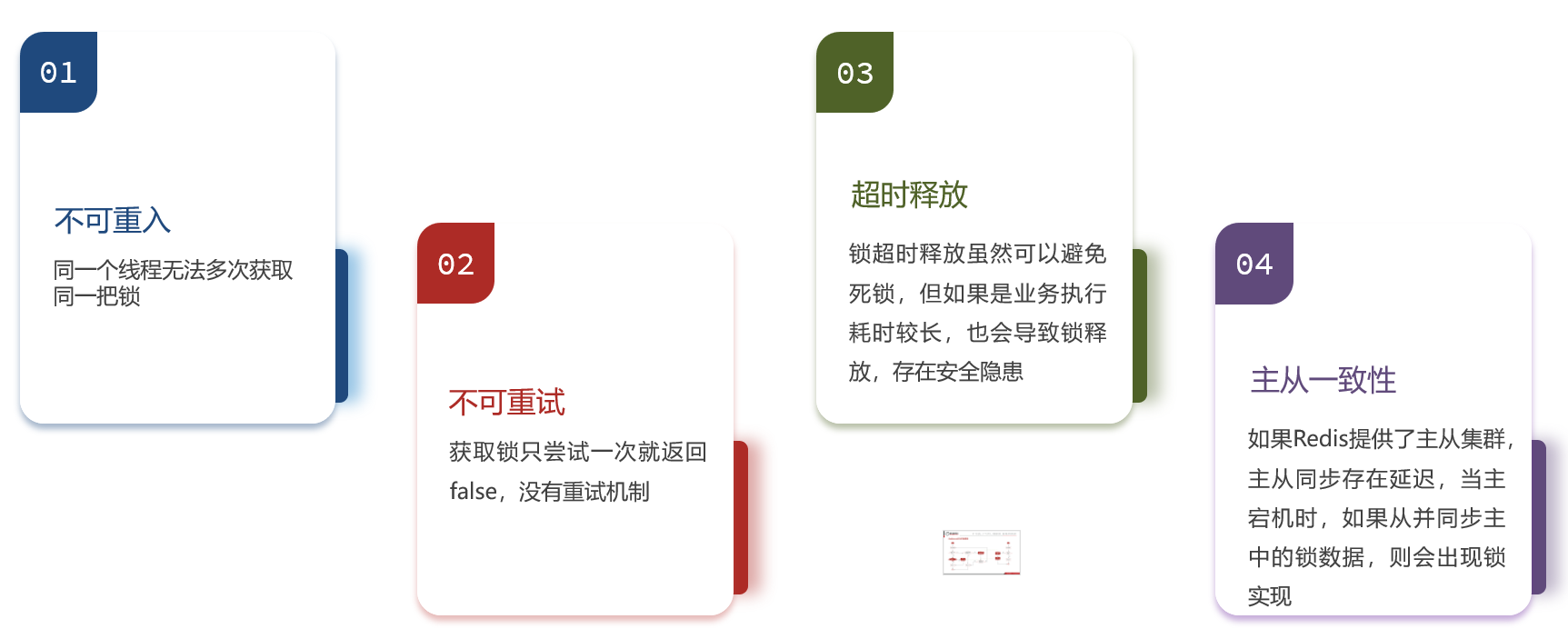
## Redisson怎么解决主从一致性问题的

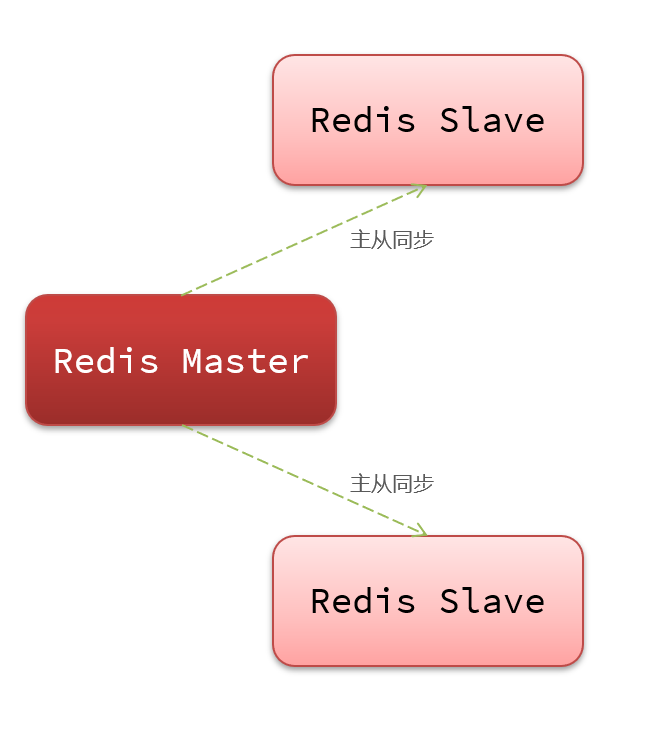
前面的学习我们已经分析了Redisson获取分布式锁如何实现锁的可重入、可重试、超时续时的原理。也就是说解决了下面的三个问题，但是现在还有最后一个问题，就是主从一致性的问题还没有得到解决，接下来我们一起分析一下Redisson如何解决主从一致性问题的！



