【分布式事务】面试题

以下面试题，基于网络整理，和自己编辑。具体参考的文章，会在文末给出所有的链接。

如果胖友有自己的疑问，欢迎在星球提问，我们一起整理吊吊的【分布式事务】面试题的大保健。

而题目的难度，尽量按照从容易到困难的顺序，逐步下去。

什么是分布式事务?

分布式事务就是指事务的参与者、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。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

或者，在换一句话说，分布式事务 = n 个本地事务。通过事务管理器，达到 n 个本地事务要么全部成功，要么全部失败。

为什么会有分布式事务?

从本地事务来看，我们可以看为两块，一个是 service 产生多个节点，另一个是 resource 产生多个节点。

😺 可能会有胖说，我们就是一个单体应用，不存在这样的情况。OK ，没问题，那么我们回过头来想想用户下单完成，我们需要给用户发短信。如果发送短信失败，可能是网络抖动的原因，我们是不应该去回滚本地事务，那么此时也可以认为是一个分布式事务。

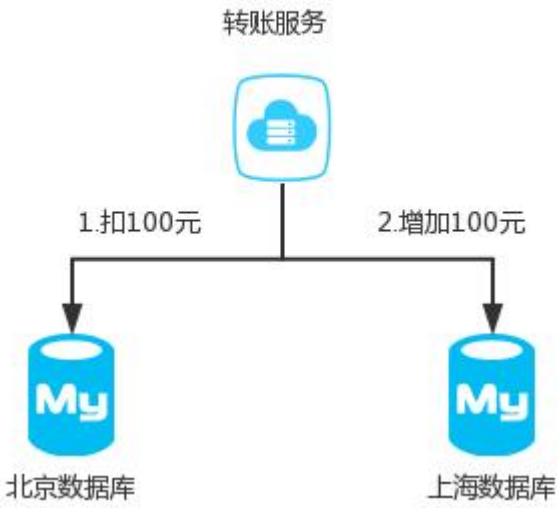
1）service 多个节点

随着互联网快速发展，微服务，SOA等服务架构模式正在被大规模的使用，举个简单的例子，一个公司之内，用户的资产可能分为好多个部分，比如余额，积分，优惠券等等。在公司内部有可能积分功能由一个微服务团队维护，优惠券又是另外的团队维护。

这样的话就无法保证积分扣减了之后，优惠券能否扣减成功。

2）resource 多个节点

同样的，互联网发展得太快了，我们的Mysql一般来说装千万级的数据就得进行分库分表，对于一个支付宝的转账业务来说，你给的朋友转钱，有可能你的数据库是在北京，而你的朋友的钱是存在上海，所以我们依然无法保证他们能同时成功。



😺 可能会有胖友说，我们数据没做分库分表，不存在这样的情况。OK，没问题，那么我们回过头来想想最常见的场景，系统里引入了 Redis 做缓存，那么 DB 和 Redis 的一致性问题，就是一种分布式事务的场景。

📣 **是否真的要分布式事务?**

在说分布式事务的方案之前，首先你一定要明确你是否真的需要分布式事务?

上面说过出现分布式事务的两个原因，其中有个原因是因为微服务过多。我见过太多团队一个人维护几个微服务，太多团队过度设计，搞得所有人疲劳不堪，而微服务过多就会引出分布式事务，这个时候我不会建议你去采用分布式事务的方案，而是请把需要事务的微服务聚合成一个单机服务，使用数据库的本地事务。因为不论任何一种方案都会增加你系统的复杂度，这样的成本实在是太高了，千万不要因为追求某些设计，而引入不必要的成本和复杂度。

当然，如果你是个人的练习 Demo 项目，请使劲的造，拼命的玩。甚至说，我建议你能读完所使用的分布式事务的方案的原理和源码。因为，一旦上了生产，出了问题，你很有可能无从下手~

所以，想清楚你是否需要分布式事务，你是否能够 hold 住分布式事务的解决方案。

分布式事务的基础?

数据库的 ACID 满足了数据库本地事务的基础，但是它无法满足分布式事务，这个时候衍生了 CAP 和 BASE 两个经典理论。

📣 **CAP 理论**

CAP 定理，又被叫作布鲁尔定理。对于设计分布式系统来说(不仅仅是分布式事务)的架构师来说，CAP 就是你的入门理论。

- C (一致性)：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）
- A (可用性)：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）
- P (分区容错性)：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。

高可用、数据一致性是很多系统设计的目标，但是分区又是不可避免的事情。我们来看一看分别拥有 CA、CP 和 AP 的情况。

- CA without P：如果不要求 P（不允许分区），则 C（强一致性）和A（可用性）是可以保证的。但其实分区不是你想不想的问题，而是始终会存在，因此 CA 的系统更多的是允许分区后各子系统依然保持 CA 。
- CP without A：如果不要求 A（可用），相当于每个请求都需要在 Server 之间强一致，而 P（分区）会导致同步时间无限延长，如此 CP 也是可以保证的。很多传统的数据库分布式事务都属于这种模式。
- AP wihtout C：要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。现在众多的NoSQL都属于此类。

可能胖友看完之后，会一脸懵逼，可以看看 《分布式系统理论（一）：CAP 定理》 文章提供的示例：

- MySQL 主从异步复制是 AP 系统。
- MySQL 主从半同步复制是 CP 系统。
- Zookeeper 是 CP 系统。
- Redis 主从同步是 AP 系统。
- Eureka 主从同步是 AP 系统。

从上的示例中，**“三选二”是一个伪命题。**不是为了 P（分区容忍性），要在 A 和 C 之间选择一个。分区很少出现，CAP 在大多数时候允许完美的 C 和 A 。但当分区存在或可感知其影响的情况下，就要预备一种策略去探知分区并显式处理其影响。

