



三

君哥的学习笔记

分布式事务之Seata

君哥 2021-06-13 21:44:15

seata

你的支持是君哥更新的动力

阿里巴巴开源分布式事务框架

加入君哥微信粉丝群：coding178

点击观看全套视频

第一章：分布式事务基础



事务

概念

事务

指的就是一个操作单元，在这个操作单元中的所有操作最终要保持一致的行为，要么所有操作都成功，要么所有的操作都被撤销。



通俗一点？举个生活中的例子：你去小卖铺买东西，“一手交钱，一手交货”就是一个事务的例子，交钱和交货必须全部成功，事务才算成功，任一个活动失败，事务将撤销所有已成功的活动。

事务可以看做是一次大的操作，它由不同的小操作组成，这些操作要么全部成功，要么全部失败。

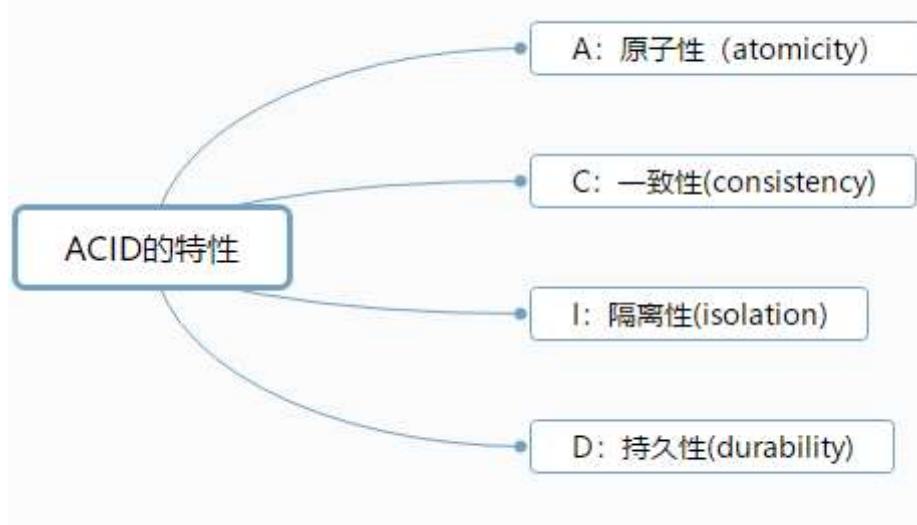
例如



君哥的学习笔记

入门 · 基础 · 深度 · 高级 · 其他

事务的4个特性 ACID



原子性 (Atomicity) : 操作这些指令时，要么全部执行成功，要么全部不执行。只要其中一个指令执行失败，所有的指令都执行失败，数据进行回滚，回到执行指令前的数据状态。
要么执行，要么不执行



一致性 (Consistency) : 事务的执行使数据从一个状态转换为另一个状态，数据库的完整性约束没有被破坏。 能量守恒，总量不变

eg: 拿转账来说，假设用户A和用户B两者的钱加起来一共是2000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是2000，这就是事务的一致性。

隔离性 (Isolation) : 隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。
信息彼此独立，互不干扰



即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

持久性 (Durability) : 当事务正确完成后，它对于数据的改变是永久性的。 不会轻易丢失

eg: 例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。



本地事务实现

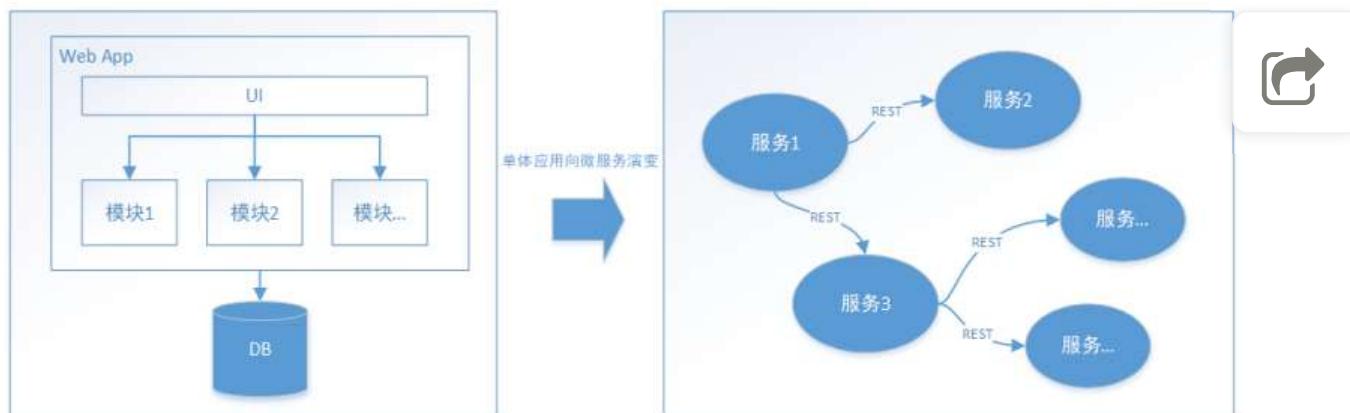
```

1 begin transaction;
2 //1.本地数据库操作：张三减少金额
3 //2.本地数据库操作：李四增加金额
4 commit transaction;

```

分布式事务

随着互联网的快速发展，软件系统由原来的单体应用转变为分布式应用，下图描述了单体应用向微服务的演变：分布式系统会把一个应用系统拆分为可独立部署的多个服务，因此需要服务与服务之间远程协作才能完成事务操作，这种分布式系统环境下由不同的服务之间通过网络远程协作完成事务称之为分布式事务，例如用户注册送积分 事务、创建订单减库存事务，银行转账事务等都是分布式事务。



典型的场景就是微服务架构 微服务之间通过远程调用完成事务操作。比如：订单微服务和库存微服务，下单的同时订单微服务请求库存微服务减库存。简言之：跨JVM进程产生分布式事务。



分布式事务实现

```

1 begin transaction;
2 //1.本地数据库操作：张三减少金额
3 //2.远程调用：让李四增加金额
4 commit transaction;

```



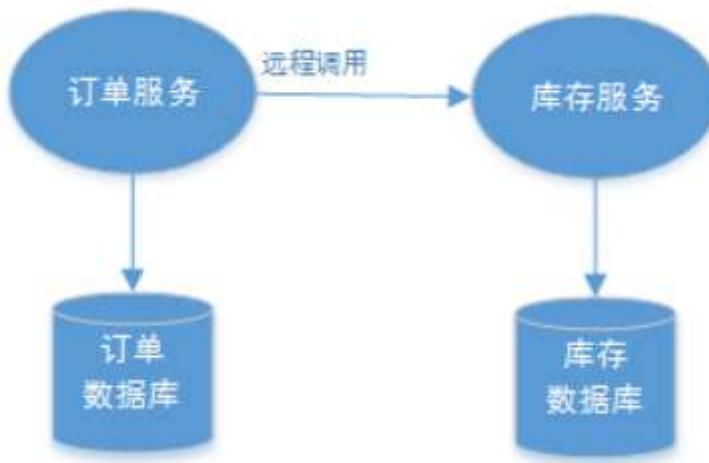
三 君哥的学习笔记

分布式架构的基础上，传统数据库事务就无法使用了，张三和李四的账户不在一个数据库中甚至不在一个应用系统里，实现转账事务需要通过远程调用，由于网络问题就会导致分布式事务问题。

分布式事务产生场景

1. 跨JVM进程产生分布式事务

典型的场景就是微服务架构 微服务之间通过远程调用完成事务操作。比如：订单微服务和库存微服务，下单的同时订单微服务请求库存微服务减库存。



2. 跨数据库实例

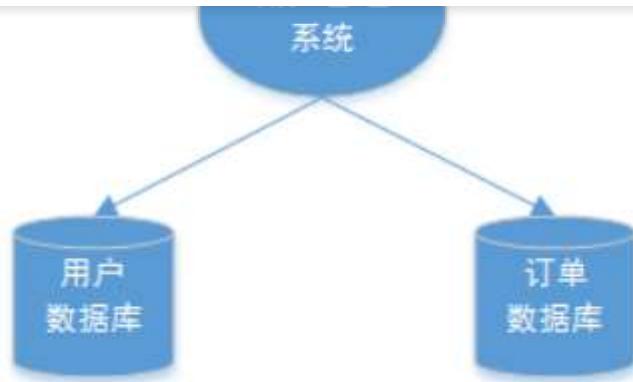
单体系统访问多个数据库实例 当单体系统需要访问多个数据库（实例）时就会产生分布式事务。比如：用户信息和订单信息分别在两个MySQL实例存储，用户管理系统删除用户信息，需要分别删除用户信息及用户的订单信息，由于数据分布在不同的数据实例，需要通过不同的数据库链接去操作数据，此时产生分布式事务。





三

君哥的学习笔记

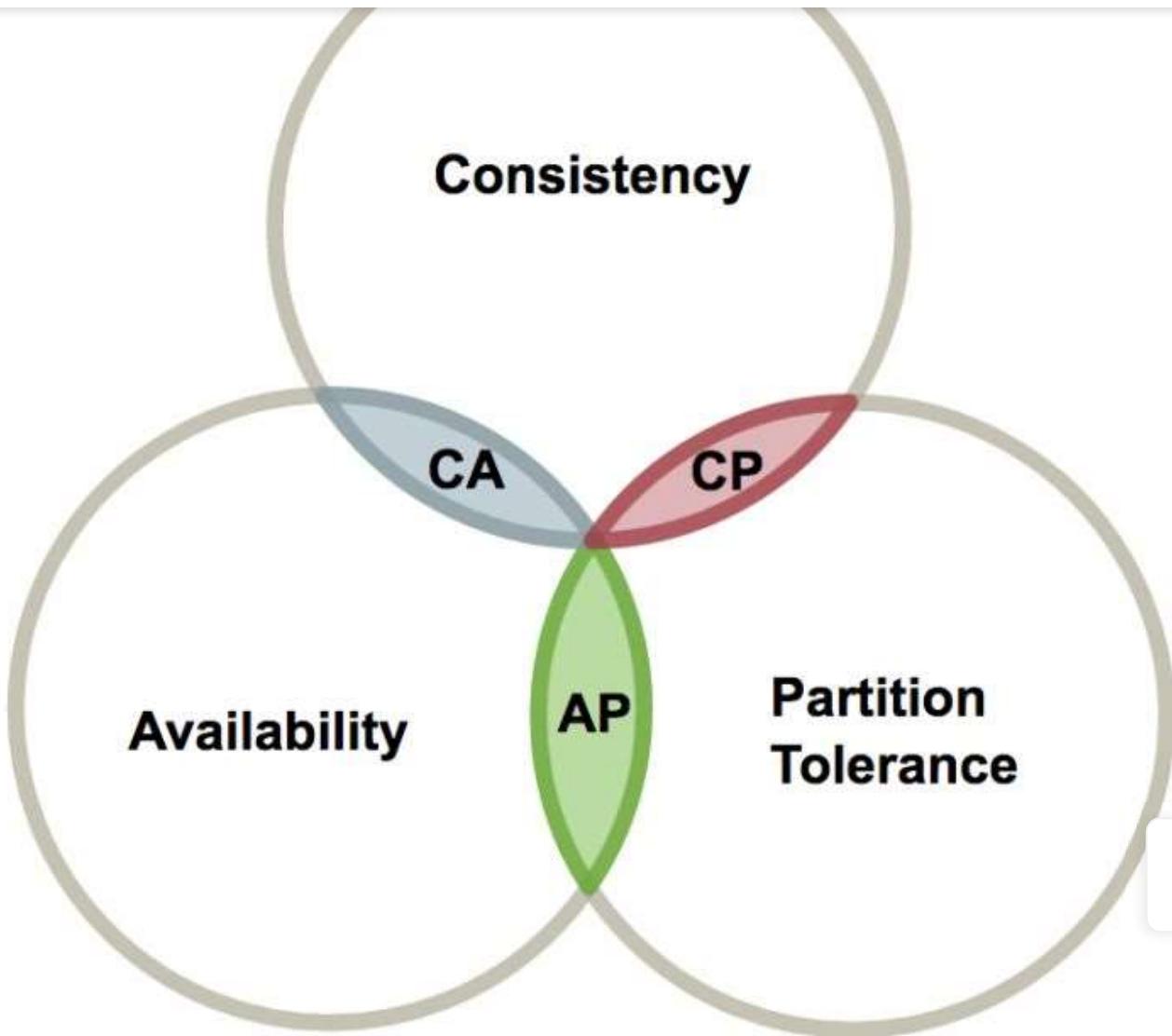


我们了解到了分布式事务的基础概念。与本地事务不同的是，分布式系统之所以叫分布式，是因为提供服务的各个节点分布在不同机器上，相互之间通过网络交互。不能因为有一点网络问题就导致整个系统无法提供服务，网络因素成为了分布式事务的考量标准之一。因此，分布式事务需要更进一步的理论支持，接下来，我们先来学习一下分布式事务的CAP理论。

CAP原则

CAP原则又叫CAP定理，同时又被称作布鲁尔定理（Brewer's theorem），指的是在一个分布式系统中，**不可能同时满足以下三点**。





- 一致性 (Consistency) 副本最新 : 指强一致性，在写操作完成后开始的任何读操作都必须返回该值，或者后续写操作的结果。

也就是说，在一致性系统中，一旦客户端将值写入任何一台服务器并获得响应，那么之后client从其他任何服务器读取的都是刚写入的数据

一致性保证了不管向哪台服务器写入数据，其他的服务器能实时同步数据



- 可用性 (Availability) 高可用 : 可用性是指，每次向未崩溃的节点发送请求，总能保证收到响应数据（允许不是最新数据）

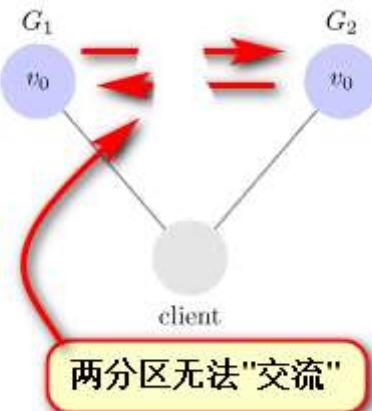
什么是分区？

在分布式系统中，不同的节点分布在不同的子网络中，由于一些特殊的原因，这些子节点之间出现了网络不通的状态，但他们的内部子网络是正常的。从而导致了整个系统的环境被切分成了若干个孤立的区域。这就是分区。



三 君哥的学习笔记

方的任何消息都是可以放弃的，也就是说A和B可能因为各种意外情况，导致无法成功进行同步，分布式系统要能容忍这种情况。除非整个网络环境都发生了故障。



容许节点 G_1/G_2 间传递消息的差错（延迟或丢失），而不影响系统继续运行。

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。

为什么只能在A和C之间做出取舍？

分布式系统中，必须满足 CAP 中的 P，此时只能在 C/A 之间作出取舍。



整个系统由两个节点配合组成，之间通过网络通信，当节点 A 进行更新数据库操作的时候，需要同时更新节点 B 的数据库（这是一个原子的操作）。

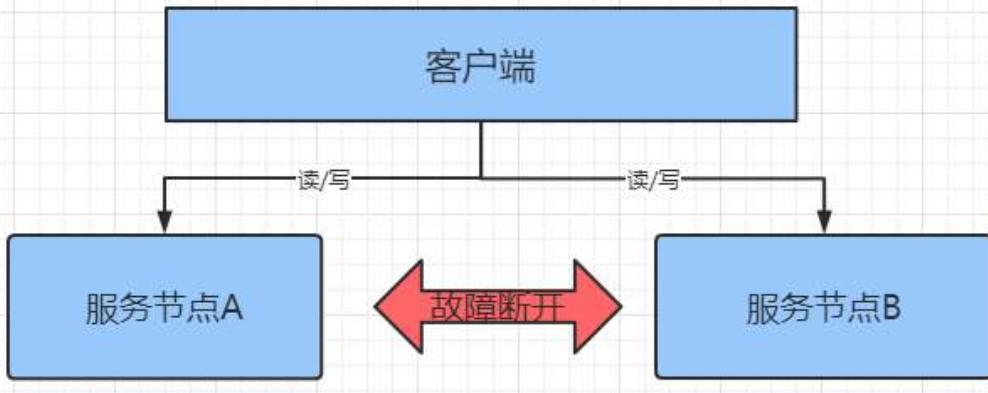
下面这个系统怎么满足 CAP 呢？我们用反证法 假设可以同时满足一致性、可用性、分区容错这三个特性，由于满足分区容错，可以切断 A/B 的连线，如下图：





三 君哥的学习笔记

- 一致性：当节点A更新的时候，节点B也要更新(同步)
- 可用性：必须保证两个节点都是可用的
- 当节点 A,B 出现了网络分区，必须保证对外可用
- 3点能同时满足吗？



可见，根本完成不了，只要出现了网络分区，A 就无法满足，因为节点 A 根本连接不上节点 B。如果强行满足 C 原子性，就必须停止服务运行，从而放弃可用性 C。

- 若要保证一致性：则必须进行节点间数据同步，同步期间数据锁定，导致期间的读取失败或超时，破坏了可用性；
- 若要保证可用性：则不允许节点间同步期间锁定，这又破坏了一致性。



所以，最多满足两个条件：

组合	分析结果
CA	满足原子和可用，放弃分区容错。说白了，就是一个整体的应用。
CP	满足原子和分区容错，也就是说，要放弃可用。当系统被分区，为了保证原子性，必须放弃可用性，让服务停用。
AP	满足可用性和分区容错，当出现分区，同时为了保证可用性，必须让节点继续对外服务，这样必然导致失去原子性。



如何权衡 保C还是保A？

取舍

- 舍弃P(选择C/A)：单点的传统关系型数据库 DBMS(MySQL/Oracle)，但如果采用集群就必须考虑P了；
- 舍弃A(选择C/P)：是分布式系统要保证P，而且保证一致性，如 ZooKeeper / Redis / MongoDB / HBase；



三 君哥的学习笔记

对于一个分布式系统来说，CAP三者中，

- P是基本要求，只能通过基础设施提升，无法通过降低 C/A 来提升；
- 然后在 C/A 两者之间权衡。

一个还不错的策略是：保证可用性和分区容错，舍弃强一致性，但保证最终一致性，比如一些高并发的站点（秒杀、淘宝、12306）。最终近似于兼顾了三个特性。

一致性 数据一致性

一致性可以分为强一致性与弱一致性。所谓强一致性，即复制是同步的，弱一致性，即复制是异步的。

CAP回顾

CAP理论告诉我们一个悲惨但不得不接受的事实——我们只能在C、A、P中选择两个条件。而对于业务系统而言，我们往往选择牺牲一致性来换取系统的可用性和分区容错性。不过这里要指出的是，所谓的“牺牲一致性”并不是完全放弃数据一致性，而是**牺牲强一致性换取弱一致性**。



强一致性

系统中的某个数据被成功更新后，后续任何对该数据的读取操作都将得到更新后的值；

也称为：原子一致性 (Atomic Consistency) 线性一致性 (Linearizable Consistency)

两个要求：

- 任何一次读都能读到某个数据的最近一次写的数据。
- 系统中的所有进程，看到的操作顺序，都和全局时钟下的顺序一致。



简言之，在任意时刻，所有节点中的数据是一样的。例如，对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。

总结：

- 一个集群需要对外部提供强一致性，所以只要集群内部某一台服务器的数据发生了改变，那么就需要等待集群内其他服务器的数据同步完成后，才能正常的对外提供服务。
- 保证了强一致性，务必会损耗可用性。



三 君哥的学习笔记

弱一致性

系统中的某个数据被更新后，后续对该数据的读取操作可能得到更新后的值，也可能是更改前的值。

但即使过了**不一致时间窗口**这段时间后，后续对该数据的读取也不一定是最新值

所以说，可以理解为数据更新后，如果能容忍后续的访问只能访问到部分或者全部访问不到，则是弱一致性。

最终一致性

是弱一致性的特殊形式，存储系统保证在没有新的更新的条件下，最终所有的访问都是最后更新的值。

不保证在任意时刻任意节点上的同一份数据都是相同的，但是随着时间的迁移，不同节点上的同一份数据总是在向趋同的方向变化。

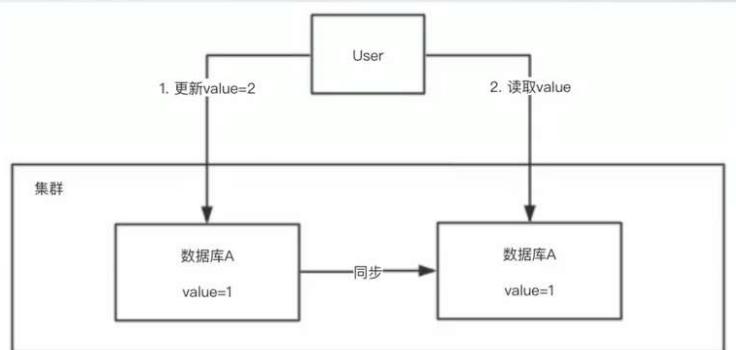
简单说，就是在一段时间后，节点间的数据会最终达到一致状态。

一致性总结

弱一致性即使过了不一致时间窗口，后续的读取也不一定能保证一致，而最终一致过了不一致窗口后，后续的读取一定一致

一致性

- 强制要求步骤2读取的时候，一定要读取的是2，不能读取到的是1，那么要求数数据库之间同步非常迅速或者在步骤2上加锁以等待数据同步完成，那么这种叫强一致性；
- 允许步骤2读取的时候，可以读取的是1，那么这种叫弱一致性，其实质就是不需要一致；
- 允许步骤2读取的时候，可以先读到1，过一段时间再读到2，那么这种叫最终一致性，就是可以等待一段时间才一致；



<https://blog.csdn.net/Soinica>



三 君哥的学习笔记

Base理论

- BA: Basic Available 基本可用
 - 整个系统在某些不可抗力的情况下，仍然能够保证“可用性”，即一定时间内仍然能够返回一个明确的结果。只不过“基本可用”和“高可用”的区别是：
 - “一定时间”可以适当延长 当举行大促时，响应时间可以适当延长
 - 给部分用户返回一个降级页面 给部分用户直接返回一个降级页面，从而缓解服务器压力。但要注意，返回降级页面仍然是返回明确结果。
- S: Soft State 柔性状态：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，即允许系统不同节点的数据副本之间进行数据同步的过程存在延时。
- E: Eventual Consistency 最终一致性：同一数据的不同副本的状态，可以不需要实时一致，但一定要保证经过一定时间后仍然是一致的。

BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。



分布式事务协议

背景

在分布式系统里，每个节点都可以知晓自己操作的成功或者失败，却无法知道其他节点操作的成功或失败。当一个事务跨多个节点时，为了保持事务的原子性与一致性，而引入一个协调者来统一掌控所有参与者的操作结果，并指示它们是否要把操作结果进行真正的提交或者回滚（rollback）。



