项目申请书

项目名称：实现基于 Mysql、Redis 等常用中间件的分布式锁

项目主导师：郑祖岭

申请人：陈建辉

日期：2023.05.11

邮箱：2294198058@qq.com

一.项目背景

1.项目基本需求

issue仓库地址：Implementing Distributed Lock Based on Common Middleware like Mysql and Redis · Issue #4 · ao-space/platform-base (github.com)

我的fork仓库地址: Huahuoao/platform-base: AO.space is focused on protecting personal data ownership and creating a truly personal owned digital space. (github.com)

（目前issue要求的**分布式锁已经实现**，在yml里面配置一下redis和mysql连接就可以运行测试类测试）

实现基于 Mysql、Redis 等常用中间件的分布式锁

描述

熟悉已发布的开源项目 ， 模仿 RedisReentrantLock 实现基于 Mysql 的分布式锁、基于 Mysql、Redis 的分布式读写锁；编写相关的测试用例；撰写相关设计、使用文档

项目产出要求

- 实现基于 Mysql 的分布式锁（互斥锁）：具备可重入特性；具备锁失效机制，防止死锁；具备非阻塞锁特性。
- 实现基于 Mysql、Redis 的分布式读写锁：具备读锁可重入、写锁可重入特性；具备锁失效机制，防止死锁；具备非阻塞锁特性。
- 编写相应的单元测试和集成测试。
- 编写相应的设计、使用文档。
- 代码符合规范： https://google.github.io/styleguide/javaguide.html

项目技术要求

- 熟悉 Java 语言
- 熟悉 Quarkus 框架
- 熟悉 Mysql、Redis 等中间件
- 熟悉多线程编程

1.Redis实现分布式锁

> Redis可以用作分布式锁的原因有以下几点：
>
> 1. **高性能和低延迟**：Redis是一个内存数据库，具有快速读写速度和低延迟的特点。这使得在Redis中实现分布式锁可以获得较高的性能，并且不会对系统的响应时间产生显著影响。
> 2. **原子操作**：Redis提供了一组原子操作，如`SETNX`（设置键不存在时才设置值）和`EXPIRE`（设置过期时间），这些操作可以保证在并发环境下对锁的获取和释放是原子的。可以使用Lua脚本保证原子性

2.Mysql实现分布式锁

> 1. 类似于Redis的机制，对于一把锁设置一个唯一的id，利用**主键唯一**原则可以保证锁的互斥性。
> 2. select xxxx for update; 可以很好的保证查询更改的**原子性**
> 3. 利用Quarkus的**事务注解**，可以保证加锁，解锁整个事务的原子性
> 4. 对于**锁的过期机制**，没有redis那种原生的过期时间，但是可以通过**懒加载**机制，对每次锁操作之前进行一次时间校验。

需要保证

- 锁可重入：需要标记线程，对同一线程的锁进行重入处理。

- 锁失效机制：配备过期时间，以及解锁。
- 防止死锁： 获取锁失败，或者运行过程中解锁失败，有自动解锁机制，避免死锁。
- 具备非阻塞锁特性：尝试多次获取锁之后自动退出争抢，避免阻塞。

