链滴

XA 规范与 TCC 事务模型

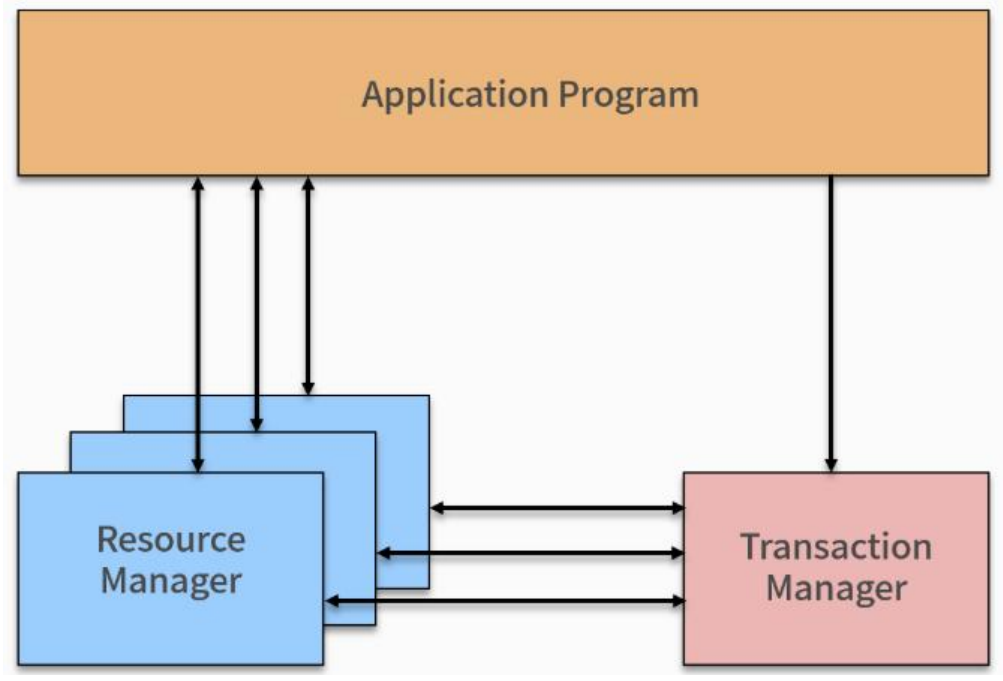
作者：jianzh5

原文链接：https://ld246.com/article/1588748307781

来源网站：链滴

XA规范

XA 是由 X/Open 组织提出的分布式事务规范，XA 规范主要定义了事务协调者（Transaction Manager）和资源管理器（Resource Manager）之间的接口。



事务协调者（Transaction Manager），因为 XA 事务是基于两阶段提交协议的，所以需要有一个调者，来保证所有的事务参与者都完成了准备工作，也就是 2PC 的第一阶段。如果事务协调者收到有参与者都准备好的消息，就会通知所有的事务都可以提交，也就是 2PC 的第二阶段。

在前面的内容中我们提到过，之所以需要引入事务协调者，是因为在分布式系统中，两台机器理论上法达到一致的状态，需要引入一个单点进行协调。协调者，也就是事务管理器控制着全局事务，管理务生命周期，并协调资源。

资源管理器（Resource Manager），负责控制和管理实际资源，比如数据库或 JMS 队列。目前，流数据库都提供了对 XA 的支持，在 JMS 规范中，即 Java 消息服务（Java Message Service）中，基于 XA 定义了对事务的支持。

XA 事务的执行流程XA

事务是两阶段提交的一种实现方式，根据 2PC 的规范，XA 将一次事务分割成了两个阶段，即 Prepar 和 Commit 阶段。

Prepare 阶段， TM 向所有 RM 发送 prepare 指令，RM 接受到指令后，执行数据修改和日志记录操作，然后返回可以提交或者不提交的消息给 TM。如果事务协调者 TM 收到所有参与者都准备好的息，会通知所有的事务提交，然后进入第二阶段。

Commit 阶段， TM 接受到所有 RM 的 prepare 结果，如果有 RM 返回是不可提交或者超时，那么所有 RM 发送 Rollback 命令；如果所有 RM 都返回可以提交，那么向所有 RM 发送 Commit 命令完成一次事务操作。

MYSQL如何实现XA规范

在 MySQL 的 InnoDB 存储引擎中，开启 binlog 的情况下，MySQL 会同时维护 binlog 日志与 InnoB 的 redo log，为了保证这两个日志的一致性，MySQL 使用了 XA 事务。

