长路漫漫踏歌而行：蚂蚁金服Service Mesh实践探索

长路漫漫踏歌而行

蚂蚁金服Service Mesh实践探索

敖小剑

大家好，我是来自蚂蚁金服中间件团队的敖小剑，目前是蚂蚁金服 Service Mesh 项目的PD。我同时也是 Servicemesher中国技术社区 的创始人，是 Service Mesh 技术在国内最早的布道师。我今天给大家带来的主题是"长路漫漫踏歌而行：蚂蚁金服Service Mesh实践探索"。

前言

在去年的QCon上海大会上，我做了一个“Servicemesh：下一代微服务”的主题演讲，布道Servicemesh技术。

今天，有幸再次来到QCon，再次给大家带来Servicemesh的内容，和上次布道不同，我给大家带来的是过去一年中Service Mesh领域的实践与探索，以及我们蚂蚁金服的Service Mesh开源项目——SOFAMesh。

在去年的QCon上海，我曾经做过一个名为 "Service Mesh：下一代微服务" 的演讲，不知道今天现场是否有当时听过去年这场演讲的同学？（备注：现场调查的结果，大概十几位听众听过去年的演讲。）

当然，今天我们的内容不是继续做 Service Mesh 的布道，按秀涛的要求，今年要好好讲一讲实践。所以今天我不会像去年那样给大家详细解释 Service Mesh 是什么，能做什么，有什么优势。而是结合过去一年中蚂蚁金服的实践经验，结合蚂蚁金服的 SOFAMesh 产品，帮助大家更深刻的理解 Service Mesh 技术。

在开始今天的内容分享之前，我们先来热个身，温习一下去年的内容。去年我是来QCon布道的，而布道的核心内容就是告诉大家：Service Mesh 是什么？

为了帮忙大家回答，我给出一个提示图片，了解 Service Mesh 的同学对这张图片应该不会陌生。

这里我们一起回顾一下Service Mesh 的正式定义：

> Service Mesh是一个**基础设施层**，用于处理服务间通讯。现代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**。

黑色加粗部分是重点：

- 基础设施层：这是 Service Mesh 的定位，今天内容的最后一个部分我会和大家详细展开这个话题
- 服务间通讯：这是 Service Mesh 的功能和范围
- 实现请求的可靠传递：是 Service Mesh 的目标
- 轻量级网络代理：是 Service Mesh 的部署方式
- 对应用程序透明：是 Service Mesh 的重要特性，零侵入，Service Mesh 的最大优势之一。

今天的内容会有这些：

- 先给大家快速介绍一下我们的 SOFAMesh 项目，让大家对故事的背景有个大致的了解
- 然后给大家介绍一下为什么我们选择了用 Golang 语言来实现数据平面，这个是过去一年中各方对我们产品方案最大的疑惑
- 再继续给大家分享一下过去一年中我们在 Service Mesh 落地中遇到的典型问题和解决方案，给大家一些比较实际感受
- 然后我们将探索一下服务间通讯的范围，看看 Service Mesh 可以在哪些领域得到应用
- 再接下来，给大家介绍一下在这一年实践中的最大感悟，和大家聊聊基础设施对服务网格的意义，这也是今年最想和大家分享的内容。
- 最后，总结一下今天的内容，分享一些信息

OK，让我们开始今天的第一个部分，给大家快速介绍一下 SOFAMesh，目标在展开我们的各种实践和探索之前，让大家了解一下背景。

SOFAMesh：Istio的落地增强版

蚂蚁金服 ANT FINANCIAL

- ✓ 原则1：跟随社区
  - SOFAMesh fork自Istio
  - 紧跟Istio最新版本
  - 开源，并反哺上游
- ✓ 原则2：实践检验
  - 在实际生产落地中，发现问题，解决问题
  - 在解决问题的实践中，追求创新
  - 扩展Istio，弥补不足和缺失

SOFAMesh 是蚂蚁金服推出的 Service Mesh 开源产品，大家可以简单的理解为是 Istio 的落地增强版本。我们有两个原则：

1. 跟随社区

   体现在 SOFAMesh 是 fork 自 Istio，而且紧跟 Istio 的最新版本，确保和上游保持同步。

   我们在 Istio 上的改动都在 SOFAMesh 项目中开源出来，而且在验证完成后我们也会联系 Istio，反哺回上游。

2. 实践检验

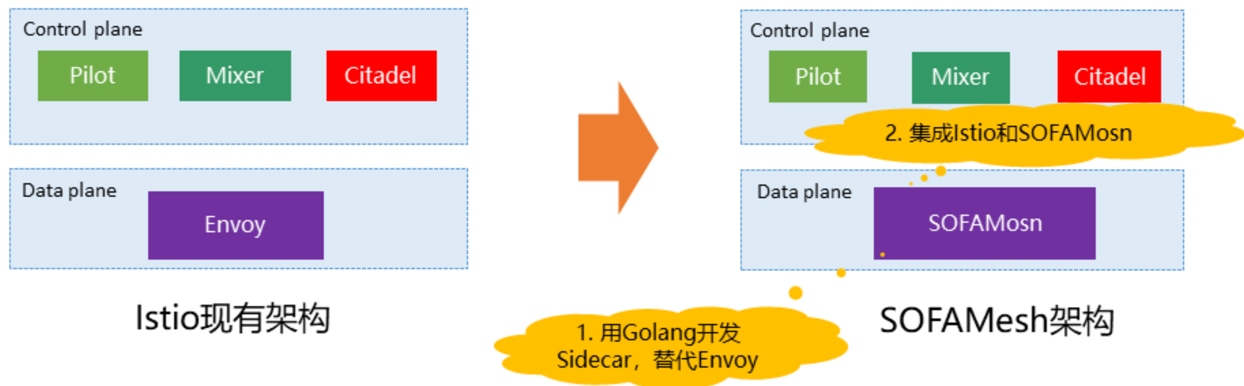
   一切从实践出发，不空谈，在实际生产落地中，发现问题，解决问题。在解决问题的过程中，不将就，不凑合，努力挖掘问题本质，然后追求以技术创新的方式来解决问题。

原则上：Istio 做好的地方，我们简单遵循，保持一致；Istio 做的不好或者疏漏的地方，我们努力改进和弥补。

所有这一切，以**实际落地**为出发点，同时满足未来的技术大方向。

SOFAMesh产品规划（第一阶段）

蚂蚁金服 ANT FINANCIAL



SOFAMesh 的产品规划，这是目前正在进行的第一阶段。架构继续延续 Istio 的数据平面和控制平面分离的方式，主要工作内容是：

1. 用 Golang 开发 Sidecar，也就是我们的 SOFAMosn 项目，替代 Envoy。
2. 集成 Istio 和 SOFAMosn，同时针对落地时的需求和问题进行扩展和补充，这是我们的 SOFAMesh 项目

在这个架构中，和 Istio 原版最大的不同在于我们没有选择 Istio 默认集成的 Envoy，而是自己用 Golang 开发了一个名为 SOFAMosn 的 Sidecar 来替代 Envoy。

为什么？

2

1 SOFAMesh快速介绍
2 为什么选择Golang?
3 落地中遇到的典型问题
4 服务间通讯范围的探索
5 基础设施对服务网格的意义
6 总结

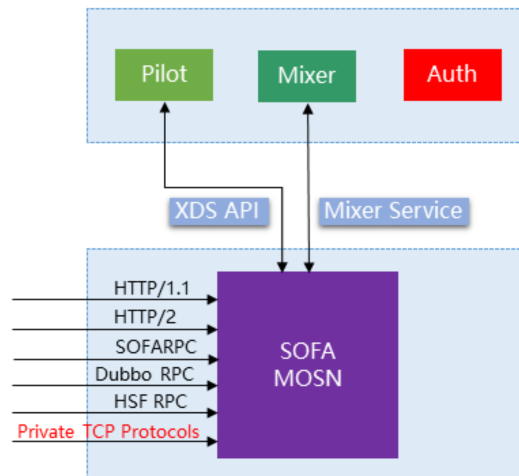
蚂蚁金服 ANT FINANCIAL

我们的第二部分内容将给大家解答这个问题。

Modular Observable Smart Network

蚂蚁金服 ANT FINANCIAL

- ✓ Sidecar模式参照Envoy的定位
- ✓ 实现XDS API
- ✓ 兼容Istio
- ✓ 支持HTTP/1.1和HTTP/2
- ✓ 扩展SOFARPC/Dubbo/HSF等RPC协议支持
- ✓ 私有TCP协议支持



