第9章 微服务架构基础

广东财经大学信息学院
罗 东 俊 博士
ZSUJONE@126.COM

（内部资料，请勿外传）

目的和要求

- 了解微服务的基本概念和常见的解决方案。
- 了解**Spring Cloud**和**Spring Cloud Alibaba**的基本组件及其功能。
- 掌握微服务注册、配置和调用原理和实践方法。
- 掌握微服务容错原理和实践方法。
- 掌握微服务网关原理和实践方法。
- 掌握微服务分布式事务原理和实践方法。
- 掌握微服务链路追踪原理和实践方法。

主要内容

- 9.1 微服务概述
- 9.2 微服务注册、配置和调用
- 9.3 微服务容错
- 9.4 微服务网关
- 9.5 微服务分布式事务
- 9.6 微服务链路追踪

9.5 微服务分布式事务

- 9.5.1 概述
- 9.5.2 使用Seata中间件

9.5.1 概述

- 1.分布式事务基础
- 2.Seata简介

1.分布式事务基础

- 1）基本概念
- 2）分布式事务场景
- 3）分布式事务解决方案

1）基本概念

- ①事务
- ②本地事物
- ③分布式事务

①事务

- 事务指的就是一个操作单元，在这个操作单元中的所有操作最终要保持一致的行为，要么所有操作都成功，要么所有的操作都被撤销。
  - ✧简单地说，事务提供一种“要么什么都不做，要么做全套”机制。

②本地事物

- 本地事物其实可以认为是数据库提供的事务机制。数据库事务具有四大特性:
  - A：原子性(Atomicity)，一个事务中的所有操作，要么全部完成，要么全部不完成
  - C：一致性(Consistency)，在一个事务执行之前和执行之后数据库都必须处于一致性状态
  - I：隔离性(Isolation)，在并发环境中，当不同的事务同时操作相同的数据时，事务之间互不影响
  - D：持久性(Durability)，指的是只要事务成功结束，它对数据库所做的更新就必须永久的保存下来
- 数据库事务在实现时会将一次事务涉及的所有操作全部纳入到一个不可分割的执行单元，该执行单元中的所有操作要么都成功，要么都失败，只要其中任一操作执行失败，都将导致整个事务的回滚。

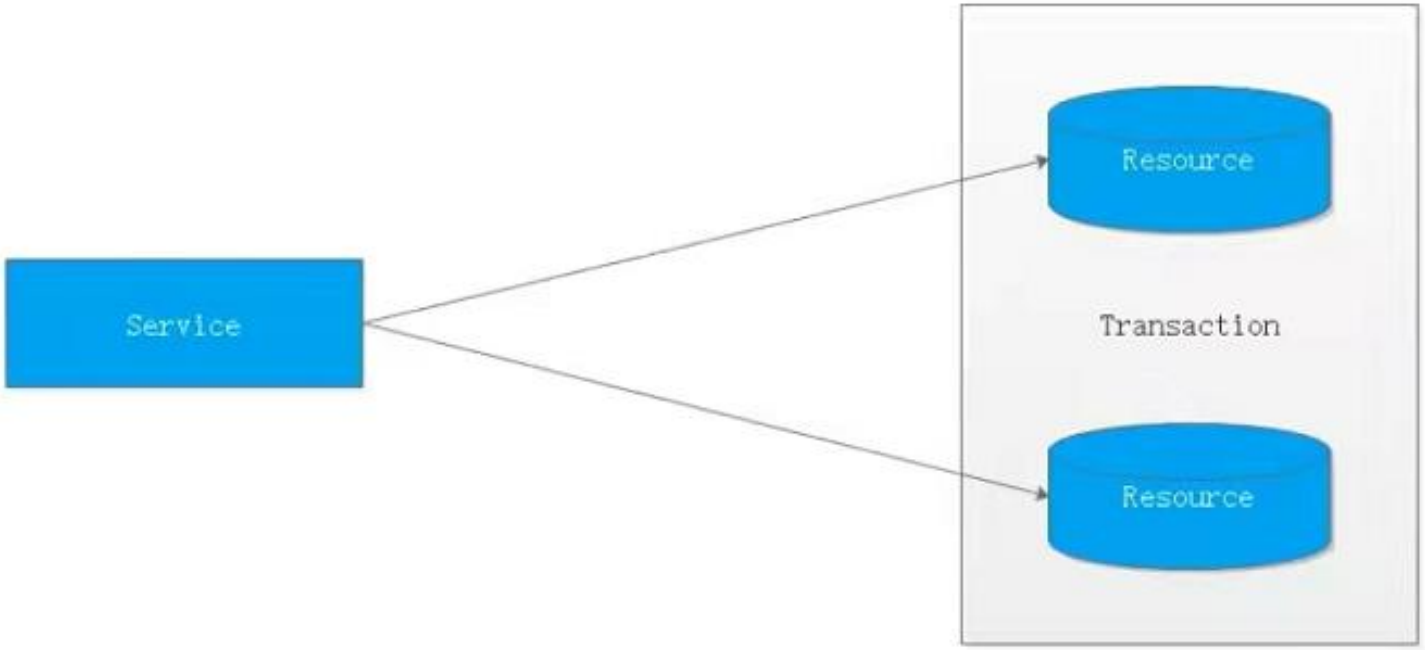
③分布式事务

- 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。
  - ✧简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，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。

