一、ShardingJDBC分布式事务快速上手

ShardingJDBC支持的分布式事务方式有三种 LOCAL, XA , BASE，这三种事务实现方式都是采用的对代码无侵入的方式实现的。具体见 TransactionTypeHolder.set(TransactionType.XA);

> 这里设置的TransactionType实际上是一个ThreadLocal的线程变量，只针对当前线程有效。并且通常用完之后都要使用TransactionTypeHolder.clear()将设置清除，以免影响线程内其他操作。

LOCAL本地事务

本地事务方式也就是使用Spring的@Transaction注解来进行配置。传统的本地事务是不具备分布式事务特性的，但是ShardingSphere对本地事务进行了增强。在ShardingSphere中，LOCAL本地事务已经完全支持由于逻辑异常导致的分布式事务问题。不过这种本地事务模式IBU支持因网络、硬件导致的跨库事务。例如同一个事务中，跨两个库更新，更新完毕后，提交之前，第一个库宕机了，则只有第二个库数据提交。

XA事务快速上手

这种模式下，是由ShardingJDBC所在的应用来作为事务协调者，通过XA方式来协调分布到多个数据库中的分库分表语句的分布式事务。

在ShardingJDBC的官方文档中，有对分布式事务的几个示例，可以用来参考下：

https://shardingsphere.apache.org/document/legacy/4.x/document/cn/manual/sharding-jdbc/usage/transaction/

简单来说，在SpringBoot中分为以下几个步骤：

1、引入maven依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-jdbc-core</artifactId>
    <version>${sharding-sphere.version}</version>
</dependency>

<!-- 使用XA事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-transaction-xa-core</artifactId>
    <version>${shardingsphere.version}</version>
</dependency>

<!-- 使用XA事务时，可以引入其他几种事务管理器 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-bitronix</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-transaction-xa-narayana</artifactId>
</dependency>
```

> XA是一种分布式事务规范，与之对应的是JAVA平台上的事务规范JTA(Java Transaction Api)。JTA定义了对XA事务的支持，实际上，JTA就是基于XA构建的。但是JTA只是相当于一组结构，定义了分布式事务的处理方式，具体实现还是需要由各个厂商提供。
>
> 目前JTA有两种实现方式，一种是由特定的J2EE容器提供，例如这里提到的 narayana 就是由JBOSS提供的。另一种就是适用于所有J2EE的通用规范，例如Atomokios，他是ShardingSphere默认使用的事务管理器。

2、配置事务管理器

```
@Configuration
@EnableTransactionManagement
public class TransactionConfiguration {

    @Bean
    public PlatformTransactionManager txManager(final DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
    //如果不使用jdbctemplate就可以不注入。
    @Bean
    public JdbcTemplate jdbcTemplate(final DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

> 使用分布式事务管理器的重点是两个地方，一是配置@EnableTransactionManagement注解，启用事务管理；二是注入TransactionManager对象，其中对于这个事务管理器的重点就是要使用ShardingDatasource。

3、在业务代码中使用

```
@Transactional
@ShardingTransactionType(TransactionType.XA)  // 支持TransactionType.LOCAL, TransactionType.XA, TransactionType.BASE
public void insert() {
    jdbcTemplate.execute("INSERT INTO t_order (user_id, status) VALUES (?, ?)", (PreparedStatementCallback<Object>) preparedStatement -> {
        preparedStatement.setObject(1, i);
        preparedStatement.setObject(2, "init");
        preparedStatement.executeUpdate();
    });
}
```

> 使用时的重点是在@ShardingTransactionType注解中声明XA类型的事务。

ShardingSphere默认是使用的Atomikos作为XA事务管理器，在项目中会生成一个xa_tx.log，这个是XA崩溃恢复所需的日志，不要删除。另外，可以在项目的classpath中添加jta.properties来定制Atomikos的配置项。具体配置项参见 https://www.atomikos.com/Documentation/JtaProperties 。

