# 目录

# Istio官方文档中文版

## 介绍

Istio是一个开放平台，提供统一的方式来集成微服务，管理跨微服务的流量，执行策略和汇总遥测数据。Istio的控制面板在底层集群管理平台（如Kubernetes，Mesos等）上提供了一个抽象层。

## 内容

这是Istio官方文档的中文翻译版。

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
- 下载pdf格式
- 下载mobi格式
- 下载epub格式



## 进度

当前已经完成内容：

# 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。这里您可以了解Istio的工作原理以及它是如何实现的。

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

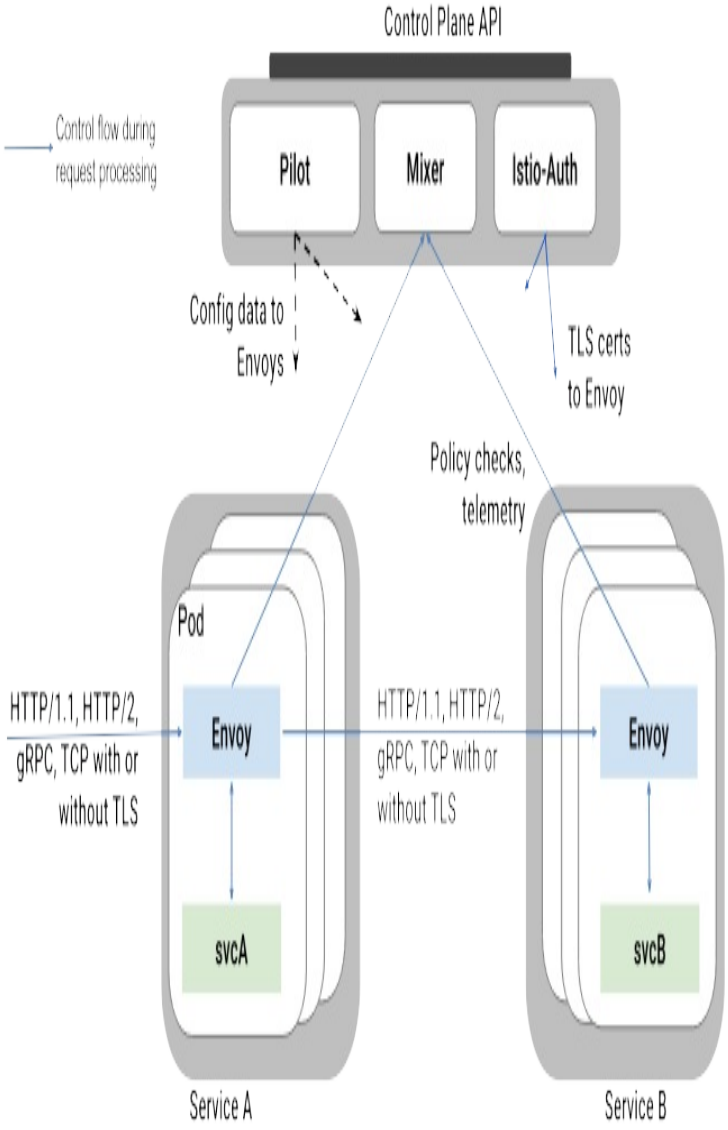
这些功能极大的减少了应用程序代码，底层平台和策略之间的耦合。耦合的减少不仅使服务更容易实现，而且还使运维人员更容易地在环境之间移动应用程序部署，或换用新的策略方案。因此，结果就是应用程序从本质上变得更容易移动。

# 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。
- 探索并尝试部署示例应用程序。
- 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的架构有几个关键设计目标，这些目标对于使系统能够大规模和高性能地处理服务至关重要。

- 最大化透明度

要让Istio被采纳，运维人员或开发人员应该能够只做很少的工作就可以从中受益。为此，Istio将自身自动注入到服务间所有的网络路径中。Istio使用边车代理来捕获流量，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。一旦注入边车代理到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，策略的应用提供了对网格行为的大量控制，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。假设你已经阅读了 Istio是什么？并熟悉Istio的高级架构。您可以在本节其他指南中了解有关单个流量管理功能的更多信息。

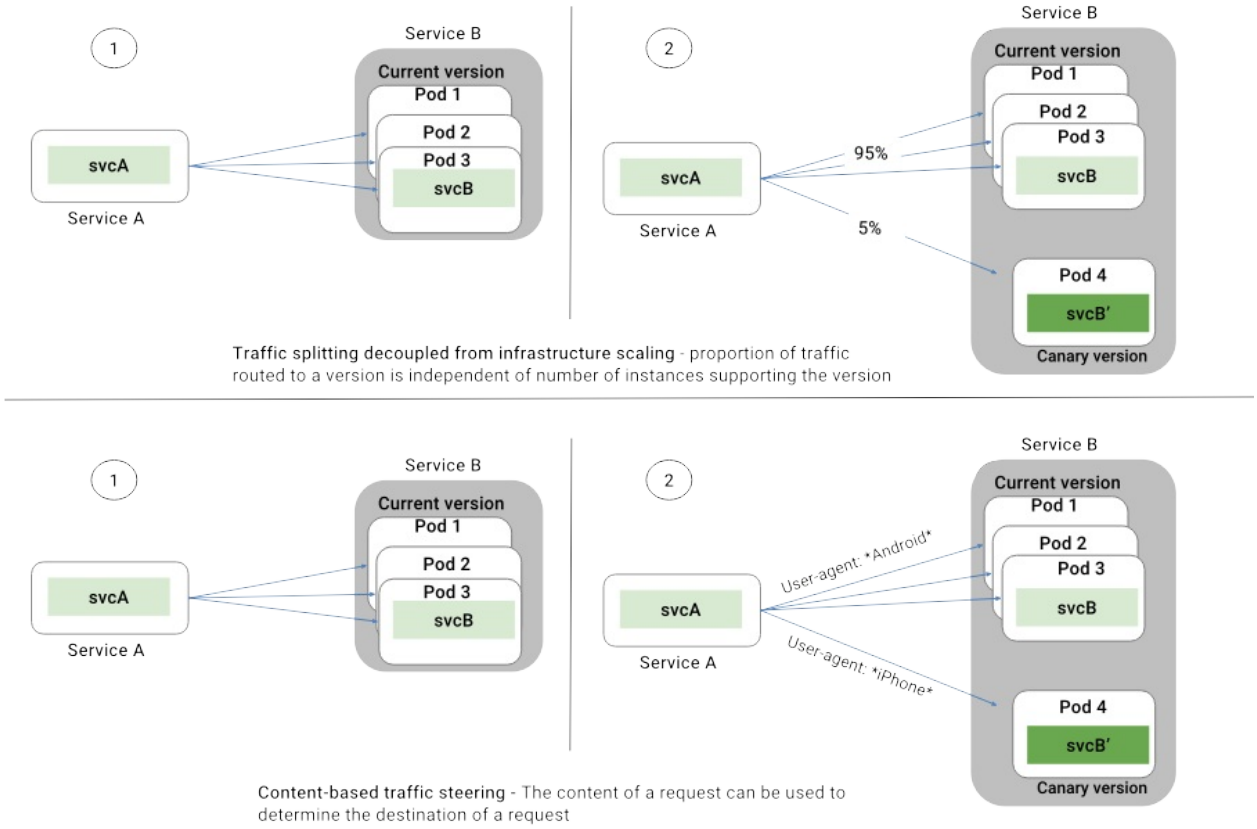
# Pilot和Envoy

用于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。它允许您指定用于在Envoy代理之间路由流量的规则，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，并使用它来通过其发现服务让Envoys了解网格中的其他实例。

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

# 流量管理的好处

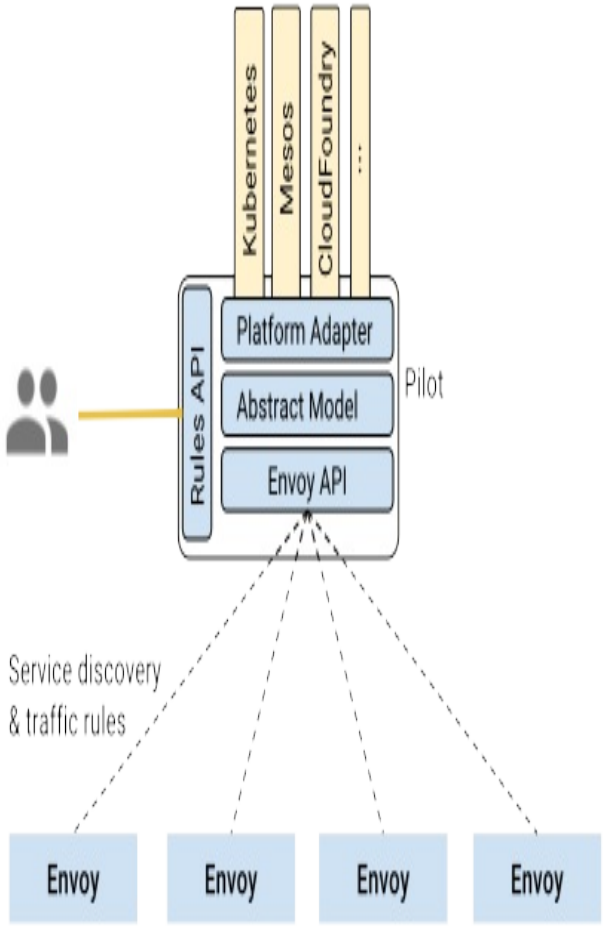
使用Istio的流量管理模型，本质上将流量和基础设施扩展解耦，让运维人员通过Pilot指定他们希望流量遵循什么规则，而不是哪些特定的pod/VM应该接收流量 - Pilot和智能Envoy代理搞定其余的。因此，例如，您可以通过Pilot指定您希望特定服务的5%流量可以转到金丝雀版本，而不考虑金丝雀部署的大小，或根据请求的内容将流量发送到特定版本。




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。




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

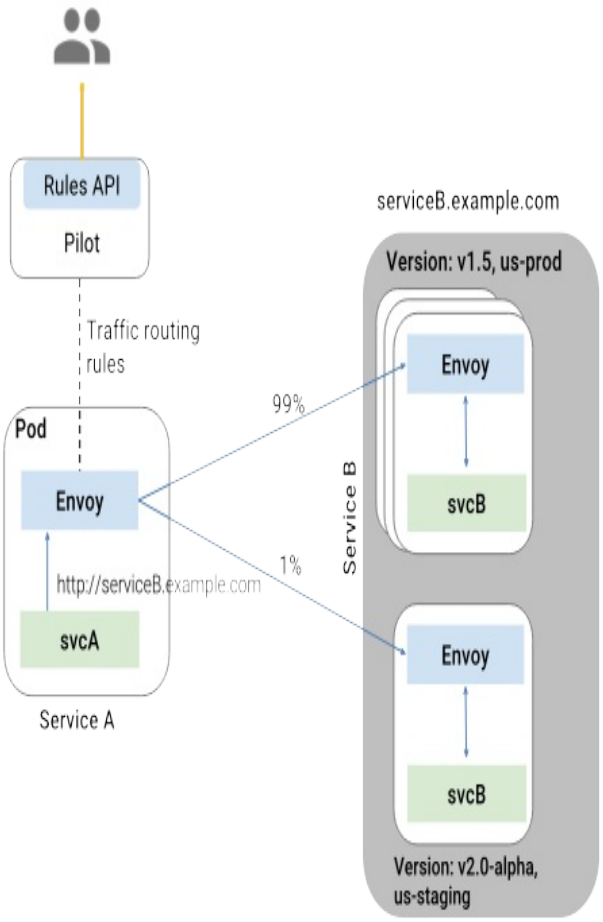
此节描述Istio服务网格中的服务之间如何路由请求。

## 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使用平台中的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之间的所有请求/响应。

Envoy根据运维人员使用Pilot指定的路由规则，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脱离其依赖服务的演进，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。

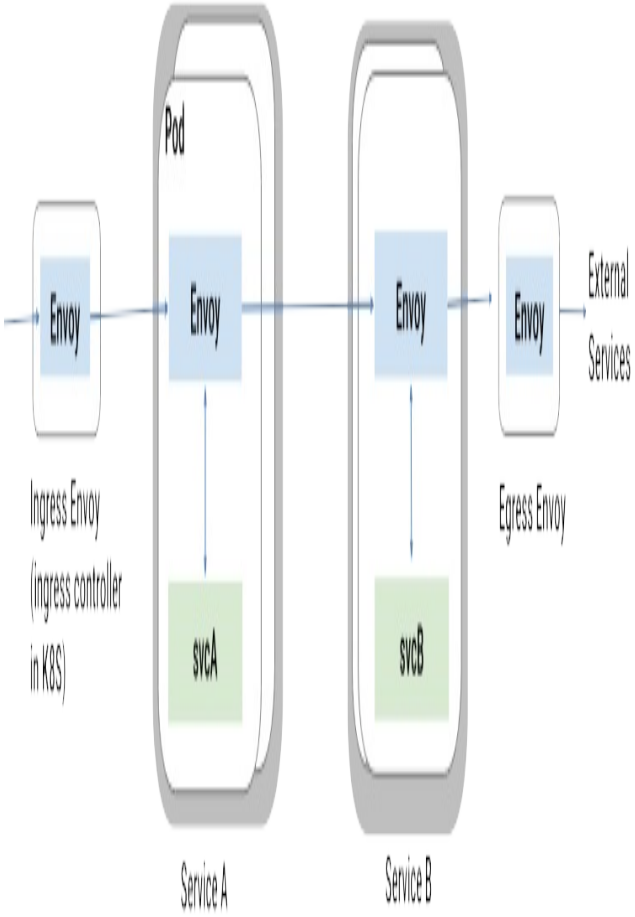
Istio还为同一服务版本的多个实例提供流量负载均衡。您可以在 服务发现和负载均衡 中找到更多信息。

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。

## 入口和出口Envoys

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，例如熔断器，通过Mixer强加限速，并使用Istio-auth提供认证。

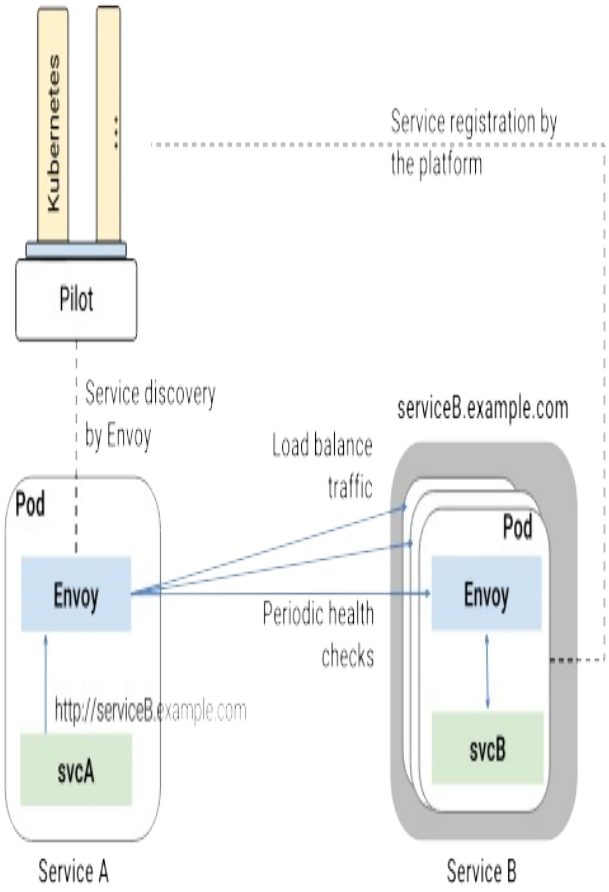


# 发现和负载均衡

本节描述Istio如何在服务网格中的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了大量的解决方案。

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名称彼此访问。绑定到服务的所有HTTP流量都会自动通过Envoy重新路由。Envoy在负载均衡池中的实例之间分发流量。虽然Envoy支持几种 复杂的负载均衡算法，但Istio目前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与和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对延迟的生影响。

