首先是锁的一个错误

在service方法上加了锁，然后在这个方法上面加了事务，结果锁不起作用

@Transactional  
public synchronized Integer changeStick(){

原因是：事务的范围比锁的范围大，事务是通过aop植入，而锁则只是在这个方法上，所以，在线程释放锁的时候事务可能还没有提交，所以别的线程会出现脏读，从而导致更新丢失

解决方案

扩大锁的范围，将synchronized放到controller上，不过不推荐

@GetMapping(value = "/stack/reduce")  
public synchronized void getPaymentById(){  
 stockService.changeStick();  
}

在上面的方法上解决了单体服务的锁的并发问题，那么在分布式环境中这样还可以有效吗？

答案是：无效，会出现更新丢失，这就是分布式锁的场景，说明一个数据库多个服务同时修改一个数据也会有分布式事务的问题。

解决分布式锁的方案，

1乐观锁和悲观锁

采用乐观锁的话可以解决分布式事务问题，但是失败率很高，一百个线程并发一次成功的只有20个左右

2 redis：下面介绍

3 zookeeper

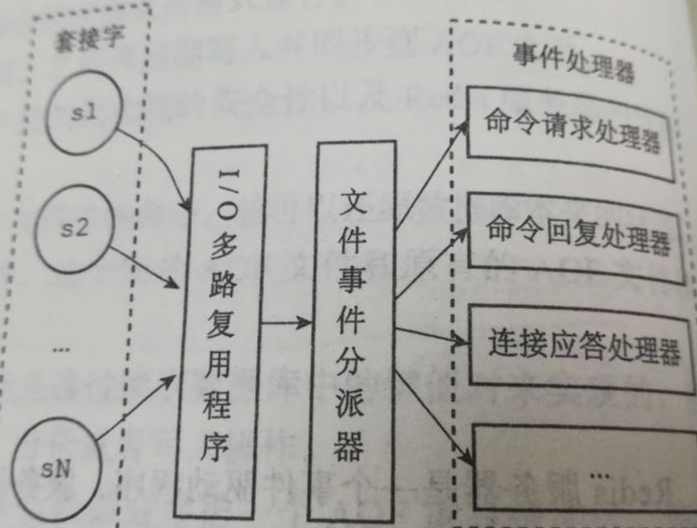
关于Redis分布式锁的笔记

**1为什么redis能实现分布式锁**

**1.1Redis事件串行处理并且单线程处理事件（任务）**

##### 文件事件

文件事件主要用于执行命令，如写入写出等。。文件事件通过套接字来进行通讯，套接字里面说明了这个事件具体是什么任务，见下图

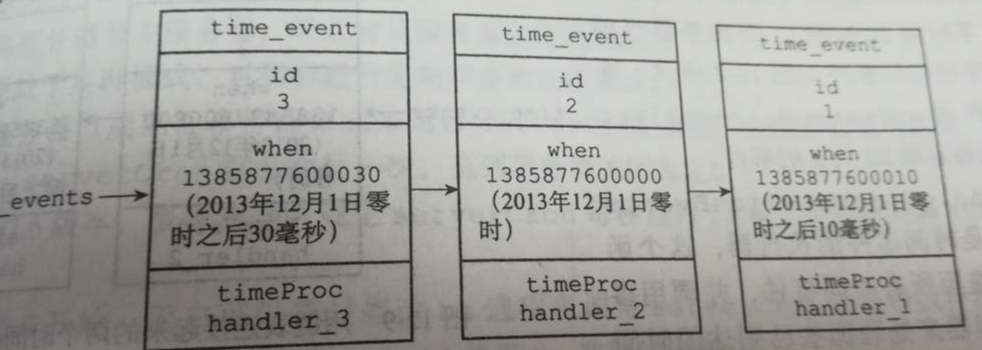


由图我们可以知道，redis的文件事件处理器由以下部分组成

1. 套接字：用于按照一定的规则封装事件的任务。
2. I/O多路复用程序：文件事件处理器使用I/O多路复用程序来监听多个套接字，并根据套接字所执行的任务为套接字 关联不同的事件处理器
3. 文件事件分派器：文件事件分派器将将所有的文件事件放在一个队列里面，然后逐个取出交给事件处理器处理，所以说redis是单线程的
4. 事件处理器：处理事件

