# 分布式锁

在很多场景中，我们为了保证数据的最终一致性，需要很多的技术方案来支持，比如分布式事务、分布式锁等。有的时候，我们需要保证一个方法在同一时间内只能被同一个线程执行。在单机环境中，Java中其实提供了很多并发处理相关的API，但是这些API在分布式场景中就无能为力了。也就是说单纯的Java Api并不能提供分布式锁的能力。所以针对分布式锁的实现目前有多种方案。

在分析这几种实现方案之前我们先来想一下，我们需要的分布式锁应该是怎么样的？

1. 可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。
2. 这把锁要是一把可重入锁（避免死锁）
3. 这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）
4. 有高可用的获取锁和释放锁功能，获取锁和释放锁的性能要好。

## 基于数据库实现分布式锁

### 基于数据库表

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。



当我们想要锁住某个方法时，执行以下SQL：



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:



**存在的问题：**

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。

**解决方法：**

1. 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。
2. 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。
3. 搞一个while循环，直到insert成功再返回成功。
4. 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。

### 基于数据库排他锁

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：



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：  
  
**优点：**

1. 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。
2. 使用这种方式，服务宕机之后数据库会自己把锁释放掉。

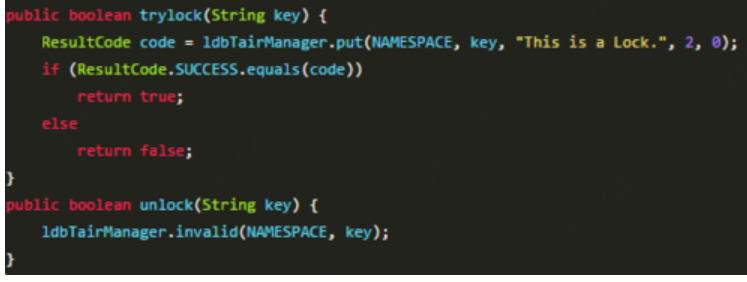
**缺点：**

无法解决数据库单点和可重入问题。

## 基于缓存实现分布式锁

相比较于基于数据库实现分布式锁的方案来说，基于缓存来实现在性能方面会表现的更好一点。而且很多缓存是可以集群部署的，可以解决单点问题。

基于Tair的实现分布式锁其实和Redis类似，其中主要的实现方式是使用TairManager.put方法来实现。



**存在的问题：**

1. 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tair中，其他线程无法再获得到锁。
2. 这把锁只能是非阻塞的，无论成功还是失败都直接返回。
3. 把锁是非重入的，一个线程获得锁之后，在释放锁之前，无法再次获得该锁，因为使用到的key在tair中已经存在。无法再执行put操作。

**解决方案：**

1. tair的put方法支持传入失效时间，到达时间之后数据会自动删除。
2. while重复执行。
3. 在一个线程获取到锁之后，把当前主机信息和线程信息保存起来，下次再获取之前先检查自己是不是当前锁的拥有者。

## 基于Zookeeper实现分布式锁

基于zookeeper临时有序节点可以实现的分布式锁。

大致思想即为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

**优点：**

1. 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。
2. 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。
3. 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。
4. 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：**

使用ZK实现的分布式锁好像完全符合了本文开头我们对一个分布式锁的所有期望。但是，其实并不是，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。

**三种方案比较**