分布式：一致性问题

1.一致性

一致性（consistency），是指对于分布式系统中的多个服务节点，给定一系列操作，**在约定协议的保障下，试图使得它们对处理结果达成"某种程度"的认同。**

分布式计算机集群系统中容易出现以下几个问题：

1. **节点之间的网络通信是不可靠的**，包括**消息延迟、乱序和内容错误**等；
2. 节点的**处理时间无法保障**，结果可能出现错误，甚至节点自身可能发生**宕机**；
3. 同步调用可以简化设计，但会严重降低分布式系统的**可扩展性**，甚至使其退化为单点系统。

对于节点的问题就要涉及到一个关于拆分的问题。拆分一般分为水平拆分和垂直拆分，这并不单指对数据库或者缓存的拆分，主要是表达一种分而治之的思想和逻辑。

水平拆分：指由于单一节点无法满足性能需求，需要扩展为多个节点，多个节点具有一致的功能，组成一个服务池，一个节点服务一部分请求量，所有节点共同处理大规模高并发的请求量。
垂直拆分：指根据功能进行拆分，简单来说就是什么专业的人干什么事，把一个复杂的功能拆分为多个单一、简单的功能，不同的单一功能组合在一起，和未拆分前完成的功能是一样的。由于每个功能职责单一、简单，使得维护和变更都变得更简单、容易、安全，所以更易于产品版本的迭代，还能够快速地进行敏捷发布和上线。
在这样的互联网时代，一致性指分布式服务化系统之间的弱一致性，包括应用系统的一致性和数据的一致性。

无论是水平拆分还是垂直拆分，都解决了特定场景下的特定问题，拆分后的系统或者服务化的系统的最大问题就是一致性问题：如何保证它们的信息、工作进度、状态一致并且协调有序地工作。

2.一致性的问题部分案例

a.下订单和扣库存

电商系统中有一个经典案例，即下订单和扣库存如何保持一致。如果先下订单，扣库存失败，那么将会导致超卖；如果下订单不成功，扣库存成功，那么会导致少卖。这两种都会导致运营成本增加，在严重的情况下需要赔付。也就是说，要将订单和库存的扣除一一对应，保障用户的买卖需求和自身的利益得失。

b.同步和异步调用的超时

1）.**同步**：主要针对的是网络问题导致的系统之间调用超时。对于系统A同步调用系统B超时，A可以得到反馈但是无法确定B是否完成预设，就无法反馈给使用方。

2）.**异步**：使用异步回调返回处理结果，系统A同步调用系统B发起指令，系统B采用受理模式，受理后则返回成功信息，然后系统B处理后异步通知系统A处理结果，那么如果对于A而言没有收到回调结果，那么这两个系统间的状态就不一致，互相认知的状态不同会导致系统间发生错误。

c.缓存问题

1）.**缓存和数据库不一致**

缓存中的数据跟数据库的数据出现了不一致，即其中一方存在脏数据的现象。需要注意的是，只有在对同一条数据并发读写的时候，才可能会出现这种问题。

如果系统并发量很低，特别是读并发很低，那么它发生缓存跟数据库数据不一致的情况相对比较少，概率比较低；
如果系统并发量很高，像淘宝、京东等电商平台，每天都是上亿级流量，每秒并发读是几万，每秒都有写请求，这种情况下出现缓存跟数据库不一致的概率就比较高；

2）.**本地缓存节点间不一致**

一个服务池上的多个节点为了满足较高的性能需求，需要使用本地缓存，这样每个节点都会有一份缓存数据的复制，如果这些数据是静态的、不变的，就永远不会有问题，但是如果这些数据是半静态的或者经常被更新的，则被更新时各个节点的更新是有先后顺序的，在更新的瞬间，在某个时间窗口内各个节点的数据是不一致的，如果这些数据是为某个开关服务的，则想象一下重复的请求进入了不同的节点，一个请求进入了开关打开的逻辑，同时另外一个请求进入了开关关闭的逻辑，会导致请求被处理两次，最坏的情况下是导致资金损失。

