、主从模式、哨兵模式、集群模式，其原理都是类同的。 只是主从模式、哨兵模式、集群模式的更加的高可用、或者更加高并发。

所以，接下来先基于单机模式，基于Jedis手工造轮子实现自己的分布式锁。

### 首先看两个命令：

Redis分布式锁机制，主要借助setnx和expire两个命令完成。

**setnx命令:**

SETNX 是SET if Not eXists的简写。将 key 的值设为 value，当且仅当 key 不存在; 若给定的 key 已经存在，则 SETNX 不做任何动作。

下面为客户端使用示例：

127.0.0.1:6379> set lock "unlock"

OK

127.0.0.1:6379> setnx lock "unlock"

(integer) 0

127.0.0.1:6379> setnx lock "lock"

(integer) 0

127.0.0.1:6379>

**expire命令:**

expire命令为 key 设置生存时间，当 key 过期时(生存时间为 0 )，它会被自动删除. 其格式为：

EXPIRE key seconds

下面为客户端使用示例：

127.0.0.1:6379> expire lock 10

(integer) 1

127.0.0.1:6379> ttl lock

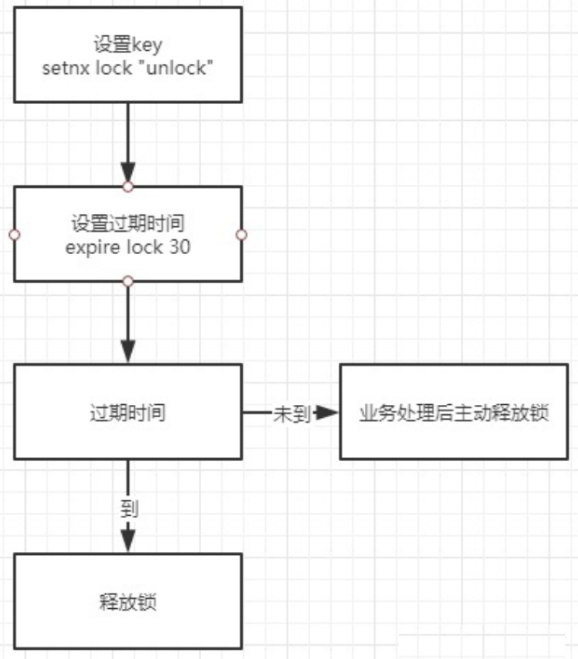
8

### 基于Jedis API的分布式锁的总体流程：

通过Redis的setnx、expire命令可以实现简单的锁机制：

* key不存在时创建，并设置value和过期时间，返回值为1；成功获取到锁；
* 如key存在时直接返回0，抢锁失败；
* 持有锁的线程释放锁时，手动删除key； 或者过期时间到，key自动删除，锁释放。

线程调用setnx方法成功返回1认为加锁成功，其他线程要等到当前线程业务操作完成释放锁后，才能再次调用setnx加锁成功。



**以上简单redis分布式锁的问题：**

如果出现了这么一个问题：如果setnx是成功的，但是expire设置失败，一旦出现了释放锁失败，或者没有手工释放，那么这个锁永远被占用，其他线程永远也抢不到锁。

所以,需要保障setnx和expire两个操作的原子性，要么全部执行，要么全部不执行，二者不能分开。

解决的办法有两种：

* 使用set的命令时，同时设置过期时间，不再单独使用 expire命令
* 使用lua脚本，将加锁的命令放在lua脚本中原子性的执行

### 简单加锁：使用set的命令时，同时设置过期时间

使用set的命令时，同时设置过期时间的示例如下：

127.0.0.1:6379> set unlock "234" EX 100 NX

(nil)

127.0.0.1:6379>

127.0.0.1:6379> set test "111" EX 100 NX

OK

这样就完美的解决了分布式锁的原子性； set 命令的完整格式：

set key value [EX seconds] [PX milliseconds] [NX|XX]

EX seconds：设置失效时长，单位秒

PX milliseconds：设置失效时长，单位毫秒

NX：key不存在时设置value，成功返回OK，失败返回(nil)

XX：key存在时设置value，成功返回OK，失败返回(nil)

使用set命令实现加锁操作，先展示加锁的简单代码实习，再带大家慢慢解释为什么这样实现。

**加锁的简单代码实现**

package com.crazymaker.springcloud.standard.lock;