二.技术方法及其可行性

1.Redis相关

- Redis可以使用Lua脚本把一系列操作变为原子操作

具体参考 REDIS | EVAL

- Redis客户端Redission有非常成熟的方案参考。

redission代码仓库

2.Mysql相关

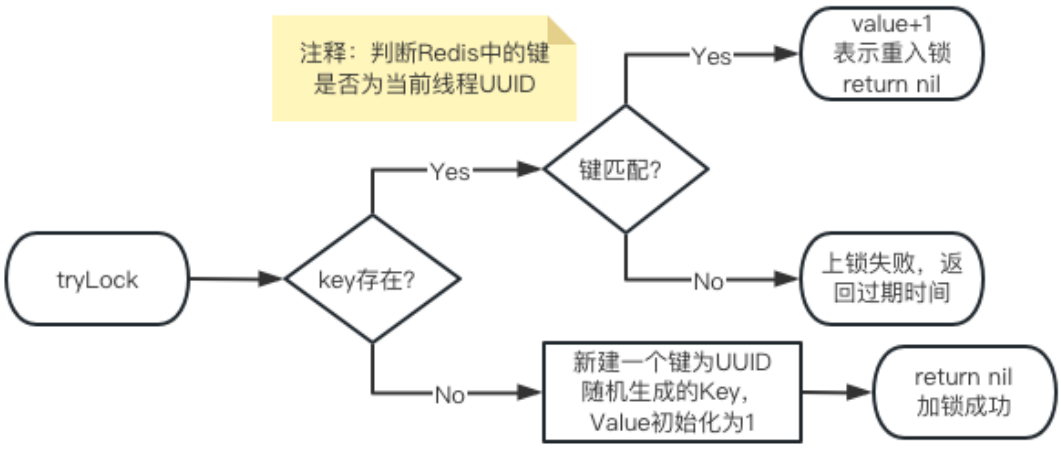
- 乐观锁
- 懒加载思想
- Quarkus事务
- IOC容器的反射机制

三.项目实现细节梳理

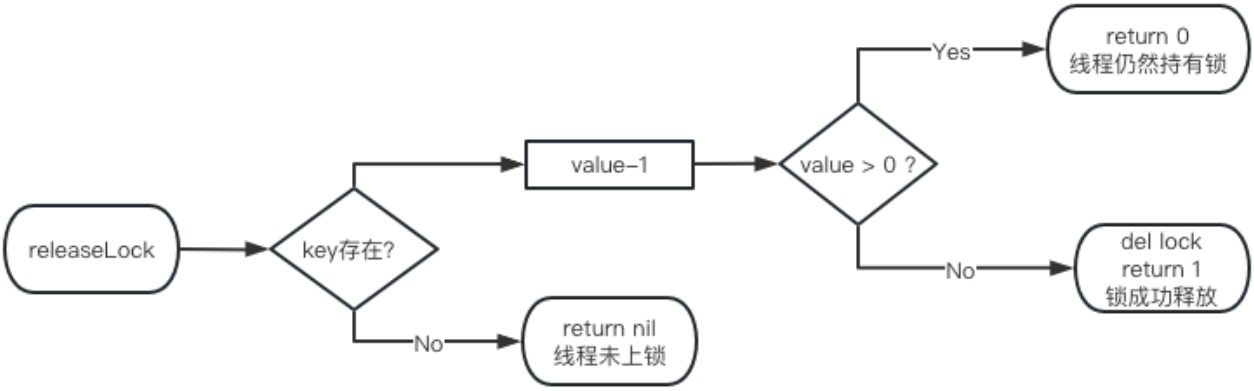
一、Redis可重入锁（社区已实现）

核心流程图

- tryLock



- unLock

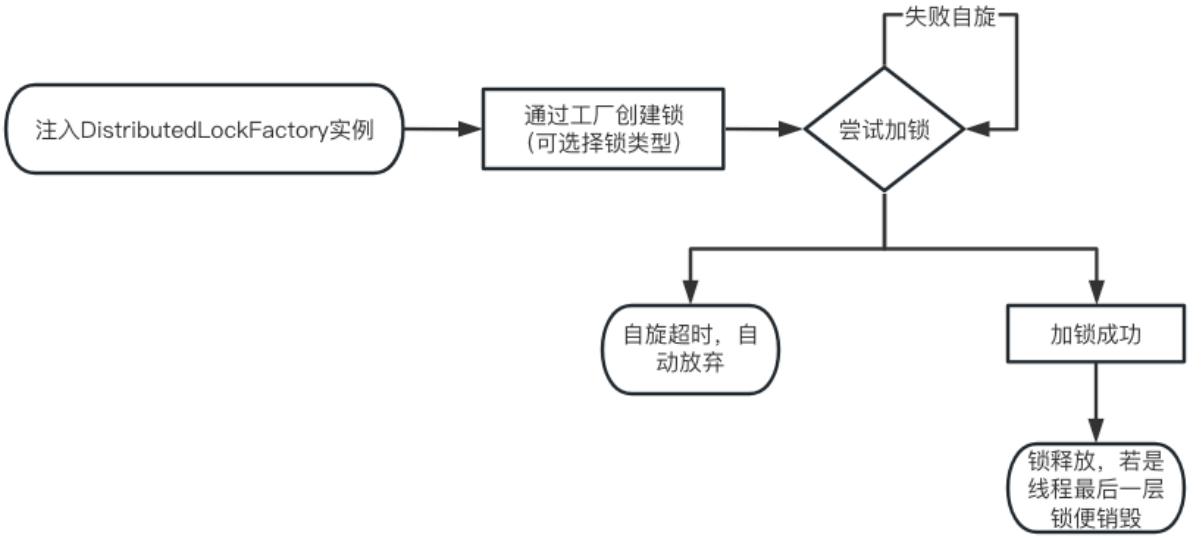


特性实现

> 具备可重入特性；具备锁失效机制，防止死锁；具备非阻塞锁特性。

- 可重入：采用value记录锁的数量，每次解锁-1
- 锁失效：设置了redis数据的过期时间
- 一次lock对应一次unlock，防止死锁
- trylock会循环若干次获取锁，若获取失败，自动退出争抢。保证了非阻塞锁的特性
- 为了保证原子性，对redis的操作均采用lua脚本实现