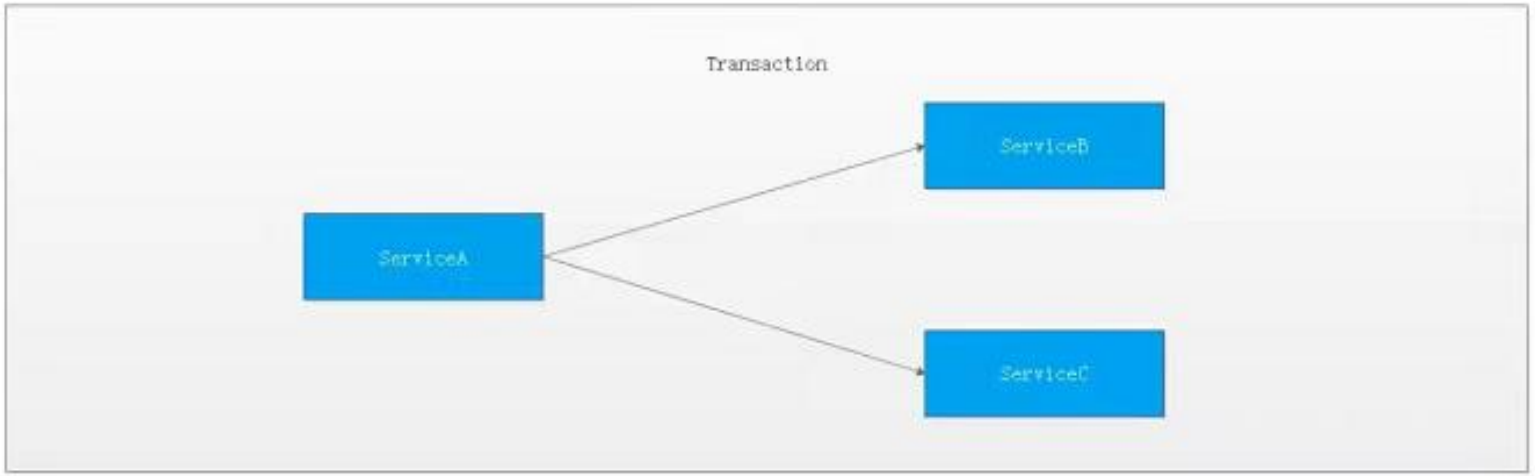
2）分布式事务场景

- ①单一服务访问多个数据库
- ②多服务访问一个数据库
- ③多服务访问多个数据库

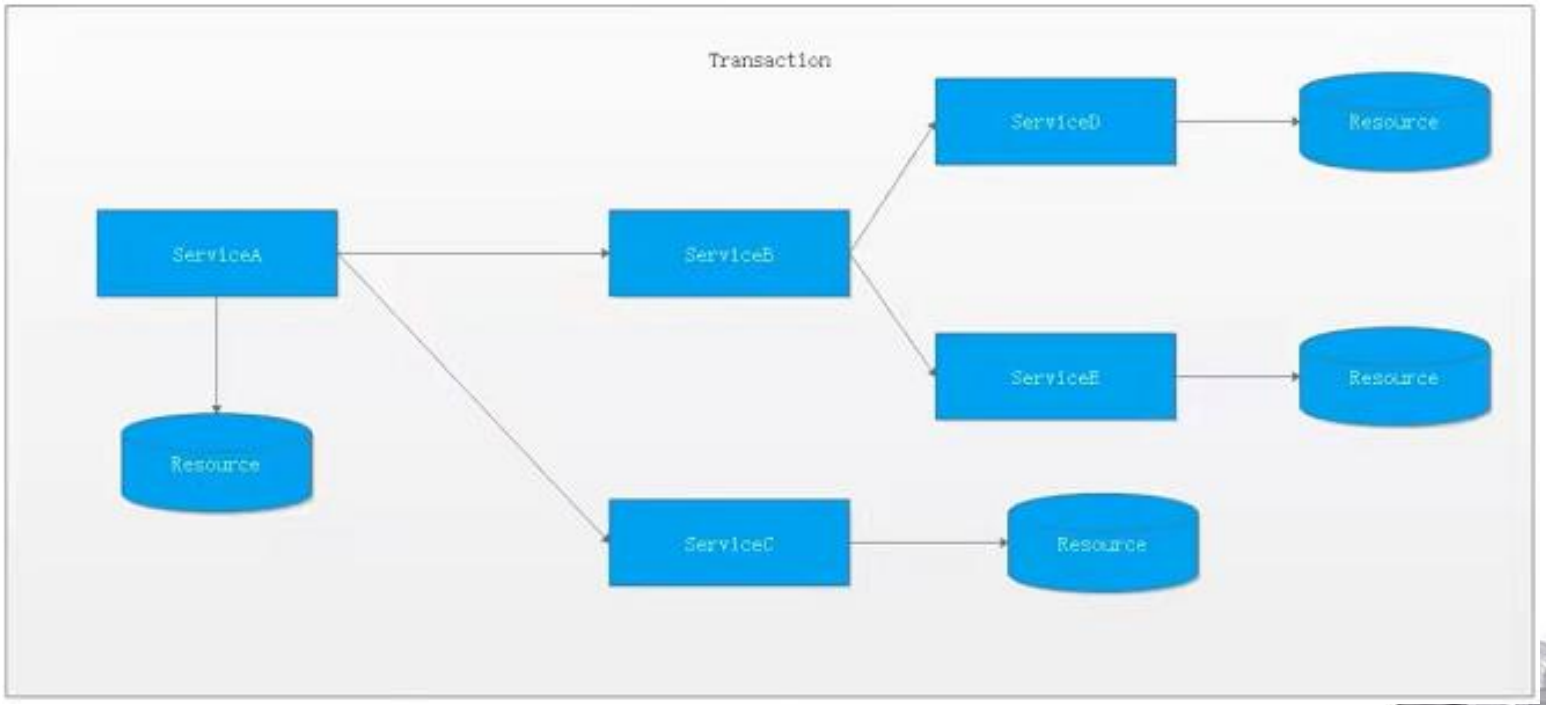
①单一服务访问多个数据库



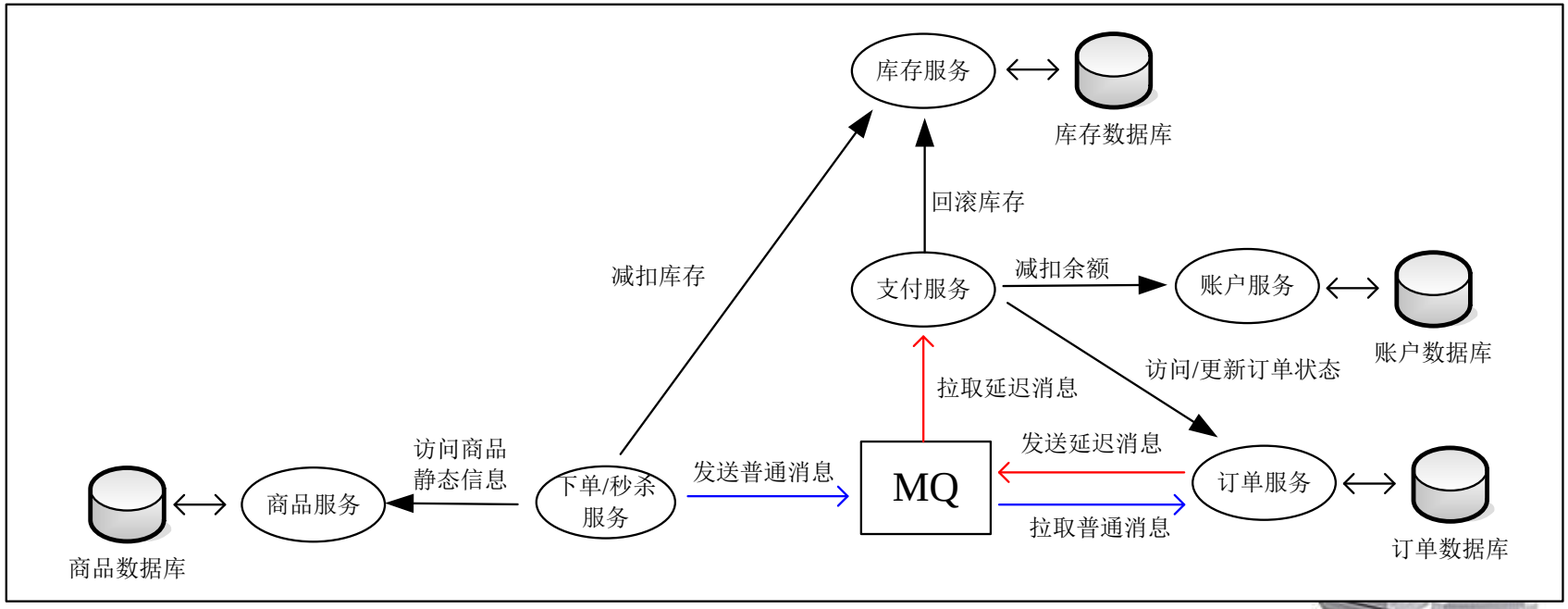
②多服务访问一个数据库



③多服务访问多个数据库



分布式事务应用架构示例



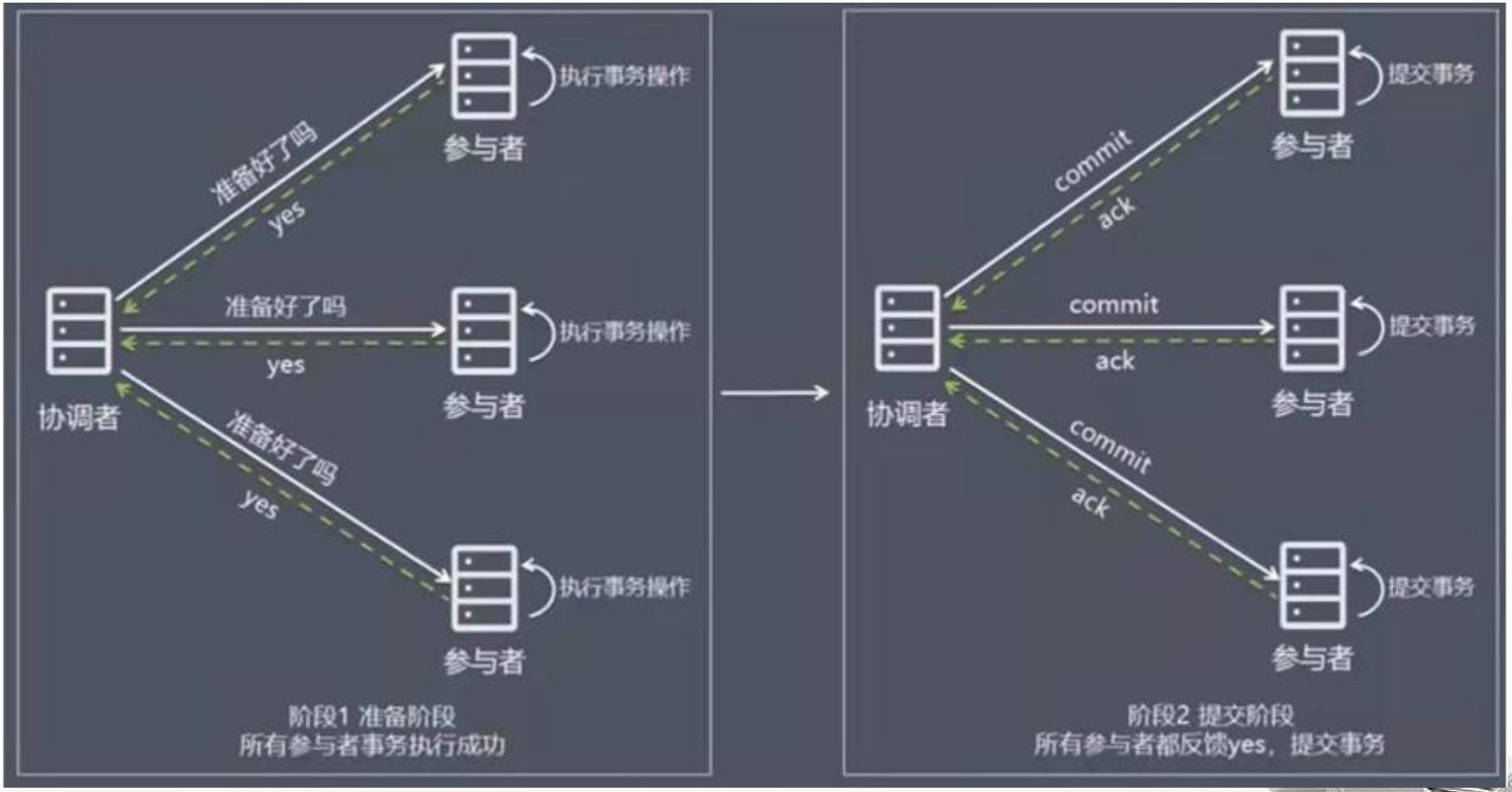
3）分布式事务解决方案

- 在分布式系统中，实现分布式事务的解决方案有：
  - ✧①两阶段提交（2PC）
  - ✧②补偿事务（TCC）
  - ✧③本地消息表
  - ✧④MQ事务消息