@Slf4j

@Data

@AllArgsConstructor

public class JedisCommandLock {

private RedisTemplate redisTemplate;

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，我们加锁用到了**Jedis的set Api**：

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前面描述的四个条件中的三个。

* 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），不会被永远占用（而发生死锁）。
* 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。

### 基于Jedis 的API实现简单解锁代码

还是先展示代码，再带大家慢慢解释为什么这样实现。

**解锁的简单代码实现**：

package com.crazymaker.springcloud.standard.lock;

@Slf4j

@Data

@AllArgsConstructor

public class JedisCommandLock {

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;

}

}

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）。

第一行代码，我们写了一个简单的Lua脚本代码。

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。那么为什么执行eval()方法可以确保原子性，源于Redis的特性.

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

**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);

}

}

## 再造轮子：基于Lua脚本实现分布式锁

### lua脚本的好处

前面提到，在redis中执行lua脚本，有如下的好处：

**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。那么为什么执行eval()方法可以确保原子性，源于Redis的特性.

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

所以：

大部分的开源框架（如 redission）中的分布式锁组件，都是用纯lua脚本实现的。

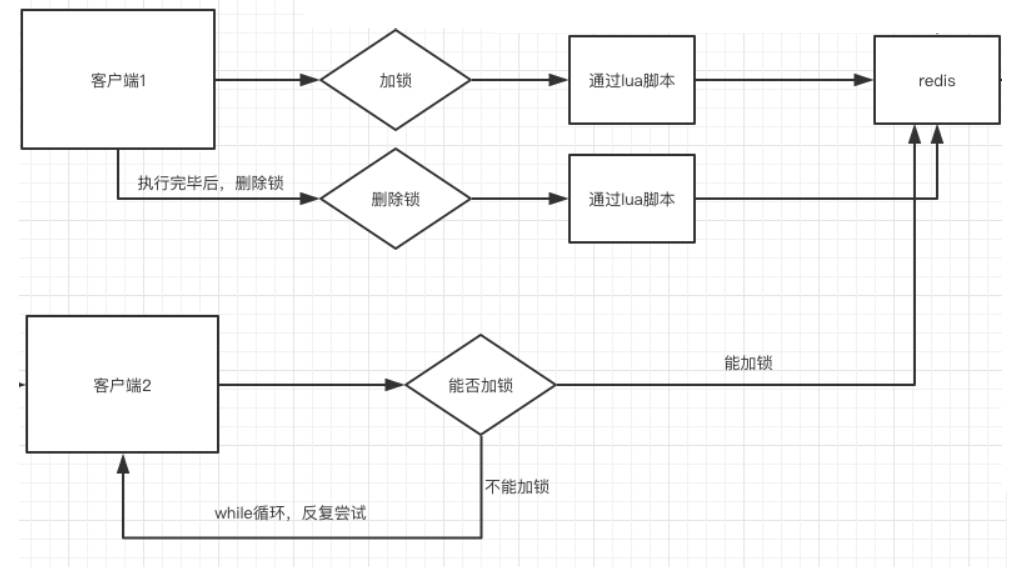
题外话： lua脚本是**高并发、高性能的必备脚本语言**

那么，我们也来模拟一下

**基于纯Lua脚本的分布式锁的执行流程**

加锁和删除锁的操作，使用纯lua进行封装，保障其执行时候的原子性。

基于纯Lua脚本实现分布式锁的执行流程，大致如下：



### 加锁的Lua脚本： lock.lua

--- -1 failed

--- 1 success

---

local key = KEYS[1]

local requestId = KEYS[2]

local ttl = tonumber(KEYS[3])

local result = redis.call('setnx', key, requestId)

if result == 1 then

--PEXPIRE:以毫秒的形式指定过期时间

redis.call('pexpire', key, ttl)

else

result = -1;