MOSN 的全称是 "Modular Observable Smart Network"，正如其名所示，这是一个模块化可观察的智能网络。这个项目有非常宏大的蓝图，由蚂蚁金服的系统部门和中间件部门联手UC大文娱基础架构部门推出，准备将原有的网络和中间件方面的各种能力在 Golang 上重新沉淀，打造成为未来新一代架构的底层平台，承载各种服务通讯的职责。

Sidecar 模式是 MOSN 目前的主要形式之一，参照 Envoy 项目的定位。我们实现了 Envoy 的 xDS API，和 Istio 保持兼容。

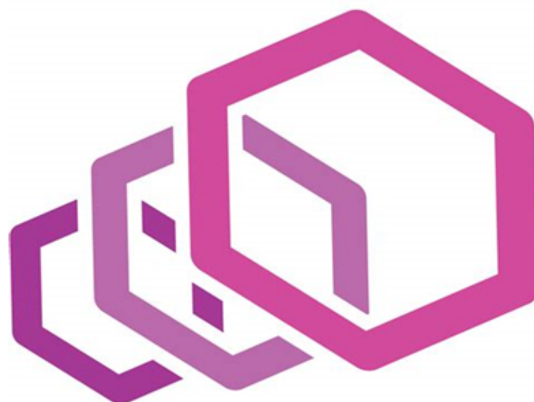
在 Istio 和 Envoy 中，对通讯协议的支持，主要体现在 HTTP/1.1 和 HTTP/2 上，这两个是 Istio / Envoy 中的一等公民。而基于 HTTP/1.1 的 REST 和基于 HTTP/2 的 gRPC，一个是目前社区最主流的通讯协议，一个是未来的主流，Google 的宠儿，CNCF 御用的 RPC 方案，这两个组成了目前 Istio 和 Envoy（乃至 CNCF 所有项目）的黄金组合。

而我们 SOFAMesh，在第一时间就遇到和 Istio/Envoy 不同的情况，我们需要支持 REST 和 gRPC 之外的众多协议：

- SOFARPC：这是蚂蚁金服大量使用的 RPC 协议(已开源)
- HSF RPC：这是阿里集团内部大量使用的 RPC 协议(未开源)
- Dubbo RPC: 这是社区广泛使用的 RPC 协议(已开源)
- 其他私有协议：在过去几个月间，我们收到需求，期望在 SOFAMesh 上运行其他 TCP 协议，大部分是私有协议

为此，我们需要考虑在 SOFAMesh 和 SOFAMosn 中增加这些通讯协议的支持，尤其是要可以让我们的客户非常方便的扩展支持各种私有TCP协议。

Why not Envoy?



Envoy

- 成熟稳定

为什么不直接使用 Envoy？

几乎所有了解 SOFAMesh 产品的同学，都会问到这个问题，也是 SOFAMesh 被质疑和非议最多的地方。因为目前 Envoy 的表现的确是性能优越，功能丰富，成熟稳定。

我们在技术选型时也是重点研究过 Envoy，可以说 Envoy 非常符合我们的需求，除了一个地方：**Envoy是c++**。

数据平面应该选择什么编程语言?

Linkerd

- Scala

Envoy

- C++

Conduit/Linkerd2

- Rust

nginmesh

- C(nginx) + Golang

CES Mesher

- Golang

Motan Mesh

- Golang

OSP Local Proxy

- Java

这里有个选择的问题，就是数据平面应该选择什么样的编程语言？

图中列出了目前市场上主要的几个 Service Mesh 类产品在数据平面上的编程语言选择。

- 首先，基于 Java 和 Scala 的第一时间排除，实践证明，JDK/JVM/字节码这些方式在部署和运行时都显得太重，不适合作为 Sidecar

- Nginmesh 的做法有些独特，通过 Golang 的 agent 得到信息然后生成配置文件塞给 nginx，实在不是一个正统的做法
- Conduit（后更名为Linkerd2.0）选择的 Rust 是个剑走偏锋的路子，Rust 本身极其适合做数据平面，但是 Rust 语言的普及程度和社区大小是个极大的欠缺，选择 Rust 意味着基本上无法从社区借力
- Envoy 选择的 c++
- 国内华为和新浪微博选择了 Golang

我们在选择之前，内部做过深入讨论，焦点在于：未来的新一代架构的底层平台，编程语言栈应该是什么？最终一致觉得应该是 Golang，配合部分 Java。

对于 Sidecar 这样一个典型场景：

- 要求高性能，低资源消耗，有大量的并发和网络编程
- 要能被团队快速掌握，尤其新人可以快速上手
- 要和底层的 k8s 等基础设施频繁交互，未来有 Cloud Native 的大背景
- 非常重要的：要能被社区和未来的潜在客户接受和快速掌握，不至于在语言层面上有过高的门槛

不考虑其他因素，满足 Sidecar 场景的最理想的编程语言，自然是非 Golang 莫属。

艰难的决定：先难后易，着眼未来

直接使用Envoy

- 优势：成熟项目，表现稳定
- 优势：Istio的默认Sidecar
- 优势：速度快，资源消耗低

- 劣势：C++带来的开发和维护成本
- 劣势：扩展协议和功能非常麻烦
- 劣势：没有可控性，创新动力不足

开发自己的SOFAMosn

- 劣势：新项目，工作量大，技术有挑战
- 劣势：需要自行完成和Istio的集成
- 劣势：要对齐Envoy需要非常大的努力

- 优势：Golang更适合云原生时代
- 优势：扩展协议和功能非常方便
- 优势：可控性好，可以快速创新和试错

但是到具体的技术选型时，面对要不要使用 Envoy，决策依然是非常艰难：关键在于，c++有 Envoy 这样成熟的产品存在，可以直接拿来用；而 Golang 没有可以和 Envoy 分庭抗礼的产品可以选择，需要白手起家。

两个选择各有优劣，短期看：

- 直接使用 Envoy，优势在于这是一个成熟项目，表现稳定，而且也是 Isto 默认的 Sidecar，本身速度快，资源消耗低。可以直接拿来用，上手超简单，投入少而收益快
- 开发自己的 Golang 版本的 Sidecar，全是劣势：这是一个全新项目，工作量非常大，而且技术上很有挑战，还有需要自行完成和 Istio 集成的额外工作量以及维护成本，最大的挑战还在于 Envoy 丰富甚至繁多的功能，要向 Envoy 对齐需要非常大的努力

可以说，短期内看，选择 Envoy 远比自行开发 Golang 版本要现实而明智。

但是，前面我们有说到，对于 MOSN 项目，我们有非常宏大的蓝图：准备将原有的网络和中间件方面的各种能力重新沉淀和打磨，打造成为未来新一代架构的底层平台，承载各种服务通讯的职责。这是一个需要一两年时间打造，满足未来三五年乃至十年需求的长期规划项目，我们如果选择以 Envoy 为基础，短期内自然一切OK，快速获得各种红利，迅速站稳脚跟。

但是：后果是什么？Envoy 是C++的，选择 Envoy 意味着我们后面沉淀和打磨的未来通讯层核心是c++的，我们的语言栈将不得不为此修改为以c++为主，这将严重偏离既定的 Golang + Java 的语言栈规划。