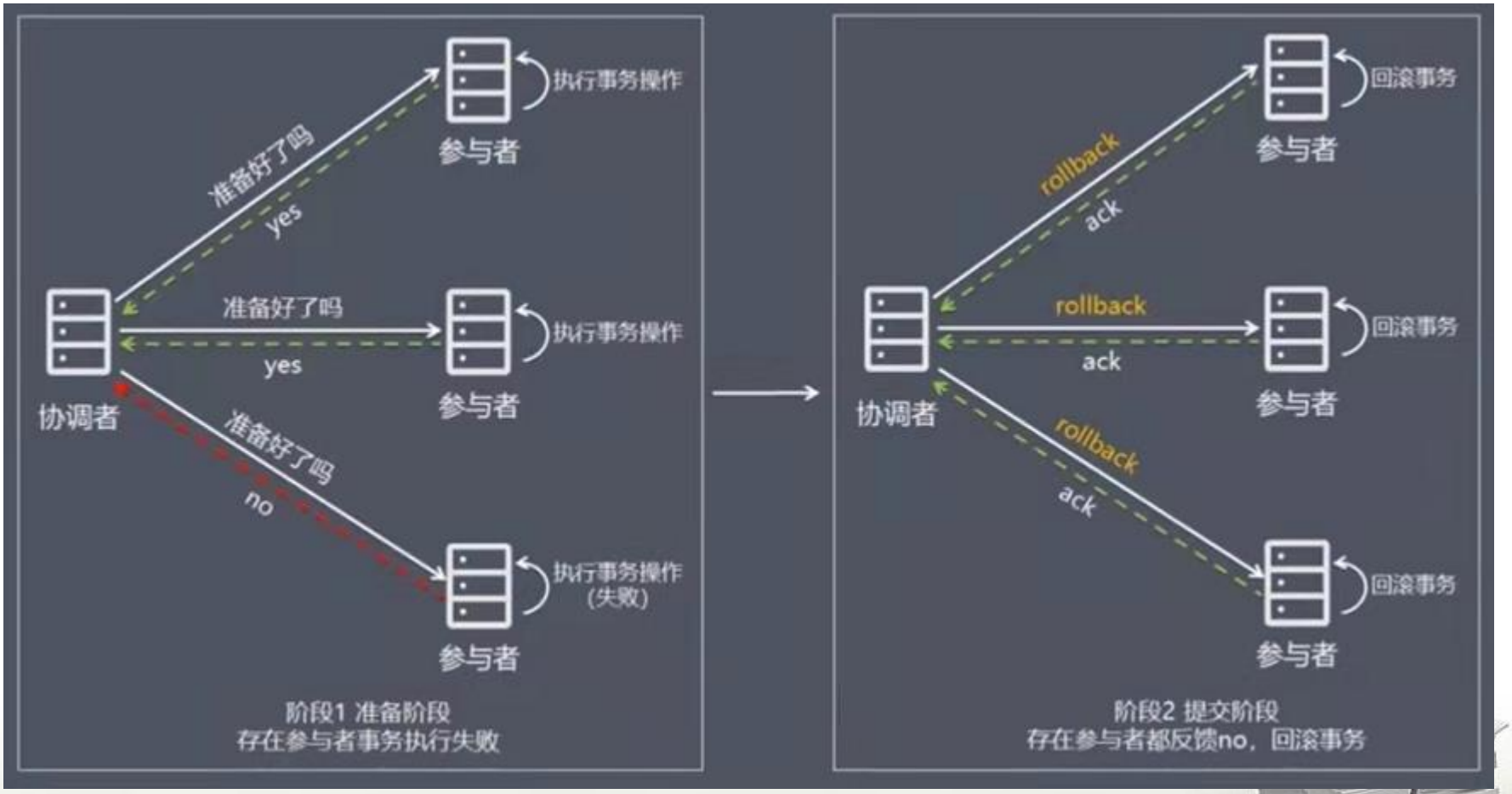
①两阶段提交（2PC）

- 2PC即两阶段XA协议，将整个事务流程分为：
  - ✧准备阶段（Prepare Phase）：参与者写本地的Undo和Redo日志，并不提交事务。
    - Undo日志记录修改前的数据，用于数据库回滚
    - Redo日志记录修改后的数据，用于提交事务后写入数据文件
  - ✧提交阶段（Commit Phase）：参与者根据事务协调者的指令执行提交或者回滚操作，并释放锁资源（仅在最后阶段释放锁资源）。

正常提交情况


协调者
准备好了吗
yes
执行事务操作
参与者
准备好了吗
yes
执行事务操作
参与者
准备好了吗
yes
执行事务操作
参与者
阶段1 准备阶段
所有参与者事务执行成功
协调者
commit
ack
提交事务
参与者
commit
ack
提交事务
参与者
commit
ack
提交事务
参与者
阶段2 提交阶段
所有参与者都反馈yes，提交事务

回滚情况


协调者
准备好了吗
yes
执行事务操作
参与者
准备好了吗
yes
执行事务操作
参与者
准备好了吗
no
执行事务操作
(失败)
参与者
阶段1 准备阶段
存在参与者事务执行失败
协调者
rollback
ack
回滚事务
参与者
rollback
ack
回滚事务
参与者
rollback
ack
回滚事务
参与者
阶段2 提交阶段
存在参与者都反馈no，回滚事务

2PC存在的问题

- 性能问题：
  - ✧所有参与者在事务提交阶段处于同步阻塞状态，会占用系统资源，容易导致性能瓶颈。
- 可靠性问题：
  - ✧如果协调者存在单点故障，参与者将一直处于锁定状态。
- 数据一致性问题：
  - ✧在提交阶段，如果局部网络问题，一部分参与者收到提交命令，而一部分参与者没收到，则会导致节点间数据不一致。

②补偿事务（TCC）

- TCC即Try-Confirm-Cancel，是服务化的2PC编程模式，其Try、Confirm、Cancel三个方法均有业务编程实现，相当于SQL事务中的Lock、Commit、Rollback。
  - ✧Try是一阶段操作，负责资源的检查和预留；
  - ✧Confirm是二阶段提交操作，负责事务的提交；
    - 只有当Try操作全部正常执行，才执行Confirm操作，从而完成一个完整的业务逻辑；
    - 如果Confirm操作执行失败，将会不断重试直到成功完成，并具有幂等性。
  - ✧Cancel是二阶段回滚操作，负责预留资源的取消。
    - 当Try操作存在执行失败，将执行Cancel操作；
    - 如果Cancel操作执行失败，将会不断重试直到成功完成，并具有幂等性。

TCC

实现

- 一个完整的业务活动由一个主业务服务与若干从业务服务组成
- 主业务服务负责发起并完成整个业务活动
- 从业务服务提供TCC型业务操作
- 业务活动管理器控制业务活动的一致性，它登记业务活动中的操作，并在业务活动提交时确认所有的TCC型操作的confirm操作，在业务活动取消时调用所有TCC型操作的cancel操作

