什么是凤凰架构

1、提出重编程能力还是重架构的问题

问：做一个高质量的软件，应该把精力集中在提升其中每一个人员、过程、产出物的能力和质量上，还是该把更多精力放在整体流程和架构上？

答：这两者都重要。前者重术，后者重道；前者更多与编码能力相关，后者更多与软件架构相关；前者主要由开发者个体水平决定，后者主要由技术决策者水平决定。

2、提出构建一个大规模但依然可靠的软件系统是否是可行的

通过**冯诺依曼研发自复制自动机**的例子举例我们一直是在用不可靠部件构造可靠的系统。比如我们开发的每个环节都是有可能出错的，但最终设计出的软件必然是不可靠的，但事实并非如此。用冯诺依曼的自动机这个例子来讲就是说这些零部件可能会出错，某个具体的零部件可能会崩溃消亡，但在存续生命的微生态系统中一定会有其后代的出现，重新代替该零部件的作用，以维持系统的整体稳定。在这个微生态里，每一个部件都可以看作一只不死鸟，它会老去然后又能涅槃重生。

3、强调架构演变最终都是为了使我们的服务更好地死去和重生

软件架构风格演变顺序：大型机 -> 原始分布式 -> 大型单体 -> 面向服务 -> 微服务 -> 服务网格 -> 无服务

技术架构上呈现出从大到小的发展趋势。作者提出了：**相比于易于伸缩拓展应对更高的性能等新架构的优点，架构演变最重要的驱动力始终都是为了方便某个服务能够顺利地“死去”与“重生”而设计的，个体服务的生死更迭，是关系到整个系统能否可靠续存的关键因素。**

服务架构演进史

原始分布式时代

UNIX 的设计原则提出了：保持接口与实现的简单性，比系统的任何其他属性，包括准确性、一致性和完整性，都来得更加重要。

负责制定 UNIX 系统技术标准的开放**软件基金会(也叫OSF)** 邀请了各大计算机厂商一起参与共同制订了**名为“分布式运算环境”(也叫DCE)的分布式技术体系**。DCE 包含一套相对完整的分布式服务组件规范与参考实现。

OSF 严格遵循 UNIX 设计风格，有一个预设的重要原则是使分布式环境中的服务调用、资源访问、数据存储等操作尽可能透明化、简单化，使开发人员不必过于关注他们访问的方法或其他资源是位于本地还是远程。这样的主旨非常符合一贯的UNIX 设计哲学。**但是实现的目标背后包含着当时根本不可能完美解决的技术困难。** 因为DCE一旦要考虑性能上的差异就不太行了。为了让程序在运行效率上可被用户接受，开发者只能在方法本身运行时间很长，可以相对忽略远程调用成本时的情况下才能考虑分布式，如果方法本身运行时间不够长，就要人为用各种方式刻意地构造出这样的场景，譬如将几个原本毫无关系的方法打包到一个方法体内，一块进行远程调用。这种构造长耗时方法本身就与期望用分布式来突破硬件算力限制、提升性能的初衷相互矛盾。并且此时的开发人员实际上仍然必须每时每刻都意识到自己是在编写分布式程序，不可轻易踏过本地与远程的界限。这和简单透明的原则相违背。

通过这个原始分布式开发得出了一个教训：**某个功能能够进行分布式，并不意味着它就应该进行分布式，强行追求透明的分布式操作，只会自寻苦果。**

基于当时的情况摆在计算机科学面前有两条通往更大规模软件系统的道路，一条是尽快提升单机的处理能力，以避免分布式带来的种种问题；另一条路是找到更完美的解决如何构筑分布式系统的解决方案

单体系统时代

单体架构中“单体”只是表明系统中主要的过程调用都是进程内调用，不会发生进程间通信。

单体架构的系统又叫巨石系统。单体架构本身具有简单的特性，简单到在相当长的时间内，大家都已经习惯了软件架构就应该是单体这种样子，所以并没有多少人将“单体”视作一种架构来看待。

和很多书中的内容不同的是，单体其实并不是一个“反派角色”，单体并没有大家口中的那么不堪。实际上，它时运行效率最高的一种架构风格。基于**软件的性能需求超过了单机，软件开发人员规模扩大**这样的情况，才体现了单体系统的不足之处。

单体架构由于所有代码都运行在同一个进程空间之内，所有模块、方法的调用都无须考虑网络分区、对象复制这些麻烦的事和性能损失。一方面获得了进程内调用的简单、高效等好处的同时，另一方面也意味着如果任何一部分代码出现了缺陷，过度消耗了进程空间内的资源，所造成的影响也是全局性的、难以隔离的。比如内存泄漏、线程爆炸、阻塞、死循环等问题，都会影响整个程序，而不仅仅是影响某一个功能、模块本身的正常运作。

同样的，由于所有代码都共享着同一个进程空间，不能隔离，也就无法做到单独停止、更新、升级某一部分代码，所以从可维护性来说，单体系统也是不占优势的。程序升级、修改缺陷往往需要制定专门的停机更新计划，做灰度发布、A/B 测试也相对更复杂。

除了以上问题是单体架构的缺陷外，作者提出，最重要的还是：**单体系统很难兼容“Phoenix”的特性**

单体架构这种风格潜在的观念是希望系统的每一个部件，每一处代码都尽量可靠，靠不出或少出缺陷来构建可靠系统。但是单体靠高质量来保证高可靠性的思路，在小规模软件上还能运作良好，但系统规模越大，交付一个可靠的单体系统就变得越来越具有挑战性。

为了允许程序出错，为了获得隔离、自治的能力，为了可以技术异构等目标，是继为了性能与算力之后，让程序再次选择分布式的理由。在单体架构后，有一段时间是在尝试将一个大的单体系统拆分为若干个更小的、不运行在同一个进程的独立服务，这些服务拆分方法后来导致了面向服务架构(Service-Oriented Architecture)的一段兴盛期，这就是SOA 时代。

SOA时代

SOA是一次具体地、系统性地成功解决分布式服务主要问题的架构模式。

三种有代表性的SOA

- 烟囱式架构

信息烟囱又叫信息孤岛。使用这种架构的系统也被称为孤岛式信息系统或者烟囱式信息系统。它指的是一种完全不与其他相关信息系统进行互操作或者协调工作的设计模式。

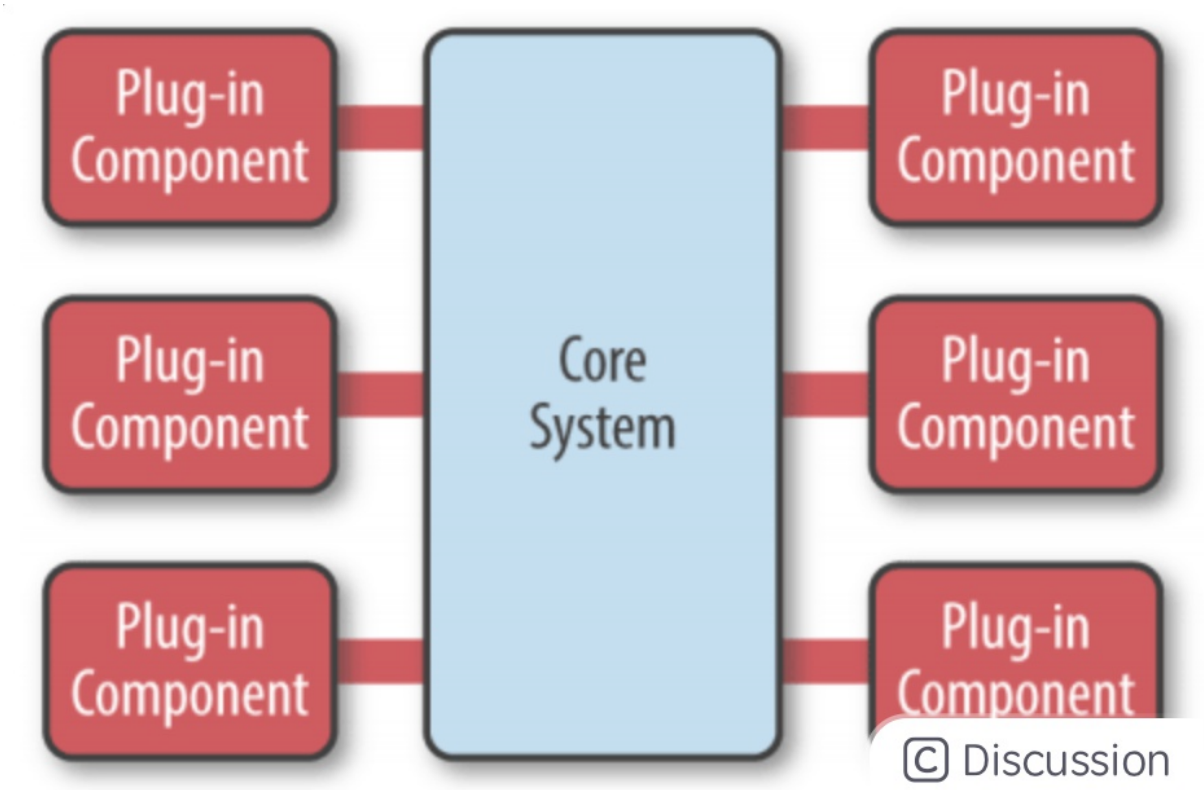
这样的系统其实并没有什么“架构设计”可言。这样完全不进行交互的模式不符合真实业务情况。

- 微内核架构

微内核架构也被称为插件式架构。微内核将主数据，连同其他可能被各子系统使用到的公共服务、数据、资源集中到一块，成为一个被所有业务系统共同依赖的核心(Kernel，也称为 Core System)，具体的业务系统以插件模块(Plug-in Modules)的形式存在，这样也可提供可扩展的、灵活的、天然隔离的功能特性。

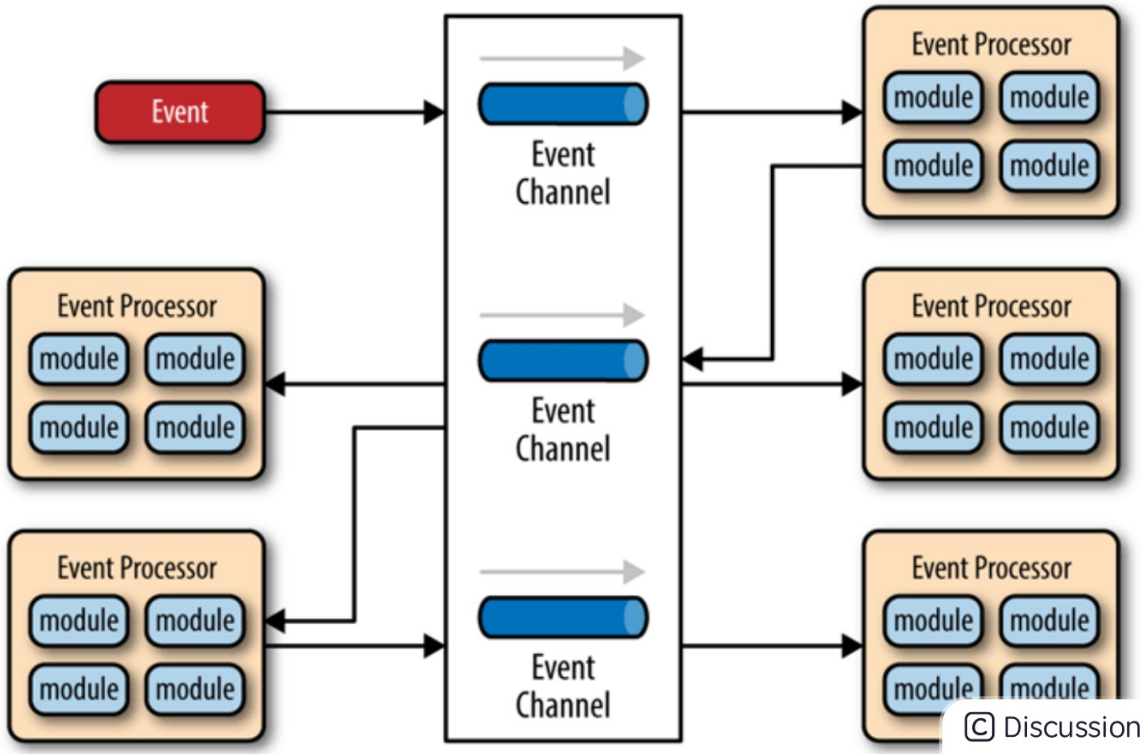
这种模式适合桌面应用程序和Web 应用程序。对于平台型应用来说，经常会加入新的功能，就很像时不时加一个新的插件模块进来所以微内核架构比较适合。微内核架构也可以嵌入到其他的架构模式之中，通过插件的方式来提供新功能的定制开发能。

微内核架构也有它的局限和使用前提，架构中这些插件可以访问内核中一些公共的资源，但不会直接交互。但是无论是企业信息系统还是互联网应用必须既能拆分出独立的系统，也能让拆分后的子系统之间顺畅地互相调用通信。



事件驱动架构

为了能让子系统互相通信，事件驱动架构的方案是在子系统之间建立一套事件队列管道，来自系统外部的消息将以事件的形式发送至管道中，各个子系统从管道里获取能够处理的事件消息，可以自己发布一些新的事件到管道队列中去，如此，每一个消息的处理者都是独立的，高度解耦的，但又能与其他处理者通过事件管道进行互动。



演化至事件驱动架构时远程服务调用迎来了 SOAP 协议的诞生

微服务时代

微服务是一种通过多个小型服务组合来构建单个应用的架构风格，这些服务围绕业务能力而非特定的技术标准来构建。各个服务可以采用不同的编程语言，不同的数据存储技术，运行在不同的进程之中。服务采取轻量级的通信机制和自动化的部署机制实现通信与运维。

“微服务”这个技术名词是由 Peter Rodgers 博士在 2005 年度的云计算博览会提出的。“Micro-Web-Service”，指的是一种专注于单一职责的、语言无关的、细粒度 Web 服务。最初的微服务可以说是 SOA 发展时催生的产物。随着时间的推进，技术的发展，微服务已经不再是维基百科定义的那样，“仅仅只是一种 SOA 的变种形式”了。

微服务真正的崛起是在 2014 年，Martin Fowler 与 James Lewis 合写的文章《Microservices: A Definition of This New Architectural Term》中 给出了现代微服务的概念：**“微服务是一种通过多个小型服务组合来构建单个应用的架构风格，这些服务围绕业务能力而非特定的技术标准来构建。各个服务可以采用不同的编程语言，不同的数据存储技术，运行在不同的进程之中。服务采取轻量级的通信机制和自动化的部署机制实现通信与运维。”**

文中列举了微服务的九个核心的业务与技术特征：

- 围绕业务能力构建
- 分散治理
- 通过服务来实现独立自治的组件
- 产品化思维
- 数据去中心化
- 强终端弱管道
- 容错性设计
- 演进式设计
- 基础设施自动化

《Microservices》文中除了定义微服务是什么，还专门申明了微服务不是什么——微服务不是 SOA 的变体或衍生品，应该明确地与 SOA 划清了界线，不再贴上任何 SOA 的标签。

微服务追求的是更加自由的架构风格，摒弃了几乎所有 SOA 里可以抛弃的约束和规定，提倡以“实践标准”代替“规范标准”。没有了统一的规范和约束，服务的注册发现、跟踪治理、负载均衡、故障隔离、认证授权、伸缩扩展、传输通信、事务处理，等等这些问题，在微服务中不再会有统一的解决方案，即使只讨论 Java 范围内会使用到的微服务，光一个服务间远程调用问题，可以列入解决方案的候选清单的就有：RMI(Sun/Oracle)、Thrift(Facebook)、Dubbo(阿里巴巴)、gRPC(Google)、Motan2(新浪)、Finagle(Twitter)、brpc(百度)、Arvo(Hadoop)、JSON-RPC、REST，等等；光一个服务发现问题，可以选择的就有：Eureka(Netflix)、Consul(HashiCorp)、Nacos(阿里巴巴)、ZooKeeper(Apache)、Etcd(CoreOS)、CoreDNS(CNCF)，等等。其他领域的情况也是与此类似，总之，完全是八仙过海，各显神通的局面。

作为一个普通的服务开发者，“螺丝钉”式的程序员，微服务架构是友善的。可是，微服务对架构者是满满的恶意，因为对架构能力要求已提升到史无前例的程度

后微服务时代

定义：从软件层面独力应对微服务架构问题，发展到软、硬一体，合力应对架构问题的时代，此即为“后微服务时代”。

当虚拟化的基础设施从单个服务的容器扩展至由多个容器构成的服务集群、通信网络和存储设施时，软件与硬件的界限便已经模糊。一旦虚拟化的硬件能够跟上软件的灵活性，那些与业务无关的技术性问题便有可能从软件层面剥离，悄无声息地解决于硬件基础设施之内，让软件得以只专注业务，真正“围绕业务能力构建”团队与产品。

