



Service Mesh: 下一代微服务

数人云 资深架构师 敖小剑

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）。



service
mesh

1

什么是Service Mesh?

2

Service Mesh的演进历程

3

为何选择Service Mesh?



什么是Service Mesh?

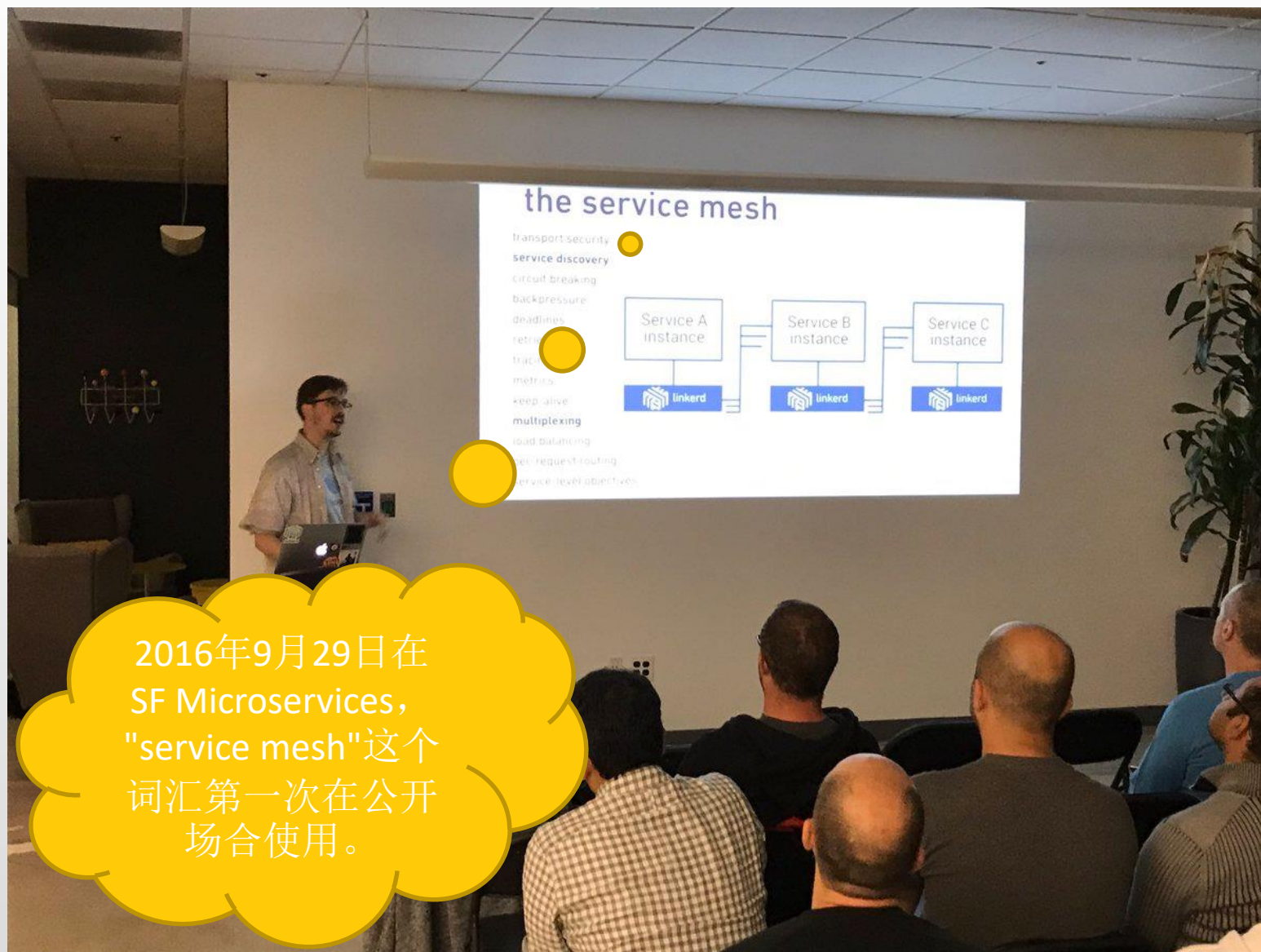
这真的是一个**新**名词

• 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
- 信息来自Buoyant公司的CEO William Morgan的twitter，在服务网格火了之后，他做了这次考古

• 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（不止我一个人这么觉得吧☺），后来逐渐改用“服务网格”