测试案例

我们可以使用第二节中的application01.properties案例来进行简单的测试。 在application01.properties中，配置了逻辑表course的两个实际表course_1和course_2。当执行下面的测试案例时，会将两种表的user_id都一起进行更新。

```
@Test
public void updateCourse(){
    Course c = new Course();
    UpdateWrapper<Course> wrapper = new UpdateWrapper<>();
    wrapper.set("user_id","5");
    courseMapper.update(c,wrapper);
}
```

现在手动给course_2表添加一个user_id字段的唯一索引。这样，再执行这个测试案例时，对于course_2分片的数据就会更新失败。这时我们可以来观察course_1分片的数据，有没有随着整个事务一起回滚。这时要注意给这个测试单元加上事务的注解。

```
@Test
@Transactional
@ShardingTransactionType(TransactionType.XA)
public void updateCourse(){
    Course c = new Course();
    UpdateWrapper<Course> wrapper = new UpdateWrapper<>();
    wrapper.set("user_id","6");
    courseMapper.update(c,wrapper);
}
```

BASE柔性事务快速上手

这种模式，是由Seata作为事务协调者，来进行协调。使用方式需要先部署seata服务。官方建议是使用seata配合nacos作为配置中心来使用。实际上是使用的seata的AT模式进行两阶段提交。

seata部署方式：

nacos： 下载压缩包，解压执行bin目录下的startup指令即可。Demo中是使用的1.4.1版本

```
--以独立方式启动
sh startup.sh -m standalone
```

seata：同样是下载发布包，并解压。Demo中使用1.4.0版本

然后往nacos上初始化配置，这个脚本会在nacos上注册一组Group=SEATA_GROUP 的配置项。

```
sh nacos-config.sh localhost
```

> seata 1.4.0版本中已经没有这个脚本了，所有需要到老版本中去找。
>
> 这个脚本会将conf目录下的config.txt里的配置信息全部推送到目标Nacos上。 这个配置挺多的，有八九十个，而且很容易出错，要非常小心。

接下来修改seata-Server的解压目录下的conf/registry.conf文件，配置seata的注册中心。

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

  nacos {
    application = "seata-server"
    serverAddr = "192.168.65.232:8848"
    namespace = "public"
    group = "SEATA_GROUP"
    cluster = "default"
    #username = "nacos"
    #password = "nacos"
  }
}

config {
  # file、nacos 、apollo、zk、consul、etcd3
  type = "nacos"

  nacos {
     application = "seata-server"
     serverAddr = "192.168.65.232:8848"
     namespace = "29ccf18e-e559-4a01-b5d4-61bad4a89ffd"
```

```
    group = "SEATA_GROUP"
    cluster = "default"
    username = "nacos"
    password = "nacos"
  }
}
```

> 这个配置里，是将seata的服务注册到nacos上，配置也从nacos上获取。registry部分对应seata注册到nacos上的服务。而config部分对应seata注册到nacos上的配置。但是配置信息是要另外手动上传到nacos中的。Seata中有专门的脚本辅助推送配置信息。
>
> serverAddr、username、password分别为nacos的服务地址、用户名(默认nacos)、密码(默认nacos)。group(默认SEATA_GROUP)、namespac(默认public)这两个属性需要跟seata在nacos上的注册情况匹配。

这样就可以启动seata了。 启动成功后，可以在Nacos控制台上看到 服务名=serverAddr服务注册列表

```
sh seata-server.sh -p $LISTEN_PORT -m $STORE_MODE -h $IP(此参数可选)
```

> 其中 **$LISTEN_PORT**: Seata-Server 服务端口。默认8848
> **$STORE_MODE**: 事务操作记录存储模式：file、db。可以在registry.conf文件中配置。
> **$IP(可选参数)**: 用于多 IP 环境下指定 Seata-Server 注册服务的IP。单网卡不需要配置。

最后给nacos发送一个put请求，定制参数