3）.**缓存数据结构不一致**

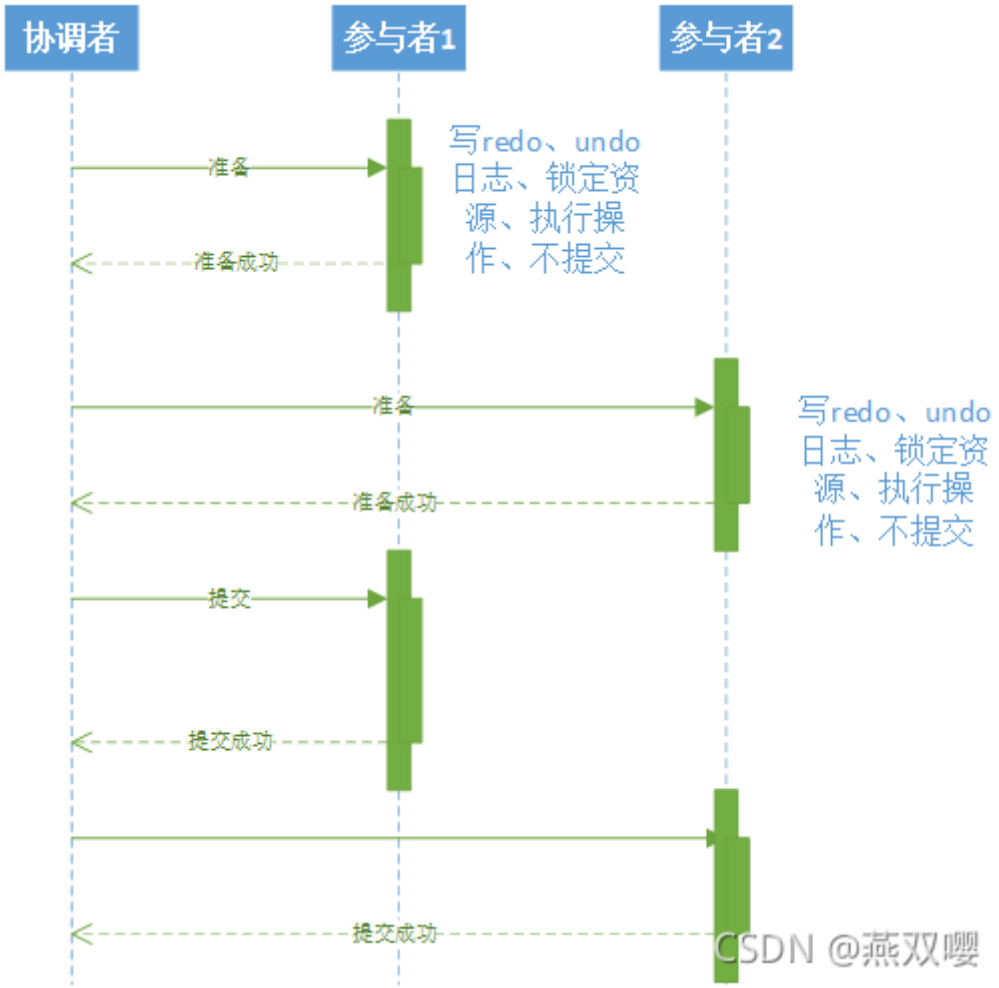
该数据由多个数据元组成，其中，某个数据元素需要从数据库或者服务中获取，如果一部分数据元素获取失败，则由程序处理不正确，仍然将不完全的数据存入缓存中，在缓存使用者使用时很有可能因为数据的不完全而抛出异常

3.解决方法

1.分布式一致性协议

a.**两阶段提交协议**

例如JEE（Java的企业版）的XA协议就是根据两阶段提交来保证事务的完整性，并实现分布式服务化的强一致性（简单的来说就是在任何时刻下，所有节点中的数据都是一样的，任何一次读都能读到某个数据的最近一次写的数据。）。



