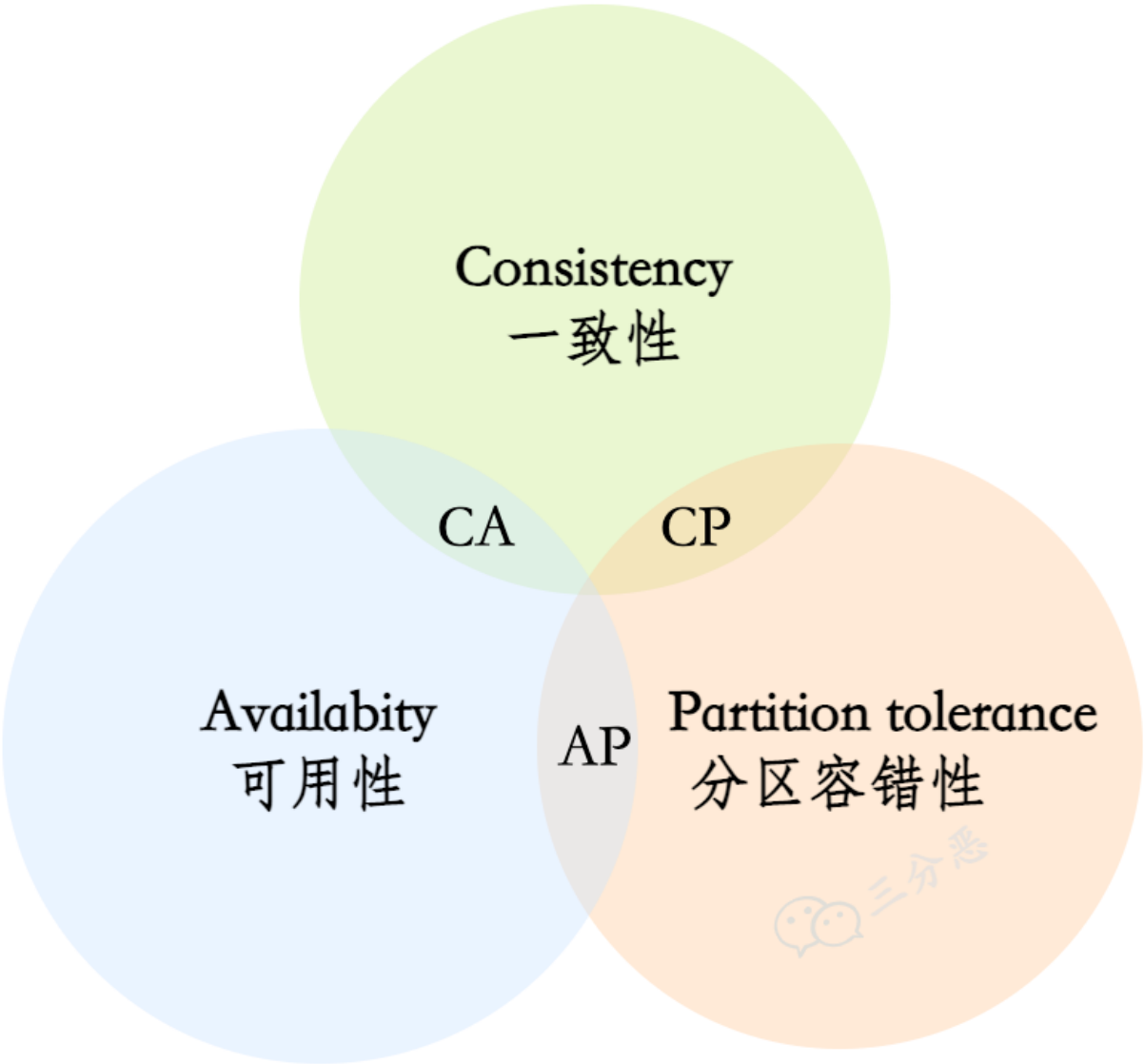
> 图文详解 12 道分布式面试高频题，这次面试，一定吊打面试官，整理：沉默王二，戳转载链接，作者：三分恶，戳原文链接。

分布式理论

1. 说说CAP原则?

CAP原则又称CAP定理，指的是在一个分布式系统中，Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性）这3个基本需求，最多只能同时满足其中的2个。



选项	描述
Consistency（一致性）	指数据在多个副本之间能够保持一致的特性（严格的一致性）
Availability（可用性）	指系统提供的服务必须一直处于可用的状态，每次请求都能获取到非错的响应（不保证获取的数据为最新数据）
Partition tolerance（分区容错性）	分布式系统在遇到任何网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务，除非整个网络环境都发生了故障

2. 为什么CAP不可兼得呢?

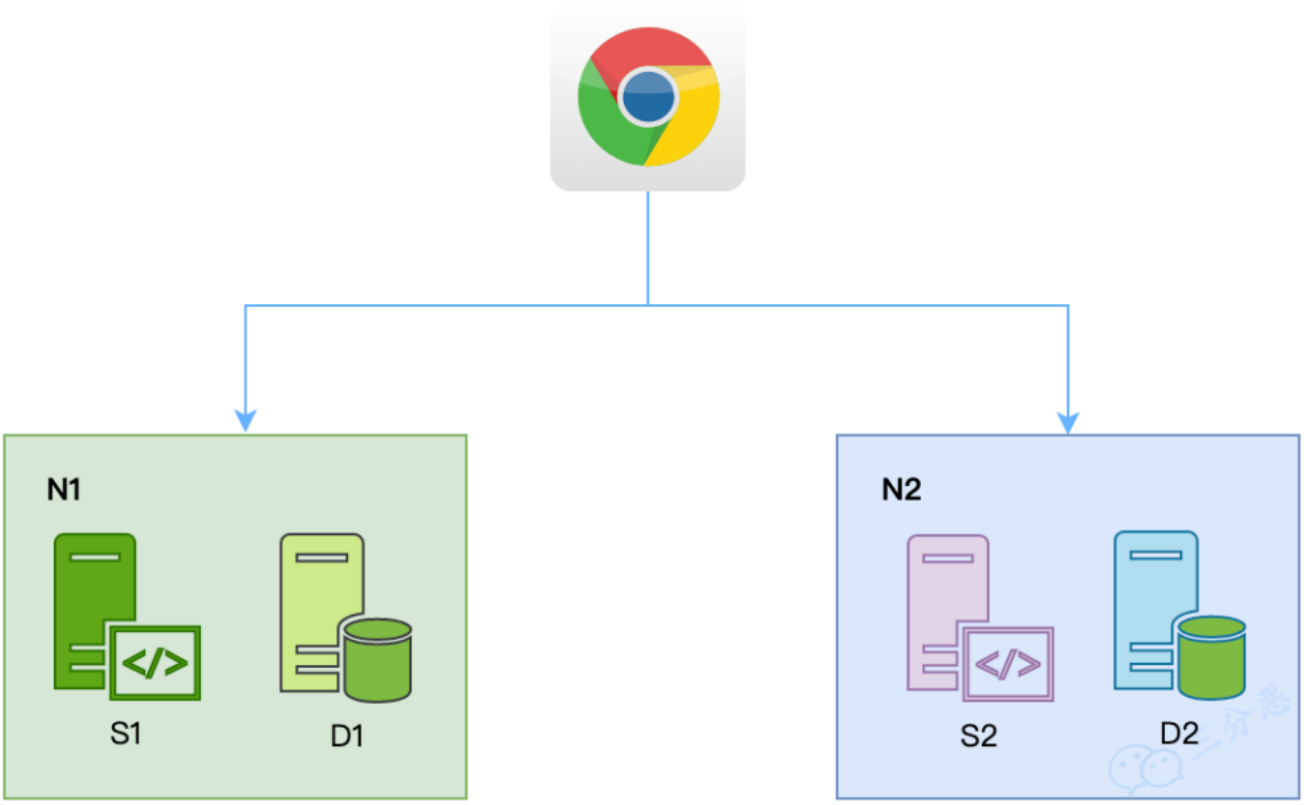
首先对于分布式系统，分区是必然存在的，所谓分区指的是分布式系统可能出现的字区域网络不通，成为孤立区域的的情况。

那么分区容错性（**P**）就必须要满足，因为如果要牺牲分区容错性，就得把服务和资源放到一个机器，或者一个“同生共死”的集群，那就违背了分布式的初衷。

那么满足分区容错的基础上，能不能同时满足`一致性`和`可用性`？

假如现在有两个分区`N1`和`N2`，N1和N2分别有不同的分区存储D1和D2，以及不同的服务S1和S2。

- 在满足`一致性`的时候，N1和N2的数据要求值一样的，D1=D2。
- 在满足`可用性`的时候，无论访问N1还是N2，都能获取及时的响应。



假如现在有这样的场景：

- 用户访问了N1，修改了D1的数据。
- 用户再次访问，请求落在了N2。此时D1和D2的数据不一致。

接下来：

- 保证`一致性`：此时D1和D2数据不一致，要保证一致性就不能返回不一致的数据，`可用性`无法保证。
- 保证`可用性`：立即响应，可用性得到了保证，但是此时响应的数据和D1不一致，`一致性`无法保证。

所以，可以看出，分区容错的前提下，`一致性`和`可用性`是矛盾的。

3. CAP对应的模型和应用?

CA without P

理论上放弃P（分区容错性），则C（强一致性）和A（可用性）是可以保证的。实际上分区是不可避免的，严格上CA指的是允许分区后各子系统依然保持CA。

CA模型的常见应用：

- 集群数据库
- xFS文件系统

CP without A

放弃A（可用），相当于每个请求都需要在Server之间强一致，而P（分区）会导致同步时间无限延长，如此CP也是可以保证的。很多传统的数据库分布式事务都属于这种模式。

CP模型的常见应用：

- 分布式数据库
- 分布式锁

AP wihtout C

要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。现在众多的NoSQL都属于此类。

AP模型常见应用：

- Web缓存
- DNS

举个大家更熟悉的例子，像我们熟悉的注册中心`ZooKeeper`、`Eureka`、`Nacos`中：

- ZooKeeper 保证的是 CP
- Eureka 保证的则是 AP
- Nacos 不仅支持 CP 也支持 AP

4. BASE理论了解吗?

BASE（Basically Available、Soft state、Eventual consistency）是基于CAP理论逐步演化而来的，核心思想是即便不能达到强一致性（Strong consistency），也可以根据应用特点采用适当的方式来达到最终一致性（Eventual consistency）的效果。

