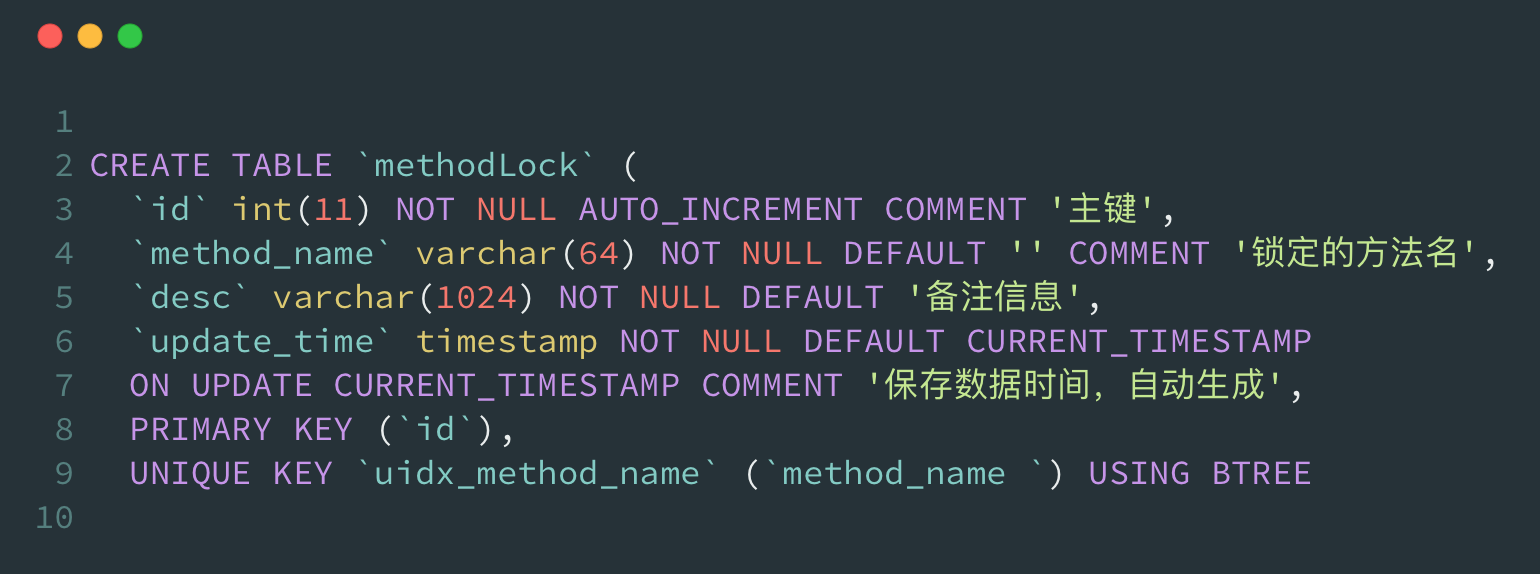
**分布式锁的实现方法：**

1. 数据库实现：

基于数据库表，直接创建一张锁表

对方法名做唯一性约束，如果有多个请求同时提交到数据库，数据库会保证只有一个操作可以成功。

问题：

1. 这个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。 解决办法：搞多个数据库做同步
2. 这个锁没有实效性，一旦解锁操作失败，其他线程无法再获得锁 解决方法：可以定一个定时任务，每隔一段时间把数据库中超时锁清除
3. 这把锁是再非阻塞的，一旦插入失败会直接报错 解决办法：while循环
4. 非重入 解决方法：在数据库加一个字段，把主机信息和线程信息保存，下次获取锁的时候先查询，如果可以查到直接获取

基于数据库的排他锁来实现：

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，当某条记录被加上排他锁后，其他线程无法在该行记录上增加排他锁，执行完通过connection.commit()方法释放锁

悲观锁 VS 乐观锁

乐观锁使用版本号实现 重试机制 悲观锁实现方式独占数据，其他线程需要等待

1. 基于redis实现

基于redis实现性能会好一些 而且缓存可以集群部署，可以解决单点问题

jedis.set(String key, String value, String nxxx, String expx, int time)

key来当锁，因为key是唯一的，value是requestId用于解锁，nxxx为当key不存在时进行set操作，若key已经存在不进行任何操作，expx为过期设置，time为过期时间。必须一条指令，如果使用jedis.set和jedis.expire不具备原子性

1. 基于ZooKeeper实现

Zookeeper的节点有4种类型：

1. 持久节点：默认的节点类型，创建的节点的客户端与zookeeper断开连接后，该节点依旧存在
2. 持久节点顺序节点：在创建节点时，zookeeper根据创建的时间顺序给该几点名称编号
3. 临时节点：当创建节点的客户端与zookeeper断开连接后，临时节点会被删除
4. 临时顺序节点：结合临时节点和顺序节点的特点

