分布式事务及实现方案

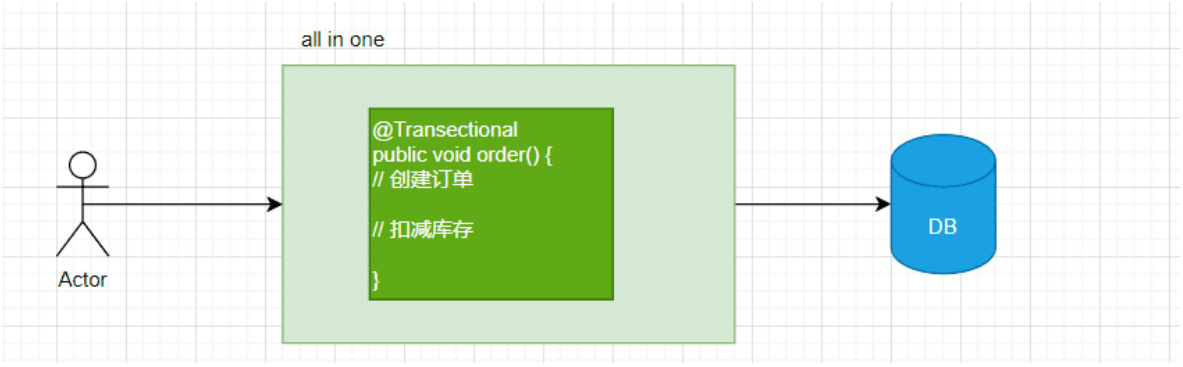
> **事务**是一个程序执行单元，里面的所有操作要么全部执行成功，要么全部执行失败。而**分布式事务**是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。

1.什么是分布式事务

> **事务**是一个程序执行单元，里面的所有操作要么全部执行成功，要么全部执行失败。在分布式系统中，这些操作可能是位于不同的服务中，那么如果也能保证这些操作要么全部执行成功要么全部执行失败呢？这便是分布式事务要解决的问题。

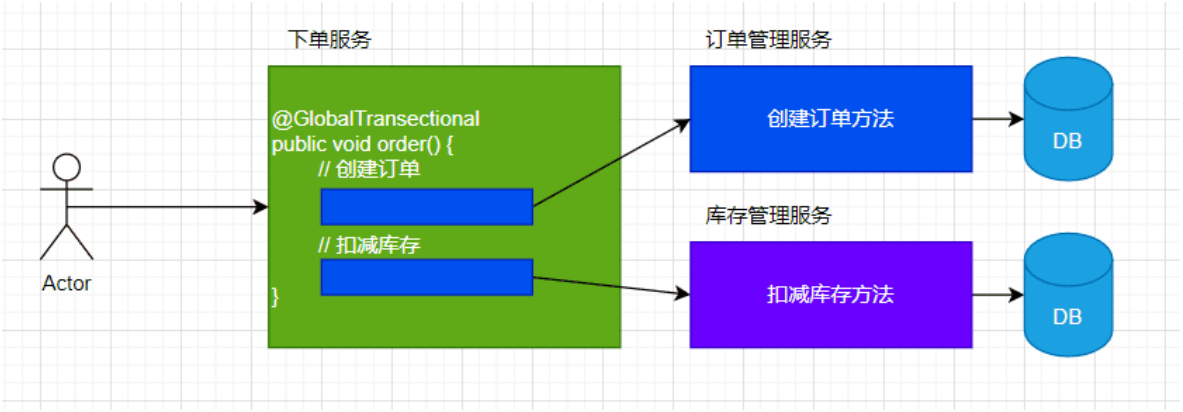
以一个网上的经典下单减库存例子为例：

单体应用所有的业务都使用一个数据库，整个下单流程或许只用在一个方法里同一个事务下操作数据库即可。此时所有操作都在一个事务里，要么全部提交，要么全部回滚。



但随着业务量不断增长，业务服务化拆分，就会分离出订单中心、库存中心等。而这样就造成业务间相互隔离，每个业务都维护着自己的数据库，数据的交换只能进行服务调用。

用户再下单时，创建订单和扣减库存，需要同时对订单DB和库存DB进行操作。两步操作必须同时成功，否则就会造成业务混乱，可此时我们只能保证自己服务的数据一致性，无法保证调用其他服务的操作是否成功，所以为了保证整个下单流程的数据一致性，就需要分布式事务介入。



2. 如何理解分布式事务

> 分布式的理论角度和分布式事务的知识体系角度理解分布式事务。

2.1 从分布式的理论的角度看

> 分布式的理论基础是CAP，由于P(分区容错）是必选项，所以只能在AP或者CP中选择。

- **分布式理论的CP** -> 刚性事务

遵循ACID，对数据要求强一致性

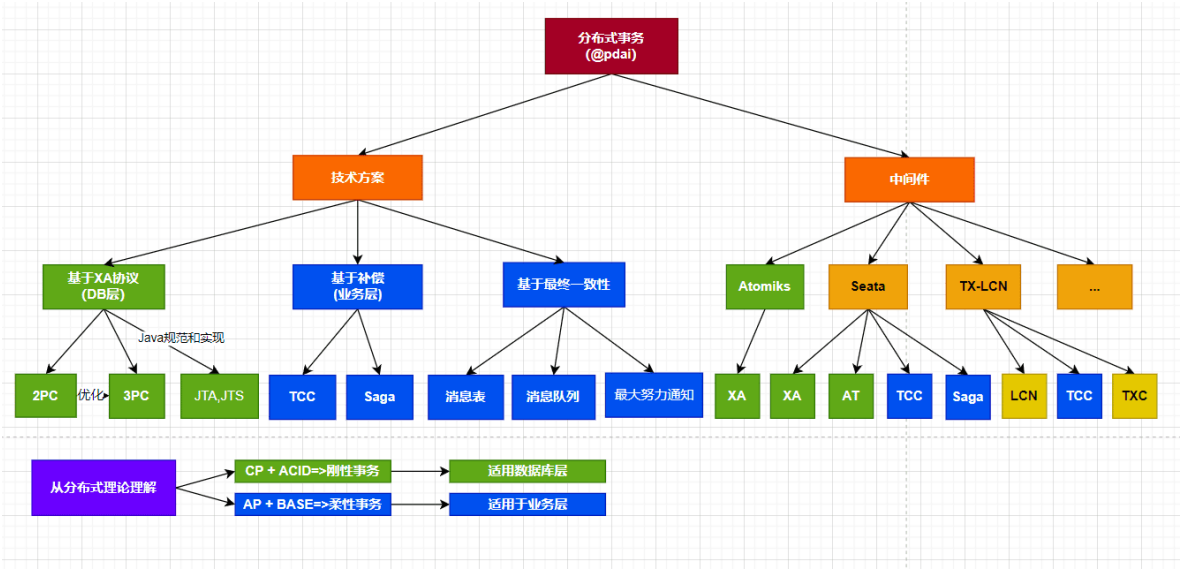
- **分布式理论的AP+BASE** -> 柔性事务

遵循BASE，允许一定时间内不同节点的数据不一致，但要求最终一致。

2.2 从分布式事务的体系看

> 我在梳理这个体系时，发现网上几乎都直接写2PC,3PC,TCC,Seata方案，而实际上这是不成体系的。从分布式事务的体系看，我认为至少应该理清楚，什么是技术方案，什么是中间件，以及形成这些方案的依据；以此，我画了如下的图。

如下图，可以帮助你构筑分布式事务的知识体系，一目了然。



- **刚性事务**：分布式理论的CP，遵循ACID，对数据要求强一致性。
  - **XA协议** 是一个基于数据库层面的分布式事务协议，其分为两部分：**事务管理器（Transaction Manager）**和**本地资源管理器（Resource Manager）**。事务管理器作为一个全局的调度者，负责对各个本地资源管理器统一号令提交或者回滚。主流的诸如Oracle、MySQL等数据库均已实现了XA接口。
    - **二阶提交协议（2PC）**：根据XA协议衍生出来而来; 引入一个作为协调者的组件来统一掌控所有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; 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。所谓的两个阶段是指：第一阶段：准备阶段 (投票阶段) 和第二阶段：提交阶段（执行阶段）
    - **三阶提交协议（3PC）**：是对两段提交（2PC）的一种升级优化，**3PC在2PC的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，各参与者节点的状态都一致。同时在协调者和参与者中都引入超时机制，当参与者各种原因未收到协调者的commit请求后，会对本地事务进行commit，不会一直阻塞等待，解决了2PC的单点故障问题，但3PC还是没能从根本上解决数据一致性的问题。
  - **Java事务规范**
    - **JTA**：Java事务API（Java Transaction API）是一个Java企业版的应用程序接口，在Java环境中，允许完成跨越多个XA资源的分布式事务。
    - **JTS**：Java事务服务（Java Transaction Service）是J2EE平台提供了分布式事务服务的具体实现规范，j2ee服务器提供商根据JTS规范实现事务并提供JTA接口。
- **柔性事务**：分布式理论的AP，遵循BASE，允许一定时间内不同节点的数据不一致，但要求最终一致。
  - **基于业务层**

- **TCC**: TCC（Try-Confirm-Cancel）又被称补偿事务，TCC与2PC的思想很相似，事务处理流程也很相似，但2PC是应用于在DB层面，TCC则可以理解为在应用层面的2PC，是需要我们编写业务逻辑来实现。
- **SAGA**：Saga是由一系列的本地事务构成。每一个本地事务在更新完数据库之后，会发布一条消息或者一个事件来触发Saga中的下一个本地事务的执行。如果一个本地事务因为某些业务规则无法满足而失败，Saga会执行在这个失败的事务之前成功提交的所有事务的补偿操作。Saga的实现有很多种方式，其中最流行的两种方式是：基于事件的方式和基于命令的方式。

- **最终一致性**
  - **消息表**：本地消息表的方案最初是由 eBay 提出，核心思路是将分布式事务拆分成本地事务进行处理。
  - **消息队列**：基于 MQ 的分布式事务方案其实是对本地消息表的封装，将本地消息表基于 MQ 内部，其他方面的协议基本与本地消息表一致。
  - **最大努力通知**：最大努力通知也称为定期校对，是对MQ事务方案的进一步优化。它在事务主动方增加了消息校对的接口，如果事务被动方没有接收到消息，此时可以调用事务主动方提供的消息校对的接口主动获取。

具体的内容可以看如下章节。

3. 分布式事务方案之刚性事务

> 说到刚性事务，首先要讲的是XA协议。XA协议是一个基于**数据库**的分布式事务协议，其分为两部分：**事务管理器（Transaction Manager）**和**本地资源管理器（Resource Manager）**。事务管理器作为一个全局的调度者，负责对各个本地资源管理器统一号令提交或者回滚。`二阶提交协议（2PC）`和`三阶提交协议（3PC）`就是根据此协议衍生出来而来。主流的诸如Oracle、MySQL等数据库均已实现了XA接口。

XA接口是双向的系统接口，在事务管理器（Transaction Manager）以及一个或多个资源管理器（Resource Manager）之间形成通信桥梁。也就是说，在基于XA的一个事务中，我们可以针对多个资源进行事务管理，例如一个系统访问多个数据库，或即访问数据库、又访问像消息中间件这样的资源。这样我们就能够实现在多个数据库和消息中间件直接实现全部提交、或全部取消的事务。**XA规范不是java的规范，而是一种通用的规范; Java 中的规范是JTA和JTS：Java事务API（Java Transaction API）是一个Java企业版的应用程序接口，在Java环境中，允许完成跨越多个XA资源的分布式事务；Java事务服务（Java Transaction Service）是J2EE平台提供了分布式事务服务的具体实现规范，j2ee服务器提供商根据JTS规范实现事务并提供JTA接口。**