而一旦将时间放到三五年乃至十年八年这个长度时，选择Envoy的劣势就出来了：

- C++ 带来的开发和维护成本时远超 Golang，时间越长，改动越多，参与人数越多，使用场景越多，差别越明显
- 从目前的需求上看，对 Envoy 的扩展会非常多，包括通讯协议和功能。考虑到未来控制平面上可能出现的各种创新，必然需要数据平面做配合，改动会长期存在
- Golang 还是更适合云原生时代，选择 Golang，除了做 Sidecar，锻炼出来的团队还可以用 Golang 去完成其他各种产品。当然选择 Envoy 也可以这么干，但是这样一来以后系统中就真的都是c++的产品了。
- 另外 Envoy 目前的官方定位只有 Sidecar 一种模式，而我们规划中的 MSON 项目覆盖了各种服务通讯的场景
- 日后如何和 Envoy 协调是个大难题。尤其我们后续会有非常多的创新想法，也会容许快速试错以鼓励创新，选择 Envoy 在这方面会有很多限制。

所以，最后我们的选择是：先难后易，着眼未来。忍痛（真的很痛）舍弃 Envoy，选择用 Golang 努力打造我们的 SOFAMosn 项目。

对于同样面临要不要选择 Envoy 的同学，我给出的建议是：Envoy 是否适合，取决于是不是想“动”它。

- 如果只是简单的使用，或者少量的扩展，那么其实你接触到的只是 Envoy 在冰山上的这一小部分，这种情况下建议你直接选择 Envoy
- 如果你和我们一样，将 Service Mesh 作为未来架构的核心，预期会有大量的改动和扩展，同时你又不愿意让自己的主流编程语言技术栈中 c++ 占据主流，那么可以参考我们的选择

当然，对于原本就是以 c/c++ 为主要编程语言栈的同学来说，不存在这个问题。

今天的第三部分，给大家介绍一下 SOFAMesh 在落地期间遇到的典型问题。

通讯协议扩展

- 支持多种多样的TCP私有协议

平滑迁移传统架构

- 让现有架构也能从Service Mesh中提前受益

适配异构体系

- Service Mesh体系和传统体系相互打通
- POC中

这里给大家列出了三个主要问题：

1. 通讯协议扩展

   前面也刚谈到过，就是我们会需要支持非常多的TCP协议，包括各种私有协议。当然这个其实更应该归为需求，后面详细展开。

2. 平滑迁移传统架构

   所谓传统架构指的是传统的 SOA 架构，如基于 Dubbo 的很多现有应用，我们希望它们能够在 Service Mesh 中直接跑起来，而不必一定要先进行微服务改造。

3. 适配异构体系

   异构体系指的是，当我们进行 Service Mesh 落地时，会存在新旧两条体系，比如说新的应用是基于 Service Mesh 开发的，而旧的应用是基于传统框架比如说 Dubbo 或者是 Spring Cloud。

   当我们做应用迁移的时候，考虑到原来的存量应用会有很多的，比如上千个应用，这些应用肯定不可能说一个晚上全部切过去。中间必然会有一个过渡阶段，在这个过渡阶段新旧体系中的应用应该怎么通讯，如何才能做到最理想。

   我们现在正在做方案，就是现在POC中。我们现在给自己设定的目标，就是希望给出一套方案，可以让现有应用不做代码改动，然后可以在新旧两边随便切，以保证平滑迁移。

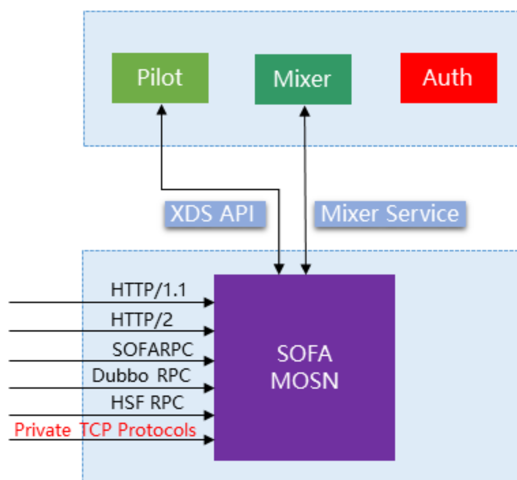
   当然这套方案现在正在做POC，方案还未最终定型，所以今天我们不会包含这一块的细节。大家如果有兴趣的话可以稍后关注我们的这个方案。

今天给大家主要分享前面两个部分，我们详细展开。

X-protocol通用解决方案：快速支持新协议

✓ 添加新通讯协议支持有大量重复工作

- 增加协议的Encoder和Decoder
- 修改Pilot下发Virtual Host等配置
- 修改Mosn实现请求匹配



增加一个新协议：只需一两百行代码，几个小时完成！

第一个要解决的问题是如何快速的扩展支持一个新的通讯协议。

这个问题主要源于现在 Istio 的设计，按照 Istio 现在的方式，如果要添加一个新的通讯协议，是有几大块工作要做的：

- 添加协议的 Encoder 和 Decoder

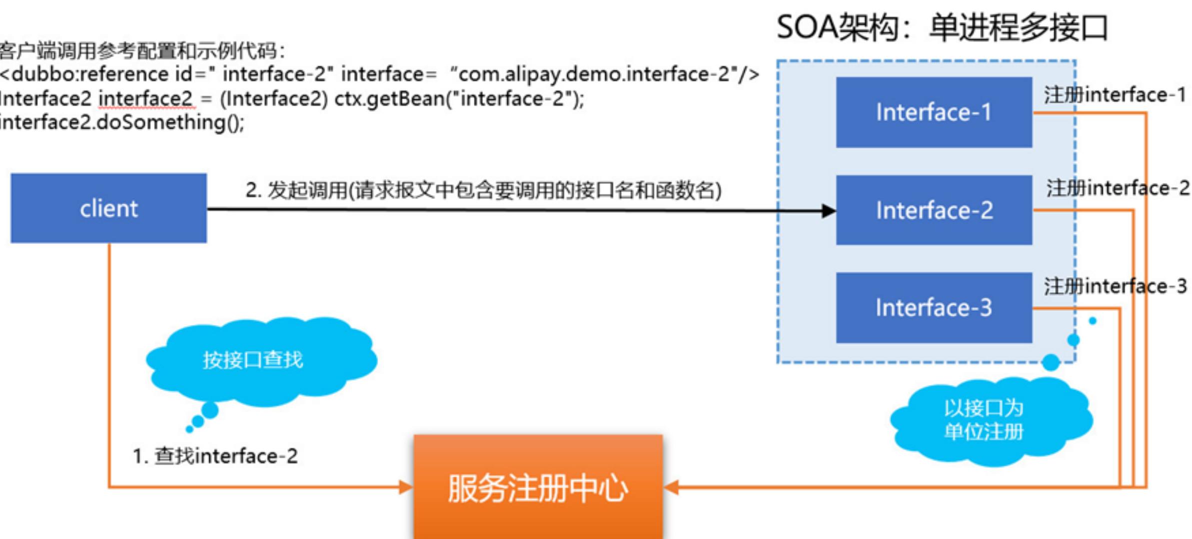
  也就是协议的编解码，这个没得说，肯定要加的。

- 修改 Pilot 下发 Virtual Host 等配置
- 修改 Sidecar 如 Envoy，MOSN 去实现请求匹配

后两者是大量重复的，就技术实现而言，需要修改的内容和现有的东西差不多，但是必须要再改出一份新的来。因为我们协议比较多，由此带来的改动量非常大。根据我们之前的实践，以这样的方式加一个新的通讯协议可能需要几天的工作量，而且每次改动都重复大量代码。

在这里我们最后给出了一个名为 **x-protocol** 的通用解决方案，细节我们这里不展开，只给大家看个结果。根据我们最新的验证情况，如果我们要添加一个新的通讯协议，大概就是一两百行代码，一两个小时就能完成。即使加上测试，基本上也可以控制在一天之内，我们就能够为 SOFOMesh 新增一个通讯协议的支持。

