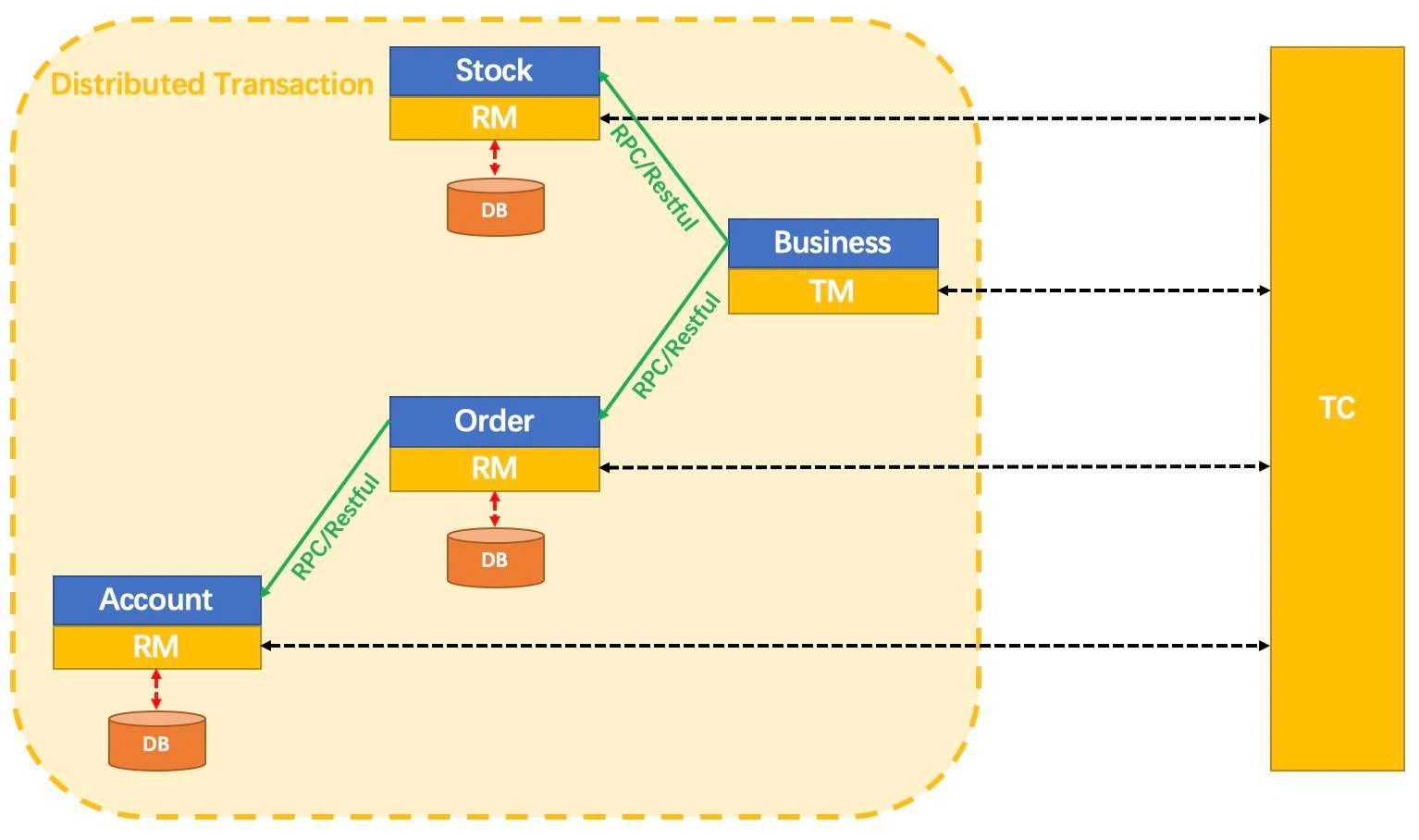
# Seata 概述

Seata 是一款开源的分布式事务解决方案，致力于提供高性能和简单易用的分布式事务服务。Seata 将为用户提供了 AT、TCC、SAGA 和 XA 事务模式，为用户打造一站式的分布式解决方案。



