

第14章 分布式事务

学习目标

- 理解什么是事务
- 理解什么是分布式事务
- 理解CAP定理

CAP不能3者同时成立

- 能说出相关的分布式事务解决方案

1. 2PC-JTA分布式事务
2. 本地消息-业务库中添加对应的消息表和业务耦合实现
3. MQ事务消息-RocketMQ
4. Seata

- 理解Seata工作流程

AT模式-表
TCC模式-代码补偿机制

- 能实现Seata案例

Seata使用案例

作业：实现项目中分布式事务控制-下单->用户微服务（增加积分）->Goods微服务(库存递减)

1 分布式事务介绍

1.1 什么是事务

数据库事务(简称：事务，Transaction)是指数据库执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成[由当前业务逻辑多个不同操作构成]。

事务拥有以下四个特性，习惯上被称为ACID特性：

原子性(Atomicity)：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。

一致性(Consistency)：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态是指数据库中的数据应满足完整性约束。除此之外，一致性还有另外一层语义，就是事务的中间状态不能被观察到(这层语义也有说应该属于原子性)。

隔离性(Isolation)：多个事务并发执行时，一个事务的执行不应影响其他事务的执行，如同只有这一个操作在被数据库所执行一样。

持久性(Durability)：已被提交的事务对数据库的修改应该永久保存在数据库中。在事务结束时，此操作将不可逆转。

1.2 本地事务

起初，事务仅限于对单一数据库资源的访问控制,架构服务化以后，事务的概念延伸到了服务中。倘若将一个单一的服务操作作为一个事务，那么整个服务操作只能涉及一个单一的数据库资源,这类基于单个服务单一数据库资源访问的事务，被称为本地事务(Local Transaction)。



1.3 什么是分布式事务

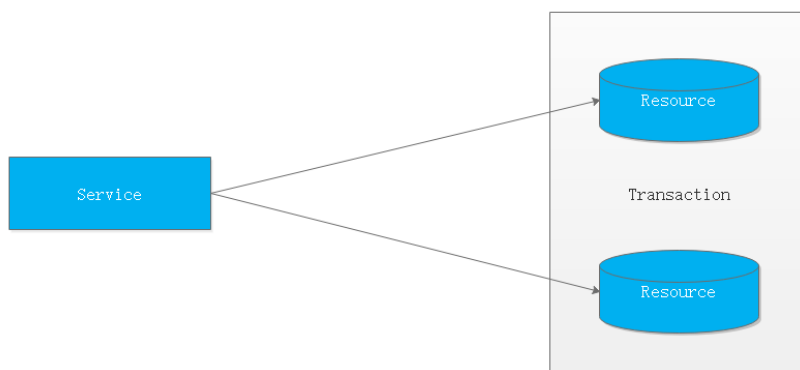
分布式事务指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上,且属于不同的应用，分布式事务需要保证这些操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

1.4 分布式事务应用架构

本地事务主要限制在单个会话内，不涉及多个数据库资源。但是在基于SOA(Service-Oriented Architecture，面向服务架构)的分布式应用环境下，越来越多的应用要求对多个数据库资源，多个服务的访问都能纳入到同一个事务当中，分布式事务应运而生。

1.4.1 单一服务分布式事务

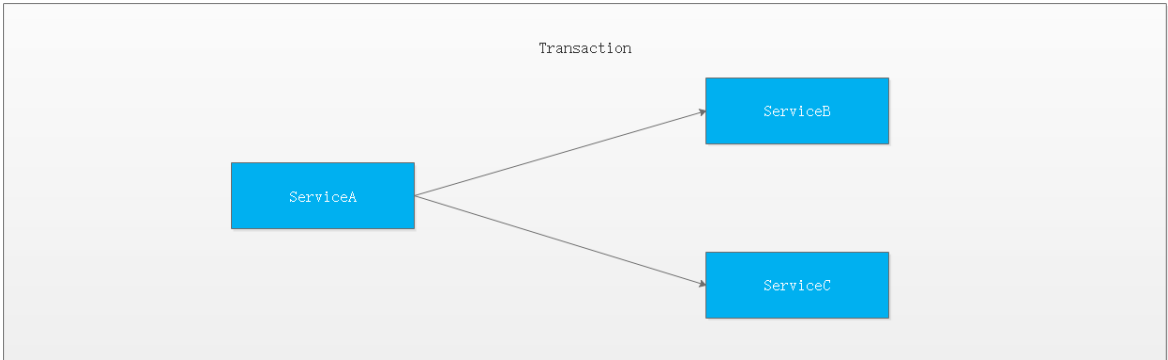
最早的分布式事务应用架构很简单，不涉及服务间的访问调用，仅仅是服务内操作涉及到对多个数据库资源的访问。



1.4.2 多服务分布式事务

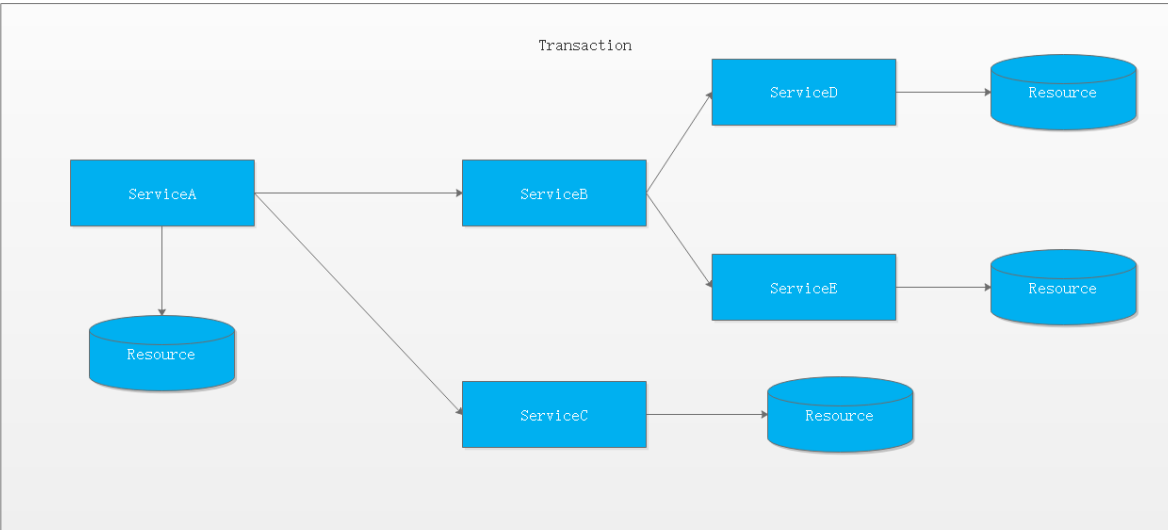
当一个服务操作访问不同的数据库资源，又希望对它们的访问具有事务特性时，就需要采用分布式事务来协调所有的事务参与者。

对于上面介绍的分布式事务应用架构，尽管一个服务操作会访问多个数据库资源，但是毕竟整个事务还是控制在单一服务的内部。如果一个服务操作需要调用另外一个服务，这时的事务就需要跨越多个服务了。在这种情况下，起始于某个服务的事务在调用另外一个服务的时候，需要以某种机制流转 to 另外一个服务，从而使被调用的服务访问的资源也自动加入到该事务当中来。下图反映了这样一个跨越多个服务的分布式事务：



1.4.3 多服务多数据源分布式事务

如果将上面这两种场景(一个服务可以调用多个数据库资源，也可以调用其他服务)结合在一起，对此进行延伸，整个分布式事务的参与者将会组成如下图所示的树形拓扑结构。在一个跨服务的分布式事务中，事务的发起者和提交均系同一个，它可以是整个调用的客户端，也可以是客户端最先调用的那个服务。



较之基于单一数据库资源访问的本地事务，分布式事务的应用架构更为复杂。在不同的分布式应用架构下，实现一个分布式事务要考虑的问题并不完全一样，比如对多资源的协调、事务的跨服务传播等，实现机制也是复杂多变。

事务的作用：

保证每个事务的数据一致性。

1.5 CAP定理

CAP 定理，又被叫作布鲁尔定理。对于设计分布式系统(不仅仅是分布式事务)的架构师来说，CAP 就是你的入门理论。

C (一致性): 对某个指定的客户端来说，读操作能返回最新的写操作。

对于数据分布在不同节点上的数据来说，如果在某个节点更新了数据，那么在其他节点如果都能读取到这个最新的数据，那么就称为强一致，如果有某个节点没有读取到，那就是分布式不一致。