传统架构的问题：SOA模型和微服务模型不匹配



第二个要解决的问题就是让传统架构的存量应用上 Service Mesh 的问题。

就是刚才说的现有大量的基于 SOA 框架的程序，这些应用以传统的 SOA 方式开发，如果直接挪到 Service Mesh 下，如Istio，会遇到问题：因为 Istio 用的服务注册是通过 k8s 来进行，而 k8s 的服务注册模型和原有的 SOA 模型是不匹配的。

SOA 框架当中，通常是以接口为单位来做服务注册，也就是一个应用里面部署多个接口的，在运行时是一个进程里面有多个接口（或者说多个服务）。实际上是以接口为粒度，服务注册和服务发现，包括服务的调用都是以接口为粒度。但是有个问题，部署到 Istio 中后，Istio 做服务注册是以服务为粒度来做服务注册，这个时候不管是注册模型，还是按接口调用的方式都不一致，就是说通过 Interface 调用是调不通的。

左边的代码实例，大家可以看得到，一般情况下 Dubbo 程序是按照 Interface 来注册和发现，调用时也是通过 Interface 来调用。另外，在这个地方，除了通过接口调用之外，还有另外一个问题：服务注册和服务发现的模型，从原来的一对N，也就是一个进程N个接口，变成了要一对一，一个进程一个服务。

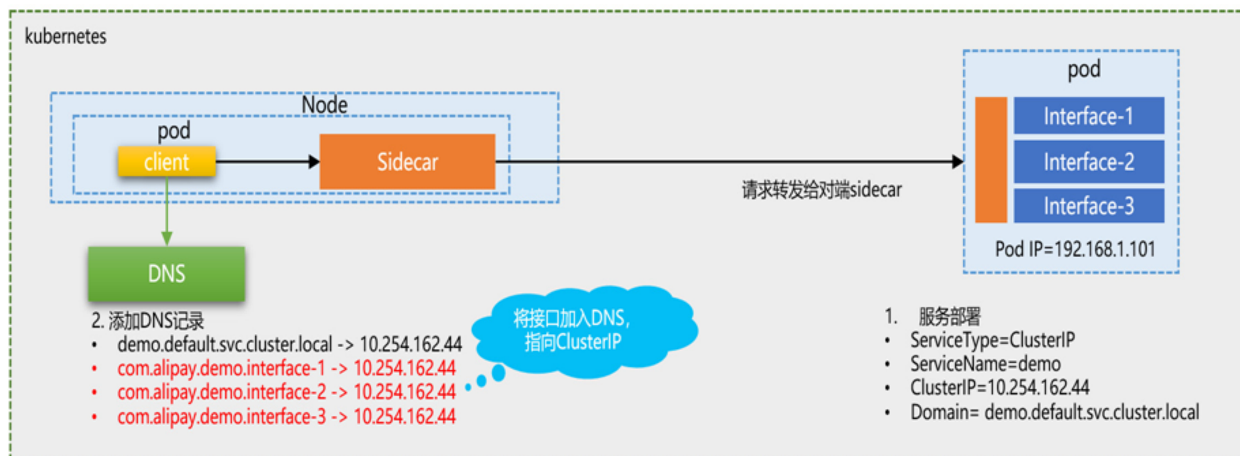
怎么解决这个问题？最正统的做法是，是先进行**微服务改造**：把原有的 SOA 的架构改成微服务的架构，把现有应用拆分为多个微服务应用，每个应用里面一个服务（或者说接口），这样应用和服务的关系就会变成一对一，服务注册模型就可以匹配。

但是在执行时会有难处，因为微服务改造是一个比较耗时间的过程。我们遇到的实际的需求是：能不能先不做微服务改造，而先上 Service Mesh ？因为 Service Mesh 的功能非常有吸引力，如流量控制，安全加密。那能不能先把应用搬迁到 Service Mesh 上来，先让应用跑起来，后面再慢慢的来做微服务改造。

这就是我们实际遇到的场景，我们需要找到方案来解决问题：注册模型不匹配，原有用接口调用的代码调不通。

DNS通用寻址方案：支持Dubbo等SOA框架以接口名调用服务

蚂蚁金服 ANT FINANCIAL



先上车后补票：提前受益于Service Mesh的强大功能

我们设计了一个名为 **DNS通用选址方案** 的解决方案，用来支持 Dubbo 等SOA框架，容许通过接口名来调用服务。

细节不太适合展开，给大家介绍最基本的一点，就是说我们会在 DNS 中增加记录，如图上左下角所示标红的三个接口名，我们会在DNS中把这个三个接口指向当前服务的 Cluster IP。k8s 的 Cluster IP 通常是一个非常固定的一个IP，每个服务在k8s部署时都会分配。

在增加完DNS记录之后，再通过 Interface 的方式去调用，中间在我们的 Service Mesh 里面，我们会基于 Cluster IP 信息完成实际的寻址，并跑通 Istio 的所有功能，和用服务名调用等同。

这个功能在现有的 SOFAMesh 中已经完全实现，大家可以去试用。稍后我们会将这个方案提交给 k8s 或者 Istio 社区，看看他们是否愿意接受这样一个更通用的寻址方式。

在这里我们提出这样一个设想：**先上车后补票**。所谓"先上车"是指说先上 Service Mesh 的车，"后补票"是指后面再去补微服务拆分的票。好处是在微服务拆分这个巨大的工作量完成之前，提前受益于 Service Mesh 提供的强大功能；同时也可以让部署变得舒服，因为不需要强制先全部完成微服务拆分才能上 Service Mesh 。有了这个方案，就可以在应用不做微服务拆分的情况下运行在 Service Mesh 上，然后再从容的继续进行微服务拆分的工作，这是我们提出这个解决方案的最大初衷。

MOSN和x-protocol的更多资料

蚂蚁金服 ANT FINANCIAL

- ✓ MOSN和x-protocol介绍
  - Service Mesh数据平面SOFAMosn深层揭秘
  - 蚂蚁金服开源Go语言版Service Mesh数据平面SOFAMosn性能报告
  - 蚂蚁金服开源的 SOFAMesh 的通用协议扩展解析
  - Dubbo on x-protocol——SOFAMesh中的x-protocol示例演示
- ✓ X-protocol特性的详细讲解
  - SOFAMesh中的多协议通用解决方案x-protocol介绍系列(1)-DNS通用寻址方案
  - SOFAMesh中的多协议通用解决方案x-protocol介绍系列(2)-快速解码转发
  - SOFAMesh中的多协议通用解决方案x-protocol介绍系列(3)-TCP协议扩展

当然，这里面有比较多的技术实现细节，里面有很多细节的东西实在是不适合在这里一一展开。同时也涉及到比较多的 k8s 和 Istio 底层技术细节，需要大家对 k8s kubeproxy 网络转发方案和 Istio 的实现有比较深的认知才能完全理解。这里给出了几篇文章，大家如果对这几个技术有兴趣，可以通过这些文章来了解里面的技术细节，今天就不在这里继续展开了。

MOSN 和 x-protocol 介绍：

- Service Mesh数据平面SOFAMosn深层揭秘
- 蚂蚁金服开源Go语言版Service Mesh数据平面SOFAMosn性能报告
- 蚂蚁金服开源的 SOFAMesh 的通用协议扩展解析
- Dubbo on x-protocol——SOFAMesh中的x-protocol示例演示

X-protocol 特性的详细讲解：

- SOFAMesh中的多协议通用解决方案x-protocol介绍系列(1)-DNS通用寻址方案
- SOFAMesh中的多协议通用解决方案x-protocol介绍系列(2)-快速解码转发
- SOFAMesh中的多协议通用解决方案x-protocol介绍系列(3)-TCP协议扩展

总结一下，我们解决了如下几个问题：

1. 可以快速的用几个小时就在 SOFAMesh 中添加一个新的通讯协议
2. 可以让 SOA 应用在 SOFAMesh 上继续通过接口进行调用，不需要改代码
3. 可以实现不做 SOA 程序的微服务改造，就直接搬迁到 SOFAMesh，提前受益

