分布式事务解决方案

- 1-常用分布式事务解决方案介绍
- 2-基于可靠消息的最终一致性方案介绍
- 3-最终一致性设计与实现——独立消息服务
- 4-最大努力通知方案(定期校对)
- 5-TCC型方案

1. 分布式事务解决方案

1.1 什么是事务?

事务由一组操作构成，我们希望这组操作能够全部正确执行，如果这一组操作中的任意一个步骤发生错误，那么就需要回滚之前已经完成的操作。也就是同一个事务中的所有操作，要么全都正确执行，要么全都不要执行。

事务类型

事务可以分为**本地事务**和**分布式事务**两种类型。这两种事务类型是根据访问并更新的数据资源的多少来进行区分的。

本地事务是在单个数据源上进行数据的访问和更新，而分布式事务是跨越多个数据源来进行数据的访问和更新。在这里要说的事务是基于数据库这种数据源的。

1.1.1 事务的四大特性 ACID

说到事务，就不得不提一下事务著名的四大特性。

- 原子性Atomic 原子性要求，事务是一个不可分割的执行单元，事务中的所有操作要么全都执行，要么全都不执行。
- 一致性 Consistency 一致性要求，事务在开始前和结束后，数据库的完整性约束没有被破坏。
- 隔离性Isolation 事务的执行是相互独立的，它们不会相互干扰，一个事务不会看到另一个正在运行过程中的事务的数据。
- 持久性Durability 持久性要求，一个事务完成之后，事务的执行结果必须是持久化保存的。即使数据库发生崩溃，在数据库恢复后事务提交的结果仍然不会丢失。

1.1.2.事务的隔离级别

这里扩展一下，对事务的**隔离性**做一个详细的解释。

在事务的四大特性ACID中，要求的隔离性是一种严格意义上的隔离，也就是多个事务是串行执行的，彼此之间不会受到任何干扰。这确实能够完全保证数据的安全性，但在实际业务系统中，这种方式性能不高。因此，数据库定义了四种隔离级别，隔离级别和数据库的性能是呈反比的，隔离级别越低，数据库性能越高，而隔离级别越高，数据库性能越差。

1.1.3.事务并发执行会出现的问题

我们先来看一下在不同的隔离级别下，数据库可能会出现的问题：

1. 更新丢失 当有两个并发执行的事务，更新同一行数据，那么有可能一个事务会把另一个事务的更新覆盖掉。 当数据库没有加任何锁操作的情况下会发生。
2. 脏读 一个事务读到另一个尚未提交的事务中的数据。 该数据可能会被回滚从而失效。 如果第一个事务拿着失效的数据去处理那就发生错误了。
3. 不可重复读 不可重复度的含义：一个事务对同一行数据读了两次，却得到了不同的结果。它具体分为如下两种情况：
    - 虚读：在事务1两次读取同一记录的过程中，事务2对该记录进行了修改，从而事务1第二次读到了不一样的记录。
    - 幻读：事务1在两次查询的过程中，事务2对该表进行了插入、删除操作，从而事务1第二次查询的结果发生了变化。

1.1.4.数据库的四种隔离级别

数据库一共有如下四种隔离级别：

1. Read uncommitted 读未提交 在该级别下，一个事务对一行数据修改的过程中，不允许另一个事务对该行数据进行修改，但允许另一个事务对该行数据读。 因此本级别下，不会出现更新丢失，但会出现脏读、不可重复读。
2. Read committed 读提交 在该级别下，未提交的写事务不允许其他事务访问该行，因此不会出现脏读；但是读取数据的事务允许其他事务的访问该行数据，因此会出现不可重复读的情况。
3. Repeatable read 重复读 在该级别下，读事务禁止写事务，但允许读事务，因此不会出现同一事务两次读到不同的数据的情况（不可重复读），且写事务禁止其他一切事务。
4. Serializable 序列化 该级别要求所有事务都必须串行执行，因此能避免一切因并发引起的问题，但效率很低。

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为Read Committed。它能够避免脏读取，而且具有较好的并发性能。尽管它会导致不可重复读、幻读和第二类丢失更新这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制。

1.2 什么是分布式事务?

到此为止，所介绍的事务都是基于单数据库的本地事务，目前的数据库仅支持单库事务，并不支持跨库事务。而随着微服务架构的普及，一个大型业务系统往往由若干个子系统构成，这些子系统又拥有各自独立的数据库。往往一个业务流程需要由多个子系统共同完成，而且这些操作可能需要在一个事务中完成。在微服务系统中，这些业务场景是普遍存在的。此时，我们就需要在数据库之上通过某种手段，实现支持跨数据库的事务支持，这也就是大家常说的“分布式事务”。

这里举一个分布式事务的典型例子——用户下单过程。 当我们的系统采用了微服务架构后，一个电商系统往往被拆分成如下几个子系统：商品系统、订单系统、支付系统、积分系统等。整个下单的过程如下：

1. 用户通过商品系统浏览商品，他看中了某一项商品，便点击下单
2. 此时订单系统会生成一条订单
3. 订单创建成功后，支付系统提供支付功能

4. 当支付完成后，由积分系统为该用户增加积分

上述步骤2、3、4需要在一个事务中完成。对于传统单体应用而言，实现事务非常简单，只需将这三个步骤放在一个方法A中，再用Spring的@Transactional注解标识该方法即可。Spring通过数据库的事务支持，保证这些步骤要么全都执行完成，要么全都不执行。但在这个微服务架构中，这三个步骤涉及三个系统，涉及三个数据库，此时我们必须在数据库和应用系统之间，通过某项黑科技，实现分布式事务的支持。

分布式系统会把一个应用系统拆分为可独立部署的多个服务，因此需要服务与服务之间远程协作才能完成事务操作， 这种分布式系统环境下由不同服务之间通过网络远程协作完成事务称之为分布式事务。

酸碱平衡

ACID能够保证事务的强一致性，即数据是实时一致的。这在本地事务中是没有问题的，在分布式事务中，强一致性会极大影响分布式系统的性能，因此**分布式系统中遵循BASE理论**即可。但分布式系统的不同业务场景对一致性的要求也不同。如交易场景下，就要求强一致性，此时就需要遵循ACID理论，而在注册成功后发送短信验证码等场景下，并不需要实时一致，因此遵循BASE理论即可。因此要根据具体业务场景，在ACID和BASE之间寻求平衡。

1.2.1 分布式数据一致性

在分布式系统中，为了保证数据的高可用，通常会将数据保留多个副本(replica)，这些副本会放置在不同的物理的机器上。

什么是数据一致性

在数据有多份副本的情况下，如果网络、服务器或者软件出现故障，会导致部分副本写入成功，部分副本写入失败。这就造成各个副本之间的数据不一致，数据内容冲突。

造成事实上的数据不一致。

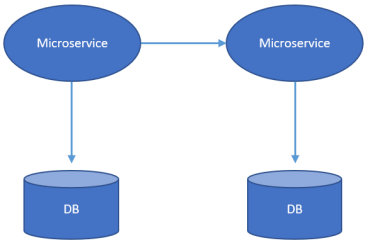
1.2.2 数据一致性模型

一些分布式系统通过复制数据来提高系统的可靠性和容错性，并且将数据的不同的副本存放在不同的机器。

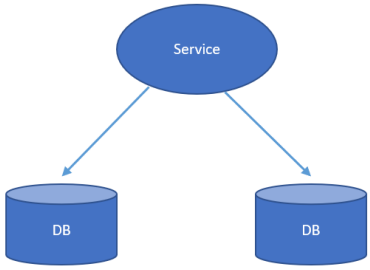
- **强一致性**： 当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据 CAP 理论，这种实现需要牺牲可用性。
- **弱一致性**： 系统并不保证续进程或者线程的访问都会返回最新的更新过的值。用户读到某一操作对系统特定数据的更新需要一段时间，我们称这段时间为“不一致性窗口”。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。
- **最终一致性**： 是弱一致性的一种特例。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。DNS 是一个典型的最终一致性系统。

分布式事务产生的场景

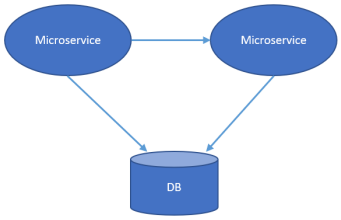
1. 典型的场景就是微服务架构



2. 单体系统访问多个数据库实例



3. 多个应用系统访问同一个数据库



1.3 CAP定理

1.3.1 标准分布式事务缺陷

- 效率非常低
    1. 全局事务方式下，全局事务管理器（TM）通过XA接口使用二阶段提交协议（ 2PC ）与资源层（如数据库）进行交互。使用全局事务，数据被Lock的时间跨整个事务，直到全局事务结束。
    2. 2PC 是反可伸缩模式，在事务处理过程中，参与者需要一直持有资源直到整个分布式事务结束。这样，当业务规模越来越大的情况下，2PC 的局限性就越来越明显，系统可伸缩性会变得很差。
    3. 与本地事务相比，XA 协议的系统开销相当大，因而应当慎重考虑是否确实需要分布式事务。而且只有支持 XA 协议的资源才能参与分布式事务。

1.3.2 CAP理论详解

- CAP原则是NOSQL数据库的基石。
- CAP原则又称CAP定理，指的是在一个分布式系统中， Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。
    1. 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）

2. 可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）
3. 分区容忍性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。(容忍网络中断)

1.4. BASE理论

> BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的简写，BASE是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的结论，是基于CAP定理逐步演化而来的，其核心思想是即使无法做到强一致性（Strong consistency），**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）。接下来我们着重对BASE中的三要素进行详细讲解。

- BA: Basic Availability 基本业务可用性（支持分区失败）:分布式系统在出现不可预知故障的时候，允许损失部分可用性
    - 响应时间上的损失：正常情况下，一个在线搜索引擎需要0.5秒内返回给用户相应的查询结果，但由于出现异常（比如系统部分机房发生断电或断网故障），查询结果的响应时间增加到了1~2秒。
    - 功能上的损失：正常情况下，在一个电子商务网站上进行购物，消费者几乎能够顺利地完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面。
- S: Soft state 柔性状态（状态允许有短时间不同步，异步）
- E: Eventual consistency 最终一致性（最终数据是一致的，但不是实时一致）
    - 因果一致性：因果一致性是指，如果进程A在更新完某个数据项后通知了进程B，那么进程B之后对该数据项的访问都应该能够获取到进程A更新后的最新值，并且如果进程B要对该数据项进行更新操作的话，务必基于进程A更新后的最新值，即不能发生丢失更新情况。与此同时，与进程A无因果关系的进程C的数据访问则没有这样的限制。
    - 读己之所写：读己之所写是指，进程A更新一个数据项之后，它自己总是能够访问到更新过的最新值，而不会看到旧值。也就是说，对于单个数据获取者而言，其读取到的数据一定不会比自己上次写入的值旧。因此，读己之所写也可以看作是一种特殊的因果一致性。
    - 会话一致性：会话一致性将对系统数据的访问过程框定在了一个会话当中：系统能保证在同一个有效的会话中实现"读己之所写"的一致性，也就是说，执行更新操作之后，客户端能够在同一个会话中始终读取到该数据项的最新值。
    - 单调读一致性：单调读一致性是指如果一个进程从系统中读取出一个数据项的某个值后，那么系统对于该进程后续的任何数据访问都不应该返回更旧的值。
    - 单调写一致性：单调写一致性是指，一个系统需要能够保证来自同一个进程的写操作被顺序地执行。

> 原子性（A）与持久性（D）必须根本保障 为了可用性、性能与降级服务的需要，只有降低一致性(C) 与 隔离性(I) 的要求

2. 分布式事务协议

下面介绍几种实现分布式事务的协议。

2.1 两阶段提交协议2PC

二阶段提交(Two-phaseCommit)是指，在计算机网络以及数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务提交时保持一致性而设计的一种算法(Algorithm)。通常，二阶段提交也被称为是一种协议(Protocol))。在分布式系统中，每个节点虽然可以知晓自己的操作时成功或者失败，却无法知道其他节点的操作的成功或失败。当一个事务跨越多个节点时，为了保持事务的ACID特性，需要引入一个作为协调者的组件来统一掌控所有节点(称作参与者)的操作结果并最终指示这些节点是否要把操作结果进行真正的提交(比如将更新后的数据写入磁盘等等)。因此，二阶段提交的算法思路可以概括为：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。

两阶段提交分为以下两个阶段：

- 准备阶段（Prepare Phase）
- 提交阶段（Commit Phase）

在两阶段提交协议中，系统一般包含两类角色：

- 协调者（Coordinator），通常一个系统中只有一个；负责决策整个分布式事务的提交和回滚。
- 参与者（Participant），一般包含多个，在数据存储系统中可以理解为数据副本的个数，负责自己本地事务的提交和回滚

一个基于两阶段提交协议的分布式事务框架

https://github.com/codingapi/tx-lcn/

常用的关系型数据库支持两阶段提交协议。

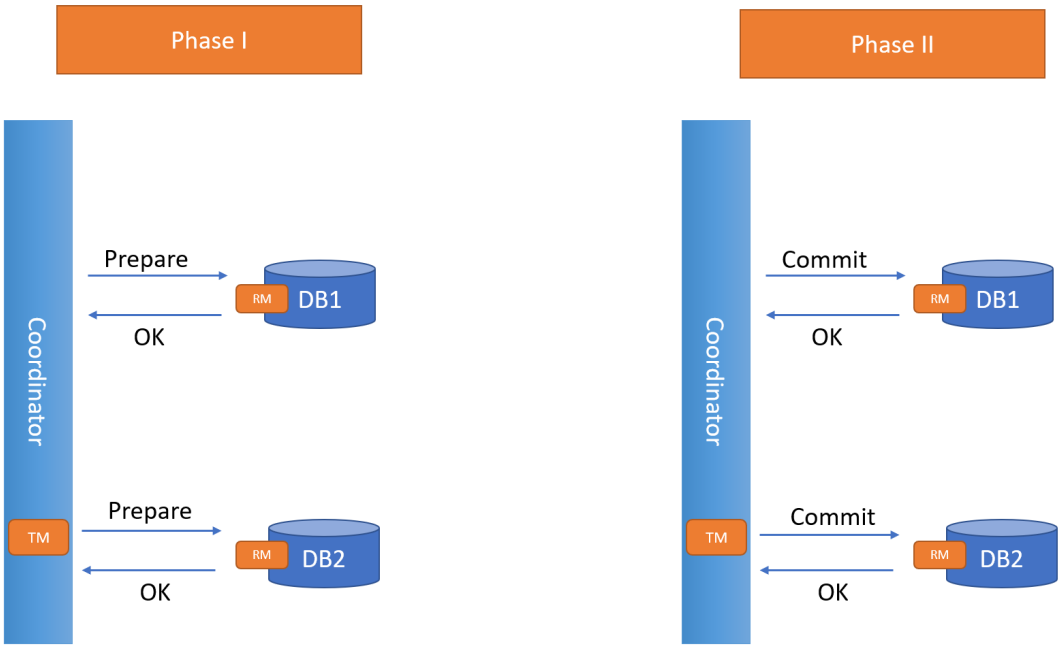
2.1.1 准备阶段

- 事务协调者(事务管理器)给每个参与者(资源管理器)发送Prepare消息，每个参与者要么直接返回失败(如权限验证失败)，要么在本地执行事务，写本地的redo和undo日志，但不提交，到达一种"万事俱备，只欠东风"的状态。
- 一般讲准备阶段分为以下三个阶段
    1. 协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应。
    2. 参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。（注意：若成功这里其实每个参与者已经执行了事务操作）
    3. 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个"同意"消息；如果参与者节点的事务操作实际执行失败，则它返回一个"中止"消息。

2.1.2 提交阶段

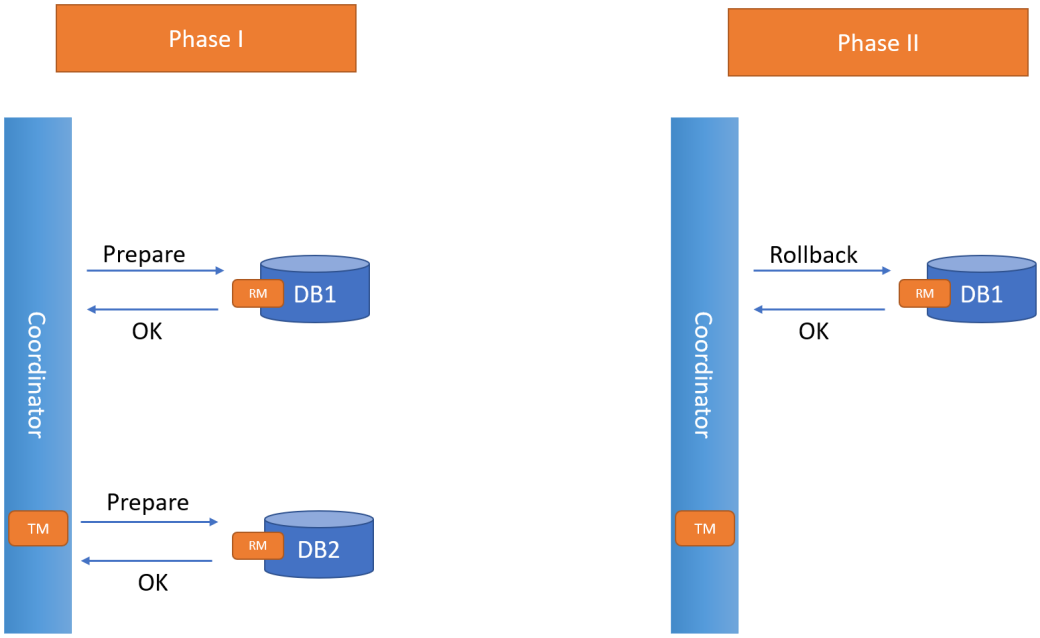
- 如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(Rollback)消息；否则，发送提交(Commit)消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意:必须在最后阶段释放锁资源)
- 当协调者节点从所有参与者节点获得的相应消息都为"同意"时:
    1. 协调者节点向所有参与者节点发出"正式提交(commit)"的请求。
    2. 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。
    3. 参与者节点向协调者节点发送"完成"消息。

4. 协调者节点受到所有参与者节点反馈的"完成"消息后，完成事务。



- 如果任一参与者节点在第一阶段返回的响应消息为"中止"，或者 协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时：

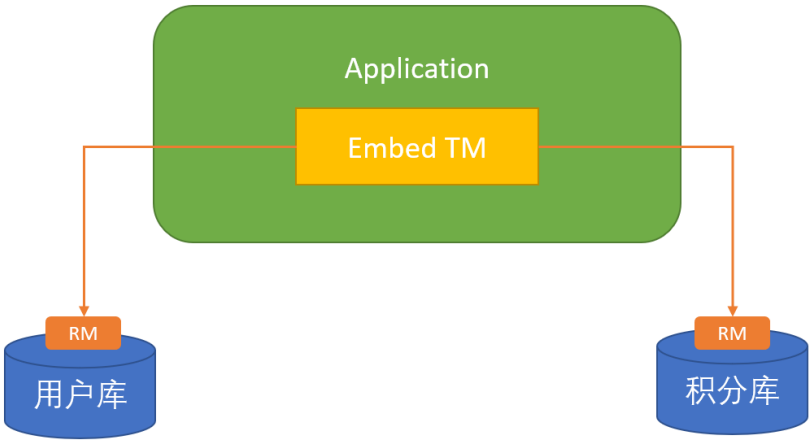
  1. 协调者节点向所有参与者节点发出"回滚操作(rollback)"的请求。
  2. 参与者节点利用之前写入的Undo信息执行回滚，并释放在整个事务期间内占用的资源。
  3. 参与者节点向协调者节点发送"回滚完成"消息。
  4. 协调者节点受到所有参与者节点反馈的"回滚完成"消息后，取消事务。



1. 解决方案

- **XA 方案**

  2PC的传统方案是在数据库层面实现的，如Oracle、MySQL都支持2PC协议，为了统一标准减少行业内不必要的对接成本，需要制定标准化的处理模型及接口标准，国际开放标准组织Open Group定义分布式事务处理模型**DTP**（Distributed Transaction Processing Reference Model）。



执行流程如下： 1、应用程序（AP）持有用户库和积分库两个数据源。 2、应用程序（AP）通过TM通知用户库RM新增用户，同时通知积分库RM为该用户新增积分，RM此时并未提交事务，此时用户和积分资源锁定。 3、TM收到执行回复，只要有一方失败则分别向其他RM发起回滚事务，回滚完毕，资源锁释放。 4、TM收到执行回复，全部成功，此时向所有RM发起提交事务，提交完毕，资源锁释放。

DTP 模型定义TM和RM之间通信的接口规范叫XA，简单理解为数据库提供的2PC接口协议，基于数据库的XA协议来实现的2PC又称为XA方案。

XA方案的问题

1. 需要本地数据库支持XA协议
2. 资源锁需要等到两个阶段结束才能结束， 性能差。

XA : XA是一个规范或是一个事务的协议.XA协议由Tuxedo首先提出的，并交给X/Open组织，作为资源管理器（数据库）与事务管理器的接口标准.

