分布式事务的应用和探讨

# 1传统架构事务

事务的主要作用是数据的一致性，也是CAP理论中的C的要求，避免出现脏数据，在程序中，事务的控制主要分为**编码式(侵入性)控制和声明式的事务控制。**

**编码式事务控制流程主要为：**在业务中进行try -catch 处理，业务方法执行没有异常则提交事务，出现异常则在catch中进行回滚。

public void test() {

Connection connection = null;

try {

connection = getConnection();

connection.setAutoCommit(false);

// 你的业务方法

connection.commit();

} catch (Exception e) {

connection.rollback();

}finally {

connection.close();

}

}

如果采用编程式事务，那么在任何需要事务的地方都要开启事务、try-catch、提交或者回滚事务，会导致重复编码、编写与业务无关的代码。基于Spring AOP的思想，我们可以利用AOP的方式，对需要使用事务的方法进行增强，将公用的部分提取出来，这种模式就是申明式事务控制。在Spring boot单体的架构中，在需要事务控制的业务方法上使用@Transactional注解就能实现声明式事务控制。

# 2 分布式架构事务

传统的单体架构下，我们可以很轻松地实现关系型数据库的事务控制，但在分布式架构下，一个请求调用链可能会经过多个服务，操作多个数据库。这种情况下，事务被分散到了各个子系统中，分布式架构事务（下面简称分布式事务）问题显得尤为重要，特别是在金融电商领域，如果分布式事务没控制好，损失的就是是真金白银。

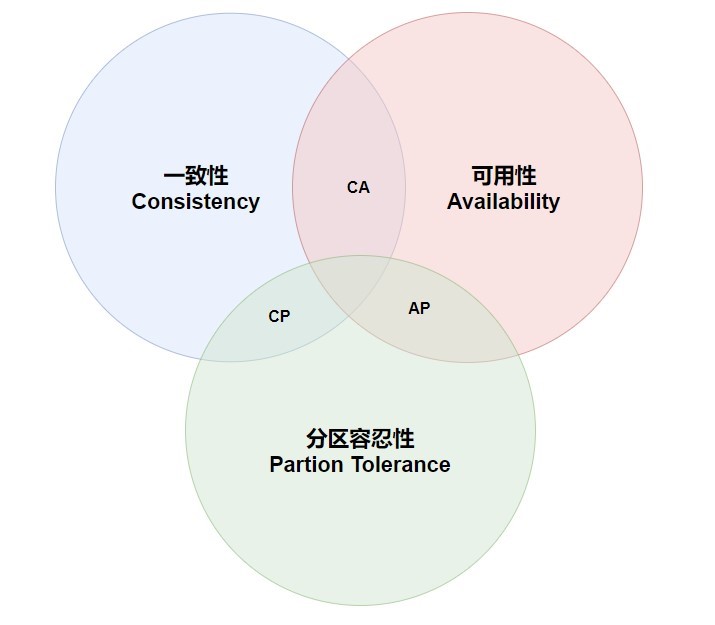
数据库的数据一致性强度可分为三种：

1. 强一致性：要求数据库中更新的数据立即对外可见。
2. 弱一致性：能够容忍数据库中部分更新后的数据，对外不可见。
3. 最终一致性：要求数据库中更新后的数据在一段时间后（这个时间区间的长度主要受网络通信延迟，系统负载等因素的影响），对外可见。

在本地事务中，关系型数据库的ACID（原子性 - Atomicity、一致性 - Consistency、隔离性 - Isolation、持久性 - Durability）特性可以确保数据的强一致性，但要让分布式环境下的数据任何时刻都保持强一致性是不可能的，我们只能根据实际业务采用折衷的方案，确保数据最终一致性即可。

分布式事务控制可以参考一些著名的理论来支撑：CAP理论和BASE理论。

**CAP理论**指的是分布式系统的三个特性，一致性（Consistency）、可用性（Availability）、分区容错性（Partition tolerance）：



|  |  |
| --- | --- |
| **选择** | **说明** |
| CA | 选择一致性和可用性放弃了分区容错性 |
| CP | 选择一致性和分区容错性，放弃了可用性 |
| AP | 选择可用性和分区容错性，放弃了（强一致性） |

分布式系统中，服务被分散到各个服务器上，服务之间的调用都是通过网络通信完成，所以网络通信问题是我们必须面对的问题。换句话说，分区容错性是一个分布式系统必然需要面对和解决的问题。因此我们只能在AP和CP上进行选择，实际业务中，我们更多的是在C（一致性）和A（可用性）之间寻求平衡。基于这个特点，业界中又衍生出了BASE理论。

