主讲老师：Fox

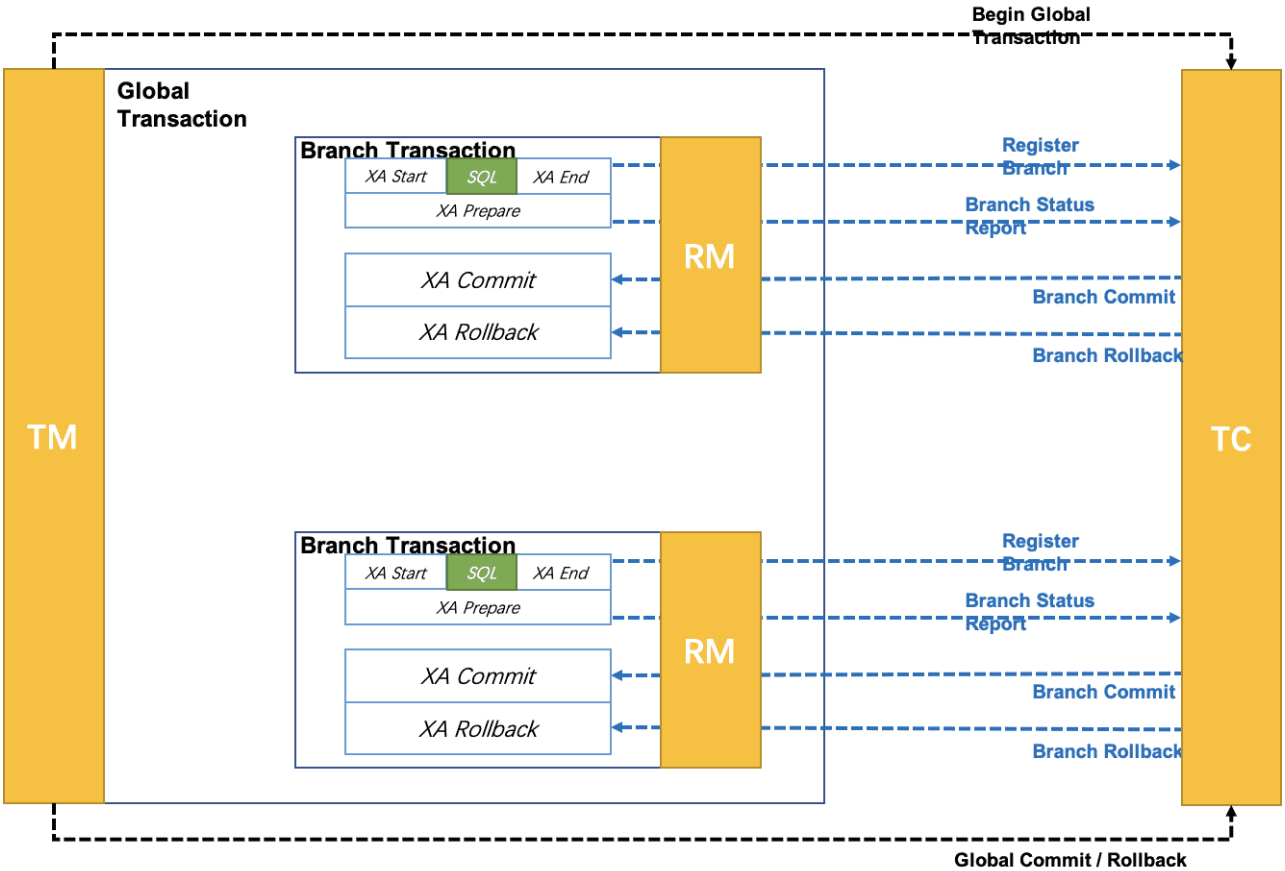
> 有道云笔记地址：https://note.youdao.com/s/Chg50Zbu