XA规范定义了:

1. TransactionManager : 这个TransactionManager可以通过管理多个ResourceManager来管理多个Resouce,也就是管理多个数据源
2. XAResource : 针对数据资源封装的一个接口
3. 两段式提交 : 多数据源事务提交的机制

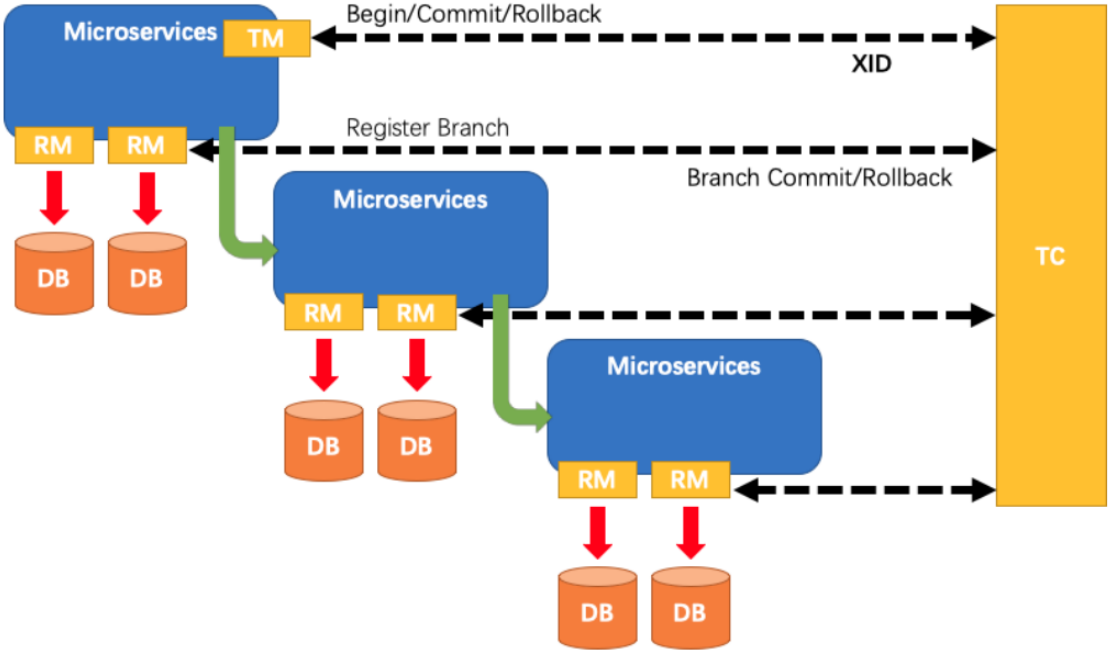
JTA(Java Transaction Manager) : 是Java规范,是XA在Java上的实现.

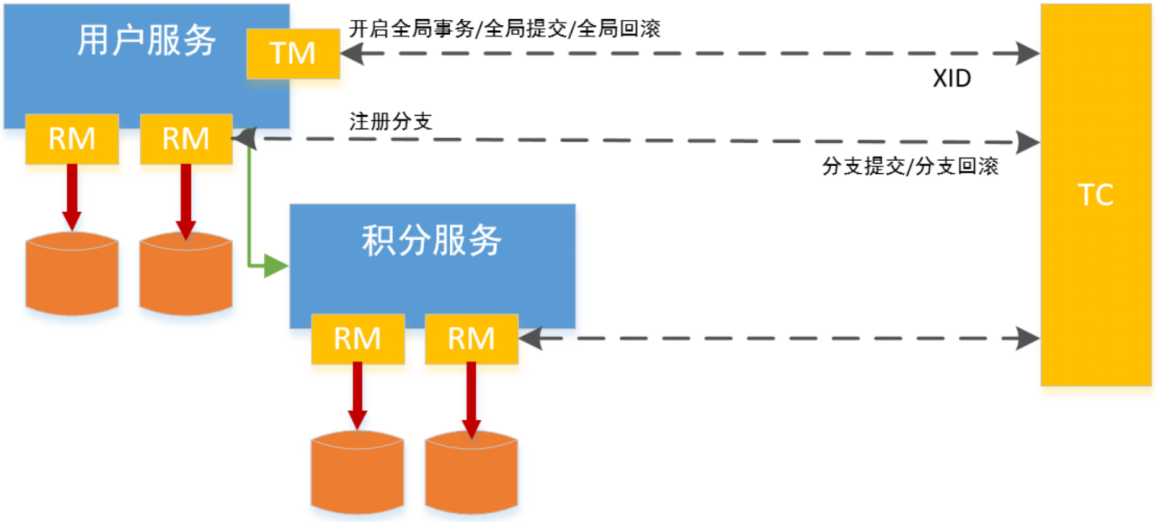
1. TransactionManager : 常用方法,可以开启,回滚,获取事务. begin(),rollback()...
2. XAResouce : 资源管理,通过Session来进行事务管理,commit(xid)...
3. XID : 每一个事务都分配一个特定的XID

- **Seata方案**

  Seata是阿里中间件团队发起的开源项目Fescar，后更名Seata，它是一个是开源的分布式事务框架。传统2PC的问题在Seata中得到了解决，它通过对本地关系数据库的分支事务的协调来驱动完成全局事务，是工作在应用层的中间件。主要优点是性能较好，且不长时间占用连接资源，它以高效并且对业务0入侵的方式解决微服务场景下面临的分布式事务问题，它目前提供**AT模式**（即2PC）及**TCC模式**的分布式事务解决方案。

Seata的设计思想如下： Seata的设计目标其一是对业务无入侵，因此从业务无入侵的2PC方案着手，在**传统2PC**的基础上演进，并解决2PC方案面临的问题。 Seata把一个分布式事务理解成一个包含来若干分支事务的全局事务。全局事务的职责是协调其下管辖的分支事务达成一致，要么一起成功提交，要么一起失败回滚。此外，通常分支事务本身就是一个关系数据库的本地事务：





执行流程

1. 用户服务的TM向TC申请开启一个全局事务，全局事务创建成功并生成一个全局唯一的XID。
2. 用户服务的RM向TC注册分支事务，该分支事务在用户服务执行新增用户逻辑，并将其纳入XID对应全局事务的管辖。
3. 用户服务执行分支事务，向用户表插入一条记录。
4. 逻辑执行到远程调用积分服务时（XID在微服务调用链路的上下文中传播）。积分服务的RM向TC注册分支事务，该分支事务执行增加积分的逻辑，并将其纳入XID对应全局事务的管辖。
5. 积分服务执行分支事务，向积分记录表插入一条记录，执行完毕后，返回用户服务。
6. 用户服务分支事务执行完毕。
7. TM向TC发起针对XID的全局提交或回滚决议。
8. TC调度XID下管辖的全部分支事务完成提交或回滚请求。

Seata实现2PC与传统2PC的差别： 架构层次方面，传统2PC方案的RM实际上是在数据库层，RM本质上就是数据库自身，通过XA协议实现，而Seata的RM是以jar包的形式作为中间件层部署在应用程序的这一侧的。 两阶段提交方面，传统2PC无论第二阶段的决议是commit还是rollbcak，事务性资源的锁都要保持到Phase2完成才释放。而Seata的做法是在Phase1就将本地事务提交，这样就可以省去Phase2持锁的时间，整体提高效率。

2. 两阶段提交的缺陷

1. 同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。
2. 单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）
3. 数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。
4. 二阶段无法解决的问题：协调者再发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

3. 两阶段提交无法解决的问题

当协调者出错，同时参与者也出错时，两阶段无法保证事务执行的完整性。 考虑协调者再发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。 那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

2.2 三阶段提交协议3PC

三阶段提交（Three-phase commit），也叫三阶段提交协议（Three-phase commit protocol），是二阶段提交（2PC）的改进版本。

三阶段提交协议与两阶段提交的不同点

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插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

> 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。

3. 相对于2PC，3PC主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

2.2.1 CanCommit阶段

3PC的CanCommit阶段其实和2PC的准备阶段很像。协调者向参与者发送commit请求，参与者如果可以提交就返回Yes响应，否则返回No响应。

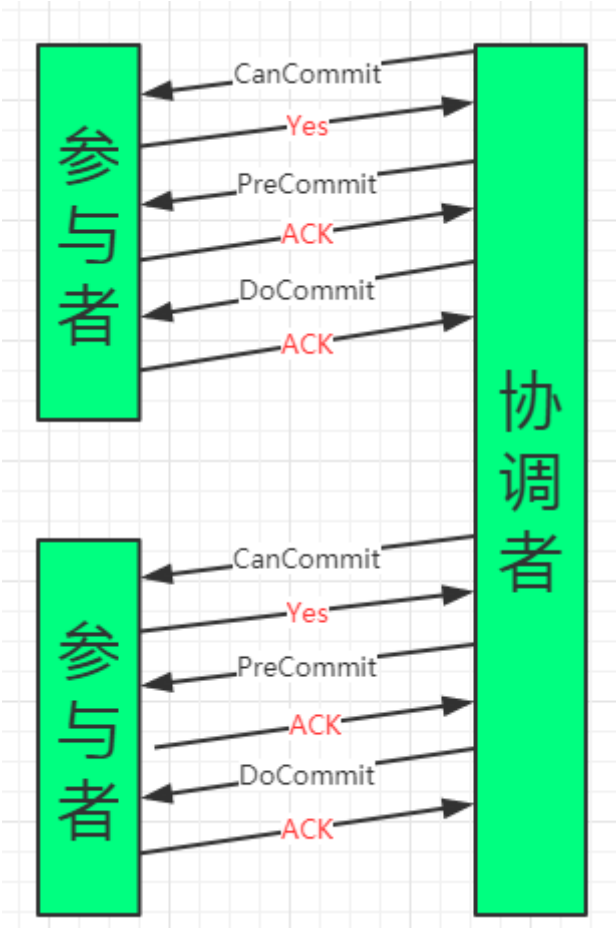
1. 事务询问 协调者向参与者发送CanCommit请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。
2. 响应反馈 参与者接到CanCommit请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回Yes响应，并进入预备状态。否则反馈No

2.2.2 PreCommit阶段

- 假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会进行事务的预执行。
    1. 发送预提交请求:协调者向参与者发送PreCommit请求，并进入Prepared阶段。
    2. 事务预提交:参与者接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。
    3. 响应反馈:如果参与者成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。
- 假如有任何一个参与者向协调者发送了No响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。
    1. 发送中断请求:协调者向所有参与者发送abort请求。
    2. 中断事务:参与者收到来自协调者的abort请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

2.2.3 doCommit阶段

- 执行提交
    1. 发送提交请求:协调接收到参与者发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送doCommit请求。
    2. 事务提交:参与者接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。
    3. 响应反馈:事务提交完之后，向协调者发送Ack响应。
    4. 完成事务:协调者接收到所有参与者的ack响应之后，完成事务。
- 中断事务:协调者没有接收到参与者发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。
    1. 发送中断请求:协调者向所有参与者发送abort请求
    2. 事务回滚:参与者接收到abort请求之后，利用其在阶段二记录的undo信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。
    3. 反馈结果:参与者完成事务回滚之后，向协调者发送ACK消息
    4. 中断事务:协调者接收到参与者反馈的ACK消息之后，执行事务的中断。



> 在doCommit阶段，如果参与者无法及时接收到来自协调者的doCommit或者rebort请求时，会在等待超时之后，会继续进行事务的提交。（其实这个应该是基于概率来决定的，当进入第三阶段时，说明参与者在第二阶段已经收到了PreCommit请求，那么协调者产生PreCommit请求的前提条件是他在第二阶段开始之前，收到所有参与者的CanCommit响应都是Yes。（一旦参与者收到了PreCommit，意味他知道大家其实都同意修改了）所以，一句话概括就是，当进入第三阶段时，由于网络超时等原因，虽然参与者没有收到commit或者abort响应，但是他有理由相信：成功提交的几率很大。）

2.3 分布式事务的解决方案

2.3.1 解决方案分类

- 刚性事务
  - 全局事务（标准的分布式事务）
- 柔性事务
  - 可靠消息最终一致（异步确何型）
  - TCC （两阶段型、补偿型）
  - 最大努力通知（非可靠消息 、 定期校对）
  - 纯补偿型（略）

2.3.2 方案1：全局事务（DTP模型）

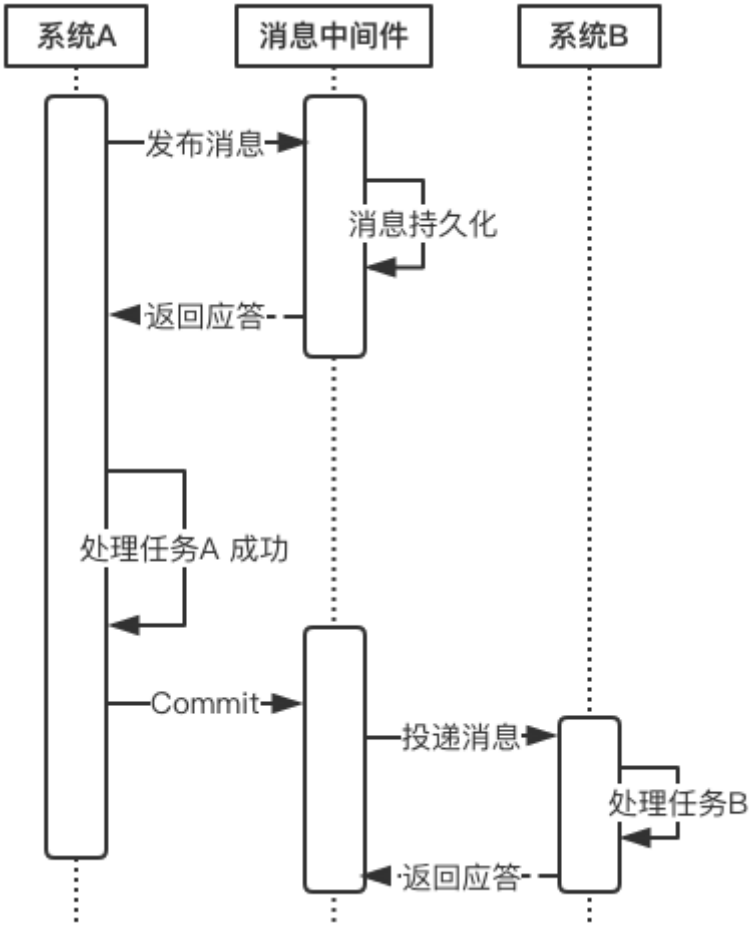
全局事务基于DTP模型实现。DTP是由X/Open组织提出的一种分布式事务模型——X/Open Distributed Transaction Processing Reference Model。它规定了要实现分布式事务，需要三种角色：

- AP：Application 应用系统 它就是我们开发的业务系统，在我们开发的过程中，可以使用资源管理器提供的事务接口来实现分布式事务。
- TM：Transaction Manager 事务管理器
  - 分布式事务的实现由事务管理器来完成，它会提供分布式事务的操作接口供我们的业务系统调用。这些接口称为TX接口。
  - 事务管理器还管理着所有的资源管理器，通过它们提供的XA接口来同一调度这些资源管理器，以实现分布式事务。
  - DTP只是一套实现分布式事务的规范，并没有定义具体如何实现分布式事务，TM可以采用2PC、3PC、Paxos等协议实现分布式事务。
- RM：Resource Manager 资源管理器
  - 能够提供数据服务的对象都可以是资源管理器，比如：数据库、消息中间件、缓存等。大部分场景下，数据库即为分布式事务中的资源管理器。
  - 资源管理器能够提供单数据库的事务能力，它们通过XA接口，将本数据库的提交、回滚等能力提供给事务管理器调用，以帮助事务管理器实现分布式的事务管理。
  - XA是DTP模型定义的接口，用于向事务管理器提供该资源管理器(该数据库)的提交、回滚等能力。
  - DTP只是一套实现分布式事务的规范，RM具体的实现是由数据库厂商来完成的。

> 1. 有没有基于DTP模型的分布式事务中间件?
> 2. DTP模型有啥优缺点?

