分布式系统一致性（ACID、CAP、BASE、二段提交、三段提交、TCC、幂等性）原理详解

1 背景

一致性是一个抽象的、具有多重含义的计算机术语，在不同应用场景下，有不同的定义和含义。在传统的 IT 时代，一致性通常指强一致性，强一致性通常体现在你中有我、我中有你、浑然一体；而在互联网时代，一致性的含义远远超出了它原有的含义，在我们讨论互联网时代的一致性之前，我们先了解一下互联网时代的特点，互联网时代信息量巨大、需要计算能力巨大，不但对用户响应速度要求快，而且吞吐量指标也要向外扩展（既：水平伸缩），于是单节点的服务器无法满足需求，服务节点开始池化，想想那个经典的故事，一只筷子一折就断，一把筷子怎么都折不断，可见人多力量大的思想是多么的重要，但是人多也不一定能解决所有事情，还得进行有序、合理的分配任务，进行有效的管理，于是互联网时代谈论最多的话题就是拆分，拆分一般分为 “水平拆分” 和“垂直拆分”（大家不要对应到数据库或者缓存拆分，这里主要表达一种逻辑）。这里，“水平拆分”指的是同一个功能由于单机节点无法满足性能需求，需要扩展成为多节点，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的请求量，团结起来共同处理大规模高并发的请求量。“垂直拆分”指

的是按照功能拆分，秉着 “专业的人干专业的事儿” 的原则，把一个复杂的功能拆分到多个单一的简单的元功能，不同的元功能组合在一起，和未拆分前完成的功能是一致的，由于每个元功能职责单一、功能简单，让维护和变更都变得更简单、安全，更易于产品版本的迭代，在这样的一个互联网的时代和环境，一致性指分布式服务化系统之间的弱一致性，包括应用系统一致性和数据一致性。

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，凡事有好的一面，都会有坏的一面，拆分后的系统或者服务化的系统最大的问题就是一致性问题，这么多个具有元功能的模块，或者同一个功能池中的多个节点之间，如何保证他们的信息是一致的、工作步伐是一致的、状态是一致的、互相协调有序的工作呢?

2 问题

本节列举不一致会导致的种种问题，这也包括一例生活中的问题。

案例 1：买房

假如你想要享受生活的随意，只想买个两居，不想让房贷有太大压力，而你媳妇却想要买个三居，还得带花园的，那么你们就不一致了，不一致导致生活不愉快、不协调，严重情况下还会吵架，可见生活中的不一致问题影响很大。

案例 2：转账

转账是经典的不一致案例，设想一下银行为你处理一笔转账，扣减你账户上的余额，然后增加别人账户的余额；如果扣减你的账户余额成功，增加别人账户余额失败，那么你就会损失这笔资金。反过来，如果扣减你的账户余额失败，增加别人账户余额成功，那么银行就会损失这笔资金，银行需要赔付。对于资金处理系统来说，上面任何一种场景都是不允许发生的，一旦发生就会有资金损失，后果是不堪设想的，严重情况会让一个公司瞬间倒闭，可参考 案例 。

案例 3：下订单和扣库存

电商系统中也有一个经典的案例，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；如果下订单没有成功，扣库存成功，那么会导致少卖。两种情况都会导致运营成本的增加，严重情况下需要赔付。

案例 4：同步超时

服务化的系统间调用常常因为网络问题导致系统间调用超时，即使是网络很好的机房，在亿次流量的基数下，同步调用超时也是家常便饭。系统 A 同步调用系统 B 超时，系统 A 可以明确得到超时反馈，但是无法确定系统 B 是否已经完成了预定的功能或者没有完成预定的功能。于是，系统 A 就迷茫了，不知道应该继续做什么，如何反馈给使用方。（曾经的一个 B2B 产品的客户要求接口超时重新通知他们，这个在技术上是难以实现的，因为服务器本身可能并不知道自己超时，可能会继续正常的返回数据，只是客户端并没有接受到结果罢了，因此这不是一个合理的解决方案）。

案例 5：异步回调超时

此案例和上一个同步超时案例类似，不过这个场景使用了异步回调，系统 A 同步调用系统 B 发起指令，系统 B 采用受理模式，受理后则返回受理成功，然后系统 B 异步通知系统 A。在这个过程中，如果系统 A 由于某种原因迟迟没有收到回调结果，那么两个系统间的状态就不一致，互相认知不同会导致系统间发生错误，严重情况下会影响核心事务，甚至会导致资金损失。

案例 6：掉单

分布式系统中，两个系统协作处理一个流程，分别为对方的上下游，如果一个系统中存在一个请求，通常指订单，另外一个系统不存在，则导致掉单，掉单的后果很严重，有时候也会导致资金损失。

案例 7：系统间状态不一致

这个案例与上面掉单案例类似，不同的是两个系统间都存在请求，但是请求的状态不一致。

案例 8：缓存和数据库不一致

交易相关系统基本离不开关系型数据库，依赖关系型数据库提供的 ACID 特性（后面介绍），但是在大规模高并发的互联网系统里，一些特殊的场景对读的性能要求极高，服务于交易的数据库难以抗住大规模的读流量，通常需要在数据库前垫缓存，那么缓存和数据库之间的数据如何保持一致性？是要保持强一致呢还是弱一致性呢？

案例 9：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，使用了本地缓存，每个节点都会有一份缓存数据的拷贝，如果这些数据是静态的、不变的，那永远都不会有问题，但是如果这些数据是半静态的或者常被更新的，当被更新的时候，各个节点更新是有先后顺序的，在更新的瞬间，各个节点的数据是不一致的，如果这些数据是为某一个开关服务的，想象一下重复的请求走进了不同的节点（在 failover 或者补偿导致的场景下，重复请求是一定会发生的，也是服务化系统必须处理的），一个请求走了开关打开的逻辑，同时另外一个请求走了开关关闭的逻辑，这导致请求被处理两次，最坏的情况下会导致灾难性的后果，就是资金损失。

案例 10：缓存数据结构不一致

这个案例会时有发生，某系统需要种某一数据结构的缓存，这一数据结构有多个数据元素组成，其中，某个数据元素都需要从数据库中或者服务中获取，如果一部分数据元素获取失败，由于程序处理不正确，仍然将不完全的数据结构存入缓存，那么缓存的消费者消费的时候很有可能因为没有合理处理异常情况而出错。

3 模式

3.1 生活中不一致问题的解决

大家回顾一下上一节列举的生活中的*案例 1 - 买房*，如果置身事外来看，解决这种不一致的办法有两个，一个是避免不一致的发生，如果已经是媳妇了就不好办了:)，还有一种方法就是慢慢的补偿，先买个两居，然后慢慢的等资金充裕了再换三居，买比特币赚了再换带花园的房子，于是问题最终被解决了，最终大家处于一致的状态，都开心了。这样可以解决案例 1 的问题，很自然由于有了过渡的方法，问题在不经意间就消失了，可见 “过渡” 也是解决一致性问题的一个模式。

从案例 1 的解决方案来看，我们要解决一致性问题，一个最直接最简单的方法就是保持强一致性，对于案例 1 的情况，尽量避免在结婚前两个人能够互相了解达成一致，避免不一致问题的发生；不过有些事情事已至此，发生了就是发生了，出现了不一致的问题，我们应该考虑去补偿，尽最大的努力从不一致状态修复到一致状态，避免损失全部或者一部分，也不失为一个好方法。

因此，避免不一致是上策，出现了不一致及时发现及时修复是中策，有问题不积极解决留给他人解决是下策。

3.2 酸碱平衡理论

ACID 在英文中的意思是 “酸”，BASE 的意识是“碱”，这一段讲的是“酸碱平衡” 的故事。

1. ACID（酸）

如何保证强一致性呢？计算机专业的童鞋在学习关系型数据库的时候都学习了 ACID 原理，这里对 ACID 做个简单的介绍。如果想全面的学习 ACID 原理，请参考 ACID 。

关系型数据库天生就是解决具有复杂事务场景的问题，关系型数据库完全满足 ACID 的特性。

ACID 指的是：

- **A: Atomicity，原子性**
- **C: Consistency，一致性**
- **I: Isolation，隔离性**
- **D: Durability，持久性**

具有 ACID 的特性的数据库支持强一致性，强一致性代表数据库本身不会出现不一致，每个事务是原子的，或者成功或者失败，事物间是隔离的，互相完全不影响，而且最终状态是持久落盘的，因此，数据库会从一个明确的状态到另外一个明确的状态，中间的临时状态是不会出现的，如果出现也会及时的自动的修复，因此是强一致的。