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然处理故障？

   是。Istio提高网格中服务的可靠性和可用性。但是，应用程序需要处理故障（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，但仍然必须测试整个应用程序的端到端故障恢复能力。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），以训练应用的弹性。

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提供了一种简单的领域特定语言（DSL）来控制应用程序部署中跨各种服务的API调用和第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。有关详细信息，请参阅 路由规则参考

例如，使用规则DSL来描述，将“reviews”服务的100％的传入流量发送到版本“v1”的简单规则可以使用规则DSL来如下描述：

```
destination: reviews.default.svc.cluster.local
route:
- tags:
    version: v1
  weight: 100
```

destination是要路由流量的服务的名称。在Istio的Kubernetes部署中，路由*tag* "version：v1"对应于Kubernetes *label* "version：v1"。该规则确保只有包含标签"version：v1"Kubernetes pod将会收到流量。可以使用 istioctl CLI 配置规则。有关示例，请参阅 配置请求路由任务。

在Istio中有两种类型的规则，**Routes/**路由 和 **Destination Policies/**目的地策略（这些与Mixer策略不同）。两种类型的规则控制请求如何路由到目标服务。

# 路由

Routes 控制请求如何路由到不同版本的服务。请求可以基于源和目标，HTTP header字段以及与个别服务版本相关联的权重进行路由。编写路由规则时，必须牢记以下重要方面：

## 通过**destination**限定规则

每个规则对应于规则中的 *destination* 字段标识的目的地服务。例如，适用于"reviews"服务调用的所有规则将包括下面字段。

```
destination: reviews.default.svc.cluster.local
```

*destination* 的值应该是一个完全限定域名（Fully Qualified Domain Name,FQDN）。Pilot用它来给服务匹配规则。例如，在Kubernetes中，服务的完全限定域名可以使用以下格式构建：*serviceName.namespace.dnsSuffix*。

## 通过source/headers限定规则

规则可以选择仅限于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。

   ```
   destination: ratings.default.svc.cluster.local
   match:
     source: reviews.default.svc.cluster.local
   ```

   *source* 的值，和 *destination* 一样，必须是一个服务的FQDN。

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。

   ```
   destination: ratings.default.svc.cluster.local
   match:
     source: reviews.default.svc.cluster.local
     sourceTags:
       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"。

   ```
   destination: reviews.default.svc.cluster.local
   match:
     httpHeaders:
       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。

```
destination: ratings.default.svc.cluster.local
match:
  source: reviews.default.svc.cluster.local
  sourceTags:
    version: v2
  httpHeaders:
    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 *tags* 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"。

```
destination: reviews.default.svc.cluster.local
route:
- tags:
    version: v2
  weight: 25
- tags:
    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：

```
destination: "ratings.default.svc.cluster.local"
route:
- tags:
    version: v1
httpReqTimeout:
  simpleTimeout:
    timeout: 10s
```

最大尝试次数，或者在默认或被覆盖的超时时间内的尽可能多，可以设置如下：

```
destination: "ratings.default.svc.cluster.local"
route:
- tags:
    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。

```
destination: reviews.default.svc.cluster.local
route:
- tags:
    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。

```
destination: "ratings.default.svc.cluster.local"
route:
- tags:
    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％：

```
destination: ratings.default.svc.cluster.local
match:
  source: reviews.default.svc.cluster.local
  sourceTags:
    version: v2
route:
- tags:
    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。

## 规则优先级

多个路由规则可以应用于同一目的地。当有多个规则时，可以通过设置规则的*precedence*字段来指定与给定目的地相对应的规则的评估顺序。

```
destination: reviews.default.svc.cluster.local
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。

广义的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，该规则通过source/header到特定destination来限定规则，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，以提供所有其他情况的流量。

例如，以下2条规则一起指定包含名为"Foo"值为"bar"的header的"reviews"服务的所有请求都将发送到"v2"实例。所有剩余的请求将被发送到"v1"。

```
destination: reviews.default.svc.cluster.local
precedence: 2
match:
  httpHeaders:
    Foo:
      exact: bar
route:
- tags:
    version: v2
---
destination: reviews.default.svc.cluster.local
precedence: 1
route:
- tags:
    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限定，例如正在调用的服务或HTTP请求header。

但是，可以限制策略适用于使用特定标签路由到后端的请求。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。

```
destination: reviews.default.svc.cluster.local
policy:
- tags:
    version: v1
  loadBalancing:
    name: RANDOM
```

## 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。

```
destination: reviews.default.svc.cluster.local
policy:
- tags:
    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。

```
destination: reviews.default.svc.cluster.local
policy:
- tags:
    version: v1
  circuitBreaker:
    simpleCb:
      maxConnections: 100
```

由于没有为"reviews"服务定义特定的路由规则，因此默认轮循路由行为将适用，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。

```
destination: reviews.default.svc.cluster.local
route:
- tags:
    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。

# 网络和认证

介绍核心网络和认证功能。

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。

# 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：

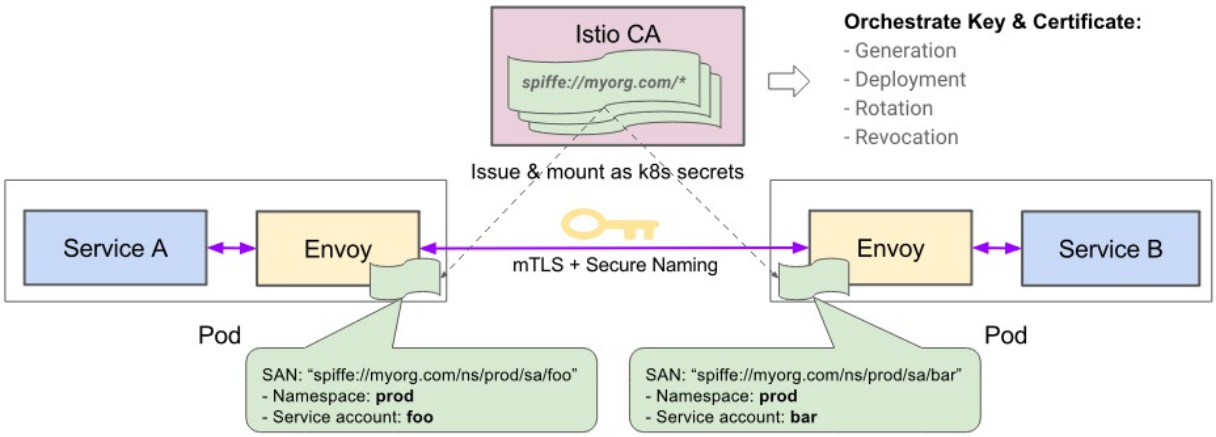
- 为每个服务提供强大的身份，代表其角色，以实现跨集群和云的互通性
- 加密服务到服务的通讯
- 提供密钥管理系统来自动执行密钥和证书的生成，分发，轮换和撤销

在将来的版本中，它还将提供：

- 加密终端用户到服务的通信
- 细粒度的授权和审核,来控制和监控访问您服务，api或资源的人员
- 多重授权机制：ABAC, RBAC, 授权钩子

## 架构

下图展示 Istio Auth 架构，其中包括三个组件：身份，密钥管理和通信安全。它描述了Istio Auth如何用于加密服务A(作为服务帐户“foo”运行)和服务B(作为服务帐户“bar”运行)之间的服务到服务通信。



# 组件

## 身分

在 Kubernetes 上运行时，由于以下原因，Istio Auth使用 Kubernetes服务帐户 来识别运行该服务的人员：

- 服务帐户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，她将能够提取一组和Bob运行这个工作负载不同的数据。
- 服务帐户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服务帐户运行）来实现强大的安全策略。
- 工作负载运行的服务帐户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服务帐户的唯一性

## 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。端到端通信通过一下方式加密：

- 服务与Envoy之间的本地TCP连接
- 代理之间的相互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 Auth 提供每群集CA（证书颁发机构）来自动化密钥和证书管理。它执行四个关键操作：

- 为每个服务帐户生成一个 SPIFFE 密钥和证书对
- 根据服务帐户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

# 工作流

Istio Auth工作流由两个阶段组成，部署和运行时。这部分涵盖他们两个。

## 部署阶段

1. Istio CA 观察 Kubernetes API Server，为每个现有和新的服务帐户创建一个 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会根据服务帐户使用 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。

## 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

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

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

# 属性

本节描述Istio属性，它们是什么以及如何使用它们。

## 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。属性是描述入口和出口流量的有名称和类型的元数据片段，以及此流量发生的环境。Istio属性携带特定信息片段，例如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

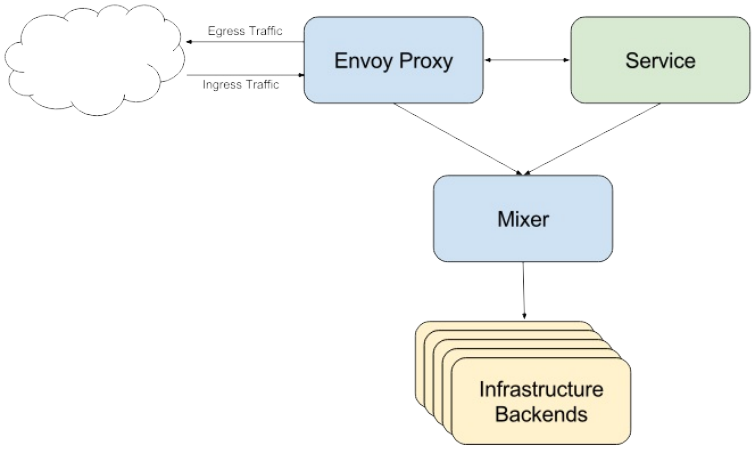
本节解释 Mixer 的角色和总体架构。

## 背景

基础设施后端设计用于提供用于构建服务的支持功能。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。服务传统上直接与这些后端系统集成，创建一个硬耦合和炙热的(baking-in)特定语义和使用选项。

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从应用层移出并用配置替代，在运维人员控制下。应用程序代码不再将应用程序代码与特定后端集成在一起，而是与Mixer进行相当简单的集成，然后 Mixer 负责与后端系统连接。

混音器不是为了在基础设施后端之上创建 可移植性层。这不是要试图定义一个通用的日志记录API，通用metric API，通用计费API等等。相反，Mixer旨在改变层之间的界限，以减少系统复杂性，从服务代码中消除策略逻辑，并替代为让运维人员控制。




Mixer 提供三个核心功能：

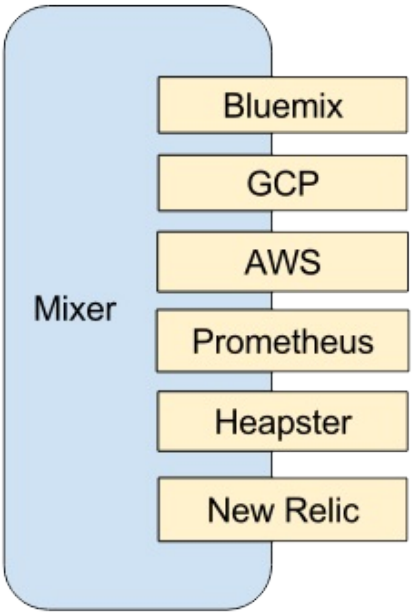
- 前提条件检查。允许服务在响应来自服务消费者的传入请求之前验证一些前提条件。前提条件可以包括服务使用者是否被正确认证，是否在服务的白名单上，是否通过ACL检查等等。
- 配额管理。使服务能够在多个维度上分配和释放配额，配额被用作相对简单的资源管理工具，以便在争取有限的资源时在服务消费者之间提供一些公平性。限速是配额的例子。
- 遥测报告。使服务能够上报日志和监控。在未来，它还将启用针对服务运营商以及服务消费者的跟踪和计费流。

这些机制的应用是基于一组 属性的,这些属性为每个请求物化到 Mixer 中。在Istio内，Envoy重度依赖Mixer。在网格内运行的服务也可以使用Mixer上报遥测或管理配额。（注意：从Istio pre0.2起，只有Envoy可以调用Mixer。）

## 适配器

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

Mixer在处理不同基础设施后端的灵活性是通过使用通用插件模型实现的。单个的插件被称为适配器，它们允许 Mixer 与不同的基础设施后端连接，这些后台可提供核心功能，例如日志，监控，配额，ACL检查等。适配器使Mixer能够暴露一个一致的API，与使用的后端无关。在运行时使用的确切的适配器套件是通过配置确定的，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` , `Report` ,和 `Quota` ）都接受来自输入的一组属性，并在输出上产生一组属性。单个方法执行的工作由输入属性集以及Mixer的当前配置决定。为此，服务运营商负责：

- 配置部署使用的一组*aspects*/切面。切面本质上是配置状态的一个存储块，它配置一个适配器（适配器是如上面所描述的二进制插件）。
- 建立Mixer可以操作的适配器参数类型。这些类型在配置中通过一组描述符 （如这里所描述的）描述

- 创建规则将每个传入请求的属性映射到特定的一组切面和适配器参数。

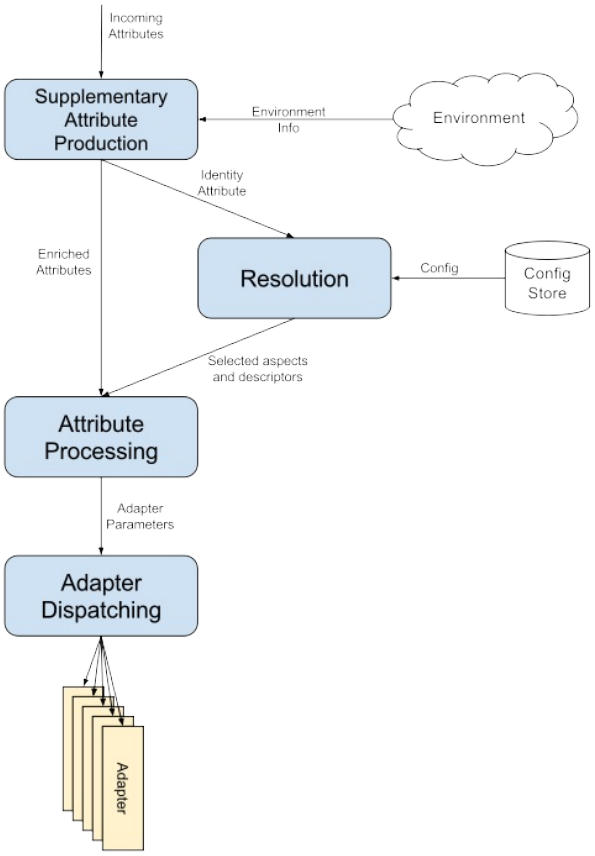
需要上述配置状态才能让 Mixer 知道如何处理传入的属性并分发到适当的基础设置后端。

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：

- 补充属性生产。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 了解解决方案的工作原理。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。属性处理通过简单声明的方式进行初始配置,如 这里 描述的。
- 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。



# 脚本

> **info::注意**
>
> 本节是初步的，可能会改变。我们仍在Mixer中实验脚本的概念。

Mixer 的属性处理阶段通过脚本语言（确切语言还未定）来实现。脚本提供了一组属性，并负责生成适配器参数和到各个已配置适配器的调度控制。

对于常见用途，运维人员通过相对简单的声明格式和表达式语法来生成适配器参数生产规则。Mixer摄取此类规则并生成脚本，该脚本执行必要的运行时工作,以访问请求的传入属性并生成必需的适配器参数。