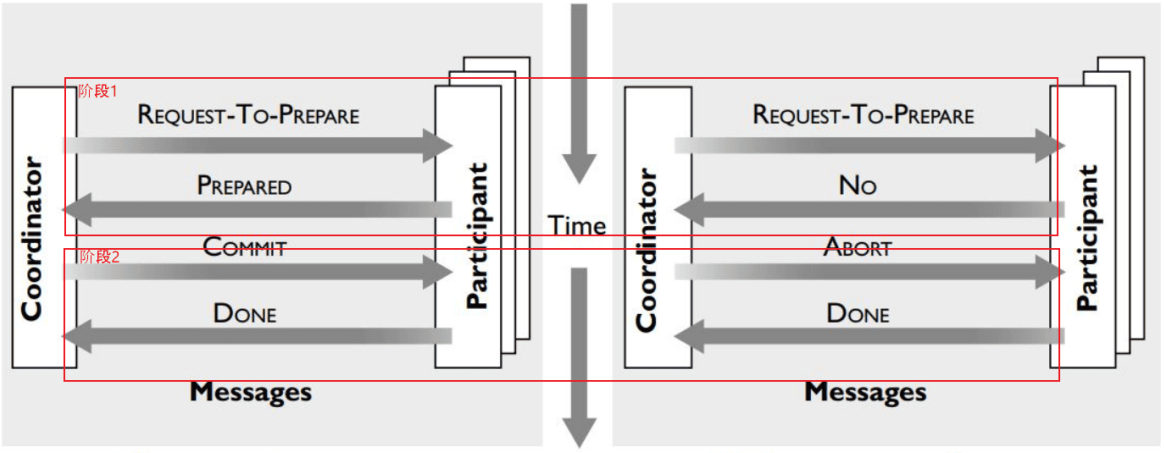
3.1 两段提交（2PC）

> 引入一个作为协调者（coordinator）的组件来统一掌控所有参与者（participant）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。

简单而言：参与者（participant）用来管理资源，协调者（coordinator）用来协调事务状态

两段提交（2PC - Prepare & Commit）是指两个阶段的提交：

- _第一阶段_: 准备阶段；
  - 协调者向所有参与者发送 REQUEST-TO-PREPARE
  - 当参与者收到REQUEST-TO-PREPARE 消息后, 它向协调者发送消息PREPARED或者NO，表示事务是否准备好；如果发送的是NO，那么事务要回滚；
- _第二阶段_: 提交阶段。
  - 协调者收集所有参与者的返回消息, 如果所有的参与者都回复的是PREPARED， 那么协调者向所有参与者发送COMMIT 消息；否则，协调者向所有回复PREPARED的参与者发送ABORT消息；
  - 参与者如果回复了PREPARED消息并且收到协调者发来的COMMIT消息，或者它收到ABORT消息，它将执行提交或回滚，并向协调者发送DONE消息以确认。



a. The transaction commits　　b. The transaction aborts

Horizontal arrows indicate messages between the coordinator and participant.
Time is moving down the page, so the first message in both cases is Request-To-Prepare.

两段提交（2PC）的缺点：

二阶段提交看似能够提供原子性的操作，但它存在着严重的缺陷：

- **网络抖动导致的数据不一致**：第二阶段中协调者向参与者发送commit命令之后，一旦此时发生网络抖动，导致一部分参与者接收到了commit请求并执行，可其他未接到commit请求的参与者无法执行事务提交。进而导致整个分布式系统出现了数据不一致。

- **超时导致的同步阻塞问题**：2PC中的所有的参与者节点都为事务阻塞型，当某一个参与者节点出现通信超时，其余参与者都会被动阻塞占用资源不能释放。
- **单点故障的风险**：由于严重的依赖协调者，一旦协调者发生故障，而此时参与者还都处于锁定资源的状态，无法完成事务commit操作。虽然协调者出现故障后，会重新选举一个协调者，可无法解决因前一个协调者宕机导致的参与者处于阻塞状态的问题。

2PC小结

2PC除本身的算法局限外，还有一个使用上的限制，就是它主要用在两个数据库之间（数据库实现了XA协议）。两个系统之间是无法使用2PC的，因为不会直接在底层的两个业务数据库之间做一致性，而是在两个服务上面实现一致性。

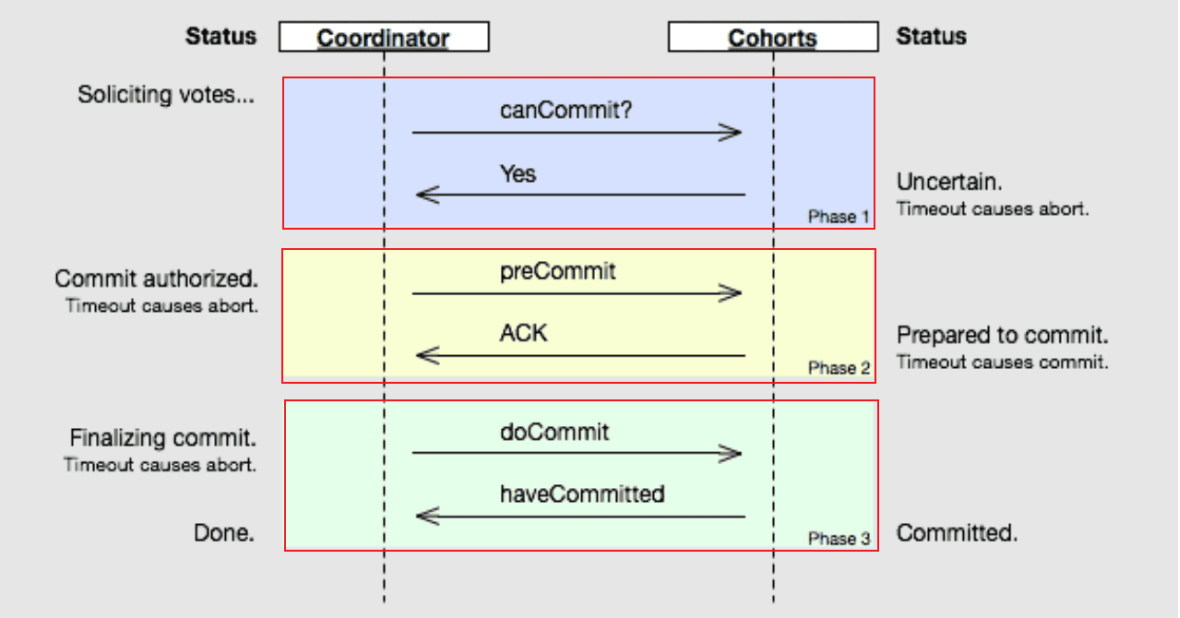
> _2PC只适用两个数据库（数据库实现了XA协议）之间；2PC有诸多问题和不便，在实践中一般很少使用_。

3.2 三段提交（3PC）

> 三段提交（3PC）是对两段提交（2PC）的一种升级优化，**3PC在2PC的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，各参与者节点的状态都一致。同时在协调者和参与者中都引入超时机制，当参与者各种原因未收到协调者的commit请求后，会对本地事务进行commit，不会一直阻塞等待，解决了2PC的单点故障问题，但3PC还是没能从根本上解决数据一致性的问题。

3PC的三个阶段分别是CanCommit、PreCommit、DoCommit：

- **CanCommit**：协调者向所有参与者发送CanCommit命令，询问是否可以执行事务提交操作。如果全部响应YES则进入下一个阶段。
- **PreCommit**：协调者向所有参与者发送PreCommit命令，询问是否可以进行事务的预提交操作，参与者接收到PreCommit请求后，如参与者成功的执行了事务操作，则返回Yes响应，进入最终commit阶段。一旦参与者中有向协调者发送了No响应，或因网络造成超时，协调者没有接到参与者的响应，协调者向所有参与者发送abort请求，参与者接受abort命令执行事务的中断。
- **DoCommit**：在前两个阶段中所有参与者的响应反馈均是YES后，协调者向参与者发送DoCommit命令正式提交事务，如协调者没有接收到参与者发送的ACK响应，会向所有参与者发送abort请求命令，执行事务的中断。



3PC存在的问题

3PC工作在同步网络模型上，它假设消息传输时间是有上界的，只存在机器失败而不存在消息失败。这个假设太强，现实的情形是，机器失败是无法完美地检测出来的，消息传输可能因为网络拥堵花费很多时间。同时, 说阻塞是相对, 存在协调者和参与者同时失败的情形下, 3PC事务依然会阻塞。实际上，很少会有系统实现3PC，多数现实的系统会通过复制状态机解决2PC阻塞的问题。比如，如果失败模型不是失败-停止, 而是消息失败（消息延迟或网络分区），那样3PC会产生不一致的情形。

3PC小结

> _3PC并没有完美解决2PC的阻塞，也引入了新的问题（不一致问题），所以3PC很少会被真正的使用_。

4. 分布式事务方案之柔性事务

> 柔性事务：分布式理论的AP，遵循BASE，允许一定时间内不同节点的数据不一致，但要求最终一致。

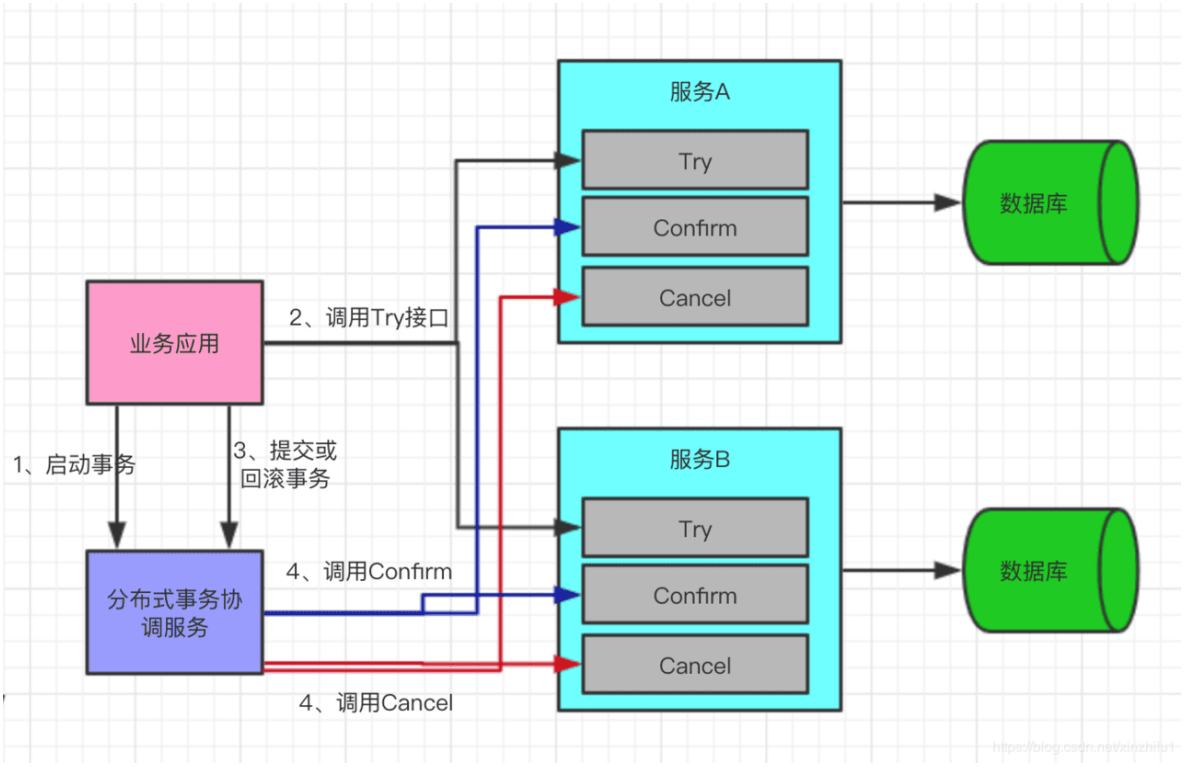
4.1 补偿事务 (TCC)

> TCC（Try-Confirm-Cancel）又被称补偿事务，TCC与2PC的思想很相似，事务处理流程也很相似，但**2PC是应用于在DB层面，TCC则可以理解为在应用层面的2PC，是需要我们编写业务逻辑来实现**。

TCC它的核心思想是："针对每个操作都要注册一个与其对应的确认（Try）和补偿（Cancel）"。

还拿下单扣库存解释下它的三个操作：

- **Try阶段**：下单时通过Try操作去扣除库存预留资源。
- **Confirm阶段**：确认执行业务操作，在只预留的资源基础上，发起购买请求。
- **Cancel阶段**：只要涉及到的相关业务中，有一个业务方预留资源未成功，则取消所有业务资源的预留请求。