-- 如果value相同，则认为是同一个线程的请求，则认为重入锁

local value = redis.call('get', key)

if (value == requestId) then

result = 1;

redis.call('pexpire', key, ttl)

end

end

-- 如果获取锁成功，则返回 1

return result

### 解锁的Lua脚本： unlock.lua：

--- -1 failed

--- 1 success

-- unlock key

local key = KEYS[1]

local requestId = KEYS[2]

local value = redis.call('get', key)

if value == requestId then

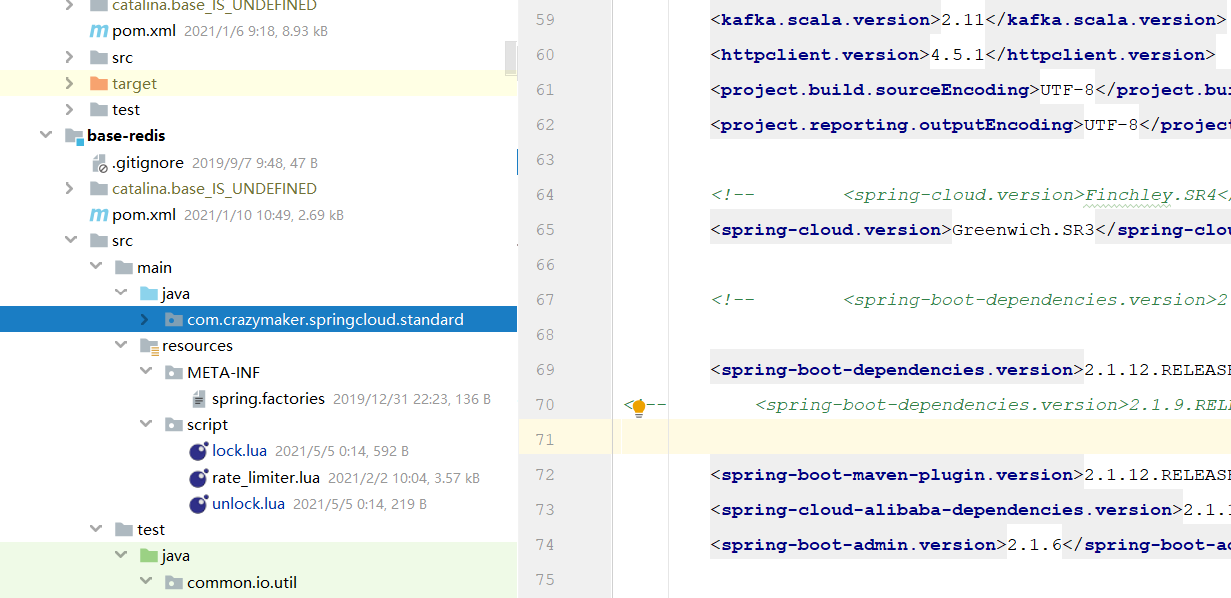
redis.call('del', key);

return 1;

end

return -1

### 两个文件，放在资源文件夹下备用：



## 在Java中调用lua脚本，完成加锁操作

package com.crazymaker.springcloud.standard.lock;

import com.crazymaker.springcloud.common.exception.BusinessException;

import com.crazymaker.springcloud.common.util.IOUtil;

import com.crazymaker.springcloud.standard.context.SpringContextUtil;

import com.crazymaker.springcloud.standard.lua.ScriptHolder;

import lombok.extern.slf4j.Slf4j;

import org.apache.commons.lang3.StringUtils;

import org.springframework.data.redis.core.RedisCallback;

import org.springframework.data.redis.core.RedisTemplate;

import org.springframework.data.redis.core.script.DefaultRedisScript;

import org.springframework.data.redis.core.script.RedisScript;

import java.util.ArrayList;

import java.util.List;

@Slf4j

public class InnerLock {

private RedisTemplate redisTemplate;

public static final Long LOCKED = Long.valueOf(1);

public static final Long UNLOCKED = Long.valueOf(1);

public static final int EXPIRE = 2000;

String key;

String requestId; // lockValue 锁的value ,代表线程的uuid

/\*\*

\* 默认为2000ms

\*/

long expire = 2000L;

private volatile boolean isLocked = false;

private RedisScript lockScript;

private RedisScript unLockScript;

public InnerLock(String lockKey, String requestId) {

this.key = lockKey;

this.requestId = requestId;

lockScript = ScriptHolder.getLockScript();

unLockScript = ScriptHolder.getUnlockScript();

}

/\*\*

\* 抢夺锁

\*/

public void lock() {

if (null == key) {

return;

}

try {

List<String> redisKeys = new ArrayList<>();

redisKeys.add(key);

redisKeys.add(requestId);

redisKeys.add(String.valueOf(expire));