对于高级用途，运维人员可以绕过声明格式，直接以脚本语言编写。这更复杂，但提供极大的灵活性。

# Mixer 配置

本节介绍 Mixer 的配置模型。

## 背景

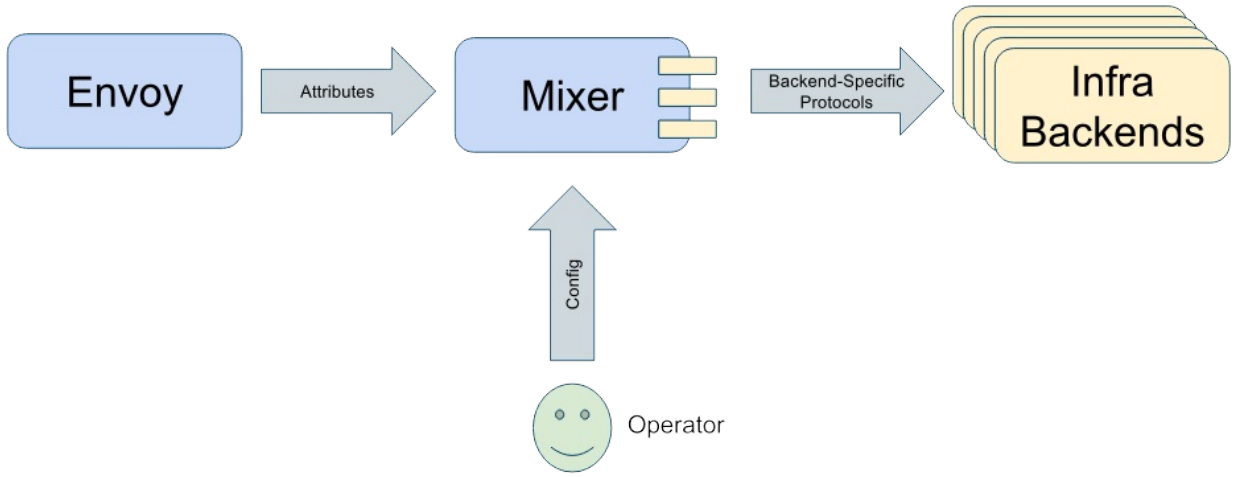
Istio是一个具有数百个独立功能的复杂系统。Istio部署可能涉及数十个服务的蔓延事件，这些服务有一群Envoy代理和Mixer实例来支持它们。在大型部署中，许多不同的运维人员（每个运维人员都有不同的范围和责任范围）可能涉及管理整体部署。

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记录来控制Mixer部署中的所有操作和策略切面。
- 范围。配置被分层描述，可以实现粗略的全局控制以及细粒度的本地控制。
- 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，以帮助减少导致服务中断的错误配置更改的可能性。
- 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。该属性集确定Mixer为给定的请求调用哪个后端以及每个给出哪些参数。为了隐藏各个后端的细节，Mixer使用称为适配器的模块。




Mixer的配置有两个中心职责：

- 描述哪些适配器正在使用以及它们的运行方式。
- 描述如何将请求属性映射到适配器参数中。

配置使用YAML格式来表示,围绕五个核心抽象构建：

| 概念 | 描述 |
| --- | --- |
| 适配器 | 适用于各个Mixer适配器的低级别的关注于操作的配置。 |
| 切面 | 适用于各个Mixer适配器的高级别的关注于意图的配置。 |
| 描述符 | 用于各个切面的参数描述。 |
| 范围 | 根据请求的属性选择要使用哪些切面和描述符的机制。 |
| 清单 | Istio部署的诸多静态特性的描述。 |

以下部分将详细介绍这些概念。

## 适配器

适配器是基础工作单位,Istio Mix围绕它而构建。适配器封装了将Mixer与特定的外部基础设施后端（如Prometheus，New Relic或Stackdriver）接入所必需的逻辑。单个适配器通常需要提供一些基本的操作参数才能完成他们的工作。例如，日志适配器可能需要知道应该将日志数据吐到哪个IP地址和端口。

Mixer可以使用一套适配器，每个都需要单独的配置参数。以下是一个示例，说明如何配置适配器：

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

该 `name` 字段为适配器配置块提供了一个名称，因此可以从别处引用它。该 `kind` 字段指示此配置适用的切面类型。该 `impl` 字段给出正在配置的适配器的名称。最后，该 `params` 部分是指定实际的适配器特定配置参数的位置。在这个案例中，这里配置的是适配器在其查询中应使用的URL，并定义刷新其本地缓存的时间间隔。

对于每个可用的适配器实现，您可以定义任意数量的独立配置块。这允许在单个部署中多次使用相同的适配器。根据具体情况，例如涉及哪个服务，将使用某一个配置块而不是其他。例如，这里有两个可以与前一个共存的配置块：

```
adapters:
  - name: mySecondaryListChecker
    kind: lists
    impl: ipListChecker
    params:
      publisherUrl: https://mysecondlistserver:912
      refreshInterval: 3600s
  - name: myTernaryListChecker
    kind: lists
    impl: genericListChecker
    params:
      listEntries:
        "400"
        "401"
        "402"
```

还有一个：

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

这将配置适配器,将数据报告给Prometheus系统。此适配器不需要任何自定义参数，因此没有 `params` 节。

每个适配器定义其自己的特定格式的配置数据。适配器的详尽集及其特定的配置格式可以在这里找到。

## 切面

切面定义高级别配置（有时称为基于意图的配置），独立于特定适配器类型的特定实现细节。而适配器专注于如何**(how)**做某些事情，切面侧重于要做什么**(what)**。

我们来看一个切面的定义：

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

该 `kind` 字段区分定义的切面的行为。支持的切面如下表所示。

| 类型 | 描述 |
| --- | --- |
| quotas | 执行配额和限速。 |
| metrics | 生成metric |
| lists | 执行基于白名单或黑名单的访问控制。 |
| access-logs | 为每个请求生成固定格式的访问日志。 |
| application-logs | 为每个请求生成灵活的应用程序日志。 |
| attributes | 为每个请求生成补充属性。 |
| denials | 按步就班地产生可预测的错误代码。 |

在上面的示例中，切面声明指定了 `lists` 类型,表示我们正在配置一个切面，其目的是使用白名单或黑名单作为访问控制的一种简单形式。

该 `adapter` 字段指示与此切面关联的适配器配置块。切面总是以这种方式与特定适配器相关联，因为适配器负责实际执行由切面配置表示的工作。在这种具体案例中，所选择的特定适配器确定要使用的名单，以执行方面的名单检查功能。

通过将切面配置与适配器配置分开，可以轻松地更改用于实现特定切面行为的适配器，而无需更改切面本身。另外，许多切面都可以引用相同的适配器配置。

`params` 节是您输入特定种类的配置参数的地方。在这个 `lists` 类型的案例中，配置参数指定名单是否是黑名单（名单中的条目导致拒绝），而不是白名单（不在列表中的条目导致拒绝）。`checkExpression` 字段指示在请求时使用的属性, 用来获取符号以相关适配器的名单进行检查.

这是另一个切面，这次是一个 `metrics` 切面：

```
aspects:
- kind: metrics
  adapter: myMetricsCollector
  params:
    metrics:
    - descriptorName: request_count
      value: "1"
      labels:
        source: source.name
        target: target.name
        service: api.name
        responseCode: response.code
```

这定义了一个切面，它产生了metrics, metrics被发送到前面定义的myMetricsCollector适配器。该 `metrics` 节定义了在请求处理期间为这个切面生成的metrics集合。该 `descriptorName` 字段指定描述符的名称，该描述符是单独的配置块，如下所述，声明了这种metrics的类型。该 `value` 字段和四个标签字段描述哪些属性在请求时使用以产生metrics。

每个切面类型都定义了自己的配置数据格式。这里 可以找到详尽的切面配置格式。

## 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适配器参数。该阶段通过评估一系列属性表达式来运行。

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：

```
  source: source.name
  target: target.name
  service: api.name
  responseCode: response.code
```

冒号右侧的序列是属性表达式的最简单形式。它们只包括属性名称。在上面，`source` 标签将被赋予 `source.name` 属性的值。以下是条件表达式的示例：

```
  service: api.name | target.name
```

使用上述方法，服务标签将被赋予 `api.name` 属性的值，或者如果没有定义该属性，它将被赋予 `target.name` 属性的值。

可以在属性表达式中使用的属性必须在部署的 属性清单 中定义。在清单中，每个属性都有类型,表示该属性所携带的数据类型。同样，属性表达式也有类型，它们的类型是从表达式中的属性和应用于这些属性的运算符派生出来的。

属性表达式的类型用于确保在什么情况下使用哪些属性的一致性。例如，如果metric描述符指定的特定标签类型为 INT64，则只能使用产生64位整数的属性表达式来填充该标签。上述 `responseCode` 标签就是这种情况。

有关详细信息，请参阅 属性表达式引用。

## 选择器

选择器是应用于某切面的注释，以确定该切面是否适用于任何给定的请求。选择器使用产生布尔值的属性表达式。如果表达式返回 `true` 则相关切面将被应用。否则会被忽略，没有任何效果。

让我们添加一个选择器到上一个切面例子：

```
aspects:
- selector: target.service == "MyService"
  kind: metrics
  adapter: myMetricsCollector
  params:
    metrics:
    - descriptorName: request_count
      value: "1"
      labels:
        source: source.name
        target: target.name
        service: api.name
        responseCode: response.code
```

`selector` 上面的字段定义了一个表达式，如果 `target.service` 属性等于"MyService"就返回 `true` 。如果表达式返回 `true` ，则切面定义对给定的请求生效，否则就像切面未定义一样。

## 描述符

描述符用于准备Mixer，其适配器及其基础设施后端以接收特定类型的数据。例如，声明一组metrics描述符告诉Mixer不同metrics将携带的数据类型，以及用于标识这些度量的不同实例的标签集。

有不同类型的描述符，每种都与特定切面类型相关联：

| 描述符类型 | 切面类型 | 描述 |
| --- | --- | --- |
| Metric Descriptor | metrics | 描述单个metric是什么样的。 |
| Log Entry Descriptor | application-logs | 描述单个日志条目是什么样的。 |
| Quota Descriptor | quotas | 描述单个配额是什么样的。 |

这是一个示例metric描述符：

```
metrics:
  - name: request_count
    kind: COUNTER
    value: INT64
    displayName: "Request Count"
    description: Request count by source, target, service, and code
    labels:
      source: STRING
      target: STRING
      service: STRING
      responseCode: INT64
```

以上是声明系统可以产生名为 `request_count` 的metrics。这样的metrics将持有64位整数值并作为绝对计数器进行管理。报告的每个metric将有四个标签，两个指定源和目标名称，一个是服务名称，另一个是请求的响应代码。对于此描述符，Mixer可以确保生成的度量标准总是正确组成的，可以安排这些metrics的高效存储，并且可以确保基础设施后端可以接受这些metrics。这些 `displayName` 和 `description` 字段是可选的，并且传达给基础设施后端，后端可以使用这些文本来增强其metric可视化界面。

明确定义描述符并使用它们来创建适配器参数类似于传统编程语言中的类型和对象。这样做可以实现以下重要场景：

- 明确定义描述符集后，Istio可以对基础设置后端进行编程，以接受Mixer生成的流量。例如，metric描述符提供所需的所有信息，来编程基础设置后端以接受符合描述符形状（它的值类型及其标签集）。
- 描述符可以被多个切面引用和重用。
- 它使Istio能够提供强类型的脚本环境，如 这里 所述

不同的描述符类型的细节在 这里。

## 范围

Istio部署可以负责管理大量服务。组织通常有数十种或数百种交互式服务，而Istio的使命是使其易于管理所有服务。Mixer的配置模式旨在支持不同运维人员管理Istio部署的不同部分，而不会踩在彼此的脚趾上，同时允许他们对其区域进行控制，但不允许其他人操作。

这一切是如何工作：

- 上一节（适配器，切面和描述符）中描述的各种配置块始终在层次结构的上下文中定义。

- 层次结构由DNS风格的点号分割的名称表示。像DNS一样，层次结构从点号分割的名称中最右边的元素开始。
- 每个配置块与范围和主题相关联，这两个对象都是点号分割的名称,表示层次结构中的位置：
    - 范围代表创建配置块的权限。层次结构中的高级别的权力比较低级别的权力更强大。
    - 主题表示层次结构中状态块的位置。在层次结构内主题必须始终处于或低于范围的级别。
- 如果配置的多个块具有相同的主题，则与层次结构中最高范围相关联的块始终优先。

构成层次结构的各个元素取决于Istio部署的具体细节。Kubernetes部署可能使用Kubernetes命名空间作为部署Istio配置状态的层次结构。例如，一个有效的范围可能是 `svc.cluster.local` 而一个主题可能是 `myservice.ns.svc.cluster.local`

范围模型旨在与访问控制模型进行配对，以限制允许哪个人容许为特定范围创建配置块。具有在层次结构更高的范围内创建块的权限的操作符可能会影响与较低范围相关联的所有配置。尽管这是设计意图，但是Mixer配置还不支持配置的访问控制，因此对于哪个操作员可以操作哪个范围没有实际的约束。

## 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 示例 找到完整的Mixer配置示例。作为具体示例，这里是 默认配置。

# Mixer Aspect 配置

说明如何配置 Mixer 切面 及其依赖项。

## 概述

Mixer配置通过指定三个关键信息来表达系统行为：采取什么行动，如何采取行动以及何时采取行动。

- 采取什么行动：*Aspect* 配置定义要采取什么行动。这些行动包括日志，metrics收集，名单检查，配额执行等。描述符 是切面配置的命名和可重用的部分。例如，`metrics` 切面定义MetricDescriptor，并按名称引用MetricDescriptor实例。
- 如何采取行动：适配器配置定义如何采取行动。metrics适配器配置包括基础设施后端的详细信息。
- 何时采取行动：选择器和`subjects`定义何时采取行动。选择器是基于属性的表达式，如`response.code == 200`，Subject是分层资源名称，如`myservice.namespace.svc.cluster.local`。

## 配置步骤

请考虑以下启用限速的切面配置。

```
- aspects:
  - kind: quotas
    params:
      quotas:
      - descriptorName: RequestCount
        maxAmount: 5
        expiration: 1s
        labels:
          label1: target.service
```

它使用`RequestCount`来描述配额。以下是`RequestCount`描述符的例子。

```
name: RequestCount
rate_limit: true
labels:
   label1: 1 # STRING
```

在此示例中，`rate_limit` 为 `true`，因此该 `aspect` 必须指定 `expiration`。类似地，该 `aspect` 必须提供一个 `string` 类型的标签。

Mixer 将使用限速的工作代理给实现 `quotas` 类型的 `adapter`。adapters.yml 定义这个配置。

```
- name: default
  kind: quotas
  impl: memQuota
  params:
    minDeduplicationDuration: 2s
```

上述示例中的 `memQuota` 适配器需要一个参数。运维人员可以通过指定备用 `quotas` 适配器从 `memQuota` 切换到 `redisQuota`。

```
- name: default
  kind: quotas
  impl: redisQuota
  params:
    redisServerUrl: redisHost:6379
    minDeduplicationDuration: 2s
```

以下示例显示了如何使用 选择器 选择性地应用限速。

```
- selector: source.labels["app"]=="reviews" && source.labels["version"] == "v3"
  aspects:
  - kind: quotas
    params:
      quotas:
      - descriptorName: RequestCount
        maxAmount: 5
        expiration: 1s
        labels:
          label1: target.service
```

