SOFAStack

分布式事务
使用指南

产品版本：AntStack Plus 1.13.1

文档版本：20230708

蚂蚁集团
ANT GROUP

法律声明

蚂蚁集团版权所有©2022，并保留一切权利。

未经蚂蚁集团事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

商标声明

蚂蚁集团 ANT GROUP及其他蚂蚁集团相关的商标均为蚂蚁集团所有。本文档涉及的第三方的注册商标，依法由权利人所有。

免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。蚂蚁集团保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在蚂蚁集团授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过蚂蚁集团授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.什么是分布式事务 06
1.1. 概述 06
1.2. 功能特性 06
1.3. 产品优势 08
1.4. 应用场景 10
1.5. 基础术语 13
2.操作指南 15
2.1. 查看统计数据 15
2.2. 创建与管理应用 15
2.2.1. 创建应用 15
2.2.2. 管理应用 18
2.3. 创建与管理应用事务 20
2.3.1. 创建应用事务 20
2.3.2. 查看应用事务详情 23
2.3.3. 管理应用事务 23
2.3.4. 查询应用事务 24
2.4. 事务执行状态说明 26
3.开发指南 28
3.1. 下载示例代码 28
3.2. 依赖与配置项 28
3.3. 参与者接入模式简介 31
3.4. TCC 模式接入 33
3.5. FMT 模式接入 42
3.6. Saga 模式接入 49
3.7. Saga 状态机配置 60
3.8. 日志说明 68

3.8.1. dtx-client-digest.log 68
3.8.2. dtx-server-digest.log 69
3.8.3. dtx-remote.log 69
3.8.4. dtx-recovery.log 72
4.进阶指南 74
4.1. TCC 防悬挂与幂等接入 74
4.2. FMT 模式行锁策略 76
4.3. FMT 模式 SQL 支持说明 76
4.4. 接入单元化能力 78
5.最佳实践 81
5.1. Saga 模式服务设计 81
5.2. Demo 应用接入 SkyWalking 82
6.权限说明 86
7.客户端（DTX-Client）配置说明 89
8.故障排查 92
8.1. 错误码 92
8.2. 事务积压排查 106
8.3. 异常事务处理方案 108
9.常见问题 110
9.1. 产品相关问题 115
9.2. 日志相关问题 116
9.3. 网络连接问题 117
9.4. 事务执行问题 117
9.5. 其他问题 120

1.什么是分布式事务

1.1. 概述

分布式事务 DTX（Distributed Transaction-eXtended）是蚂蚁集团自主研发的金融级分布式事务中间件，用来保障在大规模分布式环境下业务活动的最终一致性。在蚂蚁集团内部被广泛地应用于交易、转账、红包等核心资金链路，服务于亿级用户的资金操作。

分布式事务可以与服务框架（SOFABoot、Spring Cloud、Dubbo 等）、数据源（数据访问代理、RDS、MySQL、OceanBase 等）、消息队列等蚂蚁集团中间件产品配合使用，轻松实现服务链路级事务、跨库事务、消息事务及各种组合。

分布式事务主要涉及发起方、参与者与事务管理器三个角色，具体描述如下：

- **发起方**：分布式事务的发起方负责启动分布式事务，通过调用参与者的服务，将参与者纳入到分布式事务当中，并决定整个分布式事务是提交还是回滚。一个分布式事务有且只能有一个发起方。
- **参与者**：参与者提供分支事务服务。当一个参与者被发起方调用后，该参与者会被纳入到该发起方启动的分布式事务中，成为该分布式事务的一个分支事务。一个分布式事务可以有多个参与者。
- **事务管理器**：事务管理器是一个独立的服务，用于协调分布式事务，包括创建主事务记录、分支事务记录，并根据分布式事务的状态，调用参与者提交或回滚方法。

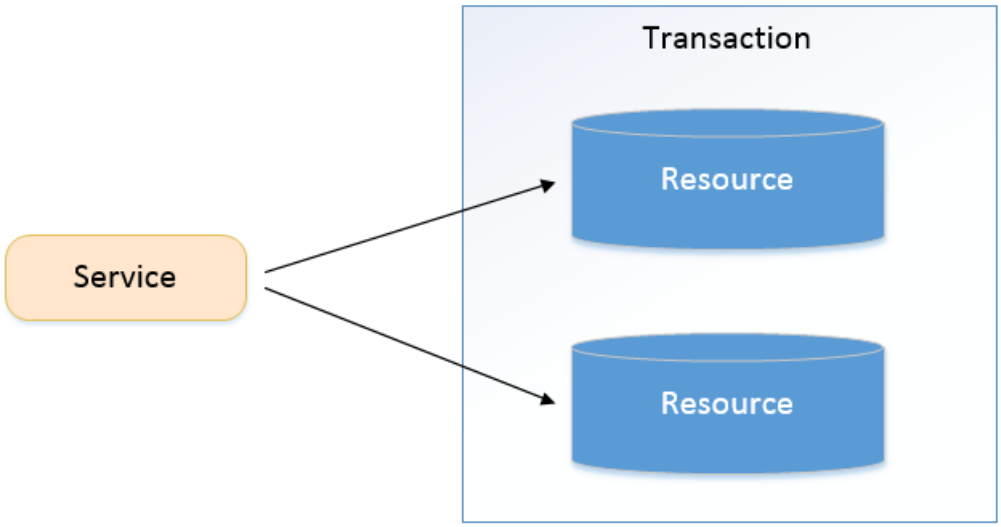
1.2. 功能特性

分布式事务（Distributed Transaction-eXtended，简称 DTX）是蚂蚁集团自主研发的金融级分布式事务中间件，支持跨数据库、跨服务以及混合的方式处理分布式应用，具备多种接入模式和金融级配套功能，本文将主要介绍分布式事务的功能特性。

核心功能

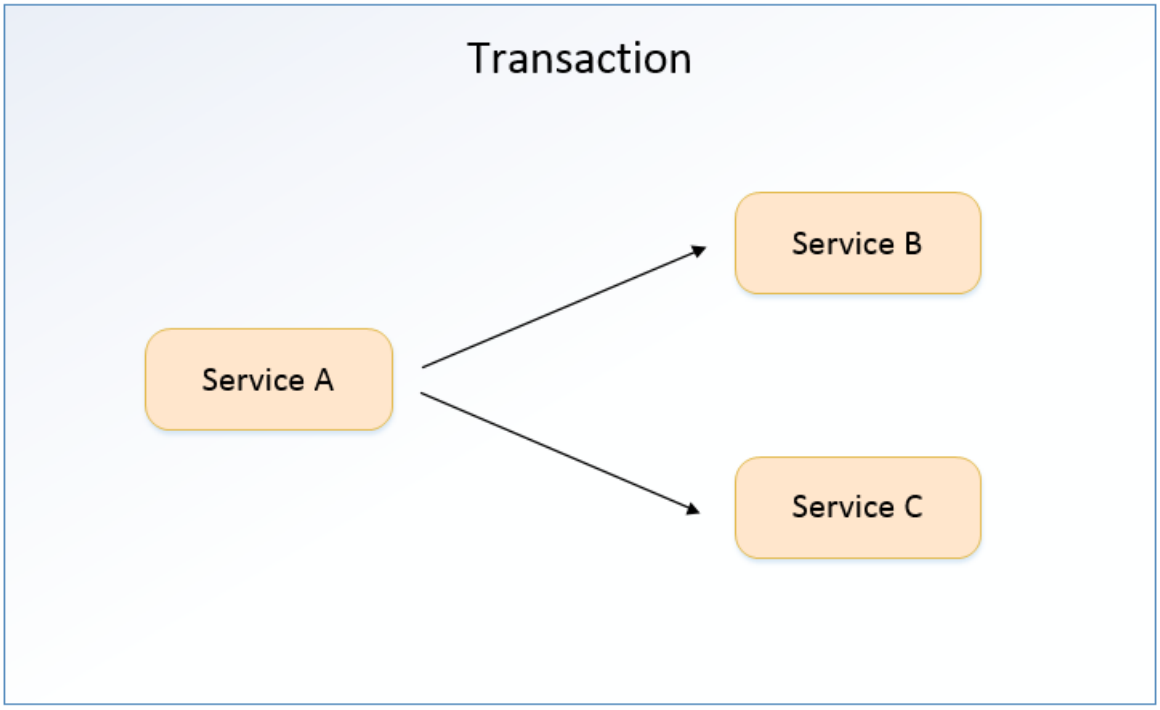
分布式事务拥有三大核心功能：跨数据库、跨服务以及混合分布式事务。

跨数据库分布式事务



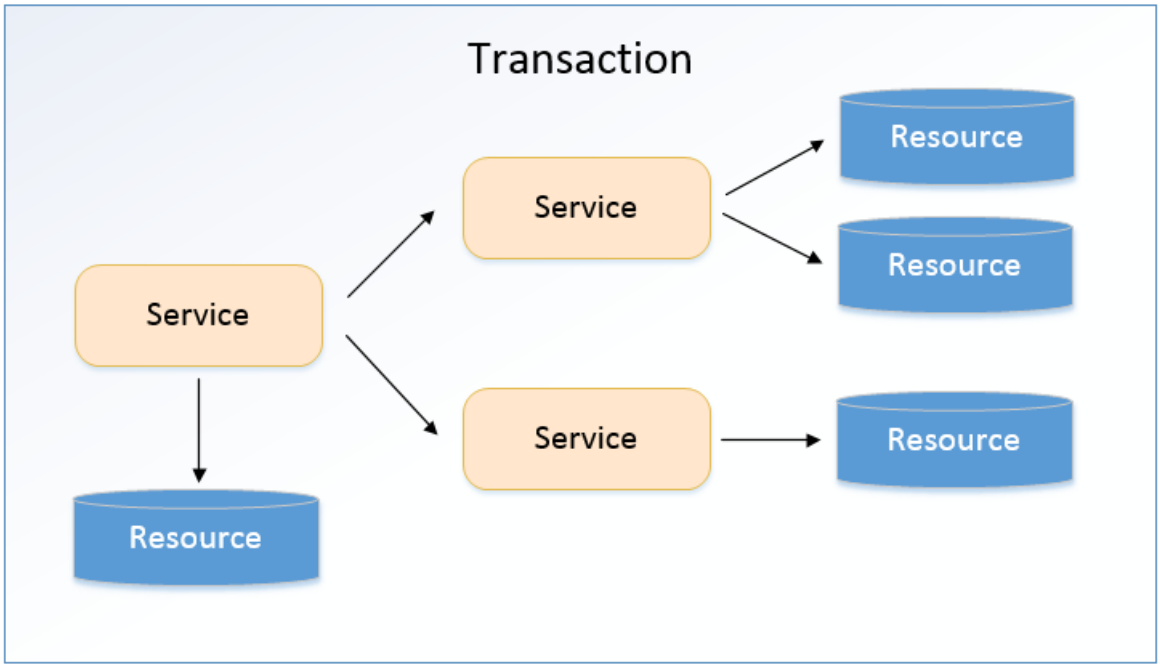
当业务规模增大，单库单表无法满足业务需求时，自然就会出现分库分表的情况。但是，单机事务又不能保证分库后的事务属性，分布式事务几乎无法避免。分布式事务可以让应用轻松具备跨库分布式事务处理能力，像使用单机数据库事务一样，透明地使用分布式事务。

跨服务的分布式事务



在基于 SOA（Service-Oriented Architecture，面向服务架构）的分布式应用环境下，系统按照功能解耦，拆分成不同的微服务，一项业务往往会涉及多个服务之间的调用。因此，为了保障业务完整，需要保证各调用服务间的数据一致性。分布式事务同样支持跨服务的分布式事务，并且可以很方便的与 SOFABoot、Spring Cloud、Dubbo 等框架打通，实现业务链路级别的分布式事务。

混合分布式事务



在一个大规模的分布式应用环境下，除了常见的微服务、数据库资源之外，还会涉及到消息队列、缓存等系统资源的使用。同时，依然需要保证这些资源间访问的数据的一致性。分布式事务支持在同一个分布式事务中引入数据库、服务、消息队列等不同类型的资源，并且支持混合接入模式。

接入模式

SOFAStack 的分布式事务产品随着业务的多样性发展而演变沉淀了多种接入模式，针对科技金融下多种业务场景灵活适配。

TCC 模式

TCC（Try-Confirm-Cancel）是一种高性能的分布式事务接入方案，该模式提供了更多的灵活性，几乎可满足任何您能想到的事务场景。TCC 模式提供自定义补偿型事务、自定义资源预留型事务、消息事务等场景，用户可以介入两阶段提交的过程，以达到特殊场景下的自定义优化及特殊功能的实现。

FMT 模式

为了解决 TCC 模式的易用性问题，分布式事务推出了框架管理事务模式（Framework-managed transactions，简称 FMT）。FMT 是一种无侵入的分布式事务解决方案，该模式解决了分布式事务的易用性问题，最大的特点是易于使用、快速接入以及对业务代码无侵入。

SAGA 模式

基于 Hector & Kenneth 发表论文 Sagas（1987）理论的长事务解决方案，在 Saga 模式中，业务流程中每个参与者都提交本地事务，当出现某一个参与者失败则补偿前面已经成功的参与者。该模式适用于业务流程长、业务流程多、参与者包含其它公司或老系统服务等场景。优势是一阶段提交本地事务无锁，事件驱动架构，参与者可异步执行，高并发高吞吐；补偿服务易于实现或老系统本身就有补偿（冲正）服务；支持服务编排、有可视化的设计器和执行轨迹监控。

XA 模式

高性能的标准 XA 事务方案，基于标准 XA 实现，消除与 JEE 开发相关的日益复杂的问题，帮助传统企业的业务无缝上云以及服务化拆分。并可以与蚂蚁集团自研数据库 OceanBase 共同打造实时数据一致性的整体解决方案。

金融级配套功能

沉淀蚂蚁集团历练多年的精粹，全方位的金融企业级能力为科技金融保驾护航。

隔离性保障

多层次全链路的纵深防御能力，金融级数据隔离能力，高度保障数据安全。

性能推进模式（Performance Bursting）

在日常高性能运转的基础上，通过性能推进模式来提升处理能力，在不改变底层基础设施的前提下从容应对峰值事务交易。

云端恢复机制

提供完备的云端服务功能，帮助业务应用在出现异常事务时进行自动恢复，避免业务损失。

1.3. 产品优势

金融场景的全面覆盖

- **支付与转账**

金融行业常见的支付、转账、账务等业务场景对于吞吐量有很高的要求。SOFAStack 的分布式事务产品在各类大促中的优异表现证明了性能不会成为瓶颈。

- **财富理财**

  这类场景中往往涉及的金额较大，所以对于产品的稳定性要求非常高，SOFAStack 的分布式事务产品拥有金融级的品质，可为业务的持续性与稳定性保驾护航。

- **保险与监管报送**

  参与方多、业务复杂度高是该类业务的典型特征，SOFAStack 的分布式事务产品历经十多年的演进历程，足以灵活应对各种场景，满足事务一致性要求，保证与各类业务完美结合。

政务领域支付更便捷

- **生活缴费**

  作为支付、转账场景的延伸，生活缴费在政务系统中不可或缺，例如水电费，电话费，上网资费等，都通过手机 APP 或者电脑端进行处理缴费。政务系统需要对缴费信息进行一致性处理，SOFAStack 的分布式事务产品可以保证关联信息同步修改，跨系统信息及时同步。

- **跨地域信息即刻同步**

  当前各地域政府机关往往有自己的数据库，人员流动、企业信息备案，都最初在本地进行登记备案。信息变更频繁的信息化时代，仅通过手工方式进行信息变更后的同步，会带来脏读和脏写的问题，采用 SOFAStack 的分布式事务产品可以保证政务机关的信息高效同步，精准一致。

泛互联网多领域场景

- **订单、会员卡、成长值、积分**

  以积分商城为例，使用会员卡余额购买商品，会涉及到扣减账户余额（数据库）、增加账户积分数量（数据库）、会员卡成长值提升、历史订单增加等服务。目前使用对账的方式来应对此类场景的性能较低，涉及业务扩展或改变时改造成本高。使用 SOFAStack 的分布式事务产品进行简单的改造接入，即可完成数据的同步。

- **担保交易**

  以电商抢购支付场景为例，秒杀抢购并发量高，性能要求高。通常流程尝试扣除用户可用资金，转移预冻结资金，增加中间账户可用资金（担保交易不能立即把钱打给商户，需要有一个中间账户来暂存），七天后需要将资金从中间账户划拨给商户。SOFAStack 的分布式事务产品可以支持大规模的抢购场景，保证客户成功支付，等到低锋期时，再慢慢消化支付数据，异步的执行资金到账流程，并且最终保证资金能顺利转入商户的账户中。

金融级品质的保障

- **金融级容灾保障**

  提供同城以及异地等多种模式以及多种级别的容灾能力，以业界最高规格的标准来保障客户业务的连续性。

- **无与伦比的性能**

  相比传统二阶段模式，减少持有锁时间，大幅提升性能。特有的性能推进模式（Performance Bursting Mode）可以大幅提升吞吐量，曾在 2021 年双 11 中支撑 25.6 万笔/秒的支付操作。

- **使用简洁易于接入**

  蚂蚁集团多年沉淀的实操经验使产品具备了快速灵活的接入能力，易于使用与运维。

- **兼容性保障**

分布式事务是一个抽象的基于 Service 层的概念，与底层事务实现无关，也就是说在分布式事务的范围内，无论是关系型数据库 MySQL、Oracle，还是 KV 存储 MemCache，或是列存数据库 HBase，只要将对它们的操作包装成分布式事务的参与者，就可以接入到分布式事务中。

1.4. 应用场景

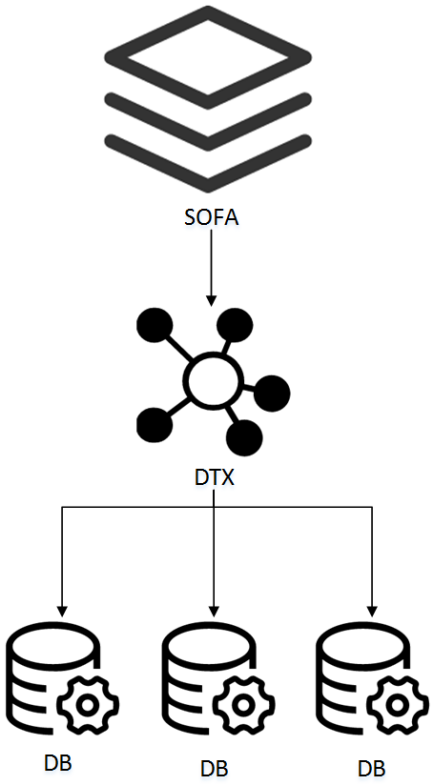
分布式事务可应用在多个涉及数据库操作的领域，尤其在金融领域可以做到全场景的覆盖与落地验证，包括：

- 支付与转账、账务：对于吞吐量有很高的要求
- 金融与理财：往往涉及的金额较大，所以对于产品的稳定性要求非常高
- 保险与监管报送：参与方多，业务复杂度高是该类业务的典型特征

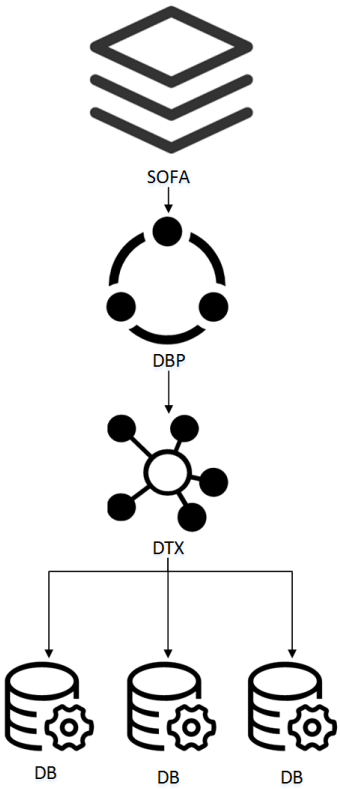
典型的应用场景如下：

分库分表后的跨数据库分布式事务

例如支付宝的交易服务，随着业务规模的增大，单个交易流水表已经不能满足业务需求，需要通过分库分表实现数据水平拆分。但是水平拆分后，单表的数据被分散到多个库的表中，原来对单表多行数据进行的变更，可能会变为对多库多表的数据变更，即单机本地事务变成了分布式事务。



使用分布式事务能够轻易获得分布式事务处理能力。如果交易服务使用数据访问代理来分库分表，虽然数据访问代理本身不支持分布式事务，但是分布式事务可以轻松和数据访问代理集成，使得数据访问代理具备分布式事务的处理能力，解决分库分表后的跨库分布式事务问题。

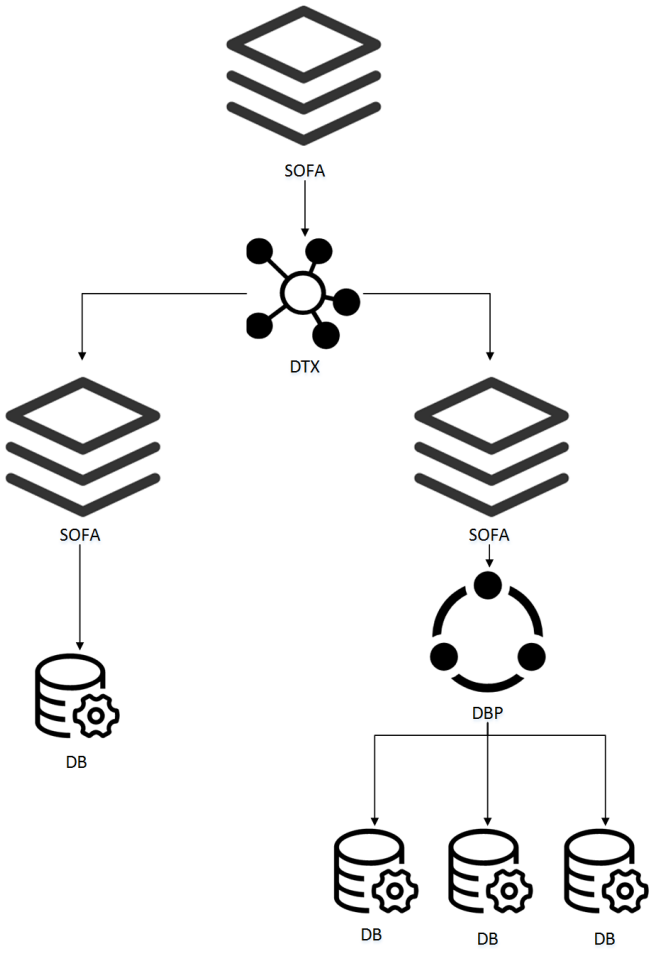


跨服务的分布式事务

例如，支付宝核心链路上的三个服务为：交易、支付、账务。当用户发起一笔交易时：

1. 首先访问交易服务，创建交易订单。
2. 然后交易服务调用支付服务为该交易创建支付订单，执行收款动作。
3. 最后支付服务调用账务服务记录账户流水和记账。

为了保证三个服务一起完成一笔交易（要么同时成功，要么同时失败），可以使用分布式事务将这三个服务放在一个分布式事务中，从而保证服务间调用的数据一致性。



混合分布式事务

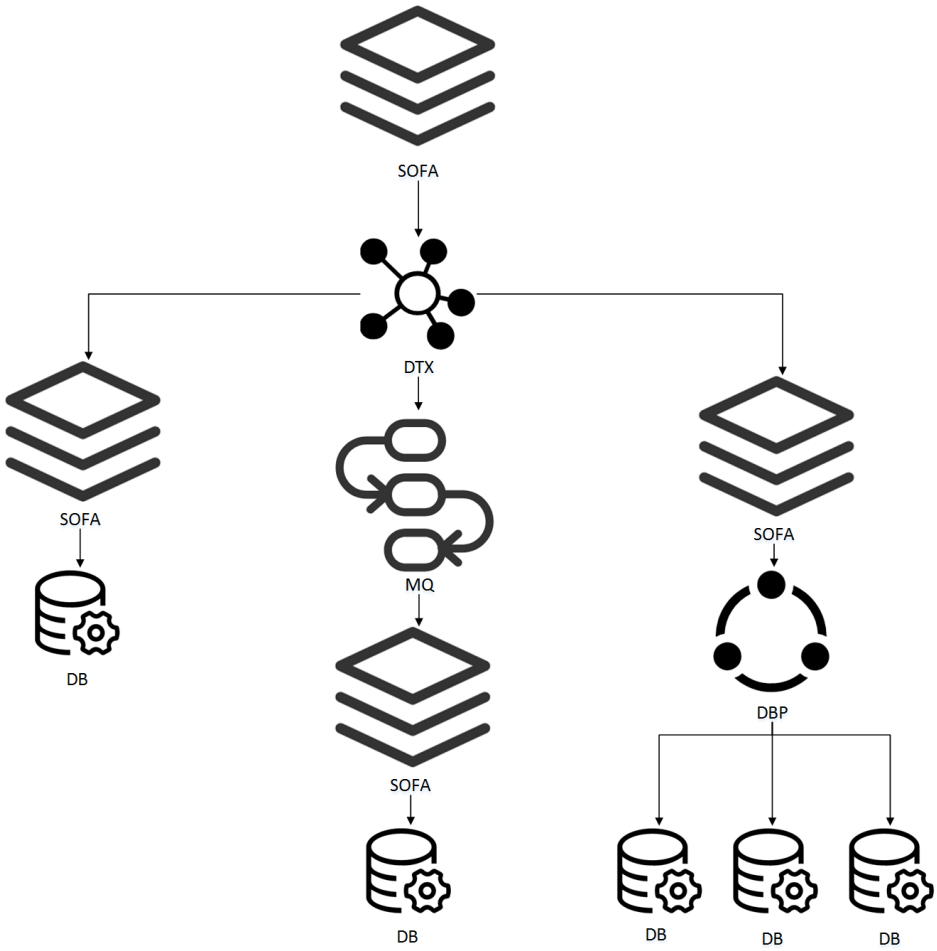
有些系统在使用数据库保证系统内数据一致的同时，也会使用消息队列作为和其他系统间的消息传递，完成不同系统间的数据一致。

例如会员注册服务和邮件发送服务，当用户注册会员成功，需要给用户发送一封邮件，告诉用户注册成功，并提示用户激活该会员。但要注意两点：

- 如果用户注册成功，一定要给用户发送一封邮件；
- 如果用户注册失败，一定不能给用户发送邮件。

因此，这同样需要会员服务和邮件服务保证原子性（要么都执行，要么都不执行）。不一样的是，邮件服务只是一种被动型的业务，并不影响用户是否能够注册成功，它只需要在用户注册成功以后发送邮件给用户即可，邮件服务不需要参与到会员服务的活动决策中。因此，可以使用消息队列来解耦会员和邮件服务。

对于此种业务场景，分布式事务可以将会员服务、消息队列组成分布式事务模型，保证事务原子性。然后通过消息队列的可靠特性，确保消息一定能够被邮件服务消费，从而使得会员与邮件服务在同一个分布式事务中。同时，邮件服务也不会影响会员服务的执行过程，只在会员服务执行成功后被动接收发送邮件的请求。



1.5. 基础术语

中文	英文	释义
事务	transaction	事务是指作为单个逻辑工作单元执行的一系列操作，要么完全执行，要么完全不执行。
分布式事务	distributed transaction	事务的发起者、资源及资源管理器和事务协调者分别位于不同的分布式系统的不同节点之上。
分支事务	action	一个分布式事务可能包含多个数据库本地事务，在分布式事务框架下，分支事务可能是一个分库上执行的 SQL 语句，或是一个自定义模式服务的调用。
发起方	initiator	分布式事务的发起方负责启动分布式事务，通过调用参与者的服务，将参与者纳入到分布式事务当中，并决定整个分布式事务是提交还是回滚。一个分布式事务有且只能有一个发起方。

中文	英文	释义
参与者	participant	参与者提供分支事务服务。当一个参与者被发起方调用，则被纳入到该发起方启动的分布式事务中，成为该分布式事务的一个分支事务。一个分布式事务可以有多个参与者。
事务管理器	transaction manager	事务管理器是一个独立的服务，用于协调分布式事务，包括创建主事务记录、分支事务记录，并根据分布式事务的状态，调用参与者提交或回滚方法。
主事务记录	activity record	又叫 Activity 记录，是整个分布式事务的主体。其最核心的数据结构是事务号（TX_ID）和事务状态（STATE），它是在启动分布式事务时持久化写入数据库的，它的状态决定了这个分布式事务的状态。
分支事务记录	action record	又叫 Action 记录，用于标识分支事务。它记录了该提供该分支事务的参与者的信息，其中包括参与者的唯一标识等。通过分支事务信息，事务管理器就可以对参与者进行提交或者回滚操作。

2.操作指南

2.1. 查看统计数据

在分布式事务控制台首页，您可以看到以下统计数据：

- **应用数**：当前环境中的应用总数。
- **client 连接数**：dtxserver 的客户端连接数。
- **当天事务数**：当天 0 点开始统计的事务总数，到第二天 0 点会重新开始统计。同时鼠标悬浮在图标上，会显示不同 BizType 的事务数占比饼图，点击某个 BizType 可以进到 BizType 的详情页面。
- **进行中事务数**：正处于进行中的事务总数。
- **异常事务数**：异常事务的总数，该指标会统计所有时间段的异常量。这部分需要用户特别关注，点击后可以自动筛选出异常事务列表，查看异常原因，并及时解决。

