分布式系统

- 为什么要进行系统拆分?
  - 分布式业务系统，把大系统拆分成多个系统，子系统之间互相调用，形成一个大系统
  - 把系统拆分成很多的服务，每个人负责一个服务，大家的代码都没有冲突，服务可以自治，自己选用什么技术都可以，每次发布如果就改动一个服务那就上线一个服务，提高开发效率
- 分布式事务
  - 分布式事务用于在分布式系统中保证不同节点之间的数据一致性。
  - 柔性事务
    - 在电商等互联网场景下，传统的事务在数据库性能和处理能力上都暴露出了瓶颈。在分布式领域基于CAP理论以及**BASE理论**，有人就提出了柔性事务的概念。
    - 基于BASE理论的设计思想，柔性事务下，在不影响系统整体可用性的情况下(Basically Available 基本可用)，允许系统存在数据不一致的中间状态(Soft State 软状态)，在经过数据同步的延时之后，最终数据能够达到一致。并不是完全放弃了ACID，而是通过**放宽一致性要求**，借助本地事务来实现最终分布式事务一致性的同时也保证系统的吞吐。
    - 柔性事务的特性
    - 这些特性在具体的方案中不一定都要满足，因为不同的方案要求不一样。
    - 可见性(对外可查询)
      - 在分布式事务执行过程中，如果某一个步骤执行出错，就需要明确的知道其他几个操作的处理情况，这就需要其他的服务都能够提供查询接口，保证可以通过查询来判断操作的处理情况。
      - 为了保证操作的可查询，需要对于每一个服务的每一次调用都有一个全局唯一的标识，可以是业务单据号（如订单号）、也可以是系统分配的操作流水号（如支付记录流水号）。除此之外，操作的时间信息也要有完整的记录。
    - 操作幂等性
      - 幂等性，其实是一个数学概念。幂等函数，或幂等方法，是指可以使用相同参数重复执行，并能获得相同结果的函数。幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。也就是说，同一个方法，使用同样的参数，调用多次产生的业务结果与调用一次产生的业务结果相同。
      - 之所以需要操作幂等性，是因为为了保证数据的最终一致性，很多事务协议都会有很多重试的操作，如果一个方法不保证幂等，那么将无法被重试。幂等操作的实现方式有多种，如在系统中缓存所有的请求与处理结果、检测到重复操作后，直接返回上一次的处理结果等。
  - 2PC方案（两阶段提交方案）- 强一致性
    - 二阶段法

    - 在分布式系统里，每个节点都可以知晓自己操作的成功或者失败，却无法知道其他节点操作的成功或失败。当一个事务跨多个节点时，为了保持事务的原子性与一致性，而引入一个协调者来统一掌控所有参与者的操作结果，并指示它们是否要把操作结果**进行真正的提交或者回滚（rollback）**。
- 算法思路
    - 二阶段提交的**算法思路**可以概括为：参与者将操作成败通知协调者（事务管理器），再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。
- 核心思想
    - **核心思想**就是对每一个事务都采用**先尝试后提交的处理方式**，处理后所有的读操作都要能获得最新的数据，因此也可以将二阶段提交看作是一个强一致性算法。
- 第一阶段：询问（准备阶段）
    - 1、协调者向所有参与者发送事务内容，询问是否可以提交事务，并等待所有参与者答复。
    - 2、各参与者执行事务操作，将undo和redo信息记入事务日志中（但不提交事务）。
    - 3、如参与者执行成功，给协调者反馈yes，即可以提交；如执行失败，给协调者反馈no，即不可提交。
- 第二阶段：执行（提交阶段）
- 情况一：当所有参与者均反馈yes，提交事务
    - 1、协调者向所有参与者发出正式提交事务的请求（即commit请求）。
    - 2、参与者执行commit请求，并释放整个事务期间占用的资源。
    - 3、各参与者向协调者反馈ack(应答)完成的消息。
    - 4、协调者收到所有参与者反馈的ack消息后，即完成事务提交。
- 情况二：当任何阶段1一个参与者反馈no，中断事务
    - 1、协调者向所有参与者发出回滚请求（即rollback请求）。
    - 2、参与者使用阶段1中的undo信息执行回滚操作，并释放整个事务期间占用的资源。
    - 3、各参与者向协调者反馈ack完成的消息。
    - 4、协调者收到所有参与者反馈的ack消息后，即完成事务中断。
- 问题
    - 1、性能问题
        - 所有参与者在事务提交阶段处于**同步阻塞**状态，占用系统资源，容易导致性能瓶颈。
    - 2、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，参与者就会一直