1. Seata XA模式实战

XA协议最主要的作用是就是定义了RM-TM的交互接口，XA规范除了定义的RM-TM交互的接口(XA Interface)之外，还对两阶段提交协议进行了优化。

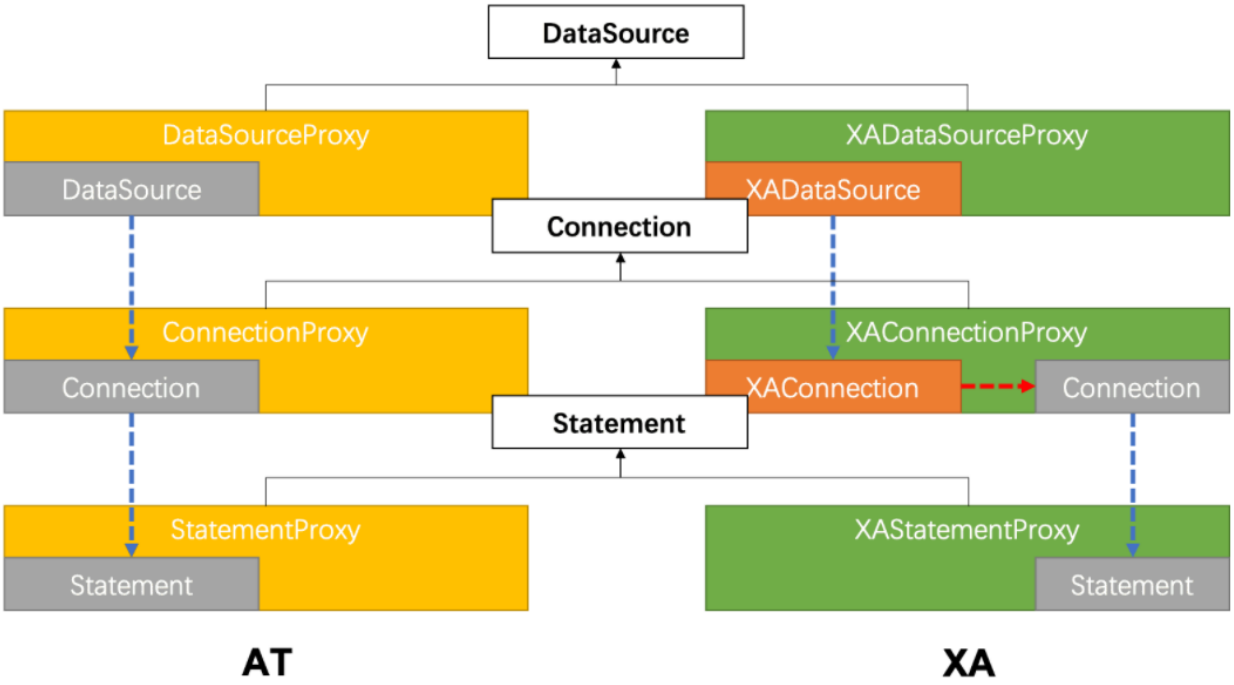
1.1 整体机制

在 Seata 定义的分布式事务框架内，利用事务资源（数据库、消息服务等）对 XA 协议的支持，以 XA 协议的机制来管理分支事务的一种 事务模式。



- 执行阶段：
  - 可回滚：业务 SQL 操作放在 XA 分支中进行，由资源对 XA 协议的支持来保证 **可回滚**
  - 持久化：XA 分支完成后，执行 XA prepare，同样，由资源对 XA 协议的支持来保证 **持久化**
- 完成阶段：
  - 分支提交：执行 XA 分支的 commit
  - 分支回滚：执行 XA 分支的 rollback

AT和XA模式数据源代理机制对比



XA 模式的使用

从编程模型上，XA 模式与 AT 模式保持完全一致。只需要修改数据源代理，即可实现 XA 模式与 AT 模式之间的切换。

```
@Bean("dataSource")
public DataSource dataSource(DruidDataSource druidDataSource) {
    // DataSourceProxy for AT mode
    // return new DataSourceProxy(druidDataSource);

    // DataSourceProxyXA for XA mode
    return new DataSourceProxyXA(druidDataSource);
}
```

1.2 Spring Cloud Alibaba整合Seata XA实战

对比Seata AT模式配置，只需修改两个地方：

- 微服务数据库不需要undo_log表，undo_log表仅用于AT模式
- 修改数据源代码模式为XA模式

```
seata:
  # 数据源代理模式 默认AT
```

```
3    data-source-proxy-mode: XA
```