3 个典型的关系型数据库 Oracle、Mysql、Db2 都能保证强一致性，Oracle 和 Mysql 使用多版本控制协议实现，而 DB2 使用改进的两阶段提交协议来实现。

如果你在为交易相关系统做技术选型，交易的存储应该只考虑关系型数据库，对于核心系统，如果需要较好的性能，可以考虑使用更强悍的硬件，这种向上扩展（升级硬件）虽然成本较高，但是是最简单粗暴有效的方式，另外，Nosql 完全不适合交易场景，Nosql 主要用来做数据分析、ETL、报表、数据挖掘、推荐、日志处理等非交易场景。

前面提到的*案例 2 - 转账*和*案例 3 - 下订单和扣库存*都可以利用关系型数据库的强一致性解决。

然而，前面提到，互联网项目多数具有大规模高并发的特性，必须应用拆分的理念，对高并发的压力采取 “大而化小、小而化了” 的方法，否则难以满足动辄亿级流量的需求，即使使用关系型数据库，单机也难以满足存储和 TPS 上的需求。为了保证*案例 2 - 转账*可以利用关系型数据库的强一致性，在拆分的时候尽量的把转账相关的账户放入一个数据库分片，对于案例 3，尽量的保证把订单和库存放入同一个数据库分片，这样通过关系型数据库自然就解决了不一致的问题。

然而，有些时候事与愿违，由于业务规则的限制，无法将相关的数据分到同一个数据库分片，这个时候我们就需要实现最终一致性。

对于*案例 2 - 转账*场景，假设账户数量巨大，对账户存储进行了拆分，关系型数据库一共分了 8 个实例，每个实例 8 个库，每个库 8 个表，共 512 张表，假如要转账的两个账户正好落在了一个库里，那么可以依赖关系型数据库的事务保持强一致性。

如果要转账的两个账户正好落在了不同的库里，转账操作是无法封装在同一个数据库事务中的，这个时候会发生一个库的账户扣减余额成功，另外一个库的账户增加余额失败的情况。

对于这种情况，我们需要继续探讨解决之道，CAP 原理和 BASE 原理，BASE 原理通过记录事务的中间的临时状态，实现最终一致性。

2. CAP（帽子理论）

如果想深入的学习 CAP 理论，请参考 CAP 。

由于对系统或者数据进行了拆分，我们的系统不再是单机系统，而是分布式系统，针对分布式系的帽子理论包含三个元素：

- **C：Consistency，一致性, 数据一致更新，所有数据变动都是同步的**
- **A：Availability，可用性, 好的响应性能，完全的可用性指的是在任何故障模型下，服务都会在有限的时间处理响应**
- **P：Partition tolerance，分区容错性，可靠性**

帽子理论证明，任何分布式系统只可同时满足二点，没法三者兼顾。关系型数据库由于关系型数据库是单节点的，因此，不具有分区容错性，但是具有一致性和可用性，而分布式的服务化系统都需要满足分区容错性，那么我们必须在一致性和可用性中进行权衡，具体表现在服务化系统处理的异常请求在某一个时间段内可能是不完全的，但是经过自动的或者手工的补偿后，达到了最终的一致性。

3. BASE（碱）

BASE 理论解决 CAP 理论提出了分布式系统的一致性和可用性不能兼得的问题，如果想全面的学习 BASE 原理，请参考 Eventual consistency 。

BASE 在英文中有 “碱” 的意思，对应本节开头的 ACID 在英文中 “酸” 的意思，基于这两个名词提出了酸碱平衡的结论，简单来说是在不同的场景下，可以分别利用 ACID 和 BASE 来解决分布式服务化系统的一致性问题。

BASE 模型与 ACID 模型截然不同，满足 CAP 理论，通过牺牲强一致性，获得可用性，一般应用在服务化系统的应用层或者大数据处理系统，通过达到最终一致性来尽量满足业务的绝大部分需求。

BASE 模型包含个三个元素：

- **BA：Basically Available，基本可用**
- **S：Soft State，软状态，状态可以有一段时间不同步**
- **E：Eventually Consistent，最终一致，最终数据是一致的就可以了，而不是时时保持强一致**

BASE 模型的软状态是实现 BASE 理论的方法，基本可用和最终一致是目标。按照 BASE 模型实现的系统，由于不保证强一致性，系统在处理请求的过程中，可以存在短暂的不一致，在短暂的不一致窗口请求处理处在临时状态中，系统在做每步操作的时候，通过记录每一个

临时状态，在系统出现故障的时候，可以从这些中间状态继续未完成的请求处理或者退回到原始状态，最后达到一致的状态。

以*案例 1 - 转账*为例，我们把用户 A 给用户 B 转账分成四个阶段，第一个阶段用户 A 准备转账，第二个阶段从用户 A 账户扣减余额，第三个阶段对用户 B 增加余额，第四个阶段完成转账。系统需要记录操作过程中每一步骤的状态，一旦系统出现故障，系统能够自动发现没有完成的任务，然后，根据任务所处的状态，继续执行任务，最终完成任务，达到一致的最终状态。

在实际应用中，上面这个过程通常是通过持久化执行任务的状态和环境信息，一旦出现问题，定时任务会捞取未执行完的任务，继续未执行完的任务，直到执行完成为止，或者取消已经完成的部分操作回到原始状态。这种方法在任务完成每个阶段的时候，都要更新数据库中任务的状态，这在大规模高并发系统中不会有太好的性能，一个更好的办法是用 Write-Ahead Log（写前日志），这和数据库的 Bin Log（操作日志）相似，在做每一个操作步骤，都先写入日志，如果操作遇到问题而停止的时候，可以读取日志按照步骤进行恢复，并且继续执行未完成的工作，最后达到一致。写前日志可以利用机械硬盘的追加写而达到较好性能，因此，这是一种专业化的实现方式，多数业务系系统还是使用数据库记录的字段来记录任务的执行状态，也就是记录中间的 “软状态”，一个任务的状态流转一般可以通过数据库的行级锁来实现，这比使用 Write-Ahead Log 实现更简单、更快速。

有了 BASE 理论作为基础，我们对复杂的分布式事务进行拆解，对其中的每一步骤都记录其状态，有问题的时候可以根据记录的状态来继续执行任务，达到最终的一致，通过这个方法我们可以解决*案例 2 - 转账*和*案例 3 - 下订单和扣库存*中遇到的问题。

4. 酸碱平衡的总结

- **使用向上扩展（强悍的硬件）运行专业的关系型数据库（例如：Oracle 或者 DB2）能够保证强一致性，钱能解决的问题就不是问题**
- **如果钱是问题，可以对廉价硬件运行的开源关系型数据库（例如：Mysql）进行分片，将相关的数据分到数据库的同一个片，仍然能够使用关系型数据库保证事务**
- **如果业务规则限制，无法将相关的数据分到同一个片，就需要实现最终一致性，通过记录事务的软状态（中间状态、临时状态），一旦处于不一致，可以通过系统自动化或者人工干预来修复不一致的情况**

3.3 分布式一致性协议

国际开放标准组织 Open Group 定义了 DTS（分布式事务处理模型），模型中包含 4 个角色：应用程序、事务管理器、资源管理器、通信资源管理器四部分。事务处理器是统管全局的管理者，资源处理器和通信资源处理器是事务的参与者。

J2EE 规范也包含此分布式事务处理模型的规范，并在所有的 AppServer 中进行实现，J2EE 规范中定义了 TX 协议和 XA 协议，TX 协议定义应用程序与事务管理器之间的接口，而 XA 协议定义了事务管理器与资源处理器之间的接口，在过去，大家使用 AppServer，例如：Websphere、Weblogic、Jboss 等配置数据源的时候会看见类似 XADatasource 的数据源，这就是实现了 DTS 的关系型数据库的数据源。企业级开发 JEE 中，关系型数据库、JMS 服务扮演资源管理器的角色，而 EJB 容器则扮演事务管理器的角色。