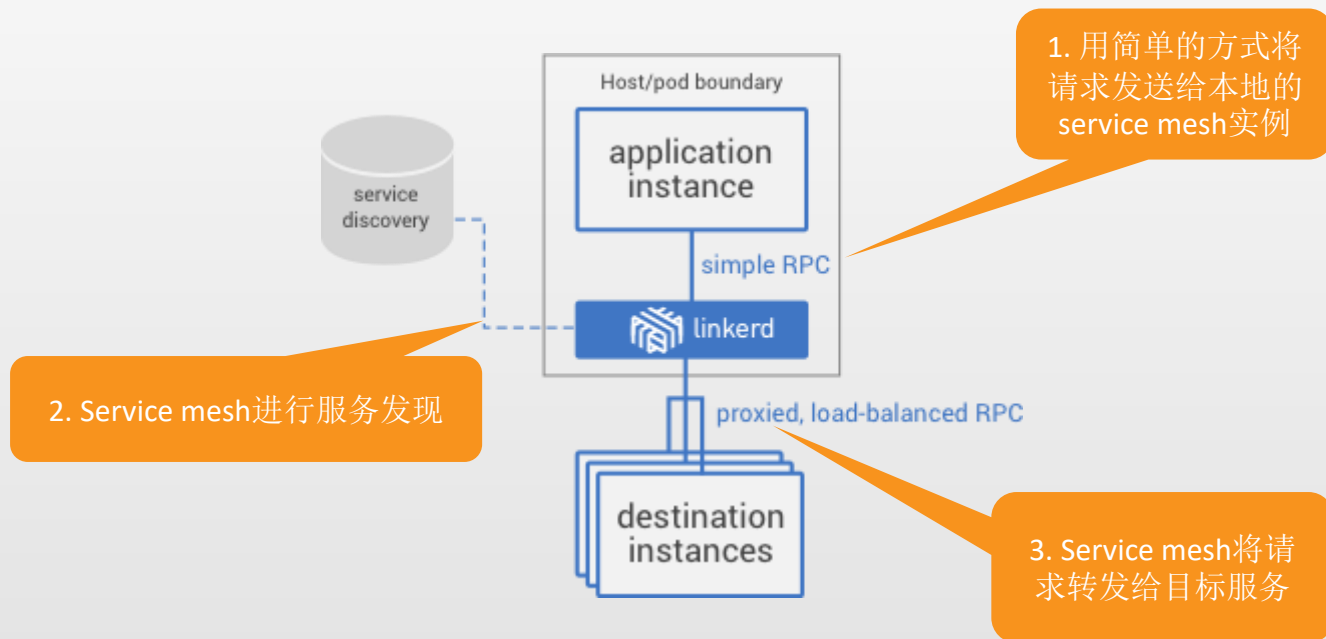
Service Mesh的定义

- **Willian Morgan (Linkerd的CEO) 给出的定义:**

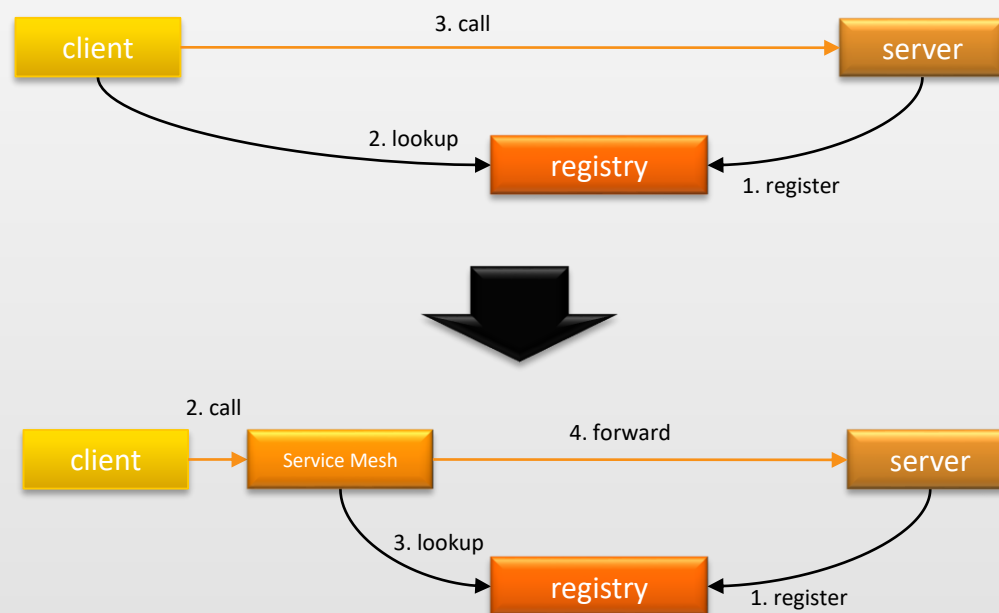
A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as 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**。

