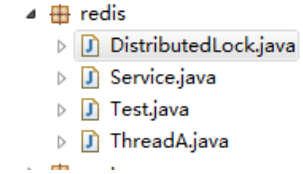
这道题包含了事务的处理，然后还有分布式锁的含义，多看看




```java
package com.redis;

import java.util.List;

public class DistributedLock {

    private final JedisPool jedisPool;

    public DistributedLock(JedisPool jedisPool) {
        super();
        this.jedisPool = jedisPool;
    }


    /**
     * 加锁
     * @param lockName  锁的key
     * @param acquireTimeout  获取超时时间
     * @param timeout         锁的超时时间
     * @return 锁标识
     */
    public String lockWithTimeout(String lockName, long acquireTimeout, long timeout) {
         Jedis conn = null;
         String retIdentifier = null;

         try {
            //获取连接
             conn = jedisPool.getResource();
             //随机生成一个value，这个是全局唯一ID
             String identifier = UUID.randomUUID().toString();
             //锁名，即key值
             String lockKey = "lock:" + lockName;
             //超过时间，上锁后超过此时间则放弃获取锁
             int lockExpire = (int)(timeout / 1000);
             //获取锁的超时时间，超过这个时间则放弃获取锁
             long end = System.currentTimeMillis() + acquireTimeout;
             while (System.currentTimeMillis() < end) {
                 //setnx是去获取锁，如果拿到了就是1，拿不到就是0
                 if(conn.setnx(lockKey, identifier) == 1) {
                     conn.expire(lockKey, lockExpire);
                     retIdentifier = identifier;
                     return retIdentifier;
                 }
                 //返回-1代表key没有设置超时时间，为key设置一个超时时间
                 if(conn.ttl(lockKey) == -1) {
                     conn.expire(lockKey, lockExpire);
                 }

                 try {
                    Thread.sleep(10);
                 } catch (Exception e) {
                    Thread.currentThread().interrupt();
                 }
             }
         } catch (Exception e) {
             // TODO: handle exception
         }finally {
             if(conn != null) {
```

```
                }
            }
        } catch (Exception e) {
            // TODO: handle exception
        }finally {
            if(conn != null) {
                conn.close();
            }
        }
         return retIdentifier;
    }

    public boolean releaseLock(String lockName, String identifier) {
        Jedis conn = null;
        String lockKey = "lock" + lockName;
        boolean retFlag = false;
        try {
            conn = jedisPool.getResource();
            while (true) {
                //监视lock，准备开始事务，当某个事务需要按条件执行时，就要使用这个命令将给定的键设置为受监控的
                conn.watch(lockKey);
                //通过前面返回的value值判断是不是该锁，若是该锁，则删除，释放锁
                if(conn.exists(lockKey) && identifier.equals(conn.get(lockKey))) {
                    //用于标记事务块的开始，redis会将后续的命令逐个放入队列中，然后才能使用exec命令原子化地执行这个命令序列
                    Transaction transaction = conn.multi();
                    transaction.del(lockKey);
                    //在一个事务中执行所有先前放入队列的命令，然后恢复到正常的连接状态
                    List<Object> results = transaction.exec();
                    if(results == null) {
                        continue;
                    }
                    retFlag = true;
                }
                //清除所有先前为一个事务监控的键
                conn.unwatch();
                break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }finally {
            if(conn != null)
                conn.close();
        }
        return retFlag;
    }

}
```

```
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;

/*
 * 这里使用50个线程模拟秒杀一个商品，使用-运算符来实现商品减少，从结果有序性就可以看出是否为加锁状态
 */
public class Service {

    private static JedisPool pool = null;

    private DistributedLock lock = new DistributedLock(pool);

    int n = 500;

    static {

        JedisPoolConfig config = new JedisPoolConfig();
        //设置最大连接数
        config.setMaxTotal(200);
        //设置最大空闲数
        config.setMaxIdle(8);
        //设置最大等待时间
        config.setMaxWaitMillis(1000 * 1000);
        //在borrow一个jedis实例时，是否需要验证，若为true，则所有jedis实例均是可用的
        config.setTestOnBorrow(true);
        pool = new JedisPool(config, "127.0.0.1", 6379, 3000);
    }

    public void seckill() {
        //返回锁的value值，供释放锁时候进行判断。
        String identifier = lock.lockWithTimeout("resource", 5000, 1000);
        System.out.println(Thread.currentThread().getName() + "获得了锁");
        System.out.println(--n);
        lock.releaseLock("resource", identifier);
    }

}
```

```java
package com.redis;

public class ThreadA extends Thread{

    private Service service;

    public ThreadA(Service service) {
        this.service = service;
    }

    public void run() {
        service.seckill();
    }
}
```

```java
package com.redis;

public class Test {

    public static void main(String [] args) {
        Service service = new Service();
        for(int i = 0; i < 50; i ++) {
            ThreadA threadA = new ThreadA(service);
            threadA.start();
        }
    }

}
```