下面我们就介绍 两阶段提交协议 、 三阶段提交协议 以及阿里巴巴提出的 TCC，它们都是根据 DTS 这一思想演变出来的。

1. 两阶段提交协议

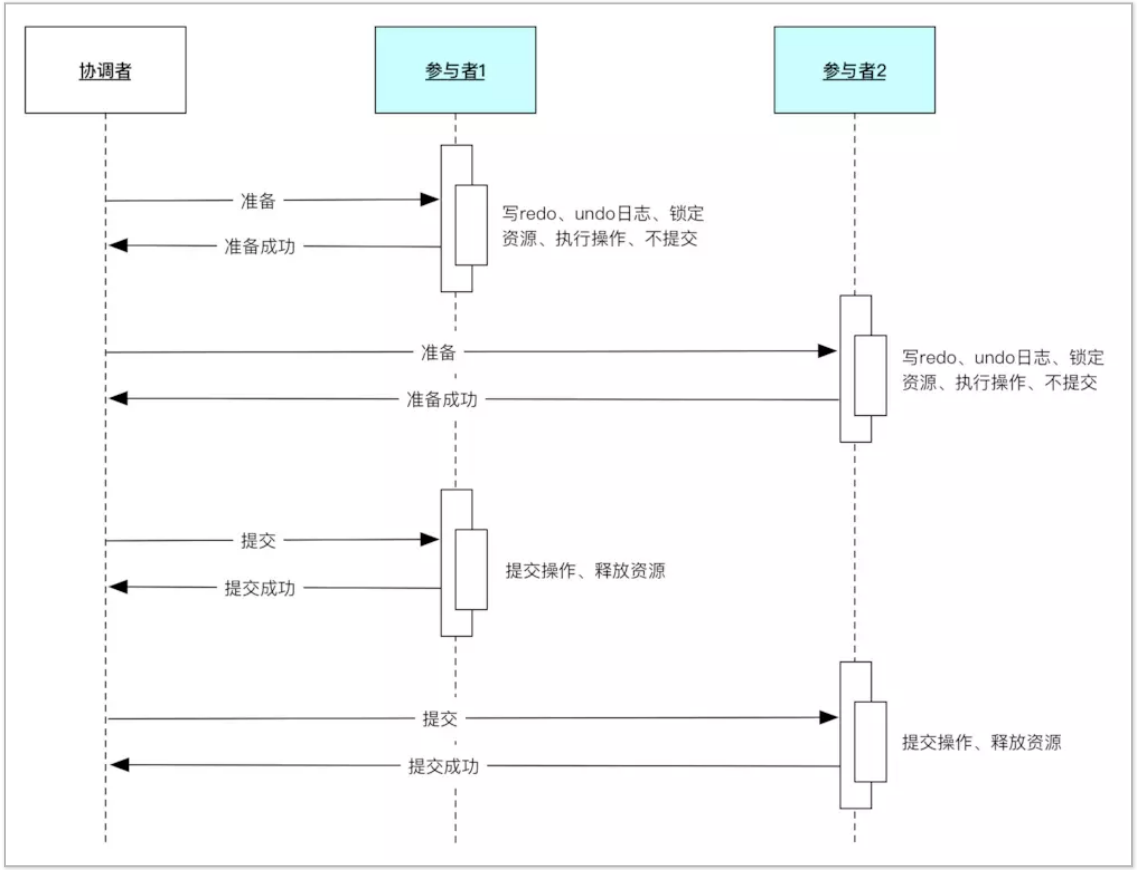
上面描述的 JEE 的 XA 协议就是根据两阶段提交来保证事务的完整性，并实现分布式服务化的强一致性。

两阶段提交协议把分布式事务分成两个过程，一个是准备阶段，一个是提交阶段，准备阶段和提交阶段都是由事务管理器发起的，为了接下来讲解方便，我们把事务管理器称为协调者，把资管管理器称为参与者。

两阶段如下：

- **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，参与者会写 redo 或者 undo 日志（这也是前面提起的 Write-Ahead Log 的一种），然后锁定资源，执行操作，但是并不提交**
- **提交阶段：如果每个参与者明确返回准备成功，也就是预留资源和执行操作成功，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，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**

两阶段提交协议成功场景示意图如下：



两阶段提交协议

我们看到两阶段提交协议在准备阶段锁定资源，是一个重量级的操作，并能保证强一致性，但是实现起来复杂、成本较高，不够灵活，更重要的是它有如下致命的问题：

- **阻塞：从上面的描述来看，对于任何一次指令必须收到明确的响应，才会继续做下一步，否则处于阻塞状态，占用的资源被一直锁定，不会被释放**

- **单点故障：如果协调者宕机，参与者没有了协调者指挥，会一直阻塞，尽管可以通过选举新的协调者替代原有协调者，但是如果之前协调者在发送一个提交指令后宕机，而提交指令仅仅被一个参与者接受，并且参与者接收后也宕机，新上任的协调者无法处理这种情况**

- **脑裂：协调者发送提交指令，有的参与者接收到执行了事务，有的参与者没有接收到事务，就没有执行事务，多个参与者之间是不一致的**

上面所有的这些问题，都是需要人工干预处理，没有自动化的解决方案，因此两阶段提交协议在正常情况下能保证系统的强一致性，但是在出现异常情况下，当前处理的操作处于错误状态，需要管理员人工干预解决，因此可用性不够好，这也符合 CAP 协议的一致性和可用性不能兼得的原理。

2. 三阶段提交协议

三阶段提交协议是两阶段提交协议的改进版本。它通过超时机制解决了阻塞的问题，并且把两个阶段增加为三个阶段：

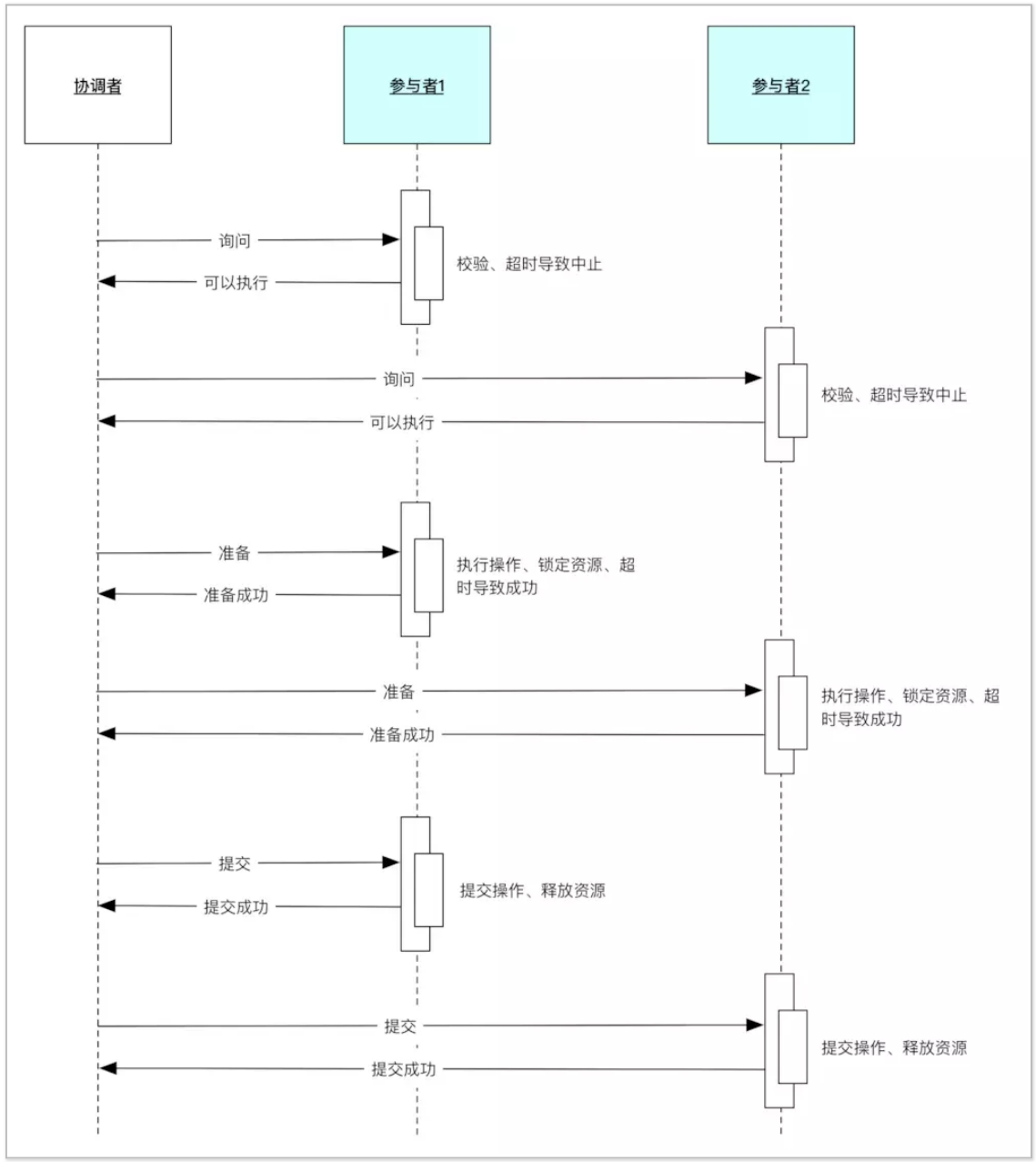
- **询问阶段：协调者询问参与者是否可以完成指令，协调者只需要回答是还是不是，而不需要做真正的操作，这个阶段超时导致中止**

