主讲老师： Fox

学习本课程的基础：
1.掌握Seata的使用，可以先学习微服务专题Seata前两节实战课（Seata AT & XA & TCC）
2.掌握Rocketmq使用

```
文档：3 下单链路分布式事务Seata&MQ可靠消息...
链接：http://note.youdao.com/noteshare?id=95572518985a8b2ce16c2a7012185a8a&sub=F337BC531C5F428EA58356759F349CFD
```


1. 分布式事务

在微服务架构中，完成某一个业务功能可能需要横跨多个服务，操作多个数据库。这就涉及到到了分布式事务，需要操作的资源位于多个资源服务器上，而应用需要保证对于多个资源服务器的数据操作，要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同资源服务器的数据一致性。

1.1 电商项目下单链路中的分布式事务场景

- **用户下单冻结库存**

com.tuling.tulingmall.ordercurr.service.impl.OmsPortalOrderServiceImpl#generateOrder

- **支付成功后修改订单状态，异步扣减真实库存**

com.tuling.tulingmall.ordercurr.service.impl.OmsPortalOrderServiceImpl#paySuccess

1.2 常见分布式事务解决方案

	2PC	TCC	可靠消息	最大努力通知
一致性	强一致性	最终一致	最终一致	最终一致
吞吐量	低	中	高	高
实现复杂度	易	难	中	易

电商项目中会结合下单的业务重点讲解两种分布式事务解决方案：

- 2PC的方案： 基于Seata AT实现
- mq可靠消息的方案：基于Rocketmq事务消息实现

2. 基于Seata实现用户下单冻结库存场景的分布式事务

分布式事务组件Seata实战

2.1 Seata架构

在 Seata 的架构中，一共有三个角色：

- **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。

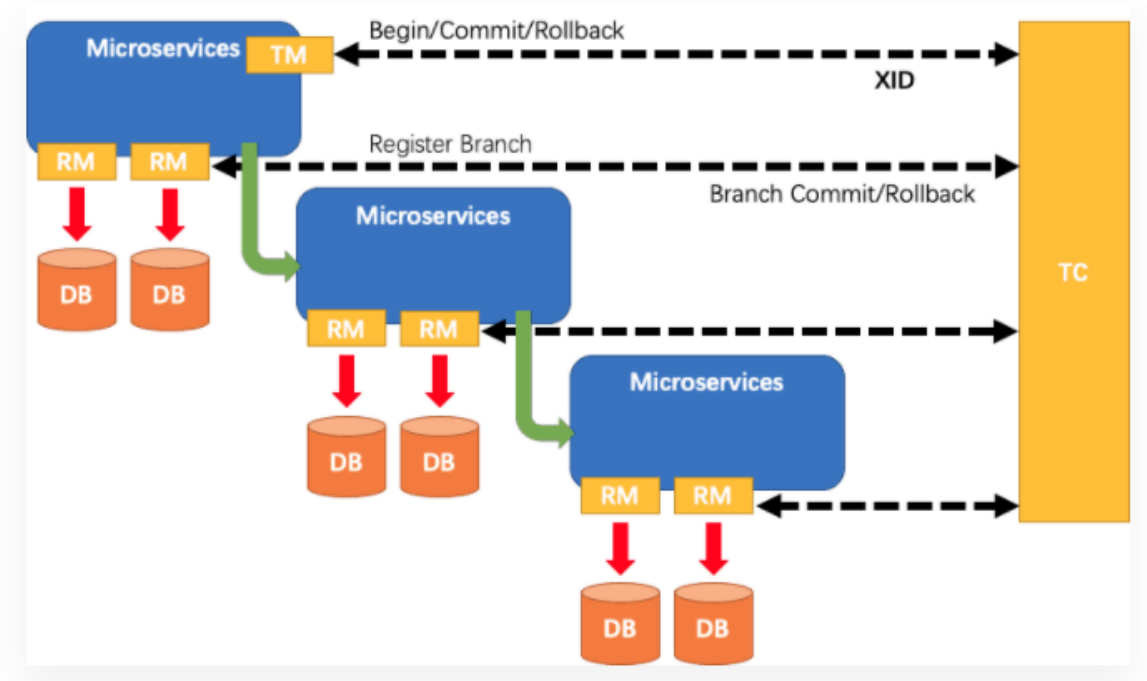
- **RM (Resource Manager) - 资源管理器**

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。

其中，TC 为单独部署的 Server 服务端，TM 和 RM 为嵌入到应用中的 Client 客户端。

在 Seata 中，一个分布式事务的生命周期如下：

1. TM 请求 TC 开启一个全局事务。TC 会生成一个 XID 作为该全局事务的编号。XID会在微服务的调用链路中传播，保证将多个微服务的子事务关联在一起。
2. RM 请求 TC 将本地事务注册为全局事务的分支事务，通过全局事务的 XID 进行关联。
3. TM 请求 TC 告诉 XID 对应的全局事务是进行提交还是回滚。
4. TC 驱动 RM 们将 XID 对应的自己的本地事务进行提交还是回滚。