> ，如果关于**“三选二”是一个伪命题**无法理解，可以回过头在看一眼“CA without P”，对比下就好理解了。对于单节点，CA 必然是可以保证的。

另外，关于 CAP 的论证过程，也是蛮有趣的一块内容，感兴趣的胖友，可以自己去搜索下。

BASE 理论

BASE 是 Basically Available(基本可用)、Soft state(软状态)和 Eventually consistent (最终一致性) 三个短语的缩写。是对 CAP 中AP 的一个扩展

1. BA 基本可用：分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用。
2. S 软状态：允许系统中存在中间状态，这个状态不影响系统可用性，这里指的是 CAP 中的不一致。
3. E 最终一致：最终一致是指经过一段时间后，所有节点数据都将会达到一致。

BASE 解决了 CAP 中理论没有网络延迟，在 BASE 中用软状态和最终一致，保证了延迟后的一致性。

BASE 和 ACID 是相反的，它完全不同于 ACID 的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。

> 对于大部分的分布式应用而言，只要数据在规定的时间内达到最终一致性即可。我们可以把符合传统的 ACID 叫做刚性事务，把满足 BASE 理论的最终一致性事务叫做柔性事务。
>
> 一味的追求强一致性，并非最佳方案。对于分布式应用来说，刚柔并济是更加合理的设计方案，即在本地服务中采用强一致事务，在跨系统调用中采用最终一致性。如何权衡系统的性能与一致性，是十分考验架构师与开发者的设计功力的。

具体到分布式事务的实现上，业界主要采用了 XA 协议的强一致规范以及柔性事务的最终一致规范。

> ：所以，市面上的分布式事务的解决方案，除了 XA 协议是强一直的，其他都是最终一致的。

分布式事务的实现主要有哪些方案?

分布式事务的实现主要有以下 6 种方案：

- XA 方案
- TCC 方案
- 本地消息表
- 可靠消息最终一致性方案
- 最大努力通知方案
- SAGA

聊聊 XA 方案?

XA 是 X/Open CAE Specification (Distributed Transaction Processing)模型，它定义的 TM（Transaction Manager）与 RM（Resource Manager）之间进行通信的接口。

Java中 的 `javax.transaction.xa.XAResource` 定义了 XA 接口，它依赖数据库厂商对 jdbc-driver 的具体实现。

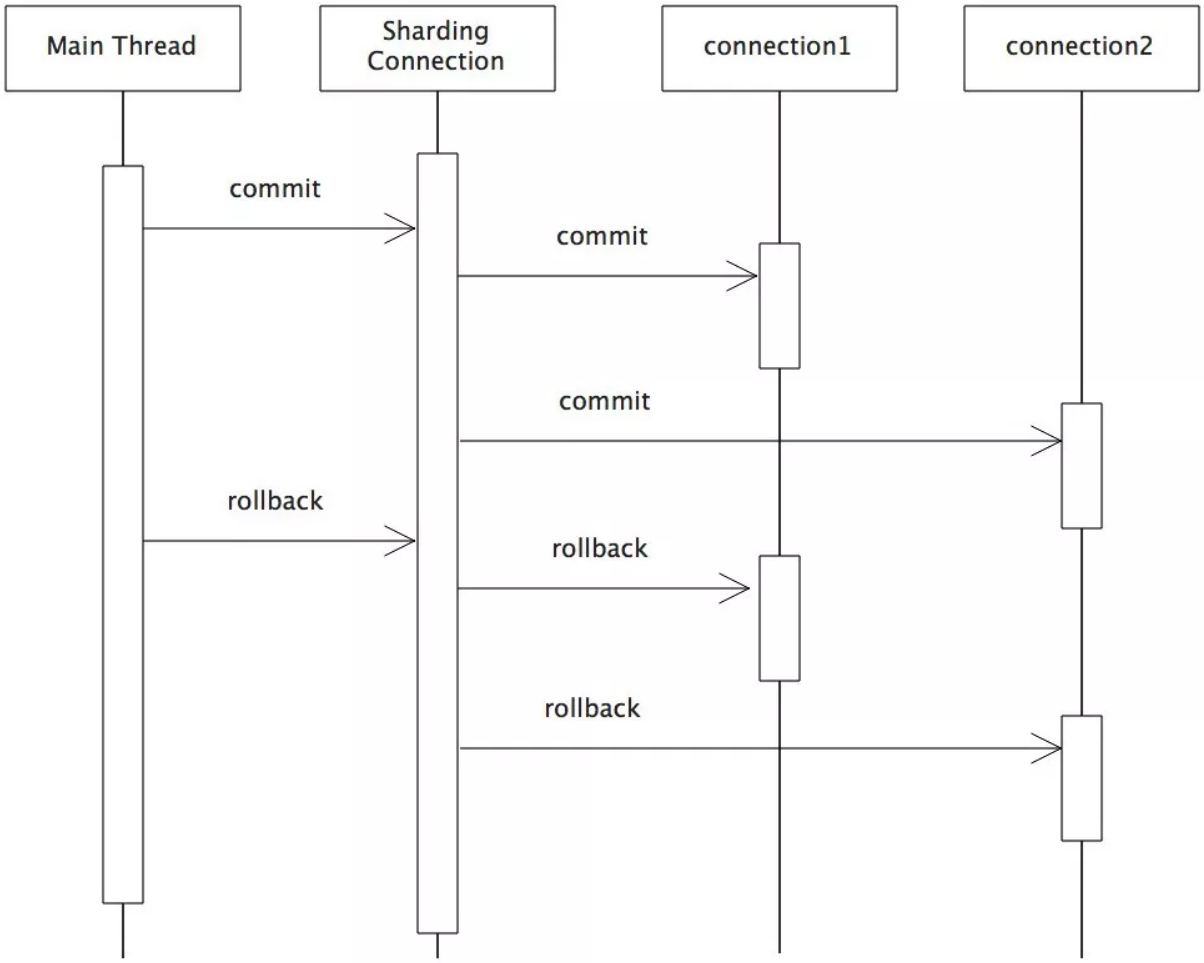
- `mysql-connector-java-5.1.30` 的实现可参 `com.mysql.jdbc.jdbc2.optional.MysqlXAConnection` 类。

