**常用:补偿事务TCC 和 SAGA 模式 :** <https://www.jianshu.com/p/b0f73cfb65cf>

XA分布式事务协议，包含二阶段提交（2PC），三阶段提交（3PC）两种实现。

**1、二阶段提交方案：强一致性**

事务的发起者称协调者，事务的执行者称参与者。

处理流程：

　　1、准备阶段

　　　　事务协调者，向所有事务参与者发送事务内容，询问是否可以提交事务，并等待参与者回复。

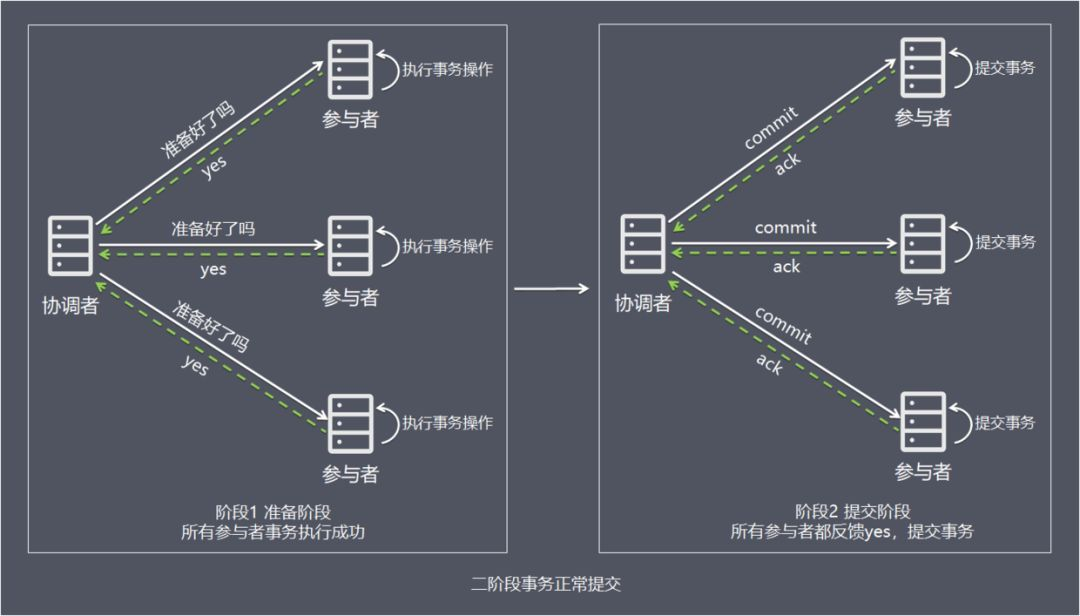
　　　　事务参与者收到事务内容，开始执行事务操作，讲 undo 和 redo  信息记入事务日志中（但此时并不提交事务）。

　　　　如果参与者执行成功，给协调者回复yes,表示可以进行事务提交。如果执行失败，给协调者回复no,表示不可提交。

　　2、提交阶段

　　　　如果协调者收到了参与者的失败信息或超时信息，直接给所有参与者发送回滚（rollback）信息进行事务回滚，否则，发送提交（commit）信息。

　　　　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) 接下来分两种情况分别讨论提交阶段的过程。



简单一点理解，可以把协调者节点比喻为带头大哥，参与者理解比喻为跟班小弟，带头大哥统一协调跟班小弟的任务执行。

阶段 1：准备阶段

准备阶段有如下三个步骤：

* 协调者向所有参与者发送事务内容，询问是否可以提交事务，并等待所有参与者答复。
* 各参与者执行事务操作，将 undo 和 redo 信息记入事务日志中（但不提交事务）。
* 如参与者执行成功，给协调者反馈 yes，即可以提交；如执行失败，给协调者反馈 no，即不可提交。

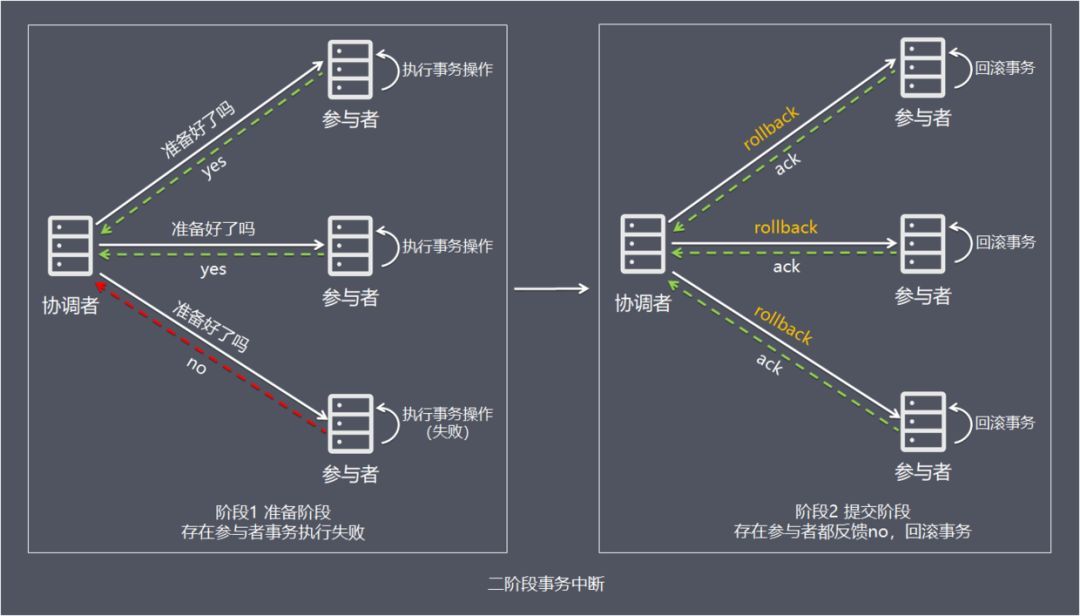
阶段 2：提交阶段

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) 接下来分两种情况分别讨论提交阶段的过程。

情况 1，当所有参与者均反馈 yes，提交事务，如上图：

* 协调者向所有参与者发出正式提交事务的请求（即 commit 请求）。
* 参与者执行 commit 请求，并释放整个事务期间占用的资源。
* 各参与者向协调者反馈 ack(应答)完成的消息。
* 协调者收到所有参与者反馈的 ack 消息后，即完成事务提交。



情况 2，当任何阶段 1 一个参与者反馈 no，中断事务，如上图：

* 协调者向所有参与者发出回滚请求（即 rollback 请求）。
* 参与者使用阶段 1 中的 undo 信息执行回滚操作，并释放整个事务期间占用的资源。
* 各参与者向协调者反馈 ack 完成的消息。
* 协调者收到所有参与者反馈的 ack 消息后，即完成事务中断。

方案总结

2PC 方案实现起来简单，实际项目中使用比较少，主要因为以下问题：

* 性能问题：所有参与者在事务提交阶段处于同步阻塞状态，占用系统资源，容易导致性能瓶颈。
* 可靠性问题：如果协调者存在单点故障问题，如果协调者出现故障，参与者将一直处于锁定状态。
* 数据一致性问题：在阶段 2 中，如果发生局部网络问题，一部分事务参与者收到了提交消息，另一部分事务参与者没收到提交消息，那么就导致了节点之间数据的不一致。

**2、3阶段提交**

　三阶段提交是在二阶段提交上的改进版本，主要是加入了超时机制。同时在协调者和参与者中都引入超时机制。

　　三阶段将二阶段的准备阶段拆分为2个阶段，插入了一个preCommit阶段，以此来处理原先二阶段，参与者准备后，参与者发生崩溃或错误，导致参与者无法知晓是否提交或回滚的不确定状态所引起的延时问题。

