目录

Istio官方文档中文版

介绍

Istio是由Google/IBM/Lyft共同开发的新一代Service Mesh开源项目。

这是Istio官方文档的中文翻译版，由Service Mesh中国社区组织翻译并更新维护。

文档内容发布于gitbook，请点击下面的链接阅读或者下载电子版本:

- 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，HTTPS
  - 国内服务器：腾讯云加速，国内网速极快，非HTTPS
- 下载pdf格式
- 下载mobi格式
- 下载epub格式

Service Mesh中国

目前Service Mesh技术还比较新颖，为了方便技术交流，我们组建了Service Mesh中国社区，欢迎加入：

http://www.servicemesh.cn/

您可以通过Awesome Service Mesh资料清单快速了解Service Mesh技术和相关的Istio/linkerd等开源项目，这份清单由Service Mesh中国社区更新维护：

https://servicemesh.gitbooks.io/awesome-servicemesh/

如果想加入Service Mesh中国社区的微信群，请联系微信ID `xiaoshu062` ，注明“服务网格”。

翻译说明

目前istio官方文档的翻译由Service Mesh中文网在主持，这是一个公益性的工作。

贡献您的力量

如果有朋友有意愿加入我们的翻译工作，可以有如下方式贡献您的力量：

1. 审核和校对：如果您在阅读时发现内容有谬误，可以通过 github issue提交issue给我们，您也可以fork出来然后通过PR提交更改
2. 直接参与翻译工作： 您可以报名加入我们的翻译小组，请先加入Service Mesh技术社区，然后在微信群中联系我们。

> 加入方式：请联系微信ID xiaoshu062 ，注明“服务网格”。

工作方式

以下是和我们翻译内容相关的信息：

- Istio官方文档在线浏览地址
- Istio官方文档源文件托管地址
- 保存翻译后的中文内容的github地址
- 中文翻译内容发布地址：目前用的是gitbook，会自动从github拉取最新内容生成html发布出来

我们目前采用github保存翻译后的内容，翻译团队使用github issue和project来管理。具体方式为：

1. 未翻译的内容会以github issue的方式拆分为多个issue
2. 请在github issue中领取感兴趣的任务，并将issue assign到自己的github账号：切记必须这样做明确认领，避免其他人在不知情的情况下重复翻译同一个内容
3. 翻译完成后在微信群中联系其他人做review
4. review完成，关闭issue

过程中：

1. 如果觉得issue包含的内容太多，可以直接修改issue，缩小内容的范围，然后为减少的内容新开其他issue（可以新开一个或者多个）
2. 直接提交翻译后的内容，哪怕还没有review（甚至只翻译了一半）
3. 只使用master分支，暂时不拉分支，避免麻烦

约定和术语表

术语表

```
service       服务
microservice 微服务
application   应用/应用程序
mutual TLS    双向TLS

next step          下一步
before you begin   开始之前
cleanup            清除
understanding What happened 理解原理
Further reading    进阶阅读

configure 配置
setting   设置
traffic   流量
authentication  认证
authorization   授权
```

约定

以下词汇不翻译：

```
sidecar
BookInfo
productpage
reviews
ratings
HTTP header
TLS
```

官方文档

译注

原英文文档地址为 https://istio.io/docs/

正文

欢迎来到Istio。

欢迎来到Istio的最新文档主页。从这里您可以通过以下链接了解有关Istio的所有信息：

- 概念

  概念解释了Istio的一些关键点。在这里您可以了解Istio的工作原理及其实现。

- 安装

  在不同的环境下（如Kubernetes、Consul等）安装Istio控制平面，以及在应用程序部署中安装sidecar。

- 任务

  向您展示如何使用Istio执行单个定向活动。

- 示例

  示例是可以完全独立工作的例子，旨在突出特定的Istio功能集。

- 参考文档

  命令行选项，配置选项，API定义和过程的详细列表。

概念

本章帮助您了解Istio系统的不同部分及其使用的抽象。

- Istio是什么？

  概述：提供Istio的概念介绍，包括其解决的问题和宏观架构。

  设计目标：描述了Istio设计时坚持的核心原则。

- 流量管理

  概述：概述Istio中的流量管理及其功能。

  Pilot：引入Pilot，负责在服务网格中管理Envoy代理的分布式部署的组件。

  请求路由：描述在Istio服务网格中服务之间如何路由请求。

  发现和负载均衡：描述在网格中的服务实例之间的流量如何负载均衡。

  处理故障：Envoy中的故障恢复功能概述，可以被未经修改的应用程序来利用，以提高鲁棒性并防止级联故障。

  故障注入：介绍系统故障注入的概念，可用于发现跨服务的冲突故障恢复策略。

  规则配置：提供Istio在服务网格中配置流量管理规则所使用的领域特定语言的高级概述。

- 网络和认证

  认证：认证设计的深层架构，为Istio提供了安全的通信通道和强有力的身份。

- 策略与控制

  属性：解释属性的重要概念，即策略和控制是如何应用于网格中的服务之上的中心机制。

  Mixer：Mixer设计的深层架构，提供服务网格内的策略和控制机制。

  Mixer配置：用于配置Mixer的关键概念的概述。

  Mixer Aspect配置：说明如何配置Mixer Aspect及其依赖项。

Istio是什么？

- 概述：提供Istio的概念介绍，包括其解决的问题和宏观架构。
- 设计目标：描述了Istio设计时坚持的核心原则。

概述

本文档介绍了Istio——一个用来连接、管理和保护微服务的开放平台。Istio提供一种简单的方式来建立已部署服务网络，具备负载均衡、服务间认证、监控等功能，而不需要改动任何服务代码。想要为服务增加对Istio的支持，您只需要在环境中部署一个特殊的边车（sidecar），使用Istio控制面板功能配置和管理代理，拦截微服务之间的所有网络通信。

Istio目前仅支持在Kubernetes上的服务部署，但未来版本中将支持其他环境。

有关Istio组件的详细概念信息，请参阅我们的其他概念指南。

为什么要使用**Istio**？

在从单体应用程序向分布式微服务架构的转型过程中，开发人员和运维人员面临诸多挑战，使用Istio可以解决这些问题。术语服务网格（**Service Mesh**）通常用于描述构成这些应用程序的微服务网络以及它们之间的交互。随着规模和复杂性的增长，服务网格越来越难以理解和管理。它的需求包括服务发现、负载均衡、故障恢复、指标收集和监控以及通常更加复杂的运维需求，例如A/B测试、金丝雀发布、限流、访问控制和端到端认证等。

Istio提供了一个完整的解决方案，通过为整个服务网格提供行为洞察和操作控制来满足微服务应用程序的多样化需求。它在服务网络中统一提供了许多关键功能：

- 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旨在可以在各种环境中运行，包括跨云、预置环境、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。

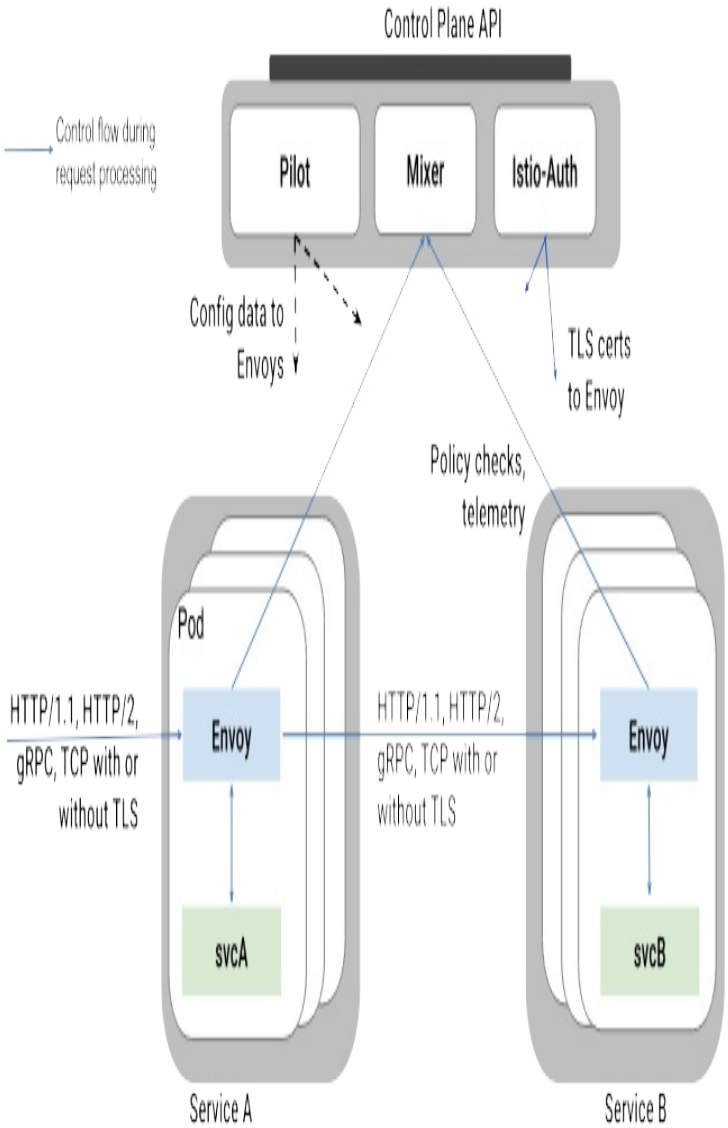
这些功能极大的减少了应用程序代码，底层平台和策略之间的耦合。耦合的减少不仅使服务更容易实现，而且还使运维人员更容易地在环境之间移动应用程序部署，或换用新的策略方案。因此，结果就是应用程序从本质上变得更容易移动。

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。

下图显示了构成每个面板的不同组件：



Envoy

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。Envoy的许多内置功能被istio发扬光大，例如动态服务发现，负载均衡，TLS终止，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，故障注入和丰富指标。

Envoy被部署为边车,和对应服务在同一个Kubernetes pod中。这允许Istio将大量关于流量行为的信号作为属性提取出来，而这些属性又可以在Mixer中用于执行策略决策，并发送给监控系统，以提供整个网格行为的信息。边车代理模型还可以将Istio的功能添加到现有部署中，而无需重新构建或重写代码。可以阅读更多来了解为什么我们在设计目标中选择这种方式。

Mixer

Mixer负责在服务网格上执行访问控制和使用策略，并从Envoy代理和其他服务收集遥测数据。代理提取请求级属性，发送到Mixer进行评估。有关属性提取和策略评估的更多信息，请参见Mixer配置。Mixer包括一个灵活的插件模型，使其能够接入到各种主机环境和基础设施后端，从这些细节中抽象出Envoy代理和Istio管理的服务。

Pilot

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，为他们提供用户服务的抽象表示，独立于底层平台。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。

Istio-Auth

Istio-Auth提供强大的服务间认证和终端用户认证，使用交互TLS，内置身份和证书管理。可以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来执行策略的能力。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。

下一步

- 了解Istio的设计目标。
- 探索我们的指南。
- 在我们其他的概念指南中详细了解Istio组件。
- 使用我们的任务指南，了解如何用自己的服务部署Istio。

设计目标

本节概述指导Istio设计的核心原则。

Istio的架构设计中有几个关键目标，这些目标对于使系统能够应对大规模流量和高性能地服务处理至关重要。

- 最大化透明度

若想Istio被采纳，应该让运维和开发人员只需付出很少的代价就可以从中受益。为此，Istio将自身自动注入到服务间所有的网络路径中。Istio使用sidecar代理来捕获流量，并且在尽可能的地方自动编程网络层，以路由流量通过这些代理，而无需对已部署的应用程序代码进行任何改动。在Kubernetes中，代理被注入到pod中，通过编写iptable规则来捕获流量。一旦注入sidecar代理到pod中并且修改路由规则，Istio就能够调解所有流量。这个原则也适用于性能。当将Istio应用于部署时，运维人员可以发现，为提供这些功能而增加的资源开销是很小的。所有组件和API在设计时都必须考虑性能和规模。

- 增量

随着运维人员和开发人员越来越依赖Istio提供的功能，系统必然和他们的需求一起成长。虽然我们期望继续自己添加新功能，但是我们预计最大的需求是扩展策略系统，集成其他策略和控制来源，并将网格行为信号传播到其他系统进行分析。策略运行时支持标准扩展机制以便插入到其他服务中。此外，它允许扩展词汇表，以允许基于网格生成的新信号来执行策略。

- 可移植性

使用Istio的生态系统将在很多维度上有差异。Istio必须能够以最少的代价运行在任何云或预置环境中。将基于Istio的服务移植到新环境应该是轻而易举的，而使用Istio将一个服务同时部署到多个环境中也是可行的（例如，在多个云上进行冗余部署）。

- 策略一致性

在服务间的API调用中，策略的应用使得可以对网格间行为进行全面的控制，但对于无需在API级别表达的资源来说，对资源应用策略也同样重要。例如，将配额应用到ML训练任务消耗的CPU数量上，比将配额应用到启动这个工作的调用上更为

有用。因此，策略系统作为独特的服务来维护，具有自己的API，而不是将其放到代理/sidecar中，这容许服务根据需要直接与其集成。

流量管理

- 概述:概述Istio中的流量管理及其功能。
- Pilot：引入Pilot，负责在服务网格中管理Envoy代理的分布式部署的组件。
- 请求路由：描述在Istio服务网格中服务之间如何路由请求。
- 发现和负载均衡:描述在网格中的服务实例之间的流量如何负载均衡。
- 处理故障: Envoy中的故障恢复功能概述，可以被未经修改的应用程序来利用，以提高鲁棒性并防止级联故障。
- 故障注入: 介绍系统故障注入的概念，可用于发现跨服务的冲突故障恢复策略。
- 规则配置: 提供Istio在服务网格中配置流量管理规则所使用的领域特定语言的高级概述。

概述

本页概述了Istio中流量管理的工作原理，包括流量管理原则的优点。本文假设你已经阅读了 Istio是什么？并熟悉Istio的高级架构。有关单个流量管理功能的更多信息，您可以在本节其他指南中了解。

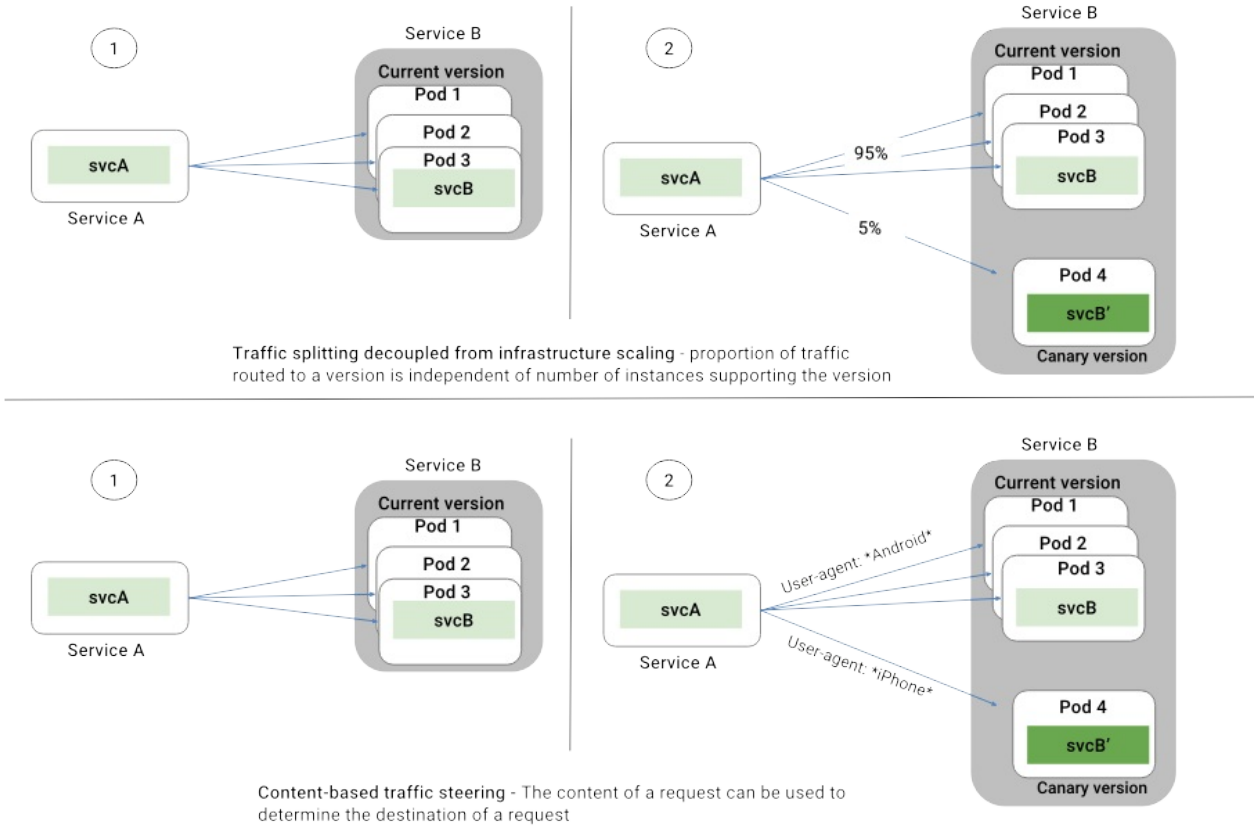
Pilot和Envoy

Istio流量管理的核心组件是Pilot，它管理和配置部署在特定Istio服务网格中的所有Envoy代理实例。它允许您指定在Envoy代理之间使用什么样的路由流量规则，并配置故障恢复功能，如超时，重试和熔断器。它还维护了网格中所有服务的规范模型，并使用这个模型，来通过发现服务让Envoy了解网格中的其他实例。

每个Envoy实例维护负载均衡信息，负载均衡信息是基于从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。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。

流量管理的好处

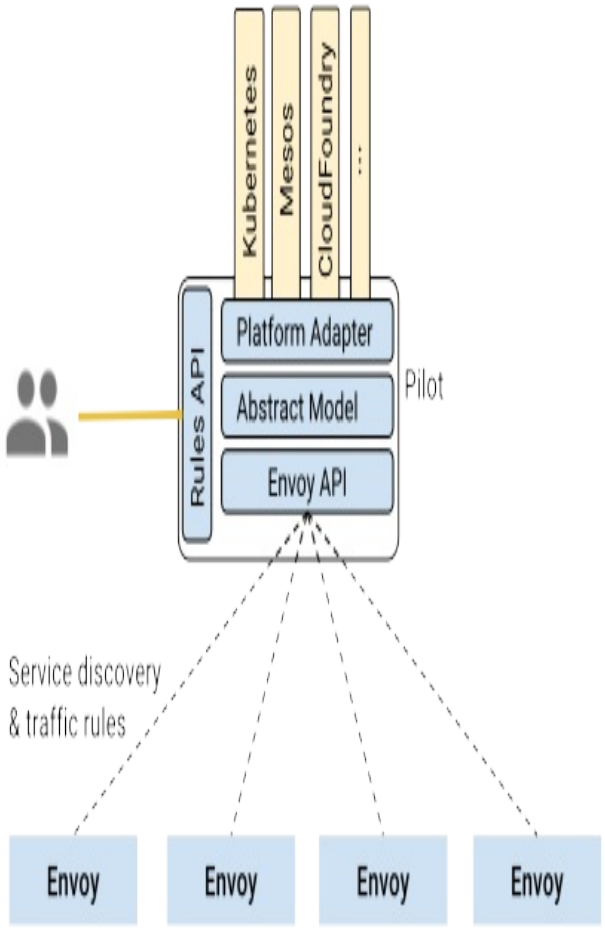
使用Istio的流量管理模型，本质上解耦流量和基础设施扩展，让运维人员通过Pilot指定他们希望流量遵循什么规则，而不是哪些特定的pod/VM应该接收流量 - Pilot和智能Envoy代理搞定其余的。因此，例如，您可以通过Pilot指定您希望特定服务的5%流量可以转到金丝雀版本，而不考虑金丝雀部署的大小，或根据请求的内容将流量发送到特定版本。



将流量从基础设施扩展中解耦，这样就可以让Istio提供各种流量管理功能，这些功能在应用程序代码之外。除了A/B测试的动态请求路由，逐步推出和金丝雀发布之外，它还使用超时，重试和熔断器处理故障恢复，最后还可以通过故障注入 来测试服务之间故障恢复策略的兼容性。这些功能都是通过在服务网格中部署的Envoy sidecar/代理来实现的。

Pilot

Pilot负责在Istio服务网格中部署的Envoy实例的生命周期。



*Pilot*架构

如上图所示，Pilot维护了网格中的服务的规范表示，这个表示是独立于底层平台的。Pilot中的平台特定适配器负责适当填充此规范模型。例如，Pilot中的Kubernetes适配器实现必要的控制器来查看Kubernetes API服务器，以得到pod注册信息的更改，入口资源以及存储流量管理规则的第三方资源。该数据被翻译成规范表示。Envoy特定配置是基于规范表示生成的。

Pilot公开了用于服务发现、负载均衡池和路由表的动态更新的API。这些API将Envoy从平台特有的细微差别中解脱出来，简化了设计并提升了跨平台的可移植性。

运维人员可以通过Pilot的Rules API指定高级流量管理规则。这些规则被翻译成低级配置，并通过discovery API分发到Envoy实例。

请求路由

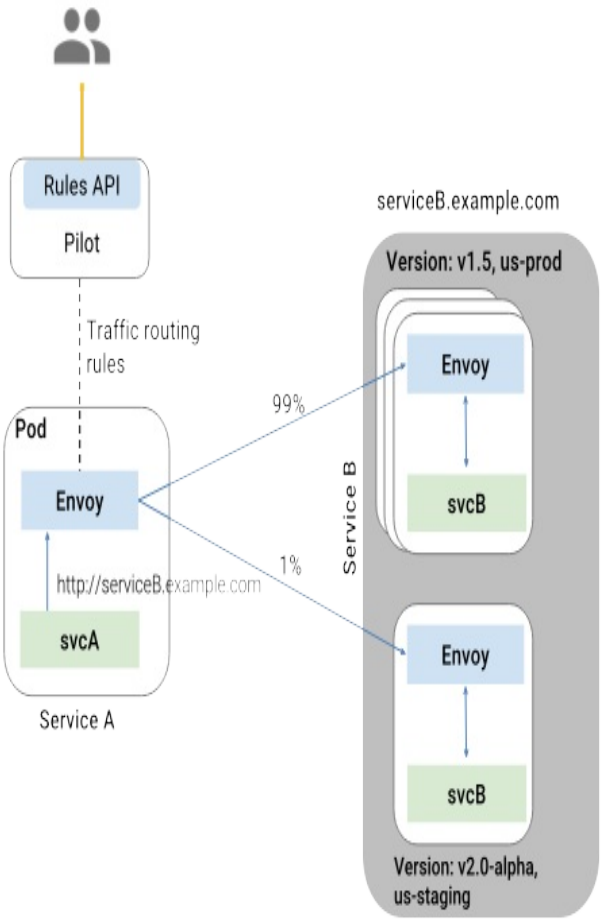
此节描述在Istio服务网格中服务之间如何路由请求。

服务模型和服务版本

如Pilot所述，特定网格中服务的规范表示由Pilot维护。服务的Istio模型和在底层平台（Kubernetes，Mesos，Cloud Foundry等）中的表示无关。特定平台的适配器负责用平台中元数据的各种字段填充内部模型表示。

Istio介绍了服务版本的概念，这是一种更细微的方法，可以通过版本（`v1`，`v2`）或环境（`staging`，`prod`）细分服务实例。这些变量不一定是API版本：它们可能是对不同环境（prod，staging，dev等）部署的相同服务的迭代更改。使用这种方式的常见场景包括A/B测试或金丝雀推出。Istio的流量路由规则可以参考服务版本，以提供对服务之间流量的附加控制。

服务之间的通讯



如上图所示，服务的客户端不知道服务不同版本间的差异。他们可以使用服务的主机名/IP地址继续访问服务。Envoy sidecar/代理拦截并转发客户端和服务器之间的所有请求/响应。

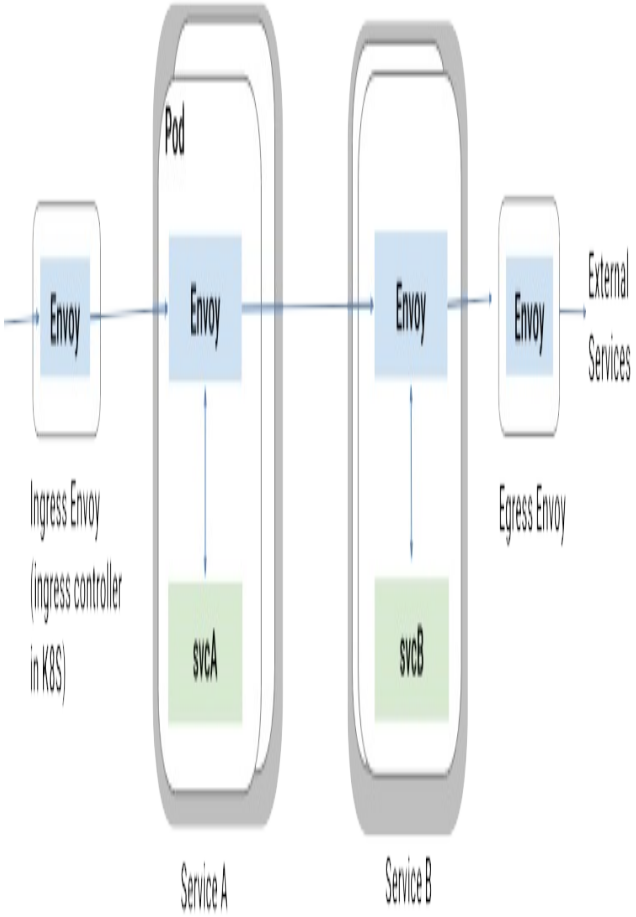
运维人员使用Pilot指定路由规则，Envoy根据这些规则动态地确定其服务版本的实际选择。该模型使应用程序代码能够将它从其依赖服务的演进中解耦出来，同时提供其他好处（参见 Mixer）。路由规则允许Envoy根据诸如header，与源/目的地相关联的标签和/或分配给每个版本的权重的标准来选择版本。

Istio还为同一服务版本的多个实例提供流量负载均衡。可以在服务发现和负载均衡中找到更多信息。

Istio不提供DNS。应用程序可以尝试使用底层平台（kube-dns，mesos-dns等）中存在的DNS服务来解析FQDN。

Ingress和Egress

Istio假定进入和离开服务网络的所有流量都会通过Envoy代理进行传输。通过将Envoy代理部署在服务之前，运维人员可以针对面向用户的服务进行A/B测试，部署金丝雀服务等。类似地，通过使用Envoy将流量路由到外部Web服务（例如，访问Maps API或视频服务API），运维人员可以添加故障恢复功能，例如超时，重试，熔断器等，并在访问这些服务的连接上获得详细指标。

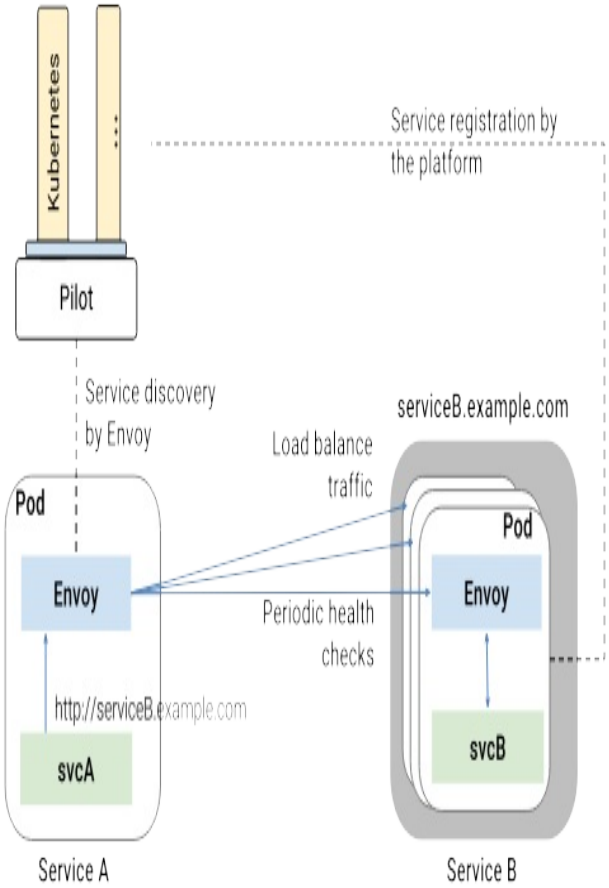


发现和负载均衡

本节描述在服务网格中Istio如何在服务实例之间实现流量的负载均衡。

服务注册: Istio假定存在服务注册表，以跟踪应用程序中服务的pod/VM。它还假设服务的新实例自动注册到服务注册表，并且不健康的实例将被自动删除。诸如Kubernetes，Mesos等平台已经为基于容器的应用程序提供了这样的功能。为基于虚拟机的应用程序提供的解决方案就更多了。

服务发现: Pilot使用来自服务注册的信息，并提供与平台无关的服务发现接口。网格中的Envoy实例执行服务发现，并相应地动态更新其负载均衡池。



如上图所示，网格中的服务使用其DNS名称访问彼此。服务的所有HTTP流量都会通过Envoy自动重新路由。Envoy在负载均衡池中的实例之间分发流量。虽然Envoy支持多种复杂的负载均衡算法，但Istio目前仅允许三种负载平衡模式：轮循，随机和带权重的最少请求。

除了负载均衡外，Envoy还会定期检查池中每个实例的运行状况。Envoy遵循熔断器风格模式，根据健康检查API调用的失败率将实例分类为不健康或健康。换句话说，当给定实例的健康检查失败次数超过预定阈值时，它将从负载均衡池中弹出。类似地，当通过的健康检查数超过预定阈值时，该实例将被添加回负载均衡池。您可以在处理故障中了解更多有关Envoy的故障处理功能。

服务可以通过使用HTTP 503响应健康检查来主动减轻负担。在这种情况下，服务实例将立即从调用者的负载均衡池中删除。

处理故障

Envoy提供了一套开箱即用，选择加入的故障恢复功能，可以在应用程序中受益。功能包括：

1. 超时
2. 带超时预算有限重试以及重试之间的可变抖动
3. 并发连接数和上游服务请求数限制
4. 对负载均衡池的每个成员进行主动（定期）运行健康检查
5. 细粒度熔断器（被动健康检查）- 适用于负载均衡池中的每个实例

这些功能可以通过Istio的流量管理规则在运行时进行动态配置。

重试之间的抖动使重试对重载的上游服务的影响最小化，而超时预算确保主叫服务在可预测的时间范围内获得响应（成功/失败）。

主动和被动健康检查（上述4和5）的组合最大限度地减少了在负载平衡池中访问不健康实例的机会。当与平台级健康检查（例如由Kubernetes或Mesos支持的检查）相结合时，应用程序可以确保将不健康的pod/container/VM快速地从服务网格中去除，从而最小化请求失败和延迟产生影响。

总之，这些功能使得服务网格能够容忍故障节点，并防止本地化的故障级联不稳定到其他节点。

微调

Istio的流量管理规则允许运维人员为每个服务/版本设置故障恢复的全局默认值。然而，服务的消费者也可以通过特殊的HTTP头提供的请求级别值覆盖超时和重试的默认值。使用Envoy代理实现，header分别是"x-envoy-upstream-rq-timeout-ms"和"x-envoy-max-retries"。

FAQ

1. 在Istio中运行应用程序是否仍需要处理故障？

是。Istio可以提高网格中服务的可靠性和可用性。但是，应用程序仍然需要处理故障（错误）并采取适当的回退操作。例如，当负载均衡池中的所有实例都失败时，Envoy将返回HTTP 503。应用程序有责任实施用于处理来自上游服务的HTTP 503错误代码所需的任何回退逻辑。

2. Envoy的故障恢复功能是否会破坏已经使用容错库（例如Hystrix）的应用程序？

   Envoy对应用程序是完全透明的。由Envoy返回的故障响应不会与进行呼叫的上游服务返回的故障响应区分开来。

3. 同时使用应用级库和Envoy时，怎样处理失败？

   为相同目的地的服务给出两个故障恢复策略（例如，两次超时 - 一个在Envoy中设置，另一个在应用程序库中），当故障发生时，两个限制都将被触发。例如，如果应用程序为服务的API调用设置了5秒的超时时间，而运维人员员配置了10秒的超时时间，那么应用程序的超时将会首先启动。同样，如果特使的熔断器在应用熔断器之前触发，服务的API呼叫将从Envoy获得503。

故障注入

虽然Envoy sidecar/proxy为在Istio上运行的服务提供了大量故障恢复机制，但测试整个应用程序端到端的故障恢复能力依然是必须的。错误配置的故障恢复策略（例如，跨服务调用的不兼容/限制性超时）可能导致应用程序中关键服务持续不可用，从而导致用户体验不佳。

Istio启用协议特定的故障注入到网络中，而不是杀死pod，延迟或在TCP层破坏数据包。我们的理由是，无论网络级别的故障如何，应用层观察到的故障都是一样的，并且可以在应用层注入更有意义的故障（例如，HTTP错误代码），以锻炼应用的弹性。

运维人员可以为符合特定条件的请求配置故障。运维人员可以进一步限制应该遭受故障的请求的百分比。可以注入两种类型的故障：延迟和中止。延迟是计时故障，模拟增加的网络延迟或过载的上游服务。中止是模拟上游服务的崩溃故障。中止通常以HTTP错误代码或TCP连接失败的形式表现。

有关详细信息，请参阅Istio的流量管理规则。

规则配置

Istio提供了简单的领域特定语言（DSL），用来控制应用部署中跨多个服务的API调用和4层流量。DSL允许运维人员配置服务级别的属性，如熔断器，超时，重试，以及设置常见的连续部署任务，如金丝雀推出，A/B测试，基于百分比流量拆分的分阶段推出等。详细信息请参阅路由规则参考。

例如，将“reviews”服务100%的传入流量发送到“v1”版本的简单规则，可以使用规则DSL进行如下描述：

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-default
spec:
  destination:
    name: reviews
  route:
  - labels:
      version: v1
    weight: 100
```

destination是服务的名称，流量将被导向到这里。route *lables*标记将接受流量的特定服务实例。例如，在Istio的Kubernetes部署中，route *lable* "version：v1"指明只有包含label “version: v1”的pod将会接收流量。

可以使用istioctl CLI配置规则，如果部署在Kubernetes中，也可以替代为使用kubectl命令。有关示例，请参阅配置请求路由任务。

在Istio中有三种类型的流量管理规则，**Route Rules/**路由规则，**DestinationPolicies/**目的地策略（这些与Mixer策略不同）和**Egress Rule/**出口规则。所有这三种类型的规则控制请求如何路由到目标服务。

路由规则

路由规则控制如何在Istio服务网格中路由请求。例如，路由规则可以将请求路由到服务的不同版本。可以基于源和目的地，HTTP header字段以及与个别服务版本相关联的权重来路由请求。编写路由规则时，必须牢记以下重要方面：.

用destination修饰规则

每个规则对应某些目的地服务，由规则中的 *destination* 字段标识。例如，应用于"reviews"服务调用的规则典型地将包括至少下面内容。

```
destination:
  name: reviews
```

destination 的值隐式或者显式地指定一个完全限定域名（Fully Qualified Domain Name,FQDN）。Istio Pilot用它来给服务匹配规则。

通常，服务的FQDN由三个部分组成：*name*, *namespace*，和*domain*:

```
FQDN = name + "." + namespace + "." + domain
```

这些字段可以用如下方式显式指定：

```
destination:
  name: reviews
  namespace: default
  domain: svc.cluster.local
```

更普遍的是，为了简化和最大限度重用规则（例如，在多个namespace和domain中使用同一个规则），规则的destination仅仅指定*name*字段，其他两个字段取决于默认值。

*namespace*的默认值是规则自身的namespace，在规则的*metadata*字段中指定，或者在规则安装期间使用 `istioctl -n <namespace> create` 或者 `kubectl -n <namespace> create` 命令指定。*domain*字段的默认值是实现特有的。例如在Kubernetes中，默认值是 `svc.cluster.local` 。

在某些情况下，比如在egress rule中引用到外部服务，或者在某些没所谓*namespace*和*domain*的平台上，可以用*service*字段替代，显式指定destination：

```
destination:
  service: my-service.com
```

当*service*字段被指定时，其他字段的所有其他隐式或者显式的值都被忽略。

通过source/headers修饰规则

规则可以选择性的修饰为仅适用于符合以下特定条件的请求：

1. 限制为特定的调用者

   例如，以下规则仅适用于来自"reviews"服务的调用。

   ```
   apiVersion: config.istio.io/v1alpha2
   kind: RouteRule
   metadata:
     name: reviews-to-ratings
   spec:
     destination:
       name: ratings
     match:
       source:
         name: reviews
     ...
   ```

   source 的值，和 *destination* 一样，指定服务的FQDN，无论是隐式还是显式。

2. 限制为调用者的特定版本

   例如，以下规则将细化上一个示例，仅适用于"reviews"服务的"v2"版本的调用。

```
  apiVersion: config.istio.io/v1alpha2
  kind: RouteRule
  metadata:
    name: reviews-v2-to-ratings
  spec:
    destination:
      name: ratings
    match:
      source:
        name: reviews
        labels:
          version: v2
    ...
```

3. 选择基于*HTTP header*的规则

   例如，以下规则仅适用于传入请求，如果它包含"cookie" header, 并且内容包含"user=jason"。

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-jason
spec:
  destination:
    name: reviews
  match:
    request:
      headers:
        cookie:
          regex: "^(.*?;)?(user=jason)(;.*)?$"
  ...
```

如果提供了多个属性值对，则所有相应的 header 必须与要应用的规则相匹配。

可以同时设置多个标准。在这种情况下，AND语义适用。例如，以下规则仅适用于请求的source为"reviews:v2"，并且存在包含"user=jason"的"cookie" header。

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-reviews-jason
spec:
  destination:
    name: ratings
  match:
    source:
      name: reviews
      labels:
        version: v2
    request:
      headers:
        cookie:
          regex: "^(.*?;)?(user=jason)(;.*)?$"
  ...
```

在服务版本之间拆分流量

当规则被激活时，每个路由规则标识一个或多个要调用的加权后端。每个后端对应于目标服务的特定版本，其中版本可以使用 *labels* 表示。

如果有多个具有指定tag的注册实例，则将根据为该服务配置的负载均衡策略来路由，或默认轮循。

例如，以下规则会将"reviews"服务的25%的流量路由到具有"v2"标签的实例，其余流量（即75%）转发到"v1"。

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-v2-rollout
spec:
  destination:
    name: reviews
  route:
  - labels:
      version: v2
    weight: 25
  - labels:
      version: v1
    weight: 75
```

超时和重试

缺省情况下，http请求的超时时间为15秒，但可以在路由规则中覆盖，如下所示：

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-timeout
spec:
  destination:
    name: ratings
  route:
  - labels:
      version: v1
  httpReqTimeout:
    simpleTimeout:
      timeout: 10s
```

对于给定的http请求，重试次数可以也可以在路由规则中指定。

可以如下设置最大尝试次数，或者在超时期限内的尽可能多，其中超时时间可以是默认或被覆盖：

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-retry
spec:
  destination:
    name: ratings
  route:
  - labels:
      version: v1
  httpReqRetries:
    simpleRetry:
      attempts: 3
```

请注意，请求超时和重试也可以根据每个请求重写。

请参阅请求超时任务 以演示超时控制。

在请求路径中注入故障

在将http请求转发到规则的相应请求目的地时，路由规则可以指定一个或多个要注入的故障。故障可能是延迟或中断。

以下示例将在"reviews"微服务的"v1"版本的10%的请求中引入5秒的延迟。

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-delay
spec:
  destination:
    name: reviews
  route:
  - labels:
      version: v1
  httpFault:
    delay:
      percent: 10
      fixedDelay: 5s
```

另一种故障，中断，可以用来提前终止请求，例如模拟故障。

以下示例将为"ratings"服务"v1"的10%请求返回HTTP 400错误代码。

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-abort
spec:
   destination:
     name: ratings
   route:
   - labels:
       version: v1
   httpFault:
     abort:
       percent: 10
       httpStatus: 400
```

有时延迟和中止故障会一起使用。例如，以下规则将所有从"reviews"服务"v2"到"ratings"服务"v1"的请求延迟5秒钟，然后中止其中的10%：

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-delay-abort
spec:
  destination:
    name: ratings
  match:
    source:
      name: reviews
      labels:
        version: v2
  route:
  - labels:
      version: v1
  httpFault:
    delay:
      fixedDelay: 5s
    abort:
      percent: 10
      httpStatus: 400
```

要查看故障注入的实际使用，请参阅故障注入任务。

规则有优先权

多个路由规则可以应用于同一目的地。当有多个规则时，可以指定规则的评估顺序。这些规则与给定目的地相对应的，通过规则的*precedence*字段来设置。

```
destination:
  name: reviews
precedence: 1
```

precedence字段是可选的整数值，默认为0。首先评估具有较高优先级值的规则。如果有多个具有相同优先级值的规则，则评估顺序是未定义的。

优先级什么时候有用？只要特定服务的路由故障纯粹是基于权重的，可以在单个规则中指定，如前面的示例所示。另一方面，当正在使用的其他标准（例如，来自特定用户的请求）来路由流量时，将需要多于一个的规则来指定路由。这是必须设置规则*precedence*字段的时候，以确保以正确的顺序对规则进行评估。

一般路由规范的通用模式是提供一个或多个较高优先级的规则，通过到特定destination的source/header来修饰规则，然后提供单个基于权重的规则，连最低优先级的匹配准则都不具备，以在所有其他情况下，提供流量的加权分发。

例如，以下2条规则一起指定，在"reviews"服务的所有请求中，如果包含名为"Foo"值为"bar"的header，则都将发送到"v2"实例。所有其他的请求将被发送到"v1"。

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-foo-bar
spec:
  destination:
    name: reviews
  precedence: 2
  match:
    request:
      headers:
        Foo: bar
  route:
  - labels:
      version: v2
---
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-default
spec:
  destination:
    name: reviews
  precedence: 1
  route:
  - labels:
      version: v1
    weight: 100
```

请注意，基于header的规则具有较高的优先级（2对1）。如果它较低，这些规则将无法正常工作，因为基于权重的规则（没有特定匹配条件）将首先被评估，然后将简单地将所有流量路由到“v1”，即使是包括匹配“Foo” header的请求。一旦找到适用于传入请求的规则，它将被执行，并且规则评估过程将终止。这就是为什么当有不止一个规则时，仔细考虑每个规则的优先级是非常重要的。

目的地策略

目的地策略描述与特定服务版本相关联的各种路由相关策略，例如负载均衡算法，熔断器配置，健康检查等。

与路由规则不同，目的地策略不能根据请求的属性进行修饰，但是，可以限制策略适用于某些请求，这些请求是使用特定标签来路由到destination后端的。例如，以下负载均衡策略仅适用于针对"reviews"微服务器的"v1"版本的请求。

```
apiVersion: config.istio.io/v1alpha2
metadata:
  name: ratings-lb-policy
spec:
  source:
    name: reviews
    labels:
      version: v2
  destination:
    name: ratings
    labels:
      version: v1
  loadBalancing:
    name: ROUND_ROBIN
```

熔断器

可以根据诸如连接和请求限制的多个标准来设置简单的熔断器。

例如，以下目的地策略设置到"reviews"服务版本"v1"后端的100个连接的限制。

```
apiVersion: config.istio.io/v1alpha2
metadata:
  name: reviews-v1-cb
spec:
  destination:
    name: reviews
    labels:
      version: v1
  circuitBreaker:
    simpleCb:
       maxConnections: 100
```

这里可以找到一整套简单的熔断器字段。

目的地策略评估

类似于路由规则，目的地策略与特定目的地相关联，但是如果它们还包括标签，则其激活取决于路由规则评估结果。

规则评估过程的第一步将评估目的地的路由规则（如果有定义），以确定当前请求将路由到的目标服务的标签（例如，特定版本）。下一步，评估目的地策略集,如果有，以确定它们是否适用。

注意：算法要注意的一个微妙之处在于，仅当对应的已标记实例被明确路由时，才会应用为特定标记目标定义的策略。例如，考虑以下规则，作为"reviews"服务的唯一规则。

```
apiVersion: config.istio.io/v1alpha2
metadata:
  name: reviews-v1-cb
spec:
  destination:
    name: reviews
    labels:
      version: v1
  circuitBreaker:
    simpleCb:
      maxConnections: 100
```

由于没有为"reviews"服务定义特定的路由规则，因此默认进行轮循路由，这有可能偶尔调用“v1”实例，如果“v1”是唯一运行的版本甚至会一致调用。然而，上述策略将永远不会被调用，因为默认路由在较低级别完成。规则评估引擎将不知道最终目的地，因此无法将目标策略与请求相匹配。

您可以通过以下两种方式之一修复上述示例。您可以从规则中删除 `tags:` ，如果“v1”是唯一的实例，或者更好地，为服务定义适当的路由规则。例如，您可以为"reviews:v1"添加一个简单的路由规则。

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-default
spec:
  destination:
    name: reviews
  route:
  - labels:
      version: v1
```

虽然默认的Istio行为可以很方便地将流量从源服务的所有版本发送到目标服务的所有版本，而不用设置任何规则，一旦需要版本区别，将需要规则。因此，从一开始就为每个服务设置默认规则通常被认为是Istio的最佳实践。

出站规则

出站规则用于配置需要在服务网格中被调用的外部服务。例如下面的规则可以被用来配置在 `*.foo.com` 域名下的外部服务。

```
apiVersion: config.istio.io/v1alpha2
kind: EgressRule
metadata:
  name: foo-egress-rule
spec:
  destination:
    service: *.foo.com
  ports:
    - port: 80
      protocol: http
    - port: 443
      protocol: https
```

外部服务的地址通过*service*字段指定,该字段可以是一个完全限定域名（Fully Qualified Domain Name,FQDN），也可以是一个带通配符的域名。该域名代表了可在服务网格中访问的外部服务的一个白名单，该白名单中包括一个(字段值为完全限定域名的情况)或多个外部服务(字段值为带通配符的域名的情况)。这里可以找到*service*字段支持的域名通配符格式。

目前Istio在服务网格内只支持通过HTTP协议访问外部服务。然而边车（sidecar）和外部服务之间的通信可以是基于TLS的。如上面的例子所示，通过把*protocol*字段设置为"https",边车（sidecar）就可以通过TLS和外部服务进行通信。此时，服务网格内的应用只能通过HTTP协议对外部服务进行调用（例如，使用 `http://secure-service.foo.com:443` ，而不是 `https://secure-service.foo.com` 来访问外部服务），然而边车（sidecar）在向外部服务转发该请求时会采用TLS。

只要这些规则采用相同的目的地配置，以指向相同的外部服务，出站规则可以很好地和路由规则及目的地策略协同工作。例如下面的规则可以和前面示例中的出站规则一起作用，将这些外部服务的调用超时设置为10秒。

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: foo-timeout-rule
spec:
  destination:
    service: *.foo.com
  httpReqTimeout:
    simpleTimeout:
      timeout: 10s
```

目的地策略和路由规则的流量重定向和前转，重试，超时，故障注入策略等特性都可以很好地支持外部服务。但是由于外部服务没有多版本的概念，和服务版本关联的按权重的路由规则是不支持的。

安全

描述Istio的授权和认证功能。

双向TLS认证。描述Istio的双向TLS认证架构，这个架构提供强服务身份和服务间加密通讯通道。

双向TLS认证

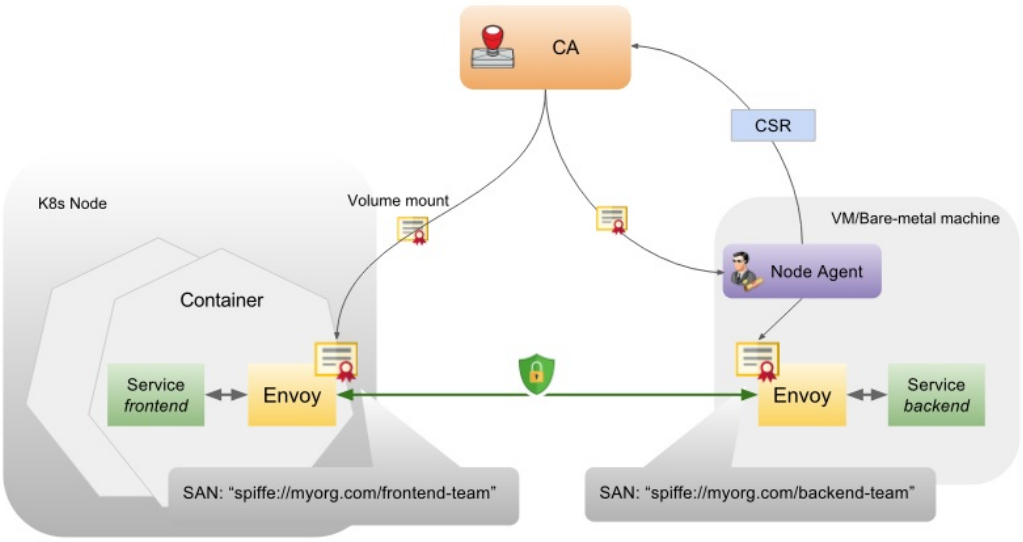
概述

Istio Auth的目标是提高微服务及其通信的安全性，而不需要修改服务代码。它负责：

- 为每个服务提供强大的身份，代表其角色，以实现跨集群和云的互通性
- 加密服务到服务的通讯
- 提供密钥管理系统来自动执行密钥和证书的生成，分发，轮换和撤销
- 即将提供的功能：
    - 强大的认证机制：ABAC, RBAC, Authorization hooks.
    - 终端用户认证
    - 证书和身份可插拔

架构

下图展示Istio Auth架构，其中包括三个主要组件：身份，密钥管理和通信安全。它描述了Istio Auth如何用于加密服务间通信，在以服务帐户“foo”运行的服务A，和以服务帐户“bar”运行的服务B之间。



如图所示，Istio Auth利用secret volume mount，从Istio CA向Kubernetes容器传递keys/certs。对于在VM/裸机上运行的服务，我们引入了节点代理，它是在每个VM/裸机上运行的进程。它在本地生成私钥和CSR（证书签名请求），将CSR发送给Istio CA进行签名，并将生成的证书与私钥一起交给Envoy。

组件

身份

在Kubernetes上运行时，Istio Auth使用Kubernetes service account 来识别谁在运行服务：

- Istio中的service account格式为 `spiffe://\<_domain_\>/ns/\<_namespace_>/sa/\<_serviceaccount_\>`
    - *domain*目前是*cluster.local*. 我们将很快支持域的定制化。
    - *namespace* 是Kubernetes service account的namespace.
    - *serviceaccount*是Kubernetes service account name
- service account是工作负载运行的身份（或角色），表示该工作负载的权限。对于需要强大安全性的系统，工作负载的特权量不应由随机字符串（如服务名称，标签等）或部署的二进制文件来标识。
    - 例如，假设我们有一个从多租户数据库中提取数据的工作负载。如果Alice

运行这个工作负载，她将能够提取的数据，和Bob运行这个工作负载时不同。

- service account通过提供灵活性来识别机器，用户，工作负载或一组工作负载（不同的工作负载可以以同一service account运行）来实现强大的安全策略。
- 工作负载运行的service account将不会在工作负载的生命周期内更改。
- 可以通过域名约束确保service account的唯一性

通讯安全

服务到服务通信通过客户端Envoy和服务器端Envoy进行隧道传送。端到端通信通过以下方式加密：

- 服务与Envoy之间的本地TCP连接
- 代理之间的双向TLS连接
- 安全命名：在握手过程中，客户端Envoy检查服务器端证书提供的服务帐号是否允许运行目标服务

密钥管理

Istio v0.2支持服务运行于Kubernetes pod和VM/裸机。对于每个场景，我们使用不同的秘钥配置机制。

在于运行在Kubernetes pod上的服务，每个集群的Istio CA（证书颁发机构）自动化执行密钥和证书管理流程。它主要执行四个关键操作：

- 为每个service account生成一个 SPIFFE 密钥和证书对
- 根据service account将密钥和证书对分发给每个pod
- 定期轮换密钥和证书
- 必要时撤销特定的密钥和证书对

对于运行在VM/裸机上的服务，上述四个操作通过Istio CA用节点代理一起执行。

工作流

Istio Auth工作流由两个阶段组成，部署和运行时。对于部署阶段，我们分别讨论这两个场景（例如，在Kubernetes中和VM/裸机），因为他们不一样。一旦秘钥和证书部署好，对于两个场景运行时阶段是相同的。在这节中我们简要介绍工作流。

部署阶段(Kubernetes场景)

1. Istio CA观察Kubernetes API Server，为每个现有和新的service account创建一个 SPIFFE 密钥和证书对，并将其发送到API服务器。
2. 当创建pod时，API Server会根据service account使用Kubernetes secrets 来挂载密钥和证书对。
3. Pilot 使用适当的密钥和证书以及安全命名信息生成配置，该信息定义了什么服务帐户可以运行某个服务，并将其传递给Envoy。

部署阶段(VM/裸机场景)

1. Istio CA创建gRPC服务来处理CSR请求。
2. 节点代理创建私钥和CSR, 发送CSR到Istio CA用于签名。
3. Istio CA验证CSR中携带的证书，并签名CSR以生成证书。
4. 节点代理将从CA接收到的证书和私钥发送给Envoy。
5. 为了轮换，上述CSR流程定期重复。

运行时阶段

1. 来自客户端服务的出站流量被重新路由到它本地的Envoy。
2. 客户端Envoy与服务器端Envoy开始相互TLS握手。在握手期间，它还进行安全的命名检查，以验证服务器证书中显示的服务帐户是否可以运行服务器服务。
3. mTLS连接建立后，流量将转发到服务器端Envoy，然后通过本地TCP连接转发到服务器服务。

最佳实践

在本节中，我们提供了一些部署指南，然后讨论了一个现实世界的场景。

部署指南

- 如果有多个服务运维人员（也称为SRE）在集群中部署不同的服务（通常在中型或大型集群中），我们建议为每个SRE团队创建一个单独的namespace，以隔离其访问。例如，您可以为team1创建一个"team1-ns"命名空间，为team2创建"team2-ns"命名空间，这样两个团队就无法访问对方的服务。
- 如果Istio CA受到威胁，则可能会暴露集群中被它管理的所有密钥和证书。我们强烈建议在专门的命名空间（例如istio-ca-ns）上运行Istio CA，只有集群管理员才能访问它。

示例

我们考虑一个三层应用程序，其中有三个服务：照片前端，照片后端和数据存储。照片前端和照片后端服务由照片SRE团队管理，而数据存储服务由数据存储SRE团队管理。照片前端可以访问照片后端，照片后端可以访问数据存储。但是，照片前端无法访问数据存储。

在这种情况下，集群管理员创建3个命名空间：istio-ca-ns，photo-ns和datastore-ns。管理员可以访问所有命名空间，每个团队只能访问自己的命名空间。照片SRE团队创建了2个服务帐户，以分别在命名空间photo-ns中运行照片前端和照片后端。数据存储SRE团队创建1个服务帐户以在命名空间datastore-ns中运行数据存储服务。此外，我们需要在 Istio Mixer 中强制执行服务访问控制，以使照片前端无法访问数据存储。

在此设置中，Istio CA能够为所有命名空间提供密钥和证书管理，并隔离彼此的微服务部署。

未来的工作

- 集群间服务到服务认证
- 强大的认证机制：ABAC, RBAC等
- 每服务认证启用的支持

- 安全Istio组件（Mixer, Pilot等）
- 使用JWT/OAuth2/OpenID_Connect 的终端用户到服务的认证
- 支持GCP service account
- 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

策略与控制

介绍策略控制机制.

- 属性: 解释属性的重要概念，这是将策略和控制应用于网格中的服务的中心机制。
- Mixer: Mixer设计的深层架构，提供服务网格内的策略和控制机制。
- Mixer配置: 用于配置Mixer的关键概念的概述。

属性

本节阐述了Istio的概念和使用方法。

背景

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。属性是具有名称和类型的元数据片段，用以描述入口和出口流量，以及这些流量所属的环境。Istio属性携带特定信息片段，例如API请求的错误代码，API请求的延迟或TCP连接的原始IP地址。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
```

属性词汇表

给定的Istio部署有一个它可以理解的固定的属性词汇表。具体词汇表由部署中使用的属性生产者集合决定。Istio的主要属性生产者是Envoy，尽管专业的Mixer适配器和服务也可以生成属性。

这里定义了大多数Istio部署中可用的常用基准属性集。

属性名

Istio属性使用类似Java的完全限定标识符作为属性名。允许的字符是 `[_.a-z0-9]` 。该字符 `"."` 用作命名空间分隔符。例如， `request.size` 和 `source.ip` 。

属性类型

Istio属性是强类型的。支持的属性类型由 ValueType 定义。

Mixer

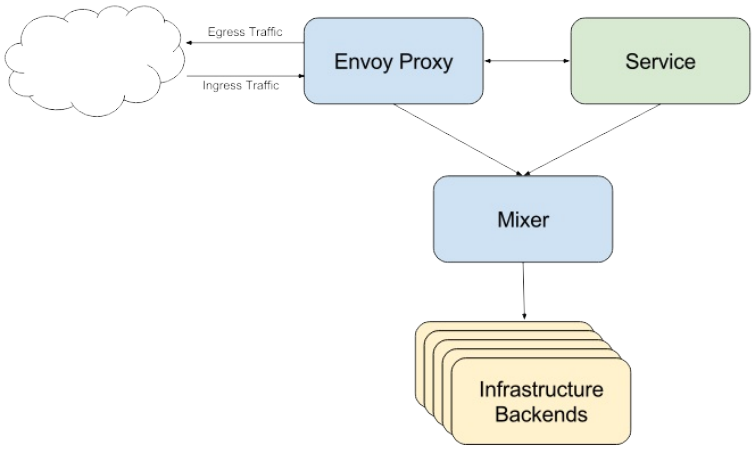
本节解释Mixer的角色和总体架构。

背景

基础设施的后端被设计用于提供建立服务支持功能。它们包括访问控制系统，遥测捕获系统，配额执行系统，计费系统等。传统服务会直接和这些后端系统打交道，和后端紧密耦合，并集成其中的个性化语义以及用法。

Mixer在应用程序代码和基础架构后端之间提供通用中介层。它的设计将策略决策移出应用层，用运维人员能够控制的配置取而代之。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。

Mixer的设计目的是改变层次之间的边界，以此来降低总体的复杂性。从服务代码中剔除策略逻辑，改由运维人员进行控制。



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。前提条件可以包括服务使用者是否被正确认证，是否在服务的白名单上，是否通过ACL检查等等。
- 配额管理。 使服务能够在分配和释放多个维度上的配额，配额这一简单的资源管理工具可以在服务消费者对有限资源发生争用时，提供相对公平的（竞争手段）。频率控制就是配额的一个实例。

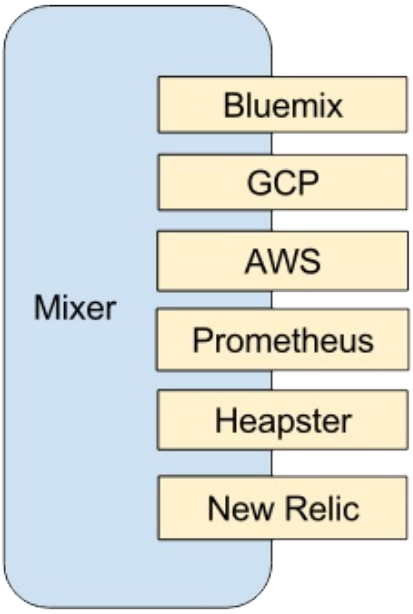
- 遥测报告。使服务能够上报日志和监控。在未来，它还将启用针对服务运营商以及服务消费者的跟踪和计费流。

这些机制的应用是基于一组属性的，每个请求都会将这些属性呈现给Mixer。在Istio中，这些属性来自于Sidecar代理（一般是Envoy）的每一次请求。

适配器

Mixer是高度模块化和可扩展的组件。他的一个关键功能就是把不同后端的策略和遥测系统的细节抽象出来，让Envoy以及基于Istio的服务能够独立于这些后端，从而保持他们的可移植性。

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能够暴露一个一致的独立于后端的API。通过配置能够决定在运行时使用的确切的适配器套件，并且可以轻松指向新的或定制的基础设施后端。



配置状态

Mixer的核心运行时方法（ `Check` 和 `Report` ）都接受来自输入的一组属性。Mixer的当前配置会根据输入属性来决定每个方法的工作内容。为此，服务运维工作包括：

- 配置一组*handelr*。Handler是配置完成的Adapter（Adapter是一种二进制插件，下面会提到）。给 `statsd` Adapter设置一个statsd后端的IP地址，就是一个Handler配置的例子。
- 基于属性和常量，为Mixer配置一组*Instance*，Instance表达了一套提供给Adapter处理的数据。例如通过配置让Mixer使用 `destination.service` 和 `response.code` 生成 `request_code` 指标。
- 配置一套*Rule*，Mixer在每次请求时都会执行这些Rule。Rule由一个匹配表达式和对应的Action构成。Mixer根据匹配表达式来判断将要执行的Action。Action中设置了需要生成的Instance和处理这些Instance所需要的Handler。例如一个Rule要求Mixer在所有的 `report` 过程中，把 `requestcount` instance发送给 `statsd` Handler。

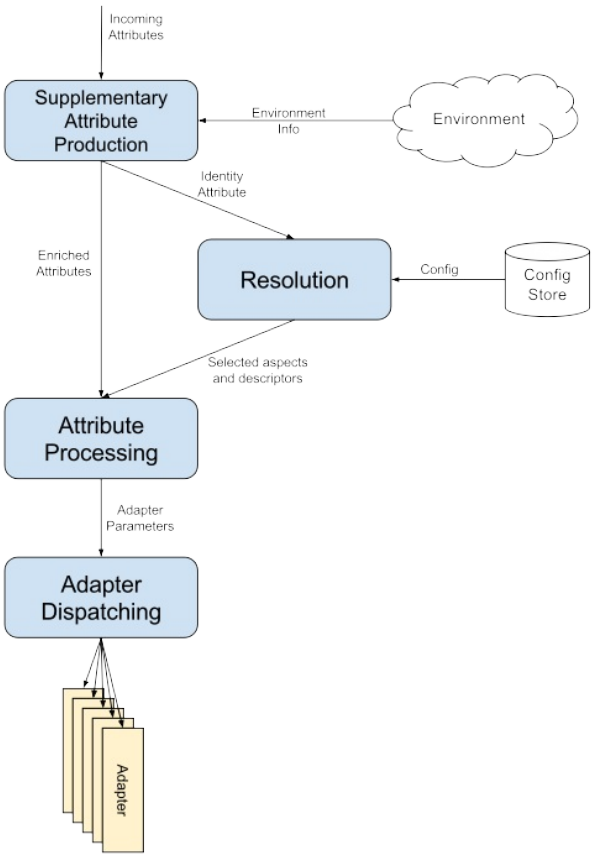
需要上述配置状态才能让Mixer知道如何处理传入的属性并分发到适当的基础设置后端。

有关Mixer配置模型的详细信息，请参阅 此处。

请求阶段

当一个请求进入Mixer时，它会经历一些不同的处理阶段：

- 生成补充属性。在Mixer中发生的第一件事是运行一组全局配置的适配器，这些适配器负责引入新的属性。这些属性与来自请求的属性组合，以形成操作的全部属性集合。
- 决议。第二阶段是评估属性集，以确定应用于请求的有效配置。请参阅此处了解这一阶段的的工作原理。有效的配置确定可用于在后续阶段处理请求的一组切面和描述符。
- 属性处理。第三阶段拿到属性总集，然后产生一组适配器参数。这里描述了使用简单的声明来初始化属性处理的过程。
- 适配器调度。决议阶段建立可用切面的集合，而属性处理阶段创建一组适配器参数。适配器调度阶段调用与每个切面相关联的适配器，并传递这些参数给它们。



下一步

阅读博客内容，理解Mixer的适配器模型。

Mixer配置

本节介绍Mixer的配置模型。

背景

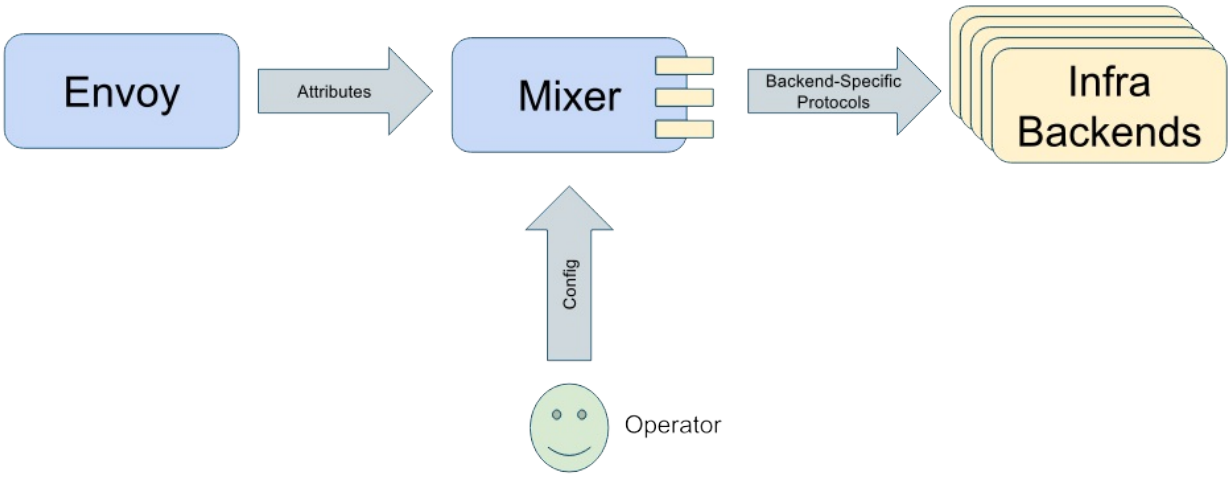
Istio是一个具有数百个独立功能的复杂系统。Istio部署可能涉及数十个服务的蔓延事件，这些服务有一群Envoy代理和Mixer实例来支持它们。在大型部署中，许多不同的运维人员（每个运维人员都有不同的范围和责任范围）可能涉及管理整体部署。

Mixer的配置模式可以利用其所有功能和灵活性，同时保持使用的相对简单。该模型的范围特征使大型支持组织能够轻松地集中管理复杂的部署。该模型的一些主要功能包括：

- 专为运维人员而设计：服务运维人员通过操纵配置资源来控制Mixer部署中的所有操作和策略切面。
- 灵活：配置模型围绕Istio的属性构建，使运维人员能够对部署中使用的策略和生成的遥测进行前所未有的控制。
- 健壮：配置模型旨在提供最大的静态正确性保证，以帮助减少因为错误的配置变更导致的服务中断事故。
- 扩展：该模型旨在支持Istio的整体可扩展性思路。可以将新的或自定义的适配器添加到Istio中，并可以使用与现有适配器相同的通用机制进行完全操作。

概念

Mixer是一种属性处理机器。请求到达Mixer时带有一组属性 ，并且基于这些属性，Mixer会生成对各种基础设施后端的调用。这些后端包括频率限制、访问控制、策略实施等各种系统。该属性集确定Mixer为给定的请求用哪些参数调用哪个后端。为了隐藏各个后端的细节，Mixer使用称为适配器的模块。



Mixer的配置有几个中心职责：

- 描述哪些适配器正在使用以及它们的操作方式。
- 描述如何将请求属性映射到适配器参数中。
- 描述使用特定参数调用适配器的时机。

配置基于适配器和模板来完成：

- 适配器 封装了Mixer和特定基础设施后端之间的接口。
- 模板 定义了从特定请求的属性到适配器输入的映射关系。一个适配器可以支持任意数量的模板。

配置使用YAML格式来表示,围绕几个核心抽象构建：

概念	描述
Handler	Handlers就是一个配置完成的适配器。适配器的构造器参数就是Handler的配置。
实例	一个（请求）实例就是请求属性到一个模板的映射结果。这种映射来自于实例的配置。
规则	规则确定了何时使用一个特定的模板配置来调用一个Handler。

配置的资源对象可以使用Kubernetes的资源语法来进行表述：

```
apiVersion: config.istio.io/v1alpha2
kind: rule, adapter kind, or template kind
metadata:
  name: shortname
  namespace: istio-system
spec:
  # 不同kind会有各自特定的配置
```

- **apiVersion**：常量，取决于Istio的版本
- **kind**：Mixer中的适配器和模板都有对应的类型。
- **name**：配置资源名称。
- **namespace**：配置所在的命名空间。
- **spec**：`kind`所对应的配置。

Handler

适配器封装了Mixer和特定外部基础设施后端进行交互的必要接口，例如Prometheus、New Relic或者Stackdriver。各种适配器都需要参数配置才能工作。例如日志适配器可能需要IP地址和端口来进行日志的输出。

这里的例子配置了一个类型为`listchecker`的适配器。listchecker适配器使用一个列表来检查输入。如果配置的是白名单模式且输入值存在于列表之中，就会返回成功的结果。

```
apiVersion: config.istio.io/v1alpha2
kind: listchecker
metadata:
  name: staticversion
  namespace: istio-system
spec:
  providerUrl: http://white_list_registry/
  blacklist: false
```

`{metadata.name}.{kind}.{metadata.namespace}` 是Handler的完全限定名。上面定义的对象的FQDN就是 `staticversion.listchecker.istio-system` ，他必须是唯一的。 `spec` 中的数据结构则依赖于对应的适配器的要求。

有些适配器实现的功能就不仅仅是把Mixer和后端连接起来。例如 `prometheus` 适配器使用一种可配置的方式，消费指标并对其进行聚合：

```
apiVersion: config.istio.io/v1alpha2
kind: prometheus
metadata:
  name: handler
  namespace: istio-system
spec:
  metrics:
  - name: request_count
    instance_name: requestcount.metric.istio-system
    kind: COUNTER
    label_names:
    - destination_service
    - destination_version
    - response_code
  - name: request_duration
    instance_name: requestduration.metric.istio-system
    kind: DISTRIBUTION
    label_names:
    - destination_service
    - destination_version
    - response_code
    buckets:
      explicit_buckets:
        bounds: [0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1, 2.5
, 5, 10]
```

每个适配器都定义了自己格式的配置数据。适配器及其配置的详尽列表可以在这里找到。

实例

配置实例将请求中的属性映射成为适配器的输入。下面的例子，是一个指标实例的配置，用于生成 `requestduration` 指标：

