istio 简介和基础组件原理（服务网格 Service Mesh）

2018 年 11 月 27 日 13:06:46 数据架构师 阅读数：173 更多

所属专栏： 微服务架构

微服务对于每个功能的开发细化了，但是对与系统的管理复杂度增强了，尤其是网络流量的管理。

试想一下：黑名单，导流，加密，访问控制，流量监控，熔断，限速，收费功能，数据流阶段延迟 这种在网络层上的功能难道要在每个应用代码中实现么？

Istio 简介

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），也不再需要自己手动实现各种复杂的服务治理功能（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端可以进行简单的直接网络访问，就可以通过将网络层委托 Istio，从而获得一系列的完备功能。可以近似的理解为：

Istio = 微服务框架 + 服务治理。

Istio 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。

微服务的两面性

最近两三年来微服务方兴未艾， 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， 让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

1、什么是 Service Mesh（服务网格）？

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。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，它们之间的服务调用关系就会形成网格。

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。

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：

如果我们可以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建的网络路径中，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)：

流量控制：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。

这些功能极大地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

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?

不妨设想一下，在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，要如何开发我们的应用程序，才可以做到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，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 Stagging 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 的住技术和时间，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。

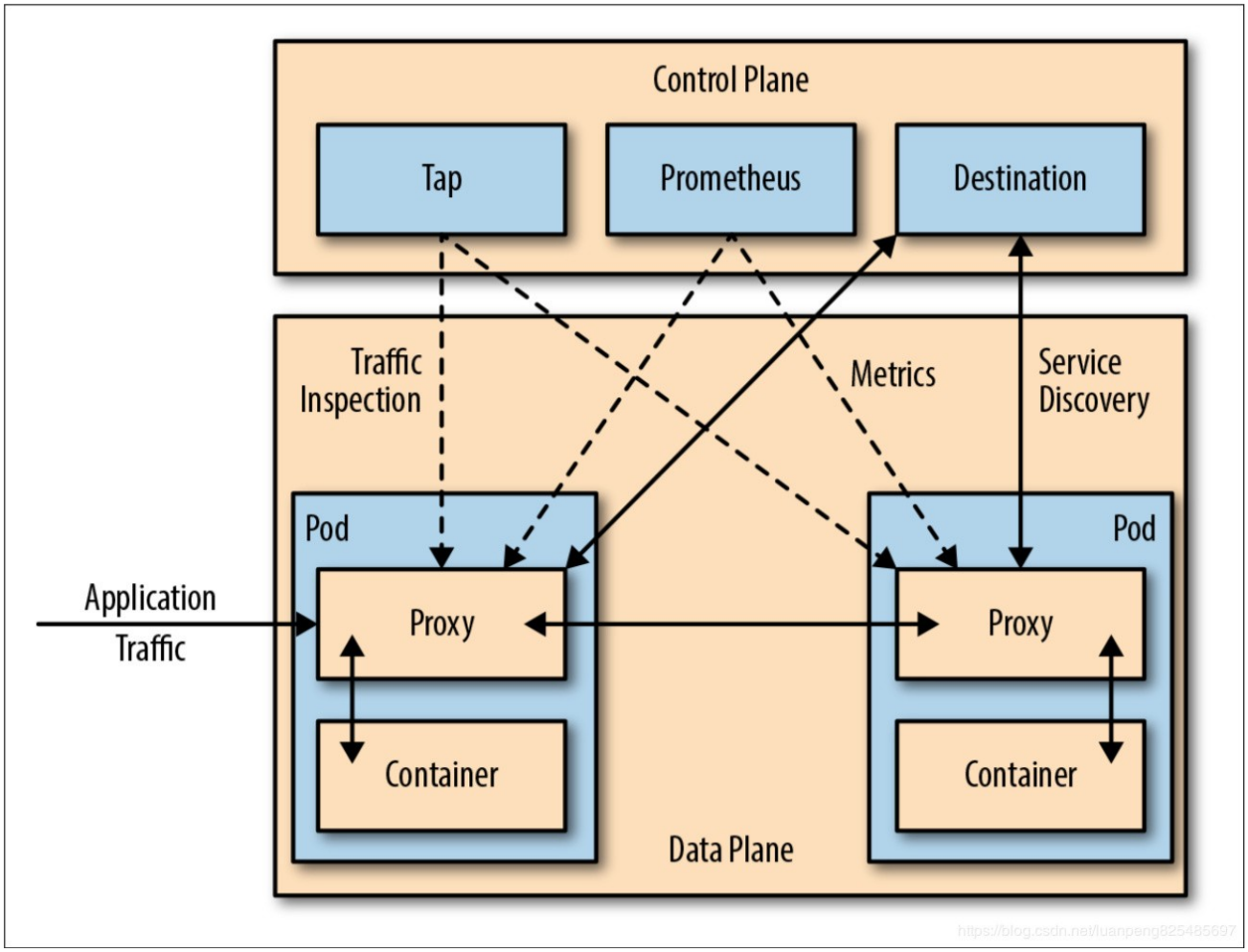
总结:

