## 一站式微服务解决方案Spring Cloud Alibaba

主讲：Cat 老师

北京动力节点教育科技有限公司  
2020 • 北京

动力节点•版权所有•禁止传播

## 快速回顾

1、Spring家族开源项目梳理

2、Spring Cloud下的开源项目梳理

3、Spring Cloud Alibaba下的开源及商业项目梳理

4、微服务的基础模型：服务消费者-注册中心-服务提供者

5、What is Nacos？

6、Nacos的运行环境部署（Java写的，springboot项目）

7、Nacos的后台web管控台

8、Nacos作为注册中心注册服务

9、Nacos作为注册中心发现/订阅服务

10、服务消费者负载均衡调用服务提供者（ribbon）restTemple、feign（openfeign）

11、Nacos宕机时服务消费者缓存注册中心信息

12、Nacos作为配置中心存储项目各种配置

13、Nacos作为配置中心支持自动配置刷新（不需要重启应用）

14、Nacos配置中心DataId+Group+Properties/yaml+配置内容（比较灵活）

15、Nacos配置中心多环境配置（profile）

即${spring.application.name}-${profile}.${file-extension:properties}

16、Nacos服务配置数据模型（命名空间、Group、Data Id）

17、Nacos数据持久化（mysql）

18、Ncaos集群部署（nginx）

19、主要调用方式：restTemplate、feign、ribbon（spring cloud）

20、流量控制Sentinel（流控、降级、热点、系统、授权 规则）

21、Sentinel Darshboard通信原理（与微服务通信）

22、Sentinel 对应用保护的三种方式

23、Sentinel整合RestTemplate流控熔断

24、Sentinel整合Feign流控熔断

25、Sentinel规则持久化（默认、pull模式、push模式）

26、Spring Cloud Gateway网关（核心概念、如何工作、路由、谓词11个、过滤器31个）

27、Spring Cloud Gateway自定义谓词

28、Spring Cloud Gateway谓词不匹配404处理

29、Spring Cloud Gateway自定义路由过滤器

30、Spring Cloud Gateway全局过滤器（默认根据条件注解生效，不需要单独配置）

31、Spring Cloud Gateway集成ribbon负载均衡

32、Spring Cloud Gateway集成Sentinel

33、Spring Cloud Gateway集成Sentinel规则持久化（文件、nacos）

34、Spring Cloud Gateway内部流程源码分析