二阶段提交 2PC

二阶段提交协议（Two-phase Commit，即 2PC）是常用的分布式事务解决方案，即将事务的提交过程分为两个阶段来进行处理。

阶段

- 准备阶段
- 提交阶段

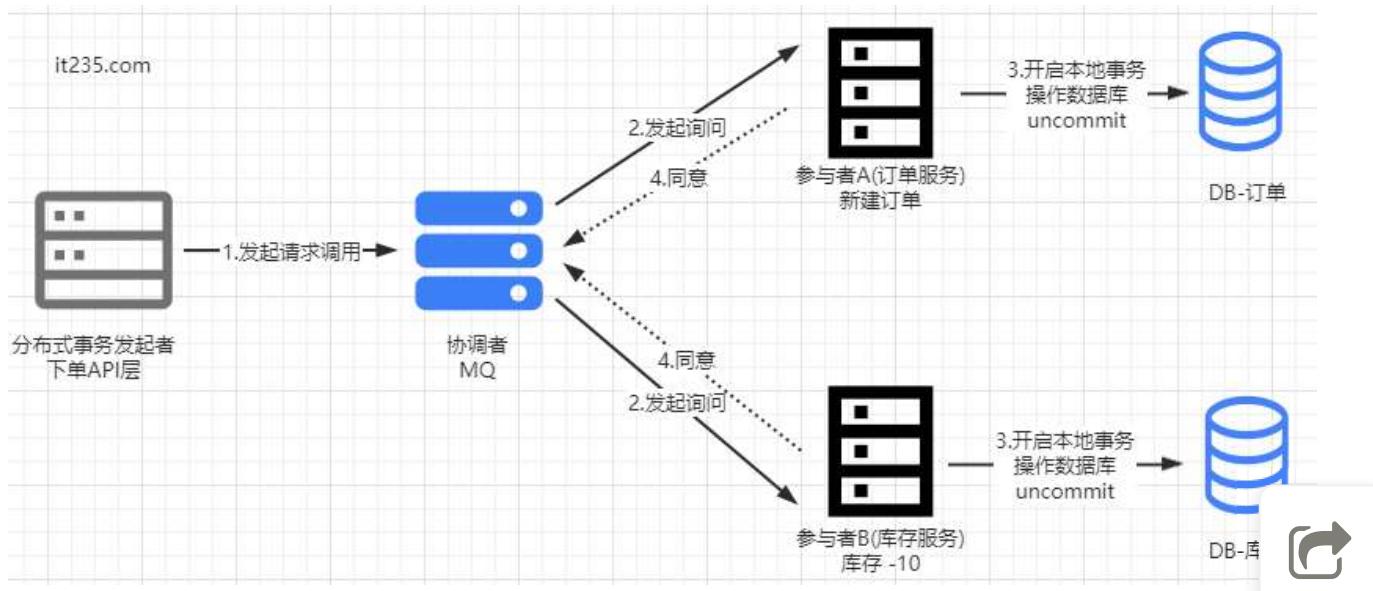
三 君哥的学习笔记

• 1. 什么事物 · 2. 参与者 · 3. 协调者

- 参与者：事务的执行者

1. 第一阶段 (voting phase 投票阶段) () :

1. 协调者向所有参与者发送事务内容，询问是否可以提交事务，并等待答复
2. 各参与者执行事务操作，将 undo 和 redo 信息记入事务日志中（但不提交事务）
3. 如参与者执行成功，给协调者反馈同意，否则反馈中止

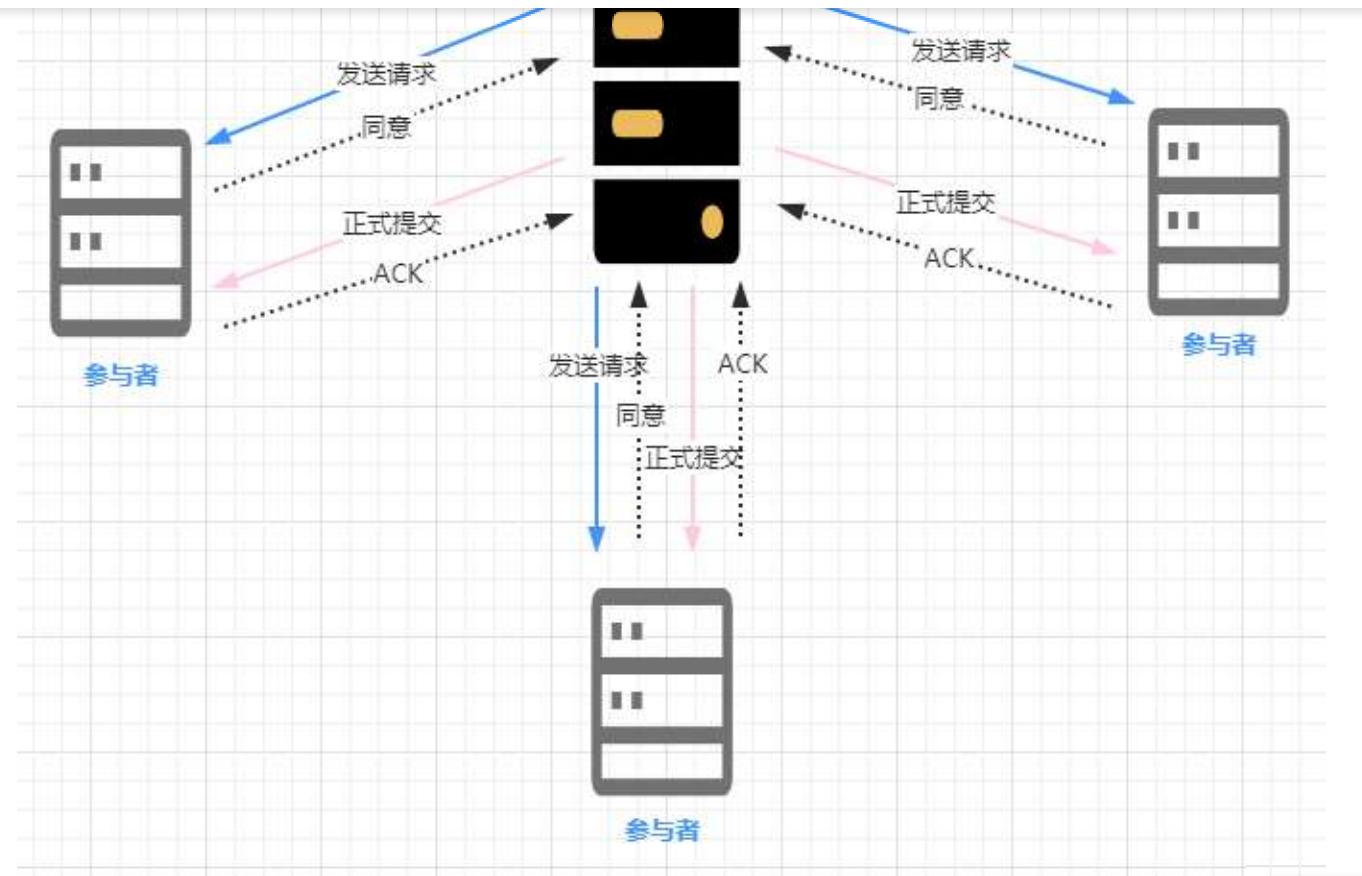


2. 第二阶段 (commit phase 提交执行阶段) () :

当协调者节点从所有参与者节点获得的相应消息都为同意时：

1. 协调者节点向所有参与者节点发出正式提交 (commit) 的请求。
2. 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。
3. 参与者节点向协调者节点发送ack完成消息。
4. 协调者节点收到所有参与者节点反馈的ack完成消息后，完成事务。

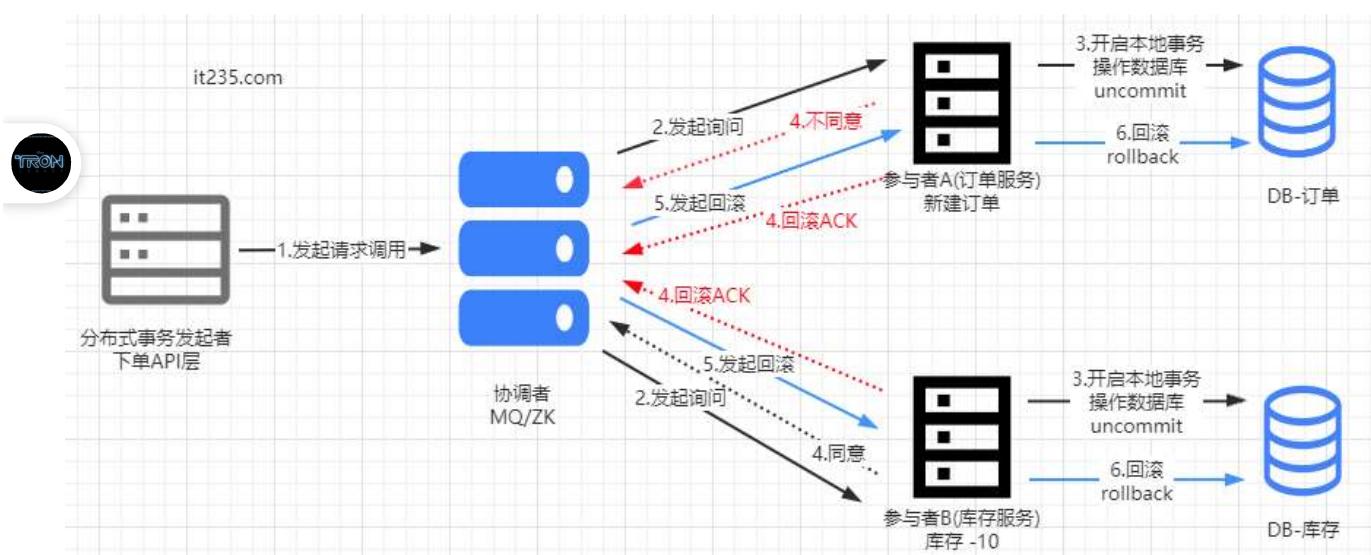


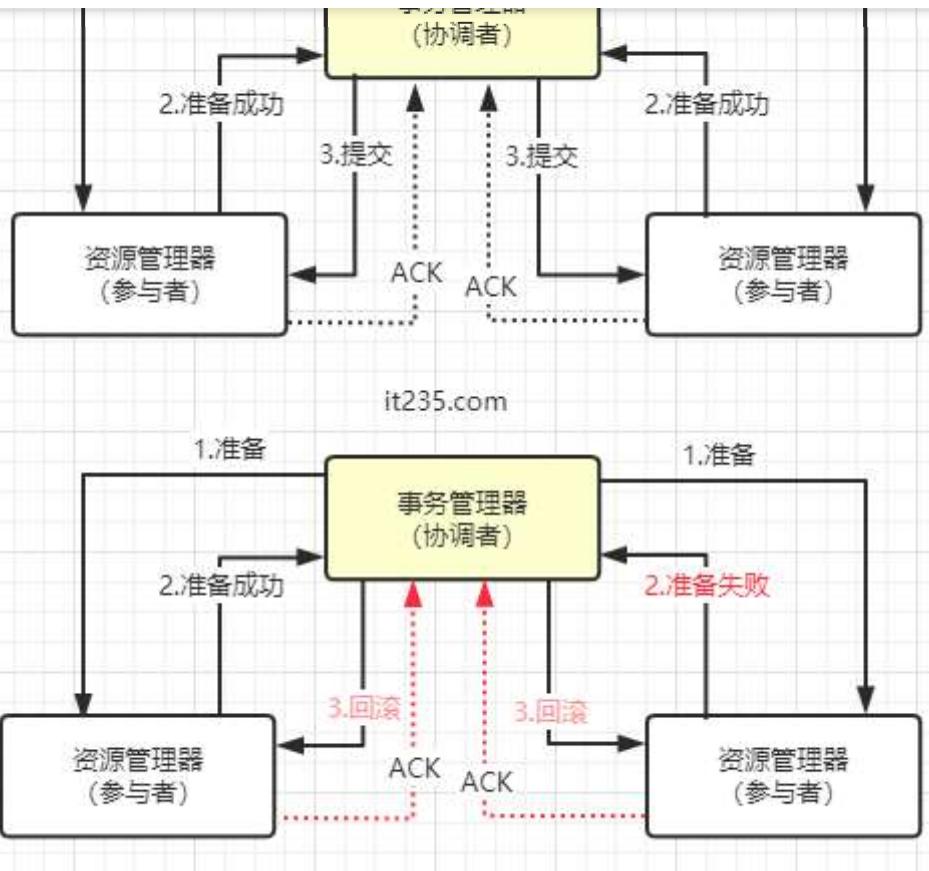


如果任一参与者节点在第一阶段返回的响应消息为**中止**，或者 协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时：

1. 协调者节点向所有参与者节点发出**回滚操作**(rollback)的请求。
2. 参与者节点利用阶段1写入的undo信息执行回滚，并释放放在整个事务期间内占用的资源。
3. 参与者节点向协调者节点发送**ack回滚完成消息**。
4. 协调者节点受到所有参与者节点反馈的**ack回滚完成消息**后，取消事务。

不管最后结果如何，第二阶段都会结束当前事务。





两阶段案例

1 学校运动会上，100米决赛正准备开始，裁判对3个人分别询问
 2 裁判：张三同学你准备好了吗？准备好了进第一赛道
 3 张三：准备好了，随即进入第一赛道做好冲击姿势
 4 裁判：李四同学你准备好了吗？准备好了进第二赛道
 5 裁判：王五同学你准备好了吗？准备好了进第三赛道
 6 王五：准备好了，.....
 7 李四：准备好了，.....
 8 ...
 9 如果有人没准备好，不同意，则裁判下达回滚指令
 10
 11 如果裁判收到了所有人的OK回复后，再次下令
 12 裁判：跑...
 13 ...
 14 张三、李四、执行完毕到达终点，汇报给了裁判
 15 王五冲刺失败，汇报给了裁判

二阶段提交看起来确实能够提供原子性的操作，但是不幸的是，二阶段提交还是有几个缺点的：



三 君哥的学习笔记

2. **可靠性问题**: 参与者发生故障。协调者需要给每个参与者额外指定超时机制，超时后整个事务失败。协调者发生故障。参与者会一直阻塞下去。需要额外的备机进行容错。

3. **数据一致性问题**: 二阶段无法解决的问题：协调者在发出 `commit` 消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

优点

尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）

缺点

实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景。

为此，Dale Skeen和 Michael Stonebraker 在“ A Formal Model of Crash Recovery in a Distributed System ”中提出了三阶段提交协议（3PC）。



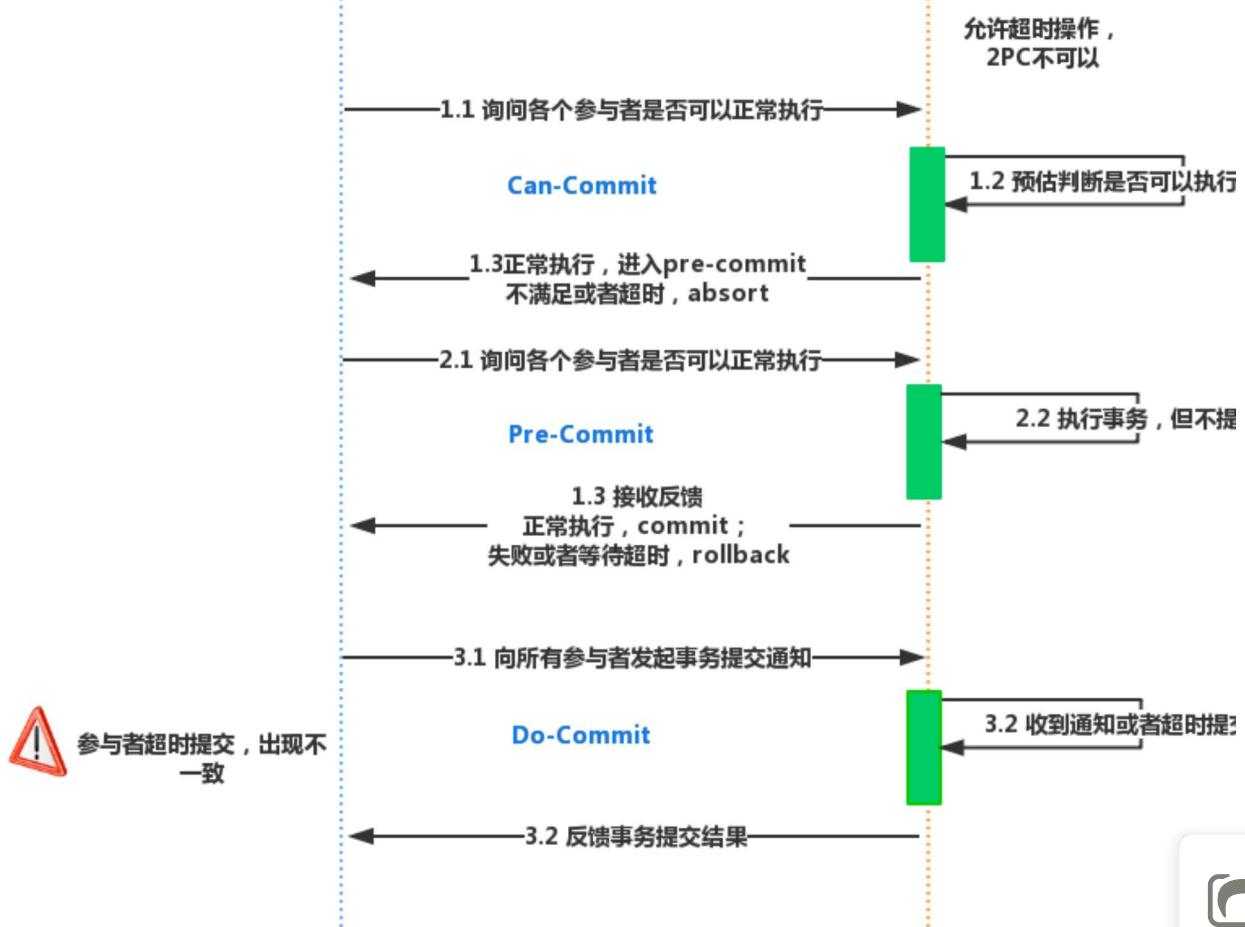
三阶段提交（3PC）

三阶段提交协议，是二阶段提交协议的改进版本，三阶段提交有两个改动点。

- 在协调者和参与者中都引入超时机制。
- 在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

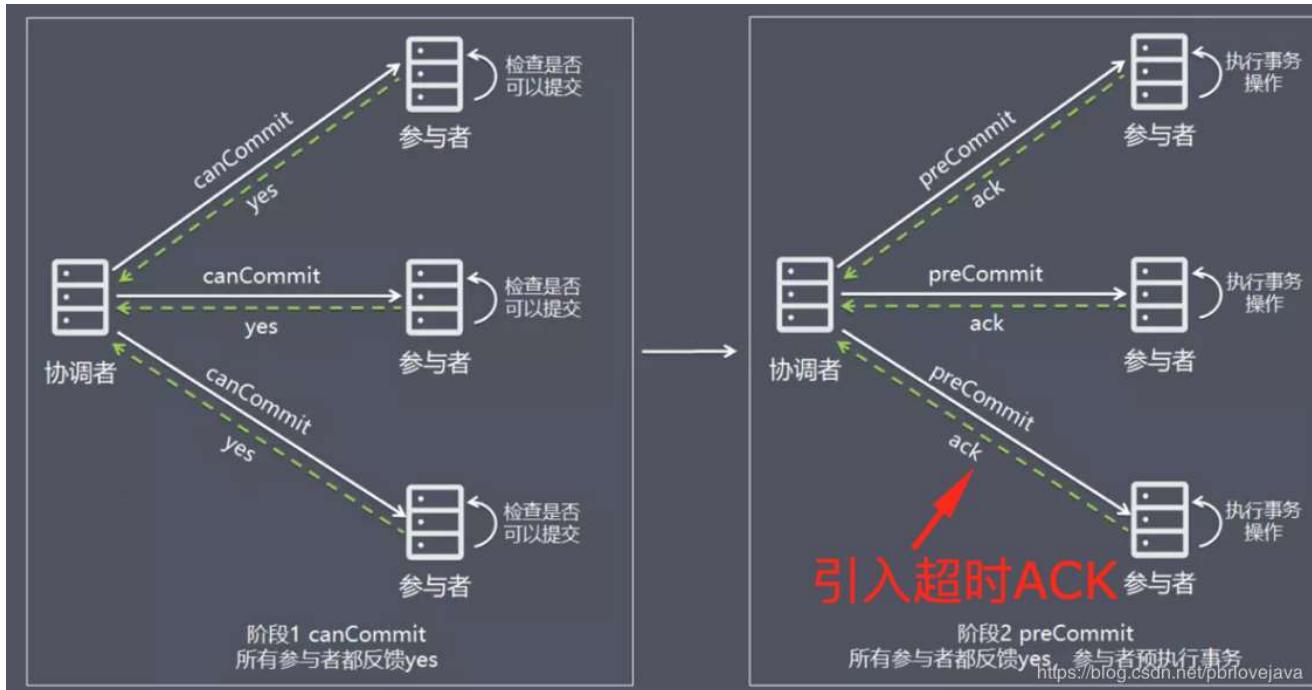
也就是说，除了引入超时机制之外，3PC把2PC的准备阶段再次一分为二，这样三阶段提交就有 `CanCommit` 、 `PreCommit` 、 `DoCommit` 三个阶段。处理流程如下：





小例子

1 班长要组织全班同学聚餐，由于大家毕业多年，所以要逐个打电话敲定时间，时间初定**10.1**日。
 2
 3 班长：小A，我们想定在**10.1**号聚会，你有时间嘛？有时间你就说YES，没有你就说NO，然后我还得
 4
 5 小A：好的，我有时间。（参与者反馈）
 6
 7 班长：小B，我们想定在**10.1**号聚会.....不用一直等我。
 8
 9 班长收集完大家的时间情况了，一看大家都有时间，那么就再次通知大家。（协调者接收到所有YI
 10
 11 班长：小A，我们确定了**10.1**号聚餐，你要把这一天的时间空出来，这一天你不能再安排其他的事
 12
 13 小A顺手在自己的日历上把**10.1**号这一天圈上了，然后跟班长说，我可以去。（参与者执行事务操
 14
 15 班长：小B，我们觉得了**10.1**号聚餐.....你就**10.1**号那天来聚餐就行了。
 16
 17 班长通知完一圈之后。所有同学都跟他说：“我已经把**10.1**号这天空出来了”。于是，他在**10.1**号



1. 阶段一：CanCommit阶段

3PC的 CanCommit 阶段其实和2PC的准备阶段很像。协调者向参与者发送 commit 请求，参与者如果可以提交就返回Yes响应，否则返回No响应。

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

2. PreCommit阶段

协调者根据参与者的反应情况来决定是否可以进行事务的 PreCommit 操作。根据响应情况，有以下两种可能。

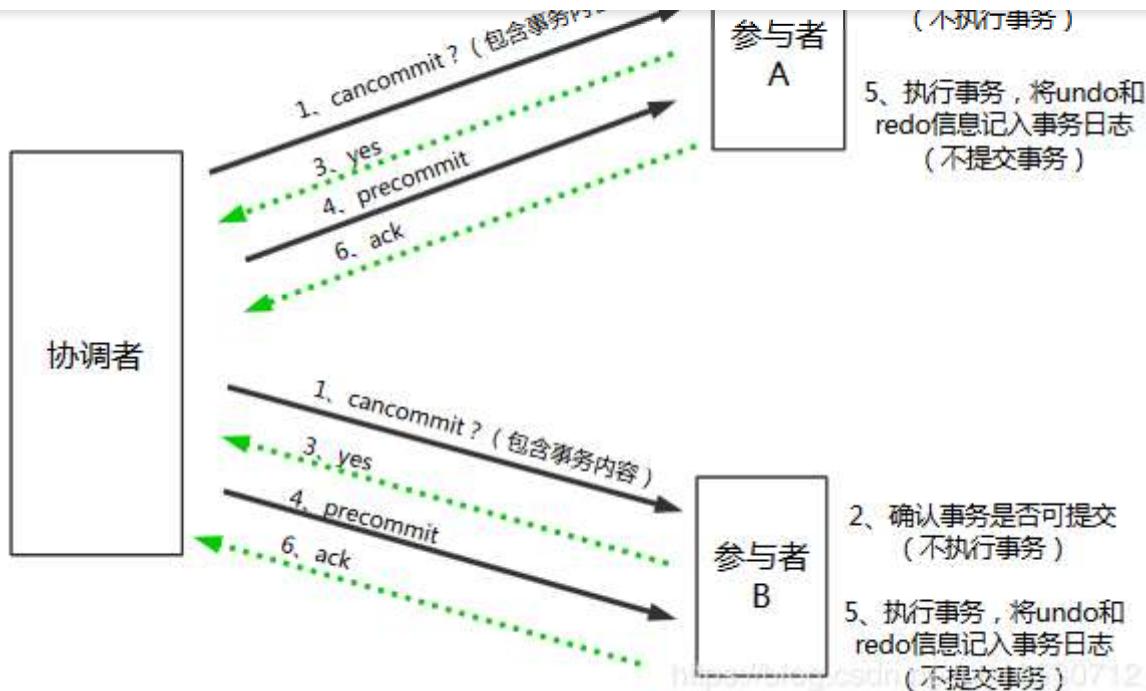


- 假如所有参与者均反馈 yes，协调者预执行事务。

1. 发送预提交请求：协调者向参与者发送 PreCommit 请求，并进入准备阶段
2. 事务预提交：参与者接收到 PreCommit 请求后，会执行事务操作，并将 undo 和 redo 信息记录到事务日志中（但不提交事务）
3. 响应反馈：如果参与者成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