##### 时间事件

时间事件分为定时事件和周期性事件

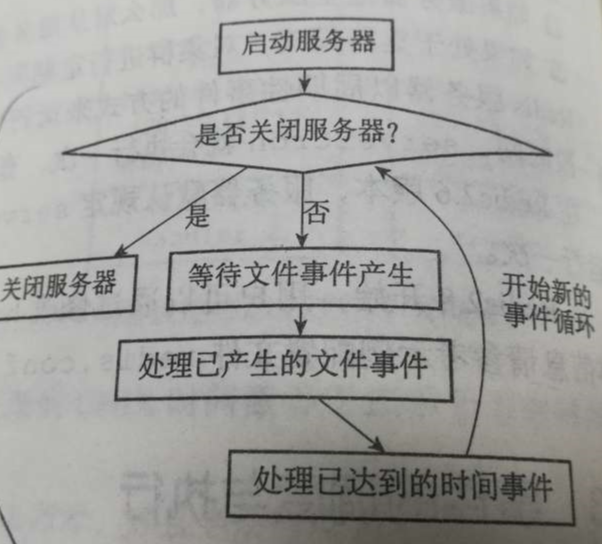


1. 定时事件：让程序在指定时间执行的定时任务
2. 周期性事件：周期性执行的事件，如过期键删除中的定期删除
3. 时间事件的实现

服务器将所有的时间事件放在一个无序的链表（不按照when来进行排序），每当时间事件执行器运行，就遍历这个队列查找已经到时间的事件然后调用相应的处理器进行处理

##### 事件调度

在文件事件和时间事件同时存在的时候，如何进行调度？



1、首先文件事件是只要有文件事件到达就马上执行

2、对于时间事件它的执行方式则是

* 2-1 获取最快要开始执行的时间事件的时间
* 2-2 阻塞等待最近的需要执行的时间事件的到来
* 2-3 时间到来，执行时间事件

3、事件调度（文件事件与时间事件共存）--》aeProcessEvent 函数,函数的原理如图（不贴代码了）见它的原理图

意思就是不断处理文件事件，当时间事件到来则开始处理时间事件，没有事件则进入阻塞状态，不断循环。但是需要注意的是，这些事件都不是抢占式的，而是一个接着一个执行，当然，如果一个事件执行的时间超越某个临界值的时候，这个事件会自动break，然后保存状态等待下次执行

4、实例

上面解释那么多肯定都不如一个实例来得清楚，那么让我们来个实例

| **开始时间** | **结束时间** | **动作** |
| --- | --- | --- |
| 0 | 10 | 创建一个在100毫秒执行的事件事件 |
| 11 | 30 | 等待文件事件 |
| 31 | 50 | 文件事件到达，处理文件事件 |
| 51 | 85 | 等待文件事件 |
| 85 | 130 | 文件事件到达，执行文件事件 |
| 130 | 150 | 处理时间事件（0毫秒创建的时间事件） |

从上面的例子可以看出文件事件到来，且处理器空闲的话就立马执行，时间事件在时间到来的时候，如果处理器空闲，则执行，处理器不空闲，需要等到当前文件事件执行完了之后才进行处理，而不进行抢占式调度。

