30-如何使用Redis实现分布式锁?

你好，我是蒋德钧。

上节课，我提到，在应对并发问题时，除了原子操作，Redis客户端还可以通过加锁的方式，来控制并发写操作对共享数据的修改，从而保证数据的正确性。

但是，Redis属于分布式系统，当有多个客户端需要争抢锁时，我们必须要保证，**这把锁不能是某个客户端本地的锁**。否则的话，其它客户端是无法访问这把锁的，当然也就不能获取这把锁了。

所以，在分布式系统中，当有多个客户端需要获取锁时，我们需要分布式锁。此时，锁是保存在一个共享存储系统中的，可以被多个客户端共享访问和获取。

Redis本身可以被多个客户端共享访问，正好就是一个共享存储系统，可以用来保存分布式锁。而且Redis的读写性能高，可以应对高并发的锁操作场景。所以，这节课，我就来和你聊聊如何基于Redis实现分布式锁。

我们日常在写程序的时候，经常会用到单机上的锁，你应该也比较熟悉了。而分布式锁和单机上的锁既有相似性，但也因为分布式锁是用在分布式场景中，所以又具有一些特殊的要求。

所以，接下来，我就先带你对比下分布式锁和单机上的锁，找出它们的联系与区别，这样就可以加深你对分布式锁的概念和实现要求的理解。

单机上的锁和分布式锁的联系与区别

我们先来看下单机上的锁。

对于在单机上运行的多线程程序来说，锁本身可以用一个变量表示。

- 变量值为0时，表示没有线程获取锁；
- 变量值为1时，表示已经有线程获取到锁了。

我们通常说的线程调用加锁和释放锁的操作，到底是啥意思呢？我来解释一下。实际上，一个线程调用加锁操作，其实就是检查锁变量值是否为0。如果是0，就把锁的变量值设置为1，表示获取到锁，如果不是0，就返回错误信息，表示加锁失败，已经有别的线程获取到锁了。而一个线程调用释放锁操作，其实就是将锁变量的值置为0，以便其它线程可以来获取锁。

我用一段代码来展示下加锁和释放锁的操作，其中，lock为锁变量。

```
acquire_lock(){
  if lock == 0
     lock = 1
     return 1
  else
     return 0
}
```

```
  lock = 0
  return 1
}
```

和单机上的锁类似，分布式锁同样可以**用一个变量来实现**。客户端加锁和释放锁的操作逻辑，也和单机上的加锁和释放锁操作逻辑一致：**加锁时同样需要判断锁变量的值，根据锁变量值来判断能否加锁成功；释放锁时需要把锁变量值设置为0，表明客户端不再持有锁**。

但是，和线程在单机上操作锁不同的是，在分布式场景下，**锁变量需要由一个共享存储系统来维护**，只有这样，多个客户端才可以通过访问共享存储系统来访问锁变量。相应的，**加锁和释放锁的操作就变成了读取、判断和设置共享存储系统中的锁变量值**。

这样一来，我们就可以得出实现分布式锁的两个要求。

- 要求一：分布式锁的加锁和释放锁的过程，涉及多个操作。所以，在实现分布式锁时，我们需要保证这些锁操作的原子性；
- 要求二：共享存储系统保存了锁变量，如果共享存储系统发生故障或宕机，那么客户端也就无法进行锁操作了。在实现分布式锁时，我们需要考虑保证共享存储系统的可靠性，进而保证锁的可靠性。

