**畅购电商系统开发**

**第15天**

传智播客.黑马程序员.深圳

# 学习目标

* 理解什么是事务
* 理解什么是分布式事务
* 理解CAP定理
* 能说出相关的分布式事务解决方案
* 理解Seata工作流程
* 能实现Seata案例

# 分布式事务介绍

## 什么是事务

数据库事务(简称：事务，Transaction)是指数据库执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。

事务拥有以下四个特性，习惯上被称为**ACID**特性：

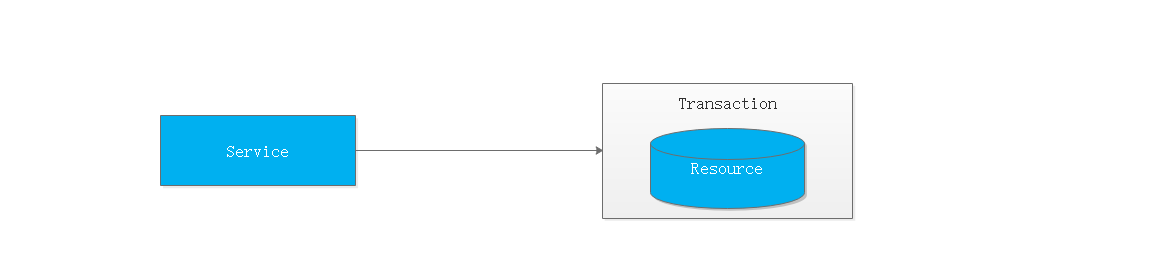
* **原子性**(Atomicity)：整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
* **一致性**(Consistency)：一个事务可以封装状态改变（除非它是一个只读的）。事务必须始终保持系统处于一致的状态，不管在任何给定的时间并发事务有多少。

举例：如果事务是并发多个，系统也必须如同串行事务一样操作。其主要特征是保护性和不变性(Preserving an Invariant)，以转账案例为例，假设有五个账户，每个账户余额是100元，那么五个账户总额是500元，如果在这个5个账户之间同时发生多个转账，无论并发多少个，比如在A与B账户之间转账5元，在C与D账户之间转账10元，在B与E之间转账15元，五个账户总额也应该还是500元，这就是保护性和不变性。

* **隔离性**(Isolation)：多个事务并发执行时，一个事务的执行不应影响其他事务的执行，如同只有这一个操作在被数据库所执行一样。
* **持久性**(Durability)：已被提交的事务对数据库的修改应该永久保存在数据库中。在事务结束时，此操作将不可逆转。

## 本地事务

起初，事务仅限于对单一数据库资源的访问控制,架构服务化以后，事务的概念延伸到了服务中。倘若将一个单一的服务操作作为一个事务，那么整个服务操作只能涉及一个单一的数据库资源,这类基于单个服务单一数据库资源访问的事务，被称为本地事务(Local Transaction)。



## 什么是分布式事务

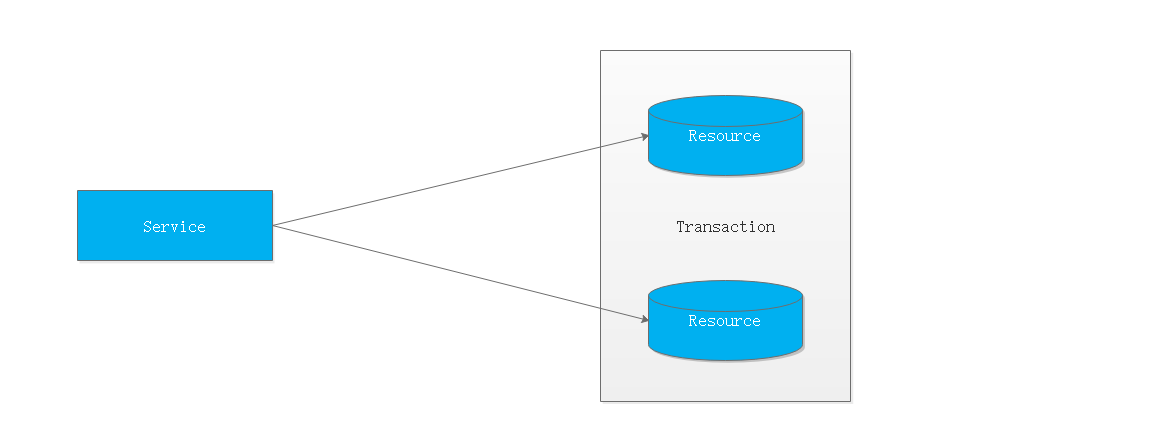
分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上,且属于不同的应用，分布式事务需要保证这些操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

## 分布式事务应用架构

本地事务主要限制在单个会话内，不涉及多个数据库资源。但是在基于SOA(Service-Oriented Architecture，面向服务架构)的分布式应用环境下，越来越多的应用要求对多个数据库资源，多个服务的访问都能纳入到同一个事务当中，分布式事务应运而生。

### 单一服务分布式事务

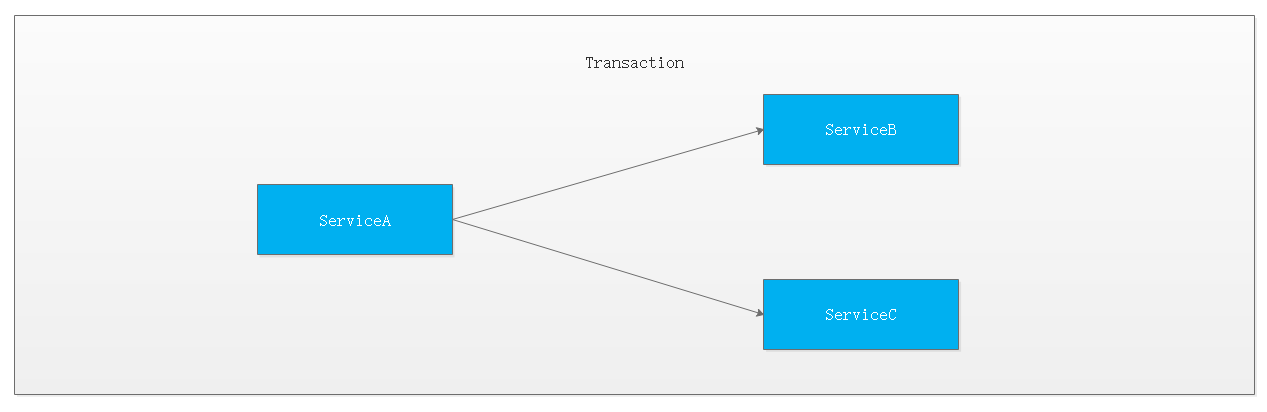
最早的分布式事务应用架构很简单，不涉及服务间的访问调用，仅仅是服务内操作涉及到对多个数据库资源的访问。



### 多服务分布式事务

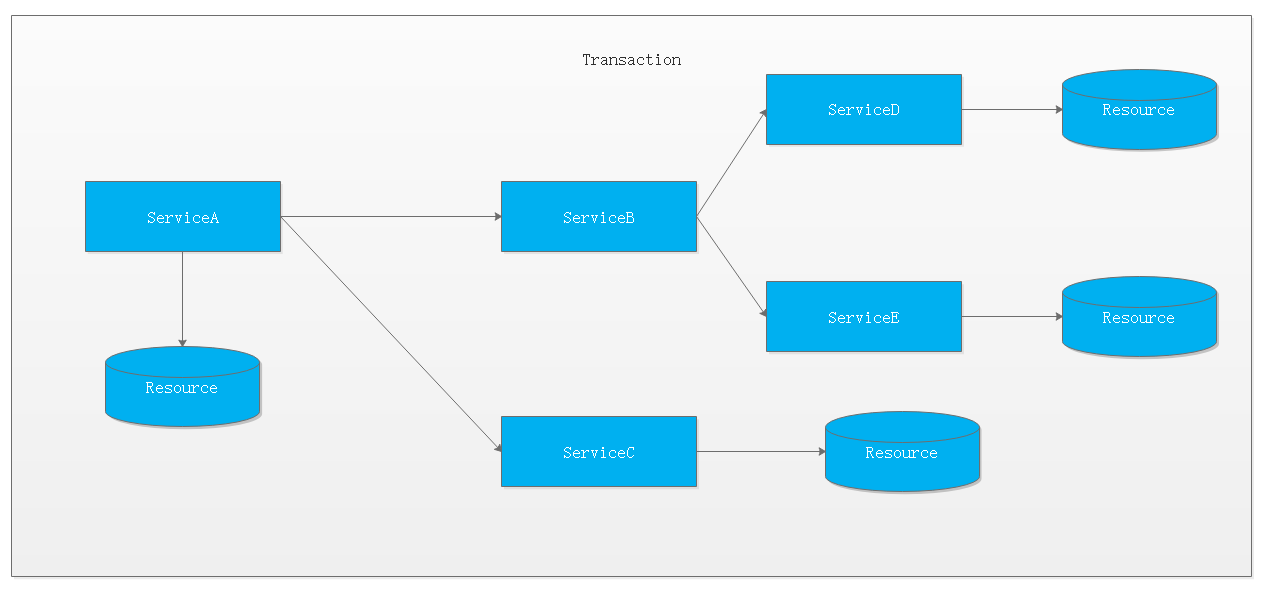
当一个服务操作访问不同的数据库资源，又希望对它们的访问具有事务特性时，就需要采用分布式事务来协调所有的事务参与者。

对于上面介绍的分布式事务应用架构，尽管一个服务操作会访问多个数据库资源，但是毕竟整个事务还是控制在单一服务的内部。如果一个服务操作需要调用另外一个服务，这时的事务就需要跨越多个服务了。在这种情况下，起始于某个服务的事务在调用另外一个服务的时候，需要以某种机制流转到另外一个服务，从而使被调用的服务访问的资源也自动加入到该事务当中来。下图反映了这样一个跨越多个服务的分布式事务：



### 多服务多数据源分布式事务

如果将上面这两种场景(一个服务可以调用多个数据库资源，也可以调用其他服务)结合在一起，对此进行延伸，整个分布式事务的参与者将会组成如下图所示的树形拓扑结构。在一个跨服务的分布式事务中，事务的发起者和提交均系同一个，它可以是整个调用的客户端，也可以是客户端最先调用的那个服务。



较之基于单一数据库资源访问的本地事务，分布式事务的应用架构更为复杂。在不同的分布式应用架构下，实现一个分布式事务要考虑的问题并不完全一样，比如对多资源的协调、事务的跨服务传播等，实现机制也是复杂多变。