处理流程

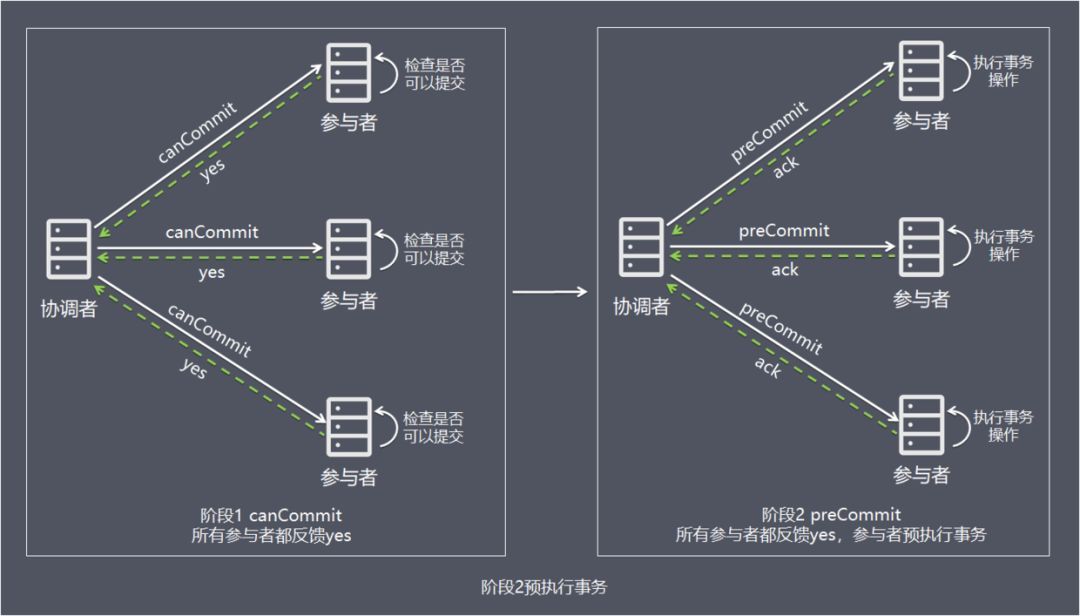
阶段 1：canCommit

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(参与者不执行事务操作)，否则返回 no 响应：

* 协调者向所有参与者发出包含事务内容的 canCommit 请求，询问是否可以提交事务，并等待所有参与者答复。
* 参与者收到 canCommit 请求后，如果认为可以执行事务操作，则反馈 yes 并进入预备状态，否则反馈 no。

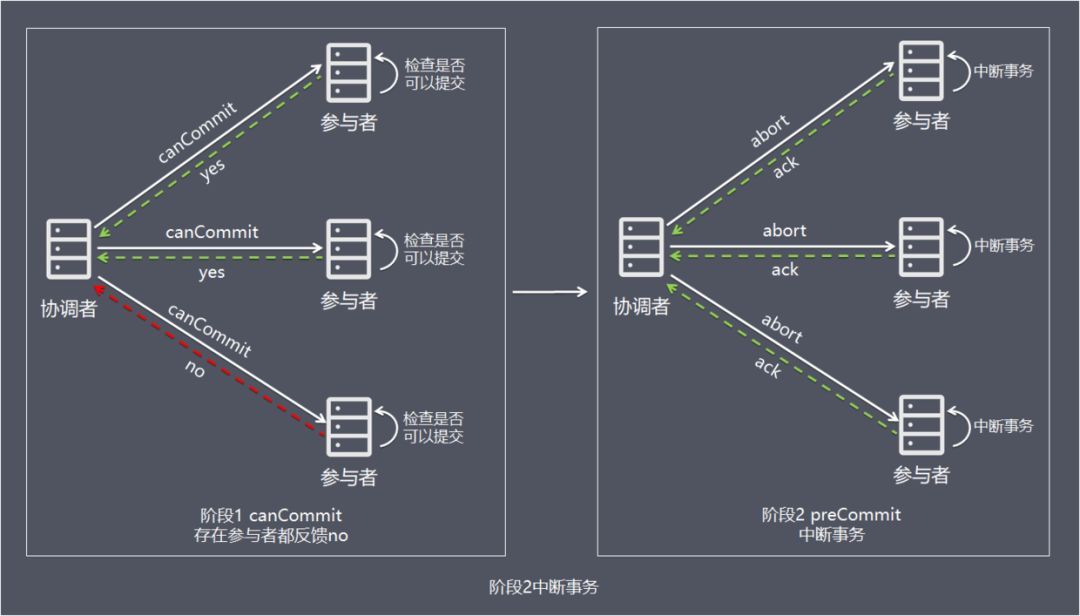
阶段 2：preCommit

协调者根据阶段 1 canCommit 参与者的反应情况来决定是否可以进行基于事务的 preCommit 操作。根据响应情况，有以下两种可能。



情况 1：阶段 1 所有参与者均反馈 yes，参与者预执行事务，如上图：

* 协调者向所有参与者发出 preCommit 请求，进入准备阶段。
* 参与者收到 preCommit 请求后，执行事务操作，将 undo 和 redo 信息记入事务日志中（但不提交事务）。
* 各参与者向协调者反馈 ack 响应或 no 响应，并等待最终指令。

