

# 正确入门Service Mesh：起源、发展和现状

原创 楚衡 阿里技术 7月28日

收录于话题

#微服务 391 #云原生 193 #容器 149



阿里妹导读：Service Mesh早已不是一个新兴的概念，但大家对Service Mesh的探索依然火热。本文将依次讲解Service Mesh的定义（什么是Service Mesh）、起因（为什么需要Service Mesh）和现状（Service Mesh的主流实现），希望通过浅显易懂的介绍，尽量帮助大家更好地理解Service Mesh。

文末福利：课程 - 【微服务实战：Service Mesh与Istio】

## 引言

随着云原生时代的来临，微服务架构与容器化部署模式越来越流行，从原来的新潮词汇慢慢演变成现代IT企业的技术标配。曾经被认为理所当然的巨无霸单体应用，被拥抱了微服务的架构师们精心拆分成了一个又一个小而独立的微服务，接着再被拥抱了容器化的工程师们打包成了自带依赖的

Docker镜像，最后通过某种神秘的DevOps流水线持续运送到前线 —— 也就是无人不知的 —— 风暴降生·谷歌之子·打碎镣铐者·云时代操作系统·Kubernetes —— 之中部署和运行。

听上去似乎一切都很美好？显然不是，这世上永远没有免费的午餐。所有美好的东西都会有它的阴暗面，微服务也不例外：

- 原来只需要部署和管理单个应用，现在一下裂变成了好几个，运维管理成本成倍上升。
- 原来各个模块之间的交互可以直接走应用内调用（进程间通信），现在都给拆分到了不同进程甚至节点上，只能使用复杂的RPC通讯。

难道辛辛苦苦落地了微服务，只能一边在老板面前强撑着“没问题，一切安好”，另一边默默忍受着研发与运维的私下抱怨？显然也不是。对于以“偷懒”著称的程序员们，办法总是比困难多。比如上面第1个问题，云原生所倡导的DevOps和容器化，就是一剂几乎完美的解药：通过自动化的CI/CD流水线，多应用的集成构建部署变得更加快捷；通过Docker镜像和K8s编排，多应用的资源调度运维管理也变得不那么痛苦。至于第2个问题，那就该看本文的主角 —— Service Mesh（服务网格），是如何力挽狂澜，近乎完美地解决微服务之间的通讯问题了。

## 什么是 Service Mesh?

### Service Mesh 诞生

从概念到落地？不，是从落地到概念。



#### Service Mesh 概念诞生

- 时间：2016年9月29日
- 场合：Buoyant公司内部分享



#### Service Mesh 最早实现

- 2016年1月15日：初次发布
- 2017年1月23日：加入CNCF
- 2017年4月25日：1.0版本发布



#### Service Mesh 革命先驱

- William Morgan (Buoyant CEO)
- 定义了 Service Mesh

时间回到2016年9月29日，那是一个即将放假迎来普天同庆的日子（是说我们）。在Buoyant公司内部一次关于微服务的分享会上，“Service Mesh”，这个接下来几年占据各种云原生头条的 buzz word，就这么被造出来了。不得不说，起名真是门艺术，Micro-Services -> Service Mesh，多

么承前启后和顺其自然啊，光看名字就能很形象地理解这玩意儿所做的事情：把微服务的各个service（服务）节点，用一张mesh（网格）连接起来。就这样，原本被拆散得七零八落的微服务们，又被 Service Mesh 这张大网紧密得连接到了一起；即使依然天各一方（进程间隔离），但也找回了当年一起挤在单体应用内抱团撒欢的亲密感（通信更容易）。

最难得的是，这么好的一个概念居然不是从PPT里走出来的，人家是真的有货（这让广大PPT创业者们情何以堪）：2016年1月15日，Service Mesh的第一个实现Linkerd [1]就已经完成了初次发布，紧接着次年1月23日 加入了CNCF，同年4月25日发布了 1.0版本。对于Buoyant公司而言，这也许只是无心插柳的一小步，但却是云原生领域迈向成熟的一大步。几年后的今天，Service Mesh概念早已深入人心，各种生产级实现和大规模实践也已遍地开花，但请不要忘记这一切背后的功臣、Service Mesh革命先驱、Buoyant公司CEO —— William Morgan，以及他对Service Mesh的定义和思考：What's a service mesh? And why do I need one?[2]

## Service Mesh 定义

别废话了，我没工夫听你说这么多，请用一句话跟我解释 Service Mesh 是什么。

好的。A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.