# 切面组合

上一节中概述的步骤适用于Mixer的所有切面。每个切面都需要特定的 `描述符` 和 `适配器`。下表列举了 `切面`，`描述符` 和 `适配器` 的有效组合。

| 切面 | 描述符 | 适配器 |
| --- | --- | --- |
| Quota enforcement | QuotaDescriptor | memQuota, redisQuota |
| Metrics collection | MetricDescriptor | prometheus,statsd |
| Whitelist/Blacklist | None | genericListChecker,ipListChecker |
| Access logs | LogEntryDescriptor | stdioLogger |
| Application logs | LogEntryDescriptor | stdioLogger |
| Deny Request | None | denyChecker |

Istio使用 `protobufs` 来定义配置模式。 编写配置 文档解释了如何将 `proto` 定义表达为 `yaml` 。

# 配置的组织

切面配置适用到 `subject` 。 `subject` 是层次结构中的资源。 通常 `subject` 是服务，命名空间或集群的完全限定名称。切面配置可以应用于 `subject` 资源及其子资源。

# 推送配置

`istioctl` 将配置更改推送到API服务器。从Alpha版本开始，API服务器支持仅推送切面规则。

临时解决方法允许您按如下所示推送 `adapters.yml` 和 `descriptors.yml` 。

1. 找到 Mixer pod

   ```
   kubectl get pods -l istio=mixer
   ```

   输出类似于：

   ```
   NAME                            READY     STATUS    RESTARTS   AGE
   istio-mixer-2657627433-3r0nn   1/1       Running   0          2d
   ```

2. 从 Mixer 中获取adapters.yml

   ```
   kubectl cp istio-mixer-2657627433-3r0nn:/etc/opt/mixer/configroot/scopes/global/ad
   apters.yml  adapters.yml
   ```

3. 编辑文件并推送回去

```
kubectl cp adapters.yml istio-mixer-2657627433-3r0nn:/etc/opt/mixer/configroot/sco
pes/global/adapters.yml
```

4. 同样更新 `/etc/opt/mixer/configroot/scopes/global/descriptors.yml`
5. 查看Mixer日志以检查验证错误，因为上述操作绕过了API服务器。

# 默认配置

Mixer 为常用的 描述符 和 适配器 提供默认定义。

# 下一步

- 了解有关 Mixer 和 Mixer 配置 的更多信息。
- 发现完整的 属性词汇 。

# 任务

向您展示如何使用Istio执行单个定向活动。

- 安装Istio：此任务展示如何搭建Istio服务网格。
- 将服务集成到网格中：此任务展示如何将应用程序与Istio服务网格集成。
- 启用入口流量：介绍如何配置Istio来暴露服务到服务网格外。
- 启用出口流量：介绍如何配置Istio来将网格中的服务流量路由到外部服务。
- 配置请求路由：此任务展示如何根据权重和HTTP header配置动态请求路由。
- 故障注入：此任务展示如何注入延迟并测试应用程序的弹性。
- 设置请求超时：此任务展示如何使用Istio在Envoy中设置请求超时。
- 启用限速：此任务展示如何使用Istio动态地限制流量到服务。
- 启用简单的访问控制：此任务展示如何使用Istio控制对服务的访问。
- 测试Istio Auth：此任务展示如何验证和测试Istio-Auth。
- 收集指标和日志：此任务展示如何配置Mixer从Envoy实例收集指标和日志。
- 分布式请求跟踪：如何配置代理向Zipkin发送跟踪请求

# 安装Istio

这个任务展示如何在Kubernetes集群中安装和配置Istio。

## 前提条件

- 以下说明假设您已经可以访问Kubernetes集群。要在本地安装Kubernetes，请尝试 minikube。
- 如果您正在使用 Google Container Engine，请找到您的集群名称和区域，并为kubectl提取凭据：

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

- 如果您正在使用 IBM Bluemix Container Service，请找到您的集群名称，并为kubectl提取凭据：

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

- 安装Kubernetes客户端kubectl，或升级到集群支持的最新版本。
- 如果您以前在此集群上安装过Istio，请先按照本节末尾的 卸载 步骤进行卸载。

## 安装步骤

您可以使用 Istio Helm chart 进行安装，或按照以下步骤。

对于 pre0.2 版本，Istio必须安装在与应用程序相同的Kubernetes命名空间中。以下说明将在 default 命名空间中部署Istio。也可以修改为部署在不同的命名空间中。

1. 转到 Istio release 页面，下载与您的操作系统对应的安装文件或运行 `curl -L https://git.io/getIstio | sh -` 自动下载并提取最新版本（在MacOS和Ubuntu上）。
2. 解压缩安装文件，并将目录更改为文件解压缩的位置。以下说明与这个安装目录相关。

   安装目录包含：

   - Kubernetes的yaml安装文件
   - 示例应用程序

- `istioctl` 客户端二进制文件，需要注入Envoy为sidecar代理，以及用于创建路由规则和策略。
- istio.VERSION配置文件。

3. 如果从 Istio release 下载安装文件，请将 `istioctl` 客户端添加到 PATH 中。例如，在Linux或MacOS系统上运行以下命令：

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

4. 运行以下命令以确定您的集群是否启用了 RBAC (Role-Based Access Control／基于角色的访问控制):

```
kubectl api-versions | grep rbac
```

- 如果命令显示错误,或不显示任何内容，这意味找集群不支持RBAC，您可以继续执行下面的步骤5。
- 如果命令显示'beta'版本或'alpha'和'beta'两者都有，请使用 istio-rbac-beta.yaml 配置，如下所示：

*(注意：如果是在 `default` 命名空间之外的其他命名空间中部署Istio，请将所有ClusterRoleBinding资源中的 `namespace: default` 行替换为实际的命名空间。)*

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

如果你收到错误:

```
 Error from server (Forbidden): error when creating "install/kubernetes/istio-rbac
-beta.yaml": clusterroles.rbac.authorization.k8s.io "istio-pilot" is forbidden: at
tempt to grant extra privileges: [{[*] [istio.io] [istioconfigs] [] []} {[*] [isti
o.io] [istioconfigs.istio.io] [] []} {[*] [extensions] [thirdpartyresources] [] []
} {[*] [extensions] [thirdpartyresources.extensions] [] []} {[*] [extensions] [ing
resses] [] []} {[*] [] [configmaps] [] []} {[*] [] [endpoints] [] []} {[*] [] [pod
s] [] []} {[*] [] [services] [] []}] user=&{user@example.org [...]
```

您需要添加以下内容：（名称用您自己的替换）

```
 kubectl create clusterrolebinding myname-cluster-admin-binding --clusterrole=clus
ter-admin --user=myname@example.org
```

- 如果命令仅显示'alpha'版本，请使用 istio-rbac-alpha.yaml 配置：

*(注意：如果是在 `default` 命名空间之外的其他命名空间中部署Istio，请将所有ClusterRoleBinding资源中的 `namespace: default` 行替换为实际的命名空间。)*

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

5. 安装Istio的核心组件。

   在这个阶段有两个相互排斥的选择：

   - 安装Istio而不启用 Istio Auth 功能:

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

     此命令将安装Pilot，Mixer，Ingress-Controller，Egress-Controller核心组件。

   - 安装Istio并启用 Istio Auth （在命名空间中部署CA并启用服务之间的mTLS）功能:

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

     该命令将安装Pilot，Mixer，Ingress-Controller和Egress-Controller，以及Istio CA（证书颁发机构）。

6. 可选: 按照以下各节所述安装用于metrics收集和/或请求追踪的插件。

## 启用metrics收集

要收集和查看Mixer提供的metrics，请安装 Prometheus，以及 Grafana 和/或ServiceGraph插件。

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

您可以在 收集指标和日志 中找到更多关于如何使用这些工具的信息。

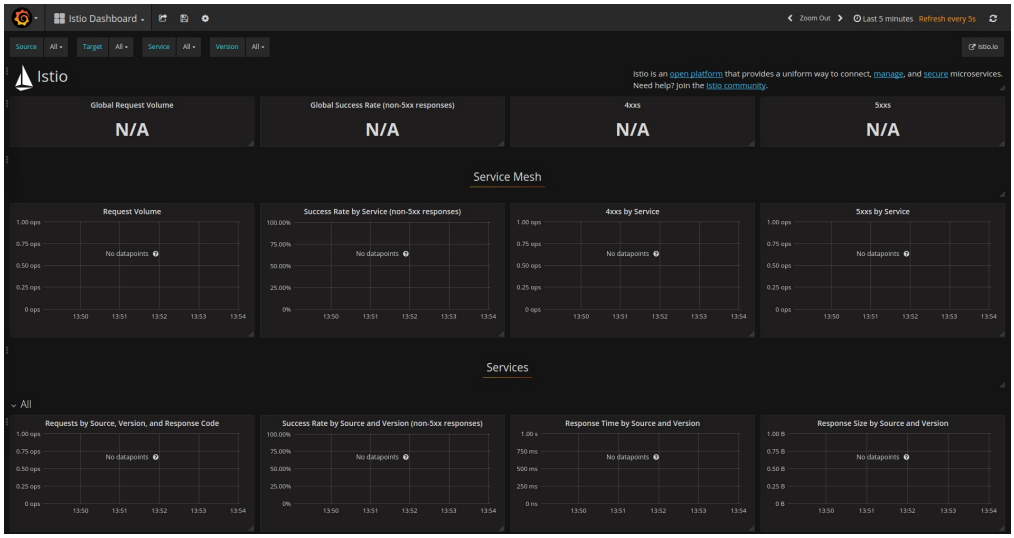
## 验证Grafana仪表盘

Grafana插件提供了Istio 仪表盘可视化的集群中的metrics（请求率，成功/失败率）。如果您安装了Grafana，请检查您是否可以访问仪表盘。

配置 `grafana` 服务的端口转发，具体如下：

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

然后将您的Web浏览器指向 http://localhost:3000/dashboard/db/istio-dashboard. 仪表盘看上去应该是这样的：



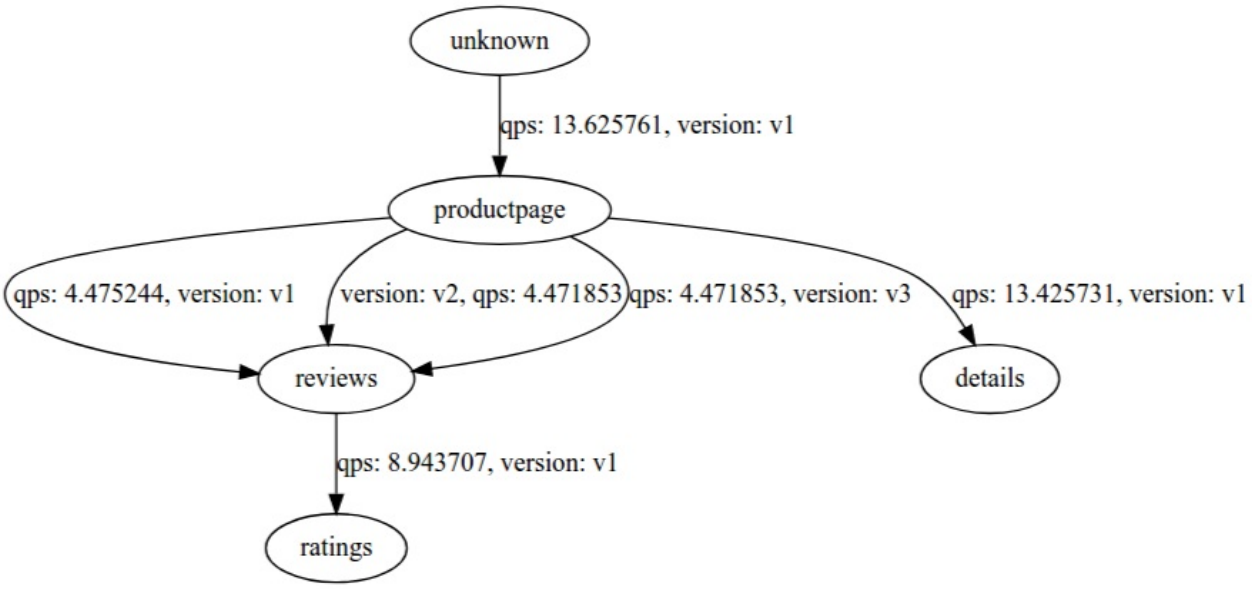

## 验证ServiceGraph服务

ServiceGraph插件为集群提供服务交互图的文本（JSON）表示和图形可视化。像Grafana一样，您可以使用端口转发，service nodePort或（如果外部负载均衡器可用）外部IP来访问servicegraph 服务。在这种情况下，服务名称是 `servicegraph` 和要访问的端口是8088：

```
kubectl port-forward $(kubectl get pod -l app=servicegraph -o jsonpath='{.items[0].metadata.name}') 8088:8088 &
```

ServiceGraph服务提供底层服务图的文本（JSON）表示（通过 `/graph` ）和图形可视化（ `/dotviz` ）。要查看图形可视化（假设您已按照上一个片段配置了端口转发），请打开您的浏览器，网址为：http://localhost:8088/dotviz。

运行某些服务后，例如，在安装 BookInfo 示例应用程序并在应用程序上生成一些负载（例如，在 `while` 循环中执行 `curl` 请求）后，生成的服务图应该看上去像这样：




## 启用与Zipkin的请求跟踪

要启用和查看分布式请求跟踪，请安装Zipkin插件：

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

Zipkin可用于分析Istio应用程序的请求流程和时间，并帮助识别瓶颈。您可以在 分布式请求跟踪 中找到有关如何访问Zipkin仪表盘并使用Zipkin的更多信息。

# 验证安装

1. 确认以下 Kubernetes 服务已经部署: "istio-pilot", "istio-mixer", "istio-ingress", "istio-egress", "istio-ca" (如果启用了Istio Auth), 和, 可选的, "grafana", "prometheus', "servicegraph" and "zipkin".