## AT 模式

使用条件：

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

### 整体机制

* 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。

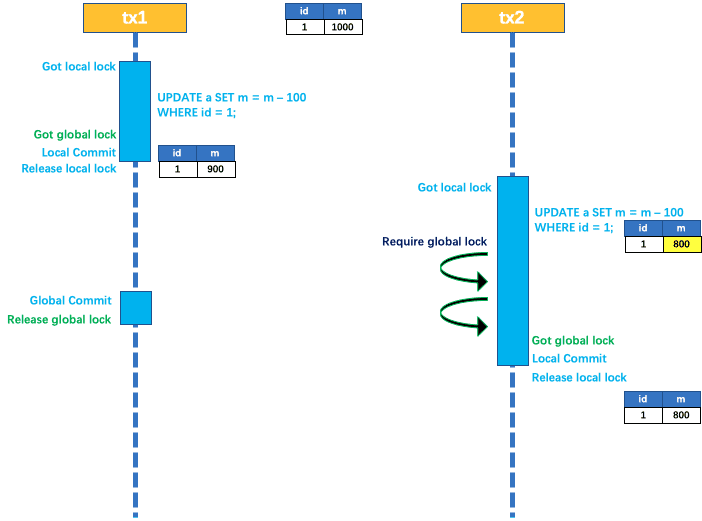
### 写隔离

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

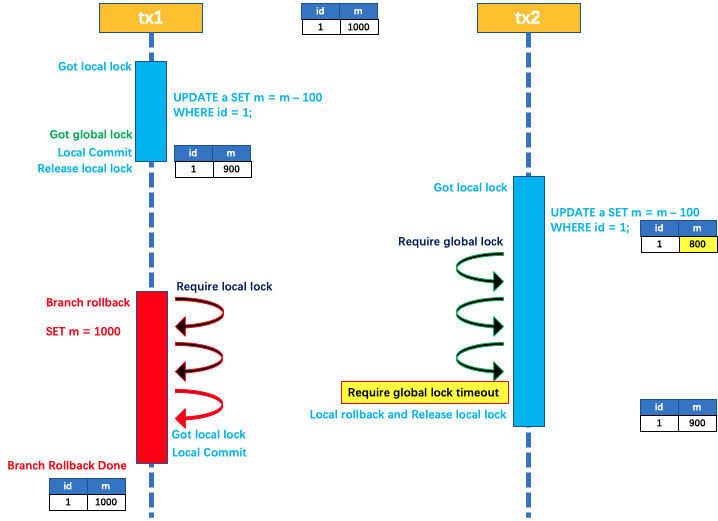
示例说明：

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

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



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



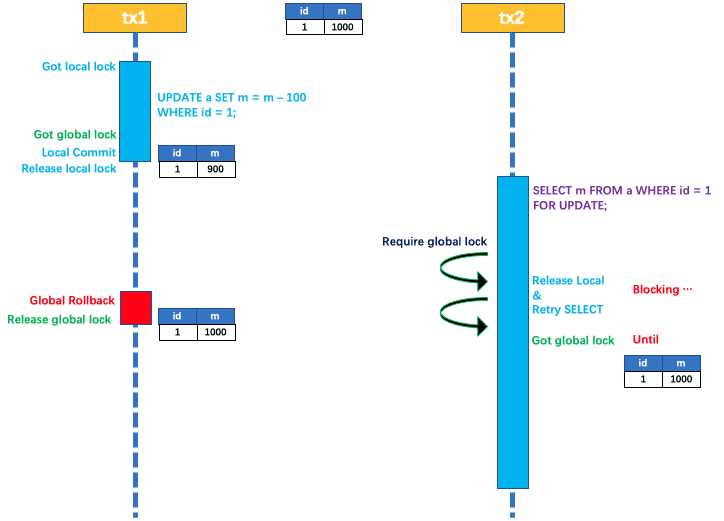
如果tx1的二阶段全局回滚，则tx1需要重新获取该数据的本地锁，进行反向补偿的更新操作，实现分支的回滚。

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

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

### 读隔离

在数据库本地事务隔离级别读已提交（Read Committed） 或以上的基础上，Seata（AT 模式）的默认全局隔离级别是读未提交（Read Uncommitted）。如果应用在特定场景下，必需要求全局的 读已提交 ，目前 Seata 的方式是通过 SELECT FOR UPDATE 语句的代理。



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

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

### 工作机制

以一个示例来说明整个 AT 分支的工作过程。

业务表：product

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Key** |
| id | bigint(20) | PRI |
| name | varchar(100) |  |
| since | varchar(100) |  |

AT 分支事务的业务逻辑：

update product set name = 'GTS' where name = 'TXC';

#### 一阶段

过程：

1. 解析 SQL：得到 SQL 的类型（UPDATE），表（product），条件（where name = 'TXC'）等相关的信息。
2. 查询前镜像：根据解析得到的条件信息，生成查询语句，定位数据。

select id, name, since from product where name = 'TXC';

得到前镜像：

| **id** | **name** | **since** |
| --- | --- | --- |
| 1 | TXC | 2014 |

1. 执行业务 SQL：更新这条记录的 name 为 'GTS'。
2. 查询后镜像：根据前镜像的结果，通过主键定位数据。

select id, name, since from product where id = 1;

得到后镜像：

| **id** | **name** | **since** |
| --- | --- | --- |
| 1 | GTS | 2014 |

1. 插入回滚日志：把前后镜像数据以及业务 SQL 相关的信息组成一条回滚日志记录，插入到 UNDO\_LOG 表中。

{

"branchId": 641789253,

"undoItems": [{

"afterImage": {

"rows": [{

"fields": [{

"name": "id",

"type": 4,

"value": 1

}, {

"name": "name",

"type": 12,

"value": "GTS"

}, {

"name": "since",

"type": 12,

"value": "2014"

}]

}],

"tableName": "product"

},

"beforeImage": {

"rows": [{

"fields": [{

"name": "id",

"type": 4,

"value": 1

}, {

"name": "name",

"type": 12,

"value": "TXC"