在 XA 规范中，数据库充当 RM 角色，应用需要充当 TM 的角色，即生成全局的 txId ，调用 XAResource 接口，把多个本地事务协调为全局统一的分布式事务。

目前 XA 有两种实现：

- 基于一阶段提交(1PC) 的**弱** XA 。
- 基于二阶段提交(2PC) 的**强** XA 。

弱 XA



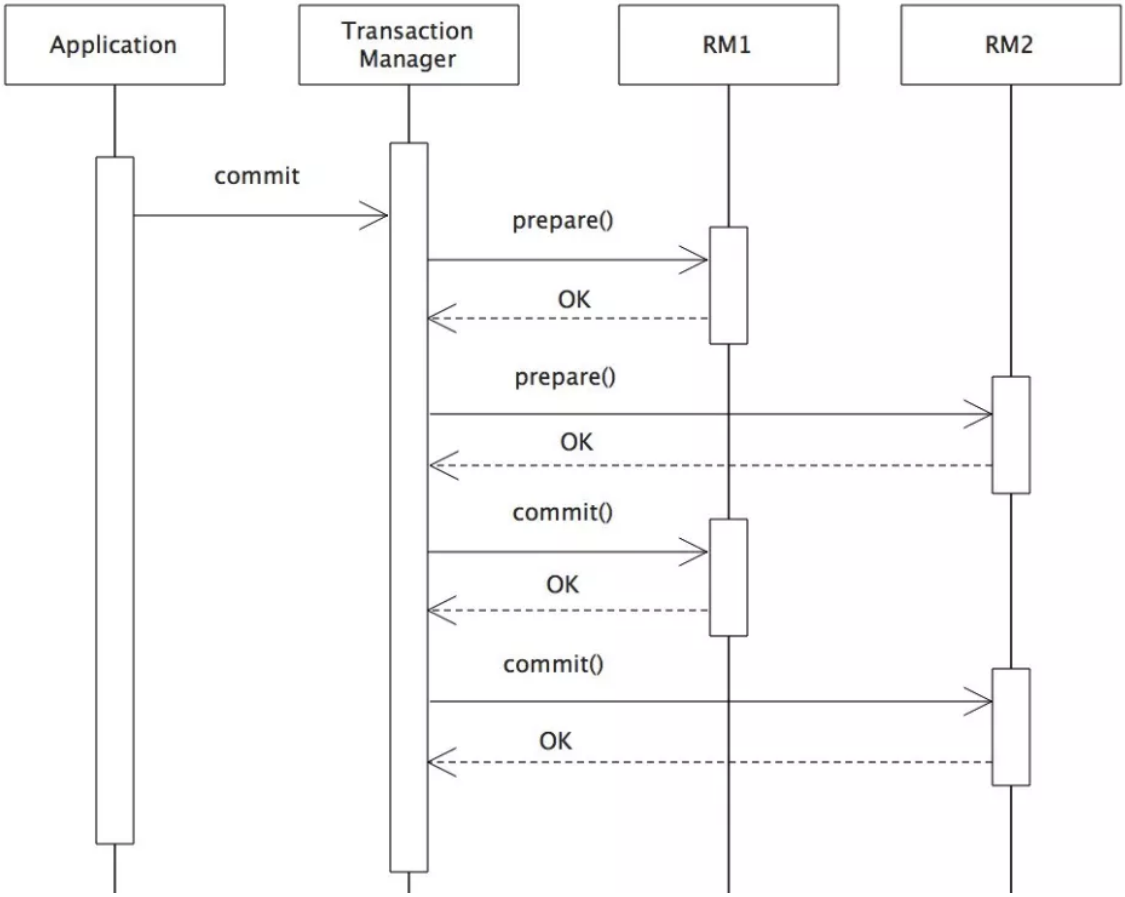
- 弱 XA 通过去掉 XA 的 Prepare 阶段，以达到减少资源锁定范围而提升并发性能的效果。典型的实现为在一个业务线程中，遍历所有的数据库连接，依次做 commit 或者 rollback 。
- 弱 XA 同本地事务相比，性能损耗低，但在事务提交的执行过程中，若出现网络故障、数据库宕机等预期之外的异常，将会造成数据不一致，且无法进行回滚。

📣 解决方案?

基于弱 XA 的事务无需额外的实现成本，相对容易。目前支持的有：

- MyCAT ，具体的源码解析，可以看看 《MyCAT 源码分析 —— XA 分布式事务》 。
- Sharding-Sphere 默认支持。

强 XA



- 二阶段提交是 XA 的标准实现。它将分布式事务的提交拆分为 2 个阶段：prepare 和 commit/rollback 。
  - 第一阶段：事务管理器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交。
  - 第二阶段：事务协调器要求每个数据库提交数据，或者回滚数据。
- 开启 XA 全局事务后，所有子事务会按照本地默认的隔离级别锁定资源，并记录 undo 和 redo 日志。然后由 TM 发起 prepare 投票，询问所有的子事务是否可以进行提交：
  - 当所有子事务反馈的结果为 “yes” 时，TM 再发起 commit 。
  - 若其中任何一个子事务反馈的结果为“no”，TM 则发起 rollback 。
  - 如果在 prepare 阶段的反馈结果为 “yes” ，而 commit 的过程中出现宕机等异常时，则在节点服务重启后，可根据 XA recover 再次进行 commit 补偿，以保证数据的一致性。

📣 优点?

