Service Mesh发展趋势：云原生中流砥柱

前言

本文内容整理自5月25日在 Kubernetes & Cloud Native Meetup 上海站发表的主题演讲，主要介绍了ServiceMesh最新的产品动态，分析其发展趋势和未来走向；结合蚂蚁的上云实践，阐述在云原生背景下Service Mesh的核心价值，以及对云原生落地的关键作用。

内容主要有三个部分：

1. Service Mesh产品动态：介绍最近半年 Service Mesh 的产品动态，包括开源项目和云厂商推出的云上服务
2. Service Mesh发展趋势：根据最近的产品动态，总结 Service Mesh 的发展趋势，推断未来的走向
3. Service Mesh与云原生：结合云原生，更好的理解 Service Mesh 的价值和作用

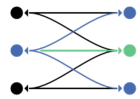
Service Mesh产品动态

Istio1.1发布

Istio是目前 Service Mesh 社区最引人注目的开源项目，在今年的3月份发布了期待已久的 Istio 1.1 版本，我们来看看Istio最近几个版本的发布情况：

- 2018年6月1日，Istio 发布了 0.8 版本，这是Istio历史上第一个LTS版本，也是Istio历史上变动最大的一个版本
- 2018年7月31日，Istio发布了1.0版本，号称 "Product Ready"
- 然后就是漫长的等待，Istio 1.0 系列以每个月一个小版本的方式一路发布了1.0.1 到 1.0.6，然后才开始 1.1.0 snapshot 1到6，再 1.1.0-rc 1到6，终于在2019年3月20日发布了 1.1 版本，号称 "Enterprise Ready"。

从 Istio 1.0 到 Istio 1.1，中间的时间跨度高达9个月！我们来看看经过这漫长的开发时间才发布的 Istio 1.1 版本带来了哪些新的东西：



流量管理

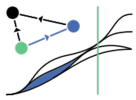
- 新增 Sidecar CRD
- 限制服务可见性：exportTo
- 区域感知路由
- 大幅改进的多集群路由
- 缺省开放 Egress 通信
- 更新了ServiceEntry的资源
- 弃用Istio Ingress

安全

- Readiness and Liveness 探针
- 集群RBAC配置
- 基于SDS的身份设置
- 对 TCP 服务提供鉴权支持
- 最终用户组授权管理
- Gateway上外部证书管理
- 集成Vault PKI
- 自定义的可信域

策略和遥测

- 缺省关闭 Mixer 策略检查
- Kiali 替代 ServiceGraph
- 性能改进，降低开销
- 控制请求头和路由
- 进程外适配器生产可用
- 多方面增强Tracing的能力
- 默认的TCP指标



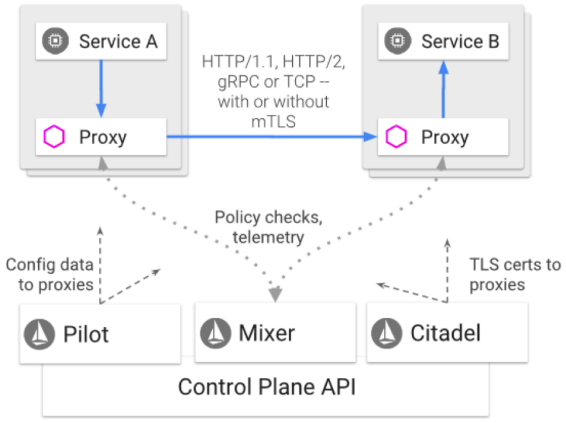
配置管理

- Galley负责配置管理和分发机制
- 引入MCP协议（进行中）

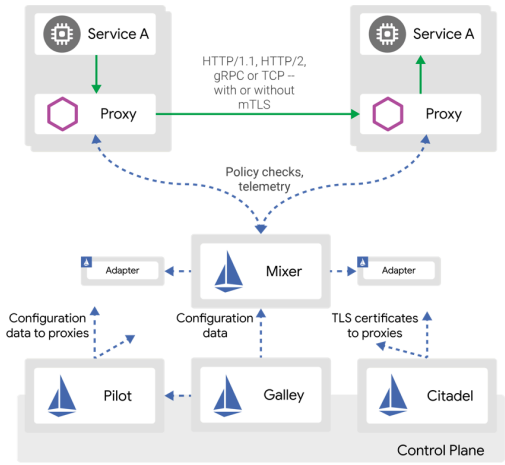
图中标红的部分，涉及到 Istio 的架构调整，下面将详细介绍 Istio 1.1 版本中带来的架构变化。

Istio 1.1架构变化

下图是 Istio 1.0 和 Istio 1.1 的架构图对比：



Istio 1.0

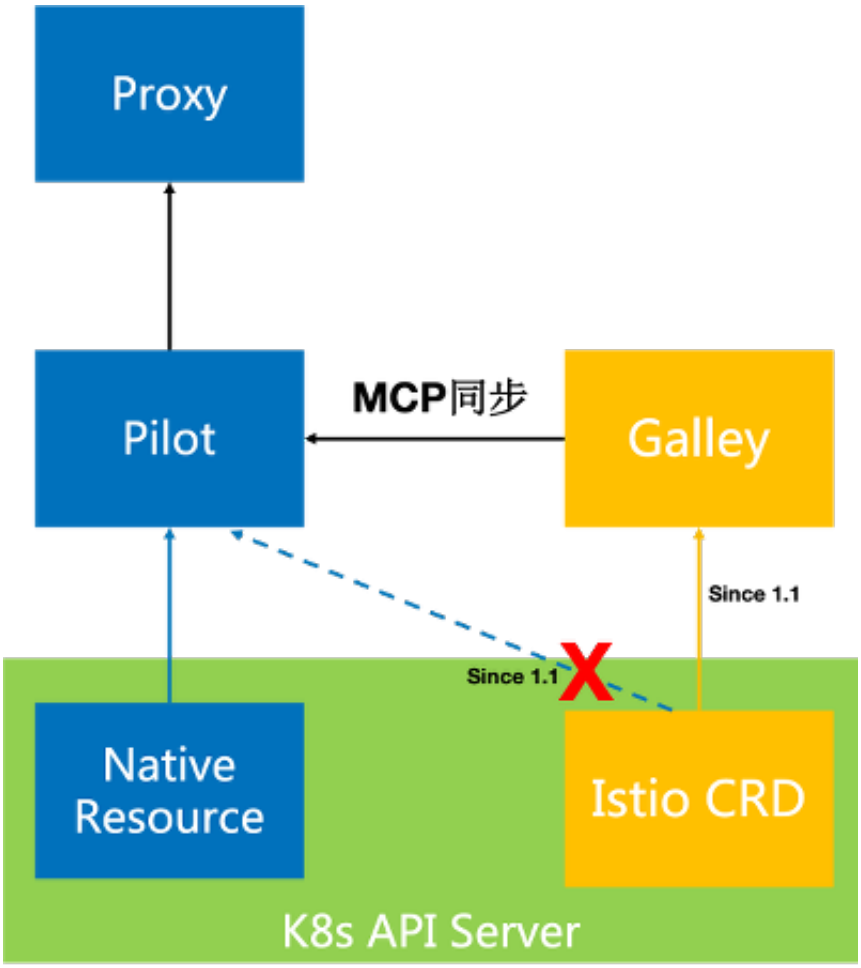


Istio 1.1

Istio 1.1的第一个架构变化来自 Galley：在 Istio 1.1 的架构图中增加了 Galley 组件。但是实际上在 Istio 1.0 版本中 Gallay 组件就已经存在，只是当时 Galley 的功能非常简单，只是做配置更新之后的验证（Validation），在 Istio 1.0 的架构图中都没有出现。而在 Istio 1.1 版本之后，Galley 的定位发生了巨大的变化：Galley开始分担 Pilot/Mixer 的职责。

在 Istio 1.1 版本之前的设计中，Istio的三大组件 Pilot/Mixer/Citadel 都需要访问 kubernetes 的 API Server，以获取服务注册信息和配置信息，包括kubernetes原生资源如service/deployment/pod等，还有 Istio 的自定义资源（数量多达50多个的 CRD）。这个设计导致Istio的各个组件都不得不和 kubernetes 的 API Server产生强绑定，不仅仅代码大量冗余，而且在测试中也因为需要和kubernetes 的 API Server 交互导致 Pilot/Mixer 模块测试困难。

为了解决这个问题，在 Istio 1.1 之后，访问 kubernetes 的 API Server 的工作将逐渐交给 Galley 组件，而其他组件如 Pilot/Mixer 就会和 kubernetes 解耦。