**1.2Redis的 setNx命令**

SetNx的做法就是当存入新的key和value的时候，如果已经存在相同的key则将放弃存入这个键值对，基于这个特性则多个线程只有一个线程可以设置成功

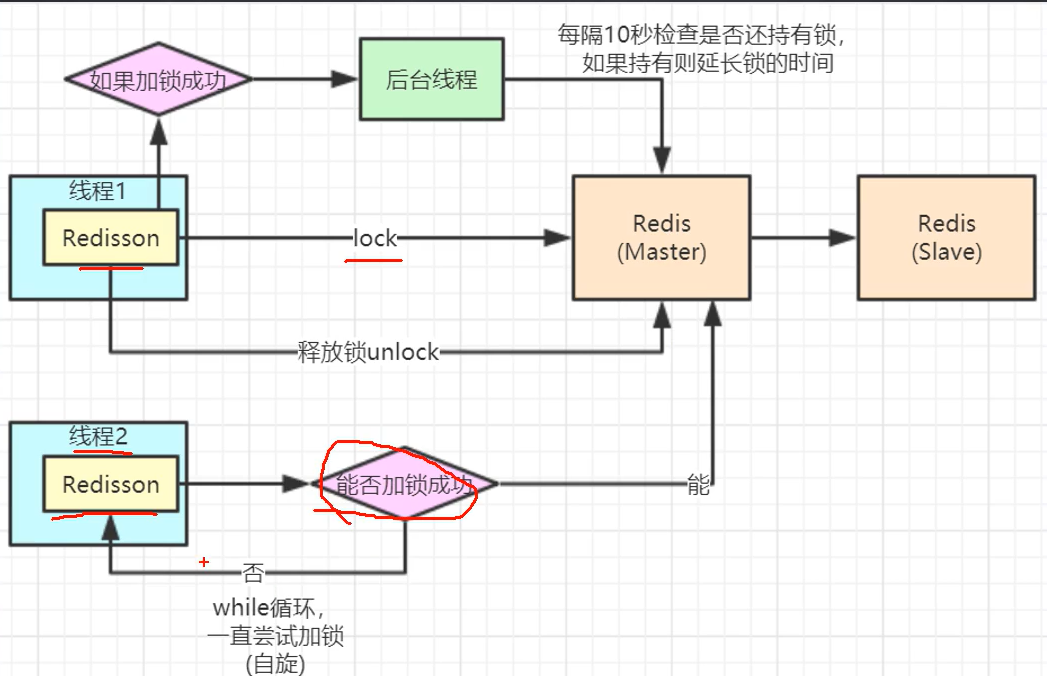
**1.3高吞吐**

Redis基于内存和单线程处理事件，拥有较高的并发量可以满足较大并发的场景

**2如何实现分布式锁**

**2.1基于Redission实现**

原理Redison用自己设定的key和线程id为value向redis执行setnx命令，如果设置成功则加锁成功，并且开启新线程每过一段时间（自己设置的ttl的1/3）进行续命，解锁的时候才用当前线程的线程id为匹配点进行解锁，保证自己加的锁自己解



**2.2源码**

**lock.lock(30, TimeUnit.SECONDS);**

**public void** lockInterruptibly(**long** leaseTime, TimeUnit unit) **throws** InterruptedException {  
 **long** threadId = Thread.currentThread().getId();  
 Long ttl = **this**.tryAcquire(leaseTime, unit, threadId);  
 **if** (ttl != **null**) {  
 RFuture<RedissonLockEntry> future = **this**.subscribe(threadId);  
 **this**.commandExecutor.syncSubscription(future);  
  
 **try** {  
 **while**(**true**) {  
 ttl = **this**.tryAcquire(leaseTime, unit, threadId);  
 **if** (ttl == **null**) {  
 **return**;  
 }  
  
 **if** (ttl >= 0L) {  
 **this**.getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);  
 } **else** {  
 **this**.getEntry(threadId).getLatch().acquire();  
 }  
 }  
 } **finally** {  
 **this**.unsubscribe(future, threadId);  
 }  
 }  
}

