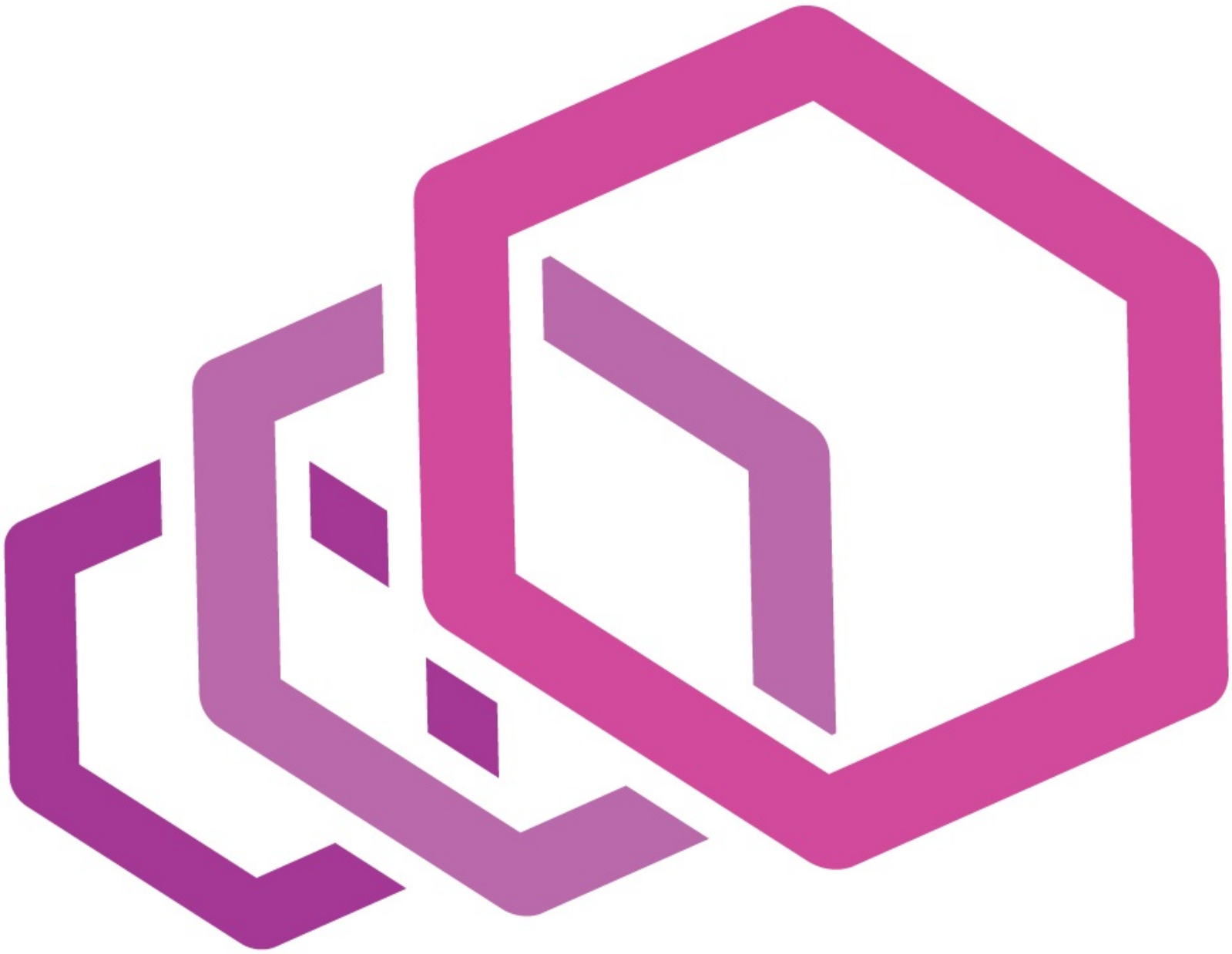


envoy V1.5.0

中文参考文档

目　录

首页

ENVOY智能代理中文参考文档 v1.5.0

注意：水平有限，仅供参考，如果有任何疑问或者建议，请提交issue讨论。

联系方式：linimbus@126.com

关于本文档

- 简介：介绍Envoy相关概念，以及总体的架构设计和常用的部署方式等
- 安装：如何通过Docker构建、安装Envoy
- 配置：既有的V1/V2 API的详细配置说明，以及相关的统计、运行时的APIs
- 操作：关于Envoy操作方式，如命令行、热更新、管理接口、统计概况等
- 扩展：关于如何为Envoy编写自定义过滤器
- V1 API 参考：老的V1 API的详细配置说明
- V2 API 参考：新的V2 API的详细配置说明

相关连接

- 官方手册
- 官方内源项目

正文目录

参考配置

- V1 API 概述
- V2 API 概述
- 监听器
- 网络过滤器
- HTTP连接管理器
- HTTP过滤器
- 集群管理
- 访问日志
- 限速服务
- 运行时配置
- 路由表检查工具

运维&管理

- 命令行选项
- 热重启
- 管理接口
- 统计概述
- 运行时配置
- 文件系统

自定义扩展示例

V1 API参考

- 监听器
- 网络过滤器
- HTTP路由配置
- HTTP过滤器
- 集群管理
- 访问日志
- 管理接口
- 限速服务
- 运行时配置
- 跟踪

V2 API参考

- 启动引导
- 监听&监听发现
- 集群&集群发现
- 服务发现
- 健康检查

- HTTP路由管理&发现
- TLS配置
- 通用的类型
- 网络地址
- 协议选项
- 发现API
- 限速组件
- 过滤器

FAQ

简介

简介

- 部署
  - 仅服务之间
  - 带前端代理的服务间
  - 服务间, 前端代理, 双向代理
- 对比相似系统
  - nginx
  - haproxy
  - AWS ELB
  - SmartStack
  - Finagle
  - proxygen&wangle
  - gRPC
  - linkerd
  - nghttp2
- 获得帮助
  - 安全漏洞报告
- 历史版本
  - 1.4.0
  - 1.3.0
  - 1.2.0
  - 1.1.0
  - 1.0.0

返回

- 首页目录

Envoy是什么

Envoy是什么？

Envoy 是一个面向服务架构的L7代理和通信总线而设计的，这个项目诞生是出于以下目标：

对于应用程序而言，网络应该是透明的，当发生网络和应用程序故障时，能够很容易定位出问题的根源。

实际上，实现前面提到的目标是非常困难的，Envoy 试图通过提供以下高级特性：

外置进程架构：Envoy是一个独立的进程，与应用程序一起运行。所有的Envoy形成一个对应用透明的通信网格，每个应用程序通过本地发送和接受消息，并不感知网络拓扑结构。这个外置进程的架构相比传统的基于library库服务通信，有两个优势；

- Envoy可以与任何语言开发的应用一起工作。Java, C++, Go, PHP, Python等都可以基于Envoy部署成一个服务网格，在面向服务的架构体系中，使用多语言开发应用越来越普遍，Envoy填补了这一空白。
- 任何参与面向服务的大型架构里工作的人都知道，部署升级library是非常令人痛苦的，而现在可以在整个基础设施上快速升级Envoy。

基于新C++11编码：Envoy是基于C++11编写的，之所以这样选择，是因为我们认为，Envoy这样的体系结构组件能够快速有效的开发出来。而现代应用程序开发者已经很难在云环境部署和使用它；通常会选择性能不高，但是能够快速提升开发效率的PHP、Python、Ruby、Scala等语言，并且能够解决复杂的外部环境依赖。并不像本地开发代码那样使用如C、C++能够提供高效的性能。

L3/L4过滤器：Envoy其核心是一个L3/L4网络代理，能够作为一个可编程过滤器实现不同TCP代理任务，插入到主服务当中。通过编写过滤器来支持各种任务，如原始TCP代理、HTTP代理、TLS客户端证书身份验证等。

HTTP L7过滤器：在现代应用架构中，HTTP是非常关键的部件，Envoy支持一个额外的HTTP L7过滤层。HTTP过滤器作为一个插件，插入到HTTP链接管理子系统中，从而执行不同的任务，如缓冲，速率限制，路由/转发，嗅探Amazon的DynamoDB等等。

支持HTTP/2：在HTTP模式下，Envoy支持HTTP/1.1、HTTP/2，并且支持HTTP/1.1、HTTP/2双向代理。这意味着HTTP/1.1和HTTP/2，在客户机和目标服务器的任何组合都可以桥接。建议在服务间的配置使用时，Envoy之间采用HTTP/2来创建持久网络连接，这样请求和响应可以被多路复用。Envoy并不支持被淘汰SPDY协议。

HTTP L7路由：在HTTP模式下运行时，Envoy支持根据content type、runtime values等，基于path的路由和重定向。当服务构建到服务网格时，Envoy为前端/边缘代理，这个功能是非常有用的。

支持gRPC：gRPC是一个来自谷歌的RPC框架，使用HTTP/2作为底层的多路传输。HTTP/2承载的gRPC请求和应答，都可以使用Envoy的路由和LB能力。所以两个系统非常互补。

支持MongoDB L7：现代Web应用程序中，MongoDB是一个非常流行的数据库应用，Envoy支持获取统计和连接记录等信息。

支持DynamoDB L7：DynamoDB是亚马逊托管的NoSQL Key/Value存储。Envoy支持获取统计和连接等信息。

服务发现：服务发现是面向服务体系架构的重要组成部分。Envoy支持多种服务发现方法，包括异步DNS解析和通过REST请求服务发现服务。

健康检查：构建Envoy网格的推荐方法，是将服务发现视为一个最终一致的方法。Envoy含有一个健康检查子系统，它可以对上游服务集群进行主动的健康检查。然后，Envoy联合服务发现、健康检查信息来判定健康的LB对象。Envoy作为一个外置健康检查子系统，也支持被动健康检查。

高级LB：在分布式系统中，不同组件之间LB也是一个复杂的问题。Envoy是一个独立的代理进程，不是一个lib库，所以他能够在一个地方实现高级LB，并且能够被任何应用程序访问。目前，包括自动重试、断路器，全局限速，阻隔请求，异常检测。将来还会支持按计划进行请求速率控制。

前端代理：虽然Envoy作为服务间的通信系统而设计，但是（调试，管理、服务发现、LB算法等）同样可以适用于前端，Envoy提供足够的特性，能够作为绝大多数Web应用的前端代理，包括TLS、HTTP/1.1、HTTP/2，以及HTTP L7路由。

极好的可观察性：如上所述，Envoy目标是使得网络更加透明。而然，无论是网络层还是应用层，都可能会出现问题。Envoy包括对所有子系统，提供了可靠的统计能力。目前只支持statsd以及兼容的统计库，虽然支持另一种并不复杂。还可以通过管理端口查看统计信息，Envoy还支持第三方的分布式跟踪机制。

动态配置：Envoy提供分层的动态配置API，用户可以使用这些API构建复杂的集中管理部署。

设计目标

简要说明：Envoy并不是很慢（我们已经花了相当长的时间来优化关键路径）。基于模块化编码，易于测试，而不是性能最优。我们的观点是，在其他语言或者运行效率低很多的系统中，部署和使用Envoy能够带来很好的运行效率。

返回

- 简介
- 首页目录

架构介绍

架构概述

- 术语
- 线程模型
- 监听器
- L3/L4网络过滤器
- HTTP连接管理
  - HTTP协议
  - 头域审查
  - 路由表配置
- HTTP过滤器
- HTTP路由
  - 路由表
  - 重试
  - 优先级
- gRPC
- WebSocket支持
  - 连接语义
- 集群管理
- 服务发现
  - 支持的服务发现类型
  - 最终一致的服务发现
- 健康检查
  - 被动健康检查
  - 连接池交互
  - HTTP健康检查过滤器
  - 主动健康检查&快速失败
  - 健康检查身份
- 连接池
  - HTTP/1.1
  - HTTP/2
  - 健康检查交互

- 负载均衡
  - 支持负载均衡
  - 阈值
  - 区域感知路由
  - 负载平衡器子集
- 异常检测
  - 逐出算法
  - 检测类型
  - 逐出事件记录
  - 配置参考
- 熔断
- 全局限速
- TLS
  - 底层的实现
  - 认证过滤器
- 统计
- 运行时配置
- 跟踪
  - 概述
  - 如何初始化跟踪
  - 跟踪上下文传递
  - 每个跟踪包含哪些数据？
- TCP代理
- 访问日志
- MongoDB
- DynamoDB
- Redis
  - 配置
  - 支持的命令
  - 失败模型
- 热重启
- 动态配置
  - 全静态
  - 仅SDS/EDS
  - SDS/EDS, CDS
  - SDS/EDS, CDS, RDS

    - SDS/EDS, CDS, RDS, LDS

- 初始化
- 逐出
- 脚本

返回

- 简介
- 首页目录

术语

术语

在我们深入到主要的体系结构文档之前，需要明确一些定义。其中一些定义在行业中有些争议，在整个Envoy文档和代码库中如何使用它们的，下面就会展开。

主机（Host）：能够进行网络通信的实体（如手机，服务器等上的应用程序）。在这个文档中，主机是一个逻辑网络应用程序。一个物理硬件可能有多个主机上运行，只要他们可以独立寻址。

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

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

监听器（Listener）：侦听器是可以被下游客户端连接的命名网络位置（例如，端口，unix域套接字等）。Envoy公开一个或多个下游主机连接的侦听器。

群集（Cluster）：：群集是指Envoy连接到的一组逻辑上相似的上游主机。Envoy通过服务发现发现一个集群的成员。它可以通过主动健康检查来确定集群成员的健康度，从而Envoy通过负载均衡策略将请求路由到相应的集群成员。

网格（Mesh）：协调一致以提供一致的网络拓扑的一组主机。在本文档中，“Envoy mesh”是一组Envoy代理，它们构成了由多种不同服务和应用程序平台组成的分布式系统的消息传递基础。

运行时配置（Runtime configuration）：与Envoy一起部署的外置实时配置系统。可以更改配置设置，可以无需重启Envoy或更改主要配置。

返回

- 架构介绍
- 简介
- 首页目录

线程模型

线程模型

Envoy使用单个进程多线程体系架构。一个主线程控制各个零散的协作任务，如一些工作线程执行监听、过滤和转发任务。一旦某个连接被一个监听器接受，这个连接将会一直运行在一个工作线程上。这使得大多数Envoy在很大程度上是单线程的（令人尴尬的并行），而在工作线程之间有少量复杂的逻辑处理。通常Envoy是100%非阻塞模式，对于大多数工作负载，我们建议将工作线程的数量配置等同于机器上硬线程的数量。

返回

- 架构介绍
- 简介
- 首页目录

监听器

监听器

Envoy配置支持单个进程中的任意数量的监听器。通常我们建议每台机器运行一个Envoy，而不关心监听器数量。这样可以使操作更简单，统计也更简单。目前Envoy只支持监听TCP。

每个监听器都独立配置一定数量的（L3 / L4）网络过滤器。当监听器接收到新连接时，实例化相应过滤器，并开始处理后续事务。通用监听器用于执行不同代理任务（例如，速率限制，TLS客户机认证，HTTP连接管理，MongoDB嗅探，原始TCP代理等）。

监听器也可以通过监听器发现服务（LDS）动态获取。

相关配置

返回

- 架构介绍
- 简介
- 首页目录

L3/L4网络过滤器

HTTP连接管理

HTTP连接管理

HTTP是现代服务架构的关键组件，Envoy实现了大量HTTP定制的功能。Envoy有一个内置的网络级过滤器，称为HTTP连接管理器。该过滤器将原始字节转换为HTTP级别消息和事件（例如，接收到的头域，接收到的body数据，接收的尾部等）。它还会处理所有HTTP连接和访问记录，请求ID生成和跟踪，请求/响应头域处理，路由表管理和统计等请求。

HTTP连接管理器配置

HTTP协议

Envoy的HTTP连接管理器支持HTTP/1.1，WebSockets，和HTTP/2，但它不支持SPDY。Envoy的HTTP默认被设计成一个HTTP/2多路复用代理。在其内部用HTTP/2术语用于描述系统组件。例如，HTTP请求和响应发生在流上。编解码器API用于从不同的协议转换，用于stream、请求、响应等的不可知的形式。在HTTP/1.1的情况下，编解码器将协议的serial/pipeline功能转换为类似于HTTP/2到更高层的东西。这意味着大多数代码不需要理解stream是否是HTTP/1.1还是HTTP/2。

HTTP头域清理

出于安全原因，HTTP连接管理器将会清理各种头域敏感数据。

路由表配置

每个HTTP连接管理器过滤器都有一个关联的路由表。路由表可以通过以下两种方式之一来指定：

- 静态配置
- 通过RDS API动态配置

返回

- 架构介绍
- 简介
- 首页目录

HTTP过滤器

HTTP过滤器

就像网络级别的过滤器一样，Envoy在连接管理器中支持HTTP级别的过滤器。过滤器支持可编写，无需关心底层链路协议（HTTP/1.1，HTTP/2等）或是否使能多路复用的情况下，可直接对HTTP层消息进行操作。有三种类型的HTTP级别过滤器：

- 解码器（Decoder）：解码器过滤器在连接管理器正在解码请求流的部分（头域，正文和尾部）时被调用。
- 编码器（Encoder）：编码器过滤器在连接管理器即将编码部分响应流（头域，正文和尾部）时被调用。
- 解码器/编码器（Decoder/Encoder）：解码器/编码器过滤器在连接管理器正在解码请求流的部分时以及连接管理器将要对部分响应流进行编码时被调用。

HTTP级别过滤器API允许在不知道底层协议的情况下运行。 像网络级别的过滤器一样，HTTP过滤器可以停止并继续迭代到后续的过滤器。这可以实现更复杂的场景，例如运行状况检查处理，调用速率限制服务，缓冲，路由，为应用程序流量（例如DynamoDB等）生成统计信息。Envoy已经包含了几个HTTP级别的过滤器，配置参考。

返回

- 架构介绍
- 简介
- 首页目录

HTTP路由

HTTP路由

Envoy包含一个HTTP路由过滤器，可以用它来执行高级路由任务。这对于处理前端流量（传统方式用反向代理来处理请求），或者构建服务间的Envoy网格是很有用的（通常根据HTTP头部上字段可以路由以到达指定的上游服务集群）。 Envoy也支持配置为正向代理。在转发代理配置中，网络客户端可以通过将他们的http代理适当地配置为Envoy。在高层次上，路由器接收一个传入的HTTP请求，将其与上游集群进行匹配，获取到上游集群中主机的连接池，并转发该请求。路由器过滤器支持以下功能：

- 将domains/authorities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（区分大小写和不区分大小写）。当前不支持正则表达式/slug匹配，主要是因为它无法精确辨别路由规则是否相互冲突。由于这个原因，我们不建议在反向代理级别使用正则表达式/slug路由，但是我们可能会根据需求添加支持
- 根据TLS重定向到虚拟主机
- 根据path/host重定向的路由规则
- 显式host改写
- 根据上游主机的DNS名称自动重写host
- Prefix重写
- Websocket在路由级别升级
- 通过HTTP头或通过路由配置请求重试
- 通过HTTP头或通过路由配置指定的请求超时
- 通过运行时间值将流量从一个上游群集转移到另一个上（请参阅流量转移/分流）
- 使用基于权重/百分比的路由（请参阅流量转移/拆分）跨多个上游群集进行流量分流
- 任意头匹配路由规则
- 虚拟集群场景，指定虚拟主机作为虚拟群集，由Envoy用于在标准群集级别之上生成附加统计信息。虚拟群集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
- 非转发代理支持绝对url

路由表

HTTP连接管理器的配置，拥有所有配置的HTTP过滤器使用的路由表。虽然路由表主要用于路由过滤器，但是如果他们想根据请求的最终目的地做出决定，其他过滤器也可以访问。例如，内置的速率限制过滤器参考路由表来确定是否应该基于路由来调用全局速率限制服务。即使决策涉及随机性（例如，在运行时配置路由规则的情况下），连接管理器也确保所有获取路由的调用对于特定请求是稳定的。

重试语义

Envoy允许在路由配置中以及通过请求头对特定请求配置重试。存在如下可能的配置：

- 最大重试次数：Envoy将继续重试任意次数。在每次重试之间使用指数退避算法。此外，所有重试都包含在整个请求超时内。由于大量的重试，这避免了很长的请求时间。
- 条件重试：Envoy可以根据应用要求在不同类型的条件下重试。例如，网络故障，所有5xx响应码，幂等4xx响应码等。

请注意：根据x-envoy重载的内容，重试可能无法使用。

优先级路由
Envoy支持路由级别的优先级路由。当前的优先级实现针对每个优先级级别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，两个物理连接也将被用于上游主机。未来，Envoy可能会在单一连接上支持真正的HTTP/2优先级。

目前支持默认和高两个优先级。

返回

- 架构介绍
- 简介
- 首页目录

gRPC

gRPC

gRPC是来自Google的RPC框架。它使用协议缓冲区作为基础的序列化/IDL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很好的支持gRPC：

- gRPC使用HTTP/2尾部来传送请求状态。 Envoy是能够正确支持HTTP/2尾部的少数几个HTTP代理之一，因此是少数可以传输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传输到目标服务器。该响应被转换回HTTP/1.1。
- 安装后，网桥过滤器除了统计全局HTTP统计数据之外，还会根据RPC统计信息进行收集。
- gRPC-Web由过滤器支持，它允许gRPC-Web客户端通过HTTP/1.1向Envoy发送请求并代理到gRPC服务器。目前正处于积极的发展阶段，预计将成为gRPC桥接过滤器的后续产品。
- gRPC-JSON代码转换器由一个过滤器支持，该过滤器允许RESTful JSON API客户端通过HTTP向Envoy发送请求并代理到gRPC服务。

返回

- 架构介绍
- 简介
- 首页目录

WebSocket支持

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头并且匹配的HTTP路由显式配置为使用WebSocket（use_websocket）时才允许连接升级。如果一个请求到达启用了WebSocket的路由而没有必要的升级头（upgrade headers），它将被视为任何常规的HTTP/1.1请求。

由于Envoy将WebSocket连接视为纯TCP连接，因此它支持WebSocket协议的所有草案，而与它们的连线格式无关。WebSocket路由不支持某些HTTP请求级别的功能，如重定向，超时，重试，速率限制和阴影（shadowing）。然而，支持前缀重写，显式和自动主机重写，流量转移和分离。

连接语义

即使通过HTTP/1.1连接进行WebSocket升级，WebSockets代理也与普通TCP代理类似，即Envoy不会解释WebSocket帧。下游客户端和/或上游服务器负责正确终止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。特使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，这将导致Envoy终止相应的下游客户端连接。

返回

- 架构介绍
- 简介
- 首页目录

集群管理

集群管理

Envoy的集群管理器管理所有配置的上游集群。就像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。

上游集群和主机从网络/HTTP过滤器堆栈中抽象出来，因为上游集群和主机可以用于任意数量的不同代理任务。集群管理器向过滤器堆栈公开API，允许过滤器获得到上游集群的L3/L4连接，或者到上游集群的抽象HTTP连接池的句柄（无论上游主机是支持HTTP/1.1还是HTTP/2被隐藏）。筛选器阶段确定是否需要L3/L4连接或新的HTTP流，并且集群管理器处理了知道哪些主机可用且健康，负载平衡，上游连接数据的线程本地存储的所有复杂性（因为大多数Envoy代码被写为单线程），上游连接类型（TCP/IP，UDS），适用的上游协议（HTTP/1.1，HTTP/2）等

群集管理器已知的群集可以静态配置，也可以通过群集发现服务（CDS）API动态获取。动态集群提取允许将更多配置存储在中央配置服务器中，因此需要更少的Envoy重新启动和配置分配。

- 集群管理器配置
- CDS配置

返回

- 架构介绍
- 简介
- 首页目录

服务发现

服务发现

在配置上游群集时，Envoy需要知道解析这些群集的成员。这被称为服务发现。

支持的服务发现类型

静态

静态是最简单的服务发现类型。通过静态配置明确每个上游主机的网络名称（IP地址/端口，unix域套接字等）。

严格（Strict）DNS

当使用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地址将被视为上游群集中的显式主机。 这意味着如果查询返回三个IP地址，Envoy将假定集群有三个主机，并且三个主机都应该负载平衡。如果主机从结果中删除，则Envoy认为它不再存在，并将从任何现有的连接池中摄取流量。请注意，Envoy绝不会在转发路径中同步解析DNS。以最终一致性为代价，永远不会担心长时间运行的DNS查询会受到阻塞。

逻辑（Logical）DNS

逻辑DNS使用类似的异步机制来解析严格DNS。但是，并不是严格考虑DNS查询的结果，而是假设它们构成整个上游集群，而逻辑DNS集群仅使用在需要启动新连接时返回的第一个IP地址。因此，单个逻辑连接池可以包含到各种不同上游主机的物理连接。连接永远不会流失。此服务发现类型适用于必须通过DNS访问的大型Web服务。这种服务通常使用循环法的DNS来返回许多不同的IP地址。通常会为每个查询返回不同的结果。如果在这种情况下使用严格的DNS，Envoy会认为集群的成员在每个解决时间间隔期间都会发生变化，这会导致连接池，连接循环等消失。相反，使用逻辑DNS，连接将保持活动状态，直到它们循环。在与大型Web服务交互时，这是所有可能的世界中最好的：异步/最终一致的DNS解析，长期连接，以及转发路径中的零阻塞。

服务发现服务（SDS）

Envoy通过REST API向服务发现服务，获取集群的成员。Lyft通过Python提供了一个发现服务的参考实现。该实现是使用AWS DynamoDB作为后端存储，但该API非常简单，可以轻松地在各种不同的后备存储之上实现。对于每个SDS群集，Envoy将定期从发现服务中获取群集成员。SDS做为首选的服务发现机制，原因如下：

- Envoy能够对每个上游主机都有更加详细的信息（相比通过DNS解析），并能做出更智能的负载均衡决策。
- 在每个主机发现的API响应中，通过携带附加的属性，如负载均衡权重，灰度状态，区域等。这些附加属性在负载均衡，统计信息收集等过程中由Envoy网络全局使用。

通常，主动健康检查与最终一致的服务发现服务结合起来使用，以进行负载平衡和路由决策。这将在下一节进一步讨论。

最终一致的服务发现

许多现有的RPC系统将服务发现视为完全一致的过程。为此，他们使用支持完全一致的leader选举存储，如Zookeeper，etcd，Consul等。我们的经验是，在大规模操作这些存储会很痛苦。

Envoy从一开始就设计了服务发现不需要完全一致的想法。相反，Envoy假定主机以一种最终一致的方式来自网格。在部署服务间Envoy网格时，我们推荐使用最终一致的服务发现，结合主动健康检查（Envoy主动健康检查上游集群成员状态）来确定集群运行状况。这种范例有许多好处：

- 有的健康决定是完全分配的。 因此，网络分区被正常处理（应用程序是否正常处理分区是另一回事）。
- 为上游群集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：

服务发现状态	健康检查正常	健康检查失败
存在	路由	停止路由
不存在	路由	停止路由/删除

- 主机被发现/健康检查确定

  特使将路由到目标主机。

- 主机无法被发现/健康检查确定

  Envoy将路由到目标主机。这是非常重要的，因为设计假定发现服务可以随时失败。如果主机即使在发现数据缺失之后仍继续通过健康检查，Envoy仍将路由。 虽然在这种情况下添加新主机是不可能的，但现有的主机仍然可以正常运行。 当发现服务再次正常运行时，数据将最终重新收敛。

- 主机被发现/健康检查失败

  Envoy不会路由到目标主机。假设健康检查数据比发现数据更准确。

- 主机无法被发现/健康检查失败

  Envoy不会路由并将删除目标主机。这是Envoy将清除主机数据的唯一状态。

返回

- 架构介绍
- 简介
- 首页目录

健康检查

健康检查

主动健康检查是以每个上游服务群集进行配置。如服务发现部分所述，主动健康检查与SDS服务发现配合使用。但是，即使使用其他服务发现方式，也有相应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以及相应设置（检查时间间隔，因故障标记主机不健康，成功之后标记主机健康等）：

- HTTP：HTTP健康检查期间，Envoy将向上游主机发送HTTP请求。如果主机健康，预计会有200返回码。如果上游主机想立即通知下游主机不再向其转发流量，则返回503返回码。
- 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报文。如果主机期望被认为是健康的，则在响应中回应相应的报文。Envoy也支持只连接L3/L4健康检查。
- Redis：Envoy将发送一个Redis PING命令，并期待一个PONG响应。上游Redis服务器可以使用PONG以外的任何其他响应，来立即触发的健康检查失败。

被动健康检查

Envoy还支持通过检测异常值，来进行被动健康检查。

连接池交互

浏览此处获取更多信息。

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的流量。Envoy提供一个可配置HTTP健康检查过滤器，并安装在HTTP监听器中。这个过滤器有几种不同的操作模式：

- 不通过：在此模式下，健康检查请求永远不会传递到本地服务。Envoy将根据服务器当前的耗尽状态，以200或503响应。
- 通过：在这种模式下，Envoy会将每个健康检查请求传递给本地服务。预计该服务将返回200或503取决于其健康状况。
- 通过缓存：在这种模式下，Envoy会将健康检查请求传递给本地服务，但是会将结果缓存一段时间。随后在缓存有效时间内，进行的健康检查都会从缓存获取结果。当缓存超时后，下一个健康检查请求将被传递给本地服务。在使用较大的Envoy网格时，这是推荐的操作模式。Envoy使用持久性连接进行健康检查，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，而没有使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口(/healthcheck/fail)
- 健康检查成功管理端口(/healthcheck/ok)

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的主动健康检查流量。在这种情况下，在使用健康检查失败管理端口时，对能够快速排除上游主机，仍然很有用。为了支持这个，路由器过滤器将响应x-envoy-immediate-health-check-fail头。如果此报头由上游主机设置，则Envoy将立即将主机标记为主动健康检查失败。请注意，只有在主机集群中配置了主动健康检查，才会出现此情况。如果Envoy已通过健康检查失败管理端口标记为失败，则运行状况检查过滤器将自动设置此标头。

健康检查身份识别
只要验证上游主机对特定健康检查URL的响应，并不一定意味着上游主机是有效的。例如，在云自动扩展或容器环境中，使用最终一致的服务发现时，主机可能会消失，然后以相同的IP地址返回，但会以不同的主机类型返回。解决这个问题的一个办法是为每个服务类型设置不同的HTTP健康检查URL。这种方法的缺点是使得整体配置变得更加复杂，因为每个健康检查URL都是完全自定义的。

Envoy的HTTP健康检查支持service_name选项。如果设置了此选项，运行健康检查程序会将x-envoy-upstream-healthchecked-cluster响应头域的值与service_name进行比较。如果值不匹配，健康检查不通过。上游运行状况检查过滤器会将x-envoy-upstream-healthchecked-cluster附加到响应头域。附加值由--service-cluster命令行选项确定。

返回

- 架构介绍
- 简介
- 首页目录

连接池

连接池

对于HTTP流量，Envoy支持在基础协议（HTTP/1.1或HTTP/2）之上分层的抽象连接池。过滤器代码不需要知道底层协议，是否支持复用。实际上，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（达到断路极限）。当连接可用时，请求被绑定到连接上，或者是因为连接完成处理先前的请求，或者是因为新的连接准备好接收其第一请求。HTTP/1.1连接池不使用流水线，因此如果上游连接被切断，则只有一个下游请求必须被重置。

HTTP/2

HTTP/2连接池获取与上游主机的单个连接。所有请求都通过此连接复用。如果收到一个GOAWAY帧，或者如果连接达到最大流限制，连接池将创建一个新的连接并且耗尽现有连接，HTTP/2是首选的通信协议，因为连接很少被切断。

与健康检查交互

如果将Envoy配置为进行主动或被动健康检查，这意味着一个主机从正常状态转换为不健康状态后，将会关闭所有连接池连接。如果主机重新进入负载均衡轮训，它将创建新的连接，这将最大限度地提供机会处理坏流量（由于ECMP路由或其他）。

返回

- 架构介绍
- 简介
- 首页目录

负载均衡

负载均衡

当过滤器需要获取到上游群集中的主机连接时，群集管理器使用负载平衡策略来确定选择哪个主机。负载平衡策略是可插入的，并且在配置中以每个上游集群为单位进行指定。请注意，如果没有为群集配置积极的健康检查策略，则所有上游群集成员都被视为健康。

支持的负载平衡策略

轮训

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。

权重最小请求

请求最少的负载均衡器使用O(1)算法来选择两个随机的健康主机，并选择活跃请求较少的主机。（研究表明，这种方法几乎与O(N)全扫描一样好）。如果群集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，然后使用该主机<权重>次数的模式。这个算法对于负载测试来说简单而充分。在需要真正的加权最小请求行为的情况下（通常如果请求持续时间可变且长度较长），不应使用它。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覆盖这个用例。

哈希环

环/模哈希负载平衡器对上游主机执行一致的哈希。该算法基于将所有主机映射到一个圆上，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。一致的散列负载均衡器只有在使用指定要散列的值的协议路由时才有效。目前唯一实现的机制是通过HTTP路由器过滤器中的HTTP头值进行散列。最小环大小默认是在运行时指定的。最小环大小控制环中每个主机的副本数。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。环哈希负载平衡器当前不支持加权重。

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，那么随机负载均衡器通常比循环更好。随机选择可以避免在发生故障的主机之后，对集群中的主机造成不均衡。

原始目的地

这是一个特殊用途的负载平衡器，只能与原始目标群集一起使用。上游主机是基于下游连接元数据选择的，即，连接被打开到与连接被重定向到Envoy之前传入连接的目的地地址相同的地址。新的目的地由负载均衡器按需添加到集群，并且集群定期清除集群中未使用的主机。原始目标群集不能使用其他负载平衡类型。

恐慌阈值

在负载均衡期间，Envoy通常只考虑上游群集中的健康主机。但是，如果群体中的健康主机的百分比变得太低，Envoy就会忽略所有主机中的健康状况和平衡。这被称为恐慌阈值。默认的恐慌阈值是50%。这可以通过运行时配置。恐慌阈值用于避免因主机故障，造成整个集群负荷加重的情况。

优先级
在负载均衡期间，Envoy通常只考虑配置在最高优先级的主机。对于每个EDS LocalityLbEndpoints，还可以指定一个可选的优先级。目前，使得从一个优先级到另一个优先级的路由故障转移机制，变得相当简单：根据给定的优先级，直到它具有零个健康的主机，在这一点上，切换到下一个最高优先级很难失败。

区域感知路由
我们会使用以下术语：

- 始发/上游集群：Envoy将来自原始集群的请求路由到上游集群。
- 本地区域：包含始发和上游群集中的主机，同属一个区域。
- 区域感知路由：尽力将请求路由到本地区域中的上游群集主机。

在原始和上游群集中的主机属于不同区域的部署中，Envoy执行区域感知路由。在区域感知路由执行之前，有几个先决条件：

- 发起和上游集群都不处于恐慌状态。
- 区域感知路由已启用。
- 源群集与上游群集具有相同的区域数量。
- 上游集群有足够的主机。 浏览此处获取更多信息。

区域感知路由的目的是尽可能多地向本地区域所在的上游群集发送流量，同时在所有上游主机（取决于负载平衡策略），每个主机每秒大致保持相同数量的请求。

只要维持上游集群中每台主机的请求数量大致相同，Envoy就会尝试尽可能多地将流量推送到本地上游区域。决定Envoy路由到本地区域还是执行跨区域路由，取决于本地区域中始发群集和上游群集中健康主机的百分比。在原始和上游集群之间的本地区的百分比关系有两种情况：

- 源集群本地区域百分比大于上游群集中的百分比。在这种情况下，我们不能将来自原始集群的本地区域的所有请求路由到上游集群的本地区域，因为这将导致所有上游主机的请求不平衡。相反，Envoy会计算可以直接路由到上游群集的本地区域的请求的百分比。其余的请求被路由到跨区域。特定区域是根据区域的剩余容量（该区域将获得一些本地区域业务量并且可能具有Envoy可用于跨区域业务量的额外容量）来选择。
- 发起群集本地区域百分比小于上游群集中的百分比。在这种情况下，上游集群的本地区域可以获得来自原始集群本地区域的所有请求，并且还有一定的空间允许来自发起集群中其他区域的流量（如果需要）。

负载平衡器子集
Envoy可能被配置为根据附加到主机的元数据，将上游集群中的主机划分为多个子集。然后可以指定主机必须匹配的元数据，提供给负载平衡器进行路由，并且也可以选择回退到预定义的一组主机（包括任何主机）。

子集使用集群指定的负载平衡器策略。原来的目标策略可能不能与子集一起使用，因为上游主机事先不知

道。子集与区域感知路由兼容，但请注意，使用子集可能很容易违反上述的最小主机条件。

如果子集已配置且路由未指定元数据或没有与元数据匹配的子集，则子集负载均衡器将启动其后备策略。默认策略是NO_ENDPOINT，在这种情况下，请求失败，就好像群集没有主机一样。相反，ANY_ENDPOINT后备策略会在群集中的所有主机之间进行负载均衡，而不考虑主机元数据。最后，DEFAULT_SUBSET会导致回退至与Envoy元数据集匹配的主机之间进行负载均衡。

子集必须预定义为允许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Key，可以转换为零个或多个子集。从概念上讲，每个具有定义中所有Key的元数据值的主机都将被添加到特定于其Key值对的子集中。如果没有主机拥有所有的密钥，那么定义就不会产生子集。可以提供多个定义，并且如果单个主机匹配多个定义，则其可以出现在多个子集中。

在路由期间，路由的元数据匹配这些配置（用于查找特定的子集）。如果存在具有由路由指定的确切Key和Value的子集，则该子集用于负载平衡。否则，使用回退策略。因此，集群的子集配置必须包含与给定路由具有相同Key的定义，以便使能子集负载平衡。

此功能只能使用V2 API启用配置。此外，主机元数据仅支持在群集使用EDS发现类型时使用。子集负载平衡的主机元数据必须放在过滤器名称“envoy.lb”下。同样，路由元数据匹配条件使用“envoy.lb”过滤器名称。主机元数据可以是分层的（例如，顶级key的值可以是结构化值或列表），但子集负载平衡器仅比较顶级key和value。因此，当使用结构化值时，如果主机的元数据中出现相同的结构化值，那么路由只会匹配相同的结构化值。

示例
我们将使用所有值都是字符串的简单元数据。假定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

集群启用子集负载平衡，如下所示：

```
---
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
```

下表介绍了一些路由及其在集群中的应用结果。通常，根据请求特征与路由匹配一起使用，例如路径或报文头信息。

匹配准则	落点	原因
stage: canary	host3	主机的子集匹配
v: 1.2-pre, stage: dev	host4	主机的子集匹配
v: 1.0	host1, host2	回退：没有单独的“v”的子集
other: x	host1, host2	回退：没有“other”的子集
(none)	host1, host2	回退：没有要求的子集