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 isito 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。

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。



当前流行的两款开源的服务网格 Istio 和 Linkerd 实际上都是这种构造，只不过 Istio 的划分更清晰，而且部署更零散，很多组件都被拆分，控制平面中包括 Mixer、Pilot、Auth，数据平面默认是用 Envoy；而 Linkerd 中只分为 Linkerd 做数据平面，namerd 作为控制平面。

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。

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。

控制平面的特点：

```
不直接解析数据包

与控制平面中的代理通信，下发策略和配置
```

```
负责网络行为的可视化

通常提供 API 或者命令行工具可用于配置版本化管理，便于持续集成和部署
```

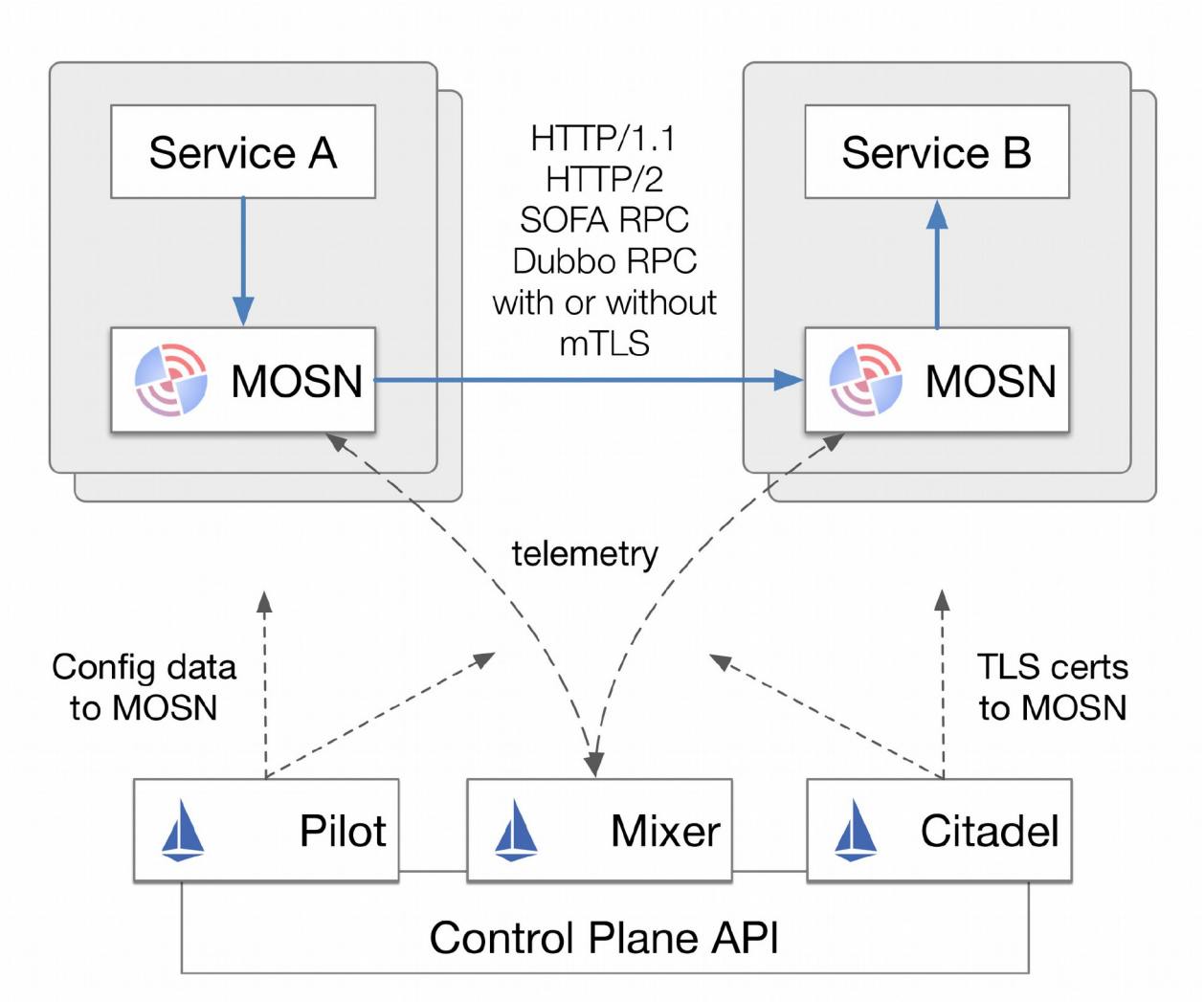
数据平面的特点：

```
通常是按照无状态目标设计的，但实际上为了提高流量转发性能，需要缓存一些数据，因此无状态也是有争议的

直接处理入站和出站数据包，转发、路由、健康检查、负载均衡、认证、鉴权、产生监控数据等

对应用来说透明，即可以做到无感知部署
```