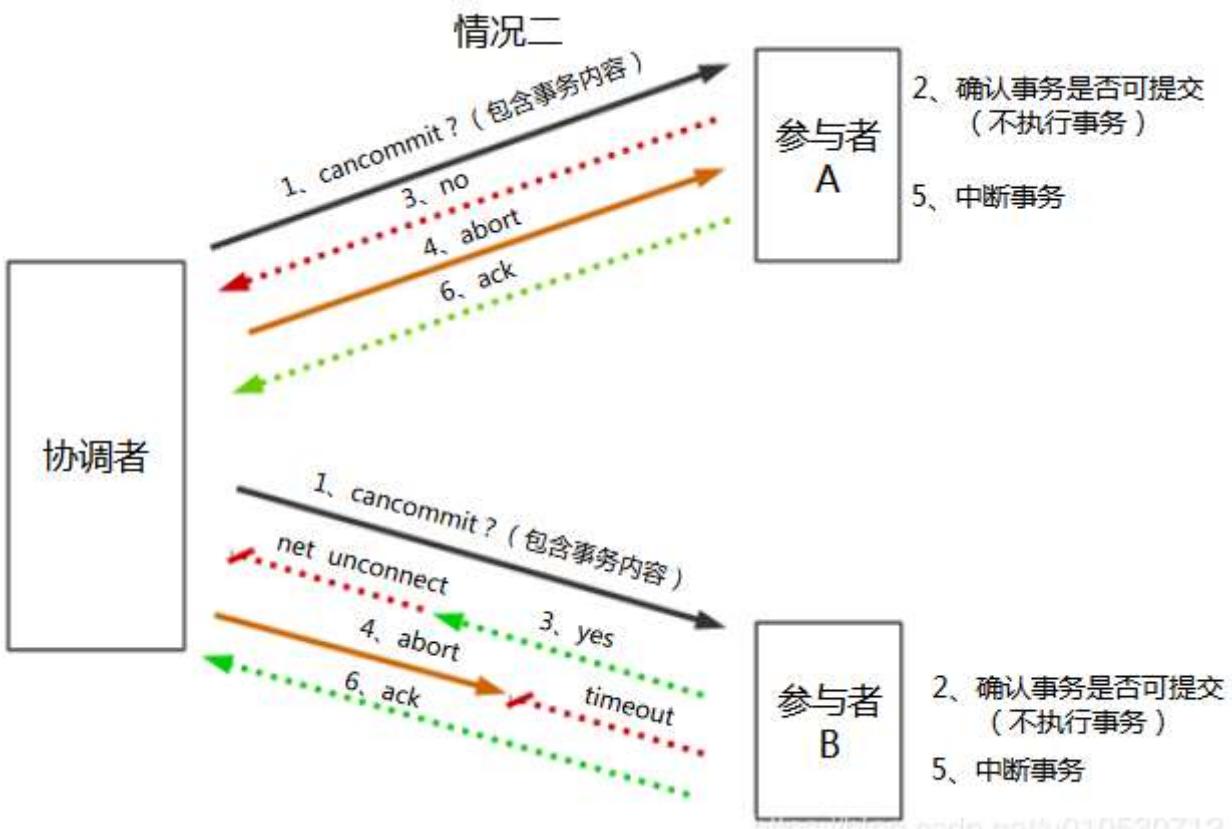


君哥的学习笔记



- 假如有任何一个参与者向协调者发送了No响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

- 发送中断请求：协调者向所有参与者发送 **abort** 请求。
- 中断事务：参与者收到来自协调者的 **abort** 请求之后（或超时之后，仍未收到协调的请求），执行事务的中断。





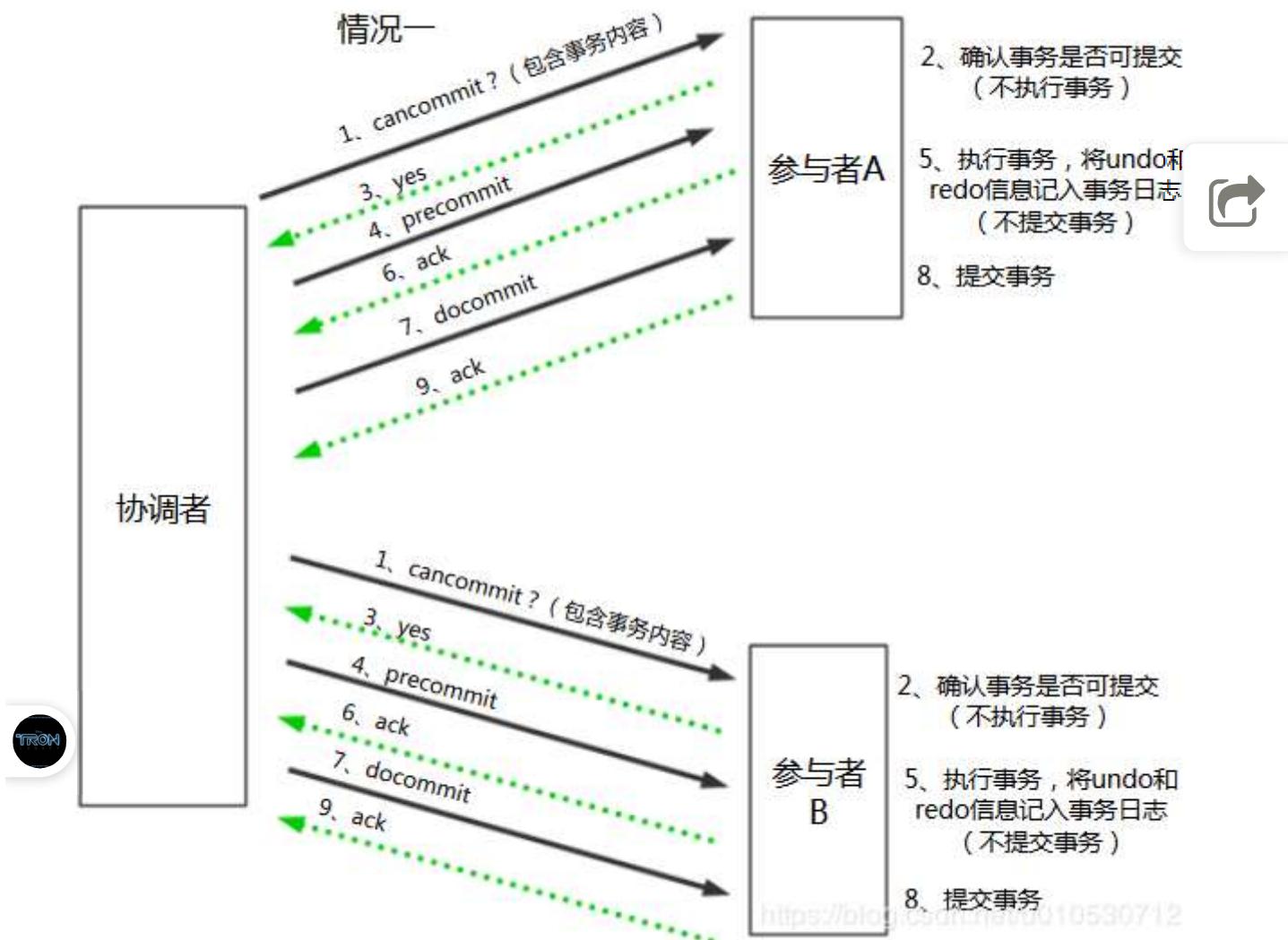
三 君哥的学习笔记

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

3.1 执行提交

所有参与者均反馈 ack 响应，执行真正的事务提交

1. 发送提交请求 协调接收到参与者发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送 doCommit 请求。
2. 事务提交 参与者接收到 doCommit 请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。
3. 响应反馈 事务提交完之后，向协调者发送ack响应。
4. 完成事务 协调者接收到所有参与者的ack响应之后，完成事务。



3.2 中断事务

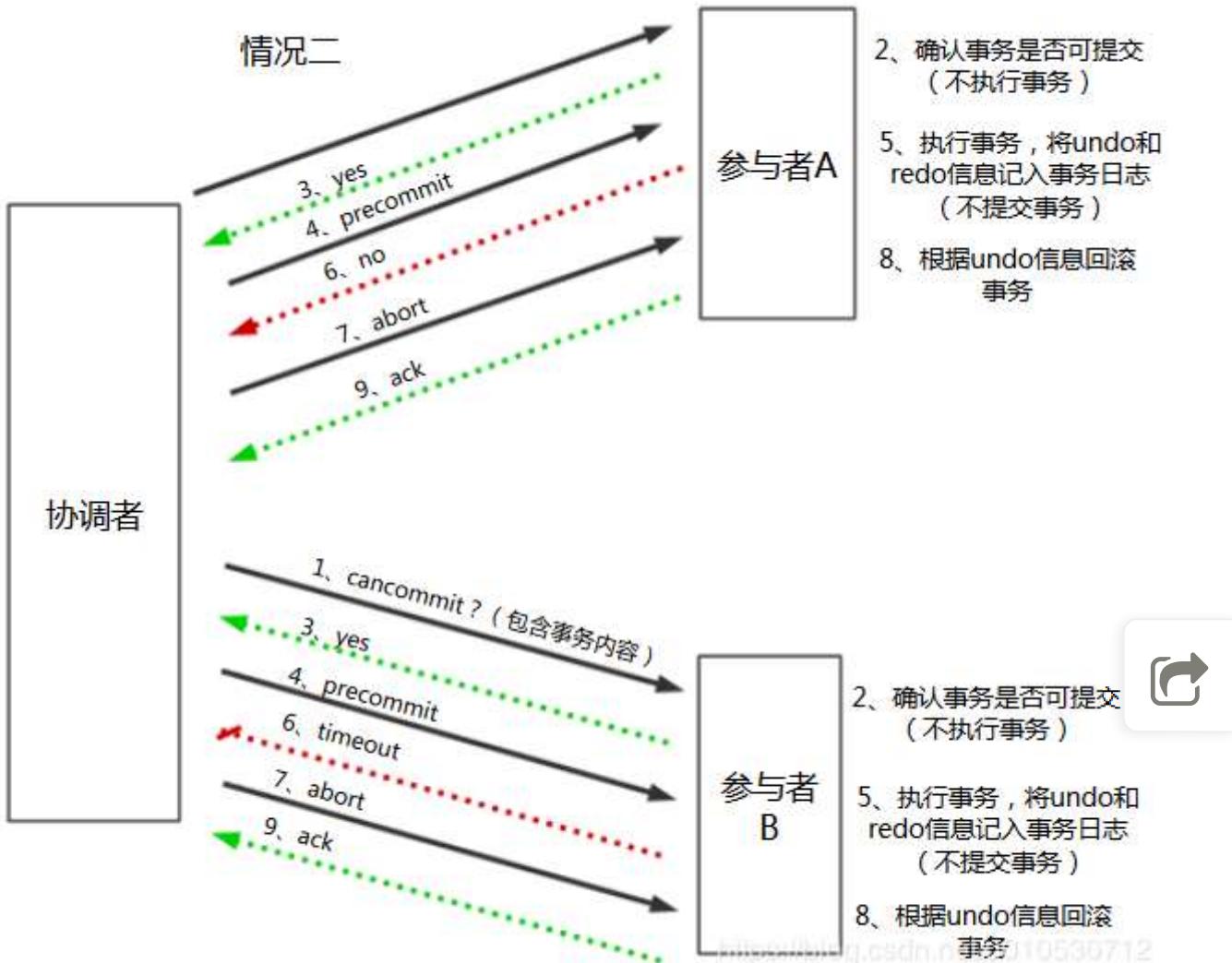
任何一个参与者反馈 no，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务



三 君哥的学习笔记

滚操作，并在完成回滚之后释放所有的事务资源。

3. 反馈结果 参与者完成事务回滚之后，向协调者反馈ACK消息
4. 中断事务 协调者接收到参与者反馈的ACK消息之后，执行事务的中断。



注意

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，会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。）



三 君哥的学习笔记

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

分布式事务解决方案

- TCC
- 全局消息
- 基于可靠消息服务的分布式事务
- 最大努力通知

事务补偿 (TCC)

TCC方案是一种应用层面侵入业务的两阶段提交。是目前最火的一种柔性事务方案，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作

1. 第一阶段

Try (尝试) : 主要是对业务系统做检测及资源预留 (加锁, 锁住资源)



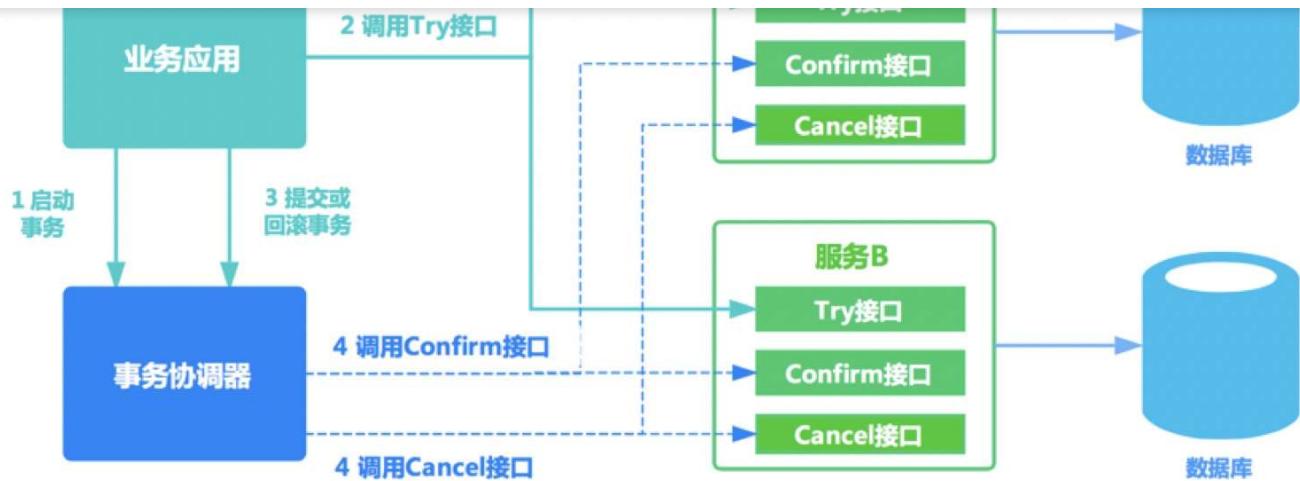
2. 第二阶段

本阶段根据第一阶段的结果，决定是执行confirm还是cancel

Confirm (确认) : 执行真正的业务 执行业务, 释放锁

Cancel (取消) : 是预留资源的取消 出问题, 释放锁





案例

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

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

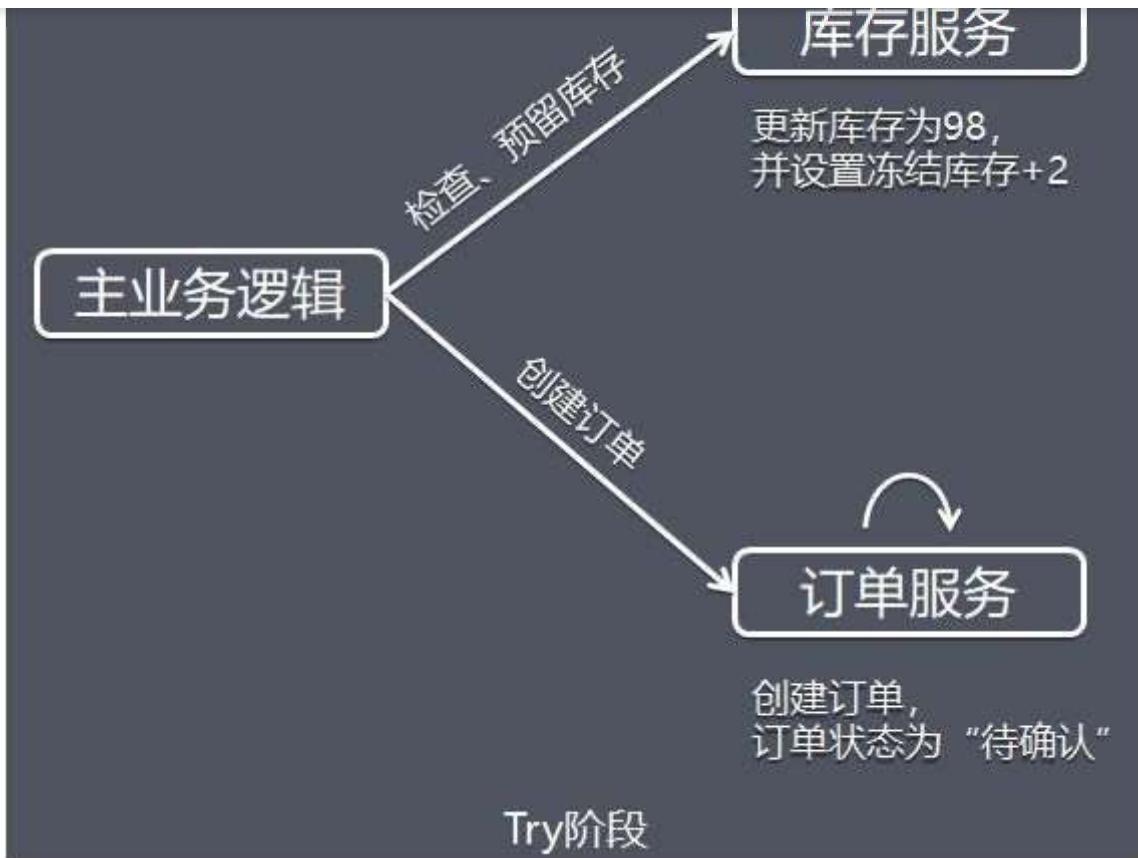
①Try 阶段



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

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



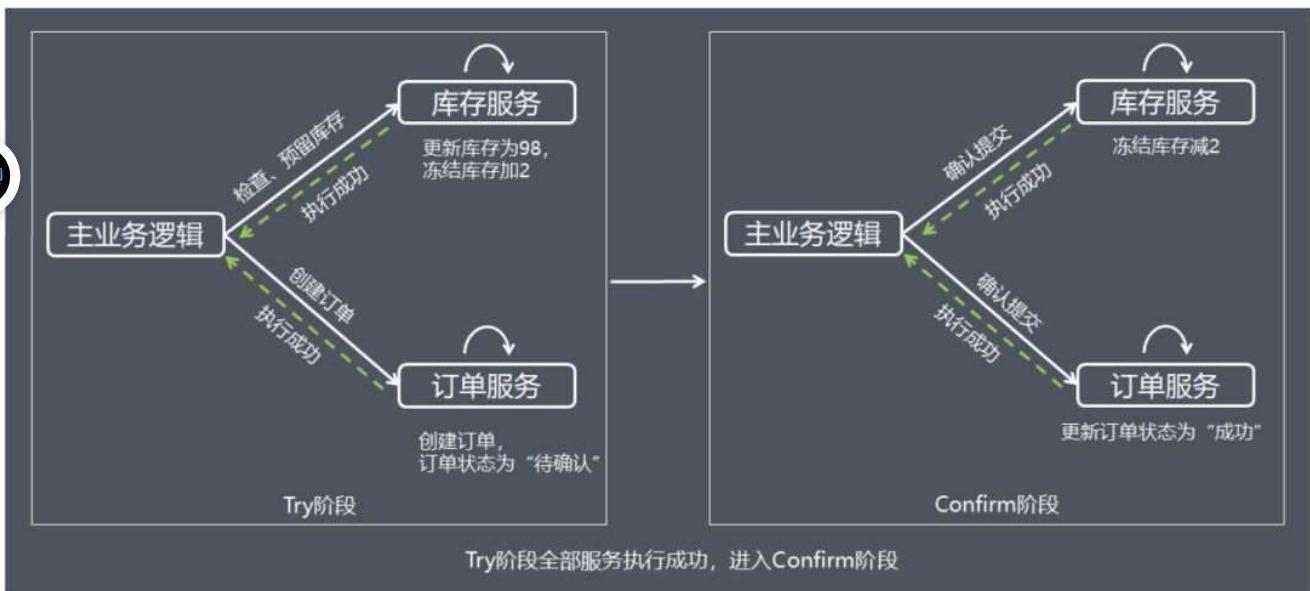


②Confirm / Cancel 阶段

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

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

Confirm：当 Try 阶段服务全部正常执行，执行确认业务逻辑操作

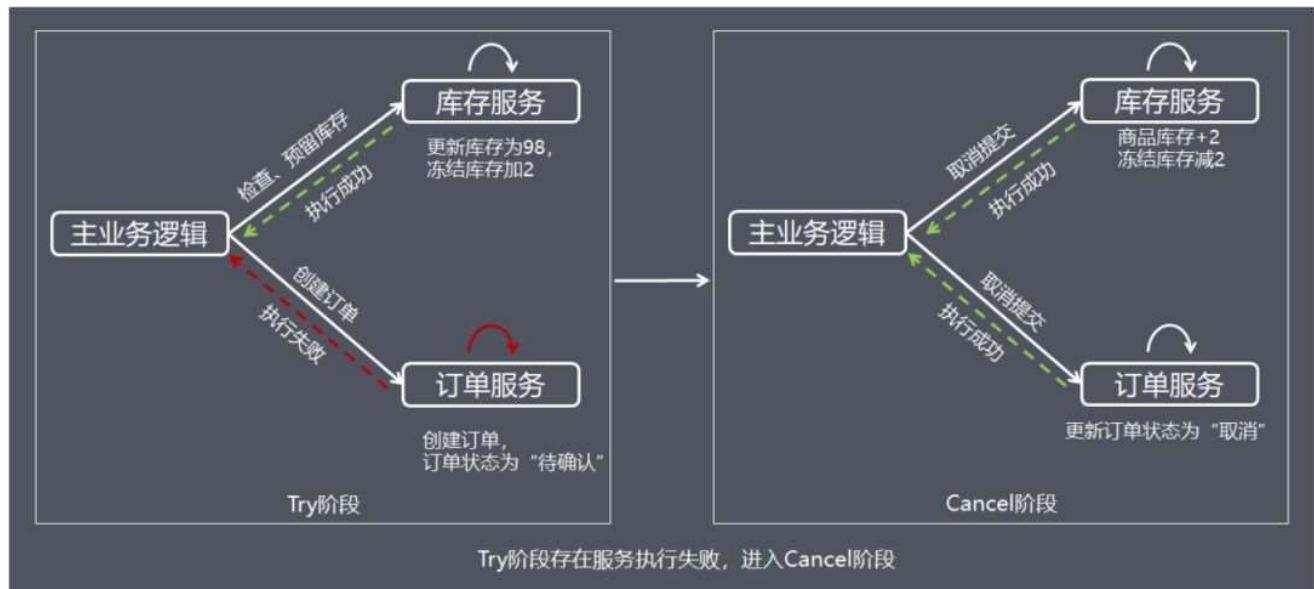


三 君哥的学习笔记



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

Cancel: 当 Try 阶段存在服务执行失败，进入 Cancel 阶段



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



最终一致性保证

- TCC 事务机制以初步操作 (Try) 为中心的，确认操作 (Confirm) 和取消操作 (Cancel) 都是围绕初步操作 (Try) 而展开。因此，Try 阶段中的操作，其保障性是最好的，即使失败，仍然有取消操作 (Cancel) 可以将其执行结果撤销。
- Try阶段执行成功并开始执行 Confirm 阶段时，默认 Confirm 阶段是不会出错的。也就是说只要 Try 成功， Confirm 一定成功 TCC设计之初的定义。
- Confirm与Cancel如果失败，由TCC框架进行==重试==补偿
- 存在极低概率在CC环节彻底失败，则需要定时任务或人工介入



