

Envoy 官方文档 中文版

v1.7

为云原生应用而设计，开源的边缘和服务代理



目录

说明

说明	1.1
关于本文档	1.2

简介

Envoy 是什么?	2.1
架构概览	2.2
术语	2.2.1
线程模型	2.2.2
监听器	2.2.3
监听器过滤器	2.2.4
网络 (L3/L4) 过滤器	2.2.5
HTTP 连接管理	2.2.6
HTTP 过滤器	2.2.7
HTTP 路由	2.2.8
gRPC	2.2.9
WebSocket 支持	2.2.10
集群管理器	2.2.11
服务发现	2.2.12
健康检查	2.2.13
连接池	2.2.14
负载均衡	2.2.15
异常点检测	2.2.16
断路	2.2.17
全局速率限制	2.2.18
TLS	2.2.19
统计	2.2.20
运行时配置	2.2.21
追踪	2.2.22
TCP 代理	2.2.23
访问记录	2.2.24
MongoDB	2.2.25
DynamoDB	2.2.26
Redis	2.2.27
热重启	2.2.28

动态配置	2.2.29
初始化	2.2.30
排除	2.2.31
脚本	2.2.32
部署类型	2.3
仅服务之间	2.3.1
服务之间外加前端代理	2.3.2
服务间、前端代理、双向代理	2.3.3
与类似系统比较	2.4
获取帮助	2.5
历史版本	2.6

入门指南

入门指南	3.1
Sandbox	3.2
前端代理	3.2.1
Zipkin 追踪	3.2.2
Jaeger 追踪	3.2.3
Jaeger 原生追踪	3.2.4
gRPC 网桥	3.3
其他用例	3.4
Envoy 作为 Kubernetes 的 API 网关	3.4.1

构建和安装

构建	4.1
构建 Envoy Docker 镜像	4.1.1
参考配置	4.2
工具	4.3
配置负载检查工具	4.3.1
路由表检查工具	4.3.2
Schema 验证器检查工具	4.3.3

配置参考

v1 API 概览	5.1
v2 API 概览	5.2
监听器	5.3
统计	5.3.1
运行时	5.3.2

监听器发现服务 (LDS)	5.3.3
监听器过滤器	5.4
原始目的地	5.4.1
TLS 检查器	5.4.2
网络过滤器	5.5
客户端 TLS 身份验证	5.5.1
回写	5.5.2
Mongo 代理	5.5.3
速率限制	5.5.4
Redis 代理	5.5.5
TCP 代理	5.5.6
HTTP 连接管理器	5.6
路由匹配	5.6.1
流量转移/拆分	5.6.2
HTTP 标头操作	5.6.3
HTTP 标头清理	5.6.4
统计	5.6.5
运行时	5.6.6
路由发现服务 (RDS)	5.6.7
HTTP 过滤器	5.7
缓冲区	5.7.1
CORS	5.7.2
DynamoDB	5.7.3
故障注入	5.7.4
gRPC HTTP/1.1 桥接	5.7.5
gRPC-JSON 转码	5.7.6
gRPC-Web	5.7.7
Gzip	5.7.8
健康检查	5.7.9
IP 标签	5.7.10
Lua	5.7.11
速率限制	5.7.12
路由	5.7.13
Squash	5.7.14
集群管理器	5.8
统计	5.8.1
运行时	5.8.2
集群发现服务 (CDS)	5.8.3
健康检查	5.8.4
断路	5.8.5

健康检查器	5.9
Redis	5.9.1
访问记录	5.10
速率限制服务	5.11
运行时	5.12
统计	5.13
路由表检查工具	5.14

运维管理

命令行选项	6.1
热重启 Python 包装器	6.2
管理接口	6.3
统计概览	6.4
运行时	6.5
文件系统标志	6.6
流量捕获	6.7
为自定义用例扩展 Envoy	6.8

API 参考

v1 API 参考	7.1
v2 API 参考	7.2

常见问题

Envoy 有多快?	8.1
从哪里能获得二进制文件?	8.2
如何设置 SNI?	8.3
如何设置 zone 可感知路由?	8.4
如何设置 Zipkin 追踪?	8.5
我设置了健康检查，但是当有节点失败时，Envoy 又路由到那些节点，这是怎么回事?	8.6
为什么 Round Robin 负载均衡看起来不起作用?	8.7

Envoy 官方文档中文版

Envoy 官方文档中文版，基于 Envoy 最新的 1.7 版本。

本项目文档地址：<https://github.com/servicemesher/envoy>

英文官方文档地址：<https://www.envoyproxy.io/docs/envoy/latest>

在线阅读地址：<https://servicemesher.github.io/envoy/>

本文档使用 *Gitbook* 生成。

参与

参与翻译请先阅读[翻译规范](#)。

如果你发现文中的纰漏与错误请提交 [Issue](#)。

致谢

感谢所有为本文档作出贡献的 Service Mesh 爱好者。

[查看贡献者名单](#)

封面图片：杭州中河路西湖大道立交桥 by [Jimmy Song](#)

注意

本书中不包含官方文档中的 **v1 API 参考** 和 **v2 API 参考** 部分，书中凡是指向 API 参考的链接都直接跳转到官方页面。

ServiceMesher 微信公众号



Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 21:43:56

关于本文档

Envoy 文档由以下几个主要部分组成：

- 简介：本部分介绍 Envoy 的概况、体系结构概述、典型部署方式等。
- 入门：使用 Docker 快速开始使用 Envoy。
- 安装：如何使用 Docker 构建/安装 Envoy。
- 配置：遗留 v1 API 和新 v2 API 共同的详细配置指令。相关时，配置指南还包含有关统计信息、运行时配置和 API 的信息。
- 操作：有关如何操作 Envoy 的常规信息，包括命令行界面、热重启包装器、管理界面、常规统计概览等。
- 扩展 Envoy：有关如何为 Envoy 编写自定义过滤器的信息。
- v1 API 参考：特定于遗留 v1 API 的配置详细信息。
- v2 API 参考：特定于新 v2 API 的配置细节。
- Envoy 常见问题：有疑问？希望我们的答案能让您满意。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-16 17:43:44

Envoy 是什么?

Envoy 是专为大型现代 SOA (面向服务架构) 架构设计的 L7 代理和通信总线。该项目源于以下理念:

网络对应用程序来说应该是透明的。当网络和应用程序出现问题时，应该很容易确定问题的根源。

实际上，实现上述的目标是非常困难的。为了做到这一点，Envoy 提供了以下高级功能:

进程外架构: Envoy 是一个独立进程，设计为伴随每个应用程序服务运行。所有的 Envoy 形成一个透明的通信网格，每个应用程序发送消息到本地主机或从本地主机接收消息，但不知道网络拓扑。在服务间通信的场景下，进程外架构与传统的代码库方式相比，具有两大优点：

- Envoy 可以使用任何应用程序语言。Envoy 部署可以在 Java、C++、Go、PHP、Python 等之间形成一个网格。面向服务架构使用多个应用程序框架和语言的趋势越来越普遍。Envoy 透明地弥合了它们之间的差异。
- 任何做过大型面向服务架构的人都知道，升级部署库可能会非常痛苦。Envoy 可以透明地在整个基础架构上快速部署和升级。

现代 C++11 代码库: Envoy 是用 C++11 编写的。之所以选择（系统）原生代码是因为我们认为像 Envoy 这样的基础架构组件应该尽可能避让（资源争用）。由于在共享云环境中部署以及使用了非常有生产力但不是特别高效的语言（如 PHP、Python、Ruby、Scala 等），现代应用程序开发人员已经难以找出延迟的原因。原生代码通常提供了优秀的延迟属性，不会对已混乱的系统增加额外负担。与用 C 编写的其他原生代码代理的解决方案不同，C++11 具有出色的开发生产力和性能。

L3/L4 filter 架构: Envoy 的核心是一个 L3/L4 网络代理。可插入 filter 链机制允许开发人员编写 filter 来执行不同的 TCP 代理任务并将其插入到主体服务中。现在已有很多用来支持各种任务的 filter，如原始 [TCP 代理](#)、[HTTP 代理](#)、[TLS 客户端证书认证](#) 等。

HTTP L7 filter 架构: HTTP 是现代应用程序体系结构的关键组件，Envoy 支持额外的 HTTP L7 filter 层。可以将 HTTP filter 插入执行不同任务的 HTTP 连接管理子系统，例如[缓存](#)，[速率限制](#)，[路由/转发](#)，嗅探 Amazon 的 [DynamoDB](#) 等等。

顶级 HTTP/2 支持: 当以 HTTP 模式运行时，Envoy 同时支持 HTTP/1.1 和 HTTP/2。Envoy 可以作为 HTTP/1.1 和 HTTP/2 之间的双向透明代理。这意味着它可以桥接 HTTP/1.1 和 HTTP/2 客户端以及目标服务器的任意组合。建议配置所有服务之间的 Envoy 使用 HTTP/2 来创建持久连接的网格，以便可以复用请求和响应。随着协议的逐步淘汰，Envoy 将不支持 SPDY。

HTTP L7 路由: 当以 HTTP 模式运行时，Envoy 支持一种[路由](#)子系统，能够根据路径、权限、内容类型、[运行时](#)及参数值等对请求进行路由和重定向。这项功能在将 Envoy 用作前端/边缘代理时非常有用，同时，在构建服务网格时也会使用此功能。

gRPC 支持: [gRPC](#) 是一个来自 Google 的 RPC 框架，它使用 HTTP/2 作为底层多路复用传输协议。Envoy 支持被 gRPC 请求和响应的作为路由和负载均衡底层的所有 HTTP/2 功能。这两个系统是非常互补的。

MongoDB L7 支持: [MongoDB](#) 是一种用于现代 Web 应用程序的流行数据库。Envoy 支持对 MongoDB 连接进行 L7 嗅探、统计和日志记录。

DynamoDB L7 支持: [DynamoDB](#) 是亚马逊的托管键/值 NOSQL 数据存储。Envoy 支持对 DynamoDB 连接进行 L7 嗅探和统计。

服务发现: [服务发现](#)是面向服务架构的关键组件。Envoy 通过一种[服务发现服务](#)（service discovery service）的方式支持多种服务发现方式，包括异步 DNS 解析和基于 REST 的查找。

健康检查: 推荐使用将服务发现视为最终一致的过程的方式来建立 Envoy 网格。Envoy 包含了一个[健康检查](#)子系统，可以选择对上游服务集群执行主动健康检查。然后，Envoy 联合使用服务发现和健康检查信息来确定健康的负载均衡目标。Envoy 还通过[异常检测](#)子系统支持被动健康检查。

高级负载均衡：负载均衡是分布式系统中不同组件之间的一个复杂问题。由于 Envoy 是一个独立代理而不是库，因此可以独立实现高级负载均衡以供任何应用程序访问。目前，Envoy 支持[自动重试](#)、[熔断](#)、通过外部速率限制服务的[全局速率限制](#)、[请求映射](#)和[异常点检测](#)。未来还计划支持请求竞争。

前端/边缘代理支持：尽管 Envoy 主要设计用来作为一个服务间的通信系统，但在系统边缘使用相同的软件也是大有好处的（可观察性、管理、相同的服务发现和负载均衡算法等）。Envoy 包含足够多的功能，使其可作为大多数现代 Web 应用程序的边缘代理。这包括[TLS 终止](#)、HTTP/1.1 和 HTTP/2 支持，以及 HTTP L7 路由。

最佳的可观察性：如上所述，Envoy 的主要目标是使网络透明。但是，问题在网络层面和应用层面都可能会出现。Envoy 包含对所有子系统强大的[统计功能支持](#)。目前支持[statsd](#)（和兼容的提供程序）作为统计信息接收器，但是插入不同的接收器并不困难。统计信息也可以通过[管理端口](#)查看。Envoy 还通过第三方提供商支持分布式[追踪](#)。

动态配置：Envoy 可以选择使用[动态配置 API](#) 的分层集合。如果需要，实现者可以使用这些 API 来构建复杂的集中管理部署。

设计目标

关于 Envoy 本身设计目标的简短说明：尽管 Envoy 绝对不慢（我们用了大量的时间来优化某些快速路径），我们的代码是按照模块化和易于测试的方式来编写的，而不是最大限度地实现最佳性能。我们认为这会更有效的利用时间，因为 Envoy 通常会和比它自身慢数倍、内存占用高数倍的语言和运行时部署在一起。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-28 17:20:09

术语

在深入主架构文档之前的一些定义。部分定义在行业中略有争议，下面将展开在 Envoy 文档和代码库中如何使用它们。

Host/主机: 能够进行网络通信的实体（如移动设备、服务器上的应用程序）。在此文档中，主机是逻辑网络应用程序。一块物理硬件上可能运行有多个主机，只要它们是可以独立寻址的。

Downstream/下游: 下游主机连接到 Envoy，发送请求并接收响应。

Upstream/上游: 上游主机接收来自 Envoy 的连接和请求，并返回响应。

Listener/监听器: 监听器是命名网地址（例如，端口、unix domain socket等），可以被下游客户端连接。Envoy 暴露一个或者多个监听器给下游主机连接。

Cluster/集群: 集群是指 Envoy 连接到的逻辑上相同的一组上游主机。Envoy 通过[服务发现](#)来发现集群的成员。可以选择通过[主动健康检查](#)来确定集群成员的健康状态。Envoy 通过[负载均衡策略](#)决定将请求路由到哪个集群成员。

Mesh/网格: 一组主机，协调好以提供一致的网络拓扑。在本文档中，“Envoy mesh”是一组 Envoy 代理，它们构成了分布式系统的消息传递基础，这个分布式系统由很多不同服务和应用程序平台组成。

Runtime configuration/运行时配置: 外置实时配置系统，和 Envoy 一起部署。可以更改配置设置，影响操作，而无需重启 Envoy 或更改主要配置。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-17 15:18:23

线程模型

Envoy 使用单进程 - 多线程的架构模型。一个 *master* 线程管理各种琐碎的任务，而一些 *worker* 线程则负责执行监听、过滤和转发。当监听器接收到一个连接请求时，该连接将其生命周期绑定到一个单独的 *worker* 线程。这使得 Envoy 主要使用大量单线程（embarrassingly parallel）处理工作，并且只有少量的复杂代码用于实现 *worker* 线程之间的协调工作。通常情况下，Envoy 实现了100%的非阻塞。对于大多数工作负载，我们建议将 *worker* 线程数配置为物理机器的线程数。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-18 17:26:22

监听器

Envoy 配置支持在单个进程中启用任意数量的监听器。通常建议每台机器运行一个 Envoy，而不必介意配置的监听器数量。这样运维更简单，而且只有单个统计来源。目前 Envoy 只支持 TCP 监听器。

每个监听器都独立配置有一些网络级别（L3/L4）的[过滤器](#)。当监听器接收到新连接时，配置好的连接本地过滤器将被实例化，并开始处理后续事件。通用监听器架构用于执行绝大多数不同的代理任务（例如，[限速](#)、[TLS客户端认证](#)、[HTTP 连接管理](#)、MongoDB [sniffing](#)、原始 [TCP 代理](#)等）。

监听器也可以选择性的配置某些[监听器过滤器](#)。这些过滤器的处理在网络过滤器之前进行，并有机会操纵连接元数据，通常会影响后续过滤器或集群处理连接的方式。

监听器也可以通过[监听器发现服务 \(LDS\)](#)动态获取。

[监听器配置](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-28 17:51:31

监听器过滤器

在[监听器](#)一节中讨论到，监听器过滤器可以用于操纵连接元数据。监听器过滤器的主要目的是来更方便地添加系统集成功能，而无需更改 Envoy 核心功能，并且还可以让多个此类功能之间的交互更加明确。

监听器过滤器的API相对简单，因为最终这些过滤器都在新接收的套接字上操作。可停止链中的过滤器并继续执行后续的过滤器。这允许去运作更复杂的业务场景，例如调用[限速服务](#)等。Envoy 包含多个监听器过滤器，这些过滤器在此架构概述以及[配置参考](#)中都有记录。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 10:25:39

网络 (L3/L4) 过滤器

如[监听器](#)一节所述，网络 (L3/L4) 过滤器构成 Envoy 连接处理的核心。过滤器 API 允许混合不同的过滤器组合，并匹配和附加到给定的监听器。有三种不同类型的网络过滤器：

- **读**: 当 Envoy 从下游连接接收数据时，调用读过滤器。
- **写**: 当 Envoy 要发送数据到下游连接时，调用写过滤器。
- **读/写**: 当 Envoy 从下游连接接收数据和要发送数据到下游连接时，调用读/写过滤器。

网络级过滤器的 API 相对简单，因为最终过滤器只操作原始字节和少量连接事件（例如，TLS 握手完成、连接在本地或远程断开等）。可停止链中的过滤器并继续执行后续的过滤器。这允许去应对更复杂的业务场景，例如调用[限速服务](#)等。Envoy 包含多个网络过滤器，这些过滤器在此架构概述和[配置参考](#)中都有说明。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 23:18:54

HTTP 连接管理

HTTP 是现代面向服务架构中非常关键的组件，因此 Envoy 实现了大量 HTTP 特有的功能点。Envoy 有一个内置的网络过滤器名为 [HTTP 连接管理器](#)。该管理器将原始字节翻译为 HTTP 消息和事件。(即，收到的标头、收到的正文数据、收到的标尾等等)。它还处理对所有 HTTP 连接以及请求的共性功能，如[访问日志](#)、[请求 ID 生成和追踪](#)、[请求/响应头操控](#)、[路由表](#)管理以及[统计](#)。

HTTP 连接管理器[配置](#)。

HTTP 协议

Envoy 的 HTTP 连接管理器内置支持 HTTP/1.1、WebSockets 和 HTTP/2，不支持 SPDY。Envoy 的 HTTP 支持首先被设计为一个 HTTP/2 多路复用代理。在内部，HTTP/2 这一名词用于描述系统元素。例如，HTTP 请求和响应发生在流上。编解码器 API 用于将不同的连线协议转换为针对流、请求、响应等不可知的形式。对于 HTTP/1.1，编解码器将协议的串行/流水线能力转换为更高层的 HTTP/2。这意味着大部分代码不需要了解流是源于 HTTP/1.1 还是 HTTP/2 连接。

HTTP 头净化

出于安全原因，HTTP 连接管理器执行不同的[头净化](#)操作。

路由表配置

每个 [HTTP 连接管理器过滤器](#)都有一个相关的[路由表](#)。路由表可以用以下任意一种方式指定：

- 静态设置。
- 通过 [RDS API](#) 动态地设置。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 18:14:11

HTTP 过滤器

与[网络级别过滤器](#)栈类似，Envoy 在连接管理器内支持 HTTP 过滤器栈。

我们可以在不了解底层物理协议（如 HTTP/1.1、HTTP/2 等）或多路复用技术相关知识的情况下，写过滤器去操作 HTTP 的信息。

有三种类型的 HTTP 过滤器：

- **解码器**：解码器过滤器当连接管理器在解码部分请求流（头部、正文和尾部）时被调用。
- **编码器**：编码器过滤器当连接管理器即将对响应流(头部、正文和尾部) 的部分进行编码时被调用。
- **解码器/编码器**：解码器/编码器过滤器当连接管理器在解码请求流时以及当连接管理器将要编码响应流时被调用。

HTTP 过滤器的 API 允许过滤器在不知道底层协议的情况下运行。就如网络过滤器一样，可停止 HTTP 过滤器并继续执行后续的过滤器。由此可以实现更复杂的业务场景，例如运行状况检查处理、调用限速服务、缓冲、路由、为应用程序流量（例如 DynamoDB 等）生成统计数据。Envoy 已包含多个 HTTP 过滤器，相关文档可查阅这份体系结构概述以及[配置参考](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 19:28:56

HTTP路由

Envoy 包括一个 HTTP 路由器过滤器，可以安装它来执行高级路由任务。这对于处理边缘流量（传统的反向代理请求处理）以及为服务 Envoy 网格构建服务（通常是通过主机/授权 HTTP 头的路由到达特定的上游服务集群）非常有用。

Envoy 也可以被配置为转发代理。在正向代理配置中，网格客户端可以通过适当地配置他们的 http 代理来成为 Envoy。路由在一个高层级接受一个传入的 HTTP 请求，将其与上游集群相匹配，在上游集群中获得一个连接池，并转发请求。路由过滤器支持以下特性：

- 将域/授权映射到一组路由规则的虚拟主机。
- 前缀和精确路径匹配规则（包括大小写敏感和大小写不敏感）。目前还不支持正则/ slug 匹配，这主要是因为难以用编程方式确定路由规则是否相互冲突。因此我们并不建议在反向代理级别使用正则/ slug 路由，但是我们将来可能会根据用户需求量增加对这个特性的支持。
- 在虚拟主机级别上的 [TLS 重定向](#)。
- 在路由级别的[路径/主机重定向](#)。
- 在路由级别的直接（非代理）HTTP 响应。
- 显式重写。
- [自动主机重写](#)，基于所选的上游主机的 DNS 名称。
- 前缀重写。
- 在路由级别上的 [Websocket升级](#)。
- 请求通过 HTTP 头或路由配置指定[请求重试](#)。
- 请求通过 [HTTP头](#) 或[路径配置](#)指定超时。
- 通过[运行时](#)从一个上游集群转移到另一个集群（参见[流量转移/分割](#)）。
- 使用基于[权重/百分比的路由](#)的流量跨越多个上游集群（参见[流量转移/分割](#)）。
- 任意标题匹配[路由规则](#)。
- 虚拟集群规范。虚拟集群是在虚拟主机级别上指定的，由 Envoy 使用，在标准集群级别上生成额外的统计信息。虚拟集群可以使用正则表达式匹配。
- 基于[优先级](#)的路由。
- 基于[哈希策略](#)的路由。
- [绝对 url](#) 支持非 tls 转发代理。

路由表

HTTP 连接管理器的配置拥有所有配置的 HTTP 过滤器所使用的[路由表](#)。尽管路由器过滤器是路由表的主要使用者，但是其他过滤器也有访问权限，以防它们想根据请求的最终目的地做出决策。例如，内置的限速过滤器查询路由表，以确定是否应该根据路由调用全局限速服务。连接管理器确保对特定请求的所有调用都是稳定的，即使决策涉及到随机性（例如，在运行时配置路由规则的情况下）。

重试语义

Envoy 允许在[路由配置](#)和通过[请求头](#)的特定请求中配置重试。以下配置是可能的：

- **最大数量的重试:** Envoy 将继续进行多次尝试。在每次重试之间使用一个指数回溯算法。此外，所有重试都包含在总体请求超时中。这避免了由于大量重试而导致的长时间请求。
- **重试条件:** Envoy 可以根据应用程序的要求对不同类型的条件进行重试。例如，网络故障，所有5xx响应代码，幂等4xx响应码，等等。

请注意，根据 [x-envoy-overloaded](#) 的内容，重试可能会被禁用。

优先级路由

Envoy 支持[路由](#)级别的优先路由。当前的优先级实现为每个优先级使用不同的[连接池](#)和[断路设置](#)。这意味着，即使对于 HTTP/2 请求，也将使用两个物理连接到上游主机。在未来，Envoy 可能会通过单一连接支持真正的 HTTP/2 优先级。

当前支持的优先级是默认的和高的。

直接响应

Envoy 支持发送“直接”回应。这些是预先配置的 HTTP 响应，不需要代理到上游服务器。

有两种方法可以在一条路由中指定直接响应：

- 设置 `direct_response` 字段。这适用于所有 HTTP 响应状态。
- 设置[重定向](#)。这适用于重定向响应状态，但它简化了位置头的设置。

直接响应有一个 HTTP 状态码和一个可选体。路由配置可以内联地指定响应体，或者指定包含体的文件的路径名。如果路线配置指定了一个文件路径名，Envoy 将在配置负载上读取文件并缓存内容。

注意

如果指定了响应体，那么它的大小必须不超过 4KB，不管它是内联还是在文件中。Envoy 目前将所有正文保存在内存中，所以 4KB 的限制本意上是为了防止代理的内存占用太大。

如果路由或相关虚拟主机上设置了 `response_headers_to_add`，Envoy 将在直接 HTTP 响应中包含指定的标头。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 10:44:03

gRPC

gRpc 是来自 Google 的 RPC 框架。它使用协议缓冲区作为底层 序列化 /IDL(接口描述语言的缩写) 格式。在传输层，它使用 HTTP/2 进行请求/响应复用。Envoy 在传输层和应用层都提供对 gRPC 的一流支持：

- gRPC 使用 HTTP/2 trailers 特性 (可以在 HTTP 请求和响应报文后追加 HTTP Header) 来传送请求状态。Envoy 是能够正确支持 HTTP/2 trailers 的少数几个 HTTP 代理之一，因此也是可以传输 gRPC 请求和响应的代理之一。
- 某些语言的 gRPC 运行时相对不成熟。Envoy 支持 gRPC 桥接过滤器，允许 gRPC 请求通过 HTTP/1.1 发送给 Envoy。然后，Envoy 将请求转换为 HTTP/2 以传输到目标服务器。该响应被转换回 HTTP/1.1。
- 安装后，除了标准的全局 HTTP 统计数据之外，桥接过滤器还会根据每个 RPC 统计数据进行收集。
- gRPC-Web 由一个指定的过滤器支持，该过滤器允许 gRPC-Web 客户端通过 HTTP/1.1 向 Envoy 发送请求并代理到 gRPC 服务器。目前相关团队正在积极开发中，预计它将成为 gRPC 桥接过滤器的后续产品。
- gRPC-JSON 转码器由一个指定的过滤器支持，该过滤器允许 RESTful JSON API 客户端通过 HTTP 向 Envoy 发送请求并获取代理到 gRPC 服务。

gRPC 服务

除了在数据层面上代理 gRPC 外，Envoy 在控制层面也使用了 gRPC，它从中[获取管理服务器的配置](#)以及过滤器中的配置，例如用于[速率限制](#)或授权检查。我们称之为 *gRPC 服务*。

当指定 gRPC 服务时，必须指定使用 [Envoy gRPC 客户端](#)或 [Google C ++ gRPC 客户端](#)。我们在下面的这个选择中讨论权衡。

Envoy gRPC 客户端是使用 Envoy 的 HTTP/2 上行连接管理的 gRPC 的最小自定义实现。服务被指定为常规 Envoy 集群，定期处理超时、重试、终端发现、负载平衡、故障转移、负载报告、断路、健康检查、异常检测。它们与 Envoy 的数据层面共享相同的连接池机制。同样，集群统计信息可用于 gRPC 服务。由于客户端是简化版的 gRPC 实现，因此不包括诸如 OAuth2 或 gRPC-LB 之类的高级 gRPC 功能后备。

Google C++ gRPC 客户端基于 Google 在 <https://github.com/grpc/grpc> 上提供的 gRPC 参考实现。它提供了 Envoy gRPC 客户端中缺少的高级 gRPC 功能。Google C++ gRPC 客户端独立于 Envoy 的集群管理，执行自己的负载平衡、重试、超时、端点管理等。Google C++ gRPC 客户端还支持[自定义身份验证插件](#)。

在大多数情况下，当你不需要 Google C++ gRPC 客户端的高级功能时，建议使用 Envoy gRPC 客户端。这使得配置和监控更加简单。如果 Envoy gRPC 客户端中缺少你所需要的功能，则应该使用 Google C++ gRPC 客户端。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 17:55:48

WebSocket 支持

Envoy 支持将 HTTP/1.1 连接升级到 WebSocket 连接。仅当下游客户端发送正确的升级请求头，并且被匹配的 HTTP 路由必须明确配置使用了 WebSocket ([use_websocket](#)) 时才允许连接升级。如果请求到达 WebSocket 的路由没有必要的升级头，它将被视为常规的 HTTP/1.1 请求。

由于 Envoy 将 WebSocket 连接视为纯 TCP 连接，因此它支持 WebSocket 协议的所有内容，而与它们的报文格式无关。WebSocket 路由不支持某些 HTTP 请求级别的功能，例如重定向、超时、重试、速率限制和阴影。但是，支持前缀重、显式和自动主机重写、流量转移和拆分。

连接语义

尽管 WebSocket 升级可以通过 HTTP/1.1 连接进行，但 WebSockets 代理的工作模式与普通 TCP 代理类似，即 Envoy 不会解析 websocket 报文帧。下游客户端和/或上游服务器负责终止 WebSocket 连接（例如，通过发送[关闭帧](#)）和底层 TCP 连接。

当连接管理器通过支持 WebSocket 的路由接收到 WebSocket 升级请求时，它通过 TCP 连接将请求转发给上游服务器。Envoy 不知道上游服务器是否拒绝了升级请求。上游服务器负责终止 TCP 连接，这会导致 Envoy 终止相应的下游客户端连接。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 10:44:51

集群管理器

Envoy 集群管理器管理所有配置的上游集群。正如 Envoy 配置可以包含任意数量的监听器一样，配置也可以包含任意数量的独立配置的上游集群。

上游集群和主机是从网络/HTTP 过滤器堆栈中抽象而来，因为上游集群和主机可用于任意数量的不同代理任务。集群管理器向过滤器堆栈暴露 API，允许过滤器获得连接到上游集群的 L3/L4 连接，或者连接到上游集群的抽象 HTTP 连接池的句柄（上游主机是否支持 HTTP/1.1 或 HTTP/2 是隐藏的）。过滤器阶段判断是否需要 L3/L4 连接或新的 HTTP 流，而集群管理器处理所有的复杂性，包括获知哪些主机可用并且健康、负载均衡、上游连接数据的线程本地存储（因为大多数 Envoy 代码以单线程编写）、上游连接类型（TCP/IP、UDS）、适用的上游协议（HTTP/1.1、HTTP/2）等。

集群管理器获知集群的方式可以是静态配置，也可以通过集群发现服务（CDS）API 动态获取。动态集群获取允许将更多配置存储在中央配置服务器中，这样就可以减少重启 Envoy 和重新分配配置的次数。

- 集群管理器 [配置](#)。
- CDS [配置](#)。

集群热身

当集群在服务器启动或者通过 CDS 进行初始化时，它们会“热身”。这意味着集群在下列操作发生之前不可用。

- 初始服务发现加载 (例如，DNS 解析、EDS 更新等等)。
- 初始主动[健康检查](#)通过，如果配置了主动健康检查。Envoy 将发送健康检查请求到每个被发现的主机上，以此来判断它的初始健康状态。

以上几项能够确保 Envoy 在开始将集群用于流量服务之前具有准确的集群视图。

在讨论集群热身时，集群“变为可用”意味着：

- 对于新加入的集群，在集群热身之前对于 Envoy 的其余部分来说是不可见的。即引用集群的 HTTP 路由将导致 404 或 503 (取决于配置)。
- 对于更新后的集群，旧集群将继续存在并服务流量。当新集群被加热后，它将与旧集群进行原子交换，从而不会发生流量中断。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-28 19:46:25

服务发现

当上游集群在[配置](#)中定义时，Envoy 需要知道如何解析集群中的成员。这被称为服务发现。

支持的服务发现类型

静态类型

静态类型是最简单的服务发现类型。配置中会明确指定每个上游主机的已解析网络名称（IP 地址、端口、unix 域套接字等）。

严格的 DNS 类型

当使用严格的 DNS 服务发现时，Envoy 将持续并异步地解析指定的 DNS 目标。DNS 结果中的每个返回的 IP 地址将被视为上游集群中的显式主机。这意味着如果查询返回三个 IP 地址，Envoy 将假定该集群有三台主机，并且所有三台主机应该负载均衡。如果有主机从结果中删除，则 Envoy 会认为它不再存在，并且会将它从所有的当前连接池中排除。请注意，Envoy 从不同步解析转发路径中的 DNS。基于最终一致性的策略，永远不用担心长时间运行的 DNS 查询会受到阻塞。

逻辑 DNS 类型

逻辑 DNS 使用类似异步解析机制来遵循严格的 DNS 策略。但是，这种策略并非严格基于 DNS 的查询结果，逻辑 DNS 集群假设它们构成了整个上游集群并在需要初始化新连接时仅使用返回的第一个 IP 地址。因此，单个逻辑连接池可能包含各种不同上游主机的物理连接。并且连接永远不会流失。这种服务发现类型适用于必须通过 DNS 访问的大型 Web 服务。此类服务通常使用循环 DNS 来返回许多不同的 IP 地址。通常每次查询返回不同的结果。如果在这种情况下使用了严格的 DNS 服务发现类型，Envoy 会假定集群的成员在每个解析间隔内都会发生变化，这将导致连接池枯竭，连接循环等。相反，使用逻辑 DNS，连接会一直保持活动状态直到它们循环。在与大型 Web 服务交互时，这几乎是所有可行策略中最好的方式：异步、最终一致的 DNS 解析、长连接、转发路径零阻塞。

原始目的地类型

当传入连接通过 iptables 重定向或 TPROXY（查看真实 IP）目标或使用代理协议重定向到 Envoy 时，可以使用原始目标集群。在这些情况下，使用元数据重定向的策略会使路由到原始目标集群的请求转发到上游主机，并且不需要任何明确的主机配置或上游主机发现机制。与上游主机的连接会被放入链接，并且闲置的主机在空闲时间比 `cleanup_interval_ms` 长（默认值为 5000 ms）时会被刷新。如果原始目标地址不可用，则不会打开上游连接。这种原始目的地服务发现的方式必须与原始目的地[负载均衡](#)一起使用。

服务发现服务 (SDS)

Envoy 通过一个发现服务的服务获取集群成员，它是一个[REST 风格的 API](#)。Lyft 提供了基于 Python 语言的[发现服务](#)参考实现。该实现使用 AWS DynamoDB 作为存储类型，但该 API 非常简单，可以轻松地在各种不同的存储类型之上实施。对于每个 SDS 集群，Envoy 将定期从发现服务中获取集群成员。由于以下几个原因，SDS 是首选的服务发现机制：

- Envoy 对每个上游主机都有明确的了解（与通过 DNS 解析的负载均衡进行路由相比而言），并可以做出更智能的负载均衡决策。
- 在每个主机的发现 API 响应中携带的额外属性通知 Envoy 负载均衡权重、金丝雀状态、区域等。这些附加属性在负载均衡、统计信息收集等过程中会被 Envoy 网格全局使用。

通常，主动健康检查与最终一致的服务发现服务数据结合使用，以进行负载均衡和路由决策。这将在下一节中进一步讨论。

关于服务发现的最终一致性

许多现有的 RPC 系统将服务发现视为完全一致的过程。为此，它们使用完全一致的领导选举算法，如 Zookeeper、etcd、Consul 等。我们的经验是，大规模使用这些策略是很痛苦的。

Envoy 从一开始就设计服务发现不需要完全一致性。相反的，Envoy 假定主机以最终一致的方式来回穿过网格。我们推荐的将服务部署到服务 Envoy 网格配置的方式是使用最终一致的服务发现以及主动运行[健康检查](#)（Envoy 显式地对上游集群成员进行健康检查）来确定集群运行状况。这种方式有许多好处：

- 所有健康决定都是完全分布的。因此，网络分区可以得到适当的处理（应用程序是否正常处理分区是另一回事）。
- 为上游集群配置运行状况检查时，Envoy 使用 2x2 矩阵来确定是否路由到主机：

发现状态	健康检查成功	健康检查失败
发现	路由	不要路由
未发现	路由	不用路由、删除

发现主机，健康检查成功

Envoy 将路由到目标主机

未发现主机，健康检查成功

Envoy 将路由到目标主机。这一点非常重要，因为我们在设计时假定发现服务随时可能失败。即使发现主机数据缺失，但健康检查通过，Envoy 仍会路由。虽然在这种情况下添加新主机是不可能的，但现有主机仍将正常运行。当发现服务再次正常运行时，数据将最终重新收集。

发现主机，健康检查失败

Envoy 不会路由到目标主机。我们假设健康检查数据比发现数据更准确。

未发现主机，健康检查失败

Envoy 不会路由并将删除目标主机。这是 Envoy 清除主机数据的唯一一种情况。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-28 20:52:46

健康检查

主动健康检查可以在每个上游集群的基础上进行[配置](#)。如[服务发现](#)部分所述，主动运行状况检查和 SDS 服务发现类型会同时进行。但是，即使使用其他服务发现类型，也有其他需要进行主动健康检查的情况。Envoy 支持三种不同类型的健康检查及各种设置（检查时间间隔、主机不健康标记为故障、主机健康时标记为成功等）：

- **HTTP**: 在 HTTP 健康检查期间，Envoy 将向上游主机发送 HTTP 请求。如果主机是健康的，会有 200 响应。如果上游主机想立即通知下游主机不再转发流量，则返回 503。
- **L3/L4**: 在 L3/L4 健康检查期间，Envoy 会向上游主机发送一个可配置的字节缓冲区。如果主机被认为是健康的，字节缓冲区在响应中会被显示出来。Envoy 还支持仅连接 L3/L4 健康检查。
- **Redis**: Envoy 将发送 Redis PING 命令并期望 PONG 响应。如果上游 Redis 服务器使用 PONG 以外的任何其他响应命令，则会导致健康检查失败。或者，Envoy 可以在用户指定的密钥上执行 EXISTS。如果密钥不存在，则认为它是合格的健康检查。这允许用户通过将指定的密钥设置为任意值来标记 Redis 实例以进行维护直至流量耗尽。请参阅 [redis_key](#)。

被动健康检查

Envoy 还支持通过[异常值检测](#)进行被动健康检查。

连接池交互

请参阅[此处](#)了解更多信息

HTTP 健康检查过滤器

当部署 Envoy 网格并在集群之间进行主动健康检查时，会生成大量健康检查流量。Envoy 包含一个HTTP 健康检查过滤器，可以安装在配置的 HTTP 倾听器中。这个过滤器有几种不同的操作模式：

- **不通过**: 在此模式下，运行状况检查请求永远不会传递给本地服务。Envoy 会根据服务器当前的耗尽状态以200或503响应。
- **从上游集群运行健康状况计算得出的不通过**: 在此模式下，运行健康检查筛选器将返回200或503，具体的值取决于在一个或多个上游集群中是否存在至少一个[指定百分比的服务器健康](#)。（但是，如果 Envoy 服务器处于排空状态，则无论上游集群运行状况如何，它都将以503响应。）
- **通过**: 在此模式下，Envoy 会将每个健康检查请求传递给本地服务。根据该服务的健康状态返回200或503。
- **通过缓存传递**: 在此模式下，Envoy 会将健康检查请求传递给本地服务，但会将结果缓存一段时间。如果在缓存有效期，在随后的健康检查请求会直接获取缓存的值。缓存到期后，下一个运行健康检查请求将传递给本地服务。操作大型网格时，这是推荐的操作模式。Envoy 会保存进行健康检查的连接，因此健康检查请求对 Envoy 本身的成本很低。因此，这种操作模式产生了每个上游主机的健康状态的最终一致的视图，而没有用大量的健康检查请求压倒本地服务

进一步阅读：

- [健康检查过滤器配置](#)。
- [/健康检查/失败](#)管理端点。
- [/健康检查/通过](#)管理员端点。

主动健康检查快速失败

当使用主动健康检查和被动健康检查（[异常检测](#)）时，通常使用较长的健康检查间隔来避免大量主动健康检查流量。在这种情况下，当时使用[/健康检查/失败](#)管理端点时，它对能够快速排除上游主机仍然很有用。为了支持这个，[路由过滤器](#)将响应 `x-envoy-immediate-health-check-fail` 头。如果此头由上游主机设置，Envoy 会立即将主机标记为主动运行状况检查失败。请注意，如果主机的集群已检查活动的健康才会出现这种情况[配置](#)。如果在 Envoy 已通过[/健康检查/失败](#) admin 端点标记为失败，健康检查过滤器会自动设置此标头。

健康检查身份

只验证上游主机是否响应特定的运行状况检查 URL 并不一定意味着上游主机有效。例如，当在自动缩放的云环境或容器环境中使用最终一致的服务发现时，被检查主机可能会消失，但是其他主机会以相同的 IP 地址返回。解决此问题的一个办法是针对每种服务类型都有不同的 HTTP 健康检查 URL。该方法的缺点是整体配置会变得更加复杂，因为每个健康检查 URL 都是完全自定义的。

Envoy HTTP 健康检查器支持 `service_name` 选项。如果设置了此选项，健康检查程序还会使用 `x-envoy-upstream-healthchecked-cluster` 响应标头的值与 `service_name` 进行比较。如果值不匹配，健康检查不通过。上游健康检查过滤器会将 `x-envoy-upstream-healthchecked-cluster` 附加到响应头。这个值由 `--service-cluster` 命令行选项决定。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-28 21:15:18

连接池

对于 HTTP 流量，Envoy 支持在底层协议（HTTP/1.1 或 HTTP/2）之上的抽象连接池。过滤器代码不需要知道底层协议是否支持真正的复用。在实践中，底层实现具有以下高级属性：

HTTP/1.1

HTTP/1.1 连接池根据需要获取上游主机的连接（取决于断路限制）。当连接变得可用时，请求被绑定到连接，这可能是因为连接完成先前请求的处理，或者因为新的连接已经准备好可以接收第一次请求。HTTP/1.1 连接池不使用流水线，因此如果上游连接被切断，只有一个下游请求必须重置。

HTTP/2

HTTP/2 连接池获取到上游主机的单个连接。所有请求都通过此连接复用。如果收到 GOAWAY 帧，或者连接达到最大流限制，连接池将创建新的连接并且耗尽现有连接。HTTP/2 是首选的通信协议，因为连接很少会被切断。

健康检查交互

如果 Envoy 配置有主动或被动[健康检查](#)，则代表从健康状态转换为不健康状态的主机将关闭所有连接池连接。如果主机重新进入负载均衡轮换，它将创建新的连接，这将最大化解决流量不佳（由于 ECMP 路由或其他原因）的机会。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-17 15:13:32

负载均衡

当过滤器需要获取到上游集群中主机的连接时，集群管理器将使用负载均衡策略来确定选择哪个主机。负载均衡策略是可拔插的，并且在[配置](#)中以每个上游集群为基础进行指定。请注意，如果没有为集群[配置](#)活动的运行状况检查策略，则所有上游集群成员都认为是健康的。