流量劫持方案的考虑和优化

蚂蚁金服 ANT FINANCIAL

- 实践：优化iptables
  - 减少对Host主机的影响
  - iptables pod only & 最小化
- 调研： IPVS方案
  - Iptables被限制使用的场合
  - IPVS pod only做NAT,已初步验证方案可行
- 实践：轻量级RPC客户端
  - 通过环境变量给出Sidecar地址
  - SDK直接发送请求到Sidecar，不做流量劫持
- 密切关注：Cilium + eBPF的思路

第四块，涉及到流量劫持的方案。

Service Mesh 有一个很重要的特性，就是无侵入，而无侵入通常是通过流量劫持来实现的。通过劫持流量，在客户端服务器端无感知的情况下，可以将 Service Mesh 的功能插进去。通常特别适合于类似安全加密等和现有应用的业务逻辑完全分离的场合。

但 Istio 的流量劫持方案做的还不够好，目前 Istio 只给了一个方案就是 iptables。这个方案有比较多的问题，所以我们有几个思路：

1. 优化 iptables

   优化 iptables 主要是为了减少对Host主机的影响。

   用 iptables 有两种思路：一个是 pod only，就是说在pod里面做 iptables，这个Istio的官方做法，但是这样需要ROOT权限以便修改iptables配置；还有一种思路用Host主机上的 iptables，这个话可以不用ROOT权限。我们对比之后，还是觉得放在pod里面更好一点，因为性能损耗比较小一些，所以暂时我们用的是在pod中方案，但我们会进行优化，比如把 iptables 的模块简化到最小。

2. 调研IPVS方案

   我们现在正在调研IPVS方案。主要是 iptables 方案存在部署问题，就是 iptables 这个模块经常被限制使用。现场有没有做运维的同学？你们的机器上开启了 iptables 吗？我能告诉大家的是，到目前为止，蚂蚁金服内部的机器上，iptables不仅仅是禁用，而是整个 iptables 模块都被卸载。原因是性能、安全、维护等大家周知的原因，总之我们蚂蚁金服内部是没有这个模块的。

   为了解决这个问题，我们现在正在调研IPVS的方案，准备用IPVS来替换 iptables。这块工作正在进行，目前方案已经验证，但是还有一些细节没有完善，后面有更多的消息也给大家继续介绍。

3. 轻量级客户端的实践

   另外还有一个实践是考虑不做流量劫持。比如说，最典型的RPC方案，因为RPC通常来说总是会有一个客户端的。在上 Service Mesh 之后，可以将原来的客户端的一些功能如服务发现、负载均衡、限流等精简，形成一个新的轻量级客户端，但此时终究还是有一个客户端在的。

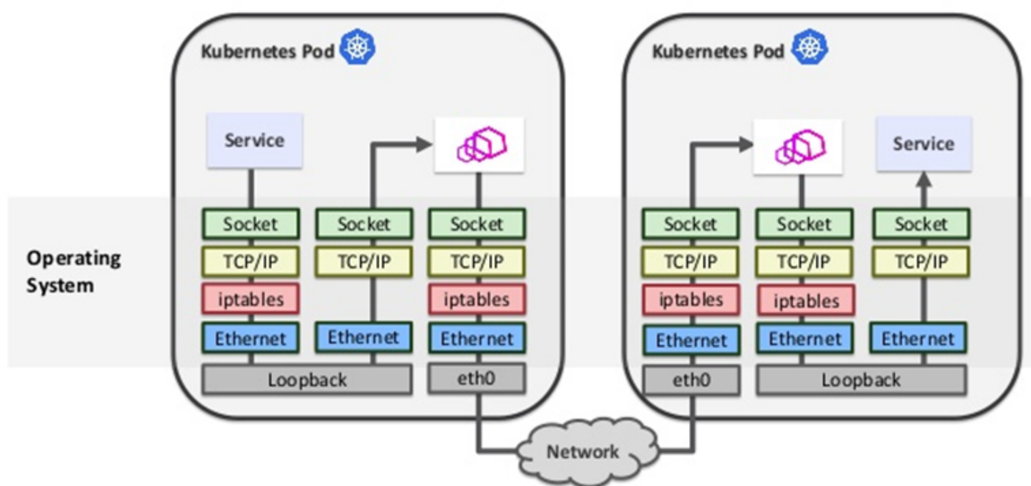
这个时候，如果能知道 Sidecar 的访问地址，是可以不进行流量劫持的，由客户端直接将请求发给 Sidecar 就好了。所以，最基本的想法就是通过环境变量或者配置给出 Sidecar 的地址，告诉客户端 Sidecar 就在 localhost 的8080端口。然后客户端SDK简单读取一下，接下来直接将请求发过去就好了。这个方案可以轻松的绕开流量劫持的问题。

这个方案我们内部也实践过，早期用来替代多语言客户端的版本用的是这个方案。当然，实践中发现流量劫持还是有必要的，这是另外一个话题，后面有机会再详细解释。

但上面这三个都不是今天的重点，今天的重点是下面这个 Cilium + eBPF 的思路，这是我们目前最密切关注的一个方案。

iptables和轻量级客户端：还是要走TCP堆栈

Sidecar Injection (Transparent)



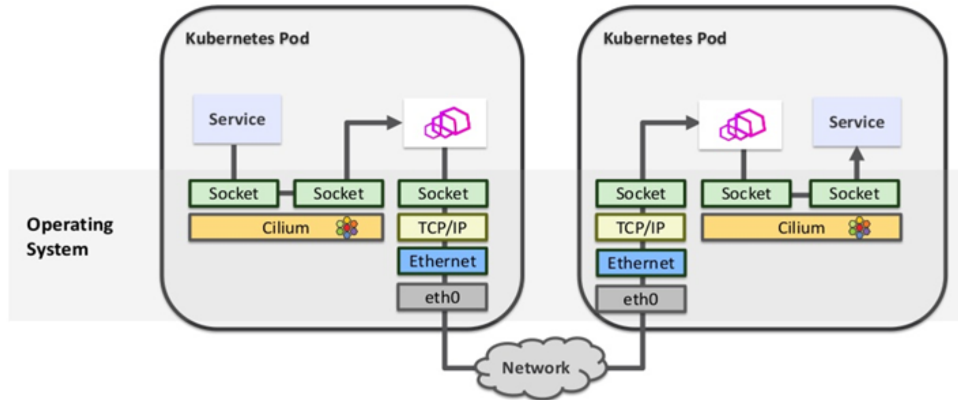
Cilium是一个很新的项目，Cilium的思路中会涉及到底层通讯中TCP堆栈的问题。

这里的图片显示了用 iptables 和轻量级客户端方案的网络调用细节，左边是客户端 Service 和它的 Sidecar 之间的调用过程，可以看到它走了两次的TCP堆栈。然后还有 iptables 的拦截。轻量级客户端方案和流量劫持方案的差别在于减少一次iptables，避开了iptables的性能消耗。但是即使没有 iptables，最终还是要走整个调用流程，虽然 Loopback 环回地址比network 网络通讯快很多，但是它终究还是走了两次TCP堆栈，两次TCP堆栈这里是有性能消耗的。

Cilium的新思路：绕开TCP堆栈

蚂蚁金服 ANT FINANCIAL

Transparent Sidecar Injection with Cilium



Cilium劫持比轻量级客户端不劫持更快！

而Cilium则给出了一个很好的思路：想办法绕开TCP堆栈。

Cilium方案的好处，就在于在 socket 这个层面就完成了请求的转发，通过 sockmap 技术实现 redirect，当然这个技术细节咱们就不在这里展开。今天主要是讲一下这个思路的好处和价值。Cilium 方案最大的好处，是可以绕开两次TCP堆栈，绕开两次TCP堆栈的好处，则会带来一个出乎意外甚至违背常识的结果：Cilium 劫持可以比轻量级客户端不劫持更快！这可能颠覆大家的观念。