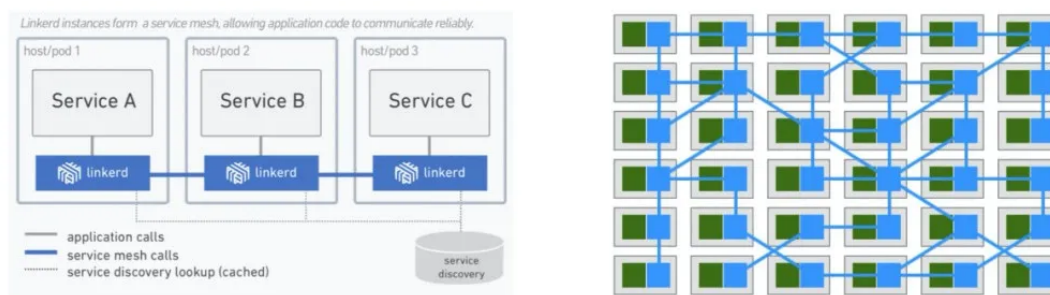
这就是上面那位又帅又能写的CEO，对Service Mesh的权威定义。我把其中一些重点词汇做了高亮：

- “dedicated infrastructure layer”：Service Mesh 不是用来解决业务领域问题的，而是一层专门的基础设施（中间件）。
- “service-to-service communication”：Service Mesh 的定位很简单也很清晰，就是用来处理服务与服务之间的通讯。
- “reliable delivery of requests”：服务间通讯为什么需要特殊处理？因为网络是不可靠的，Service Mesh 的愿景就是让服务间的请求传递变得可靠。

- “cloud native application”：Service Mesh 从一开始就是为现代化的云原生应用而生，瞄准了未来的技术发展趋势。
- “network proxies”：具体 Service Mesh 应该怎么实现？典型方式都是通过一组轻量级的网络代理，在应用无感知的情况下偷偷就把这事给干了。
- “deployed alongside application code”：这些网络代理一定是跟应用部署在一起，一对一近距离贴心服务（比房产中介专一得多）；否则，如果应用与代理之间也还是远程不靠谱通讯，这事儿就没完了。

## Service Mesh 形态

想致富，先修路；但大马路可不是给马走的，是给更现代化的汽车。



Service Mesh (服务网格)

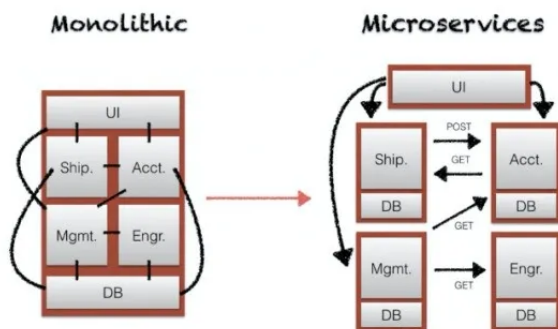
左边这张图是Linkerd的部署示意图，其中每个微服务所处的主机（host）或容器组（pod）中都会部署一个Linkerd代理软件，用于代理微服务应用实例之间的RPC调用。对于应用而言，这一切都是无感知的：它还是照常发起自己的RPC调用，只是不再需要关心对端服务方的地址，因为服务发现都由代理节点给cover了。

右边是一张更高维和抽象的大图，可以更形象地理解 Service Mesh 的逻辑形态 —— 想象这就是一个生产级的大规模微服务集群，其中部署了上百个服务实例以及对应的 Service Mesh 代理节点；所有微服务之间的通讯都会流经这些密密麻麻的代理节点，它们共同组成了一张川流不息的现代化交通网络。

## 为什么需要 Service Mesh ?

## 微服务的崛起

The Big Bang：大爆炸后的混乱之治。



### 微服务带来的福利

- ▶ 单一职责：易于开发、理解和维护
- ▶ 架构灵活：可以自由选择技术栈
- ▶ 部署隔离：快速部署，持续集成
- ▶ 独立扩展：不同服务按需扩展

### 引入的服务通讯问题

- ▶ 如何找到服务的提供方？
- ▶ 如何保证远程调用的可靠性？
- ▶ 如何降低服务调用的延迟？
- ▶ 如何保证服务调用的安全性？

大多数人都曾经历过那个单体应用为王的时代。所谓“单体”（Monolithic），就是把所有组件都塞在同一个应用内，因此这些组件天然就紧密联系在一起：基于相同技术栈开发、访问共享的数据库、共同部署运维和扩容。同时，这些组件之间的通讯也趋向于频繁和耦合——不过就是一句函数调用的事，何乐而不为。这样做本身也没什么错，毕竟那时的软件系统相对简单，可能一个人写个两万行代码的单体应用，就能轻松搞定所有业务场景。

