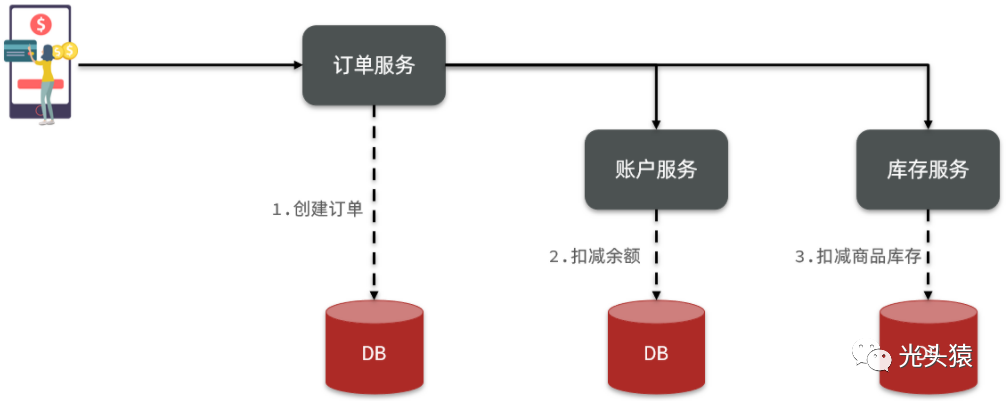
# 1、什么是分布式事务？

概述：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。

如下所示：

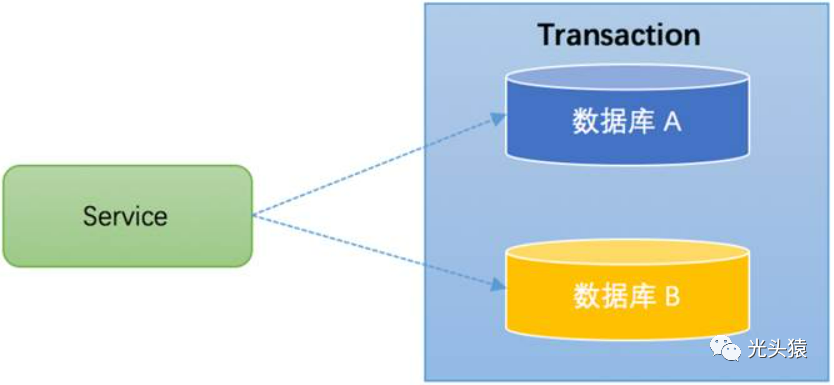


某电商系统的下单操作，需要请求三个服务来完成，这三个服务分别是：订单服务，账户服务，库存服务。当订单生成完毕以后，就需要分别请求账户服务和库存服务进行进行账户余额的扣减和库存扣减。假设都扣减成功了，此时在执行下单的后续操作时出现了问题，那么订单数据库就进行事务回滚，订单生成失败，而账户余额和扣减则都扣减成功了。这就出现了问题，而分布式事务就是解决上述这种不一致问题的。

