1 分布式事务;

1-0: 特性: 原子性/一致性/持久性/隔离性;

1-1产生原因: 数据库分库分表; 应用的SOA化;

1-2分布式理论: CAP理论: 一致性/可用性/分区容错性 ,一个web应用最多同时支持两个属性;

Base理论: 基本可用/软状态/最终一致性; 无法做到强一致性,可用结合自身业务特点,使系统达到最终一致性;

1-3: 解决方案:

两阶段提交(2PC); 补偿事务(TCC); 本地消息表(异步确保); MQ事务消息(RocketMQ)

1-4: 实际项目使用: 本地消息表(将分布式事务拆分成本地事务 + 消息表)

优点: 避免了分布式事务,实现了最终一致性;

缺点:业务耦合,频繁轮询数据库,增大了数据库负载,不适合高并发场景,中等规模还是可以满足的呢;

1-5: 详细使用过程;

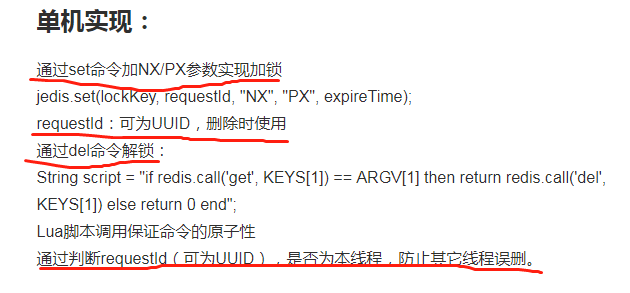
2 悲观锁/乐观锁; ( 使用定时任务扫描消息表---用到乐观锁;) <https://blog.csdn.net/qq_34337272/article/details/81072874>

**悲观锁(排它锁)**:每次拿数据的时候都认为会修改,eg: mysql中行锁/表锁/读锁/写锁; Java中Synchronized和ReentrantLock;**多写的场景**

**乐观锁**:每次拿数据的时候认为别人不会修改,所以不会上锁,在更新的时候会判断下别人有没有去更新这个数据,使用版本号和CAS算法实现; 适用于: **多读的应用类型**,可以提高吞吐量;

解决方式: 表中添加Version版本字段/TimeStamp时间戳---看看实际怎么操作?

3分布式锁---参考: <https://www.jianshu.com/p/da41fe072d11>

**基于Redis;**

基于数据库;

基于Zookeeper;

4 接口幂等性;

幂等性值同一个操作无论请求多少次,其结果都相同;

操作方式: 1 操作之前在业务方法进行判断, 如果执行过了就不再执行;

2 缓存所有请求和处理结果,已经处理的请求则直接返回结果;

3 在数据库表中加一个状态字段(未处理/已处理),数据操作时判断未处理时再处理;