Kubernetes 成为容器战争胜利者标志着后微服务时代的开端，但 Kubernetes 仍然没有能够完美解决全部的分布式问题——“不完美”的意思是，仅从功能上看，单纯的 Kubernetes 反而不如之前的 Spring Cloud 方案。这是因为有一些问题处于应用系统与基础设施的边缘，使得完全在基础设施层面中确实很难精细化地处理。

举个例子，微服务 A 调用了微服务 B 的两个服务，称为 B1和 B2，假设 B1表现正常但 B2出现了持续的 500 错，那在达到一定阈值之后就应该对 B2进行熔断，以避免产生雪崩效应。如果仅在基础设施层面来处理，这会遇到一个两难问题，切断 A 到 B 的网络通路则会影响到 B1的正常调用，不切断的话则持续受 B2的错误影响。

以上问题在通过 Spring Cloud 这类应用代码实现的微服务是可以处理和解决的，只受限于开发人员的想象力与技术能力，但基础设施是针对整个容器来管理的，粒度相对粗旷，只能到容器层面，对单个远程服务就很难有效管控。类似的情况不仅仅在断路器上出现，服务的监控、认证、授权、安全、负载均衡等都有可能面临细化管理的需求，譬如服务调用时的负载均衡，往往需要根据流量特征，调整负载均衡的层次、

算法，等等，而 DNS 尽管能实现一定程度的负载均衡，但通常并不能满足这些额外的需求。

为了解决这一类问题，虚拟化的基础设施很快完成了第二次进化，引入了“服务网格”(Service Mesh)的“边车代理模式”(Sidecar Proxy)，所谓的“边车”是由系统自动在服务容器(通常是指 Kubernetes 的 Pod)中注入一个通信代理服务器，以类似网络安全里中间人攻击的方式进行流量劫持，在应用毫无感知的情况下，悄然接管应用所有对外通信。这个代理除了实现正常的服务间通信外(称为数据平面通信)，还接收来自控制器的指令(称为控制平面通信)，根据控制平面中的配置，对数据平面通信的内容进行分析处理，以实现熔断、认证、度量、监控、负载均衡等各种附加功能。这样便实现了既不需要在应用层面加入额外的处理代码，也提供了几乎不亚于程序代码的精细管理能力。

很难从概念上判定清楚一个与应用系统运行于同一资源容器之内的代理服务到底应该算软件还是算基础设施，但它对应用是透明的，不需要改动任何软件代码就可以实现服务治理，这便足够了。服务网格在 2018 年才火起来，今天它仍然是个新潮的概念，仍然未完全成熟，甚至连 Kubernetes 也还算是个新生事物。但作者提出，未来 Kubernetes 将会成为服务器端标准的运行环境，如同现在 Linux 系统；服务网格将会成为微服务之间通信交互的主流模式，把“选择什么通信协议”、“怎样调度流量”、“如何认证授权”之类的技术问题隔离于程序代码之外，取代今天 Spring Cloud 全家桶中大部分组件的功能，微服务只需要考虑业务本身的逻辑，这才是最理想的解决方案。

无服务时代

如果说微服务架构是分布式系统这条路的极致，那无服务架构，也许就是“不分布式”的云端系统这条路的起点。

虽然发展到了微服务架构解决了单台机器的性能无法满足系统的运行需要的问题，但是获得更好性能的需求在架构设计中依然占很大的比重。对软件研发而言，不去做分布式无疑才是最简单的，如果单台服务器的性能可以是无限的，那架构演进一定不是像今天这个样子。

绝对意义上的无限性能必然是不存在的，但在云计算落地已有十年时间的今日，相对意义的无限性能已经成为了现实。2012 年，Iron.io 公司率先提出了“无服务”的概念，2014 年开始，亚马逊发布了名为 Lambda 的商业化无服务应用，并在后续的几年里逐步得到开发者认可，发展成目前世界上最大的无服务的运行平台；到了 2018 年，中国的阿里云、腾讯云等厂商也开始跟进，发布了旗下的无服务的产品，“无服务”已成了近期技术圈里的“新网红”之一。

无服务现在还没有一个特别权威的“官方”定义，但它的概念并没有前面各种架构那么复杂，本来无服务也是以“简单”为主要卖点的，它只涉及两块内容：后端设施和函数。

- **后端设施**是指数据库、消息队列、日志、存储，等等这一类用于支撑业务逻辑运行，但本身无业务含义的技术组件，这些后端设施都运行在云中，无服务中称其为“后端即服务”(Backend as a Service，BaaS)。
- **函数**是指业务逻辑代码，这里函数的概念与粒度，都已经很接近于程序编码角度的函数了，其区别是无服务中的函数运行在云端，不必考虑算力问题，不必考虑容量规划，无服务中称其为“函数即服务”(Function as a Service，FaaS)。

无服务的愿景是让开发者只需要纯粹地关注业务，不需要考虑技术组件，后端的技术组件是现成的，可以直接取用，没有采购、版权和选型的烦恼；不需要考虑如何部署，部署过程完全是托管到云端的，工作由云端自动完成；不需要考虑算力，有整个数据中心支撑，算力可以认为是无限的；也不需要操心运维，维护系统持续平稳运行是云计算服务商的责任而不再是开发者的责任。

作者认为无服务很难成为一种普适性的架构模式，因为无服务不适配于所有的应用。对于那些信息管理系统、网络游戏等应用，所有具有业务逻辑复杂，依赖服务端状态，响应速度要求较高，需要长链接等这些特征的应用，至少目前是相对并不适合的。因为无服务天生“无限算力”的假设决定了它必须要按使用量计费

以控制消耗算力的规模，所以函数不会一直以活动状态常驻服务器，请求到了才会开始运行，这导致了函数不便依赖服务端状态，也导致了函数会有冷启动时间，响应的性能不可能太好

作者认为软件开发的未来不会只存在某一种“最先进的”架构风格，多种具针对性的架构风格同时并存，是软件产业更有生命力的形态。笔者同样相信软件开发的未来，多种架构风格将会融合互补，“分布式”与“不分布式”的边界将逐渐模糊，两条路线在云端的数据中心中交汇。

架构师的视角

访问远程服务

远程服务调用

进程间通信

RPC 出现的最初目的，就是为了让计算机能够跟调用本地方法一样去调用远程方法。

进程间通信的方式有

- 管道，又叫具名管道

管道类似于两个进程间的桥梁，可通过管道在进程间传递少量的字符流或字节流。普通管道只用于有亲缘关系进程(由一个进程启动的另外一个进程)间的通信，具名管道摆脱了普通管道没有名字的限制，除具有管道所有的功能外，它还允许无亲缘关系进程间的通信。管道典型的应用就是命令行中的|操作符，比如：

```
ps -ef | grep java
```

ps与grep都有独立的进程，以上命令就通过管道操作符|将ps命令的标准输出连接到grep命令的标准输入上。

- 信号

信号用于通知目标进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程自身。信号的典型应用是kill命令，比如：

```
kill -9 pid
```

以上就是由 Shell 进程向指定 PID 的进程发送 SIGKILL 信号。

- 信号量

信号量用于两个进程之间同步协作手段，它相当于操作系统提供的一个特殊变量，程序可以在上面进行wait()和notify()操作。

- 消息队列

以上三种方式只适合传递传递少量信息，消息队列用于进程间数据量较多的通信。进程可以向队列添加消息，被赋予读权限的进程则可以从队列消费消息。消息队列克服了信号承载信息量少，管道只能用于无格式字节流以及缓冲区大小受限等缺点，但实时性相对受限。

- 共享内存

允许多个进程访问同一块公共的内存空间，这是效率最高的进程间通信形式。原本每个进程的内存地址空间都是相互隔离的，但操作系统提供了让进程主动创建、映射、分离、控制某一块内存的程序接口。当一块内存被多进程共享时，各个进程往往会与其它通信机制，譬如信号量结合使用，来达到进程间同步及互斥的协调操作。

- 套接字接口

以上两种方式只适合单机多进程间的通信，套接字接口是更为普适的进程间通信机制，可用于不同机器之间的进程通信。套接字(Socket)起初是由 UNIX 系统的 BSD 分支开发出来的，现在已经移植到所有主流的操作系统上。出于效率考虑，当仅限于本机进程间通信时，套接字接口是被优化过的，不会经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等操作，只是简单地将应用层数据从一个进程拷贝到另一个进程，这种进程间通信方式有个专名的名称：UNIX Domain Socket，又叫做 IPC Socket

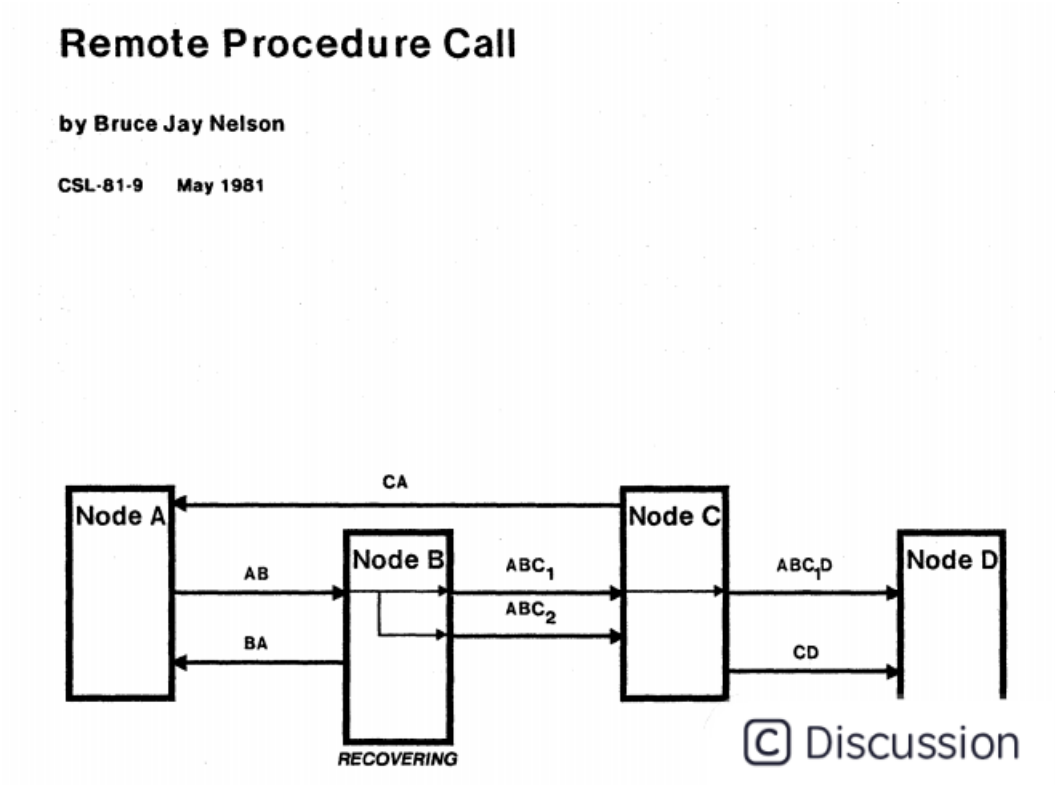
通信的成本

因为Socket是网络栈的统一接口，所以基于套接字接口的通信方式不仅适用于本地相同机器的不同进程间通信，也能支持基于网络的跨机器的进程间通信。比如 Linux 系统的图形化界面中，X Window 服务器和 GUI 程序之间的交互就是由这套机制来实现。由于 Socket 是各个操作系统都有提供的标准接口，所以可以把远程方法调用的通信细节隐藏在操作系统底层，从应用层面上看来可以做到远程调用与本地的进程间通信在编码上完全一致。这种透明的调用形式却造成了程序员误以为**通信是无成本的假象**，因而被滥用以致于显著降低了分布式系统的性能。1987 年，在“透明的 RPC 调用”一度成为主流范式的时候，Andrew Tanenbaum 教授曾发表了论文对这种透明的 RPC 范式提出了一系列质问，论文的中心观点是，本地调用与远程调用当做一样处理，这是犯了方向性的错误，把系统间的调用做成透明，反而会增加程序员工作的复杂度。此后几年，关于 RPC 应该如何发展、如何实现的论文层出不穷。最终，到 1994 年至 1997 年间，一众大佬们共同总结了**通过网络进行分布式运算的八宗罪**

1. The network is reliable —— 网络是可靠的。
2. Latency is zero —— 延迟是不存在的。
3. Bandwidth is infinite —— 带宽是无限的。
4. The network is secure —— 网络是安全的。
5. Topology doesn't change —— 拓扑结构是一成不变的。
6. There is one administrator —— 总会有一个管理员。
7. Transport cost is zero —— 不必考虑传输成本。
8. The network is homogeneous —— 网络是同质化的。

以上这八条反话被认为是程序员在网络编程中经常被忽略的八大问题，潜台词就是如果远程服务调用要弄透明化的话，就必须为这些罪过埋单，这算是给 RPC 是否能等同于 IPC 来实现**暂时**定下了一个具有公信力的结论。至此，RPC 应该是一种高层次的或者说语言层次的特征，而不是像 IPC 那样，是低层次的或者说系统层次的特征成为工业界、学术界的主流观点。

远程服务调用的定义：远程服务调用是指位于互不重合的内存地址空间中的两个程序，在语言层面上，以同步的方式使用带宽有限的信道来传输程序控制信息。



三个基本问题

从20 世纪 80 年代中后期开始直至接下来几十年来所有流行过的 RPC 协议，都不外乎变着花样使用各种手段来解决以下三个基本问题：

- 如何表示数据

这里数据包括了传递给方法的参数，以及方法执行后的返回值。

- 如何传递数据

如何通过网络，在两个服务的 Endpoint 之间相互操作、交换数据。

- 如何确定方法

“如何表示同一个方法”，“如何找到对应的方法”还是得弄个跨语言的统一的标准才行。

以上 RPC 中的三个基本问题，全部都可以在本地方法调用过程中找到相对应的操作。RPC 的想法始于本地方法调用，尽管早已不再追求实现成与本地方法调用完全一致，但其设计思路仍然带有本地方法调用的深刻烙印。

统一的RPC

1、面向透明的、简单的 RPC 协议，如 DCE/RPC、DCOM、Java RMI，**要么依赖于操作系统，要么依赖于特定语言，总有一些先天约束**；

2、面向通用的、普适的 RPC 协议；如 CORBA，就无法逃过使用**复杂性的困扰**，CORBA 烦琐的 OMG IDL、ORB 都是很好的佐证；

3、通过技术手段来屏蔽复杂性的 RPC 协议，如 Web Service，又不免受到**性能问题**的束缚。

对于RPC协议，简单、普适、高性能这三点，似乎真的难以同时满足。

分裂的RPC

由于一直没有一个同时满足以上三点的完美 RPC 协议出现，所以RPC这个领域里逐渐进入了百家争鸣并一直延续至今。现在，任何一款具有生命力的 RPC 框架，都不再去追求大而全的完美，而是有自己的针对性特点作为主要的发展方向，举例分析如下。

- 朝着**面向对象**发展

不满足于 RPC 将面向过程的编码方式带到分布式，希望在分布式系统中也能够进行跨进程的面向对象编程，代表为 RMI、.NET Remoting，这个分支也有个别名叫做分布式对象。

- 朝着**性能**发展

代表为 gRPC 和 Thrift。决定 RPC 性能的主要就两个因素：**序列化效率和信息密度**。序列化输出结果的容量越小，速度越快，效率自然越高；信息密度则取决于协议中有效荷载所占总传输数据的比例大小，使用传输协议的层次越高，信息密度就越低。gRPC 和 Thrift 都有自己优秀的专有序列化器，而传输协议方面，gRPC 是基于 HTTP/2 的，支持多路复用和 Header 压缩，Thrift 则直接基于传输层的 TCP 协议来实现，省去了额外应用层协议的开销。

- 朝着**简化**发展

代表为 JSON-RPC，说要选功能最强、速度最快的 RPC 可能会很有争议，但选功能弱的、速度慢的，JSON-RPC 肯定会候选人中之一。牺牲了功能和效率，换来的是协议的简单轻便，接口与格式都更为通用，尤其适合用于 Web 浏览器这类一般不会有额外协议支持、额外客户端支持的应用场合。

- .....