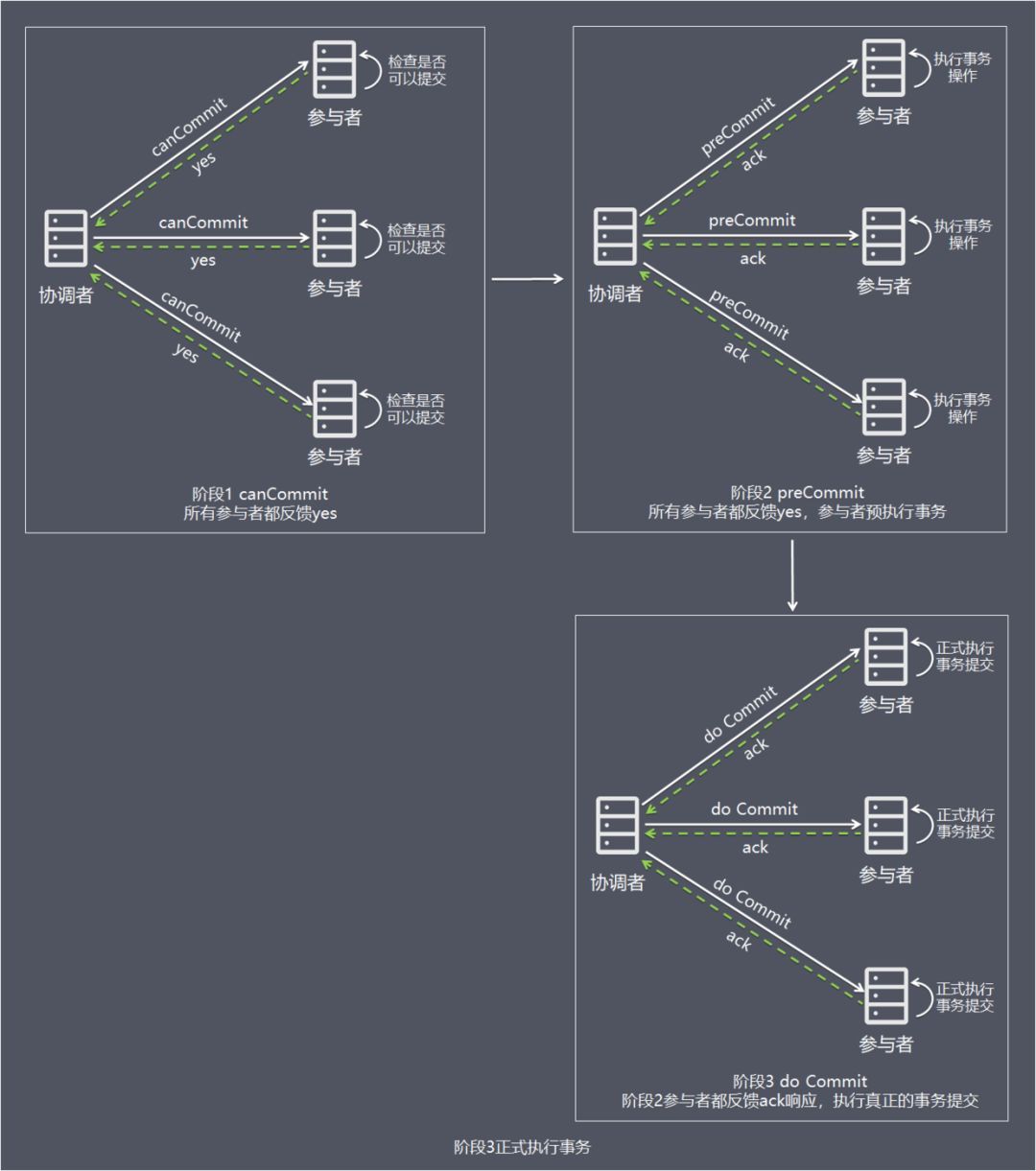


情况 2：阶段 1 任何一个参与者反馈 no，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务，如上图：

* 协调者向所有参与者发出 abort 请求。
* 无论收到协调者发出的 abort 请求，或者在等待协调者请求过程中出现超时，参与者均会中断事务。

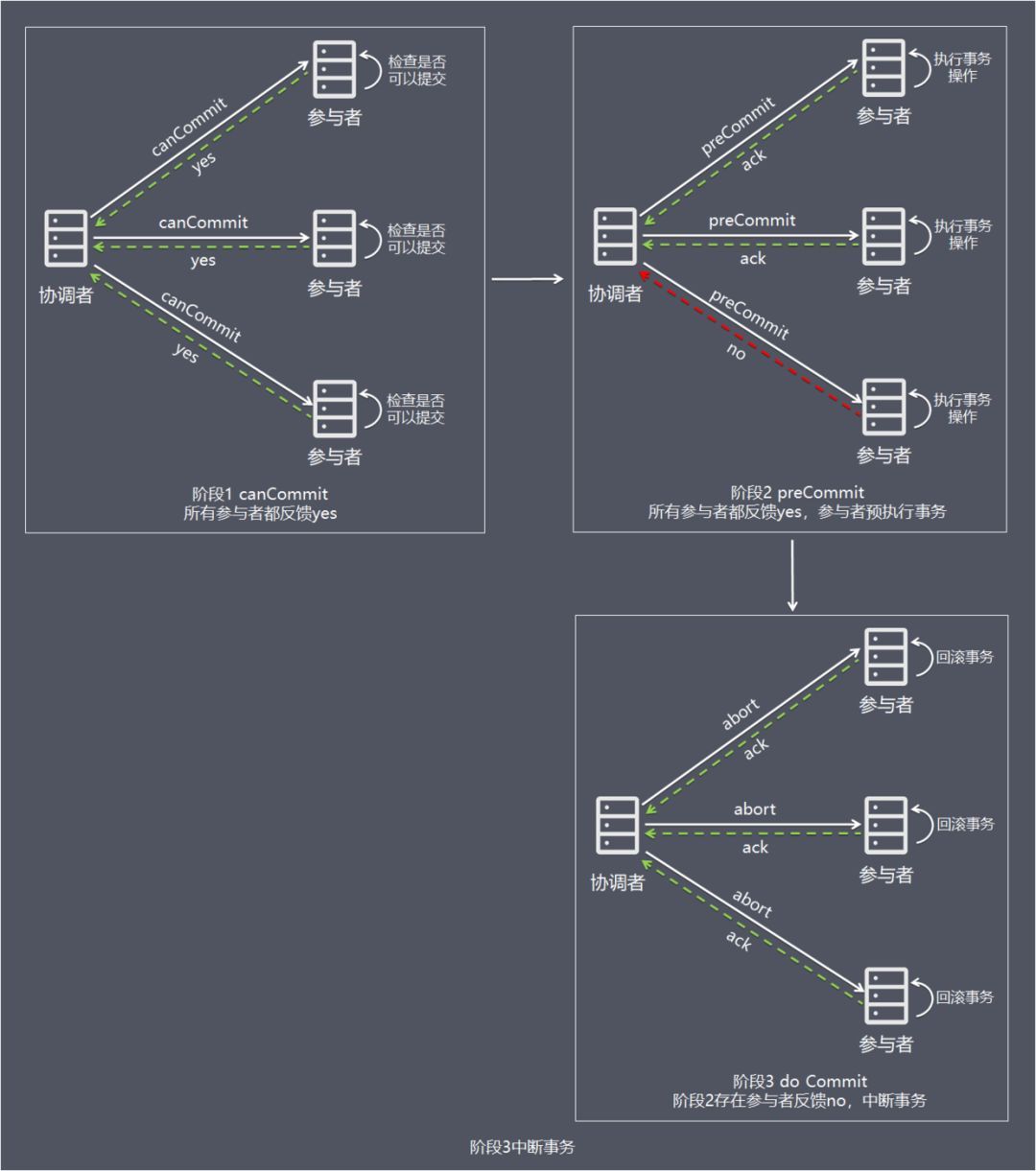
阶段 3：do Commit

该阶段进行真正的事务提交，也可以分为以下两种情况。



情况 1：阶段 2 所有参与者均反馈 ack 响应，执行真正的事务提交，如上图：

* 如果协调者处于工作状态，则向所有参与者发出 do Commit 请求。
* 参与者收到 do Commit 请求后，会正式执行事务提交，并释放整个事务期间占用的资源。
* 各参与者向协调者反馈 ack 完成的消息。
* 协调者收到所有参与者反馈的 ack 消息后，即完成事务提交。



情况 2：阶段 2 任何一个参与者反馈 no，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务，如上图：

* 如果协调者处于工作状态，向所有参与者发出 abort 请求。
* 参与者使用阶段 1 中的 undo 信息执行回滚操作，并释放整个事务期间占用的资源。
* 各参与者向协调者反馈 ack 完成的消息。
* 协调者收到所有参与者反馈的 ack 消息后，即完成事务中断。

注意：进入阶段 3 后，无论协调者出现问题，或者协调者与参与者网络出现问题，都会导致参与者无法接收到协调者发出的 do Commit 请求或 abort 请求。此时，参与者都会在等待超时之后，继续执行事务提交。

方案总结

优点：相比二阶段提交，三阶段提交降低了阻塞范围，在等待超时后协调者或参与者会中断事务。避免了协调者单点问题，阶段 3 中协调者出现问题时，参与者会继续提交事务。

缺点：数据不一致问题依然存在，当在参与者收到 preCommit 请求后等待 do commite 指令时，此时如果协调者请求中断事务，而协调者无法与参与者正常通信，会导致参与者继续提交事务，造成数据不一致。

TCC 事务：最终一致性

方案简介

TCC（Try-Confirm-Cancel）的概念，最早是由 Pat Helland 于 2007 年发表的一篇名为《Life beyond Distributed Transactions:an Apostate’s Opinion》的论文提出。