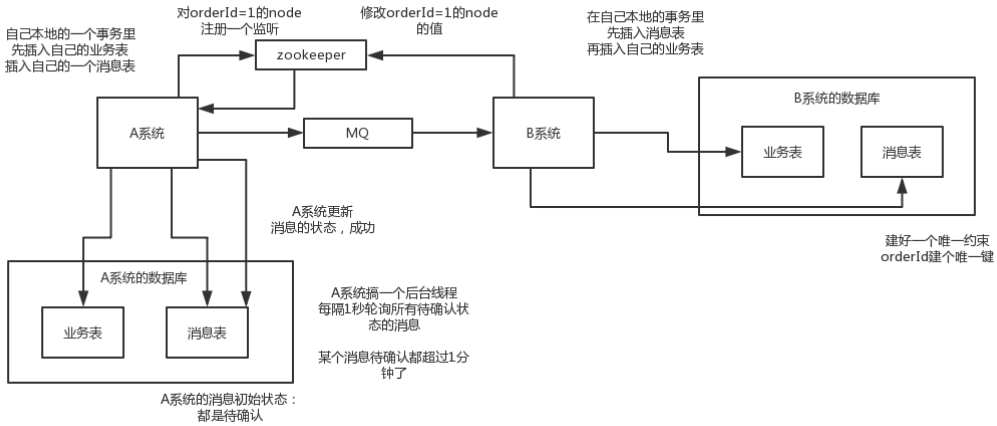
阻塞，导致数据库无法使用。
    - 如果协调者存在单点故障问题，如果协调者出现故障，参与者将一直处于锁定状态。
  - 3、数据一致性问题
    - 在阶段2中，如果发生局部网络问题，只有一部分事务参与者收到了提交消息，另一部分事务参与者没收到提交消息，那么就导致了节点之间数据的不一致。
- 3PC（三阶段提交方案）
  - 三阶段提交协议，是二阶段提交协议的改进版本，与二阶段提交不同的是，引入超时机制。同时在协调者和参与者中都引入超时机制。
  - 三阶段提交将二阶段的准备阶段拆分为2个阶段，插入了一个preCommit阶段，使得原先在二阶段提交中，参与者在准备之后，由于协调者发生崩溃或错误，而导致参与者处于无法知晓是否提交或者中止的“不确定状态”所产生的可能相当长的延时的问题得以解决。
  - 阶段1：canCommit
    - 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(参与者不执行事务操作)，否则返回no响应：
    - 1、协调者向所有参与者发出包含事务内容的canCommit请求，询问是否可以提交事务，并等待所有参与者答复。
    - 2、参与者收到canCommit请求后，如果认为可以执行事务操作，则反馈yes并进入预备状态，否则反馈no。
  - 阶段2：preCommit
    - 协调者根据阶段1 canCommit参与者的反应情况来决定是否可以基于事务的preCommit操作。根据响应情况，有以下两种可能。
  - 情况1：阶段1所有参与者均反馈yes，参与者预执行事务：
    - 1、协调者向所有参与者发出preCommit请求，进入准备阶段。
    - 2、参与者收到preCommit请求后，执行事务操作，将undo和redo信息记入事务日志中（但不提交事务）。
    - 3、各参与者向协调者反馈ack响应或no响应，并等待最终指令。
  - 情况2：阶段1任何一个参与者反馈no，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务:
    - 1、协调者向所有参与者发出abort请求。
    - 2、无论收到协调者发出的abort请求，或者在等待协调者请求过程中出现超时，参与者均会中断事务。
  - 阶段3：do Commit
    - 该阶段进行**真正的事务提交**，也可以分为以下两种情况：
  - 情况1：阶段2所有参与者均反馈ack响应，执行真正的事务提交：