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++，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。

控制平面-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 指定它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。

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。

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

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。

Polit 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

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

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。

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。

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

控制平面-Mixer

Mixer 翻译成中文是混音器,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，并在运维人员控制线。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。也不是视图定义一个通用的 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。

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。

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。

控制平面-Istio-Auth（Citadel）

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

包括三个组件：身份，密钥管理和通信安全。

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

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。

数据平面-Envoy

Istio 作为控制平面，在每个服务中注入一个 Envoy 代理以 Sidecar 形式运行来拦截所有进出服务的流量，同时对流量加以控制。所以我们需要关心的是 istio 是如何进行 Sidecar 注入到应用 pod 中的。下面我们就来了解一下

Sidecar 模式

什么是 Sidecar 模式

将应用程序的功能划分为单独的进程可以被视为 Sidecar 模式。Sidecar 设计模式允许你为应用程序添加许多功能，而无需额外第三方组件的配置和代码。

就如 Sidecar 连接着摩托车一样，类似地在软件架构中， Sidecar 应用是连接到父应用并且为其扩展或者增强功能。Sidecar 应用与主应用程序松散耦合。

让我用一个例子解释一下。想象一下假如你有 6 个微服务相互通信以确定一个包裹的成本。

每个微服务都需要具有可观察性、监控、日志记录、配置、断路器等功能。所有这些功能都是根据一些行业标准的第三方库在每个微服务中实现的。

但再想一想，这不是多余吗？它不会增加应用程序的整体复杂性吗？如果你的应用程序是用不同的语言编写时会发生什么——如何合并那些特定用于 .Net、Java、Python 等语言的第三方库。

