分布式事务解决方案

1. 事务简介

事务(Transaction)是访问并可能更新数据库中各种数据项的一个程序执行单元(unit)。在关系数据库中，一个事务由一组SQL语句组成。事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

原子性（atomicity）：个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。

一致性（consistency）：事务必须是使数据库从一个一致性状态变到另一个一致性状态，事务的中间状态不能被观察到的。

隔离性（isolation）：一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。隔离性又分为四个级别：读未提交(read uncommitted)、读已提交(read committed，解决脏读)、可重复读(repeatable read，解决虚读)、串行化(serializable，解决幻读)。

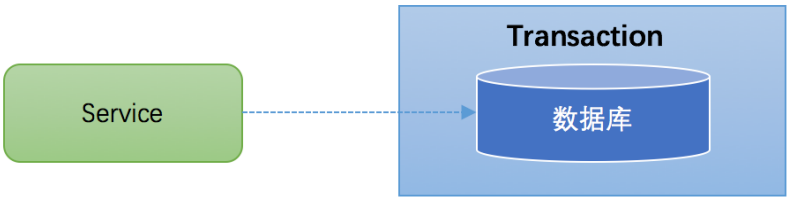
持久性（durability）：持久性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

任何事务机制在实现时，都应该考虑事务的ACID特性，包括：本地事务、分布式事务，及时不能都很好的满足，也要考虑支持到什么程度。

1.2. 本地事务

@Transational

大多数场景下，我们的应用都只需要操作单一的数据库，这种情况下的事务称之为本地事务(`Local Transaction`)。本地事务的ACID特性是数据库直接提供支持。本地事务应用架构如下所示：



在JDBC编程中，我们通过`java.sql.Connection`对象来开启、关闭或者提交事务。代码如下所示：

```
Connection conn = ... //获取数据库连接
conn.setAutoCommit(false); //开启事务
try{
    //...执行增删改查sql
    conn.commit(); //提交事务
}catch (Exception e) {
  conn.rollback();//事务回滚
}finally{
    conn.close();//关闭链接
}
```

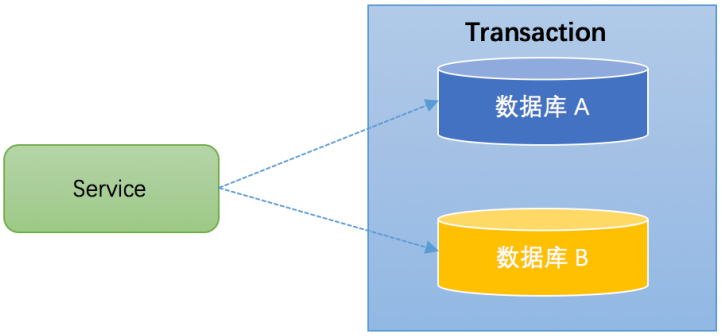
1.3. 分布式事务典型场景

当下互联网发展如火如荼，绝大部分公司都进行了数据库拆分和服务化(SOA)。在这种情况下，完成某一个业务功能可能需要横跨多个服务，操作多个数据库。这就涉及到到了分布式事务，用需要操作的资源位于多个资源服务器上，而应用需要保证对于多个资源服务器的数据的操作，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资源服务器的数据一致性。

典型的分布式事务场景：

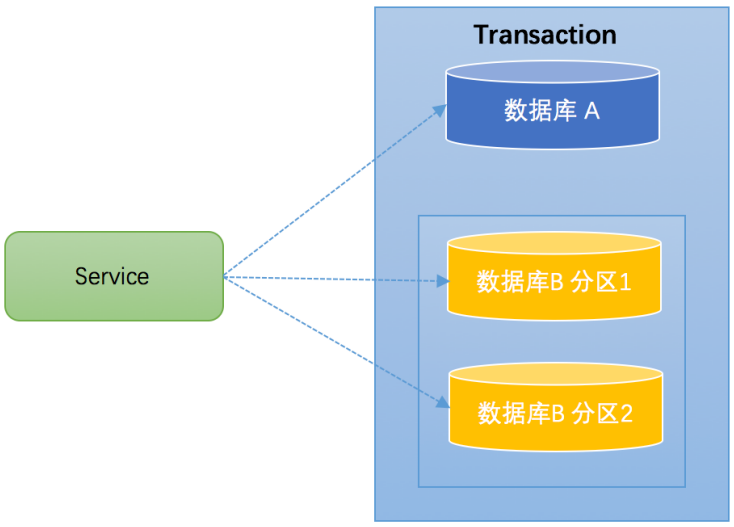
- **跨库事务**

跨库事务指的是，一个应用某个功能需要操作多个库，不同的库中存储不同的业务数据。笔者见过一个相对比较复杂的业务，一个业务中同时操作了9个库。下图演示了一个服务同时操作2个库的情况：



- **分库分表**

通常一个库数据量比较大或者预期未来的数据量比较大，都会进行水平拆分，也就是分库分表。如下图，将数据库B拆分成了2个库：

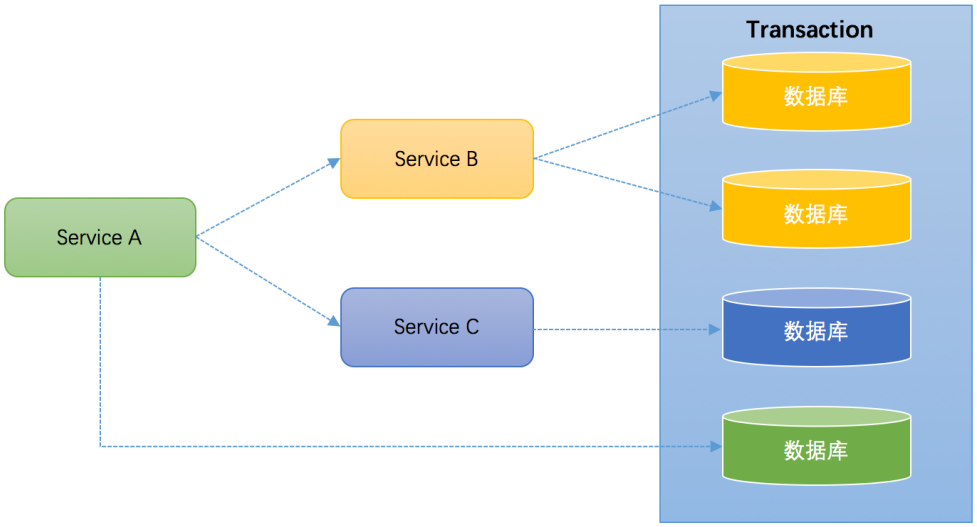


对于分库分表的情况，一般开发人员都会使用一些数据库中间件来降低sql操作的复杂性。如，对于sql：insert into user(id,name) values (1,"张三"),(2,"李四")。这条sql是操作单库的语法，单库情况下，可以保证事务的一致性。

