

1.Seata简介

1.1 简介

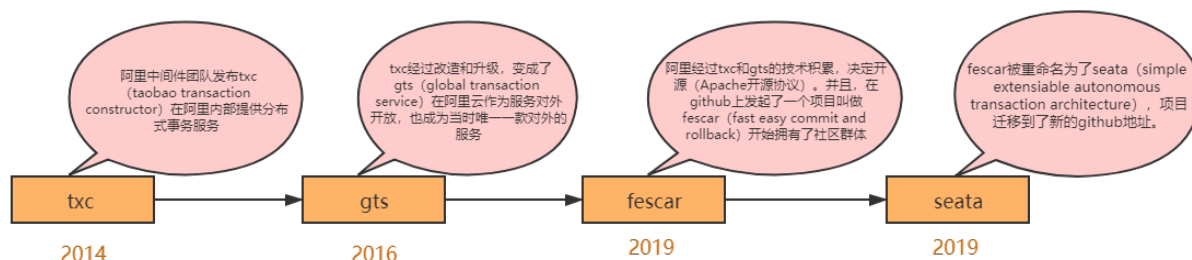
Seata 是一款开源的分布式事务框架。致力于在微服务架构下提供高性能和简单易用的分布式事务服务。在 Seata 开源之前，Seata 对应的内部版本在阿里经济体内部一直扮演着分布式一致性中间件的角色，帮助经济体平稳的度过历年的双11，对各业务单元业务进行了有力的支撑。经过多年沉淀与积累，商业化产品先后在阿里云、金融云进行售卖。2019.1 为了打造更加完善的技术生态和普惠技术成果，Seata 正式宣布对外开源，未来 Seata 将以社区共建的形式帮助其技术更加可靠与完备。

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



Simple Extensible Autonomous Transaction Architecture

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

为长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

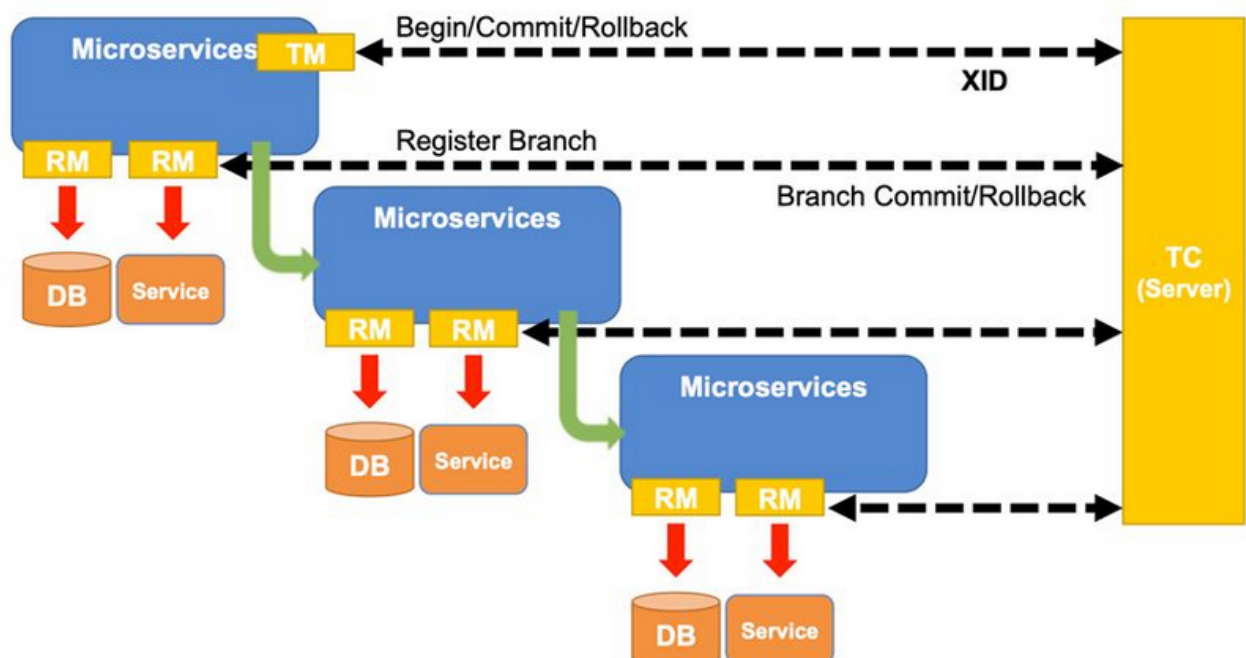
维护全局和分支事务的状态，驱动全局事务提交或回滚。

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：

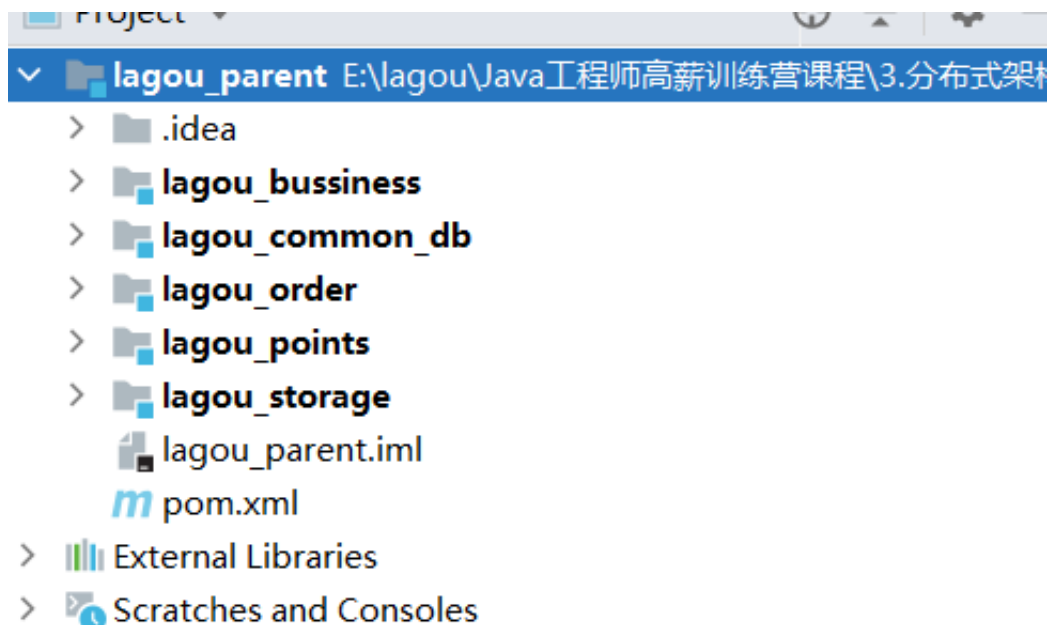
- TM 开启分布式事务, TM会 向 TC 注册全局事务记录；
- 操作具体业务模块的数据库操作之前, RM 会向 TC 注册分支事务；
- 当业务操作完事后.TM会通知 TC 提交/回滚分布式事务；

- 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lagou_parent工程.



2. 执行初始化SQL脚本,首先创建4个数据库
seata_bussiness/seata_order/seata_points/seata_storage,在各自数据库执行SQL脚本

ringCloud组件设计原理及实战（下） > seata分布式事务 > 3.资料 > sql初始化脚本

名称	修改日期	类型	大小
seata_order.sql	2020/12/2 星期三 16:...	SQL 文件	2 KB
seata_points.sql	2020/12/2 星期三 16:...	SQL 文件	2 KB
seata_storage.sql	2020/12/2 星期三 16:...	SQL 文件	2 KB

seata_order数据库

```
/*
Navicat Premium Data Transfer

Source Server        : localhost
Source Server Type   : MySQL
Source Server Version : 80020
Source Host          : localhost:3306
Source Schema        : seata_order

Target Server Type   : MySQL
Target Server Version : 80020
File Encoding        : 65001

Date: 02/12/2020 16:28:18
```

```

*/

SET NAMES utf8mb4;
SET FOREIGN_KEY_CHECKS = 0;

-- -----
-- Table structure for t_order
-- -----

DROP TABLE IF EXISTS `t_order`;
CREATE TABLE `t_order` (
  `id` bigint(0) NOT NULL COMMENT '订单id',
  `goods_Id` int(0) NULL DEFAULT NULL COMMENT '商品ID',
  `num` int(0) NULL DEFAULT NULL COMMENT '商品数量',
  `money` decimal(10, 0) NULL DEFAULT NULL COMMENT '商品总金额',
  `create_time` datetime(0) NULL DEFAULT NULL COMMENT '订单创建时间',
  `username` varchar(50) CHARACTER SET utf8 COLLATE utf8_bin NULL DEFAULT
NULL COMMENT '用户名称',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 49 CHARACTER SET = utf8 COLLATE =
utf8_bin ROW_FORMAT = Dynamic;

-- -----
-- Records of t_order
-- -----

SET FOREIGN_KEY_CHECKS = 1;

```

seata_points数据库

```

/*
Navicat Premium Data Transfer

Source Server        : localhost
Source Server Type   : MySQL
Source Server Version : 80020
Source Host          : localhost:3306
Source Schema        : seata_points

Target Server Type    : MySQL
Target Server Version : 80020
File Encoding         : 65001

Date: 02/12/2020 16:28:00
*/

SET NAMES utf8mb4;
SET FOREIGN_KEY_CHECKS = 0;

```

```

-- -----
-- Table structure for t_points
-- -----
DROP TABLE IF EXISTS `t_points`;
CREATE TABLE `t_points` (
  `id` int(0) NOT NULL AUTO_INCREMENT COMMENT '积分ID',
  `username` varchar(50) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL
COMMENT '用户名',
  `points` int(0) NULL DEFAULT NULL COMMENT '用户积分',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 3 CHARACTER SET = utf8 COLLATE =
utf8_bin ROW_FORMAT = Dynamic;

-- -----
-- Records of t_points
-- -----

SET FOREIGN_KEY_CHECKS = 1;

```

seata_storage数据库

```

/*
Navicat Premium Data Transfer

Source Server        : localhost
Source Server Type   : MySQL
Source Server Version : 80020
Source Host          : localhost:3306
Source Schema        : seata_storage

Target Server Type    : MySQL
Target Server Version : 80020
File Encoding         : 65001

Date: 02/12/2020 16:32:20
*/

SET NAMES utf8mb4;
SET FOREIGN_KEY_CHECKS = 0;

-- -----
-- Table structure for t_storage
-- -----
DROP TABLE IF EXISTS `t_storage`;
CREATE TABLE `t_storage` (
  `id` int(0) NOT NULL AUTO_INCREMENT COMMENT '库存ID',
  `goods_id` int(0) NULL DEFAULT NULL COMMENT '商品ID',

```

```

`storage` int(0) NULL DEFAULT NULL COMMENT '库存量',
PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 2 CHARACTER SET = utf8 COLLATE =
utf8_bin ROW_FORMAT = Dynamic;

-----
-- Records of t_storage
-----

INSERT INTO `t_storage` VALUES (1, 1, 100);

SET FOREIGN_KEY_CHECKS = 1;

```

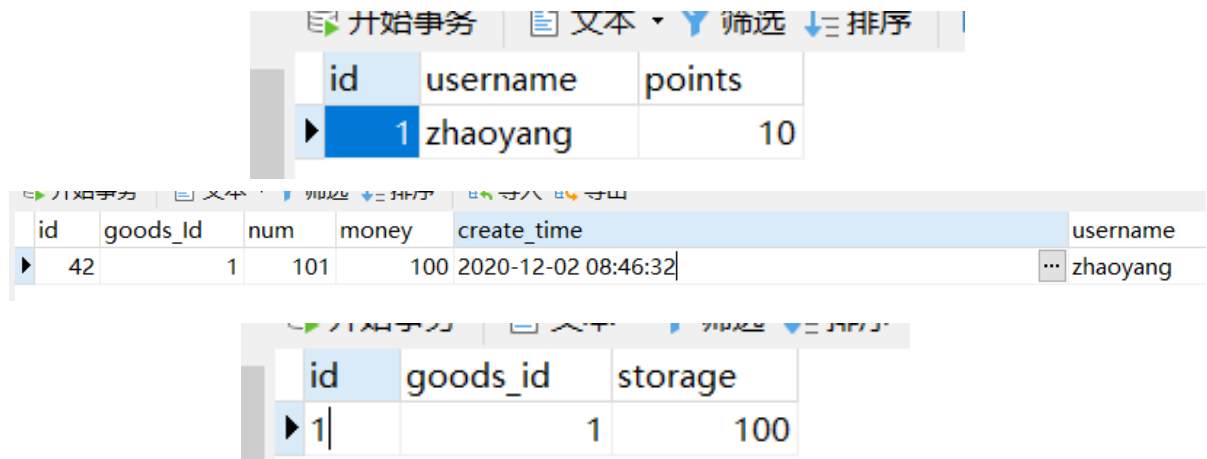
3. 案例测试

依次将4个服务启动.和nacos服务

访问路径为:

<http://localhost:8000/test1> 正常访问数据分别入库

<http://localhost:8000/test2> 访问出错库存不足,导致服务调用失败.则观察数据库, 经发现订单与积分数据库都已改变,而库存数据库没有减少库存, 所以不满足事务的特性.



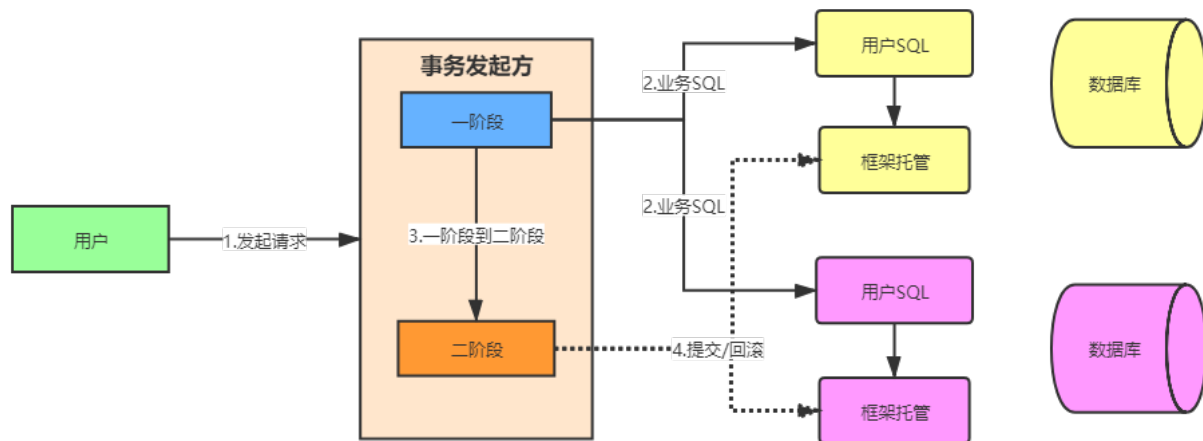
id	username	points
1	zhaoyang	10

id	goods_id	num	money	create_time	username
42	1	101	100	2020-12-02 08:46:32	zhaoyang

id	goods_id	storage
1	1	100

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。

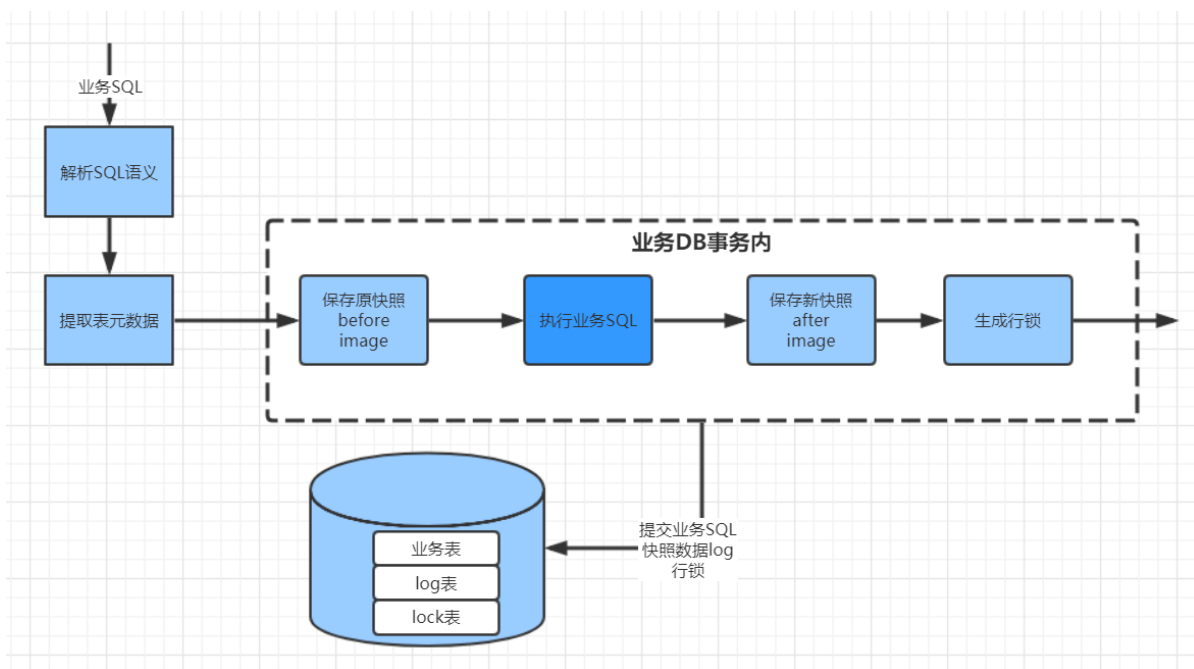


2.3 AT模式原理

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

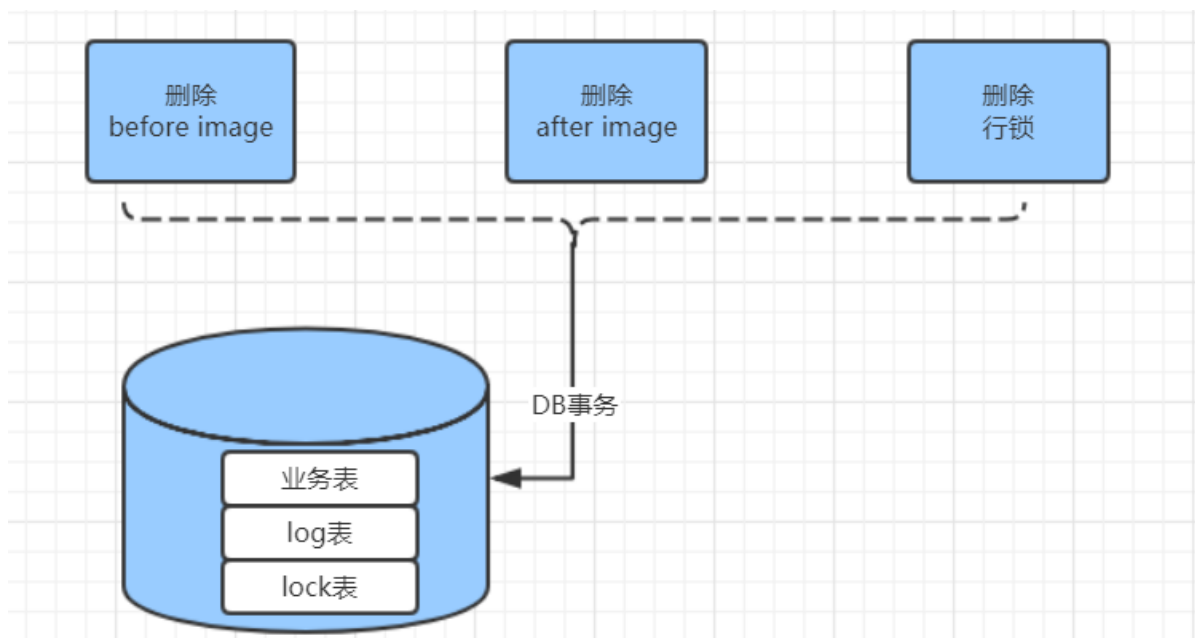
在一阶段, Seata 会拦截“业务 SQL”, 首先解析 SQL 语义, 找到“业务 SQL”要更新的业务数据, 在业务数据被更新前, 将其保存成“before image”, 然后执行“业务 SQL”更新业务数据, 在业务数据更新之后, 再将其保存成“after image”, 最后生成行锁。以上操作全部在一个数据库事务内完成, 这样保证了一阶段操作的原子性。



