Sharding分库分表实战 下

图灵：楼兰

一、上一节课的课后作业分析：

1、定制主键生成策略

分布式主键要怎么设计？1、全局唯一，2、高性能，3、高可用，4、趋势递增

数据库自增长： 单机下自增长没有问题，但是分布式情况下会造成主键冲突。解决方案：集群模式，设置起始值和自增步长。例如三个集群，第一个台机器初始值0，自增长3；第二台机器初始值1，自增长3；第三台机器初始值2，自增长3。优点：解决DB单点问题。缺点：不利于扩容，高并发下每个数据库的自身压力还是会非常大。

ShardingSphere默认提供的两种分布式主键生成策略： UUID， SnowFlake。

UUID有什么问题？优点：生成简单、无网络消耗、全局唯一。缺点：无序字符串、没有具体业务含义、长度太长，对MySQL 的性能消耗较大。

SnowFlake有什么问题？优点：趋势递增、全局唯一、无网络消耗。缺点：时间戳依赖于机器时钟。如果机器时间回调，还是可能会有冲突。

扩展方式：基于Redis扩展自己的分布式主键。--基于SPI机制扩展 OrderByRedisKeyGenerator

> 如何在ShardingProxy中使用自定义的主键生成策略？

2、如何定制适合自己项目的分片策略

上一节课提到了我们是采用的取模平均分配的方式，这种分片策略数据分布得比较平均，但是不容易扩展，以后如果需要加减机器，都需要进行庞大的数据迁移。所以可以实现一种定制的分片策略，在整体上是按年份进行分片，在每年的数据内再按照取模进行分片。

实现方式：1、对于oms_order表，要实现的是按照id=xxx这样的逻辑进行分片。包含insert 语句和select 语句。所以，要采用Standard中的precise精确分片方式。在分库分表配置文件中，增加配置信息：

```
spring.shardingsphere.sharding.tables.oms_order.table-strategy.standard.precise-algorithm-class-name=com.tuling.tulingmall.sharding.OrderPreciseShardingAlgorithm
```

然后就可以在分片逻辑实现类中实现对id先按年划分范围，然后按2取模平均分配数据

```
public class OrderPreciseShardingAlgorithm implements PreciseShardingAlgorithm {
    @Override
    public String doSharding(Collection collection, PreciseShardingValue preciseShardingValue) {
        // collection中保存所有真实节点
        collection.stream().forEach((item)->{
            log.info("actual node table:{}",item);
        });
        final Long columnValue = (Long)preciseShardingValue.getValue();
        final String columnName = preciseShardingValue.getColumnName();
        final String logicTableName = preciseShardingValue.getLogicTableName();

        //preciseShardingValue中保存分片逻辑，包含逻辑表，分片字段和查询的条件值
        log.info("logic table name:{},rout column:{}", logicTableName, columnName);
        log.info("column value:{}", columnValue);

        //采用自定义的ID生成器后，ID的前四位数字就是年份
        final String year = columnValue.toString().substring(0, 4);

        RedisOpsUtil redisOpsUtil = TulingmallOrderApplication.getBean("redisOpsUtil");
        final Integer shardingKey = redisOpsUtil.get("shardingYear:" + year, Integer.class);

        String actualTableName = logicTableName+"_"+year+"_"+(columnValue % shardingKey);
        //返回的结构就是具体的实际表。sharddingsphere会把逻辑表改成这个实际表
        return actualTableName;
    }
}
```

> 这是分表的逻辑。分库如何做？

3、关于ShardingSphere的扩展点：

ShardingSphere基于SPI机制提供了非常多的扩展点，具体可以参见估官方文档，开发者手册部分。

ShardingAlgorithm扩展点就列出了ShardingSphere默认提供的多种分片策略：
InlineShardingAlgorithm：基于行表达式的分片算法
ModShardingAlgorithm：基于取模的分片算法
HashModShardingAlgorithm：基于哈希取模的分片算法
FixedIntervalShardingAlgorithm：基于固定时间范围的分片算法
MutableIntervalShardingAlgorithm：基于可变时间范围的分片算法
VolumeBasedRangeShardingAlgorithm：基于分片容量的范围分片算法
BoundaryBasedRangeShardingAlgorithm：基于分片边界的范围分片算法