## CAP定理

1998年，加州大学的计算机科学家 Eric Brewer(布鲁尔) 提出，分布式系统有三个指标。

**核心思想是：在分布式系统中一致性(C)、 可用性(A)、分区容错性(P)，三者不可兼得。**

**Consistency（一致性）**：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）。意思是，写操作之后的读操作，必须返回写后的结果。

**Availability（可用性）**：在集群中一部分节点故障后，集群整体是否还能满足响应客户端的读写请求。（对数据更新具备高可用性）。意思是只要收到用户的请求，服务器就必须给出回应。

**Partition tolerance（分区容错性）**：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。意思是，区间通信可能失败。比如，一台服务器放在中国，另一台服务器放在美国，这就是两个区，它们之间可能无法通信。

CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。**而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容错性是我们必须需要实现的。**所以我们只能在**一致性**和**可用性**之间**进行权衡**。

如果我们选择了 CA 而放弃了 P，那么当发生分区现象时，为了保证一致性，这个时候必须拒绝请求，但是 A 又不允许，所以分布式系统理论上不可能选择 CA 架构，只能选择 CP 或者 AP 架构。

对于 CP 来说，放弃可用性，追求一致性和分区容错性，我们的 ZooKeeper 其实就是追求的强一致。

对于 AP 来说，放弃一致性(这里说的一致性是强一致性)，追求分区容错性和可用性，这是很多分布式系统设计时的选择，后面的 BASE 也是根据 AP 来扩展。

顺便一提，CAP 理论中是忽略网络延迟，也就是当事务提交时，从节点 A 复制到节点 B 没有延迟，但是在现实中这个是明显不可能的，所以总会有一定的时间是不一致。

同时 CAP 中选择两个，比如你选择了 CP，并不是叫你放弃 A。因为 P 出现的概率实在是太小了，大部分的时间你仍然需要保证 CA。

就算分区出现了你也要为后来的 A 做准备，比如通过一些日志的手段，是其他机器回复至可用。

## BASE理论

刚才分析中，CAP不可能同时满足，而分区容错是对于分布式系统而言，是必须的。Base 理论是对 CAP 中一致性和可用性权衡的结果，其来源于对大型互联网分布式实践的总结，是基于 CAP 定理逐步演化而来的，由 ebay 的架构师提出。

全称为：

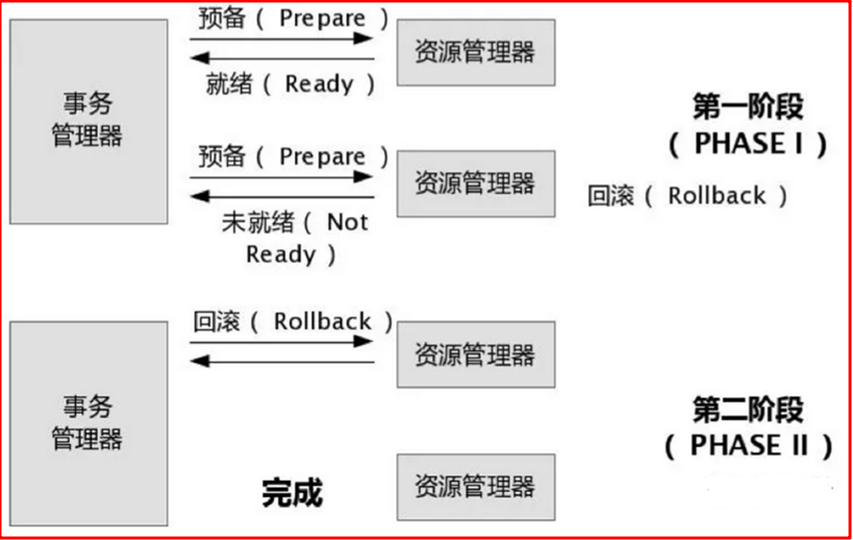
* Basically Available（基本可用）
* Soft state（软状态）
* Eventually consistent（最终一致性）

BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：既是无法做到强一致性（Strong consistency），但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。分布式事务解决方案。

# 分布式事务解决方案

## 基于XA协议的两阶段提交(2PC)

XA是一个分布式事务协议，由Tuxedo(分布式操作扩展之后的Unix事务系统)提出。XA中大致分为两部分：事务管理器和本地资源管理器。其中本地资源管理器往往由数据库实现，比如Oracle、DB2这些商业数据库都实现了XA接口，而事务管理器作为全局的调度者，负责各个本地资源的提交和回滚。XA实现分布式事务的原理如下：



在 XA 协议中分为两阶段：

* 事务管理器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交。
* 事务协调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，实现成本较低，在各大主流数据库都有自己实现，对于 MySQL 是从 5.5 开始支持。

**缺点**：

* 单点问题：事务管理器在整个流程中扮演的角色很关键，如果其宕机，比如在第一阶段已经完成，在第二阶段正准备提交的时候事务管理器宕机，资源管理器就会一直阻塞，导致数据库无法使用。
* 同步阻塞：在准备就绪之后，资源管理器中的资源一直处于阻塞，直到提交完成，释放资源。
* 数据不一致：两阶段提交协议虽然为分布式数据强一致性所设计，但仍然存在数据不一致性的可能。比如在第二阶段中，假设协调者发出了事务 Commit 的通知，但是因为网络问题该通知仅被一部分参与者所收到并执行了 Commit 操作，其余的参与者则因为没有收到通知一直处于阻塞状态，这时候就产生了数据的不一致性。

总的来说，XA 协议比较简单，成本较低，但是其单点问题，以及不能支持高并发(由于同步阻塞)依然是其最大的弱点。

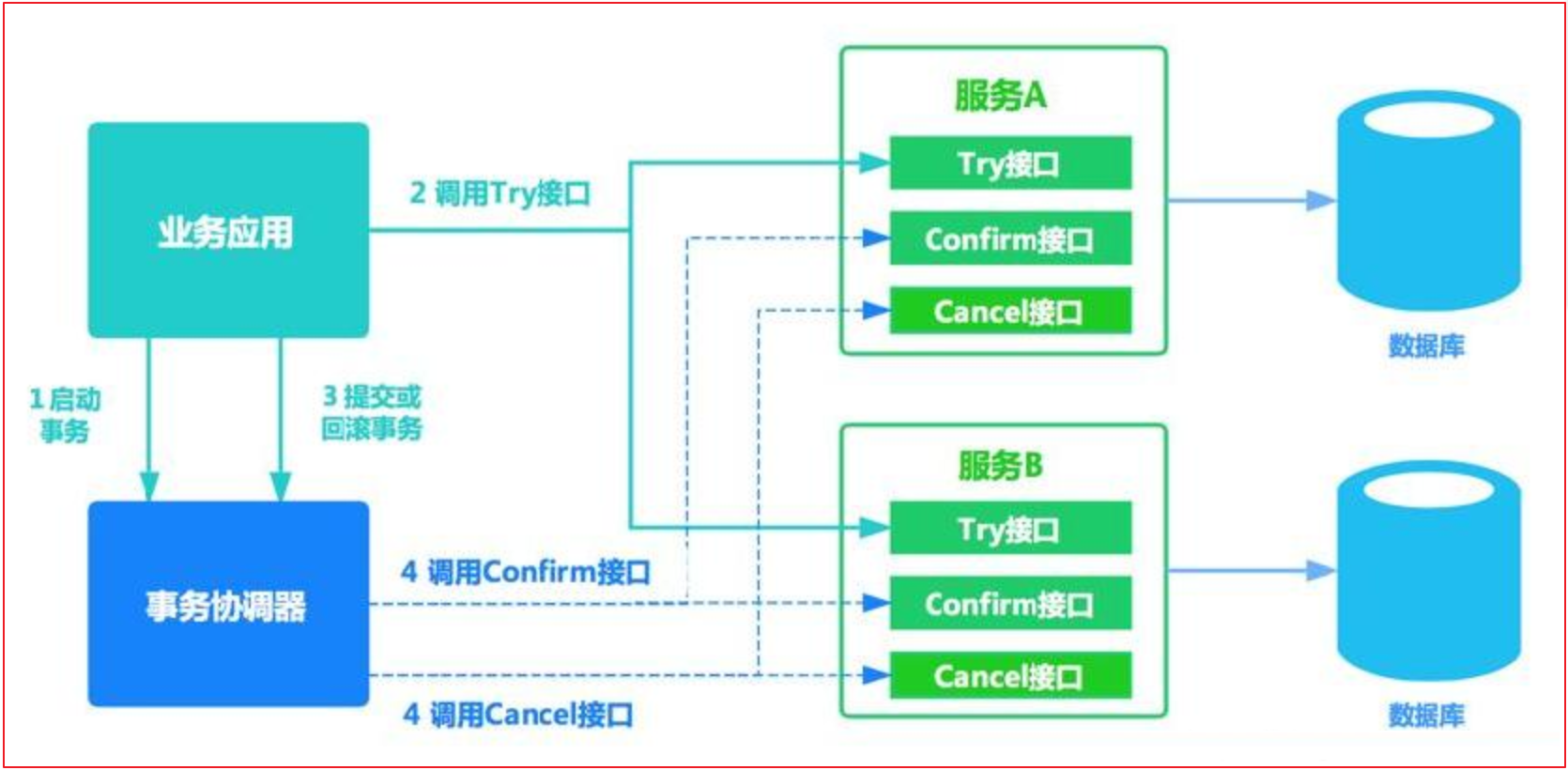
## TCC编程事务

关于 TCC（Try-Confirm-Cancel）的概念，最早是由 Pat Helland 于 2007 年发表的一篇名为《Life beyond Distributed Transactions：an Apostate’s Opinion》的论文提出。

TCC就是提供了一个编程框架，将整个业务逻辑分为三块：Try、Confirm和Cancel三个操作。以在线下单为例，Try阶段会去扣库存，Confirm阶段则是去更新订单状态，如果更新订单失败，则进入Cancel阶段，会去恢复库存。总之，TCC就是通过代码人为实现了两阶段提交，不同的业务场景所写的代码都不一样，复杂度也不一样。

TCC 事务机制相比于上面介绍的 XA，解决了如下几个缺点：

* 解决了协调者单点，由主业务方发起并完成这个业务活动。业务活动管理器也变成多点，引入集群。
* 同步阻塞：引入超时，超时后进行补偿，并且不会锁定整个资源，将资源转换为业务逻辑形式，粒度变小。
* 数据一致性，有了补偿机制之后，由业务活动管理器控制一致性。