但是由于现在进行了分库分表，开发人员希望将1号记录插入分库1，2号记录插入分库2。所以数据库中间件要将其改写为2条sql，分别插入两个不同的分库，此时要保证两个库要不都成功，要不都失败，因此基本上所有的数据库中间件都面临着分布式事务的问题。

- **服务化**

微服务架构是目前一个比较一个比较火的概念。例如上面笔者提到的一个案例，某个应用同时操作了9个库，这样的应用业务逻辑必然非常复杂，对于开发人员是极大的挑战，应该拆分成不同的独立服务，以简化业务逻辑。拆分后，独立服务之间通过RPC框架来进行远程调用，实现彼此的通信。下图演示了一个3个服务之间彼此调用的架构：



Service A完成某个功能需要直接操作数据库，同时需要调用Service B和Service C，而Service B又同时操作了2个数据库，Service C也操作了一个库。需要保证这些跨服务的对多个数据库的操作要不都成功，要不都失败，实际上这可能是最典型的分布式事务场景。

小结：上述讨论的分布式事务场景中，无一例外的都直接或者间接的操作了多个数据库。如何保证事务的ACID特性，对于分布式事务实现方案而言，是非常大的挑战。同时，分布式事务实现方案还必须要考虑性能的问题，如果为了严格保证ACID特性，导致性能严重下降，那么对于一些要求快速响应的业务，是无法接受的。

常见分布式事务解决方案

1、seata 阿里分布式事务框架

2、消息队列

3、saga

4、XA

他们有一个共同点，都是“两阶段(2PC)”。“两阶段”是指完成整个分布式事务，划分成两个步骤完成。

实际上，这四种常见的分布式事务解决方案，分别对应着分布式事务的四种模式：AT、TCC、Saga、XA；四种分布式事务模式，都有各自的理论基础，分别在不同的时间被提出；每种模式都有它的适用场景，同样每个模式也都诞生有各自的代表产品；而这些代表产品，可能就是我们常见的(全局事务、基于可靠消息、最大努力通知、TCC)。

今天，我们会分别来看4种模式（AT、TCC、Saga、XA）的分布式事务实现。

在看具体实现之前，先讲下分布式事务的理论基础。

分布式事务理论基础

解决分布式事务，也有相应的规范和协议。分布式事务相关的协议有2PC、3PC。

由于三阶段提交协议3PC非常难实现，目前市面主流的分布式事务解决方案都是2PC协议。这就是文章开始提及的常见分布式事务解决方案里面，那些列举的都有一个共同点“两阶段”的内在原因。

有些文章分析2PC时，几乎都会用TCC两阶段的例子，第一阶段try，第二阶段完成confirm或cancel。其实2PC并不是专为实现TCC设计的，2PC具有普适性——协议一样的存在，目前绝大多数分布式解决方案都是以两阶段提交协议2PC为基础的。

TCC（Try-Confirm-Cancel） 实际上是服务化的两阶段提交协议。

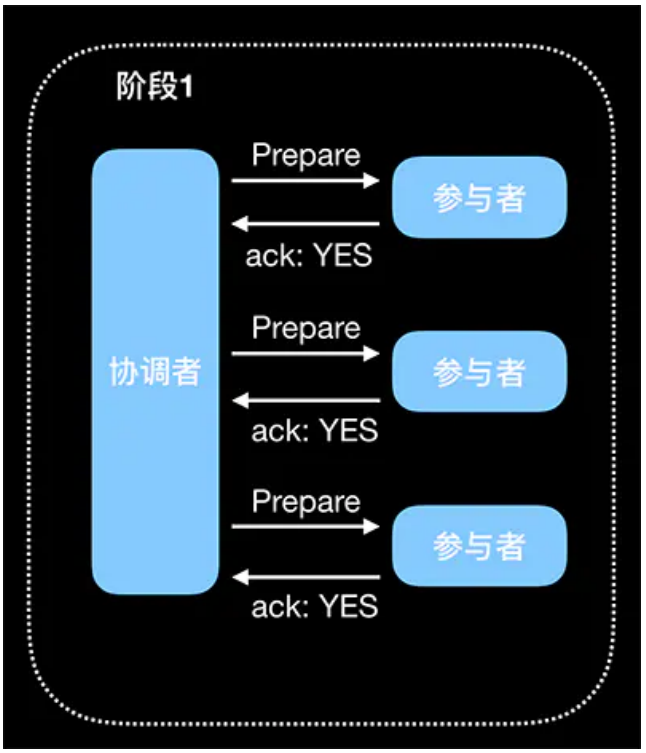
2PC两阶段提交协议：

2PC(两阶段提交，Two-Phase Commit)

