

目录

Introduction	1.1
时间线	1.2
业界精英	1.3
Christian Posta	1.3.1
Daniel Bryant	1.3.2
Phil Calçado	1.3.3
William Morgan	1.3.4
Service Mesh	1.4
Service Mesh:下一代微服务	1.4.1
应用网络功能，ESB，API管理，而今..Service Mesh?	1.4.2
模式之服务网格	1.4.3
什么是服务网格以及为什么我们需要服务网格？	1.4.4
Istio	1.5
Istio:服务网格新生力量	1.5.1
Linkerd	1.6
标签总览	1.7

Awesome Service Mesh

Service Mesh是一个新兴的技术，资料相对比较少，为了便于收集整理Service Mesh相关的博客/文章/演讲/视频等资料，Service Mesh中文网决定创建并维护这份Awesome Service Mesh资料清单。

资料内容来自国内公开媒体，转载时我们会指明出处，标注原作者和原译者，如果是原创或者原创翻译也会标明作者或者译者身份。

首页为所有内容的列表和访问链接，后续内容是我们认为非常有阅读价值的中文内容，以方便大家集中浏览或者下载后阅读：

- 在线阅读
 - 国外服务器：gitbook提供的托管，服务器在国外，速度比较慢，偶尔被墙
 - 国内服务器：TBD，稍后提供
- 下载pdf格式
- 下载mobi格式
- 下载epub格式

Service Mesh中文网

Service Mesh中文网是由Service Mesh中国社区发起成立的公益性纯技术交流网站，访问地址：

<http://www.servicemesh.cn/>

注：由于国家政策的要求，网站备案时已经无法再使用诸如"中国"/"社区"等字眼，所以原定名称"Service Mesh中国社区"被迫修改为"Service Mesh中文网"。

为了方便社区交流，我们建立了微信技术讨论群，有意加入请联系微信id `xiaoshu062`，注明"service mesh"或者"服务网格"，目前：

- 1群/2群500人已满
- 3群/4群超过400人

Awesome Service Mesh

- [Service Mesh](#)
- [Istio](#)
- [Conduit](#)

- Envoy
- Linkerd
- Nginmesh
- Aspenmesh

ServiceMesh

时间	标题	作者	中文翻译	备注
2018-1	解读2017之Service Mesh： 群雄逐鹿烽烟起 	敖小剑		Infoq年度总结系列文章
2017-12	山雨欲来风满楼： Service Mesh时代的选边与站队	敖小剑		北京meetup演讲实录
2017-10	Service Mesh：下一代微服务	敖小剑		QCON演讲实录
2017-08	Pattern: Service Mesh	Phil Calçado	模式之服务网格	详细介绍Service Mesh模式和演进过程，强烈推荐
2017-08	Application Network Functions With ESBs, API Management, and Now.. Service Mesh?	Christian Posta	应用网络功能，EBS，API管理，而现在..Service Mesh?	
2017-07	O'Reilly 2017: "Introduction to Service Meshes" 	Daniel Bryant	PPT的中文版本	演讲的PPT
2017-04	What's a service mesh? And why do I need one?	William Morgan	什么是服务网格以及为什么我们需要服务网格？	应该是业界第一篇详细介绍Service Mesh理念的文章，强烈推荐

Istio

文档

- istio官方文档中文版

工具

- 在线体验istio: 实测可用
- istio-workshop

文章

时间	标题	作者	中文翻译	备注
2017-11	 8 Steps to Becoming Awesome with Kubernetes	Burr Sutter		PPT 国内下载(免翻墙)
2017-10	Kubernetes, Microservices, and Istio — A Great Fit!	Animesh Singh		
2017-09	什么是Service Mesh (服务网格)	宋净超		
2017-09	服务网格新生代--Istio	敖小剑		线上分享实录
2017-09	What is Istio? It's a service mesh. Great. What's a service mesh?	Nick Chase	什么是Istio? 它是服务网格。棒极了，那什么是服务网格？	访谈录
2017-05	Google, IBM and Lyft launch Istio, an open-source platform for managing and securing microservices	Frederic Lardinois	Istio开源平台发布，Google、IBM和Lyft分别承担什么角色？	
2017-05	Google, IBM, Lyft partner on open source microservices management platform Istio	Conner Forrest		-

Conduit

文档

- conduit官方文档中文版

文章

时间	标题	作者	中文翻译	备注
2017-12	Conduit AMA session recap 	George Miranda	Conduit AMA 活动回放	介绍conduit项目的情况
2017-12	Introducing Conduit	William Morgan	Conduit登场	conduit第一次亮相

Envoy

Linkerd

时间	标题	作者	中文翻译	备注
2017-10	A SERVICE MESH FOR KUBERNETES		Kubernetes的服务 mesh	Linkerd官方博客上连载的系列文章,部分内容有翻译

Nginxmesh

时间	标题	作者	中文翻译	备注
2017-09	NGINX Releases Microservices Platform, OpenShift Ingress Controller, and Service Mesh Preview	Daniel Bryant	NGINX 发布微服务平台、OpenShift Ingress Controller 和 Service Mesh预览版	-

Aspenmesh

时间	标题	作者	中文翻译	备注
2017-12	REDtalks.live – Aspen Mesh 	Shawn Wormke		对Aspen Mesh的Sr. Director的采访视频

时间线

2015

- 2015年初，**Envoy**诞生

Matt Klein在Lyft开始Envoy的早期开发。

2016

- 2016年1月15日，**Linkerd**诞生

离开twitter的基础设施工程师William Morgan和Oliver Gould，在github上发布了linkerd 0.0.7版本。他们还组建了一个创业小公司buoyant。业界第一个Service Mesh项目诞生。

- 2016年9月13日，**Envoy**开源

Matt Klein宣布Envoy在github开源，直接发布v1.0.0版本

- 2016年9月29日，"service mesh"进入社区视野

在SF Microservices，"service mesh"这个词汇第一次在公开场合使用。service mesh这个概念，从buoyant公司内容走出，开始被社区了解。

- 2016年10月，"A Service Mesh for Kubernetes"系列博客发表

Alex Leong开始在buoyant公司的官方Blog中这个"A Service Mesh for Kubernetes"系列博客的连载。随着"The services must mesh"口号的喊出，buoyant和Linkerd开始service mesh概念的布道。

- 2016年中，**istio**诞生

google和IBM联手，开始istio项目，lyft以贡献envoy的方式加入。

2017

- 2017年1月23日，**Linkerd**加入CNCF
- 2017年4月25日，Linkerd1.0版本发布
- 2017年4月25日，**Service Mesh**权威定义

William Morgan发布博文"What's a service mesh? And why do I need one?"。正式给Service Mesh做了一个权威定义。

- 2017年5月24日，**Istio**诞生 0.1 release版本发布

- 2017年7月11日，Linkerd开始集成Istio

Linkerd发布版本1.1.1,宣布可以和istio项目集成。并发表博文，"Linkerd and Istio: like peanut butter and jelly"。

- 2017年9月14日，**Envoy**加入**CNCF**

- 2017年10月4日，Istio 0.2 release版本发布

- 2017年10月16日，Service Mesh亮相国内

在2017 QCon上海大会上，"Service Mesh：下一代微服务"的演讲，成为Service Mesh技术在国内大型技术峰会上的第一次亮相。

- 2017年12月1日，Istio 0.3 release版本发布

- 2017年12月，Service Mesh成为KubeConf大热门

- 2017年12月5日，**Conduit**诞生 0.1.0版本发布

2018

值得期待!

业界精英

我们在这里列出Service Mesh领域的各界精英，并收集他们的个人信息和曾经发布的技术资料。

有兴趣的同学可以通过Follow他们的twitter等方式关注他们。

Christian Posta

tags: Christian Posta

个人信息

以下信息来自他的blog：

<http://blog.christianposta.com/>



- Chief Architect
- cloud application development @Red Hat
- author Microservices for Java Developers
- open-source enthusiast
- committer @ Apache
- Cloud, Integration, Kubernetes, Docker, OpenShift, Fabric8

Red Hat Cloud的首席架构师，"Microservices for Java Developers"一书的作者，apache committer。

联系方式：

- Twitter : <http://twitter.com/christianposta>

- Google+ : <http://plus.google.com/+christianposta>
- LinkedIn : <http://linkedin.com/pub/christian-posta/15/412/199>
- Github : <http://github.com/christian-posta>
- Stackoverflow : <http://stackoverflow.com/users/248392/ceposta>

Daniel Bryant

tags: Daniel Bryant

个人信息

独立咨询师，OpenCredo首席科学家，SpectoLabs CTO，InfoQ编辑，大会演讲者



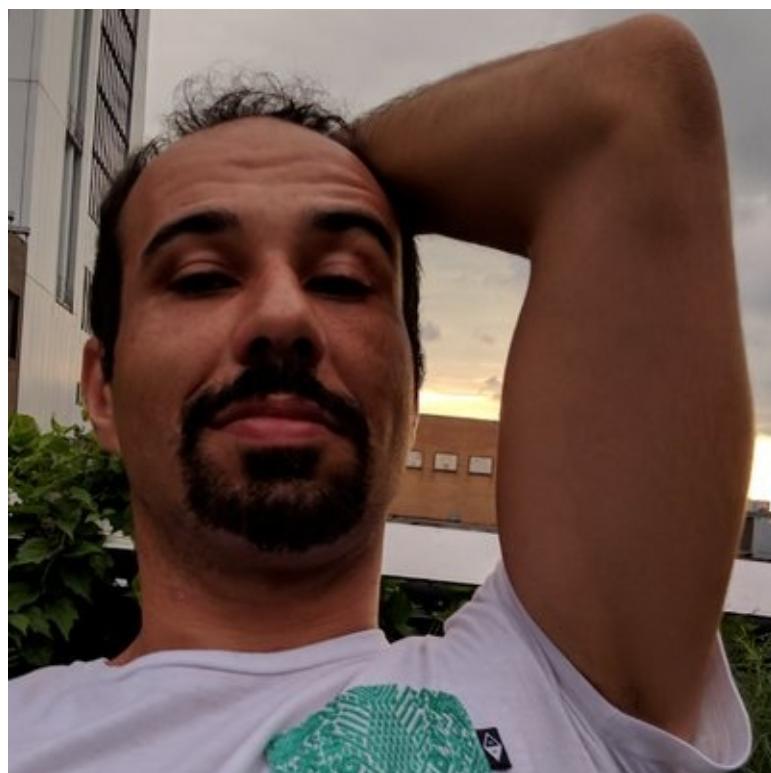
- 个人网站：<https://taidevcouk.wordpress.com/>
- 演讲@slideshare: https://www.slideshare.net/dbryant_uk
- Twitter：<https://twitter.com/danielbryantuk>
- InfoQ: <https://www.infoq.com/profile/Daniel-Bryant>

Phil Calçado

tags: Phil Calçado

个人信息

现在工作于Buoyant，曾就职于SoundCloud。twitter自我介绍中说“Rio to Melbourne to Sydney to London to Berlin and now New York City.”



- 个人网站：<http://philcalcado.com/>
- 演讲@slideshare：<https://www.slideshare.net/pcalcado>
- Twitter：<http://twitter.com/pcalcado>
- LinkedIn：<http://www.linkedin.com/in/pcalcado>
- Github：<http://github.com/pcalcado>

William Morgan

tags: William Morgan

个人信息

Buoyant CEO，service mesh理念的提出者和先行者，service mesh最早的布道师。曾就职于Twitter，powerset，adaptv。



- Twitter : <https://twitter.com/wm>
- LinkedIn : <https://www.linkedin.com/in/wmorgan/>

Service Mesh

Service Mesh：下一代微服务

tags: 敖小剑

背景：这是2017年11月4日，QCon 2017年上海会议上的演讲实录，为PPT图片加文字内容整理。这是Service Mesh技术第一次在国内技术社区正式亮相。

Slides下载：[PDF格式](#)

作者：敖小剑

主持人：接下来分享的嘉宾是数人云资深架构师敖小剑老师，分享的主题是“Service Mesh：下一代微服务”。



Service Mesh：下一代微服务

数人云 资深架构师 敖小剑

敖小剑：首先感谢这么多朋友来听我的演讲，今天我们分享的是Service Mesh，下一代微服务。我是敖小剑，来自数人云的资深架构师。

前言

懵懂2014 热点2015 普及2016 反省2017

在过去的三年中，微服务成为技术热点，大量互联网公司开始落地微服务架构，而对于传统企业用户，以微服务和容器为核心的互联网技术转型已是大势所趋。

Java社区中，伴随微服务大潮，以 Spring Cloud 为代表的微服务开发框架迅速普及。

而在 Spring Cloud 之外，新一代的微服务开发技术正在悄然兴起，这就是今天要给大家带来的 Service Mesh (服务网格)。

简单回顾一下过去三年微服务的发展历程。在过去三年当中，微服务成为我们的业界技术热点，我们看到大量的互联网公司都在做微服务架构的落地。也有很多传统企业在做互联网技术转型，基本上还是以微服务和容器为核心。

在这个技术转型中，我们发现有一个大的趋势，伴随着微服务的大潮，Spring Cloud微服务开发框架非常普及。而今天讲的内容在Spring Cloud之外，我们发现最近新一代的微服务开发技术正在悄然兴起，就是今天要给大家带来的Service Mesh/服务网格。



Service
Mesh

1

什么是Service Mesh ?

2

Service Mesh的演进历程

3

为何选择Service Mesh ?

我做一个小小的现场调查，今天在座的各位，有没有之前了解过服务网格的，请举手。（备注：调查结果，现场数百人仅有3个人举手）

既然大家都不了解，那我来给大家介绍。首先，什么是Service Mesh？然后给大家讲一下Service Mesh的演进历程，以及为什么选择Service Mesh以及为什么我将它称之为下一代的微服务，这是我们今天的内容。

1

What

什么是Service Mesh？

这真的是一个新名词

• 最早使用

- Service Mesh最早是由开发Linkerd的Buoyant公司提出，并在内部使用
- 2016年9月29日第一次公开使用

• 国内翻译

- 2017年初，随着Linkerd的传入，Service Mesh进入国内技术社区的视野
- 国内最早翻译为“服务啮合层”
- 但是这个词非常的拗口，后来逐渐改用“服务网格”



我们首先说一下Service Mesh这个词，这确实是一个非常非常新的名词，像刚才调查的，大部分的同学都没听过。

这个词最早使用由开发Linkerd的Buoyant公司提出，并在内部使用。2016年9月29日第一次公开使用这个术语。2017年的时候随着Linkerd的传入，Service Mesh进入国内技术社区的视野。最早翻译为“服务啮合层”，这个词比较拗口。用了几个月之后改成了服务网格。后面我会给大家介绍为什么叫网格。

Service Mesh的定义

- Willian Morgan (Linkerd的CEO) 给出的定义：

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.

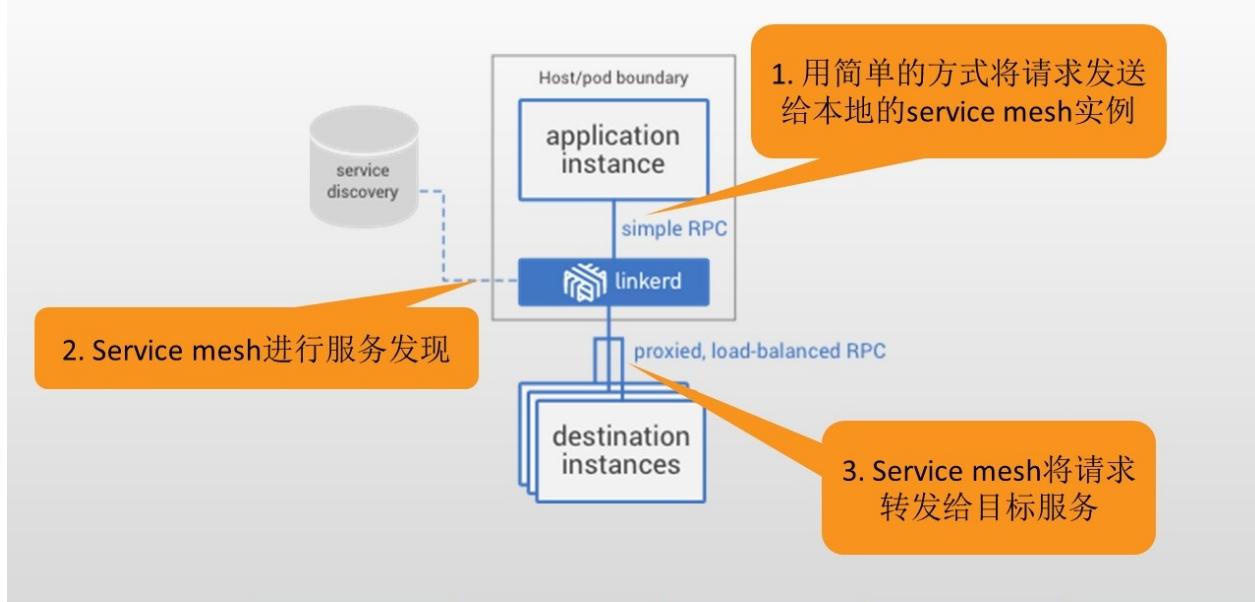
服务网格是一个**基础设施层**，用于处理服务间通信。云原生应用有着复杂的服务拓扑，服务网格负责在这些拓扑中**实现请求的可靠传递**。在实践中，服务网格通常实现为一组**轻量级网络代理**，它们与应用程序部署在一起，而对应用程序透明。

先看一下Service Mesh的定义，这个定义是由Linkerd的CEO William给出来的。Linkerd是业界第一个Service Mesh，也是他们创造了Service Mesh这个词汇的，所以这个定义比较官方和权威。

我们看一下中文翻译，首先服务网格是一个基础设施层，功能在于处理服务间通信，职责是负责实现请求的可靠传递。在实践中，服务网格通常实现为轻量级网络代理，通常与应用程序部署在一起，但是对应用程序透明。

这个定义直接看文字大家可能会觉得比较空洞，不太容易理解到底是什么。我们来看点具体的东西。

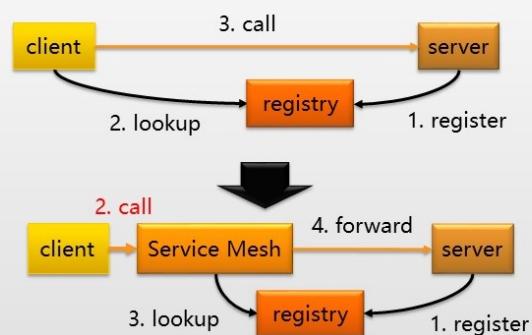
部署模型：单个服务调用，表现为sidecar



Service Mesh的部署模型，先看单个的，对于一个简单请求，作为请求发起者的客户端应用实例，会首先用简单方式将请求发送到本地的Service Mesh实例。这是两个独立进程，他们之间是远程调用。

Service Mesh会完成完整的服务间调用流程，如服务发现负载均衡，最后将请求发送给目标服务。这表现为 Sidecar。

Sidecar : 这个不是新名词



Sidecar这个词中文翻译为边车，或者车斗，也有一个乡土气息浓重的翻译叫做边三轮。Sidecar这个东西出现的时间挺长的，它在原有的客户端和服务端之间加多了一个代理。

部署模型：多个服务调用，表现为通讯层

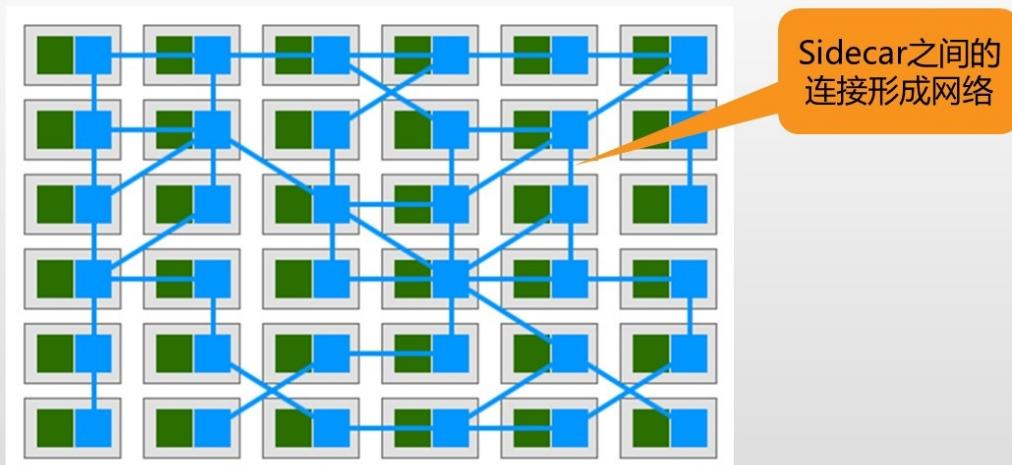
Service Mesh接管整个网络,负责转发请求

服务只管发送和接收处理请求，中间环节被剥离

服务间通信专用基础设施层

多个服务调用的情况，在这个图上我们可以看到Service Mesh在所有的服务的下面，这一层被称之为服务间通讯专用基础设施层。Service Mesh会接管整个网络，把所有的请求在服务之间做转发。在这种情况下，我们会看到上面的服务不再负责传递请求的具体逻辑，只负责完成业务处理。服务间通讯的环节就从应用里面剥离出来，呈现出一个抽象层。

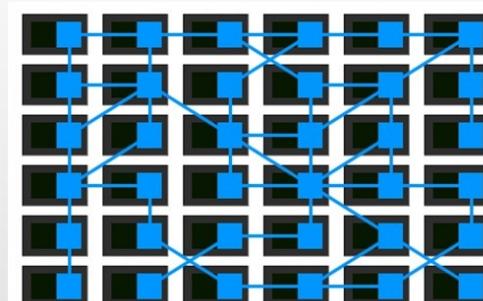
部署模型：有大量服务，表现为网络



如果有大量的服务，就会表现出来网格。图中左边绿色方格是应用，右边蓝色的方框是Service Mesh，蓝色之间的线条是表示服务之间的调用关系。Sidecar之间的连接就会形成一个网络，这个就是服务网格名字的由来。这个时候代理体现出来的就和前面的sidecar不一样了，形成网状。

Service Mesh定义回顾

- 抽象：基础设施层
- 功能：实现请求的可靠传递
- 部署：轻量级网络代理
- 关键：对应用程序透明



不再将代理视为单独的组件，而是强调由这些代理连接而形成的网络

再来看看前面给出的定义，大家回头看这四个关键词。首先第一个，服务网格是抽象的，实际上是抽象出了一个基础设施层，在应用之外。其次，功能是实现请求的可靠传递。部署上体现为轻量级的网络代理。最后一个关键词是，对应用程序透明。

大家注意看，上面的图中，网络在这种情况下，可能不是特别明显。但是如果把左边的应用程序去掉，现在只呈现出来Service Mesh和他们之间的调用，这个时候关系就会特别清晰，就是一个完整的网络。这是Service Mesh定义当中一个非常重要的关键点，和Sidecar不相同的地方：不再将代理视为单独的组件，而是强调由这些代理连接而形成的网络。在Service Mesh里面非常强调代理连接组成的网络，而不像sidecar那样看待个体。

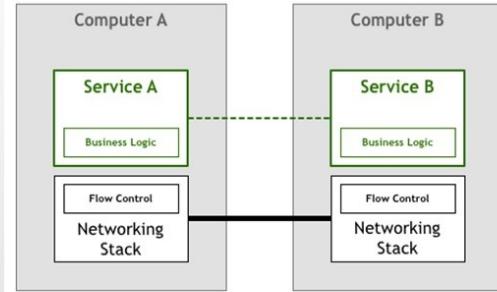
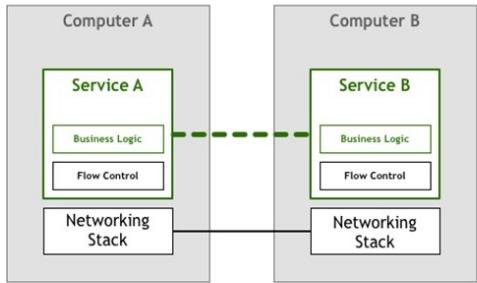
现在我们基本上把Service Mesh的定义介绍清楚了，大家应该可以大概了解什么是Service Mesh了。



Service Mesh的演进历程

第二个部分和大家追溯一下Service Mesh的演进历程。要注意，虽然Service Mesh这个词汇直到2016年9才有，但是它表述的东西很早以前就出现了。

Service Mesh的演进1：微服务之前



远古时代：第一代网络计算机系统

开发人员需要在自己的代码里处理网络通讯的细节问题，如数据包顺序，流量控制等。结果就是应用程序需要处理网络逻辑，导致网络逻辑和业务逻辑混杂在一起。

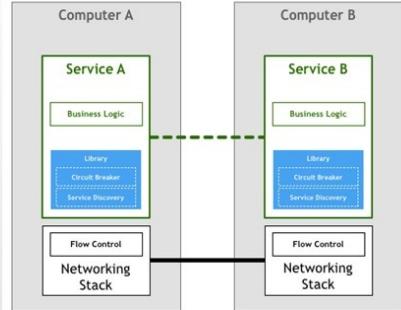
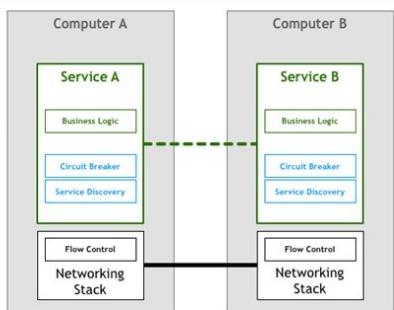
TCP/IP出现

解决了流量控制等问题。尽管网络逻辑的代码依然存在，但已经从应用程序里抽取出来，成为操作系统网络层的一部分。应用程序和开发人员得以解脱😊

首先看“远古时代”，第一代网络计算机系统，最早的时候开发人员需要在自己的代码里处理网络通讯的细节问题，比如说数据包顺序、流量控制等等，导致网络逻辑和业务逻辑混杂在一起，这样是不行的。接下来出现了TCP/IP技术，解决了流量控制问题，从右边的图上可以看到，功能其实没发生变化：所有的功能都在，代码还是要写。但是，最重要的事情，流程控制，已经从应用程序里面抽出来了。对比左边两边的图，抽出来之后被做成了操作系统网络层的一部分，这就是TCP/IP，这样的话应用的结构就简单了。

现在写应有，就不用考虑网卡到底怎么发。在TCP/IP之后，这是完全不需要考虑的。上面说的是非常遥远的事情，大概发生在五十年前。

Service Mesh的演进2：微服务时代



第一代微服务架构

开发人员需要在自己的代码里处理一系列问题，如服务发现，负载均衡，熔断，重试等。导致应用程序中，在业务逻辑外混杂大量非功能的代码。

类库和框架出现

典型如Netflix OSS套件，Spring Cloud框架，开发人员只要写少量代码，甚至几个注解就搞定。Spring Cloud因此风靡一时，几乎成为微服务的代名词。好像一切都很完美样子？:)