经历了 多种RPC 框架百家争鸣，大家都认识到了不同的 RPC 框架所提供的特性或多或少是有矛盾的，很难有某一种框架能十全十美。因为必须有取舍，所以导致不断有新的 RPC 轮子出现，决定了选择框架时在获得一些利益的同时，要付出另外一些代价。

到了最近几年，RPC 框架不仅仅负责调用远程服务，还管理远程服务，不再追求独立地解决 RPC 的全部三个问题(表示数据、传递数据、表示方法)，而是将一部分功能设计成扩展点，让用户自己去选择。框架聚焦于提供核心的、更高层次的能力，比如提供负载均衡、服务注册、可观察性等方面的支持。这一类框架的代表有 Facebook 的 Thrift 与阿里的 Dubbo。

REST 设计风格

很多人会拿 REST 与 RPC 互相比较，但是REST 和RPC本质上并不是同一类型的东西，无论是在思想上、概念上，还是使用范围上，与 RPC 都只能算有一些相似。REST 只能说是风格而不是规范、协议，REST 与 RPC 作为主流的两种远程调用方式，在使用上是确有重合的。

RESTful的系统

一套理想的、完全满足 REST 风格的系统应该满足以下六大原则。

- **服务端与客户端分离**

- **无状态**

无状态是 REST 的一条核心原则。REST 希望服务器不要去负责维护状态，每一次从客户端发送的请求中，应包括所有的必要的上下文信息，会话信息也由客户端负责保存维护，服务端依据客户端传递的状态来执行业务处理逻辑，驱动整个应用的状态变迁。

- **可缓存**
- **分层系统**

这里所指的并不是表示层、服务层、持久层这种意义上的分层。而是指客户端一般不需要知道是否直接连接到了最终的服务器，抑或连接到路径上的中间服务器。中间服务器可以通过负载均衡和共享缓存的机制提高系统的可扩展性，这样也便于缓存、伸缩和安全策略的部署。

- **统一接口**

这是 REST 的另一条核心原则，REST 希望开发者面向资源编程，希望软件系统设计的重点放在抽象系统该有哪些资源上，而不是抽象系统该有哪些行为(服务)上。

- **按需代码**

这是一条可选原则。它是指任何按照客户端的请求，将可执行的软件程序从服务器发送到客户端的技术，按需代码赋予了客户端无需事先知道所有来自服务端的信息应该如何处理、如何运行的宽容度。

RMM的成熟度

《RESTful Web APIs》和《RESTful Web Services》的作者 Leonard Richardson 提出过一个衡量“服务有多么 REST”的 Richardson 成熟度模型。Richardson 将服务接口“REST 的程度”从低到高，分为 1 至 4 级：

1. The Swamp of Plain Old XML：完全不 REST。
2. Resources：开始引入资源的概念。
3. HTTP Verbs：引入统一接口，映射到 HTTP 协议的方法上。
4. Hypermedia Controls：超文本驱动。

事务处理

事务处理存在的意义是为了保证系统中所有的数据都是符合期望的，且相互关联的数据之间不会产生矛盾，即数据状态的一致性(Consistency)。

事务的三个重点方面：

- **原子性**(Atomic)

  在同一项业务处理过程中，事务保证了对多个数据的修改，要么同时成功，要么同时被撤销。

- **隔离性**(Isolation)

  在不同的业务处理过程中，事务保证了各自业务正在读、写的数据互相独立，不会彼此影响。

- **持久性**(Durability)

  事务应当保证所有成功被提交的数据修改都能够正确地被持久化，不丢失数据。

四种属性即事务的ACID特性

事务的概念最初起源于数据库系统但已经有所延伸，而不再局限于数据库本身了。所有需要保证数据一致性的应用场景，都有可能会用到事务。

- 当一个服务只使用一个数据源时，通过 A、I、D 来获得一致性是最经典的做法，也是相对容易的。此

时，多个并发事务所读写的数据能够被数据源感知是否存在冲突，并发事务的读写在时间线上的最终顺序是由数据源来确定的，这种事务间一致性被称为“内部一致性”。
- 当一个服务使用到多个不同的数据源，甚至多个不同服务同时涉及多个不同的数据源时，问题就变得相对困难了许多。此时，并发执行甚至是先后执行的多个事务，在时间线上的顺序并不由任何一个数据源来决定，这种涉及多个数据源的事务间一致性被称为“外部一致性”。

外部一致性问题通常很难再使用 A、I、D 来解决，因为这样需要付出很大乃至不切实际的代价；但是外部一致性又是分布式系统中必然会遇到且必须要解决的问题，为此将一致性从“是或否”的二元属性转变为可以按不同强度分开讨论的多元属性，在确保代价可承受的前提下获得强度尽可能高的一致性保障，也正因如此，事务处理才从一个具体操作上的“编程问题”上升成一个需要全局权衡的“架构问题”。

本地事务

本地事务是最基础的一种事务解决方案，只适用于单个服务使用单个数据源的场景。从应用角度看，它是直接依赖于数据源本身提供的事务能力来工作的，在程序代码层面，最多只能对事务接口做一层标准化的包装(如 JDBC 接口)，并不能深入参与到事务的运作过程当中，事务的开启、终止、提交、回滚、嵌套、设置隔离级别，乃至与应用代码贴近的事务传播方式，全部都要依赖底层数据源的支持才能工作。

举个例子，假设你的代码调用了 JDBC 中的`Transaction::rollback()`方法，方法的成功执行也并不一定代表事务就已经被成功回滚，如果数据表采用的引擎是MyISAM，那`rollback()`方法便是一项没有意义的空操作。因此，我们要想深入地讨论本地事务，便不得不越过应用代码的层次，去了解一些数据库本身的事务实现原理，弄明白传统数据库管理系统是如何通过 ACID 来实现事务的。

实现原子性和持久性

Commit Logging

原子性和持久性在事务里是密切相关的两个属性，原子性保证了事务的多个操作要么都生效要么都不生效，不会存在中间状态；持久性保证了一旦事务生效，就不会再因为任何原因而导致其修改的内容被撤销或丢失。实现原子性和持久性的最大困难是“写入磁盘”这个操作并不是原子的，不仅有“写入”与“未写入”状态，还客观地存在着“正在写”的中间状态。正因为写入中间状态与崩溃都不可能消除，所以如果不做额外保障措施的话，将内存中的数据写入磁盘，并不能保证原子性与持久性。

比如购买图书的场景，在用户账户中减去货款、在商家账户中增加货款、在商品仓库中标记一本书为配送状态。由于写入存在中间状态，所以可能发生以下情形。

- **未提交事务，写入后崩溃**：程序还没修改完三个数据，但数据库已经将其中一个或两个数据的变动写入磁盘，此时出现崩溃，一旦重启之后，数据库必须要有办法得知崩溃前发生过一次不完整的购物操作，将已经修改过的数据从磁盘中恢复成没有改过的样子，以保证原子性。
- **已提交事务，写入前崩溃**：程序已经修改完三个数据，但数据库还未将全部三个数据的变动都写入到磁盘，此时出现崩溃，一旦重启之后，数据库必须要有办法得知崩溃前发生过一次完整的购物操作，将还没来得及写入磁盘的那部分数据重新写入，以保证持久性。

由于写入中间状态与崩溃都是无法避免的，为了保证原子性和持久性，就只能在崩溃后采取恢复的补救措施，这种数据恢复操作被称为“崩溃恢复”。为了能够顺利地完成崩溃恢复，在磁盘中写入数据就不能像程序修改内存中变量值那样，直接改变某表某行某列的某个值，而是必须将修改数据这个操作所需的全部信息，包括修改什么数据、数据物理上位于哪个内存页和磁盘块中、从什么值改成什么值，等等，以日志的形式——即仅进行顺序追加的文件写入的形式先记录到磁盘中。只有在日志记录全部都安全落盘，数据库在日志中看到代表事务成功提交的“提交记录”后，才会根据日志上的信息对真正的数据进行修改，修改完成后，再在日志中加入一条“结束记录”表示事务已完成持久化，这种事务实现方法被称为“Commit Logging”(提交日志)。

Commit Logging 保障数据持久性，日志一旦成功写入 Commit Record，那整个事务就是成功的，即使真正修改数据时崩溃了，重启后根据已经写入磁盘的日志信息恢复现场、继续修改数据即可，这保证了持久性；其次，如果日志没有成功写入 Commit Record 就发生崩溃，那整个事务就是失败的，系统重启后会看到一部分没有 Commit Record 的日志，那将这部分日志标记为回滚状态即可，整个事务就像完全没好有发生过一样，这保证了原子性。

Write-Ahead Logging

Commit Logging 存在一个大缺点，就是所有对数据的真实修改都必须发生在事务提交以后，无论有何种理由，都不允许在事务提交之前就修改磁盘上的数据，对提升数据库的性能十分不利。

为了解决这个问题，ARIES 提出了“Write-Ahead Logging”的日志改进方案，所谓“提前写入”(Write-Ahead)，就是允许在事务提交之前，提前写入变动数据的意思。

Write-Ahead Logging 先将何时写入变动数据，按照事务提交时点为界，划分为 FORCE 和 STEAL 两类情况。

- **FORCE**：当事务提交后，要求变动数据必须同时完成写入则称为 FORCE，如果不强制变动数据必须同时完成写入则称为 NO-FORCE。现实中绝大多数数据库采用的都是 NO-FORCE 策略，因为只要有了日志，变动数据随时可以持久化，从优化磁盘 I/O 性能考虑，没有必要强制数据写入立即进行。
- **STEAL**：在事务提交前，允许变动数据提前写入则称为 STEAL，不允许则称为 NO-STEAL。从优化磁盘 I/O 性能考虑，允许数据提前写入，有利于利用空闲 I/O 资源，也有利于节省数据库缓存区的内存。

Commit Logging 允许 NO-FORCE，但不允许 STEAL。因为假如事务提交前就有部分变动数据写入磁盘，那一旦事务要回滚，或者发生了崩溃，这些提前写入的变动数据就都成了错误。

Write-Ahead Logging 允许 NO-FORCE，也允许 STEAL，它给出的解决办法是增加了另一种被称为 Undo Log 的日志类型，当变动数据写入磁盘前，必须先记录 Undo Log，注明修改了哪个位置的数据、从什么值改成什么值，等等。以便在事务回滚或者崩溃恢复时根据 Undo Log 对提前写入的数据变动进行擦除。Undo Log 现在一般被翻译为“**回滚日志**”，此前记录的用于崩溃恢复时重演数据变动的日志就相应被命名为 Redo Log，一般翻译为“**重做日志**”。由于 Undo Log 的加入，Write-Ahead Logging 在崩溃恢复时会执行以下三个阶段的操作。

- **分析阶段**

  该阶段从最后一次检查点开始扫描日志，找出所有没有 End Record 的事务，组成待恢复的事务集合，这个集合至少会包括 Transaction Table 和 Dirty Page Table 两个组成部分。

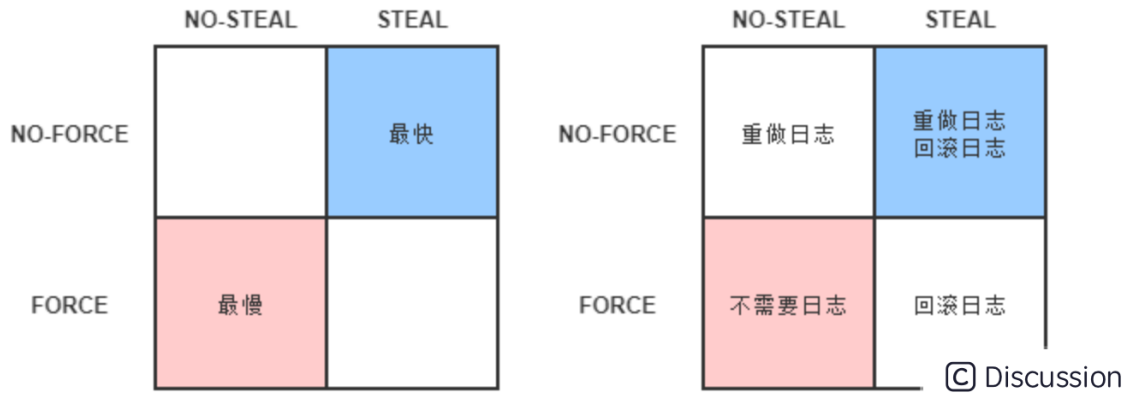
- **重做阶段**

  该阶段依据分析阶段中产生的待恢复的事务集合来重演历史，具体操作为：找出所有包含 Commit Record 的日志，将这些日志修改的数据写入磁盘，写入完成后在日志中增加一条 End Record，然后移除出待恢复事务集合。

- **回滚阶段**

  该阶段处理经过分析、重做阶段后剩余的恢复事务集合，此时剩下的都是需要回滚的事务，它们被称为 Loser，根据 Undo Log 中的信息，将已经提前写入磁盘的信息重新改写回去，以达到回滚这些 Loser 事务的目的。

重做阶段和回滚阶段的操作都应该设计为幂等的。



实现隔离性

隔离性保证了每个事务各自读、写的数据互相独立，不会彼此影响。如果没有并发，所有事务全都是串行的，那就不需要任何隔离，如果有并发，就需要加锁同步。

数据库均提供了以下三种锁

- **写锁**

  也叫作排他锁，如果数据有加写锁，就只有持有写锁的事务才能对数据进行写入操作，数据加持着写锁时，其他事务不能写入数据，也不能施加读锁。

- **读锁**

  也叫作共享锁，多个事务可以对同一个数据添加多个读锁，数据被加上读锁后就不能再被加上写锁，所以其他事务不能对该数据进行写入，但仍然可以读取。对于持有读锁的事务，如果该数据只有它自己一个事务加了读锁，允许直接将其升级为写锁，然后写入数据。

- **范围锁**

  对于某个范围直接加排他锁，在这个范围内的数据不能被写入。

事务的隔离级别

- 可串行化

串行化访问提供了强度最高的隔离性。不考虑性能优化的话，对事务所有读、写的数据全都加上读锁、写锁和范围锁即可做到`可串行化`

- 可重复读

`可重复读`对事务所涉及的数据加读锁和写锁，且一直持有至事务结束，但不再加范围锁。相比于可串行化可能出现幻读问题(指在事务执行过程中，两个完全相同的范围查询得到了不同的结果集)

- 读已提交

`读已提交`对事务涉及的数据加的写锁会一直持续到事务结束，但加的读锁在查询操作完成后就马上会释放。相比于可重复度多了不可重复读的问题(在事务执行过程中，对同一行数据的两次查询得到了不同的结果)

- 读未提交

读未提交就是“完全不隔离”，读、写锁都不加。`读未提交`会有脏读问题，但不会有脏写问题

幻读、不可重复读、脏读等问题都是由于一个事务在读数据过程中，受另外一个写数据的事务影响而破坏了隔离性。针对这种“一个事务读+另一个事务写”的隔离问题，有一种叫做多版本并发控制”(Multi-Version Concurrency Control，MVCC)的无锁优化方案被主流的数据库广泛采用。

MVCC 是一种读取优化策略，它的“无锁”是特指读取时不需要加锁。**MVCC 的基本思路是对数据库的任何修改都不会直接覆盖之前的数据，而是产生一个新版副本与老版本共存，以此达到读取时可以完全不加锁的目的。** “版本”是个关键词，可以理解为数据库中每一行记录都存在两个看不见的字段：CREATE*VERSION 和 DELETE*VERSION，这两个字段记录的值都是事务 ID，事务 ID 是一个全局严格递增的数值，然后根据以下规则写入数据。

- 插入数据时：CREATE*VERSION 记录插入数据的事务 ID，DELETE*VERSION 为空。
- 删除数据时：DELETE*VERSION 记录删除数据的事务 ID，CREATE*VERSION 为空。
- 修改数据时：将修改数据视为“删除旧数据，插入新数据”的组合，即先将原有数据复制一份，原有数据的 DELETE*VERSION 记录修改数据的事务 ID，CREATE*VERSION 为空。复制出来的新数据的 CREATE*VERSION 记录修改数据的事务 ID，DELETE*VERSION 为空。

此时，如有另外一个事务要读取这些发生了变化的数据，将根据隔离级别来决定到底应该读取哪个版本的数据。

- 隔离级别是`可重复读`：总是读取 CREATE_VERSION 小于或等于当前事务 ID 的记录，在这个前提下，如果数据仍有多个版本，则取最新(事务 ID 最大)的。
- 隔离级别是`读已提交`：总是取最新的版本即可，即最近被 Commit 的那个版本的数据记录。

另外两个隔离级别都没有必要用到 MVCC，因为`读未提交`直接修改原始数据即可，其他事务查看数据的时候立刻可以看到，根本无须版本字段。`可串行化`本来的语义就是要阻塞其他事务的读取操作，而 MVCC 是做读取时无锁优化的。

MVCC 是只针对“读+写”场景的优化，如果是两个事务同时修改数据，即“写+写”的情况，那就没有多少优化的空间了，此时加锁几乎是唯一可行的解决方案，唯一需要讨论的就是加锁的策略采取乐观锁还是悲观锁。相对地，乐观锁策略的思路被称为乐观并发控制，没有必要迷信什么乐观锁要比悲观锁更快的说法，这纯粹看竞争的剧烈程度，如果竞争剧烈的话，乐观锁反而更慢。

全局事务

为了解决分布式事务的一致性问题，X/Open组织在1991年提出了一套叫XA的(eXtended Architecture 的缩写)处理事务架构，其核心内容是定义了全局的事务管理器和局部的资源管理器之间的通信接口。

XA 接口是双向的，能在一个事务管理器和多个资源管理器之间形成通信桥梁，通过协调多个数据源的一致动作，实现全局事务的统一提交或者统一回滚。基于 XA 模式在 Java 语言中的实现了全局事务处理的标准，这也就是我们现在所熟知的 JTA。

