# 事务的基本概念

**事务的四大特性**

### 1.1.1 原子性

事务的原子性指的是构成事务的所有操作要么全部执行完成，要么全部不执行，不可能出现部分执行完成，部分执行失败的情况。

例如，在转账业务中，张三向李四转账100元，张三账户的余额减少100元，李四账户的余额增加100元。在开启事务的情况下，这两个操作全部执行完成，或者全部不执行，不可能出现只为张三的账户减少100元的操作，也不能出现只为李四的账户增加100元的操作。

### 1.1.2 一致性

事务的一致性指的是在事务执行之前和事务执行之后，数据始终处于一致的状态。

例如，同样是转账业务，张三向李四转账100元，转账前和转账后的数据是正确的，也就是说，转账后张三账户的余额会减少100元，李四账户的余额会增加100元，这叫作数据的一致性。如果出现张三账户的余额减少了100元，而李四账户的余额没有增加100元，这就出现了数据的不一致。

### 1.1.3 隔离性

事务的隔离性指的是并发执行的两个事务之间互不干扰。一个事务在执行过程中，不能看到其他事务运行过程的中间状态。

例如，在张三向李四转账的业务场景中，存在两个并发执行的事务A和事务B，事务A执行扣减张三账户余额的操作和增加李四账户余额的操作，事务B执行查询张三账户余额的操作。在事务A未提交完成之前，事务B读取的张三的账户余额仍然为未扣减之前的账户余额，不能读取到扣减后的账户余额。

注：MySQL通过锁机制来保证事务的隔离性。

### 1.1.4 持久性

事务的持久性指的是事务提交完成后，此事务对数据的更改操作会被持久化到数据库中，并且不会被回滚。

例如，在张三向李四转账的业务场景中，在同一事务中执行扣减张三账户余额和增加李四账户余额的操作，只要事务提交完成后，这种对数据的修改操作就会被持久化到数据库中，这种修改是永久的，不会被回滚。

注意：数据库的事务在实现时，会将一次事务中包含的所有操作全部封装成一个不可分割的执行单元，这个单元中的所有操作要么全部成功，要么全部失败。只要其中的任意一个操作执行失败，整个事务就会执行回滚操作。

# 事务的类型

### 扁平事务

扁平事务是涉及事务操作中最常见，也是最简单和最常用的一种事务。在数据库中，其通常使用begin或者start transaction字段开始，由commit或者rollback字段结束。在这之间的所有操作要么全部提交成功，要么全部提交失败（回滚）。当今主流的数据库几乎都支持扁平事务的实现。

扁平事务虽然是最常见、最简单和最常用的事务，但是其无法提交或者回滚整个事务中的部分事务，只能把整个事务全部提交或者回滚。为了解决这个问题，带有保存点的扁平事务出现了。

### 1.2.2 带有保存点的扁平事务

在事务内部设置了保存点，就是带有保存点的扁平事务。带有保存点的扁平事务通过在事务内部的某个位置设置savepoint，来达到将当前事务回滚到此位置的目的。

从本质上讲，普通的扁平事务也是有保存点的，只不过普通的扁平事务只有一个隐式的保存点，并且这个隐式的保存点会在事务启动的时候，自动设置为当前事务的开始位置。也就是说，普通的扁平事务具有保存点，而且保存点默认是事务的开始位置。

### 1.2.3 链式事务

链式事务是在带有保存点的事务的基础上，自动将当前事务的上下文隐式地传递给下一个事务。也就是说，一个事务的提交操作和下一个事务的开始操作具备原子性，上一个事务的处理结果对下一个事务是可见的。事务与事务之间就像链条一样传递下去。

注意：链式事务在提交的时候，会释放要提交的事务中的所有锁和所有保存点。也就是说，链式事务的回滚操作只能回滚到当前所在事务的保存点，而不能回滚到已提交的事务的保存点。

### 1.2.4 嵌套事务

顾名思义，嵌套事务就是有多个事务处于嵌套状态，共同完成一项任务的处理，整个任务具备原子性。嵌套事务最外层有一个顶层事务，这个顶层事务控制着所有的内部子事务，内部子事务提交完成后，整体事务并不会提交，只有等到最外层的顶层事务提交完成后，整体事务才算提交完成。

关于嵌套事务需要注意以下几点。

1）回滚嵌套事务内部的子事务时，会将事务回滚到外部顶层事务的开始位置。

2）嵌套事务的提交是从内部的子事务向外依次提交的，直到最外层的顶层事务提交完成。

3）回滚嵌套事务的最外层的顶层事务时，会回滚嵌套事务中包含的所有事务，包括已提交的内部事务。

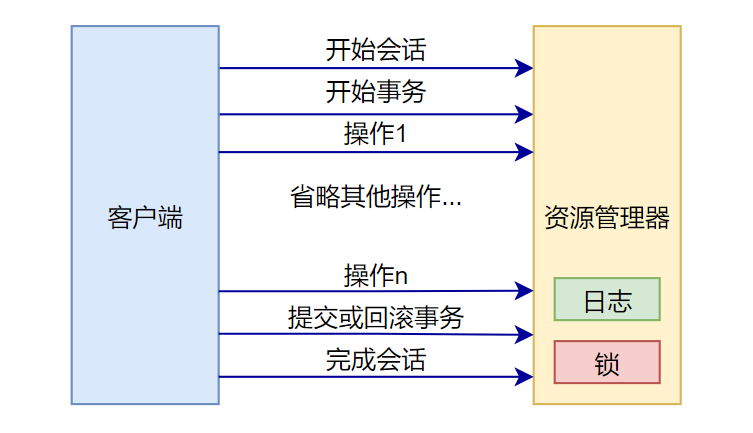
在主流的关系型数据库中，MySQL不支持原生的嵌套事务，SQL Server支持。这里，笔者不建议使用嵌套事务。

### 1.2.5 分布式事务

分布式事务指的是事务的参与者、事务所在的服务器、涉及的资源服务器以及事务管理器等分别位于不同的分布式系统的不同服务或数据库的节点上。简单点说：分布式事务就是一个在不同环境（比如不同的数据库、不同的服务器）下运行的整体事务。这个整体事务中包含一个或者多个分支事务，并且这个整体事务中的所有分支事务要么全部提交成功，要么全部不提交。

例如，在电商系统的下单减库存业务中，订单业务所在的数据库为事务A的节点，库存业务所在的数据库为事务B的节点。事务A和事务B共同组成了一个分布式事务，而且事务A和事务B组成的分布式事务具备ACID属性，整体要么全部提交成功，要么全部提交失败。