```
curl -X PUT 'localhost:8848/nacos/v1/ns/operator/switches?entry=serverMode&value=AP'
```

客户端使用Base事务

使用BASE柔性事务需要引入maven依赖

```
<!-- 使用BASE事务时，需要引入此模块 -->
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-transaction-base-seata-at</artifactId>
```

```
    <version>${sharding-sphere.version}</version>
</dependency>
<dependency>
    <groupId>io.seata</groupId>
    <artifactId>seata-all</artifactId>
    <version>1.4.0</version>
</dependency>
<dependency>
    <groupId>com.alibaba.nacos</groupId>
    <artifactId>nacos-client</artifactId>
    <version>1.4.1</version>
</dependency>
```

> 特别要注意seata的版本，必须与服务端匹配。 nacos版本与服务端不匹配的话，大部分情况下还不会有问题。但是如果seata的版本不匹配，那会出现很多莫名其妙的问题。

接下来，要使用Seata的AT模式，还需要在每个分片建立一个undo_log表

```
CREATE TABLE IF NOT EXISTS `undo_log`
(
  `id`            BIGINT(20)   NOT NULL AUTO_INCREMENT COMMENT 'increment id',
  `branch_id`     BIGINT(20)   NOT NULL COMMENT 'branch transaction id',
  `xid`           VARCHAR(100) NOT NULL COMMENT 'global transaction id',
  `context`       VARCHAR(128) NOT NULL COMMENT 'undo_log context,such as serialization',
  `rollback_info` LONGBLOB     NOT NULL COMMENT 'rollback info',
  `log_status`    INT(11)      NOT NULL COMMENT '0:normal status,1:defense status',
  `log_created`   DATETIME     NOT NULL COMMENT 'create datetime',
  `log_modified`  DATETIME     NOT NULL COMMENT 'modify datetime',
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`, `branch_id`)
) ENGINE = InnoDB
  AUTO_INCREMENT = 1
  DEFAULT CHARSET = utf8 COMMENT ='AT transaction mode undo table';
```

接下来在classpath下增加seata.conf。ShardingSphere的SeataATShardingTransactionManager会读取这个配置文件。

```
client {
    application.id = example    ## 应用唯一id
    transaction.service.group = my_test_tx_group    ## 所属事务组
}
```

注意配置时，application.id可以随意配置，但是transaction.service.group这个事务组不能随意配，需要在server端进行配置。对应service.vgroupMapping.my_test_tx_group key =default 这个key中的后面一部分。

> 注意seata下的事务组配置：service.vgroupMapping.my_test_tx_group = default，其中这个my_test_tx_group 就是配置的事务组。这个事务组相当于是一个多租户的概念，不同的事务组之间的配置信息是隔离的。
>
> 然后后面的default对应的是Seata中的TC集群名。默认就是default。而这个TC集群中有哪些服务节点是要另外配置的。service.default.gouplist = 127.0.0.1:8091 这个配置中就配置了default这个集群中对应的节点列表。这些节点就会加入到同一个分布式事务中。

然后，还需要将服务端的registry.conf文件也复制到classpath目录下。也就是需要与服务端匹配。

最后使用的方式和XA基本是一样的，在声明@ShardingTransactionType注解时声明成BASE类型的就可以了。

Demo中提供了JUnit测试案例：TransactionTest

> 柔性事务使用的难点还是在seata上。用起来要非常小心。

二、分布式事务原理详解

快速上手，熟悉ShardingSphere的分布式事务处理方式后，我们再来深入理解下ShardingSphere涉及到的分布式事务。

XA事务

XA是由X/Open组织提出的分布式事务的规范。 主流的关系型 数据库产品都是实现了XA接口的。 例如在MySQL从5.0.3版本开始，就已经可以直接支持XA事务了，但是要注意只有InnoDB引擎才提供支持。

```
//1、 XA START|BEGIN 开启事务，这个test就相当于是事务ID，将事务置于ACTIVE状态
XA START 'test';
//2、对一个ACTIVE状态的XA事务，执行构成事务的SQL语句。
    insert...//business sql
```