## 主从一致性产生原因

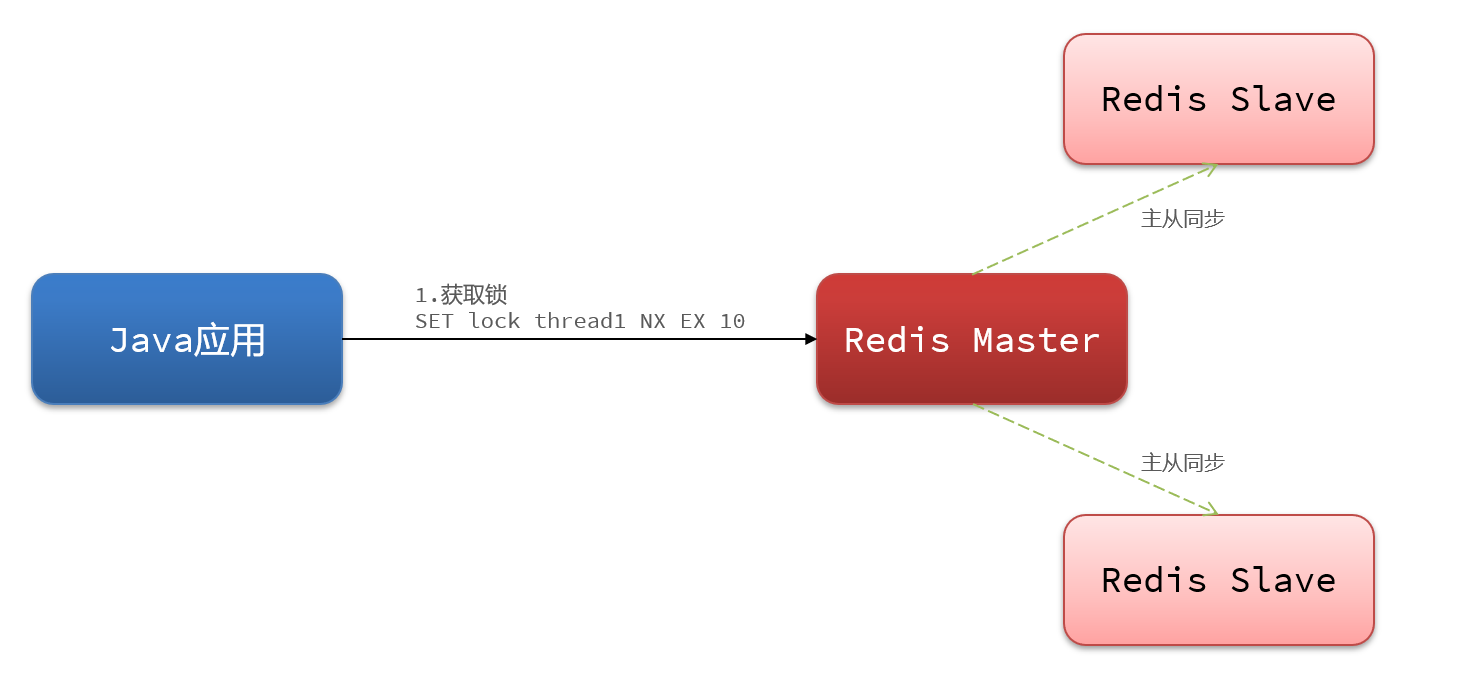
首先我们先来详细分析一下主从一致性产生的原因，在之前的案例中我们采用的都是单节点的Redis，大家不妨思考一下，如果单节点的Redis发生故障会引发什么问题！可以想象的是，我们所以依赖于Redis的业务都会发生问题，包括我们的分布式锁！显然在一些核心业务中，这样的状况肯定是不允许发生的，为了解决这个问题，提高Redisson可用性，往往在实际应用当中，我们会去搭建Redis主从模式。

什么是Redis主从呢？就是有多台Redis，只不过这些Redis角色不同，有一台Redis我们可以作为主节点，剩下的Redis作为从节点；而主从节点的职责也不一样，往往会做读写分离，也就是说在主节点里处理所有发向Reids写操作，比如说增删改；而从节点只负责Redis读操作。