### 本地事务的执行流程



1）客户端开始事务操作之前，首先需要开启一个连接会话。

2）开始会话之后，客户端会发起开始事务的指令。

3）事务开启后，客户端会发送各种SQL语句处理数据。

4）正常情况下，客户端会发起提交事务的指令，如果发生异常情况，客户端会发起回滚事务的指令。

5）在上述流程完成后，会话结束。

本地事务是由资源管理器在本地进行管理的。

### 本地事务的优缺点

优点：

1）支持严格的ACID属性，这也是本地事务得以实现的基础。

2）事务可靠，一般不会出现异常情况。

3）本地事务的执行效率比较高。

4） 事务的状态可以只在数据库中进行维护，上层的应用可不必理会事务的具体状态。

5）应用的编程模型比较简单，不会涉及复杂的网络通信。

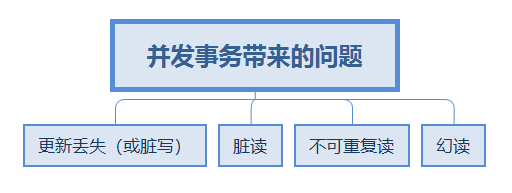
缺点：

1）不具备分布式事务的处理能力。

2）一次事务过程中只能连接一个支持事务的数据库，也就是不能用于多个事务性数据库。

### MySQL并发事务带来的问题

数据库一般会并发执行多个事务，多个事务就可能会并发地对相同的数据进行增加、删除、修改和查询操作，这就可能导致并发事务问题出现。并发事务带来的问题总体上分为更新丢失（或脏写）、脏读、不可重复读和幻读



**1.更新丢失（或脏写）**

对于同一行数据来说，一个事务对该行数据的更新操作覆盖了其他事务对该行数据的更新操作。

例如，张三的银行账户余额有100元，当前有两个事务，分别为事务A和事务B，事务A向张三的银行账户增加100元，事务B向张三的银行账户增加200元。此时，事务A和事务B同时读取到张三的余额，均为100元。事务A和事务B分别更新张三的银行账户余额，假设事务A先于事务B提交，而事务A和事务B都提交后的结果是张三的账户余额是300元。也就是说，后提交的事务B覆盖了事务A的更新操作。

更新丢失（或脏写）本质上是写操作的冲突，解决办法就是让每个事务按照串行的方式执行，按照一定的顺序依次进行写操作。

**2.脏读**

一个事务读取了另一个事务未提交的数据。

例如，当前有两个事务，分别为事务A和事务B。事务A向张三的银行账户转账100元，事务B查询张三的银行账户余额。事务A执行转账操作，此时事务A未提交，事务B查询到张三的银行账户多了100元。事务A由于某些原因，例如服务超时、系统异常等因素进行回滚操作。此时，事务B查询到的数据就是脏数据。

脏读本质上是读写操作的冲突，解决脏读的方法就是先写后读，也就是写完之后再读。

**3.不可重复读**

同一个事务，使用相同的查询语句，在不同的时刻读取的结果数据不一致。

例如，当前有两个事务，分别为事务A和事务B。事务A向张三的银行账户转账100元。事务B查询张三的账户余额，第一次查询时，事务A还没有转账；第二次查询时，事务A已经转账成功，此时，就会导致事务B两次查询出来的结果数据不一致。

不可重复读本质上是读写操作的冲突，解决办法就是先读后写，也就是读完之后再写。

**4.幻读**

一个事务两次读取一个范围的数据记录，两次读取的范围结果不同。

例如，当前有两个事务，分别为事务A和事务B。事务A两次查询张三的转账记录，事务B向张三的银行账户转账100元。事务A第一次查询时，事务B还没有转账。事务A第二次查询时，事务B已经转账成功，此时，就会导致事务A两次查询的转账明细数据不一致。

幻读本质上是读写操作的冲突，解决方法就是先读后写，也就是读完之后再写。

另外，很多人搞不懂不可重复读和幻读到底有何区别。这里，我们也来简单聊聊二者的区别。

1）不可重复读的重点在于更新和删除，而幻读的重点在于插入操作。

2）使用锁机制实现事务隔离级别时，在可重复读隔离级别中，SQL语句在第一次读取到数据后，就会将相应的数据加锁，其他事务无法修改这些数据，此时就可以实现可重复读。而这种方法无法对新插入的数据加锁。如果一个事务A读取了数据，或者修改了数据，此时，另一个事务B还是可以进行insert操作。这就会导致事务A莫名其妙地多了一条之前没有的数据，这就是幻读。

3）幻读无法通过行级锁来避免，需要使用串行化（Serializable）的事务隔离级别，但是这种事务隔离级别会极大的降低数据库的并发能力。

4）从本质上讲，不可重复读和幻读最大的区别，就在于如何通过锁机制来解决它们产生的问题。

另外，除了使用悲观锁来避免不可重复读和幻读的问题外，还可以使用乐观锁来处理，例如，MySQL、Oracle和PostgreSQL等数据库为了提高整体性能，就是使用基于乐观锁的MVCC（多版本并发控制）机制来避免不可重复读和幻读的问题。

## MySQL事务隔离级别

|  |  |  |  |
| --- | --- | --- | --- |
| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交 | 可能 | 可能 | 可能 |
| 读已提交 | 不可能 | 可能 | 可能 |
| 可重复读 | 不可能 | 不可能 | 可能 |
| 串行化 | 不可能 | 不可能 | 不可能 |

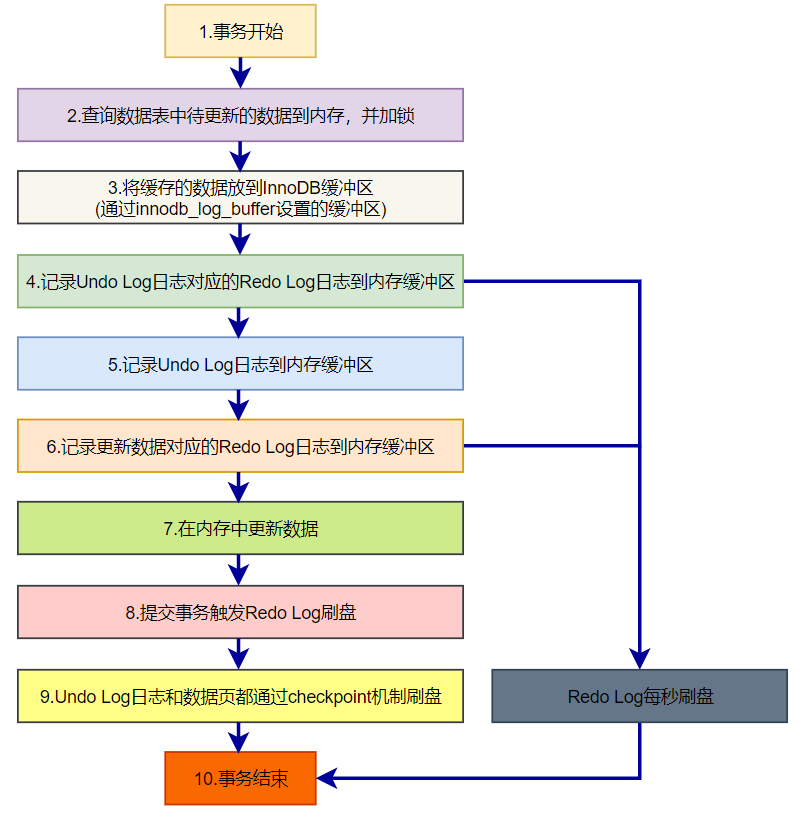
1）读未提交允许脏读，也就是在读未提交的事务隔离级别下，可能读取到其他会话未提交事务修改的数据。此时事务隔离级别下存在着脏读、不可重复读和幻读的问题。