**BASE理论**是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写，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。基于BASE理论，又衍生出了不同于ACID的刚性事务的**柔性事务**概念，基于这些理论，业界提出了相应的分布式事务解决方案

**分布式事务常见解决方案：**

（1）2PC两段提交协议

（2）3PC 三段提交协议（弥补两段提交协议缺点）

（3）TCC或者GTS（阿里解决方案）

（4）消息中间件最终一致性

## 2.1两阶段提交（2PC）

两阶段提交又称**2PC**,2PC是一个非常经典的强一致、中心化的原子提交协议。

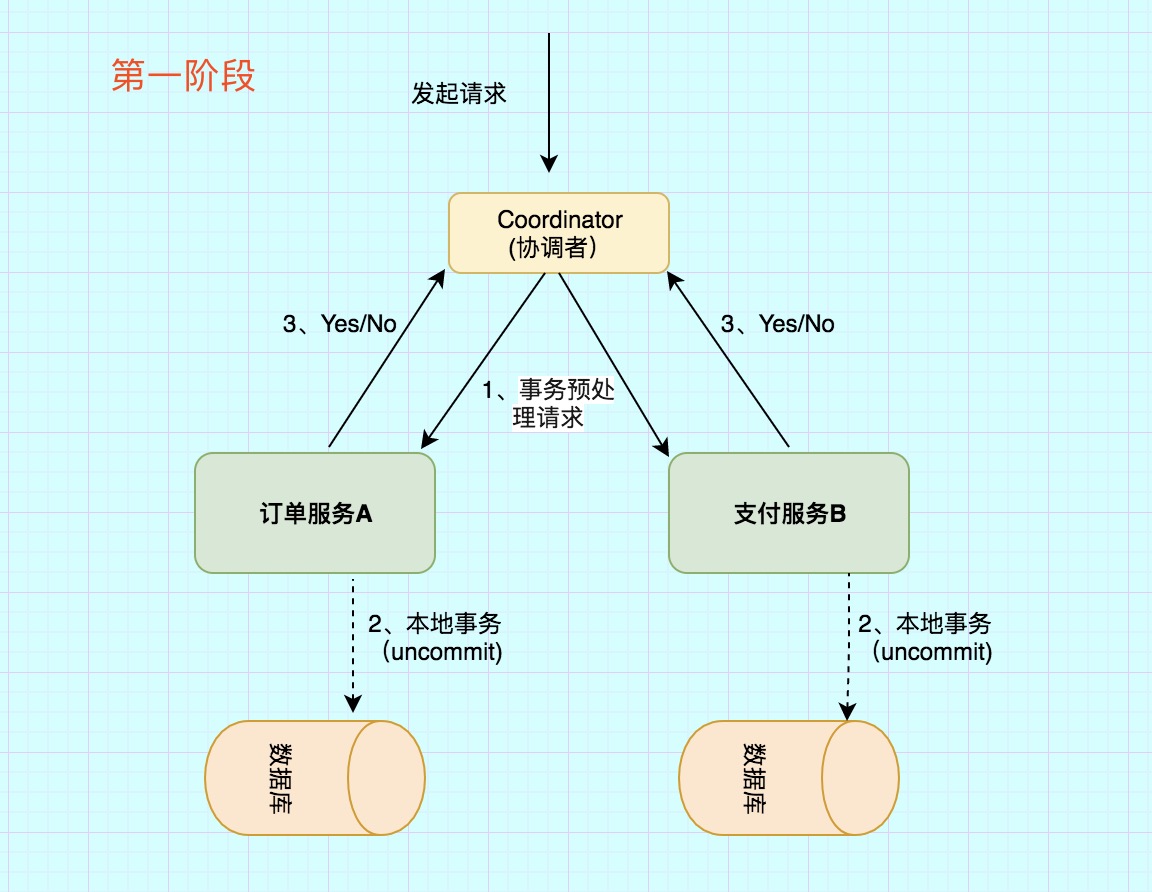
这里所说的中心化是指协议中有两类节点：一个是中心化协调者节点（coordinator）和N个参与者节点（partcipant）。

