大型平台分析与设计

刘宝

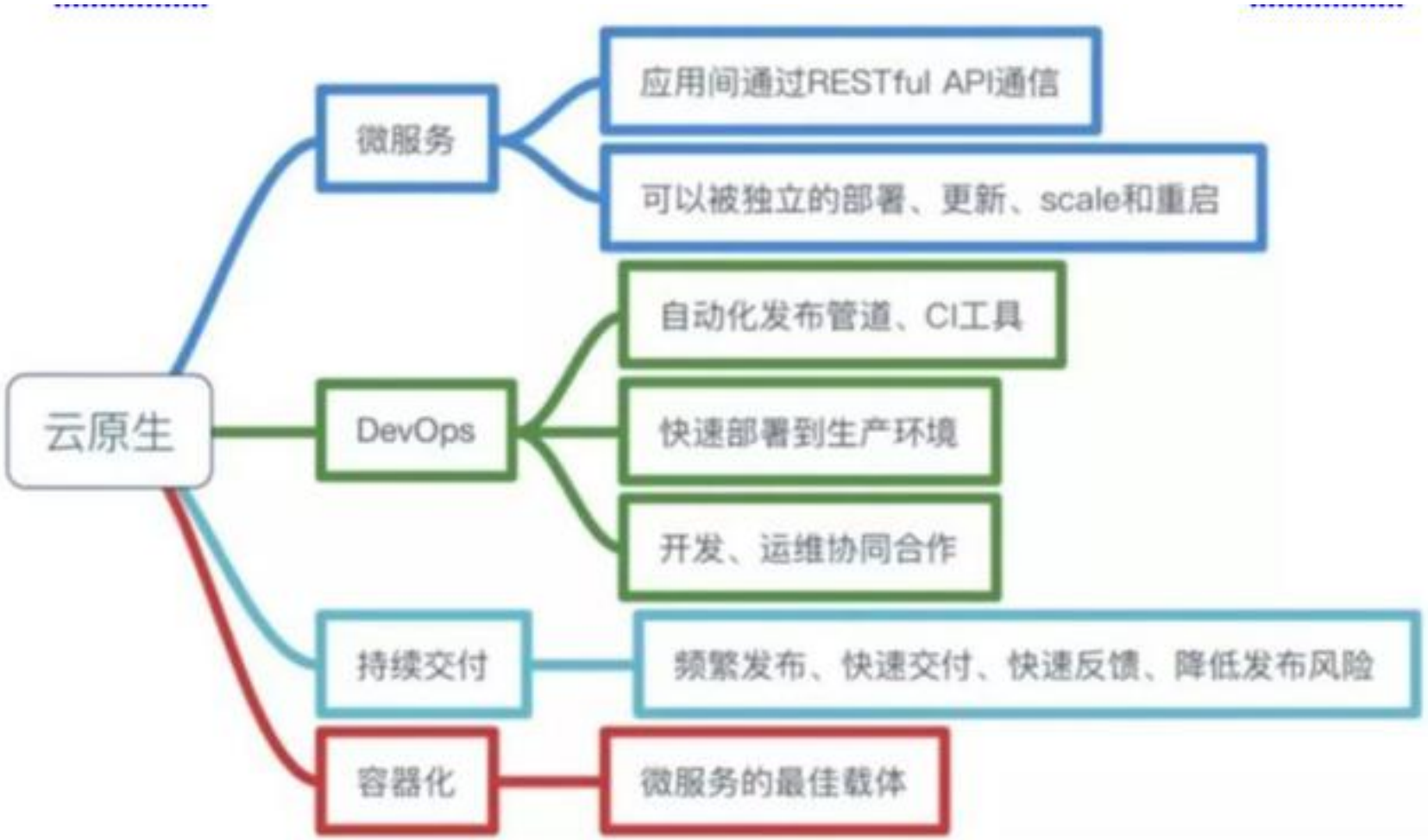
第四章：云原生

- 技术的变革，一定是思想先行，云原生是一种构建和运行应用程序的方法，是一套技术体系和方法论。云原生（CloudNative）是一个组合词，Cloud+Native。Cloud表示应用程序位于云中，而不是传统的数据中心；Native表示应用程序从设计之初即考虑到云的环境，原生为云而设计，在云上以最佳姿势运行，充分利用和发挥云平台的弹性+分布式优势。

云原生

- Pivotal公司的Matt Stine于2013年首次提出云原生（CloudNative）的概念；
- 2015年，云原生刚推广时，Matt Stine在《迁移到云原生架构》一书中定义了符合云原生架构的几个特征：微服务、自敏捷架构、基于API协作、扛脆弱性；同年，Google 主导成立了云原生计算基金会（CNCF），开始围绕云原生的概念打造云原生生态体系
- 2017年，Matt Stine在接受InfoQ采访时又改了口风，将云原生架构归纳为模块化、可观察、可部署、可测试、可替换、可处理6特质；
- 后来重新整理出了四个特性：

云原生


云原生
微服务
应用间通过RESTful API通信
可以被独立的部署、更新、scale和重启
DevOps
自动化发布管道、CI工具
快速部署到生产环境
开发、运维协同合作
持续交付
频繁发布、快速交付、快速反馈、降低发布风险
容器化
微服务的最佳载体