Long res = (Long) getRedisTemplate().execute(lockScript, redisKeys);

isLocked = false;

} catch (Exception e) {

e.printStackTrace();

throw BusinessException.builder().errMsg("抢锁失败").build();

}

}

/\*\*

\* 有返回值的抢夺锁

\*

\* @param millisToWait

\*/

public boolean lock(Long millisToWait) {

if (null == key) {

return false;

}

try {

List<String> redisKeys = new ArrayList<>();

redisKeys.add(key);

redisKeys.add(requestId);

redisKeys.add(String.valueOf(millisToWait));

Long res = (Long) getRedisTemplate().execute(lockScript, redisKeys);

return res != null && res.equals(LOCKED);

} catch (Exception e) {

e.printStackTrace();

throw BusinessException.builder().errMsg("抢锁失败").build();

}

}

//释放锁

public void unlock() {

if (key == null || requestId == null) {

return;

}

try {

List<String> redisKeys = new ArrayList<>();

redisKeys.add(key);

redisKeys.add(requestId);

Long res = (Long) getRedisTemplate().execute(unLockScript, redisKeys);

// boolean unlocked = res != null && res.equals(UNLOCKED);

} catch (Exception e) {

e.printStackTrace();

throw BusinessException.builder().errMsg("释放锁失败").build();

}

}

private RedisTemplate getRedisTemplate() {

if(null==redisTemplate)

{

redisTemplate= (RedisTemplate) SpringContextUtil.getBean("stringRedisTemplate");

}

return redisTemplate;

}

}

### 在Java中调用lua脚本，完成加锁操作

下一步，实现Lock接口, 完成JedisLock的分布式锁。

其加锁操作，通过调用 lock.lua脚本完成，代码如下：

package com.crazymaker.springcloud.standard.lock;

import com.crazymaker.springcloud.common.exception.BusinessException;

import com.crazymaker.springcloud.common.util.ThreadUtil;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.extern.slf4j.Slf4j;

import org.springframework.data.redis.core.RedisTemplate;

import org.springframework.data.redis.core.script.RedisScript;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

@Slf4j

@Data

@AllArgsConstructor

public class JedisLock implements Lock {

private RedisTemplate redisTemplate;

RedisScript<Long> lockScript = null;

RedisScript<Long> unLockScript = null;

public static final int DEFAULT\_TIMEOUT = 2000;

public static final Long LOCKED = Long.valueOf(1);

public static final Long UNLOCKED = Long.valueOf(1);

public static final Long WAIT\_GAT = Long.valueOf(200);

public static final int EXPIRE = 2000;

String key;

String lockValue; // lockValue 锁的value ,代表线程的uuid

/\*\*

\* 默认为2000ms

\*/

long expire = 2000L;

public JedisLock(String lockKey, String lockValue) {

this.key = lockKey;

this.lockValue = lockValue;

}

private volatile boolean isLocked = false;

private Thread thread;

/\*\*

\* 获取一个分布式锁 , 超时则返回失败

\*

\* @return 获锁成功 - true | 获锁失败 - false

\*/

@Override

public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {

//本地可重入

if (isLocked && thread == Thread.currentThread()) {

return true;

}

expire = unit != null ? unit.toMillis(time) : DEFAULT\_TIMEOUT;

long startMillis = System.currentTimeMillis();

Long millisToWait = expire;

boolean localLocked = false;

int turn = 1;

while (!localLocked) {

localLocked = this.lockInner(expire);

if (!localLocked) {

millisToWait = millisToWait - (System.currentTimeMillis() - startMillis);

startMillis = System.currentTimeMillis();

if (millisToWait > 0L) {

/\*\*

\* 还没有超时

\*/

ThreadUtil.sleepMilliSeconds(WAIT\_GAT);

log.info("睡眠一下，重新开始，turn:{},剩余时间：{}", turn++, millisToWait);

} else {

log.info("抢锁超时");

return false;

}

} else {

isLocked = true;

localLocked = true;

}

}

return isLocked;

}

/\*\*

\* 有返回值的抢夺锁

\*

\* @param millisToWait

\*/

public boolean lockInner(Long millisToWait) {

if (null == key) {

return false;

}

try {

List<String> redisKeys = new ArrayList<>();

redisKeys.add(key);

redisKeys.add(lockValue);

redisKeys.add(String.valueOf(millisToWait));

