# 从0搞定redis分布式锁

## 本章内容所需前置内容了解：

了解java锁的概念

了解redis 基础命令

## 分布式锁一般有三种实现方式：

1. 基于数据库实现分布式锁；2. 基于Redis的分布式锁；3. 基于ZooKeeper的分布式锁。

## 什么是分布式锁?

线程锁：主要用来给方法、代码块加锁。当某个方法或代码使用锁，在同一时刻仅有一个线程执行该方法或该代码段。线程锁只在同一JVM中有效果，因为线程锁的实现在根本上是依靠线程之间共享内存实现的，比如synchronized是共享对象头，显示锁Lock是共享某个变量（state）。

分布式锁：当多个进程不在同一个系统中，用分布式锁控制多个进程对资源的访问。

## 分布式锁的使用场景

线程间并发问题和进程间并发问题都是可以通过分布式锁解决的，但是强烈不建议这样做！因为采用分布式锁解决这些小问题是非常消耗资源的！分布式锁应该用来解决分布式情况下的多进程并发问题才是最合适的。

有这样一个情境，线程A和线程B都共享某个变量X。

如果是单机情况下（单JVM），线程之间共享内存，只要使用线程锁就可以解决并发问题。

如果是分布式情况下（多JVM），线程A和线程B很可能不是在同一JVM中，这样线程锁就无法起到作用了，这时候就要用到分布式锁来解决。

## 分布式锁的实现（Redis）

setnx() set if not exits 设置成功会返回一个1 设置失败返回一个0

分布式锁实现的关键是在分布式的应用服务器外，搭建一个存储服务器，存储锁信息，这时候我们很容易就想到了Redis。首先我们要搭建一个Redis服务器，用Redis服务器来存储锁信息。

在实现的时候要注意的几个关键点：

1、锁信息必须是会过期超时的，不能让一个线程长期占有一个锁而导致死锁；

2、同一时刻只能有一个线程获取到锁。

几个要用到的redis命令：

setnx(key, value)：“set if not exits”，若该key-value不存在，则成功加入缓存并且返回1，否则返回0。

get(key)：获得key对应的value值，若不存在则返回nil。

getset(key, value)：先获取key对应的value值，若不存在则返回nil，然后将旧的value更新为新的value。

expire(key, seconds)：设置key-value的有效期为seconds秒。

# 问题1：Redis宕机，锁无法释放

**加锁代码**

```
public  boolean tryLock(String id){
    Jedis jedis = jedisPool.getResource();
    Long start = System.currentTimeMillis();
    try{
        for(;;){
            Long setnx = jedis.setnx(lock_key, id);
            if(setnx==1){
                System.out.println("获得锁");
                return true;
            }
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }finally {
        jedis.close();
    }
}
```

**解锁代码**

```
public boolean unlock(String id){
    Jedis resource = jedisPool.getResource();
    resource.del(lock_key);
    resource.close();
    return true;
}
```

**方案1中存在的问题：**

加锁中:如果服务出现宕机，锁无法释放，导致其他服务无法获取锁，最终出现死锁。锁不具备拥有者标识，即任何客户端都可以解锁。

解锁中: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。

加锁思路：引入锁的生命时长。设置ttl

解锁思路：为锁设置一个value，先获取value 再判断value 是否相等，如果相等，进行锁删除

# 问题2：锁不具备拥有者标识，即任何客户端都可以解锁。

## 加锁代码

比较常见的错误示例就是使用jedis.setnx()和jedis.expire()组合实现加锁，代码如下：

```
Long setnx = jedis.setnx(lock_key, id);
    if(setnx==1){
        //如果获得锁成功，设置锁的生命时长
        jedis.expire(lock_key,new Long(internalLockLeaseTime).intValue());
        System.out.println("获得锁");
    return true;
}
```

setnx()方法作用就是SET IF NOT EXIST，expire()方法就是给锁加一个过期时间。乍一看好像和前面的set()方法结果一样，然而由于这是两条Redis命令，不具有原子性，如果程序在执行完setnx()之后突然崩溃，导致锁没有设置过期时间。那么将会发生死锁。网上之所以有人这样实现，是因为低版本的jedis并不支持多参数的set()方法。

