Seata技术应用方案

# 一、什么是Seata

Seata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。

# 二、什么是微服务化带来的分布式事务问题

1. 单体应用中连接统一数据源完成业务，这样使用本地事务来保证数据的一致性。

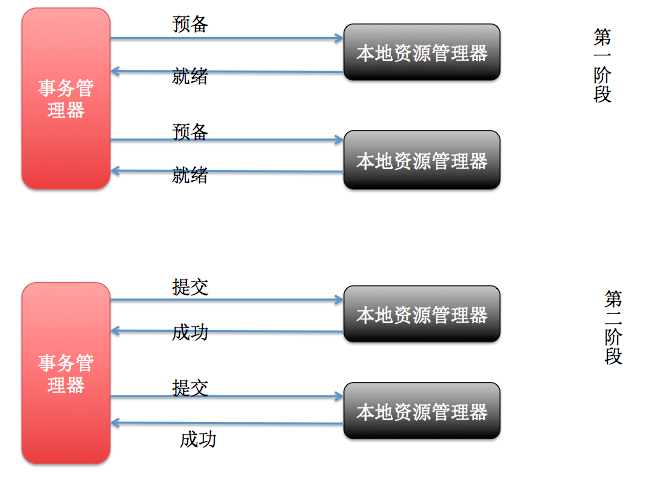
2. 单体应用被拆分为微服务，每一个微服务使用独立的数据源。这种情况下，每一个微服务的数据一致性仍可以用本地事务解决。但是，整个业务层面的全局数据一致性没有办法保证。

所以我们需要一个分布式事务的解决方案保障业务全局的数据一致性。

# 三、现有的解决方案

## 1、业务无侵入方案

XA是一个分布式事务协议，XA中大致分为两部分：事务管理器和本地资源管理器。实现分布式事务的原理如下：



可以解决问题，但是存在着以下问题：

①要求数据库提供对 XA 的支持。如果遇到不支持 XA（或支持得不好，比如 MySQL 5.7 以前的版本）的数据库，则不能使用。

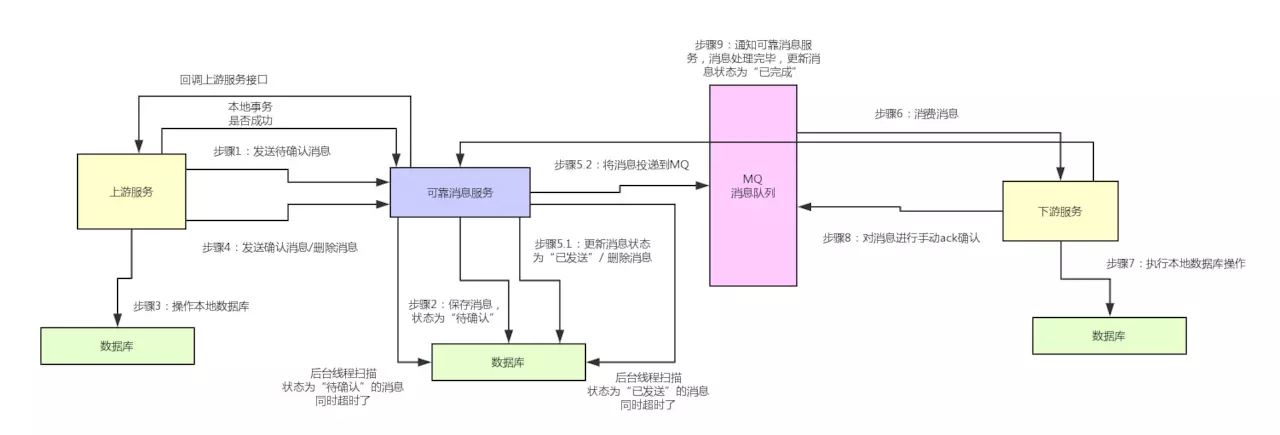
②受协议本身的约束，事务资源（数据记录、数据库连接）的锁定周期长。长周期的资源锁定从业务层面来看，往往是不必要的，而因为事务资源的管理器是数据库本身，应用层无法插手。这样形成的局面就是，基于 XA 的应用往往性能会比较差，而且很难优化。

③已经落地的基于 XA 的分布式解决方案，都依托于重量级的应用服务器（Tuxedo/WebLogic/WebSphere 等)，这是不适用于微服务架构的。

## 2、业务侵入方案

实现方式主要有：

（1）基于可靠消息的最终一致性方案



①上游服务发送消息到可靠消息服务

上游服务告诉可靠消息服务保存一条待确认的消息，然后就执行业务逻辑操作本地数据库，如果业务操作成功就告诉可靠消息服务更新消息为“已发送”并投递消息到MQ；如果业务操作失败了，就告诉可靠消息服务删除之前的“待确认”状态的消息。

如果操作本地数据库成功了（步骤3），但通知可靠消息服务更新消息状态（步骤4）时失败了怎么办？

步骤3、步骤4，置于同一本地事务中，保证一致性。如果想做的更好些，可以在可靠消息服务开一个后台线程运行定时任务，检查状态为“待确认”且已过期的消息，询问上游服务相关业务操作是否成功了，如果成功就更新消息状态为“已发送”同时投递消息到MQ；否则删除消息。

