# 分布式事务

## 什么是分布式事务？

就是要在分布式系统中实现事务，它其实是由多个本地事务组合而成

### 解决方案：

#### 消息队列（RocketMQ-事务消息）

#### AT（）

业务无侵入

#### TCC（）

业务有侵入

#### Saga

业务有侵入

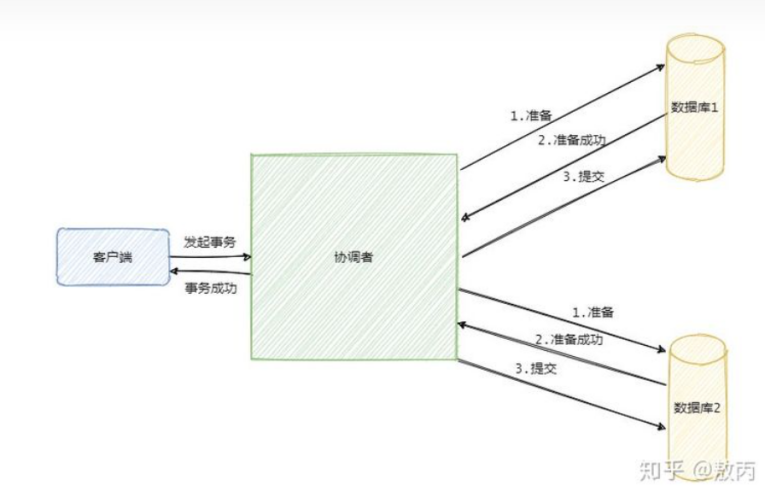
#### XA

## 原理

参考： https://zhuanlan.zhihu.com/p/183753774

### 2PC（Two-phase commit protocol）

中文叫二阶段提交，二阶段提交是一种强一致性设计，2PC 引入一个事务协调者的角色来协调管理各参与者（也可称之为各本地资源）的提交和回滚，二阶段分别指的是准备（投票）和提交两个阶段。



第一阶段提交失败：

假如在第一阶段有一个参与者返回失败，那么协调者就会向所有参与者发送回滚事务的请求，即分布式事务执行失败。

第二阶段提交失败（此时有两种情况）：

一种是**第二阶段执行的是回滚事务操作**，那么答案是不断重试，直到所有参与者都回滚了，不然那些在第一阶段准备成功的参与者会一直阻塞着。

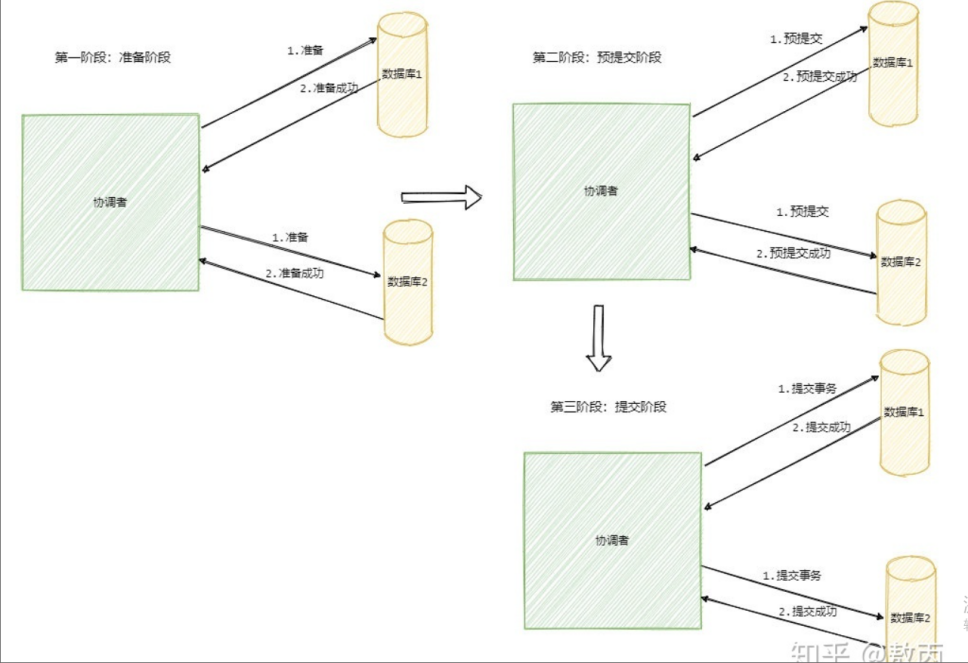
第二种是**第二阶段执行的是提交事务操作**，那么答案也是不断重试，因为有可能一些参与者的事务已经提交成功了，这个时候只有一条路，就是头铁往前冲，不断的重试，直到提交成功，到最后真的不行只能人工介入处理。