A (可用性): 非故障的节点在合理的时间内返回合理的响应(不是错误和超时的响应)。可用性的两个关键一个是合理的时间，一个是合理的响应。

合理的时间指的是请求不能无限被阻塞，应该在合理的时间给出返回。合理的响应指的是系统应该明确返回结果并且结果是正确的，这里的正确指的是比如应该返回 50，而不是返回 40。

P (分区容错性): 当出现网络分区后，系统能够继续工作。打个比方，这里集群有多台机器，有台机器网络出现了问题，但是这个集群仍然可以正常工作。

熟悉 CAP 的人都知道，三者不能共有，如果感兴趣可以搜索 CAP 的证明，在分布式系统中，网络无法 100% 可靠，分区其实是一个必然现象。

如果我们选择了 CA 而放弃了 P，那么当发生分区现象时，为了保证一致性，这个时候必须拒绝请求，但是 A 又不允许，所以分布式系统理论上不可能选择 CA 架构，只能选择 CP 或者 AP 架构。

对于 CP 来说，放弃可用性，追求一致性和分区容错性，我们的 ZooKeeper 其实就是追求的强一致。

对于 AP 来说，放弃一致性(这里说的一致性**是强一致性**)，追求分区容错性和可用性，这是很多分布式系统设计时的选择，后面的 BASE 也是根据 AP 来扩展。

顺便一提，CAP 理论中是忽略网络延迟，也就是当事务提交时，从节点 A 复制到节点 B 没有延迟，但是在现实中这个是明显不可能的，所以总会有一定的时间是不一致。

同时 CAP 中选择两个，比如你选择了 CP，并不是叫你放弃 A。因为 P 出现的概率实在是太小了，大部分的时间你仍然需要保证 CA。

就算分区出现了你也要为后来的 A 做准备，比如通过一些日志的手段，是其他机器回复至可用。

```
/**
 * C 一致性
 * A 可用性
 * P 网络分区容错
 *
 * C A    -> P X
 * A P    -> C X
 * C P    -> A X
 */
```

2 分布式事务解决方案

1.XA两段提交(低效率)-21 XA JTA分布式事务解决方案

2.TCC三段提交(2段,高效率[不推荐(补偿代码)])

3.本地消息(MQ+Table)

4.事务消息(RocketMQ[alibaba])

2.1 基于XA协议的两阶段提交(2PC)

两阶段提交协议(Two Phase Commitment Protocol)中，涉及到两种角色

==一个事务协调者== (coordinator)：负责协调多个参与者进行事务投票及提交(回滚)

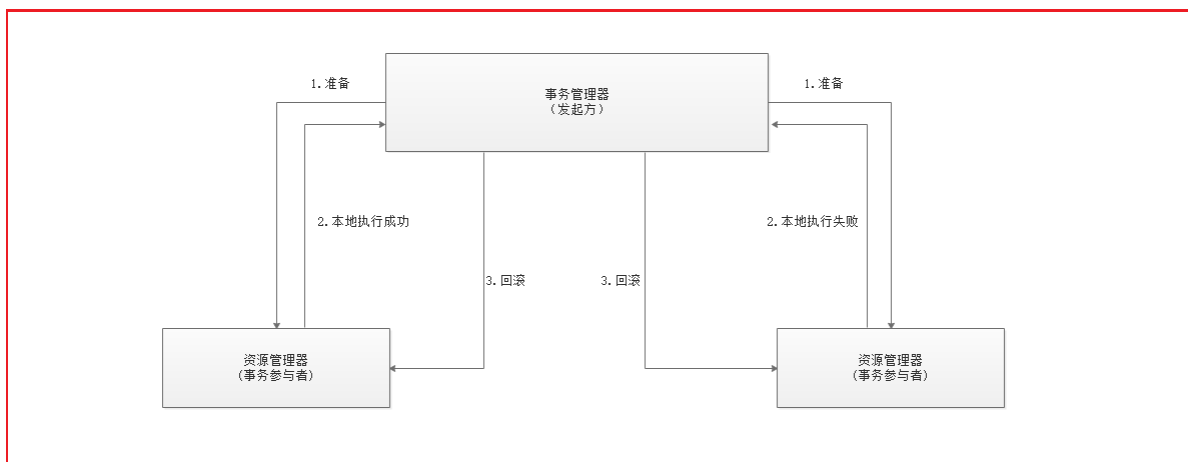
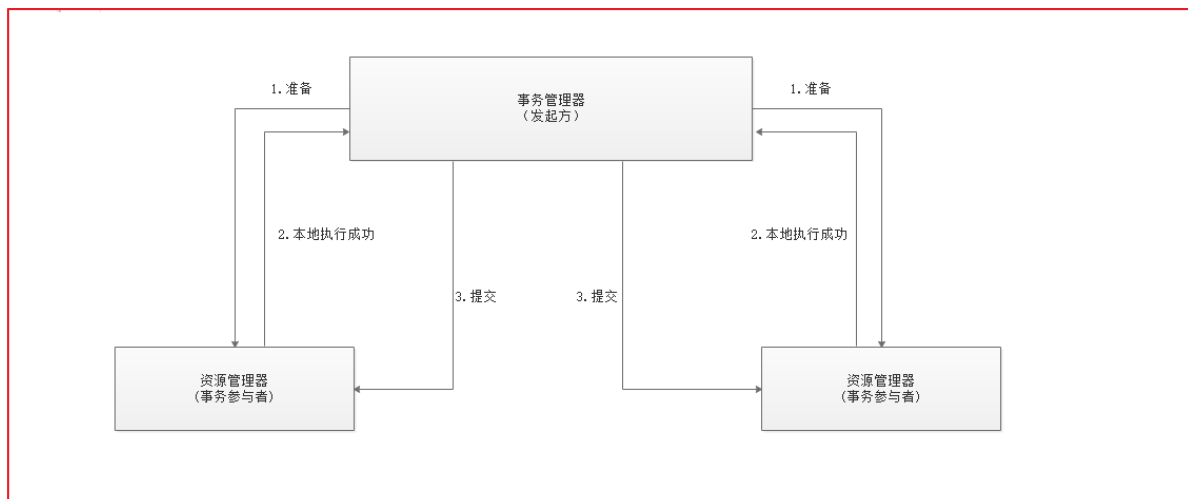
多个==事务参与者== (participants)：即本地事务执行者

总共处理步骤有两个

(1) 投票阶段 (voting phase)：协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。参与者将告知协调者自己的决策：同意（事务参与者本地事务执行成功，但未提交）或取消（本地事务执行故障）；

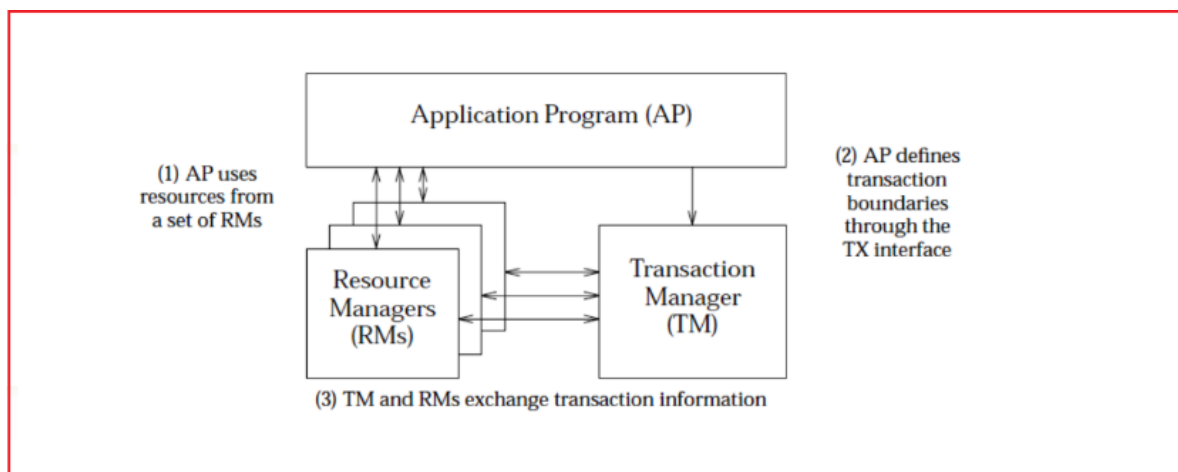
(2) 提交阶段 (commit phase)：收到参与者的通知后，协调者再向参与者发出通知，根据反馈情况决定各参与者是否要提交还是回滚；

