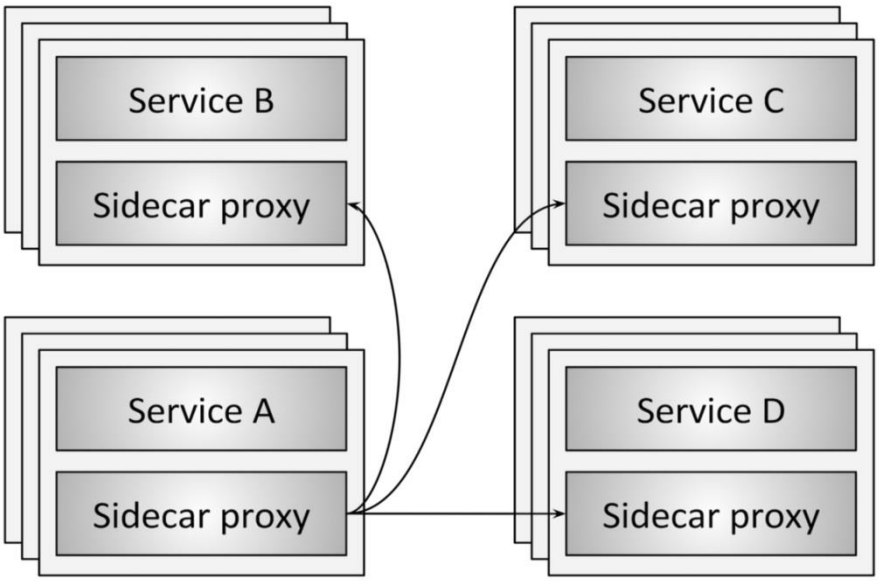
在介绍 service mesh 之前，我们先来看一下下面的图了解一个概念：

这就是吃鸡中的“三蹦子”，学名字“sidecar”，就是边三轮摩托车，即在两轮摩托车的旁边添加一个边车，两轮摩托车的驾驶者集中注意力驾驶，边车上的队友专注周围信息和地图。

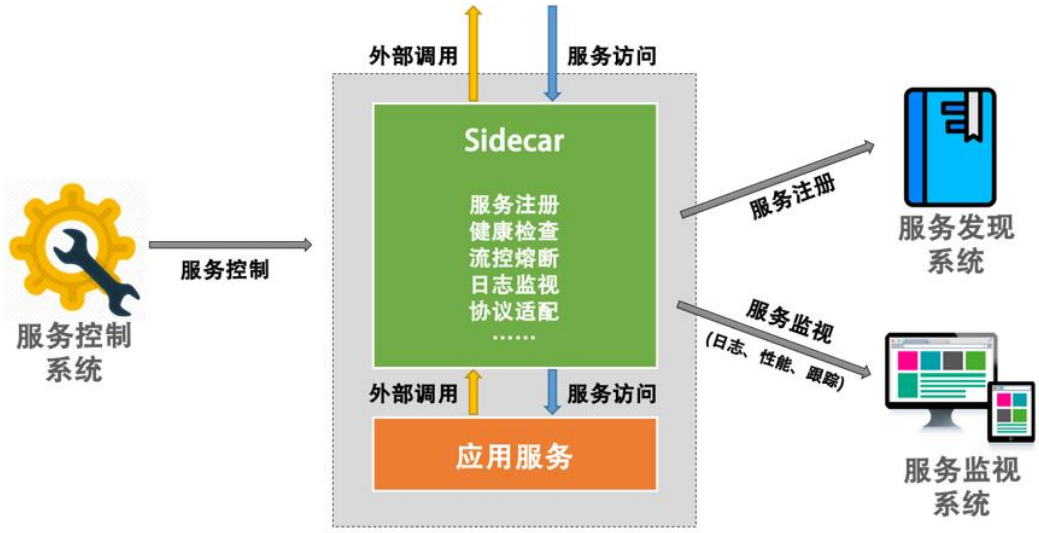
什么是 Service Mesh

Service Mesh 直译过来是“服务网格”，目的是解决系统架构微服务化后的服务间通信和治理问题。服务网格由 Sidecar 节点组成。Sidecar 在软件系统架构中特指“边车模式”。这个模式的灵感来源于最开始介绍的边三轮，它的精髓在于实现了数据面（业务逻辑）和控制面的解耦。具体到微服务架构中，即给每一个微服务实例（也可以是每个宿主机 host）同步部署一个 sidecar proxy:

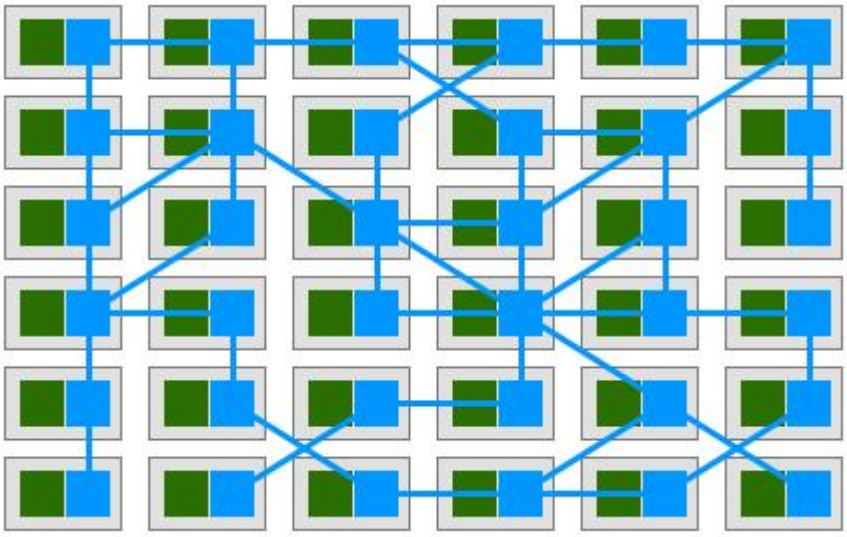


该 sidecar proxy 负责接管对应服务的入流量和出流量。并将微服务架构中以前有公共库、

framework 实现的熔断、限流、降级、服务发现、调用链分布式跟踪以及立体监控等功能从服务中抽离到该 proxy 中，服务对于代理无感知，且服务间所有通信都由代理进行路由：



当该 sidecar 在微服务中大量部署时，这些 sidecar 节点自然就形成了一个网格：



图中绿色方块为服务，蓝色方块为边车部署的服务网格，蓝色线条为服务间通讯。
这就是 service mesh，专注于处理服务和服务间的通讯。其主要负责构造一个稳定可靠的服务通讯的基础设施，并让整个架构更为的先进和 Cloud Native。在工程中，Service Mesh 基本来说是一组轻量级的与应用逻辑服务部署在一起的服务代理，并且对于应用服务是透明的。

Service Mesh 的特点

- 是一个基础设施
- 轻量级网络代理，应用程序间通讯的中间层
- 应用程序无感知，对应用程序透明无侵入
- 解耦应用程序的重试/超时、监控、追踪和服务发现等控制层面的东西

Service Mesh 有哪些开源实现

Service Mesh 的概念从 2016 年提出至今，已经发展到了第二代。

- 第一代

Linkerd（https://github.com/linkerd/linkerd）2016 年 1 月 15 日首发布，业界第一个 Service Mesh 项目，由 Buoyant 创业小公司开发（前 Twitter 工程师），2017 年 7 月 11 日，宣布和 Istio 集成，成为 Istio 的数据面板。

Envoy（https://github.com/envoyproxy/envoy）2016 年 9 月 13 日首发布，由 Matt Klein 个人开发（Lyft 工程师），之后默默发展，版本较稳定。

Linkerd 使用 Scala 编写，是业界第一个开源的 service mesh 方案。Envoy 基于 C++ 11 编写，无论是理论上还是实际上，后者性能都比 Linkerd 更好。这两个开源实现都是以 sidecar 为核心，绝大部分关注点都是如何做好 proxy，并完成一些通用控制面的功能。但是，当你在容器中大量部署 sidecar 以后，如何管理和控制这些 sidecar 本身就是一个不小的挑战。于是，

- 第二代

Istio（https://github.com/istio/istio）2017 年 5 月 24 日首发布，由 Google、IBM 和 Lyft 联合开发，只支持 Kubernetes 平台，2017 年 11 月 30 日发布 0.3 版本，开始支持非 Kubernetes 平台，之后稳定的开发和发布。

Conduit（https://github.com/runconduit/conduit）2017 年 12 月 5 日首发布，由 Buoyant 公司开发（借鉴 Istio 整体架构，部分进行了优化）。

nginMesh（https://github.com/nginmesh/nginmesh）2017 年 9 月首发布，由 Nginx 开发，定位是作为 Istio 的服务代理，也就是替代 Envoy，思路跟 Linkerd 之前和 Istio 集成很相似，极度低调。