Binlog 中的 Xid

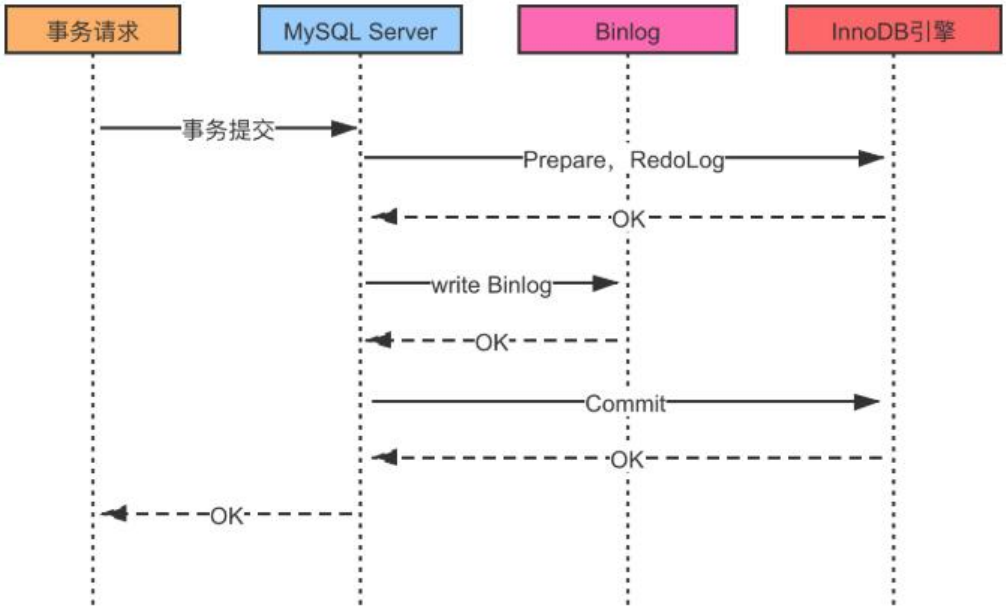
当事务提交时，在 binlog 依赖的内部 XA 中，额外添加了 Xid 结构，binlog 有多种数据类型，包括下三种：

- statement 格式，记录为基本语句，包含 Commit
- row 格式，记录为基于行
- mixed 格式，日志记录使用混合格式

不论是 statement 还是 row 格式，binlog 都会添加一个 XID_EVENT 作为事务的结束，该事件记录事务的 ID 也就是 Xid，在 MySQL 进行崩溃恢复时根据 binlog 中提交的情况来决定如何恢复。

Binlog 同步过程

下面来看看 Binlog 下的事务提交过程，整体过程是先写 redo log，再写 binlog，并以 binlog 写成为事务提交成功的标志。



当有事务提交时：

- 第一步，InnoDB 进入 Prepare 阶段，并且 write/sync redo log，写 redo log，将事务的 XID 写到 redo 日志中，binlog 不作任何操作；
- 第二步，进行 write/sync Binlog，写 binlog 日志，也会把 XID 写入到 Binlog；
- 第三步，调用 InnoDB 引擎的 Commit 完成事务的提交，将 Commit 信息写入到 redo 日志中。

如果是在第一步和第二步失败，则整个事务回滚；如果是在第三步失败，则 MySQL 在重启后会检查 XID 是否已经提交，若没有提交，也就是事务需要重新执行，就会在存储引擎中再执行一次提交操作，障 redo log 和 binlog 数据的一致性，防止数据丢失。

在实际执行中，还牵扯到操作系统缓存 Buffer 何时同步到文件系统中，所以 MySQL 支持用户自定在 Commit 时如何将 log buffer 中的日志刷到 log file 中，通过变量 innodb_flush_log_at_trx_Commit 的值来决定。在 log buffer 中的内容称为脏日志，感兴趣的话可以查询资料了解下。

