# 背景

在单机时代，虽然不需要分布式锁，但也面临过类似的问题，只不过在单机的情况下，如果有多个线程要同时访问某个共享资源的时候，我们可以采用线程间加锁的机制，即当某个线程获取到这个资源后，就立即对这个资源进行加锁，当使用完资源之后，再解锁，其它线程就可以接着使用了。例如，在JAVA中，甚至专门提供了一些处理锁机制的一些API（synchronize/Lock等）。

同步排它锁(synchronize)。但是排他锁的缺点很明显：

1、线程串行导致的性能问题，性能消耗比较大。

2、无法解决分布式部署情况下跨进程问题；

到了分布式系统的时代，这种线程之间的锁机制，就没作用了，系统可能会有多份并且部署在不同的机器上，这些资源已经不是在线程之间共享了，而是属于进程之间共享的资源。

因此，为了解决这个问题，我们就必须引入「分布式锁」。

分布式锁，是指在分布式的部署环境下，通过锁机制来让多客户端互斥的对共享资源进行访问。

综上，需要加锁的场景需要满足以下条件：

1. 共享资源
2. 共享资源互斥
3. 多任务环境

我们先来看一个业务场景：

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：



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。

# 特点

分布式锁的实现由多种方式，但是不管怎样，分布式锁一般要有以下特点：

**排他性：**任意时刻，只能有一个client能获取到锁

**容错性：**分布式锁服务一般要满足AP（可用性Availability、分区容错性Partition tolerance），也就是说，只要分布式锁服务集群节点大部分存活，client就可以进行加锁解锁操作

**避免死锁：**分布式锁一定能得到释放，即使client在释放之前崩溃或者网络不可达

除了以上特点之外，分布式锁最好也能满足可重入、高性能、阻塞锁特性（AQS这种，能够及时从阻塞状态唤醒）等。

# 方案

用到分布式锁说明遇到了多个进程共同访问同一个资源的问题，一般是在两个场景下会防止对同一个资源的重复访问：

1、提高效率。比如多个节点计算同一批任务，如果某个任务已经有节点在计算了，那其他节点就不用重复计算了，以免浪费计算资源。不过重复计算也没事，不会造成其他更大的损失。也就是允许偶尔的失败。

2、保证正确性。这种情况对锁的要求就很高了，如果重复计算，会对正确性造成影响。这种不允许失败。

引入分布式锁势必要引入一个第三方的基础设施，比如MySQL，Redis，Zookeeper等，这些实现分布式锁的基础设施出问题了，也会影响业务。

针对分布式锁的实现，目前比较常用的方案：

1. 基于数据库实现分布式锁
2. 基于缓存（redis、memcache、tair）实现分布式锁
3. 基于Zookeeper实现分布式锁

**拓展：**

DLM分布式锁的实现机制：

<https://blog.csdn.net/xabc3000/article/details/8899467>

## DB锁

### 背景

要实现分布式锁，最简单的方式可能就是直接创建一张锁表，然后通过操作该表中的数据来实现了。

当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

创建这样一张数据库表：

CREATE TABLE `methodLock` (

 `id` int(11) NOT NULL AUTO\_INCREMENT COMMENT '主键',

 `method\_name` varchar(64) NOT NULL DEFAULT '' COMMENT '锁定的方法名',

 `desc` varchar(1024) NOT NULL DEFAULT '备注信息',

 `update\_time` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP COMMENT '保存数据时间，自动生成',

PRIMARY KEY (`id`),

UNIQUE KEY `uidx\_method\_name` (`method\_name `) USING BTREE) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';

当我们想要锁住某个方法时，执行以下SQL：

insert into methodLock(method\_name,desc) values (‘method\_name’,‘desc’)

因为我们对method\_name做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

当方法执行完毕之后，想要释放锁的话，需要执行以下Sql:

delete from methodLock where method\_name ='method\_name'

上面这种简单的实现有以下几个问题：

1、这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。

2、这把**锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁**。

3、这把锁只能是**非阻塞**的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。

4、这把锁是**非重入**的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

当然，我们也可以有其他方式解决上面的问题。

数据库是单点？搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。

没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。

非阻塞的？搞一个while循环，直到insert成功再返回成功。

非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

除了可以通过增删操作数据表中的记录以外，其实还可以借助数据中自带的锁来实现分布式的锁。

我们还用刚刚创建的那张数据库表。可以通过数据库的排他锁来实现分布式锁。基于MySql的InnoDB引擎，可以使用以下方法来实现加锁操作：

public boolean lock(){

   connection.setAutoCommit(false)

   while(true){

       try{

         result = select \* from methodLock where method\_name=xxx for update;

           if(result==null){

               return true;