分布式锁生命周期



二、Redis可重入读写锁

> 写锁为排他锁，读锁允许多线程访问。但是读写锁相互排斥，同一时间检测读写锁只能有一种锁存在。

思路分析

> 读写锁（Read-Write Lock）是一种多线程同步机制，用于管理对共享资源的并发访问。它可以同时支持多个读操作，但在写操作时只能有一个线程进行访问。读写锁的目的是提高并发性能，允许多个线程同时读取共享资源，从而实现读写分离，减少了对资源的互斥访问。

1. 假设创建一个读写锁，设置key为**"taskId"**
2. 那么在读写锁处理的时候，可以把这个key添加前缀
3. 读锁 **"read-taskId"**， 写锁 **"write-taskId"**
4. 这样在读写锁判定的时候可以很方便的来查找redis数据中之间是否存在读/写锁。 （保证读写互斥性）

举例（伪代码）：

```
new lock("task1","read"); // 创建一个key为read-task1的读锁
```

5. 那么在操作redis的时候，就可以把 "read-task1"作为key来创建一个hash结构，代表创建了一个读锁。
6. 同样的，"write-task1"就是这个任务的写锁。因此在创建读/写锁之前先判断**write-task1/read-task1**是否存在。如果存在则返回创建失败，开始自旋

1.读锁

> 读锁需要考虑的问题以及解决思路
>
> 问题：1 读锁与读锁不互相排斥，并且都支持重入
>
> 2 读锁与写锁互斥
>
> 解决思路：
>
> 同名读锁之间共享一个hash结构，用field区分，并且可以分别重入。解锁的时候要判断一下如果hash结构中没有元素了，就把hash删除。如果field value = 0 ，就把这个field删除。

tryLock()核心代码

```
   if (mode.equals("read")) {
            String command = "if redis.call('exists', KEYS[2]) == 1 then "
                    + "    return nil; "    // 如果存在写锁，直接加锁失败。
                    + "else "
                    + "    if redis.call('exists', KEYS[1]) == 0 then "    // 判断指定的key是否存在
                    + "        redis.call('HSET', KEYS[1], ARGV[2], 1) "    // 不存在新增key，value为hash结构
                    + "        redis.call('PEXPIRE', KEYS[1], ARGV[1]) "  // 设置过期时间
                    + "    else "
                // key存在说明已经有读锁创建了，接下来判断这个锁是重入锁，还是其他线程的读锁
                    + "        if redis.call('HEXISTS', KEYS[1], ARGV[2]) == 1 then "

                    + "            redis.call('HINCRBY', KEYS[1], ARGV[2], 1) "  // hash中指定键的值+1
                    + "            redis.call('PEXPIRE', KEYS[1], ARGV[1]) "  // 重置过期时间
```

```
                    + "            return 1; "
                    + "        else "                          // 不是重入锁，创建新锁
                    + "            redis.call('HSET', KEYS[1], ARGV[2], 1) "
                    + "            redis.call('PEXPIRE', KEYS[1], ARGV[1]) "
                    + "        end "
                    + "    end "
                    + "    return 1; "                  // 直接返回1，表示加锁成功
                    + "end";
            //list传入 key分为两种 第一种 id 第二种 id-mode。一起创建一起删除

            list.add(command);
            list.add("2"); // keyNum
            list.add("read-" + key); //hash键   KEYS[1]
            list.add("write-" + key); //写锁的key KEYS[2]
            list.add(timeout.toString()); //ARGV[1]
            list.add(value); // 锁的内容 也就是hash的名字 ARGV[2]
```

unLock()核心代码

```
   if (mode.equals("read")) {
            String command = "if (redis.call('hexists', KEYS[1], ARGV[2]) == 0) then " +
                    "    return nil; " +  // 判断当前客户端之前是否已获取到锁，若没有直接返回null
                    "end; " +
                    "local counter = redis.call('hincrby', KEYS[1], ARGV[2], -1); " + // 锁重入次数-1
                    "if (counter > 0) then " + // 若锁尚未完全释放，需要重置过期时间
                    "    redis.call('pexpire', KEYS[1], ARGV[1]); " +
                    "    return 0; " +  // 返回0表示锁未完全释放
                    "else " +
                    "    redis.call('hdel', KEYS[1], ARGV[2]); " +  //如果hash长度还大于0说明还有读锁在里面
                    "    if redis.call('hlen', KEYS[1]) > 0 then " +
                    "        return 0; " +
                    "    end; " +
                    "    return 1; " +   // 返回1表示锁已完全释放
                    "end; " +
                    "return nil;";
            List<String> list = new ArrayList<>();
            list.add(command);
            list.add("1"); // keyNum
            list.add("read-" + key); //hash键   KEYS[1]
            list.add(timeout.toString()); //ARGV[1]
            list.add(value); // 锁的内容 也就是hash的名字 ARGV[2]
            Response result = redisClient.eval(list);
```