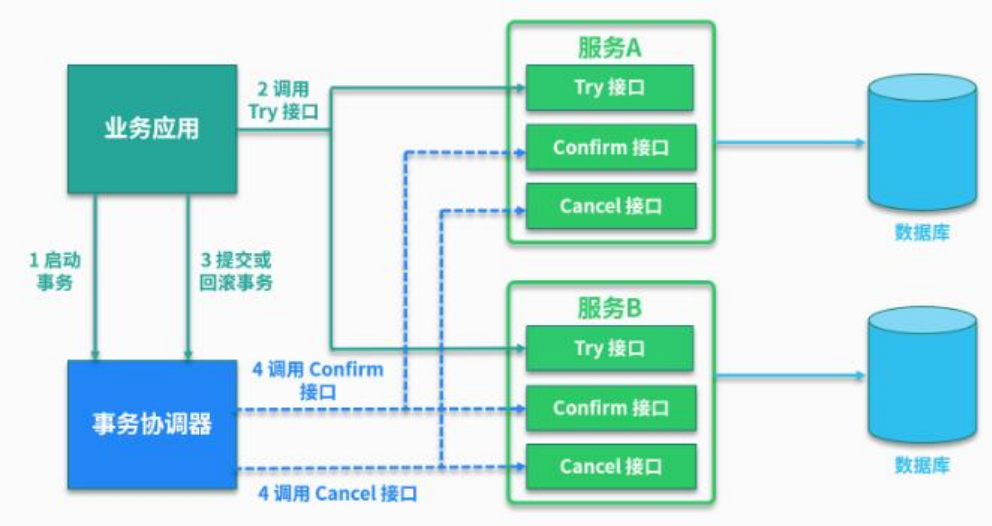
TCC事务模型

TCC（Try-Confirm-Cancel）的概念来源于 Pat Helland 发表的一篇名为“Life beyond Distributed Transactions:an Apostate’ s Opinion”的论文。

TCC 提出了一种新的事务模型，基于业务层面的事务定义，锁粒度完全由业务自己控制，目的是解决杂业务中，跨表跨库等大颗粒度资源锁定的问题。TCC 把事务运行过程分成 Try、Confirm / Cancel 个阶段，每个阶段的逻辑由业务代码控制，避免了长事务，可以获取更高的性能。

TCC的各个阶段

TCC 的具体流程如下图所示：



Try 阶段： 调用 Try 接口，尝试执行业务，完成所有业务检查，预留业务资源。

Confirm 或 Cancel 阶段： 两者是互斥的，只能进入其中一个，并且都满足幂等性，允许失败重试。

- **Confirm 操作：** 对业务系统做确认提交，确认执行业务操作，不做其他业务检查，只使用 Try 阶预留的业务资源。
- **Cancel 操作：** 在业务执行错误，需要回滚的状态下执行业务取消，释放预留资源。

Try 阶段失败可以 Cancel，如果 Confirm 和 Cancel 阶段失败了怎么办?

TCC 中会添加事务日志，如果 Confirm 或者 Cancel 阶段出错，则会进行重试，所以这两个阶段需支持幂等；如果重试失败，则需要人工介入进行恢复和处理等。

应用 TCC 的优缺点

实际开发中，TCC 的本质是把数据库的二阶段提交上升到微服务来实现，从而避免数据库二阶段中长务引起的低性能风险。

所以说，TCC 解决了跨服务的业务操作原子性问题，比如下订单减库存，多渠道组合支付等场景，通 TCC 对业务进行拆解，可以让应用自己定义数据库操作的粒度，可以降低锁冲突，提高系统的业务吐量。

TCC 的不足主要体现在对微服务的侵入性强，TCC 需要对业务系统进行改造，业务逻辑的每个分支需要实现 try、Confirm、Cancel 三个操作，并且 Confirm、Cancel 必须保证幂等。

另外 TCC 的事务管理器要记录事务日志，也会损耗一定的性能。

从真实业务场景分析 TCC

下面以一个电商中的支付业务来演示，用户在支付以后，需要进行更新订单状态、扣减账户余额、增账户积分和扣减商品操作。

在实际业务中为了防止超卖，有下单减库存和付款减库存的区别，支付除了账户余额，还有各种第三支付等，这里我们为了描述方便，统一使用扣款减库存，扣款来源是用户账户余额。

业务逻辑拆解

我们把订单业务拆解为以下几个步骤：

- 订单更新为支付完成状态
- 扣减用户账户余额
- 增加用户账户积分
- 扣减当前商品的库存

如果不使用事务，上面的几个步骤都可能出现失败，最终会造成大量的数据不一致，比如订单状态更失败，扣款却成功了；或者扣款失败，库存却扣减了等情况，这个在业务上是不能接受的，会出现大的客诉。

如果直接应用事务，不使用分布式事务，比如在代码中添加 Spring 的声明式事务 @Transactional 解，这样做实际上是在事务中嵌套了远程服务调用，一旦服务调用出现超时，事务无法提交，就会导数据库连接被占用，出现大量的阻塞和失败，会导致服务宕机。另一方面，如果没有定义额外的回滚作，比如遇到异常，非 DB 的服务调用失败时，则无法正确执行回滚。

业务系统改造

为了应用 TCC 事务模型，需要对业务代码改造，抽象 Try、Confirm 和 Cancel 阶段。

- Try

Try 操作一般都是锁定某个资源，设置一个预备的状态，冻结部分数据。比如，订单服务添加一个预状态，修改为 UPDATING，也就是更新中的意思，冻结当前订单的操作，而不是直接修改为支付成。