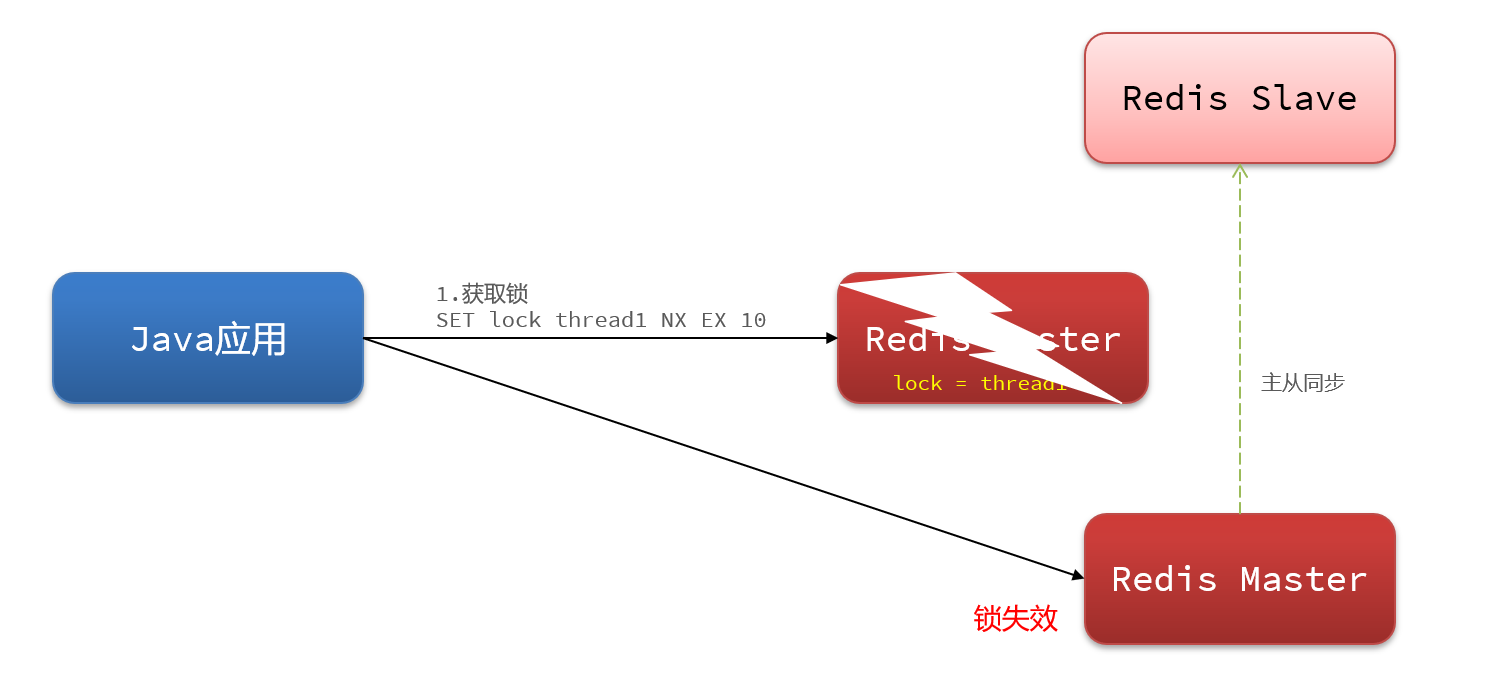


主从节点读写分离这样一来就有问题了，既然主节点处理写操作，那所有的数据都是在主节点上存在的，从节点没有数据怎么来处理Redis读请求呢？所以啊，主节点和从节点之间需要做数据的同步，主节点会不断地把自己的数据同步给从节点，确保主从之间数据是一致的。但是毕竟主从就不是同一台Reids，主节点与从节点之间会有一定的延迟，所以主从数据的同步也会有一定的延迟，尽管延迟很多，到那时也存在延迟！我们所说的主从一致性问题正是因为这样的延迟而导致的。

比方说我们现在有一个Java应用，java应用现在需要来获取锁，比如要执行一个获取锁命令：set lick thread1 NX EX 10，这不就是一个Redis的写操作吗？这个操作执行到主节点的时候，主节点上就会去保存锁的标识：lock = thread1，而后主节点就会向从节点进行数据同步。

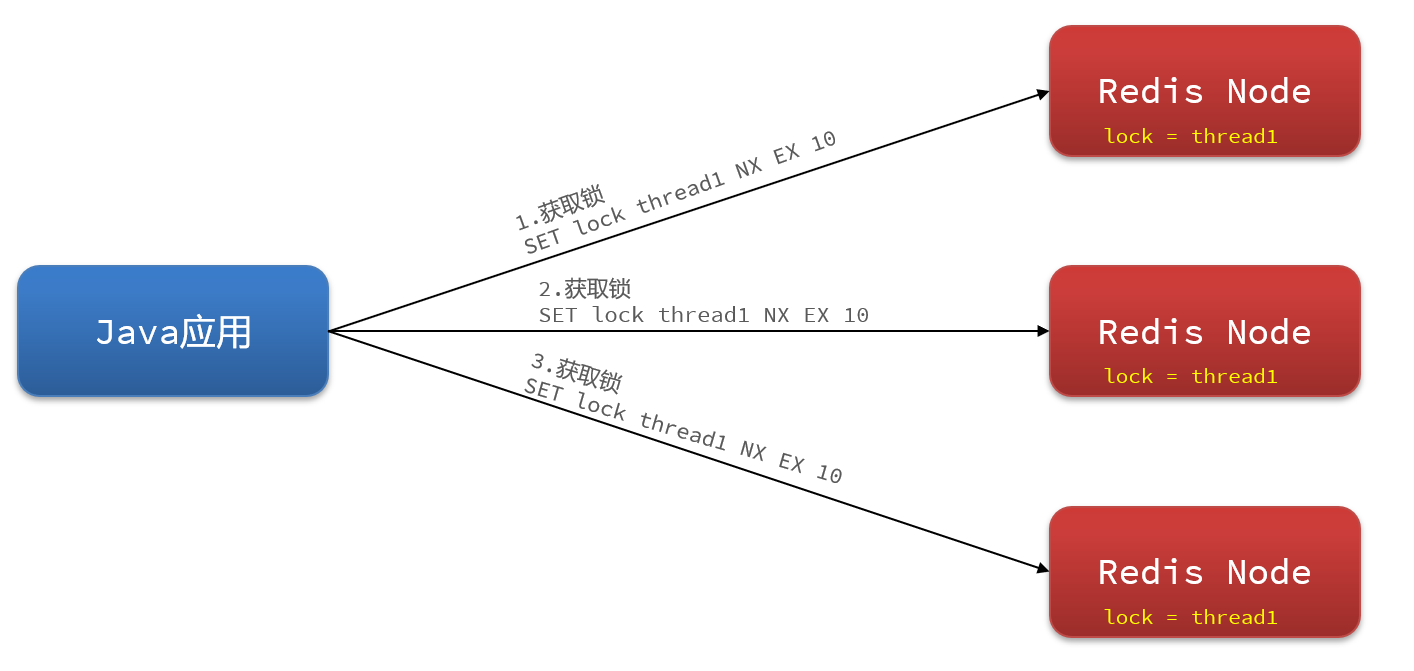


到那时就在同步的时候，主节点发生了故障，也就是说数据同步尚未完成！这个时候Redis主从集群会有哨兵在监控集群状态，当哨兵发现主节点宕机之后，首先java应用客户端和主节点的连接会断开，而后哨兵会在从节点中选出一个节点作为一个新的主节点，但是因为之前主从同步未完成啊，也就是说锁已经丢失了，所以此时我们的Java应用再来访问新的主节点时，就会发现锁已经不存在了，也就是锁失效了！此时再有其他线程来获取锁也能获取成功！是不是就发生了并发安全问题！这就是主从一致性导致的锁失效问题！