2. Seata TCC模式实战

2.1 什么是TCC

TCC 基于分布式事务中的二阶段提交协议实现，它的全称为 Try-Confirm-Cancel，即资源预留（Try）、确认操作（Confirm）、取消操作（Cancel），他们的具体含义如下：

1. Try：对业务资源的检查并预留；
2. Confirm：对业务处理进行提交，即 commit 操作，只要 Try 成功，那么该步骤一定成功；
3. Cancel：对业务处理进行取消，即回滚操作，该步骤回对 Try 预留的资源进行释放。

- **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 是一种侵入式的分布式事务解决方案，以上三个操作都需要业务系统自行实现，对业务系统有着非常大的入侵性，设计相对复杂，但优点是 TCC 完全不依赖数据库，能够实现跨数据库、跨应用资源管理，对这些不同数据访问通过侵入式的编码方式实现一个原子操作，更好地解决了在各种复杂业务场景下的分布式事务问题。

常见开源TCC框架:

- Seata TCC
- Hmily
- Tcc-Transaction
- ByteTCC
- EasyTransaction

2.2 以用户下单为例

try-commit

try 阶段首先进行预留资源，然后在 commit 阶段扣除资源。如下图：

try-cancel

try 阶段首先进行预留资源，预留资源时扣减库存失败导致全局事务回滚，在 cancel 阶段释放资源。

如下图：

2.3 Seata TCC 模式

一个分布式的全局事务，整体是 两阶段提交 的模型。全局事务是由若干分支事务组成的，分支事务要满足 两阶段提交 的模型要求，即需要每个分支事务都具备自己的：

- 一阶段 prepare 行为
- 二阶段 commit 或 rollback 行为

在Seata中，AT模式与TCC模式事实上都是两阶段提交的具体实现，他们的区别在于：

AT 模式基于 支持本地 ACID 事务的关系型数据库：

- 一阶段 prepare 行为：在本地事务中，一并提交业务数据更新和相应回滚日志记录。
- 二阶段 commit 行为：马上成功结束，自动异步批量清理回滚日志。
- 二阶段 rollback 行为：通过回滚日志，自动生成补偿操作，完成数据回滚。

相应的，TCC 模式不依赖于底层数据资源的事务支持：

- 一阶段 prepare 行为：调用自定义的 prepare 逻辑。
- 二阶段 commit 行为：调用自定义的 commit 逻辑。
- 二阶段 rollback 行为：调用自定义的 rollback 逻辑。

简单点概括，SEATA的TCC模式就是手工的AT模式，它允许你自定义两阶段的处理逻辑而不依赖AT模式的undo_log。

2.4 Seata TCC模式接口如何改造

假设现有一个业务需要同时使用服务 A 和服务 B 完成一个事务操作，我们在服务 A 定义该服务的一个 TCC 接口：

```
1 public interface TccActionOne {
2     @TwoPhaseBusinessAction(name = "prepare", commitMethod = "commit", rollbackMethod = "rollback")
3     public boolean prepare(BusinessActionContext actionContext, @BusinessActionContextParameter(paramName = "a") String a);
4
```

```
    public boolean commit(BusinessActionContext actionContext);

    public boolean rollback(BusinessActionContext actionContext);
}
```

同样，在服务 B 定义该服务的一个 TCC 接口：

```
public interface TccActionTwo {
    @TwoPhaseBusinessAction(name = "prepare", commitMethod = "commit", rollbackMethod = "rollback")
    public void prepare(BusinessActionContext actionContext, @BusinessActionContextParameter(paramName = "b") String b);

    public void commit(BusinessActionContext actionContext);

    public void rollback(BusinessActionContext actionContext);
}
```

在业务所在系统中开启全局事务并执行服务 A 和服务 B 的 TCC 预留资源方法：

```
@GlobalTransactional
public String doTransactionCommit(){
    //服务A事务参与者
    tccActionOne.prepare(null,"one");
    //服务B事务参与者
    tccActionTwo.prepare(null,"two");
}
```

以上就是使用 Seata TCC 模式实现一个全局事务的例子，TCC 模式同样使用 @GlobalTransactional 注解开启全局事务，而服务 A 和服务 B 的 TCC 接口为事务参与者，Seata 会把一个 TCC 接口当成一个 Resource，也叫 TCC Resource。