        - 1、如果协调者处于工作状态，则向所有参与者发出do Commit请求。
        - 2、参与者收到do Commit请求后，会正式执行事务提交，并释放整个事务期间占用的资源。
        - 3、各参与者向协调者反馈ack完成的消息。
        - 4、协调者收到所有参与者反馈的ack消息后，即完成事务提交。
    - 阶段2任何一个参与者反馈no，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务：
        - 1、如果协调者处于工作状态，向所有参与者发出abort请求。
        - 2、参与者使用阶段1中的undo信息执行回滚操作，并释放整个事务期间占用的资源。
        - 3、各参与者向协调者反馈ack完成的消息。
        - 4、协调者收到所有参与者反馈的ack消息后，即完成事务中断。
    - 注意：进入阶段3后，无论协调者出现问题，或者协调者与参与者网络出现问题，都会导致参与者无法接收到协调者发出的do Commit请求或abort请求。此时，参与者都会在等待超时之后，继续执行事务提交。
    - 优点
        - 相比二阶段提交，三阶段贴近降低了阻塞范围，在等待超时后协调者或参与者会中断事务。避免了协调者单点问题，阶段3中协调者出现问题时，参与者会继续提交事务。
    - 缺点
        - 数据不一致问题依然存在，当在参与者收到preCommit请求后等待do commite指令时，此时如果协调者请求中断事务，而协调者无法与参与者正常通信，会导致参与者继续提交事务，造成数据不一致。
- TCC 方案 - 最终一致性
    - TCC 的全称是：Try、Confirm、Cancel
    - Confirm操作作为二阶段提交操作，执行真正的业务。
    - Cancel是预留资源的取消。
    - Try 阶段：这个阶段说的是对各个服务的资源**做检测**以及**对资源进行锁定或者预留。**
        - 完成所有业务检查(一致性)
        - 预留必须业务资源(准隔离性)
        - Try 尝试执行业务
    - Confirm 阶段：确认执行真正执行业务，不作任何业务检查，只使用Try阶段预留的业务资源
        - 根据Try阶段服务是否全部正常执行，继续执行确认操作（Confirm）或取消操作（Cancel）。
        - Confirm和Cancel操作满足幂等性，如果Confirm或Cancel操作执行失败会不断重试

- Cancel 阶段：如果任何一个服务的业务方法执行出错（即Confirm阶段），那么这里就需要进行补偿，就是执行已经执行成功的业务逻辑的**回滚操作**。预留资源的取消（把那些执行成功的回滚）
- TCC事务机制相比于上面的XA（两阶段法），解决了其几个缺点:
  - 1.解决了协调者单点，由主业务方发起并完成这个业务活动。业务活动管理器也变成多点，引入集群。
  - 2.同步阻塞:引入超时，超时后进行补偿，并且不会锁定整个资源，将资源转换为业务逻辑形式，粒度变小。
  - 3.数据一致性，有了补偿机制之后，由业务活动管理器控制一致性
- 缺点：
  - Try、Confirm和Cancel操作功能要按具体业务来实现，业务耦合度较高，提高了开发成本。这种方案用的也比较少，但是也有使用的场景。因为这个事务回滚实际上是严重**依赖于你自己写代码来回滚和补偿了，会造成补偿代码巨大**，非常之恶心。
- 比如说我们，一般来说跟**钱**相关的，跟钱打交道的，**支付**、**交易**相关的场景，我们会用 TCC，**严格保证**分布式事务要么全部成功，要么全部自动回滚，严格保证资金的正确性，保证在资金上不会出现问题。执行时间也要比较短。

