[1. Java基础 2](#_Toc26672)

[2.Java高级 2](#_Toc23892)

[3.Java虚拟机 2](#_Toc25760)

[4.Spring 2](#_Toc1983)

[5.Mybatis 2](#_Toc27594)

[6.SpringBoot 2](#_Toc11816)

[7.SpringCloud 2](#_Toc22403)

[8.Redis 2](#_Toc5698)

[9.ElasticSearch 2](#_Toc17667)

[10.Zookeeper 2](#_Toc25969)

[11.Dobblo 2](#_Toc29957)

[12.MySQL 2](#_Toc20193)

[13.MyCat 2](#_Toc22002)

[14.RocketMQ 3](#_Toc14207)

[15.Linux 3](#_Toc20616)

[16.设计模式 3](#_Toc9867)

[17.分布式锁 3](#_Toc29339)

[18.事务 4](#_Toc20437)

[18.1数据库本地事务：ACID 4](#_Toc12502)

[18.2 InnoDB 实现原理 4](#_Toc11505)

[18.3分布式事务 4](#_Toc32693)

[18.3.1分布式的理论CAP 5](#_Toc1243)

[18.3.2 BASE理论 5](#_Toc17670)

[18.4分布式事务的解决方案 5](#_Toc32224)

[18.4.1 2PC/3PC 6](#_Toc28337)

[18.4.2 SAGA事务 7](#_Toc29651)

[18.4.3 TCC 8](#_Toc8352)

[18.4.4 MQ事务 9](#_Toc23758)

# Java基础

# 2.Java高级

# 3.Java虚拟机

# 4.Spring

# 5.Mybatis

# 6.SpringBoot

# 7.SpringCloud

# 8.Redis

# 9.ElasticSearch

# 10.Zookeeper

# 11.Dobblo

# 12.MySQL

# 13.MyCat

# 14.RocketMQ

分布式事务

[https://www.infoq.cn/article/2018%2F08%2Frocketmq-4.3-release](https://www.infoq.cn/article/2018/08/rocketmq-4.3-release)

# 15.Linux

# 16.设计模式

# 17.分布式锁

# 18.事务

事务提供一种机制将一个活动涉及的所有操作纳入到一个不可分割的执行单元，组成事务的所有操作只有在所有操作均能正常执行的情况下方能提交，只要其中任一操作执行失败，都将导致整个事务的回滚。简单地说，事务提供一种 "要么什么都不做，要么做全套（All or Nothing）" 机制。

## 18.1数据库本地事务：ACID

ACID是四个单词首字母的组合，即使Atomicity（原子性），Consistency（一致性），Isolation（隔离性）和Durability（持久性）。

**Atomicity（原子性）**

要么全部完成，要么全部不完成。

**Consistency（一致性）**

事务执行成功，所有变化（一般指数据）均正确操作。

**Isolation（隔离性）**

在并发环境下，不同事务操作相同数据，每个事务都有各自完整的数据空间。 事务不会查看到中间数据。

**Durability（持久性）**

事务只要成功结束，那么对数据库数据的更新便会永久保存下来。

## 18.2 InnoDB 实现原理

InnoDB是MySQL的一种存储引擎，它的实现是基于InnoDB的日志和锁来保证的。其中，**隔离性**是通过锁来保证的，**原子性**和**一致性**是**Undolog**来保证的，而**持久性**是通过**RedLog**保证的。

UndoLog：在操作任何数据前，将数据备份到一个地方（数据存储备份的地方叫做 UndoLog）。然后进行修改，如果事务执行失败了，那么系统可以利用备份数据恢复到事务执行前的状态。

RedoLog：记录新数据的备份，在事务提交前，只要将 RedoLog 持久化即可，不需要将数据持久化。当系统崩溃时，系统会根据 RedoLog 的内容，将数据更新至最新状态。

## 18.3分布式事务

数据库本地事务这个好控制，但是分布式事务就复杂多了，它是由多个操作组成，而这些操作不是基于一个数据库进行的，这便是它的难点。要么都全部成功，要么都全部失败。既All or Nothting。

