



北京科蓝

分布式事务解决方案

DTM

用户手册

北京科蓝软件系统股份有限公司

2022 年 5 月

目录

发展历程	3
产品介绍	3
术语	4
控制台	6
事务信息查询	6
全局锁信息查询	6
事务模式	6
AT 模式	6
前提	7
整体机制	7
TCC 模式	7
空回滚	9
幂等	10
悬挂	11
SAGA 模式	12
适用场景	13
优势	13
缺点	13
允许空补偿	13
防悬挂控制	14
幂等控制	14
缺乏隔离性的应对	14
XA 模式	14
前提	15
配置中心的支持	15
注册中心的支持	15
微服务框架的支持	16
ORM 框架支持	16
数据库的支持	16
AT 模式	16
TCC 模式	16
SAGA 模式	17
XA 模式	17
高可用部署	17
科蓝升级计划	18
附录	18
JDK 版本要求	18
全局事务表状态	19
分支事务状态表	20

发展历程

科蓝分布式解决方案最早可以追溯到 2017 年 NJ 银行互金核心平台建设项目中的 AE 资产交换应用，项目中 AE 是基于阿里云的 DTX(阿里云分布式事务中间件)，AE 作为 TCC 方案中的事务发起方存在，AE 既是事务的发起方，也是资金流转的一个串联方。

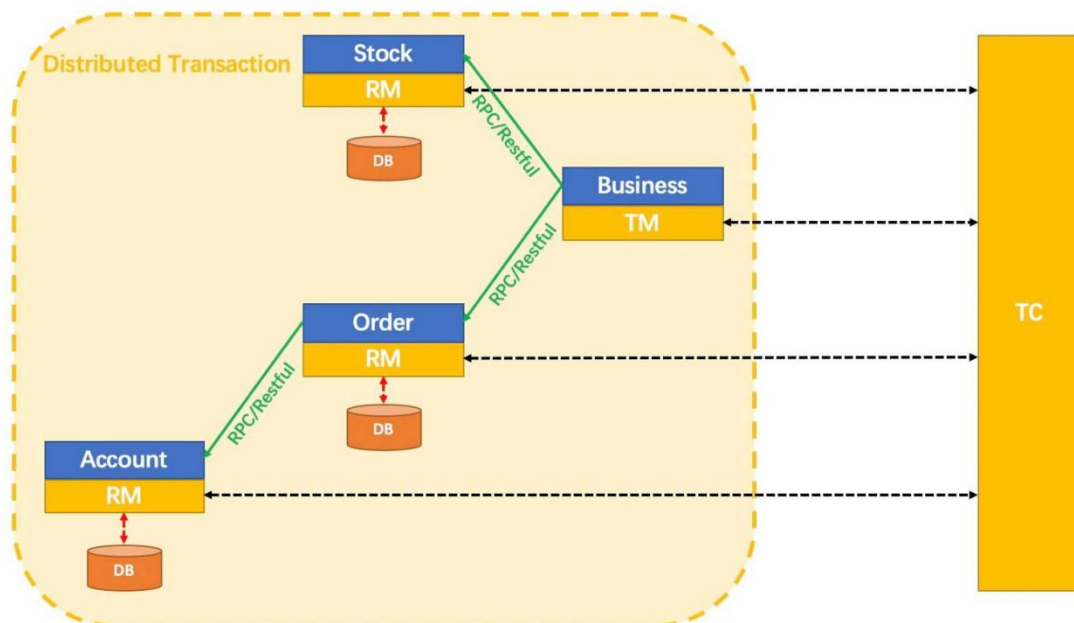
2018 年 WF 银行互金平台建设项目，科蓝公司自研了科蓝分布式事务平台 1.0 版本。该版本基于 AE 应用，与应用结合比较紧密，支持 TCC 模式，因为引用了科蓝 PE 框架，利用桥接模式，可支持任何一种微服务框架及数据库，AE 本身是个微服务应用，天然支持多实例高可用部署。WF 银行互金平台项目中微服务框架为腾讯 TSF 框架，TDSql 分布式数据库。

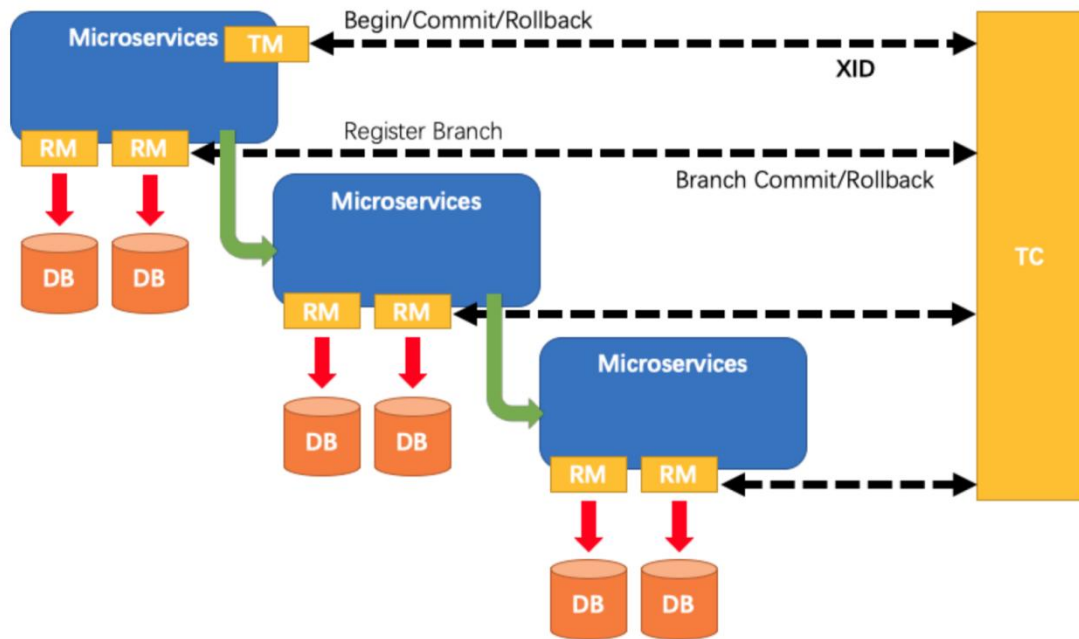
2018 年到 2019 年，科蓝分布式事务平台经过了多个 POC、实施项目的验证及优化。如 TJ 银行核心 POC、ZD 银行用户中心 POC、TJ 银行用户中心项目、JC 银行网贷核算项目、HX 银行开发技术平台 POC。