顾名思义，分为两个阶段：Prepare 和 Commit

Prepare：提交事务请求

基本流程如下图：



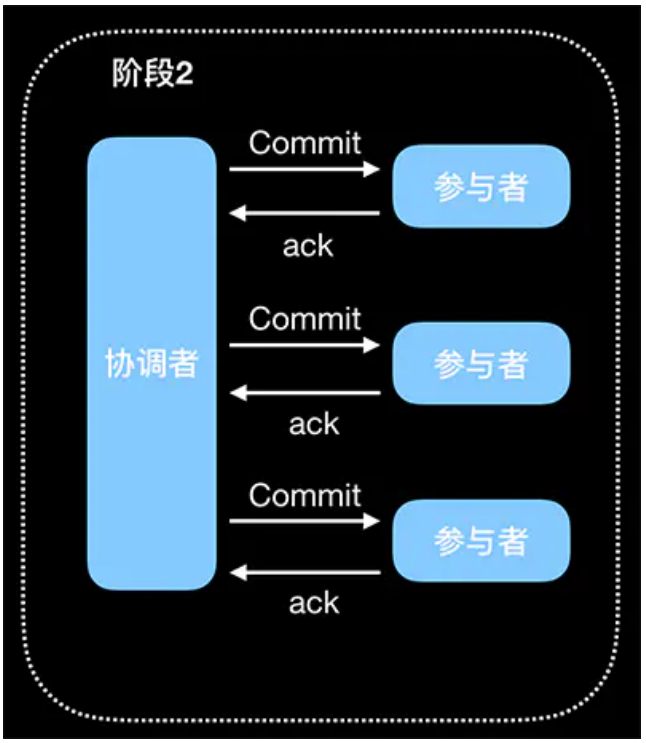
1. 询问 协调者向所有参与者发送事务请求，询问是否可执行事务操作，然后等待各个参与者的响应。
2. 执行 各个参与者接收到协调者事务请求后，执行事务操作(例如更新一个关系型数据库表中的记录)，并将 Undo 和 Redo 信息记录事务日志中。
3. 响应 如果参与者成功执行了事务并写入 Undo 和 Redo 信息，则向协调者返回 YES 响应，否则返回 NO 响应。当然，参与者也可能宕机，从而不会返回响应。

Commit：执行事务提交

执行事务提交分为两种情况，正常提交和回退。

正常提交事务

流程如下图：



1. commit 请求 协调者向所有参与者发送 Commit 请求。

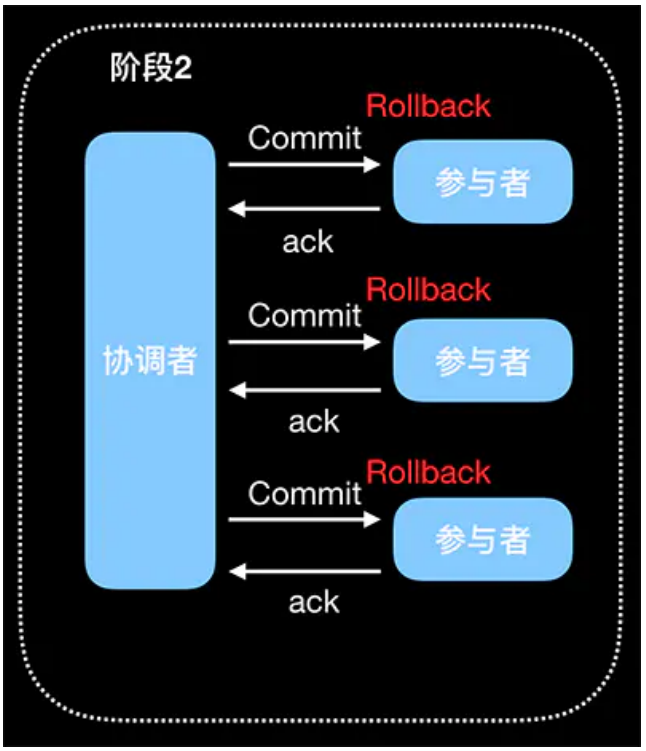
2. 事务提交 参与者收到 Commit 请求后，执行事务提交，提交完成后释放事务执行期占用的所有资源。

3. 反馈结果 参与者执行事务提交后向协调者发送 Ack 响应。

4. 完成事务 接收到所有参与者的 Ack 响应后，完成事务提交。

中断事务

在执行 Prepare 步骤过程中，如果某些参与者执行事务失败、宕机或与协调者之间的网络中断，那么协调者就无法收到所有参与者的 YES 响应，或者某个参与者返回了 No 响应，此时，协调者就会进入回退流程，对事务进行回退。流程如下图红色部分(将 Commit 请求替换为红色的 Rollback 请求)：



1. rollback 请求 协调者向所有参与者发送 Rollback 请求。

2. 事务回滚 参与者收到 Rollback 后，使用 Prepare 阶段的 Undo 日志执行事务回滚，完成后释放事务执行期占用的所有资源。

3. 反馈结果 参与者执行事务回滚后向协调者发送 Ack 响应。

4. 中断事务 接收到所有参与者的 Ack 响应后，完成事务中断。

2PC 的问题

1. 同步阻塞 参与者在等待协调者的指令时，其实是在等待其他参与者的响应，在此过程中，参与者是无法进行其他操作的，也就是阻塞了其运行。 倘若参与者与协调者之间网络异常导致参与者一直收不到协调者信息，那么会导致参与者一直阻塞下去。

2. 单点 在 2PC 中，一切请求都来自协调者，所以协调者的地位是至关重要的，如果协调者宕机，那么就会使参与者一直阻塞并一直占用事务资源。

如果协调者也是分布式，使用选主方式提供服务，那么在一个协调者挂掉后，可以选取另一个协调者继续后续的服务，可以解决单点问题。但是，新协调者无法知道上一个事务的全部状态信息(例如已等待 Prepare 响应的时长等)，所以也无法顺利处理上一个事务。

3. 数据不一致 Commit 事务过程中 Commit 请求/Rollback 请求可能因为协调者宕机或协调者与参与者网络问题丢失，那么就导致了部分参与者没有收到 Commit/Rollback 请求，而其他参与者则正常收到执行了 Commit/Rollback 操作，没有收到请求的参与者则继续阻塞。这时，参与者之间的数据就不再一致了。

当参与者执行 Commit/Rollback 后会向协调者发送 Ack，然而协调者不论是否收到所有的参与者的 Ack，该事务也不会再有其他补救措施了，协调者能做的也就是等待超时后像事务发起者返回一个“我不确定该事务是否成