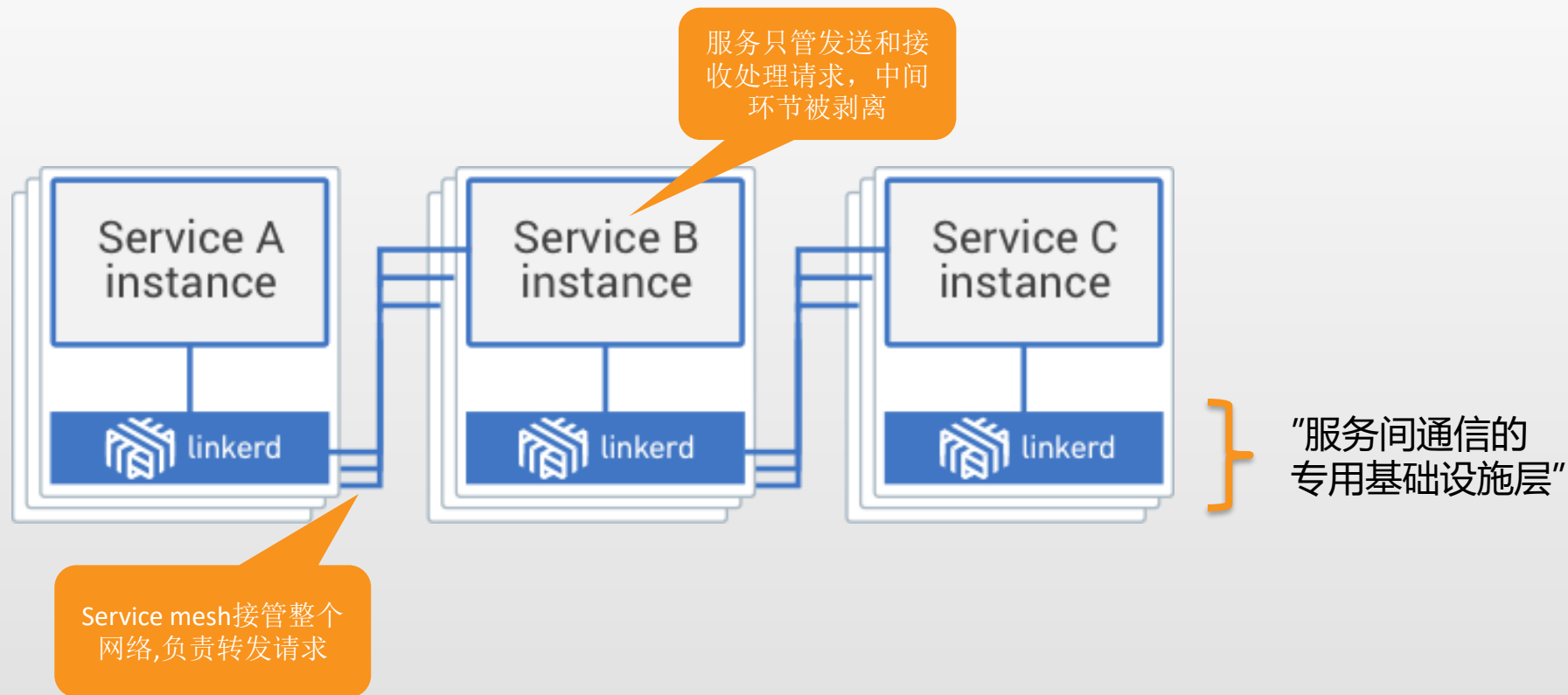
部署模型：单个服务调用，表现为sidecar



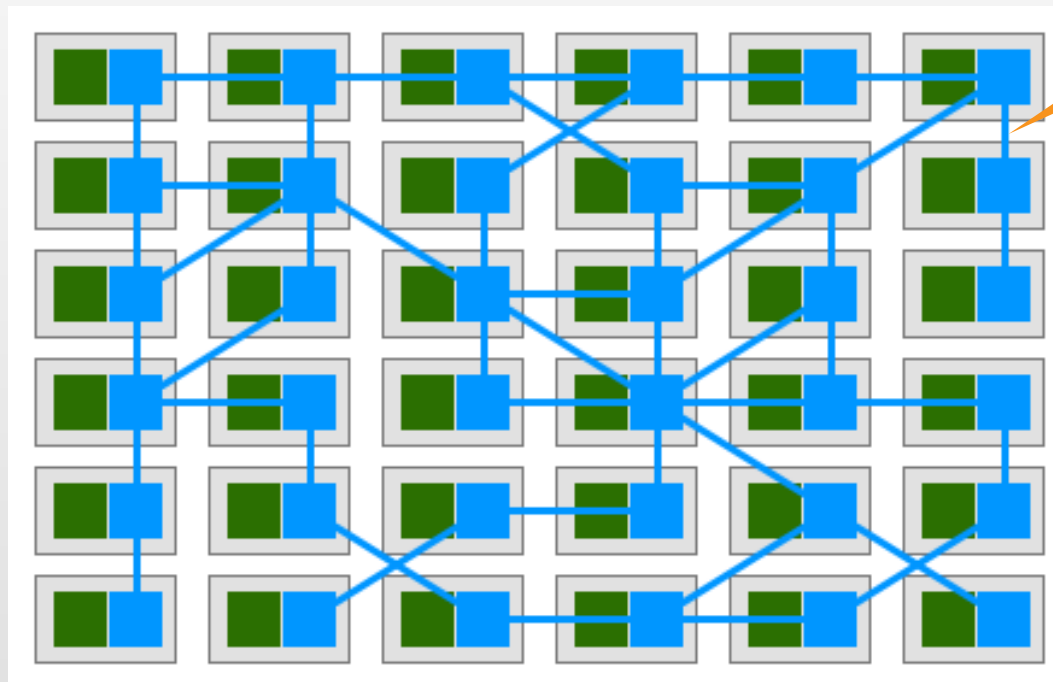
Sidecar: 这个不是新名词



部署模型：多个服务调用，表现为通讯层



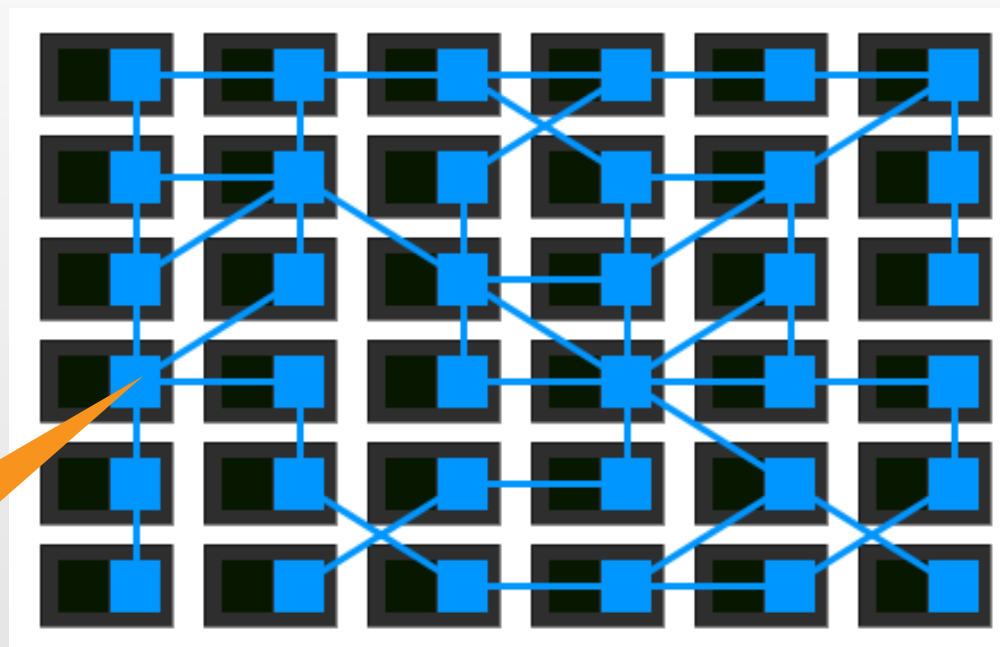
部署模型：有大量服务，表现为网络



Sidecar之间的连接
形成网络

Service Mesh定义回顾

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



Service Mesh和Sidecar的差异:

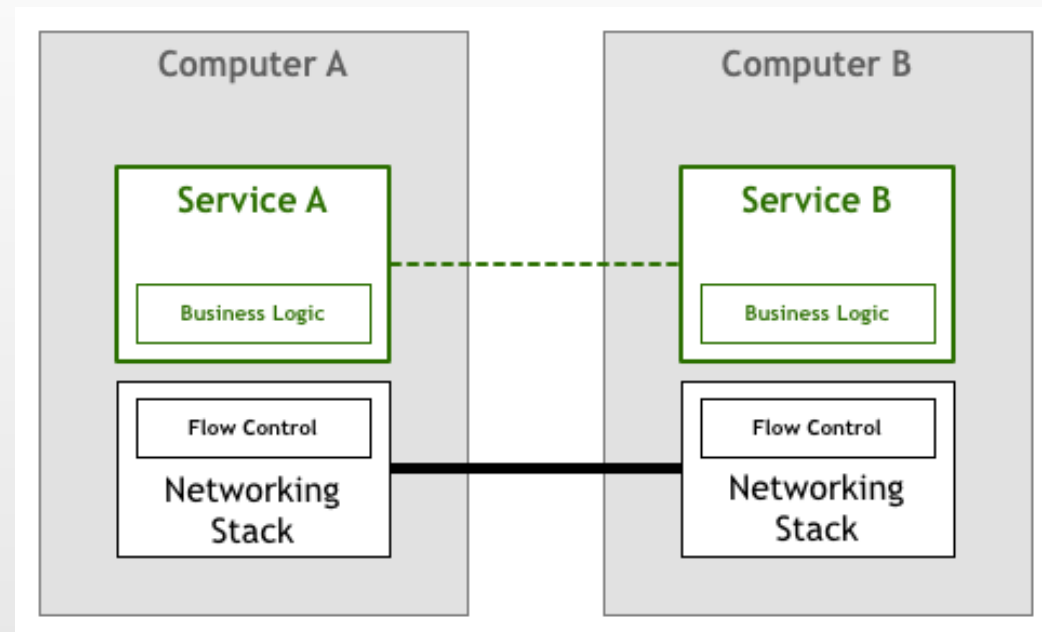
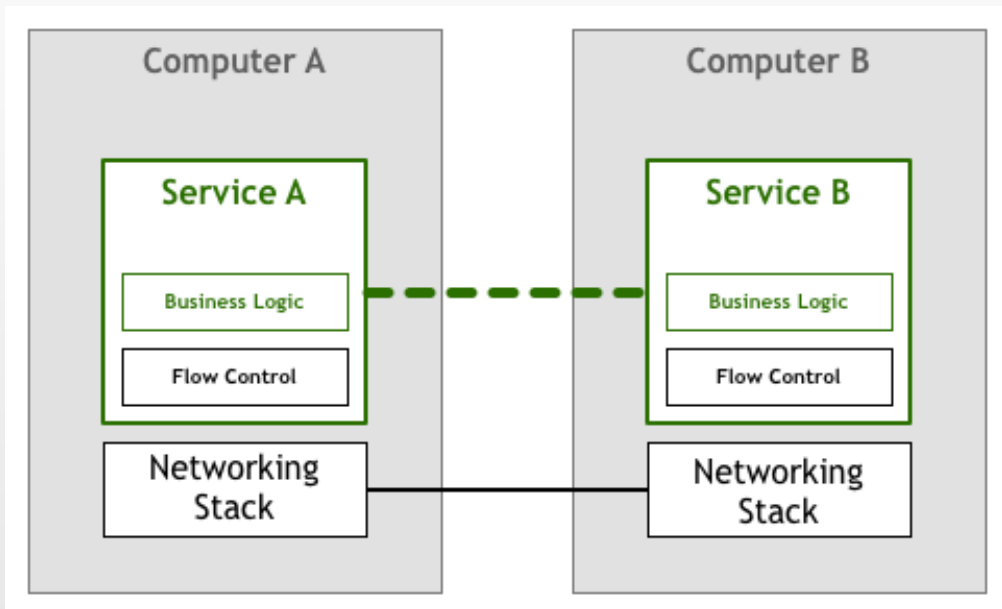
不再将代理视为单独的组件
而是强调由这些代理连接而形成的网络



