Gmall

分布式事务

# 简介

分布式事务是企业集成中的一个技术难点，也是每一个分布式系统架构中都会涉及到的一个东西，特别是在微服务架构中，几乎可以说是无法避免。

# 二、数据库事务

数据库事务的几个特性：原子性(Atomicity )、一致性( Consistency )、隔离性或独立性( Isolation)和持久性(Durabilily)，简称就是ACID。

分布式的网络环境很复杂，容易出现的问题：

机器宕机、**网络异常**、消息丢失、消息乱序、数据错误、不可靠的TCP、存储数据丢失...

# 三、分布式系统CAP与BASE

## CAP定理

CAP定理是由加州大学伯克利分校Eric Brewer教授提出来的，他指出WEB服务无法同时满足一下3个属性：

一致性(Consistency) ： 客户端知道一系列的操作都会同时发生(生效)

可用性(Availability) ： 每个操作都必须以可预期的响应结束

分区容错性(Partition tolerance) ： 即使出现单个组件无法可用,操作依然可以完成

## BASE理论

在分布式系统中，我们往往追求的是可用性，它的重要程序比一致性要高，那么如何实现高可用性呢？ 前人已经给我们提出来了另外一个理论，就是BASE理论，它是用来对CAP定理进行进一步扩充的。BASE理论指的是：

Basically Available（基本可用）

Soft state（软状态）

Eventually consistent（最终一致性）

**BASE**理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：**我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。**

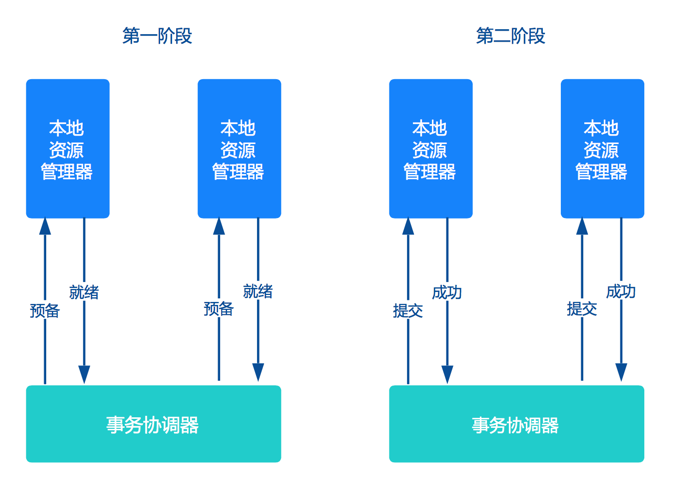
# 四、分布式事务解决方案

## 1、基于XA协议的两阶段提交方案

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。

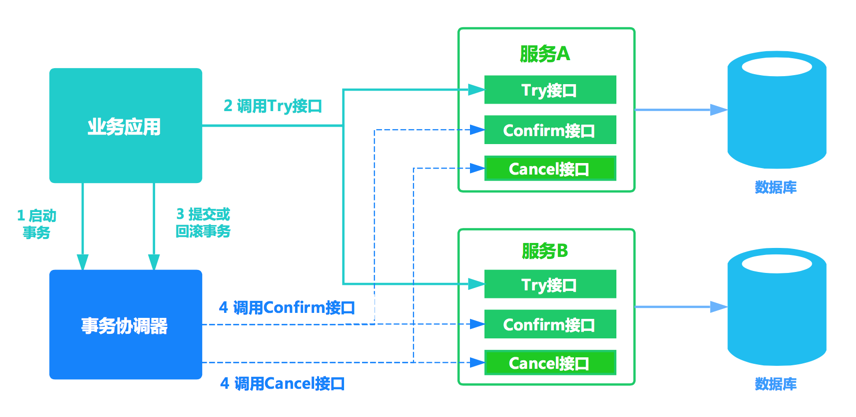
第一阶段是表决阶段，所有参与者都将本事务能否成功的信息反馈发给协调者；

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。



## 2、TCC方案

TCC方案在电商、金融领域落地较多。TCC方案其实是两阶段提交的一种改进。其将整个业务逻辑的每个分支显式的分成了Try、Confirm、Cancel三个操作。Try部分完成业务的准备工作，confirm部分完成业务的提交，cancel部分完成事务的回滚。基本原理如下图所示。



