NX 奈学教育

分布式事务框架Seata深入剖析与应用实践
分布式事务剖析篇

主讲人：陈东

2020.7.14

目录

NX 奈学教育

- 分布式场景下数据一致性问题分析
- 分布式事务解决方案深入对比剖析
- 分布式事务框架Seata实现原理剖析
- Seata AT模式设计与实现原理剖析

NX 奈学教育

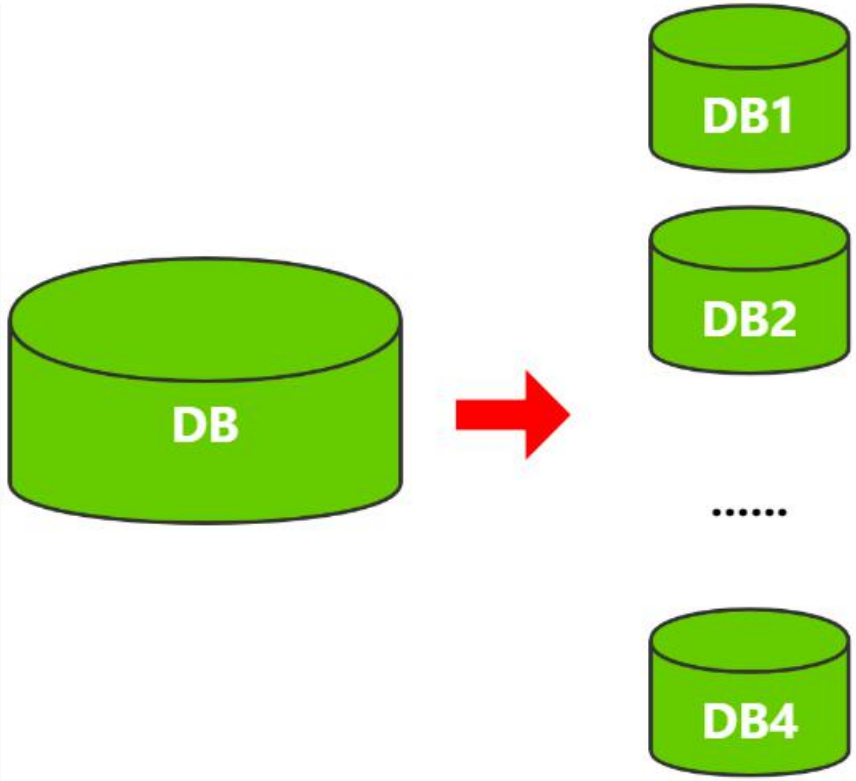
01.分布式场景下数据一致性问题分析

分布式场景下数据一致性问题分析

奈学教育

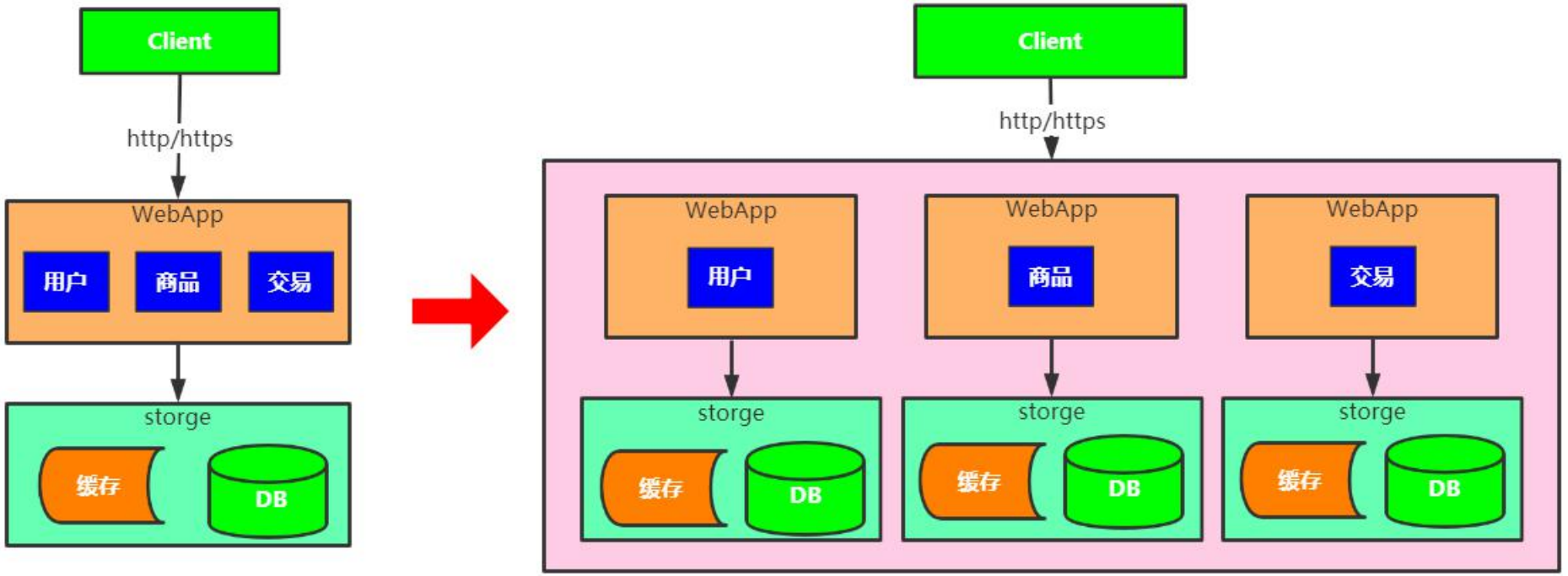
分布式事务背景

- 关系库常用扩展方式
  - **水平分库**
  - 垂直分库



分布式事务背景

- 关系库常用扩展方式
  - 水平分库
  - **垂直分库**



分布式场景下数据一致性问题分析

分布式事务背景

- 跨数据库分布式事务
  - 数据库的物理分割下保障跨库操作的ACID。
- 跨服务分布式事务
  - 服务的网络分割下保障多服务的事务完整性。
- 混合式分布式事务
  - 跨数据库分布式事务 + 跨服务分布式事务。

核心问题：事务参与者出现在不同的数据库实例，需要网络通讯进行交互。

NX 奈学教育

02.分布式事务解决方案深入对比剖析

分布式事务解决方案深入对比剖析

NX 奈学教育

分布式事务分类

- 刚性事务
  - XA
  - 2PC
  - 3PC
- 柔型事务
  - TCC
  - Saga
  - 事务消息
  - 最大努力通知事务

	刚性事务	柔性事务
业务改造	无	有
一致性	强一致	最终一致
隔离性	原生支持	实现资源锁定接口
并发性能	严重衰退	略微衰退
适合场景	短事务 并发较低	长事务 高并发

分布式事务解决方案深入对比剖析

奈学教育

事务模型对比

- 一致性保障：XA > TCC = SAGA > 事务消息；
- 业务友好性：XA > 事务消息 > SAGA > TCC;
- 性能损耗：XA > TCC = SAGA = 事务消息

03.分布式事务框架Seata实现原理剖析

Seata介绍

- Simple Extensible Autonomous Transaction Architecture，简单可扩展自治事务框架
- 事务模型
  - AT 模式：原始支持，早期还有MT模式(0.4.2废弃)
  - TCC 模式：0.4版本支持，19.03.19
  - Saga 模式：0.9版本支持，19.10.16
  - XA 模式：1.2版本支持，20.04.21
- 愿景
  - 像使用本地事务一样使用分布式事务，提供一站式的分布式事务解决方案