- 尽量保证了数据的强一致，实现成本较低，在各大主流数据库都有自己实现，对于 MySQL 是从 5.5 开始支持。

📣 缺点?

- 单点问题：事务管理器在整个流程中扮演的角色很关键，如果其宕机，比如在第一阶段已经完成，在第二阶段正准备提交的时候事务管理器宕机，资源管理器就会一直阻塞，导致数据库无法使用。

  > ：如果事务管理器是 Proxy 模式的数据库中间件，并且实现高可用，可能可以解决这个问题。不太肯定，需要到时翻下 Sharding Sphere 的源码。TODO

- 同步阻塞：在准备就绪之后，资源管理器中的资源一直处于阻塞，直到提交完成，释放资源。

- 数据不一致：两阶段提交协议虽然为分布式数据强一致性所设计，但仍然存在数据不一致性的可能，比如在第二阶段中，假设协调者发出了事务commit 的通知，但是因为网络问题该通知仅被一部分参与者所收到并执行了 commit 操作，其余的参与者则因为没有收到通知一直处于阻塞状态，这时候就产生了数据的不一致性。

  > ：此处的数据不一致也问题不大，因为使用 xa 会锁定记录，无法被访问。

📣 解决方案?

- Sharding Sphere

  > Sharding Sphere 支持基于 XA 的强一致性事务解决方案，可以通过 SPI 注入不同的第三方组件作为事务管理器实现 XA 协议，如 Atomikos 和 Narayana 。

- Spring JTA + Atomikos

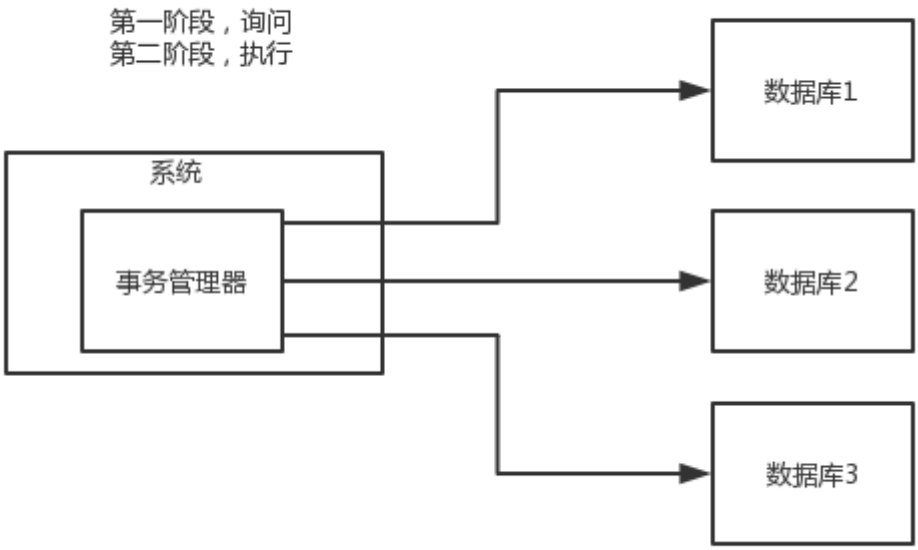
应用场景

这种分布式事务方案，比较适合单块应用里，跨多个库的分布式事务，而且因为严重依赖于数据库层面来搞定复杂的事务，效率很低，绝对不适合高并发的场景。

这个方案，我们很少用，一般来说**某个系统内部如果出现跨多个库**的这么一个操作，是**不合规**的。我可以给大家介绍一下， 现在微服务，一个大的系统分成几百个服务，几十个服务。一般来说，我们的规定和规范，是要求**每个服务只能操作自己对应的一个数据库。**

如果你要操作别的服务对应的库，不允许直连别的服务的库，违反微服务架构的规范，你随便交叉胡乱访问，几百个服务的话，全体乱套，这样的一套服务是没法管理的，没法治理的，可能会出现数据被别人改错，自己的库被别人写挂等情况。

如果你要操作别人的服务的库，你必须是通过**调用别的服务的接口**来实现，绝对不允许交叉访问别人的数据库。



聊聊 TCC 方案?