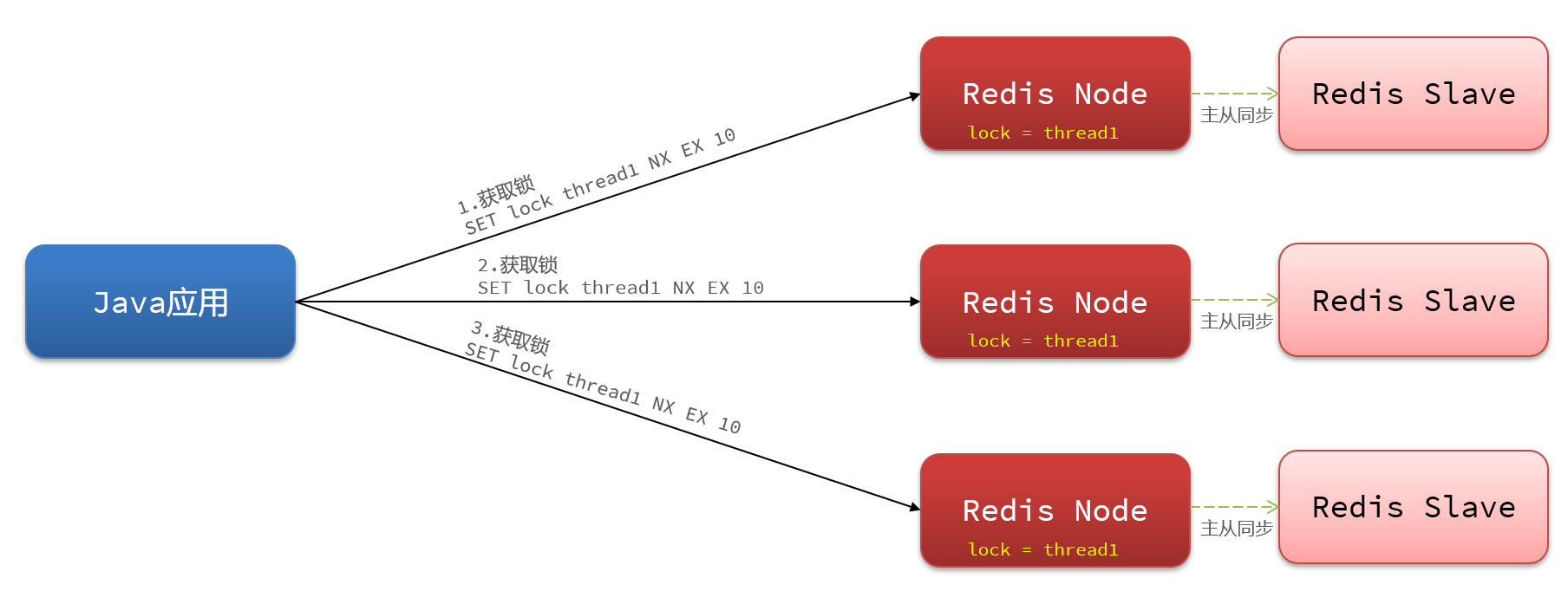
## Redisson解决主从一致性原理

那么Redisson又是怎么解决这个问题的呢？Redisson的思路非常简单粗暴！既然主从关系是导致一致性问题发生的原因，干脆Redis我就不要主从了呗，我的所有Redis节点都变成独立的Redis节点，相互之间没有任何关系，即没有主从，都可以去做读写。那么此时，我们获取锁的方式就变了，以前我们获取锁只需要找到主节点，然后在主节点中获取锁就行了，但是现在必须依次地向多个Redis节点都去获取锁，不管你是3个Redis节点还是5个Redis节点，获取锁时必须依次都获取锁，都保存锁的标识，才算是获取锁成功！

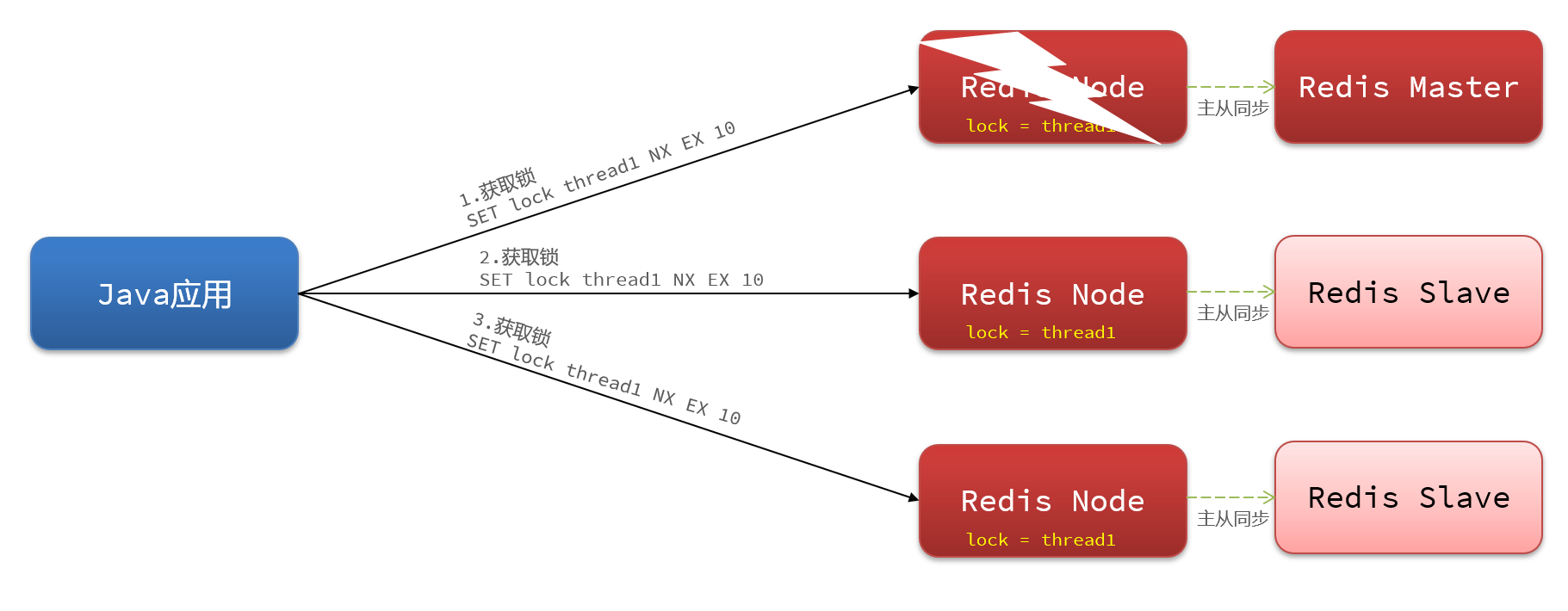


那么此时会不会出现安全问题？首先，因为我们没有主从，就不会发生主从数据同步，所以就不会有主从一致性问题！其次可用性，现在如果真的有一台节点宕机了，我们的Redis还是可用的，因为你宕机了还有别的Redis节点啊！只要别的节点存活着，我们的锁就依然有效，就是说可用性随着Redis节点数量的增多而越来越高！