2. 二阶段

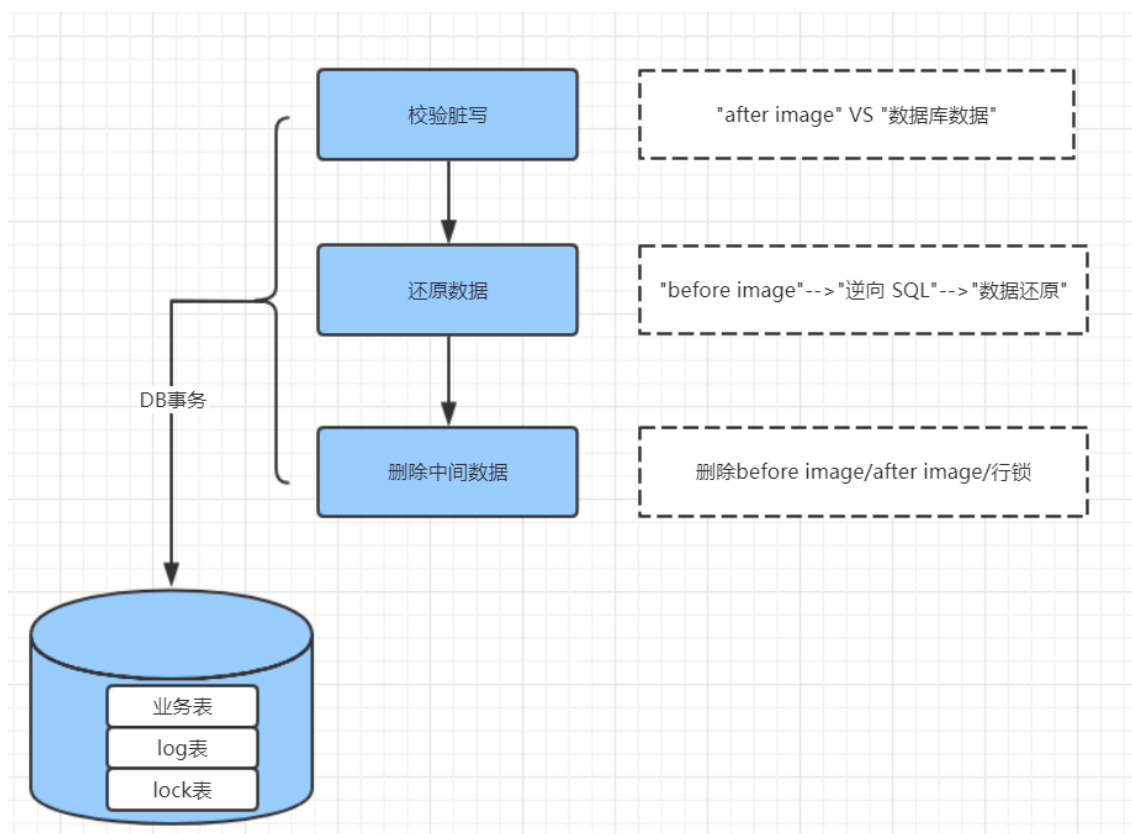
◦ 提交

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。



○ 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，对比“数据库当前业务数据”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案。

2.4 AT模式改造案例

2.4.1 Seata Server - TC全局事务协调器

介绍了 seata 事务的三个模块：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>

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>

develop seata / script / config-center / Go to file

Is9527 optimize: change client.log.exceptionRate to log.exceptionRate (#3247)	✓ 5e51d5e 9 days ago	History
..		
apollo	bugfix: fix configuration item containing spaces (#2390)	9 months ago
consul	bugfix: fix configuration item containing spaces (#2390)	9 months ago
etcd3	bugfix: fix configuration item containing spaces (#2390)	9 months ago
nacos	bugfix: nacos-config.py script could not run with namespace (#2932)	4 months ago
zk	bugfix: fix configuration item containing spaces (#2390)	9 months ago
README.md	bugfix: nacos-script adapt to nacos 1.2 on permission control. (#2610)	7 months ago
config.txt	optimize: change client.log.exceptionRate to log.exceptionRate (#3247)	9 days ago

<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
- 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本次课程采用db数据库模式

■ 存储模式

```
store.mode=db
```

■ mysql数据库连接信息


```

        `transaction_id` BIGINT,
        `status` TINYINT NOT NULL,
        `application_id` VARCHAR(32),
        `transaction_service_group` VARCHAR(32),
        `transaction_name` VARCHAR(128),
        `timeout` INT,
        `begin_time` BIGINT,
        `application_data` VARCHAR(2000),
        `gmt_create` DATETIME,
        `gmt_modified` DATETIME,
        PRIMARY KEY (`xid`),
        KEY `idx_gmt_modified_status` (`gmt_modified`, `status`),
        KEY `idx_transaction_id` (`transaction_id`)
    ) ENGINE = InnoDB
    DEFAULT CHARSET = utf8;

```

-- the table to store BranchSession data

```

CREATE TABLE IF NOT EXISTS `branch_table`
(
    `branch_id` BIGINT NOT NULL,
    `xid` VARCHAR(128) NOT NULL,
    `transaction_id` BIGINT,
    `resource_group_id` VARCHAR(32),
    `resource_id` VARCHAR(256),
    `branch_type` VARCHAR(8),
    `status` TINYINT,
    `client_id` VARCHAR(64),
    `application_data` VARCHAR(2000),
    `gmt_create` DATETIME(6),
    `gmt_modified` DATETIME(6),
    PRIMARY KEY (`branch_id`),
    KEY `idx_xid` (`xid`)
) ENGINE = InnoDB
    DEFAULT CHARSET = utf8;

```

-- the table to store lock data

```

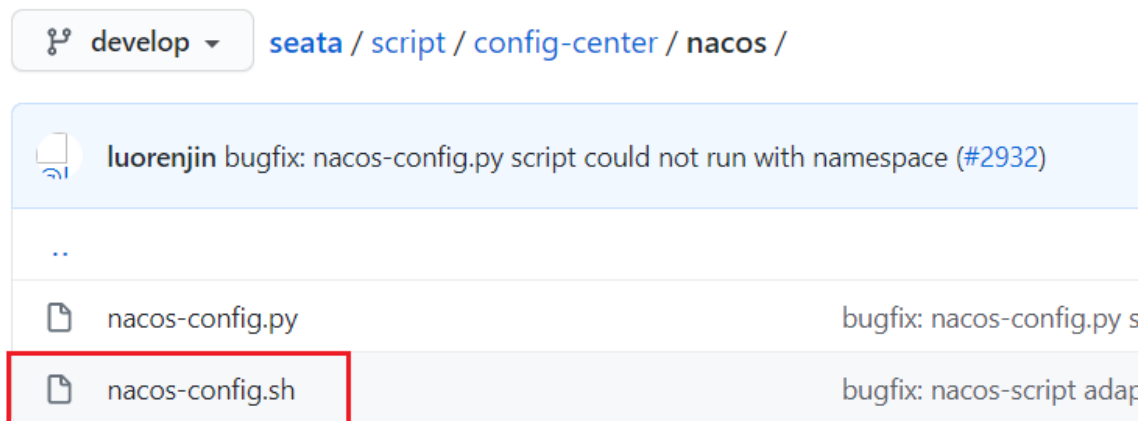
CREATE TABLE IF NOT EXISTS `lock_table`
(
    `row_key` VARCHAR(128) NOT NULL,
    `xid` VARCHAR(96),
    `transaction_id` BIGINT,
    `branch_id` BIGINT NOT NULL,
    `resource_id` VARCHAR(256),
    `table_name` VARCHAR(32),
    `pk` VARCHAR(36),
    `gmt_create` DATETIME,
    `gmt_modified` DATETIME,
    PRIMARY KEY (`row_key`),
    KEY `idx_branch_id` (`branch_id`)
)

```

```
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8;
...
```

- 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>

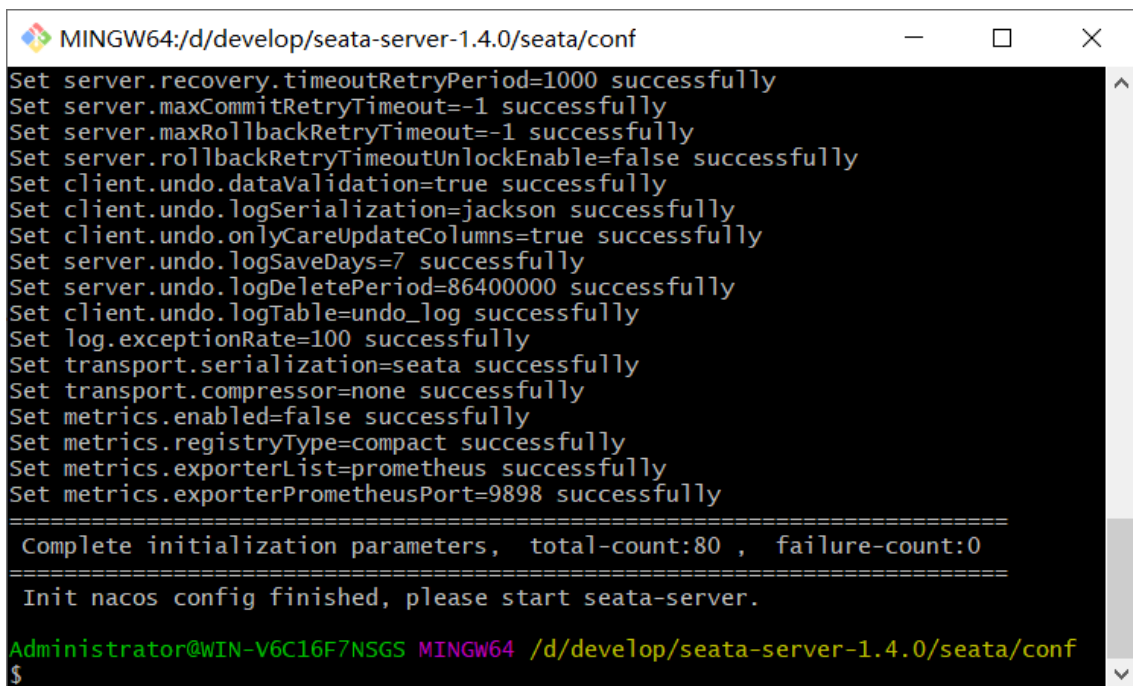
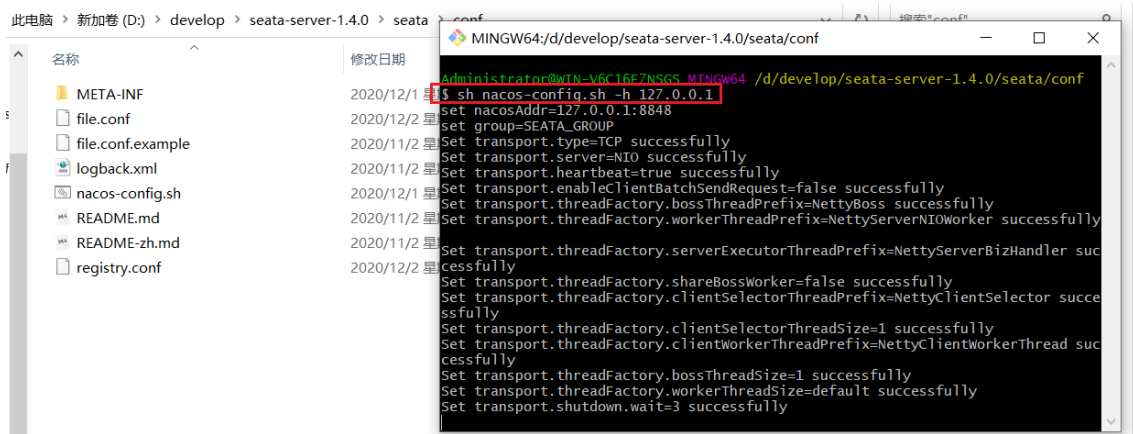


- 将nacos-config.sh放在seata/conf文件夹中

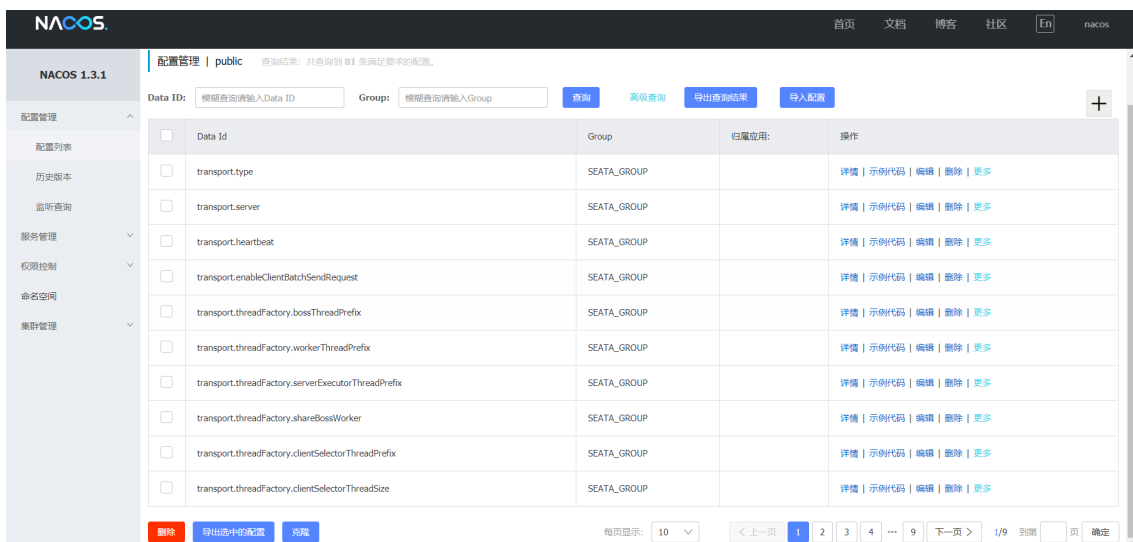


- 打开git bash here 执行nacos-config.sh,需要提前将nacos启动
输入命令：

```
sh nacos-config.sh -h 127.0.0.1
```

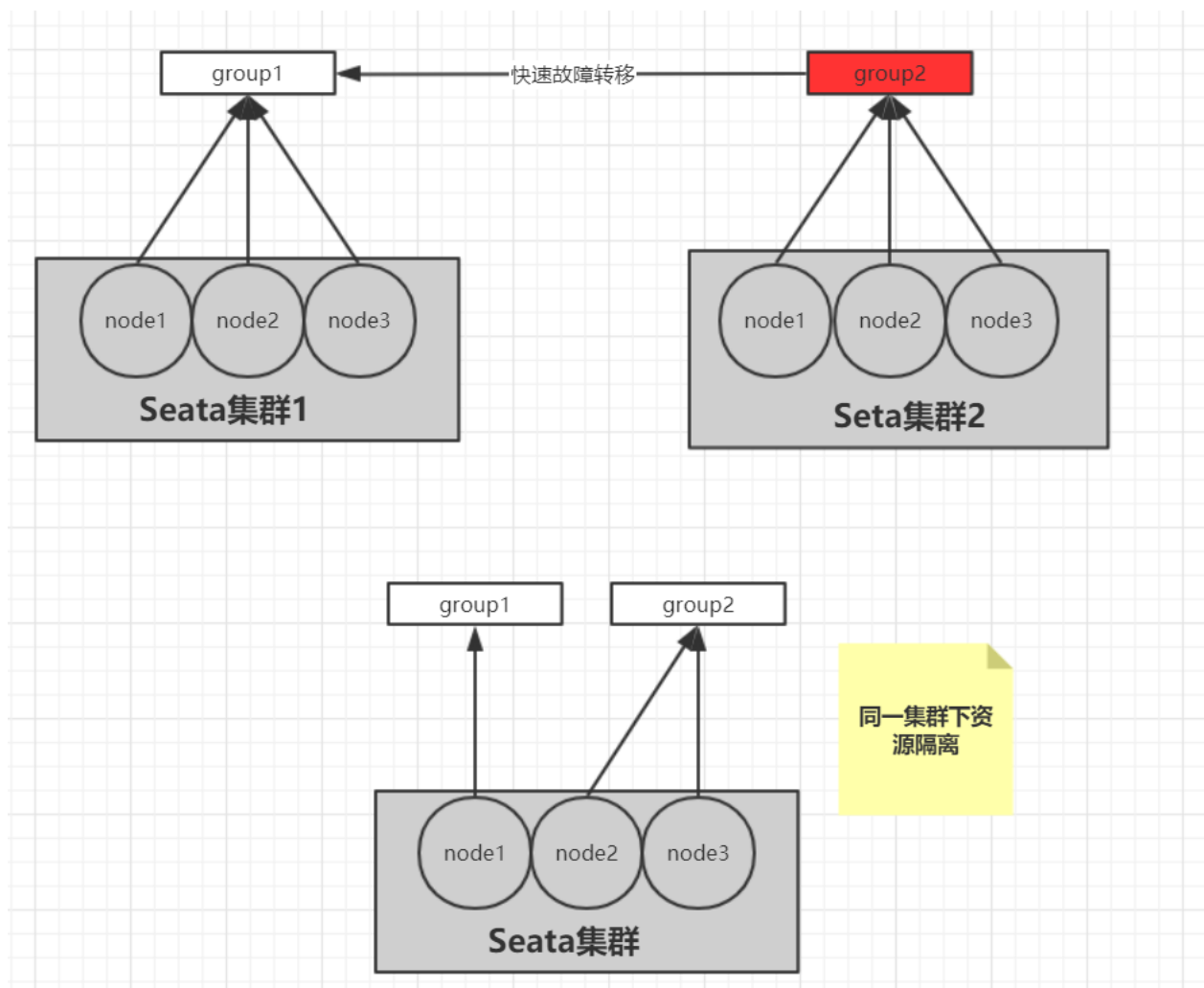


登录nacos查看配置信息



3. 启动seata-server

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lagou_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lagou_common_db完成前4步骤即可

