【REDIS】RedLock多节点分布式锁算法分析

Original XUANXUAN丶 xuanxuan96 2 days ago

之前在《【REDIS】脑裂问题及其解决方案》已经简单分析过，对于分布式锁，如一般的setNX命令、Lua脚本或第三方实现的Redisson等，都是只能够用在单节点，**多节点情况下可能由于数据同步不及时**。

正如前文所述，从Redis 2.8版本开始，可以配置主服务器连接N个以上从服务器才允许对主服务器进行写操作。

但是，这个只能保证slave心跳时刻能有几个slave节点在连接，**无法确保心跳回复后的瞬间状态（因为心跳是有频率的）。**

因为Redis主从复制是异步的！是异步的！异步的！异步！没办法确保从服务器确实收到了要写入的数据，所以还是有一定的数据丢失的可能性。

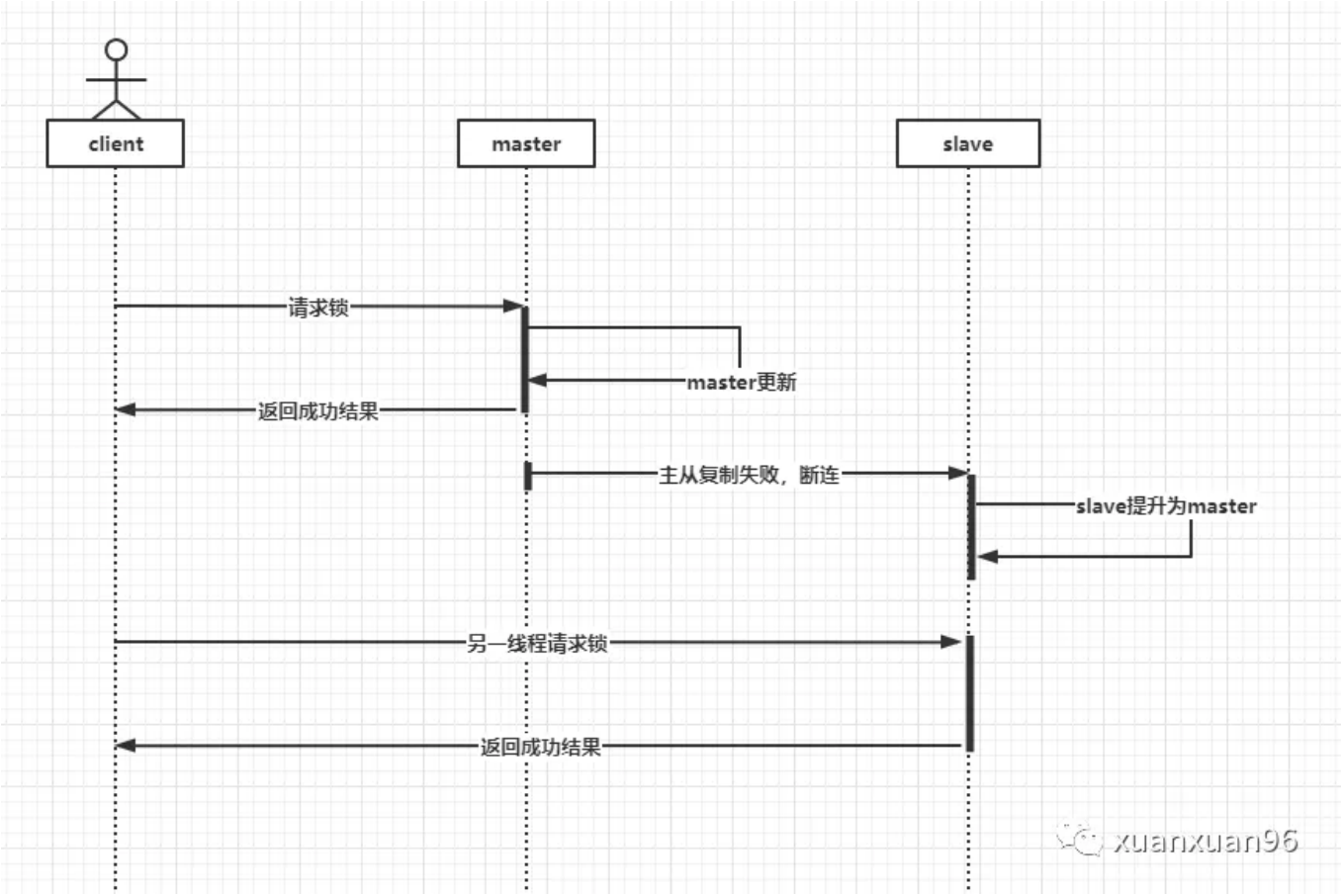
这一特性的工作原理如下：

1）slave服务器每秒钟ping一次master服务器，确认处理的复制流数量。
2）master服务器记住每个slave服务器最近一次ping的时间。
3）用户可以配置最少要有N个slave服务器，每个小于M秒的确认延迟。
4）**如果有N个以上slave服务器，并且确认延迟小于M秒，master服务器接受写操作。**