从上可知其实主要就是用线程id作为lockKey的value进行加锁，并且设置过期时间和续命

**unlock()方法则是用线程id解锁**  
**public void** unlock() {  
 Boolean opStatus = (Boolean)**this**.get(**this**.unlockInnerAsync(Thread.currentThread().getId()));  
 **if** (opStatus == **null**) {  
 **throw new** IllegalMonitorStateException(**"attempt to unlock lock, not locked by current thread by node id: "** + **this**.id + **" thread-id: "** + Thread.currentThread().getId());  
 } **else** {  
 **if** (opStatus) {  
 **this**.cancelExpirationRenewal();  
 }  
  
 }  
}

**简单代码实现分布式锁库存扣减**

@RestController  
@RequestMapping(**"redisLock"**)  
**public class** Redislockctrl {  
  
 @Autowired  
 **private** StringRedisTemplate **stringRedisTemplate** ;  
  
 @Autowired  
 **private** Redisson **redisson**;  
  
 @GetMapping  
 **public** String RedisLock(){  
 String lockKey = **"product01"**;

//用rdission打造一把锁  
 RLock lock = **redisson**.getLock(lockKey);  
 **try**{

//尝试加锁，原理就是看能不能在redis用setnx命令插入一条数据，key为**product01，value为线程id**

Lock不成功会阻塞，自旋等待或者超时退出  
 lock.lock(30, TimeUnit.***SECONDS***);

//开始执行业务  
 **int** stock = Integer.*parseInt*(**stringRedisTemplate**.opsForValue().get(**"stock"**));  
 **if** (stock>1){  
 stock = stock - 1;  
 **stringRedisTemplate**.opsForValue().set(**"stock"**,stock+**""**);  
 System.***out***.println(**"扣減庫存成功，剩餘庫存"**+stock);  
 }  
 **else** {  
 System.***out***.println(**"庫存不足"**);  
 }  
 **return "success"**;  
 }**finally** {

//解锁，就是删掉了key为product01的字符串  
 lock.unlock();  
 }  
  
 }  
}

**3Redisson实现分布式锁的一些优点和问题**

**优点**

简单效率高

**缺点及问题**

**Redis集群下主从切换如何保证锁不丢失**

采用redisson的redLock，redLock会对集群中的2n+1个主节点进行加锁，只有2n+1个主节点均加锁成功才算加锁成功，如三个节点的集群必须获得两个节点的锁才能成功，如果其中一个主节点宕机，因为剩下的两个节点中有一个节点里面存储了分布式锁，新的线程在进行加锁的时候无法获取2n+1个节点故而无法加锁成功，不会导致锁丢失

**redLock节点宕机情况导致共同持有锁**

如

客户端1成功锁住了A, B, C，获取锁成功（但D和E没有锁住）。

节点C崩溃重启了，但客户端1在C上加的锁没有持久化下来，丢失了。

节点C重启后，客户端2锁住了C, D, E，获取锁成功。

客户端1和客户端2同时获得了锁。

解决办法

为了应对这一问题，antirez又提出了延迟重启(delayed restarts)的概念。也就是说，一个节点崩溃后，先不立即重启它，而是等待一段时间再重启，这段时间应该大于锁的有效时间(lock validity time)。这样的话，这个节点在重启前所参与的锁都会过期，它在重启后就不会对现有的锁造成影响。

**如何提高redis性能**

采用分段存储的概念分段加锁

**4redisson的其他方法（api）**[https://github.com/redisson/redisson/wiki/8.-%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81%E5%92%8C%E5%90%8C%E6%AD%A5%E5%99%A8#81-%E5%8F%AF%E9%87%8D%E5%85%A5%E9%94%81reentrant-lock](https://github.com/redisson/redisson/wiki/8.-%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81%E5%92%8C%E5%90%8C%E6%AD%A5%E5%99%A8" \l "81-%E5%8F%AF%E9%87%8D%E5%85%A5%E9%94%81reentrant-lock)