

Seata 分布式事务框架

**1.Seata**简介

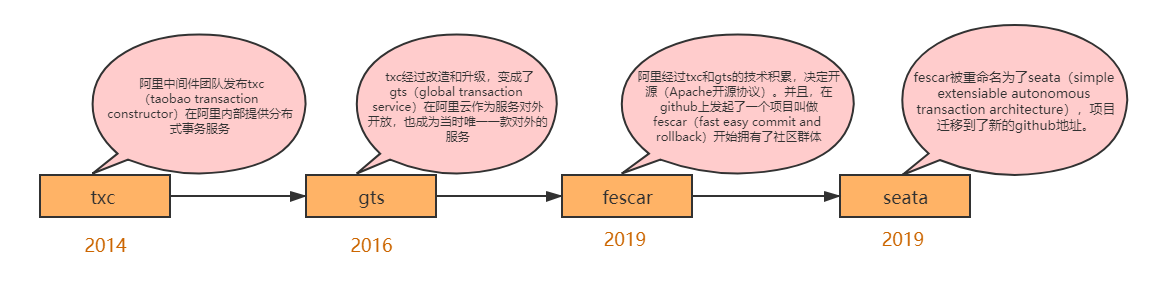
**1.1** 简介

Seata 是⼀款开源的分布式事务框架。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。在 Seata 开源之前， Seata 对应的内部版本在阿⾥经济体内部⼀直扮演着分布式⼀致性中间件的⻆ ⾊，帮助经济体平稳的度过历年的双11，对各业务单元业务进⾏了有⼒的⽀撑。经过多年沉淀与积累， 商业化产品先后在阿⾥云、⾦融云进⾏售卖。 2019.1 为了打造更加完善的技术⽣态和普惠技术成果， Seata 正式宣布对外开源，未来 Seata 将以社区共建的形式帮助其技术更加可靠与完备。

Seata： <https://seata.io/zh-cn/index.html>



发展史



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

**1.2** 特⾊功能

1. 微服务框架⽀持

⽬前已⽀持 Dubbo 、 Spring Cloud 、 Sofa-RPC 、 Motan 和 grpc 等RPC框架，其他框架持续集成中

2. AT 模式

提供⽆侵⼊⾃动补偿的事务模式，⽬前已⽀持 MySQL 、 Oracle 、 PostgreSQL和 TiDB的AT模式， H2 开发中

3. TCC 模式

⽀持 TCC 模式并可与AT 混⽤，灵活度更⾼

4. SAGA 模式

为⻓事务提供有效的解决⽅案

5. XA 模式

⽀持已实现 XA 接⼝的数据库的 XA 模式

6. ⾼可⽤

⽀持基于数据库存储的集群模式，⽔平扩展能⼒强

**1.3** **Seata** 产品模块

Seata 中有三⼤模块，分别是 TM 、RM 和 TC。其中 TM 和 RM 是作为 Seata 的客户端与业务系统集 成在⼀起， TC 作为 Seata 的服务端独⽴部署。

**TC** **(Transaction** **Coordinator)** **-** 事务协调者

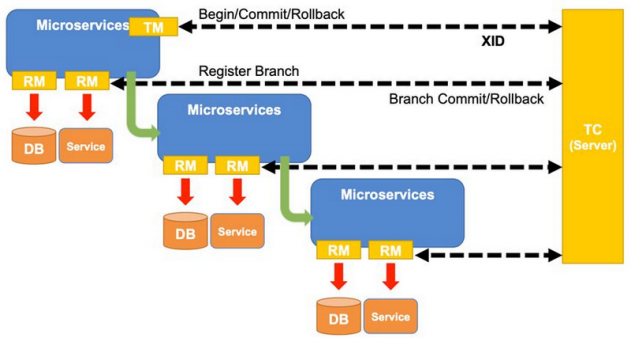
维护全局和分⽀事务的状态，驱动全局事务提交或回滚。

**TM** **(Transaction** **Manager)** **-** 事务管理器

定义全局事务的范围：开始全局事务、提交或回滚全局事务。

**RM** **(Resource** **Manager)** **-** 资源管理器

管理分⽀事务处理的资源，与TC交谈以注册分⽀事务和报告分⽀事务的状态，并驱动分⽀事务提交或 回滚。



在 Seata 中，分布式事务的执⾏流程：

 TM 开启分布式事务, TM会 向 TC 注册全局事务记录；

 操作具体业务模块的数据库操作之前, RM 会向 TC 注册分⽀事务；

 当业务操作完事后.TM会通知 TC 提交/回滚分布式事务；



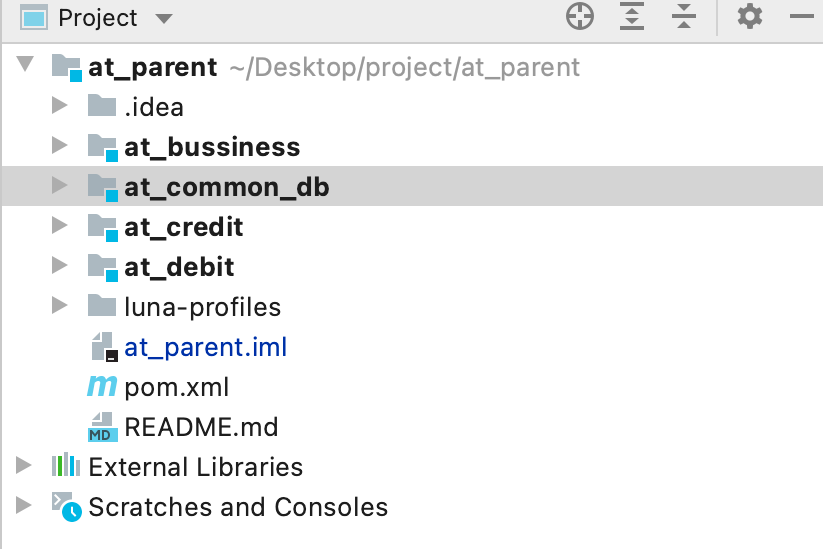
 TC 汇总事务信息，决定分布式事务是提交还是回滚；

 TC 通知所有 RM 提交/回滚 资源，事务⼆阶段结束。

**2.Seata-AT**模式

**2.1** 案例引⼊及问题剖析

1. 导⼊at\_parent⼯程 .



2. 执⾏初始化SQL脚本,⾸先创建2个数据库

seata\_bussiness/at\_credit/at\_debit在各⾃数据库执⾏SQL脚

use seata\_01；

SET NAMES utf8mb4;

SET FOREIGN\_KEY\_CHECKS = 0;

-- ----------------------------

-- Table structure for t\_acct\_balance

-- ----------------------------

DROP TABLE IF EXISTS `t\_acct\_balance`;

CREATE TABLE `t\_acct\_balance` (

`id` bigint NOT NULL COMMENT '账户主键',

`acct\_no` varchar(20) COLLATE utf8\_bin DEFAULT NULL COMMENT '账户号',

`amt` decimal(17,2) DEFAULT NULL COMMENT '账户余额额',

`tran\_time` datetime DEFAULT NULL COMMENT '交易时间',

`frozen\_amt` decimal(17,2) DEFAULT NULL COMMENT '冻结金额',

`client\_no` varchar(10) CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci NOT NULL DEFAULT '' COMMENT 'client\_no',

PRIMARY KEY (`id`) USING BTREE

) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8\_bin ROW\_FORMAT=DYNAMIC;

、

BEGIN;

INSERT INTO `t\_acct\_balance` VALUES (1, '6100010000100002', 1000.00, '2022-03-17 15:03:47', 0.00, '100001');

COMMIT;

SET FOREIGN\_KEY\_CHECKS = 1;

use seata\_02;

SET NAMES utf8mb4;

SET FOREIGN\_KEY\_CHECKS = 0;

-- ----------------------------

-- Table structure for t\_acct\_balance

-- ----------------------------

DROP TABLE IF EXISTS `t\_acct\_balance`;

CREATE TABLE `t\_acct\_balance` (

`id` bigint NOT NULL COMMENT '账户主键',

`acct\_no` varchar(20) COLLATE utf8\_bin DEFAULT NULL COMMENT '账户号',

`amt` decimal(17,2) DEFAULT NULL COMMENT '账户余额额',

`frozen\_amt` decimal(17,2) DEFAULT NULL COMMENT '冻结金额',

`client\_no` varchar(10) CHARACTER SET utf8 COLLATE utf8\_bin DEFAULT NULL,

`create\_time` datetime DEFAULT NULL COMMENT '交易时间',

PRIMARY KEY (`id`) USING BTREE

) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8\_bin ROW\_FORMAT=DYNAMIC;

-- ----------------------------

-- Records of t\_acct\_balance

-- ----------------------------

BEGIN;

INSERT INTO `t\_acct\_balance` VALUES (2, '6100010000100001', 100.00, 0.00, '100001', '2022-03-17 15:10:45');

COMMIT;

SET FOREIGN\_KEY\_CHECKS = 1;

seata\_points数据库

3. 案例测试

依次将**3**个服务启动**.**和**nacos**服务

访问路径为:

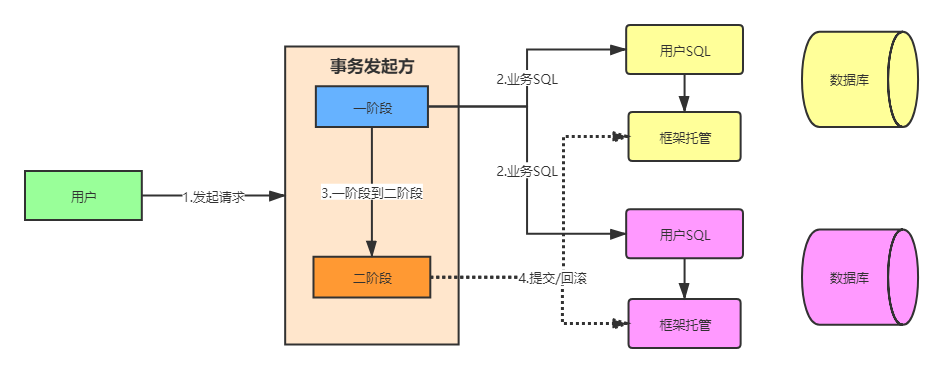
[http://localhost:9000/withdrawal](http://localhost:8000/test1)/sucess 正常访问数据完成转账

[http://localhost:9000/withdrawal](http://localhost:8000/test2)/error 账户不存在或已经销户 ,导致服务调⽤失败.则观察数据库, 经查账户'6100010000100002'余额减少但账户'6100010000100001'余额未变,产生挂账, 所以不满⾜事务的特性.