元数据匹配标准也可以在路由的加权群集上指定。来自所选加权群集的元数据匹配条件将与路线中的条件合并并覆盖该条件：

路由匹配准则	加权集群匹配准则	最终匹配准则
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

具有元数据主机的示例
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'
```

具有元数据路由的示例
具有元数据匹配的RDS路由标准：

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

返回

- 架构介绍
- 简介
- 首页目录

异常检测

异常检测

异常值检测和逐出是动态确定上游群集中，某些主机是否正在执行不同于其他主机的过程，并将其从正常负载平衡集中移除。 性能可能会受到不同程度的影响，例如连续的故障，时间成功率，时间延迟等。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可以一起使用或独立使用，形成整体上游健康检查解决方案的基础。

逐出算法

取决于异常值检测的类型，弹出或者以行内（例如在连续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。逐出算法的工作原理如下：

1. 主机被确定为异常。
2. Envoy检查以确保逐出的主机数量低于允许的阈值（通过 `outlier_detection.max_ejection_percent` 设置指定）。如果逐出的主机数量超过阈值，主机不会被逐出。
3. 主机被逐出几毫秒。意味着主机被标记为不健康，在负载平衡期间不会使用，除非负载平衡器处于紧急情况。毫秒数等于 `outlier_detection.base_ejection_time_ms` 值乘以主机被逐出的次数。这会导致主机如果继续失败，则会被逐出更长和更长的时间。
4. 逐出的主机将在逐出时间满之后自动重新投入使用。一般而言，异常值检测与主动健康检查一起使用，用于全面的健康检查解决方案。

检测类型

Envoy支持以下异常检测类型：

连续5xx

如果上游主机返回一些连续的5xx，它将被逐出。请注意，在这种情况下，5xx意味着一个实际的5xx响应代码，或者一个会导致HTTP路由器代表上游返回的事件（复位，连接失败等）。逐出所需的连续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），它将被逐出。请注意，这包括HTTP路由代表上游返回其中一个状态码的事件（重置，连接失败等）。逐出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逐出汇总来自群集中每个主机的成功率数据。然后以给定的时间间隔，基于统计异常值检测来逐出主机。如果主机在一个时间间隔内的，请求量小于 `outlier_detection.success_rate_request_volume` 值，则不会为认为该主机成功率异常值。此外，如果一个时间间隔内请求量最小的主机，请求数小于

`outlier_detection.success_rate_minimum_hosts`值，则不会对群集执行检测。

逐出事件记录
Envoy可以选择生成异常值逐出事件日志。这在日常操作中非常有用，因为全局统计数据，不能提供有关哪些主机被逐出的信息以及原因。下面是一条JSON格式的日志记录：

```
{
  "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：

自从上一次操作（逐出或未逐出）发生以来的时间，以秒为单位。如果是第一次，之前没有动作，该值将为-1。

cluster：

被逐出主机所在的群集。

upstream_url：

被逐出的主机URL。例如，`tcp://1.2.3.4:80`。

action：

触发的动作（`eject`/`uneject`）

type：

如果`action`是`eject`，这里描述的是`eject类型`；如`5xx`、`GatewayFailure`、`SuccessRate`。

num_ejections：

```
如果`action`是`eject`，指定主机被逐出的累计次数（对于Envoy而言是本地的，并且如果主机被重新添加到集群，那么这个数值会被重置）
```

enforced：

```
如果`action`是`eject`，指定逐出是否被强制执行。`true`表示主机被强制逐出。`false`表示着事件被记录了，但是主机并没有被逐出。
```

host_success_rate：

```
如果`action`是`eject`，并且`type`是`SuccessRate`，主机在被逐出时的成功率（0~100范围）。
```

cluster_success_rate_average：

```
如果`action`是`eject`，并且`type`是`SuccessRate`，主机在被逐出时所在的集群平均成功率（0~100范围）。
```

cluster_success_rate_ejection_threshold：

```
如果`action`是`eject`，并且`type`是`SuccessRate`，指定逐出事件的成功率阈值。
```

配置参考

- 集群管理全局配置
- 每个群集配置
- 运行时设置
- 统计参考

返回

- 架构介绍
- 简介
- 首页目录

熔断

熔断

熔断是分布式系统的重要组成部分。快速失败并尽快给下游施加压力，几乎总是好的。这是Envoy网格的主要优点之一，Envoy在网络级别实现强制断路限制，而不必独立配置和编写每个应用程序。Envoy支持各种类型的完全分布（不协调）的熔断：

- 群集最大连接数：Envoy将为上游群集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群集，因为HTTP/2使用到每个主机的单个连接。
- 群集最大挂起请求数：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群集，因为HTTP/2连接池不会排队请求。HTTP/2请求立即复用。如果这个断路器溢出，集群的`upstream_rq_pending_overflow`计数器将增加。
- 群集最大请求数：在任何给定时间，群集中所有主机可以处理的最大请求数。实际上，这适用于HTTP/2群集，因为HTTP/1.1群集由最大连接断路器控制。如果这个断路器溢出，集群的`upstream_rq_pending_overflow`计数器将增加。
- 集群最大活动重试次数：在任何给定时间，集群中所有主机可以执行的最大重试次数。一般来说，我们建议积极进行断路重试，以便允许零星故障重试，但整体重试量不能爆炸并导致大规模级联故障。如果这个断路器溢出，集群的`upstream_rq_retry_overflow`计数器将递增。

每个熔断阈值可以按照每个上游集群和每个优先级进行配置和跟踪。这允许分布式系统的不同组件被独立地调整并且具有不同的熔断配置。

请注意，在HTTP请求的情况下，断路将导致`x-envoy-overloaded`报文头被路由过滤器设置。

返回

- 架构介绍
- 简介
- 首页目录

全局限速

全局限速

尽管分布式熔断在控制分布式系统中的吞吐量方面通常是非常有效的，但是有时并不是非常有效，所以需要全局速率限制。最常见的情况是，大量主机转发请求到少量主机，并且平均请求延迟较低（例如连接到数据库服务器的请求）。如果目标主机被备份，则下游主机将压倒上游集群。在这种情况下，要在每个下游主机上配置足够严格的熔断限制是非常困难的，这样系统将在典型的请求模式期间正常运行，但仍然可以防止系统因发生故障而引发的级联故障。全球限速是这种情况的一个很好的解决方案。

Envoy直接与全球gRPC限速服务集成。尽管可以使用任何实现定义的RPC/IDL协议的服务，但Lyft提供了使用Redis后端的Go编写的参考实现。Envoy的限速具有以下特点：

- 网络级别限制过滤器：Envoy将为安装过滤器的侦听器上的每个新连接调用速率限制服务。配置指定一个特定的域和描述符设置为速率限制。这对速率限制每秒传送收听者的连接的最终效果。配置参考。
- HTTP级别限制过滤器：Envoy将为安装过滤器的侦听器上的每个新请求调用速率限制服务，并且路由表指定应调用全局速率限制服务。对目标上游群集的所有请求以及从始发群集到目标群集的所有请求都可能受到速率限制。配置参考

限速服务配置

返回

- 架构介绍
- 简介
- 首页目录

TLS

TLS

在与上游集群连接时，Envoy支持在监听器中的终止以及发起TLS。对于Envoy来说，足以支持为现代Web服务执行标准的前端代理职责，并启动与具有高级TLS要求（TLS1.2，SNI等）的外部服务的连接。Envoy支持以下TLS特性：

- 密码可配置：每个TLS监听者和客户端可以指定它支持的密码。
- 客户端证书：除服务器证书验证之外，上游/客户端连接还可以提供客户端证书。
- 证书验证和固定：证书验证选项包括基本链验证，验证标题名称和哈希固定。
- ALPN：TLS监听器支持ALPN。HTTP连接管理器使用这个信息来确定客户端是HTTP/1.1还是HTTP/2。
- SNI：SNI当前支持客户端连接。监听器可能会在未来添加支持。
- 会话恢复：服务器连接支持通过TLS会话票据恢复以前的会话（请参阅RFC 5077）。可以在热启动之间和并行Envoy实例之间执行恢复（通常在前端代理配置中使用）。

基础实施

目前Envoy被写入使用BoringSSL作为TLS提供者。

认证过滤器

Envoy提供了一个网络过滤器，通过从REST VPN服务获取的主体执行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机列表进行匹配，以确定是否允许连接。可选IP白名单也可以配置。该功能可用于为Web基础架构VPN前端代理。

客户端TLS认证过滤器配置参考。

返回

- 架构介绍
- 简介
- 首页目录

统计

统计

使网络可以理解是Envoy的主要目标之一。Envoy根据配置如何获取大量的统计数据。一般来说，统计分为两类：

- 下游：下游统计涉及传入的连接/请求。它们由监听器，HTTP连接管理器，TCP代理过滤器等发出
- 上游：上游统计涉及传出连接/请求。它们由连接池，路由过滤器，TCP代理过滤器等。

单个代理场景通常涉及下游和上游统计信息。这两种类型可以用来获得特定网络跳跃的详细图谱。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谱。所发出的统计数据在操作指南中详细记录。

Envoy使用statsd作为统计输出格式，虽然插入不同的数据统计集并不困难。支持TCP和UDP statsd。在内部，计数器和计量器被分批并定期刷新以提高性能。统计分布会在收到时写入。注意：以前称为时序的统计数据变成分布图，唯一区别就是单位。

- v1 API 参考
- v2 API 参考

返回

- 架构介绍
- 简介
- 首页目录

运行时配置

运行时配置

Envoy支持“运行时”配置（也称为“功能标志”和“决策者”）。可以更改配置设置，这将影响操作，而无需重启Envoy或更改主配置。当前支持的实现使用文件系统文件。Envoy监视配置目录中的符号链接交换，并在发生这种情况时重新加载。这种类型的系统通常在大型分布式系统中部署。其他实现并不难实现。受支持的运行时配置设置记录在操作指南的相关部分。Envoy将使用默认运行时值和“空”提供程序正确运行，因此不需要运行Envoy这样的系统。

运行时配置

返回

- 架构介绍
- 简介
- 首页目录

跟踪

跟踪
概述
分布式跟踪使开发人员可以在大型面向服务的体系结构中获得调用流的可视化。在理解序列化，并行性和延迟来源方面，这是非常宝贵的。Envoy支持系统范围与跟踪相关的三个功能：

- 请求ID生成：Envoy将在需要时生成UUID并填充 `x-request-id` HTTP头。 应用程序可以转发 `x-request-id` 头以进行统一日志记录以及跟踪。
- 外部跟踪服务集成：Envoy支持可插入的外部跟踪可视化提供程序。 目前，Envoy支持LightStep，Zipkin或Zipkin兼容后端（例如Jaeger）。但是，并不支持其他调用跟踪机制。
- 客户端跟踪ID加入：用于将不可信 `x-client-trace-id` 请求ID头连接到可信的内部 `x-request-id` 。

如何启动跟踪
处理请求的HTTP连接管理器必须设置跟踪对象。有几种方法可以启动跟踪：

- 外部客户端通过 `x-client-trace-id` 头域字段请求。
- 通过 `x-envoy-force-trace` 头域字段请求内部服务。
- 通过设置随机采样进行采样。

路由过滤器还可以通过 `start_child_span` 选项为出站调用发起跟踪。

Envoy提供报告有关网格中服务之间通信的跟踪信息的功能。但是，为了能够关联调用流中各个代理生成的跟踪信息，服务必须在入站和出站请求之间传播特定的跟踪上下文。

无论使用哪个跟踪机制，该服务都应该传播 `x-request-id` ，以便使得在被调用服务日志中记录相关信息。

跟踪机制还需要支持额外的上下文记录，以便开发者更加容易理解（如：逻辑工作单元）之间的父/子关系。这可以通过在服务本身内直接使用LightStep（通过OpenTracing API）或Zipkin tracer来实现，以从入站请求中提取跟踪上下文，并将其注入到任何后续的出站请求中。这种方法还可以使服务创建额外的spans，描述在服务内部完成的工作，这在检查端到端跟踪时可能是有用的。

跟踪上下文可以由服务手动传播：

- 当使用LightStep跟踪器时，Envoy依靠该服务传播 `x-ot-span-context` HTTP头，同时向其他服务发送HTTP请求。
- 当使用Zipkin示踪器时，Envoy依靠该服务来传播官方的B3 HTTP报头（ `x-b3-traceid` ， `x-b3-spanid` ， `x-b3-parentspanid` ， `x-b3-sampled` 和 `x-b3-flags` ）或为了方便起见，也可以传播 `x-ot-span-context` HTTP头。

注意：分布式跟踪社区中正在进行工作以定义跟踪上下文传播的标准。 一旦采用了合适的方法，用于传播Zipkin跟踪上下文的非标准单头 `x-ot-span-context` 的使用将被替换。

每个跟踪包含哪些数据
端到端跟踪由一个或多个spans组成。spans表示具有开始时间和持续时间的逻辑工作单元，并且可以包含与其关联的元数据。 Envoy生成的每个spans包含以下数据：

- 通过 `--service-cluster` 设置始发服务集群。
- 开始时间和请求的持续时间。
- 始发主机通过 `--service-node` 设置。
- 通过 `x-envoy-downstream-service-cluster` 头设置下游集群。
- HTTP网址。
- HTTP方法。
- HTTP响应代码。
- 跟踪系统中的特定元数据。

范围还包括一个名称（或操作），默认情况下被定义为被调用的服务的主机。但是，这可以使用路线上的装饰器来定制。该名称也可以使用 `x-envoy-decorator-operation` 头域字段替代。

Envoy自动发送spans跟踪收集。根据跟踪收集器的不同，使用通用信息（如全局唯一请求标识 `x-request-id` （LightStep）或跟踪标识配置（Zipkin））将多个spans拼接在一起。

关于如何在Envoy中设置跟踪详细信息，请参考手册。

- v1 API参考
- v2 API参考

返回

- 架构介绍
- 简介
- 首页目录

TCP代理

TCP代理

由于Envoy基本上是作为L3/L4服务器编写的，因此很容易实现基本的L3/L4代理。TCP代理过滤器在下游客户端和上游群集之间执行基本的1：1网络连接代理。它本身可以用作替代通道，或与其他过滤器（如MongoDB过滤器或速率限制过滤器）结合使用。

TCP代理过滤器将遵守每个上游集群的全局资源管理器施加的连接限制。TCP代理过滤器检查上游集群的资源管理器是否可以创建连接，而不会超过该集群的最大连接数，如果它不能通过TCP代理进行连接。

TCP代理过滤器配置参考。

返回

- 架构介绍
- 简介
- 首页目录

访问日志

访问日志

HTTP连接管理器和tcp代理支持具有以下可扩展的访问日志记录功能：

- 每个连接管理器或tcp代理的任意数量的访问日志。
- 异步IO非阻塞架构。 访问日志记录不会阻塞主要的网络处理线程。
- 可定制的访问日志格式使用预定义的字段以及任意的HTTP请求和响应头。
- 可自定义的访问日志过滤器，允许将不同类型的请求和响应写入不同的访问日志。

访问日志配置。

返回

- 架构介绍
- 简介
- 首页目录

MongoDB

MongoDB

Envoy支持具有以下功能的网络级别MongoDB过滤器：

- MongoDB格式的BSON解析器。
- 详细的MongoDB查询/操作统计信息，包括路由集群的计时和 `scatter/multi-get` 计数。
- 查询记录。
- 每个通过 `$comment` 查询参数的 `callsite` 统计信息。
- 故障注入。

MongoDB过滤器是Envoy的可扩展性和核心抽象的一个很好的例子。 在Lyft中，我们在所有应用程序和数据库之间使用这个过滤器。它提供了对应用程序平台，及正在使用的特定MongoDB驱动程序不可知的重要数据源。

MongoDB代理过滤器配置参考。

返回

- 架构介绍
- 简介
- 首页目录

DynamoDB

DynamoDB

Envoy支持具有以下功能的HTTP级别DynamoDB过滤器：

- DynamoDB API请求/响应解析器。
- DynamoDB每个操作/表/分区和操作统计。
- 4xx响应的失败类型统计信息，从响应JSON分析，例如 `ProvisionedThroughputExceededException`。
- 批量操作部分的失败统计。

DynamoDB过滤器是Envoy在HTTP层的可扩展性和核心抽象的一个很好的例子。在Lyft中，我们使用此过滤器与DynamoDB进行所有应用程序通信。 它为使用中的应用程序平台和特定的AWS SDK提供了宝贵的数据不可知的来源。

DynamoDB过滤器配置。

返回

- 架构介绍
- 简介
- 首页目录

Redis

Redis

Envoy可以做为Redis代理，在集群中的实例之间对命令进行分区。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。将Envoy与Redis集群进行比较时，这是重点。Envoy的缓存设计的不足够强大，这意味着它不会尝试协调不一致的数据，无法保持全局一致的群集成员关系视图。

Redis项目提供了与Redis相关的分区的全面参考。请参阅“分区：如何在多个Redis实例之间分割数据”。

Envoy Redis的特点：

- Redis协议编解码
- 基于Hash散列的分区
- Ketama发行
- 详细的命令统计
- 主动和被动健康检查

未来计划增强特性：

- 补充时间统计
- 断路
- 请求分散命令
- 复制
- 内置重试
- 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。

配置相应集群所定义的Hash环负载平衡。

如果需要进行主动健康检查，则应该对群集使用Redis健康检查配置。

如果需要被动健康检查，还要配置异常检测。

为了进行被动健康检查，将超时，命令超时和连接关闭映射连接到5xx。来自Redis的所有其他响应被视为成功。

支持的命令

在协议级别，支持管道。不是MULTI（事务块）。尽可能使用流水线来获得最佳性能。

在命令级别，Envoy仅支持可靠地散列到服务器的命令。因此，所有支持的命令都包含一个key。受支持的命令在功能上与原始Redis命令相同，除非可能出现故障。

有关每个命令用法的详细信息，请参阅官方的Redis命令参考。

命令	组
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
HDEL	Hash
HEXISTS	Hash
HGET	Hash
HGETALL	Hash
HINCRBY	Hash
HINCRBYFLOAT	Hash
HKEYS	Hash
HLEN	Hash
HMGET	Hash
HMSET	Hash

HSCAN	Hash
HSET	Hash
HSETNX	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

失败模型

如果Redis抛出一个错误，我们把这个错误作为响应传递给这个命令。 Envoy将错误数据类型的Redis响应视为正常响应，并将其传递给调用者。

Envoy也可以产生自己的错误来回应客户。

错误	含义
no upstream host	哈希环负载均衡器在为Key选择的环位置上没有可用的主机。
upstream failure	后端在超时期限内没有响应或关闭连接。
invalid request	命令由于数据类型或长度而被命令拆分器的第一阶段拒绝。
unsupported command	该命令不被Envoy识别，因此不能被服务，因为它不能被散列到后端服务器。
finished with n errors	对响应进行求和的分段命令（例如，DEL）将返回接收到的错误的总数。
upstream protocol error	碎片命令接收到意外的数据类型或后端以不符合Redis协议的响应进行响应。
wrong number of arguments for command	特定的命令检查Envoy参数的数量是正确的。

在MGET的情况下，无法提取单独Key所产生错误响应。例如，如果我们获取五个Keys和两个Keys的后端超时，我们会得到一个错误的响应代替每个值。

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

返回

- 架构介绍
- 简介
- 首页目录

热重启

热重启

易用性是Envoy的主要设计目标之一。除了强大的统计数据和本地管理界面之外，Envoy还具有“热”或“实时”重启的能力。这意味着Envoy可以完全重新加载自己（代码和配置）而不会丢失任何连接。热重启功能具有以下通用架构：

- 统计和一些锁保存在共享内存区域。这意味着在重启过程中，资源将在两个进程中保持一致。
- 两个活动进程使用基本的RPC协议通过unix域套接字相互通信。
- 新进程完全初始化自己（加载配置，执行初始服务发现和健康检查阶段等），然后再向旧进程请求监听套接字的副本。新流程开始监听，然后告诉旧流程开始退出。
- 在退出阶段，旧的进程试图正常关闭现有的连接。如何完成取决于配置的过滤器。退出时间可通过 `--drain-time-s` 选型进行配置，并且随着退出时间的增加，退出更加积极。
- 退出顺序后，新的Envoy进程告诉旧的Envoy进程关闭自己。这一次可以通过 `--parent-shutdown-time-s` 选项来配置。
- Envoy的热重启支持被设计成，即使新的Envoy进程和旧的特使进程在不同的容器内运行，它也能正常工作。进程之间的通信仅使用unix域套接字进行。
- 源代码发行版中包含以Python编写的示例重启/父进程。这个父进程可用于标准的进程控制工具，如monit/runit/等。

返回

- 架构介绍
- 简介
- 首页目录

动态配置

动态配置

Envoy架构支持多种的配置管理方法。采用哪种部署方法，取决于需求实现者。可以采用全静态的配置方式，实现简单的部署。更复杂的动态部署，需要采用更复杂的动态配置，需要基于实现者提供一个或多个外部REST的配置API。本文档概述了可用的配置选项。

- 全量参考配置
- 安装参考配置
- Envoy v2 API概述

术语

- SDS（Service Discovery Service）
- EDS（Endpoint Discovery Service）
- CDS（Cluster Discovery Service）
- RDS（Route Discovery Service）
- LDS（Listener Discovery Service）

完全静态

在完全静态配置中，实现者提供了一组监听器、过滤器链、集群以及HTTP路由配置可选。只能通过DNS服务来实现动态主机发现。必须通过内置的热启动机制进行配置的重新加载。

虽然简单，当然可以使用静态配置和优雅的热重启，来实现比较复杂的部署。

仅限SDS/EDS

服务发现服务（SDS）API提供了一种更高级的机制，Envoy可以通过该机制发现上游群集的成员。SDS已在v2API中重命名为Endpoint Discovery Service（EDS）。在静态配置的基础上，SDS允许Envoy部署避开DNS的限制（响应中的最大记录等），并消耗更多信息用于负载平衡和路由（例如，灰度发布，区域等）。

SDS/EDS和CDS

群集发现服务（CDS）API层上Envoy可以发现路由期间使用的上游群集的机制。Envoy将优雅地添加，更新和删除由API指定的集群。这个API允许实现者构建一个拓扑，在这个拓扑中，Envoy在初始配置时，不需要知道所有的上游集群。通常，在与CDS一起进行HTTP路由（但没有路由发现服务）时，实现者将利用路由器将请求转发到HTTP请求标头中指定的集群的能力。

虽然可以通过指定完全静态集群来使用没有SDS/EDS的CDS，但我们建议仍然使用SDS/EDS API来通过CDS指定集群。在内部，更新集群定义时，操作是优雅的。但是，所有现有的连接池将被断开并重新连接。SDS/EDS不受此限制。当通过SDS/EDS添加和删除主机时，群集中的现有主机不受影响。

SDS/EDS，CDS和RDS
路由发现服务（RDS）API层，Envoy可以在运行时发现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。该API与SDS/EDS和CDS一起使用时，允许执行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，不需要任何Envoy重启。

SDS/EDS，CDS，RDS和LDS
监听器发现服务（LDS）在Envoy可以在运行时发现整个监听器的机制上分层。这包括所有的过滤器堆栈，并包含嵌入式参考RDS的HTTP过滤器。在混合中添加LDS可以使Envoy的几乎所有方面都能够进行动态配置。只有非常少见的配置更改（管理员，跟踪驱动等）或二进制更新时才需要热启动。

返回

- 架构介绍
- 简介
- 首页目录

初始化

初始化

Envoy在启动过程中，初始化是很复杂的。本节将从高层次解释工作流程。以下所有情况，都发生在任何新连接接收之前。

- 在启动过程中，集群管理器会经历多阶段初始化，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，等待一个响应（或失败），并执行CDS提供的集群相同的主/次初始化。
- 如果群集使用主动健康检查，Envoy也做一个活跃的健康检查环节。
- 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请求至少有一个响应（或失败）之前，服务器不会开始接受连接。
- 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，这个过程通过LDS发生在每个未来的收听者身上，并被称为收听者变暖。
- 在所有先前的步骤发生之后，听众开始接受新的连接。该流程确保在热启动期间，新进程在旧进程完全退出时，才可以接收并处理新的连接。

返回

- 架构介绍
- 简介
- 首页目录

逐出

退出

退出是指Envoy试图优雅让连接退出各种事件的过程。退出发生在下列时刻：

- 服务器已通过健康检查失败管理端口进行手动触发失败。有关更多信息，请参阅健康检查过滤器体系架构。
- 服务器正在热启动。
- 个别监听器正在通过LDS进行修改或删除。

每个配置的监听器都有一个drain_type设置，用于控制何时发生退出。目前支持的值是：

默认：

Envoy会响应上述三种情况（管理员触发，热启动和LDS更新/删除）。这是默认设置。

modify_only：

Envoy只会响应上述第二、三种情况（热启动和LDS更新/删除）。如果Envoy同时拥有入口和出口监听器，则此设置很有用。可能需要在出口监听器上设置modify_only，以便在尝试进行受控关闭时，依靠入口监听器退出来执行完整的服务器关闭，它们只在修改期间关闭。

请注意，虽然退出是每个监听器的概念，但它必须在网络过滤器级别上得到支持。目前唯一支持正常退出过滤器是HTTP连接管理器，Redis和Mongo。

返回

- 架构介绍
- 简介
- 首页目录

脚本

脚本
Envoy支持使用Lua脚本作为专用HTTP过滤器调试。

返回

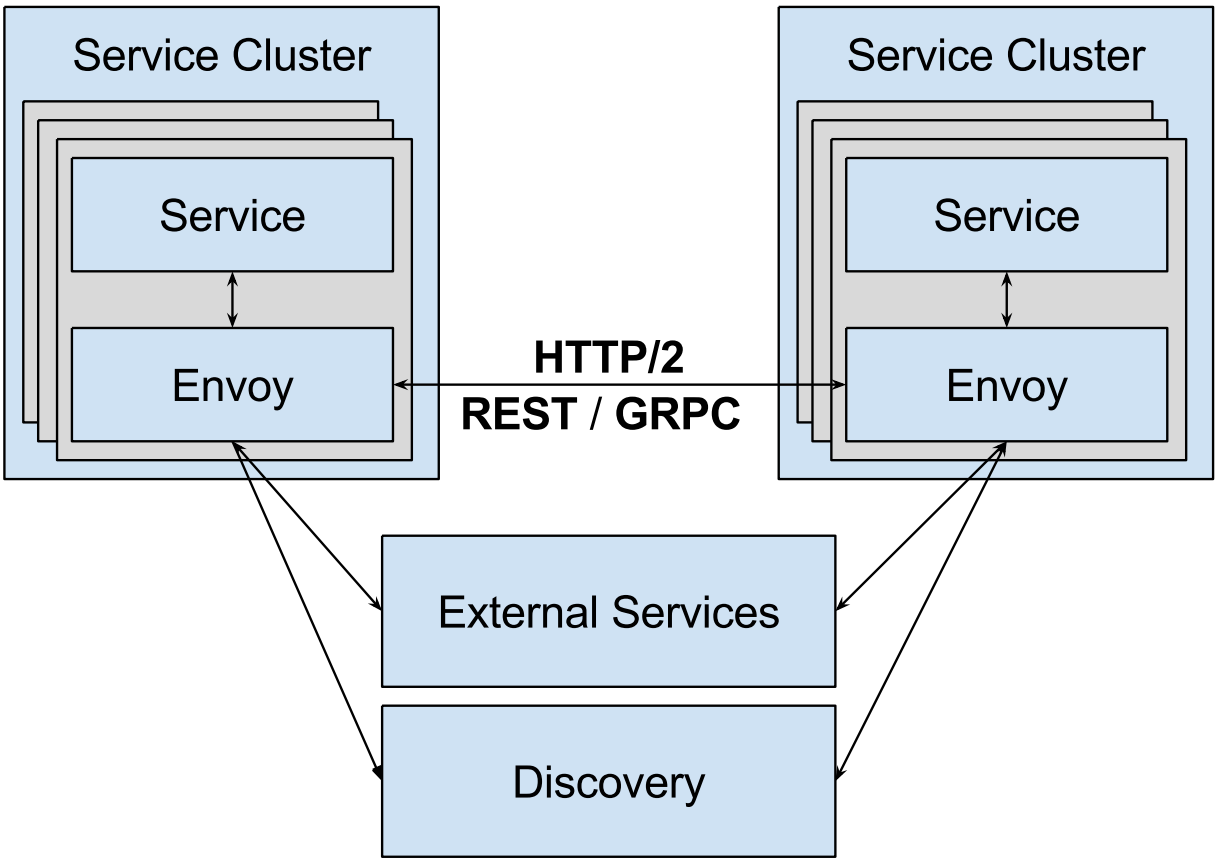
- 架构介绍
- 简介
- 首页目录

部署

部署

Envoy可用于各种不同的场景，但是在跨基础架构中进行所有主机网格部署时，它是最有用的。本节介绍三种推荐的部署方式，其复杂程度越来越高。

服务间



上图显示了最简单的Envoy部署方式，使用Envoy作为通信总线，承担面向服务架构（SOA）内部所有的流量。在这种情况下，Envoy公开了几个用于本地来源流量的监听器，以及用于处理服务的流量。

服务间出口监听器

这是应用程序与基础结构中的其他服务交互的端口。例如，http://localhost:9001 HTTP和gRPC请求使用HTTP/1.1主机头或HTTP/2：根据头来指导请求发往哪个远程群集。Envoy根据详细的配置处理服务发现，负载平衡，速率限制等。服务只需要了解本地的Envoy，不需要关心网络拓扑结构，无论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。

服务间入口监听器
这是远程Envoy想要与当地Envoy交谈时使用的端口。例如，http://localhost:9211 传入的请求被路由到配置的端口上的本地服务。可能会涉及多个应用程序端口，具体取决于应用程序或负载平衡需求（例如，如果服务同时需要HTTP端口和gRPC端口）。本地Envoy根据需要进行缓冲，断路等。

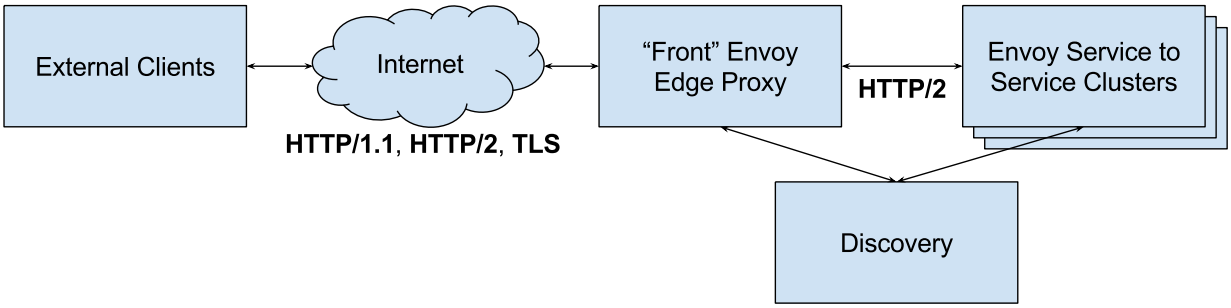
我们的默认配置对所有Envoy通信都使用HTTP/2，而不管应用程序在离开本地Envoy时是否使用HTTP/1.1或HTTP/2。HTTP/2支持长连接和显式重置通知，能够提供更好的性能。

可选的外部服务出口监听器
通常，本地服务要与之通话的每个外部服务都使用明确的出口端口。这样做是因为一些外部服务SDK不轻易理解主机报文头，以支持标准的HTTP反向代理能力。例如，http://localhost:9250 可能被分配给发往DynamoDB的连接。我们建议为所有外部服务保持一致并使用本地端口路由，而不是为某些外部服务使用主机路由，为其他服务使用专用本地端口路由。

集成发现服务
建议的配置使用外部发现服务进行所有群集发现。这为Envoy提供了在执行负载平衡，统计收集等时可能使用的详细的服务发现信息。

配置模板
源代码发行版包含一个配置示例，与Lyft在生产环境中运行的版本非常相似。浏览此处获取更多信息。

服务间+前端代理

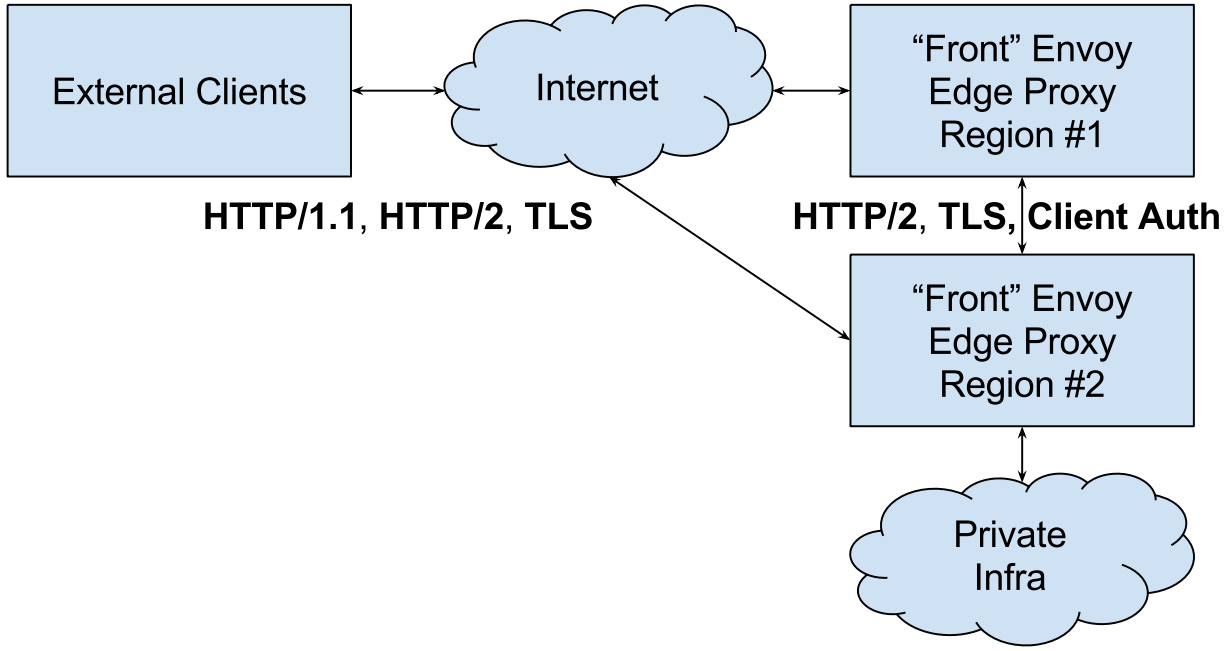


上图显示了服务部署，Envoy作为HTTP L7前端反向代理的群集。反向代理提供以下功能：

- 对外提供TLS安全，对内屏蔽TLS
- 支持HTTP/1.1和HTTP/2
- 完整的HTTP L7路由支持
- 提供标准入站端口，来访问Envoy集群服务，并使用发现服务进行主机查找。因此，前面的Envoy和任何其他的Envoy一样工作，除了他们没有与另一个服务进程搭配在一起。这意味着可以以相同的方式运行，并发出相同的统计数据。

配置模板
源代码发行包含一个与Lyft在生产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。

服务间、前端代理和双重代理



上图显示了作为双重代理，运行了另一个Envoy做为前端代理。双重代理背后的想法是，尽可能地将TLS和客户端连接终止到用户（TLS握手的更短的往返时间，更快的TCP CWND扩展，更少的数据包丢失机会等），会更高效。在双重代理中终止的连接，然后被复用到在主数据中心中运行的HTTP/2长连接。

在上图中，在区域1中，运行的前端Envoy代理通过固定证书与在区域2中运行的前端Envoy代理进行身份验证。这允许在区域2中运行的前端Envoy实例，信任之前不能信任的入站转发请求（例如 `x-forwarded-for` 的HTTP头）。

配置模板

源码分发包含一个与Lyft在生产中运行的版本非常相似的示例双重代理配置。浏览此处获取更多信息。

返回

- 简介
- 首页目录

业界对比

业界对比

总的来说，我们相信Envoy为现代服务导向架构提供了独特且引人注目的功能。下面我们比较一下Envoy和其他相关的系统。尽管在任何特定的领域（边缘代理，软件负载平衡器，服务消息传递层），Envoy可能不像下面的一些解决方案那样具有丰富的功能，但总体而言，没有其他解决方案将相同的整体特征提供到单个自包含的高性能套餐。

注：以下大部分项目也都正在积极开发中。因此，一些信息可能会过时。如果是这种情况，请让我们知道，我们会解决它。

nginx

nginx是规范的现代Web服务器。它支持静态内容服务，HTTP L7反向代理和负载均衡，HTTP/2和许多其他功能。尽管我们认为大多数现代的面向服务的体系结构通常不使用它们，但nginx比Envoy更具有整体特性作为前端反向代理。与nginx相比，Envoy提供了以下主要优势作为前端代理：

- 完整的HTTP/2透明代理。Envoy支持下行和上行通信的HTTP/2。nginx仅支持HTTP/2用于下游连接。
- 自由提供高级负载均衡。只有nginx plus（付费服务器）支持与Envoy类似的高级负载均衡功能。
- 能够在前端以及每个服务节点上运行相同的软件。许多基础设施运行nginx和haproxy的混合。从操作的角度来看，每一跳的单一代理解决方案都要简单得多。

HAProxy

haproxy是规范的现代负载平衡器软件。它也支持基本的HTTP反向代理功能。作为负载均衡器，Envoy提供了比haproxy更多的主要优势：

- HTTP/2支持。
- 可插拔的架构。
- 多线程架构。当每台机器部署单个进程而不是多个进程时，操作和配置电路中断设置要容易得多。
- 与远程服务发现服务集成。
- 与远程全球限速服务集成。
- 能够热重启。
- 更详细的统计。

AWS ELB

Amazon的ELB是用于EC2应用程序服务发现和负载平衡的标准解决方案。Envoy相比ELB作为负载平衡器和服务发现系统，主要提供了以下优势：

- 统计和日志记录（CloudWatch统计信息滞后，并且极其缺乏细节，日志必须从S3中检索，并具有固定格式）。

- 稳定性（使用ELB时偶尔会出现不稳定情况，最终无法调试）。
- 先进的负载平衡，并且与节点之间的直接连接。Envoy网格通过硬件的可伸缩性，避免了额外的网络跳跃。负载平衡器可以做出更好的决策，并根据区域，灰度状态等收集更有价值的统计信息。负载平衡器还支持诸如重试等高级功能。

AWS最近发布了应用程序负载均衡器产品。该产品将支持HTTP/2以及基本HTTP L7请求路由，并添加到多个后端群集。与Envoy相比，这个功能还是很小的，性能和稳定性是未知的，但显然AWS将来会继续在这个领域进行投资。

SmartStack

SmartStack是一个有趣的解决方案，它提供了haproxy之上，额外的支持了服务发现和健康检查。在高层次上，SmartStack与Envoy（外置进程架构，应用程序平台不感知等）具有大部分相同的目标。作为负载平衡器和服务发现软件包，Envoy相对于SmartStack提供了以下主要优势：

- 与haproxy相比，包括前面提到的所有优点。
- 集成服务发现和主动健康检查。Envoy在一个单一的高性能组件中提供了一切。

Finagle

Finagle是Twitter的Scala/JVM服务，用于服务通信库。Twitter和许多其他公司主要使用基于JVM的体系架构。它具有许多与Envoy相同的功能，例如服务发现，负载平衡，过滤器等。与Finagle相比，Envoy作为负载平衡器和服务发现软件包提供了以下主要优势：

- 通过分布式主动健康检查实现最终一致的服务发现。
- 所有度量标准（内存消耗，CPU使用率和P99延迟属性）的性能级别都较好。
- 外置进程和应用程序平台不感知的架构。Envoy可以与任何应用程序工作。

proxygen和wangle

proxygen是Facebook的高性能C++11 HTTP代理库，写在一个叫做wangle的Finagle之类的C++库之上。从代码角度来看，Envoy使用与proxygen大部分相同的技术来获得作为HTTP 库/代理的高性能。除此之外，这两个项目并没有真正的可比性，因为Envoy是一个完整的包含大型功能的独立服务，而不是每个项目都需要单独构建的库。

gRPC

gRPC是一种新的多平台消息传递系统。它使用IDL来描述RPC库，然后为各种不同的语言实现特定于应用程序的运行时。底层传输是HTTP/2。尽管gRPC可能有将来实现许多类似于Envoy的特性（负载平衡等）的目标，但是到目前为止，各种运行时并不成熟，主要关注于序列化/反序列化。我们认为gRPC和Envoy是合作伙伴，并非竞争对手。这里描述了Envoy如何与gRPC集成。

linkerd

linkerd是一个基于Netty和Finagle（Scala/JVM）的独立开源RPC路由代理。linkerd提供了许多Finagle的功能，包括延迟感知负载平衡，连接池，断路，重试阈值，截止期限，跟踪，细粒度检测以及用于请求级路由的流量路由层。linkerd提供了一个可插拔的服务发现接口（标准支持Consul和ZooKeeper以及

Marathon和KubernetesAPI）。

linkerd的内存和CPU要求明显高于Envoy的。与Envoy相比，linkerd提供了极简配置语言，并且明确地不支持热重启，而是依赖于动态配置和服务抽象。linkerd支持HTTP/1.1，Thrift，ThriftMux，HTTP/2（试验阶段）和gRPC（试验阶段）。

nghttp2

nghttp2是一个包含几个不同东西的项目。它主要包含一个实现HTTP/2协议的库（nghttp2）。Envoy使用这个库（在顶层简单封装）来支持HTTP/2。该项目还包含一个非常有用的负载测试工具（h2load）以及一个反向代理（nghttpx）。从比较的角度来看，Envoy与nghttpx最为相似。nghttpx是一个透明的HTTP/1、HTTP/2反向代理，支持TLS终止代理，正确支持gRPC代理以及其他各种功能。有了这个说法，我们认为nghttpx是各种代理功能的一个很好的例子，而不是一个强大的生产就绪解决方案。Envoy的重点更多地集中在可观察性，一般操作敏捷性和高级负载平衡功能上。

返回

- 简介
- 首页目录

获得帮助

获得帮助

我们非常有兴趣在Envoy周围建立一个社区。如果您有兴趣使用它，需要帮助或想贡献，请联系我们。

请参阅联系信息

报告安全漏洞

请参阅安全联系信息

返回

- 简介
- 首页目录

历史版本

历史版本
1.5.0

- access log: added fields for UPSTREAM_LOCAL_ADDRESS and DOWNSTREAM_ADDRESS.
- admin: added JSON output for stats admin endpoint.
- admin: added basic Prometheus output for stats admin endpoint. Histograms are not currently output.
- admin: added version_info to the /clusters admin endpoint.
- config: the v2 API is now considered production ready.
- config: added --v2-config-only CLI flag.
- cors: added CORS filter.
- health check: added x-envoy-immediate-health-check-fail header support.
- health check: added reuse_connection option.
- http: added per-listener stats.
- http: end-to-end HTTP flow control is now complete across both connections, streams, and filters.
- load balancer: added subset load balancer.
- load balancer: added 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.
- log: added the ability to optionally log to a file instead of stderr via the --log-path option.
- listeners: added drain_type option.
- lua: added experimental Lua filter.
- mongo filter: added fault injection.
- mongo filter: added “drain close” support.
- outlier detection: added HTTP gateway failure type. See DEPRECATED.md for outlier detection stats deprecations in this release.
- redis: the redis proxy filter is now considered production ready.
- redis: added “drain close” functionality.
- router: added x-envoy-overloaded support.
- router: added regex route matching.
- router: added custom request headers for upstream requests.
- router: added downstream IP hashing for HTTP ketama routing.
- router: added cookie hashing.
- router: added start_child_span option to create child span for egress calls.
- router: added optional upstream logs.