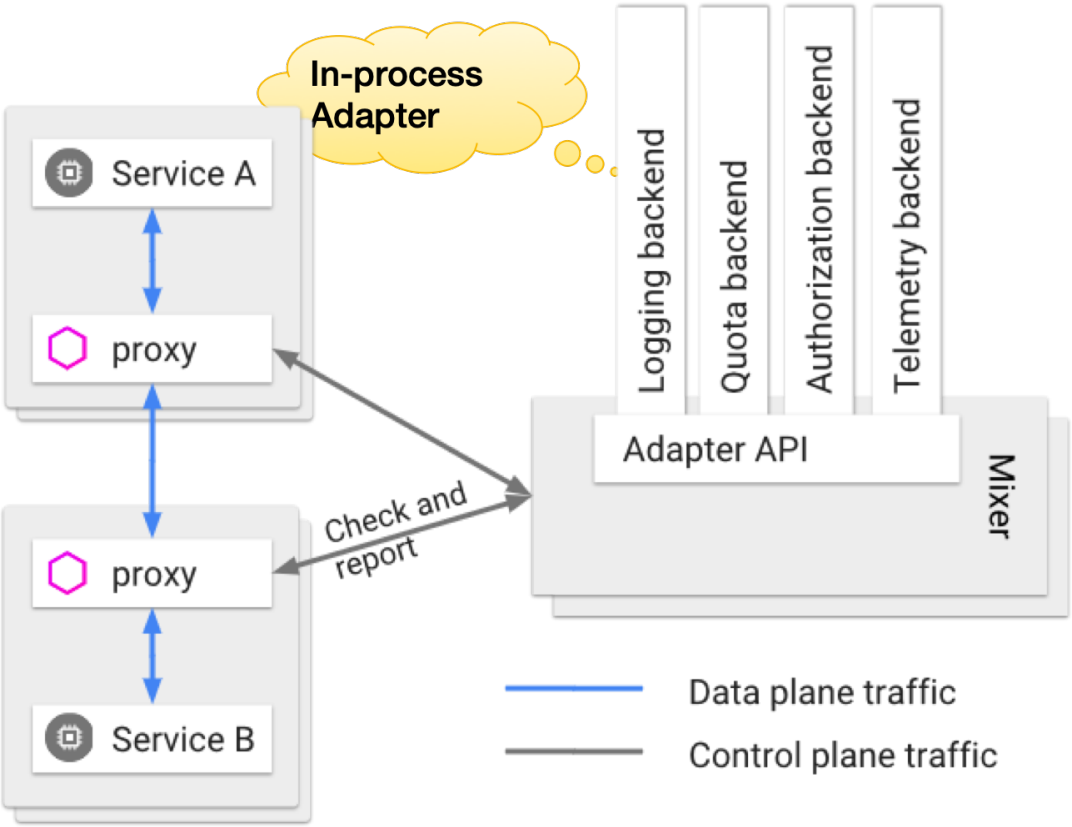


这个工作还在进行中，目前 Istio 的CRD 已经修改为由 Galley 读取，而 K8s 的原生资源（Service/Deployment/Pod等），暂时还是由 Pilot 读取。

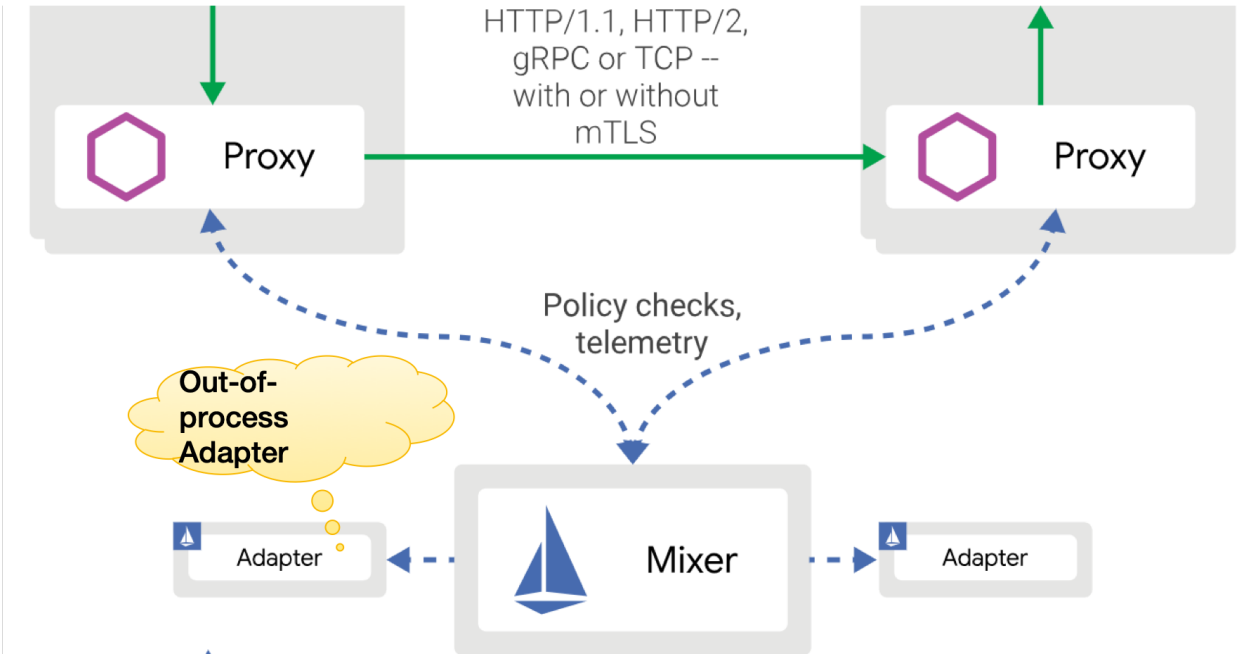
为了方便在各个组件中同步数据，Istio 引入了MCP（Mesh Configuration Protocol）协议。在 Istio 1.1 版本中，Pilot 通过MCP协议从 Galley 同步数据。MCP是受 xDS v2 协议（准确说是 aDS）的启发而制定的新协议，用于在Istio各模块之间同步数据。

Istio 1.1的第二个架构变化来自于 Mixer，在 Istio 1.1 版本中，推荐使用 Out-of-Process Adapter，即进程外适配器。Istio预计下一个版本将弃用 In-Proxy Adapter，目前所有的 Adapter 都将改为 Out-of-Process adapter。

什么是In-Proxy Adapter？下图是 Mixer 的架构图，在 Istio 的设计中，Mixer 是一个独立进程，Proxy 通过远程调用来和 Mixer 交互。而 Mixer 的实现了 Adapter 模式，定义了 Adapter API，然后内建了数量非常多的各种Adapter。这些Adatper的代码存放在 Mixer 代码中，运行时也在 Mixer 的进程内，因此称为 In-Proxy Adapter。



In-Proxy Adapter 的问题在于所有的 Adapter 的实现都和 mixer 直接绑定，包括代码和运行时。因此当 Adapter 需要更新时就需要更新整个 Mixer，任意一个 Adapter 的实现出现问题也会影响整个 Mixer，而且数量众多的Adapter也带来了数量众多的CRD。为此，Istio 1.1 版本中通过引入 Out-of-Process Adapter 来解决这个问题。



Out-of-Process Adapter以独立进程的方式运行在Mixer进程之外，因此 Out-of-Process Adapter 的开发/部署和配置都可以独立与 Mixer，从而将Mixer从adaper的细节实现中解脱出来。

但是，Out-of-Process Adapter的引入，会导致新的性能问题：原来 Mixer 和 In-Proxy Adapter 之间是方法调用，现在 改成 Out-of-Process Adapter 之后就变成远程调用了。而Mixer一直以来都是Istio架构设计中最大的争议，之前 Proxy 和 Mixer 之间的远程调用，已经造成非常大的性能瓶颈所在，而引入 Out-of-Process Adapter 之后远程调用会从一次会变成多次（每个配置生效的 Out-of-Process Adapter 就意味着一次远程调用），这会让性能雪上加霜。

总结 Out-of-Process Adapter 的引入：**架构更加的优雅，性能更加的糟糕**。

在 Istio 1.1 为了架构而不顾性能的同时，Istio 内部也有其他的声音传出，如正在规划中的 Mixer v2。这个规划最终的决策就是放弃 Mixer 独立进程的想法，改为将 Mixer 的功能合并到 Envoy 中，从而消除 Envoy 和 Mixer 之间远程调用的开销。关于 Mixer 的性能问题和 Mixer 合并的思路，蚂蚁金服在去年六月份开始 SOFAMesh 项目时就有清晰的认识和计划，时隔一年，终于欣喜的看到 Istio 开始正视 Mixer 的架构设计问题并回到正确的方向上。

对此有兴趣的朋友可以通过阅读下面的文章获取更详细的信息（发表于一年前，但是依然有效）：

- 大规模微服务架构下的Service Mesh探索之路: 第二节架构设计中的"合并部分Mixer功能"
- Service Mesh架构反思：数据平面和控制平面的界线该如何划定？
- Mixer Cache: Istio的阿克琉斯之踵?： 系列文章，有两篇
- Istio Mixer Cache工作原理与源码分析: 系列文章，有四篇

目前 Mixer v2 的规划还处于 Review 状态，实现方式尚未有明确决定。如果要合并 Mixer，考虑到目前 Mixer 是基于 Golang 编写，而 Envoy 是基于c++ ，这意味着需要用c++重写所有的 Adapter，工作量巨大，恐怕不是短期之类能够完成的。当然也有另外一个新颖（或者说脑洞大开）的思路：引入 Web Assembly（WASM）。目前 Envoy 在进行支持 Web Assembly 的尝试，如果成功，则通过 Web Assembly 的方式来支持 Mixer Adapter 不失为一个好选择。