好了，知道了具体的要求，接下来，我们就来学习下Redis是怎么实现分布式锁的。

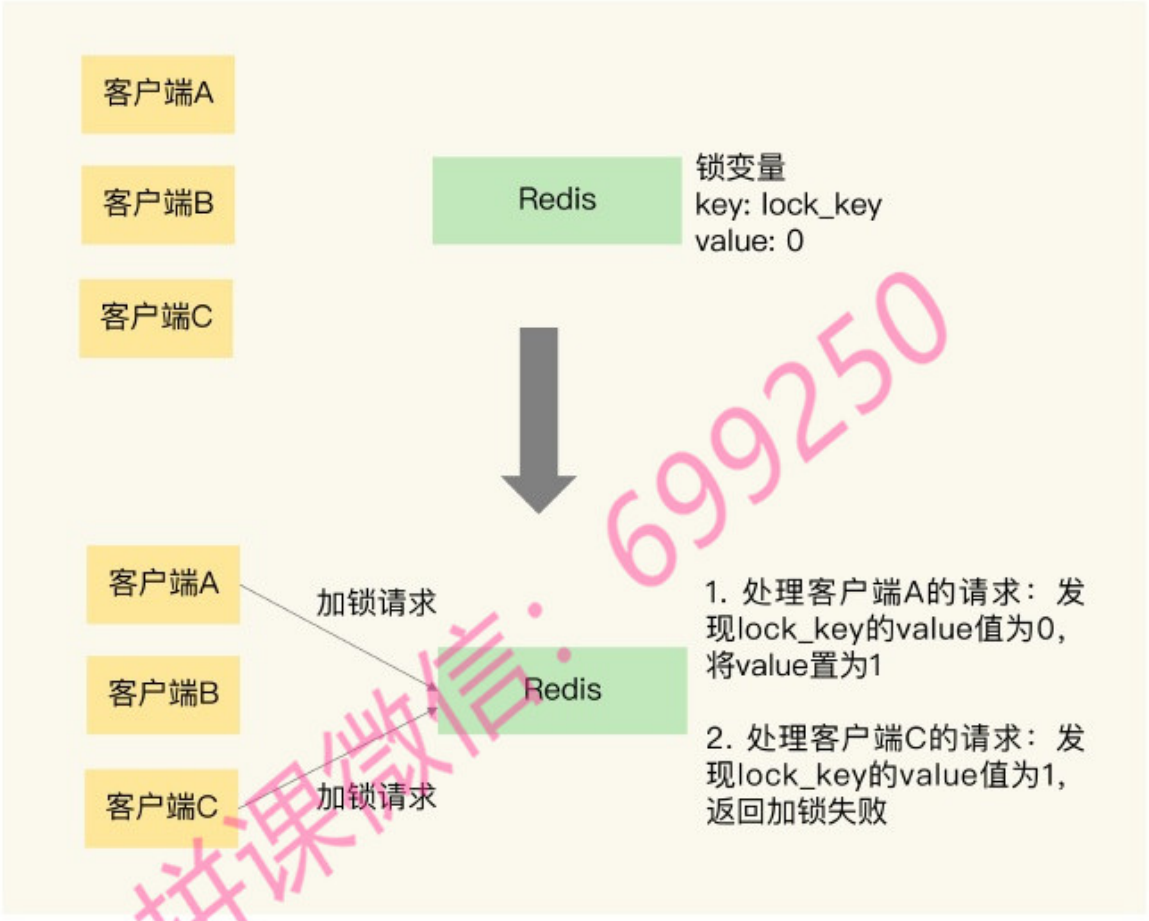
其实，我们既可以基于单个Redis节点来实现，也可以使用多个Redis节点实现。在这两种情况下，锁的可靠性是不一样的。我们先来看基于单个Redis节点的实现方法。

基于单个Redis节点实现分布式锁

作为分布式锁实现过程中的共享存储系统，Redis可以使用键值对来保存锁变量，再接收和处理不同客户端发送的加锁和释放锁的操作请求。那么，键值对的键和值具体是怎么定的呢？

我们要赋予锁变量一个变量名，把这个变量名作为键值对的键，而锁变量的值，则是键值对的值，这样一来，Redis就能保存锁变量了，客户端也就可以通过Redis的命令操作来实现锁操作。