2.2. 创建与管理应用

2.2.1. 创建应用

本文将引导您快速创建一个应用，以便在之后创建事务时使用。

根据事务数据存储位置不同，下面分别介绍 **异库模式** 和 **同库模式** 的应用创建。推荐采用同库存储。

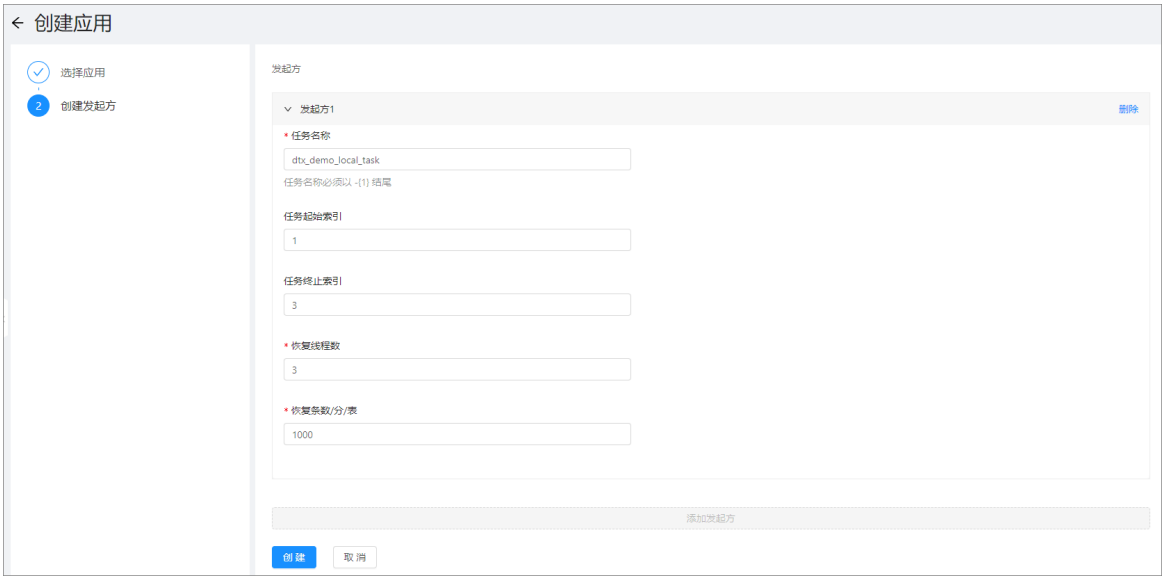
同库存储

1. 登录分布式事务控制台。
2. 在左侧导航栏上，单击 **应用事务** > **应用配置**。
3. 单击 **创建应用**，在 **创建应用** 页面配置以下信息：

参数	是否必填	参数说明
应用名称	必填	输入应用名称。
事务数据存储位置	必填	选择 **同库存储**，表示事务数据存储在与业务系统相同的数据库。 Client 端直接写入到与业务同库的事务日志表中。适合对吞吐量和性能要求比较高的业务使用，比如核心的交易、支付、账务等业务。

4. 单击 下一步。
5. 添加发起方。
   i. 单击 **添加发起方**。
   ii. 单击发起方名称旁的展开图标 `>`。

iii. 在 **创建发起方** 页面配置以下信息。



参数	是否必填	参数说明
任务名称	必填	输入任务名称，最长输入 64 个字符。 **说明** 任务名称与发起方配置的同库恢复任务执行器的 taskName 要保持一致。
任务起始索引	选填	设置搜索起始索引，用于筛选线程列表。
任务终止索引	选填	设置搜索终止索引，用于筛选线程列表。
恢复线程数	必填	表示客户端需开启的处理恢复任务的线程数量。默认值为 3。
恢复条数/分/表	必填	表示客户端一次恢复任务会读取每个分表多少条事务进行恢复。默认值为 1000。

iv. 单击 **创建**。

异库存储

1. 登录分布式事务控制台。
2. 在左侧导航栏上，单击 **应用事务 > 应用配置**。
3. 单击 **创建应用**，在 **创建应用** 页面配置以下信息：

参数	是否必填	参数说明
应用名称	必填	输入应用名称。
事务数据存储位置	必填	选择 **异库存储**，表示事务存储在事务服务器端。异库存储下，无需进行发起方、参与方配置。 Client 通过长连接把日志送到 Sever 端，存储在 Server 端的数据库中。适合需要将业务数据单独存储的场景。
拆分模式	必填	选择 **单库单表模式**，表示事务数据均存储在一个业务库和一个数据表中。

4. 单击 **创建**。

2.2.2. 管理应用

编辑应用

根据事务数据存储位置不同，下面分别介绍 **异库模式** 和 **同库模式** 的编辑。

异库模式

异库模式仅支持修改拆分模式。操作步骤如下：

在 **应用配置** 页面，单击目标应用 **操作** 列下的 **编辑**。

同库模式

同库模式仅支持修改发起方配置。操作步骤如下：

1. 在 **应用配置** 页面，单击目标应用 **操作** 列下的 **编辑**。

2. 在应用详情页，单击目标发起方右侧的 **编辑**。

3. 在编辑发起方页修改相关信息，然后单击 **保存**。

导入应用配置

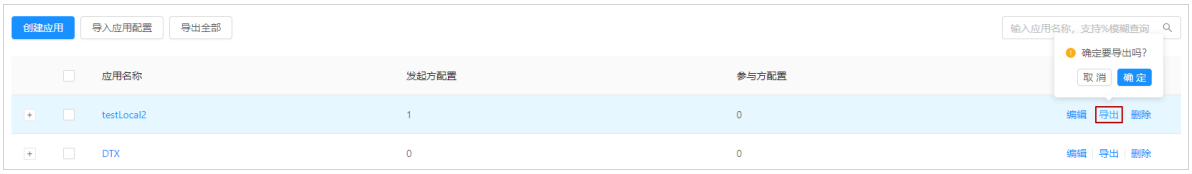
1. 在 **应用配置** 页面，单击 **导入应用配置**。

2. 在新弹出窗口中，选择待导入文件（支持的文件扩展名：`.txt`、`.json`）后，单击 **确定**。

导出应用配置

- **导出全部应用配置**：在 **应用配置** 页面，单击 **导出全部** > **确定** 导出全部应用配置。

- **导出单个应用配置**：在 **应用配置** 页面，单击目标应用 **操作** 列下的 **导出**，单击 **确定**。



删除应用

在 **应用配置** 页面，单击目标应用 **操作** 列下的 **删除**，单击 **确定**。

> **重要**
>
> 删除后，此应用将无法被使用，请谨慎操作。

2.3. 创建与管理应用事务

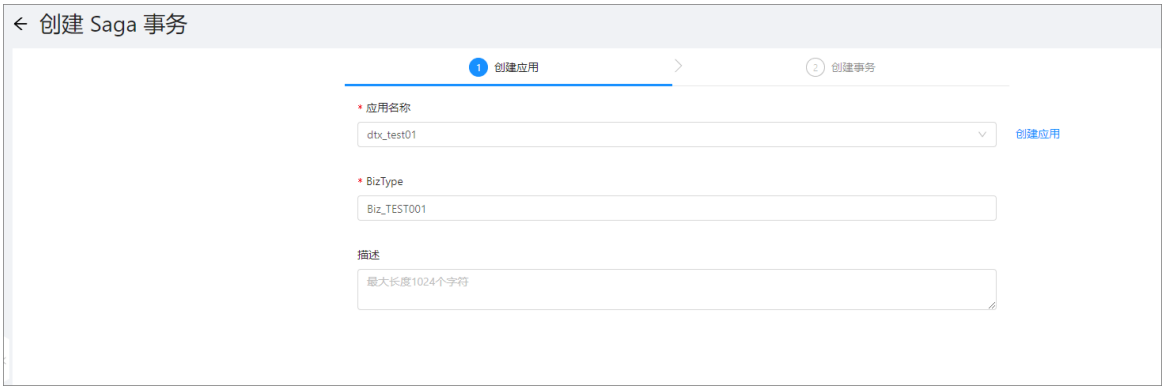
2.3.1. 创建应用事务

本文将介绍如何创建应用事务，获取相应的 BizType，并关联至相应的发起方或参与方应用，用于在本地应用中接入该事务。

创建 Saga 模式的应用事务

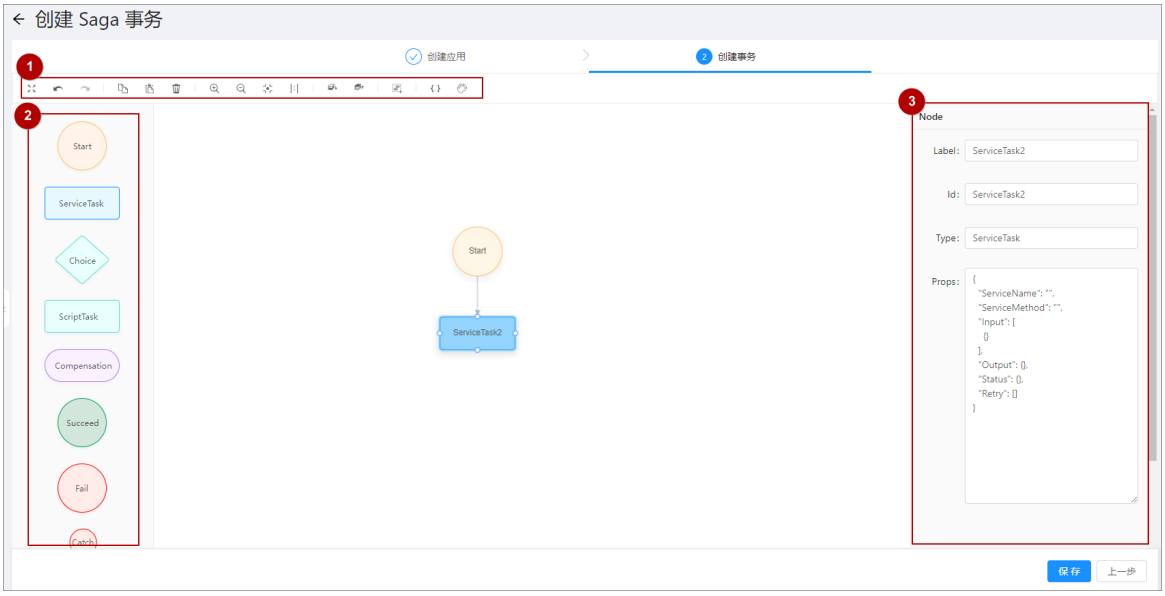
> **说明**
>
> Saga 模式是 SEATA 提供的长事务解决方案。Saga 模式下，分布式事务内存在多个参与者，每一个参与者都是一个冲正补偿服务，用户需要根据业务场景实现其正向操作（原服务）和逆向回滚操作（补偿服务）。

1. 登录分布式事务控制台。
2. 在左侧导航栏上，单击 **应用事务 > 事务配置**。
3. 单击 **创建应用事务 > Saga 事务**。
   i. 创建应用。

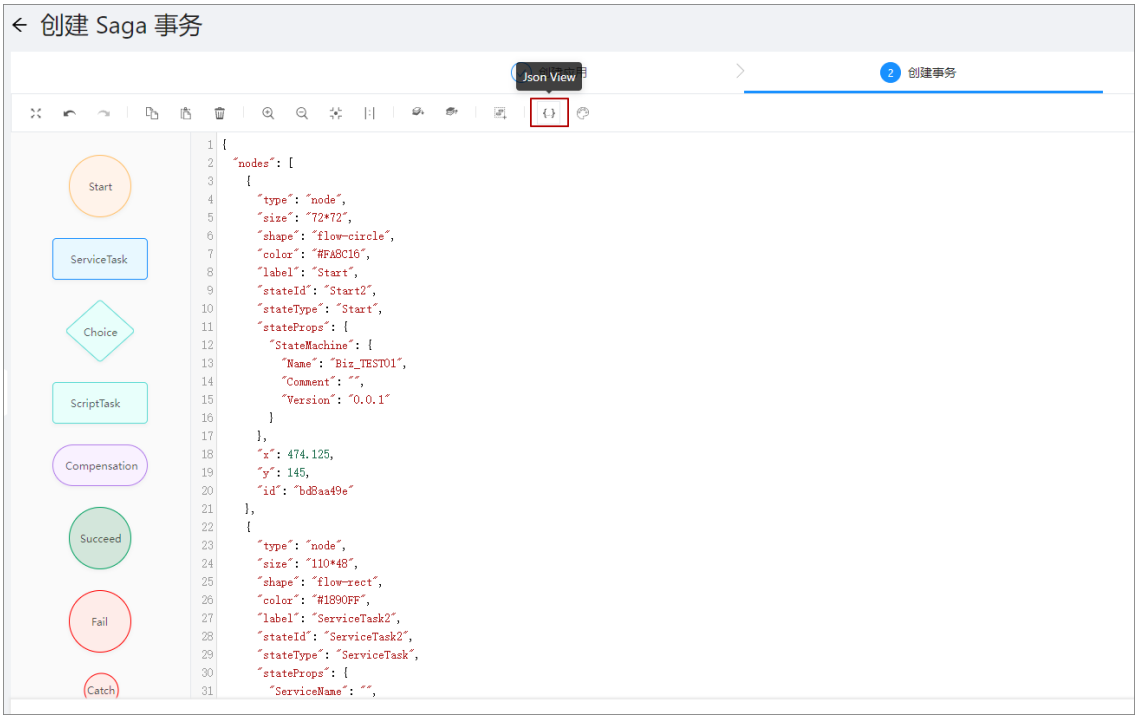


参数	是否必填	参数说明
应用名称	必填	选择开启分布式事务的发起方应用系统。
BizType	必填	自定义的业务类型，支持英文、数字、下划线和逗号。
描述	选填	输入对该应用事务的备注信息等，如业务意义等。

   ii. 单击 **下一步**。
   iii. 在状态设计器中，编排设计 Saga 模式应用事务流程。操作说明如下：



- 在上方导航栏内（图示中①），你可以执行放大流程图、撤销、复制、删除等操作。单击编辑区上方的 {..} 图标，可将图形界面切换成代码界面，查看代码。



- 在左侧边栏内（图示中②），选择并拖拽您需要的组件进入中间的编辑区。
- 在右边窗口中（图示中③），即可定义该组件的详情，包括标签、ID、类型以及属性。属性配置可参考 Saga 状态机配置。

4. 单击 **保存**，完成事务创建。

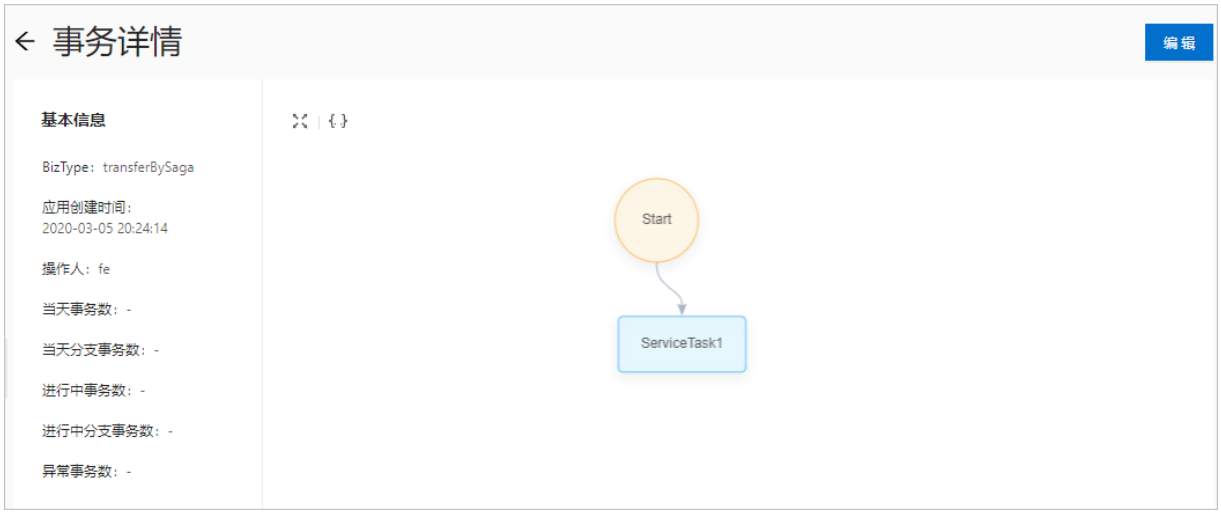
后续操作

应用事务创建完成后，您即可开始本地接入该应用事务，详情参见 参与者接入模式。

2.3.2. 查看应用事务详情

在 **事务配置** 页面中，单击目标 BizType 进入事务详情页。基本信息统计了当前 BizType 的相关数据，包括当天事务数、进行中事务数、异常事务数、当天分支事务数、进行中分支事务数。

对于 Saga 模式的应用事务，基本信息区域还展示了具体事务的节点状态图，如下图所示。



2.3.3. 管理应用事务

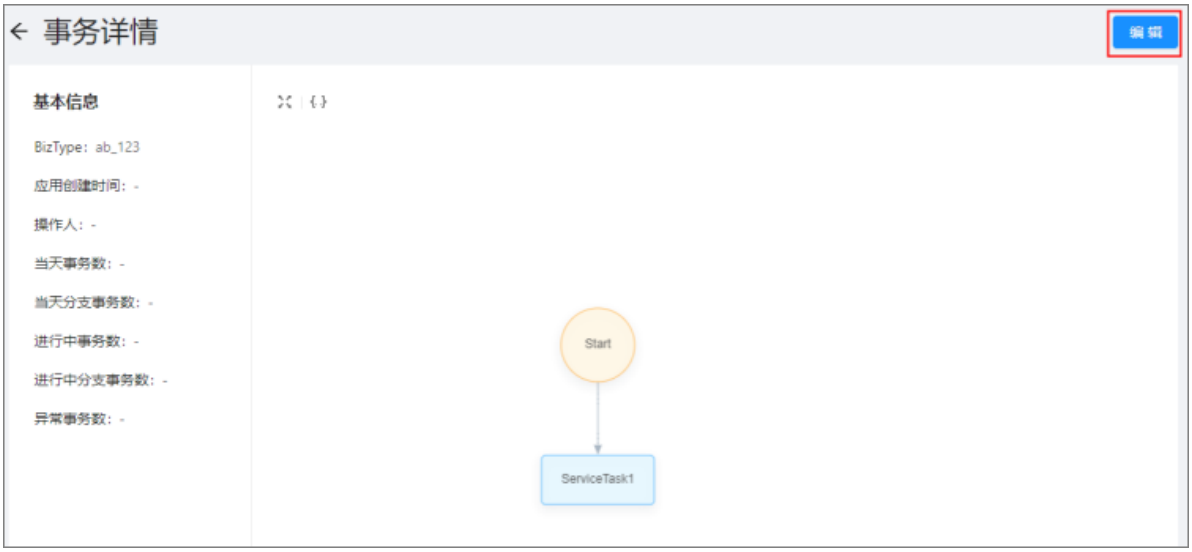
在 **事务配置** 页面列出了所有应用事务及其事务相关信息，包括了发起方、所有参与方总数、当天事务数、异常事务数等。

BizType	事务类型	当天事务数	异常事务数	应用创建时间	操作
[illegible]	Saga 模式	0	0	2023-02-01 16:24:35	编辑 | 导出 | 删除
[illegible]	Saga 模式	0	0	2023-02-01 16:05:20	编辑 | 导出 | 删除

编辑应用事务

您可以通过以下任一方式进入事务编辑页面，编辑应用事务信息：

- 在 **事务配置** 页面，单击目标事务 **操作** 列下的 **编辑**。
- 在 **事务配置** 页面，单击目标事务名称进入事务详情页，单击右上角的 **编辑**。



导入事务配置

1. 在 **事务配置** 页面，单击 **导入事务配置**。

2. 在新弹出窗口中，选择待导入文件（支持的文件扩展名：`.txt`、`.json`）。
3. 单击 **确定**。

导出事务配置

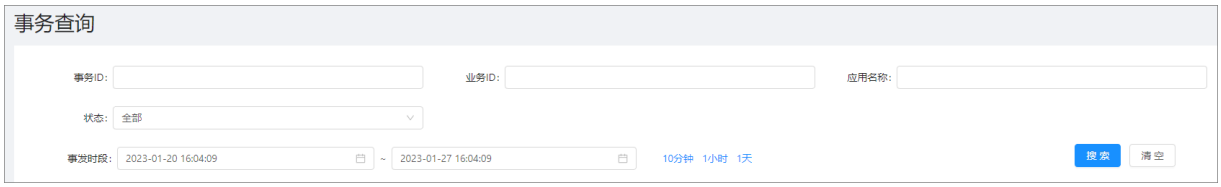
- **导出全部事务配置**：在 **事务配置** 页面，单击 **导出全部** > **确定** 导出全部事务配置。

- **导出单个事务配置**：在 **事务配置** 页面，单击目标应用 **操作** 列下的 **导出**，并确认。

2.3.4. 查询应用事务

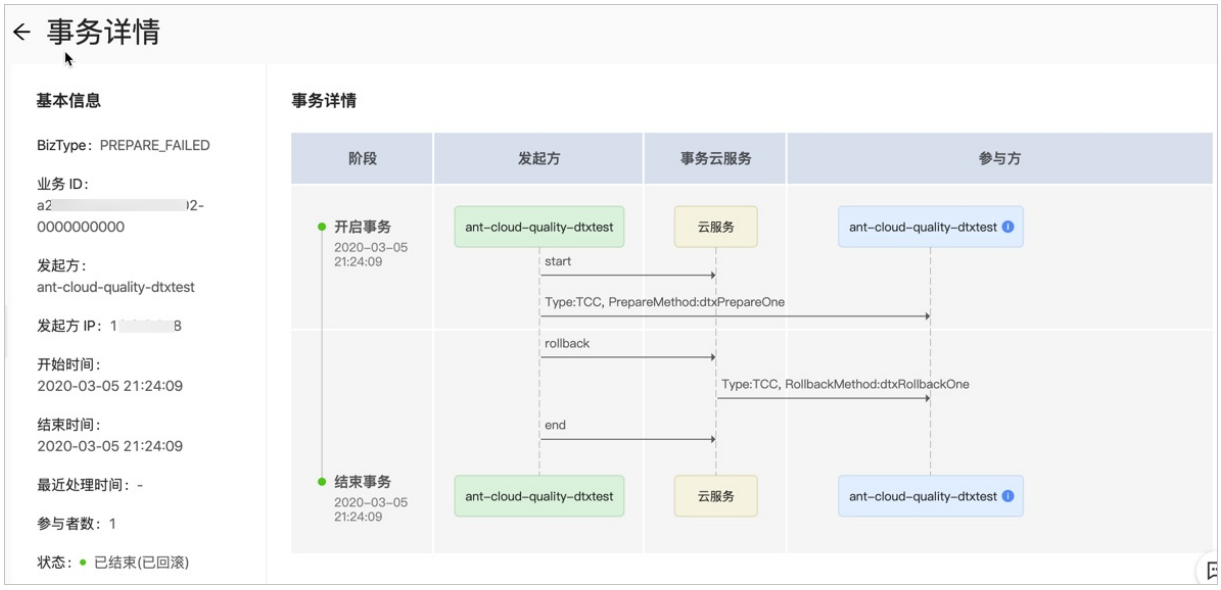
本文介绍如何查询应用事务信息，包括基本信息和详情信息。

在左侧导航栏上，单击 **事务监控** > **事务查询**。您可以通过以下查询条件快速搜索出您想要的事务：



- 事务 ID：想要查询的事务的唯一标识。
- 业务 ID：业务的唯一标识。
- 应用名称：输入事务对应的应用名称。
- 事务状态：包括全部、进行中、异常、已完成。
- 事发时段：可以根据需求快速选取最近时间范围（10 分钟、1 小时、1 天），也支持自定义时段。

在搜索结果中，单击目标事务右侧的 **详情** 即可查看该事务的相关信息。



基本信息

基本信息 栏包含以下信息：

- BizType：业务类型，由用户根据自己业务场景自定义。
- 发起方 IP：发起事务的应用的 IP。
- 业务 ID：业务的唯一标识。
- 事务状态：详见 事务执行状态说明。
- 发起时间：事务开始的日期与时间。
- 结束时间：事务结束的日期与时间。
- 参与方数量：事务参与方的总数。

事务详情

您可以在 **事务详情** 栏查看事务各阶段的状态，以及分支事务信息。

单击分支事务中的应用名称即可查看相应的分支事务信息。

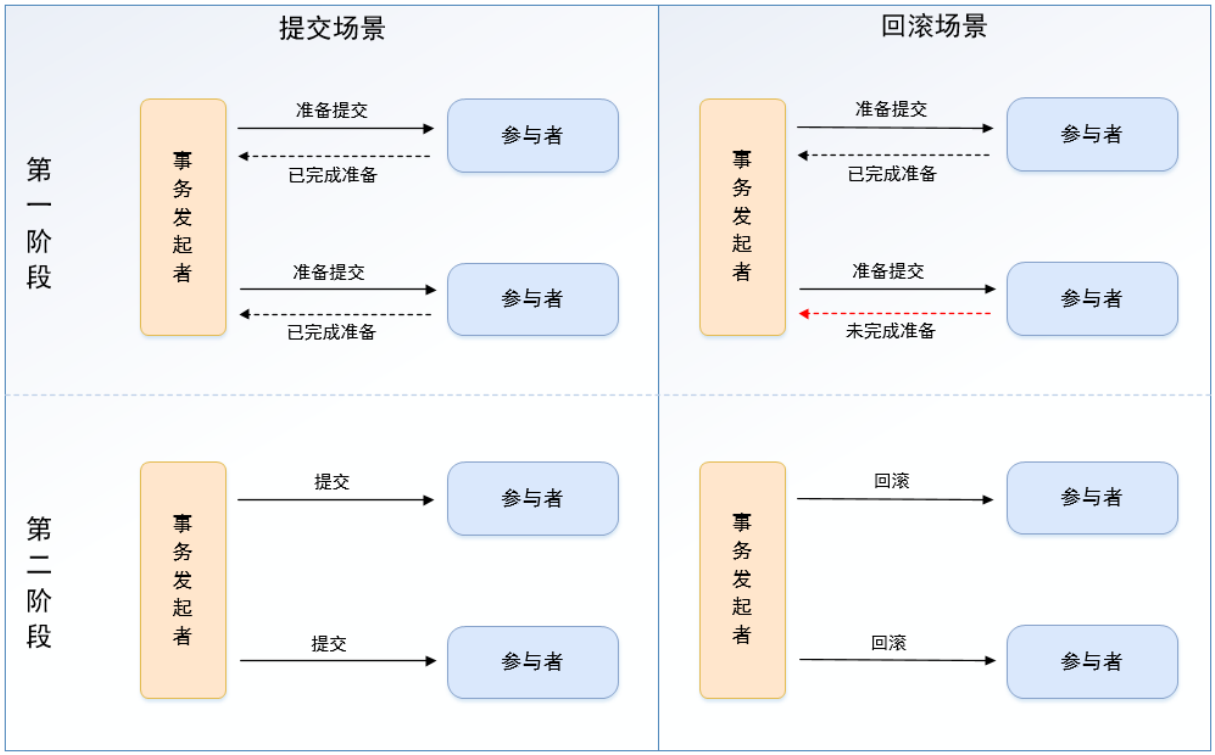
- 应用名称。
- 分布式事务的开启、结束时间。
- 发起方、参与方信息。
- 事务操作过程，包括 **开启**、**提交** 和 **回滚** 事务。
- 参与方类型（Type），详见 参与者接入模式。
- 如果事务执行报错，显示具体错误信息。

2.4. 事务执行状态说明

分布式事务使用两阶段提交协议（Two-Phase Commit Protocol，简称 2PC）来保证事务执行的原子性。

2PC 包含两个阶段：

- 第一阶段，也称准备阶段。由事务发起者向各参与者发送请求，询问参与者是否准备好执行事务。
- 第二阶段，也称提交阶段。在一阶段所有参与者都确认可以执行事务后，各参与者开始分别执行事务分支。所有分支都成功则事务数据提交成功，否则，所有分支都进行数据回滚操作。



事务在某一时间点可能处在以下状态之一：

- 初始化：应用发起事务
- 进行中
  - 准备中：一阶段操作中
  - 提交中：一阶段结束，正在二阶段的提交操作中
  - 回滚中：一阶段结束，因为某些业务失败，正在二阶段的回滚操作中

- 已结束
  - 已提交：事务结束，事务执行的数据变更已提交
  - 已回滚：事务结束，事务执行的数据变更已回滚
- 异常
  - 提交异常：一阶段结束，二阶段处理提交操作时出现异常
  - 回滚异常：一阶段结束，二阶段处理回滚操作时出现异常
  - 回查异常：一阶段结束，二阶段处理回查业务接口时出现异常

3.开发指南

3.1. 下载示例代码

为快速进行分布式事务工程的本地开发，您可以单击此处下载分布式事务的 示例工程。

该示例代码包提供了以下示例工程：

- `account-demo-local`：TCC 模式调用本地服务
- `account-demo-remote`：TCC 模式调用远程服务
- `account-demo-saga`：SAGA 模式

示例工程下载完成后，即可将其导入 IDE 工具，通过运行 AccountMain 的 main 函数调用。

3.2. 依赖与配置项

分布式事务不仅支持 SOFABoot 框架，也支持纯 Spring 和 Spring Cloud 的应用场景。本文将分别介绍针对不同的服务框架所要预先完成的依赖与配置项操作。无论是事务发起方还是参与者均需要完成本文的依赖与配置项操作。