```
apiVersion: config.istio.io/v1alpha2
kind: metric
metadata:
  name: requestduration
  namespace: istio-system
spec:
  value: response.duration | "0ms"
  dimensions:
    destination_service: destination.service | "unknown"
    destination_version: destination.labels["version"] | "unknow
n"
    response_code: response.code | 200
  monitored_resource_type: '"UNSPECIFIED"'
```

注意Handler配置中需要的所有维度都定义在这一映射之中。

每个模板都有自己格式的配置数据。完整的模板及其特定配置格式可以在这里查阅。

规则

规则用于指定使用特定实例配置调用某一Handler的时机。比如我们想要把 `service1` 服务中，请求头中带有 `x-user` 的请求的 `requestduration` 指标发送给Prometheus Handler:

```
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  name: promhttp
  namespace: istio-system
spec:
  match: destination.service == "service1.ns.svc.cluster.local"
&& request.headers["x-user"] == "user1"
  actions:
  - handler: handler.prometheus
    instances:
    - requestduration.metric.istio-system
```

规则对象中包含有一个 `match` 元素，用于前置检查，如果检查通过则会执行动作列表。动作中包含了一个实例列表，这个列表将会分发给Handler。规则必须使用Handler和实例的完全限定名。如果规则、Handler以及实例全都在同一个命名空间，命名空间后缀就可以在FQDN中省略，例如 `handler.prometheus` 。

匹配检查使用的是属性表达式，下面将会讲解。

属性表达式

Mixer具有多个独立的请求处理阶段 。属性处理阶段 负责摄取一组属性，并产生调用各个适配器时需要的模板的实例。该阶段通过评估一系列的 属性表达式 来运行。

在前面的例子中，我们已经看到了一些简单的属性表达式。特别是：

```
  destination_service: destination.service
  response_code: response.code
  destination_version: destination.labels["version"] | "unknown"
```

冒号右侧的序列是属性表达式的最简单形式。前两行只包括了属性名称。 `response_code` 标签的内容来自于 `request.code` 属性。

以下是条件表达式的示例：

```
destination_version: destination.labels["version"] | "unknown"
```

上面的表达式里，`destination_version` 标签被赋值为 `destination.labels["version"]` ，如果 `destination.labels["version"]` 为空，则使用 `"unknown"` 代替。

在属性表达式中可用的属性必须符合该部署中的属性清单。在清单中，每个属性都有一个用于描述属性所代表数据的数据类型。同样的，属性表达式也是有类型的，表达式中的属性类型以对这些属性的操作决定了表达式的数据类型。

有关详细信息，请参阅属性表达式引用。

决议

当请求到达时，Mixer会经过多个请求处理阶段。决议阶段涉及确定要用于处理传入请求的确切配置块。例如，到达 Mixer 的 service A 的请求可能与 service B 的请求有一些配置差异。决议决定哪个配置用于请求。

决议依赖众所周知的属性来指导其选择，即所谓的身份属性。该属性的值是一个点号分隔的名称，它决定了Mixer在层次结构中从哪里开始查找用于请求的配置块。

这是它的工作原理：

1. 请求到达, Mixer提取身份属性的值以产生当前的查找值。
2. Mixer查找主题与查找值匹配的所有配置块。
3. 如果Mixer找到多个匹配的块，则它只保留具有最大范围的块。
4. Mixer从查找值的点号分割名称中截断最低元素。如果查找值不为空，则Mixer将返回上述步骤2。

在此过程中发现的所有块都组合在一起，形成用于最终的有效配置评估当前的请求。

清单

清单捕获特定Istio部署中涉及到的组件的不变量。当前唯一支持的清单是属性清单，用于定义由各个组件生成的属性的精确集合。清单由组件生成器提供并插入到部署的配置中。

以下是Istio代理的清单的一部分：

```
manifests:
  - name: istio-proxy
    revision: "1"
    attributes:
      source.name:
        valueType: STRING
        description: The name of the source.
      target.name:
        valueType: STRING
        description: The name of the target
      source.ip:
        valueType: IP_ADDRESS
        description: Did you know that descriptions are optional
?
      origin.user:
        valueType: STRING
      request.time:
        valueType: TIMESTAMP
      request.method:
        valueType: STRING
      response.code:
        valueType: INT64
```

例子

您可以通过访问指南找到Mixer配置的完整示例。这里有一些示例配置。

下一步

- 阅读阐述Mixer适配器模型的博客文章

安装

在不同的环境下（如 Kubernetes、Consul 等）安装 Istio 控制平面，以及在应用程序部署中安装 sidecar。

- Kubernetes
  - 快速开始：在 kubernetes 集群中快速安装 Istio service mesh 的说明。
  - 安装 Istio sidecar：使用Istio初始化工具或者使用 istioctl 命令行工具在应用程序的 pod 中安装 Istio sidecar 的说明。
  - 拓展 Istio Mesh：将虚拟机或裸机集成到安装在 kubernetes 集群上的 Istio mesh 中的说明。
- Nomad 和 Consul
  - 使用 Docker 快速开始：使用 Docker Compose 快速安装 Istio service mesh 指南。
  - 安装：在基于 Consul 的环境中安装 Istio 控制平面，不论是否使用 Nomad。
- Eureka
  - 使用 Docker 快速开始：使用 Docker Compose 快速安装 Istio service mesh 指南。
  - 安装：如何在基于 Eureka 的环境中安装 Isto 控制平面的说明。
- Cloud Foundry
  - 安装
- Mesos
  - 安装

Kubernetes

关于如何在 Kubernetes 集群中安装 Istio 控制平面和添加虚拟机到 mesh 中的说明。

- 快速开始：在 kubernetes 集群中快速安装 Istio service mesh 的说明。
- 安装 Istio sidecar：使用Istio初始化工具或者使用 istioctl 命令行工具在应用程序的 pod 中安装 Istio sidecar 的说明。
- 拓展 Istio Mesh：将虚拟机或裸机集成到部署在 kubernetes 集群上的 Istio mesh 中的说明。

快速开始

前置条件

下面的操作说明需要您可以访问 kubernetes **1.7.3** 后更高版本 的集群，并且启用了 RBAC (基于角色的访问控制)。您需要安装了 **1.7.3** 或更高版本 的 `kubectl` 命令。如果您希望启用 自动注入 sidecar，您需要启用 kubernetes 集群的 alpha 功能。

> 注意：如果您安装了 Istio 0.1.x，在安装新版本前请先 卸载 它们（包括已启用 Istio 应用程序 Pod 中的 sidecar）。

- 安装或更新 kubernetes 命令行工具 kubectl 以匹配集群的版本（1.7 或者更高，支持CRD功能）
- 取决于您的 kubernetes 提供商：
  - 本地安装 Istio，安装最新版本的 Minikube (version 0.22.1 或者更高)。
  - Google Container Engine
    - 使用 kubectl 获取证书（使用您自己的集群的名字替换 `<cluster-name>` ，使用集群实际所在的位置替换 `<zone>` ）：

      ```
      gcloud container clusters get-credentials <cluster-name> --zone <zone> --project <project-name>
      ```

    - 将集群管理员权限授予当前用户（需要管理员权限才能为Istio创建必要的RBAC规则）：

      ```
      kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin --user=$(gcloud config get-value core/account)
      ```

  - IBM Bluemix Container Service
    - 使用 kubectl 获取证书（使用您自己的集群的名字替换 `<cluster-`

name> ）：

```
$(bx cs cluster-config <cluster-name>|grep "export KU
BECONFIG")
```

- IBM Cloud Private 版本 2.1 或者更高：
  - 配置 `kubectl` 用以访问基于IBM Cloud的私有集群的步骤可以参考这里。
- Openshift Origin 3.7 或者以上版本：
  - 默认情况下，Openshift 不允许以 UID 0运行容器。为 Istio 的入口（ingress）以及附加组件Prometheus 和 Grafana 的 service account 启用使用UID 0运行的容器：

```
oc adm policy add-scc-to-user anyuid -z istio-ingress
-service-account -n istio-system
oc adm policy add-scc-to-user anyuid -z istio-grafana
-service-account -n istio-system
oc adm policy add-scc-to-user anyuid -z istio-prometh
eus-service-account -n istio-system
```

  - 运行应用程序 Pod 的 service account 需要特权安全性上下文限制，以此作为 sidecar 注入的一部分:

```
oc adm policy add-scc-to-user privileged -z default -
n <target-namespace>
```

安装步骤

从 0.2 版本开始，Istio 安装到 `istio-system` namespace 下，即可以管理所有其它 namespace 下的微服务。

1. 到 Istio release 页面上，根据您的操作系统下载对应的发行版。如果您使用的是 MacOS 或者 Linux 系统，可以使用下面的额命令自动下载和解压最新的发行版：

```
curl -L https://git.io/getLatestIstio | sh -
```

2. 解压安装文件，切换到文件所在目录。安装文件目录下包含：
   - `install/` 目录下是 kubernetes 使用的 `.yaml` 安装文件
   - `samples/` 目录下是示例程序
   - `istioctl` 客户端二进制文件在 `bin` 目录下。`istioctl` 文件用户手动注入 Envoy sidecar 代理、创建路由和策略等。
   - `istio.VERSION` 配置文件
3. 切换到 istio 包的解压目录。例如 istio-0.4.0：

```
cd istio-0.4.0
```

4. 将 `istioctl` 客户端二进制文件加到 PATH 中。

   例如，在 MacOS 或 Linux 系统上执行下面的命令：

```
export PATH=$PWD/bin:$PATH
```

5. 安装 Istio 的核心部分。选择面两个 互斥 选项中的之一：

   a) 安装 Istio 的时候不启用 sidecar 之间的 TLS 双向认证：

   为具有现在应用程序的集群选择该选项，使用 Istio sidecar 的服务需要能够与非 Istio Kubernetes 服务以及使用 liveliness 和 readiness 探针、headless service 和 StatefulSet 的应用程序通信。

```
kubectl apply -f install/kubernetes/istio.yaml
```

   或者

   b) 安装 Istio 的时候启用 sidecar 之间的 TLS 双向认证：

```
kubectl apply -f install/kubernetes/istio-auth.yaml
```

这两个选项都会创建 `istio-system` 命名空间以及所需的 RBAC 权限，并部署 Istio-Pilot、Istio-Mixer、Istio-Ingress、Istio-Egress 和 Istio-CA（证书颁发机构）。

i. 可选的：如果您的 kubernetes 集群开启了 alpha 功能，并想要启用 自动注入 sidecar，需要安装 Istio-Initializer：

```
kubectl apply -f install/kubernetes/istio-initializer.yaml
```

验证安装

1. 确认系列 kubernetes 服务已经部署了：`istio-pilot`、`istio-mixer`、`istio-ingress`：

```
kubectl get svc -n istio-system
```

```
NAME            CLUSTER-IP      EXTERNAL-IP       PORT(S)
                    AGE
istio-ingress   10.83.245.171   35.184.245.62     80:32730/T
CP,443:30574/TCP    5h
istio-pilot     10.83.251.173   <none>            8080/TCP,8
081/TCP             5h
istio-mixer     10.83.244.253   <none>            9091/TCP,9
094/TCP,42422/TCP   5h
```

注意：如果您运行的集群不支持外部负载均衡器（如 minikube），`istio-ingress` 服务的 `EXTERNAL-IP` 显示 `<pending>`。你必须改为使用 NodePort service 或者 端口转发方式来访问应用程序。

2. 确认对应的 Kubernetes pod 已部署并且所有的容器都启动并运行：`istio-pilot-*`、`istio-mixer-*`、`istio-ingress-*`、`istio-ca-*`，`istio-initializer-*` 是可以选的。

```
  kubectl get pods -n istio-system
```

```
istio-ca-3657790228-j21b9           1/1       Running   0
       5h
istio-ingress-1842462111-j3vcs      1/1       Running   0
       5h
istio-initializer-184129454-zdgf5   1/1       Running   0
       5h
istio-pilot-2275554717-93c43        1/1       Running   0
       5h
istio-mixer-2104784889-20rm8        2/2       Running   0
       5h
```

部署应用

您可以部署自己的应用或者示例应用程序如 BookInfo。 注意：应用程序必须使用 HTTP/1.1 或 HTTP/2.0 协议来传递 HTTP 流量，因为 HTTP/1.0 已经不再支持。

如果您启动了 Istio-Initializer，如上所示，您可以使用 `kubectl create` 直接部署应用。Istio-Initializer 会向应用程序的 pod 中自动注入 Envoy 容器：

```
kubectl create -f <your-app-spec>.yaml
```

如果您没有安装 Istio-initializer 的话，您必须使用 istioctl kube-inject 命令在部署应用之前向应用程序的 pod 中手动注入 Envoy 容器：

```
kubectl create -f <(istioctl kube-inject -f <your-app-spec>.yaml
)
```

卸载

- 卸载 Istio initializer:

  如果您安装 Isto 的时候启用了 initializer，请卸载它：

  ```
  kubectl delete -f install/kubernetes/istio-initializer.yaml
  ```

- 卸载 Istio 核心组件。对于 Istio 0.4.0 版本，删除 RBAC 权限，`istio-system` namespace，和该命名空间的下的各层级资源。

  不必理会在层级删除过程中的各种报错，因为这些资源可能已经被删除的。

  a) 如果您在安装 Istio 的时候关闭了 TLS 双向认证：

  ```
  kubectl delete -f install/kubernetes/istio.yaml
  ```

  或者

  b) 如果您在安装 Istio 的时候启用了 TLS 双向认证：

  ```
  kubectl delete -f install/kubernetes/istio-auth.yaml
  ```

下一步

- 查看 BookInfo 应用程序示例
- 查看如何 验证 Istio 双向TLS认证

安装 Istio sidecar

备注：以下需要 Istio 0.5.0 或更高。 0.4.0 及 以前版本参见 https://archive.istio.io/v0.4/docs/setup/kubernetes/sidecar-injection。

Pod Spec 中需满足的条件

为了成为 Service Mesh 中的一部分，kubernetes 集群中的每个 Pod 都必须满足如下条件：

1. **Service** 关联：每个 pod 都必须只属于某一个 Kubernetes Service （当前不支持一个 pod 同时属于多个 service）。
2. 命名的端口：Service 的端口必须命名。端口的名字必须遵循如下格式 `<protocol>[-<suffix>]` ，可以是*http*、*http2*、 *grpc*、 *mongo*、 或者 *redis* 作为 `<protocol>` ，这样才能使用 Istio 的路由功能。例如 `name: http2-foo` 和 `name: http` 都是有效的端口名称，而 `name: http2foo` 不是。如果端口的名称是不可识别的前缀或者未命名，那么该端口上的流量就会作为普通的 TCP 流量（除非使用 `Protocol: UDP` 明确声明使用 UDP 端口）。
3. 带有 **app label** 的 **Deployment**：我们建议 kubernetes 的 `Deploymenet` 资源的配置文件中为 Pod 明确指定 `app` label。每个Deployment 的配置中都需要有个不同的有意义的 `app` 标签。 `app` label 用于在分布式坠重中添加上下文信息。
4. **Mesh** 中的每个 **pod** 里都有一个 **Sidecar**：最后，Mesh 中的每个 pod 都必须运行与 Istio 兼容的sidecar。以下部分介绍了将 sidecar 注入到 pod 中的两种方法：使用 `istioctl` 命令行工具手动注入，或者使用 istio initializer 自动注入。注意 sidecar 不涉及到容器间的流量，因为他们都在同一个 pod 中。

注入

手动注入需要修改控制的配置文件，如 deployment。通过修改 deployment 文件中的 pod 模板规范可达到该deployment 下创建的所有 pod 都注入 sidecar。添加/更新/删除 sidecar 需要修改整个 deployment。

自动注入会在 pod 创建的时候注入，无需控制资源。sidecars 可通过以下方式被更新：有选择性地手工删除 pod或者有条理地进行 deployment 滚动更新。

手动或者自动注入都使用同样的模板配置。自动注入会从 `istio-system` 命名空间下获取 `istio-inject` 的 ConfigMap。手动注入的模板可以是本地文件或者 Configmap 。

两个变种的注入模板在默认安装中也会被提供：`istio-sidecar-injector-configmap-release.yaml` 和 `istio-sidecar-injector-configmap-debug.yaml` 。调试的版本包含调试的代理镜像、附加的日志和core dump功能以用于调试 sidecar 代理。

手动注入 sidecar

手工注入使用默认的模板和动态从 istio ConfigMap中获取服务网格的配置文件。附件参数覆盖可以通过程序的flags设置（参见： `istioctl kube-inject –help` ）

```
kubectl apply -f <(~istioctl kube-inject -f samples/sleep/sleep.
yaml)
```

`kube-inject` 也可以在没有 kubernetes 集群的情况下运行。创建本地的注入和网格配置。

```
kubectl create -f install/kubernetes/istio-sidecar-injector-conf
igmap-release.yaml \
    --dry-run \
    -o=jsonpath='{.data.config}' > inject-config.yaml

kubectl -n istio-system get configmap istio -o=jsonpath='{.data.
mesh}' > mesh-config.yaml
```

在输入的文件上运行 `kube-inject` ：

```
istioctl kube-inject \
    --injectConfigFile inject-config.yaml \
    --meshConfigFile mesh-config.yaml \
    --filename samples/sleep/sleep.yaml \
    --output sleep-injected.yaml
```

部署注入后的 YAML 文件：

```
kubectl apply -f sleep-injected.yaml
```

验证 sidecar 已经注入到 Deployment 中：

```
kubectl get deployment sleep -o wide
```

```
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
 CONTAINERS          IMAGES                              SELECTOR
sleep     1         1         1            1           2h
 sleep,istio-proxy   tutum/curl,unknown/proxy:unknown   app=slee
p
```

自动注入 sidecar

关于Webhook管理控制的整体介绍可以参见：validatingadmissionwebhook-alpha-in-18-beta-in-19。

前置条件

kubernetes 1.9 的集群需要启动 `admissionregistration.k8s.io/v1beta1` 。

```
kubectl api-versions | grep admissionregistration.k8s.io/v1beta1
```

```
Copyadmissionregistration.k8s.io/v1beta1
```

GKE

1.9.1 版本中已经具备非白名单早期用户访问的 alpha 集群 (参考：https://cloud.google.com/kubernetes-engine/release-notes#january-16-2018）。

```
gcloud container clusters create <cluster-name> \
    --enable-kubernetes-alpha
    --cluster-version=1.9.1-gke.0
    --zone=<zone>
    --project <project-name>
```

```
Copygcloud container clusters get-credentials <cluster-name> \
    --zone <zone> \
    --project <project-name>
```

```
Copykubectl create clusterrolebinding cluster-admin-binding \
    --clusterrole=cluster-admin \
    --user=$(gcloud config get-value core/account)
```

minikube

TODO(https://github.com/istio/istio.github.io/issues/885)

IBM Cloud Container Service

TODO(https://github.com/istio/istio.github.io/issues/887)

AWS with Kops

TODO(https://github.com/istio/istio.github.io/issues/886)

安装 Webhook

安装基本的 Istio

```
kubectl apply -f install/kubernetes/istio.yaml
```

Webhook 需要签名的 cert/key 对。使用 `install/kubernetes/webhook-create-signed-cert.sh` 生成 kuberntes CA 签发的 cert/key 对，结果文件 cert/key 被 sidecar 注入的 webhook 当做 kuberntes 密钥去使用。

备注：Kubernetes CA 批准（approval）需要权限能够创建和验证 CSR。更多信息参见：https://kubernetes.io/docs/tasks/tls/managing-tls-in-a-cluster 和 `install/kubernetes/webhook-create-signed-cert.sh` 。

```
./install/kubernetes/webhook-create-signed-cert.sh \
    --service istio-sidecar-injector \
    --namespace istio-system \
    --secret sidecar-injector-certs
```

安装 sidecar 注入的 configmap：

```
kubectl apply -f install/kubernetes/istio-sidecar-injector-configmap-release.yaml
```

设置安装 yaml 中 webhook 的 `caBundle` ，以便 api-server能够用来调用 webhook。

```
cat install/kubernetes/istio-sidecar-injector.yaml | \
     ./install/kubernetes/webhook-patch-ca-bundle.sh > \
     install/kubernetes/istio-sidecar-injector-with-ca-bundle.yaml
```

安装 sidecar 注册器 webhook。

```
kubectl apply -f install/kubernetes/istio-sidecar-injector-with-ca-bundle.yaml
```

sidecar 注入的 webhook 应该已经运行。

```
kubectl -n istio-system get deployment -listio=sidecar-injector
```

```
NAME                     DESIRED   CURRENT   UP-TO-DATE   AVAILA
BLE   AGE
istio-sidecar-injector   1         1         1            1
      1d
```

NamespaceSelector 决定是否在一个对象上运行 webhook，取决于该对象的所在的 namespace 是否匹配选择器

(参见： https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors)。默认webhook 的配置使用 `istio-injection=enabled` .

采用标签 `istio-injection` 查看 namespae，用以确认 `default` 命名空间下没有被打标签。

```
Copykubectl get namespace -L istio-injection

CopyNAME           STATUS        AGE       ISTIO-INJECTION
default        Active        1h
istio-system   Active        1h
kube-public    Active        1h
kube-system    Active        1h
```

部署程序

部署 sleep 程序。验证 deployment 和 pod 都有一个单独的container。

```
kubectl apply -f samples/sleep/sleep.yaml
```

```
kubectl get deployment -o wide
```

```
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
 CONTAINERS   IMAGES       SELECTOR
sleep     1         1         1            1           12m
 sleep        tutum/curl   app=sleep
```

```
kubectl get pod
```

```
NAME                     READY     STATUS        RESTARTS   AGE
sleep-776b7bcdcd-7hpnk   1/1       Running       0          4
```

为 `default` 命名空间下的设置标签 `istio-injection=enabled`

```
kubectl label namespace default istio-injection=enabled
```

```
kubectl get namespace -L istio-injection
```

```
NAME           STATUS    AGE       ISTIO-INJECTION
default        Active    1h        enabled
istio-system   Active    1h
kube-public    Active    1h
kube-system    Active    1h
```

注入动作会在 pod 创建的时候发生。杀掉运行的 pod 并验新创建的 pod 已经被注入 sidecar。原来老的 pod 具有 1/1 就绪的的 containers 而被注入的新的 pod 具有 2/2 就绪的 containers。

```
kubectl delete pod sleep-776b7bcdcd-7hpnk
```

```
kubectl get pod
```

```
NAME                     READY     STATUS        RESTARTS   AGE
sleep-776b7bcdcd-7hpnk   1/1       Terminating   0          1m
sleep-776b7bcdcd-bhn9m   2/2       Running       0          7s
```

在 `default` 命名空间下禁用注入，验证新的 pod 中 sidecar 不再存在。

```
kubectl label namespace default istio-injection-
```

```
kubectl delete pod sleep-776b7bcdcd-bhn9m
```

```
kubectl get pod
```

```
NAME                     READY     STATUS        RESTARTS   AGE
sleep-776b7bcdcd-bhn9m   2/2       Terminating   0          2m
sleep-776b7bcdcd-gmvnr   1/1       Running       0          2s
```

了解发生什么

admissionregistration.k8s.io/v1alpha1#MutatingWebhookConfiguration 配置 webhook 何时被 kubernetes 调用。Istio 默认选择 namespace 下被打了标签 `istio-injection=enabled` 的Pod。这个行为可以通过修改 `install/kubernetes/istio-sidecar-injector-with-ca-bundle.yaml` 中的 MutatingWebhookConfiguration 配置。

`istio-system` 命名空间下的 `istio-inject` ConfigMap 保存了默认注入策略（policy）和 sidecar 注入模板（template）。

策略（***policy***）

`disabled` - sidecar 注入器默认不会注入到 pod 中。添加pod模板定义中的注解 `sidecar.istio.io/inject` 值为 `true` 会启用注入功能。

`enabled` - sidecar 注入器默认会注入到 pod 中。添加pod模板定义中的注解 `sidecar.istio.io/inject` 值为 `false` 会禁止注入功能。

模板（***template***）

sidecar 注入模板使用 https://golang.org/pkg/text/template，当解析和执行后会被解码成以下的 struct 结构，包含了一系列要注入 pod 的 containers 和 volumes。

```
    type SidecarInjectionSpec struct {
          InitContainers []v1.Container `yaml:"initContainers"`
          Containers     []v1.Container `yaml:"containers"`
          Volumes        []v1.Volume    `yaml:"volumes"`
    }
```

模板会在运行的过程中被应用成以下的数据 struct：

```
type SidecarTemplateData struct {
    ObjectMeta  *metav1.ObjectMeta
    Spec        *v1.PodSpec
    ProxyConfig *meshconfig.ProxyConfig  // Defined by https://istio.io/docs/reference/config/service-mesh.html#proxyconfig
    MeshConfig  *meshconfig.MeshConfig   // Defined by https://istio.io/docs/reference/config/service-mesh.html#meshconfig
}
```

`ObjectMeta` 和 `Spec` 来自于 pod。 `ProxyConfig` 和 `MeshConfig` 来自于 namespace `istio-system` 下的 `istio` ConfigMap。模板可以用于条件定义注入的 containers 和 volumes。

例如，以下的模板片段，来自于 install/kubernetes/istio-sidecar-injector-configmap-release.yaml

```
containers:
- name: istio-proxy
  image: istio.io/proxy:0.5.0
  args:
  - proxy
  - sidecar
  - --configPath
  - {{ .ProxyConfig.ConfigPath }}
  - --binaryPath
  - {{ .ProxyConfig.BinaryPath }}
  - --serviceCluster
  {{ if ne "" (index .ObjectMeta.Labels "app") -}}
  - {{ index .ObjectMeta.Labels "app" }}
  {{ else -}}
  - "istio-proxy"
  {{ end -}}
```

当 `sample/sleep/sleep.yaml` 中的pod定义被应用后会被扩展成：

```
containers:
- name: istio-proxy
  image: istio.io/proxy:0.5.0
  args:
  - proxy
  - sidecar
  - --configPath
  - /etc/istio/proxy
  - --binaryPath
  - /usr/local/bin/envoy
  - --serviceCluster
  - sleep
```

卸载 webhook

```
kubectl delete -f install/kubernetes/istio-sidecar-injector-with
-ca-bundle.yaml
```

上述命令并不移除已经已经注入到 Pod 中的 Sidecar。如有需要可以通过滚动升级（rolling update）或者简单删除 pods 使 部署（deployment）重新创建 Pod。

拓展 Istio Mesh

将虚拟机或裸机集成到部署在 kubernetes 集群上的 Istio mesh 中的说明。

前置条件

- 按照 安装指南 在 kubernetes 集群上安装 Istio service mesh。
- 机器必须具有到 mesh 端点的 IP 地址连接。这通常需要一个 VPC 或者 VPN，以及一个向端点提供直接路由（没有 NAT 或者防火墙拒绝）的容器网络。及其不需要访问有 Kubernetes 分配的 cluster IP。
- 虚拟机必须可以访问到 Istio 控制平面服务（如Pilot、Mixer、CA）和 Kubernetes DNS 服务器。通常使用 内部负载均衡器 来实现。

  您也可以使用 NodePort，在虚拟机上运行 Istio 的组件，或者使用自定义网络配置，有几个单独的文档会涵盖这些高级配置。

安装步骤

安装过程包括准备用于拓展的 mesh 和安装和配置虚拟机。

install/tools/setupMeshEx.sh ：这是一个帮助大家设置 kubernetes 环境的示例脚本。检查脚本内容和支持的环境变量（如 GCP_OPTS）。

install/tools/setupIstioVM.sh：这是一个用于配置主机环境的示例脚本。 您应该根据您的配置工具和DNS要求对其进行自定义。

准备要拓展的 Kubernetes 集群：

- 为 Kube DNS、Pilot、Mixer 和 CA 安装内部负载均衡器（ILB）。每个云供应商的配置都有所不同，根据具体情况修改注解。

> 0.2.7 版本的 YAML 文件的 DNS ILB 的 namespace 配置不正确。使用 这一个 替代。 `setupMeshEx.sh` 中也有错误。使用上面链接中的最新文件或者从 GitHub.com/istio/istio 克隆。

```
kubectl apply -f install/kubernetes/mesh-expansion.yaml
```

- 生成要部署到虚拟机上的 Istio `cluster.env` 配置。该文件中包含要拦截的 cluster IP 地址范围。

```
export GCP_OPTS="--zone MY_ZONE --project MY_PROJECT"
```

```
install/tools/setupMeshEx.sh generateClusterEnv MY_CLUSTER_NAME
```

该示例生成的文件：

```
cat cluster.env
```

```
ISTIO_SERVICE_CIDR=10.63.240.0/20
```

- 产生虚拟机使用的 DNS 配置文件。这样可以让虚拟机上的应用程序解析到集群中的服务名称，这些名称将被 sidecar 拦截和转发。

```
# Make sure your kubectl context is set to your cluster
install/tools/setupMeshEx.sh generateDnsmasq
```

该示例生成的文件：

```
cat kubedns
```

```
server=/svc.cluster.local/10.150.0.7
address=/istio-mixer/10.150.0.8
address=/istio-pilot/10.150.0.6
address=/istio-ca/10.150.0.9
address=/istio-mixer.istio-system/10.150.0.8
address=/istio-pilot.istio-system/10.150.0.6
address=/istio-ca.istio-system/10.150.0.9
```

设置机器

例如，您可以使用下面的“一条龙”脚本复制和安装配置：

```
# 检查该脚本看看它是否满足您的需求
# 在 Mac 上，使用 brew install base64 或者 set BASE64_DECODE="/usr
/bin/base64 -D"
export GCP_OPTS="--zone MY_ZONE --project MY_PROJECT"
```

```
install/tools/setupMeshEx.sh machineSetup VM_NAME
```

或者等效得手动安装步骤如下：

------ 手动安装步骤开始 ------

- 将配置文件和 Istio 的 Debian 文件复制到要加入到集群的每台机器上。重命名为 `/etc/dnsmasq.d/kubedns` 和 `/var/lib/istio/envoy/cluster.env` 。
- 配置和验证 DNS 配置。需要安装 `dnsmasq` 或者直接将其添加到 `/etc/resolv.conf` 中，或者通过 DHCP 脚本。验证配置是否有效，检查虚拟机是否可以解析和连接到 pilot，例如：

在虚拟机或外部主机上：

```
host istio-pilot.istio-system
```

产生的消息示例：

```
# Verify you get the same address as shown as "EXTERNAL-IP" in '
kubectl get svc -n istio-system istio-pilot-ilb'
istio-pilot.istio-system has address 10.150.0.6
```

检查是否可以解析 cluster IP。实际地址取决您的 deployment：

```
host istio-pilot.istio-system.svc.cluster.local.
```

该示例产生的消息：

```
istio-pilot.istio-system.svc.cluster.local has address 10.63.247
.248
```

同样检查 istio-ingress：

```
host istio-ingress.istio-system.svc.cluster.local.
```

该示例产生的消息：

```
istio-ingress.istio-system.svc.cluster.local has address 10.63.2
43.30
```

- 验证连接性，检查迅即是否可以连接到 Pilot 的端点：

```
curl 'http://istio-pilot.istio-system:8080/v1/registration/istio
-pilot.istio-system.svc.cluster.local|http-discovery'
```

```
{
  "hosts": [
   {
    "ip_address": "10.60.1.4",
    "port": 8080
   }
  ]
}
```

```
# 在虚拟机上使用上面的地址。将直接连接到运行 istio-pilot 的 pod。
curl 'http://10.60.1.4:8080/v1/registration/istio-pilot.istio-sy
stem.svc.cluster.local|http-discovery'
```

- 提取出实话 Istio 认证的 secret 并将它复制到机器上。Istio 的默认安装中包括 CA，即使是禁用了自动 `mTLS` 设置（她为每个 service account 创建

secret，secret 命名为 `istio.<serviceaccount>` ）也会生成 Istio secret。建议您执行此步骤，以便日后启用 mTLS，并升级到默认启用 mTLS 的未来版本。

```
# ACCOUNT 默认是 'default'，SERVICE_ACCOUNT 是环境变量
# NAMESPACE 默认为当前 namespace，SERVICE_NAMESPACE 是环境变量
# （这一步由 machineSetup 完成）
# 在 Mac 上执行 brew install base64 或者 set BASE64_DECODE="/usr/bin/base64 -D"
install/tools/setupMeshEx.sh machineCerts ACCOUNT NAMESPACE
```

生成的文件（ `key.pem` , `root-cert.pem` , `cert-chain.pem` ）必须拷贝到每台主机的 /etc/certs 目录，并且让 istio-proxy 可读。

- 安装 Istio Debian 文件，启动 `istio` 和 `istio-auth-node-agent` 服务。从 github releases 获取 Debian 安装包：

```
# 注意：在软件源配置好后，下面的额命令可以使用 'apt-get' 命令替代。

source istio.VERSION # defines version and URLs env var
curl -L ${PILOT_DEBIAN_URL}/istio-agent.deb > ${ISTIO_STAGING}/istio-agent.deb
curl -L ${AUTH_DEBIAN_URL}/istio-auth-node-agent.deb > ${ISTIO_STAGING}/istio-auth-node-agent.deb
curl -L ${PROXY_DEBIAN_URL}/istio-proxy.deb > ${ISTIO_STAGING}/istio-proxy.deb

dpkg -i istio-proxy-envoy.deb
dpkg -i istio-agent.deb
dpkg -i istio-auth-node-agent.deb

systemctl start istio
systemctl start istio-auth-node-agent
```

------ 手动安装步骤结束 ------

安装完成后，机器就能访问运行在 Kubernetes 集群上的服务或者其他的 mesh 拓展的机器。

```
# Assuming you install bookinfo in 'bookinfo' namespace
curl productpage.bookinfo.svc.cluster.local:9080
```

```
... html content ...
```

检查进程是否正在运行：

```
ps aux |grep istio
```

```
root      6941  0.0  0.2  75392 16820 ?        Ssl  21:32   0:00
 /usr/local/istio/bin/node_agent --logtostderr
root      6955  0.0  0.0  49344  3048 ?        Ss   21:32   0:00
 su -s /bin/bash -c INSTANCE_IP=10.150.0.5 POD_NAME=demo-vm-1 PO
D_NAMESPACE=default exec /usr/local/bin/pilot-agent proxy > /var
/log/istio/istio.log istio-proxy
istio-p+  7016  0.0  0.1 215172 12096 ?        Ssl  21:32   0:00
 /usr/local/bin/pilot-agent proxy
istio-p+  7094  4.0  0.3  69540 24800 ?        Sl   21:32   0:37
 /usr/local/bin/envoy -c /etc/istio/proxy/envoy-rev1.json --rest
art-epoch 1 --drain-time-s 2 --parent-shutdown-time-s 3 --servic
e-cluster istio-proxy --service-node sidecar~10.150.0.5~demo-vm-
1.default~default.svc.cluster.local
```

检查 Istio auth-node-agent 是否健康：

```
sudo systemctl status istio-auth-node-agent
```

```
● istio-auth-node-agent.service - istio-auth-node-agent: The Ist
io auth node agent
   Loaded: loaded (/lib/systemd/system/istio-auth-node-agent.ser
vice; disabled; vendor preset: enabled)
   Active: active (running) since Fri 2017-10-13 21:32:29 UTC; 9
s ago
     Docs: http://istio.io/
 Main PID: 6941 (node_agent)
    Tasks: 5
   Memory: 5.9M
      CPU: 92ms
   CGroup: /system.slice/istio-auth-node-agent.service
           └─6941 /usr/local/istio/bin/node_agent --logtostderr