TCC 模型是把锁的粒度完全交给业务处理，它需要每个子事务业务都实现Try-Confirm / Cancel 接口。

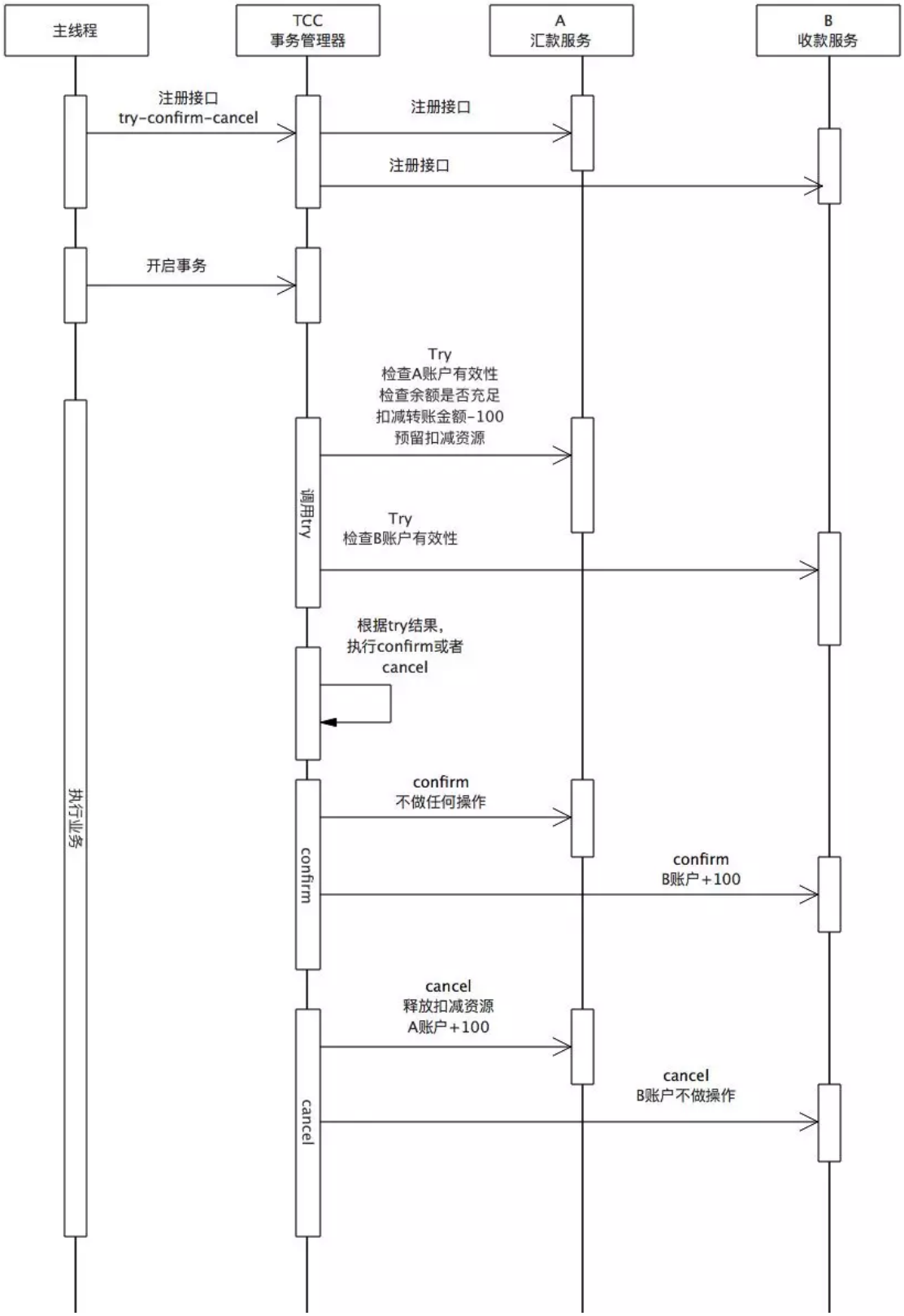
> TCC 模式本质也是 2PC ， 只是 TCC 在应用层控制。

- 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操作满足幂等性

这三个阶段，都会按本地事务的方式执行。不同于 XA的prepare ， TCC 无需将 XA 的投票期间的所有资源挂起，因此极大的提高了吞吐量。

🚀 应用场景

下面对TCC模式下，A账户往B账户汇款100元为例子，对业务的改造进行详细的分析：



- 汇款服务和收款服务分别需要实现，Try-Confirm-Cancel 接口，并在业务初始化阶段将其注入到 TCC 事务管理器中。

汇款服务

- Try：
  - 检查A账户有效性，即查看A账户的状态是否为“转帐中”或者“冻结”

    - 检查A账户余额是否充足
    - 从A账户中扣减 100 元，并将状态置为“转账中”
    - 预留扣减资源，将从 A 往 B 账户转账 100 元这个事件存入消息或者日志中
- Confirm：
    - 不做任何操作
- Cancel：
    - A 账户增加 100 元
    - 从日志或者消息中，释放扣减资源

收款服务

- Try：
    - 检查 B 账户账户是否有效；
- Confirm：
    - 读取日志或者消息，B 账户增加 100 元
    - 从日志或者消息中，释放扣减资源；
- Cancel：
    - 不做任何操作

由此可以看出，TCC 模型对业务的侵入强，改造的难度大。

但是，在需要前置资源锁定的场景，不得不使用 XA 或 TCC 的方式。再例如说，下单场景，在订单创建之前，需要扣除如下几个资源：

- 优惠券
- 钱包余额
- 积分
- ....

那么，不得不进行前置多资源锁定，无非是使用 XA 的强锁，还是 TCC 的弱锁。在 oceans 的 tag `0.0.1` 中，在未使用 TCC 的情况下，模拟 TCC 的效果的苦闷。