Long res = (Long) redisTemplate.execute(lockScript, redisKeys);

return res != null && res.equals(LOCKED);

} catch (Exception e) {

e.printStackTrace();

throw BusinessException.builder().errMsg("抢锁失败").build();

}

}

}

### 实现JUC的Lock显示锁接口，实现一个简单的分布式锁

其解锁操作，通过调用unlock.lua脚本完成，代码如下：

package com.crazymaker.springcloud.standard.lock;

import com.crazymaker.springcloud.common.exception.BusinessException;

import com.crazymaker.springcloud.common.util.ThreadUtil;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.extern.slf4j.Slf4j;

import org.springframework.data.redis.core.RedisTemplate;

import org.springframework.data.redis.core.script.RedisScript;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

@Slf4j

@Data

@AllArgsConstructor

public class JedisLock implements Lock {

private RedisTemplate redisTemplate;

RedisScript<Long> lockScript = null;

RedisScript<Long> unLockScript = null;

//释放锁

@Override

public void unlock() {

if (key == null || requestId == null) {

return;

}

try {

List<String> redisKeys = new ArrayList<>();

redisKeys.add(key);

redisKeys.add(requestId);

Long res = (Long) redisTemplate.execute(unLockScript, redisKeys);

} catch (Exception e) {

e.printStackTrace();

throw BusinessException.builder().errMsg("释放锁失败").build();

}

}

}

### 编写RedisLockService用于管理JedisLock

编写个分布式锁服务，用于加载lua脚本，创建 分布式锁，代码如下：

package com.crazymaker.springcloud.standard.lock;

import com.crazymaker.springcloud.common.util.IOUtil;

import lombok.Data;

import lombok.extern.slf4j.Slf4j;

import org.apache.commons.lang3.StringUtils;

import org.springframework.data.redis.core.RedisTemplate;

import org.springframework.data.redis.core.script.DefaultRedisScript;

import org.springframework.data.redis.core.script.RedisScript;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Lock;

@Slf4j

@Data

public class RedisLockService

{

private RedisTemplate redisTemplate;

static String lockLua = "script/lock.lua";

static String unLockLua = "script/unlock.lua";

static RedisScript<Long> lockScript = null;

static RedisScript<Long> unLockScript = null;

{

String script = IOUtil.loadJarFile(RedisLockService.class.getClassLoader(),lockLua);

// String script = FileUtil.readString(lockLua, Charset.forName("UTF-8" ));

if(StringUtils.isEmpty(script))

{

log.error("lua load failed:"+lockLua);

}

lockScript = new DefaultRedisScript<>(script, Long.class);

// script = FileUtil.readString(unLockLua, Charset.forName("UTF-8" ));

script = IOUtil.loadJarFile(RedisLockService.class.getClassLoader(),unLockLua);

if(StringUtils.isEmpty(script))

{

log.error("lua load failed:"+unLockLua);

}

unLockScript = new DefaultRedisScript<>(script, Long.class);

}

public RedisLockService(RedisTemplate redisTemplate)

{

this.redisTemplate = redisTemplate;

}

public Lock getLock(String lockKey, String lockValue) {

JedisLock lock=new JedisLock(lockKey,lockValue);

lock.setRedisTemplate(redisTemplate);

lock.setLockScript(lockScript);

lock.setUnLockScript(unLockScript);

return lock;

}

}

### 测试用例

接下来，终于可以上测试用例了

package com.crazymaker.springcloud.lock;

@Slf4j

@RunWith(SpringRunner.class)

@SpringBootTest(classes = {DemoCloudApplication.class})

// 指定启动类

public class RedisLockTest {

@Resource

RedisLockService redisLockService;

private ExecutorService pool = Executors.newFixedThreadPool(10);

@Test

public void testLock() {

int threads = 10;

final int[] count = {0};

CountDownLatch countDownLatch = new CountDownLatch(threads);

long start = System.currentTimeMillis();

for (int i = 0; i < threads; i++) {

pool.submit(() ->

{

String lockValue = UUID.randomUUID().toString();

try {

Lock lock = redisLockService.getLock("test:lock:1", lockValue);

boolean locked = lock.tryLock(10, TimeUnit.SECONDS);

if (locked) {

for (int j = 0; j < 1000; j++) {

count[0]++;