支持的负载均衡器

Round robin

这是一个简单的策略，每个健康的上游主机按循环顺序选择。如果将[权重](#)分配给本地的端点，则使用加权 round robin（循环）调度，其中较高权重的端点将更频繁地出现在循环中以实现有效权重。

加权最少请求

请求最少的负载均衡器使用 $O(1)$ 算法选择两个随机健康主机，并选择主动请求较少的主机（[研究](#)表明这种方法几乎与 $O(N)$ 全扫描一样好）。如果集群中的任何主机的负载均衡权重大于 1，则负载均衡器将转换为随机选择主机并使用该主机<权重>时间的模式。该算法对于负载测试来说简单且足够。它不应该用于需要真正的加权最小请求的地方（通常请求持续时间可变且较长）。我们可能会在将来添加一个真正的全扫描加权最小请求变体来涵盖此用例。

环哈希

Ring/modulo 哈希负载均衡器实现对上游主机的一致性哈希。该算法基于将所有主机映射到一个环上，使得从主机集添加或移除主机的更改仅影响 $1/N$ 个请求。这种技术通常也被称为“ketama”哈希。一致的哈希负载均衡器只有在使用指定哈希值的协议路由时才有效。最小环大小控制环中每个主机的复制因子。例如，如果最小环大小为 1024 并且有 16 个主机，则每个主机将被复制 64 次。环哈希负载均衡器当前不支持加权。

当使用基于优先级的负载均衡时，优先级也通过哈希选择，因此当后端集合稳定时，选定的端点仍将保持一致。

注意

环哈希负载均衡器不支持所在地加权负载均衡。

Maglev

Maglev（磁悬浮）负载均衡器对上游主机实施一致性的哈希。它使用[本文](#)第 3.4 节中描述的算法，固定表大小为 65537（参见同一论文的第 5.3 节）。Maglev 可以用作环哈希负载均衡器的替代品，可以在任何需要一致性哈希的地方使用。就像环哈希负载均衡器一样，只有在使用指定哈希值的协议路由时，一致性哈希负载均衡器才有效。

一般来说，与环形散列（“ketama”）算法相比，Maglev 具有快得多的查表编译时间以及主机选择时间（当使用 256K 条目的大环时大约分别为 10 倍和 5 倍）。Maglev 的缺点是它不像环哈希那样稳定。当主机被移除时，更多的键将移动位置（模拟显示键将移动大约两倍）。据说，对于包括 Redis 在内的许多应用程序来说，Maglev 很可能是环形哈希替代品的一大优势。高级读者可以使用这个 [benchmark](#) 来比较具有不同参数的环形哈希与 Maglev。

随机

随机负载均衡器选择一个随机的健康主机。如果没有配置健康检查策略，则随机负载均衡器通常比 round robin 更好。随机选择可以避免主机出现故障后对集合中的主机造成偏见。

原始目的地

这是一种专用的负载均衡器，只能与[原始目的集群](#)一起使用。上游主机是根据下游连接元数据选择的，即连接被打开到与连接重定向到 envoy 之前传入与连接的目标地址相同的地址。新的目标由负载均衡器按需添加到集群，并且集群会[定期](#)清除集群中未使用的主机。原始目标集群不能使用其他[负载均衡类型](#)。

恐慌阈值

在负载均衡期间，Envoy 通常只会考虑上游集群中健康的主机。但是，如果集群中健康主机的百分比变得过低，envoy 将忽视所有主机中的健康状况和均衡。这被称为恐慌阈值。缺省恐慌阈值是 50%。这可以通过运行时以及[集群配置](#)进行[配置](#)。恐慌阈值用于避免在负载增加时主机故障导致整个集群中级联故障的情况。

请注意，恐慌阈值是有优先级的。这意味着如果单个优先级中健康节点的百分比低于阈值，该优先级将进入恐慌模式。一般而言，不鼓励将恐慌阈值与优先级结合使用，因为当有足够的节点的状态不健康触发恐慌阈值时，大部分流量应该已经溢出到下一个优先级。

优先级划分

在负载均衡期间，Envoy 通常只考虑配置为最高优先级的主机。对于每个 EDS [LocalityLbEndpoints](#)，还可以指定一个可选的优先级。当最高优先级（P=0）的端点健康时，所有流量将以该优先级落在端点上。由于最高优先级的端点变得不健康，因此流量将开始慢慢降低优先级。

目前，假定每个优先级级别都由因子 1.4（硬编码）过度配置的。因此，如果 80% 的端点是健康的，那么优先级仍然被认为是健康的，因为 $80 * 1.4 > 100$ 。当健康端点的数量下降到 72% 以下时，优先级的健康状况低于 100。此时，相当于到 P=0 健康状态的流量的百分比将转到 P=1，剩余流量将流向 P=1。

假设一个简单的设置有 2 个优先级，P=1 100% 健康。

P=0 健康端点	到 P=0 的流量百分比	到 P=1 的流量百分比
100%	100%	0%
72%	100%	0%
71%	99%	1%
50%	70%	30%
25%	35%	65%
0%	0%	100%

如果 P=1 变得不健康，它将继续从 P=0 溢出负载，直到健康 P=0 + P=1 的总和低于 100 为止。此时，健康状态将被放大到 100% 的“有效”健康状态。

P=0 健康端点	P=1 健康端点	Traffic to P=0	到 P=1 的流量
100%	100%	100%	0%
72%	72%	100%	0%
71%	71%	99%	1%
50%	50%	70%	30%
25%	100%	35%	65%
25%	25%	50%	50%

随着更多的优先级被添加，每个级别消耗等于其“缩放”有效健康的负载，所以如果 P=0 + P=1 的组合健康度小于 100，则 P=2 将仅接收流量。

P=0 健康端点	P=1 健康端点	P=2 健康端点	到 P=0 的流量	到 P=1 的流量	到 P=2 的流量
100%	100%	100%	100%	0%	0%
72%	72%	100%	100%	0%	0%
71%	71%	100%	99%	1%	0%
50%	50%	100%	70%	30%	0%
25%	100%	100%	35%	65%	0%
25%	25%	100%	25%	25%	50%

在伪代码中加和：

```
load to P_0 = min(100, health(P_0) * 100 / total_health)
health(P_X) = 140 * healthy_P_X_backends / total_P_X_backends
total_health = min(100, Σ(health(P_0)...health(P_X)))
load to P_X = 100 - Σ(percent_load(P_0)..percent_load(P_X-1))
```

Zone 感知路由

我们使用以下术语：

- **始发/上游集群**: Envoy 将来自始发集群的请求路由到上游集群中。
- **本地 zone**: 包含始发和上游集群中主机子集的同一区域。
- **Zone 感知路由**: 尽量将请求路由到本地 zone 的上游集群主机上。

当始发和上游集群中的主机部署不同 zone 时，Envoy 执行 zone 感知路由。在执行 zone 感知路由之前有几个先决条件：

- 始发和上游集群都不处于恐慌模式。
- Zone 感知路由已启用。
- 始发集群与上游集群具有相同数量的 zone。
- 上游集群有足够的主机。浏览[此处](#)获取更多信息。

Zone 感知路由的目的是尽可能多地向上游集群的本地 zone 中发送流量，同时大致保持上游集群中的所有主机拥有相同的（取决于负载均衡策略）的每秒请求数量。

只要上游集群中每台主机的请求数量保持大致相同，Envoy 就会尝试尽可能多地将流量推送到本地上游区域。Envoy 路由到本地 zone 还是执行跨 zone 路由，这取决于本地 zone 中始发集群和上游集群中健康主机的百分比。关于始发和上游集群之间的本地 zone 百分比关系有以下两种情况：

- 始发集群的本地 zone 百分比大于上游集群中本地 zone 的百分比。在这种情况下，我们无法将来自始发集群本地 zone 的所有请求路由到上游集群的本地 zone，因为这会导致所有上游主机请求不均衡。相反，Envoy 计算可以直接路由到上游集群的本地 zone 的请求的百分比。其余的请求被路由到跨 zone。特定 zone 根据 zone 的剩余容量（该 zone 将获得一些本地 zone 流量并且可能具有 Envoy 可用于跨 zone 业务量的额外容量）来选择。
- 始发集群本地 zone 百分比小于上游集群中的本地 zone 百分比。在这种情况下，上游集群的本地 zone 可以获取来自始发集群本地 zone 的所有请求，并且还有一定空间允许来自集群中其他 zone 的流量（如果有必要的话）。

请注意，使用多个优先级时，zone 感知路由当前仅支持 P=0。

所在地加权负载均衡

另一种用于确定如何在不同 zone 和地理位置之间分配权重的方式是使用 [LocalityLbEndpoints](#) 消息中通过 EDS 提供的显式权重。这种方式与上述 zone 感知路由相互排斥，因为在所在地感知 LB 的情况下，我们通过管理服务器来提供所在地加权，而不是在 zone 感知路由中使用的 Envoy 侧启发式的方式。

当所有端点健康时，使用加权循环 round-robin 来挑选本地节点，其中地点权重用于加权。当某地的某些端点不健康时，我们通过调整地点的权重来反映这一点。与优先级一样，我们设置了一个过度提供因子（目前硬编码为 1.4），这意味着当一个地区只有少数端点不健康时，我们不会进行任何权重调整。

假设一个简单的设置，包含 2 个地点 X 和 Y，其中 X 的所在地权重为 1，Y 的所在地权重为 2， $L=Y$ 100% 健康。

$L=X$ 健康端点	到 $L=X$ 的流量百分比	到 $L=Y$ 的流量百分比
100%	33%	67%
70%	33%	67%
69%	32%	68%
50%	26%	74%
25%	15%	85%
0%	0%	100%

在伪代码中加和：

```
health(L_X) = 140 * healthy_X_backends / total_X_backends
effective_weight(L_X) = locality_weight_X * min(100, health(L_X))
load to L_X = effective_weight(L_X) / Σ_C(effective_weight(L_c))
```

请注意，在挑选优先级之后进行所在地加权选取。负载均衡器遵循以下步骤：

1. 挑选优先级别。
2. 从（1）中选择优先级别的所在地（如本节所述）。
3. 从（2）中选择使用集群指定的负载均衡器所在地范围内的端点。

通过在集群配置中设置 [locality_weighted_lb_config](#) 并通过 [load_balancing_weight](#) 在 [LocalityLbEndpoints](#) 中提供权重来配置所在地加权负载均衡。

此功能与负载均衡器子集设置不兼容，因为将个别子集的所在地级权重与显而易见的权重进行协调并不容易。

负载均衡器子集

根据附加在主机上的元数据将上游集群中的主机划分为子集，可以这样来配置 Envoy。然后，路由可以指定主机必须匹配的元数据，有了这些元数据负载均衡器才能选择路由，也可以选择回退到预定义的主机集（包括任何主机）。

子集使用集群指定的负载均衡器策略。原始目的地策略不能与子集一起使用，因为上游主机预先不知道这些策略。子集与 zone 感知路由兼容，但请注意，子集的使用可能很容易违反上述最小主机条件。

如果已配置的子集路由未指定元数据或没有匹配元数据的子集存在，则子集负载均衡器将启动其回退策略。默认策略是 `NO_ENDPOINT`，在这种情况下，请求失败，就好像该集群没有主机一样。相反，`ANY_ENDPOINT` 后备策略会在集群中的所有主机上进行负载均衡，而不考虑主机元数据。最后，`DEFAULT_SUBSET` 会导致回退到与特定元数据集匹配的主机之间进行负载均衡。

子集必须被预定义才能让子集负载均衡器有效地选择正确的主机子集。每个定义都是一组密钥，可以转换为零个或多个子集。从概念上讲，具有定义中所有键的元数据值的每个主机都会添加到其键-值对特定的子集。如果所有主机都不拥有密钥，那么定义不会产生子集。可以提供多个定义，并且如果单个主机与多个定义匹配，则可以在多个子集中出现。

在路由期间，路由的元数据匹配配置将用于查找特定的子集。如果存在具有路由指定的确切密钥和值的子集，则该子集将用于负载均衡。否则，使用回退策略。因此，集群的子集配置必须包含一个与给定路由具有相同密钥的定义才能实现子集负载均衡。

此功能只能在 V2 配置 API 中使用。此外，主机元数据仅在集群使用 EDS 发现类型时支持。子集负载均衡的主机元数据必须放在过滤器名称 “envoy.1b” 下。同样，路由元数据匹配条件使用 “envoy.1b” 过滤器名称。主机元数据可以是分层的（例如，顶级密钥的值可以是结构化值或列表），但子集负载均衡器仅比较顶级密钥和值。因此，在使用结构化值时，如果主机的元数据中出现相同的结构化值，则路由的匹配条件会匹配。

示例

我们将使用所有值都是字符串的简单单元数据。假设定义了以下主机并将其与集群关联：

主机	元数据
host1	v: 1.0, stage: prod
host2	v: 1.0, stage: prod
host3	v: 1.1, stage: canary
host4	v: 1.2-pre, stage: dev

集群可以像这样启用子集负载均衡：

```
---
name: cluster-name
type: EDS
eds_cluster_config:
  eds_config:
    path: '../eds.conf'
connect_timeout:
  seconds: 10
lb_policy: LEAST_REQUEST
lb_subset_config:
  fallback_policy: DEFAULT_SUBSET
  default_subset:
    stage: prod
  subset_selectors:
    - keys:
        - v
        - stage
    - keys:
        - stage
```

下表描述了一些路由及其对集群的应用结果。通常，匹配标准将与匹配请求的特定方面的路由一起使用，例如路径或 header 信息。

匹配标准	负载均衡位于	原因
stage: canary	host3	所选主机的子集
v: 1.2-pre, stage: dev	host4	所选主机的子集
v: 1.0	host1, host2	回退：没有仅具有“v”的子集选择器
other: x	host1, host2	回退：没有“other”子集选择器
(none)	host1, host2	回退：未请求子集

元数据匹配标准也可以在路由的加权集群上指定。来自选定加权集群的元数据匹配条件将与路由中的条件合并，并覆盖该原路由器中的条件：

路由匹配条件	加权集群匹配条件	最终匹配条件
stage: canary	stage: prod	stage: prod
v: 1.0	stage: prod	v: 1.0, stage: prod
v: 1.0, stage: prod	stage: canary	v: 1.0, stage: canary
v: 1.0, stage: prod	v: 1.1, stage: canary	v: 1.1, stage: canary
(none)	v: 1.0	v: 1.0
v: 1.0	(none)	v: 1.0

具有元数据的示例主机

具有主机元数据的EDS `LbEndpoint` :

```
---
endpoint:
  address:
    socket_address:
      protocol: TCP
      address: 127.0.0.1
      port_value: 8888
  metadata:
    filter_metadata:
      envoy.lb:
        version: '1.0'
        stage: 'prod'
```

具有元数据匹配标准的示例路由

具有元数据匹配标准的 RDS `Route` :

```
---
match:
  prefix: /
route:
  cluster: cluster-name
  metadata_match:
    filter_metadata:
      envoy.lb:
        version: '1.0'
        stage: 'prod'
```

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-28 18:37:48

异常点检测

异常点检测和驱逐是用来动态确定上游集群中是否有主机的表现不同于其他主机的过程，并将它们从健康[负载均衡](#)集中移除。性能可能沿着不同的轴，例如连续失败、一时的成功率下降、一时的延迟等。异常检测是被动健康检查的一种形式。Envoy 还支持[主动健康检查](#)。被动和主动健康检查既可以一起也可以独立使用，它们共同形成整体上游健康检查解决方案的基础。

驱逐算法

这取决于异常点检测的类型，驱逐或者以内联（例如在连续的 5xx 的情况下）或以指定的间隔（例如在定期成功率的情况下）运行。驱逐算法的工作原理如下：

1. 主机被确定为异常点。
2. 如果没有主机被驱逐，Envoy 会立即驱逐主机。否则，它会检查以确保驱逐主机的数量低于允许的阈值（通过 `outlier_detection.max_ejection_percent` 设置指定）。如果驱逐的主机数量超过阈值，则主机不会被驱逐。
3. 主机被驱逐需要几毫秒。主机被驱逐意味着主机被标记为不健康，并且在负载均衡期间不会被使用，除非负载均衡器处于恐慌情形中。毫秒数等于 `outlier_detection.base_ejection_time_msvalue` 乘以主机被驱逐的次数。这会导致主机被弹出需要越来越长的时间（如果它们继续失败）。
4. 被驱逐的主机将在弹出时间满足后自动重新投入使用。通常，异常检测与主动健康检查一起用于全面的健康检查解决方案。

检测类型

Envoy 支持以下的异常点检测类型：

连续的 5xx

如果上游主机返回一些连续的 5xx，它将被驱逐。请注意，在这种情况下，5xx 意味着实际的 5xx 响应代码，或者会导致 HTTP 路由器代表上游返回一个事件（重置、连接失败等）的事件。主机被弹出时所需的连续 5xx 数量由 `outlier_detection.consecutive_5xx` 值控制。

连续的网关故障

如果上游主机返回一些连续的“网关错误”（502、503 或 504 状态码），它将被驱逐。请注意，这包括会导致 HTTP 路由器代表上游返回其中一个状态码的事件（重置、连接失败等）。主机被驱逐所需的连续网关故障的数量由 `outlier_detection.consecutive_gateway_failure` 值控制。

成功率

基于成功率的异常值驱逐聚合来自集群中每个主机的成功率数据。然后以给定的间隔基于统计异常值检测来驱逐主机。如果主机在聚合时间间隔内的请求量小于 `outlier_detection.success_rate_request_volume` 值，则无法为主机计算成功率异常点驱逐。此外，如果某个时间间隔内所需的最小请求量的主机数小于 `outlier_detection.success_rate_minimum_hosts` 值，则不会对集群执行检测。

驱逐事件记录

Envoy 可以选择生成异常点驱逐事件日志。这在日常操作中非常有用，因为全局统计信息不能提供关于哪些主机正在被驱逐以及被驱逐的原因。日志使用 JSON 格式，每行一个对象：

```
{
  "time": "...",
  "secs_since_last_action": "...",
  "cluster": "...",
  "upstream_url": "...",
  "action": "...",
  "type": "...",
  "num_ejections": "...",
  "enforced": "...",
  "host_success_rate": "...",
  "cluster_success_rate_average": "...",
  "cluster_success_rate_ejection_threshold": "..."
}
```

- time

事件发生的时间。

- secs_since_last_action

自上次动作（被驱逐或未被驱逐）发生时到现的时间（以秒为单位）。由于在第一次驱逐之前没有动作，所以该值将为 `-1`。

- cluster

拥有被驱逐主机的 [集群](#)。

- upstream_url

被驱逐主机的 URL，例如 `tcp://1.2.3.4:80`。

- action

发生的动作。如果主机被驱逐，则为 `eject`；如果主机恢复服务，则为 `uneject`。

- type

如果 `action` 为 `eject`，`type` 指定发生的驱逐类型。当前类型可以是 `5xx`、`GatewayFailure` 或 `SuccessRate`。

- num_ejections

如果 `action` 为 `eject`，则指定主机被驱逐的次数（对于 Envoy 而言是本地的，并且如果主机由于任何原因从上游集群移除然后被重新添加则重置该值）。

- enforced

如果 `action` 为 `eject`，则指定驱逐是否被强制执行。`true` 表示主机被驱逐。`false` 表示事件已记录，但主机未被实际驱逐。

- host_success_rate

如果 `action` 为 `eject`，指定发生驱逐事件时主机在 `0-100` 范围内的成功率。

- cluster_success_rate_average

如果 `action` 为 `eject`，指定发生驱逐事件时集群中主机在 `0-100` 范围内的平均成功率。

- cluster_success_rate_ejection_threshold

如果 `action` 为 `eject` 并且 `type` 为 `SuccessRate`，指定发生驱逐事件时的成功率驱逐阈值。

参考配置

- 集群管理器[全局配置](#)
- 单集群[配置](#)
- 运行时[设置](#)
- 统计[参考](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 15:04:59

断路

断路（circuit breaking）是分布式系统的关键组件。尽快失败以及尽快地将压力回馈下游，基本上都会卓有成效。Envoy 网格的主要优点之一就是 Envoy 在网络级别强制实现断路，而不必为每个应用程序单独配置或编程。。Envoy 支持各种类型的完全分布式（非协调的）断路：

- **集群最大连接数**：Envoy 将为上游集群中的所有主机建立的最大连接数。实际上，这仅适用于 HTTP/1.1 集群，因为 HTTP/2 使用到每个主机的单个连接。
- **集群最大挂起请求数**：在等待就绪连接池连接时将排队的最大请求数。实际上，这仅适用于 HTTP/1.1 集群，因为 HTTP/2 连接池不会排队请求。HTTP/2 请求会立即复用。如果该断路器溢出，则集群的 `upstream_rq_pending_overflowcounter` 计数器将增加。
- **集群最大请求数**：在任何给定时间内，集群中所有主机可以处理的最大请求数。实际上，这适用于仅 HTTP/2 集群，因为 HTTP/1.1 集群由最大连接断路器控制。如果该断路器溢出，集群的 `upstream_rq_pending_overflow` 计数器将递增。
- **集群最大活动重试次数**：在任何给定时间内，集群中所有主机可以执行的最大重试次数。一般而言，我们建议积极进行断路重试，以便做零星故障重试而又不会爆炸式地增加整体重试数量从而引致大规模级联故障。如果该断路器溢出，集群的 `upstream_rq_retry_overflow` 计数器将递增。

断路可以根据每个上游集群和优先级进行[配置](#)和跟踪。这使得分布式系统的不同组件可以独立调整并具有不同的限制。

请注意，如果断路是源自于 HTTP 请求的情况下，路由过滤器将会设置 `x-envoy-overloaded` 标头。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-23 10:33:57

全局速率限制

在分布式系统中，多数情况下，分布式熔断对于吞吐的控制都是非常有效的；但是在有些例外情况下，是需要进行全局的速率控制的，例如大量主机（的请求）被转发给少量主机，并且平均请求延迟很低（比如访问数据库服务器的例子）。如果目标主机因故负载能力降低，下游主机就会影响到上游的集群。这种场景下，要不影响正常通信，又要防止主机故障引起的雪崩，想要为每个下游主机分别配置合适的断路器是很难的。这种案例就是使用全局速率控制的典型场景。

Envoy 直接集成了一个全局 gRPC 速率限制服务。所有实现了预定义 RPC/IDL 协议的服务都可以使用这一服务，Lyft 还提供了一个 [Go 编写的实现参考](#)，其中使用 Redis 作为后端。Envoy 的速率限制集成具有如下功能：

- 网络层的速率限制过滤器：安装了这一过滤器的 Envoy，当监听器上新建连接的时候，就会调用速率限制服务。配置中会指定一个特定的域和描述符来设置速率限制。这种方式对监听器上的每秒连接次数进行了限制，从而达到了在监听器中进行速率限制的效果。[配置参考](#)
- *HTTP* 级的速率限制过滤器：在安装了这一过滤器并且路由表要求调用全局速率限制服务的时候，Envoy 会在监听器的每次新请求的时候调用速率限制服务。所有到目标上游服务器、所有来自于源集群到目标集群的请求都可以进行速率限制。[配置参考](#)

[速率限制服务配置](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 11:29:04

TLS

Envoy 同时支持监听器中的 [TLS 终止](#)和与上游集群建立连接时的 [TLS 发起](#)。不管是为现代 web 服务提供标准的边缘代理功能，还是同具有高级 TLS 要求（TLS1.2, SNI, 等等）的外部服务建立连接，Envoy 都提供了充分的支持。Envoy 支持以下的 TLS 特性：

- **加密可配置**: 每个 TLS 监听器和客户端都可以指定它支持的加密算法。
- **客户端证书**: 除服务器证书验证外，上游/客户端连接还可以提供一个客户端证书。
- **证书验证和固定**: 证书验证选项包括基本的证书链验证、Subject 名称验证和哈希固定。
- **证书撤销**: 如果[提供了](#)证书撤销列表（CRL），Envoy 可以根据它检查对等证书。
- **ALPN**: TLS 监听器支持 ALPN. HTTP 连接管理器使用这个信息（以及协议接口）来确定客户端使用的是 HTTP/1.1 还是 HTTP/2.
- **SNI**: SNI 同时支持服务端（监听器）连接和客户端（上游）连接。
- **会话恢复**: 服务器连接支持通过 TLS 会话票据恢复之前的会话（参见[RFC 5077](#)）。可以在热重启时和并行 Envoy 实例之间执行恢复（通常在前端代理配置中很有用）。

底层实现

目前 Envoy 被编写为使用 [BoringSSL](#) 作为 TLS 提供者。

启用证书验证

除非验证上下文指定了一个或多个可信授权证书，否则上游和下游连接的证书验证都不会启用。

配置示例

```
static_resources:
  listeners:
    - name: listener_0
      address: { socket_address: { address: 127.0.0.1, port_value: 10000 } }
      filter_chains:
        - filters:
            - name: envoy.http_connection_manager
              # ...
        tls_context:
          common_tls_context:
            validation_context:
              trusted_ca:
                filename: /usr/local/my-client-ca.crt
  clusters:
    - name: some_service
      connect_timeout: 0.25s
      type: STATIC
      lb_policy: ROUND_ROBIN
      hosts: [{ socket_address: { address: 127.0.0.2, port_value: 1234 }}]
      tls_context:
        common_tls_context:
          validation_context:
            trusted_ca:
              filename: /etc/ssl/certs/ca-certificates.crt
```

`/etc/ssl/certs/ca-certificates.crt` 是 Debian 系统上 CA 包文件的默认路径。它使得 Envoy 在验证 `127.0.0.2:1234` 的服务器身份时，使用与标准安装的 Debian 系统上的 cURL 等相同的方式。Linux 和 BSD 系统上常见的系统 CA 包文件路径有

- `/etc/ssl/certs/ca-certificates.crt` (Debian/Ubuntu/Gentoo etc.)
- `/etc/pki/ca-trust/extracted/pem/tls-ca-bundle.pem` (CentOS/RHEL 7)
- `/etc/pki/tls/certs/ca-bundle.crt` (Fedora/RHEL 6)
- `/etc/ssl/ca-bundle.pem` (OpenSUSE)
- `/usr/local/etc/ssl/cert.pem` (FreeBSD)
- `/etc/ssl/cert.pem` (OpenBSD)

对于其他的 TLS 选项，请参见 [UpstreamTlsContexts](#) 和 [DownstreamTlsContexts](#) 的参考手册。

身份验证过滤器

Envoy 提供了一个网络过滤器，通过从 REST VPN 服务拉取的主体来进行 TLS 客户端身份验证。此过滤器将提供的客户端证书哈希与主体列表进行匹配，以确定是否允许连接。另外也可以配置一个可选的 IP 白名单。此功能可用于为 Web 基础架构构建边缘代理 VPN 支持。

这里是客户端 TSL 验证过滤器的[配置参考](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-23 21:24:35

统计

Envoy 的主要目标之一是使网络可以理解。Envoy 根据不同的配置收集了大量的统计数据。一般来说，统计数据分为以下三类：

- **Downstream**: 下游统计涉及传入的连接/请求。它们由侦听器、HTTP 连接管理器、TCP 代理过滤器等生成。
- **Upstream**: 上游统计涉及传出连接/请求。它们由连接池、路由器过滤器、TCP 代理过滤器等生成
- **Server**: 服务器统计信息描述了 Envoy 服务器实例的工作方式。服务器正常运行时间或分配内存量等统计信息在此处分类。

单个代理场景通常涉及下游和上游统计信息。这两种类型可用于获取该特定网络跃点的详细图。来自整个网格的统计数据给出了每一跳和整体网络健康状况的非常详细的图。发布的统计数据在操作指南中详细记录。

在 v1 API 中，Envoy 仅支持 statsd 作为统计输出格式。支持 TCP 和 UDP statsd。截至 v2 API，Envoy 有能力支持自定义、可插拔接收器。Envoy 中包含了一些标准接收器实现。有些接收器也支持使用标签/维度发布统计信息。

在 Envoy 和整个文档中，统计信息由规范的字符串表示来标识。这些字符串的动态部分被剥离成为标签。用户可以通过 Tag Specifier 配置此行为。

特使发出三种类型的值作为统计数据：

- **Counter (计数器)** : 无符号整数只会增加而不会减少。例如，总请求。
- **Gauge (量表)** : 无符号整数，既有增加也有减少。例如，当前有效的请求。
- **Histogram (直方图)** : 作为值流的一部分的无符号整数，然后由收集器进行汇总以最终生成汇总百分点值。例如，上游请求时间。

在内部，计数器和计量器被分批并定期冲洗以提高性能。直方图会在收到时写入。注意：以前称为定时器的内容已成为直方图，因为这两种表示法之间的唯一区别就是单位。

- [v1 API 参考](#)
- [v2 API 参考](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 11:09:39

运行时配置

Envoy 支持“运行时”配置（也称为“功能标志”和“决策者”）。更改配置设置后将立即影响运算，而无需重新启动 Envoy 或更改主配置。当前实现使用文件系统树。Envoy 监视配置目录中的符号链接交换，并在发生交换情况时重新加载树。这种类型的系统通常在大型分布式系统中部署。其他实现也并不难。可在操作指南的相关章节找到支持的运行时配置设置。使用默认的运行时值和“null”提供者，Envoy 也能正确运行，因此不需要存在一个类似体系用来支撑 Envoy 运行。

[运行时配置](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-25 18:57:14

追踪

概览

在较大的面向服务的架构中，分布式跟踪系统让开发者能够得到可视化的调用流程展示。在了解序列化、并行情况以及延迟分析的时候，这个功能至关重要。Envoy 用三个功能来支撑系统范围内的跟踪：

- 生成 *Request ID*: Envoy 会在需要的时候生成 UUID，并操作名为 [x-request-id] 的 HTTP Header。应用可以转发这个 Header 用于统一的记录和跟踪。
- 集成外部跟踪服务: Envoy 支持可插接的外部跟踪可视化服务。目前支持有 LightStep、Zipkin 或者 Zipkin 兼容的后端（比如说 Jaeger）。另外要添加其他的跟踪服务也并非难事。
- 客户端跟踪 *ID* 连接: `x-client-trace-id` Header 可以用来把不信任的请求 ID 连接到受信的 `x-request-id` Header 上。

如何初始化追踪

处理请求的 HTTP 连接管理器必须设置[跟踪](#)对象。有多种途径可以初始化跟踪：

- 外部客户端，使用 `x-client-trace-id` Header。
- 内部服务，使用 `x-envoy-force-trace` Header。
- 随机采样使用运行时设置：`random_sampling`

路由过滤器可以使用 `start_child_span` 来为 egress 调用创建下级 Span。

跟踪上下文信息的传播

Envoy 提供了报告网格内服务间通信跟踪信息的能力。然而一个调用流中，会有多个代理服务器生成的跟踪信息碎片，跟踪服务必须在出入站的请求中传播上下文信息，才能拼出完整的跟踪信息。

不管使用的是哪个跟踪服务，都应该传播 `x-request-id`，这样在被调用服务中启动相关性的记录。

为了理解 Span（本地的作业单元）之间的父子关系，跟踪服务自身也需要更多的上下文信息。这种需要可以通过在跟踪服务自身中直接使用 LightStep (OpenTracing API) 或者 Zipkin 跟踪器来完成，从而在入站请求中提取跟踪上下文，然后把上下文信息注入到后续的出站请求中。这种方法还可以让服务能够创建附加的 Span，用来描述服务内部完成的工作。这对端到端跟踪的检查是很有帮助的。

另外跟踪上下文也可以被服务手工进行传播：

- 如果使用了 LightStep 跟踪器，在发送 HTTP 请求到其他服务时，Envoy 依赖这个服务来传播 `x-ot-span-context` Header。
- 如果使用的是 Zipkin，Envoy 要传播的是 B3 Header。（`x-b3-traceid`, `x-b3-spanid`, `x-b3-parentspanid`, `x-b3-sampled`, 以及 `x-b3-flags`. `x-b3-sampled`）也可以由外部客户端提出，用来启用或者禁用某个服务的跟踪请求。

每条追踪中包含哪些数据

端到端的跟踪会包含亦或者多个 Span。一个 Span 就是一个逻辑上的工作单元，具有一个启动时间和时长，其中还会包含特定的 Metadata，Envoy 生成的 Span 包含如下信息：

- 使用 `--service-cluster` 设置的始发服务集群。
- 请求的启动时间和持续时间。
- 使用 `--service-node` 设置的始发服务主机。

- [x-envoy-downstream-service-cluster](#) Header 设置的下游集群。
- HTTP URL。
- HTTP 方法。
- HTTP 响应码。
- 跟踪系统的特定信息。

Span 中还要包括一个名字（或者操作名），这一信息通常是由服务所属主机定义的。也可以在路由上使用 [Decorator](#) 来进行自定义。还可以使用 [x-envoy-decorator-operation](#) Header 来设置名称。

Envoy 自动把 Span 发送给跟踪信息采集器。跟踪服务会使用其中的上下文信息来对多个 Span 进行拼装。例如[x-request-id](#) (LightStep) 或者 Zipkin 中的 trace ID 配置。

要获取更多 Envoy 跟踪设置方面的信息，可以阅读如下链接：

- [v1 API 参考](#)
- [v2 API 参考](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:43:22

TCP 代理

Envoy 本质上是一个 L3/L4 服务器，实现 L3/L4 的代理功能是一件轻而易举的事情。TCP 代理过滤器可以在下游的客户端与上游的集群间实现最基本的 1:1 网络连接代理功能。

这可以用来做一个安全通道的替代品，或者与其他的过滤器聚合如 [MongoDB 过滤器](#)或[速率限制过滤器](#)。

TCP 代理过滤器会被上游集群全局资源管理器中使用的[连接限制](#)约束。

TCP 代理过滤器需要和上游集群的资源管理器协商是否能在不逾越集群最大连接数的前提下，建立一个新连接，如果答案为否则 TCP 代理将不可以建立新连接。

TCP 代理过滤器[参考配置](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:41:23

访问记录

[HTTP 连接管理](#)与 [tcp 代理](#)支持可扩展的访问记录，并拥有以下特性：

- 可按每个 HTTP 连接管理或 tcp 代理记录任何数量的访问记录。
- 异步 IO 架构。访问记录将永远不会阻塞主要的网络处理线程。
- 可定制化的访问记录格式,可使用预制定的字段,更可使用任意的 HTTP request 以及 response。
- 可定制化的访问记录过滤器,可允许不同类型的请求以及回复写入至不同的访问记录。

访问记录 [配置](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-18 10:20:10

MongoDB

Envoy 支持网络层的 MongoDB 嗅探过滤器，并拥有如下特性：

- MongoDB wire 格式的 BSON 转换器。
- 考虑周全的 MongoDB 查询/操作统计分析报告，包括了时间以及路由集群中的散列/多次获取次数。
- 查询记录。
- 通过 \$comment 参数做每个调用点的统计分析报告。
- 故障注入。

MongoDB 过滤器是表现 Envoy 扩展性以及核心抽象能力的典范案例。在 Lyft 我们将这个过滤器应用在所有的应用以及数据库中。它提供了对应用程序平台和正在使用的特定 MongoDB 驱动程序不可知的重要数据源。

MongoDB 代理过滤器[参考配置](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-18 15:49:53

DynamoDB

Envoy 支持一个 HTTP 级 DynamoDB 嗅探过滤器，该过滤器有以下特性：

- DynamoDB API 请求/响应解析。
- DynamoDB 按每操作、每数据库表、每分区以及操作的统计信息。
- 故障类型统计覆盖 4xx 响应、由响应 JSON 解析，例如 ProvisionedThroughputExceededException。
- 批处理操作部分失败的统计信息。

它为应用程序平台以及特定的 AWS SDK 提供了宝贵的对数据无感知的来源。

DynamoDB 过滤器 [配置](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 10:41:03

Redis

Envoy 可以作为 Redis 的代理，在集群的实例间对命令进行分区。在这种模式下，Envoy 的目标是保持可用性和分区容错的一致性。这是将 Envoy 和 [Redis 集群](#) 进行比较时的重点。Envoy 被设计为尽力而为的缓存，这意味着它不会尝试协调不一致的数据或保持全局一致的集群成员关系视图。

Redis 项目中提供了与 Redis 分区相关的全面参考。请参阅 "[Partitioning: how to split data among multiple Redis instances](#)"。

Envoy Redis 的特点:

- [Redis 协议](#) 编解码器
- 基于散列的分区
- Ketama 一致性哈希算法
- 详细的命令统计
- 主动和被动的健康检查

有计划的未来增强功能:

- 额外的时间统计
- 断路器
- 对分散的命令进行请求折叠
- 复制
- 内置的重试功能
- 跟踪
- 哈希标记

配置

有关过滤器配置的详细信息，请参阅 [Redis 代理过滤器配置参考](#)。

相应的集群定义应该配置为 [环哈希负载均衡](#)。

如果需要进行主动健康检查，则应该对集群配置使用 [Redis 健康检查](#)。

如果需要被动健康检查，还需要配置 [异常检测](#)。

为了进行被动健康检查，需要将连接超时，命令超时和连接关闭都映射到 5xx 响应，而来自 Redis 的所有其他响应都视为成功。

支持的命令

在协议层面，支持流水线，但不支持 MULTI(事务块)。应该尽可能的使用流水线以获得最佳性能。

在命令层面，Envoy 仅支持可能的散列到服务器的命令。PING 命令是唯一的例外，Envoy 将立即使用 PONG 作为回复。对 PING 命令使用参数是不被支持的。所有其他支持的命令都必须包含一个键。受支持的命令在功能上与原始的 Redis 命令相同，除非出现了故障情况。

有关每个命令用法的详细信息，请参阅官方的 [Redis 命令参考](#)。

Command	Group
PING	Connection

DEL	Generic
DUMP	Generic
EXISTS	Generic
EXPIRE	Generic
EXPIREAT	Generic
PERSIST	Generic
PEXPIRE	Generic
PEXPIREAT	Generic
PTTL	Generic
RESTORE	Generic
TOUCH	Generic
TTL	Generic
TYPE	Generic
UNLINK	Generic
GEOADD	Geo
GEODIST	Geo
GEOHASH	Geo
GEOPOS	Geo
GEORADIUS_RO	Geo
GEORADIUSBYMEMBER_RO	Geo
HDEL	Hash
HEXISTS	Hash
HGET	Hash
HGETALL	Hash
HINCRBY	Hash
HINCRBYFLOAT	Hash
HKEYS	Hash
HLEN	Hash
HMGET	Hash
HMSET	Hash
HSCAN	Hash
HSET	Hash
HSETEX	Hash
HSTRLEN	Hash
HVALS	Hash
LINDEX	List

LINSERT	List
LLEN	List
LPOP	List
LPUSH	List
LPUSHX	List
LRANGE	List
LREM	List
LSET	List
LTRIM	List
RPOP	List
RPUSH	List
RPUSHX	List
EVAL	Scripting
EVALSHA	Scripting
SADD	Set
SCARD	Set
SISMEMBER	Set
SMEMBERS	Set
SPOP	Set
SRANDMEMBER	Set
SREM	Set
SSCAN	Set
ZADD	Sorted Set
ZCARD	Sorted Set
ZCOUNT	Sorted Set
ZINCRBY	Sorted Set
ZLEXCOUNT	Sorted Set
ZRANGE	Sorted Set
ZRANGEBYLEX	Sorted Set
ZRANGEBYSCORE	Sorted Set
ZRANK	Sorted Set
ZREM	Sorted Set
ZREMRANGEBYLEX	Sorted Set
ZREMRANGEBYRANK	Sorted Set
ZREMRANGEBYSCORE	Sorted Set
ZREVRANGE	Sorted Set

ZREVRANGEBYLEX	Sorted Set
ZREVRANGEBYSCORE	Sorted Set
ZREVRANK	Sorted Set
ZSCAN	Sorted Set
ZSCORE	Sorted Set
APPEND	String
BITCOUNT	String
BITFIELD	String
BITPOS	String
DECR	String
DECRBY	String
GET	String
GETBIT	String
GETRANGE	String
GETSET	String
INCR	String
INCRBY	String
INCRBYFLOAT	String
MGET	String
MSET	String
PSETEX	String
SET	String
SETBIT	String
SETEX	String
SETNX	String
SETRANGE	String
STRLEN	String

失败模式

如果 Redis 抛出了错误，我们会将这个错误作为响应传递给命令。Envoy 将来自 Redis 的响应与错误数据类型视为正常响应，并将它传递给调用者。

Envoy 也会产生自己的错误来响应客户端。

错误	含义
没有上游主机	环哈希负载均衡器在为键的选择的位置上没有可用的主机
上游主机错误	后端没有在超时期限内进行响应或关闭连接

无效的请求	由于数据类型或长度的原因，命令在命令拆分器的第一阶段被拒绝
不支持的命令	该命令未被 Envoy 识别，因此不能被服务，因为它不能被哈希到后端的服务器
返回了多个错误	分段的命令将会组合多个响应(例如 DEL 命令)，将返回接收到的错误总数
上游协议错误	分段命令收到意外的数据类型或后端响应的数据不符合 Redis 协议
命令参数数量错误	Envoy 中的命令参数数量检查未通过

在 MGET 命令中，每个无法被获取的键都会产生一个错误响应。例如，如果我们获取五个键的指并且其中有两个键出现了后端响应超时的错误，我们将会获得每个值得错误响应信息。

```
$ redis-cli MGET a b c d e
1) "alpha"
2) "bravo"
3) (error) upstream failure
4) (error) upstream failure
5) "echo"
```

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 10:21:03

热重启

易于使用是 Envoy 的一个重要的使命。除了健壮性统计以及本地的管理接口，Envoy 还可以“热”或者说是“活”重启自己。

这意味着 Envoy 可以在不需丢失任何连接的情况下完全地重载自己(包括代码与配置)。热重启功能实现如下的一些架构：

- 统计以及部分的锁将被保存在共享内存区。这也就是意味着这些标准即使在重启时还可以在进程间保持持久化。
- 两个活跃进程间通讯时，会在 unix domain socket 的基础上使用基础的 RPC 协议。
- 新进程首先将所有自身的初始化进程完成(包括载入配置、初始化服务发现和健康检查步骤等等)，然后再去请求旧的进程取得侦听 socket 的拷贝。