2.写锁

> 写锁比读锁逻辑稍微简单一点，主要和之前的可重入锁逻辑类似，最前面加一条判断现在是否有读锁存在就可以了，保证和读锁的互斥性。

tryLock()核心代码

```
 else if (mode.equals("write")) {
            String command = "if redis.call('exists', KEYS[2]) == 1 then "
                    + "    return 1; " + // 如果存在读锁，直接加锁失败。
                    "end; " +
                    "if (redis.call('exists', KEYS[1]) == 0) then " +  //判断指定的key是否存在
                    "    redis.call('hset', KEYS[1], ARGV[2], 1); " +  //新增key，value为hash结构
                    "    redis.call('pexpire', KEYS[1], ARGV[1]); " + //设置过期时间
                    "    return nil; " +  //直接返回null，表示加锁成功
                    "end; " +
                    "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +  //判断hash中是否存在指定的建
                    "    redis.call('hincrby', KEYS[1], ARGV[2], 1); " + //hash中指定键的值+1
                    "    redis.call('pexpire', KEYS[1], ARGV[1]); " +   //重置过期时间
                    "    return nil; " +     //返回null，表示加锁成功
                    "end; " +
                    "return redis.call('pttl', KEYS[1]);"; //返回key的剩余过期时间，表示加锁失败

            list.add(command);
            list.add("2"); // keyNum
            list.add("write-" + key); //hash键   KEYS[1]
            list.add("read-" + key); //读锁的key KEYS[2]
            list.add(timeout.toString()); //ARGV[1]
            list.add(value); // 锁的内容 也就是hash的名字 ARGV[2]
```

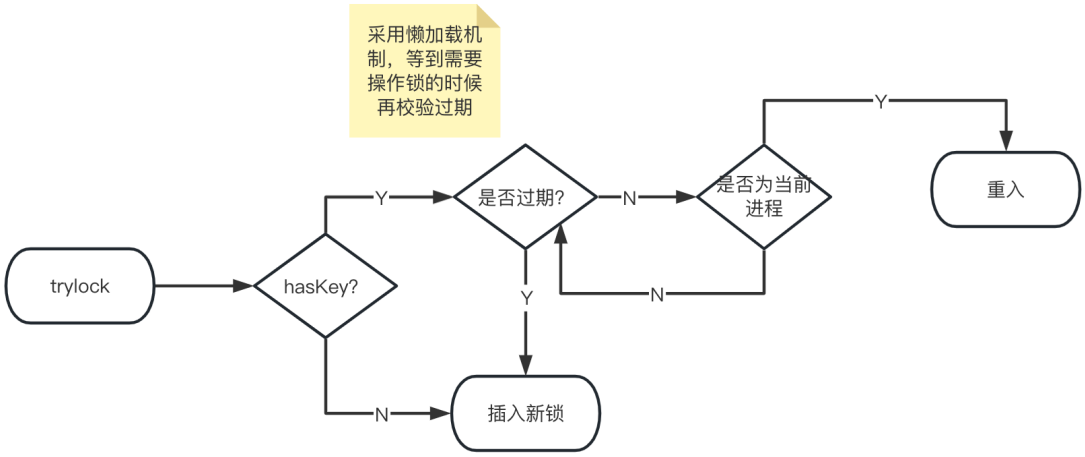
unLock()核心代码

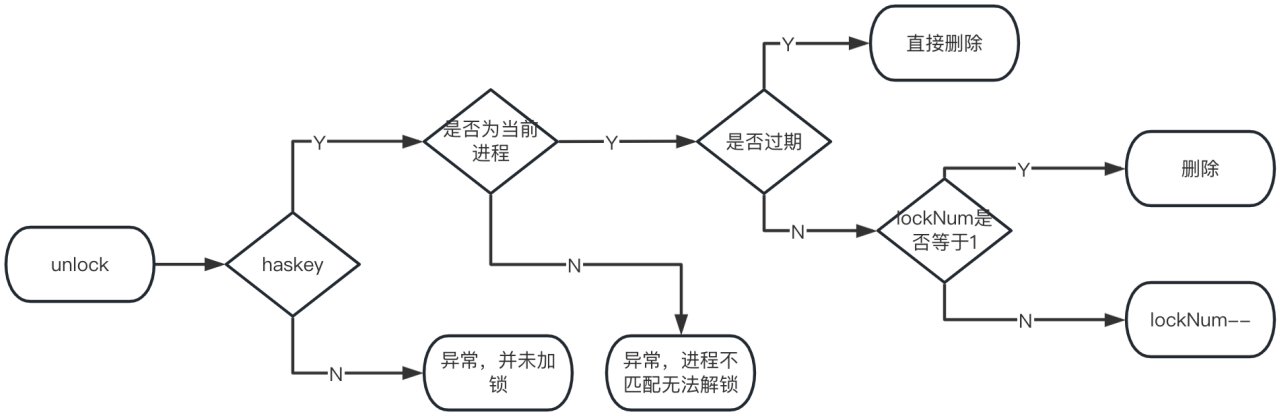
```
private void releaseLock(String key, String value, Integer timeout, String mode) {
        if (mode.equals("read")) {
            String command = "if (redis.call('hexists', KEYS[1], ARGV[2]) == 0) then " +
                    "    return nil; " +  // 判断当前客户端之前是否已获取到锁，若没有直接返回null
                    "end; " +
                    "local counter = redis.call('hincrby', KEYS[1], ARGV[2], -1); " + // 锁重入次数-1
                    "if (counter > 0) then " +  // 若锁尚未完全释放，需要重置过期时间
                    "    redis.call('pexpire', KEYS[1], ARGV[1]); " +
```

```
                    "    return 0; " + // 返回0表示锁未完全释放
                    "else " +
                    "    redis.call('hdel', KEYS[1], ARGV[2]); " + //如果hash长度还大于0说明还有读锁在里面
                    "    if redis.call('hlen', KEYS[1]) > 0 then " +
                    "        return 0; " +
                    "    end; " +
                    "    return 1; " + // 返回1表示锁已完全释放
                    "end; " +
                    "return nil;";
            List<String> list = new ArrayList<>();
            list.add(command);
            list.add("1"); // keyNum
            list.add("read-" + key); //hash键    KEYS[1]
            list.add(timeout.toString()); //ARGV[1]
            list.add(value); // 锁的内容 也就是hash的名字 ARGV[2]
```