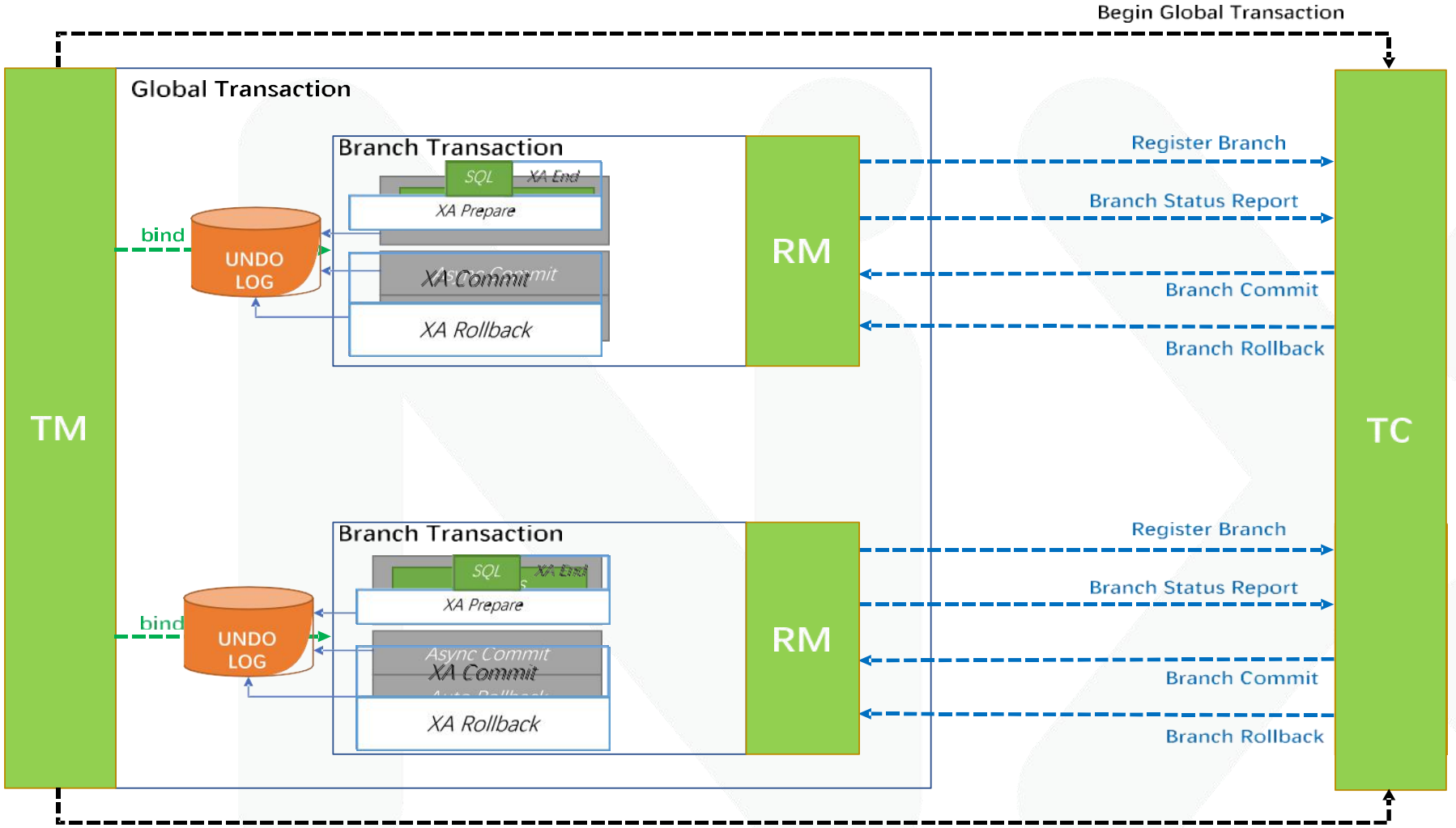
Seata特性

- 支持多个微服务框架
  - Dubbo、Spring Cloud、Sofa-RPC、Motan 和 gRPC 等RPC框架
- 高可用：
  - 支持基于数据库存储的集群模式，水平扩展能力强
- 高可扩展性
  - 支持配置中心、注册中心、SPI扩展