如果所示 1-2为第一阶段， 2-3为第二阶段



如果任一资源管理器在第一阶段返回准备失败，那么事务管理器会要求所有资源管理器在第二阶段执行回滚操作。通过事务管理器的两阶段协调，最终所有资源管理器要么全部提交，要么全部回滚，最终状态都是一致的

官方解决方案图例如下：



优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。

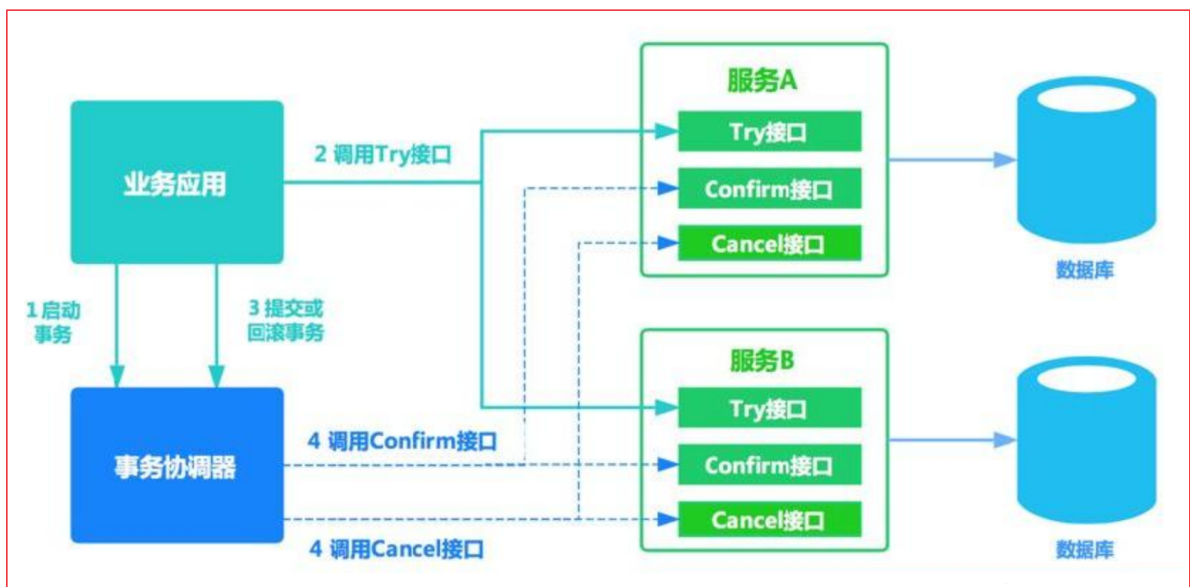
缺点： 牺牲了可用性，对性能影响较大，不适合高并发高性能场景，如果分布式系统跨接口调用，目前.NET 界还没有实现方案。

2.2 补偿事务 (TCC)

TCC 将事务提交分为 Try(method1) - Confirm(method2) - Cancel(method3) 3个操作。其和两阶段提交有点类似，Try为第一阶段，Confirm - Cancel为第二阶段，是一种应用层面侵入业务的两阶段提交。

| 操作方法 | 含义 |
|---------|--|
| Try | 预留业务资源/数据效验 |
| Confirm | 确认执行业务操作，实际提交数据，不做任何业务检查，try成功，confirm必定成功，需保证幂等 |
| Cancel | 取消执行业务操作，实际回滚数据，需保证幂等 |

其核心在于将业务分为两个操作步骤完成。不依赖 RM 对分布式事务的支持，而是通过对业务逻辑的分解来实现分布式事务。



例如：A要向B转账，思路大概是：

我们有一个本地方法，里面依次调用

- 1、首先在 Try 阶段，要先调用远程接口把 B和 A的钱给冻结起来。
- 2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。
- 3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法（Cancel）。

假设用户user表中有两个字段：可用余额(available_money)、冻结余额(frozen_money)

A扣钱对应服务A(ServiceA)

B加钱对应服务B(ServiceB)

转账订单服务(OrderService)

业务转账方法服务(BusinessService)

ServiceA, ServiceB, OrderService都需分别实现try(), confirm(), cacle()方法，方法对应业务逻辑如下

| 操作方法 | ServiceA | ServiceB | OrderService |
|-----------|------------------------------------|-----------|--------------|
| try() | 校验余额(并发控制) 冻结余额+1000 余额-1000 | 冻结余额+1000 | 创建转账订单，状态待转账 |
| confirm() | 冻结余额-1000 | | 状态变为转账成功 |
| cacle() | 冻结余额-1000 余额+1000 | | 状态变为转账失败 |

其中业务调用方BusinessService中就需要调用

ServiceA.try()

ServiceB.try()

OrderService.try()

- 1、当所有try()方法均执行成功时，对全局事物进行提交，即由事物管理器调用每个微服务的confirm()方法

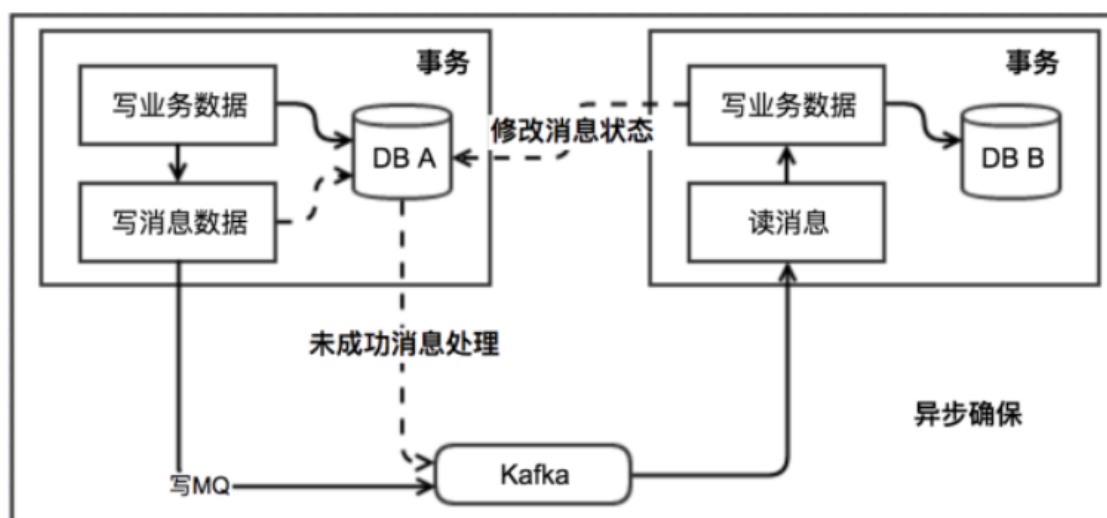
2、当任意一个方法try()失败(预留资源不足, 抑或网络异常, 代码异常等任何异常), 由事物管理器调用每个微服务的candle()方法对全局事务进行回滚

优点: 跟2PC比起来, 实现以及流程相对简单了一些, 但数据的一致性比2PC也要差一些

缺点: 缺点还是比较明显的, 在2,3步中都有可能失败。TCC属于应用层的一种补偿方式, 所以需要程序员在实现的时候多写很多补偿的代码, 在一些场景中, 一些业务流程可能用TCC不太好定义及处理。

2.3 本地消息表 (异步确保)

本地消息表这种实现方式应该是业界使用最多的, 其核心思想是将分布式事务拆分成本地事务进行处理, 这种思路是来源于ebay。我们可以从下面的流程图中看出其中的一些细节:



基本思路就是:

消息生产方, 需要额外建一个消息表, 并记录消息发送状态。消息表和业务数据要在一个事务里提交, 也就是说他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败, 会进行重试发送。

消息消费方, 需要处理这个消息, 并完成自己的业务逻辑。此时如果本地事务处理成功, 表明已经处理成功了, 如果处理失败, 那么就会重试执行。如果是业务上面的失败, 可以给生产方发送一个业务补偿消息, 通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表, 把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑, 这种方案还是非常实用的。

这种方案遵循BASE理论, 采用的是最终一致性, 笔者认为这是这几种方案里面比较适合实际业务场景的, 即不会出现像2PC那样复杂的实现(当调用链很长的时候, 2PC的可用性是非常低的), 也不会像TCC那样可能出现确认或者回滚不了的情况。