BASE的主要含义：

- **Basically Available（基本可用）**

什么是基本可用呢？假设系统出现了不可预知的故障，但还是能用，只是相比较正常的系统而言，可能会有响应时间上的损失，或者功能上的降级。

- **Soft State（软状态）**

什么是硬状态呢？要求多个节点的数据副本都是一致的，这是一种“硬状态”。

软状态也称为弱状态，相比较硬状态而言，允许系统中的数据存在中间状态，并认为该状态不影响系统的整体可用性，即允许系统在多个不同节点的数据副本存在数据延时。

- **Eventually Consistent（最终一致性）**

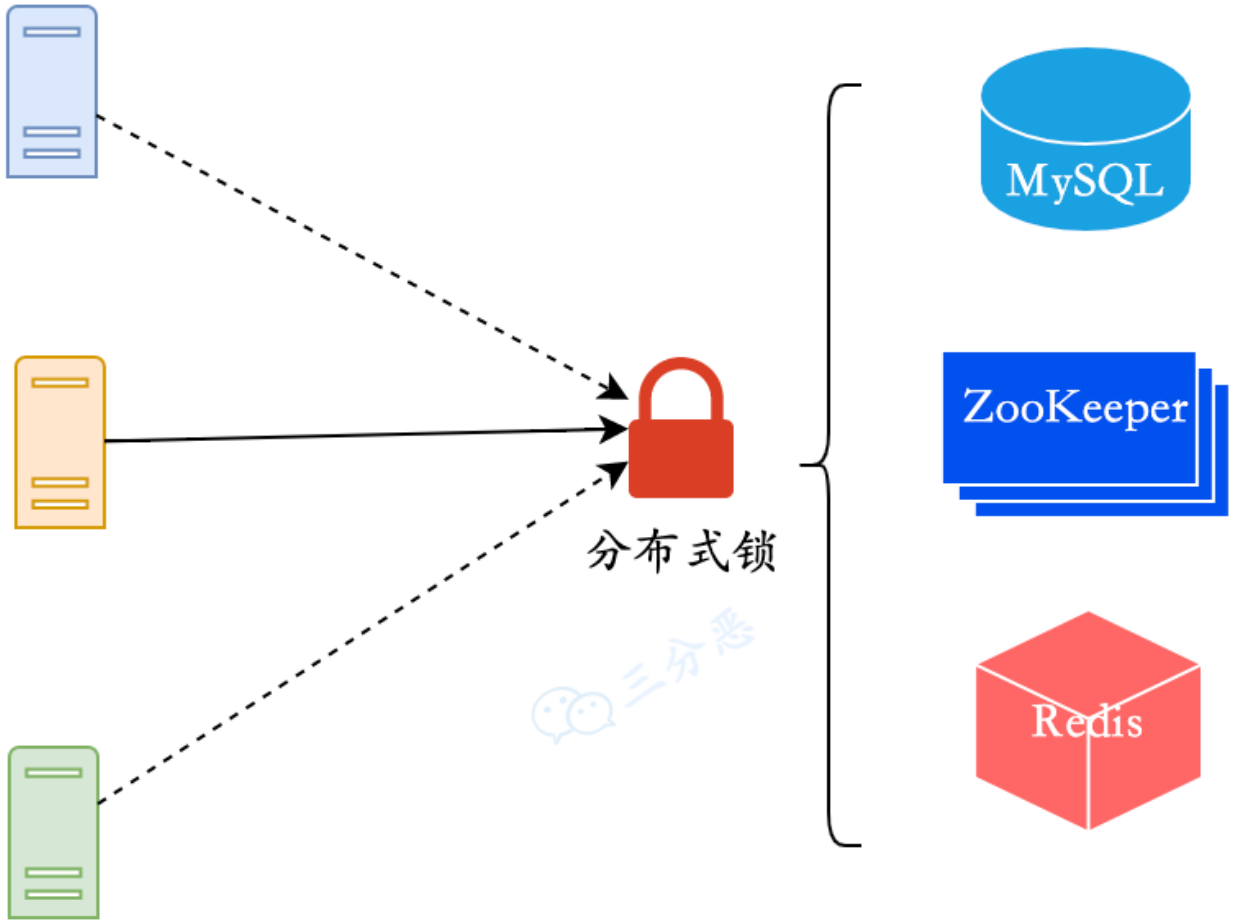
上面说了软状态，但是不应该一直都是软状态。在一定时间后，应该到达一个最终的状态，保证所有副本保持数据一致性，从而达到数据的最终一致性。这个时间取决于网络延时、系统负载、数据复制方案设计等等因素。

分布式锁

单体时代，可以直接用本地锁来实现对竞争资源的加锁，分布式环境下就要用到分布式锁了。

5. 有哪些分布式锁的实现方案呢?

常见的分布式锁实现方案有三种：`MySQL分布式锁`、`ZooKepper分布式锁`、`Redis分布式锁`。



5.1 MySQL分布式锁如何实现呢?

用数据库实现分布式锁比较简单，就是创建一张锁表，数据库对字段作唯一性约束。

加锁的时候，在锁表中增加一条记录即可；释放锁的时候删除记录就行。

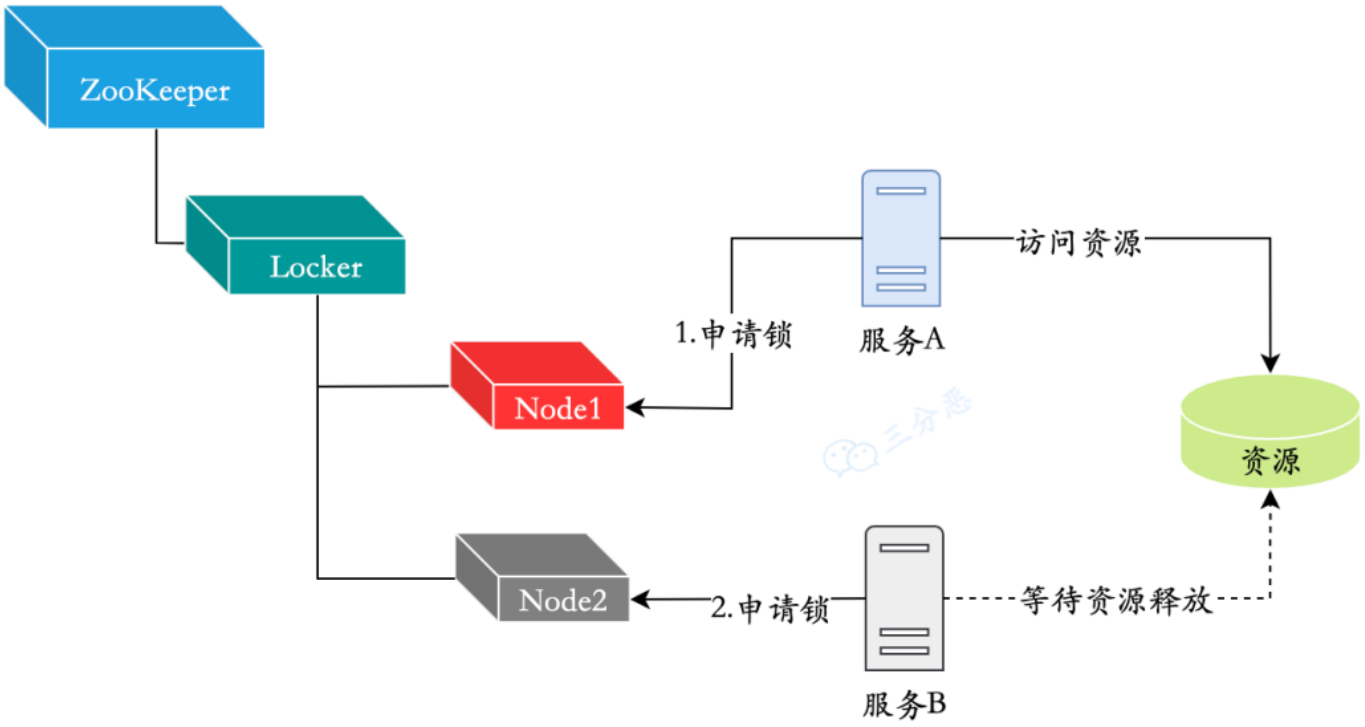
如果有并发请求同时提交到数据库，数据库会保证只有一个请求能够得到锁。

这种属于数据库 IO 操作，效率不高，而且频繁操作会增大数据库的开销，因此这种方式在高并发、高性能的场景中用的不多。

5.2 ZooKeeper如何实现分布式锁?

ZooKeeper也是常见分布式锁实现方法。

ZooKeeper的数据节点和文件目录类似，例如有一个lock节点，在此节点下建立子节点是可以保证先后顺序的，即便是两个进程同时申请新建节点，也会按照先后顺序建立两个节点。



所以我们可以用此特性实现分布式锁。以某个资源为目录，然后这个目录下面的节点就是我们需要获取锁的客户端，每个服务在目录下创建节点，如果它的节点，序号在目录下最小，那么就获取到锁，否则等待。释放锁，就是删除服务创建的节点。

ZK实际上是一个比较重的分布式组件，实际上应用没那么多了，所以用ZK实现分布式锁，其实相对也比较少。

5.3 Redis怎么实现分布式锁?

Redis实现分布式锁，是当前应用最广泛的分布式锁实现方式。

Redis执行命令是单线程的，Redis实现分布式锁就是利用这个特性。

实现分布式锁最简单的一个命令：setNx(set if not exist)，如果不存在则更新：

```
setNx resourceName value
```

加锁了之后如果机器宕机，那我这个锁就无法释放，所以需要加入过期时间，而且过期时间需要和setNx同一个原子操作，在Redis2.8之前需要用lua脚本，但是redis2.8之后redis支持nx和ex操作是同一原子操作。

```
set resourceName value ex 5 nx
```

- **Redission**

当然，一般生产中都是使用Redission客户端，非常良好地封装了分布式锁的api，而且支持RedLock。

最近整理了一份牛逼的学习资料，包括但不限于Java基础部分（JVM、Java集合框架、多线程），还囊括了 **数据库、计算机网络、算法与数据结构、设计模式、框架类Spring、Netty、微服务（Dubbo，消息队列） 网关** 等等等等……详情戳：可以说是2022年全网最全的学习和找工作的PDF资源了

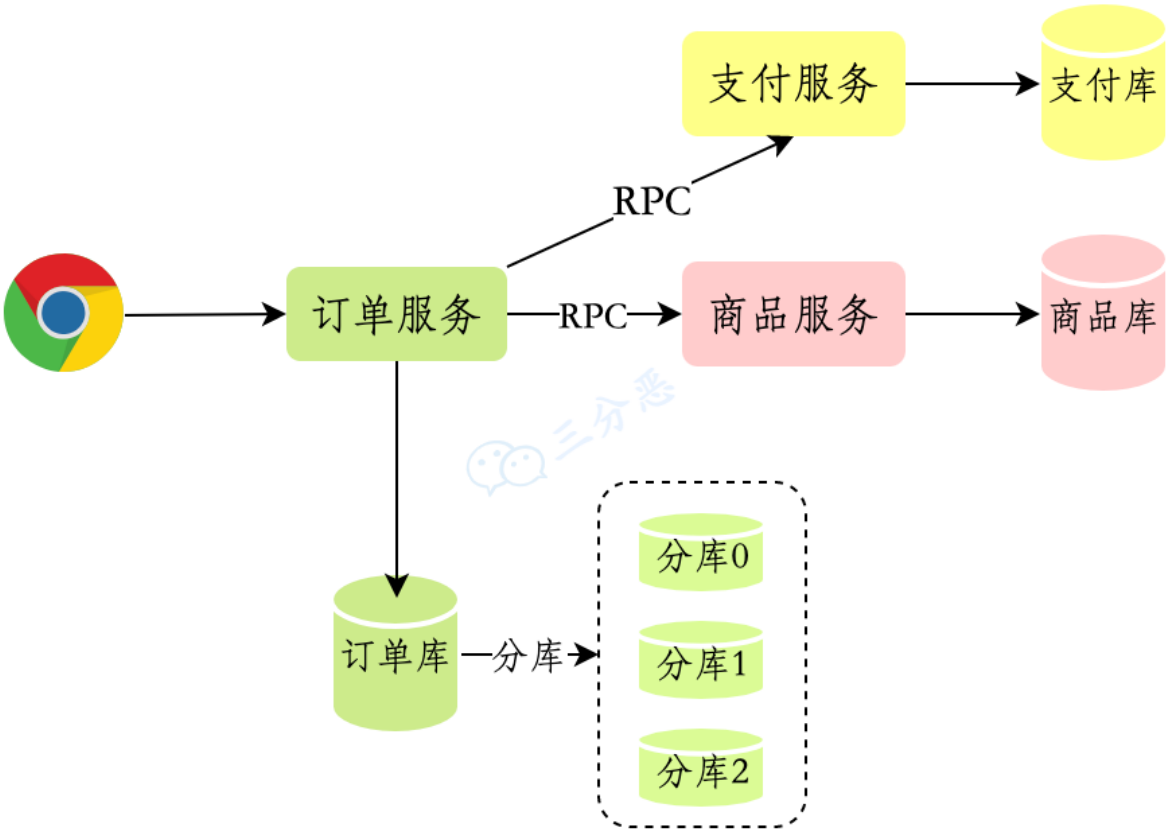
微信搜 **沉默王二** 或扫描下方二维码关注二哥的原创公众号沉默王二，回复 **111** 即可免费领取。

分布式事务

6.什么是分布式事务?

