**redis命令解释**

说道Redis的分布式锁都是通过**setNx命令结合getset**来实现的，在讲之前我们先了解下setNx和getset的意思，在redis官网是这样解释的   
注：redis的命令都是原子操作

**SETNX key value(加入没有则设置，有则不设置了)**

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。   
SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写。   
**可用版本：**   
1.0.0+   
**时间复杂度：**   
O(1)   
**返回值：**   
设置成功，返回 1 。   
设置失败，返回 0 。

redis> EXISTS job # job 不存在

(integer) 0

redis> SETNX job "programmer" # job 设置成功

(integer) 1

redis> SETNX job "code-farmer" # 尝试覆盖 job ，失败

(integer) 0

redis> GET job # 没有被覆盖

"programmer"

**GETSET key value（获取旧值再设置新值）**

将给定 key 的值设为 value ，并返回 key 的旧值(old value)。   
当 key 存在但不是字符串类型时，返回一个错误。   
**可用版本：**   
1.0.0+   
**时间复杂度：**   
O(1)   
**返回值：**   
返回给定 key 的旧值。   
当 key 没有旧值时，也即是， key 不存在时，返回 nil 。

redis> GETSET db mongodb # 没有旧值，返回 nil

(nil)

redis> GET db

"mongodb"

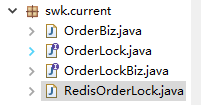
redis> GETSET db redis # 返回旧值 mongodb

"mongodb"

redis> GET db

"redis"

**代码示例**

注意：为了让分布式锁的算法更稳键些，持有锁的客户端在解锁之前应该再检查一次自己的锁是否已经超时，再去做DEL操作，因为可能客户端因为某个耗时的操作而挂起，操作完的时候锁因为超时已经被别人获得，这时就不必解锁了。   
我们看下代码涉及以下几个类，这里有关业务逻辑相关的只定义了方法没有具体实现，**关键是学习思路 ，还有里面的红色字体的备注。**  
   
OrderBiz.java

/\*\*

\* 使用redis锁来控制并发抢单

\* @author fuyuwei

\*/

public class OrderBiz {

public int createOrder(){

// 下单之前的参数、合法性校验这里就不在演示

OrderLock<Boolean> orderLock = new RedisOrderLock<Boolean>("pro-12345678901");

boolean isSyn = orderLock.isSyn(new OrderLockBiz<Boolean>(){

@Override

public Boolean createOrder() {

// 省去创建订单逻辑

return null;

}

});

if(!isSyn){

BizLogger.info("创建订单失败");

}

return 0;

}

｝

OrderLock.java

public interface OrderLock<T> {

public boolean isSyn(OrderLockBiz<T> orderBiz);

}

OrderLockBiz.java

public interface OrderLockBiz<T> {

public T createOrder();

}

RedisOrderLock.java

public class RedisOrderLock<T> implements OrderLock<T> {

// 锁等待超时，防止线程饥饿，永远没有入锁执行代码的机会

public static final long timeout = 10000;//ms

// 锁持有超时，防止线程在入锁以后，无限的执行下去，让锁无法释放

public static final long expireMsecs = 10000;// ms

public String lockKey = "orderLockKey";

public Jedis jedis;

private static volatile JedisPool jedisPool;

public RedisOrderLock(String lockKey) {

this.lockKey = lockKey;

}

/\*\*

\* 初始化redis

\* @return

\*/

public Jedis getInstance() {

if(jedisPool == null) {

synchronized(RedisOrderLock.class) {

if(jedisPool == null) {

JedisPoolConfig config = new JedisPoolConfig();

config.setMaxIdle(100);

jedisPool = new JedisPool(config,"localhost",6379, 3000,"test");

}

}

}

return jedisPool.getResource();

}

/\*\*

\* 线程安全的业务逻辑处理

\*/

@Override

public boolean isSyn(OrderLockBiz<T> orderBiz) {

jedis = this.getInstance();

try {

// 获取到锁

if(acquire(jedis)){

// 执行创建订单逻辑

orderBiz.createOrder();

}else{

BizLogger.info("waiting other thread creating");

}

} catch (Exception e) {

BizLogger.error(e,"acquire lock failre");

}finally{

// 解锁

this.releaseLock(jedis);

}

return false;

}

/\*\*

\* accqure lock

\* @param jedis

\* @return

\* @throws InterruptedException

\*/

public synchronized boolean acquire(Jedis jedis){

boolean locked = false;

while(timeout > 0){

long expires = System.currentTimeMillis() + expireMsecs + 1;

// 10秒之后锁到期

String expiresStr = String.valueOf(expires);

**// 获取到锁**

if(jedis.setnx(lockKey, expiresStr) == 1){

locked = true;

return locked;

}

**// 没有获取到锁，获得old**

String oldValue = jedis.get(lockKey);

**// expireMsecs（10秒）锁的有效期内无法进入if判断，如果锁超时了**

if(oldValue != null

&& Long.parseLong(oldValue) < System.currentTimeMillis()){

**// 如果锁超时重新设置（则在这可以获得新锁）**

String oldValue\_ = jedis.getSet(lockKey, expiresStr);

**// 值相同说明是同一个线程的操作，获取锁成功（把redis里的锁头set为新值，但是get还是得到旧值。所以可以对比，比如A先设置，则取到旧的，设置A的，若B在A后面，则B设置了，但是去到A的值，所以可以做下面的判断。取到旧的才是取得锁头成功。）**

if(Long.valueOf(oldValue\_) == Long.valueOf(oldValue)){

locked = true;

}else{

// 被其他线程抢先获取锁

locked = false;

}

}

// 锁没有超时，继续等待

return false;

}

}

/\*\*

\* 释放锁

\* @param jedis

\*/

public synchronized void releaseLock(Jedis jedis){

try {

long current = System.currentTimeMillis();

// 避免删除非自己获取得到的锁

if (current < Long.valueOf(jedis.get(lockKey)))

jedis.del(lockKey);

} catch (Exception e) {

e.printStackTrace();

}finally{

// 把用完的连接放到连接池汇中供其他线程调用

jedisPool.returnResource(jedis);

}

}

}

但是原先会有这种问题：

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。

public class RedisTool {

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;

}

}

[复制代码](javascript:void(0);)

可以看到，我们加锁就一行代码：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（userid），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。(解决问题1，2)**2. 已有锁存在，不做任何操作。

1. 用set： key为productid，value为userid，设置过期时间。解决要求客户端时间一样和getset问题**。(解决问题1，2)**
2. 解锁要先验证userid，如果不是，则业务回滚；如果是，则解锁；即谁去解锁，要它的锁才能给他解，防止很多客户端都可以解锁，使用这种方式释放锁可以避免别的客户端获取成功的锁被删除。**(解决问题3，4)（这个有点类似CAS操作。）**

举个例子：客户端A取得资源锁，但是紧接着被一个其他操作阻塞了，当客户端A运行完毕其他操作后要释放锁时，原来的锁早已超时并且被Redis自动释放，并且在这期间资源锁又被客户端B再次获取到。如果仅使用DEL命令将key删除，那么这种情况就会把客户端B的锁给删除掉。

附（原来的3个问题）

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。

4. 关于过期时间的设置，如果时间设置得太短。有可能某线程挂起导致锁过期了没有锁，还导致线程成功操作后。

残留的未解决的问题：公平性和过期时间的设置。

* 1. 使用一个消息队列，像IO多路复用那种思想可以解决公平性；
  2. 可以使用zookeeper解决；在底层，给每个线程分节点ID，从小排序到大。

过期时间长度，需要怎么权衡，暂时还没有想到。