```
//3、发布一个XA END指令，将事务置于IDLE状态
XA END 'test'; //事务结束
//4、对于IDLE状态的XACT事务，执行XA PREPARED指令 将事务置于PREPARED状态。
//也可以执行 XA COMMIT 'test' ON PHASE 将预备和提交一起操作。
XA PREPARE 'test'; //准备事务
//PREPARED状态的事务可以用XA RECOVER指令列出。列出的事务ID会包含
gtrid,bqual,formatID和data四个字段。
XA RECOVER;
//5、对于PREPARED状态的XA事务，可以进行提交或者回滚。
XA COMMIT 'test'; //提交事务
XA ROLLBACK 'test'; //回滚事务。
```

XA事务中，事务都是有状态控制的，例如如果对于一个ACTIVE状态的事务进行COMMIT提交，mysql就会抛出异常

> ERROR 1399 (XAE07): XAER_RMFAIL: The command cannot be executed when global transaction is in the ACTIVE state

而MySQL的JDBC连接驱动包从5.0.0版本开始，也已经直接支持XA事务。

```
public class MysqlXAConnectionTest {
   public static void main(String[] args) throws SQLException {
      //true表示打印XA语句,, 用于调试
      boolean logXaCommands = true;
      // 获得资源管理器操作接口实例 RM1
      Connection conn1 =
DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root",
"root");
      XAConnection xaConn1 = new
MysqlXAConnection((com.mysql.jdbc.Connection) conn1, logXaCommands);
      XAResource rm1 = xaConn1.getXAResource();
      // 获得资源管理器操作接口实例 RM2
      Connection conn2 =
DriverManager.getConnection("jdbc:mysql://localhost:3306/test",
"root","root");
      XAConnection xaConn2 = new
MysqlXAConnection((com.mysql.jdbc.Connection) conn2, logXaCommands);
      XAResource rm2 = xaConn2.getXAResource();
      // AP请求TM执行一个分布式事务，TM生成全局事务id
      byte[] gtrid = "g12345".getBytes();
      int formatId = 1;
      try {
         // ==============分别执行RM1和RM2上的事务分支====================
         // TM生成rm1上的事务分支id
         byte[] bqual1 = "b00001".getBytes();
         Xid xid1 = new MysqlXid(gtrid, bqual1, formatId);
         // 执行rm1上的事务分支
```

