

# 第13章 分布式事务

## 学习目标

- 理解什么是事务
- 理解什么是分布式事务

1. 多个应用
2. 多个数据库

- 理解CAP定理

1. CAP不能三者同时成立

- 能说出相关的分布式事务解决方案(重点理解)

1. 2PC-JTA分布式事务
2. 本地消息-业务库中添加对应的消息表和业务耦合实现
3. MQ事务消息-RocketMQ
4. Seata

- 理解Seata工作流程

1. AT模式-表(重点理解)
2. TCC模式-代码补偿机制

- 能实现Seata案例

1. Seata使用案例

- 订单事务控制

1. 下单(订单微服务)->Feign:商品微服务->库存递减
2. ->Feign:用户微服务->增加积分

## 1 分布式事务介绍

### 1.1 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[由当前业务逻辑多个不同操作构成]。

事务拥有以下四个特性, 习惯上被称为ACID特性:

**原子性(Atomicity):** 事务作为一个整体被执行, 包含在其中的对数据库的操作要么全部被执行, 要么都不执行。记录之前的版本, 允许回滚。

**一致性(Consistency):** 一致性是指事务使得系统从一个一致的状态转换到另一个一致状态。事务的一致性决定了一个系统设计和实现的复杂度, 也导致了事务的不同隔离级别。事务开始和结束之间的中间状态不会被其他事务看到。

**隔离性(Isolation):** 多个事务并发执行时, 并发事务之间互相影响的程度, 比如一个事务会不会读取到另一个未提交的事务修改的数据。适当的破坏一致性来提升性能与并行度

**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。每一次的事务提交后就会保证不会丢失。

## 1.2 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)。



## 1.3 什么是分布式事务

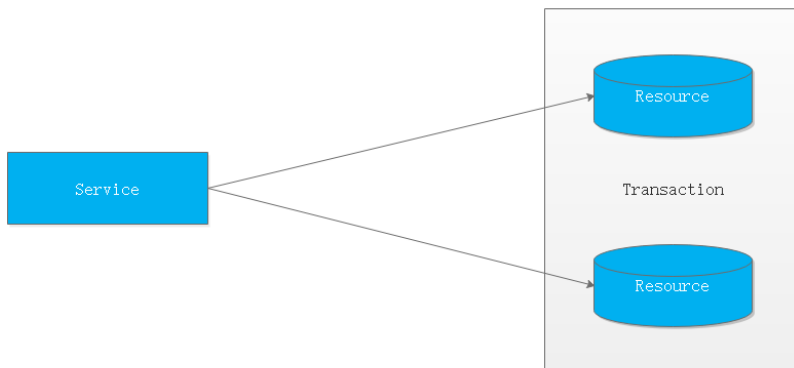
分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上, 且属于不同的应用, 分布式事务需要保证这些操作要么全部成功, 要么全部失败。本质上来说, 分布式事务就是为了保证不同数据库的数据一致性。

## 1.4 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)或者微服务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。

### 1.4.1 单一服务分布式事务

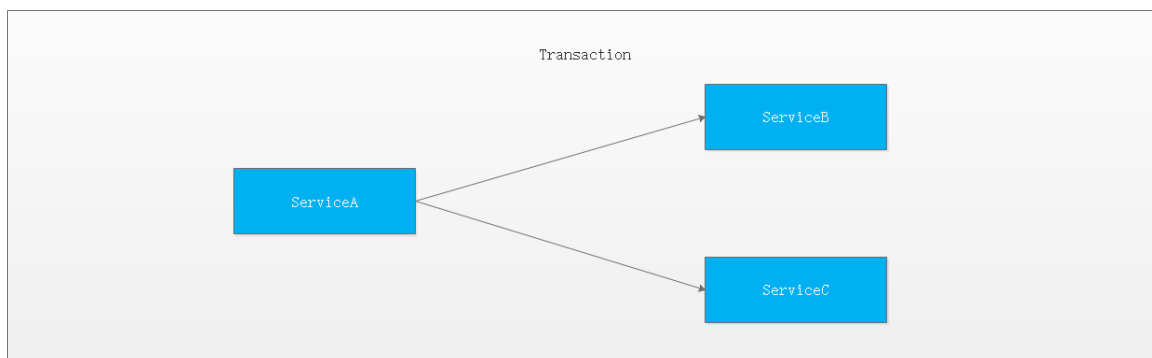
最早的分布式事务应用架构很简单, 不涉及服务间的访问调用, 仅仅是服务内操作涉及到对多个数据库资源的访问。



### 1.4.2 多服务分布式事务

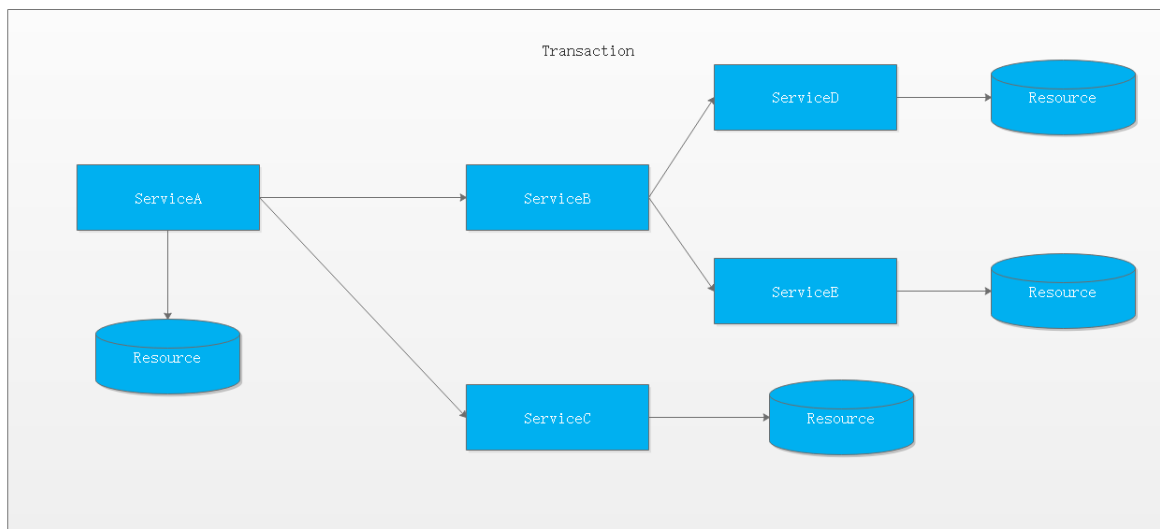
当一个服务操作访问不同的数据库资源，又希望对它们的访问具有事务特性时，就需要采用分布式事务来协调所有的事务参与者。

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 to 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：



### 1.4.3 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。

事务的作用：

- 1 | 保证每个事务的数据一致性。

## 1.5 CAP定理

CAP 定理，又被叫作布鲁尔定理。对于设计分布式系统(不仅仅是分布式事务)的架构师来说，CAP 就是你的入门理论。

**C (一致性):** 对某个指定的客户端来说，读操作能返回最新的写操作。

对于数据分布在不同节点上的数据来说，如果在某个节点更新了数据，那么在其他节点如果都能读取到这个最新的数据，那么就称为强一致，如果有某个节点没有读取到，那就是分布式不一致。

**A (可用性):** 非故障的节点在合理的时间内返回合理的响应(不是错误和超时的响应)。可用性的两个关键一个是合理的时间，一个是合理的响应。

合理的时间指的是请求不能无限被阻塞，应该在合理的时间给出返回。合理的响应指的是系统应该明确返回结果并且结果是正确的，这里的正确指的是比如应该返回 50，而不是返回 40。

**P (网络分区容错性):** 当出现网络分区后，系统能够继续工作。打个比方，这里集群有多台机器，有台机器网络出现了问题，但是这个集群仍然可以正常工作。

熟悉 CAP 的人都知道，三者不能共有，如果感兴趣可以搜索 CAP 的证明，在分布式系统中，网络无法 100% 可靠，分区其实是一个必然现象。

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(这里说的一致性是指强一致性)，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。

