

## 第2章 分布式事务框架

### 1 分布式事务实战-seata的AT模式

#### 1.1 fescar/seata概述

##### 1.1.1 fescar简介

FESCAR是阿里巴巴 开源的分布式事务中间件，以高效并且对业务0侵入的方式，解决微服务场景下面临的分布式事务问题。它全称是Fast & EaSy Commit And Rollback，是阿里内部的TxC（Taobao Transaction Constructor）和阿里云上的GTS（Global Transaction Service）的提炼出的产物，作为开源的分布式事务解决方案，被很多项目团队青睐。

##### 1.1.2 fescar和seata的关系

Seata是fescar的升级版本，从2019年4月起，更名为seata。seata也是由一组单词组成：**Simple Extensible Autonomous Transaction Architecture**。[Git网址](#)提供了对他的详细介绍，包括[Quick Start](#)。



Seata: Simple Extensible Autonomous Transaction Architecture

build passing codecov 50% license Apache-2.0 maven-central v1.2.0 Follow 61

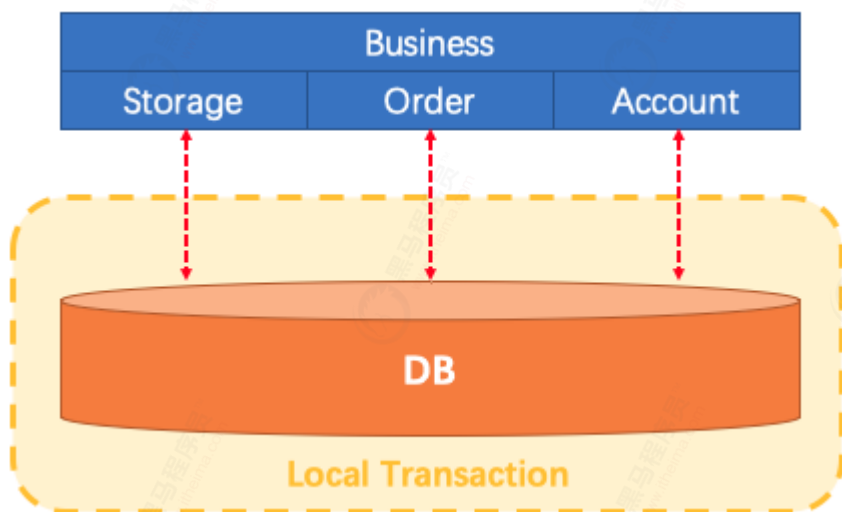
#### What is Seata?

A distributed transaction solution with high performance and ease of use for microservices architecture.

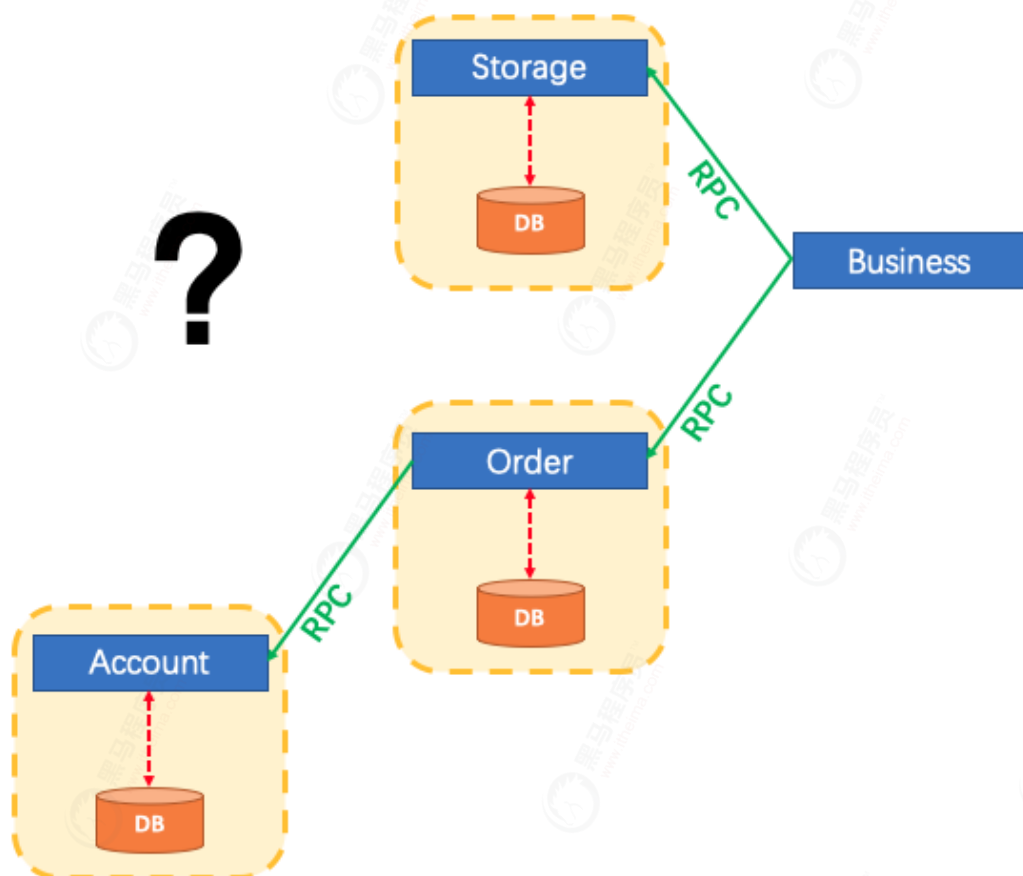
##### 1.1.3 seata的典型案例分析

在[seata的github网站](#)中提供了一套详细的业务流程介绍，我们就以此为例，展开讲解。只不过原网站是英文的，我们把它简单翻译一下。

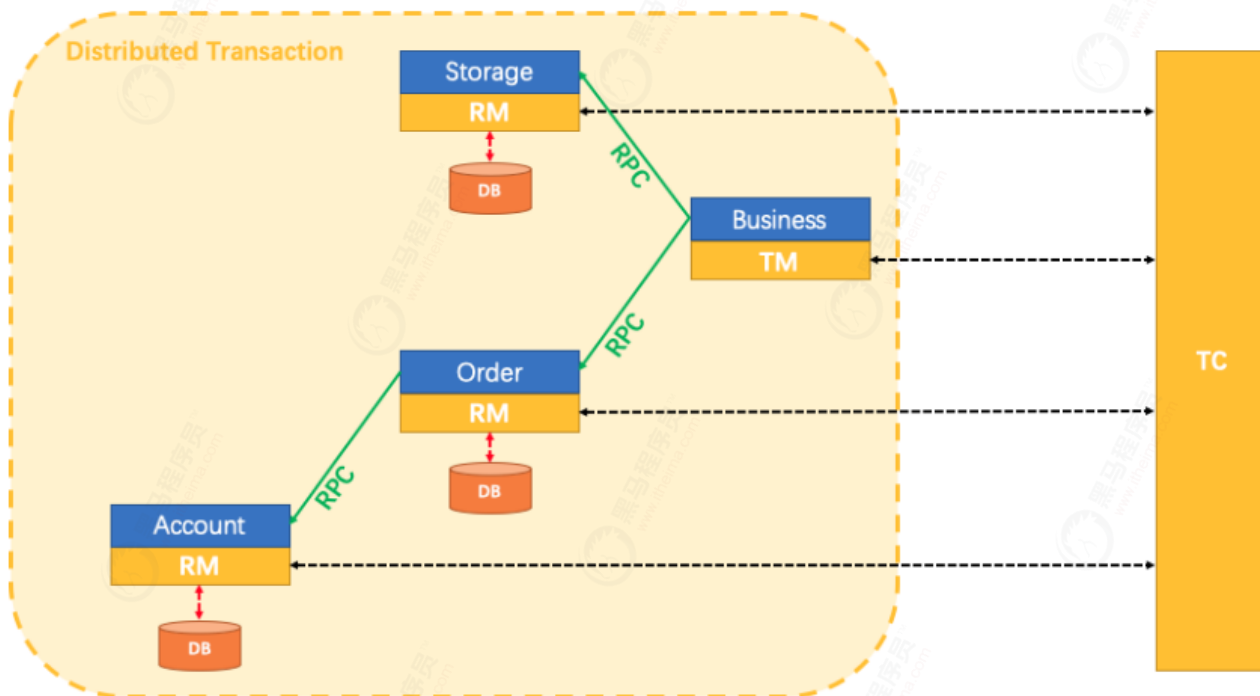
首先是单体架构的事务场景介绍：电商购物中的3个业务模块（库存、订单和账户）。他们操作本地数据库，由于同库同源，所以本地事务可以保证其一致性，如下图所示：



但是在微服务架构中，由于每个业务模块变成了独立的服务，且每个服务连接自己的数据库，此时由原来同库同源变成了不同数据库不同数据源的情况，虽然每个服务自己的数据库操作仍然可以使用本地事务控制，但是要让服务间的事务保持一致性就需要用到分布式事务了。如下图所示：



