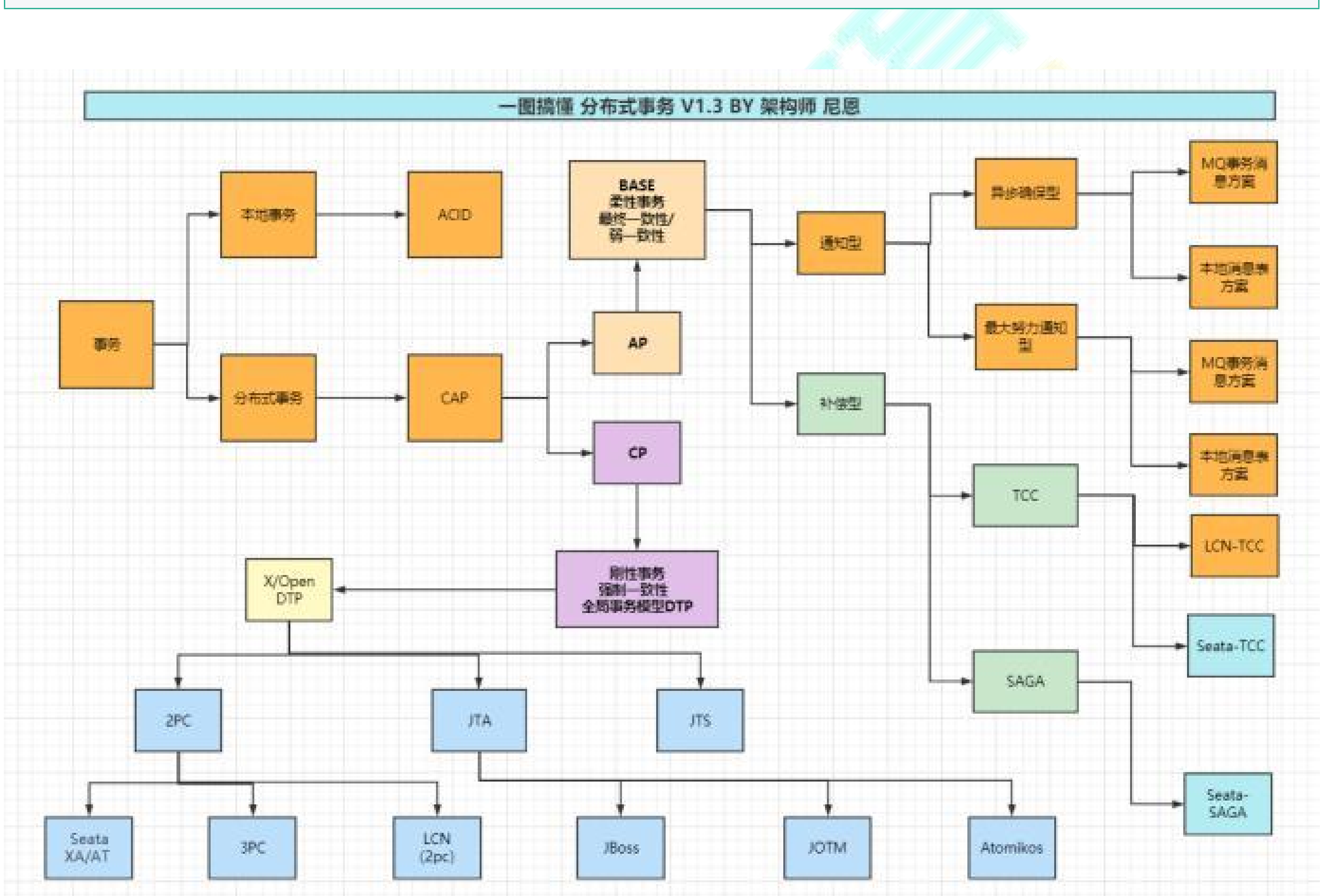
友情提示：

看完此文，在分布式事务这块，基本可以做到吊打面试官了。

一图解读分布式事务

首先奉上一张全网最为牛逼的图，给大家做个总览：



名词解释

- 事务：事务是由□ 组操作构成的可靠的□□ 的□ 作单元，事务具备ACID的特性，即原子性、一致性、隔离性和持久性。
- 本地事务：当事务由资源管理器本地管理时被称作本地事务。本地事务

的优点就是支持严格的ACID特性，高效，可靠，状态可以只在资源管理器中维护，而且应用编程模型简单。但是本地事务不具备分布式事务的处理能力，隔离的最小单位受限于资源管理器。

- 全局事务：当事务由全局事务管理器进行全局管理时成为全局事务，事务管理器负责管理全局的事务状态和参与的资源，协同资源的一致提交回滚。
- TX协议：应用或者应用服务器与事务管理器的接口。
- XA协议：全局事务管理器与资源管理器的接口。 XA是由X/Open组织提出的分布式事务规范。该规范主要定义了全局事务管理器和局部资源管理器之间的接口。主流的数据库产品都实现了XA接口。 XA接口是一个双向的系统接口，在事务管理器以及多个资源管理器之间作为通信桥梁。之所以需要XA是因为在分布式系统中从理论上讲两台机器是无法达到一致性状态的，因此引入一个单点进行协调。由全局事务管理器管理和协调的事务可以跨越多个资源和进程。全局事务管理器一般使用XA二阶段协议与数据库进行交互。
- AP：应用程序，可以理解为使用DTP（Data Tools Platform）的程序。
- RM：资源管理器，这里可以是一个DBMS或者消息服务器管理系统，应用程序通过资源管理器对资源进行控制，资源必须实现XA定义的接口。资源管理器负责控制和管理实际的资源。
- TM：事务管理器，负责协调和管理事务，提供给AP编程接口以及管理资源管理器。事务管理器控制着全局事务，管理事务的生命周期，并且协调资源。
- 两阶段提交协议： XA用于在全局事务中协调多个资源的机制。 TM和RM之间采取两阶段提交的方案来解决一致性问题。两节点提交需要一个协调者（TM）来掌控所有参与者（RM）节点的操作结果并且指引这些节点是否需要最终提交。两阶段提交的局限在于协议成本，准备阶段的持久成本，全局事务状态的持久成本，潜在故障点多带来的脆弱性，准备后，提交前的故障引发一系列隔离与恢复难题。
- BASE理论： BA指的是基本业务可用性，支持分区失败， S表示柔性状

态，也就是允许短时间内不同步， E表示最终一致性，数据最终是一致的，但是实时是不一致的。原子性和持久性必须从根本上保障，为了可用性、性能和服务降级的需要，只有降低一致性和隔离性的要求。

- CAP定理：对于共享数据系统，最多只能同时拥有CAP其中的两个，任意两个都有其适应的场景，真是的业务系统中通常是ACID与CAP的混合体。分布式系统中最重要的是满足业务需求，而不是追求高度抽象，绝对的系统特性。 C表示一致性，也就是所有用户看到的数据是一样的。 A表示可用性，是指总能找到一个可用的数据副本。 P表示分区容错性，能够容忍网络中断等故障。

分布式事务与分布式锁的区别：

> 分布式锁解决的是分布式资源抢占的问题；分布式事务和本地事务是解决流程化提交问题。

事务简介

事务(Transaction)是操作数据库中某个数据项的一个程序执行单元(unit)。

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。

事务的四个特征：

1、Atomic原子性

事务必须是一个原子的操作序列单元，事务中包含的各项操作在一次执行过程中，要么全部执行成功，要么全部不执行，任何一项失败，整个事务回滚，只有全部都执行成功，整个事务才算成功。

2、Consistency一致性

事务的执行不能破坏数据库数据的完整性和一致性，事务在执行之前和之后，数据库都必须处于一致性状态。

3、Isolation隔离性

在并发环境中，并发的事务是相互隔离的，一个事务的执行不能被其他事务干扰。即不同的事务并发操纵相同的数据时，每个事务都有各自完整的数据空间，即一个事务内部的操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能相互干扰。

SQL中的4个事务隔离级别：

（1）读未提交

> 允许脏读。如果一个事务正在处理某一数据，并对其进行了更新，但同时尚未完成事务，因此事务没有提交，与此同时，允许另一个事务也能够访问该数据。例如A将变量n从0累加到10才提交事务，此时B可能读到n变量从0到10之间的所有中间值。

（2）读已提交

允许不可重复读。只允许读到已经提交的数据。即事务A在将n从0累加到10的过程中， B无法看到n的中间值，之中只能看到10。同时有事务C进行从10到20的累加，此时B在同一个事务内再次读时，读到的是20。

（3）可重复读

允许幻读。保证在事务处理过程中，多次读取同一个数据时，其值都和事务开始时刻时是一致的。禁止脏读、不可重复读。幻读即同样的事务操作，在前后两个时间段内执行对同一个数据项的读取，可能出现不一致的结果。保证B在同一个事务内，多次读取n的值，读到的都是初始值0。幻读，就是不同事务，读到的n的数据可能是0，可能10，可能是20

（4）串行化

最严格的事务，要求所有事务被串行执行，不能并发执行。

如果不对事务进行并发控制，我们看看数据库并发操作是会有那些异常情形

- （1）一类丢失更新：两个事物读同一数据，一个修改字段1，一个修改字段2，后提交的恢复了先提交修改的字段。
- （2）二类丢失更新：两个事物读同一数据，都修改同一字段，后提交的覆盖了先提交的修改。
- （3）脏读：读到了未提交的值，万一该事物回滚，则产生脏读。
- （4）不可重复读：两个查询之间，被另外一个事务修改了数据的内容，产生内容的不一致。
- （5）幻读：两个查询之间，被另外一个事务插入或删除了记录，产生结果集的不一致。

4、Durability持久性

持久性（durability）：持久性也称永久性（permanence），指一个事务一旦提交，它对数据库中对应数据的状态变更就应该是永久性的。

即使发生系统崩溃或机器宕机，只要数据库能够重新启动，那么一定能够将其恢复到事务成功结束时的状态。

> 比方说：一个人买东西的时候需要记录在账本上，即使老板忘记了那也有据可查。

MySQL的本地事务实现方案

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。

了解过MySQL事务的同学，就会知道，为了达成本地事务， MySQL做了很多的工作，比如回滚日志，重做日志， MVCC，读写锁等。

MySQL数据库的事务实现原理

以MySQL 的InnoDB （InnoDB 是 MySQL 的一个存储引擎）为例，介绍一下单一数据库的事务实现原理。

InnoDB 是通过 日志和锁 来保证的事务的 ACID特性，具体如下：

（1）通过数据库锁的机制，保障事务的隔离性；

（2）通过 Redo Log （重做日志）来，保障事务的持久性；

（3）通过 Undo Log （撤销日志）来，保障事务的原子性；

（4）通过 Undo Log （撤销日志）来，保障事务的一致性；

Undo Log 如何保障事务的原子性呢?

具体的方式为：在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为 Undo Log），然后进行数据的修改。如果出现了错误或者用户执行了 Rollback 语句，系统可以利用 Undo Log 中的备份将数据恢复到事务开始之前的状态。

Redo Log如何保障事务的持久性呢?

具体的方式为： Redo Log 记录的是新数据的备份（和 Undo Log 相反）。在事务提交前，只要将 Redo Log 持久化即可，不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是 Redo Log 已经持久化。系统可以根据 Redo Log 的内容，将所有数据恢复到崩溃之前的状态。

脏读、幻读、不可重复读

在多个事务并发操作时，数据库中会出现下面三种问题： **脏读，幻读，不可重复读。**

脏读（Dirty Read）

事务A读到了事务B还未提交的数据：

事务A读取的数据，事务B对该数据进行修改还未提交数据之前，事务A再次读取数据会读到事务B已经修改后的数据，如果此时事务B进行回滚或再次修改该数据然后提交，事务A读到的数据就是脏数据，这个情况被称为脏读（Dirty Read）。

脏读	
事务A	事务B
select uage from csdn where uname='wyk'; -- res:28	
	update csdn set uage=30 where uname='wyk';
select uage from csdn where uname='wyk'; -- res:30（脏读）	
	rollback; -- uname:wyk, uage:28 或 update csdn set uage=33 where uname='wyk'; commit; -- uname:wyk, uage:33

幻读（Phantom Read）

事务A进行范围查询时，事务B中新增了满足该范围条件的记录，当事务A再次按该条件进行范围查询，会查到在事务B中提交的新的满足条件的记录（**幻行** Phantom Row）。

幻读	
事务A	事务B
select * from csdn where uage > 25; -- res count:2	
	insert into csdn(uname,uage,ucompany) values ('wyk3','28','csdn');
	commit;
select * from csdn where uage > 25; -- res count:3（幻读） ('wyk1','28','csdn') ('wyk2','28','csdn') ('wyk3','28','csdn')幻行	

不可重复读（Unrepeatable Read）

事务A在读取某些数据后，再次读取该数据，发现读出的该数据已经在事务B中发生了变更或删除。

不可重复读	
事务A	事务B
select * from csdn where uname='wyk3'; res: ('wyk3','28','csdn')	
	update csdn set uage=30 where uname='wyk3';
	commit;
select * from csdn where uname='wyk3'; res: ('wyk3','30','csdn') （不可重复读）	

幻读和不可重复度的区别：

- **幻读**：在同一事务中，相同条件下，两次查询出来的 **记录数** 不一样；
- **不可重复读**：在同一事务中，相同条件下，两次查询出来的 **数据** 不一样；

事务的隔离级别

为了解决数据库中事务并发所产生的问题，在标准SQL规范中，定义了四种事务隔离级别，每一种级别都规定了一个事务中所做的修改，哪些在事务内和事务间是可见的，哪些是不可见的。