三、Mysql可重入锁

1.整体程序逻辑





2.数据表设计

```
create table distributed_lock
(
    id              bigint auto_increment //主键
        primary key,
    lock_key        varchar(255) null, //锁的内容
    lock_num        int          null, //锁的重入次数 默认1
    expiration_time datetime     null, //锁的过期时间
    task_name       varchar(255) null, //任务名称，独立区分
    created_at      datetime     null, //以下继承BaseEntity
    updated_at      datetime     null,
    version         int          null,  //版本号，用来做乐观锁，避免资源争抢导致的并发问题
    constraint id
        unique (id)
);
```

3.难点攻破

1. 为了保证mysql的原子性，必须使用Qurkus的事务处理机制，但是new lock( ) 出来的对象不被框架管理，无法使用框架的事务处理机制，于是实现了一个Service层，但是new出来的对象也无法@Inject，所以就用到了IOC的反射机制，并且在new lock( )的时候采用构造函数获取到这个被框架管理的Service。于是这个Service就拥有了使用事务的能力，通过lock对象传递参数（并且在这个方法加上一层同步类级锁来保证线程安全）。全部交给Service来处理，并且使用事务包裹，同时配合采用了Quarkus自带的@Version（乐观锁）处理机制。

```
 public MysqlReentrantLock(String keyName, String lockValue, Integer timeout) {
        this.keyName = keyName;
        this.lockValue = lockValue;
        this.timeout = timeout * 1000;
        this.service = Arc.container().instance(MysqlLockService.class).get();
    } //构造函数，通过Arc注入
```

2. mysql没有原生的数据过期策略，于是借用了懒加载的思路来实现过期策略，当需要操作锁的时候进数据库查询是否过期。尽可能的不影响整体系统的性能。

4.关键代码

```
 //尝试获取锁
@Transactional
    public boolean tryLock(String id, String uuid) throws InterruptedException {
        DistributedLockEntity lock = this.getLock(id);
        System.out.println("当前操作线程===>" + Thread.currentThread().getName());
        if (lock == null) { //  如果没有这个锁，那就创建
```

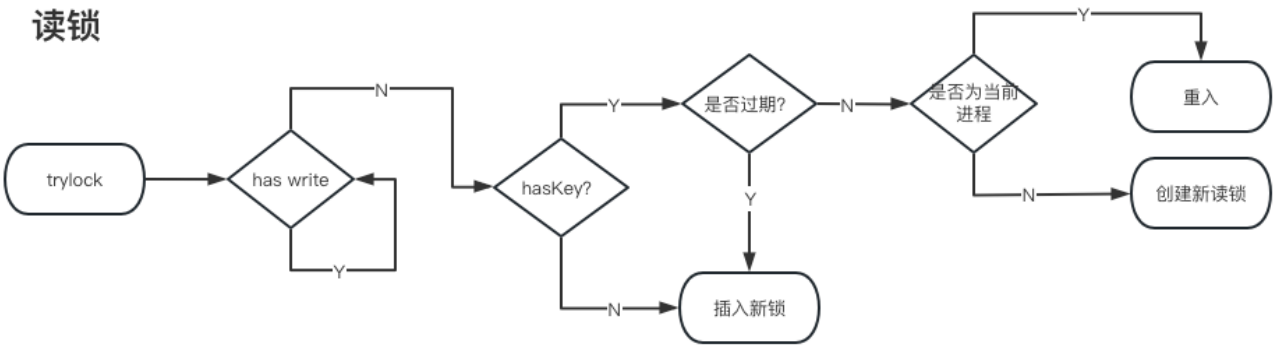
```
            this.newLock(id, uuid);
            return true;
        } else if (lock != null) {  //如果锁存在就要判断锁有没有过期
            boolean expired = this.isExpired(lock.getExpirationTime());
            if (expired) { //如果过期
                this.deleteLock(id); //删除这个锁
                this.newLock(id, uuid); //创建新锁
                      return true;
            } else { //锁没过期
                if (uuid.equals(lock.getLockKey())) { //是否为重入锁
                    this.updateNum(lock.getLockNum() + 1, lock.getTaskName());
                    return true;
                } else {
                    return false;
                }
            }
        }
        return false;
    }
```

