# 分布式基础

## 分布式系统的特点

随着移动互联网的快速发展，互联网的用户数量越来越多，产生的数据规模也越来越大，对应用系统提出了更高的要求，我们的系统必须**支持高并发访问和海量数据处理**。

**分布式系统技术就是用来解决集中式架构的性能瓶颈问题**，来适应快速发展的业务规模，一般来说，分布式系统是建立在网络之上的硬件或者软件系统，彼此之间通过消息等方式进行通信和协调。

分布式系统的核心是可扩展性，通过对服务、存储的扩展，来提高系统的处理能力，通过对多台服务器协同工作，来完成单台服务器无法处理的任务，尤其是高并发或者大数据量的任务。

除了对可扩展性的需求，分布式系统还有不出现单点故障、服务或者存储无状态等特点。

单点故障（Single Point Failure）是指在系统中某个组件一旦失效，这会让整个系统无法工作，而不出现单点故障，单点不影响整体，就是分布式系统的设计目标之一；

无状态，是因为无状态的服务才能满足部分机器宕机不影响全部，可以随时进行扩展的需求。

**由于分布式系统的特点，在分布式环境中更容易出现问题，比如节点之间通信失败、网络分区故障、多个副本的数据不一致等**

## CAP理论

CAP理论可以表述为，一个分布式系统最多只能满足一致性(Consistency)，可用性(Availability)，分区容错性这**三项中的两项**

一致性是指所有节点同时看到相同的数据，即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，等同于所有节点拥有数据的最新版本。

可用性是指“任何时候，读写都是成功的”，即服务一直可用，而且是正常响应时间

分区容忍性具体是指“当部分节点出现消息丢失或者分区故障的时候，分布式系统仍然能够继续运行”，即系统容忍网络出现分区，并且在遇到某节点或网络分区之间网络不可达的情况下，仍然能够对外提供满足一致性和可用性的服务。

### CAP 理论的应用

一个分布式系统中不能同时满足CAP这3个需求，而只能满足其中两个，因此在使用cap定理的时候，我们需要抛出2其中一个

**不同业务对于一致性的要求是不同的**。举个例来讲，在微博上发表评论和点赞，用户对不一致是不敏感的，可以容忍相对较长时间的不一致，只要做好本地的交互，并不会影响用户体验；而我们在电商购物时，产品价格数据则是要求强一致性的，如果商家更改价格不能实时生效，则会对交易成功率有非常大的影响。

需要注意的是，CAP 理论中是忽略网络延迟的，也就是当事务提交时，节点间的数据复制一定是需要花费时间的。即使是同一个机房，从节点 A 复制到节点 B，由于现实中网络不是实时的，所以总会有一定的时间不一致。

### CA和CP架构的取舍

在分布式架构中为了保证数据的高可用，通常会把数据保存在多个副本中，网络分区是既成的现实，于是只能在可用性和一致性两者间做出选择。CAP 理论关注的是在绝对情况下，在工程上，**可用性和一致性并不是完全对立的**，我们关注的往往是如何在保持相对一致性的前提下，提高系统的可用性。

CP 架构：对于 CP 来说，放弃可用性，追求一致性和分区容错性。

ZooKeeper就是采用了 CP 一致性，ZooKeeper 是一个分布式的服务框架，主要用来解决分布式集群中应用系统的协调和一致性问题。其核心算法是 Zab，所有设计都是为了一致性。在 CAP 模型中，ZooKeeper 是 CP，这意味着面对网络分区时，为了保持一致性，它是不可用的。

AP 架构：对于 AP 来说，**放弃强一致性**，追求分区容错性和可用性，这是很多分布式系统设计时的选择，后面的 Base 也是根据 AP 来扩展的。

Eureka 是 Spring Cloud 微服务技术栈中的服务发现组件，Eureka 的各个节点都是平等的，几个节点挂掉不影响正常节点的工作，剩余的节点依然可以提供注册和查询服务，只要有一台 Eureka 还在，就能保证注册服务可用，只不过查到的信息可能不是最新的版本，不保证一致性。

## Base理论

Base 是三个短语的简写，即基本可用（Basically Available）、软状态（Soft State）和最终一致性（Eventually Consistent）。

Base理论的核心思想就是最终一致性，既然无法做到强一致性，但是每个应用可以根据自身的业务特点，采用适当的方式来实现最终一致性

### 基本可用

强调在系统出现不可预知故障的时候，允许丢失一部分可用性，相比正常系统，可能是延长响应，或者服务降级（举个例子，在双十一秒杀活动中，如果抢购人数太多超过了系统的 QPS 峰值，可能会排队或者提示限流，这就是通过合理的手段保护系统的稳定性，保证主要的服务正常，保证基本可用。）

### 软状态

软状态可用对应acid的原子性，在acid事务中，实现的是强一致性，要么全做要么全不做，所有用户看到的数据都是一致的。

原子性可以理解为一种硬状态，软状态则是允许在系统中的数据存在一个中间状态，并在该状态先不影响系统整体的可用性，即允许系统在多个不同节点的数据副本存在数据延时。

### 最终一致性

数据不可能一直是软状态，必须在一个时间期限之后达到各个节点的一致性，在期限过后，应当保证所有副本保持数据一致性，也就是达到数据的最终一致性。

强一致性

当更新操作完成之后，任何多个后续进程的访问都会返回最新的更新过的值，这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据 CAP 理论，这种实现需要牺牲可用性。

弱一致性

系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。用户读到某一操作对系统数据的更新需要一段时间，我们称这段时间为“不一致性窗口”。

最终一致性

最终一致性是弱一致性的特例，强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的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。

## CAP 及 Base 的关系

Base 理论是在 CAP 上发展的，CAP 理论描述了分布式系统中数据一致性、可用性、分区容错性之间的制约关系，当你选择了其中的两个时，就不得不对剩下的一个做一定程度的牺牲。

Base 理论则是对 CAP 理论的实际应用，也就是在分区和副本存在的前提下，通过一定的系统设计方案，放弃强一致性，实现基本可用，这是大部分分布式系统的选择，比如 NoSQL 系统、微服务架构。在这个前提下，如何把基本可用做到最好，就是分布式工程师们追求的，在这个课程中，我们也会有专门的模块来讲解高可用。

