# **分布式锁用Redis还是Zookeeper？**

https://zhuanlan.zhihu.com/p/73807097

[](https://www.zhihu.com/people/cong-cong-cong-cong-qian-zhu-you-zuo-shan)

**[明月照我心](https://www.zhihu.com/people/cong-cong-cong-cong-qian-zhu-you-zuo-shan" \t "https://zhuanlan.zhihu.com/p/_blank)**

精通敲键盘

273 人赞同了该文章

作者：jianfeng 来源：[石杉的架构笔记](https://link.zhihu.com/?target=http://jianfeng/" \t "https://zhuanlan.zhihu.com/p/_blank)

为什么用分布式锁？在讨论这个问题之前，我们先来看一个业务场景。



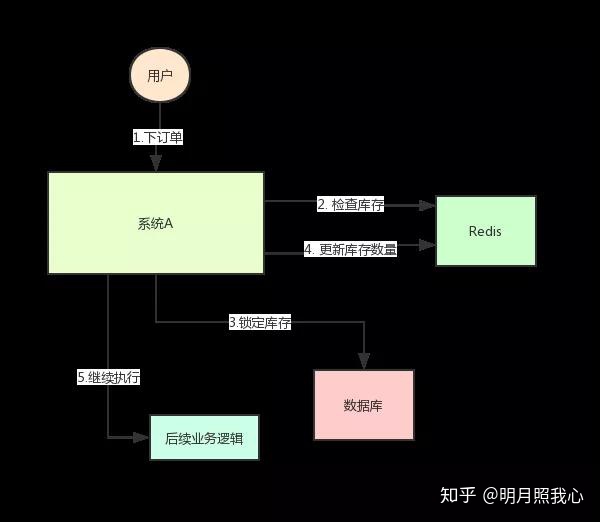
图片来自 Pexels

**为什么用分布式锁？**

系统 A 是一个电商系统，目前是一台机器部署，系统中有一个用户下订单的接口，但是用户下订单之前一定要去检查一下库存，确保库存足够了才会给用户下单。

由于系统有一定的并发，所以会预先将商品的库存保存在 Redis 中，用户下单的时候会更新 Redis 的库存。

此时系统架构如下：

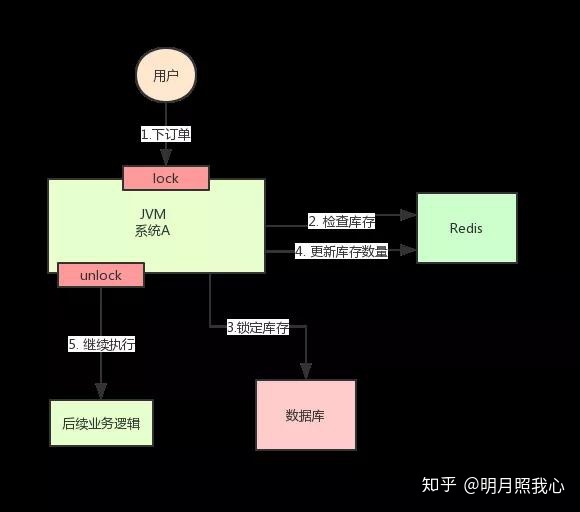


但是这样一来会产生一个问题：假如某个时刻，Redis 里面的某个商品库存为 1。

此时两个请求同时到来，其中一个请求执行到上图的第 3 步，更新数据库的库存为 0，但是第 4 步还没有执行。

而另外一个请求执行到了第 2 步，发现库存还是 1，就继续执行第 3 步。这样的结果，是导致卖出了 2 个商品，然而其实库存只有 1 个。

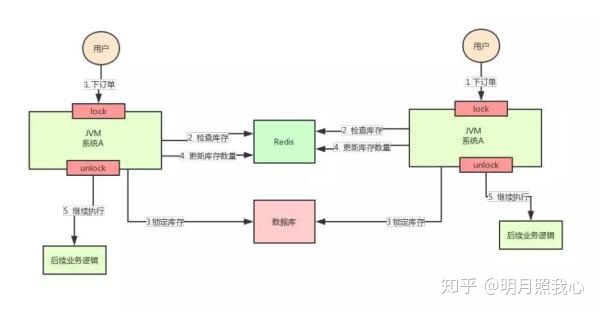
很明显不对啊！这就是典型的库存超卖问题。此时，我们很容易想到解决方案：用锁把 2、3、4 步锁住，让他们执行完之后，另一个线程才能进来执行第 2 步。