2.2 整合Seata实战

Seata分TC、TM和RM三个角色，TC（Server端）为单独服务端部署，TM和RM（Client端）由业务系统集成。

Seata的版本选择

注意：微服务组件整合的时候需要考虑兼容性问题

电商项目选择的Spring Cloud Alibaba版本是2.2.6.RELEASE，所整合的seata版本是1.3.0。但是低版本的seata环境搭建繁琐，而且bug多，所以整合的时候尽量选择更高的版本，比如1.5.x

- Seata server版本选择1.5.1
- 微服务引入Seata依赖替换为1.5.1

```
<!--分布式事务 seata依赖-->
```

```
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
 <version>2.2.8.RELEASE</version>
 <exclusions>
 <exclusion>
 <groupId>io.seata</groupId>
 <artifactId>seata-spring-boot-starter</artifactId>
  </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>io.seata</groupId>
  <artifactId>seata-spring-boot-starter</artifactId>
  <version>1.5.1</version>
</dependency>
```

Seata Server（TC）环境搭建

参考第五期微服务专题seata的课程笔记搭建TC环境： 分布式事务组件Seata实战

Seata接入微服务

1）引入依赖

spring-cloud-starter-alibaba-seata内部集成了seata，并实现了xid传递

```
<!--分布式事务 seata依赖-->
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
 <version>2.2.8.RELEASE</version>
 <exclusions>
 <exclusion>
 <groupId>io.seata</groupId>
 <artifactId>seata-spring-boot-starter</artifactId>
  </exclusion>
  </exclusions>
</dependency>

<dependency>
 <groupId>io.seata</groupId>
 <artifactId>seata-spring-boot-starter</artifactId>
```

```
  <version>1.5.1</version>
</dependency>
```

2）微服务对应数据库中添加undo_log表(仅AT模式)

```
-- for AT mode you must to init this sql for you business database. the seata server not need it.
CREATE TABLE IF NOT EXISTS `undo_log`
(
 `branch_id` BIGINT NOT NULL COMMENT 'branch transaction id',
 `xid` VARCHAR(128) 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(6) NOT NULL COMMENT 'create datetime',
 `log_modified` DATETIME(6) NOT NULL COMMENT 'modify datetime',
 UNIQUE KEY `ux_undo_log` (`xid`, `branch_id`)
) ENGINE = InnoDB
 AUTO_INCREMENT = 1
 DEFAULT CHARSET = utf8mb4 COMMENT ='AT transaction mode undo table';
```

3）微服务application.yml中添加seata配置

```
#seata 配置
seata:
 application-id: tulingmall-product
 # seata 服务分组，要与服务端配置service.vgroup_mapping的后缀对应
 tx-service-group: tuling-order-group
 registry:
 # 指定nacos作为注册中心
 type: nacos
 nacos:
  application: seata-server
  server-addr: 192.168.65.103:8848
  group: SEATA_GROUP

 config:
 # 指定nacos作为配置中心
 type: nacos
 nacos:
  server-addr: 192.168.65.103:8848
  namespace: 7e838c12-8554-4231-82d5-6d93573ddf32
```

```
20   group: SEATA_GROUP
21   data-id: seataServer.properties
```

注意：请确保client与server的注册中心和配置中心namespace和group一致

4）全局事务发起者开启全局事务配置

此处是本项目接入seata最难的地方：原因在于订单表用了分库分表技术（shardingsphere），seata不能对逻辑表进行解析。不能简单的在全局事务发起方使用@GlobalTransactional