使用 Sidecar 模式的优势

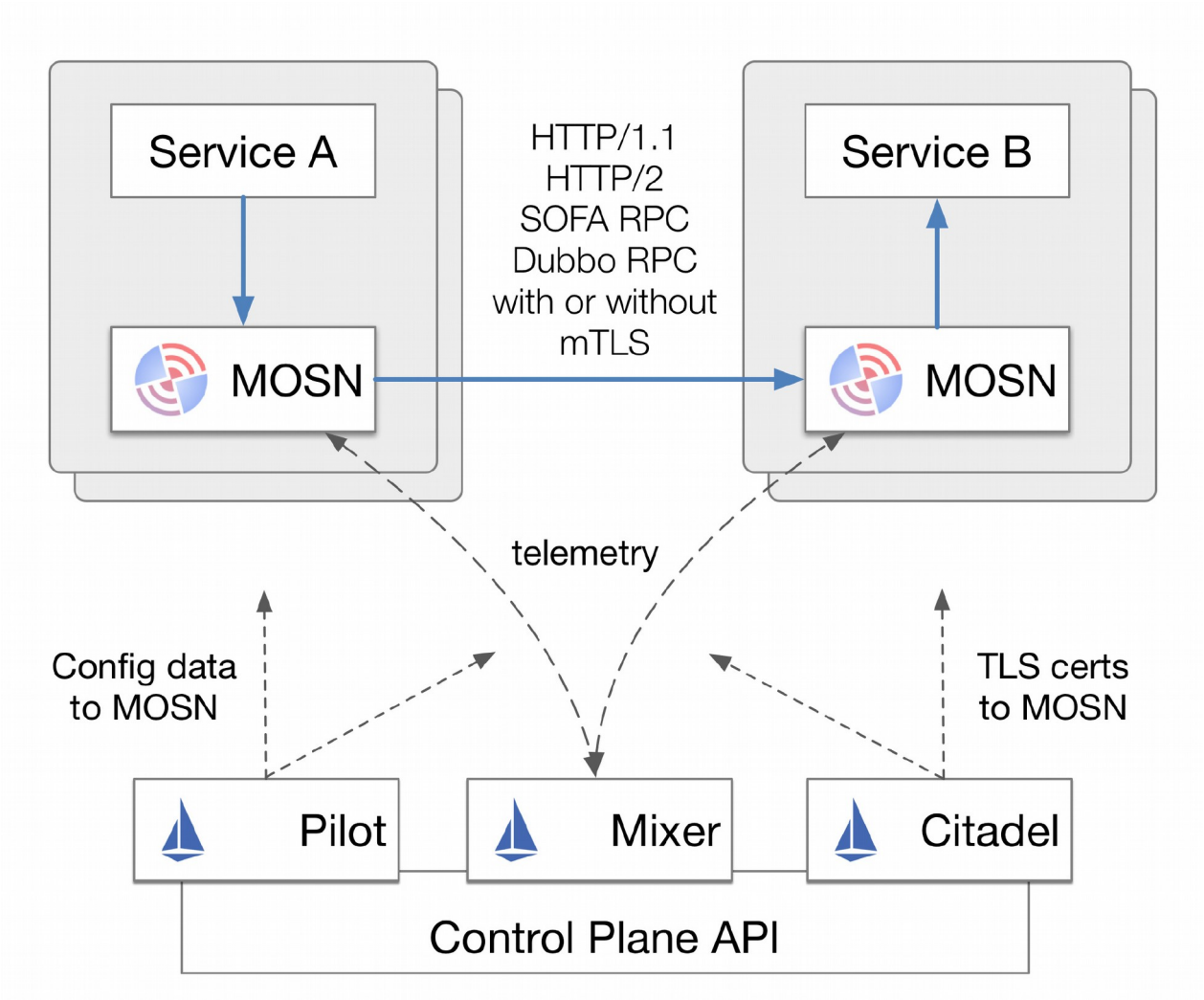
- 通过抽象出与功能相关的共同基础设施到一个不同层降低了微服务代码的复杂度。
- 因为你不再需要编写相同的第三方组件配置文件和代码，所以能够降低微服务架构中的代码重复度。
- 降低应用程序代码和底层平台的耦合度。

Sidecar 模式如何工作

Sidecar 是容器应用模式的一种，也是在 Service Mesh 中发扬光大的一种模式，详见 Service Mesh 架构解析，其中详细描述了节点代理和 Sidecar 模式的 Service Mesh 架构。

使用 Sidecar 模式部署服务网格时，无需在节点上运行代理（因此您不需要基础结构的协作），但是集群中将运行多个相同的 Sidecar 副本。从另一个角度看：我可以将一组微服务部署到一个服务网格中，你也可以部署一个有特定实现的服务网格。在 Sidecar 部署方式中，**你会为每个应用的容器部署一个伴生容器。Sidecar 接管进出应用容器的所有流量**。在 Kubernetes 的 Pod 中，**在原有的应用容器旁边运行一个 Sidecar 容器，可以理解为两个容器共享存储、网络等资源，可以广义的将这个注入了 Sidecar 容器的 Pod 理解为一台主机，两个容器共享主机资源**。

例如下图 SOFAMesh & SOFA MOSN—基于 Istio 构建的用于应对大规模流量的 Service Mesh 解决方案的架构图中描述的，MOSN 作为 Sidecar 的方式和应用运行在同一个 Pod 中，拦截所有进出应用容器的流量，SOFAMesh 兼容 Istio，其中使用 Go 语言开发的 SOFAMosn 替换了 Envoy。