天下大事，分久必合，合久必分。现代化软件系统的复杂度不断提升，协作人数也越来越多，单体应用的固有局限性开始暴露。就仿佛宇宙大爆炸前的那个奇点，单体应用开始加速膨胀，最终在几年前达到了临界点，然后“砰”的一声就炸开了。就这样，微服务时代王者降临，让软件开发重新变得“小而美”：

- 单一职责：拆分后的单个微服务，通常只负责单个高内聚自闭环功能，因此很易于开发、理解和维护。
- 架构灵活：不同微服务应用之间在技术选型层面几乎是独立的，可以自由选择最适合的技术栈。
- 部署隔离：相比巨无霸单体应用，单个微服务应用的代码和产物体积大大减少，更容易持续集成和快速部署；同时，通过进程级别的隔离，也不再像单体应用一样只能同生共死，故障隔离效果显著提升。

- 独立扩展：单体应用时代，某个模块如果存在资源瓶颈（e.g. CPU/内存），只能跟随整个应用一起扩容，白白浪费很多资源。微服务化后，扩展的粒度细化到了微服务级别，可以更精确地按需独立扩展。

但显然，微服务也不是银弹。大爆炸虽然打破了单体应用的独裁统治，但那一声声炸裂之后的微服务新宇宙，显然也不会立即就尘埃落定，而是需要经历很长一段时间的混乱之治。适应了单体时代的开发者们，被迫需要拥抱微服务所带来的一系列变化。其中最大的变化，就是服务间通讯：

## 如何找到服务的提供方？

微服务通讯必须走远程过程调用（HTTP/REST本质上也属于RPC），当其中一个应用需要消费另一个应用的服务时，无法再像单体应用一样通过简单的进程内机制（e.g. Spring的依赖注入）就能获取到服务实例；你甚至都不知道有没有这个服务方。

## 如何保证远程调用的可靠性？

既然是RPC，那必然要走IP网络，而我们都知网络（相比计算和存储）是软件世界里最不可靠的东西。虽然有TCP这种可靠传输协议，但频繁丢包、交换机故障甚至电缆被挖断也常有发生；即使网络是好的，如果对方机器宕机了，或者进程负载过高不响应呢？

## 如何降低服务调用的延迟？

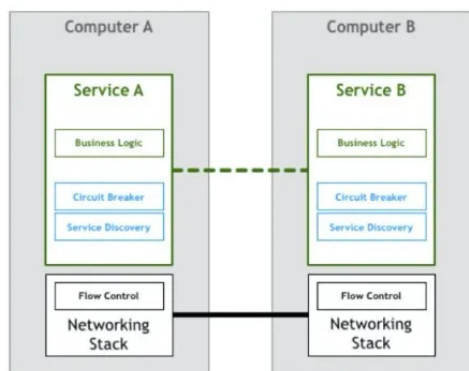
网络不只是不可靠，还有延迟的问题。虽然相同系统内的微服务应用通常都部署在一起，同机房内调用延迟很小；但对于较复杂的业务链路，很可能一次业务访问就会包括数十次RPC调用，累积起来的延迟就很可观了。

## 如何保证服务调用的安全性？

网络不只是不可靠和有延迟，还是不安全的。互联网时代，你永远不知道屏幕对面坐的是人还是狗；同样，微服务间通讯时，如果直接走裸的通讯协议，你也永远不知道对端是否真的就是自己人，或者传输的机密信息是否有被中间人偷听。

## 服务通讯：石器时代

毛主席说：自己动手，丰衣足食。



### 自己动手，丰衣足食

- ▶ 服务发现 (Service Discovery)
- ▶ 服务熔断 (Circuit Breaker)
- ▶ 负载均衡 (Load Balancing)
- ▶ 协议加密、身份认证、访问鉴权