TCC的缺点：

1.空回滚

当一个分支事务所在的服务发生宕机或者网络异常导致调用失败，并未执行try方法，当恢复后事务执行回滚操作就会调用此分支事务的cancel方法,如果cancel方法不能处理此种情况就会出现空回滚。

是否出现空回滚，我们需要需要判断是否执行了try方法，如果执行了就没有空回滚。解决方法就是当主业务发起事务时，生成一个全局事务记录，并生成一个全局唯一ID，贯穿整个事务，再创建一张分支事务记录表，用于记录分支事务，try执行时将全局事务ID和分支事务ID存入分支事务表中，表示执行了try阶段，当cancel执行时，先判断表中是否有该全局事务ID的数据，如果有则回滚，否则不做任何操作。比如seata的AT模式中就有分支事务表。

2.幂等问题

由于服务宕机或者网络问题，方法的调用可能出现超时，为了保证事务正常执行我们往往会加入重试的机制，因此就需要保证confirm和cancel阶段操作的幂等性。

我们可以在分支事务记录表中增加事务执行状态，每次执行confirm和cancel方法时都查询该事务的执行状态，以此判断事务的幂等性。

3.悬挂问题

TCC中，在调用try之前会先注册分支事务，注册分支事务之后，调用出现超时，此时try请求还未到达对应的服务，因为调用超时了，所以会执行cancel调用，此时cancel已经执行完了，然而这个时候try请求到达了，这个时候执行了try之后就没有后续的操作了，就会导致资源挂起，无法释放。

执行try方法时我们可以判断confirm或者cancel方法是否执行，如果执行了那么就不执行try阶段。同样借助分支事务表中事务的执行状态。如果已经执行了confirm或者cancel那么try就执行。

4.2 Saga事务

> Saga是分布式事务领域最有名气的解决方案之一，最初出现在1987年Hector Garcaa-Molrna & Kenneth Salem发表的论文SAGAS里。 如下内容主要来源于这里在新窗口打开，然后由flyingww翻译整理在《分布式事务系列三：Saga》。

Saga是由一系列的本地事务构成。每一个本地事务在更新完数据库之后，会发布一条消息或者一个事件来触发Saga中的下一个本地事务的执行。如果一个本地事务因为某些业务规则无法满足而失败，Saga会执行在这个失败的事务之前成功提交的所有事务的补偿操作。

Saga的实现有很多种方式，其中最流行的两种方式是：

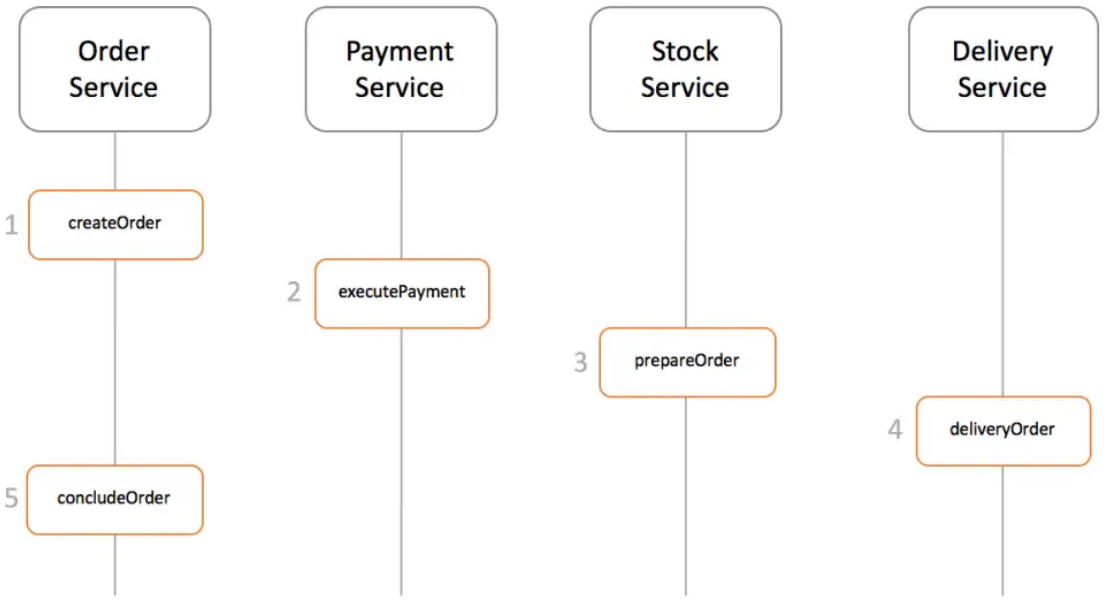
- **基于事件的方式。**这种方式没有协调中心，整个模式的工作方式就像舞蹈一样，各个舞蹈演员按照预先编排的动作和走位各自表演，最终形成一只舞蹈。处于当前Saga下的各个服务，会产生某类事件，或者监听其它服务产生的事件并决定是否需要针对监听到的事件做出响应。
- **基于命令的方式。**这种方式的工作形式就像一只乐队，由一个指挥家（协调中心）来协调大家的工作。协调中心来告诉Saga的参与方应该执行哪一个本地事务。

我们继续以订单流程为例，说明一下该模式。

假设一个完整的订单流程包含了如下几个服务：

1. Order Service：订单服务
2. Payment Service：支付服务
3. Stock Service：库存服务

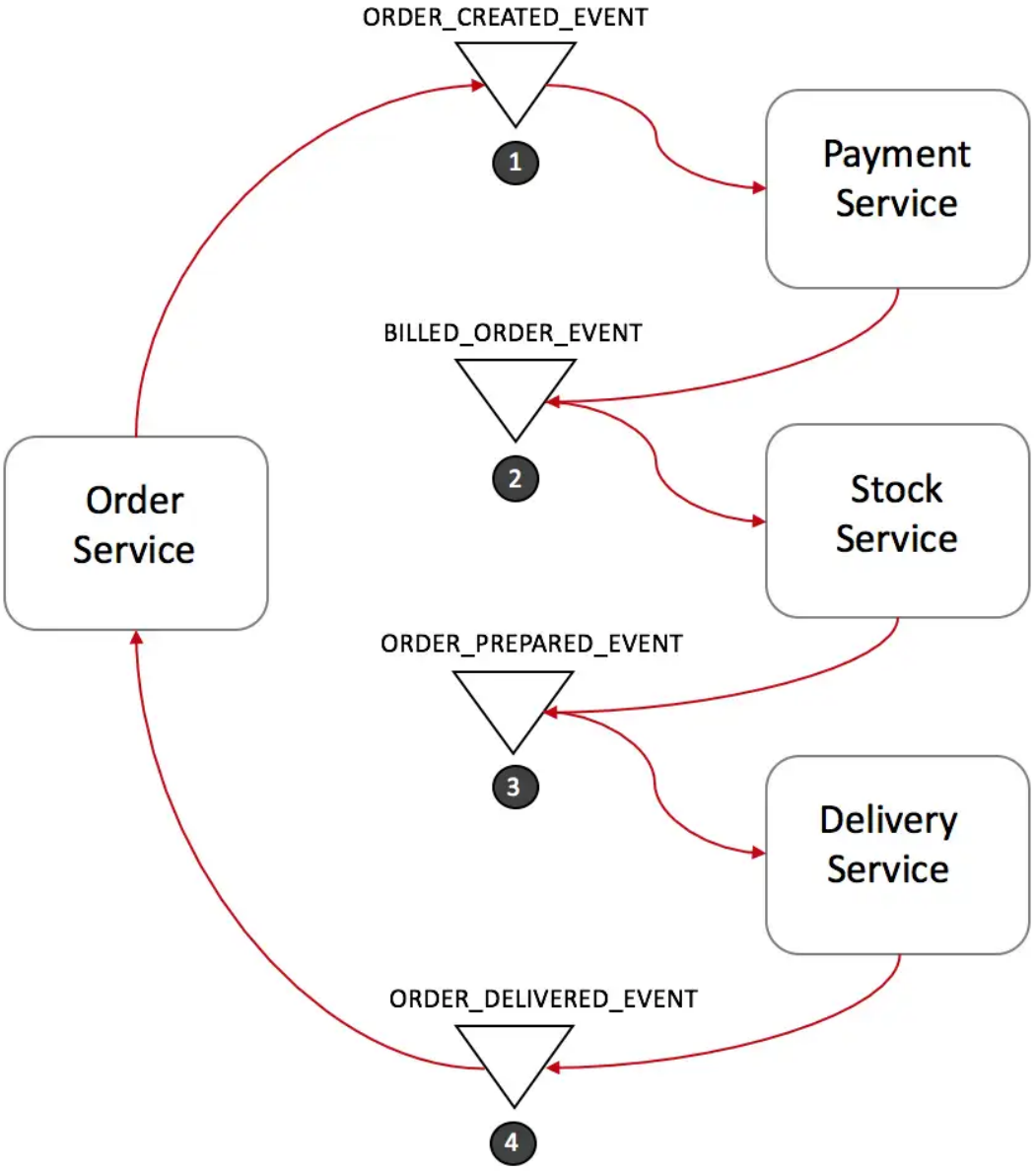
4. Delivery Service：物流服务



4.2.1 基于事件的方式

在基于事件的方式中，第一个服务执行完本地事务之后，会产生一个事件。其它服务会监听这个事件，触发该服务本地事务的执行，并产生新的事件。

采用基于事件的saga模式的订单处理流程如下：



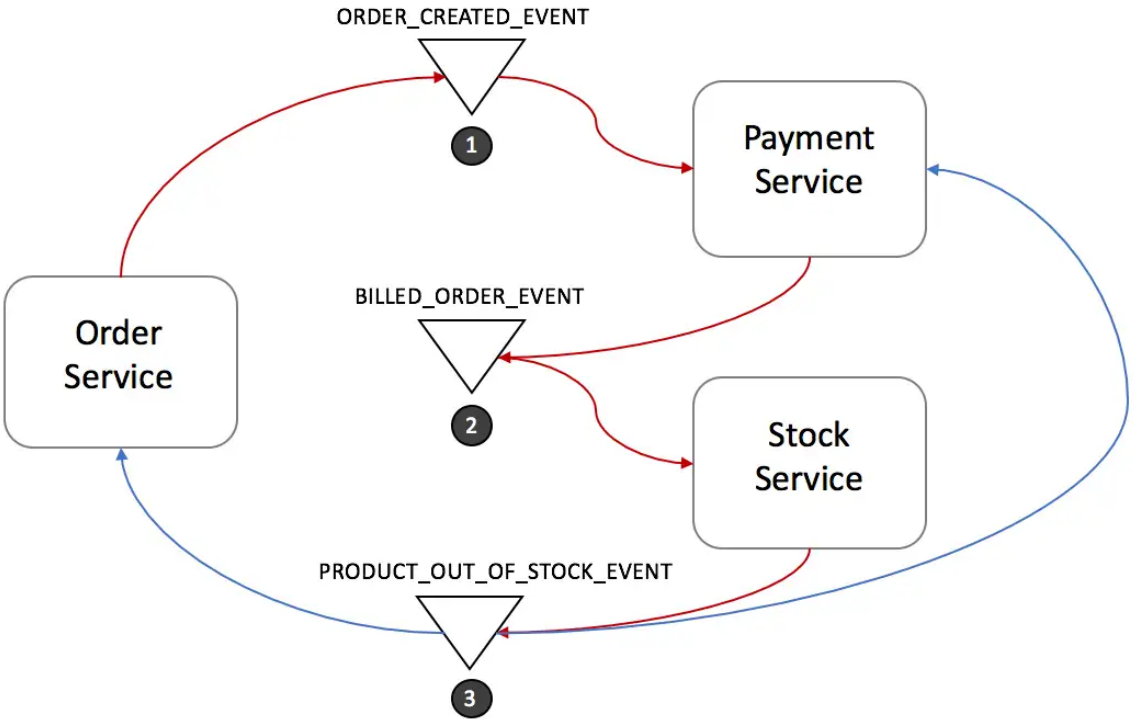
1. 订单服务创建一笔新订单，将订单状态设置为"待处理"，产生事件ORDER_CREATED_EVENT。
2. 支付服务监听ORDER_CREATED_EVENT，完成扣款并产生事件BILLED_ORDER_EVENT。
3. 库存服务监听BILLED_ORDER_EVENT，完成库存扣减和备货，产生事件ORDER_PREPARED_EVENT。
4. 物流服务监听ORDER_PREPARED_EVENT，完成商品配送，产生事件ORDER_DELIVERED_EVENT。
5. 订单服务监听ORDER_DELIVERED_EVENT，将订单状态更新为"完成"。