```
            rm1.start(xid1, XAResource.TMNOFLAGS);//One of TMNOFLAGS, TMJOIN,
or TMRESUME.
            PreparedStatement ps1 = conn1.prepareStatement("INSERT into
user(name) VALUES ('tianshouzhi')");
            ps1.execute();
            rm1.end(xid1, XAResource.TMSUCCESS);
            // TM生成rm2上的事务分支id
            byte[] bqual2 = "b00002".getBytes();
            Xid xid2 = new MysqlXid(gtrid, bqual2, formatId);
            // 执行rm2上的事务分支
            rm2.start(xid2, XAResource.TMNOFLAGS);
            PreparedStatement ps2 = conn2.prepareStatement("INSERT into
user(name) VALUES ('wangxiaoxiao')");
            ps2.execute();
            rm2.end(xid2, XAResource.TMSUCCESS);
            // ===================两阶段提交==============================
            // phase1：询问所有的RM 准备提交事务分支
            int rm1_prepare = rm1.prepare(xid1);
            int rm2_prepare = rm2.prepare(xid2);
            // phase2：提交所有事务分支
            boolean onePhase = false; //TM判断有2个事务分支，所以不能优化为一阶段提交
            if (rm1_prepare == XAResource.XA_OK
                  && rm2_prepare == XAResource.XA_OK
                  ) {//所有事务分支都prepare成功，提交所有事务分支
               rm1.commit(xid1, onePhase);
               rm2.commit(xid2, onePhase);
            } else {//如果有事务分支没有成功，则回滚
               rm1.rollback(xid1);
               rm1.rollback(xid2);
            }
         } catch (XAException e) {
            // 如果出现异常，也要进行回滚
            e.printStackTrace();
         }
      }
```

这其中，XA标准规范了事务XID的格式。有三个部分: gtrid [, bqual [, formatID]] 其中

- gtrid 是一个全局事务标识符 global transaction identifier
- bqual 是一个分支限定符 branch qualifier 。如果没有提供，会使用默认值就是一个空字符串。
- formatID 是一个数字，用于标记gtrid和bqual值的格式，这是一个正整数，最小为0，默认值就是1。

但是使用XA事务时需要注意以下几点：

- XA事务无法自动提交
- XA事务效率非常低下，全局事务的状态都需要持久化。性能非常低下，通常耗时能达到本地事务的10倍。
- XA事务在提交前出现故障的话，很难将问题隔离开。

Base柔性事务

柔性事务是指 Basic Available(基本可用)、Soft-state(软状态/柔性事务)、Eventual Consistency(最终一致性)。他的核心思想是既然无法保证分布式事务每时每刻的强一致性，那就根据每个业务自身的特点，采用合适的方式来使系统达到最终一致性。这里所谓强一致性，就是指在任何时刻，分布式事务的各个参与方的事务状态都是对齐的。典型的强一致性场景就是操作系统的文件系统。不管有多少个软件操作同一个文件，文件的状态始终是一致的。

要保证分布式事务的强一致性，难度太大，所以实际业务中，只能根据业务特点进行适当的妥协。而阿里经过不断研究后，最终提出了柔性事务的妥协方式。大体上来说，形成了以下几种处理模式：

- 最大努力通知型： 即分布式事务参与方都努力将自己的事务处理结果通知给分布式事务的其他参与方，也就是只保证尽力而为，不保证一定成功。适用于很多跨公司、流程复杂的场景。例如 电商完成一笔支付需要电商自己更改订单状态，同时需要调用支付宝完成实际支付。这种场景下，如果支付宝处理订单支付出错了，就只能尽力将错误结果通知给电商网站，让电商网站回退订单状态。
- 补偿性：不保证事务实时的对齐状态，对于未对齐的事务，事后进行补偿。同样在电商调用支付宝的这个场景中，就只能通过定期对账的方式保证在一个账期内，双方的事务最终是对齐的，至于具体的每一笔订单，只能进行最大努力通知，不保证事务对齐。
- 异步确保型： 典型的场景就是RocketMQ的事务消息机制。通过不断的异步确认，保证分布式事务的最终一致性。
- 两阶段型： 通常用于都是操作数据库的分布式事务场景。 第一阶段准备阶段：分布式事务的各个参与方都提交自己的本地事务，并且锁定相关的资源。第二阶段提交阶段：由一个第三方的事务协调者综合处理各方的事务执行情况，通知各个参与方统一进行事务提交或者回退。

> 与两阶段协议对应的是增强版的三阶段协议。他们的本质区别在于，两阶段协议在准备阶段需要锁定资源，例如在数据库中，就是要加行锁。防止其他事务对数据做了调整，这样会导致在第二个阶段数据无法正常回滚。而对于Redis等其他的一些数据源，无法提供对应的锁资源操作。

> 为了适应这样的场景，就在两阶段的准备阶段之前加一个询问阶段，在这一阶段，事务协调者只是询问各个参与方是否做好了准备。例如对于Redis，可能就是表示创建好了Redis连接。对于数据库，就只是表示已经创建好了JDBC连接。然后在准备阶段，参与者统一去写redo和undo日志，记录自己的事务提交状态。然后在最后的提交阶段，由事务协调者通知各个参与方统一进行事务提交或者回滚。
>
> **两阶段协议与三阶段协议的本质区别在于要不要锁资源。三阶段不用锁资源，所以适用性更强，并且对于事务的一致性强度也更高。**但是在编程实现上，两阶段对业务的侵入比较小，在很多框架中，直接声明一个注解就可以完成了。而三阶段对业务的侵入就比较大了，需要所有业务都按照三阶段的要求改造成TCC的模式。所以三阶段适合于一些对分布式事务准确性和时效性要求非常高的场景，比如很多银行系统。例如在一个典型的订单那支付操作中，A需要向B支付100元。使用TCC，在try阶段，通常会要求给订单设定一个状态UPDATING，同时A减少100元，B增加100元，并且将A需要减少的100元与B需要增加的100元这两个数据都单独记录下来，相当于锁定库存。这样可以用来实现类似锁资源的效果。然后在后续的confirm或者cancel操作中，将事务最终进行对齐。在这一步，首先需要修改订单状态，然后修改A和B的账户。这里注意，给A和B调整的账户都需要从锁定的资源中取，而不能凭空修改账户的数据。

- SAGA模式：由分布式事务的各个参与方自己提供正向的提交操作以及逆向的回滚操作。事务协调者可以在各个参与方提交事务后，随时协调各个事务参与方进行回滚。具体来说，每个SAGA事务包含T1,T2,T3....Tn操作，每个操作都对应具体的补偿操作C1,C2,C3....Cn。那么SAGA事务就需要保证： 1、所遇事务T1,T2,T3...Tn执行成功(最佳情况)，2、如果有事务执行失败了，T1,T2,T3....Tj,Cj,....C3,C2,C1执行成功(0<j<n)。例如对于客户扣款100块钱的操作，电商网站和支付宝都提供扣减客户100块钱的操作作为正向事务，同时也提供给客户加100块钱余额的操作作为逆向操作。这样事务协调者可以在检查电商网站和支付宝的扣款行为后，随时通知他们进行回滚。 这种方式对业务的影响也是比较大的。适合于事务流程比较长，参与方比较多的场景。

所以从广义上来看，ShardingSphere支持的这种XA事务其实也是属于一种柔性事务。但是一般情况下，BASE柔性事务特指Seata框架提供的柔性事务，因为BASE实际上是集成了阿里对于分布式事务的所有研究，而阿里的这些研究成果，最终都沉淀到了Seata框架中。ShardingSphere中对于柔性事务的支持，其实也是更多的基于Seata的AT模式，来实现的两阶段提交。这里要注意的是，虽然XA和AT都是基

于两阶段协议提供的实现，但是AT模式相比XA模式，简化了对于资源锁的要求，所以可以认为在大部分的业务场景下，AT模式比XA模式性能稍高。

ShardingJDBC扩展分布式事务管理器

分布式事务相关的扩展点，可以参见ShardingSphere的官方说明，也可以参考源码下的docs\document\content\dev-manual\transaction.cn.md。

事务管理器的父接口是ShardingTransactionManager，下面提供了SeataATShardingTransactionManager和XAShardingTransactionManager两个实现类，也可以通过SPI机制扩展出自己的分布式事务管理器。

ShardingTransactionManager接口的源码如下：