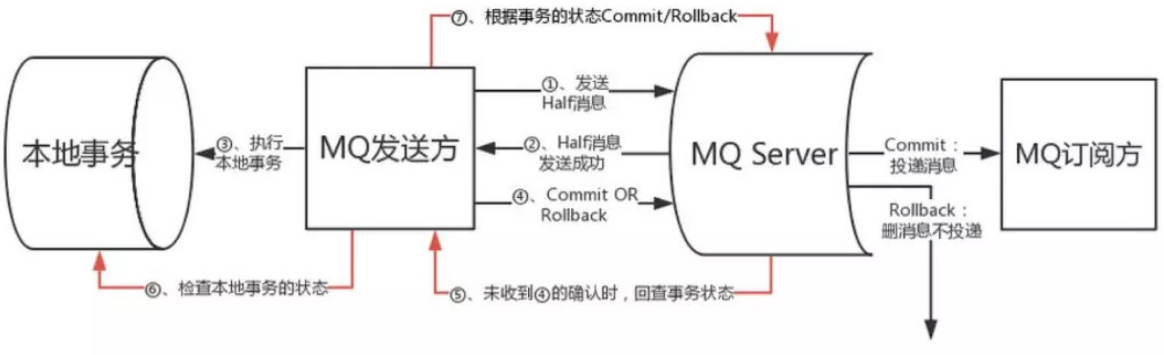
- 本地消息表 - 最终一致性
  - **核心思路是将分布式事务拆分成本地事务进行处理**
  - 方案通过在事务主动发起方额外新建事务消息表，事务发起方处理业务和记录事务消息在本地事务中完成，轮询事务消息表的数据发送事务消息，事务被动方基于消息中间件消费事务消息表中的事务。
  - 这样设计可以避免”业务处理成功 + 事务消息发送失败"，或"业务处理失败 + 事务消息发送成功"的棘手情况出现，保证2个系统事务的数据一致性。
  - 1）A系统在自己本地一个事务里操作同时，插入一条数据到消息表；（先插入业务表，在插入消息表）
  - 2）接着 A 系统将这个消息发送到 MQ 中去；
  - 3）B 系统接收到消息之后，在一个事务里，先往自己本地消息表里插入一条数据，接着执行业务操作（先插入消息表（不重复消费），在插入业务表）
  - 4）B 系统执行成功之后，就会更新自己本地消息表的状态；
  - 4.1）B系统通知A系统修改消息表的事务状态
  - 5）如果 B 系统处理失败了，那么就不会更新消息表状态，那么此时 A 系统会定时扫描自己的消息表，如果有未处理的消息，会再次发送到 MQ 中去，让 B 再次处理；
  - 6）这个方案保证了最终一致性，哪怕 B 事务失败了，但是 A 会**不断重发消息**，直到 B 那边成功为止。
  - 优点
    - 不依赖于消息中间件
    - 方案轻量，容易实现

- 缺点
  - 这个方案说实话最大的问题就在于**严重依赖于数据库的消息表**来管理事务啥的，如果是高并发场景性能会受到影响
  - 耦合性强，不可公用，要与具体的业务场景结合
- 示例图
  -



- MQ事务 - 最终一致性方案
  - 在RocketMQ中实现了分布式事务，实际上其实是对本地消息表的一个封装，将本地消息表移动到了MQ内部
  - 相比本地消息表方案，MQ事务方案优点是：
    - 消息数据独立存储，降低业务系统与消息系统之间的耦合。
    - 吞吐量由于使用本地消息表方案。
  - 缺点是：
    - 一次消息发送需要两次网络请求(half消息 + commit/rollback消息)
    - 业务处理服务需要实现消息状态回查接口
  - 正常情况——事务主动方发消息
    - 图中1、发送方向 MQ服务端(MQ Server)发送half消息。
    - 图中2、MQ Server 将消息持久化成功之后，向发送方 ACK 确认消息已经发送成功。
    - 图中3、发送方开始执行本地事务逻辑。
    - 图中7、发送方根据本地事务执行结果向 MQ Server 提交二次确认（commit 或是 rollback）。
    - MQ Server 收到 commit 状态则将半消息标记为可投递，订阅方最终将收到该消息；MQ Server 收到 rollback 状态则删除半消息，订阅方将不会接受该消息。
  - 异常情况——事务主动方消息恢复
    - 在断网或者应用重启等异常情况下，图中4提交的二次确认超时未到达 MQ Server，此时处理逻辑如下：

    - 异常情况——事务主动方消息恢复
    - 图中5、MQ Server 对该消息发起消息回查。
    - 图中6、发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
    - 图中7、发送方根据检查得到的本地事务的最终状态再次提交二次确认
    - 图中8、MQ Server基于commit / rollback 对消息进行投递或者删除
  - 图解