Oct 13 21:32:29 demo-vm-1 systemd[1]: Started istio-auth-node-ag
ent: The Istio auth node agent.
Oct 13 21:32:29 demo-vm-1 node_agent[6941]: I1013 21:32:29.46931
4    6941 main.go:66] Starting Node Agent
Oct 13 21:32:29 demo-vm-1 node_agent[6941]: I1013 21:32:29.46936
5    6941 nodeagent.go:96] Node Agent starts successfully.
Oct 13 21:32:29 demo-vm-1 node_agent[6941]: I1013 21:32:29.48332
4    6941 nodeagent.go:112] Sending CSR (retrial #0) ...
Oct 13 21:32:29 demo-vm-1 node_agent[6941]: I1013 21:32:29.86257
5    6941 nodeagent.go:128] CSR is approved successfully. Will r
enew cert in 29m59.137732603s
```

在拓展的 **mesh** 中的机器上运行服务

- 配置 sidecar 拦截端口。在 `/var/lib/istio/envoy/sidecar.env` 中通过 `ISTIO_INBOUND_PORTS` 环境变量配置。

  例如（运行服务的虚拟机）：

  ```
  echo "ISTIO_INBOUND_PORTS=27017,3306,8080" > /var/lib/istio
  /envoy/sidecar.env
  systemctl restart istio
  ```

- 手动配置 selector-less 的 service 和 endpoint。“selector-less” service 用于那些不依托 Kubernetes pod 的 service。

  例如，在有权限的机器上修改 Kubernetes 中的 service：

  ```
  # istioctl register servicename machine-ip portname:port
  istioctl -n onprem register mysql 1.2.3.4 3306
  istioctl -n onprem register svc1 1.2.3.4 http:7000
  ```

安装完成后，Kubernetes pod 和其它 mesh 扩展将能够访问集群上运行的服务。

整合到一起

请参阅 拓展 BookInfo Mesh 指南。

Nomad 和 Consul

在基于 Consul 的环境中安装 Istio 控制平面，不论是否使用 Nomad。

- 使用 Docker 快速开始：使用 Docker Compose 快速安装 Istio service mesh 指南。
- 安装：在基于 Consul 的环境中安装 Istio 控制平面，不论是否使用 Nomad。
- FAQ：常见问题，当前的限制与问题排查。

使用 Docker 快速开始

使用 Docker Compose 快速安装 Istio service mesh 指南。

前置条件

- Docker
- Docker Compose

安装步骤

1. 到 Istio release 页面根据您自己的操作系统下载相应的安装文件。如果您使用的是 MacOS 或者 Linux 系统的话，可以直接执行下面的命令自动下载和安装：

```
curl -L https://git.io/getLatestIstio | sh -
```

2. 解压下载好的文件，切换到文件在目录。安装文件目录下包含：
    - `install/` 目录下是 kubernetes 使用的 `.yaml` 安装文件
        - `samples/` 目录下是示例程序
        - `istioctl` 客户端二进制文件在 `bin` 目录下。`istioctl` 文件用户手动注入 Envoy sidecar 代理、创建路由和策略等。
        - `istio.VERSION` 配置文件
3. 将 `istioctl` 客户端二进制文件加到 PATH 中。例如，在 MacOS 或 Linux 系统上执行下面的命令：

```
export PATH=$PWD/bin:$PATH
```

4. 对于 Linux 用户，配置 `DOCKER_GATEWAY` 环境变量：

```
export DOCKER_GATEWAY=172.28.0.1:
```

5. 切换根目录到 Istio 的安装目录。
6. 启动 Istio 控制平面容器：

```
docker-compose -f install/consul/istio.yaml up -d
```

7. 确认所有的 docker 容器都在运行：

```
docker ps -a
```

> 如果 Istio Pilot 容器终止了，确认你使用了 `istio context-create` 命令并重新运行上一步。

8. 配置 `istioctl` 使用 Istio API server 映射的本地端口：

```
istioctl context-create --api-server http://localhost:8080
```

部署应用

现在您可以部署自己的应用了，也可以部署我们提供的示例应用，例如 BookInfo.

> 注意 1：由于在安装 Docker 的时候没有 pod 的概念，sidecar 和应用都运行在同一个容器里。我们会使用 Registrator 在 Consul 服务注册表中注册服务的实例。
>
> 注意 2：应用程序必须使用 HTTP/1.1 或 HTTP/2.0 协议进行 HTTP 请求，因为不支持 HTTP/1.0 的流量。

```
docker-compose -f <your-app-spec>.yaml up -d
```

卸载

1. 卸载 Istio 核心组件，只要删除 docker 容器即可：

```
docker-compose -f install/consul/istio.yaml down
```

下一步

- 参阅 BookInfo 应用程序示例。

安装

> 注意：在 Nomad 上安装还没有测试过。

在非 kubernetes 环境中使用 Istio 有几个关键问题：

1. 使用 Istio API server 创建 Istio 控制平面
2. 向服务的每个实例中注入 Istio sidecar
3. 确认通过 sidecar 来路由请求

安装控制平面

Istio 控制平面由四个主要服务组成：Pilot、Mixer、CA 和 API server。

API Server

Istio API server（基于 Kubernetes API server）提供了诸如配置管理和基于角色的访问控制（RBAC）等功能。API server 需要一个 etcd 集群 作为持久化存储。可以在 这里 找到关于设置 API server 的详细说明。

关于设置 Kubernetes API server 的选项的说明请参阅 这里。

本地安装

处于 POC（概念验证） 的目的，可以使用 Docker-compose 文件来启用一个简单的单容器 API server：

```
version: '2'
services:
  etcd:
    image: quay.io/coreos/etcd:latest
    networks:
      istiomesh:
        aliases:
          - etcd
    ports:
      - "4001:4001"
```

```
      - "2380:2380"
      - "2379:2379"
    environment:
      - SERVICE_IGNORE=1
    command: [
              "/usr/local/bin/etcd",
              "-advertise-client-urls=http://0.0.0.0:2379",
              "-listen-client-urls=http://0.0.0.0:2379"
             ]

  istio-apiserver:
    image: gcr.io/google_containers/kube-apiserver-amd64:v1.7.3
    networks:
      istiomesh:
        ipv4_address: 172.28.0.13
        aliases:
          - apiserver
    ports:
      - "8080:8080"
    privileged: true
    environment:
      - SERVICE_IGNORE=1
    command: [
               "kube-apiserver", "--etcd-servers", "http://etcd:
2379",
               "--service-cluster-ip-range", "10.99.0.0/16",
               "--insecure-port", "8080",
               "-v", "2",
               "--insecure-bind-address", "0.0.0.0"
             ]
```

其他 Istio 组件

Istio Pilot、Mixer 和 CA 的 Debian 软件包可以通过 Isito 的发行版获得。或者可以以 docker 容器的方式运行（docker.io/istio/pilot、docker.io/istio/mixer、docker.io/istio/istio-ca）。请注意，这些组件是无状态的，可以水平缩放。这些组

件都依赖于 Istio API server，而 Istio API server 由依赖 etcd 集群来实现持久化。为了实现高可用，每个控制平面服务都可以在 Nomad 中以 job 的方式运行，其中 service stanza 可以用来描述控制平面服务的期望属性。

向服务实例中添加 Sidecar

应用程序的每个实例中都必须伴有一个 Istio sidecar。根据您的安装单位（Docker 容器、虚拟机或者是裸机），Istio sidecar 都需要安装到这些组件中。例如，如果您的基础架构使用的是虚拟机，则必须在每台虚拟机上运行一个 Istio sidecar 进程才能将这台迅即添加到 service mesh 中。

有种将 sidecar 打包到机遇 Nomad 的部署中去的方式，就是将 Istio sidecar 进程作为 任务组 中的一个进程。任务组是一个或多个相关任务的集合，这些任务保证驻留在同一台主机上。但是，与 kuberntes pod 不同的是，统一组中的任务不共享相同的网络命名空间。因此，当使用 `iptables` 规则透明地重新路由所有网络流量时，必须注意确保每台主机上只运行一个任务组。当 Istio 支持非透明代理（应用程序明确的与 sidecar 通信）时，将不会再有此限制。

通过 Istio Sidecar 路由流量

部分 Sidecar 安装时会设置适当的 IP Table 规则，以通过 Istio sidecar 透明地路由应用程序的网络流量。在 这里 可以找到设置这种转发规则的 IP Table 脚本。

> 注意：该脚本必须再启动应用程序和 sidecar 之前执行。

Eureka

如何在基于 Eureka 的环境中安装 Isto 控制平面的说明。

- 使用 Docker 快速开始：使用 Docker Compose 快速安装 Istio service mesh 指南。
- 安装：如何在基于 Eureka 的环境中安装 Isto 控制平面的说明。
- FAQ：常见问题，当前限制和故障排查。

使用 Docker 快速开始

使用 Docker Compose 安装和配置 Istio service mesh 的快速入门指南。

前置条件

- Docker
- Docker Compose

安装步骤

1. 到 Istio release 页面，下载适合自己的操作系统的最新安装文件。 如果你用的是 MacOS 或者 Linux ，也可以运行下面的命令自动下载并解压最新的发布包。

   ```
   curl -L https://git.io/getLatestIstio | sh -
   ```

2. 解压安装文件，并进入解压后的文件夹。安装目录包含以下内容：
   - 示例应用在 `samples/` 下面。
   - `istioctl` 客户端在 `bin/` 文件夹下。 `istioctl` 用来创建路由以及策略。
   - `istio.VERSION` 配置文件。
3. 添加 `istioctl` 客户端到你的 PATH 环境变量中。在 MacOS 或者 Linux 系统上可以运行以下命令：

   ```
   export PATH=$PWD/bin:$PATH
   ```

4. 跳转到 Istio 的安装根目录。
5. 启动 Istio 控制面板容器：

   ```
   docker-compose -f install/eureka/istio.yaml up -d
   ```

6. 确保所有的 docker 容器都运行正常：

```
docker ps -a
```

> 如果 Istio Pilot 容器终止了，确保你运行了 `istioctl context-create` 命令，然后重新运行前一步的 docker-compose 命令。

7. 配置 `istioctl` 使用 Istio API 服务的本地映射端口：

```
 istioctl context-create --context istio-local --api-server
http://localhost:8080
```

部署应用

现在你可以部署你自己的应用或者安装程序中提供的示例应用，比如 BookInfo。

> 注意 1: 因为 Docker 安装环境下没有 pods 的概念，Istio 的 sidecar 和应用程序会运行在同一个容器中。我们使用 Registrator 自动注册服务实例到 Console 服务注册中心。
>
> 注意 2: 应用的 HTTP 流量必须必须使用 HTTP/1.1 或者 HTTP/2.0 协议，因为 Istio 不支持 HTTP/1.0。

```
docker-compose -f <your-app-spec>.yaml up -d
```

卸载

1. 卸载 Istio 核心组件只需要删除 docker 容器：

```
docker-compose -f install/eureka/istio.yaml down
```

下一步

- 参看 BookInfo 示例应用。

安装

在非 kubernetes 环境中使用 Istio 主要包含以下关键任务：

1. 使用 Istio API 服务搭建和设置 Istio 控制面板
2. 添加 Istio sidecar到每个服务的实例中
3. 确保请求是通过 sidecar 路由的

安装控制平面

Istio 控制面板由四个主要服务组成： Pilot, Mixer, CA, 以及 API 服务.

API 服务器

Istio 的 API 服务器（基于 Kubernetes 的 API 服务器）提供了配置管理以及 RBAC（基于角色的访问控制）等关键的功能。API 服务器需要一个 etcd 集群作为持久化存储。搭建 API 服务器的详细说明在这里可以找到。Kubernetes API 服务器的配置选项文档在这里。

本地安装

为了进行概念验证(*proof of concept*)，可以使用以下 Docker Compose 文件安装简单的单容器 API 服务器：

```
version: '2'
services:
  etcd:
    image: quay.io/coreos/etcd:latest
    networks:
      default:
        aliases:
          - etcd
    ports:
      - "4001:4001"
      - "2380:2380"
      - "2379:2379"
```

```
    environment:
      - SERVICE_IGNORE=1
    command: [
              "/usr/local/bin/etcd",
              "-advertise-client-urls=http://0.0.0.0:2379",
              "-listen-client-urls=http://0.0.0.0:2379"
             ]

  istio-apiserver:
    image: gcr.io/google_containers/kube-apiserver-amd64:v1.7.3
    networks:
      default:
        aliases:
          - apiserver
    ports:
      - "8080:8080"
    privileged: true
    environment:
      - SERVICE_IGNORE=1
    command: [
               "kube-apiserver", "--etcd-servers", "http://etcd:
2379",
               "--service-cluster-ip-range", "10.99.0.0/16",
               "--insecure-port", "8080",
               "-v", "2",
               "--insecure-bind-address", "0.0.0.0"
             ]
```

其它 Istio 组件

Istio 的 Pilot，Mixer 和 CA 的 Debian 系统的软件包可通过 Istio 发行版获得。或者，这些组件也可以通过 Docker 容器运行（docker.io/istio/pilot，docker.io/istio/mixer，docker.io/istio/istio-ca）。注意，这些组件是无状态的，所以可以水平伸缩。每个组件都依赖于 Istio API 服务器，而 Istio API 服务器又依赖于 etcd 集群来实现持久化。

向服务实例中添加 Sidecar

应用程序中的每个服务实例必须伴随着 Istio 的 sidecar。 根据你的安装单元（Docker 容器，虚拟机，裸机节点），Istio sidecar 需要同时安装到这些组件中。例如，如果你的基础架构使用虚拟机，则必须在每个需要成为 service mesh 一部分的虚拟机上运行 Istio sidecar 程序进程。

通过 Istio Sidecar 路由流量

sidecar 安装应该包括设置适当 iptable 规则，将应用程序的网络流量透明路由到 Istio sidecar。 可以在这里可以找到设置这种转发的 iptables 脚本。

注意：这个脚本必须在启动应用程序或sidecar进程之前执行。

> 注意：这个脚本必须在启动应用程序或 sidecar 进程之前执行。

Cloud Foundry

我们正在与 Cloud Foundry 开发人员合作，将 Istio 原生整合到 Cloud Foundry 平台中。访问 Istio Developers 邮件列表以获得更新。

安装

我们正在与 Cloud Foundry 开发人员合作，将 Istio 原生整合到 Cloud Foundry 平台中。访问 Istio Developers 邮件列表以获得更新。

Mesos

在 Apache Mesos 中安装 Istio 控制平面。

- 安装：如何在 Apache Mesos 中安装 Istio 控制平面的说明。

安装

目前 Istio 还不能原生支持 Mesos。然而，您可以利用我们的 Consul 集成，与 Consul 一起在 Mesos 上运行 Istio Mesh。更多详细信息，请参考 Consul 安装。

任务

任务展示如何用Istio系统实现一个单独特定的有目标的行为。

- 流量管理
  - 配置请求路由。这个任务展示如何基于权重和HTTP header配置动态请求路由。
  - 故障注入。这个任务展示如何注入延迟并测试应用的弹性。
  - 流量转移。这个任务展示如何将服务的流量从旧版本转移到新版本。
  - 设置请求超时。这个任务展示如何使用Istio在Envoy中设置请求超时。
  - Istio Ingress控制器。描述如何在Kubernetes上配置Istio Ingress控制器。
  - 控制Egress流量。描述如何控制Isto来路由流量，从mesh中的服务到外部服务。
  - 熔断。这个任务展示熔断能力以构建有弹性的应用
- 策略实施
  - 开启限流。这个任务展示如何使用Istio来动态限制到服务的流量
- Metrics，日志和跟踪
  - 分布式跟踪。如何配置代理，以便向Zipkin或Jaeger发送跟踪请求
  - 收集metrics和日志。这个任务展示如何配置Istio来收集metrics和日志。
  - 收集TCP服务的Metrics。这个任务展示如何为TCP服务收集metrics和日志。
  - 从Prometheus中查询Metrics。这个任务展示如何使用Prometheus查询metrics。
  - 使用Grafana可视化Metrics。这个任务展示如何安装并使用Istio的Dashboard来监控网格流量
  - 生成服务图。这个任务展示如何把Istio网格中的服务生成服务图

    - 使用Fluentd记录日志。这个任务展示如何配置Istio来将日志记录到Fluentd后台服务

- 安全

    - 验证Istio双向TLS认证。这个任务展示如何验证并测试Istio的自动交互TLS认证。
    - 配置基础访问控制。这个任务展示如何使用Kubernetes标签控制对服务的访问。
    - 配置安全访问控制。这个任务展示如何使用服务账号来安全的控制对服务的访问。
    - 启用每服务双向认证。这个任务展示如何为单个服务改变双向TLS认证。
    - 插入CA证书和密钥。这个任务展示运维人员如何插入已有证书和密钥到Istio CA中。

流量管理

用于展示Isito服务网格的流量路由特性的任务。

配置请求路由。这个任务展示如何基于权重和HTTP header配置动态请求路由。

故障注入。这个任务展示如何注入延迟并测试应用的弹性。

流量转移。这个任务展示如何将服务的流量从旧版本转移到新版本。

设置请求超时。这个任务展示如何使用Istio在Envoy中设置请求超时。

Istio Ingress控制器。描述如何在Kubernetes上配置Istio Ingress控制器。

控制Egress流量。描述如何控制Isto来路由流量，从mesh中的服务到外部服务。

熔断。这个任务展示熔断能力以构建有弹性的应用

镜像。演示Istio的流量镜像功能。

配置请求路由

此任务将演示如何根据权重和HTTP header配置动态请求路由。

前提条件

- 参照文档安装指南中的步骤安装Istio。
- 部署BookInfo 示例应用程序。

> 请注意：本文档假设示例应用程序通过kubernetes进行部署。所有的示例命令行都采用规则yaml文件（例如 `samples/bookinfo/kube/route-rule-all-v1.yaml` ）指定的kubernetes版本。如果在不同的环境下运行本任务，请将 `kube` 修改为运行环境中相应的目录（例如，对基于Consul的运行环境，目录就是 `samples/bookinfo/consul/route-rule-all-v1.yaml` ）。

基于内容的路由

BookInfo示例部署了三个版本的reviews服务，因此需要设置一个缺省路由。否则当多次访问该应用程序时，会发现有时输出会包含带星级的评价内容，有时又没有。出现该现象的原因是当没有为应用显式指定缺省路由时，Istio会将请求随机路由到该服务的所有可用版本上。

> 请注意：本文档假设还没有设置任何路由规则。如果已经为示例应用程序创建了存在冲突的路由规则，则需要在下面的命令中使用 `replace` 关键字代替 `create` 。

1. 将所有微服务的缺省版本设置为v1。

```
istioctl create -f samples/bookinfo/kube/route-rule-all-v1.yaml
```

> 请注意：在kubernetes中部署Istio时，可以在上面及其它所有命令行中用 `kubectl` 代替 `istioctl` 。但是目前 `kubectl` 不提供对命令输入参数的验证。

可以通过下面的命令来显示所有以创建的路由规则。

```
istioctl get routerules -o yaml
```

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: details-default
  namespace: default
  ...
spec:
  destination:
    name: details
  precedence: 1
  route:
  - labels:
      version: v1
---
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: productpage-default
  namespace: default
  ...
spec:
  destination:
    name: productpage
  precedence: 1
  route:
  - labels:
      version: v1
---
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-default
  namespace: default
  ...
spec:
```

```
  destination:
    name: ratings
  precedence: 1
  route:
  - labels:
      version: v1
---
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-default
  namespace: default
  ...
spec:
  destination:
    name: reviews
  precedence: 1
  route:
  - labels:
      version: v1
---
```

由于路由规则是通过异步方式分发到代理的，过一段时间后规则才会同步到所有pod上。因此需要等几秒钟后再尝试访问应用。

2. 在浏览器中打开BookInfo应用程序的URL(http://$GATEWAY_URL/productpage)。

   可以看到BookInfo应用程序的productpage页面。

   请注意 `productpage` 页面显示的内容中不包含带星的评价信息，这是为 `reviews:v1` 服务不会访问 `ratings` 服务。

3. 将来自特定用户的请求路由到 `reviews:v2` 。

   把来自测试用户"jason"的请求路由到 `reviews:v2` ，以启用 `ratings` 服务。

```
istioctl create -f samples/bookinfo/kube/route-rule-reviews-test-v2.yaml
```

确认规则已创建：

```
istioctl get routerule reviews-test-v2 -o yaml
```

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-test-v2
  namespace: default
  ...
spec:
  destination:
    name: reviews
  match:
    request:
      headers:
        cookie:
          regex: ^(.*?;)?(user=jason)(;.*)?$
  precedence: 2
  route:
  - labels:
      version: v2
```

4. 以"jason"用户登录 `productpage` 页面。

   此时应该可以在每条评价后面看到星级信息。请注意如果以别的用户登录，还是只能看到 `reviews:v1` 版本服务呈现出的内容，即不包含星级信息的内容。

理解原理

在这个任务中，我们首先使用Istio将100%的请求流量都路由到了BookInfo服务的v1版本。 然后再设置了一条路由规则，该路由规则基于请求的header（例如一个用户cookie）选择性地将特定的流量路由到了reviews服务的v2版本。

一旦reviews服务的v2版本经过测试后满足要求，我们就可以使用Istio将来自所有用户的流量一次性或者渐进地路由到v2版本。我们将在另一个单独的任务中对此进行尝试。

清理

- 删除路由规则

```
istioctl delete -f samples/bookinfo/kube/route-rule-all-v1.yaml
istioctl delete -f samples/bookinfo/kube/route-rule-reviews-test-v2.yaml
```

- 如果不打算尝试后面的任务，请参照BookInfo cleanup 中的步骤关闭应用程序。

下一步

- 更多的内容请参见请求路由。

故障注入

本任务将演示如何注入延迟并测试应用弹性。

开始之前

- 参考文档安装指南中的步骤安装Istio。
- 部署BookInfo示例应用。
- 首先通过请求路由任务，或通过执行下列命令，来初始化应用的版本路由信息：

  > 注意：这里假设尚未设置任何路由。如果已经为示例创建了存在冲突的路由规则，则需要在下列两条命令或其中之一使用 `replace` 代替 `create` 。

  ```
  istioctl create -f samples/bookinfo/kube/route-rule-all-v1.yaml
  istioctl create -f samples/bookinfo/kube/route-rule-reviews-test-v2.yaml
  ```

> 注意：本任务假设将通过Kubernetes来部署应用。所有的示例命令都采用规则yaml文件（如 `samples/bookinfo/kube/route-rule-all-v1.yaml` ）指定的Kubernetes版本。如果在不同的环境下运行此任务，请将 `kube` 修改为运行环境中（比如基于Consul运行环境就是 `samples/bookinfo/consul/route-rule-all-v1.yaml` ）相应的目录。

故障注入

为了测试BookInfo微服务应用的弹性，我们计划针对"jason"用户在reviews:v2和ratings服务之间 注入7秒的延迟 。由于 *reviews:v2* 服务针对调用ratings服务设置了10秒的超时，因此期望端到端的流程能无错持续。

1. 创建一个故障注入规则，来延迟来自用户"jason"（本次的测试用户）的流量：

```
istioctl create -f samples/bookinfo/kube/route-rule-ratings-
test-delay.yaml
```

确认规则已创建：

```
istioctl get routerule ratings-test-delay -o yaml
```

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: ratings-test-delay
  namespace: default
  ...
spec:
  destination:
    name: ratings
  httpFault:
    delay:
      fixedDelay: 7.000s
      percent: 100
  match:
    request:
      headers:
        cookie:
          regex: ^(.*?;)?(user=jason)(;.*)?$
  precedence: 2
  route:
  - labels:
      version: v1
```

规则分发延迟到所有的pod需要数秒钟的时间。

2. 观察应用行为

使用用户"jason"登录。如果应用的首页已设置正确地处理延迟，那首页将会在约7秒钟内加载完。为了解网页的响应时间，请打开IE、Chrome或Firefox（一般快捷键是*Ctrl+Shift+I* 或 *Alt+Cmd+I*）的*Developer Tools*菜单，切换到

Network标签，并reload `productpage` 页面。

将会看到网页在约6秒钟内加载完成。review部分将会显示 *Sorry, product reviews are currently unavailable for this book*。

理解原理

整个review服务失败的原因是，BookInfo应用有一个bug。productpage和review服务之间的超时小于（3秒加上一次重试，总共6秒）review服务和rating服务之间的超时（10秒）。在由不同开发团队负责独立开发不同微服务的典型企业应用中，这类bug就会发生。Istio的故障注入规则有助于识别这些异常，而无需影响到最终用户。

> 需要注意的是，这里只针对用户"jason"设置了故障影响。如果使用其他用户登录，则不会有任何延迟。

解决**bug:** 至此，按理来说已经能解决这个问题了，通过增加productpage的超时，或者减少review服务到rating服务的超时，终止并重启相关的微服务，然后确认 `productpage` 不报错并正常返回结果即可。

不过，在review服务的v3版本中已经解决了此问题，所以只要按照traffic shifting 任务所述，转移所有流量到 `reviews:v3` 就能简单地修复问题。

（这里为读者留一个练习——修改延迟规则，改为2.8秒的延迟，然后运行v3版本的review服务。）

清除

- 清除应用的路由规则：

```
  istioctl delete -f samples/bookinfo/kube/route-rule-all-v1
.yaml
  istioctl delete -f samples/bookinfo/kube/route-rule-review
s-test-v2.yaml
  istioctl delete -f samples/bookinfo/kube/route-rule-rating
s-test-delay.yaml
```

- 如果并不计划了解更多后续任务，参考BookInfo cleanup的说明来关闭应用。

进阶阅读

- 了解更多关于故障注入的内容。

流量转移

本任务将演示如何将应用流量逐渐从旧版本的服务迁移到新版本。通过Istio，可以使用一系列不同权重的规则（10%，20%，… 100%）将流量平缓地从旧版本服务迁移到新版本服务。为简单起见，本任务将采用两步将流量从 `reviews:v1` 迁移到 `reviews:v3` ，权重分别为50%，100%。

开始之前

- 参照文档安装指南中的步骤安装Istio。
- 部署BookInfo示例应用程序。

> 请注意：本文档假设示采用kubernetes部署示例应用程序。所有的示例命令都采用规则yaml文件（例如 `samples/bookinfo/kube/route-rule-all-v1.yaml` ）指定的kubernetes版本。如果在不同的环境下运行本任务，请将 `kube` 修改为运行环境中相应的目录（例如，对基于Consul的运行环境，目录就是 `samples/bookinfo/consul/route-rule-all-v1.yaml` ）。

基于权重的版本路由

1. 将所有微服务的缺省版本设置为v1.

   ```
   istioctl create -f samples/bookinfo/kube/route-rule-all-v1.yaml
   ```

2. 在浏览器中打开http://$GATEWAY_URL/productpage, 确认 `reviews` 服务目前的活动版本是v1。

   可以看到BooInfo应用的productpage页面被显示出来。注意 `productpage` 显示的评价内容不带星级，因为 `reviews:v1` 不会访问 `ratings` 服务。

   > 注意：如果之前执行过 配置请求路由任务，则需要先注销测试用户“jason”或者删除之前单独为该用户创建的测试规则：

```
  istioctl delete routerule reviews-test-v2
```

3. 首先，使用下面的命令把50%的流量从 `reviews:v1` 转移到 `reviews:v3`：

```
istioctl replace -f samples/bookinfo/kube/route-rule-reviews
-50-v3.yaml
```

注意这里使用了 `istioctl replace` 而不是 `create`。

确认规则被替换：

```
istioctl get routerule reviews-default -o yaml
```

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-default
  namespace: default
spec:
  destination:
    name: reviews
  precedence: 1
  route:
  - labels:
      version: v1
    weight: 50
  - labels:
      version: v3
    weight: 50
```

4. 在浏览器中多次刷新 `productpage` 页面，大约有50%的几率会看到页面中出现带红星的评价内容。

> 请注意：在目前的Envoy sidecar实现中，可能需要刷新 `productpage` 很多次才能看到流量分发的效果。在看到页面出现变化前，有可能需要刷新15次或者更多。如果修改规则，将90%的流量路由到v3，可以看到更明显的效果。

5. 当v3版本的 `reviews` 服务已经稳定运行后，可以将100%的流量路由到 `reviews:v3` ：

   ```
   istioctl replace -f samples/bookinfo/kube/route-rule-reviews
   -v3.yaml
   ```

   此时，以任何用户登录到 `productpage` 页面，都可以看到带红星的评价信息。

理解原理

在这个任务中，我们使用了Istio的带权重路由的特性将流量从老版本的 `reviews` 服务逐渐迁移到了新版本服务中。

注意该方式和使用容器编排平台的部署特性来进行版本迁移是完全不同的。容器编排平台使用了实例scaling来对流量进行管理。而通过Istio，两个版本的 `reviews` 服务可以独立地进行scale up和scale down，并不会影响这两个版本服务之间的流量分发。

想了解更多支持scaling的按版本路由功能，请查看Canary Deployments using Istio。

清理

- 删除路由规则。

  ```
  istioctl delete -f samples/bookinfo/kube/route-rule-all-v1.y
  aml
  ```

- 如果不打算尝试后面的任务，请参照BookInfo cleanup 中的步骤关闭应用程序。

进阶阅读

- 更多的内容请参见请求路由。

设置请求超时

这个任务用于展示如何使用Istio在Envoy中设置请求超时。

开始之前

- 遵循安装指南 的指导安装Istio。
- 部署BookInfo 示例程序。
- 执行如下命令，初始化基于应用版本的路由。

```
  istioctl create -f samples/bookinfo/kube/route-rule-all-v1
.yaml
```

> 注意：本任务假设在Kubernetes上部署这一应用。所有涉及的命令都使用的是Kubernetes版本的yaml文件（例如 `samples/bookinfo/kube/route-rule-all-v1.yaml` ）。如果要在不同环境运行这一任务，则需要把路径中的 `kube` 替换成对应的环境（例如 `samples/bookinfo/consul/route-rule-all-v1.yaml` ）。

请求超时

http的请求超时可以在路由规则中的 `httpReqTimeout` 字段来设置。缺省的超时时间是15秒，在下面我们会把 `reviews` 服务的超时时间设置为一秒钟。为了展示效果，我们还需要在调用 `ratings` 服务的时候加入两秒钟的延迟。

1. 路由请求到 `reviews` 服务的v2版本，例如，这个版本的 `reviews` 服务会调用 `ratings` 服务

```
cat <<EOF | istioctl replace -f -
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
 name: reviews-default
spec:
 destination:
   name: reviews
 route:
 - labels:
     version: v2
EOF
```

2. 对 `ratings` 服务的调用加入两秒钟的延迟

```
cat <<EOF | istioctl replace -f -
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
 name: ratings-default
spec:
 destination:
   name: ratings
 route:
 - labels:
     version: v1
 httpFault:
   delay:
     percent: 100
     fixedDelay: 2s
EOF
```

3. 在浏览器中打开 BookInfo 的网址（`http://$GATEWAY_URL/productpage`）。

   会看到BookInfo应用在正常运行（并且显示了评级的星星），但是你也会注意到，每次刷新页面的时候会有两秒钟的延迟。

4. 接下来我们给对 `reviews` 的调用加入一秒钟的请求超时

```
cat <<EOF | istioctl replace -f -
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
 name: reviews-default
spec:
 destination:
   name: reviews
 route:
 - labels:
     version: v2
 httpReqTimeout:
   simpleTimeout:
     timeout: 1s
EOF
```

5. 刷新BookInfo页面

   现在会看到，只有一秒钟就刷新完成（而不是之前的两秒钟），但是Reviews部分不再可用。

幕后故事

这一任务中，使用Istio为 `reviews` 服务设置了一秒钟的请求超时（而不是缺省的15 秒）。由于 `reviews` 服务在处理请求的时候会调用下游的 `ratings` 服务，而 `ratings` 服务已经被我们利用Istio注入了两秒钟的延迟，这一变动会让 `reviews` 服务的处理时间超过一秒，故而引发了超时。

你会看到BookInfo productpage（需要调用 `reviews` 服务来生成页面）不会显示review信息，而是显示为：“Sorry, product reviews are currently unavailable for this book. ”，这是 `reviews` 服务的超时引发的错误造成的结果。

如果检查一下错误注入任务，会发现 `productpage` 服务在调用 `reviews` 的时候，也有自己的应用级别的超时（三秒钟）。我们前面的测试中使用了一秒钟的超时，如果我们把下游服务的超时时间设置为大于三秒钟的值（例如四秒），由于三

秒钟的限制更为严格，会被优先处理，因此过大的超时时间就不会生效了。参看错误处理一节会有更详细的信息。

还需要注意的就是，除了在路由规则中设置超时之外，还可以在请求中加入“x-envoy-upstream-rq-timeout-ms”头来设置超时，在这一设置中的时间单位是毫秒而不是秒。

清理

- 删除应用路由规则。

```
  istioctl delete -f samples/bookinfo/kube/route-rule-all-v1
.yaml
```

- 如果没有计划进一步运行下面的任务，可以参照BookInfo 清理 中的介绍来关闭这一应用。

延伸阅读

- 更多关于错误处理
- 更多关于路由规则

Istio Ingress控制器

此任务将演示如何通过配置Istio将服务发布到service mesh集群外部。在Kubernetes环境中，Kubernetes Ingress Resources 允许用户指定某个服务是否要公开到集群外部。然而，Ingress Resource规范非常精简，只允许用户设置主机，路径，以及后端服务。下面是Istio ingress已知的局限：

1. Istio支持不使用annotation的标准Kubernetes Ingress规范。不支持Ingress资源规范中的 `ingress.kubernetes.io` annotation。 `kubernetes.io/ingress.class: istio` 之外的任何annotation将被忽略。
2. 不支持路径中的正则表达式
3. Ingress中不支持故障注入

前提条件

- 参照文档安装指南中的步骤安装Istio。
- 确保当前的目录是 `istio` 目录。
- 启动 httpbin 示例, 我们会把这个服务作为目标服务发布到外部。

  如果你安装了Istio-Initializer, 请执行：

  ```
  kubectl apply -f samples/httpbin/httpbin.yaml
  ```

  如果没有Istio-Initializer, 请执行：

  ```
  kubectl apply -f <(istioctl kube-inject -f samples/httpbin
  /httpbin.yaml)
  ```

配置ingress (HTTP)

1. 为httpbin服务创建一个基本的Ingress Resource

```
cat <<EOF | kubectl create -f -
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: simple-ingress
  annotations:
    kubernetes.io/ingress.class: istio
spec:
  rules:
  - http:
      paths:
      - path: /status/.*
        backend:
          serviceName: httpbin
          servicePort: 8000
      - path: /delay/.*
        backend:
          serviceName: httpbin
          servicePort: 8000
EOF
```

`/.*` 是特殊的Istio表示方法，用于表示前缀匹配，特别是(prefix: /)形式的[规则匹配配置]（前缀：/）。

验证 ingress

1. 确定ingress URL：
    - 如果你的集群运行的环境支持外部的负载均衡器，请使用ingress的外部地址：

      ```
      kubectl get ingress simple-ingress -o wide
      ```

```
NAME             HOSTS     ADDRESS                 PORTS
    AGE
simple-ingress   *         130.211.10.121          80
    1d
```

```
export INGRESS_HOST=130.211.10.121
```

- 如果不支持负载均衡器，使用ingress控制器的pod的hostIP：

```
kubectl get po -l istio=ingress -o jsonpath='{.items[0].s
tatus.hostIP}'
```

```
169.47.243.100
```

同时也找到istio-ingress服务的80端口的nodePort映射端口:

```
kubectl -n istio-system get svc istio-ingress
```

```
NAME            CLUSTER-IP     EXTERNAL-IP   PORT(S)
                 AGE
istio-ingress   10.10.10.155   <pending>     80:31486/TCP
,443:32254/TCP   32m
```