Kong（https://github.com/Kong/kong）比 nginMesh 更加低调，默默发展中。

第二代 service mesh 主要改进集中在更加强大的控制面功能（与之对应的 sidecar proxy 被称之为数据面），典型代表有 Istio 和 Conduit。

ISTIO 介绍

Istio：来自希腊语，英文意思是"Sail"， 翻译为中文是"启航"。官方定义为：一个连接，管理和保护微服务的开放平台。

简单的说，有了 Istio，你的服务就不再需要任何微服务开发框架（典型如 Spring Cloud、Dubbo），也不再需要自己动手实现各种复杂的服务治理的功能（很多是 Spring Cloud 和 Dubbo 也不能提供的，需要自己动手）。只要服务的客户端和服务器可以进行简单的直接网络访问，就可以通过将网络层委托给 Istio，从而获得一系列的完备功能。

可以近似的理解为：Istio = 微服务框架 + 服务治理

Istio 能做什么

流量管理、可观察性、策略执行、服务身份和安全、平台支持、集成和定制……

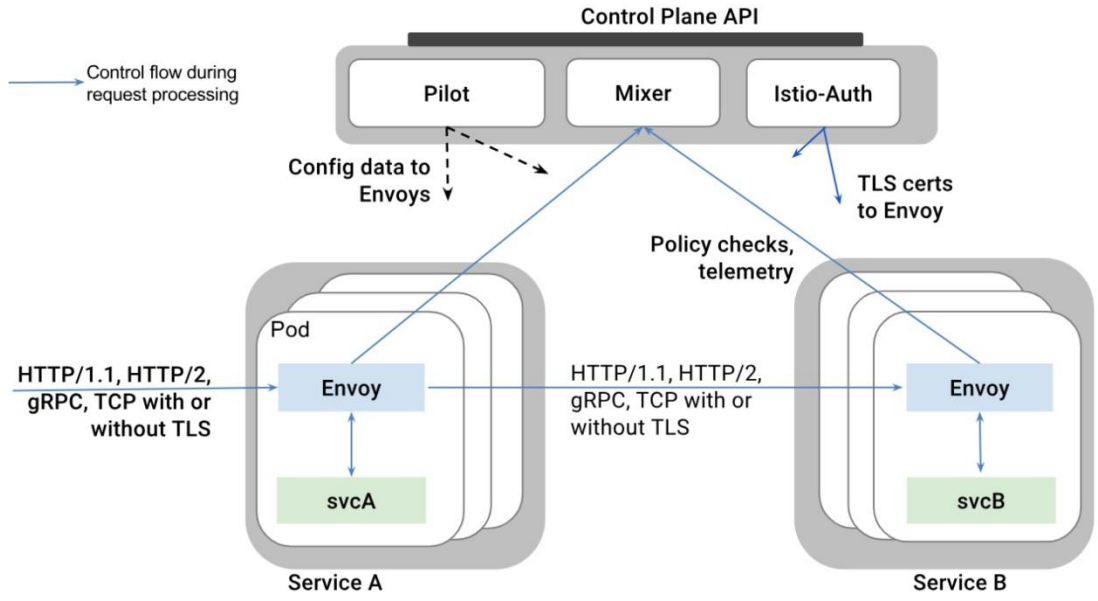
理论上说，任何微服务框架，只要愿意往上面堆功能，早晚都可以实现这些。那，关键在哪？假如在没有 Istio 这样的服务网格的情况下，如何开发微服务才可以做到前面列出的这些丰富多彩的功能？无外乎，找个 Spring Cloud 或者 Dubbo 等成熟的框架。但是，搞定的一大堆问题之后要给个配置说明到运维，告诉他说如何灰度、限速……然而如果有个需求加个黑名单、做个特殊的灰度、将来自 iPhone 的用户流量导 1%到 Stagging 环境的 2.0 新版本……你准备往你的业务程序里面塞多少管理和运维的功能？

Istio 超越 spring cloud 和 dubbo 等传统开发框架之处，就在于不仅仅带来了远超这些框架所能提供的功能， 而且也不需要应用程序为此做大量的改动，开发人员也不必为上面的功能实现进行大量的知识储备。