  - 作用：发送消息的一方我们称之为Producer，接收消费消息的一方我们称之为Consumer。如果Producer自身业务逻辑本地事务执行成功与否希望和消息的发送保持一个原子性（也就是说如果Producer本地事务执行成功，那么这笔消息就一定要被成功的发送到RocketMQ服务的指定Topic，并且Consumer一定要被消费成功；反之，如果Producer本地事务执行失败，那么这笔消息就应该被RocketMQ服务器丢弃）
- 最大努力通知方案
  - 系统 A 本地事务执行完之后，发送个消息到 MQ；
  - 这里会有个专门消费 MQ 的最大努力通知服务，这个服务会消费 MQ 然后写入数据库中记录下来，或者是放入个内存队列也可以，接着调用系统 B 的接口；
  - 要是系统 B 执行成功就 ok 了；要是系统 B 执行失败了，那么最大努力通知服务就定时尝试重新调用系统 B，反复 N 次，最后还是不行就放弃。
  - （可以一定程度上允许少数分布式事务失败，一般用在对分布式事务要求不严格的场景）
- 方案比较
  - 2PC/3PC
    - 依赖于数据库，能够很好的提供强一致性和强事务性，但相对来说延迟比较高，比较适合传统的单体应用，在同一个方法中存在跨库操作的情况，不适合高并发和高性能要求的场景。
  - TCC
    - 适用于执行时间确定且较短，实时性要求高，对数据一致性要求高，比如互联网金融企业最核心的三个服务：交易、支付、账务。
  - 本地消息表/MQ事务

        - 都适用于事务中参与方支持操作幂等，对一致性要求不高，业务上能容忍数据不一致到一个人工检查周期，事务涉及的参与方、参与环节较少，业务上有对账/校验系统兜底。
    - Saga事务
        - 由于Saga事务不能保证隔离性，需要在业务层控制并发，适合于业务场景事务并发操作同一资源较少的情况。
        - aga相比缺少预提交动作，导致补偿动作的实现比较麻烦，例如业务是发送短信，补偿动作则得再发送一次短信说明撤销，用户体验比较差。Saga事务较适用于补偿动作容易处理的场景
- 分布式锁
    - 分布式锁的实现
    - 1.Memcached分布式锁
        - 利用Memcached的add命令。此命令是原子性操作，只有在key不存在的情况下，才能add成功，也就意味着线程得到了锁。
    - 2.Redis分布式锁
        - setnx id:1:lock 随机值 过时时间
        - 此命令同样是原子性操作，只有在key不存在的情况下，才能set成功。
        - 过了30秒之后key自动删除；删除这把锁的时候执行lua脚本，找到该key的value跟自己传过去的进行比较，相同则删除（需要随机值的原因为了不删除别的节点加的锁）
        - 让获得锁的线程开启一个守护线程，用来给快要过期的锁“续航”。
    - RedLock算法
        - 1.获取当前时间戳，单位是毫秒；
        - 2.跟上面类似，轮流尝试在每个 master 节点上创建锁，过期时间较短，一般就几十毫秒；
        - 3.尝试在**大多数节点上建立一个锁**，比如 5 个节点就要求是 3 个节点 n / 2 + 1；
        - 4.客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了；
        - 5.要是锁建立失败了，那么就依次之前建立过的锁删除；
        - 6.只要别人建立了一把分布式锁，你就得不断轮询去尝试获取锁。
    - 3.Zookeeper分布式锁
        - 利用Zookeeper的顺序临时节点，来实现分布式锁和等待队列。Zookeeper设计的初衷，就是为了实现分布式锁服务的。
        - 1、首先，在Zookeeper当中创建一个持久节点ParentLock。当第一个客户端想要获得锁时，需要在ParentLock这个节点下面创建一个临时顺序节点 Lock1。
        - 2、如果再有一个客户端 Client2 前来获取锁，则在ParentLock下载再创建一个临时顺序节点Lock2。
        - 3、Client2查找ParentLock下面所有的临时顺序节点并排序，结果发现节点Lock2并不是最小的。于是，Client2向排序仅比它靠前的节点Lock1注册Watcher，用于监听

Lock1节点是否存在。这意味着Client2抢锁失败，进入了等待状态。

- 4、当任务完成时，Client1会显示调用删除节点Lock1的指令。任务执行过程中，客户端崩溃，自动删除。
- 5、由于Client2一直监听着Lock1的存在状态，当Lock1节点被删除，Client2会立刻收到通知。再次查询ParentLock下面的所有节点，确认自己创建的节点Lock2是不是目前最小的节点。如果是最小，则Client2顺理成章获得了锁。

- redis 分布式锁和 zk 分布式锁的对比
  - redis 分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能。
  - zk 分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小。
  - 另外一点就是，如果是 redis 获取锁的那个客户端 出现 bug 挂了，那么只能等待超时时间之后才能释放锁；而 zk 的话，因为创建的是临时 znode，只要客户端挂了，znode 就没了，此时就自动释放锁。
  -