在这个流程中，订单服务很可能还会监听BILLED_ORDER_EVENT，ORDER_PREPARED_EVENT来完成订单状态的实时更新。将订单状态分别更新为"已经支付"和"已经出库"等状态来及时反映订单的最新状态。

该模式下分布式事务的回滚

为了在异常情况下回滚整个分布式事务，我们需要为相关服务提供补偿操作接口。

假设库存服务由于库存不足没能正确完成备货，我们可以按照下面的流程来回滚整个Saga事务：



1. 库存服务产生事件PRODUCT_OUT_OF_STOCK_EVENT。
2. 订单服务和支付服务都会监听该事件并做出响应：
    1. 支付服务完成退款。
    2. 订单服务将订单状态设置为"失败"。

基于事件方式的优缺点

优点：简单且容易理解。各参与方相互之间无直接沟通，完全解耦。这种方式比较适合整个分布式事务只有2-4个步骤的情形。

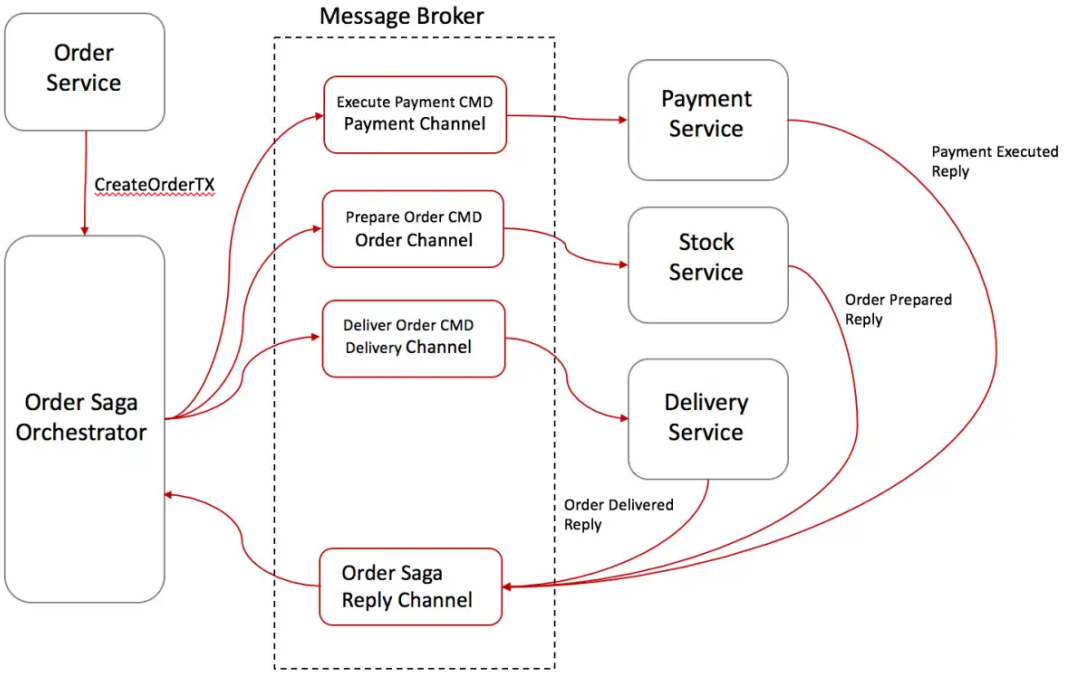
缺点：这种方式如果涉及比较多的业务参与方，则比较容易失控。各业务参与方可随意监听对方的消息，以至于最后没人知道到底有哪些系统在监听哪些消息。更悲催的是，这个模式还可能产生环形监听，也就是两个业务方相互监听对方所产生的事件。

接下来，我们将介绍如何使用命令的方式来克服上面提到的缺点。

4.2.2 基于命令的方式

在基于命令的方式中，我们会定义一个新的服务，这个服务扮演的角色就和一支交响乐乐队的指挥一样，告诉各个业务参与方，在什么时候做什么事情。我们管这个新服务叫做协调中心。协调中心通过命令/回复的方式来和Saga中其它服务进行交互。

我们继续以之前的订单流程来举例。下图中的Order Saga Orchestrator就是新引入的协调中心。

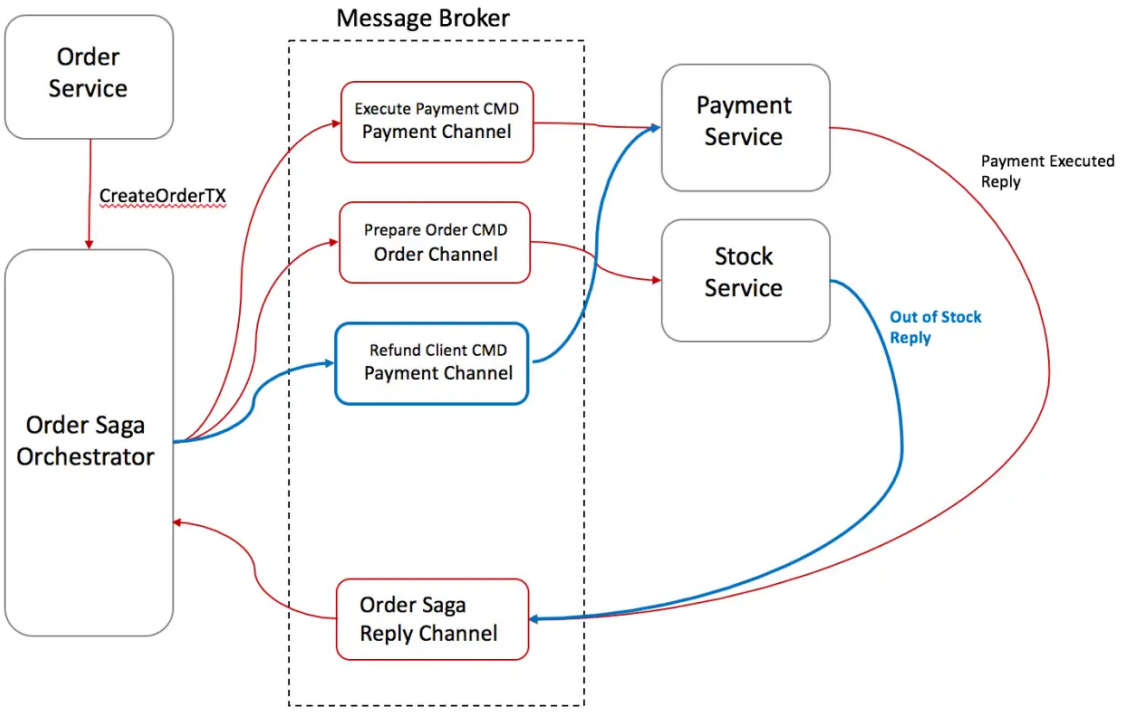


1. 订单服务创建一笔新订单，将订单状态设置为"待处理"，然后让Order Saga Orchestrator（OSO）开启创建订单事务。
2. OSO发送一个"支付命令"给支付服务，支付服务完成扣款并回复"支付完成"消息。
3. OSO发送一个"备货命令"给库存服务，库存服务完成库存扣减和备货，并回复"出库"消息。
4. OSO发送一个"配送命令"给物流服务，物流服务完成配送，并回复"配送完成"消息。
5. OSO向订单服务发送"订单结束命令"给订单服务，订单服务将订单状态设置为"完成"。
6. OSO清楚一个订单处理Saga的具体流程，并在出现异常时向相关服务发送补偿命令来回滚整个分布式事务。

实现协调中心的一个比较好的方式是使用**状态机(Sate Machine)**。

该模式下分布式事务的回滚

该模式下的回滚流程如下：



1. 库存服务回复OSO一个"库存不足"消息。
2. OSO意识到该分布式事务失败了，触发回滚流程：
3. OSO发送"退款命令"给支付服务，支付服务完成退款并回复"退款成功"消息。
4. OSO向订单服务发送"将订单状态改为失败命令"，订单服务将订单状态更新为"失败"。

基于命令方式的优缺点

优点：

1. 避免了业务方之间的环形依赖。
2. 将分布式事务的管理交由协调中心管理，协调中心对整个逻辑非常清楚。
3. 减少了业务参与方的复杂度。这些业务参与方不再需要监听不同的消息，只是需要响应命令并回复消息。
4. 测试更容易（分布式事务逻辑存在于协调中心，而不是分散在各业务方）。
5. 回滚也更容易。

缺点：

1. 一个可能的缺点就是需要维护协调中心，而这个协调中心并不属于任何业务方。

4.2.3 Saga模式建议

1，给每一个分布式事务创建一个唯一的Tx id。这个唯一的Tx id可以用来在各个业务参与方沟通时精确定位哪一笔分布式事务。

2，对于基于命令的方式，在命令中携带回复地址。这种方式可以让服务同时响应多个协调中心请求。

3，幂等性。幂等性能够增加系统的容错性，让各个业务参与方服务提供幂等性操作，能够在遇到异常情况下进行重试。

4，尽量在命令或者消息中携带下游处理需要的业务数据，避免下游处理时需要调用消息产生方接口获取更多数据。减少系统之间的相互依赖。

4.3 本地消息表

本地消息表的方案最初是由 eBay 提出，核心思路是将分布式事务拆分成本地事务进行处理。此章节来源于CSDN博主不才陈某这里在新窗口打开

角色：

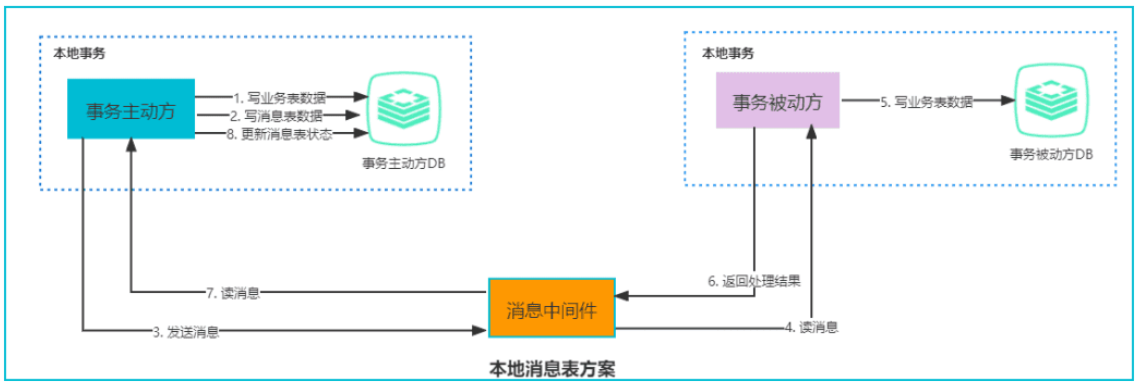
- 事务主动方
- 事务被动方

通过在事务主动发起方额外新建事务消息表，事务发起方处理业务和记录事务消息在本地事务中完成，轮询事务消息表的数据发送事务消息，事务被动方基于消息中间件消费事务消息表中的事务。

这样可以避免以下两种情况导致的数据不一致性：

- 业务处理成功、事务消息发送失败
- 业务处理失败、事务消息发送成功

整体的流程如下图：



上图中整体的处理步骤如下：

1. 事务主动方在同一个本地事务中处理业务和写消息表操作
2. 事务主动方通过消息中间件，通知事务被动方处理事务通知事务待消息。消息中间件可以基于 Kafka、RocketMQ 消息队列，事务主动方主动写消息到消息队列，事务消费方消费并处理消息队列中的消息。
3. 事务被动方通过消息中间件，通知事务主动方事务已处理的消息。
4. 事务主动方接收中间件的消息，更新消息表的状态为已处理。