2.5 TCC如何控制异常

在 TCC 模型执行的过程中，还可能会出现各种异常，其中最为常见的有空回滚、幂等、悬挂等。TCC 模式是分布式事务中非常重要的事务模式，但是幂等、悬挂和空回滚一直是 TCC 模式需要考虑的问题，Seata 框架在 1.5.1 版本完美解决了这些问题。

如何处理空回滚

空回滚指的是在一个分布式事务中，在没有调用参与方的 Try 方法的情况下，TM 驱动二阶段回滚调用了参与方的 Cancel 方法。

那么空回滚是如何产生的呢?

如上图所示，全局事务开启后，参与者 A 分支注册完成之后会执行参与者一阶段 RPC 方法，如果此时参与者 A 所在的机器发生宕机，网络异常，都会造成 RPC 调用失败，即参与者 A 一阶段方法未成功执行，但是此时全局事务已经开启，Seata 必须要推进到终态，在全局事务回滚时会调用参与者 A 的 Cancel 方法，从而造成空回滚。

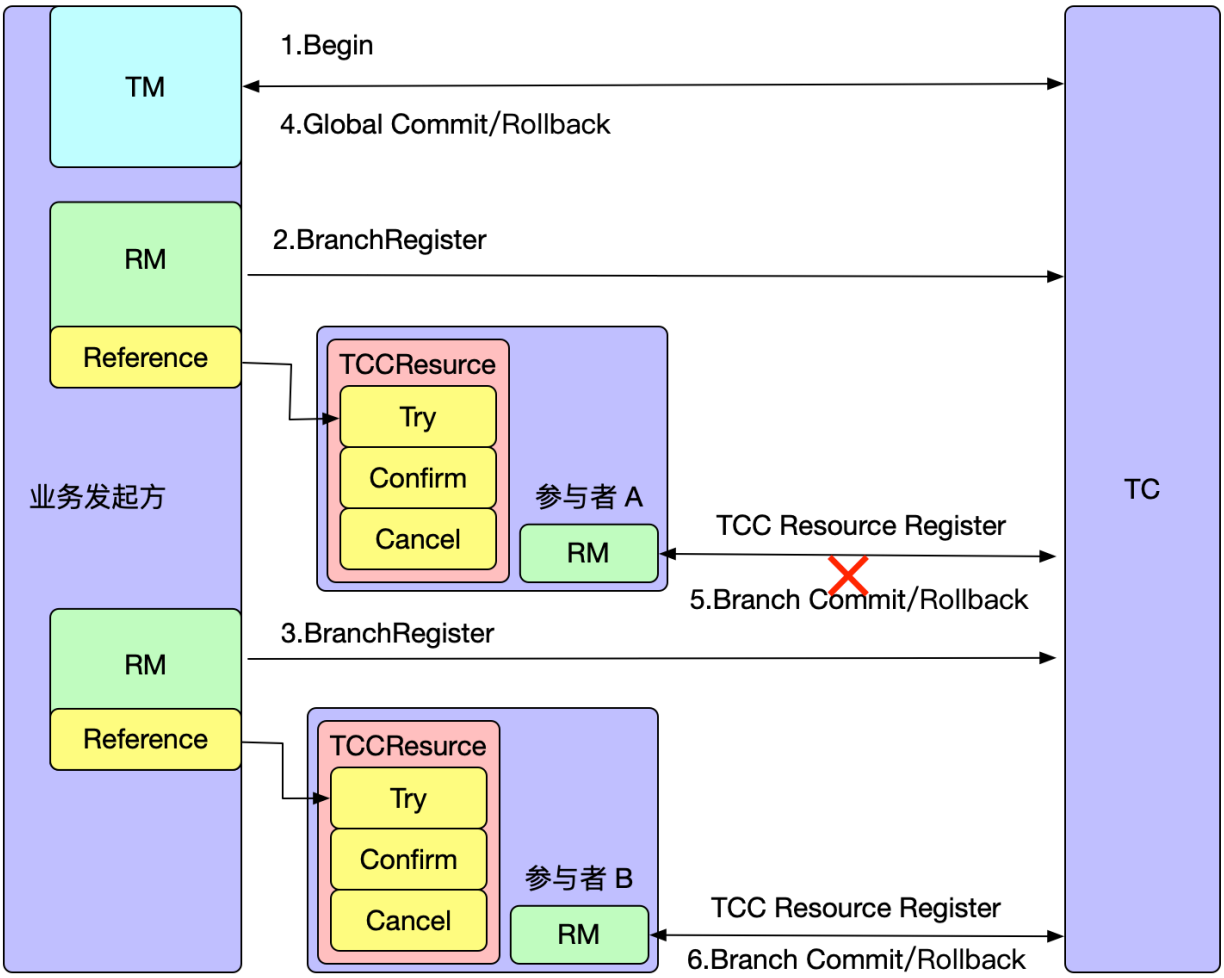
要想防止空回滚，那么必须在 Cancel 方法中识别这是一个空回滚，Seata 是如何做的呢?