## 2 分布式事务解决方案

1.XA两段提交(低效率)-21 XA JTA分布式事务解决方案

2.TCC三段提交(3段,高效率[不推荐(补偿代码)])

3.本地消息表(MQ+Table)

4.事务消息(RocketMQ[alibaba])

5.Seata(alibaba)

6.RabbitMQ的ACK机制实现分布式事务(作业)

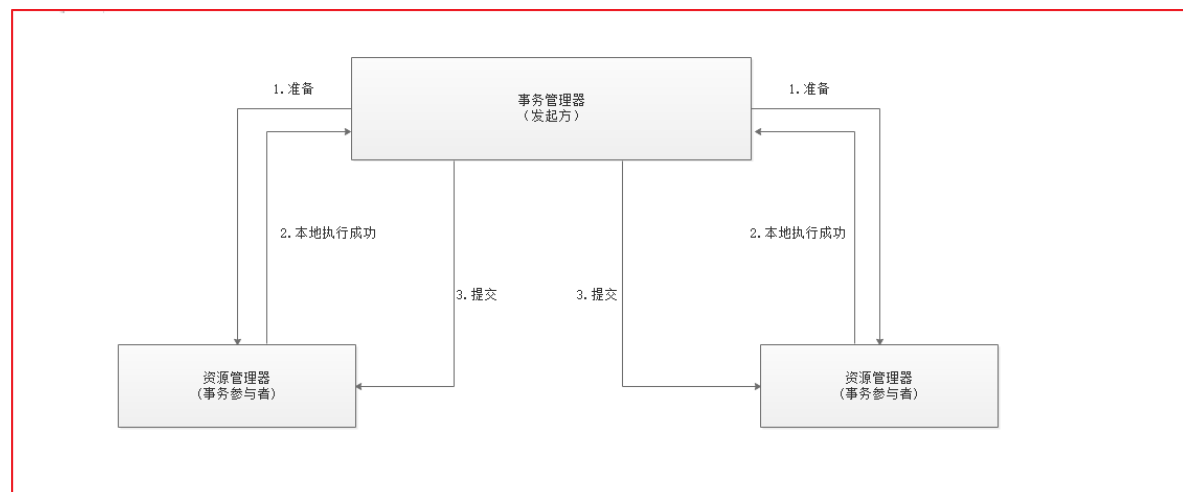
### 2.1 基于XA协议的两阶段提交(2PC)

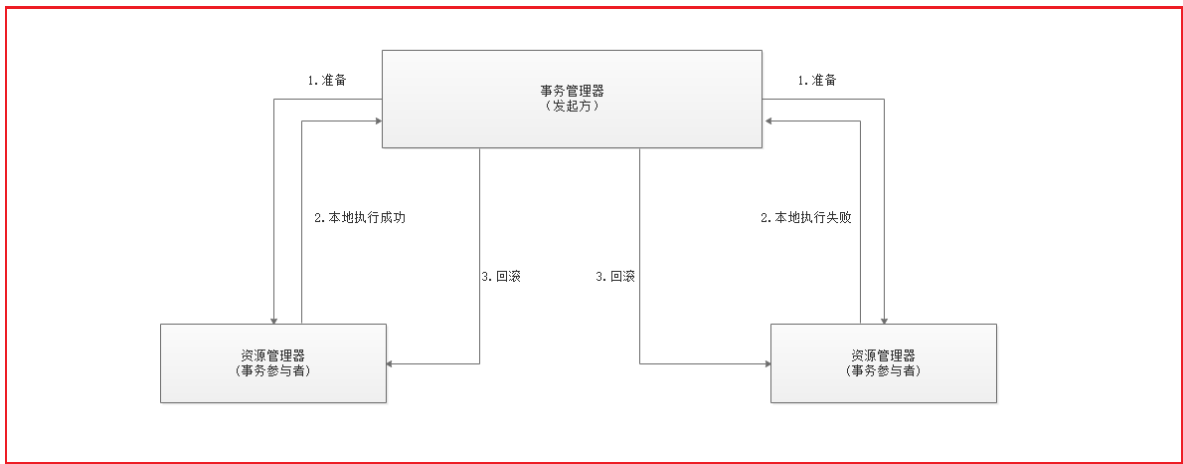
两阶段提交协议(Two Phase Commitment Protocol)中，涉及到两种角色

一个事务协调者 (coordinator)：负责协调多个参与者进行事务投票及提交(回滚) 多个事务参与者 (participants)：即本地事务执行者

总共处理步骤有两个 (1) 投票阶段 (voting phase)：协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。参与者将告知协调者自己的决策：同意（事务参与者本地事务执行成功，但未提交）或取消（本地事务执行故障）； (2) 提交阶段 (commit phase)：收到参与者的通知后，协调者再向参与者发出通知，根据反馈情况决定各参与者是否要提交还是回滚；

如果所示 1-2为第一阶段，2-3为第二阶段





如果任一资源管理器在第一阶段返回准备失败，那么事务管理器会要求所有资源管理器在第二阶段执行回滚操作。通过事务管理器的两阶段协调，最终所有资源管理器要么全部提交，要么全部回滚，最终状态都是一致的

**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。

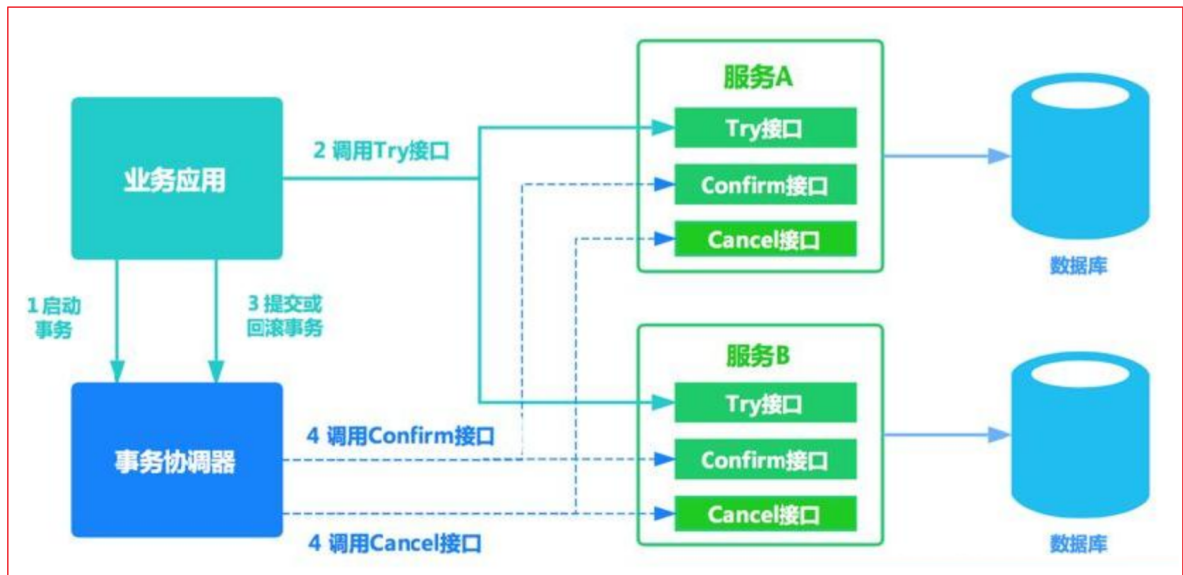
**缺点：** 牺牲了可用性，对性能影响较大，不适合高并发高性能场景，如果分布式系统跨接口调用，目前.NET 界还没有实现方案。

## 2.2 补偿(代码补偿)事务 (TCC) -3PC

TCC 将事务提交分为 Try(method1) - Confirm(method2) - Cancel(method3) 3个操作。其和两阶段提交有点类似，Try为第一阶段，Confirm - Cancel为第二阶段，是一种应用层面侵入业务的两阶段提交。

操作方法	含义
Try	预留业务资源/数据效验-尝试检查当前操作是否可执行
Confirm	确认执行业务操作，实际提交数据，不做任何业务检查，try成功，confirm必定成功，需保证幂等
Cancel	取消执行业务操作，实际回滚数据，需保证幂等