- 分布式会话Session
  - session
    - session 是啥？浏览器有个 cookie，在一段时间内这个 cookie 都存在，然后每次发请求过来都带上一个特殊的 jsessionid cookie，就根据这个东西，在服务端可以维护一个对应的 session 域，里面可以放点数据。
    - 一般的话只要你没关掉浏览器，cookie 还在，那么对应的那个 session 就在，但是如果 cookie 没了，session 也就没了。常见于什么购物车之类的东西，还有登录状态保存之类的
  - tomcat + Redis
    - 使用 session 的代码，跟以前一样，还是基于 tomcat 原生的 session 支持即可，然后就是用一个叫做 Tomcat RedisSessionManager 的东西，让所有我们部署的 tomcat 都将 session 数据存储到 redis 即可。
    - 缺点：
    - 与 tomcat 容器重耦合，如果我要将 web 容器迁移成 jetty，难道还要重新把 jetty 都配置一遍？该方案会严重依赖于web容器，不好将代码移植到其他 web 容器上去
  - Spring session + Redis
    - 给 sping session 配置基于 redis 来存储 session 数据，然后配置了一个 spring session 的过滤器，这样的话，session 相关操作都会交给 spring session 来管了。接着在代码中，就用原生的 session 操作，就是直接基于 spring sesion 从 redis 中获取数据了。
    - 实现分布式的会话有很多种方式，我说的只不过是比较常见的几种方式，tomcat + redis 早期比较常用，但是会重耦合到 tomcat 中；近些年，通过 spring session 来实现。
  - Session默认是存储在单机服务器内存中的，因此在分布式环境下同一个用户发送的多次HTTP请求可能会先后落到不同的服务器上，导致后面发起的HTTP请求无法拿到之前的

HTTP请求存储在服务器中的Session数据，从而使得Session机制在分布式环境下失效。

- 1）Session集中式存储
  - Session集中式存储是指将原先存储在单机服务器内存中的Session数据集中存储起来，比如说存储在分布式缓存Redis集群中。其原理是以SessionId作为Key，序列化后的HttpSession对象作为Value存储在Redis中，然后将SessionId返回给客户端，当下一次客户端发送HTTP请求到服务器的时候，会带上这个SessionId，服务器再根据SessionId从Redis拿到相应的Session数据并反序列化成HttpSession对象。由于对HttpSession对象进行了集中存储，而不是存储在服务器本地内存，所以即使同个用户的多次HTTP请求落到不同的服务器上，也能将SessionId与相应的HttpSession对象关联起来，从而实现分布式环境下的Session共享
- 2）Session复制
  - 当客户端在某台服务器上存储了Session数据的时候，我们可以手动地将该Session信息同步到集群中的其他服务器上面去，这样一来所有服务器就都存储了所有客户端的Session信息了，因此即使同个客户端的多次HTTP请求落到不同的服务器上面去，也还是能够拿到相应的Session信息，故而也解决了Session分布式问题。实际上，一些开源的Web容器可以支持Session复制，如：Tomcat，因此实施起来难度不大。
- 3）Session Sticky
  - 从另一个角度来说，分布式环境下Session失效也可以说是因为同个客户端在多次请求之间落到不同的服务器上导致的。因此，如果能够保证同个客户端发起的HTTP请求都由相同的服务器进行处理，那么也可以避免Session失效的问题。比如说，Nginx有一种称为IP Hash的负载均衡策略，其可以实现相同IP发送的HTTP请求都路由到同一台服务器上面去，故而也能解决Session分布式问题
- 4）Cookie Bases
  - 前面所讲的方案都是将Session数据存储在服务器中，其实要实现会话保持，还可以将Session数据存储在客户端，常见的方案是存储在Cookie中，这样一来服务器就无需维护客户端的状态，即服务器“无状态化”，无状态化的好处是利于服务器水平扩展。比如：JWT就是基于Cookie实现用户认证功能的

- 负载均衡算法
  - 加权随机
    - **默认**情况下，dubbo 是 random load balance ，即**随机**调用实现负载均衡，可以对 provider 不同实例**设置不同的权重**，会按照权重来负载均衡，权重越大分配流量越高，一般就用这个默认的就可以了。
    - a b a c a b
  - 加权轮询
    - 均匀地将流量打到各个机器上去，但是如果各个机器的性能不一样，容易导致性能差的机器负载过高。所以此时需要调整权重，让性能差的机器承载权重小一些，流量少一些。
    - a a a b b c
  - 最少活跃调用数均衡算法