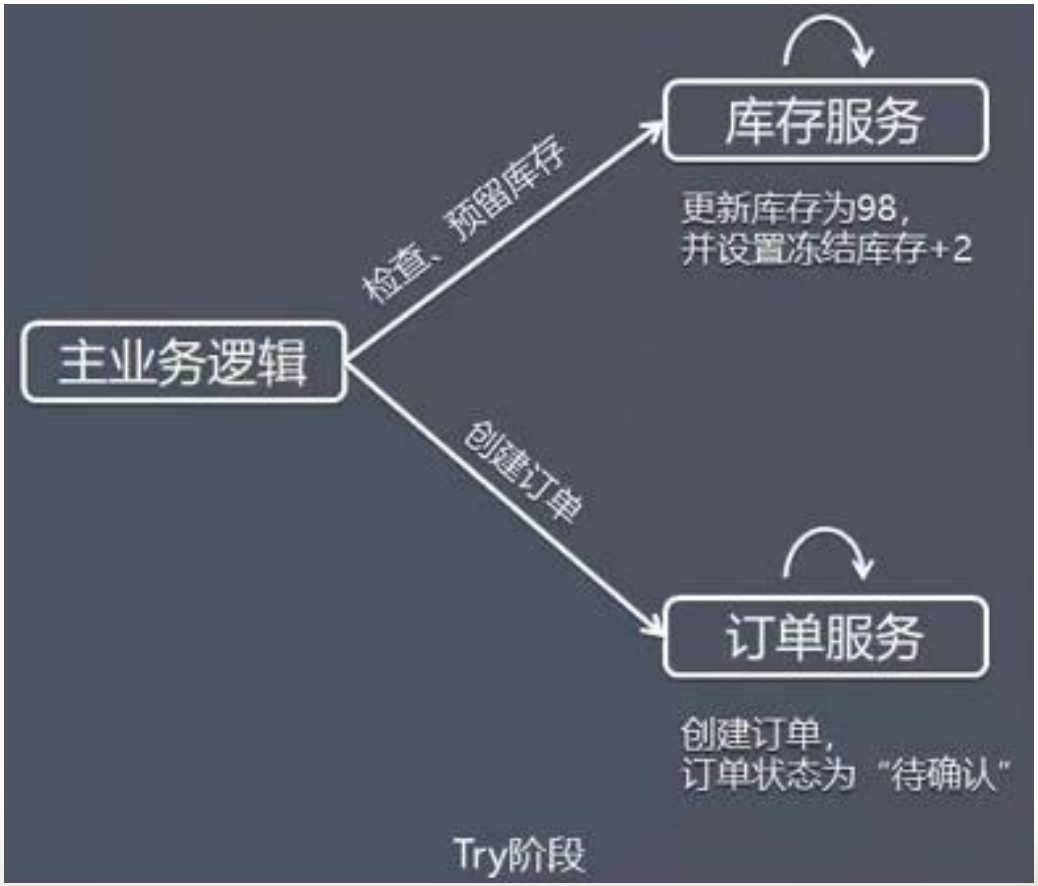
成本

- 实现TCC操作的成本
- 业务活动结束时confirm或cancel操作的执行成本
- 业务活动日志成本

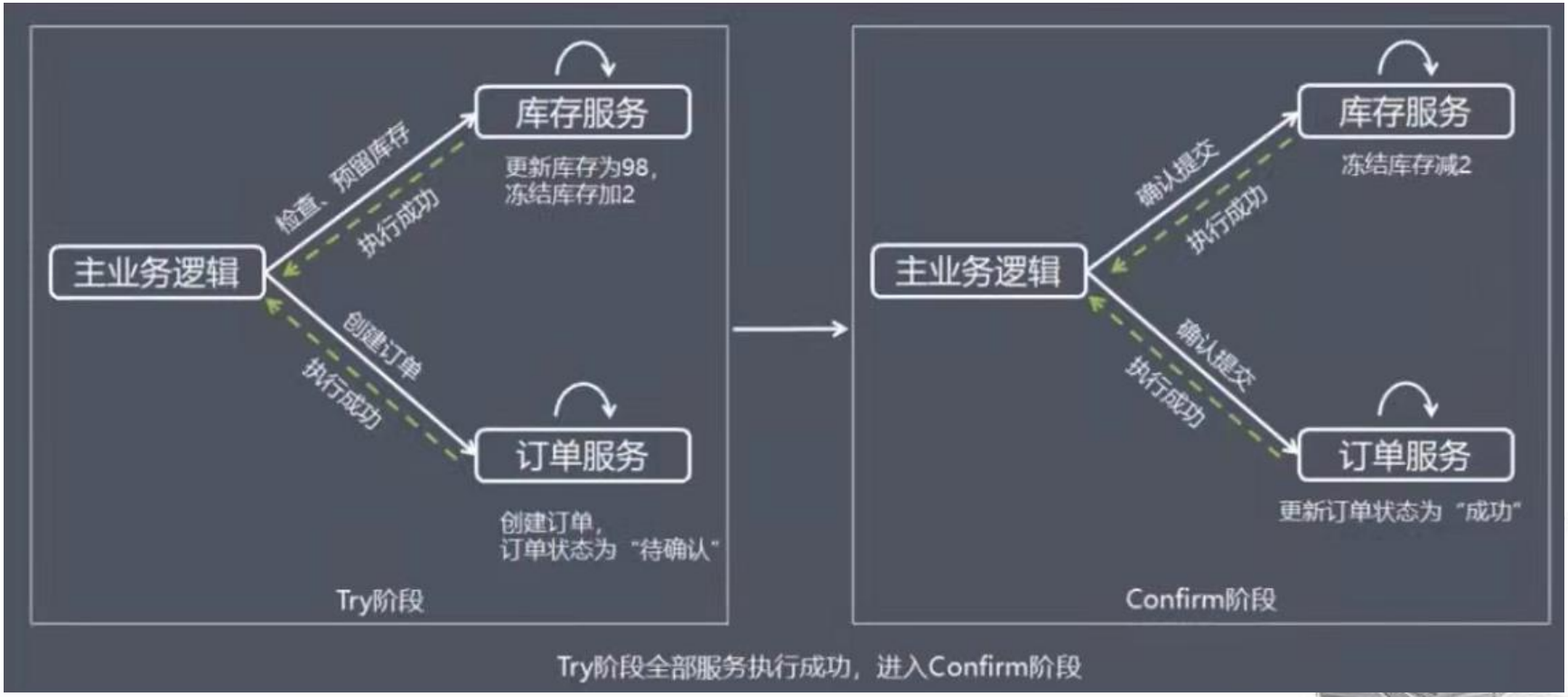
适用范围

- 强隔离性、严格一致性要求的业务活动
- 适用于执行时间较短的业务

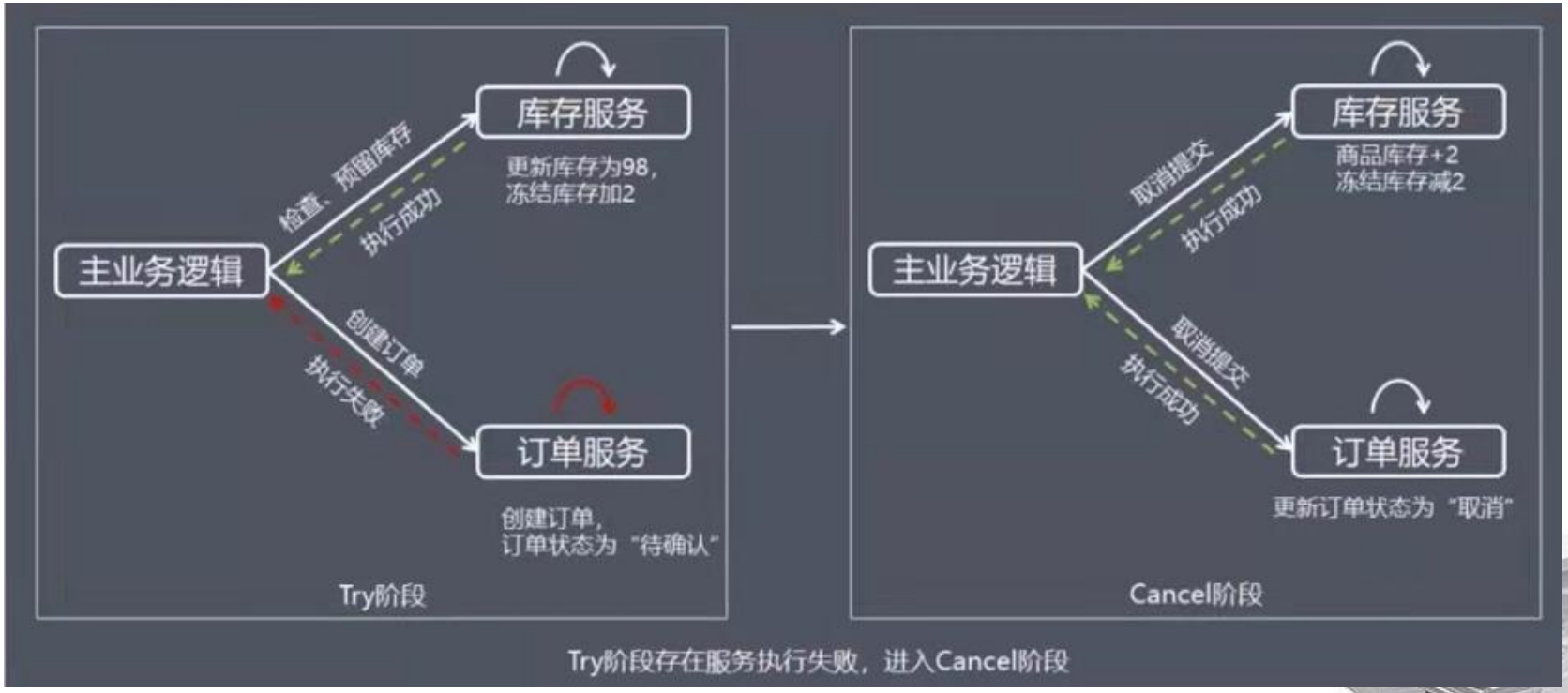
Try阶段


库存服务
更新库存为98，
并设置冻结库存+2
检查、预留库存
主业务逻辑
创建订单
订单服务
创建订单，
订单状态为“待确认”
Try阶段

Confirm阶段


主业务逻辑
检查、预留库存
执行成功
库存服务
更新库存为98，
冻结库存加2
创建订单
执行成功
订单服务
创建订单，
订单状态为“待确认”
Try阶段
主业务逻辑
确认提交
执行成功
库存服务
冻结库存减2
确认提交
执行成功
订单服务
更新订单状态为“成功”
Confirm阶段
Try阶段全部服务执行成功，进入Confirm阶段

Cancel阶段


主业务逻辑
检查、预留库存
执行成功
库存服务
更新库存为98，
冻结库存加2
创建订单
执行失败
订单服务
创建订单，
订单状态为“待确认”
Try阶段
主业务逻辑
取消提交
执行成功
库存服务
商品库存+2
冻结库存减2
取消提交
执行成功
订单服务
更新订单状态为“取消”
Cancel阶段
Try阶段存在服务执行失败，进入Cancel阶段

TCC与XA的异同

- TCC和XA都是两阶段提交。
- 区别是：
  - XA是资源层面的分布式事务，强一致性，在两阶段提交的整个过程中，一直会持有资源的锁。
  - TCC是业务层面的分布式事务，最终一致性，不会一直持有资源的锁。

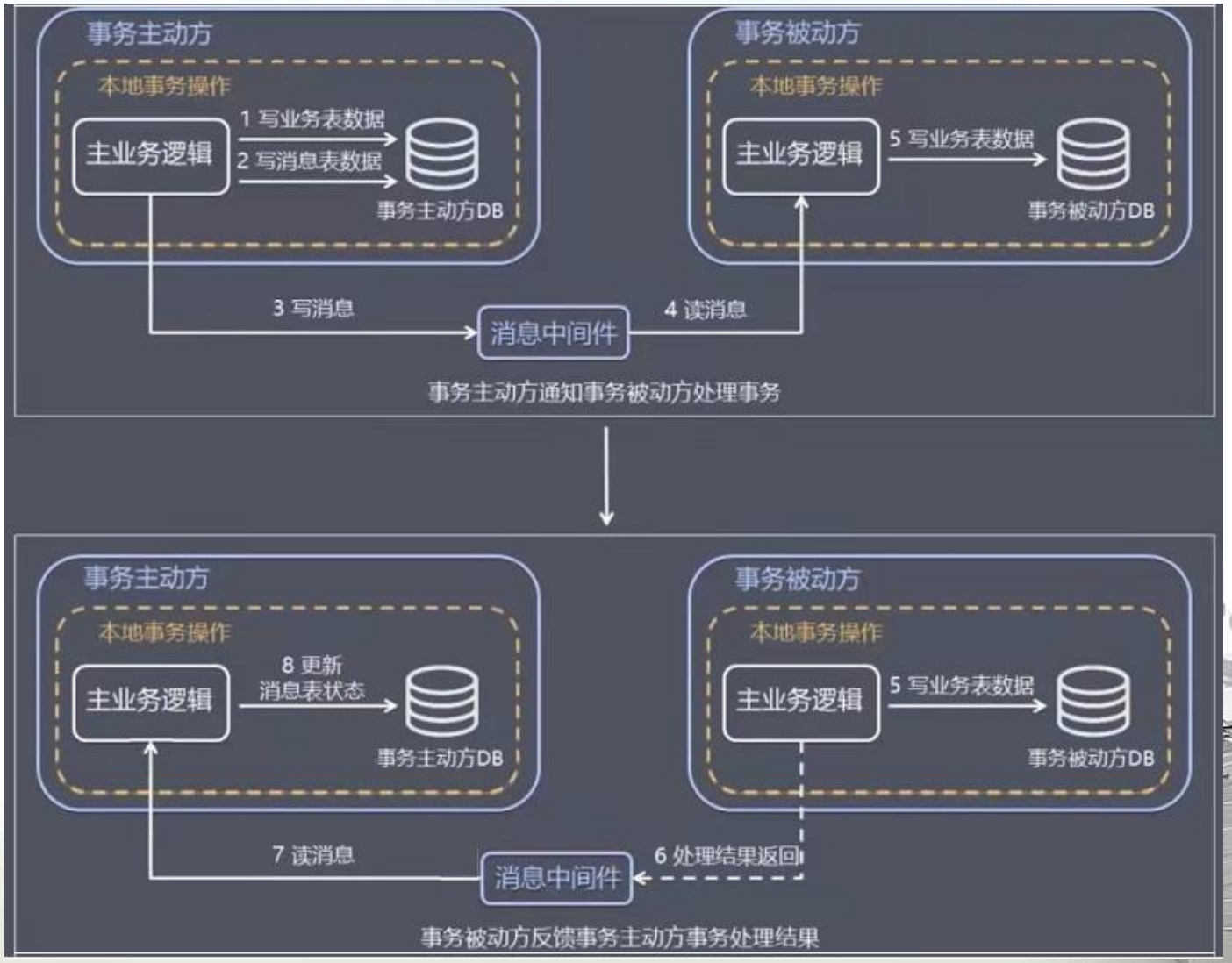
TCC的优缺点

- 优点：
  - 性能提升，具体业务来实现控制资源，锁的粒度变小，不会锁定整个资源。
  - 数据最终一致性：基于Confirm和Cancel的幂等性，保证事务最终完成确认或取消，保证数据最终一致性。
  - 可靠性：解决了XA协议的协调者单点故障问题，由主业务方发起并控制整个业务活动，业务活动管理器也变成多点，引入了集群。
- 缺点：
  - TCC的Try、Confirm和Cancel操作功能需业务提供，开发成本高。

③本地消息表

- 本地消息表核心思路是将分布式事务拆分成本地事务进行处理。
  - ✧在事务主动发起方额外新建一个事务消息表，轮询表中的事务数据，发送事务消息，并记录消息发送状态；
  - ✧事务被动方基于消息中间件消费事务消息，完成自己业务逻辑。

本地消息表方案


事务主动方
本地事务操作
主业务逻辑
1 写业务表数据
2 写消息表数据
事务主动方DB
事务被动方
本地事务操作
主业务逻辑
5 写业务表数据
事务被动方DB
3 写消息
消息中间件
4 读消息
事务主动方通知事务被动方处理事务
事务主动方
本地事务操作
主业务逻辑
8 更新
消息表状态
事务主动方DB
事务被动方
本地事务操作
主业务逻辑
5 写业务表数据
事务被动方DB
7 读消息
消息中间件
6 处理结果返回
事务被动方反馈事务主动方事务处理结果