           }

       }catch(Exception e){

       }

       sleep(1000);

   }

   return false;}

在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁（这里再多提一句，InnoDB引擎在加锁的时候，只有通过索引进行检索的时候才会使用行级锁，否则会使用表级锁。这里我们希望使用行级锁，就要给method\_name添加索引，值得注意的是，这个索引一定要创建成唯一索引，否则会出现多个重载方法之间无法同时被访问的问题。重载方法的话建议把参数类型也加上）。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。

我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，再通过以下方法解锁：

public void unlock(){

   connection.commit();}

通过connection.commit()操作来释放锁。

这种方法可以有效的解决上面提到的无法释放锁和阻塞锁的问题。

阻塞锁？ for update语句会在执行成功后立即返回，在执行失败时一直处于阻塞状态，直到成功。

锁定之后服务宕机，无法释放？使用这种方式，服务宕机之后数据库会自己把锁释放掉。

但是还是无法直接解决数据库单点和可重入问题。

这里还可能存在另外一个问题，虽然我们对method\_name 使用了唯一索引，并且显示使用for update来使用行级锁。但是，MySql会对查询进行优化，即便在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫效率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 将使用表锁，而不是行锁。如果发生这种情况就悲剧了。

还有一个问题，就是我们要使用排他锁来进行分布式锁的lock，那么一个排他锁长时间不提交，就会占用数据库连接。一旦类似的连接变得多了，就可能把数据库连接池撑爆。

### 概述

数据库行锁来锁住这条数据，这种方案相比排它锁解决了跨进程的问题，但是依然有缺点。

其中一个缺点就是性能问题，在数据库层面会一直阻塞，直到事务提交，这里也是串行执行；

第二个需要注意设置事务的隔离级别是Read Committed，否则并发情况下，另外的事务无法看到提交的数据，依然会导致超卖问题；

缺点三是容易打满数据库连接，如果事务中有第三方接口交互(存在超时的可能性)，会导致这个事务的连接一直阻塞，打满数据库连接。

最后一个缺点，容易产生交叉死锁，如果多个业务的加锁控制不好，就会发生AB两条记录的交叉死锁。

### 方案

1、唯一约束

2、基于数据库来做分布式锁的话，通常有两种做法：

基于数据库的乐观锁（lock in share mode）

基于数据库的悲观锁（select ... for update）

### 特点

#### 优点

直接借助数据库，容易理解。

#### 缺点

存在的问题：

1. 可用性差（这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用），数据库挂掉会导致业务系统不可用；
2. **数据库性能存在瓶颈**，不适合高并发场景；
3. 锁的失效时间难以控制，删除锁失败容易导致死锁。

即这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。

注：对于分布式数据库，可以通过引入超时机制和横向分片减少可能产生死锁和性能瓶颈的情况，提高并发。具体就是，将查询尽可能分发到不同数据节点，这样就减少一个节点上的锁问题。