> 低级别的隔离级一般支持更高的并发处理，并拥有更低的系统开销。

MySQL事务隔离级别：https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html

通过修改MySQL系统参数来控制事务的隔离级别，在MySQL8中该参数为 **transaction_isolation** ，在MySQL5中该参数为 **tx_isolation** ：

```
MySQL8：
-- 查看系统隔离级别：
SELECT @@global.transaction_isolation;

-- 查看当前会话隔离级别
SELECT @@transaction_isolation;

-- 设置当前会话事务隔离级别
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- 设置全局事务隔离级别
SET GLOBAL TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

事务的四个隔离级别：

- **未提交读（READ UNCOMMITTED）** ：所有事务都可以看到其他事务未提交的修改。一般很少使用；

- **提交读（READ COMMITTED）**：Oracle默认隔离级别，事务之间只能看到彼此已提交的变更修改；
- **可重复读（REPEATABLE READ）**：MySQL默认隔离级别，同一事务中的多次查询会看到相同的数据行；可以解决不可重复读，但可能出现幻读；
- **可串行化（SERIALIZABLE）**：最高的隔离级别，事务串行的执行，前一个事务执行完，后面的事务会执行。读取每条数据都会加锁，会导致大量的超时和锁争用问题；

隔离级别	脏读可能性	不可重复读可能性	幻读可能性	加锁读
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	No
REPEATABLE READ	No	No	Yes	No
SERIALIZABLE	No	No	No	Yes

问：如何保证 REPEATABLE READ 级别绝对不产生幻读?

答：在SQL中加入 for update (排他锁) 或 lock in share mode (共享锁)语句实现。就是锁住了可能造成幻读的数据，阻止数据的写入操作。

分布式事务的基本概念

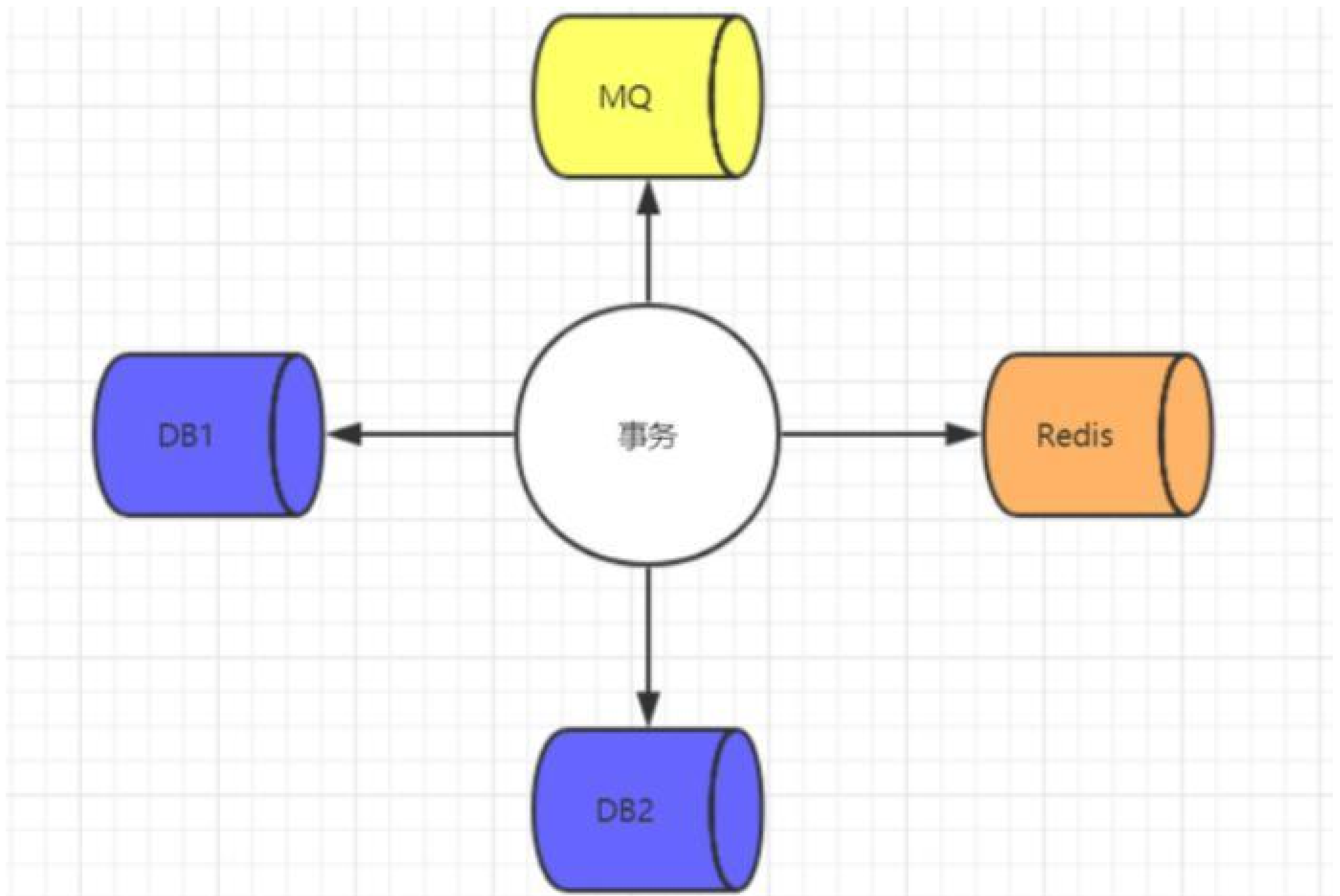
分布式环境的事务复杂性

当本地事务要扩展到分布式时，它的复杂性进一步增加了。

存储端的多样性。

首先就是存储端的多样性。本地事务的情况下，所有数据都会落到同一个DB中，但是，在分布式的情况下，就会出现数据可能要落到多个DB，或者还会落到Redis，落到MQ等中。

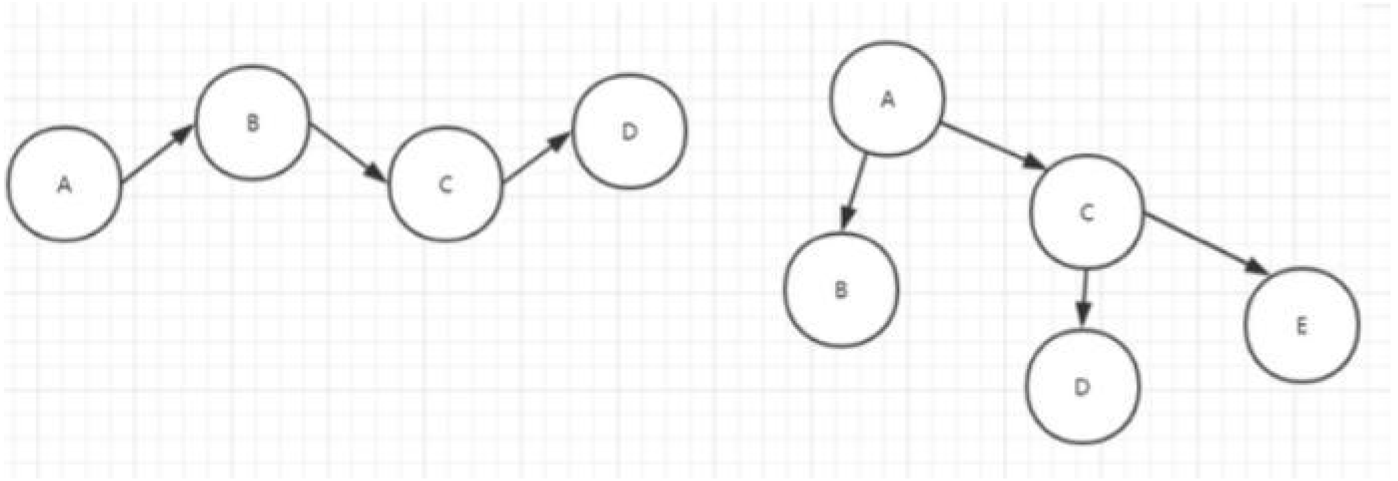
存储端多样性, 如下图所示:



事务链路的延展性

本地事务的情况下，通常所有事务相关的业务操作，会被我们封装到一个Service方法中。而在分布式的情况下，请求链路被延展，拉长，一个操作会被拆分成多个服务，它们呈现线状或网状，依靠网络通信构建成一个整体。在这种情况下，事务无疑变得更复杂。

事务链路延展性, 如下图所示:



基于上述两个复杂性，期望有一个统一的分布式事务方案，能够像本地事务一样，以几乎无侵入的方式，满足各种存储介质，各种复杂链路，是不现实的。

至少，在当前，还没有一个十分成熟的解决方案。所以，一般情况下，在分布式下，事务会被拆分解决，并根据不同的情况，采用不同的解决方案。

什么是分布式事务?

对于分布式系统而言，需要保证分布式系统中的数据一致性，保证数据在子系统中始终保持一致，避免业务出现问题。分布式系统中对数要么一起成功，要么一起失败，必须是一个整体性的事务。

分布式事务指事务的参与者、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，在分布式系统上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节点上，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。

举个例子：在电商网站中，用户对商品进行下单，需要在订单表中创建一

条订单数据，同时需要在库存表中修改当前商品的剩余库存数量，两步操作一个添加，一个修改，我们一定要保证这两步操作一定同时操作成功或失败，否则业务就会出现问题。

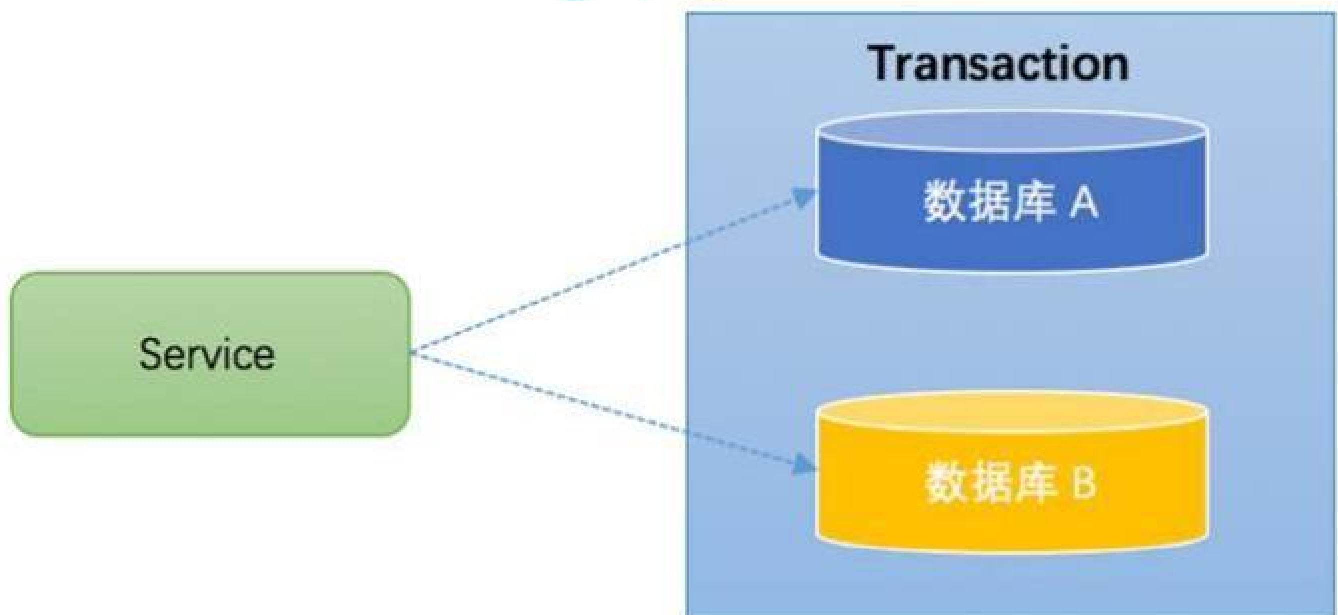
任何事务机制在实现时，都应该考虑事务的ACID特性，包括：本地事务、分布式事务。对于分布式事务而言，即使不能都很好的满足，也要考虑支持到什么程度。

典型的分布式事务场景：

1. 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。

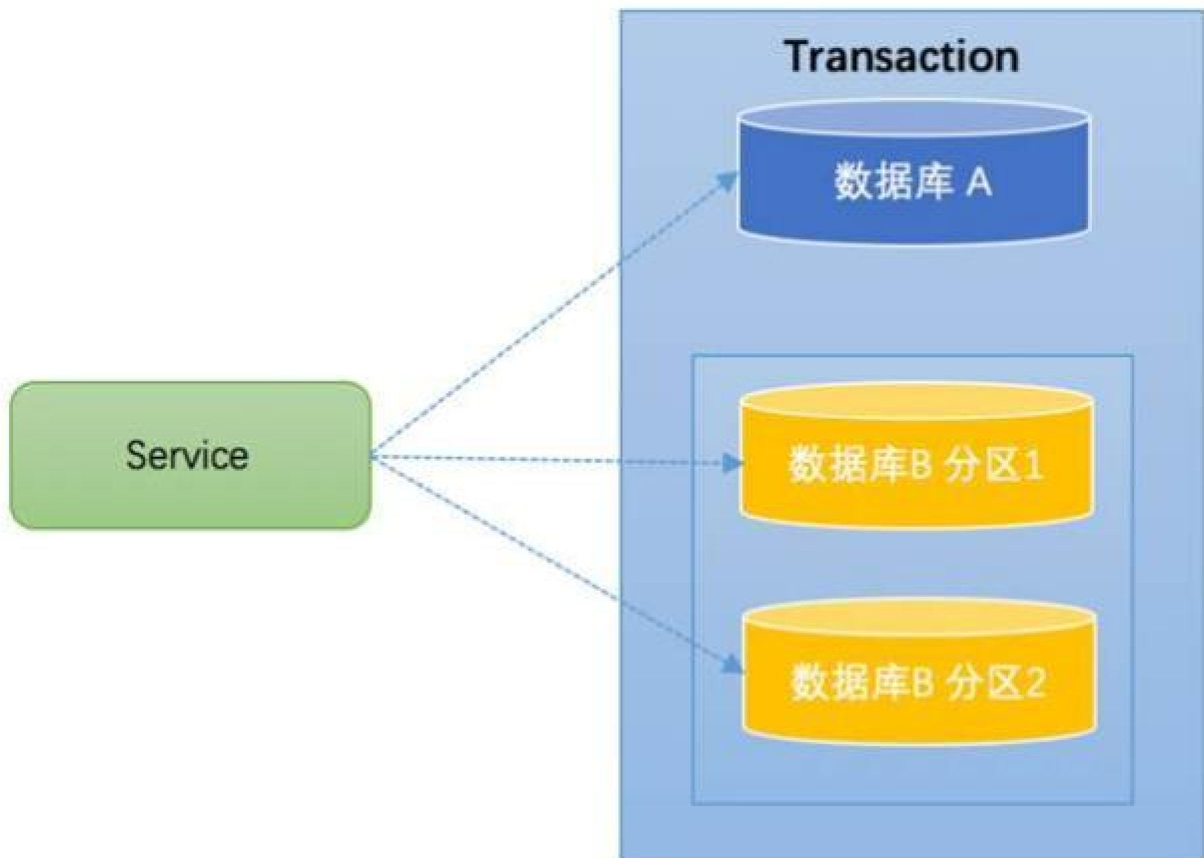
下图演示了一个服务同时操作2个库的情况：



2. 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。

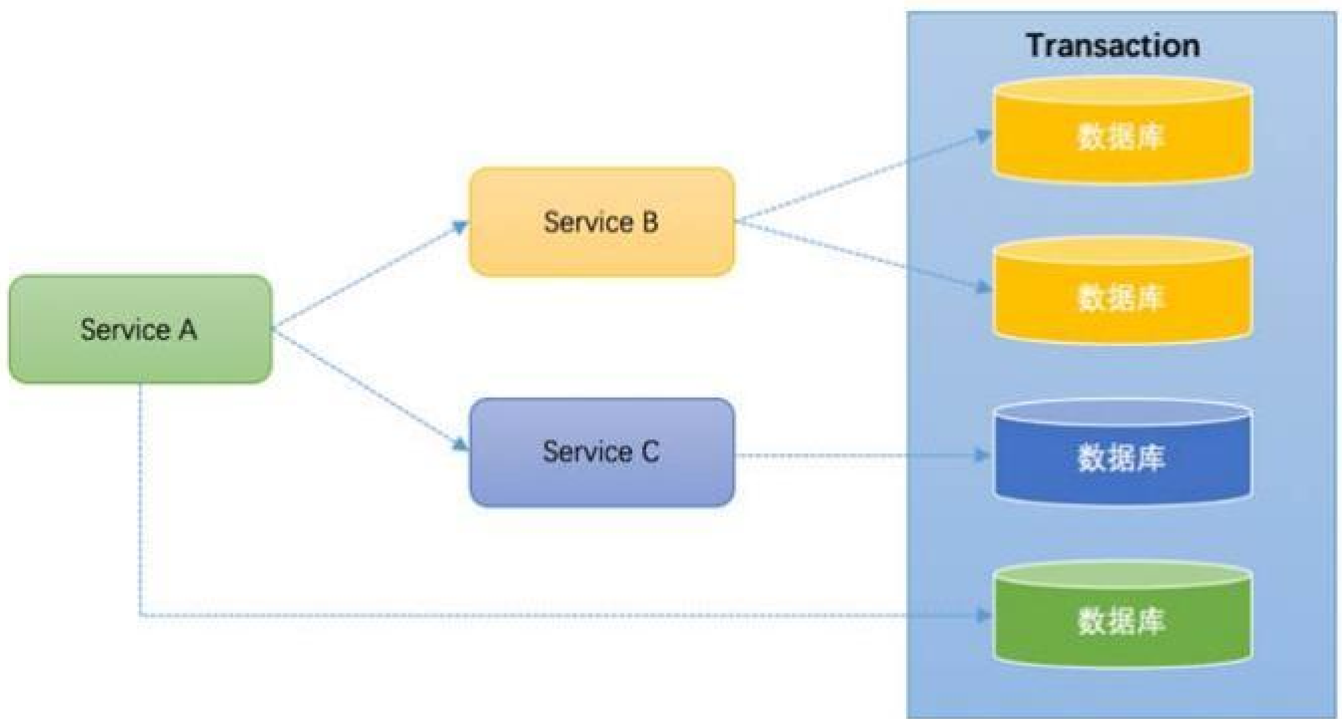
如，对于sql： insert into user(id,name) values (1,"tianshouzhi"),

(2,"wangxiaoxiao")。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。

3. 微服务化

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□ 员是极□ 的挑战，应该拆分成不同的□□ 服务，以简化业务逻辑。拆分后，□□ 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。

分布式事务实现方案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。

CAP定理

分布式事务的理论基础

数据库事务ACID 四大特性，无法满足分布式事务的实际需求，这个时候又有一些新的大牛提出一些新的理论。

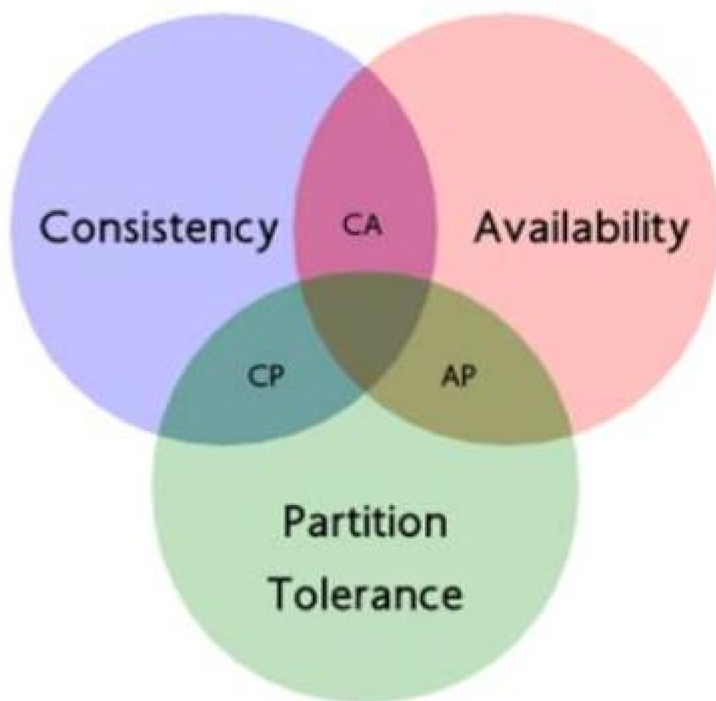
CAP定理

CAP定理是由加州大学伯克利分校Eric Brewer教授提出来的，他指出WEB服务无法同时满足一下3个属性：

- 一致性(Consistency)： 客户端知道一系列的操作都会同时发生(生效)
- 可用性(Availability)： 每个操作都必须以可预期的响应结束
- 分区容错性(Partition tolerance)： 即使出现单个组件无法可用，操作依然可以完成

具体地讲在分布式系统中，一个Web应用至多只能同时支持上面的两个属性。因此，设计人员必须在一致性与可用性之间做出选择。

2000年7月Eric Brewer教授仅仅提出来的是一个猜想， 2年后，麻省理工学院的Seth Gilbert和Nancy Lynch从理论上证明了CAP理论，并且而一个分布式系统最多只能满足CAP中的2项。之后， CAP理论正式成为分布式计算领域的公认定理。



所以， CAP定理在迄今为止的分布式系统中都是适用的！

CAP的一致性、可用性、分区容错性 具体如下：

1、一致性

数据一致性指“all nodes see the same data at the same time”，即更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致，不能存在中间状态。

分布式环境中，一致性是指多个副本之间能否保持一致的特性。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作后，应该保证系统的数据仍然处理一致的状态。

例如对于电商系统用户下单操作，库存减少、用户资金账户扣减、积分增加等操作必须在用户下单操作完成后必须是一致的。不能出现类似于库存已经减少，而用户资金账户尚未扣减，积分也未增加的情况。如果出现了这种情况，那么就认为是不一致的。

数据一致性分为强一致性、弱一致性、最终一致性。

- 如果的确能像上面描述的那样时刻保证客户端看到的数据都是一致的，那么称之为强一致性。
- 如果允许存在中间状态，只要求经过一段时间后，数据最终是一致的，则称之为最终一致性。
- 此外，如果允许存在部分数据不一致，那么就称之为弱一致性。

面试题：什么是数据一致性？　　现在知道怎么回答了吧！

2、可用性

系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。

两个度量的维度：

（1）有限时间内

对于用户的一个操作请求，系统必须能够在指定的时间（响应时间）内返回对应的处理结果，　**如果超过了这个时间范围，那么系统就被认为是不可用的。**即这个响应时间必须在一个合理的值内，不让用户感到失望。

试想，如果一个下单操作，为了保证分布式事务的一致性，需要10分钟才能处理完，那么用户显然是无法忍受的。

（2）返回正常结果

要求系统在完成对用户请求的处理后，返回一个正常的响应结果。正常的响应结果通常能够明确地反映出对请求的处理结果，即成功或失败，而不是一个让用户感到困惑的返回结果。比如返回一个系统错误如OutOfMemory，则认为系统是不可用的。

“返回结果”是可用性的另一个非常重要的指标，它要求系统在完成对用户请求的处理后，返回一个正常的响应结果，不论这个结果是成功还是失败。

3、分区容错性

即分布式系统在遇到任何网络分区故障时，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障。

网络分区，是指分布式系统中，不同的节点分布在不同的子网络（机房/异地网络）中，由于一些特殊的原因导致这些子网络之间出现网络不连通的状态，但各个子网络的内部网络是正常的，从而导致整个系统的网络环境被切分成了若干孤立的区域。组成一个分布式系统的每个节点的加入与退出都可以看做是一个特殊的网络分区。

CAP的应用

1、放弃P

放弃分区容错性的话，则放弃了分布式，放弃了系统的可扩展性

2、放弃A

放弃可用性的话，则在遇到网络分区或其他故障时，受影响的服务需要等待一定的时间，再此期间无法对外提供政策的服务，即不可用

3、放弃C

放弃一致性的话（这里指强一致），则系统无法保证数据保持实时的一致性，在数据达到最终一致性时，有个时间窗口，在时间窗口内，数据是不一致的。

对于分布式系统来说， P是不能放弃的，因此架构师通常是在可用性和一致性之间权衡。

CAP 理论告诉我们:

目前很多大型网站及应用都是分布式部署的，分布式场景中的数据一致性问题一直是一个比较重要的话题。

基于 CAP理论，很多系统在设计之初就要对这三者做出取舍:

> 任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。在互联网领域的绝大多数的场景中，都需要**牺牲强一致性来换取系统的高可用性**，系统往往只需要保证最终一致性。

> **问：为什么分布式系统中无法同时保证一致性和可用性?**

答：首先一个前提，对于分布式系统而言，分区容错性是一个最基本的要求，因此基本上我们在设计分布式系统的时候只能从一致性（C）和可用性（A）之间进行取舍。

如果保证了一致性（C）：对于节点N1和N2，当往N1里写数据时，N2上的操作必须被暂停，只有当N1同步数据到N2时才能对N2进行读写请求，在N2被暂停操作期间客户端提交的请求会收到失败或超时。显然，这与可用性是相悖的。

如果保证了可用性（A）：那就不能暂停N2的读写操作，但同时N1在写数据的话，这就违背了一致性的要求。

CAP 权衡

通过 CAP 理论，我们知道无法同时满足一致性、可用性和分区容错性这三个特性，那要舍弃哪个呢?

对于多数大型互联网应用的场景，主机众多、部署分散，而且现在的集群规模越来越大，所以节点故障、网络故障是常态，而且要保证服务可用性达到 N 个 9，即保证 P 和 A，舍弃C （退而求其次保证最终一致性）。虽然某些地方会影响客户体验，但没达到造成用户流程的严重程度。

对于涉及到钱财这样不能有一丝让步的场景， C 必须保证。网络发生故障宁可停止服务，这是保证 CA，舍弃 P。貌似这几年国内银行业发生了不下 10 起事故，但影响面不大，报道也不多，广大群众知道的少。还有一种是保证 CP，舍弃 A。例如网络故障是只读不写。

CAP和ACID中的A和C是完全不一样的

A的区别：

- ACID中的A指的是原子性(Atomicity)，是指事务被视为一个不可分割的最小工作单元，事务中的所有操作要么全部提交成功，要么全部失败回滚；
- CAP中的A指的是可用性(Availability)，是指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；

C的区别：

- ACID一致性是有关数据库规则，数据库总是从一个一致性的状态转换到另外一个一致性的状态；
- CAP的一致性是分布式多服务器之间复制数据令这些服务器拥有同样的数据，由于网速限制，这种复制在不同的服务器上所消耗的时间是不固定的，集群通过组织客户端查看不同节点上还未同步的数据维持逻辑视图，这是一种分布式领域的一致性概念；

总之：

ACID里的一致性指的是事务执行前后，数据库完整性，而CAP的一致性，指的是分布式节点的数据的一致性。背景不同，无从可比

BASE定理

> CAP是分布式系统设计理论，　BASE是CAP理论中AP方案的延伸，对于C我们采用的方式和策略就是保证最终一致性；

eBay的架构师Dan Pritchett源于对大规模分布式系统的实践总结，在ACM上发表文章提出BASE理论， BASE理论是对CAP理论的延伸，核心思想是即使无法做到强一致性（StrongConsistency，CAP的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）。

BASE定理

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。 BASE基于CAP定理演化而来，核心思想是即时无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。

1、Basically Available （基本可用）

> 基本可用是指分布式系统在出现不可预知的故障的时候，允许损失部分可用性，但不等于系统不可用。

（1）响应时间上的损失

> 当出现故障时，响应时间增加

（2）功能上的损失

当流量高峰期时，屏蔽一些功能的使用以保证系统稳定性（服务降级）

2、Soft state （软状态）

指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性。

与硬状态相对，即是指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。

3、Eventually consistent （最终一致性）

强调系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。其本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

最终一致性可分为如下几种：

- （1）因果一致性（Causal consistency）

  即进程A在更新完数据后通知进程B，那么之后进程B对该项数据的范围都是进程A更新后的最新值。

- （2）读己之所写（Read your writes）

  进程A更新一项数据后，它自己总是能访问到自己更新过的最新值。
- （3）会话一致性（Session consistency）

  将数据一致性框定在会话当中，在一个会话当中实现读己之所写的一致性。即执行更新后，客户端在同一个会话中始终能读到该项数据的最新值
- （4）单调读一致性（Monotonic read consistency）

  如果一个进程从系统中读取出一个数据项的某个值后，那么系统对于该进程后续的任何数据访问都不应该返回更旧的值。
- （5）单调写一致性（Monotoic write consistency）

  一个系统需要保证来自同一个进程的写操作被顺序执行。

BASE理论是提出通过牺牲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。

BASE理论的特点：

BASE理论面向的是大型高可用可扩展的分布式系统，和传统的事物ACID特性是相反的。

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。

但同时，在实际的分布式场景中，不同业务单元和组件对数据一致性的要求是不同的，因此在具体的分布式系统架构设计过程中， ACID特性和BASE理论往往又会结合在一起。

BASE理论与CAP的关系

BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结， 是基于CAP定理逐步演化而来的。 BASE理论的核心思想是： **即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。**

BASE理论其实就是对CAP理论的延伸和补充，主要是对AP的补充。牺牲数据的强一致性，来保证数据的可用性，虽然存在中间装填，但数据最终一致。

ACID 和 BASE 的区别与联系

ACID 是传统数据库常用的设计理念，追求强一致性模型。 BASE 支持的是大型分布式系统，提出通过牺牲强一致性获得高可用性。

ACID 和 BASE 代表了两种截然相反的设计哲学，在分布式系统设计的场景中，系统组件对一致性要求是不同的，因此 ACID 和 BASE 又会结合使用。

分布式事务分类：柔性事务和刚性事务

分布式场景下，多个服务同时对服务一个流程，比如电商下单场景，需要支付服务进行支付、库存服务扣减库存、订单服务进行订单生成、物流服务更新物流信息等。如果某一个服务执行失败，或者网络不通引起的请求丢失，那么整个系统可能出现数据不一致的原因。

上述场景就是分布式一致性问题，追根到底，分布式一致性的根本原因在于数据的分布式操作，引起的本地事务无法保障数据的原子性引起。

分布式一致性问题的解决思路有两种，一种是分布式事务，一种是尽量通过业务流程避免分布式事务。分布式事务是直接解决问题，而业务规避其实通过解决出问题的地方(解决提问题的人)。其实在真实业务场景中，如果业务规避不是很麻烦的前提，最优雅的解决方案就是业务规避。

分布式事务分类

分布式事务实现方案从类型上去分刚性事务、柔型事务：

- 刚性事务满足CAP的CP理论
- 柔性事务满足BASE理论（基本可用，最终一致）

刚性事务

刚性事务：通常无业务改造，强一致性，原生支持回滚/隔离性，低并发，适合短事务。

原则：刚性事务满足足CAP的CP理论

> 刚性事务指的是，要使分布式事务，达到像本地式事务一样，具备数据强一致性，从CAP来看，就是说，要达到CP状态。

刚性事务： XA 协议（2PC、JTA、JTS）、 3PC，但由于同步阻塞，处理效率低，不适合大型网站分布式场景。

柔性事务

柔性事务指的是，不要求强一致性，而是要求最终一致性，允许有中间状态，也就是Base理论，换句话说，就是AP状态。

> 与刚性事务相比，柔性事务的特点为：有业务改造，最终一致性，实现补偿接口，实现资源锁定接口，高并发，适合长事务。

柔性事务分为：

- 补偿型
- 异步确保型
- 最大努力通知型。

柔型事务： TCC/FMT、Saga （状态机模式、 Aop模式）、本地事务消息、消息事务（半消息）

刚性事务： XA模型、 XA接口规范、 XA实现

XA模型 或者 X/Open DTP模型

> X/OPEN是一个组织.X/Open国际联盟有限公司是一个欧洲基金会，它的建立是为了向UNIX环境提供标准。它主要的目标是促进对UNIX语言、接口、网络和应用的开放式系统协议的制定。它还促进在不同的UNIX环境之间的应用程序的互操作性，以及支持对电气电子工程师协会（IEEE）对UNIX的可移植操作系统接口（POSIX）规范。

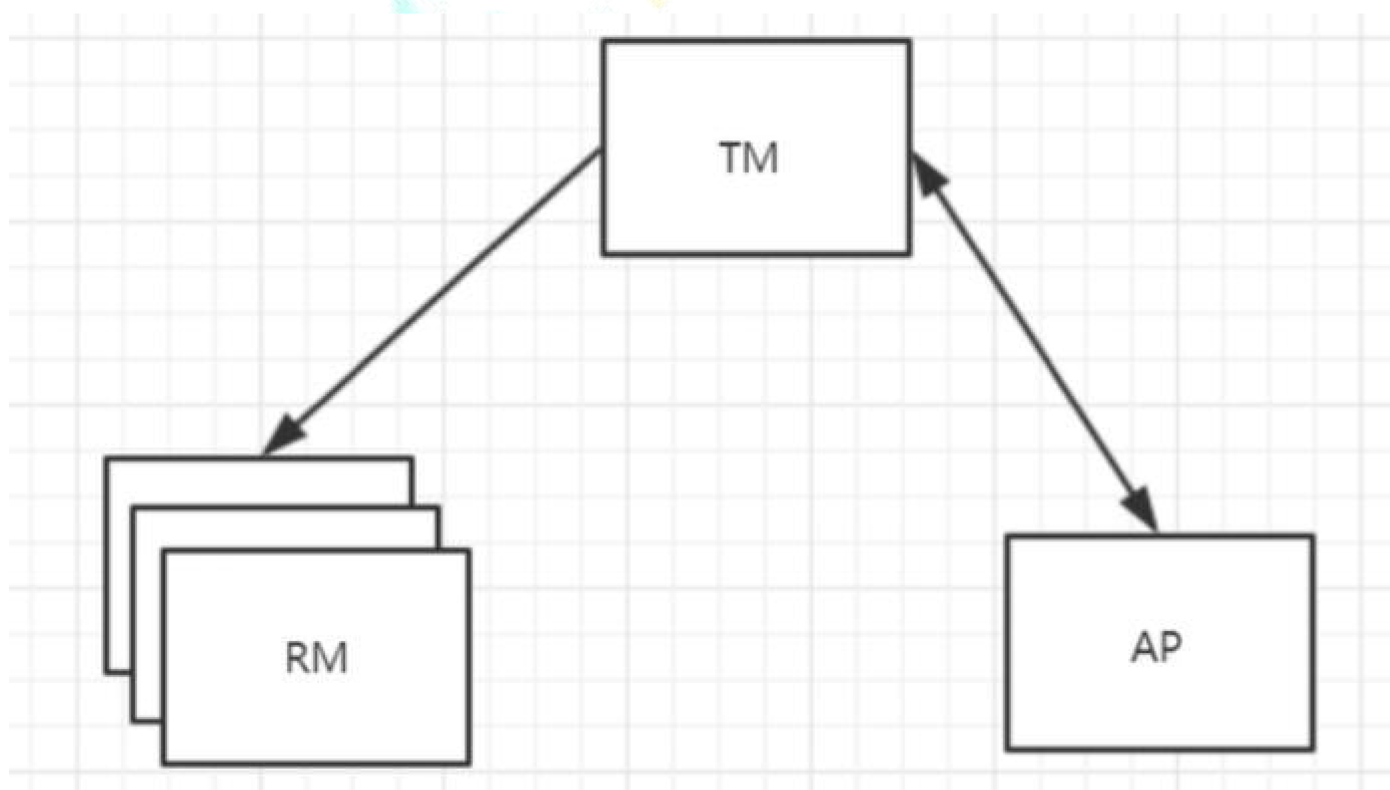
X/Open **DTP(Distributed Transaction Process)** 是一个分布式事务模型。这个模型主要使用了两段提交(2PC - Two-Phase-Commit)来保证分布式事务的完整性。

在X/Open **DTP(Distributed Transaction Process)**模型里面，有三个角色：

AP: Application，应用程序。也就是业务层。哪些操作属于一个事务，就是AP定义的。

TM: Transaction Manager，事务管理器。接收AP的事务请求，对全局事务进行管理，管理事务分支状态，协调RM的处理，通知RM哪些操作属于哪些全局事务以及事务分支等等。这个也是整个事务调度模型的核心部分。

RM： Resource Manager，资源管理器。一般是数据库，也可以是其他的资源管理器，如消息队列(如JMS数据源)，文件系统等。



XA把参与事务的角色分成AP， RM， TM。

AP， 即应用，也就是我们的业务服务。

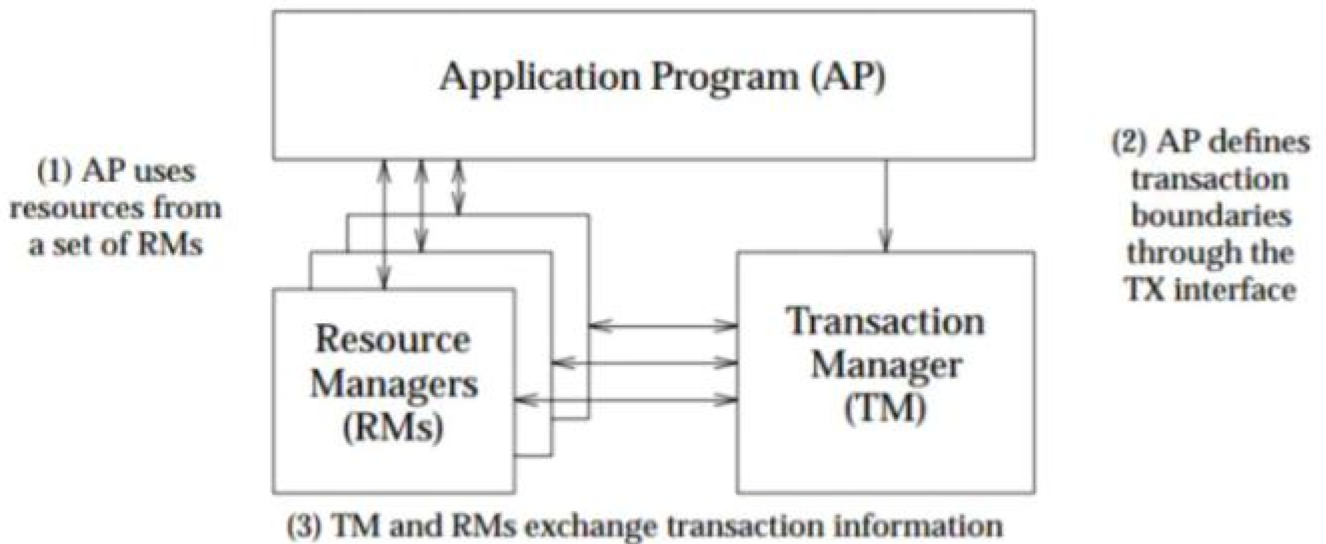
RM指的是资源管理器，即DB， MQ等。

TM则是事务管理器。

> AP自己操作TM，当需要事务时， AP向TM请求发起事务， TM负责整个事务的提交，回滚等。

XA规范主要定义了(全局)事务管理器(Transaction Manager)和(局部)资源管理器(Resource Manager)之间的接口。 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。

XA之所以需要引入事务管理器是因为，在分布式系统中，从理论上讲（参考Fischer等的论文）， 两台机器理论上无法达到一致的状态，需要引入一个单点进行协调。事务管理器控制着全局事务，管理事务生命周期，并协调资源。资源管理器负责控制和管理实际资源（如数据库或JMS队列）



XA规范

什么是XA

用非常官方的话来说：

- XA 规范 是 X/Open 组织定义的分布式事务处理（DTP， Distributed Transaction Processing）标准。
- XA 规范 描述了全局的事务管理器与局部的资源管理器之间的接口。

  XA规范 的目的是允许的多个资源（如数据库，应用服务器，消息队列等）在同一事务中访问，这样可以使 ACID 属性跨越应用程序而保持有效。
- XA 规范 使用两阶段提交（2PC， Two-Phase Commit）协议来保证所有资源同时提交或回滚任何特定的事务。
- XA 规范 在上世纪 90 年代初就被提出。目前，几乎所有主流的数据库都对 XA 规范 提供了支持。

XA规范(XA Specification) 是X/OPEN 提出的分布式事务处理规范。 XA则规范了TM与RM之间的通信接口，在TM与多个RM之间形成一个双向通信桥梁，从而在多个数据库资源下保证ACID四个特性。目前知名的数据库，

如Oracle，DB2,mysql等，都是实现了XA接口的，都可以作为RM。

XA是数据库的分布式事务，强一致性，在整个过程中，数据一张锁住状态，即从prepare到commit、rollback的整个过程中， TM一直把持折数据库的锁，如果有其他人要修改数据库的该条数据，就必须等待锁的释放，存在长事务风险。

以下的函数使事务管理器可以对资源管理器进行的操作：

1） xa_open,xa_close：建立和关闭与资源管理器的连接。

2） xa_start,xa_end：开始和结束一个本地事务。

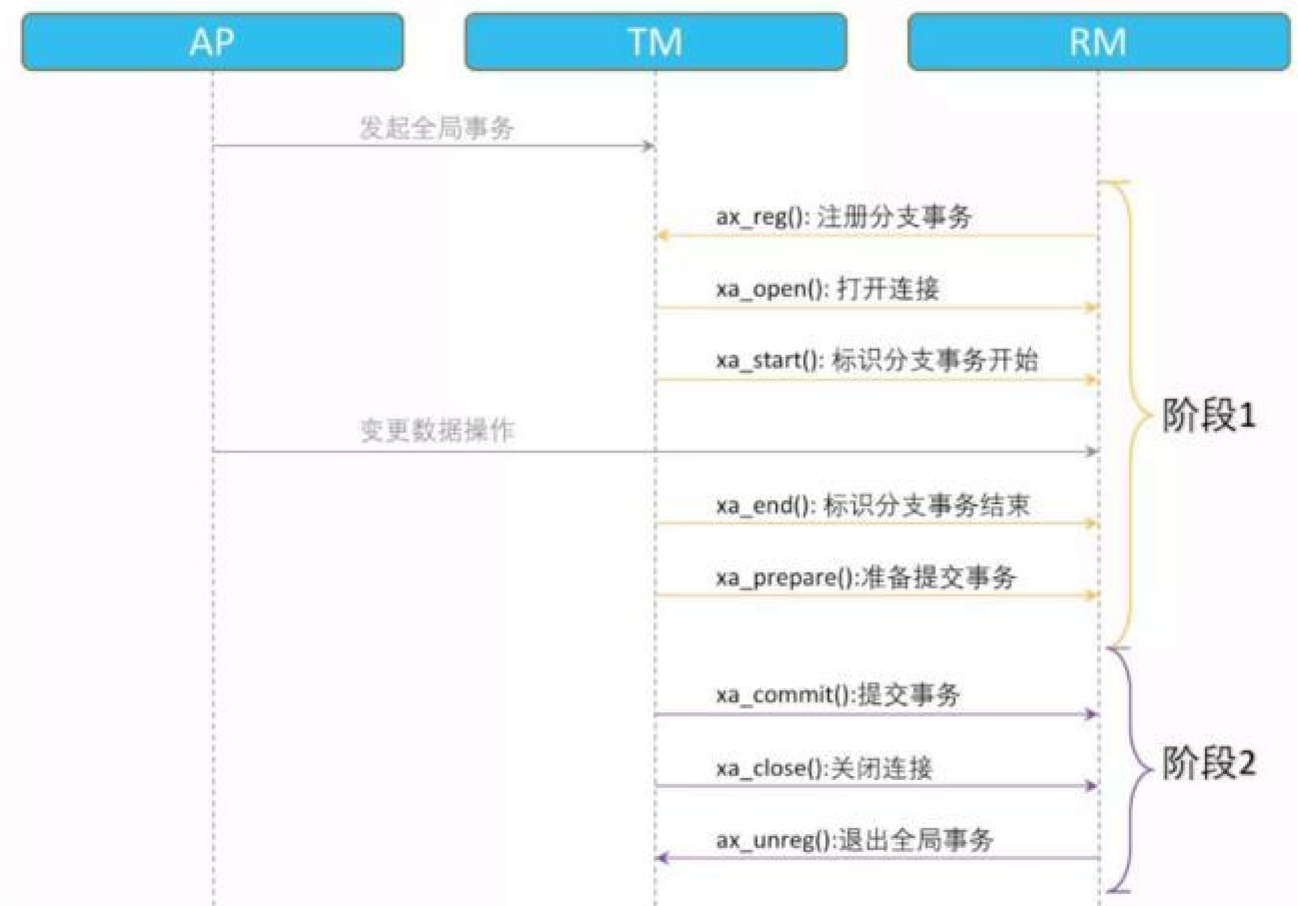
3） xa_prepare,xa_commit,xa_rollback：预提交、提交和回滚一个本地事务。

4） xa_recover：回滚一个已进行预提交的事务。

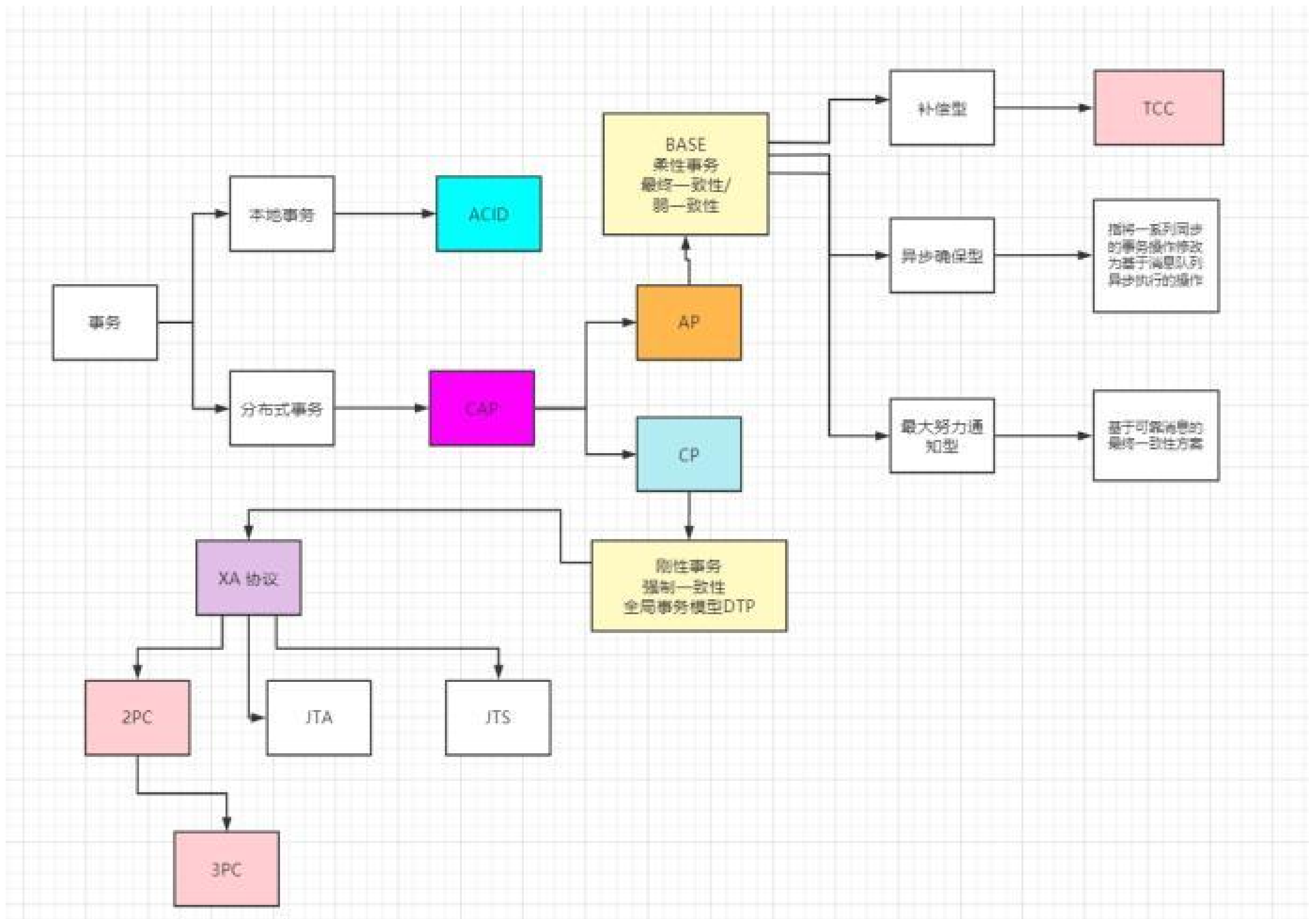
5） ax_开头的函数使资源管理器可以动态地在事务管理器中进行注册，并可以对XID(TRANSACTION IDS)进行操作。

6） ax_reg,ax_unreg；允许一个资源管理器在一个TMS(TRANSACTION MANAGER SERVER)中动态注册或撤消注册。

XA各个阶段的处理流程



XA协议的实现



2PC/3PC协议

两阶段提交（2PC）协议是XA规范定义的 数据一致性协议。

三阶段提交（3PC）协议对 2PC协议的一种扩展。

Seata

> Seata，官网，github，1万多星

Seata 是一款开源的分布式事务解决方案，致力于在微服务架构下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

在 Seata 开源之前， Seata 对应的内部版本在阿里经济体内部一直扮演着分布式一致性中间件的角色，帮助经济体平稳的度过历年的双11，对各BU业务进行了有力的支撑。商业化产品GTS 先后在阿里云、金融云进行售卖

Jta规范

作为java平台上事务规范 JTA（Java Transaction API）也定义了对XA事务的支持，实际上， JTA是基于XA架构上建模的，在JTA 中，事务管理器抽象为javax.transaction.TransactionManager接口，并通过底层事务服务（即JTS）实现。像很多其他的java规范一样， JTA仅仅定义了接口，具体的实现则是由供应商(如J2EE厂商)负责提供，目前JTA的实现主要由以下几种:

1.J2EE容器所提供的JTA实现(JBoss)
2.□ □ 的JTA实现:如JOTM， Atomikos.

这些实现可以应用在那些不使用J2EE应用服务器的环境里用以提供分布事事务保证。如Tomcat,Jetty以及普通的java应用。

JTS规范

事务是编程中必不可少的一项内容，基于此，为了规范事务开发， Java增加了关于事务的规范，即JTA和JTS

JTA定义了一套接口，其中约定了几种主要的角色：TransactionManager、UserTransaction、Transaction、XAResource，并定义了这些角色之间需要遵守的规范，如Transaction的委托给TransactionManager等。

JTS也是一组规范，上面提到JTA中需要角色之间的交互，那应该如何交互？ JTS就是约定了交互细节的规范。

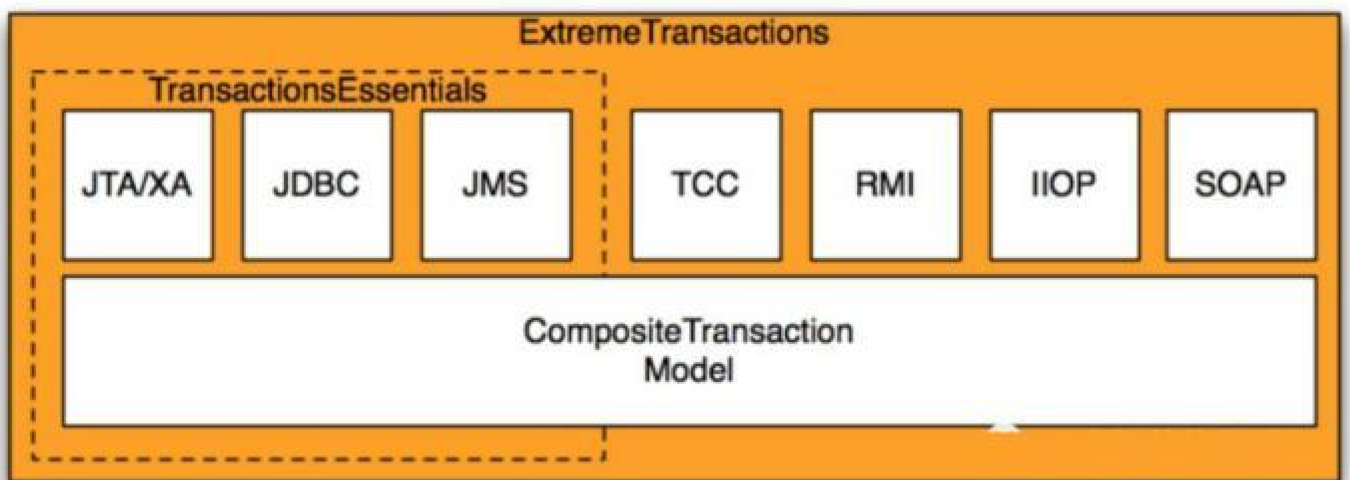
总体上来说JTA更多的是从框架的角度来约定程序角色的接口，而JTS则是从具体实现的角度来约定程序角色之间的接口，两者各司其职。

Atomikos分布式事务实现

Atomikos公司旗下有两款著名的分布事务产品：

- TransactionEssentials：开源的免费产品
- ExtremeTransactions：商业版，需要收费

这两个产品的关系如下图所示：



可以看到，在开源版本中支持JTA/XA、JDBC、JMS的事务。

atomikos也支持与spring事务整合。

spring事务管理器的顶级抽象是PlatformTransactionManager接口，其提供了个重要的实现类：

- DataSourceTransactionManager：用于实现本地事务
- **JTATransactionManager**：用于实现分布式事务

显然，在这里，我们需要配置的是JTATransactionManager。

```
public class JTAService {
    @Autowired
    private UserMapper userMapper;//操作db_user库
    @Autowired
    private AccountMapper accountMapper;//操作db_account
库