当然，如果能不用 TCC 的情况下，尽量不要用 TCC 。因为，编写回滚逻辑的代码，可能会比较恶心。

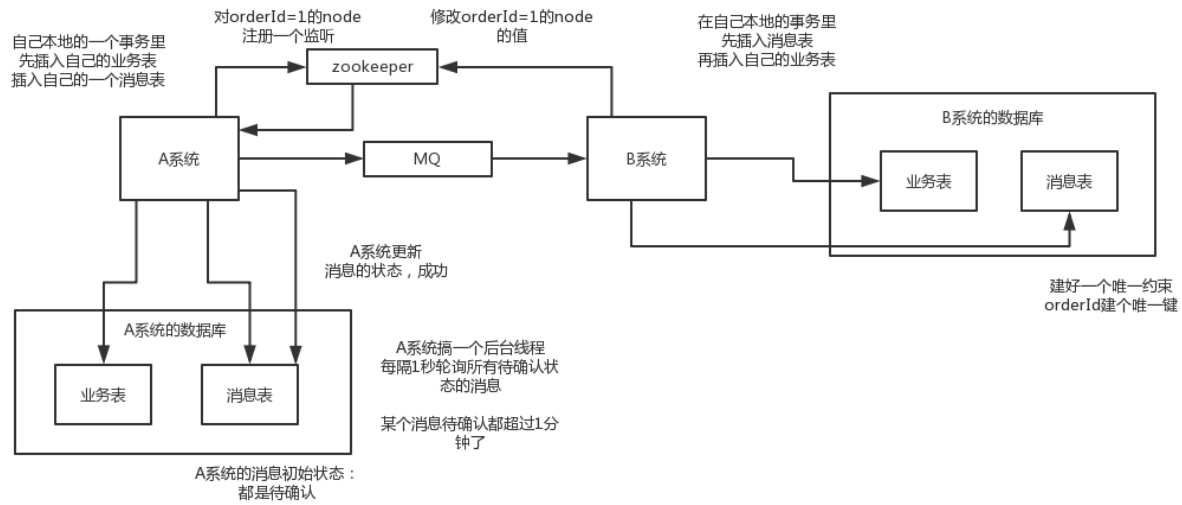
📣 **解决方案?**

- TCC-Transaction ，听说喜马拉雅在用。具体的源码解析，可以看看 《TCC-Transaction 源码分析》 。
- Hmily ，具体的源码解析，可以看看 《Hmily 实现原理与源码解析系列 —— 精品合集》 。
- ByteTCC

聊聊本地消息表?

本地消息表，其实是 国外的 Ebay 搞出来的这么一套思想 。

这个大概意思是这样的：



1. A 系统在自己本地一个事务里操作同时，插入一条数据到消息表；
2. 接着 A 系统将这个消息发送到 MQ 中去；
3. B 系统接收到消息之后，在一个事务里，往自己本地消息表里插入一条数据，同时执行其他的业务操作，如果这个消息已经被处理过了，那么此时这个事务会回滚，这样**保证不会重复处理消息**；
4. B 系统执行成功之后，就会更新自己本地消息表的状态以及 A 系统消息表的状态；
5. 如果 B 系统处理失败了，那么就不会更新消息表状态，那么此时 A 系统会定时扫描自己的消息表，如果有未处理的消息，会再次发送到 MQ 中去，让 B 再次处理；
6. 这个方案保证了最终一致性，哪怕 B 事务失败了，但是 A 会不断重发消息，直到 B 那边成功为止。

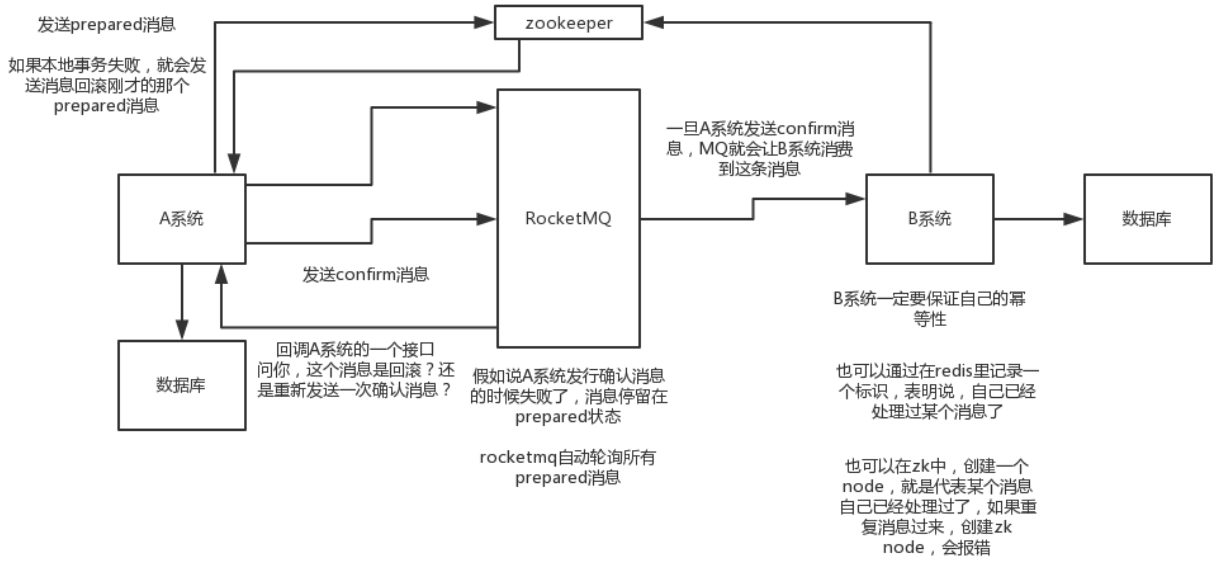
这个方案说实话最大的问题就在于**严重依赖于数据库的消息表来管理事务**啥的，会导致如果是高并发场景咋办呢？咋扩展呢？所以一般确实很少用。

本地消息队列是 BASE 理论，是最终一致模型，适用于对一致性要求不高的。实现这个模型时需要注意重试的幂等。

聊聊可靠消息最终一致性方案？

这个的意思，就是干脆不要用本地的消息表了，直接基于 MQ 来实现事务。比如阿里的 RocketMQ 就支持消息事务。

大概的意思就是：



1. A 系统先发送一个 prepared 消息到 mq，如果这个 prepared 消息发送失败那么就直接取消操作别执行了；
2. 如果这个消息发送成功过了，那么接着执行本地事务，如果成功就告诉 mq 发送确认消息，如果失败就告诉 mq 回滚消息；
3. 如果发送了确认消息，那么此时 B 系统会接收到确认消息，然后执行本地的事务；
4. mq 会自动**定时轮询**所有 prepared 消息回调你的接口，问你，这个消息是不是本地事务处理失败了，所有没发送确认的消息，是继续重试还是回滚？一般来说这里你就可以查下数据库看之前本地事务是否执行，如果回滚了，那么这里也回滚吧。这个就是避免可能本地事务执行成功了，而确认消息却发送失败了。
5. 这个方案里，要是系统 B 的事务失败了咋办？重试咯，自动不断重试直到成功，如果实在是不行，要么就是针对重要的资金类业务进行回滚，比如 B 系统本地回滚后，想办法通知系统 A 也回滚；或者是发送报警由人工来手工回滚和补偿。