**2.2** **AT**模式介绍

AT 模式是⼀种⽆侵⼊的分布式事务解决⽅案。在 AT 模式下，⽤户只需关注⾃⼰的“业务 SQL”，⽤户

的 “业务 SQL” 作为⼀阶段， Seata 框架会⾃动⽣成事务的⼆阶段提交和回滚操作。



**2.3** **AT**模式原理

在介绍AT 模式的时候它是⽆侵⼊的分布式事务解决⽅案, 那么如何做到对业务的⽆侵⼊的呢?

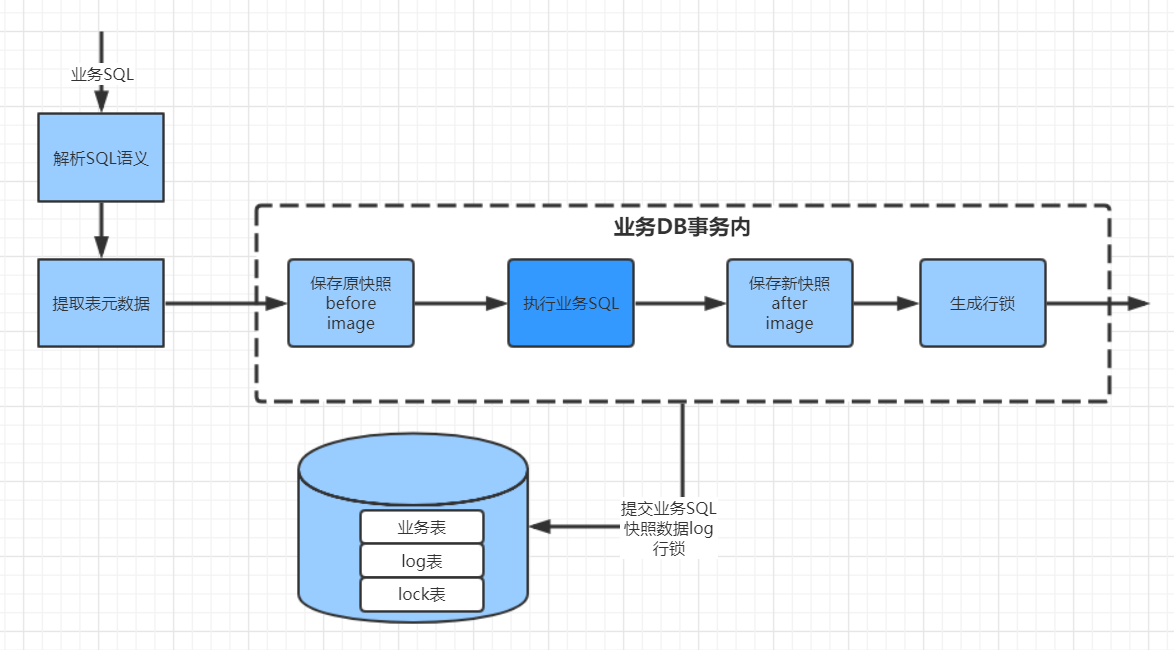
1. ⼀阶段

在⼀阶段，Seata 会拦截“业务 SQL”，⾸先解析 SQL 语义，找到“业务 SQL”要更新的业务数 据，

在业务数据被更新前，将其保存成“before image”，然后执⾏“业务 SQL”更新业务数据，在业 务数据更

新之后，再将其保存成“after image”，最后⽣成⾏锁。以上操作全部在⼀个数据库事务 内完成，这样

保证了⼀阶段操作的原⼦性。

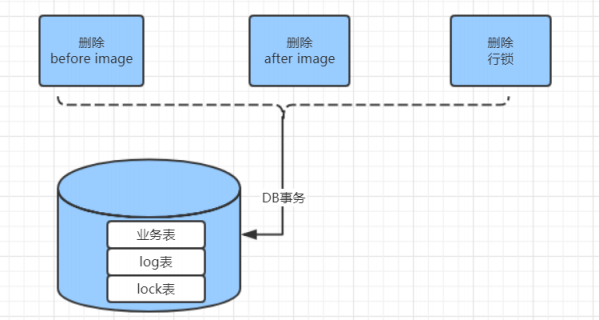


2. ⼆阶段

 提交

⼆阶段如果是提交的话，因为“业务 SQL”在⼀阶段已经提交⾄数据库， 所以 Seata 框架只需将

⼀阶段保存的快照数据和⾏锁删掉，完成数据清理即可。



1. 回滚

⼆阶段如果是回滚的话， Seata 就需要回滚⼀阶段已经执⾏的“业务 SQL”，还原业务数 据。回滚⽅

式便是⽤“before image”还原业务数据；但在还原前要⾸先要校验脏写，对据库当前业务数据”和

“after image”，如果两份数据完全⼀致就说明没有脏写，可以还原业 务数据，如果不⼀致就说明有脏

写，出现脏写就需要转⼈⼯处理。

AT 模式的⼀阶段、⼆阶段提交和回滚均由 Seata 框架⾃动⽣成，⽤户只需编写“业务 SQL” ，便能

轻松接⼊分布式事务， AT 模式是⼀种对业务⽆任何侵⼊的分布式事务解决⽅

**2.4** **AT**模式改造案例

**2.4.1** **Seata** **Server** **-** **TC**全局事务协调器

seata 事务分为三个模块： TC （事务协调器）、 TM （事务管理器）和RM （资源管理器），其 中 TM 和 RM 是嵌⼊在业务应⽤中的，⽽ TC 则是⼀个独⽴服务。

[Seata Server 就是 TC，直接从官⽅仓库下载启动即可，下载地址：](https://github.com/seata/seata/releases)  https://github.com/seata/seata/releases

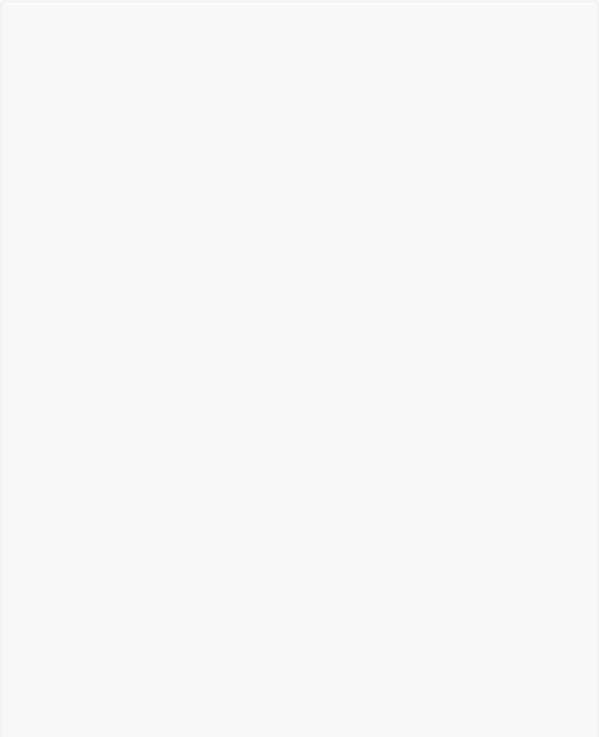
1. registry.conf

Seata Server 要向注册中⼼进⾏注册，这样，其他服务就可以通过注册中⼼去发现Seata Server，与 Seata Server 进⾏通信。

Seata ⽀持多款注册中⼼服务： nacos 、 eureka 、 redis 、zk 、 consul 、 etcd3 、 sofa。 我们项⽬中

要使⽤ nacos注册中⼼，nacos服务的连接地址、注册的服务名，这需要在 seata/conf/registry.conf

⽂件中进⾏配置：

registry { #注册中⼼

# file 、 nacos 、 eureka、 redis、 zk、 consul、 etcd3、 sofa

# 这⾥选择 nacos 注册配置

type = "nacos"

loadBalance = "RandomLoadBalance"

loadBalanceVirtualNodes = 10

nacos {

application = "seata-server" # 服务名称

serverAddr = "127.0.0.1:8848" # 服务地址

group = "SEATA\_GROUP" # 分组

namespace = ""

cluster = "default" # 集群

username = "nacos" # ⽤户名

password = "nacos" # 密码

}

eureka {

serviceUrl = "http://localhost:8761/eureka"

application = "default"

weight = "1"

}

redis {

serverAddr = "localhost:6379"

db = 0

password = ""

cluster = "default"

timeout = 0

}

zk {

cluster = "default"

serverAddr = "127.0.0.1:2181"

sessionTimeout = 6000

connectTimeout = 2000

username = ""

password = ""

|  |
| --- |
| } |
| 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 { #配置中⼼ |
| # file、 nacos 、 apollo、 zk、 consul、 etcd3 |
| type = "nacos" |
| nacos { |
| serverAddr = "127.0.0.1:8848" |
| namespace = "" |
| group = "SEATA\_GROUP" |
| username = "nacos" |
| password = "nacos" |
| } |
| consul { |
| serverAddr = "127.0.0.1:8500" |
| } |
| apollo { |
| appId = "seata-server" |
| apolloMeta = "http://192.168.1.204:8801" |
| namespace = "application" |
| apolloAccesskeySecret = "" |
| } |
| zk { |
| serverAddr = "127.0.0.1:2181" |
| sessionTimeout = 6000 |
| connectTimeout = 2000 |
| username = "" |
| password = ""  }  etcd3 {  serverAddr = "http://localhost:2379"  }  file {  name = "file.conf"  }  } |

2. 向nacos中添加配置信息

下载配置config.txt <https://github.com/seata/seata/tree/develop/script/config-center>

<https://seata.io/zh-cn/docs/user/configurations.html>针对每个⼀项配置介绍

将config.txt⽂件放⼊seata⽬录下⾯

修改config.txt信息

Server端存储的模式（store.mode）现有file,db,redis三种。主要存储全局事务会话信息,

分⽀事务信息, 锁记录表信息,seata-server默认是file模式。 file只能⽀持单机模式, 如果想要

⾼可⽤模式的话可以切换db或者redis.为了⽅便查看全局事务会话信息我们采⽤db数据库模

式。

 存储模式

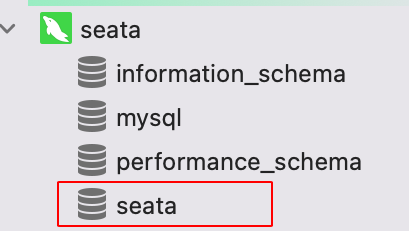


store.mode=db

|  |
| --- |
| store.db.datasource=druid  store.db.dbType=mysql  store.db.driverClassName=com.mysql.jdbc.Driver  store.db.url=jdbc:mysql://127.0.0.1:3306/seata?useUnicode=true store.db.user=root  store.db.password=root  store.db.minConn=5  store.db.maxConn=30  store.db.globalTable=global\_table  store.db.branchTable=branch\_table  store.db.queryLimit=100  store.db.lockTable=lock\_table  store.db.maxWait=5000 |

