author：编程界的小学生

date：2021/03/08

零、分布式锁的必要条件

- 互斥性
- 防止死锁
- 高性能
- 可重入

一、MySQL方案

MySQL分布式锁利用唯一key插入冲突报错来解决，报错后就休息一段时间继续重试，或者CAS一直重试几次。

不符合分布式锁的必要条件的两点

- 高性能
- 可重入

需要注意如下两点

- 需要创建个表，来存放锁。
- 并发小能用，并发大别用，MySQL容易卡死，效率低。
- 如果解锁失败的话，写个定时器每隔几秒清理下MySQL的失败记录。

具体实现egg

- 表SQL

```
DROP TABLE IF EXISTS `tbl_order`;
CREATE TABLE `tbl_order` (
`order_id` int(8) NOT NULL,
`order_status` int(8) DEFAULT NULL,
PRIMARY KEY (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of tbl_order
-- ----------------------------
INSERT INTO `tbl_order` VALUES ('1', '1');
```

```
-- ----------------------------
-- Table structure for tbl_order_lock
-- ----------------------------
DROP TABLE IF EXISTS `tbl_order_lock`;
CREATE TABLE `tbl_order_lock` (
`order_id` int(8) NOT NULL,
`driver_id` int(8) DEFAULT NULL,
PRIMARY KEY (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

- 大致代码

```
@Autowired
private MysqlLock lock;

// lock
lock.lock();
// 执行业务
try {
    ...
}
finally {
 // 释放锁
 lock.unlock();
}
```

```
@Service
public class MysqlLock implements Lock {
    @Autowired
    private TblOrderLockDao mapper;

    private ThreadLocal<TblOrderLock> orderLockThreadLocal ;

    @Override
    public void lock() {
        // 1、尝试加锁
        if(tryLock()) {
            System.out.println("尝试加锁");
            return;
        }
        // 2.休眠
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 3.递归再次调用
        lock();
    }

    /**
     *  非阻塞式加锁，成功，就成功，失败就失败。直接返回
     */
    @Override
    public boolean tryLock() {
```

```
        try {
            TblOrderLock tblOrderLock = orderLockThreadLocal.get();
            mapper.insertSelective(tblOrderLock);
            System.out.println("加锁对象："+orderLockThreadLocal.get());
            return true;
        }catch (Exception e) {
            // 如果加锁失败，比如并发了造成订单主键冲突，那么就返回false，
            // 然后调用者一直死循环重试，直到这把锁被释放后即可上锁。
            return false;
        }


    }

    @Override
    public void unlock() {
        // 解锁

mapper.deleteByPrimaryKey(orderLockThreadLocal.get().getOrderId());
        System.out.println("解锁对象："+orderLockThreadLocal.get());
        // 记得放到finally里
        orderLockThreadLocal.remove();
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {
        // TODO Auto-generated method stub

    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws
InterruptedException {
        // TODO Auto-generated method stub
        return false;
    }


    @Override
    public Condition newCondition() {
        // TODO Auto-generated method stub
        return null;
    }
}
```

二、Redis方案

1、关键点

- 保证原子性

> 比如：setnx。避免set/setIfAbsent结合expire，如果非要这么整的话，放到lua脚本里。

- 锁要带过期时间

> 因为如果上锁成功了，还没释放呢，服务宕机了，这把锁将永驻，死锁了。

- 正确释放锁，别释放了别人加的锁

> 问题：释放锁可能释放了别人的锁，比如上锁时间是5s，程序执行了6s，但是这把锁5s就过期了，意味着其他线程在5s过后能继续给这个订单id上锁，但是可能出现你其他线程刚上锁1s后就被之前那个执行了6s的线程给释放了。
>
> 解决1：key肯定是相同的，因为同一个订单嘛，key不相同的话那就不需要分布式锁了，操作的都是不同的东西， 所以需要从value入手，解锁前先判断下这个key的value是不是自己加的，value不能是线程id，因为分布式环境线程id会重复，所以可以换成类似userid等业务主键。
>
> 解决2：可以起个守护线程续期 ， 上锁的时候就起个守护线程进行死循环续期，检查时间过了三分之一了就给他重新续期为上锁时间（比如5s）。

- 锁续期

> 可以起个守护线程续期 ， 上锁的时候就起个守护线程进行死循环续期，检查时间过了三分之一了就给他重新续期为上锁时间（比如5s）。

2、错误示范一

不加过期时间

```
boolean lockStatus = stringRedisTemplate.opsForValue().setIfAbsent(orderId, userId);
```

3、错误示范二

两个独立语句，非原子性操作

```
boolean lockStatus = stringRedisTemplate.opsForValue().setIfAbsent(orderId, userId);
stringRedisTemplate.expire(lockId, 30L, TimeUnit.SECONDS);
```

4、正确姿势

不符合分布式锁的必要条件的一点

- 可重入

采取setnx命令保证原子性且添加过期时间