### 3PC（仅作了解）

3PC 的出现是为了解决 2PC 的一些问题，相比于 2PC 它在参与者中也引入了超时机制，并且新增了一个阶段使得参与者可以利用这一个阶段统一各自的状态。

3PC 包含了三个阶段，分别是准备阶段、预提交阶段和提交阶段，对应的英文就是：CanCommit、PreCommit 和 DoCommit。

看起来是把 2PC 的提交阶段变成了预提交阶段和提交阶段，但是 3PC 的准备阶段协调者只是询问参与者的自身状况，比如你现在还好吗？负载重不重？这类的。而预提交阶段就是和 2PC 的准备阶段一样，除了事务的提交该做的都做了。提交阶段和 2PC 的一样，让我们来看一下图。



不管哪一个阶段有参与者返回失败都会宣布事务失败，这和 2PC 是一样的（当然到最后的提交阶段和 2PC 一样只要是提交请求就只能不断重试）。

准备阶段的变更成不会直接执行事务，而是会先去询问此时的参与者是否有条件接这个事务，因此不会一来就干活直接锁资源，使得在某些资源不可用的情况下所有参与者都阻塞着。而预提交阶段的引入起到了一个统一状态的作用，它像一道栅栏，表明在预提交阶段前所有参与者其实还未都回应，在预处理阶段表明所有参与者都已经回应了。

3PC 相对于 2PC 做了一定的改进：引入了参与者超时机制，并且增加了预提交阶段使得故障恢复之后协调者的决策复杂度降低，但整体的交互过程更长了，性能有所下降，并且还是会存在数据不一致问题。

### TCC（Try - Confirm - Cancel）

2PC 和 3PC 都是数据库层面的，而 TCC 是业务层面的分布式事务，就像我前面说的分布式事务不仅仅包括数据库的操作，还包括发送短信等，这时候 TCC 就派上用场了！

Try 指的是预留，即资源的预留和锁定，注意是预留。

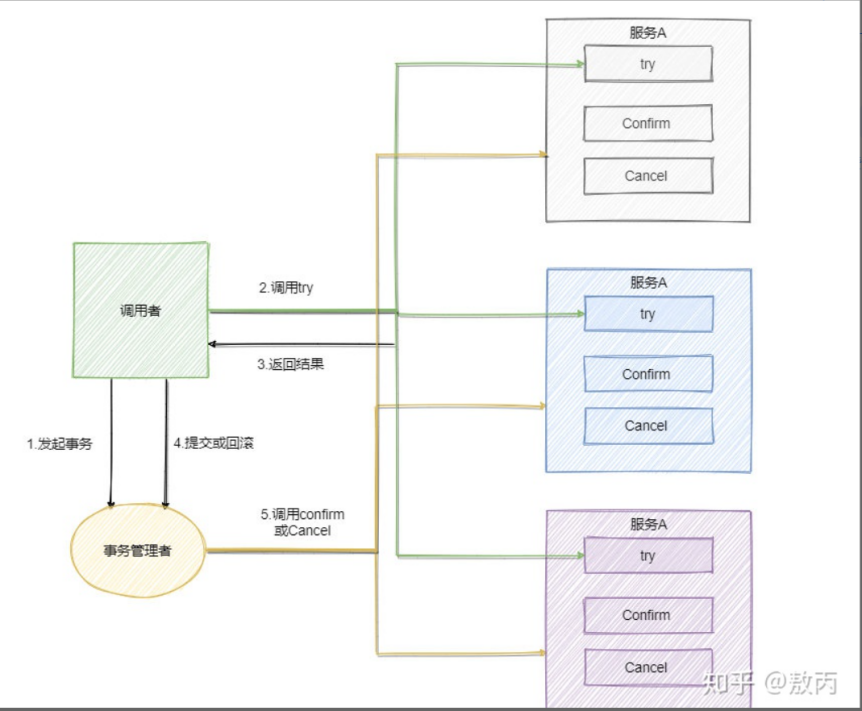
Confirm 指的是确认操作，这一步其实就是真正的执行了。

Cancel 指的是撤销操作，可以理解为把预留阶段的动作撤销了

其实从思想上看和 2PC 差不多，都是先试探性的执行，如果都可以那就真正的执行，如果不行就回滚。

比如说一个事务要执行A、B、C三个操作，那么先对三个操作执行预留动作。如果都预留成功了那么就执行确认操作，如果有一个预留失败那就都执行撤销动作。

我们来看下流程，TCC模型还有个事务管理者的角色，用来记录TCC全局事务状态并提交或者回滚事务。



可以看到流程还是很简单的，难点在于业务上的定义，对于每一个操作你都需要定义三个动作分别对应Try - Confirm - Cancel。因此 TCC 对业务的侵入较大和业务紧耦合，需要根据特定的场景和业务逻辑来设计相应的操作。