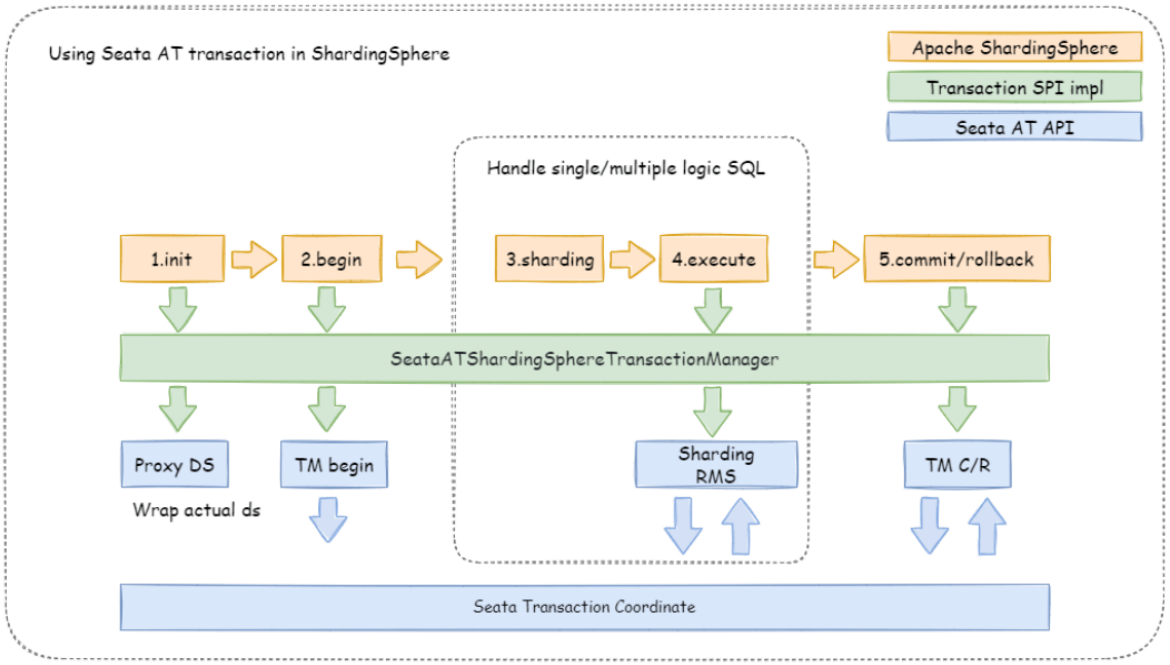
```
1  //此处不能使用@GlobalTransactional
2  @GlobalTransactional(name = "generateOrder",rollbackFor = Exception.class)
3  public CommonResult generateOrder(OrderParam orderParam, Long memberId) {
```

这个问题应该如何解决呢?

Apache ShardingSphere 分布式事务

- 基于 XA 协议的两阶段事务
- 基于 Seata 的柔性事务

整合 Seata AT 事务时，需要将 TM，RM 和 TC 的模型融入 Apache ShardingSphere 的分布式事务生态中。 在数据库资源上，Seata 通过对接 DataSource 接口，让 JDBC 操作可以同 TC 进行远程通信。 同样，Apache ShardingSphere 也是面向 DataSource 接口，对用户配置的数据源进行聚合。 因此，将 DataSource 封装为 基于Seata 的 DataSource 后，就可以将 Seata AT 事务融入到 Apache ShardingSphere的分片生态中。



ShardingSphere整合Seata

1）引入依赖

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

2）配置seata.conf

包含 Seata 柔性事务的应用启动时，用户配置的数据源会根据 seata.conf 的配置，适配为 Seata 事务所需的 DataSourceProxy，并且注册至 RM 中。

```
client {
 application.id = tulingmall-order-curr
 transaction.service.group = tuling-order-group
}
```

3）开启全局事务配置

```
//全局事务交给SeataATShardingTransactionManager管理
@ShardingTransactionType(TransactionType.BASE)
@Transactional
public CommonResult generateOrder(OrderParam orderParam, Long memberId) {
```

注意：GlobalTransactional和ShardingTransactionType不能同时出现，此处不能使用@GlobalTransactional。同时需要关闭数据源自动代理

```
seata:
 enable-auto-data-source-proxy: false #关闭数据源自动代理，交给sharding-jdbc那边
```