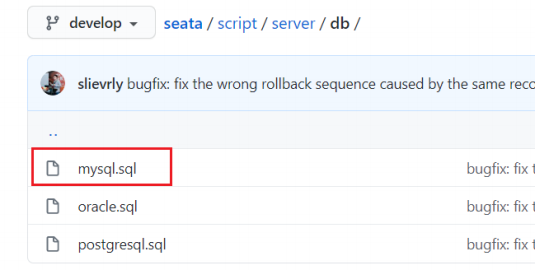
 mysql数据库连接信息

注意**1:** 需要创建seata数据库



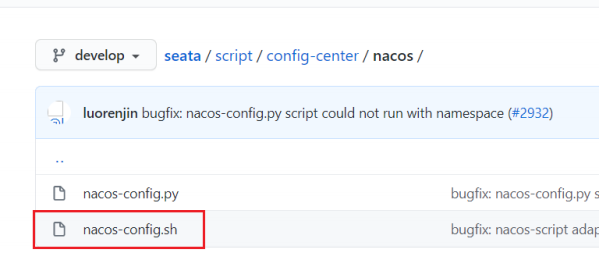
**注意2:** 需要创建global\_table/branch\_table/lock\_table三张表,seata1.0以上就不⾃带数

据库⽂件了，要⾃⼰去github下载，https://github.com/seata/seata/tree/develop/script/server/db

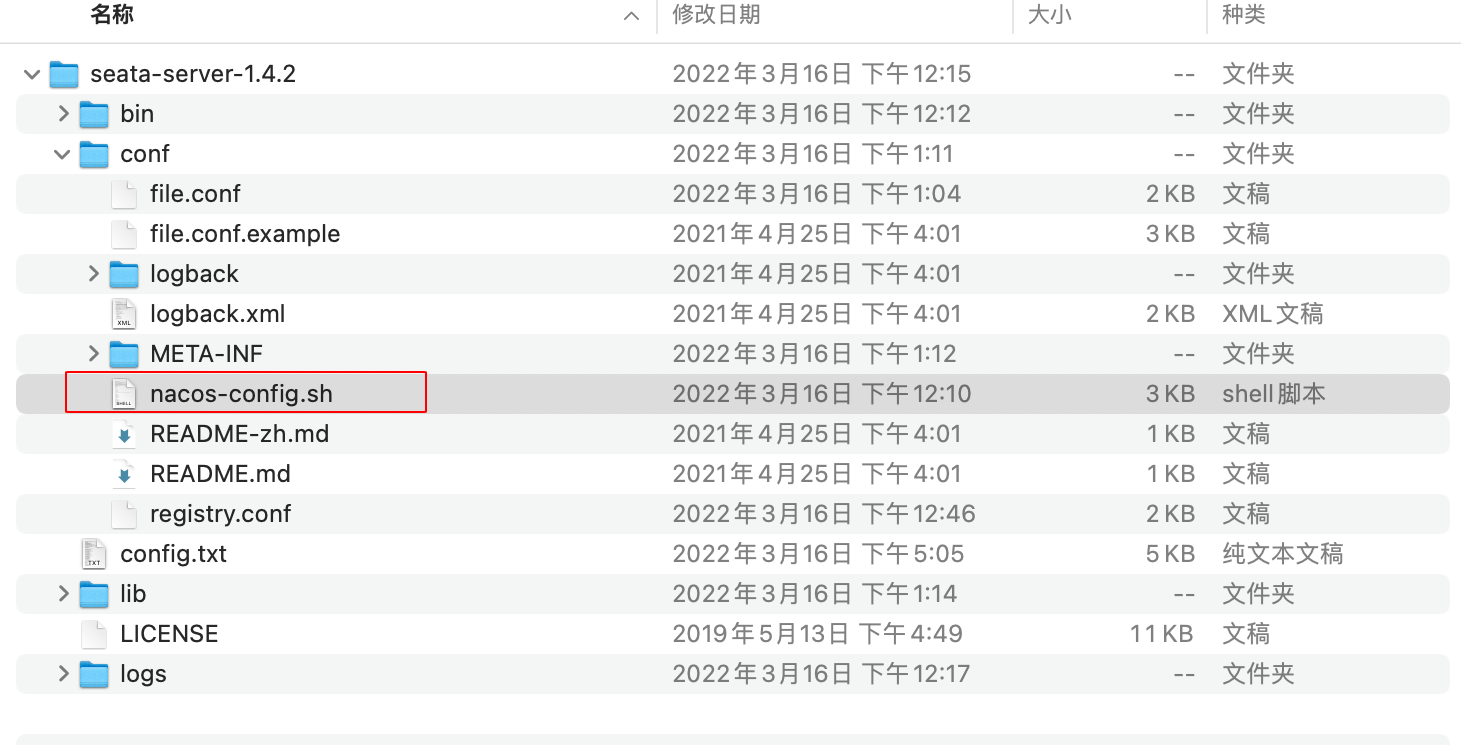


使⽤ nacos-config.sh ⽤于向 Nacos 中添加配置

下载地址:<https://github.com/seata/seata/tree/develop/script/config-center/nacos>



 将nacos-config.sh放在seata/conf⽂件夹中



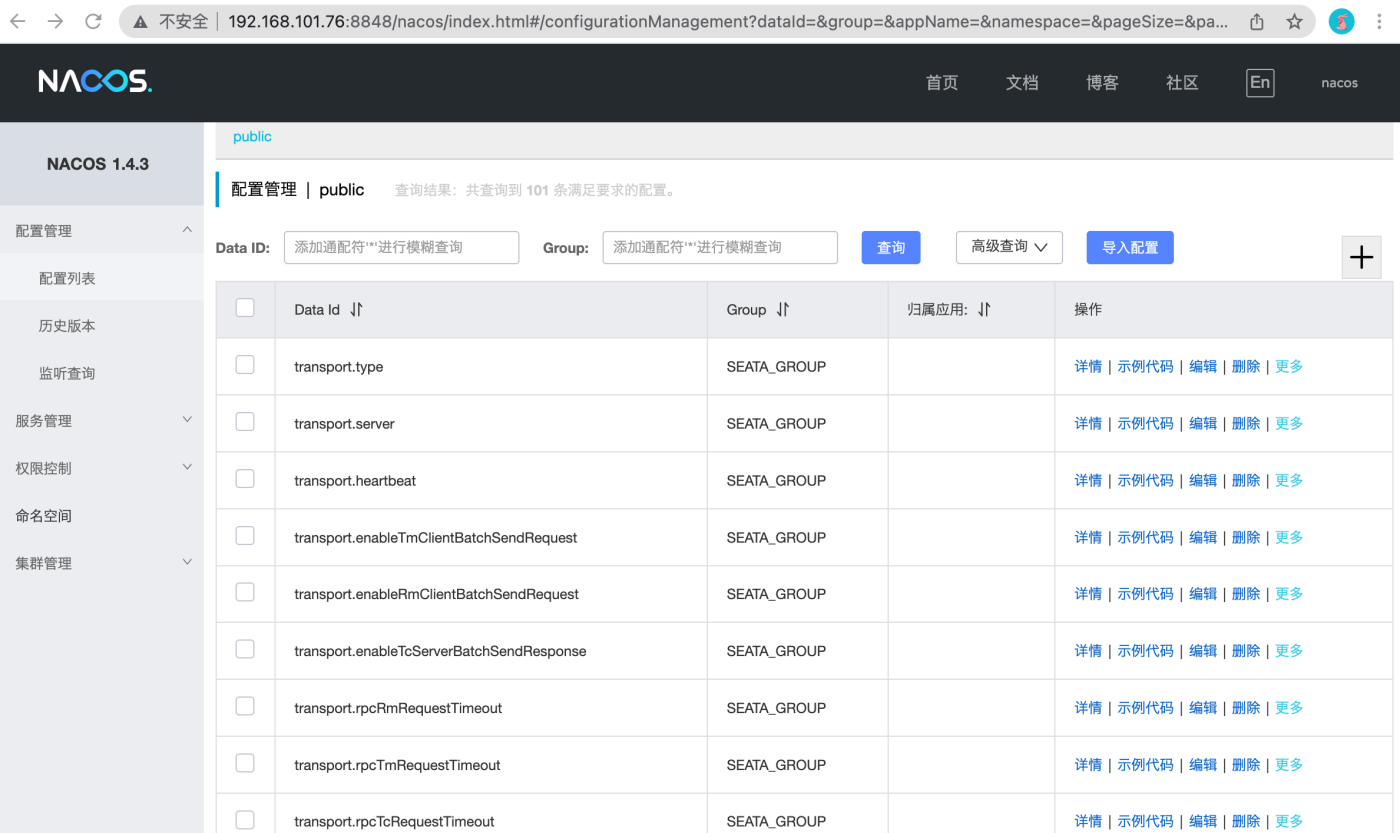
 打开git bash here 执⾏nacos-config.sh,需要提前将nacos启动