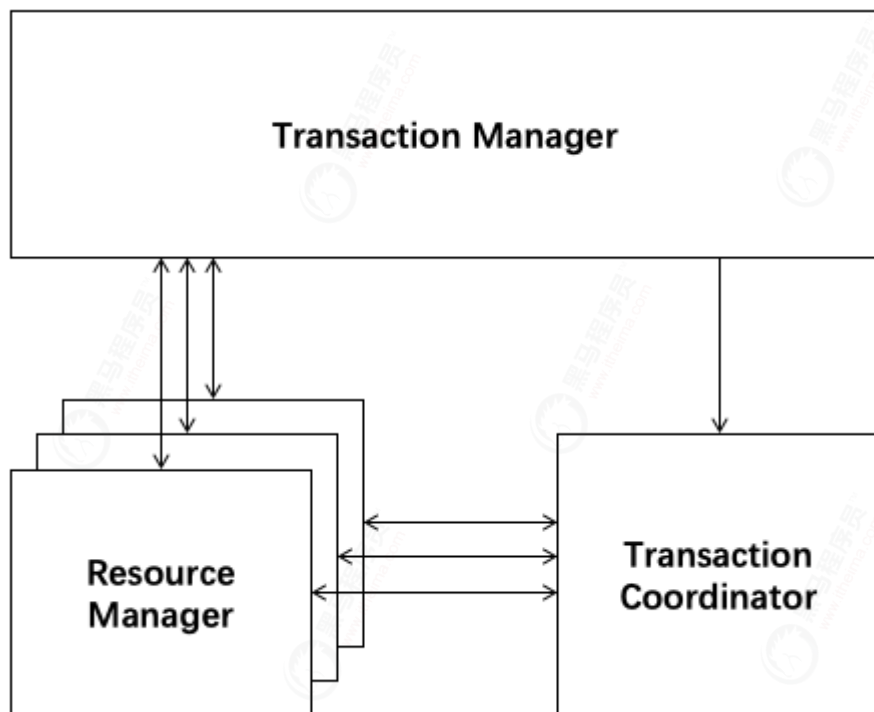
在此时，引入了seata，它提供了一个“完美”的解决方案。如下图所示：



在图中的三个和seata相关的组件，前面在介绍fescar的时候已经介绍了，它也是seata的基本组件，他们分别是：

- 1) 事务协调器（TC）：维护全局和分支事务的状态，驱动全局提交或回滚。
- 2) 事务管理器（TM）：定义全局事务的范围：开始全局事务，提交或回滚全局事务。
- 3) 资源管理器（RM）：管理分支事务的资源，与TC通信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。

他们的结构图如下：



A typical lifecycle of Seata managed distributed transaction: 一个典型的seata管理分布式事务的生命周期

1. TM asks TC to begin a new global transaction. TC generates an XID representing the global transaction.
2. XID is propagated through microservices' invoke chain.
3. RM register local transaction as a branch of the corresponding global transaction of XID to TC.
4. TM asks TC for committing or rollbacking the corresponding global transaction of XID.
5. TC drives all branch transactions under the corresponding global transaction of XID to finish branch committing or rollbacking.

1.TM要求TC开始新的全局事务。TC生成表示全局事务的XID。

2.XID通过微服务的调用链传播。

3.RM将本地事务注册为XID到TC的相应全局事务的分支。

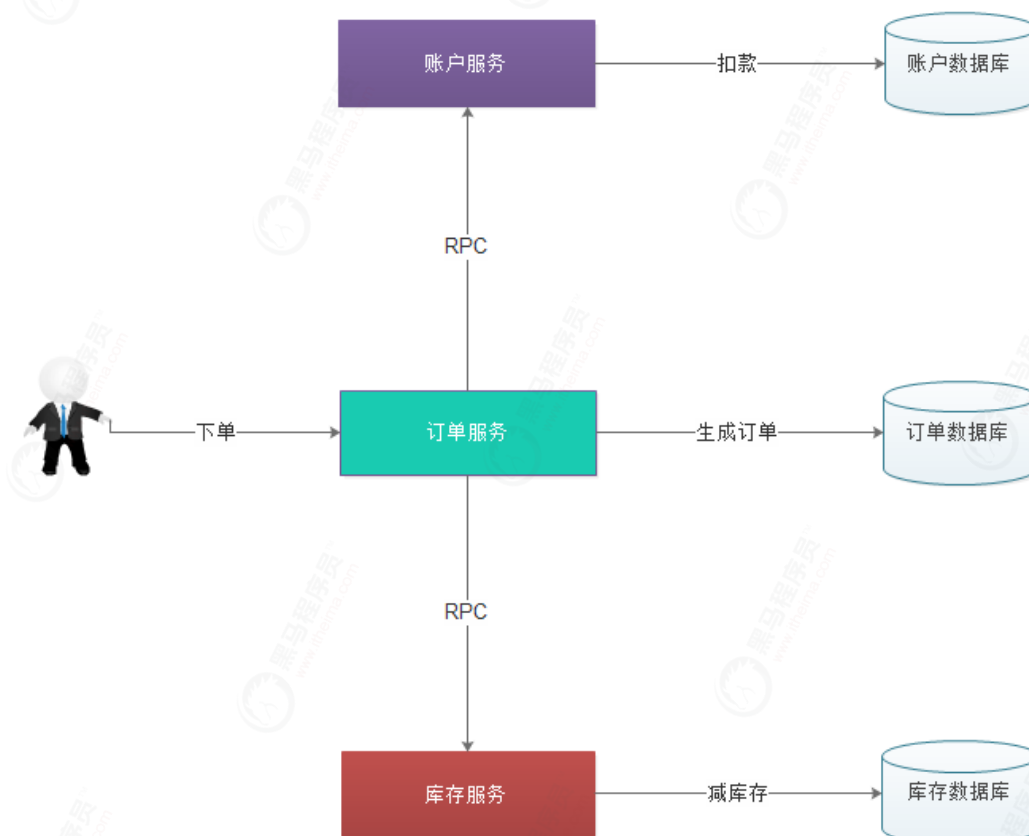
4.TM要求TC提交或回滚XID的相应全局事务。

5.TC在XID的相应全局事务下驱动所有分支事务，以完成分支提交或回滚。

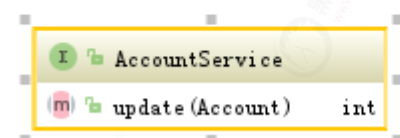
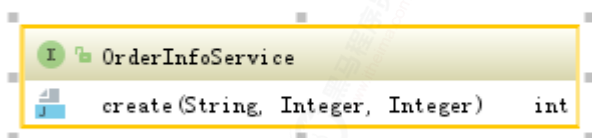
## 1.2 案例简介

### 1.2.1 场景说明

我们的案例就参照seata官网上提供的业务模型，服务框架我们采用的是dubbo+zk的组合。



### 1.2.2 功能说明



## order库

```
SET FOREIGN_KEY_CHECKS=0;
```

```
-- Table structure for `order_info`
```

```
DROP TABLE IF EXISTS `order_info`;  
CREATE TABLE `order_info` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `money` bigint(20) NOT NULL,  
  `createtime` datetime NOT NULL,  
  `usernumber` varchar(20) DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8;
```

```
-- Records of order_info
```

```
-- Table structure for `undo_log`
```

```
DROP TABLE IF EXISTS `undo_log`;  
CREATE TABLE `undo_log` (  
  `id` bigint(20) NOT NULL AUTO_INCREMENT,  
  `branch_id` bigint(20) NOT NULL,  
  `xid` varchar(100) NOT NULL,  
  `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`),  
  KEY `idx_unionkey` (`xid`,`branch_id`)  
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;
```

```
-- Records of undo_log
```

```
INSERT INTO `undo_log` VALUES ('1', '2007238082', '192.168.211.1:8091:2007238081',  
0x7B226272616E63684964223A323030373233383038322C2273716C556E646F4C6F6773223A5B7B226166746572496D  
616765223A7B22726F7773223A5B7B226669656C6473223A5B7B226B657954797065223A225072696D6172794B657922  
2C226E616D65223A226964222C2274797065223A342C2276616C7565223A357D2C7B226B657954797065223A224E554C  
4C222C226E616D65223A226D6F6E6579222C2274797065223A2D352C2276616C7565223A3530307D2C7B226B65795479  
7065223A224E554C4C222C226E616D65223A2263726561746574696D65222C2274797065223A39332C2276616C756522  
3A22323031392D30332D32352031383A34303A3034227D2C7B226B657954797065223A224E554C4C222C226E616D6522  
3A22757365726E756D626572222C2274797065223A31327D5D7D5D2C227461626C654E616D65223A226F726465725F69  
6E666F227D2C226265666F7265496D616765223A7B22726F7773223A5B5D2C227461626C654E616D65223A226F726465  
725F696E666F227D2C2273716C54797065223A22494E53455254222C227461626C654E616D65223A226F726465725F69  
6E666F227D5D2C22786964223A223139322E3136382E3231312E313A383039313A32303037323338303831227D, '0',  
'2019-03-25 18:40:04', '2019-03-25 18:40:04', null);
```

```
INSERT INTO `undo_log` VALUES ('3', '2007238092', '192.168.211.1:8091:2007238091',
```

```
0x7B226272616E63684964223A323030373233383039322C2273716C556E646F4C6F6773223A5B7B226166746572496D
616765223A7B22726F7773223A5B7B226669656C6473223A5B7B226B657954797065223A225072696D6172794B657922
2C226E616D65223A226964222C2274797065223A342C2276616C7565223A377D2C7B226B657954797065223A224E554C
4C222C226E616D65223A226D6F6E6579222C2274797065223A2D352C2276616C7565223A3530307D2C7B226B65795479
7065223A224E554C4C222C226E616D65223A2263726561746574696D65222C2274797065223A39332C2276616C756522
3A22323031392D30332D32352031393A32313A3436227D2C7B226B657954797065223A224E554C4C222C226E616D6522
3A22757365726E756D626572222C2274797065223A31327D5D7D5D2C227461626C654E616D65223A226F726465725F69
6E666F227D2C226265666F7265496D616765223A7B22726F7773223A5B5D2C227461626C654E616D65223A226F726465
725F696E666F227D2C2273716C54797065223A22494E53455254222C227461626C654E616D65223A226F726465725F69
6E666F227D5D2C22786964223A223139322E3136382E3231312E313A383039313A32303037323338303931227D, '0',
'2019-03-25 19:21:46', '2019-03-25 19:21:46', null);
INSERT INTO `undo_log` VALUES ('4', '2007238096', '192.168.211.1:8091:2007238095',
0x7B226272616E63684964223A323030373233383039362C2273716C556E646F4C6F6773223A5B7B226166746572496D
616765223A7B22726F7773223A5B7B226669656C6473223A5B7B226B657954797065223A225072696D6172794B657922
2C226E616D65223A226964222C2274797065223A342C2276616C7565223A387D2C7B226B657954797065223A224E554C
4C222C226E616D65223A226D6F6E6579222C2274797065223A2D352C2276616C7565223A3530307D2C7B226B65795479
7065223A224E554C4C222C226E616D65223A2263726561746574696D65222C2274797065223A39332C2276616C756522
3A22323031392D30332D32352031393A32333A3031227D2C7B226B657954797065223A224E554C4C222C226E616D6522
3A22757365726E756D626572222C2274797065223A31327D5D7D5D2C227461626C654E616D65223A226F726465725F69
6E666F227D2C226265666F7265496D616765223A7B22726F7773223A5B5D2C227461626C654E616D65223A226F726465
725F696E666F227D2C2273716C54797065223A22494E53455254222C227461626C654E616D65223A226F726465725F69
6E666F227D5D2C22786964223A223139322E3136382E3231312E313A383039313A32303037323338303935227D, '0',
'2019-03-25 19:23:18', '2019-03-25 19:23:18', null);
```

item库

```

SET FOREIGN_KEY_CHECKS=0;

-----
-- Table structure for `item`
-----
DROP TABLE IF EXISTS `item`;
CREATE TABLE `item` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` varchar(100) DEFAULT NULL,
  `price` bigint(20) NOT NULL,
  `num` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

-----
-- Records of item
-----
INSERT INTO `item` VALUES ('1', '华为荣耀4', '100', '100');

-----
-- Table structure for `undo_log`
-----
DROP TABLE IF EXISTS `undo_log`;
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `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`),
  KEY `idx_unionkey` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;

-----
-- Records of undo_log
-----