}, {

"name": "since",

"type": 12,

"value": "2014"

}]

}],

"tableName": "product"

},

"sqlType": "UPDATE"

}],

"xid": "xid:xxx"

}

1. 提交前，向 TC 注册分支：申请 product 表中，主键值等于 1 的记录的 全局锁 。
2. 本地事务提交：业务数据的更新和前面步骤中生成的 UNDO LOG 一并提交。
3. 将本地事务提交的结果上报给 TC。

#### 二阶段-回滚

1. 收到 TC 的分支回滚请求，开启一个本地事务，执行如下操作。
2. 通过 XID 和 Branch ID 查找到相应的 UNDO LOG 记录。
3. 数据校验：拿 UNDO LOG 中的后镜与当前数据进行比较，如果有不同，说明数据被当前全局事务之外的动作做了修改。这种情况，需要根据配置策略来做处理，详细的说明在另外的文档中介绍。
4. 根据 UNDO LOG 中的前镜像和业务 SQL 的相关信息生成并执行回滚的语句：

update product set name = 'TXC' where id = 1;

1. 提交本地事务。并把本地事务的执行结果（即分支事务回滚的结果）上报给 TC。

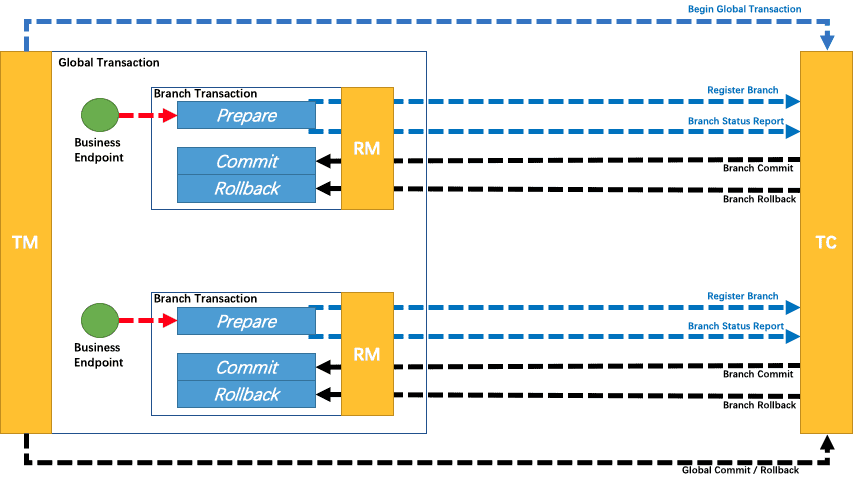
#### 二阶段-提交

1. 收到 TC 的分支提交请求，把请求放入一个异步任务的队列中，马上返回提交成功的结果给 TC。
2. 异步任务阶段的分支提交请求将异步和批量地删除相应 UNDO LOG 记录。

## TCC 模式

回顾总览中的描述：一个分布式的全局事务，整体是 两阶段提交 的模型。全局事务是由若干分支事务组成的，分支事务要满足 两阶段提交 的模型要求，即需要每个分支事务都具备自己的：

* 一阶段 prepare 行为
* 二阶段 commit 或 rollback 行为



根据两阶段行为模式的不同，我们将分支事务划分为 Automatic (Branch) Transaction Mode 和 Manual (Branch) Transaction Mode. AT 模式（参考链接 TBD）基于 支持本地 ACID 事务 的 关系型数据库：

* 一阶段 prepare 行为：在本地事务中，一并提交业务数据更新和相应回滚日志记录。
* 二阶段 commit 行为：马上成功结束，自动 异步批量清理回滚日志。
* 二阶段 rollback 行为：通过回滚日志，自动 生成补偿操作，完成数据回滚。

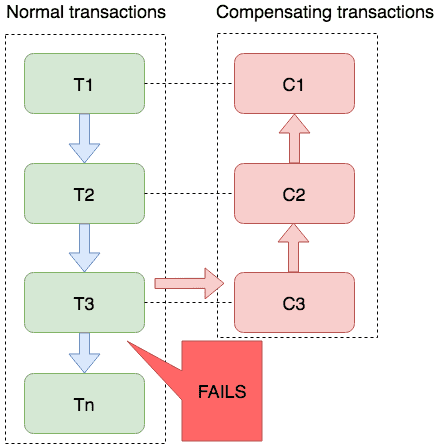
相应的，TCC 模式，不依赖于底层数据资源的事务支持：

* 一阶段 prepare 行为：调用 自定义 的 prepare 逻辑。
* 二阶段 commit 行为：调用 自定义 的 commit 逻辑。
* 二阶段 rollback 行为：调用 自定义 的 rollback 逻辑。

所谓 TCC 模式，是指支持把 自定义 的分支事务纳入到全局事务的管理中。

## Saga 模式

Saga模式是SEATA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

* 业务流程长、业务流程多
* 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

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

### 缺点

* 不保证隔离性

## Seata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。

# 事务分组

* 事务分组：seata的资源逻辑，可以按微服务的需要，在应用程序（客户端）对自行定义事务分组，每组取一个名字。
* 集群：seata-server服务端一个或多个节点组成的集群cluster。 应用程序（客户端）使用时需要指定事务逻辑分组与Seata服务端集群的映射关系。