### But，时间都去哪了？

- ▶ 需要编写大量非功能性代码
  - ▶ 如何集中精力专注业务创新？
- ▶ 维护这份代码也需要大量精力
  - ▶ 代码混杂：分布式相关Bug

为了解决上述微服务引入的问题，最早那批吃螃蟹的工程师们，开始了各自的造轮子之旅：

- 服务发现 (Service Discovery)：解决“我想调用你，如何找到你”的问题。
- 服务熔断 (Circuit Breaker)：缓解服务之间依赖的不可靠问题。
- 负载均衡 (Load Balancing)：通过均匀分配流量，让请求处理更加及时。
- 安全通讯：包括协议加密 (TLS)、身份认证 (证书/签名)、访问鉴权 (RBAC) 等。

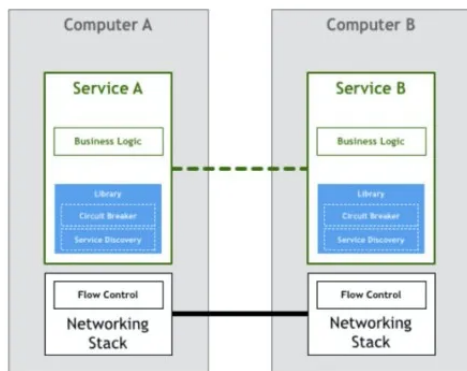
用自己的代码解决问题，这确实是程序员们能干出来的事，没毛病。But，时间都去哪了？

- 重复造轮子：需要编写和维护大量非功能性代码，如何集中精力专注业务创新？
- 与业务耦合：服务通讯逻辑与业务代码逻辑混在一起，动不动还会遇到点匪夷所思的分布式bug。

### 服务通讯：摩登时代

社会主义精神：共享和复用。





## 软件复用：类库 / 框架



NETFLIX  
OSS



gRPC

## Good enough ?

- ▶ 并非完全透明：仍需正确理解和使用库
- ▶ 限制技术选择：与语言/框架强绑定
- ▶ 维护成本高：库版本升级，牵连应用

更有思想觉悟的那批工程师们坐不住了：你们这是违背了共享和复用原则，对不起GNU那帮祖师爷！于是，各种高质量、标准化、期望能大一统的精品轮子们应运而生，包括 Apache Dubbo（手动置顶）、Spring Cloud、Netflix OSS、gRPC 等等等。

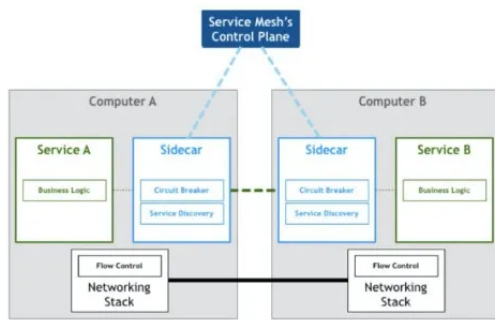
这些可复用的类库和框架，确确实实带来了质量和效率上的大幅提升，但是足够好使了吗？Not enough：

- 并非完全透明：程序员们仍然需要正确理解和使用这些库，上手成本和出错概率依然很高。
- 限制技术选择：使用这些技术后，应用很容易就会被对应的语言和框架强绑定（vendor-lock）。
- 维护成本高：库版本升级，需要牵连应用一起重新构建和部署；麻烦不说，还要祈祷别出故障。

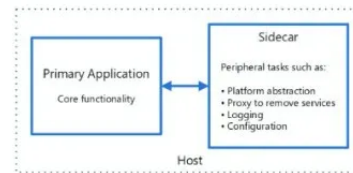
## 服务通讯：新生代

Service Mesh：我只是一个搬运工而已。





Service Mesh Pattern



Sidecar Pattern

Service Mesh 的诞生，彻底解决了上述所有问题。听上去很神奇，究竟是如何办到的呢？简单来说，Service Mesh 就是通过 Sidecar 模式[3]，将上述类库和框架要干的事情从应用中彻底剥离了出来，并统一下沉到了基础设施层。这是一种什么思想？这是一种古老操作系统中早就有了的抽象和分层思想（应用程序并不需要关心网络协议栈），也是一种现代云计算平台自底向上逐步托管的软件服务化思想（IaaS -> CaaS -> PaaS -> SaaS）。