两个阶段：第一阶段：**投票阶段** 和第二阶段：**提交/执行阶段**。

举例 **订单服务A**，需要调用 **支付服务B** 去支付，支付成功则处理消费订单为待**发货状态**，否则就需要将购物订单处理为失败状态。

2PC处理流程：

1. **投票阶段**



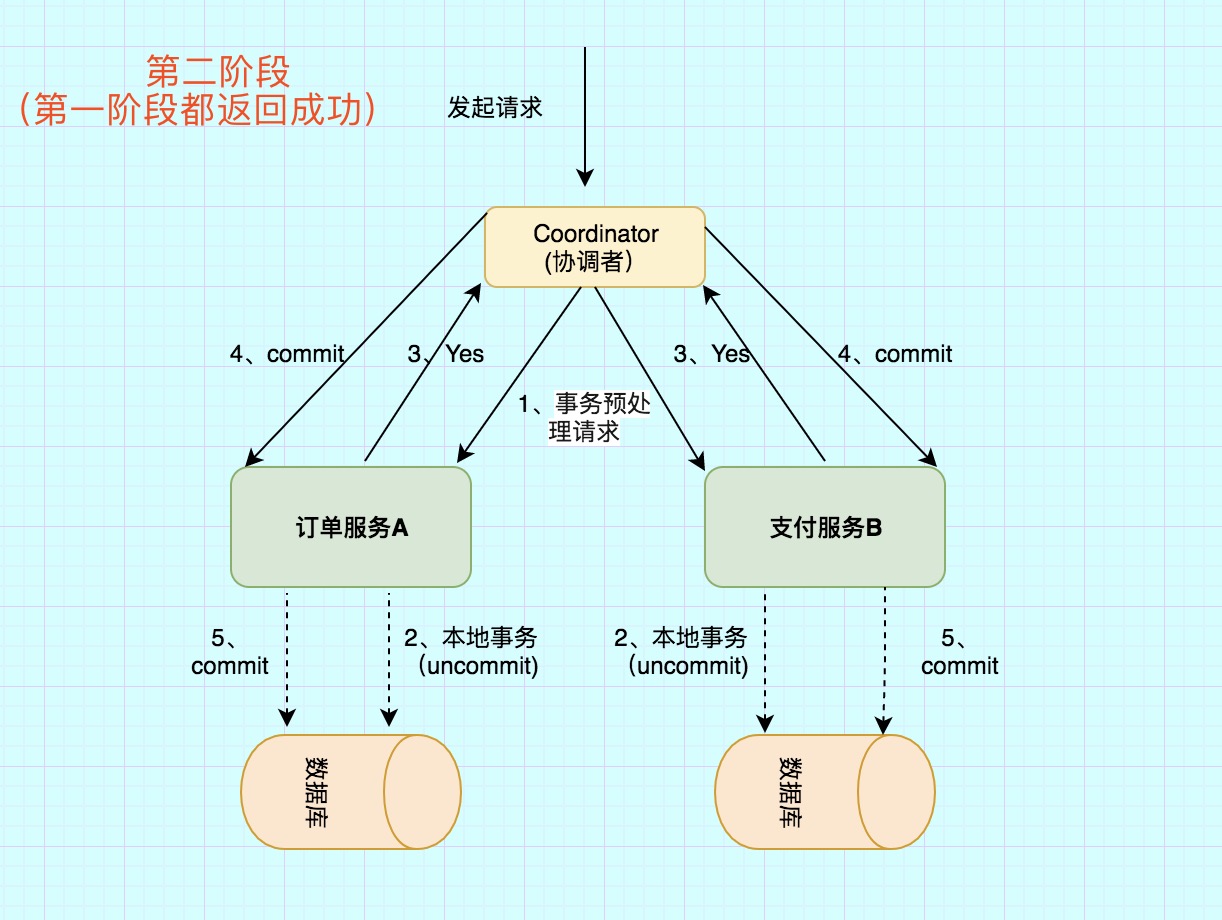
第一阶段可分为三个步骤：

1. 事务询问：协调者向所有的参与者发送事务预处理请求（Prepare），开始等待参与者的响应。
2. 执行本地事务：各参与者（节点）执行本地事务操作，但在执行完成后（write log）并不会真正提交本地事务。
3. 各参与者向协调者反馈事务的询问的响应：

如果 **参与者** 成功执行了事务操作,那么就反馈给协调者 **Yes** 响应,表示事务可以执行,如果没有 **参与者** 成功执行事务,那么就反馈给协调者 **No** 响应,表示事务不可以执行。