新进程开始进行侦听并通知旧的线程开始删除动作。

- 在删除的旧进程的时候，旧进程会尝试优雅地关闭现有的连接。可以怎么做到这点呢？这取决于配置过滤器。可以使用 `--drain-time-s` 参数进行配置删除时间，如果传的时间参数越大，则删除动作会越发激进。
- 在删除完成后，新的 Envoy 进程将告知旧的 Envoy 进程将自己关掉。这个时间可以通过 `--parent-shutdown-time-s` 参数进行配置。
- Envoy 的热重启支持时经过精心设计的，重启机制可以在新 Envoy 进程与旧 Envoy 进程在不同容器运行时发挥良好作用。

进程间通讯只依赖于 unix domain sockets。

- 在源码分发包里，你可以找到一个使用 Python 写的重启/父进程的案例。父进程在标准的流程控制工具中大有用武之地，例如 monit、runit 等等。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:04:21

动态配置

Envoy 的架构使得使用不同类型的配置管理方法成为可能。部署中采用的方法将取决于实现者的需求。简单部署可以通过全静态配置来实现。更复杂的部署可以递增地添加更复杂的动态配置，缺点是实现者必须提供一个或多个基于外部 REST 配置的提供者 API。本文档概述了当前可用的选项。

- [顶级配置参考](#)
- [参考配置](#)
- [Envoy v2 API 概述](#)

全静态

在全静态配置中，实现者提供一组[监听器](#)（和[过滤器链](#)）、[集群](#)和可选的[HTTP 路由配置](#)。动态主机发现仅能通过基于DNS 的[服务发现](#)。配置重载必须通过内置的[热重启](#)机制进行。

虽然简单，但可以使用静态配置和优雅的热重启来创建相当复杂的部署。

仅 SDS/EDS

服务发现服务（SDS）API 提供更高级的机制，Envoy 可以通过该机制发现上游集群中的成员。SDS 已在 v2 API 中重命名为 Endpoint Discovery Service（EDS）。在静态配置之上，SDS 允许 Envoy 部署规避 DNS 的局限性（响应中的最大记录等），并使用更多可用于负载均衡和路由的信息（例如，金丝雀状态、区域等）。

SDS/EDS 和 CDS

[集群发现服务（CDS）API](#) 是 Envoy 的一种机制，在路由期间可以用来发现使用的上游集群。Envoy 将优雅地添加、更新和删除由 API 指定的集群。该 API 允许实现者构建拓扑，在其中 Envoy 在初始配置时不需要知道所有上游群集。通常，在与 CDS（但没有路由发现服务）一起进行 HTTP 路由时，实现者将利用路由器的能力将请求转发到在[HTTP 请求头](#)中指定的集群。

尽管可以通过指定全静态集群来使用不带 SDS/EDS 的 CDS，但我们仍建议为通过 CDS 指定的集群使用 SDS/EDS API。在内部，更新集群定义时，操作是优雅的。但是，所有现有的连接池都将被排空并重新连接。SDS/EDS 不受此限制。当通过 SDS/EDS 添加和删除主机时，集群中的现有主机不受影响。

SDS/EDS、CDS 和 RDS

[路由发现服务（RDS）API](#) 是 Envoy 的一种机制，可以在运行时发现用于 HTTP 连接管理器过滤器的整个路由配置。路由配置将优雅地交换，而不会影响现有的请求。这个 API 与 SDS/EDS 和 CDS 一起使用时，允许实现者构建复杂的路由拓扑（[流量转移](#)、蓝绿部署等），除了获取新的 Envoy 二进制文件外，不需要重启 Envoy。

SDS/EDS、CDS、RDS 和 LDS

[监听器发现服务（LDS）](#) 是 Envoy 的一种机制，可以在运行时发现整个监听器。这包括所有的过滤器堆栈，直到并包括带有内嵌到 RDS 的应用的 HTTP 过滤器。将 LDS 添加到混和中，几乎可以动态配置 Envoy 的各个方面。只有非常少见的配置更改（管理员、追踪驱动程序等）或二进制更新时才需要热启动。

初始化

Envoy 在启动时的初始化是很复杂的。本章将在高级别解释这个过程是如何工作的。以下会在任何监听器启动监听并接收新连接前发生。

- 启动期间，[集群管理器](#) 会首先进行多阶段初始化，首先初始化静态/ DNS 集群，然后是预定义的 [SDS](#) 集群。然后，如果适用，它会初始化 [CDS](#)，等待响应（或失败），并为 CDS 提供的集群执行相同主/次初始化。
- 如果集群使用 [主动健康检查](#)，Envoy 也会执行单个主动HC轮次。
- 集群管理器初始化完成后，[RDS](#) 和 [LDS](#) 将初始化（如果适用）。在 LDS / RDS 请求至少有一次响应（或失败）之后，服务器才开始接受连接。
- 如果 LDS 本身返回需要 RDS 响应的监听器，则 Envoy 会进一步等待，直到收到 RDS 响应（或失败）。请注意，这个过程发生在未来的每个通过 LDS 添加的监听器上，并且被称为 [监听器热身](#)。
- 在先前所有的步骤发生之后，监听器开始接受新的连接。该流程可确保在热启动期间新流程完全能够在旧流程排除之前接受并处理新连接。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-17 13:29:18

排除

排除是 Envoy 相应各种事件的要求试图优雅地关闭连接的过程。排除发生在以下时机：

- 服务器通过 [healthcheck/fail](#) 管理端点手动设置健康状况检查失败。有关更多信息，请参阅 [健康检查过滤器](#) 架构概述。
- 服务器 [热重启](#)。
- 通过 [LDS](#) 修改或者移除单个监听器。

每个 [已配置监听器](#) 有 `drain_type` 设置，用来控制排除何时发生。目前支持的值有：

- `default`

对于所有上述三种情况（管理排除，热重启和 LDS 更新/删除），Envoy 将排除监听器。这是默认设置。

- `modify_only`

Envoy 只会响应上述第二和第三种情况（热重启和 LDS 更新/删除）而排除监听器。如果 Envoy 同时托管 ingress 和 egress 监听器，此设置很有用。需要在 egress 监听器上设置 `modify_only`，以便在尝试执行受控关闭，依赖 ingress 监听器排除来执行全服务器排除时，它们只在修改期间排除。

请注意，虽然排除是每监听器的概念，但它必须在网络过滤器级别被支持。目前支持优雅排除的过滤器只有 [HTTP 连接管理器](#)、[Redis](#) 和 [Mongo](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:40:17

脚本

Envoy试验性的支持 [Lua](#) 脚本作为专用 [HTTP 过滤器](#) 的一部分。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-17 14:58:23

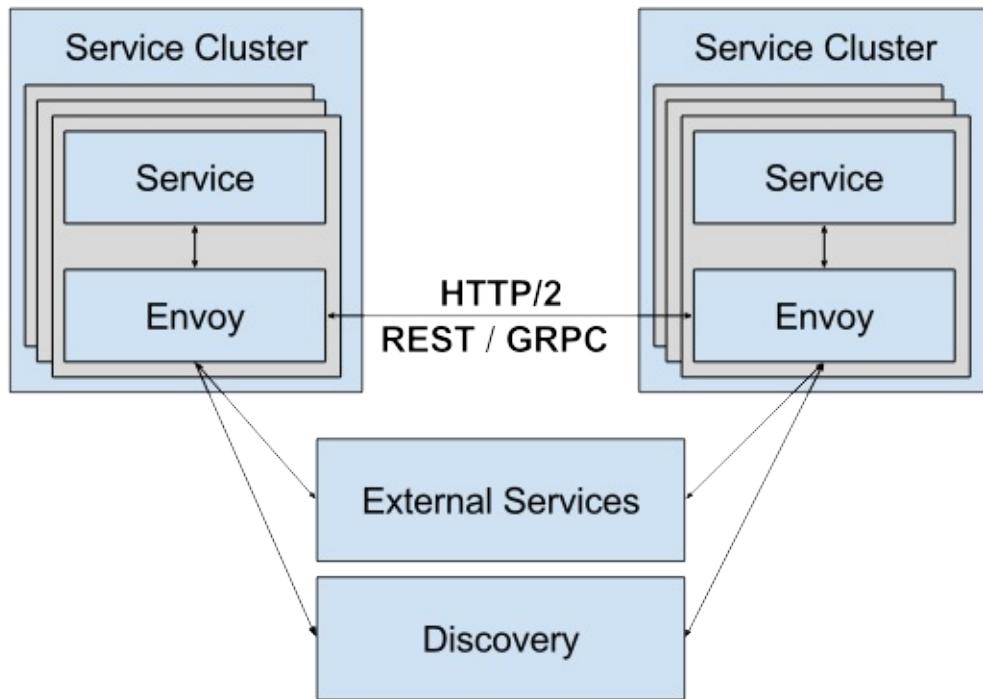
部署类型

Envoy 有多种使用场景，其中更多情况下会作为 *mesh* 被部署在基础设施的不同主机上。本章节将按照复杂性上升的顺序讨论 envoy 的三种推荐部署类型。

- 仅服务之间
 - [服务间 egress listener](#)
 - [服务间 ingress listener](#)
 - [可选外部服务 egress listener](#)
 - [发现服务集成](#)
 - [配置模板](#)
- 服务见外加前端代理
 - [配置模板](#)
- 服务间、前端代理、双向代理
 - [配置模板](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:07:04

仅服务之间



上图显示了最简单的 Envoy 部署，它使用 Envoy 作为面向服务架构（SOA）内部的所有流量的通信总线。在这种情况下，Envoy 暴露了几个用于本地来源流量以及服务流量的监听器。

服务间 egress listener

这是应用程序与基础结构中的其他服务对话所使用的端口。例如，<http://localhost:9001>。HTTP 和 gRPC 请求使用 HTTP/1.1 *host* 头或 HTTP/2:*authority* 头来表示请求指向哪个远程集群。根据配置中的详细信息，Envoy 处理服务发现、负载平衡、速率限制等。服务只需要了解本地 Envoy，不需要关心网络拓扑结构，无论他们是在开发还是在生产中运行。

此监听器支持 HTTP/1.1 或 HTTP/2，具体取决于应用程序的功能。

服务间 ingress listener

这是远程 Envoy 想要与本地 Envoy 通信时使用的端口。例如，<http://localhost:9211>。入向请求将通过配置的端口路由到本地服务。根据应用程序或负载平衡需求（例如，如果服务需要 HTTP 端口和 gRPC 端口），可能会涉及多个应用程序端口。本地 Envoy 根据需要执行缓冲、熔断等。

我们的默认配置对所有 Envoy 相互间的通信使用 HTTP/2 协议，无论应用程序在通过本地 Envoy 出向时使用的是 HTTP/1.1 还是 HTTP/2。HTTP/2 通过长连接和显式重置通知提供更好的性能。

可选外部服务 egress listener

通常，本地服务要通信的每个外部服务都要使用显式定义的出口端口。这是因为一些外部服务 SDK 不会轻易支持覆盖 *host* 头以允许标准的 HTTP 反向代理行为。例如，<http://localhost:9250> 可以被分配作为 DynamoDB 的连接地址。我们建议为所有外部服务使用本地端口路由，而不是为一些外部服务使用 *host* 路由，另一些使用专用本地端口路由。

发现服务集成

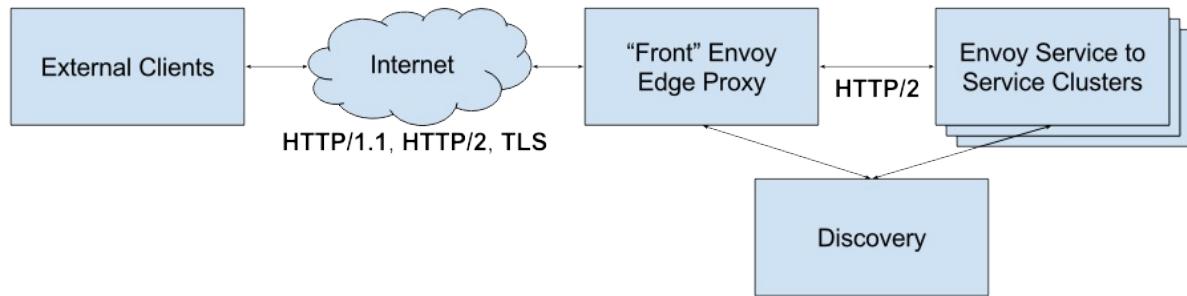
建议的服务之间的配置对所有集群查找使用外部发现服务。这为 Envoy 在执行负载平衡，统计收集等时提供了可能的最详细信息。

配置模板

源代码发行版包含一个服务配置示例服务，与 Lyft 在生产环境中运行的版本非常相似。有关更多信息，请参阅[这里](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:06:42

服务间外加前端代理



上图显示了作为 HTTP L7 边缘反向代理 Envoy 群集的 [服务到服务](#) 配置。反向代理提供以下功能：

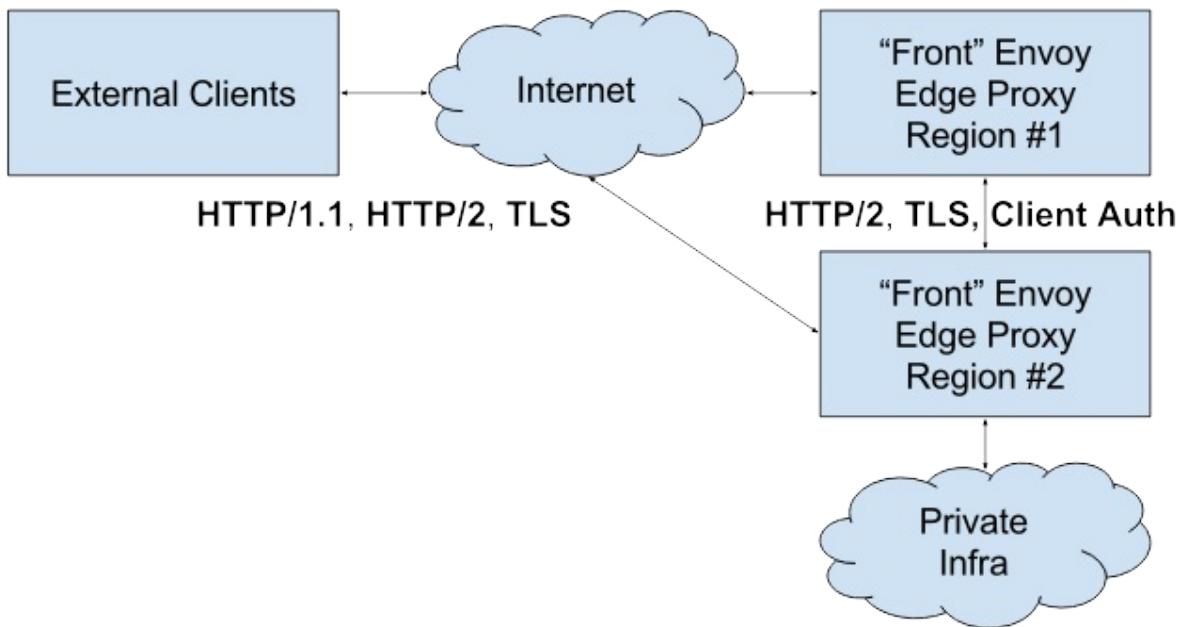
- 终止 TLS。
- 支持 HTTP/1.1 和 HTTP/2。
- HTTP L7 全路由支持。
- 与服务到服务的 Envoy 集群使用标准 [ingress port](#) 通信，使用使用发现服务进行主机查找。因此，前端 Envoy 主机与任何其他 Envoy 主机的工作方式相同，除了他们不与其他服务搭配运行。这意味着以相同的方式操作他们并发出相同的统计数据。

配置模板

源代码发行版包含一个与 Lyft 在生产环境中运行的版本非常相似的示例前端代理配置。浏览[此处](#)获取更多信息。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-23 13:25:49

服务间、前端代理、双向代理



上图展示了 [前端代理](#) 与另一个 Envoy 集群组成的 双向代理 的配置。双向代理背后的想法是，尽可能接近用户地终止 TLS 和客户端连接（TLS 握手的更短往返时间，更快的 TCP CWND 扩展，更小几率的数据包丢失等）会更高效。连接在双向代理中被终止，然后被复用到运行在主数据中心的 HTTP/2 长连接中。

在上图中，在区域1中运行的 Envoy 前端代理通过 TLS 相互身份验证和固定证书与在区域2中运行的 Envoy 前端代理进行身份验证。这使得在区域2中运行的 Envoy 前置实例能信任通常不可信的传入请求的元素（例如 x-forwarded-for HTTP 头）。

配置模板

源码包含一个与 Lyft 在生产环境中运行的版本非常相似的示例双向代理配置。有关更多信息，请参阅 [这里](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-23 15:30:32

与类似系统比较

一言概之，我们相信 Envoy 有一套独有特色的优秀功能集合，以支撑现代面向服务的架构。以下我们将以 Envoy 与类似的其余系统进行比对。虽然在某些特定领域(例如边缘代理，软负载均衡，服务消息传送层)，Envoy 未如以下的某些系统提供更加完备的支持，但在完整比对后，没有其他任何一个系统能提供一套例如 Envoy 一般拥有完备功能，自包含，以及高性能的解决方案。

NOTE: 以下的许多项目还处在一个开发活跃期，我们所描述的信息也许会与项目最新的状态脱节，如果您发现这些情况，请立即告知我们，我们将会进行修正。

nginx

nginx 是一个经典的现代 web 服务器。它支持静态内容展现，HTTP L7 反向代理 负载均衡，HTTP/2，以及其他许多特性。作为一个边缘反向代理，nginx 提供了远远多于 Envoy 的功能特性，但我们认为大多数的现代面向服务的架构其实不需要用到那么多的特性。而 Envoy 在下列边缘代理的特性做得比 nginx 更为出色：

- 完备的 HTTP/2 透明代理支持。Envoy 支持 HTTP/2 包括上游连接以及下游连接在内的双向通信。而 nginx 仅仅支持 HTTP/2 下游连接。
- 免费的高级负载功能。而在 nginx 的世界，只有付费的 nginx plus 服务器才能提供类同于 Envoy 的高级负载功能。
- 可以在每一个服务节点的边界运行同样一套软件来处理事务。在许多架构体系中，需要使用 nginx 与 haproxy 的混合部署架构。相比之下，一个独立的代理解决方案会更有利于后续的运维维护。

haproxy

haproxy 是一个经典的现代软负载均衡服务器。它提供基本的 HTTP 反向代理功能。而 Envoy 在下列负载均衡的特性做得比 haproxy 更为出色：

- HTTP/2 支持。
- 可插拔架构。
- 与远程服务发现服务的整合。
- 与远程全局限速服务的整合。
- 提供大量的更为细致的统计分析。

AWS ELB

Amazon 的 ELB 为在 Amazon EC2 上运行的程序提供服务发现以及负载均衡服务。而 Envoy 在下列负载均衡以及服务发现的特性做得比 ELB 更为出色：

- 统计与日志 (CloudWatch 的统计有延迟，并且在一些细节上严重缺乏，而 Amazon 的日志只能在 S3 以特定格式获取)
- 稳定性(使用 ELB 时不时会遇到不稳定事故，并且难以进行 debug 排查问题)
- 更为高级的负载均衡以及节点间的直连功能。Envoy mesh 在进行硬件弹性伸缩时并不需要额外的网络跳转。负载均衡可以根据区域，金丝雀状态的回馈提供更好的决策以及收集到许多有意思的统计分析结果。而且负载均衡还支持例如重试这样的高级特性。

AWS 最近发布了一个 *application load balancer* 产品。这个产品增加了 HTTP/2 支持，可以将 HTTP/2 与基本的 HTTP L7 请求一般流转到不同的后端集群。相比 Envoy，这个产品所能提供的功能偏少，且性能与稳定性还未知，但不容置疑的是 AWS 将会在这个领域持续地进行研发。

SmartStack

SmartStack 这个方案特别有意思，它在 haproxy 的基础上提供了更多的服务发现以及健康检查支持。抽象地说，SmartStack 在大多数目标上与 Envoy 保持一致（与进程无关的架构，对应用平台的不可知性等）。而 Envoy 在下列负载均衡以及服务发现的特性做得比 SmartStack 更为出色：

- 上述所提及的做的比haproxy好的所有特性。
- 整合服务发现与积极的健康检查。Envoy 将所有的功能都在一个高性能的方案内完整提供。

Finagle

Finagle 是 Twitter 基于 Scala/JVM 开发的服务通讯库。在Twitter 以及其他公司的基于 JVM 的架构中普遍使用。它提供了服务发现，负载均衡，过滤器等 Envoy 也具有的功能。而 Envoy 在下列负载均衡以及服务发现的特性做得比 Finagle 更为出色：

- 最终一致的服务发现，基于在分布式的积极健康检查基础上实现
- 在各个维度上都有更优越的性能表现(内存占用率，CPU 使用率，P99 时延)
- 与进程无关的架构，对应用平台的不可知性的架构。Envoy与不同的应用技术栈都可一起工作。

proxygen 和 wangle

proxygen 是 Facebook 使用 C++11 开发的高性能 HTTP 代理库，是基于一个类同 Finagle 的 C++ 库 wangle 进行开发。在代码实现层面，Envoy 使用了类同的技术去实现一个高性能的的 HTTP 库/代理。除此之外，两个项目不具备可比性。Envoy 是一个完备的自包含服务，提供了大量的功能点。相比之下，proxygen 仅仅是一个库，各个项目还需要基于它继续构建功能。

gRPC

gRPC 是一个由 google 研发的跨平台的消息传递系统。它使用 IDL 去描述 RPC 库，然后基于 IDL 为各种不同的编程语言生成不同的应用运行时。它底层基于 HTTP/2 去实现传输层。gRPC 似乎有一个终极的目标，在将来去实现许多 Envoy 的功能点(例如负载均衡)，但在我们写就这篇文档的时候，还有不少的运行时库还不成熟并且集中在序列以及反序列化上。我们认为 gRPC 是 Envoy 的搭档，而不是竞争者。Envoy 如何与 gRPC 进行整合，您可以查看[此链接](#)。

linkerd

linkerd 是一个独立的，开源的 RPC 路由代理，它基于 Netty 与 Finagle(Scala/JVM) 研发。linkerd 提供了许多 Finagle 的特性，包括对延时敏感的负载均衡，连接池，断路器，重试预算，截至线，追踪，细粒度的插入，以及对请求的流量路由层。linkerd 提供一个可插拔的服务发现层 (Consul 以及 Zookeeper 为此提供标准支持，就如 Marathon 以及 Kubernetes 的 API)。

linkerd 的内存消耗以及 CPU 的要求都远远高于 Envoy。对比 Envoy，linkerd 仅提供了一个最小可用的配置语言，且不支持热加载，而是用动态配置以及服务抽象的方式变相地提供类似的功能。linkerd 支持 HTTP/1.1，Thrift，ThriftMux，HTTP/2 (experimental) 和 gRPC (experimental)。

nghttp2

nghttp2 是一个包含了不同事物的项目。首先，它包含了一个库 (nghttp2)，用来实现HTTP/2协议。Envoy使用这个库 (在这个库上做了一层很浅的封装)来做HTTP/2的支持。这个项目还包含了一个非常有用的压力测试工具(h2load) 以及一个反向代理 (nghttpx)。如果要做个比对，在nghttp2 所实现的众多功能点中，我们认为 Envoy更类似于 nghttpx，nghttpx是一个透明的 HTTP/1 <-> HTTP/2 反向代理，支持 TLS 终止，支持gRPC代理。我们认为 nghttpx 是一个表现不同代理功能点的杰出范例，而并不是一个健壮的生产可用的解决方案。Envoy的产品焦点更多地放在可监控性，操作的敏捷性，以及高级的负载均衡功能。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-17 18:15:43

获取帮助

我们非常有兴致地正在建立一个围绕 Envoy 产品的技术社区。如果您在使用 Envoy 上需要帮助或者是您想为社区做出贡献, 请尝试联系我们。

请查看 [联系信息](#)。

报告安全性缺陷

请查看 [安全联系信息](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-17 18:15:43

历史版本

1.7.0 (日期待定)

- 访问日志：能够将响应尾存入日志
- 访问日志：能够格式化 START_TIME
- 访问日志：添加了 DYNAMIC_METADATA [访问日志格式化器](#)。
- 访问日志：添加了 HeaderFilter 以基于请求头过滤日志
- 管理：添加了 `GET /config_dump` 用于保存当前配置和相关 xDS 版本信息 (如果可用)。
- 管理：添加了 `GET /stats/prometheus`，作为以 prometheus 格式获得状态的一个替代的端点。
- 管理：添加了 `/runtime_modify` 端点 以添加或改变运行时值
- 管理：突变必须作为 POST 而不是 GET 发送。突变包括: `POST /cpuprofiler`, `POST /healthcheck/fail`, `POST /healthcheck/ok`, `POST /logging`, `POST /quitquitquit`, `POST /reset_counters`, `POST /runtime_modify?key1=value1&key2=value2&keyN=valueN`。
- 管理：移除了 `/routes` 端点；路由配置现在可以在 `/config_dump` 端点找到。
- 缓存过滤器:缓存过滤可以被可选地用路由本地配置[关闭](#) 或者 [被撤销](#)。
- cli: 添加了 `--config-yaml` 标志到 Envoy 二进制文件。当将其值被解释为引导配置和撤销 `--config-path` 的一个 yaml 表示。
- 集群：添加[选项](#)在健康状况检查失败时关闭 `tcp_proxy` 上游连接。
- 集群：添加[选项](#)在来自主机的连接被从服务发现中移除后耗尽它们，与健康状态无关。
- 集群：修复了阻止删除同一优先级的所有端点的缺陷。
- 健康状况检查：添加了设置[额外 HTTP 头](#)用于 HTTP 健康状况检查的能力。
- 健康状况检查：添加了支持 EDS 发送的[端点健康状况](#)。
- 健康状况检查：添加了健康状态转换的区间撤销，[健康到不健康](#)，[不健康到健康](#) 并后续对[不健康的主机](#)的检查。
- 健康状况检查 http 过滤器：添加了[通用头匹配](#) 以触发健康状况检查响应。弃用了[端点选项](#)。
- 健康状况检查：添加了支持[定制健康状况检查](#)。
- http：过滤器现在可以可选地支持[虚拟主机](#)，[路由](#)，和[加权集群](#)本地配置。
- http：添加了传送在 `x-forwarded-client-cert` 头中的客户端证书 DNS 类型主题替代名 (Subject Alternative Names) 的能力。
- 监听器：添加了 `tcp_fast_open_queue_length` 选项。
- 负载均衡器：添加了[加权轮询](#) 支持。轮询调度群现在关心端点权重并且也提升了跨选择的保真度。
- 负载均衡器：[本地化加权负载均衡](#)现在被支持。
- 负载均衡器：能够[通过 API](#) 设置配置域感知负载均衡器。
- logger：添加了通过 `--log-format` 选项可选地设置日志格式的能力。
- logger：所有[日志级别](#) 可以在运行时配置: trace debug info warning error critical。
- sockets：添加了[捕捉传输套接字扩展](#) 以支持记录纯文本流量和 PCAP 生成。
- sockets：添加了 IP_FREEBIND 套接字选项，通过[集群管理器范围](#) 和[集群特定的](#) 选项，支持[监听器](#) 和上游连接。
- sockets：添加了 IP_TRANSPARENT 套接字选项支持[监听器](#)。
- sockets：添加了 SO_KEEPALIVE 套接字选项用于[每个集群](#)的上游连接。
- stats：添加了支持柱状图。
- stats：添加了[选项配置 statsd 前缀](#)。
- tls：移除了遗留的 SHA-2 CBC 密码套件。
- 追踪：采样决策现在授予了追踪器，允许追踪器决定何时以及是否使用它。例如，如果 `x-b3-sampled` 头被以客户端请求提供，它的值会覆盖由 Envoy 代理做的任何决策。
- websocket：支持配置[空闲](#)和[max_connect_attempts](#)。

1.6.0 (2018年3月20日)

- 访问日志：添加了 DOWNSTREAM_REMOTE_ADDRESS、DOWNSTREAM_REMOTE_ADDRESS_WITHOUT_PORT 和 DOWNSTREAM_LOCAL_ADDRESS 访问日志格式化器。DOWNSTREAM_ADDRESS 访问日志格式化器被弃用。
- 访问日志：添加了小于或等于 (LE) 比较过滤器。
- 访问日志：添加了对运行时过滤器的配置，用于设置默认采样速率，因子，和是否使用独立随机。
- 管理：添加了 /runtime admin 端点读取当前运行时的值。
- 构建：添加了支持用输出的符合构建 Envoy。这个变化运行脚本用 Lua 过滤器加载，以加载共享对象库，如那些通过LuaRocks安装的库。
- 配置：添加了支持在 DiscoveryRequest 中发送错误细节，如 grpc.rpc.Status。
- 配置：添加了支持内置交付 TLS 证书和私钥。
- 配置：添加了 xDS 源恢复配置源的限制。对基于 xDS 的文件系统，文件必须在配置时存在。对于基于 xDS 的集群，恢复集群必须被静态定义并有非 EDS 类型。
- grpc：谷歌 gRPC C++ 库客户端现在作为在 gRPC 服务概览 和 [GrpcService] (https://www.envoyproxy.io/docs/envoy/latest/api-v2/api/v2/core/grpc_service.proto.html#envoy-api-msg-core-grpcservice) 中指定支持。
- grpc-json：添加了支持内置描述符。
- 健康状况检查：添加了基于 grpc.health.v1.Health 服务的 gRPC 健康状况检查。
- 健康状况检查：添加了设置 主机头值 用于健康状况检查的能力。
- 健康状况检查：扩展了健康状况检查过滤器以支持基于上游集群中健康的服务器百分比的健康状况检查计算。
- 健康状况检查：添加了设置 非流量区间。
- http：为上游 http 连接添加了空闲超时。
- http：添加了支持代理 100-Continue 响应。
- http：添加了在 x-forwarded-client-cert 头中传递一个 URL 编码的 PEM 对等证书的能力。
- http：添加了支持在 x-forwarded-for 请求头中信任额外的跳步。
- http：添加了支持进来的 HTTP/1.0。
- 热重启：添加了 SIGTERM 传递给子进程到 hot-restarter.py，这使得可以作为容器的一个父进程使用它。
- ip 标签：添加了 HTTP IP 标签过滤器。
- 监听器：添加了支持在绑定到 :: 时 监听IPv4 和 IPv6。
- 监听器：添加了 support for listening on UNIX domain sockets。
- 监听器：添加了 Linux 系统上的 抽象 unix 域套接字 sockets 支持。抽象命名空间可通过在一个套接字前面加上 '@' 使用。
- 负载均衡器：添加了对 健康崩溃阈值 百分比的集群配置。
- 负载均衡器：添加了 Maglev 一致哈希负载均衡器。
- 负载均衡器：添加了支持 LocalityLbEndpoints priorities。
- lua：添加了头 replace() API。
- lua：扩展支持元数据对象 API。
- redis：添加了对 Redis 过滤器的本地 PING 支持。
- redis：添加了 GEORADIUS_RO 和 GEORADIUS_BYMEMBER_RO 到 the Redis command splitter whitelist。
- 路由器：添加了 DOWNSTREAM_REMOTE_ADDRESS_WITHOUT_PORT, DOWNSTREAM_LOCAL_ADDRESS, DOWNSTREAM_LOCAL_ADDRESS_WITHOUT_PORT, PROTOCOL, and UPSTREAM_METADATA 头格式化器。CLIENT_IP 头格式化器被弃用。
- 路由器：添加了网关错误 retry-on 策略。
- 路由器：添加了对基于 URL 查询字符串参数 的路由匹配支持。
- 路由器：添加了通过允许 total_weight 在配置中指定支持更细粒度的加权集群路由。
- 路由器：添加了支持带有混合静态和动态值的定制请求/响应头。
- 路由器：添加了对直接响应的支持。即，发送一个预配置的 HTTP 响应而无需任何代理。
- 路由器：添加了在特定路由上对 HTTPS 重定向的支持。
- 路由器：添加了用于重定向的 prefix_rewrite 支持。

- 路由器：添加了对重定向的剥离查询字符串的支持。
- 路由器：添加了在[加权集群](#)中支持下游请求/上游响应头操作。
- 路由器：添加了支持用于请求路由的[基于范文的头匹配](#)。
- squash：添加了支持[Squash 微服务调试程序](#)。允许调试一个对网络中某个微服务的请求。
- stats：添加了度量服务 API 实现。
- stats：添加了原生[DogStatsd](#) 支持。
- stats：添加了支持[固定状态标签值](#)，这将被添加到所有度量标准中。
- tcp 代理：添加了支持在 tcp 过滤器中为上游集群指定一个[元数据匹配器](#)。
- tcp 代理：改善了 TCP 代理以正确代理 TCP 半关闭。
- tcp 代理：添加了[空闲超时](#)。
- tcp 代理：访问日志现在在使用 DOWNSTREAM_ADDRESS 时带来一个 IP address 而没有端口。使用 DOWNSTREAM_REMOTE_ADDRESS 替代。
- 追踪：添加了对动态加载[OpenTracing](#)追踪器的支持。
- 追踪：当使用 Zipkin 追踪器时，现在客户端程序可以指定取样决策 (使用 `x-b3-sampled` 头) 并将决策传递给后续被启动的服务。
- 追踪：当使用 Zipkin 追踪器时，不再需要传递 `x-ot-span-context` 头。关于追踪上下文传递的更多内容，参见[这里](#)。
- 传输套接字：添加了传输套接字接口以允许传输套接字的定制实现。一个传输套接字提供带缓冲加密和解密(如果可用)的读和写逻辑。现存的 TLS 实现已被使用该接口重构。
- 上游：添加了支持在为集群放出状态时指定一个[替代状态名](#)。
- 很多小的缺陷修复和性能提高未列出。

1.5.0 (2017年12月4日)

- 访问日志：为 `UPSTREAM_LOCAL_ADDRESS` 和 `DOWNSTREAM_ADDRESS` 增加了字段。
- 管理：为 `stats admin` 端点添加 `JSON` 输出。
- 管理：为 `stats admin` 端点添加基本 `Prometheus` 输出。当前柱状图不输出。
- 管理：添加 `version_info` 到 `/clusters admin` 端点。
- 配置：`v2 API` 现在被认为是可用于生产系统的。
- 配置：添加了 `--v2-config-only` CLI 标志。
- cors：添加了 `CORS` 过滤器。
- 健康状况检查：添加了 `x-envoy-immediate-health-check-fail` 头支持。
- 健康状况检查：添加了 `reuse_connection` 选项。
- http：添加了 `单个监听器状态`。
- http：端到端 HTTP 流控现在跨连接、流和过滤器是完整的。
- 负载均衡器：添加了 `子集负载均衡器`。
- 负载均衡器：添加了 ring size and hash `configuration options`. This used to be configurable via runtime. The runtime configuration was deleted without deprecation as we are fairly certain no one is using it.
- 日志：添加了通过 `--log-path` 选项可选地将日志发到一个文件而不是标准输出的能力。
- 监听器：添加了 `drain_type` option。
- lua：添加了试验性 `Lua` 过滤器。
- mongo 过滤器：添加了 `故障注入`。
- mongo 过滤器：添加了“耗尽关闭”支持。
- 异常值检测：添加了 `HTTP 网关失败类型`。对这个发布中的异常值检测状态弃用，参见 [DEPRECATED.md](#)。
- redis：`redis` 代理过滤器 现在被认为可用于生产系统。
- redis：添加了“耗尽关闭 close”功能。
- 路由器：添加了 `x-envoy-overloaded` 支持。
- 路由器：添加了 `正则` 路由匹配。
- 路由器：为上游请求添加了 `定制请求头`。
- 路由器：为 HTTP ketama 路由添加了 `下游 IP 哈希`。

- 路由器：添加了 [cookie 哈希](#)。
- 路由器：添加了 [start_child_span](#) 选项为呼出调用创建子范围。
- 路由器：添加了可选的[上游日志](#)。
- 路由器：添加了请求/响应头的完整的[定制附加/覆盖/移除 支持](#)。
- 路由器：添加了对[在重定向期间指定响应代码](#)的支持。
- 路由器：添加了[配置](#)，如果上游集群不存在，返回 404 或 503。
- 运行时：添加了[评论能力](#)。
- 服务器：改变默认日志级别为([-1](#)) info。
- stats：最大 stat/名字尺寸和stats最大数现在可通过 [--max-obj-name-len](#) 和 [--max-stats](#) 选项变化。
- tcp 代理：添加了[访问日志](#)。
- tcp 代理：添加了[可配置连接重试](#)。
- tcp 代理：启用[异常值检测器](#)。
- tls：添加了[SNI 支持](#)。
- tls：添加了支持指定[TLS 会话票键](#)。
- tls：运行配置 [min](#) 和 [max](#) TLS 协议版本。
- 追踪：添加了[定制追踪范围装饰器](#)。
- 很多小的缺陷修复和性能提高未列出。

1.4.0 (2017年8月24日)

- macOS 被[支持](#)。(不少特性没有，如热重启和最初目的地路由)。
- 现在直接支持[配置文件](#)的YAML。
- 添加 /routes admin 端点。
- 现在支持对 TCP 代理、HTTP/1 和 HTTP/2 的端到端流控。包含过滤器缓存的 HTTP 流控不完整并将在 1.5.0 中实现。
- 添加日志详细程度 [编译时间标志]。(<https://github.com/envoyproxy/envoy/blob/master//bazel#log-verbosity>)。
- 添加热重启[编译时间标志](#)。
- 添加独创的目的地[集群](#)和[负载均衡器](#)。
- 现在支持[WebSocket](#)。
- 虚拟集群优先级被硬性移除而没有过渡，因为我们有理由确信没有人使用这一特性。
- 添加路由 [validate_clusters](#) 选项。
- 添加 [x-envoy-downstream-service-node](#) 头。
- 添加 [x-forwarded-client-cert](#) 头。
- 第一次为绝对 URLs 添加了 HTTP/1 转发代理支持。
- HTTP/2 编解码器设置现在是[可配置的](#)。
- 添加 gRPC/JSON 转码[过滤器](#)。
- 添加 gRPC web[过滤器](#)。
- 网络 中的速率限制服务调用和 HTTP 速率限制过滤器中的可配置超时。
- 添加 [x-envoy-retry-grpc-on](#) 头。
- 添加 [LDS API](#)。
- 添加 TLS [require_client_certificate](#) 选项。
- 添加 [Configuration 检查工具](#)。
- 添加 [JSON schema 检查工具](#)。
- 通过 [--mode](#) 选项添加配置验证模式。
- 添加 [--local-address-ip-version](#) 选项。
- IPv6 现在被完全支持。
- 添加 UDP [statsd_ip_address](#) 选项。
- 添加针对每个集群的[DNS 解析器](#)。
- [故障过滤器](#)增强和修复。
- 几个特性 在 1.4.0 发布中不再支持。它们将在 1.5.0 发布周期的开始被移除。我们将明确声明

HttpFilterConfigFactory 过滤器 API 被 NamedHttpFilterConfigFactory 代替。

- 很多小的缺陷修复和性能提高未列出。

1.3.0 (2017年5月17日)

- 从这个发布起，我们有了一个官方的 [打破改变策略](#)。请注意，在这个发布中有无数的打破配置改变。这里没有列出。将来的发布会遵循这一策略并有关于废除和修改的清晰文档。
- Bazel 现在是正式的构建系统 (取代 CMake)。开发/构建/测试流程有大量的变化。更多信息参见 [/bazel/README.md](#) 和 [/ci/README.md](#)。
- [异常值检测](#) 被扩展包括成功率变化，以及现在所有在运行时和 JSON 配置中可配置的参数。
- TCP 层 [监听器](#) 和 [集群](#) 连接现在有了可配置的接收缓冲区限制，在这个点，连接层回压被应用。完整的端对端流控将在未来的发布中可用。
- [Redis 健康状况检查](#) 被作为一个主动健康检查类型加入。完整的 Redis 支持将在 1.4.0 中被记录和支持。
- [TCP 健康状况检查](#) 现在支持“仅连接”模式，只检查远端的服务器是否可以被连接而不需要读/写任何数据。
- [BoringSSL](#) 是现在唯一支持的 TLS 提供者。默认的加密套件和 ECDH 曲线已经为 [监听器](#) 和 [集群](#) 连接更新了更现代的默认值。
- [头值匹配速率限制行为](#) 被扩展包括 *expect match* 参数。
- 路由层 HTTP 速率限制配置现在默认不继承虚拟主机层配置。如果想要，[include_vh_rate_limits](#) 继承虚拟主机层选项。
- HTTP 路由现在可以通过 [request_headers_to_add](#) 选项对每个路由和每个虚拟主机添加请求头。
- [配置例子](#) 已经被更新以展示最新的特性。
- [per_try_timeout_ms](#) 除了通过 [x-envoy-upstream-rq-per-try-timeout-ms](#) HTTP 头，现在可以被配置在一个路由项的策略中。
- [HTTP 虚拟主机匹配](#) 现在包括支持前缀通配符域名 (如, *.lyft.com)。
- 对于追踪随机取样的默认值被改为 100% 而且仍然是可以在[运行时](#)中配置。
- [HTTP 追踪配置](#) 被扩展以允许标签从任意 HTTP 头迁移。
- [HTTP 速率限制过滤器](#) 现在可以通过 [request_type](#) 被应用于内部、外部，或者任意请求。
- [监听器绑定](#) 现在需要指定一个地址与。这也被用于绑定一个监听器到一个特定地址以及一个端口。
- [MongoDB 过滤器](#) 现在为没有 \$maxTimeMS 集合的查询输出一个状态。
- [MongoDB 过滤器](#) 现在输出完全合法的 JSON 格式日志。
- The CPU profiler output path is now [configurable](#).
- 添加了一个 [看门狗系统](#)，如果检测到死锁可以杀掉服务器。
- 添加了一个 [路由表检查工具](#)，用于在使用前测试路由表。
- 我们已经添加了一个 [示例代码库](#)，展示如何编译/链接一个定制过滤器。
- 添加了额外的与异常值检测相关的集群范围信息到 [/clusters admin](#) 端点。
- 多个 SAN 现在可以通过 [verify_subject_alt_name](#) 设置验证。另外，URI 类型 SAN 可以被验证。
- HTTP 过滤器现在可以被传递给按照每个路由指定的[不透明的配置](#)。
- Envoy 现在默认有一个内建的崩溃处理程序，它将打印一条堆栈信息。如果需要，这个行为可以通过 `--define=signal_trace=disabled` Bazel 选项关掉。
- 添加了 Zipkin 作为一个支持的 [追踪提供者](#)。
- 此处未列出的数量庞大的小的变化和修复。