1. 工程中添加Seata依赖

lagou_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>1.3.0</version>
        </dependency>
    </dependencies>
</dependencyManagement>

```

在lagou_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>1.3.0</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_test_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@Transactional

```

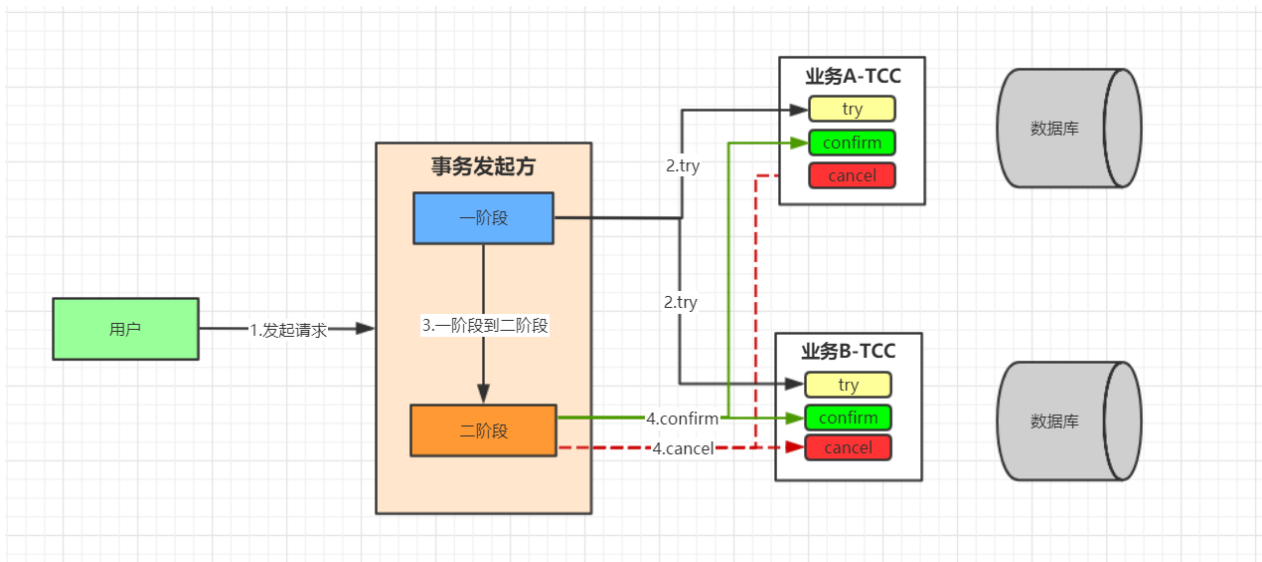
/**
 * 商品销售
 *
 * @param goodsId 商品id
 * @param num      销售数量
 * @param username 用户名
 * @param money     金额
 */
@Transactional(name = "sale", timeoutMills = 100000, rollbackFor =
Exception.class)
//@Transactional
public void sale(Integer goodsId, Integer num, Double money, String
username) {
    //创建订单
    orderServiceFeign.addOrder(idworker.nextId(),goodsId, num, money,
username);
    //增加积分
    pointsServiceFeign.increase(username, (int) (money / 10));
    //扣减库存
    storageServiceFeign.decrease(goodsId, num);
}

```

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。

➤ 一阶段（Try）：检查余额，预留其中 30 元；



➤ 二阶段提交（Confirm）：扣除 30 元；



➤ 二阶段回滚（Cancel）：释放预留的 30 元。



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。

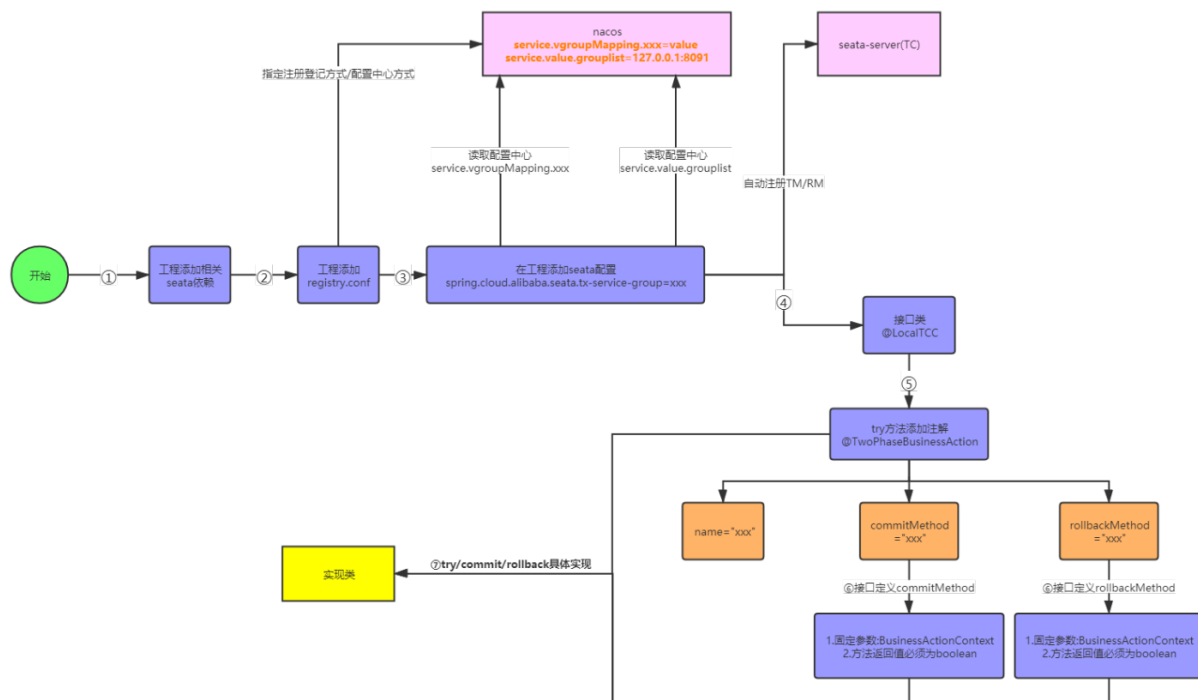
如果二阶段是回滚的话，就需要在 Cancel 方法内释放一阶段 Try 冻结的 30 元，使账号 A 的回到初始状态，100 元全部可用。

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_order`.`t_order` ADD COLUMN `status` int(0) NULL
COMMENT '订单状态-0不可用,事务未提交 , 1-可用,事务提交' ;

ALTER TABLE `seata_points`.`t_points` ADD COLUMN `frozen_points` int(0)
NULL DEFAULT 0 COMMENT '冻结积分' AFTER `points`;

ALTER TABLE `seata_storage`.`t_storage` ADD COLUMN `frozen_storage` int(0)
NULL DEFAULT 0 COMMENT '冻结库存' AFTER `goods_id`;
```

2. lagou_order工程改造

◦ 接口

```
package com.lagou.order.service;

import com.baomidou.mybatisplus.extension.service.IService;
```

```

import com.lagou.order.entity.Order;
import io.seata.rm.tcc.api.BusinessActionContext;
import io.seata.rm.tcc.api.BusinessActionContextParameter;
import io.seata.rm.tcc.api.LocalTCC;
import io.seata.rm.tcc.api.TwoPhaseBusinessAction;

/**
 * @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 OrderService extends IService<Order> {
    /**
     * @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
     * @BusinessActionContextParameter 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 = "addTcc",
        commitMethod = "addCommit", rollbackMethod =
"addRollBack")
    void add(@BusinessActionContextParameter(paramName = "order")
        Order order);

    public boolean addCommit(BusinessActionContext context);

    public boolean addRollBack(BusinessActionContext context);
}

```

○ 实现类

```

package com.lagou.order.service.impl;

import com.alibaba.fastjson.JSON;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.lagou.order.entity.Order;
import com.lagou.order.mapper.OrderMapper;
import com.lagou.order.service.OrderService;
import io.seata.core.context.RootContext;
import io.seata.rm.tcc.api.BusinessActionContext;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;

```



```

import java.util.Date;

@Slf4j
@Service
public class OrderServiceImpl extends ServiceImpl<OrderMapper, Order>
implements OrderService {

    @Override
    public void add(Order order) {
        order.setCreateTime(new Date()); //设置订单创建时间
        order.setStatus(0); //try阶段-预检查
        this.save(order); //保存订单
    }

    @Override
    public boolean addCommit(BusinessActionContext context) {
        Order order =
JSON.parseObject(context.getActionContext("order").toString(),
Order.class);
        order = this.getById(order.getId());
        if (order != null) {
            order.setStatus(1); //commit阶段-提交事务
            this.saveOrUpdate(order); //修改订单
        }
        log.info("----->xid=" + context.getXid() + " 提交成功!");
        return true;
    }

    @Override
    public boolean addRollBack(BusinessActionContext context) {
        Order order =
JSON.parseObject(context.getActionContext("order").toString(),
Order.class);
        order = this.getById(order.getId());
        if (order != null) {
            this.removeById(order.getId()); //删除订单
        }
        log.info("----->xid=" + context.getXid() + " 回滚成功!");
        return true;
    }
}

```

3. lagou_points工程改造

- 接口改造

```
package com.lagou.points.service;
```

```

import com.baomidou.mybatisplus.extension.service.IService;
import com.lagou.points.entity.Points;
import io.seata.rm.tcc.api.BusinessActionContext;
import io.seata.rm.tcc.api.BusinessActionContextParameter;
import io.seata.rm.tcc.api.LocalTCC;
import io.seata.rm.tcc.api.TwoPhaseBusinessAction;

@LocalTCC
public interface PointsService extends IService<Points> {

    @TwoPhaseBusinessAction(name = "increaseTcc", commitMethod =
        "increaseCommit"
        , rollbackMethod = "increaseRollback")
    public void increase(@BusinessActionContextParameter(paramName =
        "username")String username,
        @BusinessActionContextParameter(paramName =
        "points")Integer points);

    public boolean increaseCommit(BusinessActionContext context);

    public boolean increaseRollback(BusinessActionContext context);

}

```

- 实现类改造

```

package com.lagou.points.service.impl;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.lagou.points.mapper.PointsMapper;
import com.lagou.points.entity.Points;
import com.lagou.points.service.PointsService;
import io.seata.rm.tcc.api.BusinessActionContext;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

/**
 * 会员积分服务
 */
@Slf4j
@Service
public class PointsServiceImpl extends ServiceImpl<PointsMapper,
    Points> implements PointsService {

```

```

/**
 * 会员增加积分
 *
 * @param username 用户名
 * @param points 增加的积分
 * @return 积分对象
 */
public void increase(String username, Integer points) {
    QueryWrapper<Points> wrapper = new QueryWrapper<Points>();
    wrapper.lambda().eq(Points::getUsername, username);
    Points userPoints = this.getOne(wrapper);
    if (userPoints == null) {
        userPoints = new Points();
        userPoints.setUsername(username);
        //userPoints.setPoints(points); 不直接增加积分
        userPoints.setFrozenPoints(points); //设置冻结积分
        this.save(userPoints);
    } else {
        userPoints.setFrozenPoints(points); //设置冻结积分
        this.saveOrUpdate(userPoints);
    }
}

@Override
public boolean increaseCommit(BusinessActionContext context) {
    //查询用户积分
    QueryWrapper<Points> wrapper = new QueryWrapper<Points>();
    wrapper.lambda().eq(Points::getUsername,
context.getActionContext("username"));
    Points userPoints = this.getOne(wrapper);
    if (userPoints != null) {
        //增加用户积分
        userPoints.setPoints(userPoints.getPoints() +
userPoints.getFrozenPoints());
        //冻结积分清零
        userPoints.setFrozenPoints(0);
        this.saveOrUpdate(userPoints);
    }
    log.info("----->xid=" + context.getXid() + " 提交成功!");
    return true;
}

@Override
public boolean increaseRollback(BusinessActionContext context) {
    //查询用户积分
    QueryWrapper<Points> wrapper = new QueryWrapper<Points>();
    wrapper.lambda().eq(Points::getUsername,
context.getActionContext("username"));
    Points userPoints = this.getOne(wrapper);

```

```

        if (userPoints != null) {
            //冻结积分清零
            userPoints.setFrozenPoints(0);
            this.saveOrUpdate(userPoints);
        }
        log.info("----->xid=" + context.getXid() + " 回滚成功!");
        return true;
    }
}

```

4. lagou_stroage工程改造

◦ 接口改造

```

package com.lagou.storage.service;

import com.baomidou.mybatisplus.extension.service.IService;
import com.lagou.storage.entity.Storage;
import io.seata.rm.tcc.api.BusinessActionContext;
import io.seata.rm.tcc.api.BusinessActionContextParameter;
import io.seata.rm.tcc.api.LocalTCC;
import io.seata.rm.tcc.api.TwoPhaseBusinessAction;

/**
 * 仓库服务
 */
@LocalTCC
public interface StorageService extends IService<Storage> {
    @TwoPhaseBusinessAction(name = "decreaseTcc", commitMethod =
        "decreaseCommit"
        , rollbackMethod = "decreaseRollback")
    public void decrease(@BusinessActionContextParameter(paramName =
        "goodsId")Integer goodsId,
        @BusinessActionContextParameter(paramName =
        "quantity")Integer quantity);

    public boolean decreaseCommit(BusinessActionContext context);

    public boolean decreaseRollback(BusinessActionContext context);
}

```

◦ 实现类改造

```

package com.lagou.storage.service.impl;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.lagou.storage.entity.Storage;

```

```

import com.lagou.storage.mapper.StorageMapper;
import com.lagou.storage.service.StorageService;
import io.seata.rm.tcc.api.BusinessActionContext;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;

/**
 * 仓库服务
 */
@Slf4j
@Service
public class StorageServiceImpl extends ServiceImpl<StorageMapper,
Storage> implements StorageService {

    /**
     * 减少库存
     *
     * @param goodsId 商品ID
     * @param quantity 减少数量
     * @return 库存对象
     */
    public void decrease(Integer goodsId, Integer quantity) {
        QueryWrapper<Storage> wrapper = new QueryWrapper<Storage>();
        wrapper.lambda().eq(Storage::getGoodsId, goodsId);
        Storage goodsStorage = this.getOne(wrapper);
        if (goodsStorage.getStorage() >= quantity) {
            //goodsStorage.setStorage(goodsStorage.getStorage() -
quantity);
            //设置冻结库存
            goodsStorage.setFrozenStorage(quantity);
        } else {
            throw new RuntimeException(goodsId + "库存不足,目前剩余库存:"
+ goodsStorage.getStorage());
        }
        this.saveOrUpdate(goodsStorage);
    }

    @Override
    public boolean decreaseCommit(BusinessActionContext context) {
        QueryWrapper<Storage> wrapper = new QueryWrapper<Storage>();
        wrapper.lambda().eq(Storage::getGoodsId,
context.getActionContext("goodsId"));
        Storage goodsStorage = this.getOne(wrapper);
        if (goodsStorage != null) {
            //扣减库存
            goodsStorage.setStorage(goodsStorage.getStorage() -
goodsStorage.getFrozenStorage());

```

```

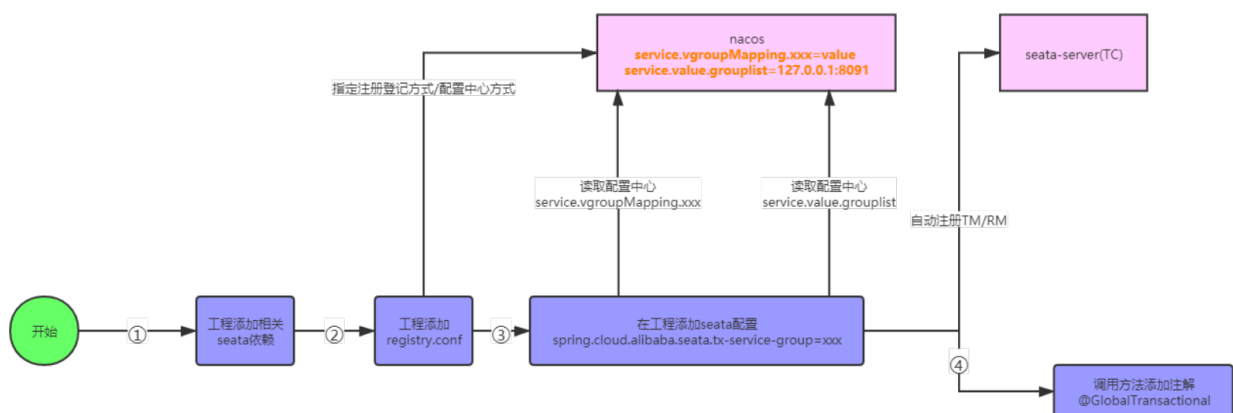
        //冻结库存清零
        goodsStorage.setFrozenStorage(0);
        this.saveOrUpdate(goodsStorage);
    }
    log.info("----->xid=" + context.getXid() + " 提交成功!");
    return true;
}

@Override
public boolean decreaseRollback(BusinessActionContext context) {
    QueryWrapper<Storage> wrapper = new QueryWrapper<Storage>();
    wrapper.lambda().eq(Storage::getGoodsId,
context.getActionContext("goodsId"));
    Storage goodsStorage = this.getOne(wrapper);
    if (goodsStorage != null) {
        //冻结库存清零
        goodsStorage.setFrozenStorage(0);
        this.saveOrUpdate(goodsStorage);
    }
    log.info("----->xid=" + context.getXid() + " 回滚成功!");
    return true;
}
}

```

3.2.2 TM端改造

针对我们工程lagou_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lagou_common_db完成即可,因为lagou_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lagou_bussiness也对数据库操作了.也需要完成try/commit/rollback的实现.