按照上面的图，在执行第 2 步时，使用 Java 提供的 Synchronized 或者 ReentrantLock 来锁住，然后在第 4 步执行完之后才释放锁。

这样一来，2、3、4 这 3 个步骤就被“锁”住了，多个线程之间只能串行化执行。

但是好景不长，整个系统的并发飙升，一台机器扛不住了。现在要增加一台机器，如下图：



增加机器之后，系统变成上图所示，我的天！假设此时两个用户的请求同时到来，但是落在了不同的机器上，那么这两个请求是可以同时执行了，还是会出现库存超卖的问题。

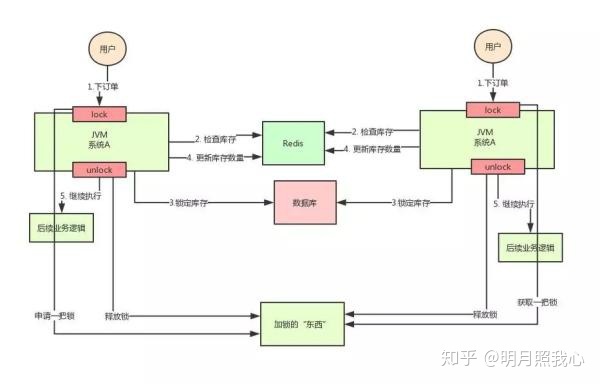
为什么呢？因为上图中的两个 A 系统，运行在两个不同的 JVM 里面，他们加的锁只对属于自己 JVM 里面的线程有效，对于其他 JVM 的线程是无效的。

因此，这里的问题是：Java 提供的原生锁机制在多机部署场景下失效了，这是因为两台机器加的锁不是同一个锁（两个锁在不同的 JVM 里面）。

那么，我们只要保证两台机器加的锁是同一个锁，问题不就解决了吗？此时，就该分布式锁隆重登场了。

分布式锁的思路是：在整个系统提供一个全局、唯一的获取锁的“东西”，然后每个系统在需要加锁时，都去问这个“东西”拿到一把锁，这样不同的系统拿到的就可以认为是同一把锁。

至于这个“东西”，可以是 Redis、Zookeeper，也可以是数据库。文字描述不太直观，我们来看下图：



通过上面的分析，我们知道了库存超卖场景在分布式部署系统的情况下使用 Java 原生的锁机制无法保证线程安全，所以我们需要用到分布式锁的方案。

那么，如何实现分布式锁呢？接着往下看！

**基于 Redis 实现分布式锁**

上面分析为啥要使用分布式锁了，这里我们来具体看看分布式锁落地的时候应该怎么样处理。

**①常见的一种方案就是使用 Redis 做分布式锁**

使用 Redis 做分布式锁的思路大概是这样的：在 Redis 中设置一个值表示加了锁，然后释放锁的时候就把这个 Key 删除。

具体代码是这样的：

// 获取锁

// NX是指如果key不存在就成功，key存在返回false，PX可以指定过期时间

SET anyLock unique\_value NX PX 30000

// 释放锁：通过执行一段lua脚本

// 释放锁涉及到两条指令，这两条指令不是原子性的

// 需要用到redis的lua脚本支持特性，redis执行lua脚本是原子性的

if redis.call("get",KEYS[1]) == ARGV[1] then

return redis.call("del",KEYS[1])

else

return 0

end

这种方式有几大要点：

* 一定要用 SET key value NX PX milliseconds 命令。如果不用，先设置了值，再设置过期时间，这个不是原子性操作，有可能在设置过期时间之前宕机，会造成死锁(Key 永久存在)
* Value 要具有唯一性。这个是为了在解锁的时候，需要验证 Value 是和加锁的一致才删除 Key。

这时避免了一种情况：假设 A 获取了锁，过期时间 30s，此时 35s 之后，锁已经自动释放了，A 去释放锁，但是此时可能 B 获取了锁。A 客户端就不能删除 B 的锁了。