1.2.0 (2017年3月7日)

- [集群发现服务 \(CDS\) API](#)。
- [异常值检测](#) (主动健康检查)。
- Envoy 配置现在针对 [JSON schema](#) 检查。
- [环哈希](#) 一致性负载均衡器，以及 [基于策略](#) 的 HTTP 一致性哈希路由。
- 提供 HTTP 速率限制过滤器极大地 [增强的全局速率限制配置](#)。

- HTTP 路由到一个 [从头获取的](#)集群。
- [加权的集群](#) HTTP 路由。
- 在 HTTP 路由过程中[自动主机重写](#)。
- 在 HTTP 路由过程中[正则头部匹配 header matching](#)。
- HTTP 访问日志[运行时过滤器](#)。
- LightStep 追踪器[父/子跨关联](#)。
- [路由发现服务 \(RDS\)](#) API。
- HTTP 路由器 `x-envoy-upstream-rq-timeout-alt-response` header 支持。
- `use_original_dst` 和 `bind_to_port` 监听器选项 (对基于iptables 的透明代理支持有用)。
- TCP 代理过滤器[路由表支持](#)。
- 可配置[状态写入磁盘间隔](#)。
- 各种[第三方库升级](#), 包括使用 BoringSSL 作为默认的 SSL 提供者。
- 不再为优先级计算维护关闭的 HTTP/2 流。导致对大型网络实质性的内存节省。
- 此处未列出的数量庞大的小的变化和修复。

1.1.0 (2016年9月30日)

- 为了我们的 JSON 库从 Jansson 转换为 RapidJSON (在 1.2.0 中允许一个配置模式)。
- 升级各种其他库的[建议版本](#)。
- 用于 DNS 发现类型的[可配置 DNS 刷新速率](#)。
- 上游电路断路器配置可[通过运行时撤销](#)。
- [域感知路由支持](#)。
- 通用[头匹配路由规则](#)。
- HTTP/2 [优雅连接耗尽](#) (双 GOAWAY)。
- DynamoDB 过滤器[按片统计](#) (预发布 AWS 特性)。
- 故障注入[HTTP 过滤器](#)的第一版发布。
- HTTP [速率限制过滤器](#) 增强 (注意HTTP 速率限制的配置讲座 1.2.0 中被修正)。
- 添加了[被拒绝流重试策略](#)。
- 用于上游集群的多[优先级队列](#) (可针对每个路由基础, 用各自的连接池, 电路断路器, 等等, 配置)。
- 添加了最大连接电路断路器到[TCP 带理过滤器](#)。
- 添加了[CLI](#) 选项用于设置日志文件写入磁盘的时间间隔以及在热重启期间耗尽/关机时间。
- 非常大数量的核心 HTTP/TCP 流性能提升, 以及不少新的配置标志以允许关掉昂贵的特性, 如果它们不需要 (特别是请求 ID 生成和动态响应代码状态)。
- 在[Mongo 嗅探过滤器](#)中支持 Mongo 3.2。
- 很多其它没有列出的小的修复和强化。

1.0.0 (2016年9月12日)

第一版开源发布。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 14:30:49

入门指南

本章节会介绍非常简单的配置，并提供一些示例。

Envoy 目前不提供单独的预先编译好的二进制文件，但提供了 Docker 镜像。这是开始使用 Envoy 的最快方式。如果您希望在 Docker 容器外使用 Envoy，则需要[构建它](#)。

这些示例使用 [v2 Envoy API](#)，但仅使用 API 的静态配置功能，这对于简单的需求非常有用。更复杂的需求是由[动态配置](#)来支持的。

快速开始运行简单示例

根据 Envoy 存储库中的文件运行这些命令。下面的部分给出了配置文件和执行步骤更详细的解释。

[configs/google_com_proxy.v2.yaml](#) 中提供了一个非常简单的可用于验证基于纯 HTTP 代理的 Envoy 配置。这不表示实际的 Envoy 部署：

```
$ docker pull envoyproxy/envoy:latest
$ docker run --rm -d -p 10000:10000 envoyproxy/envoy:latest
$ curl -v localhost:10000
```

使用的 Docker 镜像将包含最新版本的 Envoy 和一个基本的 Envoy 配置。此基本配置告诉 Envoy 将入站请求路由到 *.google.com。

简单的配置

Envoy 通过 YAML 文件中传入的参数来进行配置。

[admin message](#) 是 administration 服务必须的配置。[address](#) 键指定监听地址，下面的例子监听地址是 0.0.0.0:9901。

```
admin:
  access_log_path: /tmp/admin_access.log
  address:
    socket_address: { address: 0.0.0.0, port_value: 9901 }
```

[static_resources](#) 包含 Envoy 启动时静态配置的所有内容，而不是 Envoy 在运行时动态配置的资源。[v2 API Overview](#) 描述了这一点。

```
static_resources:
```

[listeners](#) 规范

```
listeners:
- name: listener_0
  address:
    socket_address: { address: 0.0.0.0, port_value: 10000 }
  filter_chains:
  - filters:
    - name: envoy.http_connection_manager
      config:
        stat_prefix: ingress_http
        codec_type: AUTO
        route_config:
```

```

name: local_route
virtual_hosts:
- name: local_service
  domains: ["*"]
  routes:
  - match: { prefix: "/" }
    route: { host_rewrite: www.google.com, cluster: service_google }
http_filters:
- name: envoy.router

```

clusters 规范

```

clusters:
- name: service_google
  connect_timeout: 0.25s
  type: LOGICAL_DNS
# Comment out the following line to test on v6 networks
dns_lookup_family: V4_ONLY
lb_policy: ROUND_ROBIN
hosts: [{ socket_address: { address: google.com, port_value: 443 }}]
tls_context: { sni: www.google.com }

```

使用 Envoy Docker 镜像

创建一个简单的 Dockerfile 来执行 Envoy，假定 `envoy.yaml`（如上所述）位于本地目录中。您可以参考[命令行选项](#)。

```

FROM envoyproxy/envoy:latest
COPY envoy.yaml /etc/envoy/envoy.yaml

```

使用以下命令构建您配置的 Docker 镜像：

```
$ docker build -t envoy:v1
```

现在您可以执行它：

```
$ docker run -d --name envoy -p 9901:9901 -p 10000:10000 envoy:v1
```

最后测试使用：

```
$ curl -v localhost:10000
```

如果您想通过 docker-compose 使用 envoy，则可以使用 volume 覆盖提供的配置文件。

Sandbox

我们使用 Docker Compose 创建了许多 sandbox，这些 sandbox 设置了不同的环境来测试 Envoy 的功能并显示示例配置。当我们觉得人们更有兴趣时，将添加和展示更多不同特征的 sandbox。以下 sandbox 可用：

- [Front Proxy](#)
- [Zipkin Tracing](#)
- [Jaeger Tracing](#)
- [Jaeger Native Tracing](#)
- [gRPC Bridge](#)

其他用例

除代理本身之外， Envoy 还被几个特定用例捆绑为开源发行版的一部分。

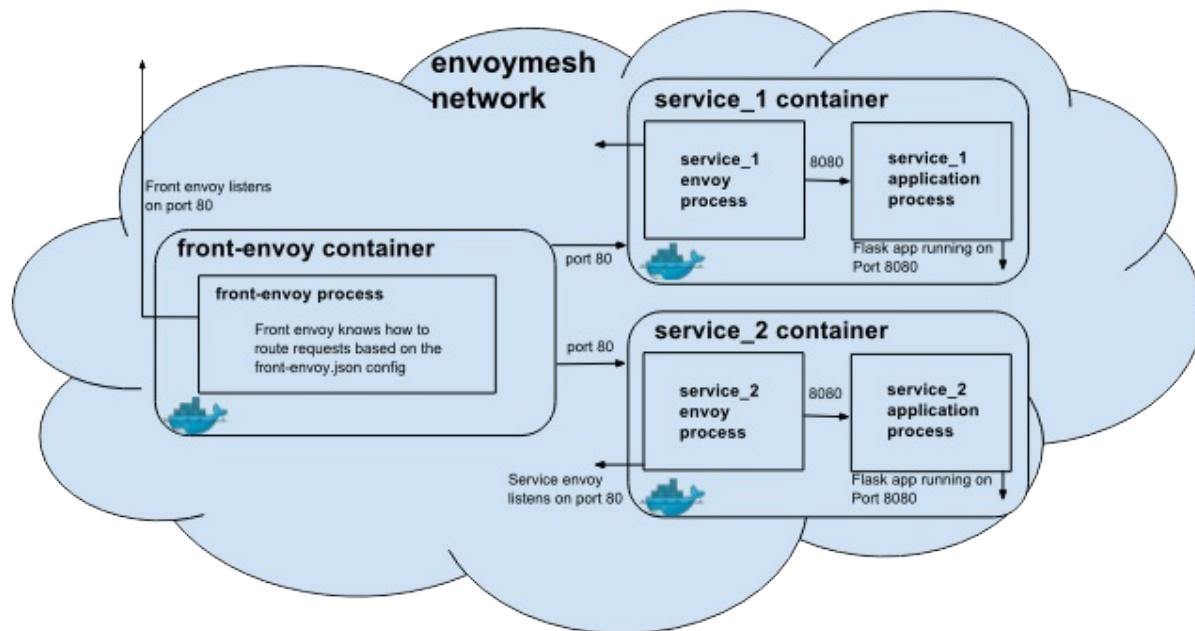
- [Envoy as an API Gateway in Kubernetes](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-17 13:29:48

前端代理

为了帮助大家了解如何使用 Envoy 作为前端代理，我们发布了一个 `docker compose` 沙箱，该沙箱中部署了一个前端 envoy 以及与服务 envoy 搭配的一组服务（简单的沙箱应用）。这三个容器将被部署在名为 `envoymesh` 的虚拟网络中。

下面是使用 `docker compose` 部署的架构图：



所有传入的请求都通过前端 envoy 进行路由，该 envoy 充当位于 `envoymesh` 网络边缘的反向代理。通过 `docker compose` 将端口 80 映射到 8000 端口（请参阅 [/examples/front-proxy/docker-compose.yml](#)）。此外，请注意，由前端 envoy 由到服务容器的所有流量实际上路由到服务 envoy（在 [/examples/front-proxy/front-envoy.yaml](#) 中设置的路由）。反过来，服务 envoy 通过回环地址（[/examples/front-proxy/service-envoy.yaml](#) 中的路由设置）将请求路由到 flask 应用程序。此设置说明了运行服务 envoy 与您的服务搭配的优势：所有请求都由服务 envoy 处理，并有效地路由到您的服务。

运行 Sandbox

以下文档通过按照上图中所述组织的 envoy 集群的设置运行。

步骤 1：安装 Docker

确保您已经安装了最新版本的 `docker`、`docker-compose` 和 `docker-machine`。

安装这些软件最简单的方式是使用 [Docker Toolbox](#)。

步骤 2：配置 Docker Machine

首先创建一个容纳容器的新机器：

```
$ docker-machine create --driver virtualbox default
$ eval $(docker-machine env default)
```

步骤 3：克隆 Envoy repo，启动所有的容器

如果您还没有克隆 envoy repo，执行 `git clone git@github.com:envoyproxy/envoy` 或者 `git clone https://github.com/envoyproxy/envoy.git` 来克隆。

```
$ pwd
envoy/examples/front-proxy
$ docker-compose up --build -d
$ docker-compose ps
      Name           Command    State     Ports
----- 
example_service1_1   /bin/sh -c /usr/local/bin/ ... Up        80/tcp
example_service2_1   /bin/sh -c /usr/local/bin/ ... Up        80/tcp
example_front-envoy_1 /bin/sh -c /usr/local/bin/ ... Up        0.0.0.0:8000->80/tcp, 0.0.0.0:8001->8001/tcp
```

步骤 4：测试 Envoy 的路由能力

您现在可以通过前端 envoy 向两项服务发送请求。

向 service1：

```
$ curl -v $(docker-machine ip default):8000/service/1
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:39:19 GMT
< x-envoy-protocol-version: HTTP/1.1
<
Hello from behind Envoy (service 1)! hostname: f26027f1ce28 resolvedhostname: 172.19.0.6
* Connection #0 to host 192.168.99.100 left intact
```

向 service2：

```
$ curl -v $(docker-machine ip default):8000/service/2
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/2 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 2
< server: envoy
< date: Fri, 26 Aug 2016 19:39:23 GMT
< x-envoy-protocol-version: HTTP/1.1
<
Hello from behind Envoy (service 2)! hostname: 92f4a3737bbc resolvedhostname: 172.19.0.2
* Connection #0 to host 192.168.99.100 left intact
```

请注意，每个请求在发送给前端 envoy 时已正确路由到相应的应用程序。

步骤 5：测试 Envoy 的负载均衡能力

Now let's scale up our service1 nodes to demonstrate the clustering abilities of envoy.:

现在扩展我们的 service1 节点来演示 envoy 的集群能力：

```
$ docker-compose scale service1=3
Creating and starting example_service1_2 ... done
Creating and starting example_service1_3 ... done
```

现在，如果我们多次向 service1 发送请求，前端 envoy 将通过循环轮询三台 service1 机器来负载均衡请求：

```
$ curl -v $(docker-machine ip default):8000/service/1
* Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:40:21 GMT
< x-envoy-protocol-version: HTTP/1.1
<

Hello from behind Envoy (service 1)! hostname: 85ac151715c6 resolvedhostname: 172.19.0.3
* Connection #0 to host 192.168.99.100 left intact
$ curl -v $(docker-machine ip default):8000/service/1
* Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:40:22 GMT
< x-envoy-protocol-version: HTTP/1.1
<

Hello from behind Envoy (service 1)! hostname: 20da22cf955 resolvedhostname: 172.19.0.5
* Connection #0 to host 192.168.99.100 left intact
$ curl -v $(docker-machine ip default):8000/service/1
* Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /service/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:40:24 GMT
< x-envoy-protocol-version: HTTP/1.1
<

Hello from behind Envoy (service 1)! hostname: f26027f1ce28 resolvedhostname: 172.19.0.6
* Connection #0 to host 192.168.99.100 left intact
```

步骤 6：进入容器并 curl 服务

In addition of using `curl` from your host machine, you can also enter the containers themselves and `curl` from inside them. To enter a container you can use `docker-compose exec <container_name> /bin/bash`. For example we can enter the `front-envoy` container, and `curl` for services locally:

除了使用主机上的 `curl` 外，您还可以进入容器并从容器里面 `curl`。要进入容器，可以使用 `docker-compose exec <容器名> /bin/bash`。例如，我们可以进入前端 `envoy` 容器，并在本地 `curl` 服务：

```
$ docker-compose exec front-envoy /bin/bash
root@81288499f9d7:/# curl localhost:80/service/
Hello from behind Envoy (service 1)! hostname: 85ac151715c6 resolvedhostname: 172.19.0.3
root@81288499f9d7:/# curl localhost:80/service/
Hello from behind Envoy (service 1)! hostname: 20da22cfc955 resolvedhostname: 172.19.0.5
root@81288499f9d7:/# curl localhost:80/service/
Hello from behind Envoy (service 1)! hostname: f26027f1ce28 resolvedhostname: 172.19.0.6
root@81288499f9d7:/# curl localhost:80/service/
Hello from behind Envoy (service 2)! hostname: 92f4a3737bbc resolvedhostname: 172.19.0.2
```

步骤7：进入容器并 curl admin

当 `envoy` 运行时，它也会将 `admin` 连接到所需的端口。在示例配置 `admin` 绑定到 8001 端口。我们可以 `curl` 它获得有用的信息。例如，您可以 `curl /server_info` 来获取正在运行的 `envoy` 版本的信息。此外，你可以 `curl /stats` 来获得统计数据。例如在 `frontenvoy` 里面我们可以得到：

```
$ docker-compose exec front-envoy /bin/bash
root@e654c2c83277:/# curl localhost:8001/server_info
envoy 10e00b/RELEASE live 142 142 0
root@e654c2c83277:/# curl localhost:8001/stats
cluster.service1.external.upstream_rq_200: 7
...
cluster.service1.membership_change: 2
cluster.service1.membership_total: 3
...
cluster.service1.upstream_cx_http2_total: 3
...
cluster.service1.upstream_rq_total: 7
...
cluster.service2.external.upstream_rq_200: 2
...
cluster.service2.membership_change: 1
cluster.service2.membership_total: 1
...
cluster.service2.upstream_cx_http2_total: 1
...
cluster.service2.upstream_rq_total: 2
...
```

请注意，我们可以获取上游集群的成员数量，它们实现的请求数量，有关 http 入口的信息以及大量其他有用的统计信息。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-16 22:33:38

Zipkin 追踪

Zipkin 追踪 sandbox 使用 Zipkin 作为追踪提供者演示 Envoy 的请求追踪 功能。此沙箱与上述前端代理体系结构非常相似，但有一点不同：在返回响应之前，service1 会对 service2 进行 API 调用。这三个容器将被部署在名为 `envoymesh` 的虚拟网络中。

所有传入的请求都通过前端 envoy 进行路由，该 envoy 充当位于 `envoymesh` 网络边缘的反向代理。端口 80 通过 docker compose 映射到端口 8000（参见 [/examples/zipkin-tracing/docker-compose.yml](#)）。请注意，所有 envoy 都配置为收集请求跟踪（例如 `http_connection_manager/config/tracing` 中的设置 [/examples/zipkin-tracing/front-envoy-zipkin.yaml](#)）并设置为将 Zipkin 追踪器生成的跨度传播到 Zipkin 集群中（跟踪驱动程序设置 [/examples/zipkin-tracing/front-envoy-zipkin.yaml](#)）。

在将请求路由到相应的服务 envoy 或应用程序之前，Envoy 将负责为跟踪生成适当的跨度（父/子/共享上下文跨度）。在高层次上，每个跨度记录上游API调用的延迟以及将跨度与其他相关跨度（例如跟踪ID）关联所需的信息。

从 Envoy 进行跟踪的最重要的好处之一是，它将负责将跟踪传播到 Zipkin 服务群集。但是，为了充分利用跟踪，应用程序必须传播 Envoy 生成的跟踪标头，同时调用其他服务。在我们提供的沙箱中，简单的应用程序（请参阅 [/examples/front-proxy/service.py](#)）作为 service1 传播跟踪头，同时对 service2 进行出站呼叫。

运行 Sandbox

以下文档通过按照上图中所述组织的 envoy 集群的设置运行。

第1步：构建 sandbox

要构建这个沙盒示例，并启动示例应用程序，请运行以下命令：

```
$ pwd
envoy/examples/zipkin-tracing
$ docker-compose up --build -d
$ docker-compose ps
      Name           Command     State    Ports
----- 
zipkintracing_service1_1   /bin/sh -c /usr/local/bin/ ... Up        80/tcp
zipkintracing_service2_1   /bin/sh -c /usr/local/bin/ ... Up        80/tcp
zipkintracing_front-envoy_1 /bin/sh -c /usr/local/bin/ ... Up        0.0.0.0:8000->80/tcp, 0.0.0.0:8001->8001/tcp
```

第2步：生成一些负载

您现在可以通过前台特使向 service1 发送请求，如下所示：

```
$ curl -v $(docker-machine ip default):8000/trace/1
* Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /trace/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:39:19 GMT
< x-envoy-protocol-version: HTTP/1.1
<
```

```
Hello from behind Envoy (service 1)! hostname: f26027f1ce28 resolvedhostname: 172.19.0.6
* Connection #0 to host 192.168.99.100 Left intact
```

第3步：在 Zipkin UI 中查看追踪

使用您的浏览器访问 <http://localhost:9411>。你应该看到 Zipkin 仪表板。如果这个 IP 地址不正确，你可以运行：
\$ docker-machine ip default 来找到正确的地址。, 将服务设置为“前台代理”，并将开始时间设置为在测试开始前几分钟（步骤2）并按回车。你应该看到前端代理的痕迹。单击一个跟踪来探索从前端代理到service1到service2的请求所采用的路径，以及每个跃点产生的延迟。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-18 23:43:49

Jaeger 追踪

Jaeger 追踪沙箱展示了 Envoy 的 [请求追踪](#) 能力，它使用 Jaeger 作为追踪的提供者。此沙箱与上述前端代理体系结构非常相似，但有一点不同：在返回响应之前，service1 会对 service2 进行一次 API 调用。这三个容器将被部署在名为 `envoymesh` 的虚拟网络中。（注意：沙箱只能在 x86-64 上运行）。

所有传入的请求都通过前端 envoy 进行路由，该 envoy 充当位于 `envoymesh` 网络边缘的反向代理。端口 `80` 被 docker compose 映射到端口 `8000`（参见[/examples/jaeger-native-tracing/docker-compose.yml](#)）。请注意，所有的 envoy 都被配置为收集请求跟踪信息（例如，[/examples/jaeger-tracing/front-envoy-jaeger.yaml](#) 中配置的 `http_connection_manager/config/tracing`），并将 Jaeger 追踪器生成的 span 传播到 Jaeger 集群中（跟踪驱动在 [/examples/jaeger-tracing/front-envoy-jaeger.yaml](#) 中配置）。

在将请求路由到相应的服务 envoy 或应用之前，Envoy 将负责为追踪生成恰当的 span（父/子上下文 span）。在高层次上，每个 span 将记录上游 API 调用的延迟以及将该 span 与其他相关 span 进行关联所需的信息（例如 trace ID）。

Envoy 追踪最重要的好处之一是它将负责将追踪行为传播到 Jaeger 服务集群中。然而，为了充分利用追踪机制，在对其余服务进行请求时，应用必须传播 Envoy 生成的追踪 header。在我们提供的沙箱中，一个简单的 flask 应用（请参阅追踪函数 [/examples/front-proxy/service.py](#)）将作为 service1，在对外请求 service2 时传播追踪 header。

运行沙箱

以下文档按照上图描述对一个 envoy 集群的配置过程进行了演练。

步骤1：建立沙箱

要构建这个沙箱示例并启动示例应用程序，请运行以下命令：

```
$ pwd
envoy/examples/jaeger-tracing
$ docker-compose up --build -d
$ docker-compose ps
      Name           Command       State    Ports
----- 
jaegertracing_service1_1   /bin/sh -c /usr/local/bin/ ... Up        80/tcp
jaegertracing_service2_1   /bin/sh -c /usr/local/bin/ ... Up        80/tcp
jaegertracing_front-envoy_1 /bin/sh -c /usr/local/bin/ ... Up        0.0.0.0:8000->80/tcp, 0.0.0.0:8001->8001/tcp
```

步骤2：生成一些负载

您现在可以通过前端 envoy (front-envoy) 向 service1 发送请求，如下所示：

```
$ curl -v $(docker-machine ip default):8000/trace/1
*   Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /trace/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:39:19 GMT
< x-envoy-protocol-version: HTTP/1.1
<
```

```
Hello from behind Envoy (service 1)! hostname: f26027f1ce28 resolvedhostname: 172.19.0.6
* Connection #0 to host 192.168.99.100 Left intact
```

步骤3：在 Jaeger UI 中查看追踪

在浏览器中打开 <http://localhost:16686>。您应该可以看到 Jaeger 仪表盘。设置服务为“front-proxy”并点击‘Find Traces’。您应该可以看到从 front-proxy 发起的追踪信息。请单击一个追踪来查看从 front-proxy 到 service1 再到 service2 的请求路径，以及每一跳产生的延迟。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 15:51:50

Jaeger 原生追踪

Jaeger 追踪沙箱展示了 Envoy 的 [请求追踪](#) 能力，它使用 Jaeger 作为追踪的提供者，并且使用 Jaeger 的原生 C++ 客户端作为插件。使用 Jaeger 及其原生客户端替代 Envoy 内置的 Zipkin 客户端有以下优势：

- Jaeger 可使追踪传播（trace propagation）与其他服务一起工作而无需进行配置 [更改](#)。
- 可以使用各种不同的 [采样策略](#)，包括可以从 Jaeger 后端集中控制的概率采样策略和远程采样策略。
- Spans 以更高效的二进制编码发送到采集器。

此沙箱与上述前端代理体系结构非常相似，但有一点不同：在返回响应之前，service1 会对 service2 进行一次 API 调用。这三个容器将被部署在名为 `envoymesh` 的虚拟网络中。（注意：沙箱只能在 x86-64 上运行）。

所有传入的请求都通过前端 envoy 进行路由，该 envoy 充当位于 `envoymesh` 网络边缘的反向代理。端口 `80` 被 docker compose 映射到端口 `8000`（参见[/examples/jaeger-native-tracing/docker-compose.yml](#)）。请注意，所有的 envoy 都被配置为收集请求跟踪信息（例如，[/examples/jaeger-native-tracing/front-envoy-jaeger.yaml](#) 中配置的 `http_connection_manager/config/tracing`），并将 Jaeger 追踪器生成的 span 传播到 Jaeger 集群中（跟踪驱动在 [/examples/jaeger-native-tracing/front-envoy-jaeger.yaml](#) 中配置）。

在将请求路由到相应的服务 envoy 或应用之前，Envoy 将负责为追踪生成恰当的 span（父/子上下文 span）。在高层次上，每个 span 将记录上游 API 调用的延迟以及将该 span 与其他相关 span 进行关联所需的信息（例如 trace ID）。

Envoy 追踪最重要的好处之一是它将负责将追踪行为传播到 Jaeger 服务集群中。然而，为了充分利用追踪机制，在对其余服务进行请求时，应用必须传播 Envoy 生成的追踪 header。在我们提供的沙箱中，一个简单的 flask 应用（请参阅追踪函数 [/examples/front-proxy/service.py](#)）将作为 service1，在对外请求 service2 时传播追踪 header。

运行沙箱

以下文档按照上图描述对一个 envoy 集群的配置过程进行了演练。

步骤1：建立沙箱

要构建这个沙箱示例并启动示例应用程序，请运行以下命令：

```
$ pwd
envoy/examples/jaeger-native-tracing
$ docker-compose up --build -d
$ docker-compose ps
      Name           Command     State    Ports
----- 
jaegertracing_service1_1   /bin/sh -c /usr/local/bin/ ... Up        80/tcp
jaegertracing_service2_1   /bin/sh -c /usr/local/bin/ ... Up        80/tcp
jaegertracing_front-envoy_1 /bin/sh -c /usr/local/bin/ ... Up        0.0.0.0:8000->80/tcp, 0.0.0.0:8001->8001/tcp
```

步骤2：生成一些负载

您现在可以通过前端 envoy (front-envoy) 向 service1 发送请求，如下所示：

```
$ curl -v $(docker-machine ip default):8000/trace/1
* Trying 192.168.99.100...
* Connected to 192.168.99.100 (192.168.99.100) port 8000 (#0)
> GET /trace/1 HTTP/1.1
> Host: 192.168.99.100:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: text/html; charset=utf-8
```

```
< content-length: 89
< x-envoy-upstream-service-time: 1
< server: envoy
< date: Fri, 26 Aug 2016 19:39:19 GMT
< x-envoy-protocol-version: HTTP/1.1
<
Hello from behind Envoy (service 1)! hostname: f26027f1ce28 resolvedhostname: 172.19.0.6
* Connection #0 to host 192.168.99.100 left intact
```

步骤3：在 Jaeger UI 中查看追踪

在浏览器中打开 <http://localhost:16686>。您应该可以看到 Jaeger 仪表盘。设置服务为 “front-proxy” 并点击 ‘Find Traces’。您应该可以看到从 front-proxy 发起的追踪信息。请单击一个追踪来查看从 front-proxy 到 service1 再到 service2 的请求路径，以及每一跳产生的延迟。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 15:51:00

gRPC 网桥

Envoy gRPC

gRPC 网桥沙箱是 Envoy 的 [gRPC 网桥过滤器](#)的一个实例。包含在沙箱中的是带有 Python HTTP 客户端的gRPC 内存键/值存储。Python客户端通过 Envoy sidecar 进程发出 HTTP/1请求，并将其升级为 HTTP/2 gRPC 请求。然后响应 trailer 被缓冲并作为 HTTP/1 标头的有效载荷发送回客户端。

本例中演示的 Envoy 的另一个功能是通过其路由配置执行权威基础路由。

构建 Go 服务

运行下面的命令构建 Go gRPC 服务：

```
$ pwd
envoy/examples/grpc-bridge
$ script/bootstrap
$ script/build
```

注意： `build` 命令要求 Envoy 代码库（或其工作副本）位于 `$GOPATH/src/github.com/envoyproxy/envoy`。

Docker compose

运行 docker compose 文件，同时运行 Python 和 gRPC 容器：

```
$ pwd
envoy/examples/grpc-bridge
$ docker-compose up --build
```

向键/值存储发送请求

使用 Python 服务，发送 gRPC 请求：

```
$ pwd
envoy/examples/grpc-bridge
# set a key
$ docker-compose exec python /client/client.py set foo bar
setf foo to bar

# get a key
$ docker-compose exec python /client/client.py get foo
bar

# modify an existing key
$ docker-compose exec python /client/client.py set foo baz
setf foo to baz

# get the modified key
$ docker-compose exec python /client/client.py get foo
baz
```

在运行的 docker-compose 容器中，您应该可以看到 gRPC 服务打印的活动的记录：

```
grpc_1    | 2017/05/30 12:05:09 set: foo = bar
grpc_1    | 2017/05/30 12:05:12 get: foo
grpc_1    | 2017/05/30 12:05:18 set: foo = baz
```

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-26 11:14:08

Envoy 作为 Kubernetes 的 API 网关

使用 Ambassador 的一个常见场景是将其部署为 Kubernetes 的 edge 服务（API 网关）。Ambassador 是开源 Envoy 的分布式版本，专门为 kubernetes 设计的。

本例将介绍如何通过 Ambassador 在 Kubernetes 上部署 Ambassador。

部署 Ambassador

Ambassador 的设置是通过 kubernetes 部署的。为了在 kubernetes 安装 Ambassador/Envoy，如果你的集群启动了 RBAC：

```
kubectl apply -f https://www.getambassador.io/yaml/ambassador/ambassador-rbac.yaml
```

如果您没启动 RBAC：

```
kubectl apply -f https://www.getambassador.io/yaml/ambassador/ambassador-no-rbac.yaml
```

上面的 YAML 将会为 Ambassador 创建 kubernetes 部署，包含 readiness 和 liveness 检查。默认，将会创建3个 Ambassador 实例。每一个 Ambassador 实例包含一个 Envoy 代理以及一个 Ambassador 控制器。

我们现在需要创建一个 Kubernetes 服务来指向 Ambassador 的部署，我们将使用 `LoadBalancer` 服务。如果你的集群不支持 `LoadBalancer` 服务，你需要改成 `NodePort` 或者 `ClusterIP`。

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    service: ambassador
  name: ambassador
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    service: ambassador
```

将上面的 YAML 文件保存成 `ambassador-svc.yaml` 文件。然后将这个服务部署到 kubernetes：

```
kubectl apply -f ambassador-svc.yaml
```

这时候 Envoy 和 Ambassador 控制器已经在你的集群上运行。

配置 Ambassador

Ambassador 使用 Kubernetes 注解来添加或删除配置。这个示例 YAML 将添加一条到 Google 的路由，类似于[入门指南](#)中的基本配置示例。

```
---
apiVersion: v1
```

```

kind: Service
metadata:
  name: google
  annotations:
    getambassador.io/config: |
      ---
      apiVersion: ambassador/v0
      kind: Mapping
      name: google_mapping
      prefix: /google/
      service: https://google.com:443
      host_rewrite: www.google.com
spec:
  type: ClusterIP
  clusterIP: None

```

保存上面的文件，命名为 `google.yaml`。然后运行：

```
kubectl apply -f google.yaml
```

Ambassador 将发现您的 Kubernetes 注解的更改，并添加到 Envoy 的路由。注意，我们在这个例子中使用了一个虚拟服务；通常，您会将注解与真正的 Kubernetes 服务关联起来。

测试映射

您可以通过获得 Ambassador 服务的外部 IP 地址来测试这个映射，然后通过 `curl` 发送请求：

```

$ kubectl get svc ambassador
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
ambassador  10.19.241.98  35.225.154.81  80:32491/TCP  15m
$ curl -v 35.225.154.81/google/

```

更多

Ambassador 在上公开了多个 Envoy 的特性映射，比如 CORS、加权循环调度算法、gRPC、TLS 和超时设定。要了解更多信息，请阅读[配置文档](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-17 19:50:36

构建

Envoy 构建系统使用了 Bazel。为了简化初始构建和快速启动，我们提供了一个基于 Ubuntu 16 的 docker 容器，它内部包含了所需的所有东西用来构建和静态链接 envoy，详参 [ci/README.md](#)

为了手动创建，遵循 [bazel/README.md](#) 的说明。

要求

Envoy 最初是在 Ubuntu 14 LTS 上开发和部署的。它应该适用于任何最近的 Linux，包括 Ubuntu 16 LTS。

构建 Envoy 有以下要求：

- GCC 5+ (对 C++14 支持)。
- 这些 [预先构建](#) 的第三方依赖。
- 这些 [Bazel native](#) 的依赖。

请阅读 [CI](#) 和 [Bazel](#) 的文档。获取有关执行手动构建的更多信息。

预构建的二进制文件

在每一个 master 提交中，我们创建了一组轻量级 Docker 镜像，其中包含 Envoy 的二进制文件。我们在发布官方版本时也会使用发布版本号来标记 docker 镜像。

- [envoyproxy/envoy](#): 发布在 Ubuntu Xenial 基础之上带有标记的二进制文件
- [envoyproxy/envoy-alpine](#): 发布带有在 glibc 基础之上标记的二进制文件
- [envoyproxy/envoy-alpine-debug](#): 发布带有在 glibc 基础之上 debug 标记的二进制文件

我们将考虑在帮助 CI、包等方面创建额外的二进制类型，如果需要的话，请在 GitHub 上打开一个 [issue](#)。

修改 Envoy

如果你对修改 Envoy 和测试你的改变感兴趣，那么一种方法就是使用 Docker。本指南将介绍构建您自己的 Envoy 二进制文件的过程，并将二进制文件放入一个 Ubuntu 容器中。

- [构建 Envoy Docker 镜像](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-18 10:34:31

构建 Envoy Docker 镜像

下面的步骤将指导您构建自己的 Envoy 二进制文件，并将其放入一个干净的 Ubuntu 容器中。

第一步：构建 Envoy

使用 `envoyproxy/envoy-build` 你可以编译 Envoy。这个镜像包含你构建 Envoy 所有所需的软件。在你的 Envoy 文件中：

```
$ pwd  
src/envoy  
$ ./ci/run_envoy_docker.sh './ci/do_ci.sh bazel.release'
```

这个命令需要一些时间才能运行，因为它正在编译一个 Envoy 二进制文件并运行测试。

有关构建和不同构建目标的更多信息，请参阅 [repose:ci/README.md](#)。

第二步：只使用 envoy 的二进制文件构建镜像

在这一步中，我们将构建一个只有 Envoy 二进制的镜像，而没有一个软件用于构建它：

```
$ pwd  
src/envoy/  
$ docker build -f ci/Dockerfile-envoy-image -t envoy .
```

现在如果您更改了任何 Dockerfile 中的 `FROM` 行，您可以使用这个 `envoy` 镜像构建任何沙箱。

如果您对修改和测试 Envoy 感兴趣，那么这将特别有用。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-23 11:38:36

参考配置

源代码分发包里为以下三种 Envoy 的主要部署类型准备了一整套的配置案例模版：

- 服务间
- 前端代理
- 双向代理

这一套的配置案例模版主要用来展现 Envoy 在复杂部署下的全部能力。并不是所有的功能点都能适用于所有的应用场景。详细文档可查看[配置参考](#)。

配置生成器

Envoy 的配置开始变得越发复杂。在 Lyft 我们用 [jinja](#) 模版让生产以及管理配置的工作变得轻松一些。源代码分发包里便包含了其中一个版本的配置生成器，这个版本的配置生成器非常接近我们在 Lyft 使用的版本。我们同时为上述的三种场景都准备了相应的范例配置模版。

- 脚本生成器: [configs/configgen.py](#)
- 服务间模版: [configs/envoy_service_to_service.template.json](#)
- 前端代理模版: [configs/envoy_front_proxy.template.json](#)
- 双向代理模版: [configs/envoy_double_proxy.template.json](#)

可以从 repo 的根目录运行以下命令以生成相关的范例配置：

```
mkdir -p generated/configs
bazel build //configs:example_configs
tar xvf $PWD/bazel-genfiles/configs/example_configs.tar -C generated/configs
```

上面的命令将生成三个完全可扩展配置，而配置中所使用到的其中某些变量被定义在 configgen.py 里。可以查看 configgen.py 里的注释以学习如何让不同的扩展工作。

关于范例配置，在此分享一些笔记：

- 一个假定运行在 discovery.yourcompany.net 上的[服务发现](#)实例。
- yourcompany.net 的 DNS 做了许多配置。可在配置模版中查找基于其实现的各种实例。
- 为 [LightStep](#) 而配置的追踪。为了禁止或启用 Zipkin <http://zipkin.io> 追踪，而删除或改变相应的[追踪配置](#)。
- 用于演示如何使用[全局速率限制服务](#)的示例配置。可以通过删除速率限制配置以禁用此服务。
- 为服务间参考配置而设定的[路由发现服务](#)，此服务假定运行在 rds.yourcompany.net 上。
- 为服务间参考配置而设定的[集群发现服务](#)，此服务假定运行在 cds.yourcompany.net 上。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:26:04

配置负载检查工具

配置负载检查工具检查 JSON 格式的配置文件是否使用有效的 JSON 编写，并符合 Envoy JSON 模式。该工具利用 `test/config_test/config_test.cc` 中的配置测试。该测试加载 JSON 配置文件并使用它运行服务器配置初始化。

- 输入

该工具需要一个指向保存 JSON Envoy 配置文件的根目录的 PATH 变量。该工具将以递归方式遍历文件系统树，并对每个找到的文件运行配置测试。请记住，该工具将尝试加载路径中找到的所有文件。

- 输出

该工具使用它当前正在测试的配置初始化服务器配置时，将输出 Envoy 日志。如果存在 JSON 文件格式错误或不符合 Envoy JSON 模式的配置文件，则该工具将以状态 `EXIT_FAILURE` 退出。如果该工具成功加载找到的所有配置文件，它将以状态 `EXIT_SUCCESS` 退出。

- 构建

我们可以使用 Bazel 在本地构建该工具。
`bazel build //test/tools/config_load_check:config_load_check_tool`

- 运行

该工具将需要如上所述的 PATH 变量。
`bazel bin/test/tools/config_load_check/config_load_check_tool PATH`

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 10:41:37

路由表检查工具

路由表检查工具检查路由返回的路由参数是否符合预期。该工具还可以用于检查路径重定向、路径重写或主机重写是否符合预期。

- 输入

该工具期望两个输入 JSON 文件：一个路由配置 JSON 文件。在一个工具配置 JSON 文件中找到了路由配置 JSON 文件架构。配置 JSON 文件模板的工具在 [config](#) 中。工具配置输入文件指定 URL（由权限和路径组成）以及期望的路由参数值。其他参数（如附加标头）是可选的。

- 输出

如果任何测试用例与预期的路由参数值不匹配，那么程序将以状态 EXIT_FAILURE 退出。`--detail` 选项打印出每个测试的详细信息。第一行表示测试名称。如果测试失败，则打印失败的测试用例的详细信息。第一个字段是预期路由参数值。第二个字段是实际路由参数值。第三个字段表示比较的参数。在下面的例子中，`test_2` 和 `test_5` 失败了而其他测试通过了。在失败的测试用例中，会打印冲突细节。`Test_1 Test_2 default other virtual_host_name Test_3 Test_4 Test_5 locations ats cluster_name Test_6` 目前不支持使用有效[运行时值](#)进行测试，这可能会在以后的工作中添加。

- 构建

工具可以在本地使用 Bazel 构建。`bazel build //test/tools/router_check:router_check_tool`

- 运行

该工具接受两个输入 json 文件和一个可选的命令行参数 `--details`。命令行参数的预期顺序是：1. 路由配置 json 文件；2. 工具配置 json 文件；3. 可选项。`bazel-bin/test/tools/router_check/router_check_tool router_config.json tool_config.json bazel-bin/test/tools/router_check/router_check_tool router_config.json tool_config.json --details`

- 测试

bash shell 脚本测试可以使用 `bazel` 运行。该测试比较了使用不同路由和工具配置 json 文件的路由。配置文件可以在 `test/tools/router_check/test/config/...` 找到。`bazel test //test/tools/router_check/...`。

Schema 验证器检查工具

Schema 验证器工具验证被传入的 JSON 符合配置中的某个 schema。为验证整个配置，请参考[配置负载检查工具](#)。当前，仅[路由配置](#) schema 验证被支持。

- 输入

工具期望两个输入：检查传入的 JSON 所用的 schema 类型。对于[路由配置](#)验证被支持的类型是 :route。JSON 所在的路径。

- 输出

如果 JSON 符合 schema，工具将以状态 EXIT_SUCCESS 退出。如果 JSON 不符合 schema，会输出一条错误消息告知不符合 schema 的细节。工具将以 EXIT_FAILURE 状态退出。

- 构建

工具可以在本地使用 Bazel 构建。`bazel build //test/tools/schema_validator:schema_validator_tool`

- 运行

工具采用上面描述的一条路径。`bazel-bin/test/tools/schema_validator/schema_validator_tool --schema-type SCHEMA_TYPE -json-path PATH`

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 10:40:56

v1 API 概览

注意

v1 配置 API 现在被认为是过时了而且宣布了[废弃时间表](#)。请升级并使用[v2 配置 API](#)。

Envoy 配置格式是 JSON 的并用一个 JSON schema 验证。Schema 可以在 [source/common/json/config_schemas.cc](#) 找到。服务器的主配置被包含在监听器和集群管理器部分。其他顶级元素指定各种配置。

为语法方便，也为手写的配置提供了 YAML 支持。如果一个文件路径的结尾是 .yaml，Envoy 将在内部将 YAML 转换为 JSON。在配置文档的剩余部分，我们仅指 JSON。Envoy 期望一个清晰的 YAML 标量，因此，如果一个集群名字（应该是一个字符串）被称为 *true*，它应该在配置 YAML 写为 “*true*”。同样适用于整数和浮点值(即 1 vs. 1.0 vs. “1.0”)。