## Redis锁

### 概述

Redis是单线程的，这里的单线程指的是网络请求模块使用了一个线程（所以不需考虑并发安全性），即一个线程处理所有网络请求，其他模块仍用了多个线程。

### 方案

加锁和解锁的锁必须是同一个，常见的解决方案是**给每个锁一个钥匙（唯一ID），加锁时生成，解锁时判断**。

具体代码：

// 获取锁  
 // NX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

#### redis原子操作

1、基于Redis实现的锁机制，主要是依赖redis自身的原子操作。

##### 加锁

setnx命令加锁，并设置锁的有效时间和持有人标识：

**SET user\_key user\_value NX PX 100**

redis从2.6.12版本开始，SET命令才支持这些参数：

NX：只在在键不存在时，才对键进行设置操作，SET key value NX 效果等同于 SETNX key value

PX millisecond：设置键的过期时间为millisecond毫秒，当超过这个时间后，设置的键会自动失效

**说明：**

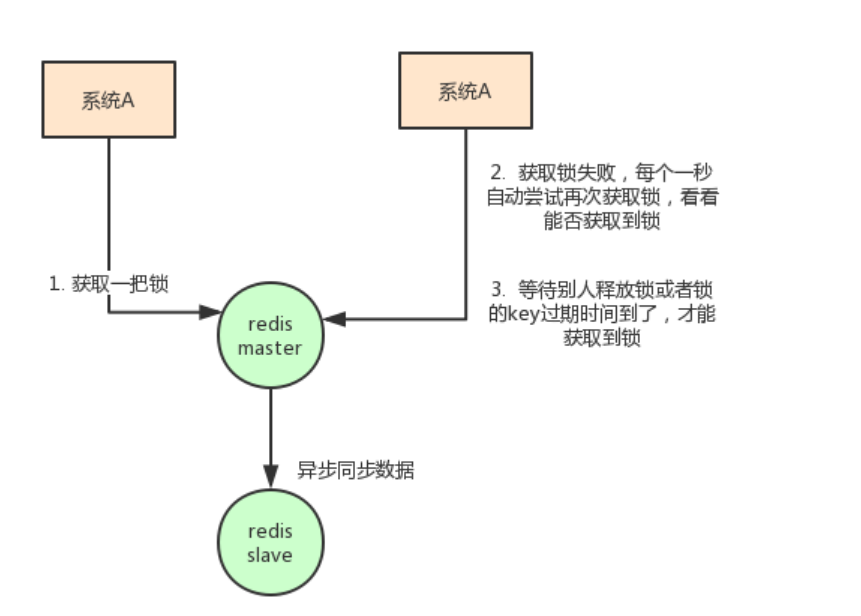
1、一定要用SET key value NX PX milliseconds命令

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)

2、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。

这是避免了一种情况：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。



##### 解锁

检查是否持有锁，然后删除锁

delete values命令删除锁

value具有唯一性，这是避免了一种情况：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。



**为什么这个命令可以帮我们实现锁机制呢？**

因为这个命令是只有在某个key不存在的时候，才会执行成功。那么当多个进程同时并发的去设置同一个key的时候，就永远只会有一个进程成功。

当某个进程设置成功之后，就可以去执行业务逻辑了，等业务逻辑执行完毕之后，再去进行解锁。

解锁很简单，只需要删除这个key就可以了，不过删除之前需要判断，这个key对应的value是当初自己设置的那个。

##### Redlock

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（哨兵）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，其实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：

1、获取当前时间戳，单位是毫秒

2、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

3、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）

4、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

5、要是锁建立失败了，那么就依次删除这个锁

6、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**。



另外，针对redis集群模式的分布式锁，可以采用redis的Redlock机制。

Redlock为了解决单机的问题，需要多个（大于2）redis的master节点，多个master节点互相独立，没有数据同步。

Redlock的实现如下：

1、获取当前时间。

2、依次获取N个节点的锁。每个节点加锁的实现方式同上。这里有个细节，就是每次获取锁的时候的过期时间都不同，需要减去之前获取锁的操作的耗时，

比如传入的锁的过期时间为500ms，

获取第一个节点的锁花了1ms，那么第一个节点的锁的过期时间就是499ms，

获取第二个节点的锁花了2ms，那么第二个节点的锁的过期时间就是497ms

如果锁的过期时间小于等于0了，说明整个获取锁的操作超时了，整个操作失败

1. 判断是否获取锁成功。

如果client在上述步骤中获取到了(N/2 + 1)个节点锁，并且每个锁的过期时间都是大于0的，则获取锁成功，否则失败。失败时释放锁。

1. 释放锁。

对所有节点发送释放锁的指令，每个节点的实现逻辑和上面的简单实现一样。为什么要对所有节点操作？因为分布式场景下从一个节点获取锁失败不代表在那个节点上加锁失败，可能实际上加锁已经成功了，但是返回时因为网络抖动超时了。