分布式事务需求由来是因为微服务架构的兴起，兴趣的根本原因是单机无法处理大的数据量了，需要进行分库分表，分布在不同的机器一起进行。而事务这块是必不可少的，因而带来分布式事务的问题。

### 18.3.1分布式的理论CAP

CAP定律，又成为布鲁尔定理，CAP来自三个英文单词首字母的组合，既

**C** Consistency （一致性）

*所有数据备份，在同一时刻是否同样的值*

**A** Availability（可用性）

*集群中的部分节点出现故障，集群整体是否还能相应客户端读写*

**P** Partition tolerance（分区容错性）

*分区相当于对通信的时限要求，系统如果不能在指定时限内达成数据一致性，就意味着发生了分区，需要在 C 与 A 之间选择。*

CAP不是三剑客，三者不能共存，最多保证两种，其中P分区容错性是一样要保证的。这一点是因为无法100%的保证网络可靠，所以分区容错性必要得保证的。

CAP最多保证两种，因此有3套方案，如下：

1. CA 理论上不可能存在。由于网络无法 100% 可靠，故分布式系统中必定会有分区。分区发生时，为了保证 C，这个时候一定要拒绝请求，但是 A 不允许。

2. CP 放弃可用性，追求强一致

3. AP 放弃一致性（强一致性），追求分区容错性和可用性。BASE 也是根据 AP 的拓展。

### 18.3.2 BASE理论

BASE其实讲的三点，如下：

1.Basically Available(基本可用)

*分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用。*

2.Soft State(软状态)

*允许系统中存在中间状态，这个状态不影响系统可用性，这里指的是CAP中的不一致。*

3.Eventually Consistent(最终一致性)

*最终一致是指经过一段时间后，所有节点数据都将会达到一致。*

上面讲过，CASE理论是CAP中AP方案的扩展。BASE解决了CAP中理论没有网络延迟，在BASE中用软状态和最终一致，保证了延迟后的一致性。BASE和ACID是相反的，它完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。