JTA 最主要的两个接口是：

- 事务管理器的接口：`javax.transaction.TransactionManager`。这套接口是给 Java EE 服务器提供容器事务(由容器自动负责事务管理)使用的，还提供了另外一套`javax.transaction.UserTransaction`接口，用于通过程序代码手动开启、提交和回滚事务。
- 满足 XA 规范的资源定义接口：`javax.transaction.xa.XAResource`，任何资源(JDBC、JMS 等等)如果想要支持 JTA，只要实现 XAResource 接口中的方法即可。

XA 将事务提交拆分成为两阶段过程：

- **准备阶段**

又叫作投票阶段，在这个阶段，协调者询问事务的所有参与者是否准备好提交，参与者如果已经准备好提交则回复 Prepared，否则回复 Non-Prepared。准备操作是在重做日志中记录全部事务提交操作所要做的内容，它与本地事务中真正提交的区别只是暂不写入最后一条 Commit Record 而已，这意味着在做完数据持久化后仍继续持有锁，维持数据对其他非事务内观察者的隔离状态。

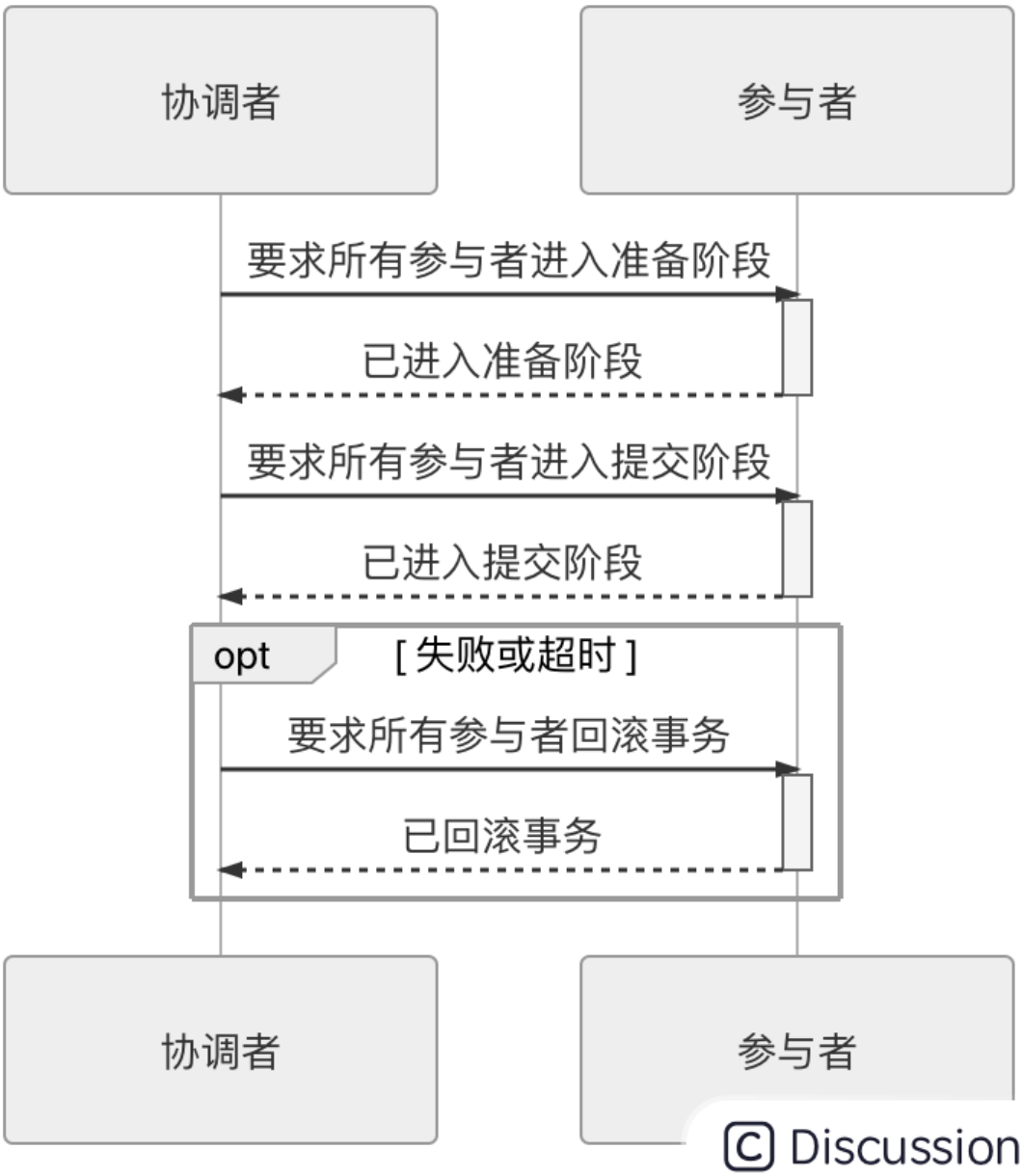
- **提交阶段**

  又叫作执行阶段，协调者如果在上一阶段收到所有事务参与者回复的 Prepared 消息，则先自己在本地持久化事务状态为 Commit，在此操作完成后向所有参与者发送 Commit 指令，所有参与者立即执行提交操作；否则，任意一个参与者回复了 Non-Prepared 消息，或任意一个参与者超时未回复，协调者将将自己的事务状态持久化为 Abort 之后，向所有参与者发送 Abort 指令，参与者立即执行回滚操作。对于数据库来说，这个阶段的提交操作应是很轻量的，仅仅是持久化一条 Commit Record 而已，通常能够快速完成，只有收到 Abort 指令时，才需要根据回滚日志清理已提交的数据，这可能是相对重负载操作。

以上这两个过程被称为“两段式提交”(2 Phase Commit，2PC)协议，它能够成功保证一致性还需要一些其他前提条件：

- 网络在提交阶段的短时间内是可靠的，保证提交阶段不会丢失消息。同时网络通信在全过程都不会出现误差，保证可以丢失消息，但不会传递错误的消息。两段式提交中投票阶段失败了可以补救(回滚)，而提交阶段失败了无法补救(不再改变提交或回滚的结果，只能等崩溃的节点重新恢复)，因而此阶段耗时应尽可能短，这也是为了尽量控制网络风险的考虑。
- 必须假设因为网络分区、机器崩溃或者其他原因而导致失联的节点最终能够恢复，不会永久性地处于失联状态。由于在准备阶段已经写入了完整的重做日志，所以当失联机器一旦恢复，就能够从日志中找出已准备妥当但并未提交的事务数据，再而向协调者查询该事务的状态，确定下一步应该进行提交还是回滚操作。

协调者、参与者都是可以由数据库自己来扮演的，不需要应用程序介入。协调者一般是在参与者之间选举产生的，而应用程序相对于数据库来说只扮演客户端的角色。



两段式提交原理简单，但有几个非常显著的缺点：

- **单点问题**

  协调者等待参与者回复时可以有超时机制，允许参与者宕机，但参与者等待协调者指令时无法做超时处理。**一旦协调者宕机所有参与者都会受到影响。** 如果协调者一直没有恢复，没有正常发送 Commit 或者 Rollback 的指令，那所有参与者都必须一直等待。

- **性能问题**

  两段提交过程中，所有参与者相当于被绑定成为一个统一调度的整体，期间要经过两次远程服务调用，三次数据持久化，整个过程将持续到参与者集群中最慢的那一个处理操作结束为止，这决定了两段式提交的**性能通常都较差**。

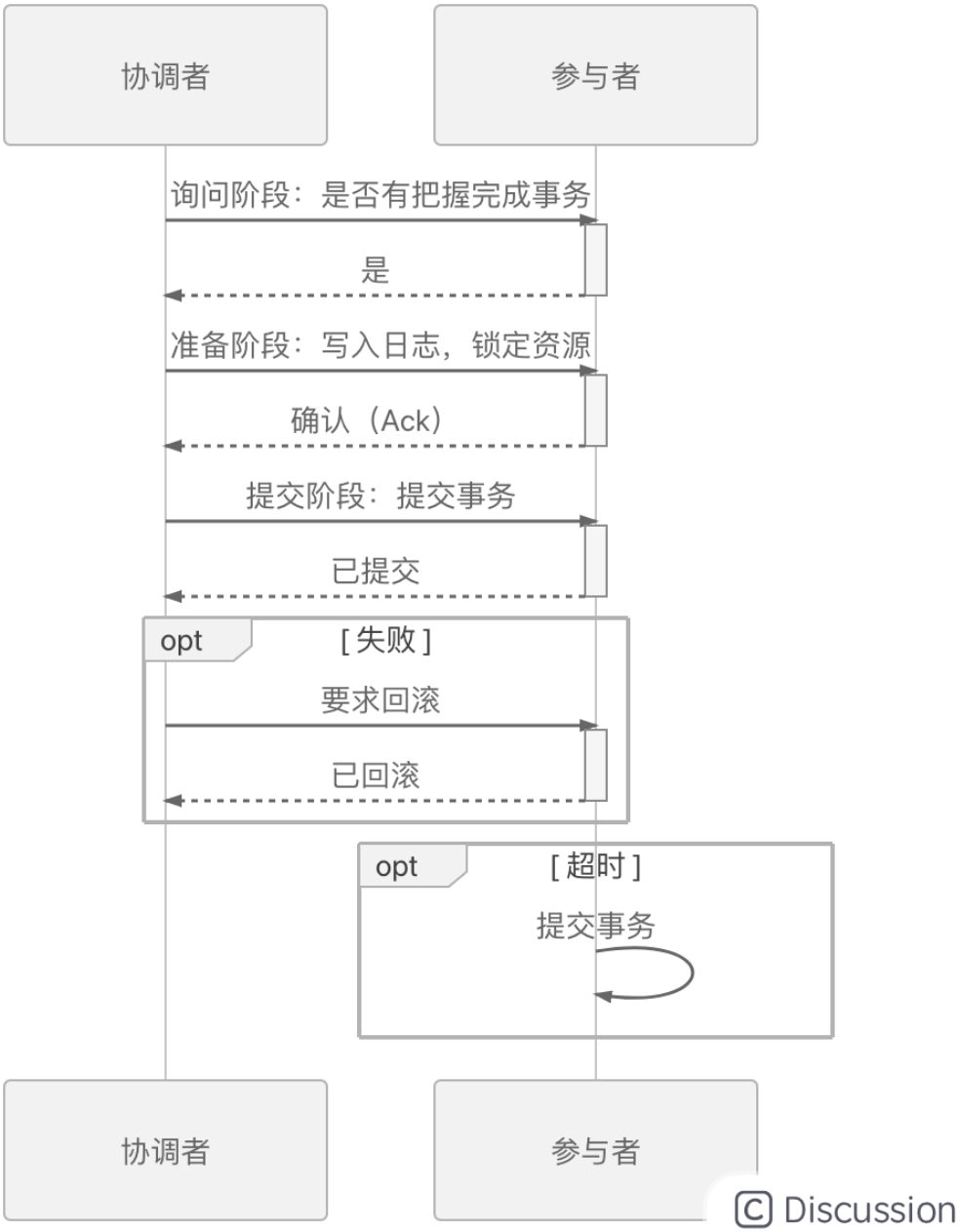
- **一致性风险**

  当网络不稳定或宕机无法恢复可能出现一致性问题。尽管提交阶段时间很短，但仍存在风险。如果协调者在发出准备指令后，根据收到各个参与者发回的信息确定事务状态是可以提交的，协调者会先持久化事务状态，并提交自己的事务，如果这时候网络忽然被断开，无法再通过网络向所有参与者发出 Commit 指令的话，就会导致部分数据(协调者的)已提交，但部分数据(参与者的)既未提交，也没有

办法回滚，产生了数据不一致的问题。

为了缓解两段式提交的单点问题和准备阶段的性能问题，后续发展出了三段式提交(3 Phase Commit，3PC)协议。

三段式提交把原本的两段式提交的准备阶段再细分为两个阶段，分别称为 CanCommit、PreCommit，把提交阶段改称为 DoCommit 阶段。其中，新增的 CanCommit 是一个询问阶段，协调者让每个参与的数据库根据自身状态，评估该事务是否有可能顺利完成。将准备阶段一分为二的理由是这个阶段是重负载的操作，一旦协调者发出开始准备的消息，每个参与者都将马上开始写重做日志，它们所涉及的数据资源即被锁住，如果此时某一个参与者宣告无法完成提交，相当于大家都白做了一轮无用功。所以，增加一轮询问阶段，如果都得到了正面的响应，那事务能够成功提交的把握就比较大了，这也意味着因某个参与者提交时发生崩溃而导致大家全部回滚的风险相对变小。因此，在事务需要回滚的场景中，三段式的性能通常是要比两段式好很多的，但在事务能够正常提交的场景中，两者的性能都依然很差，甚至三段式因为多了一次询问，还要稍微更差一些。



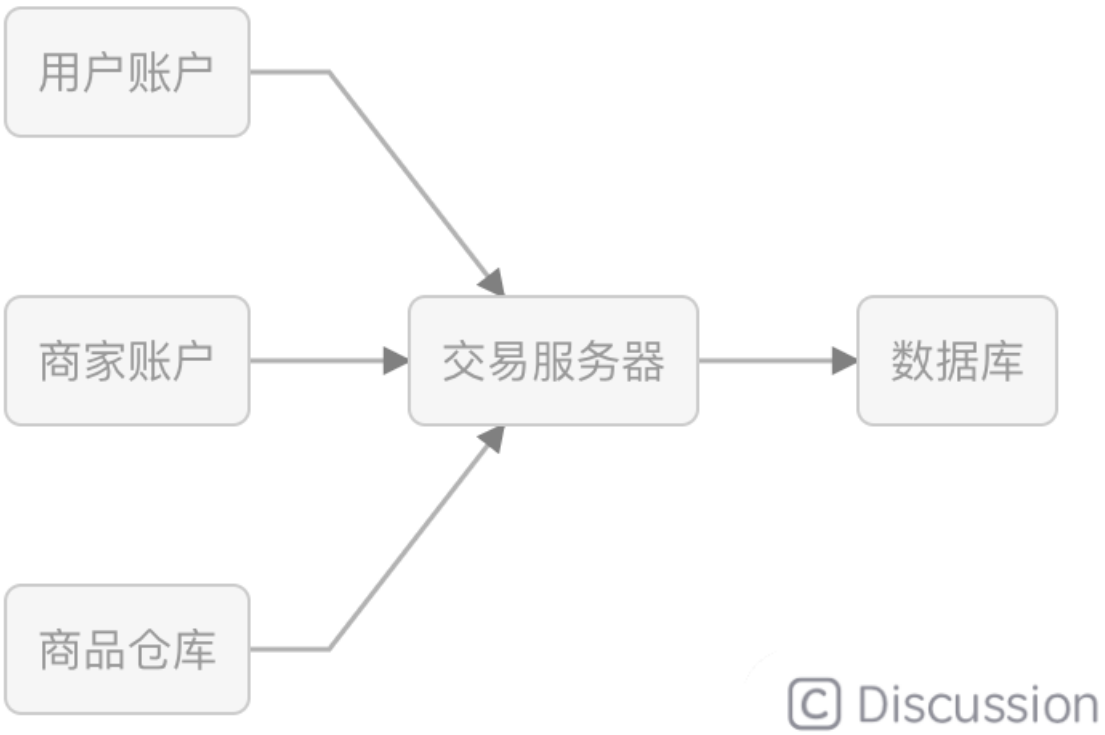
三段式提交对单点问题和回滚时的性能问题有所改善，但是它对一致性风险问题并未有任何改进，在这方面它面临的风险甚至反而是略有增加了的。比如，进入 PreCommit 阶段之后，协调者发出的指令不是 Ack 而是 Abort，而此时因网络问题，有部分参与者直至超时都未能收到协调者的 Abort 指令的话，这些参与者

将会错误地提交事务，这就产生了不同参与者之间数据不一致的问题。

共享事务

共享事务是指多个服务共用同一个数据源。

> 数据源和数据库的区别：数据源是指提供数据的逻辑设备，不必与物理设备一一对应。



分布式事务

分布式事务指多个服务同时访问多个数据源的事务处理机制

CAP 与 ACID

CAP 定理(Consistency、Availability、Partition Tolerance Theorem)，也称为 Brewer 定理，为分布式计算领域所公认的著名定理。这个定理里描述了一个分布式的系统中，涉及共享数据问题时，以下三个特性最多只能同时满足其中两个：

- **一致性**(**C**onsistency)

代表数据在任何时刻、任何分布式节点中所看到的都是符合预期的。

- **可用性**(**A**vailability)