其他社区产品动态

最近，CNCF 在筹建 Universal Data Plane API （UDPA/通用数据平面API）工作组，以制定数据平面的标准API，为L4/L7数据平面配置提供事实上的标准。Universal Data Plane API 的创意来自 Envoy，实现为 xDS API。而目前 xDS v2 API 已经是数据平面API的事实标准，这次的 UDPA 会以xDS v2 API 为基础。工作组的初始成员来自包括 Envoy 和 gRPC 项目的代表，蚂蚁金服也在积极参与 UDPA 工作组，目前还处于非常早期的筹备阶段。

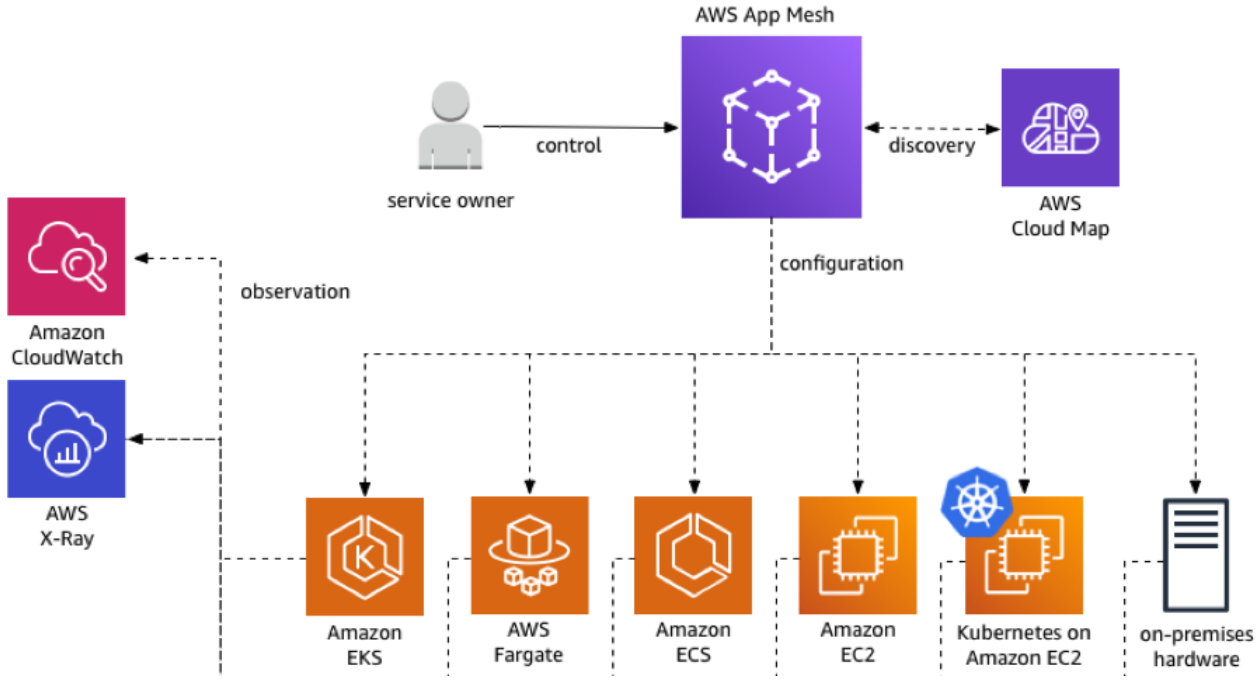
Linkerd2 在2019年4月17日发布了最新的稳定版本 Linkerd 2.3 版本。Linkerd2 是目前开源产品中唯一正面对抗 Istio 的存在，不过在国内知名度不高，使用者也很少。比较有意思的是，Buoyant 最近的B轮融资来自 Google 的投资部门。

云厂商的产品动态

随着 Service Mesh 技术的发展，和各方对 Service Mesh 前景的看好，各大主流云提供商都开始在 Service Mesh 技术上发力。

首先看AWS，在2019年4月，AWS宣布App Mesh GA。App Mesh 是 AWS 推出的AWS原生服务网格，与AWS完全集成，包括：

- 网络（AWS cloud map）
- 计算（Amazon EC2和AWS Fargate）
- 编排工具（AWS EKS，Amazon ECS和EC2上客户管理的k8s）

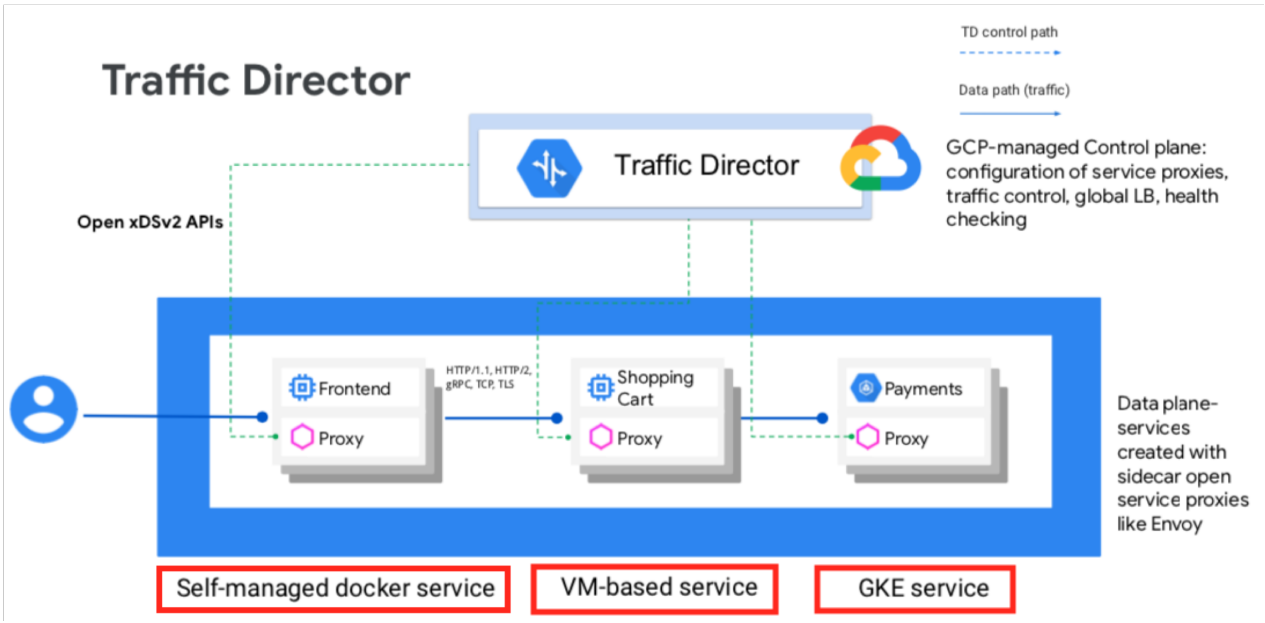


App Mesh的数据平面采用 Envoy，产品非常有创意的选择同时支持VM和容器。

> AWS App Mesh 的更多详细内容，请浏览翻译文章 用AWS App Mesh重新定义应用通讯 。

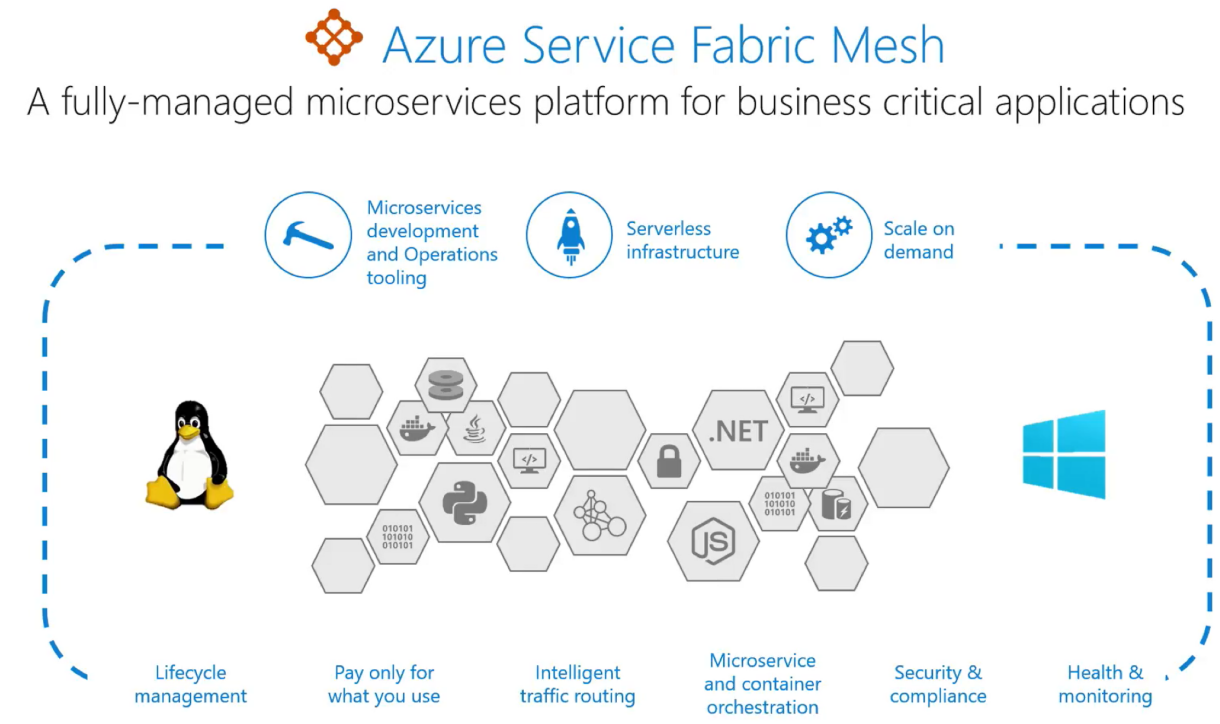
Google 的打法则是围绕 Istio 。首先是在2018年底推出了 Istio on GKE，即"一键集成Istio"，并提供遥测、日志、负载均衡、路由和mTLS 安全能力。接着 Google 又推出 Google Cloud Service Mesh，这是 Istio的完全托管版本，不仅仅提供Istio开源版本的完整特性，还集成了google cloud上的重要产品 stackdriver 。