```
export INGRESS_HOST=169.47.243.100:31486
```

2. 使用*curl*访问httpbin服务：

```
curl -I http://$INGRESS_HOST/status/200
```

```
HTTP/1.1 200 OK
server: envoy
date: Mon, 29 Jan 2018 04:45:49 GMT
content-type: text/html; charset=utf-8
access-control-allow-origin: *
access-control-allow-credentials: true
content-length: 0
x-envoy-upstream-service-time: 48
```

3. 如果访问其他的没有明确公开的URL，应该收到HTTP404错误

```
curl -I http://$INGRESS_HOST/headers
```

```
HTTP/1.1 404 Not Found
date: Mon, 29 Jan 2018 04:45:49 GMT
server: envoy
content-length: 0
```

配置安全ingress(HTTPS)

1. 为ingress生成证书和密钥

   使用OpenSSL创建测试私钥和证书

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
/tmp/tls.key -out /tmp/tls.crt -subj "/CN=foo.bar.com"
```

2. 创建secret

   使用 `kubectl` 在命名空间 `istio-system` 创建 `istio-ingress-certs` secret。

   > 注意：secret在命名空间 `istio-system` 中必须被称为 `istio-ingress-certs` ，因为它要被加载到Isito Ingress。

```
kubectl create -n istio-system secret tls istio-ingress-cert
s --key /tmp/tls.key --cert /tmp/tls.crt
```

3.为httpbin服务创建Ingress Resource

```
```bash
cat <<EOF | kubectl create -f -
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: secure-ingress
  annotations:
    kubernetes.io/ingress.class: istio
spec:
  tls:
    - secretName: istio-ingress-certs # currently ignored
  rules:
  - http:
      paths:
      - path: /status/.*
        backend:
          serviceName: httpbin
          servicePort: 8000
      - path: /delay/.*
        backend:
          serviceName: httpbin
          servicePort: 8000
EOF
```
```

> 注意: Envoy当前只允许一个TLS ingress密钥，因为SNI 尚未支持。也就是说看，ingress 中的 secretName 字段并没有被使用。

再次验证ingress

> 译者注：注意这里的命令和输出与前面的稍有不同。

1. 确定ingress URL：
    - 如果你的集群运行的环境支持外部的负载均衡器，请使用ingress的外部地址：

```
kubectl get ingress secure-ingress -o wide
```

```
NAME             HOSTS     ADDRESS                 PORTS
    AGE
secure-ingress   *         130.211.10.121          80
    1d
```

```
export INGRESS_HOST=130.211.10.121
```

    - 如果不支持负载均衡器，使用ingress控制器的pod的hostIP：

```
kubectl -n istio-system get po -l istio=ingress -o jsonpa
th='{.items[0].status.hostIP}'
```

```
169.47.243.100
```

同时也找到istio-ingress服务的80端口的nodePort映射端口:

```
kubectl -n istio-system get svc istio-ingress
```

```
NAME            CLUSTER-IP     EXTERNAL-IP   PORT(S)
                 AGE
istio-ingress   10.10.10.155   <pending>     80:31486/TCP
,443:32254/TCP   32m
```

```
export INGRESS_HOST=169.47.243.100:31486
```

2. 使用*curl*访问httpbin服务：

```
curl -I http://$INGRESS_HOST/status/200
```

```
HTTP/1.1 200 OK
server: envoy
date: Mon, 29 Jan 2018 04:45:49 GMT
content-type: text/html; charset=utf-8
access-control-allow-origin: *
access-control-allow-credentials: true
content-length: 0
x-envoy-upstream-service-time: 96
```

3. 如果访问其他的没有明确公开的URL，应该收到HTTP404错误

```
curl -I http://$INGRESS_HOST/headers
```

```
HTTP/1.1 404 Not Found
date: Mon, 29 Jan 2018 04:45:49 GMT
server: envoy
content-length: 0
```

和ingress一起使用istio路由规则

Istio的路由规则可以在路由请求到后端服务时获取更大的控制度。例如，下列路由规则为到httpbin服务上的 `/delay` URL的所有请求设置4秒的超时时间。

```
cat <<EOF | istioctl create -f -
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: status-route
spec:
  destination:
    name: httpbin
  match:
    # Optionally limit this rule to istio ingress pods only
    source:
      name: istio-ingress
      labels:
        istio: ingress
    request:
      headers:
        uri:
          prefix: /delay/ #must match the path specified in ingr
ess spec
              # if using prefix paths (/delay/.*), omit the .*.
              # if using exact match, use exact: /status
  route:
  - weight: 100
  httpReqTimeout:
    simpleTimeout:
      timeout: 4s
EOF
```

如果用URL `http://$INGRESS_HOST/delay/10` 来发起对ingress的调用，会发现调用在4秒之后返回，而不是期待的10秒延迟。

可以使用路由规则的其他特性如重定向，重写，路由到多个版本，基于HTTP header的正则表达式匹配，websocket升级，超时，重试，等。更多详情请参考路由规则。

> 注意1： 在Ingress中故障注入不可用 注意2： 当请求匹配路由规则时，使用完全相同的路径或者前缀，就像在Ingress规范中使用的那样。

理解Ingress

Ingress为外部流量提供网关来进入Istio服务网格，并使得Istio的流量管理和策略的特性对edge服务可用。

Ingress规范中的servicePort字段可以是一个端口数字（整型）或者一个名称。为了正确工作，端口名称必须遵循Istio端口命名约定(例如，`grpc-*`，`http2-*`，`http-*`，等)。使用的名称必须匹配后端服务声明中的端口名称。

在前面的步骤中，我们在Isito服务网格中创建服务，并展示了如何暴露服务的HTTP和HTTPS终端到外部流量。也展示了如何使用Istio路由规则来控制ingress流量。

清理

1. 删除secret和Ingress Resource定义

```
kubectl delete ingress simple-ingress secure-ingress
kubectl delete -n istio-system secret istio-ingress-certs
```

2. 关闭httpbin服务

```
kubectl delete -f samples/httpbin/httpbin.yaml
```

进阶阅读

- 进一步了解和学习Ingress Resources.
- 进一步了解和学习routing rules.

控制Egress流量

缺省情况下，启用了Istio的服务是无法访问外部URL的，这是因为Pod中的iptables把所有外发传输都转向到了Sidecar代理，而这一代理只处理集群内的访问目标。

本节内容会描述如何把外部服务提供给启用了Istio的客户端服务使用，你会学到如何使用Egress规则访问外部服务，或者如何简单的让特定IP范围穿透Istio代理。

开始之前

- 遵循安装指南设置Istio
- 启动sleep示例，用于测试外部访问。

```
  kubectl apply -f <(istioctl kube-inject -f samples/sleep/s
leep.yaml)
```

注意所有可以在其中执行 `exec` 和 `curl` 的pod都可以这样做。

使用Istio的Egress规则

使用Egress规则，就可以从Istio集群内访问任何公开服务。本任务中我们会使用httpbin.org以及www.google.com作为范例。

配置外部服务

1. 创建一个Egress规则，来允许访问外部HTTP服务：

```
cat <<EOF | istioctl create -f -
apiVersion: config.istio.io/v1alpha2
kind: EgressRule
metadata:
  name: httpbin-egress-rule
spec:
  destination:
    service: httpbin.org
  ports:
    - port: 80
      protocol: http
EOF
```

2. 创建一个Egress规则，容许访问外部HTTPS服务：

```
cat <<EOF | istioctl create -f -
apiVersion: config.istio.io/v1alpha2
kind: EgressRule
metadata:
  name: google-egress-rule
spec:
  destination:
    service: www.google.com
  ports:
    - port: 443
      protocol: https
EOF
```

发出对外请求

1. 使用 `exec` 进入作为测试源使用的pod。例如，如果你正在使用sleep服务，运行下列命令：

```
 export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpat
h={.items..metadata.name})
 kubectl exec -it $SOURCE_POD -c sleep bash
```

2. 发出请求到外部HTTP服务：

```
curl http://httpbin.org/headers
```

3. 发送请求到外部HTTPS服务

   外部的HTTPS服务必须可以通过HTTP访问，在请求中指定端口：

```
curl http://www.google.com:443
```

为外部服务设置路由规则

和集群内请求类似，Istio的路由规则也可以用于设置使用了Egress规则的外部服务。为了进行演示，我们为httpbin.org服务设置一个超时。

1. 在作为测试源使用的Pod内，调用外部服务httpbin.org的 `/delay` 端点：

```
kubectl exec -it $SOURCE_POD -c sleep bash
time curl -o /dev/null -s -w "%{http_code}\n" http://httpbin.org/delay/5
```

```
200

real    0m5.024s
user    0m0.003s
sys     0m0.003s
```

   请求应该在大约五秒钟之后返回200(OK)。

2. 退出源Pod，再使用 `istioctl` 来给外部服务httpbin.org设置3秒钟的调用超时：

```
cat <<EOF | istioctl create -f -
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: httpbin-timeout-rule
spec:
  destination:
    service: httpbin.org
  http_req_timeout:
    simple_timeout:
      timeout: 3s
EOF
```

3. 等待几秒钟之后，再次发起*curl*调用：

```
kubectl exec -it $SOURCE_POD -c sleep bash
time curl -o /dev/null -s -w "%{http_code}\n" http://httpbi
n.org/delay/5
```

```
504

real    0m3.149s
user    0m0.004s
sys     0m0.004s
```

这次在大概3秒钟之后返回504（网关超时）。虽然httpbin.org在等待5秒钟，Istio在3秒钟时就切断了请求。

直接调用外部服务

目前Istio的Egress规则只提供对HTTP/HTTPS请求的支持。如果想要访问其他协议的外部服务（例如mongodb://host/database），或者让指定IP范围直接穿透Istio，就需对源服务的Envoy Sidecar进行配置，阻止其对外部请求的拦截。可以在使用istio kube-inject时启用`--includeIPRanges`参数来满足这一要求。

最简单的 `--includeIPRanges` 用法就是提交所有的内部服务的IP范围，然后让所有外部服务地址都绕过Sidecar代理。内部IP范围跟你所运行的集群有关，例如Minikube中的范围是 `10.0.0.1/24` ，所以应该这样启动sleep服务：

```
kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.y
aml --includeIPRanges=10.0.0.1/24)
```

On IBM Cloud Private, use:

1. 从 `cluster/config.yaml` 下的IBM Cloud Private配置文件中获取 `service_cluster_ip_range`

   ```
   cat cluster/config.yaml | grep service_cluster_ip_range
   ```

   下面是输出示例：

   ```
   service_cluster_ip_range: 10.0.0.1/24
   ```

2. 通过 `--includeIPRanges` ，注入 `service_cluster_ip_range` 到应用的profile，来限制Isito的流量拦截到service cluster IP range。

   ```
   kubectl apply -f <(istioctl kube-inject -f samples/sleep/sl
  eep.yaml --includeIPRanges=10.0.0.1/24)
   ```

在IBM Cloud Container Service上，使用：

```
kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.y
aml --includeIPRanges=172.30.0.0/16,172.20.0.0/16,10.10.10.0/24)
```

在Google容器引擎（GKE）上的IP范围不是固定的，所以需要运行命令 `gcloud container clusters describe` 来获取要使用的IP范围。例如：

```
gcloud container clusters describe XXXXXXX --zone=XXXXXX | grep
-e clusterIpv4Cidr -e servicesIpv4Cidr
```

```
clusterIpv4Cidr: 10.4.0.0/14
servicesIpv4Cidr: 10.7.240.0/20
```

```
kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.y
aml --includeIPRanges=10.4.0.0/14,10.7.240.0/20)
```

而在Azure容器服务（ACS）上就应该执行：

```
kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.y
aml --includeIPRanges=10.244.0.0/16,10.240.0.0/16)
```

以这种方式启动服务之后，Istio Sidecar就只会处理和管理集群内的请求，所有的外部请求就会简单的绕开Sidecar直接访问目标了：

```
export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath={.i
tems..metadata.name})
kubectl exec -it $SOURCE_POD -c sleep curl http://httpbin.org/he
aders
```

幕后故事

这个任务中我们在Istio集群中使用两种方式来访问外部服务：

1. 使用Egress规则（推荐）。
2. 配置Istio Sidecar，在他的iptables中排除对外部IP的控制。

第一种方式目前只支持HTTP(S)请求，但是可以为外部服务提供和内部服务一样的路由支持。我们使用了一个路由规则来展示了这一能力。

第二种方式绕过了Sidecar代理，让服务可以直接访问任何外部资源，只是这样的设置需要一点对集群配置的了解。

清理

1. 删除规则：

```
 istioctl delete egressrule httpbin-egress-rule google-egres
s-rule
 istioctl delete routerule httpbin-timeout-rule
```

2. 关闭sleep服务：

```
 kubectl delete -f samples/sleep/sleep.yaml
```

延伸阅读

- 关于Egress规则的更多内容。
- 如何为Egress通信设置超时、重试以及断路器。

断路器

本文中的这一任务展示了弹性应用的熔断能力。开发人员可以凭借这一能力，来限制因为故障、延迟高峰以及其他预计外的网络异常所造成的影响范围。下面将会延时如何针对连接、请求以及外部检测来进行断路器配置。

开始之前

- 遵循安装指南设置Istio。
- 启动httpbin实例作为本任务的后端：

```
 kubectl apply -f <(istioctl kube-inject -f
samples/httpbin/httpbin.yaml)
```

断路器

接下来设置一个场景来演示熔断能力。现在我们已经运行了 `httpbin` 服务。我们想要通过对istio目标策略的配置来设置一个熔断规则。在这之前首先要设置一个到这一目的的路由规则，这一规则会把所有访问 `httpbin` 的流量路由到 `version=v1` 去。

创建断路策略

1. 创建缺省路由规则，让所有目标为 `httpbin` 的流量都指向 `v1` ：

   istioctl create -f samples/httpbin/routerules/httpbin-v1.yaml

2. 创建目标策略来对 `httpbin` 的调用过程进行断路设置。

```
cat <<EOF | istioctl create -f -
apiVersion: config.istio.io/v1beta1
kind: DestinationPolicy
metadata:
 name: httpbin-circuit-breaker
spec:
 destination:
   name: httpbin
   labels:
     version: v1
 circuitBreaker:
   simpleCb:
     maxConnections: 1
     httpMaxPendingRequests: 1
     sleepWindow: 3m
     httpDetectionInterval: 1s
     httpMaxEjectionPercent: 100
     httpConsecutiveErrors: 1
     httpMaxRequestsPerConnection: 1
EOF
```

3. 确认目标策略是否正确创建

```
istioctl get destinationpolicy

NAME                    KIND
           NAMESPACE
httpbin-circuit-breaker DestinationPolicy.v1alpha2.config.is
tio.io      istio-samples
```

设置客户端

接下来我们来定义调用 `httpbin` 服务的规则。要创建一个客户端，用于向我们的服务发送流量，从而进一步体验断路器功能。我们会使用一个简单的负载测试工具fortio。使用这个客户端我们可以控制连接数量、并发数以及外发HTTP调用的延迟。这一步骤中，我们会设置一个客户端，注入istio sidecar，以此保证客户端的通信也在istio的监管之下：

```
kubectl apply -f <(istioctl kube-inject -f samples/httpbin/sampl
e-client/fortio-deploy.yaml)
```

这样我们就能够登入客户端 Pod，并使用fortio工具来调用 `httpbin` ，使用 `-curl` 开关可以指定我们只执行一次调用。

```
FORTIO_POD=$(kubectl get pod | grep fortio | awk '{ print $1 }')
kubectl exec -it $FORTIO_POD  -c fortio /usr/local/bin/fortio --
 load -curl  http://httpbin:8000/get

HTTP/1.1 200 OK
server: envoy
date: Tue, 16 Jan 2018 23:47:00 GMT
content-type: application/json
access-control-allow-origin: *
access-control-allow-credentials: true
content-length: 445
x-envoy-upstream-service-time: 36

{
  "args": {},
  "headers": {
    "Content-Length": "0",
    "Host": "httpbin:8000",
    "User-Agent": "istio/fortio-0.6.2",
    "X-B3-Sampled": "1",
    "X-B3-Spanid": "824fbd828d809bf4",
    "X-B3-Traceid": "824fbd828d809bf4",
    "X-Ot-Span-Context": "824fbd828d809bf4;824fbd828d809bf4;0000
000000000000",
    "X-Request-Id": "1ad2de20-806e-9622-949a-bd1d9735a3f4"
  },
  "origin": "127.0.0.1",
  "url": "http://httpbin:8000/get"
}
```

这里可以看到，调用成功了！接下来我们试试熔断功能。

测试断路器

在断路器设置中，我们指定 `maxConnections: 1` 以及 `httpMaxPendingRequests: 1` 。这样的设置下，如果我们并发超过一个连接和请求，istio-proxy就会断掉后续的请求和连接。我们试试两个并发链接（ `-c 2` ），发送20个请求（ `-n 20` ）：

```
kubectl exec -it $FORTIO_POD  -c fortio /usr/local/bin/fortio --
 load -c 2 -qps 0 -n 20 -loglevel Warning http://httpbin:8000/ge
t

Fortio 0.6.2 running at 0 queries per second, 2->2 procs, for 5s
: http://httpbin:8000/get
Starting at max qps with 2 thread(s) [gomax 2] for exactly 20 ca
lls (10 per thread + 0)
23:51:10 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
Ended after 106.474079ms : 20 calls. qps=187.84
Aggregated Function Time : count 20 avg 0.010215375 +/- 0.003604
 min 0.005172024 max 0.019434859 sum 0.204307492
# range, mid point, percentile, count
>= 0.00517202 <= 0.006 , 0.00558601 , 5.00, 1
> 0.006 <= 0.007 , 0.0065 , 20.00, 3
> 0.007 <= 0.008 , 0.0075 , 30.00, 2
> 0.008 <= 0.009 , 0.0085 , 40.00, 2
> 0.009 <= 0.01 , 0.0095 , 60.00, 4
> 0.01 <= 0.011 , 0.0105 , 70.00, 2
> 0.011 <= 0.012 , 0.0115 , 75.00, 1
> 0.012 <= 0.014 , 0.013 , 90.00, 3
> 0.016 <= 0.018 , 0.017 , 95.00, 1
> 0.018 <= 0.0194349 , 0.0187174 , 100.00, 1
# target 50% 0.0095
# target 75% 0.012
# target 99% 0.0191479
# target 99.9% 0.0194062
Code 200 : 19 (95.0 %)
Code 503 : 1 (5.0 %)
Response Header Sizes : count 20 avg 218.85 +/- 50.21 min 0 max
231 sum 4377
Response Body/Total Sizes : count 20 avg 652.45 +/- 99.9 min 217
 max 676 sum 13049
All done 20 calls (plus 0 warmup) 10.215 ms avg, 187.8 qps
```

会看到几乎所有请求都通过了。

```
Code 200 : 19 (95.0 %)
Code 503 : 1 (5.0 %)
```

这是因为istio-proxy允许一定的误差。我们把连接数提高到3：

```
kubectl exec -it $FORTIO_POD  -c fortio /usr/local/bin/fortio --
 load -c 3 -qps 0 -n 20 -loglevel Warning http://httpbin:8000/ge
t

Fortio 0.6.2 running at 0 queries per second, 2->2 procs, for 5s
: http://httpbin:8000/get
Starting at max qps with 3 thread(s) [gomax 2] for exactly 30 ca
lls (10 per thread + 0)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
23:51:51 W http.go:617> Parsed non ok code 503 (HTTP/1.1 503)
Ended after 71.05365ms : 30 calls. qps=422.22
Aggregated Function Time : count 30 avg 0.0053360199 +/- 0.00421
9 min 0.000487853 max 0.018906468 sum 0.160080597
# range, mid point, percentile, count
>= 0.000487853 <= 0.001 , 0.000743926 , 10.00, 3
> 0.001 <= 0.002 , 0.0015 , 30.00, 6
> 0.002 <= 0.003 , 0.0025 , 33.33, 1
> 0.003 <= 0.004 , 0.0035 , 40.00, 2
> 0.004 <= 0.005 , 0.0045 , 46.67, 2
> 0.005 <= 0.006 , 0.0055 , 60.00, 4
> 0.006 <= 0.007 , 0.0065 , 73.33, 4
> 0.007 <= 0.008 , 0.0075 , 80.00, 2
> 0.008 <= 0.009 , 0.0085 , 86.67, 2
> 0.009 <= 0.01 , 0.0095 , 93.33, 2
> 0.014 <= 0.016 , 0.015 , 96.67, 1
```

```
> 0.018 <= 0.0189065 , 0.0184532 , 100.00, 1
# target 50% 0.00525
# target 75% 0.00725
# target 99% 0.0186345
# target 99.9% 0.0188793
Code 200 : 19 (63.3 %)
Code 503 : 11 (36.7 %)
Response Header Sizes : count 30 avg 145.73333 +/- 110.9 min 0 m
ax 231 sum 4372
Response Body/Total Sizes : count 30 avg 507.13333 +/- 220.8 min
 217 max 676 sum 15214
All done 30 calls (plus 0 warmup) 5.336 ms avg, 422.2 qps
```

这样就看到，断路器开始发挥作用了：

```
Code 200 : 19 (63.3 %)
Code 503 : 11 (36.7 %)
```

只有63%的请求通过了，其他的都被断路器拒之门外，我们可以查询一下istio-proxy的统计数据：

```
kubectl exec -it $FORTIO_POD  -c istio-proxy  -- sh -c 'curl loc
alhost:15000/stats' | grep httpbin | grep pending

cluster.out.httpbin.springistio.svc.cluster.local|http|version=v
1.upstream_rq_pending_active: 0
cluster.out.httpbin.springistio.svc.cluster.local|http|version=v
1.upstream_rq_pending_failure_eject: 0
cluster.out.httpbin.springistio.svc.cluster.local|http|version=v
1.upstream_rq_pending_overflow: 12
cluster.out.httpbin.springistio.svc.cluster.local|http|version=v
1.upstream_rq_pending_total: 39
```

`upstream_rq_pending_overflow` 的值为 `12` ，说明有 `12` 次调用被断路器拦截了。

清理

1. 清除规则。

```
istioctl delete routerule httpbin-default-v1
istioctl delete destinationpolicy httpbin-circuit-breaker
```

2. 关闭httpbin服务和客户端。

```
kubectl delete deploy httpbin fortio-deploy
kubectl delete svc httpbin
```

下一步

阅读目标策略参考，其中会有更详尽的断路器相关内容。

策略实施

描述演示策略实施特性的任务

开启限流。这个任务展示如何使用Istio来动态限制到服务的流量

开启限流

本任务将演示如何使用Istio实现动态流控。

开始之前

- 根据安装指南中的快速入门指南，在Kubernetes集群中安装Istio。
- 部署BookInfo示例应用。
- 初始化应用版本，把直接来自测试用户“Jason”的 `reviews` 服务请求路由到v2版本，同时把来自其他用户的请求路由到v3版本。

```
  istioctl create -f samples/bookinfo/kube/route-rule-reviews-test-v2.yaml
  istioctl create -f samples/bookinfo/kube/route-rule-reviews-v3.yaml
```

> 注意：如果与之前的任务中设置了重复的规则，请使用 `istioctl replace` 替代 `istioctl create` 。

限流

Istio支持用户为一个服务设置流量速率限制。

可以把 `ratings` 当作一个外部付费服务，像Rotten Tomatoes®之类的，限额 `1qps` 免费。使用Istio我们能保证不会超过 `1qps` 。

1. 在浏览器中打开BookInfo的 `productpage` （http://$GATEWAY_URL/productpage）。

   如果使用用户"jason"登录，应当会看见每个审核的空白评级星号，表明 `ratings` 服务是通过 `reviews` 服务的"v2"版本调用的。

   如果使用其他用户登录（或退出），应当会看见每个审核的红色评价星号，表明 `ratings` 服务是通过 `reviews` 服务的"v3"版本调用的。

2. 配置一个速率限制的 `memquota` 适配器。

   将下列YAML片段保存到文件 `ratelimit-handler.yaml` 中：

```
apiVersion: config.istio.io/v1alpha2
kind: memquota
metadata:
  name: handler
  namespace: istio-system
spec:
  quotas:
  - name: requestcount.quota.istio-system
    # default rate limit is 5000qps
    maxAmount: 5000
    validDuration: 1s
    # The first matching override is applied.
    # A requestcount instance is checked against override di
mensions.
    overrides:
    # The following override applies to traffic from 'rewiew
s' version v2,
    # destined for the ratings service. The destinationVersi
on dimension is ignored.
    - dimensions:
        destination: ratings
        source: reviews
        sourceVersion: v2
      maxAmount: 1
      validDuration: 1s
```

   然后执行命令：

```
istioctl create -f ratelimit-handler.yaml
```

   这个配置指定了一个默认5000 qps的速率限制。通过review-v2调用评级服务的流量，受到 1qps的速率限制。在本示例中，用户"jason"被路由到reviews-v2，因此受到1qps的速率限制。

3. 配置速率限制实例和规则

创建一个限额实例，命名为 `requestcount` ，映射进入属性到限额规则，并创建一个规则用于memquota处理器。

```
apiVersion: config.istio.io/v1alpha2
kind: quota
metadata:
  name: requestcount
  namespace: istio-system
spec:
  dimensions:
    source: source.labels["app"] | source.service | "unknown"

    sourceVersion: source.labels["version"] | "unknown"
    destination: destination.labels["app"] | destination.ser
vice | "unknown"
    destinationVersion: destination.labels["version"] | "unk
nown"
---
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  name: quota
  namespace: istio-system
spec:
  actions:
  - handler: handler.memquota
    instances:
    - requestcount.quota
```

保存此配置到 `ratelimit-rule.yaml` 并执行下述命令：

```
istioctl create -f ratelimit-rule.yaml
```

4. 使用下列命令为 `productpage` 页生成负载：

```
while true; do curl -s -o /dev/null http://$GATEWAY_URL/prod
uctpage; done
```

5. 在浏览器中刷新 `productpage` 页。

   如果在负载生成器运行时（即生成大于1 req/s）使用用户"jason"登录，浏览器所生成的流量将被限流在1 qps。 review-v2服务不能访问评级服务，将不再看到星号。对所有其他用户来说默认生效的是5000qps的限流额度，将会持续看到红色星号。

带条件的速率限制

前述例子中应用了一个速率限制策略到 `ratings` 服务上，但没有考虑非规格属性。通过在限额规则中使用匹配条件，就可以有条件地应用速率限制到任意属性上。

举个例子，考虑如下配置：

```
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
 name: quota
 namespace: istio-system
spec:
 match: source.namespace != destination.namespace
 actions:
 - handler: handler.memquota
   instances:
   - requestcount.quota
```

这个配置将应用限额规则到那些源命名空间和目的命名空间不同的请求上。

理解限流

之前的例子中我们看到了Mixer如何应用速率限制到满足特定条件的请求上。

每个指定的限额实例，比如 `requestcount` ，代表着一个计数器集合。该集合由所有限额规则的笛卡尔积来定义。如果最后一个 `expiration` 周期中的请求数超过了 `maxAmount` ，Mixer就返回一个 `RESOURCE_EXHAUSTED` 消息给代理，代理就依次返回 `HTTP 429` 状态给调用方。

`memquota` 适配器使用一个亚秒级分辨率的滑动窗口来强行限流。

适配器配置中的 `maxAmount` 为一个限额实例的所有关联计数器，设置了默认的限制。如果没有任何限额重载匹配请求，则这个默认限制会生效。Memquota选择匹配请求的第一个重载。一个重载无需指定所有限额规则。在ratelimit-handler.yaml示例中， `1qps` 重载被只匹配四分之三的限额规则所选中。

如果想在指定命名空间强制执行以上策略，而在不是整个Istio网格，那么可以使用指定的命名空间来替代istio系统的所有事件。

取消流控

- 取消限流配置：

```
  istioctl delete -f ratelimit-handler.yaml
  istioctl delete -f ratelimit-rule.yaml
```

- 取消应用路由规则：

```
  istioctl delete -f samples/bookinfo/kube/route-rule-reviews-test-v2.yaml
  istioctl delete -f samples/bookinfo/kube/route-rule-reviews-v3.yaml
```

- 如果并不计划了解更多后续任务，参考BookInfo cleanup的说明来关闭应用。

进阶阅读

- 了解更多关于Mixer和Mixer Config的内容。
- 探索完整的Attribute Vocabulary。
- 阅读Writing Config的参考指南。

描述演示如何从服务网格收集遥测信息的任务。

- 分布式跟踪。如何配置代理，以便向Zipkin或Jaeger发送跟踪请求
- 收集metrics和日志。这个任务展示如何配置Istio来收集metrics和日志。
- 收集TCP服务的Metrics。这个任务展示如何为TCP服务收集metrics和日志。
- 从Prometheus中查询Metrics。这个任务展示如何使用Prometheus查询metrics。
- 使用Grafana可视化Metrics。这个任务展示如何安装并使用Istio的Dashboard来监控网格流量
- 生成服务图。这个任务展示如何把Istio网格中的服务生成服务图
- 使用Fluentd记录日志。这个任务展示如何配置Istio来将日志记录到Fluentd后台服务

分布式调用链跟踪

本章介绍如何使用Zipkin或Jaeger收集启用了Istio的应用程序的调用链信息。

完成本章后，你可以理解有关应用程序的所有假设以及如何使其参与跟踪，无论您使用何种语言/框架/平台构建应用程序。

BookInfo示例用来作为此任务的示例应用程序。

环境准备

- 参照安装指南的说明安装Istio。

  如果您在安装过程中未启动Zipkin或Jaeger插件，则可以运行以下命令启动：

  启动Zipkin：

  ```
  kubectl apply -f install/kubernetes/addons/zipkin.yaml
  ```

  启动Jaeger：

  ```
  kubectl apply -n istio-system -f https://raw.githubusercontent.com/jaegertracing/jaeger-kubernetes/master/all-in-one/jaeger-all-in-one-template.yml
  ```

- 部署BookInfo示例中的应用程序。

访问仪表盘

Zipkin

通过端口转发设置访问Zipkin dashboard URL：

```
kubectl port-forward -n istio-system $(kubectl get pod -n istio-system -l app=zipkin -o jsonpath='{.items[0].metadata.name}') 9411:9411 &
```

然后使用浏览器访问http://localhost:9411。

Jaeger

通过端口转发设置访问Jaeger dashboard URL：

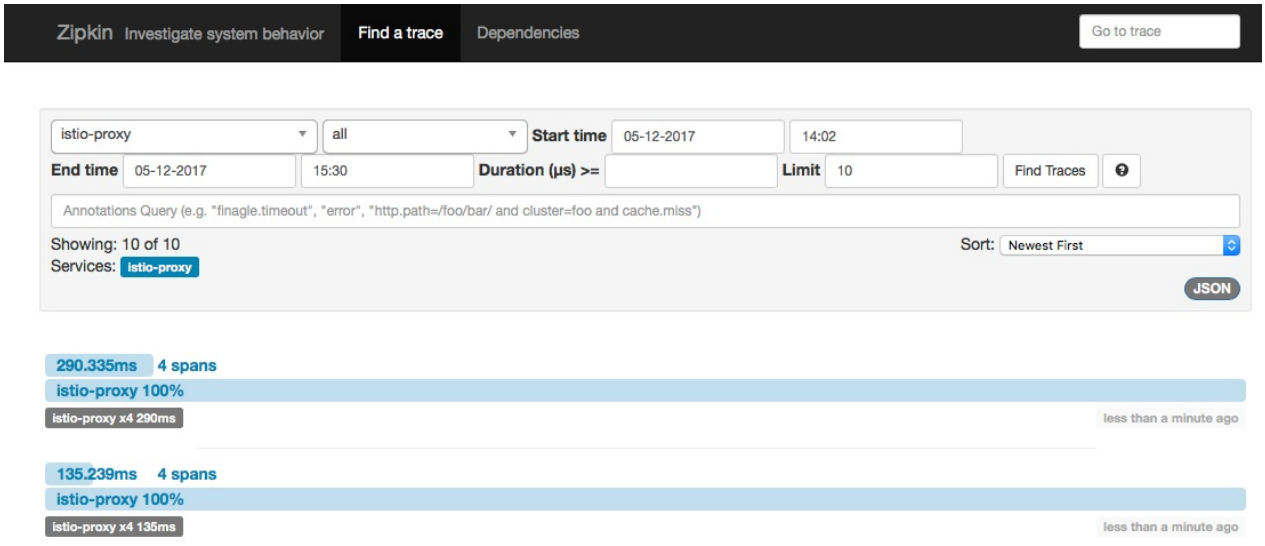
```
kubectl port-forward -n istio-system $(kubectl get pod -n istio-system -l app=jaeger -o jsonpath='{.items[0].metadata.name}') 16686:16686 &
```

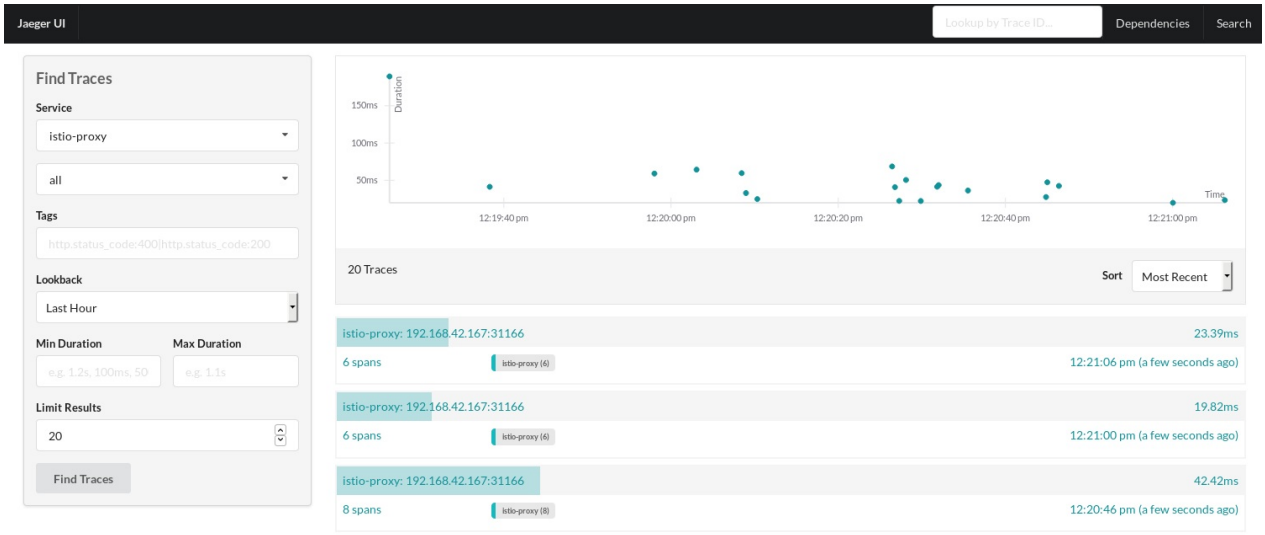
然后使用浏览器访问http://localhost:16686。

使用BookInfo示例生成调用链跟踪

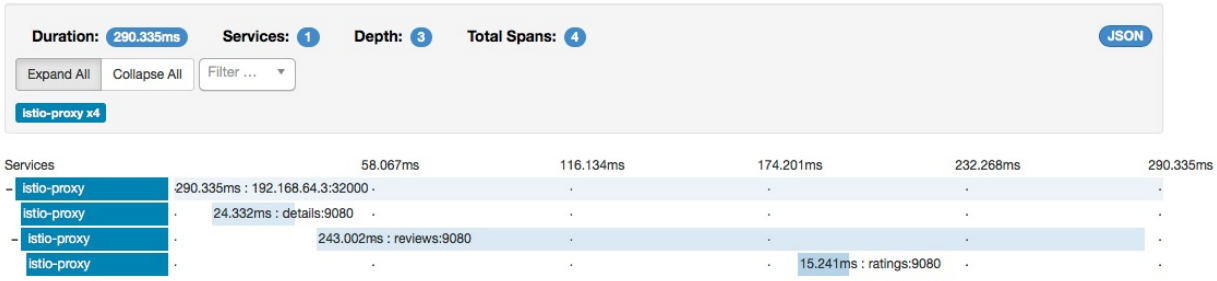
BookInfo的应用程序启动和运行后，通过访问 `http://$GATEWAY_URL/productpage` 一次或多次生成调用链信息。