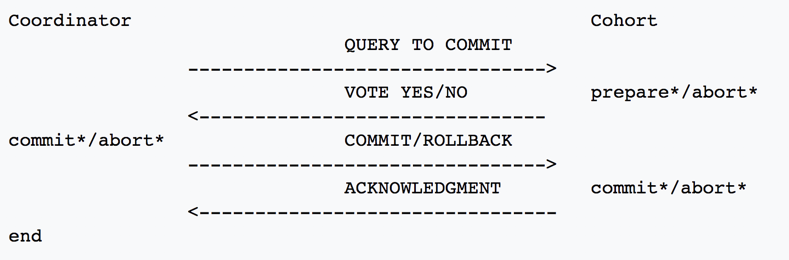
## 18.4分布式事务的解决方案

解决分布式事务，目前了解到的有4种，分别是*2PC/3PC，TCC，MQ事务，SAGA事务*。下面一一讲解这4种方案。

### 18.4.1 2PC/3PC

2PC/3PC其实是两个不同的的方案，2PC既二阶段提交，3PC自然就是3阶段提交了，这里合在一起讲解，因为3PC是2PC扩展来的。

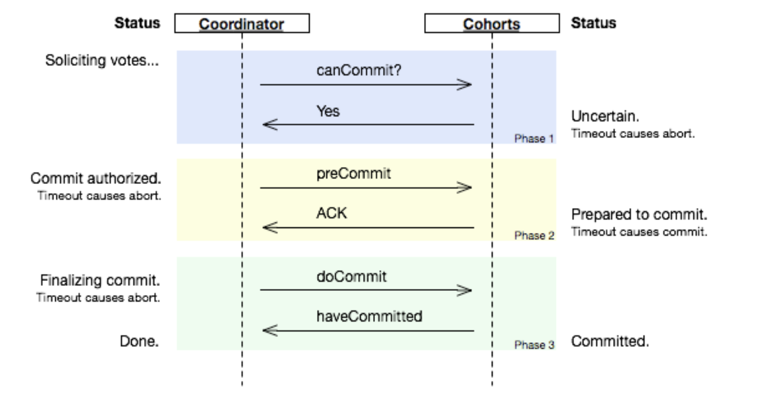
先来讲解2PC二阶段提交。2PC 把事务的执行分为两个阶段，如图所示。



**第一个阶**段即 prepare阶段，这个阶段实际上就是投票阶段，协调者向参与者确认是否可以共同提交。

**第二阶段**是得到全部参与者的所有回答后，协调者向所有的参与者发布共同提交或者共同回滚的指令，用以保证事务达到一致性。

但是分布式系统中的所有通信均存在着三种状态：***成功****，****失败****，****超时***。其中，超时状态的存在是我们在设计分布式系统时所面对的永远的痛，2PC同样存在问题，尤其是在发送完可以提交的指令后，参与者在没有收到提交或者回滚的指令时，面对已经上锁的资源，面对已经写出去的undo或者redo日志，参与者会一时陷入手足无措的状态，为了解决这个问题，3PC 应运而生，如下图所示。



3PC在commit之前增加了**preCommit**的过程，使得在参与者在收不到确认时，依然可以从容commit或者rollback，避免资源锁定太久导致浪费。但是3PC同样存在着很多问题。实现起来非常复杂，因为很难通过多次询问来解决系统间分歧问题，尤其是存在超时状态互不信任的分布式网络中。

优点：在于已经有较为成熟的实现方案，比如 XA。

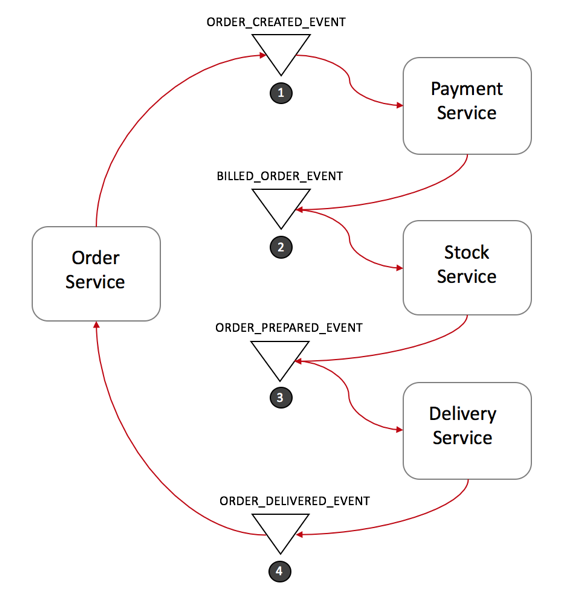
缺点：XA是一个阻塞协议。服务在投票后需要等待协调器的决定，此时服务会阻塞并锁定资源。由于其阻塞机制和最差时间复杂度高，因此，这种设计不能适应随着事务涉及的服务数量增加而扩展的需要，很难用于并发较高以及子事务声明周期较长 (long-running transactions) 的分布式服务中。

### 18.4.2 SAGA事务

SAGA算法是一种异步的分布式事务解决方案，假设所有事件按照顺序推进，总能达到系统的最终一致性，因此SAGA需要服务分别定义提交接口以及补偿接口，当某个事务分支失败时，调用其它的分支的补偿接口来进行回滚。

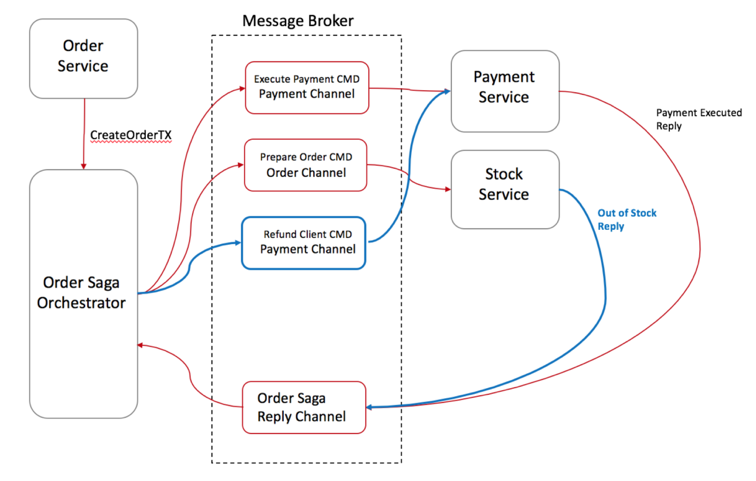
SAGA的具体实现分为两种：Choreography以及Orchestration。