#### Redisson

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。

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？如下：

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：

1、redisson所有指令都通过lua脚本执行，redis支持lua脚本原子性执行

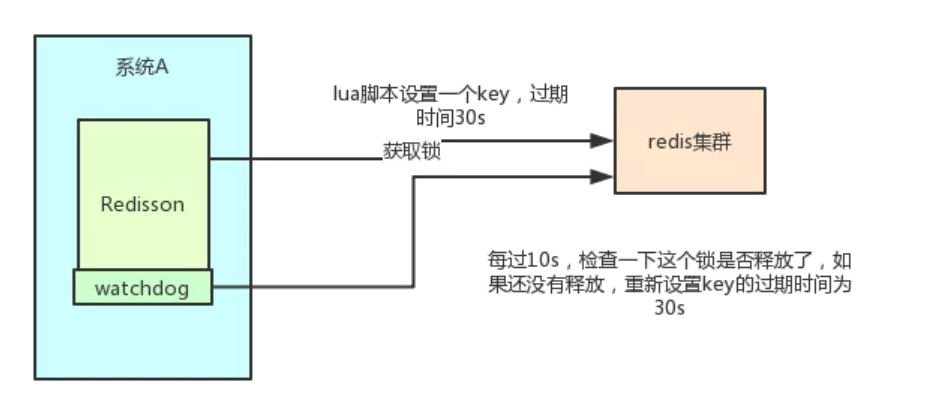
2、redisson设置一个key的默认过期时间为30s,如果某个客户端持有一个锁超过了30s怎么办？

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10秒帮你把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。

3、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)



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();

### 特点

#### 优点

性能好，实现起来较为方便。

#### 缺点

1、单点问题。这里的单点指的是单master，就算是个集群，如果加锁成功后，锁从master复制到slave的时候挂了，也是会出现同一资源被多个client加锁的。

2、执行时间超过了锁的过期时间。为了不出现一直上锁的情况，加了一个兜底的过期时间，时间到了锁自动释放，但是，如果在这期间任务并没有做完怎么办？由于GC或者网络延迟导致的任务时间变长，很难保证任务一定能在锁的过期时间内完成。

通过超时时间来控制锁的失效时间并不是十分的靠谱。

即，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。

注：在实际开发中，引入redis机制，其实就是引入一个新的网元，必然存在集群涉及问题，所以DB锁有其优点的地方就是DB本身具备锁，不需要引入过多的网元。

3、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。

4、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。

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。

### 存在问题

#### 高并发场景下的问题

以下问题不是说在并发不高的场景下不容易出现，只是在高并发场景下出现的概率更高些而已。

**性能问题**

性能问题来自于两个方面：

1. 获取锁的时间上。

如果redlock运用在高并发的场景下，存在N个master节点，一个一个去请求，耗时会比较长，从而影响性能。这个好解决。通过上面描述不难发现，从多个节点获取锁的操作并不是一个同步操作，可以是异步操作，这样可以多个节点同时获取。即使是并行处理的，还是得预估好获取锁的时间，保证锁的TTL > 获取锁的时间+任务处理时间。

2、被加锁的资源太大。加锁的方案本身就是会为了正确性而牺牲并发的，牺牲和资源大小成正比。这个时候可以考虑对资源做拆分，拆分的方式有两种：

3、从业务上将锁住的资源拆分成多段，每段分开加锁。比如，我要对一个商户做若干个操作，操作前要锁住这个商户，这时我可以将若干个操作拆成多个独立的步骤分开加锁，提高并发。

4、用分桶的思想，将一个资源拆分成多个桶，一个加锁失败立即尝试下一个。比如批量任务处理的场景，要处理200w个商户的任务，为了提高处理速度，用多个线程，每个线程取100个商户处理，就得给这100个商户加锁，如果不加处理，很难保证同一时刻两个线程加锁的商户没有重叠，这时可以按一个维度，比如某个标签，对商户进行分桶，然后一个任务处理一个分桶，处理完这个分桶再处理下一个分桶，减少竞争。

**重试的问题**

无论是简单实现还是redlock实现，都会有重试的逻辑。如果直接按上面的算法实现，是会存在多个client几乎在同一时刻获取同一个锁，然后每个client都锁住了部分节点，但是没有一个client获取大多数节点的情况。解决的方案也很常见，在重试的时候让多个节点错开，错开的方式就是在重试时间中加一个随机时间。这样并不能根治这个问题，但是可以有效缓解问题，亲试有效。