里面会存在两个角色分别为协调者和参与者（多个？）。

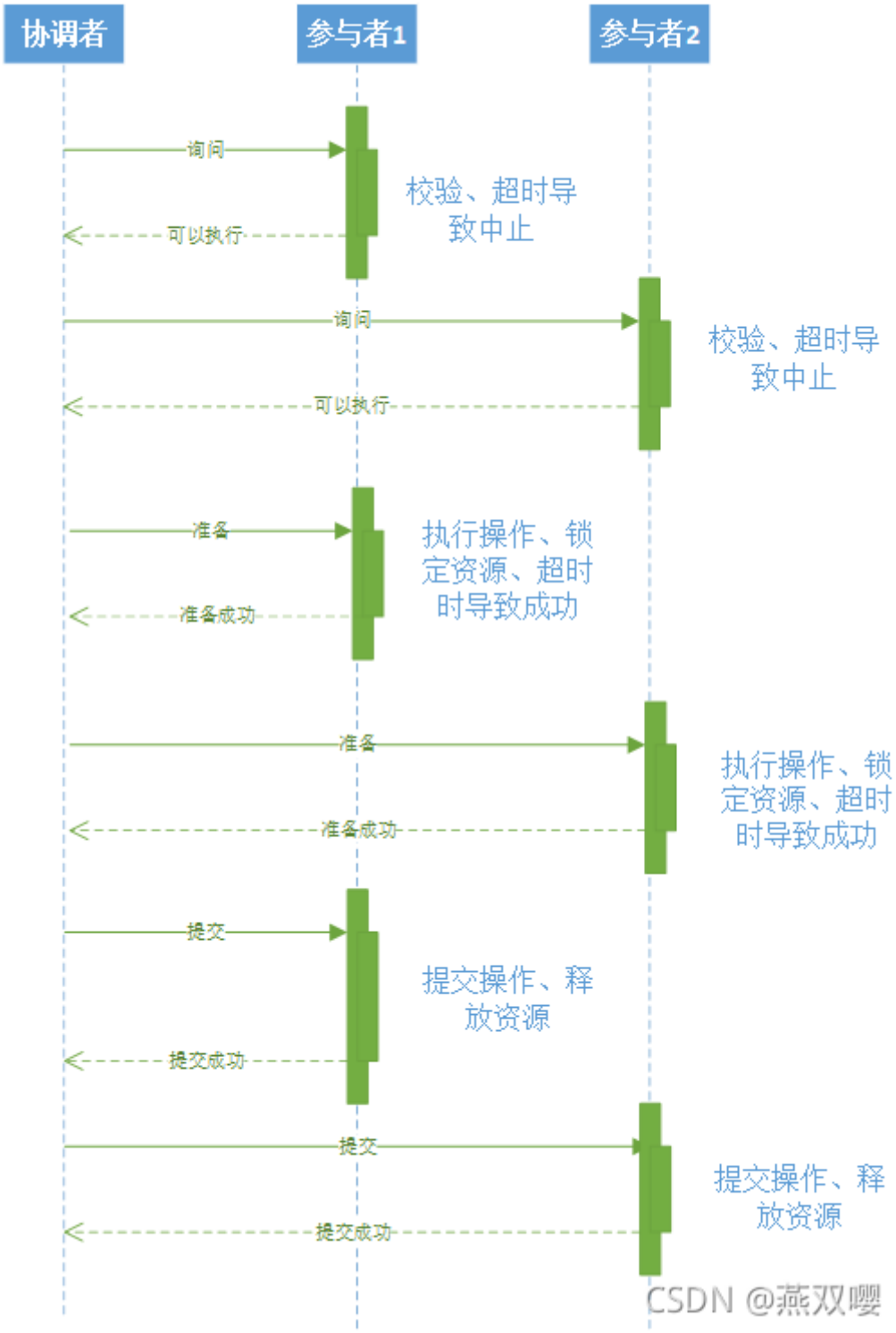
准备阶段：协调者向参与者发起指令，参与者评估自己的状态，如果参与者评估指令可以完成，则会写redo或undo日志，然后锁定资源，执行操作，但是并不提交。
提交阶段：如果每个参与者预留资源和执行操作成功，则协调者向参与者发起提交指令，参与者提交资源变更的事务，释放锁定的资源；如果任何一个参与者明确返回准备失败，也就是预留资源或者执行操作失败，则协调者向参与者发起中止指令，参与者取消已经变更的事务，执行undo日志，释放锁定的资源。