Service Mesh的演进历程

在讨论为什么要选择Service Mesh之前

Service Mesh的由来1：微服务之前



- **远古时代：第一代网络计算机系统**

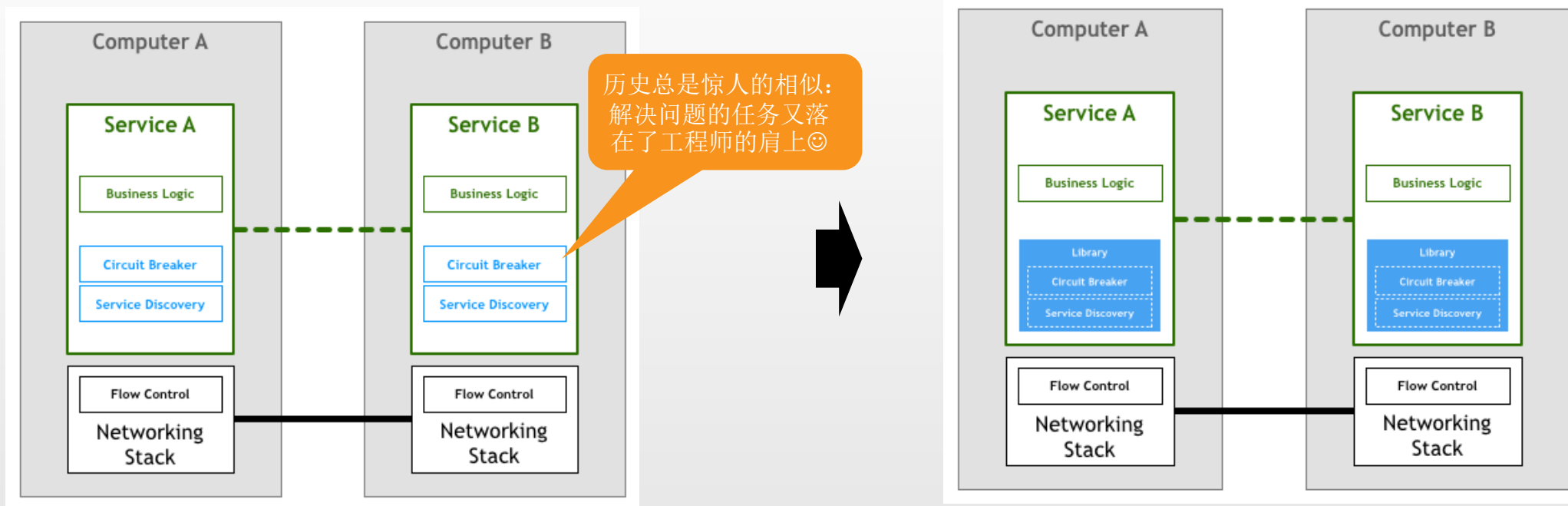
开发人员需要在自己的代码里处理网络通讯的细节问题，如数据包顺序，流量控制等。结果就是应用程序需要处理网络逻辑，导致网络逻辑和业务逻辑混杂在一起

- **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☺

Service Mesh的由来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。结果就是应用程序中业务逻辑外又混杂了一堆非业务的代码。

• 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。

好像一切都很完美的样子? :)

美中不足1: 类库内容有点多, 框架门槛还是稍高

• spring cloud

- spring-cloud-commons
- spring-cloud-Netflix
- spring-cloud-sleuth
- spring-cloud-gateway
- spring-cloud-bus
- spring-cloud-consul
- spring-cloud-config
- spring-cloud-security
- spring-cloud-zookeeper
- spring-cloud-aws
- spring-cloud-cloudfoundry

• Netflix OSS

- eureka
- hystrix
- Turbine
- archaius
- Atlas
- Feign
- Ribbon
- zuul

那么问题来了:

需要多长时间, 才能让整个
开发团队掌握并熟练使用?

不得不面对的现实

业务开发团队往往强项不在技术，而是对业务的理解

业务的核心价值在于业务实现：微服务是手段，不是目标

比微服务框架的更可怕的挑战：微服务拆分，API设计，数据一致性

对于旧有项目，如何进行微服务改造，是个更加痛苦的问题

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

美中不足2: 服务治理相关功能不够齐全

• 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
- 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
 - 可配置的规则

问题: 打算投多少时间和精力进去?

美中不足3: 说好的跨语言呢?

微服务带来的一个巨大优点, 就是容许不同的服务根据实际情况采用不同的编程语言。当我们将代码封装到类库和框架时, 问题又来了:

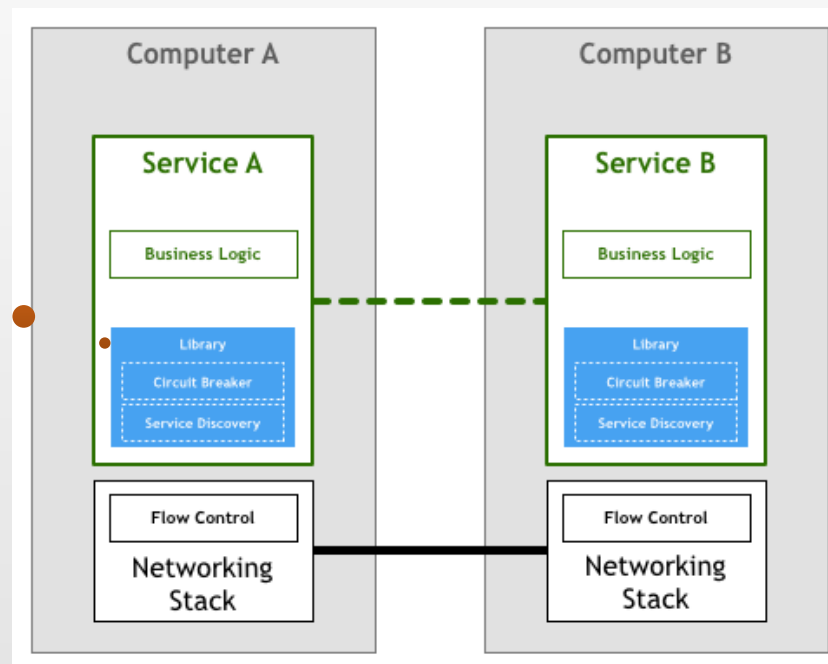
• 主流编程语言

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

我们需要提供多少种语言的类库?