以上我们不仅仅解决了主从一致性的问题，也保证了可用性！当然如果你觉得这样的可用性还不够，你还想让可用性更强一点，你担心挂一个Redis，保存锁的节点就少一台，没关系的，我们还可以给每个Reids节点去建立主从关系，让他们去做主从同步。



那加了主从同步之后会不会出现安全问题呢？其实不会！你想一下，如果真的有一台Redis出现宕机，这台Redis宕机的时候，刚好没有向从机完成同步，也就是这台宕机节点的从机就没有锁的标识，现在按照我们说得Redis集群的哨兵原理，这台从机就会变成新的主节点，新主节点没有锁标识，此时如果有一个新线程妄图乘虚而入想要来获取锁，是不可能获取成功的，因为我们上面说了，线程只有在每一台节点都拿到锁，才算获取成功！现在尽管新主节点是空缺的，你可以从这台新主节点获取锁，但是其他节点是获取不到的，所以最终还是会获取锁失败！也就是说只要我们有任意一个主节点存活着，那么其他线程就不可能拿到锁，就不会出现锁失效的问题！



你看，上面Redisson的这样一套方案，保留了主从同步机制，确保了整个Redis集群高可用性的特性，同时也避免了主从一致引发的锁失效问题！这套方案在Redisson中有一个名字，叫：MultiLock，联锁，把多个独立的锁联合在一起，变成一个联合起来的锁。

我们在使用这样的锁的时候，也是比较灵活的哦，你可以就弄几个独立节点，不建立主从关系，你也可以建立主从关系，让可用性变得更强！

接下来我们做测试的时候就不去为独立Redis建立主从关系了，一是太复杂了，需要的节点太多，因为我们测试是整三个独立节点，如果建立主从那就要整6个了，太多太麻烦太复杂了；二是因为没钱内存不够啊！