②可靠消息服务投递消息到MQ

在可靠消息服务中更新消息为“已发送”和将消息投递到MQ这两个操作要放到一个本地事务中，保证：

如果数据库里更新消息的状态失败了，那么就抛异常退出了，就别投递到MQ；

如果投递MQ失败报错了，那么就要抛异常让本地数据库事务回滚。

这俩操作必须得一起成功，或者一起失败。

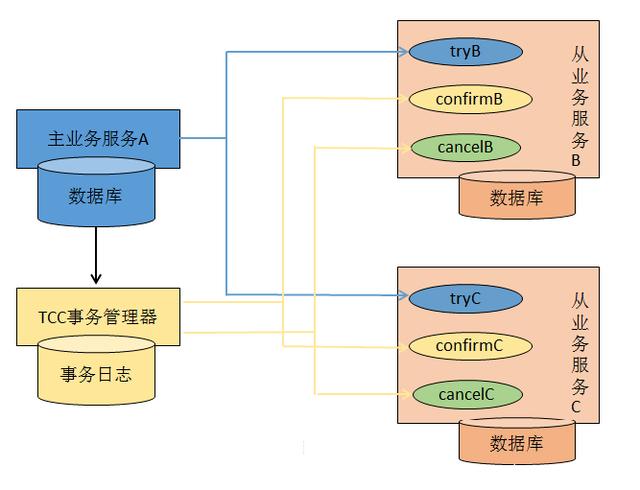
③下游服务消费MQ消息

下游服务消费MQ消息后，执行业务操作，如果成功了告知MQ消费成功，并告知可靠消息服务更新消息状态为“已完成”。

那如果下游服务消费消息出了问题，没消费到？或者是下游服务对消息的处理失败了，怎么办？

仍然可以在可靠消息服务后台开一个线程执行定时任务，检查状态为“已发送”且超时的消息，再次将此消息投递到MQ让下游消费，需要下游保证消费消息的幂等性。

（2）TCC



TCC的全程主要包括Try（尝试）、Confirm（确认）、Cancel（取消）这三个主要阶段。

* Try 阶段：主要是对业务系统做检测及资源预留
* Confirm 阶段：主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。
* Cancel 阶段：主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

TCC方案让应用自己定义数据库操作的粒度，使得降低锁冲突、提高吞吐量成为可能。 当然TCC方案也有不足之处，集中表现在以下两个方面：

①对应用的侵入性强。业务逻辑的每个分支都需要实现try、confirm、cancel三个操作，应用侵入性较强，改造成本高。

②实现难度较大。需要按照网络状态、系统故障等不同的失败原因实现不同的回滚策略。为了满足一致性的要求，confirm和cancel接口必须实现幂等。

这些方案都要求在应用的业务层面把分布式事务技术约束考虑到设计中，通常每一个服务都需要设计实现正向和反向的幂等接口。这样的设计约束，往往会导致很高的研发和维护成本。

# 四、技术选型

要求像使用本地事务一样简单，业务逻辑只关注业务层面的需求，不需要考虑事务机制上的约束。

Seata组件的AT模式由XA模式演进而来，是一个业务无侵入的解决方案，既解决了XA模式的性能问题，同时也避免了对业务的入侵

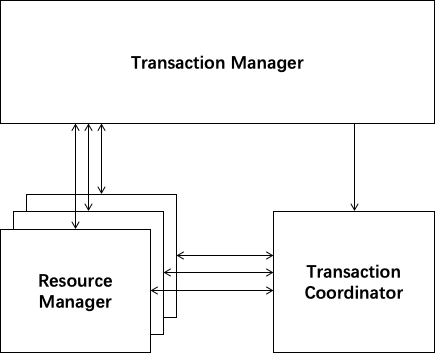
# 五、方案对比

显然，业务无侵入方案对比业务入侵方案有明显优势，可以在对业务无改动或者很少改动的情况下加入对全局事务的支持。这里将XA模式和Seata的AT 模式进行对比。

## 1、相同点：

基本认识相同：分布式事务是包含了若干分支事务的全局事务。全局事务 的职责是协调其下管辖的 分支事务 达成一致，要么一起成功提交，要么一起失败回滚。此外，通常 分支事务 本身就是一个满足 ACID 的 本地事务。

模型相同：



* 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。

过程相同：

一个典型的分布式事务过程：

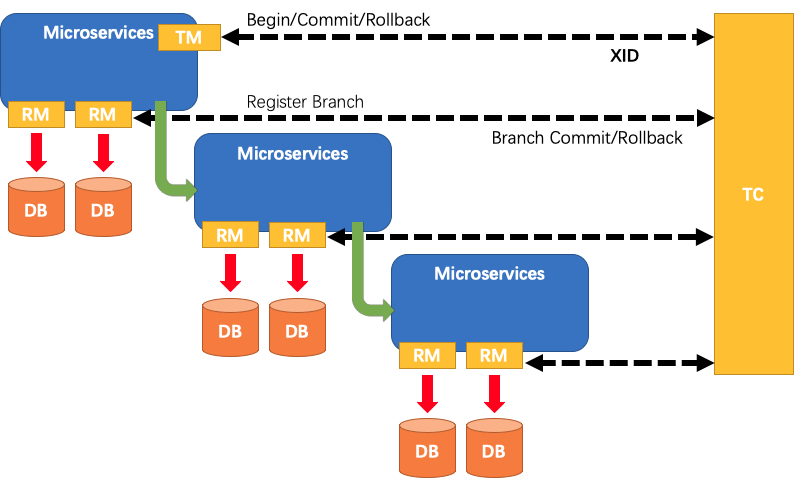
（1）TM 向 TC 申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的 XID。