**解锁代码**

```
public boolean unlock(String id){
    //redis 分布式锁，引入lua   具有原子性的
    Jedis resource = jedisPool.getResource();
    String lockValue = resource.get(lock_key);
    if(lockValue.equals(id) ){
        System.out.println("释放"+id+"锁");
        resource.del(lock_key);
    }
    resource.close();
    return true;
}
```

**方案2中存在的问题：**

加锁中:业务的执行时长大于锁的时长。那么业务没执行完，锁就进行释放了。因为nx 和 ex 代码不具备原子性，所以可能导致锁的过期时间不止原有的时长。

解锁中:这个代码的问题在于如果调用jedis.del()方法的时候，这把锁已经不属于当前客户端的时候会解除他人加的锁。那么是否真的有这种场景？答案是肯定的，比如客户端A加锁，一段时间之后客户端A解锁，在执行jedis.del()之前，锁突然过期了，此时客户端B尝试加锁成功，然后客户端A再执行del()方法，则将客户端B的锁给解除了。

加锁思路：保证获取锁和设置锁时长的原子性 （可以基于jedis3.0 后的新特性 ）

```
//SET命令的参数  Jedis3.0 引入的一个对象
SetParams params = SetParams.setParams().nx().px(internalLockLeaseTime);
jedis.set(lock_key, id, params);   //同时执行nx 和 ex
```

解锁思路：保证获取锁和删除锁的操作原子性，可以借助lua脚本执行

```
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(lock_key),
                    Collections.singletonList(id));
```

# 问题3：原子性操作

## Redis Eval 命令

 Redis 脚本

Redis Eval 命令使用 Lua 解释器执行脚本。

### 语法

redis Eval 命令基本语法如下：

```
redis 127.0.0.1:6379> EVAL script numkeys key [key ...] arg [arg ...]
```

参数说明：

- **script**： 参数是一段 Lua 5.1 脚本程序。脚本不必(也不应该)定义为一个 Lua 函数。
- **numkeys**： 用于指定键名参数的个数。
- **key [key ...]**： 从 EVAL 的第三个参数开始算起，表示在脚本中所用到的那些 Redis 键(key)，这些键名参数可以在 Lua 中通过全局变量 KEYS 数组，用 1 为基址的形式访问( KEYS[1]， KEYS[2]，以此类推)。
- **arg [arg ...]**： 附加参数，在 Lua 中通过全局变量 ARGV 数组访问，访问的形式和 KEYS 变量类似( ARGV[1]、 ARGV[2]，诸如此类)。

### 错误示例2

这一种错误示例就比较难以发现问题，而且实现也比较复杂。实现思路：使用jedis.setnx()命令实现加锁，其中key是锁，value是锁的过期时间。执行过程：1. 通过setnx()方法尝试加锁，如果当前锁不存在，返回加锁成功。2. 如果锁已经存在则获取锁的过期时间，和当前时间比较，如果锁已经过期，则设置新的过期时间，返回加锁成功。代码如下：

```
public static boolean wrongGetLock2(Jedis jedis, String lockKey, int expireTime) {
    long expires = System.currentTimeMillis() + expireTime;
    String expiresStr = String.valueOf(expires);
    // 如果当前锁不存在，返回加锁成功
    if (jedis.setnx(lockKey, expiresStr) == 1) {
        return true;
    }
    // 如果锁存在，获取锁的过期时间
    String currentValueStr = jedis.get(lockKey);
    if (currentValueStr != null && Long.parseLong(currentValueStr) < System.currentTimeMillis()) {
        // 锁已过期，获取上一个锁的过期时间，并设置现在锁的过期时间
        String oldValueStr = jedis.getSet(lockKey, expiresStr);
        if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {
            // 考虑多线程并发的情况，只有一个线程的设置值和当前值相同，它才有权利加锁
            return true;
        }
    }
    // 其他情况，一律返回加锁失败
    return false;
}
```