方案总结

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

- 性能提升：具体业务来实现控制资源锁的粒度变小，不会锁定整个资源。
- 数据最终一致性：基于 Confirm 和 Cancel 的幂等性，保证事务最终完成确认或者取消，保证数据的一致性。



三

君哥的学习笔记

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

本地消息表

方案简介

本地消息表的方案最初是由 eBay 提出，核心思路是将分布式事务拆分成本地事务进行处理。

方案通过在事务主动发起方额外新建事务消息表，事务发起方处理业务和记录事务消息在本地事务中完成，轮询事务消息表的数据发送事务消息，事务被动方基于消息中间件消费事务消息表中的事务。

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

处理流程

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

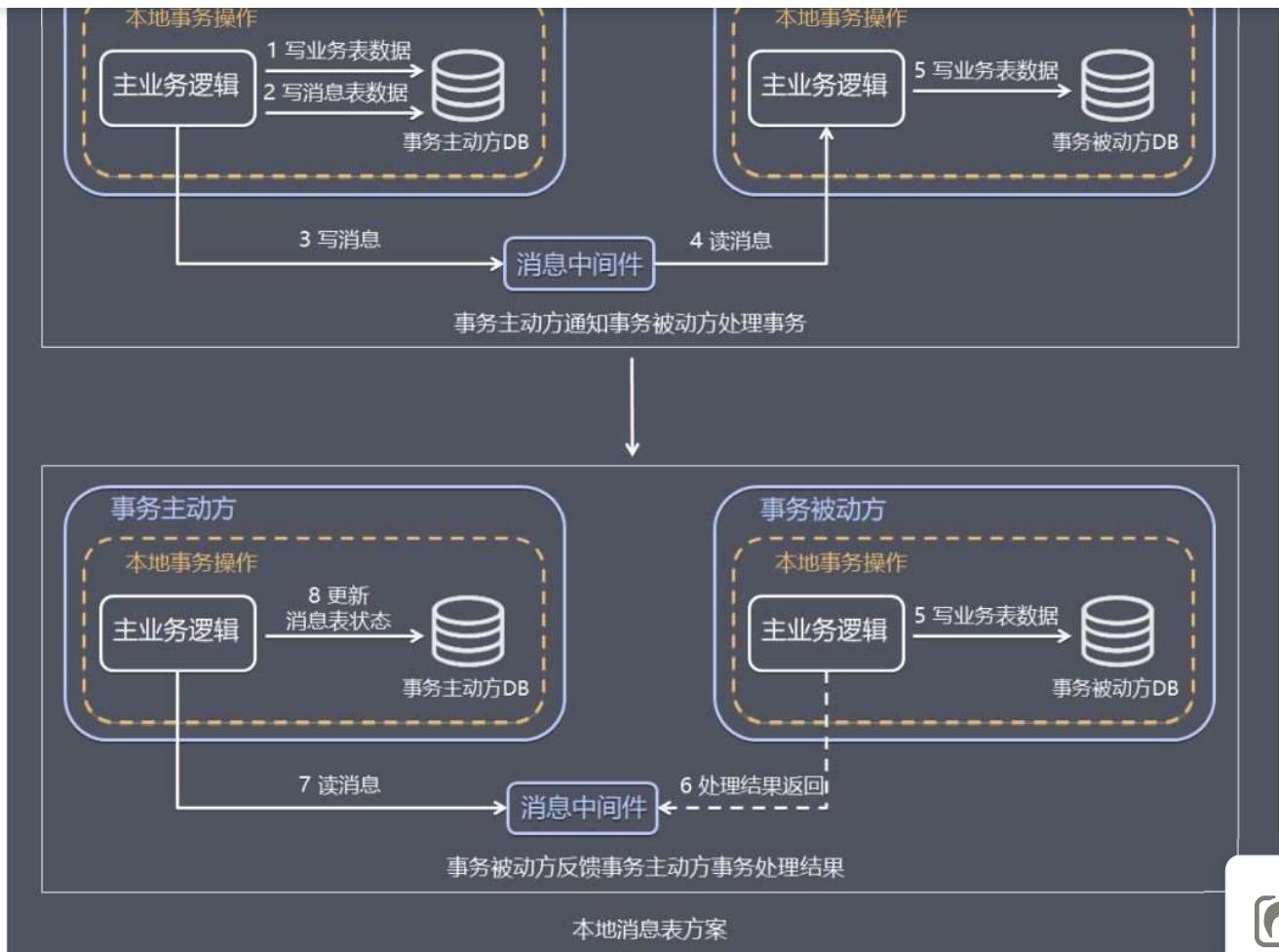
为了方便理解，下面继续以电商下单为例进行方案解析，这里把整个过程简单分为扣减库存，订单创建 2 个步骤。

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

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

整个业务处理流程如下：





步骤1：事务主动方处理本地事务。

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

步骤 2：事务主动方通过消息中间件，通知事务被动方处理事务通知事务待消息。

消息中间件可以基于 Kafka、RocketMQ 消息队列，事务主动方主动写消息到消息队列，事务消费方消费并处理消息队列中的消息。

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

步骤 3：事务被动方通过消息中间件，通知事务主动方事务已处理的消息。

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

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



三 君哥的学习笔记

- 业务主动方发送消息，写入本地消息表，同时向消息中间件发布消息。
- 当步骤 2、步骤 3 处理出错，由于未处理的事务消息还是保存在事务发送方，事务发送方可以定时轮询为超时消息数据，再次发送到消息中间件进行处理。事务被动方消费事务消息重试处理。
- 如果是业务上的失败，事务被动方可以发消息给事务主动方进行回滚。
- 如果多个事务被动方已经消费消息，事务主动方需要回滚事务时需要通知事务被动方回滚。

方案总结

方案的优点如下：

- 从应用设计开发的角度实现了消息数据的可靠性，消息数据的可靠性不依赖于消息中间件，弱化了对 MQ 中间件特性的依赖。
- 方案轻量，容易实现。

缺点如下：

- 与具体的业务场景绑定，耦合性强，不可公用。
- 消息数据与业务数据同库，占用业务系统资源。
- 业务系统在使用关系型数据库的情况下，消息服务性能会受到关系型数据库并发性能的限。



MQ 事务方案

方案简介

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

处理流程



下面主要基于 RocketMQ 4.3 之后的版本介绍 MQ 的分布式事务方案。

在本地消息表方案中，保证事务主动方发写业务表数据和写消息表数据的一致性是基于数据库事务，RocketMQ 的事务消息相对于普通 MQ，相对于提供了 2PC 的提交接口，方案如下：

正常情况：事务主动方发消息

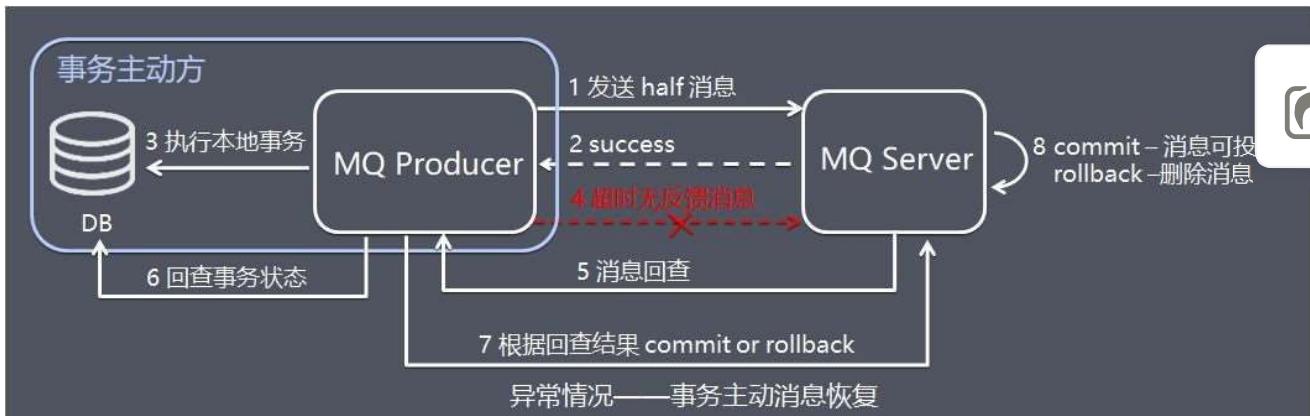
三 君哥的学习笔记



这种情况下，事务主动方服务正常，没有发生故障，发消息流程如下：

- 图中 1：发送方向 MQ 服务端(MQ Server)发送 half 消息。
- 图中 2：MQ Server 将消息持久化成功之后，向发送方 ack 确认消息已经发送成功。
- 图中 3：发送方开始执行本地事务逻辑。
- 图中 4：发送方根据本地事务执行结果向 MQ Server 提交二次确认 (commit 或是 rollback)。
- 图中 5：MQ Server 收到 commit 状态则将半消息标记为可投递，订阅方最终将收到该消息；MQ Server 收到 rollback 状态则删除半消息，订阅方将不会接受该消息。

异常情况：事务主动方消息恢复



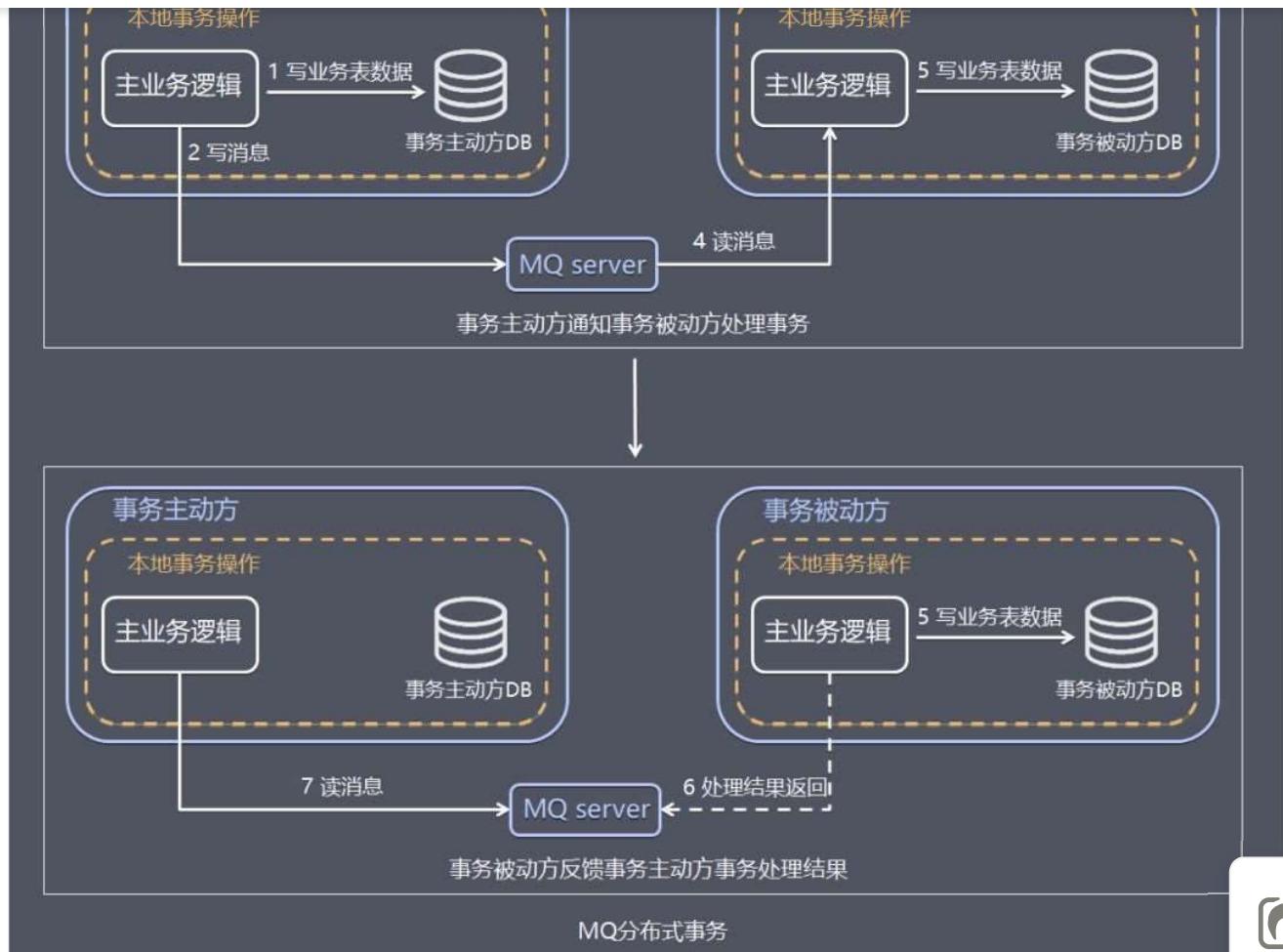
在断网或者应用重启等异常情况下，图中 4 提交的二次确认超时未到达 MQ Server，此时处理逻辑如下：

- 图中 5：MQ Server 对该消息发起消息回查。
- 图中 6：发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
- 图中 7：发送方根据检查得到的本地事务的最终状态再次提交二次确认。
- 图中 8：MQ Server 基于 commit/rollback 对消息进行投递或者删除。

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



君哥的学习笔记



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

如果事务被动方消费消息异常，需要不断重试，业务处理逻辑需要保证幂等。

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

方案总结

相比本地消息表方案，MQ 事务方案优点是：



消息数据独立存储，降低业务系统与消息系统之间的耦合。

- 吞吐量由于使用本地消息表方案。

缺点是：

- 一次消息发送需要两次网络请求(half 消息 + commit/rollback 消息)。
- 业务处理服务需要实现消息状态回查接口。

Saga 事务：最终一致性



三 君哥的学习笔记

Saga 事务长活事务 (Long lived transaction) 论文。

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

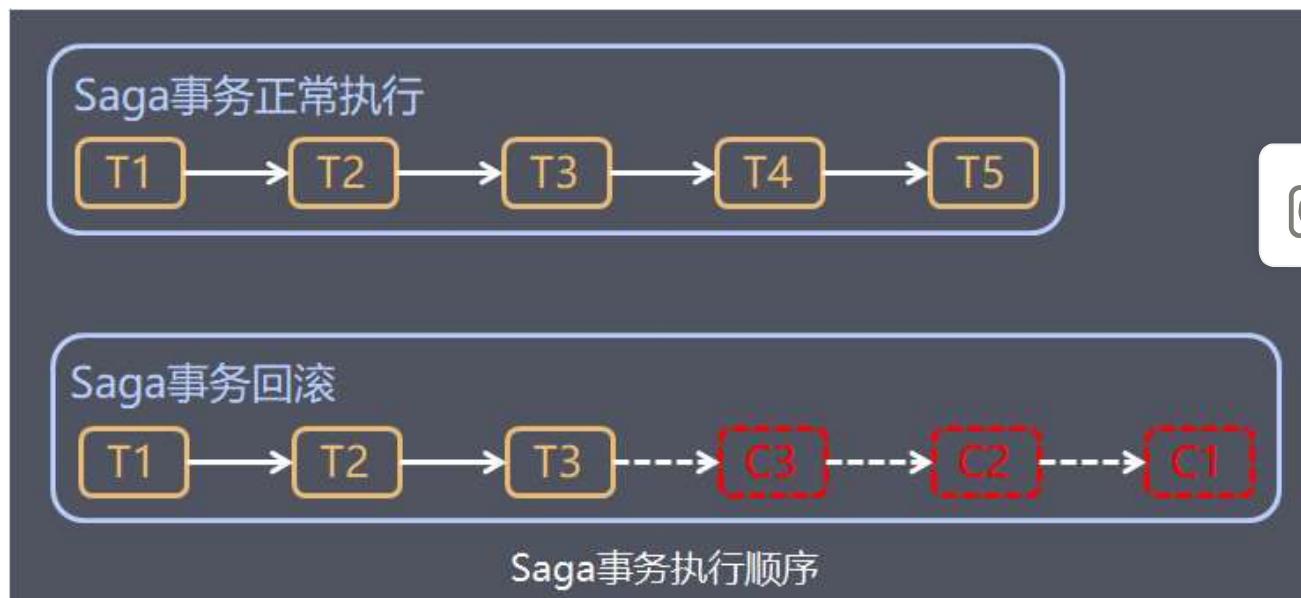
处理流程

Saga 事务基本协议如下：

- 每个 Saga 事务由一系列幂等的有序子事务(sub-transaction) T_i 组成。
- 每个 T_i 都有对应的幂等补偿动作 C_i ，补偿动作用于撤销 T_i 造成的结果。

可以看到，和 TCC 相比，Saga 没有“预留”动作，它的 T_i 就是直接提交到库。

下面以下单流程为例，整个操作包括：创建订单、扣减库存、支付、增加积分。



Saga 的执行顺序有两种，如上图：

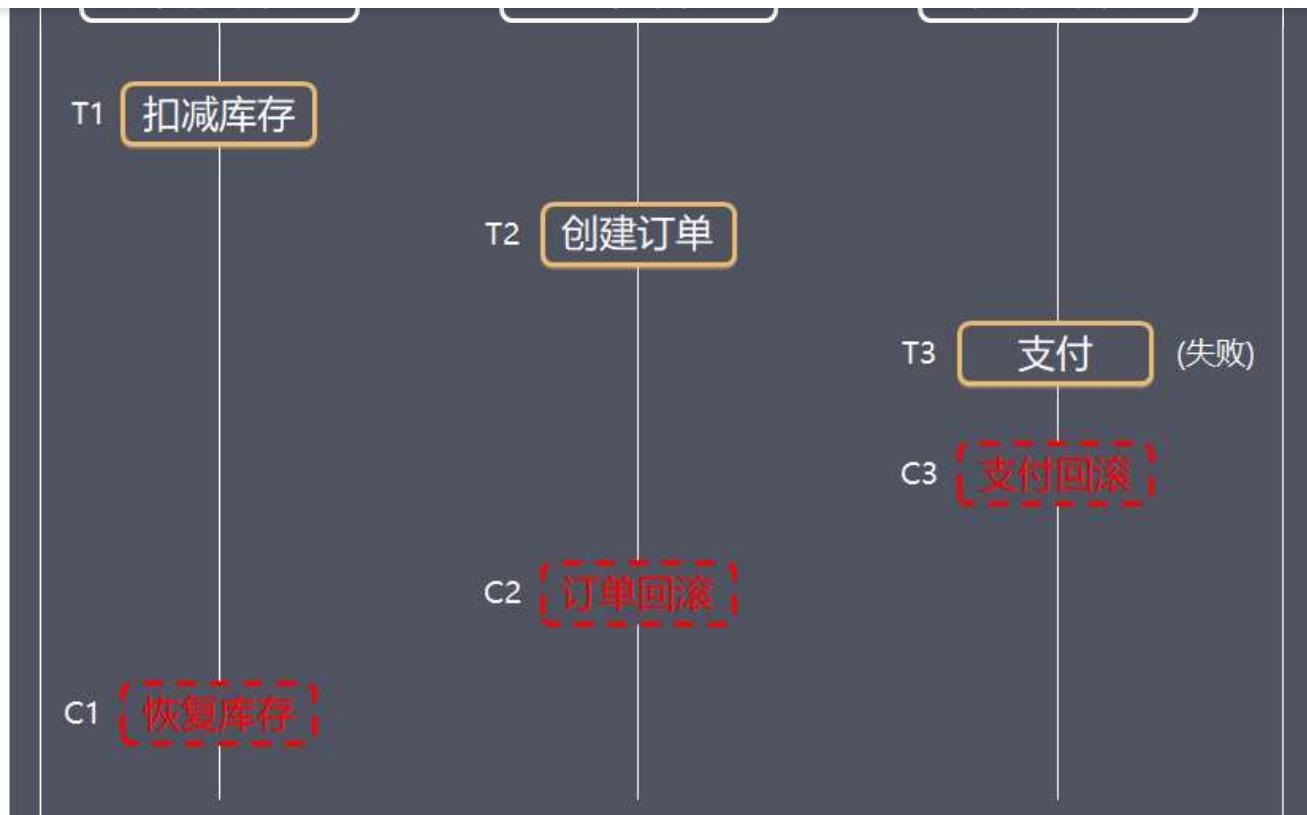
- 事务正常执行完成： $T_1, T_2, T_3, \dots, T_n$ ，例如：扣减库存(T_1)，创建订单(T_2)，支付(T_3)，依次有序完成整个事务。
- 事务回滚： $T_1, T_2, \dots, T_j, C_j, \dots, C_2, C_1$ ，其中 $0 < j < n$ ，例如：扣减库存(T_1)，创建订单(T_2)，支付(T_3 ，支付失败)，支付回滚(C_3)，订单回滚(C_2)，恢复库存(C_1)。

Saga 定义了两种恢复策略：



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





Saga事务向后恢复



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

Saga 事务常见的有两种不同的实现方式：

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

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





以电商订单的例子为例：

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

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

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

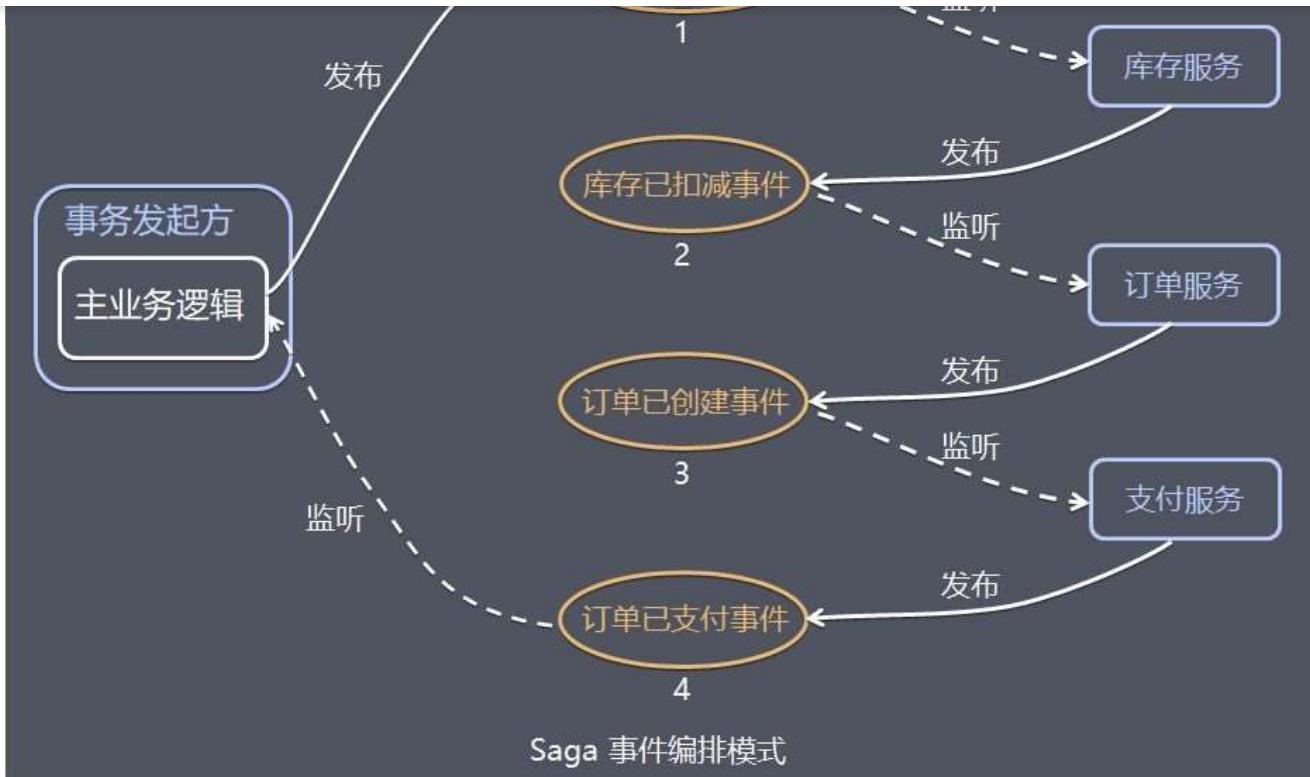
②事件编排 (Event Choreography)：没有中央协调器 (没有单点风险) 时，每个服务产生并观察其他服务的事件，并决定是否应采取行动。