第一阶段执行完后，会有两种可能。1、所有都返回Yes. 2、有一个或者多个返回No。

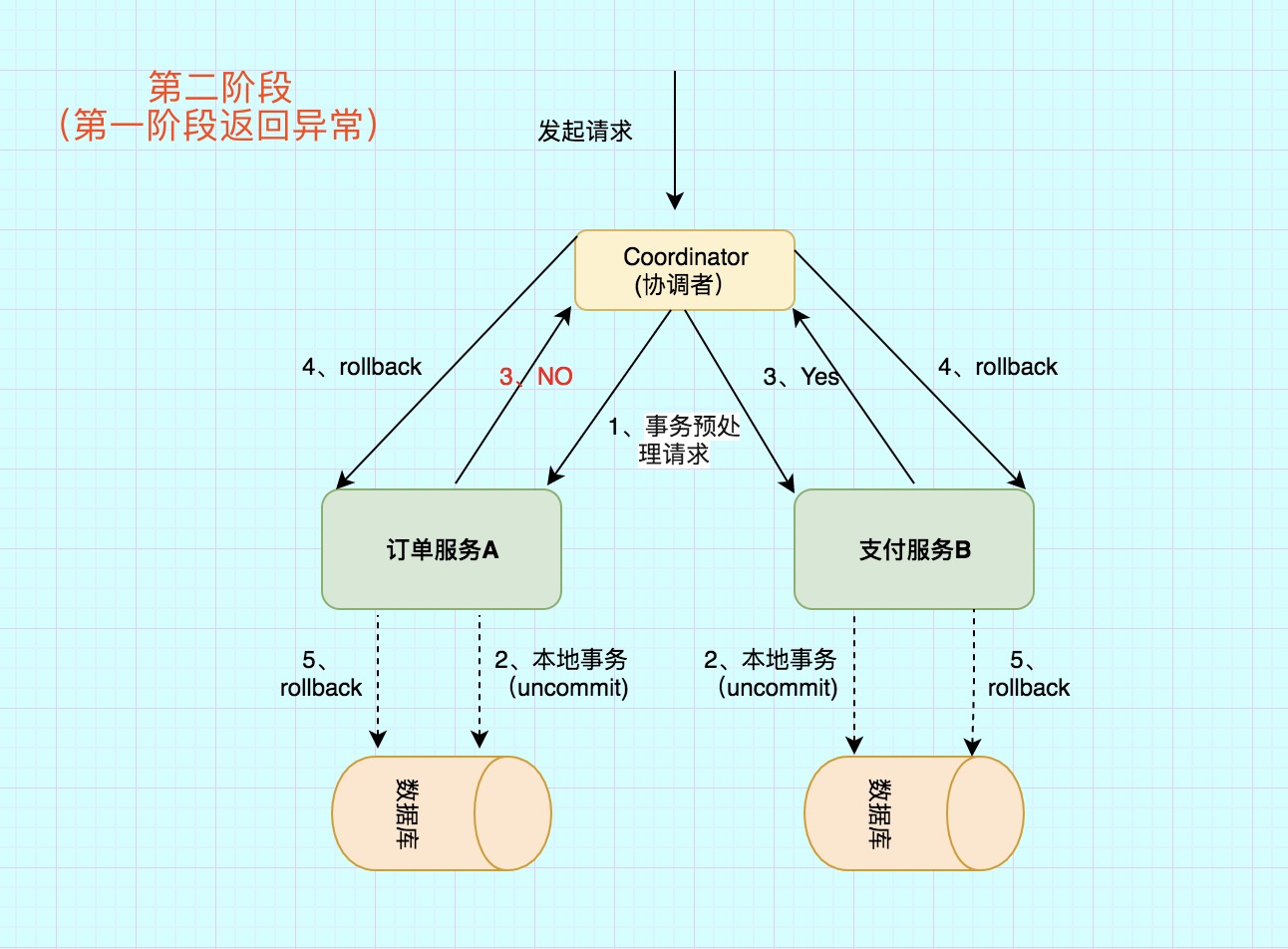
1. **提交/执行阶段(成功流程)**



**第二阶段可分为两步：**

1. 协调者向所有参与者节点发出commit 请求
2. 事务提交：参与者收到commit请求之后，就会正式执行本地事务commit操作，并在完成提交之后释放整个事务执行期间占用的事务资源。
3. **提交/执行阶段(异常流程)**