分布式事务是相对本地事务而言的，对于本地事务，利用数据库本身的事务机制，就可以保证事务的ACID特性。

而在分布式环境下，会涉及到多个数据库。



分布式事务其实就是将对同一库事务的概念扩大到了对多个库的事务。目的是为了保证分布式系统中的数据一致性。

分布式事务处理的关键是：

1. 需要记录事务在任何节点所做的所有动作；
2. 事务进行的所有操作要么全部提交，要么全部回滚。

7.分布式事务有哪些常见的实现方案?

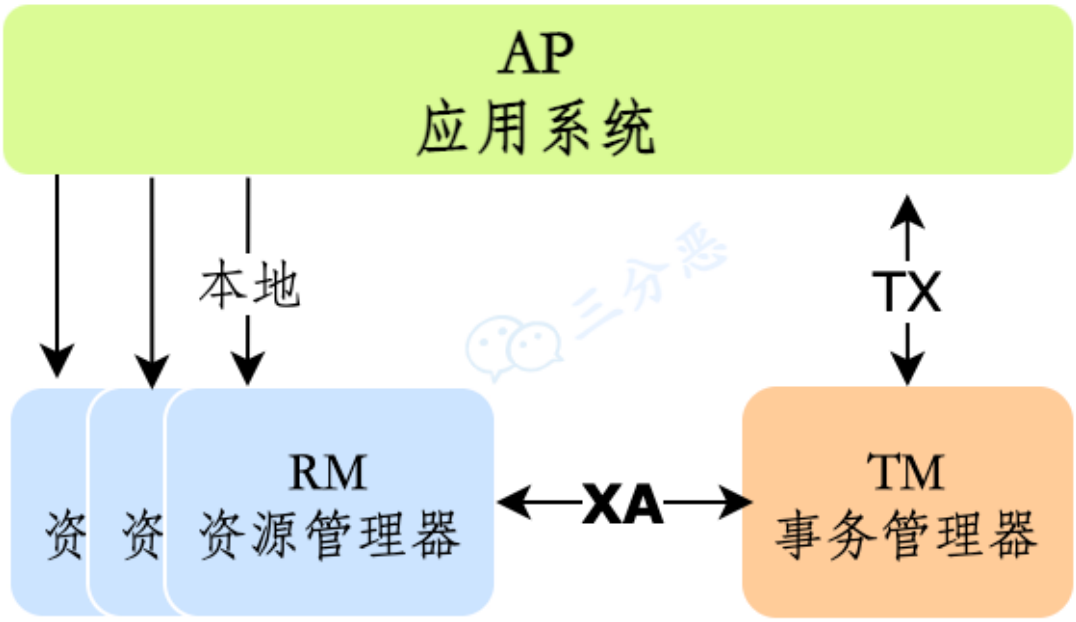
分布式常见的实现方案有 **2PC**、**3PC**、**TCC**、**本地消息表**、**MQ消息事务**、**最大努力通知**、**SAGA事务** 等等。

7.1 说说2PC两阶段提交?

说到2PC，就不得先说分布式事务中的 XA 协议。

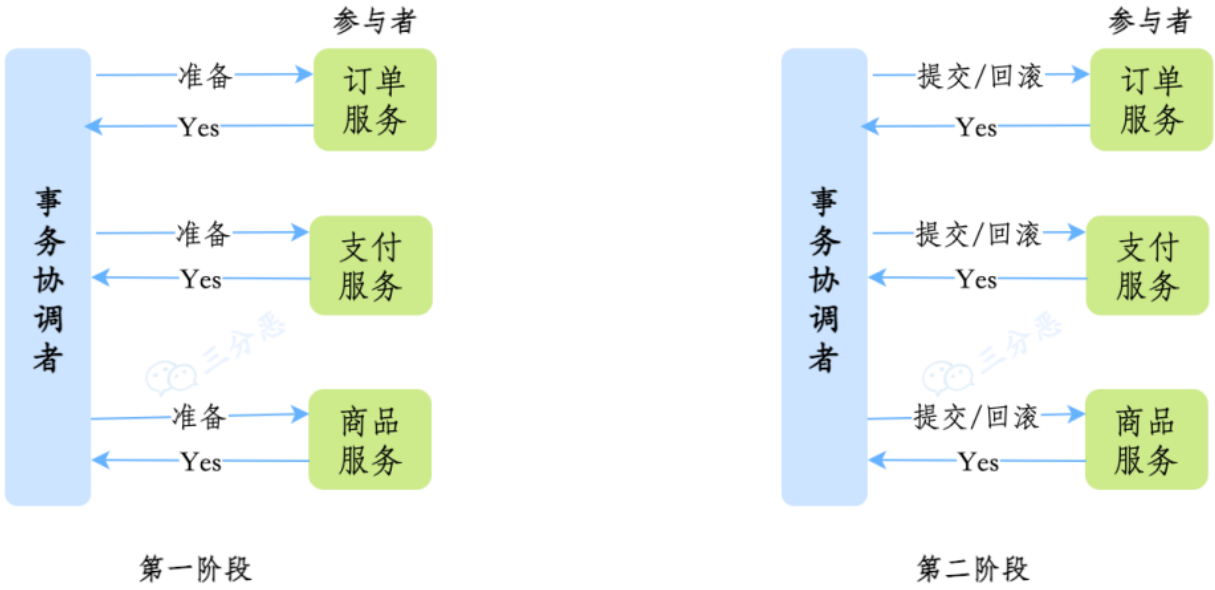
在这个协议里，有三个角色：

- **AP（Application）**：应用系统（服务）
- **TM（Transaction Manager）**：事务管理器（全局事务管理）
- **RM（Resource Manager）**：资源管理器（数据库）



XA协议采用**两阶段提交**方式来管理分布式事务。XA接口提供资源管理器与事务管理器之间进行通信的标准接口。

两阶段提交的思路可以概括为：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情况决定各参与者是否要提交操作还是回滚操作。



- 准备阶段：事务管理器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交
- 提交阶段：事务协调器要求每个数据库提交数据，或者回滚数据。

优点：尽量保证了数据的强一致，实现成本较低，在各大主流数据库都有自己实现，对于MySQL是从5.5开始支持。

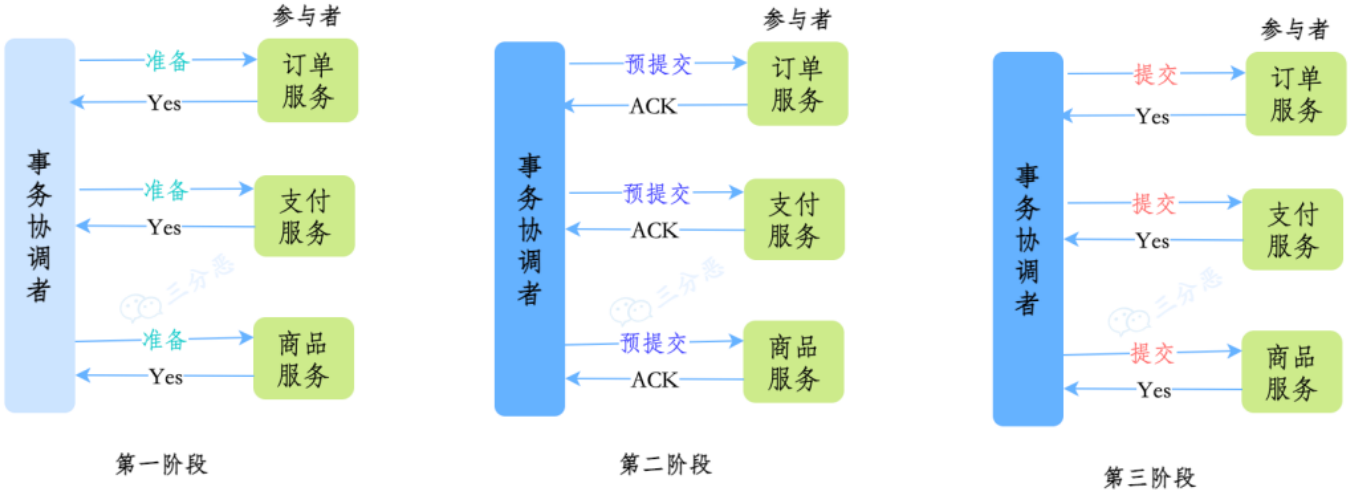
缺点:

- 单点问题：事务管理器在整个流程中扮演的角色很关键，如果其宕机，比如在第一阶段已经完成，在第二阶段正准备提交的时候事务管理器宕机，资源管理器就会一直阻塞，导致数据库无法使用。
- 同步阻塞：在准备就绪之后，资源管理器中的资源一直处于阻塞，直到提交完成，释放资源。
- 数据不一致：两阶段提交协议虽然为分布式数据强一致性所设计，但仍然存在数据不一致性的可能，比如在第二阶段中，假设协调者发出了事务commit的通知，但是因为网络问题该通知仅被一部分参与者所收到并执行了commit操作，其余的参与者则因为没有收到通知一直处于阻塞状态，这时候就产生了数据的不一致性。

7.2 3PC（三阶段提交）了解吗?