• 新兴编程语言

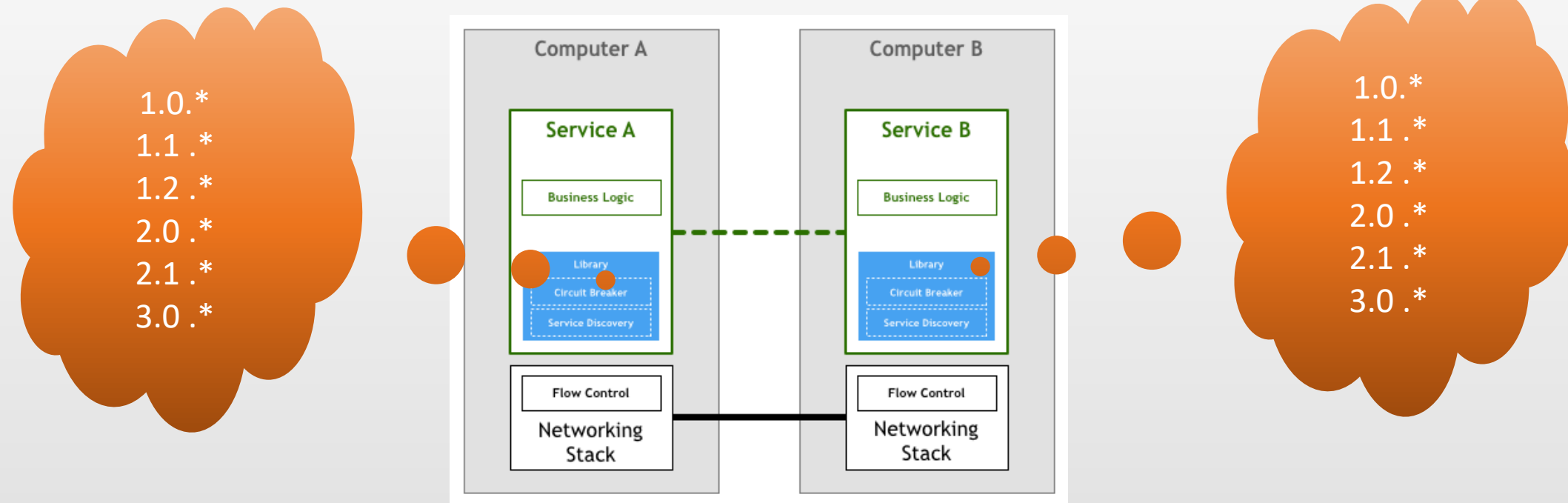
- Golang
- Node.js
- Rust
- R
- Lua/OpenResty



美中不足4: 类库要升级怎么办?