对于 TCC 的解释：

* Try 阶段：尝试执行，完成所有业务检查（一致性），预留必需业务资源（准隔离性）。
* 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，Confirm 操作满足幂等性。要求具备幂等设计，Confirm 失败后需要进行重试。
* Cancel 阶段：取消执行，释放 Try 阶段预留的业务资源，Cancel 操作满足幂等性。Cancel 阶段的异常和 Confirm 阶段异常处理方案基本上一致。

举个简单的例子：如果你用 100 元买了一瓶水， Try 阶段：你需要向你的钱包检查是否够 100 元并锁住这 100 元，水也是一样的。

如果有一个失败，则进行 Cancel(释放这 100 元和这一瓶水)，如果 Cancel 失败不论什么失败都进行重试 Cancel，所以需要保持幂等。

如果都成功，则进行 Confirm，确认这 100 元被扣，和这一瓶水被卖，如果 Confirm 失败无论什么失败则重试(会依靠活动日志进行重试)。

**对于 TCC 来说适合一些**：

强隔离性，严格一致性要求的活动业务。

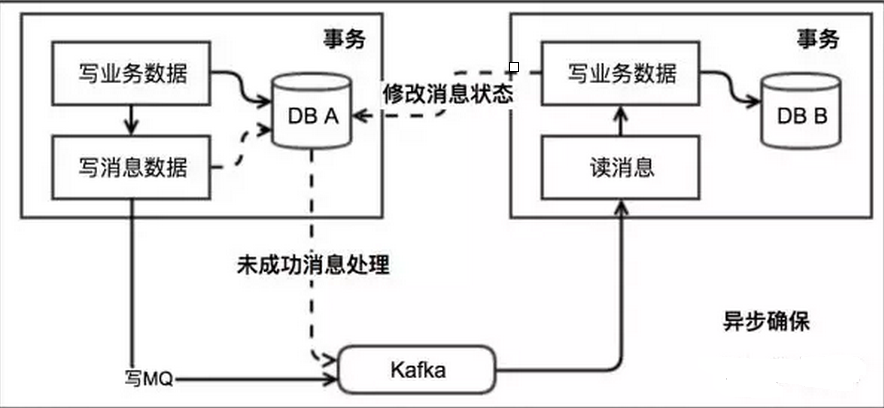
执行时间较短的业务。

## 本地消息表-异步确保

本地消息表这个方案最初是 eBay 提出的，eBay 的完整方案 https://queue.acm.org/detail.cfm?id=1394128。

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本地文本、数据库或消息队列，再通过业务规则自动或人工发起重试。

人工重试更多的是应用于支付场景，通过对账系统对事后问题的处理。



对于本地消息队列来说核心是把大事务转变为小事务。还是举上面用 100 元去买一瓶水的例子。

1. 当你扣钱的时候，你需要在你扣钱的服务器上新增加一个本地消息表，你需要把你扣钱和减去水的库存写入到本地消息表，放入同一个事务(依靠数据库本地事务保证一致性）。

2. 这个时候有个定时任务去轮询这个本地事务表，把没有发送的消息，扔给商品库存服务器，叫它减去水的库存，到达商品服务器之后，这时得先写入这个服务器的事务表，然后进行扣减，扣减成功后，更新事务表中的状态。

3. 商品服务器通过定时任务扫描消息表或者直接通知扣钱服务器，扣钱服务器在本地消息表进行状态更新。

4. 针对一些异常情况，定时扫描未成功处理的消息，进行重新发送，在商品服务器接到消息之后，首先判断是否是重复的。

如果已经接收，再判断是否执行，如果执行在马上又进行通知事务；如果未执行，需要重新执行由业务保证幂等，也就是不会多扣一瓶水。

本地消息队列是 BASE 理论，是最终一致模型，适用于对一致性要求不高的情况。实现这个模型时需要注意重试的幂等。

## MQ 事务消息

有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

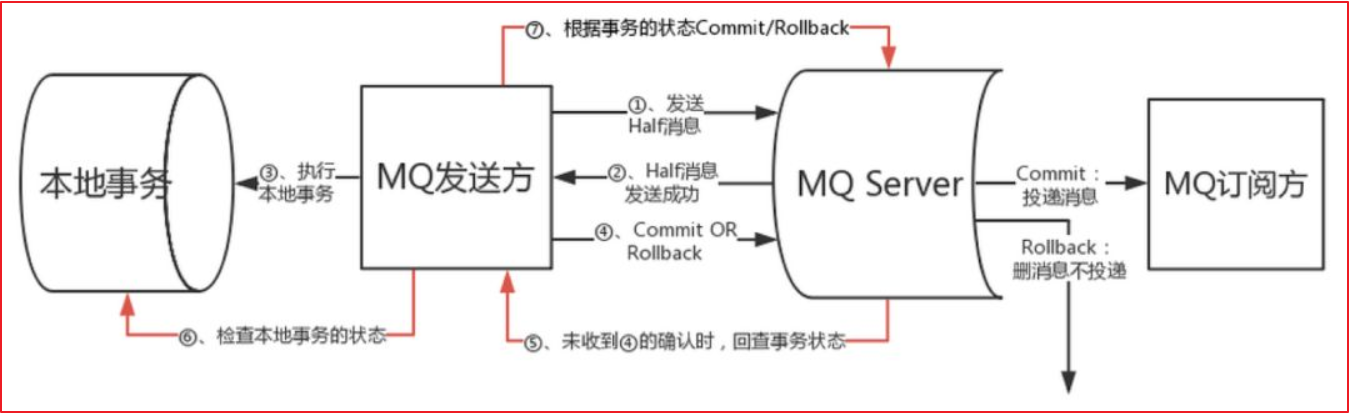
在 RocketMQ 中实现了分布式事务，实际上是对本地消息表的一个封装，将本地消息表移动到了 MQ 内部

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址。

第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。



**优点：** 实现了最终一致性，不需要依赖本地数据库事务。

**缺点：** 目前主流MQ中只有RocketMQ支持事务消息。

## Seata



Seata 是一款阿里巴巴开源的分布式事务解决方案，致力于在微服务架构下提供高性能和简单易用的分布式事务服务。

中文官网：<http://seata.io/zh-cn/index.html>

Github地址：<https://github.com/seata/seata>

2019 年 1 月，阿里巴巴中间件团队发起了开源项目 Fescar[feɪzkɑːr] 全称(Fast & EaSy Commit And Rollback)，和社区一起共建开源分布式事务解决方案。Fescar 的愿景是让分布式事务的使用像本地事务的使用一样，简单和高效，并逐步解决开发者们遇到的分布式事务方面的所有难题。

Fescar 开源后，蚂蚁金服加入 Fescar 社区参与共建，并在 Fescar 0.4.0 版本中贡献了 TCC 模式。

为了打造更中立、更开放、生态更加丰富的分布式事务开源社区，经过社区核心成员的投票，大家决定对 Fescar 进行品牌升级，并更名为 Seata，意为： Simple Extensible Autonomous Transaction Architecture(简单的可扩展自治事务体系结构)，是一套一站式分布式事务解决方案。

Seata 融合了阿里巴巴和蚂蚁金服在分布式事务技术上的积累，并沉淀了新零售、云计算和新金融等场景下丰富的实践经验。

### 发展历史

**蚂蚁金融**

* XTS：扩展交易服务。Ant Financial中间件团队自2007年以来开发了分布式事务中间件，广泛应用于Ant Financial，解决了跨数据库和服务的数据一致性问题。
* DTX：分布式事务扩展。自2013年以来，XTS已在Ant Financial Cloud上发布，名称为DTX。

**阿里巴巴**

* TXC：淘宝交易构造函数。阿里巴巴中间件团队自2014年起启动该项目，以解决因应用程序架构从单片机改为微服务而导致的分布式事务问题。
* GTS：全球交易服务。TXC作为Aliyun中间件产品，新名称GTS自2016年起发布。
* Fescar：我们从2019年开始基于TXC / GTS开源开源项目Fescar，以便在未来与社区密切合作。

**Seata社区**

* Seata：简单的可扩展自治交易架构。Ant Financial加入Fescar，使其成为一个更加中立和开放的分布式交易社区，并将Fescar更名为Seata。

### Seata设计初衷与事务模式

**解决分布式事务问题，有两个设计初衷**

**对业务无侵入：**即减少技术架构上的微服务化所带来的分布式事务问题对业务的侵入

**高性能：**减少分布式事务解决方案所带来的性能消耗

**seata中目前支持两种分布式事务实现方案，AT及TCC**

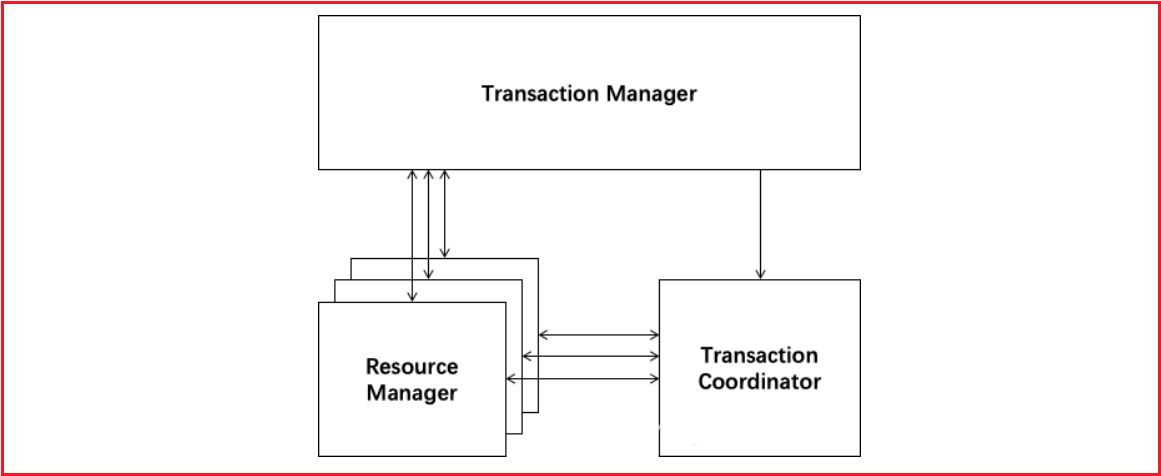
AT模式主要关注多 DB 访问的数据一致性，当然也包括多服务下的多 DB 数据访问一致性问题

TCC 模式主要关注业务拆分，在按照业务横向扩展资源时，解决微服务间调用的一致性问题

### AT模式

#### 介绍

Seata AT模式是基于XA事务演进而来的一个分布式事务中间件，XA是一个基于数据库实现的分布式事务协议，本质上和两阶段提交一样，需要数据库支持，Mysql5.5以上版本支持XA协议，其他数据库如Oracle，DB2也实现了XA接口