2.3.3 方案2：可靠消息服务的分布式事务

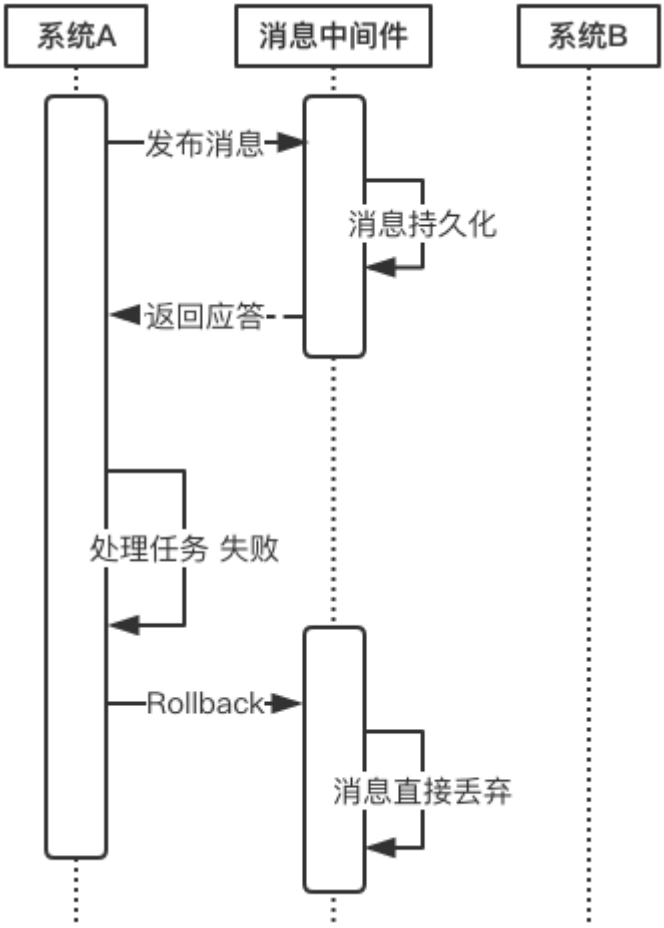
这种实现分布式事务的方式需要通过消息中间件来实现。假设有A和B两个系统，分别可以处理任务A和任务B。此时系统A中存在一个业务流程，需要将任务A和任务B在同一个事务中处理。下面来介绍基于消息中间件来实现这种分布式事务。



- 在系统A处理任务A前，首先向消息中间件发送一条消息
- 消息中间件收到后将该条消息持久化，但并不投递。此时下游系统B仍然不知道该条消息的存在。
- 消息中间件持久化成功后，便向系统A返回一个确认应答；
- 系统A收到确认应答后，则可以开始处理任务A；
- 任务A处理完成后，向消息中间件发送Commit请求。该请求发送完成后，对系统A而言，该事务的处理过程就结束了，此时它可以处理别的任务了。 但commit消息可能会在传输途中丢失，从而消息中间件并不会向系统B投递这条消息，从而系统就会出现不一致性。这个问题由消息中间件的事务回查机制完成，下文会介绍。
- 消息中间件收到Commit指令后，便向系统B投递该消息，从而触发任务B的执行；
- 当任务B执行完成后，系统B向消息中间件返回一个确认应答，告诉消息中间件该消息已经成功消费，此时，这个分布式事务完成。

> 上述过程可以得出如下几个结论：
>
> 1. 消息中间件扮演者分布式事务协调者的角色。
> 2. 系统A完成任务A后，到任务B执行完成之间，会存在一定的时间差。在这个时间差内，整个系统处于数据不一致的状态，但这短暂的不一致性是可以接受的，因为经过短暂的时间后，系统又可以保持数据一致性，满足BASE理论。

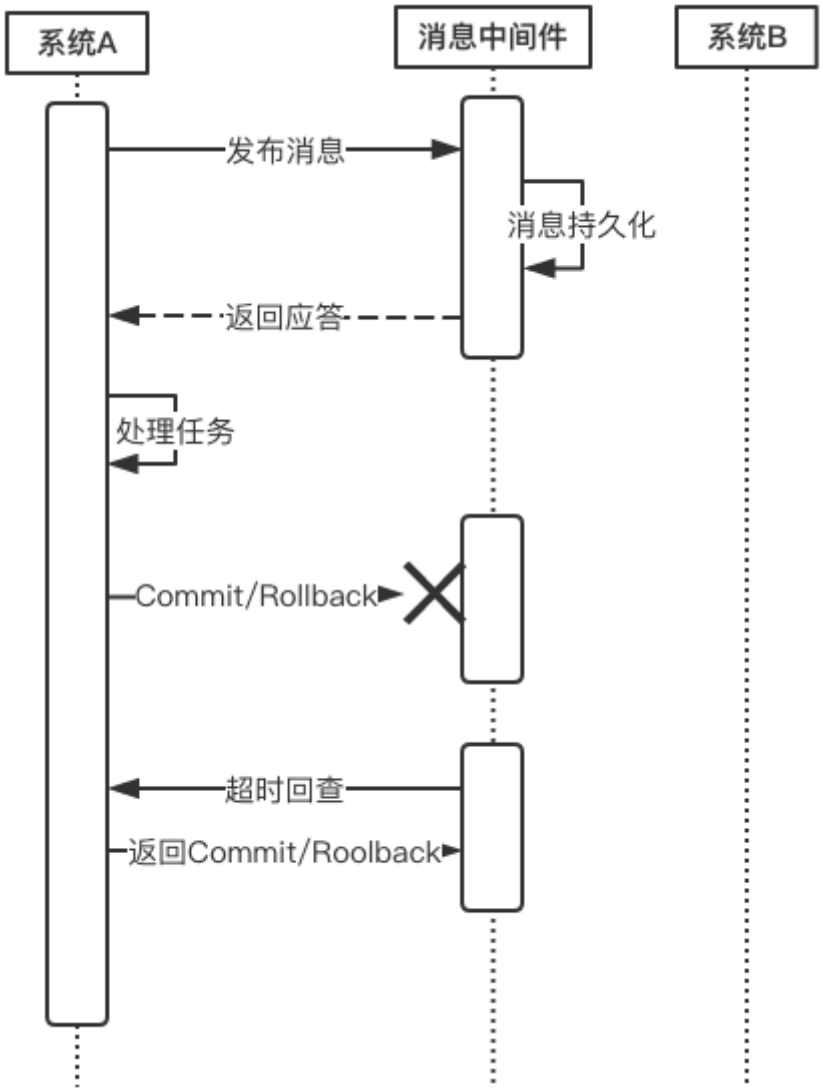
上述过程中，如果任务A处理失败，那么需要进入回滚流程，如下图所示：



- 若系统A在处理任务A时失败，那么就会向消息中间件发送Rollback请求。和发送Commit请求一样，系统A发完之后便可以认为回滚已经完成，它便可以去做其他的事情。
- 消息中间件收到回滚请求后，直接将该消息丢弃，而不投递给系统B，从而不会触发系统B的任务B。

> 此时系统又处于一致性状态，因为任务A和任务B都没有执行。

上面所介绍的Commit和Rollback都属于理想情况，但在实际系统中，Commit和Rollback指令都有可能在传输途中丢失。那么当出现这种情况的时候，消息中间件是如何保证数据一致性呢？——答案就是超时询问机制。



系统A除了实现正常的业务流程外，还需提供一个事务询问的接口，供消息中间件调用。当消息中间件收到一条事务型消息后便开始计时，如果到了超时时间也没收到系统A发来的Commit或Rollback指令的话，就会主动调用系统A提供的事务询问接口询问该系统目前的状态。该接口会返回三种结果：

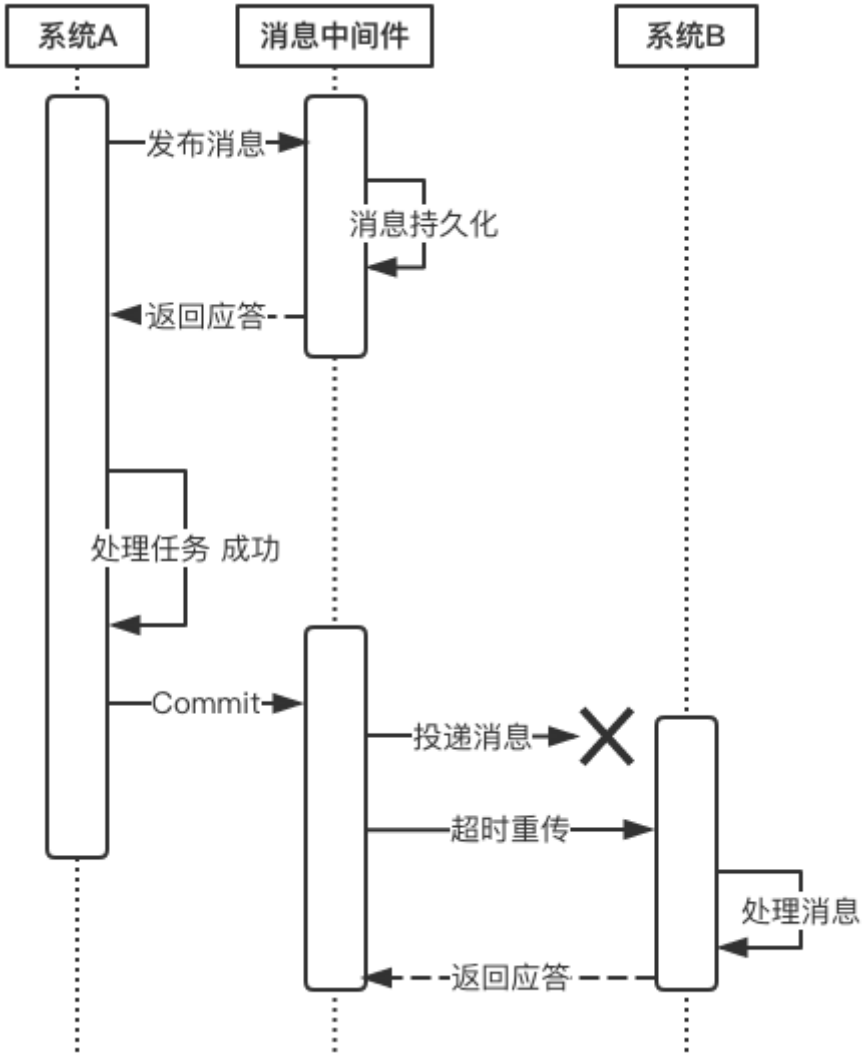
- 提交 若获得的状态是“提交”，则将该消息投递给系统B。
- 回滚 若获得的状态是“回滚”，则直接将条消息丢弃。
- 处理中 若获得的状态是“处理中”，则继续等待。

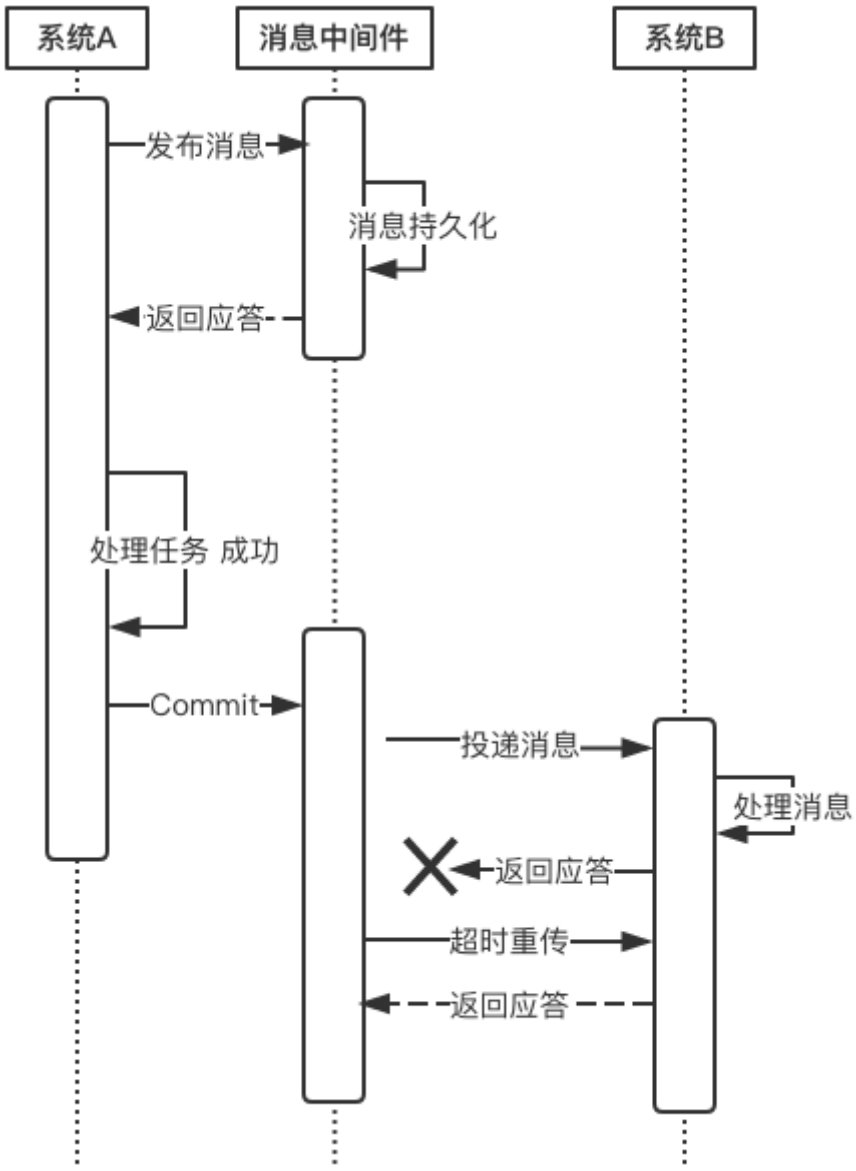
> 消息中间件的超时询问机制能够防止上游系统因在传输过程中丢失Commit/Rollback指令而导致的系统不一致情况，而且能降低上游系统的阻塞时间，上游系统只要发出Commit/Rollback指令后便可以处理其他任务，无需等待确认应答。而Commit/Rollback指令丢失的情况通过超时询问机制来弥补，这样大大降低上游系统的阻塞时间，提升系统的并发度。

下面来说一说消息投递过程的可靠性保证。 当上游系统执行完任务并向消息中间件提交了Commit指令后，便可以处理其他任务了，此时它可以认为事务已经完成，接下来消息中间件**一定会保证消息被下游系统成功消费掉！**那么这是怎么做到的呢？这由消息中间件的投递流程来保证。

消息中间件向下游系统投递完消息后便进入阻塞等待状态，下游系统便立即进行任务的处理，任务处理完成后便向消息中间件返回应答。消息中间件收到确认应答后便认为该事务处理完毕！

如果消息在投递过程中丢失，或消息的确认应答在返回途中丢失，那么消息中间件在等待确认应答超时之后就会重新投递，直到下游消费者返回消费成功响应为止。当然，一般消息中间件可以设置消息重试的次数和时间间隔，比如：当第一次投递失败后，每隔五分钟重试一次，一共重试3次。如果重试3次之后仍然投递失败，那么这条消息就需要人工干预。


系统A
消息中间件
系统B
发布消息
消息持久化
返回应答
处理任务 成功
Commit
投递消息
超时重传
处理消息
返回应答



> 有的同学可能要问：消息投递失败后为什么不回滚消息，而是不断尝试重新投递？

这就涉及到整套分布式事务系统的实现成本问题。 我们知道，当系统A将向消息中间件发送Commit指令后，它便去做别的事情了。如果此时消息投递失败，需要回滚的话，就需要让系统A事先提供回滚接口，这无疑增加了额外的开发成本，业务系统的复杂度也将提高。对于一个业务系统的设计目标是，在保证性能的前提下，最大限度地降低系统复杂度，从而能够降低系统的运维成本。

> 不知大家是否发现，上游系统A向消息中间件提交Commit/Rollback消息采用的是异步方式，也就是当上游系统提交完消息后便可以去做别的事情，接下来提交、回滚就完全交给消息中间件来完成，并且完全信任消息中间件，认为它一定能正确地完成事务的提交或回滚。然而，消息中间件向下游系统投递消息的过程是同步的。也就是消息中间件将消息投递给下游系统后，它会阻塞等待，等下游系统成功处理完任务返回确认应答后才取消阻塞等待。为什么这两者在设计上是不一致的呢？

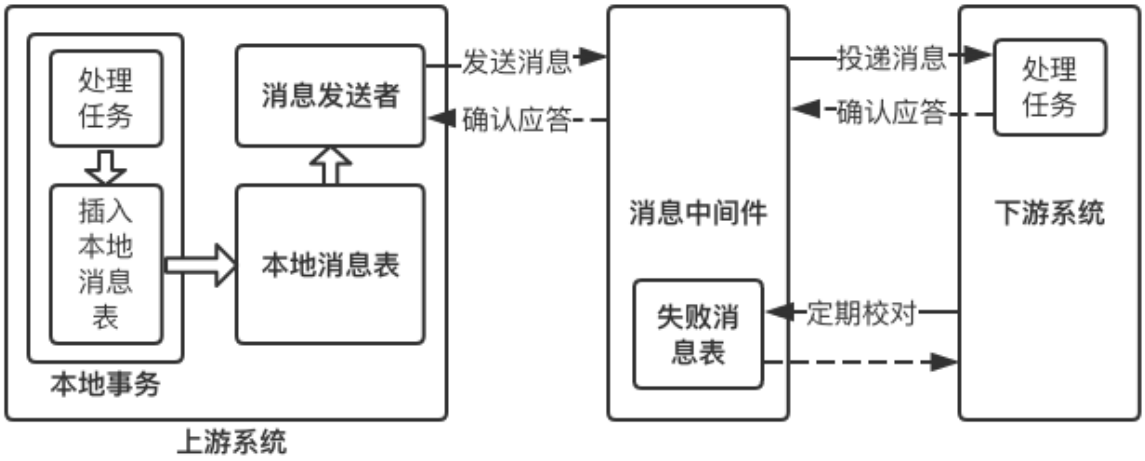
首先，上游系统和消息中间件之间采用异步通信是为了提高系统并发度。业务系统直接和用户打交道，用户体验尤为重要，因此这种异步通信方式能够极大程度地降低用户等待时间。此外，异步通信相对于同步通信而言，没有了长时间的阻塞等待，因此系统的并发性也大大增加。但异步通信可能会引起Commit/Rollback指令丢失的问题，这就由消息中间件的超时询问机制来弥补。

那么，消息中间件和下游系统之间为什么要采用同步通信呢？

异步能提升系统性能，但随之会增加系统复杂度；而同步虽然降低系统并发度，但实现成本较低。因此，在对并发度要求不是很高的情况下，或者服务器资源较为充裕的情况下，我们可以选择同步来降低系统的复杂度。 我们知道，消息中间件是一个独立于业务系统的第三方中间件，它不和任何业务系统产生直接的耦合，它也不和用户产生直接的关联，它一般部署在独立的服务器集群上，具有良好的可扩展性，所以不必太过于担心它的性能，如果处理速度无法满足我们的要求，可以增加机器来解决。而且，即使消息中间件处理速度有一定的延迟那也是可以接受的，因为前面所介绍的BASE理论就告诉我们了，我们追求的是最终一致性，而非实时一致性，因此消息中间件产生的时延导致事务短暂的不一致是可以接受的。

2.3.4 方案3：最大努力通知（定期校对）

最大努力通知也被称为定期校对，其实在方案二中已经包含，这里再单独介绍，主要是为了知识体系的完整性。这种方案也需要消息中间件的参与，其过程如下：



- 上游系统在完成任务后，向消息中间件同步地发送一条消息，确保消息中间件成功持久化这条消息，然后上游系统可以去做别的事情了；
- 消息中间件收到消息后负责将该消息同步投递给相应的下游系统，并触发下游系统的任务执行；
- 当下游系统处理成功后，向消息中间件反馈确认应答，消息中间件便可以将该条消息删除，从而该事务完成。

上面是一个理想化的过程，但在实际场景中，往往会出现如下几种意外情况：

1. 消息中间件向下游系统投递消息失败
2. 上游系统向消息中间件发送消息失败

对于第一种情况，消息中间件具有重试机制，我们可以在消息中间件中设置消息的重试次数和重试时间间隔，对于网络不稳定导致的消息投递失败的情况，往往重试几次后消息便可以成功投递，如果超过了重试的上限仍然投递失败，那么消息中间件不再投递该消息，而是记录在失败消息表中，消息中间件需要提供失败消息的查询接口，下游系统会定期查询失败消息，并将其消费，这就是所谓的“定期校对”。

如果重复投递和定期校对都不能解决问题，往往是因为下游系统出现了严重的错误，此时就需要人工干预。

对于第二种情况，需要在上游系统中建立消息重发机制。可以在上游系统建立一张本地消息表，并将 **任务处理过程** 和 **向本地消息表中插入消息** 这两个步骤放在一个本地事务中完成。如果向本地消息表插入消息失败，那么就会触发回滚，之前的任务处理结果就会被取消。如果这量步都执行成功，那么该本地事务就完成了。接下来会有一个专门的消息发送者不断地发送本地消息表中的消息，如果发送失败它会返回重试。当然，也要给消息发送者设置重试的上限，一般而言，达到重试上限仍然发送失败，那就意味着消息中间件出现严重的问题，此时也只有人工干预才能解决问题。

对于不支持事务型消息的消息中间件，如果要实现分布式事务的话，就可以采用这种方式。它能够通过**重试机制+定期校对**实现分布式事务，但相比于第二种方案，它达到数据一致性的周期较长，而且还需要在上游系统中实现消息重试发布机制，以确保消息成功发布给消息中间件，这无疑增加了业务系统的开发成本，使得业务系统不够纯粹，并且这些额外的业务逻辑无疑会占用业务系统的硬件资源，从而影响性能。

因此，尽量选择支持事务型消息的消息中间件来实现分布式事务，如RocketMQ。

2.3.5 方案4：TCC（两阶段型、补偿型）

TCC是Try、Confirm、Cancel三个词语的缩写，TCC要求每个分支事务实现三个操作：预处理Try、确认Confirm、撤销Cancel。Try操作做业务检查及资源预留，Confirm做业务确认操作，Cancel实现一个与Try相反的操作既回滚操作。TM首先发起所有的分支事务的try操作，任何一个分支事务的try操作执行失败，TM将会发起所有分支事务的Cancel操作，若try操作全部成功，TM将会发起所有分支事务的Confirm操作，其中Confirm/Cancel操作若执行失败，TM会进行重试。

TCC即为Try Confirm Cancel，它属于补偿型分布式事务。顾名思义，TCC实现分布式事务一共有三个步骤：

- Try：尝试待执行的业务
  - 这个过程并未执行业务，只是完成所有业务的一致性检查，并预留好执行所需的全部资源
- Confirm：执行业务
  - 这个过程真正开始执行业务，由于Try阶段已经完成了一致性检查，因此本过程直接执行，而不做任何检查。并且在执行的过程中，会使用到Try阶段预留的业务资源。
- Cancel：取消执行的业务
  - 若业务执行失败，则进入Cancel阶段，它会释放所有占用的业务资源，并回滚Confirm阶段执行的操作。