Istio 架构

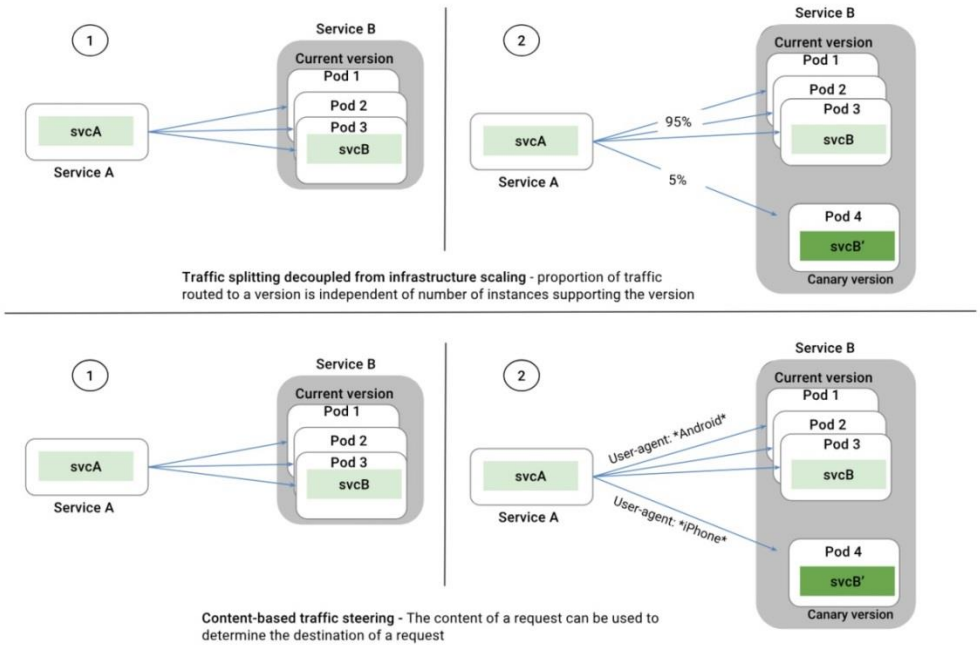
Istio 服务网格分为两大块：数据面板和控制面板。

Istio 中数据面板 (Envoy)扮演的角色是底层干活的民工，而该让这些民工如何工作，由包工头控制面板（Mixer、Pilot 和 Istio-Auth）来负责完成。



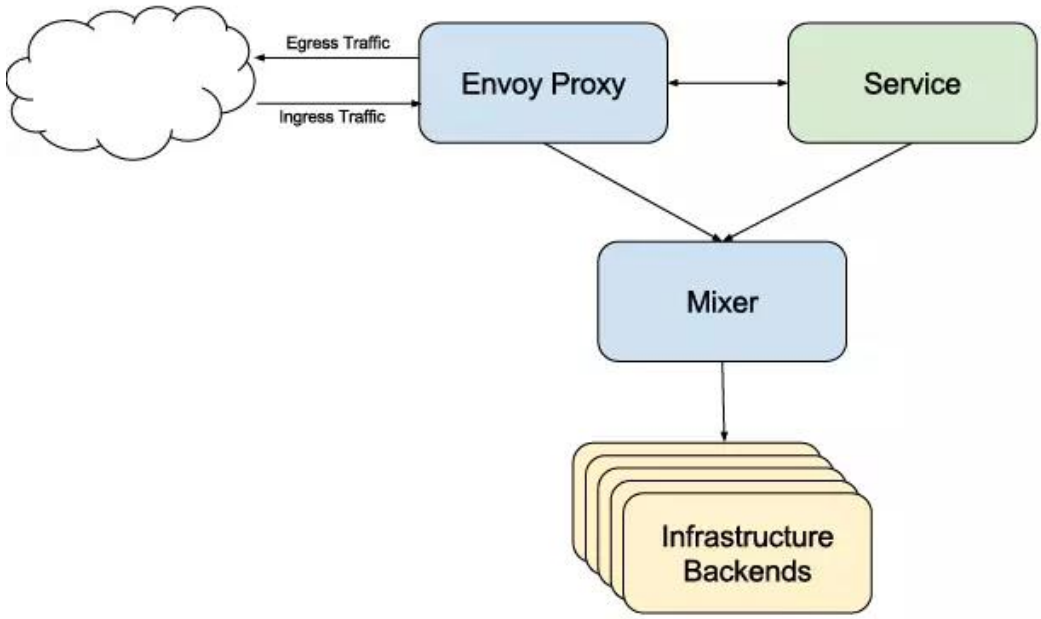
Istio Architecture

- Envoy: 扮演 sidecar 的功能，协调服务网格中所有服务的出入站流量，并提供服务发现、负载均衡、限流熔断等能力，还可以收集大量与流量相关的性能指标。
- Pilot: 负责部署在 service mesh 中的 Envoy 实例的生命周期管理。本质上是负责流量管理和控制，是将流量和基础设施扩展解耦，并不承载任何业务流量。Pilot 让运维人员通过 Pilot 指定它们希望流量遵循什么规则，因此可以非常容易的实现 A/B 测试和 Canary 测试：