## seata集群集成

1. 首先应用程序（客户端）中配置了事务分组（GlobalTransactionScanner 构造方法的txServiceGroup参数）。若应用程序是SpringBoot则通过seata.tx-service-group 配置
2. 应用程序（客户端）会通过用户配置的配置中心去寻找service.vgroupMapping .[事务分组配置项]，取得配置项的值就是TC集群的名称。若应用程序是SpringBoot则通过seata.service.vgroup-mapping.事务分组名=集群名称 配置
3. 拿到集群名称程序通过一定的前后缀+集群名称去构造服务名，各配置中心的服务名实现不同（前提是Seata-Server已经完成服务注册，且Seata-Server向注册中心报告cluster名与应用程序（客户端）配置的集群名称一致）
4. 拿到服务名去相应的注册中心去拉取相应服务名的服务列表，获得后端真实的TC服务列表（即Seata-Server集群节点列表）

这里多了一层获取事务分组到映射集群的配置。这样设计后，事务分组可以作为资源的逻辑隔离单位，出现某集群故障时可以快速failover，只切换对应分组，可以把故障缩减到服务级别，但前提也是你有足够server集群。

## 使用案例

seata注册、配置中心类型分为两大类：

* 内置File
* 第三方注册（配置）中心。如nacos等等，注册中心和配置中心之间没有约束，可各自使用不同具体选型。

### 第一类：内置File

file、db模式启动Seata Serve：

registry {

# file 、nacos 、eureka、redis、zk、consul、etcd3、sofa

type = "file" ---------------> 使用file作为注册中心

}

config {

# file、nacos 、apollo、zk、consul、etcd3

type = "file" ---------------> 使用file作为配置中心

file {

name = "file.conf"

}

}

* 读取配置 通过FileConfiguration本地加载file.conf的配置参数
* 获取事务分组(服务启动时加载配置) spring/springboot可配置在yml、properties中，对应值"my\_test\_tx\_group"即为事务分组名，若不配置则默认以：spring.application.name值+"-seata-service-group"拼接后的字符串作为分组名
* 查找TC集群名 拿到事务分组名"my\_test\_tx\_group"拼接成"service.vgroupMapping.my\_test\_tx\_group"查找TC集群名clusterName为"default"
* 查询TC服务 拼接"service."+clusterName+".grouplist"找到真实TC服务地址127.0.0.1:8091

### 第二类：注册中心和配置中心

server端registry.conf：

registry {

# file 、nacos 、eureka、redis、zk、consul、etcd3、sofa

type = "nacos" ---------------> 使用nacos作为注册中心

nacos {

application = "seata-server" ---------------> 指定注册至nacos注册中心的服务名

group = "SEATA\_GROUP" ---------------> 指定注册至nacos注册中心的分组名

serverAddr = "localhost" ---------------> nacos注册中心IP:端口

namespace = "" ---------------> nacos命名空间id，""为nacos保留public空间控件，用户勿配置namespace = "public"

cluster = "default" ---------------> 指定注册至nacos注册中心的集群名

}

}

config {

# file、nacos 、apollo、zk、consul、etcd3

type = "nacos" ------------> 使用nacos作为配置中心

nacos {

serverAddr = "localhost" ---------------> nacos注册中心IP:端口

namespace = ""

group = "SEATA\_GROUP" ---------------> nacos配置中心的分组名

dataId = "seataServer.properties" ---------------> nacos配置中心的配置ID

}

}

* 配置中心配置项：在Seata Server的安装目录conf下README-zh.md或README.md文件中介绍了Seata需要的常见脚本URL链接，包括三类：客户端的配置和SQL、SeataServer端部署所需SQL和脚本、配置中心配置项模板和脚本。
* 注册至注册中心 启动seata-server注册至nacos注册中心，查看nacos控制台服务列表确认是否成功

## 事务分组与高可用

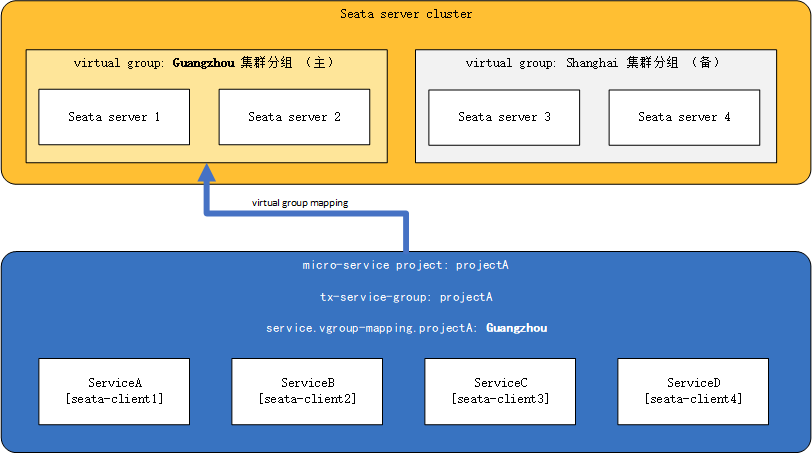
### 最佳实践1：TC的异地多机房容灾

* 假定TC集群部署在两个机房：guangzhou机房（主）和shanghai机房（备）各两个实例
* 一整套微服务架构项目：projectA
* projectA内有微服务：serviceA、serviceB、serviceC 和 serviceD