环境要求

必须使用 JDK 1.7 或以上版本。

工程依赖

基于 SOFABoot 框架

SOFABoot 框架中已经默认集成了分布式事务 SDK，不需要手动添加分布式事务的 Maven 依赖。

如果您不愿意使用 SOFABoot 提供的分布式事务 SDK，可以在工程的主 `pom.xml` 文件添加分布式事务的 Maven 依赖，依赖配置如下：

```
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>dtx-enterprise-sofa-boot-starter</artifactId>
    <version>${dtx.version}</version>
</dependency>
```

> **说明**
>
> 您可以在 `${dtx.version}` 中选择需要的版本或最新版本，如 `2.3.0` 版本。关于最新版本，参见 引入 SOFA 中间件。

基于 Spring 框架

在 Spring 框架中，您需要依次添加 Spring 依赖和分布式事务依赖。

1. 在 `pom.xml` 文件中引入 Spring 依赖。

分布式事务依赖 Spring 4 框架，需要在应用中引入 Spring JAR 包，您可以在应用中添加如下 Spring Maven 依赖：

```
<!-- spring boot and spring dependencies begin-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure</artifactId>
</dependency>
<!-- logback -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
</dependency>
<!-- test -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<!-- spring dependencies end -->
```

说明

Spring 使用默认版本即可，也可自定义需要的版本，如 `2.1.13.RELEASE` 。

2. 在 `pom.xml` 文件中引入分布式事务依赖。

```
<dependency>
     <groupId>com.alipay.dtx</groupId>
     <artifactId>dtx-sofaboot</artifactId>
     <version>${dtx.version}</version>
</dependency>
```

说明

您可以在 `${dtx.version}` 中选择需要的版本或最新版本，如 `2.3.0` 版本。

基于 Spring Cloud 框架

在 SpringBoot 框架中，您需要依次添加 SpringBoot 依赖和分布式事务依赖。

1. 在 `pom.xml` 文件中引入 Parent starter 依赖。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.5.12</version>
</parent>
```

2. 在 `pom.xml` 文件中引入 OpenFeign starter 等 Spring Cloud 依赖。

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-openfeign</artifactId>
      <version>3.0.6</version>
</dependency>
```

3. 在 `pom.xml` 文件中引入分布式事务依赖。

```
<dependency>
    <groupId>com.alipay.dtx</groupId>
    <artifactId>dtx-springboot</artifactId>
    <version>2.4.7-SNAPSHOT</version>
</dependency>
```

工程属性配置

为了获取事务管理器的地址，需要配置工程属性文件。配置说明如下：

- **SOFAStack 环境**

  在金融分布式架构 SOFAStack 环境，默认使用 AntVIP 组件寻址。工程配置文件存放路径与配置方式如下：

  - **SOFABoot 框架**：存放在 `src/main/resources/config/application.properties` 。
  - 配置项：
    - `com.alipay.appname=xxx` ：应用名称
    - `com.alipay.env=shared` ：环境标识
    - `com.alipay.instanceid=xxx` ：唯一标识
    - `com.antcloud.antvip.endpoint=xxx` ：配置 AntVIP 地址

  具体取值请参见 properties 配置项。

- **非 SOFAStack 环境**

  在非金融分布式架构 SOFAStack 环境，直接使用自定义事务管理器地址。工程配置文件存放路径与配置方式如下：

  - **纯 Spring 框架**：存放在工程的 `META-INF` 目录下。
  - 配置项：
    - `com.alipay.appname=xxx` ：应用名称
    - `com.alipay.dtx.server.hosts=xxx,xxx` ：事务管理器域名/IP 列表

配置扫描器

为了实现自动代理涉及分布式事务的类或方法，需要在 Spring 配置文件中添加以下分布式事务扫描器。

```
<bean class="com.alipay.sofa.dtx.client.aop.ComponentScanner"/>
```

3.3. 参与者接入模式简介

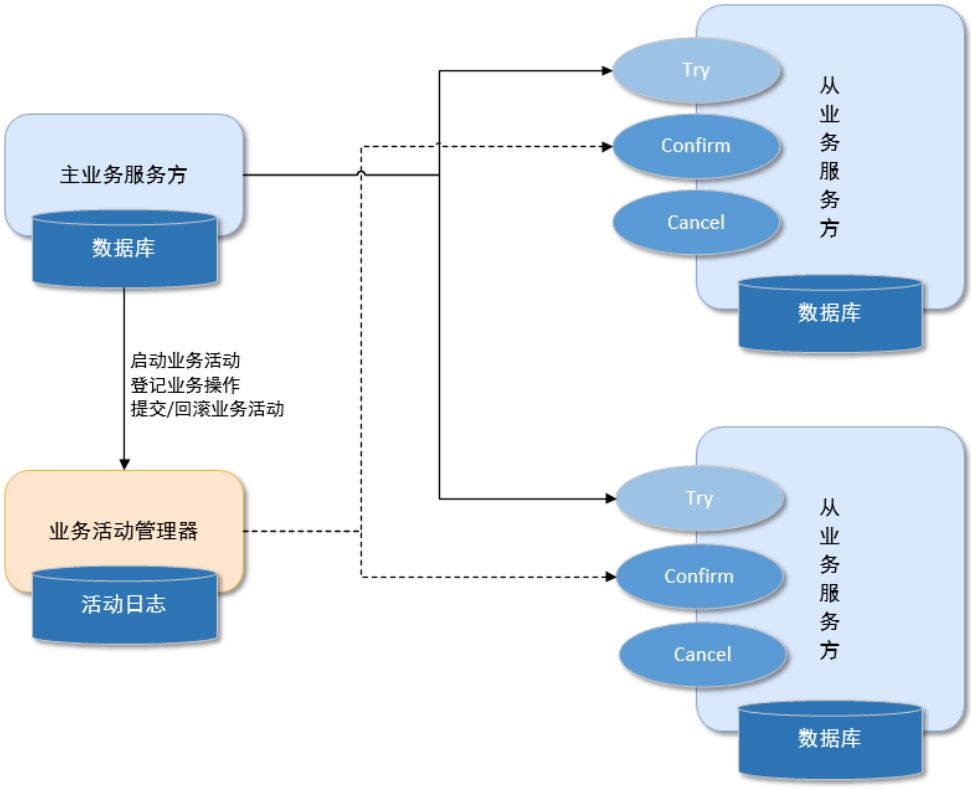
使用分布式事务时涉及发起方与参与者两个核心角色。针对参与者，分布式事务目前提供 TCC 模式、FMT 模式与 Saga 模式三种参与者接入模式。本文将具体介绍三种不同的接入模式。

> 说明
>
> **发起方**：指的是开启分布式事务的应用系统。
>
> **参与者**：指的是提供分支事务的应用系统。同一个应用系统可能兼具发起方和参与方两个角色。

TCC 模式

TCC（Try-Confirm-Cancel）是一种高性能的分布式事务接入模式，该模式提供了更多的灵活性，几乎可满足任何您能想到的事务场景。TCC 模式提供自定义补偿型事务、自定义资源预留型事务、消息事务等场景，您可以介入两阶段提交的过程，以达到特殊场景下的自定义优化及特殊功能的实现。

TCC 模式架构如下：



TCC 模式的特性如下：

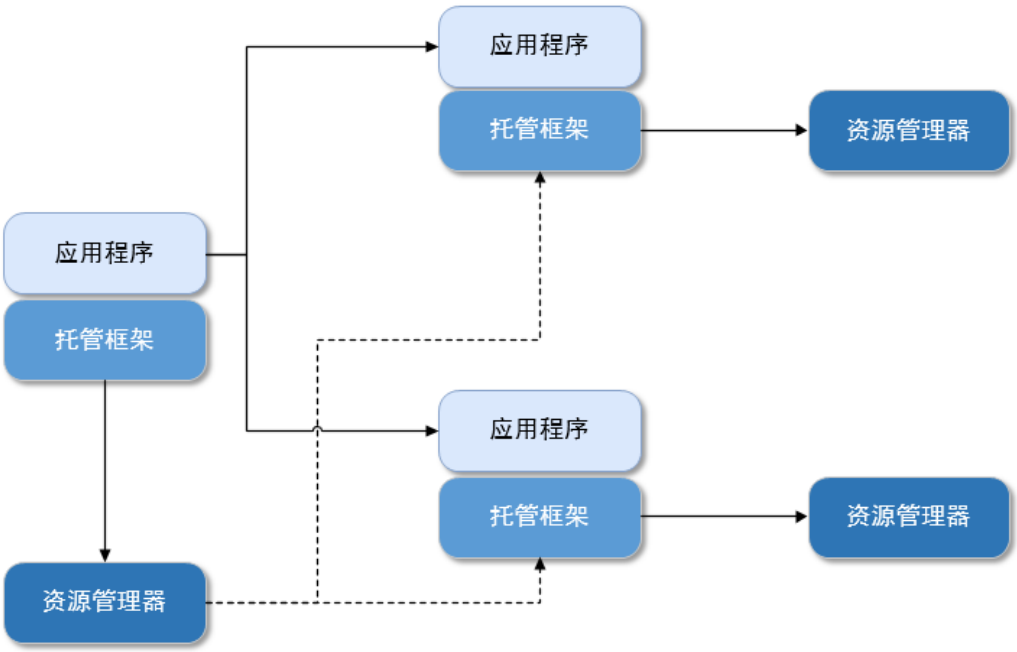
- **最终一致**：事务处理过程中，会有短暂不一致的情况，但通过恢复系统，可以让事务的数据达到最终一致的目标。

- **协议简单**：TCC 模式定义了类似 2PC 的标准两阶段接口，业务系统只需要实现对应的接口就可以使用 TCC 模式的事务功能。
- **与 RPC 服务协议无关**：在 SOA 架构下，一个或多个数据库操作往往被包装成一个或多个 Service，不同的 Service 之间通过 RPC 协议通信。TCC 模式构建在 SOA 架构上，与底层协议无关。
- **与底层事务实现无关**：TCC 是一个抽象的基于 Service 层的概念，与底层事务实现无关，也就是说在 TCC 的范围内，无论是关系型数据库 MySQL、Oracle，还是 KV 存储 MemCache，或者列存数据库 HBase，只要将对其的操作包装成 TCC 的参与者，就可以接入到 TCC 事务范围内。

FMT 模式

为了解决 TCC 模式的易用性问题，金融科技分布式事务推出了框架管理事务模式 FMT（Framework-managed transactions）。FMT 是一种无侵入的分布式事务解决方案，该模式解决了分布式事务的易用性问题，最大的特点是易于使用、快速接入以及对业务代码无侵入。

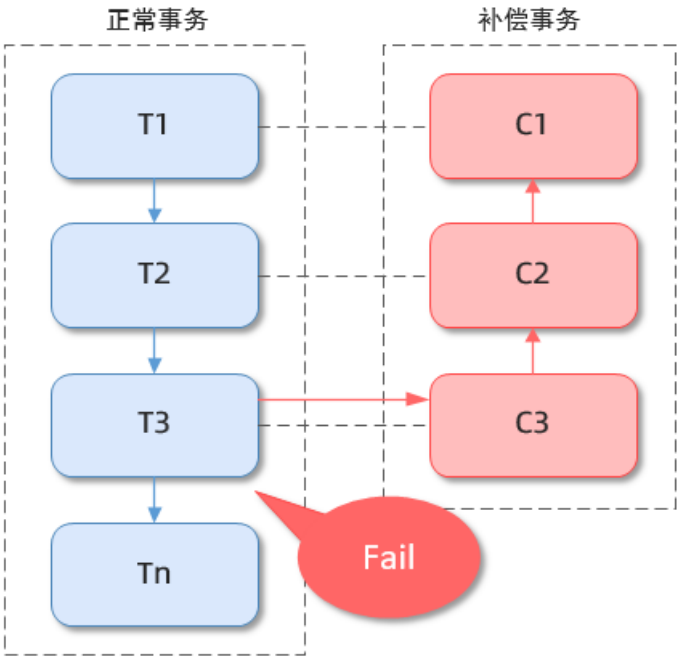
FMT 模式架构如下：



Saga 模式

Saga 模式是 SEATA 提供的长事务解决方案，适用于业务流程长、业务流程多、参与者包含其它公司或老系统服务等场景。在 Saga 模式中，业务流程中每个参与者都提交本地事务。当出现某一个参与者失败则补偿前面已经成功的参与者，一阶段正向服务和二阶段补偿服务都由业务开发实现。

Saga 模式架构如下图所示。其中，左侧是正常事务流程，当执行到 T3 时出现异常，则开始执行右侧的事务补偿流程，反向执行 T3、T2、T1 的补偿服务。



Soga 模式的特性如下：

- 一阶段提交本地数据库事务无锁，事件驱动架构，参与者可异步执行，高并发高吞吐。
- 补偿服务即正向服务的“反向”，易于理解，易于实现。
- 支持服务编排、有可视化的设计器和执行轨迹监控。

3.4. TCC 模式接入

分布式事务支持以 TCC 模式接入 SOFARPC、Dubbo 和 Spring Cloud 远程服务框架。本文将介绍如何定义接口、实现接口，并针对远程服务框架进行发布服务。

定义接口

TCC 参与者需要实现三个方法，分别是一阶段 Try 方法、二阶段 Confirm 方法以及二阶段 Cancel 方法。示例如下：

- **基于 SOFARPC 和 Dubbo 框架**

  在 TCC 参与者的接口中需要先加上 `@TwoPhaseBusinessAction` 注解（需引入 `com.alipay.dtx.client.core.spi.TwoPhaseBusinessAction` 包），并声明这个三个方法，如下所示：

```
public interface TccAction {
    @TwoPhaseBusinessAction(name = "yourTccActionName", commitMethod = "confirm", rollbac
kMethod = "cancel")
    public boolean try(
    BusinessActionContext businessActionContext, int a, int b);

    public boolean confirm(BusinessActionContext businessActionContext);

    public boolean cancel(BusinessActionContext businessActionContext);
}
```

`@TwoPhaseBusinessAction` 注解属性说明：

- `name`：TCC 参与者的名称，可自定义，但必须全局唯一。
- `commitMethod`：指定二阶段 Confirm 方法的名称，可自定义。
- `rollbackMethod`：指定二阶段 Cancel 方法的名称，可自定义。

- **基于 Spring Cloud 框架**

  i. 定义接口。

  在 Controller 中定义 Try、Cancel、Confirm。

```
//使用RestRemoting声明参与者是REST接口
@RestRemoting
public interface AccountTccAction {

    /**
     * try 尝试
     *
     * BusinessActionContext 上下文对象，用来在两个阶段之间传递数据
     * BusinessActionContextParameter 注解的参数数据会被存入 BusinessActionContext
     * TwoPhaseBusinessAction 注解中commitMethod、rollbackMethod 属性有默认值，可以不写
     *
     * @param businessActionContext
     * @param userId
     * @param money
     * @return
     */
    @TwoPhaseBusinessAction(name = "accountTccAction", appName = "account")
    boolean prepareDecreaseMoney(BusinessActionContext businessActionContext,
                                 @BusinessActionContextParameter(paramName = "userId")L
ong userId,
                                 @BusinessActionContextParameter(paramName = "money")Bi
gDecimal money);

    /**
     * commit 提交
     * @param businessActionContext
     * @return
     */
    boolean commit(BusinessActionContext businessActionContext);

    /**
     * cancel 撤销
     * @param businessActionContext
     * @return
     */
    boolean rollback(BusinessActionContext businessActionContext);
}
```

ii. 通过 Controller 暴露服务。

Controller 需要实现参与者 REST 接口。

```
@RestController
@RequestMapping("account")
public class AccountController implements AccountTccAction{
    @Autowired
    private final AccountTccService accountTccService;

    /**
     * 账户扣钱
     * @param userId
     * @param money
     * @return
     */
    @PostMapping(value = "decrease-money")
    @ResponseBody
    public boolean decreaseMoney(@RequestBody BusinessActionContext businessActionConte
xt, Long userId, BigDecimal money){
        accountTccService.prepareDecreaseMoney(businessActionContext, userId,money);
         return true;
    }

    /**
     * commit 提交
     * @param businessActionContext
     * @return
     */
    @PostMapping(value = "commit")
    @ResponseBody
    public boolean commit(@RequestBody BusinessActionContext businessActionContext){
        return accountTccService.commit(businessActionContext);
    }

    /**
     * cancel 撤销
     * @param businessActionContext
     * @return
     */
    @PostMapping(value = "rollback")
    @ResponseBody
    public boolean rollback(@RequestBody BusinessActionContext businessActionContext){
        return accountTccService.rollback(businessActionContext);
    }
}
```

TCC 方法参数说明：

- **Try**：第一个参数类型必须是 `com.alipay.dtx.client.core.api.domain.BusinessActionContext`，后续参数的个数和类型可以自定义。
- **Confirm**：有且仅有一个参数，参数类型必须是 `com.alipay.dtx.client.core.api.domain.BusinessActionContext`，后续为相应的参数名（`businessActionContext`）。

- **Cancel**：有且仅有一个参数，参数类型必须是 `com.alipay.dtx.client.core.api.domain.BusinessActionContext`，后续为相应的参数名（`businessActionContext`）。

返回类型说明：

- 二阶段的 Confirm 和 Cancel 方法的返回类型必须为 boolean 类型。
- 一阶段的 Try 方法可以为 boolean 类型，也可以自定义返回值。

实现接口

TCC 接口实现支持用户自定义，实现类框架如下：

- **基于 SOFARPC 和 Dubbo 框架**

```
public class TccActionImpl implements TccAction {
    @Override
    public boolean try(
    BusinessActionContext businessActionContext, int a, int b){

    }

    @Override
    public boolean confirm(BusinessActionContext businessActionContext) {

    }

    @Override
    public boolean cancel(BusinessActionContext businessActionContext) {

    }
}
```

- **基于 Spring Cloud 框架**

  根据参与方定义的 Try、Cancel、Confirm，定义发起方调用的 Feign API。Try 方法需要加上 `@TwoPhaseBusinessAction` 注解（需引入 `com.alipay.dtx.client.core.spi.TwoPhaseBusinessAction` 包）。

```
//使用FeignClient声明需要调用的参与方
@FeignClient(name = "account")
public interface AccountTccAction {

    /**
     * try 尝试
     *
     * BusinessActionContext 上下文对象，用来在两个阶段之间传递数据
     * BusinessActionContextParameter 注解的参数数据会被存入 BusinessActionContext
     * TwoPhaseBusinessAction 注解中commitMethod、rollbackMethod 属性有默认值，可以不写
     *
     * @param businessActionContext
     * @param userId
     * @param money
     * @return
     */
    @TwoPhaseBusinessAction(name = "accountTccAction", appName = "account")
    @PostMapping(value = "/account/decrease-money")
    boolean prepareDecreaseMoney(@RequestBody BusinessActionContext businessActionContext
,
                                 @BusinessActionContextParameter(paramName = "userId")Lon
g userId,
                                 @BusinessActionContextParameter(paramName = "money")BigD
ecimal money);

    /**
     * commit 提交
     * @param businessActionContext
     * @return
     */
    @PostMapping(value = "/account/commit")
    boolean commit(@RequestBody BusinessActionContext businessActionContext);

    /**
     * cancel 撤销
     * @param businessActionContext
     * @return
     */
    @PostMapping(value = "/account/rollback")
    boolean rollback(@RequestBody BusinessActionContext businessActionContext);
}
```

各接口作用：

- **Try**：初步操作。完成所有业务检查，预留必须的业务资源。
- **Confirm**：确认操作。真正执行的业务逻辑，不作任何业务检查，只使用 Try 阶段预留的业务资源。因此，只要 Try 操作成功，Confirm 必定能成功。另外，Confirm 操作需满足幂等性，保证一笔分布式事务能且只能成功一次。
- **Cancel**：取消操作。释放 Try 阶段预留的业务资源。同样的，Cancel 操作也需要满足幂等性。

关于 TCC 参与者的实现规范，查看 更多信息。

发布服务

在完成接口定义与实现后，参与者还需要发布服务，才能被分布式事务的发起方远程调用，从而保证分布式场景下的数据一致性。

- **基于 SOFARPC 框架**

```
<bean id="tccActionImpl" class="com.xxx.xxx.TccActionImpl"/>
<sofa:service ref="tccActionImpl" interface="com.xxx.xxx.TccAction">
     <sofa:binding.tr/>
</sofa:service>
```

- **基于 Dubbo 框架**

```
<bean id="tccActionImpl" class="com.xxx.xxx.TccActionImpl"/>
<dubbo:service ref="tccActionImpl" interface="com.xxx.xxx.TccAction"/>
```

- **基于 Spring Cloud 框架**

  发起方支持通过 `@DtxTransaction` 注解或者调用 API 开启事务。

  - 注解方式

```
//注入参与者 Feign 客户端
private OrderTccAction orderTccAction;
private AccountTccAction accountTccAction;
private StorageTccAction storageTccAction;

@DtxTransaction(bizType = "createOrderByTcc")
@Override
public void createOrder(OrderDO orderDO) {
    String orderNo=this.generateOrderNo();
    //创建订单
    orderTccAction.prepareCreateOrder(null,
            orderNo,
            orderDO.getUserId(),
            orderDO.getProductId(),
            orderDO.getAmount(),
            orderDO.getMoney());
    //扣余额
    accountTccAction.prepareDecreaseMoney(null, orderDO.getUserId(),orderDO.getMoney())
;
    //扣库存
    storageTccAction.prepareDecreaseStorage(null, orderDO.getProductId(),orderDO.getAmo
unt());
}
```

- 调用 API 的方式

```
private DtxService dtxService;
//注入参与者 Feign 客户端
private OrderTccAction orderTccAction;
private AccountTccAction accountTccAction;
private StorageTccAction storageTccAction;
@Resource(name = "dtxLocalTransactionTemplate")
private TransactionTemplate dtxLocalTransactionTemplate;

@Override
public void createOrder(OrderDO orderDO) {
    try {
        dtxLocalTransactionTemplate.execute(status -> {

            String txId = dtxService.start("createOrderByTcc", UUIDGenUtil.genUUId() +
"_FAILED");
            String orderNo=this.generateOrderNo();
            //创建订单
            orderTccAction.prepareCreateOrder(null,
                    orderNo,
                    orderDO.getUserId(),
                    orderDO.getProductId(),
                    orderDO.getAmount(),
                    orderDO.getMoney());
            //扣余额
            accountTccAction.prepareDecreaseMoney(null, orderDO.getUserId(),orderDO.get
Money());
            //扣库存
            storageTccAction.prepareDecreaseStorage(null, orderDO.getProductId(),orderD
O.getAmount());
            retrun true;
        });

    } catch (Throwable t) {
    }
}
```

发起方配置

对于发起方的应用来说，如果参与者与发起方分别属于不同的应用，则发起方需要订阅参与者 RPC 服务。

跨服务 TCC 参与者服务订阅

- 基于 SOFARPC 发布的服务订阅

```
<sofa:reference id="tccAction" interface="com.xxx.xxx.TccAction">
<sofa:binding.tr/>
</sofa:reference>
```

- 基于 Dubbo 发布的服务订阅

```
<dubbo:reference id="tccAction" interface="com.xxx.xxx.TccAction"/>
```

开启分布式事务

- 为需要开启分布式事务的接口增加分布式事务注解 @DtxTransaction（`com.alipay.sofa.dtx.client.aop.annotation.DtxTransaction`），表明此方法内部需要开启分布式事务。

注意：注解需放在方法实现之前。

- 方法内部可以执行下列事务操作（无数量与先后顺序的限制）：
  - 调用本地 TCC 参与者的第一阶段 Try 方法调用。
  - 调用基于 SOFARPC 或 Dubbo 发布的跨应用 TCC 参与者的第一阶段 Try 方法。

业务方法正常返回则分布式事务提交，业务方法抛出异常则分布式事务回滚。

示例如下：

```
public class YourClass{
 @DtxTransaction(bizType="yourbizType")
 public void yourMethod(yourParams){
 try{
 // 调用 TCC 参与者一阶段try方法，try方法第一个参数值传'null';
 TCC.try();
 // 方法正常返回，事务提交
 }catch(Throwable t){
 // 出现异常，事务回滚
 throw t;
 }
 }
}
```

> **重要**
>
> 该方法所在类需要配置成 Spring Bean 才能被分布式事务扫描器识别。

`@DtxTransaction` 注解属性说明如下表所示。

参数	说明
`bizType`	必选项。该属性指定业务类型，由用户根据自己业务场景自定义，长度不超过 30 字符。
`timeout`	该属性指定事务超时时间，默认为 30 秒，您可以根据需要自定义，单位为秒。 超时会自动触发当前事务回滚。
`isolationLevel`	该属性指定分布式事务的隔离级别，默认是为 `READ_UNCOMMITTED`（读未提交）。目前只支持 `READ_UNCOMMITTED`（读未提交）和 `READ_COMMITTED`（读已提交）两种隔离级别，可按需配置。

发起方类配置 Bean

要让 Spring 框架扫描发起方的注解，需要让 Spring 框架感知到发起方的 Bean，以上述“YourClass”为例：

```
<bean id="yourClassBean" class="com.xxx.xxx.YourClass"/>
```

更多信息

TCC 参与者实现规范

- **实现两阶段接口**：二阶段入参只包含事务数据，参与者处理业务数据之前需要根据事务数据关联到具体的业务数据。
- **控制业务幂等性**：参与者需要支持同一笔事务的重复提交和重复回滚，保证一笔事务能且只能成功一次。
- **数据可见性控制**：当一笔分布式事务正在处理中，此时如果有查询，则需要兼容未处理完的中间数据的可见性。一般通过文案展示告诉用户中间数据的存在，例如告诉用户当前冻结的金额有多少。
- **隔离性控制**：对于状态类数据，需要提供隔离性控制来允许不同事务操作同一个业务资源。例如账户余额，不同事务操作的金额是隔离的。
- **允许空回滚，拒绝空提交**：回滚请求处理时，如果对应的具体业务数据为空，则返回成功。提交请求处理时，如果对应的具体业务数据为空，则返回失败。

3.5. FMT 模式接入

分布式事务支持以 FMT 模式接入 SOFARPC 和 Dubbo 远程服务框架。FMT 模式参与者只需要替换数据源，即可将对该数据源的访问纳入到分布式事务中。一个数据源就是一个参与者。

配置数据源

您可以选择以下任一数据源配置方案：

- **封装数据源**

  您可以选择使用自选的数据源，然后使用分布式事务数据源代理（`com.alipay.sofa.dtx.datasource.WrappedDtxDataSource`）封装该数据源，配置示例如下：

  原代码：

  ```
  <!-- 自选的数据源，可选择任何类型 DataSource -->
  <bean id="xxxDataSource" class="com.xxx.xxx.XDataSource">
  <!-- 此处为自选数据源的各种参数配置 -->
  </bean>
  ```

  修改为：

  ```
  <bean id="xxxDataSource" class="com.alipay.sofa.dtx.datasource.WrappedDtxDataSource">
       <property name="targetDataSource" ref="xxxDataSource_inner"/>
       <property name="uniqueDbId" value="[数据源全局唯一的 ID]"/>
       <property name="dbType" value="[数据源类型]"/>
  </bean>
  ```

  业务应用访问数据库时，需要使用代理后的 `xxxDataSource` 作为 DataSource Bean。

  分布式事务代理数据源 WrappedDtxDataSource 属性介绍：

- `targetDataSource`：指定用户自选数据源 Bean。
- `uniqueDbId`：事务参与方标识，该标识应该是全局唯一的（与 TCC 模式注解 name 属性值作用相同）。
- `dbType`：表示数据源类型，支持 MySQL、Oracle 以及 OceanBase。

- **替换数据源**

  您可以选择直接使用分布式事务提供的数据源（`com.alipay.sofa.dtx.datasource.druid.DruidDtxDataSource`），配置示例如下：

  原代码：

```
<!-- 自选的数据源，可选择任何类型 DataSource -->
<bean id="xxxDataSource" class="com.xxx.xxx.XDataSource">
<!-- 此处为自选数据源的各种参数配置 -->
</bean>
```

  修改为：

```
<bean id="xxxDataSource" class="com.alipay.sofa.dtx.datasource.druid.DruidDtxDataSource">
     <property name="jdbcUrl" value="${jdbcUrl}"/>
     <property name="userName" value="${jdbc userName}"/>
     <property name="passWord" value="${jdbc passWord]}"/>
     <property name="uniqueDbId" value="[数据源全局唯一的ID]"/>
     <property name="dbType" value="[数据源类型]"/>
</bean>
```

> **重要**
>
> 分布式事务数据源完全兼容 Druid 数据源，如果是其他数据源，可能需要修改注入参数。

分布式事务数据源 DruidDtxDataSource 属性介绍

必填属性：

参数	说明
`jdbcUrl`	属性指定数据库的 JDBC URL。
`userName`	属性指定访问数据库的用户名。
`passWord`	属性指定访问数据库的用户密码。
`uniqueDbId`	属性表示数据源的唯一 ID，需要全局唯一，由用户自定义。
`dbType`	属性指定数据源类型，MySQL、Oracle 或 OceanBase。

可选属性：

参数	说明
`initialSize`	数据库链接池初始连接数。
`minIdle`	数据库链接池最小连接数。
`maxActive`	数据库链接池最大连接数。
`poolPreparedStatements`	是否缓存 preparedStatement。
`maxOpenPreparedStatements`	要启用 `poolPreparedStatements` ，必须配置大于 0，当大于 0 时， `poolPreparedStatements` 自动触发修改为 true。
`maxWait`	获取连接时最大等待时间，单位为毫秒。
`validationQuery`	用来检测连接是否有效的 SQL，要求是一个查询语句。如果 `validationQuery` 为 NULL，那么 `testOnBorrow` 、 `testOnReturn` 、 `testWhileIdle` 都不会起作用。
`testOnBorrow`	申请连接时执行 `validationQuery` 检测连接是否有效。
`testOnReturn`	归还连接时执行 `validationQuery` 检测连接是否有效。
`testWhileIdle`	如果空闲时间大于 `timeBetweenEvictionRunsMillis` ，执行 `validationQuery` 检测连接是否有效。
`timeBetweenEvictionRunsMillis`	Destroy 线程会检测连接的间隔时间，以 `testWhileIdle` 的判断为依据，详见 `testWhileIdle` 属性说明。