上述几张 Service Mesh 的演进图，参考自 Service Mesh Pattern[4] 一文。

## Service Mesh 主流实现

注：以下内容来自于资料搜集整理，仅供参考，更进一步学习请以最新权威资料为准。

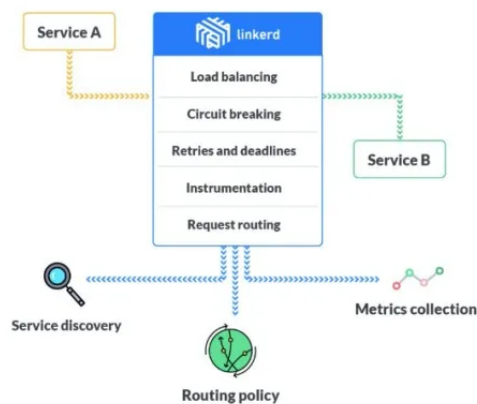
### 主流实现概览

 <b>LINKERD</b>	 <b>envoy</b>	 <b>Istio</b>	 <b>CONDUIT</b>
<ul style="list-style-type: none"> <li>• 背后公司: <b>Buoyant</b></li> <li>• 开发语言: <b>Scala</b></li> <li>• 2016年1月15日: <a href="#">初次发布</a></li> <li>• 2017年1月23日: <a href="#">加入CNCF</a></li> <li>• 2018年5月1日: <a href="#">1.4.0版本发布</a></li> </ul>	<ul style="list-style-type: none"> <li>• 背后公司: <b>Lyft</b></li> <li>• 开发语言: <b>C++ 11</b></li> <li>• 2016年9月13日: <a href="#">初次发布</a></li> <li>• 2017年9月14日: <a href="#">加入CNCF</a></li> <li>• 2018年3月21日: <a href="#">1.6.0版本发布</a></li> </ul>	<ul style="list-style-type: none"> <li>• 背后公司: <b>Google、IBM</b></li> <li>• 开发语言: <b>Go</b></li> <li>• 2017年5月10日: <a href="#">初次发布</a></li> <li>• 2018年3月31日: <a href="#">0.7.1版本发布</a></li> </ul>	<ul style="list-style-type: none"> <li>• 公司: <b>Buoyant</b></li> <li>• 语言: <b>Rust、Go</b></li> <li>• 2017年12月5日: <a href="#">初次发布</a></li> <li>• 2018年4月27日: <a href="#">0.4.1版本发布</a></li> </ul>

Service Mesh 的主流实现包括：

- Linkerd：背后公司是Buoyant，开发语言使用Scala，2016年1月15日初次发布，2017年1月23日加入CNCF，2018年5月1日发布1.4.0版本。
- Envoy：背后公司是Lyft，开发语言使用C++ 11，2016年9月13日初次发布，2017年9月14日加入CNCF，2018年3月21日发布1.6.0版本。
- Istio：背后公司是Google和IBM，开发语言使用Go，2017年5月10日初次发布，2018年3月31日发布0.7.1版本。
- Conduit：背后公司也是Buoyant，开发语言使用Rust和Go，2017年12月5日初次发布，2018年4月27日发布0.4.1版本。

## Linkerd 简介



### Linkerd 请求处理流程

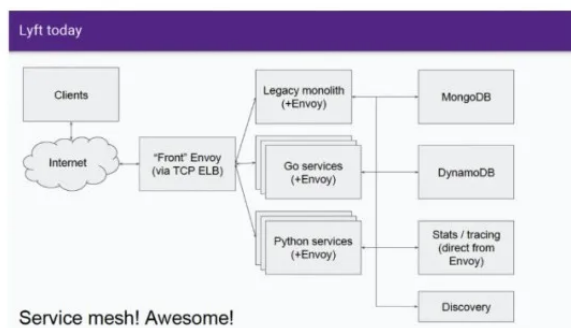
- ① 根据请求参数，确定目标服务（**动态路由**）
  - 灰度发布、A/B测试、环境隔离
- ② 获取目标服务实例列表（**服务发现**）
- ③ 选择一个合适的低延迟实例（**负载均衡**）
  - 算法：Least Loaded、Peak EWMA、...
- ④ **发送请求**到对应实例，并记录延迟和结果
- ⑤ 如果请求未响应，则选择另一个实例**重试**
  - 前提：Linkerd知道该请求是幂等的
- ⑥ 如果一个实例经常失败，剔除该实例（**熔断**）
- ⑦ 如果请求**超期**（deadline），则主动失败
  - Deadline vs. Timeout
- ⑧ 收集并上报上述行为的**Metrics**和**Tracing**