35、Spring cloud gateway跨域CORS

36、Skywalking分布式链路跟踪

37、Skywalking主要功能特性和整体架构

38、SkyWalking 环境搭建部署

39、SkyWalking Agent跟踪微服务

40、IDEA中使用Skywalking Agent跟踪运行的程序

41、Skywalking告警和回调通知

42、Skywalking持久化到elasticsearch

43、Skywalking跨多个微服务跟踪

44、自定义SkyWalking链路追踪

45、Skywalking集成日志框架logback

46、Skywalking ui页面功能

47、Skywalking集群

48、什么是分布式事务

49、What is Seata？

50、Seata TC Server运行环境部署

51、AT事务模式-单体应用多数据源应用

52、AT事务模式-微服务应用

53、AT事务模式工作机制

## AT事务模式运行机制解读

AT 模式的前提：

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；

整体机制是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；（本地事务，就已经在数据库持久化了）

二阶段：

如果没有异常提交异步化，非常快速地完成；（正常情况，那就提交了，同步一下TC Server的状态，删除回滚日志）

如果有异常回滚通过一阶段的回滚日志进行反向补偿；（比如订单删除，库存加回去，余额加回去）；

**写隔离**

一阶段本地事务提交前，需要确保先拿到全局锁；

拿不到 全局锁 ，不能提交本地事务；

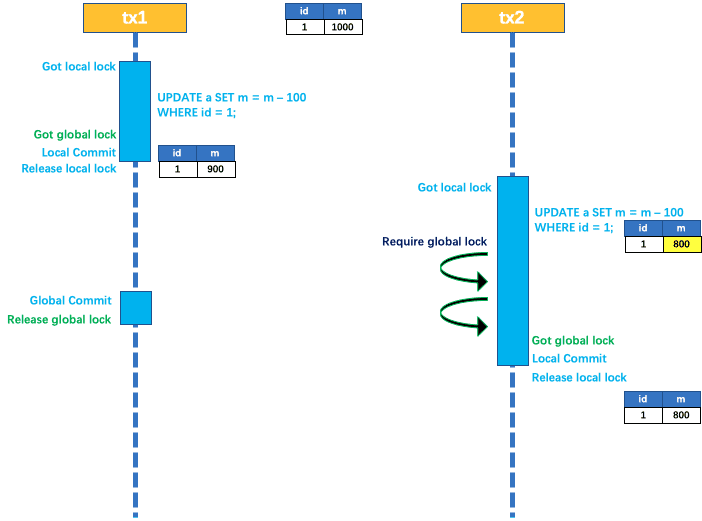
拿 全局锁 的尝试被限制在一定范围内，超出范围将放弃，并回滚本地事务，释放本地锁；

以一个示例来说明：

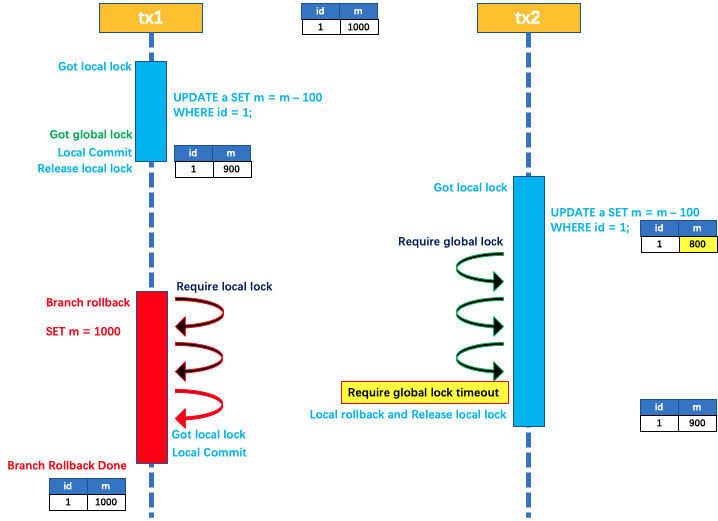
**两个**或者多个全局事务 tx1 和 tx2，分别并发对 a 表的 m 字段进行更新操作，m 的初始值 1000；

假设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 ，拿到了全局锁，本地提交并释放本地锁；

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，该记录的全局锁一直会被 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；

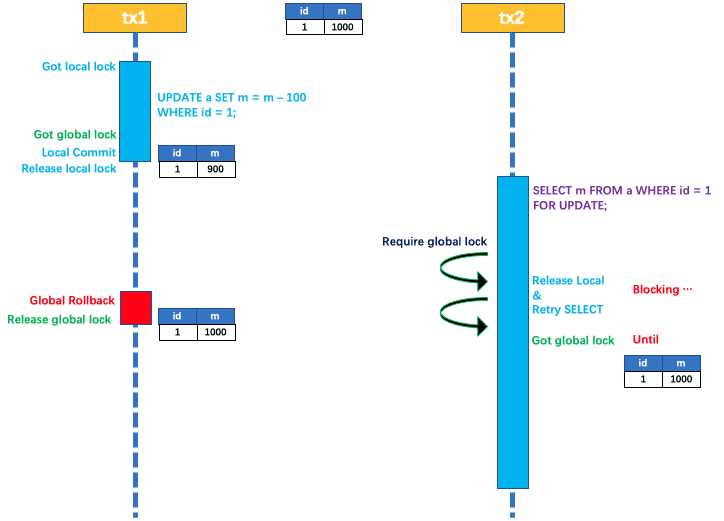
此时，如果 tx2 仍在等待该数据的 全局锁，同时持有本地锁，则 tx1 的分支回滚会失败。分支的回滚会一直重试，直到 tx2 的 全局锁 等锁超时，放弃 全局锁 并回滚本地事务释放本地锁，tx1 的分支回滚最终成功；