   ```
   kubectl get svc
   ```

```
 NAME            CLUSTER-IP      EXTERNAL-IP       PORT(S)                       A
GE
 grafana         10.83.252.16    <none>            3000:30432/TCP                5
h
 istio-egress    10.83.247.89    <none>            80/TCP                        5
h
 istio-ingress   10.83.245.171   35.184.245.62     80:32730/TCP,443:30574/TCP    5
h
 istio-pilot     10.83.251.173   <none>            8080/TCP,8081/TCP             5
h
 istio-mixer     10.83.244.253   <none>            9091/TCP,9094/TCP,42422/TCP   5
h
 kubernetes      10.83.240.1     <none>            443/TCP                       3
6d
 prometheus      10.83.247.221   <none>            9090:30398/TCP                5
h
 servicegraph    10.83.242.48    <none>            8088:31928/TCP                5
h
 zipkin          10.83.241.77    <none>            9411:30243/TCP                5
h
```

请注意，如果您的集群在不支持外部负载均衡器的环境中（例如minikube）运行，`istio-ingress` 的 `EXTERNAL-IP` 则会说 `<pending>`，您将需要使用服务NodePort或端口转发来访问应用程序。

2. 检查相应的Kubernetes pod已部署，所有容器启动并运行正常: "istio-pilot-*", "istio-mixer-*", "istio-ingress-*", "istio-egress-*", "istio-ca-*" (如果启用了Istio Auth), 和, 可选的, "grafana-*", "prometheus-*", "servicegraph-*" 和 "zipkin-*".

```
kubectl get pods
```

```
grafana-3836448452-vhc1v         1/1       Running   0          5h
istio-ca-3657790228-j21b9        1/1       Running   0          5h
istio-egress-1684034556-fhw89    1/1       Running   0          5h
istio-ingress-1842462111-j3vcs   1/1       Running   0          5h
istio-pilot-2275554717-93c43     2/2       Running   0          5h
istio-mixer-2104784889-20rm8     1/1       Running   0          5h
prometheus-3067433533-wlmt2      1/1       Running   0          5h
servicegraph-3127588006-pc5z3    1/1       Running   0          5h
zipkin-4057566570-k9m42          1/1       Running   0          5h
```

# 部署应用程序

现在您可以部署自己的应用程序，或者安装所提供的示例应用程序之一，例如 BookInfo。请注意，应用程序应使用 HTTP/1.1 或 HTTP/2.0 协议来实现其所有HTTP流量; HTTP/1.0不支持。

在部署应用程序时，必须使用 istioctl kube-inject 来自动将Envoy容器注入到应用程序容器中：

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

# 卸载

您可以使用 Istio Helm chart 进行卸载，或按照以下步骤操作。

1. 卸载Istio核心组件：
    - 如果Istio是不带Istio认证功能安装：

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

    - 如果Istio是带Istio认证功能安装：

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

2. 卸载RBAC Istio角色：
    - 如果安装了beta版本：

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

    - 如果安装了alpha版本：

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

3. 如果安装了Istio插件，请卸载它们：

   ```
   kubectl delete -f install/kubernetes/addons/
   ```

4. 删除Istio Kubernetes TPRs:

```
kubectl delete istioconfigs --all
kubectl delete thirdpartyresource istio-config.istio.io
```

# 下一步

- 请参阅 BookInfo 示例应用程序.
- 看看如何 测试 Istio Auth.

# 将服务集成到网格中

这个任务展示如何将Kubernetes上的应用程序与Istio进行集成。您将学习如何使用 istioctl kube-inject 将 Envoy sidecar 插入到部署中.

## 开始之前

这个任务假设你已经在Kubernetes上部署了Istio。如果还没有这样做，请先完成 安装步骤。

## 将Envoy sidecar注入到部署中

示例部署和用来演示此任务的服务。保存为 `apps.yaml` 。

```
apiVersion: v1
kind: Service
metadata:
  name: service-one
  labels:
    app: service-one
spec:
  ports:
  - port: 80
    targetPort: 8080
    name: http
  selector:
    app: service-one
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: service-one
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: service-one
    spec:
      containers:
      - name: app
        image: gcr.io/google_containers/echoserver:1.4
        ports:
        - containerPort: 8080
---
```

```
apiVersion: v1
kind: Service
metadata:
  name: service-two
  labels:
    app: service-two
spec:
  ports:
  - port: 80
    targetPort: 8080
    name: http-status
  selector:
    app: service-two
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: service-two
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: service-two
    spec:
      containers:
      - name: app
        image: gcr.io/google_containers/echoserver:1.4
        ports:
        - containerPort: 8080
```

Kubernetes Services 是正常使用Istio服务所必需的。服务端口必须命名，这些名称必须以 *http* 或 *grpc* 前缀开头，以利用Istio的L7路由功能，例如 `name: http-foo` 或者 `name: http` 很好。带有未命名端口或没有 *http* 或 *grpc* 前缀的服务将作为L4流量路由。

提交一个YAML资源到API服务器,带有被注入的 Envoy sidecar。以下任何一种方法都可以正常工作。

```
kubectl apply -f <(istioctl kube-inject -f apps.yaml)
```

发起一个从客户端（服务一）到服务器（service-two）的请求。

```
CLIENT=$(kubectl get pod -l app=service-one -o jsonpath='{.items[0].metadata.name}')
SERVER=$(kubectl get pod -l app=service-two -o jsonpath='{.items[0].metadata.name}')

kubectl exec -it ${CLIENT} -c app -- curl service-two:80 | grep x-request-id
```

```
x-request-id=a641eff7-eb82-4a4f-b67b-53cd3a03c399
```

验证流量被 Envoy sidecar 拦截。用sidecar访问日志比较HTTP响应中的 `x-request-id` 。`x-request-id` 是随机的. 出站请求日志中的IP是 service-two pod 的IP。

客户端pod的代理上的出站请求。

```
kubectl logs ${CLIENT} proxy | grep a641eff7-eb82-4a4f-b67b-53cd3a03c399
```

```
[2017-05-01T22:08:39.310Z] "GET / HTTP/1.1" 200 - 0 398 3 3 "-" "curl/7.47.0" "a641eff
7-eb82-4a4f-b67b-53cd3a03c399" "service-two" "10.4.180.7:8080"
```

服务器pod代理上的入站请求。

```
kubectl logs ${SERVER} proxy | grep a641eff7-eb82-4a4f-b67b-53cd3a03c399
```

```
[2017-05-01T22:08:39.310Z] "GET / HTTP/1.1" 200 - 0 398 2 0 "-" "curl/7.47.0" "a641eff
7-eb82-4a4f-b67b-53cd3a03c399" "service-two" "127.0.0.1:8080"
```

Envoy sidecar 不拦截相同 pod 内的容器到容器的流量,当流量是通过 localhost 通讯时。这是设计决定的。

```
kubectl exec -it ${SERVER} -c app -- curl localhost:8080 | grep x-request-id
```

# 理解发生了什么

`istioctl kube-inject` 在向Kubernetes API服务器提交之前，将额外的容器注入客户端的YAML资源。这个将最终被服务器端注入取代。使用

```
kubectl get deployment service-one -o yaml
```

来检查已修改的部署，并查找以下内容：

- 代理容器，其中包含Envoy proxy和agent来管理本地代理配置。
- 一个 init-container 用来编程 iptables.

代理容器以特定的UID运行，以便iptables可以将代理本身和重定向到代理的应用程序的出站流量区分开。

```
- args:
    - proxy
    - sidecar
    - "-v"
    - "2"
  env:
    -
      name: POD_NAME
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.name
    -
      name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.namespace
    -
      name: POD_IP
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: status.podIP
  image: "docker.io/istio/proxy:<...tag... >"
  imagePullPolicy: Always
  name: proxy
  securityContext:
    runAsUser: 1337
```

iptables 用于将所有入站和出站流量透明地重定向到代理。使用初始化容器有两个原因：

1. iptables 需要 NET_CAP_ADMIN.
2. sidecar的iptable规则是固定的，在pod创建后不需要更新。代理容器负责动态路由流量。

```
{
 "name":"init",
 "image":"docker.io/istio/init:<..tag...>",
 "args":[ "-p", "15001", "-u", "1337" ],
 "imagePullPolicy":"Always",
 "securityContext":{
   "capabilities":{
     "add":[
       "NET_ADMIN"
     ]
   }
 }
},
```

# 清理

删除示例服务和部署。

```
kubectl delete -f apps.yaml
```

# 下一步

- 查看 istioctl kube-inject 的完整文档
- 查看 BookInfo 示例, 更完整的Kubernetes与Istio集成的应用示例

# 启用入口流量

This task describes how to configure Istio to expose a service outside of the service mesh cluster. In a Kubernetes environment, Istio uses Kubernetes Ingress Resources to configure ingress behavior.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Make sure your current directory is the `istio` directory.
- Start the httpbin sample, which will be used as the destination service to be exposed externally.

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

## Configuring ingress (HTTP)

1. Create the Ingress Resource for the httpbin service

```
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: /headers
        backend:
          serviceName: httpbin
          servicePort: 8000
      - path: /delay/.*
        backend:
          serviceName: httpbin
          servicePort: 8000
EOF
```

Notice that in this example we are only exposing httpbin's two endpoints: `/headers` as an exact URI path and `/delay/` using an URI prefix.

2. Determine the ingress URL:

   - If your cluster is running in an environment that supports external load balancers, use the ingress' external address:

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

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

     ```
     export INGRESS_URL=130.211.10.121
     ```

   - If load balancers are not supported, use the ingress controller pod's hostIP:

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

     ```
     169.47.243.100
     ```

     along with the istio-ingress service's nodePort for port 80:

     ```
     kubectl 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_URL=169.47.243.100:31486
     ```

3. Access the httpbin service using *curl*:

   ```
   curl http://$INGRESS_URL/headers
   ```

```
{
  "headers": {
    "Accept": "*/*",
    "Content-Length": "0",
    "Host": "httpbin.default.svc.cluster.local:8000",
    "User-Agent": "curl/7.51.0",
    "X-Envoy-Expected-Rq-Timeout-Ms": "15000",
    "X-Request-Id": "3dd59054-6e26-4af5-87cf-a247bc634bab"
  }
}
```

# Configuring secure ingress (HTTPS)

1. Generate keys if necessary

   A private key and certificate can be created for testing using OpenSSL.

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

2. Create the secret using `kubectl`

   ```
   kubectl create secret tls ingress-secret --key /tmp/tls.key --cert /tmp/tls.crt
   ```

3. Create the Ingress Resource for the httpbin service

   ```
   cat <<EOF | kubectl create -f -
   apiVersion: extensions/v1beta1
   kind: Ingress
   metadata:
     name: secured-ingress
     annotations:
       kubernetes.io/ingress.class: istio
   spec:
     tls:
       - secretName: ingress-secret
     rules:
     - http:
         paths:
         - path: /ip
           backend:
             serviceName: httpbin
             servicePort: 8000
   EOF
   ```

Notice that in this example we are only exposing httpbin's `/ip` endpoint.

> Note: Envoy currently only allows a single TLS secret in the ingress since SNI is not yet supported.

4. Determine the secure ingress URL:

   - If your cluster is running in an environment that supports external load balancers, use the ingress' external address:

     ```
     kubectl get ingress secured-ingress -o wide
     ```

     ```
     NAME              HOSTS     ADDRESS                 PORTS     AGE
     secured-ingress   *         130.211.10.121          80, 443   1d
     ```

     ```
     export SECURE_INGRESS_URL=130.211.10.121
     ```

     > Note that in this case SECURE_INGRESS_URL should be the same as INGRESS_URL that you set previously.

   - If load balancers are not supported, use the ingress controller pod's hostIP:

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

     ```
     169.47.243.100
     ```

     along with the istio-ingress service's nodePort for port 443:

     ```
     kubectl 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 SECURE_INGRESS_URL=169.47.243.100:32254
     ```

5. Access the secured httpbin service using *curl*:

   ```
   curl -k https://$SECURE_INGRESS_URL/ip
   ```

```
{
  "origin": "129.42.161.35"
}
```

# Setting Istio rules on an edge service

Similar to inter-cluster requests, Istio routing rules can also be set for edge services that are called from outside the cluster. To illustrate we will use istioctl to set a timeout rule on calls to the httpbin service.

1. Invoke the httpbin `/delay` endpoint you exposed previously:

   ```
   time curl -o /dev/null -s -w "%{http_code}\n" http://$INGRESS_URL/delay/5
   ```

   ```
   200

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

   The request should return 200 (OK) in approximately 5 seconds.

2. Use `istioctl` to set a 3s timeout on calls to the httpbin service

   ```
   cat <<EOF | istioctl create
   type: route-rule
   name: httpbin-3s-rule
   spec:
     destination: httpbin.default.svc.cluster.local
     http_req_timeout:
       simple_timeout:
         timeout: 3s
   EOF
   ```

   Note that you may need to change the `default` namespace to the namespace of the `httpbin` application.

3. Wait a few seconds, then issue the *curl* request again:

   ```
   time curl -o /dev/null -s -w "%{http_code}\n" http://$INGRESS_URL/delay/5
   ```

```
504

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

This time a 504 (Gateway Timeout) appears after 3 seconds. Although httpbin was waiting 5 seconds, Istio cut off the request at 3 seconds.

> Note: HTTP fault injection (abort and delay) is not currently supported by ingress proxies.

# Understanding ingresses

Ingresses provide gateways for external traffic to enter the Istio service mesh and make the traffic management and policy features of Istio available for edge services.

In the preceding steps we created a service inside the Istio service mesh and showed how to expose both HTTP and HTTPS endpoints of the service to external traffic. We also showed how to control the ingress traffic using an Istio route rule.

# Cleanup

1. Remove the secret, Ingress Resource definitions and Istio rule.

```
istioctl delete route-rule httpbin-3s-rule
kubectl delete ingress simple-ingress secured-ingress
kubectl delete secret ingress-secret
```

2. Shutdown the httpbin service.

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

# What's next

- Learn more about routing rules.
- Learn how to expose external services by enabling egress traffic.

# 启用出口流量

By default, Istio-enabled services are unable to access URLs outside of the cluster because iptables is used in the pod to transparently redirect all outbound traffic to the sidecar proxy, which only handles intra-cluster destinations.

This task describes how to configure Istio to expose external services to Istio-enabled clients. You'll learn how to configure an external service and make requests to it via the Istio egress service or, alternatively, to simply enable direct calls to an external service.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Start the sleep sample which will be used as a test source for external calls.

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

  Note that any pod that you can `exec` and `curl` from would do.

## Using the Istio Egress service

Using the Istio Egress service, you can access any publicly accessible service from within your Istio cluster. In this task we will use httpbin.org and www.google.com as examples.

### Configuring the external services

1. Register an external HTTP service:

```
cat <<EOF | kubectl create -f -
apiVersion: v1
kind: Service
metadata:
 name: externalbin
spec:
 type: ExternalName
 externalName: httpbin.org
 ports:
 - port: 80
   # important to set protocol name
   name: http
EOF
```

2. Register an external HTTPS service:

```
cat <<EOF | kubectl create -f -
apiVersion: v1
kind: Service
metadata:
 name: securegoogle
spec:
 type: ExternalName
 externalName: www.google.com
 ports:
 - port: 443
   # important to set protocol name
   name: https
EOF
```

The `metadata.name` field is the url your internal apps will use when calling the external service. The `spec.externalName` is the DNS name of the external service. Egress Envoy expects external services to be listening on either port `80` for HTTP or port `443` for HTTPS.

## Make requests to the external services

1. Exec into the pod being used as the test source. For example, if you are using the sleep service, run the following commands:

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

2. Make a request to an external service using the `name` from the Service spec above followed by the path to the desired API endpoint:

```
curl http://externalbin/headers
```

3. For external services of type HTTPS, the port must be specified in the request. App clients should make the request over HTTP since the Egress Envoy will initiate HTTPS with the external service:

```
curl http://securegoogle:443
```

# Calling external services directly

The Istio Egress service currently only supports HTTP/HTTPS requests. If you want to access services with other protocols (e.g., mongodb://host/database), or if you simply don't want to use the Egress proxy, you will need to configure the source service's Envoy sidecar to prevent it from intercepting the external requests. This can be done using the `--includeIPRanges` option of istioctl kube-inject when starting the service.

The simplest way to use the `--includeIPRanges` option is to pass it the IP range(s) used for internal cluster services, thereby excluding external IPs from being redirected to the sidecar proxy. The values used for internal IP range(s), however, depends on where your cluster is running. For example, with Minikube the range is 10.0.0.1/24, so you would start the sleep service like this:

```
kubectl apply -f <(istioctl kube-inject -f samples/apps/sleep/sleep.yaml --includeIPRa
nges=10.0.0.1/24)
```

On IBM Bluemix, use:

```
kubectl apply -f <(istioctl kube-inject -f samples/apps/sleep/sleep.yaml --includeIPRa
nges=172.30.0.0/16,172.20.0.0/16)
```

On Google Container Engine (GKE) the ranges are not fixed, so you will need to run the `gcloud container clusters describe` command to determine the ranges to use. For example:

```
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/apps/sleep/sleep.yaml --includeIPRa
nges=10.4.0.0/14,10.7.240.0/20)
```

On Azure Container Service(ACS), use:

```
kubectl apply -f <(istioctl kube-inject -f samples/apps/sleep/sleep.yaml --includeIPRa
nges=10.244.0.0/16,10.240.0.0/16)
```

After starting your service this way, the Istio sidecar will only intercept and manage internal requests within the cluster. Any external request will simply bypass the sidecar and go straight to its intended destination.