Sidecar 注入详解

在讲解 Istio 如何将 Envoy 代理注入到应用程序 Pod 中之前，我们需要先了解以下几个概念：

- Sidecar 模式：容器应用模式之一，Service Mesh 架构的一种实现方式。
- Init 容器：Pod 中的一种专用的容器，在应用程序容器启动之前运行，用来包含一些应用镜像中不存在的实用工具或安装脚本。
- iptables：流量劫持是通过 iptables 转发实现的。

Init 容器

Init 容器是一种专用容器，它在应用程序容器启动之前运行，用来包含一些应用镜像中不存在的实用工具或安装脚本。

一个 Pod 中可以指定多个 Init 容器，如果指定了多个，那么 Init 容器将会按顺序依次运行。只有当前面的 Init 容器必须运行成功后，才可以运行下一个 Init 容器。当所有的 Init 容器运行完成后，Kubernetes 才初始化 Pod 和运行应用容器。

Init 容器使用 Linux Namespace，所以相对应用程序容器来说具有不同的文件系统视图。因此，它们能够具有访问 Secret 的权限，而应用程序容器则不能。

在 Pod 启动过程中，Init 容器会按顺序在网络和数据卷初始化之后启动。每个容器必须在下一个容器启动之前成功退出。如果由于运行时或失败退出，将导致容器启动失败，它会根据 Pod 的 restartPolicy 指定的策略进行重试。然而，如果 Pod 的 restartPolicy 设置为 Always，Init 容器失败时会使用 RestartPolicy 策略。

在所有的 Init 容器没有成功之前，Pod 将不会变成 Ready 状态。Init 容器的端口将不会在 Service 中进行聚集。 正在初始化中的 Pod 处于 Pending 状态，但应该会将 Initializing 状态设置为 true。Init 容器运行完成以后就会自动终止。

这里举一个简单的例子

我们部署一个简单的容器，在其中添加了两个 init 容器，这两个容器 init1-container 解析 myservice 域名（服务），如果不能解析（也就是服务没有部署），则会循环打印，不结束。这样后面的 init 容器和应用容器也就不会部署。容器 init2-container 解析 mydb 域名

```
apiVersion: v1

kind: Pod

metadata:

  name: myapp-pod

  labels:

    app: myapp

spec:

  containers:

  - name: myapp-container

    image: busybox
```

```
      command: ['sh', '-c', 'echo The app is running! && sleep 3600']

   initContainers:

   - name: init1-container

      image: registry.cn-hangzhou.aliyuncs.com/google_containers/coredns:1.1.3

      command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
# until 后面的 nslookup myservice 输出状态必须为非 0

   - name: init2-container

      image: registry.cn-hangzhou.aliyuncs.com/google_containers/coredns:1.1.3

      command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']   # until 后面的 nslookup myservice 输出状态必须为非 0

12345678910111213141516171 8
```

只有当我们真正部署了 myservice 和 mydb 服务，上面的 容器才能正常运行。可以通过查看 `kubectl logs myapp-pod -c init-myservice` 查看初始话容器的日志

```
kind: Service

apiVersion: v1

metadata:

   name: myservice

spec:

   ports:

      - protocol: TCP

         port: 80

         targetPort: 9376

---
```

```
kind: Service

apiVersion: v1

metadata:

    name: mydb

spec:

    ports:

        - protocol: TCP

            port: 80

            targetPort: 9377

12345678910111213141516171819
```

关于 Init 容器的详细信息请参考 Init 容器 - Kubernetes 中文指南/云原生应用架构实践手册。

Sidecar 注入示例分析

比如我们现在有一个 yaml 文件，这里使用官方的例子 bookinfo.yaml

```
apiVersion: v1

kind: Service

metadata:

    name: productpage

    labels:

        app: productpage

spec:

    ports:
```

```
  - port: 9080
    name: http
  selector:
    app: productpage
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: productpage-v1
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: productpage
        version: v1
    spec:
      containers:
      - name: productpage
        image: istio/examples-bookinfo-productpage-v1:1.8.0
        imagePullPolicy: IfNotPresent
```

```
            ports:

            - containerPort: 9080

12345678910111213141516171819202122232425262728293031
```