（2）XID 在微服务调用链路的上下文中传播。

（3）RM 向 TC 注册分支事务，将其纳入 XID 对应全局事务的管辖。

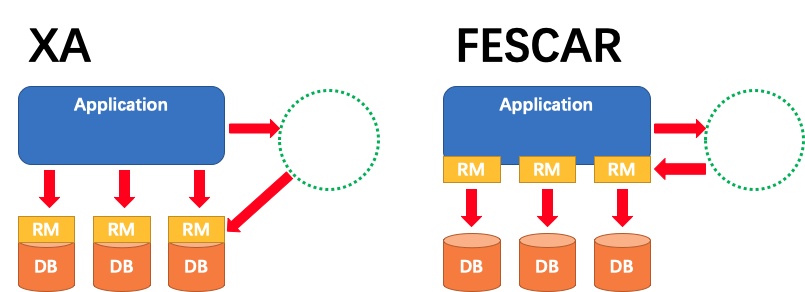
（4）TM 向 TC 发起针对 XID 的全局提交或回滚决议。

（5）TC 调度 XID 下管辖的全部分支事务完成提交或回滚请求。



## 2、不同点：

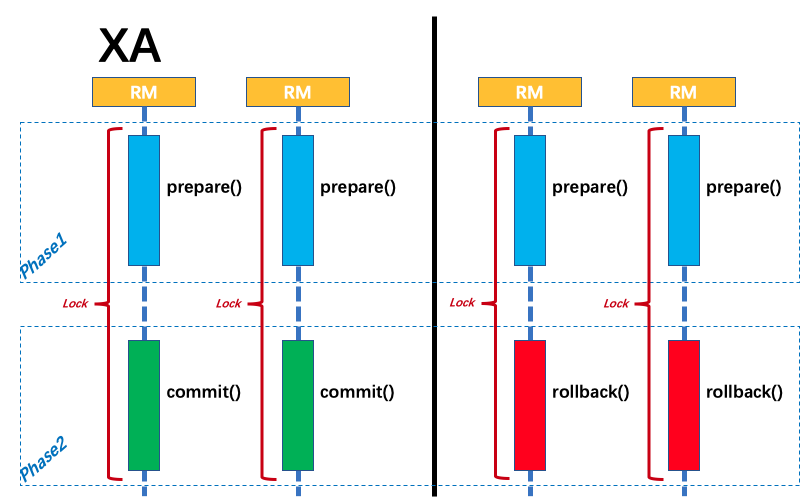
（1）架构层次不同：

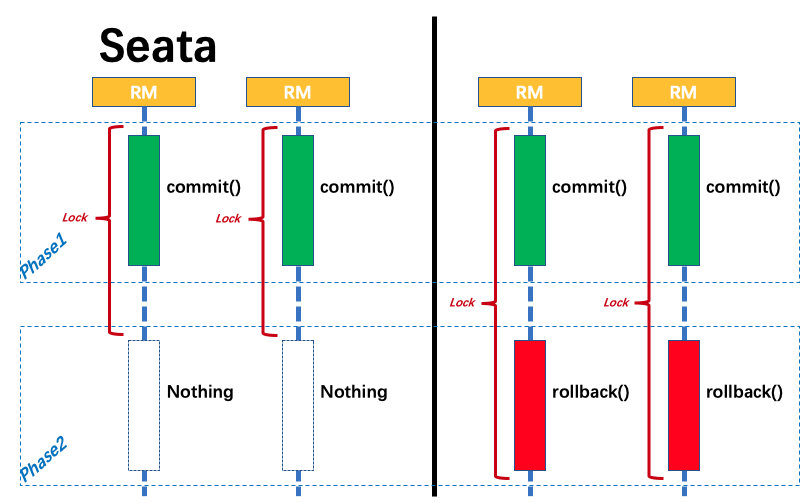


*Fescar 是Seata 的前身*

XA 方案的 RM 实际上是在数据库层，RM 本质上就是数据库自身（通过提供支持 XA 的驱动程序来供应用使用）。而 Fescar 的 RM 是以二方包的形式作为中间件层部署在应用程序这一侧的，不依赖与数据库本身对协议的支持，当然也不需要数据库支持 XA 协议。这点对于微服务化的架构来说是非常重要的：应用层不需要为本地事务和分布式事务两类不同场景来适配两套不同的数据库驱动。这个设计，剥离了分布式事务方案对数据库在 协议支持 上的要求。

（2）两阶段提交过程不同：





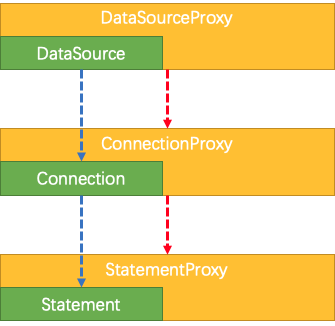
XA无论 Phase2 的决议是 commit 还是 rollback，事务性资源的锁都要保持到 Phase2 完成才释放。

* Seata分支事务中数据的本地锁由本地事务管理，在分支事务Phase1结束时释放。
* 同时，随着本地事务结束，连接 也得以释放。
* 分支事务中数据的 全局锁 在事务协调器侧管理，在决议 Phase2 全局提交时，全局锁马上可以释放。只有在决议全局回滚的情况下，全局锁 才被持有至分支的 Phase2 结束。

这个设计，极大地减少了分支事务对资源（数据和连接）的锁定时间，给整体并发和吞吐的提升提供了基础。

# 六、工作原理

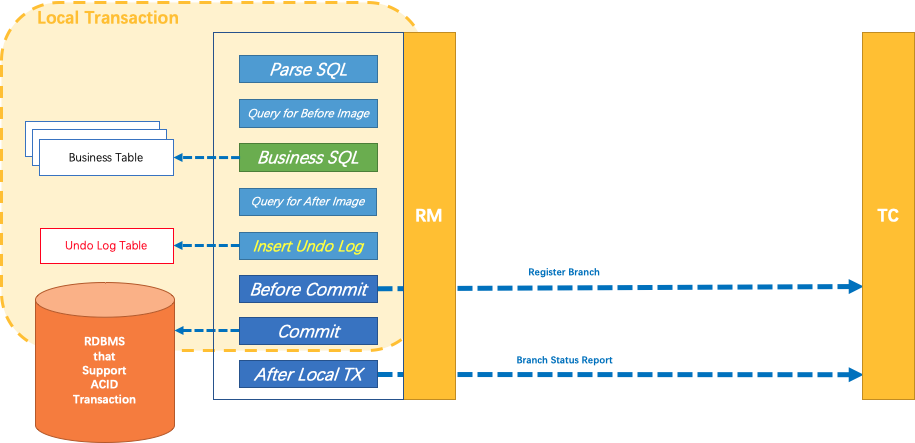
首先，应用需要使用Seata的 JDBC 数据源代理，也就是 Seata的 RM。