功”。

4. 环境可靠性依赖 协调者 Prepare 请求发出后，等待响应，然而如果有参与者宕机或与协调者之间的网络中断，都会导致协调者无法收到所有参与者的响应，那么在 2PC 中，协调者会等待一定时间，然后超时后，会触发事务中断，在这个过程中，协调者和所有其他参与者都是出于阻塞的。这种机制对网络问题常见的现实环境来说太苛刻了。

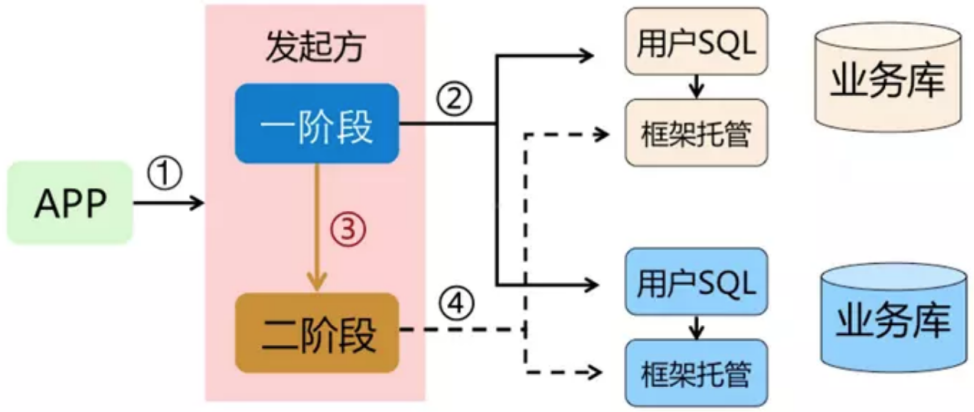
下面我们分别来看4种模式（AT、TCC、Saga、XA）的分布式事务实现。

AT模式(auto transcation)

AT 模式是一种无侵入的分布式事务解决方案。

阿里seata框架，实现了该模式。

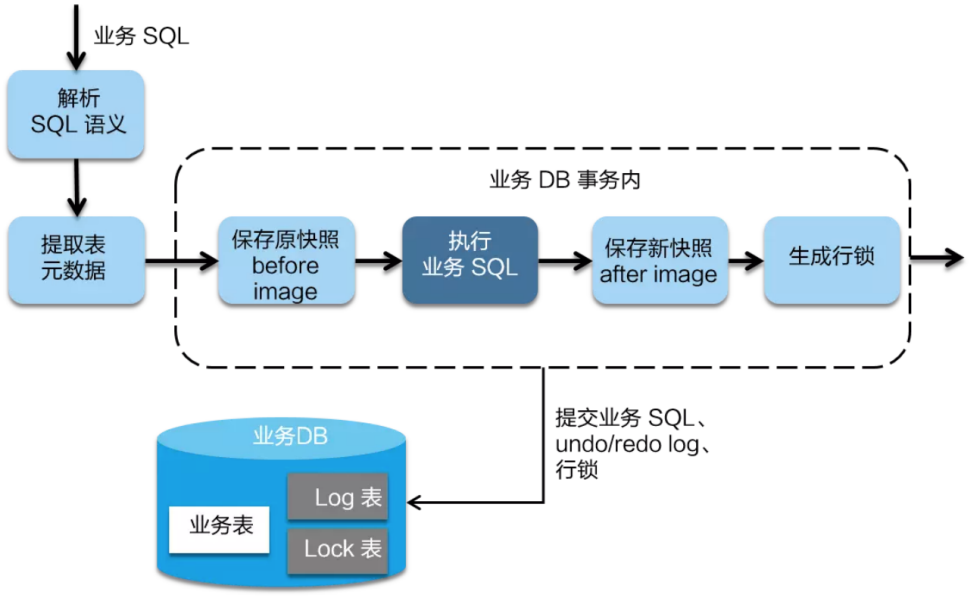
在 AT 模式下，用户只需关注自己的“业务 SQL”，用户的 “业务 SQL” 作为一阶段，Seata 框架会自动生成事务的二阶段提交和回滚操作。



AT 模式如何做到对业务的无侵入：

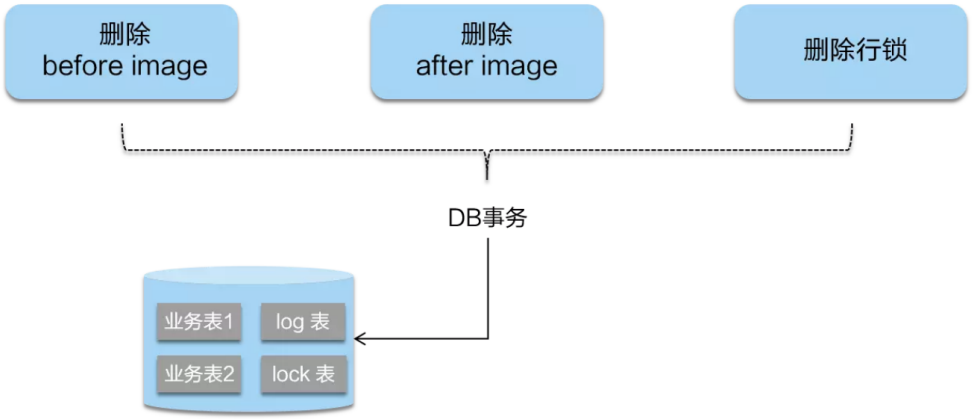
- 一阶段：

在一阶段，Seata 会拦截“业务 SQL”，首先解析 SQL 语义，找到“业务 SQL”要更新的业务数据，在业务数据被更新前，将其保存成“before image”，然后执行“业务 SQL”更新业务数据，在业务数据更新之后，再将其保存成“after image”，最后生成行锁。以上操作全部在一个数据库事务内完成，这样保证了一阶段操作的原子性。