创建数据表

FMT 模式需要在数据库中记录事务信息，所以您需要在与上述配置相同的数据库中创建分布式事务的快照数据（ `dtx_branch_info` ）和保存行锁的表（ `dtx_row_lock` ）。

如果业务库已分库分表，您只需在每一个库中分别创建一张分布式事务的表即可，这些表不需要拆分。

根据不同的数据库类型，可参考下方更多信息中的 MySQL、OceanBase、Oracle 建表语句。

发布服务

分布式事务内可能会操作多个数据库，这些数据库的访问服务由多个不同的应用提供，即跨服务 FMT 参与者。这些应用内使用了 FMT 模式参与者后，需要将包含 DAO 操作的方法发布成 SOFARPC 服务或 Dubbo 服务。这样就可以被分布式事务的发起方进行调用，来保证分布式场景下的数据一致性。

假设该应用数据访问服务实现如下：

```
public class DataServiceImpl implements com.xxx.DataService{
//服务实现
}
```

- **基于 SOFARPC 的服务发布**

  如果需要发布为 SOFARPC 服务，服务发布代码如下：

```
<bean id="dataServiceImpl" class="com.xxx.DataServiceImpl"/>

<sofa:service ref="dataServiceImpl" interface=com.xxx.DataService" >
      <sofa:binding.tr/>
</sofa:service>
```

- **基于 Dubbo 的服务发布**

  如果需要发布为 Dubbo 服务，服务发布代码如下：

```
<bean id="dataServiceImpl" class="com.xxx.DataServiceImpl"/>

<dubbo:service ref="dataServiceImpl" interface=com.xxx.DataService" />
```

发起方配置

对于发起方的应用来说，如果参与者与发起方分别属于不同的应用，则发起方需要订阅参与者 RPC 服务。

跨服务 FMT 参与者服务订阅

- **基于 SOFARPC 发布的服务订阅**

```
<sofa:reference id="dataServiceSofaRpc" interface=com.xxx.DataService" >
<sofa:binding.tr/>
</sofa:reference>
```

- **基于 Dubbo 发布的服务订阅**

```
<dubbo:reference id="dataServiceDubbo" interface=com.xxx.DataService" />
```

开启分布式事务

- 为需要开启分布式事务的接口增加分布式事务注解 `@DtxTransaction`（`com.alipay.sofa.dtx.client.aop.annotation.DtxTransaction`），表明此方法内部需要开启分布式事务。

  > **重要**
  >
  > 注解需放在方法实现之前。

- 方法内部可以执行下列事务操作（无数量与先后顺序的限制）：

- 操作本地 FMT 参与者的 DAO 操作。
- 调用基于 SOFARPC 或 Dubbo 发布的普通服务，该服务内再操作 FMT 参与者，跨服务 FMT 参与者。

业务方法正常返回则分布式事务提交，业务方法抛出异常则分布式事务回滚。

示例如下：

```
public class YourClass{
 @DtxTransaction(bizType="yourbizType")
 public void yourMethod(yourParams){
 try{
 //可以是 FMT 参与者数据源
 DAO1;

 DAO2;

 ......

 DAO N;


 // 普通 rpc 调用，在该调用方法内可以访问 FMT 参与者数据源
 dataServiceSofaRpc.method();

 // 方法正常返回，事务提交
 }catch(Throwable t){
 // 出现异常，事务回滚
 throw t;
 }
 }
}
```

重要

该方法所在类需要配置成 Spring Bean 才能被分布式事务扫描器识别。

`@DtxTransaction` 注解属性说明如下表所示。

参数	说明
`bizType`	必选项。该属性指定业务类型，由用户根据自己业务场景自定义，长度不超过 30 字符。
`timeout`	该属性指定事务超时时间，默认为 30 秒，您可以根据需要自定义，单位为秒。 超时会自动触发当前事务回滚。
`isolationLevel`	该属性指定分布式事务的隔离级别，默认是为 `READ_UNCOMMITTED` （读未提交）。目前只支持 `READ_UNCOMMITTED` （读未提交）和 `READ_COMMITTED` （读已提交）两种隔离级别，可按需配置。

发起方类配置 Bean

要让 Spring 框架扫描发起方的注解，需要让 Spring 框架感知到发起方的 Bean，以上述“YourClass”为例：

```
<bean id="yourClassBean" class="com.xxx.xxx.YourClass"/>
```

更多信息

MySQL、OceanBase、Oracle 建表语句

- **MySQL**

  记录锁表：

```
CREATE TABLE `dtx_row_lock`(
`action_id` varchar(128) NOT NULL COMMENT '分支事务号',
`tx_id` varchar(128) NOT NULL COMMENT '主事务号',
`table_name` varchar(64) DEFAULT NULL COMMENT '表名称',
`row_key` varchar(250) NOT NULL COMMENT '行唯一key',
`instance_id` varchar(32) NOT NULL COMMENT '实例id',
`context` varchar(2000) DEFAULT NULL COMMENT '分支上下文',
`feature` varchar(2000) DEFAULT NULL COMMENT '扩展属性',
`gmt_create` datetime NOT NULL COMMENT '创建时间',
`gmt_modified` datetime NOT NULL COMMENT '修改时间',
PRIMARY KEY (`row_key`),
KEY `idx_row_lock_txid_action_id`(`action_id`,`tx_id`,`instance_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='行锁'
;
```

> **重要**
>
> 原生 MySQL 中需要修改 `CHARSET=latin1` 。MySQL 每个单表中所创建的索引长度是有限制的，Innodb 引擎的长度不超过 767 字节。 `row_key` varchar(250) 表示 250 个字符，字符集使用的 utf8，可能会导致超过限制。

  分支事务记录表：

```
CREATE TABLE `dtx_branch_info`(
`action_id` varchar(128) NOT NULL COMMENT '分支事务号',
`tx_id` varchar(128)  NOT NULL  COMMENT '主事务号',
`status` varchar(4)  COMMENT '事务状态',
`log_info` blob DEFAULT NULL COMMENT 'undo/redo log',
`biz_type` varchar(32) DEFAULT NULL COMMENT '发起方业务类型',
`instance_id` varchar(32) NOT NULL COMMENT '实例id',
`context` varchar(2000) DEFAULT NULL COMMENT '分支上下文',
`feature` varchar(2000) DEFAULT NULL COMMENT '扩展属性',
`gmt_create` datetime NOT NULL COMMENT '创建时间',
`gmt_modified` datetime NOT NULL COMMENT '修改时间',
   PRIMARY KEY (`action_id`),
   KEY `idx_txId_action_id`(`action_id`,`tx_id`,`instance_id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COMMENT='分支记录日志'
;
```

- **OceanBase v1.0**

  记录锁表：

```
CREATE TABLE `dtx_row_lock`(
`action_id` varchar(128) NOT NULL COMMENT '分支事务号',
`tx_id` varchar(128) NOT NULL COMMENT '主事务号',
`table_name` varchar(64) DEFAULT NULL COMMENT '表名称',
`row_key` varchar(250) NOT NULL COMMENT '行唯一key',
`instance_id` varchar(32) NOT NULL COMMENT '实例id',
`context` varchar(2000) DEFAULT NULL COMMENT '分支上下文',
`feature` varchar(2000) DEFAULT NULL COMMENT '扩展属性',
`gmt_create` datetime NOT NULL COMMENT '创建时间',
`gmt_modified` datetime NOT NULL COMMENT '修改时间',
PRIMARY KEY (`row_key`),
KEY `idx_row_lock_txid_action_id`(`action_id`,`tx_id`,`instance_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='行锁'
;
```

  分支事务记录表：

```
CREATE TABLE `dtx_branch_info`(
`action_id` varchar(128) NOT NULL COMMENT '分支事务号',
`tx_id` varchar(128)  NOT NULL  COMMENT '主事务号',
`status` varchar(4)  COMMENT '事务状态',
`log_info` varchar(32768) DEFAULT NULL COMMENT 'undo/redo log',
`biz_type` varchar(32) DEFAULT NULL COMMENT '发起方业务类型',
`instance_id` varchar(32) NOT NULL COMMENT '实例id',
`context` varchar(2000) DEFAULT NULL COMMENT '分支上下文',
`feature` varchar(2000) DEFAULT NULL COMMENT '扩展属性',
`gmt_create` datetime NOT NULL COMMENT '创建时间',
`gmt_modified` datetime NOT NULL COMMENT '修改时间',
   PRIMARY KEY (`action_id`),
   KEY `idx_txId_action_id`(`action_id`,`tx_id`,`instance_id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COMMENT='分支记录日志'
;
```

- **Oracle**

  记录锁表：

```
CREATE TABLE dtx_row_lock(
  action_id varchar2(64) NOT NULL,
  tx_id varchar2(128) NOT NULL,
  table_name varchar2(64) DEFAULT NULL,
  row_key varchar2(250) PRIMARY KEY,
  instance_id varchar2(32) NOT NULL,
  context varchar2(2000) DEFAULT NULL,
  feature varchar2(2000) DEFAULT NULL,
  gmt_create timestamp NOT NULL,
  gmt_modified timestamp NOT NULL
);
CREATE INDEX idx_row_lock_txid_action_id ON dtx_row_lock(action_id , tx_id , instance_id)
;
```

分支事务记录表：

```
CREATE TABLE dtx_branch_info (
   action_id varchar2(64) PRIMARY KEY,
   tx_id varchar2(128) NOT NULL,
   status varchar2(4),
   log_info blob DEFAULT NULL,
   biz_type varchar2(32) DEFAULT NULL,
   instance_id varchar2(32) NOT NULL,
   context varchar2(2000) DEFAULT NULL,
   feature varchar2(2000) DEFAULT NULL,
   gmt_create timestamp NOT NULL,
   gmt_modified timestamp NOT NULL
);
CREATE INDEX idx_txId_action_id ON dtx_branch_info(action_id , tx_id , instance_id);
```

3.6. Saga 模式接入

在 Saga 模式的业务流程中，每个参与者都会提交本地事务。当某一个参与者失败，则补偿之前已经成功的参与者。一阶段正向服务和二阶段补偿服务均由业务开发实现。

本文将基于 Saga 模式的示例工程 引导您快速体验 Saga 模式的分布式应用事务开发。

1. 依赖配置：在本地工程项目中，添加分布式事务依赖与相关配置项等。
2. 参与者开发：进行本地事务参与者开发，实现相应的方法与接口。
3. 业务开发：设计编排实际业务流程，通过状态机引擎启动该业务。

依赖与配置项

具体操作步骤说明请参见依赖与配置项。

参与者开发

- **基于 SOFABoot 和 Spring 框架**

  完成依赖与配置项操作后，即可进行参与者应用开发。此处提供两个参与者示例，分别是转账交易中的扣钱与加钱参与者。您可以根据实际业务，自定义正向和补偿操作方法名。

  - **扣钱参与者**

a. 实现扣钱方法和失败补偿方法，示例如下：

```
/**
 * Saga扣钱参与者
 */
public interface FirstSagaAction{

    /**
     * 扣钱操作
     * @param businessKey
     * @param accountNo
     * @param amount
     * @return
     */
    boolean amountMinus(String businessKey,String accountNo,BigDecimal amount,Map<Str
ing,Object> extParams);

    /**
     * 补偿（冲正）扣钱操作
     * @param businessKey
     * @param accountNo
     * @return
     */
    boolean compensateAmountMinus(String businessKey,String accountNo);
}
```

b. 实现接口，示例如下：

```
public class FirstSagaActionImpl implements FirstSagaAction{

        protected final static Logger logger =LoggerFactory.getLogger(FirstSagaAction
Impl.class);

        //账户dao
        private AccountDAO            firstAccountDAO;
        //业务流水dao
        private AccountTransactionDAO firstAccountTransactionDAO;
        //事务模板
        private TransactionTemplate   firstActionTransactionTemplate;

        @Override
        public boolean amountMinus(final String businessKey,final String accountNo,fi
nal BigDecimal amount,final Map<String,Object> extParams){
            try{
                if(amount.compareTo(BigDecimal.ZERO)<0){
                    throw new RuntimeException("金额必须大小0");
                }

                AccountTransaction accountTransaction = firstAccountTransactionDAO.fi
ndTransaction(businessKey);
                if(accountTransaction !=null&&Status.SUCCESS.name().equals(accountTra
nsaction.getStatus())){
                    //幂等控制：交易已成功,直接返回成功
                    return true;
                }
```

```
                }

                return firstActionTransactionTemplate.execute(newTransactionCallback<
Boolean>(){
                    @Override
                    public Boolean doInTransaction(TransactionStatus status){
                        try{
                            //校验账户余额
                            Account account = firstAccountDAO.getAccountForUpdate(acc
ountNo);
                            if(account.getAmount().compareTo(amount)<0){
                                throw new RuntimeException("余额不足");
                            }
                            //记录账户操作流水
                            AccountTransaction accountTransaction =new AccountTransac
tion();
                            accountTransaction.setBusinessKey(businessKey);
                            accountTransaction.setAccountNo(accountNo);
                            accountTransaction.setAmount(amount);
                            accountTransaction.setType(Type.MINUS.name());
                            accountTransaction.setStatus(Status.SUCCESS.name());
                            firstAccountTransactionDAO.addTransaction(accountTransact
ion);

                            //扣钱
                            BigDecimal amount = account.getAmount().subtract(accountT
ransaction.getAmount());
                            if(amount.compareTo(BigDecimal.ZERO)<0){
                                throw new RuntimeException("余额不足");
                            }
                            account.setAmount(amount);
                            firstAccountDAO.updateAmount(account);
                        }catch(Exception e){
                            logger.error("扣钱操作失败", e);
                            throw new RuntimeException("扣钱操作失败", e);
                        }
                        return true;
                    }
                });
            }catch(Exception e){
                logger.error("扣钱操作失败", e);
                return false;
            }
        }

        @Override
        public boolean compensateAmountMinus(final String businessKey,final String ac
countNo){
            AccountTransaction accountTransaction;
            try{
                accountTransaction = firstAccountTransactionDAO.findTransaction(busin
essKey);
                if(accountTransaction ==null){
                    //原交易流水不存在，记录防悬挂(后发先至)流水
                    accountTransaction =new AccountTransaction();
```

```
                    accountTransaction.setBusinessKey(businessKey);
                    accountTransaction.setAccountNo(accountNo);
                    accountTransaction.setType(Type.MINUS.name());
                    accountTransaction.setStatus(Status.COMPENSATED.name());
                    firstAccountTransactionDAO.addTransaction(accountTransaction);
                    return true;
                }
                if(Status.COMPENSATED.name().equals(accountTransaction.getStatus())){
                    //幂等控制：补偿已成功,直接返回成功
                    return true;
                }

                final AccountTransaction accountTransactionFinal = accountTransaction
;
                return firstActionTransactionTemplate.execute(new TransactionCallback
<Boolean>(){

                    @Override
                    public Boolean doInTransaction(TransactionStatus status){
                        try{
                            //补偿已扣减金额
                            Account account = firstAccountDAO.getAccountForUpdate(acc
ountTransactionFinal.getAccountNo());
                            BigDecimal amount = account.getAmount().add(accountTransa
ctionFinal.getAmount());
                            account.setAmount(amount);
                            firstAccountDAO.updateAmount(account);
                            //更新流水状态
                            accountTransactionFinal.setStatus(Status.COMPENSATED.name
());
                            firstAccountTransactionDAO.updateTransaction(accountTrans
actionFinal);
                            return true;
                        }catch(Exception e){
                            logger.error("扣钱操作补偿失败", e);
                            status.setRollbackOnly();
                            return false;
                        }
                    }
                });
            }catch(SQLException e){
                logger.error("扣钱操作补偿失败", e);
                return false;
            }
        }

        public void setFirstAccountDAO(AccountDAO firstAccountDAO){
            this.firstAccountDAO = firstAccountDAO;
        }

        public void setFirstAccountTransactionDAO(AccountTransactionDAO firstAccountT
ransactionDAO){
            this.firstAccountTransactionDAO = firstAccountTransactionDAO;
        }
```

```
        public void setFirstActionTransactionTemplate(TransactionTemplate firstAction
TransactionTemplate){
            this.firstActionTransactionTemplate = firstActionTransactionTemplate;
        }
    }
```

- **加钱参与者**

a. 实现正向加钱方法和失败补偿方法，示例如下：

```
/**
* Saga加钱 参与者
*/
public interface SecondSagaAction{

/**
  * 加钱操作
  * @param businessKey
  * @param accountNo
  * @param amount
  * @return
  */
boolean amountAdd(String businessKey,String accountNo,BigDecimal amount,Map<String,Ob
ject> extParams);

/**
  * 补偿（冲正）加钱操作
  * @param businessKey
  * @param accountNo
  * @return
  */
boolean compensateAmountAdd(String businessKey,String accountNo);
}
```

b. 实现接口，示例如下：

```
@Override
    public boolean compensateAmountAdd(final String businessKey,final String accountN
o){

        AccountTransaction accountTransaction;
        try{
            accountTransaction = secondAccountTransactionDAO.findTransaction(business
Key);
            if(accountTransaction ==null){
                //防悬挂：原交流流水不存在，记录防悬挂（后发先至）流水
                accountTransaction =new AccountTransaction();
                accountTransaction.setBusinessKey(businessKey);
                accountTransaction.setAccountNo(accountNo);
                accountTransaction.setType(Type.ADD.name());
                accountTransaction.setStatus(Status.COMPENSATED.name());
                secondAccountTransactionDAO.addTransaction(accountTransaction);

                //允许空补偿：返回补偿成功
```

```
                return true;
            }
            if(Status.COMPENSATED.name().equals(accountTransaction.getStatus())){
                //幂等控制：补偿已成功,直接返回成功
                return true;
            }

            final AccountTransaction accountTransactionFinal = accountTransaction;
            return secondActionTransactionTemplate.execute(new TransactionCallback<Bo
olean>(){

                @Override
                public Boolean doInTransaction(TransactionStatus status){
                    try{
                        //扣回已加金额
                        Account account = secondAccountDAO.getAccountForUpdate(accoun
tTransactionFinal.getAccountNo());
                        BigDecimal amount = account.getAmount().subtract(accountTrans
actionFinal.getAmount());
                        if(amount.compareTo(BigDecimal.ZERO)<0){
                            throw new RuntimeException("余额不足, 请工人核对");
                        }
                        account.setAmount(amount);
                        secondAccountDAO.updateAmount(account);
                        //更新流水状态
                        accountTransactionFinal.setStatus(Status.COMPENSATED.name());
                        secondAccountTransactionDAO.updateTransaction(accountTransact
ionFinal);
                        return true;
                    }catch(Exception e){
                        logger.error("加钱操作补偿失败", e);
                        status.setRollbackOnly();
                        return false;
                    }
                }
            });

        }catch(SQLException e){
            logger.error("加钱操作补偿失败", e);
            return false;
        }
    }

    public void setSecondAccountDAO(AccountDAO secondAccountDAO){
        this.secondAccountDAO = secondAccountDAO;
    }

    public void setSecondAccountTransactionDAO(AccountTransactionDAO secondAccountTra
nsactionDAO){
        this.secondAccountTransactionDAO = secondAccountTransactionDAO;
    }

    public void setSecondActionTransactionTemplate(TransactionTemplate secondActionTr
ansactionTemplate){
```

```
        this.secondActionTransactionTemplate = secondActionTransactionTemplate;
    }
}
```

- **基于 Spring Cloud 框架**

  在 Controller 实现扣钱方法和失败补偿方法，示例如下：

```
@RestController
@RequestMapping("account")
public class AccountController {

    private Logger logger = LoggerFactory.getLogger(getClass());

    @Resource
    private AccountService accountServiceImpl;

    @RequestMapping("decrease")
    @ResponseBody
    public boolean decrease(@RequestParam("businessKey") String businessKey, @RequestPara
m("userId") Long userId, @RequestParam("money") BigDecimal money){
        logger.info("businessKey:{}", businessKey);
        return accountServiceImpl.decrease(userId,money);
    }

    @RequestMapping("compensateDecrease")
    public boolean compensateDecrease(@RequestParam("businessKey") String businessKey, @R
equestParam("userId") Long userId, @RequestParam("money") BigDecimal money){
        logger.info("businessKey:{}", businessKey);
        return accountServiceImpl.compensateDecrease(userId, money);
    }
}
```

发起方开发

完成参与者开发后，您需要通过分布式事务控制台提供的状态设计器进行业务服务编排，如编排成转账交易服务。操作步骤如下：

1. 基于 Spring Cloud 框架编排业务服务时需要定义调用参与者的 Feign API，对应参与者的扣钱方法和失败补偿方法。

```
//使用FeignClient声明需要调用的参与方
@FeignClient(value = "account-server")
public interface AccountApi {

    /**
     * 扣减账户余额
     * @param businessKey
     * @param userId 用户id
     * @param money 金额
     * @return
     */
    @RequestMapping("/account/decrease")
    boolean decrease(@RequestParam("businessKey") String businessKey, @RequestParam("user
Id") Long userId, @RequestParam("money") BigDecimal money);

    /**
     * 交易补偿，把扣减的账户金额加回去.
     *
     * @param businessKey
     * @param userId userId
     * @param money money
     * @return the boolean
     */
    @RequestMapping("/account/compensateDecrease")
    boolean compensateDecrease(@RequestParam("businessKey") String businessKey, @RequestP
aram("userId") Long userId, @RequestParam("money") BigDecimal money);

}
```

2. 编排事务。
   i. 登录分布式事务控制台，在左侧导航栏中选择 **应用事务 > 事务配置**。
   ii. 单击 **创建应用事务 > Saga 模式**。
   iii. 在 **创建 Saga 事务**页面，根据提示配置以下信息：
      - **应用名称**：可从下拉菜单中选择现有应用或新建一个应用。
      - **BizType**：填写要创建的 Saga 事务的业务流程名称。
      - **描述**：对该事务的备注信息。
   iv. 单击 **下一步**，进入设计器页面，开始编排业务流程。
      a. 以便您快速体验，单击此处直接下载一个已经编排好的 JSON 文件。
      b. 单击编辑区上方的 JSON View 图标，并将下载的 JSON 语句粘贴到设计器的 JSON View 框中。
      c. 单击 Designer View 图标，切换回设计器模式，即可查看具体的业务流程图。
   v. 将刚刚设计器生成的 JSON 保存到工程中。如在示例工程将其保存至 `account-demo-saga/account-demo-service/src/main/resources/statelang/transfer.json` 。

vi. 在工程中配置状态机引擎，参见示例工程文件 `account-demo-saga/account-demo-service/src/main/resources/META-INF/spring/transfer-bean.xml` 。

```
<bean id="stateMachineEngine" class="io.seata.saga.engine.impl.ProcessCtrlStateMachineEngine">
    <property name="stateMachineConfig" ref="dbStateMachineConfig"></property>
</bean>
<bean id="dbStateMachineConfig" class="com.alipay.dtx.client.saga.config.DtxSagaStateMachineConfig">
    <property name="dataSource" ref="transferDataSource"></property>
    <property name="resources" value="classpath:statelang/*.json"></property>
    <property name="enableAsync" value="true"></property>
    <property name="threadPoolExecutor" ref="threadExecutor"></property>
</bean>

<bean id="threadExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolExecutorFactoryBean">
    <property name="threadNamePrefix" value="SAGA_ASYNC_EXE_"/>
    <property name="corePoolSize" value="20"/>
    <property name="maxPoolSize" value="20"/>
    <property name="queueCapacity" value="100"/>
    <property name="rejectedExecutionHandler" ref="callerRunsPolicy"/>
</bean>

<bean name="callerRunsPolicy" class="java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy"></bean>

<bean class="io.seata.saga.rm.StateMachineEngineHolder">
    <property name="stateMachineEngine" ref="stateMachineEngine"/>
</bean>
```

说明

本文的示例工程中是用的 h2 数据源，且自动建表。Saga 事务日志建表语句，可参考 Saga 模式建表语句示例。

vii. 在 Java 代码中，通过状态机引擎启动业务流程：

- **基于 SOFABoot 和 Spring 框架**

```
/**
*  转账交易
*/
public interface TransferService{

/**
  * Saga转账
  *
  * @param from
  * @param to
  * @param amount
  * @return
  */
boolean transferBySaga(final String from,final String to,final BigDecimal amount,Map<
String,Object> extParams);
}
```

对应的接口实现代码示例如下：

```
public class TransferServiceImpl implements TransferService{

        protected final static Logger logger =LoggerFactory.getLogger(TransferService
Impl.class);

        @Autowired
        private StateMachineEngine stateMachineEngine;

        @Override
        public boolean transferBySaga(String from,String to,BigDecimal amount,Map<Str
ing,Object> extParams){
            try{
                String businessKey = UUID.randomUUID().toString().replaceAll("-","");

                Map<String,Object> params =new HashMap<>(4);
                params.put("from", from);
                params.put("to", to);
                params.put("amount", amount);
                if(extParams !=null){
                    params.put("extParams", extParams);
                }

                StateMachineInstance inst = stateMachineEngine.startWithBusinessKey("
transferBySaga",null, businessKey, params);

                if(ExecutionStatus.SU.equals(inst.getStatus())
                        && inst.getCompensationStatus() == null){
                    //正向状态为成功，补偿状态为空（没为触发回滚），则交易成功
                    return true;
                }
                else{
                    //如果执行到了Fail节点会有errorCode和errorMessage
                    String errorCode =(String) inst.getContext().get(DomainConstants.
VAR_NAME_STATEMACHINE_ERROR_CODE);
                    String errorMessage =(String) inst.getContext().get(DomainConstan
ts.VAR_NAME_STATEMACHINE_ERROR_MSG);
                    System.out.println("ErrorCode:"+ errorCode +", ErrorMsg:"+ errorM
essage +", exception: "+ inst.getException());
                    return false;
                }

            }catch(Throwable t){
                logger.error("转账交易执行失败", t);
                throw new RuntimeException(t);
            }
        }

        public void setStateMachineEngine(StateMachineEngine stateMachineEngine){
            this.stateMachineEngine = stateMachineEngine;
        }
    }
```

- 基于 Spring Cloud 框架

```
@Override
public boolean create(Order order) {
    LOGGER.info("------->交易开始");

    StateMachineEngine stateMachineEngine = (StateMachineEngine) ApplicationContextUt
ils.getApplicationContext().getBean("stateMachineEngine");

    Map<String, Object> startParams = new HashMap<>(3);
    String businessKey = String.valueOf(System.currentTimeMillis());
    startParams.put("businessKey", businessKey);
    startParams.put("order", order);
    startParams.put("mockReduceAccountFail", "true");
    startParams.put("userId", order.getUserId());
    startParams.put("money", order.getPayAmount());
    startParams.put("productId", order.getProductId());
    startParams.put("count", order.getCount());

    //sync test
    StateMachineInstance inst = stateMachineEngine.startWithBusinessKey("buyGoodsOnli
ne", null, businessKey, startParams);
    Assert.isTrue(ExecutionStatus.SU.equals(inst.getStatus()), "saga transaction exec
ute failed. XID: " + inst.getId());
    System.out.println("saga transaction commit succeed. XID: " + inst.getId());
    return true;
}
```

3.7. Saga 状态机配置

Saga 模式事务是基于 **状态机引擎** 来实现的。分布式事务控制台提供了一个状态设计器，您可以通过该设计器自编排设计状态图，定义各节点的详细信息与属性配置等。

具体的实现机制如下：

- 通过状态图定义服务调用的流程，并生成 JSON 状态语言定义文件。
- 状态图中一个节点可以是调用一个服务，节点可以配置它的补偿节点。
- JSON 状态定义文件由状态机引擎驱动执行。当出现异常时，状态引擎反向执行已成功节点对应的补偿节点将事务回滚。

> **说明**
>
> 异常发生时，您也可自定义决定是否进行补偿。可以实现服务编排需求，支持单项选择、并发、子流程、参数转换、参数映射、服务执行状态判断、异常捕获等功能。

本文详细介绍 **状态机** 与 **状态** 的各个属性，以便您进行业务流程的设计编排。

状态机的属性

状态机的属性如下：

```
{
     "Name":"reduceInventoryAndBalance",
     "Comment":"reduce inventory then reduce balance in a transaction",
     "StartState":"ReduceInventory",
     "Version":"0.0.1",
     "States":{}
}
```

属性说明

参数	说明
`Name`	表示状态机的名称，必须唯一。
`Comment`	状态机的描述。
`Version`	状态机的版本。
`StartState`	启动时运行的第一个 **状态**。
`States`	状态列表，是一个 map 结构。key 是 **状态** 的名称，在状态机内必须唯一；value 是一个 map 结构，表示 **状态** 的属性列表。

状态的属性

ServiceTask

```
"States":{
            ...
            "ReduceBalance":{
                "Type":"ServiceTask",
                "ServiceName":"balanceAction",
                "ServiceMethod":"reduce",
                "CompensateState":"CompensateReduceBalance",
                "IsForUpdate":true,
                "IsPersist":true,
                "IsAsync":false,
                "Input":[
                    "$.[businessKey]",
                    "$.[amount]",
                    {
                    "throwException":"$.[mockReduceBalanceFail]"
                    }
                ],
                "Output":{
                    "compensateReduceBalanceResult":"$.#root"
                },
                "Status":{
                    "#root == true":"SU",
                    "#root == false":"FA",
                    "$Exception{java.lang.Throwable}":"UN"
                },
                "Retry":[
                    {
                    "Exceptions":["io.seata.saga.engine.mock.DemoException"],
                    "IntervalSeconds":1.5,
                    "MaxAttempts":3,
                    "BackoffRate":1.5
                    },
                    {
                    "IntervalSeconds":1,
                    "MaxAttempts":3,
                    "BackoffRate":1.5
                    }
                ],
                "Catch":[
                    {
                    "Exceptions":[
                    "java.lang.Throwable"
                    ],
                    "Next":"CompensationTrigger"
                    }
                ],
                "Next":"Succeed"
            }
            ...
        }
```