分布式事务框架Seata实现原理剖析

奈学教育

分布式事务标准化

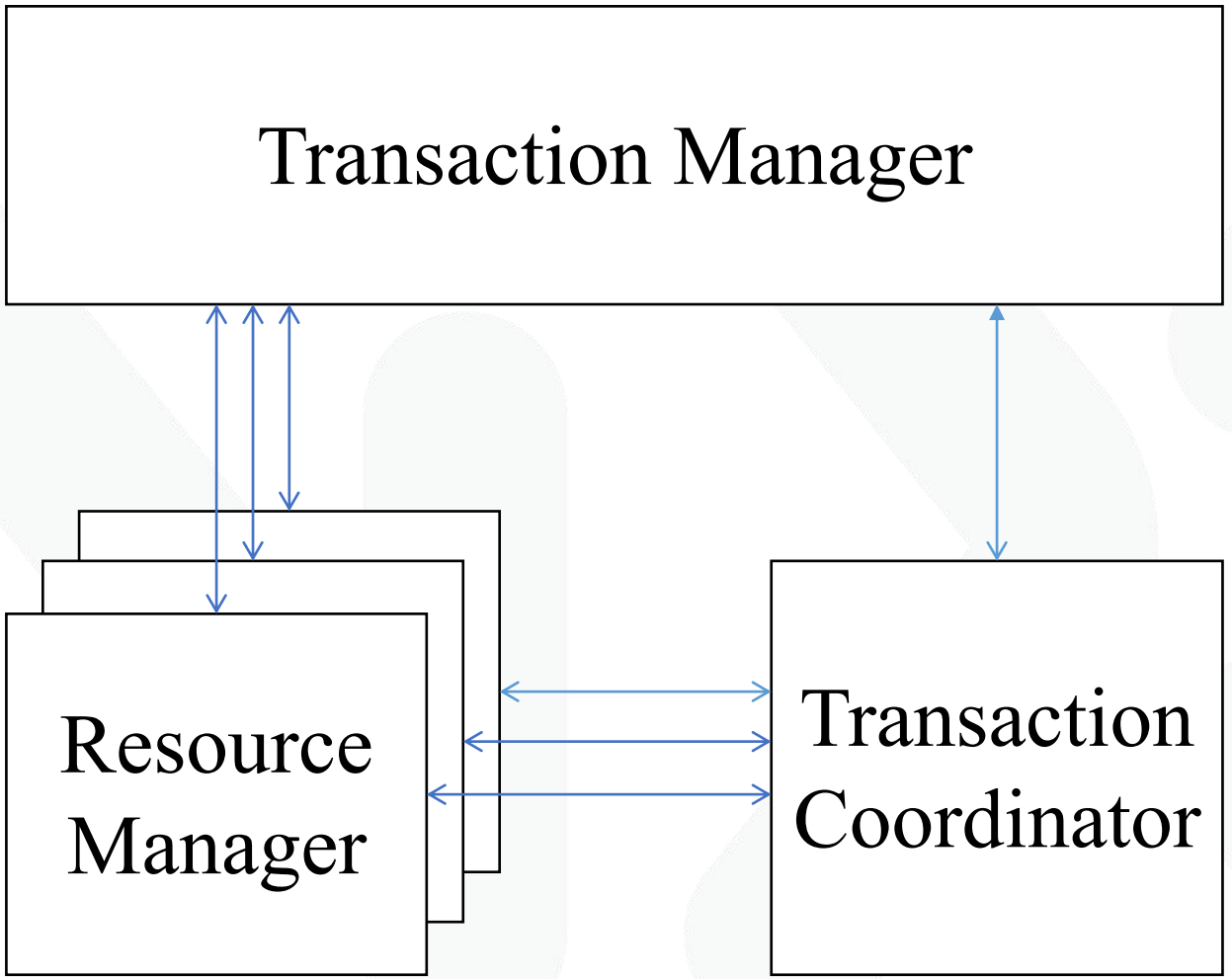


组成模块

- Transaction Coordinator (TC)
  - 事务协调器，维护全局事务的运行状态，负责协调并驱动全局事务的提交或回滚；
- Transaction Manager (TM)
  - 控制全局事务的边界，负责开启一个全局事务，并最终发起全局提交或全局回滚的决议；
- Resource Manager (RM)
  - 控制分支事务，负责分支注册、状态汇报，并接收事务协调器的指令，驱动分支（本地）事务的提交和回滚；

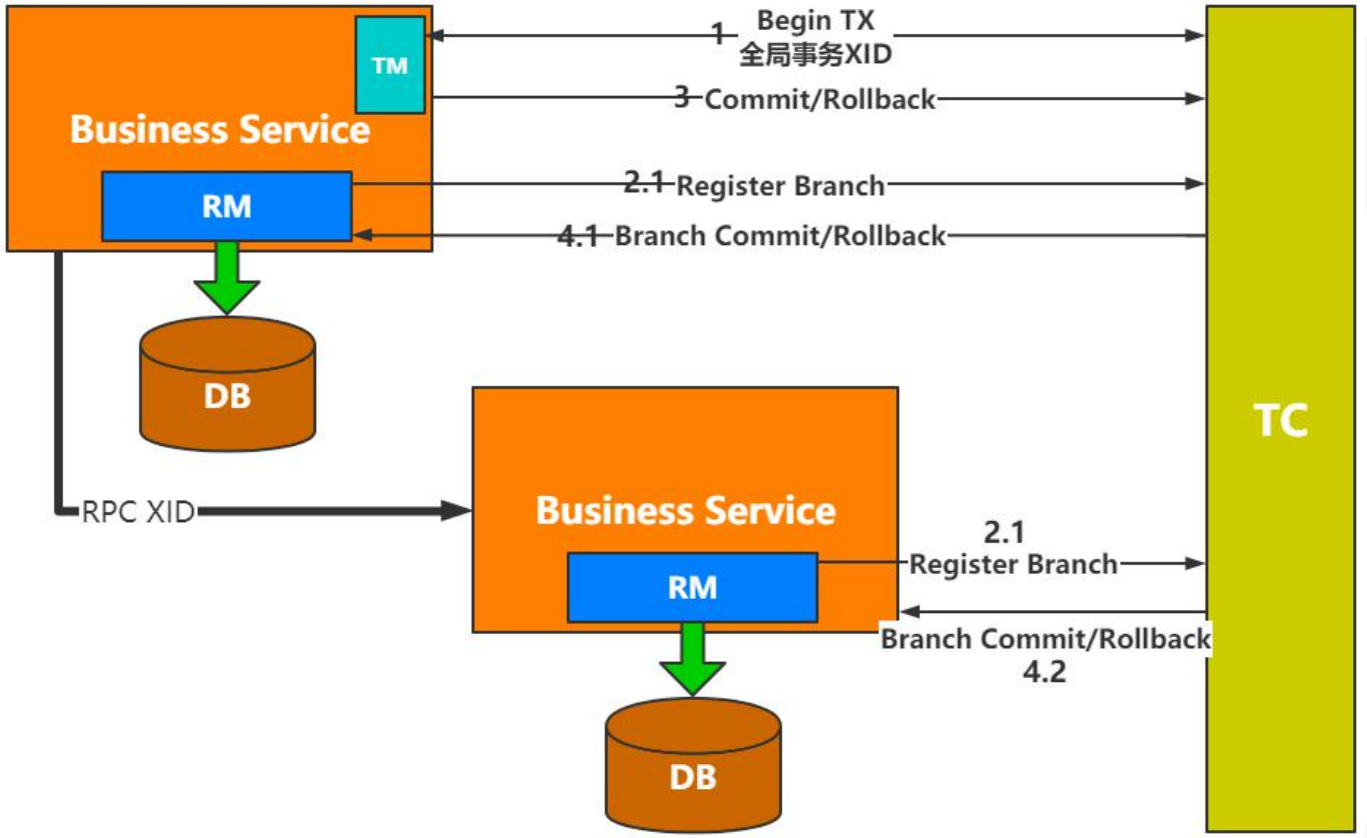
组成模块

- Transaction Coordinator (TC)
- Transaction Manager (TM)
- Resource Manager (RM)