```

account库

```

SET FOREIGN_KEY_CHECKS=0;

-----
-- Table structure for `account`
-----
DROP TABLE IF EXISTS `account`;
CREATE TABLE `account` (
  `usernumber` varchar(20) NOT NULL,
  `money` bigint(20) NOT NULL,
  `username` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`usernumber`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-----
-- Records of account
-----
INSERT INTO `account` VALUES ('itheima', '1000', '王五');

-----
-- Table structure for `undo_log`
-----
DROP TABLE IF EXISTS `undo_log`;
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `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`),
  KEY `idx_unionkey` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;

-----
-- Records of undo_log
-----

```

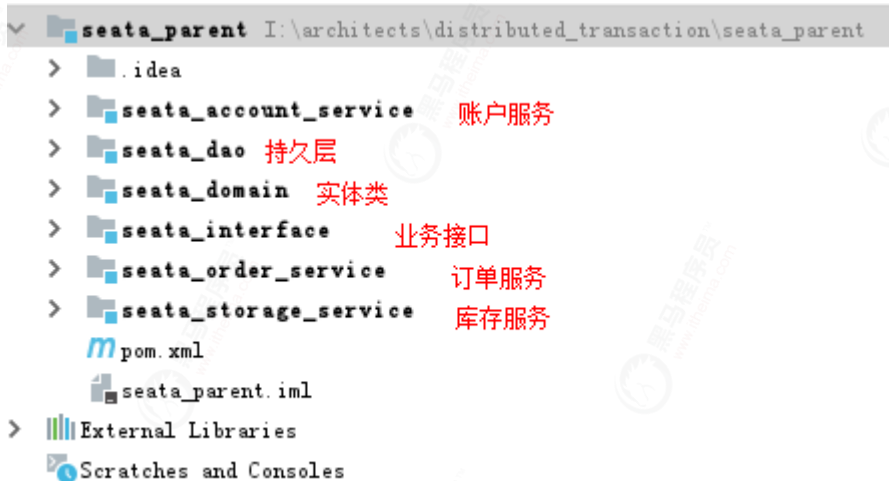
## 2) 提前准备的工程

名称 ▲

-  持久层
-  订单服务
-  父工程坐标
-  库存服务
-  实体类
-  业务层接口
-  账户服务

## 1.3 案例准备

### 1.3.1 导入工程



### 1.3.2 完善数据库相关配置

```
#account
driverClassName_account=com.mysql.jdbc.Driver
url_account=jdbc:mysql://127.0.0.1:3306/seata_account?
useUnicode=true&characterEncoding=utf8&autoReconnect=true
username_account=root
password_account=1234

#item
driverClassName_item=com.mysql.jdbc.Driver
url_item=jdbc:mysql://127.0.0.1:3306/seata_storage?
useUnicode=true&characterEncoding=utf8&autoReconnect=true
username_item=root
password_item=1234

#order
driverClassName_order=com.mysql.jdbc.Driver
url_order=jdbc:mysql://127.0.0.1:3306/seata_order?
useUnicode=true&characterEncoding=utf8&autoReconnect=true
username_order=root
password_order=1234
```

### 1.3.3 完善dubbo的配置

在账户，库存和订单服务的配置文件中提供如下配置，下面以账户服务为例：

```
<!--注册dubbo服务名字-->
<dubbo:application name="seata-account" />
<!--注册中心-->
<dubbo:registry address="zookeeper://127.0.0.1:2181" />
<!--提供服务端口-->
<dubbo:protocol name="dubbo" port="20881" /><!--port每个服务需要提供不同的-->
<dubbo:annotation package="com.itheima" />
```

### 1.3.4 准备seata-server

#### 1) 官网下载与安装

前往[下载地址](#)页面，下载最新的seata-server，如下图所示：

|                                                                                                                             |         |
|-----------------------------------------------------------------------------------------------------------------------------|---------|
| ▼ Assets 4                                                                                                                  |         |
|  <a href="#">seata-server-1.2.0.tar.gz</a> | 39.4 MB |
|  <a href="#">seata-server-1.2.0.zip</a>    | 39.4 MB |
|  <a href="#">Source code (zip)</a>         |         |
|  <a href="#">Source code (tar.gz)</a>      |         |

#### 2) 修改配置与启动

第一步：打开解压好的seata-server目录，进入conf目录，找到file.conf，按照下图修改：



```

## transaction log store, only used in seata-server
store {
    ## store mode: file、db
    mode = "db"

    ## file store property
    file {
        ## store location dir
        dir = "sessionStore"
        # branch session size , if exceeded first try compress lockkey, still exceeded throws exceptions
        maxBranchSessionSize = 16384
        # globe session size , if exceeded throws exceptions
        maxGlobalSessionSize = 512
        # file buffer size , if exceeded allocate new buffer
        fileWriteBufferCacheSize = 16384
        # when recover batch read size
        sessionReloadReadSize = 100
        # async, sync
        flushDiskMode = async
    }

    ## database store property
    db {
        ## the implement of javax.sql.DataSource, such as DruidDataSource(druid)/BasicDataSource(dbcp) etc.
        datasource = "druid"
        ## mysql/oracle/postgresql/h2/oceanbase etc.
        dbType = "mysql"
        driverClassName = "com.mysql.jdbc.Driver"
        url = "jdbc:mysql://127.0.0.1:3306/seata"
        user = "root"
        password = "1234"
        minConn = 5
        maxConn = 30
        globalTable = "global_table"
        branchTable = "branch_table"
        lockTable = "lock_table"
        queryLimit = 100
        maxWait = 5000
    }
}

```

第二步：前往[sql语句网址](#)下载三张表的SQL，在资料中也提供给同学们了。进入mysql创建seata数据库，并执行SQL语句建表。

```

----- The script used when storeMode is 'db' -----
-----
-- the table to store GlobalSession data
CREATE TABLE IF NOT EXISTS `global_table`
(
    `xid`                                VARCHAR(128) NOT NULL,
    `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,
    `gmt_modified`                       DATETIME,
    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,

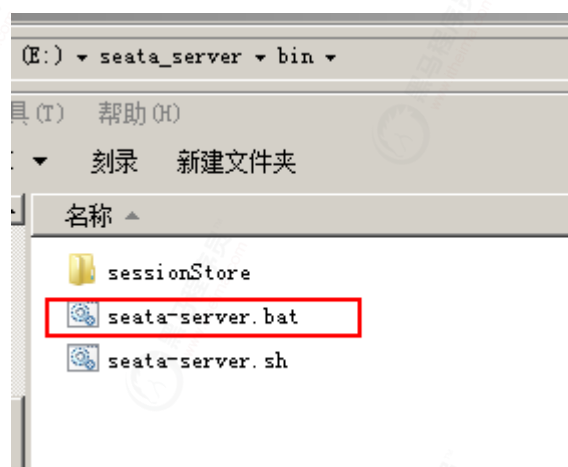
```

```
PRIMARY KEY (`row_key`),  
KEY `idx_branch_id` (`branch_id`)  
) ENGINE = InnoDB  
DEFAULT CHARSET = utf8;
```

第三步：打开register.conf文件，根据实际情况修改里面的配置，它里面提供了注册中心和配置中心的可选项，如下图所示：

```
registry {  
    # file 、nacos 、eureka、redis、zk、consul、etcd3、sofa  
    type = "file"  
  
    config {  
        # file、nacos 、apollo、zk、consul、etcd3  
        type = "file"  
    }  
}
```

第四步：运行下图中的bat文件，启动服务。



## 1.4 案例实现

### 1.4.1 导入seata坐标

```
<dependency>  
    <groupId>io.seata</groupId>  
    <artifactId>seata-all</artifactId>  
    <version>1.2.0</version>  
</dependency>
```

### 1.4.2 配置seata代理数据源

在账户，订单和库存服务的spring-dubbo.xml中配置如下内容：

```
<!--代理数据源-->
<bean id="dataSourceProxy" class="io.seata.rm.datasource.DataSourceProxy">
    <!--注入当前(被代理的)数据源-->
    <constructor-arg ref="dataSource" />
</bean>
```

### 1.4.3 改造SqlSessionFactoryBean

```
<!-- SqlSessionFactoryBean -->
<bean id="sqlSessionFactoryBean" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSourceProxy" />
</bean>
```

### 1.4.4 改造MapperScannerConfigurer

```
<!--
    配置接口扫描包
    如果是单数据源，则可以不配置sqlSessionFactoryBeanName
-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"
    p:basePackage="com.itheima.mapper"
    p:sqlSessionFactoryBeanName="sqlSessionFactoryBean" />
```