属性说明

参数	说明
`ServiceName`	服务名称，通常是服务的 beanId。
`ServiceMethod`	服务方法名称。
`CompensateState`	该状态的补偿状态。
`IsForUpdate`	标识该服务是否会更新数据，默认是 false。如果配置了 `CompensateState` ，则默认是 true。有补偿服务的服务肯定是数据更新类服务。
`IsPersist`	执行日志是否进行存储，默认是 true。有一些查询类的服务可以配置为 false，执行日志不进行存储提高性能，因为当异常恢复时可以重复执行。
`IsAsync`	异步调用服务。因为异步调用服务会忽略服务的返回结果，所以用户定义的服务执行状态映射（下文的 status 属性）将被忽略，默认为服务调用成功。如果提交异步调用失败（比如线程池已满），则为服务执行状态失败。
`Input`	调用服务的输入参数列表，是一个数组，对应于服务方法的参数列表， `$.` 表示使用表达式从状态机上下文中取参数，表达使用的 SpringEL。如果是常量，直接写值即可。
`Output`	将服务返回的参数赋值到状态机上下文中，是一个 map 结构。 • key 为放入到状态机上文时的 key（状态机上下文也是一个 map）。 • value 中 `$.` 是表示 SpringEL 表达式，表示从服务的返回参数中取值， `#root` 表示服务的整个返回参数。

`Status`	服务执行状态映射，框架定义了三个状态：`SU` 成功、`FA` 失败、`UN` 未知。需要把服务执行的状态映射成这三个状态，帮助框架判断整个事务的一致性，是一个 map 结构。 • key 是条件表达式，一般是取服务的返回值或抛出的异常进行判断，默认是 SpringEL 表达式，判断服务返回参数，带 `$Exception{` 开头表示判断异常类型。 • value 是当这个条件表达式成立时则将服务执行状态映射成这个值。
`Catch`	捕获到异常后的路由。
`Retry`	捕获异常后的重试策略，是个数组，可以配置多个规则。 • `Exceptions` 为匹配的的异常列表。 • `IntervalSeconds` 为重试间隔。 • `MaxAttempts` 为最大重试次数。 • `BackoffRate` 为下一次重试间隔相对于上一次重试间隔的倍数。例如，如果上次一重试间隔是 2 秒，而 `BackoffRate=1.5`，则下一次重试间隔是 3 秒。 • `Exceptions` 属性可以不配置。不配置时，表示框架自动匹配网络超时异常。当在重试过程中发生了别的异常，框架会重新匹配规则，并按新规则进行重试，同一种规则的总重试次数不会超过该规则的 MaxAttempts。
`Next`	服务执行完成后下一个执行的状态。

当没有配置 Status 对服务执行状态进行映射，系统会自动判断状态：

- 没有异常，则认为执行成功。
- 如果有异常，则判断异常是不是网络连接超时，如果是则认为是 `FA`。
- 如果是其它异常，且服务 `IsForUpdate=true`，则状态为 `UN`，否则为 `FA`。

整个状态机的执行状态是由框架自己判断的，状态机有两个状态：status（正向执行状态）与 compensateStatus（补偿状态）。

- 如果所有服务执行成功（事务提交成功），则 status=SU，compensateStatus=null。
- 如果有服务执行失败，且存在更新类服务执行成功，且没有进行补偿（事务提交失败），则 status=UN，compensateStatus=null。
- 如果有服务执行失败，且不存在更新类服务执行成功，且没有进行补偿（事务提交失败），则 status=FA，compensateStatus=null。

- 如果补偿成功（事务回滚成功），则 status=FA/UN，compensateStatus=SU。
- 发生补偿且有未补偿成功的服务（回滚失败），则 status=FA/UN，compensateStatus=UN。
- 存在事务提交或回滚失败的情况，Seata Sever 都会不断发起重试。

Choice

```
"ChoiceState":{
        "Type":"Choice",
        "Choices":[
            {
            "Expression":"[reduceInventoryResult] == true",
            "Next":"ReduceBalance"
            }
        ],
        "Default":"Fail"
}
```

Choice 类型的状态是单项选择路由。

- `Choices`：可选的分支列表，只会选择第一个条件成立的分支。
- `Expression`：SpringEL 表达式。
- `Next`：当 Expression 表达式成立时，执行的下一个状态。

Succeed

```
"Succeed":{
    "Type":"Succeed"
}
```

运行到 **Succeed 状态**，表示状态机正常结束。正常结束不代表成功结束，是否成功要看每个状态是否都成功。

Fail

```
"Fail":{
    "Type":"Fail",
    "ErrorCode":"PURCHASE_FAILED",
    "Message":"purchase failed"
}
```

运行到 **Fail 状态**，表示状态机异常结束。异常结束时，可以配置 ErrorCode 和 Message，表示错误码和错误信息，可以用于给调用方返回错误码和消息。

CompensationTrigger

```
"CompensationTrigger":{
    "Type":"CompensationTrigger",
    "Next":"Fail"
}
```

`CompensationTrigger` 类型的 state 是用于触发补偿事件，回滚分布式事务。`Next` 表示补偿成功后路由到的 state。

SubStateMachine

```
"CallSubStateMachine":{
        "Type":"SubStateMachine",
        "StateMachineName":"simpleCompensationStateMachine",
        "CompensateState":"CompensateSubMachine",
        "Input":[
            {
            "a":"$.1",
            "barThrowException":"$.[barThrowException]",
            "fooThrowException":"$.[fooThrowException]",
            "compensateFooThrowException":"$.[compensateFooThrowException]"
            }
        ],
        "Output":{
             "fooResult":"$.#root"
        },
        "Next":"Succeed"
}
```

`SubStateMachine` 类型的“状态”是调用子状态机。

- `StateMachineName`：要调用的子状态机名称。
- `CompensateState`：子状态机的补偿 state，可以不配置，系统会自动创建它的补偿 state。子状态机的补偿实际就是调用子状态机的 compensate 方法。所以用户并不需要自己实现一个对子状态机的补偿服务。当配置这个属性时，可以利用 Input 属性自定义传入一些变量，参见下面的 `CompensateSubMachine`。

CompensateSubMachine

```
"CompensateSubMachine":{
      "Type":"CompensateSubMachine",
      "Input":[
           {
                 "compensateFooThrowException":"$.[compensateFooThrowException]"
           }
       ]
}
```

`CompensateSubMachine` 类型的 state 是专门用于补偿一个子状态机的 state，它会调用子状态机的 compensate 方法，可以利用 Input 属性传入一些自定义的变量，Status 属性自定判断补偿是否成功。

复杂参数的 Input 定义

```
"FirstState":{
        "Type":"ServiceTask",
        "ServiceName":"demoService",
        "ServiceMethod":"complexParameterMethod",
        "Next":"ChoiceState",
        "ParameterTypes":["java.lang.String","int","io.seata.saga.engine.mock.DemoService$P
eople","[Lio.seata.saga.engine.mock.DemoService$People;","java.util.List","java.util.Map"],
```

```
"Input":[
        "$.[people].name",
        "$.[people].age",
         {
            "name":"$.[people].name",
            "age":"$.[people].age",
            "childrenArray":[
                {
                "name":"$.[people].name",
                "age":"$.[people].age"
                },
                {
                "name":"$.[people].name",
                "age":"$.[people].age"
                }
            ],
            "childrenList":[
                {
                "name":"$.[people].name",
                "age":"$.[people].age"
                },
                {
                "name":"$.[people].name",
                "age":"$.[people].age"
                }
            ],
            "childrenMap":{
                "lilei":{
                "name":"$.[people].name",
                "age":"$.[people].age"
                }
            }
         },
        [
            {
            "name":"$.[people].name",
            "age":"$.[people].age"
            },
            {
            "name":"$.[people].name",
            "age":"$.[people].age"
            }
        ],
        [
            {
            "@type":"io.seata.saga.engine.mock.DemoService$People",
            "name":"$.[people].name",
            "age":"$.[people].age"
            }
        ],
        {
        "lilei":{
        "@type":"io.seata.saga.engine.mock.DemoService$People",
        "name":"$.[people].name",
```

```
                "age":"$.[people].age"
                }
            }
        ],
        "Output":{
             "complexParameterMethodResult":"$.#root"
        }
}
```

上述的 `complexParameterMethod` 方法定义如下：

```
People complexParameterMethod(String name, int age,People people,People[] peopleArrya,List<
People> peopleList,Map<String,People> peopleMap)
class People{
    private String name;
    private int    age;
    private People[] childrenArray;
    private List<People> childrenList;
    private Map<String,People> childrenMap;
...
}
```

启动状态机时，需传入参数：

```
Map<String,Object> paramMap =new HashMap<>(1);
People people =new People();
people.setName("lilei");
people.setAge(18);
paramMap.put("people", people);
String stateMachineName ="simpleStateMachineWithComplexParams";
StateMachineInstance inst = stateMachineEngine.start(stateMachineName,null, paramMap);
```

说明

`ParameterTypes` 属性是可以不用传的，调用的方法的参数列表中有 Map、List 这种可以带泛型的集合类型。因为 Java 编译会丢失泛型，所以需要使用这个属性，同时在 Input 的 JSON 中对应的对这个 JSON 加 `@type` 来申明泛型（集合的元素类型）。

3.8. 日志说明

3.8.1. dtx-client-digest.log

DTX 客户端调用日志信息。

路径： `/home/admin/logs/tracelog/dtx-client-digest.log` 。

日志格式： `[时间],[本地应用名称],[本地应用IP],[远程应用名称],[远程应用IP],[tracerId],[rpcId],[服务名称],[方法名称],[执行耗时],[线程名称],[执行结果],[tracer系统穿透数据],[tracer穿透数据]` 。

日志示例：

```
2019-11-27 10:05:50.820,dtx-fmt-action,13.13.0.46,dtx,100.98.114.188,0d0d002e15748203499461
00164500,0.2.1,FirstAction,prepare,113ms,SOFA-SEV-BOLT-BIZ-12200-3-T1,true,,dtx_id=a52761d2
8cf84dcabad3b7d9ff1d6c4701-0000000000& 2019-11-27 10:05:51.093,dtx-fmt-action,13.13.0.46,dt
x,10.142.69.10,0d0d002e157482034994610016450 0,0.2.2,SecondAction,prepare,27ms,SOFA-SEV-BOLT
-BIZ-12200-3-T1,true,,dtx_id=a52761d28cf84dcabad3b7d9ff1d6c4701-0000000000&
```

3.8.2. dtx-server-digest.log

DTX 服务端调用日志信息。

路径： `/home/admin/logs/tracelog/dtx-server-digest.log` 。

日志格式： `[时间],[本地应用名称],[远程应用名称],[tracerId],[rpcId],[资源ID],[方法名称],[远程服务端IP],[执行耗时],[线程名称],[执行结果],[tracer系统穿透数据],[tracer穿透数据]` 。

日志示例：

```
2019-11-27 10:05:51.312,dtx-fmt-action,dtx,0d0d002e157482034994610016450 0,0.6e08d272,Second
Action,commit,10.142.168.211:14200,98ms,dtx-bolt-pool-1,true,, 2019-11-27 10:05:51.427,dtx-
fmt-action,dtx,0d0d002e157482034994610016450 0,0.6e08d272,FirstAction,commit,10.142.168.211:
14200,94ms,dtx-bolt-pool-2,true,,
```

3.8.3. dtx-remote.log

主要记录 Client 端与 Server 端的 RPC 请求日志，包括 Client 端与 Server 端建立 TCP 连接，Client 端注册信息、鉴权信息、上报事务日志信息等。

路径： `/home/admin/logs/dtxserver/dtx-remote.log` 。

记录主事务上传事务开始日志

日志格式： `start BusinessActivity, activityRequest:” + ActivityRequest` 。

示例：

```
start BusinessActivity, activityRequest:
{
//应用名称
"appName": "bff-xxxx",
//业务ID
"bizId": "221869310xxxx",
//发起方标识
"bizType": "transfer_by_tcc",
//客户端IP
"clientIp": "10.10.0.xxx:50034",
//事务上下文信息
"context": "{\"start-time\":1619439759205,\"bizType\":\"transfer_by_tcc\",\"tracerInfo\":\
"tcid=0a0a00bd16194397591765601e8166&spid=0.1.4&pspid=0.1&sample=true&\",\"dataCenter\":\"c
n-xxxx-finance-h\",\"zone\":\"RZ00A\",\"appName\":\"bff-account\",\"bizId\":\"221869310\",
\"rpcId\":\"0.1.4\",\"isLdcMode\":true,\"host-name\":\"10.10.0.xxx\",\"shardingKey\":\"22
\",\"tracerId\":\"0a0a00bd16194397591765601e8166\"}",
//业务实例ID
"instanceId": "EKHAK9xxxxx",
//事务状态
"state": "U",
//超时时间
"timeout": 30,
//事务ID
"txId": "370c6843c30740249302.22-0000000000-221869310"
}
```

记录创建主事务失败日志

日志格式： `create new activity record failed, txId:" + activityId.getTxId() + ", instanceId" + instanceId + ", appName:" + appName` 。

示例：

```
create new activity record failed, txId:370c6843c30740249302.22-0000000000-221869310, insta
nceId EKHAK9xxxxx , appName:bff-xxxx
```

记录分支事务上传事务开始日志

日志格式： `start businessAction, actionRequest:" + ActionRequest` 。

示例：

```
start businessAction, actionRequest:
{
//分支事务ID
"actionId": "7898f69496424911be7309d7f9f134de",
//分支事务上下文信息
"context": "{\"sys::rollback_method\":\"dtxRollbackOne\",\"bizType\":\"ROLLBACK_FAILED\",\
"tracerInfo\":\"tcid=ac191f1416195934028961017229 48&spid=0&pspid=&sample=true&\",\"dataCent
er\":\"\",\"appName\":\"ant-cloud-quality-dtxtest\",\"sys::commit_method\":\"dtxCommitOne\
",\"isLdcMode\":false,\"host-name\":\"172.25.31.20\",\"action-start-time\":1619593403024,\
"sys::prepare\":\"dtxPrepare\",\"zone\":\"\",\"rpcId\":\"0\",\"sys::class_name\":\"com.al
ipay.antcloud.service.DtxOneAction\",\"actionName\":\"dtxOneAction\",\"tracerId\":\"ac191f
1416195934028961017229 48\"}",
//实例ID
"instanceId": "NCCI0DCxxxx",
//资源ID
"resourceId": "SOFABOOT_RPC&dtxOneAction&&",
//资源类型
"resourceType": 2,
//分支事务状态
"state": "I",
//事务ID
"txId": "12f02cc7de84442ca402-0000000000"
}
```

记录创建分支事务失败日志

日志格式： `create new action record failed, actionRequest:" + ActionRequest` 。

示例：

```
create new action record failed, actionRequest:
{
//分支事务ID
"actionId": "7898f69496424911be7309d7f9f134de",
//分支事务上下文信息
"context": "{\"sys::rollback_method\":\"dtxRollbackOne\",\"bizType\":\"ROLLBACK_FAILED\",\
"tracerInfo\":\"tcid=ac191f1416195934028961017229 48&spid=0&pspid=&sample=true&\",\"dataCent
er\":\"\",\"appName\":\"ant-cloud-quality-dtxtest\",\"sys::commit_method\":\"dtxCommitOne\
",\"isLdcMode\":false,\"host-name\":\"172.25.31.20\",\"action-start-time\":1619593403024,\
"sys::prepare\":\"dtxPrepare\",\"zone\":\"\",\"rpcId\":\"0\",\"sys::class_name\":\"com.al
ipay.antcloud.service.DtxOneAction\",\"actionName\":\"dtxOneAction\",\"tracerId\":\"ac191f
1416195934028961017229 48\"}",
//实例ID
"instanceId": "NCCI0DCxxxx",
//资源ID
"resourceId": "SOFABOOT_RPC&dtxOneAction&&",
//资源类型
"resourceType": 2,
//分支事务状态
"state": "I",
//事务ID
"txId": "12f02cc7de84442ca402-0000000000"
}
```

记录二阶段执行完成以后，更新主事务状态成功日志

日志格式： `update activity record state to [ " + (isCommit?"commit":"rollback") + " ] success, txId:" + txId + ", instanceId:" + instanceId` 。

示例：

```
update activity record state to [ commit ] success, txId:12f02cc7de84442ca402-0000000000, i
nstanceId:NCCI0DCxxxx
```

记录二阶段执行完成以后，更新主事务状态失败日志

日志格式： `update activity record failed, txId:" + txId + ", instanceId:" + instanceId, e` 。

示例：

```
update activity record failed, txId:12f02cc7de84442ca402-0000000000, instanceId: NCCI0DCxxx
x
com.alipay.dtx.server.common.exception.DtxException: null
at com.alipay.dtx.server.remote.manager.impl.DtxClientResourceManagerImpl.doAccessPermissio
n(DtxClientResourceManagerImpl.java:207)
at com.alipay.dtx.server.remote.manager.impl.DtxClientResourceManagerImpl.addConnection(Dtx
ClientResourceManagerImpl.java:119)
.......
```

记录二阶段执行状态日志

日志格式： `transaction handle finish, txId:" + txId + ", tenant:" + instanceId + ", result:" + (isDone ? "success" : "failed")` 。

示例：

```
transaction handle finish, txId:12f02cc7de84442ca402-0000000000, tenant:NCCI0DCxxxx, result
:success
```

3.8.4. dtx-recovery.log

主要记录事务恢复相关日志。

路径： `/home/admin/logs/dtxserver/dtx-recovery.log` 。

记录需要执行恢复任务的事务数日志

日志格式： `the num of record will be recoveried:"+ (businessActivitys != null ? businessActivitys.size() : 0)+ ", taskId:" + taskId` 。

示例：

```
the num of record will be recoveried:10 , taskId: default#prod#default#dtx_remote#business_
activity#business_action
```

记录单条事务恢复记录日志

日志格式： `recovery task of single record, taskId:" + taskId+ ", businessActivity:" + businessActivity` 。

示例：

```
recovery task of single record,
taskId:default#prod#default#dtx_remote#business_activity#business_action, businessActivity:
{
//应用名称
"appName": "bff-xxxx",
//业务ID
"bizId": "221869310xxxx",
//发起方标识
"bizType": "transfer_by_tcc",
//客户端IP
"clientIp": "10.10.0.xxx:50034",
//事务上下文信息
"context": "{\"start-time\":1619439759205,\"bizType\":\"transfer_by_tcc\",\"tracerInfo\":\
"tcid=0a0a00bd16194397591765601e8166&spid=0.1.4&pspid=0.1&sample=true&\",\"dataCenter\":\"c
n-xxxx-finance-h\",\"zone\":\"RZ00A\",\"appName\":\"bff-account\",\"bizId\":\"221869310\",
\"rpcId\":\"0.1.4\",\"isLdcMode\":true,\"host-name\":\"10.10.0.xxx\",\"shardingKey\":\"22
\",\"tracerId\":\"0a0a00bd16194397591765601e8166\"}",
//业务实例ID
"instanceId": "EKHAK9xxxxx",
//事务状态
"state": "U",
//超时时间
"timeout": 30,
//事务ID
"txId": "370c6843c30740249302.22-0000000000-221869310"
}
```

记录事务状态不正确时，恢复任务未执行日志

日志格式： `unknown transaction state, txId:" + txId + ", state:"+ businessActivity.getState()` 。

示例：

```
unknown transaction state, txId:370c6843c30740249302.22-0000000000-221869310, state:U
```

记录事务恢复任务是否成功日志

日志格式： `recovery task of single record finish, txId:" + txId + ",taskId:"+ taskId + " ,result:" + (isDone ? "success" : "failed") + " .cost time:" + (System.currentTimeMillis() - startTime) + "ms"` 。

示例：

```
recovery task of single record finish, txId:2ff1f588963d43d3bd02.22-0000000000-221973521,ta
skId:default#prod#default#dtx_remote#business_activity#business_action ,result:failed .cost
time:2ms
```

4.进阶指南

4.1. TCC 防悬挂与幂等接入

TCC 防悬挂的目的是为了防止空回滚，即二阶段回滚方法比一阶段 try 方法先执行。TCC 幂等控制的目的是保证同一个分布式事务内，TCC 参与者的一阶段 try 方法会且只会被执行一次。

您可以根据以下步骤将 TCC 防悬挂与幂等配置在 TCC 参与者服务发布端的应用中：

1. 创建表
2. 配置 DAO
3. 修改定义接口
4. 修改实现接口

创建表

执行以下 DDL 语句，在业务数据库中创建表 `dtx_tcc_action`：

```
CREATE TABLE `dtx_tcc_action`(
`action_id` varchar(96) NOT NULL COMMENT '分支事务号',
`action_name` varchar(64) DEFAULT NULL COMMENT '参与者名称',
`tx_id` varchar(128) NOT NULL COMMENT '主事务号',
`action_group` varchar(32) DEFAULT NULL COMMENT 'action group',
`status` varchar(10) DEFAULT NULL COMMENT '状态',
`param_data` varchar(4000) DEFAULT NULL COMMENT '一阶段方法参数数据',
`gmt_create` datetime NOT NULL COMMENT '创建时间',
`gmt_modified` datetime NOT NULL COMMENT '修改时间',
`sharding_key` varchar(128) DEFAULT NULL COMMENT '分库分表字段',
  PRIMARY KEY (`action_id`),
  UNIQUE KEY  `idx_tx_id`(`tx_id`,`action_name`)
);
```

配置 DAO

- 如果用户使用的是 Ibastis，需要配置如下 bean：

```
<bean class="com.alipay.sofa.dtx.tcc.dao.ibatis.IbatisTccAntiSuspendDAO">
     <property name="dataSource" ref="yourDataSourceBean"/>
</bean>
```

- 如果用户使用的是 Mybatis，配置如下 bean：

```
<bean class="com.alipay.sofa.dtx.tcc.dao.mybatis.MybatisTccAntiSuspendDAO">
     <property name="dataSource" ref="yourDataSourceBean"/>
</bean>
```

> **重要**
>
> 其中的 dataSource 属性需配置成业务数据库的 DataSource bean。

修改定义接口

将 @TwoPhaseBusinessAction 注解的 `antiSuspend` 属性配置成 true，打开防悬挂功能：

```
public interface TccAction{
@TwoPhaseBusinessAction(name ="yourTccActionName", commitMethod ="confirm", rollbackMethod
="cancel", antiSuspend =true)
public boolean try(BusinessActionContext businessActionContext,@ShardingKeyint a,int b);
public boolean confirm(BusinessActionContext businessActionContext);
public boolean cancel(BusinessActionContext businessActionContext);
}
```

重要

如果业务数据库分库分表，则需要在一阶段方法的分库分表参数前添加 @ShardingKey 注解，单库单表无需添加。

修改实现接口

在 TCC 参与者一阶段 try 方法中，业务的数据库事务内，调用 `TccTransactionController.doAntiSuspendControl()` 方法添加防悬挂记录：

```
public class TccActionImpl implements TccAction {
    @Override
    public boolean try(
    BusinessActionContext businessActionContext, int a, int b)

    {

        transactionTemplate.execute(newTransactionCallback < Object > () {
        @Override
        public Object doInTransaction (TransactionStatus status){

            //手动防悬挂
            TccTransactionController.doAntiSuspendControl();

        }
    });

    }

    @Override
    public boolean confirm(BusinessActionContext businessActionContext) {

    }

    @Override
    public boolean cancel(BusinessActionContext businessActionContext) {

    }
}
```

4.2. FMT 模式行锁策略

以下配置需配置在工程属性配置文件中，该文件的位置根据框架不同而有所不同：

- SOFABoot 工程属性文件：该文件通常位于 `src\main\resources\config\application.properties` 。
- 纯 Spring 工程属性文件：该文件通常在工程的 `META-INF` 目录下。

乐观锁

分布式事务的 FMT 模式行锁默认是悲观锁。如果对数据一致性要求不高，可以开启乐观锁。

您可在 SOFABoot 的配置文件 `config/application.properties` 中添加如下乐观锁配置：

是否开启乐观锁

```
com.alipay.dtx.rowlock.optimistic=xxx;
```

其中的 `xxx` 为 true 或 false，默认值为 false。

自旋锁

自旋锁即获取行锁失败之后重试，默认情况下会重试 3 次，每次重试等待 3 ms，您可以通过如下属性自定义重试策略：

- **是否开启自旋锁**

  ```
  com.alipay.dtx.rowlock.spin=xxx;
  ```

  其中的 `xxx` 为 true 或 false，默认值为 true。

- **自旋锁等待时间**

  ```
  com.alipay.dtx.rowlock.spin.delay=xxx;
  ```

  其中的 `xxx` 为等待时间，单位为 ms，默认值为 3 ms。

- **自旋锁自旋次数**

  ```
  com.alipay.dtx.rowlock.spin.times=xxx;
  ```

  其中的 `xxx` 为自旋次数，默认值为 3 次。

4.3. FMT 模式 SQL 支持说明

SQL 语句的支持范围还在不断扩大，本文将介绍目前 SQL 语句的支持情况。

- FMT 模式支持 SELECT、INSERT、UPDATE、DELETE 四类 DML 语法的部分功能。
- FMT 模式不支持存储过程、触发器。

SQL 语句的支持范围还在不断扩大，目前 SQL 语句的支持情况如下：

> **说明**
>
> 如果需要使用其它 SQL 语句，请在控制台 提交工单 咨询。

基本 SQL 语句

类型	SQL 实例	是否支持
INSERT	INSERT INTO tb1 (col_name,...) VALUES ({expr,value},...)	是
INSERT INGORE	INSERT INGORE INTO tb1 (col_name,...) VALUES ({expr,value}, ...)	是
INSERT ... ON DUPLICATE KEY UPDATE	INSERT INTO tb1 (col_name,...) VALUES ({expr,value},...) ON DUPLICATE KEY UPDATE SET col1=expr1 [, col2=expr2 ...] [WHERE where_definition]	是
UPDATE	UPDATE tb1 SET col1=expr1 [, col2=expr2 ...] [WHERE where_definition]	是
DELETE	DELETE FROM tb1 [WHERE where_definition]	是
SELECT	SELECT select_expr, ... FROM tb1 [WHERE where_condition] [order by condition] [group by condition [having condition]] [for update] [limit statement]	是

更新 SQL 语句

类型	SQL 实例	是否支持
AND & OR	UPDATE ... WHERE col1 = expr1 AND col2 = expr2	是
LIKE	UPDATE ... WHERE col1 LIKE 'X%'	是
BETWEEN	UPDATE ... WHERE col1 BETWEEN expr1 AND expr2	是
IN	DELETE ... WHERE col1 in (expr1 ,..., exprN)	是
IS NULL	DELETE ... WHERE col1 IS NULL	是
嵌套语句	UPDATE tb1 ... WHERE col1 in (select col2 from tb2)	否
存储过程、触发器		否

SELECT 语句

类型	SQL 实例	读未提交	读已提交
AND & OR	SELECT * FROM tb1 WHERE col1 = expr1 AND col2= expr2	是	是
ORDER BY	SELECT col1, col2 FROM tb1 ORDER BY col1	是	是
GROUP BY	SELECT col1, col2 FROM tb1 GROUP BY col1 having col1 > 1	是	是
LIKE	SELECT col1, col2 FROM tb1 WHERE col1 LIKE 'X%'	是	是
IN	SELECT col1, col2 FROM tb1 WHERE col1 IN (expr1, expr2,...)	是	是
BETWEEN	SELECT col1, col2 FROM tb1 WHERE col1 BETWEEN expr1 AND expr2	是	是
LIMIT	SELECT col1, col2 FROM tb1 LIMIT x	是	是
JOIN、INNER JOIN、LEFT JOIN、RIGHT JOIN、FULL JOIN	SELECT col1, col2 FROM tb1 JOIN tb2 ON tb1.col1= tb2.col2	是	否
UNION、UNION ALL	SELECT col1, col2 FROM tb1 UNION SELECT col1, col2 FROM tb2	是	否
嵌套查询	SELECT col1, col2 FROM tb1 where col1 in (select col3 from tb2)	是	否
存储过程、触发器		否	否

4.4. 接入单元化能力

本文将引导您对本地分布式事务客户端进行改造，接入单元化能力。

> **说明**
>
> 该功能仅适用于支持 LDC 单元化架构的环境。

DTX 服务配置

1. 添加 Maven 依赖：

```
<dependency>
      <groupId>com.alipay.sofa</groupId>
      <artifactId>dtx-enterprise-sofa-boot-starter</artifactId>
      <version>${dtx.version}</version>
</dependency>
```

2. 服务配置：

```
<bean id="dtxService" class="com.alipay.dtx.client.core.api.impl.DtxServiceImpl"/>
```

发起方配置

发起方应用配置 spring 事务模板，配置示例如下：

```
<!--声明新的事务模板-->
<bean id="transactionTemplate"class="org.springframework.transaction.support.TransactionTem
plate">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="propagationBehaviorName">
               <value>PROPAGATION_REQUIRES_NEW</value>
        </property>
</bean>
```