异常条件：任何一个参与者向协调者反馈了NO响应，或者等待超时之后，协调者尚未收到所有参与者的响应。



异常流程第二阶段也可分为两步：

1. 发送回滚请求：协调者向所有参与者发送roollback请求
2. 事务回滚：参与者接收到roolback请求后，会回滚本地事务

**（4）2PC 缺点**

a、性能问题：

无论是在第一阶段的过程中,还是在第二阶段,所有的参与者资源和协调者资源都是被锁住的,只有当所有节点准备完毕，事务 协调者 才会通知进行全局提交，参与者 进行本地事务提交后才会释放资源。这样的过程会比较漫长，对性能影响比较大。

b、单点故障

## 2.2 三阶段提交（3PC）

三阶段提交协议（3PC）主要是为了解决两阶段提交协议的阻塞问题，2pc存在的问题是当协作者崩溃时，参与者不能做出最后的选择。因此参与者可能在协作者恢复之前保持阻塞。三阶段提交（Three-phase commit），是二阶段提交（2PC）的改进版本。

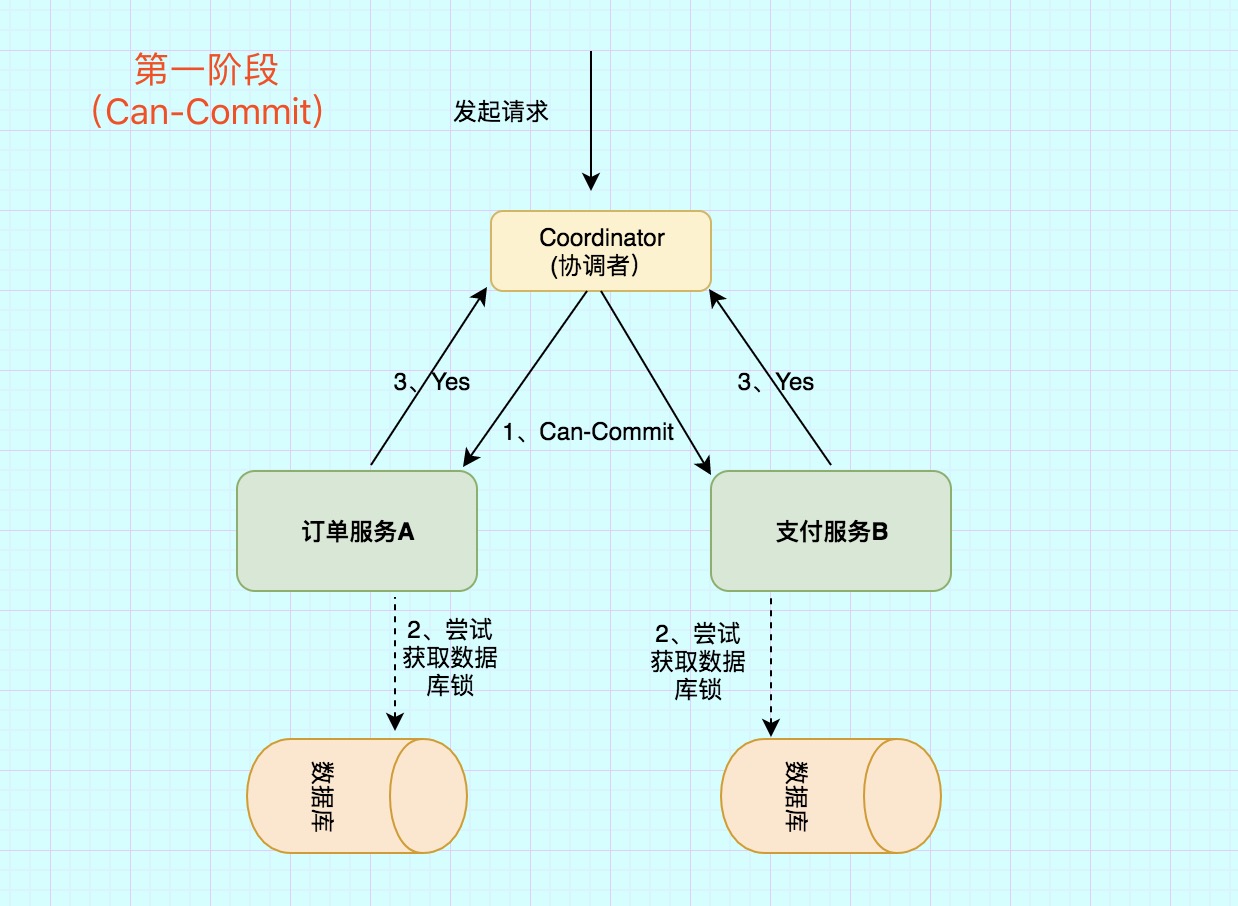
与两阶段提交不同的是，三阶段提交有两个改动点。

1. 引入超时机制，同时在协调者和参与者中都引入超时机制。
2. 在第一阶段和第二阶段中间插入一个准备阶段，保证了在最后提交阶段之前，各参与者的节点状态是一致的。