其核心在于将业务分为两个操作步骤完成。不依赖 RM 对分布式事务的支持，而是通过对业务逻辑的分解来实现分布式事务。



例如：A要向B转账，思路大概是：

- 1 假设用户user表中有两个字段：可用余额(available\_money)、冻结余额(frozen\_money)
- 2 A扣钱对应服务A(ServiceA)
- 3 B加钱对应服务B(ServiceB)
- 4 转账订单服务(OrderService)
- 5 业务转账方法服务(BusinessService)

ServiceA, ServiceB, OrderService都需分别实现try(), confirm(), cacle()方法，方法对应业务逻辑如下

操作方法	ServiceA	ServiceB	OrderService
try()	校验余额(并发控制) 冻结余额+1000 余额-1000	冻结余额+1000	创建转账订单，状态待转账
confirm()	冻结余额-1000		状态变为转账成功
cacle()	冻结余额-1000 余额+1000		状态变为转账失败

其中业务调用方BusinessService中就需要调用 ServiceA.try() ServiceB.try() OrderService.try()

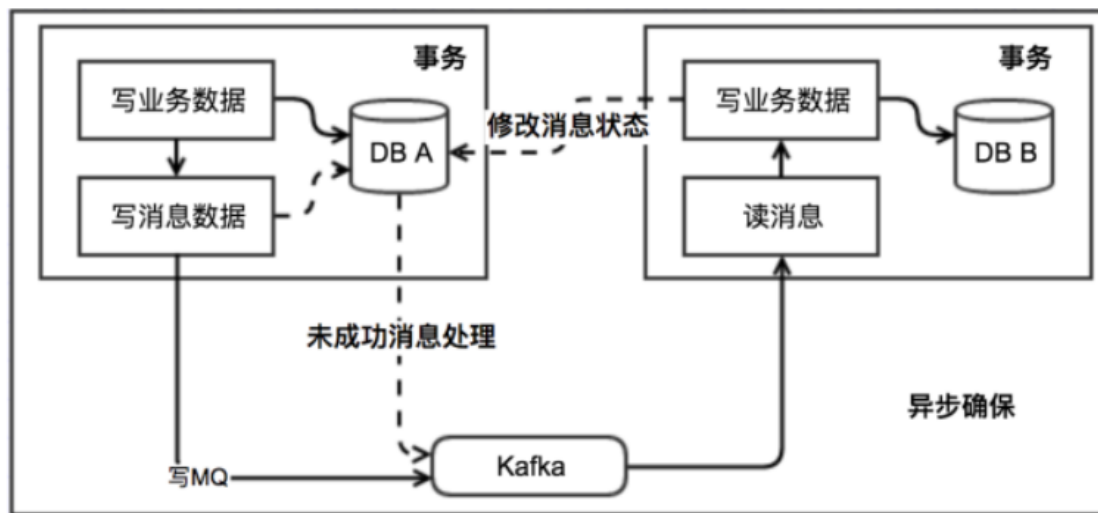
- 1、当所有try()方法均执行成功时，对全局事物进行提交，即由事物管理器调用每个微服务的confirm()方法
- 2、当任意一个方法try()失败(预留资源不足，抑或网络异常，代码异常等任何异常)，由事物管理器调用每个微服务的cacle()方法对全局事务进行回滚

**优点：**跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

**缺点：**缺点还是比较明显的，在2,3步中都有可能失败。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。还存在非幂等问题。

## 2.3 本地消息表（异步确保）- 事务最终一致性

本地消息表这种实现方式应该是业界使用最多的，其核心思想是将分布式事务拆成本地事务进行处理，这种思路是来源于ebay。我们可以从下面的流程图中看出其中的一些细节：



基本思路就是：

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

这种方案遵循BASE理论，采用的是最终一致性，笔者认为这是这几种方案里面比较适合实际业务场景的，即不会出现像2PC那样复杂的实现(当调用链很长的时候，2PC的可用性是非常低的)，也不会像TCC那样可能出现确认或者回滚不了的情况。

**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。在 .NET 中有现成的解决方案。

**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。

只要用到了MQ，都是数据可以不用强一致的操作。

## 2.4 MQ 事务消息-最终一致

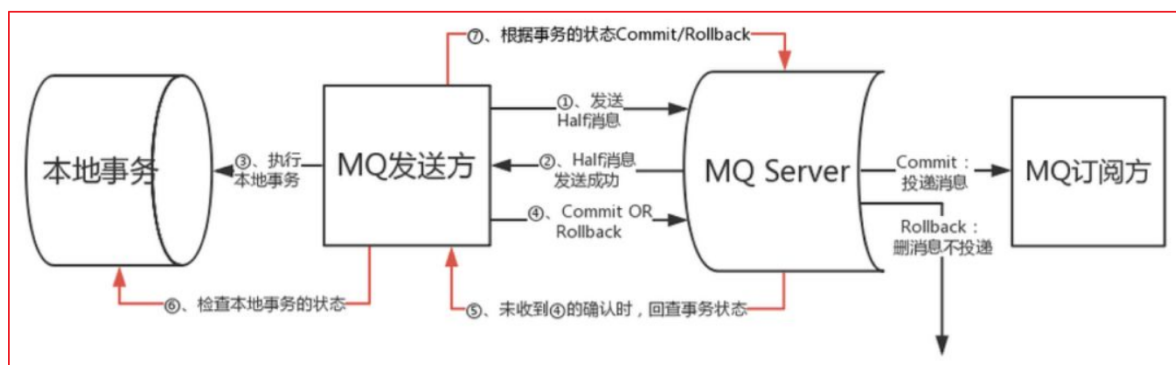
有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持（RabbitMQ、Kafka基于ACK机制）。

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。

作业：基于RabbitMQ的ACK机制实现分布式事务方案。

## 2.5 Seata 2PC

2019年1月，阿里巴巴中间件团队发起了开源项目 [Fescar](#) (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**，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。

### 2.5.1 Seata介绍

解决分布式事务问题，有两个设计初衷

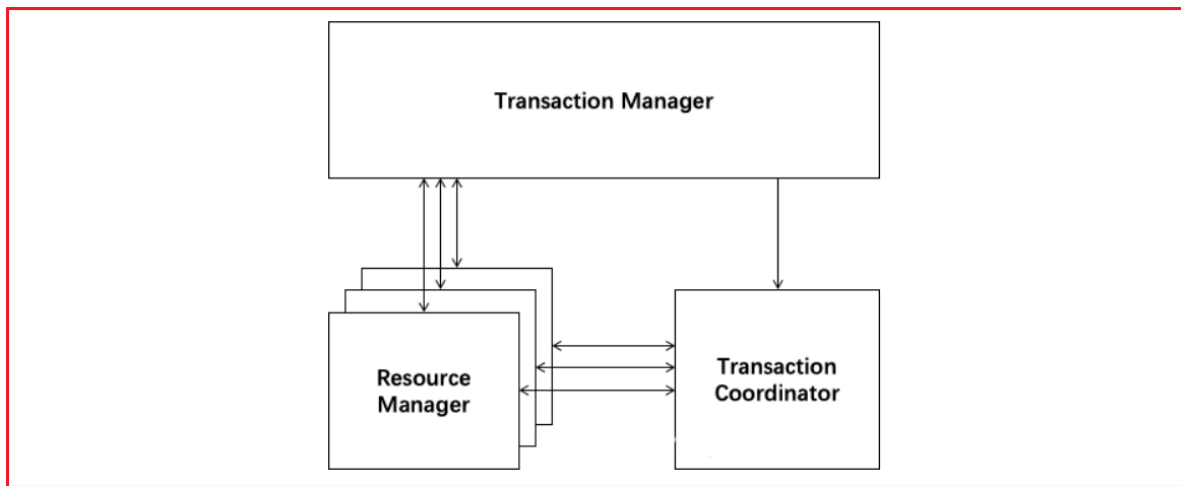
**对业务无侵入：** 即减少技术架构上的微服务化所带来的分布式事务问题对业务的侵入 **高性能：** 减少分布式事务解决方案所带来的性能消耗(2PC)

seata中有两种常见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(强一致性)，当然也包括多服务下的多 DB 数据访问一致性问题 2PC-改进
- 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

