什么是线程安全？在什么场景下需要用到 synchronized，lock，redis 分布式锁？

作者：gongxiongzhuang
原文链接：https://ld246.com/article/1559281199926
来源网站：链滴

一、什么是线程安全

百度百科上的解释：多个线程访问同一个对象时，如果不用考虑这些线程在运行时环境下的调度和交执行，也不需要进行额外的同步，或者在调用方进行任何其他操作，调用这个对象的行为都可以获得确的结果，那么这个对象就是线程安全的。

推荐一篇文章讲的通俗易懂https://www.cnblogs.com/lixinjie/p/10817860.html

二、什么场景会存在线程不安全

线程安全问题都是由全局变量及静态变量引起的。

若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

举个栗子：当某商城做秒杀活动限制1000个商品抢完为止，这时可能有2000个用户同时参与抢购，果线程不做同步或者加锁，则2000个用户同时获取的库存信息都是1000个，那么2000个用户都会成抢到商品，获取的脏数据导致数据的错乱。

三、 3种方法解决并发导致线程不安全问题

1. java关键字 synchronized

java关键字 synchronized 可以作用于方法，也可以作用于代码块；

如果一个代码块或者一个方法被synchronized关键字修饰，当一个线程获取了对应的锁，并执行该代块时，其他线程便只能一直等待直至占有锁的线程释放锁；

synchronized是java关键字，是基于jvm实现的；

优点：使用简单，能够解决并发不是很高的场景

缺点：不灵活，不利于拓展

2. Lock锁

Lock是jdk里面的一个接口类，下面是Lock接口类的几个方法

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

lock()、tryLock()、tryLock(long time, TimeUnit unit) 和 lockInterruptibly()都是用来获取锁的。u Lock()方法是用来释放锁的。

- lock()

lock()获取锁，如果锁被其他线程获取，则会一直等待其释放为止，lock锁必须主动释放锁，否则会现死锁导致线程一直等待，一般lock使用try catch finally 来运行，try执行业务，finally释放锁，如

使用：

```
//实例化创建锁
Lock lock = new ReentrantLock();
lock.lock();
try{
    //执行逻辑任务
}catch(Exception ex){
   //捕获业务逻辑异常
}finally{
    //释放锁
    lock.unlock();
}
```

● tryLock()和tryLock(long time, TimeUnit unit)

这两个方法类似，都有返回值，当返回值为true则表示拿到锁，可以操作正常的业务逻辑了；如果返值为false则表示锁已经被其他线程拿到，可以做其他操作；

tryLock()：会立即返回结果告诉是否拿到锁，不会等待；(并发场景不高情况下，当返回false可以做试机制)；

tryLock(long time, TimeUnit unit)：在拿不到锁的情况下会等待一段时间，当在时间限制内为拿到则返回false，如果在时间范围类拿到锁则返回true；

```
Lock lock = new ReentrantLock();
//尝试获取锁，也可以使用lock.tryLock(20, TimeUnit.SECONDS)，等待20秒的时间
if(lock.tryLock()) {
    try{
        //执行逻辑任务
    }catch(Exception ex){

    }finally{
        //释放锁
        lock.unlock();
    }
}else {
   //如果不能获取锁，可以处理其他任务
}
```

● lockInterruptibly()

lockInterruptibly()不常用，这里暂不说明

3. redis 分布式锁

上面两种方法讲到的是保证线程安全，锁的是某个服务器得线程，但是在分布式环境下同时操作某个据上面两种同步锁的方式可能就锁不住了，因为在不同的服务器里面不能互相锁，只能锁住自己服务线程；下面说一下使用redis 实现分布式锁来实现商品库存加减，商品销量的加减；