因此，如果就算上一刻master收到了N个slave少于M秒的答应，在主写完的下一刻还是会出现短连情况。

原因还是那句话，**redis主从复制是异步的**，上述两个配置（N和M）只是类似一个心跳机制，**不是同步的阻塞等待。**

还是上图说的清楚



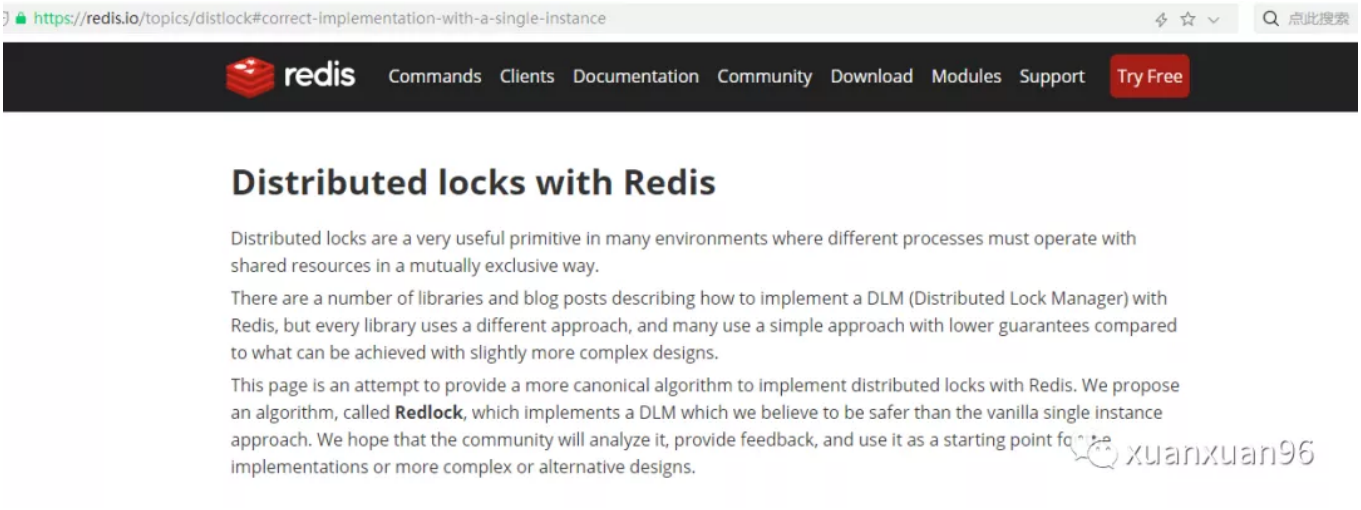
当然，也可以把这看做是CAP原则（一致性，可用性，分区容错性）不严格的一致性实现，**虽然不能百分百确保一致性，但至少保证了丢失的数据不会超过M秒内的数据量。**