TCC 是服务化的二阶段编程模型，其 Try、Confirm、Cancel 3 个方法均由业务编码实现：

* Try 操作作为一阶段，负责资源的检查和预留。
* Confirm 操作作为二阶段提交操作，执行真正的业务。
* Cancel 是预留资源的取消。

TCC 事务的 Try、Confirm、Cancel 可以理解为 SQL 事务中的 Lock、Commit、Rollback。

处理流程

为了方便理解，下面以电商下单为例进行方案解析，这里把整个过程简单分为扣减库存，订单创建 2 个步骤，库存服务和订单服务分别在不同的服务器节点上。

①Try 阶段

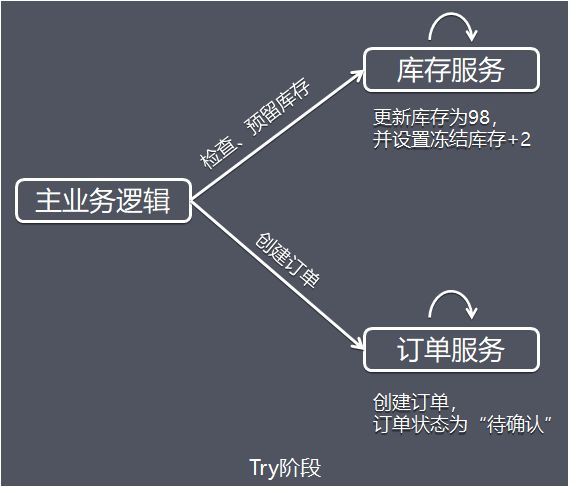
从执行阶段来看，与传统事务机制中业务逻辑相同。但从业务角度来看，却不一样。

TCC 机制中的 Try 仅是一个初步操作，它和后续的确认一起才能真正构成一个完整的业务逻辑，这个阶段主要完成：

* 完成所有业务检查( 一致性 ) 。
* 预留必须业务资源( 准隔离性 ) 。
* Try 尝试执行业务。

TCC 事务机制以初步操作（Try）为中心的，确认操作（Confirm）和取消操作（Cancel）都是围绕初步操作（Try）而展开。

因此，Try 阶段中的操作，其保障性是最好的，即使失败，仍然有取消操作（Cancel）可以将其执行结果撤销。



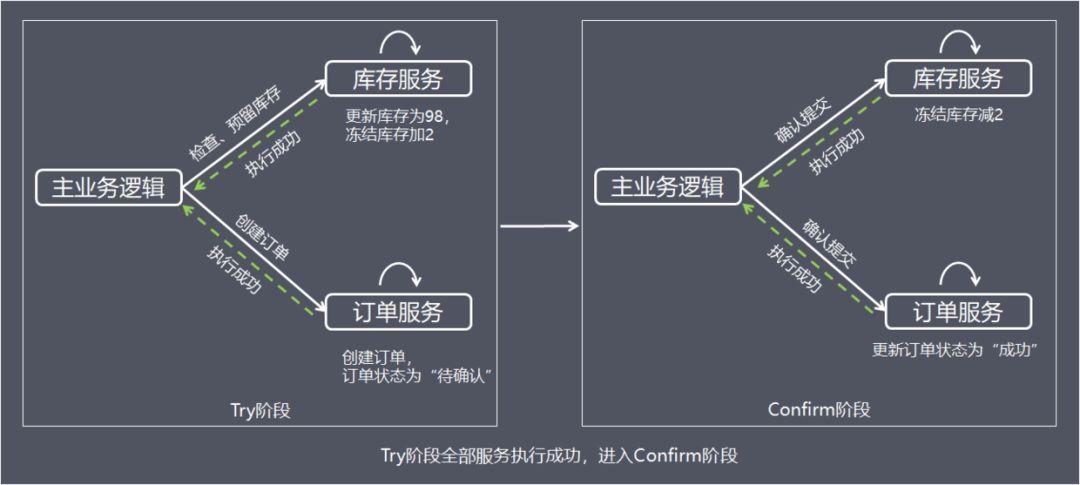
假设商品库存为 100，购买数量为 2，这里检查和更新库存的同时，冻结用户购买数量的库存，同时创建订单，订单状态为待确认。

②Confirm / Cancel 阶段

根据 Try 阶段服务是否全部正常执行，继续执行确认操作（Confirm）或取消操作（Cancel）。

Confirm 和 Cancel 操作满足幂等性，如果 Confirm 或 Cancel 操作执行失败，将会不断重试直到执行完成。

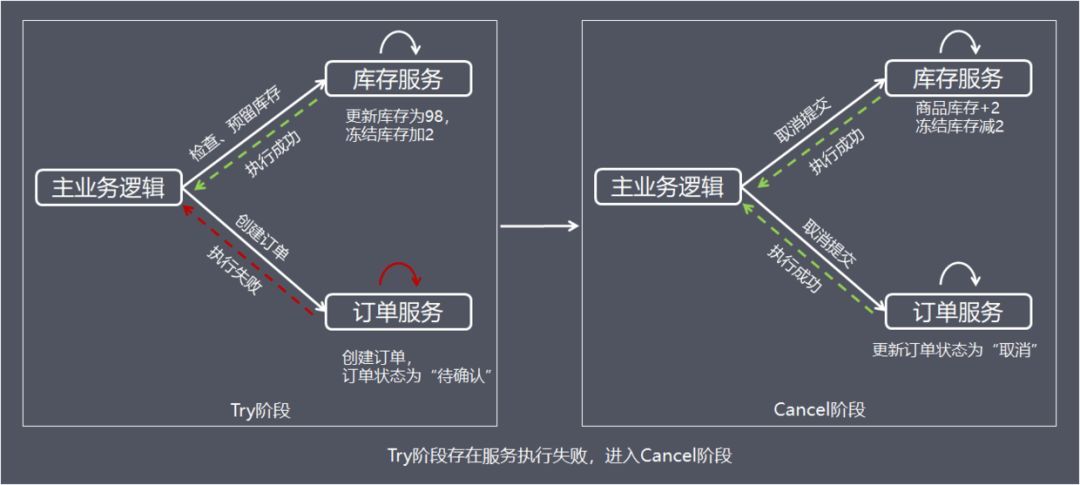
Confirm：当 Try 阶段服务全部正常执行， 执行确认业务逻辑操作



这里使用的资源一定是 Try 阶段预留的业务资源。在 TCC 事务机制中认为，如果在 Try 阶段能正常的预留资源，那 Confirm 一定能完整正确的提交。

Confirm 阶段也可以看成是对 Try 阶段的一个补充，Try+Confirm 一起组成了一个完整的业务逻辑。

Cancel：当 Try 阶段存在服务执行失败， 进入 Cancel 阶段



Cancel 取消执行，释放 Try 阶段预留的业务资源，上面的例子中，Cancel 操作会把冻结的库存释放，并更新订单状态为取消。

方案总结

TCC 事务机制相对于传统事务机制（X/Open XA），TCC 事务机制相比于上面介绍的 XA 事务机制，有以下优点：

* 性能提升：具体业务来实现控制资源锁的粒度变小，不会锁定整个资源。
* 数据最终一致性：基于 Confirm 和 Cancel 的幂等性，保证事务最终完成确认或者取消，保证数据的一致性。
* 可靠性：解决了 XA 协议的协调者单点故障问题，由主业务方发起并控制整个业务活动，业务活动管理器也变成多点，引入集群。