### 2.5.2 AT模式

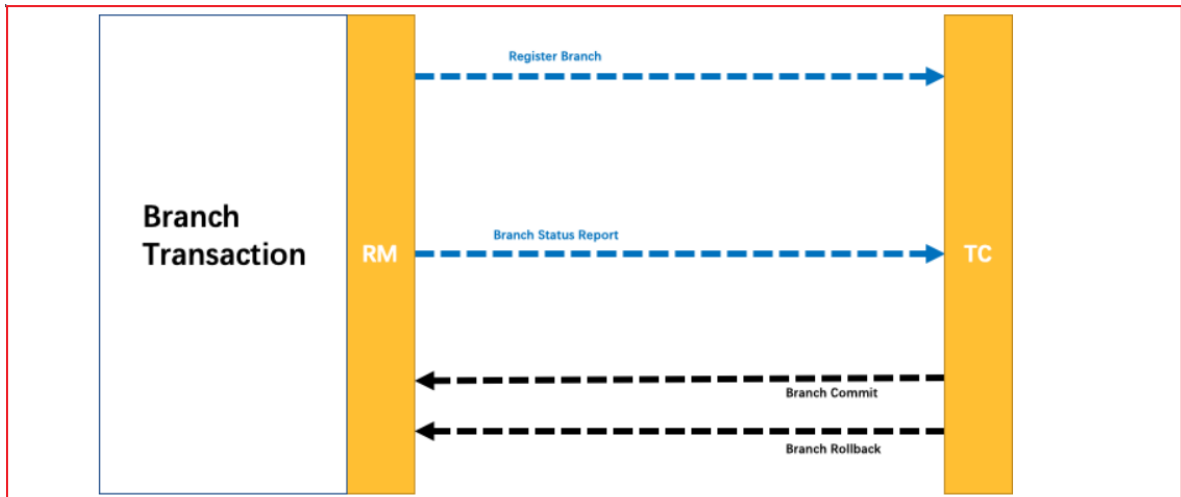
Seata AT模式是基于XA事务演进而来的一个分布式事务中间件，XA是一个基于数据库实现的分布式事务协议，本质上和两阶段提交一样，需要数据库支持，Mysql5.6以上版本支持XA协议，其他数据库如Oracle，DB2也实现了XA接口



解释：

**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。

协调执行流程如下：

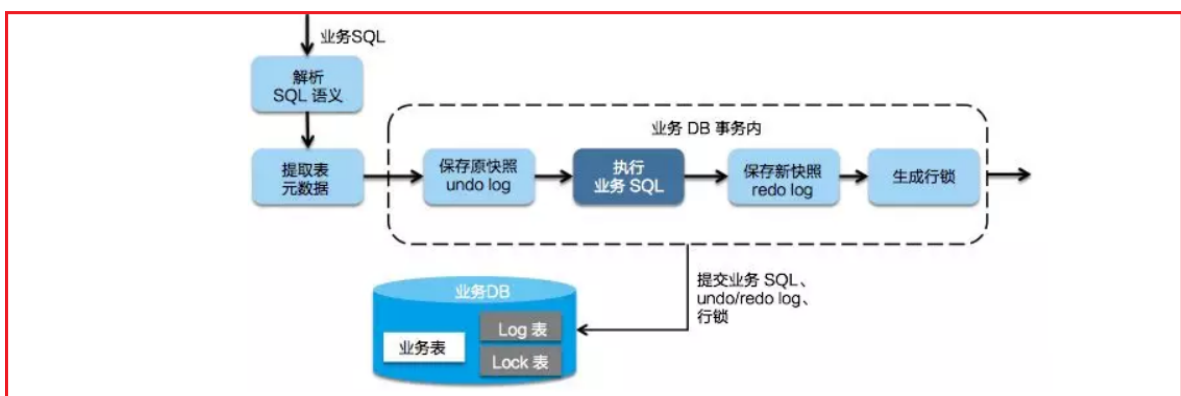


Branch就是指的分布式事务中每个独立的本地局部事务。

## 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**

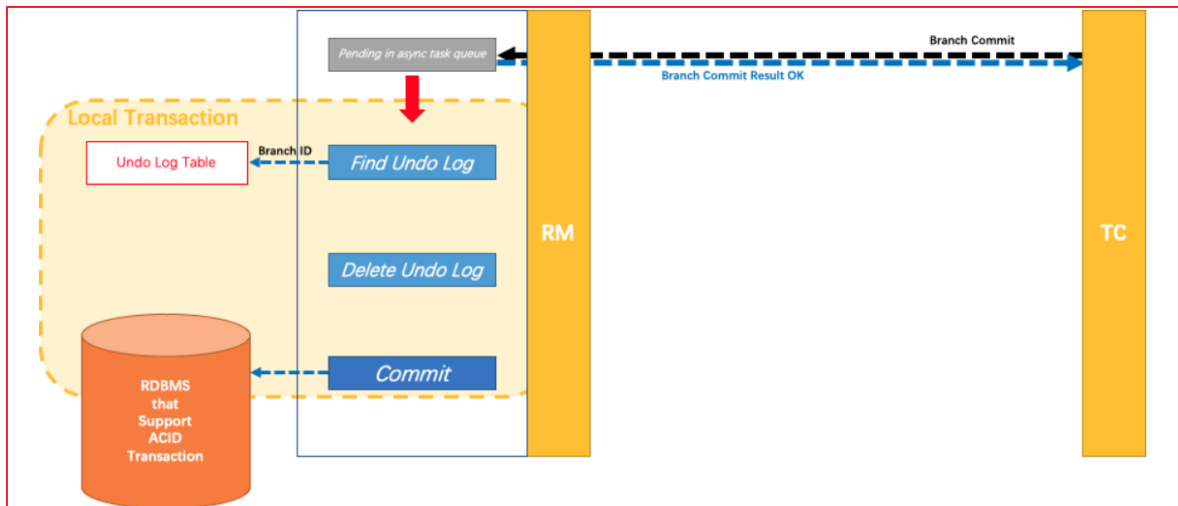


基于这样的机制，分支的本地事务便可以在全局事务的第一阶段提交，并马上释放本地事务锁定的资源。这也是Seata和XA事务的不同之处，两阶段提交往往对资源的锁定需要持续到第二阶段实际的提交或者回滚操作，而有了回滚日志之后，可以在第一阶段释放对资源的锁定，降低了锁范围，提高效率，即使第二阶段发生异常需要回滚，只需找对undolog中对应数据并反解析成sql来达到回滚目的。

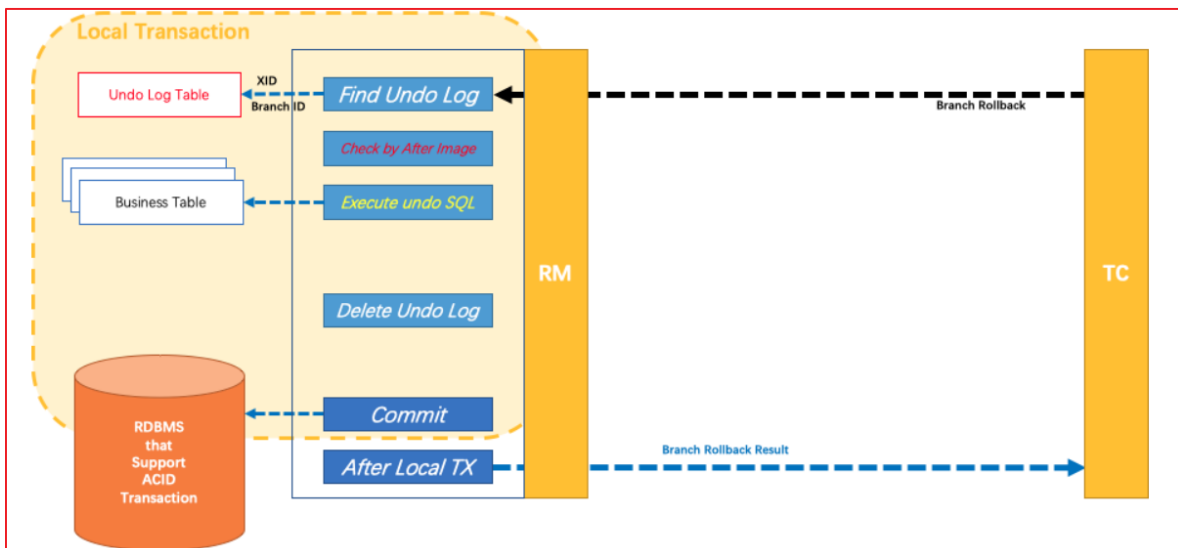
同时Seata通过代理数据源将业务sql的执行解析成undolog来与业务数据的更新同时入库，达到了对业务无侵入的效果。