Seata 的做法是新增一个 TCC 事务控制表，包含事务的 XID 和 BranchID 信息，在 Try 方法执行时插入一条记录，表示一阶段执行了，执行 Cancel 方法时读取这条记录，如果记录不存在，说明 Try 方法没有执行。

如何处理幂等

幂等问题指的是 TC 重复进行二阶段提交，因此 Confirm/Cancel 接口需要支持幂等处理，即不会产生资源重复提交或者重复释放。

那么幂等问题是如何产生的呢?



如上图所示，参与者 A 执行完二阶段之后，由于网络抖动或者宕机问题，会造成 TC 收不到参与者 A 执行二阶段的返回结果，TC 会重复发起调用，直到二阶段执行结果成功。

Seata 是如何处理幂等问题的呢?

同样的也是在 TCC 事务控制表中增加一个记录状态的字段 status，该字段有 3 个值，分别为：

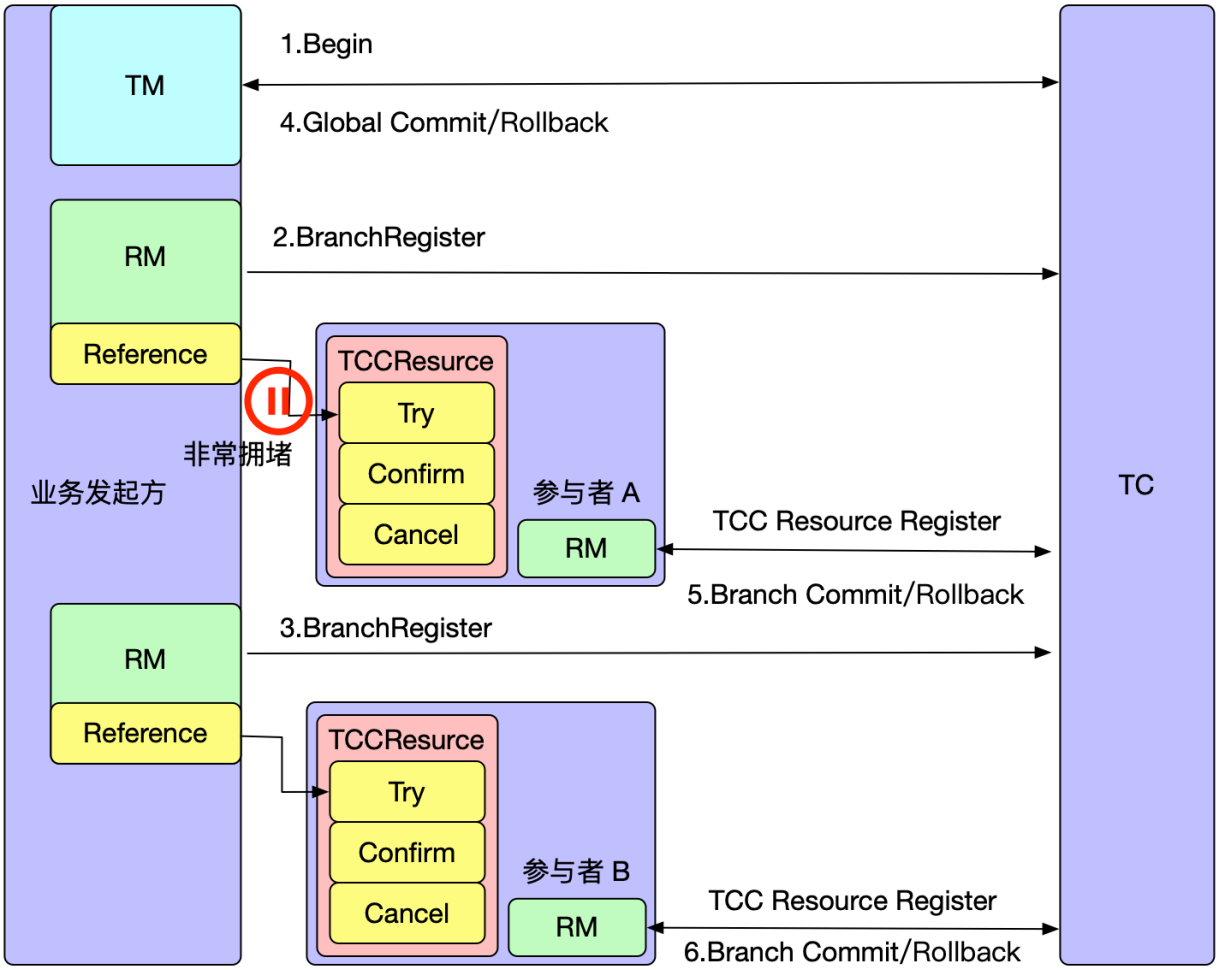
1. tried：1
2. committed：2
3. rollbacked：3