```
public interface ShardingTransactionManager extends AutoCloseable {
    // 初始化
    void init(DatabaseType databaseType, Collection<ResourceDataSource>
resourceDataSources, String transactionMangerType);
    // 获取事务类型，ShardingSphere就是通过这个事务类型去加载对应的事务管理器
    TransactionType getTransactionType();
    // 判断事务是否在进行当中
    boolean isInTransaction();
    // 获得事务连接
    Connection getConnection(String dataSourceName) throws SQLException;
    // 开始本地事务
    void begin();
    // 提交本地事务
    void commit();
    // 回滚本地事务
    void rollback();
}
```

其实，这里我们结合分布式事务的理论来看这个接口，可以看到，虽然ShardingSphere是按照两阶段协议实现的事务控制，但是光从这个接口中其实体现出的是三阶段协议的流程思想。

在TCC Try-Confirm-Cancel的三阶段协议中，init方法通常就是准备数据，建立好连接；对应的就是Try阶段，begin和commit方法提交本地事务，对应Confirm阶段；而rollback是进行事务回滚，就是Cancel阶段。当然，这只是事务管理器的流程，并不是事务真正执行的流程，所以并不存在两阶段或者三阶段的冲突，但是，由此也能了解到ShardingSphere关于分布式事务的整理处理思想。

三、ShardingProxy分布式事务示例

ShardingProxy与ShardingJDBC系出同门，接入分布式API的方式基本是一致的。同样支持LOCAL、XA、BASE类型的事务。

关于分布式事务的配置， 是由server.yaml中配置的属性 props:proxy.transaction.type: LOCAL指定的， 默认是LOCAL。

如果要使用XA事务，将这个属性调整为XA即可。ShardingProxy默认就支持XA事务，默认的事务管理器是Atomikos。

其中，ShardingProxy默认就支持XA事务，默认的事务管理器是Atomikos。不用做任何配置，默认就会使用。可以试试在ShardingProxy中执行XA事务的相关语句。

> 注意，在ShardingProxy中，不支持直接使用begin语句打开事务(mysql中是支持的)。在github上查到已经有人提了这个BUG，有个开发人员承诺会在5.x版本中改善，但是在4.x版本中不会改进了。

XA模式测试过程：