一些必要的容错处理如下：

- 当1处理出错，由于还在事务主动方的本地事务中，直接回滚即可
- 当2,3处理出错，由于事务主动方本地保存了消息，只需要轮询消息重新通过消息中间件发送，事务被动方重新读取消息处理业务即可。
- 如果是业务上处理失败，事务被动方可以发消息给事务主动方回滚事务
- 如果事务被动方已经消费了消息，事务主动方需要回滚事务的话，需要发消息通知事务主动方进行回滚事务。

优点

- 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消息中间件，弱化了对 MQ 中间件特性的依赖。
- 方案轻量，容易实现。

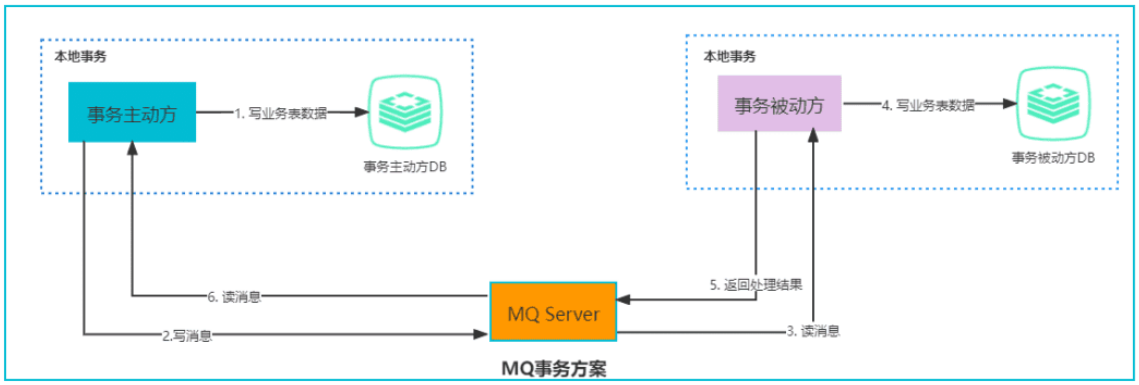
缺点

- 与具体的业务场景绑定，耦合性强，不可公用。
- 消息数据与业务数据同库，占用业务系统资源。
- 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。

4.4 MQ事务方案（可靠消息事务）

> 基于 MQ 的分布式事务方案其实是对本地消息表的封装，将本地消息表基于 MQ 内部，其他方面的协议基本与本地消息表一致。

MQ事务方案整体流程和本地消息表的流程很相似，如下图：

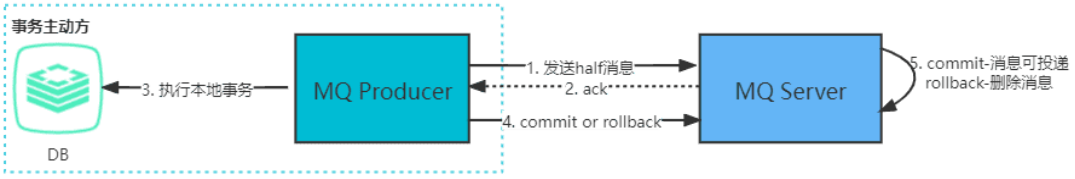


从上图可以看出和本地消息表方案唯一不同就是将本地消息表存在了MQ内部，而不是业务数据库中。

那么MQ内部的处理尤为重要，下面主要基于 RocketMQ 4.3 之后的版本介绍 MQ 的分布式事务方案。

在本地消息表方案中，保证事务主动方发写业务表数据和写消息表数据的一致性是基于数据库事务，RocketMQ 的事务消息相对于普通 MQ提供了 2PC 的提交接口，方案如下：

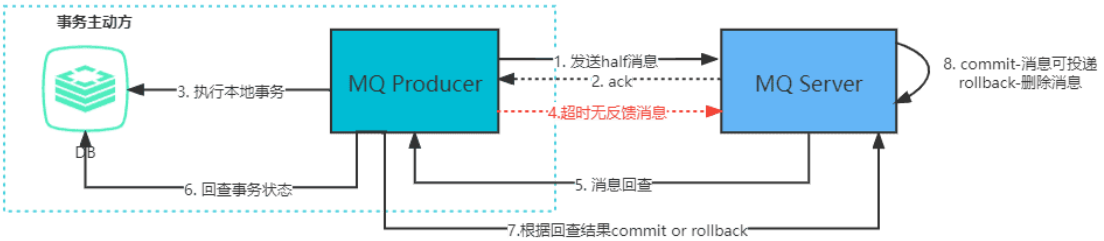
正常情况：事务主动方发消息



这种情况下，事务主动方服务正常，没有发生故障，发消息流程如下：

- 发送方向 MQ 服务端(MQ Server)发送 half 消息。
- MQ Server 将消息持久化成功之后，向发送方 ack 确认消息已经发送成功。
- 发送方开始执行本地事务逻辑。
- 发送方根据本地事务执行结果向 MQ Server 提交二次确认（commit 或是 rollback）。
- MQ Server 收到 commit 状态则将半消息标记为可投递，订阅方最终将收到该消息；MQ Server 收到 rollback 状态则删除半消息，订阅方将不会接受该消息。

异常情况：事务主动方消息恢复



在断网或者应用重启等异常情况下，图中 4 提交的二次确认超时未到达 MQ Server，此时处理逻辑如下：

- MQ Server 对该消息发起消息回查。
- 发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
- 发送方根据检查得到的本地事务的最终状态再次提交二次确认。
- MQ Server基于 commit/rollback 对消息进行投递或者删除。

优点

相比本地消息表方案，MQ 事务方案优点是：

- 消息数据独立存储，降低业务系统与消息系统之间的耦合。
- 吞吐量大于使用本地消息表方案。

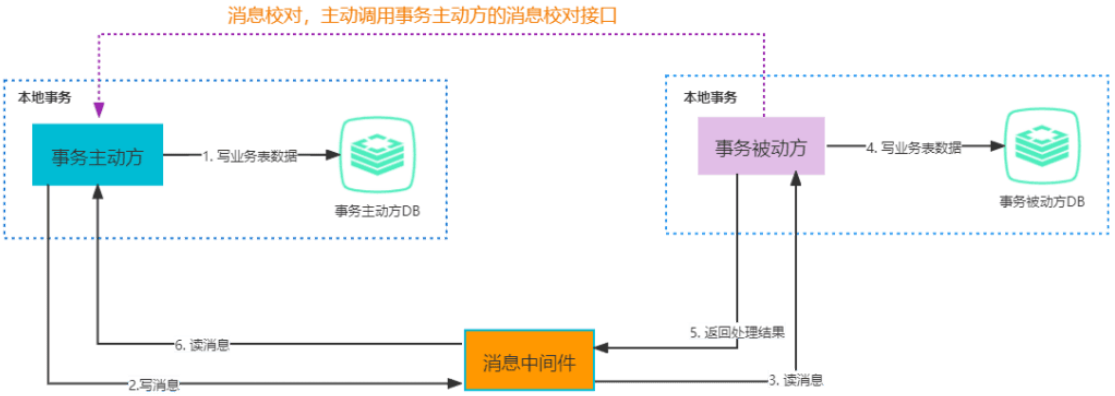
缺点

- 一次消息发送需要两次网络请求(half 消息 + commit/rollback 消息) 。
- 业务处理服务需要实现消息状态回查接口。

4.5 最大努力通知

> 最大努力通知也称为定期校对，是对MQ事务方案的进一步优化。它在事务主动方增加了消息校对的接口，如果事务被动方没有接收到消息，此时可以调用事务主动方提供的消息校对的接口主动获取。

最大努力通知的整体流程如下图：



在可靠消息事务中，事务主动方需要将消息发送出去，并且消息接收方成功接收，这种可靠性发送是由事务主动方保证的；

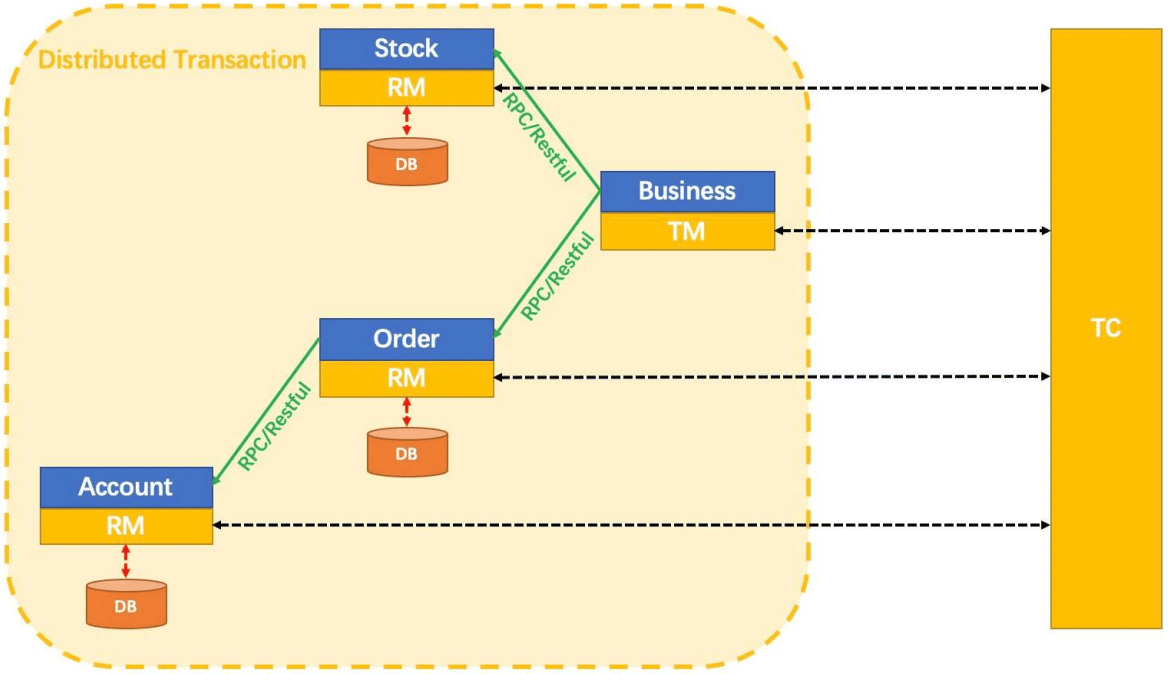
但是最大努力通知，事务主动方尽最大努力（重试，轮询....）将事务发送给事务接收方，但是仍然存在消息接收不到，此时需要事务被动方主动调用事务主动方的消息校对接口查询业务消息并消费，这种通知的可靠性是由事务被动方保证的。

最大努力通知适用于业务通知类型，例如微信交易的结果，就是通过最大努力通知方式通知各个商户，既有回调通知，也有交易查询接口。

5. 分布式事务的中间件Seata

Seata 是一款开源的分布式事务解决方案，致力于提供高性能和简单易用的分布式事务服务。Seata 将为用户提供了 AT、TCC、SAGA 和 XA 事务模式，为用户打造一站式的分布式解决方案。如下内容来源于 Seata官网在新窗口打开

- **TC (Transaction Coordinator) - 事务协调者**: 维护全局和分支事务的状态，驱动全局事务提交或回滚。
- **TM (Transaction Manager) - 事务管理器**: 定义全局事务的范围：开始全局事务、提交或回滚全局事务。
- **RM (Resource Manager) - 资源管理器**: 管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。



5.1 Seata AT 模式

5.1.1 前提

- 基于支持本地 ACID 事务的关系型数据库。
- Java 应用，通过 JDBC 访问数据库。

5.1.2 整体机制

两阶段提交协议的演变：

- 一阶段：业务数据和回滚日志记录在同一个本地事务中提交，释放本地锁和连接资源。
- 二阶段：
  - 提交异步化，非常快速地完成。
  - 回滚通过一阶段的回滚日志进行反向补偿。