输⼊命令：



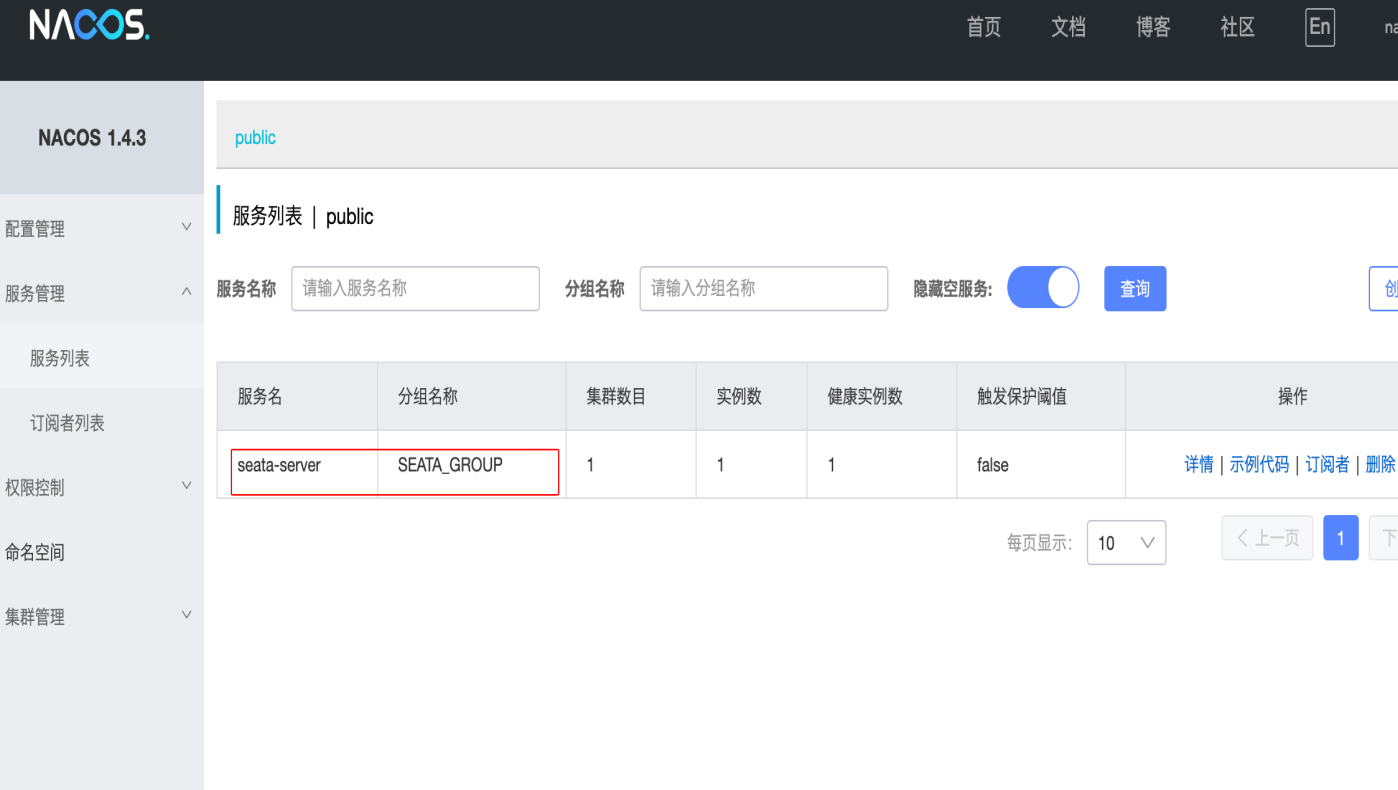
sh nacos-config.sh -h 127.0.0.1

登录nacos查看配置信息



1. 启动seata-server

观察nacos服务列表



**2.4.2** **TM/RM**端整合**Seata**

AT 模式在RM端需要 UNDO\_LOG 表,来记录每个RM的事务信息,主要包含数据修改前,后的相关信息, ⽤于回滚处理,所以在所有数据库中分别执⾏

-- 注意此处0.3.0+ 增加唯⼀索引 ux\_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,

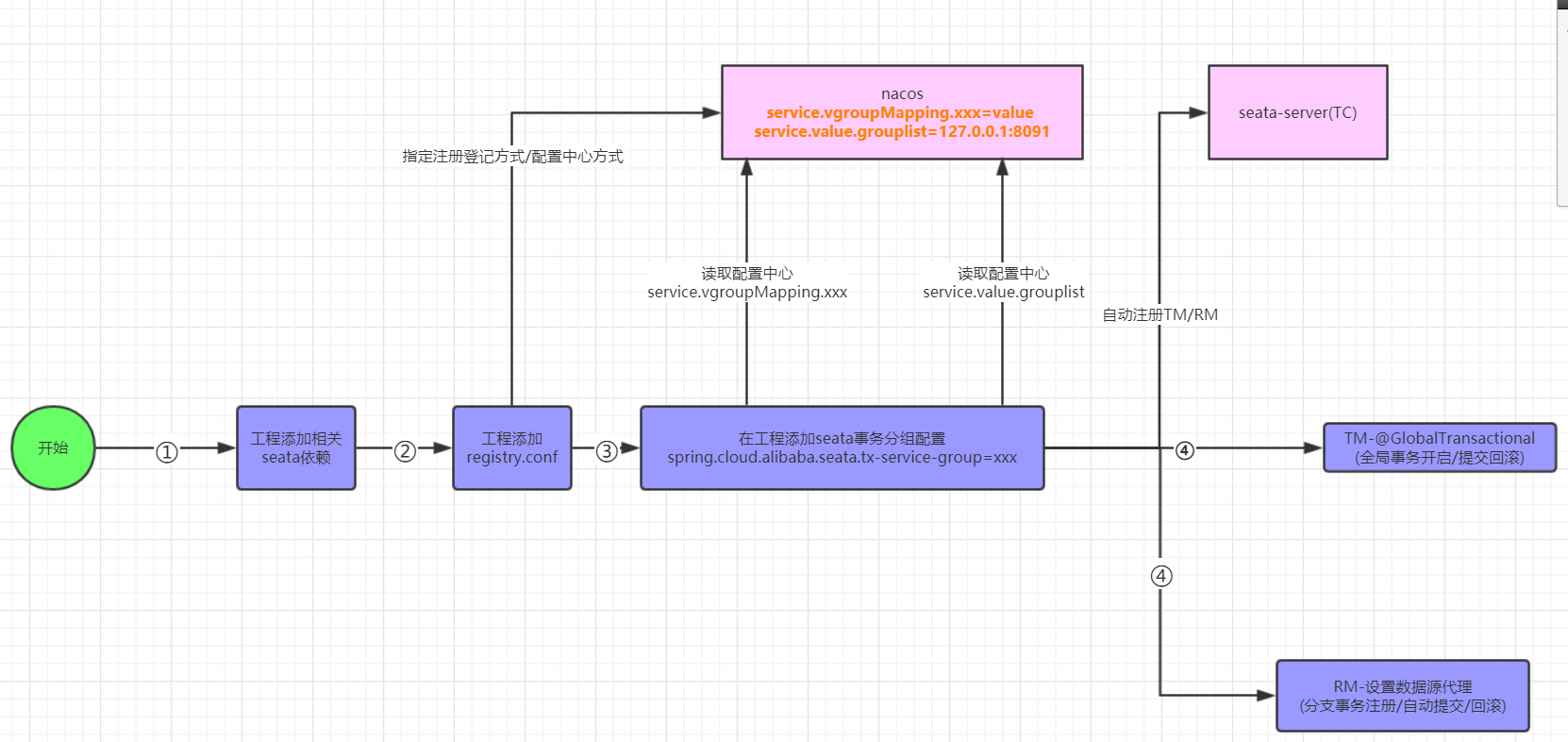
`ext` varchar(100) DEFAULT NULL,

PRIMARY KEY (`id`),

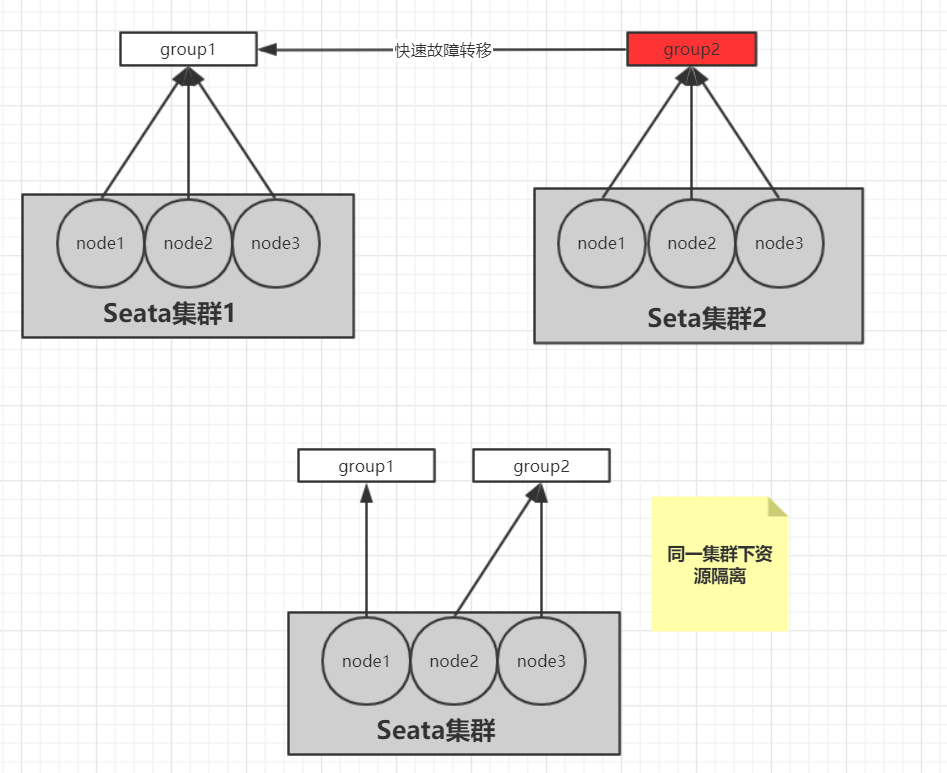
UNIQUE KEY `ux\_undo\_log` (`xid`,`branch\_id`)

) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;

TM/RM端整合seata⼀共有五个步骤:



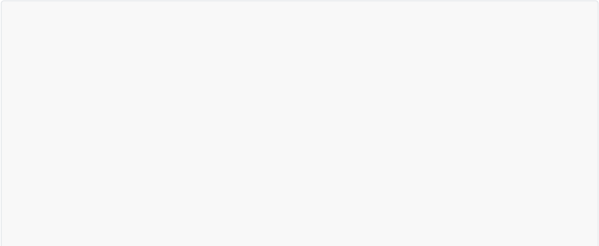
事务分组**:**



RM(事务管理器)端整合Seata与TM(事务管理器)端步骤类似,只不过不需要在⽅法添加 @GlobalTransactional注解,针对我们⼯程at\_bussiness是事务的发起者,所以是TM端,其它⼯程为 RM端. 所以我们只需要在at\_common\_db完成前4步骤即可

1. ⼯程中添加Seata依赖

at\_parent添加seata依赖管理,⽤于seata的版本锁定



<dependencyManagement>

<dependencies>

<!--spring cloud依赖管理，引⼊了Spring Cloud的版本-->

<dependency>

<groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-dependencies</artifactId> <version>Greenwich.RELEASE</version>

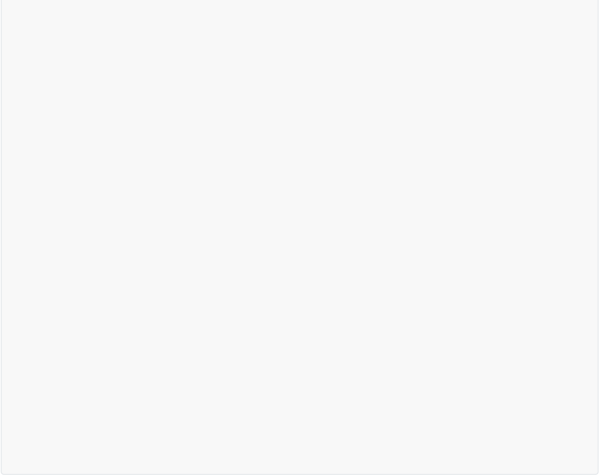
<type>pom</type>

<scope>import</scope>

</dependency>

<!--SCA -->





<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId> <version>5.1.47</version>