注意：指定sqlSessionFactoryBeanname在配置多数据源时必不可少

### 1.4.5 配置TM和TC的通讯协议

```

transport {
    # tcp udt unix-domain-socket
    type = "TCP"
    #NIO NATIVE
    server = "NIO"
    #enable heartbeat
    heartbeat = true
    #thread factory for netty
    thread-factory {
        boss-thread-prefix = "NettyBoss"
        worker-thread-prefix = "NettyServerNIOWorker"
        server-executor-thread-prefix = "NettyServerBizHandler"
        share-boss-worker = false
        client-selector-thread-prefix = "NettyClientSelector"
        client-selector-thread-size = 1
        client-worker-thread-prefix = "NettyClientWorkerThread"
        # netty boss thread size,will not be used for UDT
        boss-thread-size = 1
        #auto default pin or 8
        worker-thread-size = 8
    }
}
service {
    #vgroup->rgroup
    vgroup_mapping.my_test_tx_group = "localRgroup"
    #only support single node
    localRgroup.grouplist = "127.0.0.1:8091"
    #degrade current not support
    enableDegrade = false
    #disable
    disable = false
}

```

## 1.4.6 配置@GlobalTransaction注解（引入seata全局事务管理）

### 1) 配置全局事务扫描器

```

<!--注入全局事务管理器-->
<bean class="io.seata.spring.annotation.GlobalTransactionScanner">
    <constructor-arg value="seata-account"/>
    <constructor-arg value="my_test_tx_group"/>
</bean>

```

### 2) 改造订单服务的OrderInfoService具体实现

```

@Service
public class OrderInfoServiceImpl implements OrderInfoService {

    @Autowired
    private OrderInfoDao orderInfoDao;

    @Reference(check = false)
    private ItemService itemService;

    /**
     * 创建订单
     * 调用ItemService修改库存(调用AccountService修改余额)
     *
     * @param usernumber:购买商品的用户
     * @param id: 购买的商品ID
     * @param count:要减的数量
     */
    @GlobalTransactional(name="fescar-itheima-tx")
    @Override
    public int create(String usernumber, String id, Integer count){
        //从数据库查询商品信息
        Item item = new Item();
        item.setId(id);
        item.setNum(count);
        item.setPrice(100L);
        item.setTitle("华为荣耀4");

        //创建订单
        OrderInfo orderInfo = new OrderInfo();
        String orderId = UUID.randomUUID().toString().replace("-", "").toUpperCase();
        orderInfo.setId(orderId);
        orderInfo.setMoney(item.getPrice()*count);
        orderInfo.setCreatetime(new Date());
        orderInfo.setUsernumber(usernumber);
        int account = orderInfoDao.add(orderInfo);

        System.out.println("添加订单受影响行数: "+account);

        //制造异常
        //      System.out.println("订单添加成功后,出现异常。。。");
        //      int q=10/0;

        //调用ItemService(远程调用)
        Account account = new Account();
        account.setUsernumber(usernumber);
        account.setMoney(item.getPrice()*count); //花掉的钱
        itemService.update(item,account);

        //制造异常
        System.out.println("开始报错了。。。");
        int q=10/0;
        return account;
    }
}

```

```
}
```

### 3) 改造ItemService具体实现

```
@Service
public class ItemServcieImpl implements ItemService {

    @Autowired
    private ItemDao itemDao;

    @Reference(check = false)
    private AccountService accountService;

    /**
     * 修改商品库存，同时修改账户余额
     * @param item
     * @param account
     * @return
     */
    @Override
    public int update(Item item, Account account){
        //修改商品的库存
        int mcount = itemDao.update(item);
        System.out.println("更新库存受影响行数: "+mcount);

        //修改账户余额
        int account = accountService.update(account);
        System.out.println("账户更新余额受影响行数: "+account);
        return mcount;
    }
}
```

### 4) 改造AccountService具体实现

```
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    /**
     * 修改账户余额
     * @param account
     * @return
     */
    @Override
    public int update(Account account){
        return accountDao.update(account);
    }
}
```

### 1.4.7 编写测试代码

```
public class AccountTest {

    /**
     * 启动服务
     */
    @Test
    public void testAccount() throws IOException {
        ApplicationContext act = new ClassPathXmlApplicationContext("spring/spring-dubbo.xml");

        //阻塞线程
        System.in.read();
    }
}
```

```
public class OrderInfoTest {

    /**
     * 业务测试
     */
    @Test
    public void testOrder() throws IOException {
        ApplicationContext act = new ClassPathXmlApplicationContext("spring/spring-dubbo.xml");
        OrderInfoService orderInfoService = act.getBean(OrderInfoService.class);
        orderInfoService.create("itheima", "1", 1);
    }
}
```

```
public class ItemTest {

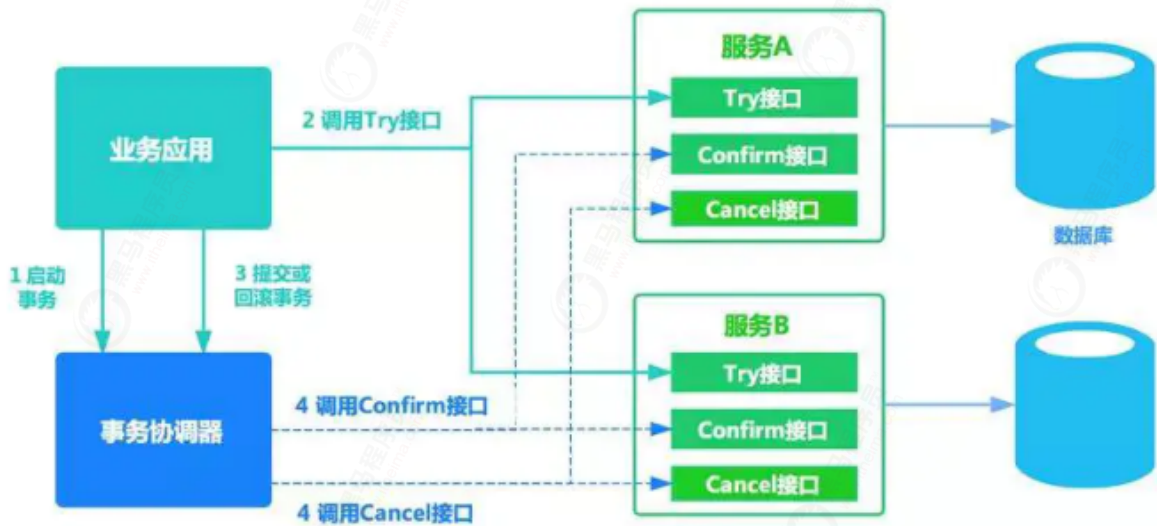
    /**
     * 启动服务
     */
    @Test
    public void testAccount() throws IOException {
        ApplicationContext act = new ClassPathXmlApplicationContext("spring/spring-dubbo.xml");

        //阻塞线程
        System.in.read();
    }
}
```