我们来体会一下。流量劫持，如 iptables，是要在原来的调用链当中插入一段，增加消耗导致性能下降，对吧？这是流量劫持最容易给人的留下的负面印象，就是流量劫持是有消耗的，所以优化的思路通常都在于减少这个消耗，或者选择不做劫持从而避开这个消耗。那 Cilium 的做法则是给出另外一种解决流量劫持问题的思路：通过绕开两次TCP堆栈和其他底层细节，更快的将请求转发给 Sidecar！

Cilium的这个思路是我们非常赞赏的，通过这样的方式减少服务和 Sidecar 之间的性能损失，可以解决 Service Mesh 中至关重要的一个问题：**性能与架构的取舍**。

熟悉 Service Mesh 技术的同学，应该多少都有这样的认知： Service Mesh 是一门中庸的艺术。在性能与架构之间， Service Mesh 选择牺牲性能来换取架构。在传统的侵入式框架中，客户端业务代码和框架代码之间是通过函数来进行调用的，速度非常快完全可以忽略。而 Service Mesh 是强行把框架和类库剥离出来，将上述的方法调用变成一个远程调用，以牺牲了一次远程调用的开销为代价来换取整个架构的优化空间。这是 Service Mesh 技术和传统侵入式框架的一个本质差异，也是 Service Mesh 技术和传统侵入式框架所有差异的源头。

这是 Service Mesh 技术最为重要的一次取舍：舍弃一次远程调用的开销，换取更富有弹性的架构和更丰富的功能。

Service Mesh 技术的发展，也由此出现两个大方向：一方面是继续在架构上获取好处，更多的功能，更丰富的使用场景，各种创新，竭尽可能的获取红利；另一方面，是在舍字上下功夫，尽可能的将性能损失降到最低，以求得到前面的最大好处而将付出的代价降到最低。

我们在前面列出的这四个实践，都可以说是在这条贪心的路上一步一步的探索和尝试，希望可以将 Service Mesh 架构上的舍弃的部分再尽可能多的要回一部分。

当然，Cilium 在实际落地的时候还是会有一些问题，比如说现在最大的问题是 Cilium 对 Linux 内核的版本要求特别高，最低要求是4.9推荐4.10，然后里面的部分特性要求是4.14。Linux 内核4.14是2017年底才发布的，而目前 Linux 内核最新版本才4.18。Cilium 要求的 Linux 内核的版本实在太新了，部署时很难满足。另外就是 Cilium 还是存在一些安全问题，主要是 eBPF 是将代码直接注入到内核运行，效率好是好，但是肯定会存在安全隐患。

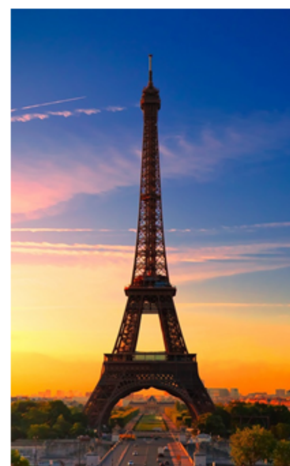
我们接下来会重点跟踪 Cilium 技术，也有可能会给出其它的类似方案，有兴趣的同学可以关注我们的进展。

继续给大家介绍今天的第四部分内容，对服务间通讯范围的探索。

服务间通讯的范围的探讨

✓ Service Mesh可以提供的功能

- 请求转发：服务发现，负载均衡
- 路由能力：
  - Content Based Routing
  - Version based Routing
- 服务治理：灰度，蓝绿，版本管理
- 安全：认证，加密，限流，熔断



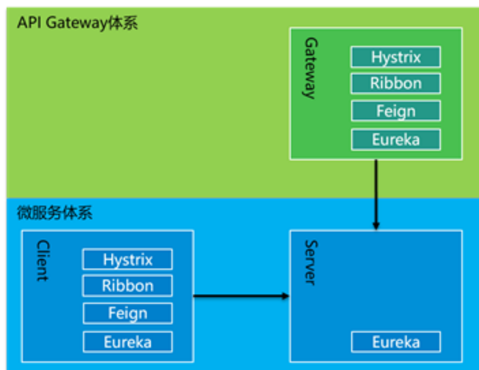
Service Mesh 起初关注的是东西向通讯，即系统内部各个服务之间的通讯，而这通常都是同步的，走REST或者RPC协议。

在Service Mesh的实践过程中，我们发现，Service Mesh 可以提供的功能：

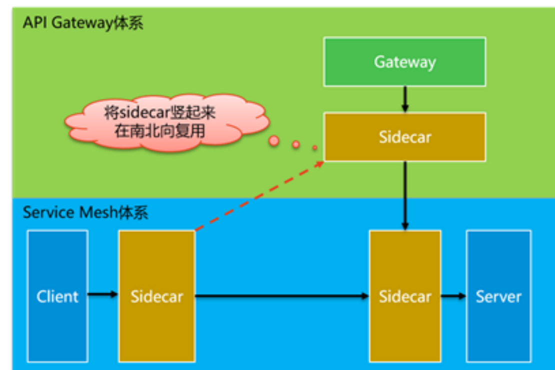
- 请求转发：如服务发现，负载均衡等
- 路由能力：如强大的 Content Based Routing 和 Version Based Routing
- 服务治理：基于路由能力而来的灰度发布，蓝绿部署，版本管理和控制
- 纠错能力：限流，熔断，重试，测试目的的错误注入
- 安全类：身份，认证，授权，鉴权，加密等

可以适用于 Service Mesh 之外的其他领域，也就是说我们可以在其他领域引入并重用这些能力，实现比单纯的东西向通讯更广泛的服务间通讯。

探索一：API Gateway



传统侵入式框架的思路



Service Mesh的思路

第一个探索的方向是 API Gateway，和东西向通讯直接对应的南北向通讯。

主要原因是南北向通讯和东西向通讯在功能上高度重叠，如服务发现，负载均衡，路由，灰度，安全，认证，加密，限流，熔断......因此，重用东西向通讯的这些能力就成为自然而然的想法。

传统侵入式框架下，重用这些能力的方式是基于类库方式，也就是在 API Gateway 的实现中，典型如 Zuul，引入东西向通讯中的类库。而 Service Mesh下，思路有所不同，重用的不再是类库，而是 Sidecar：通过将 Sidecar 用于南北向通讯，重用 Sidecar 的请求转发和服务治理功能。

将 Service Mesh 引入 API Gateway 的优势在于:

- 统一微服务和 API Gateway 两套体系
- 大量节约学习/开发/维护的成本
- 可以在南北向通讯中获得 Service Mesh 的各种特性
- 可以通过 Service Mesh 的控制平面加强对南北向通讯的控制力

这个方向上，业界也有一些探索：

- Ambassador: Kubernetes-native microservices API gateway，基于Envoy构建，开源项目
- Gloo: The Function Gateway built on top of Envoy，同样是基于Envoy，不过这个不仅仅用于传统的微服务 API Gateway，也可以用于Serverless架构的Function

- Kong：在最近宣布，即将发布的1.0版本，kong将不再是单纯的 API Gateway，而是转型为服务控制平台。可谓是一个反向的探索案例：从 API Gateway 向 Service Mesh 切。

而我们的思路也非常明确：在 SOFAMesh 和 SOFAMosn 的基础上，打造新的 API Gateway 产品，以此来统一东西向通讯和南北向通讯。目前该项目已经启动，后续也会作为开源项目公布出来，对这个话题有兴趣的同学可以保持关注。

探索二：Serverless和Knative

Google新推出的Serverless新项目，基于kubernetes和Istio，致力于serverless平台的标准化和规范化

前段时间我们在考虑 Serverless 方向时，刚好看到 Google 新推出了它的 Serverless 新项目 Knative，时间点非常的巧。和其他 Serverless 项目不同的是，Knative 项目关注的是 Serverless 平台的标准化和规范化。