除了要考虑客户端要怎么实现分布式锁之外，还需要考虑 Redis 的部署问题。

Redis 有 3 种部署方式：

* 单机模式
* Master-Slave+Sentinel 选举模式
* Redis Cluster 模式

使用 Redis 做分布式锁的缺点在于：如果采用单机部署模式，会存在单点问题，只要 Redis 故障了。加锁就不行了。

采用 Master-Slave 模式，加锁的时候只对一个节点加锁，即便通过 Sentinel 做了高可用，但是如果 Master 节点故障了，发生主从切换，此时就会有可能出现锁丢失的问题。

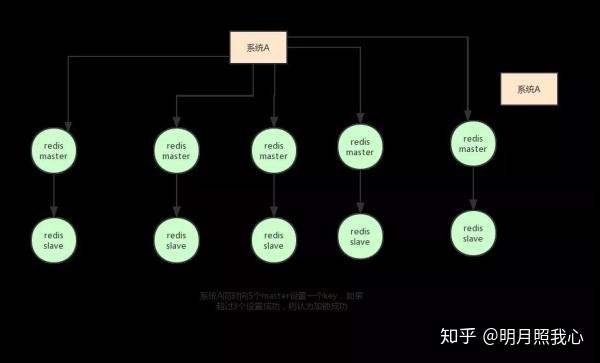
基于以上的考虑，Redis 的作者也考虑到这个问题，他提出了一个 RedLock 的算法。

这个算法的意思大概是这样的：假设 Redis 的部署模式是 Redis Cluster，总共有 5 个 Master 节点。

通过以下步骤获取一把锁：

* 获取当前时间戳，单位是毫秒。
* 轮流尝试在每个 Master 节点上创建锁，过期时间设置较短，一般就几十毫秒。
* 尝试在大多数节点上建立一个锁，比如 5 个节点就要求是 3 个节点（n / 2 +1）。
* 客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了。
* 要是锁建立失败了，那么就依次删除这个锁。
* 只要别人建立了一把分布式锁，你就得不断轮询去尝试获取锁。

但是这样的这种算法还是颇具争议的，可能还会存在不少的问题，无法保证加锁的过程一定正确。



**②另一种方式：Redisson**

此外，实现 Redis 的分布式锁，除了自己基于 Redis Client 原生 API 来实现之外，还可以使用开源框架：Redission。

Redisson 是一个企业级的开源 Redis Client，也提供了分布式锁的支持。我也非常推荐大家使用，为什么呢？

回想一下上面说的，如果自己写代码来通过 Redis 设置一个值，是通过下面这个命令设置的：

SET anyLock unique\_value NX PX 30000

这里设置的超时时间是 30s，假如我超过 30s 都还没有完成业务逻辑的情况下，Key 会过期，其他线程有可能会获取到锁。

这样一来的话，第一个线程还没执行完业务逻辑，第二个线程进来了也会出现线程安全问题。

所以我们还需要额外的去维护这个过期时间，太麻烦了~我们来看看 Redisson 是怎么实现的？

先感受一下使用 Redission 的爽：

Config config = new Config();

config.useClusterServers()

.addNodeAddress("redis://192.168.31.101:7001")

.addNodeAddress("redis://192.168.31.101:7002")

.addNodeAddress("redis://192.168.31.101:7003")

.addNodeAddress("redis://192.168.31.102:7001")

.addNodeAddress("redis://192.168.31.102:7002")

.addNodeAddress("redis://192.168.31.102:7003");

RedissonClient redisson = Redisson.create(config);

RLock lock = redisson.getLock("anyLock");

lock.lock();

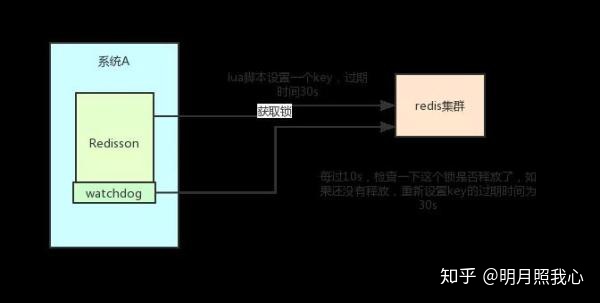
lock.unlock();

就是这么简单，我们只需要通过它的 API 中的 Lock 和 Unlock 即可完成分布式锁，他帮我们考虑了很多细节：

* Redisson 所有指令都通过 Lua 脚本执行，Redis 支持 Lua 脚本原子性执行。
* Redisson 设置一个 Key 的默认过期时间为 30s，如果某个客户端持有一个锁超过了 30s 怎么办？
* Redisson 中有一个 Watchdog 的概念，翻译过来就是看门狗，它会在你获取锁之后，每隔 10s 帮你把 Key 的超时时间设为 30s。