优点: 一种非常经典的实现, 避免了分布式事务, 实现了最终一致性。在 .NET 中有现成的解决方案。

缺点: 消息表会耦合到业务系统中, 如果没有封装好的解决方案, 会有很多杂活需要处理。

2.4 MQ 事务消息

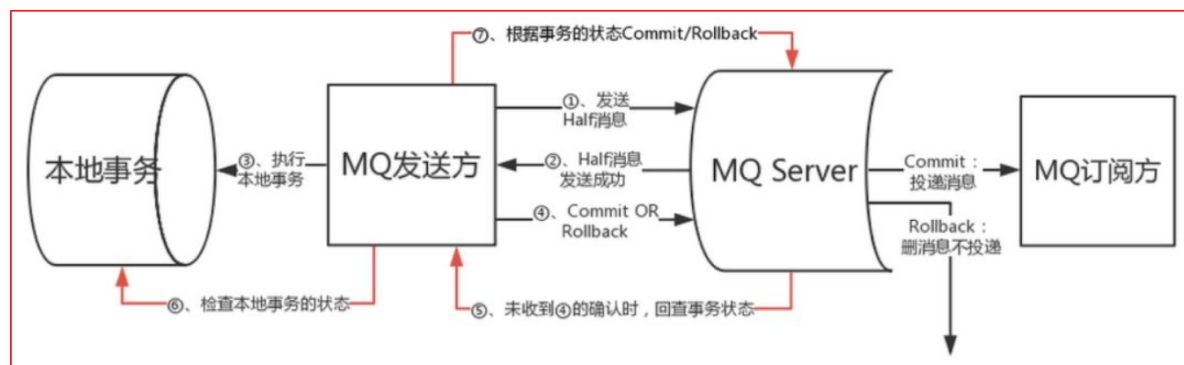
有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址。

第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。



优点： 实现了最终一致性，不需要依赖本地数据库事务。

缺点： 目前主流MQ中只有RocketMQ支持事务消息。

2.5 Seata 2PC->改进

2019 年 1 月，阿里巴巴中间件团队发起了开源项目 [Fescar](#) (Fast & EaSy Commit And Rollback)，和社区一起共建开源分布式事务解决方案。Fescar 的愿景是让分布式事务的使用像本地事务的使用一样，简单和高效，并逐步解决开发者们遇到的分布式事务方面的所有难题。

Fescar 开源后，蚂蚁金服加入 Fescar 社区参与共建，并在 Fescar 0.4.0 版本中贡献了 TCC 模式。

为了打造更中立、更开放、生态更加丰富的分布式事务开源社区，经过社区核心成员的投票，大家决定对 Fescar 进行品牌升级，并更名为 **Seata**，意为：**Simple Extensible Autonomous Transaction Architecture**，是一套一站式分布式事务解决方案。

Seata 融合了阿里巴巴和蚂蚁金服在分布式事务技术上的积累，并沉淀了新零售、云计算和新金融等场景下丰富的实践经验。

2.5.1 Seata介绍

解决分布式事务问题，有两个设计初衷

对业务无侵入： 即减少技术架构上的微服务化所带来的分布式事务问题对业务的侵入

高性能： 减少分布式事务解决方案所带来的性能消耗

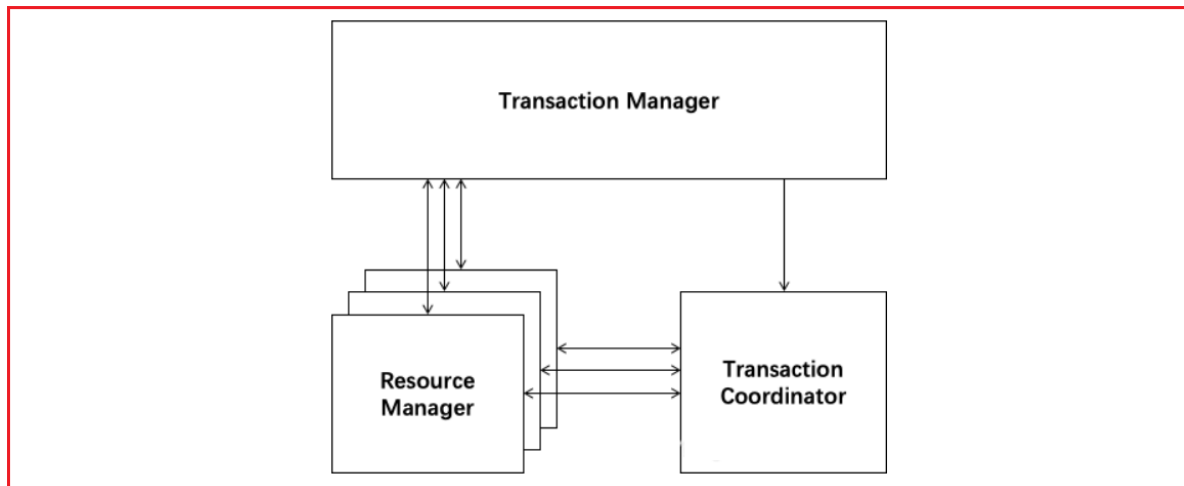
seata中有两种分布式事务实现方案，AT及TCC

- AT模式主要关注多 DB 访问的数据一致性，当然也包括多服务下的多 DB 数据访问一致性问题

- TCC 模式主要关注业务拆分，在按照业务横向扩展资源时，解决微服务间调用的一致性问题

2.5.2 AT模式

Seata AT模式是基于XA事务演进而来的一个分布式事务中间件，XA是一个基于数据库实现的分布式事务协议，本质上和两阶段提交一样，需要数据库支持，Mysql5.6以上版本支持XA协议，其他数据库如Oracle，DB2也实现了XA接口