缺点： TCC 的 Try、Confirm 和 Cancel 操作功能要按具体业务来实现，业务耦合度较高，提高了开发成本。

本地消息表：最终一致性

方案简介

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。

方案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。

这样设计可以避免”业务处理成功 + 事务消息发送失败"，或"业务处理失败 + 事务消息发送成功"的棘手情况出现，保证 2 个系统事务的数据一致性。

处理流程

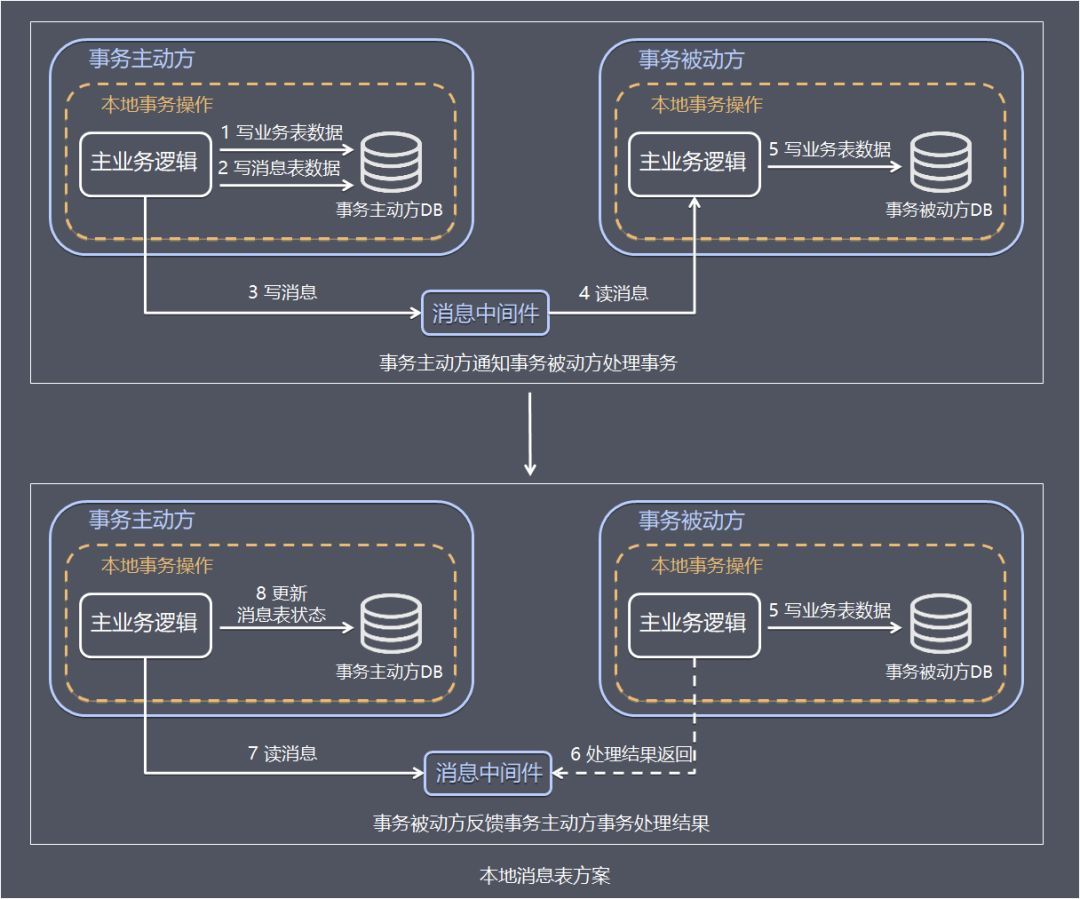
下面把分布式事务最先开始处理的事务方称为事务主动方，在事务主动方之后处理的业务内的其他事务称为事务被动方。

为了方便理解，下面继续以电商下单为例进行方案解析，这里把整个过程简单分为扣减库存，订单创建 2 个步骤。

库存服务和订单服务分别在不同的服务器节点上，其中库存服务是事务主动方，订单服务是事务被动方。

事务的主动方需要额外新建事务消息表，用于记录分布式事务的消息的发生、处理状态。

整个业务处理流程如下：



步骤1：事务主动方处理本地事务。

事务主动方在本地事务中处理业务更新操作和写消息表操作。上面例子中库存服务阶段在本地事务中完成扣减库存和写消息表(图中 1、2)。

步骤 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。

上面例子中，库存服务把事务待处理消息写到消息中间件，订单服务消费消息中间件的消息，完成新增订单（图中 3 - 5）。

步骤 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。

上面例子中，订单服务把事务已处理消息写到消息中间件，库存服务消费中间件的消息，并将事务消息的状态更新为已完成(图中 6 - 8)。

为了数据的一致性，当处理错误需要重试，事务发送方和事务接收方相关业务处理需要支持幂等。

具体保存一致性的容错处理如下：

* 当步骤 1 处理出错，事务回滚，相当于什么都没发生。
* 当步骤 2、步骤 3 处理出错，由于未处理的事务消息还是保存在事务发送方，事务发送方可以定时轮询为超时消息数据，再次发送到消息中间件进行处理。事务被动方消费事务消息重试处理。
* 如果是业务上的失败，事务被动方可以发消息给事务主动方进行回滚。
* 如果多个事务被动方已经消费消息，事务主动方需要回滚事务时需要通知事务被动方回滚。

方案总结

方案的优点如下：

* 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。

缺点如下：

* 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。

MQ 事务：最终一致性

方案简介

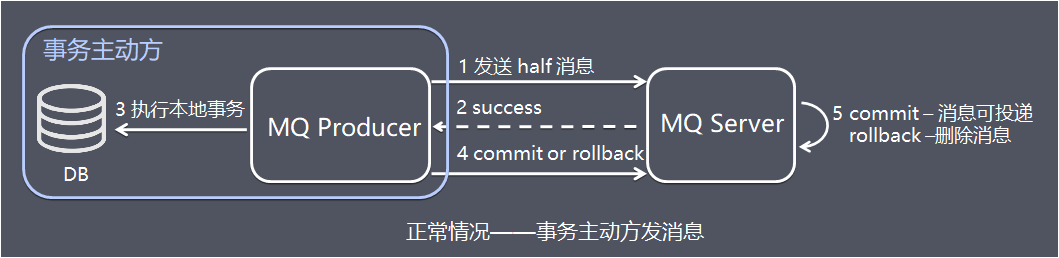
基于 MQ 的分布式事务方案其实是对本地消息表的封装，将本地消息表基于 MQ 内部，其他方面的协议基本与本地消息表一致。

处理流程

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，相对于提供了 2PC 的提交接口，方案如下：

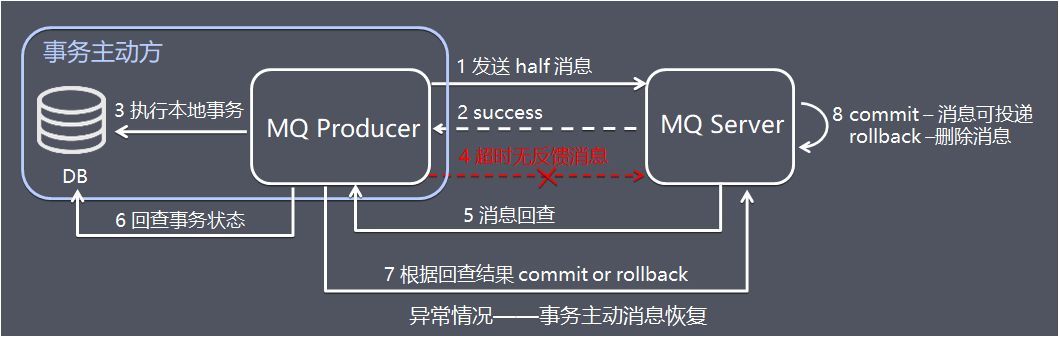
正常情况：事务主动方发消息



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：

* 图中 1：发送方向 MQ 服务端(MQ Server)发送 half 消息。
* 图中 2：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。
* 图中 3：发送方开始执行本地事务逻辑。
* 图中 4：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）。
* 图中 5：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。