客户端: 数以千计

服务器端: 数以百计



别忘了: 编程语言, 再*N 😊

反省：我们到底在图个啥？

最艰巨的挑战，与服务本身无关，而是服务间的通讯

所有的努力，都是为了保证将请求发到正确的地方

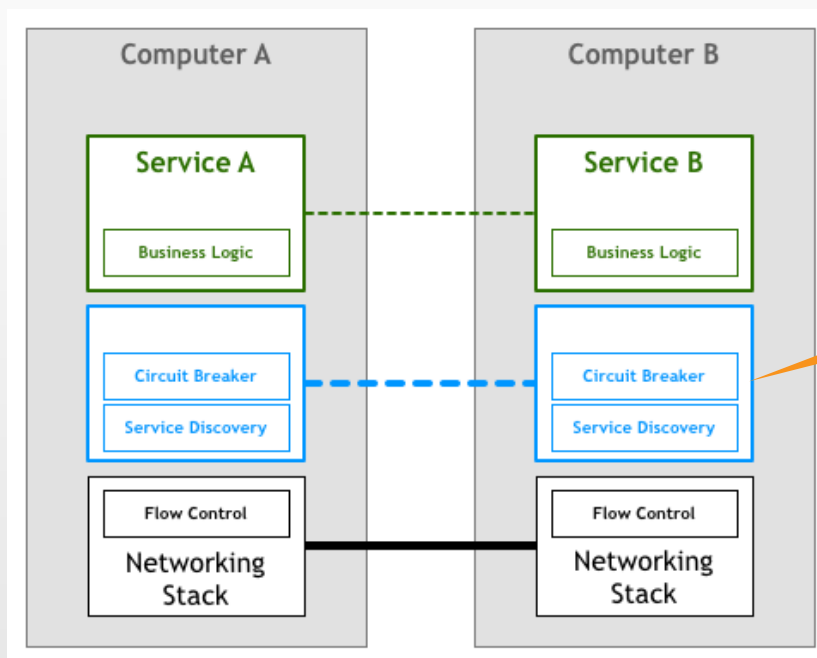
是不是有一种似曾相识的感觉？

我们基于TCP开发应用时，需要关心链路层吗？

我们基于HTTP开发应用时，需要关心TCP层吗？

问题又来了：为什么我们开发微服务时，就一定要这么关心基础设施和类库框架？

Service Mesh的由来3: 技术栈下移, 原始代理出现



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, 代理实现必须的特性。

但是: 功能过于简陋, 而且代理通常部署在服务器端

Service Mesh的由来4: Sidecar出现

- **Airbnb**

2013年, Airbnb开发了Synapse和Nerve

- **Netflix**

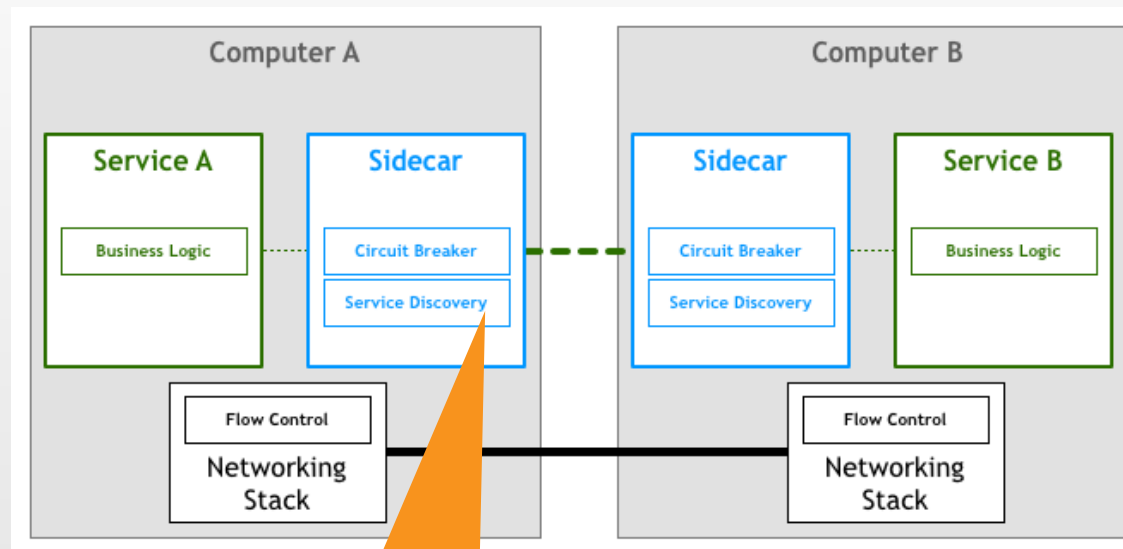
2014年, Netflix发布了Prana

- **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



Sidecar开始和客户端一对一的部署在一起了

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，接入非Java语言，解决业务部门不愿意升级的问题



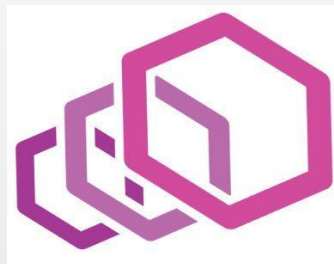
有特定
背景和需求
无法通用

Service Mesh的由来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**

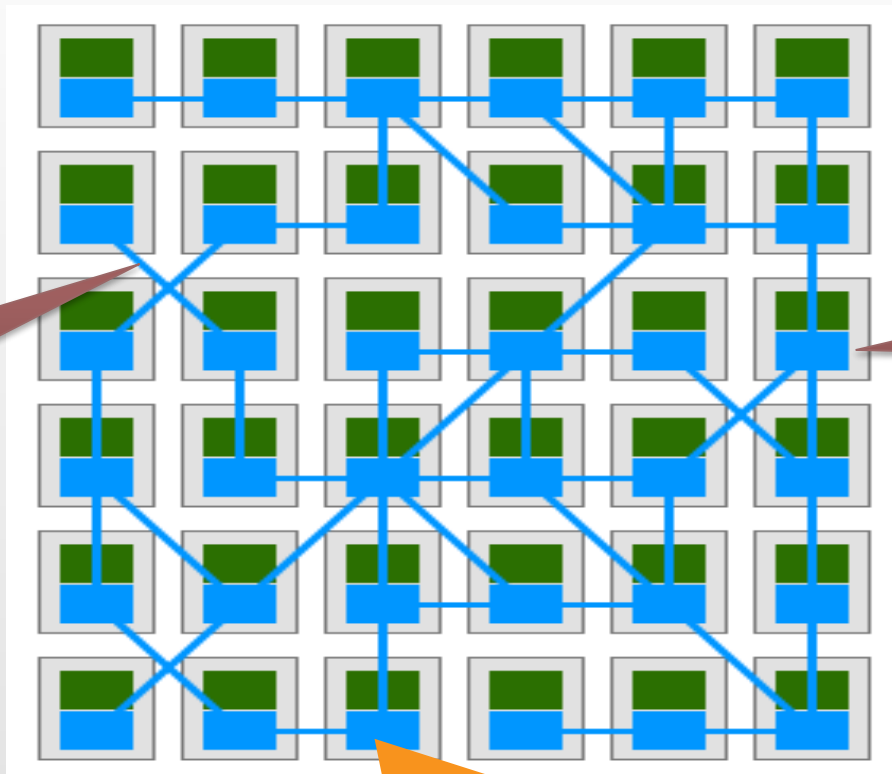
- 来自Lyft, c++语言
- 2016年9月13日, 1.0版本发布
- 2017年9月14日, 加入CNCF



- **nginmesh**

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

Service Mesh和Sidecar的差异



1.不再视为单独的组件，而是强调连接形成的网络

2.Service mesh是通用组件

3.Sidecar是可选的，容许直连。但是Service mesh要求完全掌控所有流量

Service Mesh的由来6: Istio王者风范

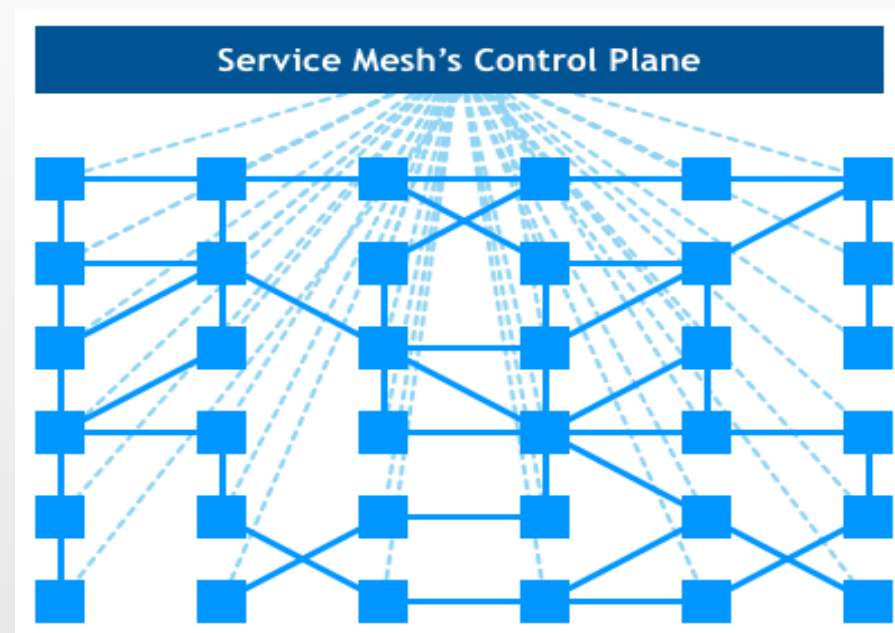
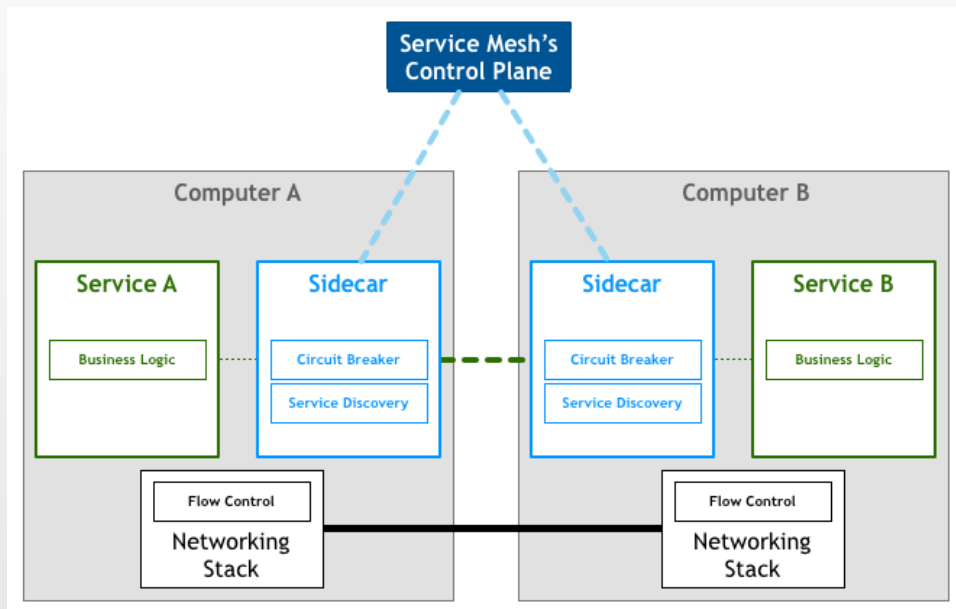