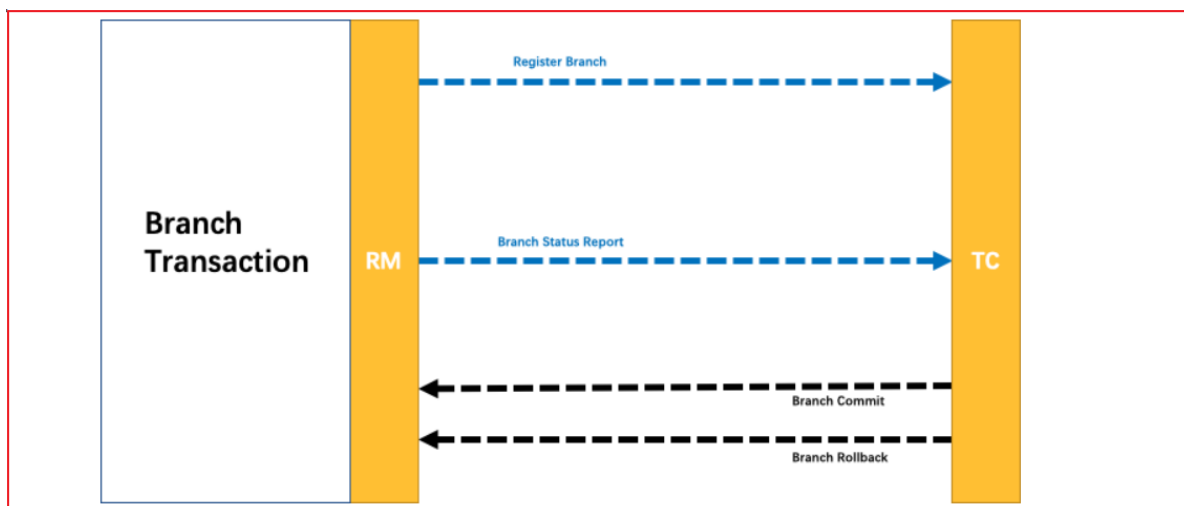
解释：

Transaction Coordinator (TC)： 事务协调器，维护全局事务的运行状态，负责协调并驱动全局事务的提交或回滚。

Transaction Manager (TM)： 控制全局事务的边界，负责开启一个全局事务，并最终发起全局提交或全局回滚的决议。

Resource Manager (RM)： 控制分支事务，负责分支注册、状态汇报，并接收事务协调器的指令，驱动分支（本地）事务的提交和回滚。

协调执行流程如下：

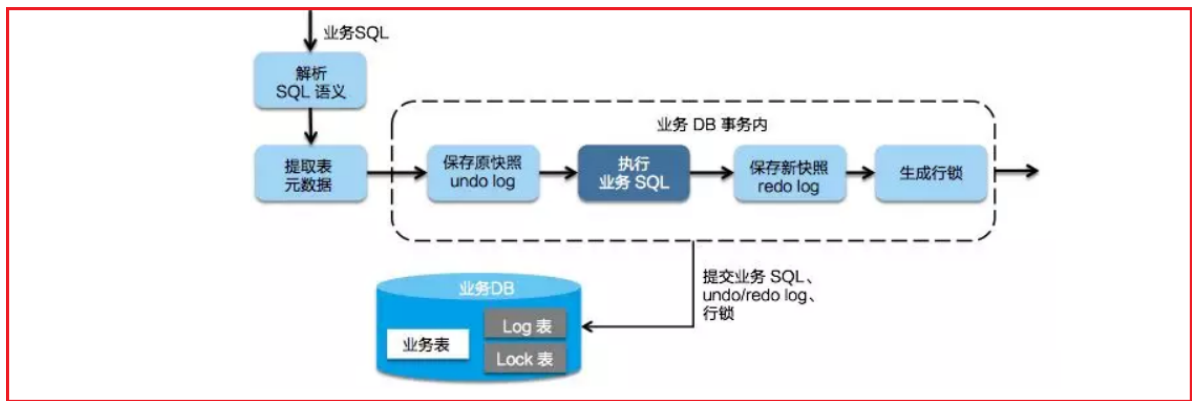


Branch就是指分布式事务中每个独立的本地局部事务。

第一阶段

Seata 的 JDBC 数据源代理通过对业务 SQL 的解析，把业务数据在更新前后的数据镜像组织成回滚日志，利用本地事务的 ACID 特性，将业务数据的更新和回滚日志的写入在同一个本地事务中提交。

这样，可以保证：**任何提交的业务数据的更新一定有相应的回滚日志存在**



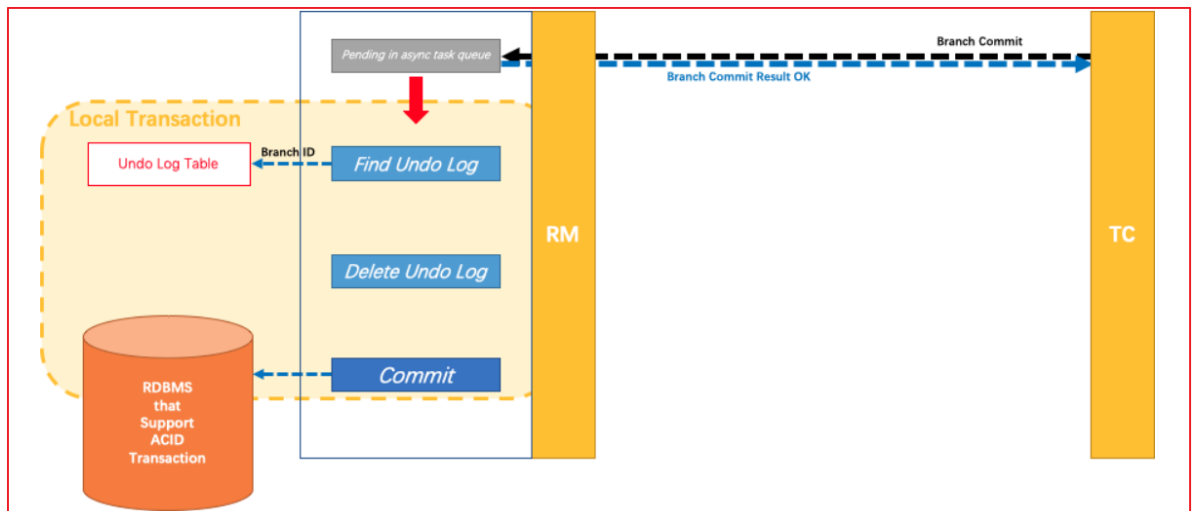
基于这样的机制，分支的本地事务便可以在全局事务的第一阶段提交，并马上释放本地事务锁定的资源

这也是Seata和XA事务的不同之处，两阶段提交往往对资源的锁定需要持续到第二阶段实际的提交或者回滚操作，而有了回滚日志之后，可以在第一阶段释放对资源的锁定，降低了锁范围，提高效率，即使第二阶段发生异常需要回滚，只需找对undolog中对应数据并反解析成sql来达到回滚目的

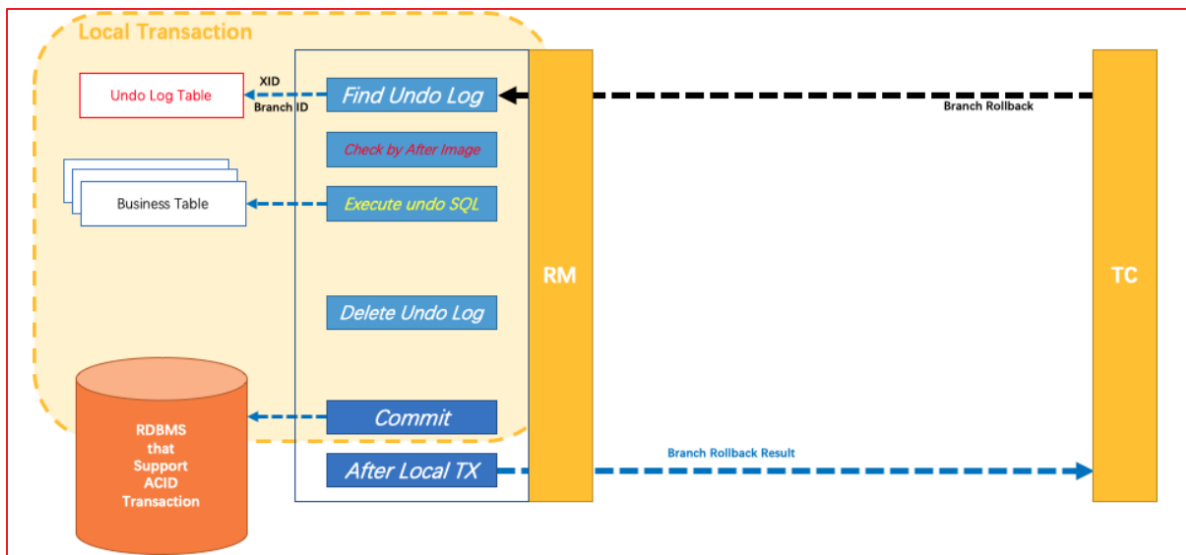
同时Seata通过代理数据源将业务sql的执行解析成undolog来与业务数据的更新同时入库，达到了对业务无侵入的效果。

第二阶段

如果决议是全局提交，此时分支事务此时已经完成提交，不需要同步协调处理（只需要异步清理回滚日志），Phase2 可以非常快速地完成。

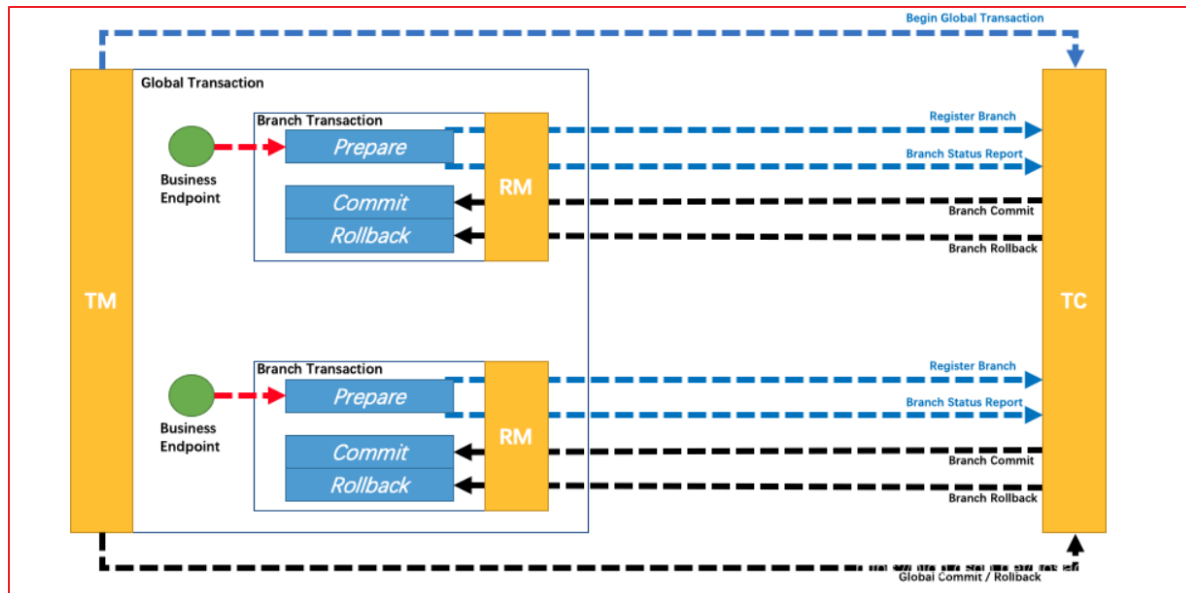


如果决议是全局回滚，RM 收到协调器发来的回滚请求，通过 XID 和 Branch ID 找到相应的回滚日志记录，通过回滚记录生成反向的更新 SQL 并执行，以完成分支的回滚