#### 节点宕机

对于单master节点且没有做持久化的场景，宕机就挂了，这个就必须在实现上支持重复操作，自己做好幂等。

对于多master的场景，比如redlock，我们来看这样一个场景：

1、假设有5个redis的节点：A、B、C、D、E，没有做持久化。

2、client1从A、B、C 3个节点获取锁成功，那么client1获取锁成功。

3、节点C挂了。

4、client2从C、D、E获取锁成功，client2也获取锁成功，那么在同一时刻client1和client2同时获取锁，redlock被玩坏了。

怎么解决呢？最容易想到的方案是打开持久化。持久化可以做到持久化每一条redis命令，但这对性能影响会很大，一般不会采用，如果不采用这种方式，在节点挂的时候肯定会损失小部分的数据，可能我们的锁就在其中。

另一个方案是延迟启动。就是一个节点挂了修复后，不立即加入，而是等待一段时间再加入，等待时间要大于宕机那一刻所有锁的最大TTL。

但这个方案依然不能解决问题，如果在上述步骤3中B和C都挂了呢，那么只剩A、D、E三个节点，从D和E获取锁成功就可以了，还是会出问题。那么只能增加master节点的总量，缓解这个问题了。增加master节点会提高稳定性，但是也增加了成本，需要在两者之间权衡。

#### 任务执行时间超过锁的TTL

之前产线上出现过因为网络延迟导致任务的执行时间远超预期，锁过期，被多个线程执行的情况。

这个问题是所有分布式锁都要面临的问题，包括基于zookeeper和DB实现的分布式锁，这是锁过期了和client不知道锁过期了之间的矛盾。

在加锁的时候，我们一般都会给一个锁的TTL，这是为了防止加锁后client宕机，锁无法被释放的问题。但是所有这种姿势的用法都会面临同一个问题，就是没发保证client的执行时间一定小于锁的TTL。虽然大多数程序员都会乐观的认为这种情况不可能发生，我也曾经这么认为，直到被现实一次又一次的打脸。

Martin Kleppmann也质疑过这一点，这里直接用他的图：



1、Client1获取到锁

2、Client1开始任务，然后发生了STW的GC，时间超过了锁的过期时间

3、Client2 获取到锁，开始了任务

4、Client1的GC结束，继续任务，这个时候Client1和Client2都认为自己获取了锁，都会处理任务，从而发生错误。

Martin Kleppmann举的是GC的例子，我碰到的是网络延迟的情况。不管是哪种情况，不可否认的是这种情况无法避免，一旦出现很容易懵逼。

如何解决呢？一种解决方案是不设置TTL，而是在获取锁成功后，给锁加一个watchdog，watchdog会起一个定时任务，在锁没有被释放且快要过期的时候会续期。

这里也提下Martin Kleppmann的解决方案，他的方案是让加锁的资源自己维护一套保证不会因加锁失败而导致多个client在同一时刻访问同一个资源的情况。



在客户端获取锁的同时，也获取到一个资源的token，这个token是单调递增的，每次在写资源时，都检查当前的token是否是较老的token，如果是就不让写。对于上面的场景，Client1获取锁的同时分配一个33的token，Client2获取锁的时候分配一个34的token，在client1 GC期间，Client2已经写了资源，这时最大的token就是34了，client1 从GC中回来，再带着33的token写资源时，会因为token过期被拒绝。这种做法需要资源那一边提供一个token生成器。

对于这种fencing的方案，我有几点问题：

1、无法保证事务。示意图中画的只有34访问了storage，但是在实际场景中，可能出现在一个任务内多次访问storage的情况，而且必须是原子的。如果client1带着33token在GC前访问过一次storage，然后发生了GC。client2获取到锁，带着34的token也访问了storage，这时两个client写入的数据是否还能保证数据正确？如果不能，那么这种方案就有缺陷，除非storage自己有其他机制可以保证，比如事务机制；如果能，那么这里的token就是多余的，fencing的方案就是多此一举。

2、高并发场景不实用。因为每次只有最大的token能写，这样storage的访问就是线性的，在高并发场景下，这种方式会极大的限制吞吐量，而分布式锁也大多是在这种场景下用的，很矛盾的设计。