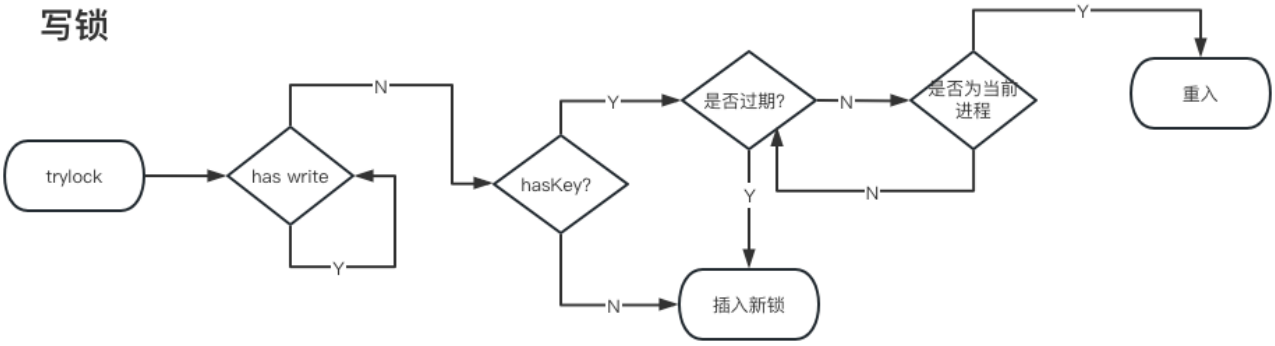
```
@Transactional
    public int unlock(String taskName, String uuid) {
        DistributedLockEntity lock = repository.getLock(taskName,uuid);
        return notHoldLock(lock);
    }

    private int notHoldLock(DistributedLockEntity lock) {
        if (lock == null) {
            return 2; // 异常：该线程没有锁或进程不匹配
        } else {
            if (this.isExpired(lock.getExpirationTime()) || lock.getLockNum() == 1) {
                repository.deleteLock(lock.getId());
            } else {
                lock.setLockNum(lock.getLockNum() - 1);
                repository.updateLockNum(lock);
                return 0; // 减少重入成功
            }
        }
        return 1; // 解锁成功
    }

    @Transactional
    public int unlock(String taskName, String uuid,String mode) {
        DistributedLockEntity lock = repository.getLock(mode+"-"+taskName,uuid);
        return notHoldLock(lock);
    }
```

5.设计架构

四、Mysql实现分布式读写锁

1.整体逻辑架构





2.介绍

> 本质上与普通分布式锁没有大的差别，依然采用分布式锁设计思路。只是在trylock之后需要判断一下数据库是否存在读/写锁，来保证读写互斥。写锁也会与自己互斥。读锁可以创建新锁。