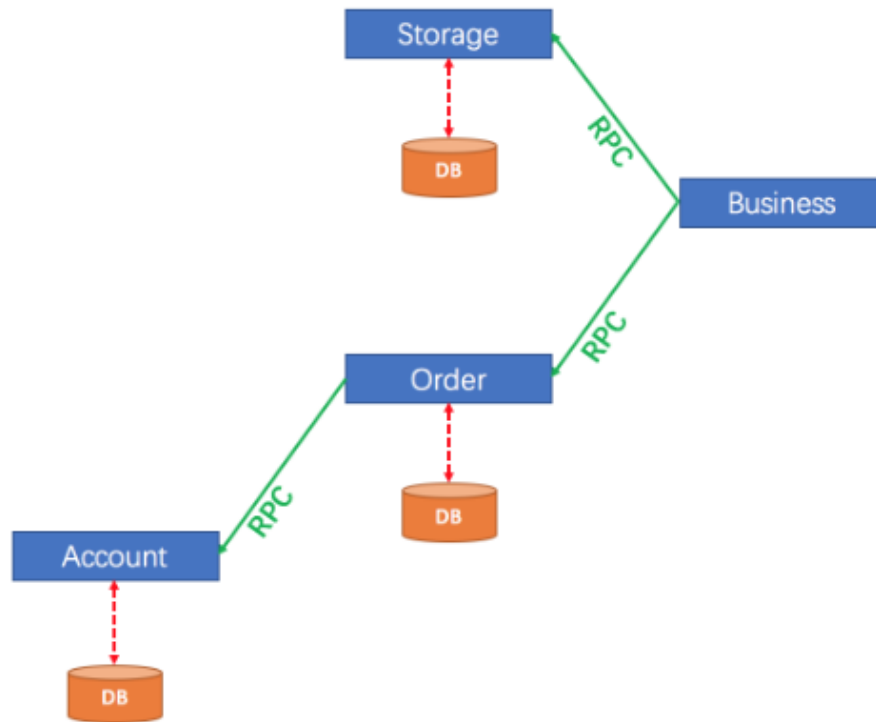
2.5.3 TCC模式

seata也针对TCC做了适配兼容，支持TCC事务方案，原理前面已经介绍过，基本思路就是使用侵入业务上的补偿及事务管理器的协调来达到全局事务的一起提交及回滚。



3 Seata案例

3.1 需求分析

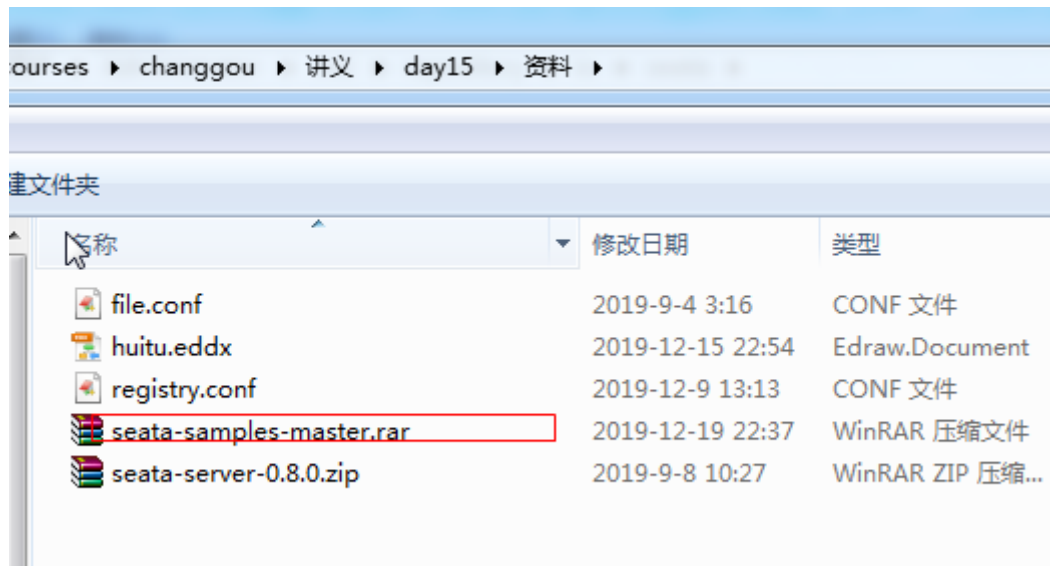


用户购买商品的业务逻辑。整个业务逻辑由3个微服务提供支持：

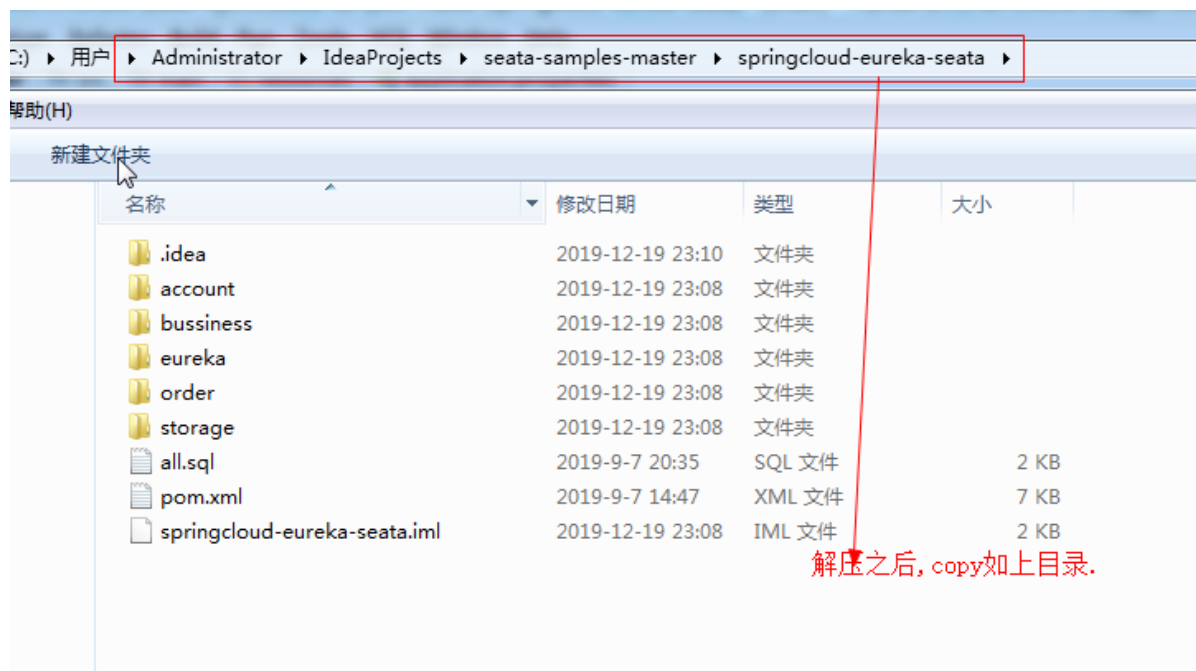
- 仓储服务：对给定的商品扣除仓储数量。
- 订单服务：根据采购需求创建订单。
- 帐户服务：从用户帐户中扣除余额。