微服务时代也面临着类似的一些东西，比如说我们在做微服务的时候要处理一系列的比较基础的事情，比如说常见的服务注册、服务发现，在得到服务器实例之后做负载均衡，为了保护服务器要熔断/重试等等。这些功能所有的微服务都跑不掉，那怎么办呢？只能写代码，把所有的功能写进来。我们发现最早的微服务又和刚才一样，应用程序里面又加上了大量的非功能性的代码。为了简化开发，我们开始使用类库，比如说典型的Netflix OSS套件。在把这些事情做好以后，开发人员的编码问题就解决了：只需要写少量代码，就可以把这些功能实现。因为这个原因，最近这些年大家看到Java社区Spring Cloud的普及程度非常快，几乎成为了微服务的代名词。

到了这个地步之后，完美了吗？当然，如果真的完美了，那我今天就不会站在这里了:)



我们看这几个被称之为痛点的东西：内容比较多，门槛比较高。调查一下，大家学Spring Cloud，到你能熟练掌握，并且在产品当中应用，可以解决出现的问题，需要多长时间？一个星期够不够？大部分人一个星期是不够的，大部分人需要三到六个月。因为你在真实落地时会遇到各种问题，要能自己解决的话，需要的时间是比较长的。这里是Spring Cloud的常见子项目，只列出了最常见的部分，其中spring cloud netflix下还有netflix OSS套件的很多内容。要真正吃透Spring Cloud，需要把这些东西全部吃透，否则遇到问题时还会非常难受。

这么多东西，在座的各位相对来说学习能力比较强一点，可能一个月就搞定了，但是问题是你的开发团队，尤其是业务开发团队需要多久，这是一个很要命的事情：业务团队往往有很多比较初级的同事。

雪上加霜：不得不面对的诸多现实

业务开发团队的强项往往不在技术，而是对业务的理解

应用的核心价值在于业务实现：微服务是手段，不是目标

比学习微服务框架更艰巨的挑战：微服务拆分，API设计，数据一致性，康威定律

对于旧有系统，如何进行微服务改造，是个更加痛苦的考验

业务开发团队往往承受极大的业务压力，时间人力永远不足

然后事情并不止这么简单，所谓雪上加霜，我们还不得不面对一堆现实。

比如说，我们的业务开发团队的强项是什么？最强的会是技术吗？不，通常来说我们的业务开发团队最强的是对业务的理解，是对整个业务体系的熟悉程度。

第二个事情，业务应用的核心价值是什么？我们辛辛苦苦写了这么多的微服务，难道是为了实现微服务吗？微服务只是我们的手段，我们最终需要实现的是业务，这是我们真正的目标。

第三个事情是，就微服务这个手段而言，有比学习微服务框架更艰巨的挑战。在做微服务的真正落地时，会有更深刻的理解。比如微服务的拆分，比如要设计一个良好的API，要保持稳定并且易于扩展，还有如果涉及到跨多个服务的数据一致性，大部分团队都会头疼。最后是康威定律，但凡做服务的同学最终都会遇到这个终极问题，而大多数情况下是欲哭无泪。

但是这些还没完，比你写一个新的微服务系统更痛苦的事情，是你要对旧有的系统进行微服务改造。

所有这些加在一起，还不够，还要再加一条，这条更要命：业务开发团队往往业务压力非常大，时间人力永远不足。说下月上线就是下月上线，说双十一促销就不会推到双十二。老板是不会管你有没有时间学习spring cloud的，也不会管你的业务团队能否搞得定微服务的方方面面。业务永远看的是结果。

痛点2：服务治理功能不够齐全

• 基本功能

- 服务注册与服务发现
 - 主动健康检查
- 负载均衡
 - 随机轮询之外的高级算法
- 故障处理和恢复
 - 超时
 - 熔断
 - 限流
 - 重试
- RPC支持
- HTTP/2支持
- 协议转换/提升

• 高级功能

- 加密
 - 密钥和证书的生成，分发，轮换和撤销
- 认证/授权/鉴权
 - OAuth
 - 多重授权机制
 - ABAC
 - RBAC
 - 授权钩子
- 分布式追踪/APM
- 监控
 - 日志
 - 度量 (Metrics)
 - 仪器仪表 (instrumentation)

• 运维测试类

- 动态请求路由
- 服务版本
- 分段服务(staging service)
- 金丝雀(canaries)
- A/B测试
- 蓝绿部署(blue-green deploy)
- 跨DC故障切换
- 黑暗流量(dark traffic)
- 故障注入
- 高级路由支持
 - 高度可定制：script, DSL
 - 可灵活配置的规则，即时生效

你打算投多少时间和精力进去？

第二个痛点，功能不够，这里列出了服务治理的常见功能。而Spring Cloud的治理功能是不够强大的，如果把这些功能一一应对做好，靠Spring Cloud直接提供的功能是远远不够的。很多功能都需要你在Spring Cloud的基础上自己解决。

问题是打算投入多少时间人力资源来做这个事情。有些人说我大不了有些功能我不做了，比如灰度，直接上线好了，但是这样做代价蛮高的。

痛点3：说好的跨语言呢？

微服务带来的一个巨大优势，就是容许不同的服务根据实际需要采用不同的编程语言。但是，当我们将代码封装到类库和框架时，有个小问题冒出来了②：

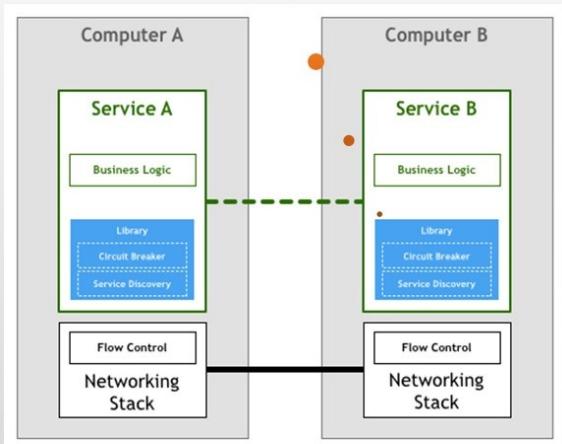
我们需要为多少种语言提供类库？

• 主流编程语言

- Java
 - Scala
 - Groovy
 - Kotlin
- C
- C++
- C#
- Python
- PHP
- Ruby

• 新兴编程语言

- Golang
- Node.js
- Rust
- R
- Lua/OpenResty



第三个痛点，跨语言。微服务在刚开始面世的时候，承诺了一个很重要的特性：就是不同的微服务可以采用自己最擅长最喜欢的最适合的编程语言来编写，这个承诺只能说有一半是OK的，但是另外一半是不行的，是假的。因为你实现的时候，通常来说是基于一个类库或者框

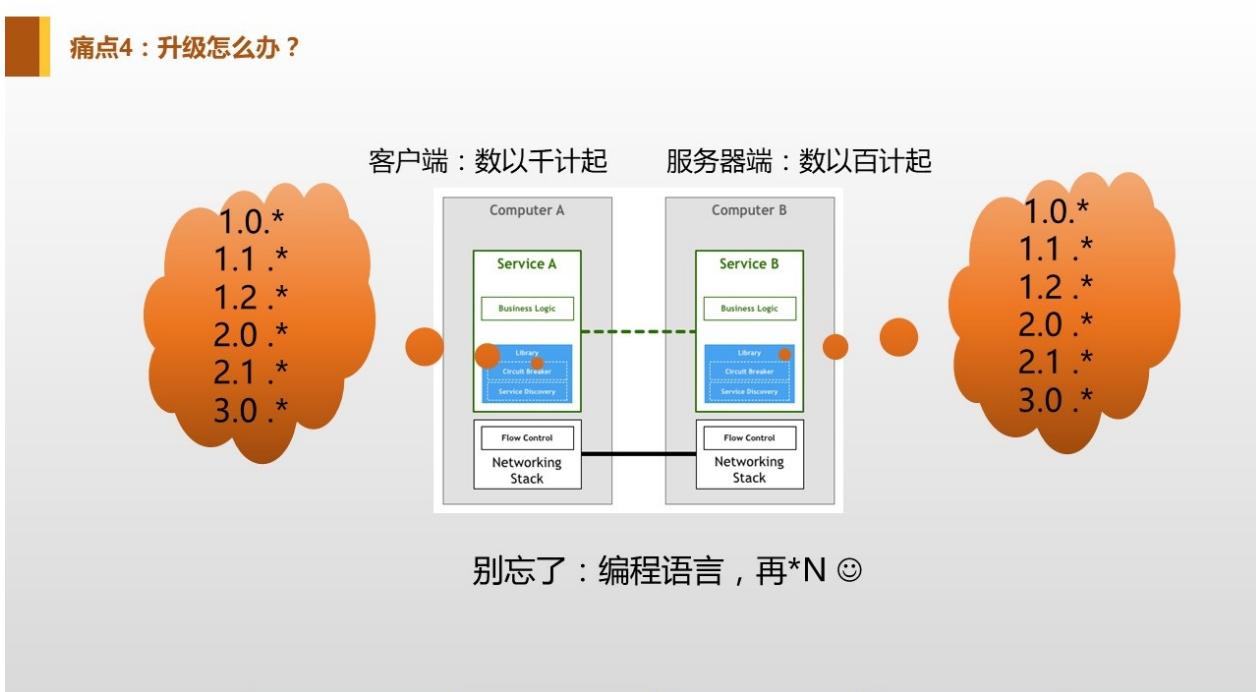
架来实现的，一旦开始用具体编程语言开始编码的时候你就会发现，好像不对了。为什么？左边是我从编程语言排行列表出来的主流编程语言，排在前面的几种，大家比较熟悉。后面还有几十种没有列出来，中间是新兴的编程语言，比较小众一点。

现在的问题在于我们到底要为多少种语言提供类库和框架。

这个问题非常尖锐，为了解决这个问题，通常只有两条路可选：

1. 一种就是统一编程语言，全公司就用一种编程语言
2. 另外一个选择，是有多少种编程语言就写多少个类库

我相信在座的如果有做基础架构的同学，就一定遇到过这个问题。



但是问题还没完，框架写好了，也有能够把各个语言都写一份。但是接下来会有第四个痛点：版本升级。

你的框架不可能一开始就完美无缺，所有功能都齐备，没有任何BUG，分发出去之后就再也不需要改动，这种理想状态不存在的。必然是1.0、2.0、3.0慢慢升级，功能逐渐增加，BUG逐渐被修复。但是分发给使用者之后，使用者会不会立马升级？实际上做不到的。

这种情况下怎么办，会出现客户端和服务端版本不一致，就要非常小心维护兼容性，然后尽量督促你的使用者：我都是3.0了，你别用1.0了，你赶紧升级吧。但是如果如果他不升级，你就继续忍着，然后努力解决你的版本兼容性。

版本兼容性有多复杂？服务端数以百计起，客户端数以千计起，每个的版本都有可能不同。这是一个笛卡尔乘积。但是别忘了，还有一个前面说的编程语言的问题，你还得再乘个N！

设想一下框架的Java1.0客户端访问node.js的3.0的服务器端会发生什么事情，c++的2.0客户端访问golang的1.0服务器端会如何？你想把所有的兼容性测试都做一遍吗？这种情况下你的兼容性测试需要写多少个case，这几乎是不可能的。



那怎么办？怎么解决这些问题，这是现实存在的问题，总是要面对的。

我们来想一想：

1. 第一个问题是这些根源在哪里：我们做了这么多痛苦的事情，面临这么多问题，这么多艰巨的挑战，这些和服务本身有关系吗？比如写一个用户服务，对用户做CRUD操作，和刚才说的这些东西有一毛钱关系吗？发现有个地方不对，这些和服务本身没关系，而是服务间的通讯，这才是我们需要解决的问题。
2. 然后看一下我们的目标是什么。我们前面所有的努力，其实都是为了保证将客户端发出的业务请求，发去一个正确的地方。什么是正确的地方？比如说有版本上的差异，应该去2.0版本，还是去1.0版本，需要用什么样的负载均衡，要不要做灰度。最终这些考虑，都是让请求去一个你需要的正确的地方。
3. 第三个，事情的本质。整个过程当中，这个请求是从来不发生更改的。比如我们前面说的用户服务，对用户做CRUD，不管请求怎么走，业务语义不会发生变化。这是事情的本质，是不发生变化的东西。
4. 这个问题具有一个高度的普适性：所有的语言，所有的框架，所有的组织，这些问题对于任何一个微服务都是相同的。

讲到这里，大家应该有感觉了：这个问题是不是和哪个问题特别像？

类比TCP

是不是有一种似曾相识的感觉？为什么TCP会出现？它是怎么解决问题的？

我们基于TCP开发应用时，应用需要关心链路层吗？

我们基于HTTP开发应用时，应用需要关心TCP层吗？

五十年前的前辈们，他们要解决的问题是什么？为什么会出现TCP，TCP解决了什么问题？又是怎么解决的？

TCP解决的问题和这个很像，都是要将请求发去一个正确的地方。所有的网络通讯，只要用到TCP协议，这四个点都是一致的。

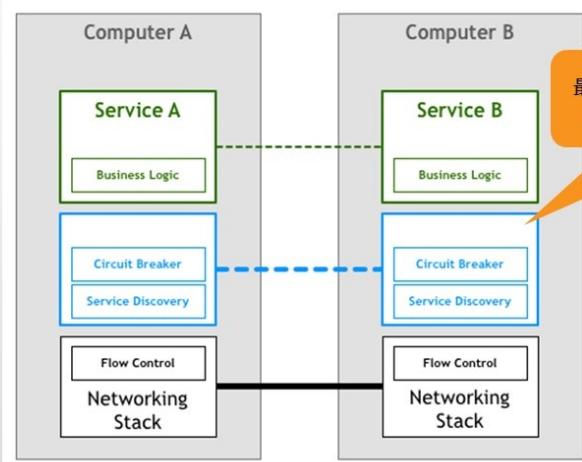
有了TCP之后会发生什么？我们有了TCP之后，我们基于TCP来开发我们的应用，我们的应用需要做什么事情？我们的应用需要关心TCP层下链路层的实现吗？不需要。同理，我们基于HTTP开发应用时，应用需要关心TCP层吗？

反省

为什么我们开发微服务应用时，就一定要这么关心服务通讯层？

为什么我们开发微服务应用的时候就要这么关心服务的通讯层？我们把服务通讯层所有的事情学一遍，做一遍，我们做这么多是什么？

Service Mesh的演进3：技术栈下移



最理想的做法是将这层加入网络协议栈，但是这个实现起来不现实

- 先驱者：使用代理解决部分问题

使用nginx, HaProxy, Apache等反向代理，避免服务间产生直接连接。所有流量都经由代理，代理实现需要的特性如负载均衡。

但是：功能过于简陋

这种情况下，自然产生了另外一个想法：既然我们可以把网络访问的技术栈向下移为TCP，我们是可以也有类似的，把微服务的技术栈向下移？

最理想的状态，就是我们在网络协议层中，增加一个微服务层来完成这个事情。但是因为标准问题，所以现在没有实现，暂时这个东西应该不太现实，当然也许未来可能出现微服务的网络层。

之前有一些先驱者，尝试过使用代理的方案，常见的nginx, haproxy, apache等代理。这些代码和微服务关系不大，但是提供了一个思路：在服务器端和客户端之间插入了一个东西完成功能，避免两者直接通讯。当然代理的功能非常简陋，开发者一看，想法不错，但是功能不够，怎么办？

Service Mesh的演进4：Sidecar出现

- **Airbnb**

2013年，Airbnb开发了Synapse和Nerve

- **Netflix**

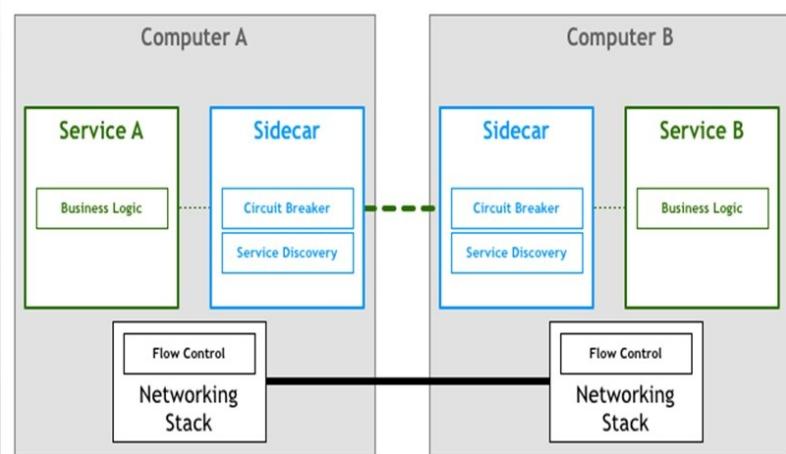
2014年，Netflix发布了Prana

- **SoundCloud**

据说也开发了一些sidecar

- **唯品会**

2015年，唯品会的OSP服务化框架，加入名为local proxy的sidecar



这种情况下，第一代的Sidecar出现了，Sidecar扮演的角色和代理很像，但是功能就齐全很多，基本上原来微服务框架在客户端实现的功能都会对应实现。

第一代的sidecar主要是列出来的这几家公司，其中最有名气的还是netflix。

在这个地方我们额外提一下，注意第四个，前面三个功能都是国外的公司，但是其实sidecar这个东西并不是只有国外的人在玩，国内也有厂商和公司在做类似的事情。比如唯品会，我当年在唯品会基础架构部工作的时候，在2015年上半年，我们的OSP服务化框架做了一个重大架构调整，加入了名为local proxy的Sidecar。注意这个时间是2015上半年，和国外差不多。相信国内肯定还有类似的产品存在，只是不为外界所知。

Sidecar的局限：为特定的基础设施而设计

- **Airbnb**

Synapse和Nerve：假设服务一定是注册到ZooKeeper上的

- **Netflix**

Prana：一定要使用Netflix自己的Eureka，目的是让非JVM应用接入Netflix OSS

- **SoundCloud**

让遗留的Ruby应用可以使用JVM的基础设施

- **唯品会**

绑定OSP框架和其他内部基础设施，接入非Java语言，解决业务部门不愿意升级的问题

注：2015年中，曾有将Local Proxy从OSP剥离，改造为通用sidecar的设想，计划支持HTTP1.1，可惜未能实现，遗憾

有特定
背景和需求

无法通用

这个时期的Sidecar是有局限性的，都是为特定的基础设施而设计，通常是和当时开发Sidecar的公司自己的基础设施和框架直接绑定的，在原有体系上搭出来的。这里面会有很多限制，一个最大的麻烦是无法通用：没办法拆出来给别人用。比如Airbnb的一定要用到zookeeper，netflix的一定要用eureka，唯品会的local proxy是绑死在osp框架和其他基础设施上的。

之所以出现这些绑定，主要原因还是和这些sidecar出现的动机有关。比如netflix是为了让非JVM语言应用接入到Netflix OSS中，soundcloud是为了让遗留的Ruby应用可以使用到JVM的基础设置。而当年我们唯品会的OSP框架，local proxy是为了解决非Java语言接入，还有前面提到的业务部门不愿意升级的问题。这些问题都比较令人头疼的，但是又不得不解决，因为逼的憋出来sidecar这个一个解决方式。

因为有这样的特殊的背景和需求，所以导致第一代的Sidecar无法通用，因为它本来就是做在原有体系之上的。虽然不能单独拿出来，但是在原有体系里面还是可以很好工作的，因此也没有动力做剥离。导致虽然之前有很多公司有Sidecar这个东西，但是其实一直没怎么流传出来，因为即使出来以后别人也用不上。

这里提一个事情，在2015年年中的时候，我们当时曾经有一个想法，将Local proxy从OSP剥离，改造为通用的Sidecar。计划支持HTTP1.1，操作http header就可以，body对我们是可以视为透明的，这样就容易实现通用了。可惜因为优先级等原因未能实现，主要是有大量的其他工作比如各种业务改造，这个事情必要性不够。

所有比较遗憾，当时我们有这个想法没做实现，这是在2015年，时间点非常早的了。如果当时有实现，很可能我们就自己折腾出业界第一个service mesh出来了。现在想想挺遗憾的。

Service Mesh的演进5：通用型的Service Mesh出现



Linkerd

- 来自buoyant，Scala语言
- Service Mesh名词的创造者
- 2016年1月15日，0.0.7发布
- 2017年1月23日，加入CNCF
- 2017年4月25日，1.0版本发布

Envoy

- 来自Lyft，c++语言
- 2016年9月13日，1.0版本发布
- 2017年9月14日，加入CNCF

nginxmesh

- 来自nginx
- 2017年9月15日发布
- 目前只有一个版本0.16

但是，不只有我们会有这想法。还有有一些人想法和我们差不多，但是比较幸运的是，他们有机会把东西做出来了。这就是第一代的Service Mesh，通用性的sidecar。

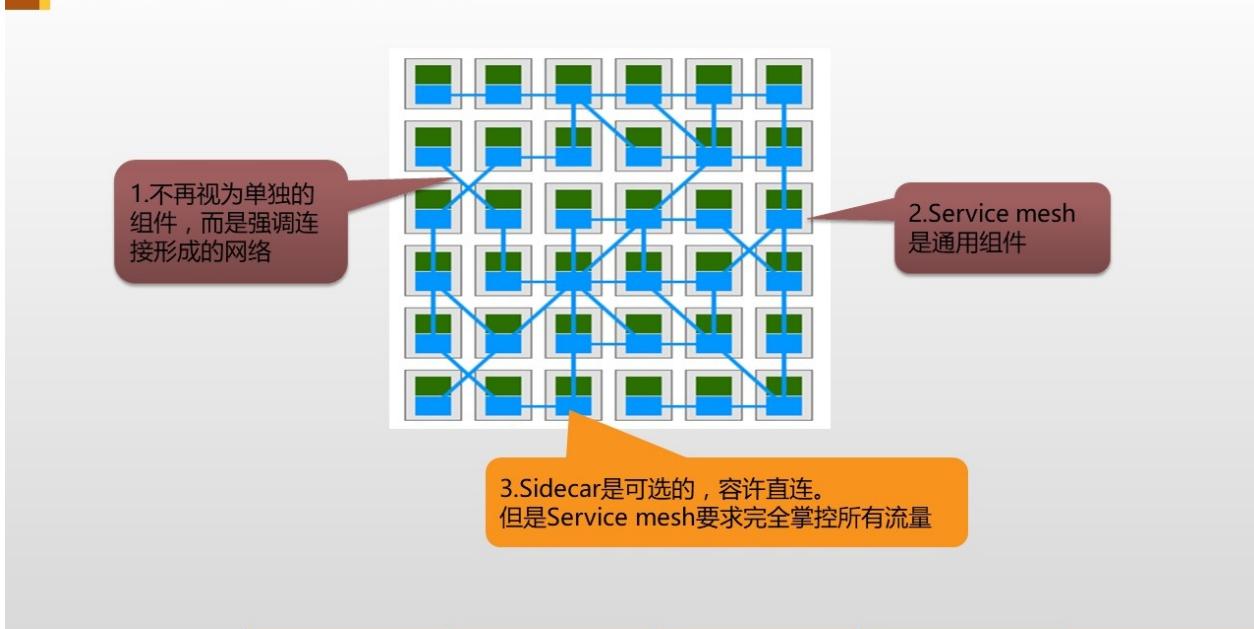
通用型的Service Mesh的出现，左边第一个Linkerd是业界第一个Service Mesh，也就是它创造了Service Mesh这个词。时间点：2016年1月15号，0.0.7发布，这是github上看到的最早的一个版本，其实这个版本离我们当时的有想法的时间点非常近。然后是1.0版本，2017年4月份发布，离现在六个月。所以说，Service Mesh是一个非常新的名词，大家没听过非常正常。

接下来是Envoy，2016年发布的1.0版本。

这里面要特别强调，Linkerd和Envoy都加入了CNCF，Linkerd在今年1月份，而Envoy进入的时间是9月份，离现在也才1个月。在座的各位应该都明白CNCF在Cloud Native领域是什么江湖地位吧？可以说CNCF在Cloud Native的地位，就跟二战后联合国在国际秩序中的地位一样。

之后出现了第三个Service Mesh，nginmesh，来自于大家熟悉的nginx，2017年9月发布了第一个版本。因为实在太新，还在刚起步，没什么可以特别介绍的。

Service Mesh和Sidecar的差异总结



我们来看一下Service Mesh和Sidecar的差异，前面两点是已经提到了：

1. 首先Service Mesh不再视为单独的组件，而是强调连接形成的网络
2. 第二Service Mesh是一个通用组件

然后要强调的是第三点，Sidecar是可选的，容许直连。通常在开发框架中，原生语言的客户端喜欢选择直连，其他语言选择走sidecar。比如java写的框架，java客户端直连，php客户端走sidecar。但是也可以都选择走sidecar，比如唯品会的OSP就是所有语言都走local proxy。在sidecar中也是可选的。但是，Service Mesh会要求完全掌控所有流量，也就是所有的请求都必须通过service mesh。

Service Mesh的演进6：Istio王者风范



Istio

- 来自Google, IBM和Lyft, Go语言
- Service Mesh集大成者
- 绝对的新鲜出炉
 - 2017年5月24日, 0.1 release版本发布
 - 2017年10月4日, 0.2 release版本发布

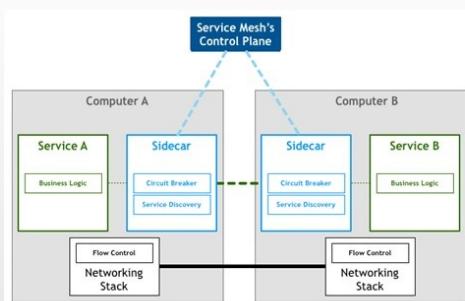


接下来给大家介绍Istio，这个东西我给它的评价是王者风范，来自于谷歌、IBM和Lyft，是Service Mesh的集大成者。

大家看它的图标，就是一个帆船。Istio是希腊语，英文语义是"sail", 翻译过来是起航的意思。大家看这个名字和图标有什么联想？google在云时代的另外一个现象级产品，K8S，kubernetes也同样起源于希腊语，船长，驾驶员或者舵手，图标是一个舵。

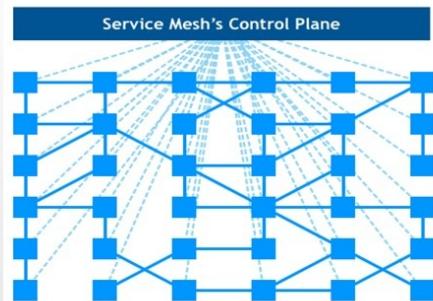
Istio名字和图标与k8s可以说是一脉相承的。这个东西在2017年5月份发布了0.1，就在两周前的10月4号发布了0.2。大家都熟悉软件开发，应该明白0.1/0.2在软件迭代中是什么阶段。0.1大概相当于婴儿刚刚出世，0.2还没断奶。但是，即使在这么早期的版本中，我对他的评价已经是集大成者，王者风范，为什么？

Istio带来的是前所未有的控制力



单个视图

以sidecar方式部署的Service Mesh控制了服务间流量，理论上Service Mesh可以对流量“为所欲为”，因此，只要能控制Service Mesh，一切就皆有可能。



整体视图

- 所有的流量都在Service Mesh控制中
- 所有的Service Mesh都在控制面板掌控
- 通过控制面板就可以控制整个系统

Istio为此带来了集中式的控制面板。

为什么说Istio王者风范？最重要的是他为Service Mesh带来了前所未有的控制力。以Sidecar方式部署的Service Mesh控制了服务间所有的流量，只要能够控制Service Mesh就能够控制所有的流量，也就可以控制系统中的所有请求。为此Istio带来了一个集中式的控制面板，让你实现控制。