近期，Google推出 Traffic Director 的 beta 测试版本，Traffic Director 是完全托管的服务网格流量控制平面，支持全局负载均衡，适用于虚拟机和容器，提供混合云和多云支持、集中式健康检查和流量控制，还有一个非常特别的特性：支持基于流量的自动伸缩。



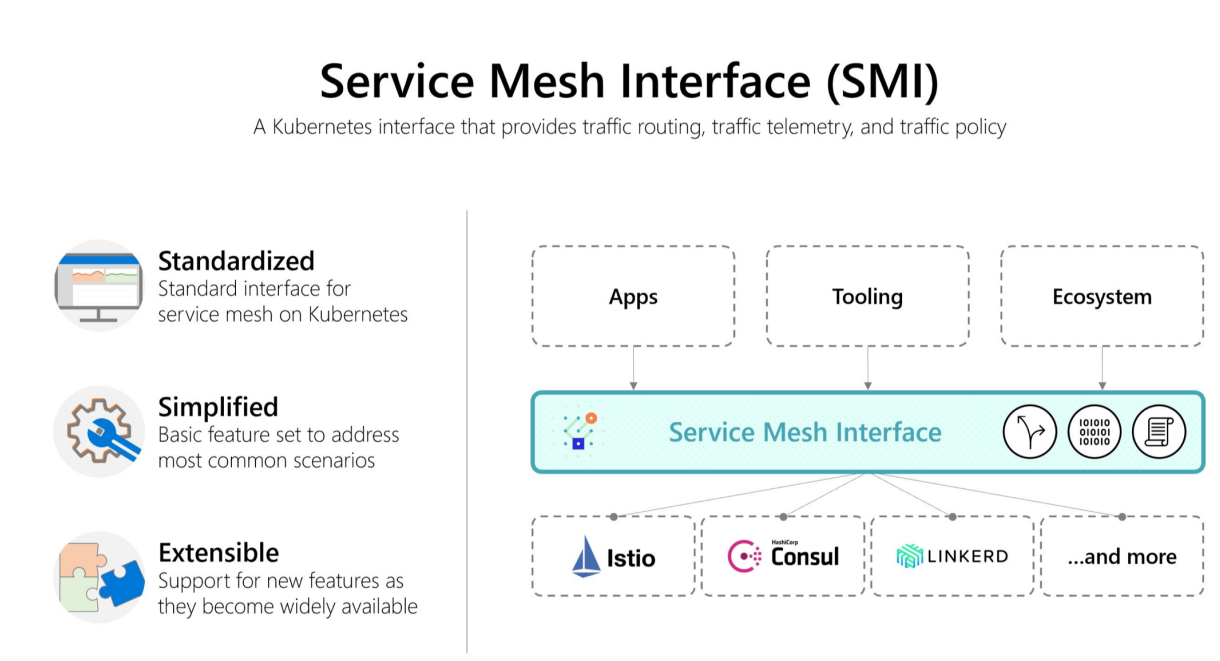
> Google Traffic Director 的详细介绍，请查看我之前的博客文章 Google Traffic Director详细介绍

微软则推出了Service Fabric Mesh。Azure Service Fabric 是Microsoft的微服务框架，设计用于公共云，内部部署以及混合和多云架构。而 Service Fabric Mesh 是Azure完全托管的产品，在2018年8月推出预览版。



上周（5月21号）最新消息，微软在 kubeconf 上推出 Service Mesh Interface。SMI 是在 Kubernetes 上运行服务网格的规范，定义了由各种供应商实现的通用标准，使得最终用户的标准化和服务网格供应商的创新可以两全其美，SMI 预期将为 Service Mesh 带来了灵活性和互通性。

SMI是一个开放项目，由微软，Linkerd，HashiCorp，Solo，Kinvolk和Weaveworks联合启动; 并得到了Aspen Mesh，Canonical，Docker，Pivotal，Rancher，Red Hat和VMware的支持。

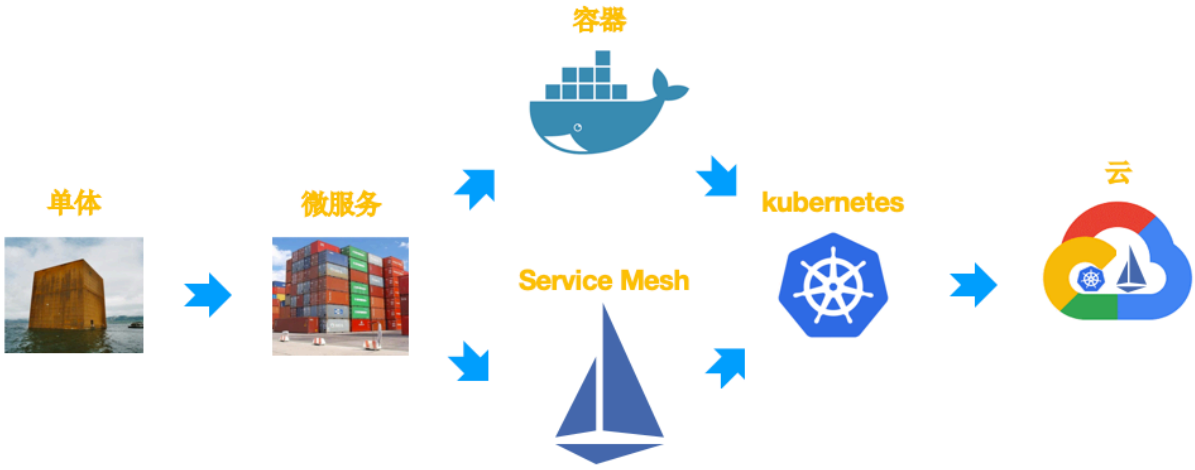


Service Mesh发展趋势

在分享完最近半年 Service Mesh 产品的动态之后，我们来分析探讨 Service Mesh 的发展趋势。

趋势1：上云+托管

在微服务/容器这些年的发展历程中，我们会发现一个很有意思（甚至有些哭笑不得）的事实：



- 为了解决单体的复杂度问题，我们引入微服务架构
- 为了解决微服务架构下大量应用部署的问题，我们引入容器
- 为了解决容器的管理和调度问题，我们引入kubernetes
- 为了解决微服务框架的侵入性问题，我们引入Service Mesh
- 为了让 Service Mesh 有更好的底层支撑，我们又将 Service Mesh 运行在 k8s上

在这个过程中，从单个应用（或者微服务）的角度看，的确自身的复杂度降低，在有底层系统支撑的情况下部署/维护/管理/控制/监控等也都大为简化。但是站在整个系统的角度，整体复杂度并没有消失，只是从单体分解到微服务，从应用下沉到Service Mesh，复杂度从总量上不但没有减少，反而大为增加。

解决这个问题最好的方式就是 **上云**，使用 **托管** 版本的 k8s 和 Service Mesh，从而将底层系统的复杂度交给云厂商，而客户只需要在云的基础上享受 Service Mesh 技术带来的使用便利和强大功能。