总的来说就是参与者按照协调者的命令根据自身条件进行是否执行提交协议。

三段式提交协议

对于两段式而言增加了一个询问阶段 。

询问阶段：协调者询问参与者是否可以完成指令，参与者只需要回答是或不是，而不需要做真正的操作，这个阶段超时会导致中止。



针对于两段式而言，增加了一个询问阶段，可以尽早发现无法执行操作而终止，但是仅仅是减少情况发生。同时如果协调者和参与者都超时了，那么就会认为还是成功。

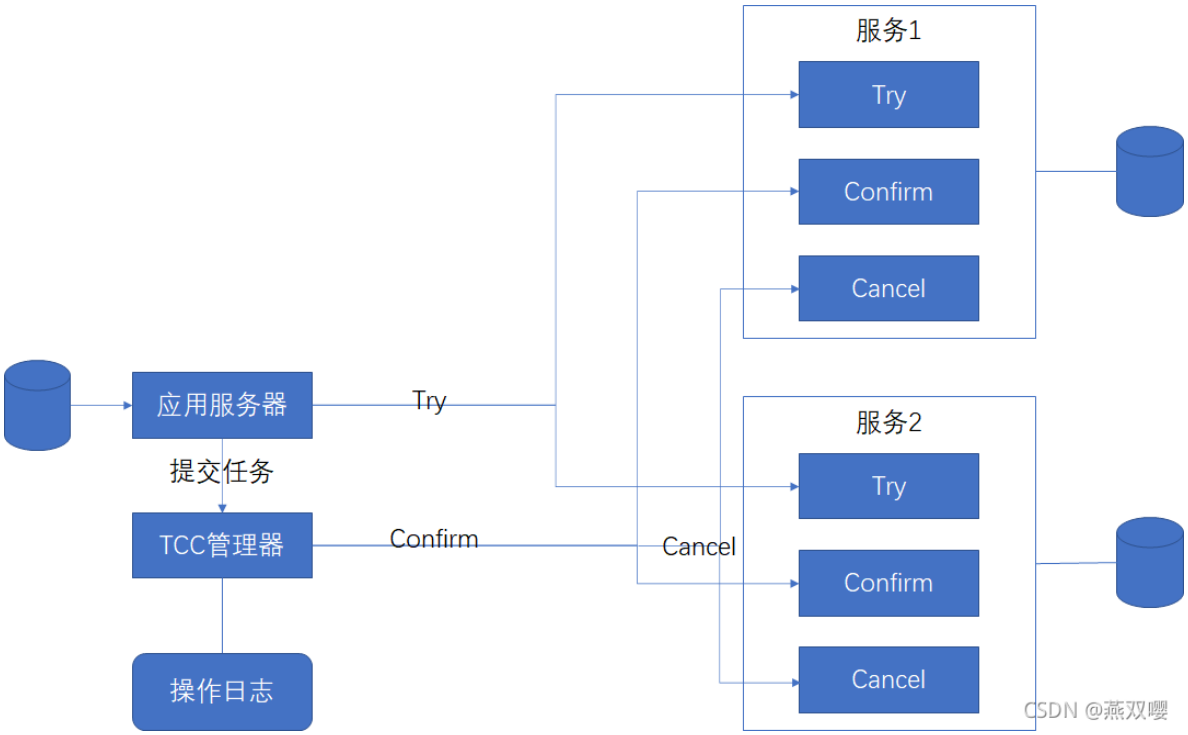
TCC

两段提交协议和三段提交协议能解决基本的分布式问题，但是遇到极端情况时，系统会产生阻塞或者不一致的问题，需要运营或者技术人员解决。两阶段及三阶段方案中都包含多个参与者、多个阶段实现一个事务，实现复杂，性能也是一个很大的问题。因此，在互联网的高并发系统中，鲜有使用两个阶段提交和三阶段提交协议的场景。