代码实现:

```

/**
 * 商品销售

```

```

*
* @param goodsId 商品id
* @param num      销售数量
* @param username 用户名
* @param money    金额
*/
@GlobalTransactional(name = "sale", timeoutMills = 100000, rollbackFor =
Exception.class)
//@Transactional
public void sale(Integer goodsId, Integer num, Double money, String username)
{
    //创建订单
    orderServiceFeign.addOrder(idworker.nextId(), goodsId, num, money,
username);
    //增加积分
    pointsServiceFeign.increase(username, (int) (money / 10));
    //扣减库存
    storageServiceFeign.decrease(goodsId, num);
}

```

4.Seata-Saga模式简介与三种模式对比

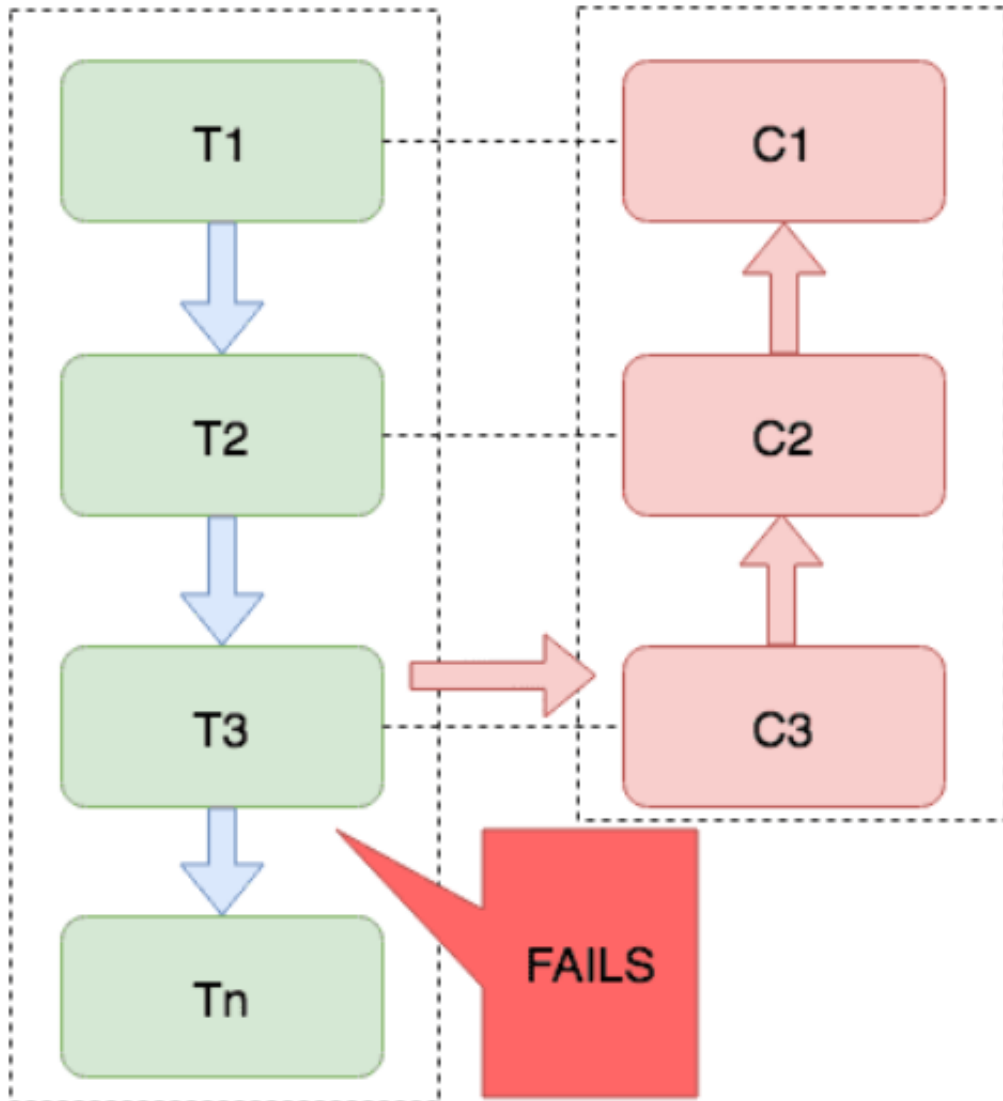
4.1 Saga模式简单介绍

Saga 模式是 Seata 开源的长事务解决方案，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。

Normal transactions

Compensating transactions



适用场景：

- 业务流程长、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

4.2 三种模式对比

	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	长流程 涉及大量第三方调用

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 模式是长事务解决方案，适用于业务流程长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，长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 模式。

5.Seata源码

5.1 Seata 源码搭建

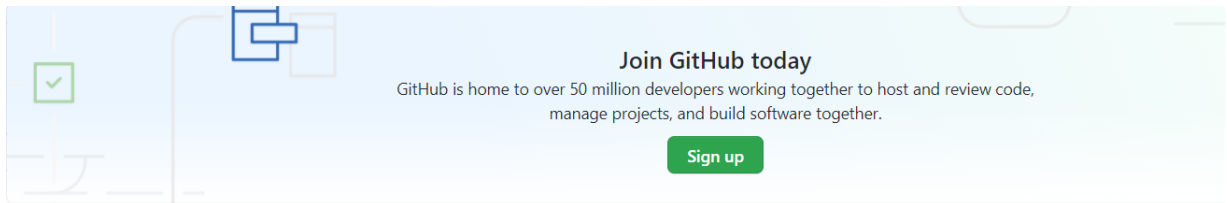
5.1.1 环境准备

- JDK 1.8
- Maven 3.2+

5.1.2 源码构建

从github上，下载seata的源码到本地,版本选择1.3.0

```
https://github.com/seata/seata/tree/1.3.0
```



1.3.0

11 branches

29 tags

Go to file

Code

This branch is 95 commits behind develop.

Pull request

Compare

slievrlly release: release 1.3.0 (#2884)

2cc875b on 15 Jul

1,029 commits

.github	Change groupId, artifactId, and package in 0.5.0 (#809)	2 years ago
.mvn/wrapper	Use maven-compiler-plugin to revision the version and add mvnw script...	2 years ago
all	release: release 1.3.0 (#2884)	5 months ago
bom	release: release 1.3.0 (#2884)	5 months ago
common	optimize: code opt format (#2799)	6 months ago
compressor	feature: add LZ4 compressor (#2510)	7 months ago
config	optimize: code opt format (#2799)	6 months ago
core	release: release 1.3.0 (#2884)	5 months ago

About

Seata is an e performance, o transaction solu

seata.io

fescar transac

microservice

data-consistency

tcc-transaction

distributed-transa

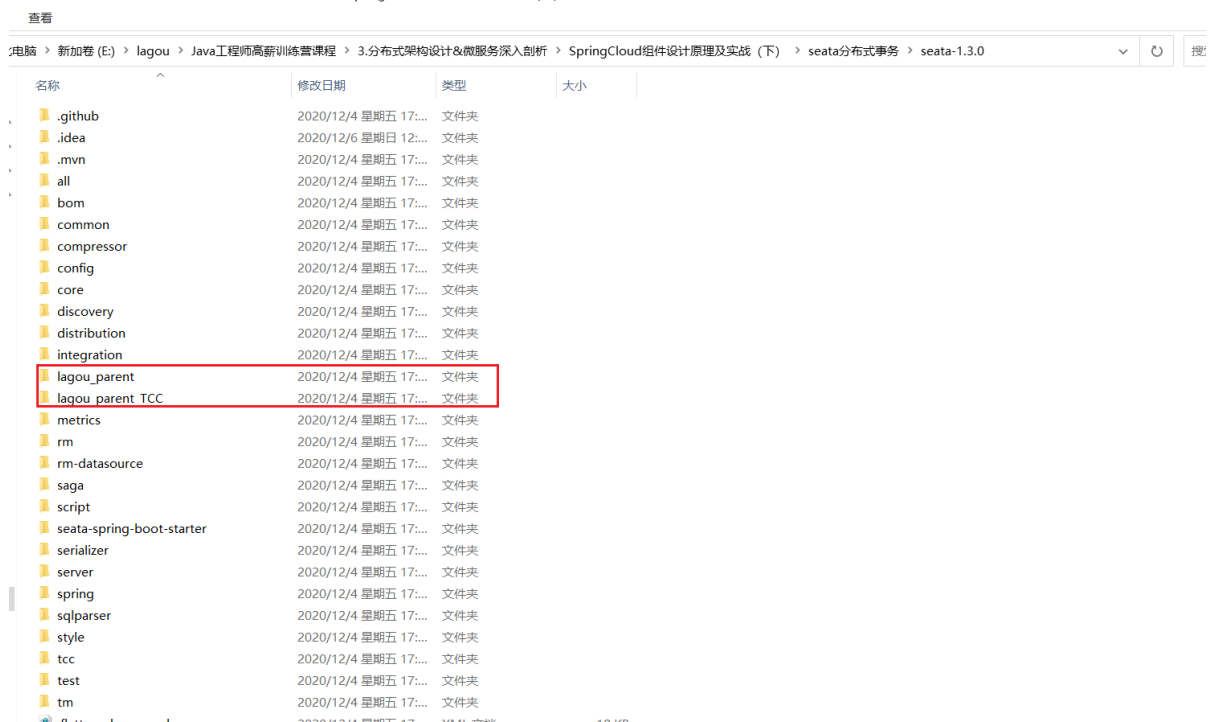
Readme

Apache-2.0 L

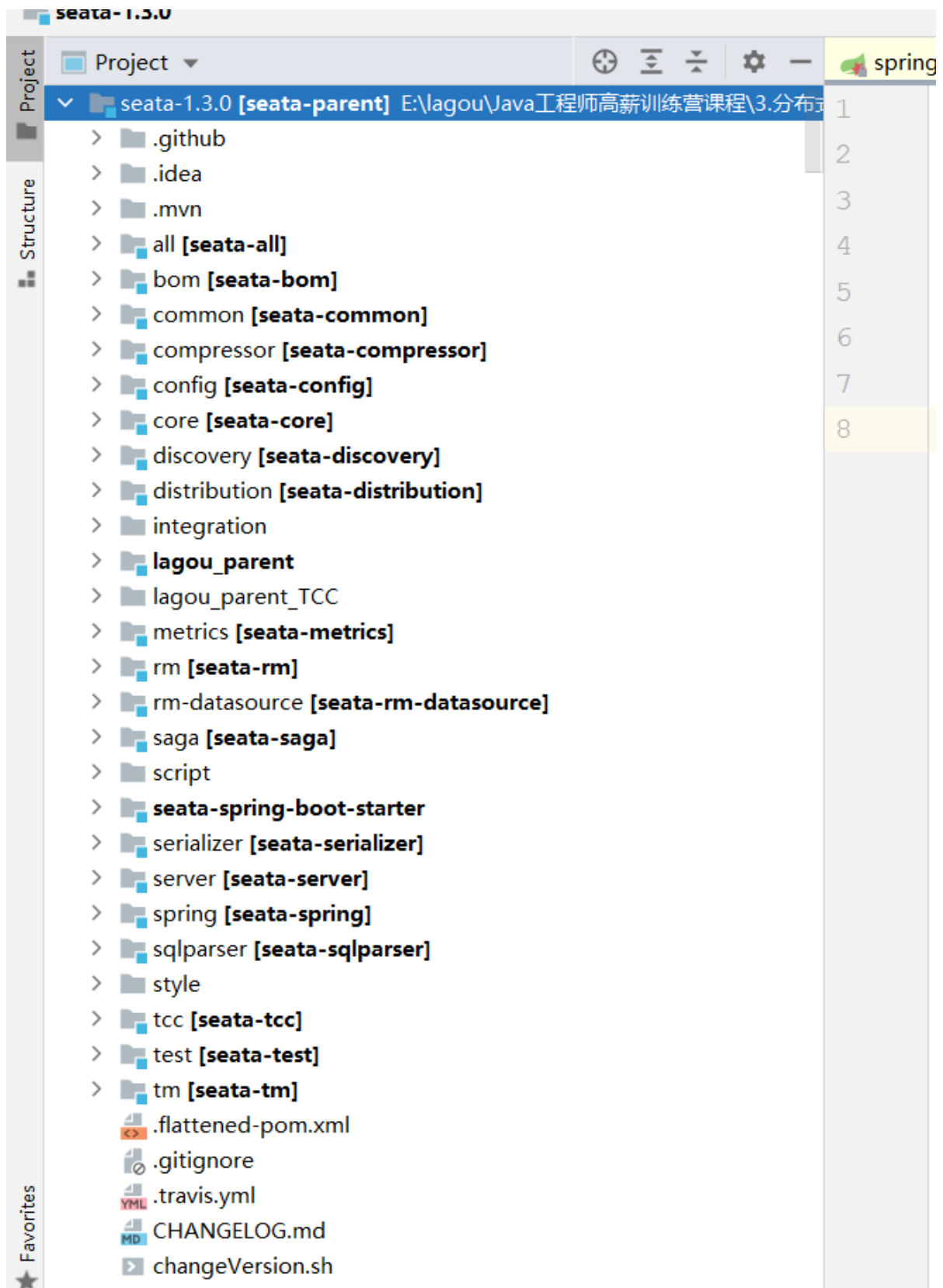
Releases 29

5.1.3 导入idea工程

1. 将lagou-parent工程放入seata源码中导入idea



2. idea工程目录



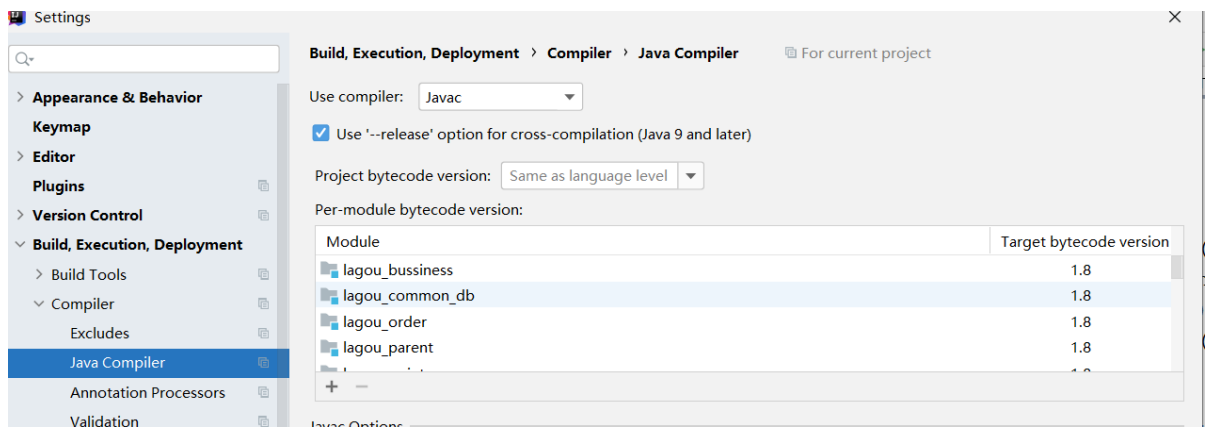
3. 设置jdk版本为1.8

Project name:
seata-parent

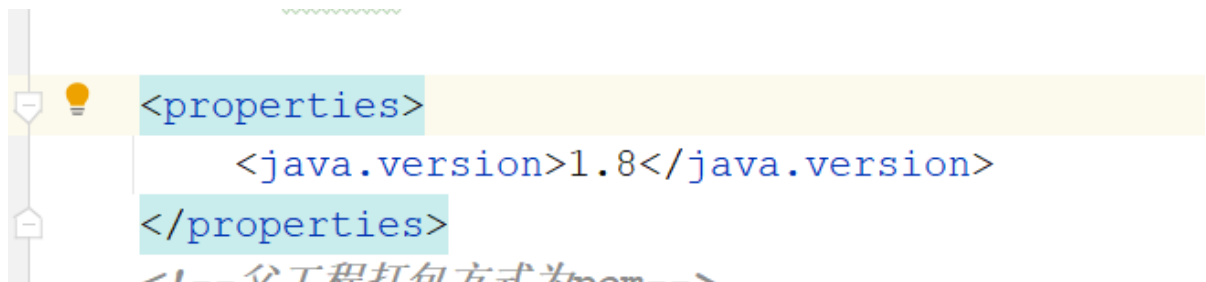
Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.
1.8 version 1.8.0_172 Edit

Project language level:
This language level is default for all project modules.
A module specific language level can be configured for each of the modules as required.
8 - Lambdas, type annotations etc.