其中，projectA所有微服务的事务分组tx-service-group设置为：projectA，projectA正常情况下使用guangzhou的TC集群（主）。那么正常情况下，client端的配置如下所示：

seata.tx-service-group=projectA

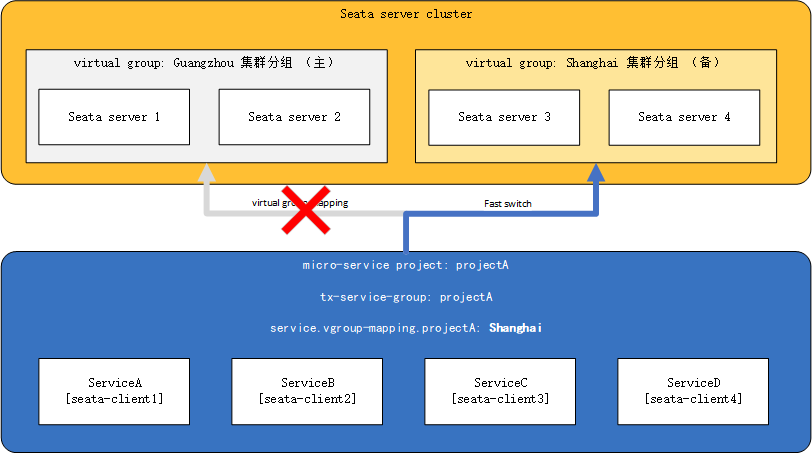
seata.service.vgroup-mapping.projectA=Guangzhou



假如此时guangzhou集群分组整个down掉，或者因为网络原因projectA暂时无法与Guangzhou机房通讯，那么我们将配置中心中的Guangzhou集群分组改为Shanghai，如下：

seata.service.vgroup-mapping.projectA=Shanghai

并推送到各个微服务，便完成了对整个projectA项目的TC集群动态切换。



### 最佳实践2：单一环境多应用接入

* 假设现在开发环境（或预发/生产）中存在一整套seata集群
* seata集群要服务于不同的微服务架构项目projectA、projectB、projectC
* projectA、projectB、projectC之间相对独立

我们将seata集群中的六个实例两两分组，使其分别服务于projectA、projectB与projectC，那么此时seata-server端的配置如下（以nacos注册中心为例）：

registry {

type = "nacos"

loadBalance = "RandomLoadBalance"

loadBalanceVirtualNodes = 10

nacos {

application = "seata-server"

serverAddr = "127.0.0.1:8848"

group = "DEFAULT\_GROUP"

namespace = "8f11aeb1-5042-461b-b88b-d47a7f7e01c0"

#同理在其他几个分组seata-server实例配置 project-b-group / project-c-group

cluster = "project-a-group"

username = "username"

password = "password"