实现原理，首先先创建一个持久节点ParentLock，当一个客户端想要获得锁的时候，在ParentLock节点下面创建一个临时节点lock1，然后他判断自己创建的节点是不是最靠前的，如果是第一个节点则获得锁，最后形成一个等待队列。

分布式事务

CAP：C一致性 A可用性 P分区容错性

解决方案：

XA协议（两段式提交）：1.事务管理器要求每个涉及到事务的数据库预提交此操作，并反映是否可以提交

2.事务管理器要求每个数据库提交数据或者回滚数据

优点：尽量保证了数据的强一致性，实现成本较低

缺点：事务管理器的单点问题；同步阻塞；数据不一致问题

TCC事务机制：

Try阶段：尝试执行，完成所有业务检查，预留必须业务资源

Confirm阶段：确认真正执行业务，不作任何业务检查，只使用Try阶段预留的业务资源

Cancel阶段：取消执行，释放Try阶段预留的业务资源

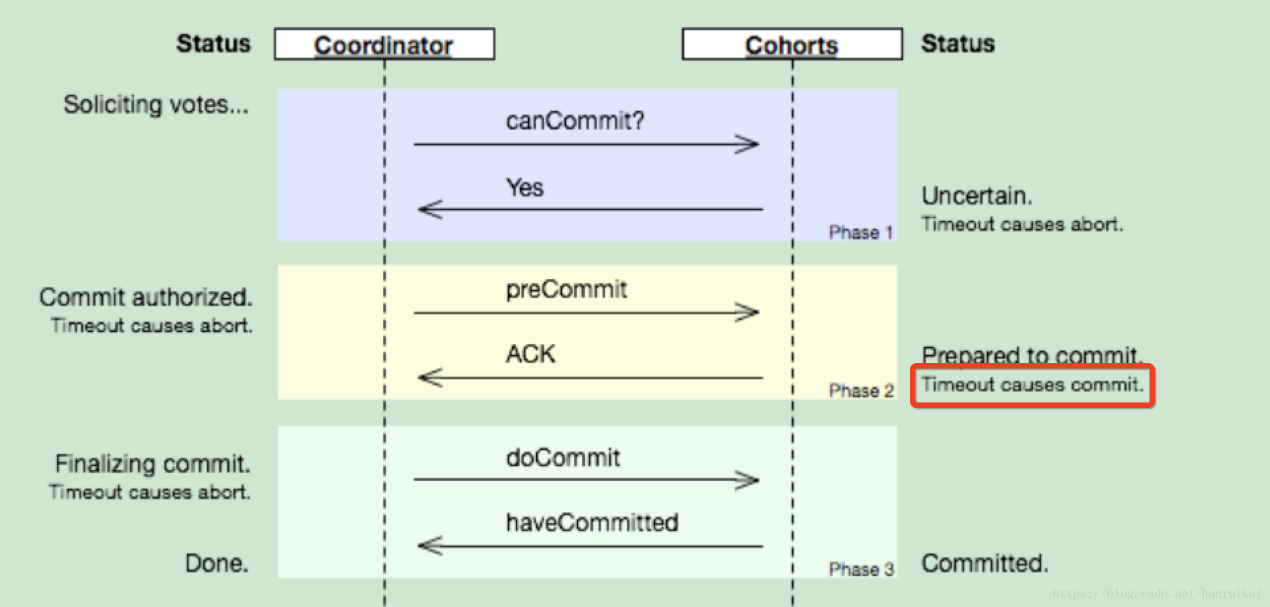
本地消息表：

此方案的核心是将需要分布式处理的任务通过消息日志的方式来异步执行。消息日志可以存储到消息队列里，通过业务规则自动发起

两段式优点：较强的一致性，当然并不是100%

缺点：耗时过长，锁定资源过长，同步阻塞，准备阶段回复后要一直等待；协调者可能出现单点故障

三段式：引入超时和准备阶段，如果因为网络问题没有接受到docommit请求，超时后默认都执行doCommit请求



优点：降低了阻塞范围

缺陷：脑裂问题依然存在，即在参与者搜到PreCommit请求后等待最终指令，如果协调者此时无法与参与者正常通信，会导致参与者继续提交事务，造成数据不一致。

TCC模式-本质也是2PC，改进：通过预留资源的方式避免了同步阻塞资源的情况，但是TCC变成需要业务自己实现try，confirm和cancle方法，对业务入侵很大，实现很复杂。

Sega模式：采用事务补偿机制，每个本地事务都存储一个副本，如果出现失败，则利用补偿机制回滚

本地消息表：通过事务消息异步地保证消息最终一定会被消费成功。

注意点：至少一次消息成功传递，需要持久化；消息去重重复，使用幂等消息