为了帮助你理解，我画了一张图片，它展示Redis使用键值对保存锁变量，以及两个客户端同时请求加锁的操作过程。



可以看到，Redis可以使用一个键值对lock_key:0来保存锁变量，其中，键是lock_key，也是锁变量的名称，锁变量的初始值是0。

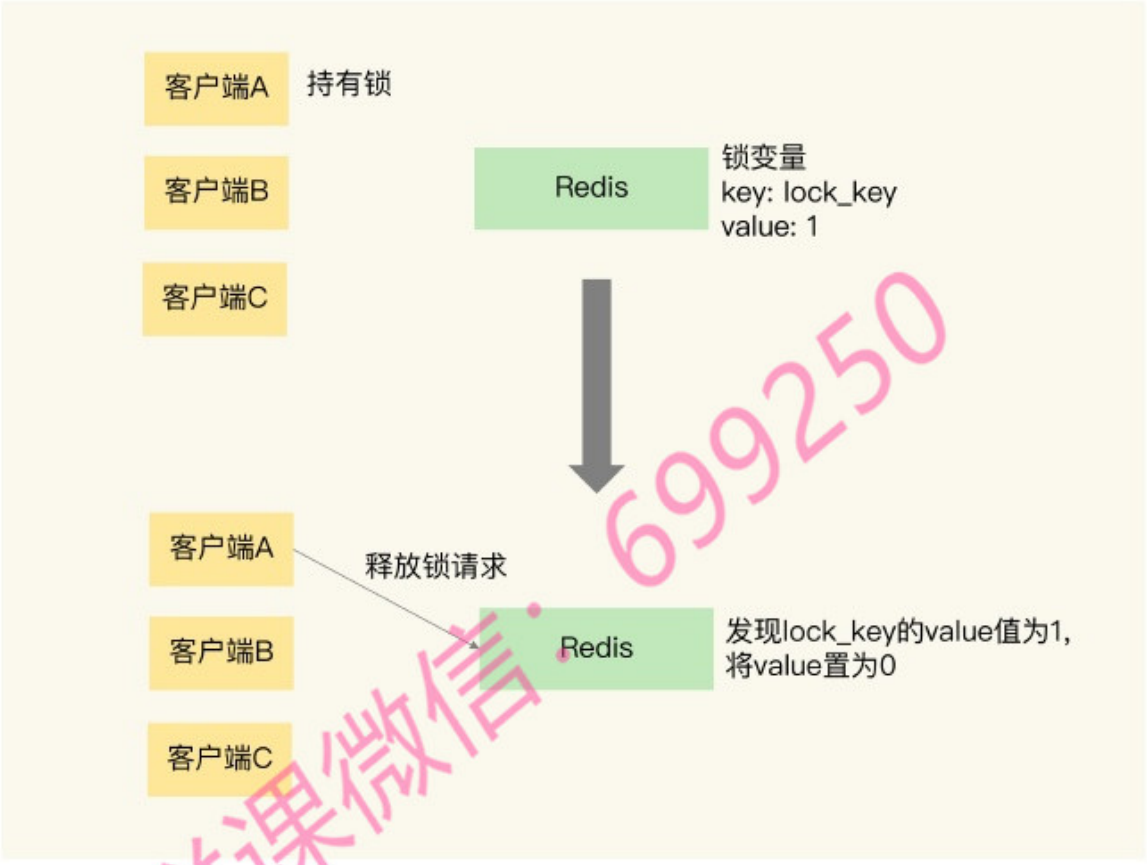
我们再来分析下加锁操作。

在图中，客户端A和C同时请求加锁。因为Redis使用单线程处理请求，所以，即使客户端A和C同时把加锁请求发给了Redis，Redis也会串行处理它们的请求。

我们假设Redis先处理客户端A的请求，读取lock_key的值，发现lock_key为0，所以，Redis就把lock_key的value置为1，表示已经加锁了。紧接着，Redis处理客户端C的请求，此时，Redis会发现lock_key的值已经为1了，所以就返回加锁失败的信息。

刚刚说的是加锁的操作，那释放锁该怎么操作呢？其实，释放锁就是直接把锁变量值设置为0。

我还是借助一张图片来解释一下。这张图片展示了客户端A请求释放锁的过程。当客户端A持有锁时，锁变量lock_key的值为1。客户端A执行释放锁操作后，Redis将lock_key的值置为0，表明已经没有客户端持有锁了。