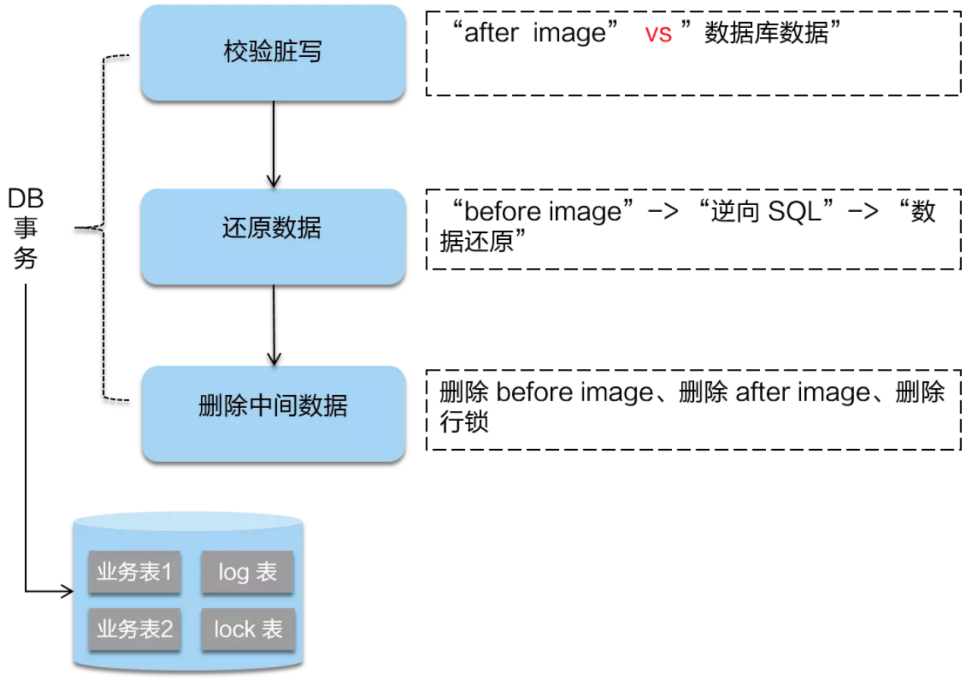
- 二阶段提交：

二阶段如果是提交的话，因为“业务 SQL”在一阶段已经提交至数据库， 所以 Seata 框架只需将一阶段保存的快照数据和行锁删掉，完成数据清理即可。



- 二阶段回滚：

二阶段如果是回滚的话，Seata 就需要回滚一阶段已经执行的“业务 SQL”，还原业务数据。回滚方式便是用“before image”还原业务数据；但在还原前要首先要校验脏写，对比“数据库当前业务数据”和 “after image”，如果两份数据完全一致就说明没有脏写，可以还原业务数据，如果不一致就说明有脏写，出现脏写就需要转人工处理。



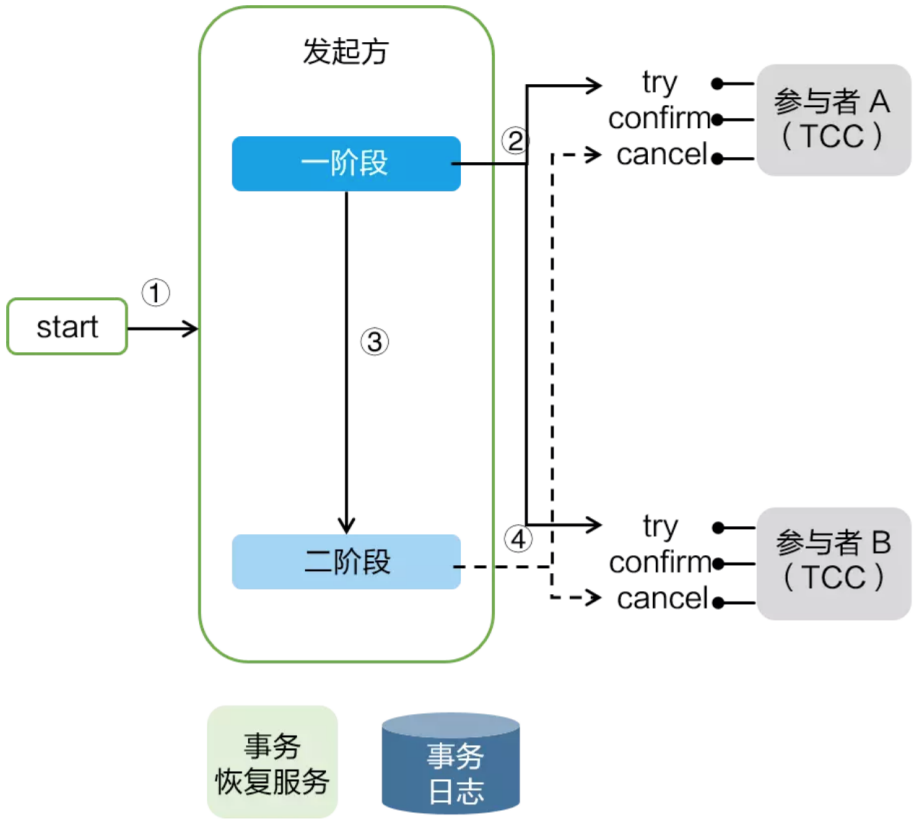
AT 模式的一阶段、二阶段提交和回滚均由 Seata 框架自动生成，用户只需编写“业务 SQL”，便能轻松接入分布式事务，AT 模式是一种对业务无任何侵入的分布式事务解决方案。

TCC 模式

1. 侵入性比较强， 并且得自己实现相关事务控制逻辑

2.在整个过程基本没有锁，性能更强

TCC 模式需要用户根据自己的业务场景实现 Try、Confirm 和 Cancel 三个操作；事务发起方在一阶段执行 Try 方式，在二阶段提交执行 Confirm 方法，二阶段回滚执行 Cancel 方法。