左边是单个视图，在sidecar上增加了控制面板来控制sidecar。这个图还不是特别明显，看右边这个图，当有大量服务时，这个服务面板的感觉就更清晰一些。在整个网络里面，所有的流量都在Service Mesh的控制当中，所有的Service Mesh都在控制面板控制当中。可以通过控制面板控制整个系统，这是Istio带来的最大的革新。

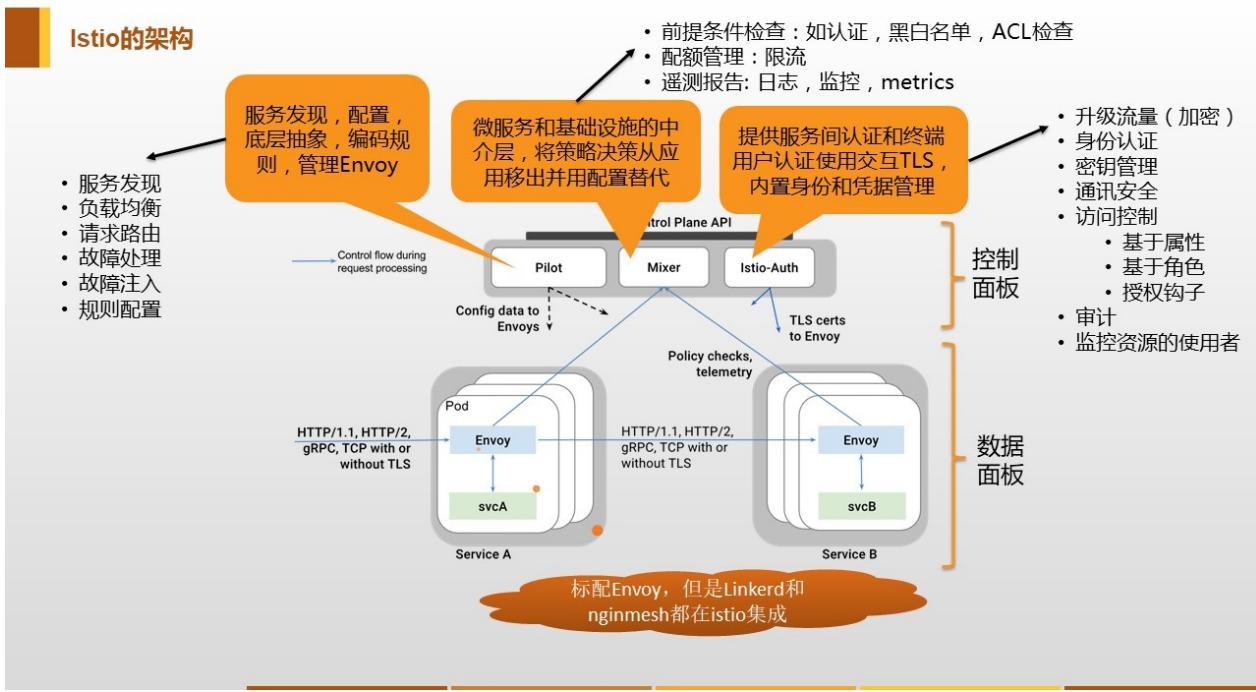


Istio由三个公司开发，前两个比较可怕，谷歌和IBM，而且都是云平台，谷歌的云平台，IBM的云平台，尤其GCP的大名想必大家都知道。所谓出身名门，大概指的就是这个样子吧？

Istio的实力非常强，我这里给了很多的赞誉：设计理念非常新颖前卫，有创意，有魄力，有追求，有格局。Istio的团队实力也非常惊人，大家有空可以去看看istio的委员会名单感受一下。Istio也是google的新的重量级的产品，很有可能成为下一个现象级的产品。Google现在的现象级产品是什么？K8s。而Istio很有可能成为下一个K8S级别的产品。

说到应时而生，什么是势？我们今天所在的是什么时代？是互联网技术大规模普及的时代，是微服务容器如日中天的时代，是Cloud Native大势已成的时代。也是传统企业进行互联网转型的时代，今天的企业用户都想转型，这个大势非常明显，大家都在转或者准备转，但是先天不足。什么叫先天不足？没基因，没能力，没经验，没人才，而且面临我们前面说的所有痛点。所有说Istio现在出现，时机非常合适。别忘了istio身后还有CNCF的背景，已经即将一统江湖的k8s。

istio在发布之后，社区响应积极，所谓应者云集。其中作为市面上仅有的几个Service Mesh之一的Envoy，甘心为istio做底层，而另外两个实现Linkerd/nginxmesh也直接放弃和istio的对抗，选择合作，积极和istio做集成。社区中的一众大佬，如这里列出来的，都在第一时间响应，要和istio做集成或者基于istio做自己的产品。为什么说是第一时间？istio出0.1版本，他们就直接表明态度开始战队了。



Istio的架构，主要分为两大块。下面的数据面板，是给传统service mesh的，目前是Envoy，但是我们前面也提到linkerd和nginxmesh都在和istio做集成，指的就是替代Envoy做数据面板。

另外一大块就是上面的控制面板，这是istio真正带来的内容。主要分成三大块，图中我列出了他们各自的职责和可以实现的功能。

因为时间有限，不在今天具体展开。

Istio的更多资料

- [万字解读:Service Mesh服务网格新生代--Istio](#)

今年9月21号我的一个线上分享，对istio进行了详细介绍，想了解更多istio细节的朋友可以看看

- [Istio官方文档中文翻译](#)

我和Service Mesh社区的朋友目前正在进行中的istio官方文档中文翻译计划

这里给大家留一个地址，是我之前做的一次线上分享，对Istio的详细介绍，内容比较多，大家看看仔细看看。

- [万字解读:Service Mesh服务网格新生代--Istio](#)

然后我们还组织了一个service mesh的技术社区，对istio的文档进行了翻译。

- [Istio官方文档中文翻译](#)

Service Mesh演进总结

Sidecar

- 2013年，Airbnb，Synapse和Nerve
- 2014年，Netflix，Prana
- 2015年，唯品会，OSP Local Proxy

远古时代

2013/2014/2015

2016/2017

2018

Istio

- 2017年5月，0.1发布
- 2017年10月，0.2发布
- 预计2018年，1.0发布，期待中

下一代
微服务

原始代理

- Nginx/Apache
- HAProxy

2016年9月"service
mesh"第一次公开使用

Service Mesh

- 2016年1月，Linkerd发布第一个版本
- 2016年9月，Envoy发布1.0版本
- 2017年1月，Linkerd加入CNCF
- 2017年4月，Linkerd发布1.0版本
- 2017年9月，Envoy加入CNCF

总结一下，service mesh这是一步一步过来的：从原始的代理，到限制很多的Sidecar，再到通用性的Service Mesh，然后到加强管理功能的Istio，在未来成为下一代的微服务。

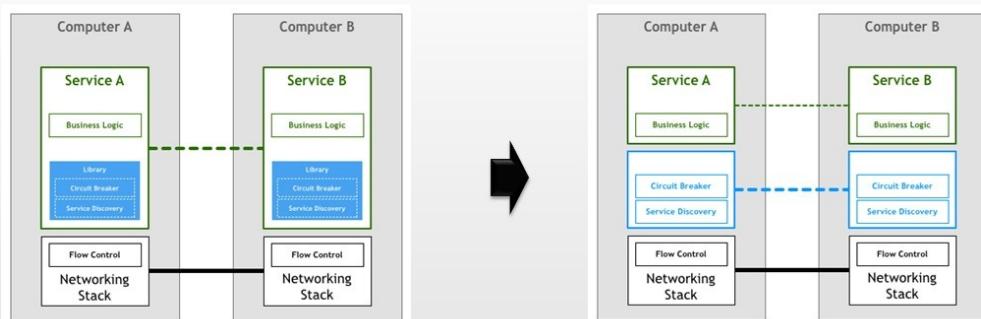
注意，离service mesh这个词汇出现的时间点，也才一年。

3

Why

为何选择Service Mesh ?

解决痛点之技术栈下移



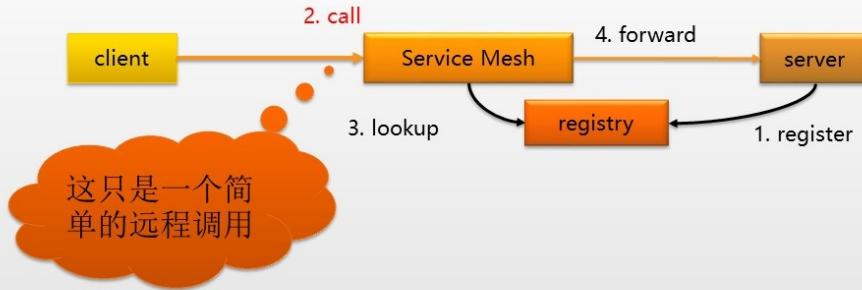
痛点1：内容多，门槛高：交给Service Mesh，应用只需关注业务逻辑

痛点2：服务治理功能不够齐全：Service Mesh把功能做齐全就好

痛点4：升级怎么办？Service Mesh可单独升级，应用程序不用改

前面三个痛点都被解决了，有了Service Mesh之后这些问题都不是问题了。升级的痛点怎么解决？Service Mesh是一个独立进程，可单独升级，而应用程序不用改。

解决痛点之代理接入



痛点3：说好的跨语言呢？可以真的可以自由选择，客户端极度简化，而服务器端只需实现服务注册

从此，只需要一套Service Mesh，就可以满足所有语言，而且功能完全一致

service mesh是以远程调用的方式让客户端接入，只要能发出请求，简单发给service mesh就可以。客户端极度简化，对于典型的rest请求，几乎所有的语言都有完善的支持。而服务器端只要做一个事情，服务注册。这样对于多语言的支持，就变得非常舒服了。现在终于可以真正的自由选择编程语言。

奇迹：鱼与熊掌兼得

对业务团队的技术要求降低
尤其对互联网技术转型的传统企业
开发测试成本降低

门槛降低

可以更多的关注微服务的其他领域
如微服务的拆分/API设计/数据一致性

远超Spring Cloud 的功能集
极大的满足各种类型的功能需求
统一化，标准化，集中式管理

功能增加

无需修改应用代码
灵活配置，按需定制

这里有一个奇迹，鱼与熊掌兼得：同时实现降低门槛，功能增加。有些信奉质量守恒的同学会感觉不科学，注意能同时实现这两个改进的原因，是把工作量最大最辛苦的事情都交给了Service Mesh。而Service Mesh是通用的，可以反复重用的。

颠覆性变革：解放业务开发团队**降低门槛**

减少了学习，开发，测试等一系列工作的时间和精力，技术转型成功的机会大增

**稳定基座**

避免自行开发可能遇到的各种bug，不稳定，性能瓶颈等，大幅降低风险

**加速转型**

无论是开发新业务，还是旧系统改型，大量的人力和时间得以从实施细节中解放出来，从而加速其他领域



Service mesh为业务开发团队带来的变革：降低入门门槛，提供稳定基座，帮助团队实现技术转型。最终达到的目的是，让业务开发团队从微服务实现的具体技术细节中解放出来，回归业务。

颠覆性变革：强化运维管理团队

Service Mesh最大的变革在于将服务间通讯以及于此相关的管理控制功能从业务程序中下移到基础设施

运维职能空前强化

前面列举的各个功能，都可以在控制面板中直接设置，无需编码，更无需业务程序事先准备

独立控制不再受限

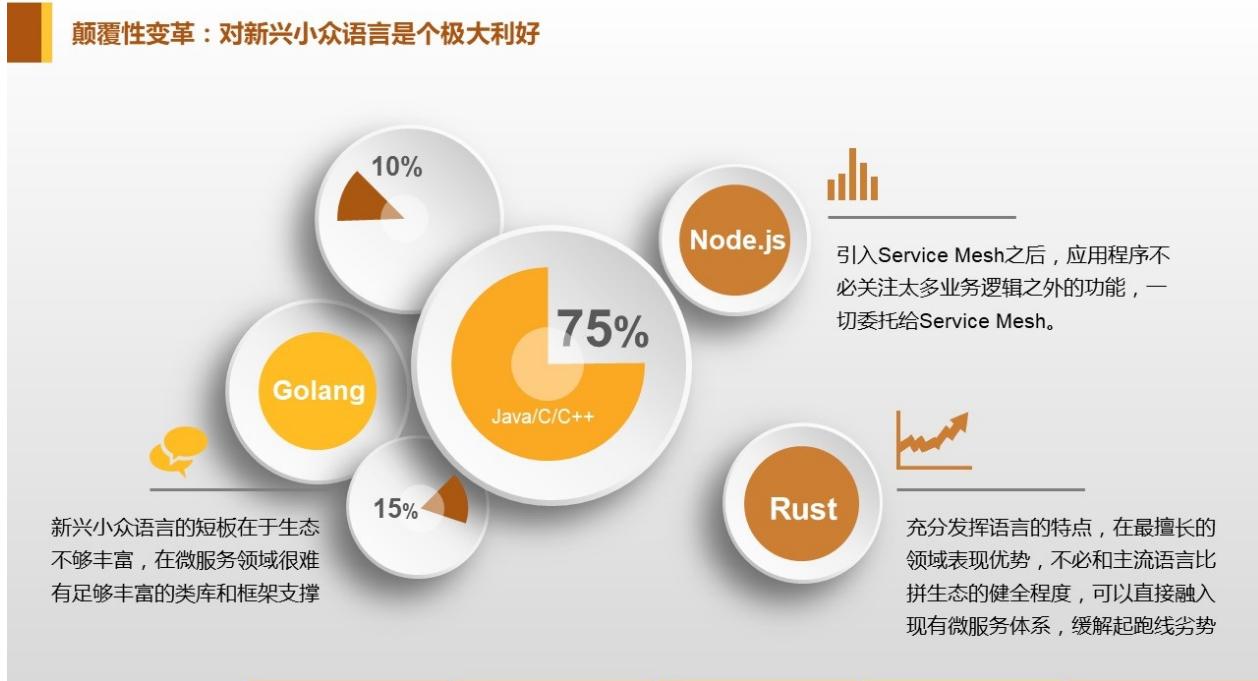
终于不再受限于业务程序代码，也就可以脱离业务开发团队自行控制系统行为，避免

及时响应快速变更

通过控制台直接配置，没有开发测试发布等整套流程，没有跨部门沟通合作的障碍

第二个变革，是对运维管理团队的强化，这里如果有做运维的同学，你们可以认真思考一下：如果有了service mesh，你们对系统的管理和控制力会有多大的？注意很多功能的实现已经不再和应用有关，都在移到service mesh中，而service mesh通常是在运维的掌控中。

颠覆性变革：对新兴小众语言是个极大利好



service mesh对于新兴小众语言是极大的利好。对于新的语言来说，在和传统的主流编程语言竞争时，最痛苦的事情是什么？是生态，比如各种类库，各种框架。在微服务这个领域，新兴小众语言想和Java等比拼，非常的难：这是用自己的劣势对上别人的优势。而有了Service Mesh之后，小众语言就有机会避开这个弊端，再不用和Java比拼生态，而是充分发挥自己的语言特点，做自己最擅长的领域。

让我们大声把口号喊出来

Service Mesh： 下一代微服务！

参考资料

- [WHAT' S A SERVICE MESH? AND WHY DO I NEED ONE?](#)

Willian Morgen对Service Mesh的解释，以及为什么云原生应用需要Service Mesh

- [Pattern: Service Mesh](#)

来自Phil Calçado的博客文章，详细解释了Service Mesh的由来，强烈推荐阅读。

注：本文引用了来自该文的大量图片，节约了大量画图的时间，**特别鸣谢**

- [模式之Service Mesh](#)

上文的中文翻译版本，译者薛命灯，翻译质量极佳，强烈推荐。

今天的内容基本上到这儿了，最后给两个资料，这两个文章，一个是对Service Mesh的解释，一个是详细介绍Service Mesh的由来，大家如果回去之后可以详细看一下。尤其第二个文章，我的PPT援引了里面的很多图片。英文不是特别好的同学可以看一下中文翻译版本，作者翻译质量非常高。

Service Mesh中国技术社区：欢迎加入



• 微信群

Service Mesh的纯技术交流群。由于技术过于新颖，大家都处于摸索阶段，互助交流尤其重要。非常欢迎大家加入一起探讨交流。

请扫描二维码(11月5日前有效)，或者联系微信ID **xiaoshu062** 申请加入。



• [ServiceMesh中文网 \(servicemesh.cn\)](http://servicemesh.cn)

Service Mesh中文技术交流网站，十一月正式开启。

最后，我们拉了一个service mesh的技术社区，有一个微信群。主要是因为Service Mesh这个技术实在是太新了，就像刚才调查的只有三个同学之前有听过。所以现在我们遇到一个大的问题：我们想交流的时候找不到人。因此我们建立了这个群，如果大家愿意对Service Mesh技术有兴趣的话，可以直接加到群来。这是一个纯技术的群，只讨论技术的东西。

然后我们的service mesh的技术社区目前正在准备中，不久将亮相，大家可以稍后关注。

我们今天的内容就到这里结束，非常感谢大家！

应用网络功能，**ESB**，**API**管理，而今..**Service Mesh**？

tags: Christian Posta

原文作者：[Christian Posta](#)

原文地址：[Application Network Functions With ESBs, API Management, and Now.. Service Mesh?](#)

转载地址：[原理解析Service Mesh与ESB、API管理与消息代理的关系](#)

译者：海松

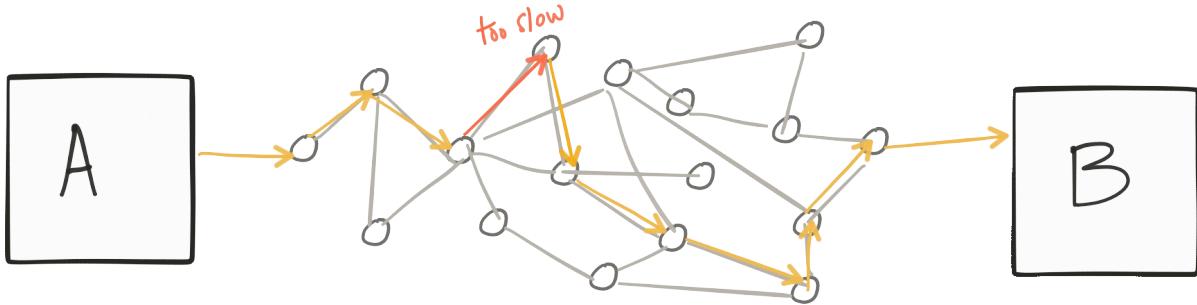
声明：转载自微信公众号EAWorld

最近我谈到了[微服务模式的发展](#)，以及像[service proxies like Envoy from Lyft](#)的文章，阐述服务代理如何将弹性、服务发现、路由、指标收集等职责推到应用程序下层。否则，我们只能寄希望于各种应用程序自己能正确实施这些关键功能，或依赖某个特定语言的库来实现这点。有趣的是，这种[service mesh](#)的思路与我们的企业领域客户熟悉的其他概念有关。我收到了很多关于这种关系的问题。具体来说，[service mesh](#)与ESB、消息代理和API Management的关系是什么？这些概念肯定有重叠，所以让我们来深挖一下。关注Twitter @christianposta了解更多信息！

四个假设

1. 服务通过网络进行通信

第一点：我们谈的服务是通过异步、包交换（packet-switched）网络进行通信和交互的服务。这意味着它们在自己的进程中运行，并且在自己的“时间边界”内（time boundaries，此处即异步的概念）中运行，且通过在网络中发送包进行通信。不幸的是，[没有任何东西可以保证异步网络交互](#)：我们最终可能会遇到交互失败、交互停滞/延迟等等，而且这些情况不好区分。



1. 如果仔细观察，这些交互并非微不足道

第二点：这些服务如何互相交互并非微不足道。我们必须处理以下事情，比如故障/部分成功、重试、重复检测、序列化/反序列化、语义/格式转换、多语言协议、路由到正确的服务来处理消息、处理消息洪泛（**floods of messages**）、服务编排、安全性影响等等。而其中许多事情可能而且必定会出错。

1. 了解网络非常重要

第三点：了解应用程序如何通信、消息如何交换以及如何控制流量非常重要。这点非常类似于我们如何看待第3/4层网络。了解以下内容是有价值的，如：哪些TCP段和IP数据包正穿过网络、控制路由它们和允许它们的规则等等。



1. 最终责任在应用程序

最后：正如通过[端到端的论证](#)我们所知道的，应用程序本身应当负责其业务逻辑的安全性，保证正确的语义实现。无论底层基础设施（即使会遇到重试、交易、重复检测等等）的可靠性如何，我们的应用程序必须防范用户做出糊涂的行为（如同一命令重复提交）。提供实现或优化的细节能有助于帮助实现。但不幸的是，没有办法解决这个问题。

应用程序网络功能

我认为，无论你更倾向于哪个服务架构（微服务、SOA、对象请求代理、客户端/服务器等），以上要点都是有效的。然而，过去我们对于哪些优化属于哪里十分模糊。在我看来，不仅横向的应用程序网络功能可以从应用程序中被优化掉（然后被投入到基础设施中 - 就像我

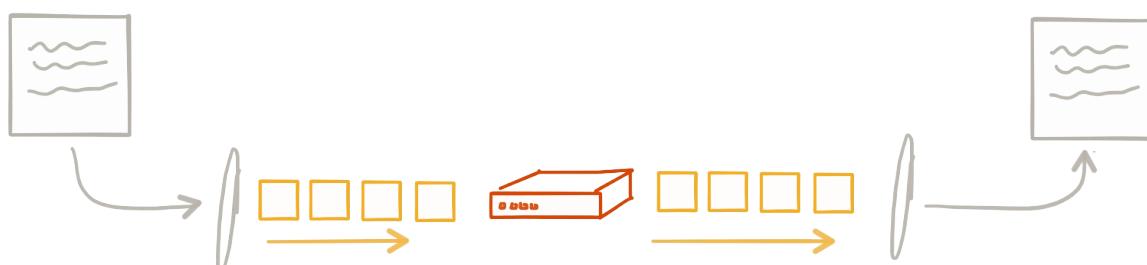
们在较低级别堆栈上所做的事情那样），而且还有其他应用程序网络功能与我们的业务逻辑更密切相关，但却不应该被轻易地“优化”（optimized）掉。

网络

让我们迅速回顾下应用程序之下的网络是什么样的，它可是地位超凡哦：）。当我们从一个服务发“消息”到另一个服务时，我们将其传递到了操作系统的网络堆栈，操作系统会尝试将这条消息放入网络中。根据网络所处级别，网络会处理传输单元（帧、数据报、数据包）等。这些传输单元通常包括一个由“数据头”和“有效负载”组成的结构，“数据头”包含关于传输单元的元数据。通过元数据，我们可以做很多基础的事情，例如路由、确认跟踪/去重等。



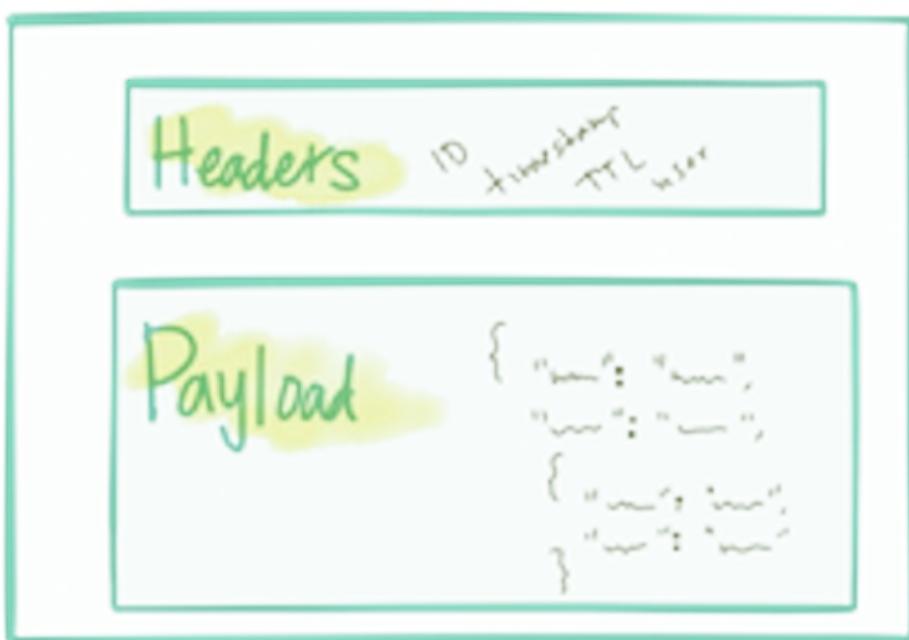
这些传输单元通过网络中的不同点进行发送，这些点决定了是否允许单元通过，是否将其路由到不同的网络，或将其传送到预期的接收者处。在路径上的任意一点，这些传输单元可能被丢弃、复制、重新排序或推迟。更高级的“可靠性”功能，如操作系统内网络堆栈中的TCP，则可以跟踪重复、确认、超时、排序、丢失的单元等东西，并可以进行故障重试、数据包重新排序等。



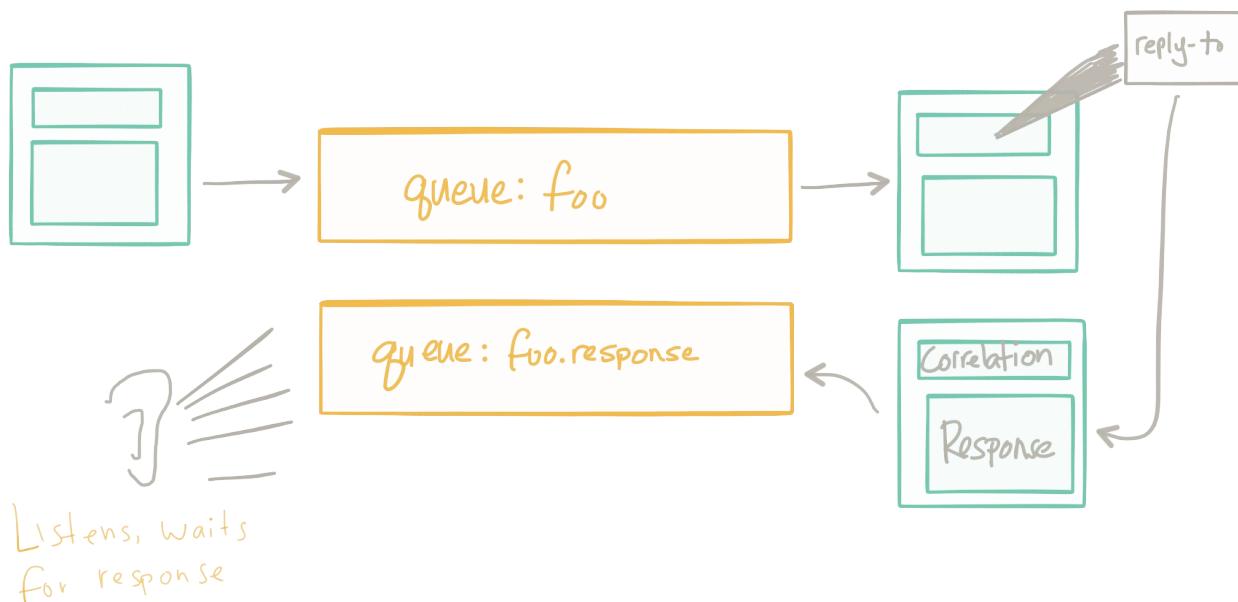
这类功能由基础设施提供，不与业务逻辑混合 – 而且其规模相当可观，达到了互联网规模！我刚看到的[Phil Calcado](#)精彩的博客很好地解释了这点。