## 测试Redisson解决主从一致性：联锁使用

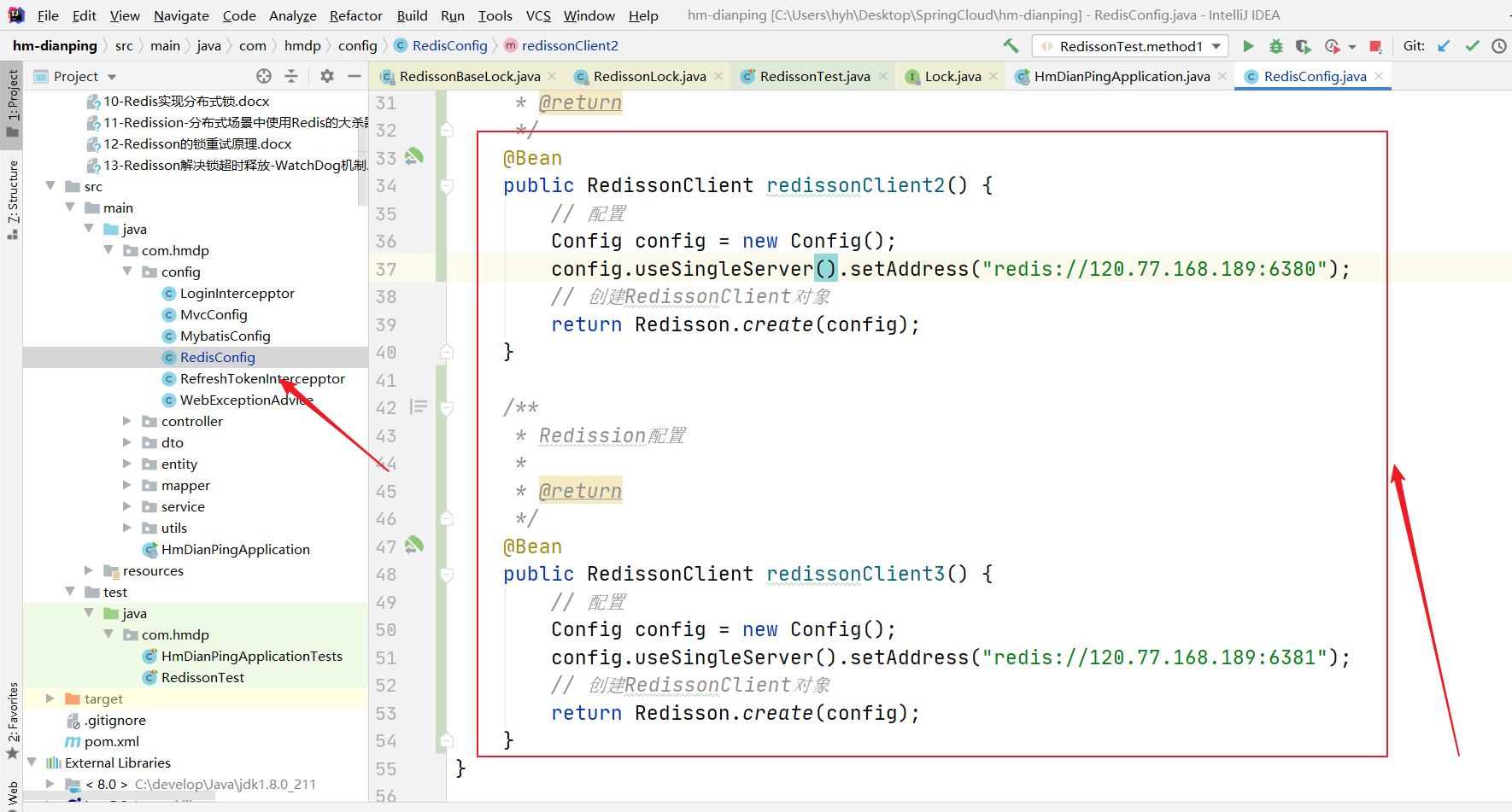
### 1：在服务器使用Docker创建3个Redis容器





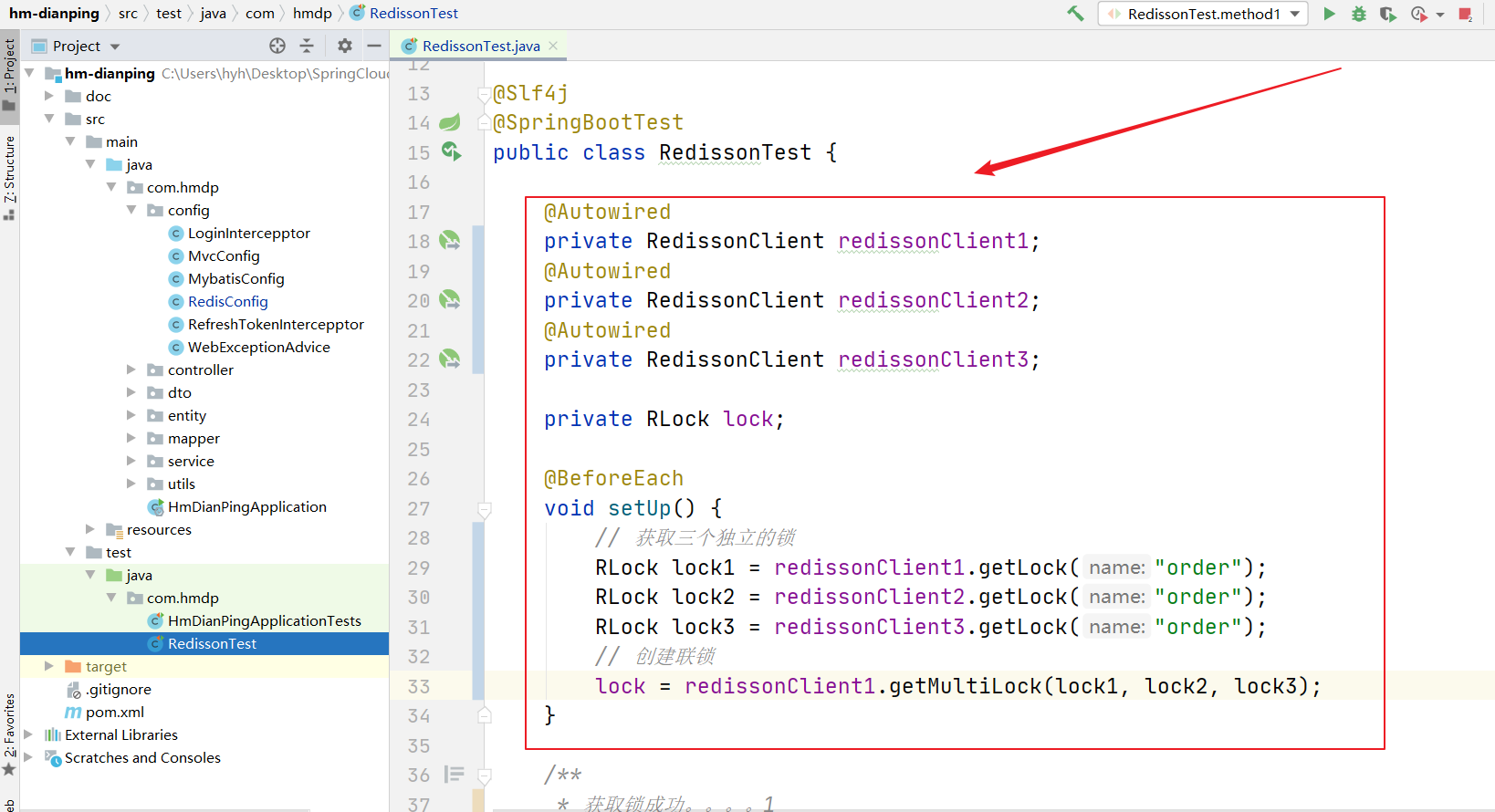
### 2：重新配置RedissonClient客户端

密码不用配置，因为我用Docker容器创建的Redis容器没有配置密码！

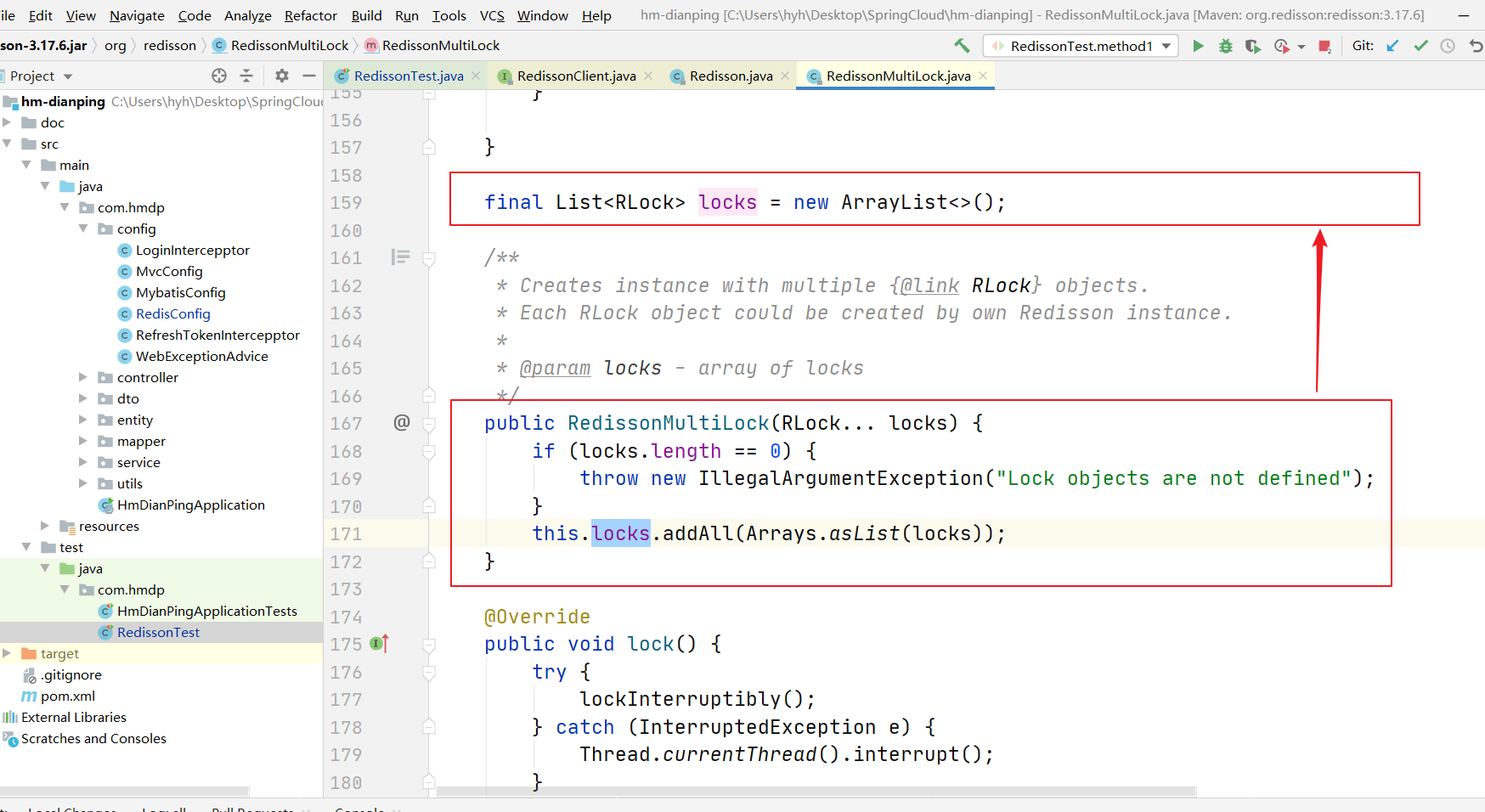


### 3：测试用例中创建联锁multiLock

我们利用3个RedisClient分别获取三个独立的锁，然后把三个独立的锁合在一起变成联锁！