因为加锁包含了三个操作（读取锁变量、判断锁变量值以及把锁变量值设置为1），而这三个操作在执行时需要保证原子性。那怎么保证原子性呢？

上节课，我们学过，要想保证操作的原子性，有两种通用的方法，分别是使用Redis的单命令操作和使用Lua脚本。那么，在分布式加锁场景下，该怎么应用这两个方法呢？

我们先来看下，Redis可以用哪些单命令操作实现加锁操作。

首先是SETNX命令，它用于设置键值对的值。具体来说，就是这个命令在执行时会判断键值对是否存在，如果不存在，就设置键值对的值，如果存在，就不做任何设置。

举个例子，如果执行下面的命令时，key不存在，那么key会被创建，并且值会被设置为value；如果key已经存在，SETNX不做任何赋值操作。

```
SETNX key value
```

对于释放锁操作来说，我们可以在执行完业务逻辑后，使用DEL命令删除锁变量。不过，你不用担心锁变量被删除后，其他客户端无法请求加锁了。因为SETNX命令在执行时，如果要设置的键值对（也就是锁变量）不存在，SETNX命令会先创建键值对，然后设置它的值。所以，释放锁之后，再有客户端请求加锁时，SETNX命令会创建保存锁变量的键值对，并设置锁变量的值，完成加锁。

总结来说，我们就可以用SETNX和DEL命令组合来实现加锁和释放锁操作。下面的伪代码示例显示了锁操作

的过程，你可以看下。

```
// 加锁
SETNX lock_key 1
// 业务逻辑
DO THINGS
// 释放锁
DEL lock_key
```

不过，使用SETNX和DEL命令组合实现分布锁，存在两个潜在的风险。

第一个风险是，假如某个客户端在执行了SETNX命令、加锁之后，紧接着却在操作共享数据时发生了异常，结果一直没有执行最后的DEL命令释放锁。因此，锁就一直被这个客户端持有，其它客户端无法拿到锁，也无法访问共享数据和执行后续操作，这会给业务应用带来影响。

针对这个问题，一个有效的解决方法是，**给锁变量设置一个过期时间**。这样一来，即使持有锁的客户端发生了异常，无法主动地释放锁，Redis也会根据锁变量的过期时间，在锁变量过期后，把它删除。其它客户端在锁变量过期后，就可以重新请求加锁，这就不会出现无法加锁的问题了。

我们再来看第二个风险。如果客户端A执行了SETNX命令加锁后，假设客户端B执行了DEL命令释放锁，此时，客户端A的锁就被误释放了。如果客户端C正好也在申请加锁，就可以成功获得锁，进而开始操作共享数据。这样一来，客户端A和C同时在对共享数据进行操作，数据就会被修改错误，这也是业务层不能接受的。

为了应对这个问题，我们需要**能区分来自不同客户端的锁操作**，具体咋做呢？其实，我们可以在锁变量的值上想想办法。

在使用SETNX命令进行加锁的方法中，我们通过把锁变量值设置为1或0，表示是否加锁成功。1和0只有两种状态，无法表示究竟是哪个客户端进行的锁操作。所以，我们在加锁操作时，可以让每个客户端给锁变量设置一个唯一值，这里的唯一值就可以用来标识当前操作的客户端。在释放锁操作时，客户端需要判断，当前锁变量的值是否和自己的唯一标识相等，只有在相等的情况下，才能释放锁。这样一来，就不会出现误释放锁的问题了。

知道了解决方案，那么，在Redis中，具体是怎么实现的呢？我们再来了解下。

在查看具体的代码前，我要先带你学习下Redis的SET命令。

我们刚刚在说SETNX命令的时候提到，对于不存在的键值对，它会先创建再设置值（也就是“不存在即设置”），为了能达到和SETNX命令一样的效果，Redis给SET命令提供了类似的选项NX，用来实现“不存在即设置”。如果使用了NX选项，SET命令只有在键值对不存在时，才会进行设置，否则不做赋值操作。此外，SET命令在执行时还可以带上EX或PX选项，用来设置键值对的过期时间。