三阶段提交（`3PC`）是二阶段提交（`2PC`）的一种改进版本 ，为解决两阶段提交协议的单点故障和同步阻塞问题。

三阶段提交有这么三个阶段：`CanCommit`， `PreCommit`， `DoCommit`三个阶段



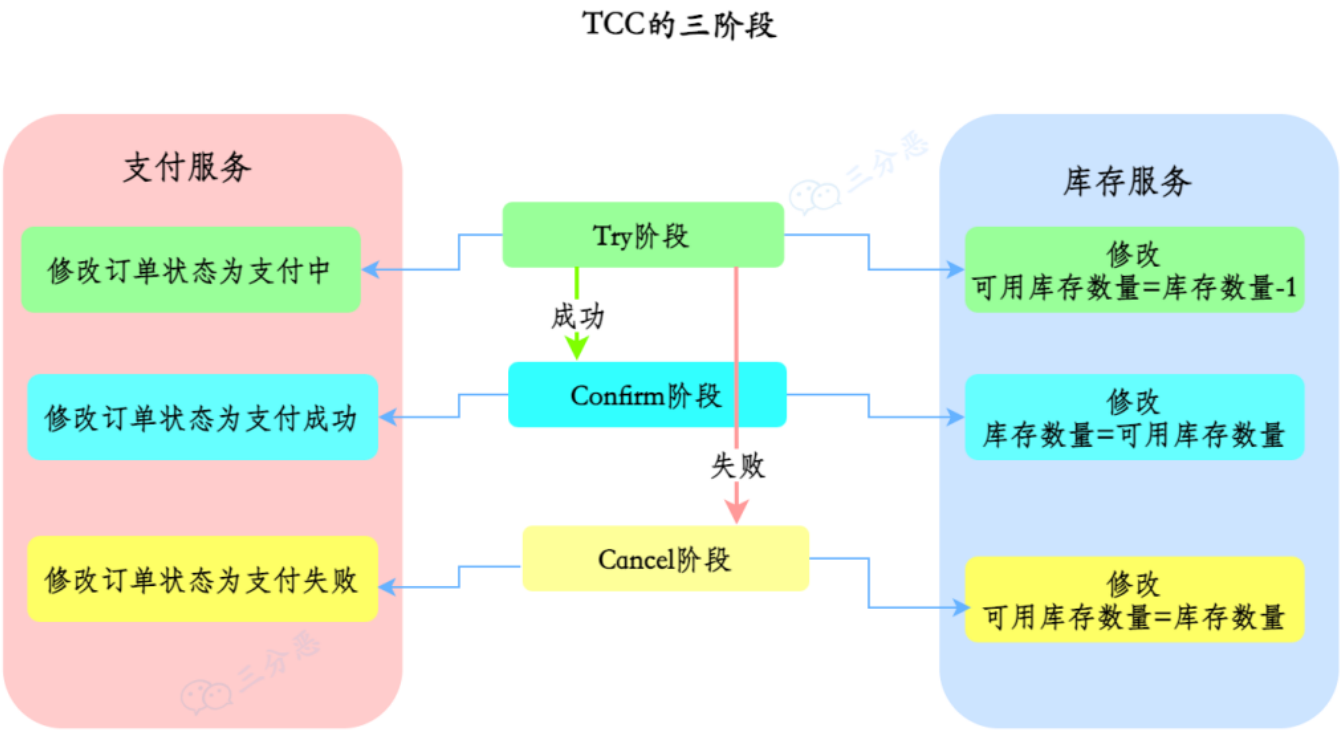
- **CanCommit**：准备阶段。协调者向参与者发送commit请求，参与者如果可以提交就返回Yes响应，否则返回No响应。
- **PreCommit**：预提交阶段。协调者根据参与者在**准备阶段**的响应判断是否执行事务还是中断事务，参与者执行完操作之后返回ACK响应，同时开始等待最终指令。
- **DoCommit**：提交阶段。协调者根据参与者在**准备阶段**的响应判断是否执行事务还是中断事务：
- 如果所有参与者都返回正确的`ACK`响应，则提交事务
- 如果参与者有一个或多个参与者收到错误的`ACK`响应或者超时，则中断事务
- 如果参与者无法及时接收到来自协调者的提交或者中断事务请求时，在等待超时之后，会继续进行事务提交

可以看出，三阶段提交解决的只是两阶段提交中**单体故障**和**同步阻塞**的问题，因为加入了超时机制，这里的超时的机制作用于 **预提交阶段** 和 **提交阶段**。如果等待 **预提交请求** 超时，参与者直接回到准备阶段之前。如果等到**提交请求**超时，那参与者就会提交事务了。

无论是2PC还是3PC都不能保证分布式系统中的数据100%一致。

7.3 TCC了解吗?

TCC（Try Confirm Cancel），是两阶段提交的一个变种，针对每个操作，都需要有一个其对应的确认和取消操作，当操作成功时调用确认操作，当操作失败时调用取消操作，类似于二阶段提交，只不过是这里的提交和回滚是针对业务上的，所以基于TCC实现的分布式事务也可以看做是对业务的一种补偿机制。



- **Try**：尝试待执行的业务。订单系统将当前订单状态设置为支付中，库存系统校验当前剩余库存数量是否大于1，然后将可用库存数量设置为库存剩余数量-1，。
- **Confirm**：确认执行业务，如果Try阶段执行成功，接着执行Confirm 阶段，将订单状态修改为支付成功，库存剩余数量修改为可用库存数量。
- **Cancel**：取消待执行的业务，如果Try阶段执行失败，执行Cancel 阶段，将订单状态修改为支付失败，可用库存数量修改为库存剩余数量。

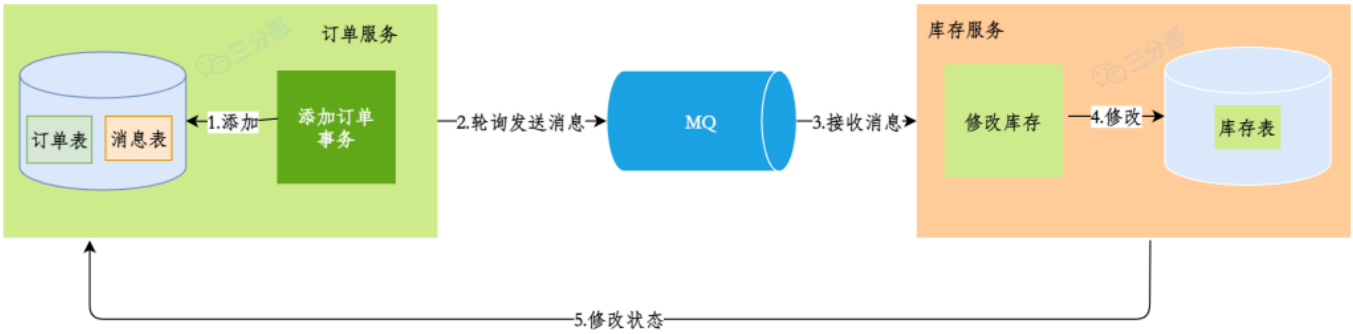
TCC 是业务层面的分布式事务，保证最终一致性，不会一直持有资源的锁。

- **优点：** 把数据库层的二阶段提交交给应用层来实现，规避了数据库的 2PC 性能低下问题
- **缺点**：TCC 的 Try、Confirm 和 Cancel 操作功能需业务提供，开发成本高。TCC 对业务的侵入较大和业务紧耦合，需要根据特定的场景和业务逻辑来设计相应的操作

7.4 本地消息表了解吗?

本地消息表的核心思想是将分布式事务拆分成本地事务进行处理。

例如，可以在订单库新增一个消息表，将新增订单和新增消息放到一个事务里完成，然后通过轮询的方式去查询消息表，将消息推送到MQ，库存服务去消费MQ。



执行流程：

1. 订单服务，添加一条订单和一条消息，在一个事务里提交
2. 订单服务，使用定时任务轮询查询状态为未同步的消息表，发送到MQ，如果发送失败，就重试发送
3. 库存服务，接收MQ消息，修改库存表，需要保证幂等操作
4. 如果修改成功，调用rpc接口修改订单系统消息表的状态为已完成或者直接删除这条消息
5. 如果修改失败，可以不做处理，等待重试

订单服务中的消息有可能由于业务问题会一直重复发送，所以为了避免这种情况可以记录一下发送次数，当达到次数限制之后报警，人工接入处理；库存服务需要保证幂等，避免同一条消息被多次消费造成数据不一致。

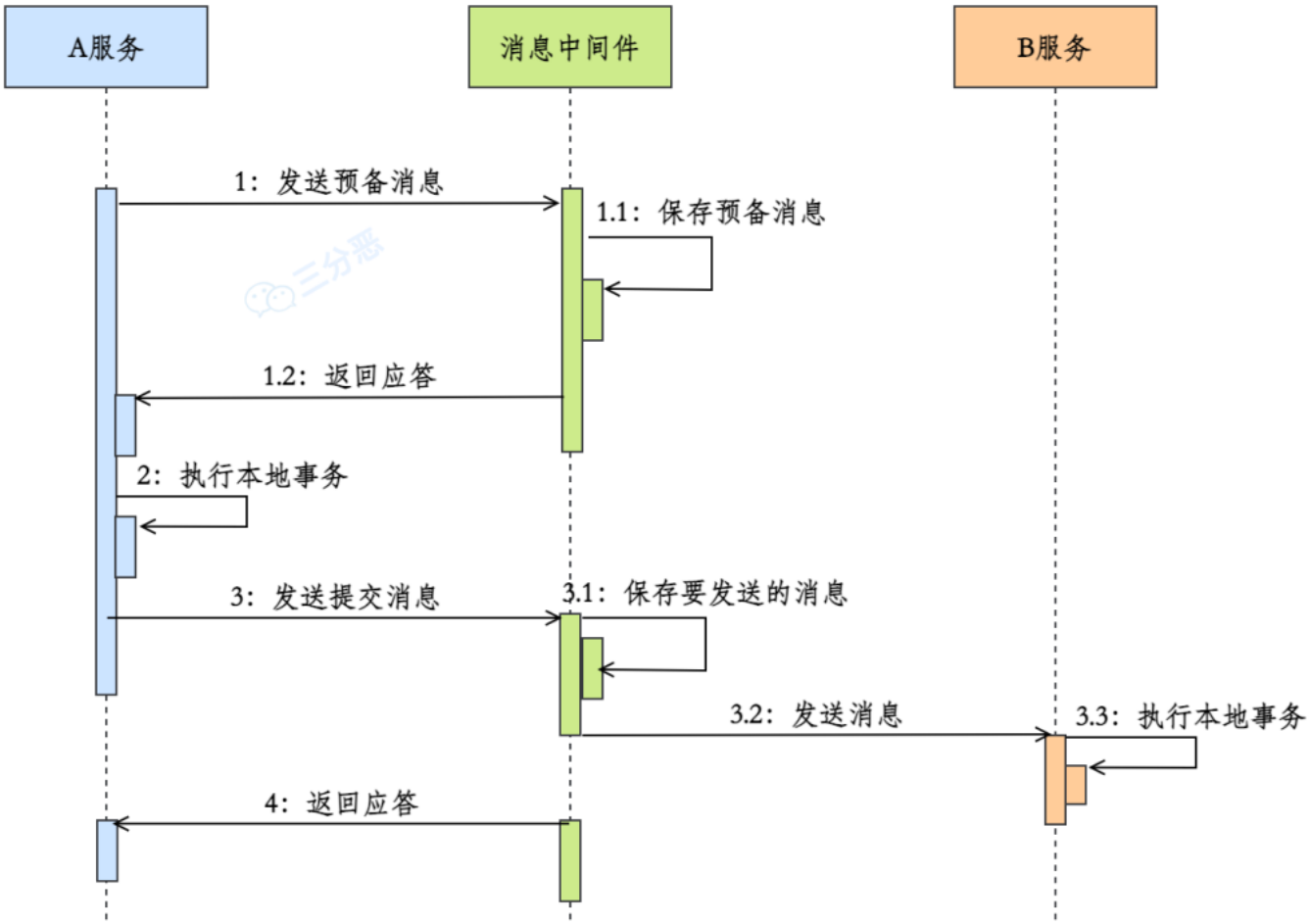
本地消息表这种方案实现了最终一致性，需要在业务系统里增加消息表，业务逻辑中多一次插入的DB操作，所以性能会有损耗，而且最终一致性的间隔主要有定时任务的间隔时间决定

7.5 MQ消息事务了解吗?

消息事务的原理是**将两个事务通过消息中间件进行异步解耦**。

订单服务执行自己的本地事务，并发送MQ消息，库存服务接收消息，执行自己的本地事务，乍一看，好像跟本地消息表的实现方案类似，只是省去 了对本地消息表的操作和轮询发送MQ的操作，但实际上两种方案的实现是不一样的。

消息事务一定要保证业务操作与消息发送的一致性，如果业务操作成功，这条消息也一定投递成功。



执行流程：

1. 发送prepare消息到消息中间件
2. 发送成功后，执行本地事务
3. 如果事务执行成功，则commit，消息中间件将消息下发至消费端
4. 如果事务执行失败，则回滚，消息中间件将这条prepare消息删除
5. 消费端接收到消息进行消费，如果消费失败，则不断重试