</dependency>

<!--SCA -->

<dependency>

<groupId>com.alibaba.cloud</groupId>

<artifactId>spring-cloud-alibaba-dependencies</artifactId> <version>2.1.0.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<!--SCA -->

<!--seata版本管理 , ⽤于锁定⾼版本的seata -->

<dependency>

<groupId>io.seata</groupId>

<artifactId>seata-all</artifactId>

<version>1.4.3</version>

</dependency>

</dependencies>

</dependencyManagement>

在at\_common\_db⼯程添加seata依赖

<!--seata依赖-->

<dependency>

<groupId>com.alibaba.cloud</groupId>

<artifactId>spring-cloud-alibaba-seata</artifactId>

<!--排除低版本seata依赖-->

<exclusions>

<exclusion>

<groupId>io.seata</groupId>

<artifactId>seata-all</artifactId>

</exclusion>

</exclusions>

</dependency>

<!--添加⾼版本seata依赖-->

<dependency>

<groupId>io.seata</groupId>

<artifactId>seata-all</artifactId>

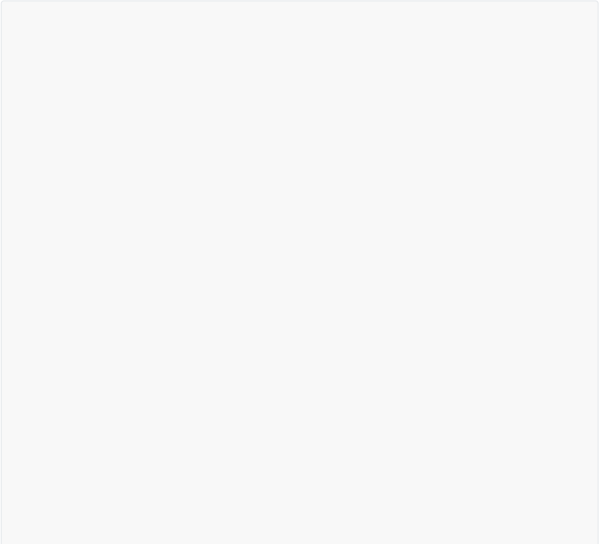
<version>1.4.3</version>

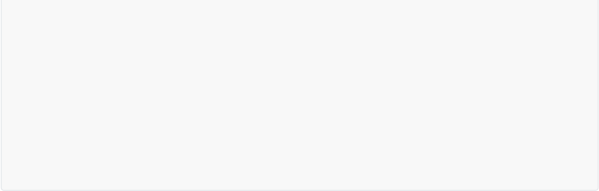
</dependency>

2. 在common⼯程添加registry.conf依赖

registry {

|  |
| --- |
| # file 、 nacos 、 eureka、 redis、 zk |
| type = "nacos" |
| nacos { |
| application = "seata-server" |
| serverAddr = "127.0.0.1:8848" |
| namespace = "" |
| group = "SEATA\_GROUP" |
| cluster = "default" |
| username = "nacos" |
| password = "nacos" |
| } |
| eureka { |
| serviceUrl = "http://127.0.0.1:8761/eureka" |
| application = "default" |
| weight = "1" |
| } |
| redis { |
| serverAddr = "localhost:6381" |
| db = "0" |
| } |
| zk { |
| cluster = "default" |
| serverAddr = "127.0.0.1:2181" |
| session.timeout = 6000 |
| connect.timeout = 2000 |
| } |
| file { |
| name = "file.conf" |
| } |
| } |
| config { |
| # file、 nacos 、 apollo、 zk |
| type = "nacos" |
| nacos { |
| application = "seata-server" |
| serverAddr = "127.0.0.1:8848" |
| group = "SEATA\_GROUP" |
| namespace = "" |
| cluster = "default" |
| username = "nacos" |
| password = "nacos" |
| } |
| apollo { |
| app.id = "fescar-server" |
| apollo.meta = "http://192.168.1.204:8801" |
| } |





zk {

serverAddr = "127.0.0.1:2181" session.timeout = 6000 connect.timeout = 2000

}

file {

name = "file.conf"

}

}

3. 添加公共配置

spring.cloud.alibaba.seata.tx-service-group=my\_seata\_tx\_group logging.level.io.seata=debug

4. 在每个模块下引⼊公共配置⽂件

profiles

active: seata

5. 编译数据源代理

package com.lagou.common\_db;

import com.alibaba.druid.pool.DruidDataSource;

import io.seata.rm.datasource.DataSourceProxy;

import

org.springframework.boot.context.properties.ConfigurationProperties; import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import javax.sql.DataSource;

@Configuration

public class DataSourceConfiguration {

/\*\*

\* 使⽤druid连接池 \*

\* @return \*/

@Bean

@ConfigurationProperties(prefix = "spring.datasource")

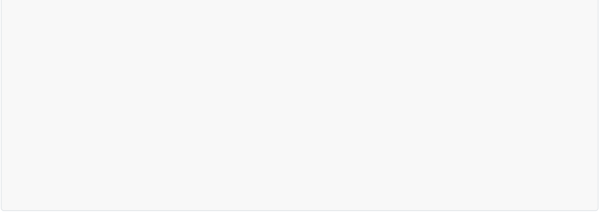
public DataSource druidDataSource() {

return new DruidDataSource();

}

/\*\*

\* 设置数据源代理-,完成分⽀事务注册/事务提交与回滚等操作



\*

\* @param druidDataSource

\* @return \*/

@Primary //设置⾸选数据源对象

@Bean("dataSource")

public DataSourceProxy dataSource(DataSource druidDataSource) { return new DataSourceProxy(druidDataSource);

}

}

启动扫描配置类,分别加载每个⼯程的启动类中

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class, scanBasePackages = "com.lagou")

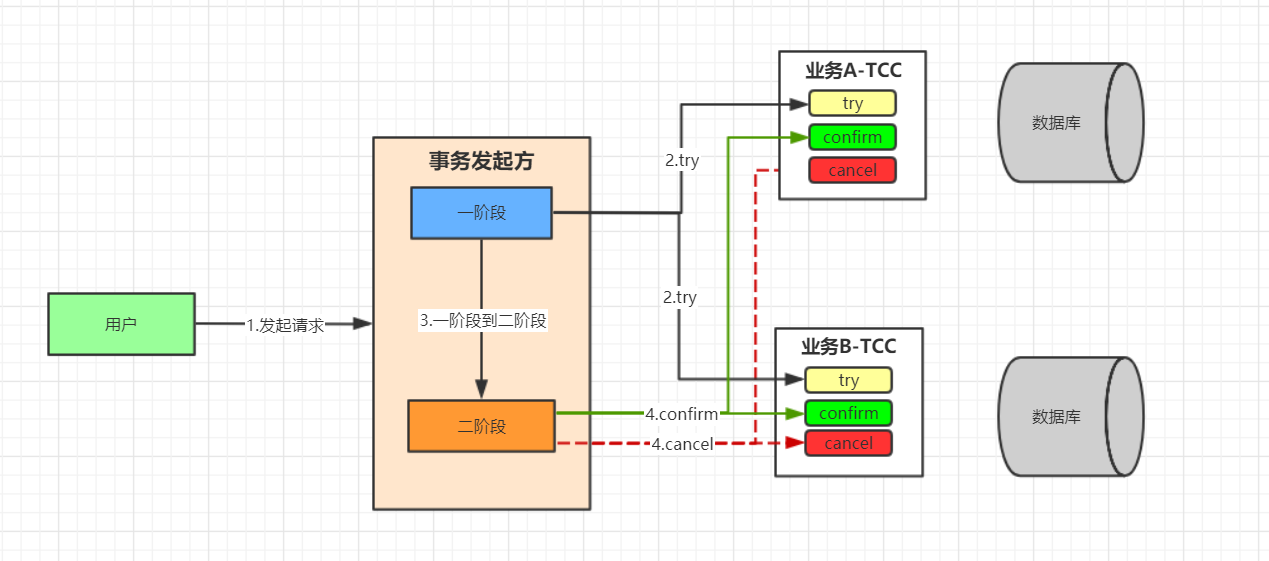
6. 添加注解@GlobalTransactional

*/\*\*  
 \* 转账支取  
 \*  
 \** ***@param acctNo*** *账号  
 \** ***@param amt*** *支取金额  
 \** ***@param clientNo*** *客户号  
 \*/*@GlobalTransactional(name = **"withdrawal"**, timeoutMills = 100000, rollbackFor = Exception.**class**)  
**public void** withdrawal(String acctNo, String settleAcctNo,  
 BigDecimal amt, String clientNo, String settleClientNo) {  
 *//交易流水* **long** refrence = **idWorker**.nextId();  
 *//先借记事件* **debitServiceFeign**.decrease(refrence, acctNo, settleAcctNo, amt, clientNo, settleClientNo);  
 *//后贷记事件* **creditServiceFeign**.increase(refrence, acctNo, settleAcctNo, amt, clientNo, settleClientNo);  
  
}

**3.Seata-TCC**模式

**3.1** **TCC**模式介绍

Seata 开源了 TCC 模式，该模式由蚂蚁⾦服贡献。 TCC 模式需要⽤户根据⾃⼰的业务场景实现 Try 、 Confirm 和 Cancel 三个操作；事务发起⽅在⼀阶段 执⾏ Try ⽅式，在⼆阶段提交执⾏ Confirm ⽅法，⼆阶段回滚执⾏ Cancel ⽅法。



TCC 三个⽅法描述：

 Try：资源的检测和预留；

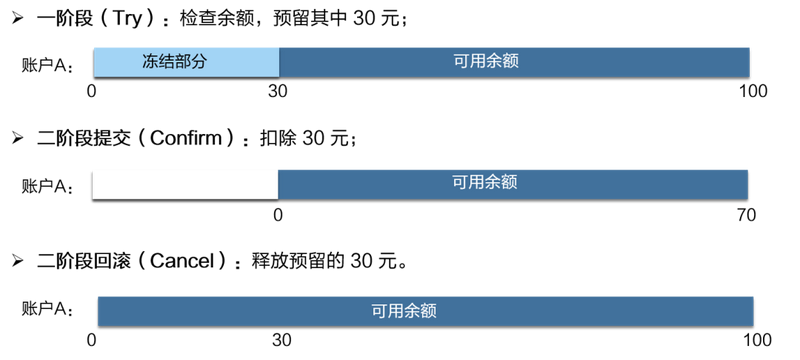
 Confirm：执⾏的业务操作提交；要求 Try 成功 Confirm ⼀定要能成功；