正常情况下我们使用

```
kubectl create -f bookinfo.yaml

1
```

完成部署。

现在我们使用 istioctl 进行一些操作，要想使用 istioctl 命令需要先下载，进入 Istio release 页面，下载对应目标操作系统的安装文件。将 istioctl 客户端二进制文件加到 PATH 中。

打开系统配置文件

```
sudo vim /etc/profile

1
```

在末尾添加 istio 文件夹下的 bin 目录

```
export PATH=/soft/istio-1.0.4/bin:$PATH

1
```

现在先用 istioctl 命令变更一下 bookinfo.yaml 文件。
（关于 istioctl 命令，可以参考 https://istio.io/zh/docs/reference/commands/istioctl/）

```
istioctl kube-inject -f bookinfo.yaml >> bookinfo_with_sidecar.yaml

1
```

我们将输出导入到一个新的 yaml 文件，bookinfo_with_sidecar.yaml，这个文件的内容就是注入了 istio 底层的 yaml 部署文件。

我们看到 Service 的配置没有变化，所有的变化都在 Deployment 里，Istio 给应用 Pod 注入的配置主要包括：

- Init 容器 istio-init：用于给 Sidecar 容器即 Envoy 代理做初始化，设置 iptables 端口转发
- Envoy sidecar 容器 istio-proxy：运行 Envoy 代理

对比 bookinfo_with_sidecar.yaml 文件和 bookinfo.yaml，可以看到该命令在 bookinfo.yaml 的基础上做了如下改动：

为每个 pod 增加了一个代理 container，该 container 用于处理应用 container 之间的通信，包括服务发现，路由规则处理等。从下面的配置文件中可以看到 proxy container 通过 15001 端口进行监听，接收应用 container 的流量。

为每个 pod 增加了一个 init-container，该 container 用于配置 iptable，将应用 container 的流量导入到代理 container 中。

通过该方式，不需要用户手动修改 kubernetes 的部署文件，即可在部署服务时将 sidecar 和应用一起部署。

接下来将分别解析下这两个容器。

Init 容器解析

Istio 在 Pod 中注入的 Init 容器名为 istio-init，我们在上面 Istio 注入完成后的 YAML 文件中看到了该容器的启动参数：

```
-p 15001 -u 1337 -m REDIRECT -i '*' -x "" -b 9080 -d ""

1
```

我们再检查下该容器的 Dockerfile 看看 ENTRYPOINT 是什么以确定启动时执行的命令。

```
FROM ubuntu:xenial

RUN apt-get update && apt-get install -y \

      iproute2 \

      iptables \

 && rm -rf /var/lib/apt/lists/*



ADD istio-iptables.sh /usr/local/bin/
```

```
ENTRYPOINT ["/usr/local/bin/istio-iptables.sh"]

12345678
```

我们看到 istio-init 容器的入口是 /usr/local/bin/istio-iptables.sh 脚本，再按图索骥看看这个脚本里到底写的什么，该脚本的位置在 Istio 源码仓库的 tools/deb/istio-iptables.sh，一共 300 多行，就不贴在这里了。下面我们就来解析下这个启动脚本。

Init 容器启动入口

Init 容器的启动入口是 /usr/local/bin/istio-iptables.sh 脚本，该脚本的用法如下：

```
$ istio-iptables.sh -p PORT -u UID -g GID [-m mode] [-b ports] [-d ports] [-i CIDR] [-x CIDR] [-h]

    -p: 指定重定向所有 TCP 流量的 Envoy 端口（默认为 $ENVOY_PORT = 15001）

    -u: 指定未应用重定向的用户的 UID。通常，这是代理容器的 UID（默认为 $ENVOY_USER 的 uid，istio_proxy 的 uid 或 1337）

    -g: 指定未应用重定向的用户的 GID。（与 -u param 相同的默认值）

    -m: 指定入站连接重定向到 Envoy 的模式，“REDIRECT” 或 “TPROXY”（默认为 $ISTIO_INBOUND_INTERCEPTION_MODE)

    -b: 逗号分隔的入站端口列表，其流量将重定向到 Envoy（可选）。使用通配符 “*” 表示重定向所有端口。为空时表示禁用所有入站重定向（默认为 $ISTIO_INBOUND_PORTS）

    -d: 指定要从重定向到 Envoy 中排除（可选）的入站端口列表，以逗号格式分隔。使用通配符“*” 表示重定向所有入站流量（默认为 $ISTIO_LOCAL_EXCLUDE_PORTS）

    -i: 指定重定向到 Envoy（可选）的 IP 地址范围，以逗号分隔的 CIDR 格式列表。使用通配符 “*” 表示重定向所有出站流量。空列表将禁用所有出站重定向（默认为 $ISTIO_SERVICE_CIDR）

    -x: 指定将从重定向中排除的 IP 地址范围，以逗号分隔的 CIDR 格式列表。使用通配符 “*” 表示重定向所有出站流量（默认为 $ISTIO_SERVICE_EXCLUDE_CIDR）。

123456789
```