下面以一个转账的例子来解释下TCC实现分布式事务的过程。

> 假设用户A用他的账户余额给用户B发一个100元的红包，并且余额系统和红包系统是两个独立的系统。

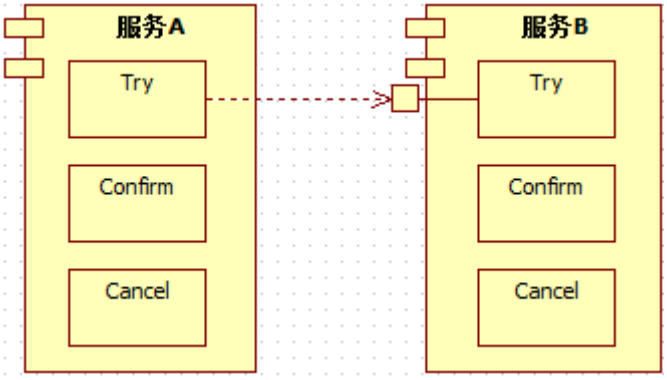
- Try
  - 创建一条转账流水，并将流水的状态设为**交易中**
  - 将用户A的账户中扣除100元（预留业务资源）
  - Try成功之后，便进入Confirm阶段
  - Try过程发生任何异常，均进入Cancel阶段
- Confirm
  - 向B用户的红包账户中增加100元
  - 将流水的状态设为**交易已完成**
  - Confirm过程发生任何异常，均进入Cancel阶段
  - Confirm过程执行成功，则该事务结束
- Cancel
  - 将用户A的账户增加100元
  - 将流水的状态设为**交易失败**

在传统事务机制中，业务逻辑的执行和事务的处理，是在不同的阶段由不同的部件来完成的：业务逻辑部分访问资源实现数据存储，其处理是由业务系统负责；事务处理部分通过协调资源管理器以实现事务管理，其处理由事务管理器来负责。二者没有太多交互的地方，所以，传统事务管理器的事务处理逻辑，仅需要着眼于事务完成（commit/rollback）阶段，而不必关注业务执行阶段。

TCC全局事务必须基于RM本地事务来实现全局事务

TCC服务是由Try/Confirm/Cancel业务构成的， 其Try/Confirm/Cancel业务在执行时，会访问资源管理器（Resource Manager，下文简称RM）来存取数据。这些存取操作，必须要参与RM本地事务，以使其更改的数据要么都commit，要么都rollback。

这一点不难理解，考虑一下如下场景：



假设图中的服务B没有基于RM本地事务（以RDBS为例，可通过设置auto-commit为true来模拟），那么一旦[B:Try]操作中途执行失败，TCC事务框架后续决定回滚全局事务时，该[B:Cancel]则需要判断[B:Try]中哪些操作已经写到DB、哪些操作还没有写到DB：假设[B:Try]业务有5个写库操作，[B:Cancel]业务则需要逐个判断这5个操作是否生效，并将生效的操作执行反向操作。

不幸的是，由于[B:Cancel]业务也有n（0<=n<=5）个反向的写库操作，此时一旦[B:Cancel]也中途出错，则后续的[B:Cancel]执行任务更加繁重。因为，相比第一次[B:Cancel]操作，后续的[B:Cancel]操作还需要判断先前的[B:Cancel]操作的n（0<=n<=5）个写库中哪几个已经执行、哪几个还没有执行，这就涉及到了幂等性问题。而对幂等性的保障，又很可能还需要涉及额外的写库操作，该写库操作又会因为没有RM本地事务的支持而存在类似问题。。。可想而知，如果不基于RM本地事务，TCC事务框架是无法有效的管理TCC全局事务的。

反之，基于RM本地事务的TCC事务，这种情况则会很容易处理：[B:Try]操作中途执行失败，TCC事务框架将其参与RM本地事务直接rollback即可。后续TCC事务框架决定回滚全局事务时，在知道“[B:Try]操作涉及的RM本地事务已经rollback”的情况下，根本无需执行[B:Cancel]操作。

换句话说，基于RM本地事务实现TCC事务框架时，一个TCC型服务的cancel业务要么执行，要么不执行，不需要考虑部分执行的情况。

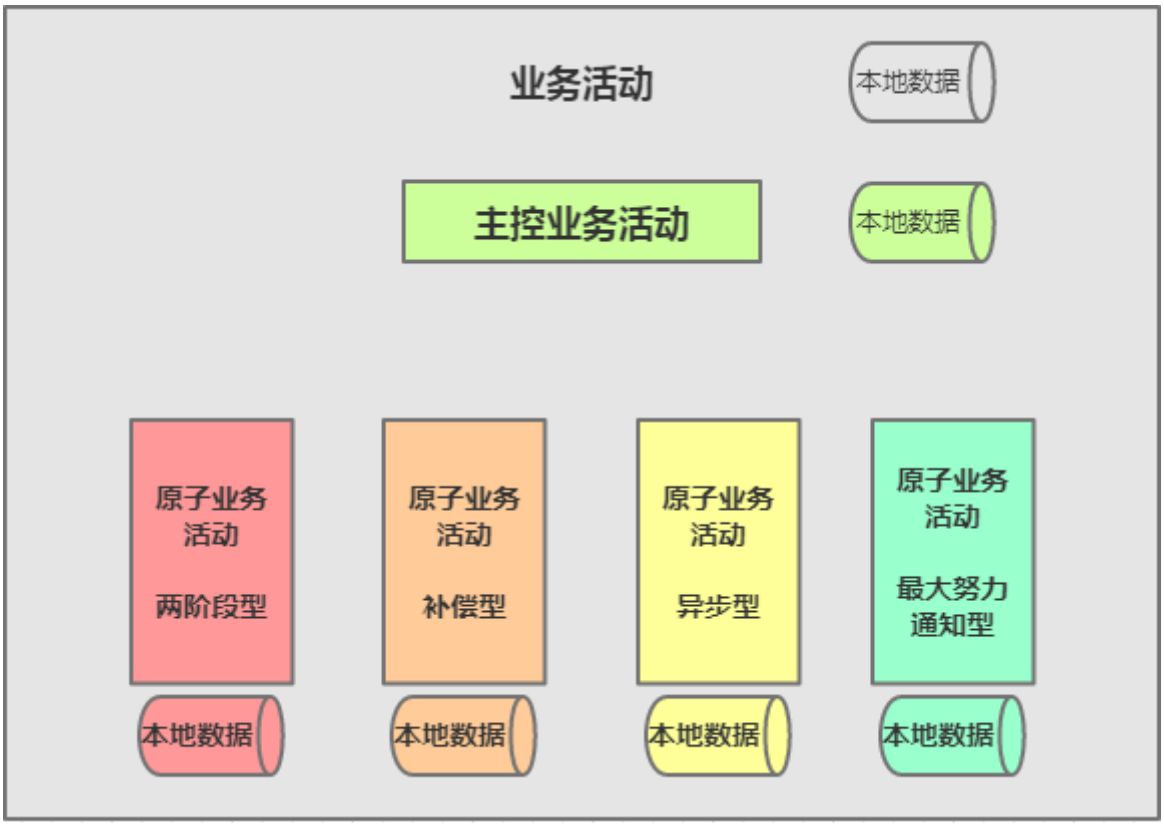
TCC事务框架应该提供Confirm/Cancel服务的幂等性保障

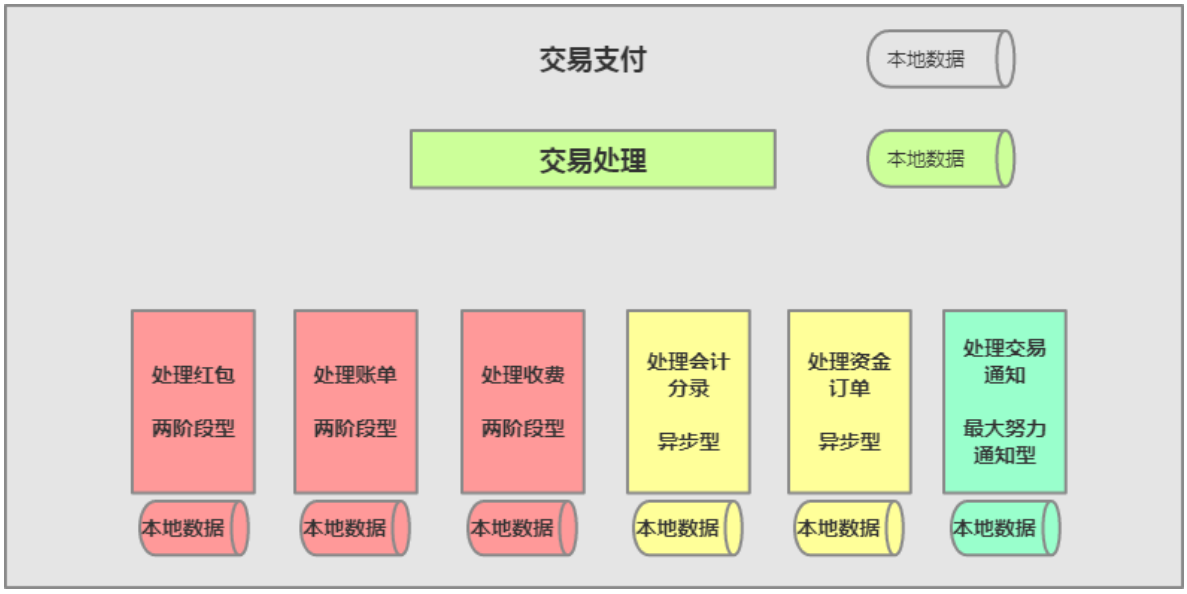
一般认为，服务的幂等性，是指针对同一个服务的多次(n>1)请求和对它的单次(n=1)请求，二者具有相同的副作用。

在TCC事务模型中，Confirm/Cancel业务可能会被重复调用，其原因很多。比如，全局事务在提交/回滚时会调用各TCC服务的Confirm/Cancel业务逻辑。执行这些Confirm/Cancel业务时，可能会出现如网络中断的故障而使得全局事务不能完成。因此，故障恢复机制后续仍然会重新提交/回滚这些未完成的全局事务，这样就会再次调用参与该全局事务的各TCC服务的Confirm/Cancel业务逻辑。

既然Confirm/Cancel业务可能会被多次调用，就需要保障其幂等性。 那么，应该由TCC事务框架来提供幂等性保障？还是应该由业务系统自行来保障幂等性呢？ 个人认为，应该是由TCC事务框架来提供幂等性保障。如果仅仅只是极个别服务存在这个问题的话，那么由业务系统来负责也是可以的；然而，这是一类公共问题，毫无疑问，所有TCC服务的Confirm/Cancel业务存在幂等性问题。TCC服务的公共问题应该由TCC事务框架来解决；而且，考虑一下由业务系统来负责幂等性需要考虑的问题，就会发现，这无疑增大了业务系统的复杂度。

2.4 柔性事务





2.4.1 柔性事务中的服务模式

> 服务模式是柔性事务流程中的特殊操作实现（实现上对应业务服务要提供相应模式的功能接口），还不算是某一种柔性事务解决方案。

- 可查询操作
  - 服务操作的可标识性
    - 服务操作具有全局唯一标识(可以使用业务单据号/使用系统分配的操作流水号使用操作资源的组合组合标识)
    - 操作有唯一的、确定的时间

    - 单笔查询
        - 使用全局唯一的服务操作标识，查询操作执行结果
        - 注意状态判断，小心“处理中”的状态
    - 批量查询
        - 使用时间区段与(或)一组服务操作的标识，查询一批操作执行结果
- 幂等操作
    - 幂等性:f(f(x)) = f(x)
    - 幂等操作:重复调用多次产生的业务结果与调用一次产生的业务结果相同
    - 实现方式一:通过业务操作本身实现幂等性
    - 实现方式二:系统缓存所有请求与处理结果,检测到重复请求之后，自动返回之前的处理结果
- TCC操作
    - Try: 尝试执行业务
        - 完成所有业务检查(一致性)
        - 预留必须业务资源(准隔离性)
    - Confirm:确认执行业务
        - 真正执行业务
        - 不作任何业务检查
        - 只使用Try阶段预留的业务资源
        - Confirm操作要满足幂等性
    - Cancel: 取消执行业务
        - 释放Try阶段预留的业务资源
        - Cancel操作要满足幂等性
    - 与2PC协议比较
        - 位于业务服务层而非资源层
        - 没有单独的准备(Prepare)阶段，Try操作兼备资源操作与准备能力
        - Try操作可以灵活选择业务资源的锁定粒度(以业务定粒度)
        - 较高开发成本

> 很多人把两阶段型操作等同于两阶段提交协议2PC操作。其实TCC操作也属于两阶段型操作。

- 可补偿操作
    - do: 真正执行业务
        - 完成业务处理
        - 业务执行结果外部可见
    - compensate:业务补偿
        - 抵销(或部分抵销)正向业务操作的业务结果
        - 补偿操作满足幂等性
    - 约束
        - 补偿在业务上可行
        - 由于业务执行结果未隔离、或者补偿不完整带来的风险与成本可控

> TCC操作中的Confirm操作和Cancel操作，其实也可以看作是补偿操作

简而言之，TCC是应用层的2PC(2 Phase Commit, 两阶段提交)，如果你将应用看做资源管理器的话。

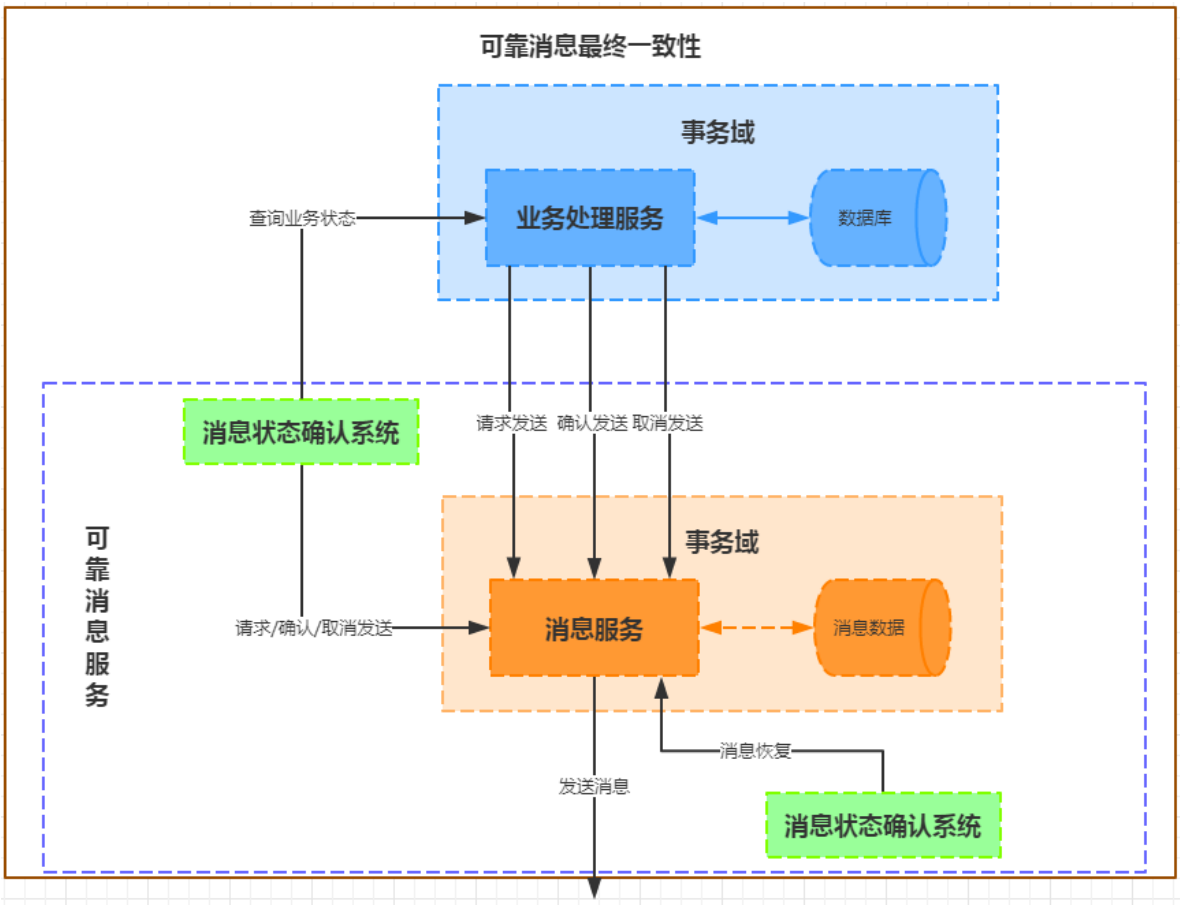
详细来说，TCC每项操作需要做的事情如下：Try：尝试执行业务, 完成所有业务检查(一致性)预留必须业务资源(准隔离性); Confirm：确认执行业务, 真正执行业务不做任何业务检查只使用Try阶段预留的业务资源; Cancel：取消执行业务释放Try阶段预留的业务资源.