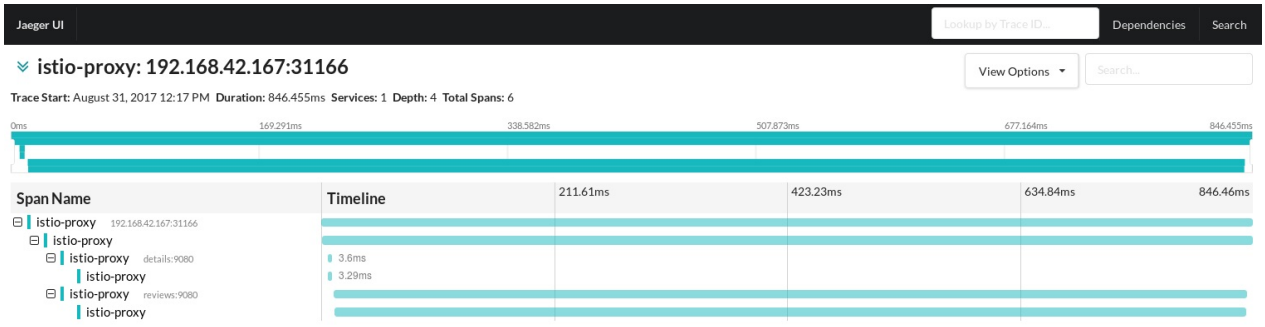
如果你查看仪表板，会看到类似以下的内容：





如果你点击调用链堆栈中（最近的）最近的一条，您应该看到刷新 `/productpage` 后最新的详细信息。页面看起来像这样：





像您看到的，调用链由spans组成，其中每个span对应于使用 `/productpage` 去调用BookInfo服务。 因为调用链堆栈是由Istio Sidecar（Envoy代理）包装实际的服务完成的，所以每个服务具有相同的标签 `istio-proxy` 。右侧的目的地标签每一行标识该服务的调用耗时。

第一行表示 `productpage` 服务被外部调用。 `192.168.64.3:32000` 标签是外部请求的主机信息（即$GATEWAY_URL）。 从调用堆栈中可以看到，请求总共耗时大约290毫秒完成。 在执行过程中， `productpage` 调用 `details` 服务，耗时约24ms，然后调用 `review` 服务。 `review` 服务耗时约243毫秒，其中包括一个15毫秒的 `ratings` 服务。

理解下发生了什么

尽管Istio代理能够自动发送spans，但他们需要一些标识来将整个调用链关系联系起来。应用程序需要传入合适的HTTP header信息，便于代理发送span信息到Zipkin或Jaeger时，span可以准确地把每次调用关联起来。

为此，应用程序需要从传入的请求中收集如下的header信息并将其传入到每个传出请求：

- `x-request-id`
- `x-b3-traceid`
- `x-b3-spanid`
- `x-b3-parentspanid`
- `x-b3-sampled`
- `x-b3-flags`
- `x-ot-span-context`

如果您细看示例的服务，可以看到productpage应用（Python应用）从HTTP请求中提取所需的header信息：

```
def getForwardHeaders(request):
    headers = {}

    user_cookie = request.cookies.get("user")
    if user_cookie:
        headers['Cookie'] = 'user=' + user_cookie

    incoming_headers = [ 'x-request-id',
                         'x-b3-traceid',
                         'x-b3-spanid',
                         'x-b3-parentspanid',
                         'x-b3-sampled',
                         'x-b3-flags',
                         'x-ot-span-context'
    ]

    for ihdr in incoming_headers:
        val = request.headers.get(ihdr)
        if val is not None:
            headers[ihdr] = val
            #print "incoming: "+ihdr+":"+val

    return headers
```

示例中reviews应用(Java应用)也做了类似的事情：

```
    @GET
    @Path("/reviews")
    public Response bookReviews(@CookieParam("user") Cookie user
,
                                @HeaderParam("x-request-id") Str
ing xreq,
                                @HeaderParam("x-b3-traceid") Str
ing xtraceid,
                                @HeaderParam("x-b3-spanid") Stri
ng xspanid,
                                @HeaderParam("x-b3-parentspanid")
 String xparentspanid,
                                @HeaderParam("x-b3-sampled") Str
ing xsampled,
                                @HeaderParam("x-b3-flags") Strin
g xflags,
                                @HeaderParam("x-ot-span-context")
 String xotspan) {
      String r1 = "";
      String r2 = "";

      if(ratings_enabled){
        JsonObject ratings = getRatings(user, xreq, xtraceid, xs
panid, xparentspanid, xsampled, xflags, xotspan);
```

在应用程序中调用其他服务时，请确保包含这些header信息。

清除

- 删除调用链跟踪的配置：

  如果使用Zipkin，请运行以下命令进行清理

  ```
  kubectl delete -f install/kubernetes/addons/zipkin.yaml
  ```

  如果使用Jaeger，请运行以下命令进行清理：

```
  kubectl delete -f https://raw.githubusercontent.com/jaeger
tracing/jaeger-kubernetes/master/all-in-one/jaeger-all-in-on
e-template.yml
```

- 如果您不打算继续后面的章节，请参阅BookInfo cleanup说明关闭应用程序。

进一步阅读

- 了解更多有关Metrics和日志。

收集Metrics和日志

本章展示如何配置Istio来自动收集mesh中服务的遥测数据。

在本章末尾，将为mesh中的服务调用启用新的metric和新的日志流。

BookInfo应用将作为介绍本章内容的示例应用。

开始之前

- 在集群中安装Istio并部署一个应用程序。

  本章假设Mixer使用默认配置（`--configDefaultNamespace=istio-system`）。如果使用不同的值，则更新这个任务中的配置和命令来匹配这个值。

- 安装Prometheus插件。

  Prometheus用于验证任务是否成功。

  ```
  kubectl apply -f install/kubernetes/addons/prometheus.yaml
  ```

  详细信息请看Prometheus。

收集新的遥测数据

1. 创建YAML文件来保存新metric和日志流的配置，Istio将自动生成和收集。

   把以下内容保存成文件`new_telemetry.yaml`：

   ```
   # Configuration for metric instances
   apiVersion: "config.istio.io/v1alpha2"
   kind: metric
   metadata:
     name: doublerequestcount
     namespace: istio-system
   spec:
   ```

```
    value: "2" # count each request twice
    dimensions:
      source: source.service | "unknown"
      destination: destination.service | "unknown"
      message: '"twice the fun!"'
    monitored_resource_type: '"UNSPECIFIED"'
---
# Configuration for a Prometheus handler
apiVersion: "config.istio.io/v1alpha2"
kind: prometheus
metadata:
  name: doublehandler
  namespace: istio-system
spec:
  metrics:
  - name: double_request_count # Prometheus metric name
    instance_name: doublerequestcount.metric.istio-system #
Mixer instance name (fully-qualified)
    kind: COUNTER
    label_names:
    - source
    - destination
    - message
---
# Rule to send metric instances to a Prometheus handler
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: doubleprom
  namespace: istio-system
spec:
  actions:
  - handler: doublehandler.prometheus
    instances:
    - doublerequestcount.metric
---
# Configuration for logentry instances
apiVersion: "config.istio.io/v1alpha2"
kind: logentry
metadata:
```

```
    name: newlog
    namespace: istio-system
  spec:
    severity: '"warning"'
    timestamp: request.time
    variables:
      source: source.labels["app"] | source.service | "unknow
n"
      user: source.user | "unknown"
      destination: destination.labels["app"] | destination.se
rvice | "unknown"
      responseCode: response.code | 0
      responseSize: response.size | 0
      latency: response.duration | "0ms"
    monitored_resource_type: '"UNSPECIFIED"'
  ---
  # Configuration for a stdio handler
  apiVersion: "config.istio.io/v1alpha2"
  kind: stdio
  metadata:
    name: newhandler
    namespace: istio-system
  spec:
   severity_levels:
     warning: 1 # Params.Level.WARNING
   outputAsJson: true
  ---
  # Rule to send logentry instances to a stdio handler
  apiVersion: "config.istio.io/v1alpha2"
  kind: rule
  metadata:
    name: newlogstdio
    namespace: istio-system
  spec:
    match: "true" # match for all requests
    actions:
     - handler: newhandler.stdio
       instances:
       - newlog.logentry
  ---
```

```
---
# Rule to send metric instances to a Prometheus handler
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: doubleprom
  namespace: istio-system
spec:
  actions:
  - handler: doublehandler.prometheus
    instances:
    - doublerequestcount.metric
---
# Configuration for logentry instances
apiVersion: "config.istio.io/v1alpha2"
kind: logentry
metadata:
  name: newlog
  namespace: istio-system
spec:
  severity: '"warning"'
  timestamp: request.time
  variables:
    source: source.labels["app"] | source.service | "unknown"

    user: source.user | "unknown"
    destination: destination.labels["app"] | destination.ser
vice | "unknown"
    responseCode: response.code | 0
    responseSize: response.size | 0
    latency: response.duration | "0ms"
  monitored_resource_type: '"UNSPECIFIED"'
---
# Configuration for a stdio handler
apiVersion: "config.istio.io/v1alpha2"
kind: stdio
metadata:
  name: newhandler
  namespace: istio-system
spec:
```

```
 severity_levels:
   warning: 1 # Params.Level.WARNING
 outputAsJson: true
---
# Rule to send logentry instances to a stdio handler
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: newlogstdio
  namespace: istio-system
spec:
  match: "true" # match for all requests
  actions:
   - handler: newhandler.stdio
     instances:
     - newlog.logentry
---
```

2. 推送新配置。

```
istioctl create -f new_telemetry.yaml
```

期待输出类似内容：

```
Created config metric/istio-system/doublerequestcount at rev
ision 1973035
Created config prometheus/istio-system/doublehandler at revi
sion 1973036
Created config rule/istio-system/doubleprom at revision 1973
037
Created config logentry/istio-system/newlog at revision 1973
038
Created config stdio/istio-system/newhandler at revision 197
3039
Created config rule/istio-system/newlogstdio at revision 197
3041
```

3. 发送流量到示例的应用程序。

   以Bookinfo的例子为例，在浏览器中访问 `http://$GATEWAY_URL/productpage` 或者使用以下命令:

```
curl http://$GATEWAY_URL/productpage
```

4. 验证已产生新的metric并且能被收集。

   在Kubernetes环境中，通过执行以下命令为Prometheus设置端口转发：

```
kubectl -n istio-system port-forward $(kubectl -n istio-syst
em get pod -l app=prometheus -o jsonpath='{.items[0].metadat
a.name}') 9090:9090 &
```

   通过Prometheus UI查看新metric信息。

   打开Prometheus UI，查询 `double_request_count` 。**Console** tab页中有类似如下信息：

```
 istio_double_request_count{destination="details.default.svc
.cluster.local",instance="istio-mixer.istio-system:42422",jo
b="istio-mesh",message="twice the fun!",source="productpage.
default.svc.cluster.local"}    2
 istio_double_request_count{destination="ingress.istio-syste
m.svc.cluster.local",instance="istio-mixer.istio-system:4242
2",job="istio-mesh",message="twice the fun!",source="unknown"
}    2
 istio_double_request_count{destination="productpage.default
.svc.cluster.local",instance="istio-mixer.istio-system:42422"
,job="istio-mesh",message="twice the fun!",source="ingress.i
stio-system.svc.cluster.local"}    2
 istio_double_request_count{destination="reviews.default.svc
.cluster.local",instance="istio-mixer.istio-system:42422",jo
b="istio-mesh",message="twice the fun!",source="productpage.
default.svc.cluster.local"}    2
```

   有关查询Prometheus获取更多metric信息，请参阅查询Istio Metrics。

5. 验证日志流是否已创建并可使用。

   在Kubernetes环境中，使用以下方式搜索Mixer pod的日志：

```
kubectl -n istio-system logs $(kubectl -n istio-system get p
ods -l istio=mixer -o jsonpath='{.items[0].metadata.name}')
mixer | grep \"instance\":\"newlog.logentry.istio-system\"
```

   输出类似以下内容：

```
 {"level":"warn","ts":"2017-09-21T04:33:31.249Z","instance":
"newlog.logentry.istio-system","destination":"details","late
ncy":"6.848ms","responseCode":200,"responseSize":178,"source"
:"productpage","user":"unknown"}
 {"level":"warn","ts":"2017-09-21T04:33:31.291Z","instance":
"newlog.logentry.istio-system","destination":"ratings","late
ncy":"6.753ms","responseCode":200,"responseSize":48,"source":
"reviews","user":"unknown"}
 {"level":"warn","ts":"2017-09-21T04:33:31.263Z","instance":
"newlog.logentry.istio-system","destination":"reviews","late
ncy":"39.848ms","responseCode":200,"responseSize":379,"sourc
e":"productpage","user":"unknown"}
 {"level":"warn","ts":"2017-09-21T04:33:31.239Z","instance":
"newlog.logentry.istio-system","destination":"productpage","
latency":"67.675ms","responseCode":200,"responseSize":5599,"
source":"ingress.istio-system.svc.cluster.local","user":"unk
nown"}
 {"level":"warn","ts":"2017-09-21T04:33:31.233Z","instance":
"newlog.logentry.istio-system","destination":"ingress.istio-
system.svc.cluster.local","latency":"74.47ms","responseCode":
200,"responseSize":5599,"source":"unknown","user":"unknown"}
```

理解监控的配置

在本章中，您添加了Istio配置，委托Mixer自动生成并报告mesh内所有流量的新metric和新日志流。

新增配置控制Mixer功能的三块：

1. *instance*生成（在本例中为metrcis值和日志），使用Istio属性。
2. *handler*（配置好的Mixer适配器）的创建，能够处理已生成的实例。
3. 根据一套规则分发*instance*到*handler*。

理解metrics配置

metrics配置指定Mixer把metrics指标值发送给Prometheus。它由三部分（或块）的配置组成：*instance*配置，*handler*配置和*rule*配置。

`kind: metric` 这部分配置定义了一个名为 `doublerequestcount` 的新metric模板去生成metric值（或实例）。这个实例的配置告诉Mixer如何根据Envoy返回的属性（同样由Mixer自身生成）为任何给定的请求生成metric。

`doublerequestcount.metric` 中的配置指定Mixer为每个instance的值是2。由于Istio为每个请求生成一个instance，这意味着这个metric的值等于收到请求总数的两倍。

每个 `doublerequestcount.metric` instance指定一个 `dimensions` 。Dimensions提供了根据不同的需求和查询方式来分割，聚合和分析度量数据的方法。例如，在对应用程序行为进行故障排除时，可能只需要关注对某个服务的请求。

根据属性值和实际的值配置Dimensions来约定Mixer的行为。例如，对于 `source dimensions` ，新的配置请求从 `source.service` 属性取值。如果该属性值未设置，则该规则中Mixer会使用默认值 `unknown` 。对于 `message` ，设置值为 `"twice the fun!"` 将用于所有实例。

`kind: prometheus` ：定义了一个名为 `doublehandler` 的*handler*。 `spec` 配置prometheus适配器代码如何把接收的metric值转换为Prometheus后端可处理的格式。此配置指定了一个名为 `double_request_count` 的新Prometheus指标，有三个标签（与 `doublerequestcount.metric` 实例配置的相匹配）。

对于 `kind：prometheus` 这个handler，Mixer实例通过 `instance_name` 参数与Prometheus的metrics相匹配。 `instance_name` 值必须和Mixer实例的名称完全一致（例如： `doublerequestcount.metric.istio-system` ）。

`kind: rule` 配置定义了一个名为 `doubleprom` 的新规则。在该规则下Mixer将所有 `doublerequestcount.metric` 实例发送到 `doublehandler.prometheus` 处理。由于规则中没有 `match` 子句，并且由于该规则位于已配置默认命名空间（ `istio-system` ），因此mesh中的所有请求都会执行该规则。

理解日志的配置

日志配置指定Mixer发送日志到stdout(标准输出)。它使用三部分（或块）配置：*instance*配置，*handler*配置和*rule*配置。

`kind: logentry` 配置定义了一个名为 `newlog` 的日志对象（或*instance*）的模型。这个配置告诉Mixer如何根据Envoy返回的属性生成日志对象。

`severity` 参数用于指定任何要生成 `logentry` 的日志级别。在这个例子中，使用了 `"warning"` 。该值将通过 `logentry` 项填充生成的日志中。

`timestamp` 参数为所有日志提供时间信息。在本例中，用Envoy提供 `request.time` 属性来设置时间。

`variables` 参数允许操作员配置每个 `logentry` 应包含哪些值。用表达式控制从Istio属性和设定值的映射关系来构成一个 `logentry` 。在此示例中，每个 `logentry` 实例都有一个名为 `latency` 的字段，对应了属性 `response.duration` 的值。如果 `response.duration` 没有值，则 `latency` 默认设置为0ms。

`kind: stdio` 定义了一个名为 `newhandler` 的*handler*。 `spec` 配置 `stdio` 适配器代码如何处理收到 `logentry` 实例。 `severity_levels` 参数控制 `severity` 字段和 `logentry` 值如何映射到已支持的日志级别。这里， `"warning"` 的值被映射到 `WARNING` 级别的日志。 `outputAsJson` 参数指定适配器生成JSON格式的日志。

`kind: rule` 定义了一个名为 `newlogstdio` 的新规则。该规则指定Mixer将 `newlog.logentry` 的所有实例发送到 `newhandler.stdio` 的handler。由于 `match` 参数设置为true，所以对mesh中的所有请求都执行该规则。

为所有请求都执行该规则是不需要明确配置 `match: true` 。 `spec` 省略 `match` 参数项配置相当于设置 `match：true` 。这里是为了说明如何使用 `match` 表达式来配置控制规则。

清除

- 删除监控配置：

```
istioctl delete -f new_telemetry.yaml
```

- 如果您不打算继续后面的章节，请参阅BookInfo cleanup的说明关闭应用程序。

进一步阅读

- 学习更多关于Mixer和Mixer Config。
- 查看完整的Attribute Vocabulary。
- 阅读参考指南编写配置。
- 请参阅深入遥测指南。

收集TCP服务的Metrics

本章介绍如何配置istio去自动收集mesh中的TCP服务指标。本章结尾部分，一个新的监控指标可以用mesh去调用TCP服务。

BookInfo应用作为介绍本章的示例应用。

开始之前

- 在集群中安装istio并部署一个应用。
- 本章假定BookInfo中的示例应用被部署在 `default` 命名空间中。如果部署在不同的命名空间，你需要修改例子中的配置和命令。
- 安装Prometheus插件，Prometheus用来验证任务是否成功执行。

  ```
  kubectl apply -f install/kubernetes/addons/prometheus.yaml
  ```

  详细内容查看Prometheus。

收集新的遥测数据

1. 创建YAML文件来保存新的metric配置，Istio将自动生成和收集。

   把以下内容保持为 `tcp_telemetry.yaml` 文件：

   ```
   # Configuration for a metric measuring bytes sent from a se
  rver
   # to a client
   apiVersion: "config.istio.io/v1alpha2"
   kind: metric
   metadata:
     name: mongosentbytes
     namespace: default
   spec:
     value: connection.sent.bytes | 0 # uses a TCP-specific at
   ```

```
tribute
   dimensions:
     source_service: source.service | "unknown"
     source_version: source.labels["version"] | "unknown"
     destination_version: destination.labels["version"] | "u
nknown"
   monitoredResourceType: '"UNSPECIFIED"'
 ---
 # Configuration for a metric measuring bytes sent from a cl
ient
 # to a server
 apiVersion: "config.istio.io/v1alpha2"
 kind: metric
 metadata:
   name: mongoreceivedbytes
   namespace: default
 spec:
   value: connection.received.bytes | 0 # uses a TCP-specifi
c attribute
   dimensions:
     source_service: source.service | "unknown"
     source_version: source.labels["version"] | "unknown"
     destination_version: destination.labels["version"] | "u
nknown"
   monitoredResourceType: '"UNSPECIFIED"'
 ---
 # Configuration for a Prometheus handler
 apiVersion: "config.istio.io/v1alpha2"
 kind: prometheus
 metadata:
   name: mongohandler
   namespace: default
 spec:
   metrics:
   - name: mongo_sent_bytes # Prometheus metric name
     instance_name: mongosentbytes.metric.default # Mixer in
stance name (fully-qualified)
     kind: COUNTER
     label_names:
     - source_service
```

```
      - source_version
      - destination_version
    - name: mongo_received_bytes # Prometheus metric name
      instance_name: mongoreceivedbytes.metric.default # Mixe
r instance name (fully-qualified)
      kind: COUNTER
      label_names:
      - source_service
      - source_version
      - destination_version
 ---
 # Rule to send metric instances to a Prometheus handler
 apiVersion: "config.istio.io/v1alpha2"
 kind: rule
 metadata:
   name: mongoprom
   namespace: default
 spec:
   match: context.protocol == "tcp"
          && destination.service == "mongodb.default.svc.clu
ster.local"
   actions:
   - handler: mongohandler.prometheus
     instances:
     - mongoreceivedbytes.metric
     - mongosentbytes.metric
```

2. 推送新的配置。

```
istioctl create -f tcp_telemetry.yaml
```

输出类似如下内容：

```
Created config metric/default/mongosentbytes at revision 385
2843
Created config metric/default/mongoreceivedbytes at revision
 3852844
Created config prometheus/default/mongohandler at revision 3
852845
Created config rule/default/mongoprom at revision 3852846
```

3. 设置并使用MongoDB

   i. 安装 `v2` 版本的 `ratings` 服务。

   如果您启用了自动注入sidecar功能的集群，只需使用 `kubectl` 部署服务即可：

   ```
    kubectl apply -f samples/bookinfo/kube/bookinfo-ratings-
   v2.yaml
   ```

   如果您正在使用手动注入sidecar，请改用以下命令：

   ```
    kubectl apply -f <(istioctl kube-inject -f samples/booki
   nfo/kube/bookinfo-ratings-v2.yaml)
   ```

   输出如下内容：

   ```
    deployment "ratings-v2" configured
   ```

   ii. 安装 `mongodb` 服务：

   如果您启用了自动注入sidecar功能的集群，只需使用 `kubectl` 部署服务即可：

   ```
    kubectl apply -f samples/bookinfo/kube/bookinfo-db.yaml
   ```

   如果您正在使用手动注入sidecar，请改用以下命令：

```
 kubectl apply -f <(istioctl kube-inject -f samples/booki
nfo/kube/bookinfo-db.yaml)
```

输出如下内容：

```
service "mongodb" configured
deployment "mongodb-v1" configured
```

iii. 添加路由规则将流量发送到 `ratings` 服务的 `v2` 版本：

```
istioctl create -f samples/bookinfo/kube/route-rule-ratings-
db.yaml
```

输出如下内容：

```
Created config route-rule//ratings-test-v2 at revision 72164
03
Created config route-rule//reviews-test-ratings-v2 at revisi
on 7216404
```

4. 把流量发送到示例的应用。

参照BookInfo中的例子，使用浏览器访问 `http://$GATEWAY_URL/productpage` 或者使用如下命令：

```
curl http://$GATEWAY_URL/productpage
```

5. 验证新的监控信息是否产生并被收集

在Kubernetes环境中，使用如下命令为Prometheus设置端口转发：

```
kubectl -n istio-system port-forward $(kubectl -n istio-syst
em get pod -l app=prometheus -o jsonpath='{.items[0].metadat
a.name}') 9090:9090 &
```

通过Prometheus UI查看新的监控指标项采集的数据。

使用提供的链接打开Prometheus UI并查询 `mongo_received_bytes` metric收集的数据。在**Console** tab页展示类似以下信息：

```
 istio_mongo_received_bytes{destination_version="v1",instanc
e="istio-mixer.istio-system:42422",job="istio-mesh",source_s
ervice="ratings.default.svc.cluster.local",source_version="v
2"}    2317
```

注意：Istio还收集MongoDB特定协议的统计信息。例如，从 `ratings` 服务发送的所有OP_QUERY信息的值，用以下metric收集: `envoy_mongo_mongo_collection_ratings_query_total_counter` （单击此处执行查询）。

理解TCP遥测收集

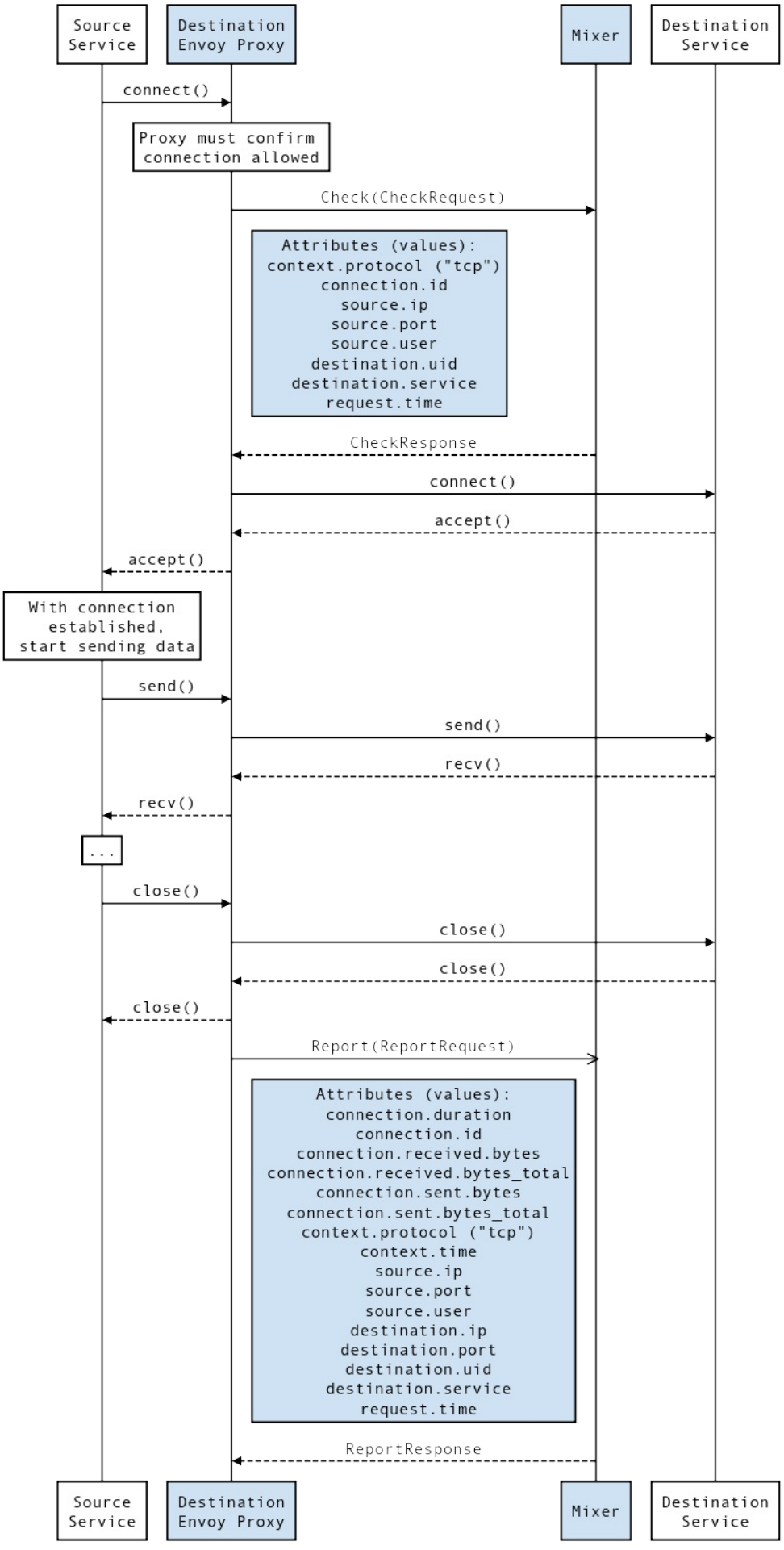
在此任务中，您添加了Istio的配置，使Mixer能自动生成mesh内TCP服务的所有流量的metric并收集。

与收集指标和日志类似，新配置由*instances*，*handler*和*rule*组成。请参阅该任务了解获取metric数据收集的完整说明。

TCP服务的监控数据收集配置仅在*instances*中有有限几个属性有所不同。

TCP属性

几个特定的TCP属性可以在Istio中启用TCP策略和控制。这些属性由服务器端的Envoy代理生成，并在连接建立和关闭时转发给Mixer。另外，上下文属性提供区分 `http` 和 `tcp` 协议的能力。


Source Service
Destination Envoy Proxy
Mixer
Destination Service
connect()
Proxy must confirm connection allowed
Check(CheckRequest)
Attributes (values):
context.protocol ("tcp")
connection.id
source.ip
source.port
source.user
destination.uid
destination.service
request.time
CheckResponse
connect()
accept()
accept()
With connection established, start sending data
send()
send()
recv()
recv()
...
close()
close()
close()
close()
Report(ReportRequest)
Attributes (values):
connection.duration
connection.id
connection.received.bytes
connection.received.bytes_total
connection.sent.bytes
connection.sent.bytes_total
context.protocol ("tcp")
context.time
source.ip
source.port
source.user
destination.ip
destination.port
destination.uid
destination.service
request.time
ReportResponse
Source Service
Destination Envoy Proxy
Mixer
Destination Service

TCP Attribute Flow

清理

- 删除新的配置：

```
istioctl delete -f tcp_telemetry.yaml
```

- 删除端口映射：

```
killall kubectl
```

- 如果您不打算继续后面的章节，请参阅BookInfo cleanup的说明关闭应用程序。

进一步阅读

- 学习更多关于Mixer和Mixer Config。
- 查看全部Attribute Vocabulary。
- 阅读编写配置的参考指南。
- 参考[深入遥测]（../../guides/telemetry.md）指南。
- 学习更多关于查询Istio Metrics。
- 学习更多关于MongoDB-specific statistics generated by Envoy。

从Prometheus中查询Metrics

此任务向您展示如何使用Prometheus查询Istio监控信息。

作为此任务的一部分，您将安装Prometheus Istio插件，并使用Web界面查询监控信息。

本章使用BookInfo示例应用程序作为例子。

开始之前

- 安装Istio，部署应用。

查询Istio metrics

1. 要查询Mixer提供的监控信息，先要安装Prometheus插件。

   在Kubernetes中，执行以下命令：

   ```
   kubectl apply -f install/kubernetes/addons/prometheus.yaml
   ```

2. 验证服务是否已正常运行。

   在Kubernetes中，执行以下命令：

   ```
   kubectl -n istio-system get svc prometheus
   ```

   会看到类似以下的输出：

   ```
   NAME         CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
   prometheus   10.59.241.54   <none>        9090/TCP   2m
   ```

3. 发送流量到mesh。

对应BookInfo示例，在浏览器中访问 `http://$GATEWAY_URL/productpage` 或者用以下命令请求：

```
curl http://$GATEWAY_URL/productpage
```

提示： `$GATEWAY_URL` 的值请参考BookInfo。

4. 打开Prometheus UI。

   在Kubernetes中，执行以下命令：

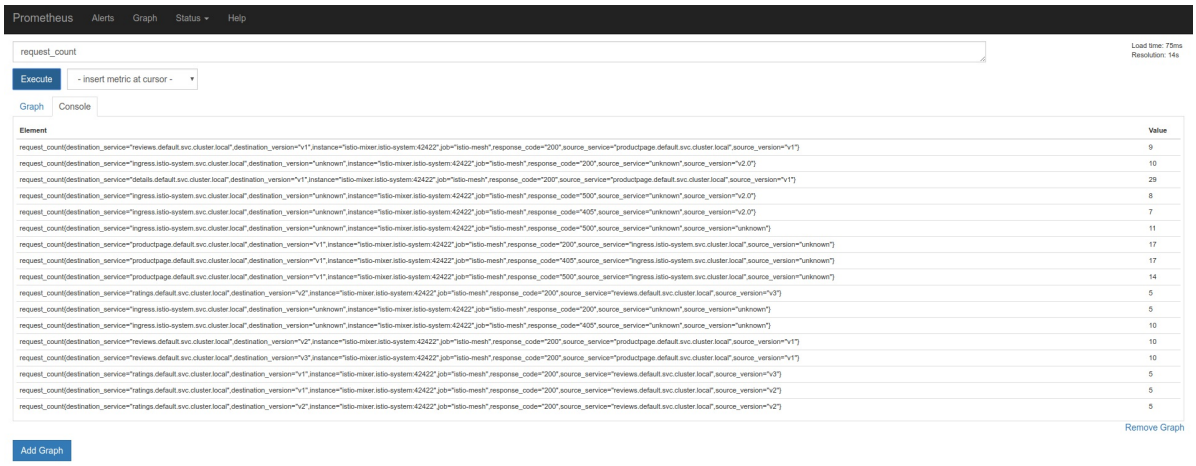
   ```
   kubectl -n istio-system port-forward $(kubectl -n istio-syst
   em get pod -l app=prometheus -o jsonpath='{.items[0].metadat
   a.name}') 9090:9090 &
   ```

   浏览器中访问：http://localhost:9090/graph。

5. Prometheus中执行查询。

   在网页的"Expression"输入框中输入文本： `istio_request_count` ，然后点击**Execute**按钮。

   会有类似下面的结果输出：



   再试试其他查询：

   - 查询调用 `productpage` 服务的总数：

   ```
   istio_request_count{destination_service="productpage.defaul
   t.svc.cluster.local"}
   ```

- 查询请求 `reviews` 的 `v3` 版本服务的总次数：

```
istio_request_count{destination_service="reviews.default.svc
.cluster.local", destination_version="v3"}
```

查询得到的结果就是当前请求 `reviews` 的 `v3` 版本服务的总数。

- 所有 `productpage` 服务在过去5分钟内的请求成功率：

```
rate(istio_request_count{destination_service=~"productpage.*"
, response_code="200"}[5m])
```

关于Prometheus插件

Mixer内置了一个Prometheus适配器，并开放了一个服务用于收集监控信息。Prometheus插件是一个Prometheus服务器，它预置了数据抓取配置，可以从Mixer收集metrics。 它提供了一个持久存储和查询Istio metrics的机制。

配置好的Prometheus插件有三部分：

1. *istio-mesh*(`istio-mixer.istio-system：42422`): 所有Mixer产生的mesh metrics。
2. *Mixer*(`istio-mixer.istio-system：9093`): 所有特定的Mixer metrics。 用于监控Mixer自身。
3. *envoy*(`istio-mixer.istio-system：9102`): 由envoy生成原始统计信息（并从statsd翻译成Prometheus）。

有关查询Prometheus的更多信息，请阅读他们的querying docs。

清理

- 在Kubernetes中，执行如下命令删除Prometheus：

```
kubectl delete -f install/kubernetes/addons/prometheus.yaml
```