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；



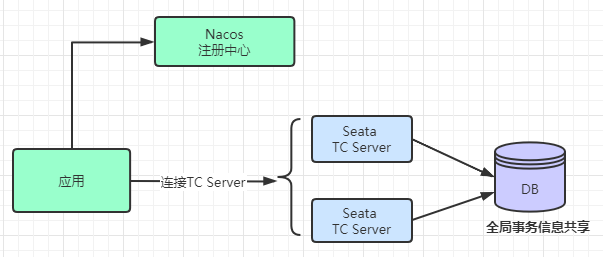
SELECT FOR UPDATE 语句的执行会申请 全局锁 ，如果 全局锁 被其他事务持有，则释放本地锁（回滚 SELECT FOR UPDATE 语句的本地执行）并重试，这个过程中，查询是被 block 住的，直到 全局锁 拿到，即读取的相关数据是 已提交 的，才返回；

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；

## Seata TC Server集群部署

生产环境下，需要部署**集群** Seata **TC** Server，实现高可用，在集群时，多个 Seata TC Server 通过 **db** 数据库或者redis实现全局事务会话信息的共享；

每个Seata TC Server注册自己到注册中心上，应用从注册中心获得Seata TC Server实例，这就是Seata TC Server的集群；



Seata TC Server 对主流的注册中心都提供了集成，Naco作为注册中心越来越流行，这里我们就采用Nacos；

Seata TC Server集群搭建具体步骤：

1、下载并解压两个seata-server-1.3.0.tar.gz；

2、初始化 Seata TC Server 的 db 数据库，在 MySQL 中，创建 seata 数据库，并在该库下执行如下SQL脚本：

使用seata-1.3.0\script\server\db脚本（网盘有共享）

3、修改 seata/conf/file.conf 配置文件，修改使用 db 数据库，实现 Seata TC Server 的全局事务会话信息的共享；

（1）mode = "db"

（2）数据库的连接信息

driverClassName = "com.mysql.cj.jdbc.Driver"

url = "jdbc:mysql://39.99.163.122:3306/seata"

user = "mysql"

password = "UoT1R8[09/VsfXoO5>6YteB"

4、设置使用 Nacos 注册中心；

修改 seata/conf/registry.conf 配置文件，设置使用 Nacos 注册中心；

1. 、type = "nacos"
2. Nacos连接信息：

nacos {

application = "seata-server"

serverAddr = "127.0.0.1:8848"

group = "SEATA\_GROUP"

namespace = ""

cluster = "default"

username = ""

password = ""

}

5、启动数据库和nacos；

6、启动两个 TC Server

执行 ./seata-server.sh -p 18091 -n 1 命令，启动第一个TC Server;

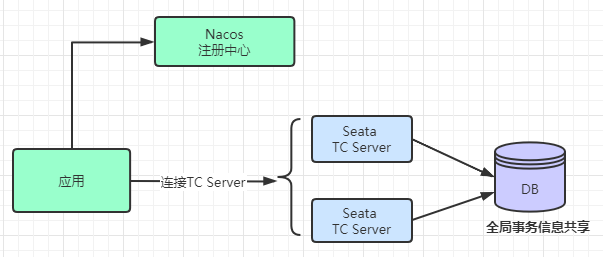
-p：Seata TC Server 监听的端口；

-n：Server node，在多个 TC Server 时，需区分各自节点，用于生成不同区间的 transactionId 事务编号，以免冲突；

执行 ./seata-server.sh -p 28091 -n 2 命令，启动第二个TC Server；

7、打开Nacos注册中心控制台，可以看到有两个Seata TC Server 实例；

8、应用测试；



**对于SpringBoot单体应用：**

1、添加nacos客户端依赖；

*<!-- nacos-client -->*<dependency>  
 <groupId>com.alibaba.nacos</groupId>  
 <artifactId>nacos-client</artifactId>  
 <version>1.3.1</version>  
</dependency>