2020 年，科蓝分布式事务平台结合开源分布式事务平台 Seata，打造了新一版的科蓝分布式事务解决方案平台 2.0。

科蓝分布式事务解决方案平台，致力于为银行等金融机构提供高性能和简单易用的分布式事务服务。该平台提供了 AT、TCC、SAGA 和 XA 事务模式，支持多种常见微服务框架，如 SpringCloud, Dubbo, SofaBoot，支持多种存储方式：文件、Redis、数据库，为客户打造一站式的分布式解决方案。

产品介绍





术语

TC (Transaction Coordinator) - 事务协调者

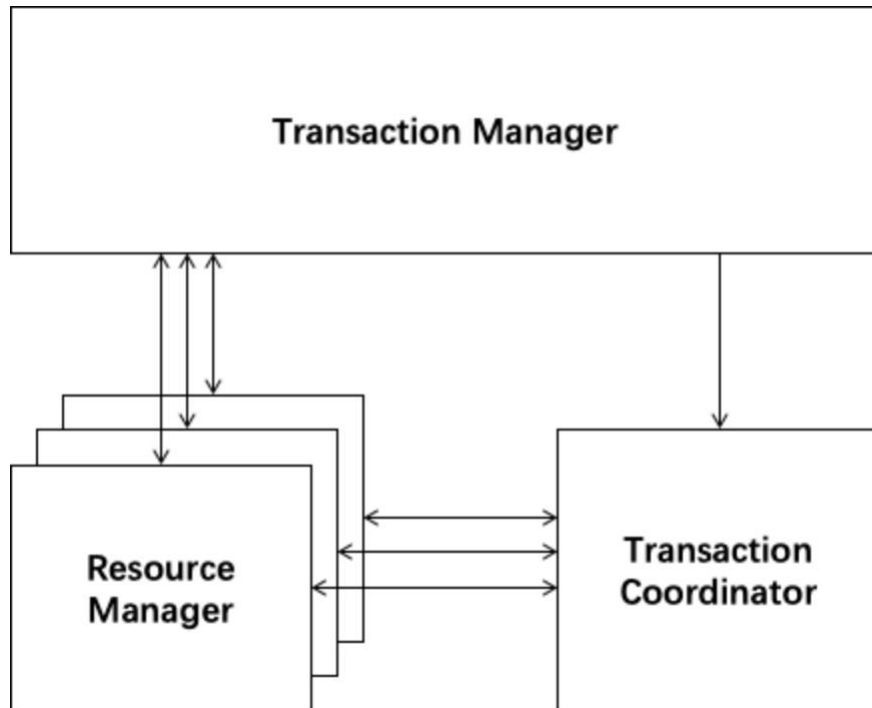
维护全局和分支事务的状态，驱动全局事务提交或回滚。

TM (Transaction Manager) - 事务管理器

定义全局事务的范围：开始全局事务、提交或回滚全局事务。

RM (Resource Manager) - 资源管理器

管理分支事务处理的资源，与 TC 交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。



一个典型的分布式事务过程：

1. TM 向 TC 申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的 XID。
2. XID 在微服务调用链路的上下文中传播。
3. RM 向 TC 注册分支事务，将其纳入 XID 对应全局事务的管辖。
4. TM 向 TC 发起针对 XID 的全局提交或回滚决议。
5. TC 调度 XID 下管辖的全部分支事务完成提交或回滚请求。

特性介绍

1. 支持 AT、TCC、SAGA、XA 模式；
2. 支持注解启动全局事务；
3. 支持接口调用启动全局事务；
4. 服务端支持数据库 MySQL、Oracle、PostgreSQL、TiDB、MariaDB、SunDB、GuassDB。
5. 支持状态机驱动 SAGA 流程；
6. 支持注解方式 SAGA 模式；
7. 支持 TCC&AT、TCC&MQ 混合模式；
8. 客户端 AT、XA 模式支持数据库 MySQL、Oracle、PostgreSQL、TiDB、MariaDB、SunDB、GuassDB；
9. 客户端 TCC、SAGA 模式不限制数据库
10. AT、XA 模式客户端支持 JAVA8，TCC、SAGA 模式客户端事务发起方支

持 JAVA8，事务参与方不限制开发语言及平台。

11. 支持 35 个扩展口，其中服务端 8 个，客户端 27 个；
12. 支持容器化部署、支持虚拟机部署；支持水平扩展及高可用部署；
13. 支持多中心部署；
14. 支持同库、异库、同服务、多服务间分布式事务一致性；
15. 支持多种监控指标的监控