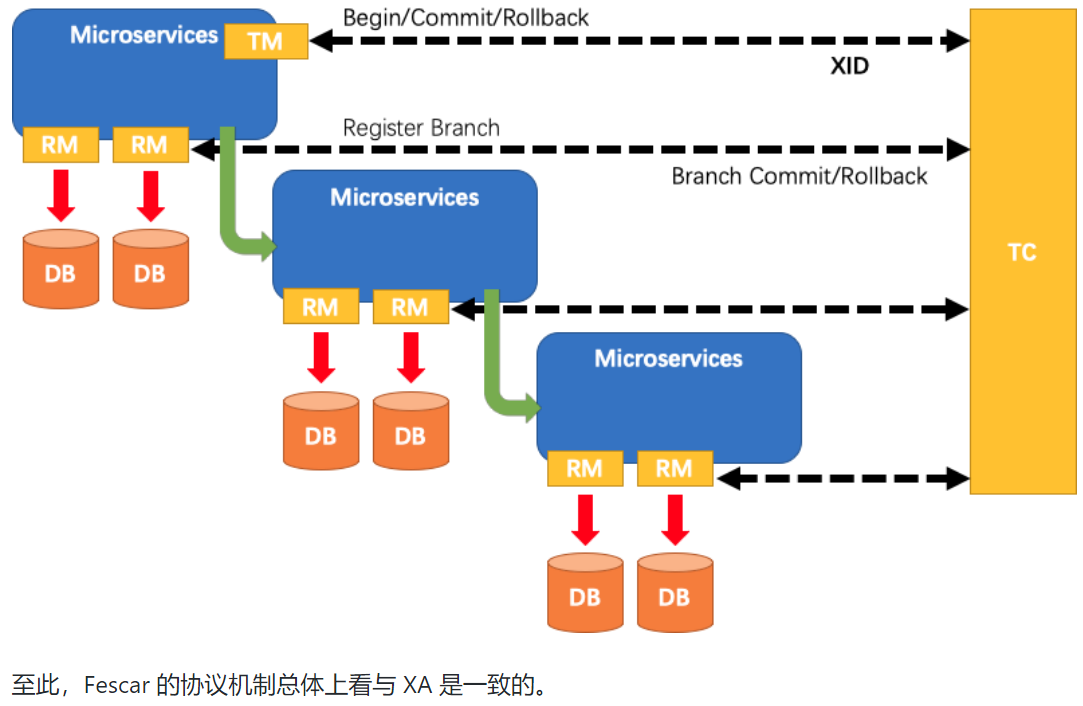
**Transaction Coordinator (TC)：** 事务协调器，维护全局事务的运行状态，负责协调并驱动全局事务的提交或回滚。

**Transaction Manager (TM)：** 控制全局事务的边界，负责开启一个全局事务，并最终发起全局提交或全局回滚的决议。

**Resource Manager (RM)：** 控制分支事务，负责分支注册、状态汇报，并接收事务协调器的指令，驱动分支（本地）事务的提交和回滚。

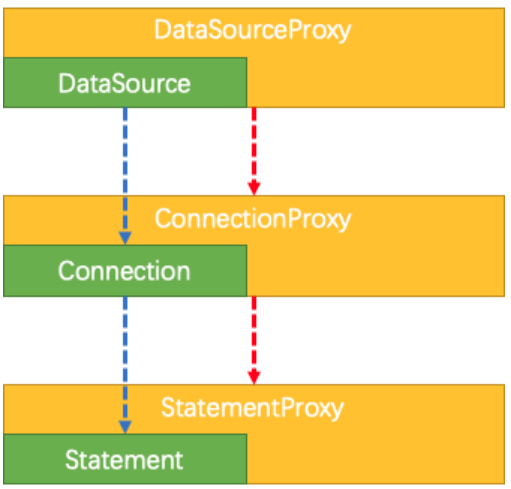
一个典型的分布式事务过程：

1. TM 向 TC 申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的 XID。
2. XID 在微服务调用链路的上下文中传播。
3. RM 向 TC 注册分支事务，将其纳入 XID 对应全局事务的管辖。
4. TM 向 TC 发起针对 XID 的全局提交或回滚决议。
5. TC 调度 XID 下管辖的全部分支事务完成提交或回滚请求。



#### 如何提交和回滚

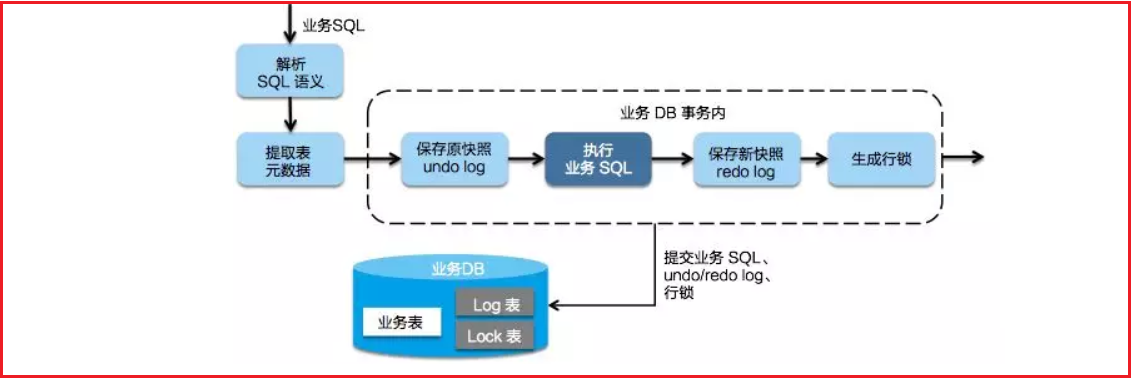
首先，应用需要使用 Fescar 的 JDBC 数据源代理，也就是 Fescar 的 RM。



##### 第一阶段

Seata 的 JDBC 数据源代理通过对业务 SQL 的解析，把业务数据在更新前后的数据镜像组织成回滚日志，利用 本地事务 的 ACID 特性，将业务数据的更新和回滚日志的写入在同一个 本地事务 中提交。

这样，可以保证：任何提交的业务数据的更新一定有相应的回滚日志存在



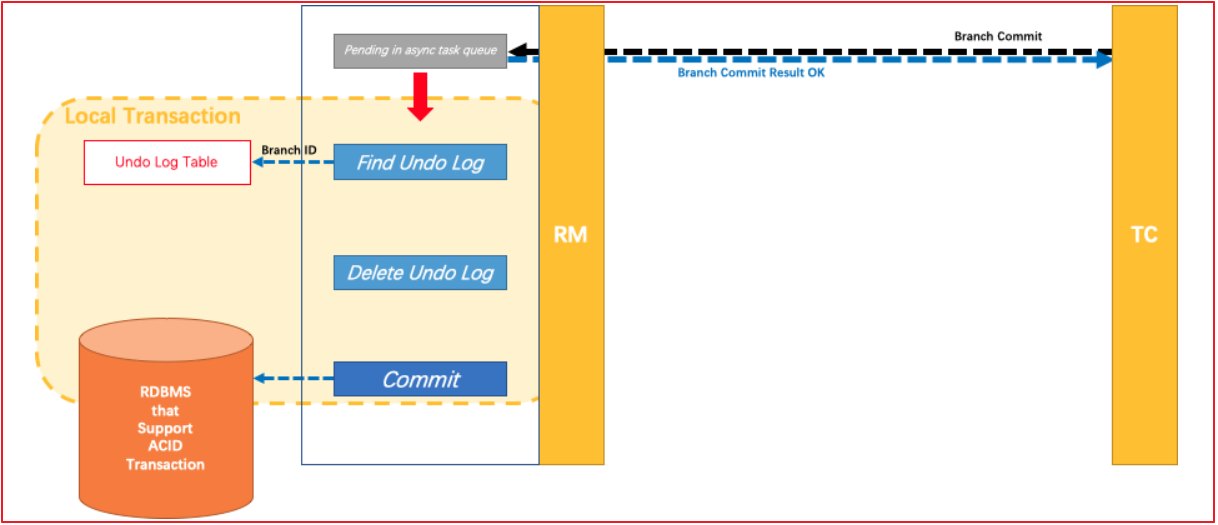
基于这样的机制，分支的本地事务便可以在全局事务的第一阶段提交，并马上释放本地事务锁定的资源

这也是Seata和XA事务的不同之处，两阶段提交往往对资源的锁定需要持续到第二阶段实际的提交或者回滚操作，而有了回滚日志之后，可以在第一阶段释放对资源的锁定，降低了锁范围，提高效率，即使第二阶段发生异常需要回滚，只需找对undolog中对应数据并反解析成sql来达到回滚目的。

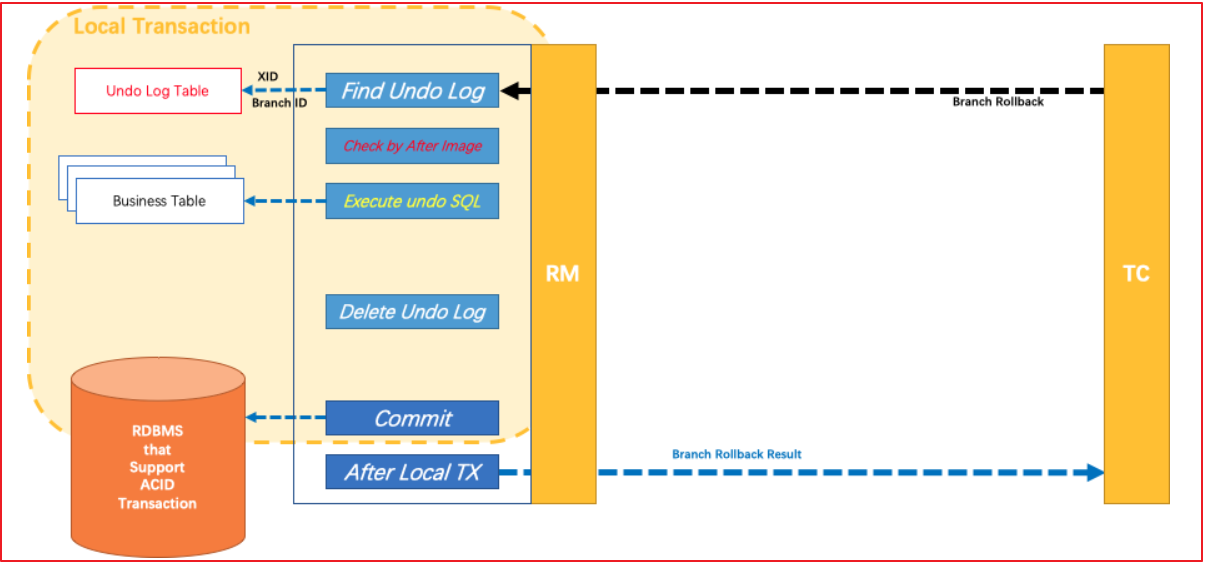
同时Seata通过代理数据源将业务sql的执行解析成undolog来与业务数据的更新同时入库，达到了对业务无侵入的效果。

##### 第二阶段

如果决议是全局提交，此时分支事务此时已经完成提交，不需要同步协调处理（只需要异步清理回滚日志），Phase2 可以非常快速地完成.