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。

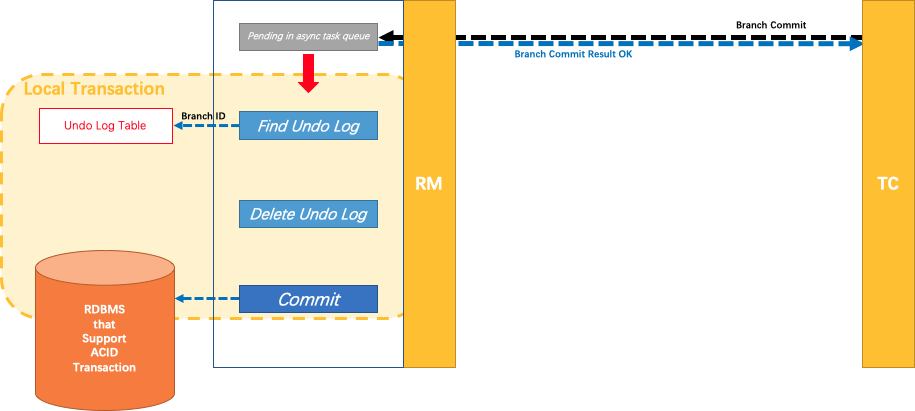
这样，可以保证：任何提交的业务数据的更新一定有相应的回滚日志存在。



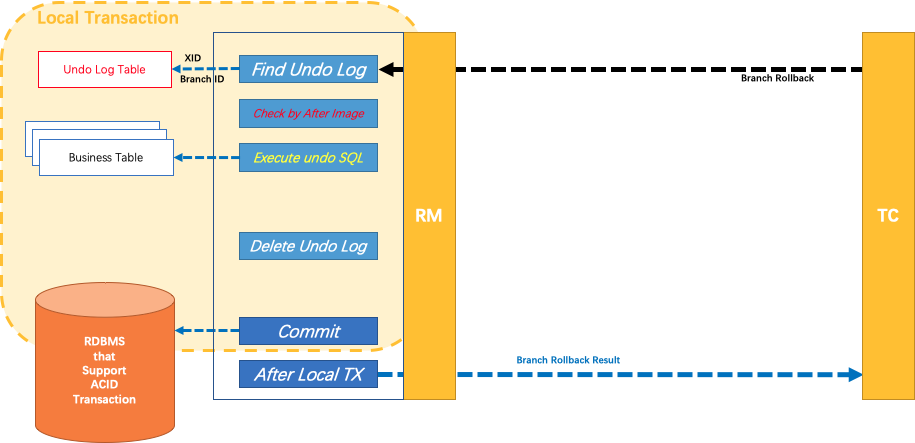
基于这样的机制，分支的本地事务便可以在全局事务的一阶段提交，马上释放本地事务锁定的资源。

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），二阶段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。



# 七、隔离性

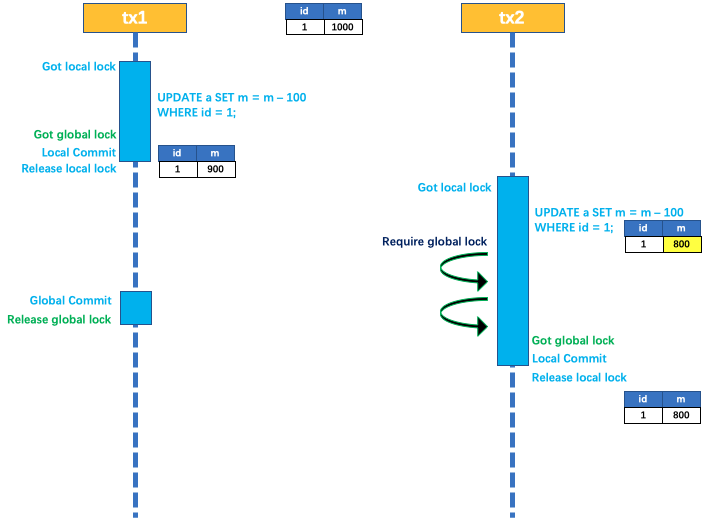
## 写隔离

* 一阶段本地事务提交前，需要确保先拿到 全局锁 。
* 拿不到 全局锁 ，不能提交本地事务。
* 拿 全局锁 的尝试被限制在一定范围内，超出范围将放弃，并回滚本地事务，释放本地锁。

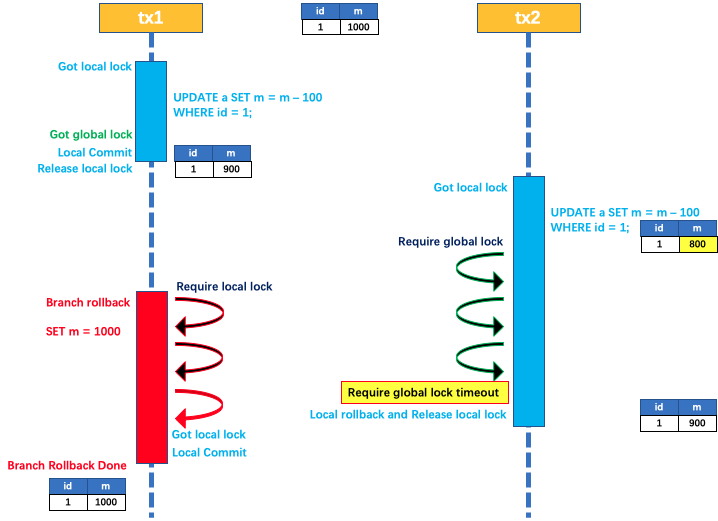
以一个示例来说明：

两个全局事务 tx1 和 tx2，分别对 a 表的 m 字段进行更新操作，m 的初始值 1000。

tx1 先开始，开启本地事务，拿到本地锁，更新操作 m = 1000 - 100 = 900。本地事务提交前，先拿到该记录的全局锁 ，本地提交释放本地锁。 tx2 后开始，开启本地事务，拿到本地锁，更新操作 m = 900 - 100 = 800。本地事务提交前，尝试拿该记录的全局锁 ，tx1全局提交前，该记录的全局锁被 tx1 持有，tx2 需要重试等待全局锁 。