 Cancel：预留资源释放。

业务模型分 **2** 阶段设计：

⽤户接⼊ TCC ，最重要的是考虑如何将⾃⼰的业务模型拆成两阶段来实现。

以“扣钱”场景为例，在接⼊ TCC 前，对 A 账户的扣钱，只需⼀条更新账户余额的 SQL 便能完成； 但是在接⼊ TCC 之后，⽤户就需要考虑如何将原来⼀步就能完成的扣钱操作，拆成两阶段，实现成三个 ⽅法，并且保证⼀阶段 Try 成功的话 ⼆阶段 Confirm ⼀定能成功。



Try ⽅法作为⼀阶段准备⽅法，需要做资源的检查和预留。在扣钱场景下， Try 要做的事情是就是 检查账户余额是否充⾜，预留转账资⾦，预留的⽅式就是冻结 A 账户的 转账资⾦。 Try ⽅法执⾏之后， 账号 A 余额虽然还是 100，但是其中 30 元已经被冻结了，不能被其他事务使⽤。

⼆阶段 Confirm ⽅法执⾏真正的扣钱操作。 Confirm 会使⽤ Try 阶段冻结的资⾦，执⾏账号扣款。 Confirm ⽅法执⾏之后，账号 A 在⼀阶段中冻结的 30 元已经被扣除，账号 A 余额变成 70 元 。

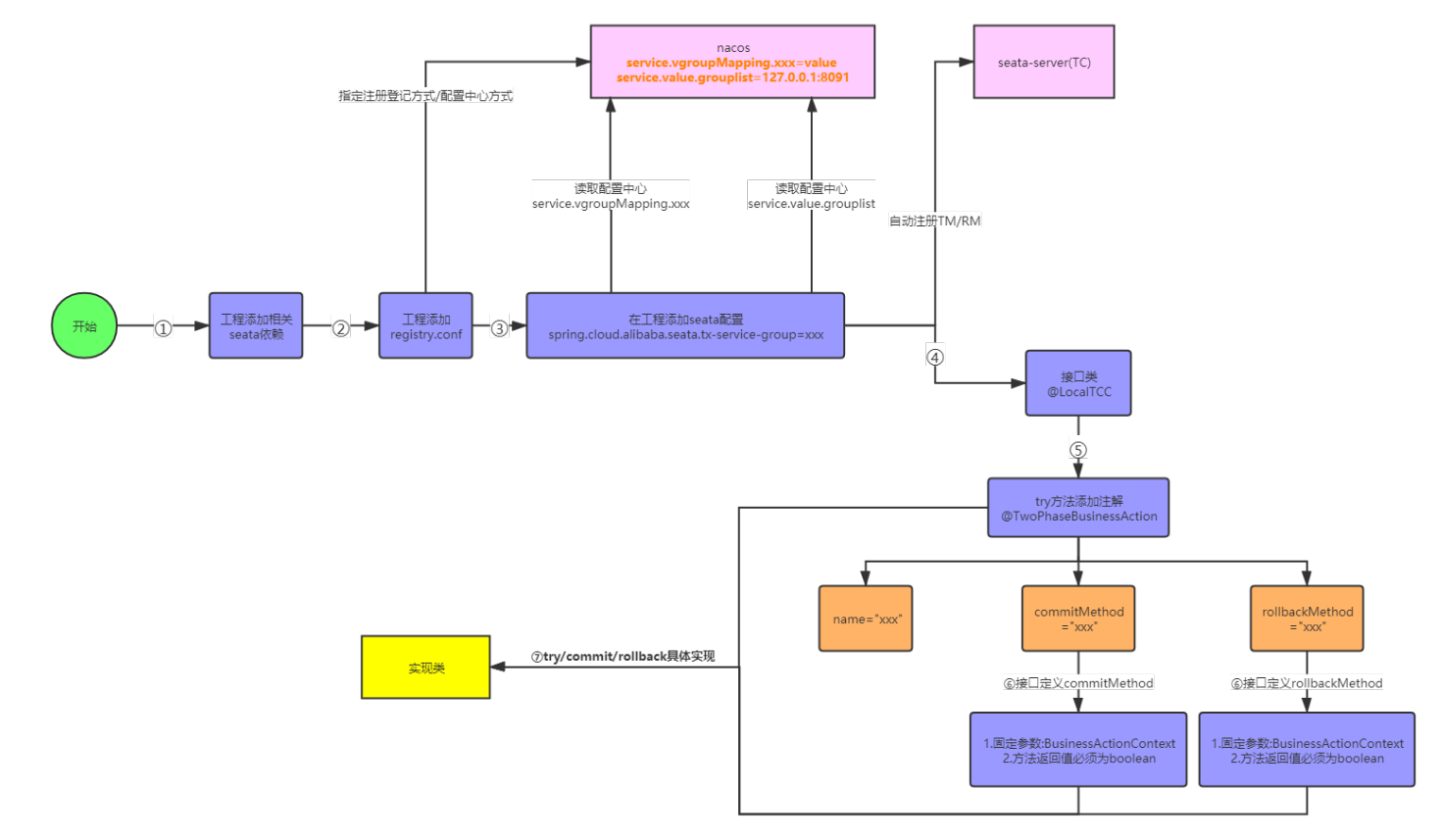
如果⼆阶段是回滚的话，就需要在 Cancel ⽅法内释放⼀阶段 Try 冻结的 30 元，使账号 A 的回到 初始状态， 100 元全部可⽤。

⽤户接⼊ **TCC** 模式，最重要的事情就是考虑如何将业务模型拆成 **2** 阶段，实现成 **TCC** 的 **3** 个⽅ 法，并且保证 **Try** 成功 **Confirm** ⼀定能成功。相对于 **AT** 模式， **TCC** 模式对业务代码有⼀定的侵⼊ 性，但是 **TCC** 模式⽆ **AT** 模式的全局⾏锁， **TCC** 性能会⽐ **AT** 模式⾼很多。

**3.2** **TCC**模式改造案例

**3.2.1** **RM**端改造

针对RM端,实现起来需要完成try/commit/rollback的实现,所以步骤相对较多但是前三步骤和AT模式⼀样

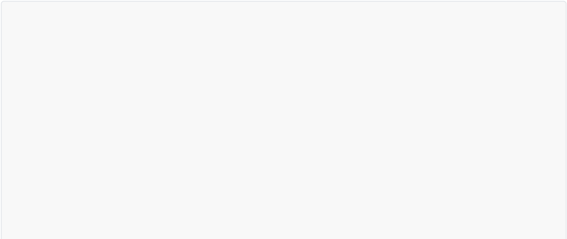
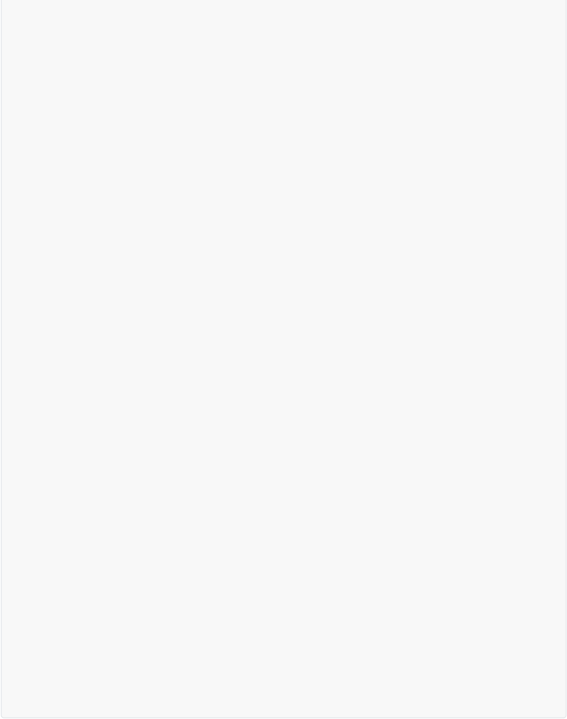


1. 修改数据库表结构,增加预留检查字段,⽤于提交和回滚

|  |
| --- |
| ALTER TABLE `seata\_01`.`t\_acct\_balance` ADD COLUMN `frozen\_amt` int(0) NULL DEFAULT 0 COMMENT '冻结金额' AFTER `amt`;  ALTER TABLE `seata\_02`.`t\_acct\_balance` ADD COLUMN `frozen\_amt` int(0) NULL DEFAULT 0 COMMENT '冻结金额' AFTER `amt`; |

2. at\_credit⼯程改造

1. 接⼝



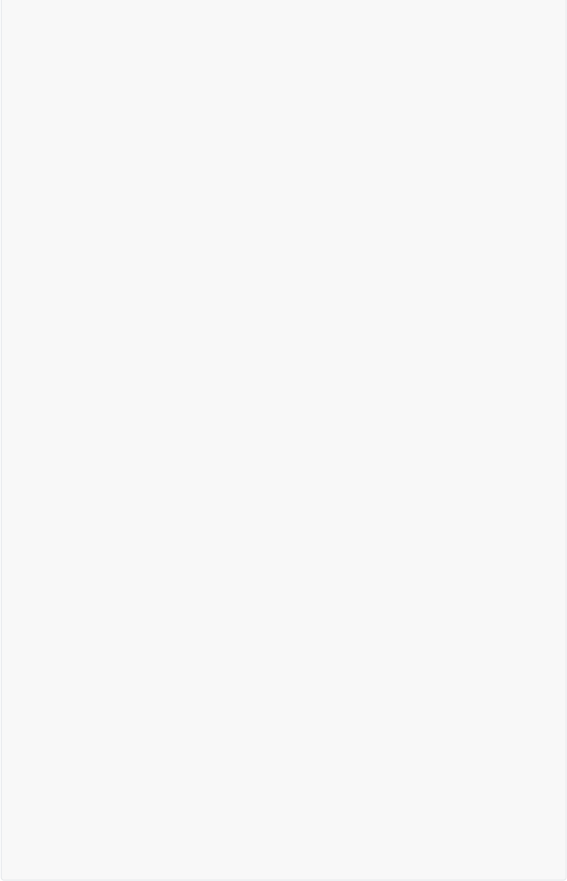
*/\*\*  
 \** ***@LocalTCC*** *该注解需要添加到上⾯描述的接⼝上，表示实现该接⼝的类被 seata 来管 理， seata 根据事务的状态，  
 \* ⾃动调⽤我们定义的⽅法，如果没问题则调⽤ Commit ⽅法，否则调⽤ Rollback ⽅ 法。  
 \*/*@LocalTCC  
**public interface** CreditService **extends** IService<Credit> {