虽然这个是很极端的情况，但是就是有可能会出现，所以就会有其他解决方案的出现。

Redis官方推荐的是RedLock算法。

RedLock，全称Redis Distributed Lock，也叫红锁，是redis官方推荐的多实例分布式锁的实现。

先看看官方的怎么说

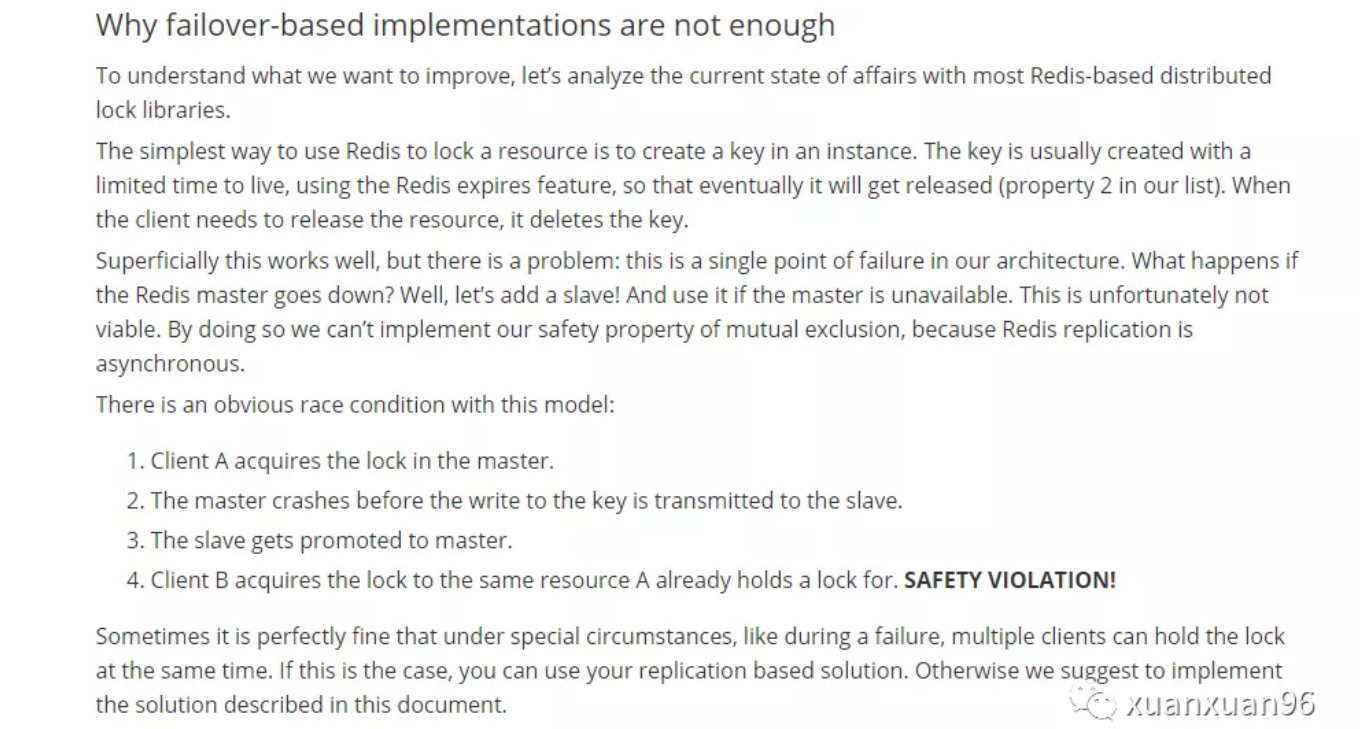


首先是一大段介绍，全英的，阿巴阿巴，除了个别单词应该都能看懂，再不行就谷歌翻译。

主要是对分布式锁描述，说分布式锁在很多情况是有用的，但在某些情况又不是有用的，简单来说，就是有用但又不完全有用。**一般的分布式锁只能单节点用，多节点的redis系统需要一个叫RedLock的算法来实现。**

大概就是这个事情。

然后又说了，为什么在基于failover机制（主备机制）下不能够满足呢?



简单翻译解析一下，跟我们上次《【REDIS】脑裂问题及其解决方案》分析的差不多。

在一般的分布式锁中，我们实现是通过判断一个key是否存在而确定是否有锁：如果不存在，获取锁成功；否则，再判断锁是否过期啥的。

当在单节点中时，这种做法是没问题的。但是当我们加入一个slave呢？这种情况就会改变。

1. **当我们在master中set了一个key**
2. **但是还没同步到slave中master就挂了，**
3. **slave提升为master**
4. **slave没有该key信息，另一个线程也能set这个key，资源冲突。**

因为主从复制是异步的！

所以还是建议换成本文提供的RedLock算法实现。

先来看看单节点的分布式锁

Correct implementation with a single instance

Before trying to overcome the limitation of the single instance setup described above, let's check how to do it correctly in this simple case, since this is actually a viable solution in applications where a race condition from time to time is acceptable, and because locking into a single instance is the foundation we'll use for the distributed algorithm described here.

To acquire the lock, the way to go is the following:

```
SET resource_name my_random_value NX PX 30000
```

The command will set the key only if it does not already exist (NX option), with an expire of 30000 milliseconds (PX option). The key is set to a value "my*random*value". This value must be unique across all clients and all lock requests.