1. spring boot 下 使用redis+lua脚本实现抢购

假设场景商品库存只有1000，但是10点有一个抢购活动，可能同时在线抢购人数超过1000，要保证100个人能够正常抢到，其他没抢到的人需要出给友好的提示。

redis是单线程处理，把判断商品库存和减库存的操作用lua脚本执行，即可保证数据的正确性。

redis 工具类

```
@Component
public class RedisLock {
    @Autowired
    private RedisTemplate<String, Object> redisTemplate;
    //lua 脚本编写判断库存，如果库存足够则减库存并更新库存
    public static final String RUSH_TO_BUY_LUA = "local stock = tonumber(redis.call('get', KEYS
1])) " +
            "if stock < tonumber(ARGV[1]) then return 0 end " +
            "stock = stock - ARGV[1] " +
            "redis.call('set', KEYS[1], tonumber(stock)) " +
            "return 1";
    /**

     * 购买商品，判断商品库存是否充足
     * @param key 商品库存key
     * @param bugNum 购买商品数量
     * @return
     */
    public boolean bugGoods(String key, String bugNum) {
        List<String> keys = new ArrayList<>();
        keys.add(key);
        List<String> args = new ArrayList<>();
        args.add(bugNum);
        // 使用lua脚本删除redis中匹配value的key，可以避免由于方法执行时间过长而redis锁自动过
失效的时候误删其他线程的锁
        // spring自带的执行脚本方法中，集群模式直接抛出不支持执行脚本的异常，所以只能拿到原re
is的connection来执行脚本
        RedisCallback<Long> callback = (connection) -> {
            Object nativeConnection = connection.getNativeConnection();
            // 集群模式和单机模式虽然执行脚本的方法一样，但是没有共同的接口，所以只能分开执行
            // 集群模式
            if (nativeConnection instanceof JedisCluster) {
                return (Long) ((JedisCluster) nativeConnection).eval(RUSH_TO_BUY_LUA, keys, args);
            }
            // 单机模式
            else if (nativeConnection instanceof Jedis) {
                return (Long) ((Jedis) nativeConnection).eval(RUSH_TO_BUY_LUA, keys, args);
            }
            return 0L;
        };
        Long result = redisTemplate.execute(callback);
        return result != null && result > 0;
    }
}
```

测试购买商品方法

```
    @ApiOperation(value = "购买商品")
    @GetMapping("/bugGoods")
    public void bugGoods() {
        ThreadPoolExecutor pool1 = (ThreadPoolExecutor) Executors.newCachedThreadPool();
        for (int i = 0; i < 2000; i++) {
```

```
            pool1.execute(()-> {
                if (redisLock.bugGoods("test", "1")) {
                    logger.info("购买成功！");
                } else {
                    System.out.println("库存不足购买失败");
                }
            });
        }
    }
```