库存服务设置冻结库存，可以扩展字段，也可以额外添加新的库存冻结表。积分服务和库存一样，添一个预增加积分，比如本次订单积分是 100，添加一个额外的存储表示等待增加的积分，账户余额服等也是一样的操作。

- Confirm

Confirm 操作就是把前边的 Try 操作锁定的资源提交，类比数据库事务中的 Commit 操作。在支付场景中，包括订单状态从准备中更新为支付成功；库存数据扣减冻结库存，积分数据增加预增加积分。

- Cancel

Cancel 操作执行的是业务上的回滚处理，类比数据库事务中的 Rollback 操作。首先订单服务，撤销备状态，还原为待支付状态或者已取消状态，库存服务删除冻结库存，添加到可销售库存中，积分服也是一样，将预增加积分扣减掉。

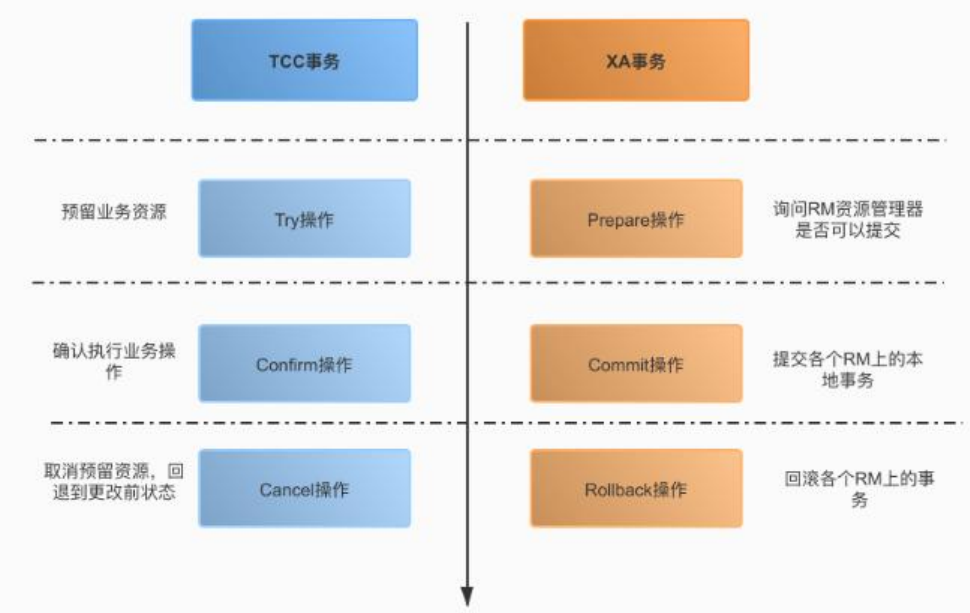
执行业务操作

下面来分析业务的实际执行操作，首先业务请求过来，开始执行 Try 操作，如果 TCC 分布式事务框感知到各个服务的 Try 阶段都成功了以后，就会执行各个服务的 Confirm 逻辑。

如果 Try 阶段有操作不能正确执行，比如订单失效、库存不足等，就会执行 Cancel 的逻辑，取消事提交。

XA VS TCC

TCC 事务模型的思想类似 2PC 提交，下面对比 TCC 和基于 2PC 事务 XA 规范对比。



● 第一阶段

在 XA 事务中，各个 RM 准备提交各自的事务分支，事实上就是准备提交资源的更新操作（insert、delete、update 等）；而在 TCC 中，是主业务操作请求各个子业务服务预留资源。

● 第二阶段

XA 事务根据第一阶段每个 RM 是否都 prepare 成功，判断是要提交还是回滚。如果都 prepare 成，那么就 commit 每个事务分支，反之则 rollback 每个事务分支。

在 TCC 中，如果在第一阶段所有业务资源都预留成功，那么进入 Confirm 步骤，提交各个子业务服，完成实际的业务处理，否则进入 Cancel 步骤，取消资源预留请求。

区别

● 2PC/XA 是数据库或者存储资源层面的事务，实现的是强一致性，在两阶段提交的整个过程中，一会持有数据库的锁。

● TCC 关注业务层的正确提交和回滚，在 Try 阶段不涉及加锁，是业务层的分布式事务，关注最终一性，不会一直持有各个业务资源的锁。

TCC 的核心思想是针对每个业务操作，都要添加一个与其对应的确认和补偿操作，同时把相关的处理从数据库转移到业务中，以此实现跨数据库的事务。