**Choreography**这种模式下不存在协调器的概念，每个节点均对自己的上下游负责，在监听处理上游节点事件的同时，对下游节点发布事件。



**Orchestration**存在中心节点的模式,该中心节点，即协调器知道整个事务的分布状态，相比于无中心节点方式，该方式有着许多优点：

* 能够避免事务之间的循环依赖关系。
* 参与者只需要执行命令/回复(其实回复消息也是一种事件消息)，降低参与者的复杂性。
* 开发测试门槛低。
* 在添加新步骤时，事务复杂性保持线性，回滚更容易管理。因此大多数saga模型实现均采用了这种思路。

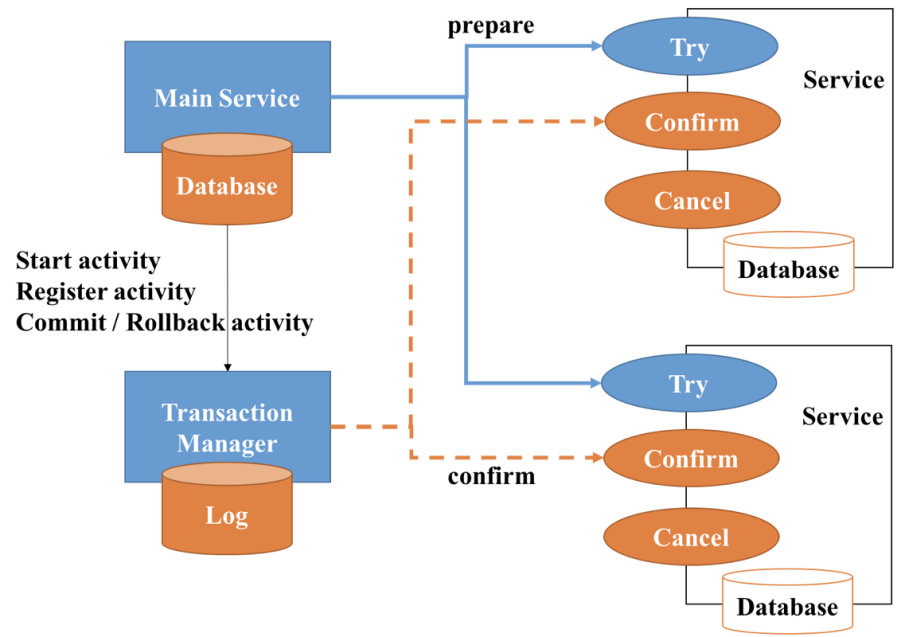


总结一下：SAGA模型的优点在于其降低了事务粒度，使得事务扩展更加容易，同时采用了异步化方式提升性能。但是其缺点在于很多时候很难定义补偿接口，回滚代价高，而且由于SAGA在执行过程中采用了先提交后补偿的思路进行操作，所以单个子事务在并发提交时的隔离性很难保证。