- **准备阶段：如果在询问阶段所有的参与者都返回可以执行操作，协调者向参与者发送预执行请求，然后参与者写 redo 和 undo 日志，执行操作，但是不提交操作；如果**

在询问阶段任何参与者返回不能执行操作的结果，则协调者向参与者发送中止请求，这里的逻辑与两阶段提交协议的的准备阶段是相似的，这个阶段超时导致成功

- **提交阶段：如果每个参与者在准备阶段返回准备成功，也就是预留资源和执行操作成功，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返回准备失败，也就是预留资源或者执行操作失败，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，这里的逻辑与两阶段提交协议的提交阶段一致**

三阶段提交协议成功场景示意图如下：


协调者
参与者1
参与者2
询问
可以执行
校验、超时导致中止
询问
可以执行
校验、超时导致中止
准备
准备成功
执行操作、锁定资源、超时导致成功
准备
准备成功
执行操作、锁定资源、超时导致成功
提交
提交成功
提交操作、释放资源
提交
提交成功
提交操作、释放资源

三阶段提交协议

然而，这里与两阶段提交协议有两个主要的不同：

- **增加了一个询问阶段，询问阶段可以确保尽可能早的发现无法执行操作而需要中止的行为，但是它并不能发现所有的这种行为，只会减少这种情况的发生**
- **在准备阶段以后，协调者和参与者执行的任务中都增加了超时，一旦超时，协调者和参与者都继续提交事务，默认为成功，这也是根据概率统计上超时后默认成功的正确性最大**

三阶段提交协议与两阶段提交协议相比，具有如上的优点，但是一旦发生超时，系统仍然会发生不一致，只不过这种情况很少见罢了，好处就是至少不会阻塞和永远锁定资源。

3. TCC

上面两节讲解了两阶段提交协议和三阶段提交协议，实际上他们能解决*案例 2 - 转账*和*案例 3 - 下订单和扣库存*中的分布式事务的问题，但是遇到极端情况，系统会发生阻塞或者不一致的问题，需要运营或者技术人工解决。无论两阶段还是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，在互联网高并发系统中，鲜有使用两阶段提交和三阶段提交协议的场景。

阿里巴巴提出了新的 TCC 协议，TCC 协议将一个任务拆分成 Try、Confirm、Cancel，正常的流程会先执行 Try，如果执行没有问题，再执行 Confirm，如果执行过程中出了问题，则执行操作的逆操 Cancel，从正常的流程上讲，这仍然是一个两阶段的提交协议，但是，在

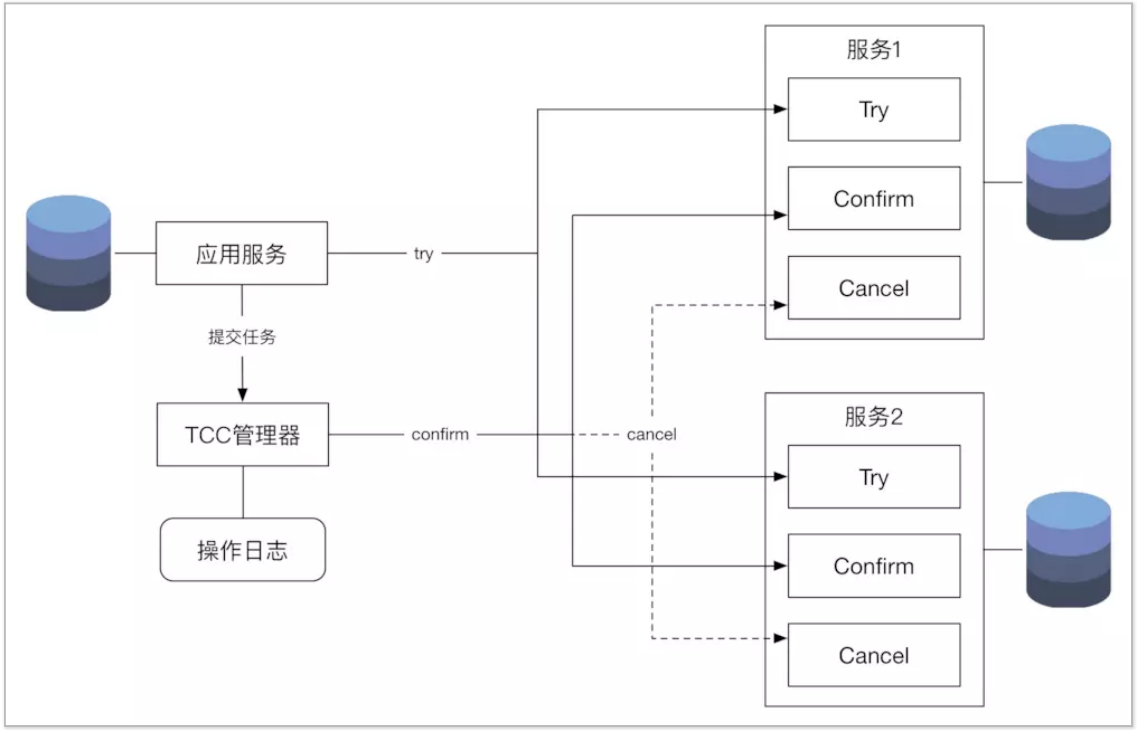
执行出现问题的时候，有一定的自我修复能力，如果任何一个参与者出现了问题，协调者通过执行操作的逆操作来取消之前的操作，达到最终的一致状态。

可以看出，从时序上，如果遇到极端情况下 TCC 会有很多问题的，例如，如果在 Cancel 的时候一些参与者收到指令，而一些参与者没有收到指令，整个系统仍然是不一致的，这种复杂的情况，系统首先会通过补偿的方式，尝试自动修复的，如果系统无法修复，必须由人工参与解决。

从 TCC 的逻辑上看，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。然而，TCC 通过自动化补偿手段，会把需要人工处理的不一致情况降到到最少，也是一种非常有用的解决方案，根据线人，阿里在内部的一些中间件上实现了 TCC 模式。

我们给出一个使用 TCC 的实际案例，在秒杀的场景，用户发起下单请求，应用层先查询库存，确认商品库存还有余量，则锁定库存，此时订单状态为待支付，然后指引用户去支付，由于某种原因用户支付失败，或者支付超时，系统会自动将锁定的库存解锁供其他用户秒杀。

TCC 协议使用场景示意图如下：



TCC

总结一下，两阶段提交协议、三阶段提交协议、TCC 协议都能保证分布式事务的一致性，他们保证的分布式系统的一致性从强到弱，TCC 达到的目标是最终一致性，其中任何一种方法都可以不同程度的解决*案例 2：转账*、*案例 3：下订单和扣库存*的问题，只是实现的一致性的级别不一样而已，对于*案例 4：同步超时*可以通过 TCC 的理念解决，如果同步调用超时，调用方可以使用 fastfail 策略，返回调用方的使用方失败的结果，同时调用服务的逆向 cancel 操作，保证服务的最终一致性。

3.4 保证最终一致性的模式

在大规模高并发服务化系统中，一个功能被拆分成多个具有单一功能的元功能，一个流程会有多个系统的多个元功能组合实现，如果使用两阶段提交协议和三阶段提交协议，确实能解决系统间一致性问题，除了这两个协议带来的自身的问题，这些协议的实现比较复杂、成本比较高，最重要的是性能并不好，相比来看，TCC 协议更简单、容易实现，但是 TCC 协议由于每个事务都需要执行 Try，再执行 Confirm，略微显得臃肿，因此，在现实的系统中，底线要求仅仅需要能达到最终一致性，而不需要实现专业的、复杂的一致性协议，实现最终一致性有一些非常有效的、简单粗暴的模式，下面就介绍这些模式及其应用场景。