    @Transactional
    public void insert() {
        User user = new User();
        user.setName("wangxiaoxiao");
        userMapper.insert(user);
        //模拟异常，  spring回滚后， db_user库中user表中也不会插
入记录
        Account account = new Account();
        account.setUserId(user.getId());
        account.setMoney(123456789);
        accountMapper.insert(account);
    }
}
```

XA的主要限制

- 必须要拿到所有数据源，而且数据源还要支持XA协议。目前MySQL中只有InnoDB存储引擎支持XA协议。
- 性能比较差，要把所有涉及到的数据都要锁定，是强一致性的，会产生长事务。

Seata AT 模式

Seata AT 模式是增强型2pc模式。

AT 模式：　两阶段提交协议的演变，没有一直锁表

- 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。或回滚通过一阶段的回滚日志进行反向补偿

LCN（2pc)

TX-LCN， 官方文档， github，3千多星， 5.0以后由于框架兼容了LCN（2pc）、 TCC、TXC 三种事务模式，为了区分LCN模式，特此将LCN分布式事务改名为TX-LCN分布式事务框架。

TX-LCN定位于一款事务协调性框架，框架其本身并不生产事务，而是本地事务的协调者，从而达到事务一致性的效果。

TX-LCN 主要有两个模块， Tx-Client(TC)， Tx-Manager™.

- TM （Tx-Manager）：是□□ 的服务，是分布式事务的控制□ ，协调分 布式事务的提交，回滚

- TC（Tx-Client）：由业务系统集成，事务发起方、参与方都由TxClient端来控制

2PC （标准XA模型）

2PC即Two-Phase Commit，二阶段提交。

详解：二个阶段

广泛应用在数据库领域，为了使得基于分布式架构的所有节点可以在进行事务处理时能够保持原子性和一致性。绝大部分关系型数据库，都是基于2PC完成分布式的事务处理。

顾名思义， 2PC分为两个阶段处理，阶段一：提交事务请求、阶段二：执行事务提交;

如果阶段一超时或者出现异常， 2PC的阶段二：中断事务

阶段一：提交事务请求

1. 事务询问。协调者向所有参与者发送事务内容，询问是否可以执行提交操作，并开始等待各参与者进行响应；
2. 执行事务。各参与者节点，执行事务操作，并将Undo和Redo操作计入本机事务日志；
3. 各参与者向协调者反馈事务问询的响应。成功执行返回Yes，否则返回No。

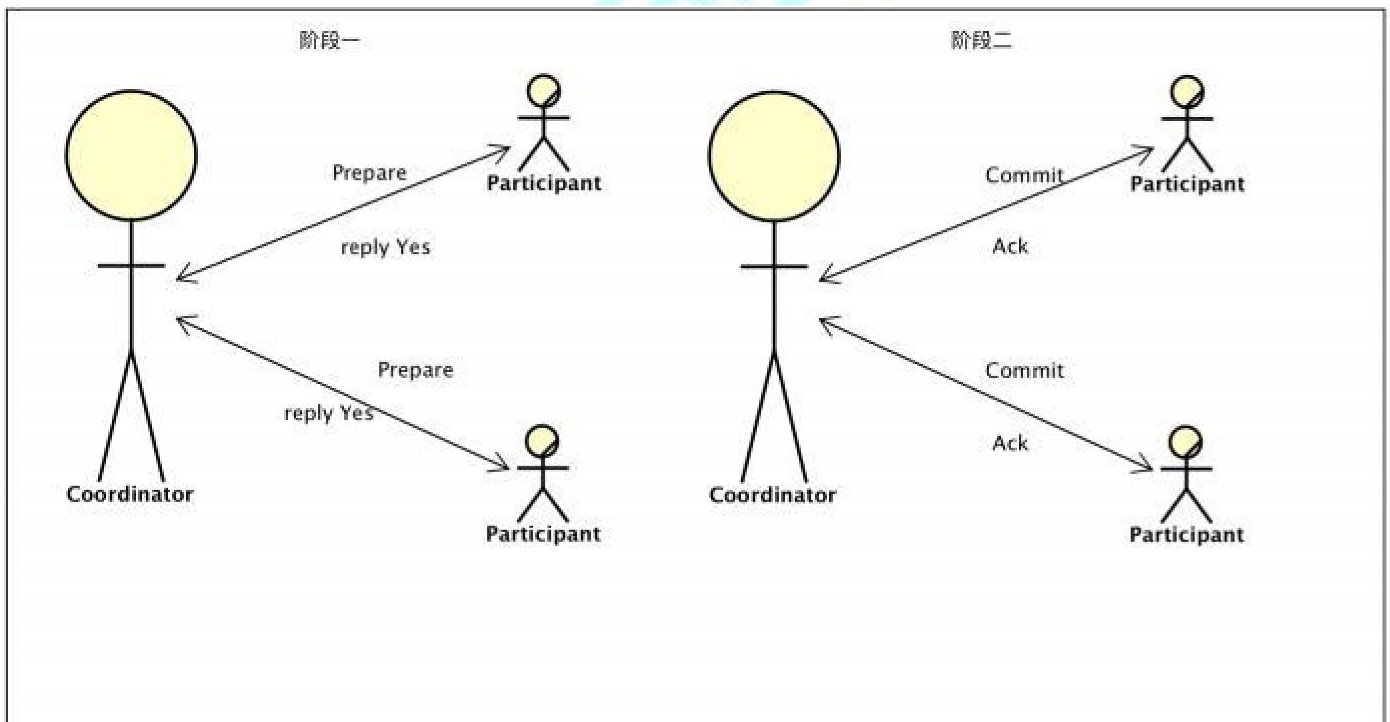
阶段二：执行事务提交

协调者在阶段二决定是否最终执行事务提交操作。这一阶段包含两种情形:

执行事务提交

所有参与者reply Yes，那么执行事务提交。

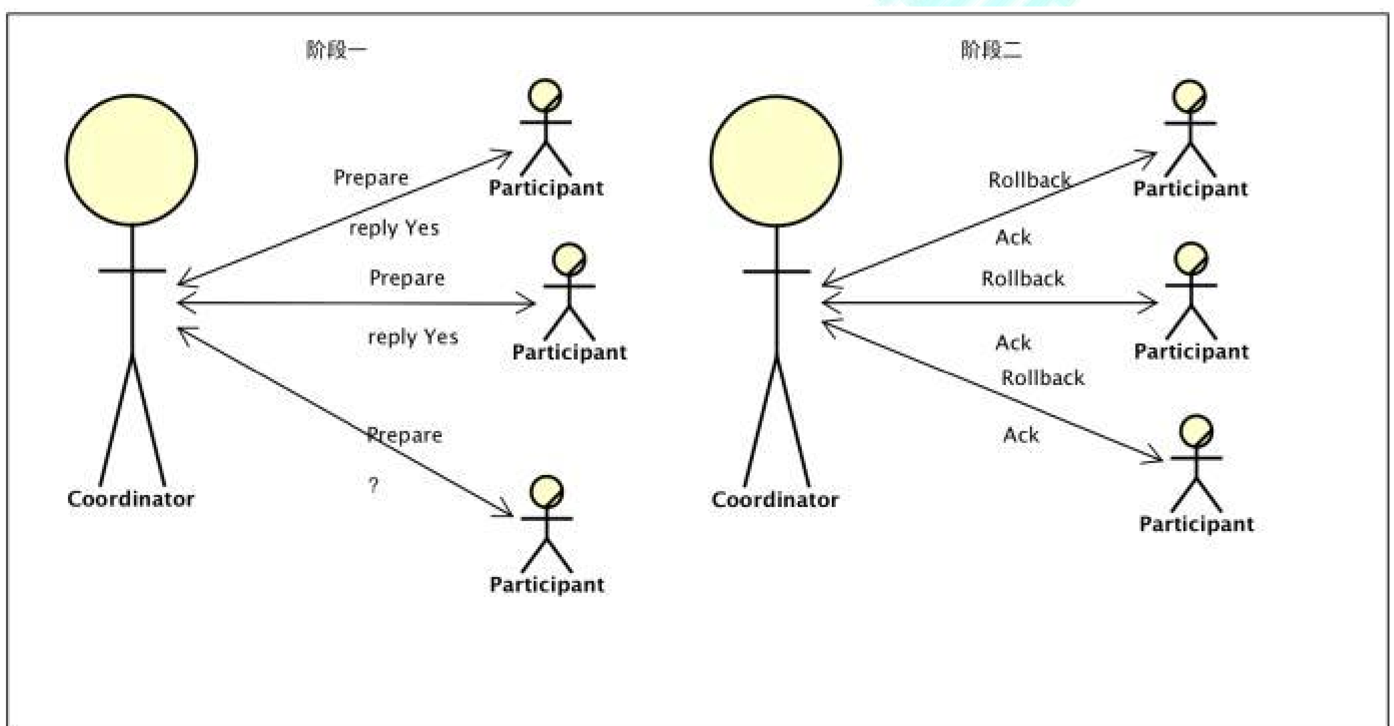
1. 发送提交请求。协调者向所有参与者发送Commit请求；
2. 事务提交。参与者收到Commit请求后，会**正式执行事务提交操作**，并在完成提交操作之后，释放在整个事务执行期间占用的资源；
3. 反馈事务提交结果。参与者在完成事务提交后，写协调者发送Ack消息确认；
4. 完成事务。协调者在收到所有参与者的Ack后，完成事务。



阶段二：中断事务

事情总会出现意外，当存在某一参与者向协调者发送No响应，或者等待超时。协调者只要无法收到所有参与者的Yes响应，就会中断事务。

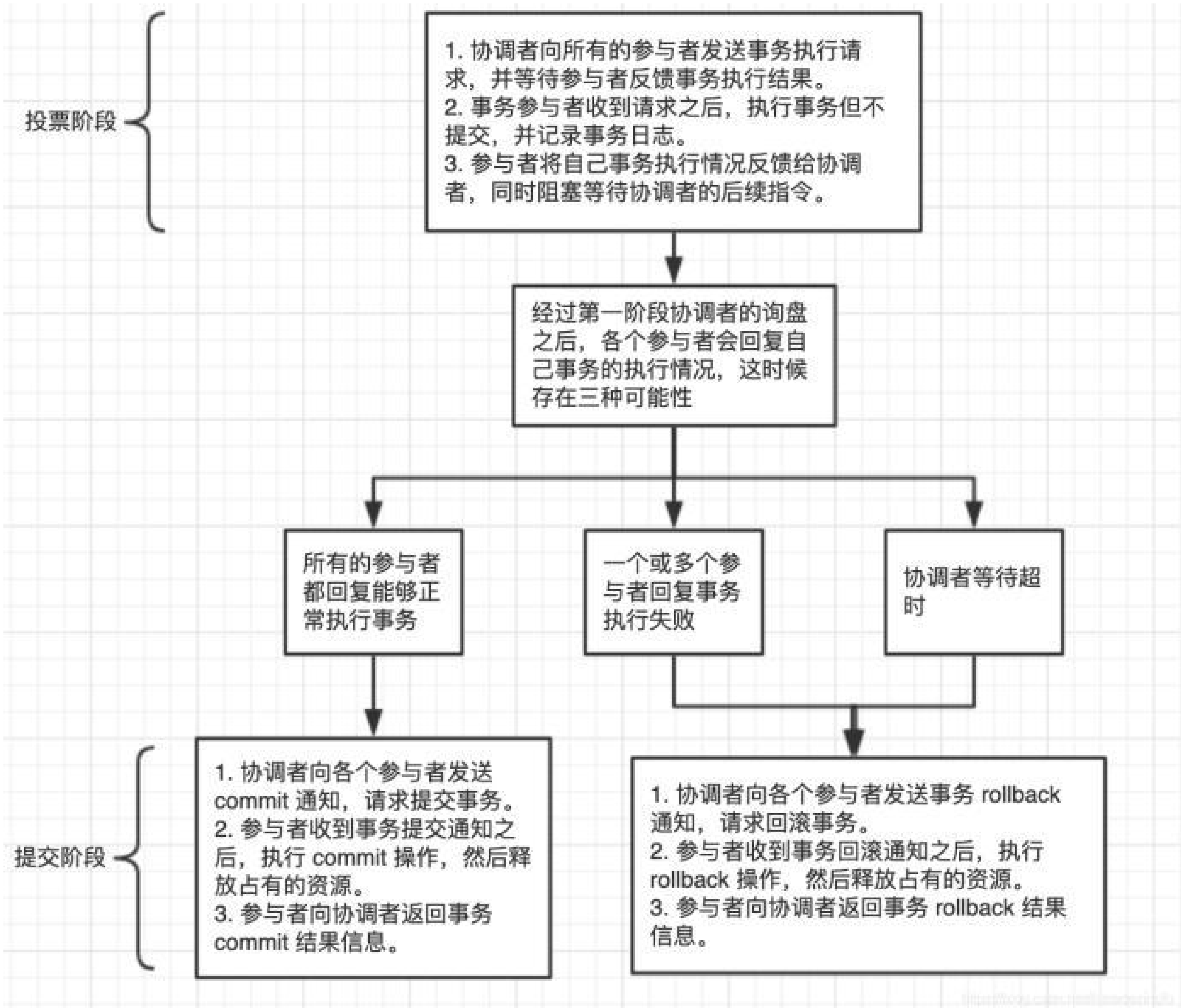
1. 发送回滚请求。协调者向所有参与者发送Rollback请求；
2. 回滚。参与者收到请求后，利用本机Undo信息，执行Rollback操作。并在回滚结束后释放该事务所占用的系统资源；
3. 反馈回滚结果。参与者在完成回滚操作后，向协调者发送Ack消息；
4. 中断事务。协调者收到所有参与者的回滚Ack消息后，完成事务中断。



2pc解决的是分布式数据强一致性问题

顾名思义，两阶段提交在处理分布式事务时分为两个阶段： voting （投票阶段，有的地方会叫做prepare阶段）和commit阶段。

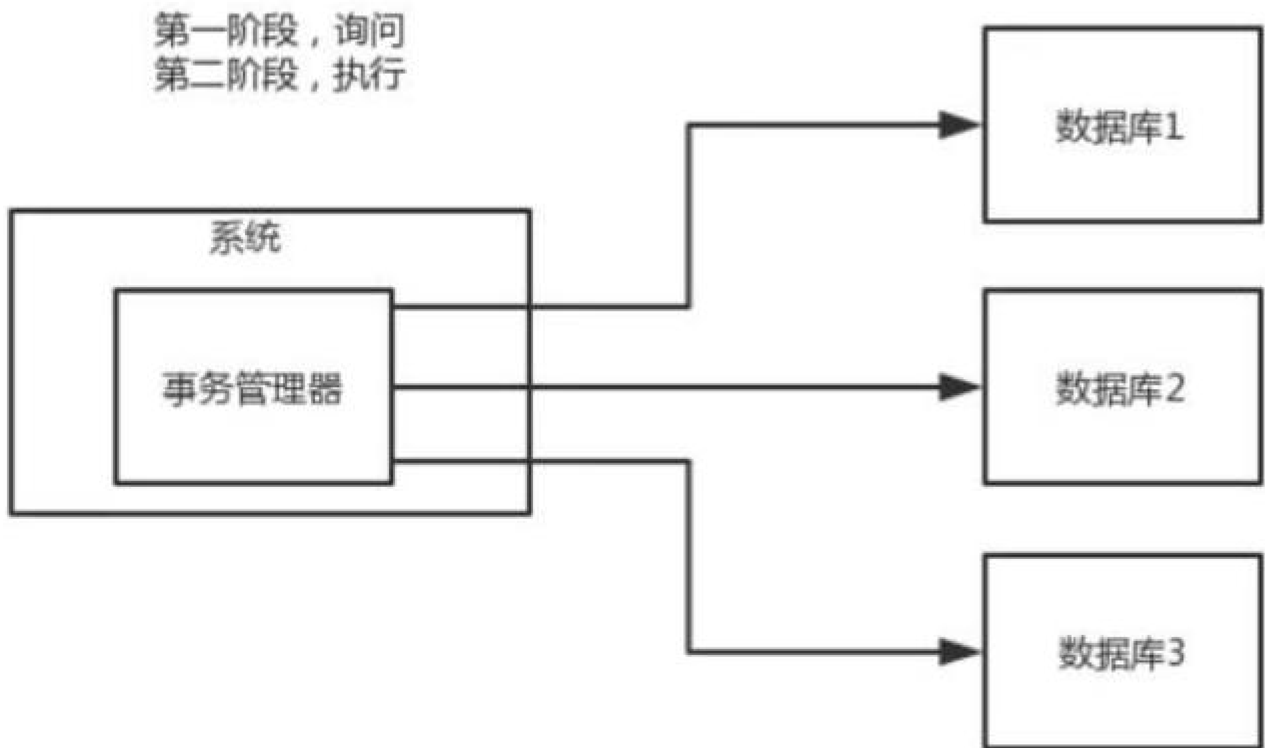
2pc中存在两个角色，事务协调者（seata、atomikos、lcn）和事务参与者，事务参与者通常是指应用的数据库。



2PC二阶段提交的特点

2PC方案比较适合单体应用

2PC 方案中，有一个事务管理器的角色，负责协调多个数据库（资源管理器）的事务，事务管理器先问问各个数据库你准备好了吗？如果每个数据库都回复 ok，那么就正式提交事务，在各个数据库上执行操作；如果任何其中一个数据库回答不 ok，那么就回滚事务。



2PC 方案比较适合单体应用里，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。

2PC 方案实际很少用，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。

2PC具有明显的优缺点：

优点主要体现在实现原理简单；
缺点比较多：

- 2PC的提交在执行过程中，所有参与事务操作的逻辑都处于阻塞状态，也就是说，各个参与者都在等待其他参与者响应，无法进行其他操作；
- 协调者是个单点，一旦出现问题，其他参与者将无法释放事务资源，也无法完成事务操作；
- 数据不一致。当执行事务提交过程中，如果协调者向所有参与者发送Commit请求后，发生局部网络异常或者协调者在尚未发送完Commit请求，即出现崩溃，最终导致只有部分参与者收到、执行请求。于是整个系统将会出现数据不一致的情形；
- 保守。 2PC没有完善的容错机制，当参与者出现故障时，协调者无法快速得知这一失败，只能严格依赖超时设置来决定是否进一步的执行提交还是中断事务。

实际上分布式事务是一件非常复杂的事情，两阶段提交只是通过增加了事务协调者（Coordinator）的角色来通过2个阶段的处理流程来解决分布式系统中一个事务需要跨多个服务节点的数据一致性问题。但是从异常情况上考虑，这个流程也并不是那么的无懈可击。

假设如果在第二个阶段中Coordinator在接收到Partcipant的"Vote_Request"后挂掉了或者网络出现了异常，那么此时Partcipant节点就会一直处于本地事务挂起的状态，从而长时间地占用资源。当然这种情况只会出现在极端情况下，然而作为一套健壮的软件系统而言，异常Case的处理才是真正考验方案正确性的地方。

总结一下： XA-两阶段提交协议中会遇到的一些问题

- **性能问题**

从流程上我们可以看得出，其最大缺点就在于它的执行过程中间，节点都处于阻塞状态。各个操作数据库的节点此时都占用着数据库资源，只有当所有节点准备完毕，事务协调者才会通知进行全局提交，参与者进行本地事务提交后才会释放资源。这样的过程会比较漫长，对性能影响比较大。

- **协调者单点故障问题**

事务协调者是整个XA模型的核心，一旦事务协调者节点挂掉，会导致参与者收不到提交或回滚的通知，从而导致参与者节点始终处于事务无法完成的中间状态。

- **丢失消息导致的数据不一致问题**

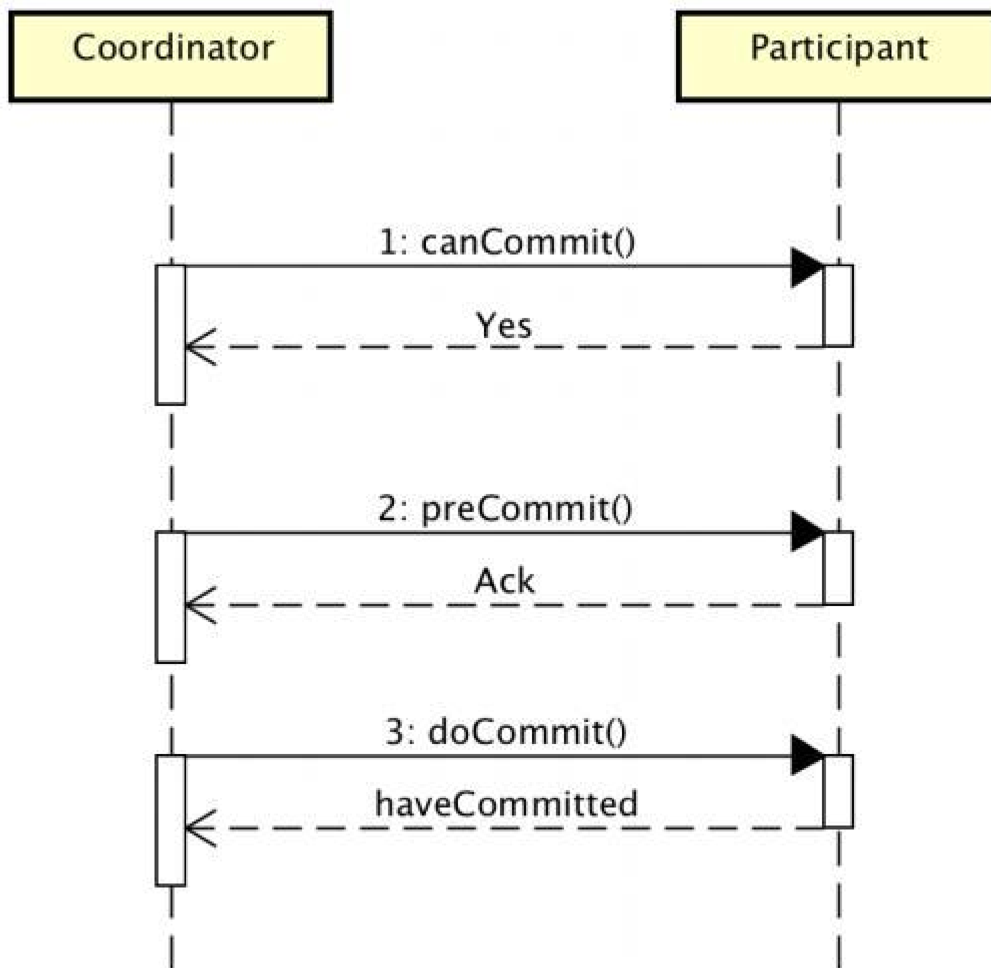
在第二个阶段，如果发生局部网络问题，一部分事务参与者收到了提交消息，另一部分事务参与者没收到提交消息，那么就会导致节点间数据的不一致问题。

3PC

针对2PC的缺点，研究者提出了3PC，即Three-Phase Commit。

作为2PC的改进版， 3PC将原有的两阶段过程，重新划分为CanCommit、PreCommit和do Commit三个阶段。

详解：三个阶段



阶段一： CanCommit

1. 事务询问。协调者向所有参与者发送包含事务内容的canCommit的请求，询问是否可以执行事务提交，并等待应答；
2. 各参与者反馈事务询问。正常情况下，如果参与者认为可以顺利执行事务，则返回Yes，否则返回No。

阶段二： PreCommit

在本阶段，协调者会根据上一阶段的反馈情况来决定是否可以执行事务的PreCommit操作。有以下两种可能：

执行事务预提交

1. 发送预提交请求。协调者向所有节点发出PreCommit请求，并进入prepared阶段；
2. 事务预提交。参与者收到PreCommit请求后，会执行事务操作，并将Undo和Redo日志写入本机事务日志；
3. 各参与者成功执行事务操作，同时将反馈以Ack响应形式发送给协调者，同事等待最终的Commit或Abort指令。

中断事务

加入任意一个参与者向协调者发送No响应，或者等待超时，协调者在没有得到所有参与者响应时，即可以中断事务：

1. 发送中断请求。 协调者向所有参与者发送Abort请求；
2. 中断事务。无论是收到协调者的Abort请求，还是等待协调者请求过程中出现超时，参与者都会中断事务；

阶段三： doCommit

在这个阶段，会真正的进行事务提交，同样存在两种可能。

执行提交

1. 发送提交请求。假如协调者收到了所有参与者的Ack响应，那么将从预提交转换到提交状态，并向所有参与者，发送doCommit请求；
2. 事务提交。参与者收到doCommit请求后，会正式执行事务提交操作，并在完成提交操作后释放占用资源；
3. 反馈事务提交结果。参与者将在完成事务提交后，向协调者发送Ack消息；
4. 完成事务。协调者接收到所有参与者的Ack消息后，完成事务。

中断事务

在该阶段，假设正常状态的协调者接收到任一个参与者发送的No响应，或在超时时间内，仍旧没收到反馈消息，就会中断事务：

1. 发送中断请求。协调者向所有的参与者发送abort请求；
2. 事务回滚。参与者收到abort请求后，会利用阶段二中的Undo消息执行事务回滚，并在完成回滚后释放占用资源；
3. 反馈事务回滚结果。参与者在完成回滚后向协调者发送Ack消息；
4. 中端事务。协调者接收到所有参与者反馈的Ack消息后，完成事务中断。

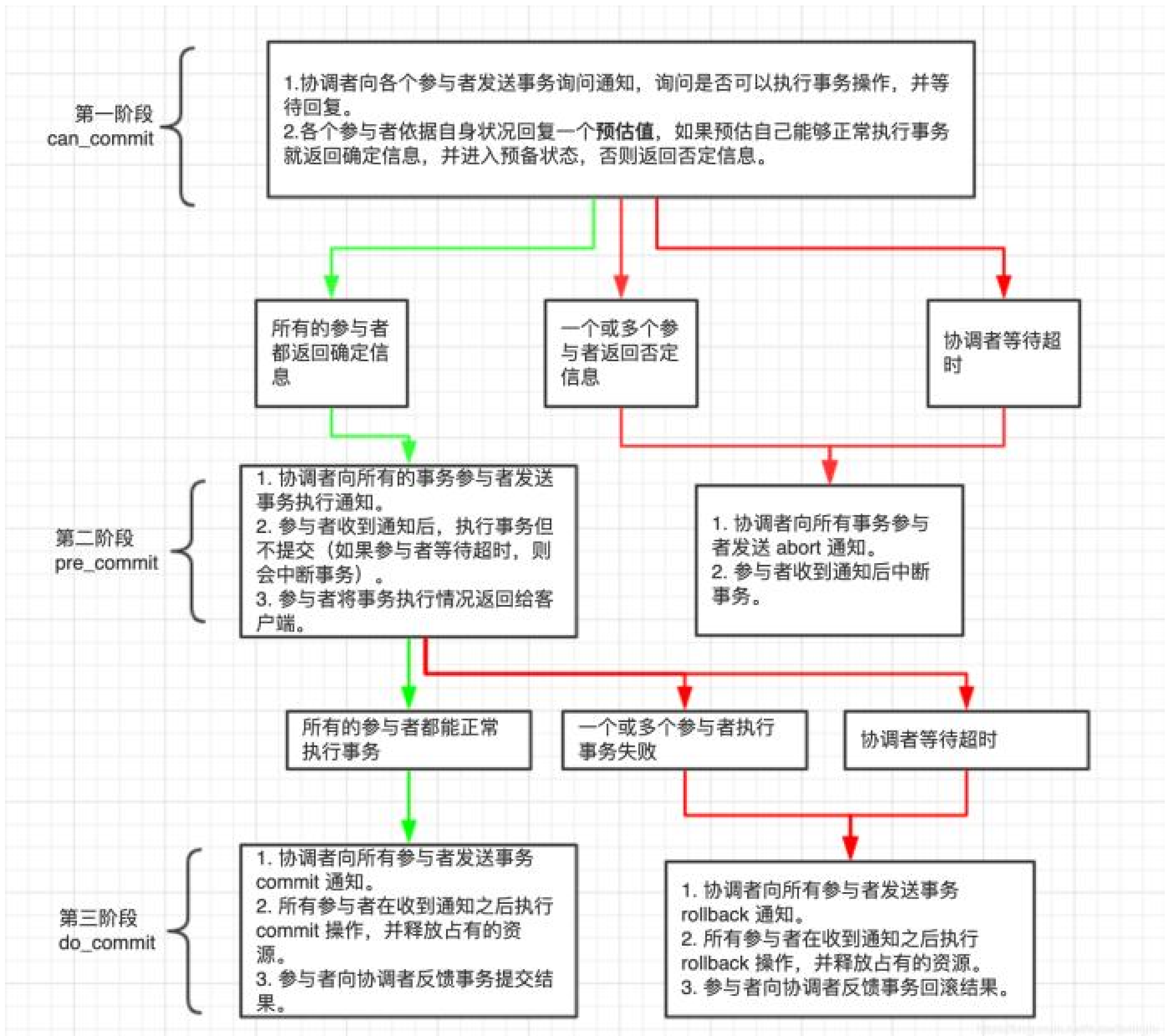
2PC和3PC的区别：

3PC有效降低了2PC带来的参与者阻塞范围，并且能够在出现单点故障后继续达成一致；

但3PC带来了新的问题，在参与者收到preCommit消息后，如果网络出现分区，协调者和参与者无法进行后续的通信，这种情况下，参与者在等待超时后，依旧会执行事务提交，这样会导致数据的不一致。

2PC和3PC的区别：

三阶段提交协议在协调者和参与者中都引入 **超时机制**，并且把两阶段提交协议的第一个阶段拆分成了两步：询问，然后再锁资源，最后真正提交。三阶段提交的三个阶段分别为： can_commit，pre_commit，do_commit。



在doCommit阶段，如果参与者无法及时接收到来自协调者的doCommit或者abort请求时，会在等待超时之后，继续进行事务的提交。（其实这个应该是基于概率来决定的，当进入第三阶段时，说明参与者在第二阶段已经收到了PreCommit请求，那么协调者产生PreCommit请求的前提条件是他在第二阶段开始之前，收到所有参与者的CanCommit响应都是Yes。（一旦参与者收到了PreCommit，意味他知道大家其实都同意修改了）所以，一句话概括就是，当进入第三阶段时， **由于网络超时等原因，虽然参与者没有收到commit或者abort响应，但是他有理由相信：成功提交的几率很大。**）

3PC主要解决的单点故障问题：

相对于2PC，3PC主要解决的单点故障问题，并减少阻塞， **因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。**

但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

"3PC相对于2PC而言到底优化了什么地方呢?"

相比较2PC而言， 3PC对于协调者（Coordinator）和参与者（Partcipant）都设置了超时时间，而2PC只有协调者才拥有超时机制。这解决了一个什么问题呢?

这个优化点，主要是避免了参与者在长时间无法与协调者节点通讯（协调者挂掉了）的情况下，无法释放资源的问题，因为参与者自身拥有超时机制会在超时后，自动进行本地commit从而进行释放资源。而这种机制也侧面降低了整个事务的阻塞时间和范围。

另外，通过CanCommit、PreCommit、DoCommit三个阶段的设计，相较于2PC而言，多设置了一个缓冲阶段保证了在最后提交阶段之前各参与节点的状态是一致的。

以上就是3PC相对于2PC的一个提高（相对缓解了2PC中的前两个问题），但是3PC依然没有完全解决数据不一致的问题。假如在 DoCommit 过程，参与者A无法接收协调者的通信，那么参与者A会自动提交，但是提交失败了，其他参与者成功了，此时数据就会不一致。

XA规范的问题

但是XA规范在1994年就出现了，至今没有大规模流行起来，必然有他一定的缺陷：

1. 数据锁定：数据在事务未结束前，为了保障一致性，根据数据隔离级别进行锁定。
2. 协议阻塞：本地事务在全局事务 没 commit 或 callback前都是阻塞等待的。
3. 性能损耗高：主要体现在事务协调增加的RT成本，并发事务数据使用锁进行竞争阻塞。

XA协议比较简单，而且一旦商业数据库实现了XA协议，使用分布式事务的成本也比较低。但是， XA也有致命的缺点，那就是性能不理想，特别是在交易下单链路，往往并发量很高， XA无法满足高并发场景。 XA目前在商业数据库支持的比较理想，在mysql数据库中支持的不太理想， mysql的XA实现，没有记录prepare阶段日志，主备切换回导致主库与备库数据不一致。许多nosql也没有支持XA，这让XA的应□ 场景变得□ 常□ 隘。

其实也并非不用，例如在IBM大型机上基于CICS很多跨资源是基于XA协议实现的分布式事务，事实上XA也算分布式事务处理的规范了，但在为什么互联网中很少使用，究其原因有以下几个：

- 性能（阻塞性协议，增加响应时间、锁时间、死锁）；

- 数据库支持完善度（MySQL 5.7之前都有缺陷）；
- 协调者依赖独立的J2EE中间件（早期重量级Weblogic、Jboss、后期轻 量级Atomikos、Narayana和Bitronix）；
- 运维复杂， DBA缺少这方面经验；
- 并不是所有资源都支持XA协议；

准确讲XA是一个规范、协议，它只是定义了一系列的接口，只是目前大多数实现XA的都是数据库或者MQ，所以提起XA往往多指基于资源层的底层分布式事务解决方案。其实现在也有些数据分片框架或者中间件也支持XA协议，毕竟它的兼容性、普遍性更好。

柔性事务的分类

> 在电商领域等互联网场景下，刚性事务在数据库性能和处理能力上都暴露出了瓶颈。
>
> 柔性事务有两个特性：基本可用和柔性状态。
>
> - 基本可用是指分布式系统出现故障的时候允许损失一部分的可用性。
> - 柔性状态是指允许系统存在中间状态，这个中间状态不会影响系统整体的可用性，比如数据库读写分离的主从同步延迟等。柔性事务的一致性指的是最终一致性。

柔性事务主要分为**补偿型**和**通知型**，

补偿型事务又分： TCC、Saga；

通知型事务分： MQ事务消息、最大努力通知型。

> 补偿型事务都是同步的，通知型事务都是异步的。

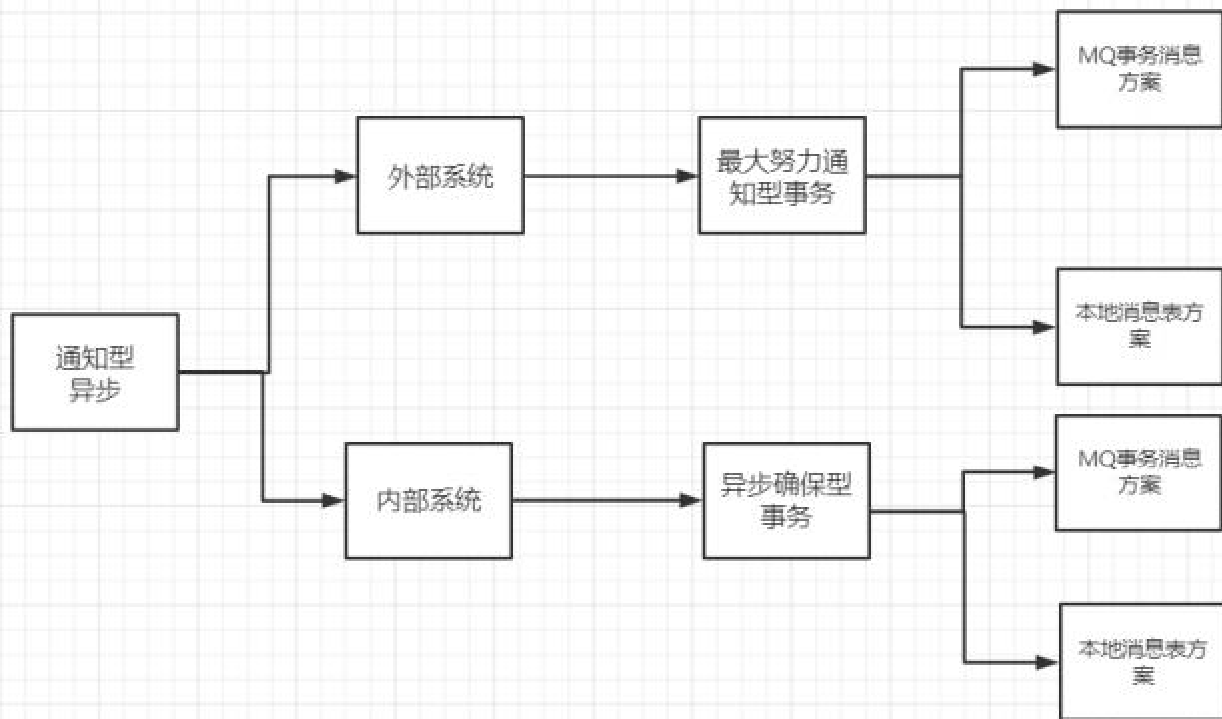
通知型事务

通知型事务的主流实现是通过MQ （消息队列）来通知其他事务参与者自己事务的执行状态，引入MQ组件，有效的将事务参与者进行解耦，各参与者都可以异步执行，所以通知型事务又被称为**异步事务**。

通知型事务主要适用于那些需要异步更新数据，并且对数据的实时性要求较低的场景，主要包含:

> **异步确保型事务**和**最大努力通知事务**两种。

- **异步确保型事务**：主要适用于内部系统的数据最终一致性保障，因为内部相对比较可控，如订单和购物车、收货与清算、支付与结算等等场景；
- **最大努力通知**：主要用于外部系统，因为外部的网络环境更加复杂和不可信，所以只能尽最大努力去通知实现数据最终一致性，比如充值平台与运营商、支付对接等等跨网络系统级别对接；



异步确保型事务

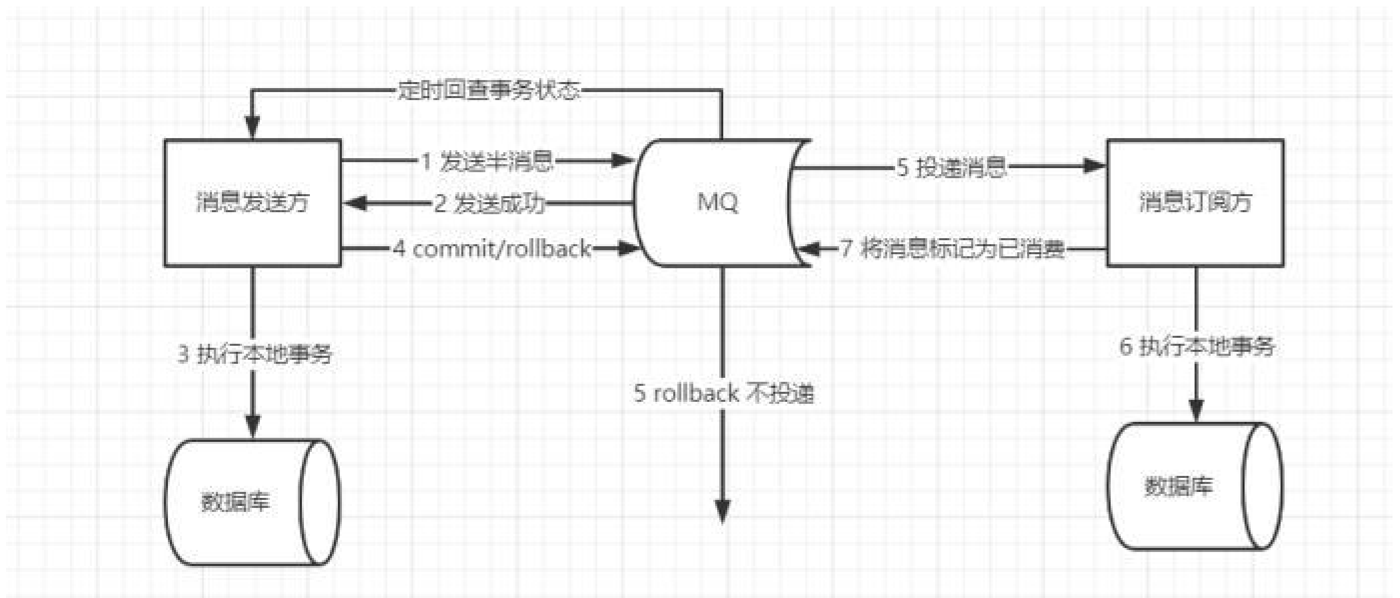
指将一系列同步的事务操作修改为基于消息队列异步执行的操作，来避免分布式事务中同步阻塞带来的数据操作性能的下降。

MQ事务消息方案

基于MQ的事务消息方案主要依靠MQ的**半消息机制**来实现投递消息和参与者自身本地事务的一致性保障。半消息机制实现原理其实借鉴的2PC的思路，是二阶段提交的广义拓展。

> **半消息**：在原有队列消息执行后的逻辑，如果后面的本地逻辑出错，则不发送该消息，如果通过则告知MQ发送；

流程



1. 事务发起方首先发送半消息到MQ；
2. MQ通知发送方消息发送成功；
3. 在发送半消息成功后执行本地事务；
4. 根据本地事务执行结果返回commit或者是rollback；
5. 如果消息是rollback，MQ将丢弃该消息不投递；如果是commit， MQ将会消息发送给消息订阅方；
6. 订阅方根据消息执行本地事务；
7. 订阅方执行本地事务成功后再从MQ中将该消息标记为已消费；
8. 如果执行本地事务过程中，执行端挂掉，或者超时， MQ服务器端将不停的询问producer来获取事务状态；
9. Consumer端的消费成功机制有MQ保证；

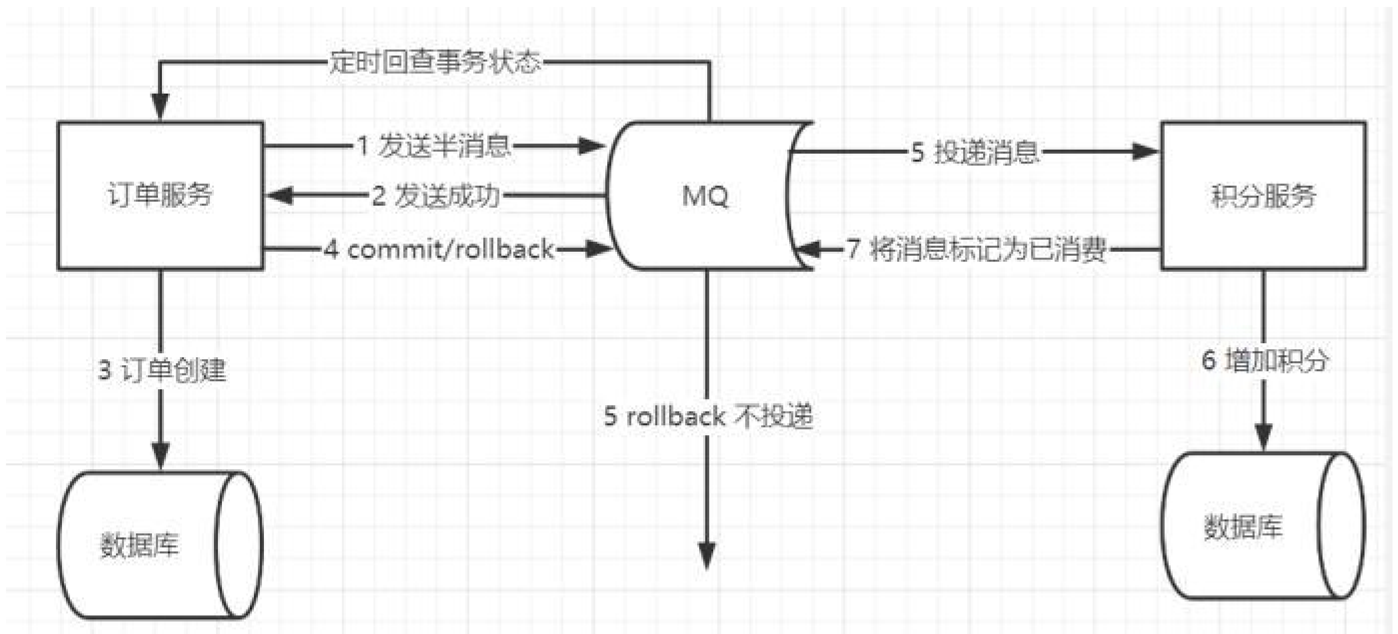
异步确保型事务使用示例

举个例子，假设存在业务规则：某笔订单成功后，为用户加一定的积分。

在这条规则里，管理订单数据源的服务为事务发起方，管理积分数据源的服务为事务跟随者。

从这个过程可以看到，基于消息队列实现的事务存在以下操作：

- 订单服务创建订单，提交本地事务
- 订单服务发布一条消息
- 积分服务收到消息后加积分



我们可以看到它的整体流程是比较简单的，同时业务开发工作量也不大：

- 编写订单服务里订单创建的逻辑
- 编写积分服务里增加积分的逻辑

可以看到该事务形态过程简单，性能消耗小，发起方与跟随方之间的流量峰谷可以使用队列填平，同时业务开发工作量也基本与单机事务没有差别，都不需要编写反向的业务逻辑过程

因此基于消息队列实现的事务是我们除了单机事务外最优先考虑使用的形态。

基于阿里 RocketMQ实现MQ异步确保型事务

> 有一些第三方的MQ是支持事务消息的，这些消息队列，支持半消息机制，比如RocketMQ， ActiveMQ。但是有一些常用的MQ也不支持事务消息，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：

1.producer(本例中指A系统)发送半消息到broker，这个半消息不是说消息内容不完整， 它包含完整的消息内容， 在producer端和普通消息的发送逻辑一致

2.broker存储半消息，半消息存储逻辑与普通消息一致，只是属性有所不同， topic是固定的RMQ_SYS_TRANS_HALF_TOPIC， queueId也是固定为0，这个tiopic中的消息对消费者是不可见的，所以里面的消息永远不会被消费。这就保证了在半消息提交成功之前，消费者是消费不到这个半消息的

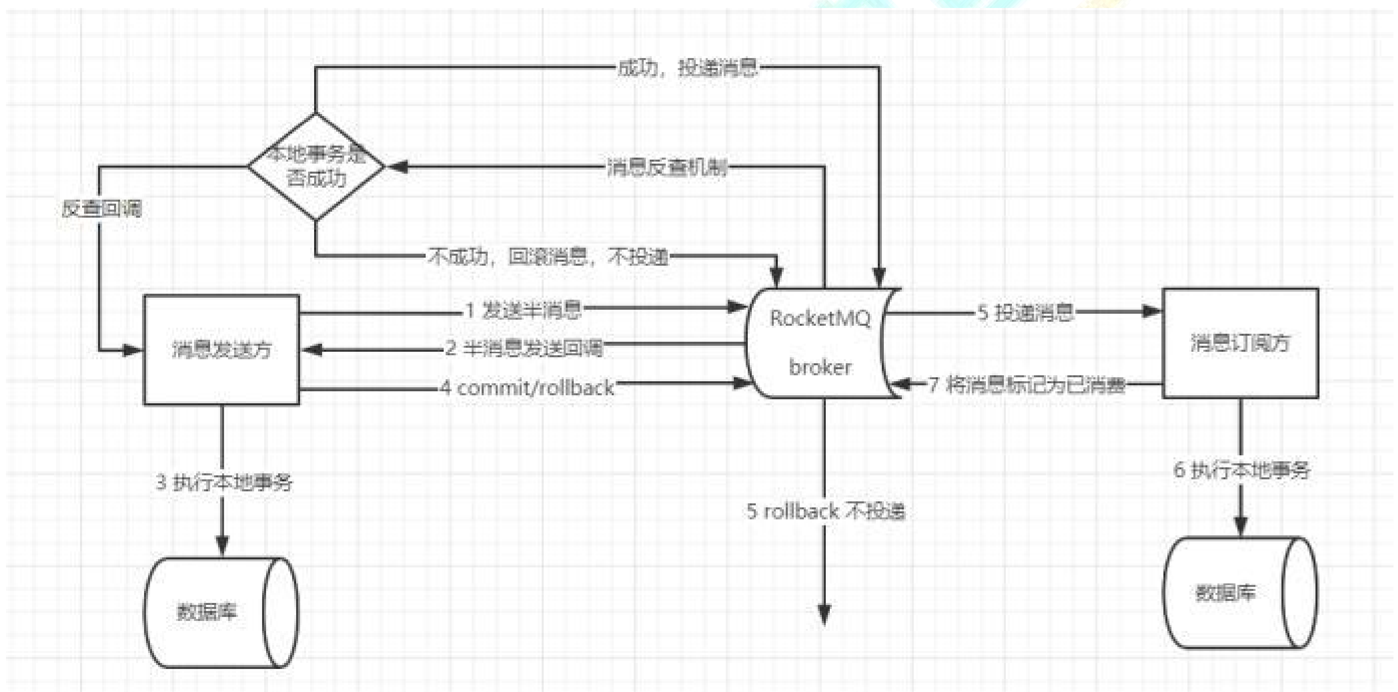
3.broker端半消息存储成功并返回后， A系统执行本地事务，并根据本地事务的执行结果来决定半消息的提交状态为提交或者回滚

4.A系统发送结束半消息的请求，并带上提交状态(提交 or 回滚)

5.broker端收到请求后，首先从RMQ_SYS_TRANS_HALF_TOPIC的queue中查出该消息，设置为完成状态。如果消息状态为提交，则把半消息从RMQ_SYS_TRANS_HALF_TOPIC队列中复制到这个消息原始topic的queue中去(之后这条消息就能被正常消费了)；如果消息状态为回滚，则什么也不做。

6.producer发送的半消息结束请求是 oneway 的，也就是发送后就不管了，只靠这个是无法保证半消息一定被提交的， rocketMq提供了一个兜底方案，这个方案叫消息反查机制， Broker启动时，会启动一个TransactionalMessageCheckService 任务，该任务会定时从半消息队列中读出所有超时未完成的半消息，针对每条未完成的消息， Broker会给对应的Producer发送一个消息反查请求，根据反查结果来决定这个半消息是需要提交还是回滚，或者后面继续来反查

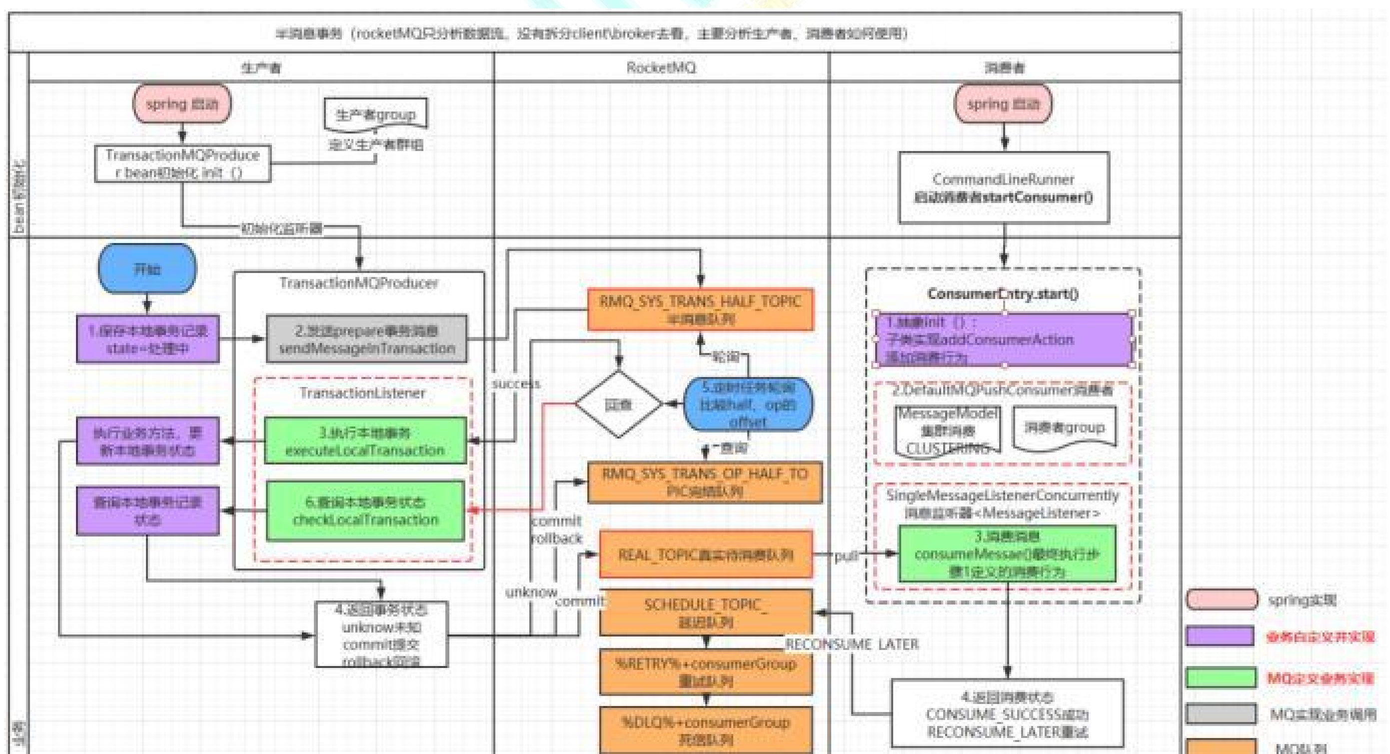
7.consumer(本例中指B系统)消费消息，执行本地数据变更(至于B是否能消费成功，消费失败是否重试，这属于正常消息消费需要考虑的问题)



在rocketMq中，不论是producer收到broker存储半消息成功返回后执行本地事务，还是broker向producer反查消息状态，都是通过**回调机制**完成，我把producer端的代码贴出来你就明白了：