这样的话，就算一直持有锁也不会出现 Key 过期了，其他线程获取到锁的问题了。

* Redisson 的“看门狗”逻辑保证了没有死锁发生。(如果机器宕机了，看门狗也就没了。此时就不会延长 Key 的过期时间，到了 30s 之后就会自动过期了，其他线程可以获取到锁)



这里稍微贴出来其实现代码：

// 加锁逻辑

private <T> RFuture<Long> tryAcquireAsync(long leaseTime, TimeUnit unit, final long threadId) {

if (leaseTime != -1) {

return tryLockInnerAsync(leaseTime, unit, threadId, RedisCommands.EVAL\_LONG);

}

// 调用一段lua脚本，设置一些key、过期时间

RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(commandExecutor.getConnectionManager().getCfg().getLockWatchdogTimeout(), TimeUnit.MILLISECONDS, threadId, RedisCommands.EVAL\_LONG);

ttlRemainingFuture.addListener(new FutureListener<Long>() {

@Override

public void operationComplete(Future<Long> future) throws Exception {

if (!future.isSuccess()) {

return;

}

Long ttlRemaining = future.getNow();

// lock acquired

if (ttlRemaining == null) {

// 看门狗逻辑

scheduleExpirationRenewal(threadId);

}

}

});

return ttlRemainingFuture;

}

<T> RFuture<T> tryLockInnerAsync(long leaseTime, TimeUnit unit, long threadId, RedisStrictCommand<T> command) {

internalLockLeaseTime = unit.toMillis(leaseTime);

return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, command,

"if (redis.call('exists', KEYS[1]) == 0) then " +

"redis.call('hset', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +

"redis.call('hincrby', KEYS[1], ARGV[2], 1); " +

"redis.call('pexpire', KEYS[1], ARGV[1]); " +

"return nil; " +

"end; " +

"return redis.call('pttl', KEYS[1]);",

Collections.<Object>singletonList(getName()), internalLockLeaseTime, getLockName(threadId));

}

// 看门狗最终会调用了这里

private void scheduleExpirationRenewal(final long threadId) {

if (expirationRenewalMap.containsKey(getEntryName())) {

return;

}

// 这个任务会延迟10s执行

Timeout task = commandExecutor.getConnectionManager().newTimeout(new TimerTask() {

@Override

public void run(Timeout timeout) throws Exception {

// 这个操作会将key的过期时间重新设置为30s

RFuture<Boolean> future = renewExpirationAsync(threadId);

future.addListener(new FutureListener<Boolean>() {

@Override

public void operationComplete(Future<Boolean> future) throws Exception {

expirationRenewalMap.remove(getEntryName());

if (!future.isSuccess()) {

log.error("Can't update lock " + getName() + " expiration", future.cause());

return;

}

if (future.getNow()) {

// reschedule itself

// 通过递归调用本方法，无限循环延长过期时间

scheduleExpirationRenewal(threadId);

}

}

});

}

}, internalLockLeaseTime / 3, TimeUnit.MILLISECONDS);

if (expirationRenewalMap.putIfAbsent(getEntryName(), new ExpirationEntry(threadId, task)) != null) {

task.cancel();

}

}

另外，Redisson 还提供了对 Redlock 算法的支持，它的用法也很简单：

RedissonClient redisson = Redisson.create(config);

RLock lock1 = redisson.getFairLock("lock1");

RLock lock2 = redisson.getFairLock("lock2");

RLock lock3 = redisson.getFairLock("lock3");

RedissonRedLock multiLock = new RedissonRedLock(lock1, lock2, lock3);

multiLock.lock();

multiLock.unlock();

小结：本节分析了使用 Redis 作为分布式锁的具体落地方案以及其一些局限性，然后介绍了一个 Redis 的客户端框架 Redisson，这也是我推荐大家使用的，比自己写代码实现会少 Care 很多细节。

**基于 Zookeeper 实现分布式锁**

常见的分布式锁实现方案里面，除了使用 Redis 来实现之外，使用 Zookeeper 也可以实现分布式锁。

在介绍 Zookeeper(下文用 ZK 代替)实现分布式锁的机制之前，先粗略介绍一下 ZK 是什么东西：ZK 是一种提供配置管理、分布式协同以及命名的中心化服务。