举个例子，执行下面的命令时，只有key不存在时，SET才会创建key，并对key进行赋值。另外，**key的存活时间由seconds或者milliseconds选项值来决定**。

```
SET key value [EX seconds | PX milliseconds]  [NX]
```

有了SET命令的NX和EX/PX选项后，我们就可以用下面的命令来实现加锁操作了。

```
// 加锁, unique_value作为客户端唯一性的标识
SET lock_key unique_value NX PX 10000
```

其中，unique_value是客户端的唯一标识，可以用一个随机生成的字符串来表示，PX 10000则表示lock_key会在10s后过期，以免客户端在这期间发生异常而无法释放锁。

因为在加锁操作中，每个客户端都使用了一个唯一标识，所以在释放锁操作时，我们需要判断锁变量的值，是否等于执行释放锁操作的客户端的唯一标识，如下所示：

```
//释放锁 比较unique_value是否相等，避免误释放
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```


这是使用Lua脚本（unlock.script）实现的释放锁操作的伪代码，其中，KEYS[1]表示lock_key，ARGV[1]是当前客户端的唯一标识，这两个值都是我们在执行Lua脚本时作为参数传入的。

最后，我们执行下面的命令，就可以完成锁释放操作了。

```
redis-cli  --eval  unlock.script lock_key , unique_value
```

你可能也注意到了，在释放锁操作中，我们使用了Lua脚本，这是因为，释放锁操作的逻辑也包含了读取锁变量、判断值、删除锁变量的多个操作，而Redis在执行Lua脚本时，可以以原子性的方式执行，从而保证了锁释放操作的原子性。

好了，到这里，你了解了如何使用SET命令和Lua脚本在Redis单节点上实现分布式锁。但是，我们现在只用了一个Redis实例来保存锁变量，如果这个Redis实例发生故障宕机了，那么锁变量就没有了。此时，客户端也无法进行锁操作了，这就会影响到业务的正常执行。所以，我们在实现分布式锁时，还需要保证锁的可靠性。那怎么提高呢？这就要提到基于多个Redis节点实现分布式锁的方式了。

基于多个Redis节点实现高可靠的分布式锁

当我们要实现高可靠的分布式锁时，就不能只依赖单个的命令操作了，我们需要按照一定的步骤和规则进行

加解锁操作，否则，就可能会出现锁无法工作的情况。“一定的步骤和规则”是指啥呢？其实就是分布式锁的算法。

为了避免Redis实例故障而导致的锁无法工作的问题，Redis的开发者Antirez提出了分布式锁算法Redlock。

Redlock算法的基本思路，是让客户端和多个独立的Redis实例依次请求加锁，如果客户端能够和半数以上的实例成功地完成加锁操作，那么我们就认为，客户端成功地获得分布式锁了，否则加锁失败。这样一来，即使有单个Redis实例发生故障，因为锁变量在其它实例上也有保存，所以，客户端仍然可以正常地进行锁操作，锁变量并不会丢失。

我们来具体看下Redlock算法的执行步骤。Redlock算法的实现需要有N个独立的Redis实例。接下来，我们可以分成3步来完成加锁操作。

第一步是，客户端获取当前时间。