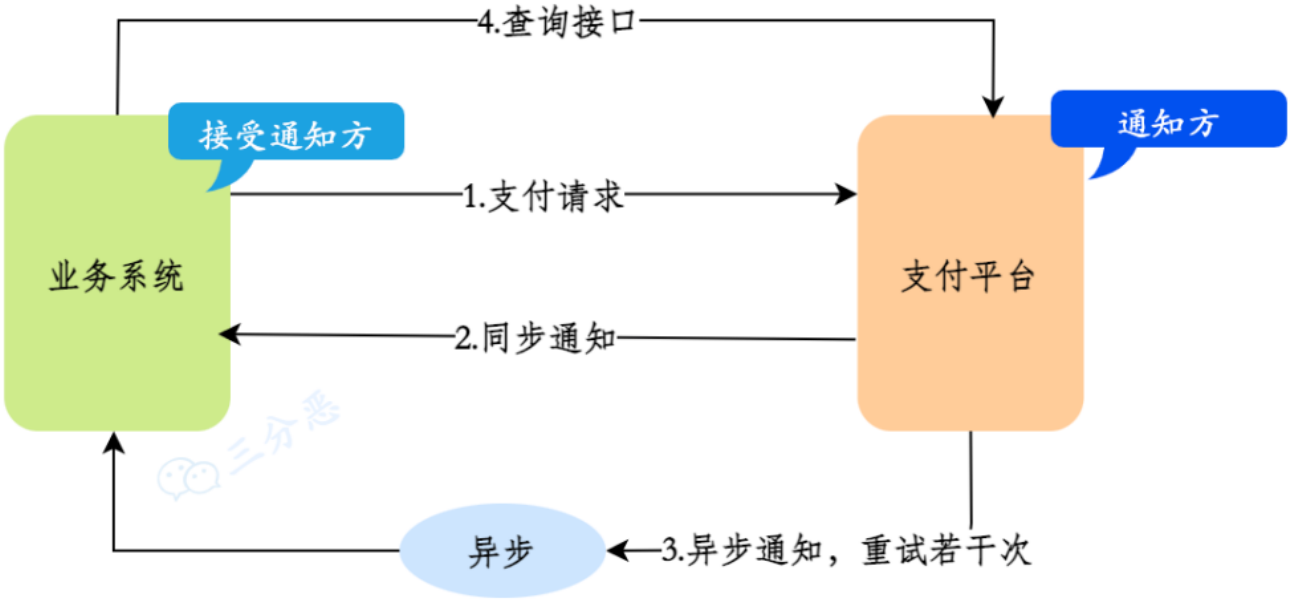
消息事务依赖于消息中间件的事务消息，例如我们熟悉的RocketMQ就支持事务消息（半消息），也就是只有收到发送方确定才会正常投递的消息。

这种方案也是实现了最终一致性，对比本地消息表实现方案，不需要再建消息表，对性能的损耗和业务的入侵更小。

7.6 最大努力通知了解吗?

最大努力通知相比实现会简单一些，适用于一些对最终一致性实时性要求没那么高的业务，比如支付通知，短信通知。

以支付通知为例，业务系统调用支付平台进行支付，支付平台进行支付，进行操作支付之后支付平台会去同步通知业务系统支付操作是否成功，如果不成功，会一直异步重试，但是会有一个最大通知次数，如果超过这个次数后还是通知失败，就不再通知，业务系统自行调用支付平台提供一个查询接口，供业务系统进行查询支付操作是否成功。



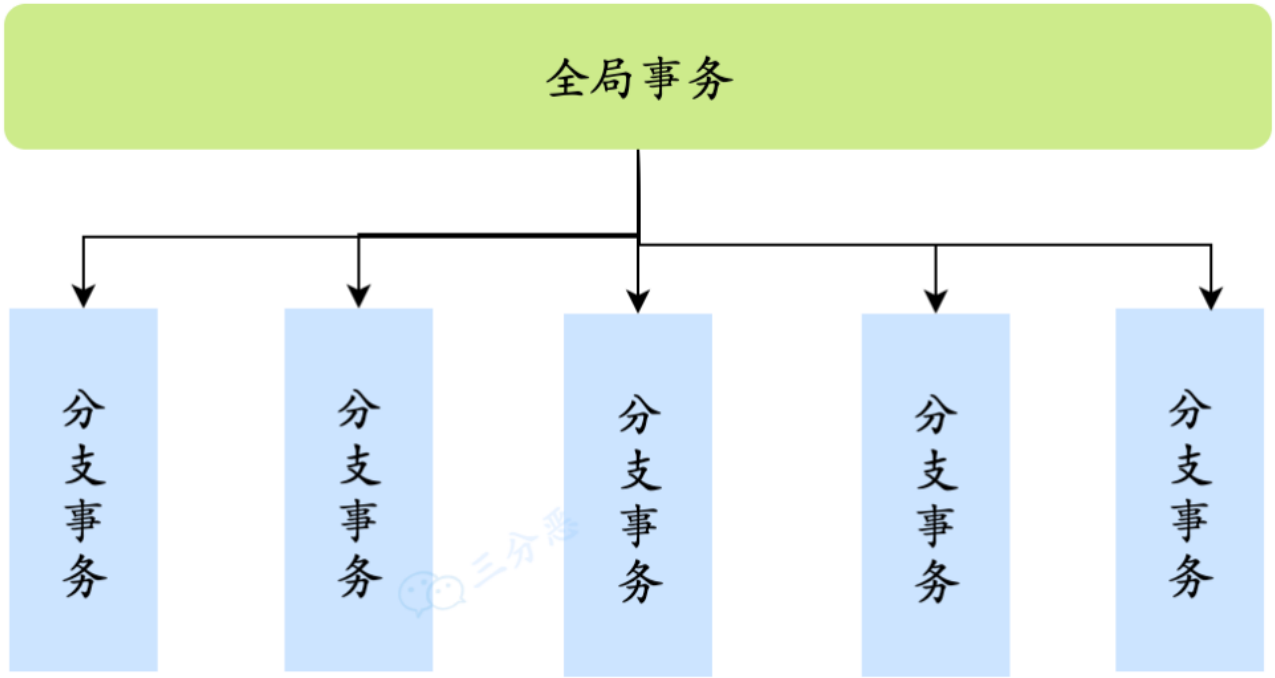
执行流程：

1. 业务系统调用支付平台支付接口， 并在本地进行记录，支付状态为支付中
2. 支付平台进行支付操作之后，无论成功还是失败，同步给业务系统一个结果通知
3. 如果通知一直失败则根据重试规则异步进行重试，达到最大通知次数后，不再通知
4. 支付平台提供查询订单支付操作结果接口
5. 业务系统根据一定业务规则去支付平台查询支付结果

8.你们用什么？能说一下Seata吗?

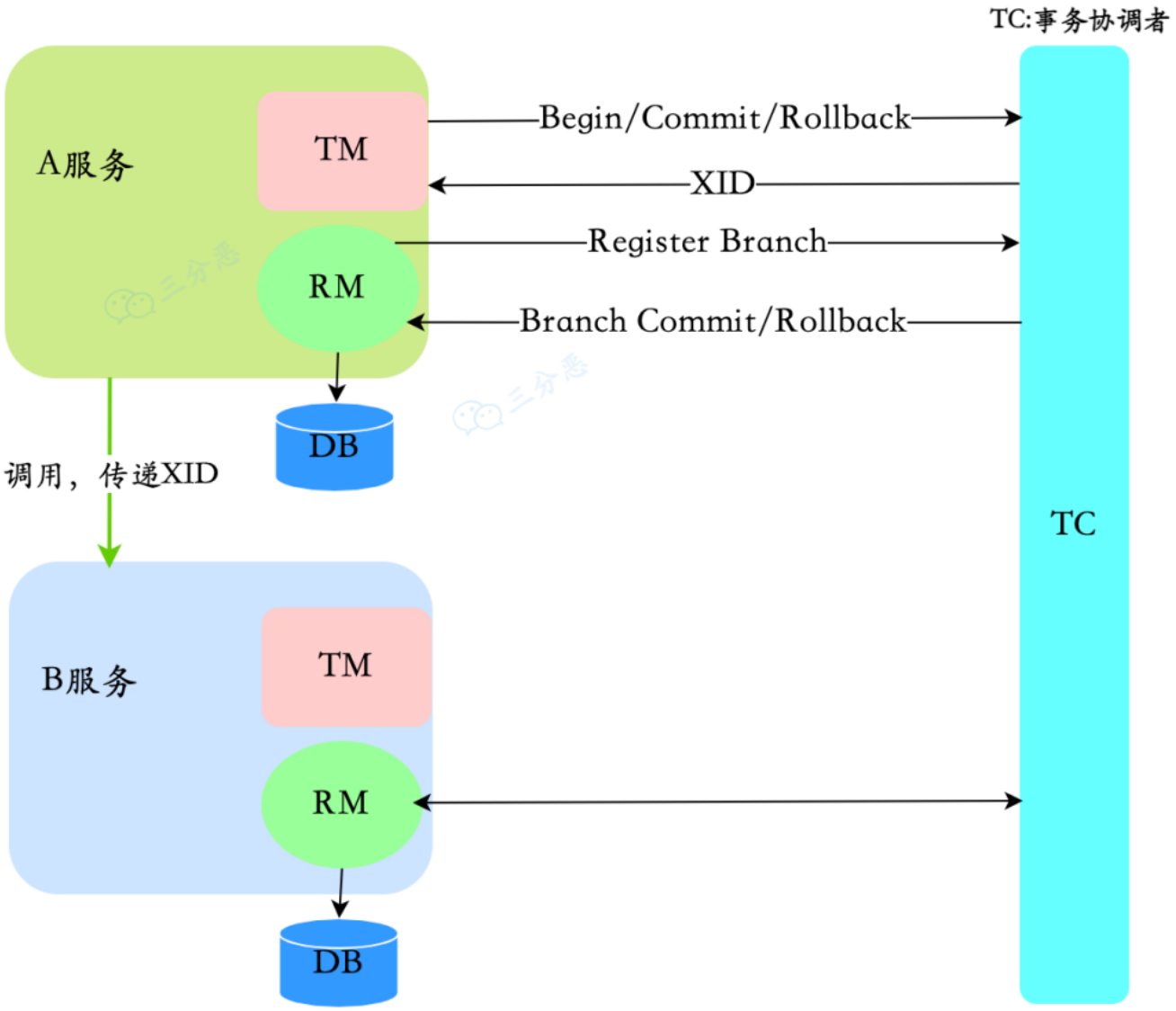
我们用比较常用的是Seata——自己去实现分布式事务调度还是比较麻烦的。

Seata 的设计目标是对业务无侵入，因此它是从业务无侵入的两阶段提交（全局事务）着手，在传统的两阶段上进行改进，他把一个分布式事务理解成一个包含了若干分支事务的全局事务。而全局事务的职责是协调它管理的分支事务达成一致性，要么一起成功提交，要么一起失败回滚。也就是一荣俱荣一损俱损~



Seata 中存在这么几种重要角色：

- **TC（Transaction Coordinator）**：事务协调者。管理全局的分支事务的状态，用于全局性事务的提交和回滚。
- **TM（Transaction Manager）**：事务管理者。用于开启、提交或回滚事务。
- **RM（Resource Manager）**：资源管理器。用于分支事务上的资源管理，向 **TC** 注册分支事务，上报分支事务的状态，接收 **TC** 的命令来提交或者回滚分支事务。



S'eata整体执行流程：

1. 服务A中的 **TM** 向 **TC** 申请开启一个全局事务，**TC** 就会创建一个全局事务并返回一个唯一的 **XID**
2. 服务A中的 **RM** 向 **TC** 注册分支事务，然后将这个分支事务纳入 **XID** 对应的全局事务管辖中
3. 服务A开始执行分支事务
4. 服务A开始远程调用B服务，此时 **XID** 会根据调用链传播
5. 服务B中的 **RM** 也向 **TC** 注册分支事务，然后将这个分支事务纳入 **XID** 对应的全局事务管辖中
6. 服务B开始执行分支事务
7. 全局事务调用处理结束后，**TM** 会根据有误异常情况，向 **TC** 发起全局事务的提交或回滚
8. **TC** 协调其管辖之下的所有分支事务，决定是提交还是回滚

分布式一致性算法

9.分布式算法paxos了解么？

`Paxos` 有点类似前面说的 `2PC`，`3PC`，但比这两种算法更加完善。在很多多大厂都得到了工程实践，比如阿里的 `OceanBase` 的 **分布式数据库**， `Google` 的 `chubby` **分布式锁** 。

Paxos算法是什么?