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

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



三 君哥的学习笔记



以电商订单的例子为例：

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

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

方案总结

命令协调设计的优点如下：

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

命令协调设计缺点如下：



三 君哥的学习笔记

事件/编排设计优点如下：

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

事件/编排设计缺点如下：

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

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

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

总结

各方案使用场景

	2PC	3PC	TCC	本地消息表	MQ 事务	Saga
数据一致性	强	强	弱	弱	弱	弱
容错性	低	低	高	高	高	高
复杂性	中	高	高	低	低	中
性能	低	低	中	中	高	中
维护成本	低	中	高	中	中	高

方案比较

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

- 2PC/3PC：依赖于数据库，能够很好的提供强一致性和强事务性，但相对来说延迟比较高，比较适合传统的单体应用，在同一个方法中存在跨库操作的情况，不适合高并发和高性能要求的场景。
- TCC：适用于执行时间确定且较短，实时性要求高，对数据一致性要求高，比如互联网金融企业最核心的三个服务：交易、支付、账务。



三 君哥的学习笔记

账/校验系统兜底。

- Saga 事务：由于 Saga 事务不能保证隔离性，需要在业务层控制并发，适合于业务场景事务并发操作同一资源较少的情况。Saga 相比缺少预提交动作，导致补偿动作的实现比较麻烦，例如业务是发送短信，补偿动作则得再发送一次短信说明撤销，用户体验比较差。Saga 事务较适用于补偿动作容易处理的场景。

分布式事务方案设计

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

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

世界上解决一个计算机问题最简单的方法：“恰好”不需要解决它！

本地事务	两阶段提交	柔性事务	
业务改造	无	无	实现相关接口
一致性	不支持	支持	最终一致
隔离性	不支持	支持	业务方保证
并发性能	无影响	严重衰退	略微衰退
适合场景	业务方处理不一致	短事务 & 低并发	长事务 & 高并发



总结

ACID



原子性、一致性、隔离性、持久性

CAP

- 一致性：数据在多个副本直接保持一致性，一个副本的更新成功，其他副本也必须更新成功，此特性要求客户端可获取到最新的数据
- 可用性：系统在合理时间内返回
- 分区容忍性：在分布式网络遇到分区故障时，依然可以提供满足一致性和可用性的服务



君哥的学习笔记

BASE

基本可用、软状态、最终一致性，采用合适的方式最终一致

分布式事务协议

- 2PC
 - 优点：实现简单、原理简单
 - 缺点：数据不一致，脑裂、单点问题、同步阻塞
- 3PC
 - 优点：减小了参与者阻塞的范围；出现单点问题后，仍然可以提交
 - 缺点：数据不一致

分布式事务解决方案

1. TCC
2. 本地消息表
3. MQ事务
4. saga



第二章：Seata基础

Seata简介



[最新活动 NEW](#) [产品分类](#) [企业应用中心](#) [解决方案](#) [云市场](#) [支持与服务](#) [合作伙伴与生态](#) [开发者](#) [云栖号](#) [了解阿里云](#)

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。

- 对业务无侵入：即减少技术架构上的微服务化所带来的分布式事务问题对业务的侵入
- 高性能：减少分布式事务解决方案所带来的性能消耗

官方站点：

- 源码：<https://github.com/seata/seata>
- 文档：<http://seata.io/zh-cn/index.html>



三 君哥的学习笔记



These are only part of the companies using Seata, for reference only. If you are using Seata, please add your company here to tell us your scenario to make Seata better.



Seata术语

TC (Transaction Coordinator) 事务协调者：维护全局和分支事务的状态，驱动全局事务提交或回滚。

TM (Transaction Manager) 事务管理器：定义全局事务的范围：开始全局事务、提交或回滚全局事务。

RM (Resource Manager) 资源管理器：管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。





```
class Bussiness{
    会员采购(){
        orderService.创建订单();
        storageService.扣减库存();
    }
}
```

TM

```
} } //订单表新增undo_log  
orderDao.insert();  
} } RM
```

```
class StorageService(){
    扣减库存(){
        //库存表新增undo_log
        storageDao.update();
    }
}
```

RM

Seata Server 1.3.0 (TC)

UNDO_LOG表: 回滚日志

1. UNDO_LOG 必须在每个业务数据库中创建，用于保存回滚操作数据
2. 当全局提交时， UNDO_LOG 记录直接删除
3. 当全局回滚时，将现有数据撤销，还原至操作前的状态，详见beforeImage和afterImage



```
1 CREATE TABLE `undo_log` (
2     `id` bigint(20) NOT NULL AUTO_INCREMENT,
3     `branch_id` bigint(20) NOT NULL,
4     `xid` varchar(100) NOT NULL,
5     `context` varchar(128) NOT NULL,
6     `rollback_info` longblob NOT NULL,
7     `log_status` int(11) NOT NULL,
8     `log_created` datetime NOT NULL,
9     `log_modified` datetime NOT NULL,
10    `ext` varchar(100) DEFAULT NULL,
11    PRIMARY KEY (`id`),
12    UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
13 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```



第三章：AT模式详解



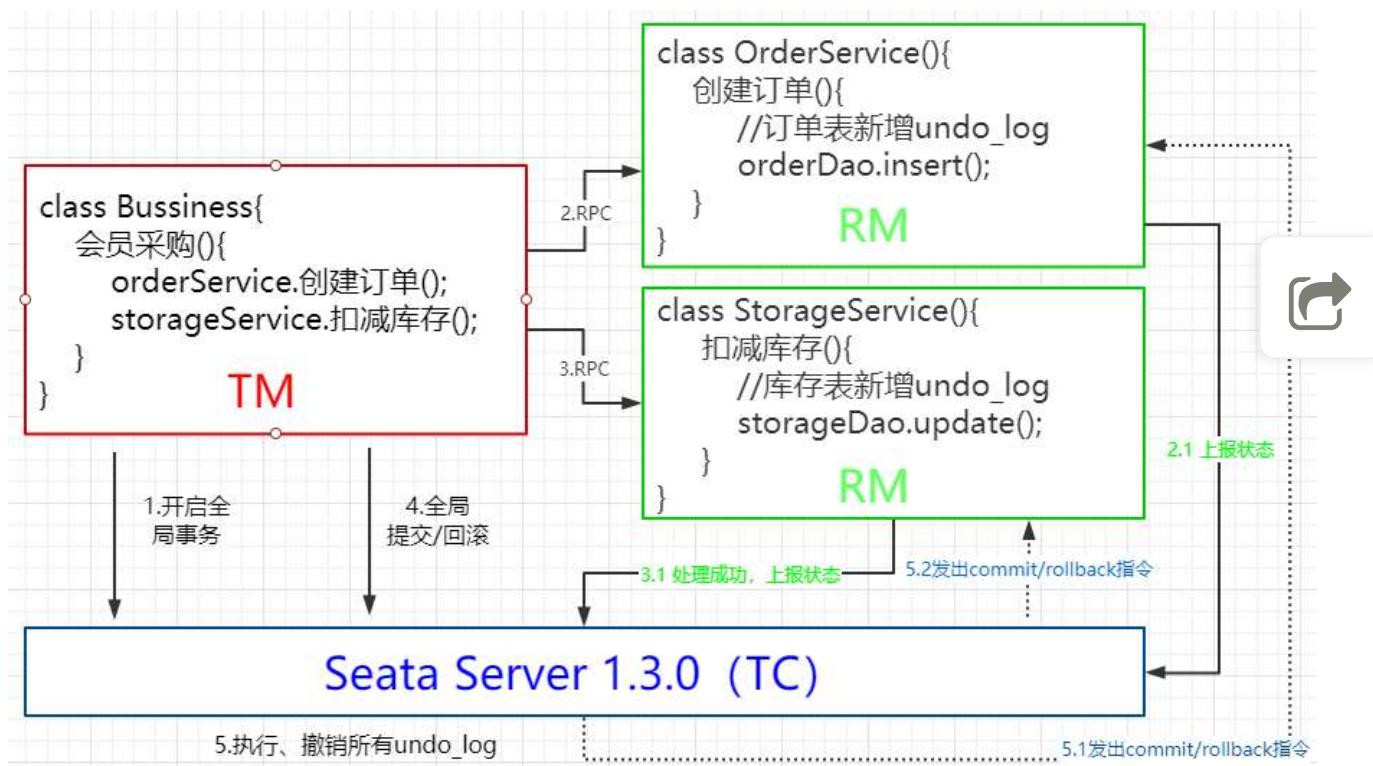
君哥的学习笔记

AT模式运行机制

AT模式的特点就是对业务无入侵式，整体机制分二阶段提交

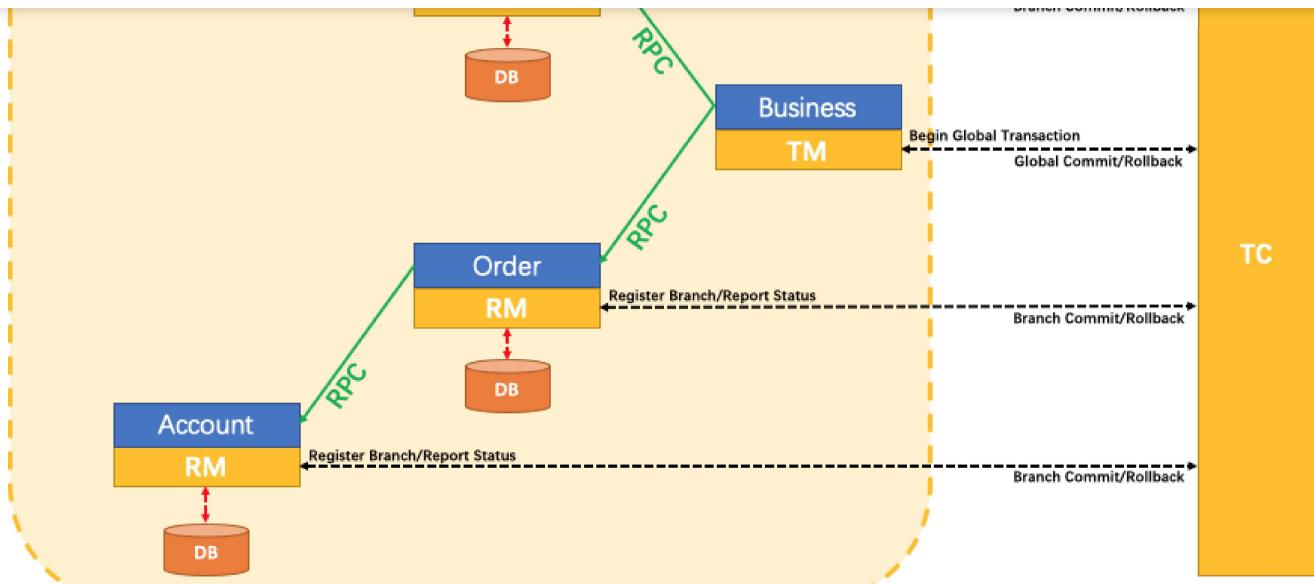
- 两阶段提交协议的演变：
 - 一阶段：业务数据和回滚日志记录在同一个本地事务中提交，释放本地锁和连接资源。
 - 二阶段：
 - 提交异步化，非常快速地完成。
 - 回滚通过一阶段的回滚日志进行反向补偿。

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。



Seata具体实现步骤

1. TM端使用 `@GlobalTransaction` 进行全局事务开启、提交、回滚
2. TM开始RPC调用远程服务
3. RM端 `seata-client` 通过扩展 `DataSourceProxy`，实现自动生成 `UNDO_LOG` 与 `TC` 上报
4. TM告知TC提交/回滚全局事务
5. TC通知RM各自执行 `commit/rollback` 操作，同时清除 `undo_log`



搭建Seata TC 协调者

1. 点击此处 [下载最新版本，并解压缩](#)



2. 修改 conf/file.conf 文件

- o 将 mode="file" 改为 mode="db"





君哥的学习笔记

```

3  store {
4      ## store mode: file、db、redis
5      mode = "db" 1 表示直接走数据库
6
7      ## file store property
8      file {
9          ## store location dir
10         dir = "sessionStore"
11         # branch session size , if exceeded first try compress lock
12         maxBranchSessionSize = 16384
13         # globe session size , if exceeded throws exceptions
14         maxGlobalSessionSize = 512
15         # file buffer size , if exceeded allocate new buffer
16         fileWriteBufferCacheSize = 16384
17         # when recover batch read size
18         sessionReloadReadSize = 100
19         # async, sync
20         flushDiskMode = async

```

- o db部分配置mysql相关信息

```

1  db {
2      DruidDataSource(druid)/BasicDataSource(dbcp)/HikariDataSource(hikar
3      datasource = "druid"
4      dbType = "mysql"
5      //注意如果是mysql8版本，此处采用cj驱动，url后缀需要加serverTimezone=Asia/
6      driverClassName = "com.mysql.cj.jdbc.Driver"
7      url = "jdbc:mysql://rm-bp17dq6iz79761b8fxo.mysql.rds.aliyuncs.com:3306/
8      //用户名和密码
9      user = "seata_test"
10     password = "- seata1234abcd!"
11     minConn = 5
12     maxConn = 30
13     globalTable = "global_table"
14     branchTable = "branch_table"
15     lockTable = "lock_table"
16     queryLimit = 100
17     maxWait = 5000
18 }

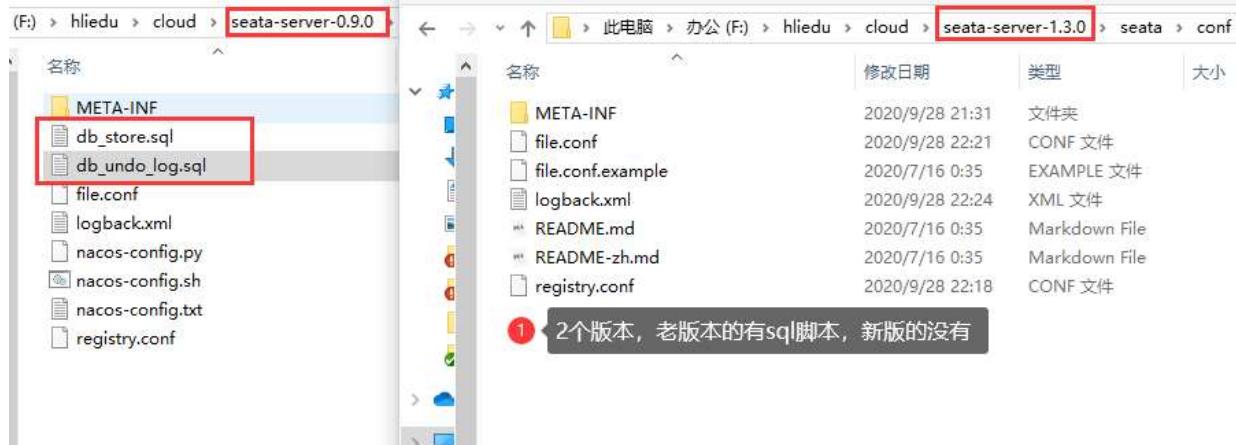
```

- o 数据库对应上述配置新建 seata 的DB，并分配给指定的用户权限



- > 函数
- > 事件
- > 查询
- > 报表
- > 备份

- 初始化table，注意此处一定要初始化，并且1.0之后的版本没有sql脚本，这里我贴出来



```

1 -- the table to store GlobalSession data
2 drop table if exists `global_table`;
3 create table `global_table` (
4     `xid` varchar(128) not null,
5     `transaction_id` bigint,
6     `status` tinyint not null,
7     `application_id` varchar(32),
8     `transaction_service_group` varchar(32),
9     `transaction_name` varchar(128),
10    `timeout` int,
11    `begin_time` bigint,
12    `application_data` varchar(2000),
13    `gmt_create` datetime,
14    `gmt_modified` datetime,
15    primary key (`xid`),
16    key `idx_gmt_modified_status` (`gmt_modified`, `status`),
17    key `idx_transaction_id` (`transaction_id`)
18);
19
20 -- the table to store BranchSession data
21 drop table if exists `branch_table`;
22 create table `branch_table` (
23     `branch_id` bigint not null,
24     `xid` varchar(128) not null,
25     `transaction_id` bigint ,

```

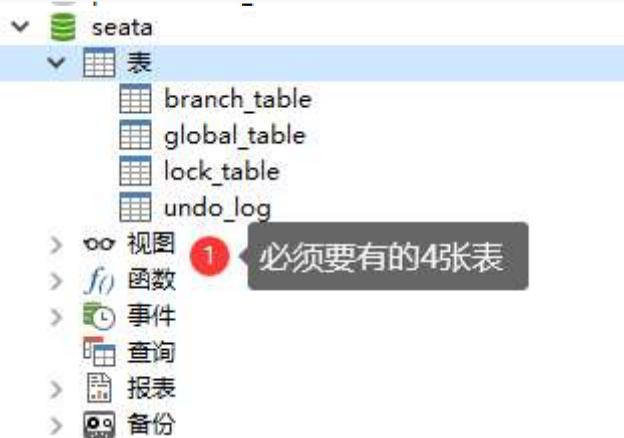


```

26
27
28
29   `branch_type` varchar(8) ,
30   `status` tinyint,
31   `client_id` varchar(64),
32   `application_data` varchar(2000),
33   `gmt_create` datetime,
34   `gmt_modified` datetime,
35   primary key (`branch_id`),
36   key `idx_xid` (`xid`)
37 );
38
39 -- the table to store lock data
40 drop table if exists `lock_table`;
41 create table `lock_table` (
42   `row_key` varchar(128) not null,
43   `xid` varchar(96),
44   `transaction_id` long ,
45   `branch_id` long,
46   `resource_id` varchar(256) ,
47   `table_name` varchar(32) ,
48   `pk` varchar(36) ,
49   `gmt_create` datetime ,
50   `gmt_modified` datetime,
51   primary key(`row_key`)
52 );
53
54 -- the table to store seata xid data
55 -- 0.7.0+ add context
56 -- you must to init this sql for you business databese. the seata server nc
57 -- 此脚本必须初始化在你当前的业务数据库中，用于AT 模式XID记录。与server端无关（
58 -- 注意此处0.3.0+ 增加唯一索引 ux_undo_log
59 drop table `undo_log`;
60 CREATE TABLE `undo_log` (
61   `id` bigint(20) NOT NULL AUTO_INCREMENT,
62   `branch_id` bigint(20) NOT NULL,
63   `xid` varchar(100) NOT NULL,
64   `context` varchar(128) NOT NULL,
65   `rollback_info` longblob NOT NULL,
66   `log_status` int(11) NOT NULL,
67   `log_created` datetime NOT NULL,
68   `log_modified` datetime NOT NULL,
69   `ext` varchar(100) DEFAULT NULL,
70   PRIMARY KEY (`id`),
71   UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
72 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

```





3. 修改 registry.conf 文件

这个文件内容分为 registry 和 config 2部分，为了方便，我们使用之前学过的 nacos 注册中心，你也可以采用默认的 file 方式

- 将 registry 和 config 部分的由 type="file" 都换成 type="nacos" (注意是2处地方)
- 将2处的 nacos 配置根据实际情况需要调整参数，本机启动 nacos 则不需要调整

```

1 registry {
2     # file、nacos、eureka、redis、zk、cor
3     type = "nacos"          registry部分
4
5 nacos {
6     application = "seata-server"
7     serverAddr = "127.0.0.1:8848"
8     group = "SEATA_GROUP"
9     namespace = ""
10    cluster = "default"
11    username = ""
12    password = ""
13 }
```





君哥的学习笔记

```

58 type = "file"
59
60 日 nacos {
61     serverAddr = "127.0.0.1:8848"
62     namespace = ""
63     group = "SEATA_GROUP"
64     username = ""
65     password = ""
66 }
67 日 consul {
68     serverAddr = "127.0.0.1:8500"
69 }

```

config部分

```

1 nacos {
2     application = "seata-server"
3     serverAddr = "nacos.it235.com:80"
4     group = "SEATA_GROUP"
5     namespace = "343f2aa2-1a42-43ea-b078-33ab7d58bd6a"
6     cluster = "default"
7     username = "nacos"
8     password = "nacos"
9 }

```



4. 修改 conf/logback.xml 的文件 (可选操作)

将 \${user.home} 改为具体的 seata 目录, 我这里是 F:\hliedu\cloud\seata-server-1.3.0\seata , 那么配置如下

```

1 <property name="LOG_HOME" value="F:\hliedu\cloud\seata-server-1.3.0\seata\log"
```



5. 双击启动 bin/seata-server.bat 文件启动服务 (有可能出现闪退)

6. 为了方便查看更加清晰的日志, 建议使用 cmd 的方式启动

- 打开CMD窗口, 使用cd命令切换到 seata 的 bin 目录
- 输入 seata-server.bat 回车

7. 启动成功的结果如下

君哥的学习笔记



```
[0T]
[23:06:17, 032 |-INFO in ch.qos.logback.core.joran.action.AppenderRefAction - Attaching appender named [ERI
OOT]
[23:06:17, 032 |-INFO in ch.qos.logback.core.joran.action.AppenderRefAction - Attaching appender named [CO
[ROOT]
[23:06:17, 032 |-INFO in ch.qos.logback.classic.joran.action.ConfigurationAction - End of configuration.
[23:06:17, 033 |-INFO in ch.qos.logback.classic.joran.JoranConfigurator@29774679 - Registering current con
fe fallback point

[2m2020-09-28 23:06:17.168 [0;39m [32m INFO [0;39m [2m-- [0;39m [2m[           main] [0;39m [36mio.
1eConfiguration [0;39m [2m: [0;39m The configuration file used is registry.conf
[2m2020-09-28 23:06:18.837 [0;39m [32m INFO [0;39m [2m-- [0;39m [2m[           main] [0;39m [36m..;
NettyServerBootstrap [0;39m [2m: [0;39m Server started, listen port: 8091
```

使用 Nacos 作为Seata的配置中心

思考

如果不使用 Nacos，那Seata的默认配置中心在哪？

我们编辑 `registry.conf` 文件，找到 `config` 部分，内容如下：

```
1 config {
2     # file、nacos、apollo、zk、consul、etcd3
3     type = "file"
4
5     nacos {
6         .....
7     }
8     consul {
9         .....
10    }
11    apollo {
12        .....
13    }
14    zk {
15        .....
16    }
17    etcd3 {
18        .....
19    }
20    -- 重点
21    file {
22        name = "file.conf"
23    }
24 }
```



君哥的学习笔记



为配置中心，我们需要按照以下步骤进行操作。

1. 保证你的 Nacos 服务已经启动，且可正常连接，对Nacos不熟悉的同学，可以点击[此处进入学习](#)
2. 添加命名空间seata，获取到 namespace 的id，我这里是 f46bbdaa-f11e-414f-9530-e6a18cbf91f6

命名空间名称	命名空间ID
public(保留空间)	
prod	343f2aa2-1a42-43ea-b078-33ab7d58bd6a
seata	f46bbdaa-f11e-414f-9530-e6a18cbf91f6
knife-boot-dev	knife-boot-dev

3. 修改 registry.conf 文件中的 registry 部分type = "nacos"，同时修改 nacos 部分的配置