微服务

- 微服务解决的是我们软件开发中一直追求的低耦合+高内聚
- 微服务可以解决这个问题，微服务的本质是把一块大饼分成若干块低耦合的小饼，比如一块小饼专门负责接收外部的数据，一块小饼专门负责响应前台的操作，小饼可以进一步拆分，比如负责接收外部数据的小饼可以继续分成多块负责接收不同类型数据的小饼，这样每个小饼出问题了，其它小饼还能正常对外提供服务。

DevOps

- DevOps的意思就是开发和运维不再是分开的两个团队，而是你中有我，我中有你的一个团队。

很多中小型公司现在开发和运维已经是一个团队了，但是运维方面的知识和经验还需要持续提高。

持续交付

- 持续交付的意思就是在不影响用户使用服务的前提下频繁把新功能发布给用户使用，要做到这点非常非常难。

中小型项目大多两周一个版本，每次上线之后都会给不同的用户造成不同程度的影响。

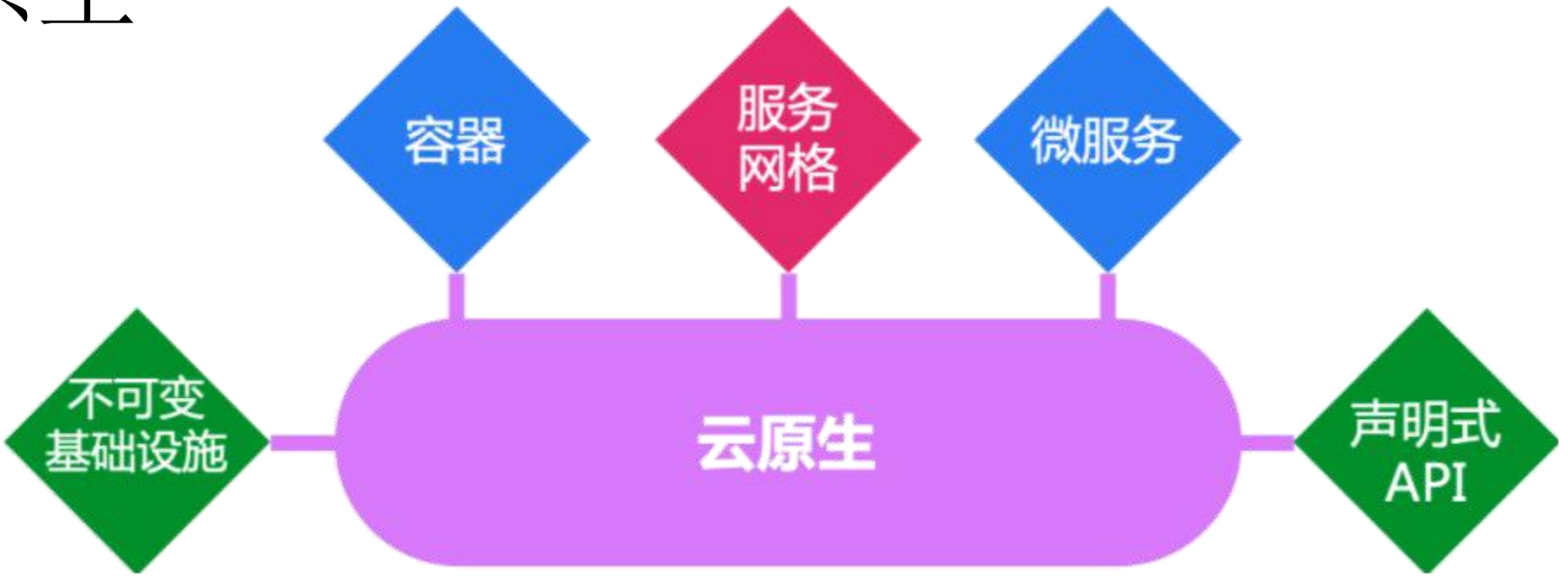
容器化

- 容器化的好处在于运维的时候不需要再关心每个服务所使用的技术栈了，每个服务都被无差别地封装在容器里，可以被无差别地管理和维护，现在比较流行的工具是docker和k8s。

云原生新的定义

- 2018 年，CNCF正式对外公布了更新之后的云原生的定义
- 云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。
- 这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。
- 云原生计算基金会（CNCF）致力于培育和维护一个厂商中立的开源生态系统，来推广云原生技术。我们通过将最前沿的模式民主化，让这些创新为大众所用。

云原生



新的定义中，继续保持原有的核心内容：容器和微服务，但是非常特别的将服务网格单独列出来，而不是将服务网格作为微服务的一个子项或者实现模式，体现了云原生中服务网格这一个新生技术的重要性。而不可变基础设施和声明式API这两个设计指导理念的加入，则强调了这两个概念对云原生架构的影响和对未来发展的指导作用。

服务网格 （Service Mesh）

- 微服务技术架构实践中主要有侵入式架构和非侵入式架构两种实现形式。侵入式架构是指服务框架嵌入程序代码，开发者组合各种组件，如RPC、负载均衡、熔断等，实现微服务架构。非侵入式架构则是以代理的形式，与应用程序部署在一起，接管应用程序的网络且对其透明，开发者只需要关注自身业务即可，以服务网格为代表。为了解决微服务框架的侵入性问题，引入Service Mesh。Serice Mesh提供了专业化的解决方案，其中所涉及的服务通信、容错、认证等功能，都是专业度极高的领城，这些领城应该出现工业级成熟度的制成品，这对于中小企业来说是一个降低成本的选择。