3、这是所有分布式锁的问题。这个方案是一个通用的方案，可以和Redlock用，也可以和其他的lock用。所以我理解仅仅是一个和Redlock无关的解决方案。

#### 系统时钟漂移

理论上是可能出现的，实际应用中不一定会出现。

redis的过期时间是依赖系统时钟的，如果时钟漂移过大时会影响到过期时间的计算。

**为什么系统时钟会存在漂移呢？**

先简单说下系统时间，linux提供了两个系统时间：clock realtime和clock monotonic。clock realtime也就是xtime/wall time，这个时间时可以被用户改变的，被NTP改变，gettimeofday拿的就是这个时间，redis的过期计算用的也是这个时间。

clock monotonic，直译过来时单调时间，不会被用户改变，但是会被NTP改变。

最理想的情况时，所有系统的时钟都时时刻刻和NTP服务器保持同步，但这显然时不可能的。导致系统时钟漂移的原因有两个：

1、系统的时钟和NTP服务器不同步，这个目前没有特别好的解决方案。

2、clock realtime被人为修改。在实现分布式锁时，不要使用clock realtime。

## Zookeeper分布式锁

### 概述

Zookeeper是一种提供配置管理、分布式协同以及命名的中心化服务。

#### 节点

zk的模型是这样的：zk包含一系列的节点，叫做znode。

ZooKeeper可以创建4种类型的节点，分别是：

持久性节点

持久性顺序节点

临时性节点

临时性顺序节点

**持久性节点和临时性节点的区别：**

持久性节点表示只要你创建了这个节点，那不管你ZooKeeper的客户端是否断开连接，ZooKeeper的服务端都会记录这个节点；

临时性节点刚好相反，一旦你ZooKeeper客户端断开了连接，那ZooKeeper服务端就不再保存这个节点；

顺便也说下顺序性节点，顺序性节点是指，在创建节点的时候，ZooKeeper会自动给节点编号比如0000001，0000002这种的。

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；

zookeeper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zookeeper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，zookeeper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，zookeeper会通知客户端。当前zookeeper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

其实基于ZooKeeper，就是使用它的临时有序节点来实现的分布式锁。

注：zookeeper基于Paxos协议改造的ZAB协议实现选举。

#### 特点

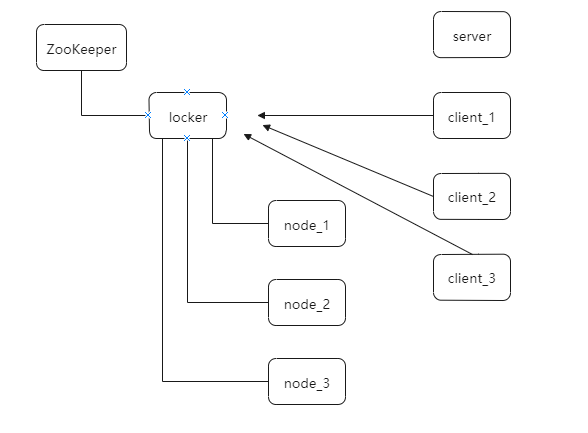
1. 统一视图（多个客户端连接访问不同节点，数据保持一致）；
2. Zookeeper可以存储数据；
3. Zookeeper目录类型分为4种：
4. 持久化目录：客户端断开连接后不会删除
5. 临时目录：客户端断开连接后自动删除
6. 持久有顺序目录
7. 临时有顺序目录
8. Zookeeper支持事件回调

zk节点目录下的文件会注册一个事件，如果该事件发生，则会调用客户端的回调函数执行相应的操作。

### 原理

当某客户端要进行逻辑的加锁时，就在zookeeper上的某个指定节点的目录下，去生成一个唯一的临时有序节点，然后判断自己是否是这些有序节点中序号最小的一个，如果是，则算是获取了锁。如果不是，则说明没有获取到锁，那么就需要在序列中找到比自己小的那个节点，并对其调用exist()方法，对其注册事件监听，当监听到这个节点被删除了，那就再去判断一次自己当初创建的节点是否变成了序列中最小的。如果是，则获取锁，如果不是，则重复上述步骤。

当释放锁的时候，只需将这个临时节点删除即可。



如图，locker是一个持久节点，node\_1/node\_2/…/node\_n 就是上面说的临时节点，由客户端client去创建的。

client\_1/client\_2/…/clien\_n 都是想去获取锁的客户端。以client\_1为例，它想去获取分布式锁，则需要跑到locker下面去创建临时节点（假如是node\_1）创建完毕后，看一下自己的节点序号是否是locker下面最小的，如果是，则获取了锁。如果不是，则去找到比自己小的那个节点（假如是node\_2），找到后，就监听node\_2，直到node\_2被删除，那么就开始再次判断自己的node\_1是不是序列中最小的，如果是，则获取锁，如果还不是，则继续找一下一个节点。