- router: added complete custom append/override/remove support of request/response headers.
- router: added support to specify response code during redirect.
- router: added configuration to return either a 404 or 503 if the upstream cluster does not exist.
- runtime: added comment capability.
- server: change default log level (-l) to info.
- stats: maximum stat/name sizes and maximum number of stats are now variable via the --max-obj-name-len and --max-stats options.
- tcp proxy: added access logging.
- tcp proxy: added configurable connect retries.
- tcp proxy: enable use of outlier detector.
- tls: added SNI support.
- tls: added support for specifying TLS session ticket keys.
- tls: allow configuration of the min and max TLS protocol versions.
- tracing: added custom trace span decorators.
- Many small bug fixes and performance improvements not listed.

1.4.0

- macOS is now supported. (A few features are missing such as hot restart and original destination routing).
- YAML is now directly supported for config files.
- Added /routes admin endpoint.
- End-to-end flow control is now supported for TCP proxy, HTTP/1, and HTTP/2. HTTP flow control that includes filter buffering is incomplete and will be implemented in 1.5.0.
- Log verbosity compile time flag added.
- Hot restart compile time flag added.
- Original destination cluster and load balancer added.
- WebSocket is now supported.
- Virtual cluster priorities have been hard removed without deprecation as we are reasonably sure no one is using this feature.
- Route validate_clusters option added.
- x-envoy-downstream-service-node header added.
- x-forwarded-client-cert header added.
- Initial HTTP/1 forward proxy support for absolute URLs has been added.
- HTTP/2 codec settings are now configurable.
- gRPC/JSON transcoder filter added.
- gRPC web filter added.
- Configurable timeout for the rate limit service call in the network and HTTP rate limit filters.

- x-envoy-retry-grpc-on header added.
- LDS API added.
- TLS require_client_certificate option added.
- Configuration check tool added.
- JSON schema check tool added.
- Config validation mode added via the --mode option.
- --local-address-ip-version option added.
- IPv6 support is now complete.
- UDP statsd_ip_address option added.
- Per-cluster DNS resolvers added.
- Fault filter enhancements and fixes.
- Several features are deprecated as of the 1.4.0 release. They will be removed at the beginning of the 1.5.0 release cycle. We explicitly call out that the HttpFilterConfigFactory filter API has been deprecated in favor of NamedHttpFilterConfigFactory.
- Many small bug fixes and performance improvements not listed.

1.3.0

- As of this release, we now have an official breaking change policy. Note that there are numerous breaking configuration changes in this release. They are not listed here. Future releases will adhere to the policy and have clear documentation on deprecations and changes.
- Bazel is now the canonical build system (replacing CMake). There have been a huge number of changes to the development/build/test flow. See /bazel/README.md and /ci/README.md for more information.
- Outlier detection has been expanded to include success rate variance, and all parameters are now configurable in both runtime and in the JSON configuration.
- TCP level listener and cluster connections now have configurable receive buffer limits at which point connection level back pressure is applied. Full end to end flow control will be available in a future release.
- Redis health checking has been added as an active health check type. Full Redis support will be documented/supported in 1.4.0.
- TCP health checking now supports a “connect only” mode that only checks if the remote server can be connected to without writing/reading any data.
- BoringSSL is now the only supported TLS provider. The default cipher suites and ECDH curves have been updated with more modern defaults for both listener and cluster connections.
- The header value match rate limit action has been expanded to include an expect match parameter.
- Route level HTTP rate limit configurations now do not inherit the virtual host level

configurations by default. The include_vh_rate_limits to inherit the virtual host level options if desired.
- HTTP routes can now add request headers on a per route and per virtual host basis via the request_headers_to_add option.
- The example configurations have been refreshed to demonstrate the latest features.
- per_try_timeout_ms can now be configured in a route’ s retry policy in addition to via the x-envoy-upstream-rq-per-try-timeout-ms HTTP header.
- HTTP virtual host matching now includes support for prefix wildcard domains (e.g., *.lyft.com).
- The default for tracing random sampling has been changed to 100% and is still configurable in runtime.
- HTTP tracing configuration has been extended to allow tags to be populated from arbitrary HTTP headers.
- The HTTP rate limit filter can now be applied to internal, external, or all requests via the request_type option.
- Listener binding now requires specifying an address field. This can be used to bind a listener to both a specific address as well as a port.
- The MongoDB filter now emits a stat for queries that do not have $maxTimeMS set.
- The MongoDB filter now emits logs that are fully valid JSON.
- The CPU profiler output path is now configurable.
- A watchdog system has been added that can kill the server if a deadlock is detected.
- A route table checking tool has been added that can be used to test route tables before use.
- We have added an example repo that shows how to compile/link a custom filter.
- Added additional cluster wide information related to outlier detection to the /clusters admin endpoint.
- Multiple SANs can now be verified via the verify_subject_alt_name setting. Additionally, URI type SANs can be verified.
- HTTP filters can now be passed opaque configuration specified on a per route basis.
- By default Envoy now has a built in crash handler that will print a back trace. This behavior can be disabled if desired via the --define=signal_trace=disabled Bazel option.
- Zipkin has been added as a supported tracing provider.
- Numerous small changes and fixes not listed here.

1.2.0

- Cluster discovery service (CDS) API.
- Outlier detection (passive health checking).
- Envoy configuration is now checked against a JSON schema.
- Ring hash consistent load balancer, as well as HTTP consistent hash routing based on a policy.

- Vastly enhanced global rate limit configuration via the HTTP rate limiting filter.
- HTTP routing to a cluster retrieved from a header.
- Weighted cluster HTTP routing.
- Auto host rewrite during HTTP routing.
- Regex header matching during HTTP routing.
- HTTP access log runtime filter.
- LightStep tracer parent/child span association.
- Route discovery service (RDS) API.
- HTTP router x-envoy-upstream-rq-timeout-alt-response header support.
- use_original_dst and bind_to_port listener options (useful for iptables based transparent proxy support).
- TCP proxy filter route table support.
- Configurable stats flush interval.
- Various third party library upgrades, including using BoringSSL as the default SSL provider.
- No longer maintain closed HTTP/2 streams for priority calculations. Leads to substantial memory savings for large meshes.
- Numerous small changes and fixes not listed here.

  1.1.0
- Switch from Jannson to RapidJSON for our JSON library (allowing for a configuration schema in 1.2.0).
- Upgrade recommended version of various other libraries.
- Configurable DNS refresh rate for DNS service discovery types.
- Upstream circuit breaker configuration can be overridden via runtime.
- Zone aware routing support.
- Generic header matching routing rule.
- HTTP/2 graceful connection draining (double GOAWAY).
- DynamoDB filter per shard statistics (pre-release AWS feature).
- Initial release of the fault injection HTTP filter.
- HTTP rate limit filter enhancements (note that the configuration for HTTP rate limiting is going to be overhauled in 1.2.0).
- Added refused-stream retry policy.
- Multiple priority queues for upstream clusters (configurable on a per route basis, with separate connection pools, circuit breakers, etc.).
- Added max connection circuit breaking to the TCP proxy filter.
- Added CLI options for setting the logging file flush interval as well as the drain/shutdown time during hot restart.
- A very large number of performance enhancements for core HTTP/TCP proxy flows as well as a few new configuration flags to allow disabling expensive features if they are not needed (specifically request ID generation and dynamic response code stats).

- Support Mongo 3.2 in the Mongo sniffing filter.
- Lots of other small fixes and enhancements not listed.

1.0.0

- Initial open source release.

返回

- 简介
- 首页目录

编译安装

编译&安装

编译

- 要求
- 预编译二进制

参考配置

- 配置生成器
- 烟雾测试配置

沙箱

- 前端代理
- Zipkin跟踪
- Jaeger跟踪
- gRPC桥接
- 构建Envoy Docker镜像

工具

- 配置加载检测工具
- 路由表检查工具
- 模式验证工具

返回

- 首页目录

编译

构建

Envoy使用Bazel工具构建系统。为了简化初次构建以及快速入门，我们提供了一个基于Ubuntu16的Docker容器镜像，其中包含了构建静态链接Envoy所需的所有内容，请参阅ci/README.md。

如果需要手动构建，请按照bazel/README.md中的说明进行操作。

要求

Envoy最初是在Ubuntu 14 LTS上开发和部署的。它也可以在任何的最新Linux上运行，包括Ubuntu 16 LTS。

构建Envoy需要满足以下要求：

- GCC 5+（用于支持C++14）。
- 预构建的第三方依赖。
- 依赖本地Bazel工具。

有关手动构建的更多信息操作，请参阅CI和Bazel文档链接。

预构建的二进制文件

在每个主提交上，我们创建一组包含Envoy二进制文件的轻量级Docker镜像。 当我们正式发布的时候，我们还会用发布版本标记Docker镜像。

- envoyproxy/envoy：基于Ubuntu Xenial存放带有符号的二进制文件版本。
- envoyproxy/envoy-alpine：基于glibc alpine无符号二进制文件版本。
- envoyproxy/envoy-alpine-debug：基于glibc alpine可调试的二进制文件版本。

我们也会考虑通过社区化运作，发挥大家兴趣来帮助CI，包装和提供额外的二进制类型。如果需要，请在GitHub中添加一个issue。

返回

- 上一级
- 首页目录

参考配置

参考配置

Envoy的源代码发行版中，包含三个主要部署类型的配置模板范例：

- 服务间
- 前端代理
- 双重代理

本文的目标展示Envoy在复杂部署场景下的全部功能。不适用于所有功能。有关完整的文档，请参阅配置参考。

配置生成器

可能配置Envoy已经变的相对复杂。在Lyft中，我们使用jinja模板来让创建和管理配置更容易。源代码发行版包含一个的配置生成器，它大致上与我们在Lyft中使用的版本相似。我们还为以上三种情况提供了三个示例配置模板。

- 配置生成器脚本：configs/configgen.py
- 服务模板：configs/envoy_service_to_service.template.json
- 前端代理模板：configs/envoy_front_proxy.template.json
- 双重代理模板：configs/envoy_double_proxy.template.json

若要生成示例配置，请在repo根目录运行以下命令：

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

上一个命令将使用`configgen.py`中定义的一些变量，展开生成三个完全的配置。 请参阅`configgen.py`中的注释，以获取有关详细扩展配置。

关于示例配置的一些注意事项：

- 假定服务发现服务的实例正在`discovery.yourcompany.net`上运行。
- 假定您的`company.net`的DNS设置了各种各样的东西。在配置模板中修改以支持不同的示例。
- 跟踪默认配置LightStep。要禁用此功能或启用Zipkin跟踪，请删除或更改相应跟踪配置。
- 该配置示例使用全局限速服务。若要禁用此功能，请删除速限相关的配置。
- 配置路由发现的服务，以便服务间引用该配置，并假定它正在`rds.yourcompany.net`上运行。
- 配置集群发现的服务，作为配置参考，假定在`cds.yourcompany.net`上运行。

配置冒烟测试

configs/google_com_proxy.json中提供了一个非常简单的Envoy配置，可用于验证基本纯HTTP代理场景。但并不代表一个实际的Envoy部署。只是用这个来冒烟测试Envoy，如下运行：

```
build/source/exe/envoy -c configs/google_com_proxy.json -l debug
curl -v localhost:10000
```

返回

- 上一级
- 首页目录

演示沙箱

沙箱演示

docker-compose 沙箱可以让你在不同的环境上来测试Envoy的功能。我们会根据大家的兴趣，将会添加更多的沙箱来展示不同的功能。以下开箱即用：

- 前端代理
- Zipkin跟踪
- Jaeger跟踪
- gRPC桥接
- 构建Envoy Docker镜像

返回

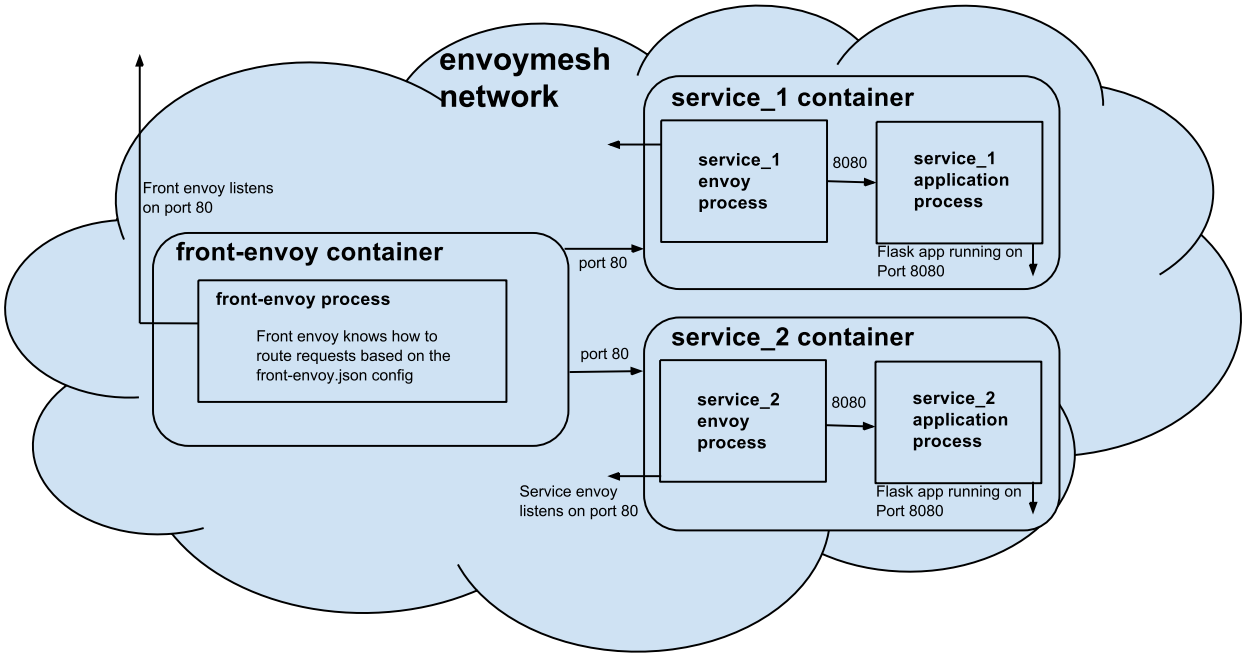
- 上一级
- 首页目录

前端代理

前端代理

为让大家尽快了解Envoy如何作为前端代理，我们发布了一个docker compose沙箱，这个沙箱部署了一个前端Envoy代理和几个后端服务（简单的flask应用），并与一个正在运行的合作的Envoy服务。这三个容器将部署在名为`envoymesh`的虚拟网格中。

该Docker compose的部署图如下所示：



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`（请参阅/examples/front-proxy/docker-compose.yml）。此外，请注意，由前端Envoy路由到容器内的服务，实际上所有流量都是路由到服务的Envoy代理（在/examples/front-proxy/front-envoy.json中设置的路由）。反过来，服务的Envoy通过环回地址（/examples/front-proxy/service-envoy.json中的路由设置）将请求路由到flask应用程序。此阐述了Envoy与您服务搭配的优势：所有请求都由Envoy代理，并有效地路由到您的服务。

运行沙箱

以下文档将按照上图中所述的envoy集群进行运行设置。

第1步：安装Docker工具集

请您确保已经安装了最新版本的`docker`，`docker-compose`和`docker-machine`。

Docker工具箱提供了简单的方法来获取这些工具。

第2步：设置Docker Machine

首先让我们创建一个新的机器来容纳容器：

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

第3步：建立本地Envoy克隆仓库，并启动所有的容器

如果你还没有克隆Envoy仓库，请用git克隆`git clone git@github.com:envoyproxy/envoy`或者`git clone https://github.com/envoyproxy/envoy.git`：

```
$ 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
```

第4步：测试Envoy的路由功能

您现在可以通过前端Envoy向两个服务发送请求。

对于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
```

对于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。

第5步：测试Envoy的负载均衡能力

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会将请求通过负载均衡发给三个service1服务：

```
$ 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: 20da22cfc955 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
```

第6步：进入容器开启curl服务

除了使用主机上的curl外，您还可以自己输入容器并从里面curl。要输入一个容器镜像，你可以使用`docker-compose exec <container_name> /bin/bash`。例如，我们可以进入`front-envoy`容器，并在执行本地的curl服务：

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

第7步：进入容器和使用curl管理

当Envoy运行时，它也将`admin`连接到所需的端口。在示例配置`admin`被绑定到`8001`端口.我们可以`curl`它获得有用的信息。例如，您可以`curl /server_info`来获取有关您正在运行的Envoy版本信息。另外，你可以·curl· ·/stats·得到统计数据。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，我们还可以获得上游群集的成员数量，完成的请求数量，有关http入站的信息以及其他大量有用的统计信息。

返回

- 上一级
- 首页目录

Zipkin跟踪

Zipkin跟踪

Zipkin跟踪演示使用Zipkin作为跟踪服务端，提供跟踪Envoy请求记录展示的功能。这个沙箱与上面描述的前端代理架构非常类似，但有一点不同：在响应返回之前，`service1`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。

所有的请求都经过前端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`（请参阅/examples/zipkin-tracing/docker-compose.yml）。请注意，所有Envoy都配置请求跟踪收集（例如，/examples/zipkin-tracing/front-envoy-zipkin.json中的`http_connection_manager/config/tracing`设置），并将Zipkin设置为跟踪所生成的spans收集器，将发送到Zipkin群集（在/examples/zipkin-tracing/front-envoy-zipkin.json中跟踪驱动程序设置）。

在将请求路由到相应的Envoy服务或应用程序之前，Envoy将负责为跟踪生成适当的spans（父/子/span上下文共享）。在高级别，每个span记录上行API调用的延迟以及将span与其他相关span（例如跟踪ID）关联所需的信息。

从Envoy跟踪的最重要的特点是，它会将调用轨迹发送到Zipkin服务群集。但是，为了充分利用跟踪，应用程序必须传播Envoy生成的跟踪头，同时调用其他服务。在我们提供的沙箱中，`service1`充当了简单`flask`应用程序（请参阅/examples/front-proxy/service.py中的跟踪函数）跟踪传播头，同时对`service2`进行远程调用。

运行沙箱

以下文档将按照上图中所述组织的envoy集群的进行设置和运行。

第1步：构建沙箱

若构建这个沙箱示例，并启动示例应用程序，请运行以下命令：

```
$ pwd
envoy/examples/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：产生一些请求负载

您现在可以通过前端Envoy向 `service1` 发送一个请求，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：通过Zipkin UI查看跟踪记录

使用您的浏览器打开 `http://192.168.99.100:9411` 。你应该看到Zipkin仪表板。 如果这个IP地址不正确，可以通过运行 `$ docker-machine ip default` 找到正确的地址。将服务设置为“前端代理”，并将开始时间设置为测试的前几分钟（如：第2步），然后按回车。你应该看到来自前端代理的跟踪记录。单击一个跟踪记录，就可以查看从前端代理到 `service1` 到 `service2` 的请求所经过的路径，以及每个调用所产生的延迟。

返回

- 上一级
- 首页目录

Jaeger跟踪

Jaeger跟踪

Jaeger跟踪沙箱演示使用Jaeger作为跟踪服务程序，为Envoy请求跟踪提供能力。这个沙盒与上面描述的前端代理架构非常类似，但有一点不同：在返回请求响应之前，`service1`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。

所有的请求都通过前端Envoy代理进行路由，该Envoy充当位于`envoymesh`网络边缘的反向代理角色。通过docker compose将端口`80`映射到端口`8000`（请参阅/examples/jaeger-tracing/docker-compose.yml）。请注意，所有Envoy都配置为请求跟踪收集（例如，/examples/jaeger-tracing/front-envoy-jaeger.json中的`http_connection_manager/config/tracing`设置），并将Jaeger设置为跟踪所生成的span收集服务集群（在/examples/jaeger-tracing/front-envoy-jaeger.json中设置跟踪驱动程序）。

在将请求路由到相应的Envoy服务或应用程序之前，Envoy将负责为跟踪生成适当的span（父/子上下文span）。在高级别，每个span会记录上行API调用的延迟以及将span与其他相关span（例如跟踪ID）关联所需的信息。

Envoy跟踪最重要的好处之一是它能够将生成的跟踪记录发送到Jaeger服务集群。但是，为了充分利用跟踪功能，在调用其他服务时，应用程序也必须传递Envoy生成的跟踪头。在我们提供的沙箱中，充当`service1`简单的`flask`应用程序（请参阅/examples/front-proxy/service.py中的跟踪函数）传播跟踪头，同时对`service2`进行远程调用。

运行沙箱

以下文档将按照上图中所述组织的envoy集群的设置运行。

第1步：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
```

第2步：产生一些负载请求

您现在可以通过前端Envoy向 `service1` 发送一个请求，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：通过Jaeger UI查看跟踪记录

使用浏览器打开 `http://localhost:16686` 。您应该可以看到Jaeger仪表板。将服务设置为“前端代理”，然后点击“查找跟踪”。你应该看到来自前端代理的跟踪信息。单击一个跟踪，就可以查看从前端代理到 `service1` 到 `service2` 的请求所经过的路径，以及每个调用产生的延迟。

返回

- 上一级
- 首页目录

gRPC桥接

gRPC桥接

Envoy gRPC

gRPC桥接沙箱是展示Envoy gRPC桥接过滤器的一个例子。其中包含带有Python HTTP客户端的gRPC内存Key/Value存储。Python客户端通过Envoy代理进程发出HTTP/1请求，并将其升级为HTTP/2的gRPC请求。响应随后缓冲，并作为HTTP/1报文负载有效信息发送回客户端。

本例还演示Envoy另一个功能，就是通过Envoy路由配置，具有基础鉴权的路由能力。

构建Go服务

构建&运行Go gRPC服务：

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

注意：构建需要的Envoy代码库（或其工作副本）位于`$GOPATH/src/github.com/envoyproxy/envoy`中。

Docker Compose

运行`docker compose`文件，并设置Python和gRPC在容器中运行：

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

发送请求到Key/Value存储

要使用Python服务并发送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
```

在正在运行的 `docker-compose` 容器中，您应该可以看到gRPC服务打印其运行的记录：

```
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
```

返回

- 上一级
- 首页目录

构建Envoy Docker镜像

构建Envoy Docker镜像

以下步骤指导您构建自己的Envoy二进制文件，并将其放入干净的Ubuntu容器中。

第1步：构建Envoy

使用 `envoyproxy/envoy-build` 编译Envoy。该镜像具有构建Envoy所需的全部软件。在您的Envoy目录中执行如下命令：

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

执行该命令需要一些时间，因为它会在编译一个Envoy二进制文件之后，运行相关测试。

有关构建和差异化构建的更多详细信息，请参阅ci/README.md。

步骤2：仅使用Envoy二进制生成映像

在这一步中，我们将构建一个只有Envoy二进制文件的映像，而不是用来构建它的软件。

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

现在，可以在任何 `Dockerfile` 中更改 `FROM` ，则可以使用此Envoy镜像来构建任何沙箱。

如果您有兴趣，欢迎您修改Envoy并测试，这将特别有用。

返回

- 上一级
- 首页目录

工具

工具

配置加载检查工具

配置加载检查工具校验JSON格式的配置文件，是否符合JSON编码规范，并符合Envoy JSON模式。 该工具利用 `test/config_test/config_test.cc` 中的配置进行校验。在服务配置初始化中加载JSON配置是，会使用到它进行校验。

输入

该工具需要一个PATH做为根目录配置，用于保存JSON Envoy配置文件。该工具将以递归方式遍历文件目录结构，并对每个文件运行配置测试。请记住，该工具将尝试所加载路径中的所有文件。

输出

该工具将输出Envoy日志，用它正在测试的配置初始化服务器配置。如果存在JSON文件格式不正确或不符合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所有找到的配置文件，它将以状态 `EXIT_SUCCESS` 退出。

构建

使用Bazel在本地构建该工具。

```
bazel build //test/tools/config_load_check:config_load_check_tool
```

运行

使用该工具时，如上所述加上PATH路径。

```
bazel-bin/test/tools/config_load_check/config_load_check_tool PATH
```

路由表检查工具

路由表检查工具检查路由器返回的路由参数是否与预期相符。该工具还可用于检查重定向路径，路径重写或主机重写是否与预期相符。

输入

该工具需要两个输入JSON文件：

1. 路由JSON配置文件。配置中找到路由配置JSON文件模式。
2. 工具JSON配置文件。配置中找到工具配置JSON文件模式。工具配置输入文件指定URL（由权限和路径组成）和期望的路由参数值。以及额外可选的参数，如附加标题。

输出

如果任何测试用例与期望的路由参数值不匹配，则程序以 `EXIT_FAILURE` 状态退出。

添加 `--details` 选项可以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。第一个字段是预期的路由参数值。第二个字段是实际的路由参数值。第三个字段表示比较的参数。在下面的示例中，Test_2和Test_5在其他测试通过时失败。该测试案例中，将打印冲突详细信息。

```
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的运行时值进行测试，这可能会在未来添加该功能。

构建

使用Bazel在本地构建该工具。

```
bazel build //test/tools/router_check:router_check_tool
```

运行

该工具需要输入两个json文件和一个可选的命令行选项 `--details` 。命令行参数期望的顺序是：
1.路由器配置json文件；
2.工具配置json文件；
3.可选的详细信息标识；

```
bazel-bin/test/tools/router_check/router_check_tool router_config.json tool_c
```

```
bazel-bin/test/tools/router_check/router_check_tool router_config.json tool_c
```

测试

通过bash的shell脚本，使用bazel运行测试。该测试使用不同的路由和工具的配置json文件，进行比较。这些配置json文件可以在 `bazel test //test/tools/router_check/...` 中找到。

```
bazel test //test/tools/router_check/...
```

模式验证检查工具

模式验证程序工具验证传入的JSON是否符合配置规范。要验证整个配置，请参阅上述配置加载检查工具章节。目前只支持路由配置模式验证。

输入

该工具需要两个输入：

1. 检查在JSON中传递的模式类型。支持的类型是：
    - route - 用于路由配置验证；
2. JSON的路径；

输出

如果JSON符合模式，则该工具将以状态`EXIT_SUCCESS`退出。 如果JSON不符合模式，则会输出错误消息，详细说明不符合模式的内容。该工具将以状态`EXIT_FAILURE`退出。

构建

使用Bazel在本地构建该工具。

```
bazel build //test/tools/schema_validator:schema_validator_tool
```

运行

该工具如上所述采取路径。

```
bazel-bin/test/tools/schema_validator/schema_validator_tool  --schema-type SC
```

返回

- 上一级
- 首页目录

配置参考

配置参考

V1 API概述

V2 API概述

- 引导配置
- 示例
- 管理服务
- 聚合服务发现
- 状态

监听器

- 统计
- 运行时
- 监听服务发现 (LDS)

网络过滤器

- TLS客户端身份认证
- Echo
- Mongo代理
- 速率限制
- Redis代理
- TCP代理

HTTP连接管理器

- 路由匹配
- 流量转移/分流
- HTTP头部操作
- HTTP头部清理
- 统计
- 运行时设置
- 路由发现服务

HTTP过滤器

- 缓存
- CORS过滤器
- 故障注入
- DynamoDB
- gRPC HTTP/1.1 桥接

- gRPC-JSON 转码过滤器
- gRPC-Web 过滤器
- 健康检查
- 速率限制
- 路由
- Lua

集群管理

- 统计
- 运行时配置
- 集群发现服务
- 健康检查
- 熔断

访问日志

- 配置
- 格式规则
- 默认格式

限速服务

- gRPC IDL

运行

- 文件系统层
- 注释
- 通过符号链接更新运行值
- 统计

路由表检查工具

返回

- 首页目录

V1 API 概述

V1 API概述

注意：当前V1配置被认为是历史遗留的。它将在未来Envoy的版本中被弃用，并最终完全删除。如果您是Envoy的新手，强烈建议从V2配置API开始。

Envoy配置格式以JSON编写，并针对JSON配置进行校验。配置模式可以在source/common/json/config_schemas.cc中找到。服务器的主要配置包含在监听器和集群管理器部分。其他配置项指定其他配置。

支持YAML提供便利的手写配置。如果配置文件路径以.yaml结尾，Envoy在内部将YAML转换为JSON。在剩下的配置文档中，我们只提供JSON的配置描述。Envoy期望明确一下YAML标量，如一个集群名称（应该是一个字符串）被置为真，它应该在YAML配置中被写为“true”。同样也适用于整数值和浮点值（例如1、1.0和“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（必选，数组）

需要由服务实例化的监听器数组。一个Envoy进程可以包含任意数量的监听器。

lds（可选，对象）

监听发现服务（LDS）的配置。如果未指定，则只加载静态监听器。

admin（必选，对象）

本地管理HTTP服务器的配置。

cluster_manager（必选，对象）

服务内所拥有的所有上游群集的群集管理器配置。

flags_path（可选，字符串）

启动在文件系统路径下搜索文件的标志。

statsd_udp_ip_address（可选，字符串）

符合`statsd`正在运行的UDP监听地址。如果指定了，则统计数据将会刷新到这个地址上。IPv4地址格式`host:port`（例如：`127.0.0.1:855`）。IPv6地址的格式`[host]:port`（例如`[::1]:855`）。

statsd_tcp_cluster_name（可选，字符串）

符合TCP的`statsd`集群管理集群的名称。如果指定，Envoy将连接到此群集以刷新统计信息。

stats_flush_interval_ms（可选，整数）

配置刷新统计信息时间（以毫秒为单位）。出于性能方面的原因，Envoy锁定计数器，并且只是周期性地刷新计数器和计量器。如果未指定，则默认值为5000毫秒（5秒）。

watchdog_miss_timeout_ms（可选，整数）

Envoy统计“server.watchdog_miss”统计信息中的线程无响应的时间（以毫秒为单位）。如果没有指定，默认是200ms。

watchdog_megamiss_timeout_ms（可选，整数）

Envoy统计“server.watchdog_mega_miss”统计信息中的线程无响应的时间（以毫秒为单位）。如果未指定，则默认值为1000毫秒。

watchdog_kill_timeout_ms（可选，整数）

监视一个线程在这个毫秒内没有响应，假定因BUG导致导致kill整个Envoy进程。设置为0表示禁用kill行为。如果未指定，则默认值为0（禁用）。

watchdog_multikill_timeout_ms（可选，整数）

如果至少有两个监视的线程至少在这个毫秒内没有响应，假定因一个真正的死锁导致kill整个Envoy进程。设置为0表示禁用kill行为。如果未指定，则默认值为0（禁用）。

tracing（可选，对象）

外部跟踪提供程序的配置。如果没有指定，则不会执行跟踪。

rate_limit_service（可选，对象）

配置外部限速服务提供商。如果没有指定，任何调用速率限制服务将立即返回成功。

runtime（可选，对象）

运行时配置提供程序的配置。如果未指定，将使用“null”提供程序，这将导致使用所有默认值。

返回

- 上一级
- 首页目录

V2 API 概述

V2 API概述

Envoy的v2 API采用Protobuf3作为数据面API集。由现有v1 API概念演变而来的：

- 基于gRPC流传输xDS API。减少对资源开销和更低的延迟。
- 基于REST-JSON API，其中JSON/YAML格式需要符合proto3-JSON规范。
- 通过文件系统，REST-JSON 或者 gRPC端口进行更新。
- 通过高级负载均衡API端口，向管理服务上报负载和资源利用率信息。
- 更强的一致性和时序属性。V2 API仍然能保持最终一致性模型。

有关Envoy与管理服务之间，消息交互方面的更多详细信息，请参阅xDS协议说明。

引导配置

若要使用v2 API，需要提供配置文件启动程序。这提供了静态服务配置，并配置Envoy以在需要时访问动态配置。与v1的JSON/YAML配置一样，这通过在命令行上提供 `-c` 标志，即：

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

其中扩展标志表示仅支持v2配置。 `--v2-config-only` 标志不是必选的，因为Envoy会尝试自动检测配置文件的版本，当配置解析失败时，这个选项可以增强调试能力。

引导是根配置。引导的一个关键概念是静态和动态资源之间的区别。 `Listener` 或 `Cluster` 等资源可以在 `static_resources` 中静态配置，也可以在 `dynamic_resources` 中配置LDS或CDS之类的动态xDS服务。

例子

下面我们将使用YAML配置原型，表示从 `127.0.0.1:10000` 到 `127.0.0.2:1234` 的HTTP服务代理的运行示例。

全静态

下面提供了一个全静态最小引导配置：

```
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 }}]
```

大部分静态&动态服务发现

下面提供了一个引导配置，从上面的示例中继续，通过在`127.0.0.3:5678`上监听的EDS gRPC管理服务进行动态`endpoint`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_name: [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：上面的`xds_cluster`被定义为Envoy的管理服务。即使在全动态的配置中，也需要定义一些静态资源，如指定Envoy对应的xDS管理服务。

在上面的例子中，请求EDS管理服务，然后返回服务发现应答的原始编码：

```
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
```

以上出现的版本控制和URL类型方案，在流式gRPC订阅协议文档中有更详细的解释。

全动态

下面提供了一个全动态的引导配置，其中除了管理服务的其他所有资源都是通过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_name: [xds_cluster]
  cds_config:
    api_config_source:
      api_type: GRPC
      cluster_name: [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 }}]
```

管理服务可以通过以下方式响应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_name: [xds_cluster]
        http_filters:
        - name: envoy.router
```

管理服务可以通过以下方式响应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 }
```

管理服务可以通过以下方式响应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_name: [xds_cluster]
```

管理服务可以通过以下方式响应EDS请求：

```
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

一个v2 xDS管理服务，需要按照gRPC/REST服务的要求提供以下API端口。在同时支持流式gRPC和REST-JSON情况下，在接收一个发现请求时，按照xDS协议返回发现响应。

gRPC流端口

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

有关服务定义，请参阅cds.proto协议。当Envoy作为客户端时，这被使用：

```
cds_config:
  api_config_source:
    api_type: GRPC
    cluster_name: [some_xds_cluster]
```

在引导配置的 `dynamic_resources` 中设置。

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

有关服务定义，请参阅eds.proto协议。当Envoy作为客户端时，这被使用：

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

在集群配置的 `eds_cluster_config` 字段中设置。

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

有关服务定义，请参阅lds.proto。当Envoy作为客户端时，这被使用：

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

在引导配置的 `dynamic_resources` 中设置。

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

有关服务定义，请参阅rds.proto。当Envoy作为客户端时，这被使用：

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

在Http Connection Manager配置的rds字段中设置。

REST端口
`POST /v2/discovery:clusters`

有关服务定义，请参阅cds.proto协议。当Envoy作为客户端时，这被使用：

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

在引导配置的 `dynamic_resources` 中设置。

`POST /v2/discovery:endpoints`

有关服务定义，请参阅eds.proto协议。当Envoy作为客户端时，这被使用：

```
eds_config:
  api_config_source:
    api_type: REST
    cluster_name: [some_xds_cluster]
```

在集群配置的 `eds_cluster_config` 字段中设置。

`POST /v2/discovery:listeners`

有关服务定义，请参阅lds.proto。当Envoy作为客户端时，这被使用：

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

在引导配置的 `dynamic_resources` 中设置。

`POST /v2/discovery:routes`

有关服务定义，请参阅rds.proto。当Envoy作为客户端时，这被使用：

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

在Http Connection Manager配置的rds字段中设置。

聚合发现服务

虽然Envoy从根本上采用了最终一致性的模型，但是ADS提供了一个机会来对API更新推送进行排序，并确保Envoy节点面向单个管理服务，进行相关API更新。ADS允许一个或多个API及其资源，由管理服务在单个双向gRPC流上进行传输。没有这个，一些API（如RDS和EDS）可能需要管理多个流和连接到不同的管理服务。

ADS将允许通过适当的排序无损地更新配置。例如，假设 foo.com 映射到集群X。我们希望将路由表中 foo.com 映射为集群Y。为此，必须首先传递包含两个集群的CDS/EDS更新X和Y。

如果没有ADS，CDS/EDS/RDS流可能指向不同的管理服务器，或者位于同一个管理服务器上的不同gRPC流/连接进行协商。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允许将这些流合并为单个流到单个管理服务器，避免了分布式同步对正确排序更新的需要。借助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（不是REST），在本文档中有更详细的描述。gRPC端口是：