2）读已提交只能读取到已经提交的数据。Oracle等数据库使用的默认事务隔离级别就是读已提交。此种事务隔离级别存在着不可重复读和幻读的问题。

3）可重复读就是在同一个事务内，无论何时查询到的数据都与开始查询到的数据一致，这是MySQL中InnoDB存储引擎默认的事务隔离级别。此种事务隔离级别下存在着幻读的问题。

4）串行化是指完全串行地读，每次读取数据库中的数据时，都需要获得表级别的共享锁，读和写都会阻塞。此种事务隔离级别解决了并发事务带来的问题，但由于其完全的串行化操作，数据库失去了并发特性。所以，此种隔离级别往往在互联网项目中不太常用。

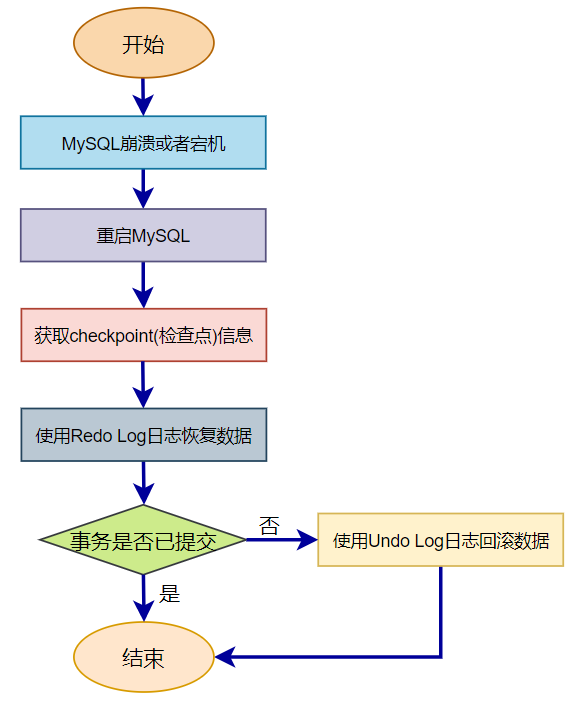
## MySQL事务的执行流程



MySQL在事务执行的过程中，会记录相应SQL语句的Undo Log和Redo Log，然后在内存中更新数据并形成数据脏页。接下来Redo Log会根据一定的规则触发刷盘操作，Undo Log和数据页则通过chekpoint机制刷盘。事务提交时，会将当前事务相关的所有Redo Log刷盘，只有当前事务相关的所有Redo Log刷盘成功，事务才算提交成功。

## MySQL事务恢复流程

按照MySQL事务执行流程图，如果事务在执行第8步之前，也就是事务还未提交，MySQL崩溃或者宕机，此时会先使用Redo Log恢复数据，然后再使用Undo Log回滚数据。如果在执行第8步之后，MySQL崩溃或者宕机，此时会使用Redo Log恢复数据，大体流程如图。



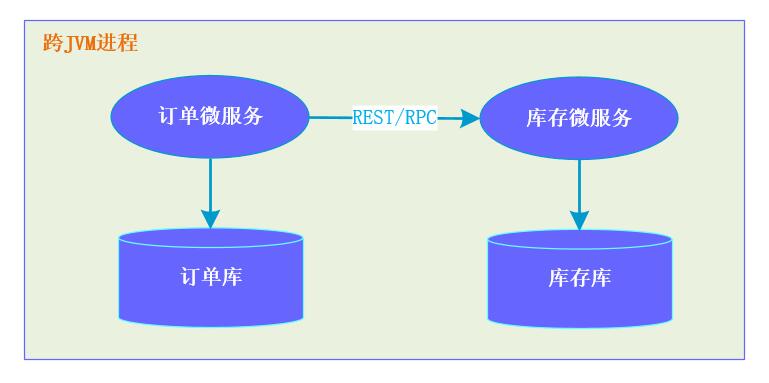
MySQL发生崩溃或者宕机，需要重启MySQL。MySQL重启之后，会获取checkpoint日志检查点信息，随后根据checkpoint日志检查点信息使用Redo Log恢复数据。如果在MySQL崩溃或者宕机之前，事务未提交，则接下来使用Undo Log回滚数据。如果在MySQL崩溃或者宕机之前，事务已经提交，则只用Redo Log恢复数据即可。

## 分布式事务场景

总体来讲，分布式事务会在三种场景产生，分别是跨JVM进程、跨数据库实例和多服务访问单数据库。

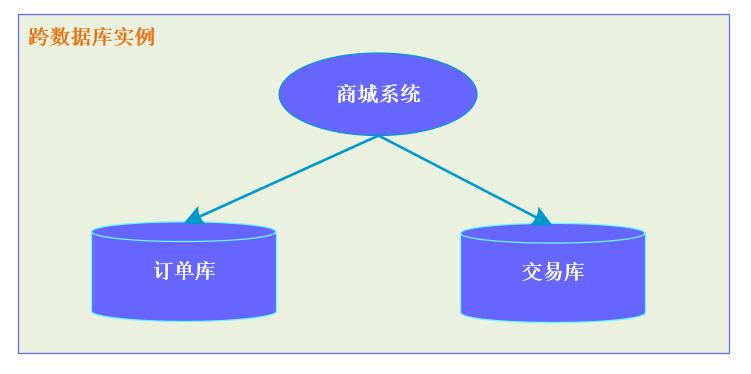
### 跨JVM进程

将单体项目拆分为分布式、微服务项目之后，各个服务之间通过远程REST或者RPC调用来协同完成业务操作。典型的场景就是：商城系统中的订单微服务和库存微服务，用户在下单时会访问订单微服务。订单微服务在生成订单记录时，会调用库存微服务来扣减库存。各个微服务是部署在不同的JVM进程中的，此时，就会产生跨JVM进程而导致的分布式事务问题。