*/\*\*  
 \** ***@TwoPhaseBusinessAction*** *描述⼆阶段提交  
 \* name: 为 tcc⽅法的 bean 名称，需要全局唯⼀，⼀般写⽅法名即可  
 \* commitMethod: Commit⽅法的⽅法名  
 \* rollbackMethod:Rollback⽅法的⽅法名  
 \** ***@BusinessActionContextParamete*** *该注解⽤来修饰 Try⽅法的⼊参，  
 \* 被修饰的⼊参可以在 Commit ⽅法和 Rollback ⽅法中通过  
 \* BusinessActionContext 获取。  
 \*/* @TwoPhaseBusinessAction(name = **"increaseTcc"**, commitMethod = **"increaseCommit"**, rollbackMethod = **"increaseRollback"**)  
 **public void** increase(@BusinessActionContextParameter(paramName = **"acctNo"**) String acctNo,  
 @BusinessActionContextParameter(paramName = **"settleAcctNo"**) String settleAcctNo,  
 @BusinessActionContextParameter(paramName = **"amt"**) BigDecimal amt,  
 @BusinessActionContextParameter(paramName = **"clientNo"**) String clientNo,  
 @BusinessActionContextParameter(paramName = **"settleClientNo"**) String settleClientNo);  
 **public boolean** increaseCommit(BusinessActionContext context);  
  
 **public boolean** increaseRollback(BusinessActionContext context);  
}

2. 实现类

*/\*\*  
 \* 转账记事件  
 \*  
 \** ***@param acctNo*** *账户  
 \** ***@param settleAcctNo*** *结算账号  
 \** ***@param tranAmt*** *交易金额  
 \** ***@param clientNo*** *客户号  
 \** ***@return*** *\*/***public void** decrease(String acctNo, String settleAcctNo, BigDecimal tranAmt, String clientNo, String settleClientNo) {  
 QueryWrapper<Debit> wrapper = **new** QueryWrapper<Debit>();

clientNo, String settleClientNo) {  
 QueryWrapper<Credit> wrapper = **new** QueryWrapper<Credit>();  
 wrapper.lambda().eq(Credit::getAcctNo, settleAcctNo);  
 wrapper.lambda().eq(Credit::getClientNo, settleClientNo);  
 Credit userCredit = **this**.getOne(wrapper);  
 **if** (userCredit == **null**) {  
 **throw new** RuntimeException(**"账户不存在，或已经销户！"**);  
 }  
 userCredit.setFrozenAmt(tranAmt);*//设置冻结金额* **this**.saveOrUpdate(userCredit);  
 }  
  
 */\*\*  
 \* 贷记提交  
 \*  
 \** ***@param context*** *上下文  
 \** ***@return*** *\*/* @Override  
 **public boolean** increaseCommit(BusinessActionContext context) {  
 *//查询账户信息* QueryWrapper<Credit> wrapper = **new** QueryWrapper<Credit>();  
 wrapper.lambda().eq(Credit::getAcctNo,  
 context.getActionContext(**"settleAcctNo"**));  
 wrapper.lambda().eq(Credit::getClientNo,  
 context.getActionContext(**"settleClientNo"**));  
 Credit userCredit = **this**.getOne(wrapper);  
 **if** (userCredit != **null**) {  
 *//增加⽤户余额* userCredit.setAmt(userCredit.getAmt().add(userCredit.getFrozenAmt()));  
 *//冻结金额清零* userCredit.setFrozenAmt(BigDecimal.***ZERO***);  
 **this**.saveOrUpdate(userCredit);  
 }  
 **return true**;  
 }  
  
 */\*\*  
 \* 贷记回退  
 \*  
 \** ***@param context*** *上下文  
 \** ***@return*** *\*/* @Override  
 **public boolean** increaseRollback(BusinessActionContext context) {  
 *//查询⽤户余额* QueryWrapper<Credit> wrapper = **new** QueryWrapper<Credit>();  
 wrapper.lambda().eq(Credit::getAcctNo,  
 context.getActionContext(**"settleAcctNo"**));  
 wrapper.lambda().eq(Credit::getClientNo,  
 context.getActionContext(**"settleClientNo"**));  
 Credit userCredit = **this**.getOne(wrapper);  
 **if** (userCredit != **null**) {  
 *//冻结金额清零* userCredit.setFrozenAmt(BigDecimal.***ZERO***);  
 **this**.saveOrUpdate(userCredit);  
 }  
 ***log***.info(**"--------->tcc="** + context.getXid() + **" 回滚成功!"**);  
 **return true**;  
 }  
}



String settleClientNo) {  
 QueryWrapper<Credit> wrapper = **new** QueryWrapper<Credit>();  
 wrapper.lambda().eq(Credit::getAcctNo, settleAcctNo);  
 wrapper.lambda().eq(Credit::getClientNo, settleClientNo);  
 Credit userCredit = **this**.getOne(wrapper);  
 **if** (userCredit == **null**) {  
 **throw new** RuntimeException(**"账户不存在，或已经销户！"**);  
 }  
 userCredit.setFrozenAmt(tranAmt);*//设置冻结金额* **this**.saveOrUpdate(userCredit);  
 }  
  
 */\*\*  
 \* 贷记提交  
 \*  
 \** ***@param context*** *上下文  
 \** ***@return*** *\*/* @Override  
 **public boolean** increaseCommit(BusinessActionContext context) {  
 *//查询账户信息* QueryWrapper<Credit> wrapper = **new** QueryWrapper<Credit>();  
 wrapper.lambda().eq(Credit::getAcctNo,  
 context.getActionContext(**"settleAcctNo"**));  
 wrapper.lambda().eq(Credit::getClientNo,  
 context.getActionContext(**"settleClientNo"**));  
 Credit userCredit = **this**.getOne(wrapper);  
 **if** (userCredit != **null**) {  
 *//增加⽤户余额* userCredit.setAmt(userCredit.getAmt().add(userCredit.getFrozenAmt()));  
 *//冻结金额清零* userCredit.setFrozenAmt(BigDecimal.***ZERO***);  
 **this**.saveOrUpdate(userCredit);  
 }  
 **return true**;  
 }

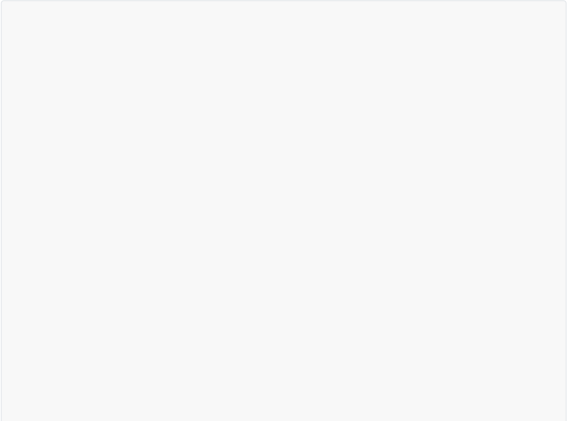
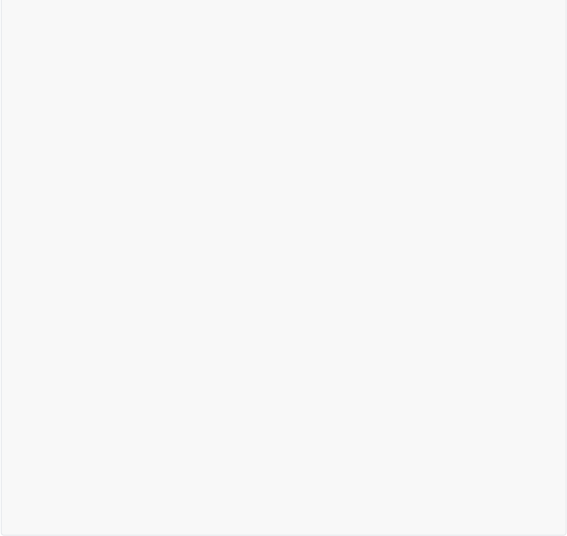


*/\*\*  
 \* 贷记回退  
 \*  
 \** ***@param context*** *上下文  
 \** ***@return*** *\*/* @Override  
 **public boolean** increaseRollback(BusinessActionContext context) {  
 *//查询⽤户余额* QueryWrapper<Credit> wrapper = **new** QueryWrapper<Credit>();  
 wrapper.lambda().eq(Credit::getAcctNo,  
 context.getActionContext(**"settleAcctNo"**));  
 wrapper.lambda().eq(Credit::getClientNo,  
 context.getActionContext(**"settleClientNo"**));  
 Credit userCredit = **this**.getOne(wrapper);  
 **if** (userCredit != **null**) {  
 *//冻结金额清零* userCredit.setFrozenAmt(BigDecimal.***ZERO***);  
 **this**.saveOrUpdate(userCredit);  
 }  
 ***log***.info(**"--------->tcc="** + context.getXid() + **" 回滚成功!"**);  
 **return true**;  
 }  
}

3. tcc\_debit⼯程改造

接⼝改造

*/\*\*  
 \** ***@LocalTCC*** *该注解需要添加到上⾯描述的接⼝上，表示实现该接⼝的类被 seata 来管 理， seata 根据事务的状态，  
 \* ⾃动调⽤我们定义的⽅法，如果没问题则调⽤ Commit ⽅法，否则调⽤ Rollback ⽅ 法。  
 \*/*@LocalTCC  
**public interface** DebitService **extends** IService<Debit> {

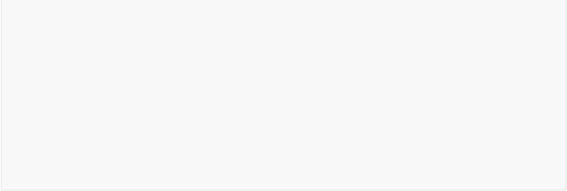


*/\*\*  
 \** ***@TwoPhaseBusinessAction*** *描述⼆阶段提交  
 \* name: 为 tcc⽅法的 bean 名称，需要全局唯⼀，⼀般写⽅法名即可  
 \* commitMethod: Commit⽅法的⽅法名  
 \* rollbackMethod:Rollback⽅法的⽅法名  
 \** ***@BusinessActionContextParamete*** *该注解⽤来修饰 Try⽅法的⼊参，  
 \* 被修饰的⼊参可以在 Commit ⽅法和 Rollback ⽅法中通过  
 \* BusinessActionContext 获取。  
 \*/* @TwoPhaseBusinessAction(name = **"decreaseTcc"**, commitMethod = **"decreaseCommit"**, rollbackMethod = **"decreaseRollback"**)  
 **public void** decrease(@BusinessActionContextParameter(paramName = **"acctNo"**)String acctNo,  
 @BusinessActionContextParameter(paramName = **"settleAcctNo"**) String settleAcctNo,  
 @BusinessActionContextParameter(paramName = **"amt"**) BigDecimal amt,  
 @BusinessActionContextParameter(paramName = **"clientNo"**) String clientNo,  
 @BusinessActionContextParameter(paramName = **"settleClientNo"**) String settleClientNo);  
  