Knative 项目是基于 kubernetes 和 Istio 的，在 Knative 中 Istio 用于实现部分组件之间的通讯。在 Knative 项目中，对于是否应该引入 Istio 存在很大争议，因为觉得 Istio 太重了，为了少量需求引入 Istio 有些兴师动众。不过这个问题对于本来就已经在用 Istio 的我们来说不是问题。

目前在 Serverless，尤其 Knative 方面，我们还在探索，目前的初步想法是这样：

- Serverless 很重要

  尤其 Knative 的出现，昭示着 Serverless 领域新的玩法出现，Serverless 平台出现标准化和统一化的契机

- Kubernetes + Serverless + Service Mesh（尤其是扩展范围之后的 Service Mesh）是一个很好的组合

  从下向上，从底层基础设施到服务间通讯再到 Function，在应用和系统之间形成了一套完整的支撑体系。

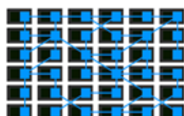
后续我们的产品策略，会继续深入调研 knative，一边POC一边规划产品，当然结合实际业务需要以落地为目标依然是基本要求。然后，非常自然的，我们会将标准版本的 Istio 替换为我们的 SOFAMesh 和 SOFAMosn。

举例，目前我们在计划尝试使用 Serverless 的典型场景：

- 小程序
- AI：Serverless AI Layer，一站式机器学习平台
- Databus： 大数据处理

更完整的服务间通讯

蚂蚁金服 ANT FINANCIAL



Service Mesh

- 东西向通讯
- 实践中：基于Istio的SOFAMesh

API Gateway

- 南北向通讯
- 探索：基于SOFAMosn开发新的API Gateway产品

Serverless

- 异步通讯，事件驱动
- Function 粒度
- 实践中：Knative

预测：云原生时代，服务间通讯的未来

这是我们目前探索和规划中的服务间通讯的完整蓝图：

- Service Mesh

  负责东西向通讯，实践中就是我们的 SOFAMesh 产品，基于 Istio 的扩展增强版

- API Gateway

  负责南北向通讯，还在探索中，我们在尝试基于 SOFAMosn 和 SOFAMesh 开发新的 API Gateway 产品

- Serverless

  负责异步通讯，事件驱动模型，粒度也从服务级别细化到Function级别，目前在积极探索和实践 knative

这里给出一个我们的预测：在云原生的时代，服务间通讯的未来都会是 Service Mesh 这种方式，将服务间通讯的职责剥离并下沉。

这是今天的最后内容。前面四个部分的内容基本上都是给大家介绍我们的产品实践，落地遇到的问题，以及我们正在做的一些探索，比较偏实际。第五部分会特殊一点，可能就有点**务虚**了。这块要讲是在过去一年当中，在项目落地的过程中的特别感受，其中最关键的一点就是基础设施和服务网格之间的关系，或者说基础设施对服务网格的意义。

时代背景：Cloud Native

蚂蚁金服 ANT FINANCIAL

2018年6月，CNCF技术监督委员会投票通过Cloud Native的定义，中文翻译如下：

云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。

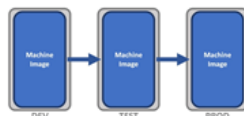
这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。

云原生计算基金会（CNCF）致力于培育和维护一个厂商中立的开源生态系统，来推广云原生技术。我们通过将最前沿的模式民主化，让这些创新为大众所用。

里面有一个时代背景：Cloud Native，云原生。而在今年6月，CNCF 技术监督委员会通过了 Cloud Native 的定义，中文翻译如上。

这里我们将关注点放在标红的这一句来：云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。

蚂蚁金服的策略：积极拥抱云原生架构



Why Declarative

容器

- 实践容器技术多年
- Sigma3.*将基于k8s

微服务

- SOA服务化实践多年
- Dubbo/HSF/SOFA名满江湖
- 正在陆续微服务改造中

不可变基础设施

- 长期实践

声明式API

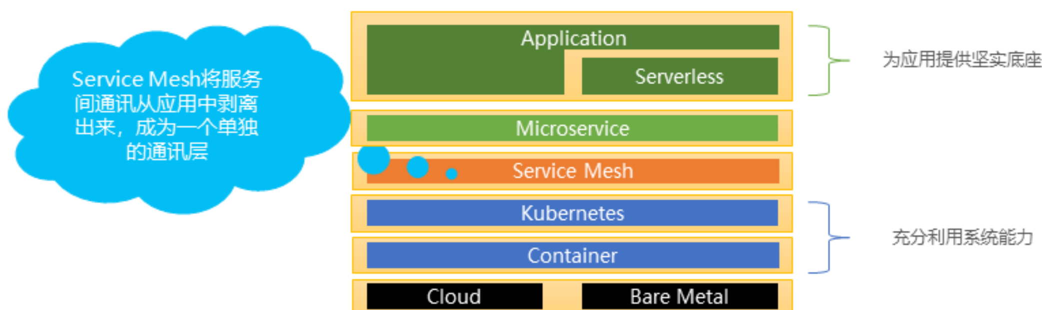
- 长期实践

对于云原生架构，蚂蚁金服的策略是：积极拥抱! 我们未来的架构也会往这个方向演进。

对于前面列举的云原生代表技术：

- 容器：大阿里在容器技术上有非常深度的积累，实践多年，而新版本的 Sigma3.* 版本也将基于 k8s。
- 微服务：微服务的前身，SOA 服务化，在大阿里也是实践多年， Dubbo / HSF / SOFA 可谓名满江湖，目前也在陆陆续续的微服务改造中。
- 不可变基础设施和声明式API：也是高度认可和长期实践的技术。

Service Mesh的定位：承上启下的重要一环

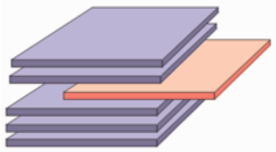


对于Service Mesh的定位，我们是这样理解的：

- Service Mesh 是承上启下的重要一环

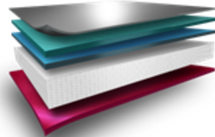
- 一方面充分利用底层系统能力
- 一方面为上层应用提供坚实的底座

Service Mesh的归宿：融入基础设施



从应用剥离

- Service Mesh将服务间通讯从程序中剥离；
- 服务间通讯不再是应用程序的一部分



下沉为抽象层

- 服务间通讯的功能下沉并形成一个抽象层，称为**服务间通讯专用基础设施层**。
- 不再以类库或者框架的形式出现。

融入基础设施

- 和底层基础设施密切联系，融入一体，成为平台系统的一部分；
- 需要协调中间件团队和基础设施团队的关系，密切合作

对于 Service Mesh，我们有一个重要判断，这也是今天最想和大家分享的一点：Service Mesh 的归宿，或者说最终的形态，是下沉到基础设施！

从 Service Mesh 的发展看，从简单的 Proxy，到功能完善的Sidecar（如Linkerd和Envoy），再到以 Istio 为代表的第二代Service Mesh，演进的方式如上图：

1. 第一步：从应用剥离

   通过将原有的方法调用改为远程调用，将类库的功能套上 Proxy 的壳子，Service Mesh 成功的将服务间通讯从程序中剥离出来，从此服务间通讯不再是应用程序的一部分。

   这一点是大家最容易接受的，对吧？这一步也是最容易实现的，只要搭起来一个 Sidecar 或者说 Proxy，将原有类库的功能塞进去就好了。

2. 第二步：下沉为抽象层

   这些剥离出来的服务间通讯的能力，在剥离之后，开始下沉，在应用程序下形成一个单独的抽象层，成为**服务间通讯专用基础设施层**。此时，这些能力以一个完成的形态出现，不再存在单独的类库或者框架形式。