ZK 的模型是这样的：ZK 包含一系列的节点，叫做 Znode，就好像文件系统一样，每个 Znode 表示一个目录。

然后 Znode 有一些特性：

* 有序节点：假如当前有一个父节点为 /lock，我们可以在这个父节点下面创建子节点，ZK 提供了一个可选的有序特性。

例如我们可以创建子节点“/lock/node-”并且指明有序，那么 ZK 在生成子节点时会根据当前的子节点数量自动添加整数序号。

也就是说，如果是第一个创建的子节点，那么生成的子节点为 /lock/node-0000000000，下一个节点则为 /lock/node-0000000001，依次类推。

* 临时节点：客户端可以建立一个临时节点，在会话结束或者会话超时后，ZK 会自动删除该节点。
* 事件监听：在读取数据时，我们可以同时对节点设置事件监听，当节点数据或结构变化时，ZK 会通知客户端。

当前 ZK 有如下四种事件：

* 节点创建
* 节点删除
* 节点数据修改
* 子节点变更

基于以上的一些 ZK 的特性，我们很容易得出使用 ZK 实现分布式锁的落地方案：

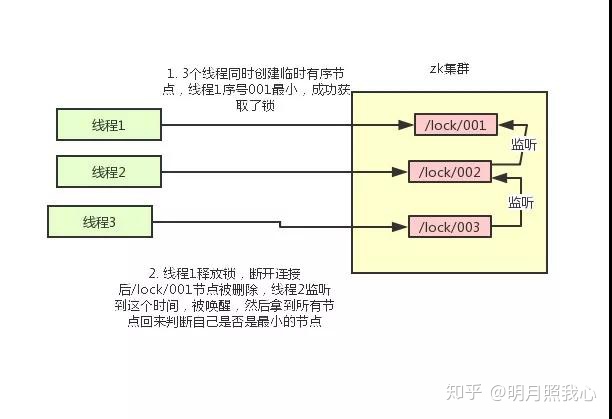
* 使用 ZK 的临时节点和有序节点，每个线程获取锁就是在 ZK 创建一个临时有序的节点，比如在 /lock/ 目录下。
* 创建节点成功后，获取 /lock 目录下的所有临时节点，再判断当前线程创建的节点是否是所有的节点的序号最小的节点。
* 如果当前线程创建的节点是所有节点序号最小的节点，则认为获取锁成功。
* 如果当前线程创建的节点不是所有节点序号最小的节点，则对节点序号的前一个节点添加一个事件监听。

比如当前线程获取到的节点序号为 /lock/003，然后所有的节点列表为[/lock/001，/lock/002，/lock/003]，则对 /lock/002 这个节点添加一个事件监听器。

如果锁释放了，会唤醒下一个序号的节点，然后重新执行第 3 步，判断是否自己的节点序号是最小。

比如 /lock/001 释放了，/lock/002 监听到时间，此时节点集合为[/lock/002，/lock/003]，则 /lock/002 为最小序号节点，获取到锁。

整个过程如下：



具体的实现思路就是这样，至于代码怎么写，这里比较复杂就不贴出来了。

**Curator 介绍**

Curator 是一个 ZK 的开源客户端，也提供了分布式锁的实现。它的使用方式也比较简单：

InterProcessMutex interProcessMutex = new InterProcessMutex(client,"/anyLock");

interProcessMutex.acquire();

interProcessMutex.release();

其实现分布式锁的核心源码如下：

private boolean internalLockLoop(long startMillis, Long millisToWait, String ourPath) throws Exception

{

boolean haveTheLock = false;

boolean doDelete = false;

try {

if ( revocable.get() != null ) {

client.getData().usingWatcher(revocableWatcher).forPath(ourPath);

}

while ( (client.getState() == CuratorFrameworkState.STARTED) && !haveTheLock ) {

// 获取当前所有节点排序后的集合

List<String> children = getSortedChildren();

// 获取当前节点的名称

String sequenceNodeName = ourPath.substring(basePath.length() + 1); // +1 to include the slash

// 判断当前节点是否是最小的节点

PredicateResults predicateResults = driver.getsTheLock(client, children, sequenceNodeName, maxLeases);

if ( predicateResults.getsTheLock() ) {

// 获取到锁

haveTheLock = true;