}

}

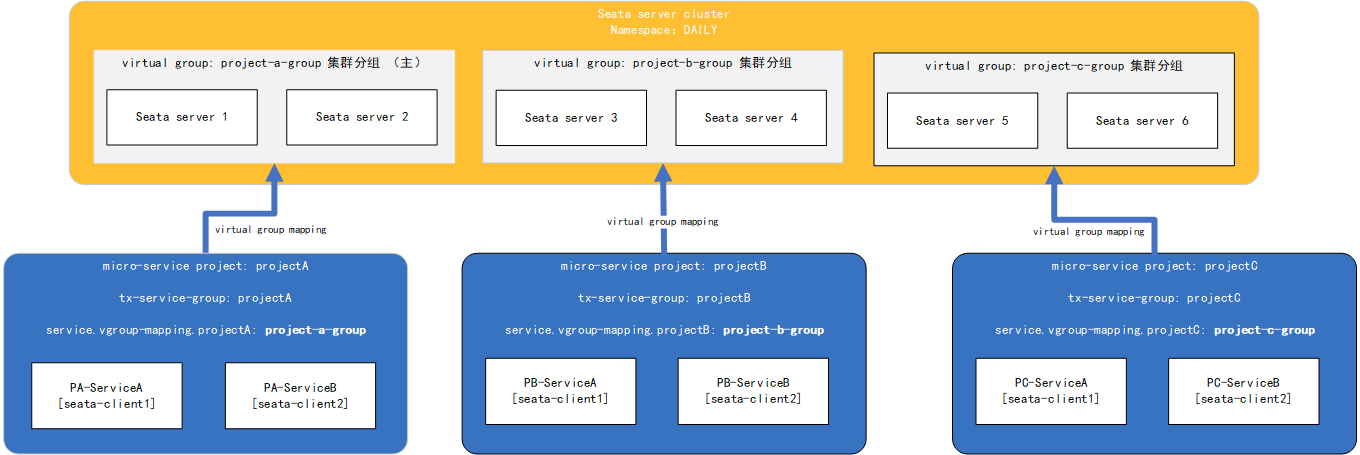
client端的配置如下：

seata.tx-service-group=projectA

#同理，projectB与projectC配置 project-b-group / project-c-group

seata.service.vgroup-mapping.projectA=project-a-group

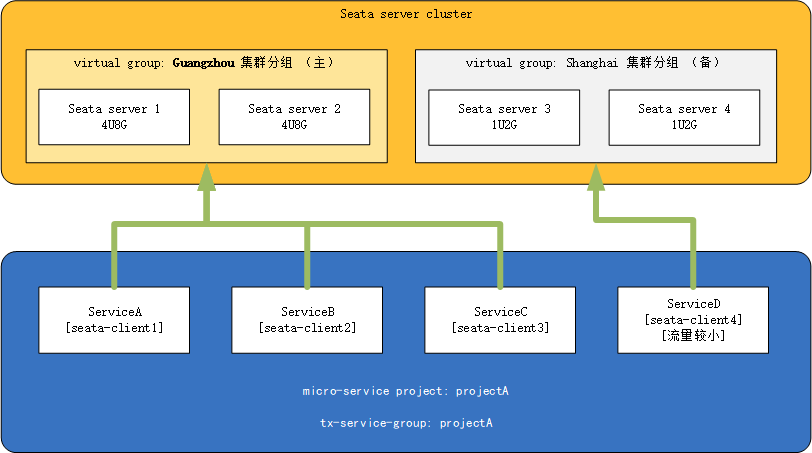
完成配置启动后，对应事务分组的TC单独为其应用服务，整体部署图如下：



### 最佳实践3：client的精细化控制

* 假定现在存在seata集群，Guangzhou机房实例运行在性能较高的机器上，Shanghai集群运行在性能较差的机器上
* 现存微服务架构项目projectA、projectA中有微服务ServiceA、ServiceB、ServiceC与ServiceD
* 其中ServiceD的流量较小，其余微服务流量较大

那么此时，我们可以将ServiceD微服务引流到Shanghai集群中去，将高性能的服务器让给其余流量较大的微服务（反之亦然，若存在某一个微服务流量特别大，我们也可以单独为此微服务开辟一个更高性能的集群，并将该client的virtual group指向该集群，其最终目的都是保证在流量洪峰时服务的可用）



### 最佳实践4：Seata的预发与生产隔离

* 大多数情况下，预发环境与生产环境会使用同一套数据库。基于这个条件，预发TC集群与生产TC集群必须使用同一个数据库保证全局事务的生效（即生产TC集群与预发TC集群使用同一个lock表，并使用不同的branch\_table与global\_table的情况）
* 我们记生产使用的branch表与global表分别为：global\_table与branch\_table；预发为global\_table\_pre，branch\_table\_pre
* 预发与生产共用lock\_table

此时，seata-server的 file.conf 配置如下：

store {

mode = "db"

db {

datasource = "druid"

dbType = "mysql"

driverClassName = "com.mysql.jdbc.Driver"

url = "jdbc:mysql://127.0.0.1:3306/seata"

user = "username"

password = "password"

minConn = 5

maxConn = 100

globalTable = "global\_table" ----> 预发为 "global\_table\_pre"

branchTable = "branch\_table" ----> 预发为 "branch\_table\_pre"

lockTable = "lock\_table"

queryLimit = 100

maxWait = 5000

}

}

seata-server的 registry.conf 配置如下（以nacos为例）

registry {

type = "nacos"

loadBalance = "RandomLoadBalance"

loadBalanceVirtualNodes = 10

nacos {

application = "seata-server"

serverAddr = "127.0.0.1:8848"

group = "DEFAULT\_GROUP"

namespace = "8f11aeb1-5042-461b-b88b-d47a7f7e01c0"

cluster = "pre-product" -->同理生产为 "product"