Application

在应用程序级别上，我们在做类似的事情。我们将与协作者服务的对话拆分成包含“消息”（请求、事件等）的传输单元。当我们通过网络进行调用时，我们必须能为应用程序消息执行超时、重试、确认、应用背压（apply backpressure）等操作。这些都是应用程序级别普遍的问题，并且在构建服务架构时总会出现。我们需要解决它们。我们需要一种实现应用网络功能的方法。

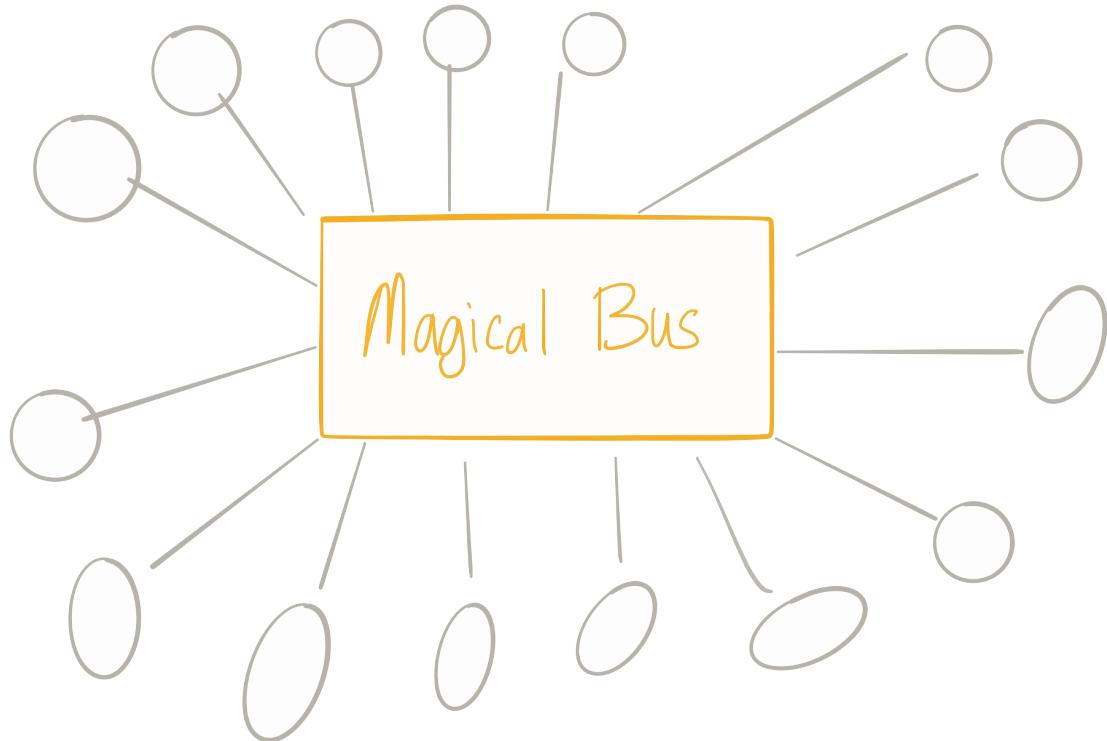


例如：过去我们尝试用邮件代理解决它们。有一组集中的面向消息的中间件（甚至可以通过多协议支持，使我们可以转换消息的有效负载，“集成”多个客户端），它们负责在客户端之间传递消息。我看到的很多例子使用的模式基本上是通过消息系统进行请求或回复（RPC）。



这间接帮助解决了应用程序网络功能中的一些问题。负载均衡、服务发现、背压、重试等工作都被委托给了消息代理。由于所有流量都要经过这些代理，所以我们有了一个中心点，从这个中心点可以观察和控制网络流量。但是，正如[@tef_ebooks](#)在Twitter上指出的那样，这种做法用力过度，有些矫枉过正了。它往往会成为架构的瓶颈，用它来进行流量控制、路由、策略执行等并不像我们想象那么容易。

所以我们也做过同样的尝试。我们觉得“好吧，就把路由、转换、策略控制”加到现有的集中式消息总线里吧。实际上这是个非常自然的演变。我们可以使用消息主干网（**messaging backbone**）来提供集中化、控制和应用程序网络功能，如服务发现、负载均衡、重试等等，但还要加入更多内容，比如协议调解、消息转换、消息路由、编排等功能，因为我们觉得如果可以将这些看似同一层面的内容加入到基础设施中，应用程序或许会更轻量、更精简、更敏捷等等。这些需求绝对是真实的，ESB演变并满足了这些需要。



正如我的同事Wolfram Richter指出，“关于ESB概念，IBM关于SOA架构的2005年白皮书（<http://signallake.com/innovation/soaNov05.pdf>第2.3.1章）提供了如下的定义：

在SOA逻辑架构中，企业服务总线（ESB）是名安静的伙伴。它在体系结构中的存在对于SOA应用程序的服务来说几乎是透明的。然而，ESB的存在是简化服务调用的基础，使我们能随时随地调用服务，而无需定位服务或是上传服务请求这些细节。

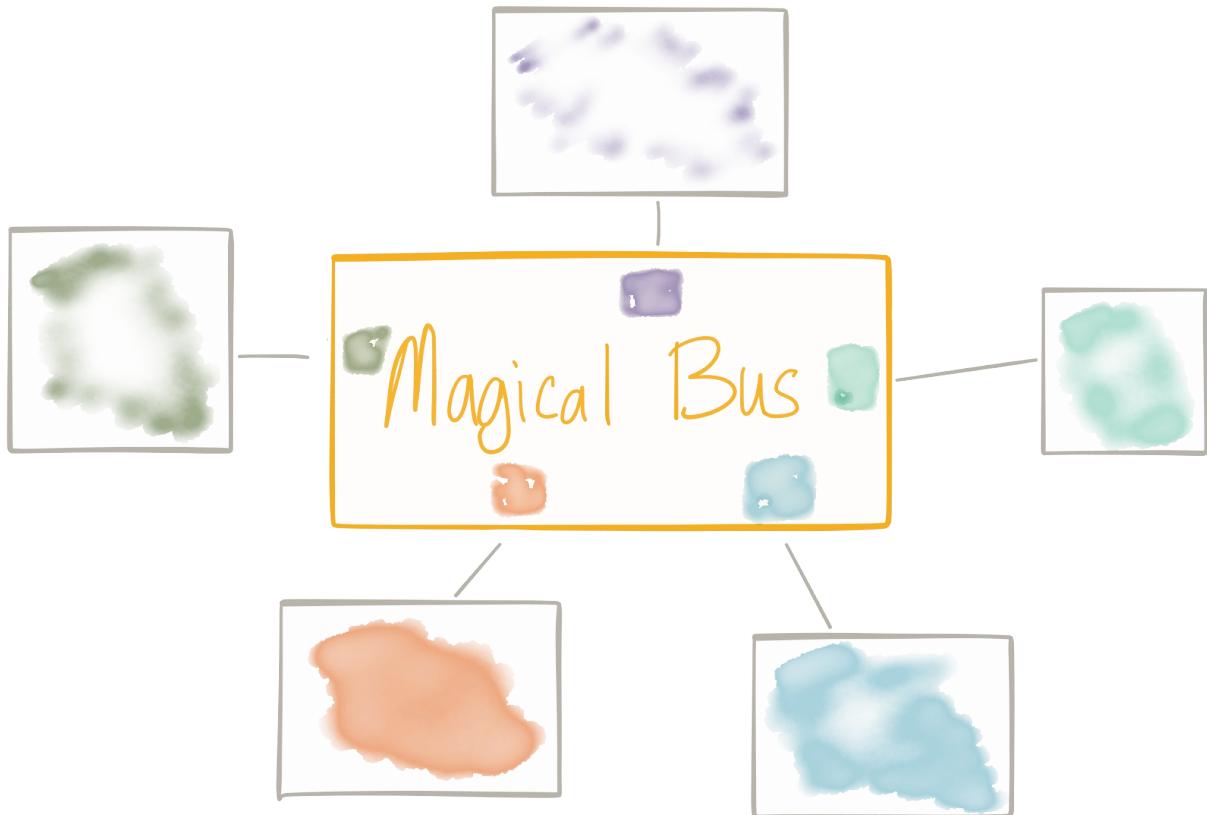
这似乎是行得通的！甚至像我们正在尝试的新技术一样。知道吗？我们的确是在尝试新技术！！！之前的问题并未奇迹般地消失，只是背景和环境发生了变化。我们希望从过去未达成的目标中吸取经验。

例如，在大型供应商展望的SOA时代（通过委员会等方式编写无穷尽的规范、重命名EAI等），我们发现三件事导致了“ESB”的承诺未能实现：

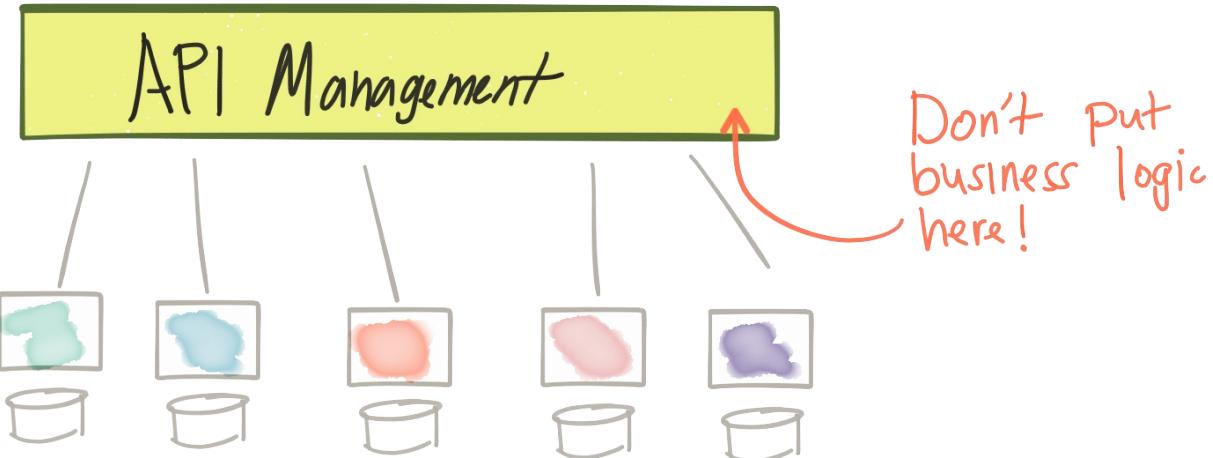
- 组织结构（让我们不停地建造一个个竖井（silo）！）
- 技术复杂（SOAP/WS-*、JBI、规范XML、专有格式等）
- 需要业务逻辑来实现路由、转换、中介、编排等

而最后一点导致了过度操作。我们希望敏捷化，但我们将重要的业务逻辑从服务中分离出来，并转移到另一个团队拥有的集成层。现在，当我们想要更改服务时（出于敏捷的需要），我们做不到；我们不得不暂停并和ESB团队产生大量同步（这会带来风险）。随着这个团队和这个架构成为应用程序的中心，我们就可以理解ESB团队为什么会被请求所淹没

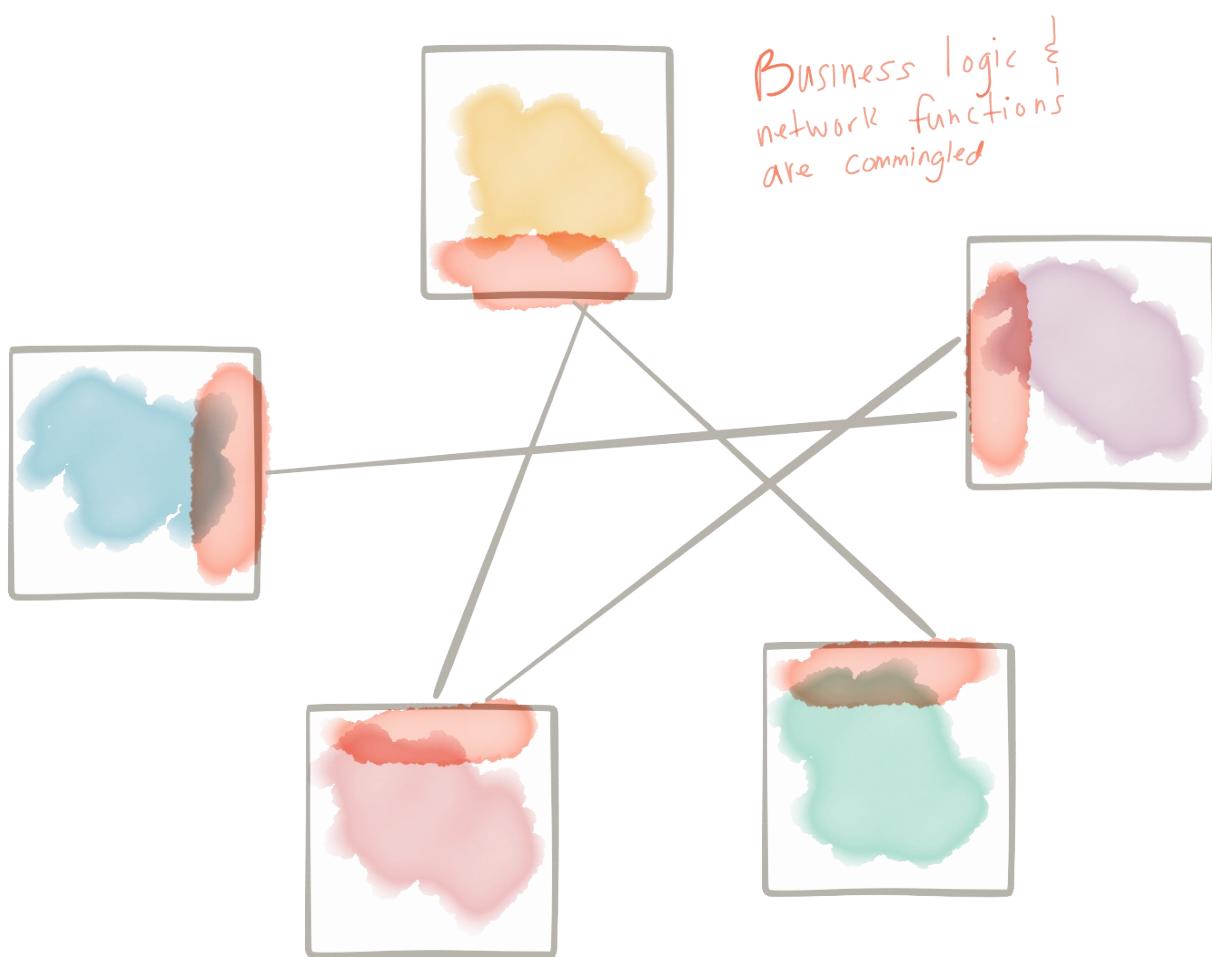
(同样是因为敏捷的需要)，无法跟上节奏了（即风险的体现）。所以尽管意图很好，但是我们发现把核心的应用程序联网功能与业务逻辑有关的功能混到一起不是个好主意。我们最终会遭遇瓶颈。



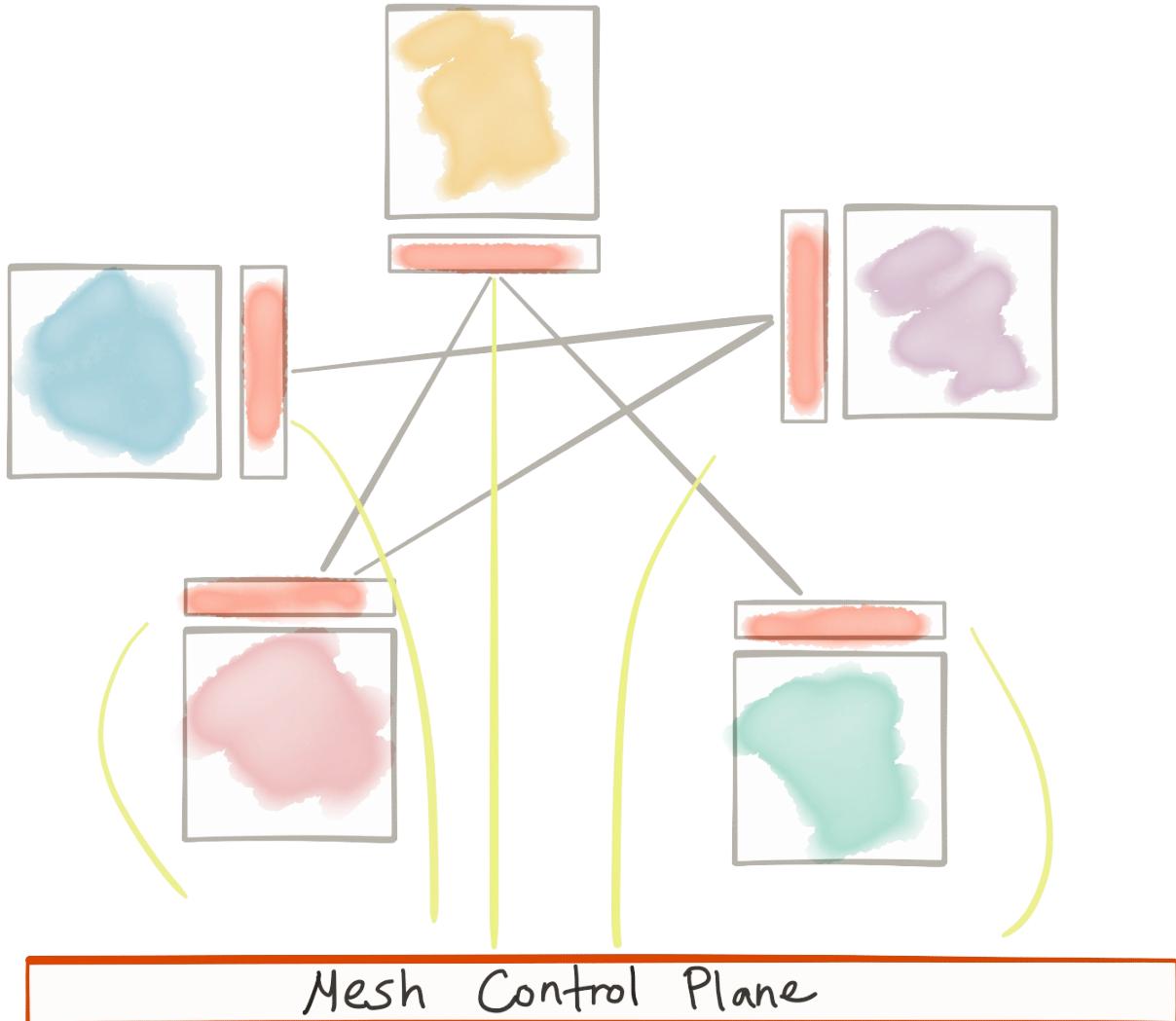
接下来出现的是REST革命和API优先的思潮。这一运动部分是出于对SOAP/ESB/SOA复杂性的抵制，部分是考虑通过API转换数据，引发新的商业模式，扩展现有模式。我们还向架构引入了新的基础设施：API管理网关。该网关让我们能集中地控制针对业务API的外部访问，它是通过安全ACL、访问配额和API使用计划、指标收集、计费、文档等实现的。然而，就像我们在前面的消息代理例子中看到的那样，当采用集中治理时，我们会有一次处理过多事情的风险。例如，我们会想，当API调用通过我们的网关时，为何不添加路由、转换和编排呢？然而，这样做的后果是我们开始妄想打造一个能够将基础设施级网络问题与业务逻辑相结合的ESB。这是一条死胡同。



但是，即使在REST/非SOAP时代，我们仍需要解决上述各项服务之间的问题，（不仅仅是所谓的“南北”流量（编者注：“North-South” traffic。网络传输的流量方向，南北一般指代纵向的不同层级间的传输流量，比如内外网，东西一般指代同一级别的传输流量，比如虚拟机之间），还包括“东西方”流量交互的问题）。更具挑战性的是，我们需要找出一种使用商用基础设施环境（又名云）的方法，而这种方法往往会加剧上述问题。传统信息代理、ESB等不适合这种模式。相反，我们最终会在业务逻辑中编写应用程序网络功能。……我们开始看到诸如[Netflix OSS堆栈](#)、[Twitter Finagle](#)这样的事物，甚至还有我们自己的[Fuse Fabric](#)，它们的确可以解决其中的一些问题。它们通常是库或框架，旨在解决上述一些问题，但它们是特定语言编写的，并且混合在业务逻辑中（或在整个基础设施中分散的业务逻辑中）。因此，这个模式也有问题。这种方法需要在每种语言/框架/运行环境上进行大量投资。我们基本上必须在跨语言/框架上加倍努力，并期望它们能够有效、正确和一致地工作。



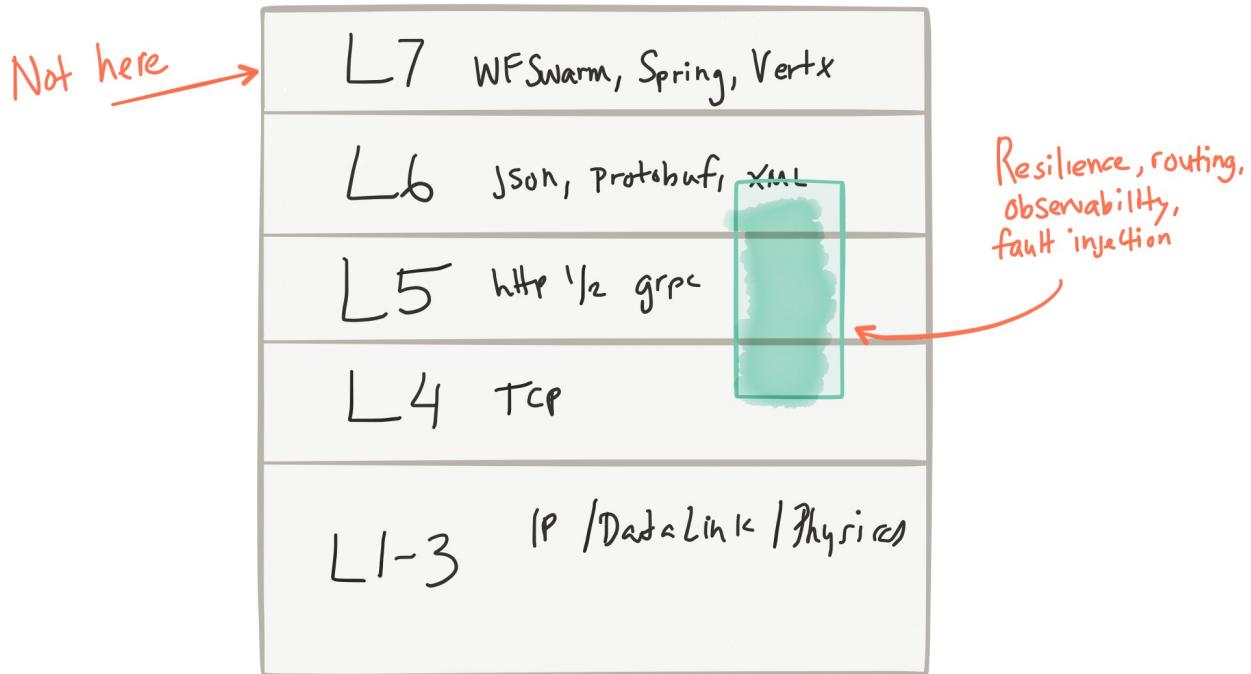
经历了上述这些困难，在有能力控制/配置/监控应用级请求的前提下，我们能用最低的成本，高度分散（high decentralization）地将应用网络功能加入到基础设施中，从而解决上述的一些问题。我们称之为“service mesh”。基于[Envoy Proxy](#)的[istio.io](#)就是个很好的例子。它使我们将应用程序网络功能的问题与业务逻辑区分的问题分离开来：



正如[Phil Calcado](#)说的那样，这与TCP/IP网络层的工作非常相似；网络功能被推入到操作系统中，但它们并不是应用程序的一部分。

与此相关的概念

通过service mesh，我们明确地将应用程序网络功能与应用程序代码、业务逻辑分离开来，我们正在将其推向另一个层次（推向基础设施，这与我们在网络堆栈、TCP等方面所做的工作类似）。



所涉及的网络功能包括：

- 简单的、基于元数据的路由
- 自适应/客户端侧负载均衡
- 服务发现
- 熔断
- 超时/重试/预算
- 速率限制
- 度量/记录/追踪
- 故障注入
- A/B 测试/流量整形/请求镜像