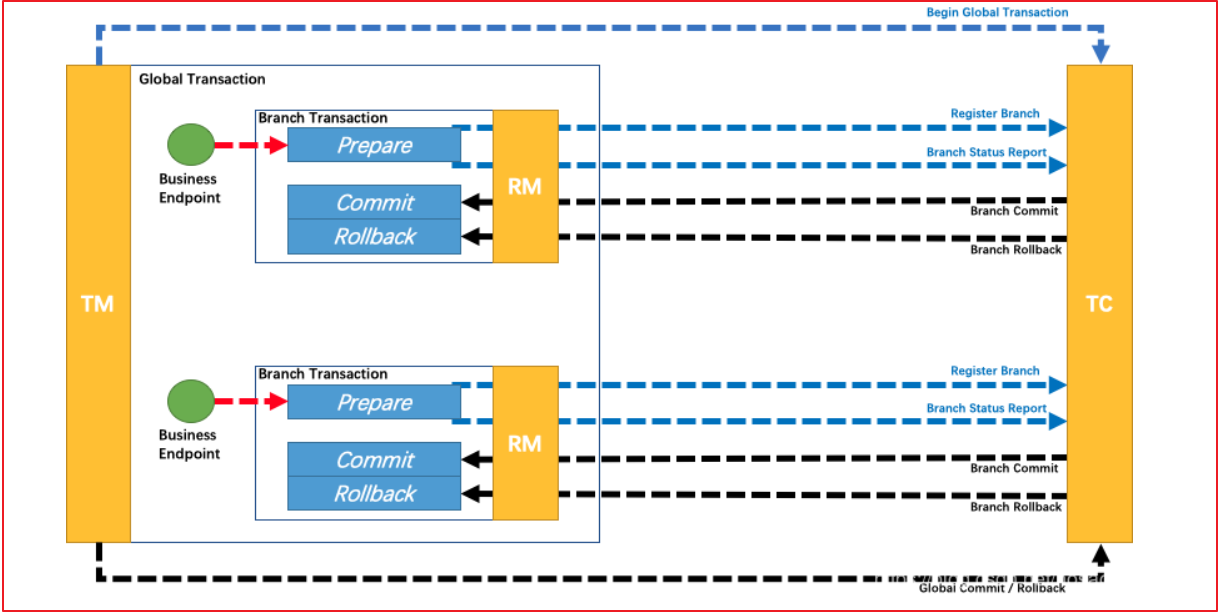


如果决议是全局回滚，RM 收到协调器发来的回滚请求，通过 XID 和 Branch ID 找到相应的回滚日志记录，\*\*通过回滚记录生成反向的更新 SQL 并执行\*\*，以完成分支的回滚。



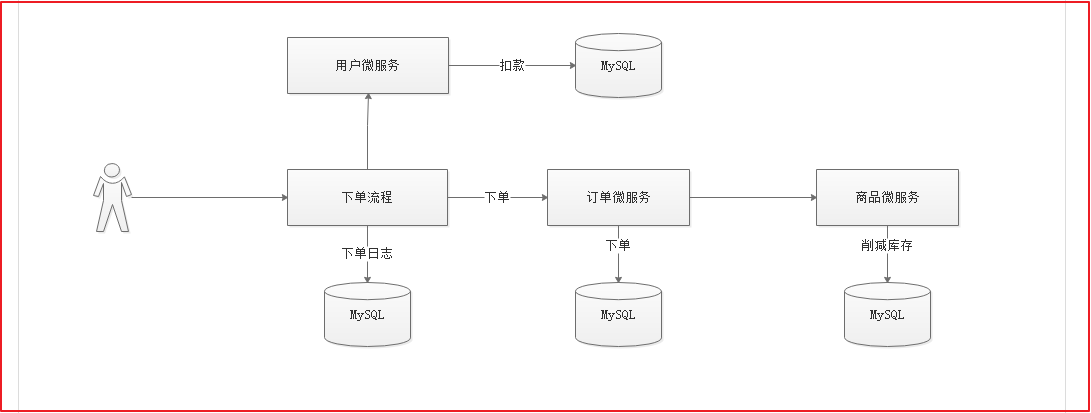
### TCC模式

seata也针对TCC做了适配兼容，支持TCC事务方案，原理前面已经介绍过，基本思路就是使用侵入业务上的补偿及事务管理器的协调来达到全局事务的一起提交及回滚。



# Seata案例

## 需求分析



完成一个案例，用户下单的时候记录下单日志，完成订单添加，完成用户账户扣款，完成商品库存削减功能，一会在任何一个微服务中制造异常，测试分布式事务。

先将 资料\seata\案例SQL脚本 数据库脚本导入到数据库中。



## 案例实现

### 父工程

搭建fescar\_parent,为了适应我们畅购工程的分布式事务，我们这里的父工程引入和畅购工程一样的依赖包。

pom.xml依赖如下：

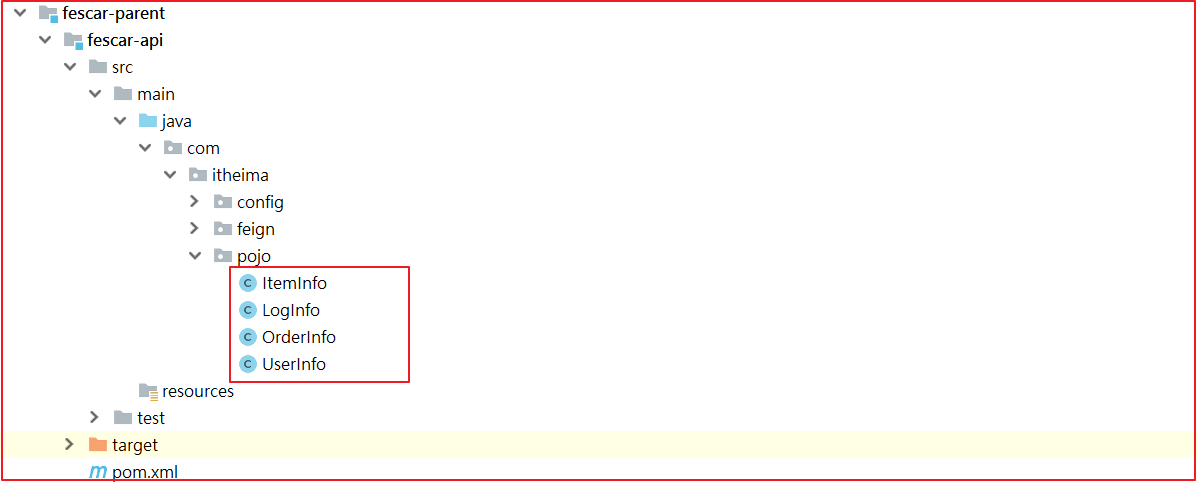
*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>fescar\_parent</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
  
 <**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.7.RELEASE</**version**>  
 </**parent**>  
  
 <**packaging**>pom</**packaging**>  
  
 *<!--跳过测试-->* <**properties**>  
 <**skipTests**>true</**skipTests**>  
 </**properties**>  
  
 *<!--依赖包-->* <**dependencies**>  
 *<!--测试包-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 </**dependency**>  
 *<!--fastjson-->* <**dependency**>  
 <**groupId**>com.alibaba</**groupId**>  
 <**artifactId**>fastjson</**artifactId**>  
 <**version**>1.2.51</**version**>  
 </**dependency**>  
 *<!--鉴权-->* <**dependency**>  
 <**groupId**>io.jsonwebtoken</**groupId**>  
 <**artifactId**>jjwt</**artifactId**>  
 <**version**>0.9.0</**version**>  
 </**dependency**>  
  
 *<!--web起步依赖-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>  
 *<!-- redis 使用-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-redis</**artifactId**>  
 </**dependency**>  
 *<!--eureka-client-->* <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  
 </**dependency**>  
 *<!--openfeign-->* <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-openfeign</**artifactId**>  
 </**dependency**>  
 *<!--微信支付-->* <**dependency**>  
 <**groupId**>com.github.wxpay</**groupId**>  
 <**artifactId**>wxpay-sdk</**artifactId**>  
 <**version**>0.0.3</**version**>  
 </**dependency**>  
 *<!--httpclient支持,微信支付-->* <**dependency**>  
 <**groupId**>org.apache.httpcomponents</**groupId**>  
 <**artifactId**>httpclient</**artifactId**>  
 </**dependency**>  
 *<!--通用mapper起步依赖,MyBatis通用Mapper封装，基于MyBatis动态SQL实现,可以实现对数据库的操作，不需要编写SQL语句-->* <**dependency**>  
 <**groupId**>tk.mybatis</**groupId**>  
 <**artifactId**>mapper-spring-boot-starter</**artifactId**>  
 <**version**>2.0.4</**version**>  
 </**dependency**>  
 *<!--MySQL数据库驱动-->* <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 </**dependency**>  
 *<!--mybatis分页插件,用于解决数据库分页实现 PageHelper.start(当前页，每页显示的条数)-->* <**dependency**>  
 <**groupId**>com.github.pagehelper</**groupId**>  
 <**artifactId**>pagehelper-spring-boot-starter</**artifactId**>  
 <**version**>1.2.3</**version**>  
 </**dependency**>  
 </**dependencies**>  
  
 <**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>Greenwich.SR1</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
 </**dependencyManagement**>  
</**project**>

### 公共工程

将所有数据库对应的Pojo/Feign抽取出一个公共工程fescar\_api,在该工程中pom如下:

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**parent**>  
 <**artifactId**>fescar\_parent</**artifactId**>  
 <**groupId**>com.itheima</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>fescar\_api</**artifactId**>  
</**project**>