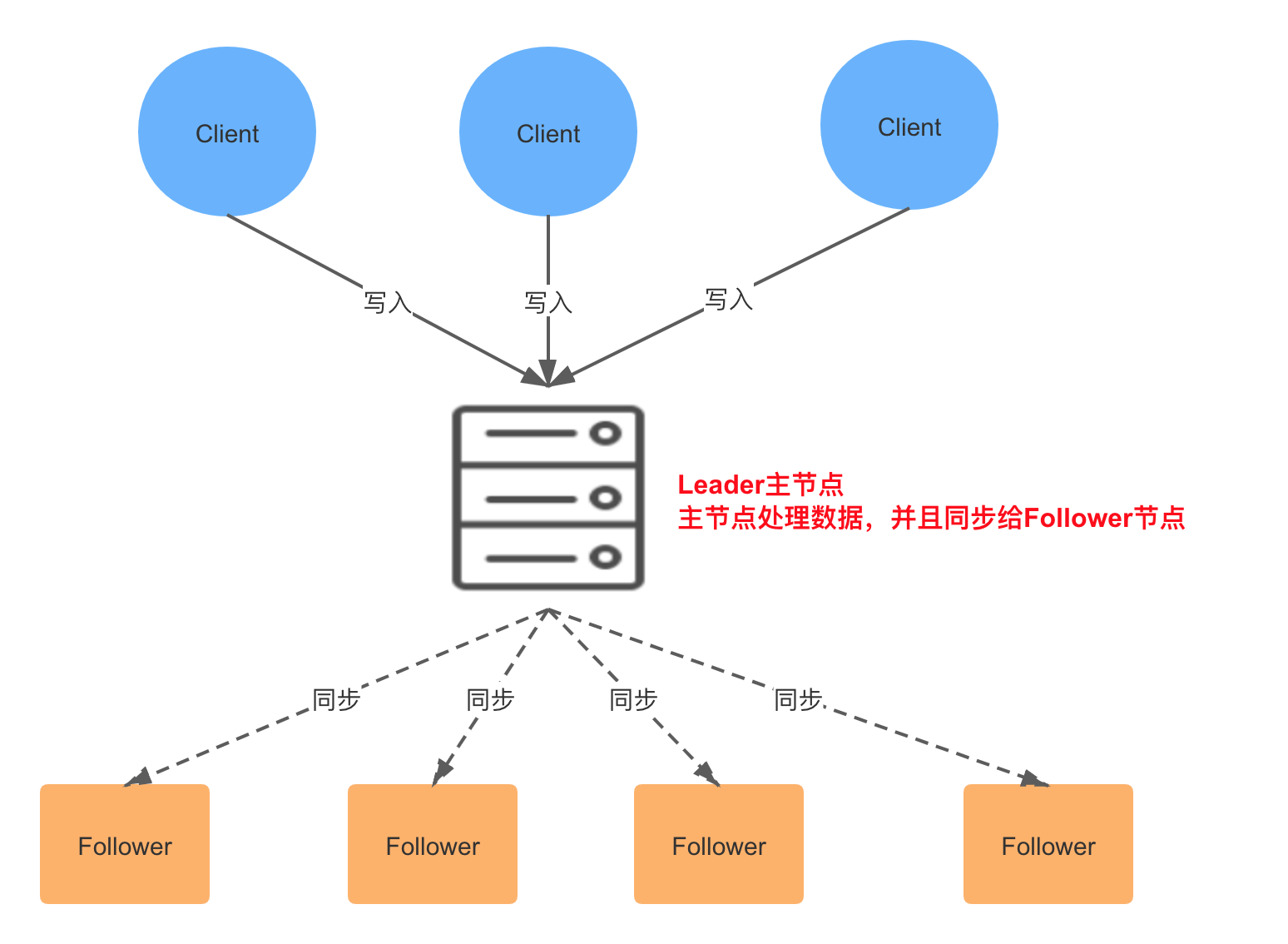
## Zookeeper如何保证数据一致性

在分布式场景中，ZooKeeper 的应用非常广泛，比如数据发布和订阅、命名服务、配置中心、注册中心、分布式锁等。

ZooKeeper 提供了一个类似于 Linux 文件系统的数据模型，和基于 Watcher 机制的分布式事件通知，这些特性都依赖 ZooKeeper 的高容错数据一致性协议。

Zab一致性协议

Zookeeper通过zab协议来保证分布式事务的一致性。Zab（原子广播协议）支持崩溃恢复，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间数据一致性。



在zookeeper集群中所有客户端的请求都是写入leader进程中，然后leader将数据同步到其他follow节点，在集群数据同步的过程中，如果出现 Follower 节点崩溃或者 Leader 进程崩溃时，都会通过 Zab 协议来保证数据一致性。

Zab协议的具体实现分为2部分

消息广播阶段

Leader 节点接受事务提交，并且将新的 Proposal 请求广播给 Follower 节点，收集各个节点的反馈，决定是否进行 Commit

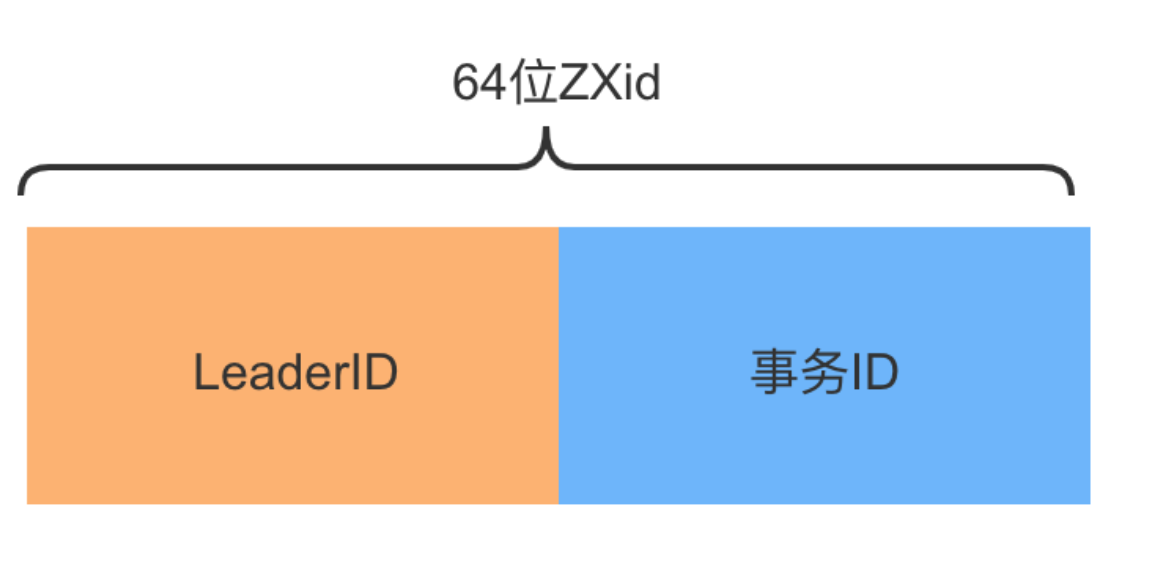
崩溃恢复阶段

如果在同步过程中出现 Leader 节点宕机，会进入崩溃恢复阶段，重新进行 Leader 选举，崩溃恢复阶段还包含数据同步操作，同步集群中最新的数据，保持集群的数据一致性。

整个 ZooKeeper 集群的一致性保证就是在上面两个状态之前切换，当 Leader 服务正常时，就是正常的消息广播模式；当 Leader 不可用时，则进入崩溃恢复模式，崩溃恢复阶段会进行数据同步，完成以后，重新进入消息广播阶段。

Zab协议中的zxid

Zxid是zab协议的一个事务编号。Zxid 是一个 64 位的数字，其中低 32 位是一个简单的单调递增计数器，针对客户端每一个事务请求，计数器加 1；而高 32 位则代表 Leader 周期年代的编号。这里 Leader 周期的英文是 epoch，可以理解为当前集群所处的年代或者周期



Zab 协议的实现也类似，每当有一个新的 Leader 选举出现时，就会从这个 Leader 服务器上取出其本地日志中最大事务的 Zxid，并从中读取 epoch 值，然后加 1，以此作为新的周期 ID。总结一下，高 32 位代表了每代 Leader 的唯一性，低 32 位则代表了每代 Leader 中事务的唯一性。

Zab流程分析

Zab 的具体流程可以拆分为消息广播、崩溃恢复和数据同步三个过程，下面我们分别进行分析



消息广播

在zookeeper中所有的事务请求都由leader节点来处理，其他服务器为follower，leader将客户端的事务请求转换为事务proposal，并将proposal转发给集群中的其他follower

完成广播之后，leader等待follower的反馈当过半的follower反馈信息后，Leader 将再次向集群内 Follower 广播 Commit 信息，Commit 信息就是确认将之前的 Proposal 提交。

Leader 节点的写入也是一个两步操作，第一步是广播事务操作，第二步是广播提交操作，其中过半数指的是反馈的节点数 >=N/2+1，N 是全部的 Follower 节点数量。

客户端的写请求进来之后，Leader 会将写请求包装成 Proposal 事务，并添加一个递增事务 ID，也就是 Zxid，Zxid 是单调递增的，以保证每个消息的先后顺序；

广播这个 Proposal 事务，Leader 节点和 Follower 节点是解耦的，通信都会经过一个先进先出的消息队列，Leader 会为每一个 Follower 服务器分配一个单独的 FIFO 队列，然后把 Proposal 放到队列中；

Follower 节点收到对应的 Proposal 之后会把它持久到磁盘上，当完全写入之后，发一个 ACK 给 Leader；

当 Leader 收到超过半数 Follower 机器的 ack 之后，会提交本地机器上的事务，同时开始广播 commit， Follower 收到 commit 之后，完成各自的事务提交。

崩溃恢复

消息广播机制quorum机制，解决了follower节点宕机的情况，当leader崩溃之后就需要zab协议来支持崩溃恢复，崩溃恢复可以保证在 Leader 进程崩溃的时候可以重新选出 Leader，并且保证数据的完整性。

进入崩溃恢复的情况

1. 初始化集群
2. Leader宕机崩溃
3. Leader 失去了半数的机器支持，与集群中超过一半的节点断连

崩溃恢复将会开启新一轮的leader选举，选举好的leader会和过半的follower节点进行数据同步，是数据一致，当与过半的机器同步完成后，就退出恢复模式， 然后进入消息广播模式。