明确不含以下项目（这些项目可能更适合于业务层级的逻辑、应用程序和服务，而不是某些集中式基础设施。）：

- 消息转换
- 消息路由（基于内容的路由）
- 服务编排

那么 service mesh 与以下事物的不同点在于

ESB

- 在某些网络功能上有重叠
- 控制点分散
- 策略针对特定应用程序

- 不处理业务逻辑问题（如映射、转换、基于内容的路由等）

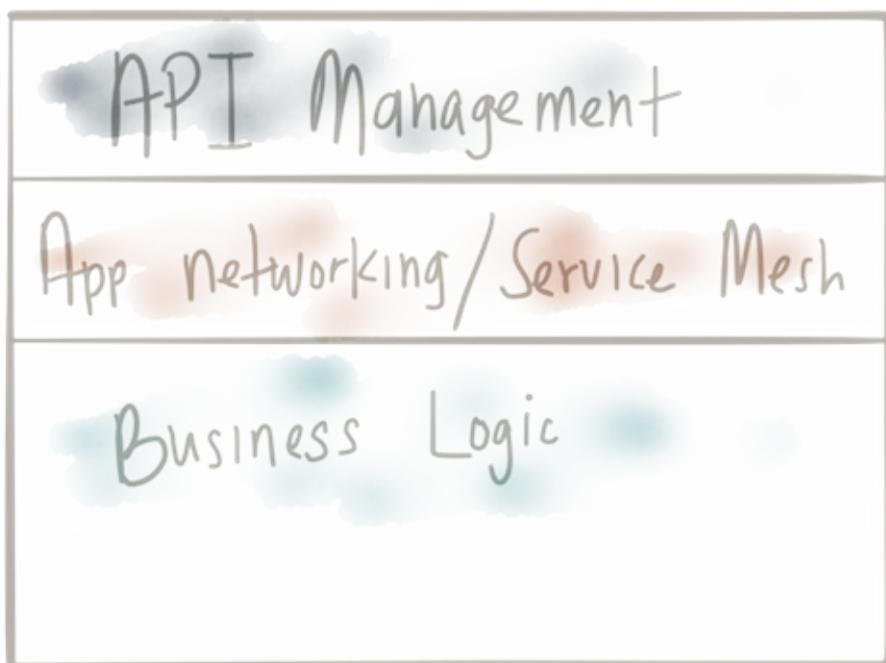
消息代理

- 在服务发现、负载均衡、重试、背压等方面有重叠（大概差着30,000英尺）
- 控制点分散
- 策略针对特定应用程序
- 不承担发消息的职责

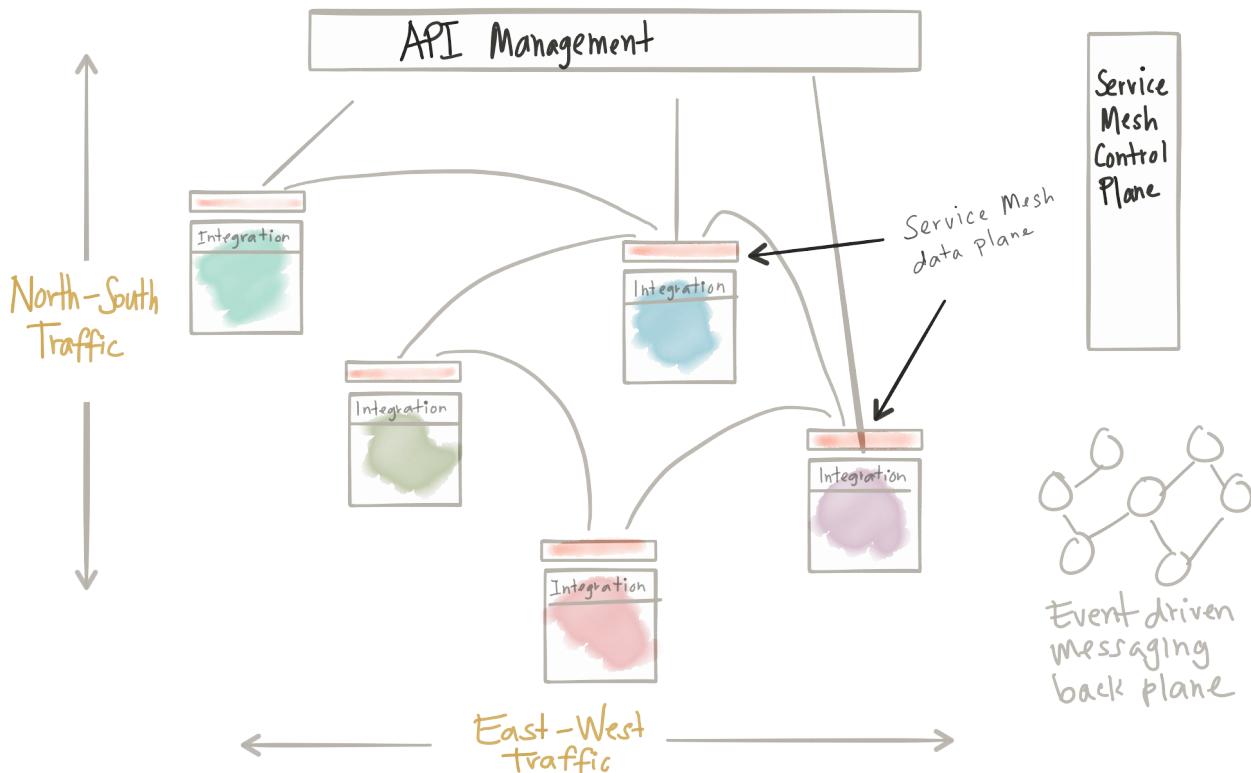
API管理

- 在策略控制、速率限制、ACL，配额安全等方面有重叠
- 不处理API的业务方面（定价、文档、用户到计划的映射等）
- 类似点在于它不实现业务逻辑

至于API管理，似乎有一些重叠，但我更倾向于把这些重叠看作是高度互补。API管理提供有关API的更高级语义（如文档、用户注册/访问、生命周期管理、开发人员API计划、计费和退款等）。调用API时，较低级别的应用程序网络，如熔断器、超时、重试等都是至关重要的，但它们很适合service mesh层。重叠点如ACL、速率限制、配额和策略执行等可以由API管理层定义，但实际上由service mesh层实施。通过这种方式，我们可以拥有完整的端到端策略和访问控制，并强化南/北流量和东/西流量的弹性。如[@ZackButcher](#)（来自Istio团队）在[twitter](#)中指出那样，“随着规模越来越大，从生产和管理服务的角度来看，东西流量开始变得更像南北流量。”



把它们都整合起来



我们需要采用API优先的方法来处理我们的系统架构。我们也要解决弹性等问题。我们也发现了在集成上的挑战。在许多方面，将基于异步事件传递和事件处理的架构作为您的API和微服务交互的底层可以帮助提高可用性、弹性和降低脆性。过去，解决这些问题是一项挑战，因为竞争产品和解决方案的关注存在重叠和混淆。随着我们转向云架构，这种情况变得越来越明显，我们需要梳理这些问题，并将它们放在我们的架构中的适当位置，否则我们就会重蹈覆辙。

从上图可以看出：

- API管理用于进入的南/北流量
- service mesh（控制+数据层）用于服务之间的应用网络功能
- service mesh执行东西流量的API管理策略
- 集成（编排、转换、反损坏（anti-corruption）层）作为应用程序的一部分
- 事件驱动的消息底层（back plane），用于真正的异步/事件驱动的交互

如果我们回顾前面提到的四个假设，那么下面是我们如何解决它们的方法：

- 第一：服务通过网络进行交互 - 我们使用service mesh数据层/服务代理
- 第二：交互并非微不足道的 - 在业务本身实现业务集成
- 第三：控制和可观察性 - 使用API管理加service mesh控制层
- 第四：您具体的业务逻辑；使用service mesh/消息传递等进行优化

业务逻辑真的可以被分离出来吗？

我想是可以的，但会存在不清晰的边界。在service mesh中，我们说应用程序应该能意识到应用程序网络功能，但是不应该在应用程序代码中被实现。如何使应用程序更清楚地意识到应用程序网络功能或service mesh层正在做什么事情，有待进一步说明。我认为在这种情况下，很多库或框架会被创建出来。例如，如果Istio service mesh触发熔断，重试一些请求，或者由于特定原因而失败，那么应用程序需要对这些场景有更多的上下文信息以供理解。我们需要一种方法来捕获这些情况或背景并将其反馈给服务。另一个例子是在服务之间传播跟踪背景（即分布式跟踪，如OpenTracing），并且透明地完成传播。我们或许会看到，这些轻型的应用程序/特定语言的库可以使应用程序/服务更智能，并允许它们追溯特定的错误。

我们该何去何从

今天这一架构的所有部分具有不同的成熟度。即使如此，对我们的服务架构采取原则化的方法是关键。业务逻辑与应用程序网络应该分开。使用service mesh实现应用程序网络，使用API管理层来处理高级别的以API为中心的问题，将让特定业务的集成放在服务层中。这样一来，我们就可以通过事件驱动的底层（backplane）构建数据密集型或数据可用系统。我认为当我们前进时，我们将不断地看到这些原则在具体的技术实现中被采用。在Red Hat（我工作的地方），我们看到诸如[3Scale](#)、[Istio.io on Kubernetes](#)、[Apache Camel](#)和诸如[ActiveMQ Artemis](#)/[Apache Qpid Dispatch Router](#)（包括非Red Hat技术，如[Apache Kafka](#)[IMHO](#)）的讯息技术正被作为强大的构建块来构建遵循以上原则的服务架构。

模式之服务网格

tags: Phil Calçado

原文作者 : Phil Calçado

原文链接 : [Pattern: Service Mesh](#)

转载地址 : [模式之服务网格](#)

译者 : 薛命灯

声明 : 转载自 InfoQ

分布式系统为我们带来了各种可能性，同时也引入了各种问题。如果系统不是很复杂，工程师们一般会尽量避免进行远程交互，降低分布式系统与生俱来的复杂性。使用分布式系统最安全的方式就是尽可能避免分布式，尽管这样会让散布的系统出现重复的逻辑和数据。

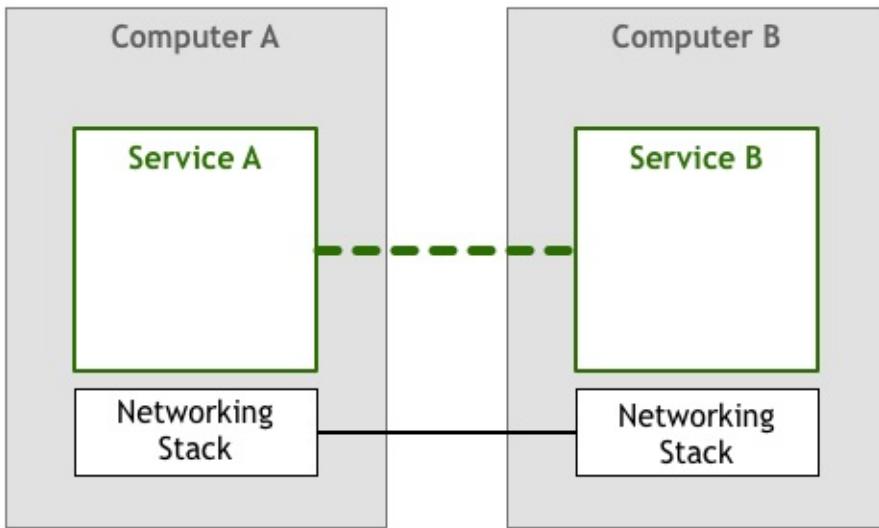
但行业的发展促使我们迈出更大的步伐，大型的集中式服务变成了数百个甚至数千个小型服务。我们不得不去解决新的挑战，先是一些个别案例，后来问题变得越来越复杂。在不断挖掘出新问题并设计出更好的解决方案之后，我们开始学会总结，将一些常见的需求总结成模式、库，甚至是平台。

最初的网络计算机交互

最开始人们想象的计算机之间的交互方式是这样的：



一个服务通过与另一个服务发生对话来完成用户的请求。我们看到的是最最简单的视图，因为底层很多信息都被屏蔽掉了，比如那些负责传输字节码的层和传输电子信号的层。下面我们将给出稍微详细一点的组件图：

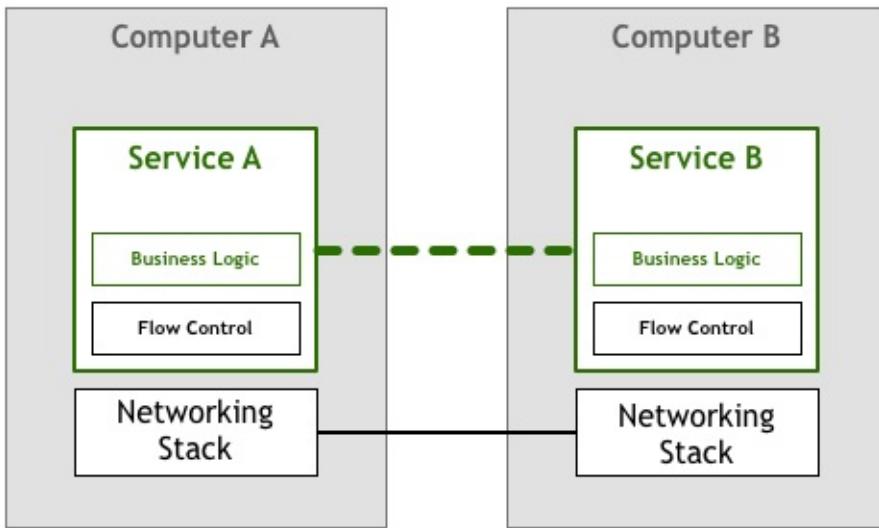


上面这个模型的变种从50年代开始就得到广泛应用。起初，计算机很稀有，也很昂贵，人们手动管理计算机之间的连接。随着计算机变得越来越普及，价格也没那么贵了，计算机之间的连接数量和数据量出现了疯狂式的增长。人们越来越依赖网络系统，工程师们必须确保他们开发的服务能够满足用户的要求。

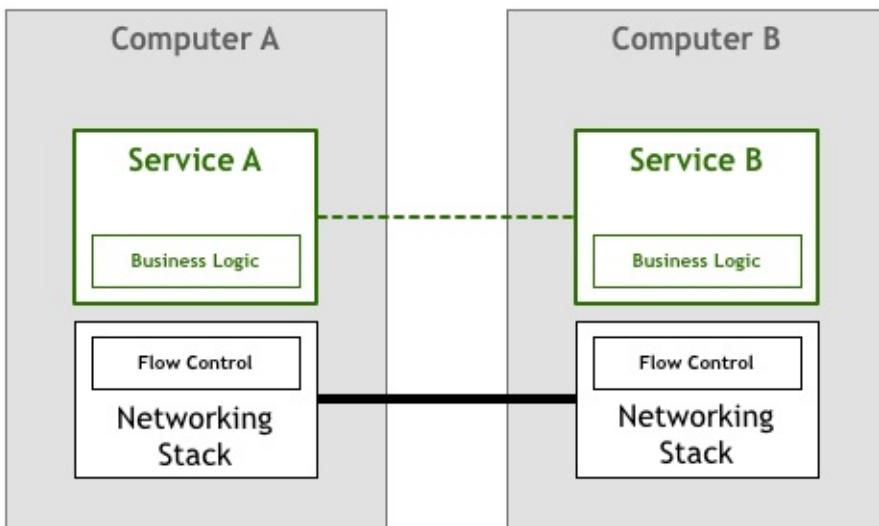
于是，如何提升系统质量成为人们关注的焦点。机器需要知道如何找到其他节点，处理同一个通道上的并发连接，与非直接连接的机器发生通信，通过网络路由数据包，加密流量……

除此之外，还需要流量控制机制。流量控制可以防止上游服务器给下游服务器发送过多的数据包。在网络系统中，至少会存在两台对彼此毫无所知的机器。比如，服务器A按照一定速率向服务器B发送数据包，但服务器B并不能保证及时处理这些数据包。服务器B有可能同时在处理其他任务，又或者数据包到达的顺序发生颠倒，服务器B一直在等待本应先到达的数据包。在这种情况下，不仅服务器A的请求无法得到响应，而且还可能让服务器B过载，因为服务器B需要把数据包放进未处理队列。

在一段时期内，开发人员需要在自己的代码里处理上述问题。在下图的示例中，为了确保不给其他服务造成过载，应用程序需要处理网络逻辑，于是网络逻辑和业务逻辑就混杂在一起。



所幸的是，技术发展得很快，TCP/IP这类标准的出现解决了流量控制等问题。尽管网络逻辑代码依然存在，但已经从应用程序里抽离出来，成为操作系统网络层的一部分。



微服务的出现

时过境迁，计算机越来越普及，也越来越便宜，上述的网络技术栈已经被证实是构建可靠连接系统行之有效的方案。相互连接的节点越来越多，业界出现了各种网络系统，如分布式代理和面向服务架构。

分布式为我们带来了更高层次的能力和好处，但挑战依然存在。有些挑战是全新的，但也有一些只是旧有问题的升级，比如我们之前讨论的网络问题。

在90年代，来自Sun Microsystems的Peter Deutsch等人发表了“[The 8 Fallacies of Distributed Computing](#)”。Peter列出了人们在使用分布式系统时容易做出的几项假设。Peter指出，如果只是针对原始网络架构或理论模型，那么这些假设或许是对的，但在现实世界中绝对不是。

1. 网络是可靠的
2. 没有延迟
3. 带宽是无限的
4. 网络是安全的
5. 拓扑结构不会发生变化
6. 存在网络管理员
7. 零传输成本
8. 网络是同质的

工程师不能忽略这些“谬论”，他们必须在开发当中处理这些问题。

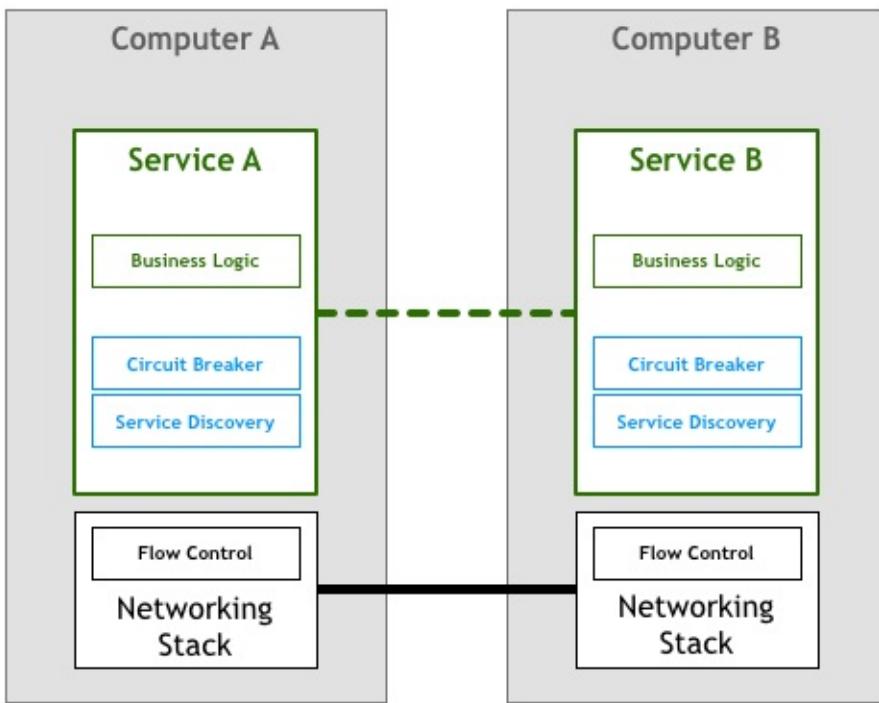
微服务架构是更为复杂的分布式系统，它给运维带来了更多挑战。我们之前已经做过一些详细的讨论，下面是之前讨论内容的要点。

1. 快速分配计算资源
2. 基本的监控
3. 快速部署
4. 易于分配的存储
5. 易于访问的边界
6. 认证和授权
7. 标准的RPC

尽管数十年前出现的TCP/IP协议栈和通用网络模型仍然是计算机通信最为有力的工具，但一些更为复杂的架构也催生了一些新的需求，工程师们需要在系统中加入一个新的层。

我们以服务发现和回路断路器为例，这两种技术被用于解决弹性和分布式问题。

历史总是惊人的相似，第一批采用微服务架构的企业遵循的是与第一代网络计算机系统类似的策略。也就是说，解决网络通信问题的任务又落在了工程师的肩上。



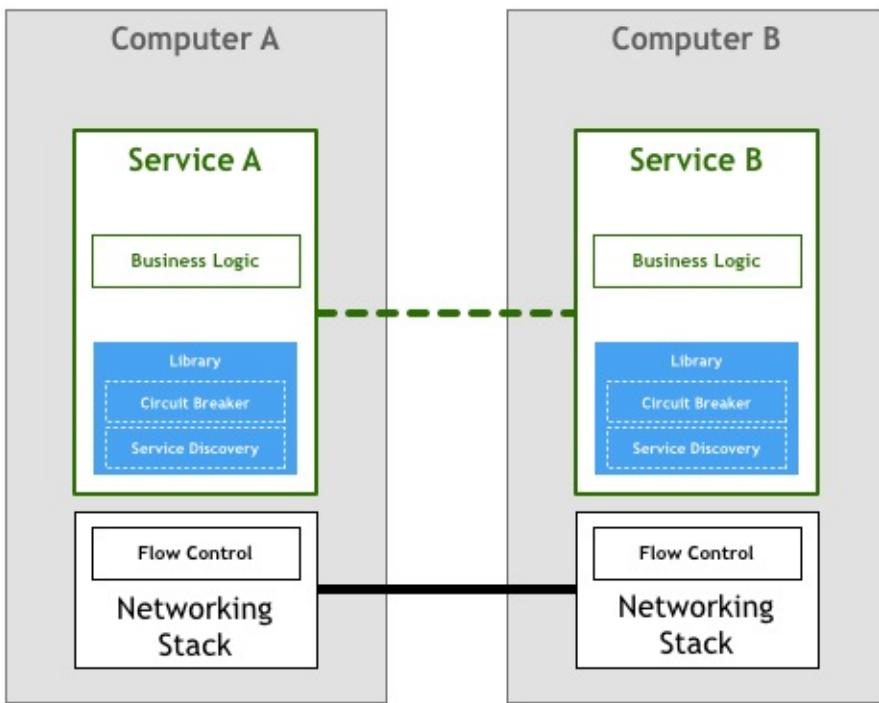
服务发现的目的是找到有能力处理请求的服务实例。比如，有个叫作Teams的服务需要找到一个叫作Players的服务实例。通过调用服务发现，可以获得一个满足条件的服务器清单。对于单体系统来说，这个可以通过DNS、负载均衡器和端口机制（比如通过HTTP服务器的8080端口来绑定服务）来实现。而在分布式系统里，事情就复杂了，服务发现需要处理更多的任务，比如客户端负载均衡、多环境（如staging环境和生产环境）、分布式服务器的物理位置等。之前可能只需要一行代码来处理服务器名问题，而现在需要更多的代码来处理更多的边界问题。

Michael Nygard在他的“[Release It](#)”一书中提到了回路断路器模式。Martin Fowler也对该模式做了总结：

回路断路器背后的原理其实很简单，将函数封装在回路断路器对象里，当故障率达到某个阈值，回路断路器开始发挥作用，所有发给回路断路器的请求都将返回错误，这些请求无法触碰到被保护的函数。通常情况下，还需要在回路断路器打开时发出告警。

这是一种非常简单但能保证服务间交互可靠性的解决方案。不过，随着分布式规模的增长，任何事情都会趋于复杂。在分布式系统里，发生问题的概率呈指数级增长，即使是最简单的事情也不能等闲视之，比如像“在回路断路器打开时发出告警”这样的小事。一个组件发生故障将会级联地影响到多个客户端，一传十，十传百，最后可能有数千个回路同时被打开。先前的问题只需要几行代码就可以解决，而现在需要更多的代码才能解决这些新问题。

要实现好上述的两种模式是很困难的，不过像Twitter的[Finagle](#)和Facebook的[Proxygen](#)这样的大型框架为我们提供了很多便利。



很多采用了微服务架构的公司遵循了上图中的模型，如Netflix、Twitter和SoundCloud。不过随着服务数量的增长，这个模型也时常出现一些问题。

即使一个公司使用了Finagle，仍然需要投入时间和精力去打通各个系统。从我在SoundCloud和DigitalOcean的经验来看，一个拥有100到250个工程师的企业，至少需要投入十分之一的人力在构建工具上。这种成本有时候是显而易见的，因为一部分工程师被安排专职负责构建工具，但有时候它也会存在于无形，并“吞噬”掉本该用于构建产品的时间。

第二个问题是，上述的方案会限制可用的工具、运行时和编程语言。微服务软件库一般专注于某个平台，使用了某种编程语言或使用了某种运行时（如JVM）。如果一个公司选择了另一个不支持原先软件库的平台，那么就要将代码重新移植到新平台上。移植过程需要耗费大量的工程时间，工程师们忙于构建工具和基础设施，无暇顾及业务和产品。这也就是为什么一些中型公司决定只支持单一的平台，比如SoundCloud仅支持Scala，DigitalOcean仅支持Go语言。

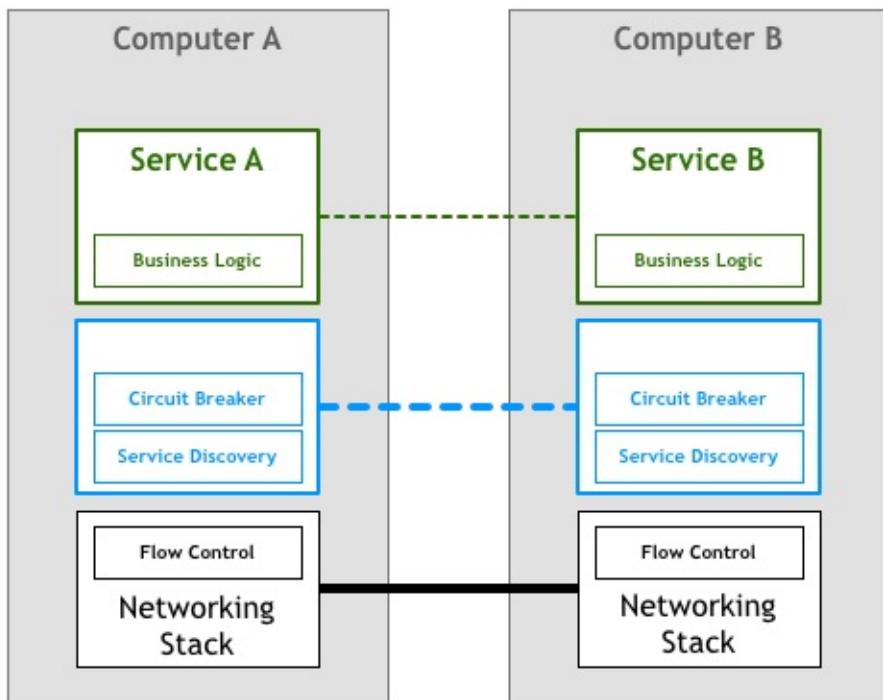
最后一个问题是监管问题。尽管软件库封装了功能，但组件本身仍然需要维护。要确保数千个服务使用的是同一个版本（或至少是兼容的版本）的软件库并不是件容易的事。每做出一次变更都需要进行集成、测试，还要重新部署所有的服务——尽管服务本身并没有发生变化。

下一步

与网络协议栈一样，我们急切地希望能够将分布式服务所需要的一些特性放到底层的平台中。

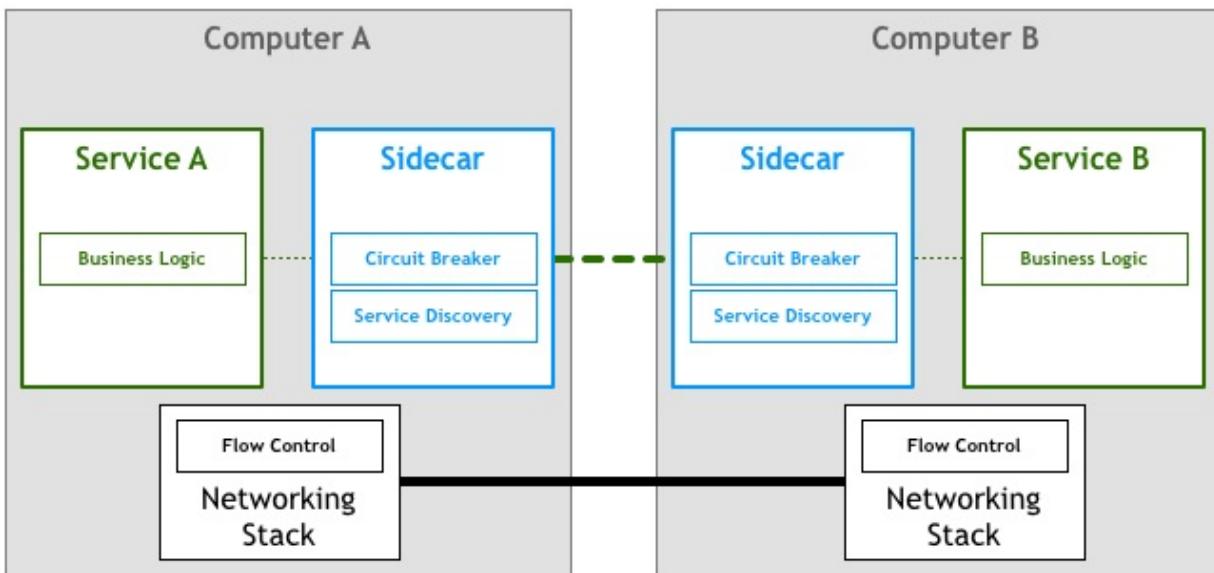
人们基于HTTP协议开发非常复杂的应用，无需关心底层TCP如何控制数据包。在开发微服务时也是类似的，工程师们聚焦在业务逻辑上，不需要浪费时间去编写服务基础设施代码或管理系统用到的软件库和框架。