tx1 二阶段全局提交，释放 全局锁 。tx2 拿到 全局锁 提交本地事务。



如果 tx1 的二阶段全局回滚，则 tx1 需要重新获取该数据的本地锁，进行反向补偿的更新操作，实现分支的回滚。

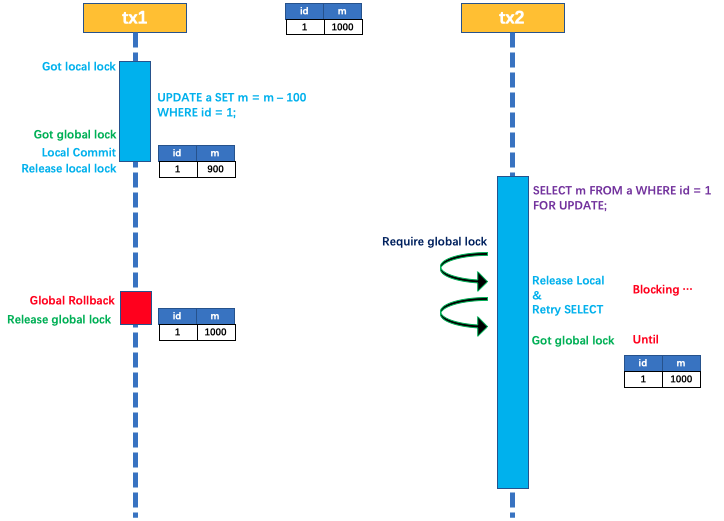
此时，如果 tx2 仍在等待该数据的 全局锁，同时持有本地锁，则 tx1 的分支回滚会失败。分支的回滚会一直重试，直到 tx2 的 全局锁 等锁超时，放弃 全局锁 并回滚本地事务释放本地锁，tx1 的分支回滚最终成功。

因为整个过程 全局锁 在 tx1 结束前一直是被 tx1 持有的，所以不会发生 脏写 的问题。

## 读隔离

在数据库本地事务隔离级别 读已提交（Read Committed） 或以上的基础上，Seata（AT 模式）的默认全局隔离级别是 读未提交（Read Uncommitted） 。

如果应用在特定场景下，必需要求全局的 读已提交 ，目前 Seata 的方式是通过 SELECT FOR UPDATE 语句的代理。



SELECT FOR UPDATE 语句的执行会申请 全局锁 ，如果 全局锁 被其他事务持有，则释放本地锁（回滚 SELECT FOR UPDATE 语句的本地执行）并重试。这个过程中，查询是被 block 住的，直到 全局锁 拿到，即读取的相关数据是 已提交 的，才返回。

出于总体性能上的考虑，Seata 目前的方案并没有对所有 SELECT 语句都进行代理，仅针对 FOR UPDATE 的 SELECT 语句。

# 八、测试

按照Seata 执行步骤对可能发生的意外情况进行测试，验证Seata 在意外情况下的处理方式。

**整体测试环境说明**：

1、测试采用本地测试，测试项目为spring-cloud-alibaba-seata-master

2、启动两台TC，分别位于localhost:8092和lcoalhost:8093

3、项目中 order-service作为TM，通过feign 接口调用 user-service 和 storage-service两个服务的接口

4、user-service 和 storage-service作为RM,分别进行和两个数据库进行交互，模拟分布式情景。

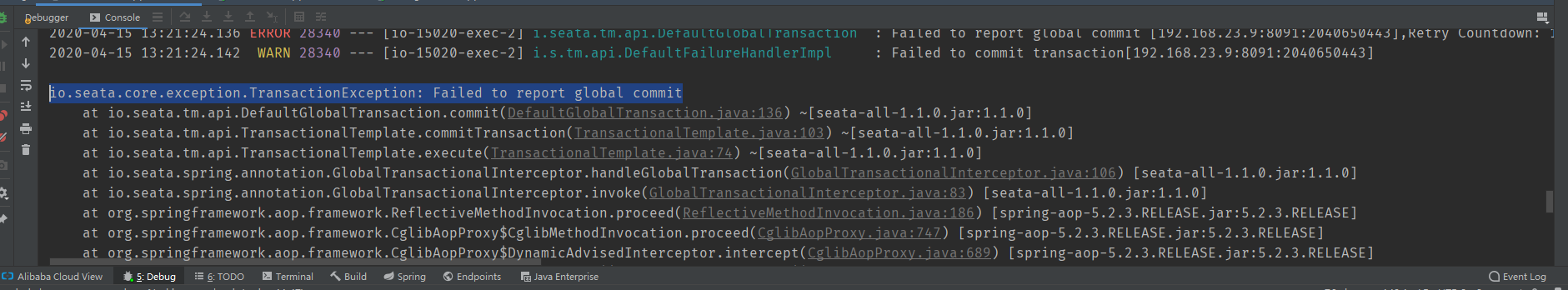
## 测试阶段1：

按照seata整体执行机制，tm需要向tc服务器申请全局事务，申请成功后会在tc服务器对应的数据库存储中创建生成全局唯一的XID对应的记录，对全局事务id创建过程拟测试TM宕机、一个TC宕机、所有TC宕机等几个场景，具体测试步骤及结果如下：