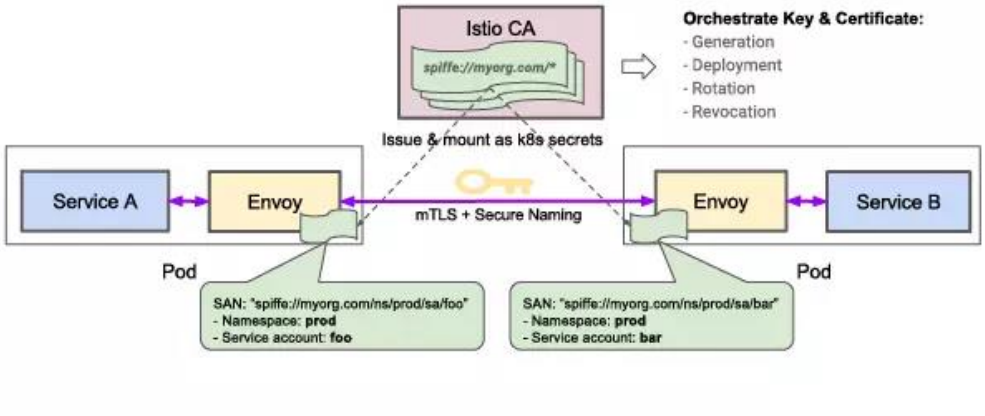


Traffic Management with Istio

- Mixer: Mixer 在应用程序代码和基础架构后端之间提供通用中介层。它的设计将策略决策移出应用层，用运维人员能够控制的配置取代。应用程序代码不再将应用程序代码与特定后端集成在一起，而是与 Mixer 进行简单的集成，然后 Mixer 负责与后端系统连接。Mixer 可以认为是其他后端基础设施（如数据库、监控、日志、配额等）的 sidecar proxy:



- Istio-Auth: 提供强大的服务间认证和终端用户认证，使用交互 TLS，内置身份和证书管理。可以升级服务网格中的未加密流量，并为运维人员提供基于服务身份而不是网络控制来执行策略的能力。Istio 的未来版本将增加细粒度的访问控制和审计，以使用各种访问控制机制（包括基于属性和角色的访问控制以及授权钩子）来控制和监视访问您的服务，API 或资源的人员。



服务 A 以服务帐户“foo”运行，服务 B 以服务帐户“bar”运行，他们之间的通讯原来是没有加密的。但是 Istio 在不修改代码的情况，依托 Envoy 形成的服务网格，直接在客户端 Envoy 和服务器端 Envoy 之间进行通讯加密。

Istio 的很多设计理念的确非常吸引人，又有 Google 和 IBM 两个巨人加持，理论上这条赛道上的其他选手都可以直接退赛回家了。但是 Istio 发布的前几个版本都在可用性和易用性上都差强人意。因此，一方面 Linkerd 宣布和 Istio 项目集成，一方面夜以继日的开发 Conduit。网上说作者放弃了 Linkerd，但是在其官网上却看到 We're happy to announce that Conduit has been merged into the Linkerd project, where it will form the basis of Linkerd 2.0。

反思

对于大规模部署微服务（微服务数>1000）、内部服务异构程度高(交互协议/开发语言类型>5)的场景，使用 service mesh 是合适的。但是，可能大部分开发者面临的微服务和内部架构异构复杂度是没有这么高的。在这种情况下，使用 service mesh 就是一个 case by case 的问题了。

理论上，service mesh 实现了业务逻辑和控制的解耦。但是这并不是免费的。由于网络中多了一跳，增加了性能和延迟的开销。另一方面，由于每个服务都需要 sidecar, 这会给本来就复杂的分布式系统更加复杂，尤其是在实施初期，运维对 service mesh 本身把控能力不足的情况下，往往会使整个系统更加难以管理。

小结

service mesh 对于大规模部署、异构复杂的微服务架构是不错的方案。对于中小规模的微服务架构，不妨尝试一下更简单可控的 gateway，在确定 gateway 已经无法解决当前问题后，再尝试渐进的完全 service mesh 化。

最后

本着对知识负责的态度，部分文档官网已经有更新，感兴趣的可以自动跳转。