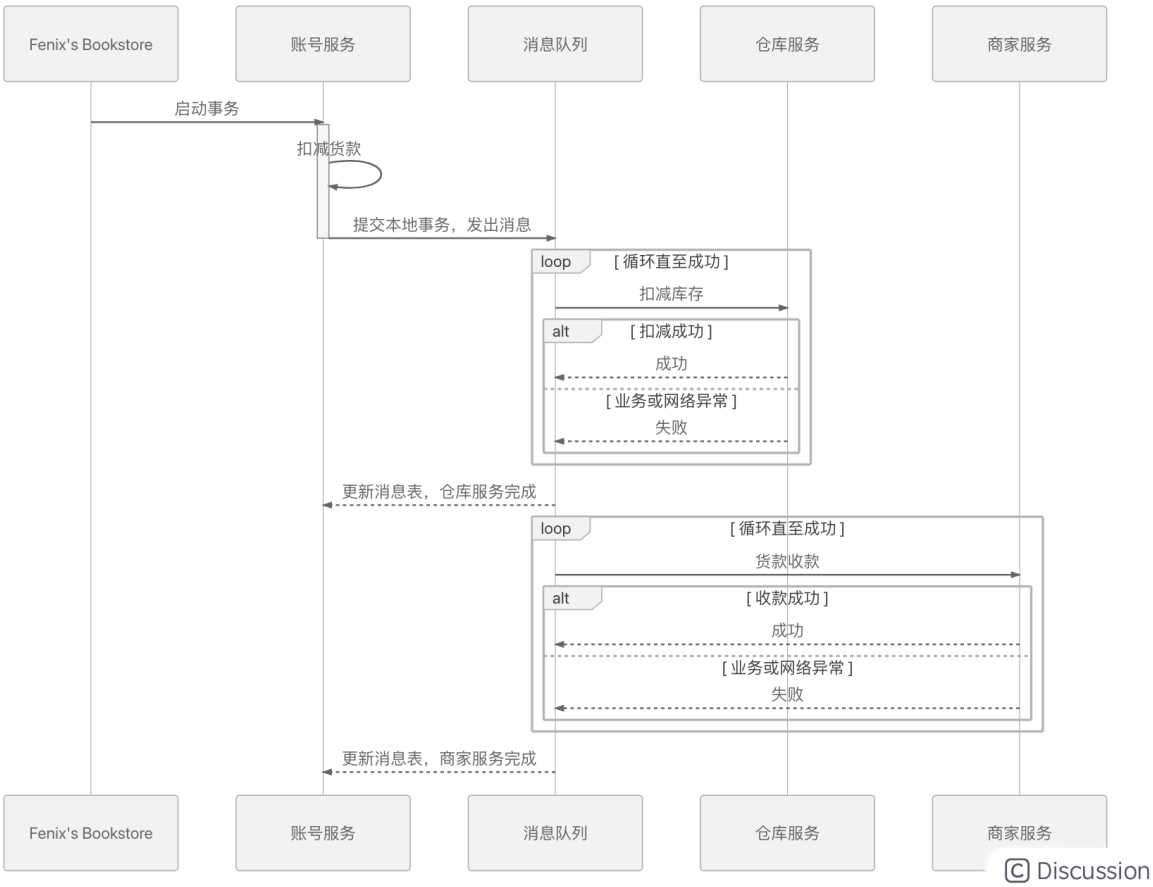
代表系统不间断地提供服务的能力，理解可用性要先理解与其密切相关两个指标：可靠性(Reliability)和可维护性(Serviceability)。可靠性使用平均无故障时间(Mean Time Between Failure，MTBF)来度量；可维护性使用平均可修复时间(Mean Time To Repair，MTTR)来度量。可用性衡量系统可以正常使用的时间与总时间之比，其表征为：A=MTBF/(MTBF+MTTR)，即可用性是由可靠性和可维护性计算得出的比例值，譬如99.9999%可用，即代表平均年故障修复时间为 32 秒。

- **分区容忍性(P**artition Tolerance)

代表分布式环境中部分节点因网络原因而彼此失联后，即与其他节点形成“网络分区”时，系统仍能正确地提供服务的能力。

可靠事件队列

eBay 的系统架构师提出了一种独立于 ACID 获得的强一致性之外的、使用 BASE 来达成一致性目的的途径。BASE 分别是基本可用性(**B**asically **A**vailable)、柔性事务(**S**oft State)和最终一致性(**E**ventually Consistent)的缩写。



TCC 事务

TCC 是另一种常见的分布式事务机制，它是“Try-Confirm-Cancel”三个单词的缩写

可靠消息队列虽能保证最终的结果是相对可靠的，过程也足够简单但整个过程完全没有任何隔离性可言，有一些业务中隔离性是无关紧要的，但有一些业务中缺乏隔离性就会带来许多麻烦。

TCC实现上较为烦琐，是一种业务侵入式较强的事务方案，要求业务处理过程必须拆分为“预留业务资源”和“确认/释放消费资源”两个子过程。它分为以下三个阶段。

- **Try**

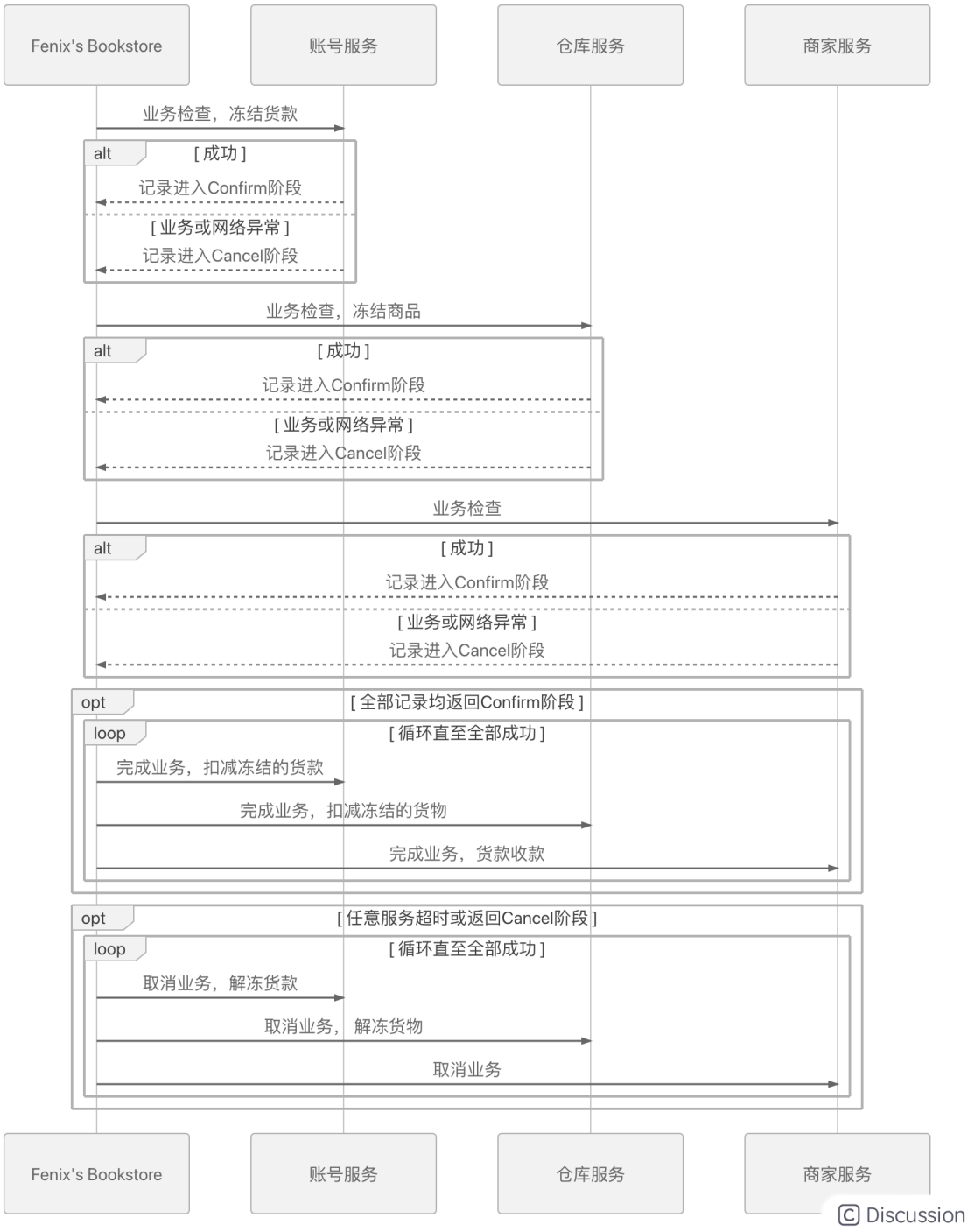
  尝试执行阶段，完成所有业务可执行性的检查，并且预留好全部需用到的业务资源。

- **Confirm**

  确认执行阶段，不进行任何业务检查，直接使用 Try 阶段准备的资源来完成业务处理。Confirm 阶段可能会重复执行，因此本阶段所执行的操作需要具备幂等性。

- **Cancel**

  取消执行阶段，释放 Try 阶段预留的业务资源。Cancel 阶段可能会重复执行，也需要满足幂等性。



TCC 类似 2PC 的准备阶段和提交阶段，但 TCC 是位于用户代码层面，而不是在基础设施层面。

SAGA 事务

TCC由于它的业务侵入性很强所以不能满足所有的场景，我们在有的时候可以考虑采用SAGA事务，SAGA在英文中是“长篇故事、长篇记叙、一长串事件”的意思。

SAGA 由两部分操作组成。

> 大事务拆分若干个小事务，将整个分布式事务 T 分解为 n 个子事务，命名为 T_1，T_2，…，T_i，…，T_n。每个子事务都应该是或者能被视为是原子行为。如果分布式事务能够正常提交，其对数据的影响（最终一致性）应与连续按顺序成功提交 T_i等价。
> 为每一个子事务设计对应的补偿动作，命名为 C_1，C_2，…，C_i，…，C_n。T_i 与 C_i必须满足以下条件：
>
> > T_i与 C_i都具备幂等性。
> > T_i与 C_i满足交换律（Commutative），即先执行 T_i还是先执行 C_i，其效果都是一样的。
> > C_i必须能成功提交，即不考虑 C_i本身提交失败被回滚的情形，如出现就必须持续重试直至成功，或者要人工介入。

如果 T_1到 T_n均成功提交，那事务顺利完成，否则，要采取以下两种恢复策略之一：

> **正向恢复**（Forward Recovery）：如果 T_i事务提交失败，则一直对 T_i进行重试，直至成功为止（最大努力交付）。这种恢复方式不需要补偿，适用于事务最终都要成功的场景，譬如在别人的银行账号中扣了款，就一定要给别人发货。正向恢复的执行模式为：T_1，T_2，…，T_i（失败），T_i（重试）…，T_{i+1}，…，T_n。
> **反向恢复**（Backward Recovery）：如果 T_i事务提交失败，则一直执行 C_i对 T_i进行补偿，直至成功为止（最大努力交付）。这里要求 C_i必须（在持续重试后）执行成功。反向恢复的执行模式为：T_1，T_2，…，T_i（失败），C_i（补偿），…，C_2，C_1。

透明多级分流系统

现代的企业级或互联网系统，“分流”是必须要考虑的设计

客户端缓存

HTTP 协议的无状态性决定了它必须依靠客户端缓存来解决网络传输效率上的缺陷。

1、强制缓存

HTTP 的强制缓存对一致性处理的策略是很直接的，强制缓存在浏览器的地址输入、页面链接跳转、新开窗口、前进和后退中均可生效，但在用户主动刷新页面时应当自动失效。HTTP 协议中设有**Expires**、**Cache-Control**两类 Header 实现强制缓存。

Expires 是 HTTP 协议最初版本中提供的缓存机制，设计非常直观易懂，但缺点有受限于客户端的本地时间、无法处理涉及到用户身份的私有资源、无法描述“不缓存”的语义。

Cache-Control 是HTTP/1.1 协议中定义的强制缓存 Header，相比于Expires语义更加丰富。

2、协商缓存

强制缓存是基于时效性的，协商缓存是基于变化检测的缓存机制基于变化检测的缓存机制，在一致性上会有比强制缓存更好的表现，但需要一次变化检测的交互开销，性能上就会略差一些。

域名解析

DNS 也许是全世界最大、使用最频繁的信息查询系统，如果没有适当的分流机制，DNS 将会成为整个网络的瓶颈。DNS 的作用是将便于人类理解的域名地址转换为便于计算机处理的 IP 地址。

最近几年出现了另一种新的 DNS 工作模式：HTTPDNS(也称为 DNS over HTTPS，DoH)。它将原本的 DNS 解析服务开放为一个基于 HTTPS 协议的查询服务，替代基于 UDP 传输协议的 DNS 域名解析，通过程序代替操作系统直接从权威 DNS 或者可靠的 Local DNS 获取解析数据，从而绕过传统 Local DNS。好处是完全免去了“中间商赚差价”的环节，不再惧怕底层的域名劫持，能够有效避免 Local DNS 不可靠导致的域名生效缓慢、来源 IP 不准确、产生的智能线路切换错误等问题。

传输链路

传输链路涉及到**连接数优化、传输压缩、快速UDP网络连接**。

内容分发网络

内容分发网络，英文名称Content Distribution Network，简称CDN，如今CDN的应用有

- 加速静态资源
- 安全防御
- 协议升级
- 状态缓存
- 修改资源
- 访问控制
- 注入功能
- 绕过XXX

负载均衡

调度后方的多台机器，以统一的接口对外提供服务，承担此职责的技术组件被称为“负载均衡”。四层负载均衡的优势是性能高，七层负载均衡的优势是功能强。做多级混合负载均衡，通常应是低层的负载均衡在前，高层的负载均衡在后。“四层”、“七层”，指的是OSI 七层模型中第四层传输层和第七层应用层

服务端缓存

软件开发中引入缓存的负面作用要明显大于硬件的缓存，引入缓存会提高系统复杂度，因为你要考虑缓存的失效、更新、一致性等问题；从运维角度来说，缓存会掩盖掉一些缺陷，让问题在更久的时间以后，出现在距离发生现场更远的位置上；从安全角度来说，缓存可能泄漏某些保密数据，也是容易受到攻击的薄弱点。

如果要冒着风险引入缓存，那么总结起来无非是两种情况：

- 为缓解 CPU 压力而做缓存：比如把方法运行结果存储起来、把原本要实时计算的内容提前算好、把

一些公用的数据进行复用，这可以节省 CPU 算力，顺带提升响应性能。
- 为缓解 I/O 压力而做缓存：比如把原本对网络、磁盘等较慢介质的读写访问变为对内存等较快介质的访问，将原本对单点部件(如数据库)的读写访问变为到可扩缩部件(如缓存中间件)的访问，顺带提升响应性能。

缓存虽然是典型以空间换时间来提升性能的手段，但它的出发点是缓解 CPU 和 I/O 资源在峰值流量下的压力，“顺带”而非“专门”地提升响应性能。如果可以通过增强 CPU、I/O 本身的性能来满足需要的话，那升级硬件往往是更好的解决方案，即使需要一些额外的投入成本，也通常要优于引入缓存后可能带来的风险。

缓存属性

设计或者选择缓存至少会考虑以下四个维度的属性：

- **吞吐量**

缓存的吞吐量使用 OPS 值(每秒操作数，Operations per Second，ops/s)来衡量，反映了对缓存进行**并发**读、写操作的效率，即缓存本身的工作效率高低。

缓存的吞吐量只在并发场景中才有统计的意义。

- **命中率**

缓存的命中率即成功从缓存中返回结果次数与总请求次数的比值，反映了引入缓存的价值高低，命中率越低，引入缓存的收益越小，价值越低。

有限的物理存储决定了任何缓存的容量都不可能是无限的，所以缓存需要在消耗空间与节约时间之间取得平衡，缓存必须能够自动或者由人工淘汰掉缓存中的低价值数据目前，最基础的淘汰策略实现方案有以下三种：

1、**FIFO**(First In First Out)：优先淘汰最早进入被缓存的数据。FIFO 实现十分简单，但一般来说它并不是优秀的淘汰策略，越是频繁被用到的数据，往往会越早被存入缓存之中。如果采用这种淘汰策略，很可能会大幅降低缓存的命中率。

2、**LRU**(Least Recent Used)：优先淘汰最久未被使用访问过的数据。LRU 通常会采用 HashMap 加 LinkedList 双重结构(如 LinkedHashMap)来实现，以 HashMap 来提供访问接口，保证常量时间复杂度的读取性能，以 LinkedList 的链表元素顺序来表示数据的时间顺序，每次缓存命中时把返回对象调整到 LinkedList 开头，每次缓存淘汰时从链表末端开始清理数据。对大多数的缓存场景来说，LRU 都明显要比 FIFO 策略合理，尤其适合用来处理短时间内频繁访问的热点对象。但相反，它的问题是如果一些热点数据在系统中经常被频繁访问，但最近一段时间因为某种原因未被访问过，此时这些热点数据依然要面临淘汰的命运，LRU 依然可能错误淘汰价值更高的数据。

3、**LFU**(Least Frequently Used)：优先淘汰最不经常使用的数据。LFU 会给每个数据添加一个访问计数器，每访问一次就加 1，需要淘汰时就清理计数器数值最小的那批数据。LFU 可以解决上面 LRU 中热点数据间隔一段时间不访问就被淘汰的问题，但同时它又引入了两个新的问题，首先是需要对每个缓存的数据专门去维护一个计数器，每次访问都要更新，在上一节“吞吐量”里解释了这样做会带来高昂的维护开销；另一个问题是不便于处理随时间变化的热度变化，譬如某个曾经频繁访问的数据现在不需要了，它也很难自动被清理出缓存。