Basically the random value is used in order to release the lock in a safe way, with a script that tells Redis: remove the key only if it exists and the value stored at the key is exactly the one I expect to be. This is accomplished by the following Lua script:

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```


如上面官方所说，单个Redis节点上加锁这样↓ 做没问题的，就是一顿setNX

```
SET resource_name my_random_value NX PX 30000
```

解锁这样↓ 也没问题

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

先看看安全又可靠的分布式锁的条件：

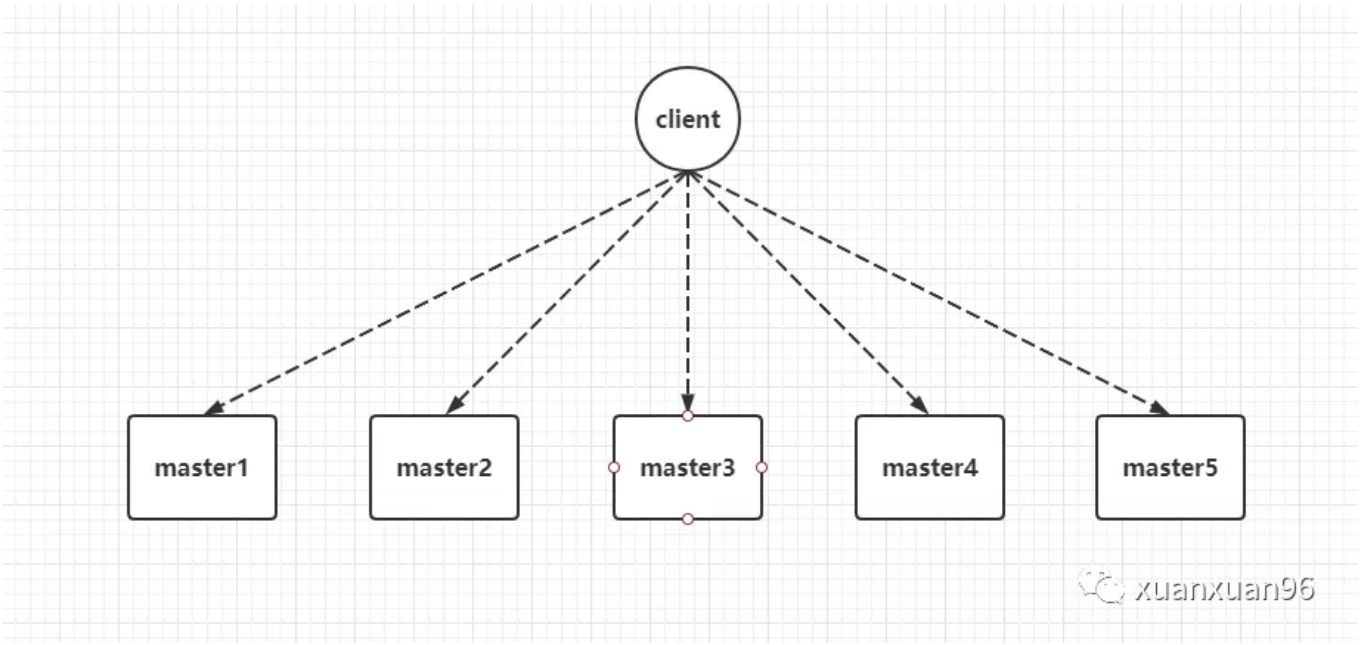
1. **一致性：互斥，任何时候只有一个客户端获取到锁。**
2. **分区可容忍性：不会死锁，就算一个持有锁的客户端宕机、业务发生死锁或网络分区等等，最终也会解锁。**
3. **可用性：只要大多数Redis节点正常工作，客户端应该都可以正常获取和释放锁。**

很明显上面的民工分布式锁就不满足这三个条件。

主要还是来看看多实例的RedLock算法吧。

RedLock算法

RedLock算法主要是在多独立节点系统中，以**“超过半数成功”**为准的算法，来判断是否成功加锁。



现在有**5个独立的redis节点。**

客户端获取锁的话，根据RedLock算法，会做以下操作。

主要流程：

1. **获取当前时间的时间戳**
2. **尝试向5个节点获取lock，就是将一个相同的Key和随机值 set到5个节点上去。**当在每个实例中设置锁时，客户端使用一个比总锁自动释放时间更小的超时来获取它。例如，TTL为10秒，获取锁的时间为5-50毫秒，否则超时。
3. **如果能在一定时间内并且半数以上（这里5个，一半以上就是3个或以上）返回的成功话，则成功获取到锁，**否则失败。如果客户端获取到锁的节点大于所有节点数量的一半，例如这里是3个，那么，这个client就获取锁成功。
4. 如果成功获取到锁，则锁的有效时间 = 总时间 - 获取锁时间。
5. **如果客户端获取锁失败（在指定时间内少于3个节点获取到锁），需要马上给所有获取节点发送解锁命令；**当然处理完所有业务后也要解锁。