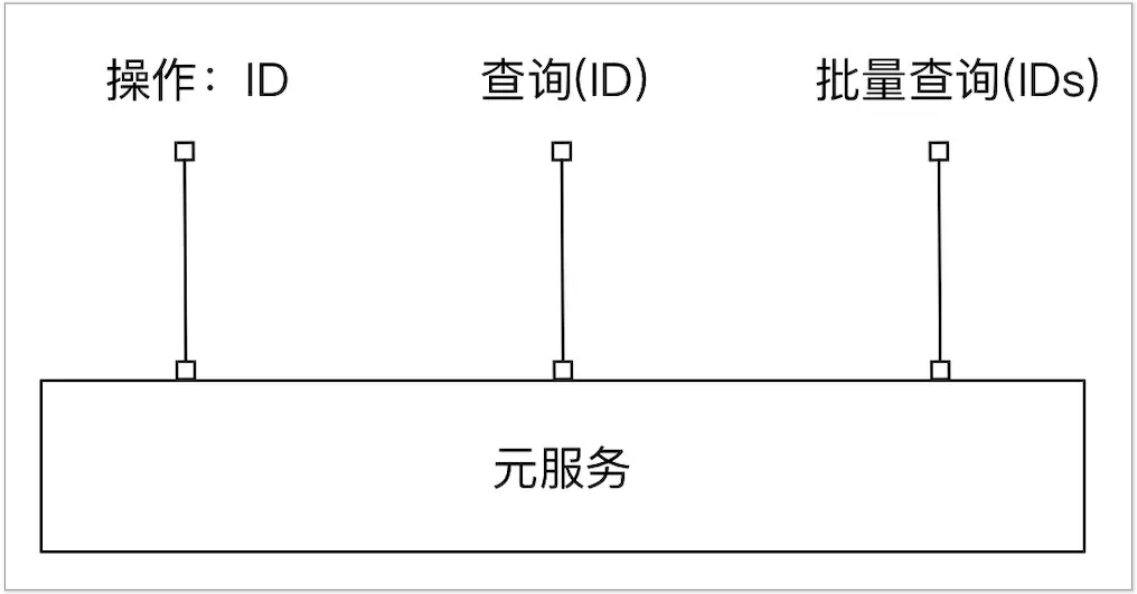
1. 查询模式

任何一个服务操作都需要提供一个查询接口，用来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，得知服务操作执行的状态，然后根据不同状态来做不同的处理操作。

为了能够实现查询，每个服务操作都需要有唯一的流水号标识，也可使用此次服务操作对应的资源 ID 来标志，例如：请求流水号、订单号等。

首先，单笔查询操作是必须提供的，我们也鼓励使用单笔订单查询，这是因为每次调用需要占用的负载是可控的，批量查询则根据需要来提供，如果使用了批量查询，需要有合理的分页机制，并且必须限制分页的大小，以及对批量查询的 QPS 需要有容量评估和流控等。

查询模式的示意图如下：



查询模式

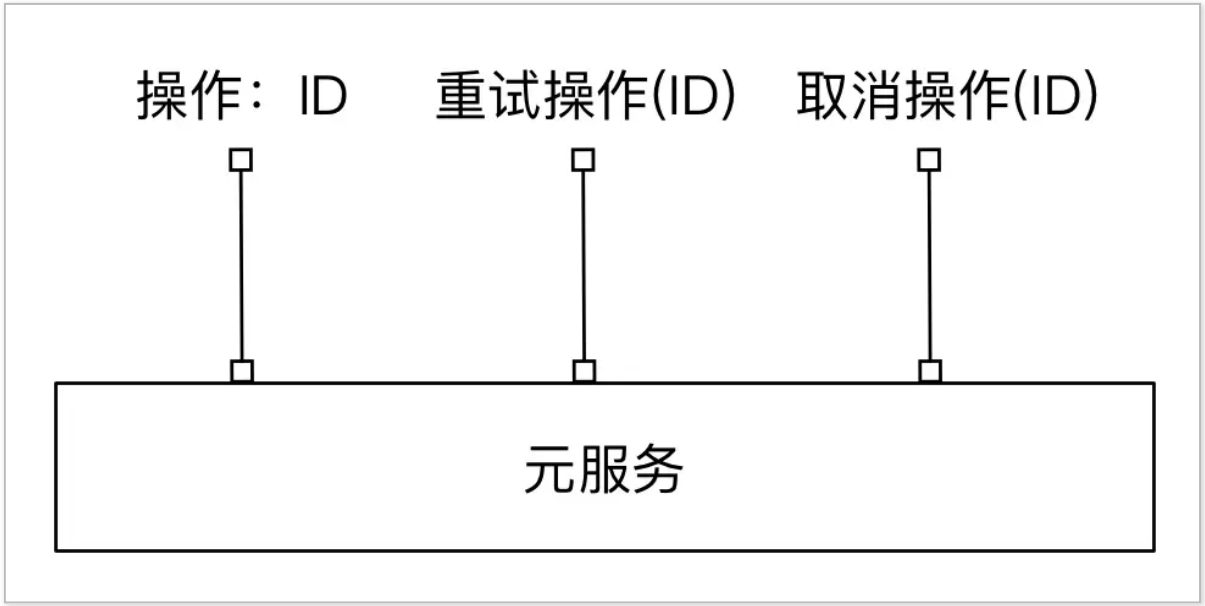
对于*案例 4：同步超时、案例 5：异步回调超时、案例 6：掉单、案例 7：系统间状态不一致*，我们都需要使用查询模式来了解被调用服务的处理情况，来决定下一步做什么：补偿未完成的操作还是回滚已经完成的操作。

2. 补偿模式

有了上面的查询模式，在任何情况下，我们都能得知具体的操作所处的状态，如果整个操作处于不正常的状态，我们需要修正操作中有问题的子操作，这可能需要重新执行未完成的子操作，后者取消已经完成的子操作，通过修复使整个分布式系统达到一致，为了让系统最终一致而做的努力都叫做补偿。

对于服务化系统中同步调用的操作，业务操作发起的主动方在还没有得到业务操作执行方的明确返回或者调用超时，场景可参考*案例 4：同步超时*，这个时候业务发起的主动方需要及时的调用业务执行方获得操作执行的状态，这里使用查询模式，获得业务操作的执行方的状态后，如果业务执行方已经完预设的工作，则业务发起方给业务的使用方返回成功，如果业务操作的执行方的状态为失败或者未知，则会立即告诉业务的使用方失败，然后调用业务操作的逆向操作，保证操作不被执行或者回滚已经执行的操作，让业务的使用方、业务发起的主动方、业务的操作方最终达成一致的状态。

补偿模式的示意图如下：



补偿模式

补偿操作根据发起形式分为：

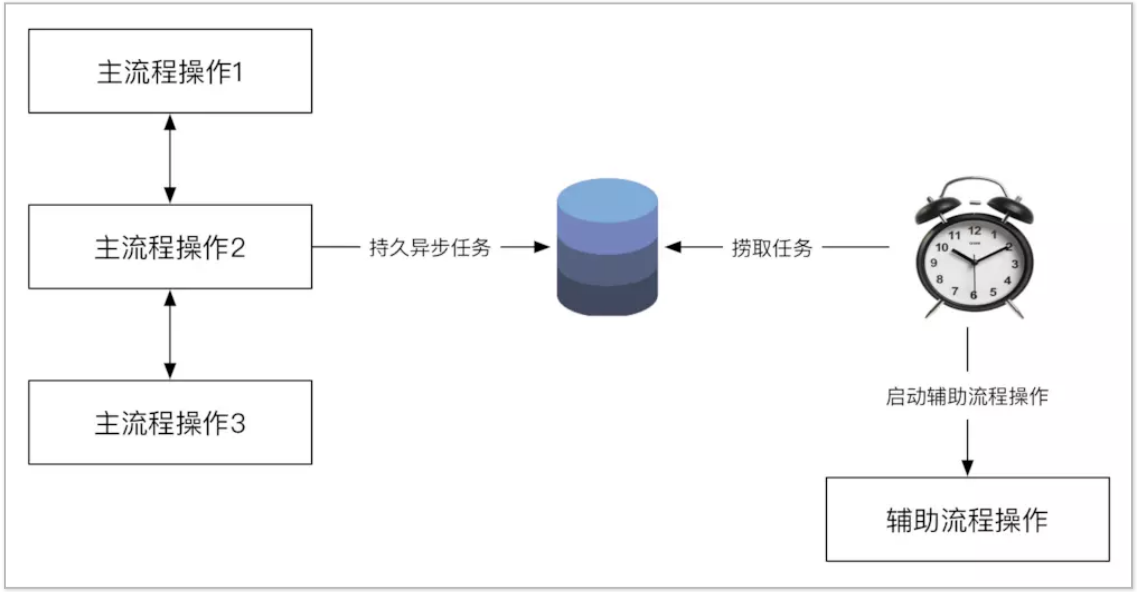
- **自动恢复：程序根据发生不一致的环境，通过继续未完成的操作，或者回滚已经完成的操作，自动来达到一致**
- **通知运营：如果程序无法自动恢复，并且设计时考虑到了不一致的场景，可以提供运营功能，通过运营手工进行补偿**
- **通知技术：如果很不巧，系统无法自动回复，又没有运营功能，那必须通过技术手段来解决，技术手段包括走数据库变更或者代码变更来解决，这是最糟的一种场景**