Project compiler output:
This path is used to store all project compilation results.
A directory corresponding to each module is created under this path.
This directory contains two subdirectories: Production and Test for production code and test sources, respectively.
A module specific compiler output path can be configured for each of the modules as required.



lagou_parent工程修改工程版本为1.8



4. registry.conf 文件替换之前已搭建好的nacos注册中心和配置中心的文件

```
registry {
    # file 、 nacos 、 eureka、 redis、 zk、 consul、 etcd3、 sofa
    type = "nacos"
    loadBalance = "RandomLoadBalance"
    loadBalanceVirtualNodes = 10

    nacos {
        application = "seata-server"
        serverAddr = "127.0.0.1:8848"
        group = "SEATA_GROUP"
    }
}
```

```

    namespace = ""
    cluster = "default"
    username = "nacos"
    password = "nacos"
}
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 {
    # 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"
}

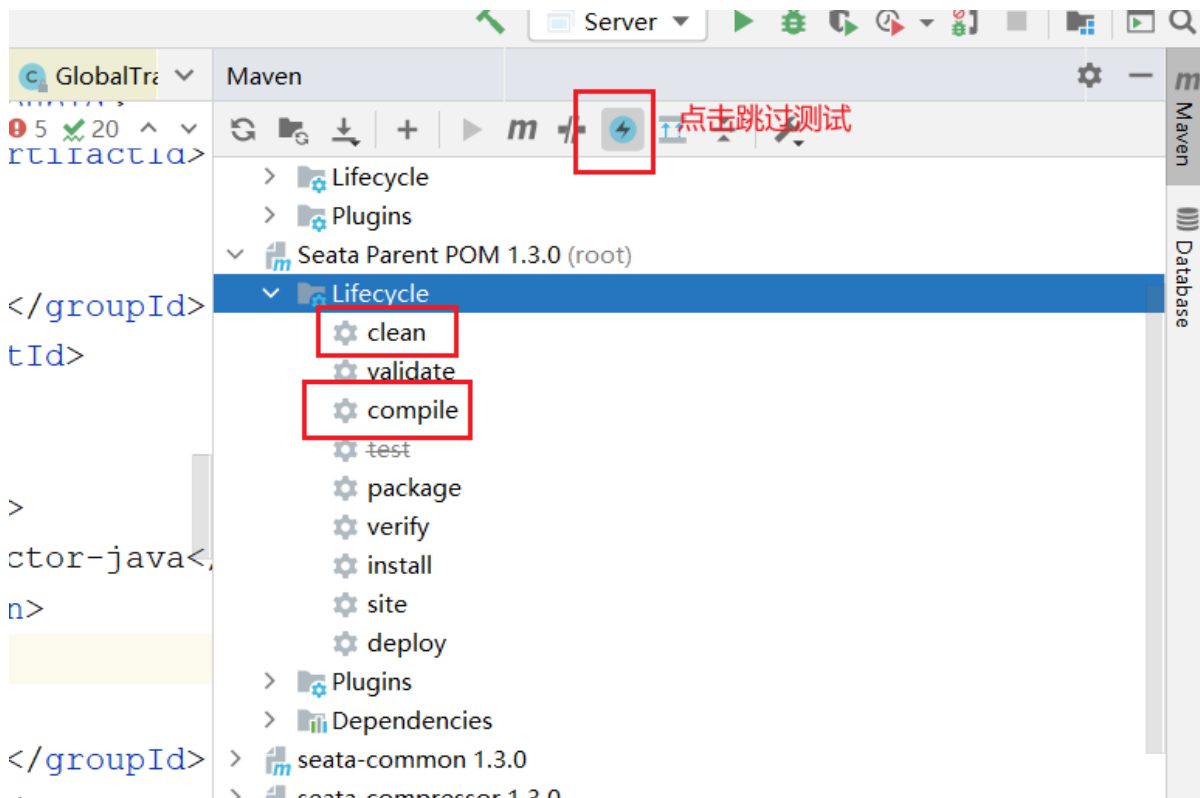
apollo {
    appId = "seata-server"
    apolloMeta = "http://192.168.1.204:8801"
    namespace = "application"
    apolloAccesskeySecret = ""
}

zk {
    serverAddr = "127.0.0.1:2181"
    sessionTimeout = 6000
    connectTimeout = 2000
    username = ""
    password = ""
}

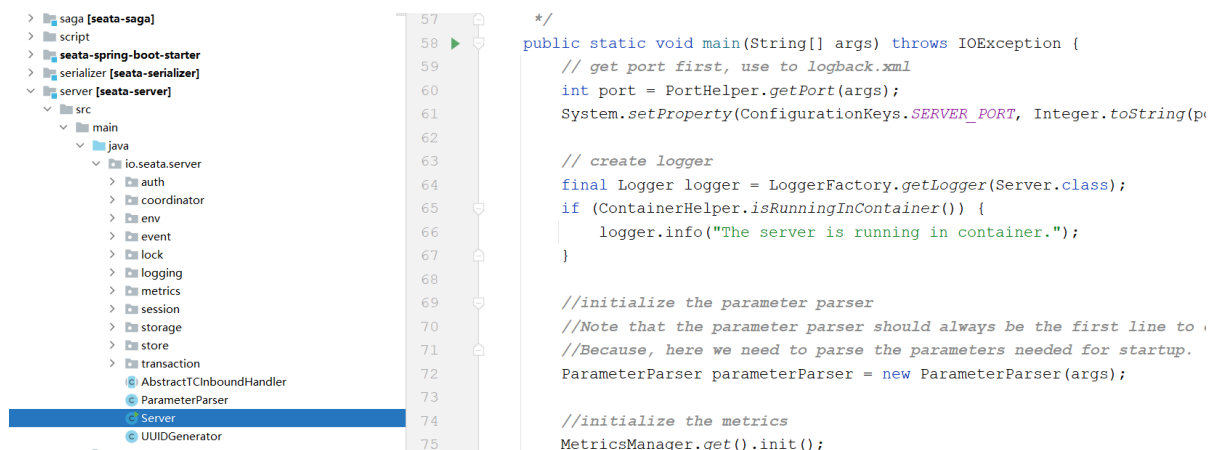
etcd3 {
    serverAddr = "http://localhost:2379"
}

file {
    name = "file.conf"
}
}
```

5. 编译工程clean-->compile



6. 工程启动seata-server 工程里面Server类的main方法

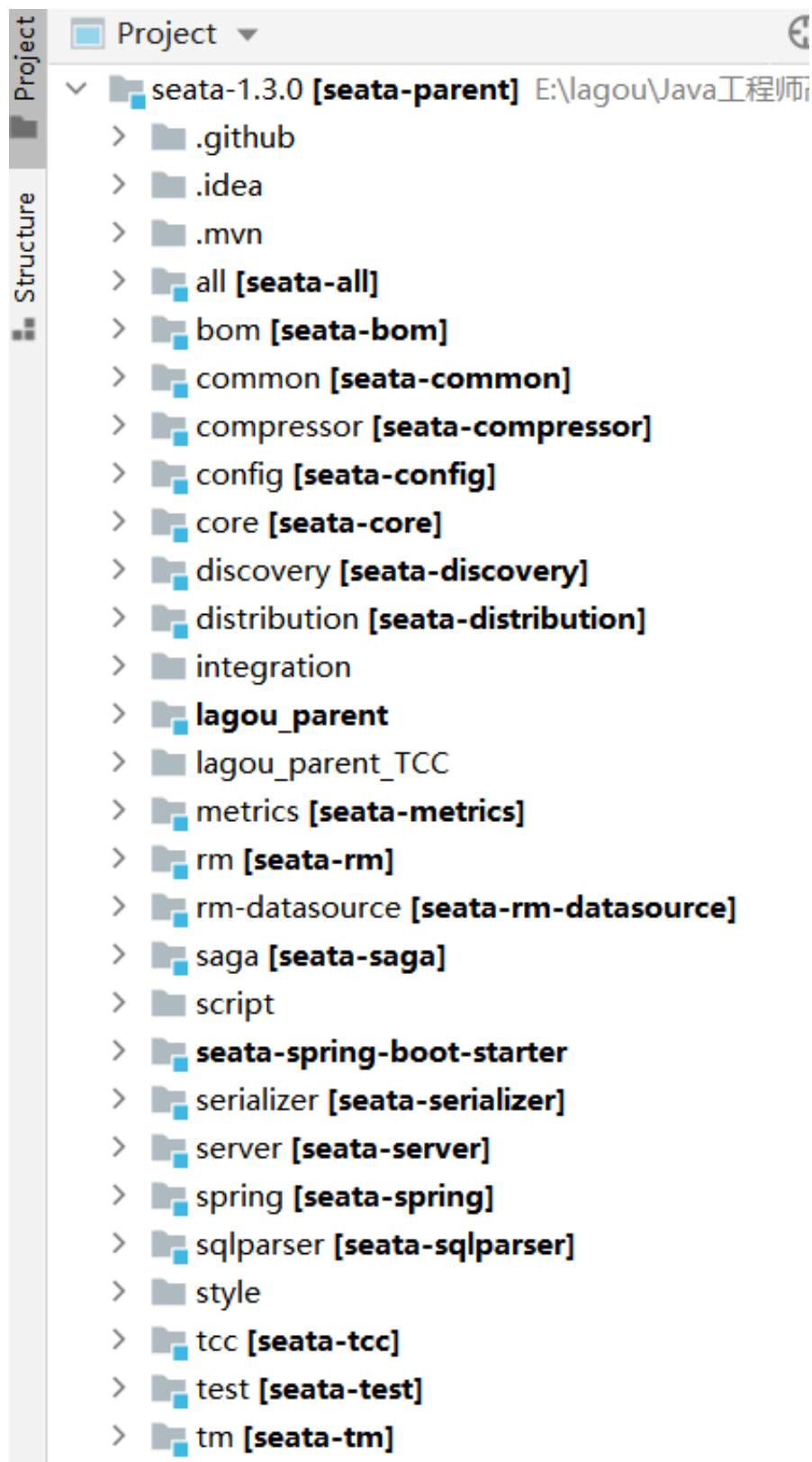


7. 查看nacos是否注册上



5.1.4 seata工程结构

先看下整体seata 的工程结构



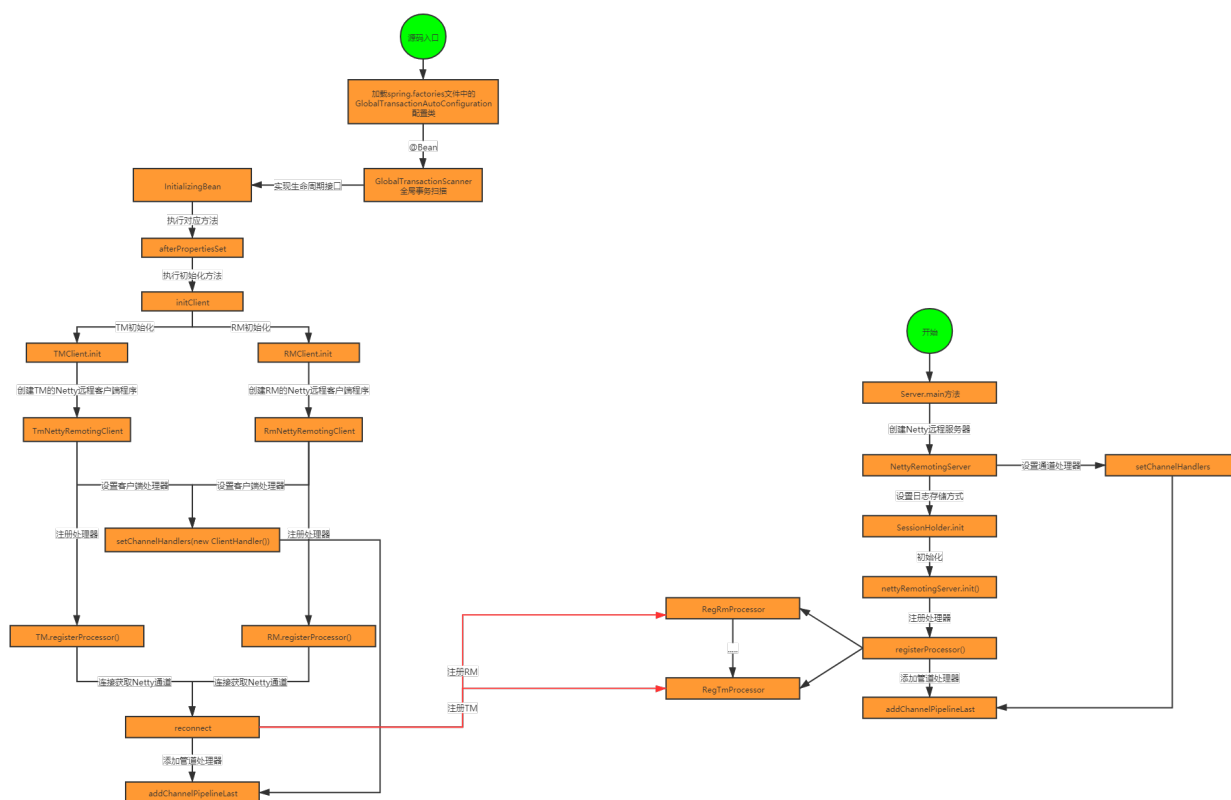
- **seata-common 模块:** seata-common 项目，提供 Seata 封装的工具类、异常类等
- **seata-core 模块:** seata-core 项目，提供 Seata 封装的 RPC、数据模型、通信消息格式等
- **seata-config 模块:** 从配置中心读取配置。
- **seata-discovery 模块:** 用于 Seata TC 注册到注册中心。用于 Seata TM 从注册中心发现 Seata TC。
- **seata-rm 模块:** seata-rm 项目，Seata 对 RM 的核心实现
- **seata-rm-datasource 模块:** seata-rm-datasource 项目，Seata 通过对 JDBC 拓展，从而实现对 MySQL 等的透明接入 Seata RM 的实现

- **seata-server 模块**:seata-server 项目, Seata 对 TC 的核心实现, 提供了事务协调、锁、事务状态、事务会话等功能
- **seata-tm 模块**:seata-tm 项目, Seata 对 TM 的实现, 提供了全局事务管理, 例如说事务的发起, 提交, 回滚等
- **seata-tcc 模块**:seata-tcc 项目, Seata 对 TCC 事务模式的实现
- **seata-spring 模块**: seata-spring 项目, Spring 对 Seata 集成的实现。例如说, 使用 @GlobalTransactional 注解, 自动创建全局事务, 就是通过 seata-spring 项目来实现的。

5.2 AT源码

5.2.1 TM / RM 初始化与服务注册TC

5.2.1.1 TM / RM 初始化与服务注册TC流程分析



5.2.2.1 主要源码跟踪

1. 导入seata的起步依赖后,会加载spring-cloud-alibaba-seata-2.1.0.RELEASE.jar下面的spring.factories文件,在文件中加载GlobalTransactionAutoConfiguration类,针对seata的自动配置类

```

spring.factories
AbstractNettyRemotingServer.java
NettyServerBootstrap.java
AbstractNettyRemoting.java
Defi
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.alibaba.cloud.seata.rest.SeataRestTemplateAutoConfiguration,\
com.alibaba.cloud.seata.web.SeataHandlerInterceptorConfiguration,\
com.alibaba.cloud.seata.GlobalTransactionAutoConfiguration,\
com.alibaba.cloud.seata.feign.SeataFeignClientAutoConfiguration,\
com.alibaba.cloud.seata.feign.hystrix.SeataHystrixAutoConfiguration
Reader Mode

```

2. 声明GlobalTransactionScanner全局事务扫描Bean

```

@Bean
public GlobalTransactionScanner globalTransactionScanner() {

    String applicationName = applicationContext.getEnvironment()
        .getProperty("spring.application.name");
    //获取事务分组名称
    String txServiceGroup = seataProperties.getTxServiceGroup();

    if (StringUtils.isEmpty(txServiceGroup)) {
        //如果没有配置事务分组默认为应用名称+"-fescar-service-group"
        txServiceGroup = applicationName + "-fescar-service-group";
        seataProperties.setTxServiceGroup(txServiceGroup);
    }
    // 创建全局事务扫描
    return new GlobalTransactionScanner(applicationName, txServiceGroup);
}

```

3. GlobalTransactionScanner实现了spring bean生命周期接口InitializingBean,会执行afterPropertiesSet方法

```

@Override
public void afterPropertiesSet() {
    if (disableGlobalTransaction) {
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info("Global transaction is disabled.");
        }
        return;
    }
    //初始化客户端
    initClient();
}

```

4. initClient方法

```

private void initClient() {
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Initializing Global Transaction Clients ... ");
    }
    if (StringUtils.isNullOrEmpty(applicationId) ||
        StringUtils.isNullOrEmpty(txServiceGroup)) {
        throw new IllegalArgumentException(String.format("applicationId: %s, txServiceGroup: %s", applicationId, txServiceGroup));
    }
    //初始化 TM
    TMClient.init(applicationId, txServiceGroup);
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Transaction Manager Client is initialized. applicationId[{}] txServiceGroup[{}]", applicationId, txServiceGroup);
    }
}

```

```

    }
    //初始化 RM
    RMClient.init(applicationId, txServiceGroup);
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Resource Manager is initialized. applicationId[{}] txServiceGroup[{}]", applicationId, txServiceGroup);
    }

    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Global Transaction Clients are initialized. ");
    }
    registersSpringShutdownHook();
}

```

5. TMClient.init方法

```

public static void init(String applicationId, String
transactionServiceGroup) {
    //创建TM的Netty远程客户端程序
    TmNettyRemotingClient tmNettyRemotingClient =
    TmNettyRemotingClient.getInstance(applicationId, transactionServiceGroup);
    tmNettyRemotingClient.init();
}

```

getInstance方法获取实例

```

public static TmNettyRemotingClient getInstance() {
    if (instance == null) {
        synchronized (TmNettyRemotingClient.class) {
            if (instance == null) {
                NettyClientConfig nettyClientConfig = new
                NettyClientConfig();
                final ThreadPoolExecutor messageExecutor = new
                ThreadPoolExecutor(
                    nettyClientConfig.getClientWorkerThreads(),
                    nettyClientConfig.getClientWorkerThreads(),
                    KEEP_ALIVE_TIME, TimeUnit.SECONDS,
                    new LinkedBlockingQueue<>(MAX_QUEUE_SIZE),
                    new
                    NamedThreadFactory(nettyClientConfig.getTmDispatchThreadPrefix(),
                        nettyClientConfig.getClientWorkerThreads()),
                    RejectedPolicies.runsOldestTaskPolicy());
                //创建TM Netty远程客户端
                instance = new TmNettyRemotingClient(nettyClientConfig,
                null, messageExecutor);
            }
        }
    }
}

```

```

    }
    return instance;
}

```

TmNettyRemotingClient构造方法

```

public AbstractNettyRemotingClient(NettyClientConfig nettyClientConfig,
    EventExecutorGroup eventExecutorGroup,
    ThreadPoolExecutor messageExecutor,
    NettyPoolKey.TransactionRole transactionRole) {
    super(messageExecutor);
    this.transactionRole = transactionRole;
    //创建Netty客户端引导类
    clientBootstrap = new NettyClientBootstrap(nettyClientConfig,
    eventExecutorGroup, transactionRole);
    //设置通道处理程序
    clientBootstrap.setChannelHandlers(new ClientHandler());
    //创建Netty客户端通道管理器
    clientChannelManager = new NettyClientChannelManager(
        new NettyPoolableFactory(this, clientBootstrap),
        getPoolKeyFunction(), nettyClientConfig);
}

```

ClientHandler客户端处理类

```

class ClientHandler extends ChannelDuplexHandler {
    /**
     * 读取通道消息
     * @param ctx
     * @param msg
     * @throws Exception
     */
    @Override
    public void channelRead(final ChannelHandlerContext ctx, Object msg)
    throws Exception {
        if (!(msg instanceof RpcMessage)) {
            return;
        }
        //处理消息
        processMessage(ctx, (RpcMessage) msg);
    }

    .....
}