```
TransactionListener transactionListener =new TransactionListener() {
    //broker返回半消息存储成功后回调该方法
    public LocalTransactionState executeLocalTransaction(Message msg, Object arg) {
        //执行本地事务,比如更新订单状态为成功
        //更新成功则return LocalTransactionState.COMMIT_MESSAGE;
        //更新失败则return LocalTransactionState.ROLLBACK_MESSAGE
    }
    //broker反查消息状态时毁掉该方法
    public LocalTransactionState checkLocalTransaction(MessageExt msg) {
        //根据消息中的订单号查询数据库中状态
        //状态为成功则return LocalTransactionState.COMMIT_MESSAGE;
        //状态为失败则return LocalTransactionState.ROLLBACK_MESSAGE
    }
};
TransactionMQProducer producer = new TransactionMQProducer("testTransactionMsg");
producer.setTransactionListener(transactionListener);
producer.start();
producer.sendMessageInTransaction(msg, arg);
```

半消息发送时，会传入一个回调类TransactionListener，使用时必须实现其中的两个方法， executeLocalTransaction 方法会在broker返回半消息存储成功后执行，我们会在其中执行本地事务； checkLocalTransaction方法会在broker向producer发起反查时执行，我们会在其中查询库表状态。两个方法的返回值都是消息状态，就是告诉broker应该提交或者回滚半消息

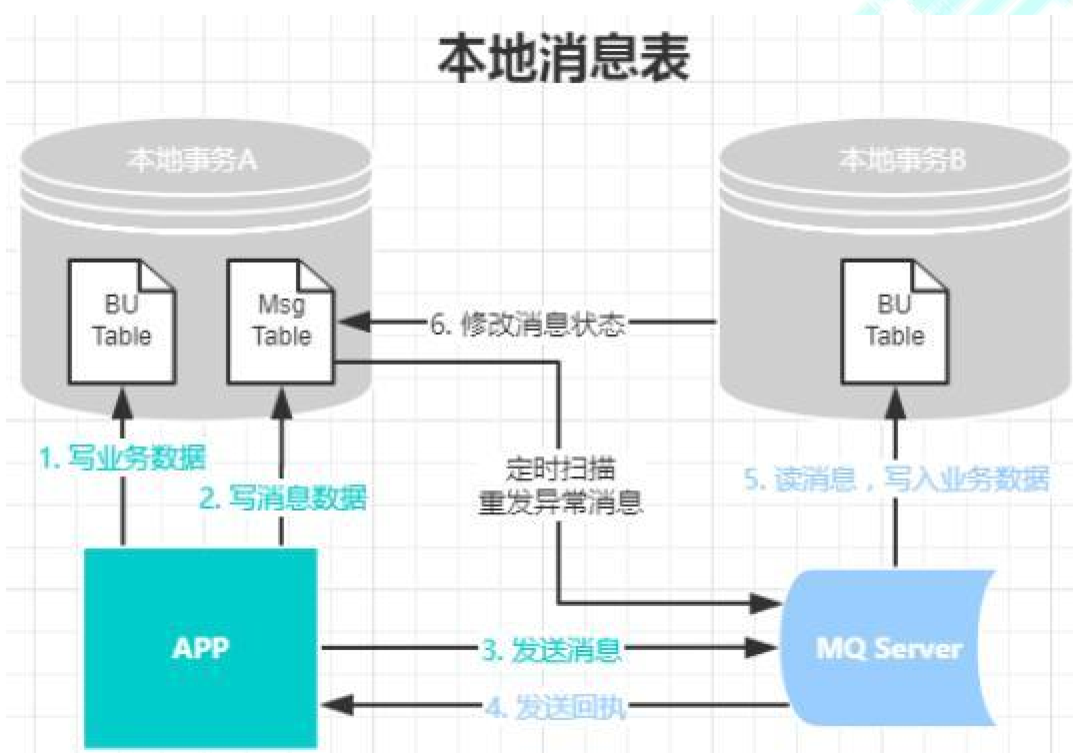


本地消息表方案

有时候我们目前的MQ组件并不支持事务消息，或者我们想尽量少的侵入业务方。这时我们需要另外一种方案“基于DB本地消息表“。

本地消息表最初由eBay 提出来解决分布式事务的问题。是目前业界使用的比较多的方案之一，它的核心思想就是将分布式事务**拆分**成本地事务进行处理。

本地消息表流程



发送消息方：

- 需要有一个消息表，记录着消息状态相关信息。
- 业务数据和消息表在同一个数据库，要保证它俩在同一个本地事务。直接利用本地事务，将业务数据和事务消息直接写入数据库。
- 在本地事务中处理完业务数据和写消息表操作后，通过写消息到 MQ 消息队列。使用专门的投递工作线程进行事务消息投递到MQ，根据投递

ACK去删除事务消息表记录
- 消息会发到消息消费方，如果发送失败，即进行重试。

消息消费方：

- 处理消息队列中的消息，完成自己的业务逻辑。
- 如果本地事务处理成功，则表明已经处理成功了。
- 如果本地事务处理失败，那么就会重试执行。
- 如果是业务层面的失败，给消息生产方发送一个业务补偿消息，通知进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

本地消息表优缺点：

优点：

- 本地消息表建设成本比较低，实现了可靠消息的传递确保了分布式事务的最终一致性。
- 无需提供回查方法，进一步减少的业务的侵入。
- 在某些场景下，还可以进一步利用注解等形式进行解耦，有可能实现无业务代码侵入式的实现。

缺点：

- 本地消息表与业务耦合在□ 起，难于做成通□ 性，不可□□ 伸缩。
- 本地消息表是基于数据库来做的，而数据库是要读写磁盘IO的，因此在高并发下是有性能瓶颈的

MQ事务消息 VS 本地消息表

二者的共性：

1、 事务消息都依赖MQ进行事务通知，所以都是异步的。

2、 事务消息在投递方都是存在重复投递的可能，需要有配套的机制去降低重复投递率，实现更友好的消息投递去重。

3、 事务消息的消费方，因为投递重复的无法避免，因此需要进行消费去重设计或者服务幂等设计。

二者的区别：

MQ事务消息：

- 需要MQ支持半消息机制或者类似特性，在重复投递上具有比较好的去重处理；
- 具有比较大的业务侵入性，需要业务方进行改造，提供对应的本地操作成功的回查功能；

DB本地消息表：

- 使用了数据库来存储事务消息，降低了对MQ的要求，但是增加了存储成本；
- 事务消息使用了异步投递，增大了消息重复投递的可能性；

分类	共同点	优势	弊端
本地消息表	都需要自己写业务补偿代码	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。	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。
MQ事务消息	都需要自己写业务补偿代码	实现了最终一致性，不需要依赖本地数据库事务。 用消息队列的方式实现分布式事务,效率较高	目前主流MQ中有ActiveMQ RocketMQ支持事务消息 实现难度较大,和业务耦合比较紧密

最大努力通知

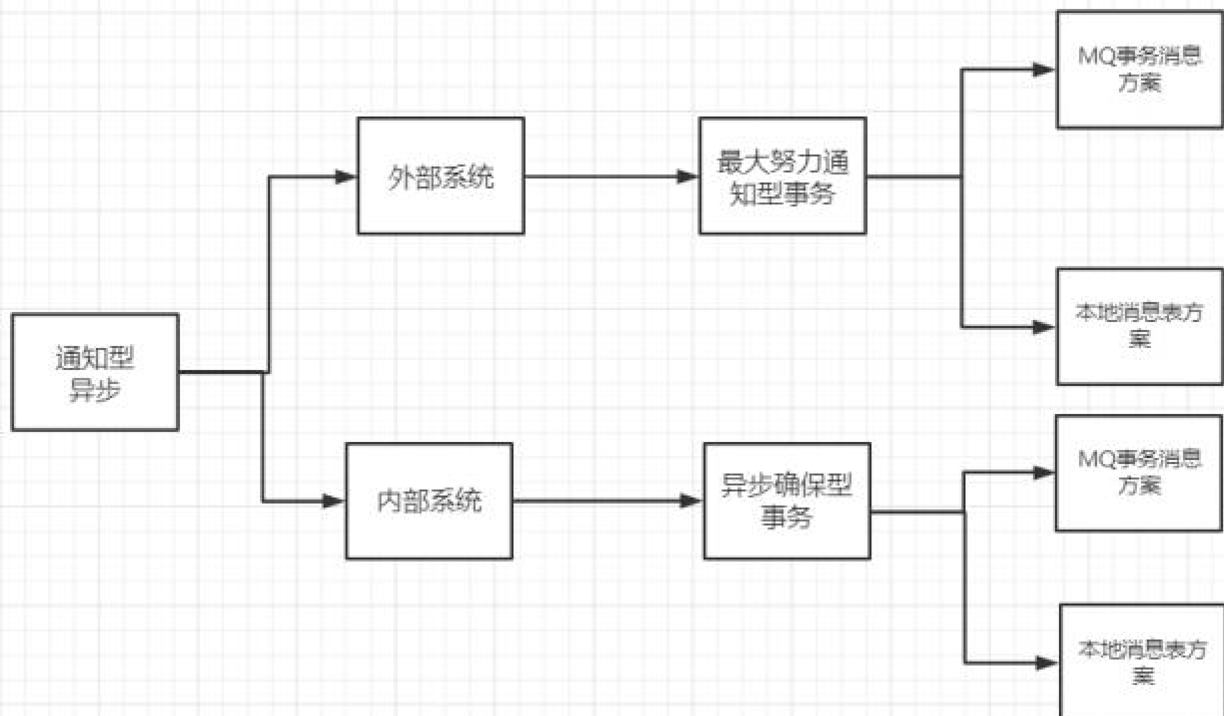
最大努力通知方案的目标，就是发起通知方通过一定的机制，最大努力将业务处理结果通知到接收方。

最大努力通知型的最终一致性：

> 本质是通过引入**定期校验机制**实现最终一致性，对业务的侵入性较低，适合于对最终一致性敏感度比较低、业务链路较短的场景。

最大努力通知事务主要用于**外部系统**，因为外部的网络环境更加复杂和不可信，所以只能尽最大努力去通知实现数据最终一致性， **比如充值平台与运营商、支付对接、商户通知等等跨平台、跨企业的系统间业务交互场景**；

而**异步确保型事务**主要适用于**内部系统**的数据最终一致性保障，因为内部相对比较可控，比如订单和购物车、收货与清算、支付与结算等等场景。



普通消息是无法解决本地事务执行和消息发送的一致性问题的。因为消息发送是一个网络通信的过程，发送消息的过程就有可能出现发送失败、或者超时的情况。超时有可能发送成功了，有可能发送失败了，消息的发送方是无法确定的，所以此时消息发送方无论是提交事务还是回滚事务，都有可能不一致性出现。

所以，通知型事务的难度在于： **投递消息和参与者本地事务的一致性保障。**

因为核心要点一致，都是为了保证消息的一致性投递，所以，最大努力通知事务在投递流程上跟异步确保型是一样的，因此也有**两个分支**：

- **基于MQ自身的事务消息方案**
- **基于DB的本地事务消息表方案**

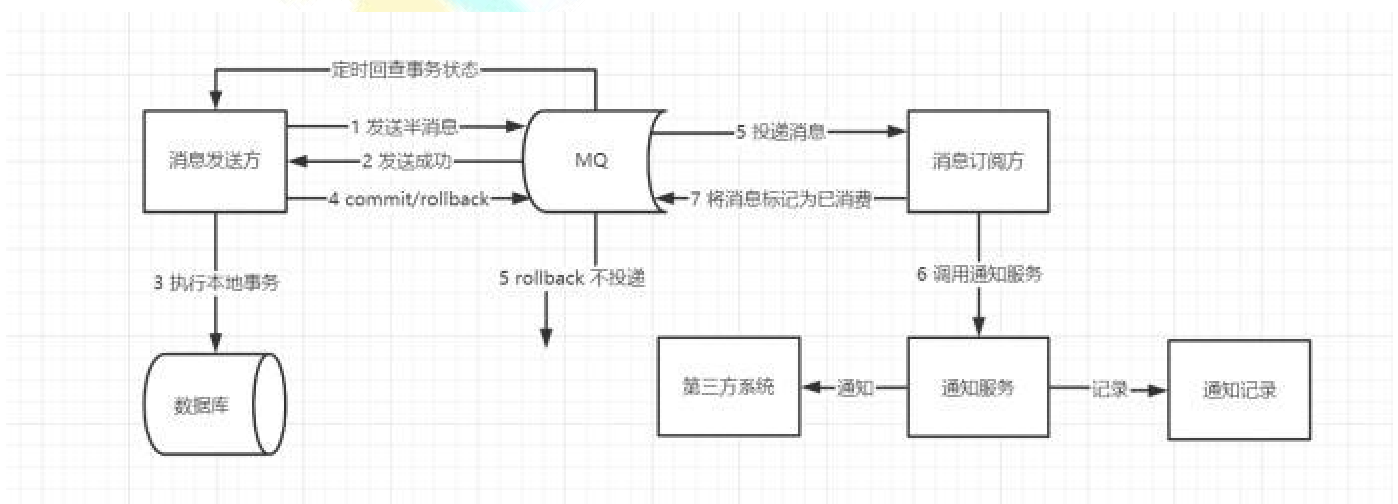
MQ事务消息方案

要实现最大努力通知，可以采用 MQ 的 ACK 机制。

最大努力通知事务在投递之前，跟异步确保型流程都差不多，关键在于投递后的处理。

因为异步确保型在于内部的事务处理，所以MQ和系统是直连并且无需严格的权限、安全等方面的思路设计。最大努力通知事务在于第三方系统的对接，所以最大努力通知事务有几个特性：

- 业务主动方在完成业务处理后，向业务被动方(第三方系统)发送通知消息，允许存在消息丢失。
- 业务主动方提供递增多挡位时间间隔(5min、10min、30min、1h、24h)，用于失败重试调用业务被动方的接口；在通知N次之后就不再通知，报警+记日志+人工介入。
- 业务被动方提供幂等的服务接口，防止通知重复消费。
- 业务主动方需要有定期校验机制，对业务数据进行兜底；防止业务被动方无法履行责任时进行业务回滚，确保数据最终一致性。



1. 业务活动的主动方，在完成业务处理之后，向业务活动的被动方发送消

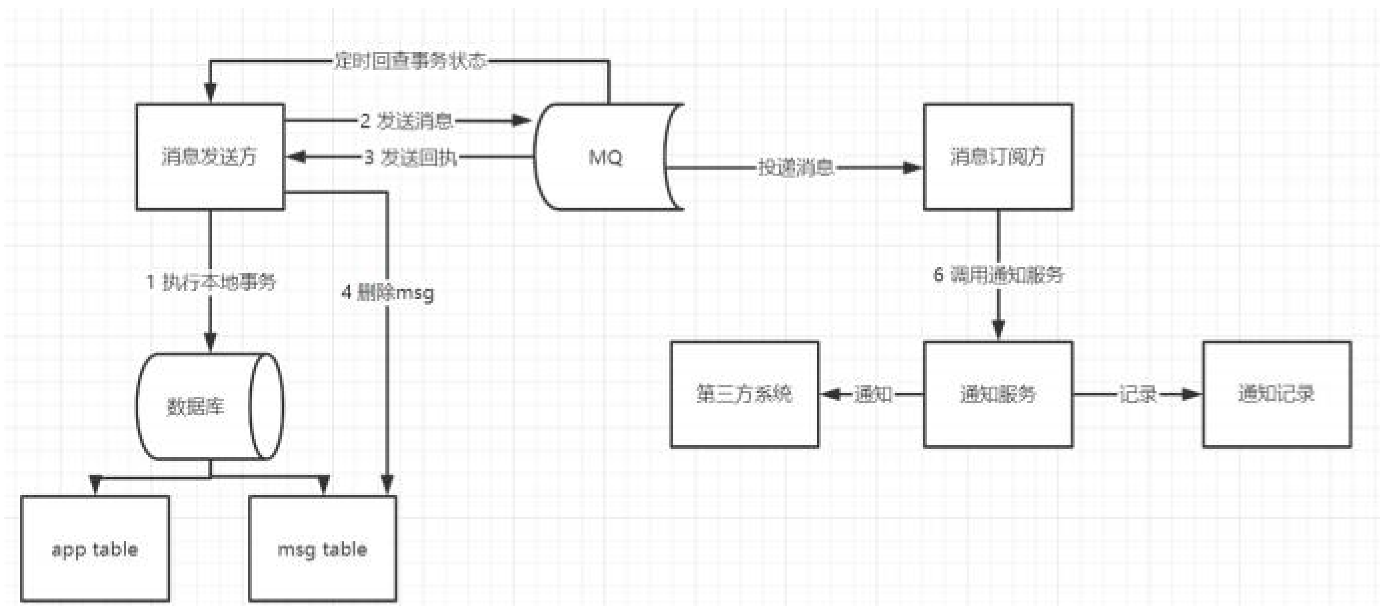
息，允许消息丢失。
2. 主动方可以设置时间阶梯型通知规则，在通知失败后按规则重复通知，直到通知N次后不再通知。
3. 主动方提供校对查询接口给被动方按需校对查询，用于恢复丢失的业务消息。
4. 业务活动的被动方如果正常接收了数据，就正常返回响应，并结束事务。
5. 如果被动方没有正常接收，根据定时策略，向业务活动主动方查询，恢复丢失的业务消息。

特点

1. 用到的服务模式：可查询操作、幂等操作；
2. 被动方的处理结果不影响主动方的处理结果；
3. 适用于对业务最终一致性的时间敏感度低的系统；
4. 适合跨企业的系统间的操作，或者企业内部□ 较□□ 的系统间的操作，比如银行通知、商户通知等；

本地消息表方案

要实现最大努力通知，可以采用 定期检查本地消息表的机制 。



发送消息方：

- 需要有一个消息表，记录着消息状态相关信息。
- 业务数据和消息表在同一个数据库，要保证它俩在同一个本地事务。直接利用本地事务，将业务数据和事务消息直接写入数据库。
- 在本地事务中处理完业务数据和写消息表操作后，通过写消息到 MQ 消息队列。使用专门的投递工作线程进行事务消息投递到MQ，根据投递ACK去删除事务消息表记录
- 消息会发到消息消费方，如果发送失败，即进行重试。
- 生产方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

最大努力通知事务在于第三方系统的对接，所以最大努力通知事务有几个特性：

- 业务主动方在完成业务处理后，向业务被动方(第三方系统)发送通知消息，允许存在消息丢失。
- 业务主动方提供递增多挡位时间间隔(5min、10min、30min、1h、

24h)，用于失败重试调用业务被动方的接口；在通知N次之后就不再通知，报警+记日志+人工介入。

- 业务被动方提供幂等的服务接口，防止通知重复消费。
- 业务主动方需要有定期校验机制，对业务数据进行兜底；防止业务被动方无法履行责任时进行业务回滚，确保数据最终一致性。

最大努力通知事务 VS 异步确保型事务

最大努力通知事务在我认知中，其实是基于异步确保型事务发展而来适用于外部对接的一种业务实现。他们主要有的是业务差别，如下：

• 从参与者来说：最大努力通知事务适用于跨平台、跨企业的系统间业务交互；异步确保型事务更适用于同网络体系的内部服务交付。

• 从消息层面说：最大努力通知事务需要主动推送并提供多档次时间的重试机制来保证数据的通知；而异步确保型事务只需要消息消费者主动去消费。

• 从数据层面说：最大努力通知事务还需额外的定期校验机制对数据进行兜底，保证数据的最终一致性；而异步确保型事务只需保证消息的可靠投递即可，自身无需对数据进行兜底处理。

通知型事务的问题

通知型事务，是无法解决本地事务执行和消息发送的一致性问题的。

因为消息发送是一个网络通信的过程，发送消息的过程就有可能出现发送失败、或者超时的情况。超时有可能发送成功了，有可能发送失败了，消息的发送方是无法确定的，所以此时消息发送方无论是提交事务还是回滚事务，都有可能不一致性出现。

消息发送一致性

消息中间件在分布式系统中的核心作用就是异步通讯、应用解耦和并发缓冲（也叫作流量削峰）。在分布式环境下，需要通过网络进行通讯，就引入了数据传输的不确定性，也就是CAP理论中的分区容错性。

消息发送一致性是指**产生消息的业务动作与消息发送动作一致**，也就是说如果业务操作成功，那么由这个业务操作所产生的消息一定要发送出去，否则就丢失。

常规MQ消息处理流程和特点

常规的MQ队列处理流程无法实现消息的一致性。所以，需要借助半消息、本地消息表，保障一致性。

消息重复发送问题和业务接口幂等性设计

对于未确认的消息，采用按规则重新投递的方式进行处理。

对于以上流程，消息重复发送会导致业务处理接口出现重复调用的问题。消息消费过程中消息重复发送的主要原因就是消费者成功接收处理完消息后，消息中间件没有及时更新投递状态导致的。如果允许消息重复发送，那么消费方应该实现业务接口的幂等性设计。

补偿型

但是基于消息实现的事务并不能解决所有的业务场景，例如以下场景：某笔订单完成时，同时扣掉用户的现金。

这里事务发起方是管理订单库的服务，但对整个事务是否提交并不能只由订单服务决定，因为还要确保用户有足够的钱，才能完成这笔交易，而这个信息在管理现金的服务里。这里我们可以引入基于补偿实现的事务，其流程如下：

- 创建订单数据，但暂不提交本地事务
- 订单服务发送远程调用到现金服务，以扣除对应的金额
- 上述步骤成功后提交订单库的事务

以上这个是正常成功的流程，异常流程需要回滚的话，将额外发送远程调用到现金服务以加上之前扣掉的金额。

以上流程比基于消息队列实现的事务的流程要复杂，同时开发的工作量也更多：

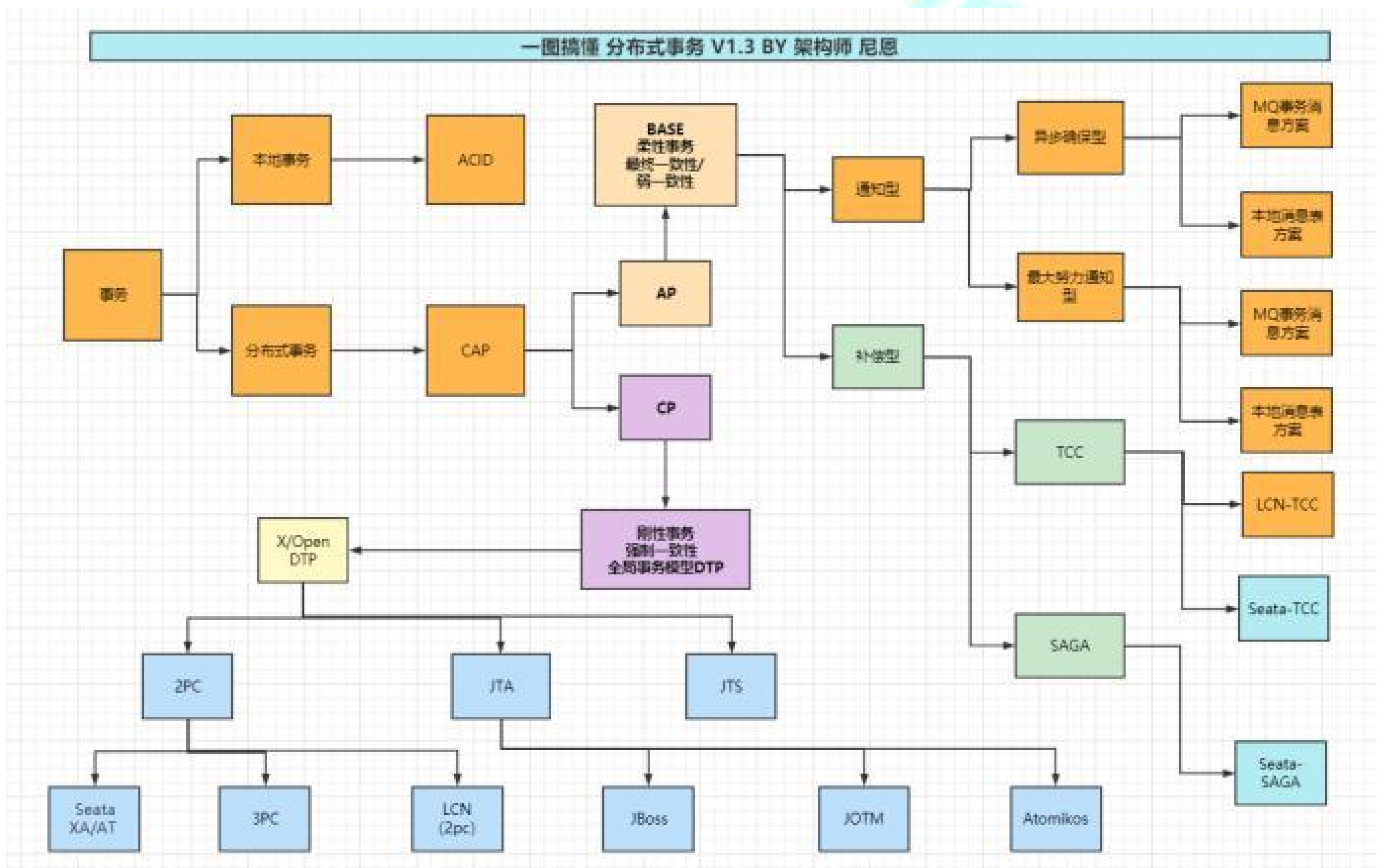
- 编写订单服务里创建订单的逻辑
- 编写现金服务里扣钱的逻辑
- 编写现金服务里补偿返还的逻辑

可以看到，该事务流程相对于基于消息实现的分布式事务更为复杂，需要额外开发相关的业务回滚方法，也失去了服务间流量削峰填谷的功能。但其仅仅只比基于消息的事务复杂多一点，若不能使用基于消息队列的最终一致性事务，那么可以优先考虑使用基于补偿的事务形态。

什么是补偿模式?

> 补偿模式使用一个额外的协调服务来协调各个需要保证一致性的业务服务，协调服务按顺序调用各个业务微服务，如果某个业务服务调用异常（包括业务异常和技术异常）就取消之前所有已经调用成功的业务服务。

补偿模式大致有TCC，和Saga两种细分的方案



TCC 事务模型

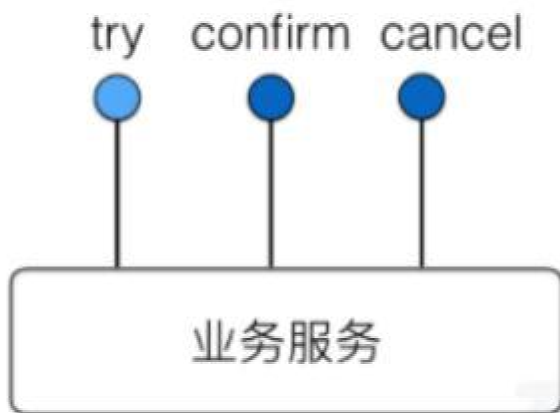
什么是TCC 事务模型

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 分布式事务模型包括三部分：

1.**主业务服务**：主业务服务为整个业务活动的发起方，服务的编排者，负责发起并完成整个业务活动。

2.**从业务服务**：从业务服务是整个业务活动的参与方，负责提供 TCC 业务操作，实现初步操作(Try)、确认操作(Confirm)、取消操作(Cancel)三个接口，供主业务服务调用。



3.**业务活动管理器**：业务活动管理器管理控制整个业务活动，包括记录维护 TCC 全局事务的事务状态和每个从业务服务的子事务状态，并在业务活动提交时调用所有从业务服务的 Confirm 操作，在业务活动取消时调用所有从业务服务的 Cancel 操作。

> 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，目的是解决复杂业务中，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 两个阶段，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的工作流程

TCC(Try-Confirm-Cancel)分布式事务模型相对于 XA 等传统模型，其特征在于**它不依赖资源管理器(RM)对分布式事务的支持，而是通过对业务逻辑的分解来实现分布式事务。**

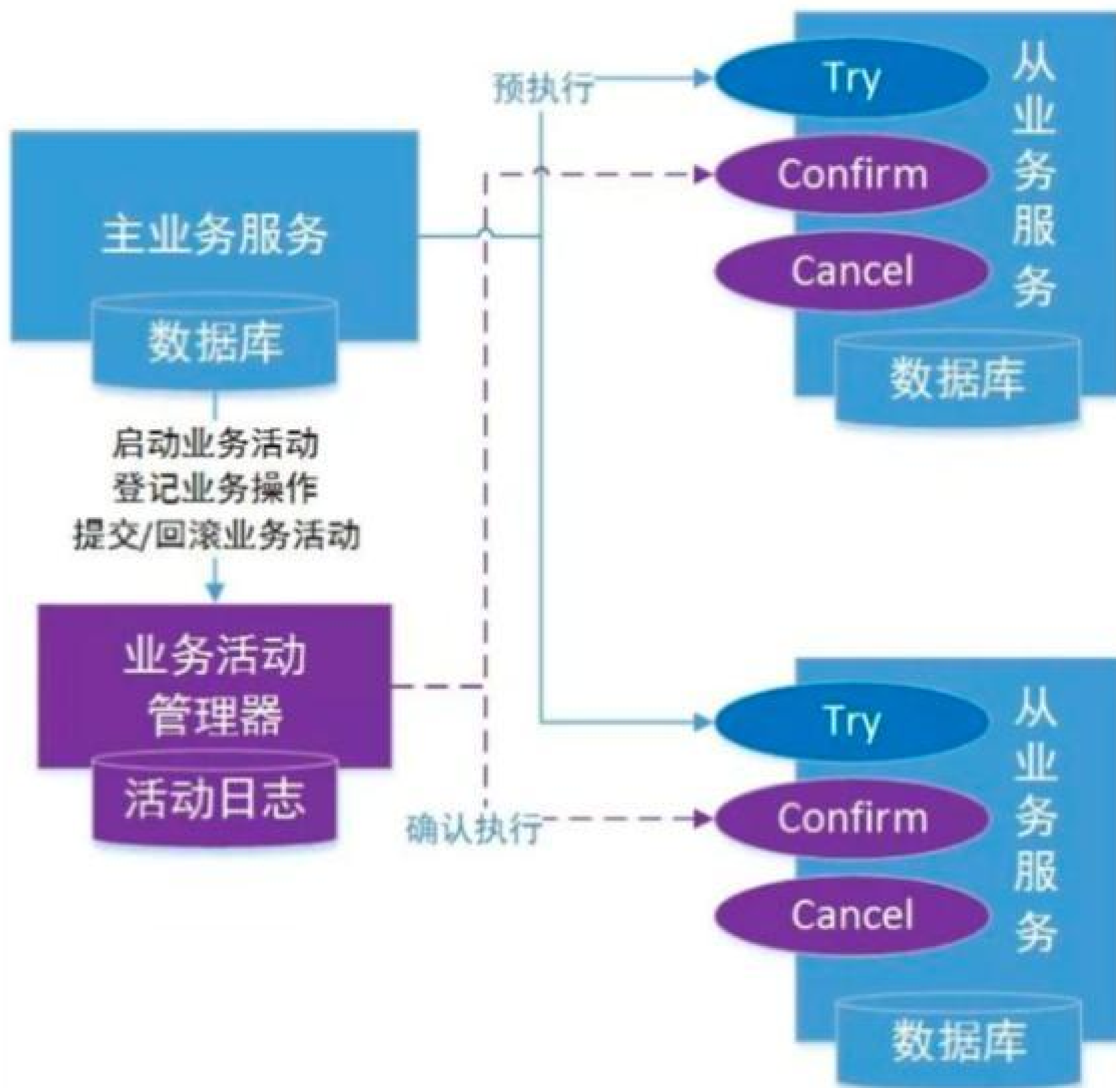
TCC 模型认为对于业务系统中一个特定的业务逻辑，其对外提供服务时，必须接受一些不确定性，即对业务逻辑初步操作的调用仅是一个临时性操作，调用它的主业务服务保留了后续的取消权。如果主业务服务认为全局事务应该回滚，它会要求取消之前的临时性操作，这就对应从业务服务的取消操作。而当主业务服务认为全局事务应该提交时，它会放弃之前临时性操作的取消权，这对应从业务服务的确认操作。每一个初步操作，最终都会被确认或取消。

因此，针对一个具体的业务服务， TCC 分布式事务模型需要业务系统提供三段业务逻辑：

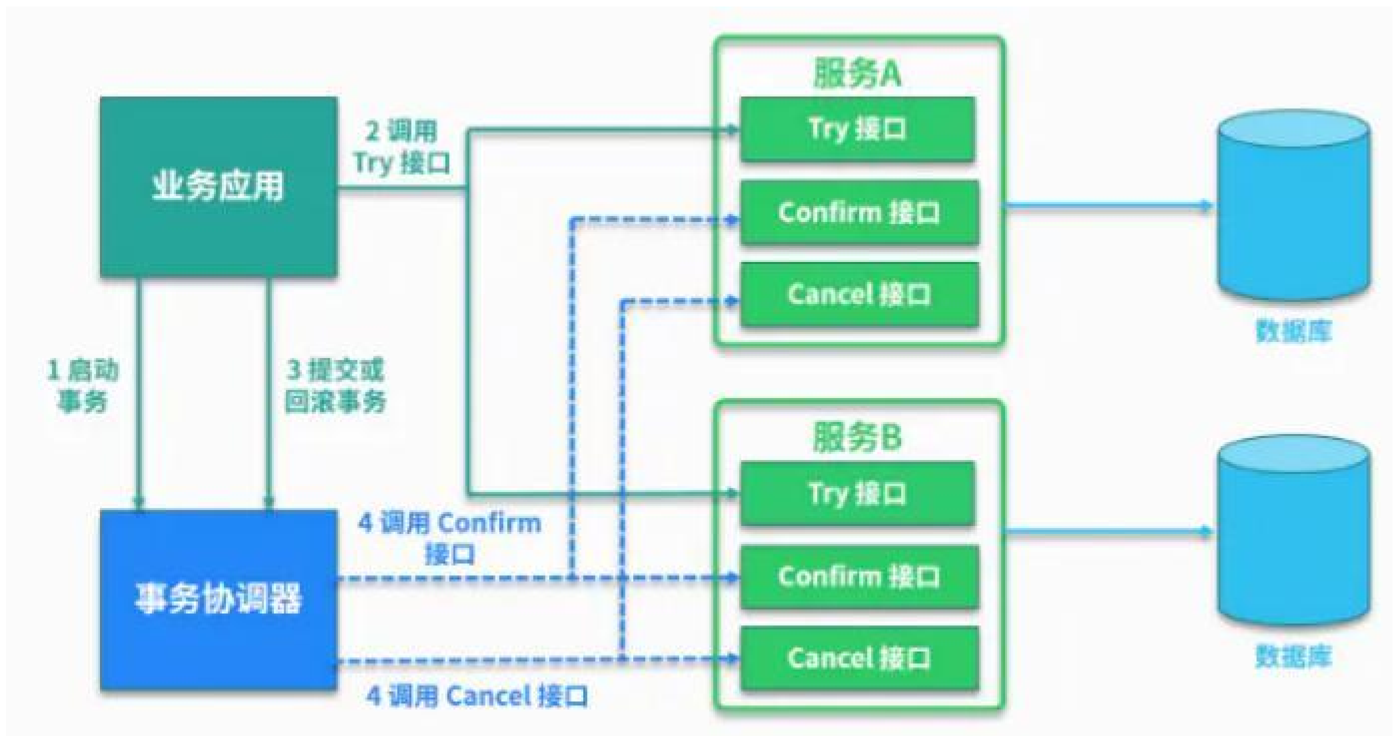
初步操作 Try：完成所有业务检查，预留必须的业务资源。

确认操作 Confirm：真正执行的业务逻辑，不作任何业务检查，只使用 Try 阶段预留的业务资源。因此，只要 Try 操作成功， Confirm 必须能成功。另外， Confirm 操作需满足幂等性，保证一笔分布式事务有且只能成功一次。

取消操作 Cancel：释放 Try 阶段预留的业务资源。同样的， Cancel 操作也需要满足幂等性。



TCC 分布式事务模型包括三部分：



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 阶段预留的业务资源。

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，所以这两个阶段需要支持幂等；如果重试失败，则需要人工介入进行恢复和处理等。

TCC事务案例

然而基于补偿的事务形态也并非能实现所有的需求，如以下场景：某笔订单完成时，同时扣掉用户的现金，但交易未完成，也未被取消时，不能让客户看到钱变少了。

这时我们可以引入TCC，其流程如下：

- 订单服务创建订单
- 订单服务发送远程调用到现金服务，冻结客户的现金
- 提交订单服务数据
- 订单服务发送远程调用到现金服务，扣除客户冻结的现金

以上是正常完成的流程，若为异常流程，则需要发送远程调用请求到现金服务，撤销冻结的金额。

以上流程比基于补偿实现的事务的流程要复杂，同时开发的工作量也更多：

- 订单服务编写创建订单的逻辑
- 现金服务编写冻结现金的逻辑
- 现金服务编写扣除现金的逻辑
- 现金服务编写解冻现金的逻辑

TCC实际上是最为复杂的一种情况，其能处理所有的业务场景，但无论出于性能上的考虑，还是开发复杂度上的考虑，都应该尽量避免该类事务。

TCC事务模型的要求：

1. 可查询操作：服务操作具有全局唯一的标识，操作唯一的确定的时间。
2. 幂等操作：重复调用多次产生的业务结果与调用一次产生的结果相同。

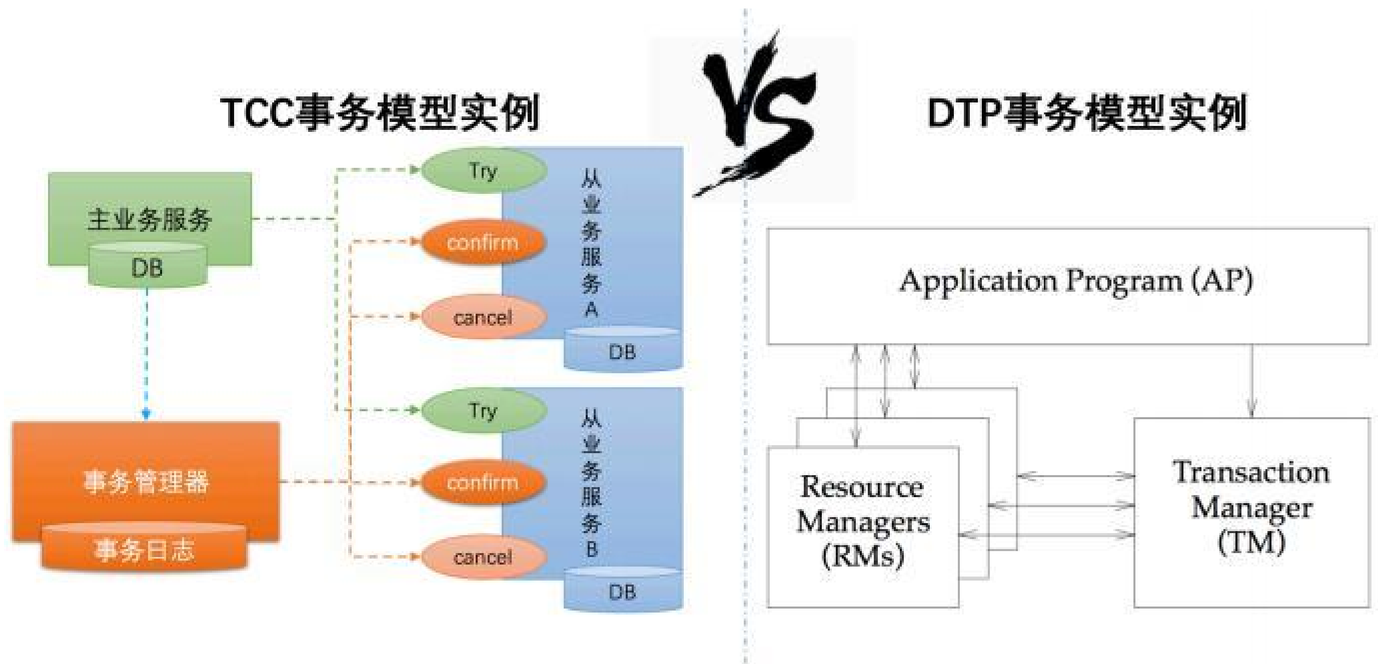
   一是通过业务操作实现幂等性，二是系统缓存所有请求与处理的结果，最后是检测到重复请求之后，自动返回之前的处理结果。

3. TCC操作： Try阶段，尝试执行业务，完成所有业务的检查，实现一致性；预留必须的业务资源，实现准隔离性。 Confirm阶段：真正的去执行业务，不做任何检查，仅适用Try阶段预留的业务资源， Confirm操作还要满足幂等性。 Cancel阶段：取消执行业务，释放Try阶段预留的业务资源， Cancel操作要满足幂等性。 TCC与2PC(两阶段提交)协议的区别： TCC位于业务服务层而不是资源层， TCC没有单独准备阶段， Try操作兼备资源操作与准备的能力， TCC中Try操作可以灵活的选择业务资源，锁定粒度。 TCC的开发成本比2PC高。实际上TCC也属于两阶段操作，但是TCC不等同于2PC操作。
4. 可补偿操作： Do阶段：真正的执行业务处理，业务处理结果外部可见。 Compensate阶段：抵消或者部分撤销正向业务操作的业务结果，

   补偿操作满足幂等性。约束：补偿操作在业务上可行，由于业务执行结果未隔离或者补偿不完整带来的风险与成本可控。实际上， TCC的Confirm和Cancel操作可以看做是补偿操作。

TCC事务模型 VS DTP事务模型

比较一下TCC事务模型和DTP事务模型，如下所示：



这两张图看起来差别较大，实际上很多地方是类似的!

1、TCC模型中的 主业务服务 相当于 DTP模型中的AP， TCC模型中的从业务服务 相当于 DTP模型中的RM

- 在DTP模型中，应用AP操作多个资源管理器RM上的资源；而在TCC模型中，是主业务服务操作多个从业务服务上的资源。例如航班预定案例中，美团App就是主业务服务，而川航和东航就是从业务服务，主业务服务需要使用从业务服务上的机票资源。不同的是DTP模型中的资源提供者是类似于Mysql这种关系型数据库，而TCC模型中资源的提供者是其他业务服务。

  2、TCC模型中，从业务服务提供的try、confirm、cancel接口 相当于DTP模型中RM提供的prepare、commit、rollback接口

- XA协议中规定了DTP模型中定RM需要提供prepare、commit、rollback接口给TM调用，以实现两阶段提交。
- 而在TCC模型中，从业务服务相当于RM，提供了类似的try、confirm、cancel接口。

3、事务管理器

- DTP模型和TCC模型中都有一个事务管理器。不同的是：
- 在DTP模型中，阶段1的(prepare)和阶段2的(commit、rollback)，都是由TM进行调用的。
- 在TCC模型中，阶段1的try接口是主业务服务调用(绿色箭头)，阶段2的(confirm、cancel接口)是事务管理器TM调用(红色箭头)。这就是 TCC 分布式事务模型的二阶段异步化功能，从业务服务的第一阶段执行成功，主业务服务就可以提交完成，然后再由事务管理器框架异步的执行各从业务服务的第二阶段。这里牺牲了一定的隔离性和一致性的，但是提高了长事务的可用性。

TCC与2PC对比

TCC其实本质和2PC是差不多的：

- T就是Try，两个C分别是Confirm和Cancel。
- Try就是尝试，请求链路中每个参与者依次执行Try逻辑，如果都成功，就再执行Confirm逻辑，如果有失败，就执行Cancel逻辑。

TCC与XA两阶段提交有着异曲同工之妙，下图列出了二者之间的对比

1. 在阶段1：

- 在XA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，是主业务活动请求(try)各个从业务服务预留资源。

1. 在阶段2：

- XA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成功，那么就commit每个事务分支，反之则rollback每个事务分支。
- TCC中，如果在第一阶段所有业务资源都预留成功，那么confirm各个从业务服务，否则取消(cancel)所有从业务服务的资源预留请求。

TCC和2PC不同的是：

- XA是资源层面的分布式事务，强一致性，在两阶段提交的整个过程中，一直会持有资源的锁。基于数据库锁实现，需要数据库支持XA协议，由于在执行事务的全程都需要对相关数据加锁，一般高并发性能会比较差
- TCC是业务层面的分布式事务，最终一致性，不会一直持有资源的锁，性能较好。但是对微服务的侵入性强，微服务的每个事务都必须实现try、confirm、cancel等3个方法，开发成本高，今后维护改造的成本也高为了达到事务的一致性要求， try、confirm、cancel接口必须实现幂等性操作由于事务管理器要记录事务日志，必定会损耗一定的性能，并使得整个TCC事务时间拉长

> TCC它会弱化每个步骤中对于资源的锁定，以达到一个能承受高并发的目的（基于最终一致性）。

具体的说明如下：

XA是资源层面的分布式事务，强一致性，在两阶段提交的整个过程中，一直会持有资源的锁。

XA事务中的两阶段提交内部过程是对开发者屏蔽的，开发者从代码层面是感知不到这个过程的。而事务管理器在两阶段提交过程中，从prepare到commit/rollback过程中，资源实际上一直都是被加锁的。如果有其他人需要更新这两条记录，那么就必须等待锁释放。

TCC是业务层面的分布式事务，最终一致性，不会一直持有资源的锁。

TCC中的两阶段提交并没有对开发者完全屏蔽，也就是说从代码层面，开发者是可以感受到两阶段提交的存在。 try、confirm/cancel在执行过程中，一般都会开启各自的本地事务，来保证方法内部业务逻辑的ACID特性。其中:

1、try过程的本地事务，是保证资源预留的业务逻辑的正确性。

2、confirm/cancel执行的本地事务逻辑确认/取消预留资源，以保证最终一致性，也就是所谓的补偿型事务(Compensation-Based Transactions)。由于是多个□□ 的本地事务，因此不会对资源□ 直加锁。

另外，这里提到confirm/cancel执行的本地事务是 **补偿性事务:**

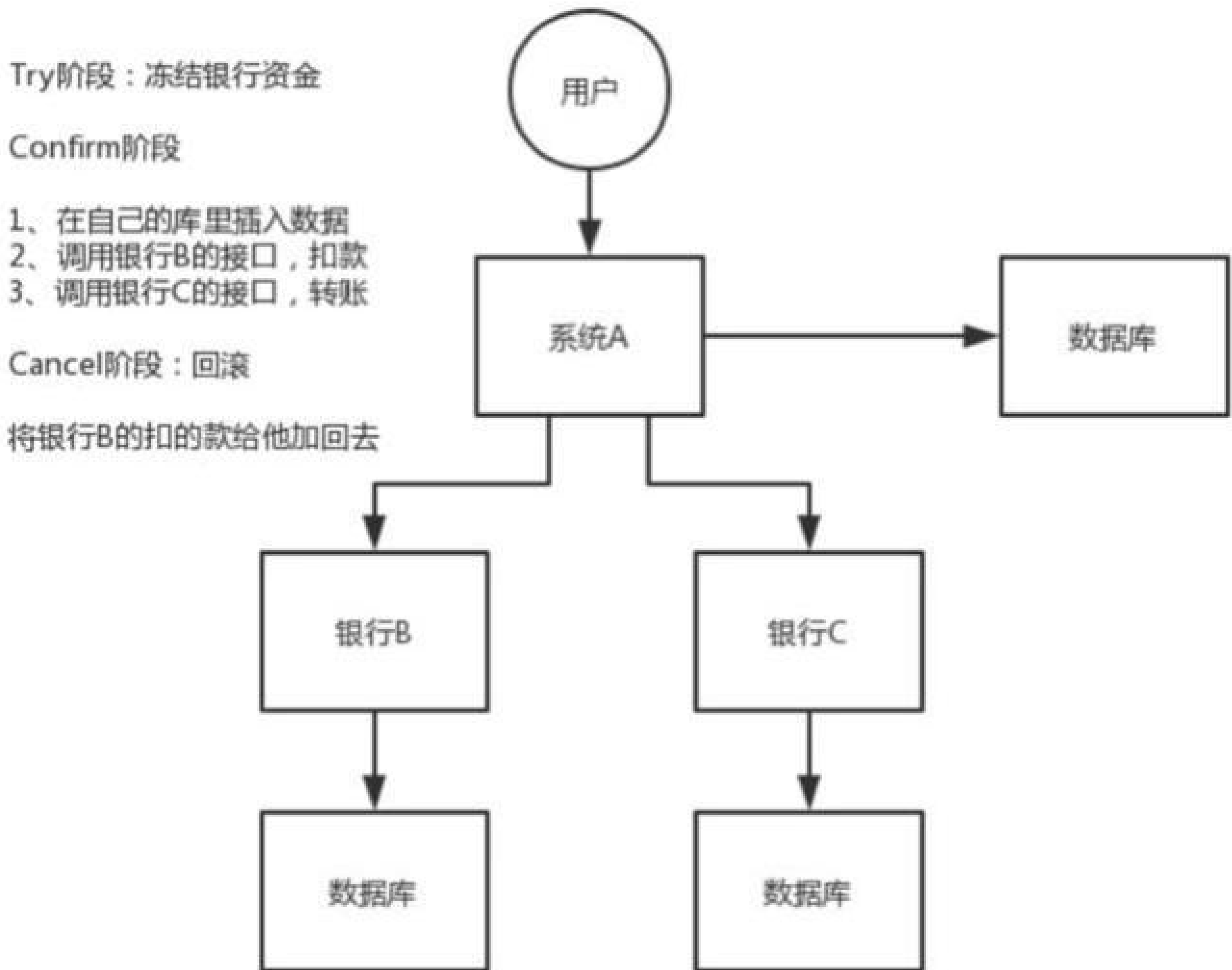
补偿是□ 个□□ 的□ 持ACID特性的本地事务，用于在逻辑上取消服务提供
者上一个ACID事务造成的影响，对于一个长事务(long-running transaction)，与其实现一个巨大的分布式ACID事务，不如使用基于补偿性的方案，把每一次服务调用当做一个较短的本地ACID事务来处理，执行完就立即提交

TCC 的使用场景

TCC是可以解决部分场景下的分布式事务的，但是，它的一个问题在于，需要每个参与者都分别实现Try， Confirm和Cancel接口及逻辑，这对于业务的侵入性是巨大的。

TCC 方案严重依赖回滚和补偿代码，最终的结果是：回滚代码逻辑复杂，业务代码很难维护。所以， TCC 方案的使用场景较少，但是也有使用的场景。

比如说跟钱打交道的，支付、交易相关的场景，大家会用 TCC方案，严格保**证分布式事务要么全部成功，要么全部自动回滚**，严格保证资金的正确性，保证在资金上不会出现问题。



SAGA长事务模型

SAGA可以看做一个异步的、利用队列实现的补偿事务。

Saga相关概念

1987年普林斯顿大学的Hector Garcia-Molina和Kenneth Salem发表了一篇Paper Sagas，讲述的是如何处理long lived transaction （长活事务）。Saga是一个长活事务可被分解成可以交错运行的子事务集合。其中每个子事务都是一个保持数据库一致性的真实事务。
论文地址： sagas

Saga模型是把一个分布式事务拆分为多个本地事务，每个本地事务都有相应的执行模块和补偿模块（对应TCC中的Confirm和Cancel），当Saga事务中任意一个本地事务出错时，可以通过调用相关的补偿方法恢复之前的事务，达到事务最终一致性。

这样的SAGA事务模型，是牺牲了一定的隔离性和一致性的，但是提高了long-running事务的可用性。

Saga 模型由三部分组成：

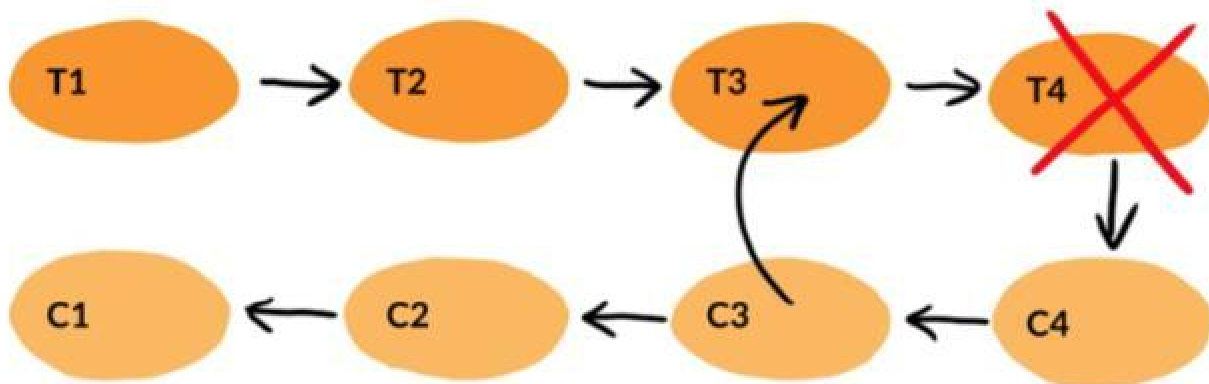
- **LLT**（Long Live Transaction）：由一个个本地事务组成的事务链。
- **本地事务**：事务链由一个个子事务（本地事务）组成， LLT = T1+T2+T3+...+Ti。
- **补偿**：每个本地事务 Ti 有对应的补偿 Ci。

Saga的执行顺序有两种：

- T1, T2, T3, ..., Tn
- T1, T2, ..., Tj, Cj,..., C2, C1，其中0 < j < n

Saga 两种恢复策略：

- **向后恢复**（Backward Recovery）：撤销掉之前所有成功子事务。如果任意本地子事务失败，则补偿已完成的事务。如异常情况的执行顺序T1,T2,T3,..Ti,Ci,...C3,C2,C1。



- **向前恢复**（Forward Recovery）：即重试失败的事务，适用于必须要成功的场景，该情况下不需要Ci。执行顺序： T1,T2,...,Tj （失败） ,Tj （重试） ,...,Ti。

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。在这种情况下我们能做些什么？ 最后的手段是提供回退措施，比如人工干预。

Saga的使用条件

Saga看起来很有希望满足我们的需求。所有长活事务都可以这样做吗？这里有一些限制：

1. Saga只允许**两个层次的嵌套**，顶级的Saga和简单子事务
2. 在外层，全原子性不能得到满足。也就是说， sagas可能会看到其他sagas的部分结果
3. 每个□ 事务应该是□□ 的原□□ 为

4. 在我们的业务场景下，各个业务环境（如：航班预订、租车、酒店预订和付款）是□ 然□□ 的□ 为，□ 且每个事务都可以□ 对应服务的数据库
保证原子操作。

补偿也有需考虑的事项：

- 补偿事务从语义角度撤消了事务Ti的行为，但未必能将数据库返回到执行Ti时的状态。（例如，如果事务触发导弹发射， 则可能无法撤消此操作）

但这对我们的业务来说不是问题。其实难以撤消的行为也有可能被补偿。例如，发送电邮的事务可以通过发送解释问题的另一封电邮来补偿。

对于ACID的保证：

Saga对于ACID的保证和TCC一样：

- 原子性（Atomicity）：正常情况下保证。
- 一致性（Consistency），在某个时间点，会出现A库和B库的数据违反一致性要求的情况，但是最终是一致的。
- 隔离性（Isolation），在某个时间点， A事务能够读到B事务部分提交的结果。
- 持久性（Durability），和本地事务一样，只要commit则数据被持久。

Saga不提供ACID保证，因为原子性和隔离性不能得到满足。原论文描述如下：