 **public boolean** decreaseCommit(BusinessActionContext context);  
  
 **public boolean** decreaseRollback(BusinessActionContext context);  
  
  
}

 实现类改造

*/\*\*  
 \* 转账贷记事件  
 \*  
 \** ***@param acctNo*** *账户  
 \** ***@param settleAcctNo*** *结算账号  
 \** ***@param tranAmt*** *交易金额  
 \** ***@param clientNo*** *客户号  
 \** ***@return*** *\*/***public void** decrease(String acctNo, String settleAcctNo, BigDecimal tranAmt, String clientNo,  
 String settleClientNo) {  
 QueryWrapper<Debit> wrapper = **new** QueryWrapper<Debit>();  
 wrapper.lambda().eq(Debit::getAcctNo, acctNo);  
 wrapper.lambda().eq(Debit::getClientNo, clientNo);  
 Debit userCredit = **this**.getOne(wrapper);  
 **if** (userCredit == **null**) {  
 **throw new** RuntimeException(**"账户不存在，或已经销户！"**);  
 }  
 userCredit.setFrozenAmt(tranAmt);*//设置冻结金额* **this**.saveOrUpdate(userCredit);  
  
}

*/\*\*  
 \* 借记提交  
 \*  
 \** ***@param context*** *上下文  
 \** ***@return*** *\*/*@Override  
**public boolean** decreaseCommit(BusinessActionContext context) {  
 *//查询账户信息* QueryWrapper<Debit> wrapper = **new** QueryWrapper<Debit>();  
 wrapper.lambda().eq(Debit::getAcctNo,  
 context.getActionContext(**"acctNo"**));  
 wrapper.lambda().eq(Debit::getClientNo,  
 context.getActionContext(**"clientNo"**));  
 Debit userCredit = **this**.getOne(wrapper);  
 **if** (userCredit != **null**) {  
 *//增加⽤户余额* userCredit.setAmt(userCredit.getAmt().subtract(userCredit.getFrozenAmt()));  
 *//冻结金额清零* userCredit.setFrozenAmt(BigDecimal.***ZERO***);  
 **this**.saveOrUpdate(userCredit);  
 }  
 ***log***.info(**"--------->tcc="** + context.getXid() + **" 提交成功!"**);  
 **return true**;  
}  
  
*/\*\*  
 \* 借记回滚  
 \*  
 \** ***@param context*** *上下文  
 \** ***@return*** *\*/*@Override  
**public boolean** decreaseRollback(BusinessActionContext context) {  
 *//查询⽤户余额* QueryWrapper<Debit> wrapper = **new** QueryWrapper<Debit>();  
 wrapper.lambda().eq(Debit::getAcctNo,  
 context.getActionContext(**"acctNo"**));  
 wrapper.lambda().eq(Debit::getClientNo,  
 context.getActionContext(**"clientNo"**));  
 Debit userCredit = **this**.getOne(wrapper);  
 **if** (userCredit != **null**) {  
 *//冻结金额清零* userCredit.setFrozenAmt(BigDecimal.***ZERO***);  
 **this**.saveOrUpdate(userCredit);  
 }



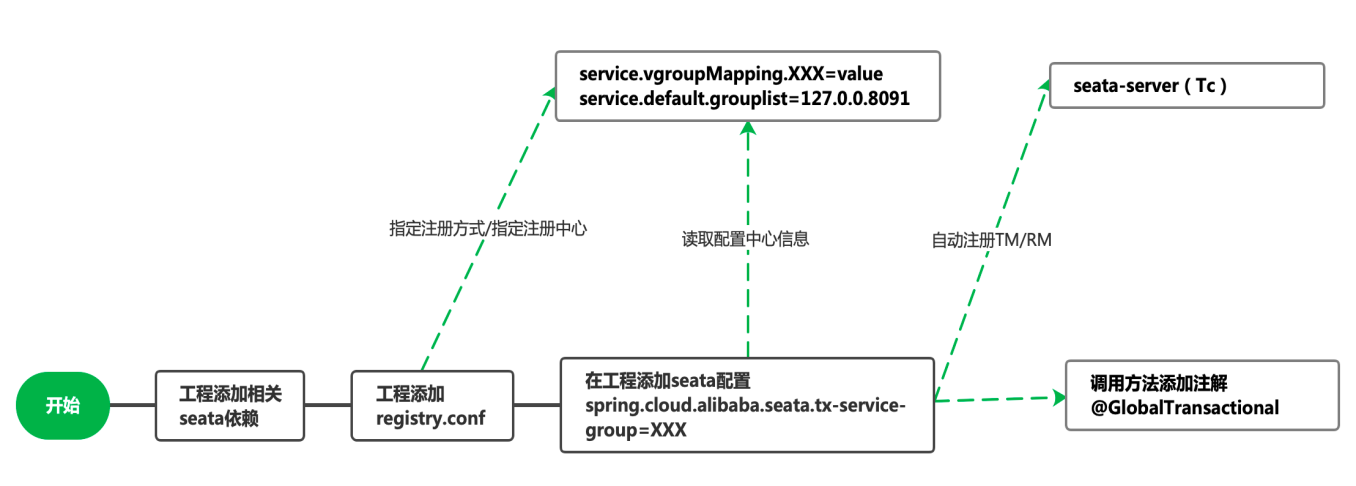
***log***.info(**"--------->tcc="** + context.getXid() + **" 回滚成功!"**);  
 **return true**;  
 }  
}

**3.2.2** **TM**端改造

针对我们⼯程tcc\_bussiness是事务的发起者,所以是TM端,其它⼯程为RM端. 所以我们只需要在 tcc\_common\_db完成即可,因为tcc\_bussiness⽅法⾥⾯没有对数据库操作 .所以只需要将之前AT模

式的代理数据源去掉即可.注意:如果tcc\_bussiness也对数据库操作了.也需要完成

try/commit/rollback的实现.



代码实现：

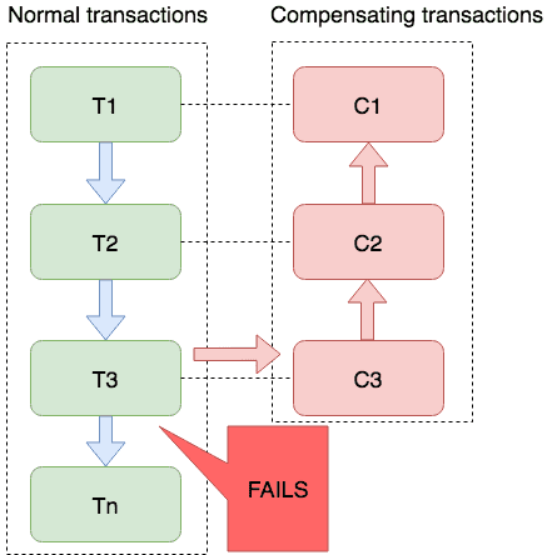
*/\*\*  
 \* 转账支取  
 \*  
 \** ***@param acctNo*** *账号  
 \** ***@param amt*** *支取金额  
 \** ***@param clientNo*** *客户号  
 \*/*@GlobalTransactional(name = **"withdrawal"**, timeoutMills = 100000, rollbackFor = Exception.**class**)  
**public void** withdrawal(String acctNo, String settleAcctNo,  
 BigDecimal amt, String clientNo, String settleClientNo) {  
 *//交易流水* **long** refrence = **idWorker**.nextId();  
  
 *//先借记事件* **debitServiceFeign**.decrease(refrence, acctNo, settleAcctNo, amt, clientNo, settleClientNo);  
 *//后贷记事件* **creditServiceFeign**.increase(refrence, acctNo, settleAcctNo, amt, clientNo, settleClientNo);  
  
}

**4.Seata-Saga**模式简介与三种模式对⽐

**4.1** **Saga**模式简单介绍

Saga 模式是 Seata 开源的⻓事务解决⽅案，将由蚂蚁⾦服主要贡献。在 Saga 模式下，分布式事务 内有多个参与者，每⼀个参与者都是⼀个冲正补偿服务，需要⽤户根据业务场景实现其正向操作和逆向 回滚操作。

分布式事务执⾏过程中，依次执⾏各参与者的正向操作，如果所有正向操作均执⾏成功，那么分布 式事务提交。如果任何⼀个正向操作执⾏失败，那么分布式事务会去退回去执⾏前⾯各参与者的逆向回 滚操作，回滚已提交的参与者，使分布式事务回到初始状态



适⽤场景：

 业务流程⻓、业务流程多

 参与者包含第三⽅公司或遗留系统服务，⽆法提供 TCC 模式要求的三个接⼝

 典型业务系统：如⾦融⽹络（与外部⾦融机构对接）、互联⽹微贷、渠道整合等业务系统

|  |  |  |  |
| --- | --- | --- | --- |
|  | **AT** | **TCC** | **Sage** |
| 集成难度 | 低 | ⾮常⾼ | 中等 |
| 隔离性 | 保证 | 保证 | 不保证 |
| 推荐度 | ⾼ | 中 | 低 |
| 数据库改造 | UNDO\_LOG | ⽆ | 流程与实例表 |
| 实现机制 | DataSource代理 | TCC实现 | 状态机 |
| 场景 | ⾃研项⽬全场景  拥有数据访问权限  快速集成场景 | 更⾼的性能要求  更复杂的场景 | ⻓流程  涉及⼤量第三⽅调⽤ |

**4.2** 三种模式对⽐

**AT** 模式是⽆侵⼊的分布式事务解决⽅案，适⽤于不希望对业务进⾏改造的场景，⼏乎**0**学习成本。 **TCC** 模式是⾼性能分布式事务解决⽅案，适⽤于核⼼系统等对性能有很⾼要求的场景。

Saga 模式是⻓事务解决⽅案，适⽤于业务流程⻓且需要保证事务最终⼀致性的业务系统， Saga 模式⼀ 阶段就会提交本地事务，⽆锁，⻓流程情况下可以保证性能，多⽤于渠道层、集成层业务系统。事务参 与者可能是其它公司的服务或者是遗留系统的服务，⽆法进⾏改造和提供 TCC 要求的接⼝，也可以使⽤ Saga 模。

At模式代码详见：<https://github.com/dingfan0327/at_parent.git>

Tcc模式代码详见： <https://github.com/dingfan0327/tcc_parent.git>

数据库脚本详见 [https://github.com/dingfan0327/at\_parent](https://github.com/dingfan0327/at_parent.git)/db目录