- 移除任何可能在运行的 `kubectl port-forward` 进程：

```
killall kubectl
```

- 如果您不打算继续后续的章节，请参阅BookInfo cleanup说明去关闭应用程序。

进一步阅读

- 请参阅深入遥测指南。

使用Grafana可视化Metrics

这个任务展示了如何设置和使用Istio Dashboard对Service Mesh中的流量进行监控。作为这个任务的一部分，需要安装Grafana的Istio插件，然后使用Web界面来查看Service Mesh的流量数据。

开始之前

- 在集群上安装Istio并部署一个应用。
- 安装Prometheus插件。

  ```
  kubectl apply -f install/kubernetes/addons/prometheus.yaml
  ```

  Istio Dashboard必须有Prometheus插件才能工作。

查看Istio Dashboard

1. 要在Dashboard上查看Istio的Metrics，首先要安装Grafana。

   在Kubernetes环境下，执行下面的命令：

   ```
   kubectl apply -f install/kubernetes/addons/grafana.yaml
   ```

2. 验证服务运行在集群中。

   在Kubernetes环境下，执行下面的命令：

   ```
   kubectl -n istio-system get svc grafana
   ```

   输出类似：

   ```
   NAME      CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
   grafana   10.59.247.103   <none>        3000/TCP   2m
   ```

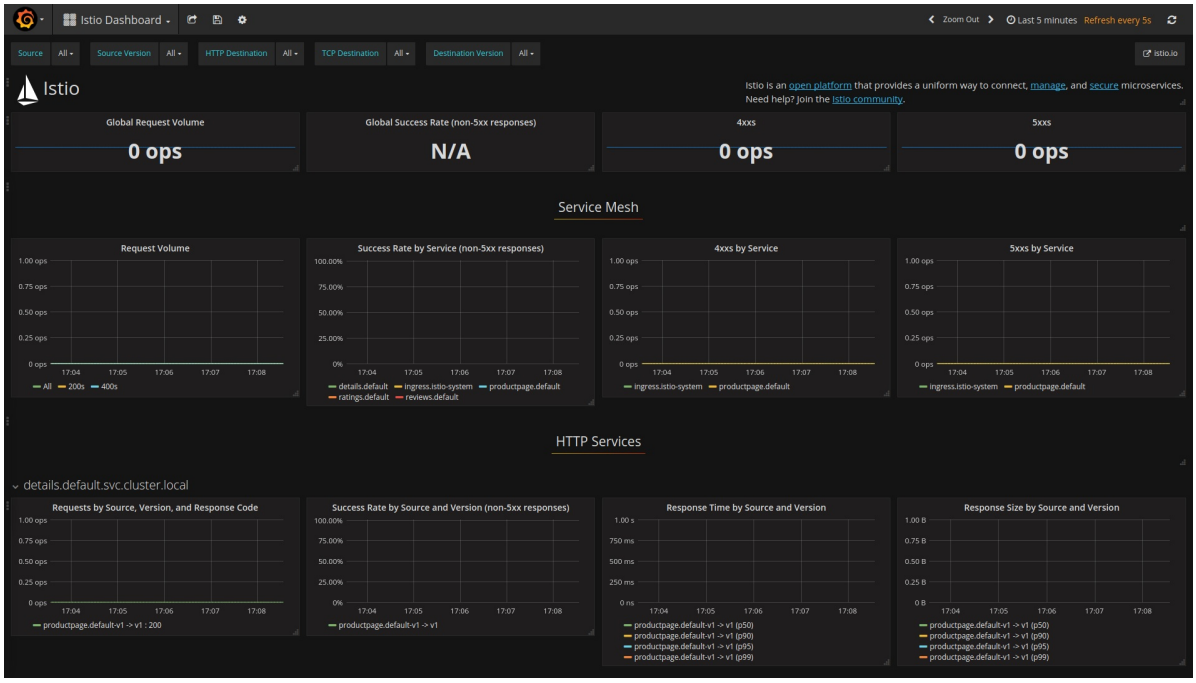
3. 通过Grafana界面，打开Istio的Dashboard

   在Kubernetes环境中，执行下面的命令：

```
 kubectl -n istio-system port-forward $(kubectl -n istio-sys
tem get pod -l app=grafana -o jsonpath='{.items[0].metadata.
name}') 3000:3000 &
```

   然后使用浏览器访问http://localhost:3000/dashboard/db/istio-dashboard：

   Istio Dashboard大概是这样的：



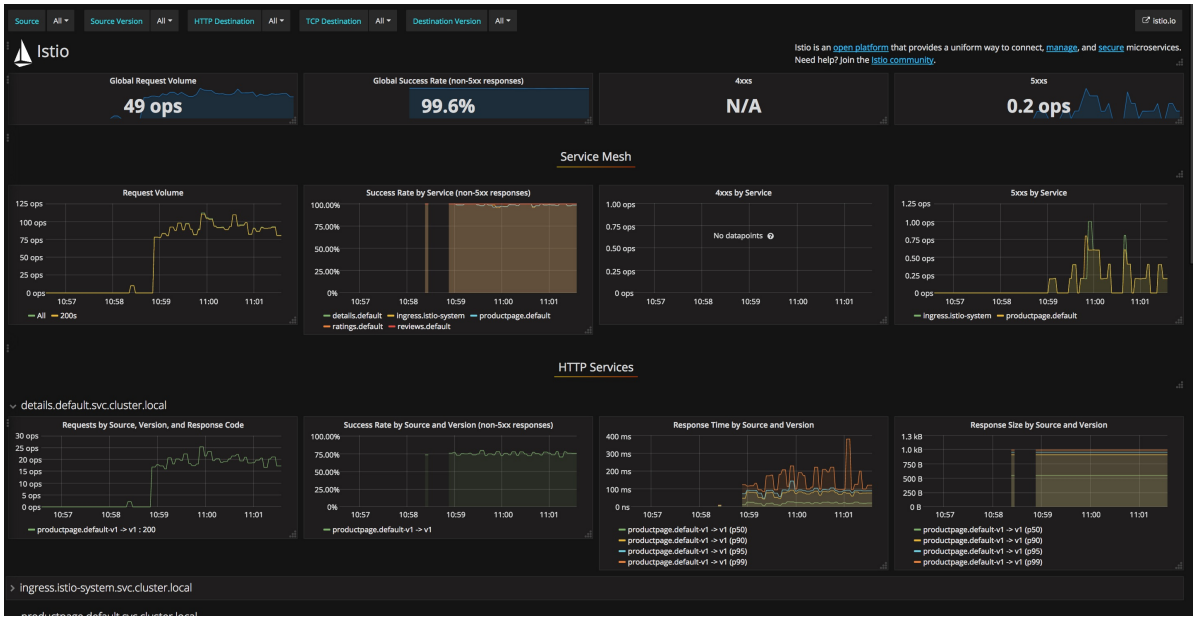
4. 向Mesh发起流量。

   对于BookInfo示例，使用浏览器打开 `http://$GATEWAY_URL/productpage` ，或者发出下列命令：

```
curl http://$GATEWAY_URL/productpage
```

   刷新几次浏览器(或者发送几次命令)，以产生少量流量：

   再次打开Istio Dashboard，它会反应刚生成的流量。看上去类似这样：



注意：`$GATEWAY_URL`的值是在BookInfo指南中设置的。

关于Grafana插件

Grafana插件是一个预先配置好的Grafana实例。我们修改了基础镜像（`grafana/grafana:4.1.2`），在其中加入了Prometheus数据源，以及安装了Istio Dashboard。Istio和Mixer的初始安装中就会初始化一个缺省的（对所有服务生效的）全局Metrics。Istio Dashboard就是依赖这一默认Istio metrics配置和Prometheus插件来完成工作的。

Istio Dashboard由三个部分组成：

1. 全局汇总视图：在Service Mesh中发生的HTTP请求流量的汇总。
2. Mesh汇总视图：比全局汇总视图详细一些，可以根据服务进行过滤和选择。
3. 单一服务视图：提供了单一服务视角下，在Service Mesh内部（HTTP和TCP）请求和响应的情况。

Grafana官方文档介绍了更多关于创建、配置和编辑Dashboard的内容。

清理

- 在Kubernetes环境中，执行下面的命令可以移除Grafana插件。

```
kubectl delete -f install/kubernetes/addons/grafana.yaml
```

- 删除所有可能在运行的的 `kubectl port-forward` 进程：

```
killall kubectl
```

- 如果不想继续后续内容，参考BookInfo清理的介绍来关闭应用。

生成服务图示

这个任务展示如何为Istio mesh中的服务生成图。作为这个任务的一部分，将安装ServiceGraph产检并使用基于web的接口来查看服务网格的服务图。

开始之前

- 在集群上安装Istio并部署应用。
- 安装Prometheus插件。

  这个插件的安装指导作为查询度量任务的一部分提供。

  Service Graph必须有Prometheus插件才能工作。

生成服务图

1. 要查看Service Mesh的图形展示，首先要安装Servicegraph插件。

   在Kubernetes环境中的安装，运行下面的命令：

   ```
   kubectl apply -f install/kubernetes/addons/servicegraph.yaml
   ```

2. 检查集群中服务的运行状态。

   在Kubernetes环境中，运行下面的命令：

   ```
   kubectl -n istio-system get svc servicegraph
   ```

   输出内容大致如下所示：

   ```
   NAME           CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
   servicegraph   10.59.253.165   <none>        8088/TCP   30s
   ```

3. 向服务发送流量。

   如果是BookInfo示例，使用浏览器打开 `http://$GATEWAY_URL/productpage` ，或者使用控制台命令：

   ```
   curl http://$GATEWAY_URL/productpage
   ```

   刷新几次浏览器，或者重复执行几次命令，产生一定数量的流量。

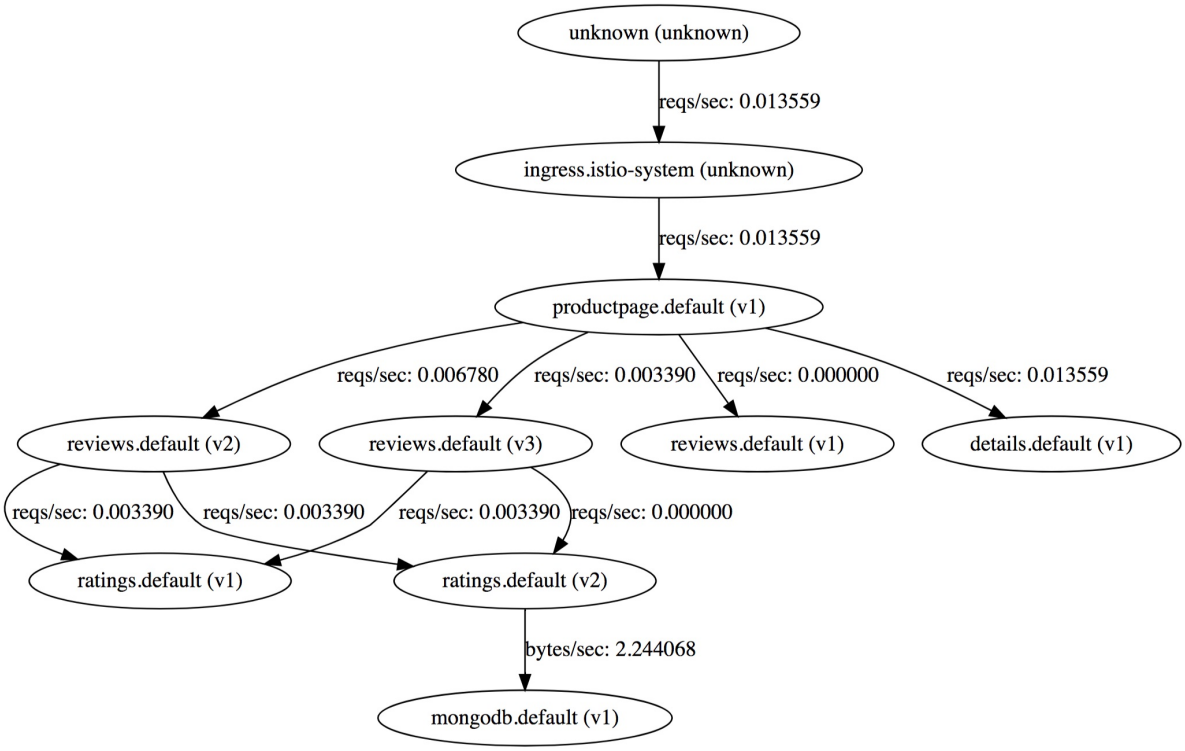
   注意： `$GATEWAY_URL` 的来由可参看BookInfo指南。

4. 打开Servicegraph UI。

   如果是Kubernetes环境，执行下面的命令：

   ```
   kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l app=servicegraph -o jsonpath='{.items[0].metadata.name}') 8088:8088 &
   ```

   使用浏览器访问http://localhost:8088/dotviz，浏览器中会展示类似的图形：



关于Servicegraph插件

Servicegraph服务是一个示例服务，他提供了一个生成和展现Service Mesh中服务关系的功能，包含如下几个服务端点：

- `/graph`：提供了servicegraph的JSON序列化展示
- `/dotgraph`：提供了servicegraph的Dot序列化展示
- `dotviz`：提供了servicegraph的可视化展示

所有的端点都可以使用一个可选参数`time_horizon`，这个参数控制图形生成的时间跨度。

另外一个可选参数就是`filter_empty=true`，在`time_horizon`所限定的时间段内，这一参数可以限制只显示流量大于零的node和edge。

Servicegraph示例构建在Prometheus查询之上。

清理

- 在Kubernetes环境中，执行下面的命令可以移除ServiceGraph插件：

```
  kubectl delete -f install/kubernetes/addons/servicegraph.y
aml
```

- 如果不想继续后续内容，参考BookInfo清理的介绍来关闭应用。

使用 Fluentd 记录日志

此任务将展示如何配置 Istio 创建自定义日志条目并且发送给 Fluentd 守护进程。Fluentd 是一个开源的日志收集器，支持多种数据输出并且有一个可插拔架构。Elasticsearch是一个流行的后端日志记录程序，Kibana 用于查看。在任务结束后，一个新的日志流将被加载发送日志到示例 Fluentd/Elasticsearch/Kibana 栈。

在任务中，将使用 BookInfo 示例应用程序作为示例应用程序。

在开始之前

- 安装 Istio 到您的集群并且部署一个应用程序。这个任务假定 Mixer 是 以默认配置设置的(`--configDefaultNamespace=istio-system`)。如果您使用不同的值，则更新此任务中的配置和命令以匹配对应的值。

安装 Fluentd

在您的群集中，您可能已经有一个 Fluentd DaemonSet 运行，就像 add-on 中这里和这里的描述, 或者特定于您的集群提供者的东西。这可能配置为将日志发送到 Elasticsearch 系统或其它日志记录提供程序。

您可以使用这些 Fluentd 守护进程或您已经设置的任何其他Fluentd守护进程，只要Fluentd守护进程正在侦听转发的日志, 并且 Istio 的 Mixer 可以连接Fluentd守护进程。为了让 Mixer 连接到正在运行的 Fluentd 守护进程, 您可能需要为 Fluentd 添加 service. 监听转发日志的 Fluentd 配置是:

```
<source>
  type forward
</source>
```

将 Mixer 连接到所有可能 Fluentd 配置的完整细节超出了此任务的范围。

Fluentd, Elasticsearch, Kibana 栈示例

为了这个任务的准备，您可以部署提供的示例栈。 该栈包括 Fluentd，Elasticsearch 和 Kibana 在一个非生产集合 Services 和 Deployments 在一个新的叫做 `logging` 的 Namespace 中.

将下面的内容保存为 `logging-stack.yaml` .

```
# Logging Namespace. All below are a part of this namespace.
apiVersion: v1
kind: Namespace
metadata:
  name: logging
---
# Elasticsearch Service
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
  namespace: logging
  labels:
    app: elasticsearch
spec:
  ports:
  - port: 9200
    protocol: TCP
    targetPort: db
  selector:
    app: elasticsearch
---
# Elasticsearch Deployment
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: elasticsearch
  namespace: logging
  labels:
    app: elasticsearch
  annotations:
    sidecar.istio.io/inject: "false"
spec:
  template:
```

```
    metadata:
      labels:
        app: elasticsearch
    spec:
      containers:
      - image: docker.elastic.co/elasticsearch/elasticsearch-oss
:6.1.1
        name: elasticsearch
        resources:
          # need more cpu upon initialization, therefore burstab
le class
          limits:
            cpu: 1000m
          requests:
            cpu: 100m
        env:
          - name: discovery.type
            value: single-node
        ports:
        - containerPort: 9200
          name: db
          protocol: TCP
        - containerPort: 9300
          name: transport
          protocol: TCP
        volumeMounts:
        - name: elasticsearch
          mountPath: /data
      volumes:
      - name: elasticsearch
        emptyDir: {}
---
# Fluentd Service
apiVersion: v1
kind: Service
metadata:
  name: fluentd-es
  namespace: logging
  labels:
    app: fluentd-es
```

```
spec:
  ports:
  - name: fluentd-tcp
    port: 24224
    protocol: TCP
    targetPort: 24224
  - name: fluentd-udp
    port: 24224
    protocol: UDP
    targetPort: 24224
  selector:
    app: fluentd-es
---
# Fluentd Deployment
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: fluentd-es
  namespace: logging
  labels:
    app: fluentd-es
  annotations:
    sidecar.istio.io/inject: "false"
spec:
  template:
    metadata:
      labels:
        app: fluentd-es
    spec:
      containers:
      - name: fluentd-es
        image: gcr.io/google-containers/fluentd-elasticsearch:v2
.0.1
        env:
        - name: FLUENTD_ARGS
          value: --no-supervisor -q
        resources:
          limits:
            memory: 500Mi
          requests:
```

```
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: config-volume
          mountPath: /etc/fluent/config.d
      terminationGracePeriodSeconds: 30
      volumes:
      - name: config-volume
        configMap:
          name: fluentd-es-config
---
# Fluentd ConfigMap, contains config files.
kind: ConfigMap
apiVersion: v1
data:
  forward.input.conf: |-
    # Takes the messages sent over TCP
    <source>
      type forward
    </source>
  output.conf: |-
    <match **>
       type elasticsearch
       log_level info
       include_tag_key true
       host elasticsearch
       port 9200
       logstash_format true
       # Set the chunk limits.
       buffer_chunk_limit 2M
       buffer_queue_limit 8
       flush_interval 5s
       # Never wait longer than 5 minutes between retries.
       max_retry_wait 30
       # Disable the limit on the number of retries (retry forev
er).
       disable_retry_limit
       # Use multiple threads for processing.
       num_threads 2
    </match>
```

```
metadata:
  name: fluentd-es-config
  namespace: logging
---
# Kibana Service
apiVersion: v1
kind: Service
metadata:
  name: kibana
  namespace: logging
  labels:
    app: kibana
spec:
  ports:
  - port: 5601
    protocol: TCP
    targetPort: ui
  selector:
    app: kibana
---
# Kibana Deployment
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kibana
  namespace: logging
  labels:
    app: kibana
  annotations:
    sidecar.istio.io/inject: "false"
spec:
  template:
    metadata:
      labels:
        app: kibana
    spec:
      containers:
      - name: kibana
        image: docker.elastic.co/kibana/kibana-oss:6.1.1
        resources:
```

```
          # need more cpu upon initialization, therefore burstab
le class
          limits:
            cpu: 1000m
          requests:
            cpu: 100m
        env:
          - name: ELASTICSEARCH_URL
            value: http://elasticsearch:9200
        ports:
        - containerPort: 5601
          name: ui
          protocol: TCP
---
```

创建资源:

```
kubectl apply -f logging-stack.yaml
```

你应该看到以下内容:

```
namespace "logging" created
service "elasticsearch" created
deployment "elasticsearch" created
service "fluentd-es" created
deployment "fluentd-es" created
configmap "fluentd-es-config" created
service "kibana" created
deployment "kibana" created
```

配置 Istio

现在有一个正在运行的 Fluentd 守护进程，请使用新的日志类型配置 Istio， 并将这些日志发送到监听守护进程。 创建一个新的 YAML 文件来保存日志流的配置，Istio 将自动生成并收集。

将下面的内容保存为 `fluentd-istio.yaml`：

```
# Configuration for logentry instances
apiVersion: "config.istio.io/v1alpha2"
kind: logentry
metadata:
  name: newlog
  namespace: istio-system
spec:
  severity: '"info"'
  timestamp: request.time
  variables:
    source: source.labels["app"] | source.service | "unknown"
    user: source.user | "unknown"
    destination: destination.labels["app"] | destination.service
 | "unknown"
    responseCode: response.code | 0
    responseSize: response.size | 0
    latency: response.duration | "0ms"
  monitored_resource_type: '"UNSPECIFIED"'
---
# Configuration for a fluentd handler
apiVersion: "config.istio.io/v1alpha2"
kind: fluentd
metadata:
  name: handler
  namespace: istio-system
spec:
  address: "fluentd-es.logging:24224"
---
# Rule to send logentry instances to the fluentd handler
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: newlogtofluentd
  namespace: istio-system
spec:
  match: "true" # match for all requests
  actions:
   - handler: handler.fluentd
     instances:
     - newlog.logentry
```

```
---
```

创建资源:

```
istioctl create -f fluentd-istio.yaml
```

预期的输出类似于:

```
Created config logentry/istio-system/newlog at revision 22374
Created config fluentd/istio-system/handler at revision 22375
Created config rule/istio-system/newlogtofluentd at revision 223
76
```

请注意在处理程序配置中 `address: "fluentd-es.logging:24224"` 行指向我们设置的Fluentd守护进程示例栈。

查看新的日志

1. 将流量发送到示例应用程序。

   对于 BookInfo 示例, 在浏览器中访问 `http://$GATEWAY_URL/productpage` 或发送以下命令:

   ```
   curl http://$GATEWAY_URL/productpage
   ```

2. 在 Kubernetes 环境中, 通过以下命令为 Kibana 建立端口转发:

   ```
   kubectl -n logging port-forward $(kubectl -n logging get pod
   -l app=kibana -o jsonpath='{.items[0].metadata.name}') 5601:
   5601
   ```

   退出运行命令。完成访问 Kibana UI 时输入 Ctrl-C 退出。

3. 导航到 Kibana UI 并点击 右上角的 "Set up index patterns"。

4. 使用 `*` 作为索引模式, 并单击 "Next step."。

5. 选择 `@timestamp` 作为时间筛选字段名称，然后单击 "Create index pattern"。
6. 现在在左侧的菜单上点击 "Discover"，并开始检索生成的日志。

清理

- 删除新的遥测配置:

```
istioctl delete -f fluentd-istio.yaml
```

- 删除 Fluentd, Elasticsearch, Kibana 示例栈:

```
kubectl delete -f logging-stack.yaml
```

- 如果您不打算探索任何后续任务, 参考 BookInfo cleanup 关闭应用程序的说明。

进一步阅读

- 收集指标和日志 有关日志配置的详细说明。
- 学习更多关于 Mixer 和 Mixer 配置。
- 查看完整 属性词汇。
- 阅读参考指南 编写配置。

安全

描述帮助保护service mesh流量的任务。

验证Istio双向TLS认证。这个任务展示如何验证并测试Istio的自动交互TLS认证。

配置基础访问控制。这个任务展示如何使用Kubernetes标签控制对服务的访问。

配置安全访问控制。这个任务展示如何使用服务账号来安全的控制对服务的访问。

启用每服务双向认证。这个任务展示如何为单个服务改变双向TLS认证。

插入CA证书和密钥。这个任务展示运维人员如何插入已有证书和密钥到Istio CA中。

验证Istio双向TLS认证

通过本任务，将学习如何：

- 验证Istio双向TLS认证配置
- 手动测试认证

开始之前

本任务假设已有一个Kubernetes集群：

- 已根据Istio安装任务安装Istio并支持双向TLS认证。

  注意，要在"Installation steps"中的第5步选择"enable Istio mutual TLS Authentication feature"。

验证Istio双向TLS认证配置

以下命令假设服务部署在默认命名空间。使用参数*-n yournamespace*来指定其他命名空间。

验证Istio CA

验证集群级别的CA已在运行：

```
kubectl get deploy -l istio=istio-ca -n istio-system
```

```
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
istio-ca   1         1         1            1           1m
```

如果"AVAILABLE"列为1表示Istio CA在运行。

验证服务安装

1. 验证ConfigMap中的AuthPolicy设置

```
kubectl get configmap istio -o yaml -n istio-system | grep a
uthPolicy | head -1
```

如果 `authPolicy: MUTUAL_TLS` 一行没有被注释（即没有 `#` ），则表示Istio的双向TLS认证是打开的。

测试认证安装

当使用双向TLS认证运行Istio时，可以在服务的envoy中使用curl来给其他服务发送请求。举个例子，启动示例应用BookInfo后，可以ssh到 `productpage` 服务的envoy容器中，并通过curl发送请求到其他服务。

有以下几个步骤：

1. 获取productpage pod名称

```
kubectl get pods -l app=productpage
```

```
NAME                              READY     STATUS    RESTAR
TS    AGE
productpage-v1-4184313719-5mxjc   2/2       Running   0
      23h
```

确认pod是"Running"状态。

2. ssh到envoy容器

```
kubectl exec -it productpage-v1-4184313719-5mxjc -c istio-pr
oxy /bin/bash
```

3. 确认 `key/cert` 在 `/etc/certs/` 目录中

```
ls /etc/certs/
```

```
cert-chain.pem   key.pem   root-cert.pem
```

注意，cert-chain.pem是envoy的证书，需要在另一方使用。key.pem是envoy的与cert-chain.pem配对的私钥。root-cert.pem是根证书，用来验证其他方的证书。目前只有一个CA，因此所有envoy都有相同的root-cert.pem。

4. 发送请求到另一个服务，比如detail：

```
curl https://details:9080/details/0 -v --key /etc/certs/key.
pem --cert /etc/certs/cert-chain.pem --cacert /etc/certs/roo
t-cert.pem -k
```

```
...
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 1867
< server: envoy
< date: Thu, 11 May 2017 18:59:42 GMT
< x-envoy-upstream-service-time: 2
...
```

服务名称和端口定义参见这里。

注意，Istio使用Kubernetes service account 作为服务身份，这比使用服务名称(参考这里得到更多信息)更安全。

因此Istio使用的证书中没有服务名称，而curl需要使用该信息来验证服务身份。结果就是，我们需要使用curl的'-k'选项来阻止curl客户端在服务（比如 productpage）认证中验证服务身份。

请检查安全命名 这里，获取更多关于Istio中客户端如何验证服务器身份的信息。

进阶阅读

- 要学习Istio服务间双向mTLS认证机制之后的设计原理，请看这个博客。

配置基础访问控制

下面的任务展示了如何使用Kubernetes标签来控制对一个服务的访问。

开始之前

- 在Kubernetes上遵循安装指南部署 Istio。
- 部署BookInfo 示例应用。
- 设置基于版本的应用路由，用户“jason”对 `reviews` 服务的访问会被指向 v2 版本，其他用户则会访问到 v3 版本。

```
  istioctl create -f samples/bookinfo/kube/route-rule-reviews-test-v2.yaml
  istioctl create -f samples/bookinfo/kube/route-rule-reviews-v3.yaml
```

> 注意：如果在前面的任务中存在有冲突的规则，可以用 `istiocl replace` 来替代 `istioctl create` 。
>
> 注意：如果使用一个非 `default` 的命名空间，需要使用 `istioctl -n namespace ...` 的方式来指定命名空间

使用 `denials` 进行访问控制

借助Istio能够根据Mixer中的任何属性来对一个服务进行访问控制。Mixer Selector 可以进行对服务请求进行有条件的拒绝，这就构成了访问控制能力的基础。

BookInfo 示例应用中的 `ratings` 服务会被几个不同版本的 `reviews` 服务所访问。我们尝试切断 `reviews` 服务的 `v3` 版本对 `ratings` 服务的访问。

1. 用浏览器访问BookInfo的 `productpage` 页面（http://$GATEWAY_URL/productpage）。

如果使用 "jason" 用户登录，会看到 `ratings` 服务展示出的星星图标是黑色的，这证明被 `ratings` 服务是由 "v2" 版本的 `reviews` 服务调用的；如果登出或者使用其他用户登录，就会看到红色的星星，这代表 "v3" 版本的 `reviews` 服务在调用 `ratings` 服务。

2. 显式的拒绝从 `v3` 版本 `reviews` 服务到 `ratings` 的访问。

   使用下列命令来创建Handler实例并拒绝规则：

   ```
    istioctl create -f samples/bookinfo/kube/mixer-rule-deny-la
   bel.yaml
   ```

   会产生类似的输出：

   ```
    Created config denier/default/denyreviewsv3handler at revis
   ion 2882105
    Created config checknothing/default/denyreviewsv3request at
    revision 2882106
    Created config rule/default/denyreviewsv3 at revision 28821
   07
   ```

   注意 `denyreviewsv3` 规则：

   ```
   match: destination.labels["app"] == "ratings" && source.labe
   ls["app"]=="reviews" && source.labels["version"] == "v3"
   ```

   代表从 `v3` 版本的 `reviews` 服务到 `ratings` 服务的请求。

   这一规则使用 `denier` 适配器来拒绝源于 `v3` 版本的 `reviews` 服务的请求。这一适配器会使用预置的状态码和消息来拒绝服务请求。状态码和消息的定义来自于 denier适配器的配置。

3. 在浏览器中刷新 `productpage`

   如果没有登录、或使用 "jason" 之外的用户登录，因为 `ratings` 服务拒绝了来自 `reviews:v3` 的请求。

   而如果使用 "jason" 的身份登录（会使用 `reviews:v2` ），就会看到黑色的星星了。

使用 `whitelists` 进行访问控制

Istio还支持基于属性的黑名单和白名单。下面的白名单配置是跟上一节中的 `denier` 配置等价的。这一规则也会拒绝来自 `v3` 版本 `reviews` 服务的请求。

1. 删除上一节加入的配置：

```
 istioctl delete -f samples/bookinfo/kube/mixer-rule-deny-la
bel.yaml
```

2. 验证未登录情况下对 `productpage` 的访问 (http://$GATEWAY_URL/productpage)，会看到红色星星。

   但是在执行下面的步骤之后，只有使用 "jason" 登录才能看到星星。

3. 创建 `listchecker` 适配器，其中包含 `v1` 和 `v2` 的列表。把下面的 YAML 文件保存为 `whitelist-handler.yaml` ：

```
 apiVersion: config.istio.io/v1alpha2
 kind: listchecker
 metadata:
  name: whitelist
 spec:
  # providerUrl: ordinarily black and white lists are mainta
ined
  # externally and fetched asynchronously using the provider
Url.
  overrides: ["v1", "v2"]  # overrides provide a static list
  blacklist: false
```

   然后运行下列命令：

```
 istioctl create -f whitelist-handler.yaml
```

4. 创建一个 `listentry` 模板的实例，用于解析版本标签。把下面的 YAML 代码保存为 `appversion-instance.yaml` ：

```
apiVersion: config.istio.io/v1alpha2
kind: listentry
metadata:
 name: appversion
spec:
 value: source.labels["version"]
```

接下来运行下列命令：

```
istioctl create -f appversion-instance.yaml
```

5. 启用 `whitelist` 来检查 `ratings` 服务。

   创建 `checkversion-rule.yaml` ：

```
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
 name: checkversion
spec:
 match: destination.labels["app"] == "ratings"
 actions:
 - handler: whitelist.listchecker
   instances:
   - appversion.listentry
```

   并运行下列命令：

```
istioctl create -f checkversion-rule.yaml
```

6. 验证，当未登录时访问Bookinfo `productpage`（http://$GATEWAY_URL/productpage），则看不到星星。

   验证当用 "jason" 登录后，就会看到黑色的星星。

清理

- 删除 Mixer 配置：

```
  istioctl delete -f checkversion-rule.yaml
  istioctl delete -f appversion-instance.yaml
  istioctl delete -f whitelist-handler.yaml
```

- 删除应用路由规则：

```
  istioctl delete -f samples/bookinfo/kube/route-rule-review
s-test-v2.yaml
  istioctl delete -f samples/bookinfo/kube/route-rule-review
s-v3.yaml
```

- 如果不准备进行下面的任务，可以参考BookInfo 清理进行善后工作。

延伸阅读

- 学习如何利用Service Account来进行访问控制：点击这里。
- 深入学习Mixer和Mixer配置。
- 属性词汇表。
- 配置编写指南
- Kubernetes的网络策略和Istio访问控制策略的差异

配置安全访问控制

本任务将演示如何通过使用Istio认证提供的服务账户，来安全地对服务做访问控制。

当Istio双向TLS认证打开时，服务器就会根据其证书来认证客户端，并从证书获取客户端的服务账户。服务账户在 `source.user` 的属性中。请参考Istio auth identity了解Istio中服务账户的格式。

开始之前

- 根据quick start的说明在开启认证的Kubernetes中安装Istio。注意，应当在installation steps中的第四步开启认证。
- 部署BookInfo示例应用。
- 执行下列命令创建 `bookinfo-productpage` 服务账户，并重新部署 `productpage` 服务及其服务账户。

```
  kubectl create -f <(istioctl kube-inject -f samples/bookinfo/kube/bookinfo-add-serviceaccount.yaml)
```

期望看到类似如下的输入：

```
  serviceaccount "bookinfo-productpage" created
  deployment "productpage-v1" configured
```

> 注意：如果使用 `default` 之外的命名空间，需要通过 `istioctl -n namespace ...` 来指定命名空间。

使用*denials*做访问控制

在示例应用BookInfo中， `productpage` 服务会同时访问 `reviews` 服务和 `details` 服务。现在让 `details` 服务拒绝来自 `productpage` 服务的请求：

1. 在浏览器中打开BookInfo的 `productpage` (http://$GATEWAY_URL/productpage)页面。

   将会在页面左下部看到"Book Details"部分，包括了类型、页数、出版社等信息。 `productpage` 服务是从 `details` 服务获取的"Book Details"信息。

2. 明确拒绝从 `productpage` 到 `details` 的请求。

   执行下列命令来安装拒绝规则及对应的handler和实例。

   ```
   istioctl create -f samples/bookinfo/kube/mixer-rule-deny-ser
   viceaccount.yaml
   ```

   将会看到类似下面的输出：

   ```
   Created config denier/default/denyproductpagehandler at revi
   sion 2877836
    Created config checknothing/default/denyproductpagerequest
   at revision 2877837
    Created config rule/default/denyproductpage at revision 287
   7838
   ```

   注意 `denyproductpage` 规则中的下述内容:

   ```
    match: destination.labels["app"] == "details" && source.use
   r == "cluster.local/ns/default/sa/bookinfo-productpage"
   ```

   匹配到了来自details服务上的服务账户 "*spiffe://cluster.local/ns/default/sa/bookinfo-productpage*"的请求。

   > 注意: 如果使用 `default` 之外的命名空间，请在 `source.user` 的值中使用自己的命名空间替换 `default` 。

   该规则使用 `denier` 适配器来拒绝这些请求。适配器通常通过一个预先配置的状态码和消息来拒绝请求。状态码和消息是在denier适配器配置中指定的。

3. 在浏览器中刷新 `productpage` 页面。

   将会看到消息

"*Error fetching product details! Sorry, product details are currently unavailable for this book.*"

出现在页面的左下部分。这就验证了从 `productpage` 到 `details` 的访问是被禁止的。

清除

- 清除mixer配置：

```
  istioctl delete -f samples/bookinfo/kube/mixer-rule-deny-s
erviceaccount.yaml
```

- 如果不打算继续接下来的更多任务，可参考BookInfo cleanup的指南来关闭应用。

进阶阅读

- 了解更多关于Mixer 和 Mixer配置的内容。
- 探索全部的属性字典。
- 阅读参考指南来编写配置。
- 通过博客文章来理解Kubernetes网络策略和Istio访问控制策略的区别。