`Paxos` 算法是 **基于消息传递** 且具有 **高效容错特性** 的一致性算法，目前公认的解决 **分布式一致性问题** 最有效的算法之一。

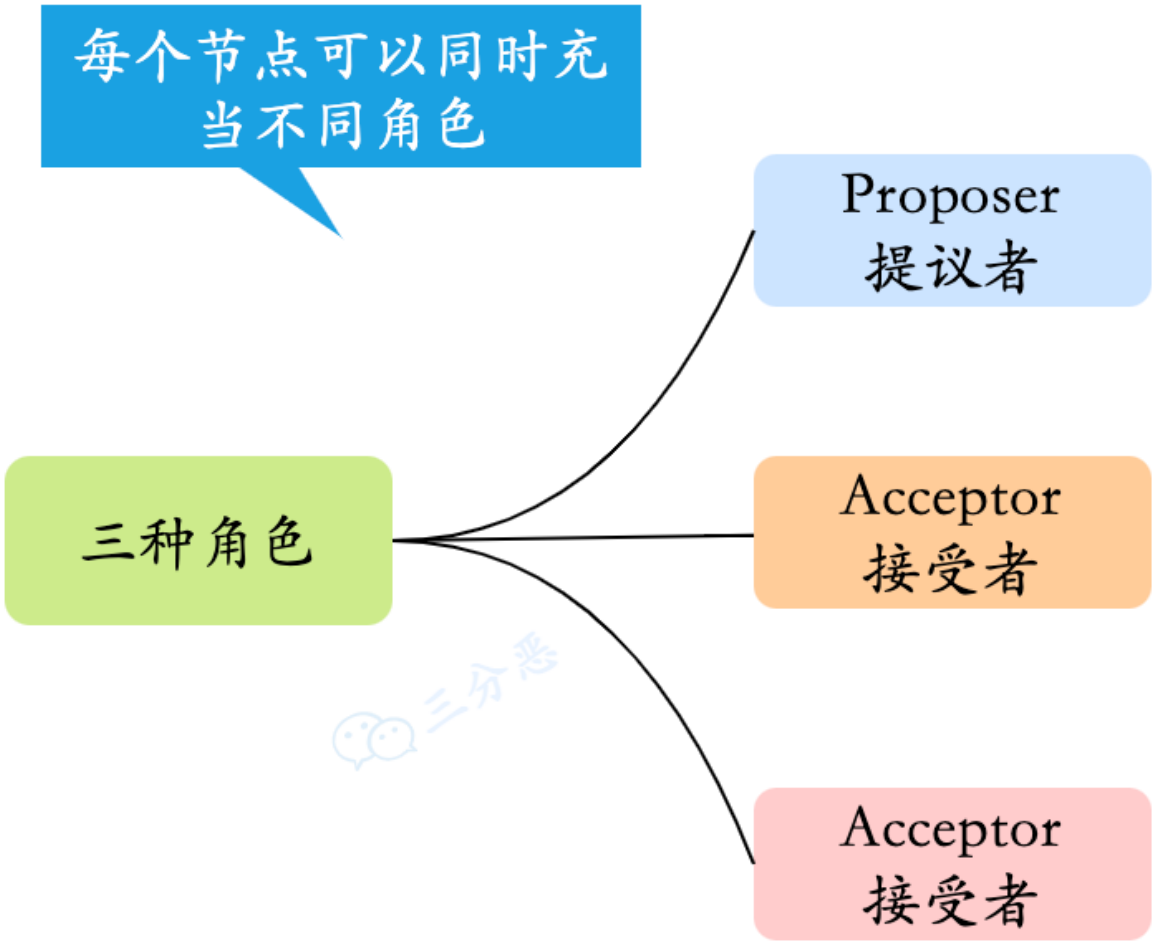
Paxos算法的工作流程?

角色

在Paxos中有这么几个角色：

1. **Proposer（提议者）**：提议者提出提案，用于投票表决。
2. **Accecptor（接受者）**：对提案进行投票，并接受达成共识的提案。
3. **Learner（学习者）**：被告知投票的结果，接受达成共识的提案。

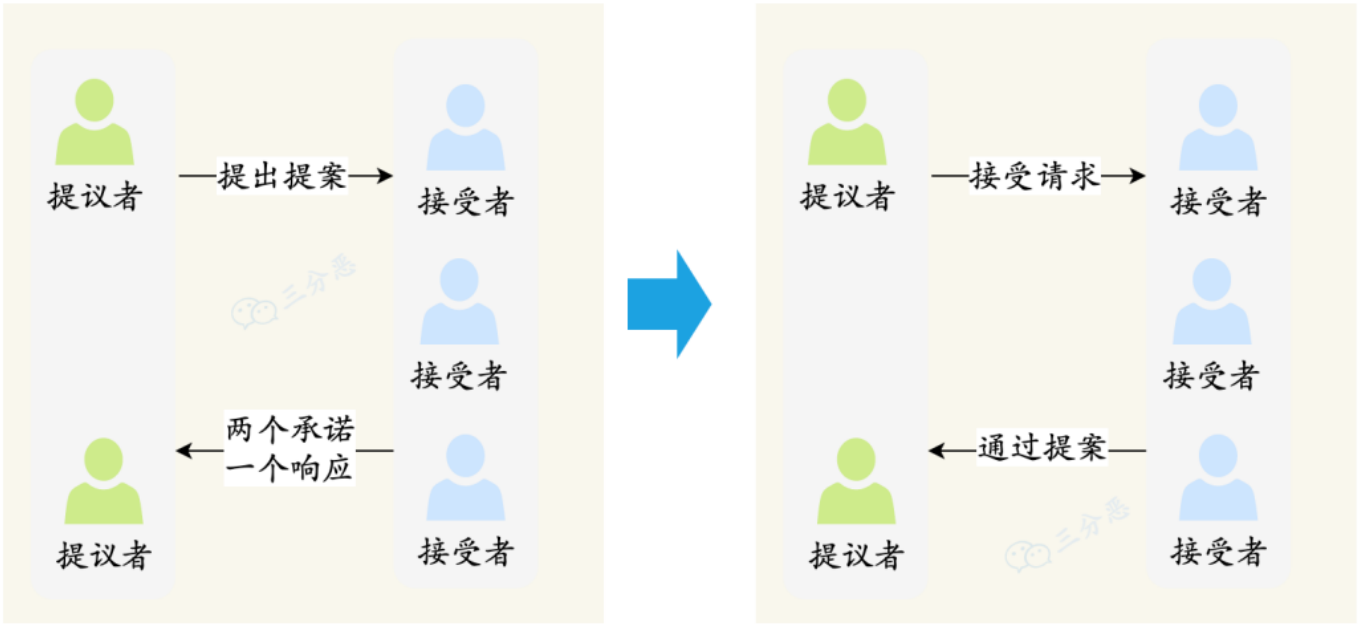
在实际中，一个节点可以同时充当不同角色。



提议者提出提案，提案=编号+value，可以表示为[M,V]，每个提案都有唯一编号，而且编号的大小是趋势递增的。

算法流程

Paxos算法包含两个阶段，第一阶段 **Prepare(准备)** 、第二阶段 **Accept(接受)** 。



第一阶段：准备阶段　　第二阶段：接受阶段

Prepare(准备)阶段

1. 提议者提议一个新的提案 P[Mn,?]，然后向接受者的某个超过半数的子集成员发送编号为Mn的准备请求
2. 如果一个接受者收到一个编号为Mn的准备请求，并且编号Mn大于它已经响应的所有准备请求的编号，那么它就会将它已经批准过的最大编号的提案作为响应反馈给提议者，同时该接受者会承诺不会再批准任何编号小于Mn的提案。

总结一下，接受者在收到提案后，会给与提议者**两个承诺**与**一个应答**：

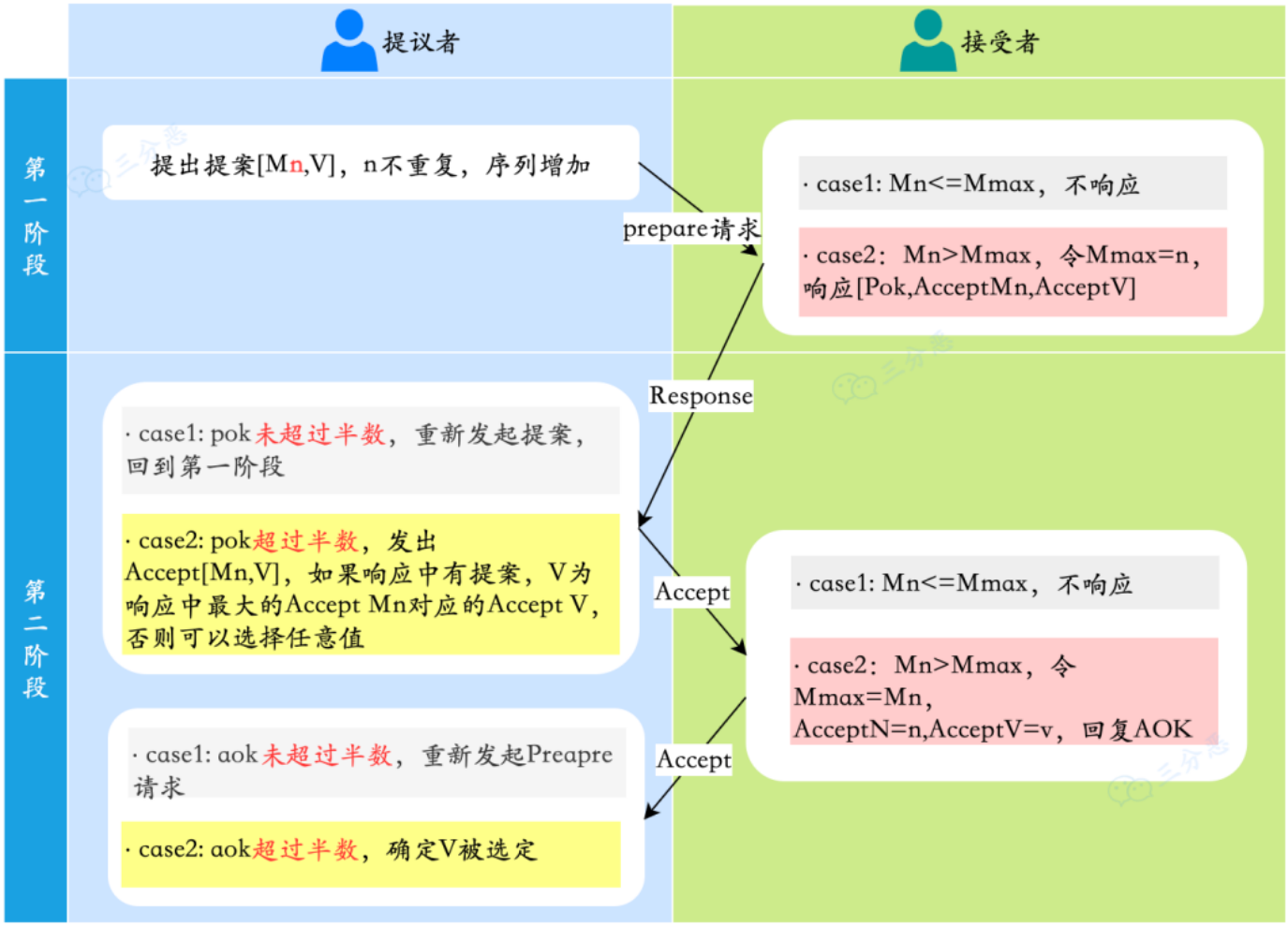
- 两个承诺：
- 承诺不会再接受提案号小于或等于 Mn 的 Prepare 请求
- 承诺不会再接受提案号小于Mn 的 Accept 请求
- 一个应答：
- 不违背以前作出的承诺的前提下，回复已经通过的提案中提案号最大的那个提案所设定的值和提案号Mmax，如果这个值从来没有被任何提案设定过，则返回空值。如果不满足已经做出的承诺，即收到的提案号并不是决策节点收到过的最大的，那允许直接对此 Prepare 请求不予理会。