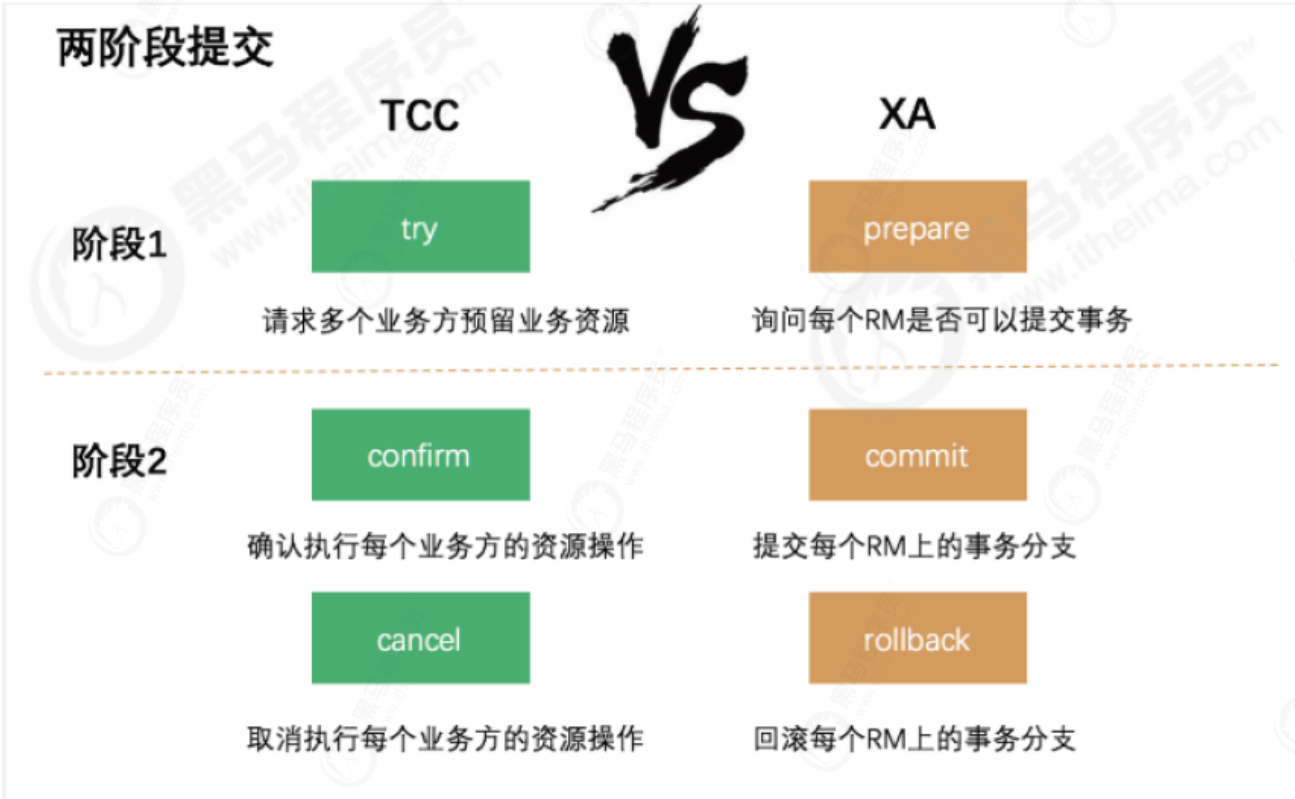
## 2 分布式事务实战-seata的TCC模式

## 2.1 TCC模式详述

TCC模式又称为补偿型分布式事务解决方案，它是通过开发者自己编写补偿代码来实现事务各个组成部分的最终一致性的。其原理图大致如下：

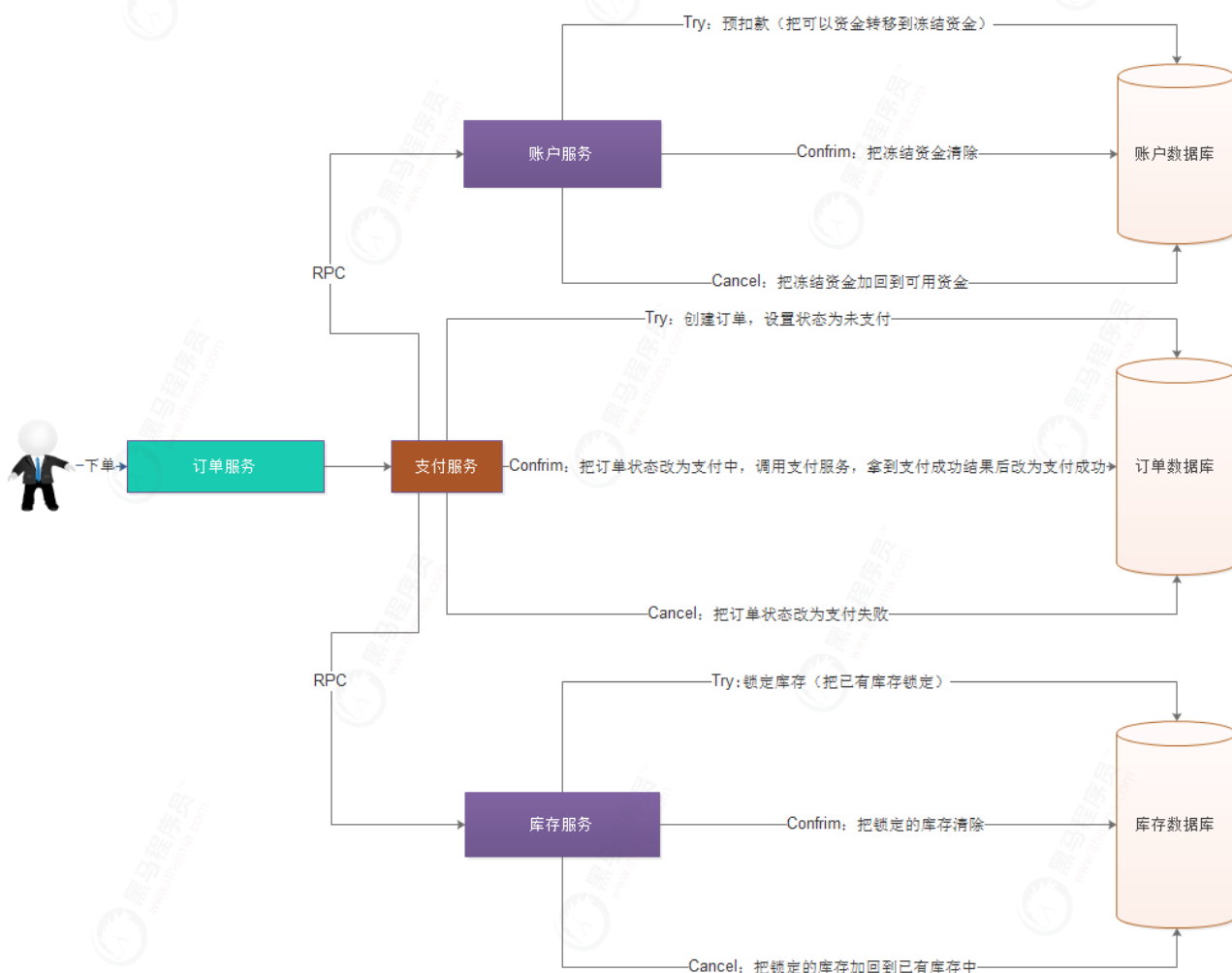


通过上图我们发现，它也是分为两个阶段执行的，和我们前面介绍的2PC方式非常类型，但是他们还是有一些区别的，下图在描述了它和二阶段提交的区别：

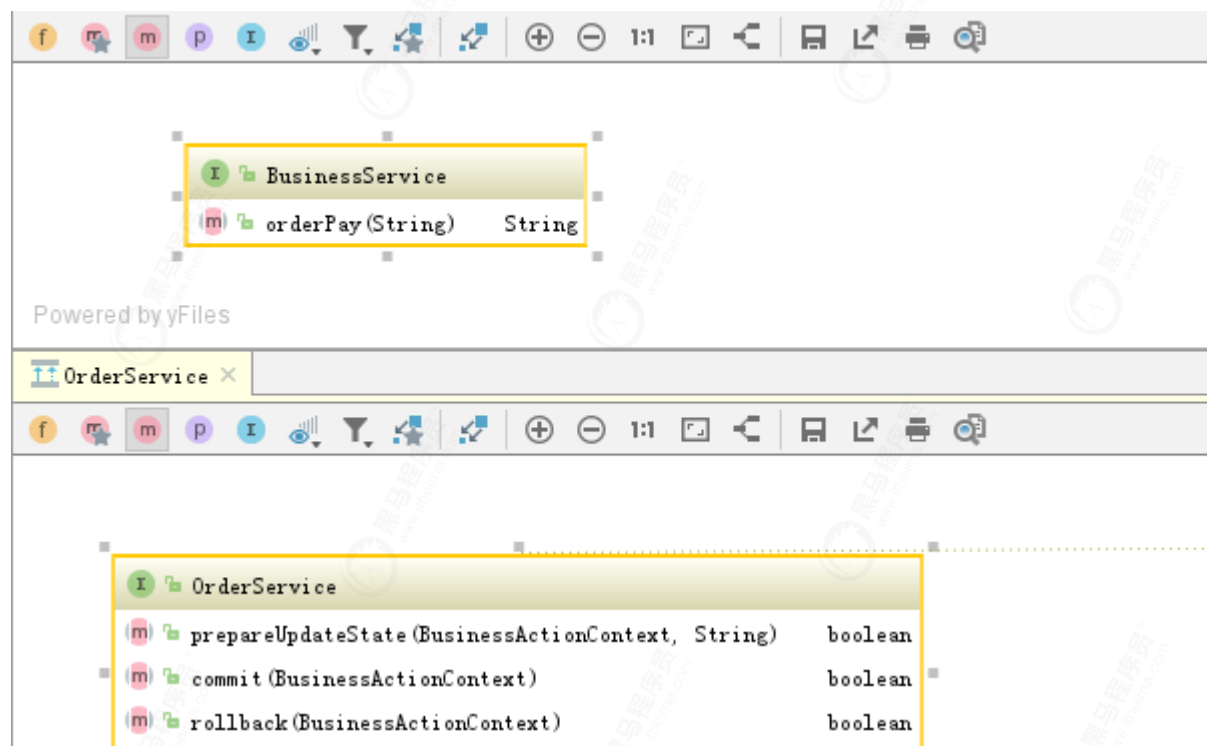


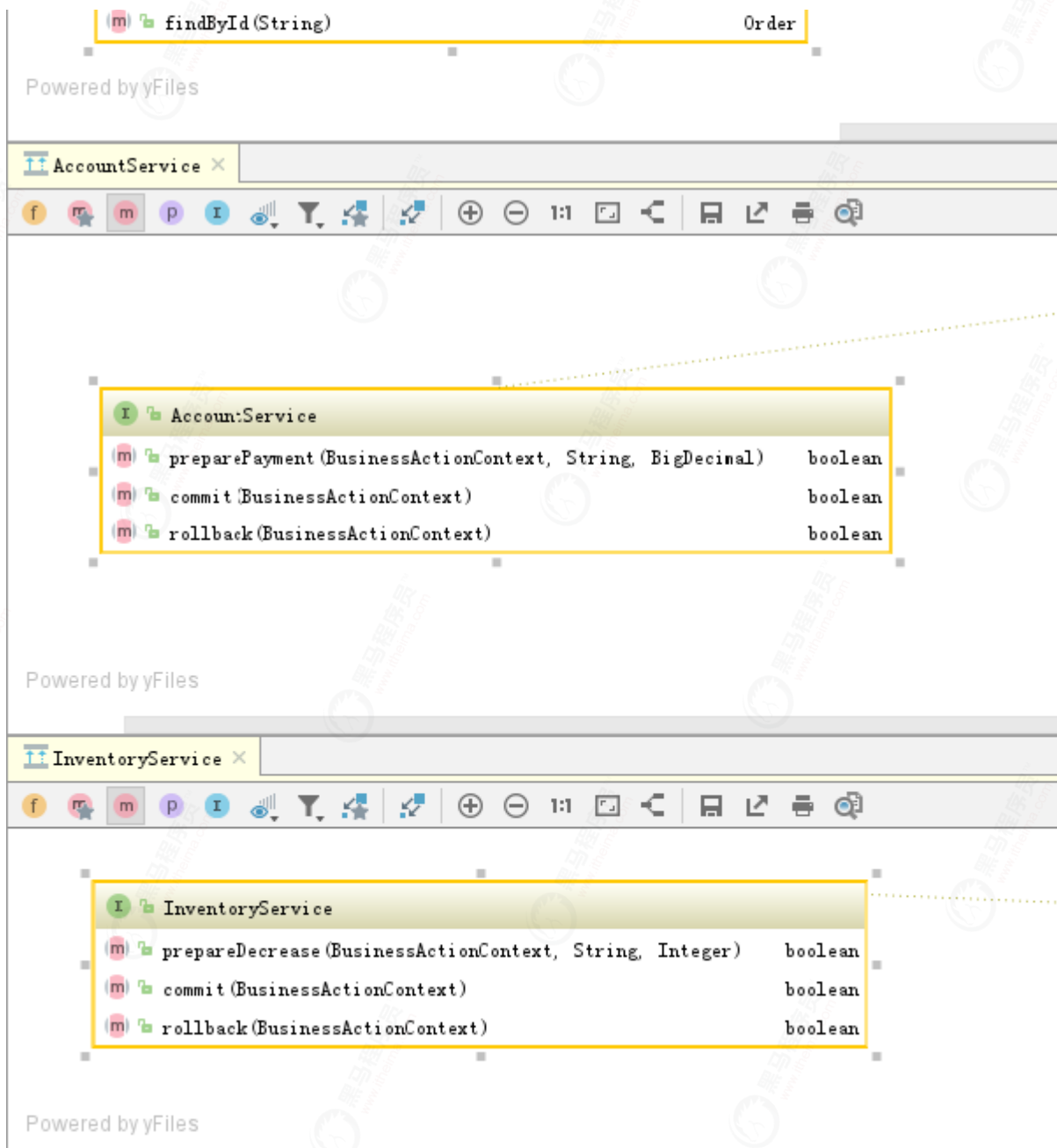
## 2.2 案例简介

## 2.2.1 场景说明



## 2.2.2 功能说明





## 2.2.3 项目简介

### 1) 数据库表结构

```

SET FOREIGN_KEY_CHECKS=0;

-----
-- Table structure for `branch_table`
-----

DROP TABLE IF EXISTS `branch_table`;
CREATE TABLE `branch_table` (
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(128) NOT NULL,
  `transaction_id` bigint(20) DEFAULT NULL,
  `resource_group_id` varchar(32) DEFAULT NULL,
  `resource_id` varchar(256) DEFAULT NULL,
  `branch_type` varchar(8) DEFAULT NULL,
  `status` tinyint(4) DEFAULT NULL,
  `client_id` varchar(64) DEFAULT NULL,
  `application_data` varchar(2000) DEFAULT NULL,
  `gmt_create` datetime DEFAULT NULL,
  `gmt_modified` datetime DEFAULT NULL,
  PRIMARY KEY (`branch_id`),
  KEY `idx_xid` (`xid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-----

-- Records of branch_table
-----

-----
-- Table structure for `global_table`
-----

DROP TABLE IF EXISTS `global_table`;
CREATE TABLE `global_table` (
  `xid` varchar(128) NOT NULL,
  `transaction_id` bigint(20) DEFAULT NULL,
  `status` tinyint(4) NOT NULL,
  `application_id` varchar(32) DEFAULT NULL,
  `transaction_service_group` varchar(32) DEFAULT NULL,
  `transaction_name` varchar(128) DEFAULT NULL,
  `timeout` int(11) DEFAULT NULL,
  `begin_time` bigint(20) DEFAULT NULL,
  `application_data` varchar(2000) DEFAULT NULL,
  `gmt_create` datetime DEFAULT NULL,
  `gmt_modified` datetime DEFAULT NULL,
  PRIMARY KEY (`xid`),
  KEY `idx_gmt_modified_status` (`gmt_modified`,`status`),
  KEY `idx_transaction_id` (`transaction_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-----

-- Records of global_table
-----

-----
-- Table structure for `lock_table`
-----

```

```
DROP TABLE IF EXISTS `lock_table`;
CREATE TABLE `lock_table` (
  `row_key` varchar(128) NOT NULL,
  `xid` varchar(96) DEFAULT NULL,
  `transaction_id` bigint(20) DEFAULT NULL,
  `branch_id` bigint(20) NOT NULL,
  `resource_id` varchar(256) DEFAULT NULL,
  `table_name` varchar(32) DEFAULT NULL,
  `pk` varchar(36) DEFAULT NULL,
  `gmt_create` datetime DEFAULT NULL,
  `gmt_modified` datetime DEFAULT NULL,
  PRIMARY KEY (`row_key`),
  KEY `idx_branch_id` (`branch_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
-----
-- Records of lock_table
-----
```

```

SET FOREIGN_KEY_CHECKS=0;

-----
-- Table structure for `inventory`
-----

DROP TABLE IF EXISTS `inventory`;
CREATE TABLE `inventory` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `product_id` varchar(128) COLLATE utf8mb4_bin NOT NULL,
  `total_inventory` int(10) NOT NULL COMMENT '总库存',
  `lock_inventory` int(10) NOT NULL COMMENT '锁定库存',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

-----
-- Records of inventory
-----

INSERT INTO `inventory` VALUES ('1', '1', '983', '0');

-----
-- Table structure for `undo_log`
-----

DROP TABLE IF EXISTS `undo_log`;
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `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,