将Pojo导入到工程中



### 商品微服务

创建fescar\_item微服务，在该工程中实现库存削减。

#### pom.xml

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**parent**>  
 <**artifactId**>fescar\_parent</**artifactId**>  
 <**groupId**>com.itheima</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>fescar\_item</**artifactId**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>fescar\_api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### Dao

创建com.itheima.dao.ItemInfoMapper,代码如下：

**public interface** ItemInfoMapper **extends** Mapper<ItemInfo> {  
}

#### Service

创建com.itheima.service.ItemInfoService接口，并创建库存递减方法，代码如下：

**public interface** ItemInfoService {  
 */\*\*  
 \* 库存递减  
 \** ***@param id*** *\** ***@param count*** *\*/* **void** decrCount(**int** id, **int** count);  
}

创建com.itheima.service.impl.ItemInfoServiceImpl实现库存递减操作，代码如下：

@Service  
**public class** ItemInfoServiceImpl **implements** ItemInfoService {  
  
 @Autowired  
 **private** ItemInfoMapper **itemInfoMapper**;  
  
 */\*\*\*  
 \* 库存递减**\*/* @Transactional(rollbackFor = Exception.**class**)  
 @Override  
 **public void** decrCount(**int** id, **int** count) {  
 *//查询商品信息* ItemInfo itemInfo = **itemInfoMapper**.selectByPrimaryKey(id);  
 itemInfo.setCount(itemInfo.getCount()-count);  
 **int** dcount = **itemInfoMapper**.updateByPrimaryKeySelective(itemInfo);  
 System.***out***.println(**"库存递减受影响行数："**+dcount);  
 *//异常准备回滚  
 //int q=10/0;* }  
}

#### Controller

创建com.itheima.controller.ItemInfoController，代码如下：

@RestController  
@RequestMapping(**"/itemInfo"**)  
@CrossOrigin  
**public class** ItemInfoController {  
  
 @Autowired  
 **private** ItemInfoService **itemInfoService**;  
 */\*\*  
 \* 库存递减**\** ***@return*** *\*/* @PostMapping(value = **"/decrCount"**)  
 **public** String decrCount(@RequestParam(value = **"id"**) **int** id, @RequestParam(value = **"count"**) **int** count){  
 *//库存递减* **itemInfoService**.decrCount(id,count);  
 **return "success"**;  
 }  
}

#### 启动类

创建com.itheima.ItemApplication代码如下：

@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients(basePackages = {**"com.itheima.feign"**})  
@MapperScan(basePackages = {**"com.itheima.dao"**})  
**public class** ItemApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(ItemApplication.**class**,args);  
 }  
}

#### application.yml

创建applicatin.yml,配置如下：

**server**:  
 **port**: 18082  
**spring**:  
 **application**:  
 **name**: item  
 **datasource**:  
 **driver-class-name**: com.mysql.cj.jdbc.Driver  
 **url**: jdbc:mysql://192.168.211.132:3306/fescar\_item?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 **username**: root  
 **password**: 123456  
 **main**:  
 **allow-bean-definition-overriding**: **true  
eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://127.0.0.1:7001/eureka  
 **instance**:  
 **prefer-ip-address**: **true  
feign**:  
 **hystrix**:  
 **enabled**: **true***#hystrix 配置***hystrix**:  
 **command**:  
 **default**:  
 **execution**:  
 **isolation**:  
 **thread**:  
 **timeoutInMilliseconds**: 10000  
 **strategy**: SEMAPHORE

### 用户微服务

创建fescar\_user微服务，并引入公共工程依赖。

#### pom.xml

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**parent**>  
 <**artifactId**>fescar\_parent</**artifactId**>  
 <**groupId**>com.itheima</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>fescar\_user</**artifactId**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>fescar\_api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### Dao

创建com.itheima.dao.UserInfoMapper，代码如下：

**public interface** UserInfoMapper **extends** Mapper<UserInfo> {  
}

#### Service

创建com.itheima.service.UserInfoService接口，代码如下：

**public interface** UserInfoService {  
  
 */\*\*\*  
 \* 账户金额递减  
 \** ***@param username*** *\** ***@param money*** *\*/* **void** decrMoney(String username, **int** money);  
}

创建com.itheima.service.impl.UserInfoServiceImpl实现用户账户扣款，代码如下：

@Service  
**public class** UserInfoServiceImpl **implements** UserInfoService {  
  
 @Autowired  
 **private** UserInfoMapper **userInfoMapper**;  
  
 */\*\*\*  
 \* 账户金额递减  
 \** ***@param username*** *\** ***@param money*** *\*/* @Transactional(rollbackFor = Exception.**class**)  
 @Override  
 **public void** decrMoney(String username, **int** money) {  
 UserInfo userInfo = **userInfoMapper**.selectByPrimaryKey(username);  
 userInfo.setMoney(userInfo.getMoney()-money);  
 **int** count = **userInfoMapper**.updateByPrimaryKeySelective(userInfo);  
 System.***out***.println(**"添加用户受影响行数："**+count);  
 *//int q=10/0;* }  
}

#### Controller

创建com.itheima.controller.UserInfoController代码如下：

@RestController  
@RequestMapping(**"/userInfo"**)  
@CrossOrigin  
**public class** UserInfoController {  
  
 @Autowired  
 **private** UserInfoService **userInfoService**;  
  
 */\*\*\*  
 \* 账户余额递减  
 \** ***@param username*** *\** ***@param money*** *\*/* @PostMapping(value = **"/add"**)  
 **public** String decrMoney(@RequestParam(value = **"username"**) String username, @RequestParam(value = **"money"**) **int** money) {  
 **userInfoService**.decrMoney(username, money);  
 **return "success"**;  
 }  
  
}

#### 启动类

创建com.itheima.UserApplication，代码如下：

@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients(basePackages = {**"com.itheima.feign"**})  
@MapperScan(basePackages = {**"com.itheima.dao"**})  
**public class** UserApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(UserApplication.**class**,args);  
 }  
}

#### application.yml

创建application.yml配置如下：

**server**:  
 **port**: 18084  
**spring**:  
 **application**:  
 **name**: user  
 **datasource**:  
 **driver-class-name**: com.mysql.cj.jdbc.Driver  
 **url**: jdbc:mysql://192.168.211.132:3306/fescar\_user?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 **username**: root  
 **password**: 123456  
 **main**:  
 **allow-bean-definition-overriding**: **true  
eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://127.0.0.1:7001/eureka  
 **instance**:  
 **prefer-ip-address**: **true  
feign**:  
 **hystrix**:  
 **enabled**: **true***#hystrix 配置***hystrix**:  
 **command**:  
 **default**:  
 **execution**:  
 **isolation**:  
 **thread**:  
 **timeoutInMilliseconds**: 10000  
 **strategy**: SEMAPHORE

### 订单微服务

创建fescar\_order服务工程，在订单微服务中实现调用商品微服务递减库存。

#### pom.xml

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**parent**>  
 <**artifactId**>fescar\_parent</**artifactId**>  
 <**groupId**>com.itheima</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>fescar\_order</**artifactId**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>fescar\_api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### Dao

创建com.itheima.dao.OrderInfoMapper，代码如下：

**public interface** OrderInfoMapper **extends** Mapper<OrderInfo> {  
}

#### Service

创建com.itheima.service.OrderInfoService实现添加订单操作，代码如下：

**public interface** OrderInfoService {  
  
 */\*\*\*  
 \* 添加订单  
 \** ***@param username*** *\** ***@param id*** *\** ***@param count*** *\*/* **void** add(String username, **int** id, **int** count);  
}

创建com.itheima.service.impl.OrderInfoServiceImpl，代码如下：

@Service  
**public class** OrderInfoServiceImpl **implements** OrderInfoService {  
  
 @Autowired  
 **private** OrderInfoMapper **orderInfoMapper**;  
  
 @Autowired  
 **private** ItemInfoFeign **itemInfoFeign**;  
  
 */\*\*\*  
 \* 添加订单  
 \** ***@param username*** *\** ***@param id*** *\** ***@param count*** *\*/* @Override  
 **public void** add(String username, **int** id, **int** count) {  
 *//添加订单* OrderInfo orderInfo = **new** OrderInfo();  
 orderInfo.setMessage(**"生成订单"**);  
 orderInfo.setMoney(10);  
 **int** icount = **orderInfoMapper**.insertSelective(orderInfo);  
 System.***out***.println(**"添加订单受影响函数："**+icount);  
  
 *//递减库存* **itemInfoFeign**.decrCount(id,count);  
 }  
}

#### Controller

创建com.itheima.controller.OrderInfoController调用下单操作，代码如下：

@RestController  
@RequestMapping(**"/orderInfo"**)  
@CrossOrigin  
**public class** OrderInfoController {  
  
 @Autowired  
 **private** OrderInfoService **orderInfoService**;  
  
 */\*\*  
 \* 增加订单  
 \** ***@param username*** *\** ***@param id*** *\** ***@param count*** *\*/* @PostMapping(value = **"/add"**)  
 **public** String add(@RequestParam(value = **"name"**) String username, @RequestParam(value = **"id"**) **int** id, @RequestParam(value = **"count"**) **int** count){  
 *//添加订单* **orderInfoService**.add(username,id,count);  
 **return "success"**;  
 }  
  
}

#### 启动类

创建com.itheima.OrderApplication启动类，代码如下：

@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients(basePackages = {**"com.itheima.feign"**})  
@MapperScan(basePackages = {**"com.itheima.dao"**})  
**public class** OrderApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(OrderApplication.**class**,args);  
 }  
}

#### application.yml配置

**server**:  
 **port**: 18083  
**spring**:  
 **application**:  
 **name**: order  
 **datasource**:  
 **driver-class-name**: com.mysql.cj.jdbc.Driver  
 **url**: jdbc:mysql://192.168.211.132:3306/fescar\_order?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 **username**: root  
 **password**: 123456  
 **main**:  
 **allow-bean-definition-overriding**: **true  
eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://127.0.0.1:7001/eureka  
 **instance**:  
 **prefer-ip-address**: **true  
feign**:  
 **hystrix**:  
 **enabled**: **true***#hystrix 配置***hystrix**:  
 **command**:  
 **default**:  
 **execution**:  
 **isolation**:  
 **thread**:  
 **timeoutInMilliseconds**: 10000  
 **strategy**: SEMAPHORE

### 业务微服务

创建fescar\_business业务微服务，在该微服务中实现分布式事务控制，下单入口从这里开始。

#### pom.xml

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**parent**>  
 <**artifactId**>fescar\_parent</**artifactId**>  
 <**groupId**>com.itheima</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
 <**artifactId**>fescar\_business</**artifactId**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>fescar\_api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### Dao