从上面我们可以看出，**只要半数以上的节点在规定时间内获取锁成功，那么这个分布式锁就是成功的。**

算法分析

假设现在有一个客户端经过上面步骤后获取到了锁，那么

- 每个Redis节点中的锁的剩余存活时间相等为**TTL**。

- 每个锁请求到达各个Redis节点中的时间有差异。第一个锁成功请求最先在**T1**后返回，最后返回的请求在**T2**后返回。(T1,T2都小于最大失败时间)
- 并且每个实例之间存在时钟漂移**CLOCK_DRIFT(Time Drift)**

所以锁的实际有效时间 = TTL - (T2 - T1) - CLOCK_DIRFT，也就是说这段时间之内这个客户端占有了锁，其他客户端无法获取锁，这个时间之后锁就会过期。

（时钟漂移大概意思就是，每个节点不是同一台机器，所以他们每个节点每过一秒的时间不能完全一致，可能会有一点微小的差距，几乎可以忽略不计，但是又不能完全忽略不计，严谨，讲究。）

问题分析

节点取多少合适

官方建议5个。在算法的分布式版本中，我们假设我们有 N 个 Redis 主节点。这些节点是完全独立的，所以我们不使用复制或任何其他隐式协调系统。

由于获取锁成功的标志是大于等于（N / 2）+ 1个节点获取到锁，所以节点数最好取单数，所以5是一个比较合理的数字。

节点获取锁超时问题

获取锁时，客户端尝试获取所有 N 个实例中的锁，在所有实例中使用相同的键名和随机值。**在第 2 步中，当在每个实例中设置锁时，客户端使用一个比总锁自动释放时间更小的超时来获取它。**

例如，如果自动释放时间为 10 秒，则超时可能在 5-50 毫秒范围内，这样可以使得锁的有效时间更接近10秒，同时可以防止客户端长时间处于阻塞状态；但如果实例不可用，我们应该尽快尝试与下一个实例通信。

释放锁

这个跟之前的“解铃还须系铃人”一致，在锁的有效时间范围内，解锁的只能是占有锁的客户端。在锁过期后，这个占用就无效了，是个人都能解锁有手就行。当然如果获取锁失败，也要及时释放获取到了锁的节点，不能一直无效占用。

获取锁失败问题

如果客户端由于某种原因获取锁失败（或者它无法锁定 N/2+1 个实例或有效时间为负），它将尝试解锁所有实例（即使是它认为没有锁定的实例）能够锁定），从而不会一直等待一

个没有获取到锁的key在一个节点上占用着使其他客户端获取失败。

节点部署问题

这个模式跟以前的主从、哨兵或集群模式都不太一样，各个节点之间都是独立的，**没有相互复制的操作，并且建议各个节点隔离部署在不同的物理机器或虚拟机器上独立开来。**

客户端使用问题

客户端跟各个节点之间的请求尽量同时，因为同步的话就会浪费更多的等待时间。**比较理想的做法就是客户端同时向N个Redis节点发送异步set请求，所以，在客户端与N个Redis节点通信时，必须使用多路发送的方式(multiplex)，减少通信延时。**
同时，**客户端的锁超时时间要合理设置，**如果一个客户端获取到锁后宕机或网络分区了，那么其他客户端就一直等待直到锁过期才能释放。

潜在问题

故障恢复问题

如果5个Redis节点，有一个客户端在3个节点上获取到了锁，那么这个客户端就是成功获取到锁了。**但是马上就有一个获取到锁的节点宕机了，如果没有持久化数据就是没有了，那么（N / 2 + 1）的条件就不成立了就会有多个客户端获取到锁的问题。**
所以一定要持久化。如果使用AOF方式持久化还好，例如我们可以向某个实例发送shutdown和restart命令。即使节点被关闭，EX设置的时间仍在计算，锁的排他性仍能保证。但是如果是断电的情况那就不好办了，可以把持久化设为Always，但是性能就会大幅下降。
另一种解决方案是在这个Redis节点重新启动后，令其在一定时间内不参与任何加锁。在间隔了一整个锁生命周期后，重新参与到锁服务中。这样可以保证所有在这台实例宕机期间内的key都已经过期或被释放。

延时重启机制能够保证Redis即使不使用任何持久化策略，仍能保证锁的可靠性。但是这种策略可能会牺牲掉一部分可用性。
例如集群中超过半数的实例都宕机了，那么整个分布式锁系统需要等待一整个锁有效期的时间才能重新提供锁服务。

当然RedLock也不是完美的，留个坑有空填。

好了先这样！

参考：https://redis.io/topics/distlock#correct-implementation-with-a-single-instance

---