4、**TinyLFU**(Tiny Least Frequently Used)：TinyLFU 是 LFU 的改进版本。为了缓解 LFU 每次访问都要修改计数器所带来的性能负担，TinyLFU 会首先采用 Sketch 对访问数据进行分析，所谓 Sketch 是统计学上的概念，指用少量的样本数据来估计全体数据的特征，这种做法显然牺牲了一定程度的准确性，但是只要样本数据与全体数据具有相同的概率分布，Sketch 得出的结论仍不失为一种高效与准确之间权衡的有效结论。借助Count–Min Sketch算法(可视为布隆过滤器的一种等价变种结构)，TinyLFU 可以用相对小得多的记

录频率和空间来近似地找出缓存中的低价值数据。为了解决 LFU 不便于处理随时间变化的热度变化问题，TinyLFU 采用了基于“滑动时间窗”的热度衰减算法，简单理解就是每隔一段时间，便会把计数器的数值减半，以此解决“旧热点”数据难以清除的问题。

- **扩展功能**

缓存除了基本读写功能外，还提供哪些额外的管理功能，比如最大容量、失效时间、失效事件、命中率统计，等等。

- **分布式支持**

缓存可分为“进程内缓存”和“分布式缓存”两大类，前者只为节点本身提供服务，无网络访问操作，速度快但缓存的数据不能在各个服务节点中共享，后者则相反。

表 4-4 几款主流进程内缓存方案对比

	ConcurrentHashMap	Ehcache	Guava Cache	Caffeine
访问性能	最高	一般	良好	优秀 接近于 ConcurrentHashMap
淘汰策略	无	支持多种淘汰策略 FIFO、LRU、LFU 等	LRU	W-TinyLFU
扩展功能	只提供基础的访问接口	并发级别控制 失效策略 容量控制 事件通知 统计信息 ……	大致同左	大致同左

缓存风险

缓存不是多多益善，它属于有利有弊，是真正到必须使用时才考虑的解决方案。

缓存穿透

如果查询的数据在数据库中根本不存在的话，缓存里自然也不会有，这类请求的流量每次都不会命中，每次都会触及到末端的数据库，缓存就起不到缓解压力的作用了，这种查询不存在数据的现象被称为缓存穿透。

为了解决缓存穿透，通常会采取下面两种办法：

1. 约定在一定时间内对返回为空的 Key 值依然进行缓存，使得在一段时间内缓存最多被穿透一次。
2. 在缓存之前设置一个布隆过滤器来解决。布隆过滤器是用最小的代价来判断某个元素是否存在于某个集合的办法。如果布隆过滤器给出的判定结果是请求的数据不存在，那就直接返回即可，连缓存都不必去查。虽然维护布隆过滤器本身需要一定的成本，但比起攻击造成的资源损耗仍然是值得的。

缓存击穿

缓存中某些热点数据由于超期而失效，此时又有多个针对该数据的请求同时发送过来，这些请求将全部未能命中缓存，都到达真实数据源中去，导致其压力剧增，这种现象被称为缓存击穿。要避免缓存击穿问题，通常会采取下面的两种办法：

1. 加锁同步，以请求该数据的 Key 值为锁，使得只有第一个请求可以流入到真实的数据源中，其他线程采取阻塞或重试策略。如果是进程内缓存出现问题，施加普通互斥锁即可，如果是分布式缓存中出现的问题，就施加分布式锁，这样数据源就不会同时收到大量针对同一个数据的请求了。
2. 热点数据由代码来手动管理，缓存击穿是仅针对热点数据被自动失效才引发的问题，对于这类数据，

可以直接由开发者通过代码来有计划地完成更新、失效，避免由缓存的策略自动管理。

缓存雪崩

缓存击穿是针对单个热点数据失效，由大量请求击穿缓存而给真实数据源带来压力。有另一种可能是更普遍的情况，不需要是针对单个热点数据的大量请求，而是由于大批不同的数据在短时间内一起失效，导致了这些数据的请求都击穿了缓存到达数据源，同样令数据源在短时间内压力剧增。

出现这种情况，往往是系统有专门的缓存预热功能，也可能大量公共数据是由某一次冷操作加载的，这样都可能出现由此载入缓存的大批数据具有相同的过期时间，在同一时刻一起失效。还有一种情况是缓存服务由于某些原因崩溃后重启，此时也会造成大量数据同时失效，这种现象被称为缓存雪崩。要避免缓存雪崩问题，通常会采取下面的三种办法：

1. 提升缓存系统可用性，建设分布式缓存的集群。
2. 启用透明多级缓存，各个服务节点一级缓存中的数据通常会具有不一样的加载时间，也就分散了它们的过期时间。
3. 将缓存的生存期从固定时间改为一个时间段内的随机时间，譬如原本是一个小时过期，那可以缓存不同数据时，设置生存期为 55 分钟到 65 分钟之间的某个随机时间。

缓存污染

缓存污染是指缓存中的数据与真实数据源中的数据不一致的现象。尽管缓存通常不追求强一致性，但这显然不能等同于缓存和数据源间连最终的一致性都可以不要求了。

缓存污染多数是由开发者更新缓存不规范造成的，从缓存中获得了某个对象，更新了对象的属性，但最后因为某些原因，比如后续业务发生异常回滚了，最终没有成功写入到数据库，此时缓存的数据是新的，数据库中的数据是旧的。

为了尽可能的提高使用缓存时的一致性，已经总结不少更新缓存可以遵循设计模式，譬如 Cache Aside、Read/Write Through、Write Behind Caching 等。其中最简单、成本最低的 Cache Aside 模式是指：

- 读数据时，先读缓存，缓存没有的话，再读数据源，然后将数据放入缓存，再响应请求。
- 写数据时，先写数据源，然后失效(而不是更新)掉缓存。

架构安全性

认证

认证、授权和凭证可以说是一个系统中最基础的安全设计，哪怕再简陋的信息系统也不可能忽略掉“用户登录”功能。

基于通信协议和通信内容的两种认证方式。

- HTTP认证

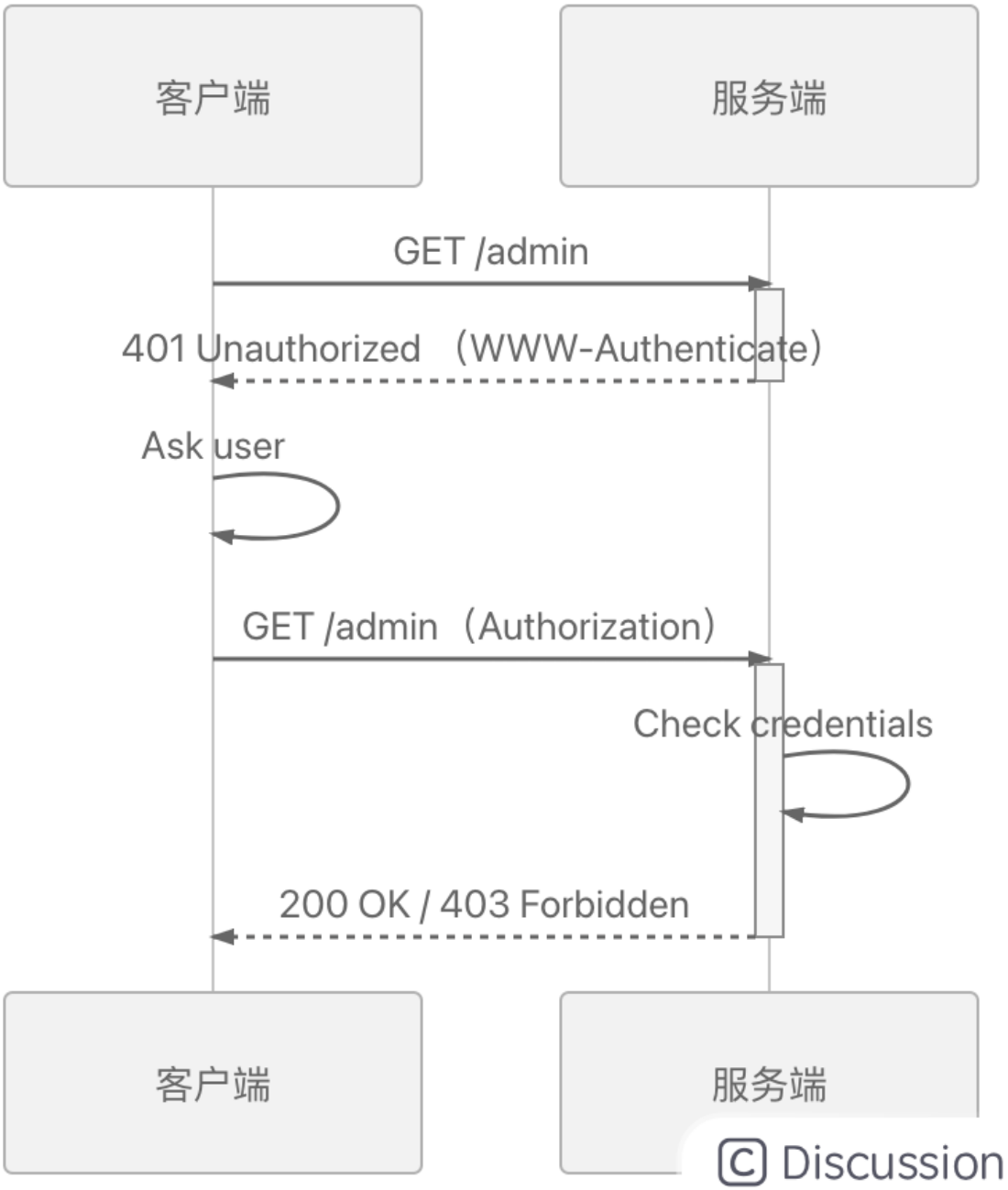
所有支持 HTTP 协议的服务器，在未授权的用户意图访问服务端保护区域的资源时，应返回 401 Unauthorized 的状态码，同时应在响应报文头里附带以下两个分别代表网页认证和代理认证的 Header 之一，告知客户端应该采取何种方式产生能代表访问者身份的凭证信息：

`WWW-Authenticate: <认证方案> realm=<保护区域的描述信息> Proxy-Authenticate: <认证方案> realm=<保护区域的描述信息>`

接收到该响应后，客户端必须遵循服务端指定的认证方案，在请求资源的报文头中加入身份凭证信息，由服务端核实通过后才会允许该请求正常返回，否则将返回 403 Forbidden 错误。请求头报文应包含以下 Header 项之一：

```
Authorization: <认证方案> <凭证内容> Proxy-Authorization: <认证方案> <凭证内容>
```

HTTP 认证框架提出认证方案是希望能把认证“要产生身份凭证”的目的与“具体如何产生凭证”的实现分离开来，无论客户端通过生物信息(指纹、人脸)、用户密码、数字证书抑或其他方式来生成凭证，都属于是如何生成凭证的具体实现，都可以包容在 HTTP 协议预设的框架之内。HTTP 认证框架的工作流程如下。

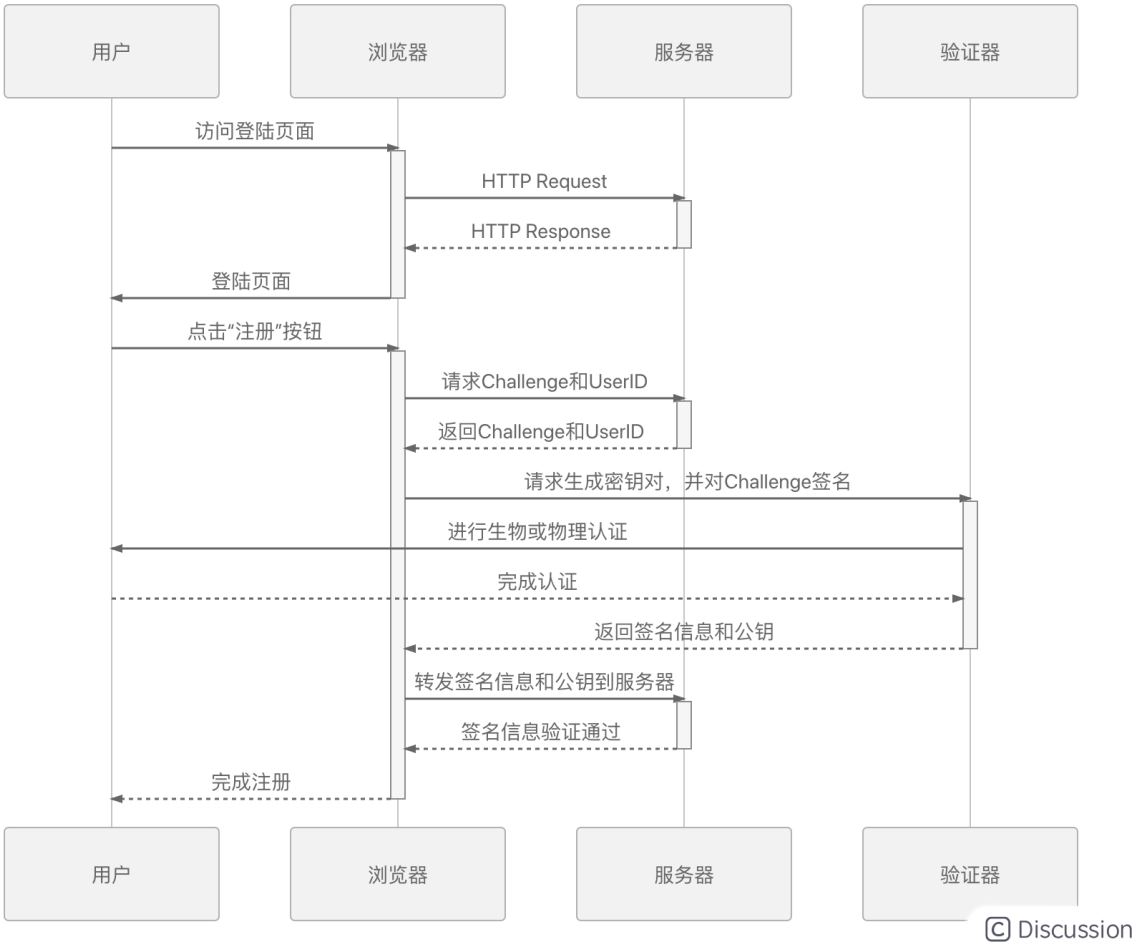


- Web认证

目前的信息系统，尤其是在系统对终端用户的认证场景中，直接采用 HTTP 认证框架的比例其实十分低，因为以 HTTP 协议为基础的认证框架也只能面向传输协议而不是具体传输内容来设计，如果用户想要从服务器中下载文件，弹出一个 HTTP 服务器的对话框，让用户登录是可接受的。但如果用户访问信息系统中的具体服务，身份认证肯定希望是由系统本身的功能去完成的，而不是由 HTTP 服务器来负责认证。这种依靠内容而不是传输协议来实现的认证方式，在万维网里被称为“Web 认证”，由于实现形式上登录表单占了绝对的主流，因此通常也被称为“表单认证"。

WebAuthn 规范涵盖了“注册”与“认证”两大流程，先来介绍注册流程，它大致可以分为以下步骤：

1. 用户进入系统的注册页面，这个页面的格式、内容和用户注册时需要填写的信息均不包含在 WebAuthn 标准的定义范围内。
2. 当用户填写完信息，点击“提交注册信息”的按钮后，服务端先暂存用户提交的数据，生成一个随机字符串(规范中称为 Challenge)和用户的 UserID(在规范中称作凭证 ID)，返回给客户端。
3. 客户端的 WebAuthn API 接收到 Challenge 和 UserID，把这些信息发送给验证器(Authenticator)，验证器可理解为用户设备上 TouchID、FaceID、实体密钥等认证设备的统一接口。
4. 验证器提示用户进行验证，如果支持多种认证设备，还会提示用户选择一个想要使用的设备。验证的结果是生成一个密钥对(公钥和私钥)，由验证器存储私钥、用户信息以及当前的域名。然后使用私钥对 Challenge 进行签名，并将签名结果、UserID 和公钥一起返回客户端。
5. 浏览器将验证器返回的结果转发给服务器。
6. 服务器核验信息，检查 UserID 与之前发送的是否一致，并用公钥解密后得到的结果与之前发送的 Challenge 相比较，一致即表明注册通过，由服务端存储该 UserID 对应的公钥。