Zab 中的节点有三种状态，伴随着的 Zab 不同阶段的转换，节点状态也在变化：

Following：当前节点是follower节点

Leading：当前节点是leader节点，负者协调事务

Election、looking：节点处于选举状态

选举过程

。。。。

数据同步

崩溃恢复完成选举以后，接下来的工作就是数据同步，在选举过程中，通过投票已经确认 Leader 服务器是最大Zxid 的节点，同步阶段就是利用 Leader 前一阶段获得的最新Proposal历史，同步集群中所有的副本。

# 分布式事务

## 分布式事务解决方案

分布式事务关注的是分布式场景下如何处理事务，是指事务的参与者、支持事务操作的服务器、存储等资源分别位于分布式系统的不同节点之上。

简单来说，分布式事务就是一个业务操作，是由多个细分操作完成的，而这些细分操作又分布在不同的服务器上；事务，就是这些操作要么全部成功执行，要么全部不执行。

### 分布式事务产生的原因

分布式事务是伴随着系统拆分出现的，分布式系统解决了海量数据服务对扩展性的要求，但是增加了架构上的复杂性，在这一点上，分布式事务就是典型的体现。

在实际开发中，分布式事务产生的原因主要来**源于存储**和**服务的拆分**。

### 分布式事务解决方案

分布式事务的解决方案，典型的有两阶段和三阶段提交协议、 TCC 分段提交，和基于消息队列的最终一致性设计。

2PC 两阶段提交

两阶段提交（2PC，Two-phase Commit Protocol）是非常经典的强一致性、中心化的原子提交协议，在各种事务和一致性的解决方案中，都能看到两阶段提交的应用。

3PC 三阶段提交

三阶段提交协议（3PC，Three-phase\_commit\_protocol）是在 2PC 之上扩展的提交协议，主要是为了解决两阶段提交协议的阻塞问题，从原来的两个阶段扩展为三个阶段，增加了超时机制。

TCC 分段提交

TCC 是一个分布式事务的处理模型，将事务过程拆分为 Try、Confirm、Cancel 三个步骤，在保证强一致性的同时，最大限度提高系统的可伸缩性与可用性。

两阶段、三阶段以及 TCC 协议在后面的课程中我会详细介绍，接下来介绍几种系统设计中常用的一致性解决方案。

基于消息补偿的最终一致性

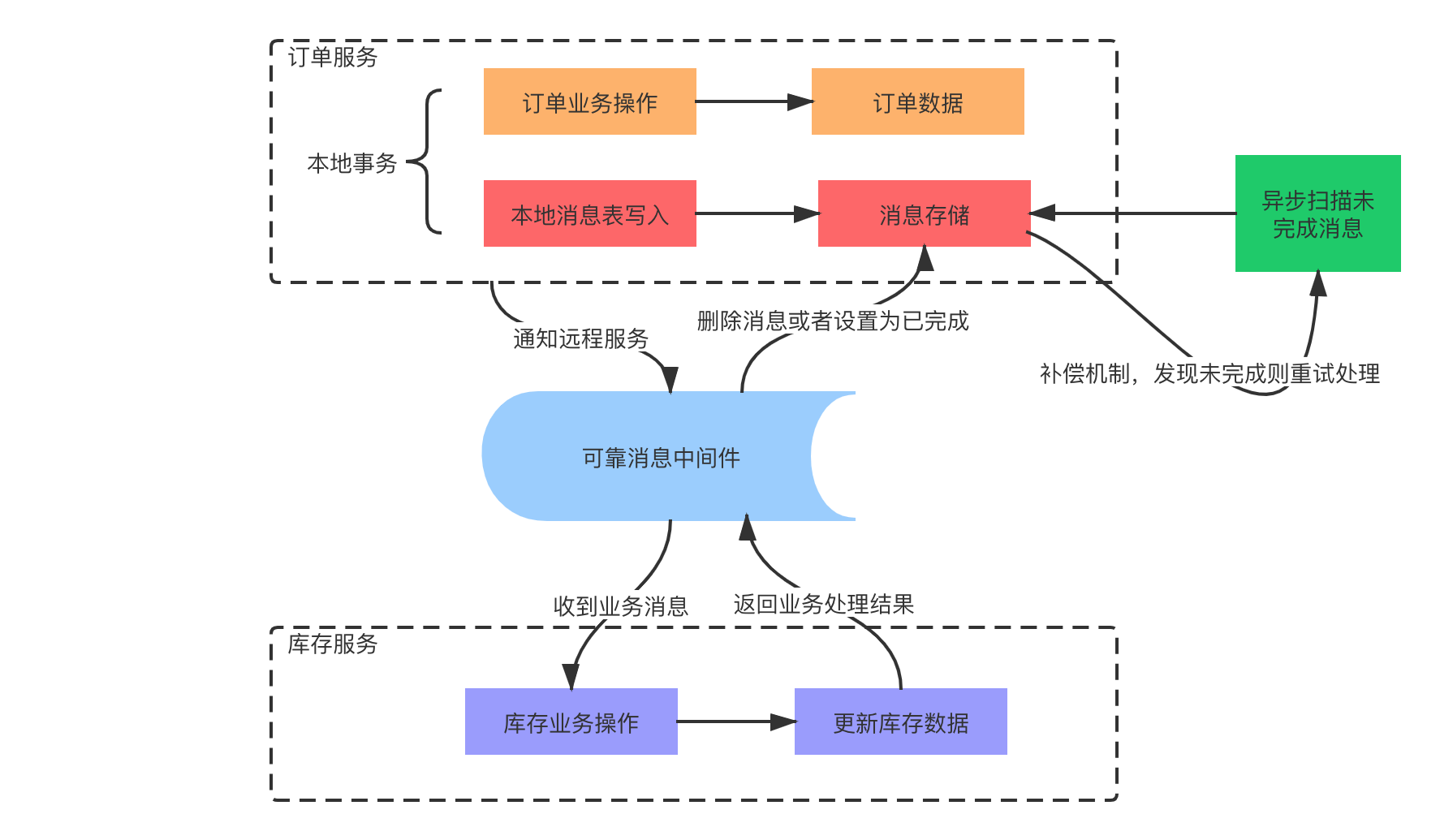
异步化在分布式系统设计中随处可见，基于消息队列的最终一致性就是一种异步事务机制，在业务中广泛应用。

在具体实现上，基于消息补偿的一致性主要有本地消息表和第三方可靠消息队列等。

下面介绍一下本地消息表，本地消息表的方案最初是由 ebay 的工程师提出，核心思想是将分布式事务拆分成本地事务进行处理，通过消息日志的方式来异步执行。

本地消息表是一种业务耦合的设计，消息生产方需要额外建一个事务消息表，并记录消息发送状态，消息消费方需要处理这个消息，并完成自己的业务逻辑，另外会有一个异步机制来定期扫描未完成的消息，确保最终一致性。

下面我们用下单减库存业务来简单模拟本地消息表的实现过程



（1）系统收到下单请求，将订单业务数据存入到订单库中，并且同时存储该订单对应的消息数据，比如购买商品的 ID 和数量，消息数据与订单库为同一库，更新订单和存储消息为一个本地事务，要么都成功，要么都失败。

（2）库存服务通过消息中间件收到库存更新消息，调用库存服务进行业务操作，同时返回业务处理结果。

（3）消息生产方，也就是订单服务收到处理结果后，将本地消息表的数据删除或者设置为已完成。

（4）设置异步任务，定时去扫描本地消息表，发现有未完成的任务则重试，保证最终一致性。

以上就是基于本地消息表一致性的主流程，在具体实践中，还有许多分支情况，比如消息发送失败、下游业务方处理失败等，

## 二阶段提交和三阶段提交

二阶段和三阶段提交协议都是引入了一个协调者的组件来统一调度所有分布式节点的执行，让当前节点知道其他节点的任务执行状态，通过通知和表决的方式，决定执行 Commit 还是 Rollback 操作。