把这种想法囊括进我们的架构中，就是下图所示的样子。



不过，在网络协议栈中加入这样的一个层是不实际的。很多先驱者们使用一系列代理，也就是说，一个服务不会直接与它的依赖项发生连接，所有的流量都会流经代理，代理会实现所有必需的特性。

在首批有文档记录的开发案例中，出现了边车（[sidecar](#)）这个概念。边车就是与应用程序一起运行的独立进程，为应用程序提供额外的功能。2013年，Airbnb开发了[Synapse](#)和[Nerve](#)，也就是边车的一种开源实现。一年之后，Netflix发布了[Prana](#)，它也是一个边车，可以让非JVM应用接入他们的NetflixOSS生态系统。在SoundCloud，我们也开发了一些边车，让遗留的Ruby应用可以使用我们为JVM微服务而构建的基础设施。

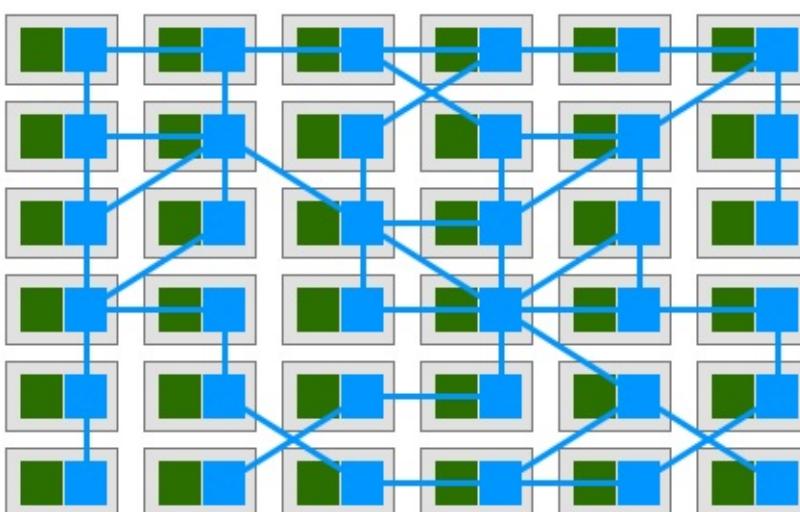


大多数开源的代理是为特定的基础设施组件而设计的。例如，Airbnb的Nerve和Synapse假设服务一定是注册到ZooKeeper上的，而Prana要求一定要使用Netflix自己的[Eureka注册服务](#)。

随着微服务架构日渐流行，我们也看到了新一波的代理可以用在不同基础设施组件上。[Linkerd](#)就是其中较为出名的一个，它是Buoyant公司基于Twitter微服务平台而开发的。而不久之前，Lyft宣布[Envoy](#)成为CNCF的官方项目。

Service Mesh（服务网格）

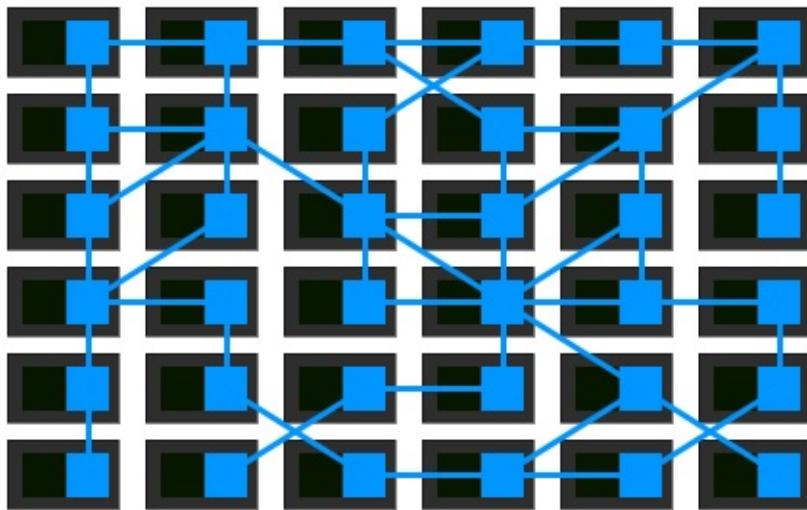
在这样的模型里，每个服务都会有一个边车代理与之配对。服务间通信都是通过边车代理进行的，于是我们就会得到如下的部署图。



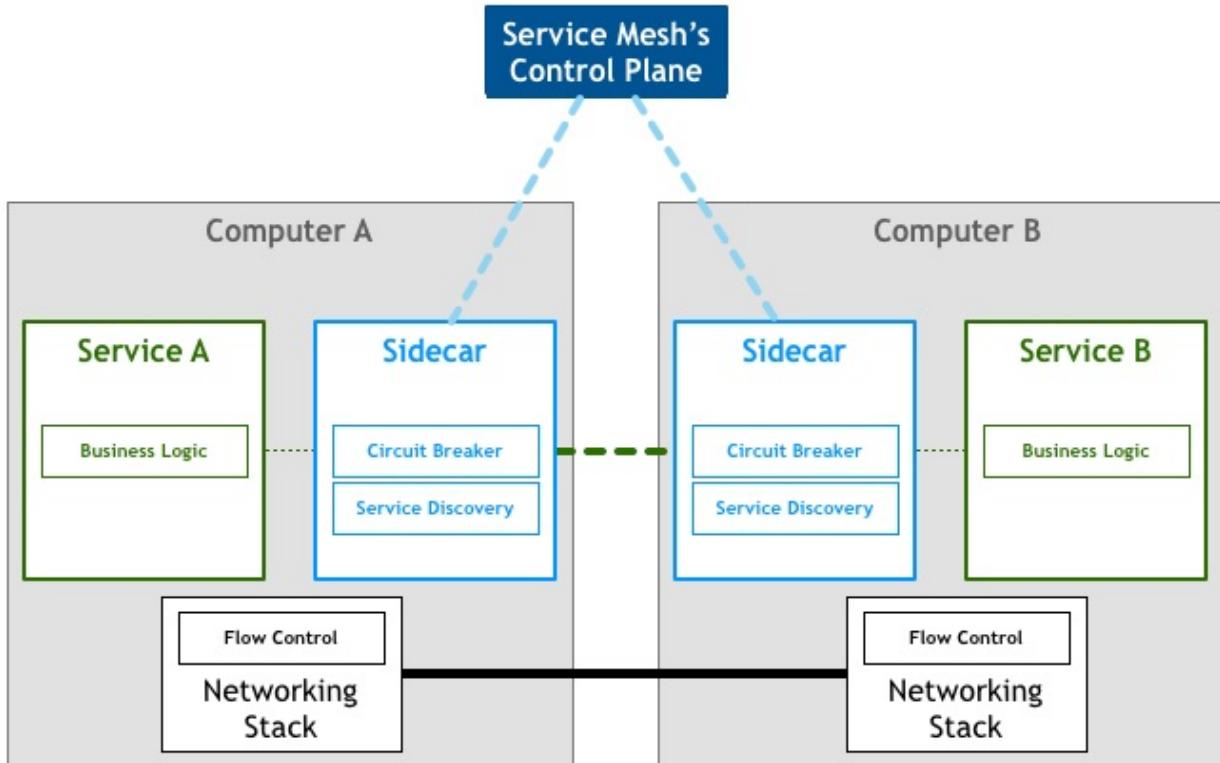
Buoyant的CEO [William Morgan](#)发现，代理之间的连接形成了一种网格网络。2017年初，William对这样的平台做出了定义，并称之为Service Mesh：

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，服务网格保证请求可以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，但应用程序不需要知道它们的存在。

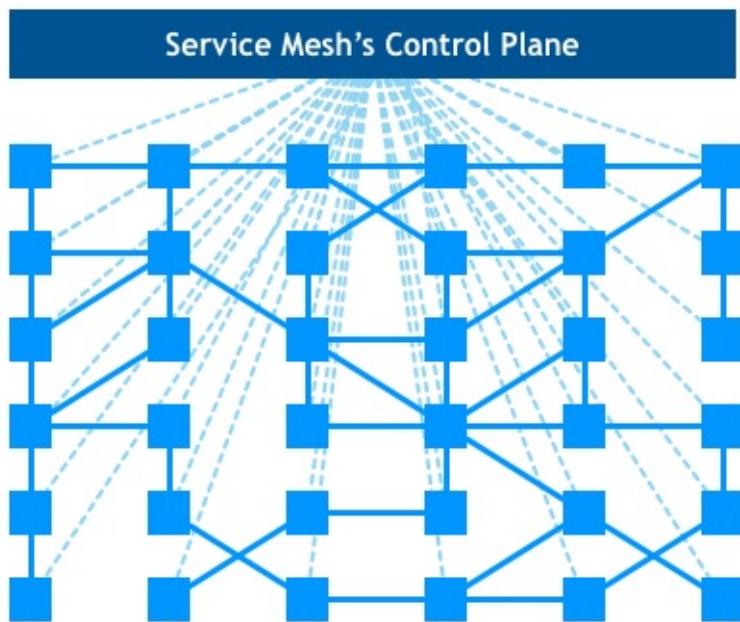
这个定义最强有力的部分在于，它不再把代理看成单独的组件，并强调了这些代理所形成的网络的重要性。



组织开始将他们的微服务部署到更为复杂的运行时（如Kubernetes和Mesos）上，并开始使用这些平台提供的网格网络工具。他们正从使用一系列独立运行的代理转向使用集中式的控制面板。



从鸟瞰图中可以看到，服务的实际流量仍然在代理间流转，不过控制面板对每一个代理实例了如指掌，通过控制面板可以实现代理的访问控制和度量指标收集。



最近发布的[Istio](#)就是这类系统最为突出的代表。

我们还无法对Service Mesh将给大规模系统带来怎样的影响做出全面的定论，不过我们至少可以看到两个方面的优势。首先，微服务架构的一些公共组件已经是现成的，很多小公司可以享受到之前只有大公司才能享受的一些特性。其次，我们或许能够因此使用最好的工具和编程语言，而无需担心不同平台对软件库和模式的支持存在差异。

什么是服务网格以及为什么我们需要服务网格？

tags: William Morgan

原文作者 : William Morgan

原文地址 : [WHAT'S A SERVICE MESH? AND WHY DO I NEED ONE?](#)

转载地址 : 什么是服务网格以及为什么我们需要服务网格？

译者 : 薛命灯

声明 : 转载自InfoQ

Service Mesh（服务网格）是一个基础设施层，让服务之间的通信更安全、快速和可靠。如果你在构建云原生应用，那么就需要Service Mesh。

在过去的一年中，Service Mesh已经成为云原生技术栈里的一个关键组件。很多拥有高负载业务流量的公司都在他们的生产应用里加入了Service Mesh，如PayPal、Lyft、Ticketmaster和Credit Karma等。今年一月份，Service Mesh组件Linkerd成为CNCF（Cloud Native Computing Foundation）的官方项目。不过话说回来，Service Mesh到底是什么？为什么它突然间变得如此重要？

在这篇文章里，我将给出Service Mesh的定义，并追溯过去十年间Service Mesh在应用架构中的演变过程。我会解释Service Mesh与API网关、边缘代理（Edge Proxy）和企业服务总线之间的区别。最后，我会描述Service Mesh将何去何从以及我们可以作何期待。

什么是Service Mesh？

Service Mesh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，Service Mesh保证请求可以在这些拓扑中可靠地穿梭。在实际应用当中，Service Mesh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，但应用程序不需要知道它们的存在。

随着云原生应用的崛起，Service Mesh逐渐成为一个独立的基础设施层。在云原生模型里，一个应用可以由数百个服务组成，每个服务可能有数千个实例，而每个实例可能会持续地发生变化。服务间通信不仅异常复杂，而且也是运行时行为的基础。管理好服务间通信对于保证端到端的性能和可靠性来说是非常重要的。

Service Mesh是一种网络模型吗？

Service Mesh实际上就是处于TCP/IP之上的一层抽象层，它假设底层的L3/L4网络能够点对点地传输字节（当然，它也假设网络环境是不可靠的，所以Service Mesh必须具备处理网络故障的能力）。

从某种程度上说，Service Mesh有点类似TCP/IP。TCP对网络端点间传输字节的机制进行了抽象，而Service Mesh则是对服务节点间请求的路由机制进行了抽象。Service Mesh不关心消息体是什么，也不关心它们是如何编码的。应用程序的目标是“将某些东西从A传送到B”，而Service Mesh所要做的就是实现这个目标，并处理传送过程中可能出现的任何故障。

与TCP不同的是，Service Mesh有着更高的目标：为应用运行时提供统一的、应用层面的可见性和可控性。Service Mesh将服务间通信从底层的基础设施中分离出来，让它成为整个生态系统的一等公民——它因此可以被监控、托管和控制。

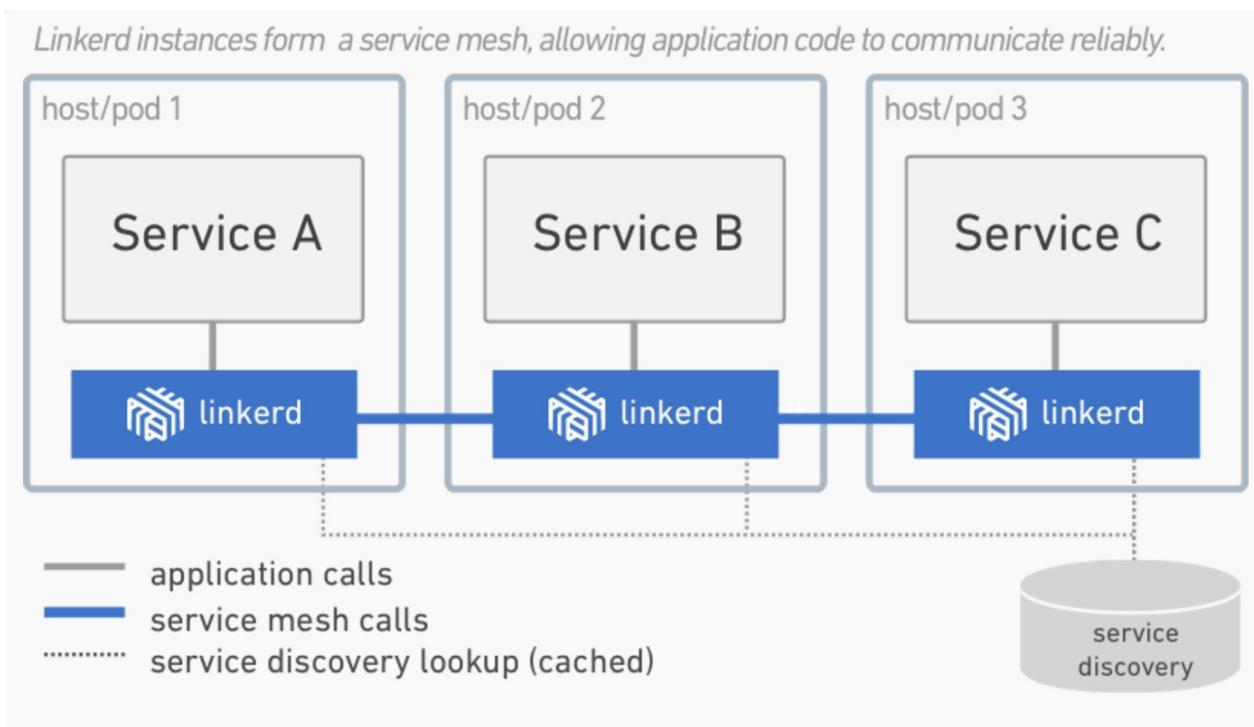
Service Mesh可以做什么？

在云原生应用中传输服务请求是一项非常复杂的任务。以Linkerd为例，它使用了一系列强大的技术来管理这种复杂性：回路断路器、负载均衡、延迟感知、最终一致性服务发现、重试和超时。这些技术需要组合在一起，并互相协调，它们与环境之间的交互也非常微妙。

举个例子，当一个请求流经Linkerd时，会发生如下的一系列事件。

1. Linkerd根据动态路由规则确定请求是发给哪个服务的，比如是发给生产环境里的服务还是发给staging环境里的服务？是发给本地数据中心的服务还是发给云端的服务？是发给最新版本的服务还是发给旧版本的服务？这些路由规则可以动态配置，可以应用在全局的流量上，也可以应用在部分流量上。
2. 在确定了请求的目标服务后，Linkerd从服务发现端点获取相应的服务实例。如果服务实例的信息出现了偏差，Linkerd需要决定哪个信息来源更值得信任。
3. Linkerd基于某些因素（比如最近处理请求的延迟情况）选择更有可能快速返回响应的实例。
4. Linkerd向选中的实例发送请求，并把延迟情况和响应类型记录下来。
5. 如果选中的实例发生宕机、没有响应或无法处理请求，Linkerd就把请求发给另一个实例（前提是请求必须是幂等的）。
6. 如果一个实例持续返回错误，Linkerd就会将其从负载均衡池中移除，并在稍后定时重试（这个实例有可能只是临时发生故障）。
7. 如果请求超时，Linkerd会主动放弃请求，不会进行额外的重试。
8. Linkerd以度量指标和分布式日志的方式记录上述各种行为，然后将度量指标发送给中心度量指标系统。

除此之外，Linkerd还能发起和终止TLS、执行协议升级、动态调整流量、在数据中心之间进行失效备援。



Linkerd的这些特性可以保证局部的弹性和应用层面的弹性。大规模分布式系统有一个共性：局部故障累积到一定程度就会造成系统层面的灾难。Service Mesh的作用就是在底层系统的负载达到上限之前通过分散流量和快速失效来防止这些故障破坏到整个系统。

为什么我们需要Service Mesh？

Service Mesh并非新出现的功能。一直以来，Web应用程序需要自己管理复杂的服务间通信，从过去十多年间应用程序的演化就可以看到Service Mesh的影子。

2000年左右的中型Web应用一般使用了三层模型：应用逻辑层、Web服务逻辑层和存储逻辑层。层与层之间的交互虽然也不算简单，但复杂性是很有限的，毕竟一个请求最多只需要两个跳转。虽然这里不存在“网格”，但仍然存在跳转通信逻辑。

随着规模的增长，这种架构就显得力不从心了。像Google、Netflix、Twitter这样的公司面临着大规模流量的挑战，他们实现了一种高效的解决方案，也就是云原生应用的前身：应用层被拆分为多个服务（也叫作微服务），这个时候层就变成了一种拓扑结构。这样的系统需要一个通用的通信层，以一个“富客户端”包的形式存在，如Twitter的Finagle、Netflix的Hystrix和Google的Stubby。

一般来说，像Finagle、Stubby和Hystrix这样的包就是最初的Service Mesh。云原生模型在原先的微服务模型中加入了两个额外的元素：容器（比如Docker）和编排层（如Kubernetes）。容器提供了资源隔离和依赖管理，编排层对底层的硬件进行抽象池化。

这三个组件让应用程序在云环境中具备了伸缩能力和处理局部故障的能力。但随着服务和实例的数量增长，编排层需要无时不刻地调度实例，请求在服务拓扑间穿梭的路线也变得异常复杂，再加上可以使用任意语言来开发不同的服务，所以之前那种“富客户端”包的方式就行不通了。

这种复杂性和迫切性催生了服务间通信层的出现，这个层既不会与应用程序的代码耦合，又能捕捉到底层环境高度动态的特点，它就是Service Mesh。

Service Mesh的未来

尽管Service Mesh在云原生系统方面的应用已经有了快速的增长，但仍然存在巨大的提升空间。无服务器(Serverless)计算（如Amazon的Lambda）正好需要Service Mesh的命名和链接模型，这让Service Mesh在云原生生态系统中的角色得到了彰显。服务识别和访问策略在云原生环境中仍显初级，而Service Mesh毫无疑问将成为这方面不可或缺的基础。就像TCP/IP一样，Service Mesh将在底层基础设施这条道上更进一步。

结论

Service Mesh是云原生技术栈中一个非常关键的组件。Linkerd项目在启动一年多之后正式成为CNCF的官方项目，并拥有了众多的贡献者和用户。Linkerd的用户横跨初创公司（如Monzo）到大规模的互联网公司（如PayPal、Ticketmaster、Credit Karma），再到拥有数百年历史的老牌公司（如Houghton Mifflin Harcourt）。

Linkerd

Istio:服务网格新生力量

tags: 敖小剑

背景：这是2017年9月的一次线上分享的内容整理。这是Istio技术在国内第一次详细介绍。

作者：敖小剑

内容简介: Service Mesh新秀，初出茅庐便声势浩荡，前有Google，IBM和lyft倾情奉献，后有业界大佬俯首膜拜，这就是今天将要介绍的主角，扛起Service Mesh大旗，掀起新一轮微服务开发浪潮的Istio！

大家晚上好，欢迎参与直播，我是今天的讲师，来自数人云的资深架构师，敖小剑。

相信大家进来前都有看到这次分享的介绍，今天的主角名叫 istio，我估计很多同学在此之前可能完全没有听过这个名字。请不必介意，没听过很正常，因为istio的确是一个非常新的东西，出世也才四个月而已。

今天的内容将会分成三个部分：

1. 介绍：让大家了解istio是什么，以及有什么好处，以及istio背后的开发团队
 2. 架构：介绍istio的整体架构和四个主要功能模块的具体功能，这块内容会比较偏技术
 3. 展望：介绍istio的后续开发计划，探讨未来的发展预期
-

介绍

istio是什么

istio是Google/IBM/Lyft联合开发的开源项目,2017年5月发布第一个release 0.1.0, 官方定义为:

Istio：一个连接，管理和保护微服务的开放平台。

按照isito文档中给出的定义：

Istio提供一种简单的方式来建立已部署的服务的网络，具备负载均衡，服务到服务认证，监控等等功能，而不需要改动任何服务代码。

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

名字和图标

Istio来自希腊语，英文意思是"sail"，翻译为中文是“启航”。它的图标如下：



可以类比google的另外一个相关产品,Kubernetes,名字也是同样起源于古希腊，是船长或者驾驶员的意思。下图是Kubernetes的图标：



后面我们会看到，istio和kubernetes的关系，就像他们的名字和图标一样，可谓"一脉相传"。

主要特性

Istio的关键功能：

- HTTP/1.1，HTTP/2，gRPC和TCP流量的自动区域感知负载平衡和故障切换。

- 通过丰富的路由规则，容错和故障注入，对流行为的细粒度控制。
- 支持访问控制，速率限制和配额的可插拔策略层和配置API。
- 集群内所有流量的自动量度，日志和跟踪，包括集群入口和出口。
- 安全的服务到服务身份验证，在集群中的服务之间具有强大的身份标识。

这些特性我们在稍后的架构章节时会有介绍。

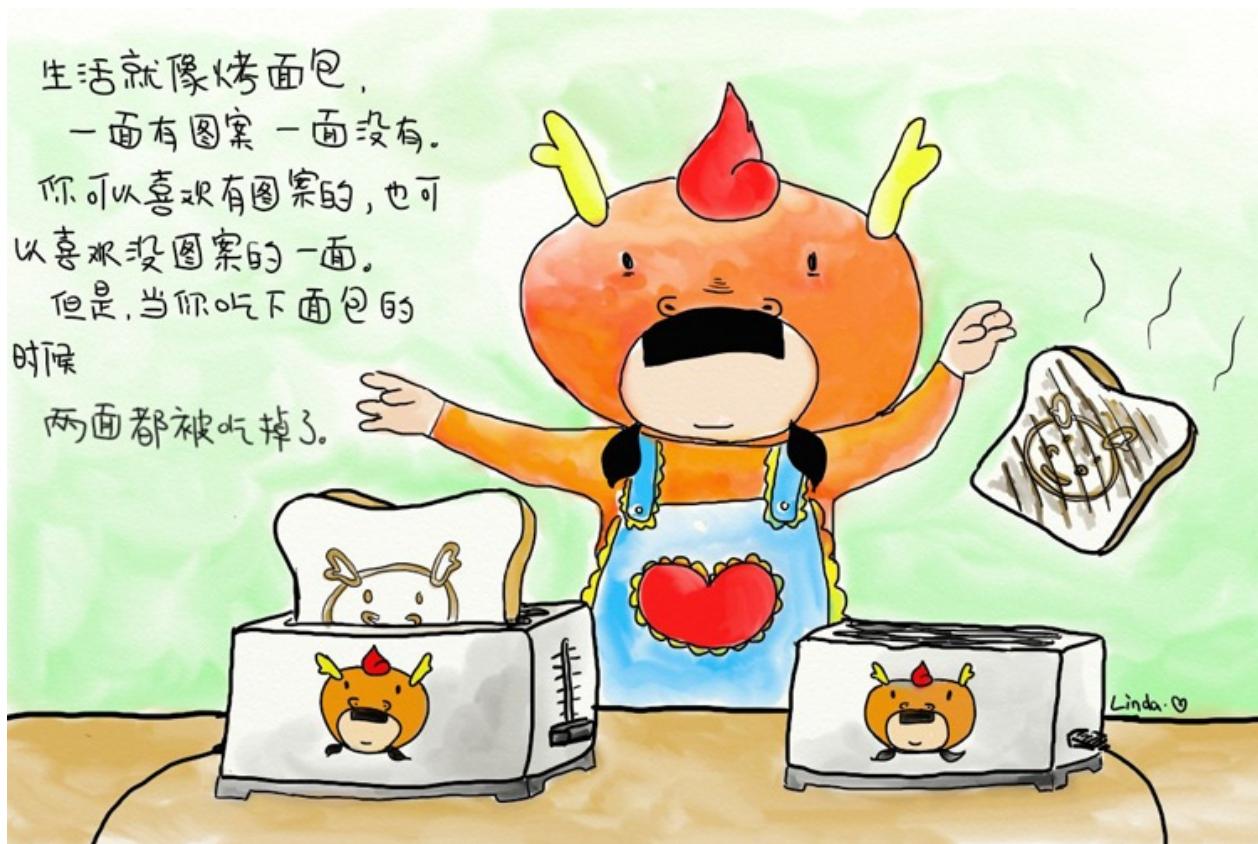
为什么要使用Istio？

在深入istio细节之前,我们先来看看,为什么要使用Istio？它可以帮我们解决什么问题？

微服务的两面性

最近两三年来微服务方兴未艾, 我们可以看到越来越多的公司和开发人员陆陆续续投身到微服务架构, 我们也看到一个一个的微服务项目落地.