* TM宕机

测试方式：本地进行测试，使用编辑器在事务开启之前添加断点，请求接口，代码执行到断点后关闭TM服务。

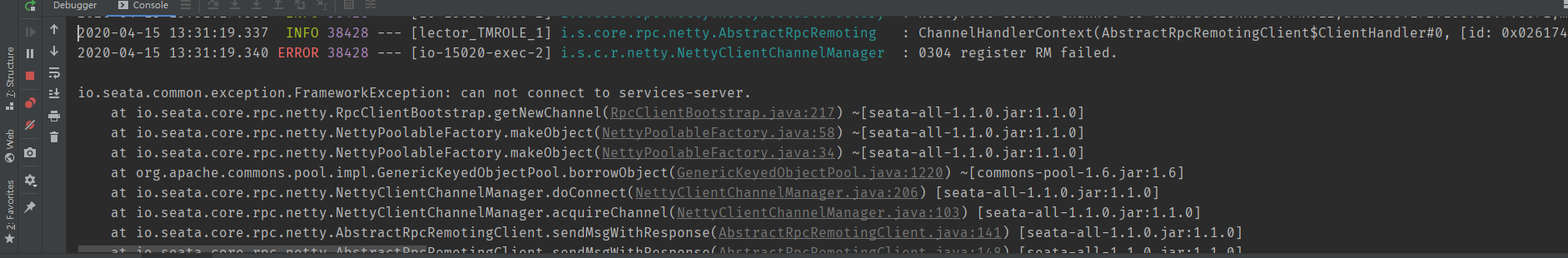
结论:客户端报错io.seata.core.exception.TransactionException: Failed to report global commit,全局事务未开启，TC未打印XID,各数据库数据没有改变。



* 一个TC宕机：

测试方式：代码执行到断点，关闭其中一个TC的进程

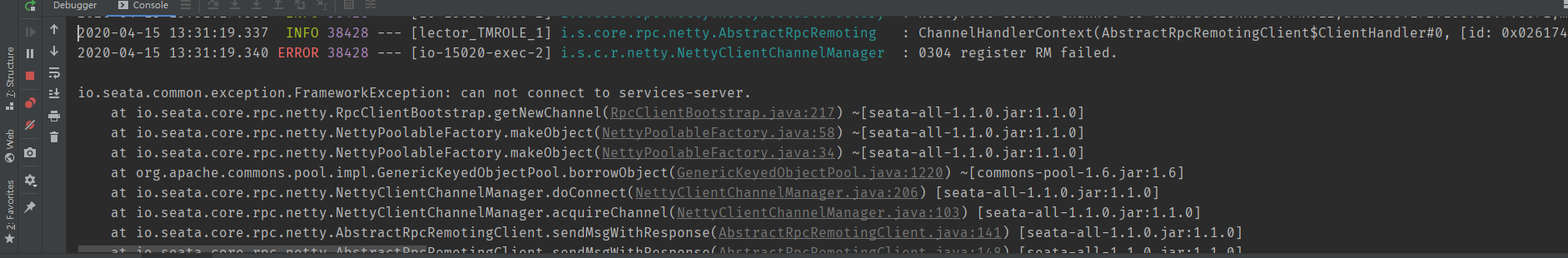
结论:客户端报错io.seata.common.exception.FrameworkException: can not connect to services-server. 全局事务未受到影响，另外一个TC 负责发放XID，全局事务提交，业务数据正常改变



* 所有TC宕机

测试方式：代码执行到断点，关闭所有TC的进程

结论:客户端报错io.seata.common.exception.FrameworkException: can not connect to services-server. ,程序终止,各数据库数据没有改变



## 测试阶段2：

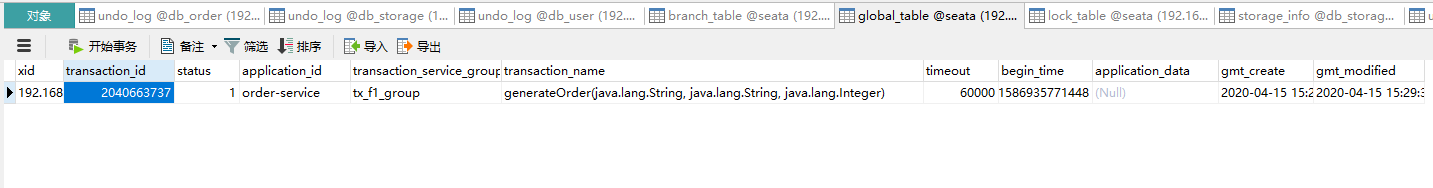
按照Seata 事务的执行流程，接下来TM需要向TC注册分支事务，将每一个分支事务纳入该全局职务ID的管辖范围内，该阶段情况较为复杂，分为RM注册前、其中一个RM注册后另一个RM注册前两种前提下，RM、TM、TC三种角色发生的意外情况。

2、RM 向 TC 注册分支事务，将其纳入 XID 对应全局事务的管辖。

* RM分支事务未注册到TC，TM宕机

测试过程：开启全局事务后，进行分支事务注册之前，将TM服务进程停止

结论：程序终结，业务数据没有变化，global\_table 表中出现一条数据



TM 重启后数据被清除

* RM分支事务未注册到TC，其中一个TC宕机

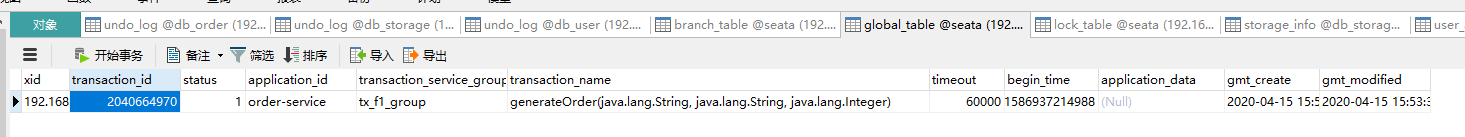
测试过程：开启全局事务后，进行分支事务注册之前，将其中一个TC服务进程停止

结论：剩余的TC完成RM分支事务的注册，并提交全局事务

* RM分支事务未注册到TC，所有TC宕机

测试过程：开启全局事务后，进行分支事务注册之前，将所有TC服务进程停止

结论：客户端报错io.seata.common.exception.FrameworkException: can not connect to services-server. ,程序终止，global\_table 表中出现一条数据