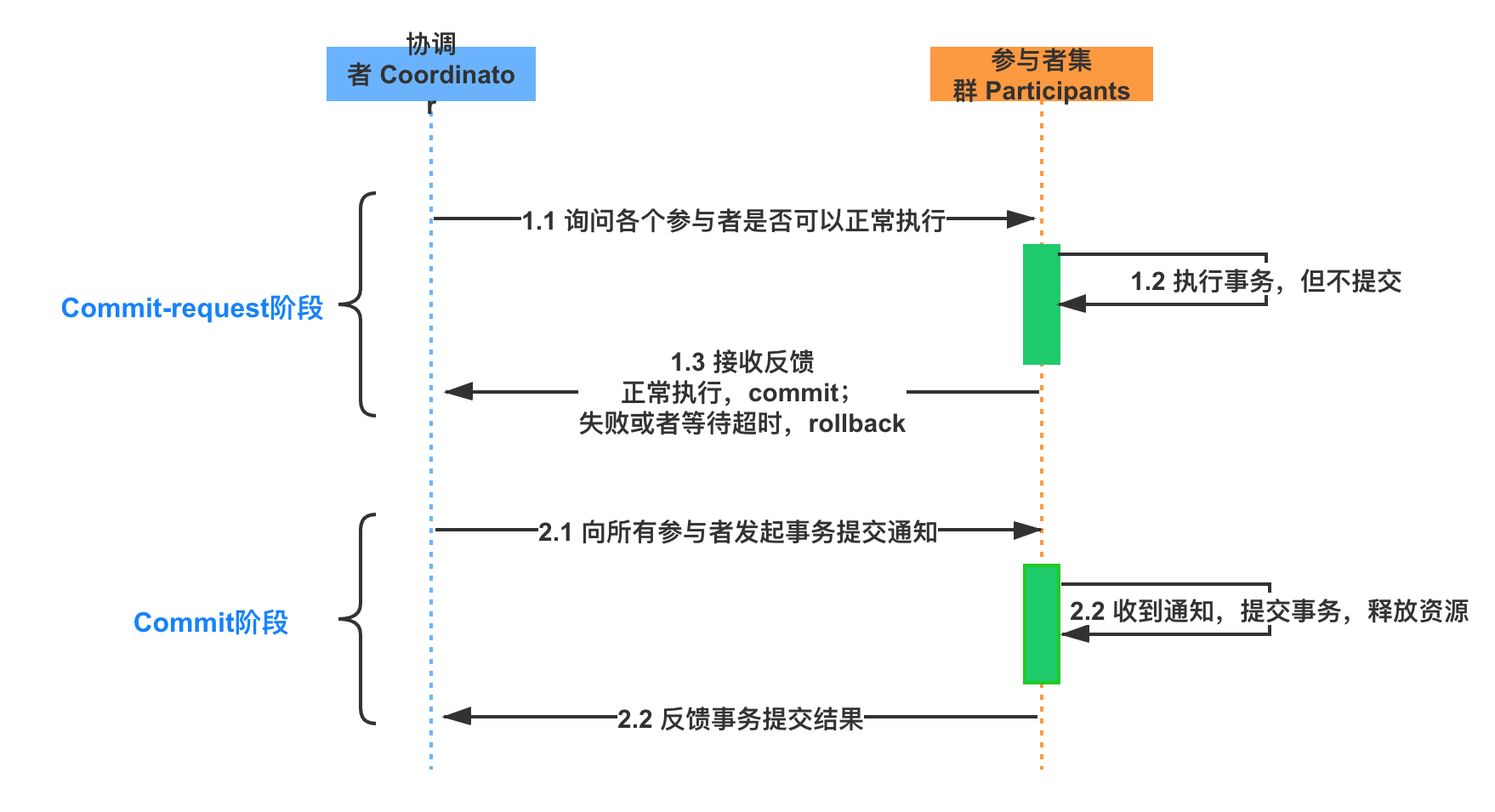
### 二阶段提交协议

二阶段提交算法的成立是基于以下假设的：

在该分布式系统中，存在一个节点作为协调者（Coordinator），其他节点作为参与者（Participants），且节点之间可以进行网络通信；

所有节点都采用预写式日志，日志被写入后被保存在可靠的存储设备上，即使节点损坏也不会导致日志数据的丢失；

所有节点不会永久性损坏，即使损坏后仍然可以恢复。

两阶段提交中的两个阶段，指的是 Commit-request 阶段和 Commit 阶段，两阶段提交的流程如下：

提交请求阶段

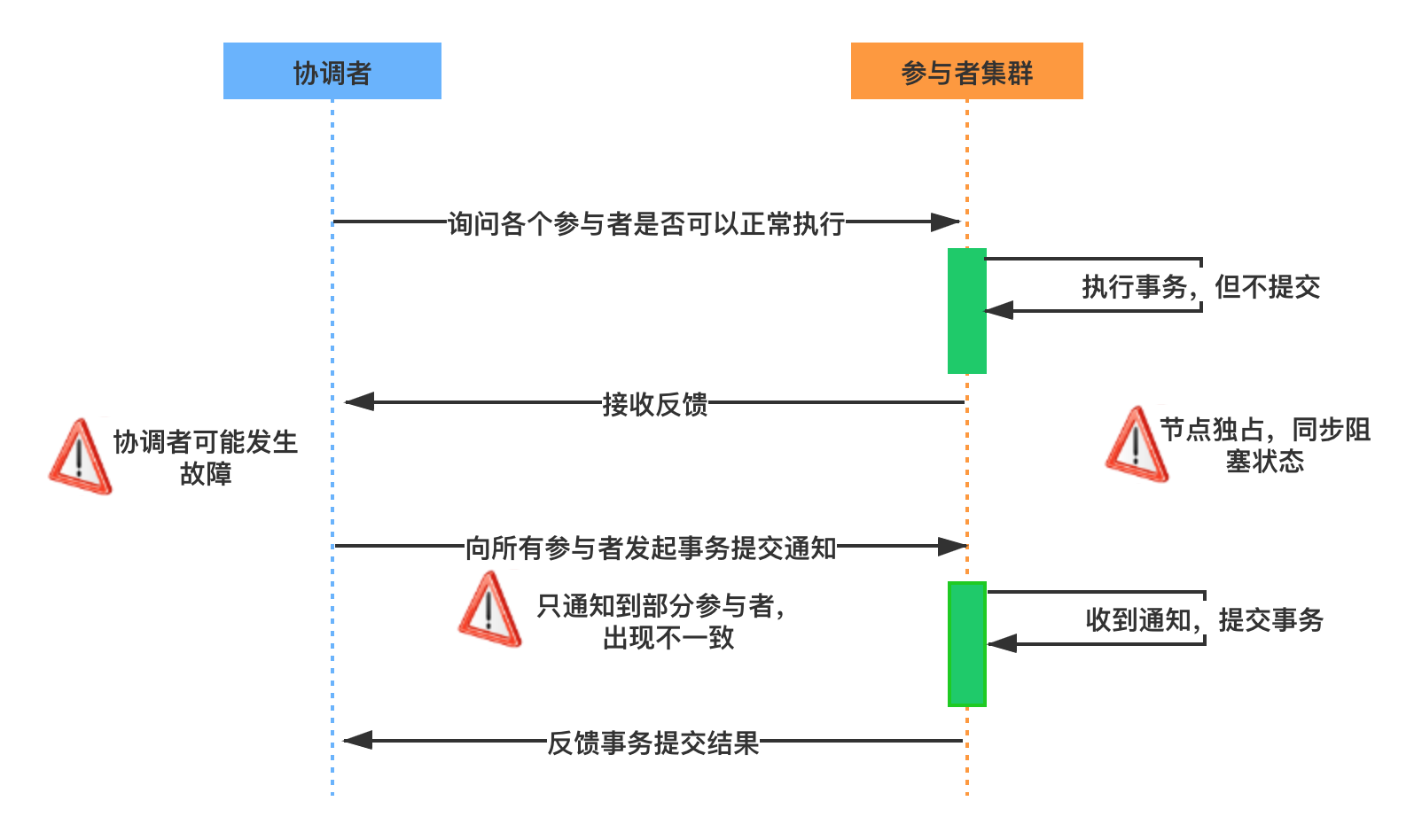
在提交请求阶段，协调者将通知事务参与者准备提交事务，然后进入表决过程。在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地事务执行成功）或取消（本地事务执行故障），在第一阶段，参与节点并没有进行Commit操作

提交阶段

在提交阶段，协调者将基于第一个阶段的投票结果进行决策：提交或取消这个事务。这个结果的处理和前面基于半数以上投票的一致性算法不同，**必须当且仅当所有的参与者同意提交，协调者才会通知各个参与者提交事务，否则协调者将通知各个参与者取消事务**。