Linkerd的核心组件就是一个服务代理，因此只要理清它的请求处理流程，就掌握了它的核心逻辑：

- 动态路由：根据上游服务请求参数，确定下游目标服务；除了常规的服务路由策略，Linkerd还可以通过这一层动态路由能力，支持灰度发布、A/B测试、环境隔离等非常有价值的场景。
- 服务发现：确定目标服务后，下一步就是获取对应的实例的地址列表（e.g. 查询service registry）。
- 负载均衡：如果列表中有多个地址，Linkerd会通过负载均衡算法（e.g. Least Loaded、Peak EWMA）选择其中一个合适的低延迟实例。
- 执行请求：发送请求到上一步所选择的实例，并记录延迟和响应结果。

- 重试处理：如果请求未响应，则选择另一个实例重试（前提：Linkerd知道该请求是幂等的）。
- 熔断处理：如果发往某个实例的请求经常失败，则主动从地址列表中剔除该实例。
- 超时处理：如果请求超期（在给定的deadline时间点之前仍未返回），则主动返回失败响应。
- 可观测性：Linkerd会持续收集和上报上述各种行为数据，包括Metrics和Tracing。

## Envoy 简介



Lyft 公司 基于 Envoy 的服务架构

### Envoy 特性总结

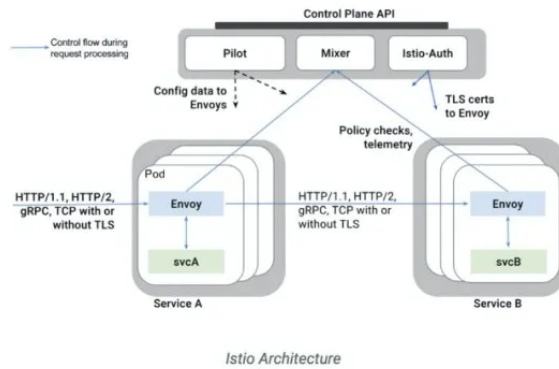
- 高性能的本地代码实现 (C++ 11)
- 符合最终一致性的服务发现
- API驱动的配置
- 基于可扩展的Filter Chain的L4 (TCP) 代理
- 并行的可插拔的L7 Filter Chain
- 双向、透明的HTTP/1 to HTTP/2代理
- 微服务架构中健壮、一致的可观察性
- 很容易调试 (Debugging)
- 高级的负载均衡，支持区域感知、重试、超时、熔断、限速、异常检测
- 基于统计、日志和分布式追踪的可观察性

Envoy是一个高性能的Service Mesh软件，主要包含如下特性：

- 高性能：基于本地代码（C++ 11）实现；相比之下，Linkerd是基于Scala编写，肯定要慢不少。
- 可扩展：L4和L7层代理功能均基于可插拔的 Filter Chain 机制（类比 netfilter、servlet filter）。
- 协议升级：支持双向、透明的 HTTP/1 to HTTP/2 代理能力。
- 其他能力：服务发现（符合最终一致性）、负载均衡（支持区域感知）、稳定性（重试、超时、熔断、限速、异常检测）、可观测性（统计/日志/追踪）、易于调试等。