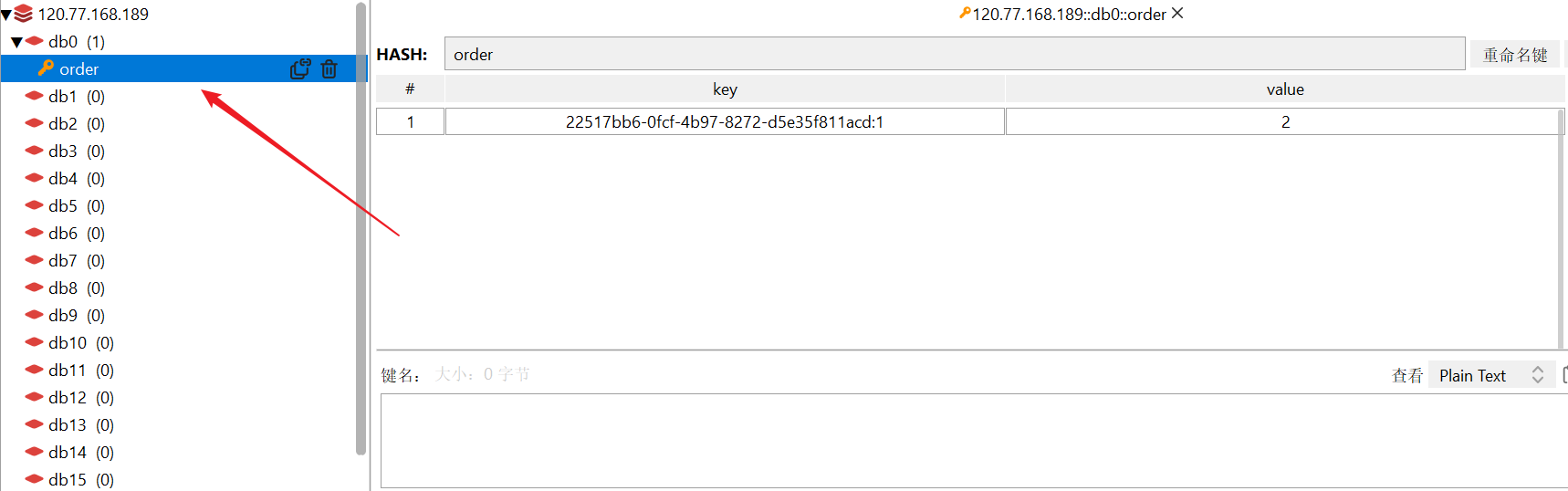
创建联锁解读：我们跟踪源码可以看到，当我们调用创建联锁的方法：RedissonClient.getMultiLock()，最终会调用RedissonMultiLock的构造方法：new RedissonMultiLock()，可以看到可变参数（可就是独立的Redis锁）在构造方法里转变成了集合一起添加到成员变量licks中了！locks就是一个集合，也就是说我们多个独立的锁都扔到这个集合locks中，按照联锁的原理，将来在获取锁时，就会依次把这个集合lcoks里每一个锁都尝试去获取一遍，都成功了才算获取锁成功！