参与者在接收到协调者发来的消息后将执行对应的操作，也就是本地 Commit 或者 Rollback。

二阶段提交存在的问题



资源被同步阻塞

在执行过程中，所有参与节点都是事务独占状态，当参与者占有公共资源时，那么第三方节点访问公共资源会被阻塞。

协调者可能出现单点故障

**一旦协调者发生故障，参与者会一直阻塞下去。**

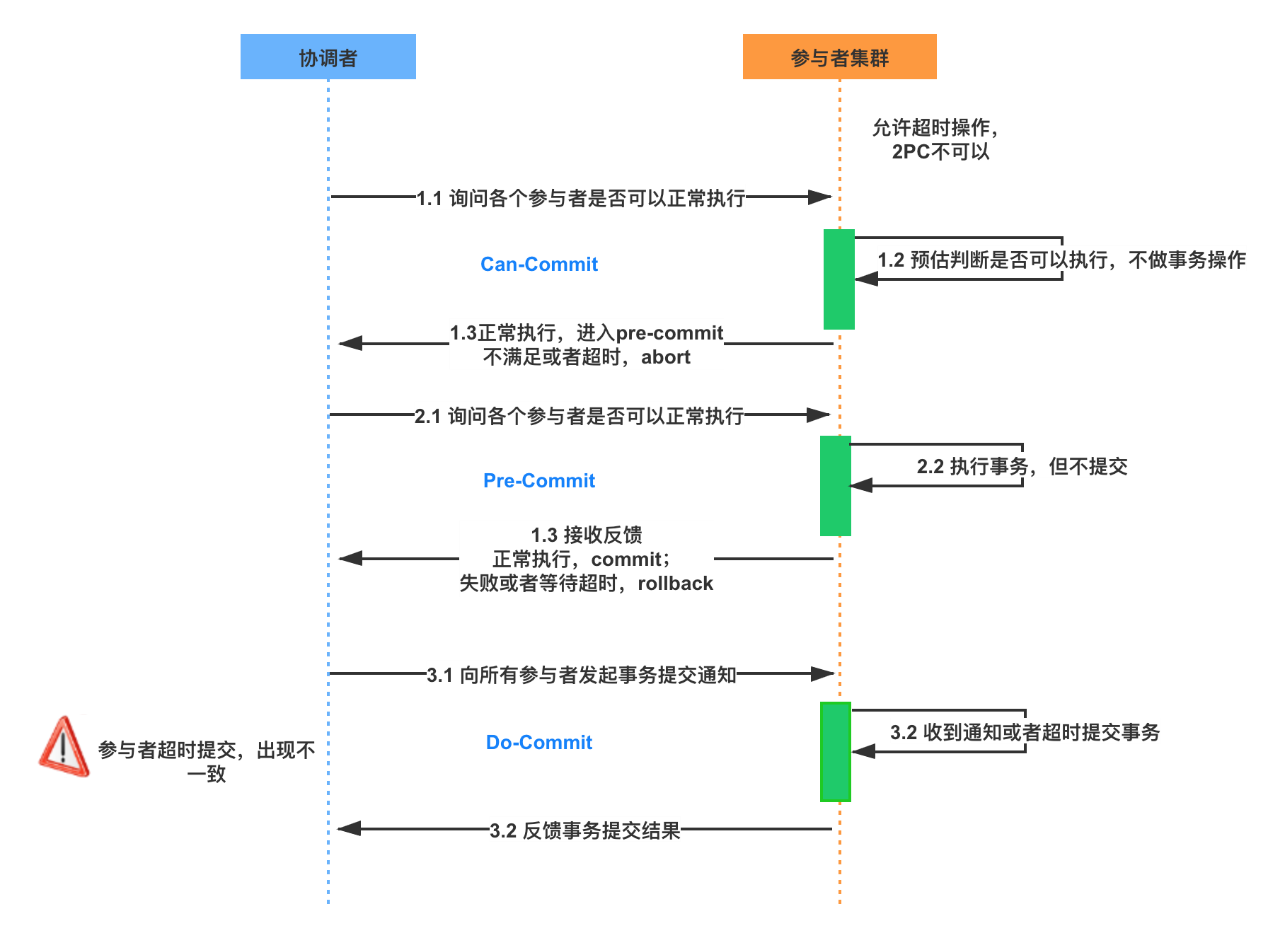
在 Commit 阶段出现数据不一致

在第二阶段中，假设协调者发出了事务 Commit 的通知，但是由于网络问题该通知仅被一部分参与者所收到并执行 Commit，其余的参与者没有收到通知，一直处于阻塞状态，那么，这段时间就产生了数据的不一致性。

### 三阶段提交协议

为了解决二阶段协议中的同步阻塞等问题，三阶段提交协议在协调者和参与者中都引入了超时机制，并且把两阶段提交协议的第一个阶段拆分成了两步：询问，然后再锁资源，最后真正提交。

三阶段中的 Three Phase 分别为 CanCommit、PreCommit、DoCommit 阶段。



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 Can-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 阶段

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

A. 假如协调者从所有的参与者获得的反馈都是 Yes 响应，那么就会进行事务的预执行：

发送预提交请求，协调者向参与者发送 PreCommit 请求，并进入 Prepared 阶段；

事务预提交，参与者接收到 PreCommit 请求后，会执行事务操作；

响应反馈，如果参与者成功执行了事务操作，则返回 ACK 响应，同时开始等待最终指令。

B. 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，那么就中断事务：

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，执行事务的中断。

DoCommit 阶段

该阶段进行真正的事务提交，也可以分为以下两种情况。

A. 执行提交

发送提交请求。协调者接收到参与者发送的 ACK 响应后，那么它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。

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。

响应反馈。事务提交完之后，向协调者发送 ACK 响应。

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。

B. 中断事务

协调者没有接收到参与者发送的 ACK 响应，可能是因为接受者发送的不是 ACK 响应，也有可能响应超时了，那么就会执行中断事务。

C.超时提交

参与者如果没有收到协调者的通知，超时之后会执行 Commit 操作

#### 三阶段提交做了哪些改进

引入超时机制

在 2PC 中，只有协调者拥有超时机制，如果在一定时间内没有收到参与者的消息则默认失败，3PC 同时在协调者和参与者中都引入超时机制。

添加预提交阶段

在 2PC 的准备阶段和提交阶段之间，插入一个准备阶段，使 3PC 拥有 CanCommit、PreCommit、DoCommit 三个阶段，PreCommit 是一个缓冲，保证了在最后提交阶段之前各参与节点的状态是一致的。

三阶段提交协议存在的问题

三阶段提交协议同样存在问题，具体表现为，在阶段三中，如果参与者接收到了 PreCommit 消息后，出现了不能与协调者正常通信的问题，在这种情况下，参与者依然会进行事务的提交，这就出现了数据的不一致性。

两阶段和三阶段提交的应用

两阶段提交是一种比较精简的一致性算法/协议，很多关系型数据库都是采用两阶段提交协议来完成分布式事务处理的，典型的比如 MySQL 的 XA 规范。

在事务处理、数据库和计算机网络中，两阶段提交协议提供了分布式设计中的数据一致性的保障，整个事务的参与者要么一致性全部提交成功，要么全部回滚。MySQL Cluster 内部数据的同步就是用的 2PC 协议。

#### MySQL 的主从复制

在 MySQL 中，二进制日志是 server 层，主要用来做主从复制和即时点恢复时使用的；而事务日志（Redo Log）是 InnoDB 存储引擎层，用来保证事务安全的。

在数据库运行中，需要保证 Binlog 和 Redo Log 的一致性，如果顺序不一致， 则意味着 Master-Slave 可能不一致。

在开启 Binlog 后，如何保证 Binlog 和 InnoDB redo 日志的一致性呢？MySQL 使用的就是二阶段提交，内部会自动将普通事务当做一个 XA 事务（内部分布式事务）来处理：

Commit 会被自动的分成 Prepare 和 Commit 两个阶段；

Binlog 会被当做事务协调者（Transaction Coordinator），Binlog Event 会被当做协调者日志。

关于 XA 规范的具体实现，会在后面的课时中分享。

## TCC 事务模型

TCC 提出了一种新的事务模型，**基于业务层面的事务定义**，**锁粒度完全由业务自己控制**，目的是解决复杂业务中，跨表跨库等大颗粒度资源锁定的问题。TCC 把事务运行过程分成 Try、Confirm / Cancel 两个阶段，每个阶段的逻辑由业务代码控制，避免了长事务，可以获取更高的性能。