```
mysql> begin;
Query OK, 0 rows affected (0.03 sec)

mysql> update sharding_db.course set user_id='4';
ERROR 1062 (23000): Duplicate entry '4' for key 'course_2.testUnique'
mysql> select * from sharding_db.course;
+--------------------+----------------+---------+---------+
| cid                | cname          | user_id | cstatus |
+--------------------+----------------+---------+---------+
|         1649499884 | shardingsphere |       4 | 1       |
|         1649500066 | shardingsphere |       4 | 1       |
| 586146123224190976 | java           |       4 | 1       |
|         1649497221 | shardingsphere |    1000 | 1       |
4 rows in set (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from sharding_db.course;
+--------------------+----------------+---------+---------+
| cid                | cname          | user_id | cstatus |
+--------------------+----------------+---------+---------+
|         1649499884 | shardingsphere |       3 | 1       |
```

```
|         1649500066 | shardingsphere |       3 | 1       |
| 586146123224190976 | java           |       3 | 1       |
|         1649497221 | shardingsphere |    1000 | 1       |
+--------------------+----------------+---------+---------+
4 rows in set (0.00 sec)
```

> 从测试过程中可以看到，在执行SQL语句时，由于course_2表的唯一索引配置，造成了多个表后的事务并没有对齐。course_1表中的两条数据和course_2表中的一条数据，user_id字段还是被更新成了4，这时，整个分库分表的事务是不对齐的，错位的。而手动将事务回滚后，各个分片的数据都一起完成了回滚，整个事务才对齐。

而如果需要使用seata的AT模式的话，需要手动将实现了SeataAT模式的SPI扩展jar包放到ShardingProxy的Lib目录当中。jar包名称sharding-transaction-base-seata-at-4.1.1.jar，和 seata相关的jar包(还包括对应的注册消息)。如果需要获得这个jar包，可以从maven仓库中下载，具体的maven仓库坐标：

```xml
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-transaction-base-seata-at</artifactId>
    <version>4.1.1</version>
</dependency>
```

然后同样还需要移植seata相关的配置文件。包括seata.conf,registry.conf,file.conf(如果需要的话)。

最后在server.yaml中，将事务类型配置成BASE。然后就可以使用seata的AT模式。

> 这种方式使用很重，一般不是非常核心的业务逻辑的话， 不会使用。

> 有道云笔记分享地址：
>
> 文档：VIP05-ShardingSphere分布式事务详解.m...
> 链接：http://note.youdao.com/noteshare?id=52b4bb1e09e0007ee0ed0c8daf5c3ccf&sub=C26D9B734E6C49EFB57B7E5673BE89E8