在发起方的 spring 事务模板内，调用 `dtxService.start(bizType, bizId, userId, context)` 方法开启分布式事务（不再使用前文中 `[@DtxTransaction](#)` 注解的方式开启事务）；其中参数 `userId` 会作为单元化架构下的路由参数，用作服务路由和数据库路由。

```
public class YourClass {

    public void yourMethod(yourParams) {

        transactionTemplate.execute(newTransactionCallback() {
            @Override
            public Object doInTransaction (TransactionStatus status){

                //开启分布式事务
                dtxService.start(bizType, bizId, userId, context)

                try {
                    // 调用 TCC1 参与者一阶段try方法，第一个参数是用户ID，try方法BusinessActionC
ontext参数值传'null';
                    TCC1. try (userId,params1,null) ;

                    // 调用 TCC2 参与者一阶段try方法，第一个参数是用户ID，try方法BusinessActionCo
ntext参数值传'null';
                    TCC2. try (userId,params2,null) ;

                    // 方法正常返回，事务提交
                } catch (Throwable t) {
                    // 出现异常，事务回滚
                    throw t;
                }
            }
        });
    }
}
```

发起方实现回查服务

当事务状态不确定时，dtxserver 需要通过回查服务询问发起方分布式事务状态，发起方需要实现 `com.alipay.dtx.client.core.api.ActivityStateResolver` 接口，在 `isDone` 方法中返回事务的状态（0 即未知，1 即提交，2 即回滚）；

```
public class ActivityStateResolverImpl implements com.alipay.dtx.client.core.api.ActivitySt
ateResolver {

    @Override
    public int isDone(String businessType, String businessId) {
        System.out.println("回查, businessType:" + businessType + ", businessId" + businessI
d);
        return ActivityStateResolver.NOT_DONE | ActivityStateResolver.DONE;
    }
}
```

回查服务配置成 spring bean：

```
<bean id="activityStateResolver" class="com.xxx.ActivityStateResolverImpl"/>
```

5.最佳实践

5.1. Saga 模式服务设计

基于 Saga 模式分布式事务的多年实践，本文提供了在 Saga 模式下服务设计的一些最佳实践与经验。

- 服务执行与补偿
- 判断服务状态
- 应对隔离性问题

服务执行与补偿

Saga 模式是 SEATA 提供的长事务解决方案。Saga 模式下，分布式事务内存在多个参与者，每一个参与者都是一个冲正补偿服务，用户需要根据业务场景实现其正向操作（原服务）和逆向回滚操作（补偿服务）。在事务执行过程中，首先会依次执行各参与者的正向操作，如所有正向操作执行成功，则事务提交。如任一正向操作执行失败，则事务会执行之前各参与者的逆向回滚操作，回滚已提交的参与者，直至事务退回至其初始状态。

允许服务空补偿

空补偿，指的是原服务未执行，补偿服务已执行。大致场景如下：

针对该问题，在服务设计时，需要允许空补偿，即在没有找到要补偿的业务主键时，返回补偿成功，并将原业务主键记录下来，标记该业务流水已补偿成功。

服务防悬挂控制

悬挂，指的是补偿服务比原服务先执行。大致场景如下：

此时，需要检查当前业务主键是否已经在空补偿记录下来的业务主键中存在，如果存在则要拒绝执行该笔服务，以免造成数据不一致。

服务幂等控制

在分布式事务执行过程中，原服务与补偿服务都需要保证幂等性。由于网络连接可能超时，可以设置重试策略，重试发生时要通过幂等控制避免业务数据重复更新。

判断服务状态

Saga 发起方作为服务方，需要根据状态机的执行状态，返回服务状态给调用方。可以参考以下代码示例：

```
  StateMachineInstance inst = stateMachineEngine.startWithBusinessKey("testTransferBySaga",
null, businessKey, params);
    if(ExecutionStatus.SU.equals(inst.getStatus()) && inst.getCompensationStatus()==null){
        //正向状态为成功，补偿状态为空（没有触发回滚），则交易成功，返回 SU
    }else if(ExecutionStatus.SU.equals(inst.getCompensationStatus())){
        //补偿状态为成功，则交易回滚成功，返回失败 FA
        //如果有多级调用，可以给调用方返回失败 FA，这样上层编排则不再进行补偿该服务
    }else if(ExecutionStatus.FA.equals(inst.getStatus()) && inst.getCompensationStatus()==n
ull){
         //正向状态为失败，补偿状态为空（没为触发回滚），则交易失败且没有数据不一致，返回FA
    }else{
        //其它情况说明是正向或补偿结果未知，返回未知UN给调用方，需要进行重试
    }
```

应对隔离性问题

Saga 模式由于一阶段已经提交本地数据库事务，且没有进行预留动作，因此无法保证隔离性。在极端情况下，可能由于脏写（即当本地事务提交之后、分布式事务完成之前，该数据被补偿之前又被其他操作修改过。）无法完成回滚操作。例如，在分布式事务中，需要先给用户 A 充值，然后给用户 B 扣减余额。如果在给用户 A 充值成功，在事务提交以前，用户 B 已经把余额消费掉了，如果事务发生回滚，此时则没有办法进行补偿了。

以上是一个极端场景下隔离性缺乏造成的典型问题。在实践中，一般可以采取以下方法应对该问题：

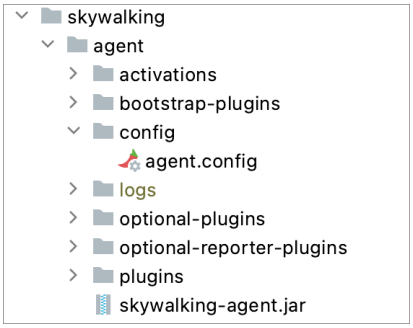
- 业务流程设计时，始终遵循“宁可长款，不可短款”的原则。长款指的是客户少了钱机构多了钱，以机构信誉可以给客户退款；反之则是短款，少的钱可能追不回来了。所以在业务流程设计上，一定是先扣款。
- 有些业务场景可以允许让业务最终成功，在回滚不了的情况下，可以继续重试完成后面的流程。所以状态机引擎除了提供“回滚”能力还需要提供“向前”恢复上下文继续执行的能力，让业务最终执行成功，达到最终一致性的目的。

5.2. Demo 应用接入 SkyWalking

SkyWalking 是一个开源的监控平台，用于从服务和云原生基础设施中收集、分析、聚合和可视化数据，是目前使用比较多的一款链路追踪产品。本文介绍如何在分布式事务中集成 SkyWalking。

依赖与配置项

1. 在工程根目录中添加 SkyWalking Agent 文件。



2. 修改 Dockerfile及 start.sh。

```
COPY skywalking /home/admin/skywalking
```

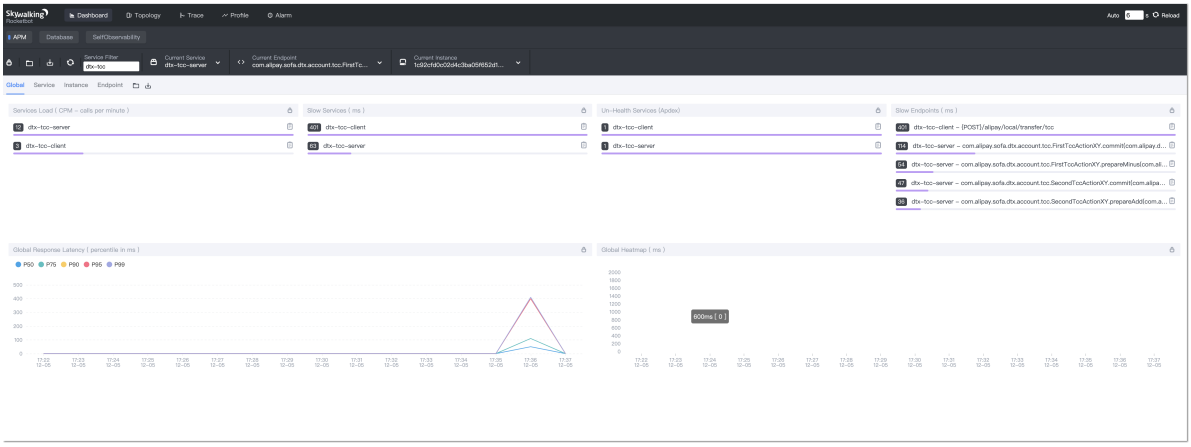
```
 su admin -c "、opt/sofaware/java/bin/java ${JAVA_OPTS} ${SKYWALKING_AGENT} ${PROFILE} -
Dcom.alipay.ldc.region=${CAFE_ENV_REGION_NAME} -Dcom,alipay.ldc.datacenter=${}"
```

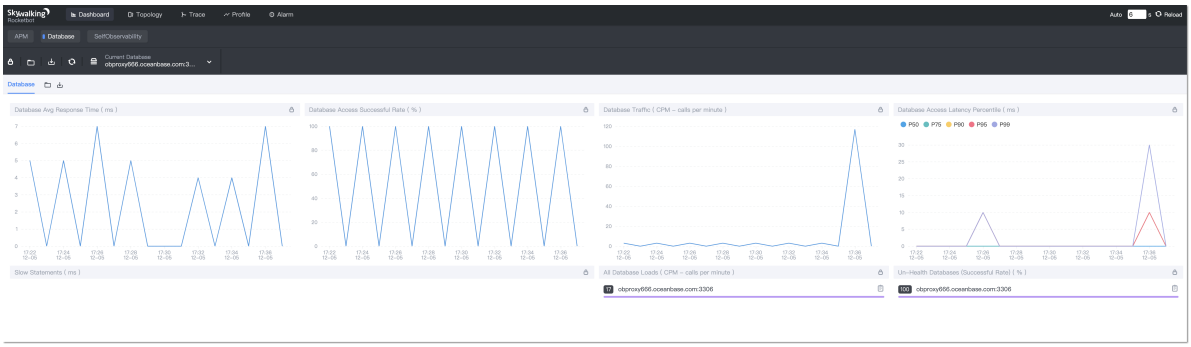
3. 在启动项 Yaml 中添加环境变量：

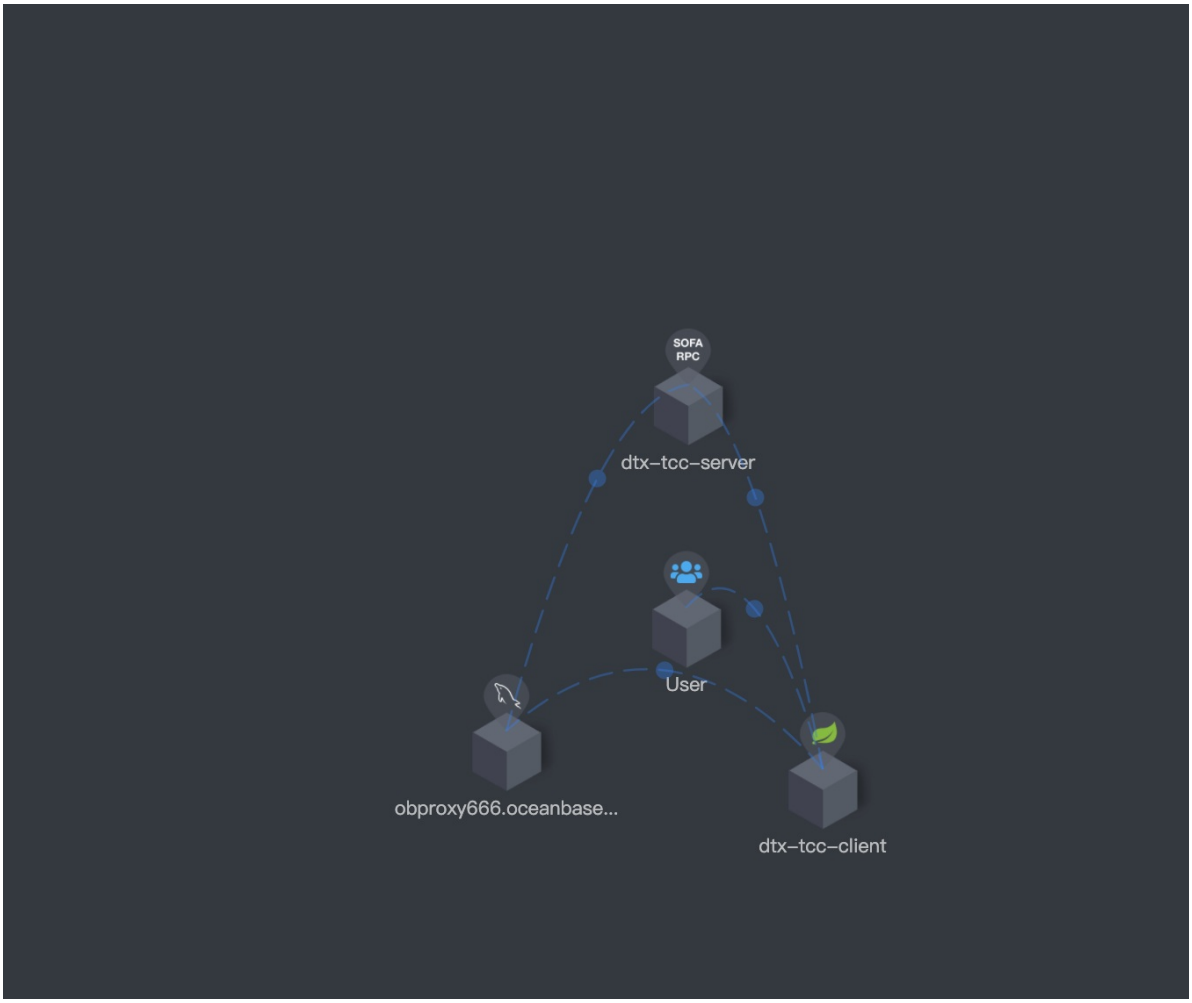
```
- name: SKYWALKING_AGENT
  value: "-javaagent:/home/admin/skywalking/agent/skywalking-agent.jar"
- name: SW_AGENT_COLLECTOR_BACKEND_SERVICES
  value: <100.XX.XXX.199:11800>
- name: SW_AGENT_NAME
  value: "dtx-tcc-client"
```

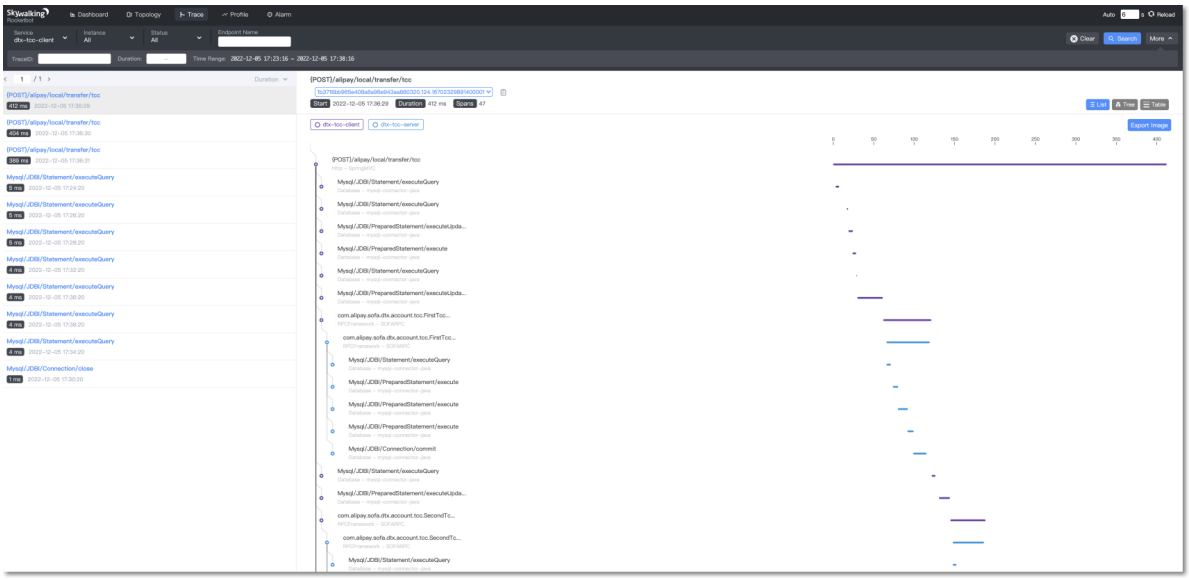
4. 启动业务应用。

   应用启动后在 SkyWalking 页面即可查询到相关信息。









Demo 应用日志集成 Skywalking Trace

1. 在 `pom.xml` 文件中依赖。

```
<dependency>
      <groupId>org.apache.skywalking</groupId>
      <artifactId>apm-toolkit-logback-1.x</artifactId>
      <version>8.1.0</version>
</dependency>
```

2. 修改 logback 配置文件。

```
<appender name="ROOT-APPENDER" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <append>true</append>
    <filter class="ch.qos.logback.classic.filter.LevelFilter">
        <level>${logging.level}</level>
        <onMatch>ACCEPT</onMatch>
        <onMismatch>DENY</onMismatch>
    </filter>
    <file>${logging.path}account-demo/common-default.log</file>
    <!-- to generate a log file everyday with a longest losting of 30 days -->
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!-- logfile name with daily rolling-->
        <FileNamePattern>
            ${logging.path}/account-demo/common-default.log.%d{yyyy-MM-dd}
        </FileNamePattern>
        <!-- log perserve days-->
        <MaxHistory>30</MaxHistory>
     </rollingPolicy>
     <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
        <layout class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.mdc.TraceIdPatt
ernLogbackLayout">
            <!-- ouput format:%d is for date,%thread is for thread name,%-5level:loglevel
with 5 character %,ag:log message,% -->
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%X{tid}] [%thread] %-5level %logger{50
} - %msg%n</pattern>
            <!-- encoding -->
        </layout>
    <charset>UTF-8</CHARSET>
    </encoder>
</appender>
```

3. 发起事务请求，Demo 应用业务日志中即可管理 Skywalking 的 TraceID 和 DTX 的事务 txid。

```
2022-12-05 08:58:19.792 [TID:1b3718bb965e408a8a96e943aa888320.118.16782306997730001] [http-nio-9999-exec-1] INFO  o.a.c.core.ContainerBase.[Tomcat].[localhost].[/] - Initializing Spring DispatcherServlet 'dispatcherServlet'
2022-12-05 08:58:19.793 [TID:1b3718bb965e408a8a96e943aa888320.118.16782306997730001] [http-nio-9999-exec-1] INFO  org.springframework.web.servlet.DispatcherServlet - Initializing Servlet 'dispatcherServlet'
2022-12-05 08:58:19.808 [TID:1b3718bb965e408a8a96e943aa888320.118.16782306997730001] [http-nio-9999-exec-1] INFO  org.springframework.web.servlet.DispatcherServlet - Completed initialization in 15 ms
2022-12-05 08:58:21.212 [TID:1b3718bb965e408a8a96e943aa888320.118.16782306997730001] [http-nio-9999-exec-1] INFO  c.a.s.dtx.account.service.impl.TransferServiceImpl - tcc starting, txId:53e092a5c51047 8fac03-0000000000-d347c9ed31ee43660b8d9ffd5aa945c6
2022-12-05 08:58:21.576 [TID:1b3718bb965e408a8a96e943aa888320.118.16782306997730001] [http-nio-9999-exec-1] INFO  c.a.s.r.s.router.rule.AppRouterRuleOrmResource - Subscriber service router rules from drm. service = com.alipay.sofa.dtx.account.tcc.FirstTccActionXY01.0#DEFAULT, traffic type = OUTBOUND
2022-12-05 08:58:21.622 [TID:1b3718bb965e408a8a96e943aa888320.118.16782306997730001] [http-nio-9999-exec-1] INFO  c.a.s.r.s.router.rule.AppRouterRuleOrmResource - Subscriber app router rules from drm. app = account-portaker, traffic type = OUTBOUND
2022-12-05 08:58:21.845 [TID:1b3718bb965e408a8a96e943aa888320.118.16782306997730001] [http-nio-9999-exec-1] INFO  c.a.s.r.s.router.rule.AppRouterRuleOrmResource - Subscriber service router rules from drm. service = com.alipay.sofa.dtx.account.tcc.SecondTccActionXY01.0#DEFAULT, traffic type = OUTBOUND
2022-12-05 08:58:21.896 [TID:1b3718bb965e408a8a96e943aa888320.118.16782306997730001] [http-nio-9999-exec-1] INFO  c.a.s.dtx.account.service.impl.TransferServiceImpl - tcc ending, txId:53e092a5c51047 8fac03-0000000000-d347c9ed31ee43660b8d9ffd5aa945c6
2022-12-05 08:58:26.364 [TID:1b3718bb965e408a8a96e943aa888320.120.16782307083420001] [http-nio-9999-exec-3] INFO  c.a.s.dtx.account.service.impl.TransferServiceImpl - tcc starting, txId:4356de34586e4114b081-0000000000-25cf22da2cbe485a8484c91766fc f03
2022-12-05 08:58:28.525 [TID:1b3718bb965e408a8a96e943aa888320.120.16782307083420001] [http-nio-9999-exec-3] INFO  c.a.s.dtx.account.service.impl.TransferServiceImpl - tcc ending, txId:4356de34586e4114b081-0000000000-25cf22da2cbe485a8484c91766fc f03
2022-12-05 08:58:30.346 [TID:1b3718bb965e408a8a96e943aa888320.122.16782307103220001] [http-nio-9999-exec-5] INFO  c.a.s.dtx.account.service.impl.TransferServiceImpl - tcc starting, txId:dbf5e318c04a42cd9e03-0000000000-8685236880b84acc933067418 0f6c4b
2022-12-05 08:58:30.508 [TID:1b3718bb965e408a8a96e943aa888320.122.16782307103220001] [http-nio-9999-exec-5] INFO  c.a.s.dtx.account.service.impl.TransferServiceImpl - tcc ending, txId:dbf5e318c04a42cd9e03-0000000000-8685236880b84acc933067418 0f6c4b
```

6.权限说明

> **说明**
>
> 需要升级到 v2.4.11 及以上版本。

分布式事务控制台使用中间件产品的用户角色定义，包括以下三种角色：

- 分布式事务-管理员
- 分布式事务-开发者
- 分布式事务-观察者

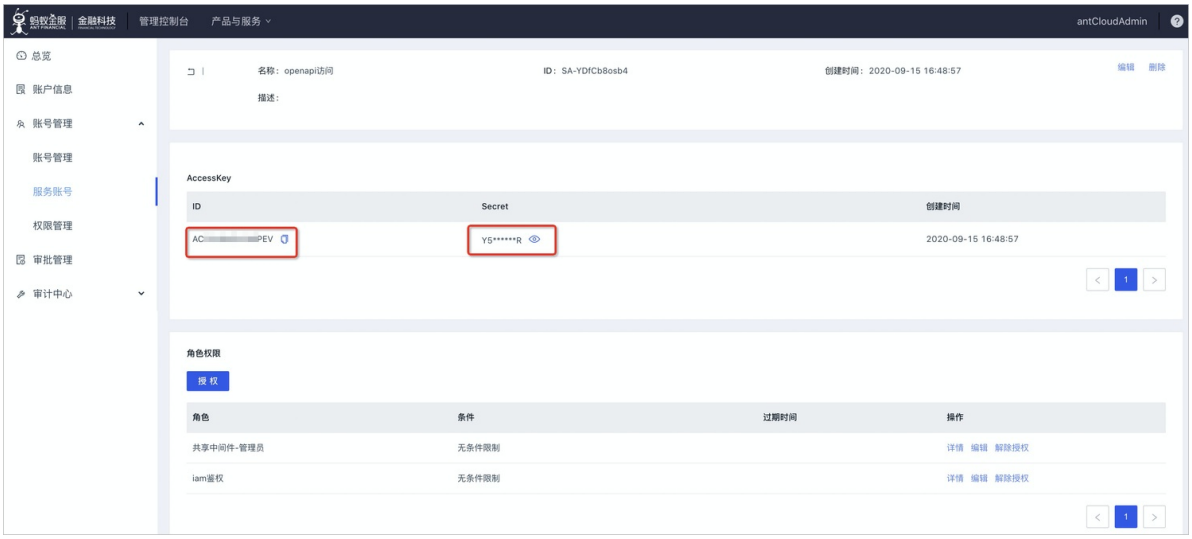
> **说明**
>
> 专有云环境若需开启鉴权，请联系技术支持工程师。开启后，您可前往 IAM 控制台获取 AK（AccessKey ID）和 SK（AccessKey Secret）信息。

获取 AK（AccessKey ID）和 SK（AccessKey Secret）

1. 生成服务账号。
   i. 登录 IAM 控制台。
   ii. 在左侧导航栏，选择 **账号管理** > **服务账号**。
   iii. 单击 **新增服务账号**，并在弹出的对话框中输入服务账号的名称和描述。
   iv. 单击 **OK**。
2. 为账号授予权限。
   i. 单击刚刚创建的账号右侧的 **授权**。
   ii. 按照步骤给账号授予合适的中间件权限。

   您可以给账号授予中间件开发者或管理员权限，或根据自己需要创建合适的权限授予该服务账号。
3. 获取 AK 和 SK。

   单击账号名称或右侧的 **详情**，进入服务账号详情页。在该页面的 AccessKey 区域查看 ID 和 Secret 信息。



操作权限说明

具体权限设置请参见下表。若要修改您的用户角色，请联系空间管理员处理。

模块	操作	分布式事务-管理员	分布式事务-开发者	分布式事务-观察者
工作台	查看统计数据	✓	✓	✓
	获取应用名列表	✓	✓	✓
	获取当前环境	✓	✓	✓
	创建/编辑应用事务	✓	✓	×
	导入应用配置	✓	✓	×
	导出应用配置	✓	✓	×
	单个发起方新增/编辑	✓	✓	×
	单个发起方删除	✓	✓	×
	批量新增 APP 下的发起方、参与者	✓	✓	×

应用事务	查询bizType的调用关系	✓	✓	✓
	获取bizType事务概览统计	✓	✓	✓
	删除应用	✓	✓	×
	查看应用配置列表	✓	✓	✓
	查看应用详情	✓	✓	✓
	导入事务配置	✓	✓	×
	导出事务配置	✓	✓	×
	删除事务	✓	✓	×
	查看应用事务列表	✓	✓	✓
	查看事务详情	✓	✓	✓
事务监控	事务高级查询	✓	✓	✓
	删除进行中的事务	✓	✓	×

7.客户端（DTX-Client）配置说明

配置项	描述	使用说明
com.alipay.instanceid	实例 ID，必填。对应具体的租户 + 环境，即每个租户每个环境对应一个 InstanceId。	dtx-demo 如果设置 InstanceId 为000001，那么以 000001 身份登入 DTX 控制台后可以看到 dtx-demo 的事务记录。
com.antcloud.antvip.endpoint	ACVIP 的域名，和“com.alipay.dtx.server.hosts”二选一。DTX Client 启动时会访问 ACVIP 获取 DTX Server 的地址。	如果 dtx-demo 设置 ACVIP 的地址是 10.51.211.190，那么 dtx-demo 启动后会从 10.51.211.190 获取 DTX Server 的一组 IP。之后，dtx-demo会依次连接各DTX Server。
com.alipay.dtx.model	DTX 使用模式，默认 remote。LOCAL 表示同库模式，remote 表示异库模式。	异库模式和同库模式的区分请参考 DTX 相关文档。
com.alipay.dtx.local.sync.record	同库模式时是否上报事务记录到 Server，默认为 true。	dtx-demo 如果是同库模式且开启上报事务记录到 Server，那么在 DTX 控制台可以看到事务日志和事务监控信息。
com.alipay.dtx.report.record.asyn	同库模式时是否异步方式上报事务记录到 Server，默认为 false。	dtx-demo 如果是同库模式且需要上报事务日志，建议将这个开关设置为 true，存在丢失事务日志的可能性。
com.alipay.dtx.recovery.expiration.minute	事务恢复超期时间（单位：分钟），默认为 7 天，-1 为不超期。	事务如果超出固定时间没有恢复成功将不会再执行恢复。那么没有得到恢复的事务需要业务手动处理。
com.alipay.dtx.server.hosts	DTX Server 的 IP，多个地址用“，”分割，和“com.antcloud.antvip.endpoint”二选一，但同时存在时优先级更高。	dtx-demo 如果设置了 DTX Server 的 IP，启动后将不访问 ACVIP，dtx-demo 直接连接设置好的 DTX Server IP。
com.alipay.dtx.local.deleteOnFinish	同库模式默认删除事务数据，默认是 true。	同库模式下，当事务结束后默认会删除业务侧（dtx-demo）的事务数据。
zmode	开启单元化。	单元化场景需要，开启之后 dtx-client 也就是 dtx-demo 会去获取单元信息。

com.alipay.ldc.datacenter	机房信息。	单元化场景需要，dtx-demo 所在的机房。
com.alipay.ldc.zone	单元信息。	单元化场景需要，dtx-demo 所在的单元。
com.alipay.ldc.region	域信息。	单元化场景需要，dtx-demo 所在的域。
logging.level.com.alipay.sofa.dtx	日志级别，默认 INFO。	日志级别。
com.alipay.dtx.server.request.timeout	请求超时时间，默认 3000ms。	dtx-demo 请求 DTX Server 的超时时间。异库模式比较关注。
com.alipay.dtx.server.tryNum	请求失败重试次数。	dtx-demo 请求 DTX Server 失败重试的次数。异库模式比较关注。
com.alipay.dtx.secondPhase.asyn	TCC 同库模式开启二阶段异步执行，默认为 false（不开启）。	TCC 同库模式建议开启，会提升 50% 的性能。
com.alipay.dtx.local.shard.store	TCC 同库模式支持分库分表路由开关，默认为 false（不开启）。	TCC 同库模式分库分表场景建议开启。
com.alipay.dtx.transaction.timeout	事务超时时间，默认 30s。	整个事务一阶段执行时间如果超过某个时间，将失败回滚。如果针对单个事务配置超时时间，可以在 `@DtxTransaction` 注解或者 `dtxservice.start()` 中方法指定。
con.alipay.dtx.framework.txId	是否将 txId 设置在 FrameWork 上下文中。	将 txId 放到 DTX 的上下文中，这样各个线程就能共享 txId 字段。
com.alipay.env	运行环境。默认为非金融云环境。当这个值为 shared 时，表示金融云环境。	金融云环境下可以依赖蚂蚁的中枢组件，比如 Antvip。如果环境里有 Antvip 组件，建议设为 shared。
com.alipay.dtx.client	客户端版本，默认为 DTX。	不用指定，使用默认值即可。
com.antcloud.mw.access	当 DTX Server 开启鉴权时（默认不开启），客户端需要设置 access 字段。	这个字段和 InstanceId 对应起来，在 IAM 的控制台的账号信息里获取。

com.antcloud.mw.secret	当 DTX Server 开启鉴权时默认不开启），客户端需要设置 secret 字段。	这个字段和 InstanceId 对应起来，在 IAM 的控制台的账号信息获取。