创建com.itheima.dao.LogInfoMapper代码如下：

**public interface** LogInfoMapper **extends** Mapper<LogInfo> {  
}

#### Service

创建com.itheima.service.BusinessService接口，代码如下：

**public interface** BusinessService {  
  
 */\*\*  
 \* 下单  
 \** ***@param username*** *\** ***@param id*** *\** ***@param count*** *\*/* **void** add(String username, **int** id, **int** count);  
}

创建com.itheima.service.impl.BusinessServiceImpl，代码如下：

@Service  
**public class** BusinessServiceImpl **implements** BusinessService {  
  
 @Autowired  
 **private** OrderInfoFeign **orderInfoFeign**;  
 @Autowired  
 **private** UserInfoFeign **userInfoFeign**;  
 @Autowired  
 **private** LogInfoMapper **logInfoMapper**;  
  
 */\*\*\*  
 \* 下单  
 \** ***@param username*** *\** ***@param id*** *\** ***@param count*** *\*/* @Override  
 **public void** add(String username, **int** id, **int** count) {  
 *//添加订单日志* LogInfo logInfo = **new** LogInfo();  
 logInfo.setContent(**"添加订单数据---"** + **new** Date());  
 logInfo.setCreatetime(**new** Date());  
 **int** logcount = **logInfoMapper**.insertSelective(logInfo);  
 System.***out***.println(**"添加日志受影响行数："** + logcount);  
  
 *//添加订单* **orderInfoFeign**.add(username, id, count);  
  
 *//用户账户余额递减* **userInfoFeign**.decrMoney(username, 10);  
 }  
}

#### Controller

创建com.itheima.controller.BusinessController，代码如下：

@RestController  
@RequestMapping(value = **"/business"**)  
**public class** BusinessController {  
  
 @Autowired  
 **private** BusinessService **businessService**;  
  
 */\*\*\*  
 \* 购买商品分布式事务测试  
 \** ***@return*** *\*/* @RequestMapping(value = **"/addorder"**)  
 **public** String order(){  
 String username=**"zhangsan"**;  
 **int** id=1;  
 **int** count=5;  
 *//下单* **businessService**.add(username,id,count);  
 **return "success"**;  
 }  
}

#### 启动类

创建启动类com.itheima.BusinessApplication，代码如下：

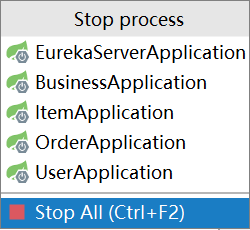
@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients(basePackages = {**"com.itheima.feign"**})  
@MapperScan(basePackages = {**"com.itheima.dao"**})  
**public class** BusinessApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(BusinessApplication.**class**,args);  
 }  
}

#### application.yml配置

**server**:  
 **port**: 18081  
**spring**:  
 **application**:  
 **name**: business  
 **datasource**:  
 **driver-class-name**: com.mysql.cj.jdbc.Driver  
 **url**: jdbc:mysql://192.168.211.132:3306/fescar\_business?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 **username**: root  
 **password**: 123456  
 **main**:  
 **allow-bean-definition-overriding**: **true  
eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://127.0.0.1:7001/eureka  
 **instance**:  
 **prefer-ip-address**: **true  
feign**:  
 **hystrix**:  
 **enabled**: **true***#读取超时设置***ribbon**:  
 **ReadTimeout**: 30000  
*#hystrix 配置***hystrix**:  
 **command**:  
 **default**:  
 **execution**:  
 **isolation**:  
 **thread**:  
 **timeoutInMilliseconds**: 10000  
 **strategy**: SEMAPHORE

### 无事务业务测试

启动changgou\_eureka再着，再启动刚才创建的所有微服务。



访问：<http://localhost:18081/business/addorder> ，查看相应的数据库变化情况，然后调手动代码在UserServiceImpl中，制造一个运行时异常，再观察数据库，看看回滚了没有？

## Spring Cloud 快速集成 Seata完成分布式事务

Seata官方原生态配置与使用相对麻烦，但是官方在原生态配置解决方案上提供了SpringCloud的支持，这时我们使用Seata将变得非常简单。

项目地址：<https://github.com/seata/seata-samples>

使用文档：<https://github.com/seata/seata-samples/blob/master/doc/quick-integration-with-spring-cloud.md>

### 添加依赖

在需要用到分布式事务的所有工程中，添加Spring Cloud Alibaba 依赖管理工具和 Seata 依赖

我们每个微服务都用到，所以就在fescar\_api中添加了：

<**dependencies**>  
 <**dependency**>  
 <**groupId**>com.alibaba.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-alibaba-seata</**artifactId**>  
 <**version**>2.1.0.RELEASE</**version**>  
 </**dependency**>  
</**dependencies**>

需要注意的是Spring Cloud Alibaba 的毕业版本的 GroupId 是 com.alibaba.cloud

spring-cloud-starter-alibaba-seata这个依赖中只依赖了spring-cloud-alibaba-seata，所以在项目中添加spring-cloud-starter-alibaba-seata和spring-cloud-alibaba-seata是一样的。

### 添加Seata 配置文件

#### registry.conf

该配置用于指定 TC 的注册中心和配置文件，默认都是 file; 如果使用其他的注册中心，要求 Seata-Server 也注册到该配置中心上。同样的我们加到fescar\_api工程中。

registry **{** *# file 、nacos 、eureka、redis、zk、consul、etcd3、sofa* type = **"file"** nacos **{** serverAddr = **"localhost"** namespace = **"public"** cluster = **"default"** }  
 eureka **{** serviceUrl = **"http://localhost:8761/eureka"** application = **"default"** weight = **"1"** }  
 redis **{** serverAddr = **"localhost:6379"** db = **"0"** }  
 zk **{** cluster = **"default"** serverAddr = **"127.0.0.1:2181"** session.timeout = **6000** connect.timeout = **2000** }  
 consul **{** cluster = **"default"** serverAddr = **"127.0.0.1:8500"** }  
 etcd3 **{** cluster = **"default"** serverAddr = **"http://localhost:2379"** }  
 sofa **{** serverAddr = **"127.0.0.1:9603"** application = **"default"** region = **"DEFAULT\_ZONE"** datacenter = **"DefaultDataCenter"** cluster = **"default"** group = **"SEATA\_GROUP"** addressWaitTime = **"3000"** }  
 file **{** name = **"file.conf"** }  
}  
  
config **{** *# file、nacos 、apollo、zk、consul、etcd3* type = **"file"** nacos **{** serverAddr = **"localhost"** namespace = **"public"** cluster = **"default"** }  
 consul **{** serverAddr = **"127.0.0.1:8500"** }  
 apollo **{** app.id = **"seata-server"** apollo.meta = **"http://192.168.1.204:8801"** }  
 zk **{** serverAddr = **"127.0.0.1:2181"** session.timeout = **6000** connect.timeout = **2000** }  
 etcd3 **{** serverAddr = **"http://localhost:2379"** }  
 file **{** name = **"file.conf"** }  
}

#### file.conf

该配置用于指定TC的相关属性；如果使用注册中心也可以将配置添加到配置中心

transport **{** *# tcp udt unix-domain-socket* type = **"TCP"** *#NIO NATIVE* server = **"NIO"** *#enable heartbeat* heartbeat = **true** *#thread factory for netty* thread-factory **{** boss-thread-prefix = **"NettyBoss"** worker-thread-prefix = **"NettyServerNIOWorker"** server-executor-thread-prefix = **"NettyServerBizHandler"** share-boss-worker = **false** client-selector-thread-prefix = **"NettyClientSelector"** client-selector-thread-size = **1** client-worker-thread-prefix = **"NettyClientWorkerThread"** *# netty boss thread size,will not be used for UDT* boss-thread-size = **1** *#auto default pin or 8* worker-thread-size = **8** }  
 shutdown **{** *# when destroy server, wait seconds* wait = **3** }  
 serialization = **"seata"** compressor = **"none"**}  
service **{** *#vgroup->rgroup* vgroup\_mapping.my\_test\_tx\_group = **"default"** *#only support single node* default.grouplist = **"127.0.0.1:8091"** *#degrade current not support* enableDegrade = **false** *#disable* disable = **false** *#unit ms,s,m,h,d represents milliseconds, seconds, minutes, hours, days, default permanent* max.commit.retry.timeout = **"-1"** max.rollback.retry.timeout = **"-1"**}  
  
client **{** async.commit.buffer.limit = **10000** lock **{** retry.internal = **10** retry.times = **30** }  
 report.retry.count = **5**}  
  
*## transaction log store*store **{** *## store mode: file、db* mode = **"file"** *## file store* file **{** dir = **"sessionStore"** *# branch session size , if exceeded first try compress lockkey, still exceeded throws exceptions* max-branch-session-size = **16384** *# globe session size , if exceeded throws exceptions* max-global-session-size = **512** *# file buffer size , if exceeded allocate new buffer* file-write-buffer-cache-size = **16384** *# when recover batch read size* session.reload.read\_size = **100** *# async, sync* flush-disk-mode = **async** }  
  
 *## database store* db **{** *## the implement of javax.sql.DataSource, such as DruidDataSource(druid)/BasicDataSource(dbcp) etc.* datasource = **"dbcp"** *## mysql/oracle/h2/oceanbase etc.* db-type = **"mysql"** url = **"jdbc:mysql://127.0.0.1:3306/seata"** user = **"mysql"** password = **"mysql"** min-conn = **1** max-conn = **3** global.table = **"global\_table"** branch.table = **"branch\_table"** lock-table = **"lock\_table"** query-limit = **100** }  
}  
lock **{** *## the lock store mode: local、remote* mode = **"remote"** local **{** *## store locks in user's database* }  
  
 remote **{** *## store locks in the seata's server* }  
}  
recovery **{** committing-retry-delay = **30** asyn-committing-retry-delay = **30** rollbacking-retry-delay = **30** timeout-retry-delay = **30**}  
  
transaction **{** undo.data.validation = **true** undo.log.serialization = **"jackson"**}  
  
*## metrics settings*metrics **{** enabled = **false** registry-type = **"compact"** *# multi exporters use comma divided* exporter-list = **"prometheus"** exporter-prometheus-port = **9898**}