TCC事务的优点如下：解决了跨应用业务操作的原子性问题，在诸如组合支付、账务拆分场景非常实用。TCC实际上把数据库层的二阶段提交上提到了应用层来实现，对于数据库来说是一阶段提交，规避了数据库层的2PC性能低下问题。 TCC事务的缺点，主要就一个:TCC的Try、Confirm和Cancel操作功能需业务提供，开发成本高。

账务拆分的业务场景如下，分别位于三个不同分库的帐户A、B、C，A和B一起向C转帐共80元：

Try：尝试执行业务。完成所有业务检查(一致性)：检查A、B、C的帐户状态是否正常，帐户A的余额是否不少于30元，帐户B的余额是否不少于50元。预留必须业务资源(准隔离性)：帐户A的冻结金额增加30元，帐户B的冻结金额增加50元，这样就保证不会出现其他并发进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。 Confirm：确认执行业务。真正执行业务：如果Try阶段帐户A、B、C状态正常，且帐户A、B余额够用，则执行帐户A给账户C转账30元、帐户B给账户C转账50元的转帐操作。不做任何业务检查：这时已经不需要做业务检查，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，则需要对帐户A做Cancel操作，将帐户A被冻结的金额解冻掉。 小结：到底要不要使用TCC 到底要不要使用TCC事务，取决于以下几点：是否真正有保证跨应用业务操作的原子性需求。研发上能否投入资源开发相对应的TCC接口。

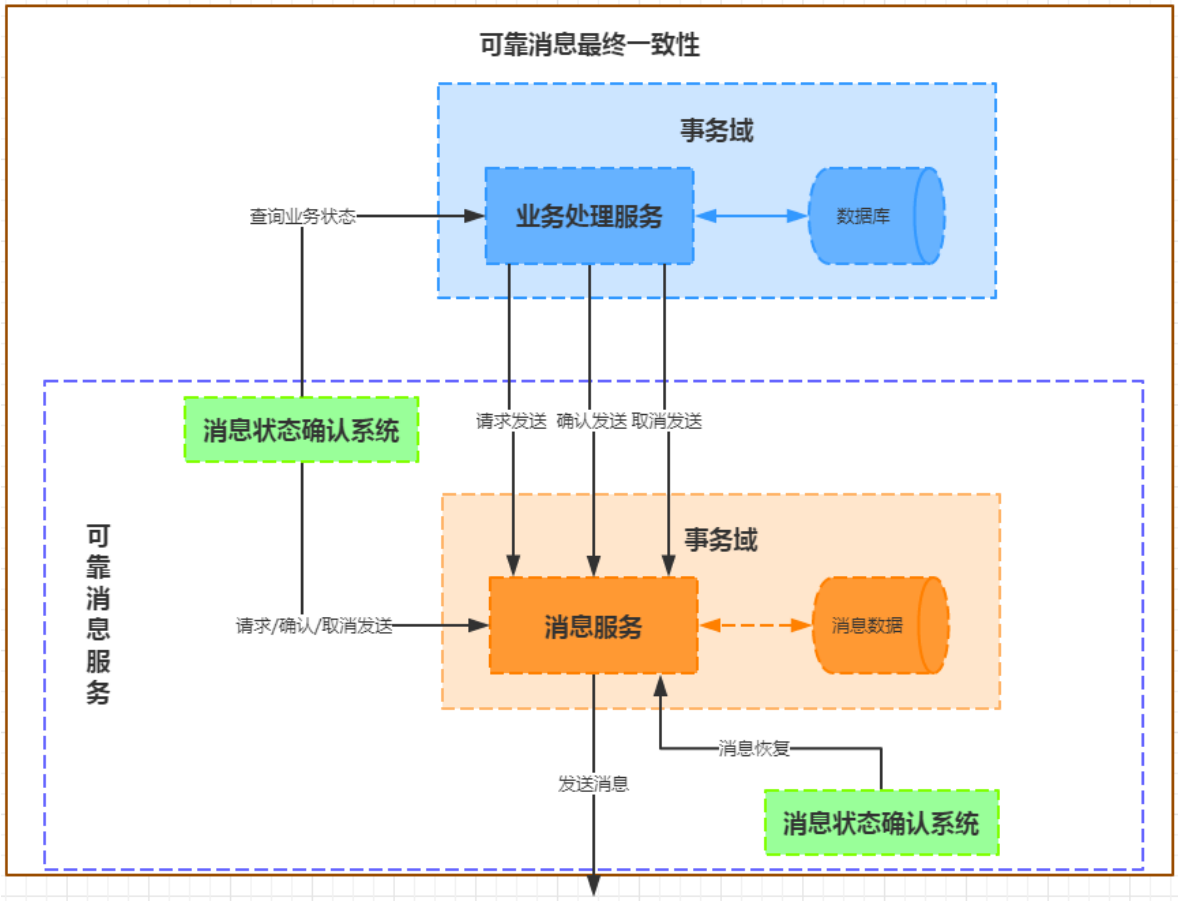
2.4.2 柔性事务解决方案:可靠消息最终一致性(/异步确保型)



- 实现
  - 业务处理服务在业务事务提交前，向实时消息服务请求发送消息，实时消息服务只记录消息数据，而不真正发送。业务处理服务在业务事务提交后，向实时消息服务确认发送。只有在得到确认发送指令后，实时消息服务才真正发送
  - 业务处理服务在业务事务回滚后，向实时消息服务取消发送。消息状态确认系统定期找到未确认发送或回滚发送的消息，向业务处理服务询问消息状态，业务处理服务根据消息ID或消息内容确定该消息是否有效
- 约束

    - 被动方的处理结果不影响主动方的处理结果， 被动方的消息处理操作是幂等操作
- 成本
    - 可靠消息系统建设成本
    - 一次消息发送需要两次请求，业务处理服务需实现消息状态回查接口
- 优点、适用范围
    - 消息数据独立存储、独立伸缩，降低业务系统与消息系统间的耦合
    - 对最终一致性时间敏感度较高，降低业务被动方实现成本
- 用到的服务模式:可查询操作、幂等操作
- 方案特点
    - 兼容所有实现JMS标准的MQ中间件
    - 确保业务数据可靠的前提下，实现业务数据的最终一致（理想状态下基本是准实时一致）

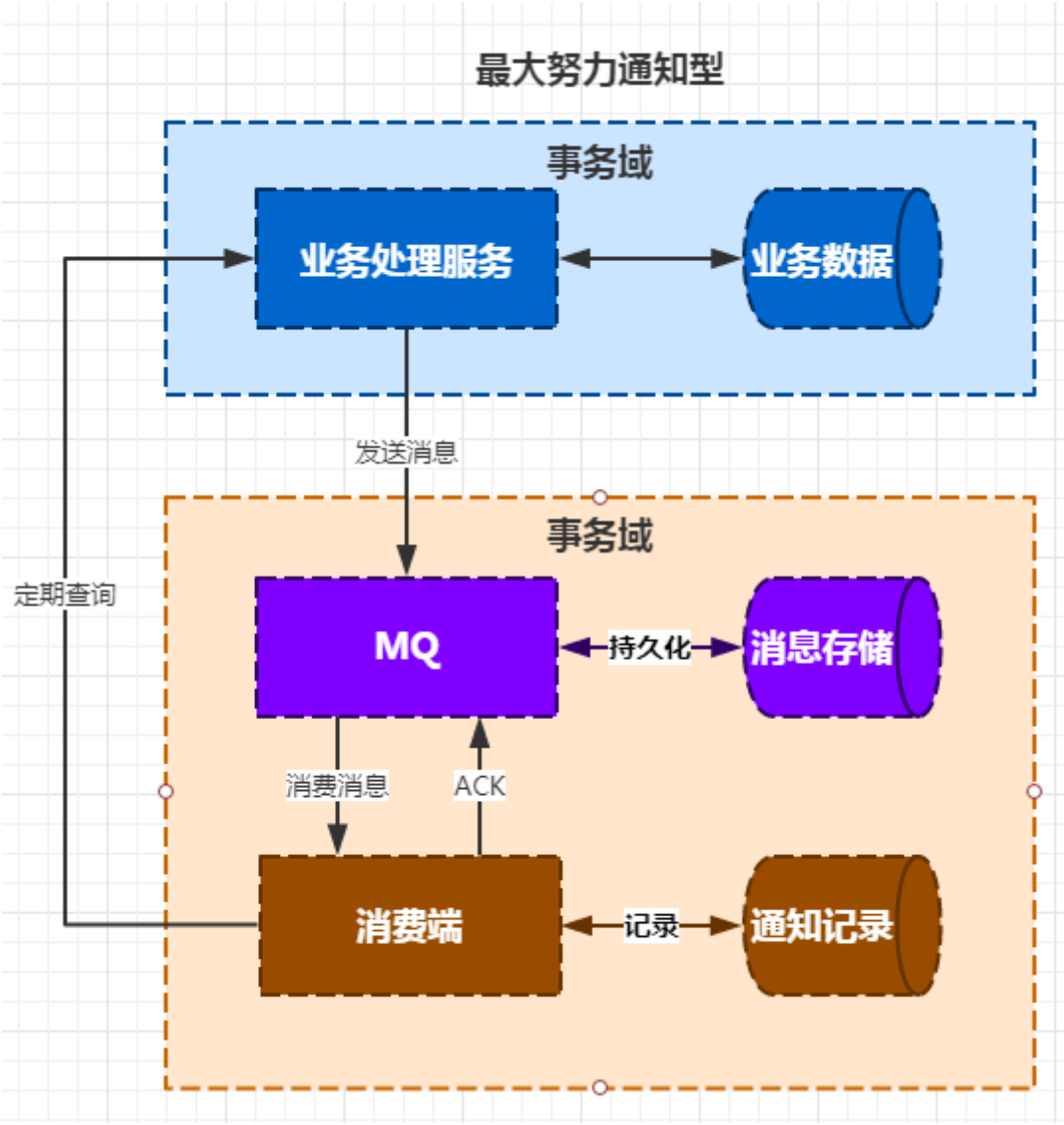
2.4.3 柔性事务解决方案:TCC(两阶段型/补偿型)



- 实现
    - 一个完整的业务活动由一个主业务服务与若干从业务服务组成
    - 主业务服务负责发起并完成整个业务活从业务服务提供TCC型业务操作
    - 业务活动管理器控制业务活动的一致性，它登记业务活动中的操作， 并在业务活动提交时确认所有的TCC型操作的confirm操作，在业务活动取消时调用所有TCC型操作的cancel操作
- 成本
    - 实现TCC操作的成本
    - 业务活动结束时confirm或cancel操作的执行成本
    - 业务活动日志成本
- 适用范围
    - 强隔离性、严格一致性要求的业务活动
    - 适用于执行时间较短的业务（比如处理账户、收费等业务）
- 用到的服务模式:TCC操作、幂等操作、可补偿操作、 可查询操作

- 方案特点:
  - 不与具体的服务框架耦合（在RPC架构中通用）
  - 位于业务服务层，而非资源层
  - 可以灵活选择业务资源的锁定粒度
  - TCC里对每个服务资源操作的是本地事务，数据被lock的时间短，可扩展性好（可以说是为独立部署的SOA服务而设计的）

2.4.4 柔性事务解决方案:最大努力通知型(定期校对)



- 实现
  - 业务活动的主动方，在完成业务处理之后，向业务活动的被动方发送消息，允许消息丢失。
  - 业务活动的被动方根据定时策略，向业务活动主动方查询，恢复丢失的业务消息。
- 约束:被动方的处理结果不影响主动方的处理结果
- 成本:业务查询与校对系统的建设成本
- 适用范围
  - 对业务最终一致性的时间敏感度低
  - 跨企业的业务活动
- 用到的服务模式:可查询操作
- 方案特点
  - 业务活动的主动方在完成业务处理后，向业务活动被动方发送通知消息（允许消息丢失）

    - 主动方可以设置时间阶梯型通知规则，在通知失败后按规则重复通知，直到通知N次后不再通知
    - 主动方提供校对查询接口给被动方按需校对查询，用于恢复丢失的业务消息
- 行业应用案例:银行通知、商户通知等

3. 可靠消息最终一致性方案

3.1 什么是可靠消息最终一致性事务

可靠消息最终一致性方案是指当事务发起执行完成本地事务后并发出一条消息，事务参与方（消息消费者）一定能够接收消息并处理事务成功，此方案强调的是只要消息发给事务参与方最终事务要达到一致。 此方案是利用消息中间件完成，如下图：

事务发起方（消息生产方）将消息发给消息中间件，事务参与方从消息中间件接收消息，事务发起方和消息中间件之间，事务参与方（消息消费方）和消息中间件之间都是通过网络通信，由于网络通信的不确定性导致分布式事务问题。

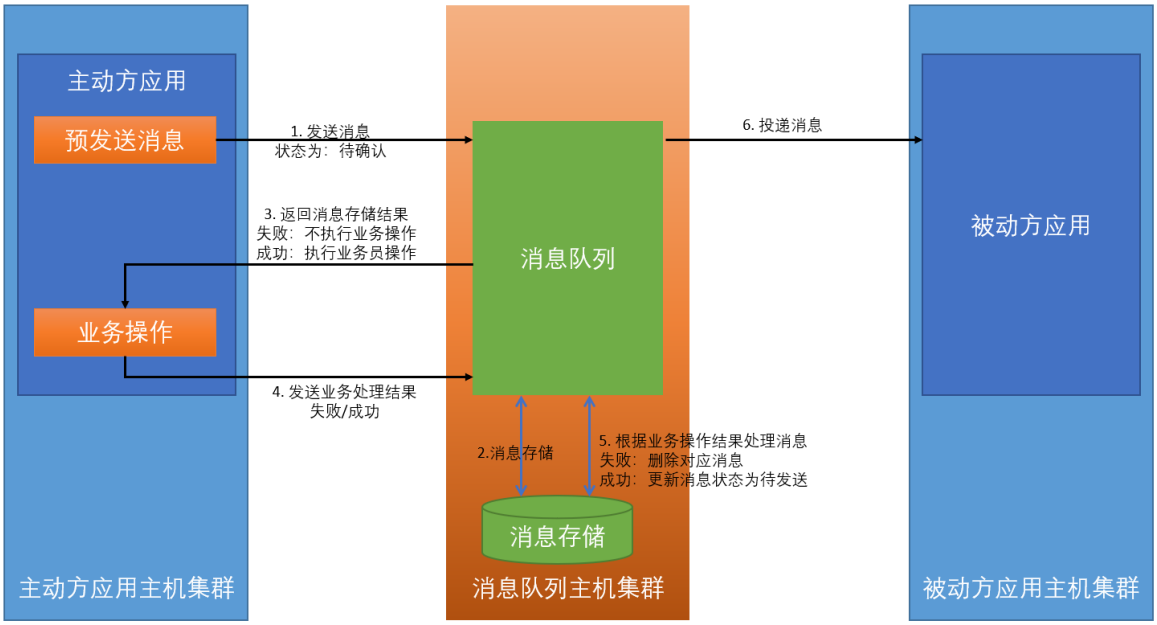
3.2 消息发送的一致性

指产生消息的业务动作与消息发送的一致。（也就是说，如果业务操作成功，那么由这个业务操作所产生的消息一定要成功投递出去，否则就丢消息）

3.2.1 如何保障消息发送一致性

- 处理方式1
    - 如果业务操作成功，执行消息发送前应用故障，消息发不出去，导致消息丢失（订单系统与会计系统的数据不一致）；
    - 如果业务操作成功，应用正常，但消息系统故障或网络故障，也会导致消息发不出去（订单系统与会计系统的数据不一致）；
- 处理方式2
    - 这种情况下，更不可控，消息发出去了，但业务可能会失败（订单系统与会计系统的数据不一致）

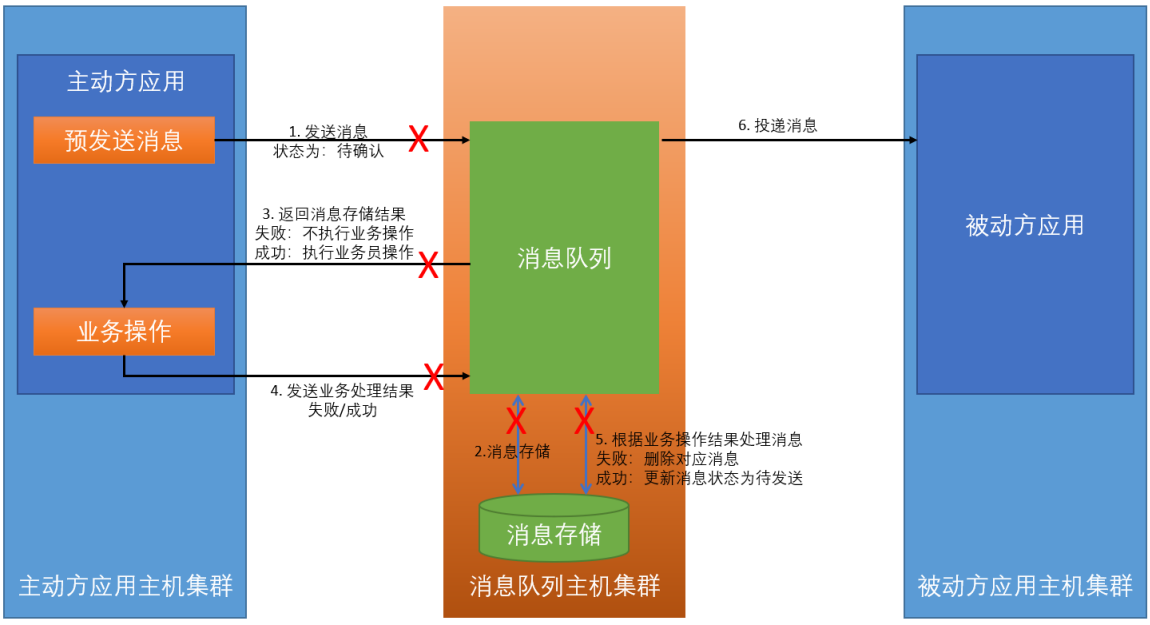
3.2.2 正向流程



1. 主动方应用先把消息发给消息中间件，消息状态标记为“待确认”；
2. 消息中间件收到消息后，把消息持久化到消息存储中，但并不向被动方应用投递消息；
3. 消息中间件返回消息持久化结果（成功/失败），主动方应用根据返回结果进行判断如何进行业务操作处理：
    - 失败：放弃业务操作处理，结束（必要时向上层返回失败结果）；
    - 成功：执行业务操作处理；
4. 业务操作完成后，把业务操作结果（成功/失败）发送给消息中间件；
5. 消息中间件收到业务操作结果后，根据业务结果进行处理；
    - 失败：删除消息存储中的消息，结束；
    - 成功：更新消息存储中的消息状态为“待发送（可发送）”，紧接着执行消息投递；
6. 前面的正向流程都成功后，向被动方应用投递消息

3.2.3 异常流程

- 异常点分析(任何一个环节都可能会出问题)



异常状况	可能的状态	一致性
预发送消息失败	消息未进存储，业务操作未执行（可能的原因：主动方应用、网络、消息中间件、消息存储）	一致
预发送消息后，主动方应用没有收到返回消息存储结果	(1)消息未进存储，业务操作未执行	一致
预发送消息后，主动方应用没有收到返回消息存储结果	(2)消息已进存储（待确认），业务操作未执行	不一致
收到消息存储成功的返回结果，但未执行业务操作就失败	消息已进存储（待确认），业务操作未执行	不一致

- 从消息中间件的角度来分析

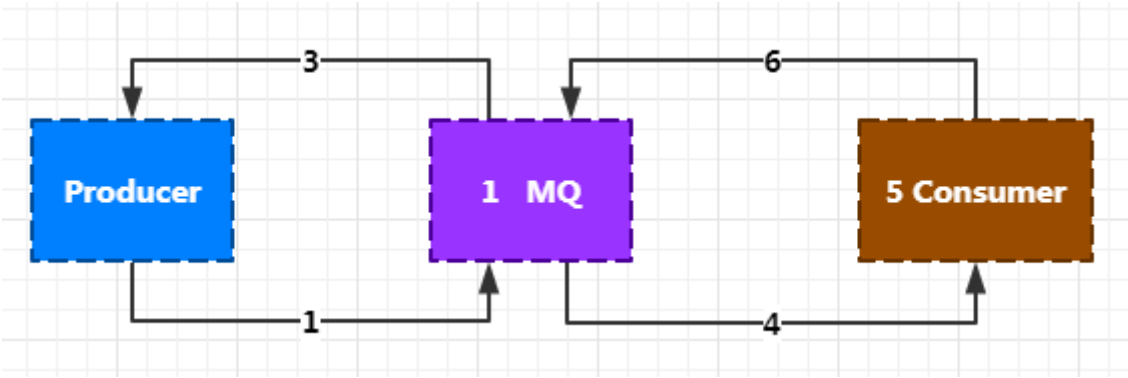
异常状况	可能的状态	一致性
消息中间件没有收到主动方应用的业务操作处理结果	(1)消息已进存储（待确认），业务操作未执行（或业务操作出错回滚了）	不一致
消息中间件没有收到主动方应用的业务操作处理结果	(2)消息已进存储（待确认），业务操作成功	不一致
消息中间件收到业务操作结果（成功/失败），但处理消息存储中的消息状态失败	(1)消息已进存储（待确认），业务操作未执行（或业务操作出错回滚了）	不一致
消息中间件收到业务操作结果（成功/失败），但处理消息存储中的消息状态失败	(2)消息已进存储（待确认），业务操作成功	不一致

- 总结

异常状况	一致性	异常处理方法
消息未进存储，业务操作未执行	一致	无需处理
消息已进存储（状态待确认），业务操作未执行	不一致	确认业务操作结果，处理消息（删除消息）
消息已进存储（状态待确认），业务操作成功,但未通知发送	不一致	确认业务操作结果，处理消息（更新消息状态，执行消息投递）

- 解决方法: 消息中间件进行对主动业务方进行一次查询确认消息的状态。

3.3 常规MQ处理流程



1. Producer生成消息并发送给MQ（同步、异步）；
2. MQ接收消息并将消息数据持久化到消息存储（持久化操作为可选配置）；
3. MQ向Producer返回消息的接收结果（返回值、异常）；
4. Consumer监听并消费MQ中的消息；
5. Consumer获取到消息后执行业务处理；
6. Consumer对已成功消费的消息向MQ进行ACK确认（确认后的消息将从MQ中删除）。

- 队列消息模型的特点：
    1. 消息生产者将消息发送到Queue中，然后消息消费者监听Queue并接收消息；
    2. 消息被确认消费以后，就会从Queue中删除，所以消息消费者不会消费到已经被消费的消息；
    3. Queue支持存在多个消费者，但是对某一个消息而言，只会有一个消费者成功消费。
- 常规MQ队列消息的处理流程无法实现消息发送一致性；
- 投递消息的流程其实就是消息的消费流程，可细化。
- 解决方案如下