//本地事务记录表；记录哪次操作，哪些表发生了哪些改变。

a-try(){

//尝试去数据库减10 log(1,-10);

}

a-confirm(){

//确认真的减10块，提交事务 拿到上次写的日志，去执行这个操作，1，-10，并且提交

}

a-cancle(){

//再去数据库取消这次操作。反向+10，拿到上次日志，反向操作，+10；

}

--------------------------------------------------------------------------

b-try(){

//尝试去加10块。 log(2,+10)

}

b-confirm(){

//确认这次修改加10块。拿到日志，给2号+10，并提交了。

//dhahjdahjdhakxx xxx 失败了，通知事务协调器，我失败了。

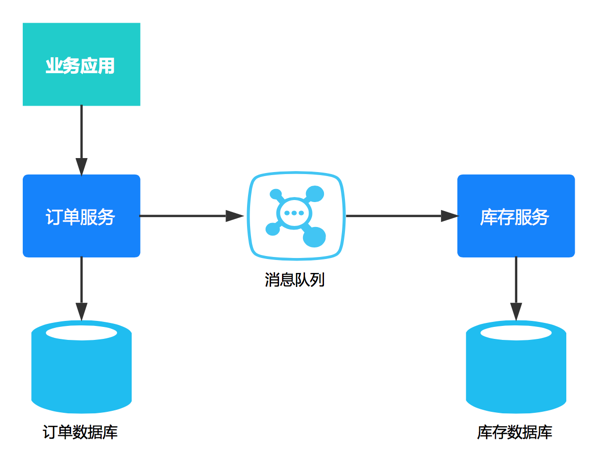
}

b-cancle(){

//减10块，拿到上次操作，反向-10

}

## 3、基于消息的最终一致性方案



## 4、阿里的GTS

记录数据库变化；回改数据库。

在业务函数外围使用@TxcTransaction注解即可开启分布式事务。Dubbo应用通过隐藏参数将GTS的事务xid传播到服务端。

|  |
| --- |
| **@TxcTransaction(timeout = 1000 \* 10)**  public void Bussiness(OrderService orderService, StockService stockService, String userId) {  //获取事务上下文  String xid = TxcContext.getCurrentXid();  //通过RpcContext将xid传到一个服务端  RpcContext.getContext().setAttachment("xid", xid);    //执行自己的业务逻辑  int productId = new Random().nextInt(100);  int productNum = new Random().nextInt(100);  OrderDO orderDO = new OrderDO(userId, productId, productNum, new Timestamp(new Date().getTime()));  orderService.createOrder(orderDO);    //通过RpcContext将xid传到另一个服务端  RpcContext.getContext().setAttachment("xid",xid);  stockService.updateStock(orderDO);  } |
| public int updateStock(OrderDO orderDO) {  //获取全局事务ID，并绑定到上下文  String xid = RpcContext.getContext().getAttachment("xid");  TxcContext.bind(xid,null);  //执行自己的业务逻辑  int ret = jdbcTemplate.update("update stock set amount = amount - ? where product\_id = ?",new Object[]{orderDO.getNumber(), orderDO.getProductId()});  TxcContext.unbind();  return ret;  } |

## 5、最佳实践

分布式事务的两个原因；

服务节点分布过多

资源（db）阶段分散

其中有个原因是因为微服务太多。太多团队一个人维护几个微服务，过度设计，搞得所有人都很疲惫。

微服务多就会引出分布式事务，这个时候不会建议任何一种分布式事务解决方案，而是将这些服务聚合成一个**单机服务**，使用数据库的本地事务。因为无论什么方案都会增加系统的复杂度和不可靠性。

分布式事务应该是我们去**积极避免**的，而不是去努力拥抱的。

一句话，**能不用就不用**。