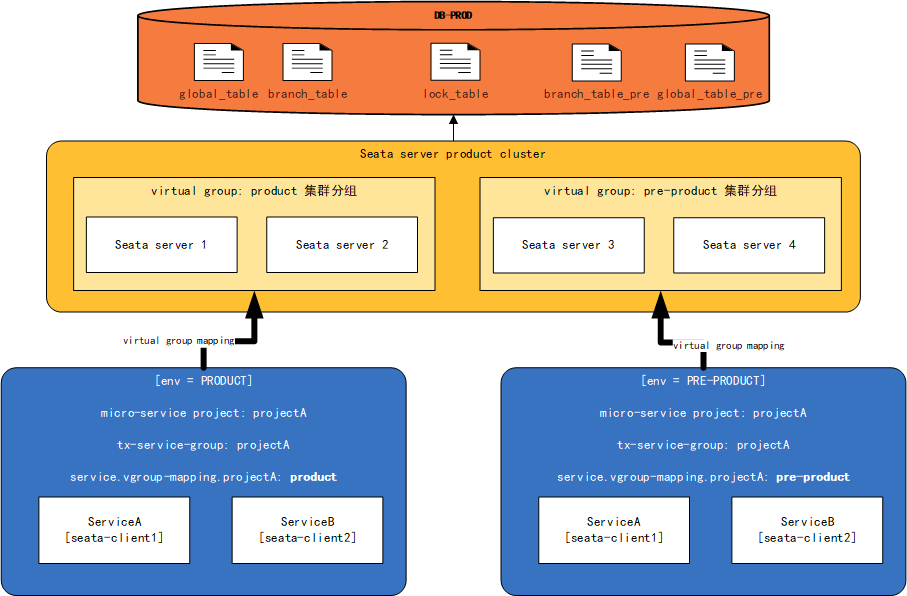
username = "username"

password = "password"

}

}

其部署图如下所示（不仅如此，你还可以将以上四个最佳实践依据你的实际生产情况组合搭配使用。）：



# Api支持

Seata API 分为两大类：High-Level API 和 Low-Level API ：

* High-Level API ：用于事务边界定义、控制及事务状态查询。
* Low-Level API ：用于控制事务上下文的传播。

## High-Level API

### GlobalTransaction

全局事务：包括开启事务、提交、回滚、获取当前状态等方法。

public interface GlobalTransaction {

/\*\*

\* 开启一个全局事务（使用默认的事务名和超时时间）

\*/

void begin() throws TransactionException;

/\*\*

\* 开启一个全局事务，并指定超时时间（使用默认的事务名）

\*/

void begin(int timeout) throws TransactionException;

/\*\*

\* 开启一个全局事务，并指定事务名和超时时间

\*/

void begin(int timeout, String name) throws TransactionException;

/\*\*

\* 全局提交

\*/

void commit() throws TransactionException;

/\*\*

\* 全局回滚

\*/

void rollback() throws TransactionException;

/\*\*

\* 获取事务的当前状态

\*/

GlobalStatus getStatus() throws TransactionException;

/\*\*

\* 获取事务的 XID

\*/

String getXid();

}

### GlobalTransactionContext

GlobalTransaction 实例的获取需要通过 GlobalTransactionContext：

/\*\*

\* 获取当前的全局事务实例，如果没有则创建一个新的实例。

\*/

public static GlobalTransaction getCurrentOrCreate() {

GlobalTransaction tx = getCurrent();

if (tx == null) {

return createNew();

}

return tx;

}

/\*\*

\* 重新载入给定 XID 的全局事务实例，这个实例不允许执行开启事务的操作。

\* 这个 API 通常用于失败的事务的后续集中处理。

\* 比如：全局提交超时，后续集中处理通过重新载入该实例，通过实例方法获取事务当前状态，并根据状态判断是否需要重试全局提交操作。

\*/

public static GlobalTransaction reload(String xid) throws TransactionException {

GlobalTransaction tx = new DefaultGlobalTransaction(xid, GlobalStatus.UnKnown, GlobalTransactionRole.Launcher) {

@Override

public void begin(int timeout, String name) throws TransactionException {

throw new IllegalStateException("Never BEGIN on a RELOADED GlobalTransaction. ");

}

};

return tx;

}

### TransactionalTemplate

事务化模板：通过上述 GlobalTransaction 和 GlobalTransactionContext API 把一个业务服务的调用包装成带有分布式事务支持的服务。

public class TransactionalTemplate {

public Object execute(TransactionalExecutor business) throws TransactionalExecutor.ExecutionException {

// 1. 获取当前全局事务实例或创建新的实例

GlobalTransaction tx = GlobalTransactionContext.getCurrentOrCreate();

// 2. 开启全局事务

try {

tx.begin(business.timeout(), business.name());

} catch (TransactionException txe) {

// 2.1 开启失败

throw new TransactionalExecutor.ExecutionException(tx, txe,

TransactionalExecutor.Code.BeginFailure);

}

Object rs = null;

try {

// 3. 调用业务服务

rs = business.execute();

} catch (Throwable ex) {

// 业务调用本身的异常

try {

// 全局回滚

tx.rollback();

// 3.1 全局回滚成功：抛出原始业务异常

throw new TransactionalExecutor.ExecutionException(tx, TransactionalExecutor.Code.RollbackDone, ex);

} catch (TransactionException txe) {

// 3.2 全局回滚失败：

throw new TransactionalExecutor.ExecutionException(tx, txe,

TransactionalExecutor.Code.RollbackFailure, ex);

}

}

// 4. 全局提交

try {

tx.commit();

} catch (TransactionException txe) {

// 4.1 全局提交失败：

throw new TransactionalExecutor.ExecutionException(tx, txe,

TransactionalExecutor.Code.CommitFailure);

}

return rs;

}

}

模板方法执行的异常：ExecutionException，

class ExecutionException extends Exception {

// 发生异常的事务实例

private GlobalTransaction transaction;