这个还是比较合适的，目前国内互联网公司大都是这么玩儿的。

解决方案

- RocketMQ 事务消息，源码解析，可见 《RocketMQ 源码分析 —— 事务消息》 。

  > 虽然 RocketMQ 早期开源事务消息后又阉割闭源，但是在 RocketMQ 4.3 版本中，又重新提供。所以，不要搞错落。

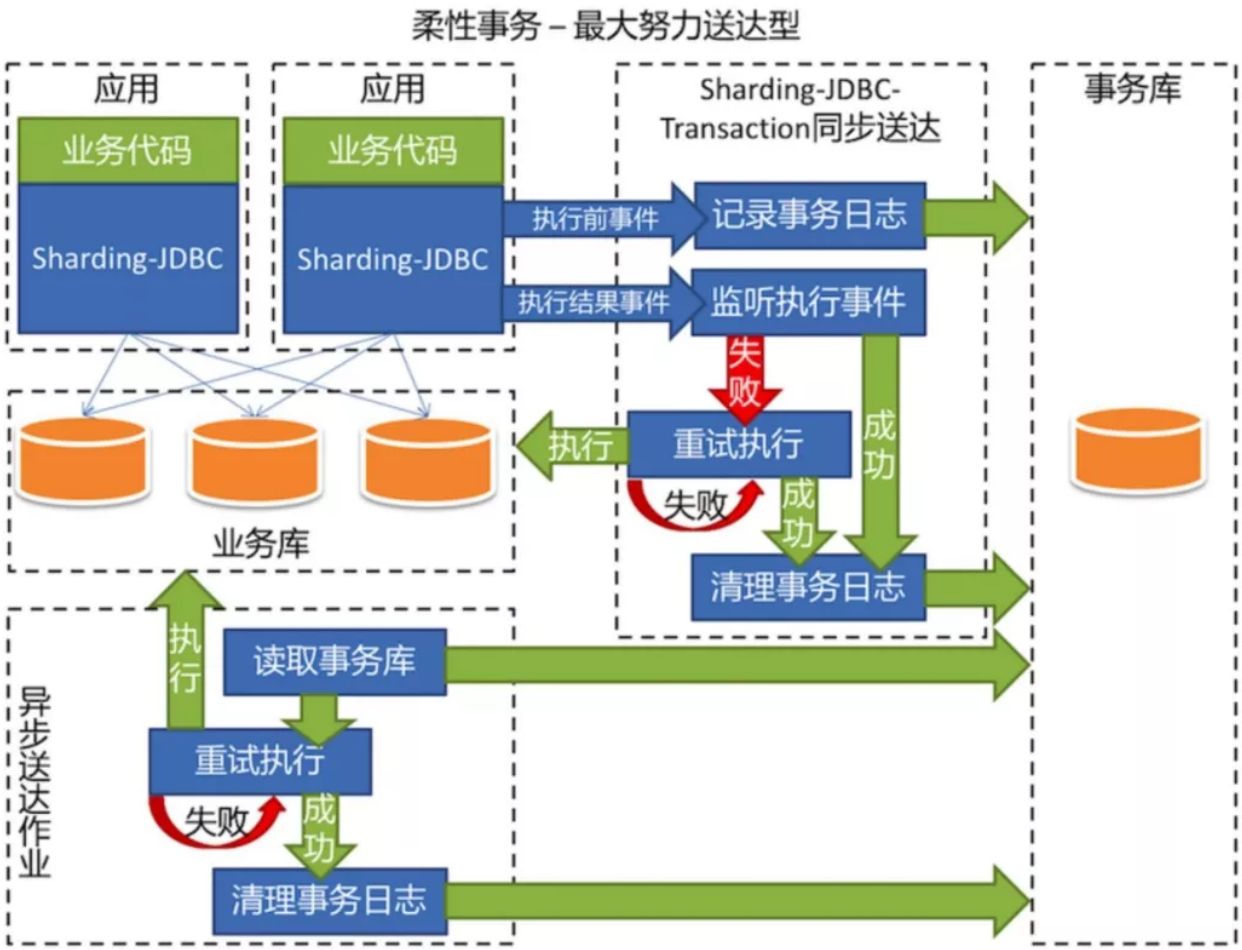
- 《RabbitMQ 之消息确认机制（事务+Confirm）》
- Kafka 事务消息，https://zhuanlan.zhihu.com/p/42046847 TODO 需要找厮大确认下

聊聊最大努力通知方案?