3.2 案例实现

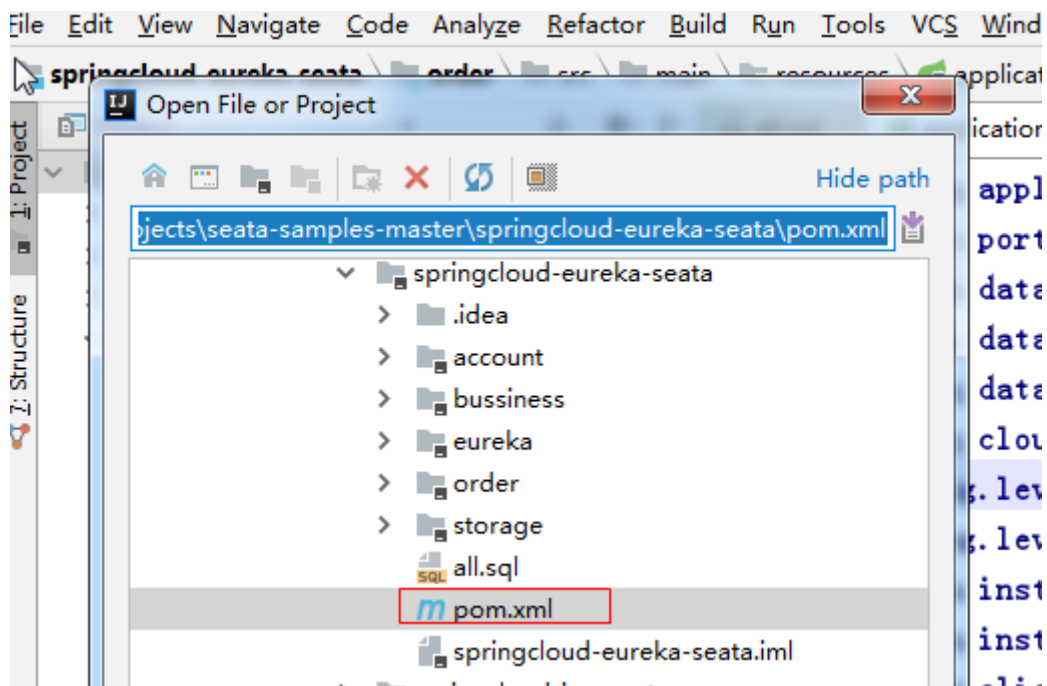
参考如下图所示的工程即可。



3.2.1 解压导入

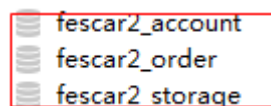


打开idea,在idea中打开项目,



选择 open as a project.即可.

3.2.2 创建3个数据库



每一个数据库都需要导入如下的sql脚本:

```
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
```

```

`context` varchar(128) 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`),
UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

account数据库执行:

```

CREATE TABLE `account_tbl` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` varchar(255) DEFAULT NULL,
  `money` int(11) DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
INSERT INTO `fescar2_account`.`account_tbl` (`id`,`user_id`,`money`) VALUES
('1', 'u100000', '10000');

```

order数据库执行:

```

CREATE TABLE `order_tbl` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` varchar(255) DEFAULT NULL,
  `commodity_code` varchar(255) DEFAULT NULL,
  `count` int(11) DEFAULT '0',
  `money` int(11) DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

storage数据库执行:

```

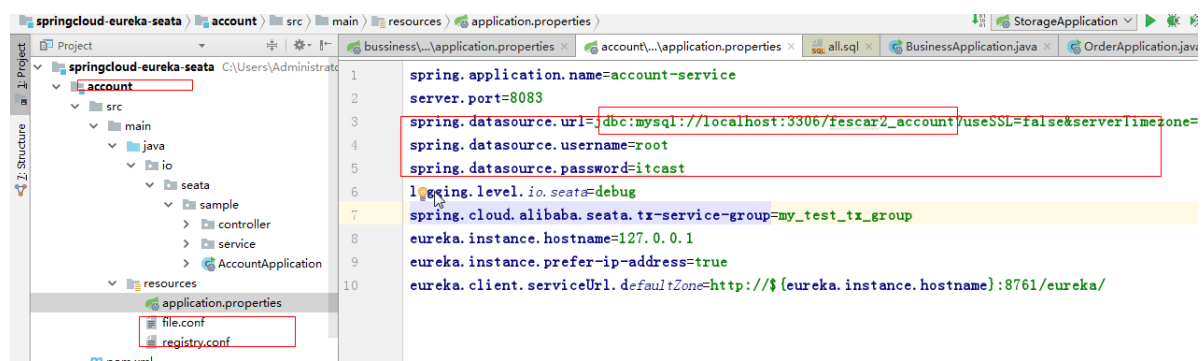
CREATE TABLE `storage_tbl` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `commodity_code` varchar(255) DEFAULT NULL,
  `count` int(11) DEFAULT '0',
  PRIMARY KEY (`id`),
  UNIQUE KEY `commodity_code` (`commodity_code`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

INSERT INTO `fescar2_storage`.`storage_tbl` (`id`,`commodity_code`,`count`)
VALUES ('1', 'c100000', '200');

```

3.2.3分析

account:



创建配置文件,和配置,如上图,设置连接的数据库



其他的也是类似.这里不再赘述.

bussines:

这个微服务不需要连接数据源,做业务处理,为全局事务的入口.需要在开启全局事务的方法上添加如下图的注解.



测试即可.

registry.conf配置文件说明:

```
registry {
  # file 、nacos 、eureka、redis、zk
  # 设置配置使用文件的方式进行注册
  type = "file"

  nacos {
    serverAddr = "localhost"
    namespace = "public"
    cluster = "default"
  }
  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 = "file"

  nacos {
    serverAddr = "localhost"
    namespace = "public"
    cluster = "default"
  }
  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"
  }
}
```

file.conf:

```

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 = "default"
  #only support single node
  default.grouplist = "127.0.0.1:8091"
  #degrade current not support
  enableDegrade = false
  #disable
  disable = false
  disableGlobalTransaction = false
}

client {
  async.commit.buffer.limit = 10000
  lock {
    retry.internal = 10
    retry.times = 30
  }
}
}

```

更多参数说明参考如下地址:





<http://seata.io/zh-cn/docs/user/configurations.html>

4 分布式事务集成到黑马头条

4.1 搭建seata服务器在linux系统（虚拟机已经有了）

(1) 从官网下载seata server端的程序包

下载地址: <https://github.com/seata/seata/releases>

| | |
|--|---------|
|  seata-server-1.3.0.tar.gz | 39.9 MB |
|  seata-server-1.3.0.zip | 39.9 MB |
|  Source code (zip) | |
|  Source code (tar.gz) | |

(2)上传到linux 系统进行解压到如图所示目录

```
drwxr-xr-x. 5 root root 55 7月 16 00:35 seata
[root@localhost server]# pwd
/usr/local/server
[root@localhost server]#
```

解压命令为如下：

```
-rw-r--r--. 1 root root 1351 11月 19 07:03 anaconda-ks.cfg
-rw-r--r--. 1 root root 185515842 2月 17 2019 jdk-8u144-linux-x64.tar.gz
drwxr-xr-x. 3 root root 19 11月 25 00:05 logs
-rw-r--r--. 1 root root 52999735 6月 3 20:02 nacos-server-1.2.1.tar.gz
-rw-r--r--. 1 root root 41888244 11月 24 15:46 seata-server-1.3.0.zip
drwxr-xr-x. 3 root root 70 11月 25 06:18 sh
[root@localhost ~]# unzip seata-server-1.3.0.zip -d /usr/local/server/
```

```
unzip seata-server-1.3.0.zip -d /usr/local/server/
```

(3) 编写脚本执行启动seata-server（创建目录，创建脚本，给脚本设置值）

创建目录：

```
mkdir -p /usr/local/server/logs
```

```
-rwxr-xr-x. 1 root root 99 11月 25 06:17 start_seata.sh
[root@localhost sh]# C
```

脚本内容为：

```
nohup sh /usr/local/server/seata/bin/seata-server.sh -p 8091 &>
/usr/local/server/logs/seata.log &
```