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

有关服务定义，请参阅discovery.proto。当Envoy作为客户端时，这被使用：

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

在引导配置的 `dynamic_resources` 中设置。

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，一个LDS配置可以从

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

修改为

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

其效果是LDS流将通过共享的ADS通道被引导到`some_ads_cluster`。

状态

除非另有说明，否则将在v2 API参考中描述的所有功能。在v2 API参考和v2 API库中，所有接口原型都被冻结，除非它们被标记为草稿或实验原型。在这里，冻结意味着我们不会打破兼容性的底线。

通过添加新的字段，以不破坏向后兼容性的方式，尽可能的进一步延长冻结原型的期限。上述原型中的字段可能会在不再使用相关功能的情况下，随着策略的改变而被弃用。虽然冻结的API保持其格式兼容性，但我们保留更改原名，文件位置和嵌套关系的权利，这可能导致代码更改中断。我们的目标是尽量减少这里的损失。

当Protos标记为草案(draft)，意味着它们已经接近完成，至少可能在Envoy中部分实施，但可能会在冻结之前破坏原型格式。

当Protos标记为实验性的(experimental)，与原始草案有相同的告警提示，并可能在Envoy执行冻结之前做出重大改变。

当前所有v2 API问题在这里被跟踪。

返回

- 上一级
- 首页目录

监听器

监听器

Envoy配置顶层包含一个监听器列表。每个单独的监听器配置具有以下格式：

- v1 API参考
- v2 API参考

统计

监听器

每个监听器都有一个以 `listener.<address>` 为根的统计树。统计如下：

名称	类型	描述
downstream_cx_total	Counter	连接总数
downstream_cx_destroy	Counter	销毁的连接总数
downstream_cx_active	Gauge	活动的连接总数
downstream_cx_length_ms	Histogram	连接时长，单位毫秒
ssl.connection_error	Counter	错误的TLS连接总数，不包括证书验证失败的
ssl.handshake	Counter	TLS连接握手成功的总数
ssl.session_reused	Counter	TLS会话恢复成功的总数
ssl.no_certificate	Counter	完全成功的TLS连接，没有客户端证书
ssl.fail_no_sni_match	Counter	由于缺少SNI匹配而被拒绝的TLS连接总数
ssl.fail_verify_no_cert	Counter	由于缺少客户端证书而失败的TLS连接总数
ssl.fail_verify_error	Counter	CA验证失败的TLS连接总数
ssl.fail_verify_san	Counter	SAN验证失败的TLS连接总数
ssl.fail_verify_cert_hash	Counter	证书锁定验证失败的TLS连接总数
ssl.cipher.	Counter	使用的TLS连接总数

监听管理器

监听器管理器的统计树以 `listener_manager` 为根。用下面的统计。统计名称中的字符被替换为_。

名称	类型	描述
listener_added	Counter	监听器被添加的总数（通过静态配置或LDS）
listener_modified	Counter	监听器被修改的总数（通过LDS）
listener_removed	Counter	监听器被删除的总数（通过LDS）
listener_create_success	Counter	监听器对象添加到工作组成功的总数。

listener_create_failure	Counter	监听器对象添加到工作组失败的总数。
total_listeners_warming	Gauge	当前正在热身的监听器的数量
total_listeners_active	Gauge	当前活动的监听器的数量
total_listeners_draining	Gauge	当前正在被引流的监听器数量

运行时

监听器支持以下运行时设置：

ssl.alt_alpn

有多少的百分比请求，使用配置的alt_alpn协议字符串。默认为0。

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。

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，那么在监听器移动到“活动”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）。这个监听者会通过上面所描述的，新增加的监听者都有“预热”过程。
- 当更新或删除监听器时，旧的监听器将被置于“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。

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
config_reload	Counter	由于配置更新，导致配置API调用总数
update_attempt	Counter	LDS配置API调用重试总数
update_success	Counter	LDS配置API调用成功总数
update_failure	Counter	LDS配置API调用失败总数（网络或模式错误）
version	Gauge	上次成功调用的内容哈希值

返回

- 上一级
- 首页目录

网络过滤器

网络过滤器

除了配置指南中足够大的HTTP连接管理器之外，Envoy还拥有内置的网络过滤器。

- TLS客户端身份认证
  - 统计
  - REST API
- Echo
- Mongo代理
  - 故障注入
  - 统计
  - 运行时
  - 访问日志格式
- 速率限制
  - 统计
  - 运行时
- Redis代理
  - 统计
  - 分配统计
  - 按命令统计
  - 运行时
- TCP代理
  - 统计

返回

- 上一级
- 首页目录

TLS客户端身份认证

TLS客户端身份认证

- 客户端TLS认证过滤器架构概述
- v1 API 参考
- v2 API 参考

统计

每个配置的TLS客户端身份验证过滤器统计信息均以 `auth.clientssl.<stat_prefix>` 为根。 统计如下：

名称	类型	描述
update_success	Counter	身份更新成功总数
update_failure	Counter	身份更新失败总数
auth_no_ssl	Counter	由于没有TLS而被忽略的连接总数
auth_ip_white_list	Counter	由于IP白名单允许的连接总数
auth_digest_match	Counter	由于证书匹配而允许连接总数
auth_digest_no_match	Counter	由于没有证书匹配，被拒绝的连接总数
total_principals	Gauge	总加载的身份信息

REST API

`GET /v1/certs/list/approved`

身份验证过滤器将每间隔调用一次此API，以刷新获取当前已批准的证书/主体列表。预期的JSON响应，如下所示：

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

certificates

(必选,数组) 列出已批准的证书/身份。

每个证书对象被定义为：

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

fingerprint_sha256

(必选,字符串) 经批准的客户端证书的SHA256哈希。Envoy将会用这个哈希值与客户端所提供的证书进行匹配，以确定是否存在摘要匹配。

返回

- 上一级
- 首页目录

Echo

Echo

Echo是一个微不足道的网络过滤器，主要是为了演示网络过滤器API。如果安装，它会将所有接收到的数据回显（写入）连接的下游客户端。

- v1 API 参考
- v2 API 参考

返回

- 上一级
- 首页目录

Mongo代理

Mongo代理

- MongoDB体系结构概述
- v1 API 参考
- v2 API 参考

故障注入

Mongo代理过滤器支持故障注入。有关如何配置，请参阅v1和v2 API参考。

统计

每个配置的MongoDB代理过滤器的统计信息都以 `mongo.<stat_prefix>` 为根。统计如下：

名称	类型	描述
decoding_error	Counter	MongoDB协议解码错误的数量
delay_injected	Counter	注入被延迟的次数
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	具有可设置cursor标示的OP_QUERY的数量
op_query_no_cursor_timeout	Counter	没有设置cursor超时标志的OP_QUERY的数量
op_query_await_data	Counter	具有等待数据标志的OP_QUERY的数量
op_query_exhaust	Counter	设置耗尽标志的OP_QUERY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	未找到设置cursor标志的OP_REPLY数量
op_reply_query_failure	Counter	设置了查询失败标志的OP_REPLY数量
op_reply_valid_cursor	Counter	具有有效cursor标志的OP_REPLY数量
cx_destroy_local_with_active_rq	Counter	使用在本地销毁查询连接总数
cx_destroy_remote_with_active_rq	Counter	使用远程销毁查询连接总数
cx_drain_close	Counter	在服务退出时，连接被优雅关闭的总数

分散GETS

Envoy将 `scatter get` 定义为任何不使用 `_id` 字段作为查询参数的查询。同时在文档以及 `_id` 的 `$query` 字段中查找。

多重GETS

Envoy将 `multi get` 定义为任何使用 `_id` 字段作为查询参数的查询，但其中 `_id` 不是标量值（即文档或数组）。同时在文档以及 `_id` 的 `$query` 字段中查找。

注释解析

如果一个查询的顶层有一个 `$comment` 字段（通常添加了一个 `$query` 字段），Envoy会将其解析为JSON并查找以下结构：

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

callingFunction

(required, string) 查询功能。如果可用，该函数将用于调用查询点的统计。

按命令统计

MongoDB过滤器将收集命名空间为 `mongo.<stat_prefix>.cmd.<cmd>.` 中命令的统计信息。

名称	类型	描述
total	Counter	命令的数量
reply_num_docs	Histogram	应答中的文件数量
reply_size	Histogram	应答的字节数（单位bytes）
reply_time_ms	Histogram	命令的时间（单位毫秒）

查询统计收集

MongoDB过滤器将收集mongo中查询的统计信息，命名空间 `mongo.<stat_prefix>.collection.<collection>.query.` 。

名称	类型	描述
total	Counter	查询数量
scatter_get	Counter	分散查询的数量
multi_get	Counter	多重查询的数量
reply_num_docs	Histogram	应答文件的数量
reply_size	Histogram	应答大小（单位：字节）
reply_time_ms	Histogram	查询时间（单位：毫秒）

查询的调用点统计收集

如果应用程序在`$comment`字段中提供调用函数，Envoy将生成每个调用点统计信息。 这些统计信息匹配每个查询信息，匹配的命名空间为`mongo.<stat_prefix>.collection.<collection>.callsite.<callsite>.query.`。

运行时

Mongo代理过滤器支持以下运行配置：

mongo.connection_logging_enabled

- 将启用日志记录连接的百分比。默认为100。若允许有百分之百连接的日志记录，但这些连接上的所有消息都将被记录。

mongo.proxy_enabled

- 将会启用代理连接的百分比。默认为100。

mongo.logging_enabled

- 将被记录的消息的百分比。默认为100，如果小于100，没有查询回复可能会被记录等。

mongo.mongo.drain_close_enabled

- 如果服务器逐出关闭，将会关闭的连接百分比，否则将尝试强制关闭。默认为100。

mongo.fault.fixed_delay.percent

- 当没有活跃的故障时，正常的MongoDB操作，受到注入故障影响的概率。默认为`percent`配置。

mongo.fault.fixed_delay.duration_ms

- 延迟时间以毫秒为单位。默认在使用`duration_ms`配置。

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

- 完整的消息被解析的系统时间，包括毫秒。

message

- 消息的文本扩展。消息是否完全展开取决于上下文。有时会提供汇总数据，以避免超大日志。

upstream_host

- 代理连接的上游主机。如果与TCP代理过滤器一起使用，则会填充此项。

返回

- 上一级
- 首页目录

速率限制

速率限制

- 全局限速架构概述
- v1 API 参考
- v2 API 参考

统计

每个配置的速率限制过滤器的统计信息均以 `ratelimit.<stat_prefix>.` 为前缀。统计如下：

名称	类型	描述
total	Counter	请求限速服务的总数
error	Counter	限速服务请求失败的总数
over_limit	Counter	限速服务的响应上限
ok	Counter	限速服务的响应下限
cx_closed	Counter	因请求超限，而关闭的总连接数
active	Gauge	限速服务活跃请求的总数

运行设置

网络速率限制过滤器支持以下运行设置：

- ratelimit.tcp_filter_enabled

  容许调用速率限制服务连接的百分比。默认为100。

- ratelimit.tcp_filter_enforcing

  调用速率限制服务并执行限速决定连接的百分比。默认为100。这可以用来测试完全速运行之前会发生什么。

返回

- 上一级
- 首页目录

Redis代理

Redis代理

- Redis架构概述
- v1 API 参考
- v2 API 参考

统计

配置Redis代理筛选器的统计信息，均以 `redis.<stat_prefix>.` 为前缀。统计如下：

名称	类型	描述
downstream_cx_active	Gauge	总活动连接
downstream_cx_protocol_error	Counter	总协议错误
downstream_cx_rx_bytes_buffered	Gauge	当前缓冲中收到的字节总数
downstream_cx_rx_bytes_total	Counter	收到的字节总数
downstream_cx_total	Counter	连接总数
downstream_cx_tx_bytes_buffered	Gauge	当前缓冲中发送字节总数
downstream_cx_tx_bytes_total	Counter	发送的字节总数
downstream_cx_drain_close	Counter	由于逐出而关闭的连接数量
downstream_rq_active	Gauge	总活跃请求数量
downstream_rq_total	Counter	总请求数

分配器统计

Redis过滤器会收集redis中命令分配器的统计信息。 `redis.<stat_prefix>.splitter.` 统计如下：

名称	类型	描述
invalid_request	Counter	参数个数错误的请求数
unsupported_command	Counter	分配器无法识别的命令总数

按命令统计

Redis过滤器将收集redis中命令的统计信息。命名空间为 `redis.<stat_prefix>.command.<command>.` 。

名称	类型	描述
total	Counter	命令总数

运行配置

Redis代理筛选器支持以下运行时设置：

redis.drain_close_enabled

如果服务器正在逐出关闭，将会关闭连接的百分比，否则将尝试前置关闭。默认为100。

返回

- 上一级
- 首页目录

TCP代理

TCP代理

- TCP代理架构概述
- v1 API 参考
- v2 API 参考

统计

TCP代理过滤器同时包括下游统计信息以及适用与多集群上游统计信息。下游统计信息的根为`tcp.<stat_prefix>.`统计如下：

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	没有找到匹配路由的连接数
downstream_cx_tx_bytes_total	Counter	写入下游连接的字节总数
downstream_cx_tx_bytes_buffered	Gauge	当前缓冲到下游连接的字节总数
downstream_flow_control_paused_reading_total	Counter	因流量控制暂停从下游读取的总次数
downstream_flow_control_resumed_reading_total	Counter	因流量控制从下游恢复读取的总次数

返回

- 上一级
- 首页目录

HTTP连接管理器

HTTP连接管理器

- HTTP 连接管理器架构概述
- HTTP 协议架构概述
- v1 API 参考
- v2 API 参考

路由匹配

流量转移/分流

- 上游主机间的流量转移
- 多上游主机的流量分流

HTTP头部操作

- 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

HTTP头部清理

- 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-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

统计

- 按代理客户统计
- 按监听端口统计

运行时设置

路由发现服务

- 统计

返回

- 上一级
- 首页目录

路由匹配

路由匹配

注意：本部分是为v1 API编写的，但这些概念也适用于v2 API。它将在未来版本的v2 API中重新描述。

Envoy的路由匹配过程如下：

1. HTTP请求的头域字段 host 或 :authority 与虚拟主机匹配。
2. 按顺序检查虚拟主机中的每个路由表。如果匹配，则使用该路由并且不再匹配路由。
3. 独立地，依次检查虚拟主机中的每个虚拟集群。如果匹配，则使用虚拟群集，不再进一步匹配集群。

返回

- 上一级
- 首页目录

流量转移/分流

流量转移/分流

注意：本文是为v1 API编写的，但这些概念也适用于v2 API。它将在未来版本的v2 API中重新描述。

Envoy路由器支持将流量分发到两个或更多上游虚拟主机群集的路由。有两种常见的场景。

1. 版本升级：到一条路由的流量逐渐从一个集群转移到另一个集群。更详细地描述参见流量转移部分。
2. A/B测试或多重测试：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不同版本相同服务的集群之间进行拆分。更详细地描述参见流量分流部分。

上游主机间的流量转移

在路由运行时配置对象中，确定选择指定路由（以及它的集群）的概率。通过使用运行时配置，虚拟主机中指定路由的流量可逐渐从一个集群转移到另一个集群。参考以下配置示例，其中名为 `helloworld` 的服务，在Envoy配置文件中声明 `helloworld_v1` 和 `helloworld_v2` 两个版本。

```
{
  "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匹配策略是匹配第一个路由。如果路由具有运行时配置对象，则会根据使用运行时值进行匹配（如果没有指定值，则为默认值）。因此，通过在上面的例子中紧挨着的两条路由配置，通过在第一个路由中指定一个运行时对象，并且改变随着时间推移，修改该运行时对象的值来实现流量转移。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` ，以便所有对 `helloworld`

服务的请求，都与v1的路由匹配，并由 `helloworld_v1` 集群提供服务。
2. 要开始将流量转移到 `helloworld_v2` 集群时，请将 `routing.traffic_shift.helloworld` 设置为 `0<x<100` 范围内。例如，设置 `90` ，对 `helloworld` 虚拟主机的每10个请求中，有1个将不匹配v1的路由，并将落入v2路由，由 `helloworld_v2` 集群提供服务。
3. 逐渐设置 `routing.traffic_shift.helloworld` 中的值变小，以便更大比例的请求匹配到v2路由。
4. 当 `routing.traffic_shift.helloworld` 设置为0时，对 `helloworld` 虚拟主机的请求将不会匹配v1路由。现在所有的流量都会流向v2路由，并由 `helloworld_v2` 集群提供服务。

多上游主机的流量分流

再次参考 `helloworld` 的例子，现在有三个版本（v1，v2和v3），而不是两个。 要在三个版本（即33%，33%，34%）之间平均分配流量，可以使用 `weighted_clusters` 选项指定每个上游群集的权重。

与前面的例子不同，一个路由条目就足够了。路由中的 `weighted_clusters` 配置可用于指定多个上游群集以及引导每个发送到上游群集的流量百分比权重。

```
{
  "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 }
              ]
            }
          }
        ]
      }
    ]
  }
}
```

可以使用以下运行时变量动态调整分配给每个群集的权重：

`routing.traffic_split.helloworld.helloworld_v1`
`routing.traffic_split.helloworld.helloworld_v2`
`routing.traffic_split.helloworld.helloworld_v3`。

返回

- 上一级

- 首页目录

HTTP头部操作

HTTP头部操作

HTTP连接管理器在解码期间（当接收到请求时）以及在编码期间（当响应被发送时）处理多个HTTP头部。

user-agent

如果启用了`add_user_agent`选项，则连接管理器在解码过程中，会设置`user-agent`头部。这个头部只有未设置的情况下才会被修改。如果连接管理器确实设置了该字段，则该值由命令行选项`--service-cluster`确定。

server

`server`头部将在编码时设置为选项server_name中的值。

x-client-trace-id

如果一个外部客户端设置了这个头部，Envoy会将客户端提供的跟踪ID和内部生成的`x-request-id`关联起来。`x-client-trace-id`需要全局唯一性，建议生成一个uuid4。如果设置了此标志，它将具有与`x-envoy-force-trace`相似的效果。 请参阅`[tracing.client_enabled](../../Configurationreference/HTTPconnectionmanager/R`运行时配置设置。

x-envoy-downstream-service-cluster

内部服务通常想知道哪个服务正在调用它们。从外部的请求中这个头部会被清除，但是对于内部请求，将包含调用者的服务集群信息。请注意，在当前的实现中，这应该被认为是一个潜规则，因为它是由调用者设置的，并且很容易被任何内部实体欺骗。将来，Envoy将支持相互认证的TLS网格，这将使这个头部完全安全。与`user-agent`一样，该值由`--service-cluster`命令行选项确定。为了启用此功能，您需要将`user_agent`选项设置为`true`。

x-envoy-downstream-service-node

内部服务可能想知道下游节点请求来自哪里。这个头与`x-envoy-downstream-service-cluster`非常相似，除了从`--service-node`选项中取值。

x-envoy-external-address

服务端想要根据客户端IP地址，执行分析是很常见的情况。在XFF(`x-forwarded-for`)漫长的讨论中，这可能会变得相当复杂。一个正确的实现需涉及转发XFF，然后从右边选择第一个非RFC1918地址。由于这个常用的功能，Envoy通过在解码过程中设置`x-envoy-external-address`来简化这种实现，当且仅当请求从外部进入（即它来自外部客户端）。对于内部请求，`x-envoy-external-address`不会设置或覆盖。为了分析需要，可以在内部服务之间安全地转发该头部，而不必关心XFF的复杂性。

x-envoy-force-trace

如果一个内部请求设置了这个头部，Envoy会修改生成的 `x-request-id` ，使得它强制收集跟踪信息。这也迫使 `x-request-id` 在响应头中返回。如果这个请求标识随后被传播到其他主机，那么这些主机上的跟踪也将被收集，这将为整个请求流提供一致的跟踪。请参阅tracing.global_enabled和tracing.random_sampling运行时配置设置。

x-envoy-internal

一个常见的情况就是服务想要知道请求是否来自内部。Envoy使用XFF来明确这一点，然后将头部的值设置为 `true` 。
这是就避免了需要解析和理解XFF的好处。

x-forwarded-client-cert

x-forward-client-cert（XFCC）是一个代理头，在从客户端到服务器的所经过的路径上，表示请求已经流经的部分或全部客户端或代理的证书信息。代理可以在请求代理之前选择清除/追加/转发XFCC头。

XFCC头部值是以逗号（ “,” ）为分隔的字符串。每个子字符串都是XFCC元素，它保存着由一个代理所添加的信息。代理可以将当前客户端证书信息作为XFCC元素，追加到请求的XFCC头部中。

每个XFCC元素都是一个分号“;”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。 `key` 不区分大小写， `value` 区分大小写。如果“,” ， “;”或“=”出现在一个值中，则该 `value` 应该用双引号。 `value` 中的双引号应该被反斜杠双引号（ “）替换。

以下键支持：

1. `By` 当前代理证书的主题备用名称（SAN:Subject Alternative Name）。
2. `Hash` 当前客户端证书的SHA256摘要。
3. `SAN` 当前客户端证书的SAN字段（URI类型）。
4. `Subject` 当前客户端证书的主题字段。这个值总是双引号。

以下是两个XFCC头部的例子：

```
x-forwarded-client-cert: By=http://frontend.lyft.com;Hash=468ed33be74eee6556d90c0149c1309e9ba61d64
25303443c0748a02dd8de688;Subject="/C=US/ST=CA/L=San Francisco/OU=Lyft/CN=Test Client";SAN=http://t
estclient.lyft.com
```

```
x-forwarded-client-cert: By=http://frontend.lyft.com;Hash=468ed33be74eee6556d90c0149c1309e9ba61d64
25303443c0748a02dd8de688;SAN=http://testclient.lyft.com,By=http://backend.lyft.com;Hash=9ba61d6425
303443c0748a02dd8de688468ed33be74eee6556d90c0149c1309e;SAN=http://frontend.lyft.com
```

Envoy处理XFCC的方式由HTTP连接管理器选项 `forward_client_cert` 和 `set_current_client_cert_details` 指定。如果未设置 `forward_client_cert` ，则默认会对XFCC头部进行清除。

x-forwarded-for

x-forwarded-for（XFF）是一个标准的代理头，它表示请求在从客户端到服务器的过程中，所经过的IP地址。一个兼容的代理服务，应在代理请求之前将最近客户端的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` (单个客户端)
2. `x-forwarded-for: 50.0.0.1, 40.0.0.1` (外部代理跳转)
3. `x-forwarded-for: 50.0.0.1, 10.0.0.1` (内部代理跳转)

如果HTTP连接管理器选项 `use_remote_address` 设置为true，Envoy将只附加到XFF。这意味着如果 `use_remote_address` 为 `false` ，则连接管理器将透明模式运行，不修改XFF的内容。这对于特定的网格部署类型是必需的，具体取决于Envoy是在边缘节点还是作为内部服务节点。

最终Envoy会根据的XFF内容来确定请求是从外部还是从内部发起的。这会影响 `x-envoy-internal` 头部是否会被设置。

关于XFF的一些非常重要的注意事项：

1. 由于IP地址被附加到XFF，只有最后一个地址（最右边）可以被信任。更具体地说，右边的第一个外部（非RFC1918）地址是唯一可信的地址。左边的任何东西都可能被欺骗。为了方便处理分析，在前面Envoy还将设置 `x-envoy-external-address` 头部。

2. XFF是Envoy用来确定请求是来自内部还是外部。它通过检查XFF是否包含一个单一的IP地址（RFC1918地址）来做到这一点。

    - 注意：如果内部服务代理到另一个内部服务的外部请求，并且包含原始的XFF头，如果 `use_remote_address` 被设置，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被转发的目的。如果没有打算，不要转发XFF，而是转发 `x-envoy-internal` 。
    - 注意：如果内部服务保留XFF并转发到另一个内部服务，Envoy不会将其视为内部服务。这是一个已知的“BUG”，因为简化XFF的解析处理，以确定请求是否是内部的。在这种情况下，不要转发XFF，并允许Envoy使用一个内部原始IP生成一个新的。

x-forwarded-proto

一个服务若想要知道前端Envoy代理的处理的始发协议类型（HTTP或HTTPS）。这是一个常见的情况。 `x-forward-proto` 包含这个信息。它将会被设置为 `http` 或 `https` 。

x-request-id

Envoy使用 `x-request-id` 头来标识请求，并执行访问日志记录和跟踪。Envoy将为所有来自的外部请求（头部被清理）生成一个 `x-request-id` 头。它还会为内部请求生成一个 `x-request-id` 头，这意味着 `x-request-id` 可以并应该在客户端应用程序之间传递，以便在整个网格中有稳定的ID。由于Envoy是外置进程架构，头部不能由Envoy自动转发。这是少数几个需要瘦客户端库就可以完成的要求之

一，关于如何完成这个，不在本文档范围。如果`x-request-id`跨所有主机传递，则可以使用以下功能：

- 通过v1 API 过滤器或v2 API过滤器进行稳定的访问日志记录。
- 当通过运行时设置`tracing.random_sampling`或通过使用`x-envoy-force-trace`和`x-client-trace-id`头强制使能跟踪，来执行随机跟踪采样。

x-ot-span-context

Envoy使用`x-ot-span-context` HTTP头在span跟踪之间建立适当的父子关系。这个头可以用于LightStep和Zipkin跟踪服务。例如，出口span是入口span（如果存在入口span）的子关系。Envoy在入口请求上注入`x-ot-span-context`头并将其转发给本地服务。Envoy依靠应用程序在出口调用时传播`x-ot-span-context`到上游。在这里查看更多信息。

x-b3-traceid

Zipkin跟踪会使用Envoy中的`x-b3-traceid` HTTP头。TraceId长度为64位，表示跟踪的总体ID。跟踪中的每个span都共享此ID。点击这里查看更多关于zipkin信息。

x-b3-spanid

Zipkin跟踪会用到Envoy中的`x-b3-spanid` HTTP头。SpanId的长度是64位，表示当前操作在跟踪树中的位置。该值不应该被解释：它可以或不可以从TraceId的值中派生出来。点击这里查看更多关于zipkin信息。

x-b3-parentspanid

Zipkin跟踪使用Envoy中的`x-b3-parentspanid` HTTP头。ParentSpanId的长度为64位，表示父操作在跟踪树中的位置。当span是跟踪树的根时，ParentSpanId不存在。点击这里查看更多关于zipkin信息。

x-b3-sampled

Zipkin跟踪会使用Envoy中的`x-b3-sampled` HTTP头。当采样标志为1时，这个索引将被报告给跟踪系统。一旦采样设置为0或1，相同的值应始终向下游发送。点击这里查看更多关于zipkin信息。

x-b3-flags

Zipkin跟踪会使用Envoy中的`x-b3-flags` HTTP头。编码一个或多个选项。例如，Debug被编码为`X-B3-Flags:1`。请参阅上的更多关于zipkin跟踪的信息。

Custom request/response headers

自定义添加请求/响应头，与特定请求/响应路由相匹配，关于虚拟主机和全局路由配置。请参阅相关的v1和v2 API文档。

注意：将按照以下顺序附加到请求/响应中：路由级别头标，虚拟主机级别头标以及最终全局级别头标。

Envoy还支持将动态值添加到请求头域。支持的动态值是：

%CLIENT_IP%

- 由Envoy添加原始客户端IP作为 `x-forwarded-for` 的请求头。

%PROTOCOL%

- 原来的协议已由Envoy添加 `x-forward-proto` 作为请求头。

返回

- 上一级
- 首页目录

HTTP头部清理

HTTP头部清理

出于安全原因，Envoy将根据请求传入的HTTP头（被"清理"的头部），来判断是内部还是外部请求。根据头部key进行清理，可能会导致添加，删除或修改头部。 最终，都是由`x-forwarded-for`标头，请求被认定是内部的还是外部的（请仔细阅读链接部分，因为关于Envoy填充头部过程复杂，并取决于use_remote_address设置）。

Envoy可能会对以下头部进行清理：

- 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-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

返回

- 上一级
- 首页目录

统计

统计

每个连接管理器都有一个以 `http.<stat_prefix>.` 为根的统计树。 统计如下：

名称	类型	描述
downstream_cx_total	Counter	连接总数
downstream_cx_ssl_total	Counter	TLS连接总数
downstream_cx_http1_total	Counter	HTTP / 1.1连接总数
downstream_cx_websocket_total	Counter	WebSocket连接总数
downstream_cx_http2_total	Counter	HTTP / 2连接总数
downstream_cx_destroy	Counter	连接关闭总数
downstream_cx_destroy_remote	Counter	因远端关闭而导致连接销毁总数
downstream_cx_destroy_local	Counter	由于本地关闭导致连接被摧毁的总数
downstream_cx_destroy_active_rq	Counter	连接被1个及以上活动请求销毁的总数
downstream_cx_destroy_local_active_rq	Counter	因1个及以上活动请求被本地销毁的连接总数
downstream_cx_destroy_remote_active_rq	Counter	因1个及以上活动请求被远端关闭而销毁的连接总数
downstream_cx_active	Gauge	活动的连接总数
downstream_cx_ssl_active	Gauge	活动TLS连接总数
downstream_cx_http1_active	Gauge	有效的HTTP / 1.1连接总数
downstream_cx_websocket_active	Gauge	有效的WebSocket连接总数
downstream_cx_http2_active	Gauge	有效的HTTP / 2连接总数
downstream_cx_protocol_error	Counter	协议错误的总数
downstream_cx_length_ms	Histogram	连接长度毫秒
downstream_cx_rx_bytes_total	Counter	接收的字节总数
downstream_cx_rx_bytes_buffered	Gauge	当前缓冲接收到的字节总数
downstream_cx_tx_bytes_total	Counter	发送的字节总数
downstream_cx_tx_bytes_buffered	Gauge	当前缓冲的发送字节总数
downstream_cx_drain_close	Counter	由逐出而造成总连接关闭总数

downstream_cx_idle_timeout	Counter	由于空闲超时而关闭的连接总数
downstream_flow_control_paused_reading_total	Counter	由于流量控制而被禁用的总读取次数
downstream_flow_control_resumed_reading_total	Counter	由于流量控制，在连接上启用的总读取次数
downstream_rq_total	Counter	总请求数
downstream_rq_http1_total	Counter	总HTTP / 1.1请求数
downstream_rq_http2_total	Counter	总HTTP / 2请求数
downstream_rq_active	Gauge	总活动请求
downstream_rq_response_before_rq_complete	Counter	在请求完成之前发送的响应总数
downstream_rq_rx_reset	Counter	收到的请求重置总数
downstream_rq_tx_reset	Counter	已发送请求重置总数
downstream_rq_non_relative_path	Counter	使用非相对HTTP路径的请求总数
downstream_rq_too_large	Counter	由于缓冲过大的Body而导致413请求总数
downstream_rq_2xx	Counter	回应2xx总数
downstream_rq_3xx	Counter	回应3xx总数
downstream_rq_4xx	Counter	回应4xx总数
downstream_rq_5xx	Counter	回应5xx总数
downstream_rq_ws_on_non_ws_route	Counter	因WebSocket升级而被拒绝的非WebSocket路由请求总数
downstream_rq_time	Histogram	请求时间，单位毫秒
rs_too_large	Counter	由于缓冲过大的Body而导致的响应错误总数

按代理客户统计

每个用户代理统计信息都以 `http.<stat_prefix>.user_agent.<user_agent>.` 为根。目前，Envoy与iOS（ios）和Android（android）的用户代理相匹配，并产生以下统计数据：

名称	类型	描述
downstream_cx_total	Counter	连接总数
downstream_cx_destroy_remote_active_rq	Counter	因1个及以上活动请求被远端关闭的连接总数
downstream_rq_total	Counter	请求总数

按监听端口统计

每个监听器统计信息的附加值均以 `listener.<address>.http.<stat_prefix>.` 为根。统计如下：

名称	类型	描述
downstream_rq_2xx	Counter	回应2xx总数
downstream_rq_3xx	Counter	回应3xx总数
downstream_rq_4xx	Counter	回应4xx总数
downstream_rq_5xx	Counter	回应5xx总数

返回

- 上一级
- 首页目录

运行时设置

运行时设置

HTTP连接管理器支持以下运行时设置：

- tracing.client_enabled

  如果设置了x-client-trace-id头部，请求将被强制跟踪的百分比。默认为100。

- tracing.global_enabled

  在所有其他检查（强制跟踪，采样等）生效之后，将被跟踪的请求百分比。默认为100。

- tracing.random_sampling

  被随机跟踪的请求万分比。浏览此处获取更多信息。此运行时间值设置范围为0-10000，默认为1000。因此，可以以0.01%的粒度设置跟踪采样。

返回

- 上一级
- 首页目录

路由发现服务

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独立地获取自己的路由配置。

- v1 API 参考
- v2 API 参考

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

返回

- 上一级
- 首页目录

HTTP过滤器

HTTP过滤器

缓存

- 统计

CORS过滤器

故障注入

- 配置
- 运行时配置
- 统计

DynamoDB

- 统计
- 运行时配置

gRPC HTTP/1.1 桥接

- 统计

gRPC-JSON 转码过滤器

gRPC-Web 过滤器

健康检查

速率限制

- 动作
- 统计
- 运行时配置

路由

- HTTP头部
- 统计
- 运行时配置

Lua

- 概述
- 当前支持的高阶特性
- 配置
- 脚本示例
- 流处理API
- 头部对象API

- 缓存API

返回

- 上一级
- 首页目录

缓存

缓存

缓冲区过滤器用于并等待并缓冲的完整请求。这在特殊场景下会很有用，如：保护一些应用程序，不必关心和处理部分请求，及高网络延迟。

- v1 API 参考
- v2 API 参考

统计

缓冲过滤器输出统计信息以 `http.<stat_prefix>.buffer.` 命名空间。 `stat_prefix` 来自拥有的HTTP连接管理器。

名称	类型	描述
rq_timeout	Counter	等待完整请求超时的请求总数

返回

- 上一级
- 首页目录

CORS过滤器

CORS过滤器

这是一个基于路由或虚拟主机的处理跨源共享资源请求(Cross-Origin Resource Sharing)的过滤器设置。标题相关的含义，请参阅下面的连接。

- 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

返回

- 上一级
- 首页目录

故障注入

故障注入

故障注入过滤器可用于测试微服务对不同形式故障的恢复能力。该过滤器可用于注入延迟和中止请求，并带有用户指定的错误代码，从而能够处理不同的故障情况，如服务故障，服务过载，高网络延迟，网络分区等。故障注入可限制在基于请求的（目的地）上游集群，以及特定的一组预定义的请求报头组。

故障注入的范围仅限于通过网络进行通信的应用程序，以及可观察到的范围。无法模拟本地主机上的CPU和磁盘故障。

目前，故障注入过滤器有以下限制：

- 中止请求的错误代码仅限于HTTP状态码
- 延迟被限制在一定的时间内

未来的版本将包括支持限制故障到特定的路由，注入gRPC和HTTP/2特定的错误代码和基于分布的持续时延。

配置

注意：故障注入过滤器必须在任何其他过滤器（包括路由器过滤器）之前插入。

- v1 API参考
- v2 API参考

运行时设置

HTTP故障注入过滤器支持以下全局运行时设置：

- fault.http.abort.abort_percent

  如果头部匹配，将被中止请求的百分比。在配置中默认使用 `abort_percent` 值。如果配置不包含 `abort` 项，则 `abort_percent` 默认为0。

- fault.http.abort.http_status

  将被用作请求的HTTP状态码，如果头部匹配，则请求将被中止。默认为配置中指定的 `http_status` 。如果配置不包含 `abort` 项，则 `http_status` 默认为0。

- fault.http.delay.fixed_delay_percent

  如果头部匹配，请求将被延迟的百分比。默认为配置中指定的 `delay_percent` ，否则为0。

- fault.http.delay.fixed_duration_ms

  延迟时间以毫秒为单位。如果未指定，则将使用配置中指定的`fixed_duration_ms`。如果在运行时和配置中缺少这个字段，则不会注入延迟。

请注意，在特定下游群集中，如果存在以下运行时配置值，则故障过滤器默认值会被覆盖。以下是下游指定的运行时配置值：

- `fault.http.<downstream-cluster>.abort.abort_percent`
- `fault.http.<downstream-cluster>.abort.http_status`
- `fault.http.<downstream-cluster>.delay.fixed_delay_percent`
- `fault.http.<downstream-cluster>.delay.fixed_duration_ms`

下游集群名称取自HTTP `x-envoy-downstream-service-cluster`头部。如果在运行系统中找不到，则默认使用全局运行时设置为缺省配置。

统计

故障过滤器输出统计信息命名空间为`http.<stat_prefix>.fault.`。`stat_prefix`来自所拥有的HTTP连接管理器。

名称	类型	描述
`delays_injected`	Counter	延迟请求总数
`aborts_injected`	Counter	已中止的请求总数
`<downstream-cluster>.delays_injected`	Counter	指定下游群集的延迟请求总数
`<downstream-cluster>.aborts_injected`	Counter	指定下游群集的中止请求总数

返回

- 上一级
- 首页目录

DynamoDB

DynamoDB

- DynamoDB架构概述
- v1 API 参考
- v2 API 参考

统计

DynamoDB过滤器输出统计信息命名空间为 `http.<stat_prefix>.dynamodb.` 。其中 `stat_prefix` 来自所拥有的HTTP连接管理器。

每个操作的统计信息可以在命名空间 `http.<stat_prefix>.dynamodb.operation.<operation_name>.` 找到。

名称	类型	描述
upstream_rq_total	Counter	Total number of requests with
upstream_rq_time	Histogram	Time spent on
upstream_rq_total_xxx	Counter	Total number of requests with per response code (503/2xx/etc)
upstream_rq_time_xxx	Histogram	Time spent on per response code (400/3xx/etc)

每个表的统计信息可以在命名空间 `http.<stat_prefix>.dynamodb.table.<table_name>.` 中找到。 DynamoDB的大部分操作都涉及单个表，但 `BatchGetItem` 和 `BatchWriteItem` 可以包含多个表，Envoy仅在所有批处理操作使用相同的表时才跟踪每个表的统计信息。

名称	类型	描述
upstream_rq_total	Counter	Total number of requests on table
upstream_rq_time	Histogram	Time spent on table
upstream_rq_total_xxx	Counter	Total number of requests on table per response code (503/2xx/etc)
upstream_rq_time_xxx	Histogram	Time spent on table per response code (400/3xx/etc)

免责声明：请注意，这是尚未广泛使用的预发行版本的Amazon DynamoDB功能。

每个分区和操作统计信息可以在命名空间 `http.<stat_prefix>.dynamodb.table.<table_name>.` 找到。对于批量操作，Envoy仅在所有操作中使用相同的表时跟踪每个分区和操作统计信息。

名称	类型	描述
capacity.	Counter	Total number of capacity for

其他详细统计信息：

- 对于4xx响应和部分批处理操作失败，在命名空间 `http.<stat_prefix>.dynamodb.error.<table_name>.` 中跟踪指定的表，失败的总次数。

名称	类型	描述
	Counter	Total number of specific
BatchFailureUnprocessedKeys	Counter	Total number of partial batch failures for a given

运行时设置
DynamoDB过滤器支持以下运行时设置：

- dynamodb.filter_enabled

  启用过滤器的请求百分比。默认是100%。

返回

- 上一级
- 首页目录

gRPC HTTP/1.1 桥接

gRPC HTTP/1.1 桥接

- gRPC 架构概述
- v1 API 参考
- v2 API 参考

这是一个简单的过滤器，可以将不支持gRPC响应的HTTP/1.1客户端桥接到兼容的gRPC服务器。它的工作原理如下：

- 发送请求时，过滤器会查看连接是否为HTTP/1.1，请求内容类型为 `application/grpc`。
- 如果是这样，当收到响应时，过滤器会缓存并等待预告片，然后检查 `grpc-status` 代码。如果不为零，则过滤器将HTTP响应代码切换为503。它还将 `grpc-status` 和 `grpc-message` 复制到响应头部中，以便客户端可以根据需要查看它们。
- 客户端应该发送翻译成以下伪首标的HTTP/1.1请求：
- `:method:` POST
- `:path:` `<gRPC-METHOD-NAME>`
- `content-type:` application/grpc
- Body应该是序列化的grpc body：
- 1个字节的零（未压缩）。
- 网络顺序4个字节的原始消息长度。
- 序列化的原始消息。
- 因为这个方案必须缓冲响应，以查找 `grpc-status`，所以它只能用于一元gRPC API。

更多信息：http://www.grpc.io/docs/guides/wire.html

此过滤器还收集所有gRPC传输请求的统计信息，即使这些请求是通过HTTP/2传输的正常gRPC请求。

统计

过滤器收集的统计信息命名空间为 `cluster.<route target cluster>.grpc.`。

名称	类型	描述
`<grpc service>.<grpc method>.success`	Counter	Total successful service/method calls
`<grpc service>.<grpc method>.failure`	Counter	Total failed service/method calls
`<grpc service>.<grpc method>.total`	Counter	Total service/method calls

返回

- 上一级

- 首页目录

gRPC-JSON 转码过滤器

gRPC-JSON 转码过滤器

- gRPC 架构概述
- v1 API 参考
- v2 API 参考

这是一个过滤器，它允许RESTful JSON API客户端通过HTTP向Envoy发送请求并代理到gRPC服务。根据自定义选项HTTP映射到gRPC服务。

返回

- 上一级
- 首页目录

gRPC-Web 过滤器

gRPC-Web 过滤器

- gRPC 架构概述
- v1 API 参考
- v2 API 参考

这是一个过滤器，通过协议定义可以将gRPC-Web客户端桥接到兼容的gRPC服务器。

返回

- 上一级
- 首页目录

健康检查

健康检查

- 健康检查架构概述
- v1 API 参考
- v2 API 参考

请注意：如果/healthcheck/fail管理端口被调用，过滤器将自动运行失败健康检查并设置x-envoy-immediate-health-check-fail头部。（通过/healthcheck/ok管理端口恢复此行为）。

返回

- 上一级
- 首页目录

速率限制

速率限制

- 速率限制架构概述
- v1 API 参考
- v2 API 参考

当请求的路由或虚拟主机有一个或多个符合过滤器设置的速率限制配置时，HTTP速率限制过滤器将调用速率限制服务。路由可以选择包含虚拟主机速率限制配置。可配置多个应用于请求。每个配置描述符都会导致被发送到速率限制服务。

如果速率限制服务被调用，并且任何响应超出限制的描述符，都将返回429响应。

组成操作

注意：本文是为v1 API编写的，但这些概念也适用于v2 API。它将在未来版本中使用v2 API重写。

路由或虚拟主机上的每个速率限制操作都需要一个描述符作为输入。若要创建更复杂的速率限制描述符，可由一组描述符按任意顺序进行组合操作。描述符将按照配置中指定操作的顺序进行填充。

例1

例如，要生成以下描述符：