}

log.info("count = " + count[0]);

lock.unlock();

} else {

System.out.println("抢锁失败");

}

} catch (Exception e) {

e.printStackTrace();

}

countDownLatch.countDown();

});

}

try {

countDownLatch.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("10个线程每个累加1000为： = " + count[0]);

//输出统计结果

float time = System.currentTimeMillis() - start;

System.out.println("运行的时长为(ms)：" + time);

System.out.println("每一次执行的时长为(ms)：" + time / count[0]);

}

}

执行用例，结果如下：

2021-05-04 23:02:11.900 INFO 22120 --- [pool-1-thread-7] c.c.springcloud.lock.RedisLockTest LN:50 count = 6000

2021-05-04 23:02:11.901 INFO 22120 --- [pool-1-thread-1] c.c.springcloud.standard.lock.JedisLock LN:81 睡眠一下，重新开始，turn:3,剩余时间：9585

2021-05-04 23:02:11.902 INFO 22120 --- [pool-1-thread-1] c.c.springcloud.lock.RedisLockTest LN:50 count = 7000

2021-05-04 23:02:12.100 INFO 22120 --- [pool-1-thread-4] c.c.springcloud.standard.lock.JedisLock LN:81 睡眠一下，重新开始，turn:3,剩余时间：9586

2021-05-04 23:02:12.101 INFO 22120 --- [pool-1-thread-5] c.c.springcloud.standard.lock.JedisLock LN:81 睡眠一下，重新开始，turn:3,剩余时间：9585

2021-05-04 23:02:12.101 INFO 22120 --- [pool-1-thread-8] c.c.springcloud.standard.lock.JedisLock LN:81 睡眠一下，重新开始，turn:3,剩余时间：9585

2021-05-04 23:02:12.101 INFO 22120 --- [pool-1-thread-4] c.c.springcloud.lock.RedisLockTest LN:50 count = 8000

2021-05-04 23:02:12.102 INFO 22120 --- [pool-1-thread-8] c.c.springcloud.lock.RedisLockTest LN:50 count = 9000

2021-05-04 23:02:12.304 INFO 22120 --- [pool-1-thread-5] c.c.springcloud.standard.lock.JedisLock LN:81 睡眠一下，重新开始，turn:4,剩余时间：9383

2021-05-04 23:02:12.307 INFO 22120 --- [pool-1-thread-5] c.c.springcloud.lock.RedisLockTest LN:50 count = 10000

10个线程每个累加1000为： = 10000

运行的时长为(ms)：827.0

每一次执行的时长为(ms)：0.0827

## STW导致的锁过期问题

下面有一个简单的使用锁的例子，在10秒内占着锁：

//写数据到文件

function writeData(filename, data) {

boolean locked = lock.tryLock(10, TimeUnit.SECONDS);

if (!locked) {

throw 'Failed to acquire lock';

}

try {

//将数据写到文件

var file = storage.readFile(filename);

var updated = updateContents(file, data);

storage.writeFile(filename, updated);

} finally {

lock.unlock();

}

}

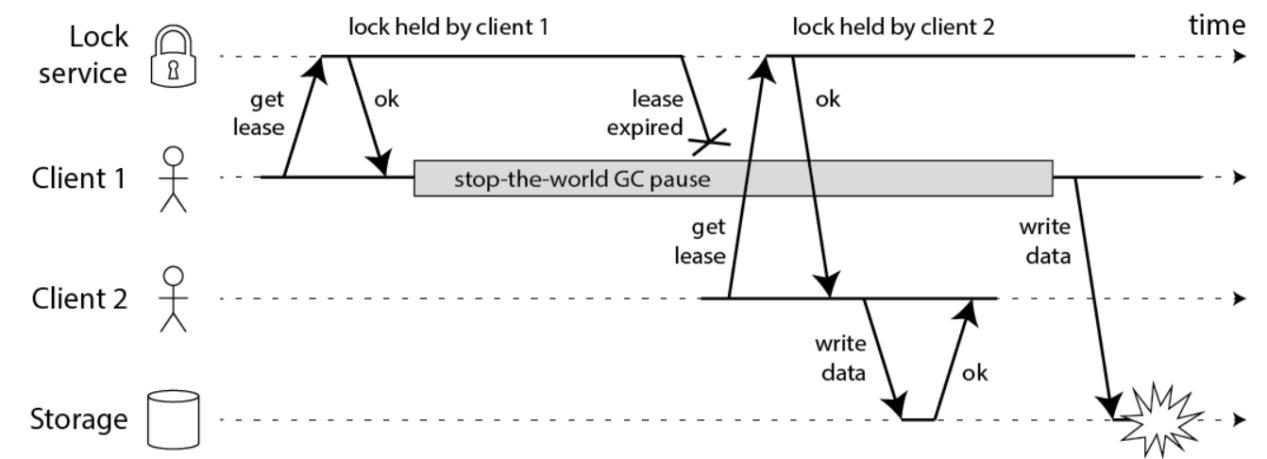
问题是：如果在写文件过程中，发生了 fullGC，并且其时间跨度较长， 超过了10秒， 那么，分布式就自动释放了。