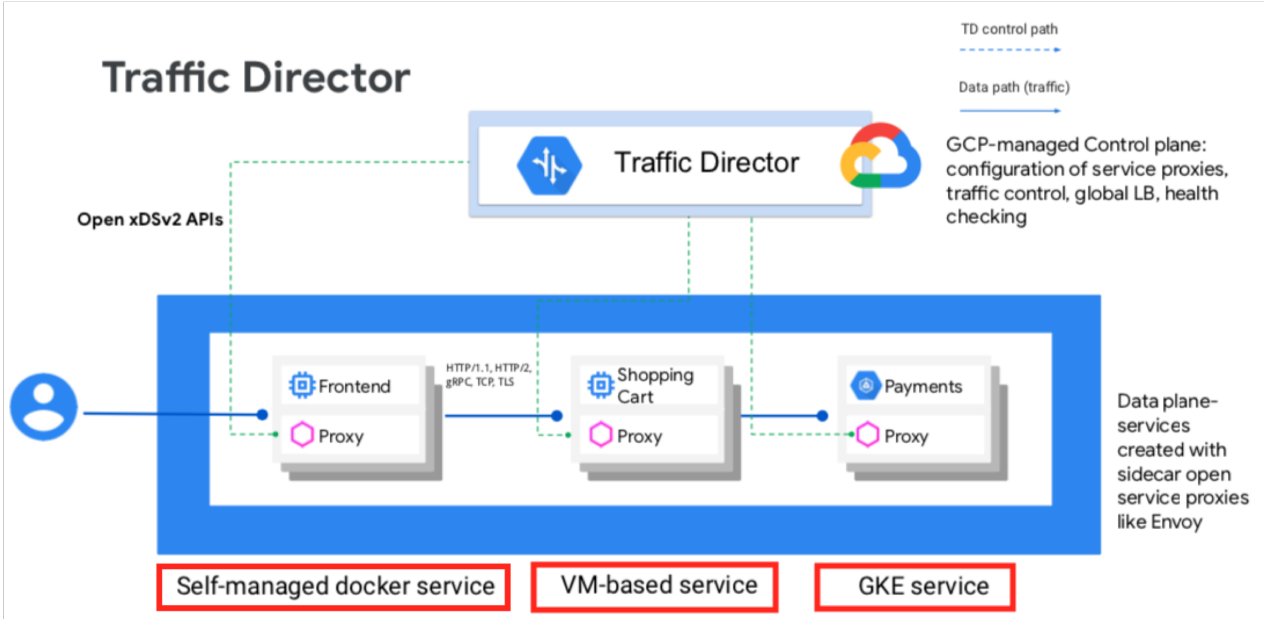
前面我们分享产品动态时，可以看到目前 Google / AWS / 微软 这三巨头都已经推出了各自的 Service Mesh 托管产品，而在国内，阿里云/华为云等也有类似的产品推出，我们蚂蚁金服也将在稍后在金融云上推出 SOFAMesh 的云上托管版本。在这里，我总结为一句话：

几乎所有的主要公有云提供商都在提供（或者准备提供）Service Mesh托管方案

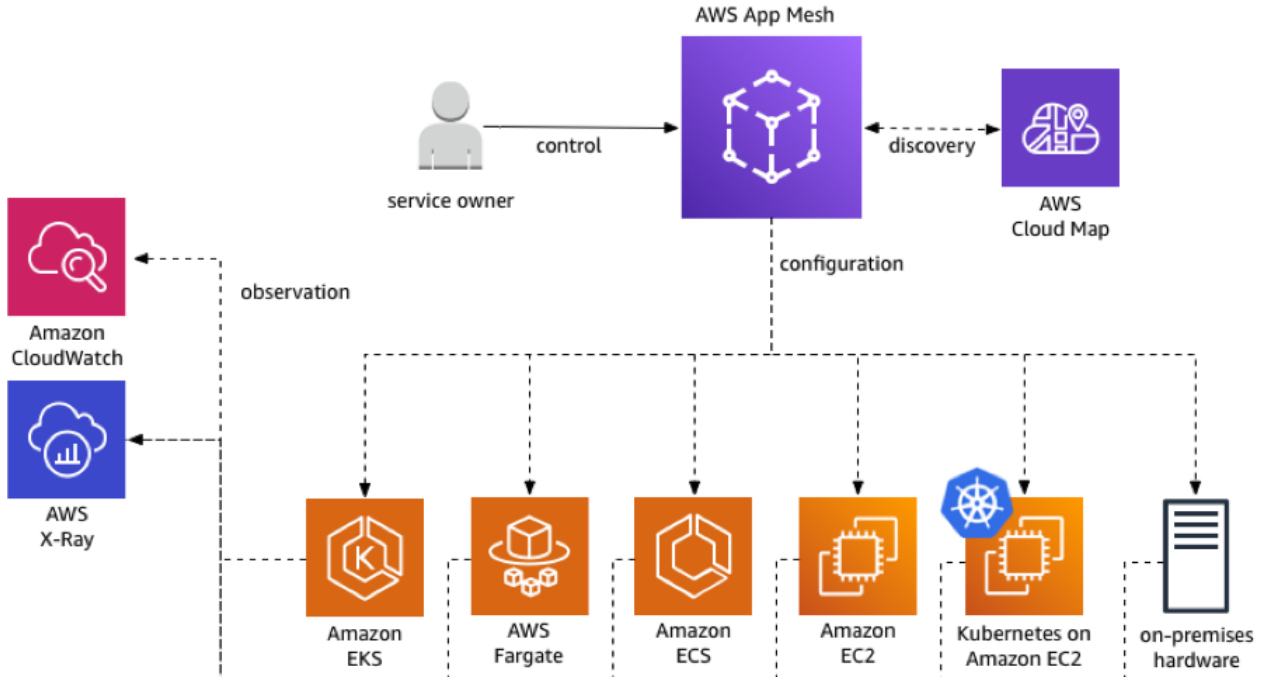
趋势2：VM和容器混用

第二个趋势就是VM和容器混用，即 Service Mesh 对服务的运行环境的支持，即支持容器（尤其指 k8s），也支持虚拟机，而且支持运行在这两个环境下的服务相互访问，甚至直接在产品层面上屏蔽两者的差异。