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

# Understanding what happened

In this task we looked at two ways to call external services from within an Istio cluster:

1. Using the Istio egress service (recommended)
2. Configuring the Istio sidecar to exclude external IPs from its remapped IP table

The first approach (Egress service) currently only supports HTTP(S) requests, but allows you to use all of the same Istio service mesh features for calls to services within or outside of the cluster.

The second approach bypasses the Istio sidecar proxy, giving your services direct access to any external URL. However, configuring the proxy this way does require cloud provider specific knowledge and configuration.

# Cleanup

1. Remove the external services.

   ```
   kubectl delete service externalbin securegoogle
   ```

2. Shutdown the sleep service.

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

# What's next

- Read more about the egress service.
- Learn how to use Istio's request routing features.

# 配置请求路由

This task shows you how to configure dynamic request routing based on weights and HTTP headers.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Deploy the BookInfo sample application.

## Content-based routing

Because the BookInfo sample deploys 3 versions of the reviews microservice, we need to set a default route. Otherwise if you access the application several times, you'll notice that sometimes the output contains star ratings. This is because without an explicit default version set, Istio will route requests to all available versions of a service in a random fashion.

> Note: This task assumes you don't have any routes set yet. If you've already created conflicting route rules for the sample, you'll need to use `replace` rather than `create` in one or both of the following commands.

1. Set the default version for all microservices to v1.

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

   You can display the routes that are defined with the following command:

   ```
   istioctl get route-rules -o yaml
   ```

```
type: route-rule
name: ratings-default
namespace: default
spec:
  destination: ratings.default.svc.cluster.local
  precedence: 1
  route:
  - tags:
      version: v1
    weight: 100
---
type: route-rule
name: reviews-default
namespace: default
spec:
  destination: reviews.default.svc.cluster.local
  precedence: 1
  route:
  - tags:
      version: v1
    weight: 100
---
type: route-rule
name: details-default
namespace: default
spec:
  destination: details.default.svc.cluster.local
  precedence: 1
  route:
  - tags:
      version: v1
    weight: 100
---
type: route-rule
name: productpage-default
namespace: default
spec:
  destination: productpage.default.svc.cluster.local
  precedence: 1
  route:
  - tags:
      version: v1
    weight: 100
---
```

Since rule propagation to the proxies is asynchronous, you should wait a few seconds for the rules to propagate to all pods before attempting to access the application.

2. Open the BookInfo URL (http://$GATEWAY_URL/productpage) in your browser

You should see the BookInfo application productpage displayed. Notice that the `productpage` is displayed with no rating stars since `reviews:v1` does not access the ratings service.

3. Route a specific user to `reviews:v2`

   Lets enable the ratings service for test user "jason" by routing productpage traffic to `reviews:v2` instances.

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

   Confirm the rule is created:

   ```
   istioctl get route-rule reviews-test-v2
   ```

   ```
   destination: reviews.default.svc.cluster.local
   match:
     httpHeaders:
       cookie:
         regex: ^(.*?;)?(user=jason)(;.*)?$
   precedence: 2
   route:
   - tags:
       version: v2
   ```

4. Log in as user "jason" at the `productpage` web page.

   You should now see ratings (1-5 stars) next to each review. Notice that if you log in as any other user, you will continue to see `reviews:v1`.

# Understanding what happened

In this task, you used Istio to send 100% of the traffic to the v1 version of each of the BookInfo services. You then set a rule to selectively send traffic to version v2 of the reviews service based on a header (i.e., a user cookie) in a request.

Once the v2 version has been tested to our satisfaction, we could use Istio to send traffic from all users to v2, optionally in a gradual fashion by using a sequence of rules with weights less than 100 to migrate traffic in steps, for example 10, 20, 30, ... 100%.

If you now proceed to the fault injection task, you will see that with simple testing, the v2 version of the reviews service has a bug, which is fixed in v3. So after exploring that task, you can route all user traffic from `reviews:v1` to `reviews:v3` in two steps as follows:

1. First, transfer 50% of traffic from `reviews:v1` to `reviews:v3` with the following command:

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

   Notice that we are using `istioctl replace` instead of `create`.

2. To see the new version you need to either Log out as test user "jason" or delete the test rules that we created exclusively for him:

   ```
   istioctl delete route-rule reviews-test-v2
   istioctl delete route-rule ratings-test-delay
   ```

   You should now see *red* colored star ratings approximately 50% of the time when you refresh the `productpage`.

   > Note: With the Envoy sidecar implementation, you may need to refresh the `productpage` multiple times to see the proper distribution. You can modify the rules to route 90% of the traffic to v3 to see red stars more often.

3. When version v3 of the reviews microservice is stable, route 100% of the traffic to `reviews:v3`:

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

   You can now log in to the `productpage` as any user and you should always see book reviews with *red* colored star ratings for each review.

# What's next

- Learn more about request routing.
- Test the BookInfo application resiliency by injecting faults.
- If you are not planning to explore any follow-on tasks, refer to the BookInfo cleanup instructions to shutdown the application and cleanup the associated rules.

# 故障注入

This task shows how to inject delays and test the resiliency of your application.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Deploy the BookInfo sample application.
- Initialize the application version routing by either first doing the request routing task or by running following commands:

  > Note: This assumes you don't have any routes set yet. If you've already created conflicting route rules for the sample, you'll need to use `replace` rather than `create` in one or both of the following commands.

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

## Fault injection

To test our BookInfo application microservices for resiliency, we will *inject a 7s delay* between the reviews:v2 and ratings microservices. Since the *reviews:v2* service has a 10s timeout for its calls to the ratings service, we expect the end-to-end flow to continue without any errors.

1. Create a fault injection rule to delay traffic coming from user "jason" (our test user)

   ```
   istioctl create -f samples/apps/bookinfo/destination-ratings-test-delay.yaml
   ```

   Confirm the rule is created:

   ```
   istioctl get route-rule ratings-test-delay
   ```

```
destination: ratings.default.svc.cluster.local
httpFault:
  delay:
    fixedDelay: 7s
    percent: 100
match:
  httpHeaders:
    cookie:
      regex: "^(.*?;)?(user=jason)(;.*)?$"
precedence: 2
route:
- tags:
    version: v1
```

Allow several seconds to account for rule propagation delay to all pods.

2. Observe application behavior

   If the application's front page was set to correctly handle delays, we expect it to load within approximately 7 seconds. To see the web page response times, open the *Developer Tools* menu in IE, Chrome or Firefox (typically, key combination *Ctrl+Shift+I* or *Alt+Cmd+I*) and reload the `productpage` web page.

   You will see that the webpage loads in about 6 seconds. The reviews section will show *Sorry, product reviews are currently unavailable for this book*.

# Understanding what happened

The reason that the entire reviews service has failed is because our BookInfo application has a bug. The timeout between the productpage and reviews service is less (3s + 1 retry = 6s total) than the timeout between the reviews and ratings service (10s). These kinds of bugs can occur in typical enterprise applications where different teams develop different microservices independently. Istio's fault injection rules help you identify such anomalies without impacting end users.

> Notice that we are restricting the failure impact to user "jason" only. If you login as any other user, you would not experience any delays.

**Fixing the bug:** At this point we would normally fix the problem by either increasing the productpage timeout or decreasing the reviews to ratings service timeout, terminate and restart the fixed microservice, and then confirm that the `productpage` returns its response without any errors.

However, we already have this fix running in v3 of the reviews service, so we can simply fix the problem by migrating all traffic to `reviews:v3` as described in the request routing task.

(Left as an exercise for the reader - change the delay rule to use a 2.8 second delay and then run it against the v3 version of reviews.)

# What's next

- Learn more about fault injection.
- Limit requests to the BookInfo `ratings` service with Istio rate limiting.
- If you are not planning to explore any follow-on tasks, refer to the BookInfo cleanup instructions to shutdown the application and cleanup the associated rules.

# 设置请求超时

This task shows you how to setup request timeouts in Envoy using Istio.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Deploy the BookInfo sample application.
- Initialize the application version routing by running the following command:

> Note: This assumes you don't have any routes set yet. If you've already created route rules for the sample, you'll need to use `replace` rather than `create` in the following command.

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

## Request timeouts

A timeout for http requests can be specified using the *httpReqTimeout* field of a routing rule. By default, the timeout is 15 seconds, but in this task we'll override the `reviews` service timeout to 1 second. To see its effect, however, we'll also introduce an artificial 2 second delay in calls to the `ratings` service.

1. Route requests to v2 of the `reviews` service, i.e., a version that calls the `ratings` service

   ```
   cat <<EOF | istioctl replace
   type: route-rule
   name: reviews-default
   spec:
     destination: reviews.default.svc.cluster.local
     route:
     - tags:
         version: v2
   EOF
   ```

2. Add a 2 second delay to calls to the `ratings` service:

```
cat <<EOF | istioctl replace
type: route-rule
name: ratings-default
spec:
  destination: ratings.default.svc.cluster.local
  route:
  - tags:
      version: v1
  httpFault:
    delay:
      percent: 100
      fixedDelay: 2s
EOF
```

3. Open the BookInfo URL (http://$GATEWAY_URL/productpage) in your browser

   You should see the BookInfo application working normally (with ratings stars displayed), but there is a 2 second delay whenever you refresh the page.

4. Now add a 1 second request timeout for calls to the `reviews` service

```
cat <<EOF | istioctl replace
type: route-rule
name: reviews-default
spec:
  destination: reviews.default.svc.cluster.local
  route:
  - tags:
      version: v2
  httpReqTimeout:
    simpleTimeout:
      timeout: 1s
EOF
```

5. Refresh the BookInfo web page

   You should now see that it returns in 1 second (instead of 2), but the reviews are unavailable.

# Understanding what happened

In this task, you used Istio to set the request timeout for calls to the `reviews` microservice to 1 second (instead of the default 15 seconds). Since the `reviews` service subsequently calls the `ratings` service when handling requests, you used Istio to inject a 2 second delay in calls to `ratings`, so that you would cause the `reviews` service to take longer than 1 second to complete and consequently you could see the timeout in action.

You observed that the BookInfo productpage (which calls the `reviews` service to populate the page), instead of displaying reviews, displayed the message: Sorry, product reviews are currently unavailable for this book. This was the result of it receiving the timeout error from the `reviews` service.

If you check out the fault injection task, you'll find out that the `productpage` microservice also has its own application-level timeout (3 seconds) for calls to the `reviews` microservice. Notice that in this task we used an Istio route rule to set the timeout to 1 second. Had you instead set the timeout to something greater than 3 seconds (e.g., 4 seconds) the timeout would have had no effect since the more restrictive of the two will take precedence. More details can be found here.

One more thing to note about timeouts in Istio is that in addition to overriding them in route rules, as you did in this task, they can also be overridden on a per-request basis if the application adds an "x-envoy-upstream-rq-timeout-ms" header on outbound requests. In the header the timeout is specified in millisecond (instead of second) units.

# What's next

- Learn more about failure handling.
- Learn more about routing rules.
- If you are not planning to explore any follow-on tasks, refer to the BookInfo cleanup instructions to shutdown the application and cleanup the associated rules.

# 启用限速

This task shows you how to use Istio to dynamically limit the traffic to a service.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Deploy the BookInfo sample application.
- Initialize the application version routing to direct `reviews` service requests from test user "jason" to version v2 and requests from any other user to v3.

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

> Note: if you have conflicting rule that you set in previous tasks, use `istioctl replace` instead of `istioctl create`.

## Rate limits

Istio enables users to rate limit traffic to a service.

Consider `ratings` as an external paid service like Rotten Tomatoes® with `1qps` free quota. Using Istio we can ensure that `1qps` is not breached.

1. Point your browser at the BookInfo `productpage` (http://$GATEWAY_URL/productpage).

   If you log in as user "jason", you should see black ratings stars with each review, indicating that the `ratings` service is being called by the "v2" version of the `reviews` service.

   If you log in as any other user (or logout) you should see red ratings stars with each review, indicating that the `ratings` service is being called by the "v3" version of the `reviews` service.

2. Configure mixer with the rate limit.

   Save this as ratelimit.yaml:

```
rules:
- aspects:
  - kind: quotas
    params:
      quotas:
      - descriptorName: RequestCount
        maxAmount: 1
        expiration: 1s
```

and then run the following command:

```
istioctl mixer rule create global ratings.default.svc.cluster.local -f ratelimit.y
aml
```

`istioctl` sets configuration for `subject=ratings.default.svc.cluster.local`

3. Generate load on the `productpage` with the following command:

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

4. Refresh the `productpage` in your browser.

   While the load generator is running (i.e., generating more than 1 req/s), the traffic generated by your browser will be rate limited. Notice that if you log in as user "jason" or any other user, the `reviews` service is unable to access the `ratings` service, so you stop seeing stars, red or black.

# Conditional rate limits

In the previous example we applied a rate limit to the `ratings` service without regard to any other attributes. It is possible to conditionally apply rate limits based on attributes like the source of the traffic.

The following configuration applies a `1qps` rate limit only to version `v3` of `reviews`.

1. Configure mixer with the conditional rate limit.

   Save this as ratelimit-conditional.yaml:

```
rules:
- selector: source.labels["app"]=="reviews" && source.labels["version"] == "v3"
  aspects:
  - kind: quotas
    params:
      quotas:
      - descriptorName: RequestCount
        maxAmount: 1
        expiration: 1s
```

and then run the following command:

```
istioctl mixer rule create global ratings.default.svc.cluster.local -f ratelimit-conditional.yaml
```

Notice the rule is the same as the previous example, only this one uses a `selector` to apply the ratelimit only for requests from `reviews:v3`.

2. Generate load on the `productpage` with the following command:

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

3. Refresh the `productpage` in your browser.

   As in the previous example, while the load generator is running (i.e., generating more than 1 req/s), the traffic generated by your browser will be rate limited, but this time only if the request is from `reviews:v3`. Notice that this time if you log in as user "jason" (the `reviews:v2` user) you will continue to see the black ratings stars. Only the other users will stop seeing the red ratings stars while the load generator is running.

# Understanding rate limits

In the preceding examples we saw how Mixer applies rate limits to requests that match certain conditions.

Every distinct rate limit configuration represents a counter. If the number of requests in the last `expiration` duration exceed `maxAmount`, Mixer returns a `RESOURCE_EXHAUSTED` message to the proxy. The proxy in turn returns status `HTTP 429` to the caller.

Multiple rate limits may apply to the same request.

Mixer `MemQuota` adapter uses a sliding window of sub second resolution to enforce rate limits.

Consider the following example

```
descriptorName: RequestCount
maxAmount: 5000
expiration: 5s
labels:
  label1: target.service
```

This defines a set of counters with a limit of `5000` per every `5 seconds`. Individual counters within the set are identified by unique keys. A key is formed on the request path by using all parameters of the configuration. Here we introduce the notion of labels that enable creation of more granular counter keys. When a request arrives at Mixer with `target.service=ratings` it forms the following counter key.

