彻底解决分布式事务

# 分布式事务产生的背景

在微服务环境下，因为会根据不同的业务会拆分成不同的服务，比如会员服务、订单服务、商品服务等，让专业的人做专业的事情，每个服务都有自己独立的数据库，并且是独立运行，互不影响。

服务与服务之间通讯采用RPC远程调用技术，但是每个服务中都有自己独立的数据源，即自己独立的本地事务。两个服务相互通讯的时候，两个本地事务互不影响，从而出现分布式事务产生的原因。

举例说明:

# 解决分布式事务基本思路

## ACID酸碱平衡理论

　如何保证强一致性呢？计算机专业的童鞋在学习关系型数据库的时候都学习了ACID原理，这里对ACID做个简单的介绍。如果想全面的学习ACID原理，请参考ACID  
　　关系型数据库天生就是解决具有复杂事务场景的问题，关系型数据库完全满足ACID的特性。

数据库管理系统中事务(transaction)的四个特性（分析时根据首字母缩写依次解释）：

原子性（Atomicity）

一致性（Consistency）

隔离性（Isolation）

持久性（Durability）

所谓事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。（执行单个逻辑功能的一组指令或操作称为事务）

## CAP（帽子 原理）

由于对系统或者数据进行了拆分，我们的系统不再是单机系统，而是分布式系统，针对分布式系统的CAP原理包含如下三个元素。

C:Consistency,致性。在分布式系统中的所有数据 备份，在同一时刻具有同样的值，所有节点在同一时刻读取的数据都是最新的数据副本。

A:Availability,可用性，好的响应性能。完全的可用性指的是在任何故障模型下，服务都会在有限的时间内处理完成并进行响应。

P: Partition tolerance,分区容忍性。尽管网络上有部分消息丢失，但系统仍然可继续工作。

CAP原理证明，任何分布式系统只可同时满足以上两点，无法三者兼顾。由于关系型数据库是单节点无复制的，因此不具有分区容忍性，但是具有一致性和可用性，而分布式的服务化系统都需要满足分区容忍性，那么我们必须在一致性和可用性之间进行权衡。如果在网络上有消息丢失，也就是出现了网络分区，则复制操作可能会被延后，如果这时我们的使用方等待复制完成再返回，则可能导致在有限时间内无法返回，就失去了可用性:而如果使用方不等待复制完成，而在主分片写完后直接返回，则具有了可用性，但是失去了一致性。

## Base（碱）

BASE 是 Basically Available（基本可用）、Soft state（软状态）和 Eventually consistent（最终一致性）三个短语的简写，由 eBay 架构师 Dan Pritchett 于 2008 年在《BASE: An Acid Alternative》（论文地址点 [这里](https://www.iteblog.com/redirect.php?url=aHR0cDovL2RlbGl2ZXJ5LmFjbS5vcmcvMTAuMTE0NS8xNDAwMDAwLzEzOTQxMjgvcDQ4LXByaXRjaGV0dC5wZGY=&article=true)）论文中首次提出。BASE 思想与 ACID 原理截然不同，它满足 CAP 原理，通过牺牲强一致性获得可用性， 一般应用于服务化系统的应用层或者大数据处理系统中，通过达到最终一致性来尽量满足业务的绝大多数需求。  
BASE 模型包含如下三个元素：

* BA：（***B***asically ***A***vailable ），基本可用。
* S：（ ***S***oft State），软状态，状态可以在一段时间内不同步。
* E：（***E***ventually Consistent ），最终一致，在一定的时间窗口内， 最终数据达成一致即可。

关于最终一致的几种变种参见上面，在实际系统实践中，可以将若干变种结合起来，来实现各种业务需求。

# 柔性事务和刚性事务

柔性事务满足BASE理论（基本可用，最终一致）

刚性事务满足ACID理论

本文主要围绕分布式事务当中的柔性事务的处理方式进行讨论。

柔性事务分为

1. 两阶段型

2. 补偿型

3. 异步确保型

4. 最大努力通知型几种。 由于支付宝整个架构是SOA架构，因此传统单机环境下数据库的ACID事务满足了分布式环境下的业务需要，以上几种事务类似就是针对分布式环境下业务需要设定的。

# 分布式事务常见解决方案

## 分布式一致性协议

### XA接口

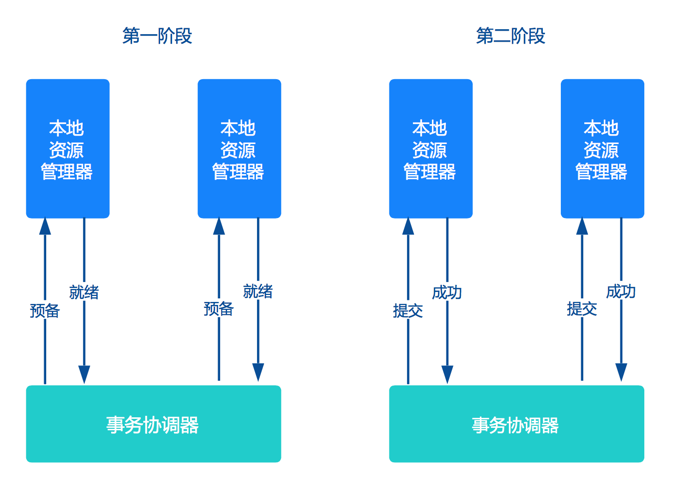
 XA是由X/Open组织提出的分布式事务的规范。XA规范主要定义了(全局)事务管理器(Transaction Manager)和(局部)资源管理器(Resource Manager)之间的接口。XA接口是双向的系统接口，在事务管理器（Transaction Manager）以及一个或多个资源管理器（Resource Manager）之间形成通信桥梁。XA之所以需要引入事务管理器是因为，在分布式系统中，从理论上讲（参考Fischer等的论文），两台机器理论上无法达到一致的状态，需要引入一个单点进行协调。事务管理器控制着全局事务，管理事务生命周期，并协调资源。资源管理器负责控制和管理实际资源（如[数据库](http://lib.csdn.net/base/mysql)或JMS队列）