TCC 的各个阶段

**Try 阶段**：调用 Try 接口，尝试执行业务，完成所有业务检查，预留业务资源。

Confirm 或 Cancel 阶段：两者是互斥的，只能进入其中一个，并且都满足幂等性，允许失败重试。

**Confirm 操作**：对业务系统做确认提交，确认执行业务操作，不做其他业务检查，只使用 Try 阶段预留的业务资源。

**Cancel 操作**：在业务执行错误，需要回滚的状态下执行业务取消，释放预留资源。

Try 阶段失败可以 Cancel，如果 Confirm 和 Cancel 阶段失败了怎么办？

TCC 中会添加事务日志，如果 Confirm 或者 Cancel 阶段出错，则会进行重试，所以这两个阶段需要支持幂等；**如果重试失败，则需要人工介入进行恢复和处理等**。

应用 TCC 的优缺点

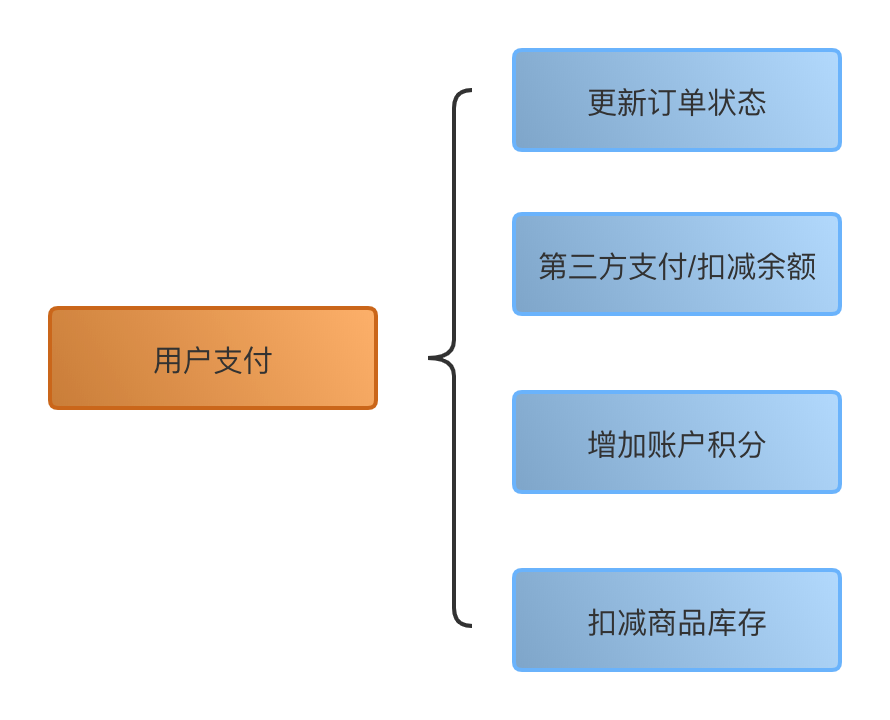
实际开发中，TCC 的本质是把数据库的二阶段提交上升到微服务来实现，从而避免数据库二阶段中长事务引起的低性能风险。

所以说，TCC 解决了跨服务的业务操作原子性问题，比如下订单减库存，多渠道组合支付等场景，通过 TCC 对业务进行拆解，可以让应用自己定义数据库操作的粒度，可以降低锁冲突，提高系统的业务吞吐量。

TCC 的不足主要体现在对微服务的侵入性强，TCC 需要对业务系统进行改造，业务逻辑的每个分支都需要实现 try、Confirm、Cancel 三个操作，并且 Confirm、Cancel 必须保证幂等。

另外 TCC 的事务管理器要记录事务日志，也会损耗一定的性能。

一个电商中的支付业务中，用户在支付以后，需要进行更新订单状态、扣减账户余额、增加账户积分和扣减商品操作。



**业务逻辑拆解**

我们把订单业务拆解为以下几个步骤：

订单更新为支付完成状态

扣减用户账户余额

增加用户账户积分

扣减当前商品的库存

如果不使用事务，上面的几个步骤都可能出现失败，最终会造成大量的数据不一致，比如订单状态更新失败，扣款却成功了；或者扣款失败，库存却扣减了等情况，这个在业务上是不能接受的，会出现大量的客诉。

如果直接应用事务，不使用分布式事务，比如在代码中添加 Spring 的声明式事务 @Transactional 注解，这样做实际上是在事务中嵌套了远程服务调用，一旦服务调用出现超时，事务无法提交，就会导致数据库连接被占用，出现大量的阻塞和失败，会导致服务宕机。另一方面，如果没有定义额外的回滚操作，比如遇到异常，非 DB 的服务调用失败时，则无法正确执行回滚。

业务系统改造

下面应用 TCC 事务，需要对业务代码改造，抽象 Try、Confirm 和 Cancel 阶段。

Try 操作

Try 操作一般都是锁定某个资源，设置一个预备的状态，冻结部分数据。比如，订单服务添加一个预备状态，修改为 UPDATING，也就是更新中的意思，冻结当前订单的操作，而不是直接修改为支付成功。

库存服务设置冻结库存，可以扩展字段，也可以额外添加新的库存冻结表。积分服务和库存一样，添加一个预增加积分，比如本次订单积分是 100，添加一个额外的存储表示等待增加的积分，账户余额服务等也是一样的操作。

Confirm 操作

Confirm 操作就是把前边的 Try 操作锁定的资源提交，类比数据库事务中的 Commit 操作。在支付的场景中，包括订单状态从准备中更新为支付成功；库存数据扣减冻结库存，积分数据增加预增加积分。

Cancel 操作

Cancel 操作执行的是业务上的回滚处理，类比数据库事务中的 Rollback 操作。首先订单服务，撤销预备状态，还原为待支付状态或者已取消状态，库存服务删除冻结库存，添加到可销售库存中，积分服务也是一样，将预增加积分扣减掉。

执行业务操作

下面来分析业务的实际执行操作，首先业务请求过来，开始执行 Try 操作，如果 TCC 分布式事务框架感知到各个服务的 Try 阶段都成功了以后，就会执行各个服务的 Confirm 逻辑。

如果 Try 阶段有操作不能正确执行，比如订单失效、库存不足等，就会执行 Cancel 的逻辑，取消事务提交。

与 2PC/XA 两阶段提交的区别

2PC/XA 是数据库或者存储资源层面的事务，实现的是强一致性，在两阶段提交的整个过程中，一直会持有数据库的锁。

TCC 关注业务层的正确提交和回滚，在 Try 阶段不涉及加锁，是业务层的分布式事务，关注最终一致性，不会一直持有各个业务资源的锁。

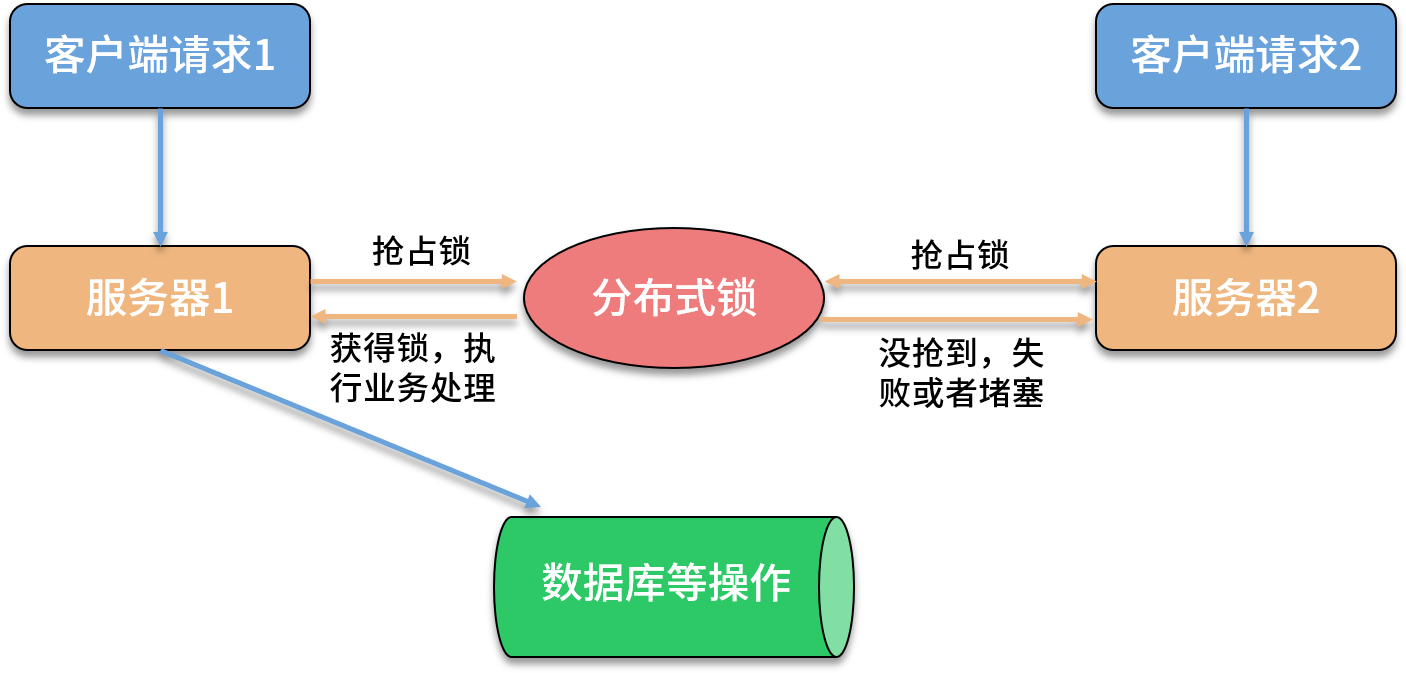
TCC 的核心思想是针对每个业务操作，都要添加一个与其对应的确认和补偿操作，同时把相关的处理，从数据库转移到业务中，以此实现跨数据库的事务。