```
("generic_key", "some_value")
("source_cluster", "from_cluster")
```

配置将是：

```
{
  "actions" : [
  {
    "type" : "generic_key",
    "descriptor_value" : "some_value"
  },
  {
    "type" : "source_cluster"
  }
  ]
}
```

例2

如果某个操作不附加描述符条目，则不会为该配置生成描述符。

对于以下配置：

```
{
  "actions" : [
  {
    "type" : "generic_key",
    "descriptor_value" : "some_value"
  },
  {
    "type" : "remote_address"
  },
  {
    "type" : "souce_cluster"
  }
  ]
}
```

如果请求没有设置x-forwarded-for，则不会生成描述符。

如果请求设置了x-forwarded-for，则会生成以下描述符：

```
("generic_key", "some_value")
("remote_address", "<trusted address from x-forwarded-for>")
("source_cluster", "from_cluster")
```

统计

缓冲区过滤器输出集群中的统计信息以 `cluster.<route target cluster>.ratelimit.` 为命名空间。429响应被发送到正常的群集动态HTTP统计。

名称	类型	描述
ok	Counter	低于限速请求的速率限制服务总数
error	Counter	连接速率限制服务的失败总数
over_limit	Counter	高于限速请求的速率限制服务总数

运行时设置

HTTP速率限制过滤器支持以下运行时设置：

- ratelimit.http_filter_enabled

  将调用速率限制服务的请求的百分比。默认为100。

- ratelimit.http_filter_enforcing

  将调用速率限制服务并执行决定的请求百分比。默认为100。这可以用来测试完全执行结果之前会发生什么。

- ratelimit.`<route_key>`.http_filter_enabled

  将调用速率限制配置中指定的给定`route_key`的速率限制服务请求的百分比。默认为100。

返回

- 上一级
- 首页目录

路由

路由

路由过滤器实现了HTTP转发。几乎所有Envoy部署的HTTP代理场景都有使用。过滤器的主要工作是遵循路由表中配置的指令。除了转发和重定向之外，过滤器还处理重试，统计等。

- v1 API 参考
- v2 API 参考

HTTP头部

路由器在出口/请求以及入口/响应路径上消费和设置各种HTTP头部。它们在本章节中有说明。

- 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

这是路由器期望请求完成的时间（以毫秒为单位）。Envoy设置这个HTTP头部，以便上游主机在接收请求后可以根据请求超时做出决定，例如提前退出。这是在内部请求上设置的，可以从x-envoy-upstream-rq-timeout-ms头部或路由超时中按顺序获取。

x-envoy-max-retries

如果有重试策略，Envoy将默认重试一次，除非明确指定。可以在路由重试配置或通过使用此头部显式设置重试次数。如果未配置重试策略，并且未指定x-envoy-retry-on或x-envoy-retry-grpc-on头，则Envoy将不会重试失败的请求。

关于Envoy如何重试的一些说明：

- 路由超时（通过x-envoy-upstream-rq-timeout-ms或路由配置设置）包括所有重试。 因此，如果

请求超时设置为3秒，并且第一次请求尝试需要2.7秒，则重试（包括退避）只有0.3s时间完成。这是设计来避免因重试/超时导致指数增长。

- Envoy使用完全抖动的指数退避算法进行重试，其基准时间为25ms。第一次重试将在0-24ms之间随机延迟，第二次在0-74ms之间，第三次在0-174ms之间，依此类推。
- 如果最大重试次数在头部和路由配置都有设置，则在会采用最大值用于请求的最大重试次数。

x-envoy-retry-on
在出口请求上设置此报头将导致Envoy尝试重试失败的请求（重试次数默认为1，可以通过x-envoy-max-retries头部或路由配置重试策略进行控制）。设置x-envoy-retry-on头的值表示重试策略。可以使用','分隔列表来指定一个或多个策略。支持的策略有：

- 5xx

  如果上游服务器响应任何5xx的响应代码，或者根本没有响应（断开/重置/读取超时），Envoy将尝试重试请求。（包括连接失败和拒绝流）

  注：当请求超过x-envoy-upstream-rq-timeout-ms（导致504误错代码）时，Envoy将不会重试。如果您想在个别请求上尝试超长时间的重试，请使用x-envoy-upstream-rq-per-try-timeout-ms。`x-envoy-upstream-rq-timeout-ms`是请求的外部时间限制，包括发生的任何重试。

- connect-failure

  如果由于上游服务器连接失败而导致请求失败（连接超时等），Envoy将尝试重试。（包含在5xx中）

  注：连接失败/超时是TCP级别，而不是请求级别。这不包括通过x-envoy-upstream-rq-timeout-ms或通过路由配置指定的上游请求超时。

- retriable-4xx

  如果上游服务器响应可回复的4xx响应代码，Envoy将尝试重试。目前，这个类别中唯一的响应代码是409。

  注：小心打开此重试类型。在某些情况下，409可能表明需要更新乐观锁版本。因此，调用者不应该重试并且需要读取，然后再尝试写入。如果在这种类型的情况下发生重试，则将总是以另一个409失败。

- refused-stream

  如果上游服务器使用`REFUSED_STREAM`错误代码重置流，Envoy将尝试重试。此重置类型表示请求可以安全地重试。（包含在5xx中）
  重试次数可以通过x-envoy-max-retries头或通过路由配置来控制。

请注意，重试策略也可以应用在路由级别。

默认情况下，Envoy不会执行重试，除非您已经按照上面的方式配置它们。

x-envoy-retry-grpc-on
在出口请求上设置此报头会导致Envoy尝试重试失败的请求（重试次数默认为1，可以通过x-envoy-max-retries头或路由配置重试策略进行控制）。gRPC重试当前仅支持响应头中的gRPC状态码。追踪者中的gRPC状态码不会触发重试逻辑。可以使用','分隔列表来指定一个或多个策略。支持的策略是：

- cancelled

  如果响应头中的gRPC状态码被“cancelled”（1），Envoy将尝试重试;

- deadline-exceeded

  如果响应头中的gRPC状态码是“deadline-exceeded”（4），Envoy将尝试重试;

- resource-exhausted

  如果响应头中的gRPC状态码是“resource-exhausted”（8），Envoy将尝试重试;

同样 `x-envoy-retry-grpc-on` 头的重试次数，可以通过x-envoy-max-retries头来控制

请注意，重试策略也可以应用在路由级别。

默认情况下，Envoy不会执行重试，除非您已经按照上面的方式配置它们。

x-envoy-upstream-alt-stat-name
在出口请求上设置这个头部会导致Envoy将上游响应代码/时间统计信息发送到双重统计树。这对于Envoy不知道的应用程序级别的情况下，可能很有用。详见统计输出文档。

x-envoy-upstream-canary
如果上游主机设置了这个头部，路由器将使用它来生成金丝雀（灰度升级）相关的统计信息。详见统计输出文档。

x-envoy-upstream-rq-timeout-alt-response
在出口请求上设置此头部会导致Envoy在请求超时的情况下设置一个204响应代码（而不是504）。头部实际的值被忽略; 只考虑它的存在。另请参见x-envoy-upstream-rq-timeout-ms。

x-envoy-upstream-rq-timeout-ms
在出口请求上设置此标头将覆盖Envoy的路由配置。超时时间必须以毫秒为单位指定。另请参阅x-envoy-upstream-rq-per-try-timeout-ms。

x-envoy-upstream-rq-per-try-timeout-ms

在出口请求上设置此头部将导致Envoy在路由请求上设置每次尝试超时。此超时必须<=全局路由超时（请参阅x-envoy-upstream-rq-timeout-ms），否则将被忽略。这允许调用者每次都尝试设置超时重试，同时保持合理的总超时。

x-envoy-upstream-service-time
包含上游主机处理请求所花费的时间（以毫秒为单位）。如果客户想要确定网络延迟与服务请求的时间做对比，这是有用的。这个头是在响应中设置的。

x-envoy-original-path
如果路由使用 `prefix_rewrite`，Envoy会将原始路径头部放在这个头部。这对日志记录和调试很有用。

x-envoy-immediate-health-check-fail
如果上游主机返回此头部（设置为任何值），则Envoy将立即判定上游主机的主动健康检查失败（如果已将群集配置为主动健康检查）。这是通过标准数据平面快速上报上游主机所发生故障，而无需等待下一个运行健康检查超时。上游主机也可以通过标准的主动健康检查再次变得健康。请参阅健康检查概述了解更多信息。

x-envoy-overloaded
如果这个头由上游主机设置了，Envoy不会重试。目前不关心头部的值，只关心是否存在。此外，如果处于维护模式或上游熔断而导致请求丢失，Envoy将在下游响应中设置此头部。

x-envoy-decorator-operation
如果此头部出现在入口请求上，则该头部的值将覆盖由跟踪机制生成的服务器span上的任何本地定义的操作名称（span name）。同样，如果在出口响应中出现此头部，其值将覆盖客户端范围上的任何本地定义的操作名称（span name）。

统计

路由器在集群命名空间中输出许多统计信息（取决于所选路由中指定的集群）。浏览此处获取更多信息。

路由器过滤器输出的统计信息命名空间为 `http.<stat_prefix>.`。其中 `stat_prefix` 来自拥有的HTTP连接管理器。

Name	Type	Description
no_route	Counter	因没有路由而导致404错误的总请求数
no_cluster	Counter	因目标集群不存在而导致404错误的总请求数
rq_redirect	Counter	导致重定向响应的总请求数
rq_total	Counter	总路由请求数

输出的虚拟集群统计信息名称空间为
`vhost.<virtual host name>.vcluster.<virtual cluster name>.`，并包含以下统计信

息：

Name	Type	Description
upstreamrq	Counter	HTTP响应代码汇总（例如，2xx，3xx等）
upstreamrq	Counter	特定的HTTP响应码（例如：201、302等）
upstream_rq_time	Histogram	请求时间，单位毫秒

运行时设置

路由器过滤器支持以下运行时设置：

- upstream.base_retry_backoff_ms

  指数重试退避时间粒度。默认为25ms。浏览此处获取更多信息。

- upstream.maintenance_mode.

  将导致立即返回503响应请求的百分比。这会覆盖针对 `<cluster name>` 集群的任何路由请求行为。这可以用于卸载，故障注入等。默认为禁用。

- upstream.use_retry

  有资格使用重试的请求百分比。在任何其他重试配置之前检查此配置，如果需要，可用于禁用所有Envoy的重试。

返回

- 上一级
- 首页目录

Lua

Lua

注意：Lua脚本HTTP过滤器是实验性的。在生产中使用需要您自担风险。它正在被公布，以便对暴露的API进行初步反馈，并进行进一步的开发，测试和验证。当我们认为Lua过滤器已经受到足够的API稳定性测试，通常称其为生产准备就绪时，该警告将被移除。

概述

HTTP Lua过滤器允许在请求和响应流程中运行Lua脚本。在运行时使用LuaJIT。正因为如此，支持的Lua版本大部分是5.1，具有一些5.2的特性。有关更多详细信息，请参阅LuaJIT文档。

该过滤器仅支持在配置中直接加载Lua代码。如果需要本地文件系统代码，则可以使用简单的内联脚本从本地环境加载代码的其余部分。

过滤器和支持Lua是基于如下高级设计：

- 所有Lua环境都是每个工作者线程。这意味着没有真正的全局数据。任何在加载时创建和填充的全局变量在工作线程之间相互隔离。真正的全局支持可能会在未来通过API添加。
- 所有脚本都以协程运行。这意味着即使它们可能执行复杂的异步任务，它们也是以同步样式编写的。这使得脚本更容易编写。所有网络/异步处理由Envoy通过一组API执行。Envoy将酌情切换(yield)脚本，并在异步任务完成时恢复脚本。
- 不要在脚本中执行阻塞操作。因为Envoy API会用于所有IO，以及性能相关的重要操作。

目前支持高级功能

注：预计随着Lua过滤器在生产中的应用，以下列表将随着时间的推移而扩大范围。这些API一直保持很小的用意。为了使脚本编写非常简单和安全。若存在非常复杂或更高性能的场景，请使用本地C++过滤器API。

- 在传输的请求，响应流或两者同时，提供头部，正文和尾部的检查；
- 对头部和尾部进行修改；
- 阻止或者缓冲完整的请求/响应正文，进行检查；
- 对上游主机执行异步HTTP调用。这样可以在缓冲正文数据的同时执行处理，以便当调用完成时，可以修改上行头部；
- 直接执行响应并跳过接下来的迭代过滤器。例如，一个脚本可以创建一个上游HTTP认证调用，然后直接用403响应代码进行响应。

配置

- v1 API 参考
- v2 API 参考

脚本示例

本节提供了一些Lua脚本的一些具体的例子，作为一个更简单介绍和快速入门。有关支持的API的更多详细信息，请参阅流处理API。

```
-- Called on the request path.
function envoy_on_request(request_handle)
  -- Wait for the entire request body and add a request header with the body size.
  request_handle:headers():add("request_body_size", request_handle:body():length())
end

-- Called on the response path.
function envoy_on_response(response_handle)
  -- Wait for the entire response body and a response header with the the body size.
  response_handle:headers():add("response_body_size", response_handle:body():length())
  -- Remove a response header named 'foo'
  response_handle:headers():remove("foo")
end
```

```
function envoy_on_request(request_handle)
  -- Make an HTTP call to an upstream host with the following headers, body, and timeout.
  local headers, body = request_handle:httpCall(
  "lua_cluster",
  {
    [":method"] = "POST",
    [":path"] = "/",
    [":authority"] = "lua_cluster"
  },
  "hello world",
  5000)

  -- Add information from the HTTP call into the headers that are about to be sent to the next
  -- filter in the filter chain.
  request_handle:headers():add("upstream_foo", headers["foo"])
  request_handle:headers():add("upstream_body_size", #body)
end
```

```
function envoy_on_request(request_handle)
  -- Make an HTTP call.
  local headers, body = request_handle:httpCall(
  "lua_cluster",
  {
    [":method"] = "POST",
    [":path"] = "/",
    [":authority"] = "lua_cluster"
  },
  "hello world",
  5000)

  -- Response directly and set a header from the HTTP call. No further filter iteration
  -- occurs.
  request_handle:respond(
    {[":status"] = "403",
     ["upstream_foo"] = headers["foo"]},
    "nope")
end
```

流处理API

当Envoy在配置中加载脚本时，它会查找脚本中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两个。在请求路径中，Envoy将运行 `envoy_on_request` 函数作为一个协程，传递一个API句柄。在响应路径中，Envoy将运行 `envoy_on_response` 作为协程，传递一个API句柄。

注意：与Envoy的所有交互，都是通过传输流来实现的。流句柄不应该被保存到任何全局变量，不应该在协程之外使用。如果句柄使用不当，Envoy将会失败。

支持以下流处理方法：

- headers()

  ```
  headers = handle:headers()
  ```

  返回流的头部对象。只要头部还没有被发送到下一个过滤器的头部链中，该头部就可以被修改。例如，可以在 `httpCall()` 之后或在 `body()` 调用返回之后修改它们。如果在任何其他情况下修改了头部，脚本将会失败。

返回一个头部对象。

- body()

  ```
  body = handle:body()
  ```

  返回流的主体（body）。这个调用将使的Envoy切换（yield）脚本，直到整个主体被缓冲。请注意，所有缓冲必须遵守流量控制政策。Envoy不会缓冲超过连接管理器所允许的数据范围。

返回一个缓冲对象。

- bodyChunks()

  ```
  iterator = handle:bodyChunks()
  ```

  返回一个迭代器，它可以用来迭代所有接收到的正文数据块。Envoy将在处理大块数据时，切换(yield)脚本，但不会缓冲它们。这可以用来检查数据流。

```
for chunk in request_handle:bodyChunks() do
request_handle:log(0, chunk:length())
end
```

每次迭代器返回都是一个缓冲对象。

- trailers()

```
trailers = handle:trailers()
```

返回流的尾部。如果没有尾部，可能会返回零。尾部在发送到下一个过滤器之前可能会被修改。

返回一个头标对象。

- *log()**

```
handle:logTrace(message)
handle:logDebug(message)
handle:logInfo(message)
handle:logWarn(message)
handle:logErr(message)
handle:logCritical(message)
```

使用Envoy的应用程序日志记录消息。参数 `message` 是需要记录的字符串。

- httpCall()

```
headers, body = handle:httpCall(cluster, headers, body, timeout)
```

对上游主机进行HTTP调用。Envoy将切换脚本，直到调用完成或有错误。 `cluster` 是对应到群集管理器配置的群集字符串。 `headers` 是要发送的 `key/value` 键值对的列表。请注意，必须设置 `:method` ， `:path` 和 `:authority` 头部。 `body` 是可选字符串，需要发送的数据主体。 `timeout` 是一个整数，指定以毫秒为单位的调用超时。

返回是响应的 `headers` ，以及其 `body` 字符串。如果没有主体可能是 `null` 。

- respond()

```
handle:respond(headers, body)
```

立即作出响应，不要进行进一步的过滤器迭代。此调用仅在请求流中有效。此外，只有在请求头还没有传递给后续过滤器的情况下，响应才是可能的。意思是，下面的Lua代码是错误的：

```
function envoy_on_request(request_handle)
for chunk in request_handle:bodyChunks() do
  request_handle:respond(
    {[":status"] = "100"},
    "nope")
end
end
```

`headers` 是要发送的键值对的列表。请注意，必须设置 `:status` 头部。 `body` 是一个字符串，作为可选的响应主体，可能是 `nil` 。

头部对象API

- add()

```
headers:add(key, value)
```

为头部对象添加一个头部。 `key` 是提供头部键的字符串。 `value` 是一个提供头部值的字符串。

- get()

```
headers:get(key)
```

获取头部对象头部值，参数 `key` 为所对应的头部键。返回一个字符串作为头部值，如果没有这样的头部则返回 `nil` 。

- __pairs()

```
for key, value in pairs(headers) do
end
```

遍历每个头部键。返回的 `key` 是头部键的字符串。返回的 `value` 是对应的头部值字符串。

注意：在当前的实现中，在迭代期间不能修改头部。 另外，如果需要在迭代之后修改头部，则必须完成迭代。意味着，不要使用break或其他机制提前退出循环。但这可能会在未来版本中解除这个限制。

- remove()

```
headers:remove(key)
```

删除头部键值对。入参 `key` 对应的头部键值将会被删除。

缓存API

- length()

```
size = buffer:length()
```

获取缓冲区的大小（以字节为单位）。返回一个整数。

- getBytes()

```
buffer:getBytes(index, length)
```

从缓冲区获取字节。默认情况下，Envoy不会将所有缓冲区字节复制到Lua中。这将导致一个缓冲区段被复制。`index`是一个整数，并提供要复制的缓冲区起始索引。`length`是一个整数并提供要复制的缓冲区长度。`index + length`必须小于缓冲区长度。

返回

- 上一级
- 首页目录

集群管理

集群管理

- 集群管理架构概述
- v1 API 参考
- v2 API 参考

统计

- 概述
- 健康检查统计
- 离群检测统计
- 动态HTTP统计
- 动态HTTP交叉树统计
- 按服务区动态HTTP统计
- 负载均衡统计
- 负载均衡子集统计

运行时设置

- 主动健康检查
- 离群异常检测
- 核心
- 区域负载均衡
- 熔断

集群发现服务

- 统计

健康检查

- TCP健康检查

熔断

- 运行时配置

返回

- 上一级
- 首页目录

统计

统计

概述

集群管理器的统计树根为`cluster_manager.`用下面的统计描述。任何`:`字符在统计名称中的被替换为`_`。

Name	Type	Description
cluster_added	Counter	总群集添加（通过静态配置或CDS）
cluster_modified	Counter	总群集修改（通过CDS）
cluster_removed	Counter	总群集删除（通过CDS）
total_clusters	Gauge	当前加载的群集数量

每个群集都有一个以`cluster.<name>.`为根的统计树。统计如下：

Name	Type	Description
upstream_cx_total	Counter	总连接数
upstream_cx_active	Gauge	总活动连接数
upstream_cx_http1_total	Counter	总HTTP/1.1连接数
upstream_cx_http2_total	Counter	总HTTP/2连接数
upstream_cx_connect_fail	Counter	总连接失败
upstream_cx_connect_timeout	Counter	总连接超时
upstream_cx_connect_attempts_exceeded	Counter	总连续的连接失败超过配置的连接尝试
upstream_cx_overflow	Counter	集群连接断路器溢出的总次数
upstream_cx_connect_ms	Histogram	连接建立毫秒
upstream_cx_length_ms	Histogram	连接长度毫秒
upstream_cx_destroy	Counter	总毁坏的连接

upstream_cx_destroy_local	Counter	总连接在本地被销毁
upstream_cx_destroy_remote	Counter	总连接被远程销毁
upstream_cx_destroy_with_active_rq	Counter	总共连接被1个活动请求销毁
upstream_cx_destroy_local_with_active_rq	Counter	总共有1个活动请求在本地销毁
upstream_cx_destroy_remote_with_active_rq	Counter	总共连接被1个活动请求远程销毁
upstream_cx_close_notify	Counter	总连接通过HTTP/1.1连接关闭标头或HTTP/2 GOAWAY关闭
upstream_cx_rx_bytes_total	Counter	收到的连接字节总数
upstream_cx_rx_bytes_buffered	Gauge	接收到当前缓冲的连接字节
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	协议错误的总连接数
upstream_cx_max_requests	Counter	由于最大请求而关闭总连接数
upstream_cx_none_healthy	Counter	由于没有健康的主机，没有建立连接总数
upstream_rq_total	Counter	总请求
upstream_rq_active	Gauge	总活动请求
upstream_rq_pending_total	Counter	挂起连接池连接的请求总数
upstream_rq_pending_overflow	Counter	连接池断路溢出并失败的请求总数
upstream_rq_pending_failure_eject	Counter	由于连接池连接失败而导致失败的总请求数
upstream_rq_pending_active	Gauge	等待连接池连接的活动请求总数
upstream_rq_cancelled	Counter	获取连接池连接之前被取消的总请求数
upstream_rq_maintenance_mode	Counter	由于维护模式而导致立即返回503错误的总请求
upstream_rq_timeout	Counter	超时等待响应的请求总数
upstream_rq_per_try_timeout	Counter	每次尝试超时的总请求数
upstream_rq_rx_reset	Counter	在远端重置的总请求数
upstream_rq_tx_reset	Counter	在本地重置的总请求数
upstream_rq_retry	Counter	请求重试次数
upstream_rq_retry_success	Counter	请求重试成功次数
upstream_rq_retry_overflow	Counter	由于熔断，未重试的总请求数

upstream_flow_control_paused_reading_total	Counter	流量控制，从上游暂停读取的总次数
upstream_flow_control_resumed_reading_total	Counter	流量控制，从上游恢复读取的总次数
upstream_flow_control_backed_up_total	Counter	上游连接备份、暂停下游读取的总次数
upstream_flow_control_drained_total	Counter	上游连接逐出、恢复下游读取的总次数
membership_change	Counter	总集群成员变化
membership_healthy	Gauge	当前群集健康成员总数（包括健康检查和异常值检测）
membership_total	Gauge	当前的集群成员总数
retry_or_shadow_abandoned	Counter	由于缓冲区限制，忽略或重试、被取消的总次数
config_reload	Counter	由于不同的配置，导致配置重新加载的API调用次数
update_attempt	Counter	总集群成员更新尝试次数
update_success	Counter	总集群成员更新成功次数
update_failure	Counter	总集群成员更新失败次数
version	Gauge	来自上次API调用加载成功的内容哈希
max_host_weight	Gauge	群集中所有主机的最大权重
bind_errors	Counter	将套接字绑定到配置的源地址错误总数

健康检查统计

如果配置了健康检查，那么集群会有一个以`cluster.<name>.health_check.`为根的统计树，统计如下：

Name	Type	Description
attempt	Counter	健康检查的次数
success	Counter	健康检查成功的次数
failure	Counter	执行健康检查快速失败的次数，（例如：HTTP 503，以及网络故障）
passive_failure	Counter	因被动事件导致的健康检查失败的次数（例如：`x-envoy-immediate-health-check-fail`）
network_failure	Counter	由于网络错误导致的健康检查失败次数
verify_cluster	Counter	尝试集群名称验证的健康检查的数量
healthy	Gauge	健康成员的数量

离群检测统计

如果为群集配置了离群异常检测，则统计信息将以 `cluster.<name>.outlier_detection.` 为根。并包含以下内容：

Name	Type	Description
ejections_enforced_total	Counter	由于任何异常类型导致的强制逐出的数量
ejections_active	Gauge	当前被逐出主机的数量
ejections_overflow	Counter	因达到最大逐出而中止次数百分占比
ejections_enforced_consecutive_5xx	Counter	执行的连续5xx逐出次数
ejections_detected_consecutive_5xx	Counter	检测到的连续5xx逐出次数（即使未被强制执行）
ejections_enforced_success_rate	Counter	执行成功率异常值逐出的次数
ejections_detected_success_rate	Counter	检测到的成功率异常值逐出次数（即使未执行）
ejections_enforced_consecutive_gateway_failure	Counter	执行的连续网关故障逐出次数
ejections_detected_consecutive_gateway_failure	Counter	检测到的连续网关故障逐出次数（即使未被强制执行）
ejections_total	Counter	已过时：由于任何异常值类型（即使未强制执行）
ejections_consecutive_5xx	Counter	已过时：连续的5xx被逐出次数（即使未被强制执行）

动态HTTP统计

若启用了HTTP，则动态HTTP响应统计信息也可用。这些由各种内部系统，以及各种路由、速率限制之类的过滤器构成的统计。以 `cluster.<name>.` 为根，并包含以下统计信息：

Name	Type	Description
upstream_rq_<*xx>	Counter	HTTP响应码汇总统计（例如：2xx，3xx等）
upstream_rq_<*>	Counter	具体的HTTP响应码统计（例如：201、302等）
upstream_rq_time	Histogram	请求时间，单位毫秒
canary.upstream_rq_<*xx>	Counter	上游灰度发布期间的HTTP响应码统计
canary.upstream_rq_<*>	Counter	上游灰度发布期间具体的HTTP响应码统计
canary.upstream_rq_time	Histogram	上游灰度发布期间请求时间毫秒
internal.upstream_rq_<*xx>	Counter	来自内部的HTTP响应码统计
internal.upstream_rq_<*>	Counter	来自内部具体的HTTP响应码统计
internal.upstream_rq_time	Histogram	来自内部请求时间，单位毫秒

external.upstream_rq_<*xx>	Counter	来自外部HTTP响应码汇总统计
external.upstream_rq_<*>	Counter	来自外部具体的HTTP响应码统计
external.upstream_rq_time	Histogram	来自外部请求时间，单位毫秒

动态HTTP交叉树统计

如果配置了交叉树统计信息，它们将以 `cluster.<name>.<alt name>.` 为命名空间。生成的统计信息与上面的动态HTTP统计信息相同。

按服务区动态HTTP统计

如果服务区可用于本地服务（通过 `--service-zone` ）和上游群集，则Envoy将以 `cluster.<name>.zone.<from_zone>.<to_zone>` 为命名空间。统计信息如下：

Name	Type	Description
upstreamrq	Counter	HTTP响应码汇总统计（例如：2xx，3xx等）
upstreamrq	Counter	具体的HTTP响应码统计（例如：201、302等）
upstream_rq_time	Histogram	请求时间，单位毫秒

负载均衡统计

监控负载均衡决策的统计信息。统计信息以 `cluster.<name>.` 为根，并包含以下统计信息：

Name	Type	Description
lb_healthy_panic	Counter	恐慌模式下承载负载均衡请求的总数
lb_zone_cluster_too_small	Counter	由于上游群集过小，无区域感知路由决策次数
lb_zone_routing_all_directly	Counter	所有请求直接发送到同一个区域决策次数
lb_zone_routing_sampled	Counter	发送一些请求到同一个区域决策次数
lb_zone_routing_cross_zone	Counter	区域感知路由模式，但必须发送交叉区域的次数
lb_local_cluster_not_ok	Counter	本地主机集未设置，或者是本地群集处于混乱模式
lb_zone_number_differs	Counter	本地和上游群集中的区域数目不同的次数

负载均衡子集统计

监控以 `<arch_overview_load_balancer_subsets>` 描述符负载均衡器子集的统计信息，统计信息以 `cluster.<name>.` 根并包含以下统计信息：

Name	Type	Description
lb_subsets_active	Gauge	当前可用子集的数量
lb_subsets_created	Counter	创建的子集数量
lb_subsets_removed	Counter	由于没有主机而被删除的子集数量
lb_subsets_selected	Counter	选择任何子集进行负载平衡的次数
lb_subsets_fallback	Counter	回退策略被调用的次数

返回

- 上一级
- 首页目录

运行时设置

运行时配置

上游集群支持以下运行时配置：

主动健康检查

- health_check.min_interval

  健康检查间隔的最小值。默认值为0。运行状况检查间隔将介于 `min_interval` 和 `max_interval` 之间。

- health_check.max_interval

  健康检查间隔的最大值。默认值是MAX_INT。健康检查间隔将在 `min_interval` 和 `max_interval` 之间。

- health_check.verify_cluster

  健康检查过滤器将远程服务集群的写入响应中针对预期的上游服务验证健康检查请求的百分比。

离群异常检测

有关离群异常检测的更多信息，请参见异常值检测架构概述。异常值检测支持的运行时参数与静态配置参数相同，即：

- outlier_detection.consecutive_5xx

  用于异常值检测的consecutive_5XX设置

- outlier_detection.consecutive_gateway_failure

  用于异常值检测的connected_gateway_failure设置

- outlier_detection.interval_ms

  在异常值检测中的interval_ms设置

- outlier_detection.base_ejection_time_ms

  基于异常值检测的base_ejection_time_ms设置

- outlier_detection.max_ejection_percent

异常值检测中的max_ejection_percent设置

- outlier_detection.enforcing_consecutive_5xx

  在异常值检测中执行enforcing_consecutive_5xx设置

- outlier_detection.enforcing_consecutive_gateway_failure

  在异常值检测中执行enforcing_consecutive_gateway_failure设置

- outlier_detection.enforcing_success_rate

  在异常值检测中执行enforcing_success_rate设置

- outlier_detection.success_rate_minimum_hosts

  异常值检测中的success_rate_minimum_hosts设置

- outlier_detection.success_rate_request_volume

  异常值检测中的success_rate_request_volume设置

- outlier_detection.success_rate_stdev_factor

  异常值检测中的success_rate_stdev_factor设置

核心

- upstream.healthy_panic_threshold

  恐慌阈值百分比设置。默认为50%。

- upstream.use_http2

  配置群集是否使用http2功能。设置为0即禁用HTTP/2。默认为启用。

- upstream.weight_enabled

  二进制开关，用于打开或关闭加权负载均衡。如果设置为非0，则启用加权负载均衡。默认为启用。

区域感知负载均衡

- upstream.zone_routing.enabled

将被路由到相同的上游区域的请求的百分比。默认为100%的请求。

- upstream.zone_routing.min_cluster_size

  可以尝试区域感知路由的上游群集的最小大小。默认值为6。如果上游群集大小小于 `min_cluster_size` ，则区域感知路由将不被执行。

熔断

- circuit_breakers.

  断路器最大连接数设置

- circuit_breakers.

  断路器最大待处理请求设置

- circuit_breakers.

  断路器最大请求数量设置

- circuit_breakers.

  断路器最大重试次数设置

返回

- 上一级
- 首页目录

集群发现服务

集群发现服务

群集发现服务（CDS）是一个可选的API，Envoy将调用该API来动态获取群集管理器成员。 Envoy还将根据API响应协调集群管理，根据需要完成添加，修改或删除已知的群集。

- v1 CDS API
- v2 CDS API

统计

CDS的统计树以 `cluster_manager.cds.` 为根，统计如下：

Name	Type	Description
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

返回

- 上一级
- 首页目录

健康检查

健康检查

- 健康检查架构概述
- 如果为集群配置了健康检查，则会触发其他统计信息。详见这里文档。
- v1 API文档
- v2 API文档

TCP健康检查

注意：本文是为v1 API编写的，但这些概念也适用于v2 API。它将在未来版本中使用v2 API重写。

执行的匹配类型如下（这是MongoDB运行健康检查请求和响应）：

```
 {
   "send": [
     {"binary": "39000000"},
     {"binary": "EEEEEEEE"},
     {"binary": "00000000"},
     {"binary": "d4070000"},
     {"binary": "00000000"},
     {"binary": "746573742e"},
     {"binary": "24636d6400"},
     {"binary": "00000000"},
     {"binary": "FFFFFFFF"},
     {"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"}
    ]
}
```

在每个健康检查周期中，所有“发送”字节都会发送到目标服务器。每个二进制块可以是任意长度的，并且在发送时只是连接在一起。（分成多个块用于可读性）。

在检查响应时，执行“模糊”匹配，使得必须找到每个二进制块，并且按照指定的顺序，但不必是连续的。因此，在上面的例子中，在“EEEEEEEE”和“01000000”之间的响应中可以插入“FFFFFFFF”，并

且检查仍然通过。这样做是为了支持在响应中插入非确定性数据（如时间）的协议。

若需要一个更复杂的健康检查模式，如发送/接收/发送/接收目前还不支持。

如果“接收”是一个空数组，则Envoy将执行“仅连接”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。为每个健康检查周期创建一个新的连接。

返回

- 上一级
- 首页目录

熔断

熔断

- 熔断架构概述
- v1 API 文档
- v2 API 文档

运行时配置

所有的断路设置都可以根据集群名称定义所有运行时配置。他们遵循以下命名规则 `circuit_breakers.<cluster_name>.<priority>.<setting>` ，其中 `cluster_name` 表示每个集群的名称，可以在Envoy配置文件中进行设置。也可用的运行时配置覆盖Envoy配置文件中设置的值。

返回

- 上一级
- 首页目录

访问日志

访问日志

配置

访问日志是HTTP连接管理器或TCP代理配置的一部分。

- v1 API 参考
- v2 API 参考

格式规则

访问日志格式字符串包含命令操作符或解释为普通字符串的其他字符。访问日志格式化程序不会做任何换行分隔符（如：“\n”），因此必须将其指定为格式字符串的一部分。请参阅示例的默认格式。请注意，访问日志行将为每个未设置/空值包含一个“-”字符。

访问日志的有些字段使用相同的格式字符串（如：HTTP和TCP）。有些字段的含义可能略有不同，具体取决于它是什么类型的日志。注意差异。

支持以下命令操作符：

- %START_TIME%

  HTTP

  请求开始时间，包括毫秒

  TCP

  下游连接开始时间，包括毫秒

- %BYTES_RECEIVED%

  HTTP

  收到主体字节

  TCP

  下行流连接时收到的字节

- %PROTOCOL%

HTTP

协议，目前是HTTP/1.1或HTTP/2

TCP

未实现（”-”）

- %RESPONSE_CODE%

  HTTP

  HTTP响应代码。请注意，响应代码“0”表示服务器从未发送响应的开始。这通常意味着（下游）客户端连接断开了。

  TCP

  未实现（”-”）

- %BYTES_SENT%

  HTTP

  正文发送的字节

  TCP

  在连接上发送到下行流字节

- %DURATION%

  HTTP

  请求从开始时间到最后一个字节输出的总持续时间（以毫秒为单位）

  TCP

  下游连接的总持续时间（以毫秒为单位）

- %RESPONSE_FLAGS%

  有关响应或连接的其他详细信息（如果有）。对于TCP连接，说明中提到的响应码不适用。可能的值

是：

HTTP and TCP

```
UH：除503响应码外，上游群集中没有健康的上游主机
UF：除503响应代码外，上游连接失败
UO：除503响应码外，上行溢出（断路）
NR：除404响应码外，没有为的请求配置可用路由
```

HTTP only