也就是说，除了引入超时机制之外，**3PC把2PC的准备阶段再次一分为二**，这样三阶段提交就有 “canCommit”、 “PreCommit”、”doCommit” 三个阶段。

1. **CanCommit 阶段**

之前2PC的一阶段是本地事务执行结束后，最后不Commit,等其它服务都执行结束并返回Yes，由协调者发生commit才真正执行commit。而这里的CanCommit指的是 **尝试获取数据库锁** 如果可以，就返回Yes。



这阶段主要分为2步：

1. 事务询问：协调者向参与者发送CanCommit请求，询问是否可以执行事务提交操作，然后开始等待参与者的响应，
2. 参与者接收到CanCommit请求之后，正常情况下，如果其自身认为可以执行事务，则返回YES响应，并进入预备状态，否则反馈NO。
3. **PreCommit阶段**

在阶段一中，如果所有的参与者都返回YES的话，那么久会进入PreCommit阶段进行事务预提交，这里的PreCommit阶段跟上面的第一阶段类似，只不过这里的协调者和参与者都引入了超时机制（2PC中只有协调者可以超时，参与者没有超时机制）

1. DoCommit阶段

这里跟2PC的二阶段差不多的。

**总结：**

相比较2PC而言，3PC对于协调者（**Coordinator**）和参与者（**Partcipant**）都设置了超时时间，而2PC只有协调者才拥有超时机制。这解决了一个什么问题呢？

这个优化点，主要是避免了参与者在长时间无法与协调者节点通讯（协调者挂掉了）的情况下，无法释放资源的问题，因为参与者自身拥有超时机制会在超时后，

自动进行本地commit从而进行释放资源。而这种机制也侧面降低了整个事务的阻塞时间和范围。另外，通过**CanCommit、PreCommit、DoCommit**三个阶段的设计，相较于2PC而言，多设置了一个**缓冲阶段**保证了在最后提交阶段之前各参与节点的状态是一致的。

以上就是3PC相对于2PC的一个提高（相对缓解了2PC中的前两个问题），但是3PC依然没有完全解决数据不一致的问题。

## 2.3 分布式事务（TCC）

### 2.3.1 Tcc补偿事务

Tcc又称补偿事务，其核心思想是：“针对每个操作都注册一个与其相对应的确认和补偿（撤销操作）”，它分为三个步骤：

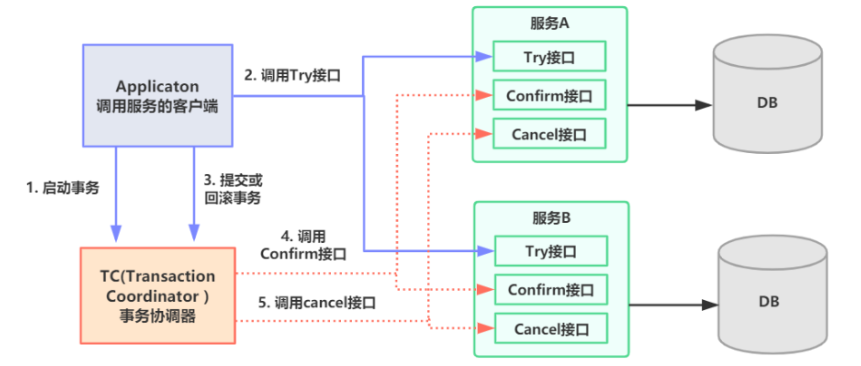
1. try阶段，主要是对业务系统做检测及资源预留。
2. confirm阶段，确认执行业务操作。
3. cancel 确认执行业务操作。

Try: Try操作是先把多个应用中的业务资源预留和锁住，为后续的确认打下基础，类似的，DML操作锁定数据库记录行、持有数据库资源。

Confirm : Confirm 操作是在try操作之中涉及的所有应用成功之后进行确认，使用预留的业务资源和confirm类似。

Cancel：Cancel 则是当try 操作中涉及的所有的应用没有全部成功，需要将已经成功的应用取消（即：Roollback回滚）其中Confirm 和 Cancel 是反向操作。