- **Istio**

- 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



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。

Istio为此带来了集中式的控制面板。

• 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的架构

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

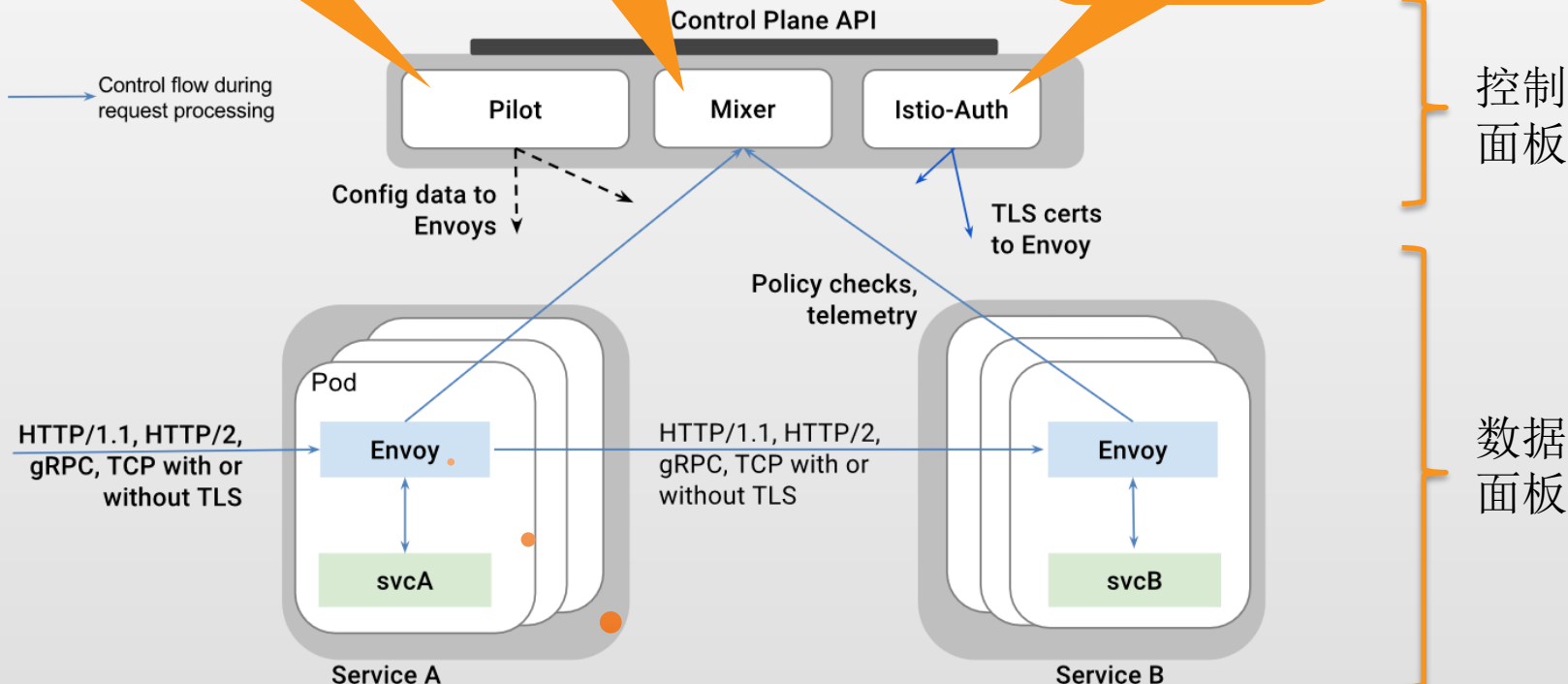
服务发现, 配置, 底层抽象, 编码规则, 管理Envoy

微服务和基础设施的中介层, 将策略决策从应用移出并用配置替代

提供服务间认证和终端用户认证使用交互TLS, 内置身份和凭据管理

- 前提条件检查: 如认证, 黑白名单, ACL检查
- 配额管理: 限流
- 遥测报告: 日志, 监控, metrics

- 升级流量 (加密)
- 身份认证
- 密钥管理
- 通讯安全
- 访问控制
 - 基于属性
 - 基于角色
 - 授权钩子
- 审计
- 监控资源的使用者



控制面板

数据面板

标配Envoy, 但是Linkerd和nginmesh都在istio集成

Istio: Service Mesh集大成者

Google Cloud Platform
IBM Cloud Platform
Lyft Envoy Team

出身名门



Linkerd/nginxmesh选择集成而非对抗
Red hat/Pivotal/Weaveworks/Tigera/Datawire
背后有CNCF，还有即将一统江湖的k8s

社区支持



微服务/容器如日中天，Cloud Native大势已成
传统企业互联网技术转型大潮汹涌却又先天技术积累不足

运势而生

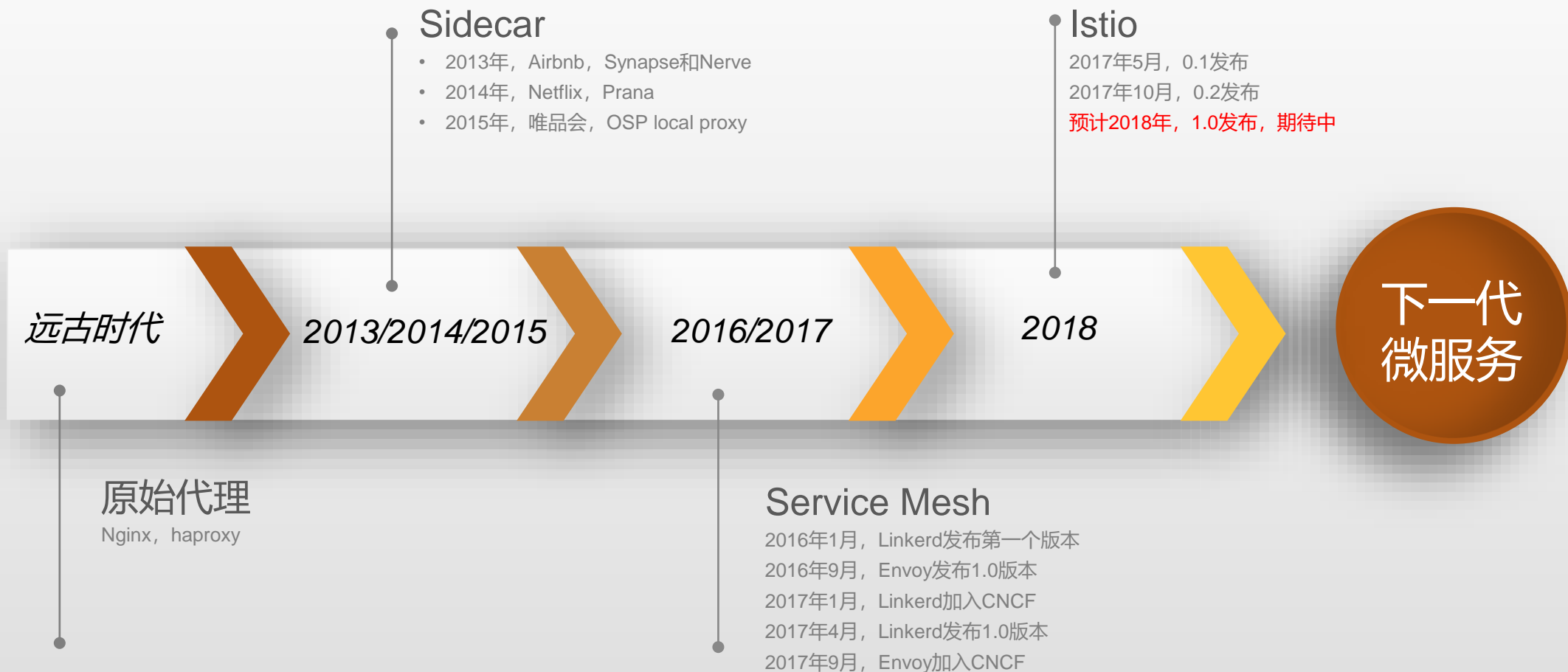


Istio的设计理念新颖前卫，极富创意
开发团队实力惊人
功能齐全
有望成为下一个k8s

超凡实力



Service Mesh演进总结





3

Why

为何选择Service Mesh?