3. 异步确保模式

异步确保模式是补偿模式的一个典型案例，经常应用到使用方对响应时间要求并不太高，我们通常把这类操作从主流程中摘除，通过异步的方式进行处理，处理后把结果通过通知系统通知给使用方，这个方案最大的好处能够对高并发流量进行消峰，例如：电商系统中的物流、配送，以及支付系统中的计费、入账等。

实践中，将要执行的异步操作封装后持久入库，然后通过定时捞取未完成的任务进行补偿操作来实现异步确保模式，只要定时系统足够健壮，任何一个任务最终会被成功执行。

异步确保模式的示意图如下：



异步确保模式

对于*案例 5：异步回调超时*，使用的就是异步确保模式，这种情况下对于某个操作，如果迟迟没有收到响应，我们通过查询模式和补偿模式来继续未完成的操作。

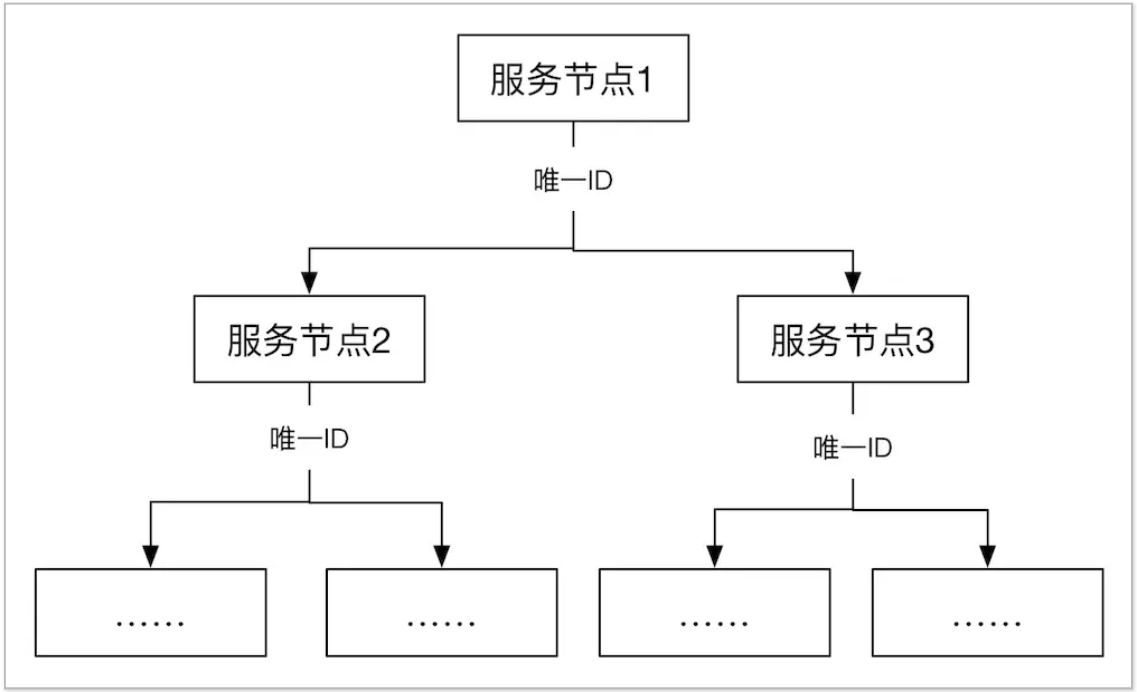
4. 定期校对模式

既然我们在系统中实现最终一致性，系统在没有达到一致之前，系统间的状态是不一致的，甚至是混乱的，需要补偿操作来达到一致的目的，但是我们如何来发现需要补偿的操作呢?

在操作的主流程中的系统间执行校对操作，我们可以事后异步的批量校对操作的状态，如果发现不一致的操作，则进行补偿，补偿操作与补偿模式中的补偿操作是一致的。

另外，实现定期校对的一个关键就是分布式系统中需要有一个自始至终唯一的 ID，ID 的生成请参考 SnowFlake 。

在分布式系统中，全局唯一 ID 的示意图如下：



唯一 ID

一般情况下，生成全局唯一 ID 有两种方法：

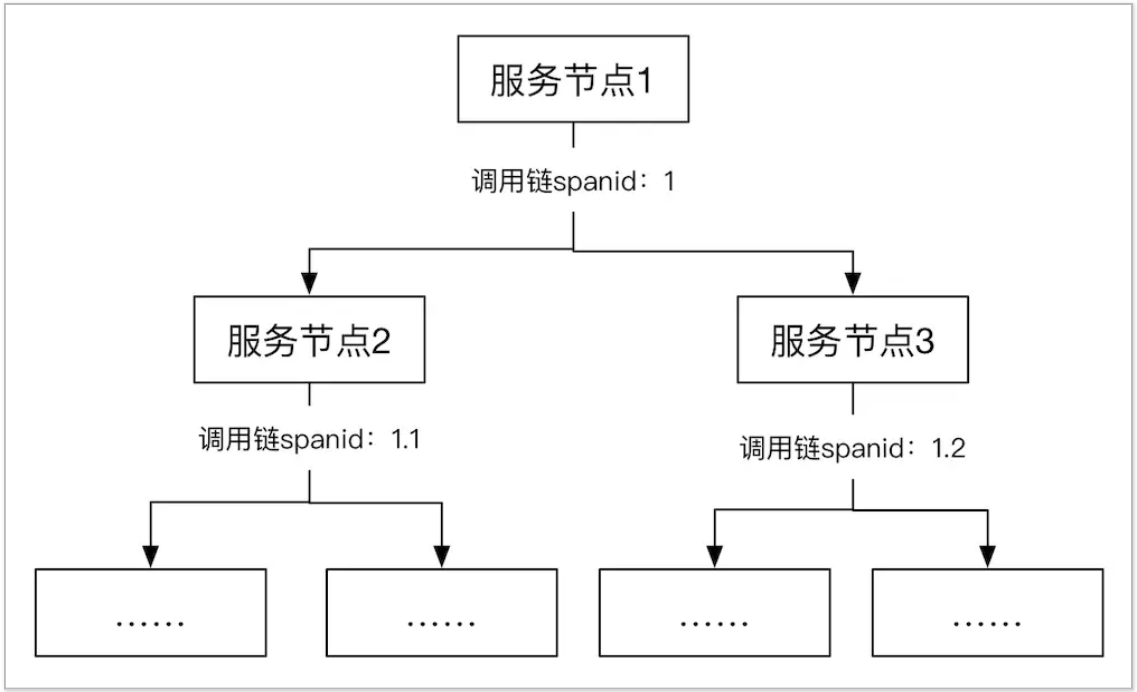
- **持久型：使用数据库表自增字段或者 Sequence 生成，为了提高效率，每个应用节点可以缓存一批次的 ID，如果机器重启可能会损失一部分 ID，但是这并不会产生任何**

问题

- **时间型：一般由机器号、业务号、时间、单节点内自增 ID 组成，由于时间一般精确到秒或者毫秒，因此不需要持久就能保证在分布式系统中全局唯一、粗略递增能特点**

实践中，为了能在分布式系统中迅速的定位问题，一般的分布式系统都有技术支持系统，它能够跟踪一个请求的调用链，调用链是在二维的维度跟踪一个调用请求，最后形成一个调用树，原理可参考谷歌的论文 Dapper, a Large-Scale Distributed Systems Tracing Infrastructure ，一个开源的参考实现为 pinpoint 。