4. 柔性事务：可靠消息最终一致性方案实现

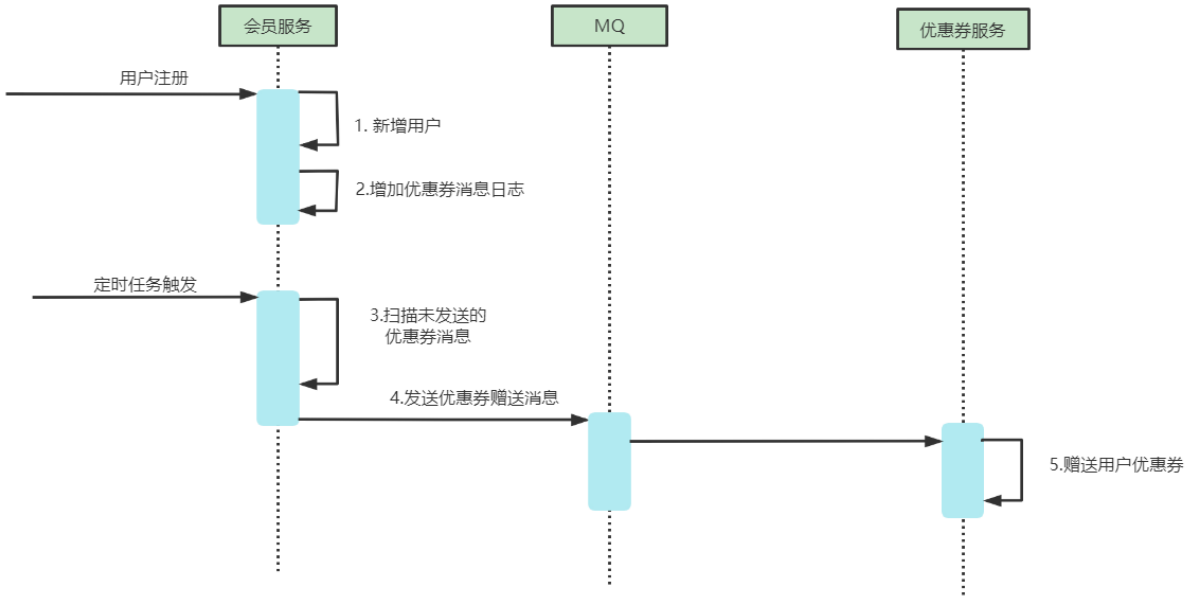
可靠消息最终一致性方案是指当事务发起执行完成本地事务后并发出一条消息，事务参与方（消息消费者）一定能够接收消息并处理事务成功，此方案强调的是只要消息发给事务参与方最终事务要达到一致。

4.1 本地消息表方案

本地消息表这个方案最初是eBay提出的，此方案的核心是通过本地事务保证数据业务操作和消息的一致性，然后通过定时任务将消息发送至消息中间件，待确认消息发送给消费方成功再将消息删除。

下面以注册送优惠券为例来说明：

共有两个微服务交互，会员服务和优惠券服务，用户服务负责添加用户，优惠券服务负责赠送优惠券。
交互流程如下：



1）用户注册

用户服务在本地事务新增用户和增加“优惠券消息日志”。（用户表和消息表通过本地事务保证一致）

下面是伪代码

```
begin transaction;
// 1.新增用户
// 2.存储优惠券消息日志
commit transation;
```

这种情况下，本地数据库操作与存储优惠券消息日志处于同一事务中，本地数据库操作与记录消息日志操作具备原子性。

2）定时任务扫描日志

如何保证将消息发送给消息队列呢?

经过第一步消息已经写到消息日志表中，可以启动独立的线程，定时对消息日志表中的消息进行扫描并发送至消息中间件，在消息中间件反馈发送成功后删除该消息日志，否则等待定时任务下一周期重试。

3）消费消息

如何保证消费者一定能消费到消息呢?

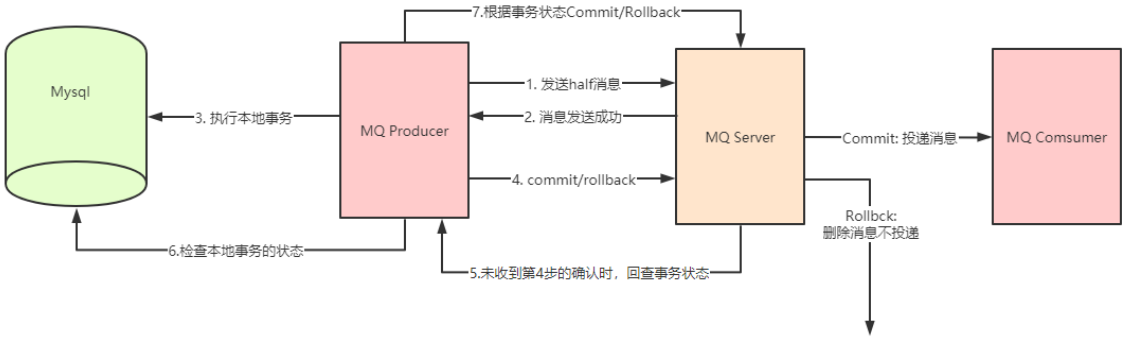
这里可以使用MQ的ack（即消息确认）机制，消费者监听MQ，如果消费者接收到消息并且业务处理完成后向MQ发送ack（即消息确认），此时说明消费者正常消费消息完成，MQ将不再向消费者推送消息，否则消费者会不断重试向消费者来发送消息。