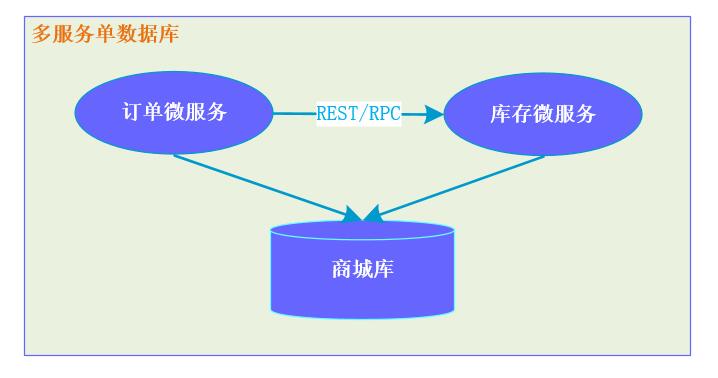
### 跨数据库实例

单体系统访问多个数据库实例，也就是跨数据源访问时会产生分布式事务。例如，我们的系统中的订单数据库和交易数据库放在不同的数据库实例中，当用户发起退款时，会同时操作用户的订单数据库和交易数据库，在交易数据库中执行退款操作，在订单数据库中将订单的状态变更为已退款。由于数据分布在不同的数据库实例，需要通过不同的数据库连接会话来操作数据库中的数据，此时，就产生了分布式事务。



### 多服务访问单数据库

多个微服务访问同一个数据库。例如，订单微服务和库存微服务访问同一个数据库就会产生分布式事务，原因是：多个微服务访问同一个数据库，本质上也是通过不同的数据库会话来操作数据库，此时就会产生分布式事务。



注意：跨数据库实例场景和多服务访问单数据库场景，本质上都会产生不同的数据库会话来操作数据库中的数据，进而产生分布式事务。这两种场景是比较容易忽略的。

# 分布式事务理论（CAP理论、Base理论）

CAP：一致性、可用性和分区容忍性。

### 一致性

一致性是指用户对数据的更新操作（包括新增、修改和删除），要么在所有的数据副本执行成功，要么在所有的数据副本执行失败。

也就是说，一致性要求对所有数据节点的数据副本的修改是原子操作。所有数据节点的数据副本的数据都是最新的，从任意数据节点读取的数据都是最新的状态。

### 可用性

可用性指的是客户端访问数据的时候，能够快速得到响应。需要注意的是，系统处于可用性状态下，每个存储节点的数据可能会不一致，并不要求应用程序向数据库写入数据时能够立刻读取到最新的数据。

也就是说，处于可用性状态下的系统，任何事务的操作都可以得到响应的结果，不会存在超时或者响应错误的情况。

### 分区容忍性

如果只是将存储系统部署并运行在一个节点上，当系统出现故障时，整个系统将不可用。如果将存储系统部署并运行在多个不同的节点上，并且这些节点处于不同的网络中，这就形成了网络分区。此时，不可避免的会出现由于网络问题而导致节点之间的通信出现失败的情况，但是，此时的系统仍能对外提供服务，这就是分区容忍性。

### CAP的组合

在分布式系统中，不会同时具备CAP三个特性，只能同时具备CAP中的两个特性。

如果要满足一致性，则需要在数据由主库同步到从库的过程中对从库加锁，防止同步的过程中应用程序向从库读取不一致的数据，数据同步完成后释放从库的锁。如果数据同步失败，则需要从数据库返回错误信息或者超时信息。

如果要满足可用性，则必须保证数据节点的可用性，无论何时查询从数据库中的数据，则从数据库都要快速响应查询结果，不会出现响应超时或者返回错误信息的情况。

如果一个系统在满足CAP理论中分区容忍性的前提下，一致性和可用性就是矛盾的。

在分布式系统中，不会同时具备CAP三个特性，只能同时具备CAP中的两个特性。那么CAP理论中的三个特性有哪些组合方式呢？很显然，有AP、CP、CA三种组合方式。

**1.AP**

放弃C，也就是放弃一致性，追求系统的可用性和分区容忍性。这也是实际工作中，大部分分布式系统在架构设计时的选择。

在实际场景中，大部分分布式系统会采用AP的方式，舍弃了一致性，并不代表就真的放弃了一致性。此时，更多的方案是采用最终一致性，允许多个节点的数据在一定的时间内存在差异，一段时间后达到数据一致的状态。

**2.CP**

放弃A，也就是放弃可用性，追求系统的一致性和分区容忍性。这种组合方式下对于数据的一致性要求比较高，追求的是强一致性。

在实际场景中，跨行转账业务需要每个银行系统都完成转账操作的整个事务才算完成，这是典型的CP方式。

**3.CA**

放弃P，也就是放弃分区容忍性，追求系统的一致性和可用性。此时的系统不会进行分区，也不会考虑网络不通和节点挂掉的问题。此时，主库和从库不再进行数据同步，此时的系统也不再是一个标准的分布式系统。

综上所述，在CAP理论中，只能同时满足其中的两个特性。

### Base理论概述

Base理论是对CAP理论中AP的一个扩展，它通过牺牲强一致性来获得可用性。当系统出现故障时，Base理论允许部分不可用，但是会保证核心功能可用，允许数据在一段时间内不一致，但是经过一段时间，数据最终是一致的。符合Base理论的事务可以称为柔性事务。

Base理论代表基本可用、软状态和最终一致性

### 基本可用

基本可用是指分布式系统出现故障时，允许其损失系统的部分可用性，比如响应时间或者功能上的损失，但是能够保证系统基本可用。

例如，在电商业务场景中，添加购物车和下单功能出现故障时，商品浏览功能仍然可用。

### 软状态

软状态是指允许系统中存在中间状态，这些中间状态不会影响系统的整体可用，允许系统各个节点之间的数据同步存在延迟。

例如，在电商业务场景中，订单中的“支付中”，“退款中”等状态就是中间状态，当达到一段时间后，就会变成“支付成功”或者“退款成功”的状态。

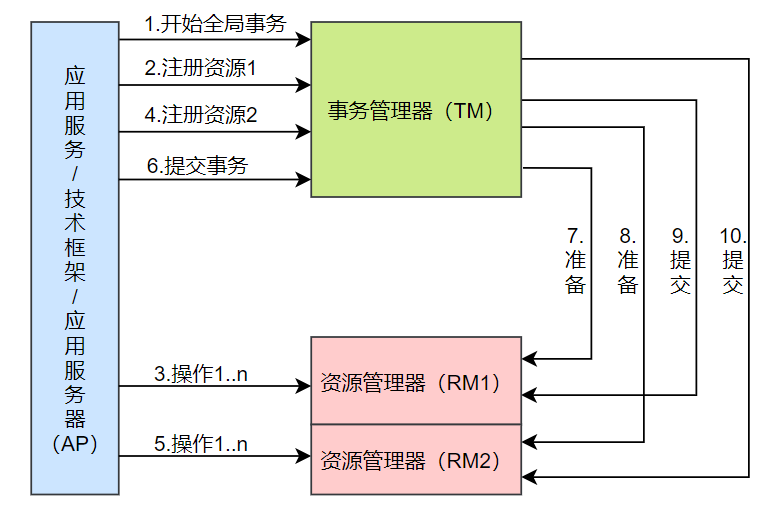
### 最终一致性

最终一致性是指系统中各个节点的数据副本，经过一段时间的同步，最终能够达到一个一致的状态。所以，最终一致性需要保证数据经过一段时间的同步，达到一致，并不要求各个节点的数据保持实时一致。