比如 Google 的 Traffic Director 产品：



AWS 的 App Mesh产品：

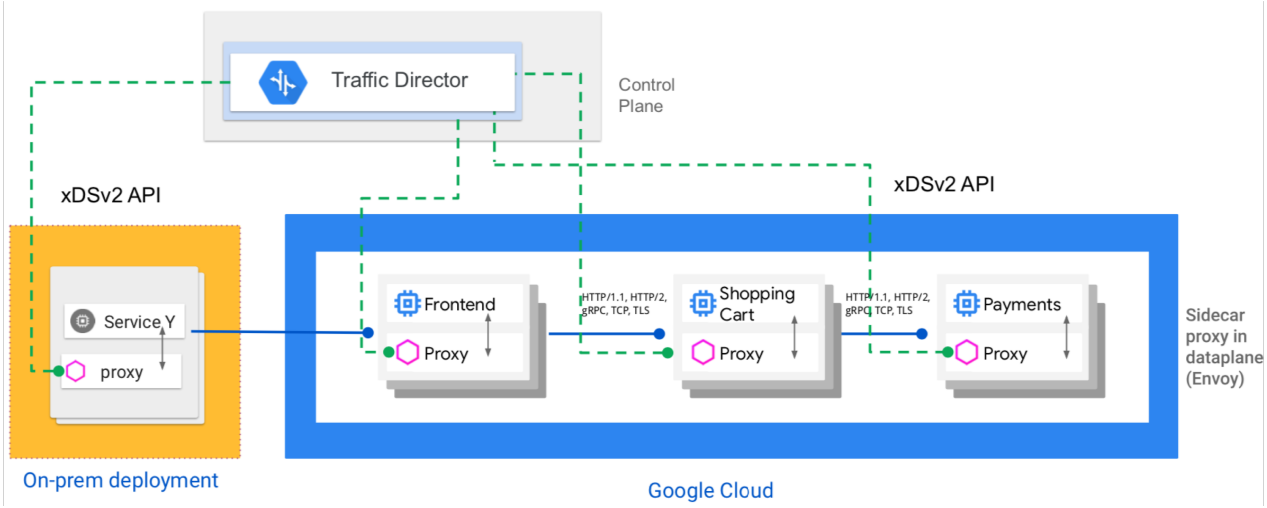


都是在产品层面直接提供VM和容器混用的支持，不管应用是运行在vm上还是容器内都可以支持，而且可以方便的迁移。

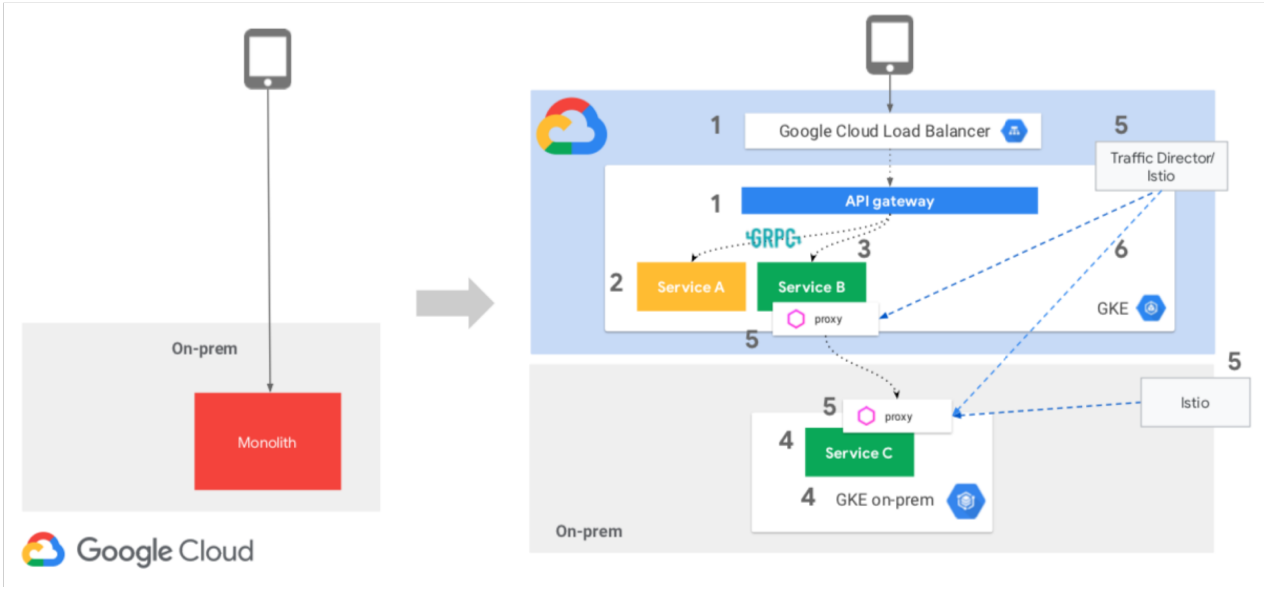
趋势3：混合云和多云支持

混合云和多云支持最近正成为一个新的技术热点和商业模式，甚至 Google Cloud 都喊出口号，要 "All in Hybrid Cloud"！

Google Traffic Director 旗帜鲜明的表达了 Google Cloud 对混合云的重视：



下图是 Google Traffic Director 给出的一个应用改造示例，从单体结构转为微服务架构，从私有云转为公有云加私有云的混合云模式。



Service Mesh 毫无疑问是实现混合云和多云支持的一个非常理想的技术方案。

趋势4：和 Serverless 的结合

Service Mesh 技术和 Serverless 技术是工作在不同纬度的两个技术：

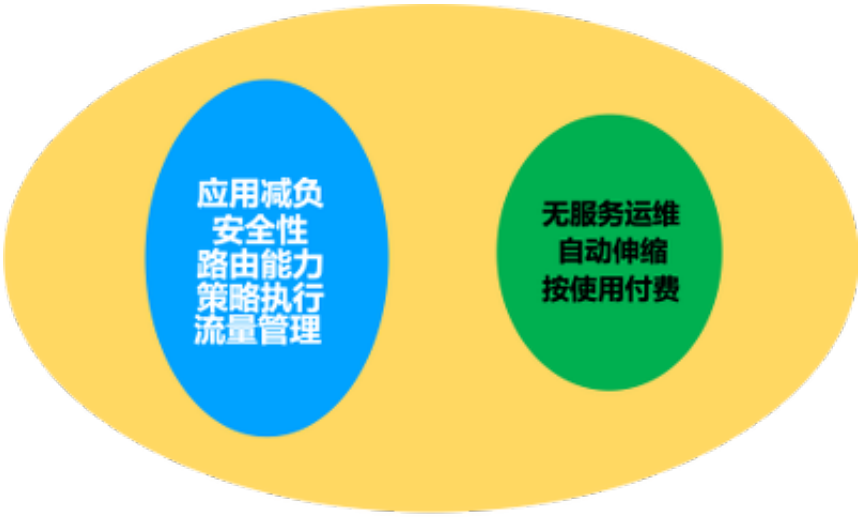
- Service Mesh技术的关注点在于**服务间通讯**，其目标是剥离客户端SDK，为应用减负，提供的能力主要包括安全性、路由、策略执行、流量管理等。
- Serverless 技术的关注点在于**服务运维**，目标是客户无需关注服务运维，提供服务实例的自动伸缩，以及按照实际使用付费。

理论上 Service Mesh 技术和 Serverless 技术并没有冲突的地方，可以结合使用。事实上目前业界也开始有这个趋势，而融合的方式有两种：

1. 在Serverless中引入Servicemesh：典型如 knative 项目和 knative 的 Google Cloud 托管版本 Google Cloud Run，通过引入对容器的支持和使用 Istio，knative 将 Serverless 的支持扩展到 Function 之外，在极大的扩展 Serverless 适用范围的前提下，也将服务间通讯的能力引入到 Serverless。
2. 在Servicemesh中引入 Serverless：典型如 Google Traffic Director 产品，在提供 Service Mesh

各种能力的同时，支持按照流量自动伸缩服务的实例数量，从而融入了部分 serverless 的特性。

对于 Serverless 和 Servicemesh 的结合，我们来展望未来形态：未来应该会出现一种新型服务模式，Serverless 和 Servicemesh 合二为一。只要将服务部署上来，就自动可以得到 Servicemesh 的服务间通讯能力和 Serverless的无服务器运维。在蚂蚁金服，我们将这理解成为是未来云原生应用的终态之一，正在积极的探索其落地的实际方式。



服务间通讯：ServiceMesh
服务运维：Serverless

趋势5：Mesh模式延伸

回顾一下 Service Mesh 模式的核心，其基本原理在于将客户端SDK剥离，以 Proxy 独立进程运行；目标是将原来存在于SDK中的各种能力下沉，为应用减负，以帮助应用云原生化。

遵循这个思路，将 Service Mesh 的应用场景泛化，不局限于服务间的同步通信，就可以推广到更多的场景：特征是有网络访问，而是通过客户端SDK来实现。

在蚂蚁金服的实践中，我们发现Mesh模式不仅仅适用于服务间同步通讯，也可以延伸到以下场景：

- Database Mesh: 数据库访问
- Message Mesh：消息机制
- Cache Mesh：缓存

以上模式的产品蚂蚁金服都在探索中，相关的产品正在开发和尝试落地。社区也有一些相关的产品，比如 Database Mesh 方面张亮同学在力推的 Apache Shardingsphere 项目。

通过更多的 Mesh 模式，我们可以覆盖更多的场景，从而实现让应用在各个方面都做到减负，而不仅仅是 Service Mesh 针对的服务间通讯，从而为后续的应用云原生化奠定基础。

趋势6：标准化，不锁定

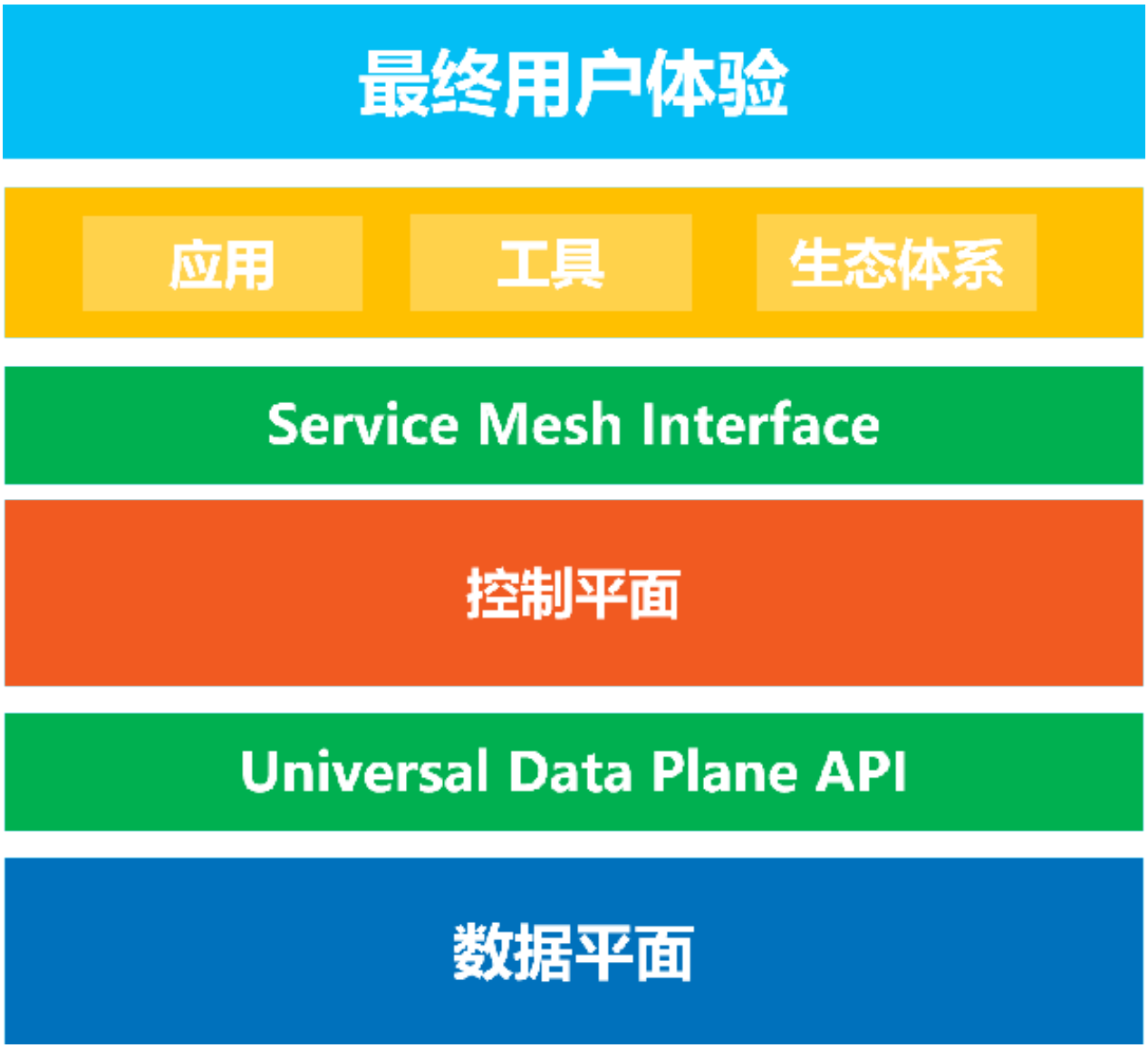
云原生的一个重要主张，就是希望在云上为用户提供一致的用户体验，提倡标准化，避免供应商绑定（Not Lock-In）。

