## 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

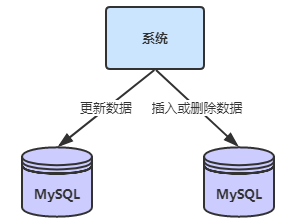
47、Skywalking集群

## 分布式事务

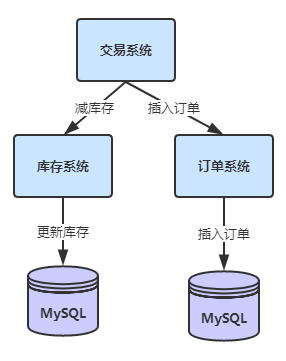
事务是数据库的概念，数据库事务（ACID：原子性、一致性、隔离性和持久性）；

分布式事务的产生，是由于数据库的拆分和分布式架构(微服务)带来的，在常规情况下，我们在一个进程中操作一个数据库，这属于本地事务，如果在一个进程中操作多个数据库，或者在多个进程中操作一个或多个数据库，就产生了分布式事务；

（1）数据库分库分表就产生了分布式事务；



（2）项目拆分服务化也产生了分布式事务；



## What is seata？

Seata是一款开源的分布式事务解决方案，致力于在微服务架构下提供高性能和简单易用的分布式事务服务；

Seata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；

四种事务模式中，XA模式正在开发中...，其他事务模式已经实现；

目前使用的流行度情况是：AT > TCC > Saga；

我们可以参看seata各公司使用列表：

<https://github.com/seata/seata/issues/1246> 大部分公司都采用的AT事务模式；

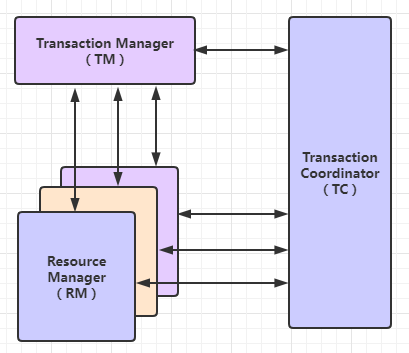
Seata已经在国内很多团队开始落地，其中不乏有大公司；

Github：<https://github.com/seata/seata>

官网：<http://seata.io/>

当前最新版本：**1.3.0**

在Seata的架构中，一共有三个角色：



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