控制台

权限控制，角色管理，用户管理、操作日志审计

事务信息查询

查询流水事务明细信息。

全局锁信息查询

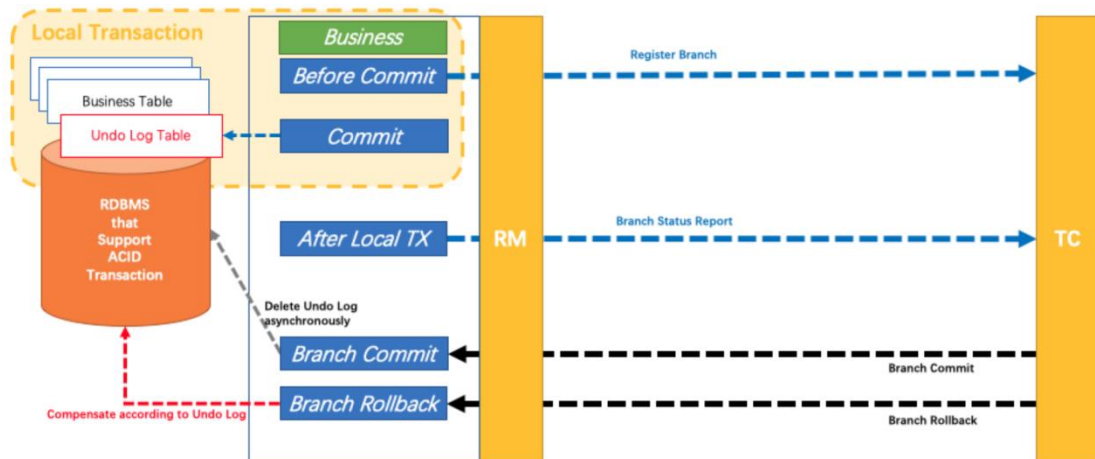
查询全局锁详细信息。

基础指标监控

业务指标监控

事务模式

AT 模式



前提

1. 基于支持本地 ACID 事务的关系型数据库。
2. Java 应用，通过 JDBC 访问数据库。

整体机制

两阶段提交协议的演变：

一阶段

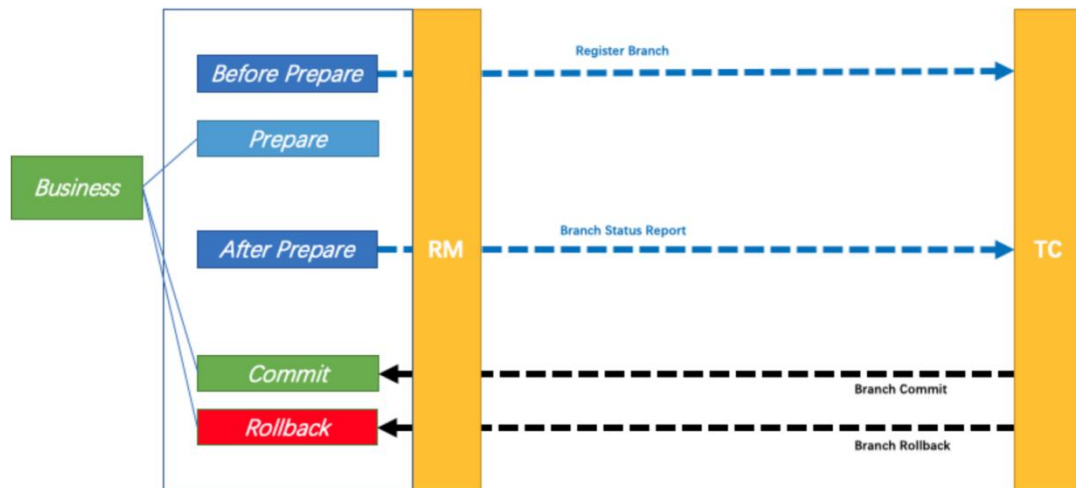
业务数据和回滚日志记录在同一个本地事务中提交，释放本地锁和连接资源。

二阶段：

提交异步化，非常快速地完成。

回滚通过一阶段的回滚日志进行反向补偿。

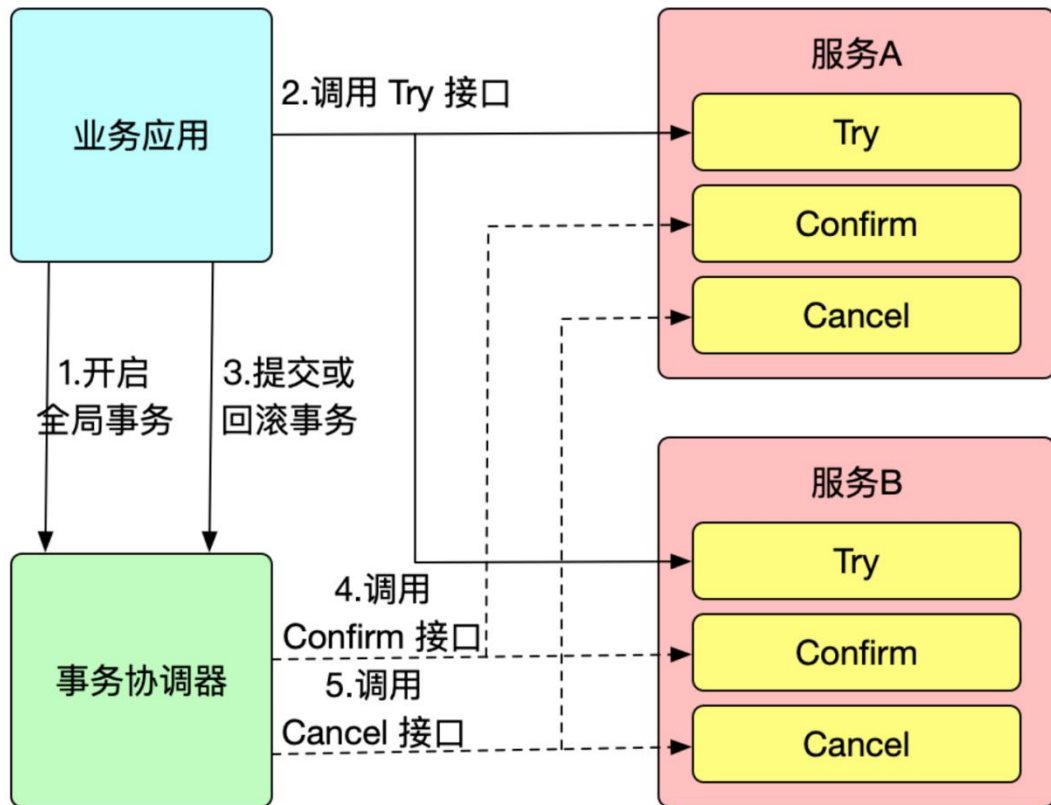
TCC 模式



TCC 是分布式事务中的二阶段提交协议，它的全称为 Try-Confirm-Cancel，即资源预留（Try）、确认操作（Confirm）、取消操作（Cancel），他们的具体含义如下：