```
LH：除503响应码之外，本地服务失败的健康检查请求
UT：除了504响应代码之外，上游请求超时
LR：除503响应码外，连接本地复位
UR：除503响应码外，还有上行远程复位
UC：除503响应码之外的上游连接终止
DI：请求处理延迟了故障注入指定的时间
FI：请求被中止，并通过故障注入指定响应代码
RL：该请求除了429响应代码之外，还由HTTP速率限制过滤器进行本地速率限制
```

- %UPSTREAM_HOST%

  上游主机URL（例如：TCP连接或者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

- %DOWNSTREAM_ADDRESS%

  下游连接的远端地址

- %REQ(X?Y):Z%

  HTTP

  一个HTTP请求头部，其中X是主HTTP头部，Y是替代头部，而Z是一个可选参数，表示截取长度为Z个字符的字符串。该值取自名为X的HTTP请求头部，如果未设置，则使用请求头部Y. 如果不存在，则使用“-”符号替代

TCP

未实现（”-”）

- %RESP(X?Y):Z%

  HTTP

  与 `%REQ(X?Y):Z%` 相同，但是来自HTTP响应头

  TCP

  未实现（”-”）

默认格式

如果未指定自定义格式，Envoy将使用以下默认格式：

```
[%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
```

默认访问日志格式的示例：

```
[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"
```

返回

- 上一级
- 首页目录

限速服务

速率限制服务

速率限制服务配置指定，Envoy在需要作出全局速率限制决策时，与之交互的全局限速服务。如果没有配置速率限制服务，则会使用“null”服务，如果调用它将总是返回OK。

- v1 API 参考
- v2 API 参考

gRPC IDL

速率限制服务支持/source/common/ratelimit/ratelimit.proto中指定的gRPC IDL。有关更多关于API工作的信息，请参阅IDL文档。请参阅Lyft的参考实现。

返回

- 上一级
- 首页目录

运行时配置

运行时配置

运行时配置是指包含本地文件的可重载配置。如果运行时未配置，则默认使用“null”，该程序会使用代码中内置的缺省值。

- v1 API 参考
- v2 API 参考

文件系统层

在配置参考的各个部分描述了可用的运行时配置。例如，以下是上游群集的运行时配置。

假定文件夹 `/srv/runtime/v1` 指向的是存放全局运行时配置的文件路径。以下是运行时的典型配置设置：

- symlink_root(根目录的符号链接): `/srv/runtime/current`
- subdirectory(子目录): `envoy`
- override_subdirectory(覆盖子目录): `envoy_override`

其中 `/srv/runtime/current` 是链接到 `/srv/runtime/v1` 目录的符号链接。

运行时配置中的每个“.”表示新的目录层级，以 `symlink_root + subdirectory` 为目录根。例如，`health_check.min_interval` 键展开之后具有以下完整文件路径（使用符号链接）：

```
/srv/runtime/current/envoy/health_check/min_interval
```

路径的末端部分是文件。文件内容为运行时配置值。在从文件中读取数值时，空格和换行将被忽略。

目录 `override_subdirectory` 与 `--service-cluster` 命令行选项配合一起使用。假设 `--service-cluster` 已被设置为 `my-cluster`。Envoy将优先在以下完整路径中查找 `health_check.min_interval` 配置项：

```
/srv/runtime/current/envoy_override/my-cluster/health_check/min_interval
```

如果找到，该值将覆盖在主路径中查找到的任何值。这允许用户在全局默认值之外，自定义单个群集的运行时配置值。

注释

以＃开头的行被视为注释。合理的注释可以帮助理解所提供值的背景/原因。另外在一个空文件增加注释也是有用的，预留文件的部署，以便在需要的时候可以使用。

通过符号链接交换更新运行值

有两个步骤来更新任何运行时值。首先，创建整个全新运行时配置或者拷贝/更新所需的运行时配置。其次，使用以下命令或者等效命令，将旧的符号链接根自动交换到新的配置目录下：

```
/srv/runtime:~$ ln -s /srv/runtime/v2 new && mv -Tf new current
```

关于如何部署文件系统数据，以及相关垃圾回收等问题，超出了本文档的范围。

统计

文件系统在运行时会收集一些统计信息，以 `runtime.` 命名空间。

名称	类型	描述
load_error	Counter	因错误而导致加载尝试总数
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	当前加载的键值数量

返回

- 上一级
- 首页目录

路由表检查工具

路由表检查工具

注意：以下配置仅适用于路由表检查工具，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，可用于验证给定配置文件的Envoy路由。

路由表检查工具的需要相应的输入，并且检查之后返回的路由配置是否符合预期。该工具可用于检查群集名称，虚拟群集名称，虚拟主机名称，手动路径重写，手动主机重写，路径重定向和标题字段匹配。可以扩展添加其他测试用例。有关安装工具和示例工具输入/输出的详细信息，请参见安装。

路由表检查工具配置由一个json测试对象数组组成。每个测试对象由三部分组成。

- test_name

  该字段指定每个测试对象的名称。

- input

  输入值字段是指定要传递给路由器的参数。示例：输入字段包括 `:authority` ， `:path` 和 `:method` 头部字段。其中 `:authority` 和 `:path` 字段是指定发送到路由器的url，并且是必填字段。所有其他字段是可选的。

- validate

  检验字段是指定要检查的期望值和测试用例。至少需要一个测试用例。

一个简单的json格式的工具配置，以及一个测试用例，写法如下。测试与“instant-server”的群集匹配。

```
[
  {
    "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) 测试对象名称

input

(required, object) 做为路由器输入，并测试返回的路由

- :authority

  (required, string) 权威的URL。此值与`:path`参数一起定义，匹配需要的`url`路径。示例值是“api.lyft.com”。

- :path

(required, string) URL路径。示例值是“/foo”。

- :method

  (optional, string) 请求方法。如果未指定，则默认方法是 `GET`。选项是 `GET`，`PUT` 或 `POST`。

- internal

  (optional, boolean) 是否将 `x-envoy-internal` 设置为“true”的标志。如果未指定，或者如果 `internal` 等于 `false`，则不会设置 `x-envoy-internal`。

- random_value

  (optional, integer) 用于标识加权群集选择的目标的整数。 `random_value` 的默认值是0。

- ssl

  (optional, boolean) 确定是否将 `x-forwarded-proto` 设置为 `https` 或 `http` 的标志。通过将 `x-forwarded-proto` 设置为给定的协议，该工具能够模拟通过 `http` 或 `https` 发出请求的客户端行为。默认情况是 `false`，与之对应 `x-forwarded-proto` 设置为 `http`。

- additional_headers

  (optional, array) 需要额外添加头部为路由器的输入。其他配置选项“:authority”，“:path”，“:method”，“x-forwarded-proto”和“x-envoy-internal”字段，不应在此设置。

  - field

    (required, string) 被添加头部的名称

  - value

    (required, string) 被添加头部的值

validate

(required, object) 校验对象是指定要匹配的路由返回的参数。至少必须指定一个测试参数。使用“”（空字符串）表示没有返回值。例如，若测试不需要集群匹配，请使用{“cluster_name”：“”}。

- cluster_name

(optional, string) 匹配的集群名称

- virutal_cluster_name

  (optional, string) 匹配的虚拟集群名称

- virtual_host_name

  (optional, string) 匹配的虚拟主机名称

- host_rewrite

  (optional, string) 重写后匹配主机头部字段

- path_rewrite

  (optional, string) 重写后匹配路径头部字段

- path_redirect

  (optional, string) 匹配返回的重定向路径

- header_fields

  (optional, array) 匹配列出的标题字段。示例标题字段包括“:path”，“cookie”和“date”字段。在所有其他测试用例之后检查标题字段。因此，检查的标题字段将是适用时重定向或重写路由的字段。

  - field

    (required, string) 要匹配的标题字段的名称

  - value

    (required, string) 要匹配的标题字段的值

返回

- 上一级
- 首页目录

运维管理

运维&管理

- 命令行选项
- 热重启
- 管理接口
- 统计概述
- 运行时配置
- 文件系统标示

返回

- 首页目录

命令行选项

命令行选项

Envoy由JSON配置文件以及一组命令行选项驱动。以下是Envoy支持的命令行选项。

- -c

  （必选）指向v1或v2 JSON/YAML/proto3配置文件的路径。若设置`-v2-config-only`选项，则将被解析为一个v2引导配置文件，如果是v1 JSON配置文件，则返回失败。对于v2配置文件，有效的扩展名是`.json`，`.yaml`，`.pb`和`.pb_text`，分别表示JSON，YAML，二进制proto3和文本proto3格式。

- --v2-config-only

  （可选）该标志表示配置文件是否仅为v2引导配置文件。如果为`false`（默认值），那么当v2引导配置解析失败时，将尝试解析为v1 JSON配置文件。

- --mode

  （可选）Envoy的其中一种操作模式：

  serve：(默认）验证JSON配置，然后正常提供流量。

  validate：验证JSON配置，然后退出，打印“OK”消息（在这种情况下退出代码为`0`）或者因配置文件的任何错误（退出代码为`1`）。不会产生网络流量，并且不会执行热重启，所以不会影响其他任何进程。

- --admin-address-path

  （可选）将输出管理员地址和端口的文件路径。

- --local-address-ip-version

  （可选）用于填充服务器本地IP地址的IP地址版本。此参数影响各种标题，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

  （可选）分配共享内存区域时使用的基本ID。Envoy在热启动期间使用共享内存区域。大多数用户不需要设置这个选项。但是，如果Envoy需要在同一台计算机上多次运行，则每个运行的Envoy都需要

一个唯一的基本ID，以便共享内存区域不会发生冲突。

- --concurrency

  （可选）要运行的工作线程数。如果未指定，则默认为机器上的硬线程数。

- -l

  （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，请参阅帮助文档。

- --log-path

  （可选）日志输出的文件路径。当 **`SIGUSR1`** 被处理时，文件会被重新打开。如果没有设置，记录到 **`stderr`** 。

- --restart-epoch

  （可选）热重启期间。（Envoy重新启动的次数，而不是重新开始）。第一次启动时默认为0。此选项告诉Envoy是否尝试创建热重启所需的共享内存区域，或者是否打开现有的共享内存区域。每次热重启时都应该增加这个值。通常在大多数情况下，应该设置此 **`RESTART_EPOCH`** 环境变量。

- --hot-restart-version

  （可选）为当前的二进制输出一个热重启兼容的版本。这可以与GET /hot_restart_version管理端口的输出相匹配，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

  （可选）定义Envoy运行的本地服务群集名称。尽管是可选的，但是如果使用以下任何特性，应该设置：statsd，健康检查集群验证，运行时配置目录覆盖，添加用户代理，HTTP全局速率限制，CDS和HTTP跟踪。

- --service-node

  （可选）定义Envoy运行的本地服务节点名称。虽然是可选的，但是如果使用以下任何功能，应该设置它们：statsd，CDS和HTTP跟踪。

- --service-zone

  （可选）定义Envoy运行的本地服务区域。尽管是可选的，但是如果使用路由发现服务并且发现服务暴露区域信息，则应该设置它。

- --file-flush-interval-msec

  （可选）文件刷新间隔，以毫秒为单位。默认为10秒。在创建文件时使用此设置来确定缓冲区刷新到文件之间的持续时间。缓冲区满或每隔一段时间刷新一次（以先到者为准）。为了获得更多（或更少）的即时刷新，调整此设置对跟踪访问日志非常有用。

- --drain-time-s

  （可选）Envoy在热重启期间将耗尽连接的时间（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，逐出时间应小于通过 `--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。在边缘情况下，可能需要耗费很长时间。在服务场景中，可能使逐出和关闭时间缩短得多（例如，60s/90s）。

- --parent-shutdown-time-s

  （可选）Envoy在热重启期间关闭父进程之前等待的时间（秒）。请参阅热启动概述了解更多信息。默认为900秒（15分钟）。

- --max-obj-name-len

  （可选） `cluster/route_config/listener` 中名称字段的最大长度（以字节为单位）。此选项通常用于群集名称自动生成的场景，通常超过会60个字符的内部限制。默认为60。

  注意：此设置会影响 `--hot-restart-version` 的输出。如果您开始使用此选项，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

  （可选）热重启之间可以共享的最大统计数量。此设置会影响 `--hot-restart-version` 的输出;必须使用相同的值来配置热重启进程。默认为16384。

返回

- 上一级
- 首页目录

热重启

热重启

通常情况下，Envoy将热启动以支持配置和二进制的更新升级。但是，在许多情况下，用户会希望使用标准的进程管理器，如monit，runit等。我们提供/restarter/hot-restarter.py来使实现这个功能。

启动程序是这样调用的：

```
hot-restarter.py start_envoy.sh
```

start_envoy.sh可以参考使用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_avail
ability-zone', 'unknown') }}
```

环境变量`RESTART_EPOCH`需要在每次重新启动时由重启动程序设置，并可以传递给`--restart-epoch`选项。

重启程序需要处理以下信号：

- SIGTERM：将干净地终止所有子进程并退出。
- SIGHUP：将重新调用热重启脚本，并且使用一个传递参数内容来重启。
- SIGCHLD：如果有任何子进程意外关闭，那么重启脚本将关闭一切并退出以避免处于意外状态。然后，进程控制管理器应该重新启动脚本以再次启动Envoy。
- SIGUSR1：将被转发给Envoy进程，作为重新打开所有访问日志的信号。这用于移动并重新打开日志原子操作。

返回

- 上一级
- 首页目录

管理接口

管理接口

Envoy公开了一个本地管理界面，可以用来查询和修改服务的不同方面：

- v1 API 参考
- v2 API 参考

GET /

打印所有可用的API清单

GET /certs

列出所有加载的TLS证书，包括文件名，序列号和到期日期。

GET /clusters

列出所有配置的集群管理器集群。此信息包括每个群集中发现的所有上游主机以及每个主机统计信息。这对服务发现的问题调试很有用。

- 集群管理器信息

  `version_info` 字符串，上次加载的CDS服务版本信息的字符串。如果envoy没有安装CDS，将会读取 `version_info::static` 输出。

- 集群信息

  - 所有优先级都设置熔断。
  - 如果使能了异常值检测，将会呈现成功率平均值和逐出阈值。如果在最后一个时间间隔内没有足够的数据来计算它们，那么这两个值都将是-1。
  - `added_via_api` 标志，如果通过静态配置添加的集群，则为 `false` ，如果通过CDS API添加，则为 `true` 。

- 按主机统计

名称	类型	描述
cx_total	Counter	连接总数
cx_active	Gauge	总活动连接数
cx_connect_fail	Counter	总连接失败数
rq_total	Counter	总请求数
rq_timeout	Counter	总请求超时数
rq_success	Counter	带有非5xx响应的总请求数

rq_error	Counter	带有5xx响应的总请求数
rq_active	Gauge	总活动请求数
healthy	String	主机的健康状况。 见下文
weight	Integer	负载平衡权重（1-100）
zone	String	所在服务区域
canary	Boolean	主机是否是金丝雀（灰度发布）状态
success_rate	Double	请求成功率（0-100）。 如果间隔中没有足够的请求量来计算它，则返回-1

- 主机健康状况

  由于一个或多个不健康的状态，主机可能是健康的或不健康的。
  如果主机健康，则会输出 `healthy` 字符串。
  如果主机不健康，则会输出以下一个或多个字符串：

  /failed_active_hc：主机主动健康检查失败。

  /failed_outlier_check：主机未通过异常值检测检查。

GET /cpuprofiler
启用或禁用CPU分析器。需要与 `gperftools` 一起进行编译。

GET /healthcheck/fail
入站健康检查失败。这需要使用HTTP健康检查过滤器。这对于将要关闭服务或完全重新启动之前，逐出服务非常有用。无论过滤器如何配置，调用此命令都将执行健康检查失败的请求。

GET /healthcheck/ok
取消 `GET /healthcheck/fail` 的作用。这需要使用HTTP健康检查过滤器。

GET /hot_restart_version
参见--hot-restart-version

GET /logging
在不同的子系统上启用/禁用不同的日志记录级别。一般只在开发过程中使用。

GET /quitquitquit
干净地退出服务

GET /reset_counters
将所有计数器清零。在调试过程中，这对 `GET /stats` 很有用。请注意，这不会影响任何发送到

`statsd`的数据。它只会影响`GET /stats`本地命令的输出。

GET /routes?route_config_name=
此接口仅在envoy具有配置RDS的HTTP路由时才可用。如果指定了查询，则此接口会转储所有已配置的HTTP路由表，或者仅转储与`route_config_name`查询匹配的HTTP路由表。

GET /server_info

输出有关运行的服务器的信息。输出示例如下所示：

```
envoy 267724/RELEASE live 1571 1571 0
```

这些字段是：

- 进程名称
- 编译SHA和生成类型
- 健康检查状态（活跃或逐出）
- 当前热重启时间，以秒为单位
- 正常运行总时间（跨所有热重启阶段），以秒为单位
- 当前热重启的迭代数

GET /stats

输出所有需要的统计数据。这只包计数和测量值。直方图不会输出，因为Envoy目前没有内置直方图，依赖`statsd`进行汇总。这个命令对本地调试非常有用。浏览此处获取更多信息。

- GET /stats?format=json

  以JSON格式的输出统计信息。这个统计信息支持编程对接。

- GET /stats?format=prometheus

  以Prometheus v0.0.4格式的输出。这可以用来与`Prometheus`服务器集成。目前，只有计数器和计量器输出。直方图将在未来版本中提供。

返回

- 上一级
- 首页目录

统计概述

统计概述

Envoy输出许多统计信息，这取决于如何配置服务器。它们可以通过本地 `GET /stats` 命令查看，通常发送到一个statsd集群。输出的统计信息记录在配置指南的相关章节中。一些常用的重要的统计数据，可以在下面的章节中找到：

- HTTP连接管理
- 上游集群管理

返回

- 上一级
- 首页目录

运行时配置

运行时配置

运行时配置可用于修改各种服务配置，而无需重新启动Envoy。可用的运行时设置取决于服务的配置方式。它们记录在配置指南的相关章节。

返回

- 上一级
- 首页目录

文件系统

文件系统标识

Envoy支持在启动时改变状态的文件系统“标识”。这用于在必要时重新启动之间保持更改。标识文件应放置在配置选项flags_path中指定的目录中。当前支持的标识文件是：

- drain

  如果此文件存在，Envoy将以HC失败模式启动，类似于执行`GET /healthcheck/fail`命令之后。

返回

- 上一级
- 首页目录

自定义扩展示例

自定义扩展示例

通过Envoy架构很容易扩展网络过滤器和HTTP过滤器。

有关如何添加网络过滤器和构建库并建立依赖关系的示例，请参见envoy-filter-example。

返回

- 首页目录

V1 API参考

V1 API参考

- 访问日志
- 管理接口
- 限速服务
- 运行时
- 跟踪

返回

- 首页目录

监听器

监听器

```
{
  "name": "...",
  "address": "...",
  "filters": [],
  "ssl_context": "{...}",
  "bind_to_port": "...",
  "use_proxy_proto": "...",
  "use_original_dst": "...",
  "per_connection_buffer_limit_bytes": "...",
  "drain_type": "..."
}
```

- name

  (optional, string) 这个已知监听器的唯一名称。如果没有提供名称，Envoy将为监听器分配一个内部UUID。如果要通过LDS动态更新或删除监听器，则必须提供唯一的名称。默认情况下，监听器名称的最大长度限制为60个字符。通过 `--max-obj-name-len` 命令行参数可以设置为所需的值，可以提高此限制。

- address

  (required, string) 监听器所监听的地址。目前仅支持监听TCP，例如“tcp://127.0.0.1:80”。请注意，“tcp://0.0.0.0:80”是匹配任何带有端口80的IPv4地址的通配符。

- filters

  (required, array) 监听器连接时需要处理的过滤器链，包含各个网络过滤器的列表。当发生连接事件时，将按顺序处理过滤器链。

  注意：如果过滤器列表为空，则默认关闭连接。

- ssl_context

  (optional, object) 监听器的TLS上下文配置。如果没有定义TLS上下文，则监听器是纯文本监听器。

- bind_to_port

  (optional, boolean) 是否将监听器绑定到相应的端口。不绑定的监听器只能接收其他监听器重定向的连接，将 `use_original_dst` 参数设置为true。默认是true。

- use_proxy_proto

  (optional, boolean) 监听器是否应该在新连接上使用PROXY协议V1头。如果启用此选项，则监听器将假定该连接的远程地址是在头部中指定的地址。包括 `AWS ELB` 的一些负载平衡器，也支持此选项。如果该选项不存在或设置为false，Envoy将使用连接的物理对等地址作为远程地址。

- use_original_dst

  (optional, boolean) 如果使用iptables重定向连接，则代理接收连接的端口可能与原始目标地址不同。当此标志设置为true时，监听器将重定向的连接切换到与原始目标地址关联的监听器。如果没有与原始目标地址关联的监听器，则连接由接收该监听器的监听器处理。默认为false。

- per_connection_buffer_limit_bytes

  (optional, integer) 监听器的新连接读取和写入缓冲区大小的软限制。如果未指定，则应用实现定义的默认值（1MB）。

- drain_type

  (optional, string) 监听器所做的逐出期间的类型。允许的值包括 `default` 和 `modify_only` 。有关更多信息，请参阅逐出架构概述。

Filters

网络过滤器架构概述。

```
{
  "name": "...",
  "config": "{...}"
}
```

- name

  (required, string) 要实例化的过滤器的名称。该名称必须与支持的过滤器名匹配。

- config

  (required, object) 指定的过滤器配置，这取决于被实例化的过滤器。有关更多文档，请参阅支持的过滤器。

TLS context

TLS架构概述。

```
{
  "cert_chain_file": "...",
  "private_key_file": "...",
  "alpn_protocols": "...",
  "alt_alpn_protocols": "...",
  "ca_cert_file": "...",
  "verify_certificate_hash": "...",
  "verify_subject_alt_name": [],
  "cipher_suites": "...",
  "ecdh_curves": "...",
  "session_ticket_key_paths": []
}
```

- cert_chain_file

  (required, string) 提供该监听器使用的证书链文件。

- private_key_file

  (required, string) 与证书链文件相对应的私钥。

- alpn_protocols

  (optional, string) 由该监听器公开的ALPN协议列表。实际上，这可能会被设置为两个值之一（有关更多信息，请参阅HTTP连接管理器中的codec_type参数）：

  - “h2,http/1.1” ：如果监听器要同时支持HTTP/2和HTTP/1.1。
    - “http/1.1” ：如果监听器只支持HTTP/1.1

- alt_alpn_protocols

  (optional, string) 一个可以通过运行时切换到的ALPN协议字符串。例如在不通过部署新的配置时，禁用HTTP/2是很有用。

- ca_cert_file

  (optional, string) 包含证书颁发机构证书的文件，用于验证客户端提供的证书。如果未指定，而客户端提供了证书，则不会进行验证。默认情况下，客户端的证书是可选的，除非还指定了其中一个附加选项（`require_client_certificate`，`verify_certificate_hash` 或 `verify_subject_alt_name`）。

- require_client_certificate

  (optional, boolean) 如果指定，Envoy将拒绝没有携带有效证书的客户端连接。

- verify_certificate_hash

  (optional, string) 如果指定，Envoy将验证（pin）所提供的客户端证书的HASH。

- verify_subject_alt_name

  (optional, array) 一组可选的标题名称，如果指定，Envoy将验证客户端证书的标题ALT名称是否与指定其中之一匹配。

- cipher_suites

  (optional, string) 如果指定，则TLS监听器将仅支持指定的加密套件。如果未指定，则默认列表为：

```
[ECDHE-ECDSA-AES128-GCM-SHA256|ECDHE-ECDSA-CHACHA20-POLY1305]
[ECDHE-RSA-AES128-GCM-SHA256|ECDHE-RSA-CHACHA20-POLY1305]
ECDHE-ECDSA-AES128-SHA256
ECDHE-RSA-AES128-SHA256
ECDHE-ECDSA-AES128-SHA
ECDHE-RSA-AES128-SHA
AES128-GCM-SHA256
AES128-SHA256
AES128-SHA
ECDHE-ECDSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES256-SHA384
ECDHE-RSA-AES256-SHA384
ECDHE-ECDSA-AES256-SHA
ECDHE-RSA-AES256-SHA
AES256-GCM-SHA384
AES256-SHA256
AES256-SHA
```

- ecdh_curves

  (optional, string) 如果指定，TLS连接将只支持指定的ECDH曲线算法。如果未指定，将使用默认曲线算法（X25519，P-256）。

- session_ticket_key_paths

  (optional, array) 用于加密和解密TLS会话凭证的密钥文件路径。数组中的第一个密钥文件包含加密由此上下文创建的所有新会话的密钥。所有的密钥都是解密收到的凭证的备选。这允许通过例如将新的密钥文件放在第一位，将先前的密钥文件放在第二位，来很容易实现密钥轮换。

  如果未指定`session_ticket_key_paths`，则TLS库仍将支持通过故障恢复会话，但会使用内部

生成和管理的密钥，因此会话不能在热重启或不同的主机上恢复。

每个密钥文件必须包含正好80个字节的密码安全随机数据。 例如，openssl rand 80的输出。

注意：使用此功能需要考虑存在的安全风险。即使支持完美前向保密的密码，对密钥的不正确处理也可能导致连接的保密性丧失。有关讨论，请参阅[连接]https://www.imperialviolet.org/2013/06/27/botchingpfs.html。为了最大限度地降低风险，您必须：

- 保持会话凭证密钥至少与TLS证书私钥一样安全
- 至少每天轮换会话凭证密钥，最好每小时轮换一次
- 始终使用密码安全的随机数据源生成密钥

Listener discovery service (LDS)

```
{
  "cluster": "...",
  "refresh_delay_ms": "..."
}
```

- cluster

  (required, string) 承载监听发现服务的上游群集名称。群集必须实现LDS HTTP API的REST服务。注：这是在群集管理器配置中定义的群集名称，而不是群集的完整定义，例如SDS和CDS的配置方式。

- refresh_delay_ms

  (optional, integer) 从LDS API获取信息的延迟（以毫秒为单位）。Envoy将在0到`refresh_delay_ms`之间的延迟上添加一个额外的随机抖动值。因此，最长的刷新延迟可能是`2*refresh_delay_ms`。默认值是30000ms（30秒）。

REST API

```
GET /v1/listeners/(string: service_cluster)/(string: service_node)
```

请求发现服务返回特定`service_cluster`和`service_node`的所有监听器。`service_cluster`对应于`--service-cluster` CLI选项。`service_node`对应于`--service-node` CLI选项。使用以下JSON格式响应：

```
{
  "listeners": []
}
```

- listeners

  (required, array) 将在监听器管理器中动态添加/修改的监听器列表。管理服务器将在每个轮询周期内，使用Envoy配置的完整监听程序进行响应。Envoy将协调此列表与当前加载的监听器，并根据需要添加/修改/删除监听器。

返回

- 上一级
- 首页目录

网络过滤器

网络过滤器

返回

- 上一级
- 首页目录

TLS客户端身份认证

TLS客户端身份认证

TLS客户端身份认证配置参考。

```
{
  "name": "client_ssl_auth",
  "config": {
    "auth_api_cluster": "...",
    "stat_prefix": "...",
    "refresh_delay_ms": "...",
    "ip_white_list": []
  }
}
```

- auth_api_cluster

  (required, string) 所运行的身份验证服务的群集名称。过滤器将每60秒连接到该服务以获取身份信息。该服务必须支持预期的REST API。

- stat_prefix

  (required, string) 发布统计信息时使用的前缀。

- refresh_delay_ms

  (optional, integer) 验证服务的身份信息刷新间隔（以毫秒为单位）。默认是60000（60s）。实际的提取时间将是这个值加0到 `refresh_delay_ms` 毫秒之间的随机抖动值。

- ip_white_list

  (optional, array) 一个可选的IP地址和子网掩码列表，提供白名单列表给过滤器使用。如果没有提供列表，则没有IP白名单。该列表如下例所示：

  ```
  [
  ```

  ```
    "192.168.3.0/24",
    "50.1.2.3/32",
    "10.15.0.0/16",
    "2001:abcd::/64"
  ```

  ```
  ]
  ```

`

返回

- 上一级
- 首页目录

Echo

Echo

Echo配置参考。

```
{
  "name": "echo",
  "config": {}
}
```

返回

- 上一级
- 首页目录

HTTP连接管理

HTTP连接管理

- HTTP连接管理架构概述。
- HTTP协议架构概述。

```
{
  "name": "http_connection_manager",
  "config": {
    "codec_type": "...",
    "stat_prefix": "...",
    "rds": "{...}",
    "route_config": "{...}",
    "filters": [],
    "add_user_agent": "...",
    "tracing": "{...}",
    "http1_settings": "{...}",
    "http2_settings": "{...}",
    "server_name": "...",
    "idle_timeout_s": "...",
    "drain_timeout_ms": "...",
    "access_log": [],
    "use_remote_address": "...",
    "forward_client_cert": "...",
    "set_current_client_cert": "...",
    "generate_request_id": "..."
  }
}
```

- codec_type

  (required, string) 提供连接管理器所使用的编解码器的类型。可能的值有：

  - http1

    连接管理器将假定客户端正在使用HTTP/1.1。
  - http2

    连接管理器将假定客户端正在使用HTTP/2（Envoy并不需要通过TLS发送HTTP/2或者使用ALPN，应预先设定好）。
  - auto

    对于每个新的连接，连接管理器将决定使用哪个编解码器。此模式支持TLS监听器的ALPN以及明文监听器的协议鉴别。如果ALPN数据可用，则优选，否则使用协议鉴别。几乎在所有情况下，设置此选择的是正确的选择。

- stat_prefix

  (required, string) 发布连接管理器的统计信息时使用的人类可读前缀。有关更多信息，请参阅统计文档。

- rds

  (sometimes required, object) 连接管理器配置必须指定选择 `rds` 或 `route_config` 其中之一。如果指定了 `rds` ，则连接管理器的路由表将通过RDS API动态加载。有关更多信息，请参阅文档。

- route_config

  (sometimes required, object) 连接管理器配置必须指定选择 `rds` 或 `route_config` 其中之一。如果指定了 `route_config` ，则认为该连接管理器的路由表是静态的，并在此属性中指定路由配置。

- filters

  (required, array) 构成该连接管理器请求的过滤器链，包含多个HTTP过滤器的列表当请求事件发生时，将按顺序处理过滤器。

- add_user_agent

  (optional, boolean) 连接管理器是否处理user-agent和x-envoy-downstream-service-cluster头。有关更多信息，请参阅相关链接。默认为false。

- tracing

  (optional, object) 定义的跟踪服务的配置对象，提供连接管理器跟踪数据发往的跟踪服务程序配置。

- http1_settings

  (optional, object) 传递给HTTP/1编解码器的扩展配置。

  - allow_absolute_url

    (optional, boolean) 在请求中处理具有绝对URL的http请求。这些请求通常由客户端发送到转发/显式代理。这允许客户端将envoy配置为他们的http代理。例如，在Unix中，这通常是通过设置http_proxy环境变量来完成的。

- http2_settings

(optional, object) 传递给HTTP/2编解码器的扩展配置。目前支持的设置有：

- hpack_table_size

  (optional, integer) 允许编码器使用的动态HPACK表的最大值（以八位字节为单位）。有效值范围从0到4294967295（2^32-1），默认值为4096. 0表示禁用头部压缩。

- max_concurrent_streams

  (optional, integer) 一个HTTP/2连接上同时允许的最大并发流。有效值范围从1到2147483647（2^31-1），默认值为2147483647。

- initial_stream_window_size

  (optional, integer) 初始时控制流窗口的大小。有效值范围从65535（2^16-1，HTTP/2默认值）到2147483647（2^31-1，HTTP/2最大值），默认值为268435456（25610241024）。

  注：65535是来自HTTP/2规范的初始窗口大小。我们现在只支持增加默认的窗口大小，所以也是最小的。

  这个字段也可以作为Envoy在HTTP/2编解码器缓冲区中每个字节缓冲的字节数的软限制。一旦缓冲区到达这个指针，将触发停止接收数据流到编解码器缓冲区。

- initial_connection_window_size

  (optional, integer) 与`initial_stream_window_size`类似，但是用于连接级流量控制窗口。目前，这与`initial_stream_window_size`具有相同的最小/最大/默认值。

这些与上游群集`http2_settings`中提供的选项相同。

- server_name

  (optional, string) 可选，连接管理器将在响应中增加服务名的头部字段。如果未设置，则默认为Envoy。

- idle_timeout_s

  (optional, integer) 由该连接管理器所管理的连接空闲超时时间（以秒为单位）。空闲超时被定义为一段时间内没有活动请求。如果没有设置，则没有空闲超时。当达到空闲超时时，连接将被关闭。如果连接是HTTP/2连接，则在关闭连接之前会发生顺序排空。请参阅drain_timeout_ms。

- drain_timeout_ms

(optional, integer) Envoy将在发送HTTP/2“关闭通知”（GOAWAY帧与最大流ID）和最终GOAWAY帧之间等待的时间。这是为了让Envoy支持与最后GOAWAY帧竞争的新的流处理，所提供的宽限期。在这个宽限期间，Envoy将继续接受新的流。在宽限期之后，最终GOAWAY帧被发送，Envoy将开始拒绝新的流。在连接遇到空闲超时或通用服务器耗尽时都会发生排空。如果未指定此选项，则默认宽限期为5000毫秒（5秒）。

- access_log

  (optional, array) 连接管理器发出的HTTP访问日志的配置。

- use_remote_address

  (optional, boolean) 如果设置为true，连接管理器将在确定内部和外部源以及操作各种头部时使用客户端连接的真实远程地址。如果设置为false或不存在，连接管理器将使用 `x-forwarded-for` HTTP头。有关更多信息，请参阅x-forwarded-for，x-envoy-internal和x-envoy-external-address的文档。

- forward_client_cert

  (optional, string) 如何处理 `x-forward-client-cert`（XFCC）HTTP头。可能的值为：

  - sanitize: 不要将XFCC头部发送到下一跳。这是默认值。
  - forward_only: 当客户端连接是mTLS（Mutual TLS）时，转发请求中的XFCC头。
  - always_forward_only: 始终在请求中转发XFCC标头，而不管客户端连接是否为mTLS。
  - append_forward: 当客户端连接是mTLS时，将客户端证书信息附加到请求的XFCC头并转发它。
  - sanitize_set: 当客户端连接是mTLS时，使用客户端证书信息重置XFCC头，并将其发送到下一个跃点。

  有关XFCC头的格式，请参阅x-forward-client-cert。

- set_current_client_cert_details

  (optional, array) 字符串列表，只有在forward_client_cert为 `append_forward` 或 `sanitize_set` 且客户端连接为 `mTLS` 时，此字段才有效。它指定要转发的客户端证书中的字段。请注意，在x-forwarded-client-cert头中，始终设置 `Hash` ，并在客户端证书显示SAN值时始终设置 `By` 。

- generate_request_id

(optional, boolean) 连接管理器是否自动生成x-request-id头，如果该头不存在。默认为true。生成一个随机的UUID4（性能代价比较大），所以在高吞吐量的情况下，这个功能是不需要的，它可以被禁用。

Tracing

```
{
  "tracing": {
    "operation_name": "...",
    "request_headers_for_tags": []
  }
}
```

- operation_name

  (required, string) span的名称从`operation_name`中获取。目前仅支持“ingress” 和“egress”。

- request_headers_for_tags

  (optional, array) 用于为活跃的span创建头部名称的列表。`header`名称用于填充`tag`名称，`header`值用于填充`tag`值。如果指定的`header`名称出现在请求头中，则会创建该`tag`。

Filters

HTTP过滤器架构概述。

```
{
  "name": "...",
  "config": "{...}"
}
```

- name

  (required, string) 要实例化的过滤器的名称。该名称必须与支持的过滤器匹配。

- config

  (required, object) 指定的过滤器配置，这取决于被实例化的过滤器类型。有关更多文档，请参阅支持的过滤器。

返回

- 上一级
- 首页目录

Mongo代理

Mongo代理

MongoDB配置参考。

```
{
  "name": "mongo_proxy",
  "config": {
    "stat_prefix": "...",
    "access_log": "...",
    "fault": {}
  }
}
```

- stat_prefix

  (required, string) 用于发布统计信息时所使用的前缀。

- access_log

  (optional, string) 用于Mongo访问日志的文件系统路径。如果未指定访问日志的路径，则不会写入访问日志。请注意，访问日志也受运行时配置控制。

- fault

  (optional, object) 如果指定，过滤器将根据 `object` 中的值注入故障。

Fault configuration

MongoDB提供持续固定时间延迟的配置。适用于以下MongoDB操作： `Query` ， `Insert` ， `GetMore` 和 `KillCursors` 。一旦配置延时并且生效，在定时器触发之前，所有输入数据的时间，也将成为延迟时间的一部分。

```
{
  "fixed_delay": {
    "percent": "...",
    "duration_ms": "..."
  }
}
```

- percent

  (required, integer) 当没有故障时，正常的MongoDB操作的受到故障注入的影响的概率。有效值是[0，100]范围内的整数。

- duration_ms

  (required, integer) 非负整数，持续延迟的时间（以毫秒为单位）。

返回

- 上一级
- 首页目录

速率限制

速率限制

速率限制配置参考。

```
{
  "name": "ratelimit",
  "config": {
    "stat_prefix": "...",
    "domain": "...",
    "descriptors": [],
    "timeout_ms": "..."
  }
}
```

- stat_prefix

  (required, string) 发布统计信息时使用的前缀。

- domain

  (required, string) 用于速率限制服务请求中的限制域。

- descriptors

  (required, array) 用于速率限制服务请求中的速率限制描述符列表。描述符按照以下示例进行指定：

  `
  [

  ```
  [
      {
          "key": "hello",
          "value": "world"
      },
      {
          "key": "foo",
          "value": "bar"
      }
  ],
  [
      {
          "key": "foo2",
          "value": "bar2"
      }
  ]
  ```

]
`

- timeout_ms

  (optional, integer) 速率限制服务RPC的超时时间（以毫秒为单位）。如果未设置，则默认为20ms。

返回

- 上一级
- 首页目录

Redis代理

Redis代理

Redis代理配置参考。

```
{
  "name": "redis_proxy",
  "config": {
    "cluster_name": "...",
    "conn_pool": "{...}",
    "stat_prefix": "..."
  }
}
```

- cluster_name

  (required, string) 对应集群管理器的集群名称。有关群集的配置建议，请参阅架构概述的配置部分。

- conn_pool

  (required, object) 连接池配置。

- stat_prefix

  (required, string) 发布统计信息时使用的前缀。

Connection pool configuration

```
{
  "op_timeout_ms": "...",
}
```

- op_timeout_ms

  (required, integer) 每操作的超时时间（单位：毫秒）。定时器是在管道的第一个命令写入后端连接时启动。从Redis收到的每个响应都会重置定时器，因为它表示下一个命令由后端处理的时间。这种行为的唯一例外是到后端的连接尚未建立。在这种情况下，将由群集上的连接超时控制，直到连接准备就绪。

返回

- 上一级
- 首页目录

TCP代理

TCP代理

TCP代理配置参考

```
{
  "name": "tcp_proxy",
  "config": {
    "stat_prefix": "...",
    "route_config": "{...}",
    "access_log": []
  }
}
```

- route_config

  (required, object) 过滤器的路由表。所有的过滤器实例都必须有一个路由表，即使它是空的。

- stat_prefix

  (required, string) 发布统计信息时使用的前缀。

- access_log

  (optional, array) 由此 `tcp_proxy` 发出的访问日志配置。

Route Configuration

```
{
  "routes": []
}
```

- routes

  (required, array) 组成路由表的一组路由条目。

Route

TCP代理的路由是由一组可选的标准L4和一个集群的名称组成。如果下游连接符合所有指定的条件，则路由中的集群将用于相应的上游连接。按照指定的顺序进行路由尝试，直到找到匹配的路由。如果找不到匹配，则连接关闭。并不是始终有相应的路由，以及对应的匹配。