Accept(接受)阶段

1. 如果提议者收到来自半数以上的接受者对于它发出的编号为Mn的准备请求的响应，那么它就会发送一个针对[Mn,Vn]的接受请求给接受者，注意Vn的值就是收到的响应中编号最大的提案的值，如果响应中不包含任何提案，那么它可以随意选定一个值。
2. 如果接受者收到这个针对[Mn,Vn]提案的接受请求，只要该接受者尚未对编号大于Mn的准备请求做出响应，它就可以通过这个提案。

当提议者收到了多数接受者的接受应答后，协商结束，共识决议形成，将形成的决议发送给所有学习节点进行学习。

所以Paxos算法的整体详细流程如下：



Paxos算法有什么缺点吗？怎么优化？

前面描述的可以称之为Basic Paxos 算法，在单提议者的前提下是没有问题的，但是假如有多个提议者互不相让，那么就可能导致整个提议的过程进入了死循环。

Lamport 提出了 Multi Paxos 的算法思想。

Multi Paxos算法思想，简单说就是在多个提议者的情况下，选出一个Leader（领导者），由领导者作为唯一的提议者，这样就可以解决提议者冲突的问题。

10.说说Raft算法?

Raft算法是什么?

`Raft` 也是一个 **一致性算法**，和 `Paxos` 目标相同。但它还有另一个名字 - **易于理解的一致性算法**。`Paxos` 和 `Raft` 都是为了实现 **一致性** 产生的。这个过程如同选举一样，**参选者** 需要说服 **大多数选民** (Server) 投票给他，一旦选定后就跟随其操作。`Paxos` 和 `Raft` 的区别在于选举的 **具体过程** 不同。

Raft算法的工作流程?

Raft算法的角色

`Raft` 协议将 `Server` 进程分为三种角色：

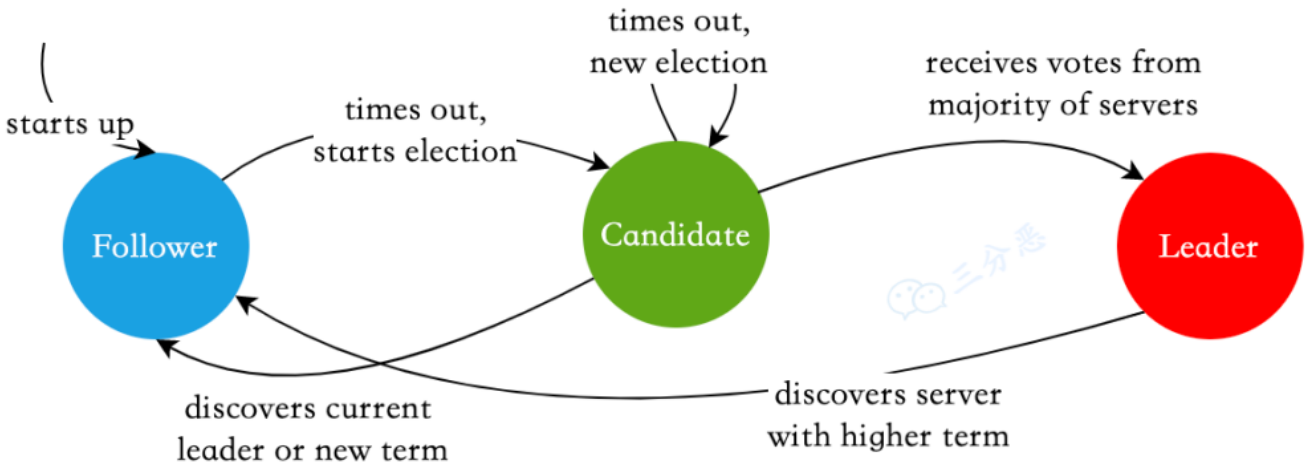
- **Leader（领导者）**
- **Follower（跟随者）**
- **Candidate（候选人）**

就像一个民主社会，领导者由跟随者投票选出。刚开始没有 **领导者**，所有集群中的 **参与者** 都是 **跟随者**。

那么首先开启一轮大选。在大选期间 **所有跟随者** 都能参与竞选，这时所有跟随者的角色就变成了 **候选人**，民主投票选出领袖后就开始了这届领袖的任期，然后选举结束，所有除 **领导者** 的 **候选人** 又变回 **跟随者** 服从领导者领导。

这里提到一个概念 **「任期」**，用术语 `Term` 表达。

三类角色的变迁图如下：

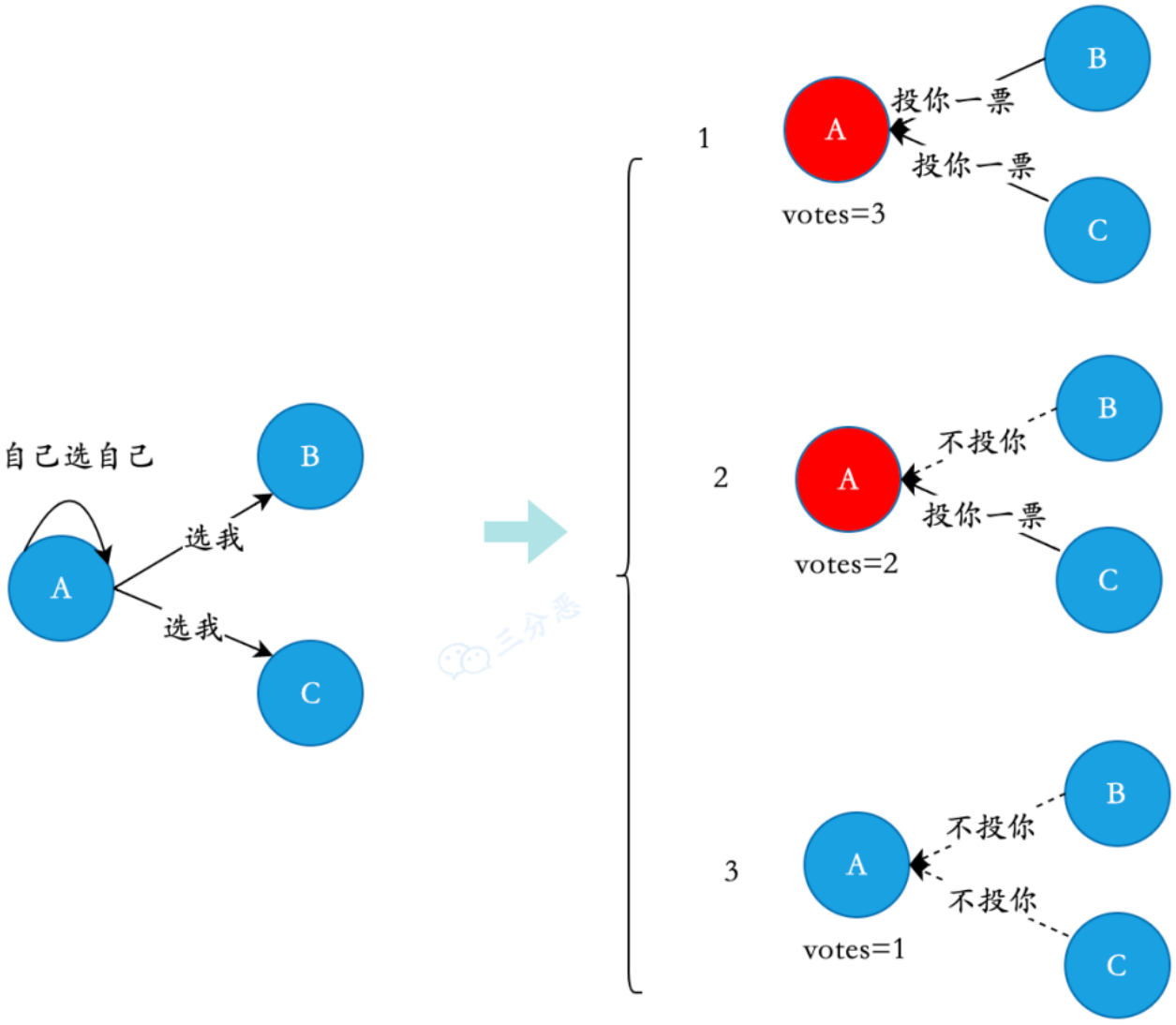


Leader选举过程

Raft 使用心跳（heartbeat）触发Leader选举。当Server启动时，初始化为Follower。Leader向所有Followers周期性发送heartbeat。如果Follower在选举超时时间内没有收到Leader的heartbeat，就会等待一段随机的时间后发起一次Leader选举。

Follower将其当前term加一然后转换为Candidate。它首先给自己投票并且给集群中的其他服务器发送 RequestVote RPC 。结果有以下三种情况：

- 赢得了多数（超过1/2）的选票，成功选举为Leader；
- 收到了Leader的消息，表示有其它服务器已经抢先当选了Leader；
- 没有Server赢得多数的选票，Leader选举失败，等待选举时间超时（`Election Timeout`）后发起下一次选举。



选出 `Leader` 后，`Leader` 通过 **定期** 向所有 `Follower` 发送 **心跳信息** 维持其统治。若 `Follower` 一段时间未收到 `Leader` 的 **心跳**，则认为 `Leader` 可能已经挂了，然后再次发起 **选举** 过程。

最近整理了一份牛逼的学习资料，包括但不限于Java基础部分（JVM、Java集合框架、多线程），还囊括了 **数据库、计算机网络、算法与数据结构、设计模式、框架类Spring、Netty、微服务（Dubbo，消息队列） 网关** 等等等等.....详情戳：可以说是2022年全网最全的学习和找工作的PDF资源了

微信搜 **沉默王二** 或扫描下方二维码关注二哥的原创公众号沉默王二，回复 **111** 即可免费领取。

分布式设计

11.说说什么是幂等性?

> 什么是幂等性?

幂等性是一个数学概念，用在接口上：用在接口上就可以理解为：**同一个接口，多次发出同一个请求，请求的结果是一致的。**

简单说，就是多次调用如一次。

> 什么是幂等性问题?

在系统的运行中，可能会出现这样的问题：

1. 用户在填写某些`form表单`时，保存按钮不小心快速点了两次，表中竟然产生了两条重复的数据，只是id不一样。
2. 开发人员在项目中为了解决`接口超时`问题，通常会引入了`重试机制`。第一次请求接口超时了，请求方没能及时获取返回结果（此时有可能已经成功了），于是会对该请求重试几次，这样也会产生重复的数据。
3. mq消费者在读取消息时，有时候会读取到`重复消息`，也会产生重复的数据。

这些都是常见的幂等性问题。

在分布式系统里，只要下游服务有写（保存、更新）的操作，都有可能会产生幂等性问题。

PS:幂等和防重有些不同，防重强调的防止数据重复，幂等强调的是多次调用如一次，防重包含幂等。

怎么保证接口幂等性?