1. Try：对业务资源的检查并预留；
2. Confirm：对业务处理进行提交，即 commit 操作，只要 Try 成功，那么该步骤一定成功；
3. Cancel：对业务处理进行取消，即回滚操作，该步骤回对 Try 预留的资源进行释放。

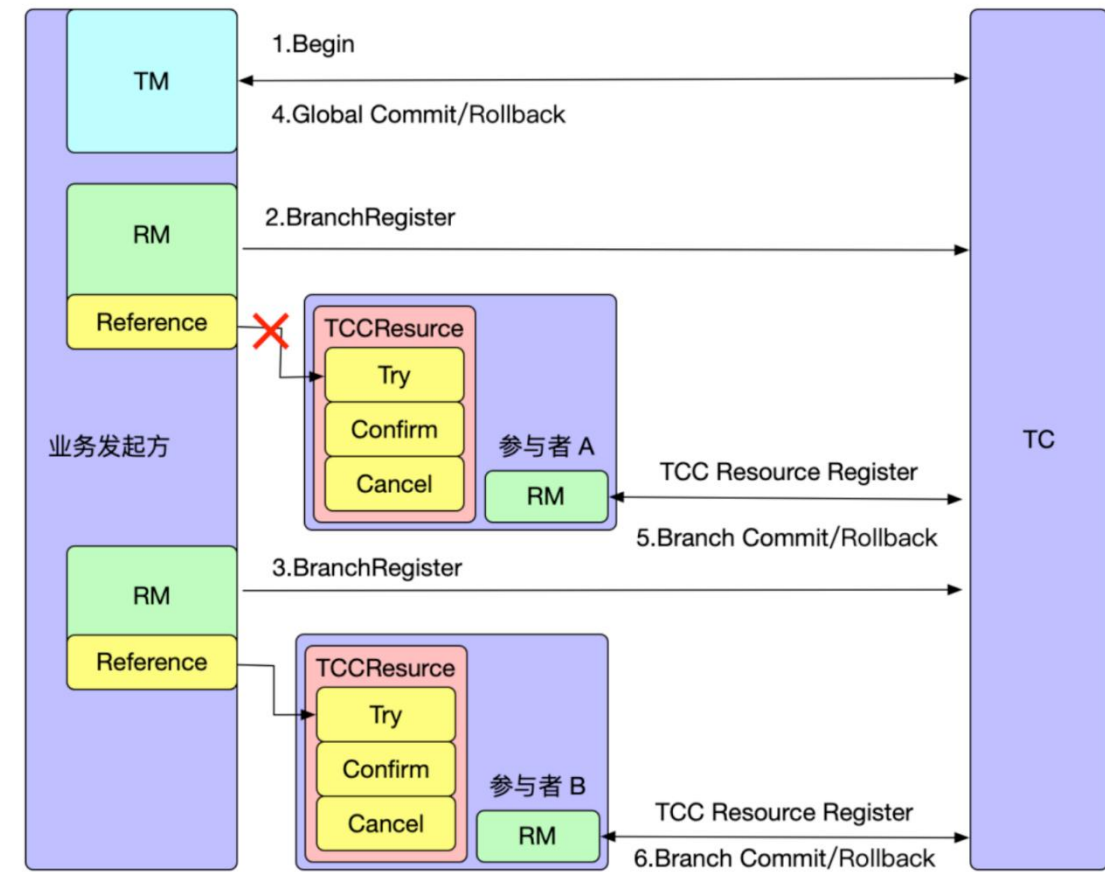
TCC 是一种侵入式的分布式事务解决方案，以上三个操作都需要业务系统自行实现，对业务系统有着非常大的入侵性，设计相对复杂，但优点是 TCC 完全不依赖数据库，能够实现跨数据库、跨应用资源管理，对这些不同数据访问通过侵入式的编码方式实现一个原子操作，更好地解决了在各种复杂业务场景下的分布式事务问题。



空回滚

空回滚指的是在一个分布式事务中，在没有调用参与方的 Try 方法的情况下，TM 驱动二阶段回滚调用了参与方的 Cancel 方法。

那么空回滚是如何产生的呢？



如上图所示，此处以 RPC 举例说明，全局事务开启后，参与者 A 分支注册完成之后会执行参与者一阶段 RPC 方法，如果此时参与者 A 所在的机器发生宕机，网络异常，都会造成 RPC 调用失败，即参与者 A 一阶段方法未成功执行，但是此时全局事务已经开启，整体事务必须要推进到终态，在全局事务回滚时会调用参与者 A 的 Cancel 方法，从而造成空回滚。

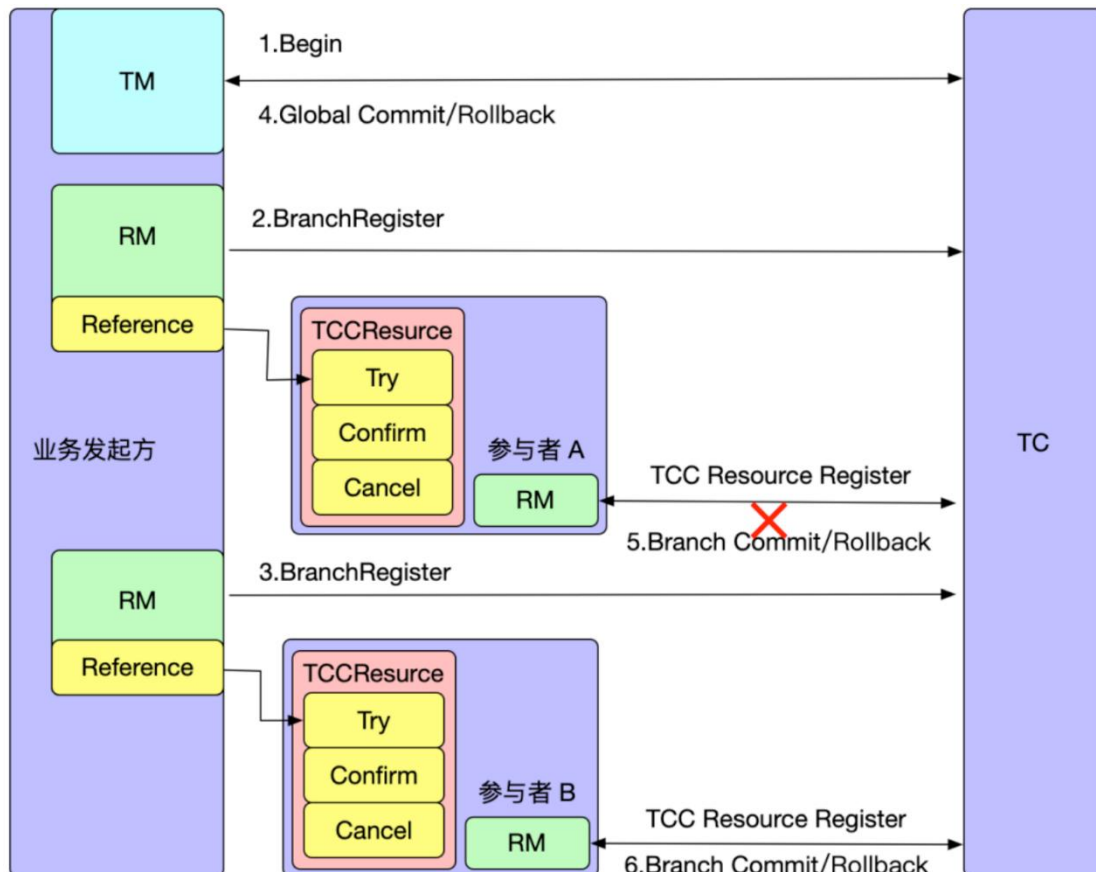
要想防止空回滚，那么必须在 Cancel 方法中识别这是一个空回滚，科蓝分布式事务平台是如何做的呢？