```
{
  "cluster": "...",
```

```
  "destination_ip_list": [],
  "destination_ports": "...",
  "source_ip_list": [],
  "source_ports": "..."
}
```

- cluster

  (required, string) 符合指定条件时要连接的下游群集。

- destination_ip_list

  (optional, array) 可选，包含IP地址子网的列表，格式为“ip_address/xx”。如果下游连接的目标IP地址包含在至少一个指定的子网中，则条件匹配。如果未指定参数或列表为空，则将忽略目标IP地址。如果连接已被重定向，则下游连接的目标IP地址可能与代理正在监听的地址不同。例：

  `
  [

  ```
  "192.168.3.0/24",
  "50.1.2.3/32",
  "10.15.0.0/16",
  "2001:abcd::/64"
  ```

  ]
  `

- destination_ports

  (optional, string) 包含`,`分隔符的端口号或端口范围列表。如果下游连接的目标端口至少包含在一个指定范围内，则条件匹配。如果未指定参数，则将忽略目标端口。如果连接已被重定向，下游连接的目标端口地址可能与代理正在监听的端口不同。例：

  `
  {

  ```
  "destination_ports": "1-1024,2048-4096,12345"
  ```

  }
  `

- source_ip_list

  (optional, array) 可选，包含IP地址子网的列表，格式为“ip_address/xx”。如果下游连接的源IP地

址包含在至少一个指定的子网中，则条件匹配。如果未指定参数或列表为空，则将忽略源IP地址。例：

```
[
  "192.168.3.0/24",
  "50.1.2.3/32",
  "10.15.0.0/16",
  "2001:abcd::/64"
]
```

- source_ports

  (optional, string) 可选，包含`,`分隔的端口号或端口范围列表。如果下游连接的源端口包含在至少一个指定范围内，则条件匹配。如果未指定参数，则源端口将被忽略。例：

```
{
  "source_ports": "1-1024,2048-4096,12345"
}
```

返回

- 上一级
- 首页目录

HTTP路由配置

HTTP路由配置

- HTTP路由架构概述
- HTTP路由过滤器

```
{
  "validate_clusters": "...",
  "virtual_hosts": [],
  "internal_only_headers": [],
  "response_headers_to_add": [],
  "response_headers_to_remove": [],
  "request_headers_to_add": []
}
```

- validate_clusters

  (optional, boolean) 可选的bool类型，是否需要集群管理器验证路由表所引用的集群。如果设置为true，若路由表中引用了不存在的集群，则路由表将不会加载。如果设置为false，若路由引用不存在的集群，则路由表将加载，如果在运行时选择路由，则路由器过滤器将返回404。如果路由表是通过`route_config`选项静态定义的，则此设置默认为true。如果路由表是通过rds选项动态加载的，则此设置默认为false。用户可以在某些情况下修改默认行为（例如，在使用静态路由表时使用cds）。

- virtual_hosts

  (required, array) 组成路由表的一组虚拟主机。

- internal_only_headers

  (optional, array) （可选）指定仅为连接管理器内部使用的HTTP头部列表。如果在外部请求中找到它们，将在过滤器调用之前清除它们。有关更多信息，请参见x-envoy-internal。以下面的形式指定头部：
  `["header1", "header2"]`

- response_headers_to_add

  (optional, array) （可选）连接管理器在编码时需要添加到每个响应中的HTTP头部列表。以下面的方式指定：
  `
  [

```
  {"key": "header1", "value": "value1"},
  {"key": "header2", "value": "value2"}
```

]
`

- response_headers_to_remove

  (optional, array)（可选）连接管理器在编码每个响应时，需要删除的HTTP头部列表。以下面的形式指定：
  `["header1", "header2"]`

- request_headers_to_add

  (optional, array) HTTP连接管理器在转发每个请求时需要添加的HTTP头部列表，以下面的形式指定：
  `
  [

```
  {"key": "header1", "value": "value1"},
  {"key": "header2", "value": "value2"}
```

  ]
  `
  有关更多信息，请参阅自定义请求头部的文档。

- 虚拟主机

- 路由
- 虚拟集群
- 速率限制配置
- 路由发现服务

返回

- 上一级
- 首页目录

虚拟主机

虚拟主机

虚拟主机是路由配置中的顶层配置。每个虚拟主机都有一个逻辑名称以及一组域列表，会根据传入请求的主机头路由到对应的域。这允许为单个监听器配置多个顶级域的路径树。一旦基于域选择了虚拟主机，就会进行路由处理，以便查找并路由到相应上游集群或者是否执行重定向。

```
{
  "name": "...",
  "domains": [],
  "routes": [],
  "require_ssl": "...",
  "virtual_clusters": [],
  "rate_limits": [],
  "request_headers_to_add": []
}
```

- name

  (required, string) 虚拟主机的逻辑名称。这在发送某些统计信息时使用，但与转发无关。默认情况下，名称的最大长度限制为60个字符。可通过`--max-obj-name-len`命令行参数设置为所需的值，以提高此限制。

- domains

  (required, array) 将与此虚拟主机相匹配的域（主机/机构头）的列表。支持通配符匹配主机，如“.foo.com”或“-bar.foo.com”。请注意，通配符不匹配空字符串。例如“-bar.foo.com”将匹配“baz-bar.foo.com”，但不匹配“-bar.foo.com”。另外，允许一个特殊的“”匹配任何主机/机构头。整个路由配置中只有一台虚拟主机可以匹配“*”。域名在所有虚拟主机中必须是唯一的，否则配置将无法加载。

- routes

  (required, array) 路由列表，将按配置顺序匹配传入请求。第一条匹配的路由将被使用。

- cors

  (optional, object) 指定虚拟主机的CORS策略。

- require_ssl

  (optional, string) 指定虚拟主机所期望的TLS认证的配置类型。可能的值有：

- all

  所有请求都必须使用TLS。如果请求不使用TLS，则会发送302重定向，通知客户端使用HTTPS。
- external_only

  来自外部请求必须使用TLS。如果请求是来自外部的，并且它没有使用TLS，则会发送302重定向，通知客户端使用HTTPS。

如果未指定此选项，则虚拟主机没有TLS要求。

- virtual_clusters

  (optional, array) 为此虚拟主机定义的虚拟集群列表。虚拟集群用于进行其他统计信息收集。

- rate_limits

  (optional, array) 应用于虚拟主机的一组速率限制配置。

- request_headers_to_add

  (optional, array) 指定此虚拟主机要添加到每个请求的HTTP头部列表。以下面的形式指定：
  `
  [

```
{"key": "header1", "value": "value1"},
{"key": "header2", "value": "value2"}
```

  ]
  `
  有关更多信息，请参阅自定义请求头部的文档。

返回

- 上一级
- 首页目录

路由

路由

路由指定请求如何匹配相应的路径以及接下来要做什么（例如，重定向，转发，重写等）的规范。

注意：Envoy支持通过匹配HTTP头中的方法进行路由。

```
{
  "prefix": "...",
  "path": "...",
  "regex": "...",
  "cluster": "...",
  "cluster_header": "...",
  "weighted_clusters" : "{...}",
  "host_redirect": "...",
  "path_redirect": "...",
  "prefix_rewrite": "...",
  "host_rewrite": "...",
  "auto_host_rewrite": "...",
  "case_sensitive": "...",
  "use_websocket": "...",
  "timeout_ms": "...",
  "runtime": "{...}",
  "retry_policy": "{...}",
  "shadow": "{...}",
  "priority": "...",
  "headers": [],
  "rate_limits": [],
  "include_vh_rate_limits" : "...",
  "hash_policy": "{...}",
  "request_headers_to_add" : [],
  "opaque_config": [],
  "cors": "{...}",
  "decorator" : "{...}"
}
```

- prefix

  (sometimes required, string) 如果指定，则路由使用匹配前缀规则，这意味着前缀必须匹配路径头的开始部分。

  注意：必须指定 `prefix` ， `path` 或 `regex` 其中之一。

- path

  (sometimes required, string) 如果指定，则路由采用一个精确的路径匹配规则，这意味着一旦路径字符串匹配，则将会被移除，

  注意：必须指定 `prefix` ， `path` 或 `regex` 其中之一。

- regex

  (sometimes required, string) 如果指定，则路由采用正则表达式匹配规则，这意味着一旦路径字符串匹配，则将会被移除，正则表达式必须匹配 `:path` 头。整个完整的路径（不含查询字符串）必须匹配正则表达式。如果只有 `:path` 头的子序列与正则表达式匹配，则规则上不匹配。这里定义了正则表达式语法。

  示例：

  - 正则表达式 /b[io]t 匹配路径 /bit
  - 正则表达式 /b[io]t 匹配路径 /bot
  - 正则表达式 /b[io]t 不匹配路径 /bite
  - 正则表达式 /b[io]t 不匹配路径 /bit/bot

  注意：必须指定 `prefix` ， `path` 或 `regex` 其中之一。

- cors

  (optional, object) 指定路由的CORS策略。

- cluster

  (sometimes required, string) 如果不是重定向路由（未指定 `host_redirect` 或 `path_redirect` ），则必须指定 `cluster` ， `cluster_header` 或 `weighted_clusters` 其中之一。指定群集时，其值指示请求应转发到的上游群集。

- cluster_header

  (sometimes required, string) 如果不是重定向路由（未指定 `host_redirect` 或 `path_redirect` ），则必须指定 `cluster` ， `cluster_header` 或 `weighted_clusters` 其中之一。当指定 `cluster_header` 时，Envoy将通过从请求头中读取由 `cluster_header` 命名的HTTP头的值，来明确要路由到的集群。如果相应的头没有找到，或者引用的群集不存在，Envoy将返回一个404响应。

  注意：在内部，Envoy始终使用HTTP/2的 `:authority` 头来表示HTTP/1主机头。因此，如果在试图匹配主机，应该匹配 `:authority` 头。

- weighted_clusters

  (sometimes required, object) 如果不是重定向路由（未指定 `host_redirect` 或 `path_redirect` ），则必须指定 `cluster` ， `cluster_header` 或 `weighted_clusters` 其中之一。使用 `weighted_clusters` 选项，可以为该路由指定多个上游群集。根据每个集群的权

重，将请求转发到其中一个上游集群。请参阅流量拆分以获取相应的文档。

- host_redirect

  (sometimes required, string) 表示该路由是重定向规则。如果匹配，则会发送301重定向的响应，该响应将使用该值交换URL的主机部分。也可以使用 `path_redirect` 选项一起指定。

- path_redirect

  (sometimes required, string) 表示该路由是重定向规则。如果匹配，则会发送一个301重定向的响应，用这个值交换URL的路径部分。可以与 `host_redirect` 选项一起指定。路由器过滤器将在重写 `x-envoy-original-path` 头之前放置原始路径。

- prefix_rewrite

  (optional, string) 表示在转发过程中，此值将替换所匹配的前缀（或路径）。当使用正则表达式路径匹配时，将替换整个路径（不包括查询字符串）。此选项允许应用程序的URL与反向代理层公开的路径不同。

- host_rewrite

  (optional, string) 表示在转发过程中，此值将替换主机头。

- auto_host_rewrite

  (optional, boolean) 表示在转发过程中，主机头将与群集管理器选择的上游主机的主机名进行交换。此选项仅适用于路由的目标群集的类型为 `strict_dns` 或 `logical_dns`。对于其他群集类型，设置为 `true` 将无效。 `auto_host_rewrite` 和 `host_rewrite` 是互斥的选项。只能指定一个。

- case_sensitive

  (optional, boolean) 表示前缀/路径匹配是否区分大小写。默认值是true。

- use_websocket

  (optional, boolean) 表示是否允许与此路由连接的HTTP/1.1客户端升级到WebSocket连接。默认值是false。

  注意：如果设置为true，Envoy期望与此路由匹配的第一个请求包含WebSocket升级头。如果升级头不存在，连接将作为普通的HTTP/1.1连接进行处理。如果升级头存在，Envoy将在客户端和上游服务器之间建立纯TCP代理。因此，如果要拒绝WebSocket升级请求，上游服务器将要负责关闭相关的连接。在关闭连接之前，Envoy将持续从客户端代理数据到上游服务器。

具有WebSocket升级头的请求不支持重定向，超时和重试。

- timeout_ms

  (optional, integer) 指定路由的超时时间。如果未指定，则默认值为15秒。请注意，此超时包括所有重试。另请参阅x-envoy-upstream-rq-timeout-ms，x-envoy-upstream-rq-per-try-timeout-ms和重试概述。

- runtime

  (optional, object) 可选，指定路由的运行时配置。

- retry_policy

  (optional, object) 可选，表示该路由具有重试策略。

- shadow

  (optional, object) 可选，表示路由具有影子策略。

- priority

  (optional, string) 可选，指定路由优先级。

- headers

  (optional, array) 指定与该路由匹配的一组头列表。路由器将根据配置中指定头检查请求的头部。如果路由中的所有头都与请求中头的值相同（或者配置中没有指定相应的值，则认定存在），则将发生匹配。

- request_headers_to_add

  (optional, array) 指定应添加到由此虚拟主机处理的每个请求的HTTP头列表。以下面的形式指定头部：
  `
  [

  ```
  {"key": "header1", "value": "value1"},
  {"key": "header2", "value": "value2"}
  ```

  ]
  `

有关更多信息，请参阅自定义请求头的文档。

- opaque_config

  (optional, array) 指定可由过滤器访问的一组可选路由配置值。

- rate_limits

  (optional, array) 指定可应用与该路由的一组速率限制配置。

- include_vh_rate_limits

  (optional, boolean) 指定速率限制过滤器是否应包含虚拟主机的速率限制。默认情况下，如果路由配置的速率限制，虚拟主机 `rate_limits` 将不适用于请求。

- hash_policy

  (optional, object) 如果上游群集使用哈希负载平衡器，则通过此选项指定路由的哈希策略。

- decorator

  (optional, object) 指定用于增强相关匹配的路由描述信息，用于信息上报。

Runtime

可用于路由的运行时配置，开展路由的逐步变更，而无需部署完整的代码/配置。请参阅流量转移文档。

```
{
  "key": "...",
  "default": "..."
}
```

- key

  (required, string) 指定运行时应查询的键的名称，以确定路由是否匹配。有关键名称如何映射到底层实现，请参阅运行时文档。

- default

  (required, integer) 一个0-100之间的整数，每当路由匹配时，会选择0-99之间的随机数。若该值<=在该键中找到的值（优先检查），或者若该键不存在，则为默认值，该路由是匹配的（假定所有的都路由匹配）。

Retry policy

HTTP重试架构概述。

```
{
  "retry_on": "...",
  "num_retries": "...",
  "per_try_timeout_ms" : "..."
}
```

- retry_on

  (required, string) 指定重试的发生条件。这些是与x-envoy-retry-on和x-envoy-retry-grpc-on记录的条件相同。

- num_retries

  (optional, integer) 指定允许的重试次数。此参数是可选的，默认值为1。这些与x-envoy-max-retries记录的条件相同。

- per_try_timeout_ms

  (optional, integer) 指定每个重试尝试的非零超时。该参数是可选的。与x-envoy-upstream-rq-per-try-timeout-ms记录的条件相同。

  注意：如果未指定，Envoy将使用全局路由请求超时。因此，当使用基于`5xx`的重试策略时，超时请求将不会被重试，因为总超时预算已经耗尽。

Shadow

路由器能够实现将流量从一个集群映射到另一个集群。目前的实现是“fire and forget”，这意味着Envoy在返回来自主集群的响应之前不会等待影子集群作出响应。所有正常的统计数据都会被收集用于阴影集群，使得该功能对测试很有用。

在shadow期间，host/authority头被改变，使得`-shadow`被附加。这对于日志记录很有用。例如：`cluster1`变为`cluster1-shadow`。

```
{
  "cluster": "...",
  "runtime_key": "..."
}
```

- cluster

  (required, string) 指定请求将被映射到的群集。群集必须存在于群集管理器配置中。

- runtime_key

  (optional, string) 如果未指定，则对目标群集的所有请求都将被映射。如果指定，Envoy将查找运行时键，以获取请求的百分比。有效值从0到10000，允许0.01%的增幅。如果运行时键在配置中指定，但运行时不存在，则默认为0，因此请求的0%将被映射。

Headers

```
{
  "name": "...",
  "value": "...",
  "regex": "..."
}
```

- name

  (required, string) 指定请求中头部的名称。

- value

  (optional, string) 指定头的值。如果值不存在，则具有头名的请求都将匹配，而不管头的值如何。

- regex

  (optional, boolean) 指定头的值是否采用正则表达式。默认为`false`。整个请求头的值必须与正则表达式匹配。如果只有请求头值的子序列与正则表达式匹配，则规则不匹配。这里定义了值字段中使用的正则表达式语法。

  示例：

  - 正则表达式d{3}匹配123值
  - 正则表达式d{3}不匹配1234值
  - 正则表达式d{3}不匹配123.456值

  注意：在内部，Envoy始终使用HTTP/2的`:authority`来表示HTTP/1主机头。因此，如果试图匹配主机，则匹配`:authority`替代。

  注意：若要使用HTTP的`method`进行路由，请使用特殊的HTTP/2的`:method`头。这适用于HTTP/1和HTTP/2，因为Envoy标准化头。例如：

  `
  {

  ```
    "name": ":method",
  ```

```
  "value": "POST"
}
`
```

Weighted Clusters

与指定单个上游群集作为请求的目标群集相比，`weighted_clusters`选项允许指定多个上游群集，以及指定要转发给每个群集的流量的百分比权重。路由器将根据权重选择上游集群。

```
{
  "clusters": [],
  "runtime_key_prefix" : "..."
}
```

- clusters

  (required, array) 指定与路由器相关的一个或多个上游群集。

  `
  {

  ```
  "name" : "...",
  "weight": "..."
  ```

  }
  `

  - name

    (required, string) 上游群集的名称。群集必须存在于群集管理器配置中。

  - weight

    (required, integer) 一个0-100之间的整数，当请求与路由匹配时，选择上游集群的权重比。集群组中所有权重之和必须为100。

- runtime_key_prefix

  (optional, string) 指定运行时每个群集相关的运行时键的前缀。当指定`runtime_key_prefix`时，路由器将在键为`runtime_key_prefix + "." + cluster[i].name`下查找与每个上游集群相关的权重。其中，`cluster[i]`表示集群组中的条目。如果群集的运行时键不存在，则配置文件中指定的值将用作默认权重。有关键名称如何映射到底层实现，请参阅运行时文档。

注意：如果运行时权重之和超过100，则流量拆分的行为将未定义（尽管请求将被路由到其中一个集群）。

Hash policy

如果上游群集使用哈希负载平衡器，则指定路由的哈希策略。

```
{
  "header_name": "..."
}
```

- header_name

  (required, string) 获取的请求头部的名称作为哈希值。如果请求头不存在，负载均衡器将使用一个随机数作为哈希值，从而使得有效地使负载均衡策略变为随机策略。

Decorator

指定路由的修饰符。

```
{
  "operation": "..."
}
```

- operation

  (required, string) 与此路由匹配的请求相关的操作名称。如果已启用跟踪，则将使用此信息作为为此请求跟踪记录的span名称。注意：对于入口（入站）请求或出口（出站）响应，此值可能会被 `x-envoy-decorator-operation` 头部覆盖。

Opaque Config

可以通过“不透明配置”机制为过滤器提供额外的配置。在路由配置中指定属性列表。该配置不能未被Envoy解释，可以在用户定义的过滤器中访问。该配置是一个通用的字符串映射。不支持嵌套对象。

```
[
  {"...": "..."}
]
```

Cors

设置路由的优先于虚拟主机的设置。

```
{
  "enabled": false,
  "allow_origin": ["http://foo.example"],
  "allow_methods": "POST, GET, OPTIONS",
```

```
    "allow_headers": "Content-Type",
    "allow_credentials": false,
    "expose_headers": "X-Custom-Header",
    "max_age": "86400"
  }
```

- enabled

  (optional, boolean) 默认为true。只有在路由上启用并设置为false，才会禁用此路由的CORS。该设置对虚拟主机没有影响。

- allow_origin

  (optional, array) 将允许做CORS请求的来源。通配符“*”表示允许任何来源。

- allow_methods

  (optional, string) `access-control-allow-methods` 头的内容。以逗号分隔的HTTP方法列表。

- allow_headers

  (optional, string) `access-control-allow-headers` 头的内容。以逗号分隔的HTTP标题列表。

- allow_credentials

  (optional, boolean) 资源是否允许凭证。

- expose_headers

  (optional, string) `access-control-expose-headers` 头的内容。以逗号分隔的HTTP标题列表。

- max_age

  (optional, string) `access-control-max-age` 头的内容。以秒为单位的值，可以缓存对预检请求的响应时间。

返回

- 上一级
- 首页目录

虚拟集群

虚拟集群

虚拟集群是一种通过正则表达式匹配相应端口的方法，例如为匹配的请求显式生成一些统计信息。在使用前缀/路径匹配时很有用，Envoy并不总是知道应用程序认为是一个端口。因此，Envoy不可能统往每个端口发送统计数据。然而，系统中往往需要具有高度关联性的端口，他们希望获得“完美”的统计数据。虚拟集群统计是完美的，针对下游散发，包含了网络级别的故障。

注意：虚拟集群是一个有用的工具，但我们不建议为每个应用程序端口设置一个虚拟集群。因为这不容易维护，因为匹配和统计输出是由代价的。

```
{
  "pattern": "...",
  "name": "...",
  "method": "..."
}
```

- pattern

  (required, string) 指定用于匹配请求的正则表达式模型。整个请求的路径必须与正则表达式匹配。所使用的正则表达式语法都在这里定义。

- name

  (required, string) 指定的虚拟群集名称。发布统计信息时会使用虚拟群集名称和虚拟主机名称。统计信息将由路由器过滤器发出，并记录在此处。

- method

  (optional, string)（可选）指定要匹配的HTTP方法。例如GET，PUT等

  示例：

  - 正则表达式 /rides/d+ 匹配路径 /rides/0
  - 正则表达式 /rides/d+ 匹配路径 /rides/123
  - 正则表达式 /rides/d+ 不匹配路径 /rides/123/456

虚拟集群统计信息的文档。

返回

- 上一级

- 首页目录

速率限制配置

速率限制配置

全局速率限制架构概述。

```
{
  "stage": "...",
  "disable_key": "...",
  "actions": []
}
```

- stage

  (optional, integer) 指在过滤器中设置的阶段。速率限制配置仅适用于具有相同阶段编号的过滤器。默认的阶段编号是0。

  注意：对于阶段编号，过滤器支持0-10的范围。

- disable_key

  (optional, string) 在运行时设置的key，用于禁用此速率限制配置。

- actions

  (required, array) 将应用于此速率限制配置的操作列表。顺序很重要，因为这些操作是按顺序处理的，描述符是通过在该顺序中附加描述符条目来组成的。如果某个操作无法附加描述符条目，则不会为该配置生成描述符。请参阅相关操作文档。

Actions

```
{
  "type": "..."
}
```

- type

  (required, string) 要执行的速率限制操作的类型。当前支持的操作类型是 `source_cluster` ，`destination_cluster` ，`request_headers` ，`remote_address` ，`generic_key` 和 `header_value_match` 。

Source Cluster

```
{
  "type": "source_cluster"
}
```

以下描述符条目被追加到描述符中：

```
("source_cluster", "<local service cluster>")
```

`<local service cluster>` 是从 `--service-cluster` 选项派生出来的.

Destination Cluster

```
{
  "type": "destination_cluster"
}
```

以下描述符条目被追加到描述符中：

```
("destination_cluster", "<routed target cluster>")
```

一旦请求与路由表规则匹配，路由的集群就由以下路由表配置设置之一确定：

- cluster 指定要路由到的上游群集。
- weighted_clusters 从一组具有权重属性的集群组中随机选择一个集群。
- cluster_header 指定从请求中的头部获取目标群集。

Request Headers

```
{
  "type": "request_headers",
  "header_name": "...",
  "descriptor_key" : "..."
}
```

- header_name

  (required, string) 要从请求头中查询的该头的名称。头的值用于填充 `descriptor_key` 的描述符条目的值。

- descriptor_key

  (required, string) 在描述符条目中使用的关键。

当一个头包含一个与 `header_name` 匹配的关键字时，附加下面的描述符条目：

```
("<descriptor_key>", "<header_value_queried_from_header>")
```

Remote Address

```
{
  "type": "remote_address"
}
```

以下描述符条目被追加到描述符中，并使用来自x-forwarded-for的可信地址填充：

```
("remote_address", "<trusted address from x-forwarded-for>")
```

Generic Key

```
{
  "type": "generic_key",
  "descriptor_value" : "..."
}
```

- descriptor_value

  (required, string) 描述符条目中使用的值。

以下描述符条目被追加到描述符中：

```
("generic_key", "<descriptor_value>")
```

Header Value Match

```
{
  "type": "header_value_match",
  "descriptor_value" : "...",
  "expect_match" : "...",
  "headers" : []
}
```

- descriptor_value

  (required, string) 描述符条目中使用的值。

- expect_match

(optional, boolean) 如果设置为true，则该操作将在请求与头部匹配时附加描述符条目。如果设置为false，则该操作将在请求与头部不匹配时附加描述符条目。默认值是true。

- headers

(required, array) 指定速率限制操作应匹配的一组头。将检查请求的头部与配置中所有指定的头部。如果配置中的所有报头都存在于具有相同值的请求中（或者如果没有配置`value`字段，则认为存在），则匹配将发生。

以下描述符条目被追加到描述符中`: .. code-block:: cpp`

```
("header_match", "<descriptor_value>")
```

返回

- 上一级
- 首页目录

路由发现服务

路由发现服务(RDS)

```
{
  "cluster": "...",
  "route_config_name": "...",
  "refresh_delay_ms": "..."
}
```

- cluster

  (required, string) 承载路由发现服务的上游群集的名称。群集必须实现和运行RDS HTTP API的REST服务。注：这是在群集管理器配置中定义的群集的名称，而不是群集的完整定义，如SDS和CDS的情况。

- route_config_name

  (required, string) 路由配置的名称。这个名字将被传递给RDS HTTP API。这允许具有多个HTTP监听器（和关联的HTTP连接管理器过滤器）的Envoy配置使用不同的路由配置。默认情况下，名称的最大长度限制为60个字符。通过 `--max-obj-name-len` 命令行参数设置为所需的值，可以提高此限制。

- refresh_delay_ms

  (optional, integer) 每次从RDS API提取的时间间隔（以毫秒为单位）。Envoy将在0和 `refresh_delay_ms` 之间的增加一个额外的随机抖动。因此，最长可能的刷新间隔是 `2*refresh_delay_ms` 。默认值是30000ms（30秒）。

REST API

```
GET /v1/routes/(string: route_config_name)/(string: service_cluster)/(string: service_node)
```

请求路由发现服务返回特定 `route_config_name` ， `service_cluster` 和 `service_node` 的路由配置。 `route_config_name` 对应于上面的RDS配置参数。 `service_cluster` 对应于 `--service-cluster` 命令行选项。 `service_node` 对应于 `--service-node` 命令行选项。响应是单个JSON对象，其中包含路由配置文档中定义的路由配置。

新的路由配置将被优雅地交换，使得现有的请求不受影响。这意味着当一个请求开始时，它会看到一个一致的快照，在请求的持续时间内不会改变路由配置。因此，例如，如果更新更改超时值，则只有新的请求将使用更新后的值。

作为性能优化，Envoy对从RDS API接收的路由配置进行哈希散列处理，并且只有在哈希值发生变化时才会执行完整的重新加载。

注意：通过RDS加载的路由配置不会检查所引用的集群是否已经在集群管理器加载。RDS API被设计为与CDS API一起工作，使得Envoy最终采取一致的更新。如果一个路由引用一个未知的集群，路由器过滤器将返回一个404响应。

返回

- 上一级
- 首页目录

HTTP过滤器

HTTP过滤器

返回

- 上一级
- 首页目录

缓存

Buffer

缓冲区配置概述。

```
{
  "name": "buffer",
  "config": {
    "max_request_bytes": "...",
    "max_request_time_s": "..."
  }
}
```

- max_request_bytes

  (required, integer) 在连接管理器停止缓冲并返回413响应之前，过滤器将缓冲的最大请求大小。

- max_request_time_s

  (required, integer) 过滤器在返回408响应之前等待完整请求的最大秒数。

返回

- 上一级
- 首页目录

CORS过滤器

CORS过滤器

Cors过滤器的配置概述。

```
{
  "name": "cors",
  "config": {}
}
```

返回

- 上一级
- 首页目录

DynamoDB

DynamoDB

DynamoDB配置概述。

```
{
  "name": "http_dynamo_filter",
  "config": {}
}
```

- name

  (required, string) 过滤器名称。目前唯一支持的值是`http_dynamo_filter`。

- config

  (required, object) 该过滤器不使用任何配置。

返回

- 上一级
- 首页目录

故障注入

故障注入

故障注入配置概述。

Configuration

```
{
  "name" : "fault",
  "config" : {
    "abort" : "{...}",
    "delay" : "{...}",
    "upstream_cluster" : "...",
    "headers" : [],
    "downstream_nodes" : []
  }
}
```

- abort

  (sometimes required, object) 如果指定，过滤器将根据对象中的值中止请求。必须指定至少 `abort` 或 `delay`。

- delay

  (sometimes required, object) 如果指定，过滤器将根据对象中的值注入延迟。必须指定至少 `abort` 或 `delay`。

- upstream_cluster

  (optional, string) 指定过滤器应匹配的（目标）上游群集的名称。故障注入将限于绑定到特定上游群集的请求。

- headers

  (optional, array) 指定过滤器应匹配的一组标题。故障注入过滤器可以选择一组头相匹配的请求，应用该故障注入。实际故障注入的机会进一步取决于abort_percent和fixed_delay_percent参数的值。过滤器会根据配置中的所指定头部，来检查请求的头。如果配置中的所有头都存在于具有相同值的请求中（或者如果没有配置 `value` 字段，则认为存在），则匹配将发生。

- downstream_nodes

  (optional, array) 针对指定的下游主机列表，注入故障。如果未设置此设置，则会为所有下游节点注

入故障。下游节点名称取自HTTP `x-envoy-downstream-service-node头，并与` downstream_nodes`列表进行比较。

Abort

```
{
  "abort_percent" : "...",
  "http_status" : "..."
}
```

- abort_percent

  (required, integer) 使用指定的 `http_status` 代码中止请求的百分比。有效值范围从0到100。

- http_status

  (required, integer) 将被用作中止请求的响应码，即HTTP状态代码。

Delay

```
{
  "type" : "...",
  "fixed_delay_percent" : "...",
  "fixed_duration_ms" : "..."
}
```

- type

  (required, string) 指定被注入的延迟类型。目前只支持 `fixed` 延迟类型（ `step function` ）。

- fixed_delay_percent

  (required, integer) 将在 `fixed_duration_ms` 指定的时间内延迟的请求的百分比。有效值范围从0到100。

- fixed_duration_ms

  (required, integer) 延迟时间，以毫秒为单位。必须大于0。

返回

- 上一级
- 首页目录

gRPC HTTP/1.1 桥接

gRPC HTTP/1.1 桥接

gRPC HTTP/1.1 桥接的配置概述。

```
{
  "name": "grpc_http1_bridge",
  "config": {}
}
```

返回

- 上一级
- 首页目录

gRPC-JSON 转码过滤器

gRPC-JSON转码过滤器

gRPC-JSON转码配置概述。

gRPC-JSON转码配置

过滤器配置所需要描述符文件以及要转码的gRPC服务列表。

```
{
  "name": "grpc_json_transcoder",
  "config": {
    "proto_descriptor": "proto.pb",
    "services": ["grpc.service.Service"],
    "print_options": {
      "add_whitespace": false,
      "always_print_primitive_fields": false,
      "always_print_enums_as_ints": false,
      "preserve_proto_field_names": false
    }
  }
}
```

- proto_descriptor

  (required, string) 为gRPC服务提供二进制 `protobuf` 描述符集。描述符集必须包含在服务中使用的所有类型。确保为 `protoc` 使用 `--include_import` 选项。

```
要为gRPC服务生成一个`protobuf`描述符集，在运行`protoc`之前，还需要从Github中克隆`googleapis`仓库，因为在`include`路径中需要`annotations.proto`。

```
git clone https://github.com/googleapis/googleapis
GOOGLEAPIS_DIR=<your-local-googleapis-folder>
```

然后运行`protoc`从`bookstore.proto`生成描述符集：

```
protoc -I$(GOOGLEAPIS_DIR) -I. --include_imports --include_source_info \
  --descriptor_set_out=proto.pb test/proto/bookstore.proto
```
如果您有多个原始源文件，您可以在一个命令中传递所有这些文件。
```

- services

  (required, array) 提供代码转换器进行转码服务的服务器名称列表。如果服务名称在 `proto_descriptor` 中不存在，Envoy将启动失败。 `proto_descriptor` 可能包含比这里指定

的服务名称更多的服务，但是它们不会进行转码。

- print_options

  (optional, object) 响应json的控制选项。这些选项直接传递给JsonPrintOptions。有效的选项是：

  - add_whitespace

    (optional, boolean) 是否添加空格，换行符和缩进以使输出的JSON易于阅读。默认为false。

  - always_print_primitive_fields

    (optional, boolean) 是否始终打印原始字段。默认情况下，具有默认值的原始字段将在JSON输出中被省略。例如，设置为0的 `int32` 字段将被省略。将此标志设置为true，将覆盖默认行为并打印原始字段，而不考虑其值。默认为false。

  - always_print_enums_as_ints

    (optional, boolean) 是否始终打印枚举作为整数。默认情况下，它们呈现为字符串。默认为false。

  - preserve_proto_field_names

    (optional, boolean) 是否保留原始字段名称。默认情况下， `protobuf` 将使用 `json_name` 选项生成JSON字段名称，或者按照下面的顺序生成较低的骆驼大小写。设置此标志将保留原始字段名称。默认为false。

返回

- 上一级
- 首页目录

gRPC-Web 过滤器

gRPC-Web过滤器

gRPC-Web过滤器配置概述。

```
{
  "name": "grpc_web",
  "config": {}
}
```

返回

- 上一级
- 首页目录

健康检查

健康检查

健康检查配置概述。

```
{
  "name": "health_check",
  "config": {
    "pass_through_mode": "...",
    "endpoint": "...",
    "cache_time_ms": "..."
   }
}
```

- pass_through_mode

  (required, boolean) 指定过滤器是否在通过模式下运行。

- endpoint

  (required, string) 指定健康检查的HTTP API端口。例如`/healthcheck`。

- cache_time_ms

  (optional, integer) 如果在通过模式下运行，则过滤器将缓存上游响应的时间（以毫秒为单位）。

返回

- 上一级
- 首页目录

Lua

Lua

Lua配置概述。

```
{
  "name": "lua",
  "config": {
    "inline_code": "..."
  }
}
```

- inline_code

  (required, string) Envoy将执行的Lua代码。这可以是一个非常小的脚本，如果需要，可以从磁盘进一步加载代码。请注意，如果使用JSON配置，则代码必须能够正确转义。YAML配置可能更容易阅读，因为YAML支持多行字符串，可以很容易地在配置中内联表示复杂的脚本。

返回

- 上一级
- 首页目录

速率限制

速率限制

速率限制配置概述。

```
{
  "name": "rate_limit",
  "config": {
    "domain": "...",
    "stage": "...",
    "request_type": "...",
    "timeout_ms": "..."
  }
}
```

- domain

  (required, string) 调用速率限制服务时使用的域。

- stage

  (optional, integer) 指定要应用于相同阶段编号的速率限制配置。如果未设置，则默认阶段编号为0。

  注意:对于阶段编号，过滤器支持0-10的范围。

- request_type

  (optional, string) 该过滤器适用的请求类型。支持的类型有 `internal` ， `external` 或者 `both` 。如果将 `x-envoy-internal` 设置为 `true` ，则将请求视为内部请求。如果 `x-envoy-internal` 未设置或为 `false` ，则请求被视为外部。过滤器默认为 `both` ，它将应用于所有的请求类型。

- timeout_ms

  (optional, integer) 速率限制服务的RPC超时时间（以毫秒为单位）。如果未设置，则默认为20ms。

返回

- 上一级
- 首页目录

路由

路由

路由配置概述

```
{
  "name": "router",
  "config": {
    "dynamic_stats": "...",
    "start_child_span": "..."
  }
}
```

- dynamic_stats

  (optional, boolean) 是否为动态集群
  生成统计信息。默认为true。可以在高性能场景下禁用。

- start_child_span

  (optional, boolean) 是否为出站路由的调用启动子跟踪
  ，在其他过滤器（auth，ratelimit等）进行出站调用，并且子父span位于同一入口的情况下，这可能很有用。默认为false。

返回

- 上一级
- 首页目录

集群管理

集群管理

集群管理架构概述。

```
{
  "clusters": [],
  "sds": "{...}",
  "local_cluster_name": "...",
  "outlier_detection": "{...}",
  "cds": "{...}"
}
```

- clusters

  (required, array) 群集管理器将执行服务发现，健康检查和负载平衡的上游群集列表。

- sds

  (sometimes required, object) 如果任何集群定义使用sds集群发现类型，则必须指定全局SDS配置。

- local_cluster_name

  (optional, string) 本地集群的名称（即拥有运行此配置的Envoy集群）。若要启用区域感知路由，必须设置此选项。如果定义了`local_cluster_name`，则群集必须包含具有相同名称的群集定义。

- outlier_detection

  (optional, object) 用于异常值检测的可选全局配置。

- cds

  (optional, object) 用于群集发现服务（CDS）API的可选配置。

子章节

- 集群
- 异常检测
- 集群发现服务
- 服务发现服务

返回

- 上一级
- 首页目录

集群

集群

```
{
  "name": "...",
  "type": "...",
  "connect_timeout_ms": "...",
  "per_connection_buffer_limit_bytes": "...",
  "lb_type": "...",
  "ring_hash_lb_config": "{...}",
  "hosts": [],
  "service_name": "...",
  "health_check": "{...}",
  "max_requests_per_connection": "...",
  "circuit_breakers": "{...}",
  "ssl_context": "{...}",
  "features": "...",
  "http2_settings": "{...}",
  "cleanup_interval_ms": "...",
  "dns_refresh_rate_ms": "...",
  "dns_lookup_family": "...",
  "dns_resolvers": [],
  "outlier_detection": "{...}"
}
```

- name

  (required, string) 群集名称，所提供群集名称在所有群集中必须唯一。发布统计信息时会使用集群名称。默认情况下，群集名称的最大长度限制为60个字符。可通过 `--max-obj-name-len` 命令行参数设置为所需的值，以提高此限制。

- type

  (required, string) 用于解析集群的服务发现类型。可能的选项有 `static` ， `strict_dns` ， `logical_dns` ， `original_dst` 和 `sds` 。

- connect_timeout_ms

  (required, integer) 指定群集中主机网络连接超时时间，以毫秒为单位。

- per_connection_buffer_limit_bytes

  (optional, integer) 集群连接的读写缓冲区大小限制。如果未指定，则默认值（1MB）。

- lb_type

(required, string) 在集群中选择主机时使用的负载均衡器类型。可能的选项有 `round_robin` ，`least_request` ，`ring_hash` ，`random` 和 `original_dst_lb` 。请注意，`original_dst_lb` 必须与 `original_dst` 类型的集群一起使用，并且不能与其他任集群类型一起使用。

- ring_hash_lb_config

  (optional, object) 可选的环哈希负载均衡器配置，当 `lb_type` 设置为 `ring_hash` 时使用。

- hosts

  (sometimes required, array) 如果服务发现类型是 `static` ，`strict_dns` 或 `logical_dns` ，则主机数组是必需的。主机数组不支持 `original_dst` 集群类型。指定方式由服务发现类型决于：

  - static

    静态集群，必须使用完整解析主机，不需要DNS查找。支持TCP和unix域套接字（UDS）地址。

    一个TCP地址如下所示：

    ```
    tcp://<ip>:<port>
    ```

    一个UDS地址如下所示：

    ```
    unix://<file name>
    ```

    以下示例指定地址列表：

    ```
    [{"url": "tcp://10.0.0.2:1234"}, {"url": "tcp://10.0.0.3:5678"}]
    ```

  - strict_dns

    严格DNS群集类型，可以指定任意数量的主机名、端口组合。将使用DNS解析所有名称，并组合在一起形成最终的群集。如果相同的名字返回多个记录，则所有将被使用。例如：

    ```
    [{"url": "tcp://foo1.bar.com:1234"}, {"url": "tcp://foo2.bar.com:567
    ```

  - logical_dns

    逻辑DNS群集类型，指定主机名称方式与严格DNS非常类似，但只有第一个主机将被使用。例

如：