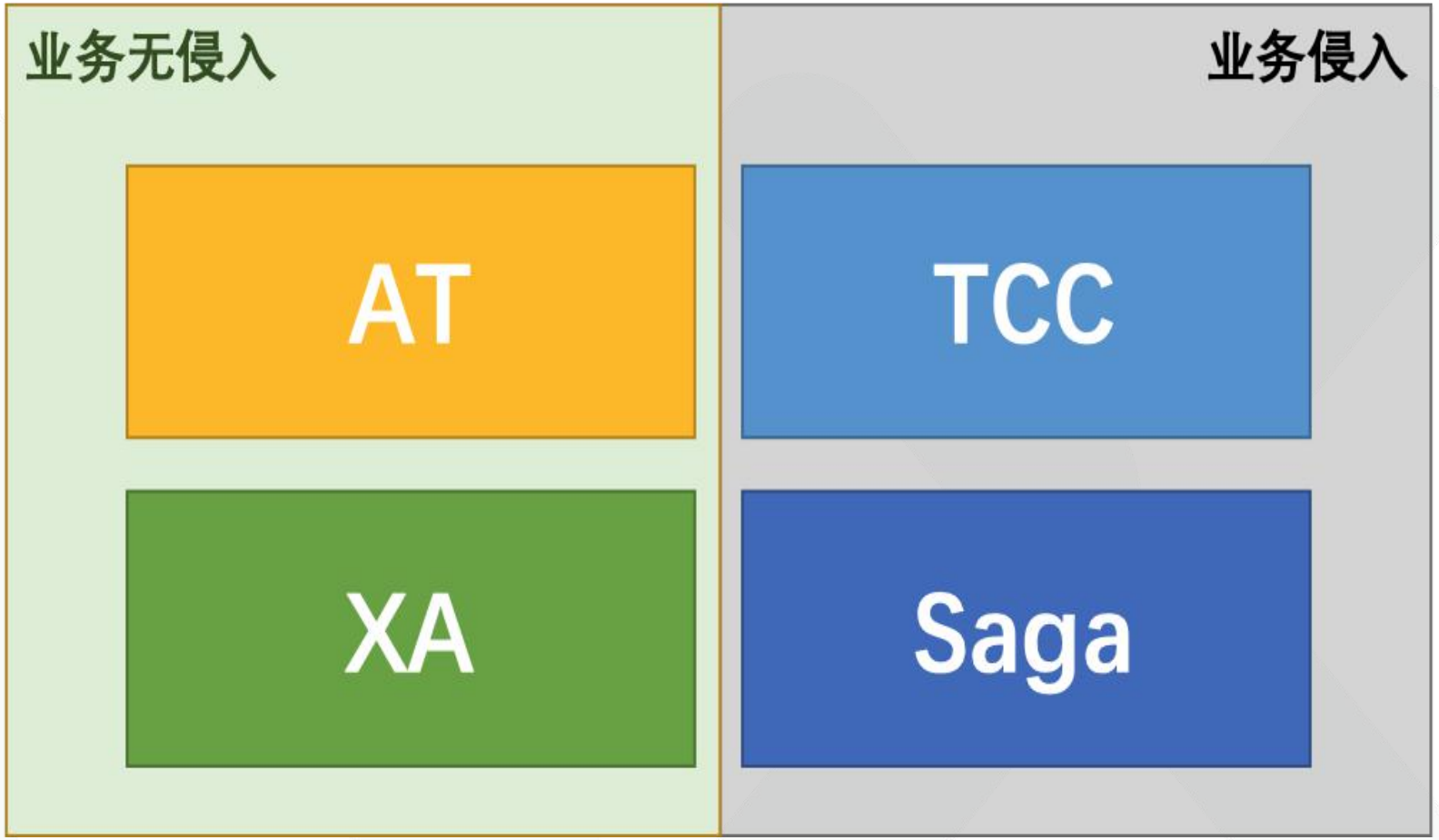
全局事务

- 定义
  - 若干分支事务的整体协调
- 事务流程
  - TM向TC 申请开启一个全局事务，TC创建全局事务后返回全局唯一的XID，XID会在全局事务的上下文中传播；
  - RM向TC注册分支事务，该分支事务归属于拥有相同XID的全局事务；
  - TM向TC发起全局提交或回滚；
  - TC调度XID下的分支事务完成提交或者回滚。



事务模式能力边界

- 条件和限制
  - 数据库：必须支持本地事务
  - 数据表：必须定义主键
  - 难以实现更高的隔离级别



NX 奈学教育

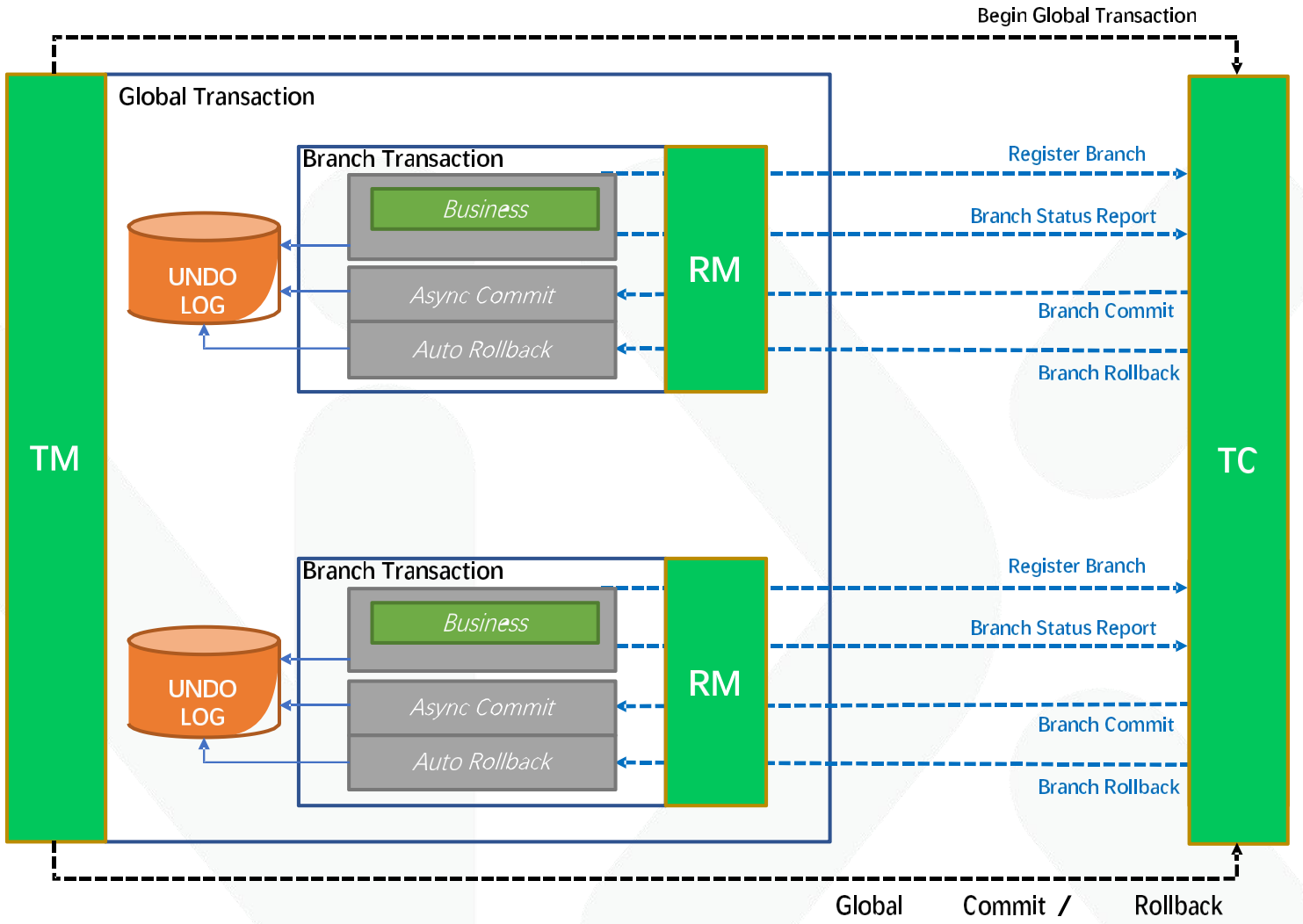
04. Seata AT模式设计与实现原理剖析

AT模式

- 介绍
  - 一种无侵入的分布式事务解决方案，2PC的广义实现。
  - 源自阿里云GTS AT模式的开源版。
- 核心价值
  - 低成本：编程模型不变，轻依赖不需要为分布式事务场景做特定设计。
  - 高性能：一阶段提交，不阻塞；连接释放，保证整个系统的吞吐。
  - 高可用：极端的异常情况下，可以暂时 跳过异常事务，保证系统可用。