新增一个 TCC 事务控制表，包含事务的 XID 和 BranchID 信息，在 Try 方法执行时插入一条记录，表示一阶段执行了，执行 Cancel 方法时读取这条记录，如果记录不存在，说明 Try 方法没有执行。

幂等

幂等问题指的是 TC 重复进行二阶段提交，因此 Confirm/Cancel 接口需要支持幂等处理，即不会产生资源重复提交或者重复释放。

那么幂等问题是如何产生的呢？



如上图所示，参与者 A 执行完二阶段之后，由于网络抖动或者宕机问题，会造成 TC 收不到参与者 A 执行二阶段的返回结果，TC 会重复发起调用，直到二阶段执行结果成功。

科蓝分布式事务平台是如何处理幂等问题的呢？

同样的也是在 TCC 事务控制表中增加一个记录状态的字段 `status`，该字段有 3 个值，分别为：

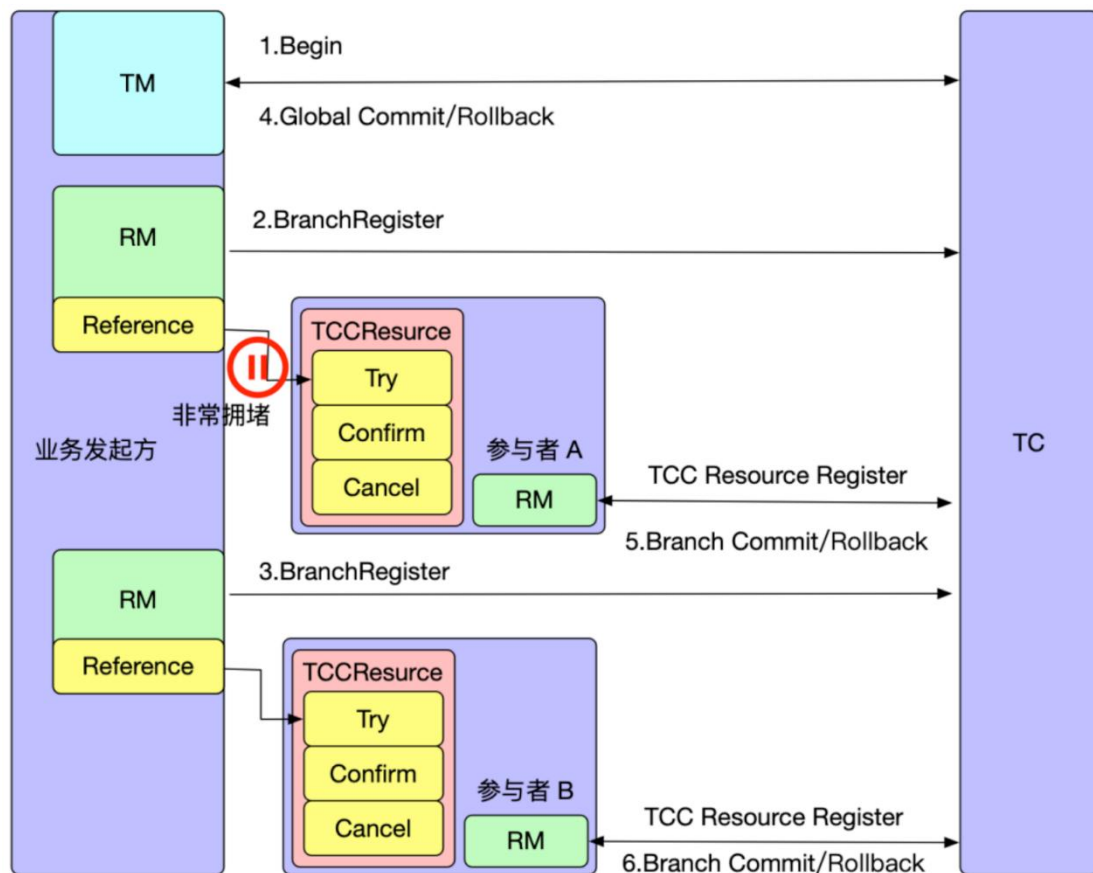
1. `tried: 1`
2. `committed: 2`
3. `rollbacked: 3`

二阶段 `Confirm/Cancel` 方法执行后，将状态改为 `committed` 或 `rollbacked` 状态。当重复调用二阶段 `Confirm/Cancel` 方法时，判断事务状态即可解决幂等问题。

悬挂

悬挂指的是二阶段 `Cancel` 方法比一阶段 `Try` 方法优先执行，由于允许空回滚的原因，在执行完二阶段 `Cancel` 方法之后直接空回滚返回成功，此时全局事务已结束，但是由于 `Try` 方法随后执行，这就会造成一阶段 `Try` 方法预留的资源永远无法提交和释放了。