例如，在电商业务场景中，订单中的“支付中”，“退款中”等状态，最终会变成“支付成功”，“退款成功”的状态。经过一段时间的延迟，能够使得订单中的状态与最终的交易结果一致。

# 强一致性分布式事务解决方案

## DTP模型



在DTP模型中，主要定义了三个核心组件，分别为AP、TM、RM。

* AP：就是应用程序，可以理解为参与DTP分布式事务模型的应用程序。
* RM：就是资源管理器，可以理解为是一个数据库管理系统，也可以是一个消息服务管理器等。应用程序可以通过资源管理器对相应的资源进行有效的控制。相应的资源需要实现XA定义的接口。
* TM：就是事务管理器，负责协调和管理DTP模型中的整个事务，为应用程序提供编程接口，同时管理资源管理器。

其中，AP 可以和TM 以及 RM 通信，TM 和 RM 互相之间可以通信，DTP模型定义了XA接口，TM和RM能够通过XA接口进行双向通信。TM控制着全局事务，管理事务的生命周期并协调资源。RM控制和管理实际的资源。

## 2PC模型

2PC模型是指两阶段提交协议模型，这种模型将整个事务流程分为两个阶段，分别为准备阶段（Prepare Phase）和提交阶段（Commit Phase）。其中，2PC中的2指的是两个阶段，P是指Prepare，即准备阶段，C是指Commit，即提交阶段。

### 事务执行成功的流程

在2PC模型中，正常情况下，分布式事务执行成功时，整体上可以分为Prepare阶段和Commit阶段。在Prepare阶段事务管理器会向各资源管理器发送Prepare消息，在Commit阶段事务管理器会向各资源管理器发送Commit消息。

事务执行成功的流程图如图6-2至6-3所示。

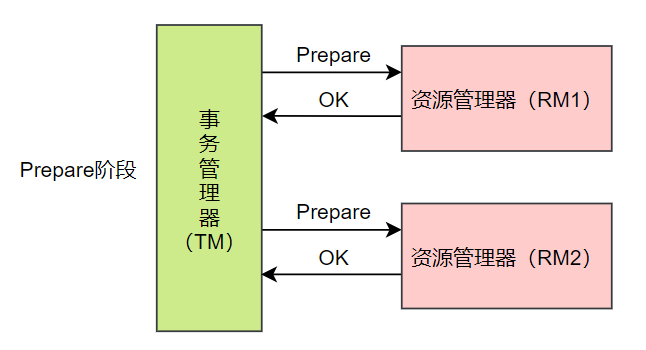


图6-2 2PC事务执行成功的Prepare阶段

由图6-2可以看出，在事务提交成功的情况下，2PC的Prepare阶段，由事务管理器向参与全局事务的资源管理器发送Prepare消息，资源管理器收到消息后，将事务写入本地的Redo Log和Undo Log日志，向事务管理器返回事务执行成功的状态。

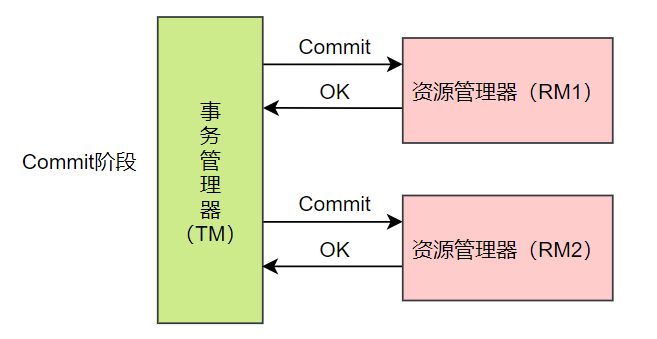


图6-3 2PC事务执行成功的Commit阶段

由图6-3可以看出，在事务执行成功的情况下，2PC的Commit阶段，由事务管理器向参与全局事务的资源管理器发送Commit消息，资源管理器收到后，提交本地事务，并将提交成功的消息返回给事务管理器，同时释放相应的锁资源。

### 事务执行失败的流程

在2PC模型中，当执行分布式事务失败时，例如，在Prepare阶段，某些资源管理器向事务管理器响应了Error消息，则在Commit阶段，事务管理器会向其他响应正常消息的资源管理器发送Rollback消息。

事务执行失败的流程图如图6-4至6-5所示。

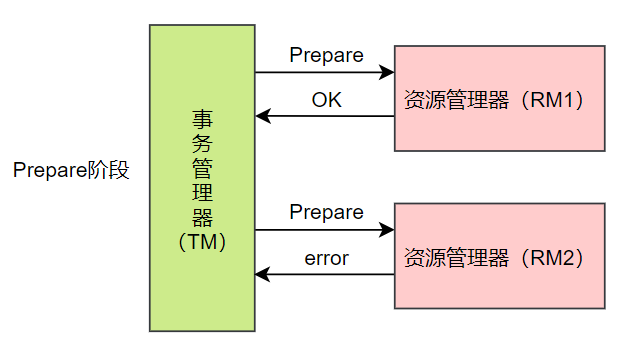


图6-4 2PC事务执行失败的Prepare阶段

由图6-4可以看出，在事务执行失败的情况下，2PC的Prepare阶段，事务管理器向资源管理器发送Prepare消息后，某些资源管理器收到消息后，将事务写入本地的Redo Log和Undo Log日志失败，会向事务管理器返回执行失败的error消息。

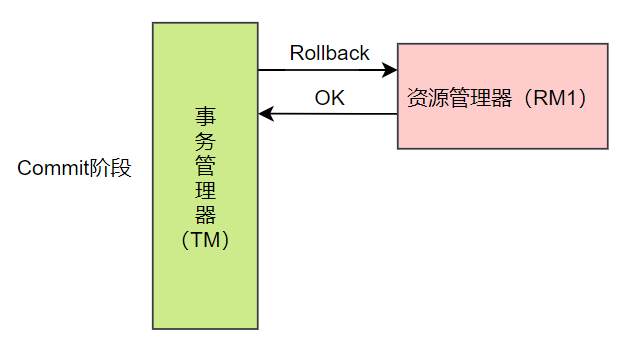


图6-5 2PC事务执行失败的Commit阶段

由图6-5可以看出，在事务执行失败的情况下，2PC的Commit阶段，事务管理器会向Prepare阶段将事务写入本地的Redo Log和Undo Log日志成功的资源管理器发送Rollback消息，对应的资源管理器收到事务管理器发送过来的Rollback消息后，回滚本地的事务，并将回滚成功的消息返回给事务管理器。