```
full atomicity is not provided. That is, sagas may view
the partial results of other sagas
```

通过saga log，saga可以保证一致性和持久性。

SAGA模型的解决方案

SAGA模型的核心思想是，通过某种方案，将分布式事务转化为本地事务，从而降低问题的复杂性。

比如以DB和MQ的场景为例，业务逻辑如下：

1. 向DB中插入一条数据。
2. 向MQ中发送一条消息。

由于上述逻辑中，对应了两种存储端，即DB和MQ，所以，简单的通过本地事务是无法解决的。那么，依照SAGA模型，可以有两种解决方案。

方案一：半消息模式。

RocketMQ新版本中，就支持了这种模式。

首先，我们理解什么是半消息。简单来说，就是在消息上加了一个状态。

当发送者第一次将消息放入MQ后，该消息为待确认状态。该状态下，该消息是不能被消费者消费的。发送者必须二次和MQ进行交互，将消息从待确认状态变更为确认状态后，消息才能被消费者消费。待确认状态的消息，就称之为半消息。

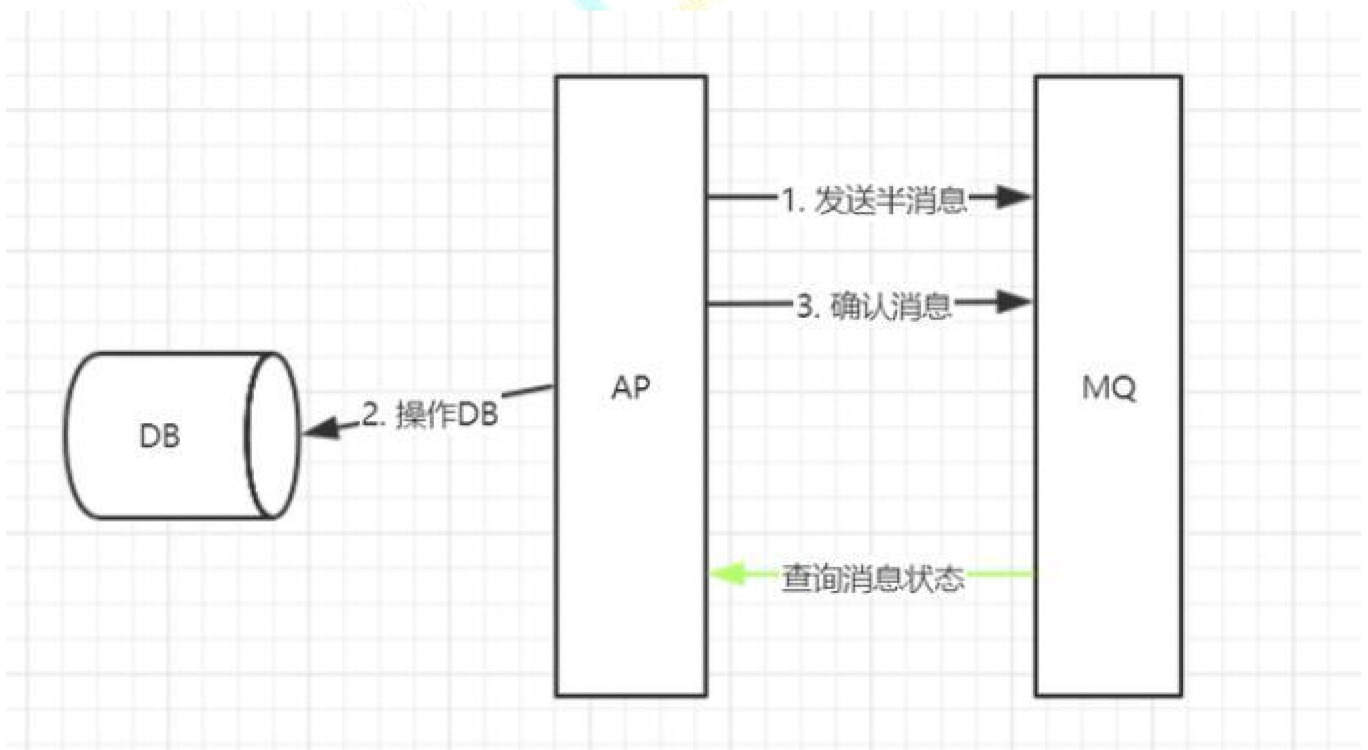
半消息的完整事务逻辑如下：

1. 向MQ发送半消息。
2. 向DB插入数据。
3. 向MQ发送确认消息。

我们发现，通过半消息的形式，将DB的操作夹在了两个MQ操作的中间。假设，第2步失败了，那么， MQ中的消息就会一直是半消息状态，也就不会被消费者消费。

那么，半消息就一直存在于MQ中吗？或者是说如果第3步失败了呢？

为了解决上面的问题， MQ引入了一个扫描的机制。即MQ会每隔一段时间，对所有的半消息进行扫描，并就扫描到的存在时间过长的半消息，向发送者进行询问，询问如果得到确认回复，则将消息改为确认状态，如得到失败回复，则将消息删除。



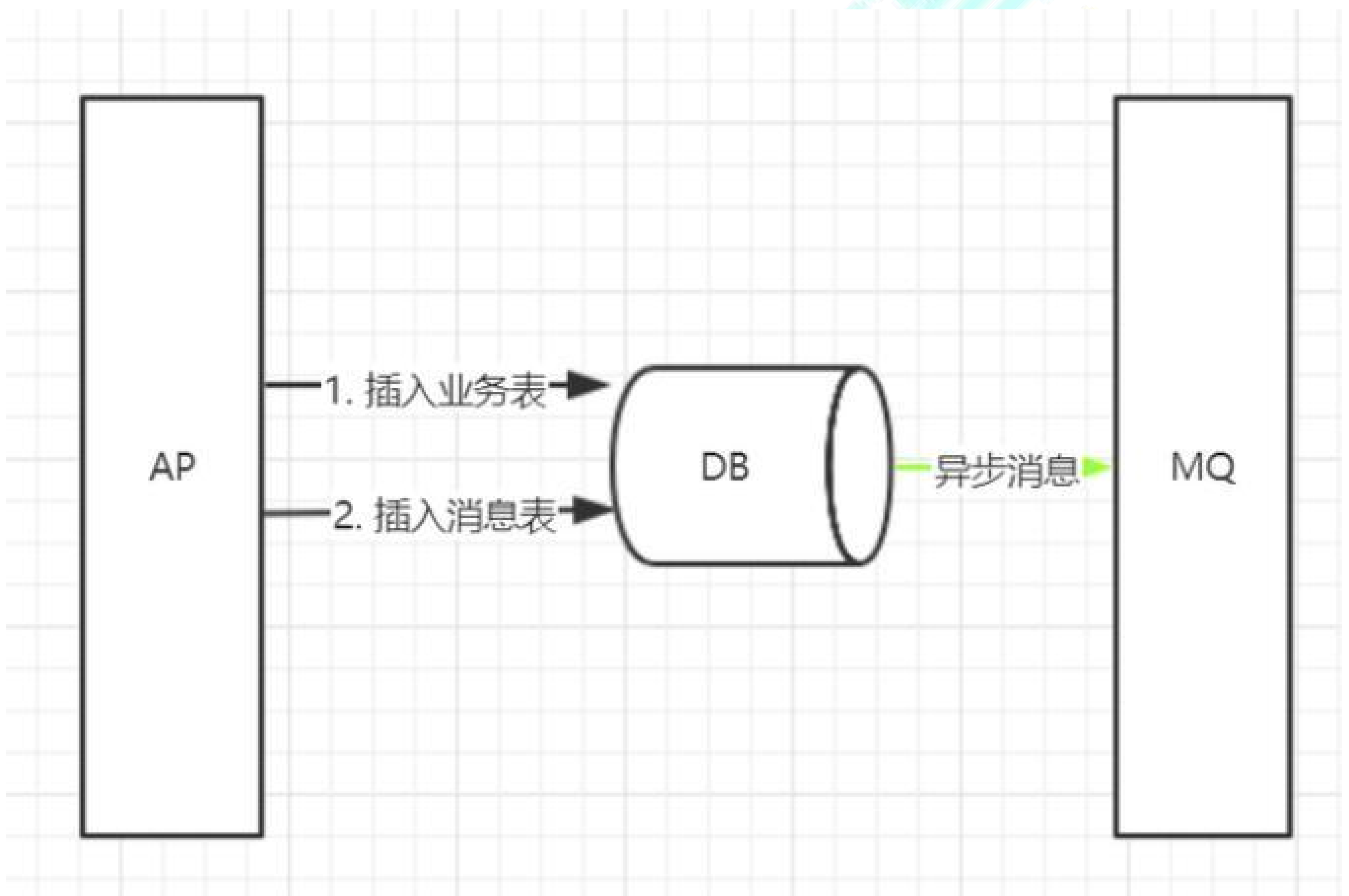
半消息

如上，半消息机制的一个问题是：要求业务方提供查询消息状态接口，对业务方依然有较大的侵入性。

方案二：本地消息表

在DB中，新增一个消息表，用于存放消息。如下：

1. 在DB业务表中插入数据。
2. 在DB消息表中插入数据。
3. 异步将消息表中的消息发送到MQ，收到ack后，删除消息表中的消息。



本地消息表

如上，通过上述逻辑，将一个分布式的事务，拆分成两大步。第1和第2，构成了一个本地的事务，从而解决了分布式事务的问题。

这种解决方案，不需要业务端提供消息查询接口，只需要稍微修改业务逻辑，侵入性是最小的。

SAGA的案例

SAGA适用于无需马上返回业务发起方最终状态的场景，例如：你的请求已提交，请稍后查询或留意通知 之类。

将上述补偿事务的场景用SAGA改写，其流程如下：

- 订单服务创建最终状态未知的订单记录，并提交事务
- 现金服务扣除所需的金额，并提交事务
- 订单服务更新订单状态为成功，并提交事务

以上为成功的流程，若现金服务扣除金额失败，那么，最后一步订单服务将会更新订单状态为失败。

其业务编码工作量比补偿事务多一点，包括以下内容：

- 订单服务创建初始订单的逻辑
- 订单服务确认订单成功的逻辑
- 订单服务确认订单失败的逻辑
- 现金服务扣除现金的逻辑
- 现金服务补偿返回现金的逻辑

但其相对于补偿事务形态有性能上的优势，所有的本地子事务执行过程中，都无需等待其调用的子事务执行，减少了加锁的时间，这在事务流程较多较长的业务中性能优势更为明显。同时，其利用队列进行进行通讯，具有削峰填谷的作用。

因此该形式适用于不需要同步返回发起方执行最终结果、可以进行补偿、对性能要求较高、不介意额外编码的业务场景。

但当然SAGA也可以进行稍微改造，变成与TCC类似、可以进行资源预留的形态

Saga和TCC对比

Saga相比TCC的缺点是缺少预留动作，导致补偿动作的实现比较麻烦： Ti就是commit，比如一个业务是发送邮件，在TCC模式下，先保存草稿（Try）再发送（Confirm），撤销的话直接删除草稿（Cancel）就行了。而Saga则就直接发送邮件了（Ti），如果要撤销则得再发送一份邮件说明撤销（Ci），实现起来有一些麻烦。

如果把上面的发邮件的例子换成： A服务在完成Ti后立即发送Event到ESB（企业服务总线，可以认为是一个消息中间件），下游服务监听到这个Event做自己的一些工作然后再发送Event到ESB，如果A服务执行补偿动作Ci，那么整个补偿动作的层级就很深。

不过没有预留动作也可以认为是优点：

- 有些业务很简单，套用TCC需要修改原来的业务逻辑，而Saga只需要添加一个补偿动作就行了。
- TCC最少通信次数为2n，而Saga为n（n=sub-transaction的数量）。
- 有些第三方服务没有Try接口， TCC模式实现起来就比较tricky了，而Saga则很简单。
- 没有预留动作就意味着不必担心资源释放的问题，异常处理起来也更简单。

Saga对比TCC

Saga和TCC都是补偿型事务，他们的区别为：

劣势：

- 无法保证隔离性；

优势：

- 一阶段提交本地事务，无锁，高性能；
- 事件驱动模式，参与者可异步执行，高吞吐；
- Saga 对业务侵入较小，只需要提供一个逆向操作的Cancel即可；而TCC需要对业务进行全局性的流程改造；

总体的方案对比：

属性	2PC	TCC	Saga	异步确保型事务	尽最大努力通知
事务一致性	强	弱	弱	弱	弱
复杂性	中	高	中	低	低
业务侵入性	小	大	小	中	中
使用局限性	大	大	中	小	中
性能	低	中	高	高	高
维护成本	低	高	中	低	中

seata

> 以下内容，来自 官网

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。

seata简介

Seata，官网，github，1万多星

- Seata 是一款开源的分布式事务解决方案，致力于在微服务架构下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
- 在 Seata 开源之前， Seata 对应的内部版本在阿里经济体内部一直扮演着分布式一致性中间件的角色，帮助经济体平稳的度过历年的双11，对各BU业务进行了有力的支撑。商业化产品GTS 先后在阿里云、金融云进行售卖

相关链接：

什么是seata： https://seata.io/zh-cn/docs/overview/what-is-seata.html

下载 https://seata.io/zh-cn/blog/download.html
官方例子 https://seata.io/zh-cn/docs/user/quickstart.html

Seata 的三大模块

Seata AT使用了增强型二阶段提交实现。

Seata 分三大模块：

- TC ：事务协调者。负责我们的事务ID的生成，事务注册、提交、回滚等。
- TM：事务发起者。定义事务的边界，负责告知 TC，分布式事务的开始，提交，回滚。
- RM：资源管理者。管理每个分支事务的资源，每一个 RM 都会作为一个分支事务注册在 TC。

在Seata的AT模式中， TM和RM都作为SDK的一部分和业务服务在一起，我们可以认为是Client。TC是□ 个□□ 的服务，通过服务的注册、发现将
自己暴露给Client们。

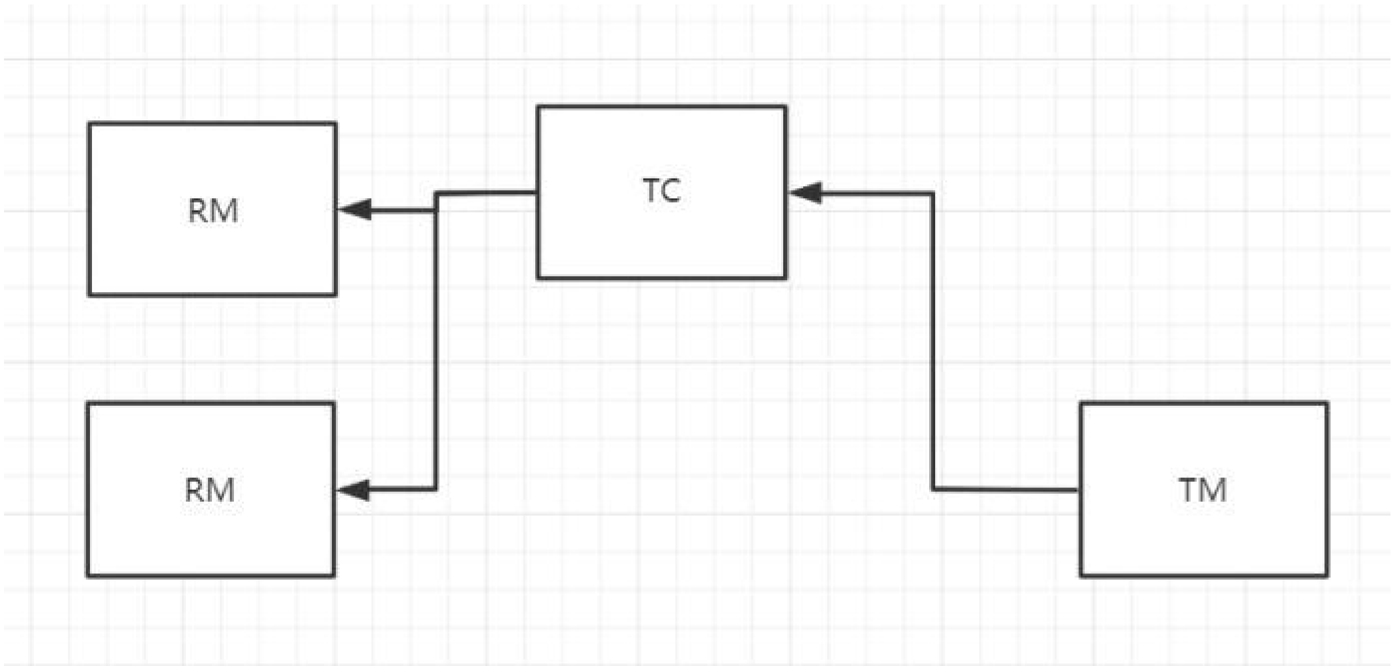
Seata 中有三大模块中， TM 和 RM 是作为 Seata 的客户端与业务系统集成在一起， TC 作为 Seata 的服务端□□ 部署。

Seata 的分布式事务的执行流程

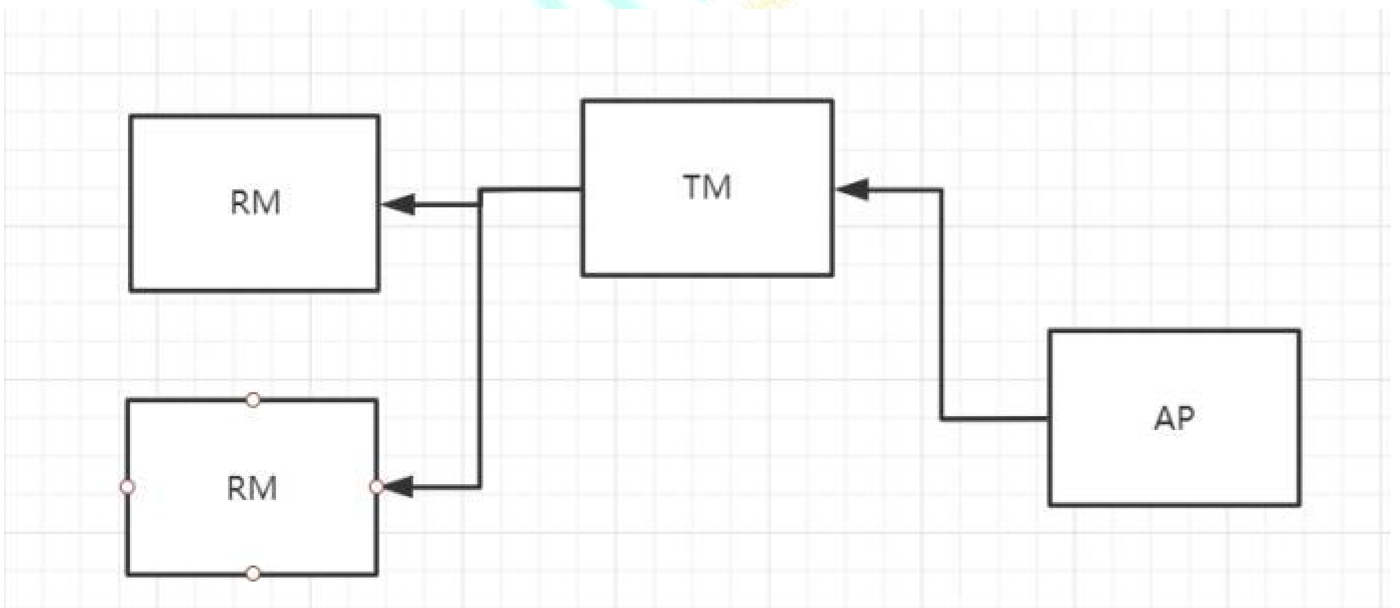
在 Seata 中，分布式事务的执行流程：

- TM 开启分布式事务（TM 向 TC 注册全局事务记录）；
- 按业务场景，编排数据库、服务等事务内资源（RM 向 TC 汇报资源准备状态 ）；
- TM 结束分布式事务，事务一阶段结束（TM 通知 TC 提交/回滚分布式事务）；
- TC 汇总事务信息，决定分布式事务是提交还是回滚；

- TC 通知所有 RM 提交/回滚 资源，事务二阶段结束；



Seata的TC、TM、RM三个角色， 是不是和XA模型很像. 下图是XA模型的事务大致流程。

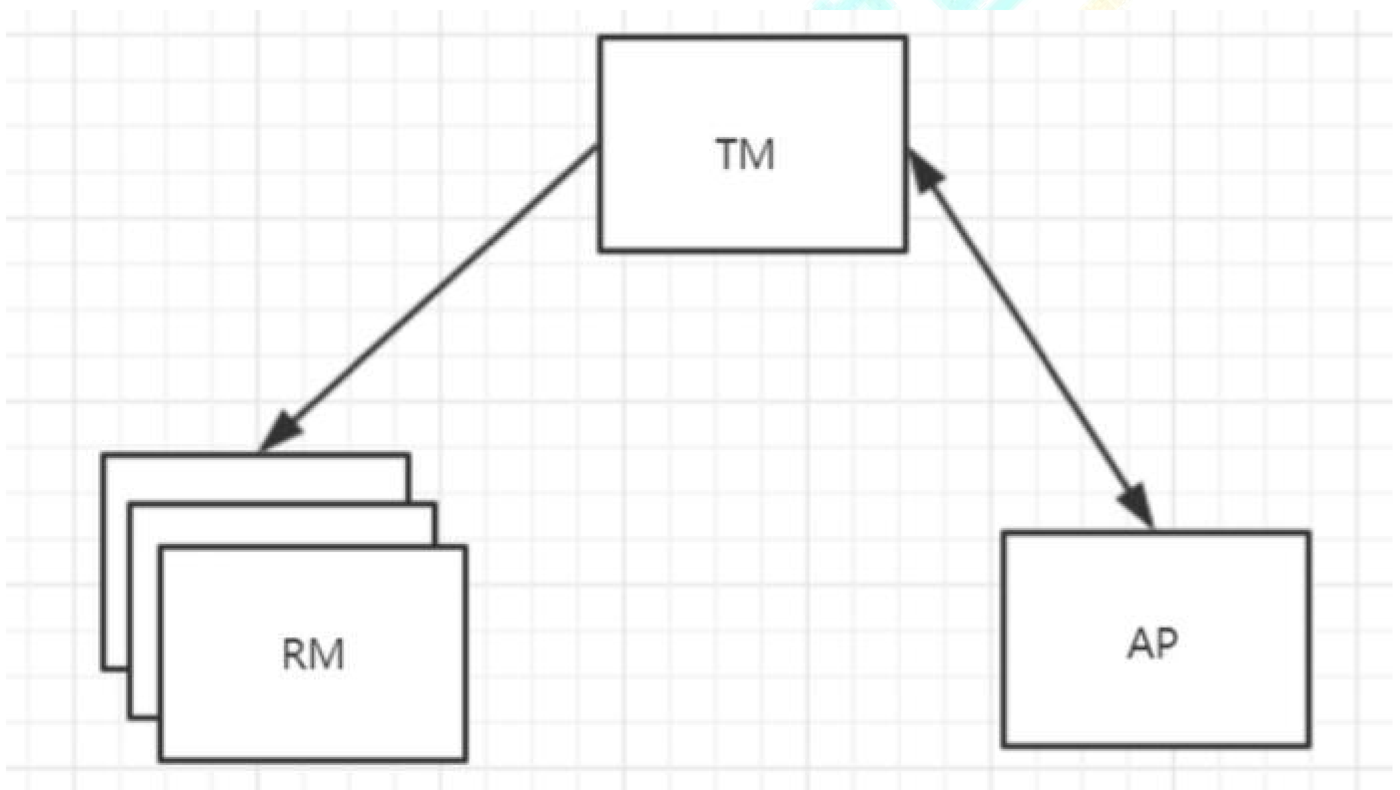


在X/Open **DTP(Distributed Transaction Process)**模型里面，有三个角色：

AP: Application，应用程序。也就是业务层。哪些操作属于一个事务，就是AP定义的。

TM: Transaction Manager，事务管理器。接收AP的事务请求，对全局事务进行管理，管理事务分支状态，协调RM的处理，通知RM哪些操作属于哪些全局事务以及事务分支等等。这个也是整个事务调度模型的核心部分。

RM： Resource Manager，资源管理器。一般是数据库，也可以是其他的资源管理器，如消息队列(如JMS数据源)，文件系统等。



4 种分布式事务解决方案

Seata 会有 4 种分布式事务解决方案，分别是 AT 模式、 TCC 模式、 Saga 模式和 XA 模式。

Seata AT模式

Seata AT模式是最早支持的模式。 AT模式是指**Automatic (Branch) Transaction Mode**自动化分支事务。

Seata AT 模式是增强型2pc模式，或者说是增强型的XA模型。

总体来说， AT 模式，是 2pc两阶段提交协议的演变，不同的地方， Seata AT 模式不会一直锁表。

Seata AT模式的使用前提

- 基于支持本地 ACID 事务的关系型数据库。

> 比如，在MySQL 5.1之前的版本中，默认的搜索引擎是MyISAM，从MySQL 5.5之后的版本中，默认的搜索引擎变更为InnoDB。
>
> MyISAM存储引擎的特点是：表级锁、不支持事务和全文索引。所以，基于MyISAM 的表，就不支持Seata AT模式。

- Java 应用，通过 JDBC 访问数据库。

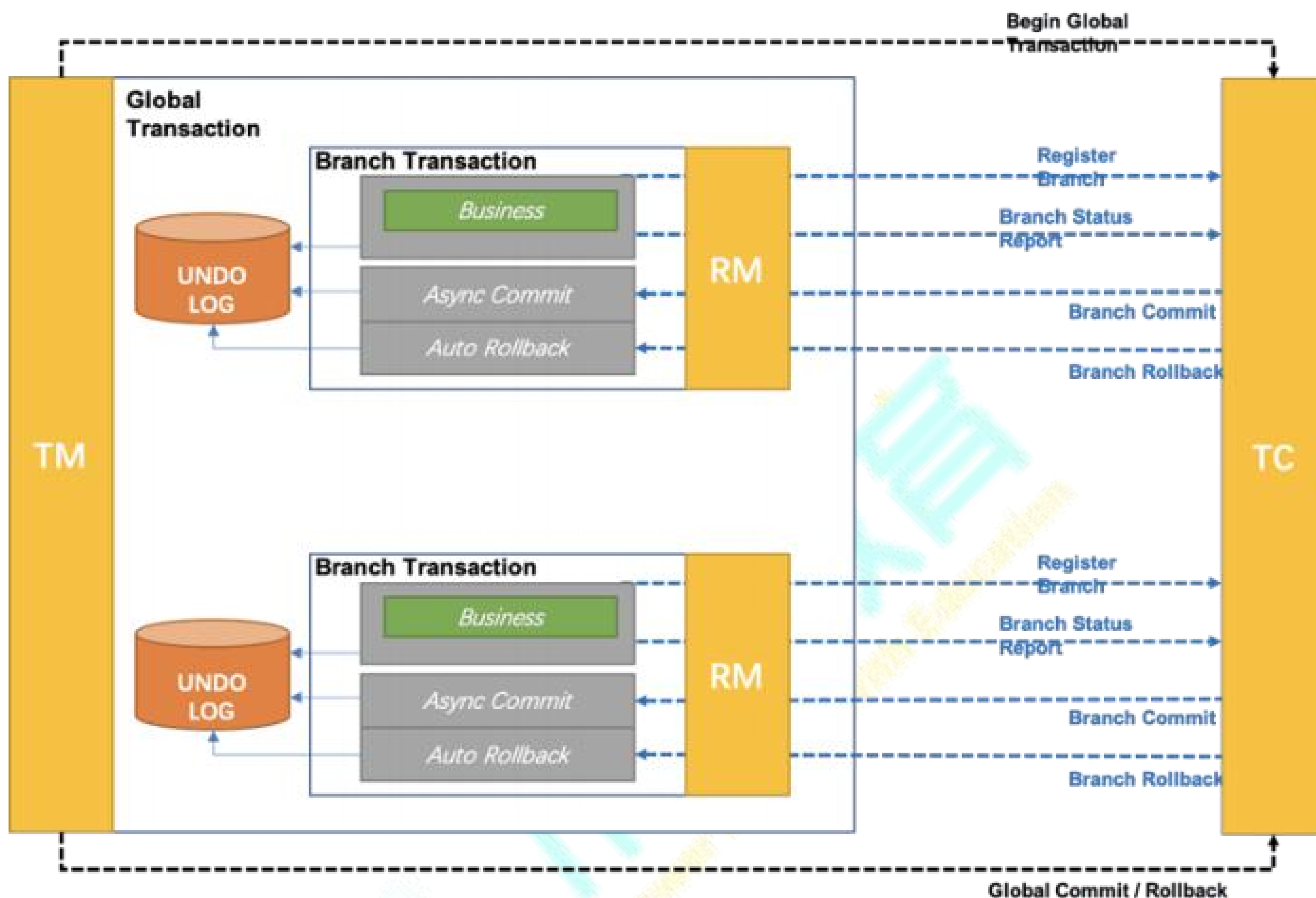
Seata AT模型图

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。

- 或回滚通过一阶段的回滚日志进行反向补偿

完整的AT在Seata所制定的事务模式下的模型图：



Seata AT模式的例子

我们用一个比较简单的业务场景来描述一下Seata AT模式的工作过程。

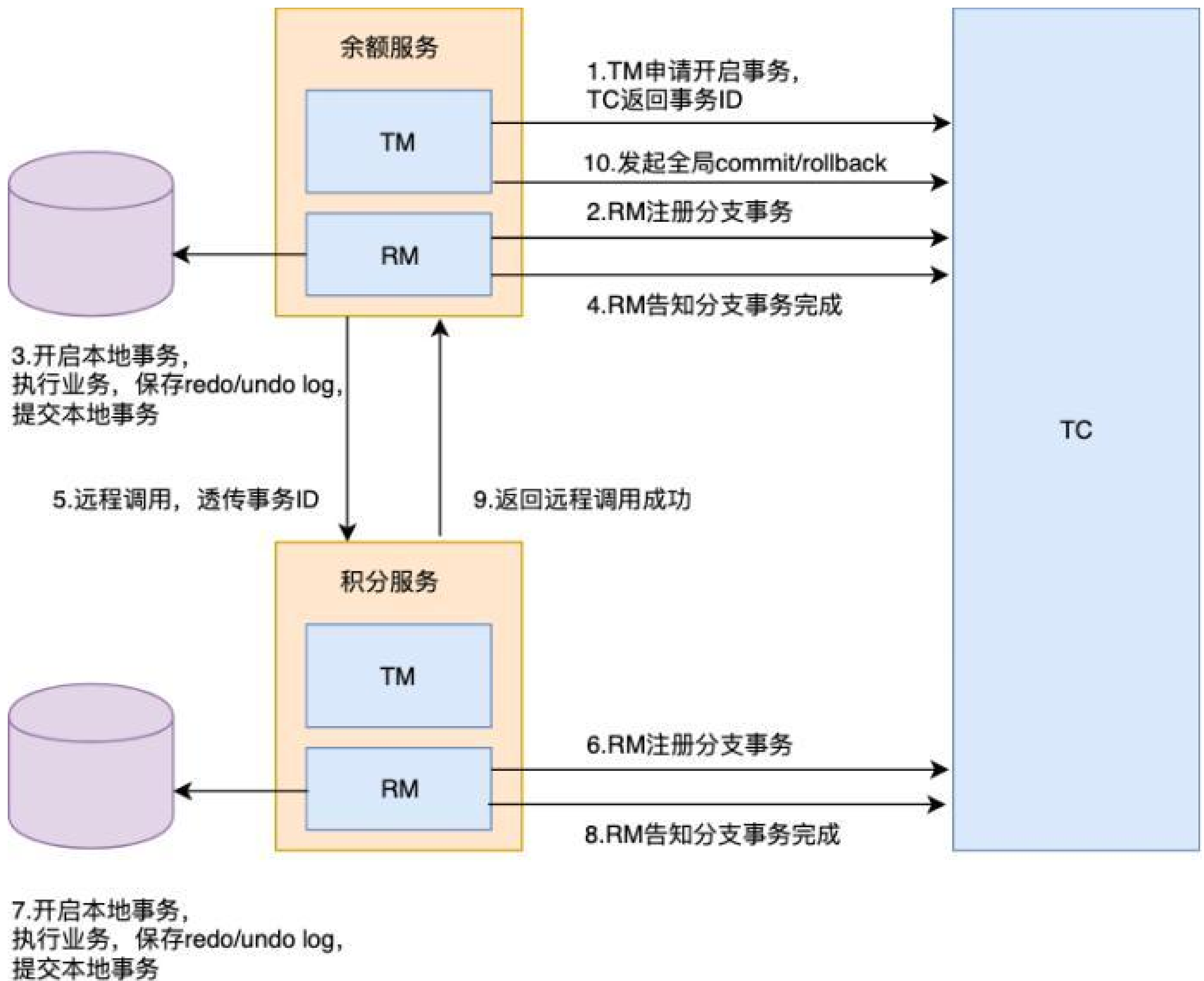
有个充值业务，现在有两个服务，一个负责管理用户的余额，另外一个负责管理用户的积分。

> 当用户充值的时候，首先增加用户账户上的余额，然后增加用户的积分。

Seata AT分为两阶段，主要逻辑全部在第一阶段，第二阶段主要做回滚或日志清理的工作。

第一阶段流程：

第一阶段流程如



1）余额服务中的TM，向TC申请开启一个全局事务， TC会返回一个全局的事务ID。

2）余额服务在执行本地业务之前， RM会先向TC注册分支事务。

3）余额服务依次生成undo log、执行本地事务、生成redo log，最后直接提交本地事务。

4）余额服务的RM向TC汇报，事务状态是成功的。

5）余额服务发起远程调用，把事务ID传给积分服务。

6）积分服务在执行本地业务之前，也会先向TC注册分支事务。

7）积分服务次生成undo log、执行本地事务、生成redo log，最后直接提交本地事务。

8）积分服务的RM向TC汇报，事务状态是成功的。

9）积分服务返回远程调用成功给余额服务。

10）余额服务的TM向TC申请全局事务的提交/回滚。

> 积分服务中也有TM，但是由于没有用到，因此直接可以忽略。

我们如果使用 Spring框架的注解式事务，远程调用会在本地事务提交之前发生。但是，先发起远程调用还是先提交本地事务，这个其实没有任何影响。

第二阶段流程如：

第二阶段的逻辑就比较简单了。

Client和TC之间是有长连接的，如果是正常全局提交，则TC通知多个RM异步清理掉本地的redo和undo log即可。如果是回滚，则TC通知每个RM回滚数据即可。

这里就会引出一个问题，由于本地事务都是自己直接提交了，后面如何回滚，由于我们在操作本地业务操作的前后，做记录了undo和redo log，因此可以通过undo log进行回滚。

由于undo和redo log和业务操作在同一个事务中，因此肯定会同时成功或同时失败。

但是还会存在一个问题，因为每个事务从本地提交到通知回滚这段时间里，可能这条数据已经被别的事务修改，如果直接用undo log回滚，会导致数据不一致的情况。

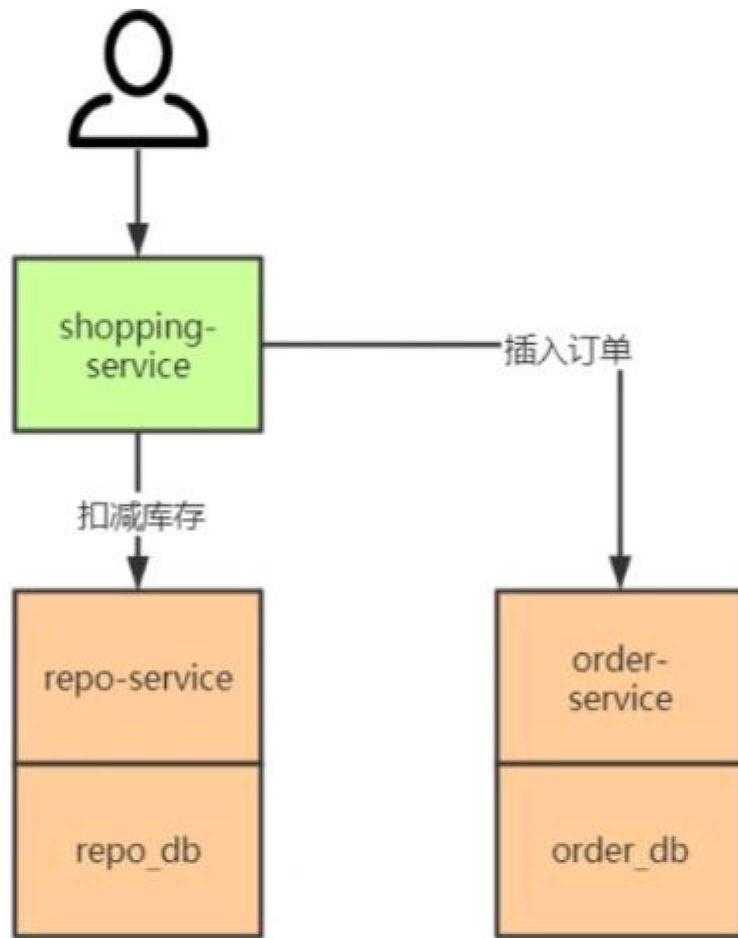
此时， RM会用redo log进行校验，对比数据是否一样，从而得知数据是否有别的事务修改过。注意： undo log是被修改前的数据，可以用于回滚；redo log是被修改后的数据，用于回滚校验。

如果数据未被其他事务修改过，则可以直接回滚；如果是脏数据，再根据不同策略处理。

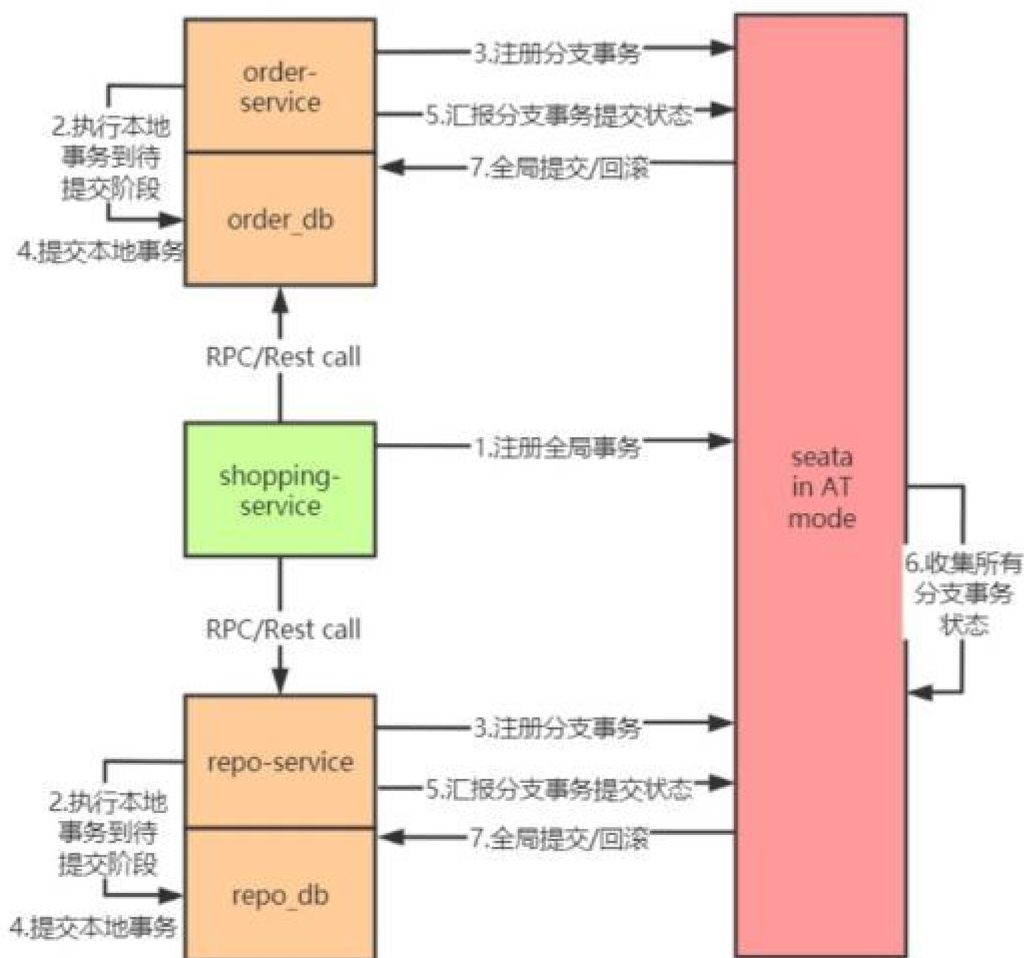
Seata AT 模式在电商下单场景的使用

下面描述 Seata AT mode 的工作原理使用的电商下单场景的使用

如下图所示：



在上图中，协调者 shopping-service 先调用参与者 repo-service 扣减库存，后调用参与者 order-service 生成订单。这个业务流使用 Seata in XA mode 后的全局事务流程如下图所示：



上图描述的全局事务执行流程为：

1） shopping-service 向 Seata 注册全局事务，并产生一个全局事务标识 XID

2）将 repo-service.repo_db、order-service.order_db 的本地事务执行到待提交阶段，事务内容包含对 repo-service.repo_db、order-service.order_db 进行的查询操作以及写每个库的 undo_log 记录

3） repo-service.repo_db、order-service.order_db 向 Seata 注册分支事务，并将其纳入该 XID 对应的全局事务范围

4）提交 repo-service.repo_db、order-service.order_db 的本地事务

5）repo-service.repo_db、order-service.order_db 向 Seata 汇报分支事务的提交状态

6）Seata 汇总所有的 DB 的分支事务的提交状态，决定全局事务是该提交还是回滚

7）Seata 通知 repo-service.repo_db、order-service.order_db 提交/回滚本地事务，若需要回滚，采取的是补偿式方法

其中 1）2）3）4）5）属于第一阶段， 6）7）属于第二阶段。

电商业务场景中 Seata in AT mode 工作流程详述

在上面的电商业务场景中，购物服务调用库存服务扣减库存，调用订单服务创建订单，显然这两个调用过程要放在一个事务里面。即：