不可变基础设施（Immutable Infrastructure）

- 不可变基础设施里的“不可变”非常类似于程序设计中的“不可变”概念。程序设计中，不可变变量（Immutable Variable）就是在完成赋值后就不能发生更改，只能创建新的来整体替换旧的。**对于基础设施的不可变性，最基本的就是指运行服务的服务器在完成部署后，就不再进行更改。**

声明式API (declarative APIs)

- 声明式 API 的 “声明式” 是什么意思？
- 对于我们使用 Kubernetes API 对象的方式，一般会编写对应 API 对象的 YAML 文件交给 Kubernetes（而不是使用一些命令来直接操作 API）。所谓“声明式”，指的就是我只需要提交一个定义好的 API 对象来“声明”（这个 YAML 文件其实就是一种“声明”），表示所期望的最终状态是什么样子。而如果提交的是一个个命令，去指导怎么一步一步达到期望状态，这就是“命令式”了。“命令式 API”接收的请求只能一个一个实现，否则会有产生冲突的可能；“声明式API”一次能处理多个写操作，并且具备 Merge 能力。Kubernetes 有很多能力，这些能力都是通过各种 API 对象来提供。也就是说，API 对象正是我们使用 Kubernetes 的接口，我们正是通过操作这些提供的 API 对象来使用 Kubernetes 能力的。
- 总结：声明式API其实就是所有资源抽象，抽象成api。这些api 标准化，相当于规范标准了。

云原生与传统应用区别

	传统应用	云原生应用
应用架构	单体式/紧密耦合	微服务/松耦合
应用构成	业务代码+中间件+数据库+运维功能	业务组件+平台
交付周期	长	短且持续
基础设施	以服务器为中心，不可移植	以容器平台为中心，可移植
运维模式	定期巡检+告警通知+人工操作	策略声明+失败自动恢复/回滚
扩容模式	提前采购冗余资源+手动扩展	自动编排，根据指标和策略来弹性伸缩

云原生与传统应用区别

	传统应用	云原生应用
开发语言	C/C++、企业级Java编写	Go、Node.js等新兴语言编写
依赖关系	单体服务耦合严重	微服务化，高内聚，低耦合
部署方式	对机器、网络资源强绑定	容器化，对网络和存储都没有这种限制
运维方式	人肉部署、手工运维	自动化部署，支撑频繁变更，持续交付，蓝绿部署
开发模式	瀑布式开发	DevOps、持续集成、敏捷开发
扩展性	运维手工扩容	动态扩缩容
可用性	故障后可能影响业务	无状态，面向失败编程
可靠性	本地资源，故障率高	云资源，可靠性高

云原生与传统应用区别

云原生应用	传统应用
可预测。云原生应用符合旨在通过可预测行为最大限 度提高弹性的框架或“合同”	不可预测。通常构建时间更长，大批量发布，只能逐 渐扩展，并且会发生更多的单点故障
操作系统抽象化	依赖操作系统
资源调度有弹性	资源冗余较多，缺乏扩展能力
团队借助DevOps更容易达成协作	部门墙导致团队彼此孤立
敏捷开发	瀑布式开发
微服务各自独立，高内聚，低耦合	单体服务耦合严重
自动化运维能力	手动运维
快速恢复	恢复缓慢

云原生与传统应用区别

- 云原生在一个更好的基础平台与设施上提供了更多的应用。因为做了容器化就不需要指定操作系统，K8S 的资源调度更有弹性，之前需要通过代码来协调实现伸缩策略，比较麻烦，借助 DevOps 会容易达成协作，因为它整个流程都是自动的，能够敏捷开发。还有微服是都是各自独立的，具有高内聚、低耦合的原则，具有自动化运维、快速恢复的特点，自愈能力强。当集群宕掉了，它会自动拉起。

- 总结： 云原生与传统应用有比较明显的区别，云原生更倡导敏捷、自动化、容错，而传统应用则大多还处于原生的瀑布开发模型和人工运维阶段。

ServiceMesh

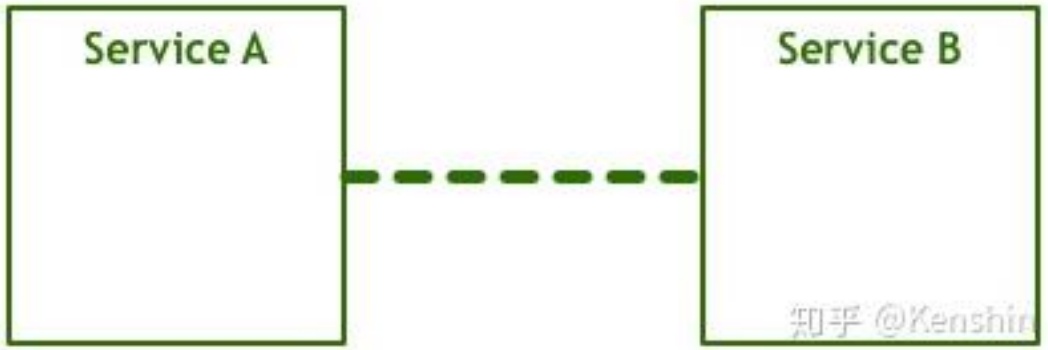
- 什么是ServiceMesh：*Service Mesh* 是微服务时代的 *TCP/IP* 协议。