第二步是，客户端按顺序依次向N个Redis实例执行加锁操作。

这里的加锁操作和在单实例上执行的加锁操作一样，使用SET命令，带上NX，EX/PX选项，以及带上客户端的唯一标识。当然，如果某个Redis实例发生故障了，为了保证在这种情况下，Redlock算法能够继续运行，我们需要给加锁操作设置一个超时时间。

如果客户端在和一个Redis实例请求加锁时，一直到超时都没有成功，那么此时，客户端会和下一个Redis实例继续请求加锁。加锁操作的超时时间需要远远地小于锁的有效时间，一般也就是设置为几十毫秒。

第三步是，一旦客户端完成了和所有Redis实例的加锁操作，客户端就要计算整个加锁过程的总耗时。

客户端只有在满足下面的这两个条件时，才能认为是加锁成功。

- 条件一：客户端从超过半数（大于等于 N/2+1）的Redis实例上成功获取到了锁；
- 条件二：客户端获取锁的总耗时没有超过锁的有效时间。

在满足了这两个条件后，我们需要重新计算这把锁的有效时间，计算的结果是锁的最初有效时间减去客户端为获取锁的总耗时。如果锁的有效时间已经来不及完成共享数据的操作了，我们可以释放锁，以免出现还没完成数据操作，锁就过期了的情况。

当然，如果客户端在和所有实例执行完加锁操作后，没能同时满足这两个条件，那么，客户端向所有Redis节点发起释放锁的操作。

在Redlock算法中，释放锁的操作和在单实例上释放锁的操作一样，只要执行释放锁的Lua脚本就可以了。这样一来，只要N个Redis实例中的半数以上实例能正常工作，就能保证分布式锁的正常工作了。

所以，在实际的业务应用中，如果你想要提升分布式锁的可靠性，就可以通过Redlock算法来实现。

小结

分布式锁是由共享存储系统维护的变量，多个客户端可以向共享存储系统发送命令进行加锁或释放锁操作。

Redis作为一个共享存储系统，可以用来实现分布式锁。

在基于单个Redis实例实现分布式锁时，对于加锁操作，我们需要满足三个条件。

1. 加锁包括了读取锁变量、检查锁变量值和设置锁变量值三个操作，但需要以原子操作的方式完成，所以，我们使用SET命令带上NX选项来实现加锁；
2. 锁变量需要设置过期时间，以免客户端拿到锁后发生异常，导致锁一直无法释放，所以，我们在SET命令执行时加上EX/PX选项，设置其过期时间；
3. 锁变量的值需要能区分来自不同客户端的加锁操作，以免在释放锁时，出现误释放操作，所以，我们使用SET命令设置锁变量值时，每个客户端设置的值是一个唯一值，用于标识客户端。

和加锁类似，释放锁也包含了读取锁变量值、判断锁变量值和删除锁变量三个操作，不过，我们无法使用单个命令来实现，所以，我们可以采用Lua脚本执行释放锁操作，通过Redis原子性地执行Lua脚本，来保证释放锁操作的原子性。

不过，基于单个Redis实例实现分布式锁时，会面临实例异常或崩溃的情况，这会导致实例无法提供锁操作，正因为此，Redis也提供了Redlock算法，用来实现基于多个实例的分布式锁。这样一来，锁变量由多个实例维护，即使有实例发生了故障，锁变量仍然是存在的，客户端还是可以完成锁操作。Redlock算法是实现高可靠分布式锁的一种有效解决方案，你可以在实际应用中把它用起来。

每课一问

按照惯例，我给你提个小问题。这节课，我提到，我们可以使用SET命令带上NX和EX/PX选项进行加锁操作，那么，我想请你再思考一下，我们是否可以用下面的方式来实现加锁操作呢？