// 异常编码：

// BeginFailure（开启事务失败）

// CommitFailure（全局提交失败）

// RollbackFailure（全局回滚失败）

// RollbackDone（全局回滚成功）

private Code code;

// 触发回滚的业务原始异常

private Throwable originalException;

外层调用逻辑 try-catch 这个异常，根据异常编码进行处理：

* BeginFailure （开启事务失败）：getCause() 得到开启事务失败的框架异常，getOriginalException() 为空。
* CommitFailure （全局提交失败）：getCause() 得到全局提交失败的框架异常，getOriginalException() 为空。
* RollbackFailure （全局回滚失败）：getCause() 得到全局回滚失败的框架异常，getOriginalException() 业务应用的原始异常。
* RollbackDone （全局回滚成功）：getCause() 为空，getOriginalException() 业务应用的原始异常。

## Low-Level API

### RootContext

事务的根上下文：负责在应用的运行时，维护 XID 。

/\*\*

\* 得到当前应用运行时的全局事务 XID

\*/

public static String getXID() {

return CONTEXT\_HOLDER.get(KEY\_XID);

}

/\*\*

\* 将全局事务 XID 绑定到当前应用的运行时中

\*/

public static void bind(String xid) {

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("bind " + xid);

}

CONTEXT\_HOLDER.put(KEY\_XID, xid);

}

/\*\*

\* 将全局事务 XID 从当前应用的运行时中解除绑定，同时将 XID 返回

\*/

public static String unbind() {

String xid = CONTEXT\_HOLDER.remove(KEY\_XID);

if (LOGGER.isDebugEnabled()) {

LOGGER.debug("unbind " + xid);

}

return xid;

}

/\*\*

\* 判断当前应用的运行时是否处于全局事务的上下文中

\*/

public static boolean inGlobalTransaction() {

return CONTEXT\_HOLDER.get(KEY\_XID) != null;

}

High-Level API 的实现都是基于 RootContext 中维护的 XID 来做的。应用的当前运行的操作是否在一个全局事务的上下文中，就是看 RootContext 中是否有 XID。RootContext 的默认实现是基于 ThreadLocal 的，即 XID 保存在当前线程上下文中。Low-Level API 的两个典型的应用场景：

* 远程调用事务上下文的传播

远程调用前获取当前 XID：String xid = RootContext.getXID();

远程调用过程把 XID 也传递到服务提供方，在执行服务提供方的业务逻辑前，把 XID 绑定到当前应用的运行时：RootContext.bind(rpcXid);

* 事务的暂停和恢复

在一个全局事务中，如果需要某些业务逻辑不在全局事务的管辖范围内，则在调用前，把 XID 解绑：String unbindXid = RootContext.unbind();

待相关业务逻辑执行完成，再把 XID 绑定回去，即可实现全局事务的恢复：RootContext.bind(unbindXid);

# 微服务框架支持

## 事务上下文

Seata 的事务上下文由 RootContext 来管理。应用开启一个全局事务后，RootContext 会自动绑定该事务的 XID，事务结束（提交或回滚完成），RootContext 会自动解绑 XID。

// 绑定 XID

RootContext.bind(xid);

// 解绑 XID

String xid = RootContext.unbind();

应用可以通过 RootContext 的 API 接口来获取当前运行时的全局事务 XID。

// 获取 XID

String xid = RootContext.getXID();

应用是否运行在一个全局事务的上下文中，就是通过 RootContext 是否绑定 XID 来判定的。

public static boolean inGlobalTransaction() {

return CONTEXT\_HOLDER.get(KEY\_XID) != null;

}

## 事务传播

Seata 全局事务的传播机制就是指事务上下文的传播，根本上，就是 XID 的应用运行时的传播方式。

### 服务内部的事务传播

默认的，RootContext 的实现是基于 ThreadLocal 的，即 XID 绑定在当前线程上下文中。

public class ThreadLocalContextCore implements ContextCore {

private ThreadLocal<Map<String, String>> threadLocal = new ThreadLocal<Map<String, String>>() {

@Override

protected Map<String, String> initialValue() {

return new HashMap<String, String>();

}

};

@Override

public String put(String key, String value) {

return threadLocal.get().put(key, value);

}

@Override

public String get(String key) {

return threadLocal.get().get(key);

}

@Override

public String remove(String key) {

return threadLocal.get().remove(key);

}

}

所以服务内部的 XID 传播通常是天然的通过同一个线程的调用链路串连起来的。默认不做任何处理，事务的上下文就是传播下去的。如果希望挂起事务上下文，则需要通过 RootContext 提供的 API 来实现：

// 挂起（暂停）

String xid = RootContext.unbind();

// TODO: 运行在全局事务外的业务逻辑

// 恢复全局事务上下文

RootContext.bind(xid);

### 跨服务调用的事务传播

通过上述基本原理，我们可以很容易理解：

跨服务调用场景下的事务传播，本质上就是要把 XID 通过服务调用传递到服务提供方，并绑定到 RootContext 中去。

只要能做到这点，理论上 Seata 可以支持任意的微服务框架。