在分布式系统中，调用链的示意图如下：



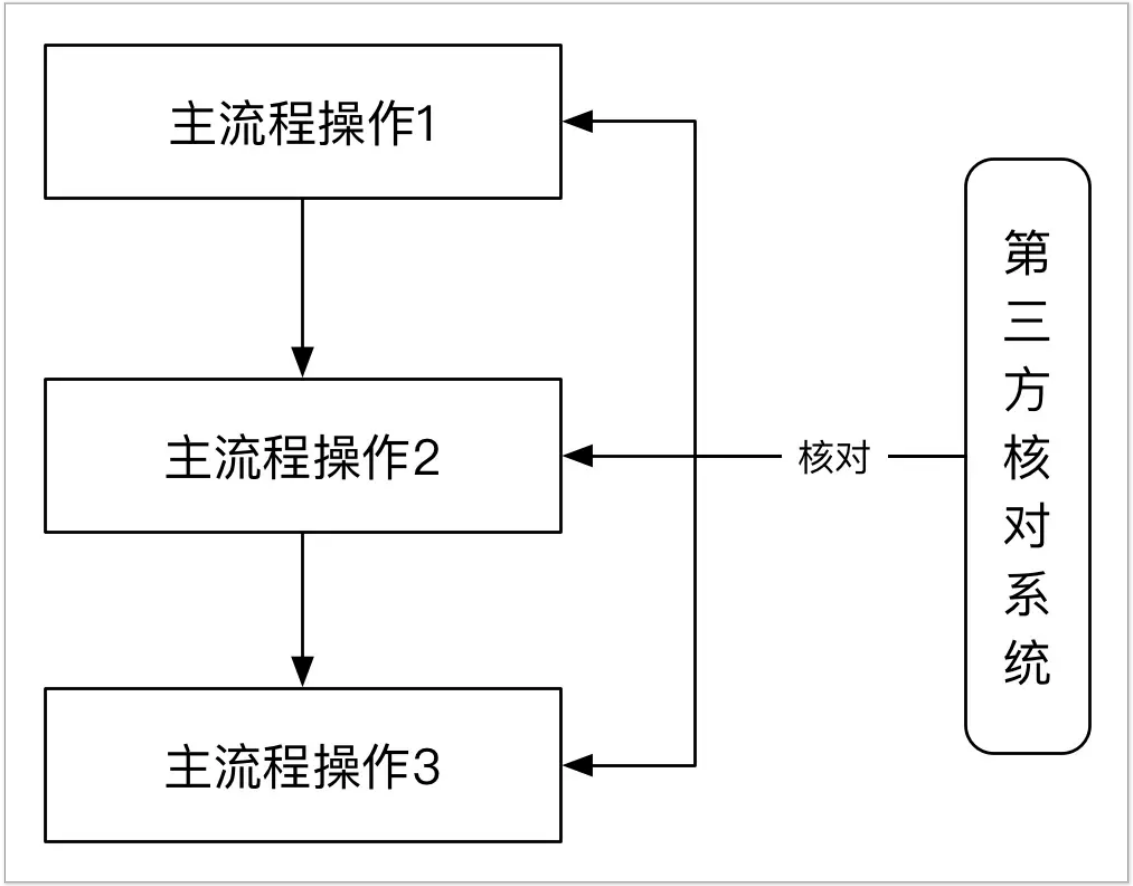
调用链

全局的唯一流水 ID 可以把一个请求在分布式系统中的流转的路径聚合，而调用链中的 spanid 可以把聚合的请求路径通过树形结构进行展示，让技术支持人员轻松的发现系统出现的问题，能够快速定位出现问题的服务节点，提高应急效率。

关于订单跟踪、调用链跟踪、业务链跟踪，我们会在后续文章中详细介绍。

在分布式系统中构建了唯一 ID，调用链等基础设施，我们很容易对系统间的不一致进行核对，通常我们需要构建第三方的定期核对系统，以第三方的角度来监控服务执行的健康程度。

定期核对系统示意图如下：



定期核对模式

对于*案例 6：掉单、案例 7：系统间状态不一致*通常通过定期校对模式发现问题，并通过补偿模式来修复，最后完成系统间的最终一致性。

定期校对模式多应用在金融系统，金融系统由于涉及到资金安全，需要保证百分之百的准确性，所以，需要多重的一致性保证机制，包括：系统间的一致性对账、现金对账、账务对

账、手续费对账等等，这些都属于定期校对模式，顺便说一下，金融系统与社交应用在技术上本质的区别在于社交应用在于量大，而金融系统在于数据的准确性。

到现在为止，我们看到通过查询模式、补偿模式、定期核对模式可以解决案例 4 到案例 7 的所有问题，对于*案例 4：同步超时*，如果同步超时，我们需要查询状态进行补偿，对于*案例 5：异步回调超时*，如果迟迟没有收到回调响应，我们也会通过查询状态进行补偿，对于*案例 6：掉单*、*案例 7：系统间状态不一致*，我们通过定期核对模式可以保证系统间操作的一致性，避免掉单和状态不一致导致问题。

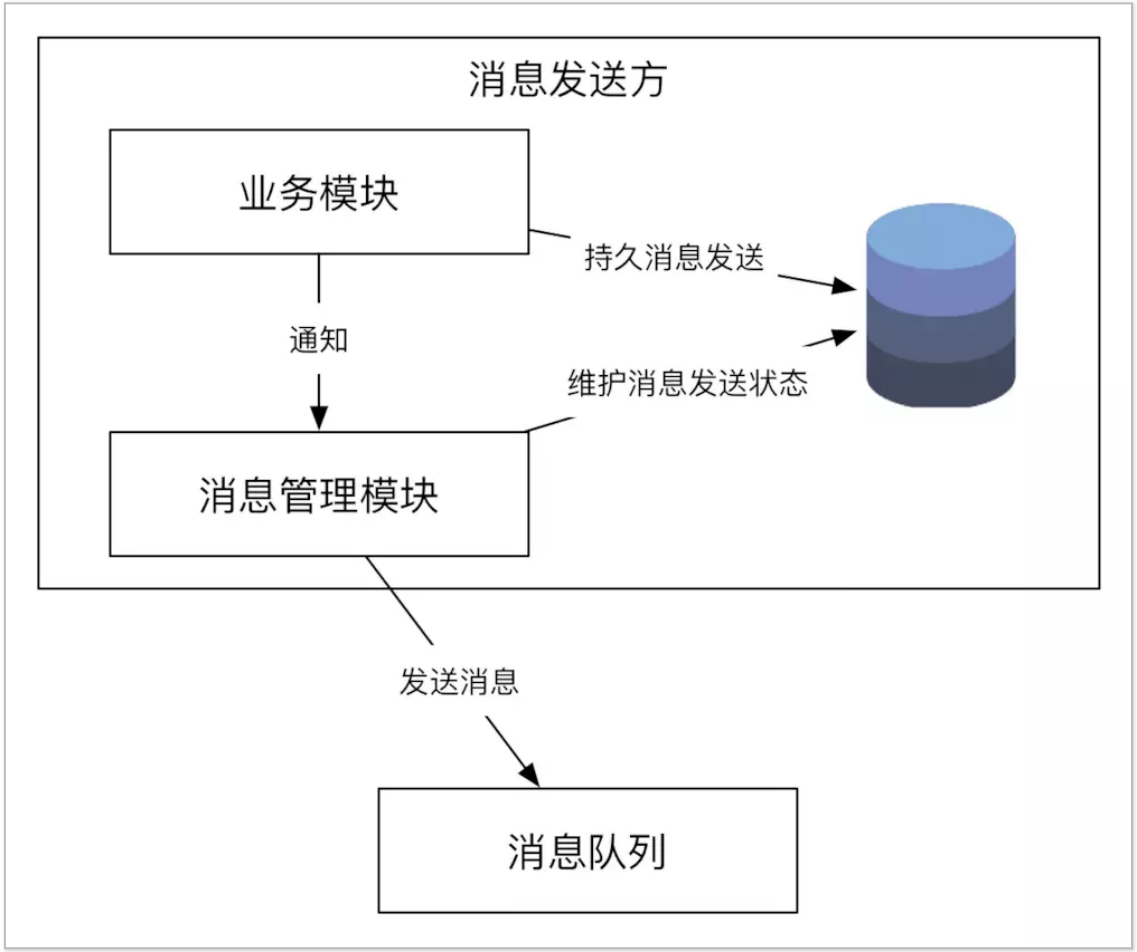
5. 可靠消息模式

在分布式系统中，对于主流程中优先级比较低的操作，大多采用异步的方式执行，也就是前面提到的异步确保型，为了让异步操作的调用方和被调用方充分的解耦，也由于专业的消息队列本身具有可伸缩、可分片、可持久等功能，我们通常通过消息队列实现异步化，对于消息队列，我们需要建立特殊的设施保证可靠的消息发送以及处理机的幂等等。