需要注意的是 service.vgroup\_mapping这个配置，在 Spring Cloud 中默认是${spring.application.name}-fescar-service-group，可以通过指定application.properties的 spring.cloud.alibaba.seata.tx-service-group这个属性覆盖，但是必须要和 file.conf 中的一致，否则会提示 no available server to connect

service.vgroup\_mapping:设定为同一个事务的成员这个名字要相同

**vgroup\_mapping.my\_test\_tx\_group** = **"default" my\_test\_tx\_group为事务的集群名称**

#### 配置相关微服务的application.yml加入事务组

*#配置分布式事务组，注意此配置需要跟在spring节点后面***cloud**:  
 **alibaba**:  
 **seata**:  
 **tx-service-group**: my\_test\_tx\_group

### 注入数据源

Seata 通过代理数据源的方式实现分支事务；MyBatis 和 JPA 都需要注入 io.seata.rm.datasource.DataSourceProxy, 不同的是，MyBatis 还需要额外注入 org.apache.ibatis.session.SqlSessionFactory

在fescar\_api工程中创建com.itheima.conf.DataSourceProxyConfig

@Configuration  
**public class** DataSourceProxyConfig {  
 */\*\*  
 \* 普通数据源-通过yml的spring.datasource配置  
 \** ***@return*** *\*/* @Bean  
 @ConfigurationProperties(prefix = **"spring.datasource"**)  
 **public** DataSource dataSource() {  
 **return new** DruidDataSource();  
 }  
  
 */\*\*  
 \* 代理数据源-引用普通数据源  
 \** ***@param dataSource*** *\** ***@return*** *\*/* @Bean  
 **public** DataSourceProxy dataSourceProxy(DataSource dataSource) {  
 **return new** DataSourceProxy(dataSource);  
 }  
  
 */\*\*  
 \* MyBatis需要手动指定SqlSessionFactory绑定代理数据源  
 \** ***@param dataSourceProxy*** *\** ***@return*** *\** ***@throws*** *Exception  
 \*/* @Bean  
 **public** SqlSessionFactory sqlSessionFactoryBean(DataSourceProxy dataSourceProxy) **throws** Exception {  
 SqlSessionFactoryBean sqlSessionFactoryBean = **new** SqlSessionFactoryBean();  
 *//使用代理数据源* sqlSessionFactoryBean.setDataSource(dataSourceProxy);  
 **return** sqlSessionFactoryBean.getObject();  
 }  
}

### 添加 undo\_log 表

在业务相关的数据库中添加 undo\_log 表，用于保存需要回滚的数据。

CREATE TABLE `undo\_log`

(

`id` BIGINT(20) NOT NULL AUTO\_INCREMENT,

`branch\_id` BIGINT(20) NOT NULL,

`xid` VARCHAR(100) NOT NULL,

`context` VARCHAR(128) NOT NULL,

`rollback\_info` LONGBLOB NOT NULL,

`log\_status` INT(11) NOT NULL,

`log\_created` DATETIME NOT NULL,

`log\_modified` DATETIME NOT NULL,

`ext` VARCHAR(100) DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `ux\_undo\_log` (`xid`, `branch\_id`)

) ENGINE = InnoDB

AUTO\_INCREMENT = 1

DEFAULT CHARSET = utf8

### 启动 Seata-Server

在 <https://github.com/seata/seata/releases> 下载相应版本的 Seata-Server，修改 registry.conf为相应的配置(如果使用 file 则不需要修改)，解压并通过以下命令启动:

Linux：sh ./bin/seata-server.sh

windows：直接双击seata-server.bat

### 使用@GlobalTransactional开启事务

在业务的发起方的方法上使用@GlobalTransactional开启全局事务，Seata 会将事务的 xid 通过拦截器添加到调用其他服务的请求中，实现分布式事务。

在fescar\_business工程修改com.itheima.service.impl.BusinessServiceImpl

*/\*\*\*  
 \* 下单**\*/*@GlobalTransactional *//调用Service入口开启分布式事务*@Override  
**public void** add(String username, **int** id, **int** count) {  
 *//添加订单日志* LogInfo logInfo = **new** LogInfo();  
 logInfo.setContent(**"添加订单数据---"** + **new** Date());  
 logInfo.setCreatetime(**new** Date());  
 **int** logcount = **logInfoMapper**.insertSelective(logInfo);  
 System.***out***.println(**"添加日志受影响行数："** + logcount);  
  
 *//添加订单* **orderInfoFeign**.add(username, id, count);  
  
 *//用户账户余额递减* **userInfoFeign**.decrMoney(username, 10);  
}

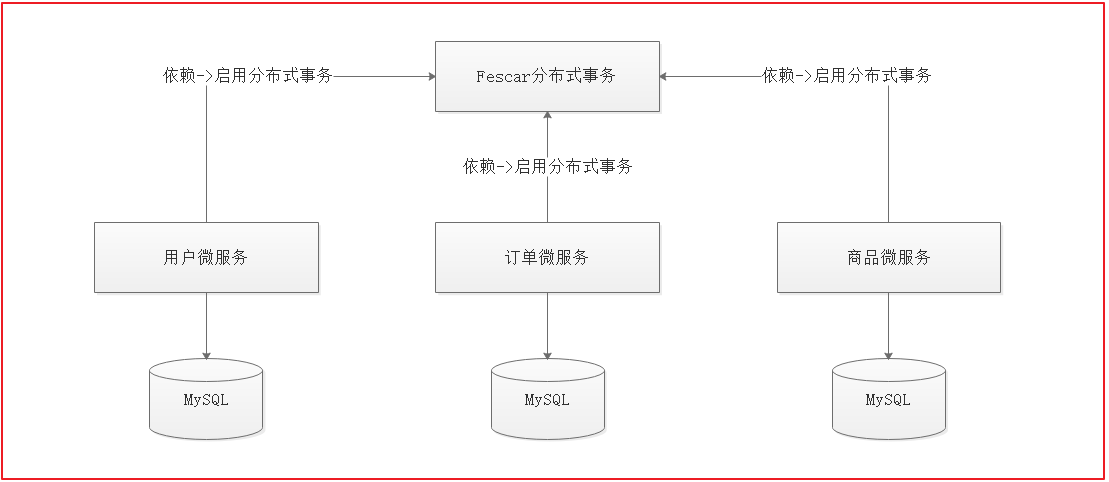
# 分布式事务实战(学员完成)

## undo\_log表结构导入

核心在于对业务sql进行解析，转换成undolog,所以只要支持Fescar分布式事务的微服务数据都需要导入该表结构，我们在每个微服务的数据库中都导入下面表结构：

**DROP TABLE IF EXISTS** `undo\_log`;  
**CREATE TABLE** `undo\_log`  
(  
 **`id` BIGINT**(20) **NOT NULL AUTO\_INCREMENT**,  
 **`branch\_id` BIGINT**(20) **NOT NULL**,  
 **`xid` VARCHAR**(100) **NOT NULL**,  
 **`context` VARCHAR**(128) **NOT NULL**,  
 **`rollback\_info` LONGBLOB NOT NULL**,  
 **`log\_status` INT**(11) **NOT NULL**,  
 **`log\_created` DATETIME NOT NULL**,  
 **`log\_modified` DATETIME NOT NULL**,  
 **`ext` VARCHAR**(100) **DEFAULT NULL**,  
 **PRIMARY KEY** (**`id`**),  
 **UNIQUE KEY** `ux\_undo\_log` (**`xid`**, **`branch\_id`**)  
) **ENGINE** = INNODB  
 **AUTO\_INCREMENT** = 1  
 **DEFAULT CHARSET** = utf8;

## Fescar工程搭建



如上图，在所有微服务工程中，不一定所有工程都需要使用分布式事务，我们可以创建一个独立的分布式事务工程，指定微服务需要支持分布式事务的时候，直接依赖独立的分布式工程即可。

搭建一个changgou-transaction-fescar提供fescar分布式事务支持。

注：此工程搭建步骤与我们刚才入门案例一致，代码略。



## 分布式事务测试

### 相关微服务添加依赖

因为所有微服务都有可能使用分布式事务，所以我们可以在每个微服务工程中添加fescar的依赖，当然，搜索工程排除，因为它不需要依赖数据库，代码如下：

<!--fescar依赖-->

<**dependency**>

<**groupId**>com.changgou</**groupId**>

<**artifactId**>changgou-transaction-fescar</**artifactId**>

<**version**>1.0-SNAPSHOT</**version**>

</**dependency**>

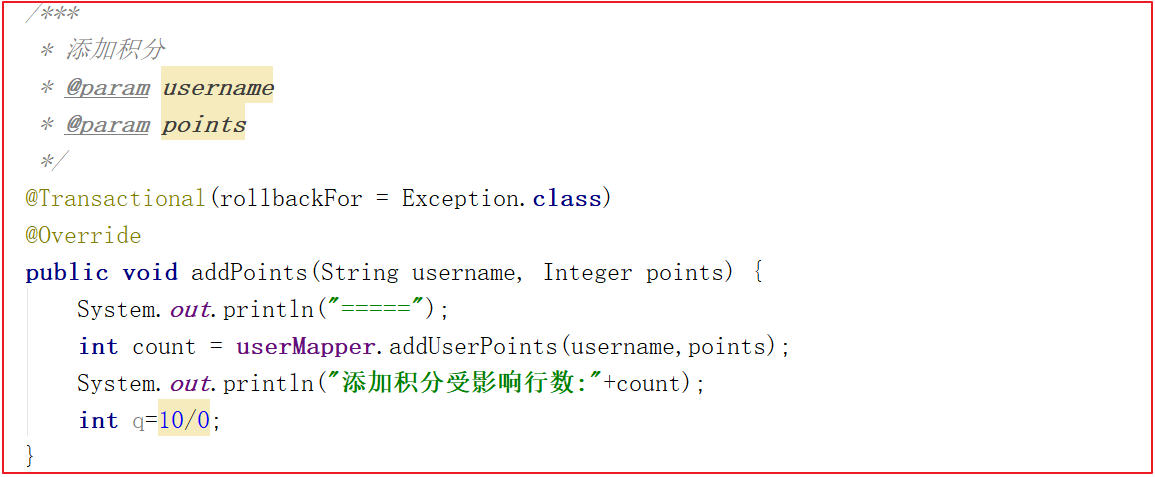
### 测试

在订单微服务的OrderServiceImpl的add方法上增加@GlobalTransactional注解，代码如下：



这里涉及到几个微服务的调用，我们先查询下数据库数据，然后再测试一次，如果输出添加订单完成和库存减少完毕则表明订单微服务和商品微服务的事务已经完成，这时候我们在添加积分的方法中制造一个异常，如果积分添加异常，而商品微服务中数据没发生变化，则表明分布式事务控制成功。

修改用户微服务，在添加用户积分的地方制造异常，代码如下：



测试前后结果一致，主要是查看库存与积分在出错情况下有没有回滚。