```
// 加锁
SETNX lock_key unique_value
EXPIRE lock_key 10S
// 业务逻辑
DO THINGS
```

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得今天的内容对你有所帮助，也欢迎你分享给你的朋友或同事。我们下节课见。

精选留言：

- Kaito 2020-10-28 00:09:37
  是否可以使用 SETNX + EXPIRE 来完成加锁操作?

  不可以这么使用。使用 2 个命令无法保证操作的原子性，在异常情况下，加锁结果会不符合预期。异常情况主要分为以下几种情况：

  1、SETNX 执行成功，执行 EXPIRE 时由于网络问题设置过期失败

  2、SETNX 执行成功，此时 Redis 实例宕机，EXPIRE 没有机会执行

如果发生以上情况，并且客户端在释放锁时发生异常，没有正常释放锁，那么这把锁就会一直无法释放，其他线程都无法再获得锁。

下面说一下关于 Redis 分布式锁可靠性的问题。

使用单个 Redis 节点（只有一个master）使用分布锁，如果实例宕机，那么无法进行锁操作了。那么采用主从集群模式部署是否可以保证锁的可靠性？

答案是也很难保证。如果在 master 上加锁成功，此时 master 宕机，由于主从复制是异步的，加锁操作的命令还未同步到 slave，此时主从切换，新 master 节点依旧会丢失该锁，对业务来说相当于锁失效了。

所以 Redis 作者才提出基于多个 Redis 节点（master节点）的 Redlock 算法，但这个算法涉及的细节很多，作者在提出这个算法时，业界的分布式系统专家还与 Redis 作者发生过一场争论，来评估这个算法的可靠性，争论的细节都是关于异常情况可能导致 Redlock 失效的场景，例如加锁过程中客户端发生了阻塞、机器时钟发生跳跃等等。

感兴趣的可以看下这篇文章，详细介绍了争论的细节，以及 Redis 分布式锁在各种异常情况是否安全的分析，收益会非常大：http://zhangtielei.com/posts/blog-redlock-reasoning.html。

简单总结，基于 Redis 使用分布锁的注意点：

1、使用 SET $lock_key $unique_val EX $second NX 命令保证加锁原子性，并为锁设置过期时间

2、锁的过期时间要提前评估好，要大于操作共享资源的时间

3、每个线程加锁时设置随机值，释放锁时判断是否和加锁设置的值一致，防止自己的锁被别人释放

4、释放锁时使用 Lua 脚本，保证操作的原子性

5、基于多个节点的 Redlock，加锁时超过半数节点操作成功，并且获取锁的耗时没有超过锁的有效时间才算加锁成功

6、Redlock 释放锁时，要对所有节点释放（即使某个节点加锁失败了），因为加锁时可能发生服务端加锁成功，由于网络问题，给客户端回复网络包失败的情况，所以需要把所有节点可能存的锁都释放掉

7、使用 Redlock 时要避免机器时钟发生跳跃，需要运维来保证，对运维有一定要求，否则可能会导致 Redlock 失效。例如共 3 个节点，线程 A 操作 2 个节点加锁成功，但其中 1 个节点机器时钟发生跳跃，锁提前过期，线程 B 正好在另外 2 个节点也加锁成功，此时 Redlock 相当于失效了（Redis 作者和分布式系统专家争论的重要点就在这）

8、如果为了效率，使用基于单个 Redis 节点的分布式锁即可，此方案缺点是允许锁偶尔失效，优点是简单效率高

9、如果是为了正确性，业务对于结果要求非常严格，建议使用 Redlock，但缺点是使用比较重，部署成本高 [22赞]

- Darren 2020-10-28 17:40:11
  其实分布式锁选择Etcd的话，可能会更好

Etcd 支持以下功能，正是依赖这些功能来实现分布式锁的：

1、Lease 即租约机制（TTL，Time To Live）机制：

Etcd 可以为存储的 KV 对设置租约，当租约到期，KV 将失效删除；同时也支持续约，即 KeepAlive；

Redis在这方面很难实现，一般假设通过SETNX设置的时间10S，如果发生网络抖动，万一业务执行超过10S，此时别的线程就能回去到锁；

2、Revision 机制：

Etcd 每个 key 带有一个 Revision 属性值，每进行一次事务对应的全局 Revision 值都会加一，因此每个 key 对应的 Revision 属性值都是全局唯一的。通过比较 Revision 的大小就可以知道进行写操作的顺序。在实现分布式锁时，多个程序同时抢锁，根据 Revision 值大小依次获得锁，可以避免 “惊群效应”，实现公平锁；

Redis很难实现公平锁，而且在某些情况下，也会产生 “惊群效应”；

3、Prefix 即前缀机制，也称目录机制：

Etcd 可以根据前缀（目录）获取该目录下所有的 key 及对应的属性（包括 key, value 以及 revision 等）；

Redis也可以的，使用keys命令或者scan，生产环境一定要使用scan；

4、Watch 机制：

Etcd Watch 机制支持 Watch 某个固定的 key，也支持 Watch 一个目录（前缀机制），当被 Watch 的 key 或目录发生变化，客户端将收到通知；

Redis只能通过客户端定时轮训的形式去判断key是否存在； [4赞]

- Geek_9a0c9f 2020-10-29 13:38:19

python的简单实现：

