**分布式锁**

# 分布式锁要点

* 互斥性

在并发情况下，只有一个client能够获取到锁

* 原子性

加锁操作必须是原子操作，否则可能会有多个client同时获取到锁

* 无死锁

Client在处理业务结束后主动释放锁，若处理过程中出现异常未能主动释放锁，需要系统能够主动释放锁，保证不会出现死锁

* 可重入

同一个client是否在获得锁后继续获得锁

* 阻塞性

加锁的操作是否是阻塞的

# 数据库实现

## 2.1.基于数据库表

创建一张锁表，当我们要锁住某个方法或资源时，只要在表中增加一条记录，释放锁的时候删除这条记录，例如我们要锁住某个方法，可以在表中设置方法名唯一，这样数据库将保证只有一个操作可以成功，我们即可认为线程获得了该方法的锁。

可能出现的问题：

1. 锁依赖于数据库的可用性，数据库是单点，一旦数据库挂掉
2. ，业务系统不可用，可以利用主从复制实现容灾
3. 锁没有失效时间，一旦解锁失败，就会导致锁记录一直存在数据库中，其他线程无法获取，可以做一个定时任务，每隔一段时间将数据库的超时数据进行清理
4. 非阻塞锁，一旦插入失败即返回，没有获得锁并不会进入排队队列，必须再次触发加锁操作，可以通过while循环直至insert成功。
5. 锁是非重入的，同一个线程释放锁之前无法再次获得该锁，因为数据库中已经存在数据，可以在数据库表中加入线程字段，下次获取的时候如果获取锁的是当前线程，则直接分配锁

## 2.2.基于数据库的排他锁

可以将需要互斥的资源写入数据库表中，通过select for update实现给数据加排他锁，这里需要注意的是指定的数据要为唯一索引，innodb在加行锁的时候，是通过索引，否则会退化为表锁。

这样，当某条记录被加上排他锁后，其他线程无法在该记录上添加排他锁。

* 排他锁解决了无法释放问题，一旦服务宕机，则数据库自动将锁释放
* 阻塞锁，for update会在执行成功后返回，如果执行失败则处于阻塞状态，直到成功。
* 依然存在单点问题
* 可重入问题无法解决
* 存储引擎在数据库表很小时，即使设置了索引检索，但可能会因为全表扫描更快而加上表锁，这样行锁失效

# 缓存实现

基于缓存的分布式锁在性能方面会更好一点，缓存多为集群部署，可以解决单点问题。

## 3.1.redis

Jedis的加锁代码：jedis.set(key,value,nx/xx,ex/px,time)：

* Key：使用key当锁，因为key是唯一的
* Value：即requestID，唯一标识加锁的client，避免其他client误删锁
* Nx/xx：nx表示set if not exist，xx表示存在设置key
* Ex/px：ex表示秒，px表示毫秒
* Time：代表key的过期时间

Set()方法的结果是当没有锁（key不存在），进行加锁操作，并设置有效期，如果锁存在，不做任何操作。

其优点如下：

1. set if not exist保证最终只有一个client加锁成功
2. 过期时间保证即使持有者故障未释放锁，锁也会因为过期时间自动解锁
3. requestID保证了解锁的是加锁的client，避免了误删锁的情况
4. 将setn和设置过期时间放到一个函数中保证了操作的原子性，如果先set后加过期时间，那么在加锁后宕机会导致死锁

## memcached

add操作增加新缓存，如果key已经存在则调用失败。

Cas类似数据库中的乐观锁，通过比较key对应value变化检测是否获取到锁

# zookeeper实现

1. 在zookeeper指定节点locks下创建临时顺序节点
2. 获取locks下所有子节点
3. 对子节点按自增序号从小到大排序
4. 最小的子节点获得锁，若当前节点不是最小，则监听比自己小的最大节点的删除事件

其优点如下：

* 由于加锁时创建临时节点，当client没有释放锁而故障，一段时间内zookeeper没有收到client的心跳判断会话失效，临时节点将被删除
* 避免了惊群问题，如果当前1000个节点在等待锁，获得锁的client释放锁后，会唤醒所有等待的client，但最终只有一个client获得锁，zookeeper通过监听比自己小但最大的节点删除事件
* 所有集群中的节点都将锁的持有者信息写入，解决了单点问题
* 锁可重入

缺点：

* 性能没有缓存那么高，因为每次创建锁和释放锁都需要动态创建销毁临时节点实现锁功能，zookeeper中的创建和删除节点只能通过leader来实现，将数据写到不同的follower节点。