在此过程中，client2 抢到锁，写了文件。

client1 的fullGC完成后，也继续写文件，**注意，此时client1 的并没有占用锁**，此时写入会导致文件数据错乱，发生线程安全问题。

这就是STW导致的锁过期问题。

**STW导致的锁过期问题，具体如下图所示：**

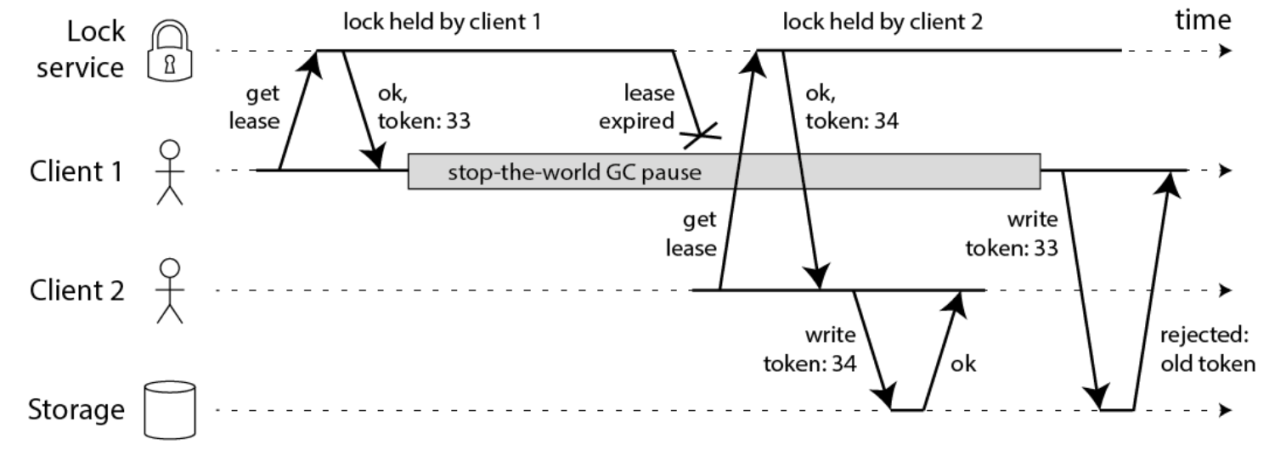


STW导致的锁过期问题,大概的解决方案，有：

1： 模拟CAS乐观锁的方式，增加版本号

2：watch dog自动延期机制

1： 模拟CAS乐观锁的方式，增加版本号（如下图中的token）



此方案如果要实现，需要调整业务逻辑，与之配合，所以会入侵代码。

2：watch dog自动延期机制

客户端1加锁的锁key默认生存时间才30秒，如果超过了30秒，客户端1还想一直持有这把锁，怎么办呢？

简单！只要客户端1一旦加锁成功，就会启动一个watch dog看门狗，他是一个后台线程，会每隔10秒检查一下，如果客户端1还持有锁key，那么就会不断的延长锁key的生存时间。

redission，采用的就是这种方案， 此方案不会入侵业务代码。

注意：

单机版的watch dog 并不能解决 STW的过期问题， 需要分布式版本的 watch dog， 独立的看门狗服务。

锁删除之后， 取消看门狗服务的 对应的key记录， 当然，这就使得系统变得复杂， 还要保证看门狗服务的高并发、高可用、数据一致性的问题。