## Istio 简介

## Istio 组件概览



### ★ Envoy

- ✦ 构成数据平面（其他组件共同构成控制平面）
- ✦ 可被替换为其他代理（Linkerd, nginMesh）

### ★ Pilot

- ✦ 负责流量管理（Traffic Management）
- ✦ 提供平台独立的服务模型定义、API以及实现

### ★ Mixer

- ✦ 负责策略与控制（Policies & Controls）
- ✦ 核心功能：前置检查、配额管理、遥测报告

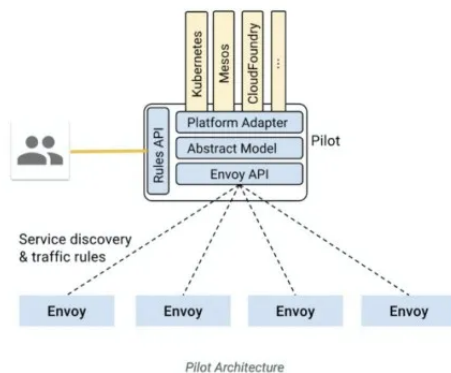
### ★ Istio-Auth

- ✦ 基于角色的访问控制（RBAC）：支持多种粒度
- ✦ 双向SSL认证：身份识别、通讯安全、密钥管理

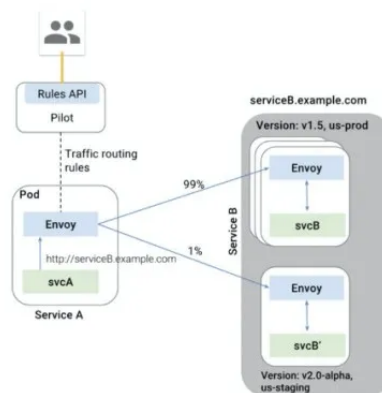
Istio是一个管控/数据平面分离的完整Service Mesh套件，包含如下组件：

- Envoy：构成数据平面（其他组件共同构成控制平面）；可被替换为其他代理（e.g. Linkerd, nginMesh）。
- Pilot：负责流量管理（Traffic Management），提供平台独立的服务模型定义、API以及实现。
- Mixer：负责策略与控制（Policies & Controls），核心功能包括：前置检查、配额管理、遥测报告。
- Istio-Auth：支持多种粒度的RBAC权限控制；支持双向SSL认证，包括身份识别、通讯安全、密钥管理。

## Istio 组件 - Pilot



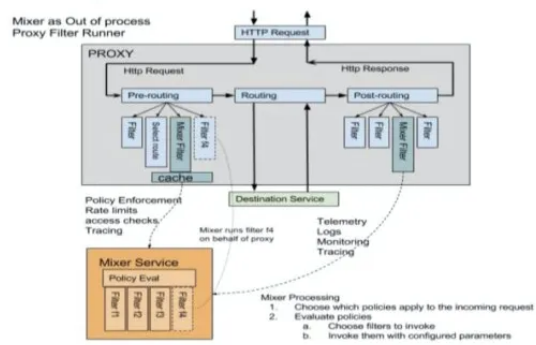
Pilot 组件架构



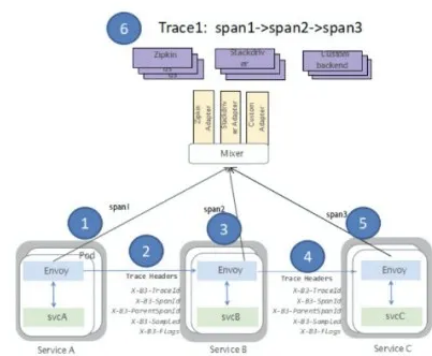
典型场景：灰度发布

Pilot组件是Istio服务网格中的“领航员”，负责管理数据平面的流量规则和服务发现。一个典型的应用场景就是灰度发布（or 金丝雀发布、蓝绿部署）：开发者通过Pilot提供的规则API，下发流量路由规则到数据平面的Envoy代理，从而实现精准的多版本流量分配（e.g. 将1%的流量分配到新版本服务）。

Istio 组件 - Mixer



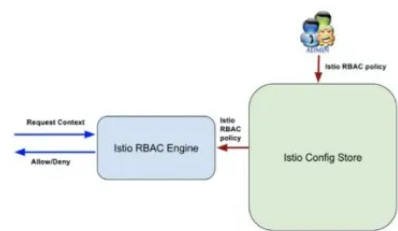
Mixer 工作原理



典型场景：分布式Tracing

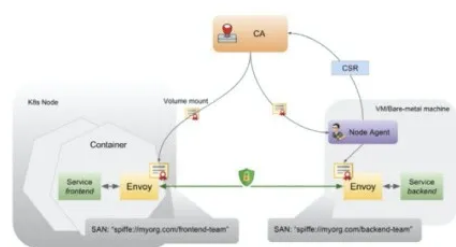
Mixer组件是Istio服务网格中的“调音师”，既负责落实各种流量策略（如访问控制、限速），也负责对流量进行观测分析（如日志、监控、追踪）。这些能力都是通过前文提到的Envoy Filter Chain 扩展机制实现：Mixer会分别在“请求路由前”（Pre-routing）扩展点和“请求路由后”（Post-routing）扩展点挂载自己的Filter实现。