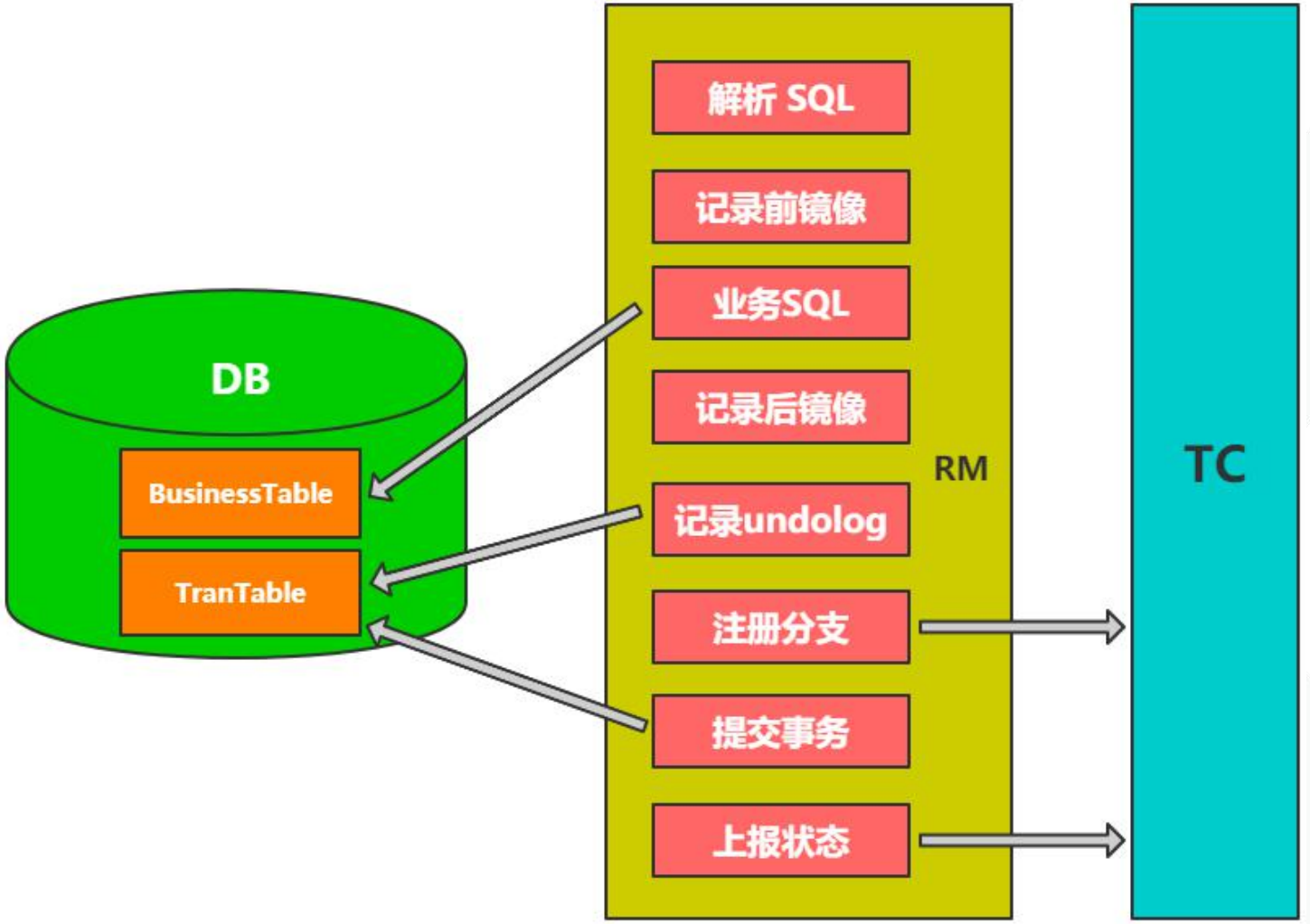
AT模式核心设计

- 一阶段：
  - 拦截“业务 SQL”；
  - 生成前镜像
  - 生成后镜像
- 二阶段
  - TC向所有RM发起提交/回滚；



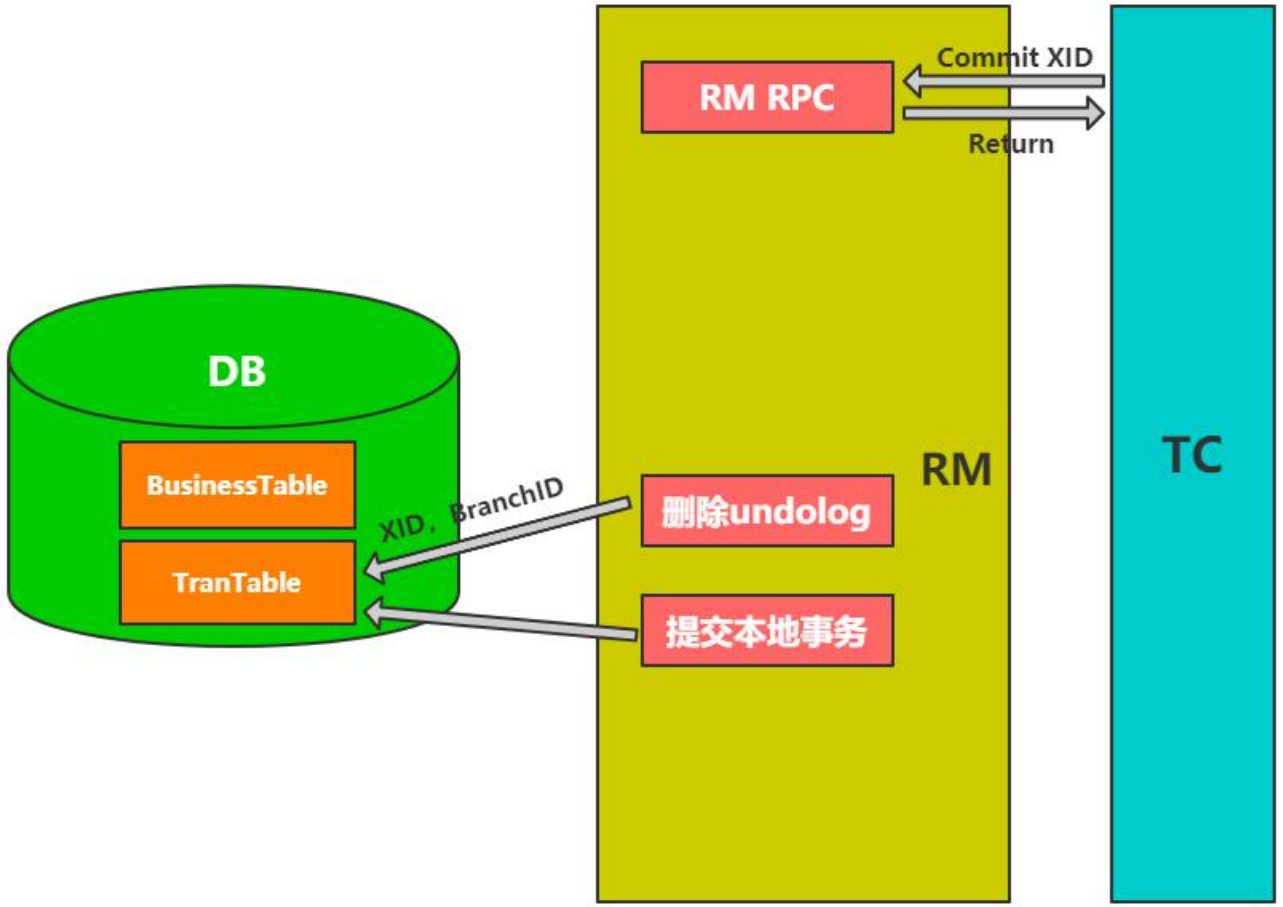
AT执行阶段

- 拦截“业务 SQL”
  - 解析SQL语义
  - 提取表元数据
  - 保存原镜像
  - 执行业务SQL
  - 保存新镜像
- 利用本地事务保证ACID



