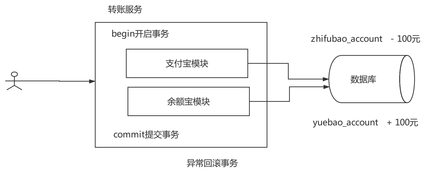
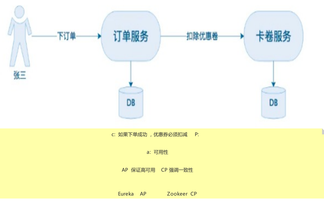


本地事务是关系型数据库中，由一组SQL组成的一个执行单元，该单元要么整体成功，要么整体失败。它的缺点就是：仅支持单库事务，并不支持跨库事务。

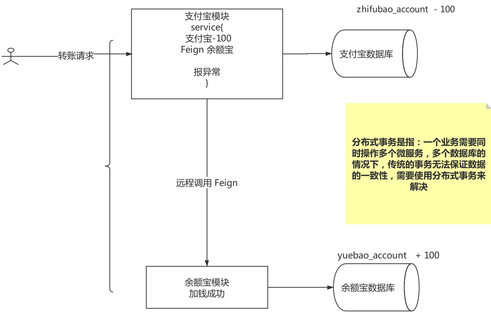


C：如果下单成功，优惠券必须扣减 P：

A：可用性

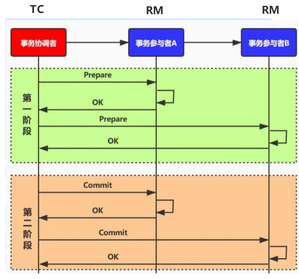
AP 保证高可用 CP 强调一致性

Eureka AP Zookeeer CP



分布式事务是指：一个业务需要同时操作多个微服务，多个数据库的情况下，传统的事务无法保证数据的一致性，需要使用分布式事务来解决

基于XA接口的二阶段提交

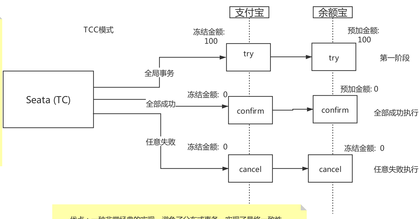


优点：尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）

缺点：实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景。

优点：相比两阶段提交，可用性
不依赖关系型数据库

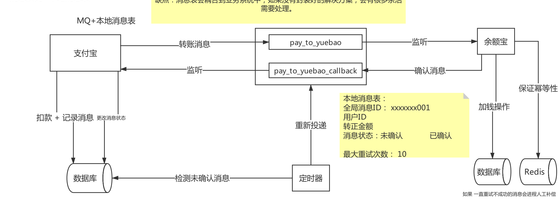
缺点：数据的一致性要差一些。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

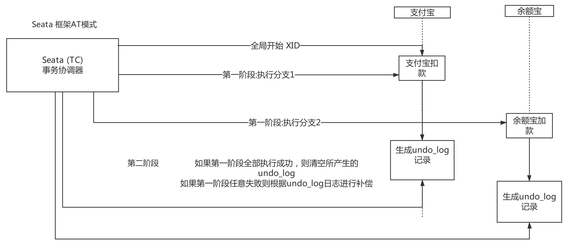


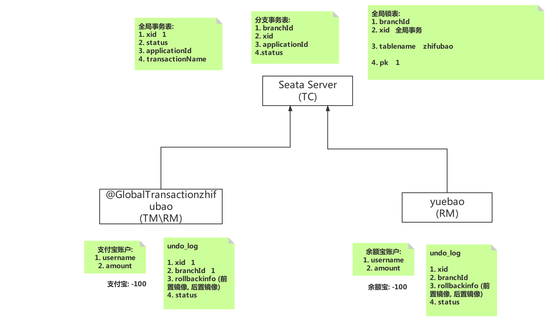
优点：一种非常经典的实现，避免了分布式事务，实现了最终一致性。

缺点：消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

消息队列+本地消息表







AT模式的工作原理？（工作流程）
TC(Seata Server): 需要单独启动事务管理器

在我们的项目中，每个微服务都会连接这个事务管理器

在需要管理事务的方法中加上注解实现分布式事务
@GlobalTransaction
(全局事务的注解)

全局事务方法被调用后
一阶段:
开启一个全局事务，每一个要执行微服务事务方法都会注册一个分支事务，每个分支事务的执行，会将我们自己的sql执行，并且根据sql生成一条undo_log,将分支事务提交，数据库数据被正式提交。

二阶段:
如果全部的分支事务都执行成功，那么第二阶段，异步删除一阶段所生成的undo日志

如果第一阶段有任意一个分支事务执行失败，那么二阶段统一执行回滚操作，根据undo日志可以生成反向sql
如: 插入一条id等于1的数据，那么undo日志可以生成一条删除id等于1的数据

反向SQL

```
insert into order values (*****)  --> orderId = 1234
delete from order where orderId = 1234
update account set balance=900 where userId=1111
update account set balance=1000  where userId=1111
delete from order where orderId = 1234
insert into order values (*****)  --> orderId = 1234
```

AT模式如何进行数据的回滚?
(需要参照官网)

利用undo_log回滚日志来进行回滚，如果手动的修改相关数据可能会造成回滚失败，回滚失败会根据配置策略进行重试回滚

优点: 基于undo日志保证数据的一致
效率由于二阶段提交，整合方案简单

缺点: 如果和其它的异步方案相比效率不高
只支持使用JDBC访问数据
只支持支持ACID的关系型数据库
默认是读未提交，如果如果改成读已提交因为开启全局锁性能会下降

如何保证Seata的高可用?

可以通过nacos搭建注册中心，及配置中心，
搭建多台Seata形成高可用集群，将事务数据存储到mysql数据库中

(全局事务表
分支事务表
全局锁表)

AT模式事务的隔离级别?

读未提交

有可能会出现脏读?

解决方案: 在select的语句后面加上 for update, 实现读已提交
(利用全局锁实现读已提交)

Seata AT模式的介绍?

Seata是阿里提出的开源的分布式事务框架，提供了多种分布式事务解决方案，现在支持 XA模式、AT模式、TCC模式、Saga模式 我们主要选择的是AT模式，AT模式(Atomic Transaction)原子事务,基于二阶段提交的思路保证了数据的一致性，而且整合相对简单，代码改动量不大 效率也可以接受