在Redis2.6.12版本之前，使用setnx命令设置key-value、使用expire命令设置key的过期时间获取分布式锁，使用del命令释放分布式锁，但是这种实现有如下一些问题：

1. setnx命令设置完key-value后，还没来得及使用expire命令设置过期时间，当前线程挂掉了，会导致当前线程设置的key一直有效，后续线程无法正常通过setnx获取锁，造成死锁；
2. 在分布式环境下，线程A通过这种实现方式获取到了锁，但是在获取到锁之后，执行被阻塞了，导致该锁失效，此时线程B获取到该锁，之后线程A恢复执行，执行完成后释放该锁，直接使用del命令，将会把线程B的锁也释放掉，而此时线程B还没执行完，将会导致不可预知的问题；
3. 为了实现高可用，将会选择主从复制机制，但是主从复制机制是异步的，会出现数据不同步的问题，可能导致多个机器的多个线程获取到同一个锁。

针对上面这些问题，有如下一些解决方案：

1. 第一个问题是因为两个命令是分开执行并且不具备原子特性，如果能将这两个命令合二为一就可以解决问题了。在Redis2.6.12版本中实现了这个功能，Redis为set命令增加了一系列选项，可以通过SET resource_name my_random_value NX PX max-lock-time来获取分布式锁，这个命令仅在不存在key(resource_name)的时候才能被执行成功（NX选项），并且这个key有一个max-lock-time秒的自动失效时间（PX属性）。这个key的值是“my_random_value”，它是一个随机值，这个值在所有的机器中必须是唯一的，用于安全释放锁。
2. 为了解决第二个问题，用到了“my_random_value”，释放锁的时候，只有key存在并且存储的“my_random_value”值和指定的值一样才执行del命令，此过程可以通过以下Lua脚本实现。

其中Lua脚本在本实例中就是：

```
//监视lock，准备开始事务，当某个事务需要按条件执行时，就要使用这个命令将给定的键设置为受监控的
conn.watch(lockKey);
//通过前面返回的value值判断是不是该锁，若是该锁，则删除，释放锁
if(conn.exists(lockKey) && identifier.equals(conn.get(lockKey))) {
    //用于标记事务块的开始，Redis会将后续的命令逐个放入队列中，然后才能使用exec命令原子化地执行这个命令序列
    Transaction transaction = conn.multi();
    transaction.del(lockKey);
    //在一个事务中执行所有先前放入队列的命令，然后恢复到正常的连接状态
    List<Object> results = transaction.exec();
    if(results == null) {
        continue;
    }
    retFlag = true;
}
```

3. 第三个问题是因为采用了主从复制导致的，解决方案是不采用主从复制，使用RedLock算法，这里引用网上一段关于RedLock算法的描述。

> 在Redis的分布式环境中，假设有5个Redis master，这些节点完全互相独立，不存在主从复制或者其他集群协调机制。为了取到锁，客户端应该执行以下操作:
>
> - 获取当前Unix时间，以毫秒为单位;
> - 依次尝试从N个实例，使用相同的key和随机值获取锁。在步骤2，当向Redis设置锁时,客户端应该设置一个网络连接和响应超时时间，这个超时时间应该小于锁的失效时间。例如你的锁自动失效时间为10秒，则超时时间应该在5-50毫秒之间。这样可以避免服务器端Redis已经挂掉的情况下，客户端还在死死地等待响应结果。如果服务器端没有在规定时间内响应，客户端应该尽快尝试另外一个Redis实例;
> - 客户端使用当前时间减去开始获取锁时间（步骤1记录的时间）就得到获取锁使用的时间。当且仅当从大多数（这里是3个节点）的Redis节点都取到锁，并且使用的时间小于锁失效时间时，锁才算获取成功;
> - 如果取到了锁，key的真正有效时间等于有效时间减去获取锁所使用的时间（步骤3计算的结果）;
> - 如果因为某些原因，获取锁失败（没有在至少N/2+1个Redis实例取到锁或者取锁时间已经超过了有效时间），客户端应该在所有的Redis实例上进行解锁（即便某些Redis实例根本就没有加锁成功）。

通过上面的解决方案可以实现一个高效、高可用的分布式锁，这里推荐一个成熟、开源的分布式锁实现，即Redisson。