  `context` varchar(1000) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_unionkey` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=utf8;

-----
-- Records of undo_log
-----

```

```

SET FOREIGN_KEY_CHECKS=0;

-----
-- Table structure for `account`
-----

DROP TABLE IF EXISTS `account`;
CREATE TABLE `account` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `user_id` varchar(128) COLLATE utf8mb4_bin NOT NULL,
  `balance` decimal(10,0) NOT NULL COMMENT '用户余额',
  `freeze_amount` decimal(10,0) NOT NULL COMMENT '冻结金额，扣款暂存余额',
  `create_time` datetime NOT NULL,
  `update_time` datetime DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

-----
-- Records of account
-----

INSERT INTO `account` VALUES ('1', '10000', '983', '0', '2017-09-18 14:54:22', '2020-07-21 11:22:46');

-----
-- Table structure for `undo_log`
-----

DROP TABLE IF EXISTS `undo_log`;
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `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,
  `context` varchar(1000) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_unionkey` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=utf8;

-----
-- Records of undo_log
-----

```

```

SET FOREIGN_KEY_CHECKS=0;

-----
-- Table structure for `order`
-----

DROP TABLE IF EXISTS `order`;
CREATE TABLE `order` (
  `id` varchar(100) COLLATE utf8mb4_bin NOT NULL,
  `create_time` datetime NOT NULL,
  `number` varchar(20) COLLATE utf8mb4_bin NOT NULL,
  `status` tinyint(4) NOT NULL,
  `product_id` varchar(128) COLLATE utf8mb4_bin NOT NULL,
  `total_amount` decimal(10,0) NOT NULL,
  `count` int(4) NOT NULL,
  `user_id` varchar(128) COLLATE utf8mb4_bin NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

-----
-- Records of order
-----

INSERT INTO `order` VALUES ('1', '2020-07-15 15:47:21', '1283307146968535040', '3', '1', '1', '1', '10000');

-----
-- Table structure for `undo_log`
-----

DROP TABLE IF EXISTS `undo_log`;
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `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,
  `context` varchar(1000) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_unionkey` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

-----
-- Records of undo_log
-----

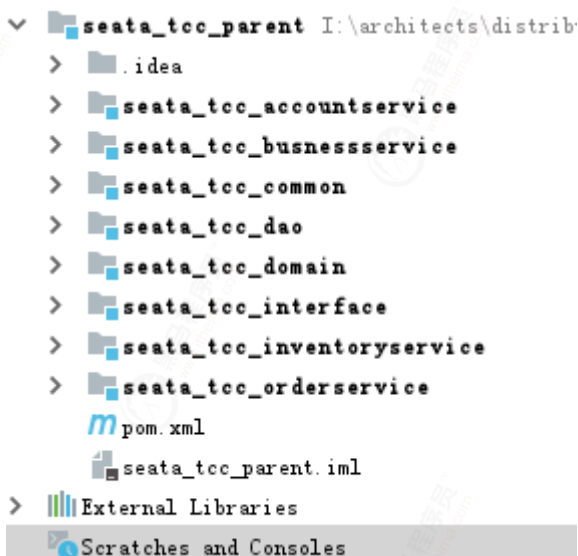
```

## 2) 提前准备的工程

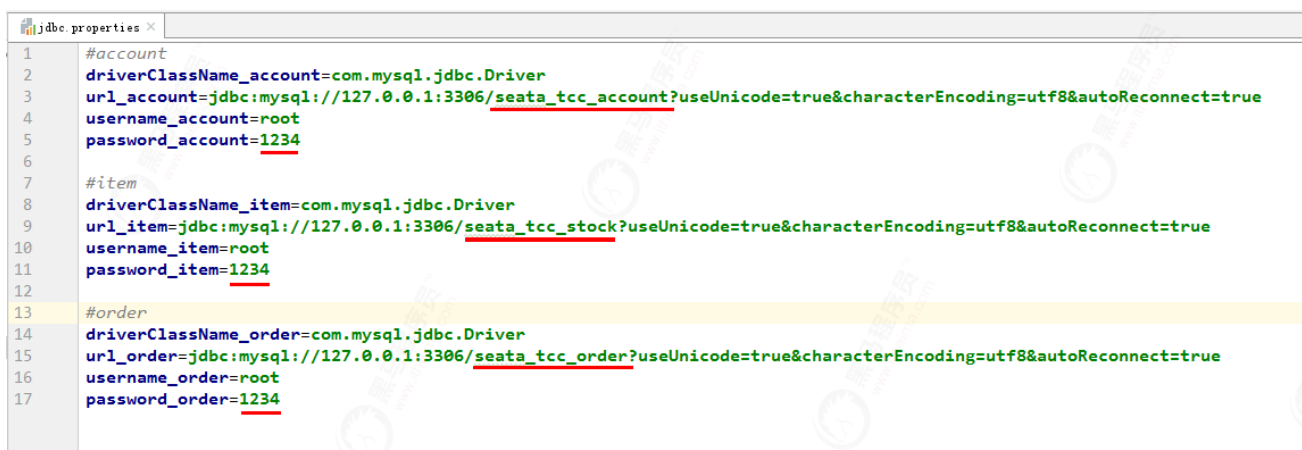
|                            |       |        |      |
|----------------------------|-------|--------|------|
| seata_tcc_accountservice   | 账户服务  | 文件夹    |      |
| seata_tcc_businessservice  | 外部服务  | 文件夹    |      |
| seata_tcc_common           | 公共模块  | 文件夹    |      |
| seata_tcc_dao              | 持久层   | 文件夹    |      |
| seata_tcc_domain           | 实体类   | 文件夹    |      |
| seata_tcc_interface        | 通用接口  | 文件夹    |      |
| seata_tcc_inventoryservice | 库存服务  | 文件夹    |      |
| seata_tcc_orderservice     | 订单服务  | 文件夹    |      |
| pom.xml                    | 父工程依赖 | XML 文件 | 6 KB |

## 2.3 案例准备

### 2.3.1 导入工程



### 2.3.2 完善数据库配置



### 2.3.3 完善dubbo配置

```

43
44 <dubbo:application name="seata-tcc-orderservice"/>
45
46
47 <dubbo:registry protocol="zookeeper" address="localhost:2181"/>
48
49 <dubbo:protocol name="dubbo" port="20886"
50     server="netty" client="netty"
51     charset="UTF-8" threadpool="fixed" threads="500"
52     queues="0" buffer="8192" accepts="0" payload="8388608"/>
53
54 <dubbo:annotation package="com.itheima.service"/>
55
56 <!--注入全局事务管理器-->