- 了解下微服务和*ServiceMesh*的发展脉络

ServiceMesh

- 时代0

时代0：开发人员想象中，不同服务间通信的方式，抽象表示如下：

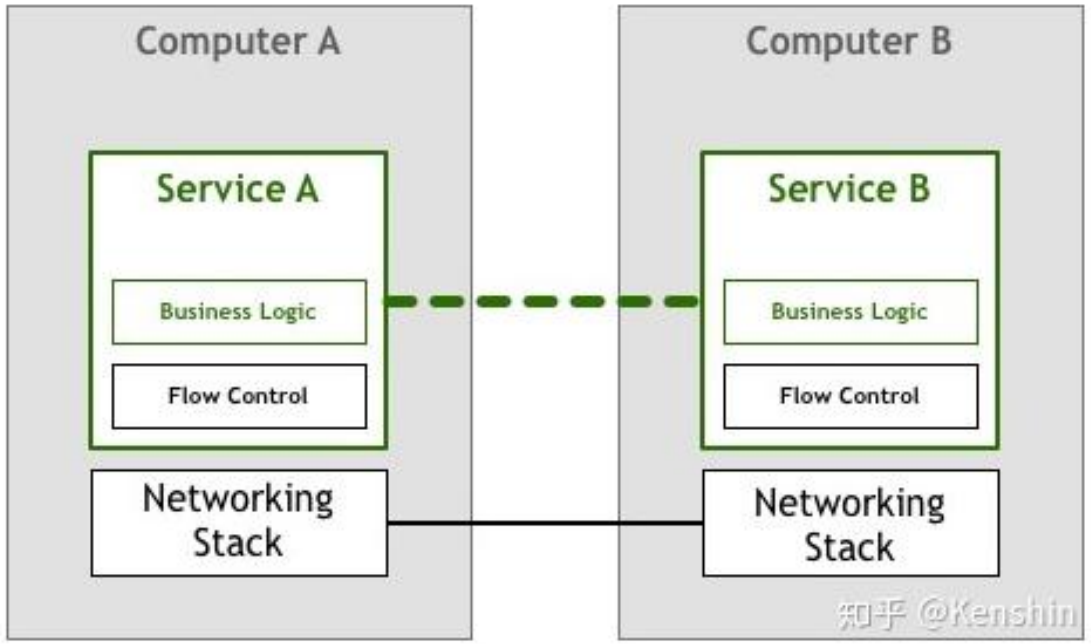


Service

- 时代1

时代1：原始通信时代

然而现实远比想象的复杂，在实际情况中，通信需要底层能够传输字节码和电子信号的物理层来完成，在TCP协议出现之前，服务需要自己处理网络通信所面临的丢包、乱序、重试等一系列流控问题，因此服务实现中，除了业务逻辑外，还夹杂着对网络传输问题的处理逻辑。

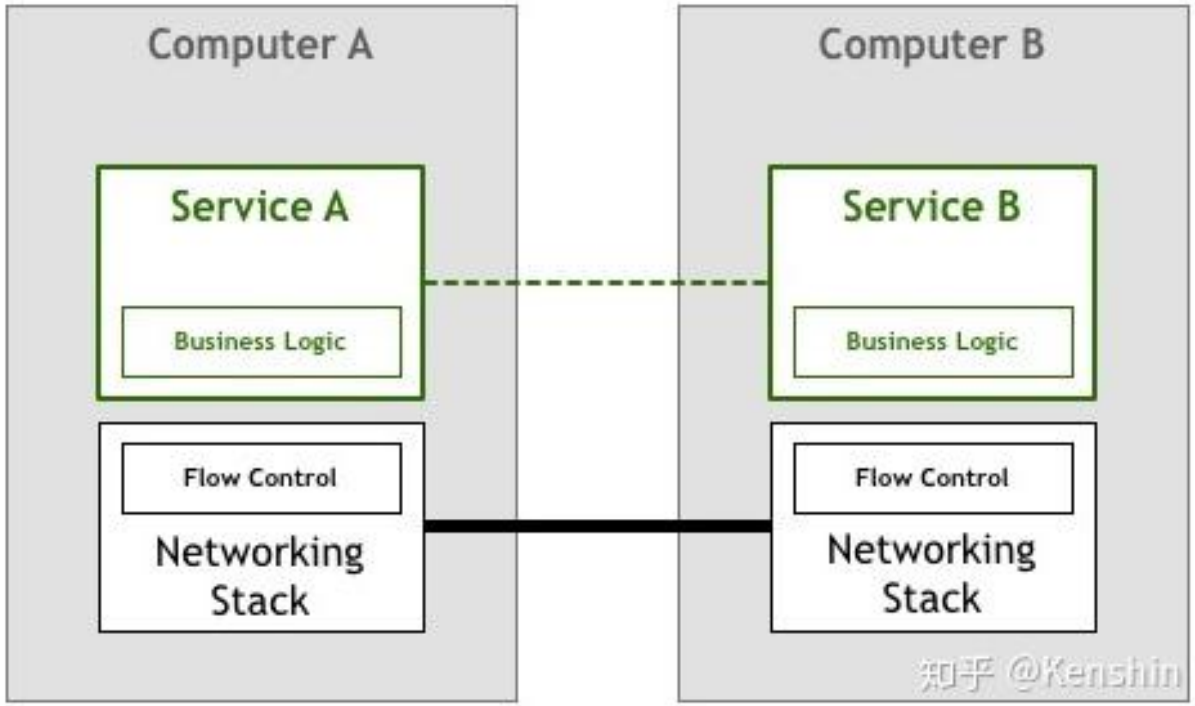


Service

- 时代2

时代2：TCP时代

为了避免每个服务都需要自己实现一套相似的网络传输处理逻辑，TCP协议出现了，它解决了网络传输中通用的流量控制问题，将技术栈下移，从服务的实现中抽离出来，成为操作系统网络层的一部分。