AT完成阶段-提交

- 清理快照数据



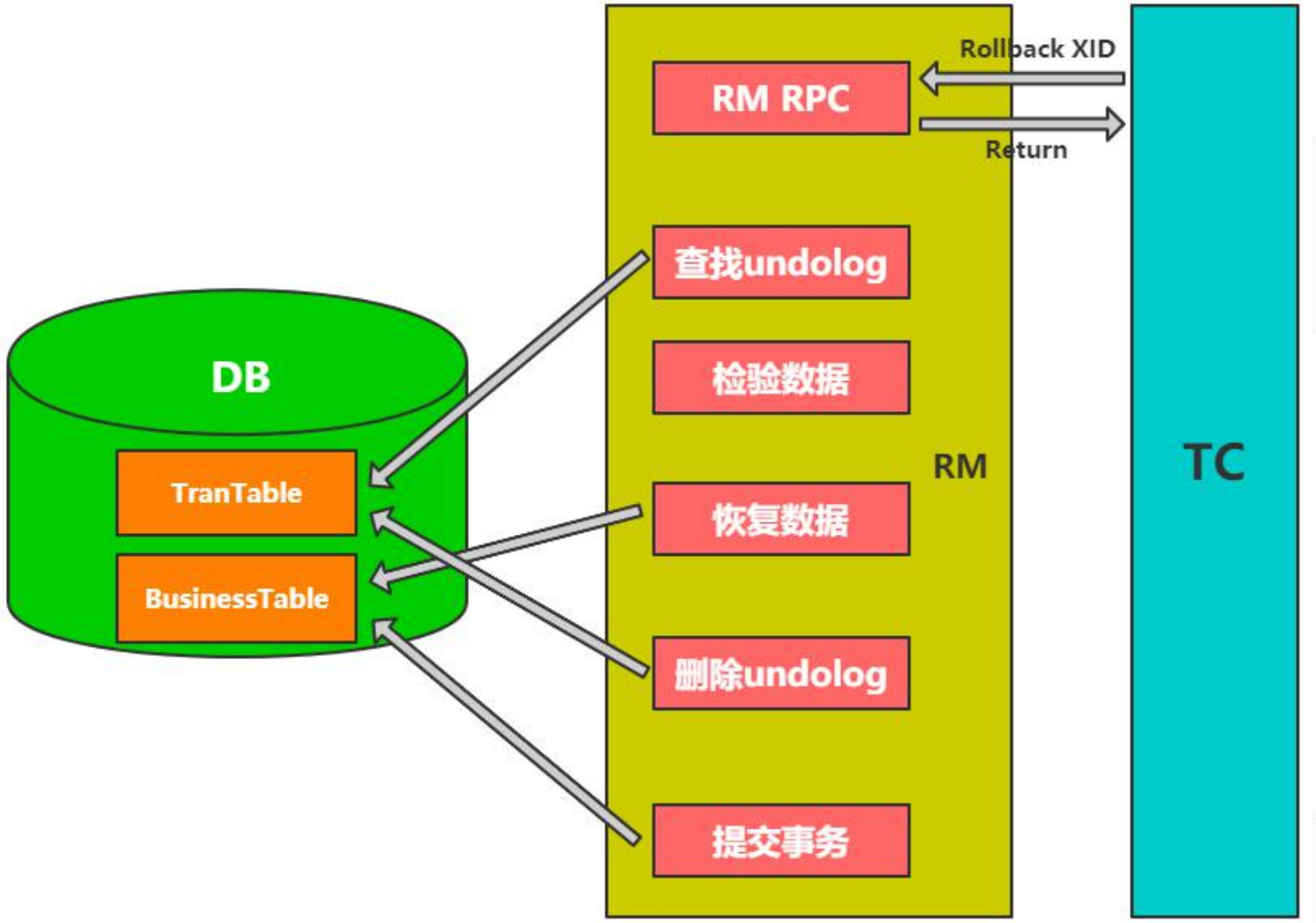
思考

- 分支事务已经提交，如何回滚；
- 其他事务对数据的修改；
  - 其他分布式事务的修改；
  - 非分布式事务的修改；

“行锁”

AT完成阶段-回滚

- 校验脏写
  - 新镜像 VS 当前数据库数据
- 还原数据
  - 根据前后镜像，生成逆向SQL
- 删除中间数据
  - 删除前后镜像



AT前后镜像

- 商品库存操作
  - 用户下单减库存

Pid	count
100	10

UPDATE product SET count = count -1 where pid = **100**;

SELECT * FROM product where pid = **100**;

Pid	count
100	10

获取前镜像

UPDATE product SET count = count -1 where pid = **100**;

SELECT * FROM product where pid = **100**;

Pid	count
100	9

获取后镜像

AT前后镜像

- 商品库存操作
  - 用户下单减库存

Pid	count
100	10

Pid	count
100	9

```
{
        "branchId": XXXX,
        "undoItems": [{
                "afterImage": {
                        "rows": [{
                                "fields": [{
                                        "name": "pid",
                                        "type": XX,
                                        "value": 100
                                }, {
                                        "name": "count",
                                        "type": XX,
                                        "value": 10
                                }]
                        }],
                        "tableName": "product"
                },
                "beforeImage": {
                        "rows": [{
                                "fields": [{
                                        "name": "pid",
                                        "type": XX,
                                        "value": 100
                                }, {
                                        "name": "count",
                                        "type": XX,
                                        "value": 9
                                }]
                        }],
                        "tableName": "product"
                },
                "sqlType": "UPDATE"
        }],
        "xid": "xid:xxx"
}
```

行锁?