那么悬挂是如何产生的呢？



如上图所示，在执行参与者 A 的一阶段 Try 方法时，出现网路拥堵，由于科蓝分布式事务平台全局事务有超时限制，执行 Try 方法超时后，TM 决议全局回滚，回滚完成后如果此时 RPC 请求才到达参与者 A，执行 Try 方法进行资源预留，从而造成悬挂。

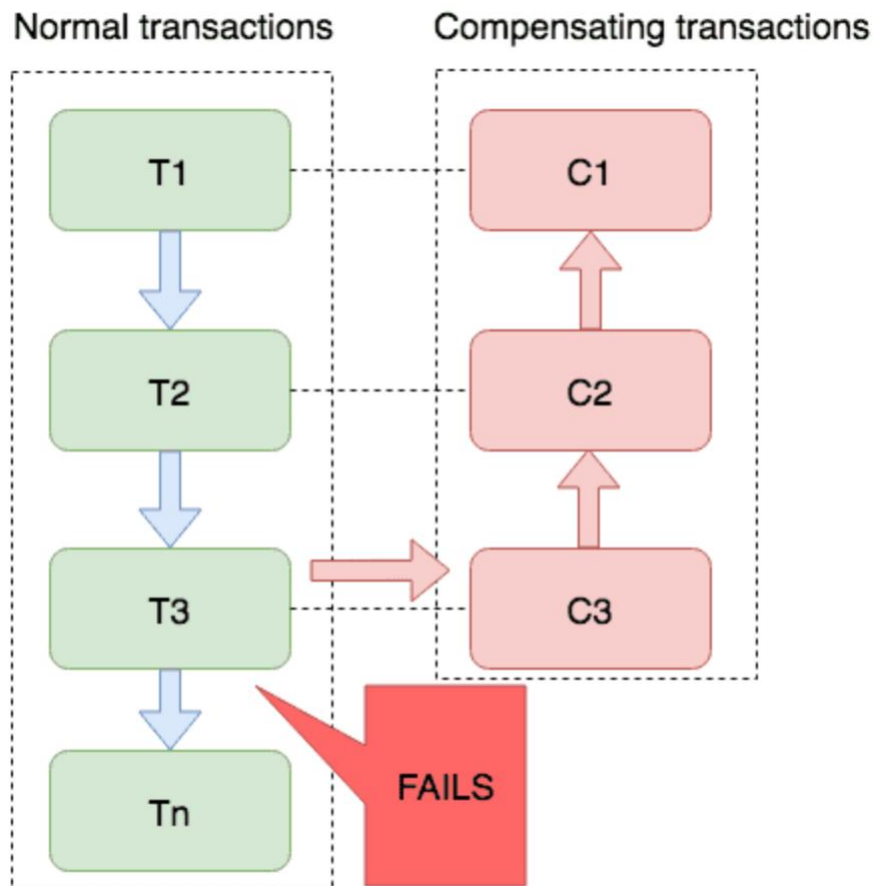
科蓝分布式事务平台是怎么处理悬挂的呢？

在 TCC 事务控制表记录状态的字段 status 中增加一个状态：suspended:
4

当执行二阶段 Cancel 方法时，如果发现 TCC 事务控制表有相关记录，说明二阶段 Cancel 方法优先一阶段 Try 方法执行，因此插入一条 status=4 状态的记录，当一阶段 Try 方法后面执行时，判断 status=4，则说明有二阶段 Cancel 已执行，并返回 false 以阻止一阶段 Try 方法执行成功。

SAGA 模式

Saga 模式是科蓝分布式事务平台提供的长事务解决方案，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，一阶段正向服务和二阶段补偿服务都由业务开发实现。



适用场景

1. 业务流程长、业务流程多
2. 参与者包含其它公司或遗留系统服务, 无法提供 TCC 模式要求的三个接口

优势

1. 一阶段提交本地事务, 无锁, 高性能
2. 事件驱动架构, 参与者可异步执行, 高吞吐
3. 补偿服务易于实现

缺点

不保证隔离性 (应对方案见后面文档)