从前面分享的 Service Mesh 产品动态可以看出，目前 Service Mesh 市场上出现了众多的供应商和产品：开源的，闭源的，大公司出的，小公司出的，市场繁荣的同时也带来了市场碎片化的问题——所有围绕业务应用的外围工作，比如通过 Service Mesh 对流量进行控制，配置各种安全/监控/策略等行为，以及在这些需求上建立起来的工具和生态系统，却不得不牢牢的绑死在某个具体的 Service Mesh 实现上，所谓"供应商锁定"。其根本问题在于各家实现不同，又没有统一标准。因此，要想解决上述问题，就必须釜底抽薪：**解决 Service Mesh 的标准化问题**。

就在最近这一个月，Service Mesh 社区出现了两个推动标准化的大事件：

1. CNCF筹建 Universal Data Plane API （通用数据平面API）工作组，计划以 xDS v2 API 为基础制定数据平面的标准API，工作组的初始成员来自包括 Envoy 和 gRPC 项目的代表（可以理解为 Google 为首）
2. 微软在 kubeconf 上推出 Service Mesh Interface，准备定义在 Kubernetes 上运行服务网格的规范，为 Service Mesh 带来了灵活性和互通性。SMI由微软牵头，联合 Linkerd，HashiCorp，Solo，Kinvolk和Weaveworks。

为了方便理解这两个标准，我为大家准备了一张图：

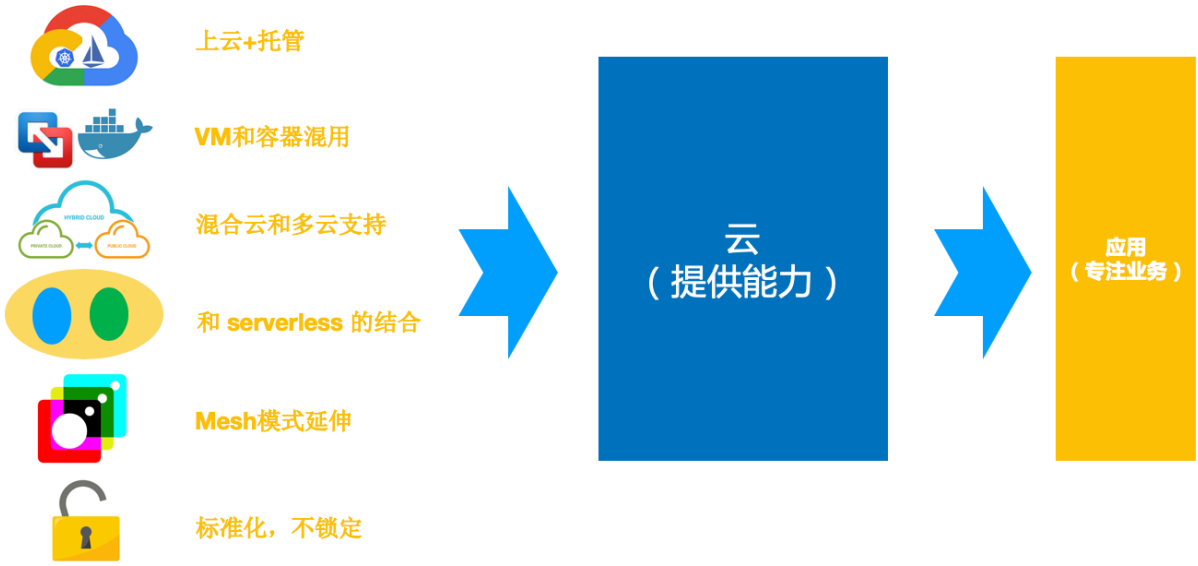


其中，Universal Data Plane API 是数据平面的标准，控制平面通过这个API来控制数据平面的行为。而 Service Mesh Interface 是控制平面的标准，上层的应用/工具/生态体系通过 Service Mesh Interface 来实现跨不同的Service Mesh实现为最终用户提供一致性的体验。

当然这两个标准化API都刚刚起步，而且，标准化的工作通常不仅仅是技术问题，涉及到复杂的利益关系，具体未来走向现在难于推断，只能密切关注。

发展趋势分析

我们总结一下上面列出的 Service Mesh 最近的6个发展趋势：



这些趋势都和云有关，核心在于让云来提供能力，包括：

- 让云承担更多职责
- 提供更高抽象
- 适用更多场景
- 减少应用负担：实现应用轻量化

最终实现让业务应用**专注业务**的战略目标。

对于 Service Mesh 技术未来的走向，我的看法是：Service Mesh 技术必然不是孤立的自行发展，而是在云原生的大环境下，与云原生的其他技术、理念、最佳实践一起相互影响，相互促进，相互提供，共同发展。云原生是一个庞大的技术体系，Service Mesh 需要在这个体系中获得各种支撑和配合，才能最大限度的发挥自身的优势。

Service Mesh与云原生

在最后一段，我们来谈谈 Service Mesh 技术和云原生的关系，也就是本次分享的标题所说的：云原生中流砥柱。

凭什么？

什么是云原生?

在解释之前，首先问另外一个问题：什么是云原生？相信这个问题很多同学都问过，或者被问过，每个人心里可能都有自己的理解和表述。在今年年初，我也特意就这个问题问了自己，然后尝试着给出了一个我的答案：

云原生指 "原生为云设计"，具体说就是：**应用原生被设计为在云上以最佳方式运行，充分发挥云的优势。**

关于云原生的理解，以及对这句话的详细阐述，这里不详细展开，有兴趣的同学可以浏览我之前的演讲内容，讲的比较深入，厚颜自荐一下：

- 畅谈云原生（上）：如何理解云原生？云原生应用应该是什么样子？云原生下的中间件该如何发展？
- 畅谈云原生（下）：云和应用该如何衔接？如何让产品更符合云原生？

Service Mesh的核心价值

关于Service Mesh的核心价值，我个人的理解，不在于 Service Mesh 提供的玲琅满目的各种功能和特性，而是：

实现业务逻辑和非业务逻辑的分离

将功能实现，从客户端SDK中剥离出来，放到独立的Proxy进程中，这是 Service Mesh 在技术实现上走出的第一步，也是至关重要的第一步：因为这一步，实现了**业务逻辑**和**非业务逻辑**的分离，而且是最彻底的物理分离，哪怕需要付出一次远程调用的代价。

而这一步迈出之后，前面就是海阔天空：

- 业务逻辑和非业务逻辑分离之后，我们就可以将这些非业务逻辑继续下沉
- 下沉到基础设施，基础设施可以是基于VM的，可以是基于容器和k8s的；也可以是VM和容器混合
- 基础设施也可以以云的形式提供，可以是公有云、私有云，也可以是混合云、多云；
- 可以选择云上托管，完全托管也好，部分托管也好，产品形态可以很灵活