TCC 三个方法描述：

- Try：资源的检测和预留；
- Confirm：执行的业务操作提交；要求 Try 成功 Confirm 一定要能成功；
- Cancel：预留资源释放；

TCC 的实践经验

蚂蚁金服TCC实践,总结以下注意事项:

➢业务模型分2阶段设计

➢并发控制

➢允许空回滚

➢防悬挂控制

➢幂等控制

1 TCC 设计 – 业务模型分 2 阶段设计：

用户接入 TCC ，最重要的是考虑如何将自己的业务模型拆成两阶段来实现。

以“扣钱”场景为例，在接入 TCC 前，对 A 账户的扣钱，只需一条更新账户余额的 SQL 便能完成；但是在接入 TCC 之后，用户就需要考虑如何将原来一步就能完成的扣钱操作，拆成两阶段，实现成三个方法，并且保证一阶段 Try 成功的话 二阶段 Confirm 一定能成功。

- 一阶段（Try）：检查余额，预留其中 30 元；

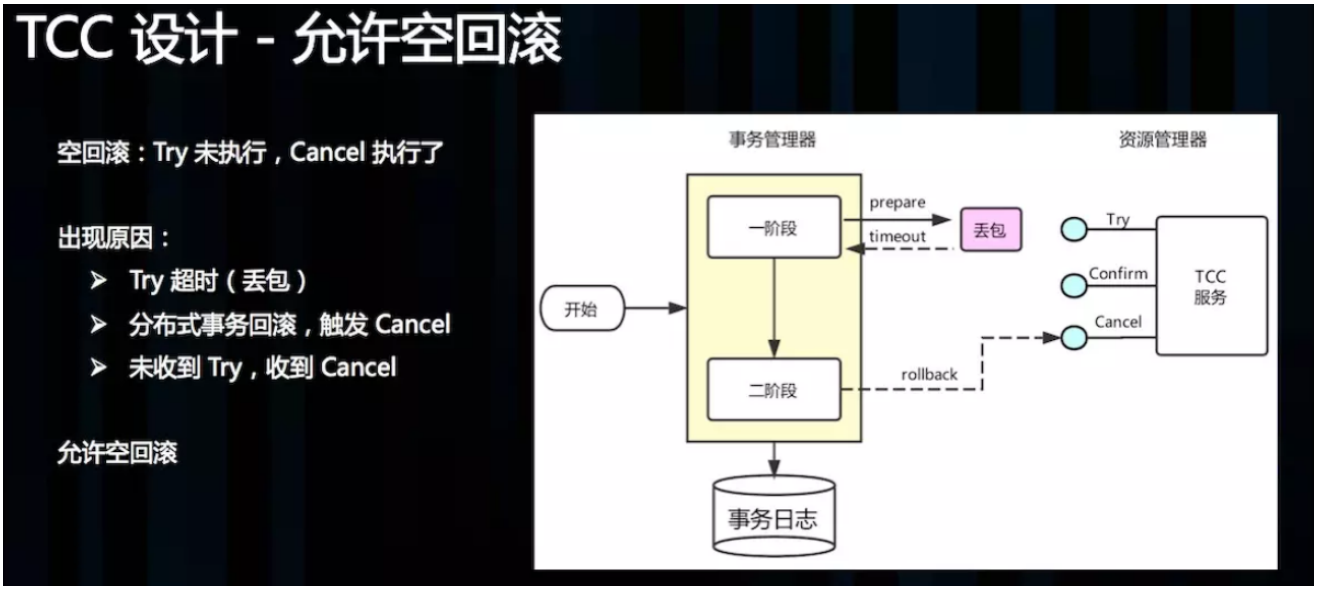
- 二阶段提交（Confirm）：扣除 30 元；

- 二阶段回滚（Cancel）：释放预留的 30 元。

如上图所示，Try 方法作为一阶段准备方法，需要做资源的检查和预留。在扣钱场景下，Try 要做的事情是就是检查账户余额是否充足，预留转账资金，预留的方式就是冻结 A 账户的 转账资金。Try 方法执行之后，账号 A 余额虽然还是 100，但是其中 30 元已经被冻结了，不能被其他事务使用。二阶段 Confirm 方法执行真正的扣钱操作。Confirm 会使用 Try 阶段冻结的资金，执行账号扣款。Confirm 方法执行之后，账号 A 在一阶段中冻结的 30 元已经被扣除，账号 A 余额变成 70 元 。如果二阶段是回滚的话，就需要在 Cancel 方法内释放一阶段 Try 冻结的 30 元，使账号 A 的回到初始状态，100 元全部可用。