8.故障排查

8.1. 错误码

SDK 集成相关

错误码	错误消息	错误说明	解决方案
DTX-000	dtx component scanner init error.	SDK 启动失败，分布式事务不可用。	-
DTX-001	Bean[%s] can not have @DtxTransaction and @TccBusinessAction both.	同一个 bean 不能既有 @DtxTransaction 又有 @TccBusinessAction 注解。	修改代码，去掉 bean 的 @DtxTransaction 注解或者 @TccBusinessAction 注解 ，只保留其中一个。
DTX-002	the length of bizType[%s] can not be more than 30.	@DtxTransaction 注解上的 BizType 长度不能超过 30 个字符。	将 BizType 长度控制在 30 字符以内。
DTX-003	init dtx configuration error.	初始化分布式事务配置项出现异常。	-
DTX-003	Class [%s] can not have @TccBusinessAction and @TwoPhaseBusinessAction both.	同一个 bean 不能既有 @TccBusinessAction 又有 @TwoPhaseBusinessAction 注解。	修改代码，去掉 bean 的 @TccBusinessAction 注解或者 @TwoPhaseBusinessAction 注解，只保留其中一个。
DTX-005	Unknown remoting service bean type, dtx will ignore this bean	未知的远程服务类型，将忽略此 bean。	-
DTX-006	tcc 2.0 prepare error, txId:[%s].	TCC 2.0 一阶段方法执行失败。	排查业务一阶段方法执行失败原因。
DTX-007	tcc 2.0 prepare control, idempotency failed, txId:[%s].	TCC 2.0 一阶段方法幂等控制执行失败。	-

DTX-008	tcc 2.0 prepare method @ShardingKey param type must be String, className:[%s].	@ShardingKey 标记参数必须是 String 类型。	修改 @ShardingKey 标记参数的类型为 String。
DTX-009	tcc 2.0 the java.sql.DataSource bean [%s] named by @TccBusinessAction.dataSourceBeanId() on class [%s] is not exist in spring applicationContext.	@TccBusinessAction的 dataSourceBeanId 属性指定的 DataSource bean 不存在。	检查 @TccBusinessAction的 dataSourceBeanId 属性指定的 DataSource bean 是否存在。
DTX-010	tcc 2.0 the interface [%s] has no @TccBusinessAction method.	TCC 2.0 的参与者接口无 @TccBusinessAction 注解。	检查确保 TCC 2.0 的参与者接口上有 @TccBusinessAction 注解。
DTX-011	tcc 2.0 action record is empty,txId:[%s],actionId:[%s].	TCC 2.0 的参与者记录为空，可能是参与者的一阶段方法未执行，或者一阶段方法未执行成功，此时执行二阶段提交出现此异常。	检查一阶段方法是否成功执行。
DTX-012	tcc 2.0 action commit failed,txId:[%s],actionId:[%s].	TCC 2.0 的参与者执行二阶段提交方法时出现异常。	排除二阶段提交方法执行异常的原因
DTX-013	tcc 2.0 commit/rolback failed, txId or actionId is empty.	TCC 2.0 的参与者执行二阶段方法出现异常，原因是事务 ID 为空。	-
DTX-014	tcc 2.0 action rollback failed,txId:[%s],actionId:[%s].	TCC 2.0 的参与者执行二阶段回滚方法出现异常。	排除二阶段回滚方法执行异常的原因。
DTX-015	tcc anti-suspend, the 'TccAntiSuspendDAO' must not be null.	开启了 TCC 防悬挂，但是未配置 `TccAntiSuspendDAO`。	配置 `TccAntiSuspendDAO`，详见 TCC 防悬挂与幂等接入。

DTX-016	tcc anti-suspend(prepare method) intercept, txId:[%s].	TCC 一阶段方法重复执行，防悬挂拦截器被触发。	检查 TCC 一阶段方法被重复执行的原因。
DTX-017	tcc anti-suspend(prepare method) , the DataSource can't be null.	配置了防悬挂 `TccAntiSuspendDAO` ，必须设置其 dataSources 属性。	配置 `TccAntiSuspendDAO` 详见 TCC 防悬挂与幂等接入。
DTX-018	tcc anti-suspend, ITccAntiSuspendDAO didn't init.	TCC 防悬挂 `TccAntiSuspendDAO` 初始化未完成。	-
DTX-019	tcc anti-suspend failed, txId:[%s].	TCC 防悬挂控制出现未知异常。	-
DTX-020	@TwoPhaseBusinessAction method , has no serviceId.	TCC 参与者接口的 @TwoPhaseBusinessAction 主键的 name 属性为空。	TCC 参与者接口的 @TwoPhaseBusinessAction 主键的 name 属性不得为空，且需要全局唯一。
DTX-021	unknown actionType:[%s].	TCC 参与者接口的 @TwoPhaseBusinessAction 主键的 actionType 属性配置了未知值。	TCC 参与者接口的 @TwoPhaseBusinessAction 主键的 actionType 属性只能是 `local` 或 `remote` 。
DTX-022	report tcc branch failed, txId:[%s],actionId:[%s], serviceId:[%s].	创建 TCC 参与者的分支事务记录出现异常	检查分布式事务 SDK 与事务云服务的网络连接是否正常、检查事务云服务返回创建分支事务记录事务的原因。
DTX-023	report tcc branch error, txId:[%s],actionId:[%s], serviceId:[%s].	创建 TCC 参与者的分支事务记录出现未知异常。	检查分布式事务 SDK 与事务云服务的网络连接是否正常。

DTX-024	Dtx transaction timeout error, total time cost :%d (s), more than %d (s).	分布式事务执行超时，导致事务超时回滚。	检查事务执行超时原因，如需自定义事务超时时间，可以设置 @DtxTransaction 注解的 timeout 属性值（单位：秒）。
DTX-025	dtx transaction rollbacked error,txId[%s].	分布式事务回滚出现异常	检查分布式事务 SDK 和事务云服务之间的网络连接是否正常。
DTX-026	dtx transaction start failed, invoke dtx-server failed, request:[%s],response:[%s]	分布式事务开启创建主事务记录，事务云服务返回创建失败。	检查事务云服务创建主事务记录失败原因。
DTX-027	dtx transaction commit failed, request:[%s],response:[%s].	分布式事务提交，事务云服务返回提交失败。	检查事务云服务提交分布式事务失败原因。
DTX-028	dtx transaction commit error, request:[%s].	分布式事务提交出现未知异常。	检查分布式事务 SDK 和事务云服务之间的网络连接是否正常。
DTX-029	dtx transaction rollback failed, request:[%s],response:[%s]	分布式事务回滚，事务云服务返回回滚失败。	检查事务云服务回滚分布式事务失败原因。
DTX-030	dtx transaction rollback error, request:[%s].	分布式事务回滚出现未知异常。	检查分布式事务 SDK 和事务云服务之间的网络连接是否正常。
DTX-031	Dtx transaction timeout error, total time cost : %d (s), more than %d (s).	分布式事务执行超时，事务回滚。	检查事务执行超时原因，如需自定义事务超时时间，可以设置 @DtxTransaction 注解的 timeout 属性值（单位：秒）。

框架相关

错误码	错误消息	错误说明	解决方案

DTX-200	There must be only one framework, but found two, [%s] and [%s].	运行框架只能有一个，但是系统识别到多个运行框架。	-
DTX-201	There are more than one remoting protocols on one bean, beanName: [%s] with type [%s].	当前运行环境下，未找到任何远程通信协议。	-
DTX-202	There must be only one configuration, but found two, [%s] and [%s].	配置项文件只能有一个，但是系统识别到至少 2 个配置项文件。	确保配置项文件只有一个。
DTX-203	There must be one IConfiguration implement.	无任何配置项文件。	确保配置项文件存在。
DTX-204	No framework found.	未识别到任何运行框架。	确保运行框架是分布式事务能支持的。

SDK 与事务云服务通信相关

错误码	错误消息	错误说明	解决方案
DTX-301	There must be one address registry, but found none.	未找到事务云服务的地址。	检查事务云服务是否健康运行。
DTX-302	init dtx-servers' tcp long-conections failed.	分布式事务 SDK 与事务云服务建立长连接失败。	检查事务云服务是否健康运行。
DTX-303	unavailable dtx-server connections,serverIp: [%s]	分布式事务 SDK 无法建立到 serverIp 的 TCP 长连接。	检查事务云服务是否健康运行。
DTX-304	rpc (to dtx-server) invoke error.	向事务云服务发送消息出现异常。	检查事务云服务是否健康运行。

SQL 解析相关

错误码	错误消息	错误说明	解决方案
DTX-400	unknown dbType:[%s]	不支持的数据库类型	目前支持 Oceanbase、MySQL、Oracle，其他数据库暂不支持。
DTX-402	parser sql error,sql:[%s]	解析 SQL 语句出现异常	检查分布式事务是否支持 SQL 语法。
DTX-403	value type of param [%s]: [%s] is not support.	不支持的 SQL 语句表达式	检查分布式事务是否支持 SQL 语法。
DTX-403	unknown dbType for in select subquery.	嵌套子查询不支持	嵌套子查询不支持，请修改 SQL 语法。
DTX-404	insert's duplicateKeyUpdateClauses is not expression.	insert on duplicate key update 语句语法有错误	检查 SQL 语句语法是符合标准 SQL 语法。
DTX-405	the insert sql's columns or valurClauses is null.	insert 语句语法有错误	检查 SQL 语句语法是符合标准 SQL 语法。
DTX-406	aggregate method [%s] is not support, only support aggregate methods: SUM, COUNT, AVG, MIN, MAX.	集合函数目前只支持 SUM、COUNT、AVG、MIN、MAX，其它集合函数不支持	集合函数目前只支持 SUM、COUNT、AVG、MIN、MAX，修改 SQL 语句。
DTX-450	There must be primaryKey in table[%s] of db[%s].	业务表必须有主键	给所有业务表添加主键。

数据源相关（FMT 模式下）

错误码	错误消息	错误说明	解决方案
DTX-500	transaction [%s]:[%s] on db [%s] flush branchInfo to db failed.	保存 `undo log` 至 `dtx\_branch\_info` 表失败	检查业务数据库是否正常。

DTX-501	transaction [%s]:[%s] on db [%s] report branchInfo to dtx-server failed.	向事务云服务发送创建分支事务记录消息失败	检查分布式事务 SDK 与事务云服务网络连接是否正常、检查事务云服务是否正常。
DTX-502	transaction [%s]:[%s] on db [%s] commit error	业务本地事务提交失败	检查业务数据库是否正常。
DTX-503	transaction [%s]:[%s] on db [%s] rollback error	业务本地事务回滚失败	检查业务数据库是否正常。
DTX-504	sql parser result is empty, sql:[%s]	SQL 语句解析结果为空	检查 SQL 语法是否为分布式事务支持。
DTX-505	'INSERT INGORE' and 'INSERT ON DUPLICATE UPDATE' statement must contain primaryKey colmun(s), sql:[%s]	`INSERT INGORE` 和 `INSERT ON DUPLICATE UPDATE` 中不得使用自增主键句式	`INSERT INGORE` 和 `INSERT ON DUPLICATE UPDATE` 不支持使用自增主键，请在 SQL 语句的列与值中添加主键字段。
DTX-506	'UPDATE' and 'DELETE' statement must have where-statement, sql:[%s]	update 和 delete 语句编写要 where 条件	修改 SQL 语句，添加 where 条件。
DTX-507	'UPDATE' statement can not update primaryKey colmun(s), sql:[%s]	update 语句不得更新主键值	修改 SQL 语句。
DTX-508	Unknown sqlType, sql:[%s]	不支持的数据库类型	目前支持 Oceanbase、MySQL、Oracle，其他数据库暂不支持。
DTX-509	In READ_COMMITTED mode, complex query is not supported, you can use READ_UNCOMMITTED mode, sql:[%s]	读已提交模式不支持复杂查询语句	修改 SQL 语句为简单查询语句，或者使用读未提交模式。
DTX-510	Insert-select statement is not supported, sql:[%s]	Insert-select 语法不支持	修改 SQL 语句。

DTX-511	branchLogItems is empty, nothing to undo, dbId:[%s]	业务库无 undo log，undo 操作返回	-
DTX-512	check dirty failed, branchLogItem:[%s].	脏写校验失败，数据回滚异常	人工介入订正数据。
DTX-513	insert branchInfo error, branchInfo:[%s]	保存 undo log 至业务数据库出现异常	检查业务数据库是否正常。
DTX-514	lock table row failed, data may be locked by another transaction. db:[%s],tableName:[%s] ,primaryKey:[%s]	添加行锁失败，数据行锁正在被其他事务占用	检查数据行锁占用事务未结束原因。
DTX-515	insert rowlock failed, txId = %s, had tried count: %d	保存行锁数据至业务数据库失败	检查业务数据库是否正常、检查数据行锁是否被其他事务占用。
DTX-516	transaction:[%s] lock table row failed;row:[%s:%s] was locked by transaction:[%s].	行锁添加失败，行锁被其他事务占用	检查数据行锁占用事务未结束原因。
DTX-517	insert statement primary-key's value must be simple value, sql:%s.	insert 语句中的主键字段必须是简单数据类型：字符串或数字	修改主键数据类型。
DTX-518	Oracle insert statement primary-key's value can not be Sequence.nextVal/Sequence.currVal, sql:%s.	Oracle 数据库 insert 语句，主键不得使用 sequence	修改 insert 语句语法。
DTX-519	Insert statement primary-key's value can not be Expression, sql:%s.	insert 语句主键值不得是表达式	修改 insert 语句语法。

DTX-520	Unsupport dbType:%s.	不支持的数据库类型	目前支持 Oceanbase、MySQL、Oracle，其他数据库暂不支持。
DTX-521	unsupport sub query in where condition.	不支持 where 条件中的子查询	修改 SQL 语句语法。
DTX-522	join select sql can not get one tableName.	join 语句获取多个表名，会产生歧义，不支持 join 语句	修改 SQL 语句语法或者改成读未提交模式。
DTX-523	select from more than on tableName or sub select statement, can not get a tableName.	join 语句获取多个表名，会产生歧义，不支持 join 语句	修改 SQL 语句语法，或者改成读未提交模式。
DTX-524	select from nest sub select statement, can not get a tableName.	不支持嵌套查询	修改 SQL 语句语法，或者改成读未提交模式。
DTX-525	select sql has no tableName.	查询语句无表名称	修改 SQL 语句语法。
DTX-526	unknown parameter, %s:%s.	不支持的数据类型	修改 SQL 语句语法。
DTX-527	executeBatch error, statement:%s.	`executeBatch` 执行出现异常	根据异常类型判断出错原因。
DTX-528	execute Statement, but sql is empty.	SQL 语句为空	
DTX-529	execute statement error, statement:%s.	执行 SQL 语句出现异常	根据异常类型判断出错原因。
DTX-530	Statements is empty, no sql need execute.	SQL 语句为空	-
DTX-531	there is not sql to executed.	SQL 语句为空	-

DTX-532	while getAutoCommit() is true can not invoke executeBatch.	`executeBatch` 方法必须在数据库事务内执行	请先开启数据库事务。
DTX-533	AddBatch count is not equal to parameters row count.	`addBatch` 参数个数不匹配	检查业务代码，确保正确使用了 `addBatch` 。
DTX-534	All parameters of batch prepareStatement must have addBatch() follow.	`executeBatch` 的参数设置之后必须调用 `addBatch` 方法	检查业务代码，确保正确使用了 `addBatch` 。
DTX-535	Invoking executeBatch() method need all sql statements are addBatch(), but sql[%s] is not.	`executeBatch` 用法不正确	检查业务代码，确保正确使用了 `addBatch` 。
DTX-536	Invoking executeBatch() method need all sql statements are addBatch(sql),but sql[%s] is not.	`executeBatch` 用法不正确	检查业务代码，确保正确使用了 `addBatch` 。
DTX-537	Invoking executeQuery/executeUpdate/execute method need all sql statements are non-addBatch sql, but sql[%s] is addBatch.	`addBatch` 的 SQL 调用了 `executeQuery/executeUpdate/execute` 方法执行	检查业务代码，确保正确使用了 JDBC。
DTX-538	PreparedStatement with addBatch sql cann't use limit-statement, sql:[%s].	`PreparedStatement` 的 `addBatch` 用法，SQL 语句中不支持带 limit 分页句式	修改业务代码及 SQL 语句。
DTX-539	parameter count ambiguous, where-statement expect the count more than %d , but parameter count is %d.	`PreparedStatement` 的 SQL 语句的参数个数与实际参数个数不匹配	检查代码，确保参数个数与 SQL 语句中参数个数相等。

DTX-540	parameter index ambiguous, where-statement expect the index is %d, but parameter count is %d.	`PreparedStatement` 的 SQL 语句的参数个数与实际参数个数不匹配	检查代码，确保参数个数与 SQL 语句中参数个数相等。
DTX-541	parameter data ambiguous, where-statement expect the parameter row count is %d, but actual is %s.	`PreparedStatement` 的批量 SQL 语句的参数行数与实际参数行数不匹配	检查代码，确保参数个数与 SQL 语句中参数个数相等。
DTX-542	query log error, querySql: [%s],prameters: [%s],isPrepare:%s.”	查询 undo log 出现异常	检验业务数据库是否可用。
DTX-543	check dirty data error,tableName: [%s],oldRows: [%s],newRows:[%s].	脏写校验失败，业务数据出现脏写	人工介入订正数据。
DTX-544	auto-generate primary-key must be only one column,sql: [%s].	自增主键不能是复合主键	修改主键为单列主键。
DTX-545	addBatch() must include primary-key(s) in sql, auto-generate primary-key don’t support,sql: [%s].	`addBatch` 的 SQL 语句中必须包含主键，不支持自增主键	修改 SQL 语句。
DTX-546	insert without primary-key(s), do not support batch insert ,sql: %s.	批量插入时，SQL 语句中必须有主键	修改 SQL 语句，添加主键列至 SQL 语句中 。
DTX-547	preparedstatement addBatch, insert sql’s values must be single,the sql:[%s].	`preparedstatement` 的 `addBatch` 用法时，insert 语句不支持批量	修改 SQL 语句。

DTX-548	parameter index ambiguous, where-statement expect the index is %d, but parameter count is %d.	SQL 语句需要的参数个数与实际的参数个数不匹配	检查业务代码和 SQL 语句的参数是否匹配。
DTX-550	parameter data ambiguous, where-statement expect the parameter row count is %d, but actual is %s.	SQL 语句需要的参数个数与实际的参数个数不匹配	检查业务代码和 SQL 语句的参数是否匹配。
DTX-551	parameter index ambiguous, where-statement expect the index is %d, but parameter count is %d.	SQL 语句需要的参数个数与实际的参数个数不匹配	检查业务代码和 SQL 语句的参数是否匹配。
DTX-552	parameter data ambiguous, where-statement expect the parameter row count is %d, but actual is %s.	SQL 语句需要的参数个数与实际的参数行数不匹配	检查业务代码和 SQL 语句的参数是否匹配 。
DTX-553	the type of primary key must be ‘String’ or ‘Number’.	主键字段的类型必须是字符串或者数字	修改主键、或者修改主键数据类型。
DTX-554	can not get the primarykey of insert statement: [%s],parameters:[%s].	自增主键的 insert 语句，无法获取 insert 数据之后的主键值	修改 insert 语句添加主键。
DTX-556	the columns count of sql’s values is not equal to the columns count of the table, table’s column count: [%d],columns values: [%d]	SQL 语句中的字段列与业务表的字段列个数不匹配	检查 SQL 语句是否正确；如果数据库表结构变更过，请重启机器刷新分布式事务缓存 。
DTX-557	check undo datas result is true, undo do nothing, branchLogItem:[%s].	undo log 数据为空，undo 操作返回	-

DTX-558	unknown sqlType, sqlType: %s, branchLogItem:[%s].	不支持的 SQL 语句类型	参考分布式事务开发指南，修改 SQL 语句为分布式事务支持的 SQL 类型。
DTX-559	check dirty datas failed, the data is dirty, tableName: [%s],postRows: [%s],dbRows: [%s],branchLogItem: [%s].	数据脏写校验失败，业务数据出现脏写	人工介入订正数据。
DTX-560	insert branch info into oracle db failed, txId:%s, actionId: %s.	向 Oracle 数据库保存 undo log 出现异常	检查业务数据库是否正常运行。
DTX-561	connection must setAutoCommit(false), conn: %s, rowLocks: %s.	保存行锁记录时，未开启数据库事务。	-
DTX-562	preparedStatement only support single sql,but sql[%s] has more one sql.	`preparedStatement` 中不支持多个 SQL 语句	修改 SQL 语句语法。
DTX-563	transaction [%s] on db [%s] executeQuery(%s) error.	`executeQuery` 方法执行出现未知异常	根据抛出的异常类型找出异常原因。
DTX-564	transaction [%s] on db [%s] executeUpdate([%s]) error.	`executeUpdate` 方法执行出现未知异常	根据抛出的异常类型找出异常原因。
DTX-565	transaction [%s] on db [%s] executeUpdate([%s], int) error.	`executeUpdate` 方法执行出现未知异常	根据抛出的异常类型找出异常原因。
DTX-566	transaction [%s] on db [%s] executeUpdate([%s], int[]) error.	`executeUpdate` 方法执行出现未知异常	根据抛出的异常类型找出异常原因。

DTX-567	transaction [%s] on db [%s] executeUpdate([%s], String[]) error.	`executeUpdate` 方法执行出现未知异常	根据抛出的异常类型找出异常原因。
DTX-568	transaction [%s] on db [%s] execute([%s]) error.	`execute` 方法执行出现未知异常	根据抛出的异常类型找出异常原因。
DTX-569	transaction [%s] on db [%s] execute([%s], int) error.	`execute` 方法执行出现未知异常	根据抛出的异常类型找出异常原因。
DTX-570	transaction [%s] on db [%s] execute([%s], int[]) error.	`execute` 方法执行出现未知异常	根据抛出的异常类型找出异常原因。
DTX-571	transaction [%s] on db [%s] execute([%s],String[]) error.	`execute` 方法执行出现未知异常	根据抛出的异常类型找出异常原因。
DTX-572	addBatch(String) is not allowed while getAutoCommit() is true.	`addBatch(String)` 方法必须在开启数据库事务之后使用	在使用 `addBatch(String)` 方法前先开启数据库事务。
DTX-573	executeBatch() is not allowed while getAutoCommit() is true.	`executeBatch` 方法必须在开启数据库事务之后使用	在使用 `executeBatch` 方法前先开启数据库事务。
DTX-574	don't support closeOnCompletion() ; need explicit invoke close() method to close the statement.	`closeOnCompletion` 方法不支持	手动调用 `close` 方法关闭 Statement。
DTX-576	WrappedDtxDataSource .getUniqueDbId() is empty.	`WrappedDtxDataSource` 的 uniqueDbId 属性不得为空	设置 `WrappedDtxDataSource` 的 uniqueDbId 属性，需要保证全局唯一。

DTX-577	WrappedDtxDataSource.targetDataSource is null.	`WrappedDtxDataSource` 的 targetDataSource 属性不得为空	设置 `WrappedDtxDataSource` 的 targetDataSource 属性。
DTX-578	WrappedDtxDataSource.targetDataSource can not be type of IDtxDataSource.	`WrappedDtxDataSource` 的 targetDataSource 属性所指的 DataSource 对象不能是 IDtxDataSource 类型	修改 `WrappedDtxDataSource` 的 targetDataSource 属性为非 IDtxDataSource 类型。
DTX-579	DruidDtxDataSource.jdbcUrl is empty.	`DruidDtxDataSource` 的 dbType 属性不得为空	设置 `DruidDtxDataSource` 的 dbType 属性。
DTX-580	DruidDtxDataSource.userName is empty.	`DruidDtxDataSource` 的 userName 属性不得为空	设置 `DruidDtxDataSource` 的 userName 属性。
DTX-581	DruidDtxDataSource.password is empty.	`DruidDtxDataSource` 的 password 属性不得为空	设置 `DruidDtxDataSource` 的 password 属性。

8.2. 事务积压排查

分布式事务常见的问题是发生事务积压，事务积压的原因通常是参与方问题导致的，可能是由于参与方二阶段执行失败或执行超时，也可能是因为参与方与分布式事务服务端连接异常等问题。本文将基于参与方的常见问题，引导您快速排查并解决事务积压问题。

参与方二阶段执行失败

在参与方的二阶段执行日志 `${user.home}/logs/dtx-drm.log` 中，搜索事务 ID，检查是否是由于参与方二阶段业务本身执行失败，返回 false，再继续分析业务逻辑找到具体的执行失败原因。

执行失败日志示例如下：

```
2019-06-2517:19:54,061 INFO  dtx-bolt-pool-12- branch commit result:[false], message:[class
:[com.alipay.dtx.rpc.message.server.request.BranchCommitRequest]:{"actionContext":{"sys::ro
llback_method":"rollback","action-start-time":1561453729969,"sys::prepare":"doPrepare","tra
cerInfo":"tcid=0d0d000615614537298321002748 10&spid={spid}&pspid={pspid}&sample=false&dtx_id
=3521a9ab55474738a296eda97fe39f6701-0000000000&","appName":"sofa-server","rpcId":"0.3949f9b
2","sys::commit_method":"commit","host-name":"10.81.60.101","tracerId":"0d0d000613333337298
32100274810","sys::class_name":"com.alipay.dtx.sample.action.facade.FirstAction","actionNam
e":"FirstAction"},"actionId":"94021b09f8314aceb3ffede77b682900","activityContext":{"bizType
":"trade","start-time":1561453729846,"tracerInfo":"tcid=0d0d000615614537298321002748 10&spid
=0&pspid=&sample=false&","appName":"sofa-client","isolationLevel":"READ_UNCOMMITTED","rpcId
":"0","host-name":"13.13.0.6","tracerId":"0d0d000613333337298321002748 10"},"instanceId":"VO
QKxxxXCWID","resourceId":"SOFABOOT_RPC&FirstAction&&","resourceType":2,"txId":"3521a9ab5547
4738a296eda97fe39f6701-0000000000"}].
```

参与方二阶段执行超时

检查 `${user.home}/admin/logs/dtxserver/common-error.log` 系统错误日志，查看是否是由于参与方二阶段执行超时问题造成的事务积压。

执行超时日志示例如下：

```
2019-06-2517:08:53.264[message-executor] ERROR DTX-REMOTE -BusinessAction handle exception
, txId:3521a9ab55474738a296eda97fe39f6701-0000000000, instanceId:VxxKVVWXCWID com.alipay.re
moting.rpc.exception.InvokeTimeoutException:Rpc invocation timeout[responseCommand TIMEOUT]
! the address is10.25.172.207:51083
at com.alipay.remoting.rpc.RpcResponseResolver.preProcess(RpcResponseResolver.java:91)
at com.alipay.remoting.rpc.RpcResponseResolver.resolveResponseObject(RpcResponseResolver.ja
va:56)
....
```

参与方初始化失败

参与者启动时，可能会因为某些原因初始化失败，从而造成了事务积压。因此，您可以根据启动时间，排查DTX 客户端的初始化日志：`${user.home}/admin/logs/dtx/dtx-client.log`。

参与方与 DTX Server 连接失败

1. 根据参与方启动时间，查看 DTX Server 的建连日志 `${user.home}/admin/logs/dtx/dtx-rpc.log`，检查参与方是否与 DTX Server 建连失败，从而造成了事务积压。
2. 在参与方的机器上执行以下命令，确认其与 DTX Server 端口的连接情况：

   ```
   netstat -anp|grep 14200
   ```

   若发现参与方与某台 DTX Server 没有建立连接，当这台 DTX Server 进行提交操作（Commit）时，则会报错：`java.lang.RuntimeException: connection is null,targetIp:{xxx},txId:{xxx}`。此时，快速的恢复方法是重启参与方的应用，后续再排查连接无法建立的具体原因。

其他原因

除以上常见的参与方问题外，还可能是由于其他原因。您可以通过查看参与方二阶段执行日志 `${user.home}/logs/dtx-drm.log` 或者DTX Server 错误日志 `${user.home}/admin/logs/dtxserver/common-error.log` 根据具体的错误信息，具体分析原因。

8.3. 异常事务处理方案

清除异常任务数据

问题现象

事务提交异常，并且业务需求不再需要提交，需要清除异常事务。

问题原因

业务代码问题导致事务提交异常。

解决方案

1. 进入 dtxserver 的 `dtx_remote` 库。
2. 使用如下命令查询 `business_action` 、 `business_activity` 表中的 txid 记录：