## 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。

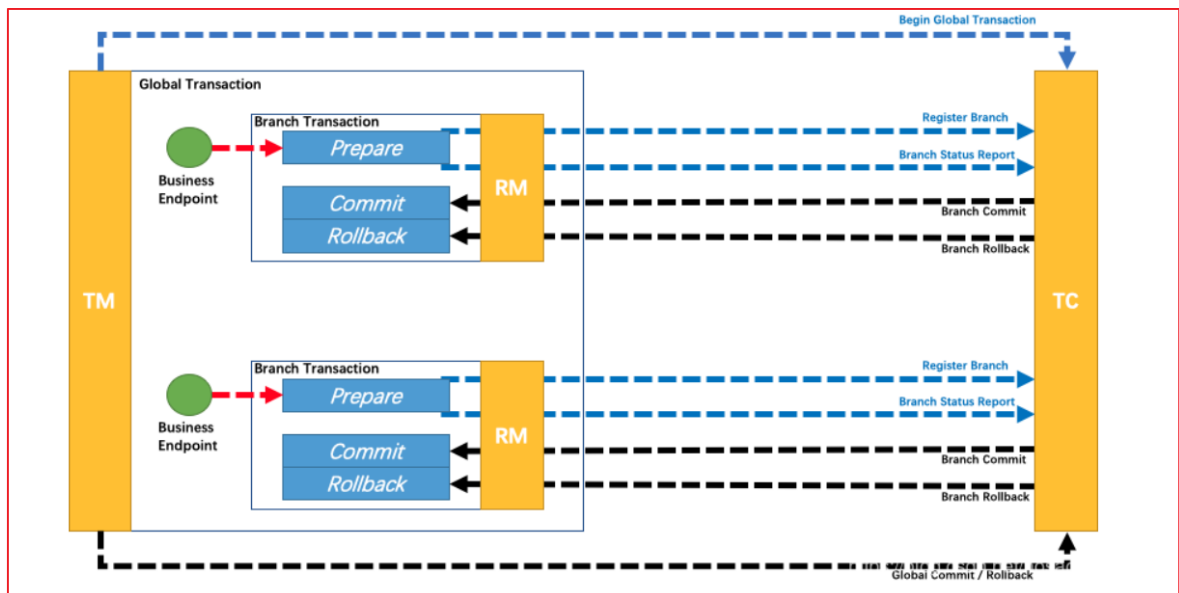


如果决议是全局回滚，RM 收到协调器发来的回滚请求，通过 XID 和 Branch ID 找到相应的回滚日志记录，通过回滚记录生成反向的更新 SQL 并执行，以完成分支的回滚。



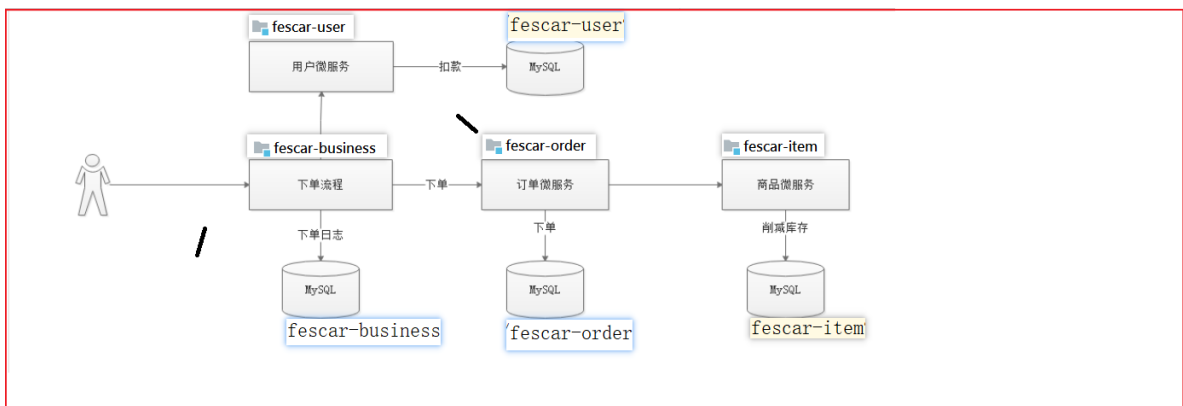
## 2.5.3 TCC模式

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。



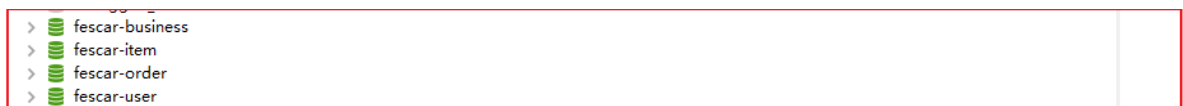
## 3 Seata案例

### 3.1 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 day13\seata\案例SQL脚本 数据库脚本导入到数据库中。



## 3.2 案例实现

### 3.2.1 父工程

搭建 seata-parent,为了适应我们易购工程的分布式事务，我们这里的父工程引入和易购工程一样的依赖包。

pom.xml依赖如下：



```
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5         <modelVersion>4.0.0</modelVersion>
6
7         <groupId>com.itheima</groupId>
8         <artifactId>seata-parent</artifactId>
9         <version>1.0-SNAPSHOT</version>
10        <description>Seata案例父工程</description>
11        <modules>
12            <module>seata-api</module>
13            <module>seata-eureka</module>
14            <module>seata-item</module>
15            <module>seata-user</module>
16            <module>seata-order</module>
17            <module>seata-business</module>
18        </modules>
19
20        <parent>
21            <groupId>org.springframework.boot</groupId>
22            <artifactId>spring-boot-starter-parent</artifactId>
23            <version>2.1.4.RELEASE</version>
24        </parent>
25        <packaging>pom</packaging>
26
27        <!--跳过测试-->
28        <properties>
29            <skipTests>true</skipTests>
30        </properties>
31
32        <!--依赖包-->
33        <dependencies>
34            <!--测试包-->
35            <dependency>
36                <groupId>org.springframework.boot</groupId>
37                <artifactId>spring-boot-starter-test</artifactId>
38            </dependency>
39
40            <!--fastjson-->
41            <dependency>
42                <groupId>com.alibaba</groupId>
43                <artifactId>fastjson</artifactId>
44                <version>1.2.51</version>
45            </dependency>
46
47            <!--鉴权-->
48            <dependency>
49                <groupId>io.jsonwebtoken</groupId>
50                <artifactId>jjwt</artifactId>
51                <version>0.9.0</version>
52            </dependency>
53
54            <!--web起步依赖-->
55            <dependency>
56                <groupId>org.springframework.boot</groupId>
57                <artifactId>spring-boot-starter-web</artifactId>
58            </dependency>
59
60            <!-- redis 使用-->
```

```

61     <dependency>
62         <groupId>org.springframework.boot</groupId>
63         <artifactId>spring-boot-starter-data-redis</artifactId>
64     </dependency>
65
66     <!--eureka-client-->
67     <dependency>
68         <groupId>org.springframework.cloud</groupId>
69         <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
70     </dependency>
71
72     <!--openfeign-->
73     <dependency>
74         <groupId>org.springframework.cloud</groupId>
75         <artifactId>spring-cloud-starter-openfeign</artifactId>
76     </dependency>
77
78     <!--微信支付-->
79     <dependency>
80         <groupId>com.github.wxpay</groupId>
81         <artifactId>wxpay-sdk</artifactId>
82         <version>0.0.3</version>
83     </dependency>
84
85     <!--httpClient支持,微信支付-->
86     <dependency>
87         <groupId>org.apache.httpcomponents</groupId>
88         <artifactId>httpClient</artifactId>
89     </dependency>
90
91     <!--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-->
92     <dependency>
93         <groupId>tk.mybatis</groupId>
94         <artifactId>mapper-spring-boot-starter</artifactId>
95         <version>2.0.4</version>
96     </dependency>
97
98     <!--MySQL数据库驱动-->
99     <dependency>
100         <groupId>mysql</groupId>
101         <artifactId>mysql-connector-java</artifactId>
102     </dependency>
103
104     <!--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)-->
105     <dependency>
106         <groupId>com.github.pagehelper</groupId>
107         <artifactId>pagehelper-spring-boot-starter</artifactId>
108         <version>1.2.3</version>
109     </dependency>
110 </dependencies>
111
112 <dependencyManagement>
113     <dependencies>
114         <dependency>
115             <groupId>org.springframework.cloud</groupId>