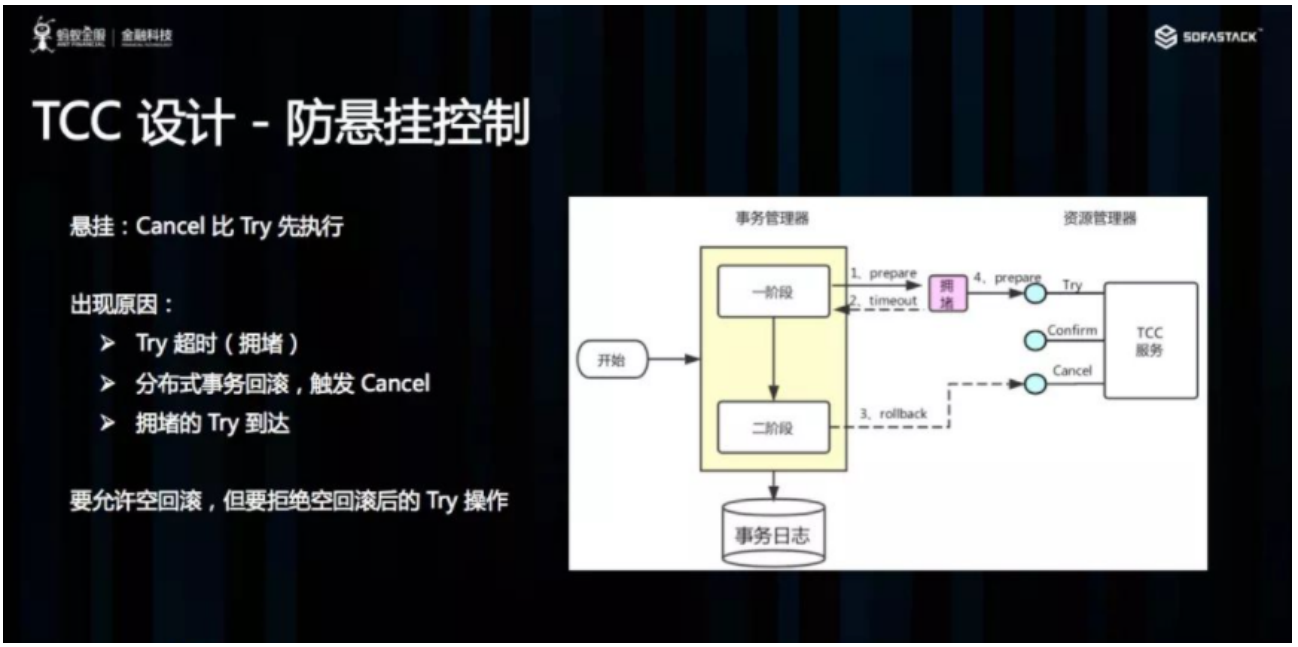
用户接入 TCC 模式，最重要的事情就是考虑如何将业务模型拆成 2 阶段，实现成 TCC 的 3 个方法，并且保证 Try 成功 Confirm 一定能成功。相对于 AT 模式，TCC 模式对业务代码有一定的侵入性，但是 TCC 模式无 AT 模式的全局行锁，TCC 性能会比 AT 模式高很多。

2 TCC 设计 – 允许空回滚：



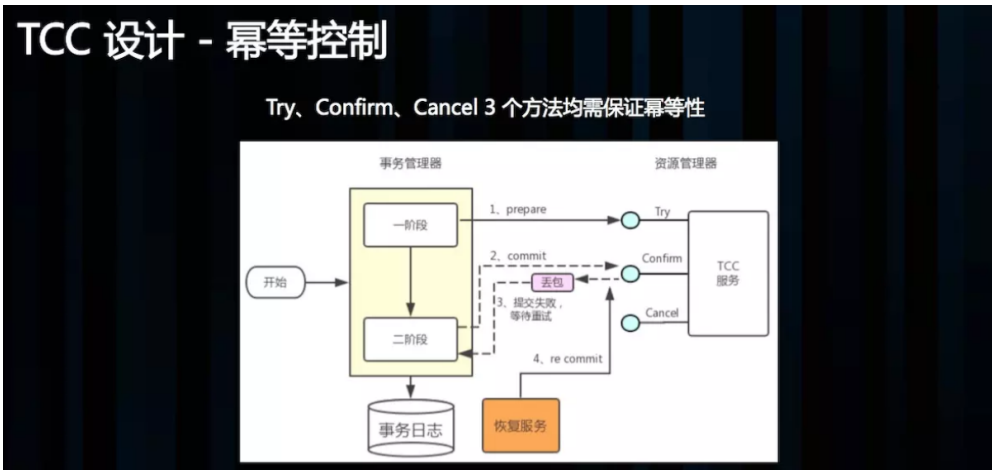
Cancel 接口设计时需要允许空回滚。在 Try 接口因为丢包时没有收到，事务管理器会触发回滚，这时会触发 Cancel 接口，这时 Cancel 执行时发现没有对应的事务 xid 或主键时，需要返回回滚成功。让事务服务管理器认为已回滚，否则会不断重试，而 Cancel 又没有对应的业务数据可以进行回滚。

3 TCC 设计 – 防悬挂控制：



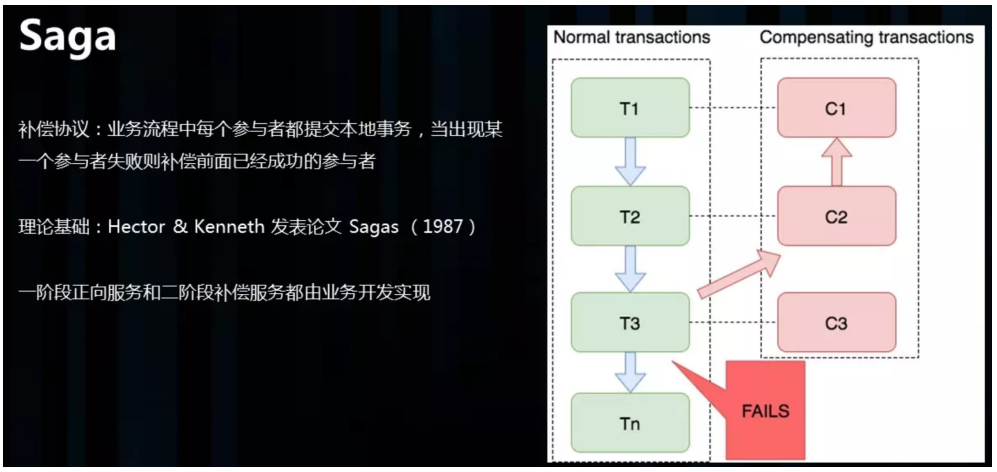
悬挂的意思是：Cancel 比 Try 接口先执行，出现的原因是 Try 由于网络拥堵而超时，事务管理器生成回滚，触发 Cancel 接口，而最终又收到了 Try 接口调用，但是 Cancel 比 Try 先到。按照前面允许空回滚的逻辑，回滚会返回成功，事务管理器认为事务已回滚成功，则此时的 Try 接口不应该执行，否则会产生数据不一致，所以我们在 Cancel 空回滚返回成功之前先记录该条事务 xid 或业务主键，标识这条记录已经回滚过，Try 接口先检查这条事务xid或业务主键如果已经标记为回滚成功过，则不执行 Try 的业务操作。