# 2、哪些场景下都会产生分布式事务？

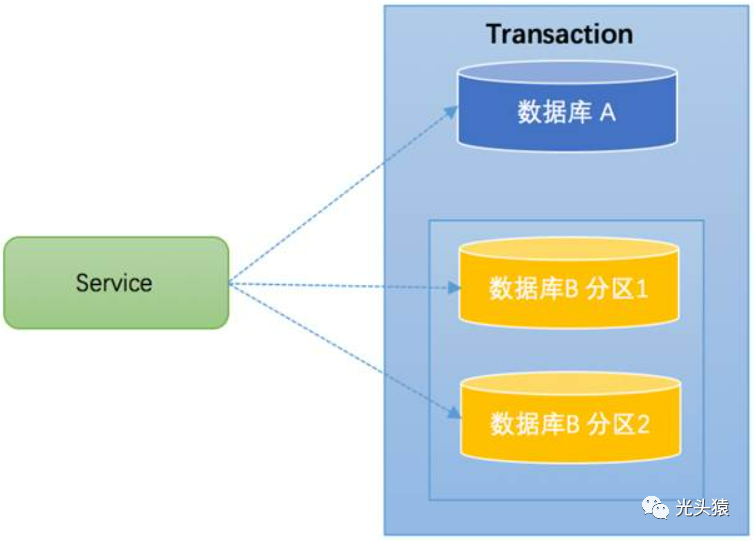
场景1：跨库事务

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。如下所示：



场景二：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。

场景三：跨服务事务

跨服务事务指的是，一个应用某个功能需要调用多个微服务进行实现，不同的微服务操作的是不同的数据库。如下所示：

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。

# 3、什么是CAP理论？

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服务无法同时满足以下3个属性：

1、一致性(Consistency) ：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(强一致性)，不能存在中间状态。

2、可用性(Availability) ：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。

3、分区容错性(Partition tolerance) ：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。

如下所示：

# 4、为什么分布式系统中无法同时保证一致性和可用性？

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。

# 5、什么是BASE理论？

CAP是分布式系统设计理论，BASE是CAP理论中AP方案的延伸，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）。它的思想包含三方面：

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。

2、Soft state（软状态）：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。

# 6、分布式事务的常见的解决方案有哪些？

## 方案一：2PC

两阶段提交又称2PC，2PC是一个非常经典的强一致、中心化的原子提交协议 。

中心化是指协议中有两类节点：一个是中心化协调者节点 （coordinator）和 N个参与者节点 （partcipant）。

两个阶段 ：

1、第一阶段：投票阶段

2、第二阶段：提交/执行阶段。

举例：订单服务A，需要调用支付服务B去支付，支付成功则处理订单状态为待发货状态，否则就需要将订单状态处理为失败状态。那么看2PC阶段是如何处理的。

阶段一：

阶段一执行流程：

1、事务询问协调者向所有的参与者发送事务预处理请求，称之为Prepare，并开始等待各参与者的响应。

2、执行本地事务各个参与者节点执行本地事务操作，但在执行完成后并不会真正提交数据库本地事务，而是先向协调者报告说：“我这边可以处理了/我这边不能处理”。

3、各参与者向协调者反馈事务询问的响应如果参与者成功执行了事务操作,那么就反馈给协调者Yes响应,表示事务可以执行,如果没有参与者成功执行事务,那么就反馈给协调者 No 响应,表示事务不可以执行。

阶段二：

阶段二执行流程：

1、所有的参与者反馈给协调者的信息都是Yes,那么就会执行事务提交协调者向所有参与者节点发出Commit请求

2、事务提交参与者收到Commit请求之后,就会正式执行本地事务Commit操作,并在完成提交之后释放整个事务执行期间占用的事务资源。

## 方案二：3PC

三阶段提交又称3PC，其在两阶段提交的基础上增加了CanCommit阶段，并引入了超时机制。一旦事务参与者迟迟没有收到协调者的Commit请求，就会自动进行本地commit，这样相对有效地解决了协调者单点故障的问题。

阶段一：

阶段一执行流程：

1、事务询问协调者向所有的参与者发送事务can commit请求，类似于2PC中的第二个阶段中的Prepare阶段，是一种事务询问操作，事务的协调者向所有参与者询问“你们是否可以完成本次事务？”，并开始等待各参与者的响应。

2、如果参与者节点认为自身可以完成事务就返回“YES”，否则“NO”。

阶段二：

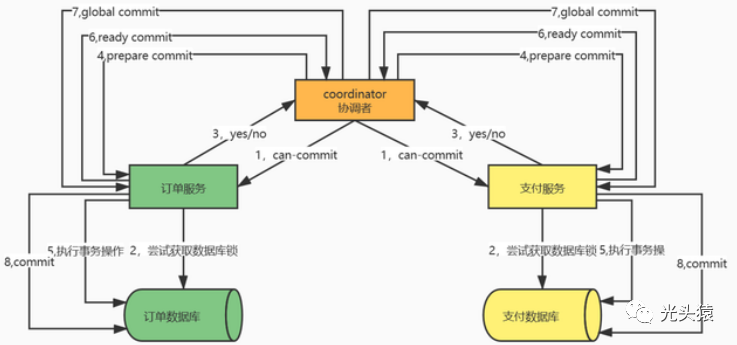
阶段二的执行流程：

1、在阶段一中，如果所有的参与者都返回Yes的话，那么就会进入PreCommit阶段进行事务预提交。此时分布式事务协调者会向所有的参与者节点发送PreCommit请求。