二阶段 Confirm/Cancel 方法执行后，将状态改为 committed 或 rollbacked 状态。当重复调用二阶段 Confirm/Cancel 方法时，判断事务状态即可解决幂等问题。

如何处理悬挂

悬挂指的是二阶段 Cancel 方法比 一阶段 Try 方法优先执行，由于允许空回滚的原因，在执行完二阶段 Cancel 方法之后直接空回滚返回成功，此时全局事务已结束，但是由于 Try 方法随后执行，这就会造成一阶段 Try 方法预留的资源永远无法提交和释放了。

那么悬挂是如何产生的呢?



如上图所示，在执行参与者 A 的一阶段 Try 方法时，出现网路拥堵，由于 Seata 全局事务有超时限制，执行 Try 方法超时后，TM 决议全局回滚，回滚完成后如果此时 RPC 请求才到达参与者 A，执行 Try 方法进行资源预留，从而造成悬挂。

Seata 是怎么处理悬挂的呢?

在 TCC 事务控制表记录状态的字段 status 中增加一个状态：

- suspended：4

当执行二阶段 Cancel 方法时，如果发现 TCC 事务控制表没有相关记录，说明二阶段 Cancel 方法优先一阶段 Try 方法执行，因此插入一条 status=4 状态的记录，当一阶段 Try 方法后面执行时，判断 status=4 ，则说明有二阶段 Cancel 已执行，并返回 false 以阻止一阶段 Try 方法执行成功。

2.6 Spring Cloud Alibaba整合Seata TCC实战

业务场景

用户下单，整个业务逻辑由三个微服务构成：

- 库存服务：对给定的商品扣除库存数量。
- 订单服务：根据采购需求创建订单。

- 帐户服务：从用户帐户中扣除余额。

1) 环境准备

- 父pom指定微服务版本

Spring Cloud Alibaba Version	Spring Cloud Version	Spring Boot Version	Seata Version
2022.0.0.0	2022.0.0	3.0.2	1.7.0

- 启动Seata Server(TC)端，Seata Server使用nacos作为配置中心和注册中心
- 启动nacos服务

2) 微服务导入seata依赖

spring-cloud-starter-alibaba-seata内部集成了seata，并实现了xid传递

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

3）微服务application.yml中添加seata配置

```
seata:
  application-id: ${spring.application.name}
  # seata 服务分组，要与服务端配置service.vgroup_mapping的后缀对应
  tx-service-group: default_tx_group
  registry:
    # 指定nacos作为注册中心
    type: nacos
    nacos:
      application: seata-server
      server-addr: 127.0.0.1:8848
      namespace:
      group: SEATA_GROUP
```

```
13
14    config:
15      # 指定nacos作为配置中心
16      type: nacos
17      nacos:
18        server-addr: 127.0.0.1:8848
19        namespace: 7e838c12-8554-4231-82d5-6d93573ddf32
20        group: SEATA_GROUP
21        data-id: seataServer.properties
```