任意一个TC重启后数据被清除

* 分支事务A注册到TC，分支事务B未注册，分支事务B所在RM宕机

测试过程：开启全局事务后，分支事务A注册到TC，程序进入断点，然后停止分支事务B所在的服务。

结论：全局事务和分支事务全部回滚，业务数据没有改变

* 分支事务A注册到TC，分支事务B未注册，分支事务A所在RM宕机

测试过程：开启全局事务后，分支事务A注册到TC，程序进入断点，然后停止分支事务A所在的服务。

结论： 分支事务提交，分支事务A操作的业务数据库数据改变，全局事务未提交，对应的undo表中和seata数据库中保留事务信息。TC不断尝试和停掉的RM进行连接。RM重启后全局事务提交，业务数据正常被修改。

* 分支事务A注册到TC，其中一个TC宕机

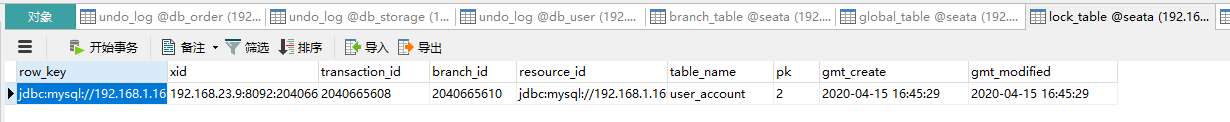
测试过程：开启全局事务后，分支事务A注册到TC，程序进入断点，然后停止其中一个TC服务进程。

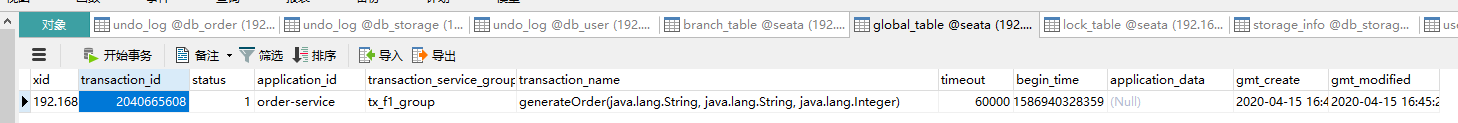
结论：全局事务和分支事务在剩余TC服务的协调下全部提交。

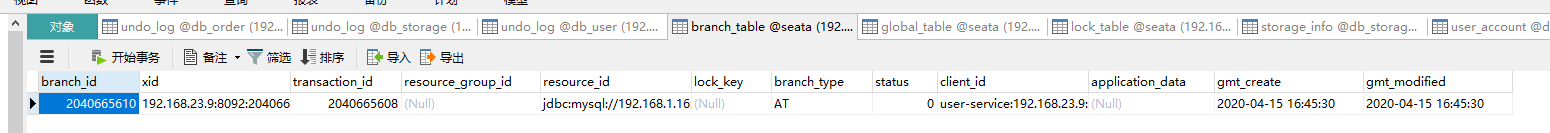
* 分支事务A注册到TC，所有TC宕机

测试过程：开启全局事务后，分支事务A注册到TC，程序进入断点，然后停止所有TC服务进程。

结论：业务数据库出现脏数据的情况，Seata相关数据库表记录了回滚信息。





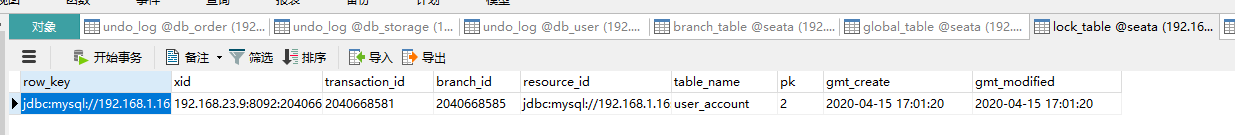
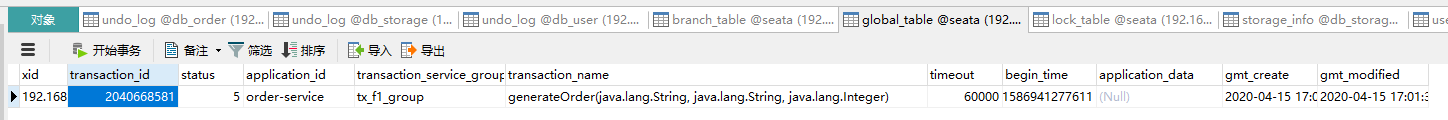
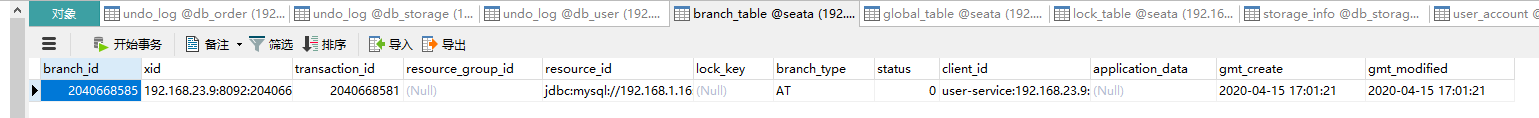
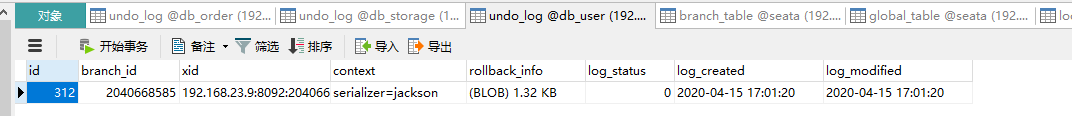


启动任意一个TC 节点后数据回滚成功

* 分支事务A注册到TC，TM报错，在回滚过程中分支事务A所在的服务宕机。

测试过程：开启全局事务后，分支事务A注册到TC，程序在TM报错之前进入断点，然后停止分支事务A所在的服务，放开断点执行。

结论：业务数据库出现脏数据，Seata相关数据库表记录了回滚信息。

Seata服务端以一秒一次的频率持续请求连接RM，服务重启后业务数据回滚

## 测试阶段3：

当RM分支事务处理完成后，由TM向TC依据XID向TC发起全局事务提交或者回滚决议，最后由TC电镀XID管辖的分支事务完成提交或回滚，该阶段对TM宕机，一个TC宕机，所有TC宕机等情况进行测试。