```
{
  "listeners": [],
  "lds": "...",
  "admin": "...",
  "cluster_manager": "...",
  "flags_path": "...",
  "statsd_udp_ip_address": "...",
  "statsd_tcp_cluster_name": "...",
  "stats_flush_interval_ms": "...",
  "watchdog_miss_timeout_ms": "...",
  "watchdog_megamiss_timeout_ms": "...",
  "watchdog_kill_timeout_ms": "...",
  "watchdog_multikill_timeout_ms": "...",
  "tracing": "...",
  "rate_limit_service": "...",
  "runtime": "...",
}
```

- [listeners](#)

(required, array) 监听器 的数组会被服务器实例化。一个单独的 Envoy 进程可以包含任意数量的监听器。

- [lds](#)

(optional, object) 对监听器发现服务(LDS)的配置。如果没有指定，仅静态监听器被加载。

- [admin](#)

(required, object) 本地管理 HTTP 服务器配置。

- [cluster_manager](#)

(required, object) 集群管理器 配置，它拥有服务器内所有上游集群。

- [flags_path](#)

(optional, string) 搜索[启动标志文件](#)的文件系统路径。

- [statsd_udp_ip_address](#)

(optional, string) 一个运行中的 遵循 statsd 的 监听器的 UDP 地址。如果被指定，[统计数据](#)将被写入到这个地址。IPv4 地址应该具有主机:端口 (例如：127.0.0.1:855)的格式。IPv6 地址应该具有 URL 格式 [主机]:端口 (例如：[::1]:855)。

- [statsd_tcp_cluster_name](#)

(*optional, string*) 一个运行遵循 TCP statsd 的监听器的集群管理器集群的名字。如果被指定，Envoy 将连接到这个集群以写入[统计数据](#)。

- **stats_flush_interval_ms**

(*optional, integer*) 两次写入到配置好的统计池之间以毫秒计的时间。出于性能原因，Envoy 锁定了计数器，仅周期性写入计数器和仪表的数据。如果未指定，默认为 5000ms (5 秒)。

- **watchdog_miss_timeout_ms**

(*optional, integer*) Envoy 在 “server.watchdog_miss” 统计中计数一个无响应的线程后以毫秒计的时间。如果未指定，默认为 200ms。

- **watchdog_megamiss_timeout_ms**

(*optional, integer*) Envoy 在 “server.watchdog_mega_miss” 统计中计数一个无响应的线程后以毫秒计的时间。如果未指定，默认为 1000ms。

- **watchdog_kill_timeout_ms**

(*optional, integer*) 如果一个被观察的线程在这么多微妙的时间内没有响应，则假设一个编程错误并杀掉整个 Envoy 进程。设为 0 以关掉整个行为。如果未指定，默认为 0 (关掉的)。

- **watchdog_multikill_timeout_ms**

(*optional, integer*) 如果至少两个被观察的线程在至少这么多微妙的时间内没有响应，则假设一个真正的死锁并杀掉整个 Envoy 进程。设为 0 以关掉整个行为。如果未指定，默认为 0 (关掉的)。

- **tracing**

(*optional, object*) 外部[追踪](#)提供者的配置。如果未指定，将不执行追踪。

- **rate_limit_service**

(*optional, object*) 外部[速率限制服务](#)提供者的配置。如果未指定，任何对速率限制服务的调用将立即返回成功。

- **runtime**

(*optional, object*) [运行时配置](#) 提供者的配置。如果没有被指定，一个“null”提供者将被使用，这将导致使用所有的默认值。配置

v2 API 概览

Envoy v2 API 被定义为 [data plane API repository](#) 中的 [proto3 Protocol Buffers](#)。其改进了现有的 [v1 API](#) 和概念以支持：

- 通过 gRPC 流式传输对 xDS API 的更新，这减少了资源的需求并且可以降低更新延迟。
- 一种新的 REST-JSON API。其中 JSON/YAML 格式是通过 [proto3 规范的 JSON 映射](#) 机械地派生出来的。
- 通过文件系统、REST-JSON 或 gRPC 端点传递更新。
- 通过扩展端点分配 API 进行高级负载平衡，并向管理服务器报告负载以及资源的利用率。
- 当需要 [更强的一致性和排序属性](#) 时，Envoy v2 API 仍然可以保持基准最终一致性模型。

Envoy与管理服务器之间的关于v2消息交换方面的更多详细信息，请参阅 [xDS协议说明](#)。

Bootstrap 配置

要使用 v2 API，需要提供引导程序配置文件。其提供了静态服务器配置以及根据需要配置 Envoy 以访问[动态配置](#)。与 v1 JSON/YAML 配置一样，可以在命令行通过 `-c` 标志提供，即：

```
./envoy -c <path to config>.{json,yaml,pb,pb_text} --v2-config-only
```

扩展映射底层 v2 配置。`--v2-config-only` 标志并不是严格要求的，因为 Envoy 会自动的尝试监测配置文件的版本，但是在配置解析失败时，该选项可以提供增强的调试体验。

[Bootstrap](#) 消息是配置的根，[Bootstrap](#) 消息中一个关键的概念是静态和动态资源的之间的区别。例如[Listener](#) 或 [Cluster](#) 这些资源可以在[static_resources](#) 静态的获得，或者具有如在 [dynamic_resources](#) 中配置的[LDS](#)或[CDS](#) 之类的 xDS 服务。

示例

下面我们将使用 [YAML](#) 表示的配置原型，以及从 127.0.0.1:10000 到 127.0.0.2:1234 的服务代理 HTTP 的运行示例。

静态

下面提供了一个最小的完全静态引导配置：

```
admin:
  access_log_path: /tmp/admin_access.log
  address:
    socket_address: { address: 127.0.0.1, port_value: 9901 }

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address: { address: 127.0.0.1, port_value: 10000 }
      filter_chains:
        - filters:
            - name: envoy.http_connection_manager
              config:
                stat_prefix: ingress_http
                codec_type: AUTO
                route_config:
                  name: local_route
                  virtual_hosts:
```

```

    - name: local_service
      domains: ["*"]
      routes:
        - match: { prefix: "/" }
          route: { cluster: some_service }
      http_filters:
        - name: envoy.router
  clusters:
    - name: some_service
      connect_timeout: 0.25s
      type: STATIC
      lb_policy: ROUND_ROBIN
      hosts: [{ socket_address: { address: 127.0.0.2, port_value: 1234 }}]

```

除了动态 EDS 大部分静态

下面提供了一个引导配置，该配置从以上示例开始，通过监听 127.0.0.3:5678 的 EDS gRPC管理服务器进行[动态端点发现](#)：

```

admin:
  access_log_path: /tmp/admin_access.log
  address:
    socket_address: { address: 127.0.0.1, port_value: 9901 }

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address: { address: 127.0.0.1, port_value: 10000 }
      filter_chains:
        - filters:
            - name: envoy.http_connection_manager
              config:
                stat_prefix: ingress_http
                codec_type: AUTO
                route_config:
                  name: local_route
                  virtual_hosts:
                    - name: local_service
                      domains: ["*"]
                      routes:
                        - match: { prefix: "/" }
                          route: { cluster: some_service }
                  http_filters:
                    - name: envoy.router
  clusters:
    - name: some_service
      connect_timeout: 0.25s
      lb_policy: ROUND_ROBIN
      type: EDS
      eds_cluster_config:
        eds_config:
          api_config_source:
            api_type: GRPC
            cluster_names: [xds_cluster]
    - name: xds_cluster
      connect_timeout: 0.25s
      type: STATIC
      lb_policy: ROUND_ROBIN
      http2_protocol_options: {}
      hosts: [{ socket_address: { address: 127.0.0.3, port_value: 5678 }}]

```

注意上面 `xds_cluster` 被定义为指向 Envoy 管理服务器。即使在完全动态的配置中，也需要定义一些静态资源，从而将 Envoy 指向其 xDS 管理服务器。

在上面的例子中，EDS 管理服务器可以返回一个[发现响应](#)的 pro 编码：

```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.api.v2.ClusterLoadAssignment
  cluster_name: some_service
  endpoints:
  - lb_endpoints:
    - endpoint:
      address:
        socket_address:
          address: 127.0.0.2
          port_value: 1234
```

上面显示的版本控制和类型 URL 方案在[流式 gRPC 订阅协议](#)文档中有更详细的解释。

动态

下面提供了完全动态的 `bootstrap` 配置，其中属于管理服务器的所有资源都是通过 xDS 发现的：

```
admin:
  access_log_path: /tmp/admin_access.log
  address:
    socket_address: { address: 127.0.0.1, port_value: 9901 }

dynamic_resources:
  lds_config:
    api_config_source:
      api_type: GRPC
      cluster_names: [xds_cluster]
  cds_config:
    api_config_source:
      api_type: GRPC
      cluster_names: [xds_cluster]

static_resources:
  clusters:
  - name: xds_cluster
    connect_timeout: 0.25s
    type: STATIC
    lb_policy: ROUND_ROBIN
    http2_protocol_options: {}
    hosts: [{ socket_address: { address: 127.0.0.3, port_value: 5678 }}]
```

管理服务器可以用 `LDS` 响应 `LDS` 请求：

```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.api.v2.Listener
  name: listener_0
  address:
    socket_address:
      address: 127.0.0.1
      port_value: 10000
  filter_chains:
  - filters:
    - name: envoy.http_connection_manager
      config:
        stat_prefix: ingress_http
        codec_type: AUTO
      rds:
        route_config_name: local_route
        config_source:
          api_config_source:
```

```

api_type: GRPC
cluster_names: [xds_cluster]
http_filters:
- name: envoy.router

```

管理服务器可以用 `RDS` 响应 `RDS` 请求：

```

version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.api.v2.RouteConfiguration
  name: local_route
  virtual_hosts:
  - name: local_service
    domains: ["*"]
    routes:
    - match: { prefix: "/" }
      route: { cluster: some_service }

```

管理服务器可以用 `CDS` 响应 `CDS` 请求：

```

version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.api.v2.Cluster
  name: some_service
  connect_timeout: 0.25s
  lb_policy: ROUND_ROBIN
  type: EDS
  eds_cluster_config:
    eds_config:
      api_config_source:
        api_type: GRPC
        cluster_names: [xds_cluster]

```

管理服务器可以用 `EDS` 请求来响应：

```

version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.api.v2.ClusterLoadAssignment
  cluster_name: some_service
  endpoints:
  - lb_endpoints:
    - endpoint:
      address:
        socket_address:
          address: 127.0.0.2
          port_value: 1234

```

管理服务器

v2 xDS 管理服务器将按照 `gRPC` 和/或 `REST` 服务的要求实现以下端点。在流式 `gRPC` 和 `REST-JSON` 两种情况下，都会发送 `DiscoveryRequest` 并根据[xDS协议](#)接收 [`DiscoveryResponse`]。

gRPC streaming 端点

- POST `/envoy.api.v2.ClusterDiscoveryService/StreamClusters`

有关服务定义，请参考 `cds.proto`。当在 `Bootstrap` 配置的 `dynamic_resources` 中设置为

```
cds_config:
```

```
api_config_source:
  api_type: GRPC
  cluster_names: [some_xds_cluster]
```

时 Envoy 将此用作客户端。

- POST /envoy.api.v2.EndpointDiscoveryService/StreamEndpoints

有关服务定义, 请参阅 [eds.proto](#)。在 `Cluster` 配置的 `eds_cluster_config` 字段中设置为

```
eds_config:
  api_config_source:
    api_type: GRPC
    cluster_names: [some_xds_cluster]
```

时, Envoy 将此用作客户端。

- POST /envoy.api.v2.ListenerDiscoveryService/StreamListeners

有关服务定义, 请参阅 [lds.proto](#)。当在 `Bootstrap` 配置的 `dynamic_resources` 中设置为

```
lds_config:
  api_config_source:
    api_type: GRPC
    cluster_names: [some_xds_cluster]
```

时, Envoy 将此用作客户端。

- POST /envoy.api.v2.RouteDiscoveryService/StreamRoutes

有关服务定义, 请参阅 [rds.proto](#)。当在 `HttpConnectionManager` 配置的 `rds` 字段中设置为

```
route_config_name: some_route_name
config_source:
  api_config_source:
    api_type: GRPC
    cluster_names: [some_xds_cluster]
```

时, Envoy 将它用作客户端。

REST 端点

- POST /v2/discovery:clusters

有关服务定义, 请参阅 [cds.proto](#)。当在 `Bootstrap` 配置的 `[dynamic_resources]` 中设置为

```
cds_config:
  api_config_source:
    api_type: REST
    cluster_names: [some_xds_cluster]
```

时, Envoy 将此用作客户端。

- POST /v2/discovery:endpoints

有关服务定义, 请参阅 [eds.proto](#)。在 `Cluster` 配置的 `eds_cluster_config` 字段中设置

```
eds_config:
  api_config_source:
    api_type: REST
```

```
cluster_names: [some_xds_cluster]
```

时，Envoy 将此用作客户端。

- POST /v2/discovery:listeners

有关服务定义，请参阅 [lds.proto](#)。当在 `Bootstrap` 配置的 `dynamic_resources` 中设置为时，Envoy 将此用作客户端。

```
lds_config:
  api_config_source:
    api_type: REST
  cluster_names: [some_xds_cluster]
```

时，Envoy 将此用作客户端。

- POST /v2/discovery:routes

有关服务定义，请参阅 [rds.proto](#)。当在 `HttpConnectionManager` 配置的 `rds` 字段中设置为

```
route_config_name: some_route_name
config_source:
  api_config_source:
    api_type: REST
  cluster_names: [some_xds_cluster]
```

时，Envoy 将它用作客户端。

聚合发现服务

虽然 Envoy 从根本上采用了最终的一致性模型，但 ADS 提供了对 API 更新推送进行排序并确保单个管理服务器与 Envoy 节点进行 API 更新相关性的机会。ADS 允许管理服务器在一个单一的双向 gRPC 流上传递一个或多个 API 及其资源。没有这些，一些如 RDS 和 EDS 的 API 就可能需要管理多个流并连接到不同的管理服务器。

ADS 将允许通过适当的排序无损的更新配置。例如，假设 `foo.com` 映射到集群 X。我们希望将路由表中的映射更改为集群 Y 中的 `foo.com`。为了做到这一点，必须首先提供包含两个集群 X 和 Y 的 `CDS/EDS` 更新。

如果没有 ADS，CDS/EDS/RDS 流可能指向不同的管理服务器，或者位于同一管理服务器上的不同 gRPC 流和连接需要协调。EDS 资源请求可以分成两个不同的流，一个用于 X，另一个用于 Y。ADS 允许将这些请求合并为单个流到单个管理服务器，避免了分布式同步的需要，以正确地对更新进行排序。依靠 ADS，管理服务器将在单个数据流上提供 CDS、EDS 和 RDS 更新。

ADS 仅适用于 gRPC 流媒体（不是 REST），在[本文档](#)中有更详细的描述。gRPC 的端点是：

- POST /envoy.api.v2.AggregatedDiscoveryService/StreamAggregatedResources

有关服务定义，请参阅 [discovery.proto](#)。当在 `Bootstrap` 配置的 `dynamic_resources` 中设置。

在 `discovery.proto` 查看服务配置。当以下用于 Envoy 的客户端配置

```
ads_config:
  api_type: GRPC
  cluster_names: [some_ads_cluster]
```

时，Envoy 将此用作客户端。

设置此项时，可以将[上述](#)任何配置源设置为使用 ADS 通道。例如，LDS 配置可以从 A

```
lds_config:
```

```
api_config_source:  
  api_type: REST  
  cluster_names: [some_xds_cluster]
```

更改为

```
lds_config: {ads: {}}
```

其效果是 LDS 流将通过共享 ADS 通道指向 *some_ads_cluster*。

管理服务器不可达

当 Envoy 实例失去与管理服务器的连接时，Envoy 将锁定到先前的配置，同时在后台主动重试以重新建立与管理服务器的连接。

Envoy 调试记录了每次尝试连接时都无法与管理服务器建立连接的事实。

[upstream_cx_connect_fail](#) 指向管理服务器的集群等级统计信息提供了用于监视此行为的信号。

状态

除非另有说明，否则将实现 [v2 API参考文档](#)中描述的所有功能。在 v2 API 参考文档和 [v2 API资源库](#)中，所有原型都被冻结，除非它们被标记为草稿或实验。在这里，冻结意味着我们不会破坏线格式兼容性。

Frozen 原型可以进一步的扩充。例如：通过添加新的字段，以不破坏向后兼容性的方式。上述原型中的字段可能会在不再使用相关功能的情况下，随着[违反变更策略](#)而被弃用。尽管 *Frozen* 的 API 保持其格式兼容性，但是保留了更改原名称空间、文件位置以及嵌套关系的权利，这可能会导致代码更改中断。我们的目标是尽量减少流失。

标记为 *draft* 的原型以为这已经接近完成，可能至少部分在 Envoy 中实现，但可能会在冻结之前破坏线路格式。

标记为 *experimental* 的原型与原始草案有相同的警告，并可能在执行和冻结之前做出重大更改。

[这里](#)可以跟踪当前开放的 v2 API 问题。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-23 22:46:32

统计

监听器

每个监听器的统计树根在 `Listener.<address>.`，有如下统计：

名称	类似	描述
downstream_cx_total	Counter	连接总数
downstream_cx_destroy	Counter	销毁连接总数
downstream_cx_active	Gauge	活动连接总数
downstream_cx_length_ms	Histogram	连接长度，单位毫秒
ssl.connection_error	Counter	TLS连接错误总数，不包括证书认证失败
ssl.handshake	Counter	TLS连接握手成功总数
ssl.session_reused	Counter	TLS会话恢复成功总数
ssl.no_certificate	Counter	不带客户端证书的TLS连接成功总数
ssl.fail_no_sni_match	Counter	因为缺少SNI匹配而被拒绝的TLS连接总数
ssl.fail_verify_no_cert	Counter	因为缺少客户端证书而失败的TLS连接总数
ssl.fail_verify_error	Counter	CA认证失败的TLS连接总数
ssl.fail_verify_san	Counter	SAN认证失败的TLS连接总数
ssl.fail_verify_cert_hash	Counter	认证pinning认证失败的TLS连接总数
ssl.cipher.	Counter	使用的TLS连接总数

监听器管理器

监听器管理器的统计树根在 `listener_manager.`，有以下统计。所有 stats 名称中的 `:` 字符被替换为 `_`。

名称	类型	描述
listener_added	Counter	添加的监听器总数（不管是通过静态配置还是 LDS）
listener_modified	Counter	修改过的监听器总数（通过LDS）
listener_removed	Counter	删除过的监听器总数（通过LDS）
listener_create_success	Counter	成功添加到 workers 的监听器对象总数
listener_create_failure	Counter	添加到 workers 失败的监听器对象总数
total_listeners_warming	Gauge	当前热身中的监听器数量
total_listeners_active	Gauge	当前活动中的监听器数量
total_listeners_draining	Gauge	当前排除中的监听器数量

运行时

监听器支持下列运行时设置:

- `ssl.alt_alpn`

使用配置的 `alt_alpn` 协议字符串的请求百分比。默认为0。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:39:25

监听器发现服务 (LDS)

监听器发现服务 (LDS) 是一个可选的 API，Envoy 将调用它来动态获取监听器。Envoy 将协调 API 响应，并根据需要添加、修改或删除已知的监听器。

w监听器更新的语义如下：

- 每个监听器必须有一个独特的[名字](#)。如果没有提供名称，Envoy 将创建一个 UUID。要动态更新的监听器，管理服务必须提供监听器的唯一名称。
- 当一个监听器被添加，在参与连接处理之前，会先进入“预热”阶段。例如，如果监听器引用 [RDS](#) 配置，那么在监听器迁移到“active”之前，将会解析并提取该配置。
- 监听器一旦创建，实际上就会保持不变。因此，更新监听器时，会创建一个全新的监听器（使用相同的侦听套接字）。新增加的监听者都会通过上面所描述的相同“预热”过程。
- 当更新或删除监听器时，旧的监听器将被置于“draining (逐出)”状态，就像整个服务重新启动时一样。监听器移除之后，该监听器所拥有的连接，经过一段时间优雅地关闭（如果可能的话）剩余的连接。逐出时间通过 [--drain-time-s](#) 选项设置。

注意

任何在 Envoy 配置中静态定义的监听器都不能通过 LDS API 进行修改或删除。

配置

- [v1 LDS API](#)
- [v2 LDS API](#)

统计

LDS 的统计树是以 `listener_manager.lds` 为根，统计如下：

名称	类型	描述
<code>config_reload</code>	Counter	由于不同的配置更新，导致配置 API 调用总数
<code>update_attempt</code>	Counter	LDS 配置 API 调用重试总数
<code>update_success</code>	Counter	LDS 配置 API 调用成功总数
<code>update_failure</code>	Counter	LDS 配置 API 调用失败总数（网络或模式错误）
<code>version</code>	Gauge	上次成功调用的内容哈希值

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 14:54:27

原始目的地

原始目的地监听器过滤器读取 SO_ORIGINAL_DST 套接字属性，这个属性在连接被重定向时设置。重定向可以通过 iptables REDIRECT target 或者 iptables TPROXY target 实现，结合使用监听器的 transparent 属性设置。Envoy 中的后续处理将恢复后的目标地址视为连接的本地地址，而不是监听器正在监听的地址。此外，[原始目标集群](#)可用于将 HTTP 请求或 TCP 连接转发到恢复后的目标地址。

- [v2 API 参考](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 14:55:54

TLS 检查器

TLS 检查器监听器筛选器允许检测传输是 TLS 还是明文，如果是 TLS，它将检测来自客户端的[服务器名称指示](#)。这可以用来通过 [FilterChainMatch](#) 的 `sni_domains` 来选择[过滤器链](#)。

- [SNI](#)
- [v2 API 参考](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 14:56:14

客户端 TLS 身份验证

- 客户端 TLS 认证过滤器[架构概览](#)
- [v1 API 参考](#)
- [v2 API 参考](#)

统计

每个配置的客户端 TLS 认证过滤器均有以 `auth.clientssl..` 开头的统计信息，其统计信息如下所示：

名称	类型	描述
<code>update_success</code>	Counter	主体更新成功总次数
<code>update_failure</code>	Counter	主体更新失败总次数
<code>auth_no_ssl</code>	Counter	无 TLS 而被忽略的连接总次数
<code>auth_ip_white_list</code>	Counter	由于 IP 白名单而被允许的连接总次数
<code>auth_digest_match</code>	Counter	由于证书匹配而被允许的连接总次数
<code>auth_digest_no_match</code>	Counter	由于证书不匹配而被拒绝的连接总次数
<code>total_principals</code>	Gauge	已加载主体总数

REST API

`GET /v1/certs/list/approved`

认证过滤器将每隔一段刷新时间调用一次这个API，来获取当前获得批准的证书/主体列表。预期的 JSON 响应如下所示：

```
{
  "certificates": []
}
```

certificates

(*required, array*)

为批准的证书/主体列表。

每个证书对象定义为：

```
{
  "fingerprint_sha256": "...",
}
```

fingerprint_sha256

(*required, string*)

为批准的客户端证书的 SHA256 hash 值。Envoy 会将此 hash 与所提供的客户端证书进行匹配，以确定是否存在摘要匹配。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 19:41:36

回写

回写是一个简单的网络过滤器，主要用于演示网络级别过滤器 API。安装此过滤器后，它会将所有接收到的数据回写（写入）回连接的下游客户端。

- [v1 API reference](#)
- [v2 API reference](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 16:37:04

Mongo 代理

- MongoDB 架构概述
- [v1 API 参考](#)
- [v2 API 参考](#)

故障注入

Mongo 代理过滤器支持故障注入。可以查看 V1 以及 V2 的 API 参考文档以了解如何进行相关配置。

统计

每一个配置的 MongoDB 代理过滤器的统计信息都以 *mongo..* 开头，其统计信息如下：

名称	类型	描述
decoding_error	Counter	MongoDB 协议解码错误的数量
delay_injected	Counter	延迟被注入的次数
op_get_more	Counter	OP_GET_MORE 消息的数量
op_insert	Counter	OP_INSERT 消息的数量
op_kill_cursors	Counter	OP_KILL_CURSORS 消息的数量
op_query	Counter	OP_QUERY 消息的数量
op_query_tailable_cursor	Counter	设置了 tailable cursor flag 的 OP_QUERY 消息的数量
op_query_no_cursor_timeout	Counter	没有设置 cursor timeout flag 的 OP_QUERY 消息的数量
op_query_await_data	Counter	设置了 await data flag 的 OP_QUERY 消息的数量
op_query_exhaust	Counter	设置了 exhaust flag 的 OP_QUERY 消息的数量
op_query_no_max_time	Counter	没有设置 maxTimeMS 的查询数量
op_query_scatter_get	Counter	分散获取查询的数量
op_query_multi_get	Counter	多重查询的次数
op_query_active	Gauge	活跃查询的数量
op_reply	Counter	OP_REPLY 消息的数量
op_reply_cursor_not_found	Counter	设置了 cursor not found flag 的 OP_REPLY 消息的数量
op_reply_query_failure	Counter	设置了 query failure flag 的 OP_REPLY 消息的数量
op_reply_valid_cursor	Counter	拥有有效的游标的 OP_REPLY 消息的数量
cx_destroy_local_with_active_rq	Counter	拥有一个活跃查询并被本地破坏的连接总数
cx_destroy_remote_with_active_rq	Counter	拥有一个活跃查询并被远程破坏的连接总数
cx_drain_close	Counter	在服务器关闭期间，在回复边界被优雅关闭的连接总数

分散获取查询

任何不使用 `_id` 作为查询参数的查询，Envoy 将其定义为一个分散获取查询。Envoy 同时在最顶级文档以及 `_id` 的 `$query` 中查找。

多重查询

任何使用 `_id` 作为查询参数的查询，且 `_id` 不是一个标量值（如文档或数组），Envoy 定义其为 多重查询。Envoy 同时在最顶级文档以及 `_id` 的 `$query` 中查找。

\$comment 解析

如果一个查询具有顶级的 `$comment` 字段（通常在 `$query` 字段的基础上添加），Envoy 会将其解析为 JSON 格式并查找以下结构：

```
{
  "callingFunction": "..."
}
```

- `callingFunction`

(*required, string*) 执行查询的函数。在可用的情况下，这个函数将会用来做 [按调用站](#) 查询统计。

按命令统计

MongoDB 过滤器将在 `mongo..cmd..` 命名空间为命令收集相应的统计信息。

名称	类型	描述
<code>total</code>	Counter	命令数量
<code>reply_num_docs</code>	Histogram	回复中的文档数量
<code>reply_size</code>	Histogram	回复的字节大小
<code>reply_time_ms</code>	Histogram	命令时间（毫秒）

按集合查询统计

MongoDB 过滤器将在 `mongo..collection..query.` 命名空间为查询收集相应的统计信息。

名称	类型	描述
<code>total</code>	Counter	查询数量
<code>scatter_get</code>	Counter	分散获取查询数量
<code>multi_get</code>	Counter	多重查询梳理
<code>reply_num_docs</code>	Histogram	回复中的文档数量
<code>reply_size</code>	Histogram	回复的字节大小
<code>reply_time_ms</code>	Histogram	查询时间（毫秒）

按集合与现场查询统计

如果应用程序在 `$comment` 字段中提供[调用函数](#)，Envoy 将相应生成按调用站点为维度的统计信息。这些统计信息与 [按集合统计](#) 相匹配，可在 `mongo..collection..callsite..query.` 命名空间中找到相关信息。

运行时

Mongo 代理过滤器支持如下运行时设置：

- `mongo.connection_logging_enabled`

启用日志记录的连接百分比。默认为100。这将只允许将指定百分比的连接做日志记录，但这些连接上的所有信息将会做日志记录。

- `mongo.proxy_enabled`

启用代理的连接百分比。默认为100。

- `mongo.logging_enabled`

启用日志记录的消息的百分比。默认值为100。如果小于100，部分查询可能会在无回复的情况下被记录。

- `mongo.mongo.drain_close_enabled`

当服务器正在被删除或尝试做强制关闭时，将被关闭的连接百分比。默认为100。

- `mongo.fault.fixed_delay.percent`

当没有活跃故障时，一个合格的 MongoDB 操作受到注入故障影响的可能性。默认为配置中指定的 *percent*。

- `mongo.fault.fixed_delay.duration_ms`

以毫秒为单位的延迟时间。默认为配置中指定的 *duration_ms*。

访问日志格式

访问日志格式不可定制，并具有以下布局：

```
{"time": "...", "message": "...", "upstream_host": "...”}
```

- `time`

解析完整信息所需的系统时间，精确度至毫秒。

- `message`

文本扩展的消息。消息是否完全可扩展取决于上下文。为避免日志超大，有时会提供摘要数据。

- `upstream_host`

连接正在被代理的上游主机，在过滤器配合 [TCP 代理过滤器](#) 时，此字段将被填充。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 19:29:44

速率限制

- 全局速率限制[架构概述](#)
- [v1 API 参考](#)
- [v2 API 参考](#)

统计

所有配置的的速率限制过滤器都有以 `ratelimit.` 开头的统计，以提供以下的统计报告：

名称	类型	描述
total	Counter	所有发给速率限制服务的请求数
error	Counter	所有联系速率限制服务的错误数
over_limit	Counter	所有由速率限制服务回复的超过速率限制的响应数
ok	Counter	所有由速率限制服务回复的未超过速率限制的响应数
cx_closed	Counter	所有因超过速率限制相应而被关闭的连接数
active	Gauge	所有发给速率限制服务的活跃请求数

运行时

网络级别速率限制过滤器支持以下运行时配置：

- `ratelimit.tcp_filter_enabled`
将调用速率限制服务的连接百分比。缺省值为100。
- `ratelimit.tcp_filter_enforcing`
将调用速率限制服务并强制执行决定的连接百分比。缺省值为100。这可以在完全执行结果之前，测试将会发生什么。

Redis 代理

- Redis 架构概述
- [v1 接口文档](#)
- [v2 接口文档](#)

统计

每个已配置的 Redis 代理过滤器都有以 `redis..` 开头的统计，并提供如下的统计报告：

Name	Type	Description
downstream_cx_active	Gauge	活跃的连接总数
downstream_cx_protocol_error	Counter	协议错误总次数
downstream_cx_rx_bytes_buffered	Gauge	当前接收并缓存的总字节数
downstream_cx_rx_bytes_total	Counter	接收的总字节数
downstream_cx_total	Counter	连接总数
downstream_cx_tx_bytes_buffered	Gauge	当前发送并缓存的总字节数
downstream_cx_tx_bytes_total	Counter	发送的总字节数
downstream_cx_drain_close	Counter	因连接耗尽而关闭的连接总数
downstream_rq_active	Gauge	活跃的请求总数
downstream_rq_total	Counter	请求总数

分离器统计

Redis 过滤器将采集命令分离器的统计信息，并存放到以 `redis..splitter` 开头的统计中，提供如下的统计报告：

Name	Type	Description
invalid_request	Counter	参数个数不正确的请求数
unsupported_command	Counter	命令分离器无法识别的命令数

每个命令的统计

Redis 过滤器将收集每个命令的统计信息，并存放到以 `redis..command.` 开头的命名空间中，提供如下的统计报告：

Name	Type	Description
total	Counter	命令数

运行时

Redis 代理过滤器支持如下的运行时设置：

- redis.drain_close_enabled

当服务器因为连接耗尽而尝试关闭连接的百分比。默认值是 100。

TCP 代理

- TCP 代理 架构概述
- [v1 接口文档](#)
- [v2 接口文档](#)

统计

在适当的情况下，TCP 代理会发出自己下游以及[上游集群的统计信息](#)。下游的统计信息都有以 *tcp.* 开头的统计，并提供如下的统计报告：

名称	类型	描述
downstream_cx_total	Counter	过滤器处理的连接总数
downstream_cx_no_route	Counter	没有找到匹配路由或没有找到路由集群的连接总数
downstream_cx_tx_bytes_total	Counter	写入下游连接的总字节数
downstream_cx_tx_bytes_buffered	Gauge	当前缓存到下游连接的总字节数
downstream_cx_rx_bytes_total	Counter	从下游连接读取的总字节数
downstream_cx_rx_bytes_buffered	Gauge	当前从下游连接缓存的总字节数
downstream_flow_control_paused_reading_total	Counter	流量控制从下游暂停读取的总次数
downstream_flow_control_resumed_reading_total	Counter	流量控制从下游恢复读取的总次数
idle_timeout	Counter	由于空闲连接超时而关闭的连接总数
upstream_flush_total	Counter	在下游连接关闭后继续刷新上游数据的连接总数
upstream_flush_active	Gauge	在下游连接关闭后，当前继续刷新上游数据的连接总数

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 14:30:54

HTTP 连接管理器

- HTTP 链接管理器[架构概览](#)
- HTTP 协议[架构概览](#)
- [v1 API 参考](#)
- [v2 API 参考](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-18 12:32:20

路由匹配

注意

本节为 v1 API 编写，但概念也适用于 v2 API。未来的版本中将以 v2 API 为目标重写。

当 Envoy 匹配路由时，它使用如下步骤：

1. HTTP 请求的 *host* 或者 *:authority* 头匹配到[虚拟主机](#)。
2. 按顺序检查虚拟主机中的每个[路由条目](#)。如果匹配，则使用该路由而不再进一步检查其他路由。
3. 独立地，按顺序检查虚拟主机中的每个[虚拟集群](#)。如果匹配，则使用该虚拟集群而不再进一步检查其他虚拟集群。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 10:15:09

流量转移拆分

注意

本节为 v1 API 编写，但概念也适用于 v2 API。未来的版本中将以 v2 API 为目标重写。

- 在两个上游之间转移流量
- 跨多个上游拆分流量

Envoy 的路由器可以跨两个或更多上游集群将流量拆分到虚拟主机中的路由。有两个常见的用例。

1. 版本升级：路由的流量逐渐从一个集群优雅地转移到另一个集群。[流量转移](#)部分更详细地描述了这个场景。
2. A/B 测试或多变量测试：同时测试两个或更多版本的相同服务。流向路由的流量必须在运行同一服务的不同版本的集群之间进行拆分。[流量拆分](#)部分更详细地描述了这种情况。

在两个上游之间转移流量

路由配置中的[运行时](#)对象判断选择特定路由（以及它的集群）的可能性（译者注：可以理解为百分比）。通过使用运行时配置，虚拟主机中到特定路由的流量可逐渐从一个集群转移到另一个集群。考虑以下示例配置，其中在 envoy 配置文件中声明了名为 `helloworld` 的服务的两个版本 `helloworld_v1` 和 `helloworld_v2`。

```
{
  "route_config": {
    "virtual_hosts": [
      {
        "name": "helloworld",
        "domains": ["*"],
        "routes": [
          {
            "prefix": "/",
            "cluster": "helloworld_v1",
            "runtime": {
              "key": "routing.traffic_shift.helloworld",
              "default": 50
            }
          },
          {
            "prefix": "/",
            "cluster": "helloworld_v2",
          }
        ]
      }
    ]
  }
}
```

Envoy 使用 `first match` 策略来匹配路由。如果路由具有运行时对象，则会根据运行时值另外匹配请求（如果未指定值，则为默认值）。因此，通过在上述示例中背靠背地放置路由并在第一个路由中指定运行时对象，可以通过更改运行时值来完成流量转移。以下是完成任务所需的大致操作顺序。

1. 在开始时，将 `routing.traffic_shift.helloworld` 设置为 `100`，因此所有到 `helloworld` 虚拟主机的请求都将匹配 `v1` 路由并由 `helloworld_v1` 集群提供服务。
2. 为了开始将流量转移到 `helloworld_v2` 集群，设置 `routing.traffic_shift.helloworld` 为 `0 < x < 100`。例如设置为 `90` 时，有 1 个不会与 `v1` 路由匹配，然后会落入 `v2` 路由。
3. 逐渐减少 `routing.traffic_shift.helloworld` 中设置的值，以便大部分请求与 `v2` 路由匹配。
4. 当 `routing.traffic_shift.helloworld` 设置为 `0` 时，到 `helloworld` 虚拟主机的请求都不会匹配 `v1` 路由。现在，所有

流量都会流向 v2 路由，并由 helloworld_v2 集群提供服务。

跨多个上游拆分流量

再次考虑 `helloworld` 示例，现在有三个版本（v1、v2和v3）而不是两个。要在三个版本间平均分配流量（即 33% 、 33% 、 34% ），可以使用 `weighted_clusters` 选项指定每个上游集群的权重。

与前面的例子不同，单个路由条目就足够了。路由中的 `weighted_clusters` 配置块可用于指定多个上游集群以及权重，权重则表示要发送到每个上游集群的流量百分比。

```
{
  "route_config": {
    "virtual_hosts": [
      {
        "name": "helloworld",
        "domains": ["*"],
        "routes": [
          {
            "prefix": "/",
            "weighted_clusters": {
              "runtime_key_prefix" : "routing.traffic_split.helloworld",
              "clusters" : [
                { "name" : "helloworld_v1", "weight" : 33 },
                { "name" : "helloworld_v2", "weight" : 33 },
                { "name" : "helloworld_v3", "weight" : 34 }
              ]
            }
          }
        ]
      }
    ]
  }
}
```

默认情况下，权重的总和必须精确地为100。在 V2 API 中，总权重默认为100，但可以修改以允许更精细的粒度。

可以使用以下运行时变量动态调整分配给每个集群的权

重：`routing.traffic_split.helloworld.helloworld_v1`、`routing.traffic_split.helloworld.helloworld_v2` 和
`routing.traffic_split.helloworld.helloworld_v3`。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 10:15:09

HTTP 标头操作

HTTP 连接管理器在解码时（当接受到请求时）以及编码时（当发送响应时）操作多种 HTTP 标头。

- [user-agent](#)
- [server](#)
- [x-client-trace-id](#)
- [x-envoy-downstream-service-cluster](#)
- [x-envoy-downstream-service-node](#)
- [x-envoy-external-address](#)
- [x-envoy-force-trace](#)
- [x-envoy-internal](#)
- [x-forwarded-client-cert](#)
- [x-forwarded-for](#)
- [x-forwarded-proto](#)
- [x-request-id](#)
- [x-ot-span-context](#)
- [x-b3-traceid](#)
- [x-b3-spanid](#)
- [x-b3-parentspanid](#)
- [x-b3-sampled](#)
- [x-b3-flags](#)
- Custom request/response headers

user-agent

当启用 [add_user_agent](#) 选项后，连接管理器可能会在解码时设定 user-agent 标头。

标头只有在尚未设置的情况下才会被修改。如果连接管理器确实设置了标头，则该值由 `--service-cluster` 命令行选项确定。

server

server 标头将在编码期间被设置为 [server_name](#) 选项中的值。

x-client-trace-id

如果外部客户端设置了该标头，Envoy 会将提供的 trace ID 与内部生产的 [x-request-id](#) 连接起来。[x-client-trace-id](#) 需要保持全局的唯一性，并且我们推荐以 `uuid4` 的方式生成 id。如果设置了此标头，它与 [x-envoy-force-trace](#) 有类似的效果。请参看 [tracing.client_enabled](#) 运行时设置。

x-envoy-downstream-service-cluster

内部服务通常想知道哪个服务正在调用它们。外部请求的这个标头被清洗，而内部请求将包含调用者的服务器集群信息。

请注意在当前的实现中，这被视为调用者设置的提示，同时容易被任意内部的实体进行欺诈。将来，Envoy 将支持相互认证的 TLS 网格，从而让这个标头完全具有安全性。类似 [user-agent](#)，该值由 `--service-cluster` 命令行决定。

为启用此功能，你需要设置 [user_agent](#) 选项为 `true`。

x-envoy-downstream-service-node

内部服务可能会想知道请求来自哪个下游节点。这个标头非常类似 [x-envoy-downstream-service-cluster](#)，除了它的值是来自 `--service-node` 选项。

x-envoy-external-address

服务希望根据原始客户端的IP地址做分析，这是一种常见的需求。

然后根据对 [XFF](#) 的冗长讨论，这事情可以变得非常复杂，因此，在请求来自外部客户端时，Envoy 通过将 `x-envoy-external-address` 设置为可信客户端地址来简化此操作。内部请求未设置或覆盖 `x-envoy-external-address`。

为了达到分析目的，可以在内部服务之间安全地转发此头文件，而无需处理复杂的 XFF。

x-envoy-force-trace

如果内部请求设置了这个标头，Envoy会修改生成的 `x-request-id`，这样它就会强制收集跟踪信息。这也迫使 `x-request-id` 在响应标头中返回。如果此请求标识随后传播到其他主机，那么这些主机上也会收集跟踪，由此这些主机将为整个请求流提供一致的跟踪。

请参看 [tracing.global_enabled](#) 与 [tracing.random_sampling](#) 运行时设置。

x-envoy-internal

服务想知道请求是否来自内部来源是一种常见的情况。Envoy 使用 [XFF](#) 来确定这一点，然后将标头值设置为 `true`。

这有利于避免解析和理解 XFF。

x-forwarded-client-cert

`x-forwarded-client-cert` (XFCC)是一个代理标头，代表从客户端到服务器的路径中的部分或全部客户端以及代理服务器的证书信息。

代理可以选择在代理请求之前清理/追加/转发 XFCC 标头。

XFCC 标头值是逗号（“，”）分隔的字符串。每个子字符串都是 XFCC 元素，它保存由单个代理添加的信息。代理可以将当前客户端证书信息作为 XFCC 元素附加到请求的 XFCC 头后面的逗号后面。

每个XFCC元素都是分号“;”分隔的字符串。每个子字符串都是一个键值对，由一个等号（“=”）组成。密钥不区分大小写，值区分大小写。如果“，”，“;”或“=”出现在一个值中，则该值应该用双引号。值中的双引号应该用反斜杠双引号（\"）替换。

支持以下键值：

1. `By` 当前代理证书的主题备用名称（URI 类型）。
2. `Hash` The SHA 256 digest of the current client certificate.
3. `Cert` 当前客户端证书的 SHA 256 摘要。
4. `Subject` 当前客户端证书的主题字段。该值总是用双引号括起来。
5. `URI` 当前客户端证书的主题备用名称（URI 类型）。
6. `DNS` 当前客户端证书的主题备用名称字段（URI 类型）。客户端证书可能包含多个DNS类型的主题备用名称，每个名称都将是一个单独的键值对。

客户端证书可能包含多个主题备用名称类型。关于不同主题备用名称类型的详细信息，请参阅 [RFC 2459](#)。

以下为 XFCC 标头的一些例子：

1. 对于只有URI类型的客户端证书使用主题备用名称： `x-forwarded-client-cert:`