优惠券服务接收到“赠送优惠券”消息，开始赠送用户优惠券，成功后消息中间件回应ack，否则消息中间件将重复投递此消息。
由于消息会重复投递，优惠券服务的“赠送优惠券”功能需要实现幂等性。

4.2 Rocketmq事务消息实现

RocketMQ事务消息设计则主要是为了解决Producer端的消息发送与本地事务执行的原子性问题，RocketMQ的设计中broker与producer端的双向通信能力，使得broker天生可以作为一个事务协调者存在；而RocketMQ本身提供的存储机制为事务消息提供了持久化能力；RocketMQ的高可用机制以及可靠消息设计则为事务消息在系统发生异常时依然能够保证达成事务的最终一致性。

在RocketMQ 4.3后实现了完整的事务消息，实际上其实是对本地消息表的一个封装，将本地消息表移动到了MQ内部，解决Producer端的消息发送与本地事务执行的原子性问题。



执行流程如下：

为方便理解我们以注册送优惠券的例子来描述整个流程。

Producer即MQ发送方，本例中是用户服务，负责新增用户。MQ订阅方即消息消费方，本例中是优惠券服务，负责新增优惠券。

1）Producer发送事务消息

Producer（MQ发送方）发送事务消息至MQ Server，MQ Server将消息状态标记为Prepared（预览状态），注意此时这条消息消费者（MQ订阅方）是无法消费到的。

2）MQ Server回应消息发送成功

MQ Server接收到Producer发送给的消息则回应发送成功表示MQ已接收到消息。

3）Producer执行本地事务

Producer端执行业务代码逻辑，通过本地数据库事务控制。

本例中，Producer执行添加用户操作。

4）消息投递

若Producer本地事务执行成功则自动向MQ Server发送commit消息，MQ Server接收到commit消息后将“增加优惠券消息”状态标记为可消费，此时MQ订阅方（优惠券服务）即正常消费消息；

若Producer 本地事务执行失败则自动向MQ Server发送rollback消息，MQ Server接收到rollback消息后将删除“增加优惠券消息”。

MQ订阅方（优惠券服务）消费消息，消费成功则向MQ回应ack，否则将重复接收消息。这里ack默认自动回应，即程序执行正常则自动回应ack。

5）事务回查

如果执行Producer端本地事务过程中，执行端挂掉，或者超时，MQ Server将会不停的询问同组的其他Producer来获取事务执行状态，这个过程叫**事务回查**。MQ Server会根据事务回查结果来决定是否投递消息。

以上主干流程已由RocketMQ实现，对用户则来说，用户需要分别实现本地事务执行以及本地事务回查方法，因此只需关注本地事务的执行状态即可。

RocketMQ提供RocketMQLocalTransactionListener接口：

```
1  public interface RocketMQLocalTransactionListener {
2   /**
3   发送prepare消息成功此方法被回调，该方法用于执行本地事务
4   @param msg 回传的消息，利用transactionId即可获取到该消息的唯一Id
5   @param arg 调用send方法时传递的参数，当send时候若有额外的参数可以传递到send方法中，这里能获取到
6   @return 返回事务状态，COMMIT ：提交 ROLLBACK ：回滚 UNKNOW ：回调
7   */
8   RocketMQLocalTransactionState executeLocalTransaction（Message msg，Object arg）；
9   /**
10   @param msg 通过获取transactionId来判断这条消息的本地事务执行状态
11   @return 返回事务状态，COMMIT ：提交 ROLLBACK ：回滚 UNKNOW ：回调
12   */
13   RocketMQLocalTransactionState checkLocalTransaction（Message msg）；
14  }
```