TCC 分布式服务组件

在业务中引入 TCC 一般是依赖单独的 TCC 事务框架，可以选择自研或者应用开源组件。TCC 框架扮演了资源管理器的角色，常用的 TCC 开源组件有 Tcc-transaction、ByteTCC、Spring-cloud-rest-tcc 等。

## 分布式锁应用场景

分布式场景下解决并发问题，需要应用分布式锁技术。如上图所示，分布式锁的目的是保证在分布式部署的应用集群中，多个服务在请求同一个方法或者同一个业务操作的情况下，对应业务逻辑只能被一台机器上的一个线程执行，避免出现并发问题。



### 分布式锁的常用实现

实现分布式锁目前有三种流行方案，即基于数据库、Redis、ZooKeeper 的方案。

基于关系型数据库

利用数据库的唯一性来实现资源锁定，比如唯一索引和主键

以唯一索引为例，创建一张锁表，定义方法或者资源名、失效时间等字段，同时针对加锁的信息添加唯一索引，比如方法名，当要锁住某个方法或资源时，就在该表中插入对应方法的一条记录，插入成功表示获取了锁，想要释放锁的时候就删除这条记录。

存在的问题

1. 超时无法失效，如果释放锁的操作失败，那么锁一直存在，其他线程或者进程无法获取到锁；解决这个问题，可以添加独立的定时任务，通过时间戳对比等方式，删除超时数据。
2. 不可重入，可重入性是锁的一个重要特性，以 Java 语言为例，常见的 Synchronize、Lock 等都支持可重入。在数据库实现方式中，同一个线程在没有释放锁之前无法再次获得该锁，因为数据已经存在，再次插入会失败。实现可重入，需要改造加锁方法，额外存储和判断线程信息，不阻塞获得锁的线程再次请求加锁。
3. 无法实现阻塞：其他线程在请求对应方法时，插入数据失败会直接返回，不会阻塞线程，如果需要阻塞其他线程，需要不断的重试 insert 操作，直到数据插入成功，这个操作是服务器和数据库资源的极大浪费。

可以看到，借助数据库实现一个完备的分布式锁，存在很多问题，并且读写数据库需要一定的性能，可能会影响业务执行的耗时。

基于Redis实现分布式锁

相比基于数据库实现分布式锁，缓存的性能更好，并且各种缓存组件也提供了多种集群方案，可以解决单点问题。

使用Redis的命令setnx和expire来实现锁

在 Redis 中，setnx 是「set if not exists」如果不存在，则 SET 的意思，当一个线程执行 setnx 返回 1，说明 key 不存在，该线程获得锁；当一个线程执行 setnx 返回 0，说明 key 已经存在，那么获取锁失败，expire 就是给锁加一个过期时间。

if(setnx(key,value)==1){

expire(key,expireTime)

try{

//业务处理

}finally{

//释放锁

del(key)

}

}

使用 setnx 和 expire 有一个问题，这两条命令可能不会同时失败，不具备原子性，如果一个线程在执行完 setnx 之后突然崩溃，导致锁没有设置过期时间，那么这个锁就会一直存在，无法被其他线程获取。

为了解决这个问题，在 Redis 2.8 版本中，**添加了 SETEX 命令，SETEX 支持 setnx 和 expire 指令组合的原子操作，解决了加锁过程中失败的问题**。

基于zookeeper实现分布式锁

Zookeeper有4中节点类型，包括持久节点，临时节点，持久顺序节点，临时顺序节点，利用zookeeper的临时顺序节点来实现分布式锁

当客户端对某个方法加锁时，在 ZooKeeper 中该方法对应的指定节点目录下，生成一个唯一的临时有序节点。判断是否获取锁，只需要判断持有的节点是否是有序节点中序号最小的一个，当释放锁的时候，将这个临时节点删除即可，这种方式可以避免服务宕机导致的锁无法释放而产生的死锁问题。

 ZooKeeper 实现分布式锁的算法流程，根节点为 /lock：

客户端连接 ZooKeeper，并在 /lock 下创建临时有序子节点，第一个客户端对应的子节点为 /lock/lock01/00000001，第二个为 /lock/lock01/00000002；

其他客户端获取 /lock01 下的子节点列表，判断自己创建的子节点是否为当前列表中序号最小的子节点；

如果是则认为获得锁，执行业务代码，否则通过 watch 事件监听 /lock01 的子节点变更消息，获得变更通知后重复此步骤直至获得锁；

完成业务流程后，删除对应的子节点，释放分布式锁。

# 分布式服务

## RPC远程服务

RPC（Remote Procedure Call）是一种进程间通信方式，RPC（远程过程调用协议），它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议

RPC 允许程序调用另一个地址空间的过程或函数，而不用程序员显式编码这个远程调用的细节。即无论是调用本地接口/服务的还是远程的接口/服务，本质上编写的调用代码基本相同。

比如两台服务器 A、B，一个应用部署在 A 服务器上，想要调用 B 服务器上应用提供的函数或者方法，由于不在一个内存空间，则不能直接调用，这时候就可以应用 RPC 框架的实现来解决。

RPC如何实现

早期的远程服务调用一般是通过 RMI 或 Hessian 等工具实现，以 Java RMI 为例，RMI 是 Java 语言中 RPC 的一种实现方式。

Java RMI（Java 远程方法调用，Java Remote Method Invocation）是 Java 编程语言里，一种用于实现远程过程调用的应用程序编程接口。应用 Java RMI，可以让某个 Java 虚拟机上的对象调用另一个 Java 虚拟机中的对象上的方法。

Java RMI 实现主要依赖 java.rmi 包下面的工具类，具体流程包括继承 Remote 实现远程接口，开发业务逻辑，创建 Server 并且注册远程对象，客户端创建 Client 调用远程方法等。关于 RMI 的实现细节，由于实际开发中很少应用，这里不展开讲解了。

### RPC框架

开源社区里有许多优秀的 RPC 框架，比如常用的 Dubbo、Thrift、gRPC 等

Apache Dubbo

Dubbo 是阿里巴巴公司开源的一个高性能 Java 分布式服务框架，目前已经成为 Apache 顶级项目。Dubbo 可以通过高性能的 RPC 实现服务的输出和输入，支持服务治理，提供了控制台界面，可以独立应用，也可以和 Spring 框架无缝集成。

Apache Thrift

Thrift 起源于 Facebook，和 Dubbo 一样，后来被提交 Apache 基金会将 Thrift 作为一个开源项目。Facebook 创造 Thrift 的目的是为了解决 Facebook 各系统间大数据量的传输通信，以及系统间语言环境不同需要跨平台的问题。

### RPC 框架用到哪些技术

实现分布式服务框架，首先要解决不同节点之间通讯的问题，需要在客户端和服务器之间建立 TCP 连接，远程过程调用的所有交换的数据都在这个连接里传输。

一般来说，建立通信可以使用成熟的网络通信框架，比如 Java 语言中的 Netty，这是一个优秀的网络通信框架。在 Dubbo、Motan 中都应用了 Netty。