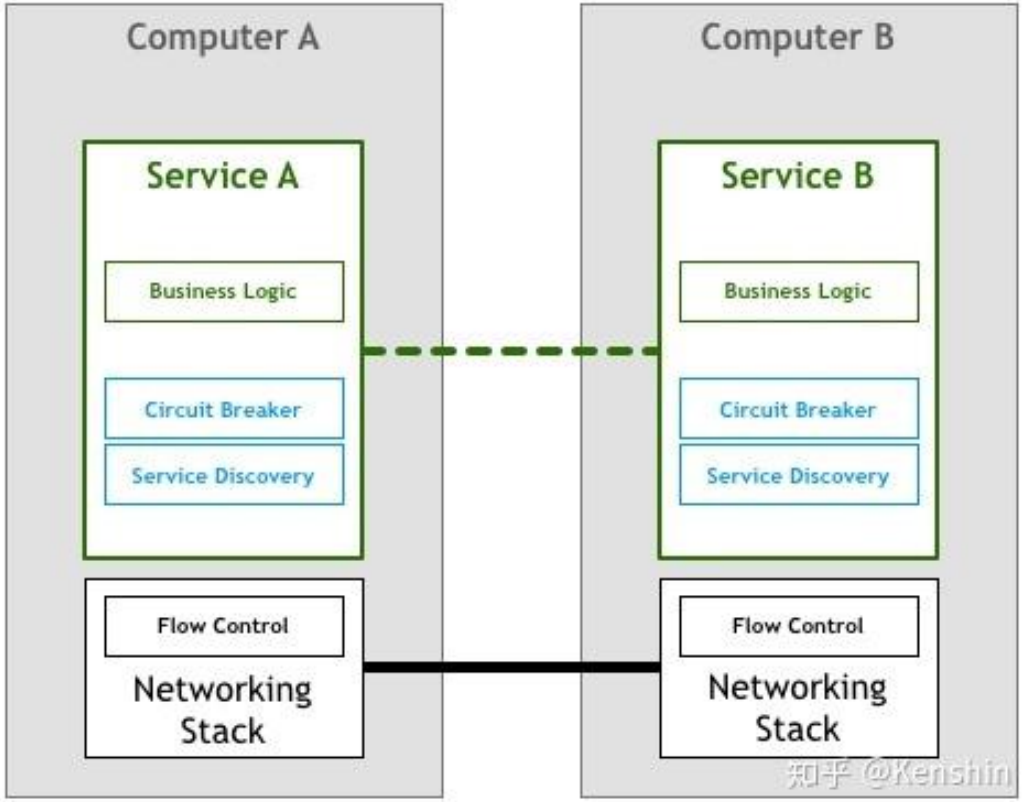


Service

- 时代3

时代3：第一代微服务

在TCP出现之后，机器之间的网络通信不再是一个难题，以GFS/BigTable/MapReduce为代表的分布式系统得以蓬勃发展。这时，分布式系统特有的通信语义又出现了，如熔断策略、负载均衡、服务发现、认证和授权、quota限制、trace和监控等等，于是服务根据业务需求来实现一部分所需的通信语义。