`$aspect_id;RequestCount;maxAmount=5000;expiration=5s;label1=ratings`

Using `target.service` in the counter key enables independent rate limits for every service. In absence of `target.service` as part of the key, the same counter location is used by all services resulting in combined rate limit of `5000` requests per `5 seconds`

Mixer supports an arbitrary number of labels by defining `QuotaDescriptors`.

```
name: RequestCount
rate_limit: true
labels:
   label1: 1 # STRING
```

Here we define `RequestCount` quota descriptor that takes 1 string label. We recommend using meaningful label names even though label names are arbitrary.

```
name: RequestCount_byService_byUser
rate_limit: true
labels:
   service: 1 # STRING
   user: 1 # STRING
```

Mixer expects `user,service` labels when the `RequestCount_byService_byUser` descriptor is used and produces the following config validation error if any labels are missing.

```
* quotas: aspect validation failed: 1 error occurred:
* quotas[RequestCount_byService_byUser].labels: wrong dimensions: descriptor expects 2
 labels, found 0 labels
```

# Cleanup

- Remove the mixer configuration rule:

```
istioctl mixer rule create global ratings.default.svc.cluster.local -f samples/app
s/bookinfo/mixer-rule-empty-rule.yaml
```

> Note: removing a rule by setting an empty rule list is a temporary workaround because `istioctl delete` does not yet support mixer rules.

- Remove the application routing rules:

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

# What's next

- Learn more about Mixer and Mixer Config.
- Discover the full Attribute Vocabulary.
- Read the reference guide to Writing Config.
- If you are not planning to explore any follow-on tasks, refer to the BookInfo cleanup instructions to shutdown the application and cleanup the associated rules.

# 启用简单的访问控制

This task shows how to use Istio to control access to a service.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.
- Deploy the BookInfo sample application.
- Initialize the application version routing to direct `reviews` service requests from test user "jason" to version v2 and requests from any other user to v3.

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

> Note: if you have conflicting rule that you set in previous tasks, use `istioctl replace` instead of `istioctl create`.

## Access control using *denials*

Using Istio you can control access to a service based on any attributes that are available within Mixer. This simple form of access control is based on conditionally denying requests using Mixer selectors.

Consider the BookInfo sample application where the `ratings` service is accessed by multiple versions of the `reviews` service. We would like to cut off access to version `v3` of the `reviews` service.

1. Point your browser at the BookInfo `productpage` (http://$GATEWAY_URL/productpage).

   If you log in as user "jason", you should see black ratings stars with each review, indicating that the `ratings` service is being called by the "v2" version of the `reviews` service.

   If you log in as any other user (or logout) you should see red ratings stars with each review, indicating that the `ratings` service is being called by the "v3" version of the `reviews` service.

2. Explicitly deny access to version `v3` of the `reviews` service.

```
istioctl mixer rule create global ratings.default.svc.cluster.local -f samples/app
s/bookinfo/mixer-rule-ratings-denial.yaml
```

This command sets configuration for `subject=ratings.default.svc.cluster.local`. You can display the current configuration with the following command:

```
istioctl mixer rule get global ratings.default.svc.cluster.local
```

which should produce:

```
rules:
- aspects:
  - kind: denials
  selector: source.labels["app"]=="reviews" && source.labels["version"] == "v3"
```

This rule uses the `denials` aspect to deny requests coming from version `v3` of the reviews service. The `denials` aspect always denies requests with a pre-configured status code and message. The status code and the message is specified in the DenyChecker adapter configuration.

3. Refresh the `productpage` in your browser.

   If you are logged out or logged in as any user other than "jason" you will no longer see red ratings stars because the `reviews:v3` service has been denied access to the `ratings` service. Notice, however, that if you log in as user "jason" (the `reviews:v2` user) you continue to see the black ratings stars.

# Access control using *whitelists*

Istio also supports attribute-based white and blacklists. Using a whitelist is a two step process.

1. Add an adapter definition for the `genericListChecker` adapter that lists versions `v1, v2`:

   ```
   - name: versionList
     impl: genericListChecker
     params:
       listEntries: ["v1", "v2"]
   ```

2. Enable `whitelist` checking by using the `lists` aspect:

```
rules:
  aspects:
  - kind: lists
    adapter: versionList
    params:
      blacklist: false
      checkExpression: source.labels["version"]
```

`checkExpression` is evaluated and checked against the list `[v1, v2]`. The check behavior can be changed to a blacklist by specifying `blacklist: true`. The expression evaluator returns the value of the `version` label as specified by the `checkExpression` key.

The current version of `istioctl` does not yet support pushing adapter configurations like the one in step 1. There is, however, a workaround that you can use if you want to try it out anyway.

# Cleanup

- Remove the mixer configuration rule:

```
istioctl mixer rule delete global ratings.default.svc.cluster.local
```

- Remove the application routing rules:

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

# What's next

- Learn more about Mixer and Mixer Config.
- Discover the full Attribute Vocabulary.
- Read the reference guide to Writing Config.

# 测试Istio Auth

Through this task, you will learn how to:

- Verify Istio Auth setup
- Manually test Istio Auth

# Before you begin

This task assumes you have:

- Installed Istio with Auth by following the Istio installation task. Note to choose "enable Istio Auth feature" at step 5 in "Installation steps".

# Verifying Istio Auth setup

The following commands assume the services are deployed in the default namespace. Use the parameter *-n yournamespace* to specify a namespace other than the default one.

## Verifying Istio CA

Verify the cluster-level CA is running:

```
kubectl get deploy -l istio=istio-ca
```

```
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
istio-ca   1         1         1            1           1m
```

Istio CA is up if the "AVAILABLE" column is 1.

## Verifying service configuration

1. Verify AuthPolicy setting in ConfigMap.

   ```
   kubectl get configmap istio -o yaml | grep authPolicy | head -1
   ```

   Istio Auth is enabled if the line `authPolicy: MUTUAL_TLS` is uncommented.

# Testing Istio Auth

When running Istio auth-enabled services, you can use curl in one service's envoy to send request to other services. For example, after starting the BookInfo sample application you can ssh into the envoy container of `productpage` service, and send request to other services by curl.

There are several steps:

1. get the productpage pod name

   ```
   kubectl get pods -l app=productpage
   ```

   ```
   NAME                              READY     STATUS    RESTARTS   AGE
   productpage-v1-4184313719-5mxjc   2/2       Running   0          23h
   ```

   Make sure the pod is "Running".

2. ssh into the envoy container

   ```
   kubectl exec -it productpage-v1-4184313719-5mxjc -c istio-proxy /bin/bash
   ```

3. make sure the key/cert is in /etc/certs/ directory

   ```
   ls /etc/certs/
   ```

   ```
   cert-chain.pem   key.pem   root-cert.pem
   ```

   Note that cert-chain.pem is envoy's cert that needs to present to the other side. key.pem is envoy's private key paired with cert-chain.pem. root-cert.pem is the root cert to verify the other side's cert. Currently we only have one CA, so all envoys have the same root-cert.pem.

4. send requests to another service, for example, details.

   ```
   curl https://details:9080 -v --key /etc/certs/key.pem --cert /etc/certs/cert-chain
   .pem --cacert /etc/certs/root-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
...
```

The service name and port are defined here.

Note that Istio uses Kubernetes service account as service identity, which offers stronger security than service name (refer here for more information). Thus the certificates used in Istio do not have service name, which is the information that curl needs to verify server identity. As a result, we use curl option '-k' to prevent the curl client from verifying service identity in server's (i.e., productpage) certificate. Please check secure naming here for more information about how the client verifies the server's identity in Istio.

# 收集指标和日志

This task shows how to configure Mixer to automatically gather telemetry for a service within a cluster. At the end of this task, a new metric and a new log stream will be enabled for calls to a specific service within your cluster.

The BookInfo sample application is used as the example application throughout this task.

## Before you begin

- Install Istio in your kubernetes cluster and deploy an application.
- Configure your environment to support calling `istioctl mixer`. This may require setting up port-forwarding for the Mixer Config API as described in the reference docs for `istioctl mixer`.
- Configure your environment to support accessing the Istio dashboard, as described in the Installation Guide. This requires installing the optional add-ons (Prometheus and Grafana), as well as verifying access to the dashboard. The Istio dashboard will be used to verify task success.

## Collecting new telemetry data

1. Create a new YAML file to hold configuration for the new metric and log stream that Istio will generate and collect automatically.

   Save the following as `new_rule.yaml`:

   ```
   
   ```

2. Pick a target service for the new rule.

   If using the BookInfo sample, select `reviews.default.svc.cluster.local`. A fully-qualified domain name for the service is required in the following steps.

3. Validate that the selected service has no service-specific rules already applied.

   ```
   istioctl mixer rule get reviews.default.svc.cluster.local reviews.default.svc.clus
   ter.local
   ```

The expected output is:

```
Error: the server could not find the requested resource
```

If your selected service has service-specific rules, update `new_rule.yaml` to include the existing rules appropriately. Append the rule from `new_rule.yaml` to the existing `rules` block and save the updated content back over `new_rule.yaml`.

4. Push the new configuration to Mixer for a specific service.

```
istioctl mixer rule create reviews.default.svc.cluster.local reviews.default.svc.c
luster.local -f new_rule.yaml
```

5. Send traffic to that service.

   For the BookInfo sample, visit `http://$GATEWAY_URL/productpage` in your web browser or issue the following command:

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

   For purposes of this task, please refresh the page several times or issue the curl command a few times to generate traffic.

6. Verify that the new metric is being collected.

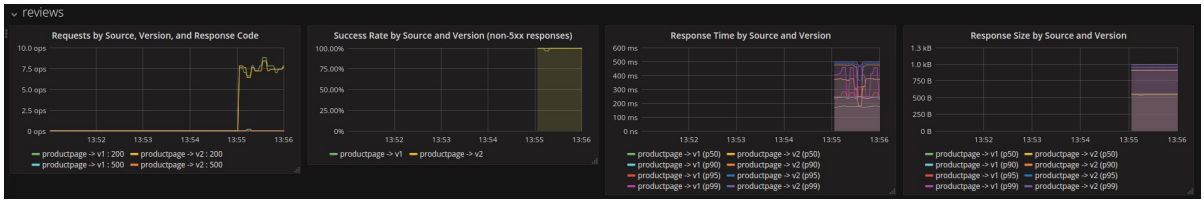
   Setup port-forwarding for Grafana:

```
kubectl port-forward $(kubectl get pod -l app=grafana -o jsonpath='{.items[0].meta
data.name}') 3000:3000 &
```

   Then open the Istio dashboard in a web browser:
   http://localhost:3000/dashboard/db/istio-dashboard

   One of the rows in the dashboard will be named "reviews". If that row is not visible, please refresh the dashboard page. The "reviews" row contains a graph entitled "Response Size by Source And Version". The graph displays a breakdown of the distribution of Response Sizes returned by the "reviews" service.

   The request from the previous step is reflected in the graphs. This looks similar to:




Istio Dashboard for Reviews Service

7. Verify that the logs stream has been created and is being populated for requests.

   Search through the logs for the Mixer pod as follows:

   ```
   kubectl logs $(kubectl get pods -l istio=mixer -o jsonpath='{.items[0].metadata.name}') | grep \"combined_log\"
   ```

   The expected output is similar to:

   ```
   {"logName":"combined_log","labels":{"referer":"","responseSize":871,"timestamp":"2017-04-29T02:11:54.989466058Z","url":"/reviews","userAgent":"python-requests/2.11.1"},"textPayload":"- - - [29/Apr/2017:02:11:54 +0000] \"- /reviews -\" - 871 - python-requests/2.11.1"}
   ```

# Understanding the new telemetry rule

In this task, you added a new rule for a service within your cluster. The new rule instructed Mixer to automatically generate and report a new metric and a new log stream for all traffic going to a specific service.

The new rule was comprised of a new `aspect` definitions. These `aspect` definitions were for the aspect kind of `metrics` and `access-logs`.

## Understanding the rule's metrics aspect

The `metrics` aspect directs Mixer to report metrics to the `prometheus` adapter. The adapter `params` tell Mixer *how* to generate metric values for any given request, based on the attributes reported by Envoy (and generated by Mixer itself).

The schema for the metric came from a predefined metric `descriptor` known to Mixer. In this task, the descriptor used was `response_size`. The `response_size` metric descriptor uses buckets to record a distribution of values, making it easier for the backend metrics systems to provide summary statistics for a bunch of requests in aggregate (as is often desirable when looking at response sizes).

The new rule instructs Mixer to generate values for the metric based on the values of the attribute `response.size`. A default values of `0` was added, in case Envoy does not report the values as expected.

A set of dimensions were also configured for the metric value, via the `labels` chunks of configuration. For the new metric, the dimensions were `source`, `target`, `service`, `version`, `method`, and `response_code`.

Dimensions provide a way to slice, aggregate, and analyze metric data according to different needs and directions of inquiry. For instance, it may be desirable to only consider response sizes for non-error responses when troubleshooting the rollout of a new application version.

The new rule instructs Mixer to populate values for these dimensions based on attribute values. For instance, for the `service` dimension, the new rule requests that the value be taken from the `target.labels["app"]` attribute. If that attribute value is not populated, the rule instructs Mixer to use a default value of `"unknown"`.

At the moment, it is not possible to programmatically generate new metric descriptors for use within Mixer. As a result, all new metric configurations must use one of the predefined metrics descriptors: `request_count`, `request_duration`, `request_size`, and `response_size`.

Work is ongoing to extend the Mixer Config API to add support for creating new descriptors.

## Understanding the rule's access_logs aspect

The `access-logs` aspect directs Mixer to send access logs to the `default` adapter (typically, `stdioLogger`). The adapter `params` tell Mixer *how* to generate the access logs for incoming requests based on attributes reported by Envoy.

The `logName` parameter is used by Mixer to identify a logs stream. In this task, the log name `combined_log` was used to identify the log stream amidst the rest of the Mixer logging output. This name should be used to uniquely identify log streams to various logging backends.

The `log` section of the rule describes the shape of the access log that Mixer will generate when the rule is applied. In this task, the pre-configured definition for an access log named `accesslog.combined` was used. It is based on the well-known Combined Log Format.

Access logs use a template to generate a plaintext log from a set of named arguments. The template is defined in the configured `descriptor` for the aspect. In this task, the template used is defined in the descriptor named `accesslog.combined`. The set of inputs to the `template_expressions` is fixed in the descriptor and cannot be altered in aspect configuration.

The `template_expressions` describe how to translate attribute values into the named arguments for the template processing. For example, the value for `userAgent` is to be derived directly from the value for the attribute `request.headers["user-agent"]`.

Mixer supports structured log generation in addition to plaintext logs. In this task, a set of `labels` to populate for structured log generation was configured. These `labels` are populated from attribute values according to attribute expressions, in exactly the same manner as the `template_expressions`.

While it is common practice to include the same set of arguments in the `labels` as in the `template_expressions`, this is not required. Mixer will generate the `labels` completely independently of the `template_expressions`.

As with metric descriptors, it is not currently possible to programmatically generate new access logs descriptors. Work is ongoing to extend the Mixer Config API to add support for creating new descriptors.

# What's next

- Learn more about Mixer and Mixer Config.
- Discover the full Attribute Vocabulary.
- Read the reference guide to Writing Config.
- If you are not planning to explore any follow-on tasks, refer to the BookInfo cleanup instructions to shutdown the application and cleanup the associated rules.

# 分布式请求跟踪

This task shows you how Istio-enabled applications can be configured to collect trace spans using Zipkin. After completing this task, you should understand all of the assumptions about your application and how to have it participate in tracing, regardless of what language/framework/platform you use to build your application.

The BookInfo sample is used as the example application for this task.

## Before you begin

- Setup Istio by following the instructions in the Installation guide.

  If you didn't start the Zipkin addon during installation, you can run the following command to start it now:

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

- Deploy the BookInfo sample application.