> 常规MQ队列消息的处理流程无法实现消息发送一致性，因此直接使用现成的MQ中间件产品无法实现可靠消息最终一致性的分布式事务解决方案。

3.4 消息幂等性

3.4.1 消息重复发送的原因

1. 被动方应用接收到消息，业务处理完成后应用出问题，消息中间件不知道消息处理结果，会重新投递消息。
2. 被动方应用接收到消息，业务处理完成后网络出问题，消息中间件收不到消息处理结果，会重新投递消息。
3. 被动方应用接收到消息，业务处理时间过长，消息中间件因消息超时未确认，会再次投递消息。
4. 被动方应用接收到消息，业务处理完成，消息中间件问题导致收不到消息处理结果，消息会重新投递。
5. 被动方应用接收到消息，业务处理完成，消息中间件收到了消息处理结果，但由于消息存储故障导致消息没能成功确认，消息会再次投递。

3.4.2 业务接口的幂等性设计

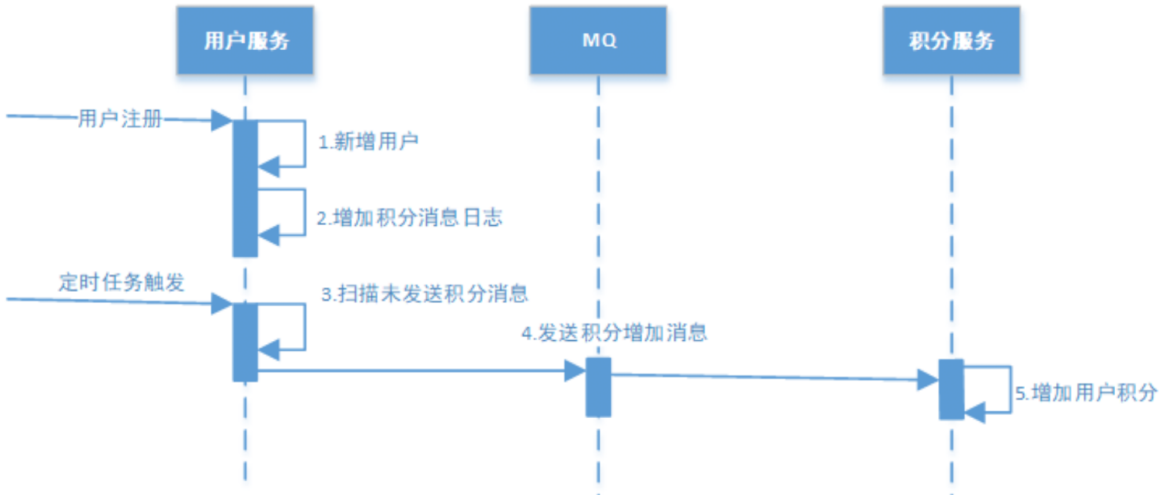
> 约束：被动方应用对于消息的业务处理要实现幂等

- 对于存在同一请求数据会发生重复调用的业务接口，接口的业务逻辑要实现幂等性设计。
- 在实际的业务应用场景中，业务接口的幂等性设计，常结合可查询操作一起使用。
    - 支付订单创建：商户编号 + 商户订单号 + 订单状态
    - 订单更新处理：平台订单号 + 订单状态
    - 会计系统记账：系统来源 + 请求号

3.5 方案一: 本地消息服务的设计

本地消息表这个方案最初是eBay提出的，此方案的核心是通过本地事务保证数据业务操作和消息的一致性，然后通过定时任务将消息发送至消息中间件，待确认消息发送给消费方成功再将消息删除。

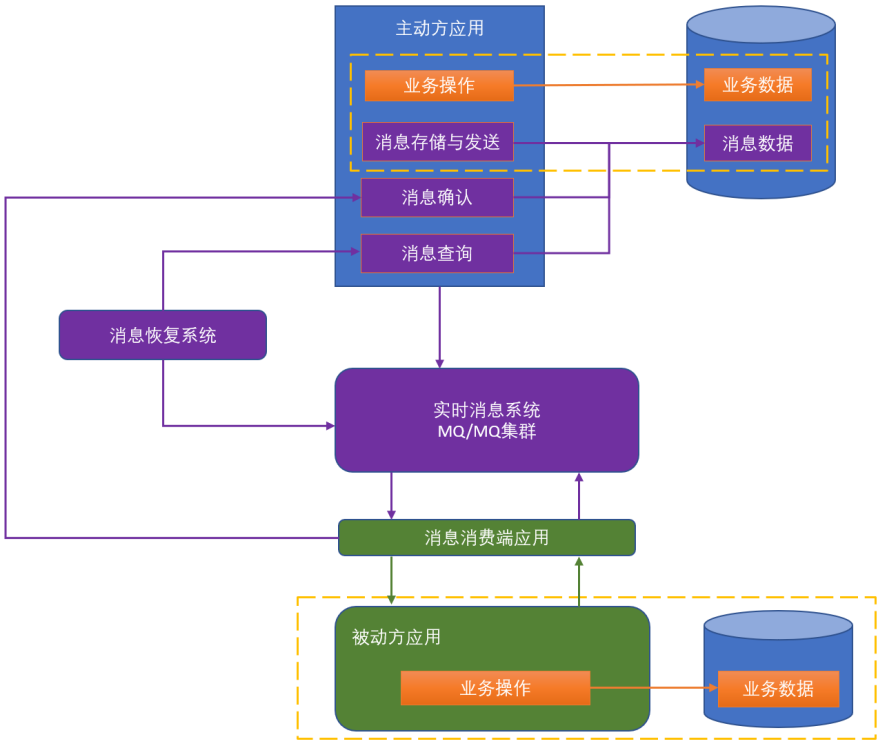
下面以注册送积分为例来说明： 下例共有两个微服务交互，用户服务和积分服务，用户服务负责添加用户，积分服务负责增加积分。



3.5.1 面临问题

1. 现成MQ中间件不支持消息发送的一致性
2. 直接改造MQ中间件难度很大
3. 有什么变通的实现方式?

3.5.2 主要流程



1. 消息存储与业务存储在同一个本地事务中进行,消息存储后设置为待确认状态,并异步将消息发送(注意需要异步发送消息,不要影响主流程).
2. 通过一定策略不断将待确认的消息重新发送.
3. 业务方回**收到消息,成功处理业务,并持久化完成后**调主动方接口,通知主动方此消息已经处理完成,主动方将数据库中消息状态改为已发送.
4. 实现一个消息管理系统,手动处理多次重发失败已死亡的消息.

3.5.3 优势

1. 消息实时性较高
2. 从应用设计开发的角度实现了消息数据的可靠性,消息数据的可靠性不依赖于MQ中间件,弱化了对MQ中间件的依赖
3. 方案轻量容易实现

3.5.4 劣势

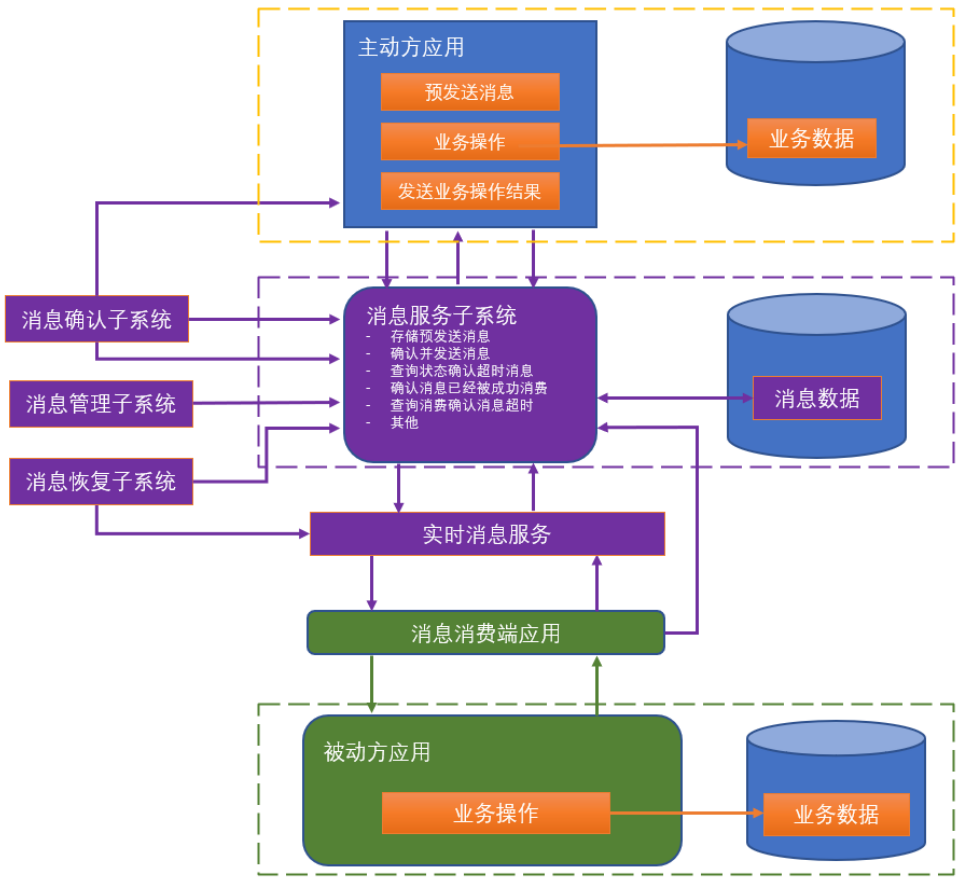
1. 业务绑定,耦合性强,不通用(如果不想直接被动方应用调用主动方应用接口,也可以使用另外一条队列来通知主动方应用)
2. 消息数据与业务数据同库,占用业务系统资源
3. 业务系统在使用关系型数据库的情况下,消息服务性能会受到关系型数据库并发性能的局限

3.6 方案二: 独立消息服务

3.6.1 面临问题

1. 现成MQ中间件不支持消息发送的一致性
2. 直接改造MQ中间件难度很大
3. 有什么变通的实现方式?

3.6.2 主要流程



1. 存储预发送消息(主动方业务执行之前进行,预发送的消息存储后状态为待确认)
2. 确认并发送消息(主动方业务完成之后,主动方或消息状态确认系统通过此接口将消息变为取消或发送中)

3. 查询状态确认超时的消息(消息状态确认系统使用)
4. 确认消息已被成功消费(被动方业务执行完成之后调用,消息队列的ACK可以在业务处理之前返回)
5. 查询消费确认超时的信息

3.6.3 优势

1. 消息服务独立部署,独立维护,独立伸缩
2. 消息存储可以按需选择不同的数据库来集成实现
3. 消息服务可以被相同的使用场景共用,降低消息重复建设消息服务的成本
4. 从应用设计开发的角度实现了消息数据的可靠性,消息数据的可靠性不依赖于MQ中间件,弱化了MQ中间件特性的依赖
5. 降低了业务系统与系统间的耦合,有利于系统的扩展维护

3.6.4 劣势

1. 一次消息需要发送两次请求
2. 主动方应用系统需要实现业务操作状态校验查询接口

3.6.5.业务系统实现

- 预存储消息接口:创建消息,将消息状态初始化为待确认,持久化
- 确认发送消息接口:将消息状态更改为发送中,将消息发送到MQ,注意这里不将消息状态改为已发送
- 存储并发送消息接口:直接存储消息并且直接发送(比如支付网关通过消息服务通知其他服务)
- 直接发送消息接口:透传,消息服务不持久化消息,相当于直接调用MQ
- 根据id重发消息接口:用于消息恢复子系统或消息管理子系统,通过此接口重新发送接口
- 将消息标记为死亡接口:用于将消息标记为死亡,不再重发
- 花样查询消息接口:花样查询各种消息的接口
- 删除消息接口:业务操作成功和主动方业务失败后用于删除消息
- 重发某个队列中的全部死亡消息:防止出现被动方应用宕机后消息积压均重发多次后进入死亡状态的结果

3.6.7 消息管理子系统

- 主要用来用于手动管理死亡消息,重发等

3.6.8 消息状态确认子系统

3.6.9 消息恢复子系统

- 主动方调用方业务执行成功,消息服务子系统中消息状态已变成发送中.我们必须保证消息被被动方消费

3.6.10 实时消息服务子系统

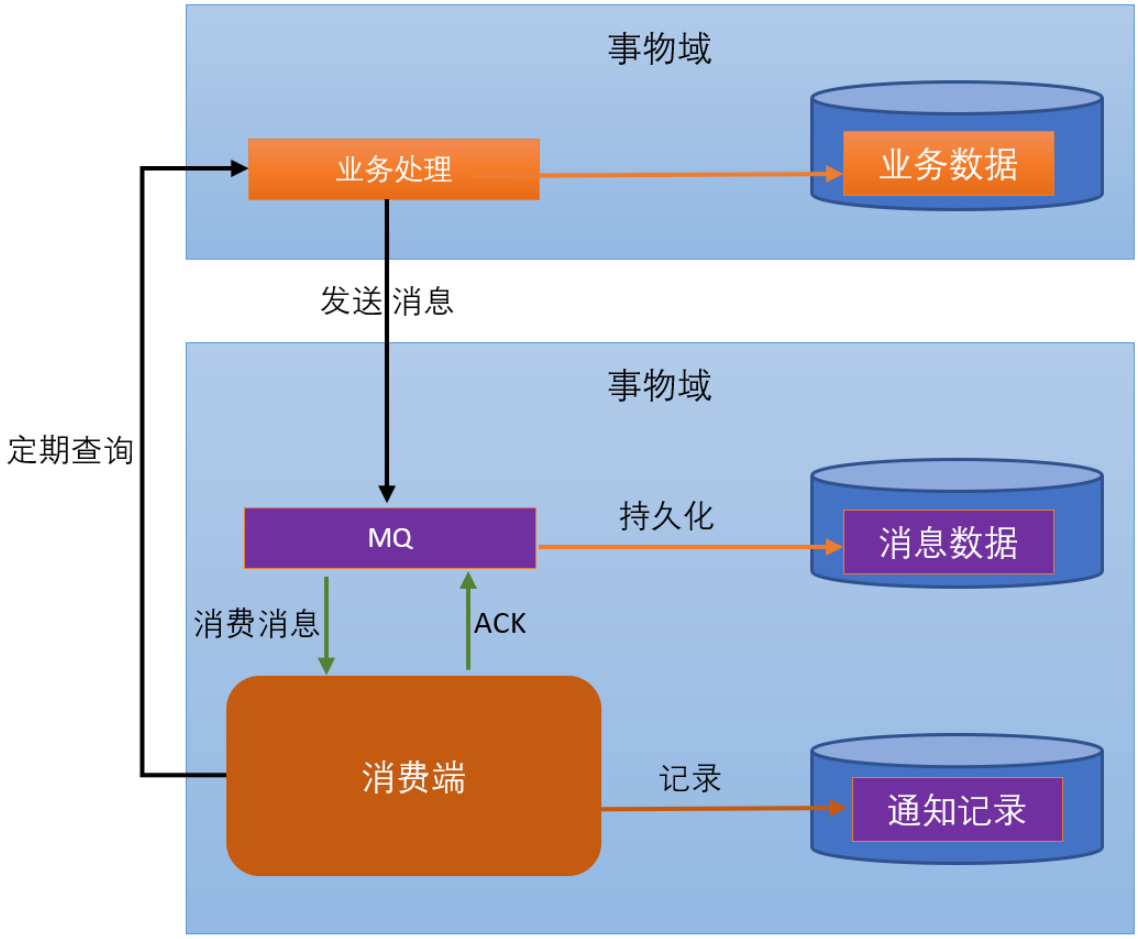
- 使用MQ实现

3.6.11 异步确认(防止可补偿流程错误导致主流程回滚)

- 主动业务方流程
    1. 预发送消息
    2. 执行业务
    3. 确认发送(如果这一步超时会回滚前面的业务,但是消息已被发送到消息服务子系统并并持久化)

4. 最大努力通知方案(定期校对)

4.1 介绍



- 实现
    - 业务活动的主动方,在完成业务活动处理后,向业务活动被动方发送消息,允许消息丢失
    - 业务活动的被动方根据定时策略,向业务活动的主动方查询,恢复丢失的业务消息
- 约束: 被动方的业务处理结果不影响主动方的业务处理
- 成本: 业务查询与校对系统建设成本

- 适用范围
  - 对时间敏感性较低的业务
  - 对账
- 用到的服务模式:可查询操作
- 方案特点
  - 业务活动的主动方在完成业务处理后,向业务活动被动方发送通知消息(允许消息丢失)
  - 主动方可以设置时间阶梯型通知规则,在通知失败后按规则重复通知,直到通知N次后不再通知
  - 主动方提供校对查询接口给被动方,被动方按需校对查询,用于恢复丢失的业务消息
- 行业应用案例
  - 银行通知,商户通知等
  - 对账文件

目标： 发起通知方通过一定的机制最大努力将业务处理结果通知到接收方。

具体包括： 1、有一定的消息重复通知机制。 因为接收通知方可能没有接收到通知，此时要有一定的机制对消息重复通知。

2、消息校对机制。 如果尽最大努力也没有通知到接收方，或者接收方消费消息后要再次消费，此时可由接收方主动向通知方查询消息信息来满足需求。

最大努力通知与可靠消息一致性有什么不同? 1、解决方案思想不同 可靠消息一致性，发起通知方需要保证将消息发出去，并且将消息发到接收通知方，消息的可靠性关键由发起通知方来保证。

最大努力通知，发起通知方尽最大的努力将业务处理结果通知为接收通知方，但是可能消息接收不到，此时需要接收通知方主动调用发起通知方的接口查询业务处理结果，通知的可靠性关键在接收通知方。

2、两者的业务应用场景不同 可靠消息一致性关注的是交易过程的事务一致，以异步的方式完成交易。最大努力通知关注的是交易后的通知事务，即将交易结果可靠的通知出去。

3、技术解决方向不同 可靠消息一致性要解决消息从发出到接收的一致性，即消息发出并且被接收到。最大努力通知无法保证消息从发出到接收的一致性，只提供消息接收的可靠性机制。可靠机制是，最大努力的将消息通知给接收方，当消息无法被接收方接收时，由接收方主动查询消费（业务处理结果）。

4.2 设计实现

- 定时任务队列

1. 发起通知方将通知发给MQ。使用普通消息机制将通知发给MQ。 注意： 如果消息没有发出去可由接收通知方主动请求发起通知方查询业务执行结果。
2. 接收通知方监听MQ。
3. 接收通知方接收消息，业务处理完成回应ack。
4. 接收通知方若没有回应ack则MQ会重复通知。 **MQ会按照间隔1min、5min、10min、30min、1h、2h、5h、10h的方式，逐步拉大通知间隔**（如果MQ采用rocketMq，在broker中可进行配置），直到达到通知要求的时间窗口上限。
5. 接收通知方可通过消息校对接口来校对消息的一致性。

4.3 优化

1. 通知记录/通知日志可视化,手工触发
2. 考虑吧通知服务做的更通用,通知队列区分,不同队列不同规则等

3. 保证通知服务的可用性,必要时建立独立的数据库
4. 要求被动方处理通知接收的业务接口要实现幂等性
5. 内存调优与流量控制(生产速率不匹配,导致大量消息驻留在消费端内存中)