(4) 启动脚本

```
sh /root/sh/start_seata.sh
```

4.2 集成seata

4.2.1 需求分析

需要在每一个需要用到的微服务中，按照刚才的案例进行配置：如下分析如果需要用到分布式事务都需要添加一个表：undo_log表，创建表语句如下

```
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `context` varchar(128) NOT NULL,
  `rollback_info` longblob NOT NULL,
  `log_status` int(11) NOT NULL,
  `log_created` datetime NOT NULL,
  `log_modified` datetime NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

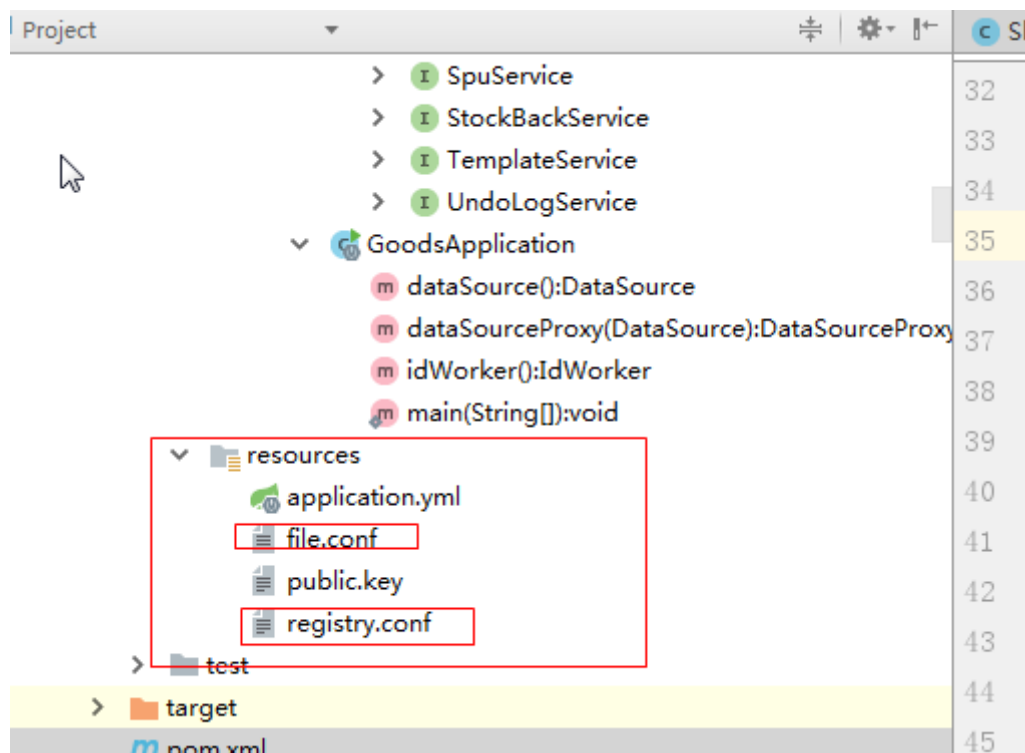
4.2.2 添加依赖

商品微服务 订单微服务 用户微服务都添加如下依赖:






```
<dependency>
  <groupId>io.seata</groupId>
  <artifactId>seata-all</artifactId>
  <version>0.8.0</version>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.21</version>
</dependency>
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-alibaba-seata</artifactId>
  <version>2.1.0.RELEASE</version>
  <exclusions>
    <exclusion>
      <groupId>io.seata</groupId>
      <artifactId>seata-all</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

4.2.3 添加配置文件

每一个微服务都需要添加如下红色框的2个文件:



如上图的文件参考下图:

| es > 黑马头条 > 黑马头条讲义 > day14 > 资料 > | |
|--|----------|
| 名称 | 修改日期 |
|  file.conf | 2019/9/4 |
|  huitu.eddx | 2019/12 |
|  registry.conf | 2019/12 |
|  seata-samples-master.rar | 2020/8/1 |
|  seata-server-0.8.0.zip | 2020/9/1 |

4.2.4 添加配置yml

每一个微服务都需要添加如下配置:

```
spring:
  cloud:
    alibaba:
      seata:
        tx-service-group: my_test_tx_group
```

4.2.5 添加配置类

每一个微服务的引导类中添加如下配置,用于设置代理数据源:

```

@Bean
@ConfigurationProperties(prefix = "spring.datasource")
public DataSource dataSource() {
    DruidDataSource druidDataSource = new DruidDataSource();
    return druidDataSource;
}

@Primary
@Bean("dataSourceProxy")
public DataSourceProxy dataSourceProxy(DataSource dataSource) {
    return new DataSourceProxy(dataSource);
}

```

4.2.6 service方法添加注解

如图,在订单微服务中订单创建的全局事务的方法上,添加如图所示的注解即可.

```

@Override
@Transactional(rollbackFor = {Exception.class})
@GlobalTransactional
public void pass(Integer id) {
    ApUserRealname entity = new ApUserRealname();
}

```

4.2.7测试

略。

4.3 优化抽取

由上边我们分析就知道, 如果需要用到seata每一个都需要设置相关的配置非常麻烦, 而我们已经学过自定义starter,那么我们可以自定义一个starter叫做: itheima-leadnews-core-seata, 在微服务使用的时候直接添加依赖即可, 简单配置就好了,不需要大量的进行配置.

面包屑

显示/隐藏

头条 > 优化升级版本 > 资料 > seata > 参考导入工程 >

| 名称 | 修改日期 | 类型 |
|---|-----------------|-----|
|  itheima-leadnews-core-seata | 2021/2/26 15:06 | 文件夹 |

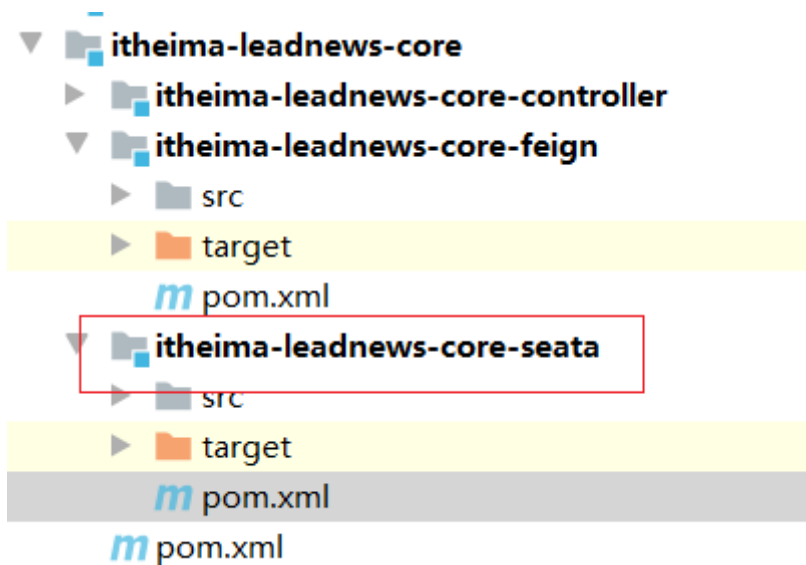
使用步骤:

0. 将之前的搭建方式全部撤销掉。
1. 在需要使用到seata的微服务中添加自定义起步依赖 并对应的数据库中创建und_log表
2. 在启动类中 排除掉datasourceAutoConfiguration
3. 在需要使用到的业务层的方法上添加注解@GlobalTransactional以及本地spring事务的注解

(0) 通用的每一个微服务对应的数据库都需要一个脚本:

```
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `context` varchar(128) NOT NULL,
  `rollback_info` longblob NOT NULL,
  `log_status` int(11) NOT NULL,
  `log_created` datetime NOT NULL,
  `log_modified` datetime NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

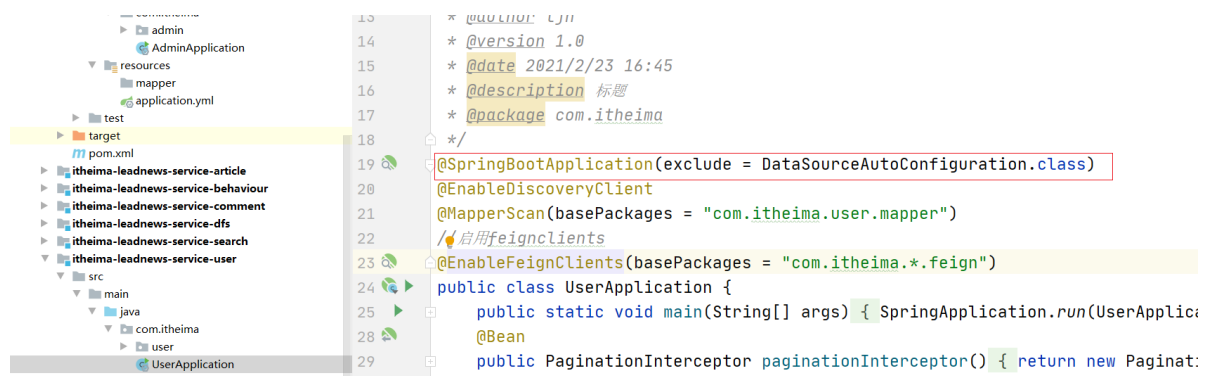
(1) 导入之后如图:



(2)在需要使用到微服务中添加如下依赖:



(3)并且在 微服务启动类中添加如下配置:



(4)在业务层方法上添加注解即可

```

@Override
@Transactional(rollbackFor = {Exception.class})
@GlobalTransactional
public void pass(Integer id) {
    ApUserRealname entity = new ApUserRealname();

```

(5) linux启动seata服务端，并进行测试