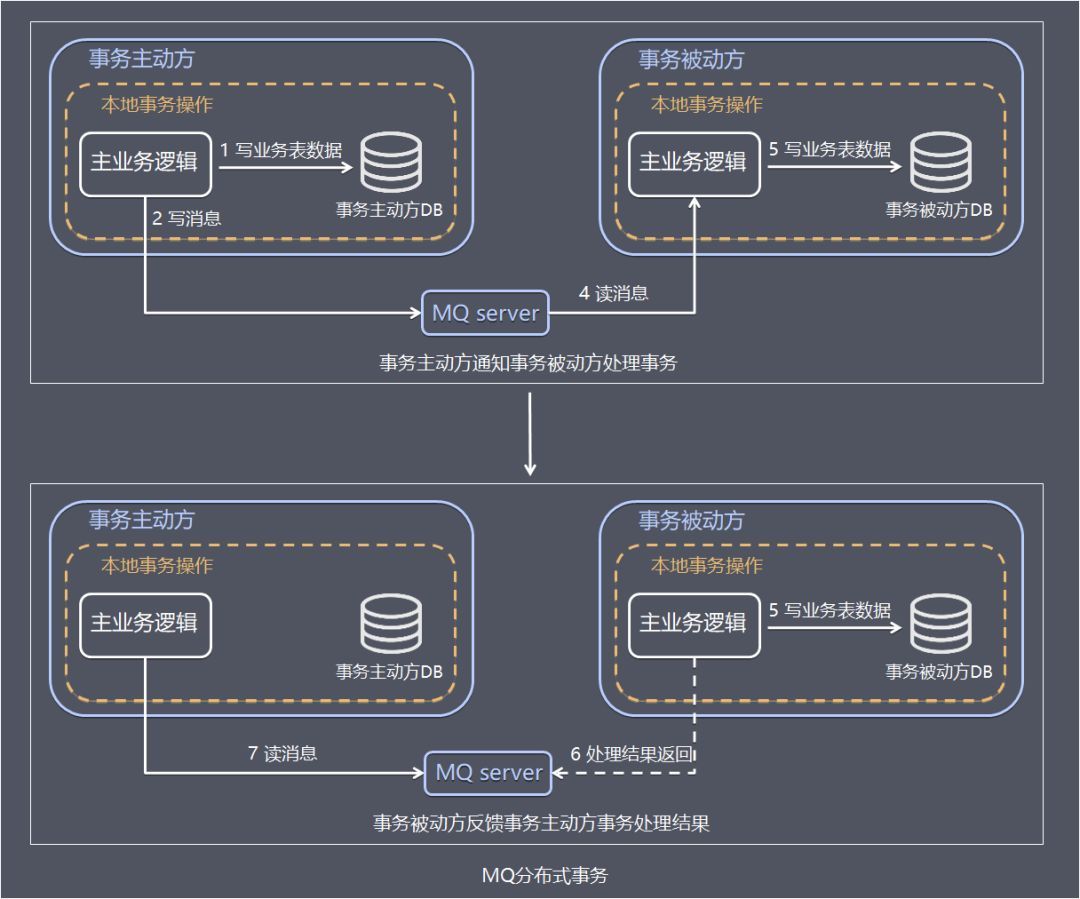
异常情况：事务主动方消息恢复



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：

* 图中 5：MQ Server 对该消息发起消息回查。
* 图中 6：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。
* 图中 7：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。
* 图中 8：MQ Server基于 commit/rollback 对消息进行投递或者删除。

介绍完 RocketMQ 的事务消息方案后，由于前面已经介绍过本地消息表方案，这里就简单介绍 RocketMQ 分布式事务：



事务主动方基于 MQ 通信通知事务被动方处理事务，事务被动方基于 MQ 返回处理结果。

如果事务被动方消费消息异常，需要不断重试，业务处理逻辑需要保证幂等。

如果是事务被动方业务上的处理失败，可以通过 MQ 通知事务主动方进行补偿或者事务回滚。

方案总结

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。
* 吞吐量由于使用本地消息表方案。

缺点是：

* 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。

Saga 事务：最终一致性

方案简介

Saga 事务源于 1987 年普林斯顿大学的 Hecto 和 Kenneth 发表的如何处理 long lived transaction（长活事务）论文。

Saga 事务核心思想是将长事务拆分为多个本地短事务，由 Saga 事务协调器协调，如果正常结束那就正常完成，如果某个步骤失败，则根据相反顺序一次调用补偿操作。

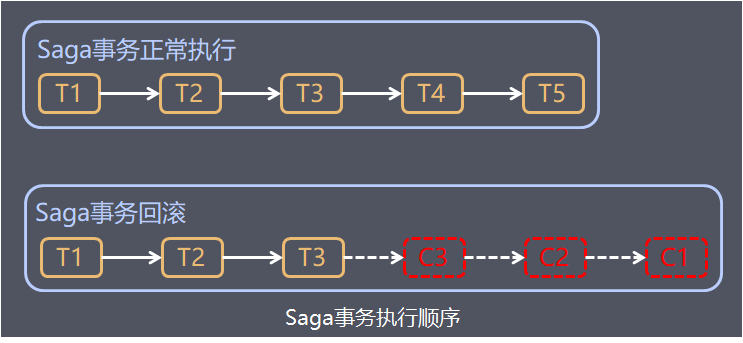
处理流程

Saga 事务基本协议如下：

* 每个 Saga 事务由一系列幂等的有序子事务(sub-transaction) Ti 组成。
* 每个 Ti 都有对应的幂等补偿动作 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。