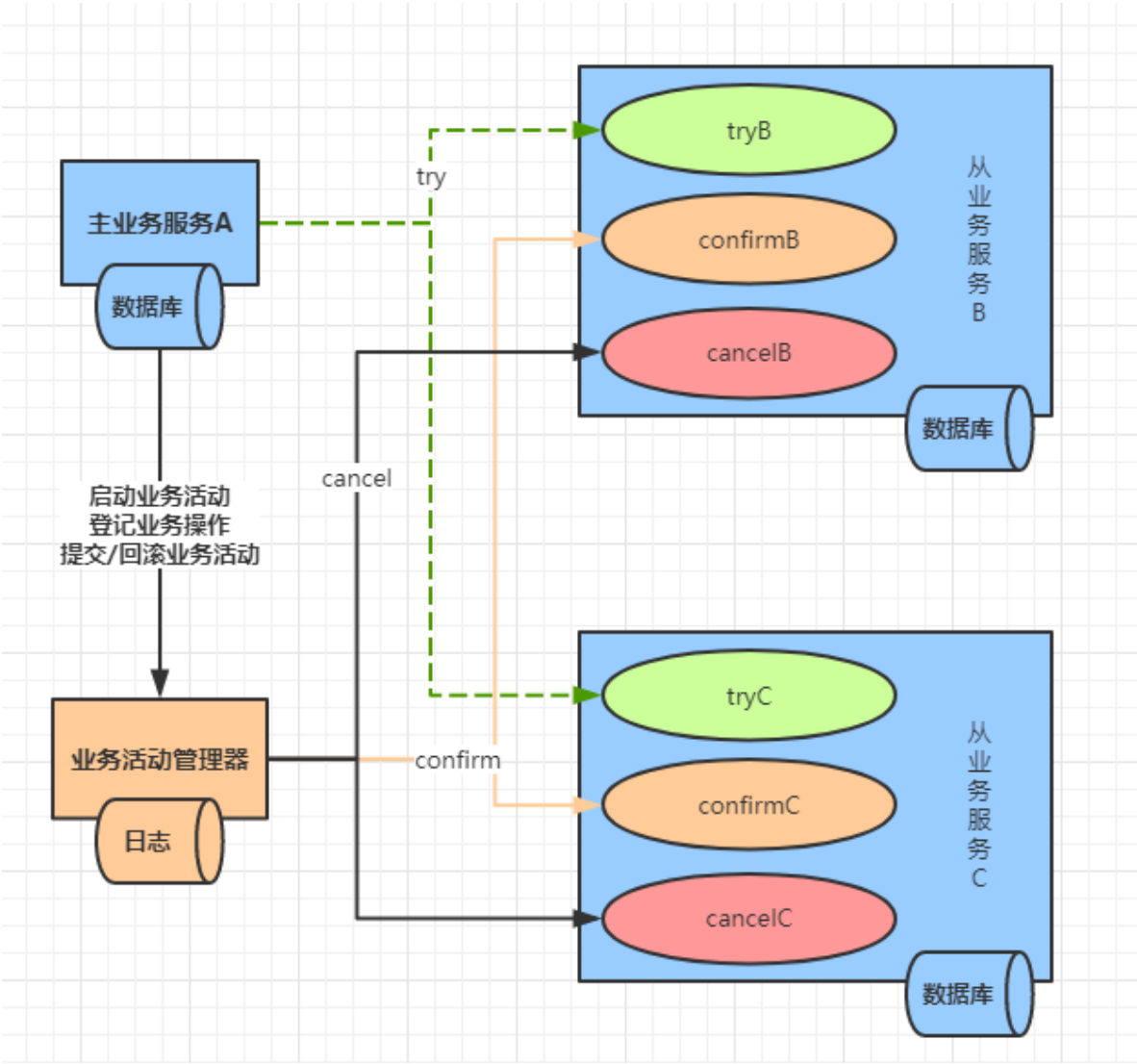
5 TCC 型方案

6.1 介绍

> TCC方案属于两阶段型/补偿型

5.1.1 实现

[



- 一个完整的业务活动由一个主业务服务与若干从业务服务组成
- 主业务服务负责发起并完成整个业务活动
- 从业务服务提供TCC型业务操作

- 业务活动管理器控制业务活动的一致性,它登记业务活动中的操作,并在业务活动提交时确认所有的TCC型操作的confirm操作,在业务活动取消时调用所有TCC型操作的cancel操作.

TCC分为三个阶段：

1. Try阶段是做业务检查（一致性）及资源预留（隔离），此阶段仅是一个初步操作，它和后续的Confirm一起才能真正构成一个完整的业务逻辑。
2. Confirm阶段是做确认提交，Try阶段所有分支事务执行成功后开始执行Confirm。通常情况下，采用TCC则认为Confirm阶段是不会出错的。即： 只要Try成功，Confirm一定成功。若Confirm阶段真的出错了，需引入重试机制或人工处理。
3. Cancel阶段是在业务执行错误需要回滚的状态下执行分支事务的业务取消，预留资源释放。通常情况下，采用TCC则认为Cancel阶段也是一定成功的。若Cancel阶段真的出错了，需引入重试机制或人工处理。
4. TM事务管理器 TM事务管理器可以实现为独立的服务，也可以让全局事务发起方充当TM的角色，TM独立出来是为了成为公用组件，是为了考虑结构和软件复用。 TM在发起全局事务时生成全局事务记录，全局事务ID贯穿整个分布式事务调用链条，用来记录事务上下文，追踪和记录状态，由于Confirm和Cancel失败需进行重试，因此需要实现为幂等性是指同一个操作无论请求多少次，其结果都相同。

5.1.2 成本

- 业务活动结束时confirm或cancel操作的执行成本
- 业务活动日志成本

5.1.3 适用范围

- 强隔离性,严格一致性要求的业务活动
- 适用于执行时间较短的业务,如处理账户,收费等业务

5.1.4 用到的服务模式

- TCC操作
- 幂等操作
- 可补偿操作
- 可查询操作

5.1.5 方案特点

- 不与具体的服务框架耦合(RPC架构通用)
- 位于业务服务层,而非资源层
- 可以灵活选择业务资源的锁定粒度
- TCC里对每个服务资源操作的是本地事务,数据被lock的时间短,可扩展性好(可以说是为独立部署的SOA服务而设计的)

5.1.6 行业应用案例

- 支付宝XTS(蚂蚁金融云的分布式事务服务DTS)

5.2 TCC框架

目前市面上的TCC框架众多比如下面这几种：

框架	Github地址	star 数量
tcc-transaction	https://github.com/changmingxie/tcc-transaction	4.6K
Hmily	https://github.com/Dromara/hmily	2.8K
ByteTCC	https://github.com/liuyangming/ByteTCC	2.3K
EasyTransaction	https://github.com/QNJR-GROUP/EasyTransaction	2K
Hulk	https://github.com/wchswchs/Hulk	33

Seata也支持TCC，但Seata的TCC模式对Spring Cloud并没有提供支持。我们的目标是理解TCC原理以及事务协调运作的过程，因此更倾向于轻量级易于理解的框架。

Hmily是一个高性能分布式事务TCC开源框架。基于Java语言来开发（JDK1.8），支持Dubbo，Spring Cloud等RPC框架进行分布式事务。它目前支持以下特性：

- 支持嵌套事务（Nested transaction support）。
- 采用disruptor框架进行事务日志的异步读写，与RPC框架的性能毫无差别。
- 支持SpringBoot-starter项目启动，使用简单。
- RPC框架支持：dubbo、motan、springcloud。
- 本地事务存储支持：redis、mongodb、zookeeper、file、mysql。
- 事务日志序列化支持：java、hessian、kryo、protostuff。
- 采用Aspect AOP切面思想与Spring无缝集成，天然支持集群。
- RPC事务恢复，超时异常恢复等。

> Hmily利用AOP对参与分布式事务的本地方法与远程方法进行拦截处理，通过多方拦截，事务参与者能透明的调用到另一方的Try、Confirm、Cancel方法；传递事务上下文；并记录事务日志，酌情进行补偿，重试等。
>
> Hmily不需要事务协调服务，但需要提供一个数据库（mysql/mongodb/zookeeper/redis/file）来进行日志存储。
>
> Hmily实现的TCC服务与普通的服务一样，只需要暴露一个接口，也就是它的Try业务。Confirm/Cancel业务逻辑，只是因为全局事务提交/回滚的需要才提供的，因此Confirm/Cancel业务只需要被Hmily TCC事务框架发现即可，不需要被调用它的其他业务服务所感知。
>
> 官网介绍：https://dromara.org/website/zh-cn/docs/hmily/index.html
>
> **TCC需要注意三种异常处理分别是空回滚、幂等、悬挂：**

空回滚： 在没有调用TCC资源Try方法的情况下，调用来二阶段的Cancel方法，Cancel方法需要识别出这是一个空回滚，然后直接返回成功。 出现原因是当一个分支事务所在服务宕机或网络异常，分支事务调用记录为失败，这个时候其实是没有执行Try阶段，当故障恢复后，分布式事务进行回滚则会调用二阶段的Cancel方法，从而形成空回滚。 解决思路是关键就是要识别出这个空回滚。思路很简单就是需要知道一阶段是否执行，如果执行来，那就是正常回滚；如果没执行，那就是空回滚。前面已经说过TM在发起全局事务时生成全局事务记录，全局事务ID贯穿整个分布式事务调用链条。再额外增加一张分支事务记录表，其中有全局事务ID和分支事务ID，第一阶段Try方法里会插入一条记录，表示一阶段执行来。Cancel接口里读取该记录，如果该记录存在，则正常回滚；如果该记录不存在，则是空回滚。

幂等： 通过前面介绍已经了解到，为了保证TCC二阶段提交重试机制不会引发数据不一致，要求TCC的二阶段Try、Confirm和Cancel接口保证幂等，这样不会重复使用或者释放资源。如果幂等控制没有做好，很有可能导致数据不一致等严重问题。 解决思路在上述“分支事务记录”中增加执行状态，每次执行前都查询该状态。

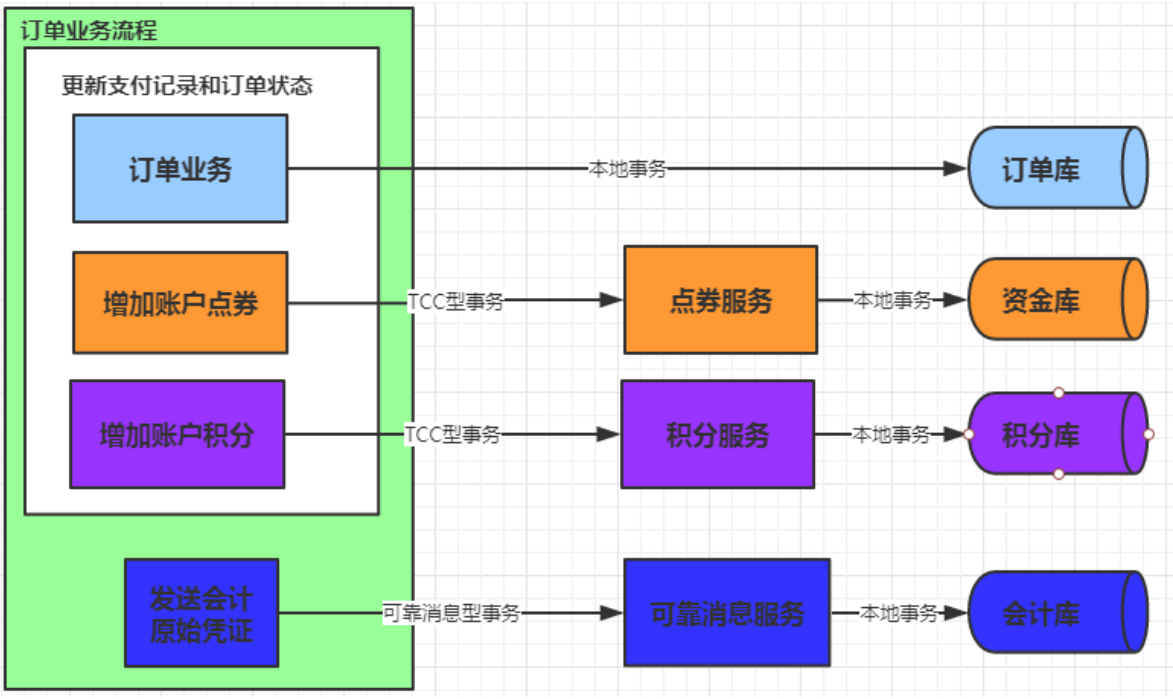
悬挂： 悬挂就是对于一个分布式事务，其二阶段Cancel接口比Try接口先执行。 出现原因是在RPC调用分支事务try时，先注册分支事务，再执行RPC调用，如果此时RPC调用的网络发生拥堵，通常RPC调用是有超时时间的，RPC超时以后，TM就会通知RM回滚该分布式事务，可能回滚完成后，RPC请求才到达参与者真正执行，而一个Try方法预留的业务资源，只有该分布式事务才能使用，该分布式事务第一阶段预留的业务资源就再也没有人能够处理了，对于这种情况，我们就称为悬挂，即业务资源预留后无法继续处理。

解决思路是如果二阶段执行完成，那一阶段就不能再继续执行。在执行一阶段事务时判断在该全局事务下，“分支事务记录”表中是否已经有二阶段事务记录，如果有则不执行Try。

5.3 TCC应用实例

5.3.1 业务流程

> 以订单处理流程为例:账户扣款->使用红包优惠券->订单状态改变为例



> 一个主服务调用从业务服务,被TCC控制的方法都应该具有三种方法:主方法(try方法)/确认方法(confirm方法)/取消方法(cancel方法)

1. 修改支付记录状态
2. 修改订单状态
3. 远程调用点券服务
4. 远程调用积分服务
5. 若远程调用存在异常

5.2.2 使用TCC-TRASACTION示例

```
@Service("pointAccountService")
public class pointAccountServiceImpl implements pointAccountService{

    private static final Logger LOG =
LoggerFactory.getLogger(pointAccountServiceImpl.class);

    @Autowired
    private pointAccountDao pointAccountDao;

    @Autowired
    private pointAccountHistoryDao pointAccountHistoryDao;

    @Override
    public void saveData(pointAccount pointAccount) {
        pointAccountDao.insert(pointAccount);
    }

    @Override
    public void updateData(pointAccount pointAccount) {
        pointAccountDao.update(pointAccount);
    }


    /**
     * 积分账户加款 Trying
     * @param transactionContext
     * @param userNo
     * @param pointAmount
     * @param requestNo
     * @param bankTrxNo
     * @param trxType
     * @param remark
     * @throws BizException
     */
    @Override
    @Transactional(rollbackFor = Exception.class)
    @Compensable(confirmMethod = "confirmCreditToPointAccountTcc",cancelMethod =
"cancelCreditToPointAccountTcc")
    public void creditToPointAccountTcc(TransactionContext transactionContext,
String userNo, Integer pointAmount, String requestNo, String bankTrxNo, String
trxType, String remark) throws BizException {

        LOG.info("===>creditToPointAccountTcc TRYING begin");

        //根据商户编号获取商户积分账户
        pointAccount pointAccount = pointAccountDao.getByUserNo(userNo);
```

```
        if (pointAccount == null){//如果不存在商户积分账户,创建一条新的积分账户
            pointAccount = new pointAccount();
            pointAccount.setBalance(0);
            pointAccount.setUserNo(userNo);
            pointAccount.setStatus(PublicEnum.YES.name());
            pointAccount.setCreateTime(new Date());
            pointAccount.setId(StringUtil.get32UUID());
            pointAccountDao.insert(pointAccount);
        }

        pointAccountHistory pointAccountHistory =
pointAccountHistoryDao.getByRequestNo(requestNo);
        // 幂等判断
        if ( pointAccountHistory == null ){//防止多次提交
            pointAccountHistory = new pointAccountHistory();
            pointAccountHistory.setId(StringUtil.get32UUID());
            pointAccountHistory.setCreateTime(new Date());

 pointAccountHistory.setStatus(PointAccountHistoryStatusEnum.TRYING.name());//消
息不可用
            pointAccountHistory.setAmount(pointAmount);///积分账户变动额
            pointAccountHistory.setBalance(pointAccount.getBalance() +
pointAmount);
            pointAccountHistory.setBankTrxNo(bankTrxNo);//银行流水号
            pointAccountHistory.setRequestNo(requestNo);//请求号

 pointAccountHistory.setFundDirection(PointAccountFundDirectionEnum.ADD.name());
            pointAccountHistory.setTrxType(trxType);
            pointAccountHistory.setRemark(remark);
            pointAccountHistory.setUserNo(userNo);
            pointAccountHistoryDao.insert(pointAccountHistory);
        }else if
(PointAccountHistoryStatusEnum.CANCEL.name().equals(pointAccountHistory.getStatu
s())){
            //如果是取消的,有可能是之前的业务出现异常问题而取消,那么重试阶段,再将状态更新为
TYING状态,而不是重新创建一条
            LOG.info("之前因为业务问题取消后,又重试的{}" ,
pointAccountHistory.getBankTrxNo());

 pointAccountHistory.setStatus(PointAccountHistoryStatusEnum.TRYING.name());
            this.pointAccountHistoryDao.update(pointAccountHistory);
        }
        //添加一条不可用的积分账户流水
        LOG.info("===>creditToPointAccountTcc TRYING end");
    }

    /**
     * 积分账户增加确认
     * @param transactionContext
     * @param userNo
     * @param pointAmount
     * @param requestNo
     * @param bankTrxNo
     * @param trxType
     * @param remark
     * @return
     * @throws BizException
     */
```

```
    @Transactional(rollbackFor = Exception.class)
    public void confirmCreditToPointAccountTcc(TransactionContext
transactionContext, String userNo, Integer pointAmount, String requestNo, String
bankTrxNo, String trxType, String remark) throws BizException {

        LOG.info("===>confirmCreditToPointAccountTcc begin");
        //根据请求号获取账户基本流水
        pointAccountHistory pointAccountHistory =
pointAccountHistoryDao.getByRequestNo(requestNo);
        // 幂等判断
        if ( pointAccountHistory == null  ||
PointAccountHistoryStatusEnum.CONFORM.name().equals(pointAccountHistory.getStatu
s())){//该笔交易流水已处理过,不需再处理
            return;
        }


 pointAccountHistory.setStatus(PointAccountHistoryStatusEnum.CONFORM.name());
        pointAccountHistoryDao.update(pointAccountHistory);

        pointAccount pointAccount = pointAccountDao.getByUserNo(userNo);//获取用
户积分账户
        pointAccount.setBalance(pointAccount.getBalance() + pointAmount);//增加账
户余额
        pointAccountDao.update(pointAccount);

        LOG.info("===>confirmCreditToPointAccountTcc end");

    }
    /**
     *积分账户增加回滚
     * @param transactionContext
     * @param userNo
     * @param pointAmount
     * @param requestNo
     * @param bankTrxNo
     * @param trxType
     * @param remark
     * @throws BizException
     */
    @Transactional(rollbackFor = Exception.class)
    public void cancelCreditToPointAccountTcc(TransactionContext
transactionContext, String userNo, Integer pointAmount, String requestNo, String
bankTrxNo, String trxType, String remark) throws BizException {
        LOG.info("===>cancelCreditToPointAccountTcc begin");
        pointAccountHistory pointAccountHistory =
pointAccountHistoryDao.getByRequestNo(requestNo);
        // 幂等判断
        if ( pointAccountHistory == null  ||
!PointAccountHistoryStatusEnum.TRYING.name().equals(pointAccountHistory.getStatu
s())){//该笔交易流水已处理过,不需再处理
            return;
        }


 pointAccountHistory.setStatus(PointAccountHistoryStatusEnum.CANCEL.name());
        pointAccountHistoryDao.update(pointAccountHistory);
```

```
        LOG.info("===>cancelCreditToPointAccountTcc end");
    }

    /**
     * 积分账户加款 Trying
     * @param userNo
     * @param pointAmount
     * @param requestNo
     * @param bankTrxNo
     * @param trxType
     * @param remark
     * @throws BizException
     */
    @Override
    @Transactional(rollbackFor = Exception.class)
    public void creditToPointAccount(String userNo, Integer pointAmount, String
requestNo, String bankTrxNo, String trxType, String remark) throws BizException
{

        //根据商户编号获取商户积分账户
        pointAccount pointAccount = pointAccountDao.getByUserNo(userNo);
        if (pointAccount == null){//如果不存在商户积分账户,创建一条新的积分账户
            pointAccount = new pointAccount();
            pointAccount.setBalance(0);
            pointAccount.setUserNo(userNo);
            pointAccount.setStatus(PublicEnum.YES.name());
            pointAccount.setCreateTime(new Date());
            pointAccount.setId(StringUtil.get32UUID());
            pointAccountDao.insert(pointAccount);
        }


    //添加一条积分历史
    pointAccountHistory pointAccountHistory =
pointAccountHistoryDao.getByRequestNo(requestNo);
    if ( pointAccountHistory == null ){//防止多次提交
            pointAccountHistory = new pointAccountHistory();
            pointAccountHistory.setId(StringUtil.get32UUID());
            pointAccountHistory.setCreateTime(new Date());

 pointAccountHistory.setStatus(PointAccountHistoryStatusEnum.CONFORM.name());//
可用
            pointAccountHistory.setAmount(pointAmount);///积分账户变动额
            pointAccountHistory.setBalance(pointAccount.getBalance() +
pointAmount);
            pointAccountHistory.setBankTrxNo(bankTrxNo);//银行流水号
            pointAccountHistory.setRequestNo(requestNo);//请求号

 pointAccountHistory.setFundDirection(PointAccountFundDirectionEnum.ADD.name());
            pointAccountHistory.setTrxType(trxType);
            pointAccountHistory.setRemark(remark);
            pointAccountHistory.setUserNo(userNo);
            pointAccountHistoryDao.insert(pointAccountHistory);
        }

        //增加积分账户
        pointAccount.setBalance(pointAccount.getBalance() + pointAmount);//增加账
户余额
```

```
        pointAccountDao.update(pointAccount);
    }

    @Override
    public pointAccount getDataById(String id) {
        return pointAccountDao.getById(id);
    }

    @Override
    public PageBean listPage(PageParam pageParam, pointAccount pointAccount) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        return pointAccountDao.listPage(pageParam, paramMap);
    }
}
```