3.关键代码逻辑

```
 @Transactional
    public boolean tryLock(String taskName, String uuid, String mode) throws
InterruptedException {
        if (mode.equals("read")) {    //如果是读锁
```

```
            DistributedLockEntity writeLock = this.getLock("write-" + taskName);
            if (writeLock != null) { //如果写锁不为空
                boolean expired = this.isExpired(writeLock.getExpirationTime());
                if (expired) {
                    this.deleteLock(writeLock.getId());
                }//写锁过期删了
                else {
                    return false;
                }
            }

            DistributedLockEntity lock = this.getLock("read-" + taskName, uuid);  //获取读锁，如果获取到就是重入锁，不是就创建新锁
            if (lock != null) {
                //判断过期
                if (this.isExpired(lock.getExpirationTime())) {
                    this.deleteLock(lock.getId());  //过期删掉
                } else { //没过期就重入
                    lock.setLockNum(lock.getLockNum() + 1);

 lock.setExpirationTime(LocalDateTime.now().plusSeconds(applicationProperties.getLockExpireTime()));
                    repository.updateLockAndTime(lock);
                    return true;
                }
            } else {
                this.newLock("read-" + taskName, uuid);
                return true;
            }
        } else if (mode.equals("write")) {  //进入写锁逻辑
            List<DistributedLockEntity> writeLocks = this.getLockReadWrite("read-" + taskName); //判断是否有读锁
            if (writeLocks != null) {
                for (DistributedLockEntity writeLock : writeLocks) {  //遍历获取到读读锁
                    if (this.isExpired(writeLock.getExpirationTime())) {
                        this.deleteLock(writeLock.getId());  //如果过期就删掉
                    } else {
                        return false; //没过期就false,但凡有一个没过期，都会失败。
                    }
                }
            }
            //这边就是没有读锁了
            DistributedLockEntity lock = this.getLock("write-" + taskName);  //获取写锁
            if (lock != null) { //获取到了就要判断过期
                if (this.isExpired(lock.getExpirationTime())) {
                    this.deleteLock(lock.getId());
                    this.newLock("write-" + taskName, uuid);
                    return true;
                } else {
```

```
                    //判断是否重入
                    if (uuid.equals(lock.getLockKey())) {
                        //重入
                        lock.setLockNum(lock.getLockNum() + 1);

 lock.setExpirationTime(LocalDateTime.now().plusSeconds(applicationProperties.getLockExpireTime()));
                        repository.updateLockAndTime(lock);
                        return true;
                    }
                    return false;
                }
            } else {
                this.newLock("write-" + taskName, uuid);
                return true;
            }
        } else {
            throw new RuntimeException("Lock mode is incorrect, expected value is \"read\" or \"write\"");
        }
        return false;
    }
```

简单测试

```
/*
 * Copyright (c) 2022 Institute of Software Chinese Academy of Sciences (ISCAS)
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */


package xyz.eulix.platform.services.lock;

import io.quarkus.test.junit.QuarkusTest;
import org.jboss.logging.Logger;
import org.junit.jupiter.api.Assertions;
```

```java
import org.junit.jupiter.api.Test;
import xyz.eulix.platform.services.lock.service.MysqlLockService;

import javax.inject.Inject;

@QuarkusTest
public class DistributedLockTest {
    private static final Logger LOG = Logger.getLogger("app.log");

    @Inject
    DistributedLockFactory lockFactory;

    @Test
    void testRedisReentrantLock() throws InterruptedException {
        String keyName = "RedisReentrantLock";
        DistributedLock lock = lockFactory.newRedisReentrantLock(keyName);
        // 加锁
        Boolean isLocked = lock.tryLock();
        if (isLocked) {
            LOG.infov("acquire lock success, keyName:{0}", keyName);
            try {
                if(lock.tryLock()){
                    // 这里写需要处理业务的业务代码
                    LOG.infov("reentrant lock success, keyName:{0}", keyName);
                    LOG.info("do something.");
                    Thread.sleep(3000);
                }
            } finally {
                // 释放锁
                lock.unlock();
                lock.unlock();
                LOG.infov("release lock success, keyName:{0}", keyName);
            }
        } else {
            LOG.infov("acquire lock fail, keyName:{0}", keyName);
        }
        Assertions.assertTrue(isLocked);
    }



    @Test
    void testRedisReadWriteLock() throws InterruptedException {
        String keyName = "RedisReadWriteLock";
        DistributedLock lock = lockFactory.newRedisReadWriteLock(keyName,"write");
        // 加锁
        Boolean isLocked = lock.tryLock();
        if (isLocked) {
            LOG.infov("acquire lock success, keyName:{0}", keyName);
```

```
            try {
                if(lock.tryLock()){
                    // 这里写需要处理业务的业务代码
                    LOG.infov("reentrant lock success, keyName:{0}", keyName);
                    LOG.info("do something.");
                    Thread.sleep(3000);
                }
            } finally {
                // 释放锁
                lock.unlock();
                lock.unlock();
                LOG.infov("release lock success, keyName:{0}", keyName);
            }
        } else {
            LOG.infov("acquire lock fail, keyName:{0}", keyName);
        }
        Assertions.assertTrue(isLocked);
    }
    @Test
    void testMysqlReentrantLock() throws InterruptedException {
        String keyName = "MysqlReentrantLock";
        DistributedLock lock = lockFactory.newMysqlReentrantLock(keyName);
        // 加锁
        Boolean isLocked = lock.tryLock();
        if (isLocked) {
            LOG.infov("acquire lock success, keyName:{0}", keyName);
            try {
                if(lock.tryLock()){
                    // 这里写需要处理业务的业务代码
                    LOG.infov("reentrant lock success, keyName:{0}", keyName);
                    LOG.info("do something.");
                    Thread.sleep(3000);
                }
            } finally {
                // 释放锁
                lock.unlock();
                lock.unlock();
                LOG.infov("release lock success, keyName:{0}", keyName);
            }
        } else {
            LOG.infov("acquire lock fail, keyName:{0}", keyName);
        }
        Assertions.assertTrue(isLocked);
    }
    @Test
    void testMysqlReadWriteLock() throws InterruptedException {
        String keyName = "MysqlReadWriteLock";
        DistributedLock lock = lockFactory.newMysqlReadWriteLock(keyName,"write");
        // 加锁
```

```
        Boolean isLocked = lock.tryLock();
        if (isLocked) {
            LOG.infov("acquire lock success, keyName:{0}", keyName);
            try {
                if(lock.tryLock()){
                    // 这里写需要处理业务的业务代码
                    LOG.infov("reentrant lock success, keyName:{0}", keyName);
                    LOG.info("do something.");
                    Thread.sleep(3000);
                }
            } finally {
                // 释放锁
                lock.unlock();
                lock.unlock();
                LOG.infov("release lock success, keyName:{0}", keyName);
            }
        } else {
            LOG.infov("acquire lock fail, keyName:{0}", keyName);
        }
        Assertions.assertTrue(isLocked);
    }
}
```