TCC的具体原理如图：



分布式事务协调者：分布式事务协调者管理控制整个业务活动，包括记录维护TCC事务的全局状态和每个从业服务的子事务状态，并在业务活动提交时确认所有的TCC型。

### 举例 2.3.2

例如：A服务专30元钱、B服务转50元，一起到C服务上。

**Try：**尝试执行业务，完成所有业务的检查（一致性）： 检查A、B、C账户的状态是否正常，账户A的余额是否不少于30元，账户B的余额是否不少于50元。预留必须的业务资源。

准隔离性：账户A的冻结金额增加30元，账户B的冻结金额增加50元，这样就保证了其他线程的闯入扣减了两个账户余额，导致在后续的操作中A、B账户的余额不足。

**Confirm:** 确认执行业务。真正的执行业务：如果Try 阶段A、B、C账户状态正常，且A、B账户余额也够用，则执行账户A给账户C转账30元，账户B给账户C转账50元。

Ps：

**这时已经不需要做任何业务检查，Try阶段已经完成了业务检查**。**只使用Try阶段预留的业务资源：只需要使用Try阶段帐户A和帐户B冻结的金额即可。**

**Cancel**: 取消执行业务释放Try阶段预留的业务资源：如果Try阶段部分成功，比如帐户A的余额够用，且冻结相应金额成功，帐户B的余额不够而冻结失败，则需要对账户A做Cancel

操作，将把账户A冻结的相应金额解冻。

### 2.3.2 Tcc 和 2PC比较

2PC是资源层面的分布式事务，强一致性，在两阶段提交的整个过程中，一直会持有资源的锁。

XA事务中的两阶段提交内部过程是对开发者屏蔽的，事务管理器在两阶段提交过程中，从prepare到commit/rollback过程中，资源实际上一直都是被加锁的。

**TCC是业务层面的分布式事务，最终一致性，不会一直持有资源的锁。**

我的理解就是当执行try接口的时候，已经把所需的资源给预扣了，比如上面举例的A服务已经预扣30元，B服务已经预扣50元，它是由try接口实现，这样就保证不会出现其他并发线程扣减了这两个帐户的余额而导致在后续的真正转帐操作过程中，帐户A和B的可用余额不够的情况，同时保证不会一直锁住整个资源。（核心点应该就在这）

TCC中的两阶段提交并没有对开发者完全屏蔽，也就是说从代码层面，开发者是可以感受到两阶段提交的存在。

1、try过程的本地事务，是保证资源预留的业务逻辑的正确性。

2、confirm/cancel执行的本地事务逻辑确认/取消预留资源，以保证最终一致性，也就是所谓的补偿型事务。由于是多个独立的本地事务，因此不会对资源一直加锁。

**总之：**

TCC 实质上是应用层的2PC()，好比把 XA 两阶段提交那种在数据资源层做的事务管理工作提到了数据应用层。

2PC是资源层面的分布式事务，是强一致性，在两阶段提交的整个过程中，\*一直会持有资源的锁\*。

TCC是业务层面的分布式事务，最终一致性，\*不会一直持有资源的锁\*。

TCC相比较于2PC来讲性能会好很多，但是因为同时需要改造try、confirm、canel3个接口，开发成本高。

如果Confirm和Cancel 操作可能被重复调用，故要求Confirm和Cancel两个接口必须是幂等的。

## 2.4 分布式事务（RocketMQ实现分布式事务）

目前流行的消息中间件如RabbitMQ、Kafka、RocketMQ等，只有RocketMQ支持事务消息。RocketMQ的事务消息模型借鉴了2PC模式。

### 2.4.1 分布式事务场景

例子：加入A给B转账100元，他们不在同一服务上。

目标：A减去100元，B增加100元。

实际情况可能有如下4种：

1. A账户减去100元（成功），B账户增加100元（成功）
2. A账户减去100元（失败），B账户增加100元（失败）
3. A账户减去100 元（成功），B账户增加100 元（失败）
4. A账户减去100元（失败），B账户增加100元（成功）