```

1  nacos {
2      application = "seata-server"
3      serverAddr = "nacos.it235.com:80"
4      group = "SEATA_GROUP"
5      namespace = "f46bbdaa-f11e-414f-9530-e6a18cbf91f6"
6      cluster = "default"
7      username = "nacos"
8      password = "nacos"
9  }

```

4. 修改 registry.conf 文件中的 config 部分type = "nacos"，同时修改 nacos 部分的配置



```

3     namespace = "f46bbdaa-f11e-414f-9530-e6a18cbf91f6"
4     group = "SEATA_GROUP"
5     username = "nacos"
6     password = "nacos"
7 }
```

5. 下载 nacos-config 脚本 和 config.txt , 点击进入下载页

- **nacos-config.sh** 和 **nacos-config.py** 选择一个：在 seata 目录下新建 script 目录，将 nacos-config.sh 放入 script 目录下
- config.txt : 该文件存放在将 seata 目录下，与 conf、lib 目录同级，seata 的非常全的配置内容，可通过 nacos-config.sh 脚本推送到 nacos 配置中心
- 修改 config.txt 的内容，下述 1 处暂时无需修改，2 处需要换成你的 DB 连接路径

```

11 transport.threadFactory.clientWorkerThreadPrefix=NettyClientWorkerThread
12 transport.threadFactory.bossThreadSize=1
13 transport.threadFactory.workerThreadSize=default
14 transport.shutdown.wait=3
15 service.vgroupMapping.order-service-tx-group=default
16 service.vgroupMapping.storage-service-tx-group=default
17 service.vgroupMapping.business-service-tx-group=default
18 service.default.groupList=127.0.0.1:8091
19 service.enableDegrade=false
20 service.disableGlobalTransaction=false
21 client.rm.asyncCommitBufferLimit=10000
22 client.rm.lock.retryInterval=10
23 client.rm.lock.retryTimes=30
24 client.rm.lock.retryPolicyBranchRollbackOnConflict=true
25 client.rm.reportRetryCount=5
26 client.rm.tableMetaCheckEnable=false
27 client.rm.tableMetaCheckerInterval=60000
28 client.rm.sqlParserType=druid
29 client.rm.reportSuccessEnable=false
30 client.rm.sagaBranchRegisterEnable=false
31 client.tm.commitRetryCount=5
32 client.tm.rollbackRetryCount=5
33 client.tm.defaultGlobalTransactionTimeout=60000
34 client.tm.degradeCheck=false
35 client.tm.degradeCheckAllowTimes=10
36 client.tm.degradeCheckPeriod=2000
37 store.mode=db
38 store.publicKey=
39 store.file.dir=file_store/data
40 store.file.maxBranchSessionSize=16384
41 store.file.maxGlobalSessionSize=512
42 store.file.writeFileBufferCacheSize=16384
43 store.file.flushDiskMode=async
44 store.file.sessionReloadReadSize=100
45 store.db.dataSource=druid
46 store.db.driverClassName=mysql
47 store.db.driverClassName=com.mysql.cj.jdbc.Driver
48 store.db.url=jdbc:mysql://rm-bp17dq6iz79761b8fxo.mysql.rds.aliyuncs.com:3306/seata?serverTimezone=Asia/Shanghai
49 store.db.user=seata_test
50 store.db.password=seata1234abcd!
51 store.db.minConn=5
52 store.db.maxConn=30

```

① 存储模式改为 db

② mysql 连接改为你自己的数据库

③ seata 全局事务分组，此处非常重要，在下一节课项目中有使用

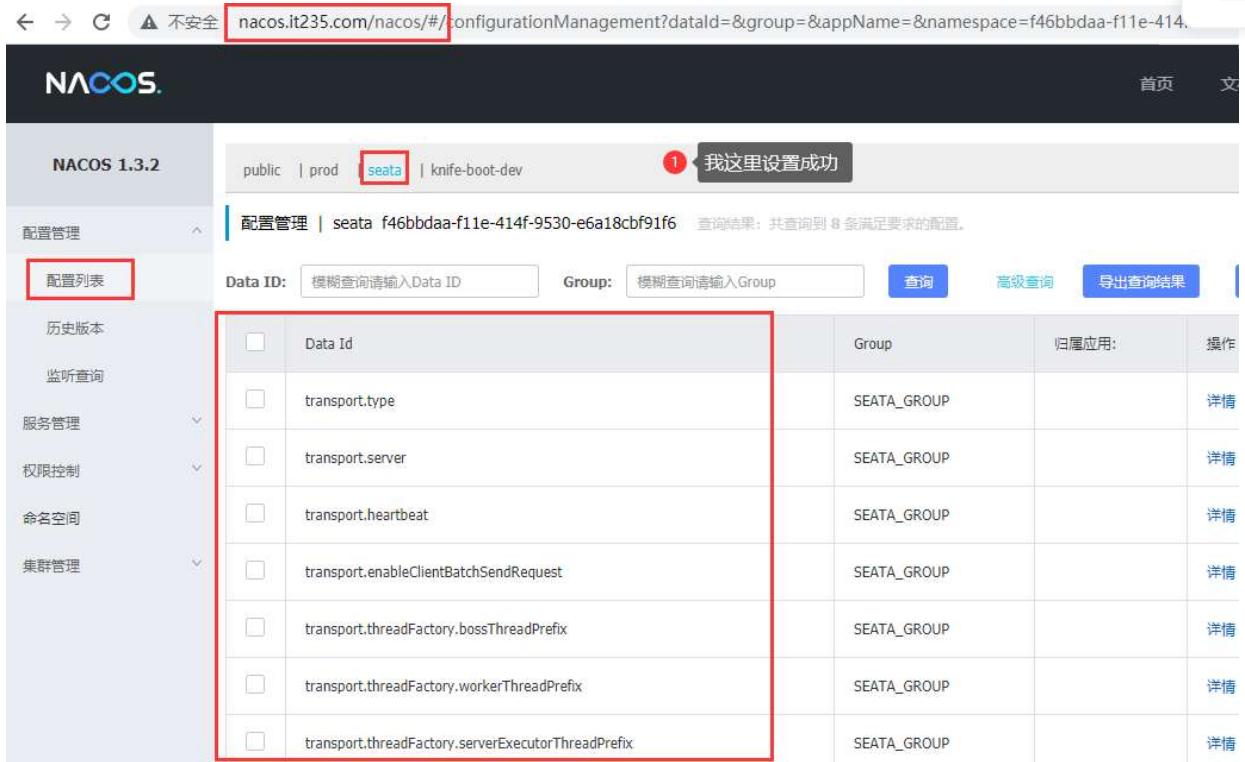
- 打开 git bash 或 linux 类命令行，执行 sh 脚本（注意脚本是否有执行的权限）



3 \$ sh nacos-config.sh -h nacos.it235.com -p 80 -g SEATA_GROUP -t f46bbdaa-f11e-414f-9530-e6a18cbf91f6

```
MINGW64:/f/work/cloud/seata-server-1.3.0/seata/script
opuseradmin@VE2G17IXC9C3D4L MINGW64 /f/work/cloud/seata-server-1.3.0/seata/script^
t
$ sh nacos-config.sh -h nacos.it235.com -p 80 -g SEATA_GROUP -t f46bbdaa-f11e-414f-9530-e6a18cbf91f6
set nacosAddr=nacos.it235.com:80
set group=SEATA_GROUP
Set transport.type=TCP successfully
Set transport.server=NIO successfully
Set transport.heartbeat=true successfully
Set transport.enableClientBatchSendRequest=false successfully
Set transport.threadFactory.bossThreadPrefix=NettyBoss successfully
Set transport.threadFactory.workerThreadPrefix=NettyServerNIOWorker successfully
Set transport.threadFactory.serverExecutorThreadPrefix=NettyServerBizHandler successfully
Set transport.threadFactory.shareBossWorker=false successfully
Set transport.threadFactory.clientSelectorThreadPrefix=NettyClientSelector successfully
Set transport.threadFactory.clientSelectorThreadSize=1 successfully
Set transport.threadFactory.clientWorkerThreadPrefix=NettyClientWorkerThread successfully
Set transport.threadFactory.bossThreadSize=1 successfully
Set transport.threadFactory.workerThreadSize=default successfully
Set transport.shutdown.wait=3 successfully
```

① 表示连接并设置成功
-h nacos服务，你可以设置为localhost
-p nacos端口，你可以设置为8848
-t 命名空间，可以不使用该参数，默认public
-u nacos用户名， -w nacos密码



The screenshot shows the Nacos configuration management interface. On the left, there's a sidebar with tabs for 'NACOS 1.3.2', '配置管理' (Configuration Management), '历史版本' (History Versions), '监听查询' (Monitoring Query), '服务管理' (Service Management), '权限控制' (Permission Control), '命名空间' (Namespace), and '集群管理' (Cluster Management). The '配置管理' tab is selected and highlighted with a red box.

In the main area, there's a breadcrumb navigation bar: 'public' > 'prod' > 'seata' > 'knife-boot-dev'. To the right of the breadcrumb, a message says '① 我这里设置成功' (I have set it successfully).

Below the breadcrumb, there are search and filter fields: 'Data ID:' (模糊查询请输入Data ID) and 'Group:' (模糊查询请输入Group). There are also buttons for '查询' (Query), '高级查询' (Advanced Query), and '导出查询结果' (Export Query Results).

The main content area displays a table of configuration items:

	Data Id	Group	归属应用:	操作
<input type="checkbox"/>	transport.type	SEATA_GROUP		详情
<input type="checkbox"/>	transport.server	SEATA_GROUP		详情
<input type="checkbox"/>	transport.heartbeat	SEATA_GROUP		详情
<input type="checkbox"/>	transport.enableClientBatchSendRequest	SEATA_GROUP		详情
<input type="checkbox"/>	transport.threadFactory.bossThreadPrefix	SEATA_GROUP		详情
<input type="checkbox"/>	transport.threadFactory.workerThreadPrefix	SEATA_GROUP		详情
<input type="checkbox"/>	transport.threadFactory.serverExecutorThreadPrefix	SEATA_GROUP		详情

- 启动 seata-server.bat，并查看 nacos 服务，若seata服务注册成功，表示注册中心和配置中心成功



服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值
seata-server	SEATA_GROUP	1	1	1	false

编写AT模式代码

[源代码地址](#)

RM实现

1. 创建订单和库存服务的DB



```

1  -- 库存服务DB执行
2  CREATE TABLE `tab_storage` (
3      `id` bigint(11) NOT NULL AUTO_INCREMENT,
4      `product_id` bigint(11) DEFAULT NULL COMMENT '产品id',
5      `total` int(11) DEFAULT NULL COMMENT '总库存',
6      `used` int(11) DEFAULT NULL COMMENT '已用库存',
7      PRIMARY KEY (`id`)
8  ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
9  INSERT INTO `tab_storage` (`product_id`, `total`, `used`)VALUES ('1', '96', '4'
10 INSERT INTO `tab_storage` (`product_id`, `total`, `used`)VALUES ('2', '100', '0
11
12  -- 订单服务DB执行
13  CREATE TABLE `tab_order` (
14      `id` bigint(11) NOT NULL AUTO_INCREMENT,
15      `user_id` bigint(11) DEFAULT NULL COMMENT '用户id',
16      `product_id` bigint(11) DEFAULT NULL COMMENT '产品id',
17      `count` int(11) DEFAULT NULL COMMENT '数量',
18      `money` decimal(11,0) DEFAULT NULL COMMENT '金额',
19      `status` int(1) DEFAULT NULL COMMENT '订单状态: 0: 创建中; 1: 已完成',
20      PRIMARY KEY (`id`)
21  ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

```





```

1 CREATE TABLE `undo_log` (
2     `id` bigint(20) NOT NULL AUTO_INCREMENT,
3     `branch_id` bigint(20) NOT NULL,
4     `xid` varchar(100) NOT NULL,
5     `context` varchar(128) NOT NULL,
6     `rollback_info` longblob NOT NULL,
7     `log_status` int(11) NOT NULL,
8     `log_created` datetime NOT NULL,
9     `log_modified` datetime NOT NULL,
10    `ext` varchar(100) DEFAULT NULL,
11    PRIMARY KEY (`id`),
12    UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
13 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

```

3. 添加 seata pom.xml 依赖

```

1 <parent>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-parent</artifactId>
4     <version>2.3.2.RELEASE</version>
5     <relativePath/>
6 </parent>
7
8     .....
9
10    <properties>
11        <springboot.verison>2.4.2.RELEASE</springboot.verison>
12        <java.version>1.8</java.version>
13        <mybatis.version>2.1.5</mybatis.version>
14        <tk-mapper.version>4.1.5</tk-mapper.version>
15        <seata.version>1.3.0</seata.version>
16    </properties>
17
18     .....
19
20     <!--demo01父模块中添加依赖-->
21     <dependencyManagement>
22         <dependencies>
23             <!--Mybatis通用Mapper-->
24             <dependency>

```



```

28      </dependency>
29      <dependency>
30          <groupId>tk.mybatis</groupId>
31          <artifactId>mapper</artifactId>
32          <version>${tk-mapper.version}</version>
33      </dependency>

34
35      <!--SpringCloud-->
36      <dependency>
37          <groupId>org.springframework.cloud</groupId>
38          <artifactId>spring-cloud-dependencies</artifactId>
39          <version>Hoxton.SR9</version>
40          <type>pom</type>
41          <scope>import</scope>
42      </dependency>

43
44      <!--Spring Alibaba Cloud-->
45      <dependency>
46          <groupId>com.alibaba.cloud</groupId>
47          <artifactId>spring-cloud-alibaba-dependencies</artifactId>
48          <version>2.2.1.RELEASE</version>
49          <type>pom</type>
50          <scope>import</scope>
51      </dependency>
52  </dependencies>
53 </dependencyManagement>

```



```

1 <!--子模块order-service和storage-service的pom中添加nacos和seata依赖-->
2 <dependencies>
3     <!--nacos注册中心和配置中心-->
4     <dependency>
5         <groupId>com.alibaba.cloud</groupId>
6         <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
7     </dependency>
8     <dependency>
9         <groupId>com.alibaba.cloud</groupId>
10        <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
11    </dependency>

12
13    <!--seata-->
14    <dependency>
15        <groupId>com.alibaba.cloud</groupId>

```





```
19      <exclusion>
20          <groupId>io.seata</groupId>
21          <artifactId>seata-spring-boot-starter</artifactId>
22      </exclusion>
23  </exclusions>
24 </dependency>
25 <!--单独添加seata 1.3.0的依赖-->
26 <dependency>
27     <groupId>io.seata</groupId>
28     <artifactId>seata-spring-boot-starter</artifactId>
29     <version>1.3.0</version>
30 </dependency>
31
32 <!--openfeign-->
33 <dependency>
34     <groupId>org.springframework.cloud</groupId>
35     <artifactId>spring-cloud-starter-openfeign</artifactId>
36 </dependency>
37 <dependency>
38     <groupId>io.github.openfeign</groupId>
39     <artifactId>feign-okhttp</artifactId>
40     <version>10.2.3</version>
41 </dependency>
42
43 <!--Mybatis通用Mapper-->
44 <dependency>
45     <groupId>tk.mybatis</groupId>
46     <artifactId>mapper-spring-boot-starter</artifactId>
47 </dependency>
48 <dependency>
49     <groupId>tk.mybatis</groupId>
50     <artifactId>mapper</artifactId>
51 </dependency>
52
53 <!--mysql-->
54 <dependency>
55     <groupId>mysql</groupId>
56     <artifactId>mysql-connector-java</artifactId>
57 </dependency>
58
59 </dependencies>
```





```
1 server:
2   port: 6770
3 spring:
4   application:
5     name: order-service
6   datasource:
7     driver-class-name: com.mysql.cj.jdbc.Driver
8     username: seata_test
9     password: 'seata1234abcd!'
10    url: jdbc:mysql://rm-bp17dq6iz79761b8fxo.mysql.rds.aliyuncs.com:3306/it23!
11 cloud:
12   nacos:
13     discovery:
14       server-addr: nacos.it235.com:80
15       register-enabled: true
16       namespace: f46bbdaa-f11e-414f-9530-e6a18cbf91f6
17     config:
18       server-addr: nacos.it235.com:80
19       enabled: true
20       file-extension: yaml
21       namespace: f46bbdaa-f11e-414f-9530-e6a18cbf91f6
22
23 seata:
24   enabled: true
25   application-id: ${spring.application.name}
26   # 事务群组（可以每个应用独立取名，也可以使用相同的名字），要与服务端nacos-config
27   tx-service-group: ${spring.application.name}-tx-group
28   config:
29     type: nacos
30     # 需要和server在同一个注册中心下
31     nacos:
32       namespace: f46bbdaa-f11e-414f-9530-e6a18cbf91f6
33       serverAddr: nacos.it235.com:80
34       # 需要server端(registry和config)、nacos配置client端(registry和config)保持
35       group: SEATA_GROUP
36       username: "nacos"
37       password: "nacos"
38   registry:
39     type: nacos
40     nacos:
41       # 需要和server端保持一致，即server在nacos中的名称，默认为seata-server
42       application: seata-server
43       server-addr: nacos.it235.com:80
44       group: SEATA_GROUP
```





```

48
49 mybatis:
50   mapperLocations: classpath:mapper/*.xml

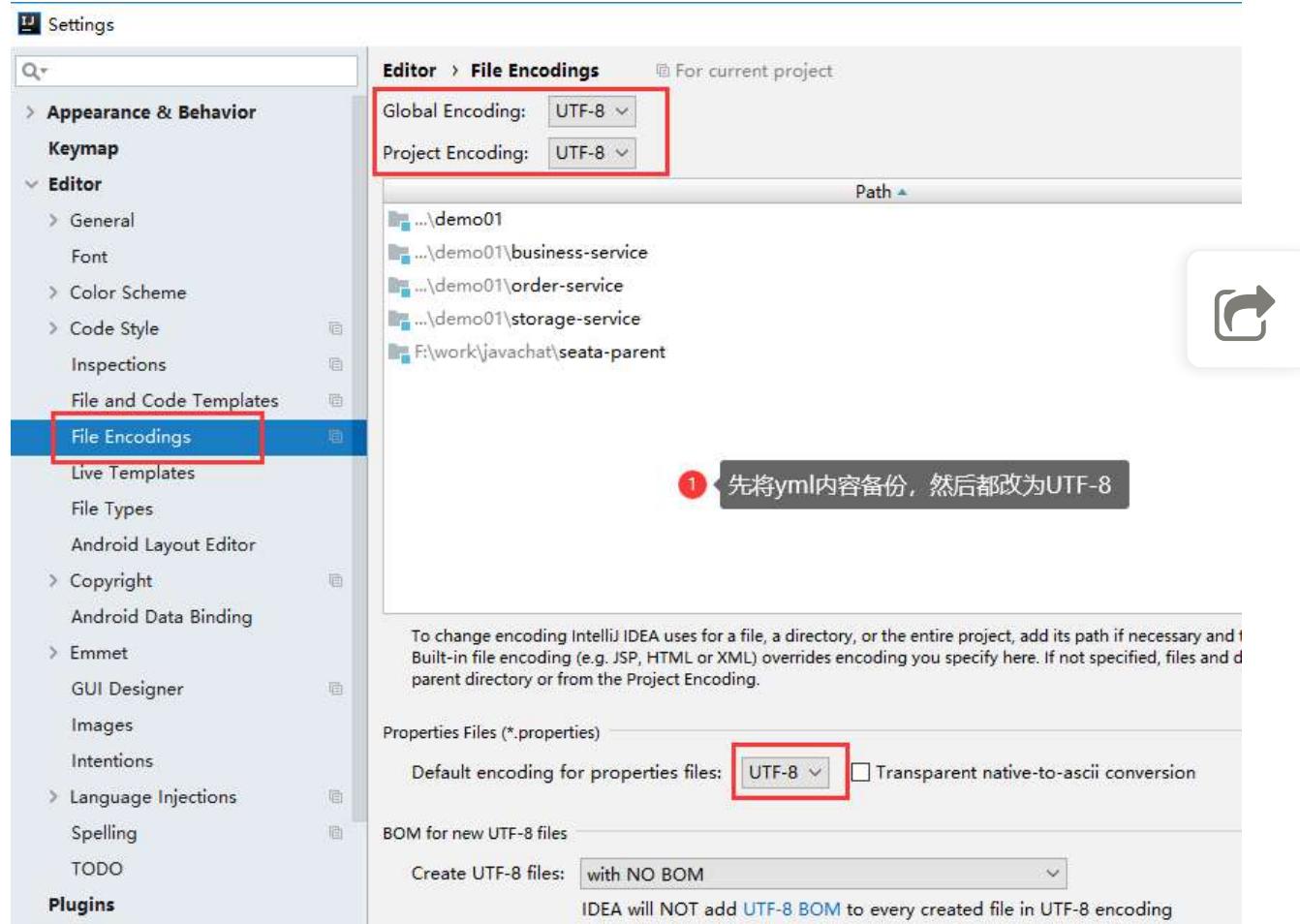
```

若出现 `bootstrap.yml` 中文注释报以下错误的情况，则需要调整编码，如下：

```

1 Caused by: org.yaml.snakeyaml.error.YAMLEException: java.nio.charset.MalformedInputException: Input length = 1
2 Caused by: java.nio.charset.MalformedInputException: Input length = 1

```



5. 编写业务代码

```

1 package com.it235.seata.order;                               java
2
3 import tk.mybatis.spring.annotation.MapperScan;
4
5 @RestController

```



君哥的学习笔记



```

9  @EnableFeignClients
10 public class OrderServiceApplication {
11
12     @Autowired
13     private OrderService orderService;
14
15     @GetMapping("order/create")
16     public Boolean create(long userId , long productId){
17         Order order = new Order();
18         order.setCount(1)
19             .setMoney(BigDecimal.valueOf(88))
20             .setProductId(productId)
21             .setUserId(userId)
22             .setStatus(0);
23         return orderService.create(order);
24     }
25
26     public static void main(String[] args) {
27         SpringApplication.run(OrderServiceApplication.class, args);
28     }
29 }
```



java

```

1  @Slf4j
2  @Service
3  public class OrderServiceImpl implements OrderService {
4
5      @Autowired
6      private OrderMapper orderMapper;
7
8      @Override
9      public boolean create(Order order) {
10         log.info("创建订单开始");
11         int index = orderMapper.insert(order);
12         log.info("创建订单结束");
13         return index > 0;
14     }
15 }
```





```

3  @Accessors(chain = true)
4  public class Order {
5
6      @Id
7      @GeneratedValue(strategy = GenerationType.IDENTITY)
8      private Long id;
9
10     private Long userId;
11
12     private Long productId;
13
14     private int count;
15
16     private BigDecimal money;
17
18     private int status;
19 }
20

```

```

1  @Repository
2  public interface OrderMapper extends Mapper<Order> {
3
4 }

```



6. 浏览器访问模拟添加订单请求, <http://localhost:6770/order/create>, 查看数据库, 此时单个服务搭建完成
7. 依葫芦画瓢, 搭建库存服务, 同时保证库存服务正常启动注册
8. 注意nacos中需要存在对应的 service.vgroupMapping



TM实现

搭建 business 服务, pom、bootstrap.yml 与RM基本一致, 并提供 FeignClient 调用组件

```

1  # feign组件超时设置, 用于查看seata数据库中的临时数据内容
2  feign:
3      client:

```



```
5    connectTimeout: 5000
6
7    read-timeout: 30000
```

FeignClient 组件代码编写

```
1 @FeignClient(name = "storage-service")
2 @Component
3 public interface StorageClient {
4
5     @GetMapping("storage/change")
6     Boolean changeStorage(@RequestParam("productId") long productId ,@RequestParam("product
7     })
8
9     @FeignClient(name = "order-service")
10    @Component
11    public interface OrderClient {
12
13        @GetMapping("order/create")
14        Boolean create(@RequestParam("userId") long userId ,@RequestParam("produ
15    }
```

java



调用层代码编写