2、参与者收到后开始执行事务操作，参与者执行完事务操作后（此时属于未提交事务的状态），就会向协调者反馈“Ack”表示我已经准备好提交了，并等待协调者的下一步指令。

3、如果阶段一中有任何一个参与者节点返回的结果是No响应，或者协调者在等待参与者节点反馈的过程中超时。整个分布式事务就会中断，协调者就会向所有的参与者发送“abort”请求。

阶段三：



阶段三执行流程：

1、在阶段二中如果所有的参与者节点都可以进行PreCommit提交，那么协调者就会从“预提交状态”-》“提交状态”。然后向所有的参与者节点发送"doCommit"请求。

2、参与者节点在收到提交请求后就会各自执行事务提交操作，并向协调者节点反馈“Ack”消息，协调者收到所有参与者的Ack消息后完成事务。

3、相反，如果有一个参与者节点未完成PreCommit的反馈或者反馈超时，那么协调者都会向所有的参与者节点发送abort请求，从而中断事务。

## 方案三：TCC

TCC（Try-Confirm-Cancel）又称补偿事务。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）"。

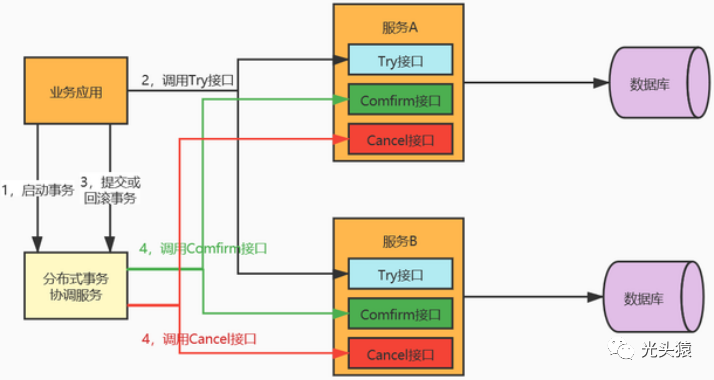
它分为三个操作：

1、Try阶段：主要是对业务系统做检测及资源预留。

2、Confirm阶段：确认执行业务操作。

3、Cancel阶段：取消执行业务操作。

如下所示：

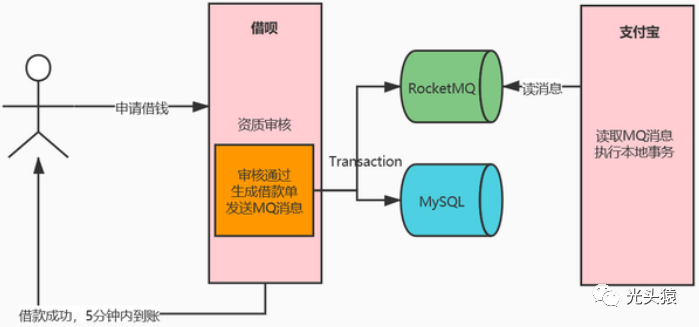


TCC事务的处理流程与2PC两阶段提交类似，不过2PC通常都是在跨库的DB层面，而TCC本质上就是一个应用层面的2PC，需要通过业务逻辑来实现。这种分布式事务的实现方式的优势在于，可以让应用自己定义数据库操作的粒度，使得降低锁冲突、提高吞吐量成为可能。不足之处则在于对应用的侵入性非常强，业务逻辑的每个分支都需要实现try、confirm、cancel三个操作。此外，其实现难度也比较大，需要按照网络状态、系统故障等不同的失败原因实现不同的回滚策略。为了满足一致性的要求，confirm和cancel接口还必须实现幂等。

## 方案四：MQ分布式事务

上面的三种分布式事务的解决方案适用于对数据一致性要求很高的场景。如果数据强一致性要求没那么高，可以采用消息中间件（MQ）实现事务最终一致。在支付系统中，常常使用的分布式事务解决方案就是基于MQ实现的，它对数据强一致性要求没那么高，但要求数据最终一致即可。