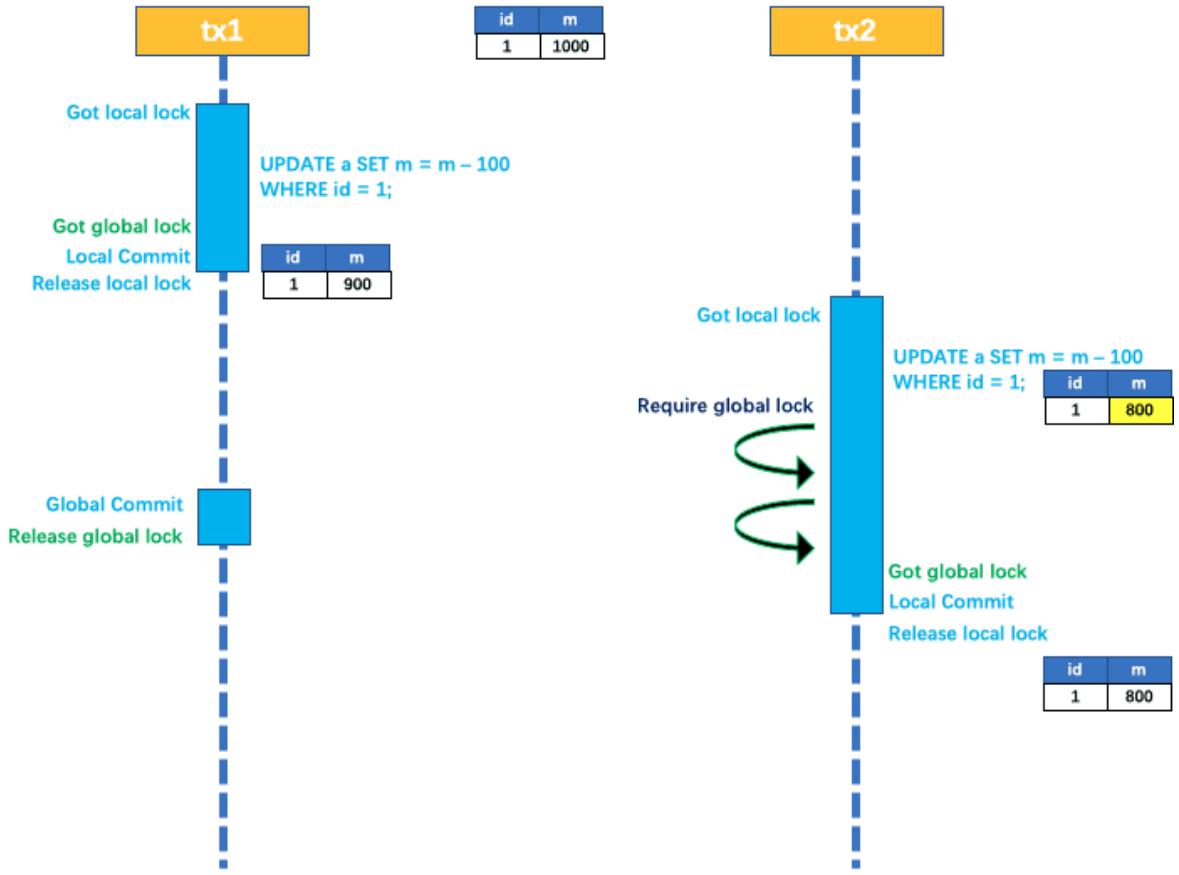
5.1.3 写隔离

- 一阶段本地事务提交前，需要确保先拿到 全局锁 。
- 拿不到 全局锁 ，不能提交本地事务。
- 拿 全局锁 的尝试被限制在一定范围内，超出范围将放弃，并回滚本地事务，释放本地锁。

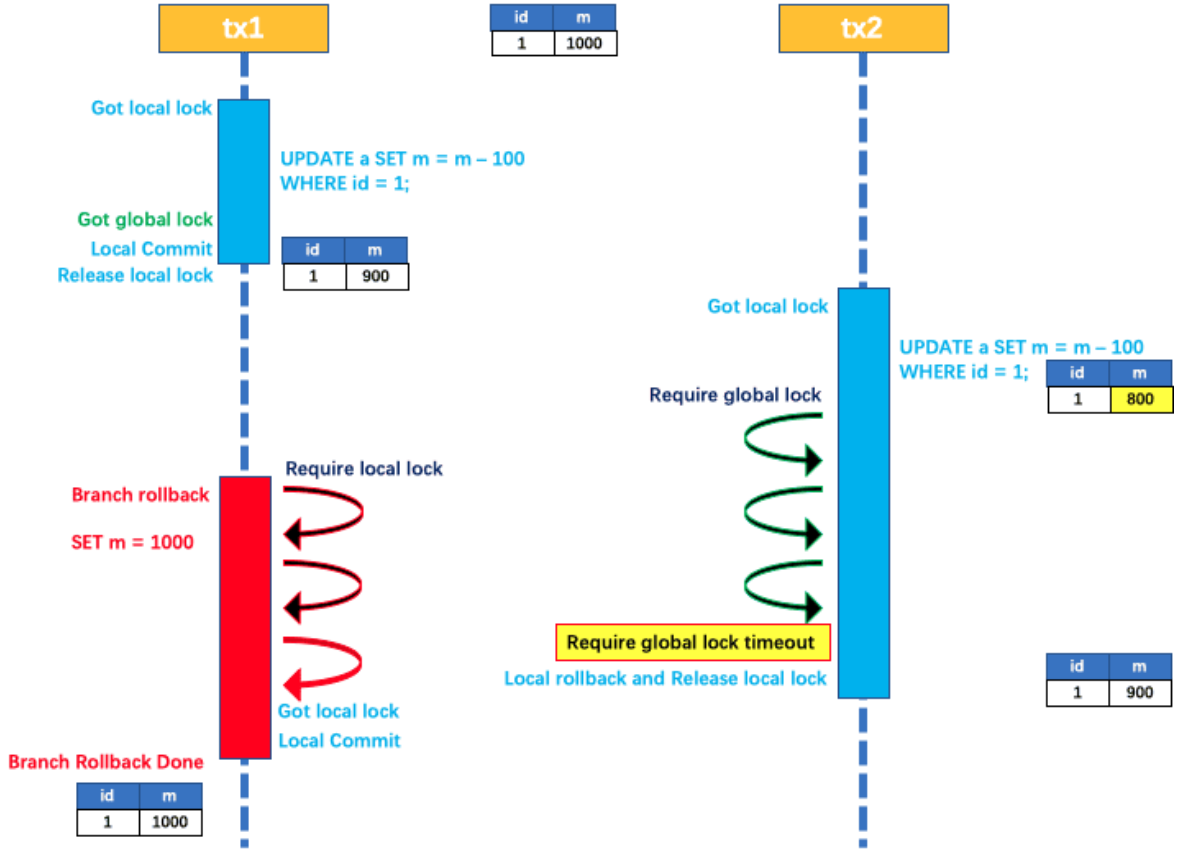
以一个示例来说明：

两个全局事务 tx1 和 tx2，分别对 a 表的 m 字段进行更新操作，m 的初始值 1000。

tx1 先开始，开启本地事务，拿到本地锁，更新操作 m = 1000 - 100 = 900。本地事务提交前，先拿到该记录的 全局锁 ，本地提交释放本地锁。 tx2 后开始，开启本地事务，拿到本地锁，更新操作 m = 900 - 100 = 800。本地事务提交前，尝试拿该记录的 全局锁 ，tx1 全局提交前，该记录的全局锁被 tx1 持有，tx2 需要重试等待 全局锁 。



tx1 二阶段全局提交，释放 全局锁 。tx2 拿到 全局锁 提交本地事务。



如果 tx1 的二阶段全局回滚，则 tx1 需要重新获取该数据的本地锁，进行反向补偿的更新操作，实现分支的回滚。

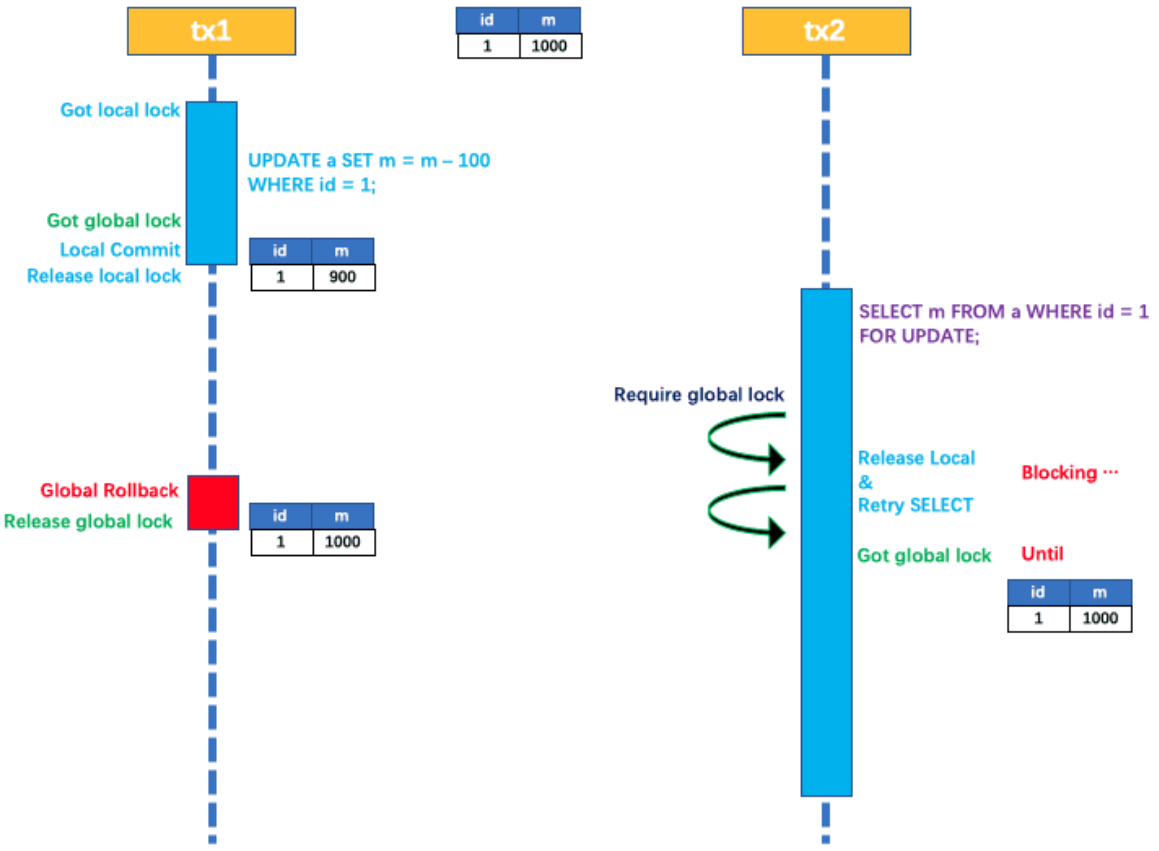
此时，如果 tx2 仍在等待该数据的 全局锁，同时持有本地锁，则 tx1 的分支回滚会失败。分支的回滚会一直重试，直到 tx2 的 全局锁 等锁超时，放弃 全局锁 并回滚本地事务释放本地锁，tx1 的分支回滚最终成功。

因为整个过程 全局锁 在 tx1 结束前一直是被 tx1 持有的，所以不会发生 脏写 的问题。

5.1.4 读隔离

在数据库本地事务隔离级别 读已提交（Read Committed） 或以上的基础上，Seata（AT 模式）的默认全局隔离级别是 读未提交（Read Uncommitted） 。

如果应用在特定场景下，必需要求全局的 读已提交 ， 目前 Seata 的方式是通过 SELECT FOR UPDATE 语句的代理。



SELECT FOR UPDATE 语句的执行会申请 全局锁 ，如果 全局锁 被其他事务持有，则释放本地锁（回滚 SELECT FOR UPDATE 语句的本地执行）并重试。这个过程中，查询是被 block 住的，直到 全局锁 拿到，即读取的相关数据是 已提交 的，才返回。