### 18.4.3 TCC

TCC(Try-Confirm-Concel) 模型是一种补偿性事务，主要分为三个阶段：

**Try**:检查、保留资源，**Confirm** :执行事务，**Concel**:释放资源，如图所示。



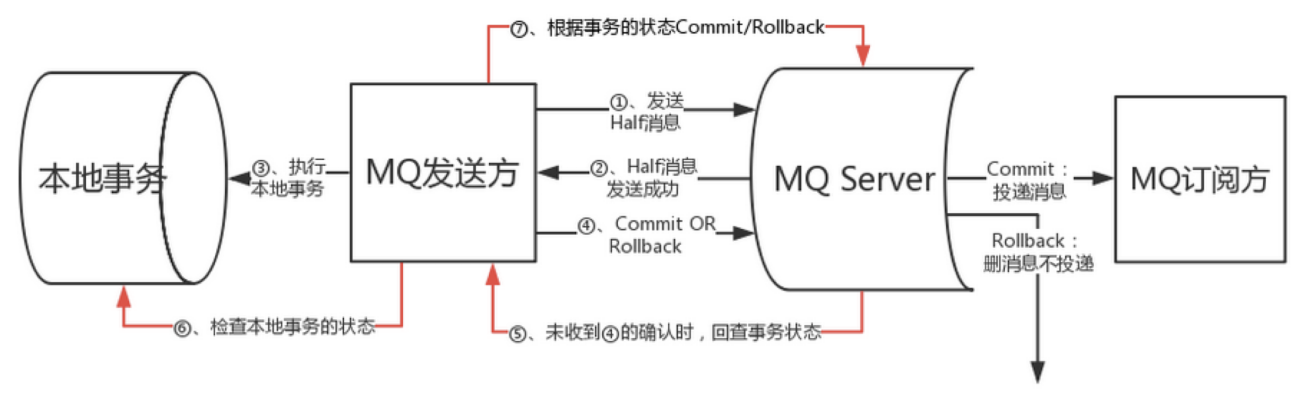
其中，活动管理器记录了全局事务的推进状态以及各子事务的执行状态，负责推进各个子事务共同进行提交或者回滚。同时负责在子事务处理超时后不停重试，重试不成功后转手工处理，用以保证事务的最终一致性。

总结一下，相比于SAGA模型，其优点在于尝试阶段仅仅只是对业务系统做检测，并保留业务资源，并没有真正提交，所以后续SAGA需要针对提交的事务做补偿，而TCC则仅仅需要释放保留资源，降低了补偿成本；并且，由于在Try阶段对资源进行了保留锁定，所以相比于SAGA模式，TCC模式拥有更高的隔离性。

缺点：相比于SAGA模式，TCC模式多增加了一个状态，导致在业务开发过程中，复杂度上升，而且协调器与子事务的通信过程增加，状态轮转处理也更为复杂。

### 18.4.4 MQ事务

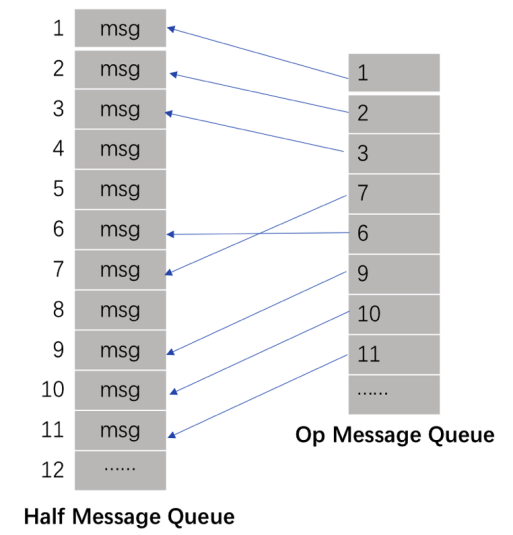
RocketMQ事务消息的设计流程同样借鉴了两阶段提交理论,如图所示：



1. 事务发起方首先发送prepare消息到MQ。
2. 在发送prepare消息成功后执行本地事务。
3. 根据本地事务执行结果返回commit或者是rollback。
4. 如果消息是rollback，MQ将删除该prepare消息不进行下发，如果是commit消息，MQ将会把这个消息发送给consumer端。
5. 如果执行本地事务过程中，执行端挂掉，或者超时，MQ将会不停的询问其同组的其它producer来获取状态。
6. Consumer端的消费成功机制有MQ保证

RocketMQ 事务消息在实现上充分利用了 RocketMQ 本身机制，在实现零依赖的基础上，同样实现了高性能、可扩展、全异步等一系列特性。

在具体实现上，RocketMQ 通过使用 Half Topic 以及 Operation Topic 两个内部队列来存储事务消息推进状态



Half Topic对应队列中存放着prepare消息，Operation Topic对应的队列则存放了prepare message对应的commit/rollback消息，消息体中则是prepare message对应的offset，服务端通过比对两个队列的差值来找到尚未提交的超时事务，进行回查。