} else {

// 没获取到锁，对当前节点的上一个节点注册一个监听器

String previousSequencePath = basePath + "/" + predicateResults.getPathToWatch();

synchronized(this){

Stat stat = client.checkExists().usingWatcher(watcher).forPath(previousSequencePath);

if ( stat != null ){

if ( millisToWait != null ){

millisToWait -= (System.currentTimeMillis() - startMillis);

startMillis = System.currentTimeMillis();

if ( millisToWait <= 0 ){

doDelete = true; // timed out - delete our node

break;

}

wait(millisToWait);

}else{

wait();

}

}

}

// else it may have been deleted (i.e. lock released). Try to acquire again

}

}

}

catch ( Exception e ) {

doDelete = true;

throw e;

} finally{

if ( doDelete ){

deleteOurPath(ourPath);

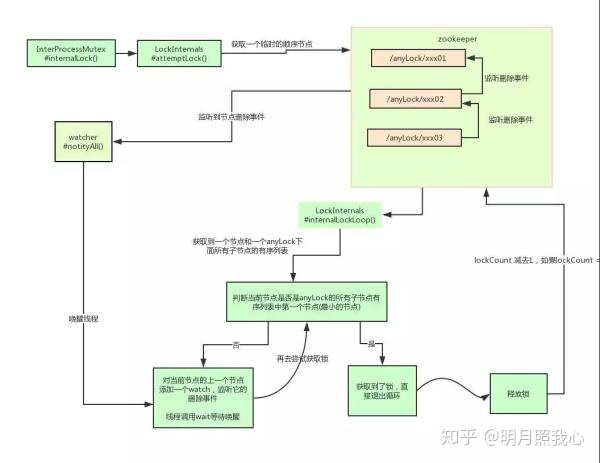
}

}

return haveTheLock;

}

其实 Curator 实现分布式锁的底层原理和上面分析的是差不多的。这里我们用一张图详细描述其原理：



小结：本节介绍了 ZK 实现分布式锁的方案以及 ZK 的开源客户端的基本使用，简要的介绍了其实现原理。

**两种方案的优缺点比较**

学完了两种分布式锁的实现方案之后，本节需要讨论的是 Redis 和 ZK 的实现方案中各自的优缺点。

对于 Redis 的分布式锁而言，它有以下缺点：

* 它获取锁的方式简单粗暴，获取不到锁直接不断尝试获取锁，比较消耗性能。
* 另外来说的话，Redis 的设计定位决定了它的数据并不是强一致性的，在某些极端情况下，可能会出现问题。锁的模型不够健壮。
* 即便使用 Redlock 算法来实现，在某些复杂场景下，也无法保证其实现 100% 没有问题，关于 Redlock 的讨论可以看 How to do distributed locking。
* Redis 分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能。

但是另一方面使用 Redis 实现分布式锁在很多企业中非常常见，而且大部分情况下都不会遇到所谓的“极端复杂场景”。

所以使用 Redis 作为分布式锁也不失为一种好的方案，最重要的一点是 Redis 的性能很高，可以支撑高并发的获取、释放锁操作。

对于 ZK 分布式锁而言:

* ZK 天生设计定位就是分布式协调，强一致性。锁的模型健壮、简单易用、适合做分布式锁。
* 如果获取不到锁，只需要添加一个监听器就可以了，不用一直轮询，性能消耗较小。

但是 ZK 也有其缺点：如果有较多的客户端频繁的申请加锁、释放锁，对于 ZK 集群的压力会比较大。

小结：综上所述，Redis 和 ZK 都有其优缺点。我们在做技术选型的时候可以根据这些问题作为参考因素。

**一些建议**

通过前面的分析，实现分布式锁的两种常见方案：Redis 和 ZK，他们各有千秋。应该如何选型呢？

就个人而言的话，我比较推崇 ZK 实现的锁：因为 Redis 是有可能存在隐患的，可能会导致数据不对的情况。但是，怎么选用要看具体在公司的场景了。

如果公司里面有 ZK 集群条件，优先选用 ZK 实现，但是如果说公司里面只有 Redis 集群，没有条件搭建 ZK 集群。

那么其实用 Redis 来实现也可以，另外还可能是系统设计者考虑到了系统已经有 Redis，但是又不希望再次引入一些外部依赖的情况下，可以选用 Redis。这个是要系统设计者基于架构来考虑了。

发布于 2019-07-16