还有一点要注意，撤销和确认操作的执行可能需要重试，因此还需要保证操作的幂等。

相对于 2PC、3PC ，TCC 适用的范围更大，但是开发量也更大，毕竟都在业务上实现，而且有时候你会发现这三个方法还真不好写。不过也因为是在业务上实现的，所以TCC可以跨数据库、跨不同的业务系统来实现事务。

### 本地消息表

### 消息事务

### 最大努力通知

总结：

**2PC** 和 **3PC** 是一种强一致性事务，不过还是有数据不一致，阻塞等风险，而且只能用在数据库层面。而 **TCC** 是一种补偿性事务思想，适用的范围更广，在业务层面实现，因此对业务的侵入性较大，每一个操作都需要实现对应的三个方法。**本地消息**、**事务消息**和**最大努力**通知其实都是最终一致性事务，因此适用于一些对时间不敏感的业务。

## 分类：

### 刚性事务:

1. XA强一致事务

基于XA协议实现的分布式事务对业务侵入很小。 它最大的优势就是对使用方透明，用户可以像使用本地事务一样使用基于XA协议的分布式事务。 XA协议能够严格保障事务ACID特性。

严格保障事务ACID特性是一把双刃剑。 事务执行在过程中需要将所需资源全部锁定，它更加适用于执行时间确定的短事务。 对于长事务来说，整个事务进行期间对数据的独占，将导致对热点数据依赖的业务系统并发性能衰退明显。 因此，在高并发的性能至上场景中，基于XA协议的分布式事务并不是最佳选择。

1. 啊打发十分
2. 啊沙发沙发

### 柔性事务：

1. ***BASE事务*：**

BASE是基本可用、柔性状态和最终一致性这三个要素的缩写。

* 基本可用（Basically Available）保证分布式事务参与方不一定同时在线。
* 柔性状态（Soft state）则允许系统状态更新有一定的延时，这个延时对客户来说不一定能够察觉。
* 而最终一致性（Eventually consistent）通常是通过消息可达的方式保证系统的最终一致性。

在ACID事务中对隔离性的要求很高，在事务执行过程中，必须将所有的资源锁定。 柔性事务的理念则是通过业务逻辑将互斥锁操作从资源层面上移至业务层面。通过放宽对强一致性要求，来换取系统吞吐量的提升。



# 分布式事务框架

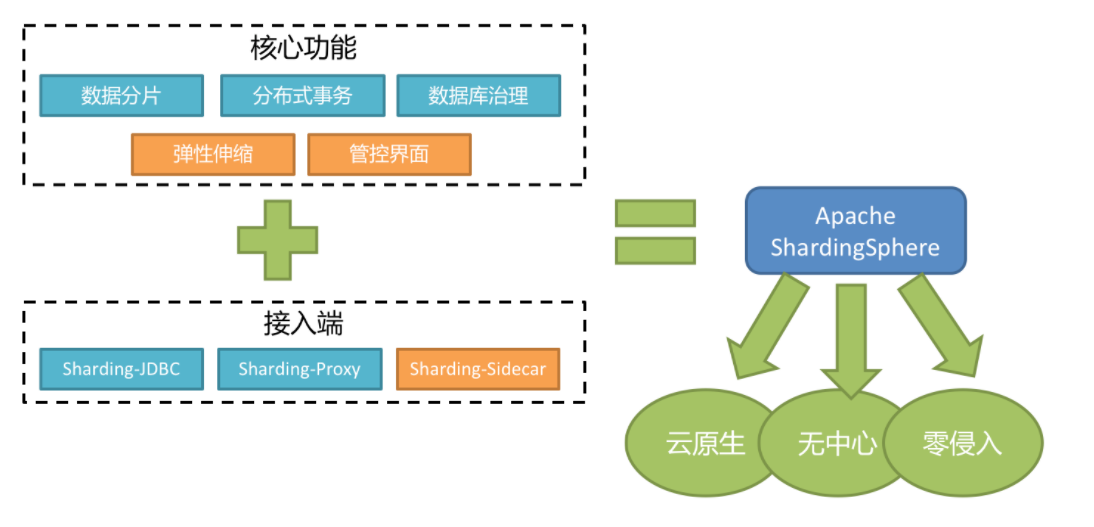
## Shardingsphere

<https://shardingsphere.apache.org/document/legacy/3.x/document/cn/overview/>

简介：

ShardingSphere是一套开源的**分布式数据库**中间件解决方案组成的生态圈，它由Sharding-JDBC、Sharding-Proxy和Sharding-Sidecar（计划中）这3款相互独立的产品组成。 他们均提供标准化的数据分片、分布式事务和数据库治理功能，可适用于如Java同构、异构语言、容器、云原生等各种多样化的应用场景。

ShardingSphere定位为关系型数据库中间件，旨在充分合理地在分布式的场景下利用关系型数据库的计算和存储能力，而并非实现一个全新的关系型数据库。



## SEATA