瞅了瞅市面上的资料，分别有两种解释。或者说，两种不同的解决方案。

解释一

最大努力送达，是针对于弱 XA 的一种补偿策略。它采用事务表记录所有的事务操作 SQL 。



- 如果子事务提交成功，将会删除事务日志。
- 如果执行失败，则会按照配置的重试次数，尝试再次提交，即最大努力的进行提交，尽量保证数据的一致性，这里可以根据不同的业务场景，平衡 C 和 A ，采用同步重试或异步重试。

📣 优点

- 无锁定资源时间，性能损耗小。

📣 缺点

- 尝试多次提交失败后，无法回滚，它仅适用于事务最终一定能够成功的业务场景。

📣 总结

- 因此 BED 是通过事务回滚功能上的妥协，来换取性能的提升。

貌似，暂时也想象不到具体的使用场景。

📣 解决方案

正如上图，提供的解决方式 Sharding-JDBC ，具体的源码解析，可见 《Sharding-JDBC 源码分析 —— 分布式事务（一）之最大努力型》 。

解释二

这个方案的大致意思就是：

1. 系统 A 本地事务执行完之后，发送个消息到 MQ；

2. 这里会有个专门消费 MQ 的**最大努力通知服务**，这个服务会消费 MQ 然后写入数据库中记录下来，或者是放入个内存队列也可以，接着调用系统 B 的接口；
3. 要是系统 B 执行成功就 ok 了；要是系统 B 执行失败了，那么最大努力通知服务就定时尝试重新调用系统 B，反复 N 次，最后还是不行就放弃。

📣 **解决方案**

按照这个解释，RocketMQ 的消息重试，符合这个解释。具体的源码解析，见 《RocketMQ 源码分析 —— 定时消息与消息重试》 。

比较常见的场景，就是支付成功后，多次回调~

聊聊 Saga 方案?

Saga 是 30 年前一篇数据库伦理提到的一个概念。其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。

Saga 的组成如下：

- 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。这里的每个 T ，都是一个本地事务。
- 可以看到，和 TCC 相比，Saga 没有"预留 try"动作 ，它的 Ti 就是直接提交到库。

Saga的执行顺序有两种：

- 子事务序列 T1, T2, ..., Tn得以完成 (最佳情况)。
- 或者序列 T1, T2, ..., Tj, Cj, ..., C2, C1, 0 < j < n, 得以完成。

Saga 定义了两种恢复策略：

- 向后恢复：补偿所有已完成的事务，如果任一子事务失败。

  > 向后恢复，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。

- 向前恢复：重试失败的事务，假设每个子事务最终都会成功。

  > 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，这时需要提供故障恢复后回退的机制，比如人工干预。

📣 **如何解决没有 Prepare阶段可能带来的问题?**

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。
- 应用层面预先冻结资源。

还是拿 100 元买一瓶水的例子来说。

- 这里定义：
  - T1=扣100元 T2=给用户加一瓶水 T3=减库存一瓶水
  - C1=加100元 C2=给用户减一瓶水 C3=给库存加一瓶水

- 我们一次进行 T1，T2，T3。如果发生问题，就执行发生问题的 C 操作的反向。上面说到的隔离性的问题会出现在，如果执行到 T3 这个时候需要执行回滚，但是这个用户已经把水喝了(**另外一个事务**)，回滚的时候就会发现，无法给用户减一瓶水了。这就是事务之间没有隔离性的问题。

  > ：也就是说，给的太早，但是可以被取消！

可以看见 Saga 模式没有隔离性的影响还是较大，可以参照华为的解决方案:

- 从业务层面入手加入一 Session 以及锁的机制来保证能够串行化操作资源。
- 也可以在业务层面通过预先冻结资金的方式隔离这部分资源，最后在业务操作的过程中可以通过及时读取当前状态的方式获取到最新的更新。

📣 **解决方案**

- Apache Service Comb 的 Saga 事务引擎
- Sharding Sphere 的 Saga 支持

> 实际是基于 Apache Service Comb 的 Saga 事务引擎之上进行开发。

你们公司是如何处理分布式事务的?

如果你真的被问到，可以这么说：

- 我们某某特别严格的场景，用的是 TCC 来保证强一致性。

  > 你找一个严格资金要求绝对不能错的场景，你可以说你是用的 TCC 方案。

- 然后其他的一些场景，基于阿里的 RocketMQ 来实现了分布式事务。

  > 如果是一般的分布式事务场景，订单插入之后要调用库存服务更新库存，库存数据没有资金那么的敏感，可以用可靠消息最终一致性方案。

什么是三阶段协议?

这个问题，严格来说不属于【分布式事务】相关，考虑到本文已经出现了一阶段提交、二阶段提交，所以这里就瞬时“硬塞”一个三阶段提交。

感兴趣的胖友，可以看看 《数据库 分布式事务 2阶提交 3阶提交》 文章。

##

事务解决方案的对比总结

总的来说，TCC 和 MQ 都是以服务为范围进行分布式事务的处理，而 XA、BED、SAGA 则是以数据库为范围进行分布式处理。

> 对于数据库中间来来说，更趋向于选择后者，对于业务而言侵入小，改造的成本低。

	XA	最大努力送达	Saga	TCC
业务改造	无	无	实现补偿接口	实现TCC接口
回滚	支持	不支持	支持	支持
一致性	强一致	最终一致	最终一致	最终一致
隔离性	原生支持	不支持	不支持	Try接口支持
并发性能	严重衰退	无影响	无影响	略微衰退
适合场景	短事务 并发较低	事务最终成功 高并发	长事务 应用方控制并发 高并发	长事务 高并发
支持现状	完成，测试中	支持	开发中	规划中

- 图中暂时未包括：1）本地消息表；2）可靠消息最终一致性方案 。因为，这个是 Sharding Sphere 官方提供的图，嘻嘻。

彩蛋

😺 虽然以前基本读过市面上很多分布式事务解决方案的源码，但是因为是断断续续的读，没有专门花时间去整理。这次重新整理了下，收获还是蛮多的。

另外，开源社区里还有两个同类型的 2PC 事务解决方案，感兴趣的胖友，也可以看看：

> ：主要我没有特别研究，所以也不太好写。

- Raincat
- LCN

参考与推荐如下文章：

- 高效运维 《分布式系统的 CAP 理论》
- 咖啡拿铁 《再有人问你分布式事务，把这篇扔给他》
- yanglbme 《分布式事务了解吗？你们如何解决分布式事务问题的？TCC 如果出现网络连不通怎么办？XA 的一致性如何保证？》