```
boolean lockStatus = stringRedisTemplate.opsForValue().setIfAbsent(orderId, userId, 30L, TimeUnit.SECONDS);
```

解锁

```
// 判断订单id的锁是自己上的方可释放。
if((userId).equals(stringRedisTemplate.opsForValue().get(orderId))) {
    stringRedisTemplate.delete(orderId);
}
```

> 上面解锁是存在问题的，因为判断里的redis获取操作和del操作是非原子的，如果你设置了超时时间，那么你也的业务在超时时间内没有执行完，那么这个锁就会被释放，其他线程拿到锁---以上恰好发生在get之后，del之前，会删除其他的锁，那么是不是就脏读了呢？但是如果有续期的话就不存在此问题。但还是尽量用lua脚本，lua脚本如下：

```
// 如果get的值等于传进来的值，就给他del
if redis.call("get",KEYS[1])==ARGV[1] then
return redis.call("del",KEYS[1])
else
return 0
end
```

为了解决解锁时避免解其他人上的锁的问题可以添加续期

```
// 抢锁成功
if (RESULT_OK.equals(client.setNxPx(key, value, ttl))) {
    // 续期
    renewalTask = new RenewTask(new IRenewalHandler() {
        @Override
        public void callBack() throws LockException {
            // 刷新值
            client.expire(key, ttl <= 0 ? 10 : ttl);
        }
    }, ttl);
    // 设置为后台线程
    renewalTask.setDaemon(true);
    renewalTask.start();
}

// 续期线程的逻辑
@Override
public void run() {
    while (isRunning) {
        try {
            // 1、续租，刷新值
            call.callBack();
            LOGGER.debug("续租成功!");
            // 2、三分之一过期时间续租
            TimeUnit.SECONDS.sleep(this.ttl * 1000 / 3);
        } catch (InterruptedException e) {
            close();
        } catch (LockException e) {
            close();
        }
    }
}

public void close() {
    isRunning = false;
}
```

5、Redisson-sentinel

主要为了解决上面的单点故障问题。

符合分布式锁的必要条件，但是会有如下新问题：

锁写到Master后，还没同步到Slave呢，Master挂了。Slave选举成了Master，但是Slave里没有锁，其他线程再次能上锁了。不安全。

内部采取的redis的hash数据结构来完成的锁重入功能，`hset key，filed，value`，value从1开始，key，field一样的话就是锁重入，就将value+1。field默认是`UUID:ThreadId`

内部采取的lua脚本来保证的原子性。

内部自带看门狗续期过期时间。

6、RedLock

RedLock和上面【Redisson】的实现方式大同小异，只是为了解决Redisson哨兵模式下产生的问题。Redisson工具包下有RedLock的具体实现。

原理如下

用Redis中的多个master实例，来获取锁，只有大多数实例获取到了锁，**也就是我不要Slave了，弄多个独立彼此不相关的Master来完成**，才算是获取成功。具体的红锁算法分为以下五步：

- 获取当前的时间（毫秒）
- 使用相同的key和随机数在N个Master节点上获取锁，这里获取锁的尝试时间要远远小于锁的超时时间，就是为了防止某个Master挂了后我们还在不断的获取锁，导致被阻塞的时间过长。也就是说，假设锁30秒过期，三个节点加锁花了31秒，自然是加锁失败了。
- 只有在大多数节点（一般是【(2/n)+1】）上获取到了锁，而且总的获取时间小于锁的超时时间的情况下，认为锁获取成功了。
- 如果锁获取成功了，锁的超时时间就是最初的锁超时时间减获取锁的总耗时时间。
- 如果锁获取失败了，不管是因为获取成功的节点的数目没有过半，还是因为获取锁的耗时超过了锁的释放时间，都会将已经设置了key的master上的key删除。

核心源码

- 入口方法：`org.redisson.RedissonMultiLock#lock()`
- 核心方法：`org.redisson.RedissonMultiLock#tryLock(long waitTime, long leaseTime, java.util.concurrent.TimeUnit unit)`
- 允许失败的节点个数 *(N-(N/2+1))* ：`org.redisson.RedissonRedLock#failedLocksLimit()`
- lua脚本：`org.redisson.RedissonLock#tryLockInnerAsync()`
- 看门狗续期：`org.redisson.RedissonLock#renewExpirationAsync()`

需要注意两点：

- Redis多个Master所在的机器时间必须同步。
- Redis红锁机器挂了的话要延迟启动1min（大于锁超时时间就行），因为：如果三台Master，写入2台成功了，加锁成功，但是挂了一个，还保留了一个Master可用，释放锁的时候自然挂了的那个不会执行del，当他瞬间再次启动的时候会发现锁还在（因为还没到过期时间），可能造成未知的问题。所以让Redis延迟启动。

但是也有大佬提了两个问题推翻RedLock的绝对安全性。感兴趣的可以搜搜看看，很好玩的。

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

推荐博文：https://www.cnblogs.com/rgcLOVEyaya/p/RGC_LOVE_YAYA_1003days.html

三、Zookeeper方案

四、ETCD方案