```

```

116         <artifactId>spring-cloud-dependencies</artifactId>
117         <version>Greenwich.SR1</version>
118         <type>pom</type>
119         <scope>import</scope>
120     </dependency>
121 </dependencies>
122 </dependencyManagement>
123 </project>

```

### 3.2.2 Eureka

搭建工程 seata-eureka

(1)pom.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5      <parent>
6          <artifactId>seata-parent</artifactId>
7          <groupId>com.itheima</groupId>
8          <version>1.0-SNAPSHOT</version>
9      </parent>
10     <modelVersion>4.0.0</modelVersion>
11     <description>eureka</description>
12     <artifactId>seata-eureka</artifactId>
13
14     <!-- 依赖包 -->
15     <dependencies>
16         <dependency>
17             <groupId>org.springframework.cloud</groupId>
18             <artifactId>spring-cloud-starter-netflix-eureka-
server</artifactId>
19         </dependency>
20     </dependencies>
21 </project>

```

(2)启动类

```

1  @SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
2  @EnableEurekaServer //开启Eureka服务
3  public class EurekaApplication {
4
5      /**
6       * 加载启动类，以启动类为当前SpringBoot的配置标准
7       * @param args
8       */
9      public static void main(String[] args) {
10         SpringApplication.run(EurekaApplication.class,args);
11     }
12 }

```

(3)application.yml

```
1 server:
2   port: 7001
3 eureka:
4   instance:
5     hostname: 127.0.0.1
6   client:
7     register-with-eureka: false #是否将自己注册到eureka中
8     fetch-registry: false      #是否从eureka中获取信息
9     service-url:
10      defaultZone: http://127.0.0.1:7001/eureka/
11 spring:
12   application:
13     name: eureka
```

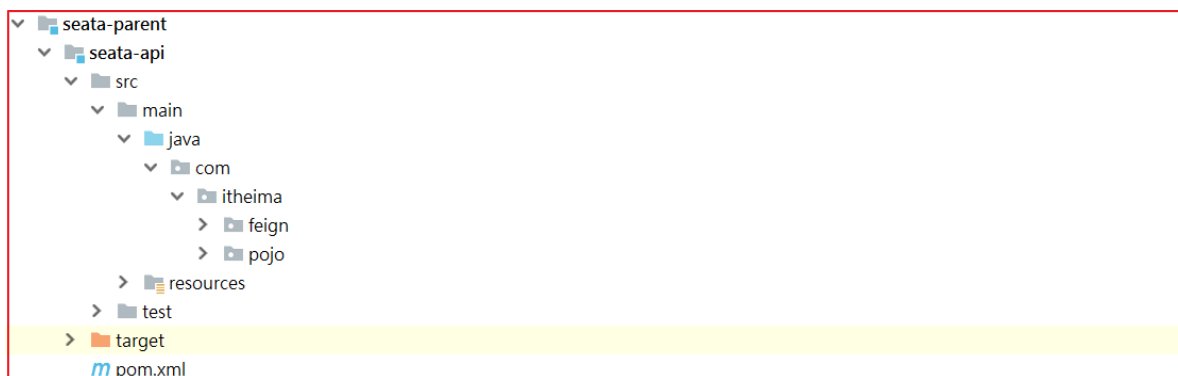
### 3.2.3 公共工程

将所有数据库对应的Pojo/Feign抽取出一个公共工程 seata-api ,在该工程中导入依赖:

(1)pom.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5     http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <parent>
7     <artifactId>seata-parent</artifactId>
8     <groupId>com.itheima</groupId>
9     <version>1.0-SNAPSHOT</version>
10   </parent>
11   <modelVersion>4.0.0</modelVersion>
12   <description>所有微服务工程公共的POJO和Feign</description>
13   <artifactId>seata-api</artifactId>
14 </project>
```

将Pojo、feign导入到工程中





### 3.2.4 商品微服务

创建 `seata-item` 微服务，在该工程中实现库存削减。

(1)pom.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5         http://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <parent>
7         <artifactId>seata-parent</artifactId>
8         <groupId>com.itheima</groupId>
9         <version>1.0-SNAPSHOT</version>
10    </parent>
11    <modelVersion>4.0.0</modelVersion>
12    <artifactId>seata-item</artifactId>
13    <dependencies>
14        <dependency>
15            <groupId>com.itheima</groupId>
16            <artifactId>seata-api</artifactId>
17            <version>1.0-SNAPSHOT</version>
18        </dependency>
19    </dependencies>
20 </project>
```

(2)Dao

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

```
1 public interface ItemInfoMapper extends Mapper<ItemInfo> {
2 }
```

(3)Service

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

```
1 public interface ItemInfoService {
2
3     /**
4      * 库存递减
5      * @param id
6      * @param count
7      */
8     void decrCount(int id, int count);
9 }
```

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

```
1 @Service
2 public class ItemInfoServiceImpl implements ItemInfoService {
```

```

3
4     @Autowired
5     private ItemInfoMapper itemInfoMapper;
6
7     /**
8      * 库存递减
9      * @param id
10     * @param count
11     */
12     @Transactional(rollbackFor = Exception.class)
13     @Override
14     public void decrCount(int id, int count) {
15         //查询商品信息
16         ItemInfo itemInfo = itemInfoMapper.selectByPrimaryKey(id);
17         itemInfo.setCount(itemInfo.getCount()-count);
18         int dcount = itemInfoMapper.updateByPrimaryKeySelective(itemInfo);
19         System.out.println("库存递减受影响行数: "+dcount);
20     }
21 }

```

#### (4)Controller

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

```

1  @RestController
2  @RequestMapping("/itemInfo")
3  @CrossOrigin
4  public class ItemInfoController {
5
6      @Autowired
7      private ItemInfoService itemInfoService;
8
9      /**
10     * 库存递减
11     * @param id
12     * @param count
13     * @return
14     */
15     @PostMapping(value = "/decrCount")
16     public String decrCount(@RequestParam(value = "id") int id,
17     @RequestParam(value = "count") int count){
18         //库存递减
19         itemInfoService.decrCount(id,count);
20         return "success";
21     }
22 }

```

#### (5)启动类

创建 `com.itheima.ItemApplication` 代码如下:

```

1  @SpringBootApplication
2  @EnableEurekaClient
3  @EnableFeignClients(basePackages = {"com.itheima.feign"})
4  @MapperScan(basePackages = {"com.itheima.dao"})
5  public class ItemApplication {
6
7      public static void main(String[] args) {
8          SpringApplication.run(ItemApplication.class,args);
9      }
10 }

```

(6)application.yml

创建applicatin.yml,配置如下:

```

1  server:
2      port: 18082
3  spring:
4      application:
5          name: item
6      datasource:
7          driver-class-name: com.mysql.jdbc.Driver
8          url: jdbc:mysql://192.168.211.132:3306/fescar-item?
9              useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
10         username: root
11         password: 123456
12      main:
13          allow-bean-definition-overriding: true
14  eureka:
15      client:
16          service-url:
17              defaultZone: http://127.0.0.1:7001/eureka/
18      instance:
19          prefer-ip-address: true

```

### 3.2.5 用户微服务

创建 seata-user 微服务，并引入公共工程依赖。

(1)pom.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5      http://maven.apache.org/xsd/maven-4.0.0.xsd">
6      <parent>
7          <artifactId>seata-parent</artifactId>
8          <groupId>com.itheima</groupId>
9          <version>1.0-SNAPSHOT</version>
10     </parent>
11     <modelVersion>4.0.0</modelVersion>

```

```

12     <artifactId>seata-user</artifactId>
13
14     <dependencies>
15         <dependency>
16             <groupId>com.itheima</groupId>
17             <artifactId>seata-api</artifactId>
18             <version>1.0-SNAPSHOT</version>
19         </dependency>
20     </dependencies>
21
22 </project>

```

## (2)Dao

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

```

1 public interface UserInfoMapper extends Mapper<UserInfo> {
2 }

```

## (3)Service

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

```

1 public interface UserInfoService {
2
3     /**
4      * 账户金额递减
5      * @param username
6      * @param money
7      */
8     void decrMoney(String username, int money);
9 }

```

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

```

1 @Service
2 public class UserInfoServiceImpl implements UserInfoService {
3
4     @Autowired
5     private UserInfoMapper userInfoMapper;
6
7     /**
8      * 账户金额递减
9      * @param username
10     * @param money
11     */
12     @Transactional(rollbackFor = Exception.class)
13     @Override
14     public void decrMoney(String username, int money) {
15         UserInfo userInfo = userInfoMapper.selectByPrimaryKey(username);
16         userInfo.setMoney(userInfo.getMoney()-money);
17         int count = userInfoMapper.updateByPrimaryKeySelective(userInfo);
18         System.out.println("添加用户受影响行数: "+count);
19     }
20 }

```

```

19     int q=10/0;
20     }
21 }

```

#### (4)Controller

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

```

1  @RestController
2  @RequestMapping("/userInfo")
3  @CrossOrigin
4  public class UserInfoController {
5
6      @Autowired
7      private UserInfoService userInfoService;
8
9      /**
10       * 账户余额递减
11       * @param username
12       * @param money
13       */
14      @PostMapping(value = "/add")
15      public String decrMoney(@RequestParam(value = "username") String
username, @RequestParam(value = "money") int money){
16          userInfoService.decrMoney(username,money);
17          return "success";
18      }
19 }

```

#### (5)启动类

创建 `com.itheima.UserApplication` , 代码如下:

```

1  @SpringBootApplication
2  @EnableEurekaClient
3  @EnableFeignClients(basePackages = {"com.itheima.feign"})
4  @MapperScan(basePackages = {"com.itheima.dao"})
5  public class UserApplication {
6
7      public static void main(String[] args) {
8          SpringApplication.run(UserApplication.class,args);
9      }
10 }

```

#### (6)application.yml

创建application.yml配置如下:

```

1  server:
2      port: 18084
3  spring:
4      application:
5          name: user

```

```

6     datasource:
7         driver-class-name: com.mysql.jdbc.Driver
8         url: jdbc:mysql://192.168.211.132:3306/fescar-user?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
9         username: root
10        password: 123456
11    main:
12        allow-bean-definition-overriding: true
13    eureka:
14        client:
15            service-url:
16                defaultZone: http://127.0.0.1:7001/eureka
17        instance:
18            prefer-ip-address: true

```

### 3.2.6 订单微服务

创建订单工程 `seata-order`，在订单微服务中实现调用商品微服务递减库存。

(1)pom.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5      <parent>
6          <artifactId>seata-parent</artifactId>
7          <groupId>com.itheima</groupId>
8          <version>1.0-SNAPSHOT</version>
9      </parent>
10     <modelVersion>4.0.0</modelVersion>
11     <artifactId>seata-order</artifactId>
12
13     <dependencies>
14         <dependency>
15             <groupId>com.itheima</groupId>
16             <artifactId>seata-api</artifactId>
17             <version>1.0-SNAPSHOT</version>
18         </dependency>
19     </dependencies>
20 </project>

```

(2)Dao

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

```

1  public interface OrderInfoMapper extends Mapper<OrderInfo> {
2  }

```

(3)Service

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

```
1 public interface OrderInfoService {
2
3     /**
4      * 添加订单
5      * @param username
6      * @param id
7      * @param count
8      */
9     void add(String username, int id, int count);
10 }
```

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

```
1 @Service
2 public class OrderInfoServiceImpl implements OrderInfoService {
3
4     @Autowired
5     private OrderInfoMapper orderInfoMapper;
6
7     @Autowired
8     private ItemInfoFeign itemInfoFeign;
9
10    /**
11     * 添加订单
12     * @param username
13     * @param id
14     * @param count
15     */
16    @Transactional
17    @Override
18    public void add(String username, int id, int count) {
19        //添加订单
20        OrderInfo orderInfo = new OrderInfo();
21        orderInfo.setMessage("生成订单");
22        orderInfo.setMoney(10);
23        int icount = orderInfoMapper.insertSelective(orderInfo);
24        System.out.println("添加订单受影响函数: "+icount);
25
26        //递减库存
27        itemInfoFeign.decrCount(id, count);
28    }
29 }
```

### (3)Controller

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

```
1 @RestController
2 @RequestMapping("/orderInfo")
3 @CrossOrigin
4 public class OrderInfoController {
5
```

```

6      @Autowired
7      private OrderInfoService orderInfoService;
8
9      /**
10     * 增加订单
11     * @param username
12     * @param id
13     * @param count
14     */
15     @PostMapping(value = "/add")
16     public String add(@RequestParam(value = "name") String username,
17 @RequestParam(value = "id") int id, @RequestParam(value = "count") int
count){
17         //添加订单
18         orderInfoService.add(username,id,count);
19         return "success";
20     }
21 }

```

#### (4)启动类

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

```

1  @SpringBootApplication
2  @EnableEurekaClient
3  @EnableFeignClients(basePackages = {"com.itheima.feign"})
4  @MapperScan(basePackages = {"com.itheima.dao"})
5  public class OrderApplication {
6
7      public static void main(String[] args) {
8          SpringApplication.run(OrderApplication.class,args);
9      }
10 }

```

#### (5)application.yml配置

```

1  server:
2      port: 18083
3  spring:
4      application:
5          name: order
6      datasource:
7          driver-class-name: com.mysql.jdbc.Driver
8          url: jdbc:mysql://192.168.211.132:3306/fescar-order?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
9          username: root
10         password: 123456
11     main:
12         allow-bean-definition-overriding: true
13     eureka:
14         client:
15             service-url:
16                 defaultZone: http://127.0.0.1:7001/eureka

```



```
17 | instance:
18 |     prefer-ip-address: true
```

### 3.2.7 业务微服务

创建 `seata-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。

(1) pom.xml

```
1 | <?xml version="1.0" encoding="UTF-8"?>
2 | <project xmlns="http://maven.apache.org/POM/4.0.0"
3 |         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 |         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5 |     <parent>
6 |         <artifactId>seata-parent</artifactId>
7 |         <groupId>com.itheima</groupId>
8 |         <version>1.0-SNAPSHOT</version>
9 |     </parent>
10 |    <modelVersion>4.0.0</modelVersion>
11 |    <artifactId>seata-business</artifactId>
12 |
13 |    <dependencies>
14 |        <dependency>
15 |            <groupId>com.itheima</groupId>
16 |            <artifactId>seata-api</artifactId>
17 |            <version>1.0-SNAPSHOT</version>
18 |        </dependency>
19 |    </dependencies>
20 | </project>
```

(2) Dao

创建 `com.itheima.dao.LogInfoMapper` 代码如下：

```
1 | public interface LogInfoMapper extends Mapper<LogInfo> {
2 | }
```

(3) Service

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

```

1 public interface BusinessService {
2
3     /**
4      * 下单
5      * @param username
6      * @param id
7      * @param count
8      */
9     void add(String username, int id, int count);
10 }

```

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