在走过Service Mesh演进的流程之后，答案很清晰

技术栈下移带来的优势



美中不足1: 类库内容有点多, 框架门槛还是稍高: 都交给Service Mesh了, 应用只需关注业务逻辑

美中不足2: 服务治理相关功能不够齐全: Service Mesh把功能做齐全就好

美中不足4: 类库要升级怎么办? 可单独升级Service Mesh, 应用程序不用改

Service Mesh带来的奇迹：鱼与熊掌兼得

对业务团队的技术要求降低
尤其对非互联网做技术转型的业务开发团队
开发测试成本降低

门槛降低

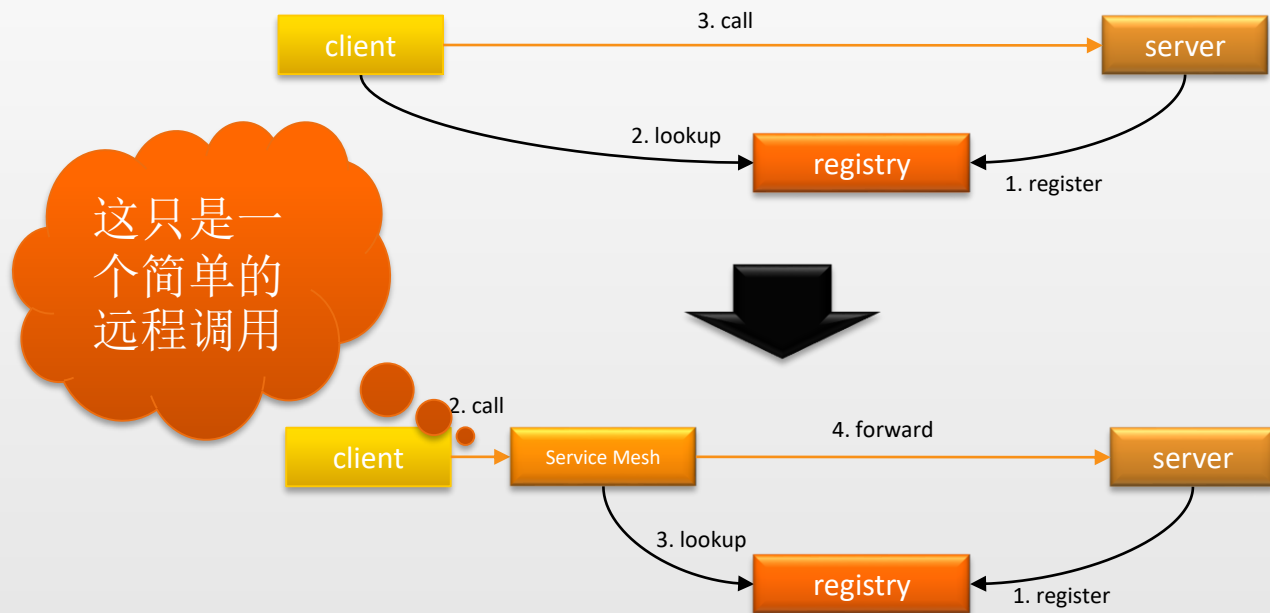
可以更多的关注微服务的其他领域
如微服务的拆分/API设计/数据一致性

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

功能增加

无需修改应用代码
灵活配置，按需定制

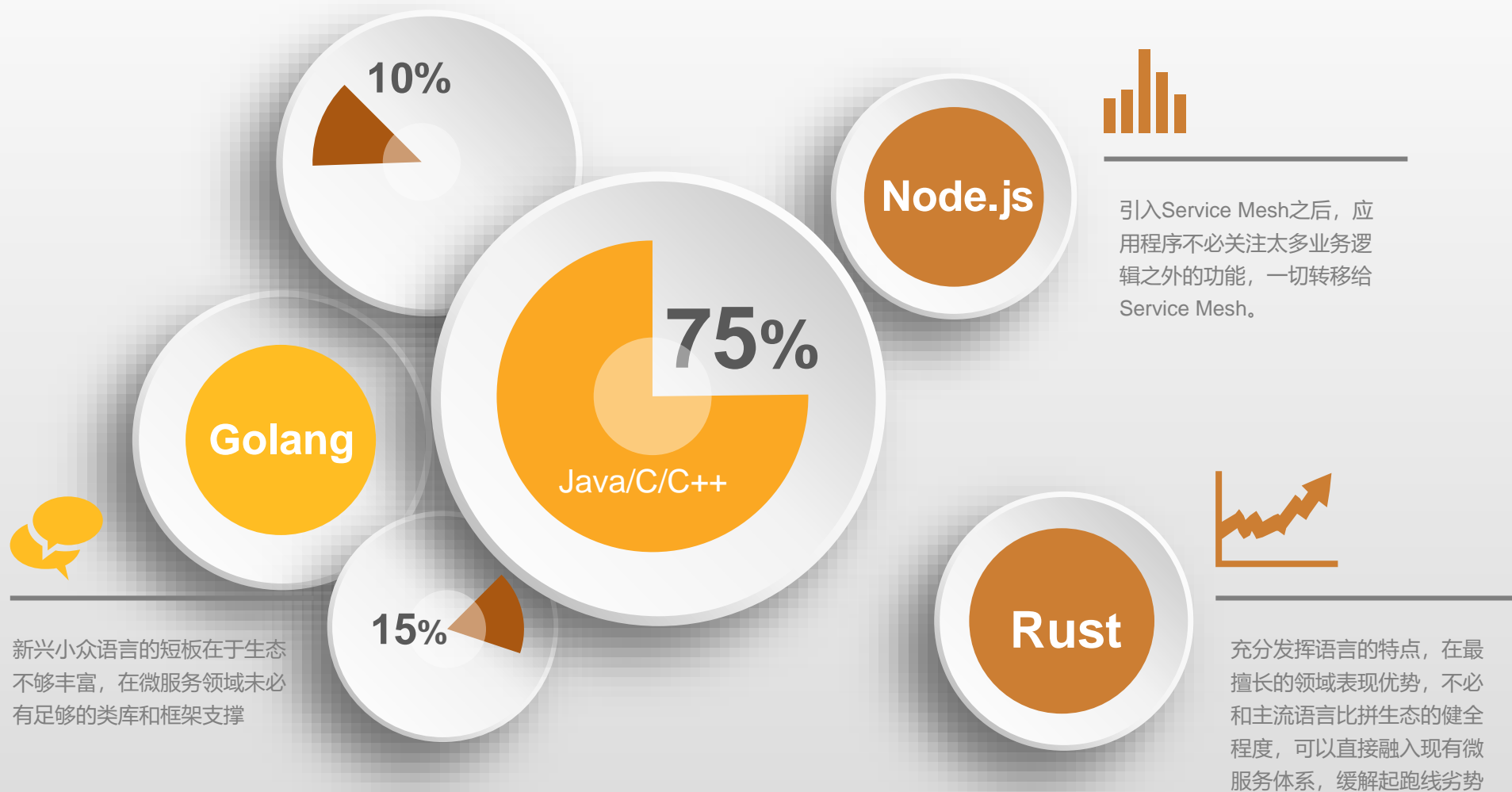
Service Mesh介入调用流程后带来的优势



美中不足3: 说好的跨语言呢? 可以自由选择, 客户端完全不影响, 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

对新兴小众语言是个极大利好



新兴小众语言的短板在于生态不够丰富，在微服务领域未必有足够的类库和框架支撑

引入Service Mesh之后，应用程序不必关注太多业务逻辑之外的功能，一切转移给Service Mesh。

充分发挥语言的特点，在最擅长的领域表现优势，不必和主流语言比拼生态的健全程度，可以直接融入现有微服务体系，缓解起跑线劣势

- [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

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

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

- [ServiceMesh中文网 \(servicemesh.cn\)](#)

ServiceMesh中文技术交流网站。

Service Mesh微信交流群, 请联系微信ID xiaoshu062 申请加入。

谢谢!