这段代码的错误之处在于：

\1. 由于是客户端自己生成过期时间，所以需要强制要求分布式下每个客户端的时间必须同步。

\2. 当锁过期的时候，如果多个客户端同时执行jedis.getSet()方法，那么虽然最终只有一个客户端可以加锁，但是这个客户端的锁的过期时间可能被其他客户端覆盖。

\3. 锁不具备拥有者标识，即任何客户端都可以解锁。

## 解锁

**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);
    }
}
```

如代码注释，这个代码的问题在于如果调用jedis.del()方法的时候，这把锁已经不属于当前客户端的时候会解除他人加的锁。那么是否真的有这种场景？答案是肯定的，比如客户端A加锁，一段时间之后客户端A解锁，在执行jedis.del()之前，锁突然过期了，此时客户端B尝试加锁成功，然后客户端A再执行del()方法，则将客户端B的锁给解除了。

# 问题4：业务的执行时长大于锁的生命时长

解决方案：锁延期

# Redission 实现分布式锁

```
public class Demo5Ticket {
    public static void main(String[] args) {
        Ticket ticket = new Ticket();
        Thread t1 = new Thread(ticket, "窗口1");
        Thread t2 = new Thread(ticket, "窗口2");
        Thread t3 = new Thread(ticket, "窗口3");
        t1.start();
        t2.start();
        t3.start();
    }
    static class Ticket implements Runnable {
        private int ticket = 10;
        private static RedissonClient redisson = RedissonManager.getRedisson();
        public void run() {
            String name = Thread.currentThread().getName();
            while (true) {
                //获得锁
                RLock lock = redisson.getLock("aa");
                boolean b = lock.tryLock();
                if (b) {
                    try {
```

```
                        if (ticket > 0) {
                            System.out.println(name + "卖票：" + ticket);
                            try {
                                Thread.sleep(200); //大于锁的时长
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                            ticket--;
                        }
                        if (ticket <= 0) {
                            break;
                        }
                    } finally {
                        //释放锁
                        lock.unlock();
                    }
                }
            }
        }
    }
}
```

# Redis分布式锁防止缓存击穿

缓存击穿是指：缓存中没有，但是数据库中存在的热点数据。

例如：首页的热点新闻，并发访问量非常大的热点数据，如果缓存过期失效，服务器会去查询DB，这时候如果大量的并发去查询DB，可能会瞬间压垮DB。

基础业务代码:

```
@RequestMapping("/test/{id}")
    public Account queryById(@PathVariable("id") int id){
        Account account = (Account) redisTemplate.boundValueOps(id).get();

        if(account !=null ){
            System.out.println("缓存中获取");
            return account;
        }
        System.out.println("mysql中获取");
        account = accountService.findById(id);
        if(account!=null){
            redisTemplate.boundValueOps(id).set(account);
            return account;
        }
        return account;

    }
```

突发流量测试:

```
public class Demo {
    private static CountDownLatch countDownLatch = new CountDownLatch(10);
```

```
    public static void main(String[] args) throws InterruptedException {
        TestJC jc = new TestJC();
        for(int i = 0;i<10;i++){
            Thread thread = new Thread(jc,"窗口"+i);
            thread.start();
            countDownLatch.countDown();
        }
        Thread.currentThread().join();
    }
    public static class TestJC implements Runnable{
        @Override
        public void run() {
            try {
                countDownLatch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            RestTemplate restTemplate =new RestTemplate();
            String s = restTemplate.getForObject("http://localhost:8080/test/1",
String.class);
            System.out.println(s);
        }
    }
}
```

使用redis 锁