undo log

业务场景分析

- 商品库存操作
  - 用户下单减库存
  - 商户增加库存

每个下都是单独立的分布式事务，不同的XID

商品增加库存，不是分布式事务

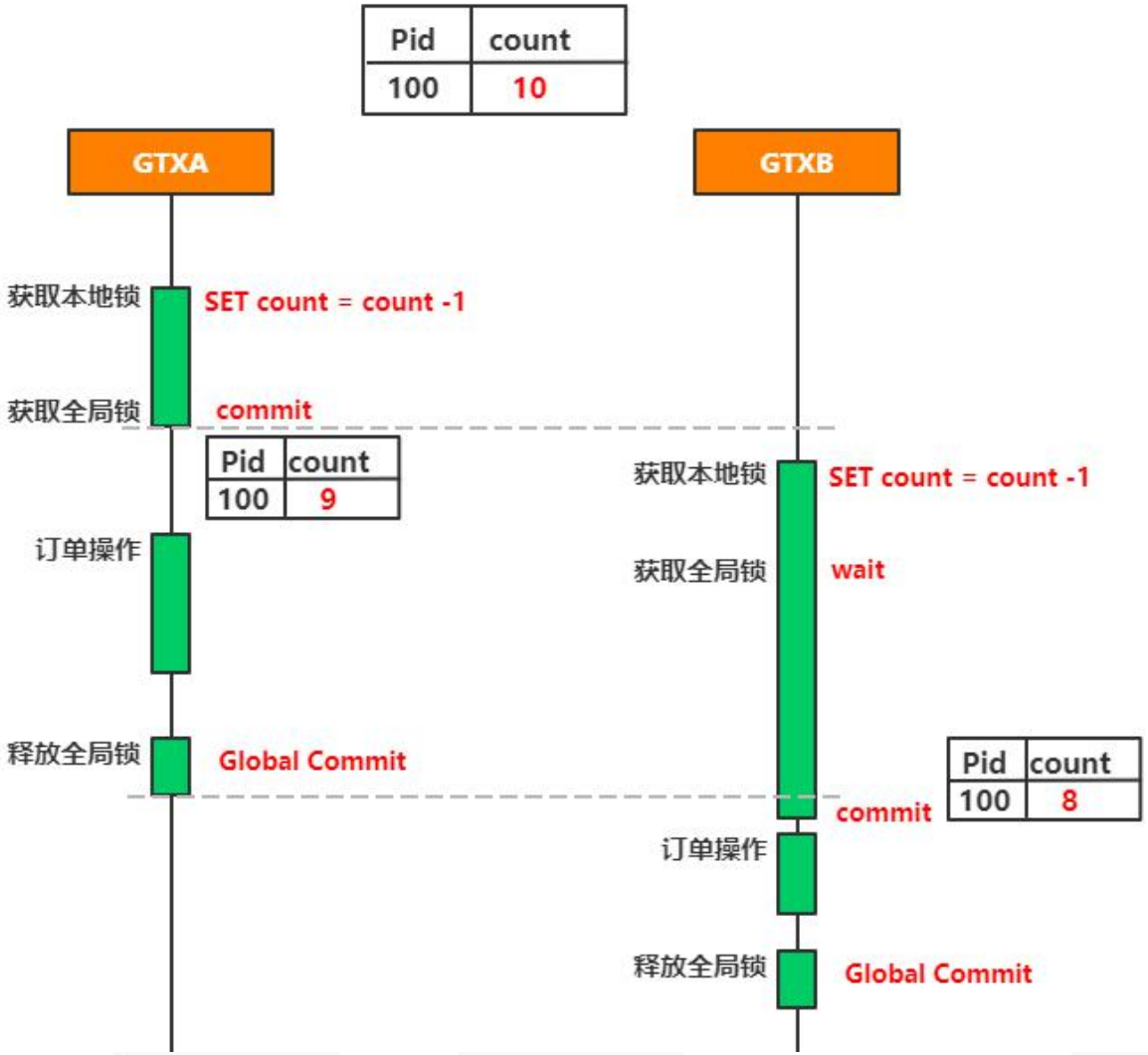
不检查全局锁，锁失效

隔离级别

- **Read Uncommitted**（读取未提交内容）：最低隔离级别，会读取到其他事务未提交的数据；
- **Read Committed**（读取提交内容）：事务过程中可以读取到其他事务已提交的数据
- **Repeatable Read**（可重复读）：每次读取相同结果集，不管其他事务是否提交；
- **Serializable**（可串行化）：事务排队，隔离级别最高，性能最差；