### 方案

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：

1、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。

2、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

3、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。

4、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：



### 特点

#### 优点

有效的解决单点问题，不可重入问题，非阻塞问题以及锁无法释放的问题。实现起来较为简单。

**锁无法释放？**

使用Zookeeper可以有效的解决锁无法释放的问题，因为在创建锁的时候，客户端会在ZK中创建一个临时节点，一旦客户端获取到锁之后突然挂掉（Session连接断开），那么这个临时节点就会自动删除掉。其他客户端就可以再次获得锁。

**非阻塞锁？**

使用Zookeeper可以实现阻塞的锁，客户端可以通过在ZK中创建顺序节点，并且在节点上绑定监听器，一旦节点有变化，Zookeeper会通知客户端，客户端可以检查自己创建的节点是不是当前所有节点中序号最小的，如果是，那么自己就获取到锁，便可以执行业务逻辑了。

**不可重入？**

使用Zookeeper也可以有效的解决不可重入的问题，客户端在创建节点的时候，把当前客户端的主机信息和线程信息直接写入到节点中，下次想要获取锁的时候和当前最小的节点中的数据比对一下就可以了。如果和自己的信息一样，那么自己直接获取到锁，如果不一样就再创建一个临时的顺序节点，参与排队。

**单点问题？**

使用Zookeeper可以有效的解决单点问题，ZK是集群部署的，只要集群中有半数以上的机器存活，就可以对外提供服务。

#### 缺点

**性能上不如使用缓存实现分布式锁**。需要对ZK的原理有所了解。

Zookeeper实现的分布式锁其实存在一个缺点，那就是性能上可能并没有缓存服务那么高。因为每次在创建锁和释放锁的过程中，都要动态创建、销毁瞬时节点来实现锁功能。ZK中创建和删除节点只能通过Leader服务器来执行，然后将数据同不到所有的Follower机器上。

其实，使用Zookeeper也有可能带来并发问题，只是并不常见而已。考虑这样的情况，由于网络抖动，客户端可ZK集群的session连接断了，那么zk以为客户端挂了，就会删除临时节点，这时候其他客户端就可以获取到分布式锁了。就可能产生并发问题。这个问题不常见是因为zk有重试机制，一旦zk集群检测不到客户端的心跳，就会重试，Curator客户端支持多种重试策略。多次重试之后还不行的话才会删除临时节点。（所以，选择一个合适的重试策略也比较重要，要在锁的粒度和并发之间找一个平衡）

对于zk分布式锁而言:

zookeeper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**。

# 总结

## 对比

基于数据库实现分布式锁：

1. 性能较差，容易出现单点故障；
2. 锁没有失效时间，容易死锁；
3. 非阻塞式的；
4. 不可重入。

基于缓存实现分布式锁：

1. 锁没有失效时间，容易死锁；
2. 非阻塞式的；
3. 不可重入。

基于Zookeeper实现分布式锁：

1. 实现相对可靠；
2. 可靠性高；
3. 性能较好。

1、从理解的难易程度角度（从低到高）

数据库 > 缓存 > Zookeeper

2、从实现的复杂性角度（从低到高）

Zookeeper >= 缓存 > 数据库

3、从性能角度（从高到低）

缓存 > Zookeeper >= 数据库

4、从可靠性角度（从高到低）

Zookeeper > 缓存 > 数据库

## 选择

综上所述：

如果系统不想引入过多网元，可以采用数据库锁实现，好处就是比较容易理解，但是这种方案业务层控制逻辑多且复杂，需要对业务侧足够了解，易于理解但是实现复杂度最高。

如果追求高性能，Redis是最佳选择，但是**redis是有可能存在隐患的，可能会导致数据不对的情况**，可靠性不如ZK。

如果系统已经存在ZK集群，优先选用ZK实现，实现最简单，且可以提供高可靠性，性能稍逊Redis缓存方案。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。

# 分布式数据库应用

分布式数据库采用DB锁，但是针对DB锁的缺点做了优化：

1. 数据库挂掉会导致业务系统不可用->主从复制
2. 数据库性能存在瓶颈，不适合高并发场景->将数据做水平分片
3. 锁的失效时间难以控制，删除锁失败容易导致死锁->引入超时