还有其他扩展点，也要学会自己去理解。

二、分布式事务处理方式梳理：

1、强一致性与最终一致性

在通常的分布式事务场景中，首先会根据业务场景要求，将事务分为强一致性和最终一致性。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。

但是通常来说，要实现强一致性的事务难度是非常大的，尤其在分布式场景下，还需要面对非常多不确定的资源，比如网络、服务状态等。所以通常情况下，只需要实现最终一致性就可以了。 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。而经过阿里的不断研究，最终提出了柔性事务BASE这样

2、最终一致性事务设计方案：

一般来说，要达成事务最终一致性，大体上有以下几种处理模式：

- 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。这是针对事务的高一致性场景进行要求的，任何资源都不能凭空产生。

- 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。

3、分布式事务的角色划分

我们都在知道，处理分布式事务时，由于每个事务参与者都无法知道整个事务的执行情况，所以需要抽取出一个第三方的事务协调者角色来对事务进行协调。那事务协调者要做哪些事情呢?

我们先来聊一个故事，分布式事务问题的起源： **拜占庭将军问题**

拜占庭的首都。由于当时拜占庭罗马帝国国土辽阔，为了达到防御目的，每个军队都分隔很远，将军与将军之间只能靠信差传消息。在战争的时候，拜占庭军队内所有将军和副官必须达成一致的共识，决定是否有赢的机会才去攻打敌人的阵营。但是，在军队内有可能存有叛徒和敌军的间谍，左右将军们的决定又扰乱整体军队的秩序。在进行共识时，结果并不代表大多数人的意见。这时候，在已知有成员谋反的情况下，其余忠诚的将军在不受叛徒的影响下如何达成一致的协议，拜占庭问题就此形成。

拜占庭将军问题是一个协议问题，拜占庭国军队的将军们必须全体一致的决定是否攻击某一支敌军。问题是这些将军在地理上是分隔开来的，并且将军中存在叛徒。叛徒可以任意行动以达到以下目标：欺骗某些将军采取进攻行动；促成一个不是所有将军都同意的决定，如当将军们不希望进攻时促成进攻行动；或者迷惑某些将军，使他们无法做出决定。如果叛徒达到了这些目的之一，则任何攻击行动的结果都是注定要失败的，只有完全达成一致的努力才能获得胜利 。

拜占庭假设是对现实世界的模型化，由于硬件错误、网络拥塞或断开以及遭到恶意攻击，计算机和网络可能出现不可预料的行为。

分布式场景下，事务参与者已经无法控制整个事务的执行情况，所以需要单独抽取出一个事务协调者对象来对整体事务进度进行控制。所以事务协调者要处理的问题就不只是简单的事务协调，还需要处理非常多的特殊情况。

三、分库分表场景下的分布式事务方案

接下来回到分库分表的场景下，如果让你来给ShardingJDBC来设计一个事务管理功能，你会怎么选择？两阶段还是三阶段？

分库分表场景下，针对的都是关系型数据库，是可以锁资源的，所以，肯定会选择两阶段的方式来设计。而两阶段主要有两种方式，就是XA事务和BASE柔性事务。

> 理解XA事务和BASE柔性事务的区别的重点在于，协调者角色是谁，以及协调者的职能要求。

1、XA事务

什么是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();
         }
      }
```

> 之前跟一个同学讨论了一个有意思的问题：在mysql当中，使用begin、commit、rollback语句也能实现二阶段执行，但是，他是不是就可以作为一个分布式事务的实现方案呢?
>
> begin;
>
> insert into user values('aaa');
>
> select * from user;
>
> rollback;
> --commit;
>
> select * from user;

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。

> XA事务只是一套规范，并不限定具体的语言。在JAVA当中有JTA规范来支持XA事务，具体有很多个实现框架。比如ShardingSphere默认采用atomikos作为实现框架。同时，也支持bitronix、narayana等框架，甚至提供了SPI扩展点，可以自行扩展具体的实现。如果对分布式事务真正感兴趣的同学，不妨自行去深入理解下这几个框架。

2、BASE柔性事务

BASE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)。从广义上来看，像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。

四、XA事务和BASE柔性事务实战

1、分库分表场景下的分布式事务问题：

修改完成后，下单操作需要插入两张逻辑表。oms_order和oms_order_item。这时候，如果插入oms_order数据成功，但是插入oms_order_item失败，就会造成事务不一致的问题。 这种情况在分库分表下会怎么样？就从一个本地事务问题升级成了分布式事务问题。

在这种场景下，适合哪种方式处理?

> 分库分表只针对关系型数据库，所以采用两阶段事务处理就足够了。

ShardingSphere基于SPI扩展分布式事务管理器。官方提供了两种实现方式， XA事务和Seata AT事务。

XA事务

需要引入maven依赖