   第二步和第一步往往是一脉相承的，一旦走出了第一步，自然而然会继续。因为服务间通讯被抽取出来之后，继续往前发展，就会很自然地把它就变成一个基础设施层。

3. 第三步：融入基础设施

   继续下沉，和底层基础设施密切联系，进而融为一体，成为平台系统的一部分，典型就是和 kubernetes 结合。

   Istio在这方面做了一个非常大的创新，Istio的创新，不仅仅在于增加控制平面，也在于和 kubernetes 的结合。

如果大家有在去年QCon听过我的演讲，会发现我在去年的时候对 Service Mesh 的理解和现在不太一样。在去年的这个时候，我认为 Istio 最大的创新是增加了控制平面。但是，今年我觉得还再加一个关键点，除了控制平面的增加之外，Istio很重要的一点是开始跟k8s融合，充分利用 k8s 的能力。k8s 代表的是底层基础设施，所有的各种能力在k8s上沉淀。在Istio上，已经能够看到这样一个非常明显的趋势： Service Mesh 已经开始和底层基础设施密切联系，融为一体，成为整个平台系统的一部分。

大家注意体会这中间的细微差异，第一步和第二步，将服务间通讯的能力抽取出来沉淀成一个抽象层，而如果止步于第二步的话，这个抽象层和底层基础设施是没有任何关系的。注意，比如说 Linkerd 或者 Envoy，在部署的时候，不管是物理机、虚拟机或者容器，都没有任何关系，本身也不利用底层的任何能力，可谓泾渭分明。但是一旦演进到了Istio，包括现在的Linkerd 2.0，就会发现转为第三步的这种形态。

今天想跟大家说的是，**Service Mesh 的未来，是将服务间通讯的能力下沉到基础设施**，然后充分利用底层基础设施的能力来架构整个体系。而不再将底层基础设施抽象成为就是一个简单的操作系统抽象：给我cpu，给我内存，给我网络，给我IO，其他的事情和底层没有任何关系，我自己上面全部搞定。这个思路在 Service Mesh 的未来发展中是不合适的， Service Mesh 未来一定是通过和基础设施融合的方式来实现。

注意这个方式跟传统方式的差异，不仅仅在于技术，而是这个方式会混淆两个传统的部门：一个叫做中间件，就像我所在的部门，或者有些公司叫做基础架构部门；还有一个部门通常是运维部门或者叫做系统部门，负责维护底层基础设施。大部分公司一般这两个部门在组织架构上是分离的。做k8s的同学，和做微服务框架比如 Dubbo，Spring Cloud 的同学，通常是两个不同的组织架构，彼此泾渭分明。第三步要走通的话，就会要求中间件部门和基础设施部门关系要协调的特别好，要密切的合作，才能将事情做好。

这是在过去这一年当中，我们在实践中得到的最大的一个感受，也是今天整个演讲中最希望给大家分享的内容。

这里抛出一个问题，和传统的 Spring Cloud，Dubbo等侵入式框架相比：

Service Mesh的本质差异在哪里？

如果去年的我来回答这个问题，那我会告诉你：下移，沉淀，形成一个通讯层。而今天，我会告诉大家，除了这点之外，还有第二点：充分利用底层基础设施。这是Dubbo，Spring Cloud从来没有做到的！

这是今天最想和大家分享的观点，也是过去一年实践中最大的感悟：

Service Mesh 和 Spring Cloud / Dubbo 的本质差异，不仅仅在于将服务间通讯从应用程序中剥离出来，更在于一路下沉到基础设施层并充分利用底层基础设施的能力。

最后，我们总结一下今天的内容：

- 给大家介绍了一下我们的 SOFAMesh 项目，如果大家有计划应用 Service Mesh 技术，想上 Istio，可以尝试了解一下我们的这个项目，会让大家落地更舒服一些
- 其次给大家介绍了选择 Golang 的原因，主要是因为语言栈的长期选择。如果有正在进行 Service Mesh 技术选择的同学，可以作为参考。如果和我们一样，更愿意在未来保持 Golang 和 Java 为主要语言栈，则可以参考我们的方案，当然我们更希望你们可以和我们一起来共建 SOFAMesh 这个开源项目
- 然后给大家分享了我们遇到的几个典型问题，如何快速的支持更多通讯协议，如何让传统的SOA架构的应用程序在不进行代码修改的情况下也能从 Service Mesh 中受益，实现系统的平滑迁移。对于准备实际落地的同学会有帮助，由于时间所限未能将细节展开，大家可以会后查看资料或者直接找我们交流
- 对服务间通讯的范围进行了探讨，从原有的东西向通讯，扩展到南北向通讯，还有在 serverless 项目中的使用。希望能够让大家了解到 Service Mesh 技术可以应用的更多场景。
- 最后谈了一下切身感受：Service Mesh 技术要想完全发挥作用，需要和底层基础设施融合，以充分发挥基础设施的能力。这块的认知，会影响到 Service Mesh 的技术选型，产品方案，甚至影响组织关系。可谓至关重要，希望每位有志于此的同学能认认真真的审视这个问题。

Service Mesh 是一个新生事物，新事物在刚出现时总是会遇到各种挑战和质疑，尤其在它自身还不够完全成熟的时候。而Service Mesh 背后的Cloud Native，更是一场前所未有的巨大变革。

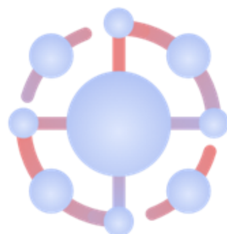
我们心怀美好愿景，憧憬未来的 Cloud Native 架构，那里有我们的 Service Mesh，有k8s，有微服务......而新的架构，新的技术，从来都不是能一蹴而就的，更不存在一帆风顺之类的美好而天真的想法。

道路从来都是人走出来的，或者说，趟出来的。作为国内 Service Mesh 技术的先驱者，我们坦言 Service Mesh 技术还不够成熟，还有很多问题等待解决，还有非常多的挑战在前面等着我们。但我们有信心相信，我们的方向是正确的，我们今天的每一份努力，每一份付出，都在让我们离目标更近一步。

鲁迅先生说：地上本没有路，走的人多了，也便成了路。在 Service Mesh 的这个方向，相信会出现越来越多努力探索的身影。这条路，我们终究会努力趟出来！

长路漫漫，吾辈当踏歌而行！

SOFAMesh @ github



SOFAMesh

https://github.com/alipay/sofa-mesh

Fork自Istio，Istio的增强落地版

SOFAMosn

https://github.com/alipay/sofa-mosn

Golang版本的Sidecar

目前 SOFAMesh 和 SOFAMosn 项目都已经在 github 开源，地址如下：

- sofa-mesh: https://github.com/alipay/sofa-mesh
- sofa-mosn: https://github.com/alipay/sofa-mosn

欢迎大家关注这两个项目的进展，如果能star一下表示支持就更好了，感激不尽！

更希望可以一起来参与这两个项目的建设，期待Issue，期待PR！

欢迎志同道合者加入Service Mesher技术社区

http://www.servicemesher.com

ServiceMesh中国技术社区

微信公众号

servicemesher

对 Service Mesh 技术感兴趣的同学，可以关注servicemesher社区，这是一个中立的纯技术社区，汇集了当前国内大部分 Service Mesh 的技术人员。我本人也是 servicemesher 社区的创始人之一，这个社区的使命是传播 Service Mesh 技术，加强行业内部交流，促进开源文化构建，推动 Service Mesh 在企业落地。

可以通过访问社区网站 http://www.servicemesher.com 获取各种技术资讯和社区活动信息，可以关注 servicemesher 社区的微信公众号得到及时的信息推动。我们拥有一个庞大的翻译组，除了翻译各种 Service Mesh 相关的技术博客和新闻，还负责 Envoy 和 Istio 两个项目官方文档的日常维护。

也欢迎大家加入 servicemesher 社区的微信交流群，请按照 servicemesher.com 网站的 "联系我们" 页面的要求加入微信交流群。

最后，厚颜推荐一下我自己的个人技术博客 https://skyao.io ，欢迎浏览和交流。

今天的内容到此结束，非常感谢大家的聆听，有缘下次再会！谢谢大家！