```

seata\_tcc\_inventoryservice\...\applicationContext.xml ×

```

39
40
41 <dubbo:application name="seata-tcc-inventoryservice"/>
42
43 <dubbo:registry protocol="zookeeper" address="localhost:2181"/>
44
45 <dubbo:protocol name="dubbo" port="20885"
46     server="netty"
47     charset="UTF-8" threadpool="fixed" threads="500"
48     queues="0" buffer="8192" accepts="0" payload="8388608" />
49 <dubbo:annotation package="com.itheima.service"/>
50

```

beans > dubbo:protocol

seata\_tcc\_accountservice\...\applicationContext.xml ×

```

39
40 <dubbo:application name="seata-tcc-accountservice"/>
41
42 <dubbo:registry protocol="zookeeper" address="localhost:2181"/>
43
44 <dubbo:protocol name="dubbo" port="20884"
45     server="netty" client="netty"
46     charset="UTF-8" threadpool="fixed" threads="500"
47     queues="0" buffer="8192" accepts="0" payload="8388608" />
48
49 <dubbo:annotation package="com.itheima.service"/>
50
51

```

```

<dubbo:application name="seata-tcc-businessservice"/>

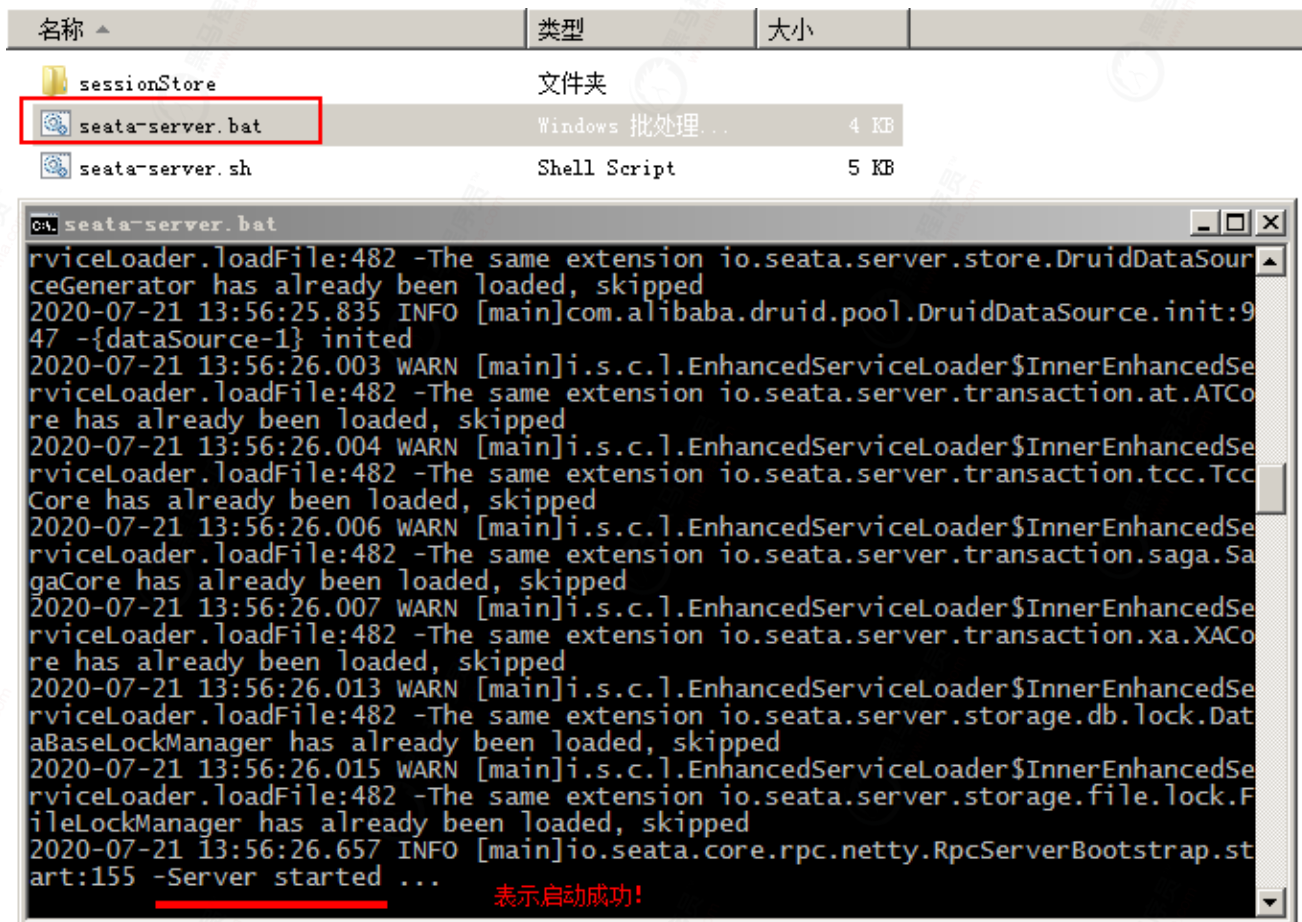
<dubbo:registry protocol="zookeeper" address="localhost:2181"/>

<dubbo:protocol name="dubbo" port="20887"
    server="netty" client="netty"
    charset="UTF-8" threadpool="fixed" threads="500"
    queues="0" buffer="8192" accepts="0" payload="8388608"/>

<dubbo:annotation package="com.itheima.service"/>

```

## 2.3.4 启动seata-server



## 2.4 案例实现

### 2.4.1 seata的前期准备

#### 1) 导入坐标

```
<!-- https://mvnrepository.com/artifact/io.seata/seata-all-->
<dependency>
  <groupId>io.seata</groupId>
  <artifactId>seata-all</artifactId>
  <version>1.2.0</version>
</dependency>
```

#### 2) 配置seata(三个服务都需要, 此处只以账户服务为例)

```

<!--注入全局事务管理器-->
<bean class="io.seata.spring.annotation.GlobalTransactionScanner">
    <constructor-arg value="seata-tcc-accountservice"/>
    <constructor-arg value="my_test_tx_group"/>
</bean>

<!--代理数据源-->
<bean id="dataSourceProxy" class="io.seata.rm.datasource.DataSourceProxy">
    <!--注入当前(被代理的)数据源-->
    <constructor-arg ref="dataSource" />
</bean>

<!--配置undoLog解析器-->
<bean id="undoLogParser" class="io.seata.rm.datasource.undo.parser.JacksonUndoLogParser">
</bean>

```

3) 改造mybatis的相关配置-引用seata提供的代理数据源(三个服务都需要, 此处只以账户服务为例)

```

<bean id="sqlSessionFactoryBean" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSourceProxy"></property>
    <property name="typeAliasesPackage" value="com.itheima.domain"></property>
</bean>

```

## 2.4.2 配置@TwoPhaseBusinessAction注解

```
@LocalTCC
public interface InventoryService {

    /**
     * 预扣减库存操作
     * @param businessActionContext
     * @param productId 商品id
     * @param count 商品数量
     * @return
     */
    @TwoPhaseBusinessAction(name="decreaseTccTryPhase",commitMethod = "commit",rollbackMethod =
"rollback")
    boolean prepareDecrease(BusinessActionContext businessActionContext,
        @BusinessActionContextParameter(paramName = "productId")String
productId,
        @BusinessActionContextParameter(paramName = "count")Integer count);

    /**
     * 扣减库存的提交补偿
     * @param businessActionContext
     * @return
     */
    public boolean commit(BusinessActionContext businessActionContext) ;

    /**
     * 扣减库存的回滚补偿
     * @param businessActionContext
     * @return
     */
    public boolean rollback(BusinessActionContext businessActionContext) ;
}
```

@LocalTCC

public interface AccountService {

/\*\*

\* 扣款支付

\* @param businessActionContext

\* @param userId 支付人id

\* @param amount 支付金额

\* @return

\*/

@TwoPhaseBusinessAction(name = "accountTccTryPhase", commitMethod = "commit", rollbackMethod = "rollback")

boolean preparePayment(BusinessActionContext businessActionContext,

@BusinessActionContextParameter(paramName = "userId")String userId,

@BusinessActionContextParameter(paramName = "amount")BigDecimal amount);

/\*\*

\* 提交方法

\* @param businessActionContext

\* @return

\*/

public boolean commit(BusinessActionContext businessActionContext);

/\*\*

\* 取消方法

\* @param businessActionContext

\* @return

\*/

public boolean rollback(BusinessActionContext businessActionContext);

}

```

@LocalTCC
public interface OrderService {

    /**
     * 创建订单并且进行扣除账户余额支付，并进行库存扣减操作
     *
     * @return string string
     */
    @TwoPhaseBusinessAction(name = "orderPayTccTryPhase",commitMethod = "commit",rollbackMethod
    = "rollback")
    boolean prepareUpdateState(BusinessActionContext
    businessActionContext,@BusinessActionContextParameter(paramName = "orderId")String orderId);

    /**
     * 修改订单状态为支付完成
     * @param businessActionContext
     */
    public boolean commit(BusinessActionContext businessActionContext);

    /**
     * 修改订单状态为支付失败
     * @param businessActionContext
     */
    public boolean rollback(BusinessActionContext businessActionContext);

    /**
     * 根据id查询订单
     * @param id
     * @return
     */
    Order findById(String id);
}

```

### 2.4.3 配置@GlobalTransactional注解

```

/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
public interface BusinessService {

    /**
     * 外部服务：下单
     * @param orderId 订单Id
     * @return
     */
    String orderPay(String orderId);
}