启用每服务双向认证

在安装指南中，我们展示了如何启用边车之间的双向TLS认证。 这些设置将应用于网格中的所有边车。

在本教程中，您将学习：

- 注解Kubernetes服务以禁用（或启用）一个选定服务的双向TLS身份验证。
- 修改Istio网格配置以解除控制服务的相互TLS身份验证。

在开始前

- 了解Isio双向TLS认证的概念。
- 熟悉验证Istio双向TLS认证。
- 参考安装指南中的说明，通过双向TLS认证安装Istio。
- 使用Istio边车启动httpbin Demo。 同样为了验证目的，启动两个休眠的实例：一个有边车，一个没有（在不同的命名空间）。 以下是帮助您启动这些服务的命令：

```
kubectl apply -f <(istioctl kube-inject -f samples/httpbin/httpb
in.yaml)
kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.y
aml)

kubectl create ns legacy && kubectl apply -f samples/sleep/sleep
.yaml -n legacy
```

在这个初始设置中，我们期望默认命名空间中的休眠实例可以与httpbin服务进行通信，但是传统命名空间中的休眠实例不能，因为它没有边车来使能mTLS。

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items.
.metadata.name}) -c sleep -- curl http://httpbin.default:8000/ip
 -s
```

```
{
  "origin": "127.0.0.1"
}
```

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items.
.metadata.name} -n legacy) -n legacy -- curl http://httpbin.defa
ult:8000/ip -s
```

```
command terminated with exit code 56
```

禁用“httpbin”服务的相互TLS身份验证

如果我们希望在不更改网格验证设置下禁用httpbin（在端口8000上）的mTLS，我们可以通过将此注释添加到httpbin服务定义来实现。

```
annotations:
  auth.istio.io/8000: NONE
```

要进行快速验证，运行**kubectl edit svc httpbin**并添加上面的注解（或者编辑原始的httpbin.yaml文件并重新应用它）。 应用更改之后，由于mTLS已被删除sleep.legacy的请求现在应该是成功的。

注意： 注解可以用于相反的方向，例如：对单个服务启用mTLS，只需使用注解值**MUTUAL_TLS**而非**NONE**。 人们可以使用此选项在选定的服务上启用mTLS，而不是在整个网格中启用它。

注解也可以用于没有边车的（服务端）服务，以指示Istio在对该服务进行调用时不为客户端应用mTLS。 事实上如果一个系统有一些服务不由Istio管理（即没有边车），这是一个推荐的解决方案，以解决与这些服务的通信问题。

禁用控制服务的相互TLS身份验证

由于不能在Istio 0.3中注解控制服务，例如：API服务器。所以在网格配置中引入mtls_excluded_services来指定不应使用mTLS的服务列表。 如果你的应用程序需要与任何控制服务通信，则应在其中列出其完全限定域名（FQDN）。

作为Demo的一部分，我们将展示该字段的影响。

默认情况下（0.3或更高版本），该列表包含**kubernetes.default.svc.cluster.local**（这是通用设置中的API服务器服务的名称）。 你可以通过运行以下命令来验证：

```
kubectl get configmap -n istio-system istio -o yaml | grep mtlsE
xcludedServices
```

```
mtlsExcludedServices: ["kubernetes.default.svc.cluster.local"]
```

然后预计请求kubernetes.default服务应该是可能的：

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items.
.metadata.name}) -c sleep -- curl https://kubernetes.default:443
/api/ -k -s
```

```
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "104.199.122.14"
    }
  ]
}
```

现在运行**kubectl edit configmap istio -n istio-system**并清除mtlsExcludedServices，在完成后重新启动pilot：

```
kubectl get pod $(kubectl get pod -l istio=pilot -n istio-system
 -o jsonpath={.items..metadata.name}) -n istio-system -o yaml |
kubectl replace --force -f -
```

上面相同的测试请求现在失败了（失败码为35），这是由于休眠的边车再次开启了mTLS：

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items.
.metadata.name}) -c sleep -- curl https://kubernetes.default:443
/api/ -k -s
```

```
command terminated with exit code 35
```

插入CA证书和密钥

本任务将演示将现有的证书和密钥添加到 Istio CA 中。

默认情况下，Istio CA 生成自签名 CA 证书和密钥，并使用它们签署工作负载证书。Istio CA 也可以使用我们指定的证书和密钥来签署工作负载证书。接下来我们将演示将证书和密钥插入 Istio CA 的示例。

开始之前

- 根据安装指南中的快速入门指南，在 Kubernetes 集群中安装 Istio。
- 在安装步骤的第四步应该启动身份验证

插入现有的证书和密钥

假设我们想让 Istio CA 使用已经存在的 ca-cert.pem 的证书和密钥。此外，证书 ca-cert.pem 由根证书 root-cert.pem 签名，我们希望使用使用 root-cert.pem 作为 Istio 工作负载的根证书。

在此示例中，由于 Istio CA 证书（ca-cert.pem）未设置为工作负载的根证书（root-cert.pem），因此工作负载无法直接从根证书验证工作负载证书。工作负载需要一个 cert-chain.pem 文件来指定信任链，该信任链应包括工作负载和根 CA 之间的所有中间 CA 的证书。在这个例子中，它只包含 Istio CA 证书，所以 cert-chain.pem 和 ca-cert.pem 是一样的。请注意，如果您的 ca-cert.pem 与 root-cert.pem 相同，您可以有一个空的 cert-chain.pem 文件。

将证书和密钥插入 Istio CA 分为以下几步：

1. 创建一个密钥 cacert 包含所有输入文件，例如：ca-cert.pem，ca-key.pem，root-cert.pem 和 cert-chain.pem

```
kubectl create secret generic cacerts -n istio-system --from
-file=install/kubernetes/ca-cert.pem --from-file=install/kub
ernetes/ca-key.pem \
--from-file=install/kubernetes/root-cert.pem --from-file=ins
tall/kubernetes/cert-chain.pem
```

2. 重新部署 Istio CA，从自定义的安装文件读取证书和密钥

```
kubectl apply -f install/kubernetes/istio-ca-plugin-certs.ya
ml
```

3. 为了确保工作负载及时获得新的证书，请删除 Istio CA 生成的密钥文件（以 istio.* 命名）。在本例中是 istio.default。Istio CA 将为工作负载颁发新的证书。

```
kubectl delete secret istio.default
```

请注意，如果你使用不同的证书/密钥文件或密钥名称，则需要更改 istio-ca-plugin-certs.yaml 中的相应参数。

验证新的证书

在本节中，我们将验证新的工作负载证书和根证书是否生效，这需要你的机器上安装 OpenSSL。

1. 按照说明部署 bookinfo 应用程序
2. 检索挂载的证书

   获取 pods：

```
kubectl get pods
```

   列举 producees：

```
NAME                                        READY     STATUS
    RESTARTS   AGE
details-v1-1520924117-48z17                 2/2       Runnin
g   0          6m
productpage-v1-560495357-jk1lz              2/2       Runnin
g   0          6m
ratings-v1-734492171-rnr5l                  2/2       Runnin
g   0          6m
reviews-v1-874083890-f0qf0                  2/2       Runnin
g   0          6m
reviews-v2-1343845940-b34q5                 2/2       Runnin
g   0          6m
reviews-v3-1813607990-8ch52                 2/2       Runnin
g   0          6m
```

接下来，我们以 pod ratings-v1-734492171-rnr5l 为例，并验证已安装的证书。运行以下命令以检索安装在代理上的证书。

```
kubectl exec -it ratings-v1-734492171-rnr5l -c istio-proxy -
- /bin/cat /etc/certs/root-cert.pem > /tmp/pod-root-cert.pem
```

文件 /tmp/pod-root-cert.pem 应该包含由操作员指定的根证书。

```
kubectl exec -it ratings-v1-734492171-rnr5l -c istio-proxy -
- /bin/cat /etc/certs/cert-chain.pem > /tmp/pod-cert-chain.p
em
```

文件 /tmp/pod-cert-chain.pem 应该包含工作负载证书和CA证书。

3. 验证根证书是否与操作员指定的相同：

```
openssl x509 -in /tmp/root-cert.pem -text -noout > /tmp/root
-cert.crt.txt
openssl x509 -in /tmp/pod-root-cert.pem -text -noout > /tmp/
pod-root-cert.crt.txt
diff /tmp/root-cert.crt.txt /tmp/pod-root-cert.crt.txt
```

4. 验证CA证书是否与操作员指定的相同：

```
tail /tmp/pod-cert-chain.pem -n 22 > /tmp/pod-cert-chain-ca.
pem
openssl x509 -in /tmp/ca-cert.pem -text -noout > /tmp/ca-cer
t.crt.txt
openssl x509 -in /tmp/pod-cert-chain-ca.pem -text -noout > /
tmp/pod-cert-chain-ca.crt.txt
diff /tmp/ca-cert.crt.txt /tmp/pod-cert-chain-ca.crt.txt
```

   预计输出为空。

5. 验证从根证书到工作负载证书的证书链：

```
head /tmp/pod-cert-chain.pem -n 18 > /tmp/pod-cert-chain-wor
kload.pem
openssl verify -CAfile <(cat /tmp/ca-cert.pem /tmp/root-cert
.pem) /tmp/pod-cert-chain-workload.pem
```

   预计输出为：

```
/tmp/pod-cert-chain-workload.pem: OK
```

清理

- 删除密钥 cacerts。

https://istio.io/docs/tasks/security/plugin-ca-cert.html

指南

指南中包括多个 Istio 使用的可完整工作的示例，你可以用来亲自部署和体验这些示例。

- Bookinfo：该示例部署由四个单独的微服务组成的简单应用程序，用于演示Istio服务网格的各种功能。
- 智能路由：这个指南演示如何使用Istio service mesh的多种流量管理能力。
- 深入遥测：这个示例演示如何使用Istio Mixer和Istio sidecar获得统一的指标，日志，跨不同服务的追踪。
- 集成虚拟机：该示例跨越 Kubernetes 集群和一组虚拟机上部署 Bookinfo 服务，描述如何使用 Istio service mesh 将此基础架构以单一 mesh 的方式操控。

BookInfo

该示例由四个独立的微服务组成，用于演示Istio服务网格的功能。

概况

在本示例中，我们将部署一个简单的应用程序，显示书籍的信息，类似于网上书店的书籍条目。在页面上有书籍的描述、详细信息（ISBN、页数等）和书评。

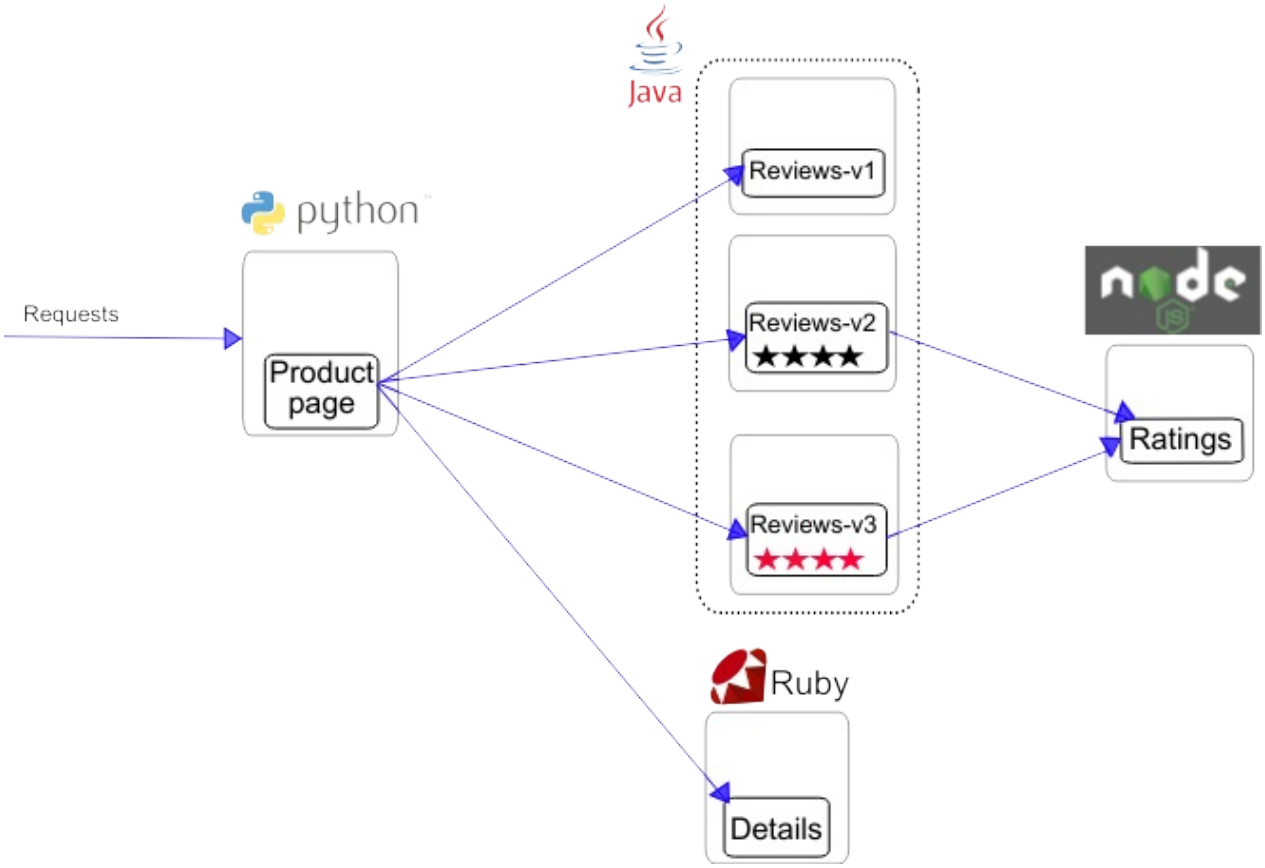
BookInfo 应用程序包括四个独立的微服务：

- *productpage*：productpage(产品页面)微服务，调用 *details* 和 *reviews* 微服务来填充页面。
- *details*：details 微服务包含书籍的详细信息。
- *reviews*：reviews 微服务包含书籍的点评。它也调用 *ratings* 微服务。
- *ratings*：ratings 微服务包含随书评一起出现的评分信息。

有3个版本的 reviews 微服务：

- 版本v1不调用 ratings 服务。
- 版本v2调用 ratings ，并将每个评级显示为1到5个黑色星。
- 版本v3调用 ratings ，并将每个评级显示为1到5个红色星。

应用程序的端到端架构如下所示。



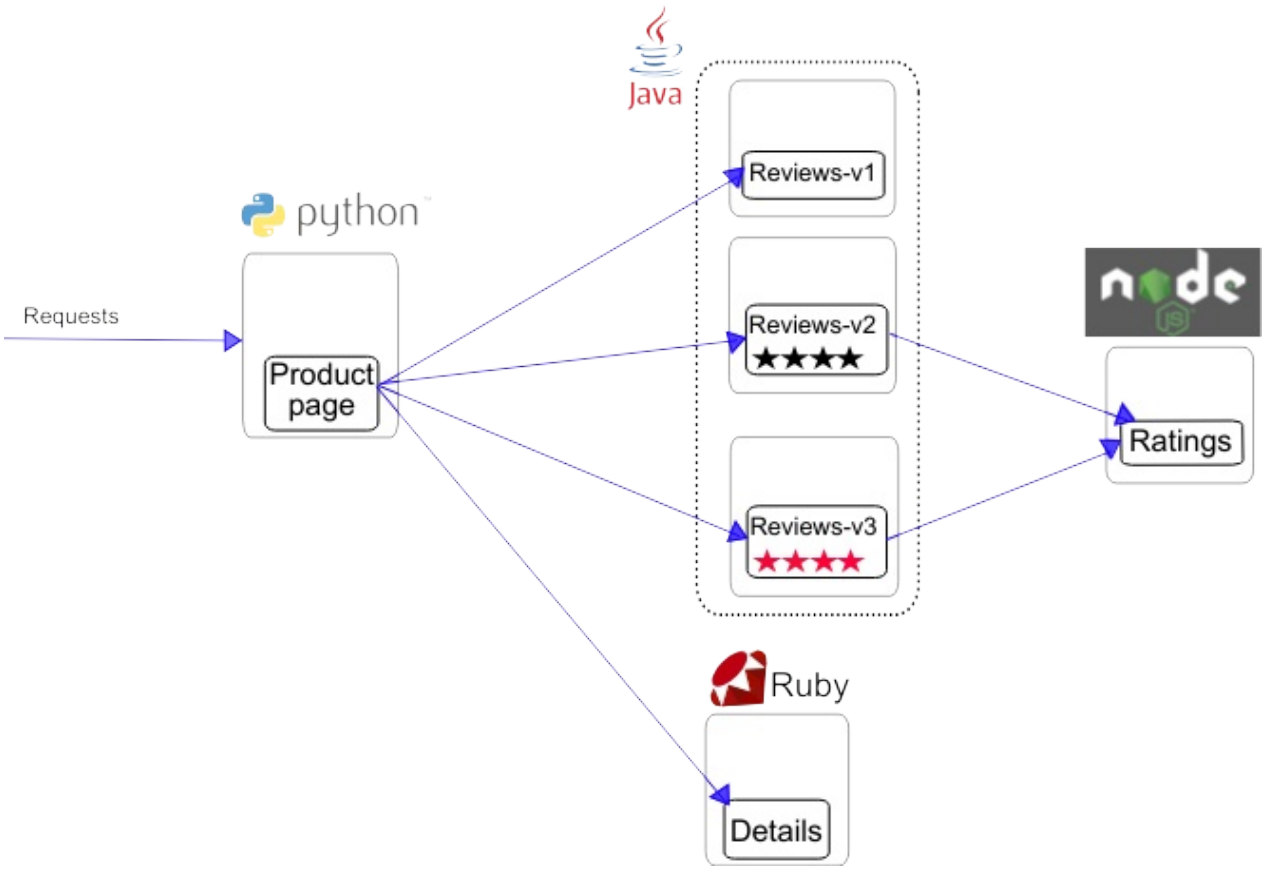
该应用程序由多语言实现，即这些微服务使用不同的语言编写。值得注意的是，这些服务与 Istio 没有任何依赖关系。这是个有趣的 Service Mesh 示例，特别是对于评论服务，有多语言的版本。

开始之前

如果您还没有这样做，请按照 安装指南 对应的说明安装Istio。

部署应用程序

使用 Istio 运行应用程序示例不需要修改应用程序本身。相反，我们只需要在支持 Istio 的环境中配置和运行服务， Envoy sidecar 将会注入到每个服务中。所需的命令和配置根据运行时环境的不同而有所不同，但在所有情况下，生成的部署将如下所示：



所有的微服务都将与一个 Envoy sidecar 一起打包，拦截这些服务的入站和出站的调用请求，提供通过 Istio 控制平面从外部控制整个应用的路由，遥测收集和策略执行所需的 hook。

要启动该应用程序，请按照以下对应于您的 Istio 运行时环境的说明进行操作。

在 **Kubernetes** 中运行

> 注意：如果您使用 GKE，请确保您的集群至少有 4 个标准的 GKE 节点。如果您使用 Minikube，请确保您至少有 4GB 内存。

1. 将目录更改为 Istio 安装目录的根目录。
2. 构建应用程序容器：

   如果您使用 自动注入 **sidecar** 的方式部署的集群，那么只需要使用 `kubectl` 命令部署服务：

   ```
   kubectl apply -f samples/bookinfo/kube/bookinfo.yaml
   ```

   如果您使用 手动注入 **sidecar** 的方式部署的集群，清使用下面的命令：

```
 kubectl apply -f <(istioctl kube-inject -f samples/bookinfo
/kube/bookinfo.yaml)
```

请注意，该 `istioctl kube-inject` 命令用于在创建部署之前修改 `bookinfo.yaml` 文件。这将把 Envoy 注入到 Kubernetes 资源,如 这里 记载的。

上述命令启动四个微服务并创建网关入口资源，如下图所示。3 个版本的评论的服务 v1、v2、v3 都已启动。

> 请注意在实际部署中，随着时间的推移部署新版本的微服务，而不是同时部署所有版本。

3. 确认所有服务和 pod 已正确定义并运行：

```
 kubectl get services
```

这将产生以下输出：

```
 NAME                       CLUSTER-IP   EXTERNAL-IP   PORT(
S)              AGE
 details                    10.0.0.31    <none>        9080/
TCP             6m
 kubernetes                 10.0.0.1     <none>        443/T
CP              7d
 productpage                10.0.0.120   <none>        9080/
TCP             6m
 ratings                    10.0.0.15    <none>        9080/
TCP             6m
 reviews                    10.0.0.170   <none>        9080/
TCP             6m
```

而且

```
 kubectl get pods
```

将产生:

```
 NAME                                        READY     STATU
S     RESTARTS   AGE
 details-v1-1520924117-48z17                 2/2       Runni
ng    0          6m
 productpage-v1-560495357-jk1lz              2/2       Runni
ng    0          6m
 ratings-v1-734492171-rnr5l                  2/2       Runni
ng    0          6m
 reviews-v1-874083890-f0qf0                  2/2       Runni
ng    0          6m
 reviews-v2-1343845940-b34q5                 2/2       Runni
ng    0          6m
 reviews-v3-1813607990-8ch52                 2/2       Runni
ng    0          6m
```

确定 ingress IP 和端口

4. 如果您的 kubernetes 集群环境支持外部负载均衡器的话，可以使用下面的命令获取 ingress 的IP地址：

```
kubectl get ingress -o wide
```

输出如下所示：

```
NAME      HOSTS     ADDRESS                 PORTS     AGE
gateway   *         130.211.10.121          80        1d
```

Ingress 服务的地址是：

```
export GATEWAY_URL=130.211.10.121:80
```

5. *GKE*：如果服务无法获取外部 IP，`kubectl get ingress -o wide` 会显示工作节点的列表。在这种情况下，您可以使用任何地址以及 NodePort 访问入口。但是，如果集群具有防火墙，则还需要创建防火墙规则以允许TCP流量到NodePort，您可以使用以下命令创建防火墙规则：

```
export GATEWAY_URL=<workerNodeAddress>:$(kubectl get svc ist
io-ingress -n istio-system -o jsonpath='{.spec.ports[0].node
Port}')
gcloud compute firewall-rules create allow-book --allow tcp:
$(kubectl get svc istio-ingress -n istio-system -o jsonpath=
'{.spec.ports[0].nodePort}')
```

6. *IBM Bluemix Free Tier*：在免费版的 Bluemix 的 kubernetes 集群中不支持外部负载均衡器。您可以使用工作节点的公共 IP，并通过 NodePort 来访问 ingress。工作节点的公共 IP可以通过如下命令获取：

```
 bx cs workers <cluster-name or id>
 export GATEWAY_URL=<public IP of the worker node>:$(kubectl
 get svc istio-ingress -n istio-system -o jsonpath='{.spec.p
orts[0].nodePort}')
```

7. *Minikube*：Minikube 不支持外部负载均衡器。您可以使用 ingress 服务的主机 IP 和 NodePort 来访问 ingress：

```
 export GATEWAY_URL=$(kubectl get po -l istio=ingress -n ist
io-system -o 'jsonpath={.items[0].status.hostIP}'):$(kubectl
 get svc istio-ingress -n istio-system -o 'jsonpath={.spec.p
orts[0].nodePort}')
```

在 Consul 或 Eureka 环境下使用 Docker 运行

1. 切换到 Istio 的安装根目录下。
2. 启动应用程序容器。
   i. 执行下面的命令测试 Consul：

```
 docker-compose -f samples/bookinfo/consul/bookinfo.yaml
up -d
```

   ii. 执行下面的命令测试 Eureka：

```
 docker-compose -f samples/bookinfo/eureka/bookinfo.yaml
up -d
```

3. 确认所有容器都在运行：

```
docker ps -a
```

> 如果 Istio Pilot 容器终止了，重新执行上面的命令重新运行。

4. 设置 `GATEWAY_URL`：

```
export GATEWAY_URL=localhost:9081
```

下一步

使用以下 `curl` 命令确认 BookInfo 应用程序正在运行:

```
curl -o /dev/null -s -w "%{http_code}\n" http://${GATEWAY_URL}/p
roductpage
```

```
200
```

你也可以通过在浏览器中打开 `http://$GATEWAY_URL/productpage` 页面访问 Bookinfo 网页。如果您多次刷新浏览器将在 productpage 中看到评论的不同的版本，它们会按照 round robin（红星、黑星、没有星星）的方式展现，因为我们还没有使用 Istio 来控制版本的路由。

现在，您可以使用此示例来尝试 Istio 的流量路由、故障注入、速率限制等功能。要继续的话，请参阅 Istio 指南，具体取决于您的兴趣。智能路由 是初学者入门的好方式。

清理

在完成 BookInfo 示例后，您可以卸载它，如下所示：

卸载 Kubernetes 环境

1. 删除路由规则，终止应用程序 pod

```
samples/bookinfo/kube/cleanup.sh
```

2. 确认关闭

```
istioctl get routerules   #-- there should be no more routing rules
kubectl get pods          #-- the BookInfo pods should be deleted
```

卸载 docker 环境

1. 删除路由规则和应用程序容器

   i. 若使用 Consul 环境安装，执行下面的命令：

   ```
   samples/bookinfo/consul/cleanup.sh
   ```

   ii. 若使用 Eureka 环境安装，执行下面的命令：

   ```
   samples/bookinfo/eureka/cleanup.sh
   ```

2. 确认清理完成：

```
istioctl get routerules   #-- there should be no more routing rules
docker ps -a              #-- the BookInfo containers should be deleted
```

智能路由

这一指南演示了Istio Service Mesh的通信管理能力。

概述

把微服务应用部署到Istio Service Mesh集群上，就可以在外部控制服务的监控、跟踪、（版本相关的）请求路由、弹性测试、安全和策略增强等，并且可以跨越服务界限，从整个应用的层面进行管理。

本文章将会使用Bookinfo示例应用来进行展示，看一个运维人员如何为一个运行中的应用动态的配置请求路由以及错误注入。

开始之前

- 依据安装指南安装Istio控制平面。
- 按照应用部署指南运行BookInfo示例应用。

任务

1. 请求路由：这一任务会首先把所有进入Bookinfo的访问请求指向 `review` 服务的v1版本。然后会在不影响其他用户的情况下，把特定测试用户的请求发送到 `review` 服务的v2版本去。
2. 错误注入：我们接下来会在 `reviews:v2` 和 `ratings` 服务之间假造一个延时，以此来测试Bookinfo应用的弹性。观察测试用户的行为，我们会注意到v2版本的 `reviews` 服务有个bug。注意所有其他用户对这一在线系统的测试都是毫不知情的。
3. 流量转移：最后我们修复了v2版本 `reviews` 服务的问题，使用Istio优雅的把所有用户的流量转移到v3版本的 `reviews` 上。

清理

Bookinfo试验结束之后，可以根据Bookinfo清理指南的介绍来清理测试环境。

深入遥测

这个例子演示了如何使用Istio Mixer和Istio Sidecar，从多个服务中获取一致的指标、日志、跟踪信息。

概述

把微服务应用部署到Istio Service Mesh集群上，就可以在外部控制服务的监控、跟踪、（版本相关的）请求路由、弹性测试、安全和策略增强等，并且可以跨越服务界限，从整个应用的层面进行管理。

本文将会使用Bookinfo示例应用，展示在无需开发改动代码的情况下，运维人员如何从运行中的多语言平台应用获取一致的指标和跟踪信息。

开始之前

- 依据安装指南安装Istio控制平面。
- 按照应用部署指南运行BookInfo示例应用。

任务

1. 指标收集：该任务会对Mixer进行配置，用统一方式从Bookinfo应用的所有服务中收集一套指标。
2. 指标查询：这里安装并配置了Prometheus，用于Istio指标的收集、查询和展示。
3. 分布式跟踪：现在我们要使用Istio来跟踪请求在一个应用的不同服务中的流向。分布式跟踪让开发者能够快速的理解最终用户所感受的延时在各个不同服务之间的分布，这一功能对分布式应用的检测和排错工作也是很有价值的。
4. 使用Istio仪表盘：这个任务安装了Grafana扩展，其中带有一个用于监控Mesh流量的预定义仪表盘。

清理

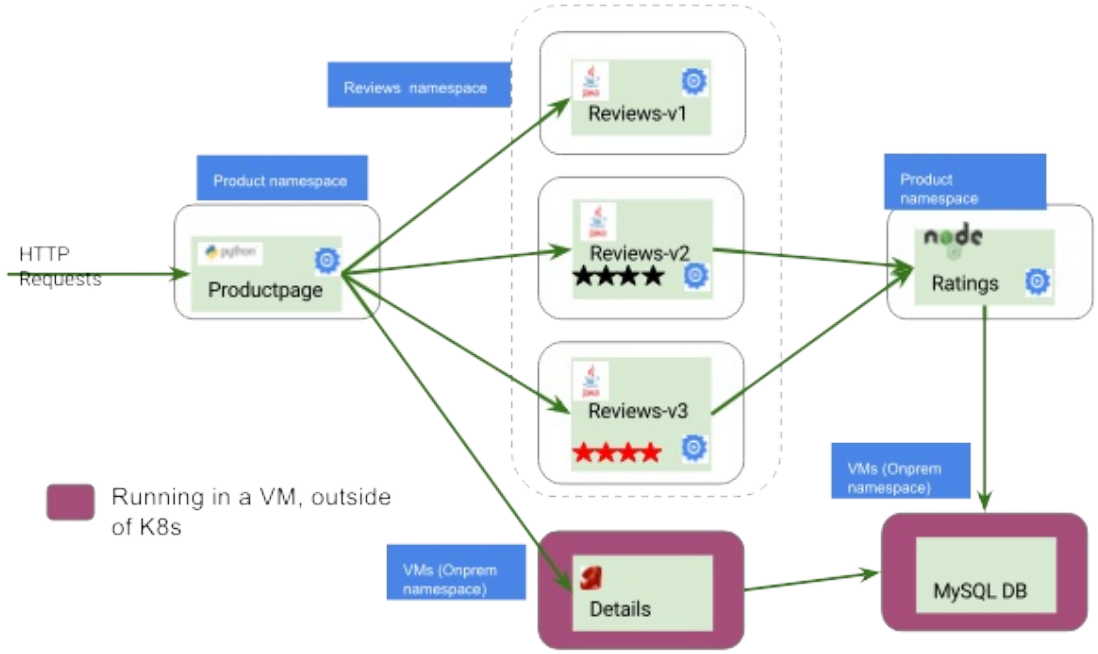
Bookinfo试验结束之后，可以根据Bookinfo清理指南的介绍来清理测试环境。

集成虚拟机

该示例跨越 Kubernetes 集群和一组虚拟机上部署 Bookinfo 服务，描述如何使用 Istio service mesh 将此基础架构以单一 mesh 的方式操控。

> 注意：本文档还在建设中，并且只在 Google Cloud Platform 上进行过测试。在 IBM Bluemix 或其它平台上，pod 的 overlay 网络跟虚拟机的网络是隔离的。即使使用 Istio，虚拟机也不能直接与 Kubernetes Pod 进行通信。

概览



开始之前

- 按照 安装指南 上的步骤部署 Istio。
- 部署 BookInfo 示例应用程序（在 `bookinfo` namespace 下）。
- 在 Istio 集群相同的项目下创建名为 `vm-1` 的虚拟机，并 加入到 Mesh。

在虚拟机上运行 **mysql**

我们将首先在虚拟机上安装 mysql，将其配置成评分服务的后台存储。

在虚拟机上执行：

```
sudo apt-get update && sudo apt-get install -y mariadb-server
sudo mysql
# 授权 root 用户访问
GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' IDENTIFIED BY
'password' WITH GRANT OPTION;
quit;
sudo systemctl restart mysql
```

关于 Mysql 的详细配置请见： Mysql。

在虚拟机上执行下面的命令，向 mysql 中添加评分数据库。

```
# 向 mysql 数据库中添加评分数据库
curl -q https://raw.githubusercontent.com/istio/istio/master/sam
ples/bookinfo/src/mysql/mysqldb-init.sql | mysql -u root -ppassw
ord
```

为了便于直观地查看bookinfo应用程序输出的差异，可以更改使用以下命令生成的评分：

```
 # 查看评分
 mysql -u root -ppassword test -e "select * from ratings;"
 +----------+--------+
 | ReviewID | Rating |
 +----------+--------+
 |        1 |      5 |
 |        2 |      4 |
 +----------+--------+
 # 修改评分
 mysql -u root -ppassword test -e  "update ratings set rating=1
where reviewid=1;select * from ratings;"
+----------+--------+
| ReviewID | Rating |
+----------+--------+
|        1 |      1 |
|        2 |      4 |
+----------+--------+
```

找出将添加到 mesh 中的虚拟机的 IP 地址

在虚拟机上执行：

```
hostname -I
```

将 mysql 服务注册到 mesh 中

在一台可以访问 `istioctl` 命令的主机上，注册该虚拟机和 mysql db service：

```
istioctl register -n vm mysqldb <ip-address-of-vm> 3306
# 示例输出
$ istioctl register mysqldb 192.168.56.112 3306
I1015 22:24:33.846492   15465 register.go:44] Registering for se
rvice 'mysqldb' ip '192.168.56.112', ports list [{3306 mysql}]
I1015 22:24:33.846550   15465 register.go:49] 0 labels ([]) and
1 annotations ([alpha.istio.io/kubernetes-serviceaccounts=defaul
t])
W1015 22:24:33.866410   15465 register.go:123] Got 'services "my
sqldb" not found' looking up svc 'mysqldb' in namespace 'default'
, attempting to create it
W1015 22:24:33.904162   15465 register.go:139] Got 'endpoints "m
ysqldb" not found' looking up endpoints for 'mysqldb' in namespa
ce 'default', attempting to create them
I1015 22:24:33.910707   15465 register.go:180] No pre existing e
xact matching ports list found, created new subset {[{192.168.56
.112  <nil> nil}] [] [{mysql 3306 }]}
I1015 22:24:33.921195   15465 register.go:191] Successfully upda
ted mysqldb, now with 1 endpoints
```

集群管理

如果你之前在 kubernetes 上运行过 mysql，您需要将 kubernetes 的 mysql service 移除：

```
kubectl delete service mysql
```

执行 istioctl 来配置 service（在您的 admin 机器上）：

```
istioctl register mysql IP mysql:PORT
```

注意：`mysqldb` 虚拟机不需要也不应该有特别的 kubernetes 权限。

使用 mysql 服务

bookinfo 中的评分服务将使用机器上的数据库。要验证它是否有效，请创建使用虚拟机上的 mysql db 的评分服务的 v2 版本。然后指定强制评论服务使用评分 v2 版本的路由规则。

```
# 创建使用 mysql 后端的评分服务版本
istioctl kube-inject -n bookinfo -f samples/bookinfo/kube/bookin
fo-ratings-v2-mysql-vm.yaml | kubectl apply -n bookinfo -f -

# 强制 bookinfo 使用评分后端的路有规则
istioctl create -n bookinfo -f samples/bookinfo/kube/route-rule-
ratings-mysql-vm.yaml
```

您可以验证 bookinfo 应用程序的输出结果，显示来自 Reviewer1 的 1 星级和来自 Reviewer2 的 4 星级，或者更改虚拟机上的评分并查看结果。

同时，您还可以在 RawVM MySQL 的文档中找到一些故障排查和其它信息。

参考文档

暂未翻译，请直接阅读英文原文文档：

https://istio.io/docs/reference/