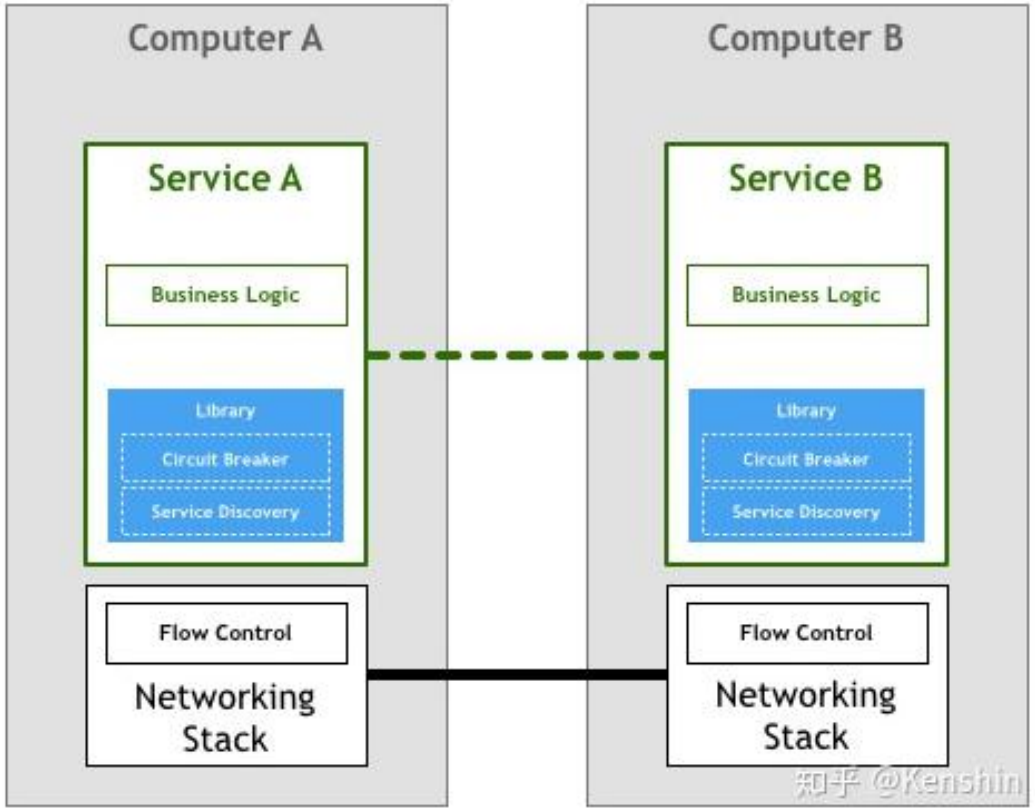


Service

- 时代4

时代4：第二代微服务

为了避免每个服务都需要自己实现一套分布式系统通信的语义功能，随着技术的发展，一些面向微服务架构的开发框架出现了，如Twitter的Finagle、Facebook的Proxygen以及Spring Cloud等等，这些框架实现了分布式系统通信需要的各种通用语义功能：如负载均衡和服务发现等，因此一定程度上屏蔽了这些通信细节，使得开发人员使用较少的框架代码就能开发出健壮的分布式系统。

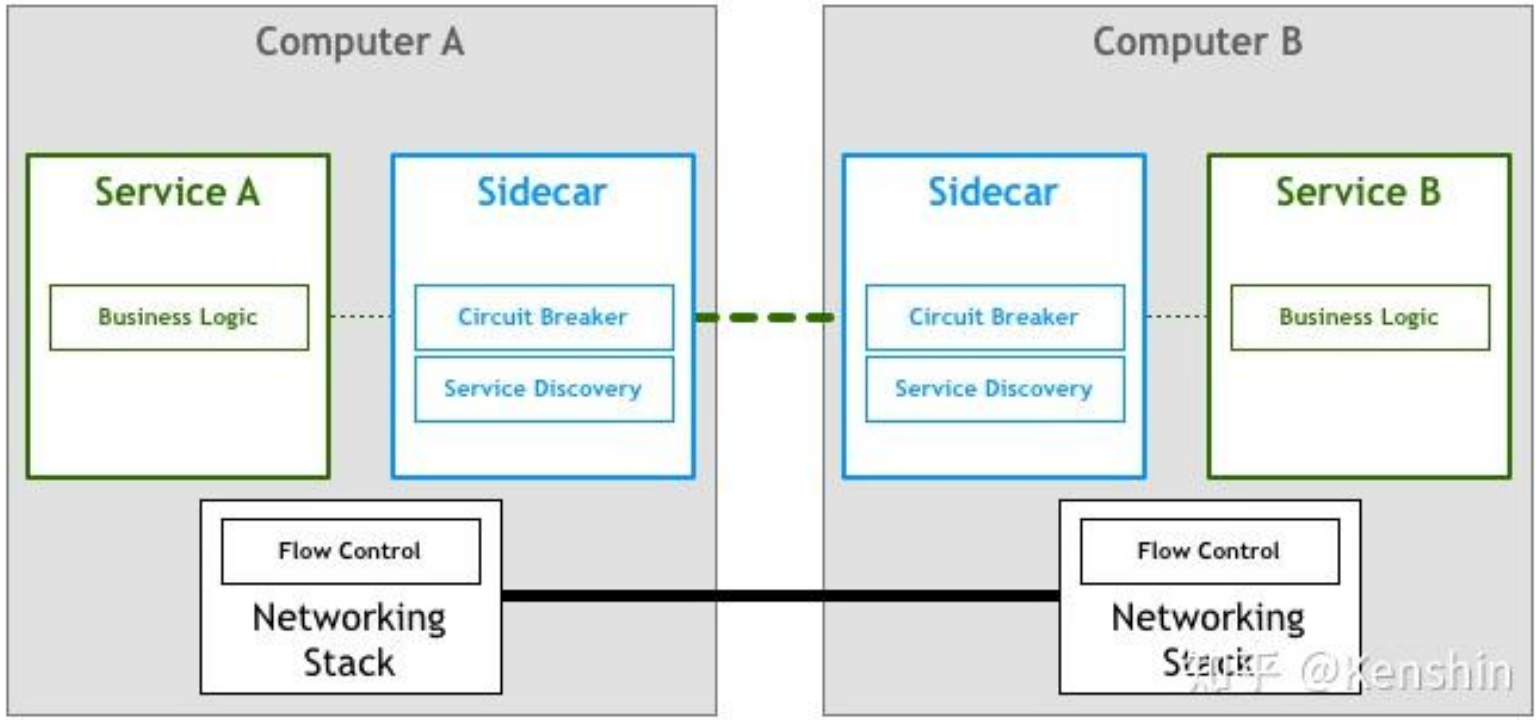


Service

- 时代5：
- 第二代微服务模式看似完美，但开发人员很快又发现，它也存在一些本质问题：
- 其一，虽然框架本身屏蔽了分布式系统通信的一些通用功能实现细节，但开发者却要花更多精力去掌握和管理复杂的框架本身，在实际应用中，去追踪和解决框架出现的问题也绝非易事；
- 其二，开发框架通常只支持一种或几种特定的语言，回过头来看文章最开始对微服务的定义，一个重要的特性就是语言无关，但那些没有框架支持的语言编写的服务，很难融入面向微服务的架构体系，想因地制宜的用多种语言实现架构体系中的不同模块也很难做到；
- 其三，框架以*lib*库的形式和服务联编，复杂项目依赖时的库版本兼容问题非常棘手，同时，框架库的升级也无法对服务透明，服务会因为和业务无关的*lib*库升级而被迫升级；