```
<!-- 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>
```

然后使用的话，需要在MyBatis中打开@EnableTransactionManagement注解。然后注入PlatformTransactionManager对象。

然后在OmsPortalOrderService的generateOrder方法上打开@Transactional注解和 @ShardingTransactionType(TransactionType.XA) 注解。

然后就可以来测试了。 把其中一个oms_order_item表删掉，然后从前端下单，来测试下。如果报错了就检查oms_order表的数据是否回滚了。--也可以从源码的BusiTest案例来测试。这个测试需要打开SeataATOrderService相关的application.properties配置文件。

> SPI扩展机制见sharding-transaction-xa-core-4.1.1.jar下的 \META-INF\services\org.apache.shardingsphere.transaction.spi.ShardingTransactionManager。这个就是SPI扩展配置文件。

Seata AT事务

首先需要提前部署好Seata，采用nacos配置中心，把Seata的一大堆配置项都提前导入到nacos中。

使用时，需要引入依赖以下依赖，并且把XA事务的依赖去掉。

```
<!-- 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>
```

然后需要在resource下添加seata.conf文件。文件内容：

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

然后需要在每个数据分片建立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';
```

测试示例在BusiTest中。尽量就不要到实际项目中去试了，seata这东西问题太多了。

> 代码已经修改完成。下单操作只需要切换Pom依赖就可以了。然后用PostMan进行下单操作(下单操作已经修改过，下单后不会删除购物车的物品，所以可以不停的重复下单)。通过调整表名制造出分布式事务的事件。

五、SharingProxy的分布式事务

对于ShardingProxy，由于他的功能并没有ShardingJDBC那么灵活，所以在ShardingProxy中使用分布式事务的方式就简单很多。

对于XA事务：可以直接在config配置文件中将事务直接配置为XA。默认就会采用Atomikos来实现，当然也可以自行选择其他的实现框架，但是需要手动替换jar包。。而如果要使用seata的BASE柔性事务，就需要自行将jar包拷贝到lib目录下，比较麻烦，一般项目中很少用到。

六、分库分表课程总结

传统的关系型数据库像mysql,oracle，他们的集群功能往往都是偏弱的。当他们面临数据量过大的问题时，往往有两个解决思路，一种就是换产品，换成VoltDB、TiDB、ES、Hbase等这些大数据产品，但是换产品往往需要大量的投入，包括资金、人力投入，更重要的是要对项目进行大的改造，难以保证项目整体的稳定性。所以最常使用的还是第二种方案，即分库分表。

分库分表看似只是将数据简单的从一个数据库分布多个数据库，但是需要面对的问题还是非常多的。而shardingsphere框架能够很好的帮我们解决分库分表面临的非常多的问题。我们只需要对核心业务进行简单配置，就能够实现分库分表。课上也带大家在实战项目中实现了对电商订单的分库分表配置。但是，框架功能虽然强大，但是要用好shrdingsphere，也对技术提出了更高的门槛。另外，ShardingSphere的功能非常多，留给用户的扩展点也很多，如果在项目中真正想要把ShardingSphere用好的话，还是需要回顾下我们的shardingsphere的VIP课程，或者自行查阅资料，加深对ShardingSphere的理解，这个框架绝对不是简单的配置下分库分表就可以用了的。

分库分表带来了数据扩展性，但是同时也带来了非常多的问题。其中分布式事务就是最为令人头疼的问题。ShardingSphere集成了主流的开源框架，同时也预留了SPI扩展点，让用户自行扩展分布式事务逻辑。但是，分布式事务是个非常非常复杂的问题，在不同场景下需要有不同的控制方式。电商项目只是向大家展示了分布式事务很小的一个场景，在项目中面临不同业务场景时，针对分布式事务也需要有不同的取舍与设计方式。这样才能设计出高质量的三高项目。

> 文档：电商VIP-Sharding分库分表实战 下.md
> 链接：http://note.youdao.com/noteshare?id=7a06369a1e0602e96bad555a539d0016&sub=22334297930043318FD3E24AA3FB8D4C