```
By=http://frontend.lyft.com;Hash=468ed33be74eee6556d90c0149c1309e9ba61d6425303443c0748a02dd8de688;Subject="/C=US/ST=CA/L=San Francisco/OU=Lyft/CN=Test Client";URI=http://testclient.lyft.com
```

2. 对于只有URI类型的两个客户端证书使用替代名称: `x-forwarded-client-cert`:

```
By=http://frontend.lyft.com;Hash=468ed33be74eee6556d90c0149c1309e9ba61d6425303443c0748a02dd8de688;URI=http://testclient.lyft.com,By=http://backend.lyft.com;Hash=9ba61d6425303443c0748a02dd8de688468ed33be74eee6556d90c0149c1309e;URI=http://frontend.lyft.com
```

3. 对于同时具有URI类型和DNS类型的一个客户端证书使用替代名称: `x-forwarded-client-cert`:

```
By=http://frontend.lyft.com;Hash=468ed33be74eee6556d90c0149c1309e9ba61d6425303443c0748a02dd8de688;Subject="/C=US/ST=CA/L=San Francisco/OU=Lyft/CN=Test Client";URI=http://testclient.lyft.com;DNS=lyft.com;DNS=www.lyft.com
```

Envoy 处理 XFCC 的方式由 `forward_client_cert` 和 `set_current_client_cert_details` HTTP 连接管理器选项指定。如果未设置 `forward_client_cert`, 则默认情况下会对 XFCC 标头进行清理。

x-forwarded-for

`x-forwarded-for` (XFF) 是一个标准的代理标头, 它表示请求在从客户端到服务器的路上经过的 IP 地址。在代理请求之前, 兼容代理会将最近客户端的 IP 地址附加到 XFF 列表中。XFF 的一些例子是:

1. `x-forwarded-for: 50.0.0.1` (单客户端)
2. `x-forwarded-for: 50.0.0.1, 40.0.0.1` (外部代理跳)
3. `x-forwarded-for: 50.0.0.1, 10.0.0.1` (内部代理跳)

仅在 `use_remote_address` HTTP 连接管理选项设置为 `true` 时, Envoy 才会追加到 XFF。这意味着如果 `use_remote_address` 为 `false` (这是默认值), 则连接管理器将以不修改 XFF 的透明模式运行。

注意

通常, 当 Envoy 作为边缘节点 (又名前端代理) 进行部署时, 应将 `use_remote_address` 设置为 `true`, 而将 Envoy 用作网格部署中的内部服务节点时, 可能需要将其设置为 `false`。

`use_remote_address` 的值控制 Envoy 如何确定可信客户端地址。如果 HTTP 请求已经通过一系列代理(零个或多个)传到 Envoy, 则可信的客户端地址是已知准确的最早的源IP地址。直接下游节点与 Envoy 连接的源 IP 地址是可信的。XFF 有时可以被信任。恶意客户可以伪造 XFF, 但如果 XFF 中的最后一个地址由可信代理放在那里, 则可以信任它。

Envoy 用于确定可信客户端地址的默认规则 (在向 XFF 添加任何内容之前) 是:

- 如果 `use_remote_address` 为 `false` 且包含至少一个 IP 地址的 XFF 出现在请求中, 则可信客户端地址是 XFF 中的最后 (最右边) IP 地址。
- 否则, 可信客户端地址是直接下游节点与 Envoy 连接的源 IP 地址。

在边缘 Envoy 实例前有一个或多个可信代理的环境中, `xff_num_trusted_hops` 配置选项可以用于信任来自 XFF 的其他地址:

- 如果 `use_remote_address` 为 `false` 且 `xff_num_trusted_hops` 设置为大于零的值 `N`, 则可信客户端地址为距 XFF 右端第 `(N+1)` 个地址。 (如果 XFF 包含的地址少于 `N+1` 个, Envoy 就会使用直接下行连接的源地址作为可信客户端地址。)
- 如果 `use_remote_address` 为 `true` 并且 `xff_num_trusted_hops` 设置为大于零的值 `N`, 则可信客户端地址是 XFF 右端的第 `N` 个地址。 (如果 XFF 包含的地址少于 `N` 个, Envoy 就会使用直接下行连接的源地址作为可信客户端地址。)

Envoy 使用可信的客户端地址内容来确定请求是发起于外部还是内部。这会影响是否设置了 `x-envoy-internal` 标头。

示例 1: Envoy 作为边缘代理, 在它前面没有可信代理

```
Settings:
  use_remote_address = true
  xff_num_trusted_hops = 0
```

```

Request details:
  Downstream IP address = 192.0.2.5
  XFF = "203.0.113.128, 203.0.113.10, 203.0.113.1"
Result:
  Trusted client address = 192.0.2.5 (XFF is ignored)
  X-Envoy-External-Address is set to 192.0.2.5
  XFF is changed to "203.0.113.128, 203.0.113.10, 203.0.113.1, 192.0.2.5"
  X-Envoy-Internal is removed (if it was present in the incoming request)

```

示例 2: Envoy 作为内部代理, 在它前面有一个如示例1一般的边缘代理

```

Settings:
  use_remote_address = false
  xff_num_trusted_hops = 0
Request details:
  Downstream IP address = 10.11.12.13 (address of the Envoy edge proxy)
  XFF = "203.0.113.128, 203.0.113.10, 203.0.113.1, 192.0.2.5"
Result:
  Trusted client address = 192.0.2.5 (last address in XFF is trusted)
  X-Envoy-External-Address is not modified
  X-Envoy-Internal is removed (if it was present in the incoming request)

```

示例 3: Envoy 作为边缘代理, 在它前面有两个信任的外部代理

```

Settings:
  use_remote_address = true
  xff_num_trusted_hops = 2
Request details:
  Downstream IP address = 192.0.2.5
  XFF = "203.0.113.128, 203.0.113.10, 203.0.113.1"
Result:
  Trusted client address = 203.0.113.10 (2nd to last address in XFF is trusted)
  X-Envoy-External-Address is set to 203.0.113.10
  XFF is changed to "203.0.113.128, 203.0.113.10, 203.0.113.1, 192.0.2.5"
  X-Envoy-Internal is removed (if it was present in the incoming request)

```

示例 4: Envoy 作为内部代理, 它前面有一个如示例3一般的边缘代理

```

Settings:
  use_remote_address = false
  xff_num_trusted_hops = 2
Request details:
  Downstream IP address = 10.11.12.13 (address of the Envoy edge proxy)
  XFF = "203.0.113.128, 203.0.113.10, 203.0.113.1, 192.0.2.5"
Result:
  Trusted client address = 203.0.113.10
  X-Envoy-External-Address is not modified
  X-Envoy-Internal is removed (if it was present in the incoming request)

```

示例 5: Envoy 作为内部代理, 接收来自一个内部客户的请求

```

Settings:
  use_remote_address = false
  xff_num_trusted_hops = 0
Request details:
  Downstream IP address = 10.20.30.40 (address of the internal client)
  XFF is not present
Result:
  Trusted client address = 10.20.30.40
  X-Envoy-External-Address remains unset
  X-Envoy-Internal is set to "true"

```

示例 6: 来自示例5的内部 Envoy, 接收由另外一个 Envoy 代理的请求

```
Settings:
  use_remote_address = false
  xff_num_trusted_hops = 0
Request details:
  Downstream IP address = 10.20.30.50 (address of the Envoy instance proxying to this one)
  XFF = "10.20.30.40"
Result:
  Trusted client address = 10.20.30.40
  X-Envoy-External-Address remains unset
  X-Envoy-Internal is set to "true"
```

关于 XFF 的一些非常重要的点:

1. 如果 `use_remote_address` 设置为 `true`, Envoy会将 `x-envoy-external-address` 标头设置为受信任的客户端地址。
2. XFF 是Envoy 用来确定请求是内部源还是外部源的。如果 `use_remote_address` 设置为 `true`, 当且仅当请求不包含 XFF 并且直接下游节点与 Envoy 的连接具有内部 (RFC1918或RFC4193) 源地址时, 该请求为内部请求。如果 `use_remote_address` 为 `false`, 则当且仅当 XFF 包含单个 RFC1918 或 RFC4193 地址时, 该请求才是内部请求。
3. 注意: 如果内部服务代理到另一个内部服务的外部请求, 并且包含原始 XFF 头, 则在设置了 `use_remote_address` 的情况下, Envoy 将在出口附加它。这会导致对方认为请求是外部的。一般来说, 这是 XFF 被转发的意图。如果没有这个意图, 请不要转发 XFF, 而是转发 `x-envoy-internal`。
4. 注意: 如果内部服务调用转发到其他内部服务 (保留XFF) , Envoy 将不会认为这是一个内部服务。这是一个已知的 "bug", 缘自 XFF 将解析以及判定一个请求是否来自内部的工作进行了简化。在此场景下, 请不要将 XFF 转发并允许 Envoy 使用一个内部原始 IP 生成一个新的。
5. 由变更管理的角度来看, 在大型多跳系统中测试 IPv6 可能非常困难。为了测试解析 XFF 标头的上游服务的 IPv6 兼容性, 可以在 v2 API 中启用 `represent_ipv4_remote_address_as_ipv4_mapped_ipv6`。Envoy 将以映射的 IPv6 格式附加 IPv4 地址, 例如: ::FFFF:50.0.0.1。此更改同样适用于`x-envoy-external-address`。

x-forwarded-proto

服务想要知道由 前端/边缘 Envoy 终止的连接的始发协议 (HTTP 或 HTTPS) , 这是一种常见的情况。 `x-forwarded-proto` 包含这些信息。 它将被设置为 `http` 或 `https` 。

x-request-id

Envoy 使用 `x-request-id` 头来唯一标识请求并执行稳定的访问日志记录和跟踪。Envoy将为所有外部来源请求生成一个 `x-request-id` 头 (标头被清理) 。

它还会为没有 `x-request-id` 头的内部请求生成一个 `x-request-id` 头。

这意味着 `x-request-id` 能且应该在客户端应用程序间传播, 以便在整个网格中拥有一个稳定的 ID。

由于 Envoy 的与流程无关的架构设计, Envoy 本身不能自动地转发标头。这是少数瘦客户端库需要做的工作之一。如何去做, 这个话题超出了本文档的范围。

如 `x-request-id` 跨所有主机传播, 则可使用如下功能:

- 稳定的 [访问记录](#) 通过 [v1 API 运行时过滤器](#) 或 [v2 API 运行时过滤器](#)。
- 进行随机抽样时稳定的追踪, 通过开启 `tracing.random_sampling` 运行时配置或是使用 `x-envoy-force-trace` 标头以及 `x-client-trace-id` 标头进行强行追踪。

x-ot-span-context

x-ot-span-context HTTP 标头用于在 Envoy 与 LightStep 追踪器间跟踪跨度时建立适当的父-子关系。

例如，出口跨度是入口跨度的子节点（如果入口跨度存在）。Envoy 在入口请求注入 x-ot-span-context 标头并将其转发给本地服务。

Envoy 依靠应用程序将出口调用的 x-ot-span-context 传播到上游。在[这里](#)可查看更多信息。

x-b3-traceid

Envoy 的 Zipkin 追踪器使用 x-b3-traceid HTTP 标头。Traceld 的长度为64字节，并反映追踪的总体 ID。追踪中的每个跨度都共享此 ID。可在<https://github.com/openzipkin/b3-propagation> 查阅 zipkin 追踪的更多信息。

x-b3-spanid

Envoy 的 Zipkin 追踪器使用 x-b3-spanid HTTP 标头。

SpanId 的长度为64字节，并反映当前操作在追踪树中的位置。该值不应被解释：它可能会或也有可能不会由 Traceld 的值派生出来。

可在<https://github.com/openzipkin/b3-propagation> 查阅 zipkin 追踪的更多信息。

x-b3-parentspanid

Envoy 的 Zipkin 追踪器使用 x-b3-parentspanid 标头。

ParentSpanId 的长度为64字节，并反映父操作在追踪树中的位置。当 span 为追踪树的根节点时，不存在 ParentSpanId。可在<https://github.com/openzipkin/b3-propagation> 查阅 zipkin 追踪的更多信息。

x-b3-sampled

当 Sampled flag 标志未被指定或设置为1时，跨度将被报告给追踪系统。一旦 Sampled 设置为0或1，将始终向下游发送同样的值。

可在<https://github.com/openzipkin/b3-propagation> 查阅 zipkin 追踪的更多信息。

x-b3-flags

Envoy 的 Zipkin 追踪器使用 x-b3-flags 标头。编码一个或多个选项。例如，Debug 被编码为 `X-B3-Flags: 1`。

可在<https://github.com/openzipkin/b3-propagation> 查阅 zipkin 追踪的更多信息。

custom-request-response-headers

自定义请求/响应标头可以在加权集群、路由、虚拟主机和/或全局路由配置级别添加到请求/响应中。具体可参看相关的 V1 以及 V2 API 文档。

标头将按照以下顺序附加到请求/响应中：加权集群级别标头、路由级别标头、虚拟主机级别标头以及全局级别标头。

Envoy 支持将变量添加到请求以及响应标头。百分号(%)用于分割变量名称。

注意

如果需要在请求/响应标头内增加一个书面的百分比符号 (%)，则需要重复它以达到转义的效果。

例如，要发送值为100%的标头，Envoy 配置中的自定义标头必须为 100%%。

支持的变量名有：

%DOWNSTREAM_REMOTE_ADDRESS_WITHOUT_PORT%

下游连接的远程地址。如果地址是 IP 地址，则输出不包含端口。

注意

如果从 [proxy proto](#) 或 [x-forwarded-for](#) 推断出地址，这非常可能不是对方的物理远程地址。

%DOWNSTREAM_LOCAL_ADDRESS%

下游连接的本地地址，如果地址是 IP 地址，则它包括地址和端口。如果原始连接被 iptables REDIRECT 重定向，则表示[原始目标过滤器](#) 使用 SO_ORIGINAL_DST Socket 选项恢复的原始目标地址。如果原始连接被 iptables TPROXY 重定向，且侦听器的透明选项设置为 true，则表示原始目标地址和端口。

%DOWNSTREAM_LOCAL_ADDRESS_WITHOUT_PORT%

与 %DOWNSTREAM_LOCAL_ADDRESS% 类同，但如果地址是 IP 地址时排除端口。

%PROTOCOL%

Envoy 已将其添加为原始协议作为 [x-forwarded-proto](#) 请求标头。

%UPSTREAM_METADATA([“namespace”, “key”, …])%

用来自路由器选择的上游主机的 [EDS端点元数据](#) 填充报头。元数据可以从任何名称空间中选择。通常，元数据值可以是字符串，数字，布尔值，列表，嵌套结构或空值。可以通过指定多个键从嵌套结构中选择上游元数据值。

否则，只支持字符串，布尔值和数值。如果未找到命名空间或键值，或者所选值不是受支持的类型，则不会发出标头。命名空间和键被指定为 JSON 字符串数组。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-23 21:33:38

HTTP 标头清理

出于安全原因考虑，Envoy 将根据请求是内部请求还是外部请求来“清理”各种传入的 HTTP 标头。清理行为取决于标头，并可能会导致添加、删除或更改。最终，一个请求被认定为内部或是外部请求都是由 `x-forwarded-for` 标头决定（请仔细阅读链接部分，因为 Envoy 填充标头的动作是非常复杂的，并取决于 `use_remote_address` 设置）。

Envoy 可能会清理以下标头：

- `x-envoy-decorator-operation`
- `x-envoy-downstream-service-cluster`
- `x-envoy-downstream-service-node`
- `x-envoy-expected-rq-timeout-ms`
- `x-envoy-external-address`
- `x-envoy-force-trace`
- `x-envoy-internal`
- `x-envoy-ip-tags`
- `x-envoy-max-retries`
- `x-envoy-retry-grpc-on`
- `x-envoy-retry-on`
- `x-envoy-upstream-alt-stat-name`
- `x-envoy-upstream-rq-per-try-timeout-ms`
- `x-envoy-upstream-rq-timeout-alt-response`
- `x-envoy-upstream-rq-timeout-ms`
- `x-forwarded-client-cert`
- `x-forwarded-for`
- `x-forwarded-proto`
- `x-request-id`

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 15:55:23

统计

每个连接管理器都有一个以 `http..` 为根的统计树，其统计信息如下：

名称	类型	描述
<code>downstream_cx_total</code>	Counter	连接总数
<code>downstream_cx_ssl_total</code>	Counter	TLS 连接总数
<code>downstream_cx_http1_total</code>	Counter	HTTP/1.1 连接总数
<code>downstream_cx_websocket_total</code>	Counter	WebSocket 连接总数
<code>downstream_cx_http2_total</code>	Counter	HTTP/2 连接总数
<code>downstream_cx_destroy</code>	Counter	被破坏的连接总数
<code>downstream_cx_destroy_remote</code>	Counter	由于远程关闭，而导致被破坏的连接总数
<code>downstream_cx_destroy_local</code>	Counter	由于本地关闭，而导致被破坏的连接总数
<code>downstream_cx_destroy_active_rq</code>	Counter	由于超过一个活跃请求，而导致被破坏的连接总数
<code>downstream_cx_destroy_local_active_rq</code>	Counter	由于超过一个活跃请求，而导致被本地破坏的连接总数
<code>downstream_cx_destroy_remote_active_rq</code>	Counter	由于超过一个活跃请求，而导致被远程破坏的连接总数
<code>downstream_cx_active</code>	Gauge	活跃连接总数
<code>downstream_cx_ssl_active</code>	Gauge	活跃 TLS 连接总数
<code>downstream_cx_http1_active</code>	Gauge	活跃 HTTP/1.1 连接总数
<code>downstream_cx_websocket_active</code>	Gauge	活跃 WebSocket 连接总数
<code>downstream_cx_http2_active</code>	Gauge	活跃 HTTP/2 连接总数
<code>downstream_cx_protocol_error</code>	Counter	协议错误总数
<code>downstream_cx_length_ms</code>	Histogram	连接长度毫秒
<code>downstream_cx_rx_bytes_total</code>	Counter	收到的总字节数
<code>downstream_cx_rx_bytes_buffered</code>	Gauge	当前收到并缓存的总字节数
<code>downstream_cx_tx_bytes_total</code>	Counter	发出的总字节数
<code>downstream_cx_tx_bytes_buffered</code>	Gauge	当前发出并缓存的总字节数
<code>downstream_cx_drain_close</code>	Counter	由于删除，而导致被关闭的连接总数
<code>downstream_cx_idle_timeout</code>	Counter	由于空闲超时，而导致被关闭的连接总数
<code>downstream_flow_control_paused_reading_total</code>	Counter	由于流量控制，而导致被禁止的总读取次数
<code>downstream_flow_control_resumed_reading_total</code>	Counter	由于流量控制，而导致在连接上启用的总读取次数
<code>downstream_rq_total</code>	Counter	请求总数
<code>downstream_rq_http1_total</code>	Counter	HTTP/1.1 总请求数

downstream_rq_http2_total	Counter	HTTP/2 请求总数
downstream_rq_active	Gauge	活跃请求总数
downstream_rq_response_before_rq_complete	Counter	在请求完成之前发送的总响应数
downstream_rq_rx_reset	Counter	收到的请求重置总数
downstream_rq_tx_reset	Counter	发出的请求重置总数
downstream_rq_non_relative_path	Counter	带有非相对 HTTP 路径的请求总数
downstream_rq_too_large	Counter	由于缓存过大正文，而导致 413 响应的请求数
downstream_rq_1xx	Counter	1xx 响应总数
downstream_rq_2xx	Counter	2xx 响应总数
downstream_rq_3xx	Counter	3xx 响应总数
downstream_rq_4xx	Counter	4xx 响应总数
downstream_rq_5xx	Counter	5xx 响应总数
downstream_rq_ws_on_non_ws_route	Counter	由于非 WebSocket 路由而被拒绝的 WebSocket 升级请求总数
downstream_rq_time	Histogram	请求时间(毫秒)
rs_too_large	Counter	由于缓存过大正文，而导致的错误响应总数

以 user agent 维度进行统计

以 user agent 维度进行的统计信息都以 `http..user_agent..` 开头。目前 Envoy 匹配 iOS (`ios`) 以及 Android (`android`) 的 user agent，并产生以下统计信息：

名称	类型	描述
downstream_cx_total	Counter	连接总数
downstream_cx_destroy_remote_active_rq	Counter	由于超过一个活跃请求，而导致被远程破坏的连接总数
downstream_rq_total	Counter	请求总数

以监听器维度进行统计

以监听器维度进行的统计信息都以 `listener..`

`.http..`

开头，并有以下统计信息：

名称	类型	描述
downstream_rq_1xx	Counter	1xx 响应总数
downstream_rq_2xx	Counter	2xx 响应总数
downstream_rq_3xx	Counter	3xx 响应总数
downstream_rq_4xx	Counter	4xx 响应总数

downstream_rq_5xx	Counter	5xx 响应总数
-------------------	---------	----------

以编解码器维度进行统计

每一个编解码器都有进行按编解码器维度进行统计的能力。目前只有 http2 有编解码器统计信息。

Http2 编解码器统计

所有的 http2 统计信息都以 *http2.* 开头

名称	类型	描述
header_overflow	Counter	由于头部大于 Envoy :: Http :: Http2 :: ConnectionImpl :: StreamImpl :: MAX_HEADER_SIZE (63k) 而重置的连接总数
headers_cb_no_stream	Counter	在没有关联流的情况下进行头部回调的错误总数。由于尚未确诊的 bug, 这会导致一些意外事件的发送
rx_messaging_error	Counter	因为违背 HTTP/2 spec 中的 section 8 而导致的无效接受帧总数。这将导致 <i>tx_reset</i>
rx_reset	Counter	Envoy 收到的重置流帧的总数
too_many_header_frames	Counter	由于接收太多头帧而导致 HTTP2 连接重置的总次数。Envoy 支持代理最多一个 100-Continue 头部帧, 一个 non-100 响应代码头部帧和一个拥有尾部的帧
trailers	Counter	由下游请求所看到的尾部总数
tx_reset	Counter	Envoy 发送的重置流帧总数

追踪统计

追踪统计信息是在做出追踪决定时发出的。所有追踪统计信息都以 *http..tracing.* 开头，并带有以下统计信息：

名称	类型	描述
random_sampling	Counter	通过随机抽样可追踪决策的总数
service_forced	Counter	通过服务器运行时标识 <i>tracing.global_enabled</i> 的可追踪决策的总数
client_enabled	Counter	通过请求头部 <i>x-envoy-force-trace</i> 设定的可追踪决策的总数
not_traceable	Counter	通过 request id 的不可追踪的决策总数
health_check	Counter	通过健康检查的不可追踪的决策总数

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 14:55:32

运行时

HTTP 连接管理支持以下运行时设定：

- `http_connection_manager.represent_ipv4_remote_address_as_ipv4_mapped_ipv6`

将其 IPv4 地址映射到 IPv6 的远程地址的请求的百分比。默认为0。需要将 `use_remote_address` 开启。详情可参看 [represent_ipv4_remote_address_as_ipv4_mapped_ipv6](#)。

- `tracing.client_enabled`

如设置 `x-client-trace-id` 头部，将被强行追踪的请求的百分比。默认为100。

- `tracing.global_enabled`

在应用所有其他检查后追踪的请求的百分比（强制追踪，采样等）。默认为100。

- `tracing.random_sampling`

将被随机追踪的请求的百分比。请查阅[此处](#)以获得更多信息。该运行时间控制在0-10000范围内指定，默认值为10000。因此，可以按0.01%的增量指定追踪采样。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 17:50:55

路由发现服务 (RDS)

路由发现服务 (RDS) 的 API 在 Envoy 里面是一个可选 API，用于动态获取[路由配置](#)。路由配置包括 HTTP 头部修改，虚拟主机以及每个虚拟主机中包含的单个路由规则。每个 [HTTP 连接管理器](#) 都可以通过 API 独立地获取自身的路由配置。

- [v1 API reference](#)
- [v2 API reference](#)

统计

RDS 的统计树以 `http.<stat_prefix>.rds.<route_config_name>.*` 为根，`<route_config_name>` 名称中的任何 `:` 字符在统计树中被替换为 `_`。统计树包含以下统计信息：

名称	类型	描述
config_reload	Counter	加载配置不同导致重新调用API的总次数
update_attempt	Counter	调用API获取资源重试总数
update_success	Counter	调用API获取资源成功总数
update_failure	Counter	调用API获取资源失败总数（因网络、句法错误）
version	Gauge	最后一次API获取资源成功的内容HASH

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 14:52:09

缓冲区

缓冲区过滤器用于停止过滤器迭代并等待完全被缓冲的完整请求。这可以在不同场景下发挥作用，包括确保应用程序不必去处理不完整请求以及高网络延迟。

- [v1 API 参考](#)
- [v2 API 参考](#)

单路由配置

通过在虚拟主机、路由或加权集群上提供 `BufferPerRoute` 配置，可达到在单路由的基础上重写或禁用缓冲区过滤器。

统计

缓冲区过滤器在 `http..buffer` 命名空间输出统计信息。[统计信息前缀](#) 来自所拥有的 HTTP 连接管理器。

名称	类型	描述
<code>rq_timeout</code>	Counter	因超时等待完整请求的总请求数

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 14:30:54

CORS

这是一个基于路由或虚拟主机设置处理跨源资源共享请求的过滤器。请参阅下面的页面以了解更多信息。

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS
- <https://www.w3.org/TR/cors/>
- [v1 API 参考](#)
- [v2 API 参考](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 14:37:49

DynamoDB

- [DynamoDB 架构概述](#)
- [v1 API 参考](#)
- [v2 API 参考](#)

统计

DynamoDB 过滤器在 `http..dynamodb.` 命名空间输出统计信息。[统计前缀](#) 来自所拥有的 HTTP 连接管理器。

可以在 `http..dynamodb.operation..` 命名空间找到按操作为维度的统计信息。

名称	类型	描述
<code>upstream_rq_total</code>	Counter	以 命名的请求总数
<code>upstream_rq_time</code>	Histogram	在 上耗费的时间
<code>upstream_rq_total_xxx</code>	Counter	以 命名的请求总数, 以响应代码为维度统计(503、2xx等)
<code>upstream_rq_time_xxx</code>	Histogram	在 上耗费的时间, 以响应代码为维度统计(400、3xx等)

可以在 `http..dynamodb.table..` 命名空间找到按数据库表为维度的统计信息。大多数的 DynamoDB 操作仅仅涉及一个数据库表, 而 BatchGetItem 以及 BatchWriteItem 会包括多个数据库表。只有在该批次的所有操作中使用的数据库表都是相同时, Envoy 才会跟踪每个数据库表的统计数据。

名称	类型	描述
<code>upstream_rq_total</code>	Counter	对数据库表 的请求总数
<code>upstream_rq_time</code>	Histogram	在数据库表 上耗费的时间
<code>upstream_rq_total_xxx</code>	Counter	对数据库表 的请求总数, 以响应代码为维度统计(503、2xx等)
<code>upstream_rq_time_xxx</code>	Histogram	在数据库表 上耗费的时间, 以响应代码为维度统计(400、3xx等)

免责声明: 请注意, 这是尚未广泛使用的预发布 *Amazon DynamoDB* 功能。按每个分区及操作为维度的统计信息可以在 `http..dynamodb.table..` 命名空间中找到。对于批量操作, 只有在该批次的所有操作中使用的数据库表都是相同时, 只有在该批次的所有操作中使用的数据库表都是相同时, Envoy 才会按分区及操作追踪统计信息。

名称	类型	描述
<code>capacity..__partition_id=</code>	Counter	指定分区 上的 数据库表 上的操作 的总容量

其他详细统计信息:

- 对于4xx响应和不完整的批处理操作失败, 将在 `http..dynamodb.error..` 命名空间内对指定数据库表以及故障的失败总数进行追踪。

名称	类型	描述
	Counter	对指定数据库表 发生故障 的总次数
<code>BatchFailureUnprocessedKeys</code>	Counter	对指定数据库表 发生不完整的批处理操作失败的次数

运行时

DynamoDB 过滤器支持以下运行时设置:

- dynamodb.filter_enabled

启用过滤器的请求的百分比。默认值是100%。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 15:44:51

故障注入

故障注入过滤器可以用来测试弹性的微服务架构中不同的错误形式，过滤器也可用用户自定义的错误代码来延时注入以及终止请求，从而提供了不同的失败场景下处理的能力，例如服务失败、服务过载、服务高延时、网络分区等。故障注入可以限定基于（目的地）上游集群中的一组特定的请求或者一组预定义的请求头。

故障可观察到的范围仅限于通过网络通信的应用程序。无法模拟本地主机上的CPU和磁盘故障。

目前，故障注入过滤器有如下局限性：

- 中止代码仅限于HTTP状态代码
- 延时仅限于固定时长

在未来的版本中将会包括对特殊路由限定故障的支持，基于分布注入`gRPC` 和 `HTTP/2` 的指定错误码和延时的持续时长

配置

注释

故障注入过滤器需要在其他过滤器之前注入，包括路由过滤器。

- [v1 API reference](#)
- [v2 API reference](#)

运行时

Http 错误注入器支持一下全局运行时配置：

- `fault.http.abort.abort_percent`

如果请求头匹配，将按照百分比终止请求。默认添加 `abort_percent` 的配置。如果配置中不包含 `abort` 语句块，则 `abort_percent` 的默认值为0。

- `fault.http.abort.http_status`

如果请求头匹配，HTTP 状态码将作为匹配时中止的状态码。默认添加 HTTP 状态码的配置。如果配置中不包含 `abort` 语句块，则 `http_status` 默认值为 0。

- `fault.http.delay.fixed_delay_percent`

如果请求头匹配，则按照百分比延时请求。默认添加 `delay_percent` 的配置，否则为0。

- `fault.http.delay.fixed_duration_ms`

延时持续毫秒数。如果不指定值，则 `fixed_duration_ms` 使用默认的配置。如果在运行时和配置中都缺少该字段，则不是用延时。

备注，如果存在指定的下游集群的故障过滤器运行时设置，将重写默认的运行时设置。以下是指定下游集群运行时的 keys：

- `fault.http..abort.abort_percent`
- `fault.http..abort.http_status`
- `fault.http..delay.fixed_delay_percent`
- `fault.http..delay.fixed_duration_ms`

可以通过 [HTTP x-envoy-downstream-service-cluster](#) 的 header 头获取下游集群的名称。如果在运行时中没有找到运行时配置，则使用全局的运行时配置作为当前配置。

统计

故障过滤器使用 `http..fault.` 作为命名空间输出统计结果。这个统计前缀 [stat prefix](#) 来源于 HTTP 链接管理器。

名称	类型	描述
<code>delays_injected</code>	Counter	延迟的总请求数
<code>aborts_injected</code>	Counter	中止的总请求数
<code>.delays_injected</code>	Counter	指定的下游集群的延迟的总请求数
<code>.aborts_injected</code>	Counter	指定的下游集群的中止的总请求数

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:34:21

gRPC HTTP/1.1 桥接

- gRPC 架构简述
- v1 API 参考
- v2 API 参考

这是一个简单的过滤器，对于不支持复杂 gRPC 服务响应尾部的 HTTP/1.1 客户端，该过滤器可以实现桥接功能。它通过以下的步骤工作：

- 当发送一个请求时，过滤器会检查该链接是否是 HTTP/1.1 以及该请求的内容类型是否是 `application/grpc`。
- 如果符合上面的条件，当收到响应时，过滤器会缓存它并等待响应的尾部，然后检查 `grpc-status` 状态码。如果状态码不为零，过滤器把 HTTP 状态码转换为 503。同时复制 `grpc-status` 和 `grpc-message` 的尾部到响应的头部，以便客户端在需要的时候可以查看他们。
- 客户端应该发送可以转换为以下伪头部的 HTTP/1.1 请求：
 - `:method:` POST
 - `:path:`
 - `content-type:` application/grpc
- 请求的主体部分应该是以下格式的序列化的 grpc 主体：
 - 一个字节的零字符(没有压缩)
 - 网络顺序的4字节的原型消息长度
 - 序列化后的原型消息
- 因为这个模式必须缓存响应以查找 `grpc-status` 尾部，因此它只对一元模式的 gRPC 接口有效。

这个过滤器同时收集所有传输的 gRPC 请求，即使这些请求是正常通过 HTTP/2 传输的 gRPC 请求。

更多的信息：有线格式地址 [gRPC 通过 HTTP/2](#)。

统计

过滤器在 `cluster..grpc` 命名空间中输出统计信息。

名称	类型	描述
<code>..success</code>	Counter	完全成功的服务或方法调用
<code>..failure</code>	Counter	完全失败的服务或方法调用
<code>..total</code>	Counter	完全服务或方法调用

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 10:30:35

gRPC-JSON 转码

- gRPC 架构简述
- [v1 API 参考](#)
- [v2 API 参考](#)

这个过滤器可以实现 Restful JSON 风格客户端通过 HTTP 协议向 Envoy 发送请求，透过代理实现对一个 gRPC 服务的访问。HTTP 与 gRPC 服务之间的映射关系定义在 [自定义选项](#)。

如何创建原型描述符

为了实现代码转换，Envoy 需要知道你的 gRPC 服务的原型描述符。

要为 gRPC 服务创建一个协议描述符，你需要在运行 protoc 之前，先在 GitHub 上把 googleapis 仓库克隆到本地，因为你需要在包含路径中包含有 annotations.proto，才能定义 HTTP 的映射。

```
git clone https://github.com/googleapis/googleapis  
GOOGLEAPIS_DIR=<your-local-googleapis-folder>
```

然后运行 protoc 从 bookstore.proto 中生成描述集：

```
protoc -I$(GOOGLEAPIS_DIR) -I. --include_imports --include_source_info \  
--descriptor_set_out=proto.pb test/proto/bookstore.proto
```

如果你有多个 proto 源文件，你可以把它们全部传递到一条命令中。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-23 15:33:43

gRPC-Web

- [gRPC 架构概述](#)
- [v1 API reference](#)
- [v2 API reference](#)

这个过滤器通过以下步骤可以将 gRPC-Web 客户端桥接到兼容的 gRPC 服务

<https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-WEB.md>。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 15:49:08

Gzip

Gzip 是一个 http 过滤器。该过滤器允许 Envoy 压缩从上游服务基于客户端请求分发的数据。在大量数据需要传输而响应时间被限定的情况下，压缩操作是有用的。

Configuration

- [v2 API reference](#)

注意

`window bits` 是一个数字。这个数字告诉压缩器，算法应该在文本之前多大范围来寻找重复的字符序列。由于在 zlib 库里的一个已知 bug，`window bits` 的值如果是 8，则这个值不能按照预期运行。因此任何小于 8 的值，应该被自动置为 9。这个问题在库的未来版本将被解决

它是如何工作的

当 gzip 过滤器被置为 `enabled` 状态，请求和响应头都被检测，从而决定是否内容需要被压缩。如果请求或响应允许的话，内容被压缩并随后带着相应的 http 头发送给客户端

在默认情况下，压缩操作被忽略。这些情况如下所列：

- 请求不包含头 `accept-encoding`
- 请求包含头 `accept-encoding`，但是不包含 “gzip”
- 响应包含头 `content-encoding`
- 响应包含头 `cache-control`，该头的值包含 “no-transform”
- 响应包含头 `transfer-encoding`，该头的值包含 “gzip”
- 响应的 `content-type` 值，不和如下的 mime-type 值相匹配，这些 mime-type 值包含：`application/javascript`, `application/json`, `application/xhtml+xml`, `image/svg+xml`, `text/css`, `text/html`, `text/plain`, `text/xml`
- 响应中及不包含 `content-type` 头，也不包含 `transfer-encoding` 头
- 响应长度小于 30 个字节（仅应用于 `transfer-encoding` 没有被分成大块）

在以下情况，压缩操作被使用：

- `content-length` 被从响应头中移除
- 响应头包含 “`transfer-encoding: chunked`” 以及 “`content-encoding: gzip`”
- 头 “`vary: accept-encoding`” 被插入到了每一个响应中

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 21:44:58

健康检查

- 健康检查过滤器架构概述
- v1 API 参考
- v2 API 参考

注意

请注意如果调用了 [/healthcheck/fail](#) 管理端点，过滤器将会自动地在健康检查中失败并标识 `x-envoy-immediate-health-check-fail` 标头。 ([/healthcheck/ok](#) 管理端点可以反转此行为）。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:11:25

IP 标签

HTTP IP 标签过滤器使用来自可信地址的 `x-forwarded-for` 的值来设置标签头 `x-envoy-ip-tags`。如果没有，则不设置。

IP 标签的实施提供了一种可扩展的方式来高效地将 IP 地址与大量的 CIDR 列表进行范围比较。用于存储标签和 IP 地址子网的基础算法在 S.Nilsson 和 G.Karlsson 的 `IP-address lookup using LC-tries` 论文中阐述。

配置

- [v2 API reference](#)

统计

IP 标签过滤器会在命名空间 `http.stat_prefix.ip_tagging` 中输出统计信息。`stat_prefix` 来自对应的 HTTP 链接管理器。

名称	类型	描述
<code>tag_name.hit</code>	Counter	已应用 <code>tag_name</code> 的请求总数
<code>no_hit</code>	Counter	没有适用IP标签的请求总数
<code>total</code>	Counter	IP 标签过滤器运行的请求数

运行时

IP 标签过滤器支持以下运行时设置：

- `ip_tagging.http_filter_enabled`

启用过滤器的请求的百分比，默认值是100。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 21:38:59

Lua

注意

默认情况下，作为共享对象安装的 Lua 模块需要的符号，Envoy 在构建的时候不会导出它们。Envoy 可能需要建立对导出符号的支持。请参阅 [Bazel 文档](#)以获取更多信息。

概览

HTTP Lua 过滤器允许 [Lua](#) 脚本在请求和响应流期间运行。[LuaJIT](#) 被作为运行时使用。因此，被支持的 Lua 版本大多是带一些 5.2 特性的 5.1 版。更多细节参见 [LuaJIT 文档](#)。

该过滤器仅支持加载配置中内置的 Lua 代码。如果需要本地文件系统中的代码，可以使用内置小脚本从本地环境加载剩余的代码。

在高层次对过滤器设计以及 Lua 支持概括如下：

- 所有 Lua 环境都是 [针对工作线程的](#)。这意味着没有真正的全局数据。任何在加载时创建和存在的全局数据将只在每个工作线程中可见。将来可能会通过添加 API 来支持真正的全局支持。
- 所有脚本作为协程运行。这意味着，即使它们可能执行复杂的异步操作，它们以同步方式编写。这使得脚本相当容易被编写。所有网络/异步处理通过一组 API 被 Envoy 执行。Envoy 将适当退出脚本并在异步任务完成后恢复。
- 不要在脚本中执行阻塞操作。** Envoy API 被用于所有 IO，这一点对性能很关键。

当前被支持的高层次特性：

注意 预计这个列表将在生产环境不断地使用该过滤器的过程中被扩充。API 的表面被有意地保持很小。以达到编写脚本极度简单和安全的目的。在非常复杂或高性能的使用案例中，默认用原生的 C++ 过滤器 API。

- 在请求流、响应流或二者同时流入时，检查头，正文和尾。
- 修改头和尾部。
- 阻塞并缓存全部请求/响应正文用作检查。
- 执行对外的异步 HTTP 调用至一个上游主机。可以在缓存正文数据时执行这样的调用，因此，当调用结束时，上游头可以被修改。
- 执行一个直接响应并略过更多的过滤器循环。例如，一个脚本可以做一个上游 HTTP 调用做鉴权，并随后直接以一个 403 响应代码响应。

配置

- [v1 API 参考](#)
- [v2 API 参考](#)

脚本示例

本节提供了一些 Lua 脚本的具体例子，作为更温和的介绍和快速开始。关于支持的 API 的更多细节，请参照[流处理 API](#)。

```
-- 在请求路径上被调用。
function envoy_on_request(request_handle)
    -- 等待整个请求正文并添加一个请求头和正文的大小。
```

```

request_handle:headers():add("request_body_size", request_handle:body():length())
end

-- 在响应路径上被调用。
function envoy_on_response(response_handle)
    -- 等待整个响应正文并添加一个请求头和正文的大小。
    response_handle:headers():add("response_body_size", response_handle:body():length())
    -- 移除一个名为 'foo' 的响应头。
    response_handle:headers():remove("foo")
end

function envoy_on_request(request_handle)
    -- 使用下面的头、正文和超时对上游主机做一个 HTTP 调用。
    local headers, body = request_handle:httpCall(
        "lua_cluster",
        {
            [":method"] = "POST",
            [":path"] = "/",
            [":authority"] = "lua_cluster"
        },
        "hello world",
        5000)