授权

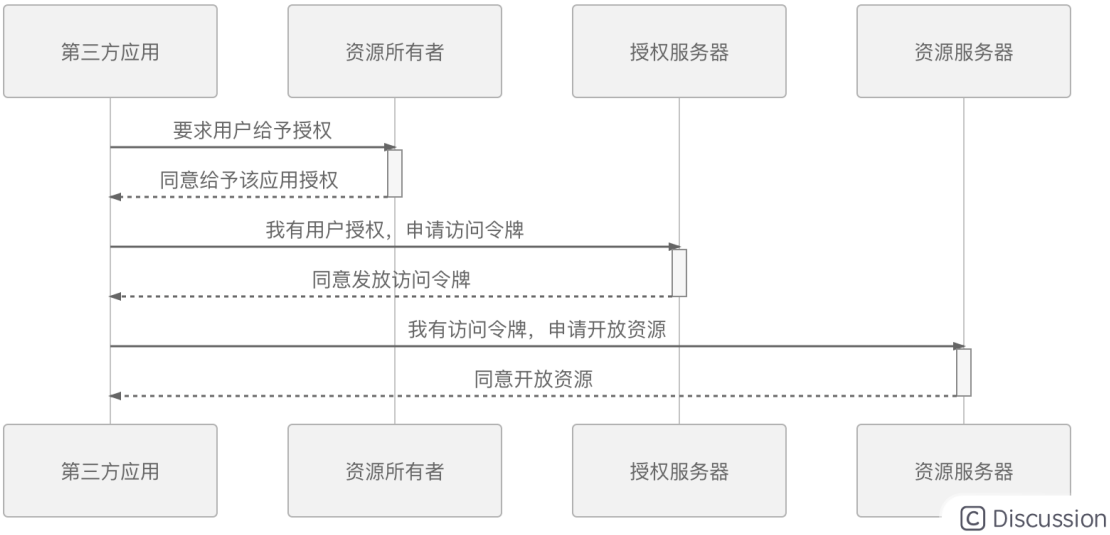
RBAC

所有的访问控制模型，实质上都是在解决同一个问题：“**谁**(User)拥有什么**权限**(Authority)去**操作**(Operation)哪些**资源**(Resource)”。

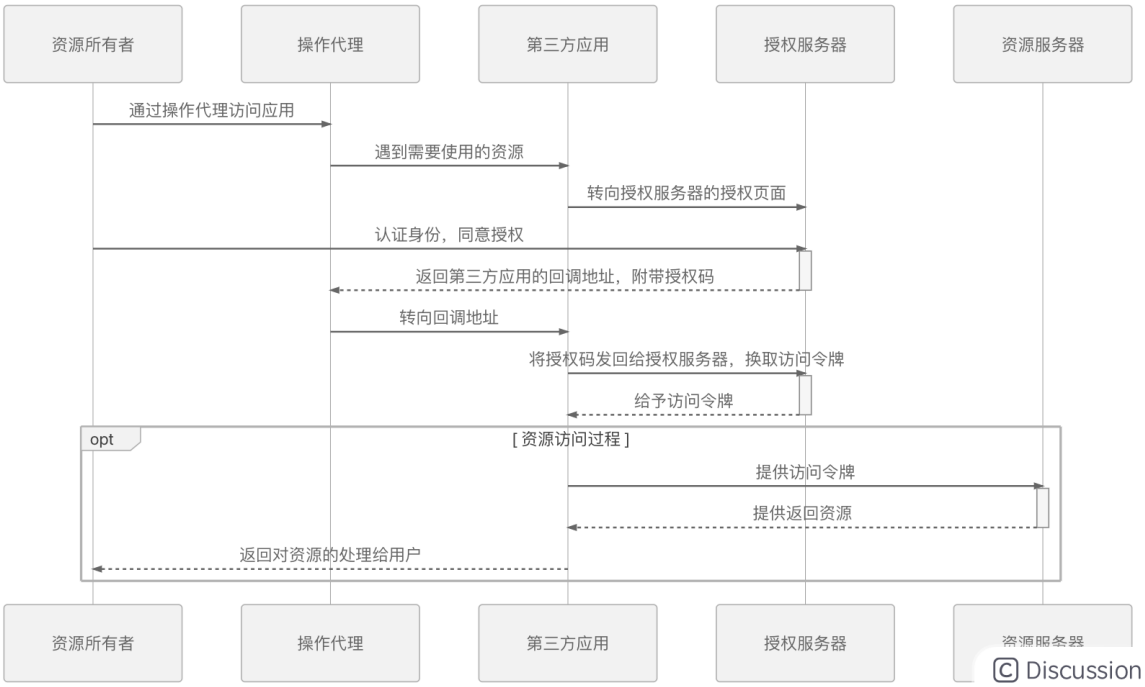
RBAC 将权限从用户身上剥离，改为绑定到“**角色**”(Role)上，将权限控制变为对“**角色**拥有操作哪些**资源**的**许可**”这个逻辑表达式的值是否为真的求解过程。

OAuth2

OAuth2 是**面向于解决第三方应用**的认证授权协议。



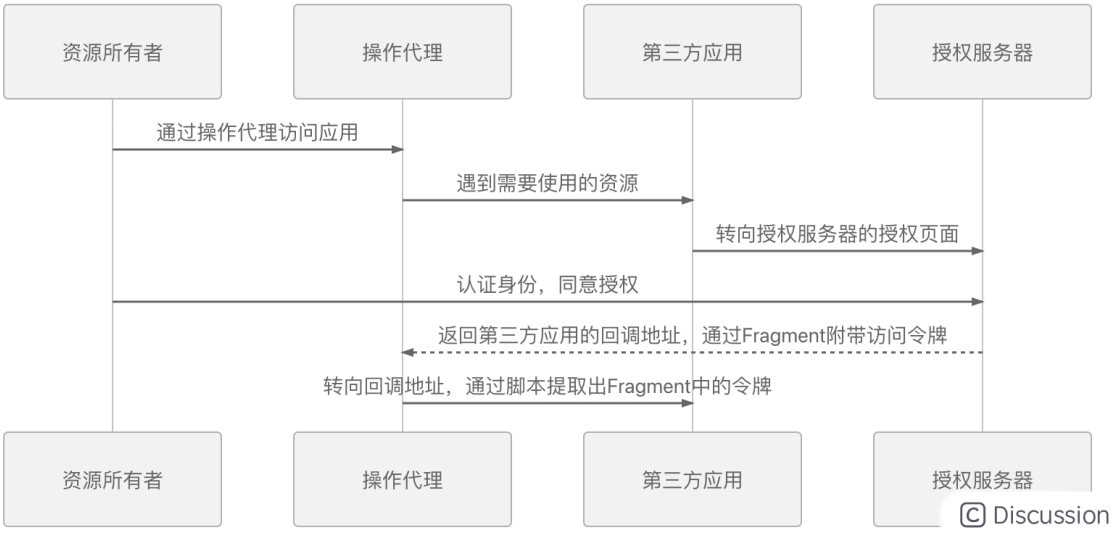
授权码



隐式授权模式

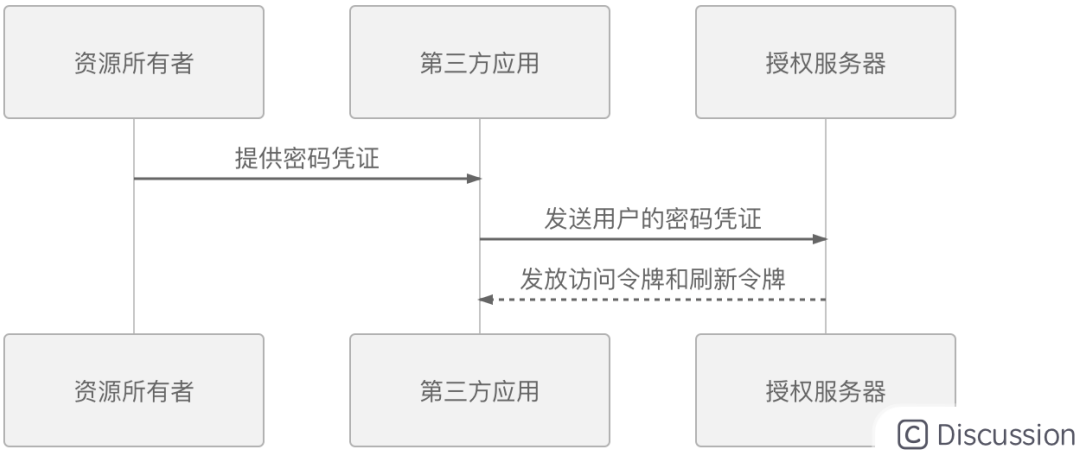
隐式授权省略掉了通过授权码换取令牌的步骤，整个授权过程都不需要服务端支持，一步到位。代价是在隐式授权中，授权服务器不会再去验证第三方应用的身份，因为已经没有应用服务器了，ClientSecret 没有人保管，就没有存在的意义了。但其实还是会限制第三方应用的回调 URI 地址必须与注册时提供的域名一致，尽管有可能被 DNS 污染之类的攻击所攻破，但仍算是尽可能努力一下。同样的原因，也不能避免令牌

暴露给资源所有者，不能避免用户机器上可能意图不轨的其他程序、HTTP 的中间人攻击等风险了。



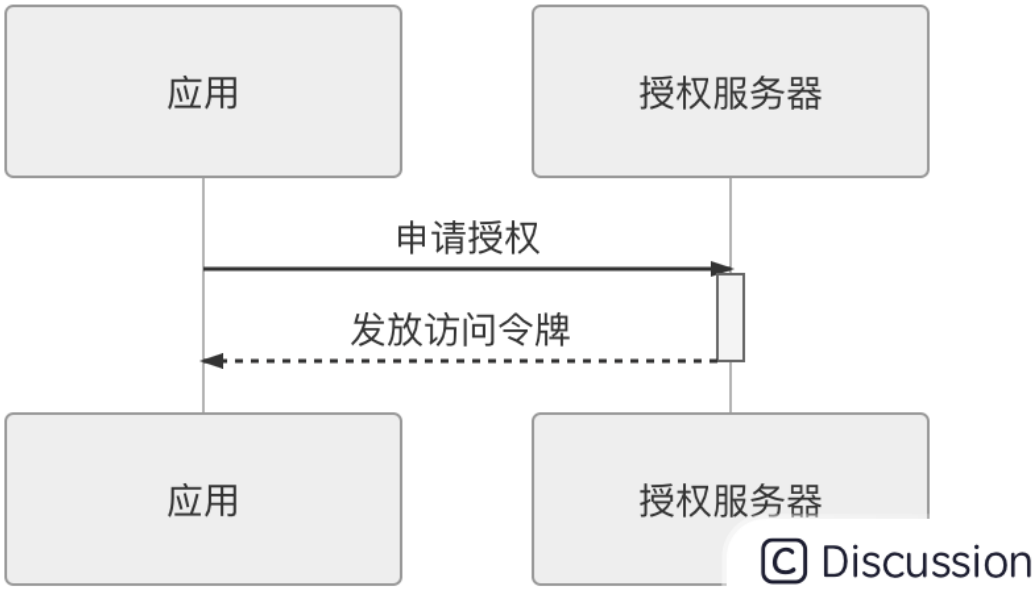
密码模式

密码模式原本的设计意图是仅限于用户对第三方应用是高度可信任的场景中使用，因为用户需要把密码明文提供给第三方应用，第三方以此向授权服务器获取令牌。如果要采用密码模式，那“第三方”属性就必须弱化，把“第三方”视作是系统中与授权服务器相对独立的子模块，在物理上独立于授权服务器部署，但是在逻辑上与授权服务器仍同属一个系统，这样将认证和授权一并完成的密码模式才会有合理的应用场景。



客户端模式

客户端模式只涉及到两个主体，第三方应用和授权服务器。客户端模式是指第三方应用以自己的名义，向授权服务器申请资源许可。



凭证

Cookie-Session

HTTP 协议是一种无状态的传输协议，每一个请求都是完全独立的，但我们也希望 HTTP 能有一种手段，让服务器至少有办法能够区分出发送请求的用户是谁。为了实现这个目的，HTTP 协议中增加了 Set-Cookie 指令，该指令的含义是以键值对的方式向客户端发送一组信息，此信息将在此后一段时间内的每次 HTTP 请求中，以名为 Cookie 的 Header 附带着重新发回给服务端，以便服务端区分来自不同客户端的请求。

根据每次请求传到服务端的 Cookie，服务器就能分辨出请求来自于哪一个用户。由于 Cookie 是放在请求头上的，属于额外的传输负担，不应该携带过多的内容，而且放在 Cookie 中传输也并不安全，容易被中间人窃取或被篡改。

JWT

当服务器存在多个，客户端只有一个时，把状态信息存储在客户端，每次随着请求发回服务器去，这样做的缺点是无法携带大量信息，而且有泄漏和篡改的安全风险。信息量受限的问题并没有太好的解决办法，但是要确保信息不被中间人篡改则还是可以实现的，JWT 便是这个问题的标准答案。

JWT 令牌是多方系统中一种优秀的凭证载体，它不需要任何一个服务节点保留任何一点状态信息，就能够保障认证服务与用户之间的承诺是双方当时真实意图的体现，是准确、完整、不可篡改、且不可抵赖的。同时，由于 JWT 本身可以携带少量信息，这十分有利于 RESTful API 的设计，能够较容易地做成无状态服务，在做水平扩展时就不需要像前面 Cookie-Session 方案那样考虑如何部署的问题。

保密

保密是加密和解密的统称。保密是有成本的，追求越高的安全等级，就要付出越多的工作量与算力消耗。

传输

验证

数据验证与程序如何编码是密切相关的，许多开发者都不会把它归入安全的范畴之中。

分布式的基石

分布式共识算法

- 如果你有一份很重要的数据，要确保它长期存储在电脑上不会丢失，你会怎么做？

答案就是去买几块硬盘，把数据在不同硬盘上多备份几个副本。

在软件系统里要**保障系统的可靠性**采用的办法和我们平时为了存储重要数据，买很多块硬盘存储多个副本是一个道理。要**保障系统的可用性**，面临的困难与硬盘备份却又有着本质的区别，需要考虑动态的数据如何在不可靠的网络通信条件下依然能在各个节点正确复制。

- 如果你有一份会随时变动的数据，要确保它正确地存储于网络中的几台不同机器之上，你会怎么做？

我们要保证数据的一致性就要做到数据同步。每当数据有变化，就要把变化情况在各个节点间做复制操作。而且这种复制操作是一种事务性的操作，只有系统里每一台机器都反馈成功地完成硬盘写入后数据的变化才算成功。

以同步为代表的数据复制方法，被称作状态转移。

- 如果你有一份会随时变动的数据，要确保它正确地存储于网络中的几台不同机器之上，并且要尽可能保证数据是随时可用的，你会怎么做？

可靠性与可用性的矛盾造成了增加机器数量反而带来可用性的降低，为缓解这个矛盾，在分布式系统里主流的数据复制方法是以操作转移为基础的。想要改变数据的状态，除了直接将目标状态赋予它之外，还有另一种常用的方法是通过某种操作，令源状态转换为目标状态。能够使用确定的操作，促使状态间产生确定的转移结果的计算模型，在计算机科学中被称为状态机。

根据状态机的特性，要让多台机器的最终状态一致，只要确保它们的初始状态是一致的，并且接收到的操作指令序列也是一致的即可，无论这个操作指令是新增、修改、删除抑或是其他任何可能的程序行为，都可以理解为要将一连串的操作日志正确地广播给各个分布式节点。广播指令与指令执行期间，允许系统内部状态存在不一致的情况，即并不要求所有节点的每一条指令都是同时开始、同步完成的，只要求在此期间的内部状态不能被外部观察到，且当操作指令序列执行完毕时，所有节点的最终的状态是一致的，这种模型就被称为状态机复制。

分布式环境下网络分区现象是不可能消除的，允许不再追求系统内所有节点在任何情况下的数据状态都一致，而是采用“少数服从多数”的原则，一旦系统中过半数的节点中完成了状态的转换，就认为数据的变化已经被正确地存储在系统当中，这样就可以容忍少数的节点失联，使得增加机器数量对系统整体的可用性变成是有益的，这种思想在分布式中被称为Quorum机制。

Paxos

Paxos是一种基于消息传递的协商共识算法，是分布式系统最重要的基础理论。

Paxos 算法将分布式系统中的节点分为三类

- **提案节点**：称为 Proposer，提出对某个值进行设置操作的节点，设置值这个行为就被称之为**提**

案(Proposal)，值一旦设置成功，就是不会丢失也不可变的。【这里的“设置值”不要类比成程序中变量赋值操作，应该类比成日志记录操作】

- **决策节点**：称为 Acceptor，是应答提案的节点，决定该提案是否可被投票、是否可被接受。提案一旦得到过半数决策节点的接受，即称该提案被**批准**(Accept)，提案被批准即意味着该值不能再被更改，也不会丢失，且最终所有节点都会接受该它。
- **记录节点**：被称为 Learner，不参与提案，也不参与决策，只是单纯地从提案、决策节点中学习已经达成共识的提案，比如少数派节点从网络分区中恢复时，将会进入这种状态。

分布式环境中的锁必须是可抢占的。Paxos 算法包括两个阶段，其中，第一阶段“准备”(Prepare)就相当于上面抢占锁的过程。如果某个提案节点准备发起提案，必须先向所有的决策节点广播一个许可申请(称为 Prepare 请求)。提案节点的 Prepare 请求中会附带一个全局唯一的数字 n 作为提案 ID，决策节点收到后，将会给予提案节点两个承诺与一个应答。