> 测试环境：macbookAir m1，8g内存
>
> 测试结果：均满足issue要求，所有锁可以保证本地线程安全以及分布式环境下线程安全。在双主机，各开20条线程的环境下测试。没有发生死锁，阻塞等情况。效率良好

锁效率时间表（参考）

> 采用24条线程同时运行的情况下。分别加锁解锁，通过原子类来计算平均时间。
>
> 因为在测试类中，**初次加锁，需要与redis和mysql建立数据库连接，所以耗时相对较长。如果在持有数据库连接的情况下，可以参考重入锁的耗时。**相对而言，redis分布式锁效率会高于mysql分布式锁。

锁分类	平均加锁时间（初次加锁）	平均加锁时间（重入锁）	平均解锁时间
Redis分布式锁	68ms	4ms	4ms
Redis读锁	65ms	3ms	3ms
Redis写锁	63ms	3ms	3ms
Mysql分布式锁	170ms	45ms	19ms
Mysql读锁	180ms	50ms	13ms
Mysql写锁	174ms	36ms	12ms

```
@Test
    void lockEfficiencyTest() throws InterruptedException { //6ms
```

```
        AtomicLong lockTime = new AtomicLong();
        AtomicLong unLockTime = new AtomicLong();
        for (int i = 0; i < 24; i++) {
            new Thread(() -> {
                DistributedLock lock =
lockFactory.newRedisReadWriteLock(String.valueOf(UUID.randomUUID()),"write");;
                try {
                    long t1 = System.currentTimeMillis();
                   lock.tryLock();
                    //lock.tryLock();
                    lockTime.addAndGet(System.currentTimeMillis() - t1);
                } catch (Exception e) {
                } finally {
                    long t2 = System.currentTimeMillis();
                  // lock.unlock();
                   lock.unlock();
                    unLockTime.addAndGet(System.currentTimeMillis() - t2);
                }
            }).start();
        }
        Thread.sleep(2000);
        System.out.println("加锁平均用时: "+lockTime.longValue()/24+"ms");
        System.out.println("解锁平均用时: "+unLockTime.longValue()/24+"ms");
    }
```

四.规划

项目第一阶段：5月10日到8月15日

- [x] 实现redis读写锁
- [x] 实现mysql分布式锁
- [x] 实现mysql读写锁
- [x] 编写测试类
- [x] 完成说明文档
- [x] 测试代码
- [ ] 持续优化代码，与社区沟通，新增优化锁特性

> 以上要求已经全部实现，测试了双节点同时运行下，每个节点24条线程并发。效率良好，并未出现错误。（模拟分布式环境下多线程操作同一资源）

项目第二阶段：8月16日到9月30日

- [ ] 与社区共同思考是否还需要优化增强的地方
- [ ] 增加新特性

- [ ] Jedis测试
- [ ] 持续优化性能，与社区一起进步完善项目

五.展望与致谢

很有幸参加这次开源之夏活动，在这次项目中，我学习了项目一开始的分布式锁设计模式。采用了工厂模式，以及责任链模式进行设计。对于该项目分布式锁功能进行增强。我期待这个项目能够实现其全部功能并保持稳定。我会不断提高该项目的质量和可扩展性。为一些应用场景提供比较完善的分布式锁解决方案。我深刻感谢项目社区导师郑导师，我数次发邮件提问，都得到了非常耐心且有效的回答，通过项目的代码也从中学习到许多设计模式与技巧。最后，我想再次感谢导师郑导师和开源之夏项目组的辛勤付出，让我有机会参与这个优秀的开源项目。我将继续为该项目贡献自己的力量，也期待在未来的学习和实践中，能够更好地理解开源文化的价值和意义，为推动开源生态系统的发展贡献自己的一份力量。