```

6. init()方法

```

@Override

```

```

public void init() {
    //创建一个延时线程,在60秒后执行
    timerExecutor.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            //连接netty服务端
            clientChannelManager.reconnect(getTransactionServiceGroup());
        }
    }, SCHEDULE_DELAY_MILLS, SCHEDULE_INTERVAL_MILLS,
    TimeUnit.MILLISECONDS);
    if (NettyClientConfig.isEnableClientBatchSendRequest()) {
        mergeSendExecutorService = new
        ThreadPoolExecutor(MAX_MERGE_SEND_THREAD,
            MAX_MERGE_SEND_THREAD,
            KEEP_ALIVE_TIME, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<>(),
            new NamedThreadFactory(getThreadPrefix(),
            MAX_MERGE_SEND_THREAD));
        mergeSendExecutorService.submit(new MergedSendRunnable());
    }
    super.init();
    //启动客户端引导类
    clientBootstrap.start();
}

```

```

@Override
public void start() {
    if (this.defaultEventExecutorGroup == null) {
        this.defaultEventExecutorGroup = new
        DefaultEventExecutorGroup(nettyClientConfig.getClientWorkerThreads(),
            new
            NamedThreadFactory(getThreadPrefix(nettyClientConfig.getClientWorkerThread
            Prefix()),
                nettyClientConfig.getClientWorkerThreads()));
    }
    this.bootstrap.group(this.eventLoopGroupworker).channel(
        nettyClientConfig.getClientChannelClazz()).option(
        ChannelOption.TCP_NODELAY,
        true).option(ChannelOption.SO_KEEPALIVE, true).option(
        ChannelOption.CONNECT_TIMEOUT_MILLIS,
        nettyClientConfig.getConnectTimeoutMillis()).option(
        ChannelOption.SO_SNDBUF,
        nettyClientConfig.getClientSocketsSndBufSize()).option(ChannelOption.SO_RCV
        BUF,
        nettyClientConfig.getClientSocketRcvBufSize());

    if (nettyClientConfig.enableNative()) {
        if (PlatformDependent.isOsx()) {
            if (LOGGER.isInfoEnabled()) {

```

```

        LOGGER.info("client run on macOS");
    }
} else {
    bootstrap.option(EpollChannelOption.EPOLL_MODE,
        EpollMode.EDGE_TRIGGERED)
        .option(EpollChannelOption.TCP_QUICKACK, true);
}

bootstrap.handler(
    new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch) {
            ChannelPipeline pipeline = ch.pipeline();
            pipeline.addLast(
                new
                IdleStateHandler(nettyClientConfig.getChannelMaxReadIdleSeconds(),
                    nettyClientConfig.getChannelMaxWriteIdleSeconds(),
                    nettyClientConfig.getChannelMaxAllIdleSeconds()))
                .addLast(new ProtocolV1Decoder())
                .addLast(new ProtocolV1Encoder());
            if (channelHandlers != null) {
                //添加管道处理器
                addChannelPipelineLast(ch, channelHandlers);
            }
        }
    });

    if (initialized.compareAndSet(false, true) && LOGGER.isInfoEnabled())
    {
        LOGGER.info("NettyClientBootstrap has started");
    }
}

```

7. reconnect方法

```

void reconnect(String transactionServiceGroup) {
    List<String> availList = null;
    try {
        //获取服务器列表
        availList = getAvailServerList(transactionServiceGroup);
    } catch (Exception e) {
        LOGGER.error("Failed to get available servers: {}",
            e.getMessage(), e);
        return;
    }
    if (CollectionUtils.isEmpty(availList)) {
        String serviceGroup = RegistryFactory.getInstance()

```

```

.getServiceGroup(transactionServiceGroup);
    LOGGER.error("no available service '{}' found, please make sure
registry config correct", serviceGroup);
    return;
}
for (String serverAddress : availList) {
    try {
        //获得通道
        acquireChannel(serverAddress);
    } catch (Exception e) {
        LOGGER.error("{} can not connect to {} cause:
{}", FrameworkErrorCode.NetConnect.getErrCode(), serverAddress,
e.getMessage(), e);
    }
}
}
}

```

8. seata-server 的 RegTmProcessor处理TM注册

```

/*
 * Copyright 1999-2019 Seata.io Group.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
 * implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package io.seata.core.rpc.processor.server;

import io.netty.channel.ChannelHandlerContext;
import io.seata.common.loader.EnhancedServiceLoader;
import io.seata.common.util.NetUtil;
import io.seata.core.protocol.RegisterTMRequest;
import io.seata.core.protocol.RegisterTMResponse;
import io.seata.core.protocol.RpcMessage;
import io.seata.core.protocol.Version;
import io.seata.core.rpc.netty.ChannelManager;
import io.seata.core.rpc.RemotingServer;
import io.seata.core.rpc.RegisterCheckAuthHandler;
import io.seata.core.rpc.processor.RemotingProcessor;

```

```

import org.apache.commons.lang.StringUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * process TM client registry message.
 * <p>
 * process message type:
 * {@link RegisterTMRequest}
 *
 * @author zhangchenghui.dev@gmail.com
 * @since 1.3.0
 */
public class RegTmProcessor implements RemotingProcessor {

    private static final Logger LOGGER =
LoggerFactory.getLogger(RegTmProcessor.class);

    private RemotingServer remotingServer;

    private RegisterCheckAuthHandler checkAuthHandler;

    public RegTmProcessor(RemotingServer remotingServer) {
        this.remotingServer = remotingServer;
        this.checkAuthHandler =
EnhancedServiceLoader.load(RegisterCheckAuthHandler.class);
    }

    @Override
    public void process(ChannelHandlerContext ctx, RpcMessage rpcMessage)
throws Exception {
        onRegTmMessage(ctx, rpcMessage);
    }

    private void onRegTmMessage(ChannelHandlerContext ctx, RpcMessage
rpcMessage) {
        RegisterTMRequest message = (RegisterTMRequest)
rpcMessage.getBody();
        String ipAndPort =
NetUtil.toStringAddress(ctx.channel().remoteAddress());
        Version.putChannelVersion(ctx.channel(), message.getVersion());
        boolean isSuccess = false;
        String errorInfo = StringUtils.EMPTY;
        try {
            if (null == checkAuthHandler ||
checkAuthHandler.regTransactionManagerCheckAuth(message)) {
                ChannelManager.registerTMChannel(message, ctx.channel());
                Version.putChannelVersion(ctx.channel(),
message.getVersion());
            }
        }
    }

```



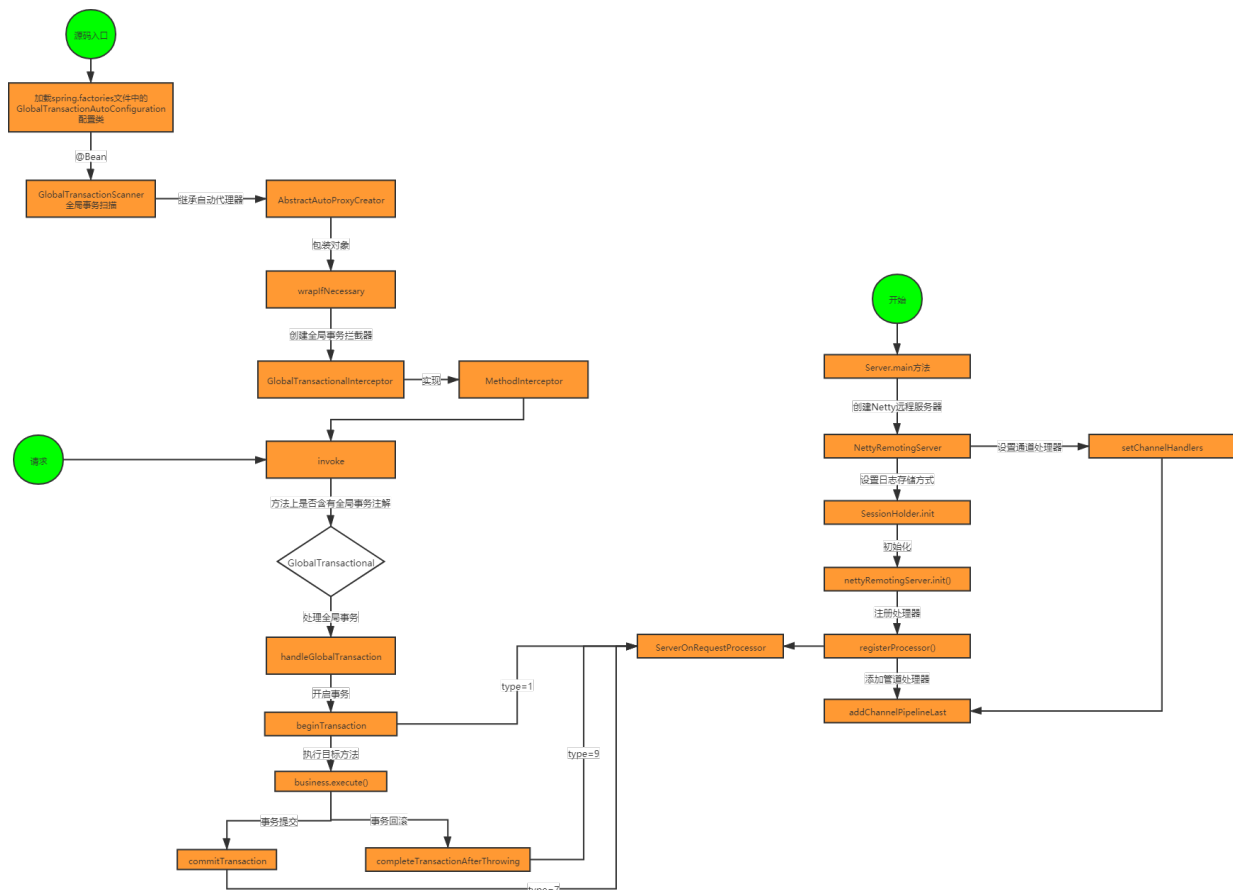
```

        isSuccess = true;
        if (LOGGER.isDebugEnabled()) {
            LOGGER.debug("checkAuth for client:{},vgroup:
{ },applicationId:{ }",
                ipAndPort, message.getTransactionServiceGroup(),
message.getApplicationId());
        }
    }
} catch (Exception exx) {
    isSuccess = false;
    errorInfo = exx.getMessage();
    LOGGER.error("TM register fail, error message:{ }", errorInfo);
}
RegisterTMResponse response = new RegisterTMResponse(isSuccess);
if (StringUtils.isNotEmpty(errorInfo)) {
    response.setMsg(errorInfo);
}
remotingServer.sendAsyncResponse(rpcMessage, ctx.channel(),
response);
if (LOGGER.isInfoEnabled()) {
    LOGGER.info("TM register success,message:{ },channel:{ },client
version:{ }", message, ctx.channel(),
        message.getVersion());
}
}
}

```

5.2.2 TM开启全局事务

5.2.2.1 TM开启全局事务流程分析



5.2.2 主要源码跟踪

1. wrapIfNecessary方法

```

protected Object wrapIfNecessary(Object bean, String beanName, Object
cacheKey) {
    if (disableGlobalTransaction) {
        return bean;
    }
    try {
        synchronized (PROXYED_SET) {
            if (PROXYED_SET.contains(beanName)) {
                return bean;
            }
            interceptor = null;
            //检查TCC代理
            if (TCCBeanParserUtils.isTccAutoProxy(bean, beanName,
applicationContext)) {
                //TCC interceptor, proxy bean of
sofa:reference/dubbo:reference, and LocalTCC
                interceptor = new
TccActionInterceptor(TCCBeanParserUtils.getRemotingDesc(beanName));
            } else {
                Class<?> serviceInterface =
SpringProxyUtils.findTargetClass(bean);
            }
        }
    }
}
  
```

```

        Class<?>[] interfacesIfJdk =
SpringProxyUtils.findInterfaces(bean);

        if (!existsAnnotation(new Class[]{serviceInterface})
            && !existsAnnotation(interfacesIfJdk)) {
            return bean;
        }

        if (interceptor == null) {
            if (globalTransactionalInterceptor == null) {
                //创建全局事务拦截器
                globalTransactionalInterceptor = new
GlobalTransactionalInterceptor(failureHandlerHook);
                ConfigurationCache.addConfigListener(
                    ConfigurationKeys.DISABLE_GLOBAL_TRANSACTION,
(ConfigurationChangeListener)globalTransactionalInterceptor);
            }
            interceptor = globalTransactionalInterceptor;
        }

        LOGGER.info("Bean[{}] with name [{}] would use interceptor
[{}]", bean.getClass().getName(), beanName,
interceptor.getClass().getName());
        if (!AopUtils.isAopProxy(bean)) {
            bean = super.wrapIfNecessary(bean, beanName, cacheKey);
        } else {
            AdvisedSupport advised =
SpringProxyUtils.getAdvisedSupport(bean);
            Advisor[] advisor = buildAdvisors(beanName,
getAdvicesAndAdvisorsForBean(null, null, null));
            for (Advisor avr : advisor) {
                advised.addAdvisor(0, avr);
            }
        }
        PROXYED_SET.add(beanName);
        return bean;
    }
} catch (Exception exx) {
    throw new RuntimeException(exx);
}
}

```

2. GlobalTransactionalInterceptor的invoke方法

```

public Object invoke(final MethodInvocation methodInvocation) throws
Throwable {
    Class<?> targetClass =

```

```

        methodInvocation.getThis() != null ?
AopUtils.getTargetClass(methodInvocation.getThis()) : null;
        Method specificMethod =
ClassUtils.getMostSpecificMethod(methodInvocation.getMethod(),
targetClass);
        if (specificMethod != null &&
!specificMethod.getDeclaringClass().equals(Object.class)) {
            final Method method =
BridgeMethodResolver.findBridgedMethod(specificMethod);
            //获取方法上GlobalTransactional注解
            final GlobalTransactional globalTransactionalAnnotation =
getAnnotation(method, targetClass, GlobalTransactional.class);
            final GlobalLock globalLockAnnotation = getAnnotation(method,
targetClass, GlobalLock.class);
            boolean localDisable = disable || (degradeCheck && degradeNum >=
degradeCheckAllowTimes);
            if (!localDisable) {
                //判断注解是否为空
                if (globalTransactionalAnnotation != null) {
                    //处理全局事务
                    return handleGlobalTransaction(methodInvocation,
globalTransactionalAnnotation);
                } else if (globalLockAnnotation != null) {
                    return handleGlobalLock(methodInvocation);
                }
            }
        }
        return methodInvocation.proceed();
    }
}

```

3. handleGlobalTransaction的execute方法

```

public Object execute(TransactionalExecutor business) throws Throwable {
    // 1 获取事务信息
    TransactionInfo txInfo = business.getTransactionInfo();
    if (txInfo == null) {
        throw new ShouldNeverHappenException("transactionInfo does not
exist");
    }
    // 1.1 获取或者创建一个全局事务
    GlobalTransaction tx = GlobalTransactionContext.getCurrentOrCreate();

    // 1.2 处理事务传播
    Propagation propagation = txInfo.getPropagation();
    SuspendedResourcesHolder suspendedResourcesHolder = null;
    try {
        switch (propagation) {
            case NOT_SUPPORTED:
                suspendedResourcesHolder = tx.suspend(true);

```

```

        return business.execute();
    case REQUIRES_NEW:
        suspendedResourcesHolder = tx.suspend(true);
        break;
    case SUPPORTS:
        if (!existingTransaction()) {
            return business.execute();
        }
        break;
    case REQUIRED:
        break;
    case NEVER:
        if (existingTransaction()) {
            throw new TransactionException(
                String.format("Existing transaction found for
transaction marked with propagation 'never',xid = %s"
                    ,RootContext.getXID()));
        } else {
            return business.execute();
        }
    case MANDATORY:
        if (!existingTransaction()) {
            throw new TransactionException("No existing
transaction found for transaction marked with propagation 'mandatory'");
        }
        break;
    default:
        throw new TransactionException("Not Supported
Propagation:" + propagation);
}

try {

    // 2. 开启事务
    beginTransaction(txInfo, tx);

    Object rs = null;
    try {

        // 执行目标方法
        rs = business.execute();

    } catch (Throwable ex) {

        // 3.回滚所需的业务异常。
        completeTransactionAfterThrowing(txInfo, tx, ex);
        throw ex;
    }
}

```

```

        // 4. 事务提交
        commitTransaction(tx);

        return rs;
    } finally {
        //5. clear
        triggerAfterCompletion();
        cleanUp();
    }
} finally {
    tx.resume(suspendedResourcesHolder);
}
}

```

4. beginTransaction方法最终会发送seata server 也就是TC

```

public String begin(String applicationId, String transactionServiceGroup,
String name, int timeout)
    throws TransactionException {
    //创建全局开启事务请求
    GlobalBeginRequest request = new GlobalBeginRequest();
    request.setTransactionName(name);
    request.setTimeout(timeout);
    //同步发送
    GlobalBeginResponse response = (GlobalBeginResponse)
syncCall(request);
    if (response.getResultCode() == ResultCode.Failed) {
        throw new
TmTransactionException(TransactionExceptionCode.BeginFailed,
response.getMsg());
    }
    return response.getXid();
}

```

5. seata-server 的ServerOnRequestProcessor处理请求

```

public class ServerOnRequestProcessor implements RemotingProcessor {

    private static final Logger LOGGER =
LoggerFactory.getLogger(ServerOnRequestProcessor.class);

    private RemotingServer remotingServer;

    private TransactionMessageHandler transactionMessageHandler;

```

```

    public ServerOnRequestProcessor(RemotingServer remotingServer,
TransactionMessageHandler transactionMessageHandler) {
        this.remotingServer = remotingServer;
        this.transactionMessageHandler = transactionMessageHandler;
    }

    @Override
    public void process(ChannelHandlerContext ctx, RpcMessage rpcMessage)
throws Exception {
        if (ChannelManager.isRegistered(ctx.channel())) {
            onRequestMessage(ctx, rpcMessage);
        } else {
            try {
                if (LOGGER.isInfoEnabled()) {
                    LOGGER.info("closeChannelHandlerContext channel:" +
ctx.channel());
                }
                ctx.disconnect();
                ctx.close();
            } catch (Exception exx) {
                LOGGER.error(exx.getMessage());
            }
            if (LOGGER.isInfoEnabled()) {
                LOGGER.info(String.format("close a unhandled connection!
[%s]", ctx.channel().toString()));
            }
        }
    }

    private void onRequestMessage(ChannelHandlerContext ctx, RpcMessage
rpcMessage) {
        Object message = rpcMessage.getBody();
        RpcContext rpcContext =
ChannelManager.getContextFromIdentified(ctx.channel());
        if (LOGGER.isDebugEnabled()) {
            LOGGER.debug("server received:{},clientId:{},vgroup:{},",
message,
                NetUtil.toIpAddress(ctx.channel().remoteAddress()),
rpcContext.getTransactionServiceGroup());
        } else {
            try {
                BatchLogHandler.INSTANCE.getLogQueue()
                    .put(message + ",clientId:" +
NetUtil.toIpAddress(ctx.channel().remoteAddress()) + ",vgroup:"
+ rpcContext.getTransactionServiceGroup());
            } catch (InterruptedException e) {
                LOGGER.error("put message to logQueue error: {}",
e.getMessage(), e);
            }
        }
    }