6. Saga方案

6.1 SAGA介绍

SAGA来源于1987年普林斯顿大学的Hector Garcia-Molina和Kenneth Salem发表了一篇Paper Sagas，讲述的是如何处理long lived transaction（长活事务）。Saga是一个长活事务可被分解成可以交错运行的子事务集合。其中每个子事务都是一个保持数据库一致性的真实事务。通俗来说，这个长活事务是由多个本地事务所组成， 每个本地事务有相应的执行模块和补偿模块，当saga事务中的任意一个本地事务出错了， 可以通过调用相关事务对应的补偿方法恢复，达到事务的最终一致性。

随着微服务的出现，越来越多的人想解决分布式事务问题，Saga也逐步受到大家的关注，是比较受欢迎的业界分布式事务解决方案之一，目前开源的框架有华为Apache ServiceComb Saga 。

6.2 SAGA组成

每个Saga由一系列sub-transaction Ti 组成

每个Ti 都有对应的补偿动作Ci，补偿动作用于撤销Ti造成的结果

可以看到，和TCC相比，Saga没有“预留”动作，它的Ti就是直接提交到库。

Saga的执行顺序有两种：

T1, T2, T3, ..., Tn

T1, T2, ..., Tj, Cj,..., C2, C1，其中0 < j < n

Saga定义了两种恢复策略：

backward recovery，向后恢复，补偿所有已完成的事务，如果任一子事务失败。即上面提到的第二种执行顺序，其中j是发生错误的sub-transaction，这种做法的效果是撤销掉之前所有成功的sub-transation，使得整个Saga的执行结果撤销。

forward recovery，向前恢复，重试失败的事务，假设每个子事务最终都会成功。适用于必须要成功的场景，执行顺序是类似于这样的：T1, T2, ..., Tj(失败), Tj(重试),..., Tn，其中j是发生错误的sub-transaction。该情况下不需要Ci。

显然，向前恢复没有必要提供补偿事务，如果你的业务中，子事务（最终）总会成功，或补偿事务难以定义或不可能，向前恢复更符合你的需求。

理论上补偿事务永不失败，然而，在分布式世界中，服务器可能会宕机，网络可能会失败，甚至数据中心也可能会停电。在这种情况下我们能做些什么？ 最后的手段是提供回退措施，比如人工干预。

6.3 SAGA的优缺点

优势：

1、丰富的理论基础，有较为成熟的开源框架，且Apache ServiceComb Saga已经进入apache项目孵化，未来也会不断的演进升级，且有比较完善的文档和用户手册

2、作为ServiceComb微服务分布式事务的解决方案，和ServiceComb天然无缝结合，方便平台在集成serviceComb微服务和分布式事务框架的时候减少工作量和阻力

3、基于BASE定理，提供基本可用的服务能力，与tcc相比：

有些业务很简单，套用TCC需要修改原来的业务逻辑，而Saga只需要添加一个补偿动作就行了。

TCC最少通信次数为2n，而Saga为n（n=sub-transaction的数量）。

有些第三方服务没有Try接口，TCC模式实现起来就比较tricky了，而Saga则很简单。

没有预留动作就意味着不必担心资源释放的问题，异常处理起来也更简单（请对比Saga的恢复策略和TCC的异常处理）

缺陷：

1、缺少预留动作，是优势也是缺点，导致补偿动作的实现比较麻烦：Ti就是commit，比如一个业务是发送邮件，在TCC模式下，先保存草稿（Try）再发送（Confirm），撤销的话直接删除草稿（Cancel）就行了。而Saga则就直接发送邮件了（Ti），如果要撤销则得再发送一份邮件说明撤销（Ci），实现起来有一些麻烦。

2、saga不保证ACID,只保持服务的基本可用和数据的最终一致性，事务隔离性差，要保证数据不被脏读需要在业务上进行相应的逻辑处理

6.4 Apache ServiceComb Saga

Apache ServiceComb Saga 是华为开源的微服务应用的数据最终一致性解决方案，已经进入apache项目孵化，是Apache ServiceComb微服务架构中的一员。

1、特性

高可用。支持集群模式。

高可靠。所有的事务事件都持久存储在数据库中。

高性能。事务事件是通过gRPC来上报的，且事务的请求信息是通过Kyro进行序列化和反序列化的。

低侵入。仅需2-3个注解和编写对应的补偿方法即可进行分布式事务。

部署简单。可通过Docker快速部署。

支持前向恢复（重试）及后向恢复（补偿）。

扩展简单。基于Pack架构很容实现多种协调机制。

2、架构

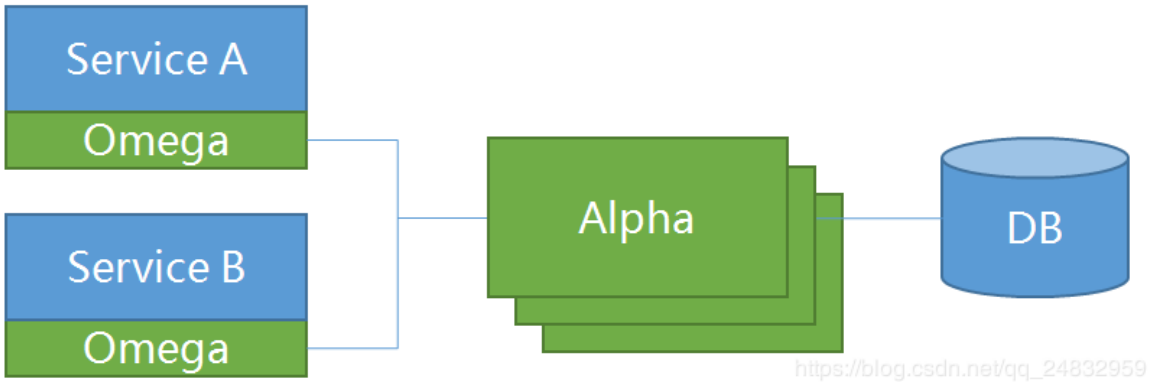
Saga Pack 架构是由alpha和omega组成，其中：

alpha充当协调者的角色，主要负责对事务进行管理和协调。

omega是微服务中内嵌的一个agent，负责对网络请求进行拦截并

向alpha上报事务事件。

下图展示了alpha, omega以及微服务三者的关系：

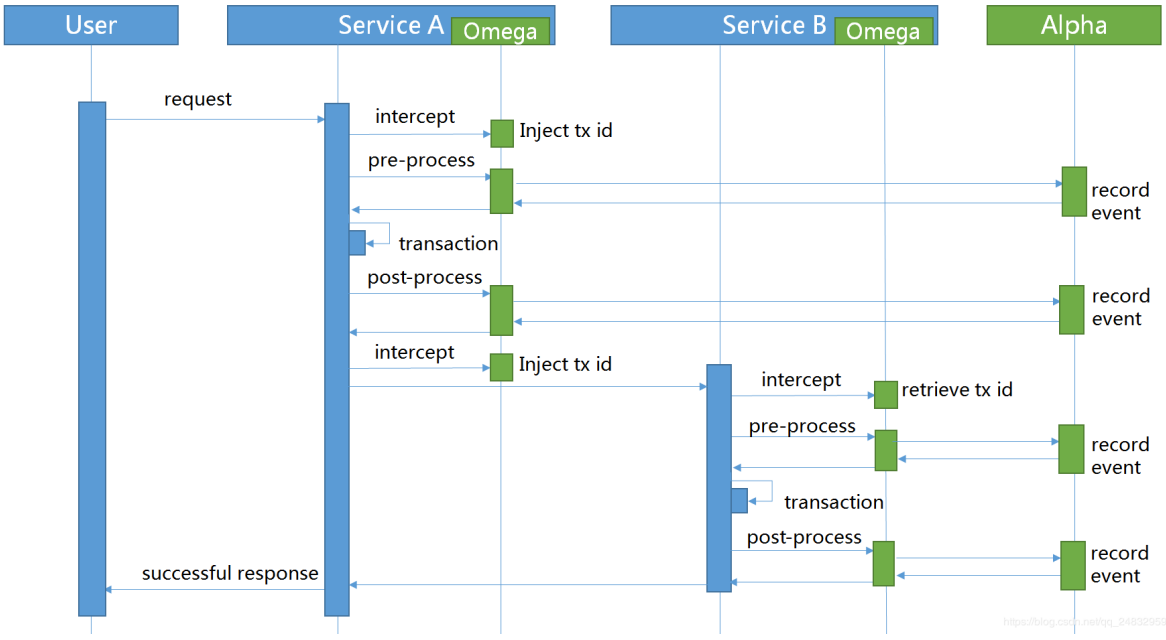


3、Saga 具体处理流程

Saga处理场景是要求相关的子事务提供事务处理函数同时也提供补偿函数。Saga协调器alpha会根据事务的执行情况向omega发送相关的指令，确定是否向前重试或者向后恢复。

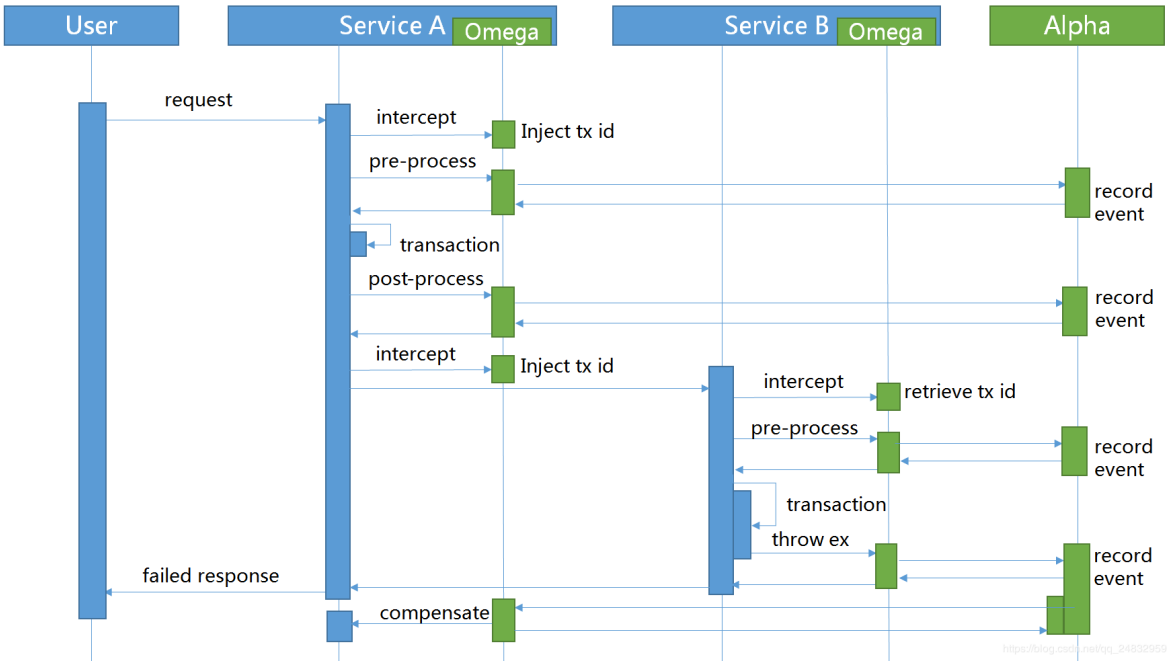
成功场景

成功场景下，每个事务都会有开始和有对应的结束事件。



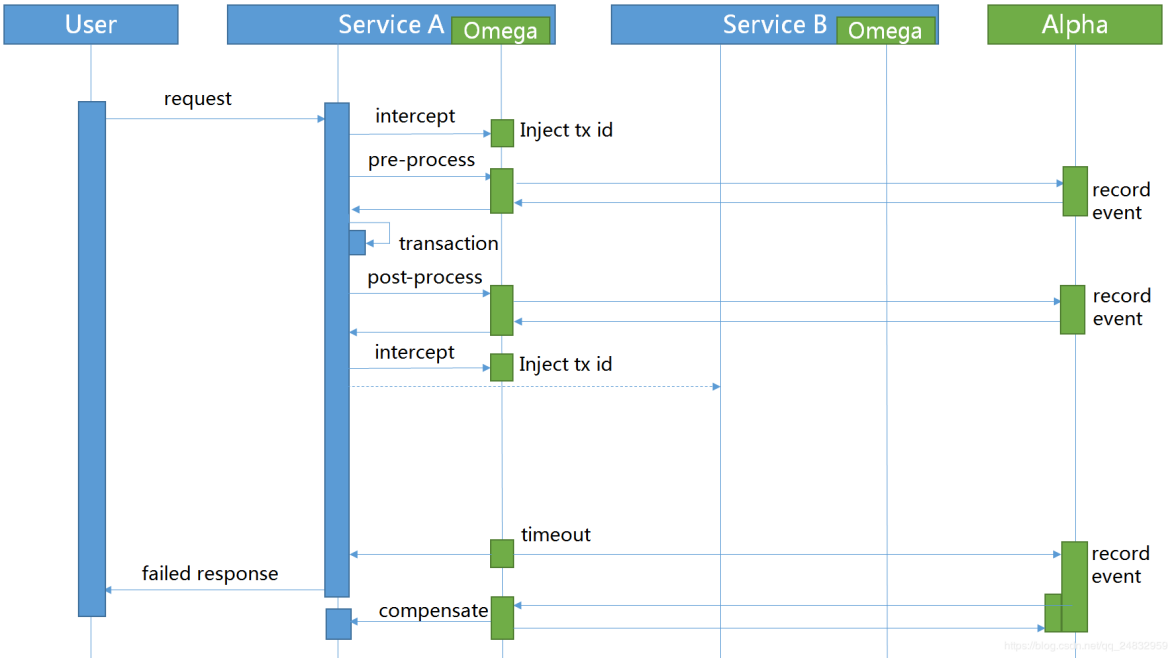
异常场景

异常场景下，omega会向alpha上报中断事件，然后alpha会向该全局事务的其它已完成的子事务发送补偿指令，确保最终所有的子事务要么都成功，要么都回滚。



超时场景 (需要调整)

超时场景下，已超时的事件会被alpha的定期扫描器检测出来，与此同时，该超时事务对应的全局事务也会被中断。



6.5 总结

1、作为分布式事务解决方案之一，Apache ServiceComb Saga支持tcc和saga两种模式，可实现业务逻辑的平滑迁移；

2、Apache ServiceComb Saga 基于spring注解和aop切面，对用户透明，业务侵入小，开发简单，部署容易；

3、在高可用方面。协调者alpha支持集群模式和本地持久化，不会 出现单点故障；

框架1：Seata

框架2： tx-lcn

框架3：tcc-transaction

框架4： Hmily

框架5：ByteTCC

框架6： EasyTransaction

框架7： Atomikos

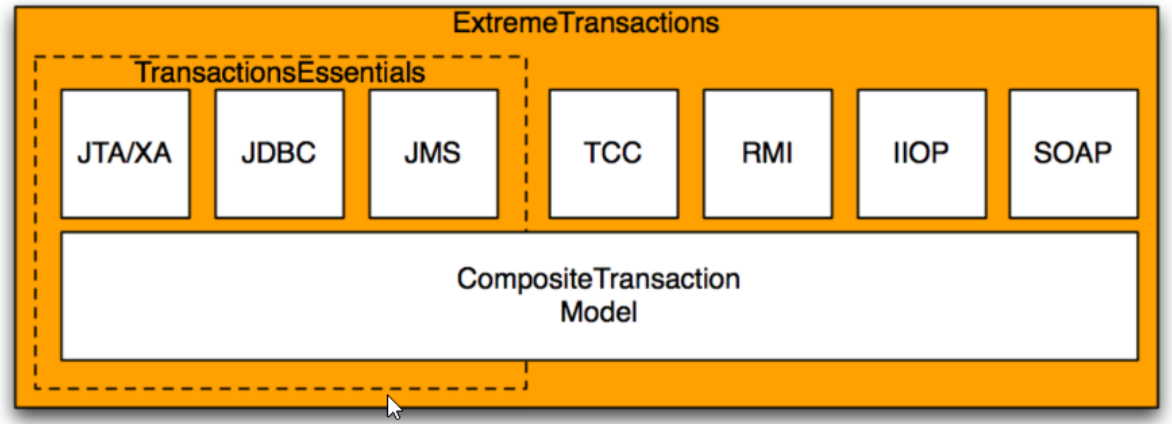
百度百科关于Atomikos介绍： https://baike.baidu.com/item/Atomikos/8482191?fr=aladdin

Atomikos公司官方网址为：https://www.atomikos.com/。其旗下最著名的产品就是事务管理器。产品分两个版本：

TransactionEssentials：开源的免费产品

ExtremeTransactions：上商业版，需要收费。

这两个产品的关系如下图所示：



TransactionEssentials：

1、实现了JTA/XA规范中的事务管理器(Transaction Manager)应该实现的相关接口，如：

UserTransaction实现是com.atomikos.icatch.jta.UserTransactionImp，用户只需要直接操作这个类

TransactionManager实现是com.atomikos.icatch.jta.UserTransactionManager

Transaction实现是com.atomikos.icatch.jta.TransactionImp

2、针对实现了JDBC规范中规定的实现了XADataSource接口的数据库连接池，以及实现了JMS规范的MQ客户端提供一层封装。

在上一节我们讲解JTA规范时，提到过XADataSource、XAConnection等接口应该由资源管理器RM来实现，而Atomikos的作用是一个事务管理器(TM)，并不需要提供对应的实现。而Atomikos对XADataSource进行封装，只是为了方便与事务管理器整合。封装XADataSource的实现类为`AtomikosDataSourceBean`。典型的XADataSource实现包括：

1、mysql官方提供的com.mysql.jdbc.jdbc2.optional.MysqlXADataSource

2、阿里巴巴开源的druid连接池，对应的实现类为com.alibaba.druid.pool.xa.DruidXADataSource

3、tomcat-jdbc连接池提供的org.apache.tomcat.jdbc.pool.XADataSource

而其他一些常用的数据库连接池，如dbcp、dbcp2或者c3p0，目前貌似尚未提供XADataSource接口的实现。如果提供给AtomikosDataSourceBean一个没有实现XADataSource接口的数据源，如c3p0的ComboPooledDataSource，则会抛出类似以下异常：

ExtremeTransactions在TransactionEssentials的基础上额外提供了以下功能：

支持TCC：这是一种柔性事务

支持通过RMI、IIOP、SOAP这些远程过程调用技术，进行事务传播。