### 2PC模型存在的问题

值得注意的是，2PC模型存在着如下的缺点。

* 同步阻塞问题：事务的执行过程中，所有参与事务的节点都会对其占用的公共资源加锁，导致其他访问公共资源的进程或者线程阻塞。
* 单点故障问题：如果事务管理器发生故障，则资源管理器会一直阻塞。
* 数据不一致问题：如果在Commit阶段，由于网络或者部分资源管理器发生故障，导致部分资源管理器没有接收到事务管理器发送过来的Commit消息，引起数据不一致的问题。
* 无法解决的问题：如果在Commit阶段，事务管理器发出Commit消息后发生宕机，并且唯一接收到这条Commit消息的资源管理器也宕机了，则无法确认事务是否已经提交了。

## 3PC模型

3PC模型是指三阶段提交模型，是在2PC模型的基础上改进的版本，并且把2PC模型中的准备阶段一分为二，形成了三个阶段：CanCommit阶段、PreCommit阶段和doCommit或者doRollback阶段。大家可以类比2PC的执行流程，这里呢，由于时间关系3PC的执行流程就不讲了。

### 3PC模型中存在的问题

与2PC模型相比，3PC模型主要解决的单点故障问题，并减少事务执行过程中产生的阻塞现象。在3PC中，如果资源管理器无法及时收到来自事务管理器发出的消息，那么资源管理器就会执行提交事务的操作，而不会一直持有事务的资源并处于阻塞状态，但是这种机制会导致数据不一致的问题。

如果由于网路故障等原因，资源管理器没有及时收到事务管理器发出的消息，则资源管理器会在一段时间后提交事务，这就导致与其他接收到消息并执行了事务回滚操作的资源管理器的数据不一致。

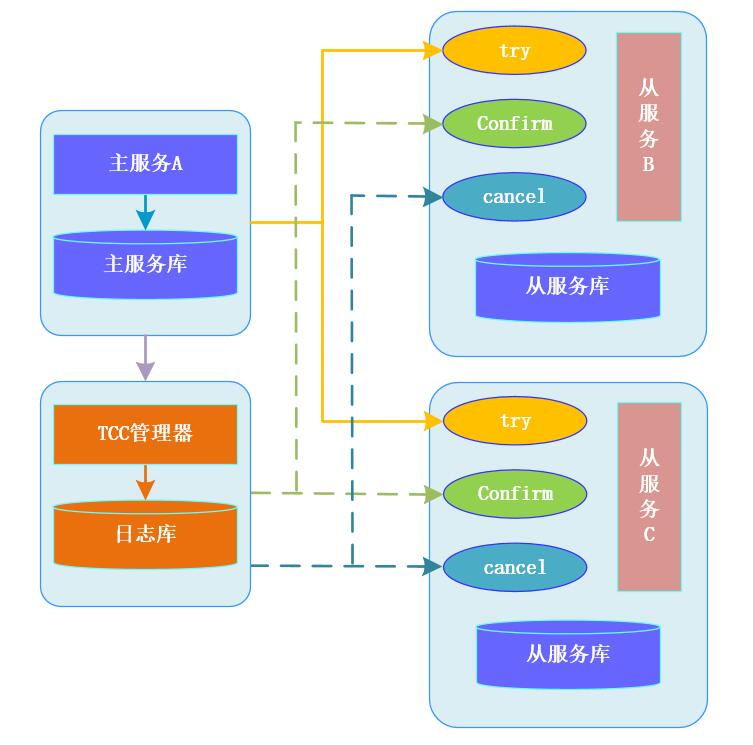
# 最终一致性分布式事务解决方案

### TCC解决方案

TCC是一种典型的解决分布式事务问题的方案，主要是解决跨服务调用场景下的分布式事务问题，广泛应用于分布式事务场景。

#### 执行流程

TCC从本质上将，是一种应用层实现的二阶段提交协议，主要分为三个阶段，分别为Try阶段、Confirm阶段和Cancel阶段。



（1）Try 阶段

不会执行任何业务逻辑，仅仅是做业务的一致性检查和预留相应的资源，这些资源能够和其他操作保持隔离。

（2）Confirm 阶段

做确认提交，当Try阶段所有分支事务执行成功后开始执行 Confirm阶段。通常情况下，采用TCC方案解决分布式事务时会认为 Confirm阶段是不会出错的。也就是说，只要Try阶段的操作执行成功了，Confirm阶段就一定会成功。如果Confirm阶段出错了，此时，就需要引入重试机制或人工处理，对出错的事务进行干预。

（3）Cancel阶段

是在业务执行异常或出现错误的情况下，需要回滚事务的操作，执行分支事务的取消操作，并且将Try阶段预留的资源释放。通常情况下，采用TCC方案解决分布式事务时，同样会认为Cancel阶段也是一定能够成功的。如果Cancel阶段出错了，也需要引入重试机制或人工处理，对出错的事务进行干预。

### 需要注意的问题

使用TCC方案解决分布式事务问题时，需要注意空回滚、幂等和悬挂的问题。

**1.空回滚问题**

如果使用TCC方案时，由于某种原因，系统没有调用Try阶段的方法，而是调用了Cancel阶段的方法，如果Cancel阶段没有做任何处理，就是造成空回滚的问题。

（1）空回滚问题出现的原因

出现空回滚的原因是当一个分支事务所在的服务器宕机或者网络发生异常，此分支事务调用失败，此时，并未执行此分支事务Try阶段的方法。当服务器或者网络恢复后，TCC分布式事务执行回滚操作，会调用分支事务的Cancel阶段的方法，如果Cancel阶段的方法没有处理这种情况，就会出现空回滚的问题。

（2）空回滚问题的解决方案

最终要的是要识别出空回滚操作。主要思路就是判断是否执行了Try阶段的方法。如果执行了Try阶段的方法，就不是空回滚，否则，就是空回滚。

具体方案就是：在主业务发起全局事务时，生成全局事务记录，并为全局事务记录生成一个全局唯一的ID，叫作全局事务ID，这个全局事务ID会贯穿整个分布式事务的执行流程。再创建一张分支事务记录表，用于记录分支事务，将全局事务ID和分支事务ID保存到分支事务表中。执行Try阶段的方法时，会向分支事务记录表中插入一条记录，其中包含全局事务ID和分支事务ID，表示执行了Try阶段。当事务回滚执行Cancel阶段的方法时，首先读取分支事务表中的数据，如果存在Try阶段插入的数据，则执行正常操作回滚事务，否则，为空回滚，不做任何操作。

**2.幂等问题**

由于服务器宕机、应用崩溃或者网络异常等原因，可能会出现方法调用超时的情况，为了保证方法的执行，往往会在TCC方案中加入超时重试机制。超时重试就有可能导致数据不一致的问题，所以，需要保证分支事务的执行以及TCC方案的Confirm阶段和Cancel阶段具备幂等性。