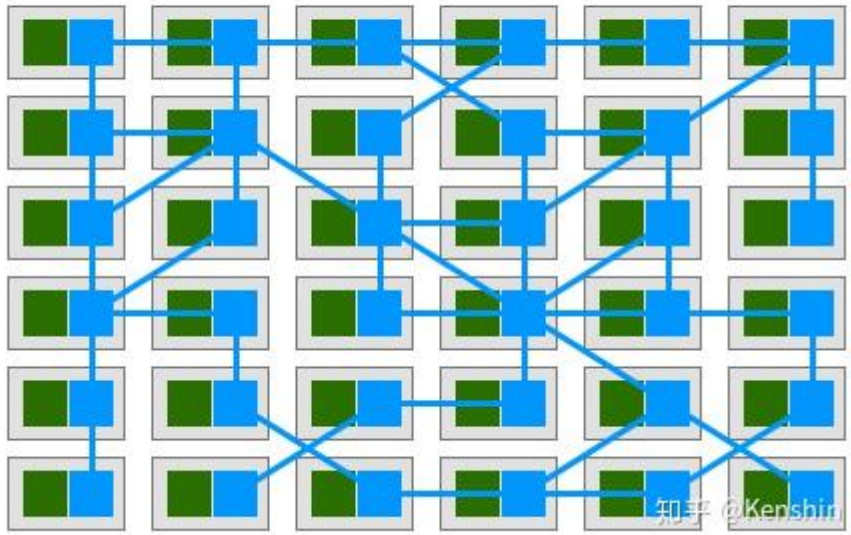
Service

- 这就是第一代*Service Mesh*，它将分布式服务的通信抽象为单独一层，在这一层中实现负载均衡、服务发现、认证授权、监控追踪、流量控制等分布式系统所需要的功能，作为一个和服务对等的代理服务，和服务部署在一起，接管服务的流量，通过代理之间的通信间接完成服务之间的通信请求，这样上边所说的三个问题也迎刃而解。