        - 这个就是自动感知一下，如果某个机器性能越差，那么接收的请求越少，越不活跃，此时就会给不活跃的性能差的机器更少的请求。
    - 一致性Hash均衡算法
        - 一致性 Hash 算法，相同参数的请求一定分发到一个 provider 上去，provider 挂掉的时候，会基于虚拟节点均匀分配剩余的流量，抖动不会太大。如果你需要的不是随机负载均衡，是要**一类请求都到一个节点**，那就走这个一致性 Hash 策略。
- 分布式服务接口的幂等性如何设计
    - 因为重试或者两次发起请求，两个相同的请求因为负载均衡算法分散在了这个服务部署的不同的机器上
    - 幂等性，就是说一个接口，多次发起同一个请求，你这个接口得保证结果是准确的，比如不能多扣款、不能多插入一条数据、不能将统计值多加了 1。这就是幂等性。
    - 这个不是技术问题，这个没有通用的一个方法，这个应该结合业务来保证幂等性。
    - 基于**单机**：在单机jvm内存中设置一个map或者set，记录该请求已经执行过了
    - 保证幂等性主要是三点
        - 对于每个请求必须有一个**唯一的标识**，举个栗子：订单支付请求，肯定得包含订单 id，一个订单 id 最多支付一次，对吧。
        - 每次处理完请求之后，必须有一个记录**标识这个请求处理**过了。常见的方案是在 mysql 中记录个状态啥的，比如支付之前记录一条这个订单的支付流水。
        - 每次接收请求需要进行判断，**判断之前是否处理过**。比如说，如果有一个订单已经支付了，就已经有了一条支付流水，那么如果重复发送这个请求，则此时先插入支付流水，orderId 已经存在了，唯一键约束生效，报错插入不进去的。然后你就不用再扣款了。
- 分布式服务接口请求的顺序性如何保证
    - 一般来说是不用保证顺序的。但是有时候可能确实是需要严格的顺序保证。
    - 一般来说，个人建议是，你们从业务逻辑上设计的这个系统最好是不需要这种顺序性的保证，因为一旦引入顺序性保障，比如使用**分布式锁**，会导致系统复杂度上升，而且会带来效率低下，热点数据压力过大等问题。
    - 下面我给个我们用过的方案吧，简单来说，首先你得用 dubbo 的一致性 hash 负载均衡策略，将比如某一个订单 id 对应的请求都给分发到某个机器上去，接着就是在那个机器上，因为可能还是多线程并发执行的，你可能得立即将某个订单 id 对应的请求扔一个内存队列里去，强制排队，这样来确保他们的顺序性。
    - 但是这样引发的后续问题就很多，比如说要是某个订单对应的请求特别多，造成某台机器成热点怎么办？解决这些问题又要开启后续一连串的复杂技术方案……
- 如何设计一个高并发系统
    - 可以分为一下6点
    - 1）系统拆分
        - 将一个系统拆分为多个子系统，用 dubbo 来搞。然后每个系统连一个数据库，这样本来就一个库，现在多个数据库，不也可以扛高并发么

- 2）缓存
  - 缓存，必须得用缓存。**大部分的高并发场景，都是读多写少**，那你完全可以在数据库和缓存里都写一份，然后读的时候大量走缓存不就得了。毕竟人家 redis 轻轻松松单机几万的并发。所以你可以考虑考虑你的项目里，那些承载主要请求的读场景，怎么用缓存来抗高并发。
- 3）MQ
  - 可能你还是会出现高并发写的场景，比如说一个业务操作里要频繁搞数据库几十次，增删改增删改，疯了。那高并发绝对搞挂你的系统，你要是用 redis 来承载写那肯定不行，人家是缓存，数据随时就被 LRU 了，数据格式还无比简单，没有事务支持。所以该用 mysql 还得用 mysql 啊。那你咋办？用 MQ 吧，大量的写请求灌入 MQ 里，排队慢慢玩儿，后边系统消费后慢慢写，控制在 mysql 承载范围之内。所以你得考虑考虑你的项目里，那些承载复杂写业务逻辑的场景里，如何用 MQ 来异步写，提升并发性。MQ 单机抗几万并发也是 ok 的。
- 4）分库分表
  - 分库分表，可能到了最后数据库层面还是免不了抗高并发的要求，好吧，那么就将一个数据库拆分为多个库，多个库来扛更高的并发；然后将一个表拆分为多个表，每个表的数据量保持少一点，提高 sql 跑的性能。
- 5）读写分离
  - 读写分离，这个就是说大部分时候数据库可能也是读多写少，没必要所有请求都集中在一个库上吧，可以搞个主从架构，主库写入，从库读取，搞一个读写分离。读流量太多的时候，还可以加更多的从库。
- ElasticSearch