* 全局事务已开启、分支事务全部注册到TC，TM 宕机：

测试过程：开启全局事务后，所有分支事务注册到TC，程序进入断点，然后停止TM服务进程。

结论：全局提交，业务数据库数据正常改变,Seata数据库未残留数据。

* 全局事务已开启、分支事务全部注册到TC，其中一个TC宕机：

测试过程：开启全局事务后，所有分支事务注册到TC，程序进入断点，然后停止一个TC服务进程。

结论：分支事务在另外一个TC的协调下全局提交，业务数据库各数据正常变化。

* 全局事务已开启、分支事务全部注册到TC，所有TC宕机：

测试过程：开启全局事务后，所有分支事务注册到TC，程序进入断点，然后停止所有TC服务进程。

结论：业务数据库出现脏数据，各RM 操作的数据库undo表和Seata数据库中记录了事务信息。重启任意一个TC节点后，数据业务数据回滚，undo表和Seata数据库中事务记录被删除。

## 事务隔离性测试：

这个测试阶段主要是对Seata全局事务隔离性的测试，包括全局事务之间的隔离、全局事务和本地事务之间的隔离，以及对@GlobalLock 注解使用测试。

**环境说明：**

1、测试采用本地测试，测试项目为平台项目

2、启动两台TC，分别位于localhost:8092和lcoalhost:8093

3、项目中f1-demo作为TM，通过feign 接口调用 f1-permission服务的接口

4、f1-permission作为RM,对数据库进行交互。

5、数据库以us\_sys中的tb\_sys\_code一条数据为例，监测在不同情境下的数据变化情况。初始数据情况如下



* 全局事务gtx1开启时，其他本地事务尝试修改记录

测试结果：



结论：该数据被修改。也就是说，在全局事务还未提交的的时候，任何一个本地事务都可以对数据进行修改，这样对数据就形成一种威胁，数据随时可能被不同事务篡改。

* 全局事务gtx1开启时，对其他本地事务添加@GlobalLock注解后尝试修改记录

测试结果：



结论：数据没有被修改，说明@GlobalLock注解可以有效防止其他事务篡改全局事务所要修改的数据，如果全局事务要修改的数据有可能被其他本地事务修改，建议使用@GlobalLock 注解来保证数据的一致性。该注解的具体原理请参照《Seata使用指南》

* 全局事务gtx1开启时， 另外一个全局事务gtx2尝试修改记录

测试结果：



结论：此时gtx1 开启全局事务，持有全局锁并未放开，gtx2获取不到全局锁，全局事务没有开启

* 全局事务gtx1全局提交， 另外一个全局事务gtx2尝试修改记录

测试结果：

gtx1全局提交后：



gtx2全局提交：



结论：gtx2只有在gtx1全局提交之后才可以全局提交，保证了数据一致性。

* 全局事务gtx1本地提交时，另外一个全局事务gtx2获得本地锁，gtx1全局回滚，gtx2请求全局锁

测试结果：

gtx1本地提交



gtx1全局回滚



gtx2请求全局锁超时，释放本地锁，gtx1获得本地锁，成功回滚。



* 结论：gtx1全局回滚时，占有全局锁的同时获取本地锁，由于此刻gtx2持有本地锁且请求全局锁，所以此刻gtx1不会回滚，一段时间后gtx2请求全局锁超时，释放本地锁，gtx1取得本地锁，成功回滚。

## 其他测试：

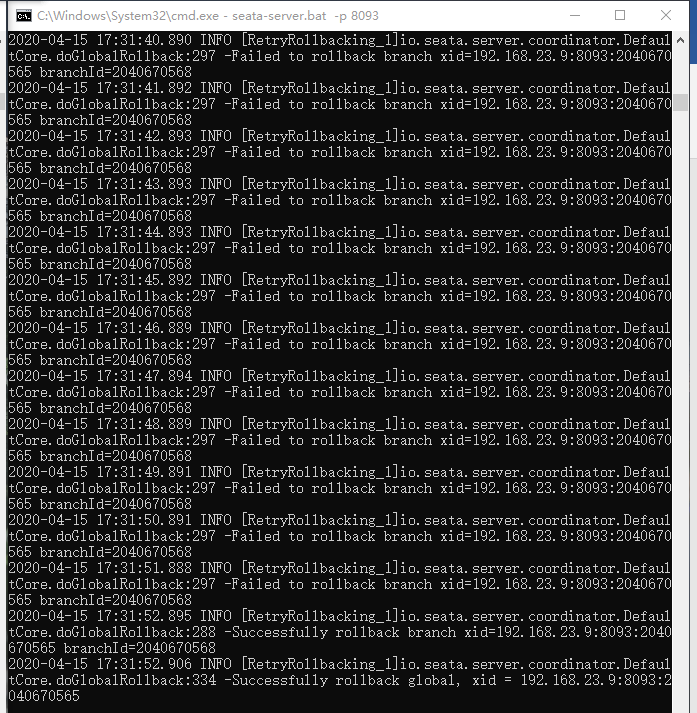
Seata 在回滚时，会把数据库的数据和后置镜像的数据进行对比，如果数据数据相同，则进行后滚。如果数据不相同会有相应的配置确定执行策略，现在对这种情况进行测试。

* 全局回滚时当前数据与后置镜像数据不同：

测试过程：程序出错之前，修改业务数据库数据，使undo表中的后置镜像的数据和业务数据修改后的数据不同。

结论：默认情况下，Server 不断显示回滚失败，业务数据库的数据并没有变化。

官方文档说 有详细的文档说明执行策略，仔细寻找后没有找到策略配置。



手动处理数据后回滚成功。

发生脏数据时，证明其他程序也可以修改该数据。

为了避免这种情况，可在其他事务上加入@GlobalLock 注解，具体使用参考《Seata使用指南》