但是, 在这一片叫好的喧闹中, 我们还是发觉一些问题, 普遍存在的问题: 虽然微服务对开发进行了简化, 通过将复杂系统切分为若干个微服务来分解和降低复杂度, 使得这些微服务易于被小型的开发团队所理解和维护。但是, 复杂度并非从此消失. 微服务拆分之后, 单个微服务的复杂度大幅降低, 但是由于系统被从一个单体拆分为几十甚至更多的微服务, 就带来了另外一个复杂度: 微服务的连接、管理和监控。



试想，对于一个大型系统，需要对多达上百个甚至上千个微服务的管理、部署、版本控制、安全、故障转移、策略执行、遥测和监控等，谈何容易。更不要说更复杂的运维需求，例如A/B测试，金丝雀发布，限流，访问控制和端到端认证。

开发人员和运维人员在单体应用程序向分布式微服务架构的转型中，不得不面临上述挑战。

服务网格

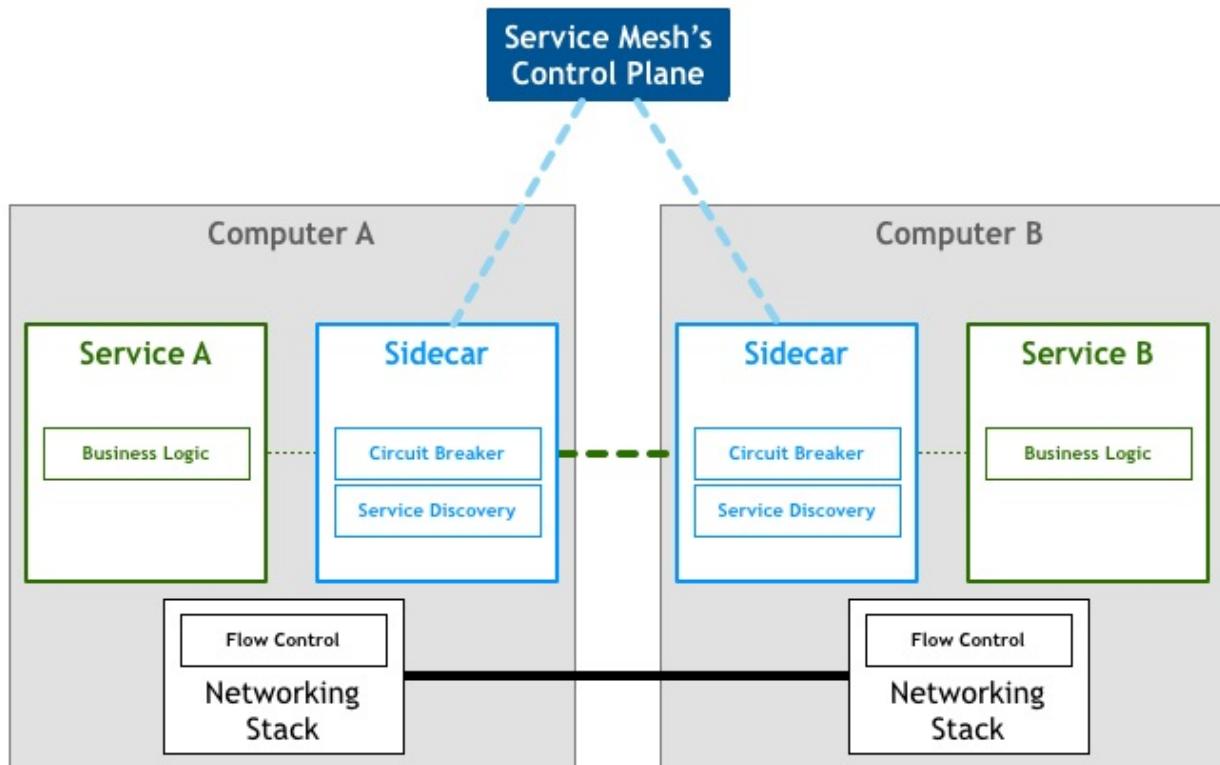
Service Mesh，服务网格，也有人翻译为"服务啮合层"。

貌似是今年才出来的新名词？反正2017年之前我是没有听过，虽然类似的产品已经存在挺长时间。

什么是Service Mesh？

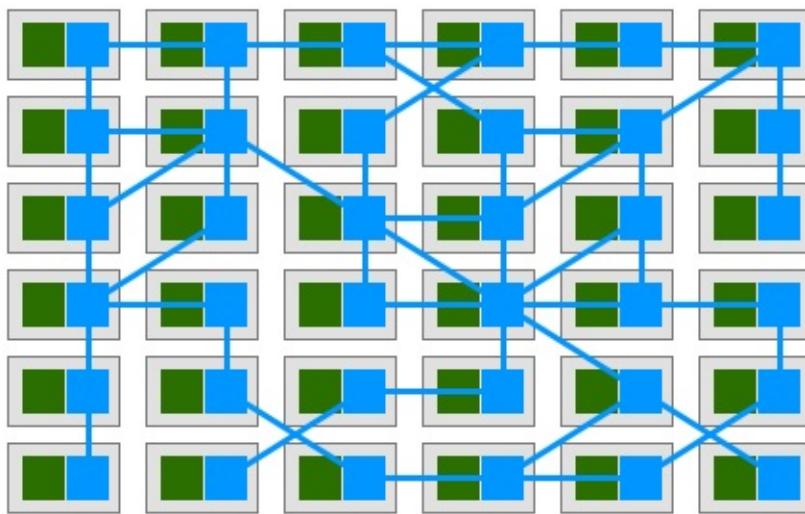
- Service Mesh是专用的基础设施层。
- 轻量级高性能网络代理。
- 提供安全的、快速的、可靠地服务间通讯。
- 与实际应用部署一起，但对应用透明。

为了帮助理解，下图展示了服务网格的典型边车部署方式：

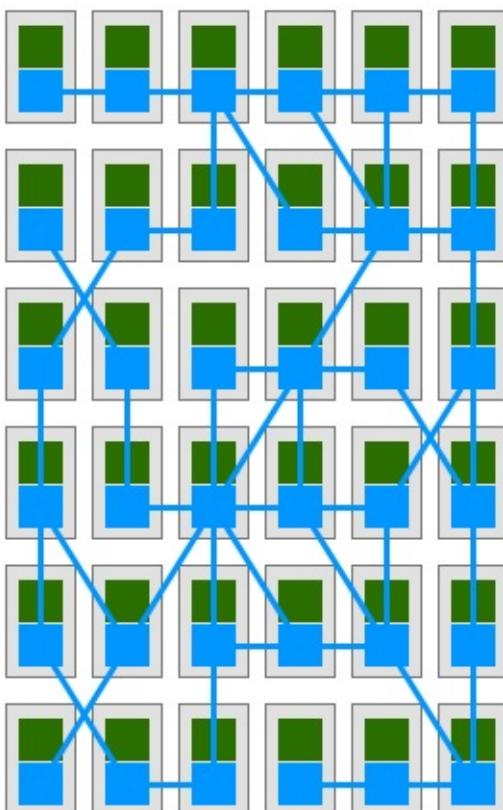


图中应用作为服务的发起方，只需要用最简单的方式将请求发送给本地的服务网格代理，然后网格代理会进行后续的操作，如服务发现，负载均衡，最后将请求转发给目标服务。

当有大量服务相互调用时，他们之间的服务调用关系就会形成网格，如下图所示：



在上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。可以看到蓝色的方块和线条组成了整个网格，我们将这个图片旋转90°，就更加明显了：服务网格呈现出一个完整的支撑态势，将所有的服务“架”在网格之上：



服务网格的细节我们今天不详细展开，详细内容大家可以参考网上资料。或者稍后我将会推出一个服务网格的专题，单独深入介绍服务网格。

Istio也可以视为是一种服务网格，在Istio网站上详细解释了这一概念：

如果我们在架构中的服务和网络间透明地注入一层，那么该层将赋予运维人员对所需功能的控制，同时将开发人员从编码实现分布式系统问题中解放出来。通常将这个统一的架构层与服务部署在一起，统称为“服务啮合层”。由于微服务有助于分离各个功能团队，因此服务啮合层有助于将运维人员从应用特性开发和发布过程中分离出来。通过系统地注入代理到微服务间的网络路径中，Istio将迥异的微服务转变成一个集成的服务啮合层。

Istio能做什么？

Istio力图解决前面我们列出的微服务实施后需要面对的问题。

Istio 首先是一个服务网络，但是istio又不仅仅是服务网格：在 Linkerd, Envoy 这样的典型服务网格之上，istio提供了一个完整的解决方案，为整个服务网格提供行为洞察和操作控制，以满足微服务应用程序的多样化需求。

istio在服务网络中统一提供了许多关键功能(以下内容来自官方文档)：

- 流量管理。控制服务之间的流量和API调用的流向，使得调用更可靠，并使网络在恶劣情况下更加健壮。
- 可观察性。了解服务之间的依赖关系，以及它们之间流量的本质和流向，从而提供快速识别问题的能力。
- 策略执行。将组织策略应用于服务之间的互动，确保访问策略得以执行，资源在消费者之间良好分配。策略的更改是通过配置网格而不是修改应用程序代码。
- 服务身份和安全。为网格中的服务提供可验证身份，并提供保护服务流量的能力，使其可以在不同可信度的网络上流转。

除此之外，Istio针对可扩展性进行了设计，以满足不同的部署需要：

- 平台支持。Istio旨在各种环境中运行，包括跨云，预置，Kubernetes，Mesos等。最初专注于Kubernetes，但很快将支持其他环境。
- 集成和定制。策略执行组件可以扩展和定制，以便与现有的ACL，日志，监控，配额，审核等解决方案集成。

这些功能极大的减少了应用程序代码，底层平台和策略之间的耦合，使微服务更容易实现，

istio的真正价值

我们在上面摘抄了istio官方的大段文档说明，洋洋洒洒的列出了istio的大把大把高大上的功能。但是这些都不是重点！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，关键在哪里？

我们不妨设想一下，在我们平时理解的微服务开发过程中，在没有istio这样的服务网格的情况下，我们要如何开发我们的应用程序，才可以做到我们前面列出的这些丰富多彩的功能？这数以几十记的各种特性，如何才可以加入到我们的应用程序？

无外乎，找个spring cloud或者dubbo的成熟框架，直接搞定服务注册，服务发现，负载均衡，熔断等基础功能。然后自己开发服务路由等高级功能，接入zipkin等apm做全链路监控，自己做加密，认证，授权，想办法搞定灰度方案，用redis等实现限速，配额。诸如此类，一大堆的事情，都需要自己做，无论是找开源项目还是自己操刀，最后整出一个带有一大堆功能的应用程序，上线部署。然后给个配置说明到运维，告诉他说如何需要灰度，要如何如何，如果要限速，配置哪里哪里。

这些工作，我相信做微服务落地的公司，基本都跑不掉，需求是现实存在的。无非能否实现，以及实现多少的问题，但是毫无疑问的是，要做到这些，绝对不是一件容易的事情。

问题是，即使费力做到这些，事情到这里还没有完：运维跑来提了点要求，在他看来很合理的要求，比如说，简单点的加个黑名单，复杂点的要做个特殊的灰度：将来自iphone的用户流量导1%到staging环境的2.0新版本...



这里就有一个很严肃的问题，给每个业务程序的开发人员：你到底想往你的业务程序里面塞多少管理和运维的功能？就算你hold得住技术和时间，你有能力一个一个的满足各种运维和管理的需求吗？当你发现你开始疲于响应各种非功能性的需求时，就该开始反省了：我们开发的是业务程序，它的核心价值在业务逻辑的处理和实现，将如此之多的时间精力花费在这些非业务功能上，这真的合理吗？而且即使是在实现层面，微服务实施时，最重要的是如何划分微服务，如何制定接口协议，你该如何分配你有限的时间和资源？

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。



总结:

istio 大幅降低微服务架构下应用程序的开发难度，势必极大的推动微服务的普及。

个人乐观估计，随着 istio 的成熟，微服务开发领域将迎来一次颠覆性的变革。

后面我们在介绍 istio 的架构和功能模块时，大家可以了解到 istio 是如何做到这些的。

开发团队

在开始介绍 istio 的架构之前，我们再详细介绍一下 istio 的开发团队，看看背后的大佬。

首先，istio 的开发团队主要来自 google, IBM 和 Lyft. 摘抄一段官方八股：

基于我们为内部和企业客户构建和运营大规模微服务的常见经验，Google，IBM 和 Lyft 联手创建 Istio，希望为微服务开发和维护提供可靠的基础。

Google 和 IBM 相信不需要介绍了，在 istio 项目中这两个公司是绝对主力。举个例子，下图是 istio Working Group 的成员列表：

Master Working Group List

Name	Leads	Mailing List	Example Topics
Core	Sven Mawson (Google), Louis Ryan (Google), Martin Taillefer (Google), Shriram Rajagopalan (IBM), Dan Berg (IBM)	istio-core@	Configuration, Performance, Stability
Security	Wencheng Lu (Google), Etailev-Ran (IBM), Michael Elder (IBM)	istio-security@	Service-to-service Auth, Identity/CA/SecretStore plugins, Identity Federation, End User Auth, Authority Delegation, Auditing
Networking	Andra Cismaru (Google), Kuat Yessenov (Google), Shriram Rajagopalan (IBM), Christopher Luciano (IBM)	istio-networking@	Pilot integration, TCP Support, Additional L7 protocols, Proxy injection
Environments	Costin Manolache (Google), Laurent Demaily (Google), Jose Ortiz (IBM)	istio-environments@	Raw VM support, Hybrid Mesh, Mac/Windows support, Cloud Foundry integration
Integrations	Martin Taillefer (Google), Todd Kaplinger (IBM)	istio-integrations@	Mixer Adapter Model, Rate Limiting, Tracing, Monitoring, Logging
API Management	Martin Taillefer (Google), Jason Allor (Google), Tony Ffrench (IBM)	istio-api-management@	API Keys, Content Mediation, Content Translation, OpenAPI Ingestion

数一下，总共18人，10个google, 8个IBM. 注意这里没有Lyft出现，因为Lyft的贡献主要集中在Envoy.

google

Istio来自鼎鼎大名的GCP/Google Cloud Platform, 这里诞生了同样大名鼎鼎的 app engine, cloud engine等重量级产品.

google为istio带来了Kubernetes和gRPC, 还有和Envoy相关的特性如安全,性能和扩展性.

八卦: 负责istio的GCP产品经理Varun Talwar, 同时也负责gRPC项目, 所以关注gRPC的同学(比如我自己)可以不用担心: istio对gRPC的支持必然是没有问题的.

IBM

IBM 的团队同来来自IBM云平台, IBM的贡献是:

除了开发Istio控制面板之外, 还有和Envoy相关的其他特性如跨服务版本的流量切分, 分布式请求追踪(zipkin)和失败注入.

Lyft

Lyft的贡献主要集中在Envoy代理，这是Lyft开源的服务网格，基于C++。据说Envoy在Lyft可以管理超过100个服务，跨越10000个虚拟机，每秒处理2百万请求。本周最新消息，Envoy刚刚加入CNCF，成为该基金会的第十一个项目。

最后，在istio的介绍完成之后，我们开始下一节内容，istio的架构。

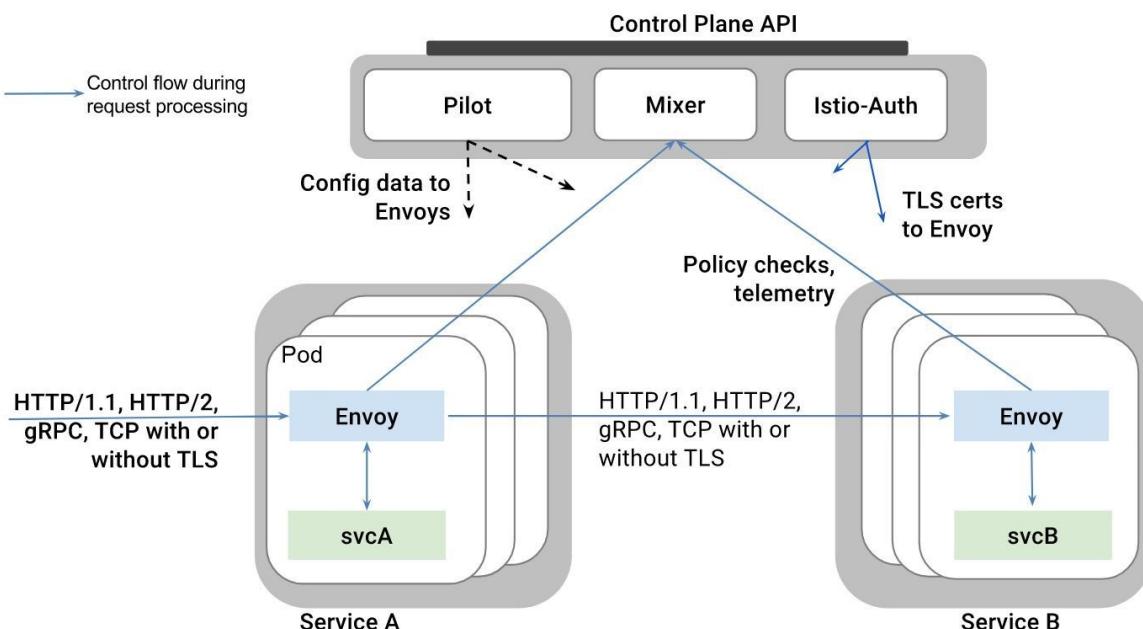
架构

整体架构

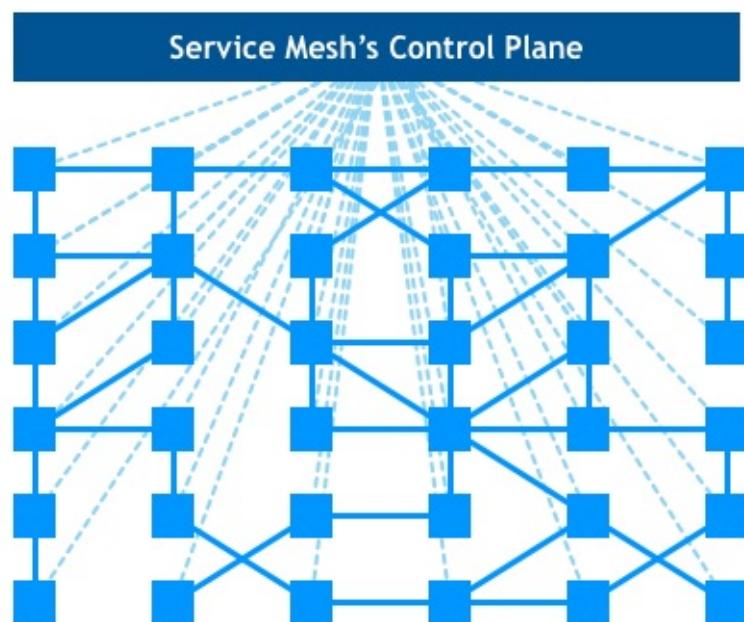
Istio服务网格逻辑上分为数据面板和控制面板。

- 数据面板由一组智能代理（Envoy）组成，代理部署为边车，调解和控制微服务之间所有的网络通信。
- 控制面板负责管理和配置代理来路由流量，以及在运行时执行策略。

下图为istio的架构详细分解图：



这是宏观视图，可以更形象的展示istio两个面板的功能和合作：



以下分别介绍 istio 中的主要模块 Envoy/Mixer/Pilot/Auth.

Envoy



以下介绍内容来自istio官方文档：

Istio 使用 Envoy 代理的扩展版本，Envoy 是以C++开发的高性能代理，用于调解服务网格中所有服务的所有入站和出站流量。

Istio利用了Envoy的许多内置功能，例如动态服务发现，负载均衡，TLS termination，HTTP/2&gRPC代理，熔断器，健康检查，基于百分比流量拆分的分段推出，故障注入和丰富的metrics。

Envoy实现了过滤和路由、服务发现、健康检查，提供了具有弹性的负载均衡。它在安全上支持TLS，在通信方面支持gRPC.

概括说，Envoy 提供的是服务间网络通讯的能力，包括(以下均可支持TLS)：

- HTTP／1.1
- HTTP/2
- gRPC
- TCP

以及网络通讯直接相关的功能：

- 服务发现：从Pilot得到服务发现信息
- 过滤
- 负载均衡
- 健康检查
- 执行路由规则(Rule): 规则来自Polit,包括路由和目的地策略
- 加密和认证: TLS certs来自 istio-Auth

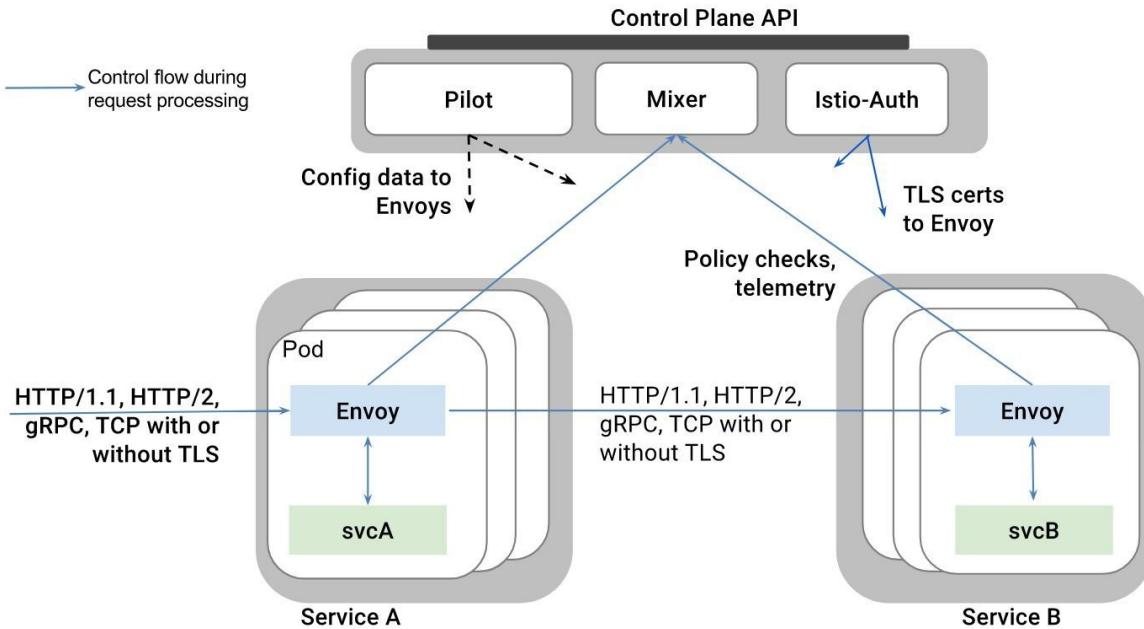
此外, Envoy 也吐出各种数据给Mixer:

- metrics
- logging
- distribution trace: 目前支持 zipkin

总结: Envoy 是 istio 中负责"干活"的模块,如果将整个 istio 体系比喻为一个施工队,那么 Envoy 就是最底层负责搬砖的民工,所有体力活都由 Envoy 完成. 所有需要控制,决策,管理的功能都是其他模块来负责,然后配置给 Envoy.

Istio架构回顾

在继续介绍istio其他的模块之前，我们来回顾一下Istio的架构.前面我们提到，istio 服务网格分为两大块:数据面板和控制面板。



我们刚刚介绍的 Envoy, 在istio中扮演的就是数据面板, 而其他我们下面将要陆续介绍的Mixer, Pilot和Auth属于控制面板. 上面我给出了一个类比: istio 中 Envoy (或者说数据面板)扮演的角色是底层干活的民工, 而该让这些民工如何工作, 由包工头控制面板来负责完成.

在istio的架构中, 这两个模块的分工非常的清晰, 体现在架构上也是经纬分明: Mixer, Pilot和Auth这三个模块都是Go语言开发, 代码托管在github上, 三个仓库分别是 istio/mixer, istio/pilot/auth. 而Envoy来自lyft, 编程语言是c++ 11, 代码托管在github但不是istio下. 从团队分工看, google和IBM关注于控制面板中的Mixer, Pilot和Auth, 而Lyft继续专注于Envoy.

Istio的这个架构设计, 将底层 service mesh的具体实现, 和istio核心的控制面板拆分开. 从而使得istio可以借助成熟的Envoy快速推出产品, 未来如果有更好的service mesh方案也方便集成.

Envoy的竞争者

谈到这里, 我们聊一下目前市面上Envoy之外的另外一个service mesh成熟产品: 基于scala的Linkerd。linkerd的功能和定位和 Envoy 非常相似, 而且就在今年上半年成功进入CNCF. 而在istio推出之后, linkerd做了一个很有意思的动作: istio推出了和istio的集成, 实际为替换Envoy作为istio的数据面板, 和istio的控制面板对接.

回到istio的架构图, 将这幅图中的 Envoy 字样替换为 Linkerd 即可. 另外还有不在图中表示的Linkerd Ingress / Linkerd Egress 用于替代 Envoy 实现k8s的Ingress/Egress.

本周最新消息: nginx推出了自己的服务网格产品nginxmesh, 功能类似, 比较有意思的地方是nginxmesh一出来就直接宣布要和istio集成, 替换Envoy。有兴趣的同学可以去见我本周翻译转载的新闻 [nginx发布微服务平台,OpenShift Ingress控制器和服务网格预览](#)

继续八卦:一出小三上位原配出局的狗血剧情貌似正在酝酿中. 结局如何我等不妨拭目以待. 还是那句话: 没有挖不倒的墙角, 只有不努力的小三! Linkerd, nginxmesh, 加油!

下面开始介绍 istio 中最核心的控制面板.

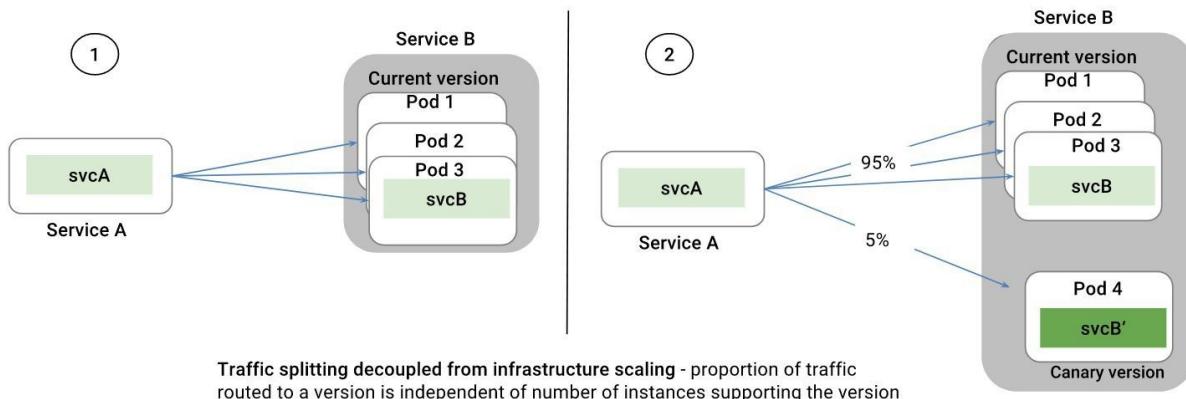
Pilot

流量管理

istio 最核心的功能是流量管理, 前面我们看到的数据面板, 由Envoy组成的服务网格, 将整个服务间通讯和入口/出口请求都承载于其上.