出于总体性能上的考虑，Seata 目前的方案并没有对所有 SELECT 语句都进行代理，仅针对 FOR UPDATE 的 SELECT 语句。

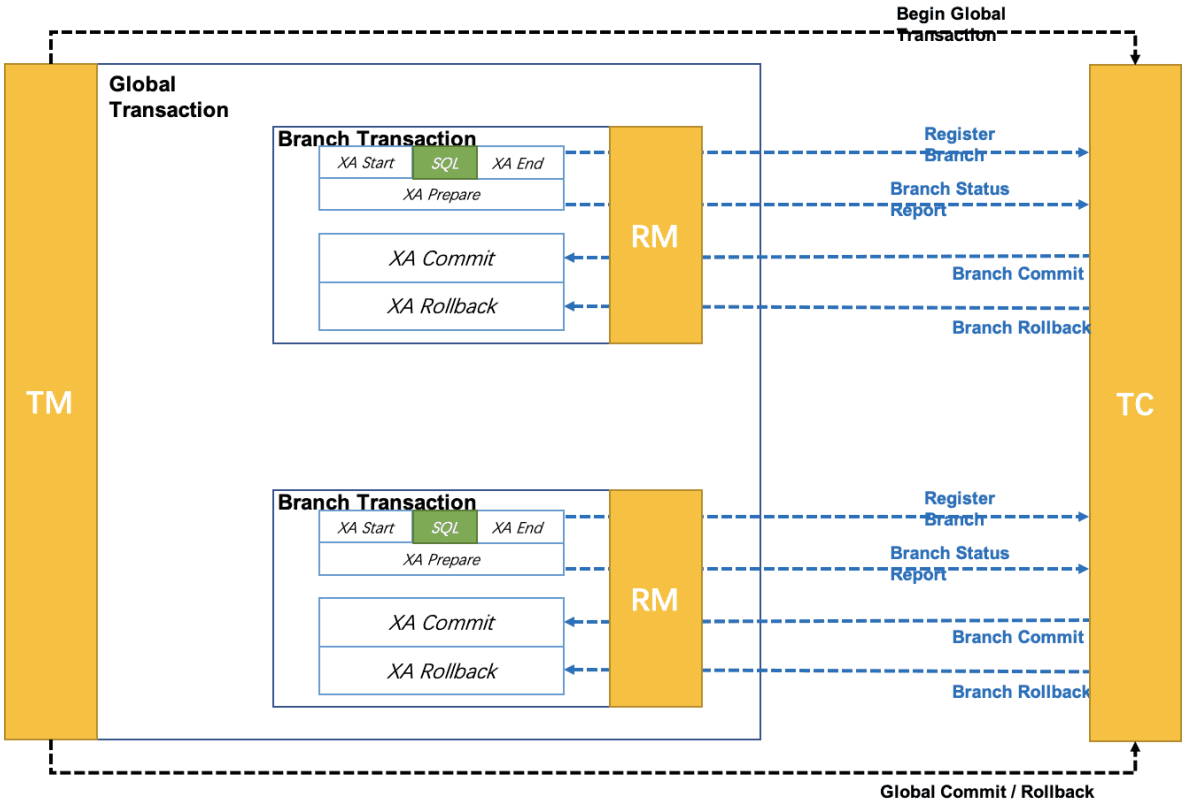
5.2 Seata XA 模式

5.2.1 前提

- 支持XA 事务的数据库。
- Java 应用，通过 JDBC 访问数据库。

5.2.2 整体机制

在 Seata 定义的分布式事务框架内，利用事务资源（数据库、消息服务等）对 XA 协议的支持，以 XA 协议的机制来管理分支事务的一种 事务模式。



执行阶段:

1. 可回滚：业务 SQL 操作放在 XA 分支中进行，由资源对 XA 协议的支持来保证 可回滚
2. 持久化：XA 分支完成后，执行 XA prepare，同样，由资源对 XA 协议的支持来保证 持久化（即，之后任何意外都不会造成无法回滚的情况）

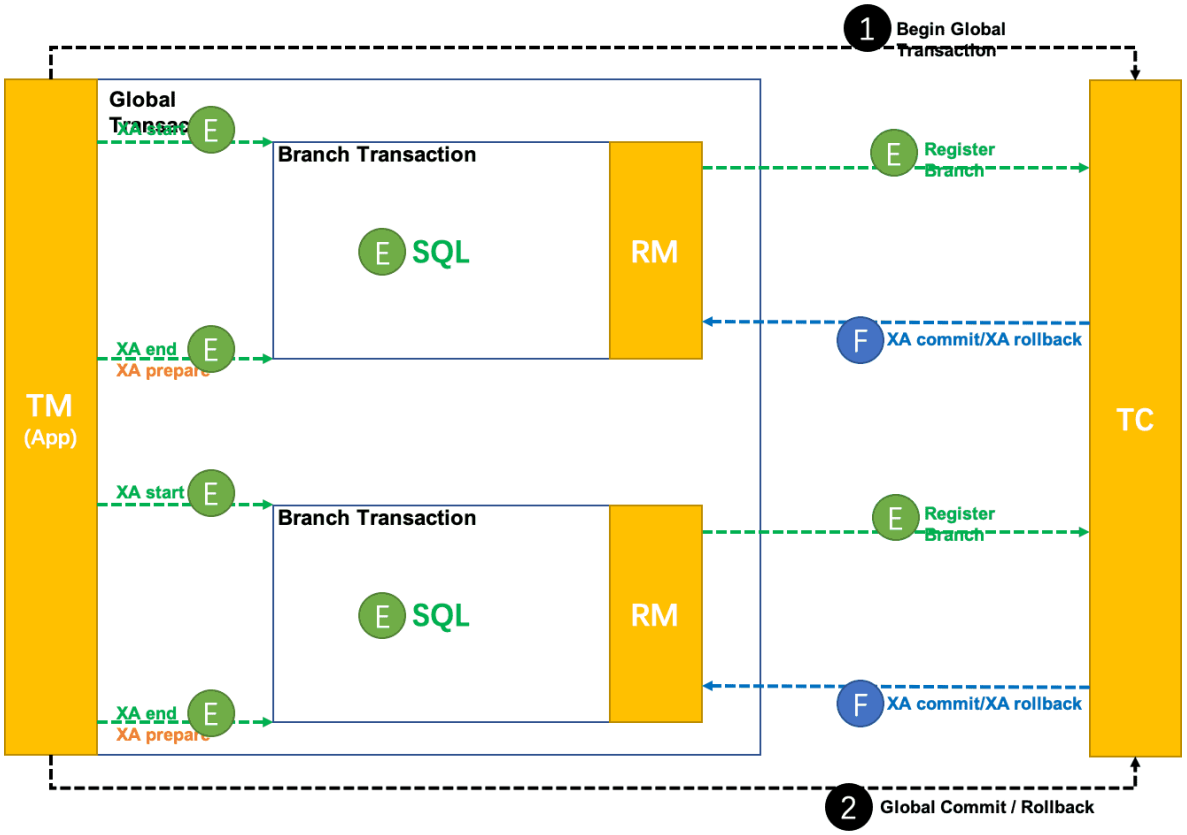
完成阶段:

1. 分支提交：执行 XA 分支的 commit
2. 分支回滚：执行 XA 分支的 rollback

5.2.3 工作机制

1. **整体运行机制**

XA 模式 运行在 Seata 定义的事务框架内:



- 执行阶段（E xecute）： XA start/XA end/XA prepare + SQL + 注册分支
- 完成阶段（F inish）：XA commit/XA rollback

2. **数据源代理**

XA 模式需要 XAConnection。

获取 XAConnection 两种方式：

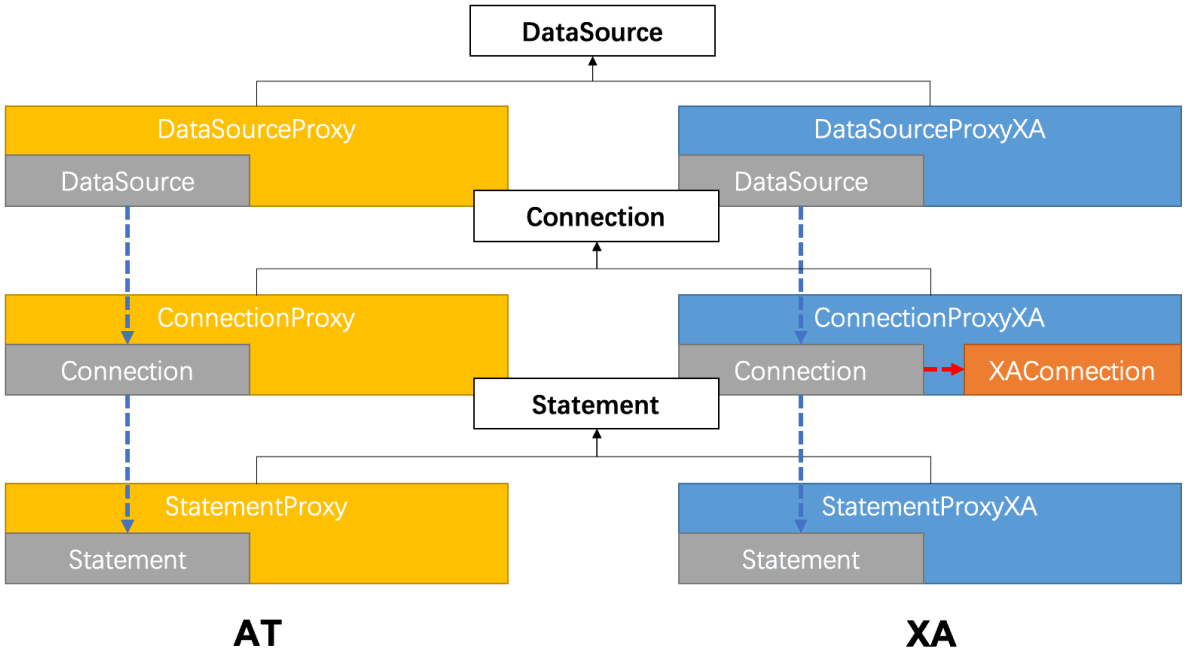
- 方式一：要求开发者配置 XADataSource
- 方式二：根据开发者的普通 DataSource 来创建

第一种方式，给开发者增加了认知负担，需要为 XA 模式专门去学习和使用 XA 数据源，与 透明化 XA 编程模型的设计目标相违背。

第二种方式，对开发者比较友好，和 AT 模式使用一样，开发者完全不必关心 XA 层面的任何问题，保持本地编程模型即可。

我们优先设计实现第二种方式：数据源代理根据普通数据源中获取的普通 JDBC 连接创建出相应的 XAConnection。

类比 AT 模式的数据源代理机制，如下：



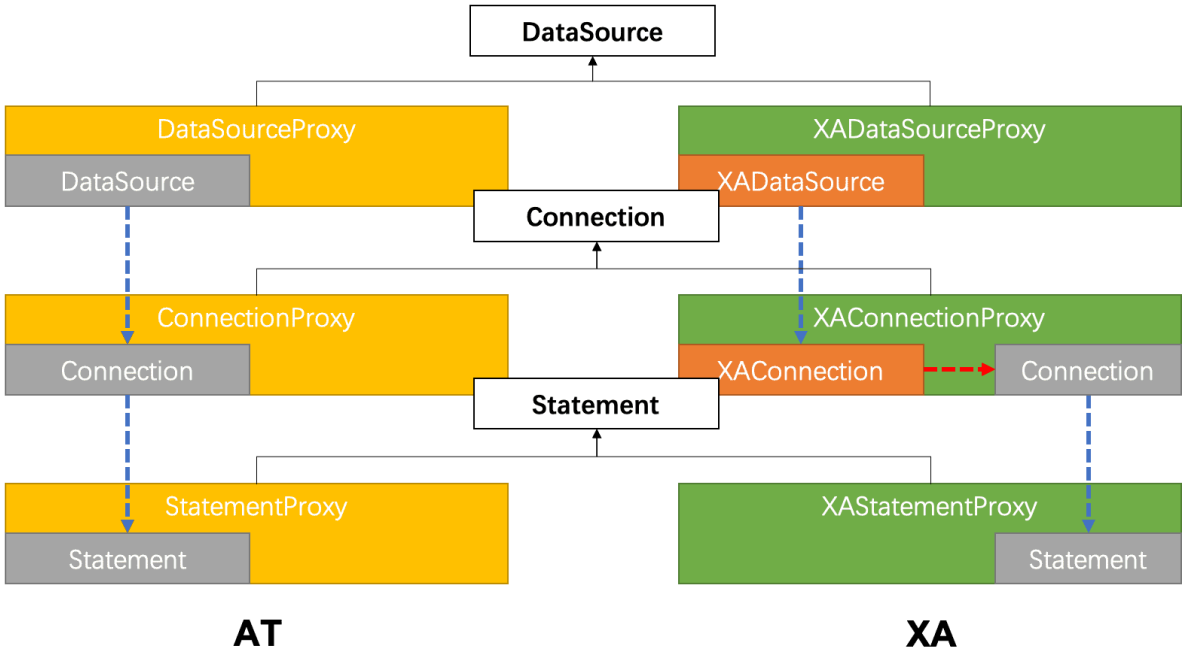
但是，第二种方法有局限：无法保证兼容的正确性。

实际上，这种方法是在做数据库驱动程序要做的事情。不同的厂商、不同版本的数据库驱动实现机制是厂商私有的，我们只能保证在充分测试过的驱动程序上是正确的，开发者使用的驱动程序版本差异很可能造成机制的失效。