2、配置application.properties文件

*#----------------------------------------------------------  
# Seata应用编号，默认为${spring.application.name}*seata.application-id=springcloud-order-seata  
*# Seata事务组编号，用于TC集群名*seata.tx-service-group=springcloud-order-seata-group  
*# 虚拟组和分组的映射*seata.service.vgroup-mapping.springcloud-order-seata-group=default  
  
*#seata-spring-boot-starter 1.1版本少一些配置项*seata.enabled=true  
seata.registry.type=nacos  
seata.registry.nacos.cluster=default  
seata.registry.nacos.server-addr=192.168.172.128:8848  
seata.registry.nacos.group=SEATA\_GROUP  
seata.registry.nacos.application=seata-server

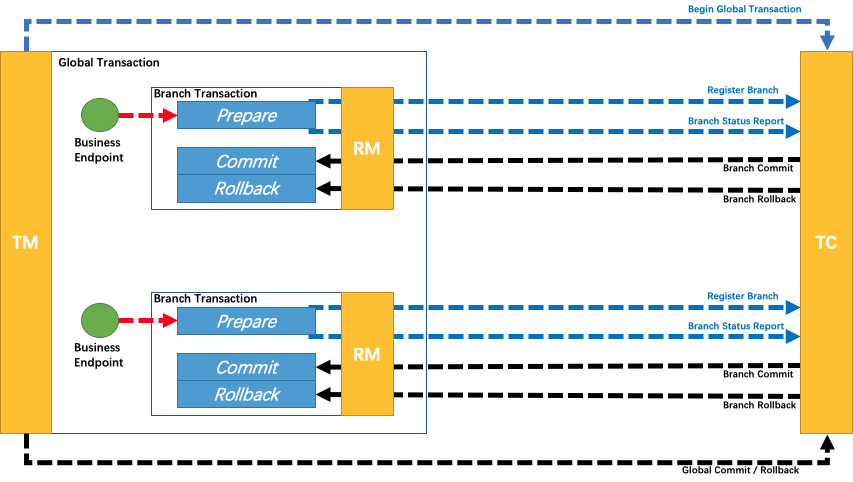
*#----------------------------------------------------------*

**对于Spring Cloud Alibaba微服务应用：**

则不需要加nacos的jar包依赖，application.properties文件配置完全一样；

## TCC事务模式执行机制

AT模式基本上能满足我们使用分布式事务大部分需求，但涉及非关系型数据库与中间件的操作、跨公司服务的调用、跨语言的应用调用就需要结合TCC模式；



一个分布式的全局事务，整体是两阶段提交（Try - [Comfirm/Cancel]）的模型，在Seata中，AT模式与TCC模式事实上都是基于两阶段提交，它们的区别在于：

AT模式基于支持本地ACID事务的关系型数据库：

1、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”；

2、二阶段 commit 行为：马上成功结束，自动异步批量清理回滚日志；

3、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；

而TCC 模式，需要我们人为编写代码实现提交和回滚：

1、一阶段 prepare 行为：调用自定义的 prepare 逻辑；(真正要做的事情，比如插入订单，更新库存，更新余额)

2、二阶段 commit 行为：调用自定义的 commit 逻辑；（自己写代码实现）

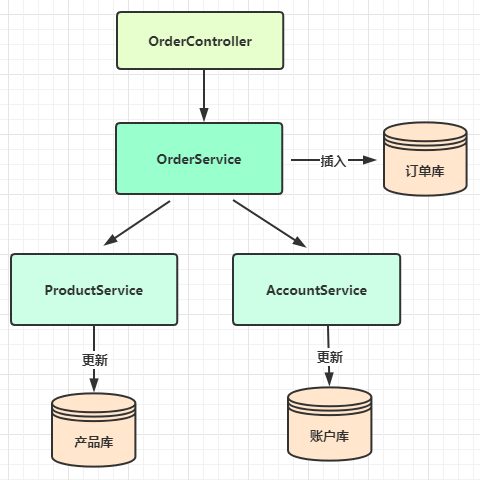
3、二阶段 rollback 行为：调用自定义的 rollback 逻辑；（自己写代码实现）