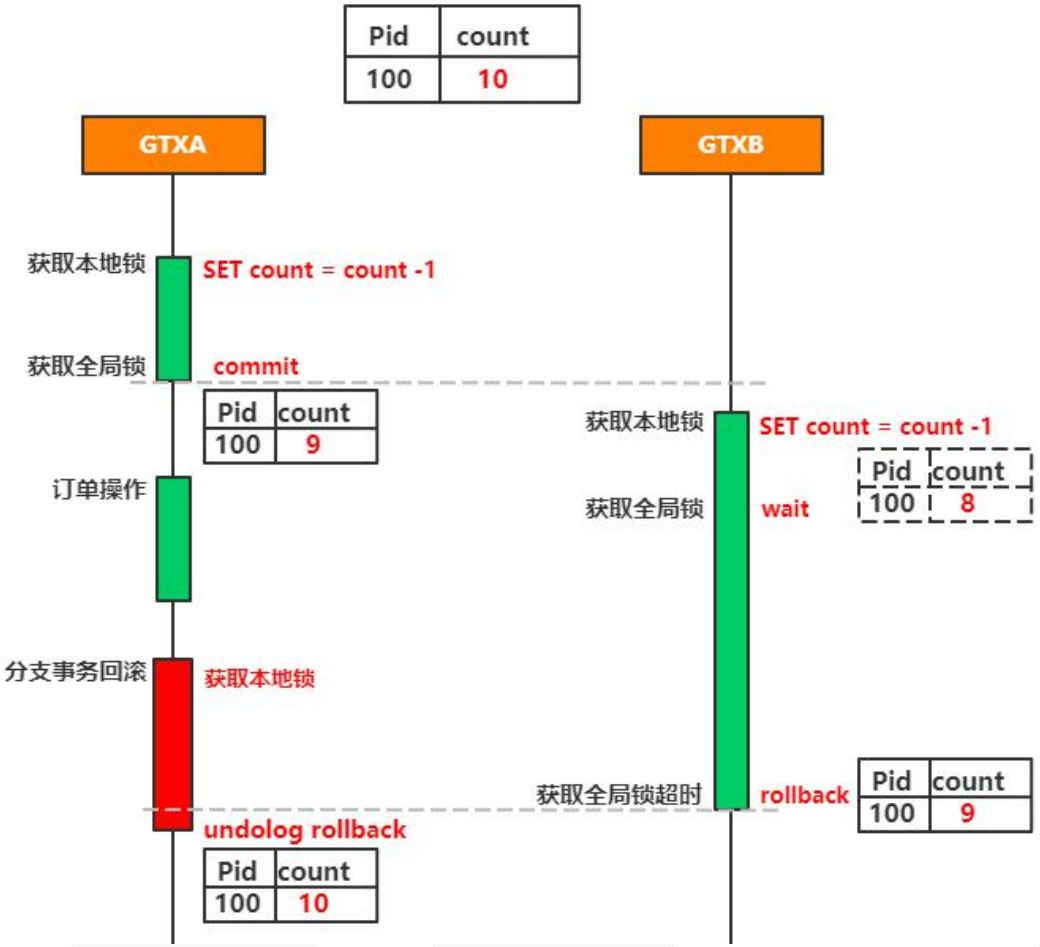
Seata隔离性

- 写隔离
  - **分支事务提交前拿到全局锁；**
  - 拿全局锁超时，回滚本地事务，释放本地锁；



Seata隔离性

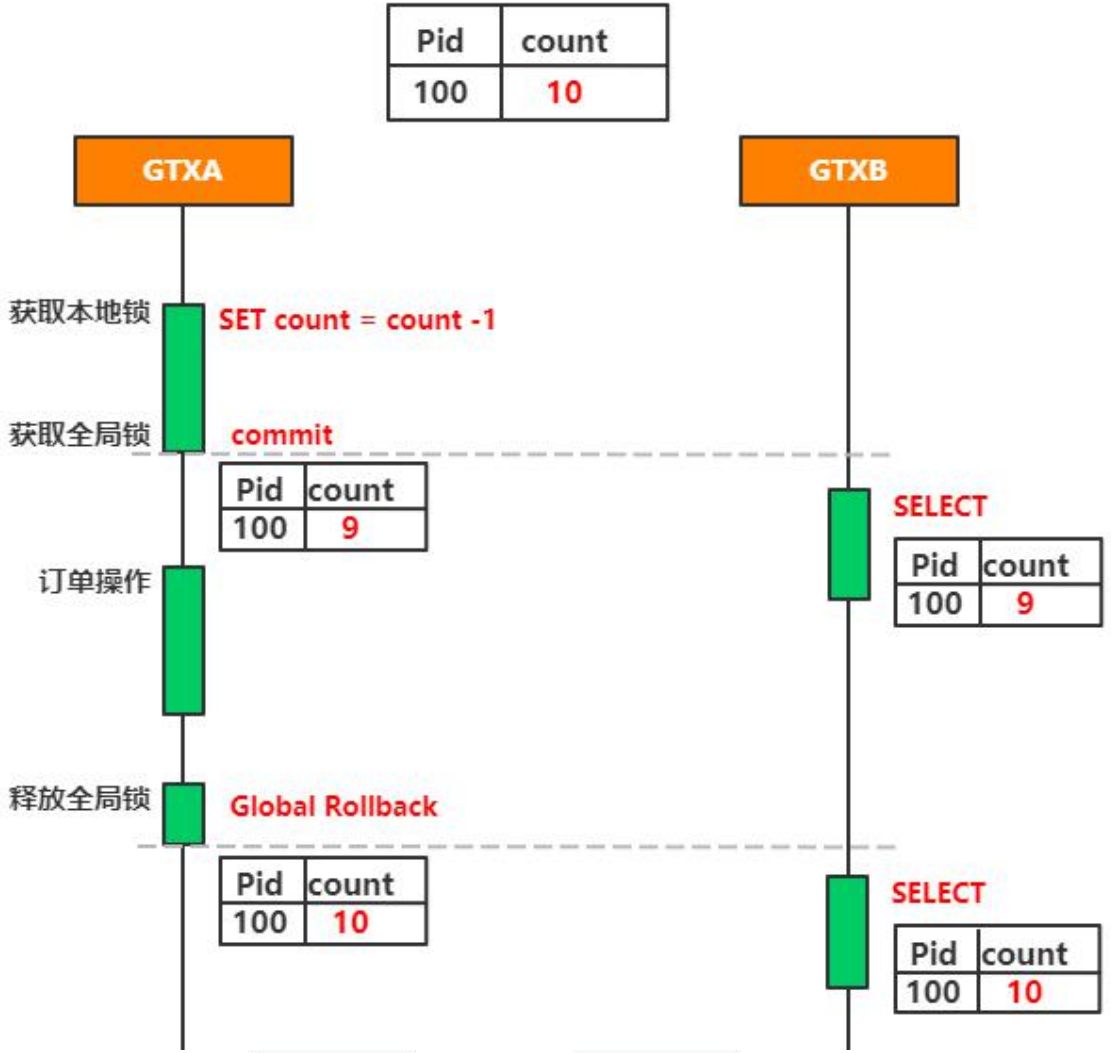
- 写隔离
  - 分支事务提交前拿到全局锁；
  - **拿全局锁超时，回滚本地事务，释放本地锁；**



Seata隔离性

- 读隔离（数据库读已提交）
  - 默认全局隔离级别是读未提交；

如何做到读已提交?

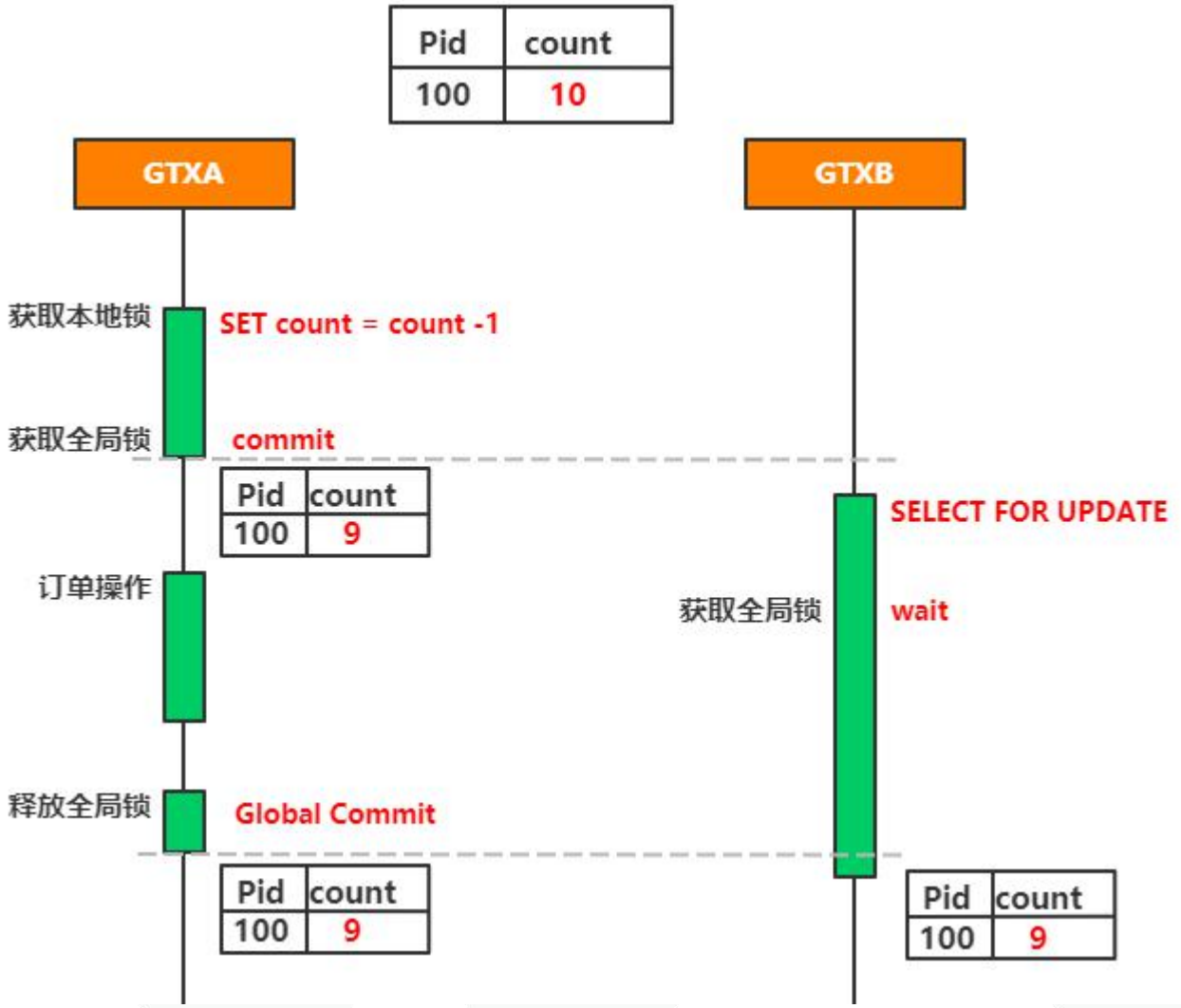


Seata隔离性

- 读隔离（数据库读已提交）
  - SELECT FOR UPDATE实现读已提交；

如何实现可重复读?

前镜像能否实现MVCC?



AT模型存在问题

- 重量级SDK;
- 依赖数据库本地事务的ACID特性;

NX 奈学教育

欢迎关注本人公众号
“架构之美”