```
start global_trx

 call 库存服务的扣减库存接口

 call 订单服务的创建订单接口

commit global_trx
```

在库存服务的数据库中，存在如下的库存表 t_repo：

id	production_code	name	count	price
10001	20001	xx 键盘	98	200.0
10002	20002	yy 鼠标	199	100.0

在订单服务的数据库中，存在如下的订单表 t_order：

id	order_code	user_id	production_code	count	price
30001	2020102500001	40001	20002	1	100.0
30002	2020102500001	40001	20001	2	400.0

现在， id 为 40002 的用户要购买一只商品代码为 20002 的鼠标，整个分布式事务的内容为：

1）在库存服务的库存表中将记录

id	production_code	name	count	price
10002	20002	yy 鼠标	199	100.0

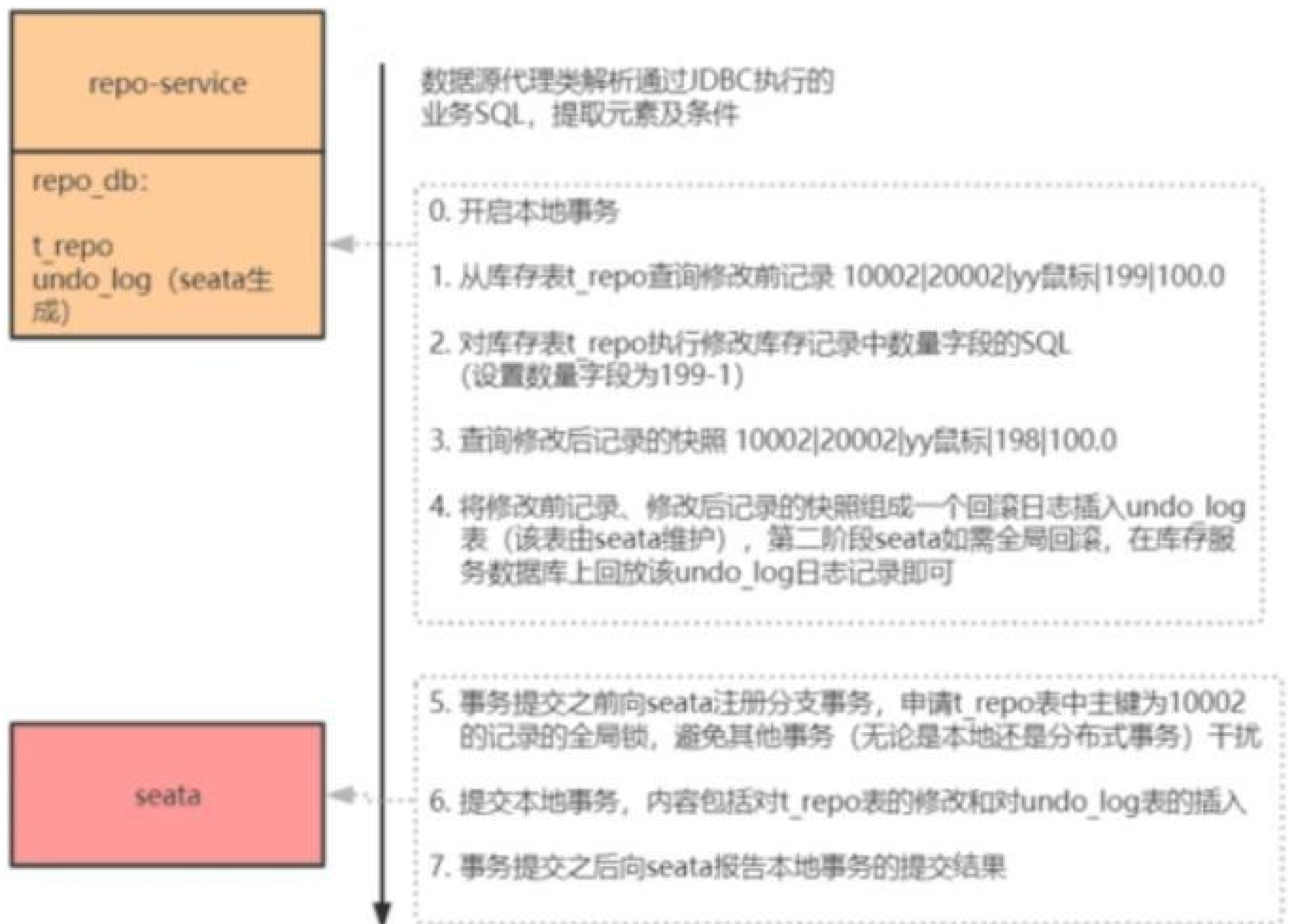
修改为

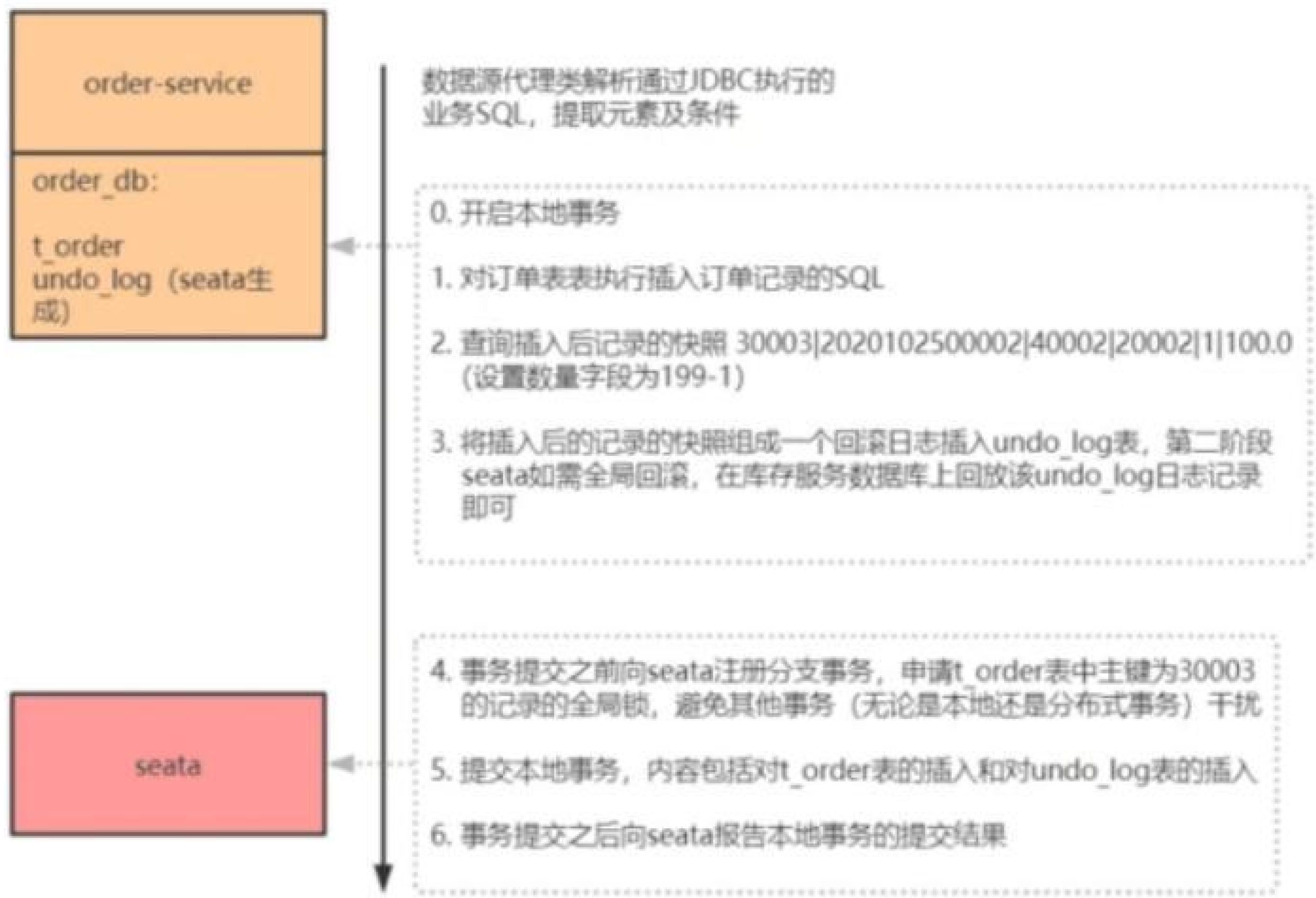
id	production_code	name	count	price
10002	20002	yy 鼠标	198	100.0

2）在订单服务的订单表中添加一条记录

id	order_code	user_id	production_code	count	price
30003	2020102500002	40002	20002	1	100.0

以上操作，在 AT 模式的第一阶段的流程图如下：





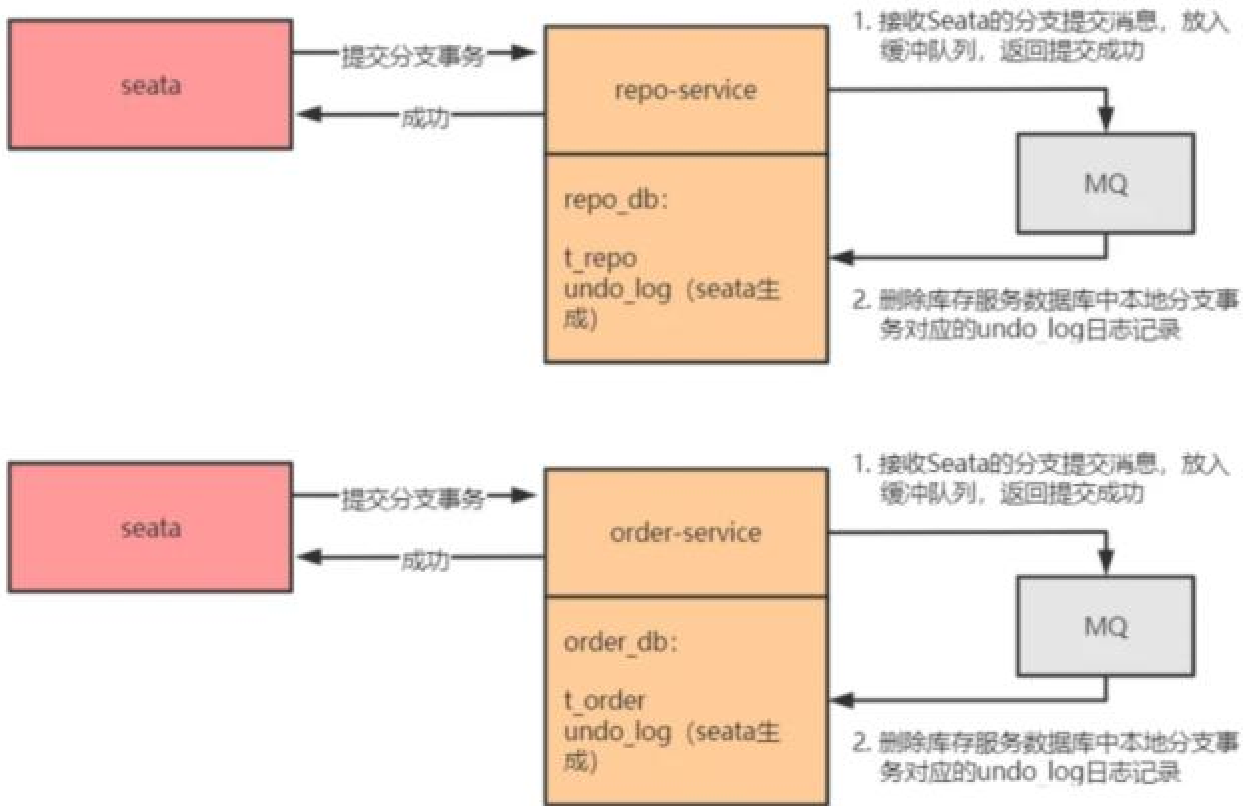
从 AT 模式第一阶段的流程来看，分支的本地事务在第一阶段提交完成之后，就会释放掉本地事务锁定的本地记录。这是 AT 模式和 XA 最大的不同点，在 XA 事务的两阶段提交中，被锁定的记录直到第二阶段结束才会被释放。所以 AT 模式减少了锁记录的时间，从而提高了分布式事务的处理效率。 AT 模式之所以能够实现第一阶段完成就释放被锁定的记录，是因为 Seata 在每个服务的数据库中维护了一张 undo_log 表，其中记录了对 t_order / t_repo 进行操作前后记录的镜像数据，即便第二阶段发生异常，只需回放每个服务的 undo_log 中的相应记录即可实现全局回滚。

undo_log 的表结构：

id	branch_id	xid	context	rollback_info	log_status	log_created
......	分支事务 ID	全局事务 ID	......	分支事务操作的记录在事务前后的记录镜像，即 beforeImage 和 afterImage	......	......

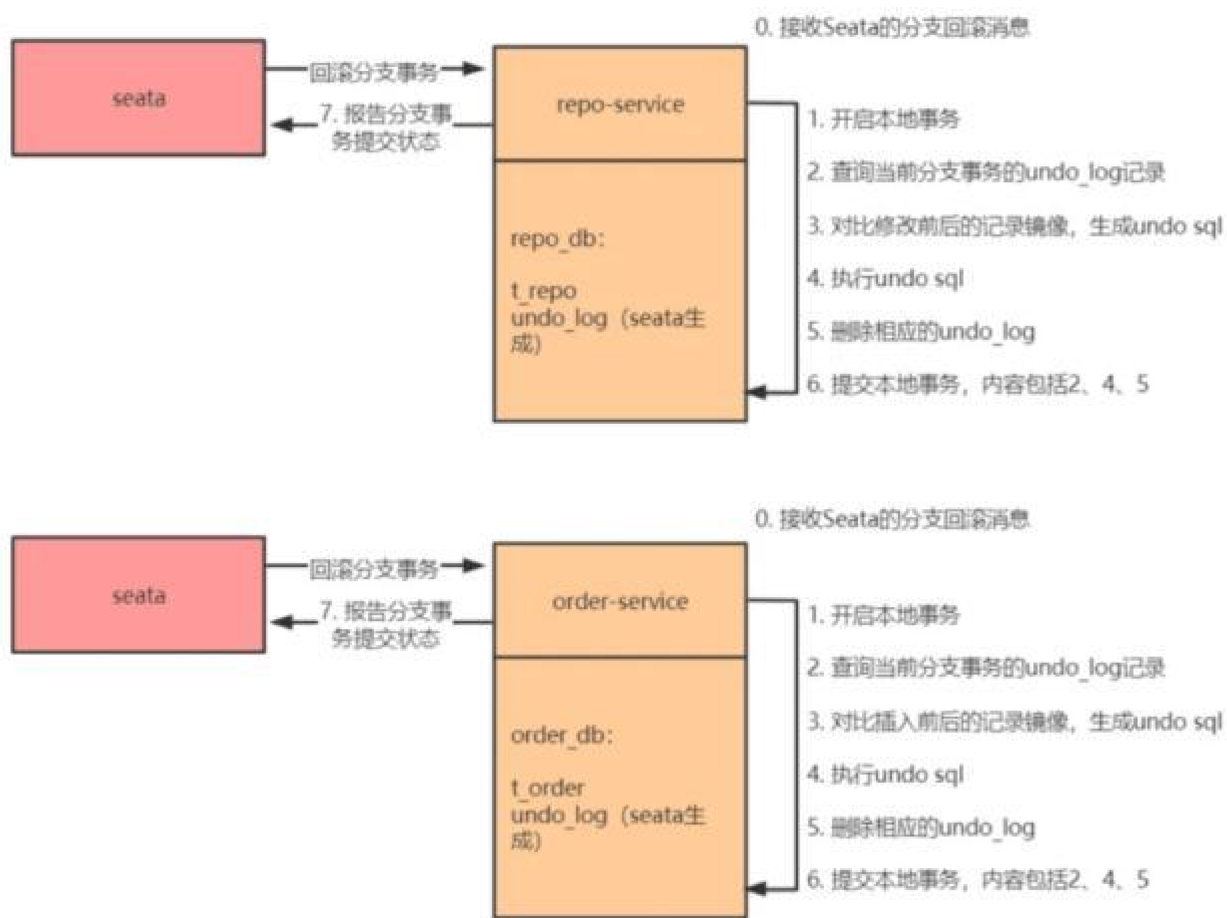
第一阶段结束之后， Seata 会接收到所有分支事务的提交状态，然后决定是提交全局事务还是回滚全局事务。

1）若所有分支事务本地提交均成功，则 Seata 决定全局提交。 Seata 将分支提交的消息发送给各个分支事务，各个分支事务收到分支提交消息后，会将消息放入一个缓冲队列，然后直接向 Seata 返回提交成功。之后，每个本地事务会慢慢处理分支提交消息，处理的方式为：删除相应分支事务的 undo_log 记录。之所以只需删除分支事务的 undo_log 记录，而不需要再做其他提交操作，是因为提交操作已经在第一阶段完成了（这也是 AT 和 XA 不同的地方）。这个过程如下图所示：



分支事务之所以能够直接返回成功给 Seata，是因为真正关键的提交操作在第一阶段已经完成了，清除 undo_log 日志只是收尾工作，即便清除失败了，也对整个分布式事务不产生实质影响。

2）若任一分支事务本地提交失败，则 Seata 决定全局回滚，将分支事务回滚消息发送给各个分支事务，由于在第一阶段各个服务的数据库上记录了 undo_log 记录，分支事务回滚操作只需根据 undo_log 记录进行补偿即可。全局事务的回滚流程如下图所示：



这里对图中的 2、3 步做进一步的说明：

1）由于上文给出了 undo_log 的表结构，所以可以通过 xid 和 branch_id 来找到当前分支事务的所有 undo_log 记录；

2）拿到当前分支事务的 undo_log 记录之后，首先要做数据校验，如果 afterImage 中的记录与当前的表记录不一致，说明从第一阶段完成到此刻期间，有别的事务修改了这些记录，这会导致分支事务无法回滚，向 Seata 反馈回滚失败；如果 afterImage 中的记录与当前的表记录一致，说明从第一阶段完成到此刻期间，没有别的事务修改这些记录，分支事务可回滚，进而根据 beforeImage 和 afterImage 计算出补偿 SQL，执行补偿 SQL 进行回滚，然后删除相应 undo_log，向 Seata 反馈回滚成功。

Seata的数据隔离性

seata的at模式主要实现逻辑是数据源代理，而数据源代理将基于如MySQL和Oracle等关系事务型数据库实现，基于数据库的隔离级别为read committed。换而言之，本地事务的支持是seata实现at模式的必要条件，这也将限制seata的at模式的使用场景。

写隔离

从前面的工作流程，我们可以很容易知道， Seata的写隔离级别是全局□
占
的。

首先，我们理解一下写隔离的流程