1. insert前先select

在保存数据的接口中，在`insert`前，先根据`requestId`等字段先`select`一下数据。如果该数据已存在，则直接返回，如果不存在，才执行 `insert`操作。

2. 加唯一索引

加唯一索引是个非常简单但很有效的办法，如果重复插入数据的话，就会抛出异常，为了保证幂等性，一般需要捕获这个异常。

如果是`java`程序需要捕获：`DuplicateKeyException`异常，如果使用了`spring`框架还需要捕获：`MySQLIntegrityConstraintViolationException`异常。

3. 加悲观锁

更新逻辑，比如更新用户账户余额，可以加悲观锁，把对应用户的哪一行数据锁住。同一时刻只允许一个请求获得锁，其他请求则等待。

```
select * from user id=123 for update;
```

这种方式有一个缺点，获取不到锁的请求一般只能报失败，比较难保证接口返回相同值。

4. 加乐观锁

更新逻辑，也可以用乐观锁，性能更好。可以在表中增加一个`timestamp`或者`version`字段，例如`version`：

在更新前，先查询一下数据，将version也作为更新的条件，同时也更新version：

```
update user set amount=amount+100,version=version+1 where id=123 and version=1;
```

更新成功后，version增加，重复更新请求进来就无法更新了。

5. 建防重表

有时候表中并非所有的场景都不允许产生重复的数据，只有某些特定场景才不允许。这时候，就可以使用防重表的方式。

例如消息消费中，创建防重表，存储消息的唯一ID，消费时先去查询是否已经消费，已经消费直接返回成功。

6. 状态机

有些业务表是有状态的，比如订单表中有：1-下单、2-已支付、3-完成、4-撤销等状态，可以通过限制状态的流动来完成幂等。

7. 分布式锁

直接在数据库上加锁的做法性能不够友好，可以使用分布式锁的方式，目前最流行的分布式锁实现是通过Redis，具体实现一般都是使用Redission框架。

8. token机制

请求接口之前，需要先获取一个唯一的token，再带着这个token去完成业务操作，服务端根据这个token是否存在，来判断是否是重复的请求。

分布式限流

12.你了解哪些限流算法?

- 计数器

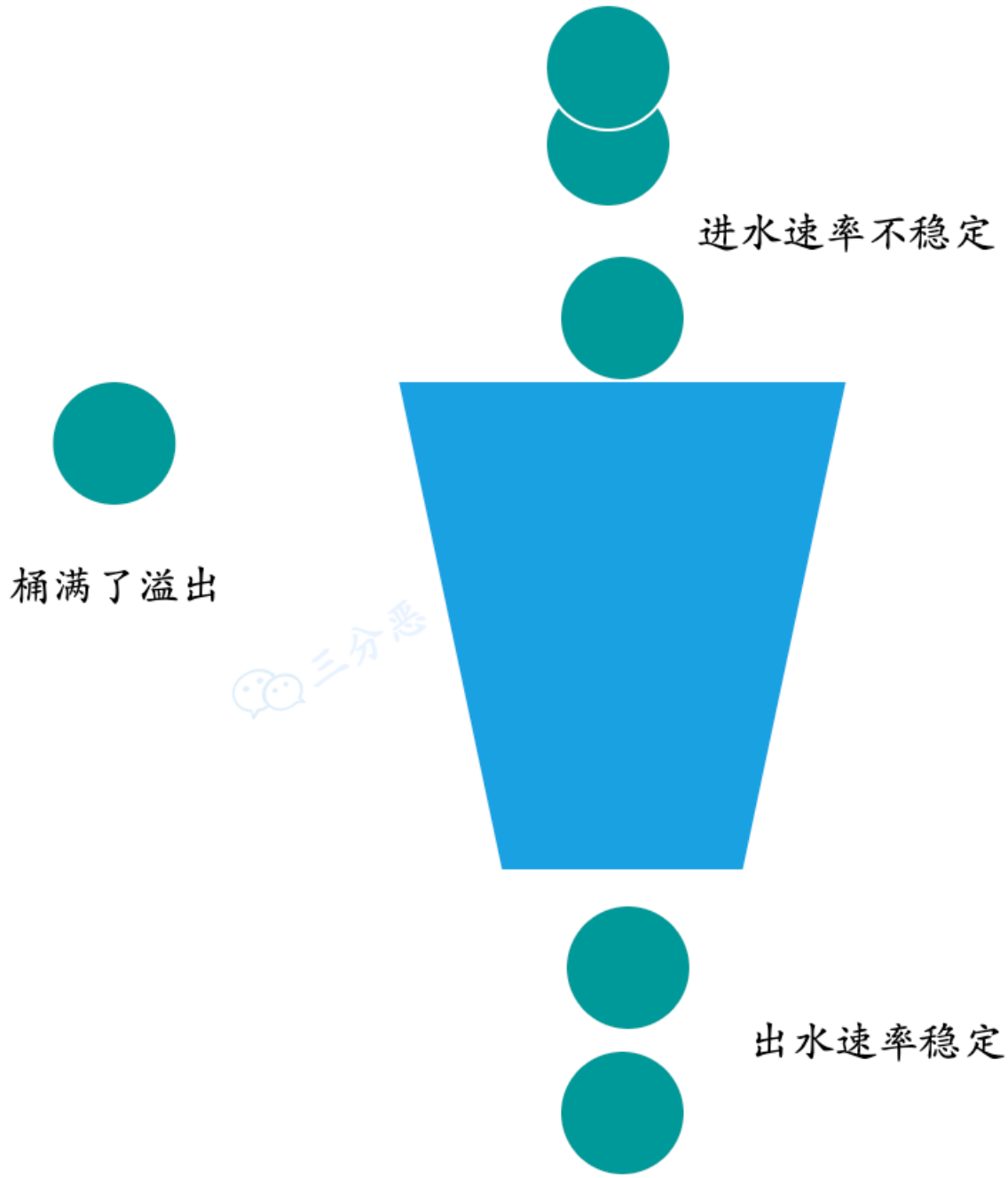
计数器比较简单粗暴，比如我们要限制1s能够通过的请求数，实现的思路就是从第一个请求进来开始计时，在接下来的1s内，每个请求进来请求数就+1，超过最大请求数的请求会被拒绝，等到1s结束后计数清零，重新开始计数。

这种方式有个很大的弊端：比如前10ms已经通过了最大的请求数，那么后面的990ms的请求只能拒绝，这种现象叫做“突刺现象”。

- 漏桶算法

就是桶底出水的速度恒定，进水的速度可能快慢不一，但是当进水量大于出水量的时候，水会被装在桶里，不会直接被丢弃；但是桶也是有容量限制的，当桶装满水后溢出的部分还是会被丢弃的。

算法实现：可以准备一个队列来保存暂时处理不了的请求，然后通过一个线程池定期从队列中获取请求来执行。

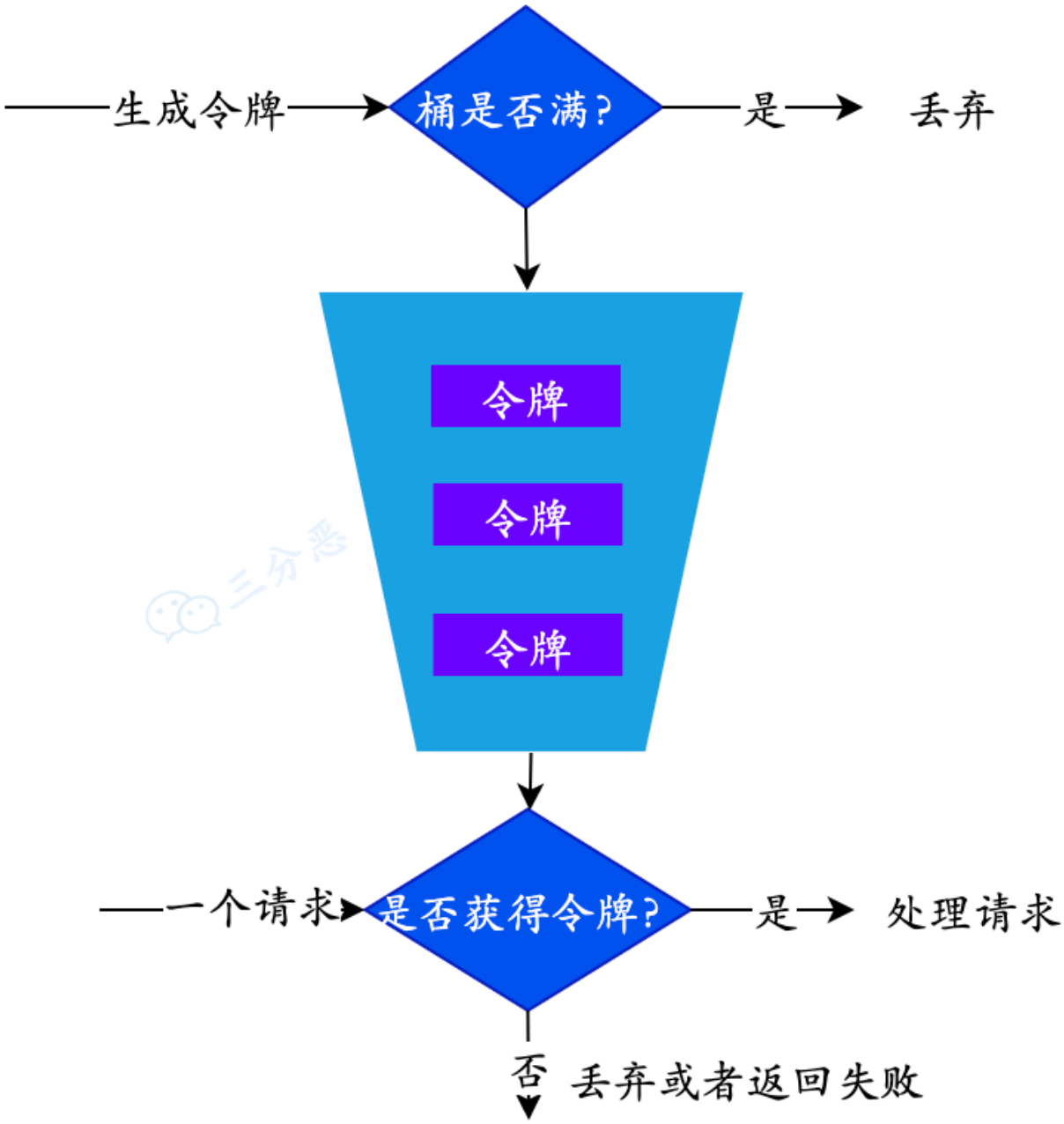


- 令牌桶算法

令牌桶就是生产访问令牌的一个地方，生产的速度恒定，用户访问的时候当桶中有令牌时就可以访问，否则将触发限流。

实现方案：Guava RateLimiter限流

Guava RateLimiter是一个谷歌提供的限流，其基于令牌桶算法，比较适用于单实例的系统。



这一期的分布式面试题就整理到这里了，主要是偏理论的一些问题，分布式其实是个很大的类型，比如分布式调用、分布式治理.....

所以，这篇文章只是个开始，后面还会有分布式调用（RPC）、微服务相关的主题文章，敬请期待。

> 图文详解 12 道分布式面试高频题，这次面试，一定吊打面试官，整理：沉默王二，戳转载链接，作者：三分恶，戳原文链接。

微信搜 **沉默王二** 或扫描下方二维码关注二哥的原创公众号沉默王二，回复 **111** 即可免费领取。