## Accessing the Zipkin dashboard

Setup access to the Zipkin dashboard URL using port-forwarding:

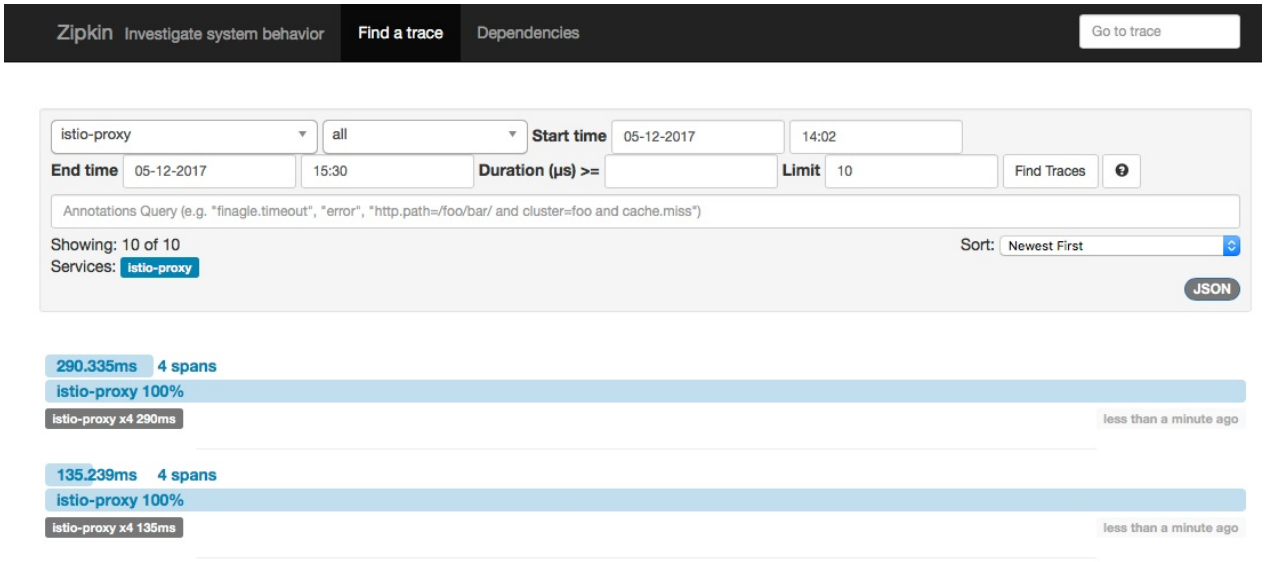
```
kubectl port-forward $(kubectl get pod -l app=zipkin -o jsonpath='{.items[0].metadata.
name}') 9411:9411 &
```

Then open your browser at http://localhost:9411

## Generating traces using the BookInfo sample

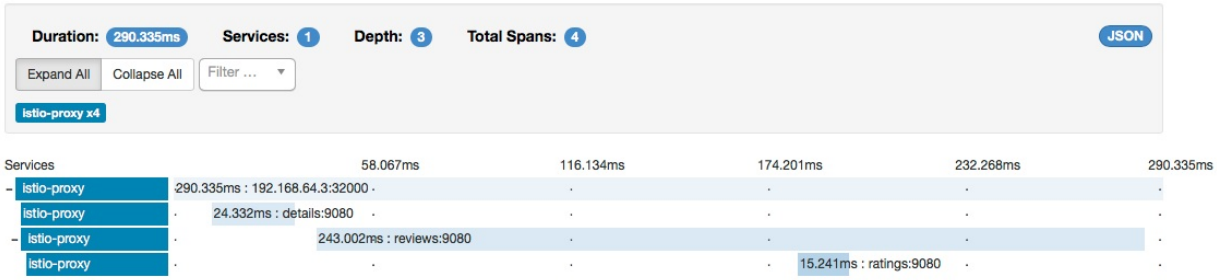
With the BookInfo application up and running, generate trace information by accessing `http://$GATEWAY_URL/productpage` one or more times.

If you now look at the Zipkin dashboard, you should see something similar to the following:




Zipkin Istio Dashboard

If you click on the top (most recent) trace, you should see the details corresponding to your latest refresh of the `/productpage`. The page should look something like this:




Zipkin Istio Dashboard

As you can see, there are 4 spans (only 3, if version v1 of the `reviews` service was used), where each span corresponds to a BookInfo service invoked during the execution of a `/productpage` request. Although every service has the same label, `istio-proxy`, because the tracing is being done by the Istio sidecar (Envoy proxy) which wraps the call to the actual service, the label of the destination (to the right) identifies the service for which the time is represented by each line.

The first line represents the external call to the `productpage` service. The label `192.168.64.3:32000` is the host value used for the external request (i.e., $GATEWAY_URL). As you can see in the trace, the request took a total of roughly 290ms to complete. During its execution, the `productpage` called the `details` service, which took about 24ms, and then called the `reviews` service. The `reviews` service took about 243ms to execute, including a 15ms call to `ratings`.

# Understanding what happened

Although Istio proxies are able to automatically send spans to Zipkin, they need some hints to tie together the entire trace. Applications need to propagate the appropriate HTTP headers so that when the proxies send span information to Zipkin, the spans can be correlated correctly into a single trace.

To do this, an application needs to collect and propagate the following headers from the incoming request to any outgoing requests:

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

If you look in the sample services, you can see that the productpage application (Python) extracts the required headers from an HTTP request:

```
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
```

The reviews application (Java) does something similar:

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

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

When you make downstream calls in your applications, make sure to include these headers.

# What's next

- Learn more about Metrics and Logs
- If you are not planning to explore any follow-on tasks, refer to the BookInfo cleanup instructions to shutdown the application and cleanup the associated rules.

# 示例

示例包括用于istio的可以完整工作的例子, 您可以实验.

- BookInfo: 该示例部署了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。

# 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。

## 开始之前

- 如果您使用GKE，请确保您的集群至少有4个标准GKE节点。
- 按照 安装指南 中的说明安装 Istio 。

## 概况

在本示例中，我们将部署一个简单的应用程序，显示有关书籍的信息，类似于在线书店的单个目录条目。在页面上显示的是书的描述，书籍详细信息（ISBN，页数等）和一点书评。

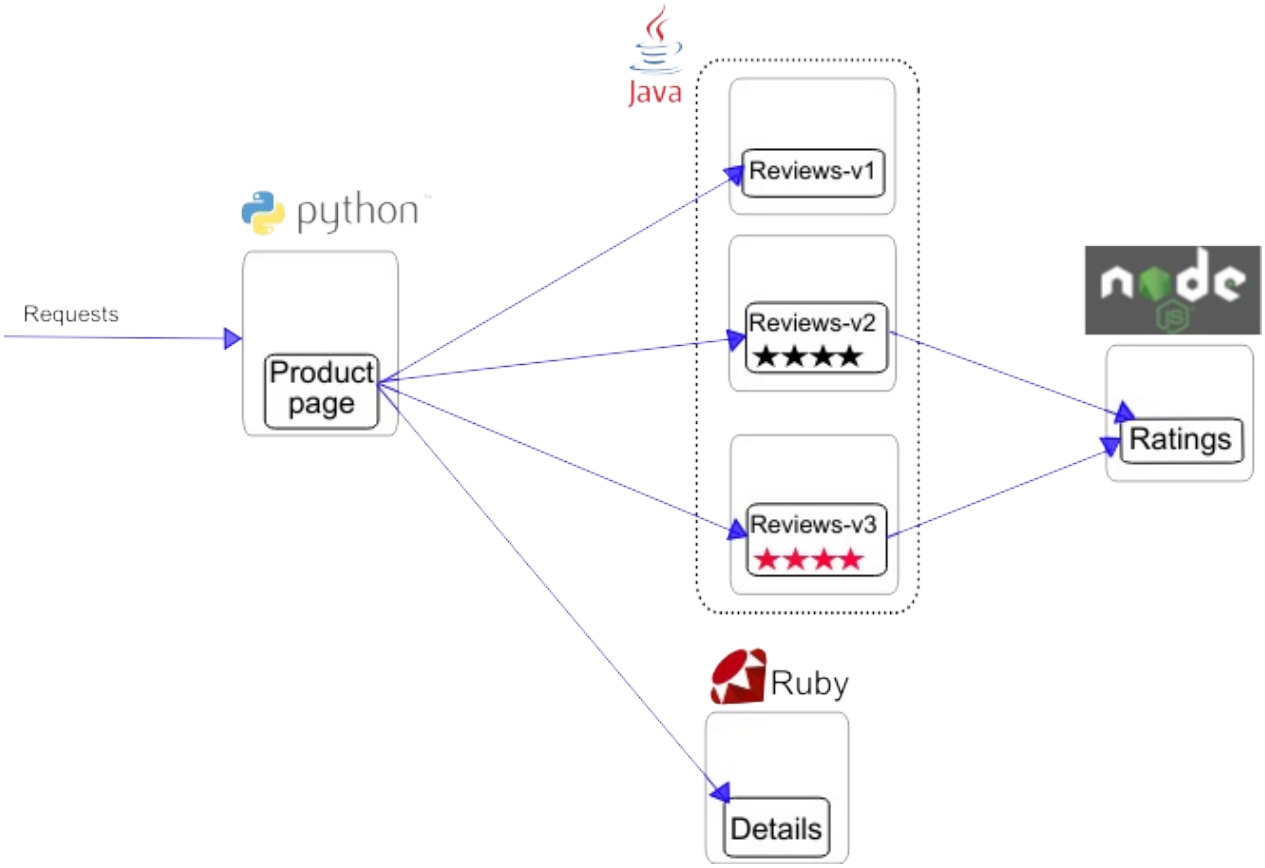
BookInfo 应用程序分为四个单独的微服务器：

- *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 微服务包含书籍的书评. 它也调用 *ratings* 微服务.
- *ratings*. ratings 微服务包含书籍的伴随书评的评级信息.

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。




该应用程序是多语言的，即微服务是用不同的语言编写的。

# 启动应用程序

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：

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

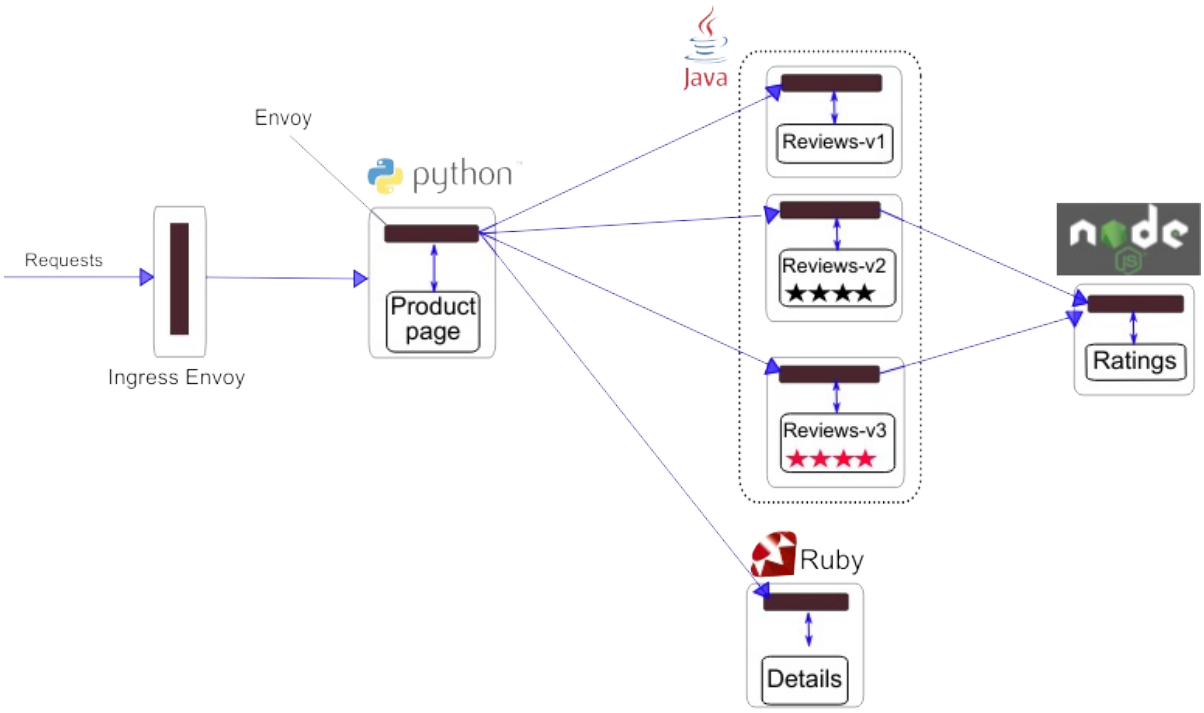
上述命令启动四个微服务器并创建网关入口资源，如下图所示。

reviews 微服务有3个版本：v1，v2和v3。

> **info::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。

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。

因此，所有的微型服务器现在都和一个能够管理呼入和呼出调用的 Envoy sidecar。更新后的图表如下所示：




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
istio-ingress              10.0.0.122   <pending>     80:31565/TCP         8m
istio-pilot                10.0.0.189   <none>        8080/TCP             8m
istio-mixer                10.0.0.132   <none>        9091/TCP,42422/TCP   8m
kubernetes                 10.0.0.1     <none>        443/TCP              14d
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     STATUS    RESTARTS   AGE
details-v1-1520924117-48z17              2/2       Running   0          6m
istio-ingress-3181829929-xrrk5           1/1       Running   0          8m
istio-pilot-175173354-d6jm7              2/2       Running   0          8m
istio-mixer-3883863574-jt09j             2/2       Running   0          8m
productpage-v1-560495357-jk1lz           2/2       Running   0          6m
ratings-v1-734492171-rnr5l               2/2       Running   0          6m
reviews-v1-874083890-f0qf0               2/2       Running   0          6m
reviews-v2-1343845940-b34q5              2/2       Running   0          6m
reviews-v3-1813607990-8ch52              2/2       Running   0          6m
```

4. 确定网关入口URL：

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

```
NAME      HOSTS     ADDRESS                 PORTS     AGE
gateway   *         130.211.10.121          80        1d
```

如果您的 Kubernetes 集群在支持外部负载均衡器的环境中运行，并且Istio入口服务能够获取外部IP，则入站资源 ADDRESS 将等于入口服务外部IP。

```
export GATEWAY_URL=130.211.10.121:80
```

> 有时当服务无法获取外部IP时，入口 ADDRESS 可能会显示 NodePort 地址列表。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。例如，在GKE中，您可以使用以下命令创建防火墙规则：

```
gcloud compute firewall-rules create allow-book --allow tcp:$(kubectl get svc ist
io-ingress -o jsonpath='{.spec.ports[0].nodePort}')
```

如果您的部署环境不支持外部负载均衡器（例如 minikube），则 ADDRESS 字段将为空。在这种情况下，您可以使用 service NodePort：

```
export GATEWAY_URL=$(kubectl get po -l istio=ingress -o 'jsonpath={.items[0].stat
us.hostIP}'):$(kubectl get svc istio-ingress -o 'jsonpath={.spec.ports[0].nodePort
}')
```

5. 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}/productpage
```

```
200
```

# 清理

在完成 BookInfo 示例后，您可以卸载它，如下所示：

1. 删除路由规则并终止应用程序pod

```
samples/apps/bookinfo/cleanup.sh
```

2. 确认关机

```
istioctl get route-rules   #-- there should be no more routing rules
kubectl get pods           #-- the BookInfo pods should be deleted
```

# 下一步

现在您已经启动并运行了 BookInfo 示例，您可以将浏览器指向 `http://$GATEWAY_URL/productpage` 来看正在运行的应用程序，并使用 Istio 来控制流量路由，注入故障，限速等。

要开始，请查看 请求路由任务。

# Tags