例如：向借呗申请借钱，借呗审核通过后支付宝的余额才会增加，但借呗和支付宝有可能不是同一个系统，这时候如何实现事务呢？实现方案如下图：



执行流程如下所示：

1、找花呗借钱

2、花呗借钱审核通过，同步生成借款单

3、借款单生成后，向MQ发送消息，通知支付宝转账

4、支付宝读取MQ消息，并增加账户余额

上图最复杂的其实是如何保障2、3在同一个事务中执行（本地事务和MQ消息发送在同一个事务执行），借款结束后，借呗数据处理就完成了，接下来支付宝才能读到消息，然后执行余额增加，这才完成整个操作。如果中途操作发生异常，例如支付宝余额增加发生问题怎么办？此时需要人工解决，没有特别好的办法，但这种事故概率极低。

# 7、Seata的架构是什么？

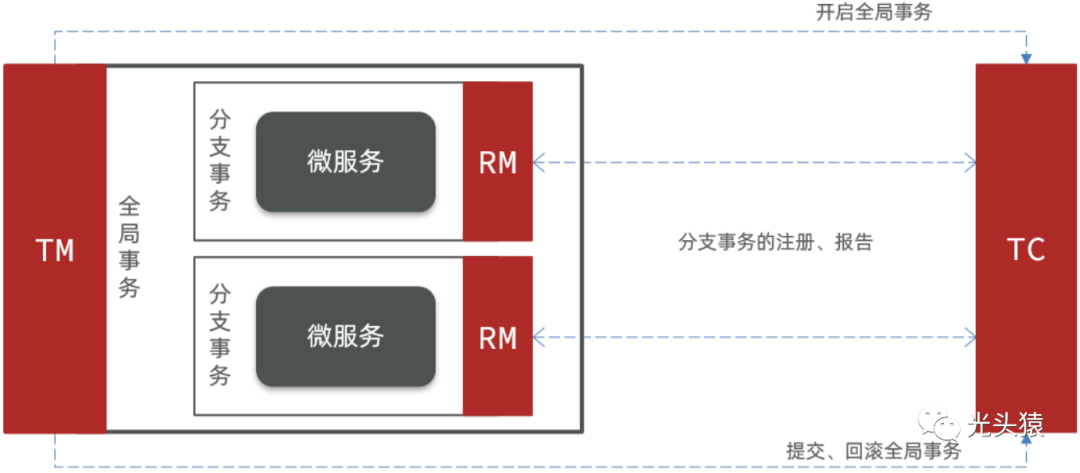
Seata事务管理中有三个重要的角色：

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

2、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。

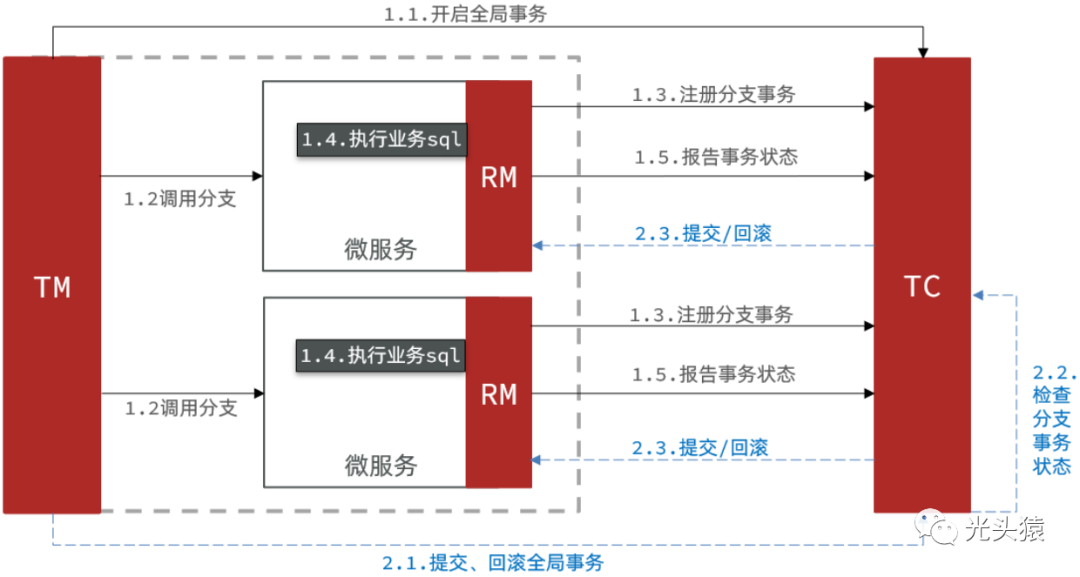
3、RM (Resource Manager) -资源管理器：管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。