```

```

    }
    if (!(message instanceof AbstractMessage)) {
        return;
    }
    if (message instanceof MergedWarpMessage) {
        AbstractResultMessage[] results = new
AbstractResultMessage[((MergedWarpMessage) message).msgs.size()];
        for (int i = 0; i < results.length; i++) {
            final AbstractMessage subMessage = ((MergedWarpMessage)
message).msgs.get(i);
            results[i] =
transactionMessageHandler.onRequest(subMessage, rpcContext);
        }
        MergeResultMessage resultMessage = new MergeResultMessage();
        resultMessage.setMsgs(results);
        remotingServer.sendAsyncResponse(rpcMessage, ctx.channel(),
resultMessage);
    } else {
        // 处理请求消息
        final AbstractMessage msg = (AbstractMessage) message;
        AbstractResultMessage result =
transactionMessageHandler.onRequest(msg, rpcContext);
        //响应
        remotingServer.sendAsyncResponse(rpcMessage, ctx.channel(),
result);
    }
}
}
}

```

6. onRequest的方法最终DefaultCoordinator的doGlobalBegin方法

```

protected void doGlobalBegin(GlobalBeginRequest request,
GlobalBeginResponse response, RpcContext rpcContext)
    throws TransactionException {
    //开启全局事务
    response.setXid(core.begin(rpcContext.getApplicationId(),
rpcContext.getTransactionServiceGroup(),
request.getTransactionName(), request.getTimeout()));
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Begin new global transaction applicationId:
{},transactionServiceGroup: {}, transactionName: {},timeout:{},xid:{",
rpcContext.getApplicationId(),
rpcContext.getTransactionServiceGroup(), request.getTransactionName(),
request.getTimeout(), response.getXid());
    }
}
}

```

core.begin开启


```

@Override
public String begin(String applicationId, String transactionServiceGroup,
String name, int timeout)
    throws TransactionException {
    //创建全局会话
    GlobalSession session =
GlobalSession.createGlobalSession(applicationId, transactionServiceGroup,
name,
    timeout);
    //添加监听器

    session.addSessionLifecycleListener(SessionHolder.getRootSessionManager()
);
    //开启
    session.begin();

    // transaction start event
    eventBus.post(new GlobalTransactionEvent(session.getTransactionId(),
GlobalTransactionEvent.ROLE_TC,
        session.getTransactionName(), session.getBeginTime(), null,
session.getStatus()));

    return session.getXid();
}

```

session.begin()-->DataBaseSessionManager的addGlobalSession

```

@Override
public void addGlobalSession(GlobalSession session) throws
TransactionException {
    if (StringUtils.isBlank(taskName)) {
        //写入会话
        boolean ret =
transactionStoreManager.writeSession(LogOperation.GLOBAL_ADD, session);
        if (!ret) {
            throw new StoreException("addGlobalSession failed.");
        }
    } else {
        boolean ret =
transactionStoreManager.writeSession(LogOperation.GLOBAL_UPDATE, session);
        if (!ret) {
            throw new StoreException("addGlobalSession failed.");
        }
    }
}
}

```

writeSession方法

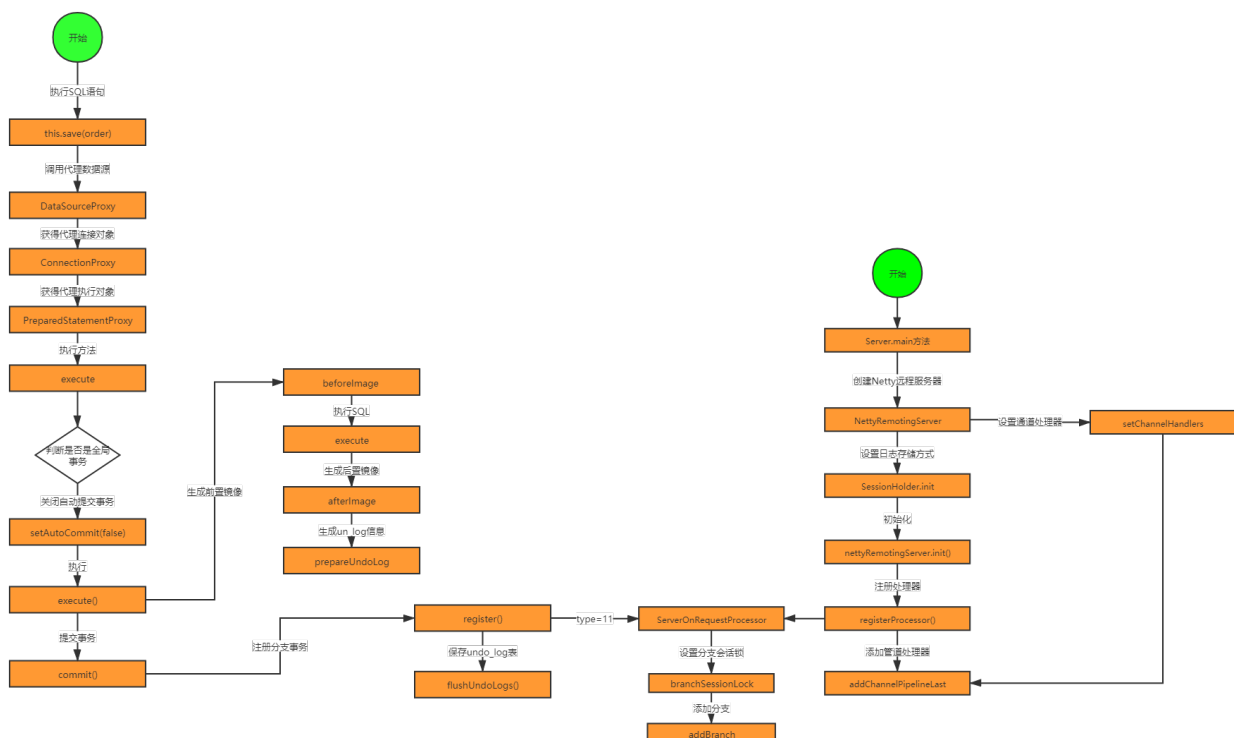
```

public boolean writeSession(LogOperation logOperation, SessionStorable
session) {
    if (LogOperation.GLOBAL_ADD.equals(logOperation)) {
        return
logStore.insertGlobalTransactionDO(convertGlobalTransactionDO(session));
    } else if (LogOperation.GLOBAL_UPDATE.equals(logOperation)) {
        return
logStore.updateGlobalTransactionDO(convertGlobalTransactionDO(session));
    } else if (LogOperation.GLOBAL_REMOVE.equals(logOperation)) {
        return
logStore.deleteGlobalTransactionDO(convertGlobalTransactionDO(session));
    } else if (LogOperation.BRANCH_ADD.equals(logOperation)) {
        return
logStore.insertBranchTransactionDO(convertBranchTransactionDO(session));
    } else if (LogOperation.BRANCH_UPDATE.equals(logOperation)) {
        return
logStore.updateBranchTransactionDO(convertBranchTransactionDO(session));
    } else if (LogOperation.BRANCH_REMOVE.equals(logOperation)) {
        return
logStore.deleteBranchTransactionDO(convertBranchTransactionDO(session));
    } else {
        throw new StoreException("Unknown LogOperation:" +
logOperation.name());
    }
}

```

5.2.3 RM分支事务注册

5.2.3.1 RM分支事务注册流程分析



5.2.3.2 主要流程源码分析

1. 代理数据源创建

```
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-可以完成分支事务开启，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);
    }
}
```

ConnectionProxy代理连接对象创建

```
@Override
public ConnectionProxy getConnection() throws SQLException {
    Connection targetConnection = targetDataSource.getConnection();
    return new ConnectionProxy(this, targetConnection);
}
```

PreparedStatementProxy代理执行对象创建

```
public PreparedStatement prepareStatement(String sql, int
autoGeneratedKeys) throws SQLException {
    PreparedStatement preparedStatement =
targetConnection.prepareStatement(sql, autoGeneratedKeys);
    return new PreparedStatementProxy(this, preparedStatement, sql);
}
```

2. 当发起数据库操作的时候最终会调用PreparedStatementProxy.execute方法

```
public boolean execute() throws SQLException {
    return ExecuteTemplate.execute(this, (statement, args) ->
statement.execute());
}
```

ExecuteTemplate.execute()

```
public static <T, S extends Statement> T execute(List<SQLRecognizer>
sqlRecognizers,
                                                StatementProxy<S>
statementProxy,
                                                StatementCallback<T, S>
statementCallback,
                                                Object... args) throws
SQLException {

    if (!RootContext.requireGlobalLock() &&
!StringUtils.equals(BranchType.AT.name(), RootContext.getBranchType())) {
        // Just work as original statement
        return
statementCallback.execute(statementProxy.getTargetStatement(), args);
    }

    String dbType = statementProxy.getConnectionProxy().getDbType();
    if (CollectionUtils.isEmpty(sqlRecognizers)) {
        sqlRecognizers = SQLVisitorFactory.get(
            statementProxy.getTargetSQL(),
            dbType);
    }
    Executor<T> executor;
    if (CollectionUtils.isEmpty(sqlRecognizers)) {
        executor = new PlainExecutor<>(statementProxy, statementCallback);
    } else {
        if (sqlRecognizers.size() == 1) {
            //获得SQL识别器
            SQLRecognizer sqlRecognizer = sqlRecognizers.get(0);
            switch (sqlRecognizer.getSQLType()) {
```

```

        case INSERT:
            executor =
EnhancedServiceLoader.load(InsertExecutor.class, dbType,
                            new Class[]{StatementProxy.class,
StatementCallback.class, SQLRecognizer.class},
                            new Object[]{statementProxy,
statementCallback, sqlRecognizer});
            break;
        case UPDATE:
            executor = new UpdateExecutor<>(statementProxy,
statementCallback, sqlRecognizer);
            break;
        case DELETE:
            executor = new DeleteExecutor<>(statementProxy,
statementCallback, sqlRecognizer);
            break;
        case SELECT_FOR_UPDATE:
            executor = new SelectForUpdateExecutor<>
(statementProxy, statementCallback, sqlRecognizer);
            break;
        default:
            executor = new PlainExecutor<>(statementProxy,
statementCallback);
            break;
    }
} else {
    executor = new MultiExecutor<>(statementProxy,
statementCallback, sqlRecognizers);
}
}
T rs;
try {
    //执行
    rs = executor.execute(args);
} catch (Throwable ex) {
    if (!(ex instanceof SQLException)) {
        // Turn other exception into SQLException
        ex = new SQLException(ex);
    }
    throw (SQLException) ex;
}
return rs;
}

```

3. executor.execute方法

```

public T execute(Object... args) throws Throwable {
    //判断是否是全局事务
    if (RootContext.inGlobalTransaction()) {
        String xid = RootContext.getXID();
        statementProxy.getConnectionProxy().bind(xid);
    }
    //设置全局锁

    statementProxy.getConnectionProxy().setGlobalLockRequire(RootContext.requireGlobalLock());
    //执行
    return doExecute(args);
}

```

4. doExecute-->executeAutoCommitTrue(args)

```

protected T executeAutoCommitTrue(Object[] args) throws Throwable {
    ConnectionProxy connectionProxy = statementProxy.getConnectionProxy();
    try {
        //关闭自动提交事务
        connectionProxy.setAutoCommit(false);
        //创建锁重试策略并返回结果
        return new LockRetryPolicy(connectionProxy).execute(() -> {
            //执行
            T result = executeAutoCommitFalse(args);
            //提交事务
            connectionProxy.commit();
            return result;
        });
    } catch (Exception e) {
        // when exception occur in finally,this exception will lost, so just print it here
        LOGGER.error("execute executeAutoCommitTrue error:{}", e.getMessage(), e);
        if (!LockRetryPolicy.isLockRetryPolicyBranchRollbackOnConflict()) {
            connectionProxy.getTargetConnection().rollback();
        }
        throw e;
    } finally {
        connectionProxy.getContext().reset();
        connectionProxy.setAutoCommit(true);
    }
}

```

5. executeAutoCommitFalse(args)方法

```

protected T executeAutoCommitFalse(Object[] args) throws Exception {

```

```

        if (!JdbcConstants.MYSQL.equalsIgnoreCase(getDbType()) &&
            getTableMeta().getPrimaryKeyOnlyName().size() > 1)
        {
            throw new NotSupportYetException("multi pk only support mysql!");
        }
        //创建前镜像
        TableRecords beforeImage = beforeImage();
        //执行SQL
        T result =
statementCallback.execute(statementProxy.getTargetStatement(), args);
        //创建后镜像
        TableRecords afterImage = afterImage(beforeImage);
        // 创建SQLUndoLog日志
        prepareUndoLog(beforeImage, afterImage);
        return result;
    }

```

6. connectionProxy.commit();提交事务方法

```

private void processGlobalTransactionCommit() throws SQLException {
    try {
        //注册分支事务
        register();
    } catch (TransactionException e) {
        recognizeLockKeyConflictException(e, context.buildLockKeys());
    }
    try {
        //保存undo_log信息

        UndoLogManagerFactory.getUndoLogManager(this.getDbType()).flushUndoLogs(this);
        targetConnection.commit();
    } catch (Throwable ex) {
        LOGGER.error("process connectionProxy commit error: {}",
ex.getMessage(), ex);
        report(false);
        throw new SQLException(ex);
    }
    if (IS_REPORT_SUCCESS_ENABLE) {
        report(true);
    }
    context.reset();
}

```

7. server端处理分支事务注册BranchRegisterRequest类的handle方法

```

public BranchRegisterResponse handle(BranchRegisterRequest request, final
RpcContext rpcContext) {

```

```

BranchRegisterResponse response = new BranchRegisterResponse();
exceptionHandleTemplate(new AbstractCallback<BranchRegisterRequest,
BranchRegisterResponse>() {
    @Override
    public void execute(BranchRegisterRequest request,
BranchRegisterResponse response)
        throws TransactionException {
        try {
            //执行分支事务注册
            doBranchRegister(request, response, rpcContext);
        } catch (StoreException e) {
            throw new
TransactionException(TransactionExceptionCode.FailedStore, String
                .format("branch register request failed. xid=%s,
msg=%s", request.getXid(), e.getMessage()), e);
        }
    }
}, request, response);
return response;
}

```

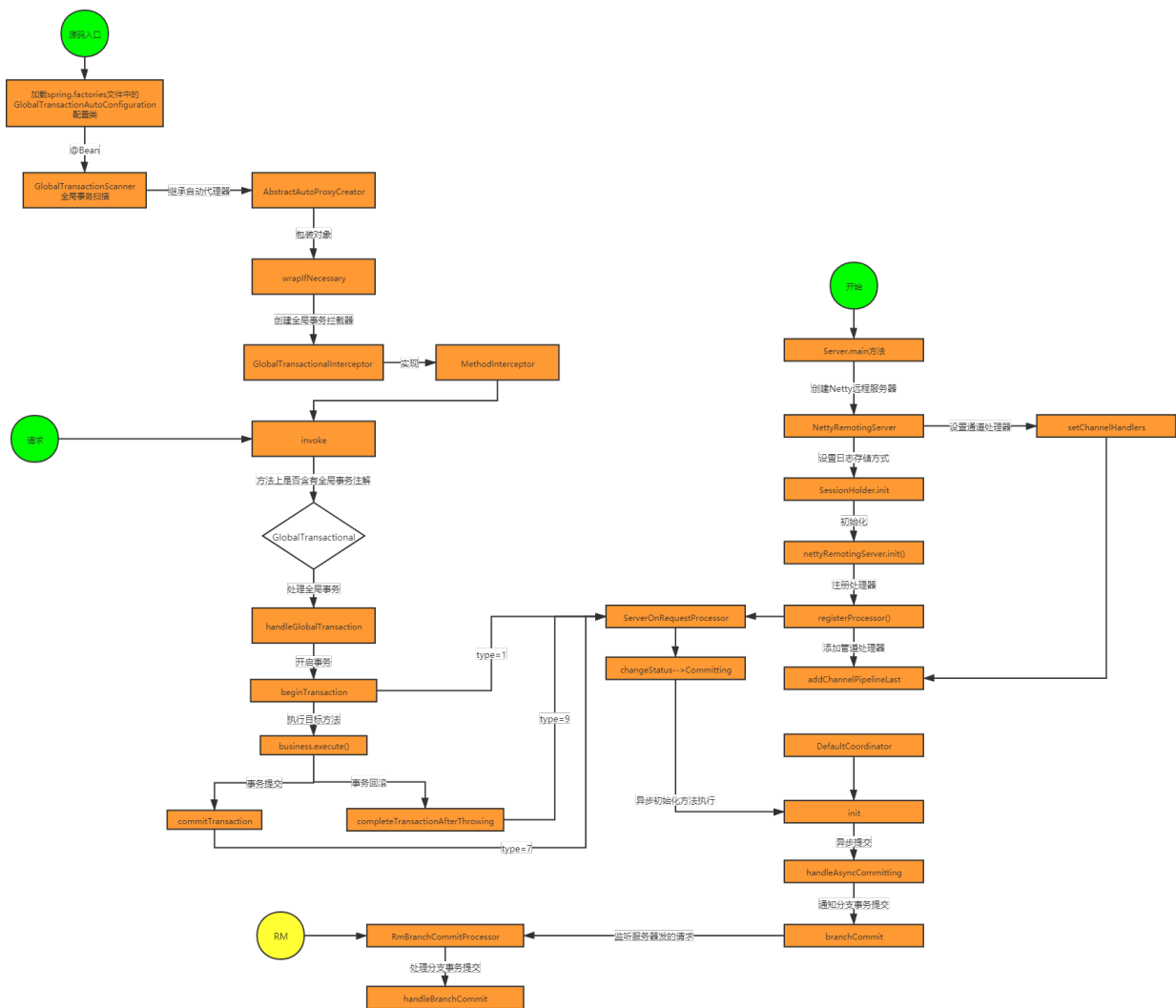
```

@Override
public boolean writeSession(LogOperation logOperation, SessionStorable
session) {
    if (LogOperation.GLOBAL_ADD.equals(logOperation)) {
        //全局事务开启
        return
logStore.insertGlobalTransactionDO(convertGlobalTransactionDO(session));
    } else if (LogOperation.GLOBAL_UPDATE.equals(logOperation)) {
        return
logStore.updateGlobalTransactionDO(convertGlobalTransactionDO(session));
    } else if (LogOperation.GLOBAL_REMOVE.equals(logOperation)) {
        return
logStore.deleteGlobalTransactionDO(convertGlobalTransactionDO(session));
    } else if (LogOperation.BRANCH_ADD.equals(logOperation)) {
        //分支事务注册
        return
logStore.insertBranchTransactionDO(convertBranchTransactionDO(session));
    } else if (LogOperation.BRANCH_UPDATE.equals(logOperation)) {
        return
logStore.updateBranchTransactionDO(convertBranchTransactionDO(session));
    } else if (LogOperation.BRANCH_REMOVE.equals(logOperation)) {
        return
logStore.deleteBranchTransactionDO(convertBranchTransactionDO(session));
    } else {
        throw new StoreException("Unknown LogOperation:" +
logOperation.name());
    }
}