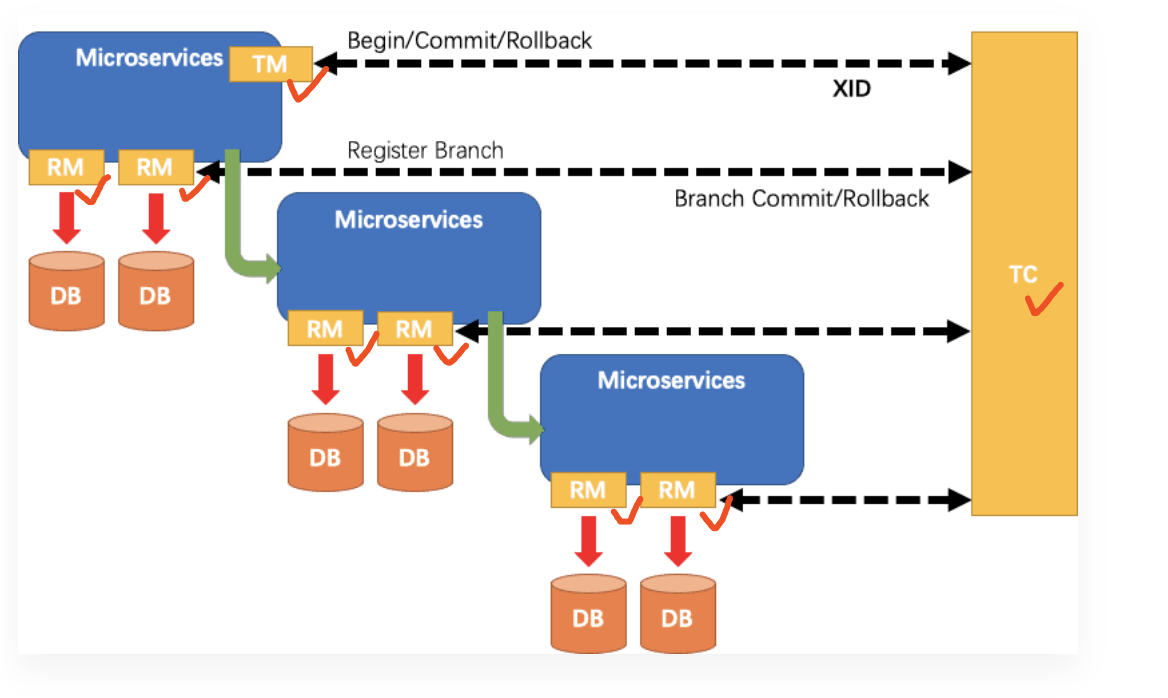
定义全局事务的范围：开始全局事务、提交或回滚全局事务；

RM (Resource Manager) - 资源管理器

管理分支事务处理的资源，与TC交互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；

其中TC为单独部署的 Server 服务端，TM和RM为嵌入到应用中的 Client 客户端；

在Seata中，一个分布式事务的生命周期如下：



TM请求TC开启一个全局事务，TC会生成一个XID作为该全局事务的编号，XID会在微服务的调用链路中传播，保证将多个微服务的子事务关联在一起；

RM请求TC将本地事务注册为全局事务的分支事务，通过全局事务的XID进行关联；

TM请求TC告诉XID对应的全局事务是进行提交还是回滚；

TC驱动RM将XID对应的自己的本地事务进行提交还是回滚；

## TC Server运行环境部署

我们先部署单机环境的 Seata TC Server，用于学习或测试，在生产环境中要部署集群环境；

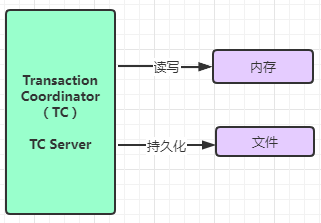
因为TC需要进行全局事务和分支事务的记录，所以需要对应的存储，目前，TC有三种存储模式( store.mode )：

file模式：适合单机模式，全局事务会话信息在内存中读写，并持久化本地文件 root.data，性能较高；

db模式：适合集群模式，全局事务会话信息通过 db 共享，相对性能差点；

redis模式：解决db存储的性能问题；

我们先采用file模式，最终我们部署单机TC Server如下图所示：

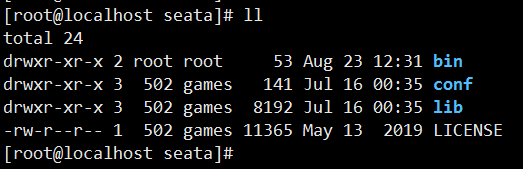


## Seata运行环境部署

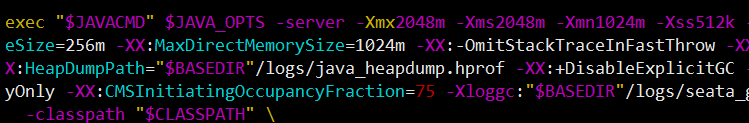
下载Seata：<http://seata.io/zh-cn/blog/download.html>