使用Istio的流量管理模型, 本质上将流量和基础设施扩展解耦, 让运维人员通过Pilot指定他们希望流量遵循什么规则, 而不是哪些特定的pod/VM应该接收流量.

对这段话的理解, 可以看下图: 假定我们原有服务B, 部署在Pod1/2/3上, 现在我们部署一个新版本在Pod4在, 我们希望实现切5%的流量到新版本.

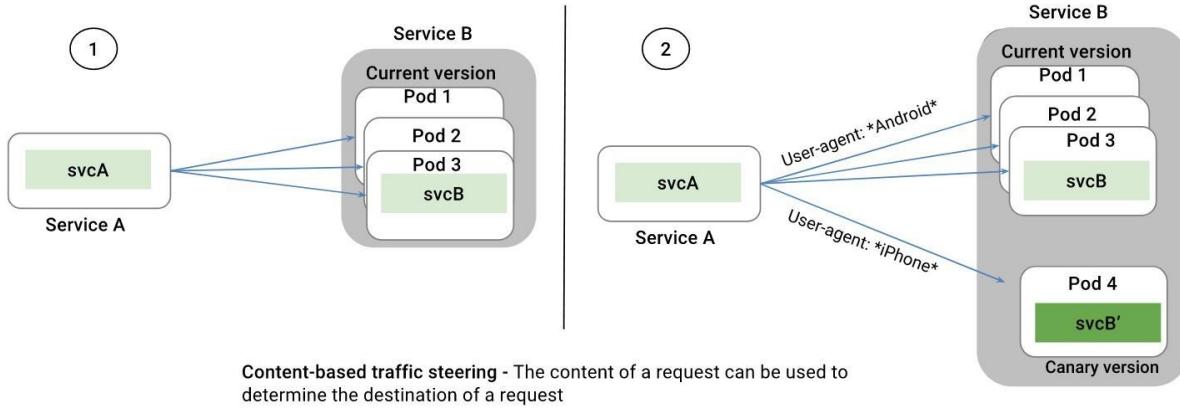


如果以基础设施为基础实现上述5%的流量切分, 则需要通过某些手段将流量切5%到Pod4这个特定的部署单位, 实施时就必须和serviceB的具体部署还有ServiceA访问ServiceB的特定方式紧密联系在一起. 比如如果两个服务之间是用nginx做反向代理, 则需要增加pod4的ip作为upstream, 并调整pod1/2/3/4的权重以实现流量切分.

如果使用istio的流量管理功能, 由于Envoy组成的服务网络完全在istio的控制之下, 因此要实现上述的流量拆分非常简单. 假定原版本为1.0, 新版本为2.0, 只要通过 Pilot 给Envoy 发送一个规则: 2.0版本5%流量, 剩下的给1.0.

这种情况下, 我们无需关注2.0版本的部署, 也无需改动任何技术设置, 更不需要在业务代码中为此提供任何配置支持和代码修改. 一切由 Pilot 和智能Envoy代理搞定。

我们还可以玩的更炫一点, 比如根据请求的内容来源将流量发送到特定版本:



后面我们会介绍如何从请求中提取出User-Agent这样的属性来配合规则进行流量控制。

Pilot的功能概述

我们在前面有强调说, Envoy在其中扮演的负责搬砖的民工角色, 而指挥Envoy工作的民工头就是Pilot模块.

官方文档中对Pilot的功能描述:

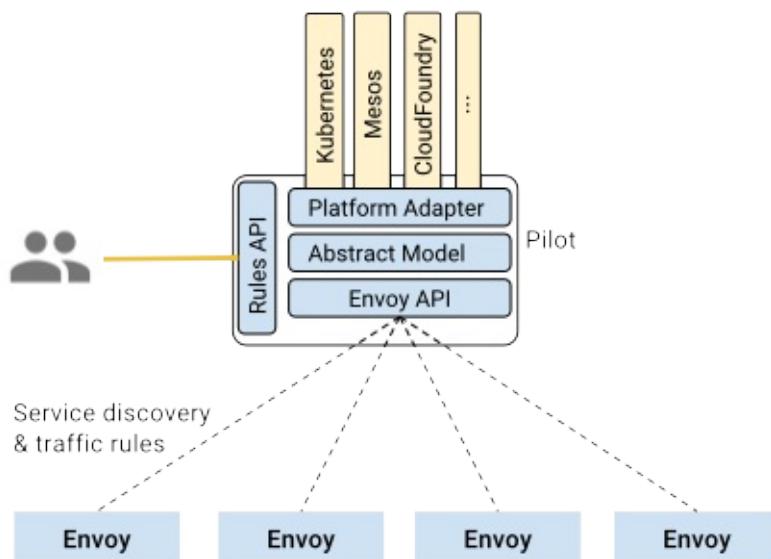
Pilot负责收集和验证配置并将其传播到各种Istio组件。它从Mixer和Envoy中抽取环境特定的实现细节，为他们提供独立于底层平台的用户服务的抽象表示。此外，流量管理规则（即通用4层规则和7层HTTP/gRPC路由规则）可以在运行时通过Pilot进行编程。

每个Envoy实例根据其从Pilot获得的信息以及其负载均衡池中的其他实例的定期健康检查来维护负载均衡信息，从而允许其在目标实例之间智能分配流量，同时遵循其指定的路由规则。

Pilot负责在Istio服务网格中部署的Envoy实例的生命周期。

Pilot的架构

下图是Pilot的架构图:



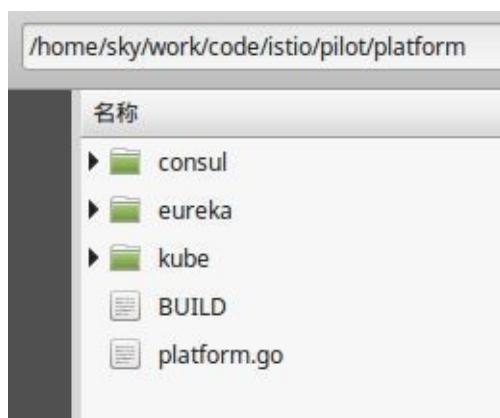
1. Envoy API 负责和 Envoy 的通讯，主要是发送服务发现信息和流量控制规则给 Envoy
2. Envoy 提供服务发现，负载均衡池和路由表的动态更新的 API。这些 API 将 istio 和 Envoy 的实现解耦。(另外，也使得 Linkerd 之类的其他服务网络实现得以平滑接管 Envoy)
3. Pilot 定了一个抽象模型，以从特定平台细节中解耦，为跨平台提供基础。
4. Platform Adapter 则是这个抽象模型的现实实现版本，用于对接外部的不同平台
5. 最后是 Rules API，提供接口给外部调用以管理 Pilot，包括命令行工具 istioctl 以及未来可能出现的第三方管理界面

服务规范和实现

Pilot 架构中，最重要的是 Abstract Model 和 Platform Adapter，我们详细介绍。

- **Abstract Model:** 是对服务网格中“服务”的规范表示，即定义在 istio 中什么是服务，这个规范独立于底层平台。
- **Platform Adapter:** 这里有各种平台的实现，目前主要是 Kubernetes，另外最新的 0.2 版本的代码中出现了 Consul 和 Eureka。

我们看一下 Pilot 0.2 的代码，pilot/platform 目录下：



瞄一眼 platform.go：

```
// ServiceRegistry 定义支持服务注册的底层平台
type ServiceRegistry string

const (
    // KubernetesRegistry environment flag
    KubernetesRegistry ServiceRegistry = "Kubernetes"
    // ConsulRegistry environment flag
    ConsulRegistry ServiceRegistry = "Consul"
    // EurekaRegistry environment flag
    EurekaRegistry ServiceRegistry = "Eureka"
)
```

服务规范的定义在 `modle/service.go` 中:

```
type Service struct {
    Hostname string `json:"hostname"`
    Address string `json:"address,omitempty"`
    Ports PortList `json:"ports,omitempty"`
    ExternalName string `json:"external"`
    ServiceAccounts []string `json:"serviceaccounts,omitempty"`
}
```

由于时间有限, 代码部分我们不深入, 只是通过上面的两段代码来展示pilot中对服务的规范定义和目前的几个实现.

暂时而言(当前版本是0.1.6, 0.2版本尚未正式发布), 目前 istio 只支持k8s一种服务发现机制.

备注: Consul的实现据说主要是为了支持后面将要支持的Cloud Foundry, Eureka 没有找到资料. Etcd3 的支持还在issue列表中, 看issue记录争执中.

pilot功能

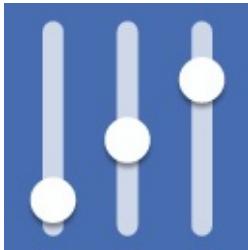
基于上述的架构设计, pilot提供以下重要功能:

- 请求路由
- 服务发现和负载均衡
- 故障处理
- 故障注入
- 规则配置

由于时间限制, 今天不逐个展开详细介绍每个功能的详情. 大家通过名字就大概可以知道是什么, 如果希望了解详情可以关注之后的分享. 或者查阅官方文档的介绍.

Mixer

Mixer翻译成中文是混音器，下面是它的图标：

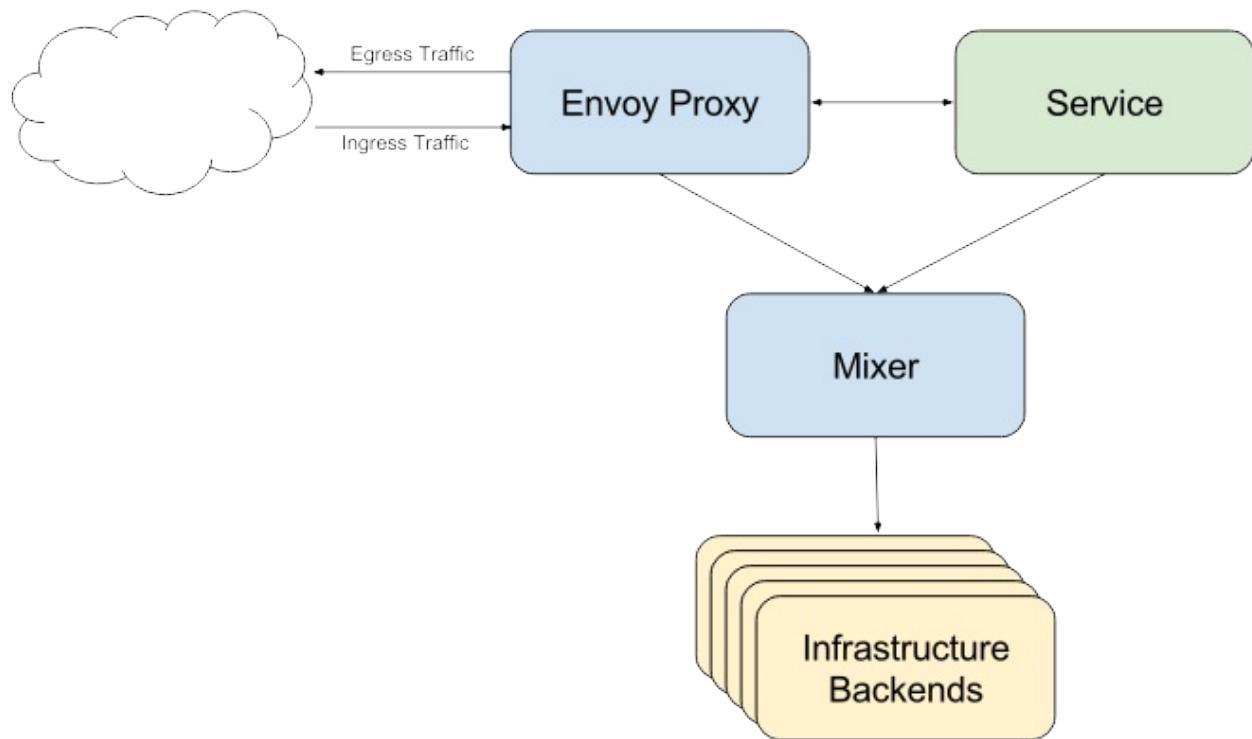


功能概括：Mixer负责在服务网格上执行访问控制和使用策略，并收集Envoy代理和其他服务的遥测数据。

Mixer的设计背景

我们的系统通常会基于大量的基础设施而构建，这些基础设施的后端服务为业务服务提供各种支持功能。包括访问控制系统，遥测捕获系统，配额执行系统，计费系统等。在传统设计中，服务直接与这些后端系统集成，容易产生硬耦合。

在istio中，为了避免应用程序的微服务和基础设施的后端服务之间的耦合，提供了 Mixer 作为两者的通用中介层：



Mixer 设计将策略决策从应用层移出并用配置替代，并在运维人员控制下。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进行相当简单的集成，然后 Mixer 负责与后端系统连接。

特别提醒：Mixer不是为了在基础设施后端之上创建一个抽象层或者可移植性层。也不是试图定义一个通用的logging API，通用的metric API，通用的计费API等等。

Mixer的设计目标是减少业务系统的复杂性，将策略逻辑从业务的微服务的代码转移到**Mixer**中，并且改为让运维人员控制。

Mixer的功能

Mixer 提供三个核心功能：

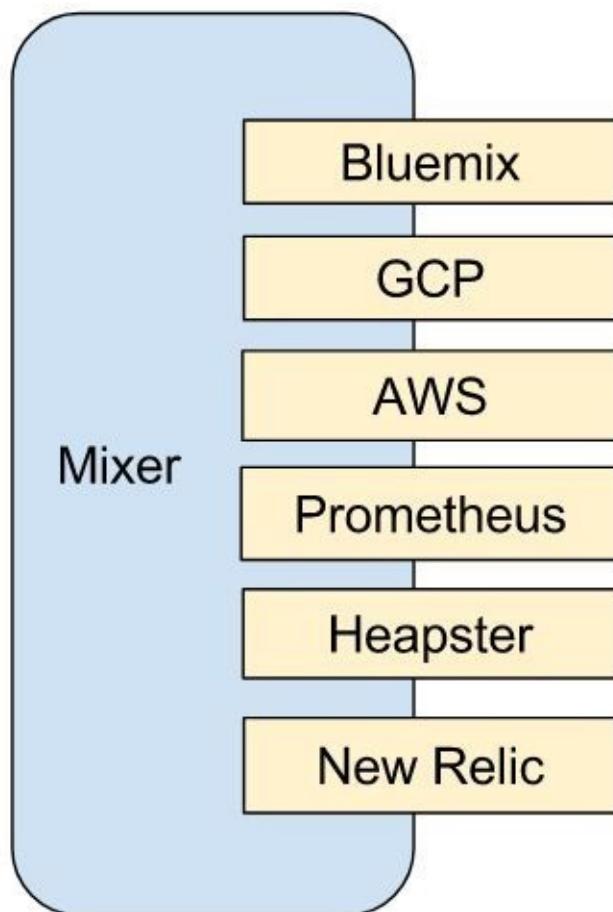
- 前提条件检查。允许服务在响应来自服务消费者的传入请求之前验证一些前提条件。前提条件包括认证，黑白名单，ACL检查等等。
- 配额管理。使服务能够在多个维度上分配和释放配额。典型例子如限速。
- 遥测报告。使服务能够上报日志和监控。

在Istio内，**Envoy**重度依赖**Mixer**。

Mixer的适配器

Mixer是高度模块化和可扩展的组件。其中一个关键功能是抽象出不同策略和遥测后端系统的细节，允许**Envoy**和基于Istio的服务与这些后端无关，从而保持他们的可移植。

Mixer在处理不同基础设施后端的灵活性是通过使用通用插件模型实现的。单个的插件被称为适配器，它们允许**Mixer**与不同的基础设施后端连接，这些后台可提供核心功能，例如日志，监控，配额，ACL检查等。适配器使**Mixer**能够暴露一致的API，与使用的后端无关。在运行时通过配置确定确切的适配器套件，并且可以轻松指向新的或定制的基础设施后端。



这个图是官网给的，列出的功能不多，我从github的代码中抓个图给大家展示一下目前已有的 mixer adapter:

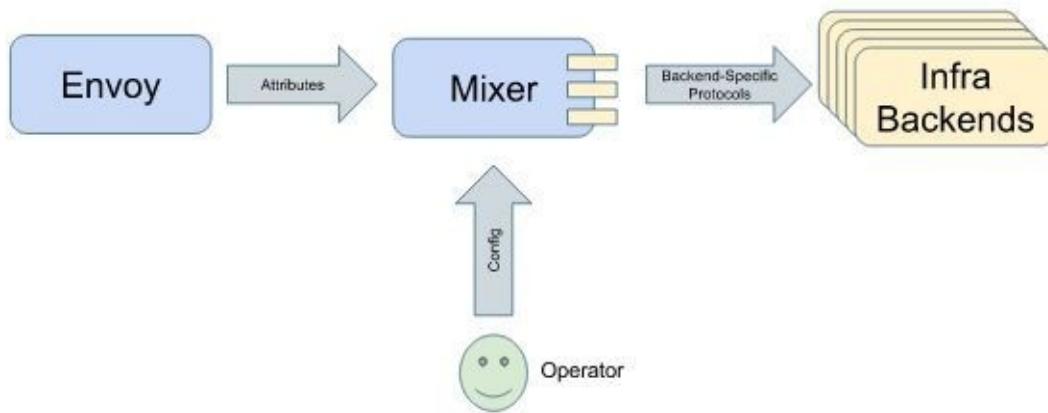
denier
denyChecker
genericListChecker
ipListChecker
kubernetes
list
memQuota
memquota2
noop
noopLegacy
prometheus
prometheus2
redisquota
serviceControl
stackdriver
statsd
statsd2
stdio
stdioLogger
svcctrl

Mixer的工作方式

Istio使用 `属性` 来控制在服务网格中运行的服务的运行时行为。属性是描述入口和出口流量的有名称和类型的元数据片段，以及此流量发生的环境。Istio属性携带特定信息片段，例如：

```
request.path: xyz/abc
request.size: 234
request.time: 12:34:56.789 04/17/2017
source.ip: 192.168.0.1
target.service: example
```

请求处理过程中，属性由Envoy收集并发送给Mixer，Mixer中根据运维人员设置的配置来处理属性。基于这些属性，Mixer会产生对各种基础设施后端的调用。



Mixer设计有一套强大(也很复杂,堪称istio中最复杂的一个部分)的配置模型来配置适配器的工作方式,设计有适配器,切面,属性表达式,选择器,描述符,manifests等一堆概念.

由于时间所限,今天不展开这块内容,这里给出两个简单的例子让大家对mixer的配置有个感性的认识:

1. 这是一个ip地址检查的adapter,实现类似黑名单或者白名单的功能:

```

adapters:
- name: myListChecker      # 这个配置块的用户定义的名称
  kind: lists                # 这个适配器可以使用的切面类型
  impl: ipListChecker        # 要使用的特定适配器组件的名称
  params:
    publisherUrl: https://mylistserver:912
    refreshInterval: 60s
  
```

2. metrics的适配器,将数据报告给Prometheus系统

```

adapters:
- name: myMetricsCollector
  kind: metrics
  impl: prometheus
  
```

3. 定义切面,使用前面定义的 myListChecker 这个adapter 对属性 source.ip 进行黑名单检查

```

aspects:
- kind: lists      # 切面的类型
  adapter: myListChecker  # 实现这个切面的适配器
  params:
    blacklist: true
    checkExpression: source.ip
  
```

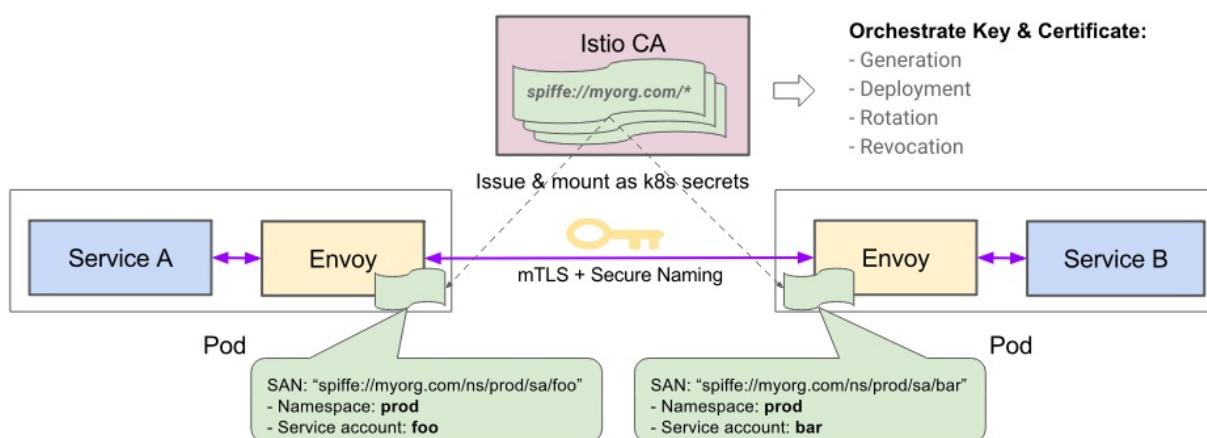
Istio-Auth

Istio-Auth提供强大的服务到服务和终端用户认证，使用交互TLS，内置身份和凭据管理。它可用于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实施策略的能力。

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。

auth的架构

下图展示Istio Auth架构，其中包括三个组件：身份，密钥管理和通信安全。



在这个例子中，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下，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。

目前在Kubernetes上运行的 istio，使用Kubernetes service account/服务帐户来识别运行该服务的人员。

未来将推出的功能

auth在目前的istio版本(0.1.6和即将发布的0.2)中，功能还不是很全，未来则规划有非常多的特性：

- 细粒度授权和审核
- 安全Istio组件（Mixer, Pilot等）

- 集群间服务到服务认证
- 使用JWT/OAuth2/OpenID_Connect终端到服务的认证
- 支持GCP服务帐户和AWS服务帐户
- 非http流量（MySQL，Redis等）支持
- Unix域套接字，用于服务和Envoy之间的本地通信
- 中间代理支持
- 可插拔密钥管理组件

需要提醒的是：这些功能都是不改动业务应用代码的前提下实现的。

回到我们前面的曾经讨论的问题，如果自己来做，完成这些功能大家觉得需要多少工作量？要把所有的业务模块都迁移到具备这些功能的框架和体系中，需要改动多少？而istio，未来就会直接将这些东西摆上我们的餐桌。

未来

前面我们介绍了istio的基本情况，还有istio的架构和主要组件。相信大家对istio应该有了一个初步的认识。

需要提醒的是，istio是一个今年5月才发布0.1.0版本的新鲜出炉的开源项目，目前该项目也才发布到0.1.6正式版本和0.2.2 pre release版本。很多地方还不完善，希望大家可以理解，有点类似于最早期阶段的Kubernetes.

在接下来的时间，我们将简单介绍一下istio后面的一些开发计划和发展预期。

运行环境支持

Istio目前只支持Kubernetes，这是令人比较遗憾的一点。不过istio给出的解释是istio未来会支持在各种环境中运行，只是目前在0.1/0.2这样的初始阶段暂时专注于Kubernetes，但很快会支持其他环境。

注意：Kubernetes平台，除了原生Kubernetes，还有诸如IBM Bluemix Container Service和RedHat OpenShift这样的商业平台。以及google自家的Google Container Engine。这是自家的东西，而且现在k8s/istio/gRPC都被划归到google cloud platform部门，自然会优先支持。

另外istio所说的其他环境指的是：

- mesos：这个估计是大多人非k8s的docker使用者最关心的了，暂时从github上的代码中未见到有开工迹象，但是istio的文档和官方声明都明显说会支持，估计还是希望很大的。
- cloud foundry：这个东东我们国内除了私有云外玩的不多，istio对它的支持似乎已经启动。比如我看到代码中已经有了consul这个服务注册的支持，从issue讨论上看到是说为上

cloud foundry做准备, 因为cloud foundry没有k8s那样的原生服务注册机制.

- VM: 这块没有看到介绍, 但是有看到istio的讨论中提到会支持容器和非容器的混合(hybrid)支持

值得特别指出的是, 目前我还没有看到istio有对docker家的swarm有支持的计划或者讨论, 目前我找到的任何istio的资料中都不存在swarm这个东东。我只能不负责任的解读为: 有人的地方就有江湖, 有江湖就自然会有江湖恩怨。

路线图

按照istio的说法, 他们计划每3个月发布一次新版本, 我们看一下目前得到的一些信息:

- 0.1 版本2017年5月发布, 只支持Kubernetes
- 0.2 即将发布, 当前是0.2.1 pre-release, 也只支持Kubernetes
- 0.3 roadmap上说要支持k8s之外的平台, "Support for Istio meshes without Kubernetes.", 但是具体哪些特性会放在0.3中, 还在讨论中.
- 1.0 版本预计今年年底发布

注: 1.0版本的发布时间官方没有明确给出, 我只是看到官网资料里面有信息透露如下:

"we invite the community to join us in shaping the project as we work toward a 1.0 release later this year."

按照上面给的信息, 简单推算: 应该是9月发0.2, 然后12月发0.3, 但是这就已经是年底了, 所以不排除1.0推迟发布的可能, 或者0.3直接当成1.0发布.

社区支持

虽然istio初出江湖, 乳臭未干, 但是凭借google和IBM的金字招牌, 还有istio前卫而实际的设计理念, 目前已经有很多公司在开始提供对istio的支持或者集成, 这是istio官方页面有记载的:

- Red Hat: Openshift and OpenShift Application Runtimes
- Pivotal: Cloud Foundry
- Weaveworks: Weave Cloud and Weave Net 2.0
- Tigera: Project Calico Network Policy Engine
- Datawire: Ambassador project

然后一些其他外围支持, 从代码中看到的:

- eureka
- consul
- etcd v3: 这个还在争执中, 作为etcd的坚定拥护者, 我对此保持密切关注.

存在的问题

Istio毕竟目前才是0.2.2 pre release版本，毕竟才出来四个月，因此还是存在大量的问题，集中表现为：

1. 只支持k8s，而且要求k8s 1.7.4+，因为使用到k8s的 CustomResourceDefinitions
2. 性能较低，从目前的测试情况看，0.1版本很糟糕，0.2版本有改善
3. 很多功能尚未完成

给大家的建议：可以密切关注Istio的动向，提前做好技术储备。但是，最起码在年底的1.0版本出来之前，别急着上生产环境。

最后的话

感谢大家在今天参与这次的Istio分享，由于时间有限，很多细节无法在今天给大家尽情展开。如果大家对Istio感兴趣，可以之后自行浏览 Istio 的官方网站，我也预期会在之后陆陆续续的给出Istio相关的文章和分享。

最后推荐给大家几个资料：

1. Istio的中文资料: istio.doczh.cn, 这是目前我和其他社区朋友正在一起翻译的Istio官方文档，进度大概30%左右，其中最适合入门了解的 concept 部分已经翻译完成.作为目前市面上唯一的一份Istio中文资料，推荐给大家阅读。
2. Service Mesh的微信技术群，由于服务网格的内容实在太新，不容易找到人交流，所以如果对这个技术有兴趣打算深入研究的，欢迎加入。请联系微信id(xiaoshu062)加群，请备注服务网格。

今天的分享到此结束，感谢大家的全程参与，下次再会！

Linkerd

Tags