这点在 Oracle 上体现非常明显。参见 Druid issue：
https://github.com/alibaba/druid/issues/3707

综合考虑，XA 模式的数据源代理设计需要同时支持第一种方式：基于 XA 数据源进行代理。

类比 AT 模式的数据源代理机制，如下：



3. **分支注册**

XA start 需要 Xid 参数。

这个 Xid 需要和 Seata 全局事务的 XID 和 BranchId 关联起来，以便由 TC 驱动 XA 分支的提交或回滚。

目前 Seata 的 BranchId 是在分支注册过程，由 TC 统一生成的，所以 XA 模式分支注册的时机需要在 XA start 之前。

将来一个可能的优化方向:

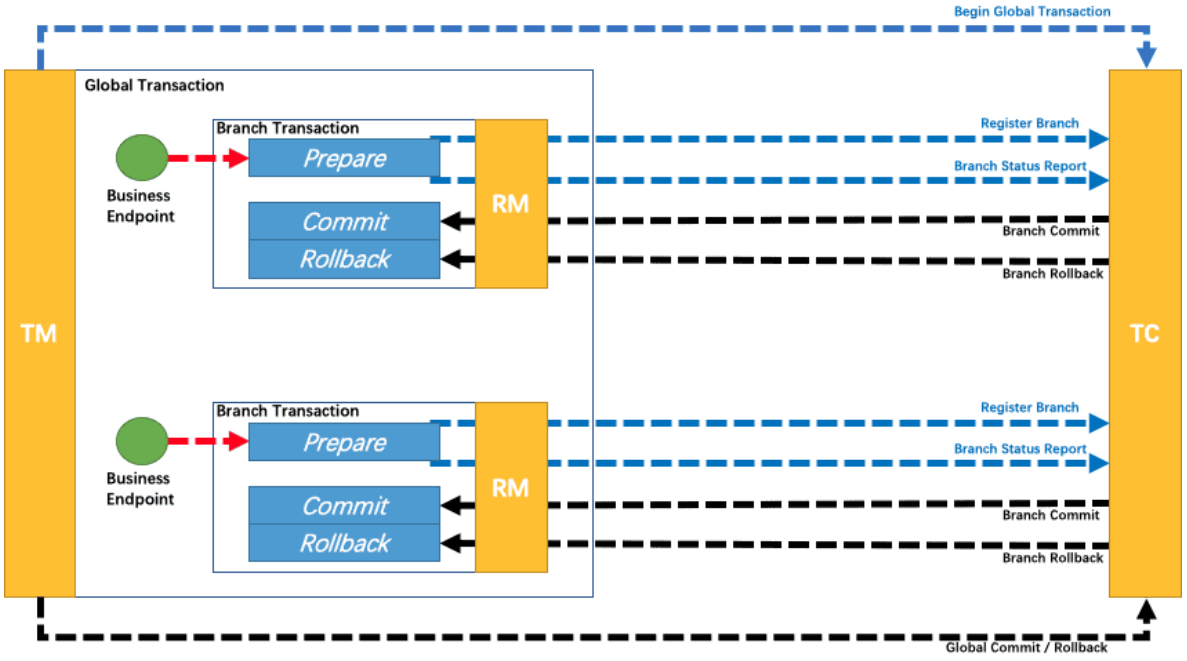
把分支注册尽量延后。类似 AT 模式在本地事务提交之前才注册分支，避免分支执行失败情况下，没有意义的分支注册。

这个优化方向需要 BranchId 生成机制的变化来配合。BranchId 不通过分支注册过程生成，而是生成后再带着 BranchId 去注册分支。

5.3 Seata TCC 模式

回顾总览中的描述：一个分布式的全局事务，整体是 两阶段提交 的模型。全局事务是由若干分支事务组成的，分支事务要满足 两阶段提交 的模型要求，即需要每个分支事务都具备自己的：

- 一阶段 prepare 行为
- 二阶段 commit 或 rollback 行为



根据两阶段行为模式的不同，我们将分支事务划分为 Automatic (Branch) Transaction Mode 和 TCC (Branch) Transaction Mode.

AT 模式基于 支持本地 ACID 事务 的 关系型数据库：

1. 一阶段 prepare 行为：在本地事务中，一并提交业务数据更新和相应回滚日志记录。
2. 二阶段 commit 行为：马上成功结束，自动 异步批量清理回滚日志。
3. 二阶段 rollback 行为：通过回滚日志，自动 生成补偿操作，完成数据回滚。

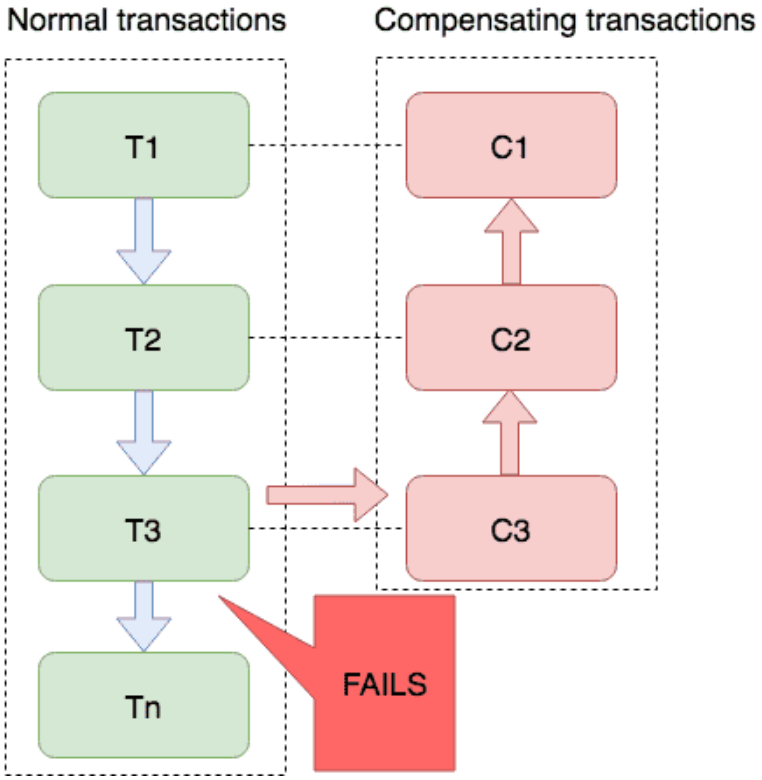
相应的，TCC 模式，不依赖于底层数据资源的事务支持：

1. 一阶段 prepare 行为：调用 自定义 的 prepare 逻辑。
2. 二阶段 commit 行为：调用 自定义 的 commit 逻辑。
3. 二阶段 rollback 行为：调用 自定义 的 rollback 逻辑。

所谓 TCC 模式，是指支持把 自定义 的分支事务纳入到全局事务的管理中。

5.4 Seata Saga 模式

> Saga模式是SEATA提供的长事务解决方案，在Saga模式中，业务流程中每个参与者都提交本地事务，当出现某一个参与者失败则补偿前面已经成功的参与者，一阶段正向服务和二阶段补偿服务都由业务开发实现。



5.4.1 概述

适用场景:

1. 业务流程长、业务流程多
2. 参与者包含其它公司或遗留系统服务，无法提供 TCC 模式要求的三个接口

优势:

1. 一阶段提交本地事务，无锁，高性能
2. 事件驱动架构，参与者可异步执行，高吞吐
3. 补偿服务易于实现

缺点:

1. 不保证隔离性（应对方案见后面文档）

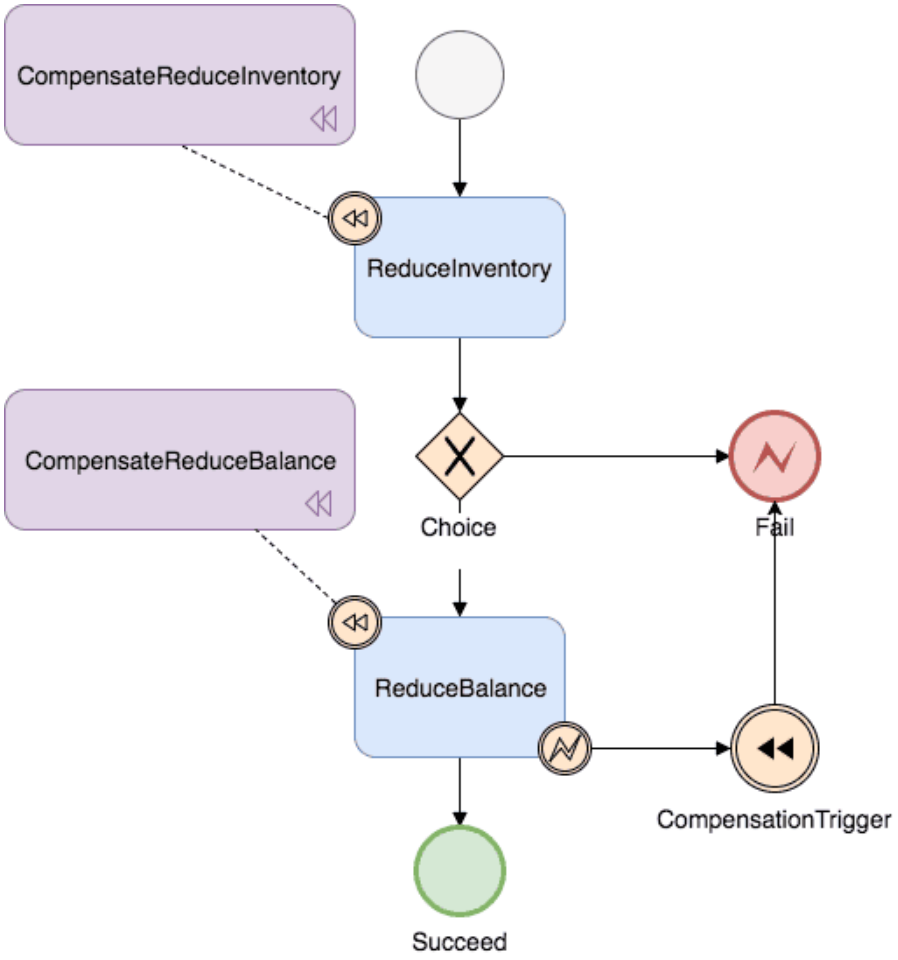
5.4.2 Saga的实现

目前SEATA提供的Saga模式是基于状态机引擎来实现的，机制是:

1. 通过状态图来定义服务调用的流程并生成 json 状态语言定义文件
2. 状态图中一个节点可以是调用一个服务，节点可以配置它的补偿节点

3. 状态图 json 由状态机引擎驱动执行，当出现异常时状态引擎反向执行已成功节点对应的补偿节点将事务回滚
    1. 注意: 异常发生时是否进行补偿也可由用户自定义决定
4. 可以实现服务编排需求，支持单项选择、并发、子流程、参数转换、参数映射、服务执行状态判断、异常捕获等功能

示例状态图:



6. 参考文章

- http://www.dockone.io/article/9903
- https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/
- https://blog.csdn.net/a745233700/article/details/122402303

- https://seata.io/zh-cn/docs/overview/what-is-seata.html
- https://www.cnblogs.com/myseries/p/10939355.html
- https://juejin.im/post/6871435457893728263
- https://juejin.im/post/6844903734753886216