允许空补偿

空补偿：原服务未执行，补偿服务执行了

出现原因：

1. 原服务 超时（丢包）；
2. Saga 事务触发 回滚；
3. 未收到 原服务请求，先收到补偿请求。

所以服务设计时需要允许空补偿，即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。

防悬挂控制

悬挂：补偿服务比原服务先执行

出现原因：

1. 原服务超时（拥堵）；
2. Saga 事务回滚，触发回滚；
3. 拥堵的原服务到达。

所以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，如果存在则要拒绝服务的执行。

幂等控制

原服务与补偿服务都需要保证幂等性，由于网络可能超时，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。

缺乏隔离性的应对

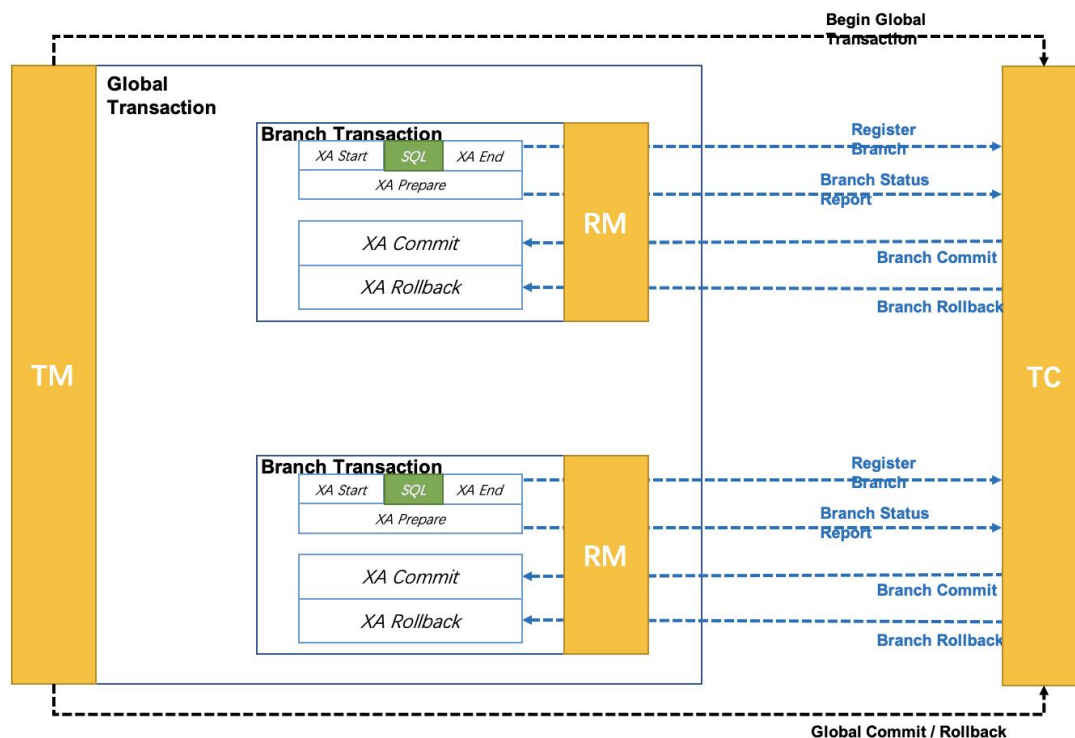
由于 Saga 事务不保证隔离性，在极端情况下可能由于脏写无法完成回滚操作，比如举一个极端的例子，分布式事务内先给用户 A 充值，然后给用户 B 扣减余额，如果在给 A 用户充值成功，在事务提交以前，A 用户把余额消费掉了，如果事务发生回滚，这时则没有办法进行补偿了。这就是缺乏隔离性造成的典型的问题，实践中一般的应对方法是：

业务流程设计时遵循“宁可长款，不可短款”的原则，长款意思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。

XA 模式

在科蓝分布式事务平台定义的分布式事务框架内，利用事务资源（数据库、消息服务等）对 XA 协议的支持，以 XA 协议的机制来管理分支事务的一种事务模式。



前提

1. 支持 XA 事务的数据库。
2. Java 应用，通过 JDBC 访问数据库。

配置中心的支持

配置中心可以说是一个“大货仓”，内部放置着各种配置文件，你可以通过自己所需进行获取配置加载到对应的客户端。比如 Seata Client 端 (TM, RM), Seata Server (TC), 会去读取全局事务开关, 事务会话存储模式等信息。

现支持的配置中心有：Nacos、Appllo、Etcd3、Consul、Zookeeper、File、ServiceComb-kie。

注：这里的配置中心支持是指分布式事务应用端与分布式事务服务端的配置中心，与应用的配置中心无关。

注册中心的支持

注册中心可以说是微服务架构中的”通讯录“，它记录了服务和服务地址的映射关系。在分布式架构中，服务会注册到这里，当服务需要调用其它服务时，就到这里找到服务的地址，进行调用。比如 Seata Client 端(TM, RM), 发现 Seata Server(TC) 集群的地址, 彼此通信。

现支持的注册中心有：Nacos、Eureka、Etcd3、Consul、Zookeeper、ServiceComb、File。

注：这里的注册中心是指分布式事务应用端与分布式事务服务端的注册中心，与应用的注册中心无关。

微服务框架的支持

1. AT 模式支持 Dubbo、Spring Cloud、Motan、gRPC 和 sofa-RPC。
2. TCC 模式支持 Dubbo、Spring Cloud 和 sofa-RPC。注：TCC 的 SpringCloud 框架的支持也是依赖 TCC 的本地模式实现的，理论上 TCC 模式是不限制微服务框架的。

ORM 框架支持

科蓝分布式事务解决方案平台虽然是保证数据一致性的组件，但对于 ORM 框架并没有特殊的要求，像主流的 Mybatis, Mybatis-Plus, Spring Data JPA, Hibernate 等都支持。这是因为 ORM 框架位于 JDBC 结构的上层，而科蓝分布式事务解决方案平台的 AT, XA 事务模式是对 JDBC 标准接口操作的拦截和增强。