```
分支事务1-开始
 |
V 获取 本地锁
 |
V 获取 全局锁        分支事务2-开始
 |                   |
V 释放 本地锁        V 获取 本地锁
 |                   |
V 释放 全局锁        V 获取 全局锁
                     |
                    V 释放 本地锁
                     |
                    V 释放 全局锁
```

如上所示，一个分布式事务的锁获取流程是这样的

1）先获取到本地锁，这样你已经可以修改本地数据了，只是还不能本地事务提交

2）而后，能否提交就是看能否获得全局锁

3）获得了全局锁，意味着可以修改了，那么提交本地事务，释放本地锁

4）当分布式事务提交，释放全局锁。这样就可以让其它事务获取全局锁，并提交它们对本地数据的修改了。

可以看到，这里有两个关键点

1）本地锁获取之前，不会去争抢全局锁

2）全局锁获取之前，不会提交本地锁

这就意味着，数据的修改将被互斥开来。也就不会造成写入脏数据。全局锁可以让分布式修改中的写数据隔离。

写隔离的原则:

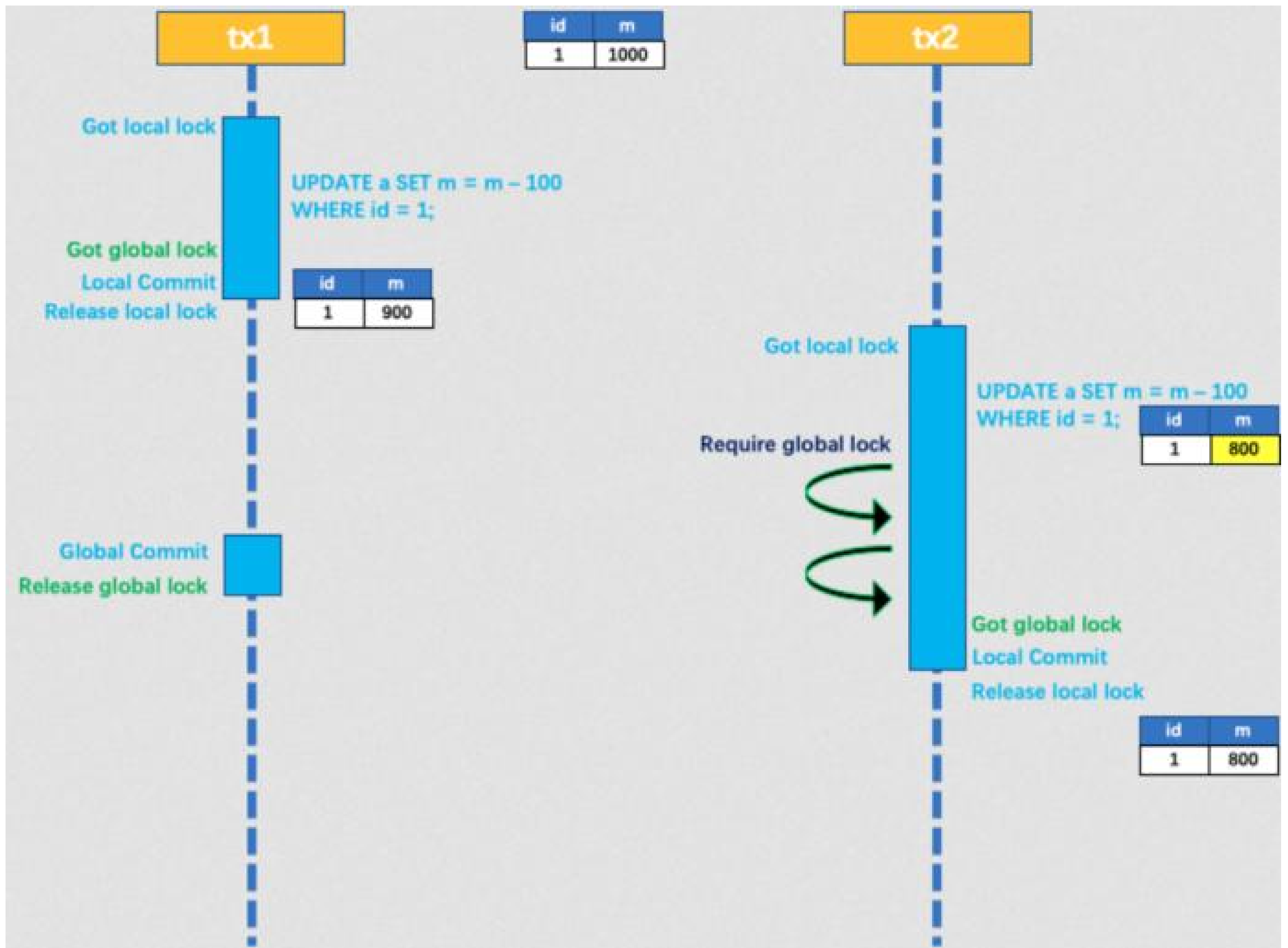
- 一阶段本地事务提交前，需要确保先拿到 **全局锁** 。
  拿不到 **全局锁** ，不能提交本地事务。
- 拿 **全局锁** 的尝试被限制在一定范围内，超出范围将放弃，并回滚本地事务，释放本地锁。

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。

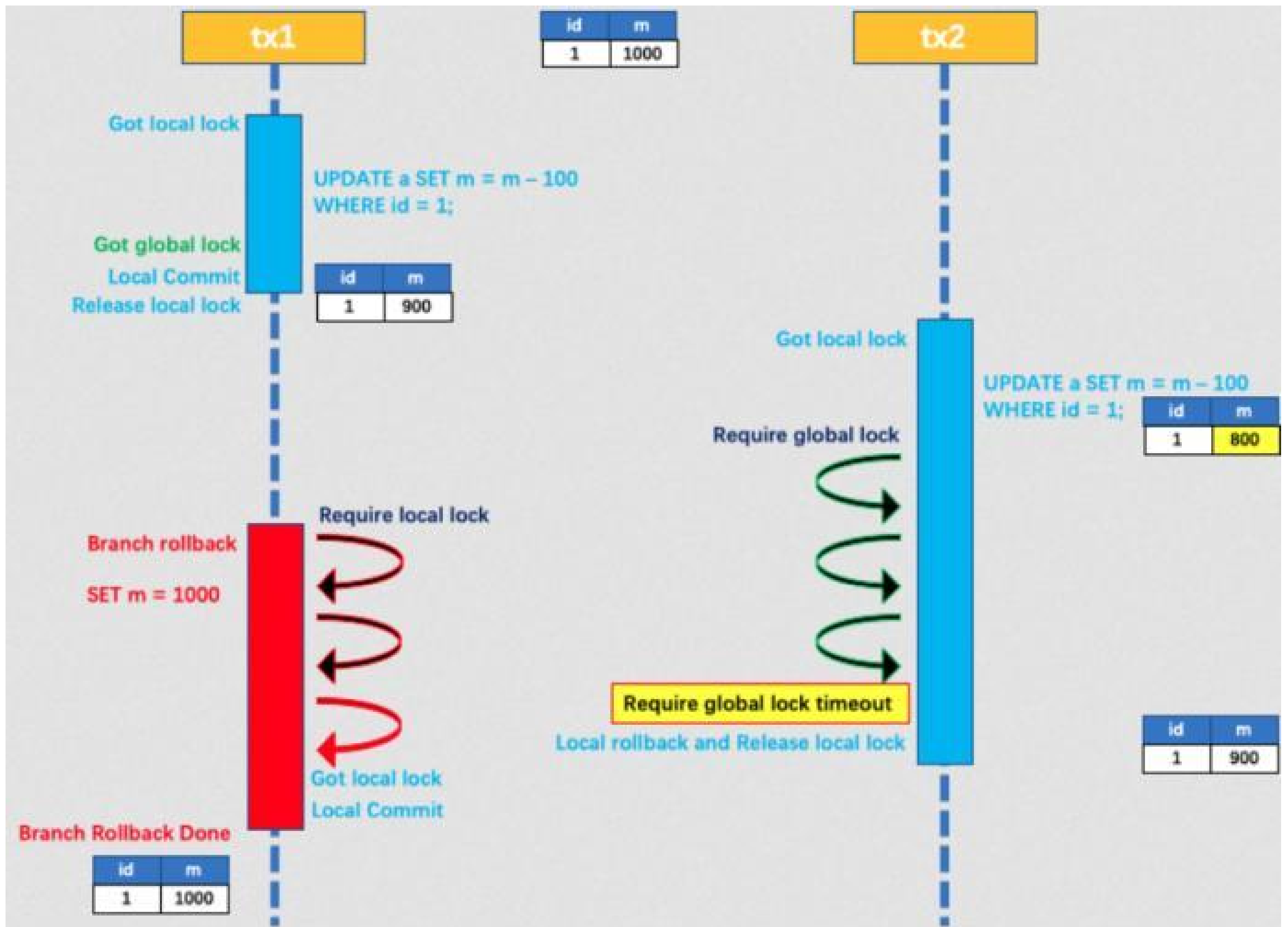
tx1 先开始，开启本地事务，拿到本地锁，更新操作 m = 1000 - 100 = 900。本地事务提交前，先拿到该记录的 **全局锁** ，本地提交释放本地锁。

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。



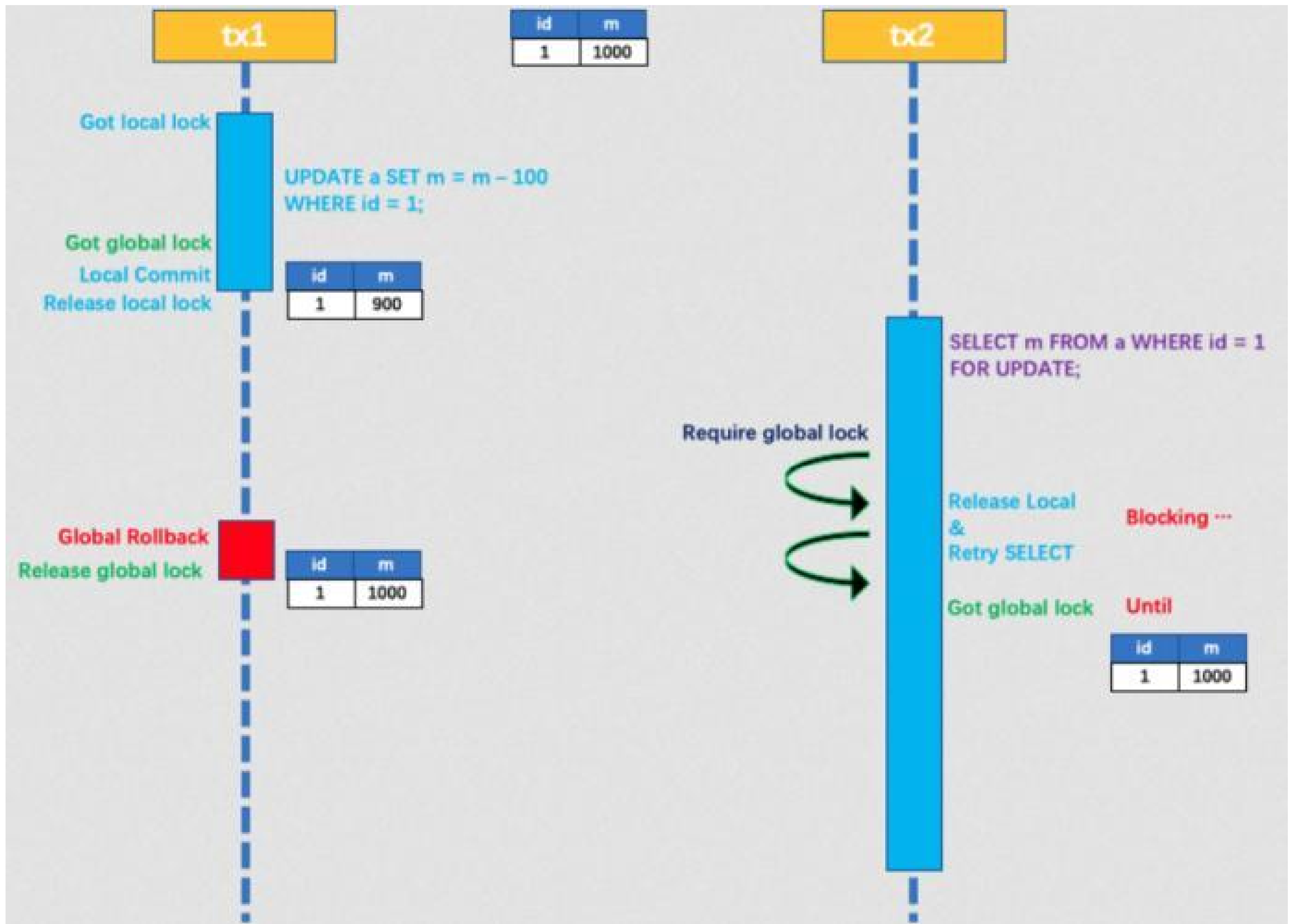
此时，如果 tx2 仍在等待该数据的 **全局锁**，同时持有本地锁，则 tx1 的分支回滚会失败。分支的回滚会一直重试，直到 tx2 的 **全局锁** 等锁超时，放弃 **全局锁** 并回滚本地事务释放本地锁， tx1 的分支回滚最终成功。

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。

读的隔离级别

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。

Spring Cloud集成Seata AT模式

AT模式是指**Automatic (Branch) Transaction Mode 自动化分支事务，** 使用AT模式的前提是

- 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。

seata-at的使用步骤

1、引入seata框架，配置好seata基本配置，建立undo_log表

2、消费者引入全局事务注解@GlobalTransactional

3、生产者引入全局事务注解@GlobalTransactional

> 此处没有写完， 尼恩的博文，都是迭代模式，后续会持续

Seata TCC 模式

简介

TCC 与 Seata AT 事务一样都是**两阶段事务**，它与 AT 事务的主要区别为：

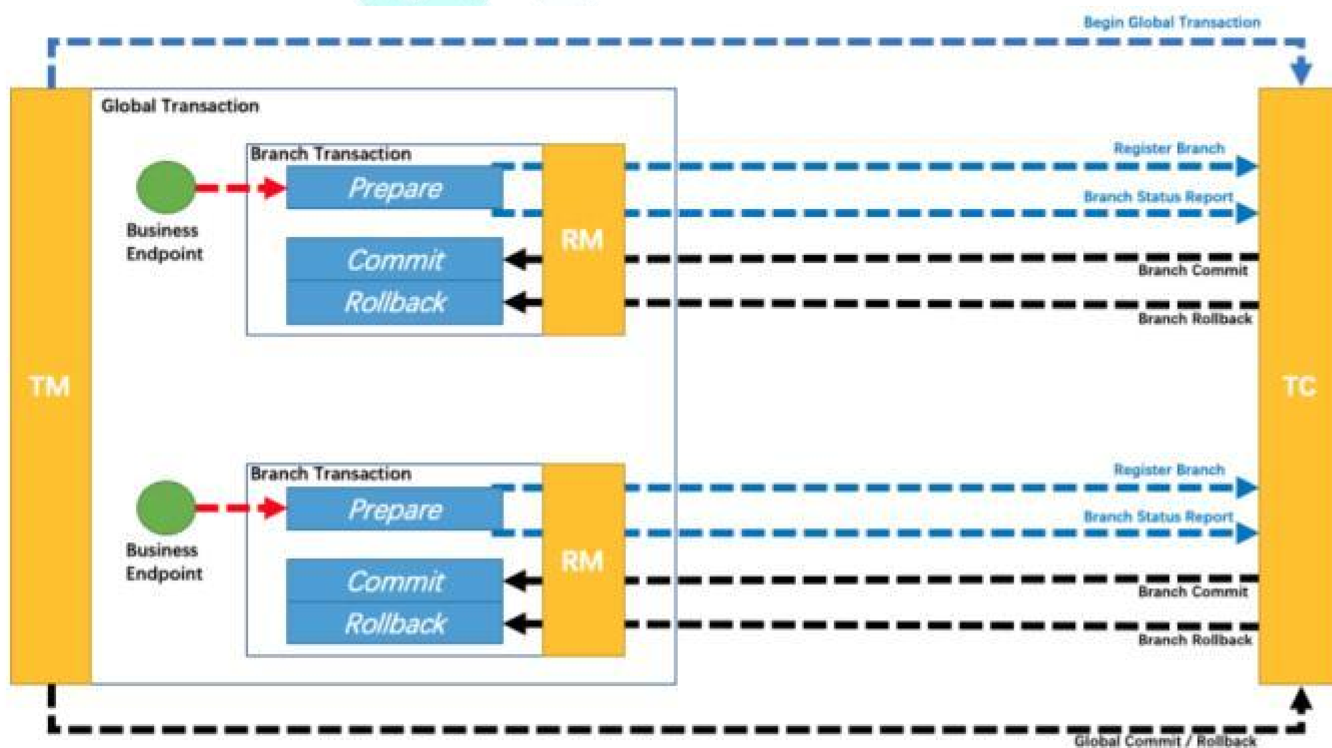
- **TCC 对业务代码侵入严重**

  每个阶段的数据操作都要自己进行编码来实现，事务框架无法自动处理。

- **TCC 性能更高**
  不必对数据加**全局锁**，允许多个事务同时操作数据。

Seata TCC 整体是 **两阶段提交** 的模型。一个分布式的全局事务，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（参考链接 TBD）基于 **支持本地 ACID 事务** 的 **关系型数据库**：

- 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。
- 二阶段 commit 行为：马上成功结束， **自动** 异步批量清理回滚日志。
- 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：

- 一阶段 prepare 行为：调用 **自定义** 的 prepare 逻辑。
- 二阶段 commit 行为：调用 **自定义** 的 commit 逻辑。
- 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。

第一阶段 Try

以账户服务为例，当下订单时要扣减用户账户金额：

下订单时，
扣减账户金额

账户表

ID	账户	已消费	可用金额	冻结
...	...	...	...	...
2	账户A	0	1000	0
...	...	...	...	...

假如用户购买 100 元商品，要扣减 100 元。

TCC 事务首先对这100元的扣减金额进行预留，或者说是先冻结这100元：

冻结要扣减的金额100

TCC事务：Try

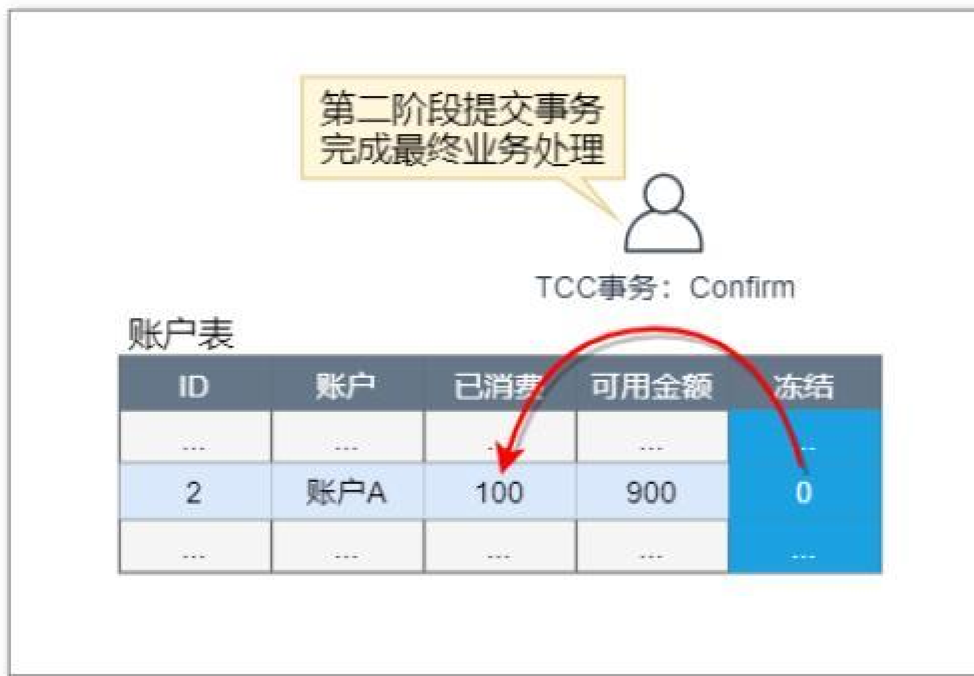
账户表

ID	账户	已消费	可用金额	冻结
...	...	...	...	...
2	账户A	0	900	100
...	...	...	...	...

第二阶段 Confirm

如果第一阶段能够顺利完成，那么说明“扣减金额”业务(分支事务)最终肯定是可以成功的。当全局事务提交时， TC会控制当前分支事务进行提交，如果提交失败， TC 会反复尝试，直到提交成功为止。

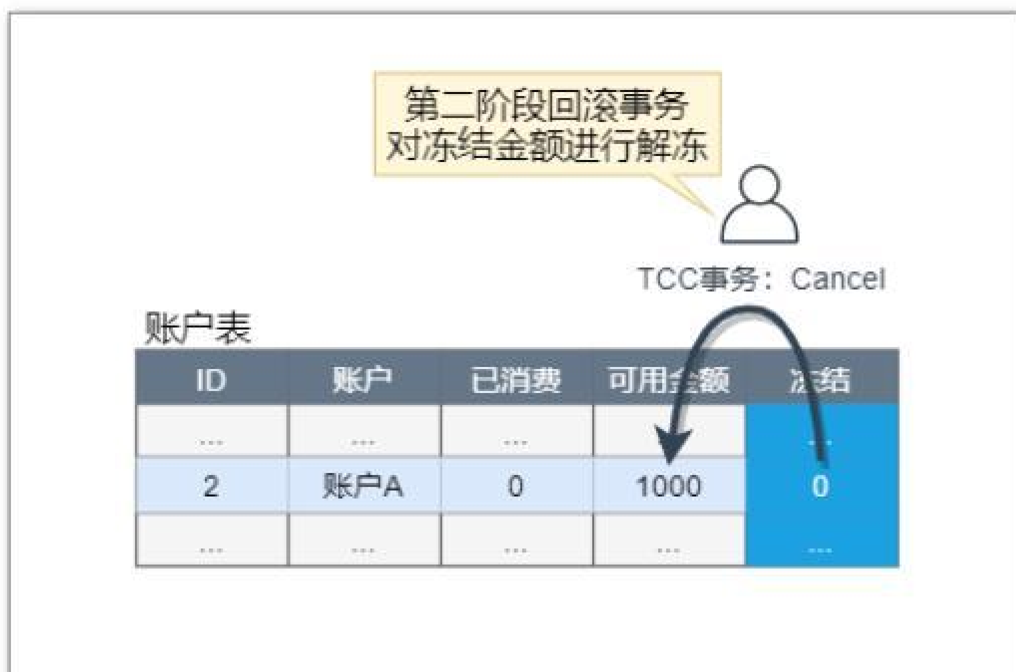
当全局事务提交时，就可以使用冻结的金额来最终实现业务数据操作：



ID	账户	已消费	可用金额	冻结
...	...	...	...	...
2	账户A	100	900	0
...	...	...	...	...

第二阶段 Cancel

如果全局事务回滚，就把冻结的金额进行解冻，恢复到以前的状态， TC 会控制当前分支事务回滚，如果回滚失败， TC 会反复尝试，直到回滚完成为止。



ID	账户	已消费	可用金额	冻结
...	...	...	...	...
2	账户A	0	1000	0
...	...	...	...	...

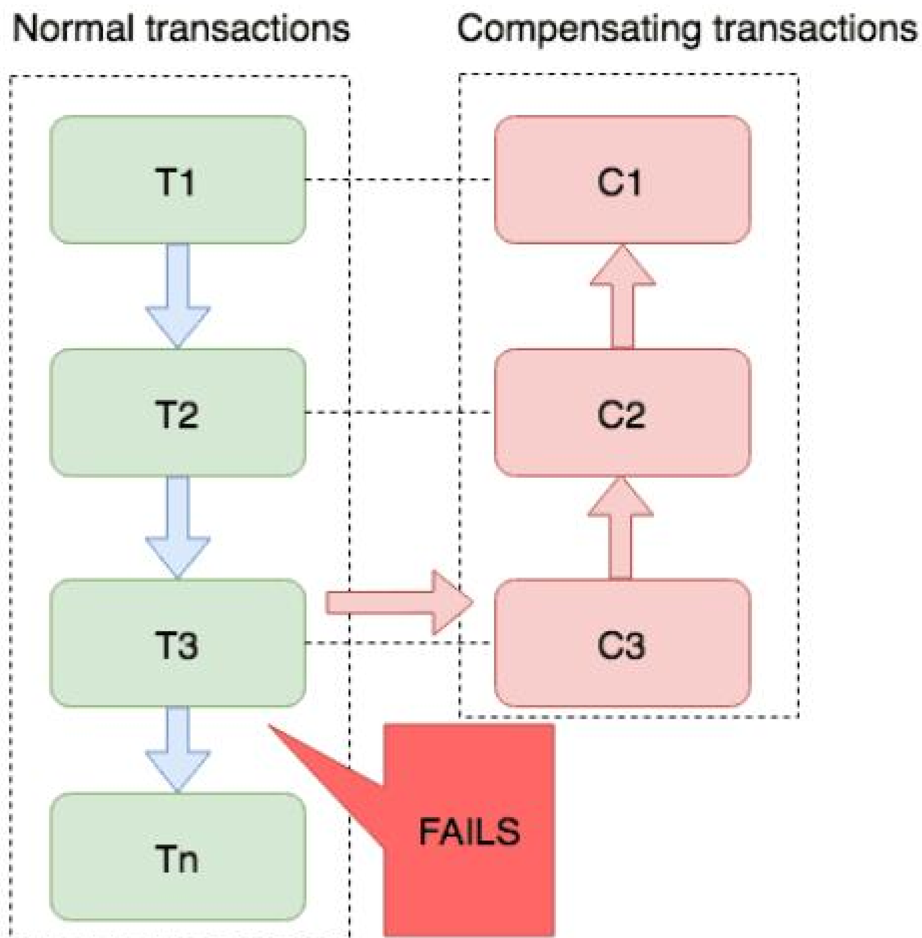
多个事务并发的情况

多个TCC全局事务允许并发，它们执行扣减金额时，只需要冻结各自的金额即可：

> Seata TCC 模式 没有写完，　尼恩的博文，都是迭代模式，后续会持续优化

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。



理论基础： Hector & Kenneth 发表论文 Sagas （1987）

适用场景：

- 业务流程长、业务流程多
- 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：

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

缺点:

不保证隔离性（应对方案见后面文档）

Saga的实现:

基于状态机引擎的 Saga 实现:

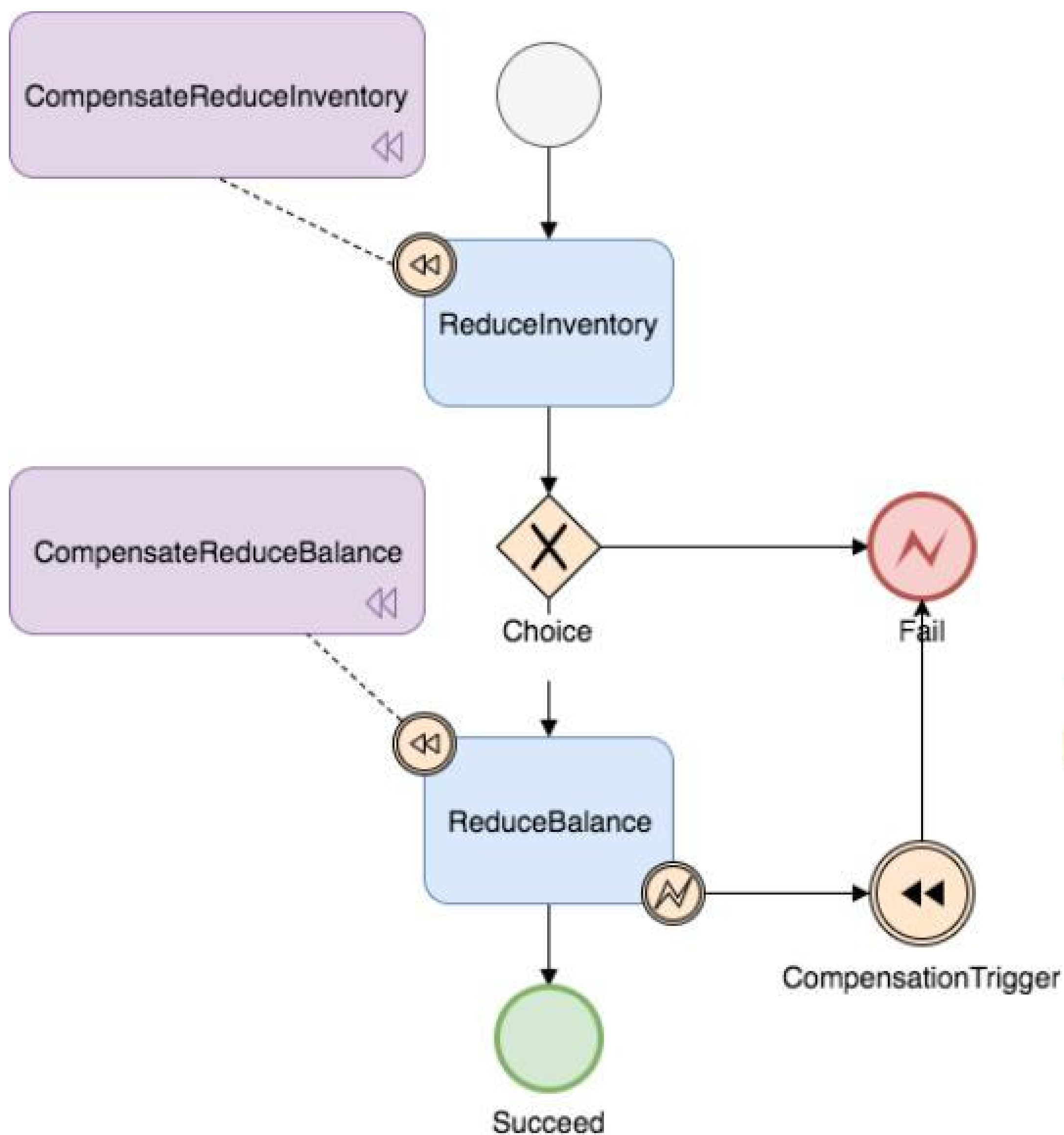
目前SEATA提供的Saga模式是基于状态机引擎来实现的，机制是:

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

> 注意: 异常发生时是否进行补偿也可由用户自定义决定

1. 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:



> Seata Saga 模式 没有写完，　尼恩的博文，都是迭代模式，后续会持续优化

Seata XA 模式

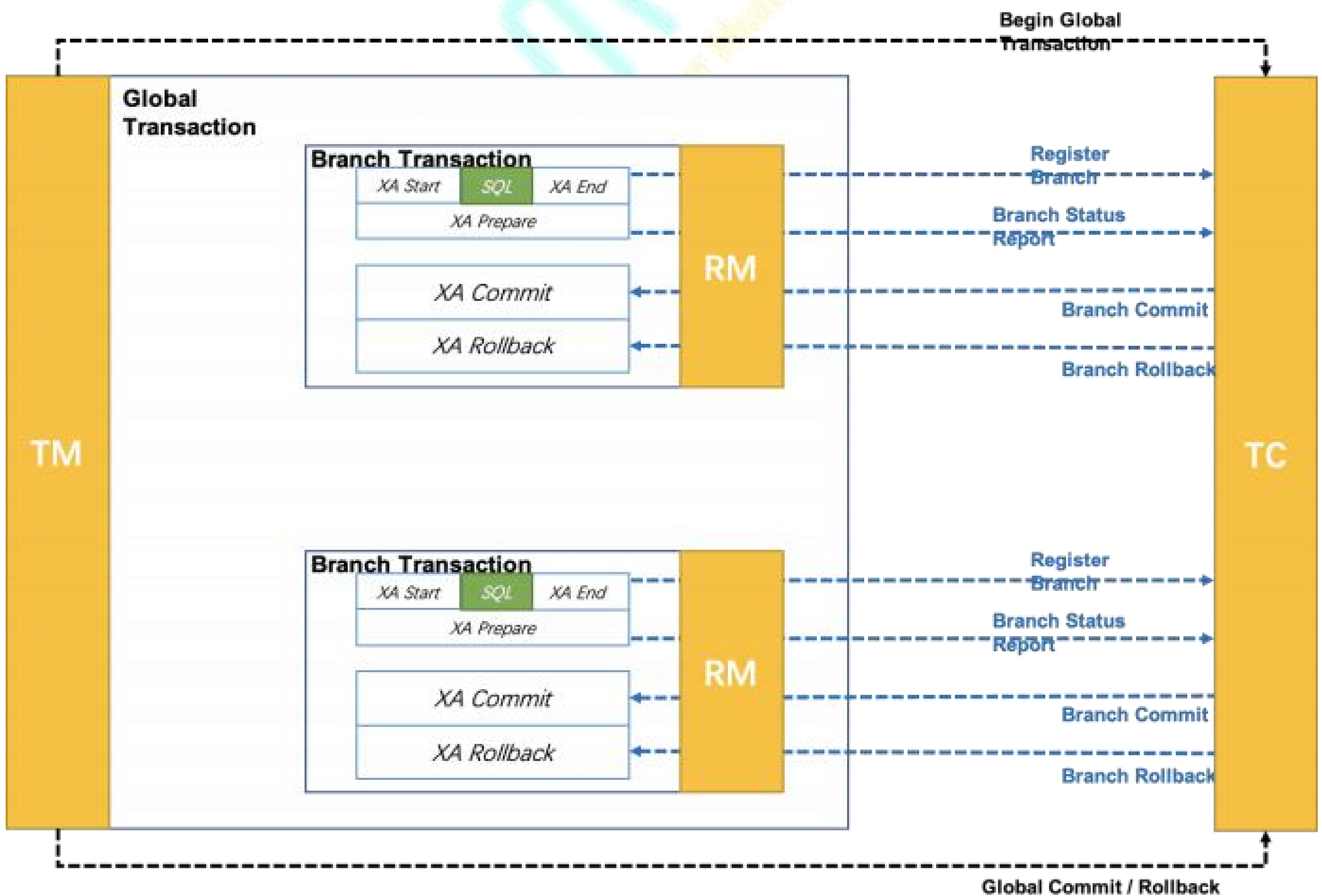
使用Seata XA 模式的前提

支持XA 事务的数据库。