```

1 @Service
2 public class BusinessServiceImpl implements BusinessService {
3
4     @Autowired
5     private OrderInfoFeign orderInfoFeign;
6
7     @Autowired
8     private UserInfoFeign userInfoFeign;
9
10    @Autowired
11    private LogInfoMapper logInfoMapper;
12
13    /**
14     * ④
15     * 下单
16     * @GlobalTransactional:全局事务入口
17     * @param username
18     * @param id
19     * @param count
20     */
21    @Override
22    public void add(String username, int id, int count) {
23        //添加订单日志
24        LogInfo logInfo = new LogInfo();
25        logInfo.setContent("添加订单数据---"+new Date());
26        logInfo.setCreatetime(new Date());
27        int logcount = logInfoMapper.insertSelective(logInfo);
28        System.out.println("添加日志受影响行数: "+logcount);
29
30        //添加订单
31        orderInfoFeign.add(username,id,count);
32
33        //用户账户余额递减
34        userInfoFeign.decrMoney(username,10);
35    }
36 }

```

#### (4)Controller

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

```

1 @RestController
2 @RequestMapping(value = "/business")

```

```

3 public class BusinessController {
4
5     @Autowired
6     private BusinessService businessService;
7
8     /**
9      * 购买商品分布式事务测试
10     * @return
11     */
12     @RequestMapping(value = "/addorder")
13     public String order(){
14         String username="zhangsan";
15         int id=1;
16         int count=5;
17         //下单
18         businessService.add(username,id,count);
19         return "success";
20     }
21 }

```

## (5)启动类

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

```

1 @SpringBootApplication
2 @EnableEurekaClient
3 @EnableFeignClients(basePackages = {"com.itheima.feign"})
4 @MapperScan(basePackages = {"com.itheima.dao"})
5 public class BusinessApplication {
6
7     public static void main(String[] args) {
8         SpringApplication.run(BusinessApplication.class,args);
9     }
10 }

```

## (6)application.yml配置

```

1 server:
2   port: 18081
3 spring:
4   application:
5     name: business
6   datasource:
7     driver-class-name: com.mysql.jdbc.Driver
8     url: jdbc:mysql://192.168.211.132:3306/fescar-business?
9     useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
10    username: root
11    password: 123456
12   main:
13     allow-bean-definition-overriding: true
14   eureka:
15     client:
16       service-url:

```

```

16     defaultZone: http://127.0.0.1:7001/eureka
17     instance:
18         prefer-ip-address: true
19     #读取超时设置
20     ribbon:
21         ReadTimeout: 30000

```

## 3.3 Seata分布式事务

### 3.3.1 Seata事务配置

Seata参考地址: <http://seata.io/zh-cn/docs/overview/what-is-seata.html>

Seata分布式事务步骤:

- 1 1. 引入依赖包
- 2 2. 更换数据源为DataSourceProxy, 主要作用绑定undo\_log表的操作
- 3 3. MyBatis集成Spring的时候, SqlSessionFactoryBean->注入代理数据源DataSourceProxy
- 4 4. TC注册和链接信息配置

#### (1)引入依赖包

修改 seata-api 引入依赖包:

```

1 <dependencies>
2     <!--Seata依赖-->
3     <dependency>
4         <groupId>com.alibaba.cloud</groupId>
5         <artifactId>spring-cloud-alibaba-seata</artifactId>
6         <version>2.1.0.RELEASE</version>
7     </dependency>
8 </dependencies>

```

#### (2)代理数据源DataSourceProxy配置

在 seata-api 下创建 com.itheima.config.DataSourceProxyConfig 用于配置代理数据源, 并且将MyBatis的数据源切换成代理数据源:

```

1 @Configuration
2 public class DataSourceProxyConfig {
3
4     /**
5      * 手动配置DataSource
6      */
7     @Bean
8     @ConfigurationProperties(prefix = "spring.datasource")
9     public DruidDataSource dataSource(){
10         return new DruidDataSource();
11     }
12
13     /**

```

```

14      * 创建DataSourceProxy对象,交给SpringIOC容器
15      */
16      @Bean
17      public DataSourceProxy dataSourceProxy(DataSource dataSource){
18          return new DataSourceProxy(dataSource);
19      }
20
21
22      /****
23      * 所有涉及到数据库操作的对象的数据源一律换成DataSourceProxy
24      * Mybatis:SqlSessionFactory->DataSource换成DataSourceProxy
25      */
26      @Bean
27      public SqlSessionFactory sqlSessionFactory(DataSourceProxy
dataSourceProxy) throws Exception{
28          //SqlSessionFactoryBean
29          SqlSessionFactoryBean sqlSessionFactoryBean = new
SqlSessionFactoryBean();
30          //注入数据源
31          sqlSessionFactoryBean.setDataSource(dataSourceProxy);
32          return sqlSessionFactoryBean.getObject();
33      }
34  }

```

### (3)TC配置文件配置

在 seata-api 中添加file.conf和registry.conf文件，该文件可以直接从课件中拷贝。



### (4)Seata分布式事务分组配置

在每个需要参与Seata分布式事务的工程的应用.yml中添加seata的分组名字

```

1  cloud:
2    alibaba:
3      seata:
4        tx-service-group: my_test_tx_group

```

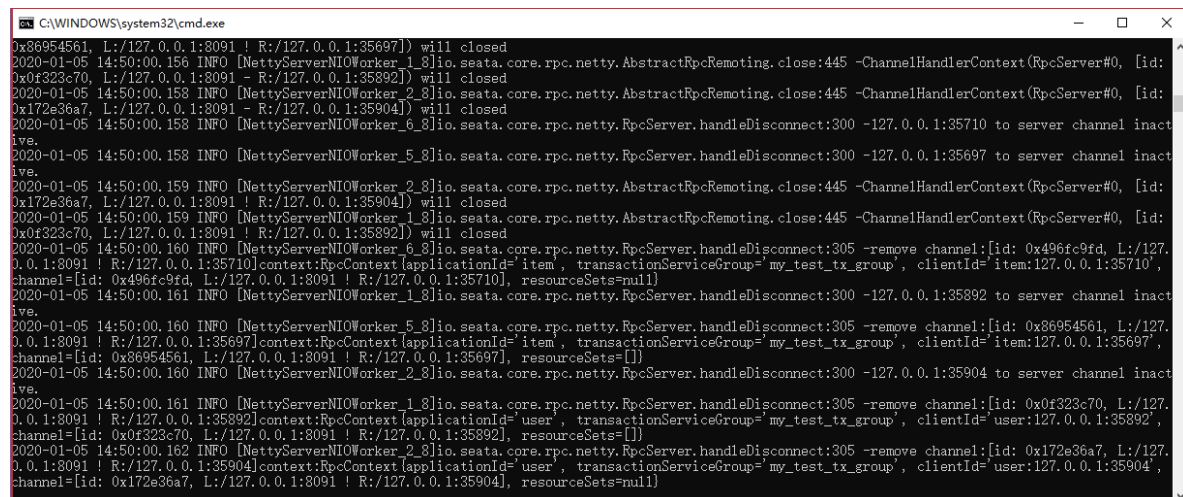
## 3.3.2 日志表添加

每个微服务对应的数据库中需要创建一张undo\_log表。

```
1 CREATE TABLE `undo_log` (  
2   `id` bigint(20) NOT NULL AUTO_INCREMENT,  
3   `branch_id` bigint(20) NOT NULL,  
4   `xid` varchar(100) NOT NULL,  
5   `context` varchar(128) NOT NULL,  
6   `rollback_info` longblob NOT NULL,  
7   `log_status` int(11) NOT NULL,  
8   `log_created` datetime NOT NULL,  
9   `log_modified` datetime NOT NULL,  
10  `ext` varchar(100) DEFAULT NULL,  
11  PRIMARY KEY (`id`),  
12  UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)  
13 ) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8;
```

### 3.3.2 测试分布式事务

启动TC



在业务微服务 seata-business 上添加 @GlobalTransactional 注解，并重新启动其他微服务，测试测试数据正确。

```
@GlobalTransactional //开启全局分布式事务  
@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, money: 10);  
}
```