本地消息表方案优缺点

- 优点：
  - ✧一种非常经典的实现，实现了最终一致性。
  - ✧可避免“业务处理成功+事务消息发送失败”，或“业务处理失败+事务消息发送成功”的棘手情况出现，弱化了对MQ中间件的依赖。
- 缺点：
  - ✧消息表会耦合到业务系统中，消息数据与业务数据在同一个数据库，占用业务系统资源。
  - ✧业务系统在使用关系型数据库情况下，消息服务性能会受到关系型数据库并发性能的局限。

④MQ事务消息

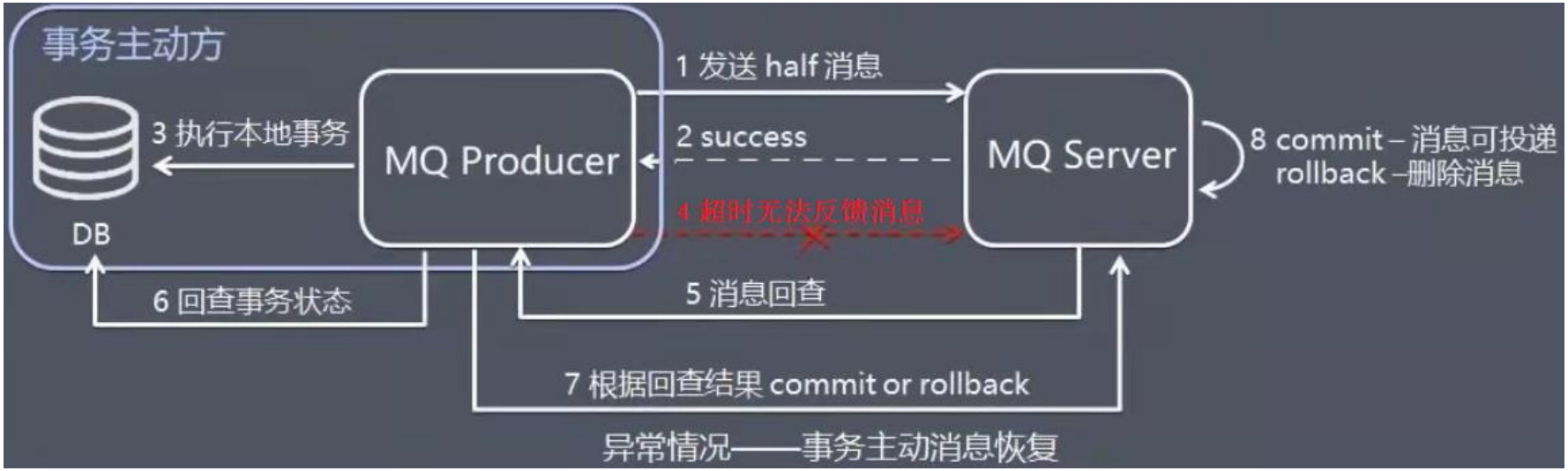
- 该方案基于RockerMQ事务消息保证上、下游应用数据操作的最终一致性，它将本地消息表封装在MQ Server中。
- 分两个流程：
  - ✧一是正常事务消息的发送及提交；
  - ✧二是事务消息的补偿流程。

正常事务消息的发送及提交

正常事务消息的发送及提交

- 正常情况下，事务消息发送方向MQ Server进行二次确认：
  - ✧步骤1：发送方向MQ Server发送半消息；
  - ✧步骤2：MQ Server将消息持久化成功后，向发送方发送ack确认消息已经发送成功；
  - ✧步骤3：发送方开始执行本地事务逻辑；
  - ✧步骤4：发送方根据本地事务执行结果向MQ Server提交二次确认（Commit或Rollback）；
  - ✧步骤5：MQ Server收到Commit状态则将半消息标记未可投递，订阅方最终将收到该消息；MQ Server收到Rollback状态则删除半消息，订阅方将不会收到该消息。

事务消息的补偿流程


事务主动方
1 发送 half 消息
3 执行本地事务
MQ Producer
2 success
MQ Server
8 commit – 消息可投递
rollback –删除消息
DB
4 超时无法反馈消息
6 回查事务状态
5 消息回查
7 根据回查结果 commit or rollback
异常情况——事务主动消息恢复

事务消息的补偿流程

- 当步骤4提交的二次确认超时未能到达MQ Server，此时MQ Server发起回查：
  - ✧步骤5’：MQ Server对该消息发起消息回查；
  - ✧步骤6’：发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果；
  - ✧步骤7’：发送方根据检查得到的本地事务的最终状态再次提交二次确认；
  - ✧步骤8’：MQ Server基于Commit或Rollback对消息进行投递或者删除。

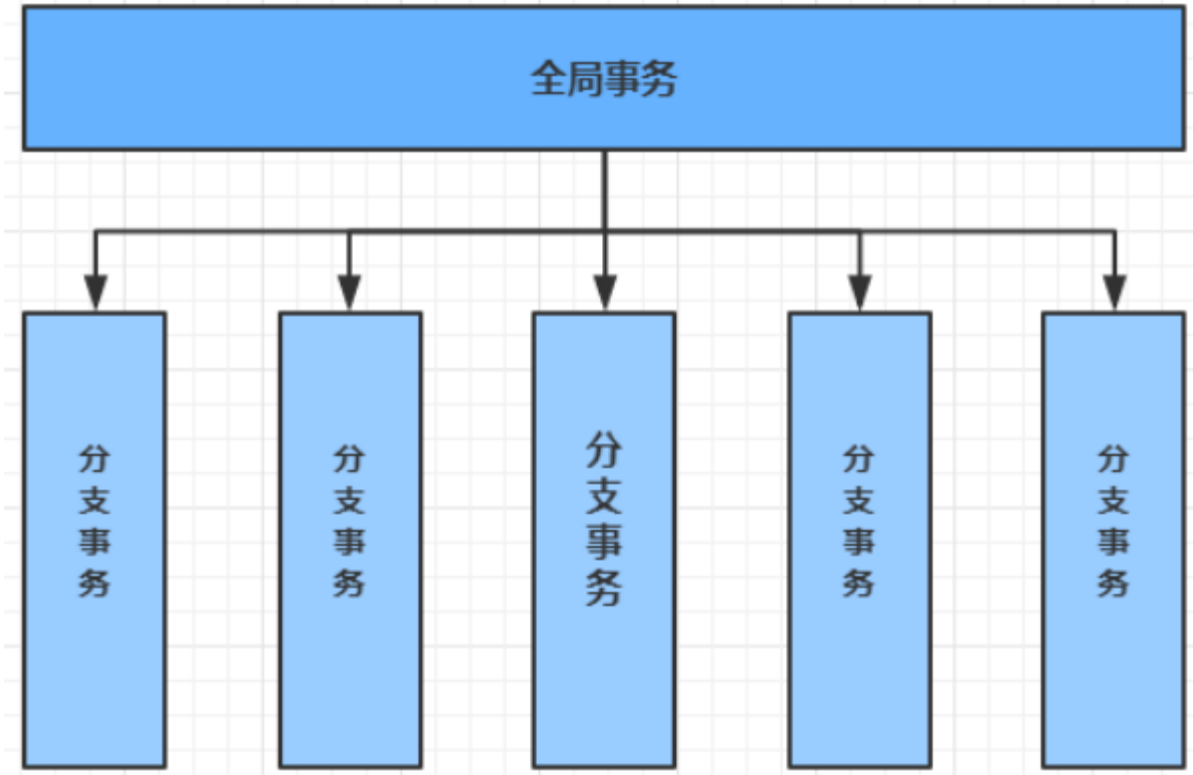
MQ事务消息方案优缺点

- 相比本地消息表方案具有以下优点：
  - ✧事务消息数据独立存储，降低业务系统与消息系统之间的耦合；
  - ✧吞吐量优于本地消息表方案；
  - ✧不需要依赖本地数据库事务实现了最终一致性。
- 缺点：
  - ✧一次消息发送需要两次网络请求（**half消息+commit/rollback消息**）；
  - ✧业务处理服务需要实现消息状态回查接口。