如下所示：



# 8、XA模式的工作流程是什么？

xa模式整个工作流程图如下所示：



分为两个阶段：

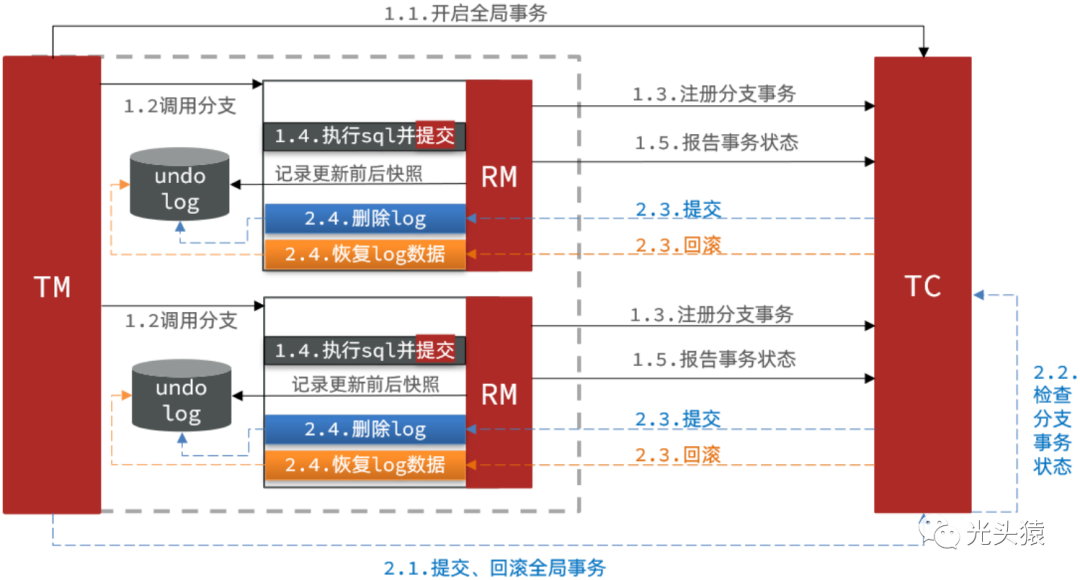
1、RM一阶段的工作：① 注册分支事务到TC ② 执行分支业务sql但不提交 ③ 报告执行状态到TC

2、TC二阶段的工作：TC检测各分支事务执行状态 ①如果都成功，通知所有RM提交事务 ②如果有失败，通知所有RM回滚事务

3、RM二阶段的工作：接收TC指令，提交或回滚事务xa模式牺牲了可用性，保证了强一致性

# 9、AT模型的工作原理是什么？

at模式的整个工作流程图如下所示：



1、阶段一RM的工作：① 注册分支事务  ② 记录undo-log（数据快照）③ 执行业务sql并提交 ④报告事务状态

2、阶段二提交时RM的工作：删除undo-log即可

3、阶段二回滚时RM的工作：根据undo-log恢复数据到更新前at模式牺牲了一致性，保证了可用性

# 10、TCC模型的工作原理是什么？

TCC模式与AT模式非常相似，每阶段都是独立事务，不同的是TCC通过人工编码来实现数据恢复。需要实现三个方法：

1、Try：资源的检测和预留；

2、Confirm：完成资源操作业务；要求 Try 成功 Confirm 一定要能成功。

3、Cancel：预留资源释放，可以理解为try的反向操作。

Seata中的tcc模型的执行流程如下所示：