后来提出了TCC协议，TCC协议将一个任务分成：Try、Confirm、Cancel三个步骤。正常的流程会先执行Try，如果执行没问题，则在执行Confirm，如果执行过程中出了问题，则执行操作的逆过程操作Cancel。从正常的流程上讲，这仍然是一个两阶段提交协议，但是在执行出现问题时有一定的自我修复能力，如果任何参与者出现了问题，则协调者通过执行操作的逆操作来Cancel之前的操作，达到最终的一致状态。

但是，从时序上来说，如果遇到极端情况，则TCC会有很多问题。例如，如果在取消时一些参与者收到指令，而另一些参与者没有收到指令，则整个系统仍然是不一致的。对于这种复杂的情况，系统首先会通过补偿的方式尝试自动修复，如果系统无法修复，则必须由人工参与解决。

从TCC逻辑上看，可以说TCC是简化版的三阶段提交协议，解决了两阶段提交协议的阻塞问题，但是没有解决极端情况下会出现不一致和脑裂的问题。然后TCC通过自动化补偿手段，将需要人工处理的不一致情况降到最少，也是一种非常有用的解决方案。



CAP

概念：CAP原则又称为帽子理论，指的是再一个分布式系统中，一致性、可用性、分区容错性，三者不可兼得。

一致性（Consistency）：数据一致更新，所有数据变动都是同步的。

可用性（Availability）：集群中部分节点故障后，集群整体还能对外提供服务。

分区容错性（Partition tolerance）：分区相当于对通信时限的要求；尽管网络有部分消息丢失，但系统任可持续工作。如果系统不能在一定时间内达成数据一致，就意味着发生了分区的情况，必须在当前操作C和A之间做出选择。

CAP原理证明任何分布式系统只可以同时满足2点，无法三者兼顾。

Zookeeper：CP模式

Spring Cloud 中的 Eureka：AP模式

最佳分布式一致性解决方案

满足我们的分布式需求、高并发高性能以及吞吐量为前提下，我认为分布式服务的一致性一定要以可靠性为基础、简洁性为目标去考量方案。我总结了几种比较高效的处理模式。

主动查询模式

所有的操作都提供一个查询接口，用于向外部输出操作执行的状态，服务操作的使用方可以通过查询接口而得知服务操作执行的状态。然后根据不同的状态来做不同的处理操作。

补偿模式

通常与主动查询模式结合使用，目的都是为了实现上下游的服务最终一致性的努力确保机制。

异步确保模式\可靠性消息模式

这两种模式也是互联网行业中经常需要使用的经典模式，很多时候我们的使用方对响应时间 要求不太高、或者说不需要特别强调实时性的场景，这一类的操作我们经常采用异步化、或者解耦的方式，把其从主流程（核心链路上摘除），或者说我们划分好业务边界，然后再可以容忍的窗口期内做异步确保和发送可靠性消息模式。这个方案最大的好处是能够对高并发流量进行削峰，从而对服务上能够提供解耦。比如我们电商系统中的物流、配送等，金融系统中的支付、计费入账等等

定期校对模式

这种方式多用于互联网金融行业，一般都是针对商户与平台与银行等第三方金融支付平台之间的一个经典场景，因为涉及资金安全，所以对于互联网金融行业会对其进行多重的一致性保证机制，比如商户交易对账、系统间一致性对账、财务对账等等。这种也是针对于场景而言的。一般对实时性要求最低，但是对准确性、一致性要求最高