所以TCC模式，就是把自定义的分支事务的提交和回滚并纳入到全局事务管理中；

通俗来说，Seata的TCC模式就是手工版本的AT模式，它允许你自定义两阶段的处理逻辑而不需要依赖AT模式的undo\_log回滚表；

## TCC事务模式应用实践

### 基于SpringBoot单体应用的TCC事务



@LocalTCC  
public interface AccountService {  
 */\*\*  
 \* 扣除余额  
 \* 定义两阶段提交  
 \* name = reduceStock为一阶段try方法  
 \* commitMethod = commitTcc 为二阶段确认方法  
 \* rollbackMethod = cancel 为二阶段取消方法  
 \* BusinessActionContextParameter注解 可传递参数到二阶段方法  
 \*  
 \* @param userId 用户ID  
 \* @param money 扣减金额  
 \* @throws Exception 失败时抛出异常  
 \*/* @TwoPhaseBusinessAction(name = "reduceBalance", commitMethod = "commitTcc", rollbackMethod = "cancelTcc")  
 void reduceBalance(@BusinessActionContextParameter(paramName = "userId") Integer userId,  
 @BusinessActionContextParameter(paramName = "money") BigDecimal money);  
  
 */\*\*  
 \* 确认方法、可以另命名，但要保证与commitMethod一致  
 \* context可以传递try方法的参数  
 \*  
 \* @param context 上下文  
 \* @return boolean  
 \*/* boolean commitTcc(BusinessActionContext context);  
  
 */\*\*  
 \* 二阶段取消方法  
 \*  
 \* @param context 上下文  
 \* @return boolean  
 \*/* boolean cancelTcc(BusinessActionContext context);  
}

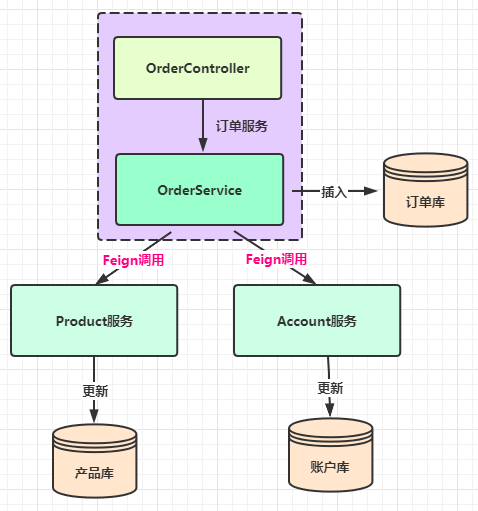
@LocalTCC注解标识此TCC为本地模式，即该事务是本地调用，非RPC调用，@LocalTCC一定需要注解在接口上，此接口可以是寻常的业务接口，只要实现了TCC的两阶段提交对应方法即可；

@TwoPhaseBusinessAction，该注解标识为TCC模式，注解try方法，其中name为当前tcc方法的bean名称，写方法名便可（全局唯一），commitMethod指提交方法，rollbackMethod指事务回滚方法，指定好三个方法之后，Seata会根据事务的成功或失败，通过动态代理去帮我们自动调用提交或者回滚；

@BusinessActionContextParameter 注解可以将参数传递到二阶段（commitMethod/rollbackMethod）的方法；

BusinessActionContext 是指TCC事务上下文，携带了业务方法的参数；

### 基于Spring Cloud Alibaba的TCC分布式事务



具体代码实现和springboot单体应用的代码实现几乎没有区别，具体参考Git上提交的代码；

由于Seata出现时间并不长，也在不断的改进中，在实际面试中应该不会问大家比较底层的实现，同学们如果感兴趣的话，基于我们已有的源码阅读经验，可以看一下Seata的源码，它如何进行事务隔离保证数据一致性，官方提供的文档并不详细；