如何进行网络传输

建立通信之后，节点之间数据传输采用什么协议，也就是选择什么样的二进制数据格式组织；传输的数据如何序列化和反序列化，比如在 Dubbo 中，传输协议默认使用 Dubbo 协议，序列化支持选择 Hessian、Kryo、Protobuf 等不同方式。

如何进行服务注册和发现

**服务注册和发现，也就是服务寻址**

服务注册，需要服务提供者启动后主动把服务注册到注册中心，注册中心存储了该服务的 IP、端口、调用方式（协议、序列化方式）等信息。

服务发现，当服务消费者第一次调用服务时，会通过注册中心找到相应的服务提供方地址列表，并缓存到本地，以供后续使用。当消费者再次调用服务时，不会再去请求注册中心，而是直接通过负载均衡算法从 IP 列表中取一个服务提供者调用服务。

## API网关

在微服务架构中，一个大应用被拆分成多个小的服务，这些微服务自成体系，可以独立部署和提供对外服务。一般来说，微服务的调用规范主要有 RPC 和 Restful API 两种，API 网关主要针对的是后面一种，也就是以 Spring Cloud 为代表的微服务解决方案。

一个实际场景

假设我们要使用微服务构建一个电商平台，一般来说需要订单服务、商品服务、交易服务、会员服务、评论服务、库存服务等。

在微服务架构下，每个服务都是独立部署，如果直接调用，系统设计可能是这样的：

客户端直接调用服务

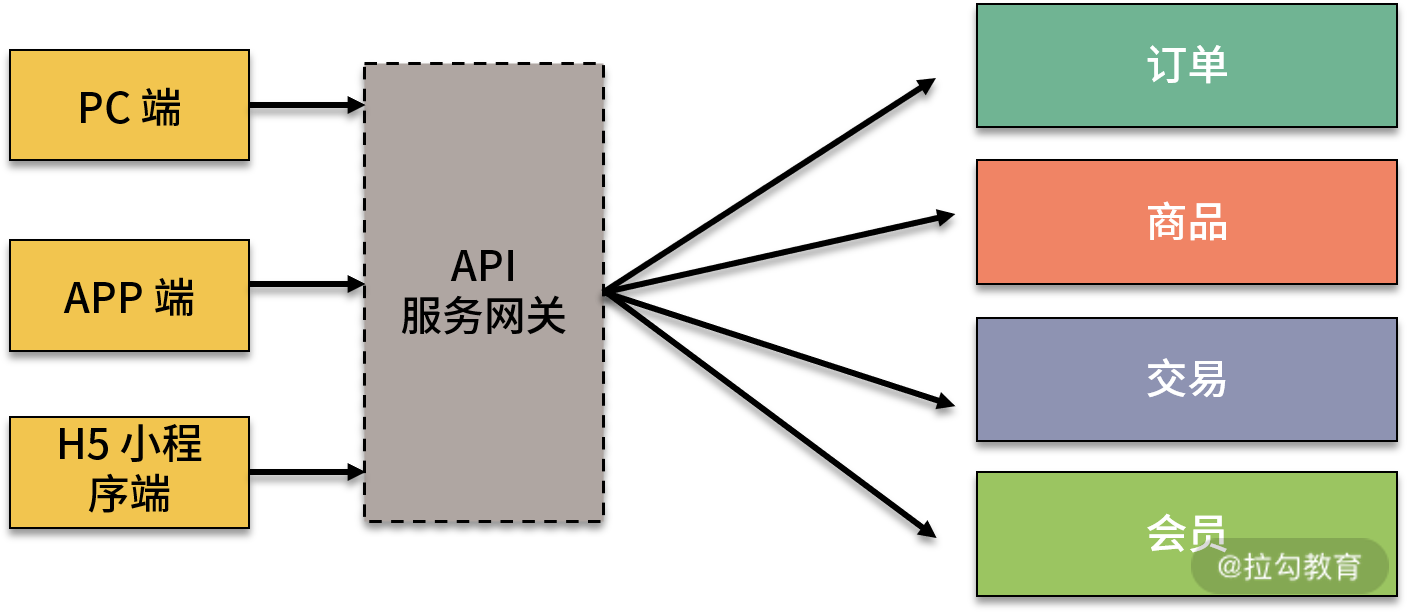


各个调用端单独去发起连接，会出现很多问题，比如不容易监控调用流量，出现问题不好确定来源，服务之间调用关系混乱等。

如何解决这个局面呢

针对这些问题，**一个常用的解决方案是使用 API 服务网关**。**在微服务设计中，需要隔离内外部调用，统一进行系统鉴权、业务监控等，API 服务网关是一个非常合适的切入口**。

**通过引入 API 网关这一角色，可以高效地实现微服务集群的输出，节约后端服务开发成本，减少上线风险，并为服务熔断、灰度发布、线上测试等提供解决方案。**



使用网关，可以优化微服务架构中系统过于分散的弊端，使得架构更加优雅，选择一个适合的 API 网关，可以有效地简化开发并提高运维与管理效率。

### 应用网关的优劣

API 网关在微服务架构中并不是一个必需项目，而是系统设计的一个解决方案，用来整合各个不同模块的微服务，统一协调服务。

API 网关自身也是一个服务，网关封装了系统内部架构，为每个客户端提供了一个定制的 API。从面向对象设计的角度看，它与外观模式（Facade Pattern）类似，外观模式的定义是，外部与一个子系统的通信必须通过一个统一的外观对象进行，为子系统中的一组接口提供一个一致的界面，这一点和 API 网关的作用非常类似。

除了封装内部系统之外，API 网关作为一个系统访问的切面，还可以添加身份验证、监控、负载均衡、限流、降级与应用检测等功能。

通过在微服务架构中引入 API 网关，可以带来以下的收益：

API 服务网关对外提供统一的入口供客户端访问，隐藏系统架构实现的细节，让微服务使用更为友好；

借助 API 服务网关可统一做切面任务，避免每个微服务自己开发，提升效率，使系统更加标准化；

通过 API 服务网关，可以将异构系统进行统一整合，比如外部 API 使用 HTTP 接口，内部微服务可以使用一些性能更高的通信协议，然后在网关中进行转换，提供统一的外部 REST 接口；

通过微服务的统一访问控制，可以更好地实现鉴权，提高系统的安全性。

API 网关并不是一个必需的角色，在系统设计中引入网关，也会导致系统复杂性增加，带来下面的问题：

在发布和部署阶段需要管理网关的配置，保证外部 API 访问的是正常的服务实例；

API 服务网关需要实现一个高可用伸缩性强的服务，避免单点失效，否则会成为系统的瓶颈；

引入API 服务网关额外添加了一个需要维护的系统，增加了开发和运维的工作量，提高了系统复杂程度。

可以看到，应用API 网关需要权衡带来的收益和因此增加的复杂性，这也是我们前面说的，分布式系统是复杂性和收益的平衡，需要针对具体业务进行合理的架构设计。

### 微服务网关选型

在微服务领域，有许多开源网关实现，应用比较多的是 Spring Cloud Zuul 和 Spring Cloud Gateway。

Spring Cloud Zuul

Spring Cloud Zuul 是 Spring Cloud Netflix 项目的核心组件之一，是 Netflix 开发的一款提供动态路由、监控、弹性、安全的网关服务。

Zuul 分为 1.x 和 2.x 两个大版本，1.x 版本是基于 Servlet 构建的，采用的是阻塞和多线程方式。1.x 版本在 Spring Cloud 中做了比较好的集成，但是性能不是很理想。后来 Netflix 宣布开发 2.x 版本，目前已经更新到了 2.x 版本，不过 Spring Cloud 官方并没有集成，而是开发了自己的 Spring Cloud Gateway。

Spring Cloud Gateway

Spring Cloud Gateway 是 Spring Cloud 体系的第二代网关组件，基于 Spring 5.0 的新特性 WebFlux 进行开发，底层网络通信框架使用的是 Netty。

Spring Cloud Gateway 可以替代第一代的网关组件 Zuul。Spring Cloud Gateway 可以通过服务发现组件自动转发请求，集成了 Ribbon 做负载均衡，支持使用 Hystrix 对网关进行保护，当然也可以选择其他的容错组件，比如集成阿里巴巴开源的 Sentinel，实现更好的限流降级等功能。