这里第1和第2种情况都能保证事务的一致性，但是第3、4无法保证事务的一致性。那我们看RocketMQ如何保证事务的一致性。

### 2.4.2 RocketMQ 实现分布式事务的原理

**1、基础概念：**

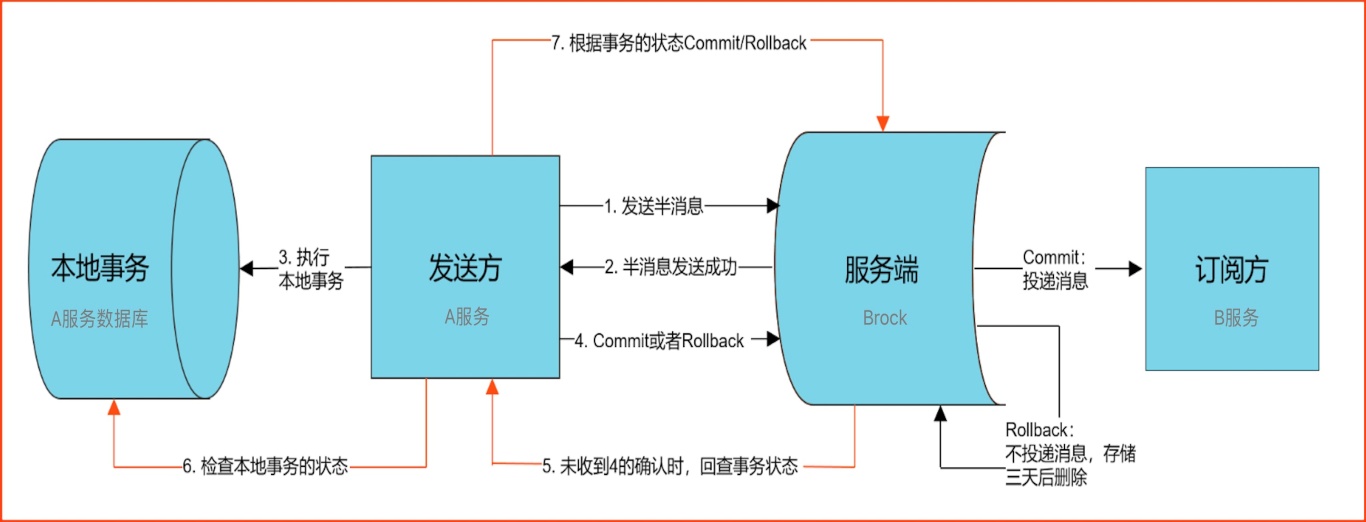
1. 最终一致性：**RocketMQ是一种最终一致性的分布式事务**，就是说它保证的是消息最终一致性，而不是像2PC、3PC、TCC那样强一致分布式事务，至于为什么说它是最终一致性事务下面会详细说明。
2. Half Message (半消息)

是指暂不能被Consumer消费的消息，Porduce已经把消息发送到Broker端，但此消息被标记为暂不能投递的状态，处于该种状态下的消息被称之为半消息，需要Producer对消息进行二次的确认后，Consumer才能消费他。

1. 消息回查：由于网络闪断，或者producer应用重启等原因，导致Producer端一直没有对Half message 进行二次确认，这时Broker服务器会定时的扫描长期处于半消息的消息，会主动询问producer端该消息的最终状态（Commit 或者 RollBack），该消息即为消息回复。

**2、分布式事务交互流程：**

理解这张阿里官方的图，就能理解RocketMQ分布式事务的原理了。



上图解释：

1. A服务先发送个Half Message给Brock端，消息中携带 B服务 即将要+100元的信息。
2. 当A服务知道Half Message 发送成功后，那么开始第三部执行本地事务。
3. 执行本地事务（会有三种情况：a、执行成功；b、执行失败、c、网络等原因导致没有响应）。

（4.1）如果本地事务成功，那么product向Broker服务器发送Commit，这样B服务就可以消费该message。

（4.2）如果本地事务失败，那么product向Broker服务器发送rollback，那么就会直接删除Broker上面的这条半消息。

（4.3）如果因为网络等原因迟迟没有返回失败还是成功，那么会执行RocketMQ的回调接口，来进行事务的回查。

**为什么要先发送Half message ？**

**可以先确认Broker服务器是否正常，如果半消息都发送失败了，说明Broker有可能已经宕机了。可以通过半消息来回查事务，如果半消息发送成功后一直没有被二次确认，那么就会回查事务状态。**

**为什么说MQ是最终一致性事务？**

**A账户减100 （失败），B账户加100 （成功）**

**因为**：如果A服务本地事务都失败了，那B服务永远不会执行任何操作，因为消息压根就不会传到B服务。

那么 **A账户减100 （成功），B账户加100 （失败）** 会不会可能存在的。

**答案是会的**

因为A服务只负责当我消息执行成功了，保证消息能够送达到B,至于B服务接到消息后最终执行结果A并不管。

**那么B服务失败怎么办？**

如果B最终执行失败，几乎可以断定就是代码有问题所以才引起的异常，因为消费端RocketMQ有重试机制，如果不是代码问题一般重试几次就能成功。

如果是代码的原因引起多次重试失败后，也没有关系，将该异常记录下来，由人工处理，人工兜底处理后，就可以让事务达到最终的一致性。

**注：**

**代码样例后期组件开发中实现**