   ```
   select * from business_action_01 where tx_id = "XXXXXX"
   select * from business_activity_01 where tx_id = "XXXXXX"
   ```

3. 清除异常任务信息。

   i. 若 `dtx_remote` 库的 `business_action` 、 `business_activity` 表不为空，则使用如下命令清除异常任务信息：

   ```
   delete from business_action_01  where tx_id ="XXXXXX"
   delete from business_activity_01 where tx_id ="XXXXXX"
   ```

   ii. 若 `dtx_remote` 库的 `business_action` 、 `business_activity` 表为空，则进行如下操作：

   a. 根据客户的分库分表规则找出对应分库。

   b. 根据事务 ID 查找到 DTX 数据库，进入 `dtx_shard_X` 分库。

   c. 使用如下命令查询分库中 `business_action` 、 `business_activity` 表中的 txid 记录：

   ```
   select * from business_action_01 where tx_id = "XXXXXX"
   select * from business_activity_01 where tx_id = "XXXXXX"
   ```

   > **说明**
   >
   > 若想要提高查询速度，您可以在 DTX 配置数据库 `dtx_config` 使用 `delete from dtx_transaction where tx_id="xxxxxx"` 命令清除事务统计表的数据：

   d. 使用如下命令清除异常任务信息：

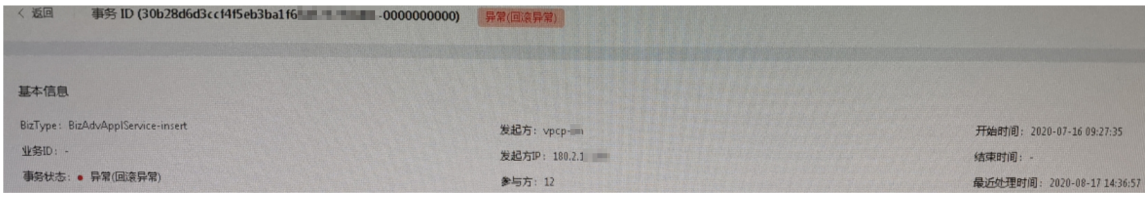
   ```
   delete from business_action_01  where tx_id ="XXXXXX"
   delete from business_activity_01 where tx_id ="XXXXXX"
   ```

4. 清除异常任务后，单击 **事务详情**，若发生报错“java.langIllegalArgumentException:发起方数据不存在，可能已经被清理”，表示已清除成功。

事务回滚异常

问题现象

- 事务状态显示：异常（回滚异常）。



- 发起方 `common-error.log` 日志出现大量报错“DTX-REMOTE - invokeAction rollback finish"。

问题原因

DTX 参与方不在线，DTX server 无法联系参与方回滚成功。

解决方案

以下提供两种解决方案：

- 通过日志排查参与方不在线原因。

  在 `dtx-remote.log` 或者 `commen-error.log` 日志中查看报错信息，根据报错信息找出参与方不在线原因。

- 若事务已不需要回滚，则删除该事务回滚即可。

  i. 进入 dtxserver 的 `dtx_remote` 库。

  ii. 使用如下命令查询 `business_action` 、 `business_activity` 表中的 txid 记录：

  ```
  select * from business_action_01 where tx_id = “XXXXXX"
  ```

  ```
  select * from business_activity_01 where tx_id = “XXXXXX"
  ```

  iii. 清除 txid 数据。

  TCC 模式

  使用如下命令将 dtxsever 的 `dtx_remote` 库中 txid 数据清除：

  ```
  delete from business_activiy where tx_id = "XXXXXX"
  ```

  ```
  delete from business_action where tx_id = "XXXXXX"
  ```

  FMT 模式

  a. 使用如下命令，将 dtxsever 的 `dtx_remote` 库中 txid 数据清除：

  ```
  delete from business_activiy where tx_id = "XXXXXX"
  ```

  ```
  delete from business_action where tx_id = "XXXXXX"
  ```

  b. 使用如下命令，将用户的业务数据库中 `dtx_branch_info` 和 `dtx_row_lock` 表中 txid 数据清除：

  ```
  delete from dtx_branch_info where tx_id = "XXXXXX"
  ```

  ```
  delete from dtx_row_lock where tx_id = "XXXXXX"
  ```

  iv. 清除异常任务后，单击 **事务详情**，若发生报错“java.langIllegalArgumentException:发起方数据不存在，可能已经被清理”，表示已清除成功。

9.常见问题

本文主要介绍在操作分布式事务过程中的常见问题。

SOFAStack 是否可以部署在虚拟机、物理机上？

可以，SOFAStack 部署支持物理机、虚机机 VM。

SOFAStack 当前是否只支持阿里云飞天 ACK？

SOFAStack 部署支持多云异构、目前支持华为云、开源 Openstack 等。

分布式事务支持哪些数据库？

目前支持 MySQL、Oracle、RDS MySQL、PostgreSQL 等主流数据库。

分布式事务是否支持 VPC 私有云部署？

支持，如有需要，请咨询售后技术支持。

在分布式事务服务挂掉的情况下，事务是否能正常回滚？

分布式事务的服务端会记录下事务的状态并持久化，即使服务挂掉，在服务重新启动后，事务仍然能够正常回滚。

建表时，是否可以使用 SQL 的关键字作为列的名称？

不可以，建表时列名不能使用 SQL 的关键字，如 DESC、TABLE 等。

接入分布式事务时需要依赖什么？

如果您使用的是数据访问代理中的分布式事务，因为数据访问代理和分布式事务已经深度集成，部署时是不需要任何依赖的，仅需一行事务开启 SQL 语句就能开启事务。

如果您是基于 SDK 方式使用分布式事务，就需要在引入的分布式事务 SDK 的 `pom.xml` 中加入相关依赖，详见开发指南。

开通分布式事务之后，SOFABoot、Dubbo、消息队列、数据访问代理、RDS、MySQL、Oracle、OceanBase 能否都加入分布式事务？

可以，分布式事务已经打通了 SOFABoot、Dubbo、消息队列、数据访问代理、RDS、MySQL、Oracle、OceanBase 之间的事务。

服务 A 调用服务 B，服务 A 上加了分布式事务并开启事务注解，服务 B 没有注解，那么 A 和 B 是否在同一个分布式事务中？

服务 A 和 B 在一个事务中。只要服务 A 的函数上加入了分布式事务的注解，那么服务 A 注解函数下面所有的被调用的服务及其子调用的服务都会加入到这个服务 A 的事务中。

分布式事务是否支持多机房高可用？

分布式事务支持同城、异地多活，高可用。当一个机房由于断网、停电等突发状况造成全机房不可用时，分布式事务能切换到备用机房继续提供服务。

如何判断分布式事务 SDK 与事务云服务的网络连接是否正常？

在事务云服务上执行 `netstat -an | grep 14200` ，看是否存在 TCP 连接。如果存在 TCP 连接，说明事务云服务启动正常；否则说明事务云服务已宕机或者网络已中断。

分布式事务 SDK 与事务云服务网络连接正常，但是通信失败，是什么原因？

TCP 连接正常，说明事务云服务启动正常。

分布式事务 SDK 与事务云服务通信返回失败消息，说明 SDK 发送给事务云服务的消息违反某些约束条件，可以根据事务 ID 在 `dtx-rpc.log` 中查询事务云服务返回的具体失败信息。

如何查看分布式事务的日志？

分布式事务的事务日志存放路径为 `${user.home}/logs/dtx-xxx.log` ，其中 `${user.home}` 为启动分布式事务 SDK 的用户账户的根目录。不同的日志文件，打印的日志内容也不同，具体如下：

- `dtx-client.log` 打印 SDK 启动和所有事务执行情况的日志。
- `dtx-rpc.log` 打印 SDK 与分布式事务服务端的所有通信信息。
- `dtx-parser.log` 打印 SQL 语句语法解析日志。
- `dtx-rm.log` 打印所有参与者（TCC、FMT）的二阶段执行日志。
- `dtx-datasource.log` 打印 FMT 模式下的事务执行日志。

查看日志时，可以按照分布式事务的事务 ID 去查看这个事务在该 SDK 都操作了哪些资源及其提交和回滚情况。

分布式事务 TCC @TwoPhaseBusinessAction 放到接口的实现类上是否有效？

没有效果，不能放到实现类上，必须放在接口上。

```
   public interface UserFacade{

        @TwoPhaseBusinessAction(name ="sendMessage", commitMethod="commit", rollbackMethod
="cancel")
        public boolean sendMessage(final BusinessActionContext businessActionContext,@Busin
essActionContextParameter(paramName ="userId")String userId);

        public boolean commit(final BusinessActionContext businessActionContext);

        public boolean cancel(final BusinessActionContext businessActionContext);

    }
```

如何传递参数给二阶段方法 Commit/Cancel？

可以通过以下任一方式：

- 使用额外的存储来记录 txid 和业务数据的关系。
- 使用 BusinessActionContext 传递。

> **说明**
>
> 不能通过直接设置 BusinessActionContext 来传递，此种方式不会被保存。需要使用 @BusinessActionContextParameter 注解。

DTX 发起方和参与方同属一个应用，能否不使用 RPC？

可以在参与方接口上添加 `@InjvmRemoting` 注解，这样发起方去调用参与方就是 JVM 内调用。而且这种情况下，DTX 服务端来远程调用二阶段方法也不会通过注册中心，而是 DTX 自闭环寻址处理。

以 TCC 模式为例：

```
 // 注解InjvmRemoting用于当发起方与参与方是同一个应用，也就是说参与方不需要被其他应用通过注册中心寻址来进行RPC调用
    @InjvmRemoting
    public interface AcctDepositTccService{

        /**
         * <p>
         * 贷记记账接口 TCC模式
         * </p>
         *
         * <p>
         * 结合账户的余额方向来对账户的资金进行流入 or 流出操作
         * </p>
         *
         * @param accountTransRequest   请求对象参数,详情请见{@link AccountTransRequest}
         * @param businessActionContext dtx框架中请求的上下文信息，框架级别参数，调用者不需传递，由dtx自动注入。
         * @return AccountTransResult 交易处理结果
         */
        @TwoPhaseBusinessAction(name ="creditAction", commitMethod ="commit", rollbackMethod ="rollback", antiSuspend =true)
        public AccountTransResult credit(AccountTransRequest accountTransRequest,@ShardingKeyString accountNo,
                BusinessActionContext businessActionContext);

        /**
         * 二阶段提交
         *
         * @param businessActionContext xts上下文
         * @return TwoPhaseResult#isSuccess() 是否成功，true-成功，false-失败
         */
        public boolean commit(BusinessActionContext businessActionContext);

        /**
         * 二阶段回滚
         *
         * @param businessActionContext xts上下文
         * @return TwoPhaseResult#isSuccess() 是否成功，true-成功，false-失败
         */
        public boolean rollback(BusinessActionContext businessActionContext);

    }
```

DTX 创建事务分支时出现“Data too long for column ‘log_Info’”报错

问题现象

DTX 创建事务分支报错：Data too long for column ‘log_Info’。

问题原因

业务代码更新的数据库内容太多，超过了 `log_info` 的最大长度。

解决方案

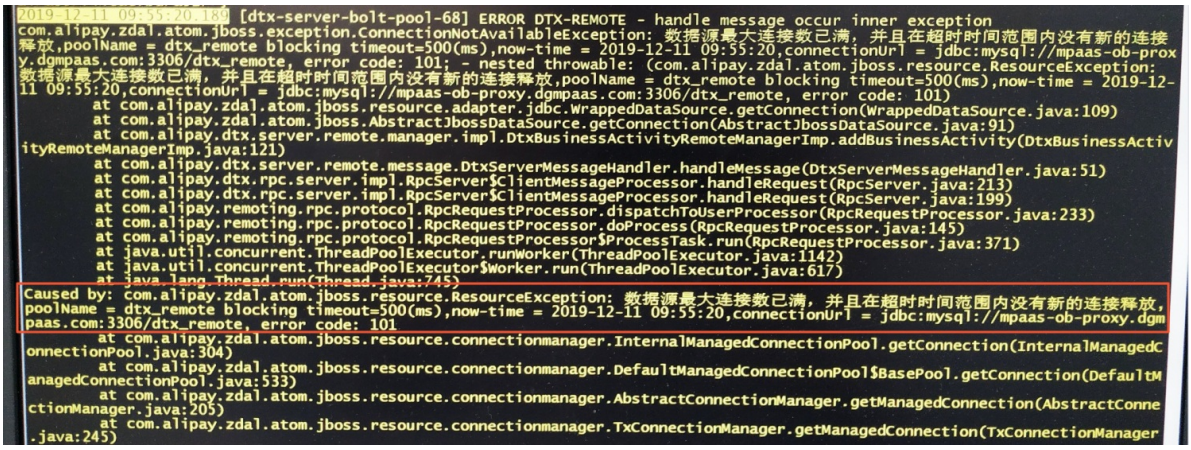
需要优化业务代码，最好是业务优化 SQL 逻辑，或者修改表中该字段的类型，比如 `LONGBLOB` 。

客户端启动分布式事务时出现“invoke dtx-server to create Activity failed”报错

问题现象

- 客户端启动 DTX 失败，报错：DTX-037: invoke dtx-server to create Activity failed.
- 在 DTX server 端，实际错误码为 `error code: 101` ，如下图所示：

- 在控制台转换字符编码，可以看到实际的错误是：数据源最大连接数已满，并且在超时时间范围内没有新的连接释放，如下图所示：



问题原因

DTX server 与 DTX DB 的数据库连接数设置成了 2。

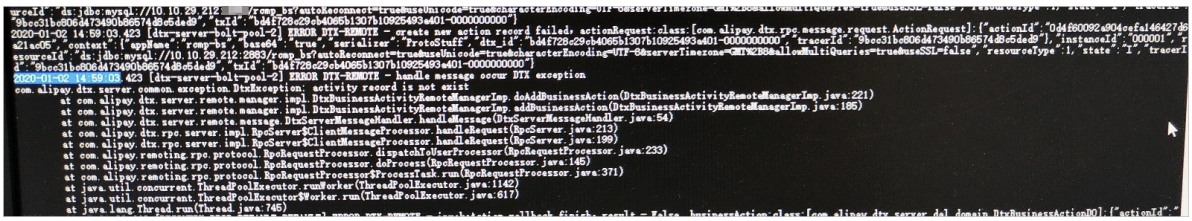
解决方案

如有需要，您可以咨询售后技术支持。

事务进行过程中发生“DTX-036 activity record is not exist”报错

问题现象

- 报错：DTX-036 activity record is not exist。
- 详细的错误信息下图所示：



问题原因

根据上图所示的错误信息，事务的请求者和事务的参与者的 instance ID 不匹配。例如事务 `ds:jdbc:mysql://10.10.29.212:XXXX` 配置错误，instance ID 配置成了 `00000X` ，使得事务请求方 `instance ID = 00000X` 。而事务的真实 instance ID 其实是 `8PCDDO6YXXXX` 。

解决方案

排查事务的参与方的 `application.properties` 配置，确保事务的请求者和事务的参与者的 instance ID 匹配。

DTX 2.2.1 版本，支持 Saga 模式吗？

不支持，如有需要，请咨询售后科技支持。

分布式事务二阶段请求未执行

问题现象

使用时出现报错：“com.alipay.dtx.common.exception.DtxException:DTX-032:TCC prepare menthod must have a parameter who's type isBusinessActionContext”。

启动时出现报错：“DTX-302:initdtx-servers' tcp long-conections failed”。

问题原因

- dtxserver 未发起二阶段请求。
- 客户端收到二阶段请求但执行失败。

解决方案

针对以上两种情况有两种解决方案：

- dtxserver 未发起二阶段请求

  登录 dtxserver 机器，在 `/home/admin/logs/dtxserver/dtx-remote.log` 日志中搜索事务 ID，查找该异常事务相关信息。

  - 若未找到该异常事务的恢复日志，说明该事务的二阶段任务未开始执行，可能是未正常执行或出现事务积压，需重新发起事务。
  - 若找到该异常事务的恢复日志，说明执行过程出错，在 `dtx-remote.log` 和 `commen-error.log` 中查看出错原因，在改进后重新发起任务。

- 客户端收到二阶段请求但执行失败

  i. 查询客户端是否收到二阶段的请求。

  在客户端 `/home/admin/logs/dtx/dtx-rpc.log` 日志中搜索事务 ID，若找到“BranchCommitRequest”或“BranchrollbackRequest”的消息，说明客户端收到了二阶段请求。

  ii. 查看二阶段请求是否成功执行。

  在 `/home/admin/logs/dtx/dtx-rm.log` 日志中搜索事务 ID，若找到“branch commit result”或“branch rollback result”相关内容，且返回结果为 true ，则表示执行成功；反之，执行失败，根据报错信息排查执行失败的原因，在改进后回滚事务重新执行。

9.1. 产品相关问题

SOFAStack 是否可以部署在虚拟机、物理机上？

可以，SOFAStack 部署支持物理机、虚机机 VM。

SOFAStack 当前是否只支持阿里云飞天 ACK？

SOFAStack 部署支持多云异构、目前支持华为云、开源 Openstack 等。

分布式事务支持哪些数据库？

目前支持 MySQL、Oracle、RDS MySQL、PostgreSQL 等主流数据库。

分布式事务是否支持 VPC 私有云部署？

支持，如有需要，请提交 工单申请，或者咨询金融科技售后。

DTX 2.2.1 版本，支持 Saga 模式吗？

不支持，如有需要，请提交 工单申请，或者咨询金融科技售后。

接入分布式事务时需要依赖什么？

如果您使用的是数据访问代理中的分布式事务，因为数据访问代理和分布式事务已经深度集成，部署时是不需要任何依赖的，仅需一行事务开启 SQL 语句就能开启事务。

如果您是基于 SDK 方式使用分布式事务，就需要在引入的分布式事务 SDK 的 `pom.xml` 中加入相关依赖，详见开发指南。

分布式事务是否支持多机房高可用？

分布式事务支持同城、异地多活、高可用。当一个机房由于断网、停电等突发状况造成全机房不可用时，分布式事务能切换到备用机房继续提供服务。

开通分布式事务之后，SOFABoot、Dubbo、消息队列、数据访问代理、RDS、MySQL、Oracle、OceanBase 能否都加入分布式事务？

可以，分布式事务已经打通了 SOFABoot、Dubbo、消息队列、数据访问代理、RDS、MySQL、Oracle、OceanBase 之间的事务。

服务 A 调用服务 B，服务 A 上加了分布式事务并开启事务注解，服务 B 没有注解，那么 A 和 B 是否在同一个分布式事务中？

服务 A 和 B 在一个事务中。只要服务 A 的函数上加入了分布式事务的注解，那么服务 A 注解函数下面所有的被调用的服务及其子调用的服务都会加入到这个服务 A 的事务中。

9.2. 日志相关问题

如何查看分布式事务的日志？

分布式事务的事务日志存放路径为 `${user.home}/logs/dtx-xxx.log` ，其中 `${user.home}` 为启动分布式事务 SDK 的用户账户的根目录。不同的日志文件，打印的日志内容也不同，具体如下：

- `dtx-client.log` 打印 SDK 启动和所有事务执行情况的日志。
- `dtx-rpc.log` 打印 SDK 与分布式事务服务端的所有通信信息。
- `dtx-parser.log` 打印 SQL 语句语法解析日志。
- `dtx-rm.log` 打印所有参与者（TCC、FMT）的二阶段执行日志。

- `dtx-datasource.log` 打印 FMT 模式下的事务执行日志。

查看日志时，可以按照分布式事务的事务 ID 去查看这个事务在该 SDK 都操作了哪些资源及其提交和回滚情况。

9.3. 网络连接问题

如何判断分布式事务 SDK 与事务云服务的网络连接是否正常？

在事务云服务上执行 `netstat -an | grep 14200` ，看是否存在 TCP 连接。如果存在 TCP 连接，说明事务云服务启动正常；否则说明事务云服务已宕机或者网络已中断。

分布式事务 SDK 与事务云服务网络连接正常，但是通信失败，是什么原因？

TCP 连接正常，说明事务云服务启动正常。

分布式事务 SDK 与事务云服务通信返回失败消息，说明 SDK 发送给事务云服务的消息违反某些约束条件，可以根据事务 ID 在 `dtx-rpc.log` 中查询事务云服务返回的具体失败信息。

9.4. 事务执行问题

在分布式事务服务挂掉的情况下，事务是否能正常回滚？

分布式事务的服务端会记录下事务的状态并持久化，即使服务挂掉，在服务重新启动后，事务仍然能够正常回滚。

DTX 创建事务分支时出现“Data too long for column ‘log_Info’”报错

问题现象

DTX 创建事务分支报错：Data too long for column ‘log_Info’。

问题原因

业务代码更新的数据库内容太多，超过了 `log_info` 的最大长度。

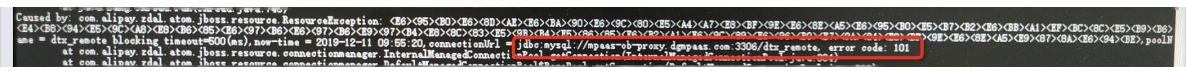
解决方案

需要优化业务代码，最好是业务优化 SQL 逻辑，或者修改表中该字段的类型，比如 `LONGBLOB` 。

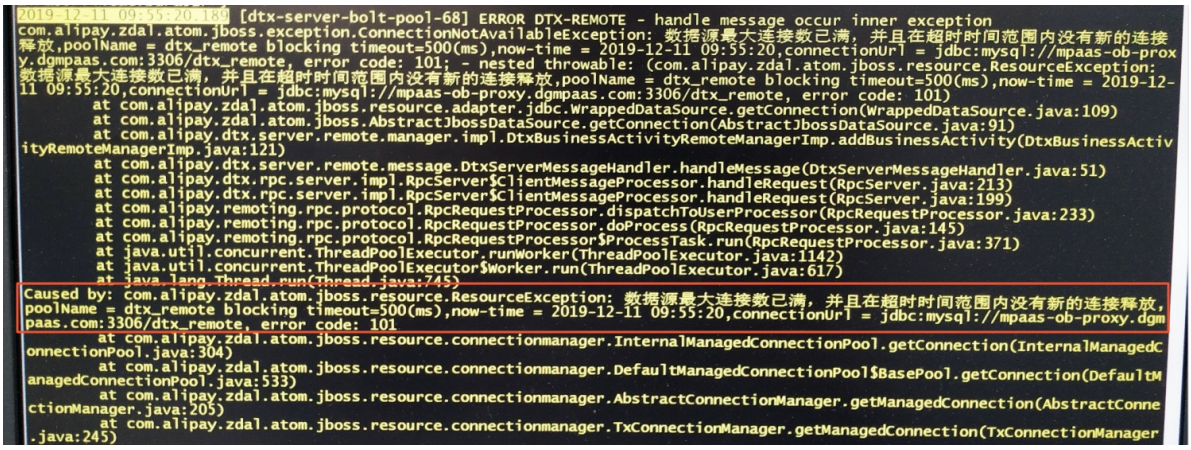
客户端启动分布式事务时出现“invoke dtx-server to create Activity failed”报错

问题现象

- 客户端启动 DTX 失败，报错：DTX-037: invoke dtx-server to create Activity failed.
- 在 DTX server 端，实际错误码为 `error code: 101` ，如下图所示：



- 在控制台转换字符编码，可以看到实际的错误是：数据源最大连接数已满，并且在超时时间范围内没有新的连接释放，如下图所示：



问题原因

DTX server 与 DTX DB 的数据库连接数设置成了 2。

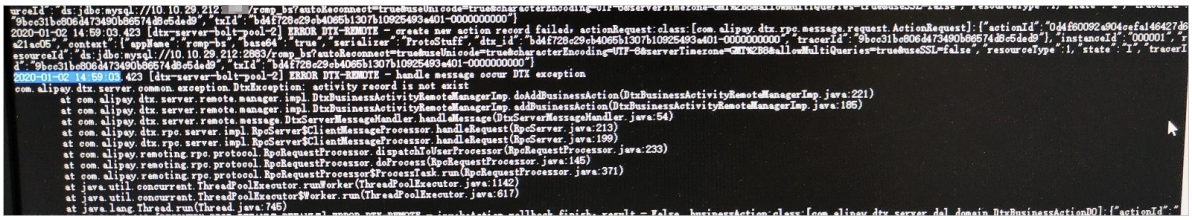
解决方案

如有需要，您可以提交 工单申请，或者咨询金融科技售后。

事务进行过程中发生“DTX-036 activity record is not exist”报错

问题现象

- 报错：DTX-036 activity record is not exist。
- 详细的错误信息下图所示：



问题原因

根据上图所示的错误信息，事务的请求者和事务的参与者的 instance ID 不匹配。例如事务 `ds:jdbc:mysql://10.10.29.212:XXXX` 配置错误，instance ID 配置成了 `00000X`，使得事务请求方 `instance ID = 00000X`。而事务的真实 instance ID 其实是 `8PCDDO6YXXXX`。

解决方案

排查事务的参与方的 `application.properties` 配置，确保事务的请求者和事务的参与者的 instance ID 匹配。

如何传递参数给二阶段方法 Commit/Cancel?

可以通过以下任一方式：

- 使用额外的存储来记录 txid 和业务数据的关系。
- 使用 BusinessActionContext 传递。

> **说明**
>
> 不能通过直接设置 BusinessActionContext 来传递，此种方式不会被保存。需要使用 @BusinessActionContextParameter 注解。

分布式事务 TCC @TwoPhaseBusinessAction 放到接口的实现类上是否有效？

没有效果，不能放到实现类上，必须放在接口上。

```
   public interface UserFacade{

        @TwoPhaseBusinessAction(name ="sendMessage", commitMethod="commit", rollbackMethod
="cancel")
        public boolean sendMessage(final BusinessActionContext businessActionContext,@Busin
essActionContextParameter(paramName ="userId")String userId);

        public boolean commit(final BusinessActionContext businessActionContext);

        public boolean cancel(final BusinessActionContext businessActionContext);

    }
```

分布式事务二阶段请求未执行

问题现象

使用时出现报错：“com.alipay.dtx.common.exception.DtxException:DTX-032:TCC prepare menthod must have a parameter who's type isBusinessActionContext”。

启动时出现报错：“DTX-302:initdtx-servers' tcp long-conections failed”。

问题原因

- dtxserver 未发起二阶段请求。
- 客户端收到二阶段请求但执行失败。

解决方案

针对以上两种情况有两种解决方案：

- dtxserver 未发起二阶段请求

  登录 dtxserver 机器，在 `/home/admin/logs/dtxserver/dtx-remote.log` 日志中搜索事务 ID，查找该异常事务相关信息。

  - 若未找到该异常事务的恢复日志，说明该事务的二阶段任务未开始执行，可能是未正常执行或出现事务积压，需重新发起事务。
  - 若找到该异常事务的恢复日志，说明执行过程出错，在 `dtx-remote.log` 和 `commen-error.log` 中查看出错原因，在改进后重新发起任务。

- 客户端收到二阶段请求但执行失败

  i. 查询客户端是否收到二阶段的请求。

在客户端 `/home/admin/logs/dtx/dtx-rpc.log` 日志中搜索事务 ID，若找到“BranchCommitRequest”或“BranchrollbackRequest”的消息，说明客户端收到了二阶段请求。

ii. 查看二阶段请求是否成功执行。

在 `/home/admin/logs/dtx/dtx-rm.log` 日志中搜索事务 ID，若找到“branch commit result”或“branch rollback result”相关内容，且返回结果为 true ，则表示执行成功；反之，执行失败，根据报错信息排查执行失败的原因，在改进后回滚事务重新执行。

9.5. 其他问题

建表时，是否可以使用 SQL 的关键字作为列的名称？

不可以，建表时列名不能使用 SQL 的关键字，如 DESC、TABLE 等。

DTX 发起方和参与方同属一个应用，能否不使用 RPC？

可以在参与方接口上添加 `@InjvmRemoting` 注解，这样发起方去调用参与方就是 JVM 内调用。而且这种情况下，DTX 服务端来远程调用二阶段方法也不会通过注册中心，而是 DTX 自闭环寻址处理。

以 TCC 模式为例：

```
 // 注解InjvmRemoting用于当发起方与参与方是同一个应用，也就是说参与方不需要被其他应用通过注册中心寻址来进行RPC调用
    @InjvmRemoting
    public interface AcctDepositTccService{

        /**
         * <p>
         * 贷记记账接口 TCC模式
         * </p>
         *
         * <p>
         * 结合账户的余额方向来对账户的资金进行流入 or 流出操作
         * </p>
         *
         * @param accountTransRequest   请求对象参数,详情请见{@link AccountTransRequest}
         * @param businessActionContext dtx框架中请求的上下文信息，框架级别参数，调用者不需传递，由dtx自动注入。
         * @return AccountTransResult 交易处理结果
         */
        @TwoPhaseBusinessAction(name ="creditAction", commitMethod ="commit", rollbackMethod ="rollback", antiSuspend =true)
        public AccountTransResult credit(AccountTransRequest accountTransRequest,@ShardingKeyString accountNo,
                BusinessActionContext businessActionContext);

        /**
         * 二阶段提交
         *
         * @param businessActionContext xts上下文
         * @return TwoPhaseResult#isSuccess() 是否成功，true-成功，false-失败
         */
        public boolean commit(BusinessActionContext businessActionContext);

        /**
         * 二阶段回滚
         *
         * @param businessActionContext xts上下文
         * @return TwoPhaseResult#isSuccess() 是否成功，true-成功，false-失败
         */
        public boolean rollback(BusinessActionContext businessActionContext);

    }
```