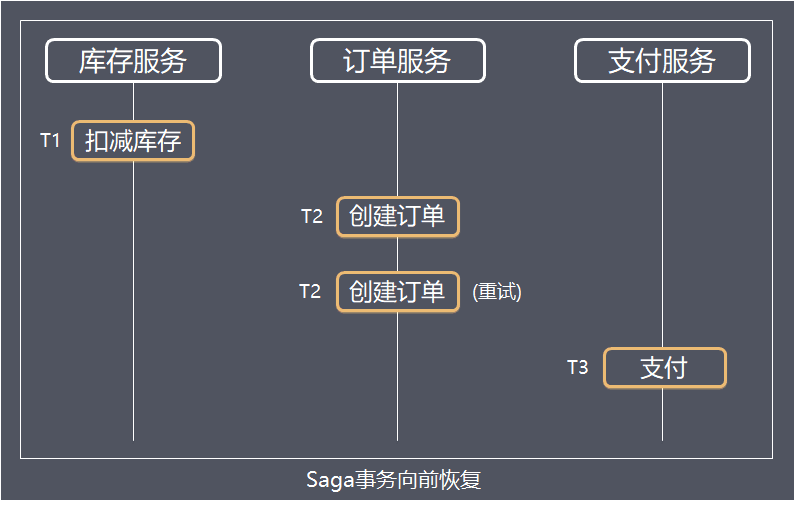
下面以下单流程为例，整个操作包括：创建订单、扣减库存、支付、增加积分。



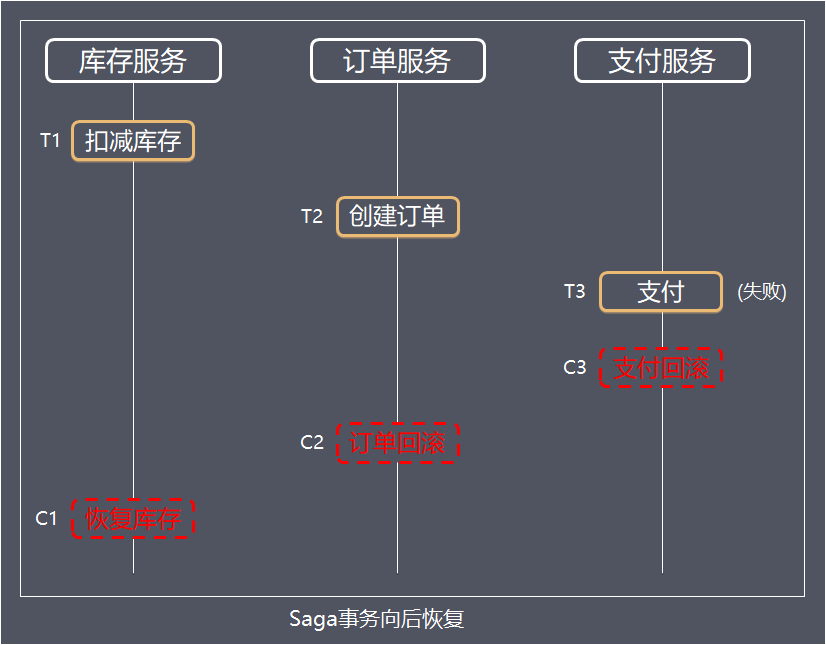
Saga 的执行顺序有两种，如上图：

* 事务正常执行完成：T1, T2, T3, ..., Tn，例如：扣减库存(T1)，创建订单(T2)，支付(T3)，依次有序完成整个事务。
* 事务回滚：T1, T2, ..., Tj, Cj,..., C2, C1，其中 0 < j < n，例如：扣减库存(T1)，创建订单(T2)，支付(T3，支付失败)，支付回滚(C3)，订单回滚(C2)，恢复库存(C1)。

Saga 定义了两种恢复策略：



向前恢复（forward recovery）：对应于上面第一种执行顺序，适用于必须要成功的场景，发生失败进行重试，执行顺序是类似于这样的：T1, T2, ..., Tj(失败), Tj(重试),..., Tn，其中j是发生错误的子事务(sub-transaction)。该情况下不需要Ci。

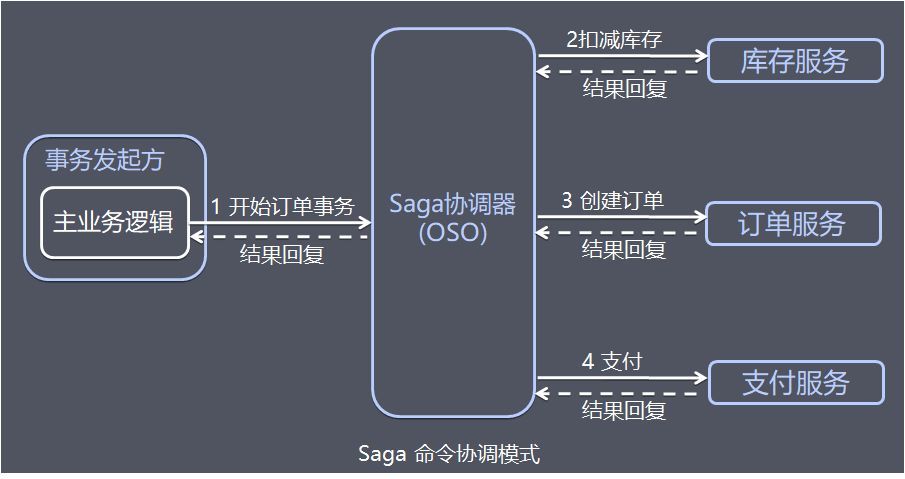


向后恢复（backward recovery）：对应于上面提到的第二种执行顺序，其中 j 是发生错误的子事务(sub-transaction)，这种做法的效果是撤销掉之前所有成功的子事务，使得整个 Saga 的执行结果撤销。

Saga 事务常见的有两种不同的实现方式：

①命令协调（Order Orchestrator）：中央协调器负责集中处理事件的决策和业务逻辑排序。

中央协调器（Orchestrator，简称 OSO）以命令/回复的方式与每项服务进行通信，全权负责告诉每个参与者该做什么以及什么时候该做什么。



以电商订单的例子为例：

* 事务发起方的主业务逻辑请求 OSO 服务开启订单事务
* OSO 向库存服务请求扣减库存，库存服务回复处理结果。
* OSO 向订单服务请求创建订单，订单服务回复创建结果。
* OSO 向支付服务请求支付，支付服务回复处理结果。
* 主业务逻辑接收并处理 OSO 事务处理结果回复。

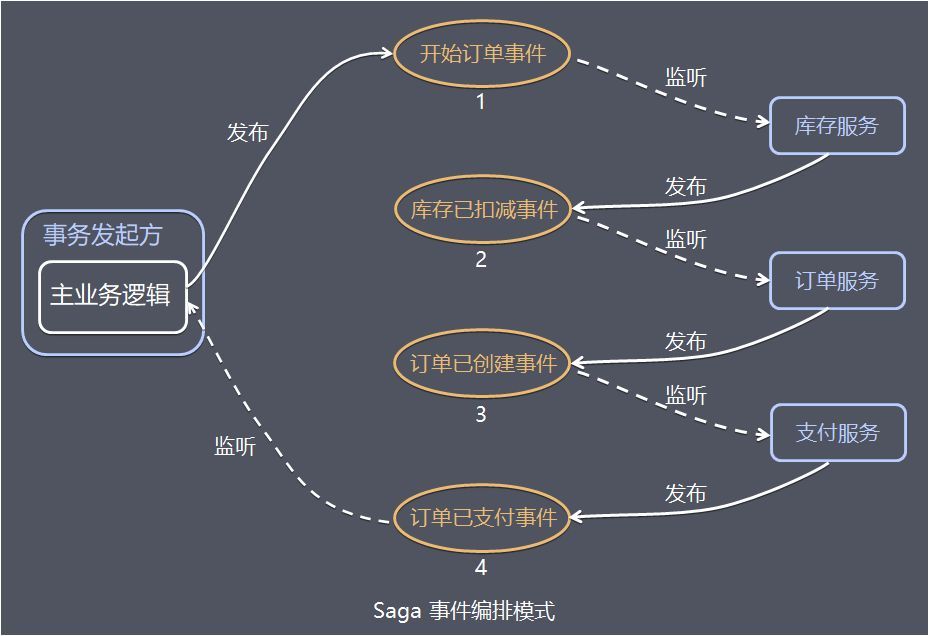
中央协调器必须事先知道执行整个订单事务所需的流程(例如通过读取配置)。如果有任何失败，它还负责通过向每个参与者发送命令来撤销之前的操作来协调分布式的回滚。

基于中央协调器协调一切时，回滚要容易得多，因为协调器默认是执行正向流程，回滚时只要执行反向流程即可。

②事件编排（Event Choreography0）：没有中央协调器（没有单点风险）时，每个服务产生并观察其他服务的事件，并决定是否应采取行动。

在事件编排方法中，第一个服务执行一个事务，然后发布一个事件。该事件被一个或多个服务进行监听，这些服务再执行本地事务并发布（或不发布）新的事件。

当最后一个服务执行本地事务并且不发布任何事件时，意味着分布式事务结束，或者它发布的事件没有被任何 Saga 参与者听到都意味着事务结束。



以电商订单的例子为例：

* 事务发起方的主业务逻辑发布开始订单事件。
* 库存服务监听开始订单事件，扣减库存，并发布库存已扣减事件。
* 订单服务监听库存已扣减事件，创建订单，并发布订单已创建事件。
* 支付服务监听订单已创建事件，进行支付，并发布订单已支付事件。
* 主业务逻辑监听订单已支付事件并处理。