    -- 添加来自 HTTP 调用的信息到将要被发送到过滤器链中下一个过滤器的头。
    request_handle:headers():add("upstream_foo", headers["foo"])
    request_handle:headers():add("upstream_body_size", #body)
end

function envoy_on_request(request_handle)
    -- 做一个 HTTP 调用。
    local headers, body = request_handle:httpCall(
        "lua_cluster",
        {
            [":method"] = "POST",
            [":path"] = "/",
            [":authority"] = "lua_cluster"
        },
        "hello world",
        5000)

    -- 直接响应并从 HTTP 调用设置一个头。没有更多的过滤器循环发生。
    request_handle:respond(
        {[":status"] = "403",
         ["upstream_foo"] = headers["foo"],
         "nope"})
end

```

流操作 API

当 Envoy 加载配置中脚本，它寻找脚本定义的两个全局函数：

```

function envoy_on_request(request_handle)
end

function envoy_on_response(response_handle)
end

```

一个脚本可以定义这两个函数中的一个或者两个。在请求路径中，Envoy 将作为一个协程运行 `envoy_on_request`，传递一个 API 句柄。在相应路径中，Envoy 将作为一个协程运行 `envoy_on_response`，传递一个 API 句柄。

注意

所有与 Envoy 通过传递的流句柄发生的交互是至关重要的。流句柄不应该被赋值给任何全局变量，而且不应该被用于协程的外部。如果该句柄使用不正确，Envoy 将失败。

下列在流句柄上的方法被支持：

headers()

```
headers = handle:headers()
```

返回流的头。只要它们还没有被发送到头链中的下一个过滤器，就可以被修改。例如，它们可以在一个 `httpCall()` 或者 `body()` 调用返回后被修改。如果头在任何其他情况下被修改，脚本将失败。

返回一个[头对象](#)。

body()

```
body = handle:body()
```

返回流的正文。这个调用将造成 Envoy 退出脚本直到整个正文被缓存。注意，所有缓存必须遵从适当的流控策略。Envoy 将不会缓存比连接管理器允许的更多的数据。

返回一个[缓存对象](#)。

bodyChunks()

```
iterator = handle:bodyChunks()
```

返回一个迭代器，它可被用于在所有正文块到达时循环访问它们。Envoy 将在块之间退出脚本，但是将不会缓存它们。这可被一个脚本用于在数据流入时做检查。

```
for chunk in request_handle:bodyChunks() do
    request_handle:log(0, chunk:length())
end
```

迭代器返回的每一块是一个[缓存对象](#)。

trailers()

```
trailers = handle:trailers()
```

返回流的尾。如果没有尾，可能返回空。尾在被发送到下一个过滤器之前可能被修改。

返回一个[头对象](#)。

log^{*}()

```
handle:logTrace(message)
handle:logDebug(message)
handle:logInfo(message)
handle:logWarn(message)
handle:logErr(message)
handle:logCritical(message)
```

使用 Envoy 的应用日志功能保存一条消息。*message* 是被保存的字符串。

httpCall()

```
headers, body = handle:httpCall(cluster, headers, body, timeout)
```

对一台上游主机做一个 HTTP 调用。Envoy 将推出脚本直到调用结束或者有一个错误。*cluster* 是一个字符串，映射为一个配置好的集群管理器。*headers* 一个要发送的键值对的表。注意，*:method*, *:path* 和 *:authority* 头必须被设置。*body* 是一个可选的要发送的正文数据的字符串。*timeout* 是一个整数，指定以微秒为单位的调用超时。

返回 *headers*，这是响应头的一个表。返回 *body*，这是响应正文的字符串。如果没有正文，则返回空。

respond()

```
handle:respond(headers, body)
```

立即响应并且不做更多的循环。这个调用仅在请求流中合法。另外，只有当请求头还没有被传递给后续过滤器时才可能。这意味着，下面的 Lua 代码是不合法的：

```
function envoy_on_request(request_handle)
    for chunk in request_handle:bodyChunks() do
        request_handle:respond(
            {[":status"] = "100",
             "nope"})
    end
end
```

headers 一张要发送的键值对的表。注意，*:status* 头必须被设置。*body* 是一个字符串并提供了可选的响应正文，可能为空。

metadata()

```
metadata = handle:metadata()
```

返回当前路由条目元数据。注意，元数据应该在过滤器名下指定，例如 *envoy.lua*。下面是一个在 [路由条目](#) 中 *metadata* 的例子。

```
metadata:
  filter_metadata:
    envoy.lua:
      foo: bar
      baz:
        - bad
        - baz
```

返回一个[元数据对象](#)。

头对象 API

add()

```
headers:add(key, value)
```

添加一个头。*key* 是一个提供头键的字符串。*value*是一个提供头值的字符串。

注意

Envoy 特殊处理特定头。这些被称为 O(1) 或 *inline* 头。一个内建头的列表可以在[这里](#)找到。如果一个内建头已经呈现在头映射中，*add()* 将没有效果。如果试图去 *add()* 一个非内建的头，附加的头将会被添加，因此，合成的头包含多个同名的头。如果想要将头换为另一个值，可以考虑使用 *replace* 函数。注意，我们理解这个行为令人迷惑并且我们可能在将来的一版中改变。

get()

```
headers:get(key)
```

得到一个头。*key* 是一个提供头键的字符串。返回一个头值的字符串，如果没有这个头则返回空。

pairs()

```
for key, value in pairs(headers) do
end
```

循环访问每个头。*key* 是一个提供头键的字符串。*value*是一个提供头值的字符串。

注意

在当前的实现中，头不能在循环期间被修改。另外，如果想要在循环后修改头，循环必须完成。这意味着，不用过早使用跳出或其他机制退出循环体。这个要求可能在将来被放松。

remove()

```
headers:remove(key)
```

移除一个头。*key* 提供了要移除的头键。

replace()

```
headers:replace(key, value)
```

替换一个头。*key* 是一个提供头键的字符串。*value*是一个提供头值的字符串。如果头不存在，它用 *add()* 函数添加。

缓存 API

length()

```
size = buffer:length()
```

获得以字节为单位的缓存的大小。返回一个整数。

getBytes()

```
buffer:getBytes(index, length)
```

从缓存中获得字节。Envoy 默认将不会拷贝所有的缓存数据给 Lua。这将造成一个缓存段被拷贝。*index* 是一个整数并提供了缓存要开始拷贝的索引。*length* 是一个整数并提供了要拷贝的缓存长度。*index + length* 必须小于缓存的长度。

元数据对象 API

get()

```
metadata:get(key)
```

获得一个元数据。*key* 是一个提供了元数据键的字符串。返回给定元数据键对应的值。值的类型可以是: *null*, *boolean*, *number*, *string* 和 *table*。

pairs()

```
for key, value in pairs(metadata) do  
end
```

循环访问每个 *metadata* 条目。*key* 是一个提供 *metadata* 键的字符串。*value* 是 *metadata* 条目值。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 10:31:36

频率限制

- 全局频率限制[架构概览](#)
- [v1 API 参考](#)
- [v2 API 参考](#)

如果一个请求的路由或者虚拟主机中设置了一个或多个符合过滤条件的[频率限制](#), HTTP 频率限制过滤器会调用频率限制服务。路由能够可选地包含虚拟主机的频率限制配置。一个请求中可以被施加一个或者多个路由限制。每个配置都会生成一个发给频率限制服务的描述符。

调用频率限制服务之后, 如果任何描述符得到了超限的返回值, 那么这个服务就会生成一个 429 的状态码。

编写 Action

注意 这一节是使用 v1 API 进行的, 但其中的概念也是适用于 v2 API 的。未来会面向 v2 重写这部分内容。

每个路由或者虚拟主机上的的[频率限制 Action](#) 都会生成一个描述符条目。描述符条目的向量组成一个描述符。Action 可以用任何顺序编写, 可以创建更加复杂的频率限制描述符。描述符会按照配置中的 Action 顺序执行。

例一

例如, 要生成下面的描述符:

```
( "generic_key", "some_value")
( "source_cluster", "from_cluster")
```

就需要编写这样的配置代码:

```
{
  "actions" : [
    {
      "type" : "generic_key",
      "descriptor_value" : "some_value"
    },
    {
      "type" : "source_cluster"
    }
  ]
}
```

例二

如果一个 Action 没有加入描述符条目, 那么这一配置就不会生成描述符。

例如下面的配置:

```
{
  "actions" : [
    {
      "type" : "generic_key",
      "descriptor_value" : "some_value"
    },
    {
      "type" : "remote_address"
    },
  ]
}
```

```
{
  "type" : "source_cluster"
}
]
}
```

如果一个请求没有设置 `x-forwarded-for`, 不会生成描述符。

如果请求中设置了 `x-forwarded-for`, 会生成如下的描述符:

```
("generic_key", "some_value")
("remote_address", "<trusted address from x-forwarded-for>")
("source_cluster", "from_cluster")
```

统计

缓存过滤器输出会在 `cluster.<route target cluster>.ratelimit.` 命名空间输出统计数据。429 这一响应码也会出现在[动态 HTTP 统计数据](#)中。

名称	类型	描述
ok	Counter	来自于频率限制服务的所有限制内响应数量
error	Counter	联系频率限制服务时的错误总数
over_limit	Counter	来自频率限制服务的所有超限响应数量

运行时

HTTP 频率限制过滤器支持如下的运行时配置:

- `ratelimit.http_filter_enabled`

调用频率限制服务的请求的百分比, 缺省为 100。

- `ratelimit.http_filter_enforcing`

调用频率限制服务并实施决策的请求的百分比。缺省为 100。这个选项可以用来在完全实施限制之前进行测试, 从而了解频率限制实施产生的后果。

- `ratelimit.<route key>.http_filter_enabled`

在[频率限制配置](#)中指定的 `route_key`, 利用这个数值调用频率限制的请求的百分比。缺省值为 100。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 10:49:14

路由

HTTP 转发是用路由过滤器实现的。在 Envoy 环境中，几乎所有 HTTP 代理场景下都会使用到这一过滤器。该过滤器的主要职能就是执行[路由表](#)中的指令。在重定向和转发这两个主要任务之外，路由过滤器还需要处理重试、统计之类的任务。

[v1 API 参考](#)

[v2 API 参考](#)

HTTP 头

不管是外发/请求，还是接收/响应的过程中，HTTP 头都是由路由过滤器消费和设置的，下面会介绍这些内容：

- [x-envoy-expected-rq-timeout-ms](#)
- [x-envoy-max-retries](#)
- [x-envoy-retry-on](#)
- [x-envoy-retry-grpc-on](#)
- [x-envoy-upstream-alt-stat-name](#)
- [x-envoy-upstream-canary](#)
- [x-envoy-upstream-rq-timeout-alt-response](#)
- [x-envoy-upstream-rq-timeout-ms](#)
- [x-envoy-upstream-rq-per-try-timeout-ms](#)
- [x-envoy-upstream-service-time](#)
- [x-envoy-original-path](#)
- [x-envoy-immediate-health-check-fail](#)
- [x-envoy-overloaded](#)
- [x-envoy-decorator-operation](#)

x-envoy-expected-rq-timeout-ms

以毫秒为单位的时间，路由器要求请求在这一时长内完成。Envoy 把这个信息写入 HTTP Header，这样被代理的主机收到这一请求之后，就可以根据这一节的内容来判断是否超时，也就是快速失败。这一信息会用在内部请求中，按照[x-envoy-upstream-rq-timeout-ms](#) 头或[路由超时](#)的顺序执行。

x-envoy-max-retries

如果使用了[重试策略](#)，且没有指定重试次数，Envoy 缺省会重试一次。重试次数可以显式的在[路由重试配置](#)或者 `x-envoy-max-retries` 头中进行设置。如果[重试策略](#)没有配置，并且 [x-envoy-retry-on](#) 或者 [x-envoy-retry-grpc-on](#) 都没有开启，Envoy 就不会重试失败的请求。

Envoy 的重试功能，有一些需要注意的：

- 路由超时（通过 [x-envoy-upstream-rq-timeout-ms](#) 或者[路由配置](#)进行设置）是包含所有重试的。如果设置请求超时为 3 秒钟，并且第一次请求消耗了 2.7 秒，那么重试（包含补偿）需要在 0.3 秒之内完成。这一设计的目的是避免重试和超时的同步激增。
- Envoy 使用了一种随机的递增算法，以 25 毫秒为基础单位进行补偿。首次重试会随机的选择 0 到 24 毫秒范围内的延时，第二次会在 0 到 74 毫秒之间，第三次就会发生 0 到 175 毫秒之间的延时。
- 如果最大重试次数同时在 HTTP 头和路由配置中都有设置，那么请求的最大重试次数会使用两个配置之中的最大值作为有效值。

x-envoy-retry-on

如果外发请求中设置了这个 Header, Envoy 就会试着重试失败的请求（重试次数缺省为 1, 可以使用 [x-envoy-max-retries](#) 或者[路由重试配置](#)进行控制）。`x-envoy-retry-on` 的值用于表达重试策略。可以使用 `,` 作为分隔符, 同时支持多个条件, 其中包括:

- *5xx*

如果上游服务器响应了 `5xx` 的状态码, 或者完全不响应 (断开、复位或者读超时)。（包含 `connect-failure` 以及 `refused-stream`）。

- 注意: 如果一个请求超出了 [x-envoy-expected-rq-timeout-ms](#) (会出现 504 错误码), Envoy 是不会重试的。
如果在某个尝试消耗太多时间的时候继续重试, 可以使用 [x-envoy-upstream-rq-timeout-ms](#), 这是一个请求的外缘时间限制, 包含了相关的重试时间。

- *gateway-error*

这个策略和 `5xx` 类似, 但是只会在收到 502、503 或者 504 时进行重试。

- *connect-failure*

Envoy 会在连接上游服务器失败的情况下进行重试 (连接超时之类)（包含在 `5xx` 内）。

- 注意: 连接失败/超时是 TCP 层而非请求层的问题。并不包含使用 [x-envoy-upstream-rq-timeout-ms](#) 或者[路由配置](#)进行设置的请求超时。

- *retryable-4xx*

如果上游服务器响应了一个可以重试的 `4xx` 状态码, Envoy 就会进行重试。目前这一类别只包含一个 `409` 状态。

- 注意: 这种策略要当心。有时 `409` 是说明有乐观锁需要更新。这种情况下调用者不应该重试, 而是应该读取然后重试其他的写入, 否则自动重试可能会导致 `409` 的持续出现。

- *refused-stream*

如果上游服务器使用 `REFUSED_STREAM` 错误码复位了数据流, Envoy 就会进行重试。这种复位类型的意义就是说明这种请求是可以安全的进行重试的 (包含在 `5xx` 范围之内)

重试次数可以使用 [x-envoy-max-retries](#) 或者[重试策略](#)进行控制。

注意重试策略还可以在[路由层](#)使用。

缺省情况下, 除非按照上述方案进行配置, 否则 Envoy 不会进行重试操作。

x-envoy-retry-grpc-on

为外发请求设置这个 Header, Envoy 在请求失败时就会重试（重试次数缺省为 1, 可以使用 [x-envoy-max-retries](#) 或者[路由重试配置](#)进行控制）。gRPC 重试目前仅支持状态码包含在响应头中的情况。在尾部包含 gRPC 状态码的情况不会触发重试逻辑。可以使用 `,` 分隔的列表来指定多个策略。支持策略包括:

- *cancelled*

如果 gRPC 响应头中的状态码是 `cancelled` (1), Envoy 会尝试进行重试。

- *deadline-exceeded*

如果 gRPC 响应头中的状态码是 `deadline-exceeded` (4), Envoy 会尝试进行重试。

- *resource-exhausted*

如果 gRPC 响应头中的状态码是 `resource-exhausted` (8), Envoy 会尝试进行重试。

在使用 `x-envoy-retry-grpc-on` header 的同时, 可以使用 [x-envoy-max-retries](#) header 进行重试次数的限制。

注意重试策略还可以在[路由层](#)使用。

缺省情况下，除非按照上述方案进行配置，否则 Envoy 不会进行重试操作。

x-envoy-upstream-alt-stat-name

在外发请求中设置这些 Header，让 Envoy 将上游的响应码和计时统计发送到另外的统计树中，这对于 Envoy 以外的应用级别分类统计很有帮助。这方面的细节，在[输出树文档](#)中有进一步的解释。

这个概念比较容易和定义集群时指定的 `alt_stat_name` 混淆，它指定的是在统计树中根目录下集群的备用名称。

x-envoy-upstream-canary

如果一个上游主机设置了这个 Header，路由会生成金丝雀服务专用的统计。[输出树文档](#)中有更详细的说明。

x-envoy-upstream-rq-timeout-alt-response

在外发请求中设置这个 Header，在请求超时的情况下 Envoy 设置一个 204 的返回码（而不是 504）。这个 Header 的值会被忽略，也就是说只要 Header 存在就可以了。可以参看 [x-envoy-upstream-rq-timeout-ms](#)。

x-envoy-upstream-rq-timeout-ms

在外发请求中设置这个 Header，Envoy 会覆盖[路由配置](#)，超时时间使用毫秒为单位。参看 [x-envoy-upstream-rq-per-try-timeout-ms](#)。

x-envoy-upstream-rq-per-try-timeout-ms

在外发请求中设置这个 Header，Envoy 会为路由的请求设置一个每次尝试的超时，这个超时必须不大于全局的路由超时（参看 [x-envoy-upstream-rq-timeout-ms](#)），否则就会被忽略。这使得调用者可以在进行重试的时候，设置一个更加严格的超时时间，从而控制整体的超时情况。

x-envoy-upstream-service-time

上游主机处理请求所消耗的时间，单位是毫秒。如果客户端希望知道服务处理时间和网络延迟，这个信息就非常有用了。这个 Header 是在响应中设置的。

x-envoy-original-path

如果路由使用了 `prefix_rewrite`，Envoy 就会把原有路径的 Header 保存到这里，以便记录和除错。

x-envoy-immediate-health-check-fail

如果上游主机返回了这个 Header（可以使任何值），Envoy 会立刻假设这个主机的[主动健康检查](#)已经失败（如果集群已经配置了主动健康检查）。这个功能可以通过标准数据面的处理，让上游主机进入故障状态，而无需等待下一次的健康检查。健康检查会让主机重新进入健康状态。[健康检查概述](#)中讲述了更多这方面内容。

x-envoy-overloaded

如果上游设置了这个 Header，Envoy 就不会进行重试了。目前这个 Header 的值是无意义的，只要 Header 本身存在即可。另外 Envoy 如果遇到[管理模式](#)或者上游[熔断](#)的情况，就会在下游响应中设置这个 Header。

x-envoy-decorator-operation

如果入站请求中有这个 Header，他的值会覆盖任何本地由跟踪系统定义的这一值。类似的如果这个 Header 存在于一个外发响应中，他的值也会覆盖客户端的定义。

统计

路由器会在集群命名空间中（依赖集群在所选路由的定义）输出很多统计数据。参考[集群统计](#)一节有更多这方面的信息。

路由过滤器在 `http.<stat_prefix>`. 命名空间输出统计信息。前缀定义来自所属的 HTTP 连接管理器。

名称	类型	描述
no_route	Counter	所有没有路由，返回 404 的请求数总
no_cluster	Counter	目标集群不存在并返回 404 的请求数总
rq_redirect	Counter	收到重定向响应的请求数总
rq_direct_response	Counter	直接响应的请求数总
rq_total	Counter	被路由的请求数总

虚拟机群统计会被输出到 `vhost.<virtual host name>.vcluster.<virtual cluster name>`. 命名空间，包括了下列内容：

名称	类型	描述
<code>upstream/rq<*xx></code>	Counter	HTTP 响应码（也就是 2xx、3xx 等）的聚合
<code>upstream/rq<*></code>	Counter	指定的 HTTP 响应码（也就是 201、302 等）
<code>upstream_rq_time</code>	Histogram	请求时间的毫秒数

运行时

路由过滤器支持下列运行时设置：

- `upstream.base_retry_backoff_ms` 基础的重试时间计数，[HTTP 路由](#)一节有更多讲述。缺省为 25 毫秒。
- `upstream.maintenance_mode.<cluster name>` 会立即得到 503 响应的请求的百分比。他会覆盖所有定义了集群名称的路由行为。可以用于负载清洗、故障注入等。缺省是禁用的。
- `upstream.use_retry` 要进行重试的请求的百分比。这一配置会在任何重试配置之前检查，可以在需要的时候完全禁止任何重试。

Squash

Squash 是 *HTTP* 过滤器，可以允许 *Envoy* 整合 *Squash* 微服务的调试器。具体代码可见：<https://github.com/solo-io/squash>，API Docs：<https://squash.solo.io/>

概述

这种过滤器主要使用场景是在 service mesh 中，例如作为将 Envoy 作为一个 sidecar 发布。在 Squash 集群服务中，当一个请求标记为调试模式进入 mesh 中，Squash Envoy 过滤器将会报告请求为 ‘location’ 模式，由于 Envoy sidecar 和应用程序之间会存在 1 比 1 的映射关系，Squash 服务可以将调试器附加到应用程序中。Squash 过滤器也可以保持请求，直到调试器被附加到应用程序容器中或发生超时。在不对集群做任何调整的情况下，允许开发者在请求到达应用程序时将本地调试器附加到请求的容器上。

配置

- [v1 API reference](#)
- [v2 API reference](#)

如何运行

当 Squash 过滤器遇到请求的header中包含 “x-squash-debug” 这样的值时，将会有如下操作：

1. 推迟请求操作
2. 链接 Squash 服务并请求创建调试附件
 - 在 Squash 服务端，Squash 服务会尝试将调试器附加到应用程序的 Envoy 代理上。如果成功，它会将调试附件的状态更改为附件
3. Squash 服务将会处于等待状态，直到更新调试附件的状态为附件或出现错误状态
4. 恢复请求操作

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 23:06:18

集群管理器

- 集群管理器 架构概览
- v1 API 参考
- v2 API 参考

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 17:56:46

统计

- 概要
- 健康检查统计
- 异常点检测统计
- 动态 HTTP 统计
- 可变树动态 HTTP 统计
- 每个服务区域动态 HTTP 统计
- 负载均衡统计
- 负载均衡子集统计

概要

集群管理器有一个以 `cluster_manager.` 为根的统计树。有以下统计项。统计名称中的任何 `:` 字符将会被替换成 `_`。

名称	类型	描述
<code>cluster_added</code>	Counter	添加的所有集群数 (通过静态配置或 CDS)
<code>cluster_modified</code>	Counter	修改的所有集群数 (通过 CDS)
<code>cluster_removed</code>	Counter	移除的所有集群数 (通过 CDS)
<code>active_clusters</code>	Gauge	当前活跃的 (预热的) 所有集群数
<code>warming_clusters</code>	Gauge	当前正在预热的活动 (非活跃的) 所有集群数

每一个集群有一个以 `cluster..` 为根的统计树。有以下统计项:

名称	类型	描述
<code>upstream_cx_total</code>	Counter	总连接数
<code>upstream_cx_active</code>	Gauge	总激活的连接数
<code>upstream_cx_http1_total</code>	Counter	HTTP/1.1 总连接数
<code>upstream_cx_http2_total</code>	Counter	Total HTTP/2 总连接数
<code>upstream_cx_connect_fail</code>	Counter	总连接失败数
<code>upstream_cx_connect_timeout</code>	Counter	总连接超时数
<code>upstream_cx_idle_timeout</code>	Counter	总连接空闲超时
<code>upstream_cx_connect_attempts_exceeded</code>	Counter	超过配置连接尝试的总连续连接失败数
<code>upstream_cx_overflow</code>	Counter	机群连接断路器溢出的总次数
<code>upstream_cx_connect_ms</code>	Histogram	连接建立毫秒
<code>upstream_cx_length_ms</code>	Histogram	连接长度毫秒
<code>upstream_cx_destroy</code>	Counter	完全破坏连接
<code>upstream_cx_destroy_local</code>	Counter	本地连接破坏
<code>upstream_cx_destroy_remote</code>	Counter	远程连接完全销毁数
<code>upstream_cx_destroy_with_active_rq</code>	Counter	用1 +主动请求销毁总连接数

upstream_cx_destroy_local_with_active_rq	Counter	用1 + 主动请求销毁本地总连接数
upstream_cx_destroy_remote_with_active_rq	Counter	用1 + 主动请求远程销毁总连接
upstream_cx_close_notify	Counter	通过 HTTP/1.1 连接关闭报头或HTTP/2 GOAWAY 关闭的总连接数
upstream_cx_rx_bytes_total	Counter	接收到的总连接字节数
upstream_cx_rx_bytes_buffered	Gauge	当前缓冲的接收连接字节
upstream_cx_tx_bytes_total	Counter	发送的连接字节总数
upstream_cx_tx_bytes_buffered	Gauge	发送当前缓冲的连接字节
upstream_cx_protocol_error	Counter	总连接协议错误
upstream_cx_max_requests	Counter	由于最大请求关闭总连接
upstream_cx_none_healthy	Counter	由于没有健康主机，连接没有建立的总次数
upstream_rq_total	Counter	总请求量
upstream_rq_active	Gauge	总活动请求
upstream_rq_pending_total	Counter	等待连接池连接的总请求
upstream_rq_pending_overflow	Counter	溢出连接池电路中断和失败的总请求
upstream_rq_pending_failure_eject	Counter	由于连接池连接失败而导致的总请求失败
upstream_rq_pending_active	Gauge	挂起连接池连接的全部活动请求
upstream_rq_cancelled	Counter	在获得连接池连接之前取消的总请求
upstream_rq_maintenance_mode	Counter	由于维护模式导致的请求总数为 503
upstream_rq_timeout	Counter	等待响应的总请求
upstream_rq_per_try_timeout	Counter	每次尝试超时的总请求
upstream_rq_rx_reset	Counter	远程重置的总请求
upstream_rq_tx_reset	Counter	本地重置的总请求
upstream_rq_retry	Counter	总请求重试
upstream_rq_retry_success	Counter	总请求重试成功率
upstream_rq_retry_overflow	Counter	由于电路断开而未重试的总请求
upstream_flow_control_paused_reading_total	Counter	上游流量控制暂停读取的总次数
upstream_flow_control_resumed_reading_total	Counter	从上游读取的流量控制的总次数
upstream_flow_control_backed_up_total	Counter	上游连接备份和暂停读取的总次数从下游读取
upstream_flow_control_drained_total	Counter	从上游读取和恢复上游连接的总次数
membership_change	Counter	总簇成员变化
membership_healthy	Gauge	当前集群健康总量（包括健康检查和异常值检测）
membership_total	Gauge	当前群集成员总数
retry_or_shadow_abandoned	Counter	由于缓冲区限制，阴影或重试缓冲的总次数被取消。
config_reload	Counter	由于配置不同，导致的 API 重新加载导致配置重新加载。

update_attempt	Counter	总簇成员更新尝试
update_success	Counter	总簇成员更新成功率
update_failure	Counter	总簇成员更新失败
update_empty	Counter	使用空集群负载分配结束的总群集成员更新，并继续使用以前配置
update_no_rebuild	Counter	没有导致任何集群负载均衡结构重建的完全成功的群集成员更新
version	Gauge	从最后一次成功的 API 获取内容的哈希
max_host_weight	Gauge	集群中任意主机的最大权重
bind_errors	Counter	将套接字绑定到已配置的源地址的总错误

健康检查统计

如果配置了健康检查，则该集群具有根于根 `cluster..health_check.` 的附加统计树。有如下统计：

名称	类型	描述
attempt	Counter	健康检查次数
success	Counter	健康检查成功次数
failure	Counter	立即失效的健康检查（例如HTTP 503）以及网络故障的次数
passive_failure	Counter	由于被动事件引起的健康检查失败的次数（例如X-Enviv-即时健康检查失败）
network_failure	Counter	网络错误引起的健康检查失败次数
verify_cluster	Counter	尝试群集名称验证的健康检查数量
healthy	Gauge	健康会员人数

异常点检测统计

如果为集群配置了[异常点检测](#)，统计将以 `cluster..outlier_detection.` 为根，包含如下：

名称	类型	描述
ejections_enforced_total	Counter	由于异常值类型强制执行的驱逐次数
ejections_active	Gauge	当前驱逐主机的数量
ejections_overflow	Counter	由于最大驱逐率而中止的驱逐数
ejections_enforced_consecutive_5xx	Counter	强制执行的连续 5xx 驱逐数
ejections_detected_consecutive_5xx	Counter	检测到的连续 5xx 驱逐数（即使未强制执行）
ejections_enforced_success_rate	Counter	强制成功率异常离群数
ejections_detected_success_rate	Counter	检测成功率离群值的数量（即使未执行）
ejections_enforced_consecutive_gateway_failure	Counter	强制连续网关失败驱逐数
ejections_detected_consecutive_gateway_failure	Counter	检测到的连续网关故障驱逐数（即使未强制执行）

ejections_total	Counter	不赞成的由于异常值类型引起的驱逐次数（即使未执行）
ejections_consecutive_5xx	Counter	不赞成的连续5xx驱逐数（即使未强制执行）

动态 HTTP 统计

如果使用 HTTP，动态 HTTP 响应代码统计也是可用的。这些是由各种内部系统发出的，以及一些过滤器，如[路由器过滤器](#)和[速率限制过滤器](#)。统计将以 `cluster..` 为根，包含如下：

名称	类型	描述
<code>upstream/rq<*xx></code>	Counter	HTTP响应代码汇总（例如，2xx 3xx，等。）
<code>upstream/rq<*></code>	Counter	特异性的HTTP响应代码（例如，201、302等。）
<code>upstream_rq_time</code>	Histogram	请求时间ms
<code>canary.upstream/rq<*xx></code>	Counter	上游的 canary 聚合 HTTP 响应代码
<code>canary.upstream/rq<*></code>	Counter	上游的 canary HTTP 特定响应代码
<code>canary.upstream_rq_time</code>	Histogram	上游的 canary 请求时间 ms
<code>internal.upstream/rq<*xx></code>	Counter	内部原始聚合 HTTP 响应代码
<code>internal.upstream/rq<*></code>	Counter	内部原始特定 HTTP 响应代码
<code>internal.upstream_rq_time</code>	Histogram	内部原始请求时间 ms
<code>external.upstream/rq<*xx></code>	Counter	HTTP响应代码聚集外部性
<code>external.upstream/rq<*></code>	Counter	HTTP响应代码和特异性
<code>external.upstream_rq_time</code>	Histogram	外部原始请求时间 ms

可变树动态 HTTP 统计

如果配置了可变树统计信息，则它们将存在于 `cluster..` 命名空间。所产生的统计数据与动态 HTTP 统计部分[以上](#)所记录的数据相同。

每个服务区域动态 HTTP 统计

如果服务区域可用于本地服务（通过 `--service-zone`）和[上游集群](#)，Envoy 在将跟踪 `cluster..zone..` 命名空间的以下统计数据。

名称	类型	描述
<code>upstream/rq<*xx></code>	Counter	聚合 HTTP 响应代码（例如，2xx 3xx，等。）
<code>upstream/rq<*></code>	Counter	特定的 HTTP 响应代码（例如，201 等）
<code>upstream_rq_time</code>	Histogram	请求时间 ms

负载均衡统计

负载均衡器决策监测统计。统计以 `cluster..` 为根，包含如下统计：

名称	类型	描述
lb_recalculate_zone_structures	Counter	重新生成局部感知路由结构的次数，用于上游位置选择的快速决策
lb_healthy_panic	Counter	在恐慌模式下与负载平衡器平衡的总请求负载
lb_zone_cluster_too_small	Counter	由于上游簇大小小，无区域感知路由
lb_zone_routing_all_directly	Counter	将所有请求直接发送到同一区域
lb_zone_routing_sampled	Counter	向同一区域发送一些请求
lb_zone_routing_cross_zone	Counter	区域感知路由模式，但必须发送跨区域
lb_local_cluster_not_ok	Counter	本地主机集未设置或是本地集群的恐慌模式
lb_zone_number_differs	Counter	本地和上游集群的区域数目不同
lb_zone_no_capacity_left	Counter	舍入误差导致随机区域选择结束的次数

负载均衡子集统计

负载均衡器子集 决策的统计监测。统计以 *cluster..* 为根，包含如下统计：

名称	类型	描述
lb_subsets_active	Gauge	当前可用子集的数目
lb_subsets_created	Counter	创建的子集数
lb_subsets_removed	Counter	由于没有主机而删除的子集数
lb_subsets_selected	Counter	选择任何子集的负载平衡次数
lb_subsets_fallback	Counter	调用回退策略的次数

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 10:30:01

运行时

上游集群支持以下运行时设置：

主动健康检查

- `health_check.min_interval`

健康检查的最小值 `interval`。默认值是0。健康检查 `interval` 的值将位于 `min_interval` 和 `max_interval` 之间。

- `health_check.max_interval`

健康检查的最大值 `interval`。默认值是 `MAX_INT`。健康检查 `interval` 的值将位于 `min_interval` 和 `max_interval` 之间。

- `health_check.verify_cluster`

当[健康检查过滤器](#)将远程服务集群写入响应时，将对[预期的上游服务](#)验证什么百分比的健康检查请求。

异常点检测

查看异常点检测[架构概览](#) 取得更多异常点检测的信息。异常点检测支持的运行时参数和[静态配置参数](#)一样，分别为：

- `outlier_detection.consecutive_5xx`

`consecutive_5XX` 在异常点检测中的配置

- `outlier_detection.consecutive_gateway_failure`

`consecutive_gateway_failure` 在异常点检测中的配置

- `outlier_detection.interval_ms`

`interval_ms` 在异常点检测中的配置

- `outlier_detection.base_ejection_time_ms`

`base_ejection_time_ms` 在异常点检测中的配置

- `outlier_detection.max_ejection_percent`

`max_ejection_percent` 在异常点检测中的配置

- `outlier_detection.enforcing_consecutive_5xx`

`enforcing_consecutive_5xx` 在异常点检测中的配置

- `outlier_detection.enforcing_consecutive_gateway_failure`

`enforcing_consecutive_gateway_failure` 在异常点检测中的配置

- `outlier_detection.enforcing_success_rate`

`enforcing_success_rate` 在异常点检测中的配置

- `outlier_detection.success_rate_minimum_hosts`

`success_rate_minimum_hosts` 在异常点检测中的配置

- outlier_detection.success_rate_request_volume
[success_rate_request_volume](#) 在异常点检测中的配置
- outlier_detection.success_rate_stdev_factor
[success_rate_stdev_factor](#) 在异常点检测中的配置

核心

- upstream.healthy_panic_threshold
设置[恐慌阈](#)百分比。默认达到 50%。
- upstream.use_http2
如果配置的话，集群是否使用 [http2 特征](#)。设置为0以禁用HTTP / 2，即使配置了该功能。默认值是关闭。
- upstream.weight_enabled
用来打开或者关闭权重负载均衡的二级制开关。如果设置成非0数值，按权重负载均衡的功能是打开的。默认值是打开。

Zone 感知负载均衡

- upstream.zone_routing.enabled
多大百分比的请求将会被路由到相同上游 zone。默认是 100% 请求。
- upstream.zone_routing.min_cluster_size
某个上游集群能被 zone 感知尝试的最小值。默认值是 6。如果上游集群数值比 [min_cluster_size](#) 小，zone 路由感知将不被执行。

断路

- circuit_breakers...max_connections
[断路器设置最大连接数](#)
- circuit_breakers...max_pending_requests
[断路器设置最大等待数](#)
- circuit_breakers...max_requests
[断路器设置最大请求数](#)
- circuit_breakers...max_retries
[断路器设置最大重试次数](#)

集群发现服务 (CDS)

集群发现服务 (CDS) 是一个可选的 API，Envoy 将调用该 API 来动态获取集群管理成员。Envoy 还将根据 API 响应协调集群管理，根据需要完成添加、修改或删除已知的集群。

注意

在 Envoy 配置中静态定义的任何集群都不能通过 CDS API 进行修改或删除。

- [v1 CDS API](#)
- [v2 CDS API](#)

统计

CDS 的统计树以 `cluster_manager.cds.` 为根，统计如下：

名字	类型	描述
config_reload	Counter	因配置不同而导致配置重新加载的总次数
update_attempt	Counter	尝试调用配置加载API的总次数
update_success	Counter	调用配置加载API成功的总次数
update_failure	Counter	调用配置加载API失败的总次数（网络或参数错误）
version	Gauge	来自上次成功调用配置加载API的内容哈希

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-29 14:57:12

健康检查

[健康检查架构概览](#)。

- 如果为集群配置运行状况检查，会有额外的数据发出。记录在[这里](#)。
- [v1 API 参考文档](#)。
- [v2 API 参考文档](#)。

TCP 健康检查

注意 本节是为 v1 API 编写的，但这些概念也适用于 v2 API。针对 V2 API 将在未来的正式版中重新定义。

执行的匹配类型如下（这是MongoDB运行状况检查请求和响应）：

```
{
  "send": [
    {"binary": "39000000"}, {"binary": "EEEEEEEE"}, {"binary": "00000000"}, {"binary": "d4070000"}, {"binary": "00000000"}, {"binary": "746573742e"}, {"binary": "24636d6400"}, {"binary": "00000000"}, {"binary": "FFFFFFF"}, {"binary": "13000000"}, {"binary": "01"}, {"binary": "70696e6700"}, {"binary": "000000000000f03f"}, {"binary": "00"}, ],
  "receive": [
    {"binary": "EEEEEEEE"}, {"binary": "01000000"}, {"binary": "00000000"}, {"binary": "0000000000000000"}, {"binary": "00000000"}, {"binary": "11000000"}, {"binary": "01"}, {"binary": "6f6b"}, {"binary": "000000000000f03f"}, {"binary": "00"} ]
}
```

在每个运行状况检查周期中，所有 "send" 字节都会发送到目标服务器。每个二进制块的长度可以是任意的，并且在发送时只是连接在一起。（分离成多个块可用于可读性）。

在检查响应时，执行“模糊”匹配，以便每个二进制块必须被找到，并且按照指定的顺序，但不一定是连续的。因此，在上面的示例中，可以在“EEEEEEEE”和“01000000”之间的响应中插入“FFFFFFF”，并且该检查仍然会通过。这样做是为了支持将非确定性数据（如时间）插入到响应中的协议。

健康检查需要更复杂的模式，如发送/接收/发送/接收目前不可能。

如果 “receive” 是一个空数组，则 Envoy 将执行 "connect only" TCP 健康检查。在每个周期中，Envoy 将尝试连接到上游主机，并且如果连接成功，则认为它是成功的。每个健康检查周期都会创建一个新连接。

断路

- 断路[架构概览](#)
- [v1 API 文档](#)
- [v2 API 文档](#)

运行时

所有的断路设置都是运行时可配置的，是基于集群名称的优先级定义。他们遵循以下命名方案 `circuit_breakers.<cluster_name>.<priority>.<setting>`。`cluster_name` 是每个群集配置中的名称字，设置在envoy的[配置文件](#)中。可用的运行时设置将覆盖envoy配置文件中的配置。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:12:20

Redis

Redis 健康检查器是一个用于检查 Redis 上游主机的定制检查器。主要通过发送 PING 命令和接收 PONG 命令进行工作。上游 Redis 主机可以使用 PONG 以外的任何其他响应来导致立即激活的运行状况检查失败。或者，Envoy 可以在用户指定的密钥上执行 EXISTS。如果密钥不存在，则认为它是合格的健康检查。这允许用户通过将执行的密钥设置为任意值并等待流量耗尽来标记 Redis 实例以进行维护。

- [v2 API reference](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:40:42

访问日志

配置

访问日志配置是 [HTTP 连接管理器配置](#) 或者 [TCP 代理配置](#) 中的一部分。

- [v1 API 参考](#)
- [v2 API 参考](#)

格式规则

访问日志格式字符串包含了命令操作符以及其他文本。访问日志的格式化过程并不会对换行符做出什么假设，所以应该将其设置为格式字符串的组成部分。请参阅示例的[缺省格式](#)。要注意的是，访问日志会在未设置、或者空值的位置加入一个字符：“-”。

不同类型的访问日志（例如 HTTP 和 TCP）共用同样的格式字符串。不同类型的日志中，某些字段可能会有不同的含义。

下面是支持的命令：

- **%START_TIME%**
 - *HTTP*
Request 的开始时间，精确到毫秒。
 - *TCP*
下游连接的开始时间，精确到毫秒。
可以用[格式化字符串](#)对 START_TIME 进行定制，例如：%START_TIME(%Y/%m/%dT%H:%M:%S%z %s)%
- **%BYTES RECEIVED%**
 - *HTTP*
收到的 Body 字节数。
 - *TCP*
下游连接上收到的字节数。
- **%PROTOCOL%**
 - *HTTP*
协议。目前取值为 *HTTP/1.1* 或 *HTTP/2*。
 - *TCP*
未实现（“-”）。
- **%RESPONSE_CODE%**
 - *HTTP*
HTTP 响应码。注意如果响应码为 0，说明服务器从未发送过响应，这一般是说明（下游）客户端断开了。
 - *TCP*

未实现 (“-”)。

- **%BYTES_SENT%**

- *HTTP*

发送出去的 Body 字节数。

- *TCP*

在连接中向下游发送的字节数。

- **%DURATION%**

- *HTTP*

一个请求从开始到最后一个字节传输完成的持续时间，单位是毫秒。

- *TCP*

下游连接的持续时间。

- **%RESPONSE_FLAGS%**

关于响应或连接的附加细节。这里描述的响应码是不适用于 TCP 连接的。可能的取值包括：

- *HTTP 和 TCP*

- *UH*: 503 响应码的补充信息，上游集群中没有健康的上游主机。
 - *UF*: 503 响应码的补充信息，上游连接失败。
 - *UO*: 503 响应码的补充信息，上游溢出（**熔断状态**）。
 - *NR*: 404 响应码的补充信息，给定请求没有**路由配置**。

- *HTTP 专用*

- *LH*: 503 响应码的补充信息，本地服务**健康检查**失败。
 - *UT*: 504 响应码的补充信息，上游请求超时。
 - *LR*: 503 响应码的补充信息，连接在本地被复位。
 - *UR*: 503 响应码的补充信息，上游复位连接。
 - *UC*: 503 响应码的补充信息，上游链接终止。
 - *DI*: 该请求受**错误注入**功能影响，延迟指定时间。
 - *FI*: 这一请求受**错误注入**功能影响，返回了指定的状态码。
 - *RL*: 429 响应码的补充信息，**HTTP 频率限制过滤器**的配置限制了这一请求。

- **%UPSTREAM_HOST%**

上游主机的 URL (对 TCP 链接来说就是：`tcp://ip:port`)

- **%UPSTREAM_CLUSTER%**

上游主机所属的上游集群。

- **%UPSTREAM_LOCAL_ADDRESS%**

上游连接的本地地址，如果是 IP 地址，那么其中会包含地址和端口。

- **%DOWNSTREAM_REMOTE_ADDRESS%**

下游连接的远程地址，如果是 IP 地址，那么其中会包含地址和端口。

注意：如果地址来自于 `proxy proto` 或者 `x-forwarded-for` 这可能不是远端节点的物理地址。

- **%DOWNSTREAM_REMOTE_ADDRESS_WITHOUT_PORT%**

下游连接的远程地址，如果是 IP 地址，那么其中不会包含地址和端口。

注意：如果地址来自于 `proxy proto` 或者 `x-forwarded-for` 这可能不是远端节点的物理地址。

- **%DOWNSTREAM_LOCAL_ADDRESS%**

下游连接的本地地址。如果是 IP 地址，那么其中会包含地址和端口。如果原始连接来自于 iptables 的 REDIRECT，那么这里的原始目标地址会由[原始目标过滤器](#)通过 `SO_ORIGINAL_DST` 这一 Socket 选项进行获取。如果原始连接是来自 iptables 的 TPROXY，并且监听器的 `transparent` 选项处于开启状态，这里会显示原始的目标地址和端口。

- **%DOWNSTREAM_LOCAL_ADDRESS_WITHOUT_PORT%**

和 `%DOWNSTREAM_LOCAL_ADDRESS%` 类似，但是不包含端口部分。

- **%REQ(X?Y):Z%**

- *HTTP*

一个 HTTP 请求 Header，X 是主要 HTTP 头，Y 是备选 Header，Z 是一个可选的参数，表示截取的字符串的长度。这个值来源于 HTTP Header，首先读取 X，如果 X 没有设置，则读取 Y。如果两个 Header 都没有，在日志中就会写入“-”

- *TCP*

未实现 (“-”)。

- **%RESP(X?Y):Z%**

- *HTTP*

和 `%REQ(X?Y):Z%` 类似，只不过值来自于响应而非请求。

- *TCP*

未实现 (“-”)。

- **%TRAILER(X?Y):Z%**

- *HTTP*

和 `%REQ(X?Y):Z%` 类似，只不过值来自于响应尾部。

- *TCP*

未实现 (“-”)。

- **%DYNAMIC_METADATA(NAMESPACE:KEY):Z%***

- *HTTP*

[动态元数据](#)信息，NAMESPACE 就是设置用于过滤命名空间的参数，KEY 是一个可选参数，用于在命名空间内查询指定的键，可以用 `:` 来表达嵌套关系，Z 是一个可选的参数，表示截断字符串的长度。动态元数据可以使用 `RequestInfo` 进行设置：`setDynamicMetadata`。这个数据以 JSON 形式进行记录，例如下面的动态元数据：