数据库的支持

AT 模式

AT 模式支持的数据库有：MySQL、Oracle、PostgreSQL、TiDB、MariaDB。

TCC 模式

TCC 模式因为实现了防悬挂措施，需要数据源支持。如果客户端直接实现防悬挂，则不依赖数据源。

SAGA 模式

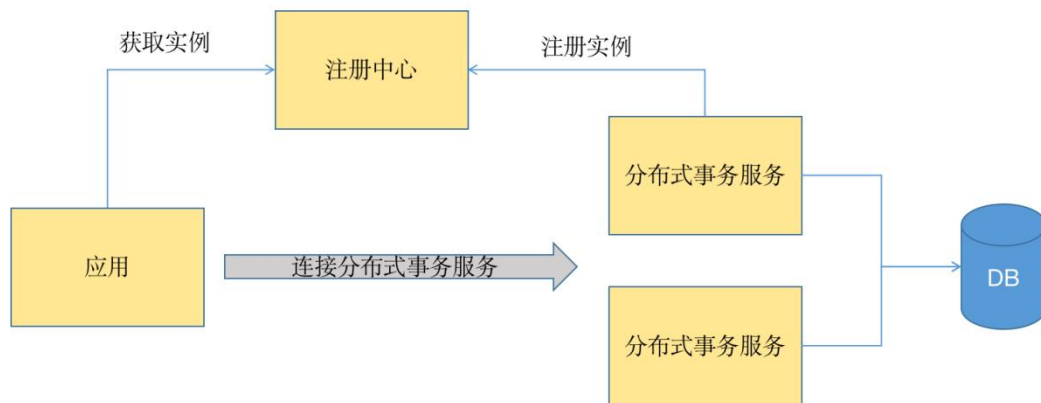
Saga 模式不依赖数据源。

XA 模式

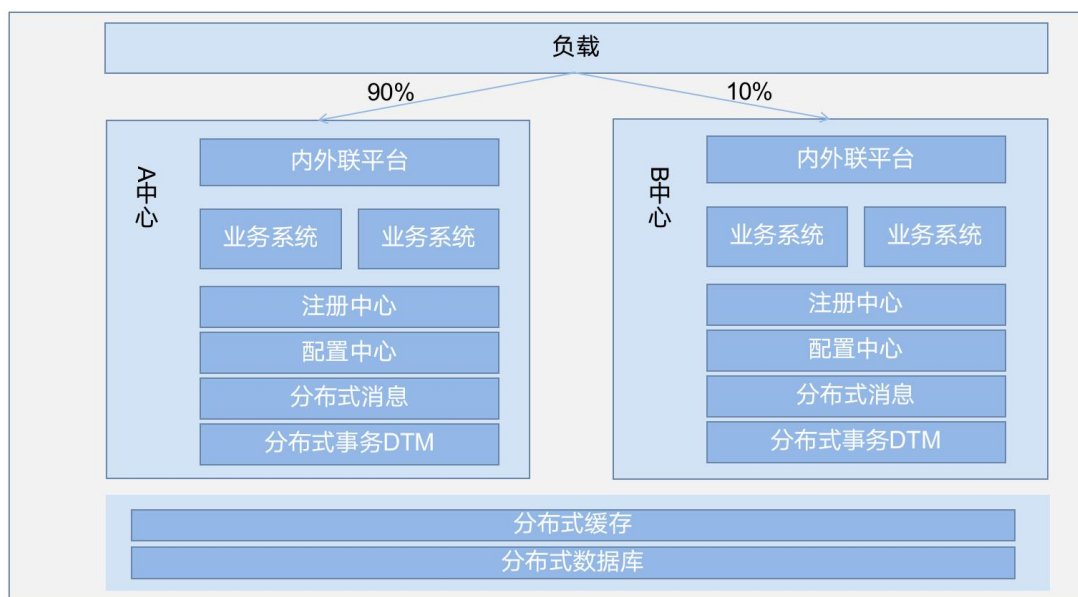
XA 模式只支持实现了 XA 协议的数据库。支持 MySQL、Oracle、PostgreSQL 和 MariaDB。

高可用部署

科蓝分布式事务解决方案平台的高可用依赖于注册中心、配置中心和数据库来实现，支持水平扩展



多中心部署方案



如上图，最下游是数据中心，如分布式数据、传统关系型数据、分布式缓存数据库等，位于整个服务架构的最底层，多个中心共享数据中心，如果考虑异地灾备中心，还需要再考虑数据库异地同步方案。

DTM 及各个中间件可以部署在各个中心的 ECS 虚拟机或者容器部署。

dtm-server 是个无状态的微服务，高可用依赖于注册中心。各个中心的 DTM 服务都是以多实例形式部署，在各中心内部达到高可用标准。

其他中间件如配置中心，分布式消息也是高可用部署的，与 DTM 不是强依赖，增加这些可以增加 DTM 的功能。

因为注册中心是各中心独立部署的，各中心之间的注册信息是不同步或者共享的，所以各中心部署的业务系统只能使用各中心的 DTM 服务。若是某个中心的 DTM 实例全部挂了，因为 DTM 的事务信息及状态信息是存在各个中心共享的数据库中的，所以其他中心的 DTM 服务会继续完成后面的工作。

科蓝升级计划

附录

JDK 版本要求

支持的 JDK 版本为 JDK8。

全局事务表状态

状态	代码	备注
全局事务开始 (Begin)	1	此状态可以接受新的分支事务注册
全局事务提交中 (Committing)	2	这个状态会随时改变
全局事务提交重试 (CommitRetry)	3	在提交异常被解决后尝试重试提交
全局事务回滚中 (Rollbacking)	4	正在重新回滚全局事务
全局事务回滚重试中 (RollbackRetrying)	5	在全局回滚异常被解决后尝试事务重试回滚中
全局事务超时回滚中 (TimeoutRollbacking)	6	全局事务超时回滚中
全局事务超时回滚重试中 (TimeoutRollbackRetrying)	7	全局事务超时回滚重试中
异步提交中 (AsyncCommitting)	8	异步提交中
二阶段已提交 (Committed)	9	二阶段已提交，此状态后全局事务状态不会再改变
二阶段提交失败 (CommitFailed)	10	二阶段提交失败
二阶段决议全局回滚 (Rollbacked)	11	二阶段决议全局回滚
二阶段全局回滚失败 (RollbackFailed)	12	二阶段全局回滚失败
二阶段超时回滚	13	二阶段超时回滚

状态	代码	备注
(TimeoutRollbacked)		
二阶段超时回滚失败 (TimeoutRollbackFailed)	14	二阶段超时回滚失败
全局事务结束 (Finished)	15	全局事务结束
二阶段提交超时 (CommitRetryTimeout)	16	二阶段提交因超过重试时间限制导致失败
二阶段回滚超时 (RollbackRetryTimeout)	17	二阶段回滚因超过重试时间限制导致失败
未知状态 (UnKnown)	0	未知状态

分支事务状态表

状态	代码	备注
分支事务注册 (Registered)	1	向 TC 注册分支事务
分支事务一阶段完成 (PhaseOne_Done)	2	分支事务一阶段业务逻辑完成
分支事务一阶段失败 (PhaseOne_Failed)	3	分支事务一阶段业务逻辑失败
分支事务一阶段超时 (PhaseOne_Timeout)	4	分支事务一阶段处理超时
分支事务二阶段已提交 (PhaseTwo_Committed)	5	分支事务二阶段提交
分支事务二阶段提交失败重试 (PhaseTwo_CommitFailed_Retr)	6	分支事务二阶段提交失败重试

yable)		
分支事务二阶段提交失败不重试 (PhaseTwo_CommitFailed_Unretryable)	7	分支事务二阶段提交失败不重试
分支事务二阶段已回滚 (PhaseTwo_Rollbacked)	8	分支事务二阶段已回滚
分支事务二阶段回滚失败重试 (PhaseTwo_RollbackFailed_Retryable)	9	分支事务二阶段回滚失败重试
分支事务二阶段回滚失败不重试 (PhaseTwo_RollbackFailed_Unretryable)	10	二阶段提交失败
未知状态 (UnKnown)	0	未知状态