```
1 @SpringBootApplication
2 @RestController
3 @EnableFeignClients
4 @EnableDiscoveryClient
5 public class BusinessServiceApplication {
6
7     @Autowired
8     private OrderClient orderClient;
9     @Autowired
10    private StorageClient storageClient;
11
12    @GetMapping("buy")
13    @GlobalTransactional
14    public String buy(long userId , long productId){
15        orderClient.create(userId , productId);
16        storageClient.changeStorage(userId , 1);
17        return "ok";
```

java





```
20
21     SpringApplication.run(BusinessServiceApplication.class, args);
22 }
23 }
```

运行测试

异常汇总

1. 没有可用服务 No available service , 该问题是没有连接到 seata-server 造成, 内容如下

```
1 | io.seata.common.exception.FrameworkException: No available service
```

seata-server的配置列表中有一项比较重要的配置，该配置需要手动添加：

service.vgroupMapping.orderServiceGroup=default , 其中 orderServiceGroup 是你手指定的名称，同时 bootstrap.yml 中的 seata.tx-service-group 的值要为 orderServiceGroup ，如没找到则会报错

2. 连接TC超时，Timeout的情况





君哥的学习笔记



```

本地连接 IPv6 地址 . . . . . : fe80::e9c3:8a8d:e131:3ff6%10
IPv4 地址 . . . . . : 192.168.2.196 ① 真实IP
子网掩码 . . . . . : 255.255.255.0
默认网关 . . . . . : 192.168.2.1

以太网适配器 VMware Network Adapter VMnet1:

连接特定的 DNS 后缀 . . . . . :
本地连接 IPv6 地址 . . . . . : fe80::b81d:cac6:d5e3:ec4e%14
IPv4 地址 . . . . . : 192.168.179.1
子网掩码 . . . . . : 255.255.255.0
默认网关 . . . . . :

以太网适配器 VMware Network Adapter VMnet8:

连接特定的 DNS 后缀 . . . . . :
本地连接 IPv6 地址 . . . . . : fe80::9da1:3b62:5955:1060%3
IPv4 地址 . . . . . : 192.168.253.1
子网掩码 . . . . . : 255.255.255.0
默认网关 . . . . . :

```



AT模式原理解析

TC相关的表解析

- `global_table` : 全局事务 每当有一个全局事务发起后，就会在该表中记录全局事务的ID
- `branch_table` : 分支事务 记录每一个分支事务的ID，分支事务操作的哪个数据库等信息
- `lock_table` : 全局锁



日志分析

1. UNDO_LOG日志分析

```
1  {
2      "@class": "io.seata.rm.datasource.undo.BranchUndoLog",
3      "xid": "192.168.2.196:8091:104983180048351232",
4      "branchId": 104983207323910145,
5      "sqlUndoLogs": ["java.util.ArrayList", [
6          "@class": "io.seata.rm.datasource.undo.SQLUndoLog",
7          "sqlType": "UPDATE",
8          "tableName": "tab_storage",
9          "beforeImage": {
10              "@class": "io.seata.rm.datasource.sql.struct.TableRecords",
11              "tableName": "tab_storage",
12              "rows": ["java.util.ArrayList", [
13                  "@class": "io.seata.rm.datasource.sql.struct.Row",
14                  "fields": ["java.util.ArrayList", [
15                      "@class": "io.seata.rm.datasource.sql.struct.Field",
16                      "name": "id",
17                      "keyType": "PRIMARY_KEY",
18                      "type": -5,
19                      "value": ["java.lang.Long", 1]
20                  }, {
21                      "@class": "io.seata.rm.datasource.sql.struct.Field",
22                      "name": "total",
23                      "keyType": "NULL",
24                      "type": 4,
25                      "value": 88
26                  }, {
27                      "@class": "io.seata.rm.datasource.sql.struct.Field",
28                      "name": "used",
29                      "keyType": "NULL",
30                      "type": 4,
31                      "value": 12
32                  }]
33              }]]
34          },
35          "afterImage": {
36              "@class": "io.seata.rm.datasource.sql.struct.TableRecords",
37              "tableName": "tab_storage",
38              "rows": ["java.util.ArrayList", [
39                  "@class": "io.seata.rm.datasource.sql.struct.Row",
```





君哥的学习笔记

```
43             "keyType": "PRIMARY_KEY",
44             "type": -5,
45             "value": ["java.lang.Long", 1]
46         },
47         {
48             "@class": "io.seata.rm.datasource.sql.struct.Field",
49             "name": "total",
50             "keyType": "NULL",
51             "type": 4,
52             "value": 87
53         },
54         {
55             "@class": "io.seata.rm.datasource.sql.struct.Field",
56             "name": "used",
57             "keyType": "NULL",
58             "type": 4,
59             "value": 13
60         }
61     }
62 }
```



```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
{
  "@class": "io.seata.rm.datasource.undo.BranchUndoLog",
  "xid": "192.168.2.196:8091:104983180048351232",
  "branchId": 104983197731536896,
  "sqlUndoLogs": ["java.util.ArrayList", [
    "@class": "io.seata.rm.datasource.undo.SQLUndoLog",
    "sqlType": "INSERT",
    "tableName": "tab_order",
    "beforeImage": {
      "@class": "io.seata.rm.datasource.sql.struct.TableRecords$EmptyTableRecords",
      "tableName": "tab_order",
      "rows": ["java.util.ArrayList", []]
    },
    "afterImage": {
      "@class": "io.seata.rm.datasource.sql.struct.TableRecords",
      "tableName": "tab_order",
      "rows": ["java.util.ArrayList", [
        "@class": "io.seata.rm.datasource.sql.struct.Row",
        "fields": ["java.util.ArrayList", [
          "@class": "io.seata.rm.datasource.sql.struct.Field",
          "name": "id",
          "value": 1
        ]]
      ]]
    }
  ]]
}
```



```

25     }, {
26         "@class": "io.seata.rm.datasource.sql.struct.Field",
27         "name": "user_id",
28         "keyType": "NULL",
29         "type": -5,
30         "value": ["java.lang.Long", 1]
31     }, {
32         "@class": "io.seata.rm.datasource.sql.struct.Field",
33         "name": "product_id",
34         "keyType": "NULL",
35         "type": -5,
36         "value": ["java.lang.Long", 1]
37     }, {
38         "@class": "io.seata.rm.datasource.sql.struct.Field",
39         "name": "count",
40         "keyType": "NULL",
41         "type": 4,
42         "value": null
43     }, {
44         "@class": "io.seata.rm.datasource.sql.struct.Field",
45         "name": "money",
46         "keyType": "NULL",
47         "type": 3,
48         "value": ["java.math.BigDecimal", 88]
49     }, {
50         "@class": "io.seata.rm.datasource.sql.struct.Field",
51         "name": "status",
52         "keyType": "NULL",
53         "type": 4,
54         "value": null
55     }]
56 }
57 }
58 }
59 }

```



2. 系统日志分析

```

1 2021-02-16 16:45:40.728 INFO --- [ServerHandlerThread_1_4_500] i.s.s.coordinat
2 2021-02-16 16:45:44.714 INFO --- [batchLoggerPrint_1_1] i.s.c.r.p.server.BatchL
3 2021-02-16 16:45:44.935 INFO --- [ServerHandlerThread_1_5_500] i.seata.servel

```



君哥的学习笔记

7 2021-02-16 16:46:42.720 INFO --- [RetryRollbacking_1_1] io.seata.server.coordinator...

AT优势

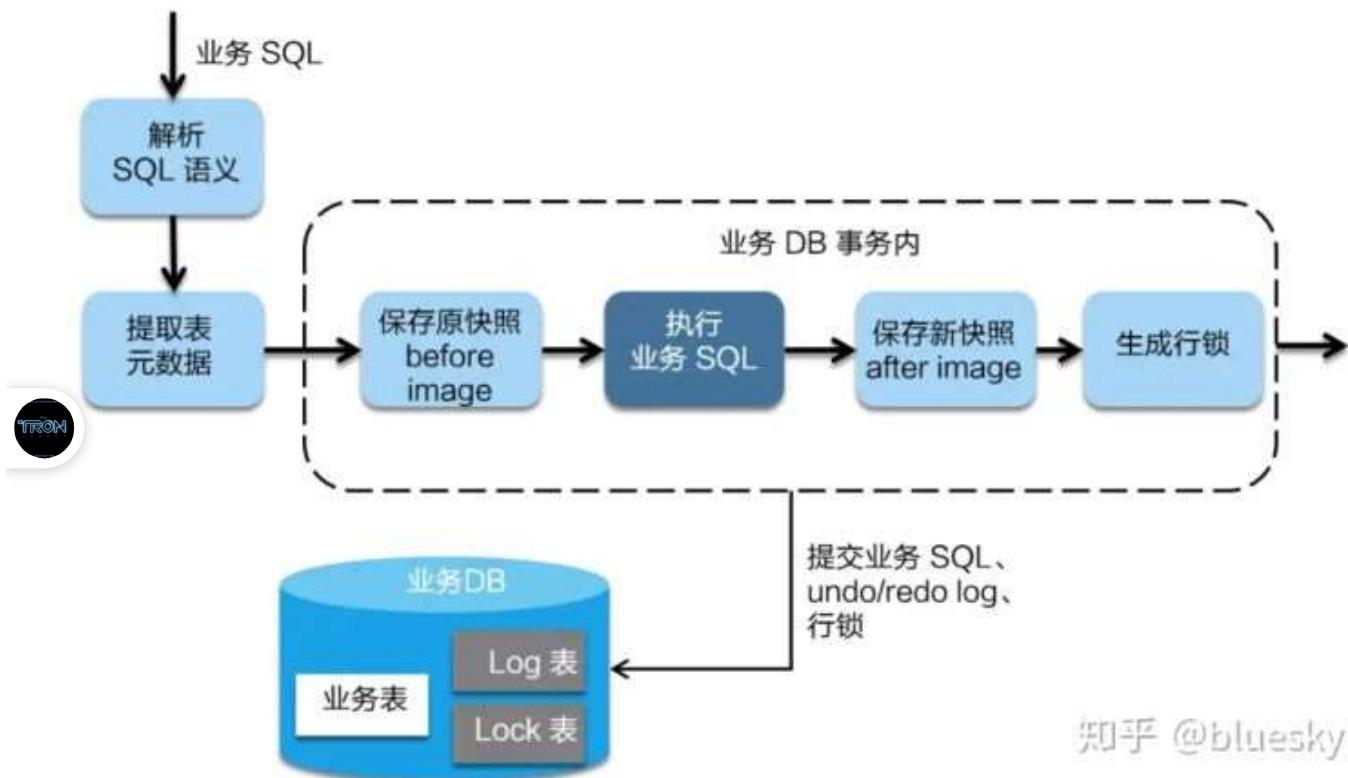
AT 模式如何做到对业务的无侵入？

一阶段步骤：

TM: business-service.buy(long, long) 方法执行时，由于该方法具有 @GlobalTransactional 标志，该TM会向TC发起全局事务，生成XID（全局锁）

- RM: OrderService.create(long, long) : 写表, UNDO_LOG记录回滚日志 (Branch ID) , 通知TC操作结果
- RM: StorageService.changeNum(long, long) : 写表, UNDO_LOG记录回滚日志 (Branch ID) , 通知TC操作结果

RM写表的过程，Seata 会拦截**业务SQL**，首先解析 SQL 语义，在业务数据被更新前，将其存成**before image**，然后执行**业务SQL**，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。

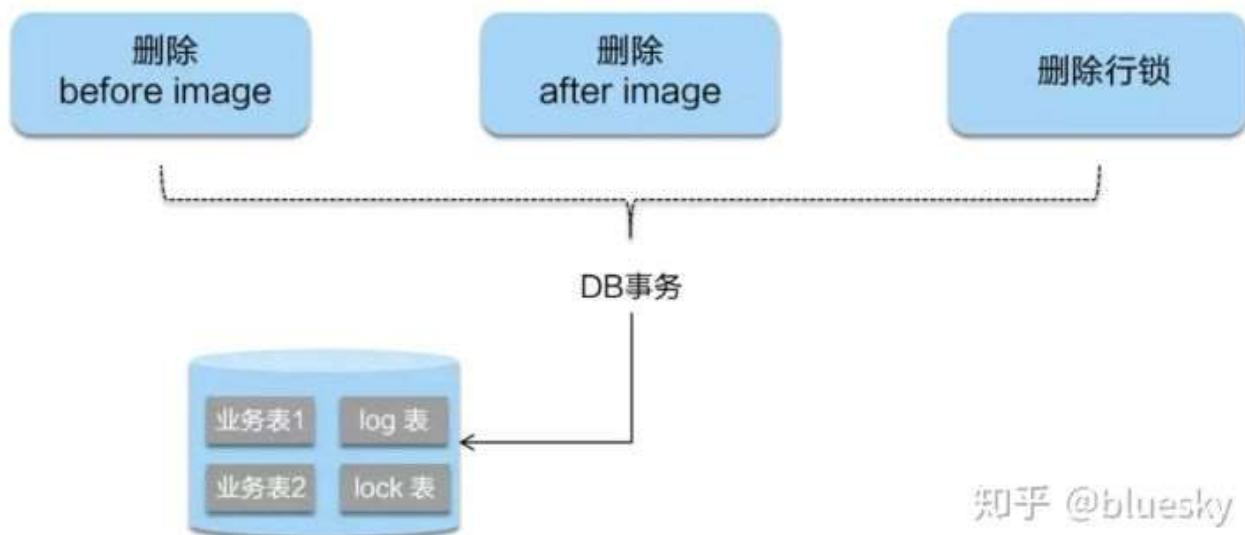




三 君哥的学习笔记

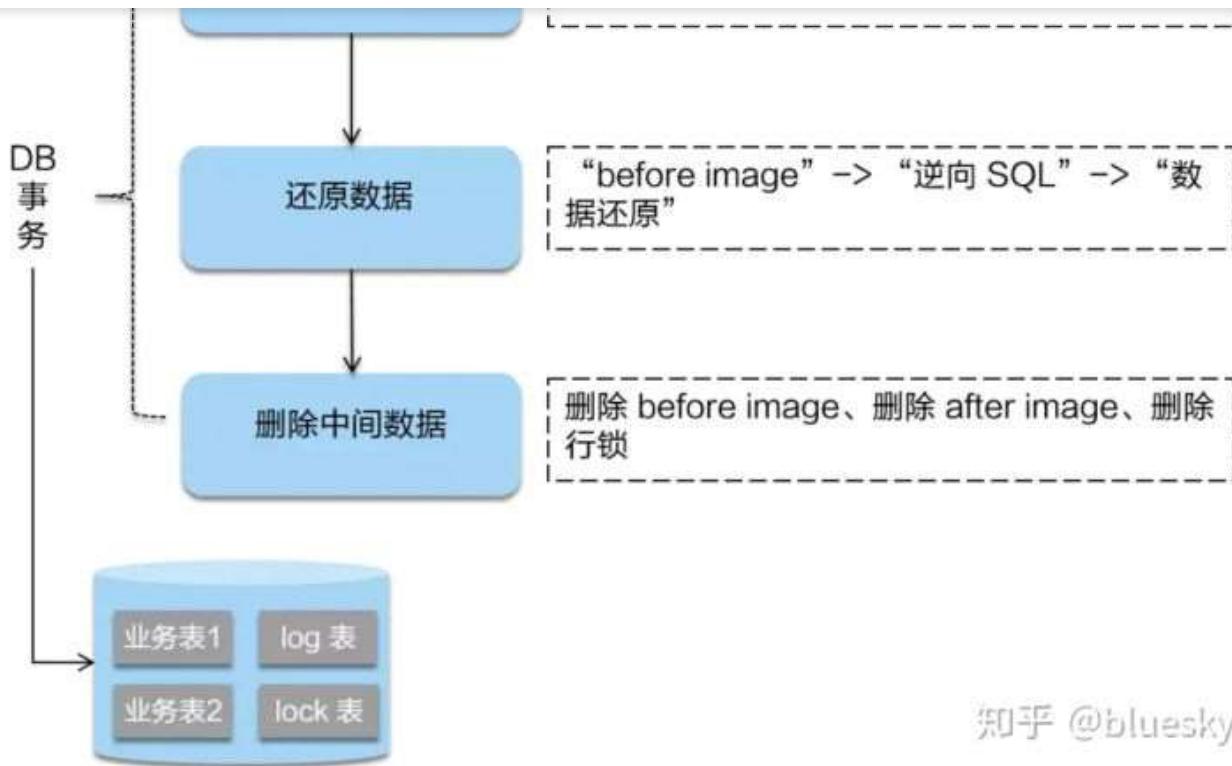
和行锁删掉，完成数据清理即可。

- 正常：TM执行成功，通知TC全局提交，TC此时通知所有的RM提交成功，删除UNDO_LOG回滚日志



- 异常：TM执行失败，通知TC全局回滚，TC此时通知所有的RM进行回滚，根据UNDO_LOG反向操作，使用before image还原业务数据，删除UNDO_LOG，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。



读写隔离

1. 写隔离

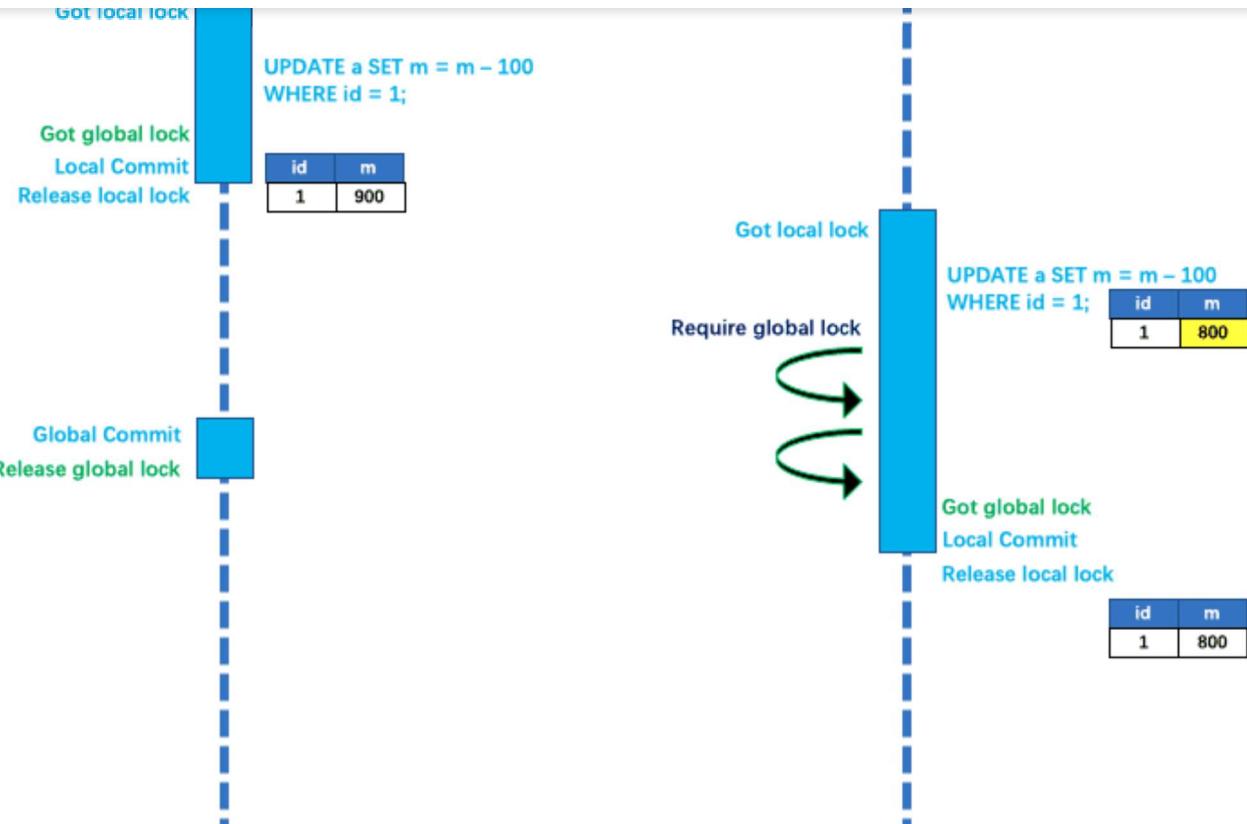
- 一阶段本地事务提交前，需要确保先拿到 **全局锁**。
- 拿不到 **全局锁**，不能提交本地事务。
- 拿 **全局锁** 的尝试被限制在一定范围内，超出范围将放弃，并回滚本地事务，释放本地锁。

以一个示例来说明：

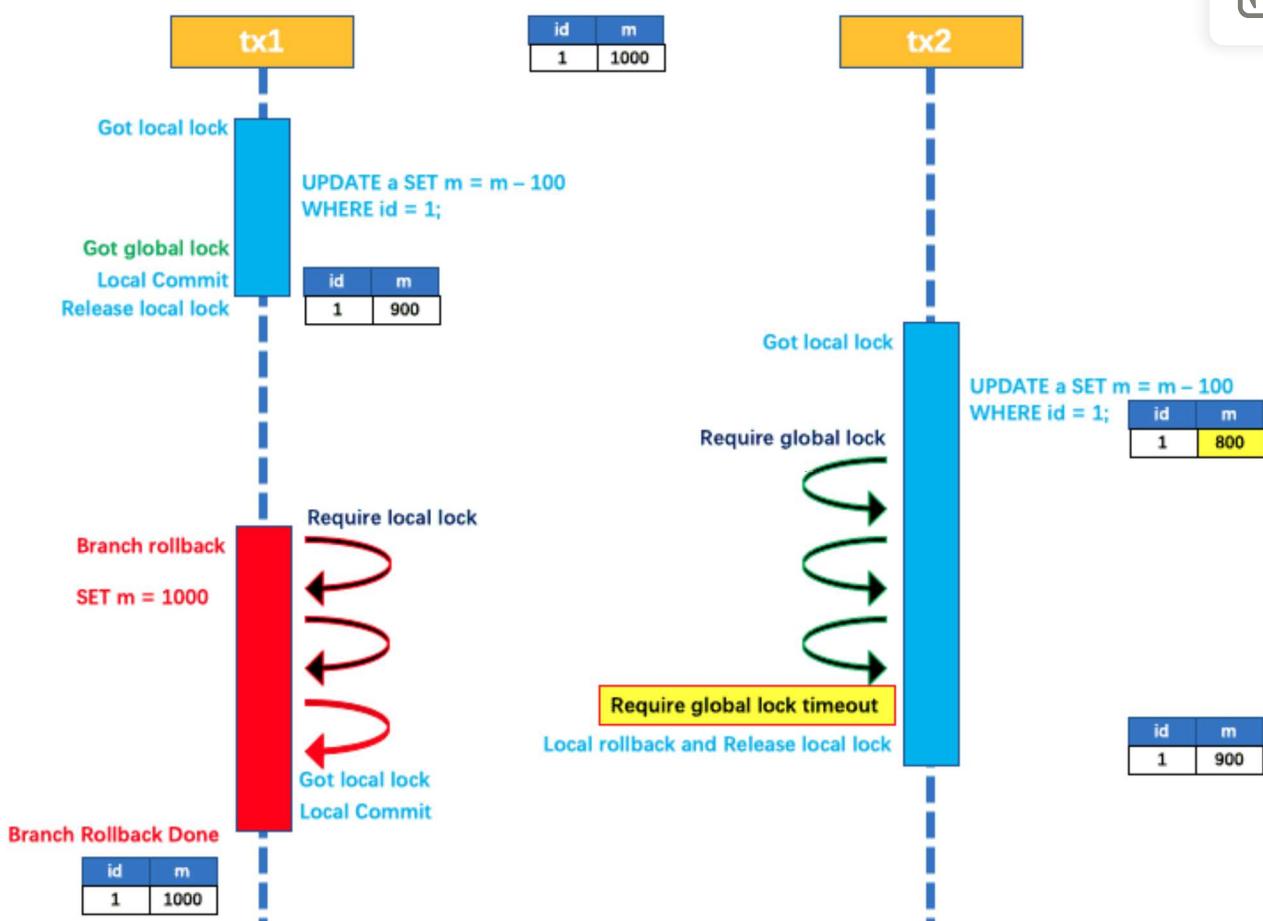


两个全局事务 tx1 和 tx2，分别对 a 表的 m 字段进行更新操作，m 的初始值 1000。

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。





君哥的学习笔记

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。

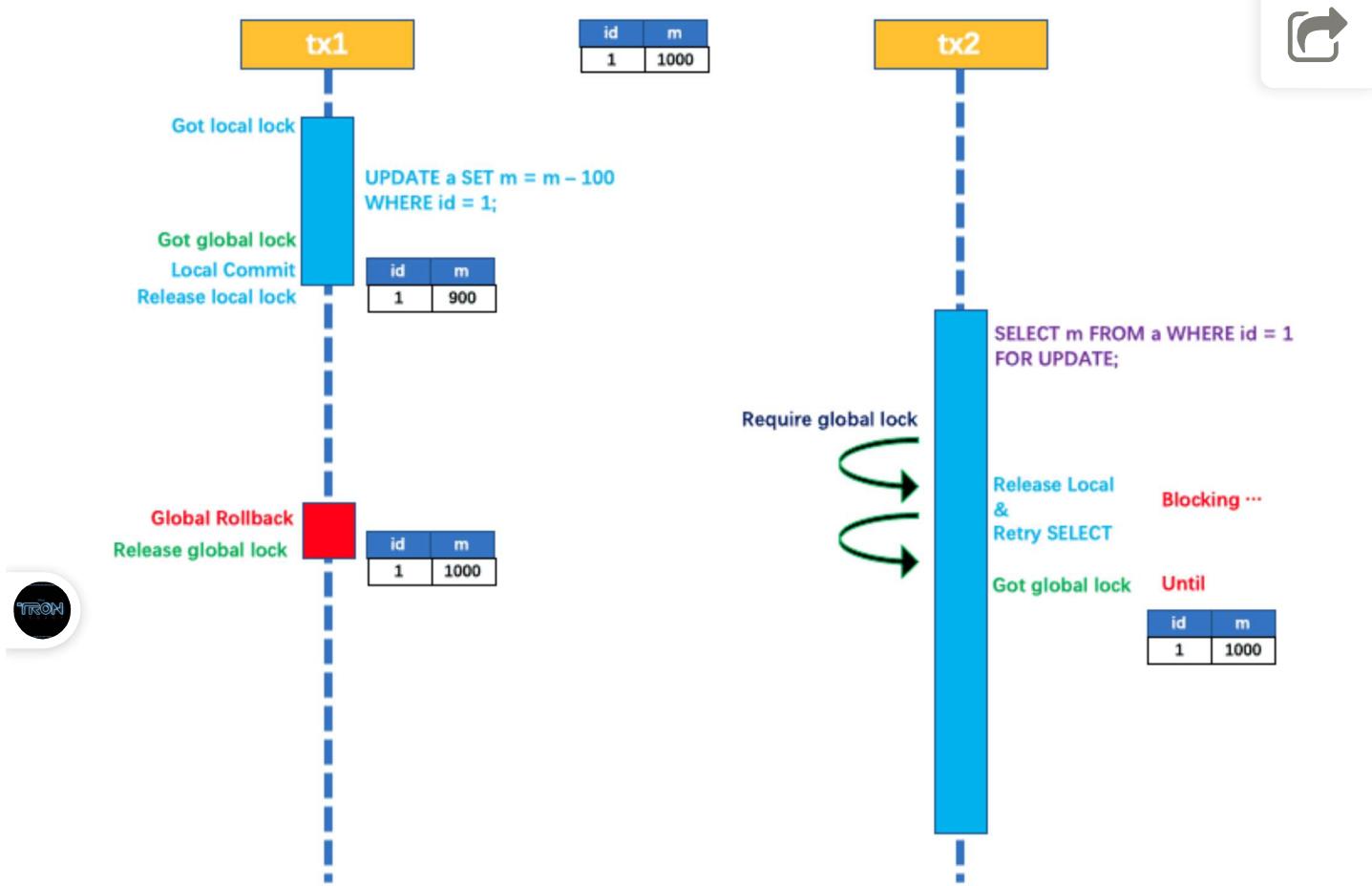
因为整个过程 **全局锁** 在 tx1 结束前一直是被 tx1 持有的，所以不会发生 **脏写** 的问题。

2. 读隔离

在数据库本地事务隔离级别 **读已提交** (Read Committed) 或以上的基础之上，Seata (AT 模式) 的默认全局隔离级别是 **读未提交** (Read Uncommitted) 。