```
com.test.my_filter: {"test_key": "foo", "test_object": {"inner_key": "bar"}}
```

- `%DYNAMIC_METADATA(com.test.my_filter)%` 会记录： `{"test_key": "foo", "test_object": {"inner_key": "bar"}}`
- `%DYNAMIC_METADATA(com.test.my_filter:test_key)%` 会记录： `"foo"`
- `%DYNAMIC_METADATA(com.test.my_filter:test_object)%` 会记录： `{"inner_key": "bar"}`
- `%DYNAMIC_METADATA(com.test.my_filter:test_object:inner_key)%` 会记录： `"bar"`
- `%DYNAMIC_METADATA(com.unknown_filter)%` 会记录： `-`
- `%DYNAMIC_METADATA(com.test.my_filter:unknown_key)%` 会记录： `-`
- `%DYNAMIC_METADATA(com.test.my_filter):25%` 会记录（截取到 25 个字符）：`{"test_key": "foo", "test``

- *TCP*

未实现 (“-”)。

缺省格式

如果没有设置自定义格式，Envoy 会使用如下的缺省格式：

```
[%START_TIME%] "%REQ(:METHOD)% %REQ(X-ENVOY-ORIGINAL-PATH?:PATH)% %PROTOCOL%"  
%RESPONSE_CODE% %RESPONSE_FLAGS% %BYTES_RECEIVED% %BYTES_SENT% %DURATION%  
%RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)% "%REQ(X-FORWARDED-FOR)%" "%REQ(USER-AGENT)%"  
"%REQ(X-REQUEST-ID)%" "%REQ(:AUTHORITY)%" "%UPSTREAM_HOST%\n
```

缺省的访问日志格式示例：

```
[2016-04-15T20:17:00.310Z] "POST /api/v1/locations HTTP/2" 204 - 154 0 226 100 "10.0.35.28" "nsq2http" "cc21d9b0-cf5c-432b  
-8c7e-98aeb7988cd2" "locations" "tcp://10.0.2.1:80"
```

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 19:40:46

速率限制服务

速率限制服务指定了全局速率限制服务 Envoy 在需要制定全局速率限制决策时应该与之交互。如果没有配置速率限制服务，则会是用“null”服务，如果配置了则会始终返回“ok”。

- [v1 API 参考](#)
- [v2 API 参考](#)

gRPC service IDL

Envoy 期望速率限制服务支持 </source/common/ratelimit/ratelimit.proto> 指定GRPC IDL。关于 API 的更多信息见IDL文件。点击[这里](#)看 Lyft 的参考实例。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 23:11:34

运行时

运行时配置指定了包含可以重载配置元素的本地文件系统数的位置。值可以在 [/runtime admin endpoint](#) 查看。值可以在 [/runtime_modify admin endpoint](#) 修改和追加。如果没用进行运行时配置，则会使用空提供程序，该提供程序会使用代码中内置的除了通过 [/runtime_modify](#) 添加的值之外的所有缺省值。

注意

要小心使用 [/runtime_modify](#) 端点。其变更是立即生效的。管理接口的[妥善的保护](#)是非常重要的。

- [v1 API 参考](#)
- [v2 API 参考](#)

文件系统设计

配置指南的各个部分描述了可用的运行时设置。例如，[这里](#)是上游集群的运行时配置。

假定文件夹 `/srv/runtime/v1` 指向存储全局运行时配置的实际文件系统路径。以下是运行时的典型配置参数：

- `symlink_root: /srv/runtime/current`
- `subdirectory: envoy`
- `override_subdirectory: envoy_override`

这里 `/srv/runtime/current` 是到 `/srv/runtime/v1` 的符号链接。

运行时配置键中的每一个‘.’都代表层次结构中的一个新的目录，扎根于 `symlink_root + subdirectory`。例如，键 `health_check.min_interval` 将具有以下完整文件系统路径（使用符号链接）：

```
/srv/runtime/current/envoy/health_check/min_interval
```

路径的末端的部分是文件。文件的内容构成运行时值。从文件读取数值时，空格和新行将被忽略。

`override_subdirectory` 与 `--service-cluster` 在命令行界面操作时一起使用的。假如 `--service-cluster` 被设置成为了 `my-cluster`。Envoy 将首先从下面完整的文件系统路径中找 `health_check.min_interval` 项：

```
/srv/runtime/current/envoy_override/my-cluster/health_check/min_interval
```

如果找到了，该值将覆盖在主查找路径中找到的任何值。这允许用户在全局默认值之上自定义单个群集的运行时值。

注释

行首为 # 的行是注释。

注释可以提供现有值的上下文。注释在其他空文件中也很有用，其可以在需要的时候保留占位符以进行部署。

通过符号链接交换更新运行时值

更新运行时值总共有两步。第一步，创建整个运行时树的硬拷贝并更新所需的运行时值。第二步，使用以下命令的等价物，将旧树中的符号链接根自动交换到新的运行时树：

```
/srv/runtime:~$ ln -s /srv/runtime/v2 new && mv -Tf new current
```

关于如何部署文件系统数据，如何收集垃圾等操作超出了本文的范围。

统计

文件系统运行时提供程序在运行时发出一些统计信息。命名空间。

名称	类型	描述
load_error	Counter	错误重新尝试加载的总数
override_dir_not_exists	Counter	未使用覆盖目录的加载总数
override_dir_exists	Counter	使用覆盖目录的加载总数
load_success	Counter	成功加载的尝试总数
num_keys	Gauge	当前加载的键数

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 19:33:47

统计

发出一些统计数据来统计系统行为:

名称	类型	描述
stats.overflow	Counter	由于共享内存不足, Envoy 无法分配统计的总次数

Server

根植与 *server*. 与服务器相关的统计信息, 统计信息如下:

名称	类型	描述
uptime	Gauge	当前 server 的启动时间
memory_allocated	Gauge	当前分配的内存大小, 以字节为单位
memory_heap_size	Gauge	当前堆预留的内存大小, 以字节为单位
live	Gauge	1: 当前server还没有耗尽; 0: 其他
parent_connections	Gauge	在热启动中所有就的 Envoy 进程的连接数
total_connections	Gauge	所有的 Envoy 进程的连接数, 包括新的、旧的
version	Gauge	整型表示基于 SCM 修订的版本号
days_until_first_cert_expiring	Gauge	下一个证书到期的天数

File system

在 *filesystem*. 中发出的与文件系统相关的统计信息, 名称空间

名称	类型	描述
write_buffered	Counter	文件数据被移动到 Envoy 内部缓冲区的总次数
write_completed	Counter	文件被写入的总次数
flushed_by_timer	Counter	由于刷新超时而导致内部刷新缓冲区写入文件的总次数
reopen_failed	Counter	文件被打开失败的总次数
write_total_buffered	Gauge	当前内部数据缓冲区的总大小, 以字节为单位

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-21 17:11:43

路由表检查工具

注意:下面的配置仅用于路由表检查工具，而不是 Envoy 二进制文件的一部分。路由表检查工具是一个独立的二进制文件，它可以用来验证 Envoy 对给定配置文件的路由。

下面指定了路由表检查工具的输入。路由表检查工具检查[路由器](#)返回的路由信息是否符合预期。该工具可用于检查集群名称、虚拟集群名称、虚拟主机名、手动路径重写、手动主机重写、路径重定向和头字段匹配。可以添加其它测试用例的扩展。安装工具和示例工具输入/输出的详细信息可以在[这里](#)找到。

路由表检查工具配置由一系列 json 测试对象组成。每个测试对象由三个部分组成。

- Test name

该字段指定每个测试对象的名称。

- Input values

输入值字段指定要传递给路由器的参数。例如输入字段包括 `:authority`、`:path`、`:method` `header fields`。权限：`path` 字段指定发送到路由器的 `url` 并是必须要添加的。所有其他输入字段都是可选的。

- Validate

`validate` 字段指定要检查的期望值和测试用例。至少需要一个测试用例。

看下面的一个简单的json工具配置参数，其包含有一个测试用例。测试期望与“instant-server”的集群名称匹配。：

```
[
  {
    "test_name": "Cluster_name_test",
    "input": {
      ":authority": "api.lyft.com",
      ":path": "/api/locations"
    },
    "validate": {
      "cluster_name": "instant-server"
    }
  }
]
```

```
[
  {
    "test_name": "...",
    "input": {
      ":authority": "...",
      ":path": "...",
      ":method": "...",
      "internal": "...",
      "random_value": "...",
      "ssl": "...",
      "additional_headers": [
        {
          "field": "...",
          "value": ...
        },
        {
          ...
        }
      ]
    },
    "validate": {
      "cluster_name": ...
    }
  }
]
```

```

    "virtual_cluster_name": "...",
    "virtual_host_name": "...",
    "host_rewrite": "...",
    "path_rewrite": "...",
    "path_redirect": "...",
    "header_fields": [
        {
            "field": "...",
            "value": ...
        },
        {
            ...
        }
    ]
},
{
    ...
}
]

```

- **test_name**

(required, string) 测试对象的名称。

- **input**路径法

(required, object) 输入值发送到路由器，以确定返回的路由信息：

`:authority (required, string) :` url 权限。这个值与 path 参数一起定义要匹配的url。例如权限的值是“`api.lyft.com`”。

`:path (required, string) :` url 的路径。例如路径的值是“`/foo`”。

`:method (optional, string) :` 请求的方法。如果没有指定，默认采用 get 方法。这里可以选择 GET 方法, PUT 方法或者 POST 方法。

`internal (optional, boolean) :` 它是否将 `x-envoy-internal` 设置为“true”的一个标志。如果没有指定或者指定为“false”。

`x-envoy-internal` 就没有配置。`random_value (optional, integer)` 用于确定加权集群选择目标的整数。默认值是0。

`ssl (optional, boolean) :` 决定将 `x-forwarded-proto` 设置为 `http` 或者 `https`。通过给定的协议设置 `x-forwarded-proto`，该工具能够模拟客户机通过 `http` 或 `https` 发出请求的行为。默认情况下 `ssl` 是 `false`，相当于将 `x-forwarded-proto` 设置为 `http`。

`additional_headers (optional, array) :` Additional headers 将作为路径确定的输入添加。

“`:authority`”, “`:path`”, “`:method`”, “`x-forwarded-proto`”以及“`x-envoy-internal`”字段应该有其他配置选项指定，而不应该在这里配置。`field (required, string)` 要添加的头字段的名称。`value (required, string)` 要添加的投字段的值。

- **validate**

(required, object) validate 对象指定要匹配的返回路由参数。至少需要指定一个参数，使用“”(空字符串)来表示没有返回值。例如，测试没有集群能够匹配 {“`cluster_name`”: “”}。

`cluster_name (optional, string) :` 匹配集群的名称。

`virtual_cluster_name (optional, string) :` 匹配虚拟集群的名称。

`virtual_host_name (optional, string) :` 匹配虚拟机的名称。

`host_rewrite (optional, string) :` 匹配重写后的主机字段头。

`path_rewrite (optional, string) :` 匹配重写后的路径字段头。

`path_redirect (optional, string)` : 匹配返回的重定向路径。

`header_fields (optional, array)` : 匹配列出的头字段。

例如头字段包括 “`:path`”, “`cookie`”, 以及 “`date`” 字段。在其它测试用例之后检查头字段。因此, 所检查的头字段将是重定向或重写路由信息。`field (required, string)` 要匹配的头字段的名称。`value (required, string)` 要匹配的头字段的值。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-22 23:16:34

命令行选项

Envoy 由 JSON 配置文件和一组命令行选项一起驱动。以下是 Envoy 支持的命令行选项。

- `-c <path string>, --config-path <path string>`

(可选) v1 或 v2 JSON/YAML/proto3 配置文件的路径。若未设置此选项，需要指定 `--config-yaml` 选项。它会首先作为 v2 引导配置文件进行解析，若解析失败，会根据 [`--v2-config-only`] 选项决定是否作为 v1 JSON 配置文件进行解析。对于 v2 的配置文件，有效的扩展名包括 `.json`、`.yaml`、`.pb` 和 `.pb_text`，分别表示 JSON、YAML、二进制 proto3 和文本 proto3 格式。

- `--config-yaml <yaml string>`

(可选) 一个 YAML 字符串，作为 v2 引导配置。若同时设置了 `--config-path`，此 YAML 字符串的值会被合并到 `--config-path` 指定的引导配置并覆盖相应选项。由于 YAML 是 JSON 的超集，也可以给 `--config-yaml` 传入一个 JSON 字符串。`--config-yaml` 与 v1 引导配置不兼容。

一个通过命令行覆盖节点 id 的例子：

```
./envoy -c bootstrap.yaml --config-yaml "node: {id: 'node1'}"
```

- `--v2-config-only`

(可选) 此标识决定配置文件是否应该只被解析为 v2 引导配置文件。若传入 `false`（默认选项），v2 引导配置解析失败时，会再次尝试将其解析为 v1 JSON 配置文件。

- `--mode <string>`

(可选) Envoy 的执行模式之一：

- `serve`：（默认选项）校验 JSON 配置，然后正常提供服务。
- `validate`：校验 JSON 配置，然后退出。会打印一条“OK”消息（若退出码为 0）或所有配置文件产生的错误（若退出码为 1）。不会产生网络流量，热重启流程也不会执行，所以不会影响到机器上其他的 Envoy 进程。

- `--admin-address-path <path string>`

(可选) 输出管理地址和端口的文件路径。

- `--local-address-ip-version <string>`

(可选) 填充服务器本地 IP 地址使用的 IP 地址版本。此参数影响各种头部信息，包括附加到 X-Forwarded-For (XFF) 头部的内容。选项是 `v4` 或 `v6`。默认是 `v4`。

- `--base-id <integer>`

(可选) 分配共享内存区域时使用的基本 ID。Envoy 在热重启时使用共享内存区域。大部分用户永远都不需要设置这个选项。不过，若需要在同一台机器上多次运行 Envoy，每个运行的 Envoy 需要一个唯一的基本 ID，以免共享内存区域产生冲突。

- `--concurrency <integer>`

(可选) 启动的工作线程数。若未指定则默认使用机器上的硬件线程数。

- `-l <string>, --log-level <string>`

(可选) 日志级别。非开发者通常不应该设置这个选项。有关可用的日志级别和默认值，请参阅帮助文本。

- `--log-path <path string>`

(可选) 输出日志的文件路径。处理 SIGUSR1 时，该文件将被重新打开。若未设置，输出到 stderr。

- `--log-format <format string>`

(可选) 用于格式化日志消息元数据的格式字符串。若未设置, 会使用一个默认的格式字符串 "[%Y-%m-%d %T.%e] [%t][%l][%n] %v".

支持的格式化标记有 (包括示例输出) :

参数	解释
%v:	要记录的实际消息 ("some user text")
%t:	线程 id ("1232")
%P:	进程 id ("3456")
%n:	记录器名称 ("filter")
%l:	消息的日志级别 ("debug"、 "info")
%L:	消息的日志级别缩写 ("D"、 "I"等)
%a:	星期的缩写呈现 ("Tue")
%A:	星期的完整名称 ("Tuesday")
%b:	月份的缩写呈现 ("Mar")
%B:	月份的完整名称 ("March")
%c:	日期和时间的呈现 ("Tue Mar 27 15:25:06 2018")
%C:	年份的 2 位呈现 ("18")
%Y:	年份的 4 位呈现 ("2018")
%D, %x:	日期 MM/DD/YY 格式的缩写呈现 ("03/27/18")
%m:	月份 01-12 ("03")
%d:	月中的日期 01-31 ("27")
%H:	24 小时制的小时 00-23 ("15")
%I:	12 小时制的小时 01-12 ("03")
%M:	分钟 00-59 ("25")
%S:	秒数 00-59 ("06")
%e:	当前秒中的毫秒部分 000-999 ("008")
%f:	当前秒中的微秒部分 000000-999999 ("008789")
%F:	当前秒中的纳秒部分 00000000-99999999 ("008789123")
%p:	上午/下午 AM/PM ("AM")
%r:	12 小时制的钟表呈现 ("03:25:06 PM")
%R:	24 小时制 HH:MM 格式的时间, 等同于 %H:%M ("15:25")
%T, %X:	ISO 8601 时间格式 (HH:MM:SS), 等同于 %H:%M:%S ("13:25:06")
%z:	ISO 8601 与 UTC 的时区偏移 ([+/-]HH:MM) ("-07:00")
%%:	% 符号 ("%")

- `--restart-epoch <integer>`

(可选) 热重启周期 (Envoy 被热重启而不是全新启动的次数)。对于第一次启动默认为 0。此选项告诉 Envoy 是尝试创建，还是打开一个已存在的热重启所需的共享内存区域。每次热重启后它都应该被增加。多数情况下热重启包装器设置的 `RESTART_EPOCH` 环境变量应该被传递给这个选项。

- `--hot-restart-version`

(可选) 为当前的二进制文件输出一个热重启兼容性版本。可以将其与 `GET /hot_restart_version` 管理接口的输出进行比较，以确定新的二进制文件和正在运行的二进制文件是否热重启兼容。

- `--service-cluster <string>`

(可选) 定义 Envoy 运行的本地服务集群名称。本地服务集群名称首先来自[引导节点](#)消息的 `cluster` 字段。此命令行选项为指定此值提供了另一种方法，并将覆盖引导配置中设置的任何值。若使用了以下任何功能，则应该通过此命令行选项或引导配置来设置它：[statsd](#), [健康检查集群验证](#), [运行时覆盖目录](#), [User Agent 添加](#), [HTTP 全局速率限制](#), [CDS](#), 和 [HTTP 跟踪](#)。

- `--service-node <string>`

(可选) 定义 Envoy 运行的本地服务节点名称。本地服务节点名称首先来自[引导节点](#)消息的消息的 `id` 字段。此命令行选项为指定此值提供了另一种方法，并将覆盖引导配置中设置的任何值。若使用了以下任何功能，则应该通过此命令行选项或引导配置来设置它：[statsd](#), [CDS](#) 和 [HTTP 跟踪](#)。

- `--service-zone <string>`

(可选) 定义 Envoy 运行的本地服务区域名称。本地服务区域名称首先来自[引导节点](#)消息的消息的 `locality.zone` 字段。此命令行选项为指定此值提供了另一种方法，并将覆盖引导配置中设置的任何值。若使用了发现服务路由且发现服务暴露出[区域数据](#)，则应该通过此命令行选项或引导配置来设置它。区域的含义是依赖于上下文的，如 AWS 上的[可用性区域 \(AZ\)](#)，GCP 上的[区域](#)，等等。

- `--file-flush-interval-msec <integer>`

(可选) 文件缓冲区刷新间隔（毫秒）。默认为 10 秒。此设置在文件创建期间用于确定缓冲区刷新到文件的间隔时间。缓冲区在每次写满时或每次间隔过后都会刷新，以先到者为准。调整此设置在跟踪输出[访问日志](#)时很有用，可以获得更多（或更少）的即时刷新。

- `--drain-time-s <integer>`

(可选) 热重启期间 Envoy 将耗尽连接的时间（秒）。请参阅[热重启概述](#)了解更多信息。默认为 600 秒（10 分钟）。通常耗尽时间应小于通过 `--parent-shutdown-time-s` 选项设置的父进程关闭时间。如何配置这两个设置取决于具体的部署。在边缘的场景下，可能需要耗费很长时间。在服务到服务的场景下，耗尽和关闭的时间可能缩短很多（例如，60s/90s）。

- `--parent-shutdown-time-s <integer>`

(可选) Envoy 在热重启时关闭父进程之前等待的时间（秒）。请参阅[热重启概述](#)了解更多信息。默认为 900 秒（15 分钟）。

- `--max-obj-name-len <uint64_t>`

(可选) `cluster/route_config/listener` 中名称字段的最大长度（以字节为单位）。此选项通常用于自动生成集群名称的场景，通常超过会 60 字符的内部限制。默认为 60。

注意：此设置会影响 `--hot-restart-version` 的输出。若您开始使用此选项并设置为非默认值，则应该使用相同的值配置到热重启的新进程。

- `--max-stats <uint64_t>`

(可选) 热重启间可以共享统计的最大数量。此设置会影响 `--hot-restart-version` 的输出；热重启必须使用相同的值。默认为 16384。

- `--disable-hot-restart`

(可选) 此标识禁用已启用热重启的 Envoy 版本的热重启。默认情况下，热重启是启用的。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:30:03

热重启 Python 包装器

通常情况下，Envoy 将会以[热重启](#)的方式进行配置变更和二进制更新。但是，在很多情况下，用户会希望使用标准进程管理器，例如 monit、runit 等。我们提供 [/restarting/hot-restart.py](#) 来使这个过程简单明了。

调用重启程序方式如下：

```
hot-restart.py start_ envoy.sh
```

start_ envoy.sh 可能像这样定义（使用 salt/jinja 类似的语法）：

```
#!/bin/bash

ulimit -n {{ pillar.get('envoy_max_open_files', '102400') }}
exec /usr/sbin/envoy -c /etc/envoy/envoy.cfg --restart-epoch $RESTART_EPOCH --service-cluster {{ grains['cluster_name'] }}
--service-node {{ grains['service_node'] }} --service-zone {{ grains.get('ec2_availability-zone', 'unknown') }}
```

在每次重启时，*RESTART_EPOCH* 环境变量是由重启程序设置，并且可以传递给 `--restart-epoch` 选项

重启程序可以处理以下信号：

- **SIGTERM**: 将干净地终止所有子进程并退出。
- **SIGHUP**: 将重新调用作为第一个参数传递给热重启程序的脚本，来进行热重启。
- **SIGCHLD**: 如果任何子进程意外关闭，那么重启脚本将关闭所有内容并退出以避免处于意外状态。随后，控制进程管理器应该重新启动重启脚本以再次启动Envoy。
- **SIGUSR1**: 将作为重新打开所有访问日志的信号，转发给Envoy。可用于原子移动以及重新打开日志轮转。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-16 22:42:07

管理接口

Envoy 在本地提供了一个管理界面，可以使用这一界面查询或修改服务器的各种数据。

- [v1 API 参考](#)
- [v2 API 参考](#)

注意

目前管理界面可以进行破坏性操作（例如关闭服务器），也可能暴露私有数据（例如统计数据、集群名称、证书信息等）。将管理界面限制到只能在安全网络之内进行访问是 **非常必要** 的。同时还要注意，提供管理界面服务的网络接口只接入到安全网络之中（防止 CSRF 攻击等目的）。要实现这些目标，可以进行相应的防火墙设置，或者只允许 `localhost` 访问。可以使用如下的 v2 配置来完成：

```
admin:  
  access_log_path: /tmp/admin_access.log  
  address:  
    socket_address: { address: 127.0.0.1, port_value: 9901 }
```

未来还会在管理界面中加入更多的安全相关的选项。这部分工作的进度在 [Github Issue #2763](#) 上进行跟踪。所有的变更操作都应该通过 HTTP POST 方式进行。一段时间之内还是允许 HTTP GET 访问的，但是会有一条 Warning 日志。

GET /

渲染一个 HTML 主页，其中包含指向所有可用选项的链接。

GET /help

以字符表格的形式输出所有可用选项。

GET /certs

列出所有载入的 TLS 认证，包括文件名、序列号以及过期时间。

GET /clusters

列出[集群管理器](#)中配置的所有集群。这种信息中包含了已被发现的每个集群中的所有上游主机，以及每个主机的统计信息。如果服务发现出现问题需要除错，这些信息就很有帮助了。

集群管理器信息

```
version_info : string, 最近载入的 CDS 的版本信息字符串。如果 Envoy 没有设置 CDS，则会输出来自于  
version_info::static。
```

集群层信息

- 各优先级的[断路器](#)设置。
- 如果设置了[外部检测](#)，则显示相关信息。目前包含了[平均成功率](#)以及[驱逐阈值](#)的检测器。上一个检测周期中所获取

的数据量不足，这些变量的值会被设置为 `-1`。

- `added_via_api` 标志：如果是静态配置添加的集群，这个项目的值就是 `false`；如果是使用 CDS API 添加的集群，这个值就会设置为 `true`。

主机统计数据

名称	类型	描述
<code>cx_total</code>	Counter	连接总数
<code>cx_active</code>	Gauge	活动连接总数
<code>cx_connect_fail</code>	Counter	连接失败总数
<code>rq_total</code>	Counter	请求总数
<code>rq_timeout</code>	Counter	超时请求总数
<code>rq_success</code>	Counter	收到非 5xx 响应的请求总数
<code>rq_error</code>	Counter	收到 5xx 响应的请求总数
<code>rq_active</code>	Gauge	活动请求总数
<code>healthy</code>	String	主机健康状态，下面会有讲解
<code>weight</code>	Integer	负载均衡权重（0-100）
<code>zone</code>	String	服务区域
<code>canary</code>	Boolean	本主机是否为金丝雀
<code>success_rate</code>	Double	请求成功率（0 - 100）。如果统计周期内没有足够的请求量进行运算，则返回 <code>-1</code>

主机健康状态

- 如果主机是健康的，那么 `healthy` 的输出为 `healthy`。
- 如果主机不健康，则 `healthy` 返回的是下面几个状态之一：
 - `/failed_active_hc`：主动健康监测失败。
 - `/failed_eds_health`：EDS 标记该主机不健康。
 - `/failed_outlier_check`：外部检测失败。

GET /config_dump

用 JSON 序列化格式从多种 Envoy 组件中导出当前的配置。可以延伸阅读 [response definition](#) 的内容，来获得更详细的信息。

POST /cpuprofiler

启用或停用 CPU Profiler。编译时需要启用 gperftools。

POST /healthcheck/fail

设置健康状况为失败。需要配合 HTTP [健康检查过滤器](#)来使用。这一功能可以停用一个服务器而无需进行关闭或者重启操作。这个命令执行之后，不论过滤器如何设置，都会把全局范围内的健康检查设为失败。

POST /healthcheck/ok

`POST /healthcheck/fail` 的逆向操作。同样需要配合 HTTP 健康检查过滤器来使用。

GET /hot_restart_version

参考 `--hot-restart-version`。

POST /logging

在不同的子组件上启用或者禁用不同级别的日志。一般只会在开发期间使用。

POST /quitquitquit

完全退出服务。

POST /reset_counters

把所有的计数器复位为 0。这在使用 `GET /stats` 协助调试的时候是很有用的功能。注意这个功能并不会删除任何发送给 statsd 的数据，它只会对 `GET /stats` 命令的输出造成影响。

GET /server_info

输出关于服务器的信息，内容格式类似：

```
envoy 267724/RELEASE live 1571 1571 0
```

其中的字段包括：

- 进程名称
- 编译的 SHA 以及 Build 类型
- 当前热启动周期的启动时间
- 总的启动时间（包括所有的热启动周期）
- 当前的热启动周期

GET /stats

按需输出所有的统计内容。这个命令对于本地调试非常有用。Histograms 能够计算分位数并进行输出，即 P0, P25, P50, P75, P90, P99, P99.9 和 P100。每个分位数都是一种（区间值，累计值）的形式，区间值代表的是上次刷新以后的数值，累计值代表的是该实例启动以后的总计值。[统计概览](#) 章节中提供了更多方面的内容。

GET /stats?format=json

使用 JSON 格式输出 `/stats`，编程访问统计信息时可以使用这一格式。Counter 和 Gauge 会以（键，值）的形式出现。Histograms 会放在 "histograms" 节点之下，其中包含了 "supported_quantiles" 节点，其中列出了支持的分位数，以及这些分位数的计算结果。只有包含值的 Histograms 才会输出。

如果一个 Histogram 在这一区间没有更新，那么这一区间的所有分位数输出都是空的。

下面是一个输出样例：

```
{
  "histograms": {
    "supported_quantiles": [
      0, 25, 50, 75, 90, 95, 99, 99.9, 100
    ],
    "computed_quantiles": [
      {
        "name": "cluster.external_auth_cluster.upstream_cx_length_ms",
        "values": [
          {"interval": 0, "cumulative": 0},
          {"interval": 0, "cumulative": 0},
          {"interval": 1.0435787, "cumulative": 1.0435787},
          {"interval": 1.0941565, "cumulative": 1.0941565},
          {"interval": 2.0860023, "cumulative": 2.0860023},
          {"interval": 3.0665233, "cumulative": 3.0665233},
          {"interval": 6.046609, "cumulative": 6.046609},
          {"interval": 229.57333, "cumulative": 229.57333},
          {"interval": 260, "cumulative": 260}
        ]
      },
      {
        "name": "http.admin.downstream_rq_time",
        "values": [
          {"interval": null, "cumulative": 0},
          {"interval": null, "cumulative": 0},
          {"interval": null, "cumulative": 1.0435787},
          {"interval": null, "cumulative": 1.0941565},
          {"interval": null, "cumulative": 2.0860023},
          {"interval": null, "cumulative": 3.0665233},
          {"interval": null, "cumulative": 6.046609},
          {"interval": null, "cumulative": 229.57333},
          {"interval": null, "cumulative": 260}
        ]
      }
    ]
  }
}
```

GET /stats?format=prometheus

或者换个方式 `GET /stats/prometheus`，使用 `Prometheus` v0.0.4 格式输出统计数据。这样就可以和 `Prometheus` 进行集成了。当前只有 Counter 和 Gauge 会进行输出。未来的更新中会输出 Histogram。

GET /runtime

使用 JSON 格式按需输出所有运行时数值。[运行时配置](#)一节中，更详细的讲述了这些值的配置和使用。输出内容包括活动的重载后的运行时数值，以及每个键的堆栈。空字符串代表没有值，来自堆栈的最终有效值会用单独的键来做出标识，例如下面的输出：

```
{
  "layers": [
    "disk",
    "override",
    "admin",
  ],
  "entries": {
    "my_key": {
      "layer_values": [
        "my_disk_value",
        "",
        ""
      ],
      "final_value": "my_disk_value"
    }
  }
}
```

```
},
"my_second_key": {
  "layer_values": [
    "my_second_disk_value",
    "my_disk_override_value",
    "my_admin_override_value"
  ],
  "final_value": "my_admin_override_value"
}
}
```

POST /runtime_modify?key1=value1&key2=value2&keyN=valueN

通过提交的参数对运行时数值进行添加或修改。要删除一个之前加入的键，只需要使用一个空值即可。注意这种删除操作，只适用于这一端点中使用重载方式加入的值；从磁盘中载入的值是能通过重载进行修改，无法删除。

注意

使用 /runtime_modify 端点要当心，这一变更是即时生效的。保障管理界面的[安全性](#)至关重要。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-18 10:56:23

统计概览

Envoy 基于服务器的配置输出数字统计信息。统计信息在本地可以通过 [GET /stats](#) 命令查看，通常情况下，统计信息会发送到 [statsd cluster](#)。输出的统计信息记录在[配置指南](#)的相关部分中。一些更重要的统计信息总是会被使用到，这些信息可以查阅以下章节：

- [HTTP 链接管理器](#)
- [Upstream 集群](#)

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-17 13:36:54

运行时

[运行时配置](#) 可被用于修改各项服务器设置而不必重启 Envoy。可用的运行时设置取决于服务器是如何配置的。它们在[配置指导](#)的相关章节中说明。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-19 16:21:24

文件系统标志

- Envoy 支持文件系统 "标志" 在启动之后改变状态。在需要的情况下，该功能用于保持重启之间的变化。标志文件应该被放置在 `flags_path` 配置选项指定的目录中。当前支持的标志文件是：

- o drain

如果这个文件存在，Envoy 将以 HC 失败模式启动，类似于 `POST /healthcheck/fail` 命令被执行之后。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-20 22:28:10

流量捕获

Envoy 当前提供了一个实验性的[传输套接字扩展](#), 用于拦截流量并写入一个[protobuf 文件](#)中。

警告

这个功能是实验性的, 并存在一个已知的问题, 当在给定的 socket 上出现很长的跟踪调用的时候会 OOM。如果担心存在安全问题, 也可以在构建时禁用它, 请参阅
<https://github.com/envoyproxy/envoy/blob/master/bazel/README.md#disabling-extensions>。

配置

捕获行为可以被配置在 [Listener](#) 和 [Cluster](#) 上, 提供了一种能力, 可以分别针对上行流量和下行流量进行拦截。

要配置流量捕获, 添加一个 `envoy.transport_sockets.capture` 配置到 listener 或 cluster 上. 如:

```
transport_socket:
  name: envoy.transport_sockets.capture
  config:
    file_sink:
      path_prefix: /some/capture/path
    transport_socket:
      name: raw_buffer
```

若支持 TLS, 如:

```
transport_socket:
  name: envoy.transport_sockets.capture
  config:
    file_sink:
      path_prefix: /some/capture/path
    transport_socket:
      name: ssl
      config: <TLS context>
```

这里的 TLS 配置会分别替换现有在 listener 或 cluster 上[下行流量](#)或[上行流量](#)的 TLS 配置.

任一的 socket 实例都会生成一个包含 path 前缀的跟踪文件. 如: /some/capture/path_0.pb

PCAP 传播

生成的跟踪文件可以被转成 [libpcap format](#), 可以使用如 [Wireshark](#) 和 [capture2pcap](#) 这样的工具进行分析, 如:

```
bazel run @envoy_api//tools:capture2pcap /some/capture/path_0.pb path_0.pcap
tshark -r path_0.pcap -d "tcp.port==10000,http2" -P
1  0.000000  127.0.0.1 → 127.0.0.1    HTTP2 157 Magic, SETTINGS, WINDOW_UPDATE, HEADERS
2  0.013713  127.0.0.1 → 127.0.0.1    HTTP2 91 SETTINGS, SETTINGS, WINDOW_UPDATE
3  0.013820  127.0.0.1 → 127.0.0.1    HTTP2 63 SETTINGS
4  0.128649  127.0.0.1 → 127.0.0.1    HTTP2 5586 HEADERS
5  0.130006  127.0.0.1 → 127.0.0.1    HTTP2 7573 DATA
6  0.131044  127.0.0.1 → 127.0.0.1    HTTP2 3152 DATA, DATA
```


为自定义用例扩展 Envoy

得益于 Envoy 的架构，通过[网络过滤器](#)和[HTTP 过滤器](#)进行扩展都相当方便。

有关如何添加网络过滤器，以及组织代码仓库和构建依赖关系的示例，请参见 [envoy-filter-example](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-23 21:26:09

Envoy 有多快?

我们经常被问到 到底 Envoy 有多快? 或是 Envoy 将在我的请求上加多少的延时? 答案是: 看情况。

性能极度依赖于哪些 Envoy 特性被使用以及 Envoy 运行在如何的环境上。除此之外, 运作准确完备的性能测试是一个非常有难度的任务, 而我们的项目组目前并没有相应的资源去支撑这个任务。

其实我们在关键路径上对 Envoy 已经做了非常多的性能调优工作, 同时我们相信这些调优工作都卓有成效, 但因为上诉观点我们并没有对此发布一个官方的基准测试报告。我们会希望用户在他们自己的环境对 Envoy 使用类同于他们计划在生产环境的配置, 然后做相应的基准测试。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:14:27

从哪里能获得二进制文件？

请参考 [这里](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-15 18:29:27

如何设置 SNI?

SNI 仅被 v2 配置/API 支持。

目前的实现中要求所有过滤器链中的过滤器 必须是相同的。在以后的发布中,这个约束将会放宽, 我们将可以将SNI运用到完全不同的过滤器链中。 我们还可以将 域名匹配 运用到HTTP 连接管理中去选择不同的路由路线。

这是截至目前最常见的 SNI 使用场景。

以下是一个满足上述条件的 YAML 范例。

```
address:
  socket_address: { address: 127.0.0.1, port_value: 1234 }
filter_chains:
- filter_chain_match:
  sni_domains: "example.com"
  tls_context:
    common_tls_context:
      tls_certificates:
        - certificate_chain: { filename: "example_com_cert.pem" }
          private_key: { filename: "example_com_key.pem" }
filters:
- name: envoy.http_connection_manager
  config:
    route_config:
      virtual_hosts:
        - routes:
          - match: { prefix: "/" }
            route: { cluster: service_foo }
- filter_chain_match:
  sni_domains: "www.example.com"
  tls_context:
    common_tls_context:
      tls_certificates:
        - certificate_chain: { filename: "www_example_com_cert.pem" }
          private_key: { filename: "www_example_com_key.pem" }
filters:
- name: envoy.http_connection_manager
  config:
    route_config:
      virtual_hosts:
        - routes:
          - match: { prefix: "/" }
            route: { cluster: service_foo }
```

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:15:24

如何设置 zone 可感知路由?

在源服务 (“cluster_a”) 和目标服务 (“cluster_b”) 之间启用[区域感知路由](#)需要执行几个步骤。

源服务上的 Envoy 配置

本节介绍与源服务一起运行的 Envoy 的具体配置。要求如下:

- Envoy 必须使用 `--service-zone` 选项启动，该选项为当前主机定义区域。
- 源和目的地集群的定义都必须具有 `sds` 类型。
- 必须将 `local_cluster_name` 设置为源集群。

以下配置中仅列出了集群管理器的主要部分。

```
{  
  "sds": "{...}",  
  "local_cluster_name": "cluster_a",  
  "clusters": [  
    {  
      "name": "cluster_a",  
      "type": "sds",  
    },  
    {  
      "name": "cluster_b",  
      "type": "sds"  
    }  
  ]  
}
```

目的地服务上的 Envoy 配置

没有必要与目的地服务并排运行 Envoy，但重要的是目的地集群中的每台主机都注册[源服务 Envoy 查询](#)的发现服务。[区域](#)信息必须作为该响应的一部分提供。

下面的应答中只列出了与区域相关的数据。

```
{  
  "tags": {  
    "az": "us-east-1d"  
  }  
}
```

基础设施搭建

上述配置对于区域感知路由是必需的，但是在[不执行](#)区域感知路由时存在某些情况。

验证步骤

- 使用[每区域](#) Envoy 统计信息来监控跨区域流量。

如何设置 zone 可感知路由?

如何设置 Zipkin 追踪?

可在 [zipkin sandbox](#) 配置中查看 zipkin 追踪配置范例。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-20 22:28:10

我设置了健康检查，但是当有节点失败时，Envoy 又路由到那些节点，这是怎么回事？

这个功能因为负载均衡中[恐慌阈值](#)为人所知。在上游站点发生大量的健康检查失败的时候，它被用来防止级联故障。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:15:48

为什么 Round Robin 负载均衡看起来不起作用?

Envoy 使用隔离方式的[线程模型](#)。这意味着工作线程以及相应的负载均衡器彼此不能互相协助。当使用例如 `round robin` 的负载均衡策略时，该策略也许不能非常好地操作多个线程。此时我们可以使用 `--concurrency` 选项来调整需要运行的工作线程数。

隔离方式的运行模型也引致了如下情形，我们时常需要为每一个上游建立多个 HTTP/2 连接；且工作线程间无法共享[连接池](#)。

Copyright © ServiceMesher 2018 all right reserved, powered by Gitbook 最后更新于 2018-05-24 15:16:04