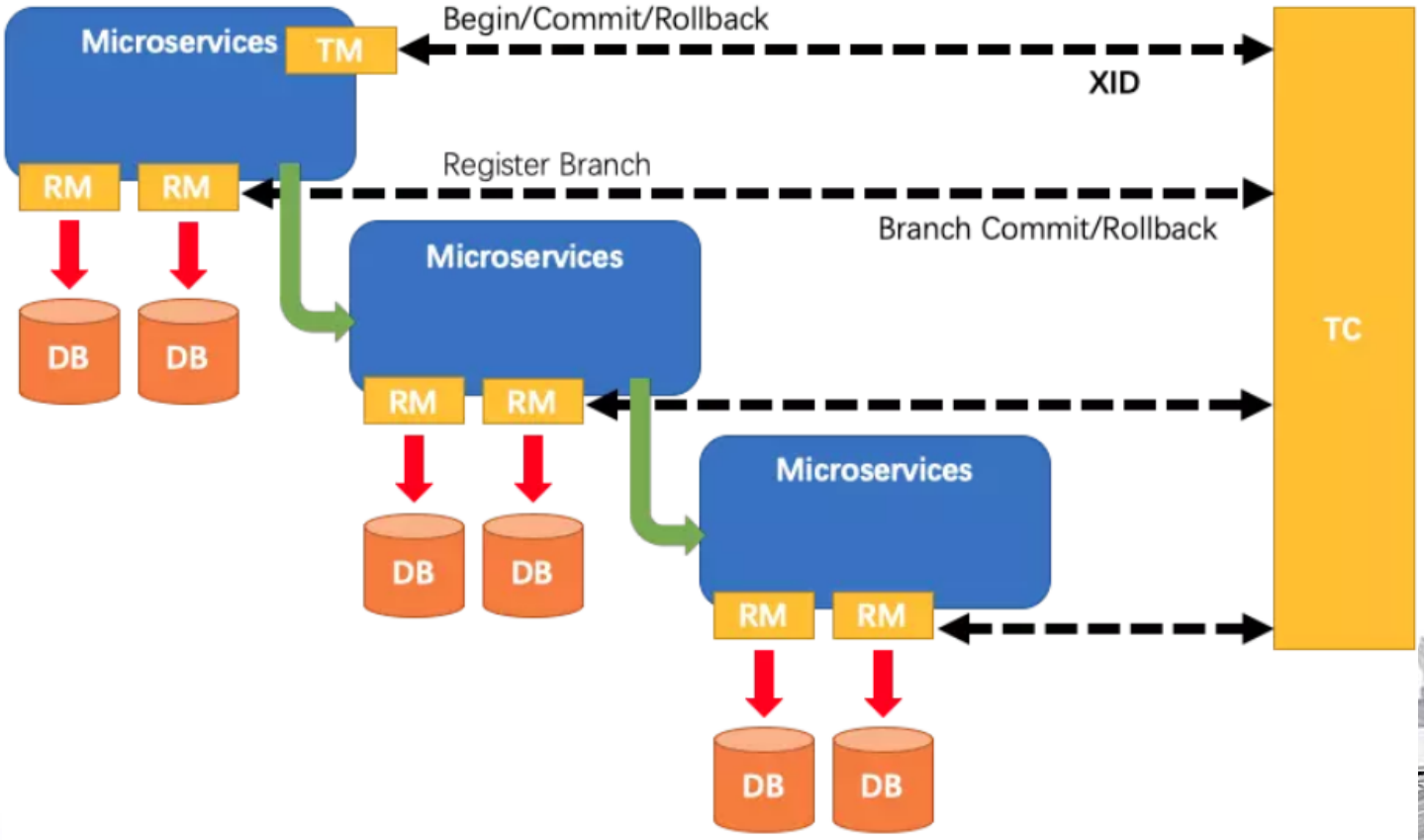
2.Seata简介

- **Seata（Simple Extensible Autonomous Transaction Architecture），是一套分布式事务解决方案，其设计目标是对业务无侵入。**
  - ✧它把一个分布式事务理解成一个包含了若干分支事务的全局事务。
  - ✧全局事务的职责是协调其下管辖的分支事务达成一致，要么一起成功提交，要么一起失败回滚。

Seata的分布式事务结构



Seata的组成


Microservices
TM
RM
RM
DB
DB
Begin/Commit/Rollback
XID
Register Branch
Branch Commit/Rollback
TC
Microservices
RM
RM
DB
DB
Microservices
RM
RM
DB
DB

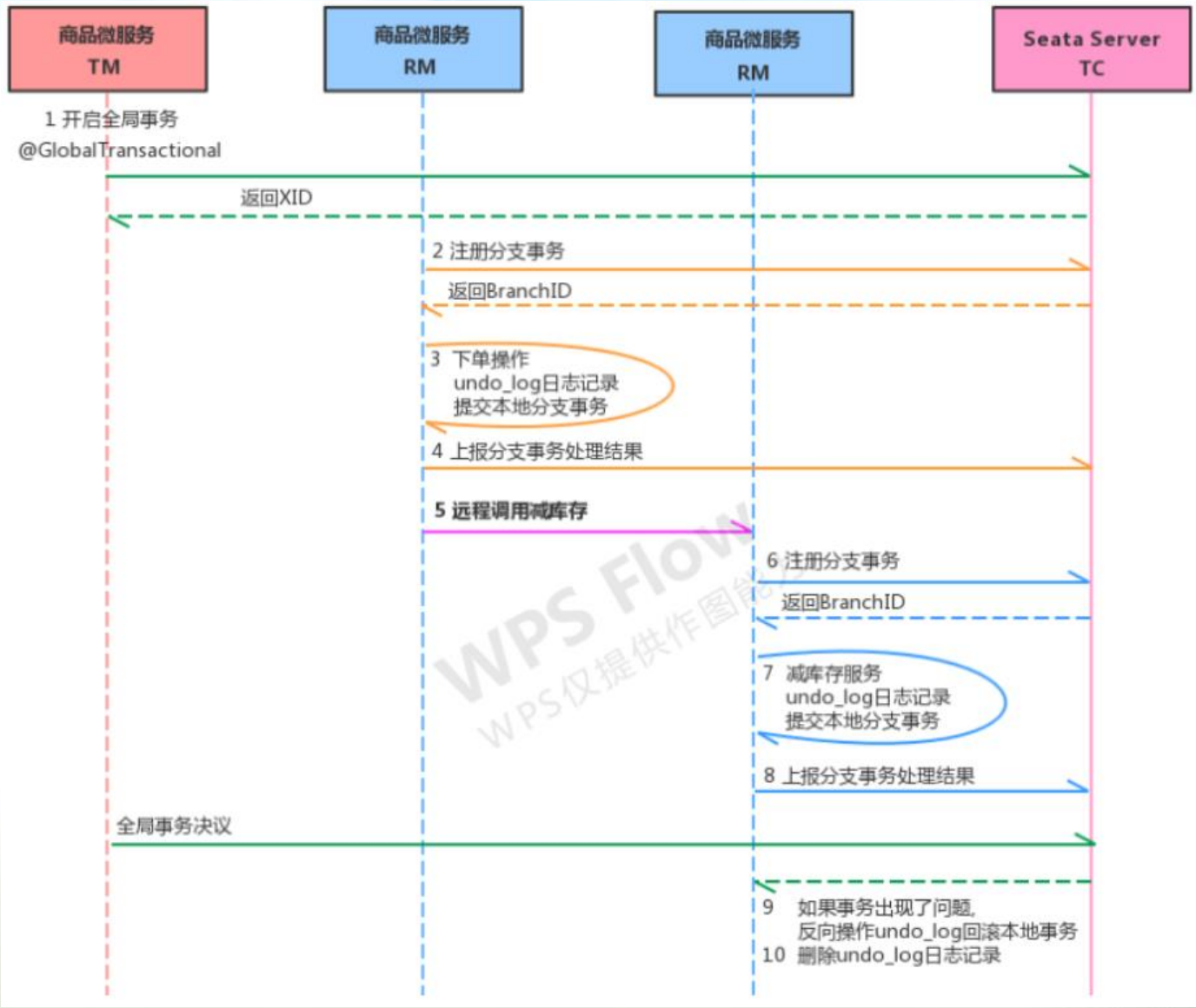
Seata的组成

- Seata主要由三个重要组件组成：
  - ✧TC：Transaction Coordinator（事务协调器），管理全局的分支事务的状态，用于全局性事务的提交和回滚。
  - ✧TM：Transaction Manager（事务管理器），用于开启、提交或者回滚全局事务。
  - ✧RM：Resource Manager（资源管理器），用于分支事务上的资源管理，向TC注册分支事务，上报分支事务的状态，接受TC的命令来提交或者回滚分支事务。

Seata的执行流程

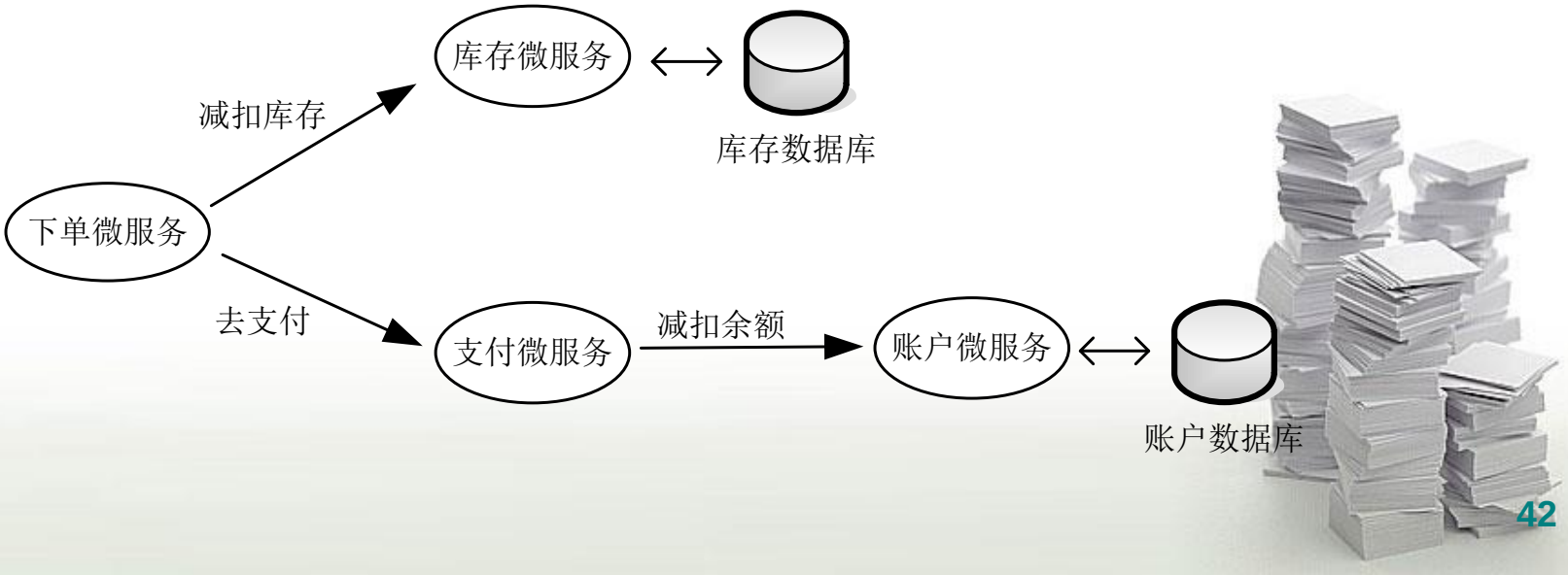
- Seata的执行流程如下：
  - ①A服务的TM向TC申请开启一个全局事务，TC就会创建一个全局事务并返回一个唯一的XID；
  - ②A服务的RM向TC注册分支事务，并将其纳入XID对应全局事务的管辖；
  - ③A服务执行分支事务，向数据库做操作；
  - ④A服务开始远程调用B服务，此时XID会在微服务的调用链上传播；
  - ⑤B服务的RM向TC注册分支事务，并将其纳入XID对应的全局事务的管辖；
  - ⑥B服务执行分支事务，向数据库做操作；
  - ⑦全局事务调用链处理完毕，TM根据有无异常向TC发起全局事务的提交或者回滚；
  - ⑧TC协调其管辖之下的所有分支事务， 决定是否回滚。

示例


商品微服务
TM
商品微服务
RM
商品微服务
RM
Seata Server
TC
1 开启全局事务
@GlobalTransactional
返回XID
2 注册分支事务
返回BranchID
3 下单操作
undo_log日志记录
提交本地分支事务
4 上报分支事务处理结果
5 远程调用减库存
6 注册分支事务
返回BranchID
7 减库存服务
undo_log日志记录
提交本地分支事务
8 上报分支事务处理结果
全局事务决议
9 如果事务出现了问题,
反向操作undo_log回滚本地事务
10 删除undo_log日志记录