```
1 -- 查看当前数据库的隔离级别, mysql默认为READ-COMMITTED
2 show variables like 'transaction_isolation';
```

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。





三 君哥的学习笔记

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。

for update扩展

1. 使用场景

如果遇到存在高并发并且对于数据的准确性很有要求的场景，需要使用for update。

比如涉及到金钱、库存等。一般这些操作都是很长一串并且是开启事务的。如果库存刚开始读的时候是1，而立马另一个进程进行了update将库存更新为0了，而事务还没有结束，会将错的数据一直执行下去，就会有问题。所以需要for update 进行数据加锁防止高并发时候数据出错。

- for update 仅适用于InnoDB，并且必须开启事务，在begin与commit之间才生效。
- 要测试for update的锁表情况，可以利用MySQL的Command Mode，开启二个视窗来做测试。

2. 窗口模拟

- 窗口A，非自动提交事务，用于for update操作；

```

1  set autocommit = 0;
2  begin;
3  select * from tab_order where id = 1 for update;
4
5  -- 等第二个窗口执行完成之后再执行commit
6  commit;

```

- 窗口B，用于普通update操作。

在b窗口对id=1的数据进行update name操作，发现失败：等待锁释放超时

```

1  update tab_order set product_id = 100 where id = 1;
2  ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction

```



```
1 update tab_order set product_id = 200 where id = 2;
2 Query OK, 1 row affected (0.00 sec)
```

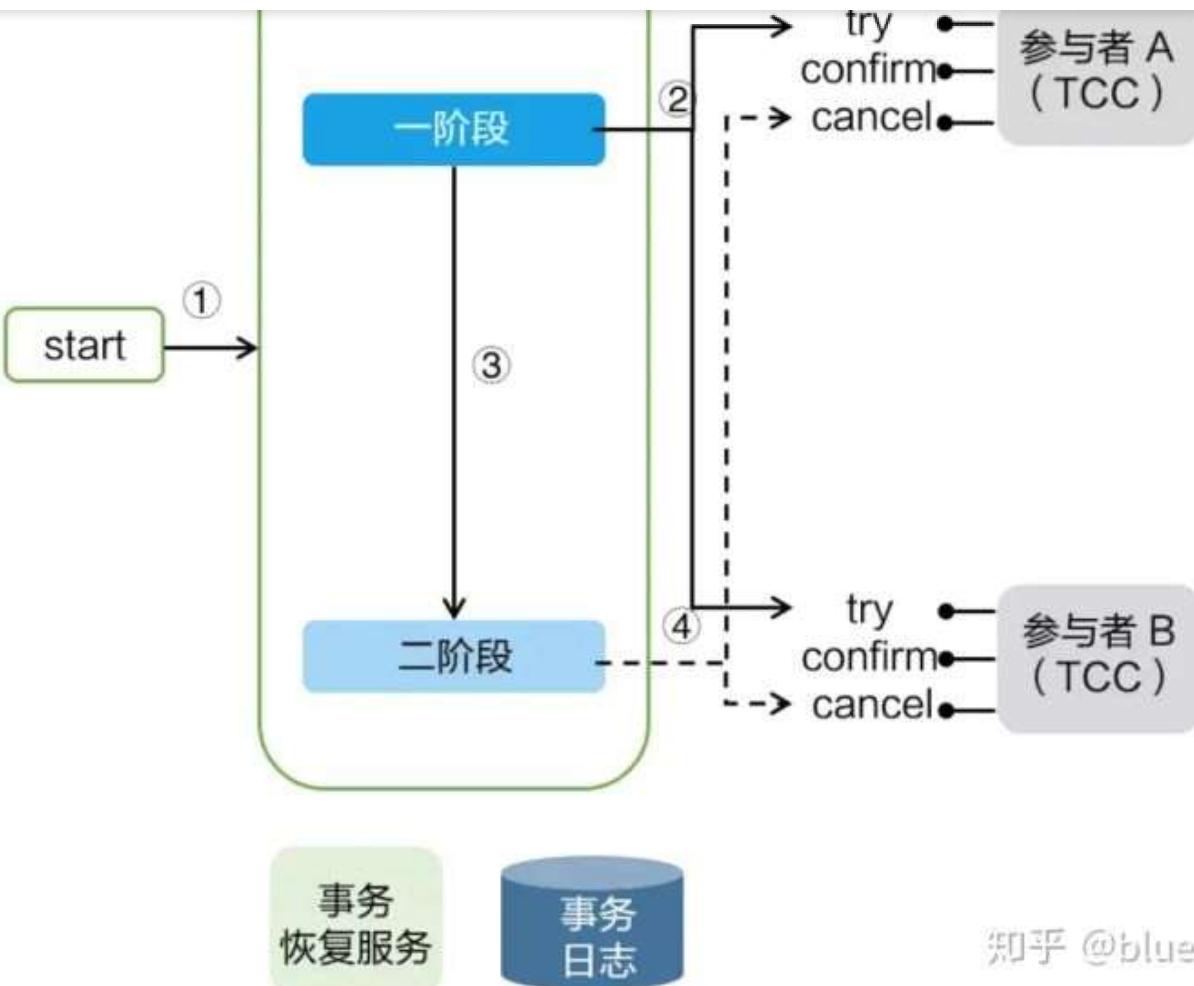
3. 总结

- for update操作在未获取到数据的时候，mysql不进行锁（no lock）
- 获取到数据的时候，进行对约束字段进行判断，存在有索引的字段则进行row lock 否则进行table lock
- 当使用 '<>','like'等关键字时，进行for update操作时，mysql进行的是table lock

第四章：TCC模式详解

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

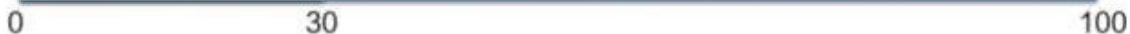
蚂蚁金服TCC实践总结以下注意事项：



> 业务模型分2阶段设计 > 并发控制 > 允许空回滚 > 防悬挂控制 > 幕等控制

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。



- 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。



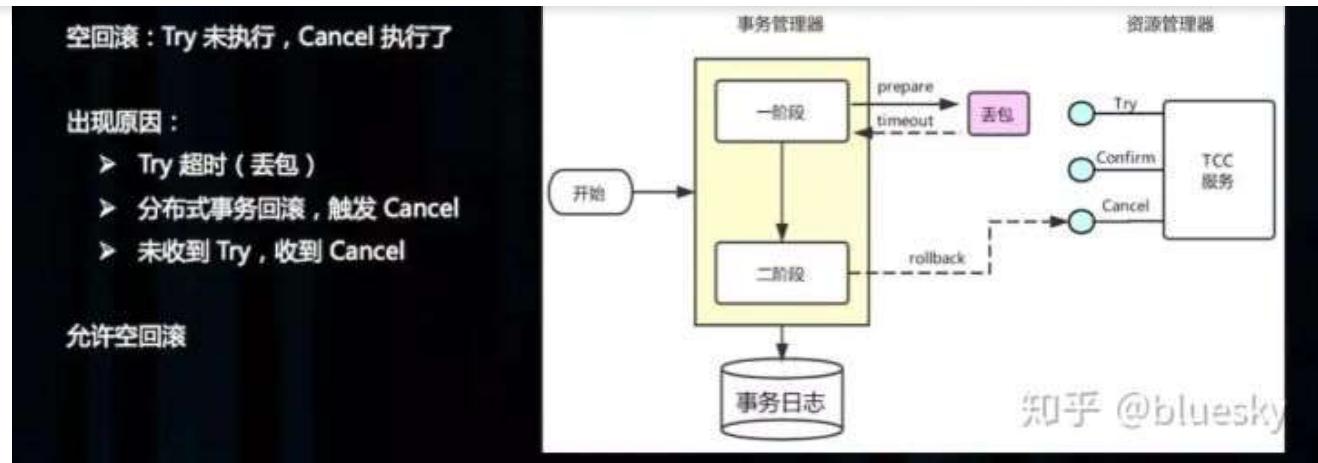
如果二阶段是回滚的话，就需要在 Cancel 方法内释放一阶段 Try 冻结的 30 元，使账号 A 的回到初始状态，100 元全部可用。

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：

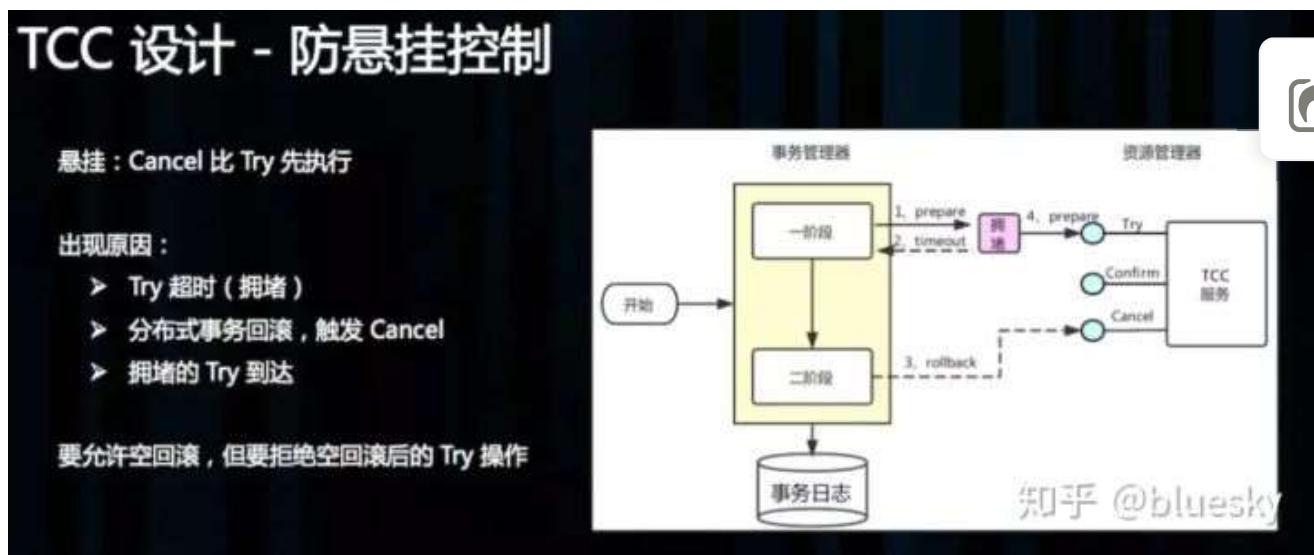


三 君哥的学习笔记



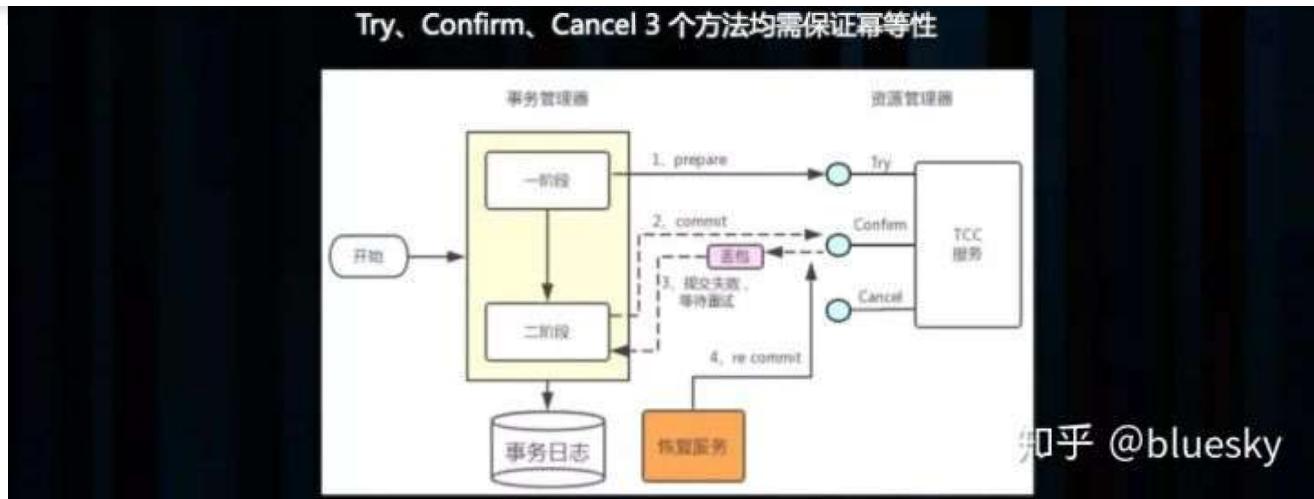
Cancel 接口设计时需要允许空回滚。在 Try 接口因为丢包时没有收到，事务管理器会触发回滚，这时会触发 Cancel 接口，这时 Cancel 执行时发现没有对应的事务 xid 或主键时，需要返回回滚成功。让事务服务管理器认为已回滚，否则会不断重试，而 Cancel 又没有对应的业务数据可以进行回滚。

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 - 幕等控制：

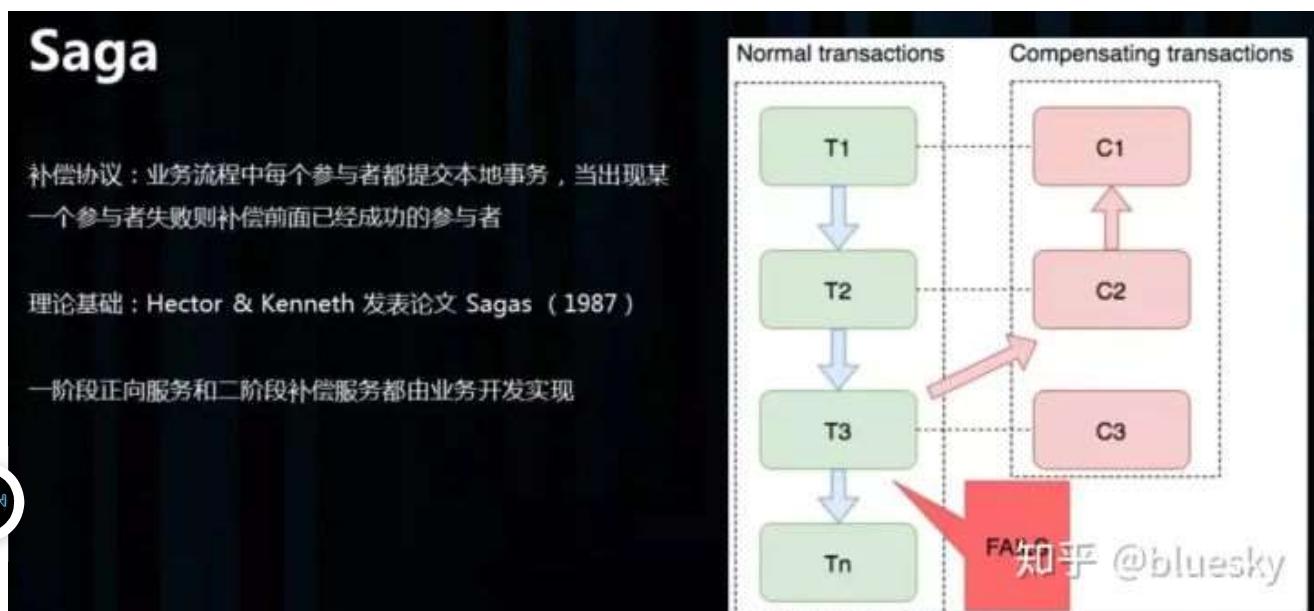


幂等性的意思是：对同一个系统，使用同样的条件，一次请求和重复的多次请求对系统资源的影响是一致的。因为网络抖动或拥堵可能会超时，事务管理器会对资源进行重试操作，所以很可能一个业务操作会被重复调用，为了不因为重复调用而多次占用资源，需要对服务设计时进行幂等控制，通常我们可以用事务 xid 或业务主键判重来控制。

第五章：Seata Saga模式详解



saga模式



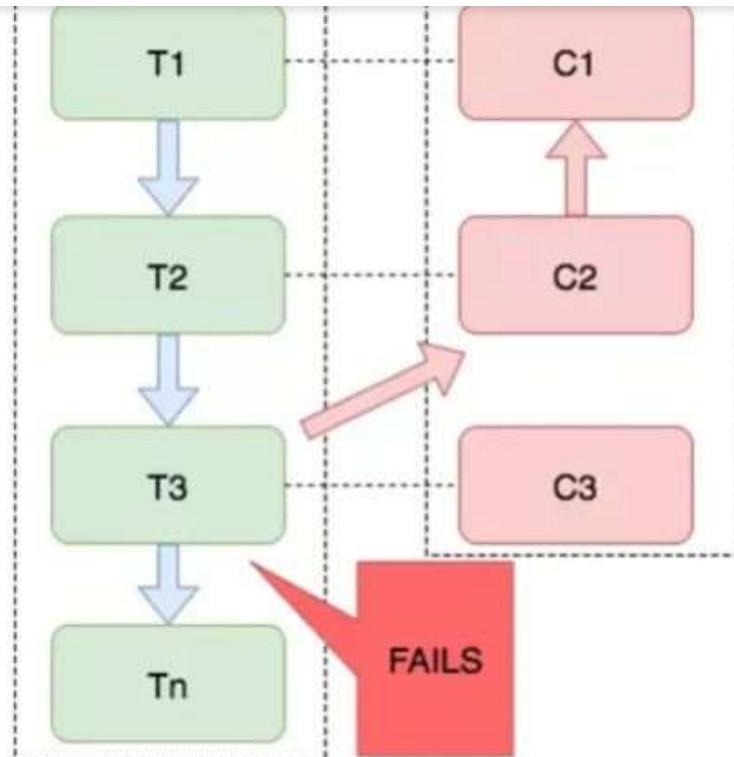
Saga 理论出自 Hector & Kenneth 1987发表的论文 Sagas。 saga模式的实现，是长事务解决方案。

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。



三

君哥的学习笔记



知乎 @bluesky

1

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：



三 君哥的学习笔记

- 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

	2PC	3PC	TCC	本地消息表	MQ事务	Saga
数据一致性	强	强	弱	弱	弱	弱
容错性	低	低	高	高	高	高
复杂性	中	高	高	低	低	高
性能	低	低	中	中	高	中
维护成本	低	中	高	中	中	高

方案比较

<https://blog.csdn.net/pbrlovejava>

2021-04-07: 6/15/2021, 5:17:04 PM

← 注册中心之Nacos详细解析

昵称

邮箱

网址(<http://>)



回复

7 评论



Anonymous

Chrome 92.0.4515.107

Windows 10.0

2021-08-08

回复

github上有代码嘛，大佬



Anonymous

Chrome 90.0.4430.212

Windows 10.0

2021-07-30



回复



Anonymous

Chrome 91.0.4472.114

Windows 10.0

2021-07-23

回复

读完感谢



Anonymous

Chrome 91.0.4472.106

Windows 10.0

2021-06-21

回复

666



君哥的学习笔记



2021-05-07

回复

6



Anonymous

Chrome 83.0.4103.61

Windows 10.0

2021-04-08

回复

秀



Anonymous

Chrome 83.0.4103.61

Windows 10.0

2021-04-08

回复

xiu

Powered By Va



v1.3.6