具体方案就是在分支事务记录表中增加事务的执行状态，每次执行分支事务以及Confirm阶段和Cancel阶段的方法时，都查询此事务的执行状态，以判断事务的幂等性。

**3.悬挂问题**

悬挂问题就是在TCC分布式事务中，Cancel阶段的方法先于Try阶段的方法执行。

（1）悬挂问题出现的原因

在TCC分布式事务中，通过RPC调用分支事务的Try阶段的方法时，会先注册分支事务，再执行RPC调用，如果此时发生服务器宕机、应用崩溃或者网路异常等情况，RPC调用就会超时。如果RPC调用超时，事务管理器就会通知对应的资源管理器回滚事务，可能资源管理器回滚完事务后，RPC请求达到了参与分支事务所在的业务方法，由于此时事务已经回滚，此时在Try阶段预留的资源就无法释放了。对于这种情况，就称为悬挂。总之，悬挂问题就是预留业务资源后，无法继续往下处理。

（2）解决悬挂问题的方案

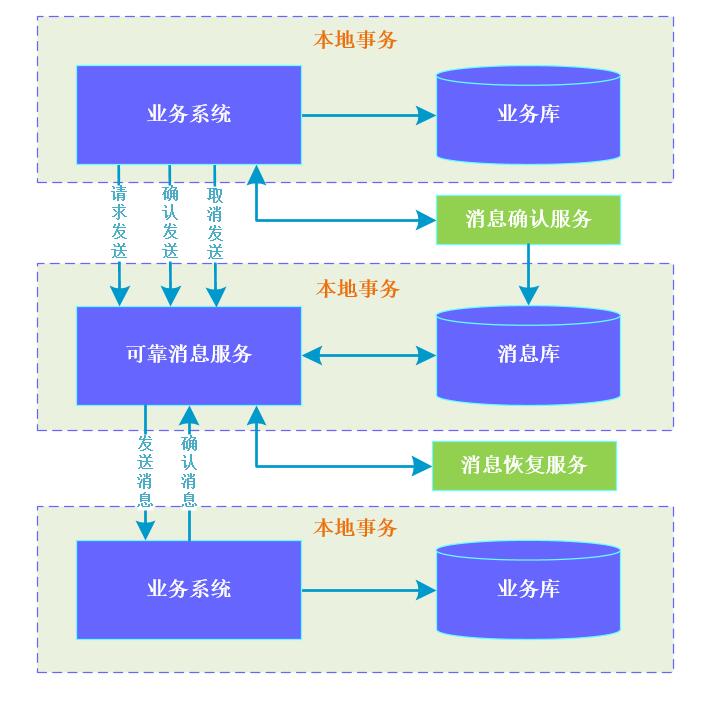
总体思路就是如果执行了Confirm阶段或者Cancel阶段的方法，则Try阶段的方法就不能再执行。具体方案就是：在执行Try阶段的方法时，判断分支记录表中是否已经存在同一全局事务下Confirm阶段或者Cancel阶段的事务记录，如果存在，则不再执行Try阶段的方法。

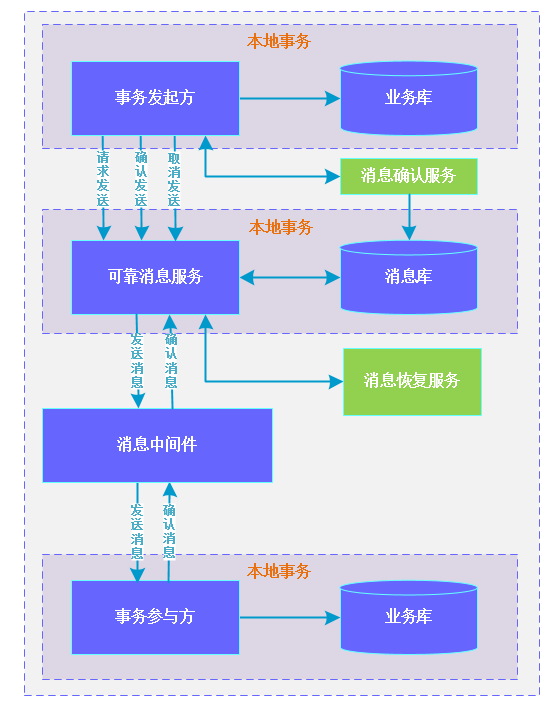
### 可靠消息最终一致性解决方案

可靠消息最终一致性分布式事务解决方案指的是事务的发起方执行完本地事务之后，发出一条消息，事务的参与方，也就是消息的消费者一定能过接收到消息并处理成功。这种方案主要强调的是只要事务发起方将消息发送给事务参与方，事务参与方就一定能够执行成功，事务最终能够达到一致的状态。

### 方案的执行流程

可靠消息最终一致性解决方案中，事务发起方执行完本地事务后会通过可靠消息服务将消息发送给事务参与方，事务参与方接收到消息后，一定能够成功执行。这里的可靠消息服务可以通过本地消息表实现，也可以通过RocketMQ消息队列实现。





事务发起方将消息放给可靠消息服务，这里的可靠消息服务可以基于本地数据表实现，也可以基于消息队列中间件实现。事务的参与方从可靠消息服务中接收消息。事务发起方和可靠消息服务之间、可靠消息服务和事务参与方之间都是通过网络进行通信的。由于网络本身的不稳定性，可能会造成分布式事务问题。所以，在实现上，需要引入消息确认服务和消息恢复服务。

消息确认服务会定期检测事务发起方业务的执行状态和消息库中的数据，如果发现事务发起方业务的执行状态与消息库中的数据不一致时，消息确认服务就会同步事务发起方的业务数据和消息库中的数据，使得事务发起方业务的执行状态和消息数据一致，确保事务发起方业务完成本地事务后消息一定会发送成功。

消息恢复服务会定期检测事务参与方业务的执行状态和消息库中的数据，如果发现事务参与方业务的执行状态与消息库中的数据不一致，这里的不一致，通常指的是事务参与方消费消息后，执行本地事务操作失败，导致事务参与方本地事务的执行状态与消息库中的数据不一致。消息恢复服务就会恢复消息库中消息的状态，使得消息的状态回滚为事务发起方发送消息成功，但未被事务参与方消费的状态。