9.5.2 使用Seata中间件

- 本小节以Seata为分布式事务解决方案，实现对本章节前面搭建的电商项目微服务架构分布式事务控制。
- 电商项目微服务架构及其数据库：



搭建步骤

- 1）搭建Seata服务
- 2）准备数据
- 3）建立ORM映射
- 4）添加Seata依赖和配置
- 5）效果测试

1）搭建Seata服务

- 搭建步骤：
  - ①安装Seata中间件
  - ②修改配置文件
  - ③启动Seata服务

①安装Seata中间件

- 下载地址:
  https://github.com/seata/seata/releases/download/v1.4.2/seata-server-1.4.2.zip
- 下载zip格式安装包后进行解压缩操作即可。

②修改配置文件

- 进入conf目录，修改以下两个配置文件中的配置：
  - ✧registry.conf：修改注册模式为Nacos，以将Seata服务注册到Nacos
  - ✧file.conf：修改存储模式，可保持默认设置
    - file模式（默认）：事务相关信息会存储在内存，并持久化到sessionStore目录下root.data文件
    - db模式：事务相关信息会存储到数据库中
    - redis模式：事务相关信息会存储到Redis服务器中

修改后的registry.conf

```
registry {
  # file 、nacos 、eureka、redis、zk、consul、etcd3、sofa
  # type = "file“
  type = "nacos“
  nacos {
    application = "seata-server“
    serverAddr = "127.0.0.1:8848“
    group = "SEATA_GROUP“
    namespace = "“
    cluster = "default“
    username = "“
    password = "“
  }
config {
  # file、nacos 、apollo、zk、consul、etcd3
  type = "file“
  file {
    name = "file.conf“
  }
}
```

修改后的file.conf

```
## transaction log store, only used in seata-server
store {
  ## store mode: file、db、redis
  mode = "file“
  ## rsa decryption public key
  publicKey = "“
  ## file store property
  file {
    ## store location dir
    dir = "sessionStore“
    # branch session size , if exceeded first try compress lockkey, still exceeded throws exceptions
    maxBranchSessionSize = 16384
    # globe session size , if exceeded throws exceptions
    maxGlobalSessionSize = 512
    # file buffer size , if exceeded allocate new buffer
    fileWriteBufferCacheSize = 16384
    # when recover batch read size
    sessionReloadReadSize = 100
    # async, sync
    flushDiskMode = async
  }
```

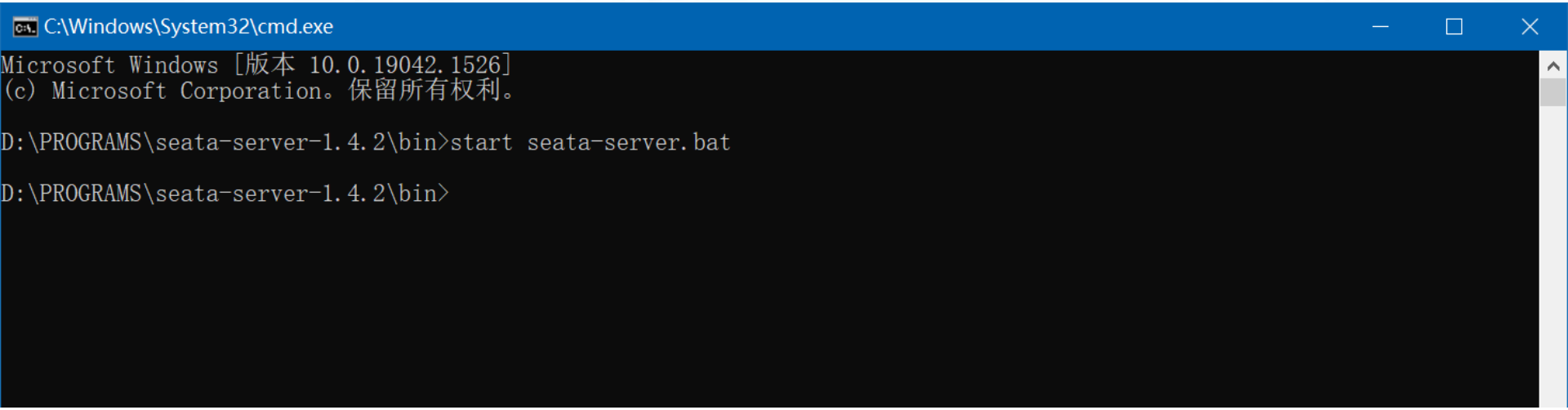
修改后的file.conf

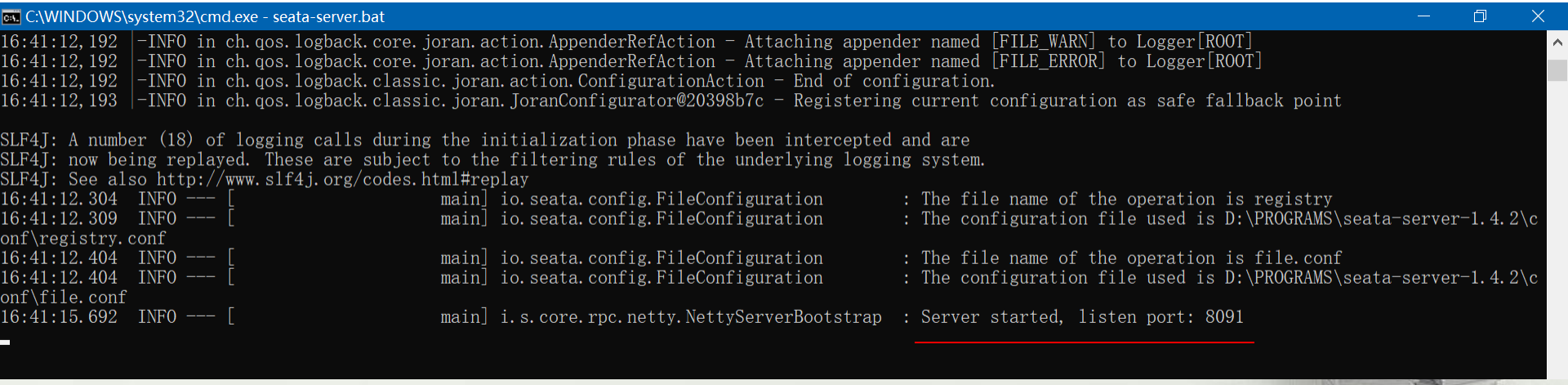
```
## database store property
 db {
   ## the implement of javax.sql.DataSource, such as DruidDataSource(druid)/BasicDataSource(dbcp)/HikariDataSource(hikari) etc.
   datasource = "druid“
   ## mysql/oracle/postgresql/h2/oceanbase etc.
   dbType = "mysql“
   driverClassName = "com.mysql.jdbc.Driver“
   ## if using mysql to store the data, recommend add rewriteBatchedStatements=true in jdbc connection param
   url = "jdbc:mysql://127.0.0.1:3306/seata?rewriteBatchedStatements=true“
   user = "mysql“
   password = "mysql“
   minConn = 5
   maxConn = 100
   globalTable = "global_table“
   branchTable = "branch_table“
   lockTable = "lock_table“
   queryLimit = 100
   maxWait = 5000
 }
```

修改后的file.conf

```
## redis store property
redis {
  ## redis mode: single、sentinel
  mode = "single“
  ## single mode property
  single {
    host = "127.0.0.1“
    port = "6379“
  }
  ## sentinel mode property
  sentinel {
    masterName = "“
    ## such as "10.28.235.65:26379,10.28.235.65:26380,10.28.235.65:26381“
    sentinelHosts = "“
  }
  password = "“
  database = "0“
  minConn = 1
  maxConn = 10
  maxTotal = 100
  queryLimit = 100  }}
```

③启动Seata服务


C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.19042.1526]
(c) Microsoft Corporation。保留所有权利。
D:\PROGRAMS\seata-server-1.4.2\bin>start seata-server.bat
D:\PROGRAMS\seata-server-1.4.2\bin>


C:\WINDOWS\system32\cmd.exe - seata-server.bat
16:41:12,192 |-INFO in ch.qos.logback.core.joran.action.AppenderRefAction - Attaching appender named [FILE_WARN] to Logger[ROOT]
16:41:12,192 |-INFO in ch.qos.logback.core.joran.action.AppenderRefAction - Attaching appender named [FILE_ERROR] to Logger[ROOT]
16:41:12,192 |-INFO in ch.qos.logback.classic.joran.action.ConfigurationAction - End of configuration.
16:41:12,193 |-INFO in ch.qos.logback.classic.joran.JoranConfigurator@20398b7c - Registering current configuration as safe fallback point
SLF4J: A number (18) of logging calls during the initialization phase have been intercepted and are
SLF4J: now being replayed. These are subject to the filtering rules of the underlying logging system.
SLF4J: See also http://www.slf4j.org/codes.html#replay
16:41:12.304 INFO --- [main] io.seata.config.FileConfiguration : The file name of the operation is registry
16:41:12.309 INFO --- [main] io.seata.config.FileConfiguration : The configuration file used is D:\PROGRAMS\seata-server-1.4.2\conf\registry.conf
16:41:12.404 INFO --- [main] io.seata.config.FileConfiguration : The file name of the operation is file.conf
16:41:12.404 INFO --- [main] io.seata.config.FileConfiguration : The configuration file used is D:\PROGRAMS\seata-server-1.4.2\conf\file.conf
16:41:15.692 INFO --- [main] i.s.core.rpc.netty.NettyServerBootstrap : Server started, listen port: 8091

seata服务成功注册到Nacos

public

服务列表 | public

服务名称 请输入服务名称 分组名称 请输入分组名称 隐藏空服务: 查询 创建服务

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
seata-server	SEATA_GROUP	1	1	1	false	详情 | 示例代码 | 订阅者 | 删除