Istio 组件 - Auth



Istio RBAC Architecture

1. 基于角色的访问控制（RBAC）

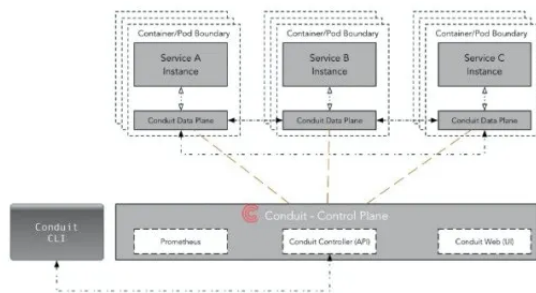


Istio Auth Architecture

2. 双向SSL认证

Auth组件是Istio服务网格中的“安全员”，负责处理服务节点之间通信的认证（Authentication）和鉴权（Authorization）问题。对于认证，Auth支持服务之间的双向SSL认证，可以让通讯的双方都彼此认可对方的身份；对于鉴权，Auth支持流行的RBAC鉴权模型，可以实现便捷和细粒度的“用户-角色-权限”多级访问控制。

## Conduit 简介



### Introducing Conduit

Istio 的挑战者：更轻量！更快！

## Conduit 主要特性



### Ultralight & blazingly fast

Conduit's data plane is written in Rust, making it incredibly small, fast, and secure. Each proxy runs in less than 10mb RSS and has sub-millisecond p99 latency, providing the functionality of the service mesh without the cost.



### Security from the start

From Rust's memory safety guarantees to TLS by default, Conduit is built to help secure cloud native environments from the ground up.



### End-to-end visibility

Conduit automatically measures and aggregates service success rates, latencies, and request volumes, giving you an unfettered view into service behavior across your infrastructure without needing to change application code.



### Kubernetes enhanced

Conduit adds reliability, visibility, and security to your Kubernetes cluster, giving you control of the runtime behavior of your applications.

Conduit是由Buoyant公司出品的下一代 Service Mesh。作为Istio的挑战者，Conduit的整体架构与Istio类似也明确区分了管控平面和数据平面，但同时它还具有如下关键特性：

- 轻量快速：Conduit的数据平面是基于原生的Rust语言编写，非常轻量、快速和安全（Rust相比C/C++的最大改进点就是安全性）。单个代理的实际内存消耗（RSS）小于10mb，延迟的p99分位点小于1ms，基本相当于能为应用程序提供免费（无额外开销）的Service Mesh功能。
- 安全保障：Conduit构建之初就考虑了云原生环境的安全性，包括Rust语言内存安全性、默认TLS加密等。
- 端到端可见性：Conduit可以自动测量和聚合服务的成功率、延迟与请求容量数据，让开发者在无需变更应用代码就能获取到服务的完整行为视图。
- Kubernetes增强：Conduit为K8s集群添加了可靠性、可见性和安全性，同时给予了开发者对自己应用程序运行时行为的完全控制。

## 结语

本文从云原生时代所面临的微服务通讯问题入手，依次介绍了Service Mesh的起源、发展和现状，希望能帮助读者建立一个初步的理解和认知。当然，实践出真知，与其临渊羡鱼（眼馋Service Mesh的技术红利），不如退而结网（自己动手织一张Service网格）。手头的工作没有可实践的业务场景？没关系，我这有：

欢迎各位技术同路人加入阿里云云原生应用研发平台EMAS团队，我们专注于广泛的云原生技术（Backend as a Service、Serverless、DevOps、低代码平台等），致力于为企业、开发者提供一站式的应用研发管理服务，内推直达邮箱：pengqun.pq # alibaba-inc.com，有信必回。

#### 相关链接

[1]<https://github.com/linkerd/linkerd>

[2]<https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/>

[3]<https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar>

[4][https://philcalcado.com/2017/08/03/pattern\\_service\\_mesh.html](https://philcalcado.com/2017/08/03/pattern_service_mesh.html)

### 技术公开课

#### 【微服务实战】Service Mesh与Istio


Istio作为一个Service Mesh开源项目，其中最重要的功能就是对网格中微服务之间的流量进行管理，包括服务发现、请求路由和服务间的可靠通信。什么是Service Mesh？Istio有哪些功能模块？如何使用Istio来进行流量管理？本课程共 8 个课时，带你了解Service Mesh的概念以及Istio的应用。

点击“阅读原文”立即学习吧~



关注「阿里技术」  
把握前沿技术脉搏



 戳我，去学习。

[阅读原文](#)

---