```

```

/**
 * @author 黑马程序员
 * @Company http://www.itheima.com
 */
@Service
public class BusinessServiceImpl implements BusinessService {

    @Reference
    private OrderService orderService;

    @Reference
    private InventoryService inventoryService;

    @Reference
    private AccountService accountService;

    @Override
    @GlobalTransactional(name = "seata-tcc-tx",rollbackFor = Exception.class,propagation =
Propagation.REQUIRED)
    public String orderPay(String orderId){
        String xid = RootContext.getXID();
        System.out.println(xid);
        //1.根据订单id查询订单
        Order order = orderService.findById(orderId);
        //2.判断是否有此订单
        if(order != null && order.getStatus().intValue() == OrderStatusEnum.NOT_PAY.getCode()) {
            //3.更新订单状态为支付中
            orderService.prepareUpdateState(null, orderId);
            //4.预扣减余额（冻结账户资金）
            accountService.preparePayment(null,order.getUserId(),order.getTotalAmount());
            //5.预扣减库存（锁定库存）
            inventoryService.prepareDecrease(null,order.getProductId(), order.getCount());
        }else {
            return "订单不存在";
        }
        return "success";
    }
}

```

## 2.5 seat的源码分析

### 2.5.1 初始化-解析@TwoPhaseBusinessAction注解

在程序加载时，解析配置文件中提供的 `GlobalTransactionScanner` 配置，该类是 `AbstractAutoProxyCreator` 的子类，里面重写了 `wrapIfNecessary` 方法。该方法里面调用了 `TCCBeanParserUtils` 类中的静态方法 `isTccAutoProxy`，而此方法内部执行了解析注解的操作方法 `parserRemotingServiceInfo`，此时它会把具有 `@TwoPhaseBusinessAction` 注解的方法解析出来，并创建成 `TccResource` 对象。

```

/**
 * parse the remoting bean info
 *
 * @param bean the bean
 * @param beanName the bean name
 * @return remoting desc
 */
public RemotingDesc parserRemotingServiceInfo(Object bean, String beanName, RemotingParser remotingParser) {
    RemotingDesc remotingBeanDesc = remotingParser.getServiceDesc(bean, beanName);
    if (remotingBeanDesc == null) {
        return null;
    }
    remotingServiceMap.put(beanName, remotingBeanDesc);

    Class<?> interfaceClass = remotingBeanDesc.getInterfaceClass();
    Method[] methods = interfaceClass.getMethods();
    if (remotingParser.isService(bean, beanName)) {
        try {
            //service bean, registry resource
            Object targetBean = remotingBeanDesc.getTargetBean();
            for (Method m : methods) {
                TwoPhaseBusinessAction twoPhaseBusinessAction = m.getAnnotation(TwoPhaseBusinessAction.class);
                if (twoPhaseBusinessAction != null) {
                    TCCResource tccResource = new TCCResource();
                    tccResource.setActionName(twoPhaseBusinessAction.name());
                    tccResource.setTargetBean(targetBean);
                    tccResource.setPrepareMethod(m);
                    tccResource.setCommitMethod(twoPhaseBusinessAction.commitMethod());
                    tccResource.setCommitMethod(ReflectionUtil
                        .getMethod(interfaceClass, twoPhaseBusinessAction.commitMethod(),
                            new Class[] { BusinessActionContext.class }));
                    tccResource.setRollbackMethodName(twoPhaseBusinessAction.rollbackMethod());
                    tccResource.setRollbackMethod(ReflectionUtil
                        .getMethod(interfaceClass, twoPhaseBusinessAction.rollbackMethod(),
                            new Class[] { BusinessActionContext.class }));
                    //registry tcc resource
                    DefaultResourceManager.get().registerResource(tccResource);
                }
            }
        } catch (Throwable t) {
            throw new FrameworkException(t, "parser remoting service error");
        }
    }
    if (remotingParser.isReference(bean, beanName)) {
        //reference bean, TCC proxy
        remotingBeanDesc.setReference(true);
    }
    return remotingBeanDesc;
}

```

## 2.5.2 资源管理器注册资源

TCCResourceManager 负责管理 TCC 模式下资源的注册、分支的注册、提交、和回滚。代码如下：

```

@Override
public void registerResource(Resource resource) {
    TCCResource tccResource = (TCCResource)resource;
    tccResourceCache.put(tccResource.getResourceId(), tccResource);
    super.registerResource(tccResource);
}

```

而与server通信的逻辑被封装在了父类 AbstractResourceManage 中，这里根据 resourceId 对 TCCResource 进行缓存。父类 AbstractResourceManage 注册资源的时候，使用 resourceGroupId + actionName，actionName 就是 @TwoPhaseBusinessAction 注解中的 name，resourceGroupId 默认是 DEFAULT。

同时，在 TccResourceManager 中还定义了分支事务的提交和回滚的方法。如下图所示：

```

/**
 * TCC branch commit
 *
 * @param branchType
 * @param xid Transaction id.
 * @param branchId Branch id.
 * @param resourceId Resource id.
 * @param applicationData Application data bind with this branch.
 * @return
 * @throws TransactionException
 */
@Override
public BranchStatus branchCommit(BranchType branchType, String xid, long branchId, String resourceId,
String applicationData) throws TransactionException {...}

/**
 * TCC branch rollback
 *
 * @param branchType the branch type
 * @param xid Transaction id.
 * @param branchId Branch id.
 * @param resourceId Resource id.
 * @param applicationData Application data bind with this branch.
 * @return
 * @throws TransactionException
 */
@Override
public BranchStatus branchRollback(BranchType branchType, String xid, long branchId, String resourceId,
String applicationData) throws TransactionException {...}

/**

```

### 2.5.3 事务控制的方法拦截器

seata的tcc模式事务控制也是基于Spring的AOP实现的，它里面也是利用了 `MethodInterceptor` 方法拦截器来进行的增强，它里面提供了一个名称为 `TccActionInterceptor` 的类，该类实现了 `MethodInterceptor` 接口，并重写了 `invoke` 方法。代码如下：

```

@Override
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) {
        //save the xid
        String xid = RootContext.getXID();
        //clear the context
        RootContext.unbind();
        RootContext.bindInterceptorType(xid, BranchType.TCC);
        try {
            Object[] methodArgs = invocation.getArguments();
            //Handler the TCC Aspect
            Map<String, Object> ret = actionInterceptorHandler.proceed(method, methodArgs,
xid, businessAction,
                invocation::proceed);
            //return the final result
            return ret.get(Constants.TCC_METHOD_RESULT);
        } finally {
            //recovery the context
            RootContext.unbindInterceptorType();
            RootContext.bind(xid);
        }
    }
    return invocation.proceed();
}

```

而此方法的核心是 `MethodInvocation` 的 `proceed` 方法，最终由 `ActionInterceptorHandler` 类中的 `proceed` 方法处理。代码如下：

```

/**
 * Handler the TCC Aspect
 *
 * @param method          the method
 * @param arguments       the arguments
 * @param businessAction  the business action
 * @param targetCallback  the target callback
 * @return map map
 * @throws Throwable the throwable
 */
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 anme
    actionContext.setActionName(actionName);

    //Creating Branch Record
    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;
}

```

而该方法中调用了 `doTccActionLogStore()` 方法，该方法干了两件重要的事情：

第一：执行 `fetchActionRequestContext(method, arguments)`，这个方法把被 `@BusinessActionContextParam` 注解的参数取出来，在下面的 `init` 方法中塞入 `BusinessActionComtext`，同时塞入的还有事务相关参数。`DefaultResourceManager.get().branchRegister(BranchType.TCC, actionName, null, xid,applicationContextStr, null)`，这个方法执行 TCC 模式下事务参与者事务分支的注册。

第二：回调执行 `targetCallback.execute()`，被代理的 bean 具体的业务，即 `prepare()` 方法。

## 3 分布式事务解决方案总结

---

### 3.1 TCC方案

---

#### 3.1.1 特点

- 严格一致性
- 执行时间短
- 实时性要求高

#### 3.1.2 适用场景

- 抢红包
- 实时转账汇款
- 收付款

### 3.2 异步确保方案

---

#### 3.2.1 特点

- 实时性不高
- 执行周期较长

#### 3.2.2 适用场景

- 非实时汇款
- 退货退款业务

### 3.3 最大努力通知方案

---

#### 3.3.1 特点

- 高并发低耦合
- 不支持回滚

#### 3.3.2 适用场景

- 获取交易结果（例如：滴滴支付，共享单车支付等等）

### 3.4 基于3种方案再谈秒杀超卖的分布式事务解决

---

#### 3.4.1 超卖现象及产生原因

##### 1) 什么是超卖现象

在高并发下，多个线程并发更新库存，导致库存为负的情况。

##### 2) 产生原因

由于sql支持并行加上事务的隔离性，所以当多个事务并行时，select出来的值并不一定准确的，进而update之后的值也就不正确了。

### 3.4.2 解决思路分析

#### 1) 使用数据库的排他锁方案——悲观锁

此种方案是最简单粗暴的方案。它是利用了select xxx for update造成的排他锁，避免了超卖的问题。

优点：由于排他，所以变成单线程应用，一定可以保证线程安全。

缺点：也正是因为排他，所以高并发场景下会导致多个请求一直等待，数据库性能下降，系统的链接数上升，负载飙升，影响系统的平均响应时间，甚至会瘫痪。

#### 2) 使用数据版本号方案——乐观锁

悲观锁的解决方案解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，系统处理队列内请求的速度根本无法和疯狂涌入队列中的数目相比，很可能一瞬间将队列内存“撑爆”，最终Web系统平均响应时候还是会大幅下降，系统还是陷入异常。

这个时候，我们就可以讨论一下“乐观锁”的思路了。乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。

#### 3) 使用Redis缓存方案

由于数据库的性能问题，无法应对高并发的秒杀场景，所以通常的解决方案是利用缓存来完成，先在缓存中完成计数，然后再通过消息队列异步地入库。redis由于其高速+单进程模型，省掉了很多并发的的问题，所以可以被选来进行高速秒杀的工作。

### 3.4.3 分布式事务选型方案

通过上述分析，秒杀系统是一个非常繁杂且细节很多的分布式系统，它里面不仅包含了事务，还包含了事务解决不了的问题（就是并发访问，且事务隔离级别处于非最高级情况下），此时当出现了大量线程涌入，如果没有完善的补偿机制，那么超卖是必然产生的。由此，我们得出一个非常清晰的结论，就是更加灵活可控的补偿型事务方案更加适合当下场景。

注：请同学们注意下，我们此处所说的是方案，不是特指某个框架。因为补偿型事务不只有Hmily框架可以实现，像Seata，LCN和tcc-transaction都可以。