集群: default 集群配置

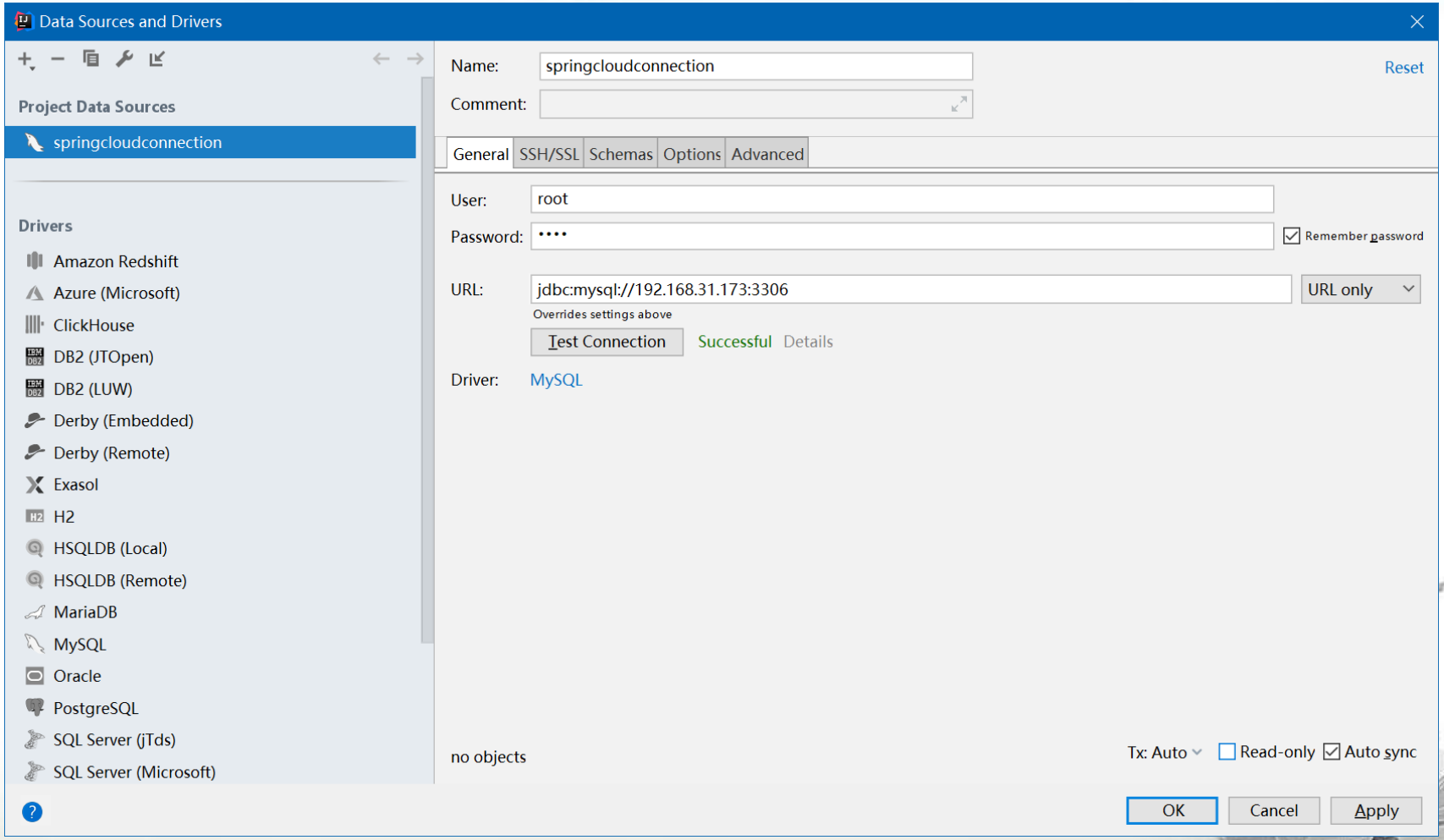
key value

IP	端口	临时实例	权重	健康状态	元数据	操作
192.168.31.103	8091	true	1	true		编辑 下线

2）准备数据

- 建立数据库连接，分别运行以下两个sql文件，为account-service模块和stock-service模块创建accountdb和stockdb数据库，并在accountdb数据库中创建t_ account和undo_log表，在stockdb数据库中创建t_ stock和undo_log表：
  - ✧①accountdb.sql
  - ✧②stockdb.sql

建立数据库连接


Data Sources and Drivers
Project Data Sources
springcloudconnection
Drivers
Amazon Redshift
Azure (Microsoft)
ClickHouse
DB2 (JTOpen)
DB2 (LUW)
Derby (Embedded)
Derby (Remote)
Exasol
H2
HSQLDB (Local)
HSQLDB (Remote)
MariaDB
MySQL
Oracle
PostgreSQL
SQL Server (jTds)
SQL Server (Microsoft)
Name: springcloudconnection
Reset
Comment:
General
SSH/SSL
Schemas
Options
Advanced
User: root
Password: ••••
Remember password
URL: jdbc:mysql://192.168.31.173:3306
URL only
Overrides settings above
Test Connection
Successful
Details
Driver: MySQL
no objects
Tx: Auto
Read-only
Auto sync
OK
Cancel
Apply

①accountdb.sql

```sql
#创建数据库
CREATE DATABASE accountdb;
#选择使用数据库
USE accountdb;
#创建表t_account并插入相关数据
DROP TABLE IF EXISTS t_account;
create table t_account
(
  id      int(20) auto_increment comment '记录id' primary key,
  a_id    varchar(200) null comment '账户号码',
  balance float default '0' null comment '账户余额',
  constraint a_id unique (a_id)
)charset = utf8;
insert into t_account values ('1', 'a001', '1000');
insert into t_account values ('2', 'a002', '2000.5');

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;
```

②stockdb.sql

```sql
#创建数据库
CREATE DATABASE stockdb;
#选择使用数据库
USE stockdb;
#创建表t_stock并插入相关数据
DROP TABLE IF EXISTS t_stock;
create table t_stock
(
  id    int(20) auto_increment comment '记录id' primary key,
  p_id  varchar(200)  null comment '商品代码',
  count int(20) default '0' null comment '库存量',
  constraint p_id unique (p_id)
)charset = utf8;
insert into t_stock values ('1', 'p001', '1000');
insert into t_stock values ('2', 'p002', '2000');

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;
```

3）建立ORM映射

- 分别在account-service模块和stock-service模块创建实体类和Mapper接口，并在pom.xml中添加相应依赖，在application.properties中添加相应配置，建立关系和对象间的映射。

①两模块添加的依赖

```
<!-- 阿里巴巴的Druid数据源依赖启动器 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>
<!--mybatis-plus的springboot支持-->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.1.1</version>
</dependency>
<!-- MySQL数据库连接驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>6.0.6</version>
    <scope>runtime</scope>
</dependency>
```

②account-service模块添加的类

- Account类
- AccountMapper接口

Account类

```
package com.scst.domain;
import lombok.Data;
import com.baomidou.mybatisplus.extension.activerecord.Model;
@Data
public class Account extends Model<Account>{
    private Integer id;
    private String aId;
    private float balance;
}
```

AccountMapper接口

```
package com.scst.mapper;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.scst.domain.Account;
import org.apache.ibatis.annotations.Mapper;
@Mapper
public interface AccountMapper extends BaseMapper<Account> {
}
```

增强后的AccountService类

```
package com.scst.service;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.scst.domain.Account;
import org.springframework.stereotype.Service;
@Service
public class AccountService {
    public boolean reduce(String accountId){
        Account account = new Account();
        QueryWrapper<Account> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("a_id",accountId);
        Account acc = account.selectOne(queryWrapper);
        if(acc!=null){
            float balance =acc.getBalance()-100;
            acc.setBalance(balance);
            acc.updateById();
            return true;
        }
        return false;
    }
}
```

③stock-service模块添加的类

- Stock类
- StockMapper接口

Stock类

```
package com.scst.domain;
import lombok.Data;
import com.baomidou.mybatisplus.extension.activerecord.Model;
@Data
public class Stock extends Model<Stock>{
    private Integer id;
    private String pId;
    private Integer count;
}
```

StockMapper接口

```
package com.scst.mapper;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.scst.domain.Stock;
import org.apache.ibatis.annotations.Mapper;
@Mapper
public interface StockMapper extends BaseMapper<Stock> {
}
```

增强后的StockService类

```
package com.scst.service;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.scst.domain.Stock;
import org.springframework.stereotype.Service;
@Service
public class StockService {
    public boolean deduct(String productId){
        Stock stock = new Stock();
        QueryWrapper<Stock> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("p_id",productId);
        Stock stk = stock.selectOne(queryWrapper);
        if(stk!=null){
            Integer count =stk.getCount()-10;
            stk.setCount(count);
            stk.updateById();
            return true;
        }
        return false;
    }
}
```

④两模块添加的配置

```
# 数据源连接配置
spring.datasource.druid.driver-class-name:com.mysql.cj.jdbc.Driver
spring.datasource.druid.url:jdbc:mysql://192.168.31.173:3306/accountdb[stockdb]?serverTimezone=UTC&us
eSSL=false&useUnicode=true&characterEncoding=utf8
spring.datasource.druid.username:root
spring.datasource.druid.password:root
spring.datasource.druid.initialSize=20
spring.datasource.druid.minIdle=10
spring.datasource.druid.maxActive=100
#配置全局的表名前缀
mybatis-plus.global-config.db-config.table-prefix=t_
#配置全局的id生成策略
mybatis-plus.global-config.db-config.id-type=auto
mybatis-plus.configuration.map-underscore-to-camel-case=true
mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl
mybatis-plus.configuration.lazyLoadingEnabled=true
```

4）添加Seata依赖和配置

- 为了访问Seata服务实现分布式事务控制，需要：
  - ①在chapter09父工程的pom.xml中引入Seata依赖；
  - ②在各模块的application.properties中添加Seata配置；
  - ③在需要分布式事务控制的方法上添加@GlobalTransactional。

①引入Seata依赖

```
<!--seata-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
</dependency>
```

②添加Seata配置

```
spring.cloud.alibaba.seata.tx-service-group=my_test_tx_group
```

③添加@GlobalTransactional

- 在OrderingController的purchase()方法上添加@GlobalTransactional。

```
    @GetMapping("/purchase/{pid}/{aid}")
    @GlobalTransactional//全局事务控制
    public String purchase(@PathVariable("pid") String productId, @PathVariable("aid") String
accountId) {
        String deductResult = stockFeignClient.deduct(productId);
        //模拟分布式事务异常
        int i = 10/0;
        String payResult = paymentFeignClient.pay(accountId);
        boolean flag = orderingService.purchase(productId,accountId);
        return deductResult+payResult;
    }
```

5）效果测试

- 启动Seata服务和各微服务，在浏览器中输入：**http://localhost:9010/ordering/purchase/p002/a001，ordering-service微服务会抛出“/ by zero”异常，stock-service微服务扣减库存会回滚。**

本章小结

- 本章具体讲解了：
  - ✧9.1 微服务概述
  - ✧9.2 微服务注册、配置和调用
  - ✧9.3 微服务容错
  - ✧9.4 微服务网关
  - ✧9.5 微服务分布式事务
  - ✧9.6 微服务链路追踪


聪明等于勤奋
劳力就是幸福