Java 应用，通过 JDBC 访问数据库。

Seata XA 模式的整体机制

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。

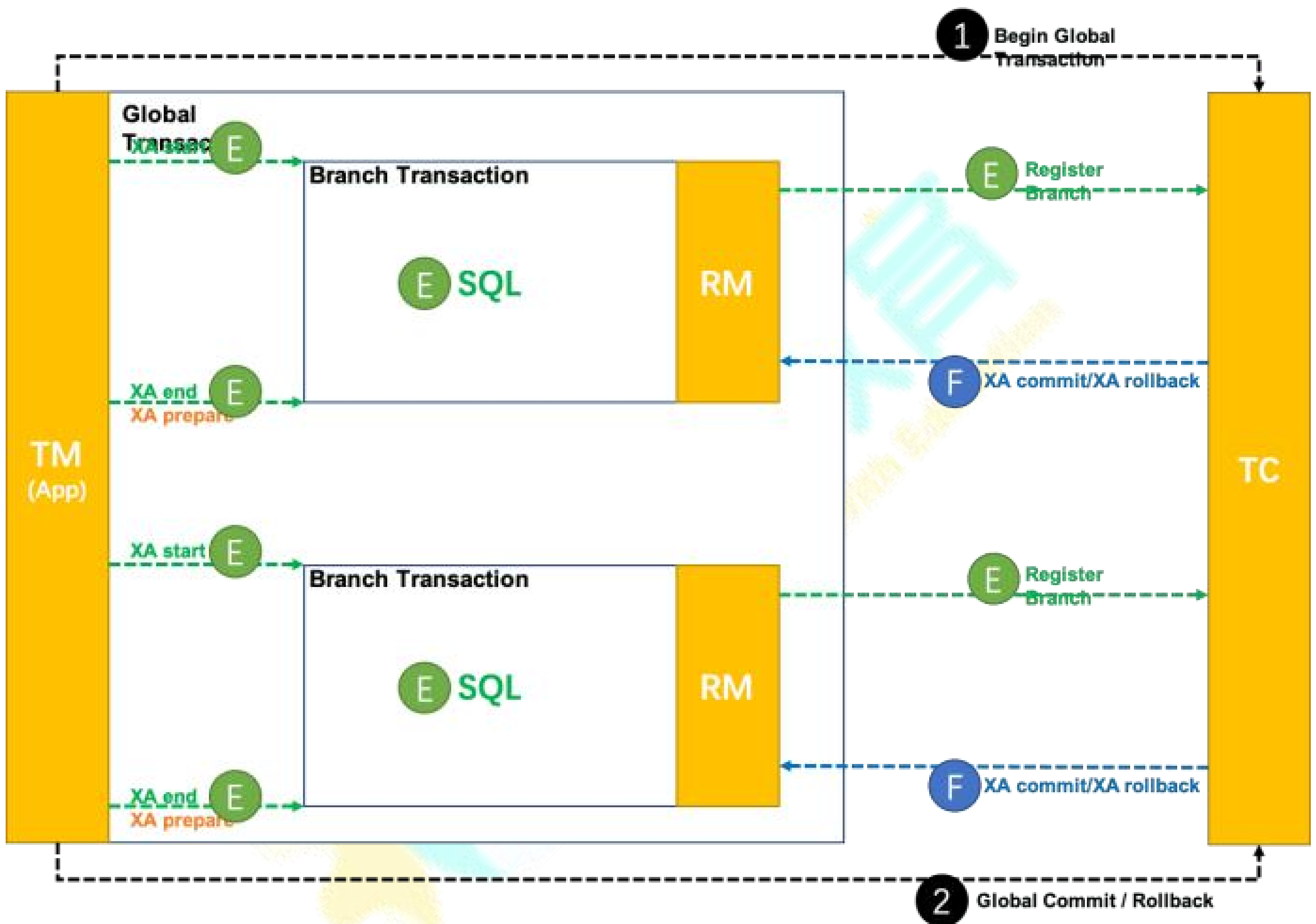
> 注意这里的重点：利用事务资源对 XA 协议的支持，以 XA 协议的机制来管理分支事务。



Seata XA 模式的工作机制

1. 整体运行机制

XA 模式 运行在 Seata 定义的事务框架内：



2. 数据源代理

XA 模式需要 XAConnection。

获取 XAConnection 两种方式：

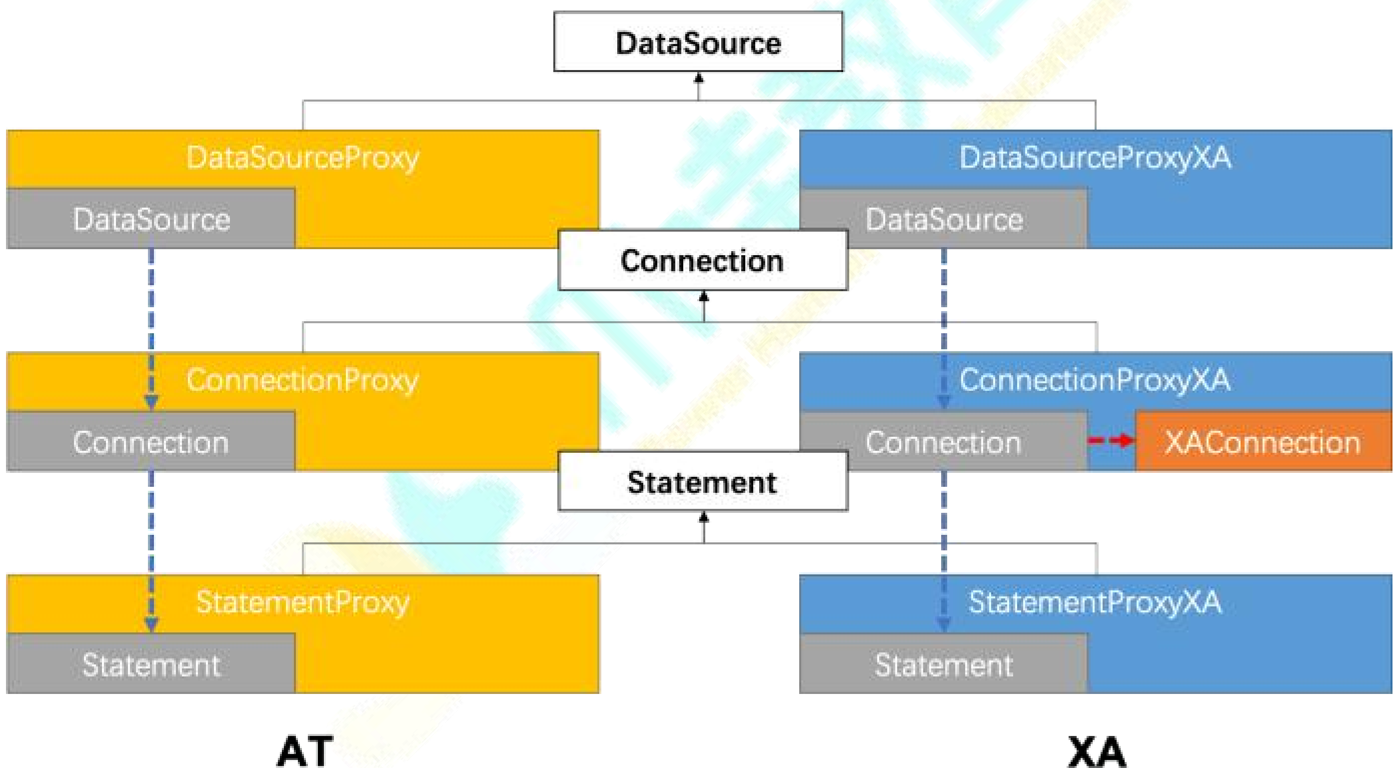
方式一：要求开发者配置 XADataSource
方式二：根据开发者的普通 DataSource 来创建

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：

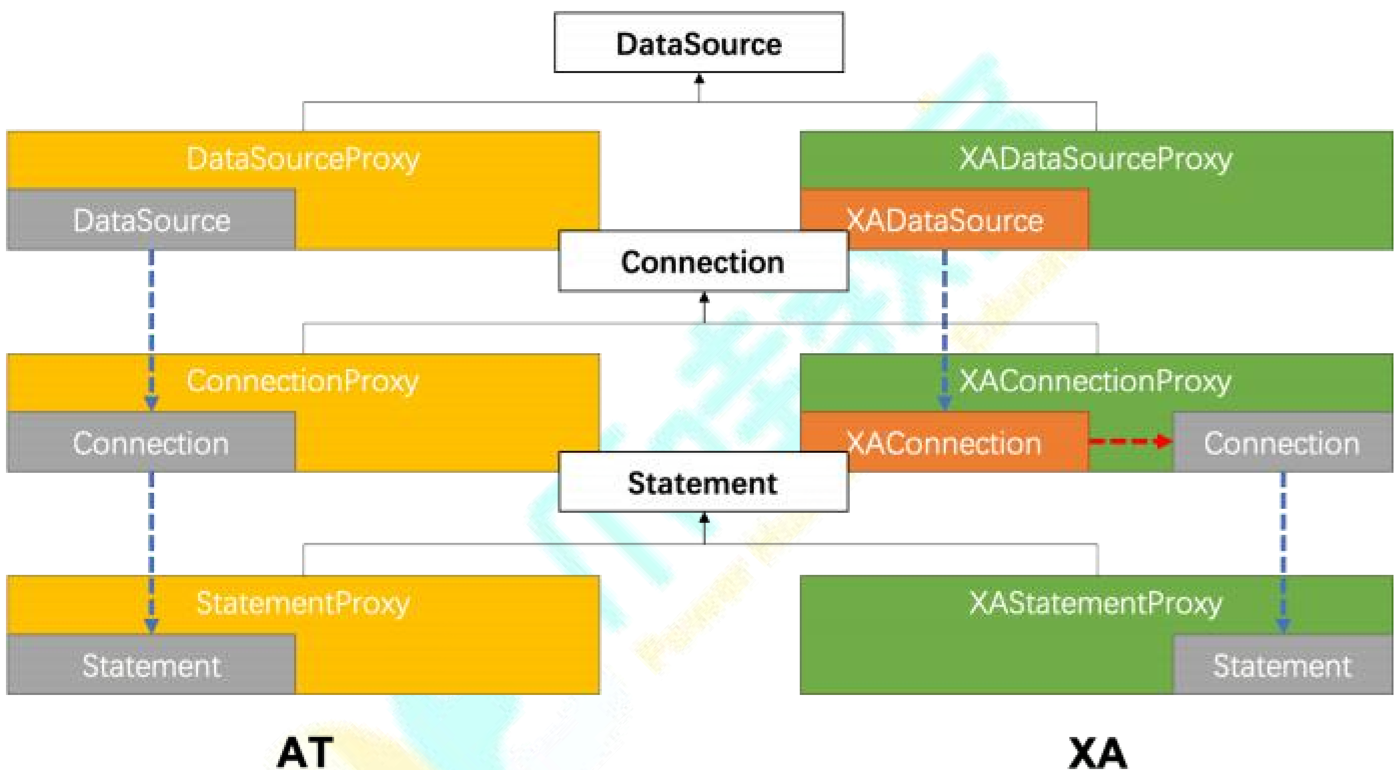


实际上，这种方法是在做数据库驱动程序要做的事情。不同的厂商、不同版本的数据库驱动实现机制是厂商私有的，我们只能保证在充分测试过的驱动程序上是正确的，开发者使用的驱动程序版本差异很可能造成机制的失效。

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：



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。

XA 模式的使用

从编程模型上， XA 模式与 AT 模式保持完全一致。

可以参考 Seata 官网的样例： seata-xa

样例场景是 Seata 经典的，涉及库存、订单、账户 3 个微服务的商品订购业务。

在样例中，上层编程模型与 AT 模式完全相同。只需要修改数据源代理，即可实现 XA 模式与 AT 模式之间的切换。

```
    @Bean("dataSource")
    public DataSource dataSource(DruidDataSource
druidDataSource) {
        // DataSourceProxy for AT mode
        // return new DataSourceProxy(druidDataSource);

        // DataSourceProxyXA for XA mode
        return new DataSourceProxyXA(druidDataSource);
    }
```

面试题标准答案：如何解决分布式事务问题的?

现在Java面试，分布式系统、分布式事务几乎是标配。而分布式系统、分布式事务本身比较复杂，大家学起来也非常头疼。

> 面试题：分布式事务了解吗？你们是如何解决分布式事务问题的?

Seata AT模式和Seata TCC是在生产中最常用。

- 强一致性模型，　Seata AT **强一致方案** 模式用于强一致主要用于核心模块，例如交易/订单等。
- 弱一致性模型。　Seata TCC **弱一致方案**一般用于边缘模块例如库存，通

过TC的协调，保证最终一致性，也可以业务解耦。

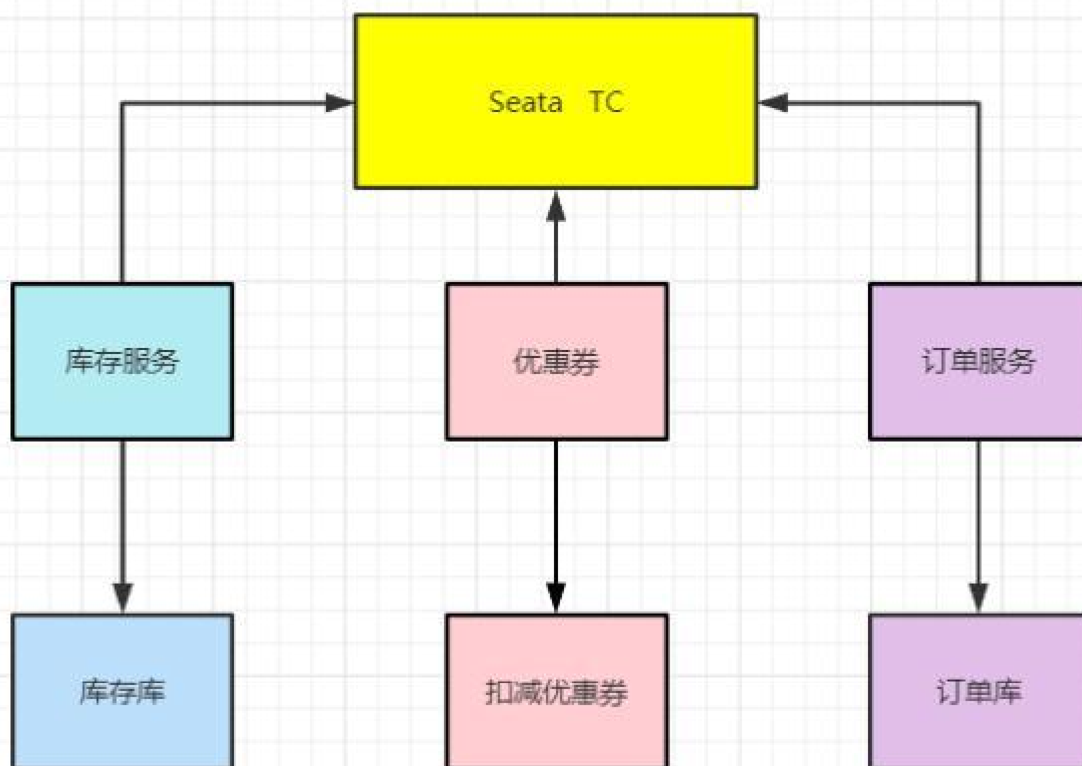
> 面试中如果你真的被问到，可以分场景回答：

（1）强一致性场景

对于那些特别严格的场景，用的是Seata AT模式来保证强一致性；

准备好例子：你找一个严格要求数据绝对不能错的场景（如电商交易交易中的库存和订单、优惠券），可以回答使用成熟的如中间件Seata AT模式。

```
阿里开源了分布式事务框架seata经历过阿里生产环境大量考验的框架。
seata支持Dubbo，Spring Cloud。
```



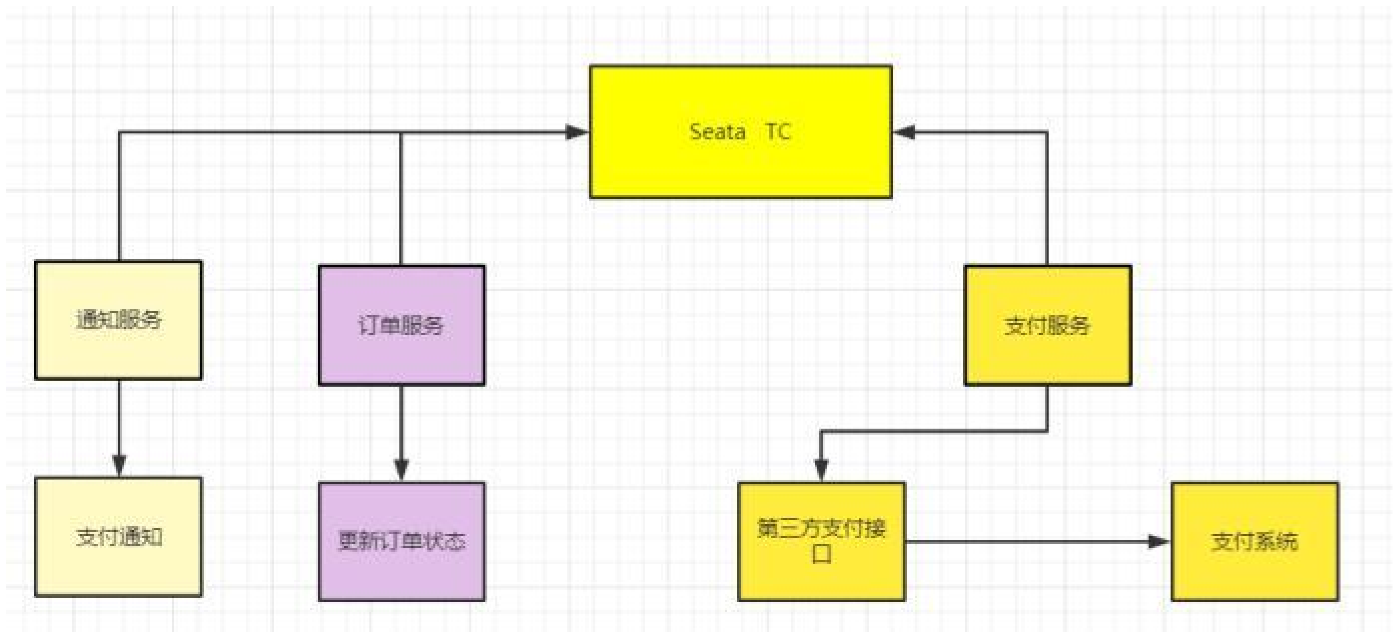
是Seata AT模式，保障强一致性，支持跨多个库修改数据；

- 订单库：增加订单
- 商品库：扣减库存
- 优惠券库：预扣优惠券

（2）弱一致性场景

对于数据一致性要求没有那些特别严格、或者由不同系统执行子事务的场景，可以回答使用Seata TCC 保障弱一致性方案

准备好例子：一个不是严格对数据一致性要求、或者由不同系统执行子事务的场景，如电商订单支付服务，更新订单状态，发送成功支付成功消息，只需要保障弱一致性即可。



Seata TCC 模式，保障弱一致性，支持跨多个服务和系统修改数据，在上面的场景中，使用Seata TCC 模式事务

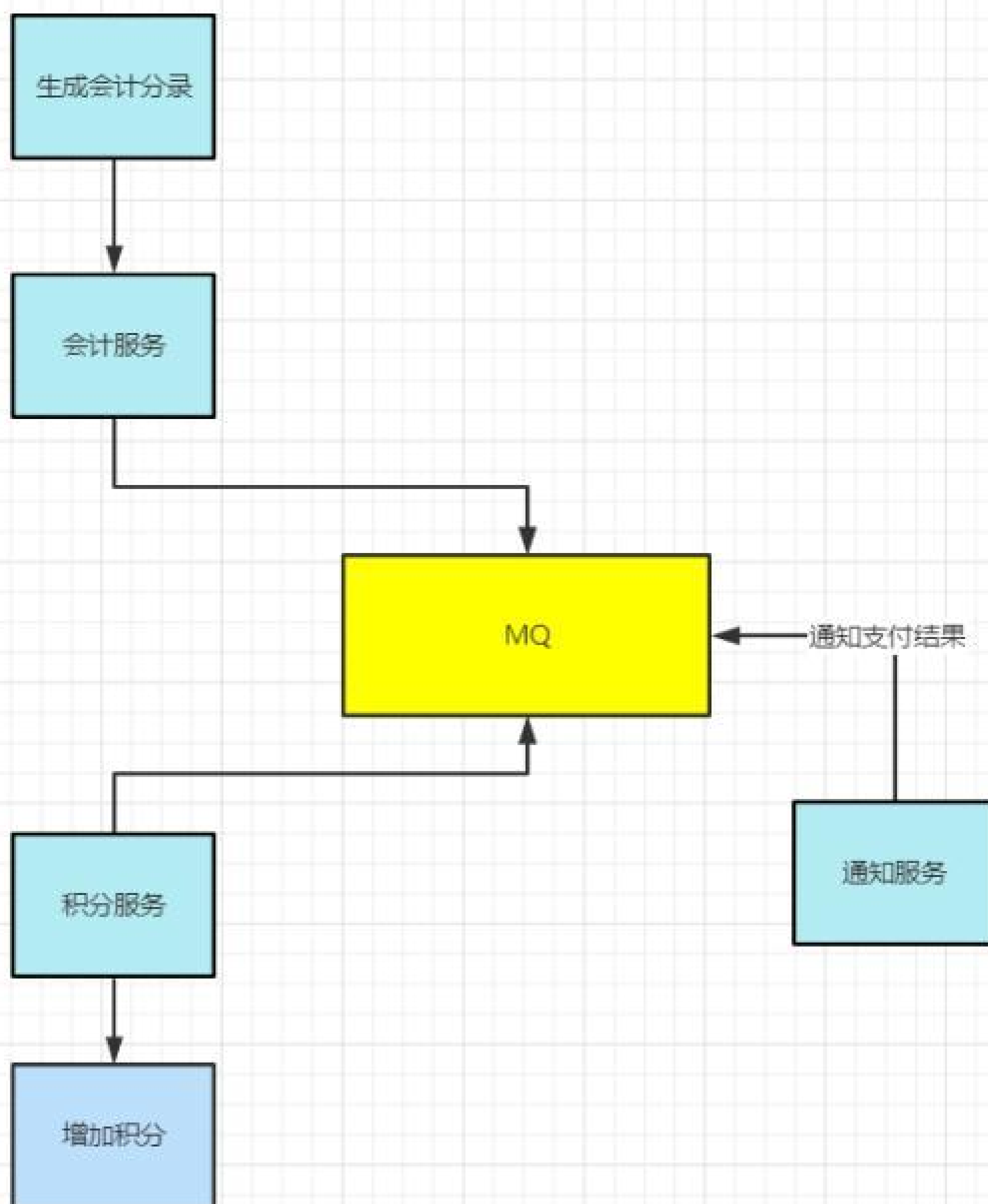
- 订单服务：修改订单状态
- 通知服务：发送支付状态

（3）最终一致性场景

基于可靠消息的最终一致性，各个子事务可以较长时间内异步，但数据绝对不能丢的场景。可以使用**异步确保型事务**事。

可以使用基于MQ的异步确保型事务，比如电商平台的通知支付结果：

- 积分服务：增加积分
- 会计服务：生成会计记录



各大模式的总体对比：

属性	2PC	TCC	Saga	异步确保型事务	尽最大努力通知
事务一致性	强	弱	弱	弱	弱
复杂性	中	高	中	低	低
业务侵入性	小	大	小	中	中
使用局限性	大	大	中	小	中
性能	低	中	高	高	高
维护成本	低	高	中	低	中

参考资料

https://blog.csdn.net/wuzhiwei549/article/details/80692278

https://www.cnblogs.com/seesun2012/p/9214653.html

https://github.com/yangliu0/DistributedLock
https://www.cnblogs.com/liuyang0/p/6744076.html
https://www.cnblogs.com/liuyang0/p/6800538.html
https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/
https://www.infoq.cn/article/cap-twelve-years-later-how-the-rules-have-changed
https://www.cnblogs.com/bluemiaomiao/p/11216380.html

https://www.jianshu.com/p/d909dbaa9d64

https://book.douban.com/subject/26292004/

https://segmentfault.com/a/1190000004468442

https://blog.csdn.net/universsky2015/article/details/105727244/

https://www.cnblogs.com/wudimanong/p/10340948.html

https://blog.csdn.net/kusedexingfu/article/details/103484198

https://www.jianshu.com/p/bfb619d3eea2

http://seata.io/zh-cn/docs/dev/mode/at-mode.html

https://blog.csdn.net/kusedexingfu/article/details/103484198

https://blog.csdn.net/wsdc0521/article/details/108223310

https://blog.csdn.net/lidatgb/article/details/38468005

https://www.cnblogs.com/cuiqq/p/12175826.html

https://blog.csdn.net/qq_22343483/article/details/99638554

https://blog.csdn.net/SOFAStack/article/details/99670033

https://seata.io/zh-cn/docs/dev/mode/at-mode.html

http://t.zoukankan.com/lay2017-p-12528071.html

https://segmentfault.com/a/1190000037757622