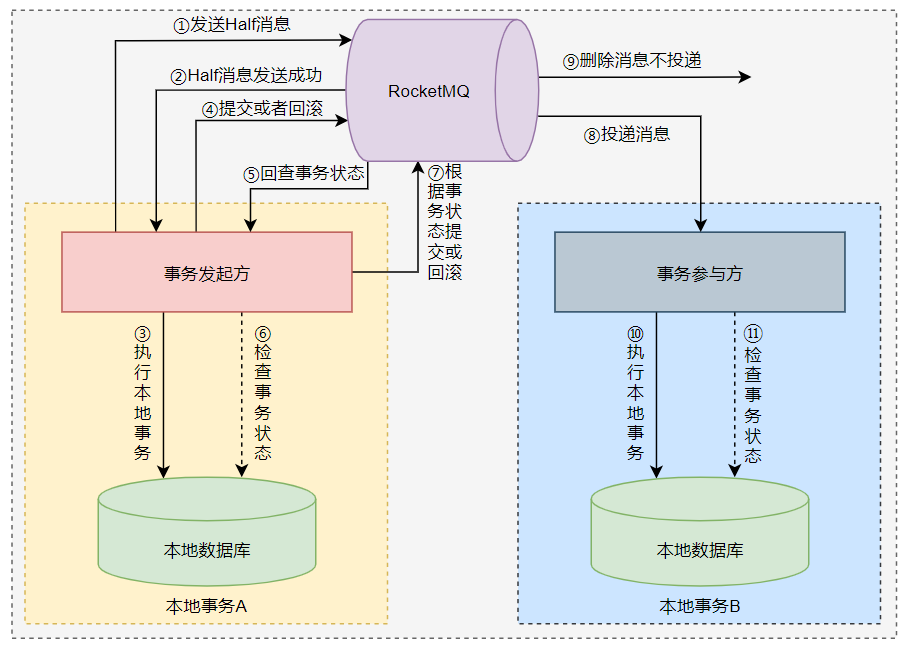


图10-4 RocketMQ实现可靠消息分布式事务解决方案的基本原理

整体流程如下所示。

1）事务发起方向RocketMQ发送Half消息。

2）RocketMQ向事务发起方响应Half消息发送成功。

3）事务发起方执行本地事务，向本地数据库中插入/更新/删除数据。

4）事务发起方向RocketMQ发送提交事务或者回滚事务的消息。

5）如果事务参与方未收到消息，或者执行事务失败，RocketMQ未删除保存的消息数据时，RocketMQ会回查事务发起方的接口，查询事务状态，以此确认是再次提交事务还是回滚事务。

6）事务发起方查询本地数据库，确认事务是否是执行成功的状态。

7）事务发起方根据查询出的事务状态，向RocketMQ发送提交事务或者回滚事务的消息。

8）如果第7步中事务发起方向RocketMQ发送的是提交事务的消息，则RocketMQ会向事务参与方投递消息。

9）如果第7步中事务发起方向RocketMQ发送的是回滚事务的消息，则RocketMQ不会向事务参与方投递消息，并且会删除内部存储的消息数据。

10）如果RocketMQ向事务参与方投递的是执行本地事务的消息，则事务参与方会执行本地事务，向本地数据库中插入/更新/删除数据。

11）如果RocketMQ想事务参与方投递的是查询本地事务状态的消息，则事务参与方会查询本地数据库中事务的执行状态。

在使用RocketMQ实现分布式事务时，上述流程中的主要部分都由RocketMQ自动实现了，开发人员只需要实现本地事务的执行逻辑和本地事务的回查方法，重点关注事务的执行状态即可。

### 需要注意的问题

使用可靠消息最终一致性方案解决分布式事务问题时，需要注意本地事务与消息发送的原子性问题、事务参与方接收消息的可靠性与幂等性问题。

**1.事务发送方本地事务与消息发送的原子性问题**

可靠消息最终一致性要求事务发起方的本地事务与消息发送的操作具有原子性，也就是事务发起方执行本地事务成功后一定要将消息发送出去，执行本地事务失败后一定要丢弃消息。执行本地事务和发送消息，要么都成功，要么都失败。

在实际的解决方案中，可以通过消息确认服务解决本地事务与消息发送的原子性问题。

**2.事务参与方接收消息的可靠性与幂等性问题**

由于服务器宕机、服务崩溃或网络异常等因素，导致事务参与方不能正常接收消息，或者接收消息后处理事务的过程中发生异常，无法将结果正确回传到消息库中。此时，可以通过消息恢复服务事务参与方可靠性的问题。

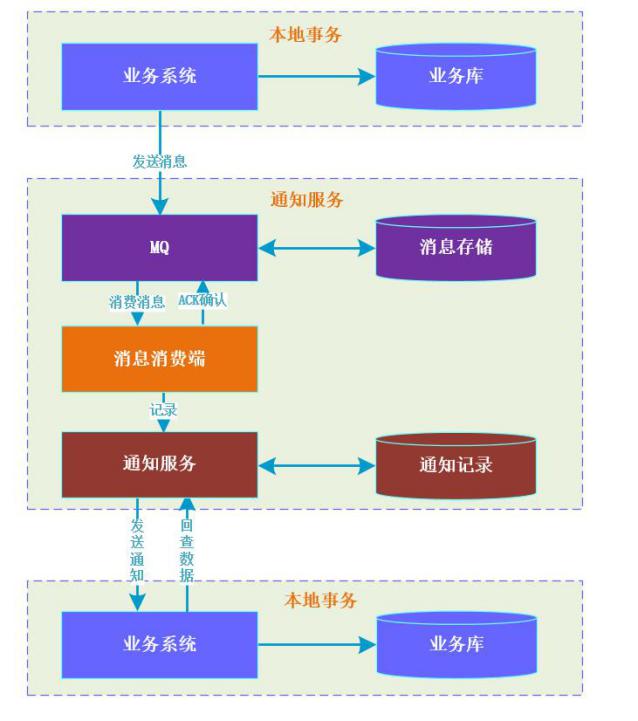
在实际场景中，可靠消息服务由于某种原因，可能会多次向事务参与方投递消息，如果事务参与方的方法不具有幂等性，就会造成消息重复消费的问题。解决的方案就是事务参与方的方法实现要具有幂等性。

### 最大努力通知型解决方案

当分布式事务跨越多个不同的系统，尤其是不同企业之间的系统时，解决分布式事务问题就需要用到最大努力通知型方案。最典型的使用场景就是：支付成功后，支付平台异步通知商户支付结果。

#### 执行流程

最大努力通知型分布式事务解决方案在执行的过程中，允许丢失消息，但需要业务主动方提供事务状态查询接口，以便业务被动方主动调用事务状态查询接口，恢复丢失的业务。



* 业务主动方在完成业务处理后，会向业务被动方发送消息通知，发送消息通知时，允许消息丢失。
* 在实现上，业务主动方可以设置时间阶梯型通知规则，在消息通知失败后，可以按照规则进行再次重复通知，直到设置的最大通知次数为止。
* 业务主动方需要提供查询接口供业务被动方按照需要查询，用于恢复丢失的消息。

### 需要注意的问题

业务被动方需要保证接收通知的方法的幂等性。

这里，关键是要业务主动方通过一定的机制最大程度的将业务的处理结果通知到业务被动方。必须实现如下两种机制。

（1）消息重复通知机制。

业务主动方发送消息通知后，业务被动方不一定能够接收到消息，所以，需要实现消息的重复通知机制。

（2）消息校对机制。

如果业务主动方尽最大努力都没有将消息通知到业务被动方，或者业务被动方接收到消息并执行完毕后，需要再次获取消息，此时，业务主动方需要提供查询消息的接口来满足业务被动方主动查询消息的需求。

另外，业务主动方在设计消息回查接口时，一定要注意接口的安全性和并发性。