4 TCC 设计 – 幂等控制：



幂等性的意思是：对同一个系统，使用同样的条件，一次请求和重复的多次请求对系统资源的影响是一致的。因为网络抖动或拥堵可能会超时，事务管理器会对资源进行重试操作，所以很可能一个业务操作会被重复调用，为了不因为重复调用而多次占用资源，需要对服务设计时进行幂等控制，通常我们可以用事务 xid 或业务主键判重来控制。

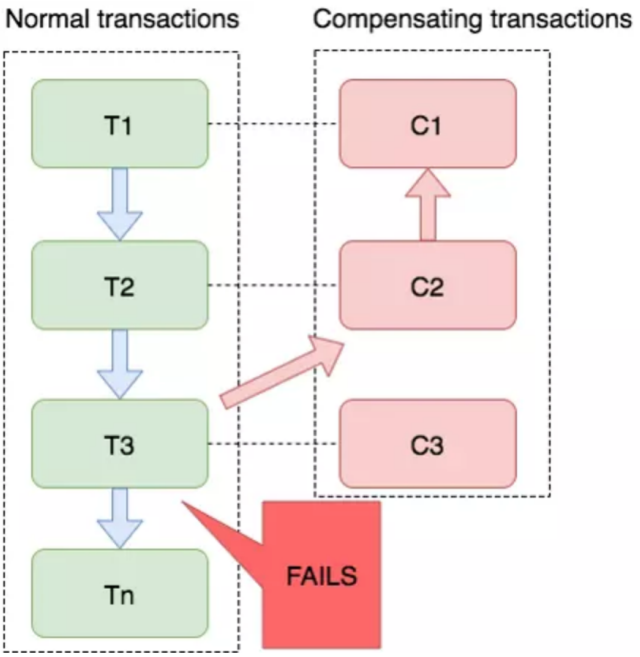
saga模式



Saga 理论出自 Hector & Kenneth 1987发表的论文 Sagas。

saga模式的实现，是长事务解决方案。

Saga 是一种补偿协议，在 Saga 模式下，分布式事务内有多个参与者，每一个参与者都是一个冲正补偿服务，需要用户根据业务场景实现其正向操作和逆向回滚操作。



如图：T1~T3都是正向的业务流程，都对应着一个冲正逆向操作C1~C3

分布式事务执行过程中，依次执行各参与者的正向操作，如果所有正向操作均执行成功，那么分布式事务提交。如果任何一个正向操作执行失败，那么分布式事务会退回去执行前面各参与者的逆向回滚操作，回滚已提交的参与者，使分布式事务回到初始状态。

Saga 正向服务与补偿服务也需要业务开发者实现。因此是业务入侵的。

Saga 模式下分布式事务通常是由事件驱动的，各个参与者之间是异步执行的，Saga 模式是一种长事务解决方案。

Saga 模式使用场景

Saga 模式适用于业务流程长且需要保证事务最终一致性的业务系统，Saga 模式一阶段就会提交本地事务，无锁、长流程情况下可以保证性能。

事务参与者可能是其它公司的服务或者是遗留系统的服务，无法进行改造和提供 TCC 要求的接口，可以使用 Saga 模式。

Saga模式的优势是：

- 一阶段提交本地数据库事务，无锁，高性能；
- 参与者可以采用事务驱动异步执行，高吞吐；
- 补偿服务即正向服务的“反向”，易于理解，易于实现；

缺点：Saga 模式由于一阶段已经提交本地数据库事务，且没有进行“预留”动作，所以不能保证隔离性。后续会讲到对于缺乏隔离性的应对措施。

与TCC实践经验相同的是，Saga 模式中，每个事务参与者的冲正、逆向操作，需要支持：

- 空补偿：逆向操作早于正向操作时；
- 防悬挂控制：空补偿后要拒绝正向操作
- 幂等

XA模式

XA是X/Open DTP组织（X/Open DTP group）定义的两阶段提交协议，XA被许多数据库（如Oracle、DB2、SQL Server、MySQL）和中间件等工具(如CICS 和 Tuxedo)本地支持 。

X/Open DTP模型（1994）包括应用程序（AP）、事务管理器（TM）、资源管理器（RM）。

XA接口函数由数据库厂商提供。XA规范的基础是两阶段提交协议2PC。

JTA(Java Transaction API) 是Java实现的XA规范的增强版 接口。

在XA模式下，需要有一个[全局]协调器，每一个数据库事务完成后，进行第一阶段预提交，并通知协调器，把结果给协调器。协调器等所有分支事务操作完成、都预提交后，进行第二步；第二步：协调器通知每个数据库进行逐个commit/rollback。

其中，这个全局协调器就是XA模型中的TM角色，每个分支事务各自的数据库就是RM。

MySQL 提供的XA实现（https://dev.mysql.com/doc/refman/5.7/en/xa.html ）

XA模式下的 开源框架有atomikos，其开发公司也有商业版本。

XA模式缺点：事务粒度大。高并发下，系统可用性低。因此很少使用。

（AT、TCC、Saga、XA）模式分析

四种分布式事务模式，分别在不同的时间被提出，每种模式都有它的适用场景

- AT 模式是无侵入的分布式事务解决方案，适用于不希望对业务进行改造的场景，几乎0学习成本。
- TCC 模式是高性能分布式事务解决方案，适用于核心系统等对性能有很高要求的场景。
- Saga 模式是长事务解决方案，适用于业务流程长且需要保证事务最终一致性的业务系统，Saga 模式一阶段就会提交本地事务，无锁，长流程情况下可以保证性能，多用于渠道层、集成层业务系统。事务参与者可能是其它公司的服务或者是遗留系统的服务，无法进行改造和提供 TCC 要求的接口，也可以使用 Saga 模式。
- XA模式是分布式强一致性的解决方案，但性能低而使用较少。

总结

分布式事务本身就是一个技术难题，业务中具体使用哪种方案还是需要不同的业务特点自行选择，但是我们也会发现，分布式事务会大大的提高流程的复杂度，会带来很多额外的开销工作，**「代码量上去了，业务复杂了，性能下跌了」**。

所以，当我们真实开发的过程中，能不使用分布式事务就不使用。