注意：请确保client与server的注册中心和配置中心namespace和group一致

4）定义TCC接口

TCC相关注解如下：

- @LocalTCC 适用于SpringCloud+Feign模式下的TCC，@LocalTCC一定需要注解在接口上，此接口可以是寻常的业务接口，只要实现了TCC的两阶段提交对应方法便可
- @TwoPhaseBusinessAction 注解try方法，其中name为当前tcc方法的bean名称，写方法名便可（全局唯一），commitMethod指向提交方法，rollbackMethod指向事务回滚方法。指定好三个方法之后，seata会根据全局事务的成功或失败，去帮我们自动调用提交方法或者回滚方法。
- @BusinessActionContextParameter 注解可以将参数传递到二阶段（commitMethod/rollbackMethod）的方法。
- BusinessActionContext 便是指TCC事务上下文

```
1  /**
2   * @author Fox
3   *
4   * 通过 @LocalTCC 这个注解，RM 初始化的时候会向 TC 注册一个分支事务。
5   */
6  @LocalTCC
7  public interface OrderService {
8
9      /**
10      * TCC的try方法：保存订单信息，状态为支付中
11      *
12      * 定义两阶段提交，在try阶段通过@TwoPhaseBusinessAction注解定义了分支事务的
   resourceId，commit和 cancel 方法
13      *  name = 该tcc的bean名称,全局唯一
14      *  commitMethod = commit 为二阶段确认方法
```

```
     *  rollbackMethod = rollback 为二阶段取消方法
     *  BusinessActionContextParameter注解 传递参数到二阶段中
     *  useTCCFence seata1.5.1的新特性，用于解决TCC幂等，悬挂，空回滚问题，需增加日志表 tcc_fence_log
     */
    @TwoPhaseBusinessAction(name = "prepareSaveOrder", commitMethod = "commit", rollbackMethod = "rollback", useTCCFence = true)
    Order prepareSaveOrder(OrderVo orderVo, @BusinessActionContextParameter(paramName = "orderId") Long orderId);

    /**
     *
     * TCC的confirm方法：订单状态改为支付成功
     *
     * 二阶段确认方法可以另命名，但要保证与commitMethod一致
     * context可以传递try方法的参数
     *
     * @param actionContext
     * @return
     */
    boolean commit(BusinessActionContext actionContext);

    /**
     * TCC的cancel方法：订单状态改为支付失败
     * 二阶段取消方法可以另命名，但要保证与rollbackMethod一致
     *
     * @param actionContext
     * @return
     */
    boolean rollback(BusinessActionContext actionContext);
}

/**
 * @author Fox
 *
 * 通过 @LocalTCC 这个注解，RM 初始化的时候会向 TC 注册一个分支事务。
 */
@LocalTCC
public interface StorageService {