环境变量位于 $ISTIO_SIDECAR_CONFIG（默认在：/var/lib/istio/envoy/sidecar.env）
通过查看该脚本你将看到，以上传入的参数都会重新组装成 **iptables 命令**的参数。

再参考 istio-init 容器的启动参数，完整的启动命令如下：

```
$ /usr/local/bin/istio-iptables.sh -p 15001 -u 1337 -m REDIRECT -i '*' -x "" -b 9080 -d ""

1
```

该容器存在的意义就是让 Envoy 代理可以拦截所有的进出 Pod 的流量，即将入站流量重定向到 Sidecar，再拦截应用容器的出站流量经过 Sidecar 处理后再出站。

命令解析

这条启动命令的作用是：

- 将应用容器的所有流量都转发到 Envoy 的 15001 端口。
- 使用 istio-proxy 用户身份运行， UID 为 1337，即 Envoy 所处的用户空间，这也是 istio-proxy 容器默认使用的用户，见 YAML 配置中的 runAsUser 字段。
- 使用默认的 REDIRECT 模式来重定向流量。
- 将所有出站流量都重定向到 Envoy 代理。
- 将所有访问 9080 端口（即应用容器 productpage 的端口）的流量重定向到 Envoy 代理。
- 因为 Init 容器初始化完毕后就会自动终止，因为我们无法登陆到容器中查看 iptables 信息，但是 Init 容器初始化结果会保留到应用容器和 Sidecar 容器中。

istio-proxy 容器解析

为了查看 iptables 配置，我们需要登陆到 Sidecar 容器中使用 root 用户来查看，因为 kubectl 无法使用特权模式来远程操作 docker 容器，所以我们需要登陆到 productpage Pod 所在的主机上使用 docker 命令登陆容器中查看。

查看 productpage Pod 所在的主机。

```
$ kubectl -n default get pod -l app=productpage -o wide

NAME                READY   STATUS   RESTARTS  AGE    IP       NODE

productpage-v1-745ffc55b7-2l2lw  2/2    Running  0      1d     172.33.78.10  node3

123
```

从输出结果中可以看到该 Pod 运行在 node3 上，使用 vagrant 命令登陆到 node3 主机中并切换为 root 用户。

```
$ vagrant ssh node3

$ sudo -i

12
```