消息的可靠发送

消息的可靠发送可以认为是尽最大努力发送消息通知，有两种实现方法：

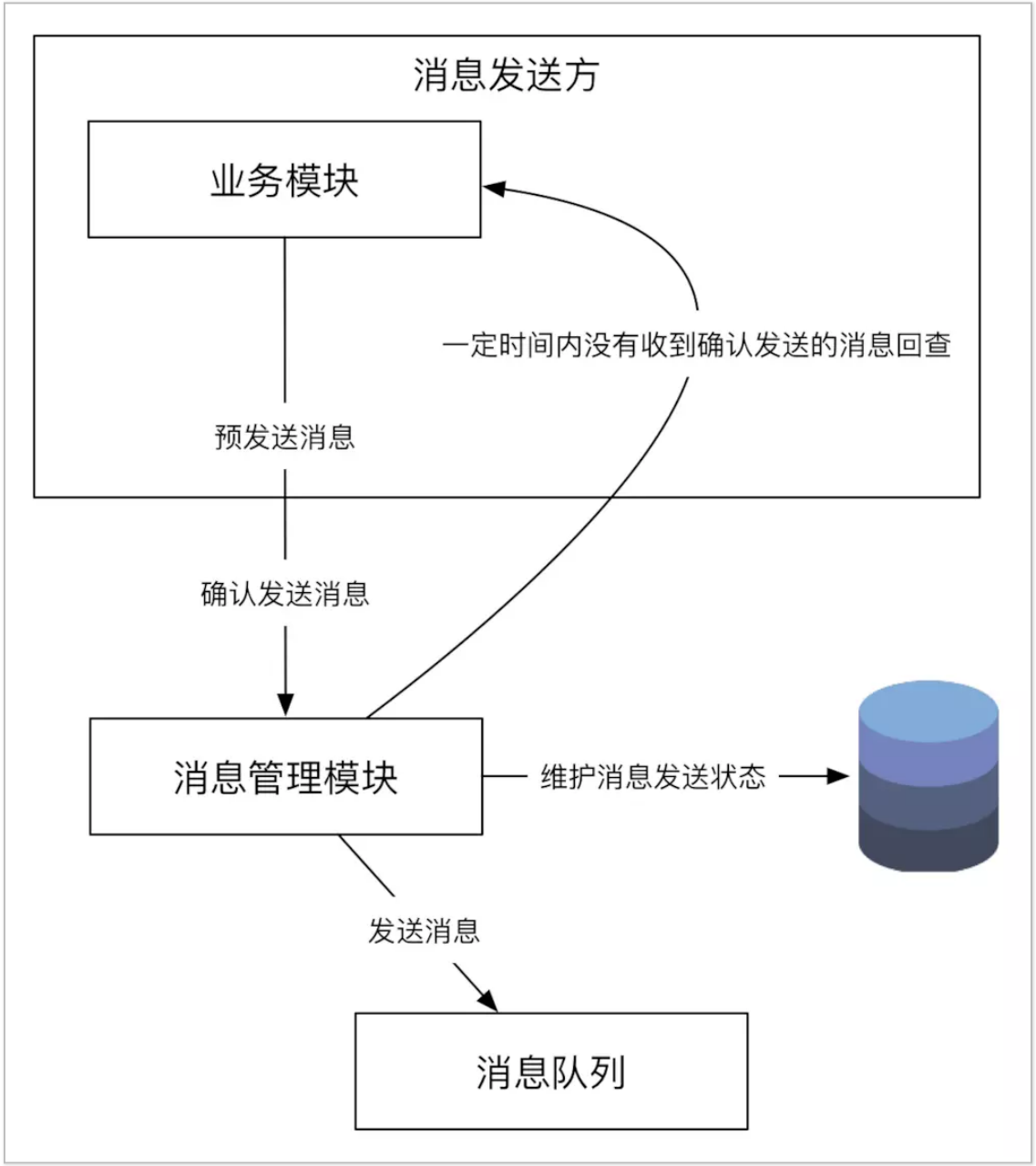
第一种，发送消息之前，把消息持久到数据库，状态标记为待发送，然后发送消息，如果发送成功，将消息改为发送成功。定时任务定时从数据库捞取一定时间内未发送的消息，将消息发送。



消息发送模式 1

第二种，实现方式与第一种类似，不同的是持久消息的数据库是独立的，并不耦合在业务系统中。发送消息之前，先发送一个预消息给某一个第三方的消息管理器，消息管理器将其持久到数据库，并标记状态为待发送，发送成功后，标记消息为发送成功。定时任务定时从数

据库捞取一定时间内未发送的消息，回查业务系统是否要继续发送，根据查询结果来确定消息的状态。


消息发送方
业务模块
一定时间内没有收到确认发送的消息回查
预发送消息
确认发送消息
消息管理模块
维护消息发送状态
发送消息
消息队列

消息发送模式 2

一些公司把消息的可靠发送实现在了中间件里，通过 Spring 的注入，在消息发送的时候自动持久消息记录，如果有消息记录没有发送成功，定时会补偿发送。

消息处理器的幂等性

如果我们要保证消息可靠的发送，简单来说，要保证消息一定要发送出去，那么就需要有重试机制，有了重试机制，消息一定会重复，那么我们需要对重复做处理。

处理重复的最佳方式为保证操作的幂等性，幂等性的数学公式为：

> $f(f(x)) = f(x)$

保证操作的幂等性常用的几个方法：

- **使用数据库表的唯一键进行滤重，拒绝重复的请求**
- **使用分布式表对请求进行滤重**
- **使用状态流转的方向性来滤重，通常使用行级锁来实现 (后续在锁相关的文章中详细说明)**
- **根据业务的特点，操作本身就是幂等的，例如：删除一个资源、增加一个资源、获得一个资源等**

6. 缓存一致性模型

大规模高并发系统中一个常见的核心需求就是亿级的读需求，显然，关系型数据库并不是解决高并发读需求的最佳方案，互联网的经典做法就是使用缓存抗读需求，下面有一些使用缓存的保证一致性的最佳实践：

- **如果性能要求不是非常的高，尽量使用分布式缓存，而不要使用本地缓存**
- **种缓存的时候一定种完全，如果缓存数据的一部分有效，一部分无效，宁可放弃种缓存，也不要把部分数据种入缓存**
- **数据库与缓存只需要保持弱一致性，而不需要强一致性，读的顺序要先缓存，后数据库，写的顺序要先数据库，后缓存**

这里的最佳实践能够解决*案例 8：缓存和数据库不一致*、*案例 9：本地缓存节点间不一致*、*案例 10：缓存数据结构不一致*的问题，对于数据存储层、缓存与数据库、Nosql 等的一致性是更深入的存储一致性技术，将会在后续文章单独介绍，这里的数据一致性主要是处理应用层与缓存、应用层与数据库、一部分的缓存与数据库的一致性。

3.5 专题模式

这一节介绍特殊场景下的一致性问题和解决方案。

迁移开关的设计

在大多数企业里，新项目和老项目一般会共存，大家都在努力的下掉老项目，但是由于种种原因总是下不掉，如果要彻底的下掉老项目，就必须要有非常完善的迁移方案，迁移是一项非常复杂而艰巨的任务，我会在将来的文章中详细探讨迁移方案、流程和技术，这里我们只对迁移中使用的开关进行描述。

迁移过程必须使用开关，开关一般都会基于多个维度来设计，例如：全局的、用户的、角色的、商户的、产品的等等，如果迁移过程中遇到问题，我们需要关闭开关，迁移回老的系统，这需要我们的新系统兼容老的数据，老的系统也兼容新的数据，从某种意义上来讲，迁移比实现新系统更加困难。

曾经看过很多简单的开关设计，有的开关设计在应用层次，通过一个 curl 语句调用，没有权限控制，这样的开关在服务池的每个节点都是不同步的、不一致的；还有的系统把开关配置放在中心化的配置系统、数据库或者缓存等，处理的每个请求都通过统一的开关来判断是否迁移等等，这样的开关有一个致命的缺点，服务请求在处理过程中，开关可能会变化，各个节点之间开关可能不同步、不一致，导致重复的请求可能走到新的逻辑又走了老的逻辑，如果新的逻辑和老的逻辑没有保证幂等性，这个请求就被重复处理了，如果是金融行业的应用，可能会导致资金损失，电商系统可能会导致发货并退款等问题。

这里面我们推荐使用订单开关，不管我们在什么维度上设计了开关，接收到服务请求后，我们在请求创建的关联实体（例如：订单）上标记开关，以后的任何处理流程，包括同步的和异步的处理流程，都通过订单上的开关来判断，而不是通过全局的或者基于配置的开关，这样在订单创建的时候，开关已经确定，不再变更，一旦一份数据不再发生变化，那么它永远是线程安全的，并且不会有不一致的问题。

这个模式在生产中使用比较频繁，建议每个企业都把这个模式作为设计评审的一项，如果不检查这一项，很多开发童鞋都会偷懒，直接在配置中或者数据库中做个开关就上线了。

4 总结

本文从一致性问题的实践出发，从大规模高并发服务化系统的实践经验中进行总结，列举导致不一致的具体问题，围绕着具体问题，总结出解决不一致的方法，并且抽象成模式，供大家在开发服务化系统的过程中参考。

另外，由于篇幅有限，还有一些关于分布式一致性的技术无法在一篇文章中与大家分享，包括：paxos 算法、raft 算法、zab 算法、nwr 算法、一致性哈希等，我会在后续文章中详细介绍。