事件/编排是实现 Saga 模式的自然方式，它很简单，容易理解，不需要太多的代码来构建。如果事务涉及 2 至 4 个步骤，则可能是非常合适的。

方案总结

命令协调设计的优点如下：

* 服务之间关系简单，避免服务之间的循环依赖关系，因为 Saga 协调器会调用 Saga 参与者，但参与者不会调用协调器。
* 程序开发简单，只需要执行命令/回复(其实回复消息也是一种事件消息)，降低参与者的复杂性。
* 易维护扩展，在添加新步骤时，事务复杂性保持线性，回滚更容易管理，更容易实施和测试。

命令协调设计缺点如下：

* 中央协调器容易处理逻辑容易过于复杂，导致难以维护。
* 存在协调器单点故障风险。

事件/编排设计优点如下：

* 避免中央协调器单点故障风险。
* 当涉及的步骤较少服务开发简单，容易实现。

事件/编排设计缺点如下：

* 服务之间存在循环依赖的风险。
* 当涉及的步骤较多，服务间关系混乱，难以追踪调测。

值得补充的是，由于 Saga 模型中没有 Prepare 阶段，因此事务间不能保证隔离性。

当多个 Saga 事务操作同一资源时，就会产生更新丢失、脏数据读取等问题，这时需要在业务层控制并发，例如：在应用层面加锁，或者应用层面预先冻结资源。

总结

各方案使用场景



介绍完分布式事务相关理论和常见解决方案后，最终的目的在实际项目中运用，因此，总结一下各个方案的常见的使用场景：

* 2PC/3PC：依赖于数据库，能够很好的提供强一致性和强事务性，但相对来说延迟比较高，比较适合传统的单体应用，在同一个方法中存在跨库操作的情况，不适合高并发和高性能要求的场景。
* TCC：适用于执行时间确定且较短，实时性要求高，对数据一致性要求高，比如互联网金融企业最核心的三个服务：交易、支付、账务。
* 本地消息表/MQ 事务：都适用于事务中参与方支持操作幂等，对一致性要求不高，业务上能容忍数据不一致到一个人工检查周期，事务涉及的参与方、参与环节较少，业务上有对账/校验系统兜底。
* Saga 事务：由于 Saga 事务不能保证隔离性，需要在业务层控制并发，适合于业务场景事务并发操作同一资源较少的情况。 Saga 相比缺少预提交动作，导致补偿动作的实现比较麻烦，例如业务是发送短信，补偿动作则得再发送一次短信说明撤销，用户体验比较差。Saga 事务较适用于补偿动作容易处理的场景。

分布式事务方案设计

本文介绍的偏向于原理，业界已经有不少开源的或者收费的解决方案，篇幅所限，就不再展开介绍。

实际运用理论时进行架构设计时，许多人容易犯“手里有了锤子，看什么都觉得像钉子”的错误，设计方案时考虑的问题场景过多，各种重试，各种补偿机制引入系统，导致系统过于复杂，落地遥遥无期。

世界上解决一个计算机问题最简单的方法：“恰好”不需要解决它！

—— 阿里中间件技术专家沈询

有些问题，看起来很重要，但实际上我们可以通过合理的设计或者将问题分解来规避。

设计分布式事务系统也不是需要考虑所有异常情况，不必过度设计各种回滚，补偿机制。

如果硬要把时间花在解决问题本身，实际上不仅效率低下，而且也是一种浪费。

如果系统要实现回滚流程的话，有可能系统复杂度将大大提升，且很容易出现 Bug，估计出现 Bug 的概率会比需要事务回滚的概率大很多。

在设计系统时，我们需要衡量是否值得花这么大的代价来解决这样一个出现概率非常小的问题，可以考虑当出现这个概率很小的问题，能否采用人工解决的方式，这也是大家在解决疑难问题时需要多多思考的地方。

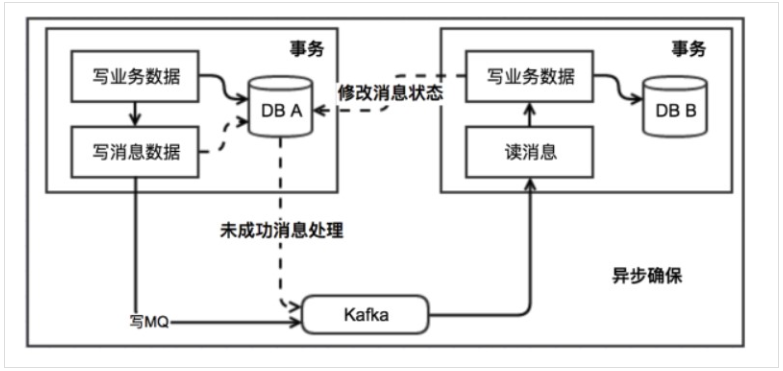
参考资料：

* technology-talk —— 事务
* MySQL 中事务的实现
* 分布式一致性算法 2PC 和 3PC
* 分布式开放消息系统(RocketMQ)的原理与实践
* RocketMQ 事务消息入门介绍
* Saga 分布式事务解决方案与实践 —— 姜宁
* 分布式事务 Saga 模式
* 从一笔金币充值去思考分布式事务

## 三、本地消息表（异步确保）

本地消息表与业务数据表处于同一个数据库中，这样就能利用本地事务来保证在对这两个表的操作满足事务特性，并且使用了消息队列来保证最终一致性。

1. 在分布式事务操作的一方完成写业务数据的操作之后向本地消息表发送一个消息，本地事务能保证这个消息一定会被写入本地消息表中。
2. 之后将本地消息表中的消息转发到 Kafka 等消息队列中，如果转发成功则将消息从本地消息表中删除，否则继续重新转发。
3. 在分布式事务操作的另一方从消息队列中读取一个消息，并执行消息中的操作。



优点： 一种非常经典的实现，避免了分布式事务，实现了最终一致性。

缺点： 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

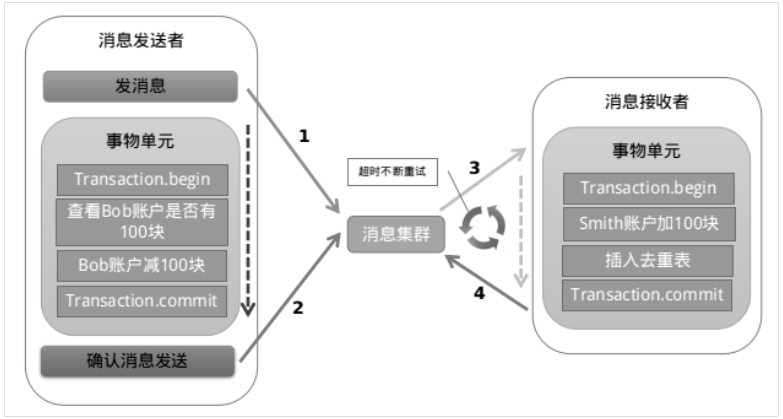
## 四、MQ 事务消息

有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址。 第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。



优点： 实现了最终一致性，不需要依赖本地数据库事务。

缺点： 实现难度大，主流MQ不支持，RocketMQ事务消息部分代码也未开源。

# 总结

通过本文我们总结并对比了几种分布式分解方案的优缺点，分布式事务本身是一个技术难题，是没有一种完美的方案应对所有场景的，具体还是要根据业务场景去抉择吧。阿里RocketMQ去实现的分布式事务，现在也有除了很多分布式事务的协调器，比如LCN等，大家可以多去尝试。