注意上面在调用redisLock.bugGoods("test", "1")之前必须要初始化库存"test"到edis缓存里面；

2. spring boot 下使用redis + lua 实现分布式锁

```
@Component
public class RedisLock {
    @Autowired
    private RedisTemplate<String, Object> redisTemplate;
    private static final String LOCK_PREFIX = "lock:";
    //线程本地变量
    private ThreadLocal<String> localKeys = new ThreadLocal<>();//设置锁的key值
    private ThreadLocal<String> localRequestIds = new ThreadLocal<>();//设置锁的唯一value值
    private static final Long LOCK_SUCCESS = 1L;
    private static final String LOCK_LUA;
    private static final String UNLOCK_LUA;
    static {
        //加锁脚本，其中KEYS[]为外部传入参数
        //KEYS[1]表示key
        //KEYS[2]表示value
        //KEYS[3]表示过期时间
        //setnx (set if not exists) 如果不存在则设置值
        LOCK_LUA = " if redis.call('setnx', KEYS[1], KEYS[2]) == 1 " +
                "then " +
                "    if KEYS[3] == '-1' then return 1 else return redis.call('pexpire', KEYS[1], KEYS[3])
nd "+
                "else " +
                "    return 0 " +
                "end ";

        //解锁脚本
        //KEYS[1]表示key
        //KEYS[2]表示value
        //return -1 表示未能获取到key或者key的值与传入的值不相等
        UNLOCK_LUA = " if redis.call('get',KEYS[1]) == KEYS[2] " +
                "then " +
                "    return redis.call('del',KEYS[1]) " +
                "else " +
                "    return -1 " +
                "end ";
    }
    /**

     * 获取锁，如果获取不到则一直等待，最长超时5分钟
     * @param key
```

```
     * @return
     */
    public boolean lock(String key) {
        return lock(key, 5*60*100, -1);
    }

    /**
     * 加锁
     * @param key Key
     * @param timeout 过期时间 单位毫秒
     * @param retryTimes 重试次数
     * @return
     */
    public boolean lock(String key, long timeout, int retryTimes) {
        try {
            DefaultRedisScript<Long> LOCK_LUA_SCRIPT = new DefaultRedisScript<>(LOCK_LUA,
Long.class);
            final String redisKey = this.getRedisKey(key);
            final String requestId = this.getRequestId();
            logger.debug("lock :::: redisKey = " + redisKey + " requestid = " + requestId);
            //组装lua脚本参数
            List<String> keys = Arrays.asList(redisKey, requestId, String.valueOf(timeout));
            //执行脚本
            Long result = redisTemplate.execute(LOCK_LUA_SCRIPT, keys);
            //加锁成功则存储本地变量
            if (result != null && result.equals(LOCK_SUCCESS)) {
                localRequestIds.set(requestId);
                localKeys.set(redisKey);
                logger.info("success to acquire lock:" + Thread.currentThread().getName() + ", Statu
 code reply:" + result);
                return true;
            } else if (retryTimes == 0) {
                //重试次数为0直接返回失败
                return false;
            } else {
                //重试获取锁
                logger.info("retry to acquire lock:" + Thread.currentThread().getName() + ", Status c
de reply:" + result);
                int count = 0;
                while (true) {
                    try {
                        //休眠一定时间后再获取锁，这里时间可以通过外部设置
                        Thread.sleep(100);
                        result = redisTemplate.execute(LOCK_LUA_SCRIPT, keys);
                        if (result != null && result.equals(LOCK_SUCCESS)) {
                            localRequestIds.set(requestId);
                            localKeys.set(redisKey);
                            logger.info("success to acquire lock:" + Thread.currentThread().getName() +
, Status code reply:" + result);
                            return true;
                        } else {
                            count++;
                            if (retryTimes == count) {
                                logger.info("fail to acquire lock for " + Thread.currentThread().getName()
```

```
", Status code reply:" + result);
                            return false;
                        } else {
                            logger.warn(count + " times try to acquire lock for " + Thread.currentThre
d().getName() + ", Status code reply:" + result);
                        }
                    }
                } catch (Exception e) {
                    logger.error("acquire redis occured an exception:" + Thread.currentThread().ge
Name(), e);
                    break;
                }
            }
        }
    } catch (Exception e1) {
        logger.error("acquire redis occured an exception:" + Thread.currentThread().getName()
e1);
    }
    return false;
}

/**
 * 获取RedisKey
 * @param key 原始KEY，如果为空，自动生成随机KEY
 * @return
 */
private String getRedisKey(String key) {
    //如果Key为空且线程已经保存，直接用，异常保护
    if (StringUtils.isEmpty(key) && !StringUtils.isEmpty(localKeys.get())) {
        return localKeys.get();
    }
    //如果都是空那就抛出异常
    if (StringUtils.isEmpty(key) && StringUtils.isEmpty(localKeys.get())) {
        throw new RuntimeException("key is null");
    }
    return LOCK_PREFIX + key;
}

/**
 * 获取随机请求ID
 * @return
 */
private String getRequestId() {
    return UUID.randomUUID().toString();
}

/**
 * 释放锁
 * @param key
 * @return
 */
public boolean unlock(String key) {
    try {
        DefaultRedisScript<Long> UNLOCK_LUA_SCRIPT = new DefaultRedisScript<>(UNLOC
```

```
_LUA, Long.class);
            String localKey = localKeys.get();
            //如果本地线程没有KEY，说明还没加锁，不能释放
            if(StringUtils.isEmpty(localKey)) {
                logger.error("release lock occured an error: lock key not found");
                return false;
            }
            String redisKey = getRedisKey(key);
            //判断KEY是否正确，不能释放其他线程的KEY
            if(!StringUtils.isEmpty(localKey) && !localKey.equals(redisKey)) {
                logger.error("release lock occured an error: illegal key:" + key);
                return false;
            }
            //组装lua脚本参数
            List<String> keys = Arrays.asList(redisKey, localRequestIds.get());
            logger.debug("unlock :::: redisKey = " + redisKey + " requestid = " + localRequestIds.g
t());
            // 使用lua脚本删除redis中匹配value的key，可以避免由于方法执行时间过长而redis锁自动
期失效的时候误删其他线程的锁
            Long result = redisTemplate.execute(UNLOCK_LUA_SCRIPT, keys);
            //如果这里抛异常，后续锁无法释放
            if (result !=null && result == 1L) {
                logger.info("release lock success:" + Thread.currentThread().getName() + ", Status c
de reply=" + result);
                return true;
            } else if (result!=null && result == -1L) {
                //返回-1说明获取到的KEY值与requestId不一致或者KEY不存在，可能已经过期或被其他
程加锁
                // 一般发生在key的过期时间短于业务处理时间，属于正常可接受情况
                logger.warn("release lock exception:" + Thread.currentThread().getName() + ", key
as expired or released. Status code reply=" + result);
            } else {
                //其他情况，一般是删除KEY失败，返回0
                logger.error("release lock failed:" + Thread.currentThread().getName() + ", del key fa
led. Status code reply=" + result);
            }
        } catch (Exception e) {
            logger.error("release lock occured an exception", e);
        } finally {
            //清除本地变量
            this.clean();
        }
        return false;
    }

    /**
     * 清除本地线程变量，防止内存泄露
     */
    private void clean() {
        localRequestIds.remove();
        localKeys.remove();
    }
}
```

运行测试

```
@ApiOperation(value = "redis 分布式锁")
@GetMapping("/lock")
public void lock() {
    ThreadPoolExecutor pool1 = (ThreadPoolExecutor) Executors.newCachedThreadPool();
    for (int i = 0; i < 1000; i++) {//testService.incr()
        pool1.execute(()-> {
            if (redisLock.lock("1")) {
                try {
                    //成功获取锁
                    logger.info("获取锁成功，继续执行任务" + Thread.currentThread().getName());
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }catch (Exception e) {
                    logger.error("Exception ", e);
                } finally {
                    redisLock.unlock("1");
                }
            }
        });
    }
}
```

参考文章：https://www.jianshu.com/p/1145cd7e0cf1

总结：

小型项目为了快速开发使用java关键字 synchronized 就可以快速解决多线程同步问题；

如果是单应用服务器，并发也不是很高，可以使用java接口类lock锁来解决同步问题；

如果是分布式项目，则考虑使用分布式锁；