- CAP理论
  - 一致性
    - 一致性指的是多个数据副本是否能保持一致的特性，在一致性的条件下，系统在执行数据更新操作之后能够从一致性状态转移到另一个一致性状态。
    - 对系统的一个数据更新成功之后，如果所有用户都能够读取到最新的值，该系统就被认为具有强一致性。
  - 可用性
    - 可用性指分布式系统在面对各种异常时可以提供正常服务的能力，可以用系统可用时间占总时间的比值来衡量，4 个 9 的可用性表示系统 99.99% 的时间是可用的。
    - 在可用性条件下，要求系统提供的服务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。
  - 分区容忍性
    - 网络分区指分布式系统中的节点被划分为多个区域，每个区域内部可以通信，但是区域之间无法通信。
    - 在分区容忍性条件下，分布式系统在遇到任何网络分区故障的时候，仍然需要能对外提供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。

- 权衡
  - 在分布式系统中，分区容忍性必不可少，因为需要总是假设网络是不可靠的。因此，CAP 理论实际上是要在可用性和一致性之间做权衡。
  - 可用性和一致性往往是冲突的，很难使它们同时满足。在多个节点之间进行数据同步时，
  - 为了保证一致性（CP），不能访问未同步完成的节点，也就失去了部分可用性；
  - 为了保证可用性（AP），允许读取所有节点的数据，但是数据可能不一致。

- CAP的延申：BASE理论
  - BASE 是指基本可用（Basically Available）、软状态（ Soft State）、最终一致性（ Eventual Consistency），**核心思想是即使无法做到强一致性（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。**
  - BA - Basically Available 基本可用
    - 分布式系统在出现故障时，允许损失部分可用性，即保证核心可用。
    - 这里的关键词是“部分”和“核心”，实际实践上，哪些是核心需要根据具体业务来权衡。例如登录功能相对注册功能更加核心，注册不了最多影响流失一部分用户，如果用户已经注册但无法登录，那就意味用户无法使用系统，造成的影响范围更大。
  - S - Soft State 软状态
    - 允许系统存在中间状态，而该中间状态不会影响系统整体可用性。这里的中间状态就是 CAP 理论中的数据不一致。
  - E - Eventual Consistency 最终一致性
    - 系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。
    - 这里的关键词是“一定时间” 和 “最终”，“一定时间”和数据的特性是强关联的，不同业务不同数据能够容忍的不一致时间是不同的。例如支付类业务是要求秒级别内达到一致，因为用户时时关注；用户发的最新微博，可以容忍30分钟内达到一致的状态，因为用户短时间看不到明星发的微博是无感知的。而“最终”的含义就是不管多长时间，最终还是要达到一致性的状态。
  - BASE 理论本质上是对 CAP 的延伸和补充，更具体地说，是对 CAP 中 AP 方案的一个补充：
    - AP 方案中牺牲一致性只是指发生分区故障期间，而不是永远放弃一致性。
    - 这一点其实就是 BASE 理论延伸的地方，**分区期间牺牲一致性，但分区故障恢复后，系统应该达到最终一致性。**

- Paxos
  - 用于达成共识性问题，即对多个节点产生的值，该算法能保证只选出唯一一个值。
  - 主要有三类节点：
    - 提议者（Proposer）：提议一个值；
    - 接受者（Acceptor）：对每个提议进行投票；
    - 告知者（Learner）：被告知投票的结果，不参与投票过程。

- 执行过程
  - 规定一个提议包含两个字段：[n, v]，其中 n 为序号（具有唯一性），v 为提议值。
- 1. Prepare 阶段
  - 下图演示了两个 Proposer 和三个 Acceptor 的系统中运行该算法的初始过程，每个 Proposer 都会向所有 Acceptor 发送 Prepare 请求。