```

```
52      /**
53       * Try: 库存-扣减数量，冻结库存+扣减数量
54       *
55       * 定义两阶段提交，在try阶段通过@TwoPhaseBusinessAction注解定义了分支事务的
    resourceId，commit和 cancel 方法
56       *  name = 该tcc的bean名称,全局唯一
57       *  commitMethod = commit 为二阶段确认方法
58       *  rollbackMethod = rollback 为二阶段取消方法
59       *  BusinessActionContextParameter注解 传递参数到二阶段中
60       *
61       * @param commodityCode 商品编号
62       * @param count 扣减数量
63       * @return
64       */
65      @TwoPhaseBusinessAction(name = "deduct", commitMethod = "commit", rollbackMethod =
    "rollback", useTCCFence = true)
66      boolean deduct(@BusinessActionContextParameter(paramName = "commodityCode") String
    commodityCode,
67                     @BusinessActionContextParameter(paramName = "count") int count);
68
69      /**
70       *
71       * Confirm: 冻结库存-扣减数量
72       * 二阶段确认方法可以另命名，但要保证与commitMethod一致
73       * context可以传递try方法的参数
74       *
75       * @param actionContext
76       * @return
77       */
78      boolean commit(BusinessActionContext actionContext);
79
80      /**
81       * Cancel: 库存+扣减数量，冻结库存-扣减数量
82       * 二阶段取消方法可以另命名，但要保证与rollbackMethod一致
83       *
84       * @param actionContext
85       * @return
86       */
87      boolean rollback(BusinessActionContext actionContext);
88  }
```

```java

/**
 * @author Fox
 *
 * 通过 @LocalTCC 这个注解，RM 初始化的时候会向 TC 注册一个分支事务。
 */
@LocalTCC
public interface AccountService {

    /**
     * 用户账户扣款
     *
     * 定义两阶段提交，在try阶段通过@TwoPhaseBusinessAction注解定义了分支事务的resourceId，commit和 cancel 方法
     *  name = 该tcc的bean名称,全局唯一
     *  commitMethod = commit 为二阶段确认方法
     *  rollbackMethod = rollback 为二阶段取消方法
     *
     * @param userId
     * @param money 从用户账户中扣除的金额
     * @return
     */
    @TwoPhaseBusinessAction(name = "debit", commitMethod = "commit", rollbackMethod = "rollback", useTCCFence = true)
    boolean debit(@BusinessActionContextParameter(paramName = "userId") String userId,
                  @BusinessActionContextParameter(paramName = "money") int money);

    /**
     * 提交事务，二阶段确认方法可以另命名，但要保证与commitMethod一致
     * context可以传递try方法的参数
     *
     * @param actionContext
     * @return
     */
    boolean commit(BusinessActionContext actionContext);

    /**
     * 回滚事务，二阶段取消方法可以另命名，但要保证与rollbackMethod一致
     *
     * @param actionContext
```

```
     * @return
     */
    boolean rollback(BusinessActionContext actionContext);
}
```

TCC 幂等、悬挂和空回滚问题如何解决?

TCC 模式中存在的三大问题是幂等、悬挂和空回滚。在 Seata1.5.1 版本中，增加了一张事务控制表，表名是 tcc_fence_log 来解决这个问题。而在@TwoPhaseBusinessAction 注解中提到的属性 useTCCFence 就是来指定是否开启这个机制，这个属性值默认是 false。

5）微服务增加tcc_fence_log日志表

```
# tcc_fence_log 建表语句如下（MySQL 语法）
CREATE TABLE IF NOT EXISTS `tcc_fence_log`
(
    `xid`           VARCHAR(128)  NOT NULL COMMENT 'global id',
    `branch_id`     BIGINT        NOT NULL COMMENT 'branch id',
    `action_name`   VARCHAR(64)   NOT NULL COMMENT 'action name',
    `status`        TINYINT       NOT NULL COMMENT
'status(tried:1;committed:2;rollbacked:3;suspended:4)',
    `gmt_create`    DATETIME(3)   NOT NULL COMMENT 'create time',
    `gmt_modified`  DATETIME(3)   NOT NULL COMMENT 'update time',
    PRIMARY KEY (`xid`, `branch_id`),
    KEY `idx_gmt_modified` (`gmt_modified`),
    KEY `idx_status` (`status`)
) ENGINE = InnoDB
DEFAULT CHARSET = utf8mb4;
```

6）TCC接口的业务实现

参考课堂代码

7） 在全局事务发起者中添加@GlobalTransactional注解

核心代码

```
@GlobalTransactional(name="createOrder",rollbackFor=Exception.class)
public Order saveOrder(OrderVo orderVo) {
```

```
    log.info("=============用户下单=================");
    log.info("当前 XID: {}", RootContext.getXID());

    //获取全局唯一订单号  测试使用
    Long orderId = UUIDGenerator.generateUUID();

    //阶段一:  创建订单
    Order order = orderService.prepareSaveOrder(orderVo,orderId);

    //扣减库存
    storageFeignService.deduct(orderVo.getCommodityCode(), orderVo.getCount());
    //扣减余额
    accountFeignService.debit(orderVo.getUserId(), orderVo.getMoney());

    return order;
}
```

8）测试分布式事务是否生效

- 分布式事务成功，模拟正常下单、扣库存，扣余额
- 分布式事务失败，模拟下单扣库存成功、扣余额失败，事务是否回滚