```
import threading
import time

import redis

redis_con = redis.StrictRedis(host='127.0.0.1', port=6379, db=0)

# redis_con.ttl('name')
redis_con.set("global", 10)

#守护进程，用来检测时间是否到期，快到期了，就续命
def add_time(redis_con,key_name):
```

```
while True:
time = redis_con.ttl(key_name)
if time == 1:
print("=====")
redis_con.expire(key_name, 2)
print(redis_con.ttl(key_name))

def test_distribute_lock():
thead_id = str(time.time())
if_has_key = redis_con.setnx("lock", thead_id)
try:
if not if_has_key:
print("等待请稍后重试")
else:
redis_con.expire('lock', 2)
# 开启守护进程
t1 = threading.Thread(target=add_time, args=(redis_con, 'lock'))
t1.setDaemon(True)
t1.start()
redis_con.decr('global', 1)
time.sleep(2)
except Exception as e:
print(e)
finally:
# 执行完毕，释放锁，保证每个线程只能删除自己的锁
if str(redis_con.get("lock"), encoding="utf-8") == thead_id:
redis_con.delete('lock')
redis_con.close()

if __name__ == '__main__':
for i in range(5):
t = threading.Thread(target=test_distribute_lock)
t.start()
t.join()
print("===>") [1赞]
```


- test 2020-10-28 08:53:01
  不能这样做，因为两个命令就不是原子操作了。

  set nx px的时候如果拿到锁的客户端在使用过程中超出了其设置的超时时间，那么就有这把锁同时被两个客户端持有的风险，所以需要在使用过程中不断去更新其过期时间。 [1赞]

- 每天晒白牙 2020-10-30 09:23:16
  是不能把setnx 和 expire 命令分开的，因为无法保证两个操作执行的原子性，可能遇到各种异常，无法满足预期

- 小鹿 2020-10-28 17:45:14
  Redlock 里面的redis所有实例是所有master机器吗，如果是的话，同样的key怎么写入到所有master，key哈希完只会写入到一个master上，这点我不太明白。

- 林肯 2020-10-28 10:56:06

  // 加锁, unique_value作为客户端唯一性的标识
  SET lock_key unique_value NX PX 10000
  每个客户端的key都不一样，那还怎么加锁呢？加锁都本质就是多个客户端操作同一个共享变量啊？

- 小狼 2020-10-28 10:40:02
  如果用 Lua 脚本来使用上述两条命令，可以保证原子性操作，进而也就可以通过使用上述两条命令来实现分布式锁；只是，这种方式的调用成本显然比单独的 setnx 命令成本要高

- 杨逸林 2020-10-28 10:37:51
  针对这个问题，一个有效的解决方法是，给锁变量设置一个过期时间。
  有一种情况，假如 A 系统获得了锁，设置了过期时间为 5s。假如 A 系统执行了一个比较慢的操作，费时 6 s，刚好在 5-6s 之间，有个 B 系统看没有人创建键值对，然后也获得了锁。这种情况怎么办

- 番茄smd 2020-10-28 09:41:23
  不知道后面会不会讲解redisson

- 三木子 2020-10-28 09:10:23
  SETNX lock_key unique_value
  EXPIRE lock_key 10S
  这是两条命令操作，不能保证原子性了

- 写点啥呢 2020-10-28 08:45:41
  请问老师，redis分布式锁有没有提供加锁失败->进程挂起->锁释放唤醒挂起进程的方案，避免SETNX失败后进程自旋。

- oops 2020-10-28 08:29:32
  不能，2个命令没法保证原子性