两个承诺是指：

- 承诺不会再接受提案 ID 小于或等于 n 的 Prepare 请求。
- 承诺不会再接受提案 ID 小于 n 的 Accept 请求。

一个应答是指：

- 不违背以前作出的承诺的前提下，回复已经批准过的提案中 ID 最大的那个提案所设定的值和提案 ID，如果该值从来没有被任何提案设定过，则返回空值。如果违反此前做出的承诺，即收到的提案 ID 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。

当提案节点收到了多数派决策节点的应答(称为 Promise 应答)后，可以开始第二阶段“批准”(Accept)过程，这时有如下两种可能的结果：

- 如果提案节点发现所有响应的决策节点此前都没有批准过该值(即为空)，那说明它是第一个设置值的节点，可以随意地决定要设定的值，将自己选定的值与提案 ID，构成一个二元组“(id, value)”，再次广播给全部的决策节点(称为 Accept 请求)。
- 如果提案节点发现响应的决策节点中，已经有至少一个节点的应答中包含有值了，那它就不能够随意取值了，必须无条件地从应答中找出提案 ID 最大的那个值并接受，构成一个二元组“(id, maxAcceptValue)”，再次广播给全部的决策节点(称为 Accept 请求)。

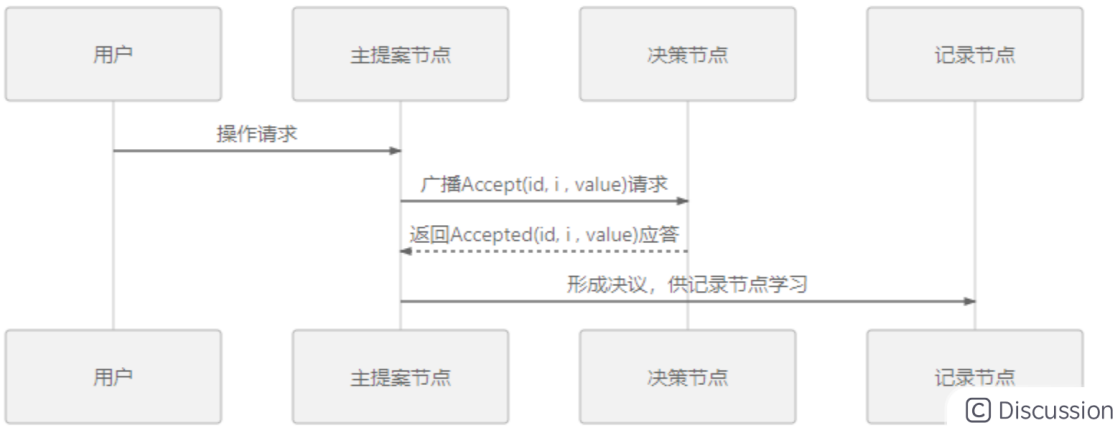
当每一个决策节点收到 Accept 请求时，都会在不违背以前作出的承诺的前提下，接收并持久化对当前提案 ID 和提案附带的值。如果违反此前做出的承诺，即收到的提案 ID 并不是决策节点收到过的最大的，那允许直接对此 Accept 请求不予理会。

当提案节点收到了多数派决策节点的应答(称为 Accepted 应答)后，协商结束，共识决议形成，将形成的决议发送给所有记录节点进行学习。

Multi Paxos

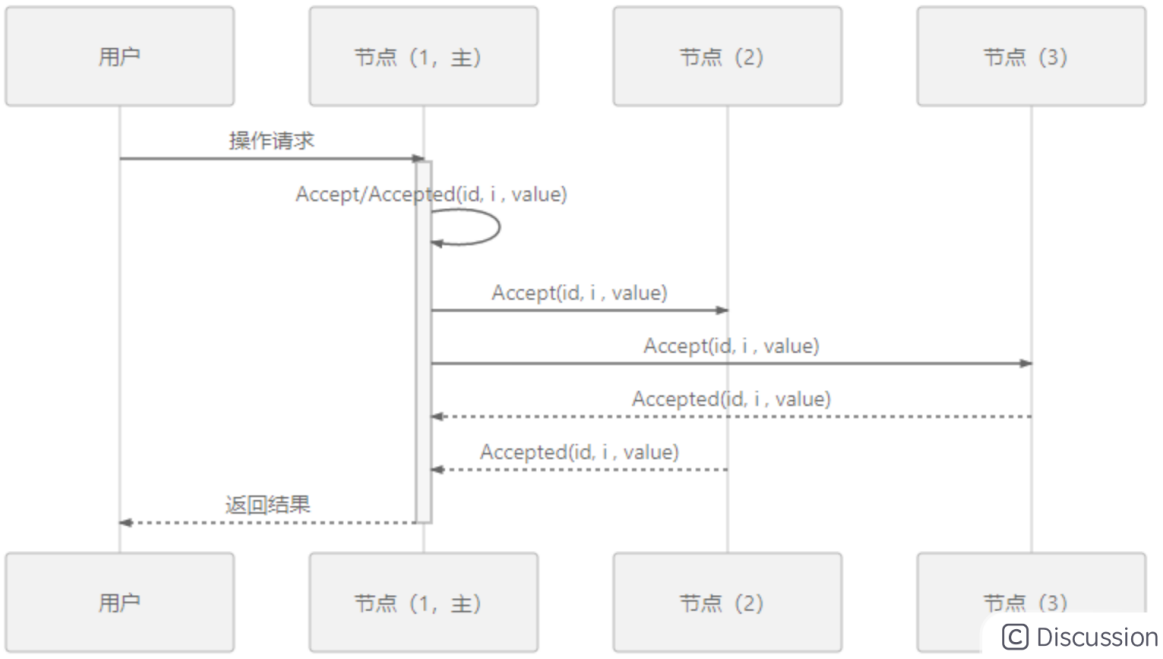
Basic Paxos存在活锁问题，两个提案节点互不相让地争相提出自己的提案，抢占同一个值的修改权限，导致整个系统在持续性地“反复横跳”，外部看起来就像被锁住了一样。

Multi Paxos 对 Basic Paxos 的核心改进是增加了“选主”的过程，提案节点会通过定时轮询(心跳)，确定当前网络中的所有节点里是否存在有一个主提案节点，一旦没有发现主节点存在，节点就会在心跳超时后使用 Basic Paxos 中定义的准备、批准的两轮网络交互过程，向所有其他节点广播自己希望竞选主节点的请求，希望整个分布式系统对“由我作为主节点”这件事情协商达成一致共识，如果得到了决策节点中多数派的批准，便宣告竞选成功。当选主完成之后，除非主节点失联之后发起重新竞选，否则从此往后，就只有主节点本身才能够提出提案。此时，无论哪个提案节点接收到客户端的操作请求，都会将请求转发给主节点来完成提案，而主节点提案的时候，也就无需再次经过准备过程，因为可以视作是经过选举时的那一次准备之后，后续的提案都是对相同提案 ID 的一连串的批准过程。也可以通俗理解为选主过后，就不会再有其他节点与它竞争，相当于是处于无并发的环境当中进行的有序操作，所以此时系统中要对某个值达成一致，只需要进行一次批准的交互即可。



这时候的二元组(id, value)已经变成了三元组(id, i, value)，这是因为需要给主节点增加一个“任期编号”，这个编号必须是严格单调递增的，以应付主节点陷入网络分区后重新恢复，但另外一部分节点仍然有多数派，且已经完成了重新选主的情况，此时必须以任期编号大的主节点为准。

有了选主机制的支持，在整体来看就可以进一步简化节点角色，节点只有主(Leader)和从(Follower)的区别。



`分布式系统中如何对某个值达成一致`这个问题可以划分做三个子问题来考虑，当三个问题同时被解决时，即等价于达成共识。

- 如何选主。
- 如何把数据复制到各个节点上。
- 如何保证过程是安全的。

Gossip 协议

它所解决的问题并不是直接与 Paxos、Raft 这些共识算法等价的，只是基于 Gossip 之上可以通过某些方法去实现与 Paxos、Raft 相类似的目标而已。

一个最典型的例子是比特币网络中使用到了 Gossip 协议，用它来在各个分布式节点中互相同步区块头和区块体的信息，这是整个网络能够正常交换信息的基础，但并不能称作共识；比特币使用工作量证明(Proof of Work，PoW)来对“这个区块由谁来记账”这一件事情在全网达成共识，这个目标才可以认为与 Paxos、Raft 的目标是一致的。

Gossip 的过程十分简单，它可以看作是以下两个步骤的简单循环：

- 如果有某一项信息需要在整个网络中所有节点中传播，那从信息源开始，选择一个固定的传播周期(比如 1 秒)，随机选择它相连接的 k 个节点(称为 Fan-Out)来传播消息。
- 每一个节点收到消息后，如果这个消息是它之前没有收到过的，将在下一个周期内，选择除了发送消息给它的那个节点外的其他相邻 k 个节点发送相同的消息，直到最终网络中所有节点都收到了消息，尽管这个过程需要一定时间，但是理论上最终网络的所有节点都会拥有相同的消息。

从类库到服务

微服务架构其中一个重要设计原则是“通过服务来实现独立自治的组件”强调应采用“服务”，而不再是“类库”来构建组件化的程序，这两者的差别在于类库是在编译期静态链接到程序中的，通过调用本地方法来使用其中的功能，而服务是进程外组件，通过调用远程方法来使用其中的功能。

采用服务来构建程序，好处是软件系统整体与部分在物理层面隔离，但是缺点是复杂性和性能都会有更大的挑战。微服务各个节点形成了一套复杂的网状调用关系，此时，至少有以下三个问题是必须考虑并得到妥善解决的：

- 对消费者来说，外部的服务由谁提供？具体在什么网络位置？
- 对生产者来说，内部哪些服务需要暴露？哪些应当隐藏？应当以何种形式暴露服务？以什么规则在集群中分配请求？
- 对调用过程来说，如何保证每个的远程服务都接收到相对平均的流量，获得尽可能高的服务质量与可靠性？

三个问题的解决方案，在微服务架构中通常被称为“服务发现”、“服务的网关路由”和“服务的负载均衡”。

服务发现

服务发现的过程

- 服务的注册

当服务启动的时候，会通过某些形式将自己的坐标信息通知到服务注册中心，这个过程可能有应用程序本身来完成，称为自注册模式，比如SpringCloud的@EnableEurekaClient注解；也可能有容器编排框架或第三方注册的工具来完成，称为第三方注册模式比如Kubernetes和Registrator

- 服务的维护

服务发现框架必须要自己去保证所维护的服务列表的正确性，以避免告知消费者服务的坐标后，得到的服务却不能使用的尴尬情况。现在的服务发现框架，往往都能支持多种协议(HTTP、TCP 等)、多种方式(长连接、心跳、探针、进程状态等)去监控服务是否健康存活，将不健康的服务自动从服务注册表中剔除。

- 服务的发现

这里的发现是特指狭义上消费者从服务发现框架中，把一个符号(比如 Eureka 中的 ServiceID、Nacos 中的服务名、或者通用的 FQDN)转换为服务实际坐标的过程，这个过程现在一般是通过 HTTP API 请求或者通过 DNS Lookup 操作来完成，也还有一些相对少用的方式，比如 Kubernetes 也支持注入环境变量来做服务发现。

网关路由

微服务中网关的首要职责就是作为统一的出口对外提供服务，将外部访问网关地址的流量，根据适当的规则路由到内部集群中正确的服务节点之上，因此，微服务中的网关，也常被称为“服务网关”或者“API 网关”。微服务中的网关首先应该是个路由器，在满足此前提的基础上，网关还可以根据需要作为流量过滤器来使用，提供某些额外的可选的功能，比如安全、认证、授权、限流、监控、缓存，等等

- 网关 = 路由器(基础职能) + 过滤器(可选职能)

针对“路由”这个基础职能，服务网关主要考量的是能够支持路由的“网络协议层次”和“性能与可用性”两方面的因素。

网关的另一个主要关注点是它的性能与可用性。由于网关是所有服务对外的总出口，是流量必经之地，所以网关的路由性能将导致全局的、系统性的影响，如果经过网关路由会有 1 毫秒的性能损失，就意味着整个系统所有服务的响应延迟都会增加 1 毫秒。

在套接字接口抽象下，网络 I/O 的出入口就是 Socket 的读和写，Socket 在操作系统接口中被抽象为数据

流，网络 I/O 可以理解为对流的操作。每一次网络访问，从远程主机返回的数据会先存放到操作系统内核的缓冲区中，然后内核的缓冲区复制到应用程序的地址空间，所以当发生一次网络请求发生后，将会按顺序经历“等待数据从远程主机到达缓冲区”和“将数据从缓冲区拷贝到应用程序地址空间”两个阶段，根据实现这两个阶段的不同方法，人们把网络 I/O 模型总结为两类、五种模型：两类是指同步 I/O与异步 I/O，五种是指在同步 IO 中又分有划分出阻塞 I/O、非阻塞 I/O、多路复用 I/O和信号驱动 I/O四种细分模型。

同步是指调用端发出请求之后，得到结果之前必须一直等待，与之相对的就是异步，发出调用请求之后将立即返回，不会马上得到处理结果，结果将通过状态变化和回调来通知调用者。

阻塞和非诸塞是针对请求处理过程，指收到调用请求之后，返回结果之前，当前处理线程是否会被挂起。

对网关的可用性方面，应该考虑到以下几点

- 网关应尽可能轻量，尽管网关作为服务集群统一的出入口，可以很方便地做安全、认证、授权、限流、监控，等等的功能，但给网关附加这些能力时还是要仔细权衡，取得功能性与可用性之间的平衡，过度增加网关的职责是危险的。
- 网关选型时，应该尽可能选择较成熟的产品实现，譬如 Nginx Ingress Controller、KONG、Zuul 这些经受过长期考验的产品，而不能一味只考虑性能选择最新的产品，性能与可用性之间的平衡也需要权衡。
- 在需要高可用的生产环境中，应当考虑在网关之前部署负载均衡器或者等价路由器(ECMP)，让那些更成熟健壮的设施(往往是硬件物理设备)去充当整个系统的入口地址，这样网关也可以进行扩展了。

客户端负载均衡

流量治理

服务容错

容错性设计源于分布式系统的本质是不可靠的，一个大的服务集群中，程序可能崩溃、节点可能宕机、网络可能中断，这些“意外情况”其实全部都在“意料之中”。

容错策略

- 故障转移
- 快速失败
- 安全失败
- 沉默失败
- 故障恢复
- 并行调用
- 广播调用

容错策略	优点	缺点	应用场景
故障转移	系统自动处理，调用者对失败的信息不可见	增加调用时间，额外的资源开销	调用幂等服务 对调用时间不敏感的场景
快速失败	调用者有对失败的处理完全控制权 不依赖服务的幂等性	调用者必须正确处理失败逻辑，如果一味只是对外抛异常，容易引起雪崩	调用非幂等的服务 超时阈值较低的场景
安全失败	不影响主路逻辑	只适用于旁路调用	调用链中的旁路服务
沉默失败	控制错误不影响全局	出错的地方将在一段时间类不可用	频繁超时的服务
故障恢复	调用失败后自动重试，也不影响主路逻辑	重试任务可能产生堆积，重试仍然可能失败	调用链中的旁路服务 对实时性要求不高的主路逻辑也可以使用
并行调用	尽可能在最短时间内获得最高的成功率	额外消耗机器资源，大部分调用可能都是无用功	资源充足且对失败容忍度低的场景
广播调用	支持同时对批量的服务提供者发起调用	资源消耗大，失败概率高	只适用于批量操作的场

流量控制

任何一个系统的运算、存储、网络资源都不是无限的，当系统资源不足以支撑外部超过预期的突发流量时，便应该要有取舍，建立面对超额流量自我保护的机制，这个机制就是微服务中常说的“限流”。

一个健壮的系统需要做到恰当的流量控制，更具体地说，需要妥善解决以下三个问题：

- **依据什么限流？**：要不要控制流量，要控制哪些流量，控制力度要有多大，等等这些操作都没法在系统设计阶段静态地给出确定的结论，必须根据系统此前一段时间的运行状况，甚至未来一段时间的预测情况来动态决定。
- **具体如何限流？**：解决系统具体是如何做到允许一部分请求能够通行，而另外一部分流量实行受控制的失败降级，这必须了解掌握常用的服务限流算法和设计模式。
- **超额流量如何处理？**：超额流量可以有不同的处理策略，也许会直接返回失败(如 429 Too Many Requests)，或者被迫使它们进入降级逻辑，这种被称为否决式限流。也可能让请求排队等待，暂时阻塞一段时间后继续处理，这种被称为阻塞式限流。

流量统计指标

做流量控制，首先要弄清楚到底哪些指标能反映系统的流量压力大小。

经常用于衡量服务流量压力，但又较容易混淆的三个指标的定义：

- **每秒事务数**(Transactions per Second，TPS)：TPS 是衡量信息系统吞吐量的最终标准。
- **每秒请求数**(Hits per Second，HPS)：HPS 是指每秒从客户端发向服务端的请求数
- **每秒查询数**(Queries per Second，QPS)：QPS 是指一台服务器能够响应的查询次数