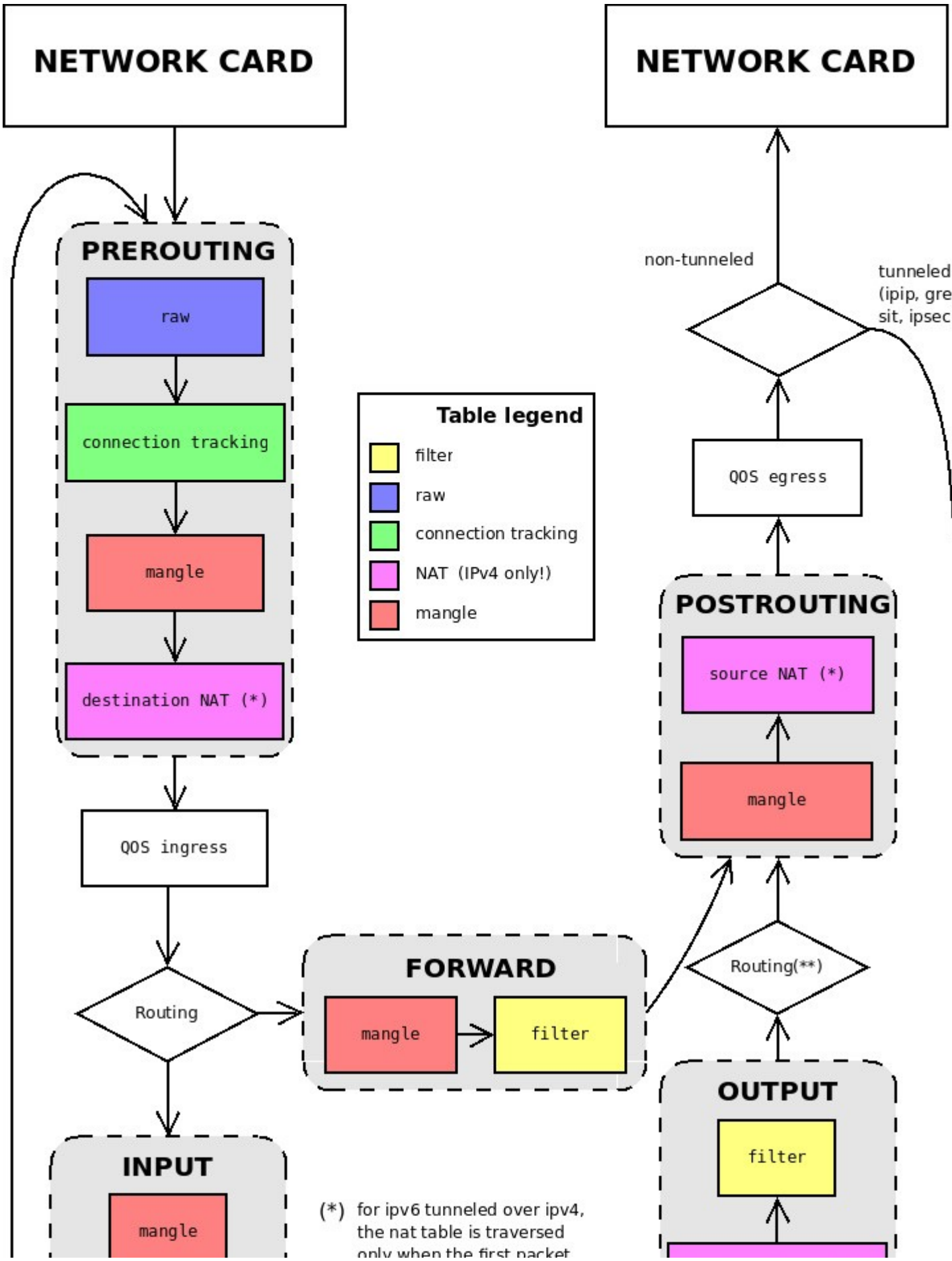
查看 iptables 配置，列出 NAT（网络地址转换）表的所有规则，因为在 Init 容器启动的时候选择给 istio-iptables.sh 传递的参数中指定将入站流量重定向到 Envoy 的模式为“REDIRECT”，因此在 iptables 中将只有 NAT 表的规格配置，如果选择 TPROXY 还会有 mangle 表配置。iptables 命令的详细用法请参考 iptables，规则配置请参考 iptables 规则配置。

理解 iptables

iptables 是 Linux 内核中的防火墙软件 netfilter 的管理工具，位于用户空间，同时也是 netfilter 的一部分。Netfilter 位于内核空间，不仅有网络地址转换的功能，也具备数据包内容修改、以及数据包过滤等防火墙功能。

在了解 Init 容器初始化的 iptables 之前，我们先来了解下 iptables 和规则配置。

下图展示了 iptables 调用链。


NETWORK CARD
NETWORK CARD
PREROUTING
raw
connection tracking
mangle
destination NAT (*)
QOS ingress
Routing
FORWARD
mangle
filter
INPUT
mangle
Table legend
filter
raw
connection tracking
NAT (IPv4 only!)
mangle
non-tunneled
tunneled
(ipip, gre
sit, ipsec
QOS egress
POSTROUTING
source NAT (*)
mangle
Routing(**)
OUTPUT
filter
(*) for ipv6 tunneled over ipv4,
the nat table is traversed

详细的 iptable 就不讲解了，读者可以参考 https://jimmysong.io/istio-handbook/concepts/sidecar-injection-deep-dive.html

Istio 部署

Istio 部署教程：https://blog.csdn.net/luanpeng825485697/article/details/84500612

服务网格的实现模式

下图是使用 Service Mesh 架构的最终形式