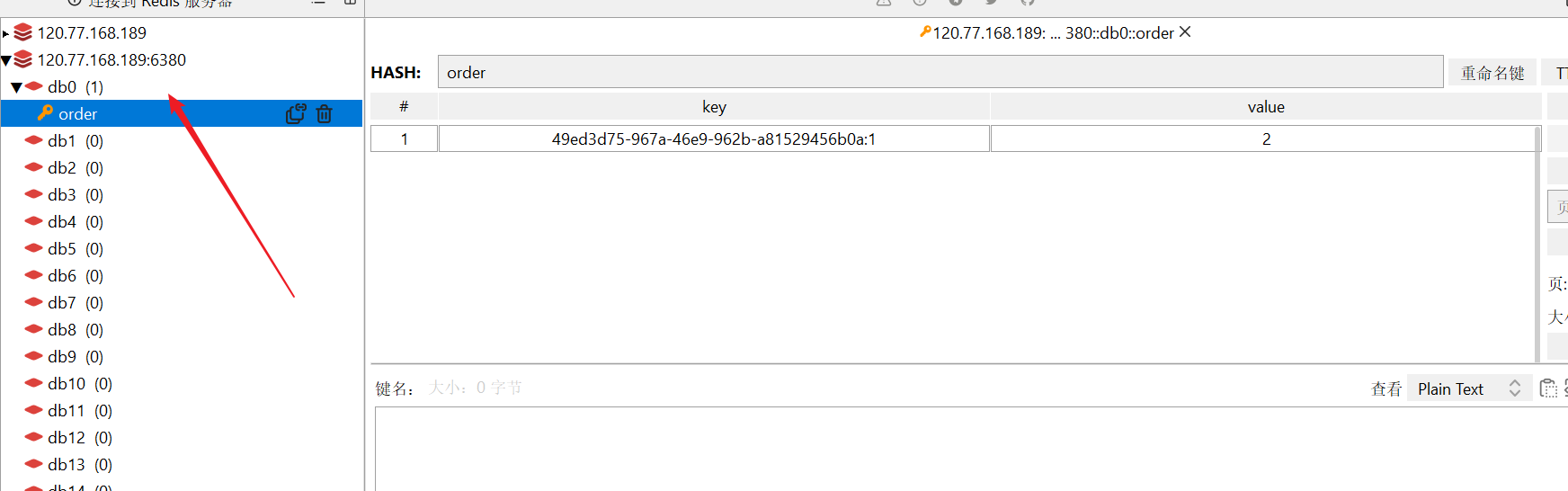
### 4：使用联锁

拿到联锁以后，使用和以前没什么区别！代码都不用动！

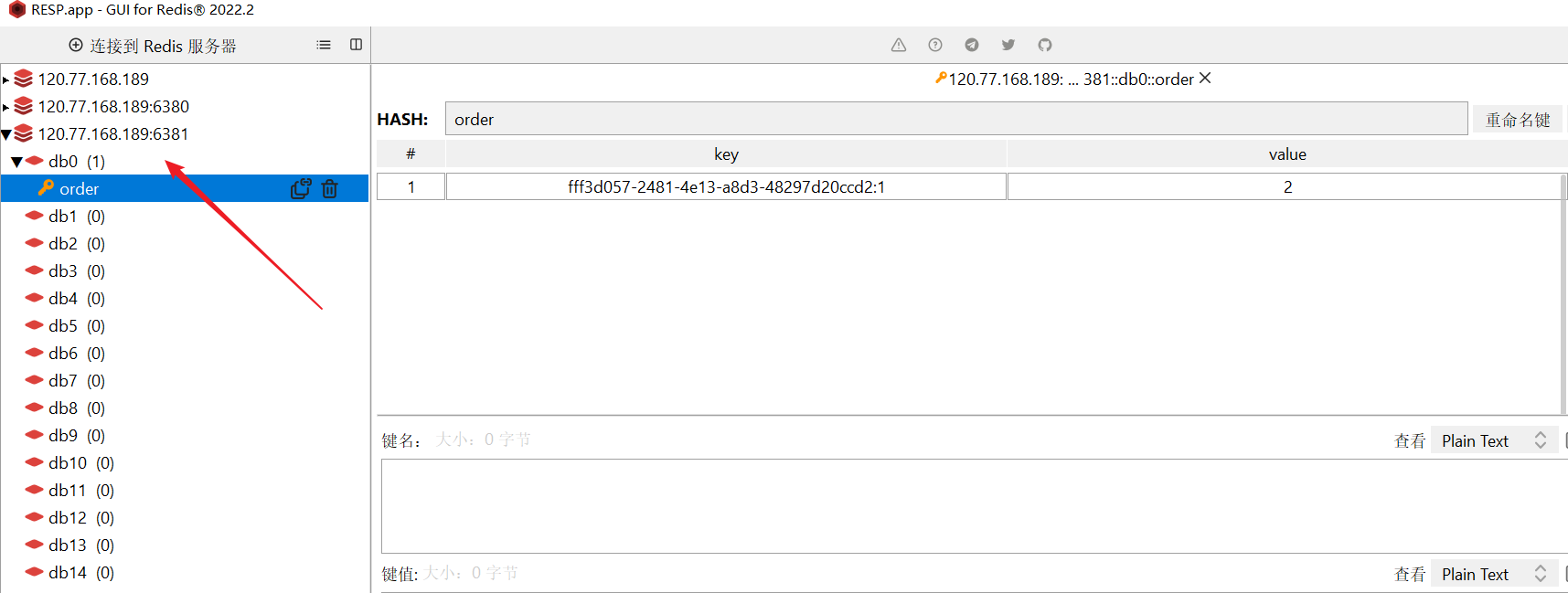
6379节点：



6380节点：



6381节点：



可以看到联合锁中的每一个独立的锁都是一个可重入锁，与之前的实现没什么差别，只不过联锁是把三个独立的锁联合起来分别去获取！

所谓的联锁就是多个独立的锁联合在一起，而每一个独立的锁就更跟我们之前讲的Redisson获取锁的原理时一毛一样的！