### Jta规范

作为java平台上事务规范JTA（Java Transaction API）也定义了对XA事务的支持，实际上，JTA是基于XA架构上建模的，在JTA 中，事务管理器抽象为javax.transaction.TransactionManager接口，并通过底层事务服务（即JTS）实现。像很多其他的java规范一样，JTA仅仅定义了接口，具体的实现则是由供应商(如J2EE厂商)负责提供，目前JTA的实现主要由以下几种：  
1.J2EE容器所提供的JTA实现(JBoss)  
2.独立的JTA实现:如JOTM，Atomikos.这些实现可以应用在那些不使用J2EE应用服务器的环境里用以提供分布事事务保证。如Tomcat,Jetty以及普通的java应用。

## 两段提交协议

交易中间件与数据库通过 XA 接口规范，使用两阶段提交来完成一个全局事务， XA 规范的基础是两阶段提交协议。  
第一阶段是表决阶段，所有参与者都将本事务能否成功的信息反馈发给协调者；第二阶段是执行阶段，协调者根据所有参与者的反馈，通知所有参与者，步调一致地在所有分支上提交或者回滚。



两阶段提交方案应用非常广泛，几乎所有商业OLTP数据库都支持XA协议。但是两阶段提交方案锁定资源时间长，对性能影响很大，基本不适合解决微服务事务问题。

## 三段提交协议

## TCC

## 异步回调模式

## 最终一致性模式

## 可靠消息模式

# 基于LCN框架解决分布式事务

LCN官网 <https://www.txlcn.org/>

"LCN并不生产事务，LCN只是本地事务的搬运工"

兼容 dubbo、springcloud、motan 框架，支持各种关系型数据库

# LCN框架底层实现原理

详细参考: https://github.com/codingapi/tx-lcn/wiki/LCN%E5%8E%9F%E7%90%86

# SpringCloud2.0整合LCN

目前LCN版本已经升级为4.0了，但是官方没有SpringCloud2.0的demo案例。

因为LCN本身是开源的，网上有大牛对LCN框架源码做修改，可以支持SpringCloud2.0版本。

# SpringCloud2.0客户端集成

## Maven依赖信息

|  |
| --- |
| <dependency>  <groupId>com.codingapi</groupId>  <artifactId>transaction-springcloud</artifactId>  <version>4.1.2</version>  <exclusions>  <exclusion>  <groupId>org.slf4j</groupId>  <artifactId>\*</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>com.codingapi</groupId>  <artifactId>tx-plugins-db</artifactId>  <version>4.1.2</version>  <exclusions>  <exclusion>  <groupId>org.slf4j</groupId>  <artifactId>\*</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency> |

## application.yml引入

|  |
| --- |
| tm:  manager:  url: http://127.0.0.1:8899/tx/manager/ |

## LCN基本配置代码

|  |
| --- |
| @Service  **public** **class** TxManagerTxUrlServiceImpl **implements** TxManagerTxUrlService {  @Value("${tm.manager.url}")  **private** String url;  @Override  **public** String getTxUrl() {  System.***out***.println("load tm.manager.url ");  **return** url;  }  }  @Service  **public** **class** TxManagerHttpRequestServiceImpl **implements** TxManagerHttpRequestService {  @Override  **public** String httpGet(String url) {  System.***out***.println("httpGet-start");  String res = HttpUtils.*get*(url);  System.***out***.println("httpGet-end");  **return** res;  }  @Override  **public** String httpPost(String url, String params) {  System.***out***.println("httpPost-start");  String res = HttpUtils.*post*(url, params);  System.***out***.println("httpPost-end");  **return** res;  }  } |

## 分布式事务案例

|  |
| --- |
| // 下单扣库存  @TxTransaction(isStart = **true**)  @Transactional  @GetMapping(value = "/addOrderAndStock")  **public** ResponseBase addOrderAndStock(**int** i) {  OrderEntity orderEntity = **new** OrderEntity();  orderEntity.setName("蚂蚁课堂永久会员充值");  orderEntity.setOrderCreatetime(**new** Date());  // 价格是300元  orderEntity.setOrderMoney(300d);  // 状态为 未支付  orderEntity.setOrderState(0);  Long commodityId = 30l;  // 商品id  orderEntity.setCommodityId(commodityId);  // 1.先下单，创建订单  **int** orderResult = orderMapper.addOrder(orderEntity);  System.***out***.println("orderResult:" + orderResult);  // 2.下单成功后,调用库存服务  ResponseBase inventoryReduction = stockFeign.inventoryReduction(commodityId);  // if (inventoryReduction.getRtnCode() != 200) {  // // 手动回滚事务  // }  **int** reuslt = 1 / i;  **return** setResultSuccess("下单成功!");  } |

使用@TxTransaction解决分布式事务 isStart true 是:是发起方 false 否:是参与方

同理生产者 同样配置集成即可，只需要把Transaction 中的isStart改为fasle即可。