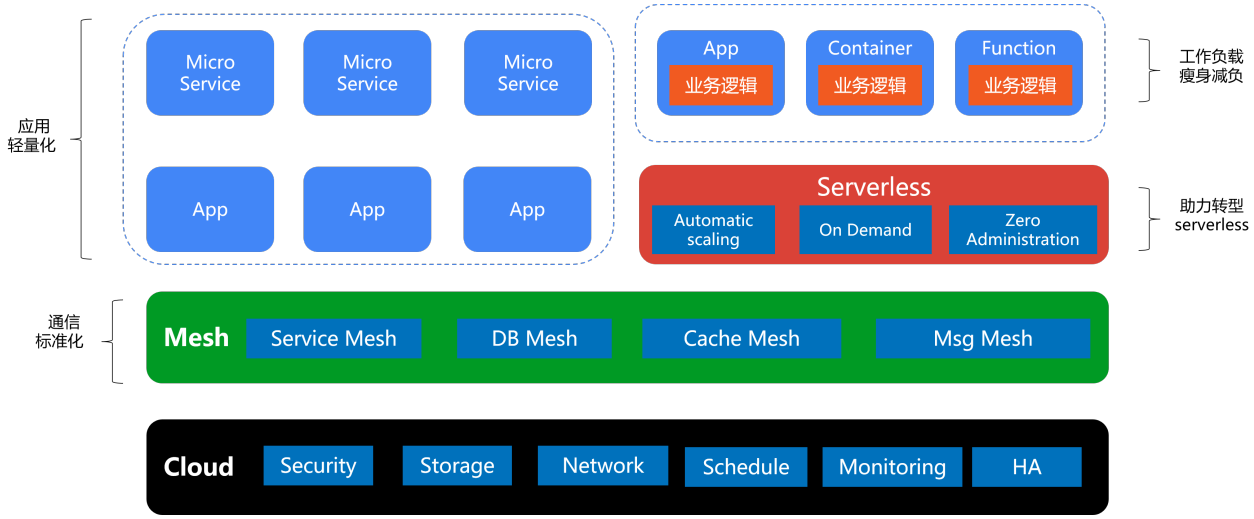
总结说，业务逻辑和非业务逻辑的分离：

- 为下沉到基础设施提供可能
- 为上云提供可能
- 为应用轻量化提供可能

> 备注：这里说的上云，指的是上云原生(Cloud Native)的云，而不是上云就绪(Cloud Ready)的云。

Mesh化是云原生落地的关键步骤

在过去一年中，蚂蚁金服一直在努力探索云原生落地的方式，在这个过程中，我们有一些感悟，其中非常重要的一条就是：Mesh化是云原生落地的关键步骤。

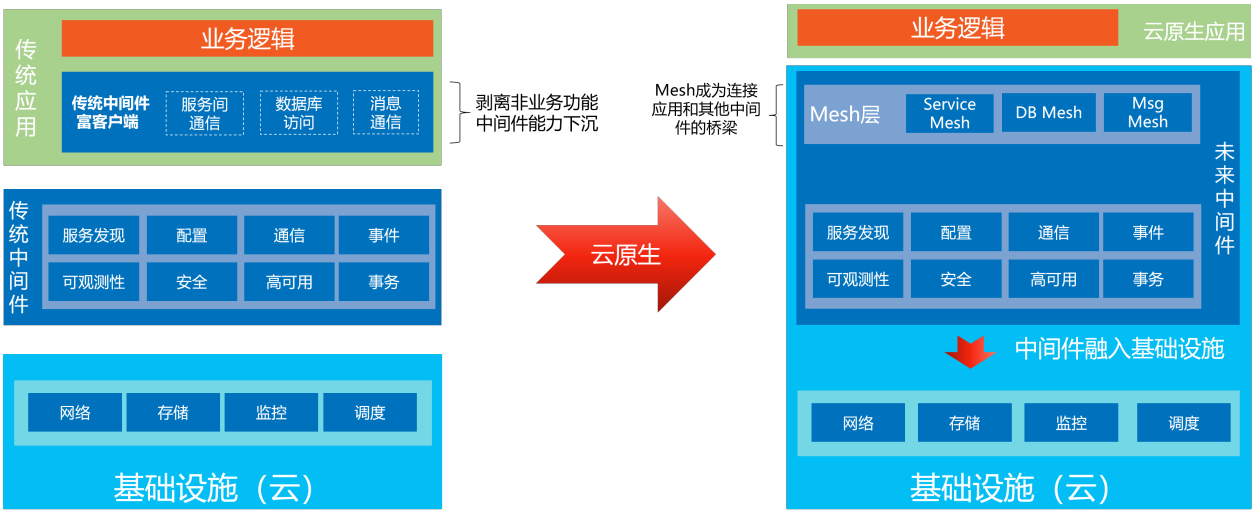


如上图所示：

- 最下方是云，基于k8s和容器打造，提供各种基础能力，这些能力有一部分来自传统中间件的下沉
- 在云上是 Mesh 层，包含 Service Mesh 以及我们前面提到的各种扩展的Mesh模式，实现了通信的标准化
- 在通过 Mesh 剥离非业务功能并下沉之后，应用实现了轻量化，传统的App和新兴的微服务都可以受益于此
- 更进一步，轻量化之后的业务应用，其工作负载在瘦身减负之后变得相当的干净，基本只剩业务逻辑，包括传统的App，以Container形式运行的服务和新颖的Function，这些负载在往 serverless 形态转换时相对要轻松很多

配合 Serverless 技术领域最新的技术潮流和产品发展（如以 knative 项目为代表，Serverless 不再仅仅是 Function 形式，也支持 BaaS 等偏传统的工作负载），Mesh化为现有应用转型为 Serverless 模式提供助力。

在这里我们再分享一下蚂蚁金服对未来中间件产品发展的感悟，我们认为中间件的未来在于**Mesh化，并融入基础设施**，如下图所示：



左边是传统的中间件形态，在云原生时代，我们希望将非业务功能从传统中间件的富客户端中剥离出来，然后将这些能力以及这些能力背后的中间件能力，下沉到基础设施，下沉到云。而中间件产品，也会由此融入基础设施，如图中右边所示。未来的中间件将作为基础设施和云的一部分，而 Mesh 则成为连接应用和基础设施以及其他中间件产品的桥梁。

更重要的是：业务应用因此而实现轻量化，在剥离各种非业务功能之后，业务应用就实现了关注业务逻辑的战略目标，从而实现从传统应用到云原生应用的转型。

总结：通过 Service Mesh 技术，我们实现了业务逻辑和非业务逻辑的分离，为应用的轻量化和云原生化提供可能；并通过将非业务逻辑的各种功能下沉到基础设施和云，极大的增强了基础设施和云的能力，为云原生的落地提供了极大助力。

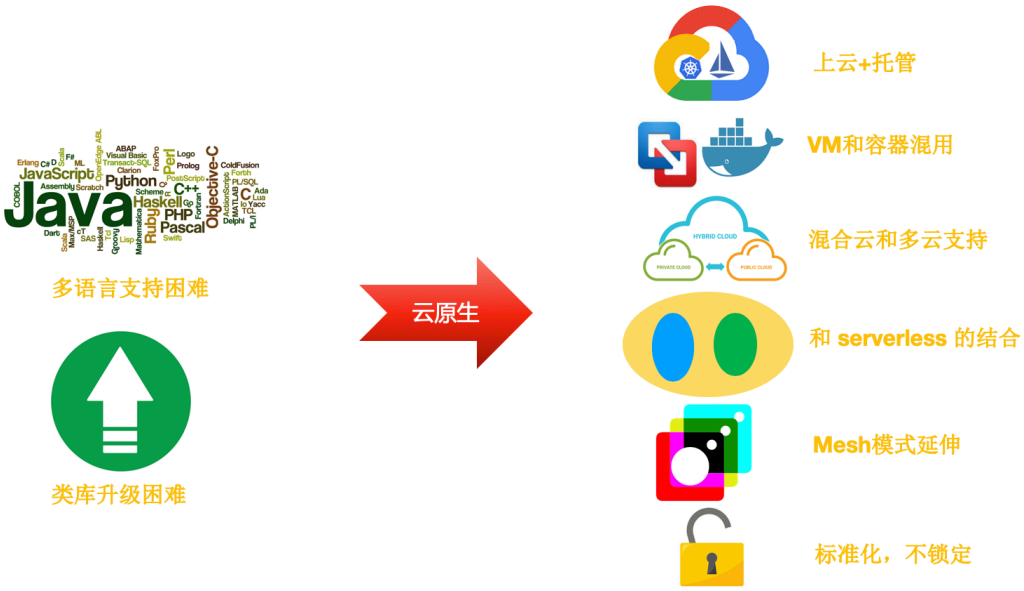
因此，我们认为： Service Mesh技术在云原生落地中扮演了非常重要的作用，不可或缺。

Service Mesh发展展望

最后再次展望一下 Service Mesh 的未来发展。

左边是 Service Mesh 发展初期的最重要的两个原始动力：**多语言支持**和**类库升级**。关于这两点，最近这两年中介绍和推广 Service Mesh 理念和产品的同学基本都讲过，但是今天我想指出的是：这只是 Service Mesh 的**起点**，而远不是 Service Mesh 的终点。

Service Mesh 的未来，不会停留在仅仅满足多语言支持和类库升级，而是跟随云原生的大潮，解决各种实际需求，同时又尽量维持上层业务应用的简单直白。



在这次分享的最后，我想给大家一个留一个课外作业，有心的同学可以尝试一下：如果您想更深入的理解 Service Mesh 的价值，想对 Service Mesh 未来的发展方向有更清晰的认识，尤其是希望能通过自身的思考和感悟来理解 Service Mesh 而不是简单的被灌输（包括被我灌输），那么请尝试独立的做如下思考：

1. 抛开左边的这两点，不要将思维局限在这个范围内
2. 考虑云原生的大背景，结合您自身对云原生的理解，和对云的期望
3. 针对右边的 Service Mesh 的六个趋势，忘记我前面讲述的内容，只考虑其背后的实际场景和客户需求，以及这个场景带来的业务价值，然后认真对比使用 Service Mesh 和不使用 Service Mesh 两种情况下的解决方案
4. 请在上述思考的过程中，更多的从业务应用的角度来看待问题，假设你是那个云上的应用（还记得前面图上的小baby吗？），你会希望被如何对待

如果您有所收获，欢迎和我联系。

尾声

最后做个广告，欢迎大家参加 SOFAStack 云原生工作坊，我们将在这次活动中，推出蚂蚁金服金融云完全托管版本的SOFAMesh的体验版本，欢迎报名参加。我将在上海 kubeconf 的 workshop 现场恭候大家。

提示：请使用**钉钉**扫描上面的二维码。或者直接访问 kubeconf的活动页面 查看这次 workshop 的详细内容。