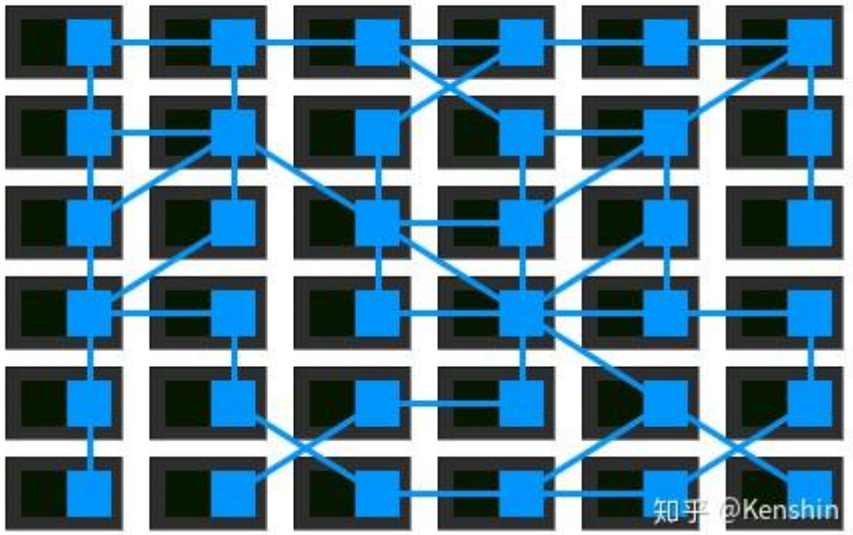


Service

如果我们从一个全局视角来看，就会得到如下部署图：



如果我们暂时略去服务，只看Service Mesh的单机组件组成的网络：

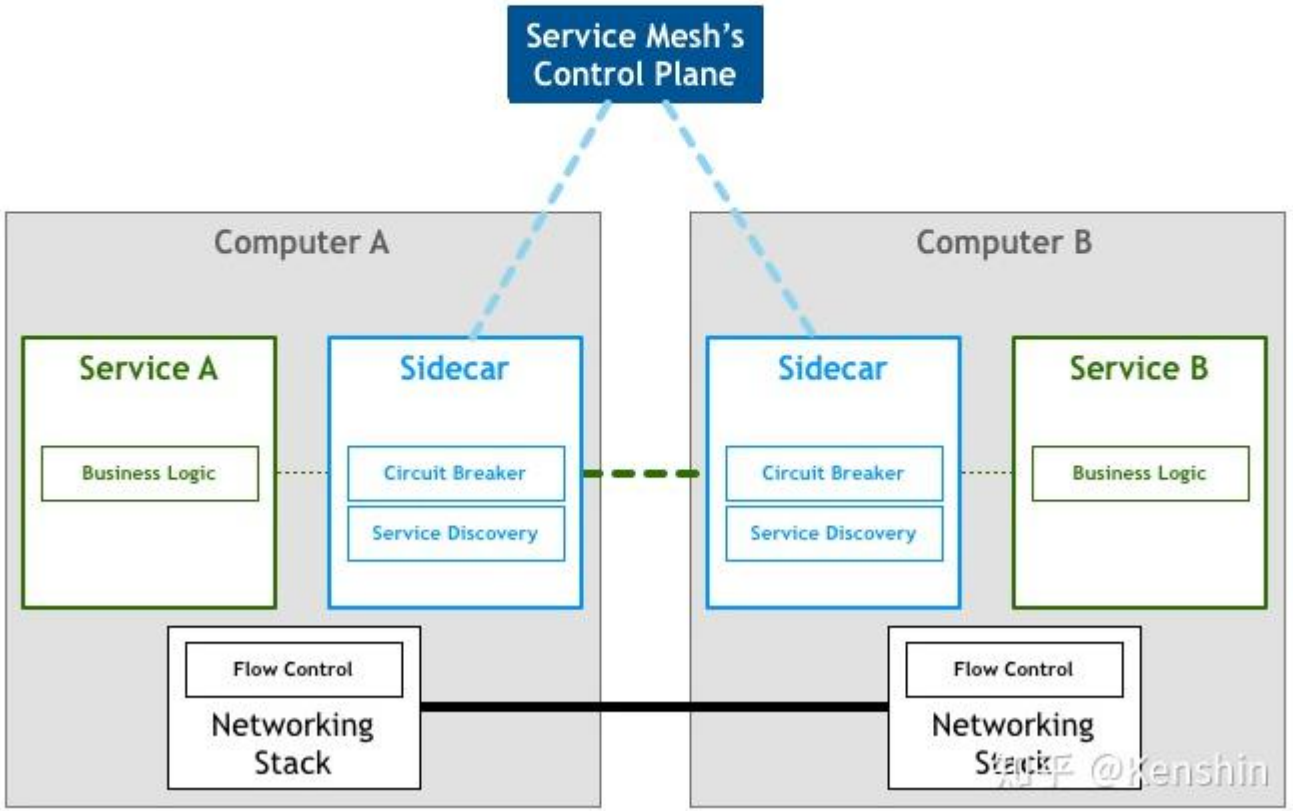


ServiceMesh

时代六

时代6：第二代Service Mesh

第一代Service Mesh由一系列独立运行的单机代理服务构成，为了提供统一的上层运维入口，演化出了集中式的控制面板，所有的单机代理组件通过和控制面板交互进行网络拓扑策略的更新和单机数据的汇报。这就是以Istio为代表的第二代Service Mesh。

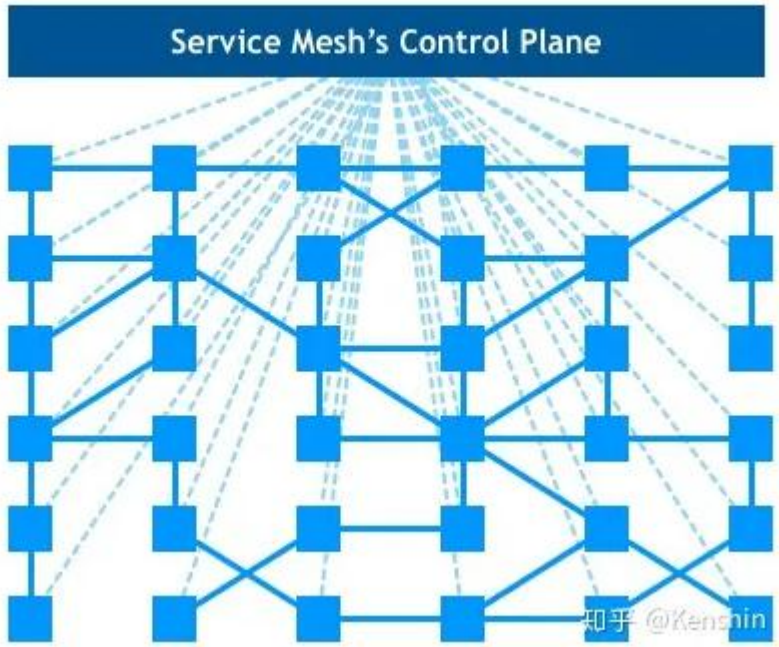


ServiceMesh

至此，见证了6个时代的变迁，大家一定清楚了Service Mesh技术到底是什么，以及是如何一步步演化到今天这样一个形态。

> 服务网格是一个***基础设施层***，用于处理服务间通信。云原生应用有着复杂的服务拓扑，服务网格保证***请求在这些拓扑中可靠地穿梭***。在实际应用当中，服务网格通常是由一系列轻量级的***网络代理***组成的，它们与应用程序部署在一起，但***对应用程序透明***。

只看单机代理组件(数据面板)和控制面板的Service Mesh全局部署视图如下：



ServiceMesh

这个定义中，有四个关键词：

基础设施层+请求在这些拓扑中可靠穿梭：这两个词加起来描述了Service Mesh的定位和功能，是不是似曾相识？没错，你一定想到了TCP；

网络代理：这描述了Service Mesh的实现形态；

对应用透明：这描述了Service Mesh的关键特点，正是由于这个特点，Service Mesh能够解决以Spring Cloud为代表的第二代微服务框架所面临的三个本质问题；

ServiceMesh

- 总结一下，Service Mesh具有如下优点：
- 屏蔽分布式系统通信的复杂性(负载均衡、服务发现、认证授权、监控追踪、流量控制等等)，服务只用关注业务逻辑；
- 真正的语言无关，服务可以用任何语言编写，只需和Service Mesh通信即可；
- 对应用透明，Service Mesh组件可以单独升级；

- 当然，Service Mesh目前也面临一些挑战：
- Service Mesh组件以代理模式计算并转发请求，一定程度上会降低通信系统性能，并增加系统资源开销；
- Service Mesh组件接管了网络流量，因此服务的整体稳定性依赖于Service MeshService Mesh服务实例的运维和管理也是一个挑战；

感谢