```


5.2.4 TM/RM 事务提交

5.2.4.1 TM/RM 事务提交流程分析



5.2.4.2 主要流程源码分析

1. DefaultCore的commit方法

```
public GlobalStatus commit(String xid) throws TransactionException {
    GlobalSession globalSession = SessionHolder.findGlobalSession(xid);
    if (globalSession == null) {
        return GlobalStatus.Finished;
    }

    globalSession.addSessionLifecycleListener(SessionHolder.getRootSessionManager());
    // just lock changeStatus

    boolean shouldCommit = SessionHolder.lockAndExecute(globalSession, ()
-> {
        // the lock should release after branch commit
    });
}
```

```

        // Highlight: Firstly, close the session, then no more branch can
        be registered.
        globalSession.closeAndClean();
        if (globalSession.getStatus() == GlobalStatus.Begin) {
            //修改全局事务状态为提交
            globalSession.changeStatus(GlobalStatus.Committing);
            return true;
        }
        return false;
    });
    if (!shouldCommit) {
        return globalSession.getStatus();
    }
    if (globalSession.canBeCommittedAsync()) {
        globalSession.asyncCommit();
        return GlobalStatus.Committed;
    } else {
        doGlobalCommit(globalSession, false);
    }
    return globalSession.getStatus();
}

```

2. DefaultCoordinator的init方法

```

public void init() {
    retryRollbacking.scheduleAtFixedRate(() -> {
        try {
            handleRetryRollbacking();
        } catch (Exception e) {
            LOGGER.info("Exception retry rollbacking ... ", e);
        }
    }, 0, ROLLBACKING_RETRY_PERIOD, TimeUnit.MILLISECONDS);

    retryCommitting.scheduleAtFixedRate(() -> {
        try {
            handleRetryCommitting();
        } catch (Exception e) {
            LOGGER.info("Exception retry committing ... ", e);
        }
    }, 0, COMMITTING_RETRY_PERIOD, TimeUnit.MILLISECONDS);

    asyncCommitting.scheduleAtFixedRate(() -> {
        try {
            //异步处理提交事务
            handleAsyncCommitting();
        } catch (Exception e) {
            LOGGER.info("Exception async committing ... ", e);
        }
    }, 0, ASYNC_COMMITTING_RETRY_PERIOD, TimeUnit.MILLISECONDS);
}

```

```

timeoutCheck.scheduleAtFixedRate(() -> {
    try {
        timeoutCheck();
    } catch (Exception e) {
        LOGGER.info("Exception timeout checking ... ", e);
    }
}, 0, TIMEOUT_RETRY_PERIOD, TimeUnit.MILLISECONDS);

undoLogDelete.scheduleAtFixedRate(() -> {
    try {
        undoLogDelete();
    } catch (Exception e) {
        LOGGER.info("Exception undoLog deleting ... ", e);
    }
}, UNDO_LOG_DELAY_DELETE_PERIOD, UNDO_LOG_DELETE_PERIOD,
TimeUnit.MILLISECONDS);
}

```

handleAsyncCommitting方法

```

protected void handleAsyncCommitting() {
    //查找所有全局事务会话
    Collection<GlobalSession> asyncCommittingSessions =
    SessionHolder.getAsyncCommittingSessionManager()
        .allSessions();
    if (CollectionUtils.isEmpty(asyncCommittingSessions)) {
        return;
    }
    for (GlobalSession asyncCommittingSession : asyncCommittingSessions) {
        try {
            // 判断事务状态为提交
            if (GlobalStatus.AsyncCommitting !=
            asyncCommittingSession.getStatus()) {
                continue;
            }

            asyncCommittingSession.addSessionLifecycleListener(SessionHolder.getRootS
            essionManager());
            //执行全局事务提交
            core.doGlobalCommit(asyncCommittingSession, true);
        } catch (TransactionException ex) {
            LOGGER.error("Failed to async committing [{}] {} {}",
            asyncCommittingSession.getXid(), ex.getCode(), ex.getMessage(), ex);
        }
    }
}
}

```

doGlobalCommit方法

```

public boolean doGlobalCommit(GlobalSession globalSession, boolean
retrying) throws TransactionException {
    boolean success = true;
    // start committing event
    eventBus.post(new
GlobalTransactionEvent(globalSession.getTransactionId(),
GlobalTransactionEvent.ROLE_TC,
        globalSession.getTransactionName(), globalSession.getBeginTime(),
null, globalSession.getStatus()));

    if (globalSession.isSaga()) {
        success = getCore(BranchType.SAGA).doGlobalCommit(globalSession,
retrying);
    } else {
        for (BranchSession branchSession :
globalSession.getSortedBranches()) {
            BranchStatus currentStatus = branchSession.getStatus();
            if (currentStatus == BranchStatus.PhaseOne_Failed) {
                globalSession.removeBranch(branchSession);
                continue;
            }
            try {
                //通知分支事务提交
                BranchStatus branchStatus =
getCore(branchSession.getBranchType()).branchCommit(globalSession,
branchSession);

                switch (branchStatus) {
                    case PhaseTwo_Committed:
                        globalSession.removeBranch(branchSession);
                        continue;
                    case PhaseTwo_CommitFailed_Unretryable:
                        if (globalSession.canBeCommittedAsync()) {
                            LOGGER.error(
                                "Committing branch transaction[{}],
status: PhaseTwo_CommitFailed_Unretryable, please check the business
log.", branchSession.getBranchId());
                            continue;
                        } else {
                            SessionHelper.endCommitFailed(globalSession);
                            LOGGER.error("Committing global
transaction[{}] finally failed, caused by branch transaction[{}] commit
failed.", globalSession.getXid(), branchSession.getBranchId());
                            return false;
                        }
                    default:
                        if (!retrying) {
                            globalSession.queueToRetryCommit();
                            return false;
                        }
                }
            } catch (TransactionException e) {
                // ...
            }
        }
    }
}

```

```

        }
        if (globalSession.canBeCommittedAsync()) {
            LOGGER.error("Committing branch
transaction[{}], status:{} and will retry later",
                        branchSession.getBranchId(),
branchStatus);

            continue;
        } else {
            LOGGER.error(
                "Committing global transaction[{}] failed,
caused by branch transaction[{}] commit failed, will retry later.",
globalSession.getXid(), branchSession.getBranchId());
            return false;
        }
    }
} catch (Exception ex) {
    StackTraceLogger.error(LOGGER, ex, "Committing branch
transaction exception: {}");
    new String[] {branchSession.toString()});
    if (!retrying) {
        globalSession.queueToRetryCommit();
        throw new TransactionException(ex);
    }
}
}
if (globalSession.hasBranch()) {
    LOGGER.info("Committing global transaction is NOT done, xid =
{}.", globalSession.getXid());
    return false;
}
}
if (success) {
    SessionHelper.endCommitted(globalSession);

    // committed event
    eventBus.post(new
GlobalTransactionEvent(globalSession.getTransactionId(),
GlobalTransactionEvent.ROLE_TC,
                        globalSession.getTransactionName(),
globalSession.getBeginTime(), System.currentTimeMillis(),
                        globalSession.getStatus()));

    LOGGER.info("Committing global transaction is successfully done,
xid = {}.", globalSession.getXid());
}
return success;
}

```

3. RmBranchCommitProcessor方法

```

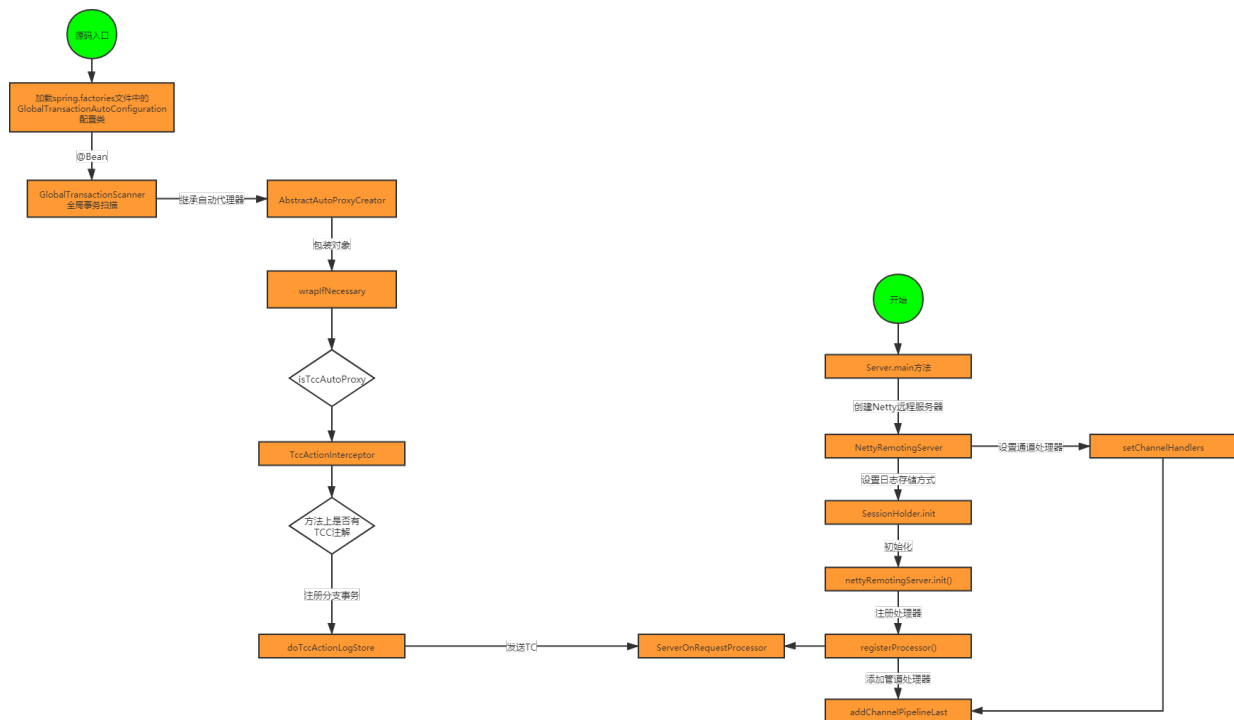
public void process(ChannelHandlerContext ctx, RpcMessage rpcMessage)
throws Exception {
    String remoteAddress =
NetUtil.toStringAddress(ctx.channel().remoteAddress());
    Object msg = rpcMessage.getBody();
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("rm client handle branch commit process:" + msg);
    }
    //处理分支事务提交
    handleBranchCommit(rpcMessage, remoteAddress, (BranchCommitRequest)
msg);
}

private void handleBranchCommit(RpcMessage request, String serverAddress,
BranchCommitRequest branchCommitRequest) {
    BranchCommitResponse resultMessage;
    //发送BranchCommitRequest请求
    resultMessage = (BranchCommitResponse)
handler.onRequest(branchCommitRequest, null);
    if (LOGGER.isDebugEnabled()) {
        LOGGER.debug("branch commit result:" + resultMessage);
    }
    try {
        //响应结果
        this.remotingClient.sendAsyncResponse(serverAddress, request,
resultMessage);
    } catch (Throwable throwable) {
        LOGGER.error("branch commit error: {}", throwable.getMessage(),
throwable);
    }
}
}

```

5.3 TCC源码

5.3.1 TCC源码分支事务注册流程分析



5.3.2 主要流程源码分析

1. GlobalTransactionScanner的wrapIfNecessary方法

```

protected Object wrapIfNecessary(Object bean, String beanName, Object
cacheKey) {
    if (disableGlobalTransaction) {
        return bean;
    }
    try {
        synchronized (PROXYED_SET) {
            if (PROXYED_SET.contains(beanName)) {
                return bean;
            }
            interceptor = null;
            //检查是否是TCC
            if (TCCBeanParserUtils.isTccAutoProxy(bean, beanName,
applicationContext)) {
                //TCC interceptor, proxy bean of
sofa:reference/dubbo:reference, and LocalTCC
                interceptor = new
TccActionInterceptor(TCCBeanParserUtils.getRemotingDesc(beanName));
            } else {
                Class<?> serviceInterface =
SpringProxyUtils.findTargetClass(bean);
                Class<?>[] interfacesIfJdk =
SpringProxyUtils.findInterfaces(bean);

                if (!existsAnnotation(new Class[]{serviceInterface})
&& !existsAnnotation(interfacesIfJdk)) {

```

```

        return bean;
    }

    if (interceptor == null) {
        if (globalTransactionalInterceptor == null) {
            globalTransactionalInterceptor = new
GlobalTransactionalInterceptor(failureHandlerHook);
            ConfigurationCache.addConfigListener(
                ConfigurationKeys.DISABLE_GLOBAL_TRANSACTION,
                (ConfigurationChangeListener)globalTransactionalInterceptor);
        }
        interceptor = globalTransactionalInterceptor;
    }
}

LOGGER.info("Bean[{}] with name [{}] would use interceptor
[{}]", bean.getClass().getName(), beanName,
interceptor.getClass().getName());
if (!AopUtils.isAopProxy(bean)) {
    bean = super.wrapIfNecessary(bean, beanName, cacheKey);
} else {
    AdvisedSupport advised =
SpringProxyUtils.getAdvisedSupport(bean);
    Advisor[] advisor = buildAdvisors(beanName,
getAdvicesAndAdvisorsForBean(null, null, null));
    for (Advisor avr : advisor) {
        advised.addAdvisor(0, avr);
    }
}
PROXYED_SET.add(beanName);
return bean;
}
} catch (Exception exx) {
    throw new RuntimeException(exx);
}
}

```

2. TccActionInterceptor的invoke方法

```

public Object invoke(final MethodInvocation invocation) throws Throwable {
    if (!RootContext.inGlobalTransaction()) {
        //not in transaction
        return invocation.proceed();
    }
    Method method = getActionInterfaceMethod(invocation);
    TwoPhaseBusinessAction businessAction =
method.getAnnotation(TwoPhaseBusinessAction.class);
    //try method
}

```



```

    if (businessAction != null) {
        //保存事务ID
        String xid = RootContext.getXID();
        //保存事务类型
        String previousBranchType = RootContext.getBranchType();
        RootContext.bindBranchType(BranchType.TCC);
        try {
            Object[] methodArgs = invocation.getArguments();
            //处理TCC
            Map<String, Object> ret =
actionInterceptorHandler.proceed(method, methodArgs, xid, businessAction,
                                invocation::proceed);
            //return the final result
            return ret.get(Constants.TCC_METHOD_RESULT);
        }
        finally {
            RootContext.unbindBranchType();
            //restore the TCC branchType if exists
            if (StringUtils.equals(BranchType.TCC.name(),
previousBranchType)) {
                RootContext.bindBranchType(BranchType.TCC);
            }
        }
    }
    return invocation.proceed();
}

```

actionInterceptorHandler.proceed方法

```

public Map<String, Object> proceed(Method method, Object[] arguments,
String xid, TwoPhaseBusinessAction businessAction,
                                Callback<Object> targetCallback) throws
Throwable {
    Map<String, Object> ret = new HashMap<>(4);

    //TCC name
    String actionName = businessAction.name();
    BusinessActionContext actionContext = new BusinessActionContext();
    actionContext.setXid(xid);
    //set action name
    actionContext.setActionName(actionName);

    //创建分支
    String branchId = doTccActionLogStore(method, arguments,
businessAction, actionContext);
    actionContext.setBranchId(branchId);

    //set the parameter whose type is BusinessActionContext
    Class<?>[] types = method.getParameterTypes();
}

```

```

int argIndex = 0;
for (Class<?> cls : types) {
    if (cls.getName().equals(BusinessActionContext.class.getName())) {
        arguments[argIndex] = actionContext;
        break;
    }
    argIndex++;
}
//the final parameters of the try method
ret.put(Constants.TCC_METHOD_ARGUMENTS, arguments);
//the final result
ret.put(Constants.TCC_METHOD_RESULT, targetCallback.execute());
return ret;
}

```

3. TM在发起全局事务提交后, RM端会根据事务类型选择TCCResourceManager的branchCommit方法

```

public BranchStatus branchCommit(BranchType branchType, String xid, long
branchId, String resourceId,
                                String applicationData) throws
TransactionException {
    TCCResource tccResource =
(TCCResource)tccResourceCache.get(resourceId);
    if (tccResource == null) {
        throw new ShouldNeverHappenException(String.format("TCC resource
is not exist, resourceId: %s", resourceId));
    }
    Object targetTCCBean = tccResource.getTargetBean();
    Method commitMethod = tccResource.getCommitMethod();
    if (targetTCCBean == null || commitMethod == null) {
        throw new ShouldNeverHappenException(String.format("TCC resource
is not available, resourceId: %s", resourceId));
    }
    try {
        //BusinessActionContext
        BusinessActionContext businessActionContext =
getBusinessActionContext(xid, branchId, resourceId,
applicationData);
        //执行自定义commit方法
        Object ret = commitMethod.invoke(targetTCCBean,
businessActionContext);
        LOGGER.info("TCC resource commit result : {}, xid: {}, branchId:
{}, resourceId: {}", ret, xid, branchId, resourceId);
        boolean result;
        if (ret != null) {
            if (ret instanceof TwoPhaseResult) {
                result = ((TwoPhaseResult)ret).isSuccess();
            } else {

```

```
        result = (boolean)ret;
    }
} else {
    result = true;
}
return result ? BranchStatus.PhaseTwo_Committed :
BranchStatus.PhaseTwo_CommitFailed_Retryable;
} catch (Throwable t) {
    String msg = String.format("commit TCC resource error, resourceId:
%s, xid: %s.", resourceId, xid);
    LOGGER.error(msg, t);
    return BranchStatus.PhaseTwo_CommitFailed_Retryable;
}
}
```