解压：tar -zxvf seata-server-1.3.0.tar.gz

切换cd seata



默认seata-server.sh脚本设置的jvm内存参数2G，我们再虚拟机里面做实验，可以改小一点；



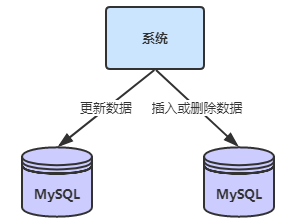
在bin目录下启动：./seata-server.sh

默认配置下，Seata TC Server 启动在 8091 端口；

因为我们没有修改任何配置文件，默认情况seata使用的是file模式进行数据持久化，所以可以看到用于持久化的本地文件 root.data；

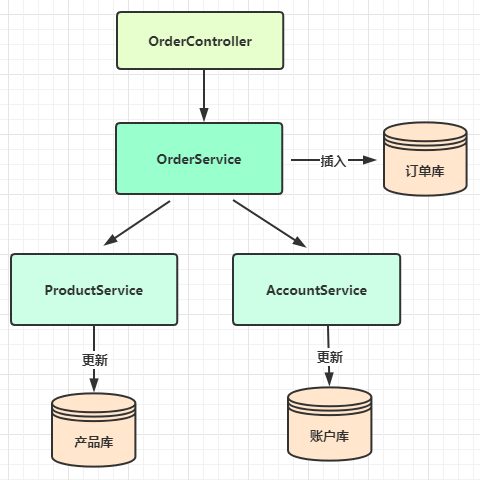
## AT模式事务案例

### 单体应用多数据源分布式事务



在Spring Boot单体项目中，如果使用了多数据源，就需要考虑多个数据源的数据一致性，即产生了分布式事务的问题，我们采用Seata的AT事务模式来解决该分布式事务问题；

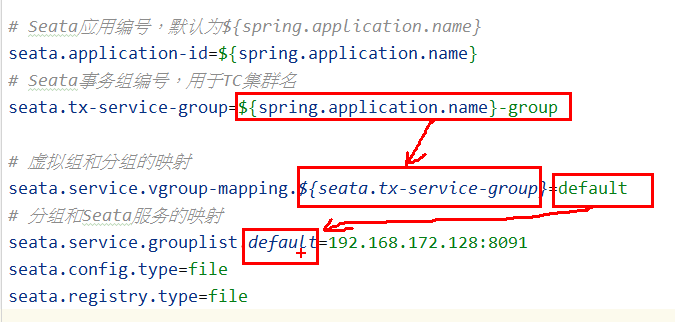
以电商购物下单为例：



1. 准备数据库表和数据；

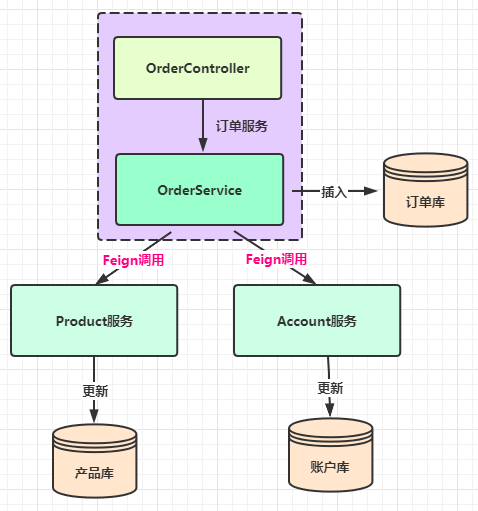
其中每个库中的undo\_log表，是 Seata AT模式必须创建的表，主要用于分支事务的回滚；

1. 开发一个SpringBoot单体应用；



测试：http://localhost:8080/order?userId=1&productId=1

### 微服务的分布式事务



## AT事务模式分布式事务工作机制

**前提**

基于支持本地 ACID 事务的关系型数据库；（mysql、oracle）

Java 应用，通过JDBC访问数据库；

**整体机制**

就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；

二阶段：

如果没有异常异步化提交，非常快速地完成；

如果有异常回滚通过一阶段的回滚日志进行反向补偿；

具体举例说明整个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

3、执行业务 SQL：更新这条记录的 name 为 'GTS'；

4、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

5、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；



6、分支事务提交前，向TC注册分支，申请product表中，主键值等于1的记录的全局锁（在当前的同一个全局事务id范围内是可以申请到全局锁的，不同的全局事务id才会排斥）；

7、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；

8、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 中的后镜像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，需要人工来处理；

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;

5、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；

回滚日志表：

**Field Type**

branch\_id bigint PK

xid varchar(100)

context varchar(128)

rollback\_info longblob

log\_status tinyint

log\_created datetime

log\_modified datetime

SQL建表语句：

CREATE TABLE `undo\_log` (

`id` bigint NOT NULL AUTO\_INCREMENT,

`branch\_id` bigint NOT NULL,

`xid` varchar(100) NOT NULL,

`context` varchar(128) NOT NULL,

`rollback\_info` longblob NOT NULL,

`log\_status` int NOT NULL,

`log\_created` datetime NOT NULL,

`log\_modified` datetime NOT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `ux\_undo\_log` (`xid`,`branch\_id`)

) ENGINE=InnoDB AUTO\_INCREMENT=4 DEFAULT CHARSET=utf8;