```
[{"url": "tcp://foo1.bar.com:1234"}]
```

- service_name

  (sometimes required, string) 如果服务发现类型是sds，则此参数是必需的。当获取集群成员时，它将被传递给SDS API。

- health_check

  (optional, object) 可选的群集主动健康检查配置。如果未指定配置，则不会执行健康检查，并且所有群集成员都将始终处于健康状态。

- max_requests_per_connection

  (optional, integer) 可选的单个上游连接的最大请求数。HTTP/1.1和HTTP/2连接池，都遵循此参数。如果没有指定，则没有限制。将此参数设置为1，将有效地禁用保活状态的请求。

- circuit_breakers

  (optional, object) 可选的群集熔断配置。

- ssl_context

  (optional, object) 上游群集的TLS配置。如果没有指定TLS配置，则新的连接将不会使用TLS。

- features

  (optional, string) 上游群集支持的功能，以逗号分隔的字符串列表。目前支持的功能为：

  - http2

    如果指定了http2，Envoy将假定在建立新的HTTP连接时，支持HTTP/2。目前，Envoy仅支持上游连接的预知。即使TLS与ALPN一起使用，也必须指定http2。另外，这允许通过纯文本建立HTTP/2连接。

- http2_settings

  (optional, object) 在启动HTTP连接池时，直接传递给HTTP/2编解码器的相关设置。这些与HTTP连接管理器http2_settings选项相同。

- cleanup_interval_ms

  (optional, integer) 从 `original_dst` 集群类型中删除旧主机的时间间隔。如果主机在这段时间内，没有被用作上游目的地址，则认为它们是旧的。随着重定向到Envoy的新连接，新的主机被按需添加到原始目标集群中，从而导致集群中的主机数量随着时间而增长。若没有旧主机（它们被主动用作目的地址）被保存在群集中，从而允许与它们的连接保持打开状态，从而节省了打开新连接所花费的等待时间。如果未指定此设置，则默认为5000.对于 `original_dst` 以外的群集类型，此设置将被忽略。

- dns_refresh_rate_ms

  (optional, integer) 如果指定了dns刷新率，并且群集类型为 `strict_dns` 或 `logical_dns` ，则将此值用作群集的dns刷新率。如果未指定此设置，则该值默认为5000。对于 `strict_dns` 和 `logical_dns` 以外的群集类型，此设置将被忽略。

- dns_lookup_family

  (optional, string) DNS IP地址解析策略。选项为 `v4_only` ， `v6_only` 和 `auto` 。如果未指定此设置，则该值默认为 `v4_only` 。当选择 `v4_only` 时，DNS解析器将仅执行IPv4系列的地址查找。如果选择 `v6_only` ，则DNS解析程序将仅执行IPv6系列的地址查找。如果指定了 `auto` ，则DNS解析器将首先执行IPv6系列的地址查找操作，并然会再回到IPv4系列的地址查找。对于 `strict_dns` 和 `logical_dns` 以外的集群类型，该设置将被忽略。

- dns_resolvers

  (optional, array) 如果指定DNS解析程序，并且群集类型是 `strict_dns` 或 `logical_dns` ，则此值用于指定群集的dns解析程序。如果未指定此设置，则该值默认为使用 `/etc/resolv.conf` 配置的默认解析器。对于除 `strict_dns` 和 `logical_dns` 以外的集群类型，此设置将被忽略。

- outlier_detection

  (optional, object) 如果指定，则会为此上游群集启用异常值检测。有关异常值检测的更多信息，请参阅相应架构概述。

子章节

- 健康检查
- 熔断
- TLS上下文
- 异常值检测
- 环哈希负载均衡配置

返回

- 上一级
- 首页目录

健康检查

健康检查

- 健康检查架构概述。
- 如果为集群配置了健康检查，则会发出相应的统计信息。并且记录在这里。

```
{
  "type": "...",
  "timeout_ms": "...",
  "interval_ms": "...",
  "unhealthy_threshold": "...",
  "healthy_threshold": "...",
  "path": "...",
  "send": [],
  "receive": [],
  "interval_jitter_ms": "...",
  "service_name": "..."
}
```

- type

  (required, string) 健康检查的类型。目前支持的类型有 `http` ， `redis` 和 `tcp` 。请参阅架构概述以获取更多信息。

- timeout_ms

  (required, integer) 等待健康检查响应的时间（以毫秒为单位）。如果达到超时时间，则该健康检查将被视为失败。

- interval_ms

  (required, integer) 每次健康检查的时间间隔，以毫秒为单位。

- unhealthy_threshold

  (required, integer) 在主机被标记为不健康之前，需要进行健康检查次数。请注意，对于 `http` 健康检查类型，如果主机响应503，则此阈值将被忽略，并且主机立即被视为不健康。

- healthy_threshold

  (required, integer) 在主机被标记为健康之前，需要进行健康检查次数。请注意，在启动过程中，只需要一次成功的健康检查即可将主机标记为健康状态。

- path

  (sometimes required, string) 如果是`http`类型，则此参数是必需的。它会在健康检查过程中，请求的`HTTP`路径。例如`/healthcheck`。

- send

  (sometimes required, array) 如果是`tcp`类型，则此参数是必需的。它指定了为健康检查请求发送的字节。如下例所示，它是一个十六进制字符串数组：

  `
  [

  ```
  {"binary": "01"},
  {"binary": "000000FF"}
  ```

  ]
  `

  在"connect only"健康检查的情况下，数组允许为空。

- receive

  (sometimes required, array) 如果是tcp类型，则此参数是必需的。它指定了成功的健康检查响应中预期的字节。它是一个与`send`参数指定类似的十六进制字符串数组。 在"connect only"健康检查的情况下，数组允许为空。

- interval_jitter_ms

  (optional, integer) 可选的抖动量（以毫秒为单位）。如果指定的话，Enovy在每个间隔内，都会添加0到`interval_jitter_ms`的等待时间。

- service_name

  (optional, string) 可选的服务名称参数，用于验证健康检查的群集的身份。请参阅架构概述以获取更多信息。

返回

- 上一级
- 首页目录

熔断

熔断

- 熔断架构概述。
- 优先级路由架构概述。

可以为每个优先级的定义单独指定熔断设置。关于不同优先级如何使用，详见[配置指南]章节。

```
{
  "default": "{...}",
  "high": "{...}"
}
```

- default

  (optional, object) 设置默认优先级的配置对象。

- high

  (optional, object) 设置高优先级的配置对象。

优先级设置

```
{
  "max_connections": "...",
  "max_pending_requests": "...",
  "max_requests": "...",
  "max_retries": "...",
}
```

- max_connections

  (optional, integer) Envoy将允许上游群集的最大连接数。如果未指定，则默认值为1024。

- max_pending_requests

  (optional, integer) Envoy将允许上游集群的最大待处理请求数。如果未指定，则默认值为1024。

- max_requests

  (optional, integer) Envoy将对上游群集执行的最大并行请求数。如果未指定，则默认值为1024。

- max_retries

(optional, integer) Envoy允许上游集群执行的最大并行重试次数。如果未指定，则默认值为3。

有关更多信息，请参阅熔断概述。

返回

- 上一级
- 首页目录

TLS上下文

TLS上下文

```
{
  "alpn_protocols": "...",
  "cert_chain_file": "...",
  "private_key_file": "...",
  "ca_cert_file": "...",
  "verify_certificate_hash": "...",
  "verify_subject_alt_name": [],
  "cipher_suites": "...",
  "ecdh_curves": "...",
  "sni": "..."
}
```

- alpn_protocols

  (optional, string) 提供请求连接的ALPN协议列表。在实践中，这可能会被设置为一个单一的值或根本不设置：

  - "h2"：指定上游连接使用HTTP/2。在当前的实现中，这必须与集群的 `http2` 功能选项一起设置。这两个选项一起使用ALPN，来告诉Envoy服务器，期望支持HTTP/2的ALPN。然后http2功能将导致新的连接使用HTTP/2。

- cert_chain_file

  (optional, string) 用于连接使用的证书文件链。这用于上游主机提供给客户端的TLS证书。

- private_key_file

  (optional, string) 与证书文件链相对应的私钥。

- ca_cert_file

  (optional, string) 包含证书颁发机构的证书文件，用于验证服务器提供的证书。

- verify_certificate_hash

  (optional, string) 如果指定，Envoy将验证（pin）所服务器提供的证书哈希。

- verify_subject_alt_name

  (optional, array) 可选的标题替代名称列表。如果指定，Envoy将验证服务器证书的 `alt` 标题名称是

否与指定值之一匹配。

- cipher_suites

  (optional, string) 如果指定，连接将支持指定的加密套件列表。如果未指定，则默认列表：

```
[ECDHE-ECDSA-AES128-GCM-SHA256|ECDHE-ECDSA-CHACHA20-POLY1305]
[ECDHE-RSA-AES128-GCM-SHA256|ECDHE-RSA-CHACHA20-POLY1305]
ECDHE-ECDSA-AES128-SHA256
ECDHE-RSA-AES128-SHA256
ECDHE-ECDSA-AES128-SHA
ECDHE-RSA-AES128-SHA
AES128-GCM-SHA256
AES128-SHA256
AES128-SHA
ECDHE-ECDSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES256-SHA384
ECDHE-RSA-AES256-SHA384
ECDHE-ECDSA-AES256-SHA
ECDHE-RSA-AES256-SHA
AES256-GCM-SHA384
AES256-SHA256
AES256-SHA
```

将会被使用；

- ecdh_curves

  (optional, string) 如果指定，TLS连接将只支持指定的ECDH曲线算法。如果未指定，将使用默认（X25519，P-256）曲线算法。

- sni

  (optional, string) 如果指定，则字符串将在TLS握手期间作为SNI呈现。

返回

- 上一级
- 首页目录

异常值检测

异常值检测

```
{
  "consecutive_5xx": "...",
  "consecutive_gateway_failure": "...",
  "interval_ms": "...",
  "base_ejection_time_ms": "...",
  "max_ejection_percent": "...",
  "enforcing_consecutive_5xx" : "...",
  "enforcing_consecutive_gateway_failure" : "...",
  "enforcing_success_rate" : "...",
  "success_rate_minimum_hosts" : "...",
  "success_rate_request_volume" : "...",
  "success_rate_stdev_factor" : "..."
}
```

- consecutive_5xx

  (optional, integer) 发生连续5xx逐出主机之前，连续5xx响应的数量。默认为5。

- consecutive_gateway_failure

  (optional, integer) 逐出之前连续发生的连续“gateway errors”数量，包括（502,503,504状态或连接错误，映射到其中一个状态代码）默认为5。

- interval_ms

  (optional, integer) 每次异常值分析扫描的时间间隔，这可能导致新抛出异常以及主机被重新添加到服务集群。默认为10000ms或10s。

- base_ejection_time_ms

  (optional, integer) 主机被逐出的基准时间。实际时间等于基本时间乘以主机被逐出的次数。默认为30000ms或30s。

- max_ejection_percent

  (optional, integer) 由于异常检测而逐出的主机占上游群集的最大百分比。默认为10%。

- enforcing_consecutive_5xx

  (optional, integer) 当通过连续5xx检测到异常状态时，主机实际被逐出的几率百分比。这个设置可

以用来禁止逐出或者缓慢地加速。默认为100。

- enforcing_consecutive_gateway_failure

  (optional, integer) 当通过连续的网关故障检测到异常状态时，主机实际被逐出的几率百分比。这个设置可以用来禁止逐出或者缓慢地加速。默认为0。

- enforcing_success_rate

  (optional, integer) 通过成功率统计检测到异常状态时，主机实际被逐出的几率百分比。这个设置可以用来禁止逐出或者缓慢地加速。默认为100。

- success_rate_minimum_hosts

  (optional, integer) 必须具有足够的请求量来检测成功率异常值的群集中的主机数量。如果主机数量小于此设置，则不会为群集中的任何主机执行通过成功率统计信息的异常值检测。默认为5。

- success_rate_request_volume

  (optional, integer) 在一个时间间隔内（如上述定义的时间间隔）必须收集的最小请求总数，以便将此主机包含在基于成功率的异常值检测中。如果低于此设置，则不会为该主机执行通过成功率统计的异常值检测。默认为100。

- success_rate_stdev_factor

  (optional, integer) 这个因子被用来确定异常逐出成功率的阈值。逐出阈值是平均成功率与该因子与平均成功率的标准偏差的乘积之差：`mean-(stdev* success_rate_stdev_factor)`。这个因子除以一千得到一个两位小数值。也就是说，如果期望的因子是1.9，运行时间值应该是1900，默认为1900。

上述每个配置值都可以通过运行时值覆盖。

返回

- 上一级
- 首页目录

HASH环负载均衡配置

环哈希负载均衡配置

当集群管理器中的 `lb_type` 设置为 `ring_hash` 时，将使用环哈希负载平衡策略。

```
{
  "minimum_ring_size": "...",
  "use_std_hash": "..."
}
```

- minimum_ring_size

  (optional, integer) 最小哈希环大小，即虚拟节点总数。尺寸更大可以提供更好的请求分布，因为群集中的每个主机将具有更多的虚拟节点。默认为1024。若主机总数大于最小值的情况下，每个主机将被分配一个虚拟节点。

- use_std_hash

  (optional, boolean) 默认为 `true` ，这意味着 `std::hash` 用于将主机散列到 `ketama` 环上。`std::hash` 可能因平台而异。为此，Envoy默认最终会使用xxHash。该字段用于迁移目的，最终将被弃用。现在将其设置为 `false` 以使用 `xxHash` 。

返回

- 上一级
- 首页目录

异常检测

异常值检测

异常值检测架构概述。

```
{
  "event_log_path": "..."
}
```

- event_log_path

  (optional, string) 指定异常事件日志的路径。

返回

- 上一级
- 首页目录

集群发现服务

集群发现服务(CDS)

```
{
  "cluster": "{...}",
  "refresh_delay_ms": "..."
}
```

- clusters

  (required, object) 承载群集发现服务的上游群集的定义。群集必须实现并运行CDS HTTP API的REST服务。

- refresh_delay_ms

  (optional, integer) 每次从CDS API刷新的延迟（以毫秒为单位）。Envoy将在`0-refresh_delay_ms`之间，添加一个额外的随机抖动。因此，最长可能的刷新延迟是2*refresh_delay_ms。默认值是30000ms（30秒）。

REST API

```
GET /v1/clusters/(string: service_cluster)/(string: service_node)
```

集群发现服务返回`service_cluster`和`service_node`的所有群集定义。`service_cluster`对应于`--service-cluster` CLI选项。`service_node`对应于`--service-node` CLI选项。使用以下JSON格式响应:

```
{
  "clusters": []
}
```

- clusters

  (required, array) 将在集群管理器中动态添加/修改的集群列表。Envoy将协调此列表与当前加载的集群，并根据需要添加/修改/删除集群。请注意，在Envoy配置中静态定义的任何群集，都不能通过CDS API进行修改。

返回

- 上一级
- 首页目录

服务发现服务

服务发现服务(SDS)

服务发现服务[架构概述]。

```
{
  "cluster": "{...}",
  "refresh_delay_ms": "{...}"
}
```

- cluster

  (required, object) 承载服务发现服务的上游群集的标准定义。该群集必须实现和运行SDS HTTP API的REST服务。

- refresh_delay_ms

  (required, integer) 每次访问SDS群集的API延迟（以毫秒为单位）。Envoy将在`0~refresh_delay_ms`之间，添加一个额外的随机抖动。因此，最长可能的延迟是`2*refresh_delay_ms`。

REST API

Envoy希望服务发现服务提供一下API（请参阅Lyft的参考实现）：

```
GET /v1/registration/(string: service_name)
```

请求发现服务返回指定`service_name`的所有主机。`service_name`对应于群集参数`service_name`。使用以下JSON格式响应：

```
{
  "hosts": []
}
```

- hosts

  (required, array) 组成该服务的主机列表。

Host(JSON)

```
{
  "ip_address": "...",
```

```
    "port": "...",
    "tags": {
      "az": "...",
      "canary": "...",
      "load_balancing_weight": "..."
    }
  }
```

- ip_address

  (required, string) 上游主机的IP地址。

- port

  (required, integer) 上游主机的端口。

- az

  (optional, string) 可选的上游主机的区域。Envoy使用该区域记录相应的各种统计和负载平衡任务。

- canary

  (optional, boolean) 可选的上游主机金丝雀（灰度发布）状态。 Envoy将Canary状态用于相应的各种统计和负载平衡任务。

- load_balancing_weight

  (optional, integer) 可选的上游主机负载平衡权重，范围为1-100. Envoy在某些内置负载平衡器中会使用到负载平衡权重。

返回

- 上一级
- 首页目录

访问日志

访问日志

配置项

```
{
  "access_log": [
    {
      "path": "...",
      "format": "...",
      "filter": "{...}",
    },
  ]
}
```

- path

  (required, string) 写入访问日志的路径。

- format

  (optional, string) 访问日志格式。Envoy支持自定义访问日志格式以及默认格式。

- filter

  (optional, object) 用于明确是否需要写入的访问日志过滤器。

过滤器

Envoy支持以下访问日志过滤器：

- Status code
- Duration
- Not health check
- Traceable
- Runtime
- And
- Or

Status code

```
{
  "filter": {
    "type": "status_code",
    "op": "...",
    "value": "...",
```

```
    "runtime_key": "..."
  }
}
```

基于HTTP响应/状态代码的过滤器。

- op

  (required, string) 比较运算符。目前仅支持 `>=` 和 `=` 运算符。

- value

  (required, integer) 如果运行时值不可用，则使用默认值进行比较。

- runtime_key

  (optional, string) 运行时的key，用于获取比较值。如果定义，则使用此值。

Duration

```
{
  "filter": {
    "type": "duration",
    "op": "..",
    "value": "...",
    "runtime_key": "..."
  }
}
```

请求持续的总时间过滤器，以毫秒为单位。

- op

  (required, string) 比较运算符。目前仅支持 `>=` 和 `=` 运算符。

- value

  (required, integer) 如果运行时值不可用，则使用默认值进行比较。

- runtime_key

  (optional, string) 运行时的key，用于获取比较值。如果定义，则使用此值。

Not health check

```
{
  "filter": {
```

```
    "type": "not_healthcheck"
  }
}
```

筛选健康检查失败的请求。健康检查的请求由健康检查过滤器进行标记。

Traceable

```
{
  "filter": {
    "type": "traceable_request"
  }
}
```

筛选可跟踪的请求。请参阅跟踪概述，以获取有关请求如何使能跟踪的详细信息。

Runtime

```
{
  "filter": {
    "type": "runtime",
    "key" : "..."
  }
}
```

使用随机采样请求。在 `x-request-id` 头存在的情况下进行采样。如果存在 `x-request-id` ，则过滤器将根据运行时键值和从 `x-request-id` 提取的值在多个主机上持续采样。如果缺失，过滤器将根据运行时键值随机采样。

- key

  (required, string) 通过运行时key获取要采样的请求的百分比。此运行时值控制在 `0-100` 范围内，默认为0。

And

```
{
  "filter": {
    "type": "logical_and",
    "filters": []
  }
}
```

对过滤器中每个过滤器的结果执行逻辑"和"运算。过滤器将按顺序进行评估，如果其中一个返回false，则过滤器立即返回false。

Or

```
{
```

```
    "filter": {
      "type": "logical_or",
      "filters": []
    }
  }
```

对每个单独的过滤器的结果执行逻辑"或"操作。过滤器将按顺序进行评估，如果其中一个返回true，则过滤器会立即返回true。

返回

- 上一级
- 首页目录

管理接口

管理接口

管理接口操作文档。

```
{
  "access_log_path": "...",
  "profile_path": "...",
  "address": "..."
}
```

- access_log_path

  (required, string) 管理服务器的访问日志的路径。如果不需要访问日志，则指定"/dev/null"。

- profile_path

  (optional, string) 管理服务器的cpu分析器的文件路径。如果配置未指定，则默认路径为"/var/log/envoy/envoy.prof"。

- address

  (required, string) 管理服务器将监听的TCP地址，例如"tcp://127.0.0.1:1234"。请注意，"tcp://0.0.0.0:1234"将匹配任何带有端口1234的IPv4地址。

返回

- 上一级
- 首页目录

限速服务

速率限制服务

速率限制配置概述。

```
{
  "type": "grpc_service",
  "config": {
    "cluster_name": "..."
  }
}
```

- type

  (required, string) 指定要调用的速率限制服务的类型。目前唯一支持的选项是 `grpc_service` ，它指定了Lyft的全局速率限制服务和关联的IDL。

- config

  (required, object) 特定速率限制服务类型的配置。

- cluster_name

  (required, string) 指定所承载的速率限制服务的群集名称。当需要进行速率限制服务请求时，客户端将连接到该群集。

返回

- 上一级
- 首页目录

运行时配置

运行时配置

运行时配置概述。

```
{
  "symlink_root": "...",
  "subdirectory": "...",
  "override_subdirectory": "..."
}
```

- symlink_root

  (required, string) 当前的实现是假定文件系统目录是通过符号链接方式进行访问。在切换到新文件目录时，应该使用原子链接交换。此参数是指定链接符号的路径。Envoy将观察连接是否更改，并在发生更改时重新加载文件目录树。

- subdirectory

  (required, string) 指定在根目录中加载的子目录。如果有多个系统的运行时配置共享相同的链接交换机制，这很有用。Envoy配置元素可以包含在一个专用的子目录中。

- override_subdirectory

  (optional, string) 指定在根目录中加载的可选子目录。如果指定并且目录存在，则此目录中的配置值将覆盖在主子目录中找到的值。当Envoy跨多种不同类型的服务器部署时，这是非常有用的。有时为运行时配置提供每个服务集群目录是有用的。请参阅下面，明确如何使用覆盖目录。

返回

- 上一级
- 首页目录

跟踪

跟踪

跟踪配置指定了Envoy使用的HTTP跟踪器的全局设置。在服务的顶层配置上定义。未来，Envoy可能会支持其他跟踪器，但现在HTTP跟踪器是唯一支持的跟踪器。

```
{
  "http": {
    "driver": "{...}"
  }
}
```

- http

  (optional, object) 提供HTTP跟踪器的配置。

- driver

  (optional, object) 提供处理跟踪和创建span的驱动程序。

目前支持LightStep和Zipkin驱动程序。

LightStep跟踪驱动

```
{
  "type": "lightstep",
  "config": {
    "access_token_file": "...",
    "collector_cluster": "..."
  }
}
```

- access_token_file

  (required, string) 包含访问到LightStep API的令牌文件。

- collector_cluster

  (required, string) 承载LightStep集群的集群管理器。

Zipkin跟踪驱动

```
{
  "type": "zipkin",
  "config": {
```

```
    "collector_cluster": "...",
    "collector_endpoint": "..."
  }
}
```

- collector_cluster

  (required, string) 承载Zipkin集群的群集管理器。请注意，Zipkin集群必须在集群管理器定义相应的集群配置。

- collector_endpoint

  (optional, string) 将span发送的Zipkin服务的API端口。当安装标准的Zipkin时，API端口通常是/api/v1/span，同时也是默认值。

返回

- 上一级
- 首页目录

V2 API参考

V2 API参考

- 启动引导
  - Bootstrap
  - Bootstrap.StaticResources
  - Bootstrap.DynamicResources
  - Bootstrap.DynamicResources.DeprecatedV1
  - LightstepConfig
  - ZipkinConfig
  - Tracing
  - Tracing.Http
  - Admin
  - ClusterManager
  - ClusterManager.OutlierDetection
  - StatsdSink
  - StatsSink
  - TagSpecifier
  - StatsConfig
  - Watchdog
  - Runtime
  - RateLimitServiceConfig

- 监听&监听发现
  - Listener
  - Listener.DrainType(Enum)
  - Filter
  - FilterChainMatch
  - FilterChain

- 集群&集群发现
  - Cluster
  - Cluster.EdsClusterConfig
  - Cluster.OutlierDetection
  - Cluster.LbSubsetConfig
  - Cluster.LbSubsetConfig.LbSubsetSelector
  - Cluster.LbSubsetConfig.LbSubsetFallbackPolicy(Enum)
  - Cluster.RingHashLbConfig

    - Cluster.RingHashLbConfig.DeprecatedV1
    - Cluster.DiscoveryType(Enum)
    - Cluster.LbPolicy(Enum)
    - Cluster.DnsLookupFamily(Enum)
    - UpstreamBindConfig
    - CircuitBreakers
    - CircuitBreakers.Thresholds

- 服务发现
    - LbEndpoint
    - LocalityLbEndpoints
    - ClusterLoadAssignment
    - ClusterLoadAssignment.Policy

- 健康检查
    - HealthCheck
    - HealthCheck.Payload
    - HealthCheck.HttpHealthCheck
    - HealthCheck.TcpHealthCheck
    - HealthCheck.RedisHealthCheck

- HTTP路由管理&发现
    - RouteConfiguration
    - VirtualHost
    - VirtualHost.TlsRequirementType(Enum)
    - Route
    - WeightedCluster
    - WeightedCluster.ClusterWeight
    - RouteMatch
    - CorsPolicy
    - RouteAction
    - RouteAction.RetryPolicy
    - RouteAction.RequestMirrorPolicy
    - RouteAction.HashPolicy
    - RouteAction.HashPolicy.Header
    - RouteAction.HashPolicy.Cookie
    - RouteAction.HashPolicy.ConnectionProperties
    - RouteAction.ClusterNotFoundResponseCode(Enum)
    - RedirectAction
    - RedirectAction.RedirectResponseCode(Enum)

- Decorator
- VirtualCluster
- RateLimit
- RateLimit.Action
- RateLimit.Action.SourceCluster
- RateLimit.Action.DestinationCluster
- RateLimit.Action.RequestHeaders
- RateLimit.Action.RemoteAddress
- RateLimit.Action.GenericKey
- RateLimit.Action.HeaderValueMatch
- HeaderMatcher

- TLS配置
  - DataSource
  - TlsParameters
  - TlsParameters.TlsProtocol(Enum)
  - TlsCertificate
  - TlsSessionTicketKeys
  - CertificateValidationContext
  - CommonTlsContext
  - UpstreamTlsContext
  - DownstreamTlsContext

- 通用类型
  - Locality
  - Node
  - Endpoint
  - Metadata
  - RuntimeUInt32
  - HeaderValue
  - HeaderValueOption
  - ApiConfigSource
  - ApiConfigSource.ApiType(Enum)
  - AggregatedConfigSource
  - ConfigSource
  - TransportSocket
  - RoutingPriority(Enum)
  - RequestMethod(Enum)

- 网络地址

    - Pipe
    - SocketAddress
    - SocketAddress.Protocol(Enum)
    - BindConfig
    - Address
    - CidrRange

- HTTP协议选项
    - Http1ProtocolOptions
    - Http2ProtocolOptions

- 通用发现接口
    - DiscoveryRequest
    - DiscoveryResponse

- 通用速率限制
    - RateLimitDescriptor
    - RateLimitDescriptor.Entry

- 过滤器
    - 网络过滤器
    - HTTP过滤器
    - 常见访问日志类型
    - 常见故障注入类型

返回

- 首页目录

启动引导

启动引导

该协议通过命令行 `-c` 参数提供，并作为Envoy v2根配置。有关更多详细信息，请参阅v2配置概述。

- Bootstrap
- Bootstrap.StaticResources
- Bootstrap.DynamicResources
- Bootstrap.DynamicResources.DeprecatedV1
- LightstepConfig
- ZipkinConfig
- Tracing
- Tracing.Http
- Admin
- ClusterManager
- ClusterManager.OutlierDetection
- StatsdSink
- StatsSink
- TagSpecifier
- StatsConfig
- Watchdog
- Runtime
- RateLimitServiceConfig

Bootstrap

Bootstrap proto

启动引导配置概述

```
{
  "node": "{...}",
  "static_resources": "{...}",
  "dynamic_resources": "{...}",
  "cluster_manager": "{...}",
  "flags_path": "...",
  "stats_sinks": [],
  "stats_config": "{...}",
  "stats_flush_interval": "{...}",
  "watchdog": "{...}",
  "tracing": "{...}",
  "rate_limit_service": "{...}",
  "runtime": "{...}",
  "admin": "{...}"
}
```

- node

  (Node) 节点标识，会在管理服务器中呈现，用于标识目的（例如，头域中生成相应的字段）

- static_resources

  (Bootstrap.StaticResources) 指定静态资源配置

- dynamic_resources

  (Bootstrap.DynamicResources) 动态发现服务源配置

- cluster_manager

  (ClusterManager) 该服务所有的上游群集的群集管理配置

- flags_path

  (string) 可选，用于启动文件标志的路径。

- stats_sinks

  (StatsSink) 可选，统计汇总设置

- stats_config

  (StatsConfig) 配置内部处理统计信息

- stats_flush_interval

  (Duration) 可选，刷新到统计信息服务的周期时间。出于性能方面的考虑，Envoy锁定计数器，并且只是周期性地刷新计数器和计量器。如果未指定，则默认值为5000毫秒（5秒）。

- watchdog

  (Watchdog) 可选，看门狗配置

- tracing

  (Tracing) 配置外置的跟踪服务程序。如果没有指定，则不会执行跟踪。

- rate_limit_service

(RateLimitServiceConfig) 配置外置的速率限制服务。如果没有指定，任何调用速率限制服务将立即返回成功。

- runtime

  (Runtime) 配置运行时配置分发服务程序。如果未指定，这将导致使用所有默认值。

- admin

  (Admin, 必选) 配置本地管理的HTTP服务。

Bootstrap.StaticResources

Bootstrap.StaticResources proto

```
{
  "listeners": [],
  "clusters": []
}
```

- listeners

  (Listener) 静态监听器。无论LDS如何配置，这些监听器都将可用。

- clusters

  (Cluster) 如果cds_config指定了基于网络的配置源，则需要提供一些初始集群定义，以便Envoy知道如何与管理服务交互。这些集群定义可能不使用EDS（即它们应该是基于静态IP或DNS）。

Bootstrap.DynamicResources

Bootstrap.DynamicResources proto

```
{
  "lds_config": "{...}",
  "cds_config": "{...}",
  "ads_config": "{...}",
  "deprecated_v1": "{...}"
}
```

- lds_config

  (ConfigSource) 所有的监听器配置定义都由一个LDS配置源提供配置。

- cds_config

(ConfigSource) 所有后启动引导群集定义由单个CDS配置源提供配置。

- ads_config

  (ApiConfigSource) 可选，指定一个ADS源。这必须将 `api_type` 配置为 `GRPC` 。在配置源中仅ADS基于流传输通道。

- deprecated_v1

  (Bootstrap.DynamicResources.DeprecatedV1)

Bootstrap.DynamicResources.DeprecatedV1

Bootstrap.DynamicResources.DeprecatedV1 proto

```
{
  "sds_config": "{...}"
}
```

- sds_config

  (ConfigSource) 这是使用v1 REST API提供CDS/EDS服务发现的全局SDS配置。

LightstepConfig

LightstepConfig proto

Configuration for the LightStep tracer.

```
{
  "collector_cluster": "...",
  "access_token_file": "..."
}
```

- collector_cluster

  (string, 必选) 配置LightStep服务的群集管理

- access_token_file

  (string, 必选) 包含访问LightStep API的 `token` 文件。

ZipkinConfig

ZipkinConfig proto

```
{
  "collector_cluster": "...",
  "collector_endpoint": "..."
}
```

- collector_cluster

  (string, 必选)
  配置Zipkin服务的群集管理器。请注意，Zipkin群集必须在Bootstrap静态群集资源中定义。

- collector_endpoint

  (string, 必选) 将 `span` 发送的Zipkin服务的API端口。当安装标准的Zipkin服务时，API端口通常是 `/api/v1/spans` ，这也是默认配置值。

Tracing

Tracing proto

跟踪配置，指定Envoy使用的HTTP跟踪全局设置。该配置由Bootstrap tracing字段定义。未来，Envoy可能会支持其他跟踪器，但现在仅支持HTTP跟踪器。

```
{
  "http": "{...}"
}
```

- http

  (Tracing.Http) 提供HTTP跟踪器的配置

Tracing.Http

Tracing.Http proto

```
{
  "name": "...",
  "config": "{...}"
}
```

- name

  (string, 必选) 将要实例化的HTTP跟踪驱动程序的名称。该名称必须与支持的HTTP跟踪驱动程序相匹配。目前支持 `envoy.lightstep` 和 `envoy.zipkin` 内置的跟踪驱动程序。

- config

(Struct) 跟踪驱动程序的具体配置，这取决于实例化的驱动程序。 有关示例，请参阅LightstepConfig和ZipkinConfig跟踪驱动程序配置。

Admin

Admin proto

管理接口操作文档

```
{
  "access_log_path": "...",
  "profile_path": "...",
  "address": "{...}"
}
```

- access_log_path

  (string, 必选) 配置管理服务的访问日志路径。如果不需要访问日志，则指定"/dev/null"。

- profile_path

  (string) 管理服务的cpu分析器输出路径。如果未指定配置文件路径，则默认为"/var/log/envoy/envoy.prof"。

- address

  (Address, 必选) 管理服务要监听的TCP地址

ClusterManager

ClusterManager proto

Cluster manager architecture overview.

```
{
  "local_cluster_name": "...",
  "outlier_detection": "{...}",
  "upstream_bind_config": "{...}",
  "load_stats_config": "{...}"
}
```

- local_cluster_name

  (string) 本地集群的名称（即拥有运行此配置的Envoy集群）。为了启用区域感知路由，必须设置此选项。如果定义了`local_cluster_name`，则必须在Bootstrap.StaticResources资源中定义集

群。这与 `--service-cluster` 选项无关，该选项不影响区域感知路由。

- outlier_detection

  (ClusterManager.OutlierDetection) 可选，全局的异常值检测配置。

- upstream_bind_config

  (BindConfig) 可选，用于绑定新建立的上游连接。这可以通过 `cds_config` 中的 `upstream_bind_config` 以群集为基础进行覆盖。

- load_stats_config

  (ApiConfigSource) 管理服务端口负载统计信息通过 `StreamLoadStats` 流传输。这必须将 `api_type` 配置为 `GRPC` 。

ClusterManager.OutlierDetection

ClusterManager.OutlierDetection proto

```
{
  "event_log_path": "..."
}
```

- event_log_path

  (string) 指定异常事件日志的路径

StatsdSink

StatsdSink proto

统计配置，用于内置 `envoy.statsd` 接收器的proto类型定义。

```
{
  "address": "{...}",
  "tcp_cluster_name": "..."
}
```

- address

  (Address) 可用的 `statsd` 兼容收集器的UDP地址。如果指定，统计数据将被刷新到这个地址。

- tcp_cluster_name

(string) 提供基于TCP 的statsd兼容收集器群集名称。如果指定，Envoy将连接到此群集以刷新统计信息。

StatsSink

StatsSink proto

可插拔的统计接收器配置

```
{
  "name": "...",
  "config": "{...}"
}
```

- name

  (string) 统计信息的名称可以进行实例化。该名称必须与受支持的统计信息库相匹配。`envoy.statsd` 是一个适合发送到 `statsd` 的内置收集器。

- config

  (Struct) 特定的统计收集器配置，这取决于被实例化的收集器。以StatsdSink为例。

TagSpecifier

TagSpecifier proto

从指定标签抽取和剥离的标签名称，并将其作为所有统计信息的指定标签使用。只有当名称的任何部分与有一个或多个捕获组正则表达式匹配时才会发生这种情况。

```
{
  "tag_name": "...",
  "regex": "..."
}
```

- tag_name

  (string) 将标识符附加到标签值以标识接收器中的标签。 Envoy有一组默认名称和正则表达式来提取现有统计信息的动态部分，这些信息可以在Envoy存储库中的 `well_known_names.h` 中找到。如果在配置中提供了一个 `tag_name` ，那么Envoy会尝试在默认设置中找到该名称，并使用附带的正则表达式。

  注意：如果任何默认标签被指定两次，配置将被视为无效。

- regex

(string) 第一个捕获组识别要删除的名称部分。第二个捕获组（通常嵌套在第一个捕获组中）将指定统计信息的标记值。如果没有提供第二个捕获组，第一个也将用于设置标签的值。所有其他捕获组将被忽略。

例如，一个统计名称为`cluster.foo_cluster.upstream_rq_timeout`和

```
{
"tag_name": "envoy.cluster_name",
"regex": "^cluster\.((.+?)\.)"
}
```

请注意，正则表达式将删除`foo_cluster.`使标签名称为`cluster.upstream_rq_timeout`，并且将`envoy.cluster_name`的标签增加`foo_cluster`（注意：由于第二个捕获组，将不会有`.`字符）。

有两个正则表达式和统计信息名称的示例`http.connection_manager_1.user_agent.ios.downstream_cx_total`：

```
[
{
  "tag_name": "envoy.http_user_agent",
  "regex": "^http(?=\.).*?\.user_agent\.((.+?)\.)\w+?$"
},
{
  "tag_name": "envoy.http_conn_manager_prefix",
  "regex": "^http\.((.*?)\.)"
}
]
```

第一个正则表达式将删除`ios`，留下标签提取名称`http.connection_manager_1.user_agent.downstream_cx_total`。 标签`envoy.http_user_agent`将添加`ios标签值。

第二个正则表达式将删除`connection_manager_1`。从第一个正则表达式`http.connection_manager_1.user_agent.downstream_cx_total`生成的标签提取名称中，留下`http.user_agent.downstream_cx_total`作为标签提取名称。在标签`envoy.http_conn_manager_prefix`将添加`connection_manager_1`标签值。

StatsConfig

StatsConfig proto

统计架构概述。

```
{
  "stats_tags": [],
  "use_all_default_tags": "{...}"
}
```

- stats_tags

  (TagSpecifier) 每个统计信息的名称都会通过这些标记符进行迭代处理。当标签匹配时，将从第一个捕获组名称中删除，后面的 `TagSpecifiers` 不能匹配相同的部分。

- use_all_default_tags

  (BoolValue) 使用Envoy中指定的所有默认正则表达式标记。这些可以与 `stats_tags` 中自定义标签结合使用。他们将在自定义标签之前先进行处理。

  注意：如果任何默认标签被指定两次，则配置将被视为无效。

  有关Envoy中默认标记的列表，请参阅well_known_names.h。

  如果没有提供，则该值默认为真。

Watchdog

Watchdog proto

Envoy进程的看门狗配置。配置后，将监视未响应的线程，并在达到配置的阈值后终止本进程。

```
{
  "miss_timeout": "{...}",
  "megamiss_timeout": "{...}",
  "kill_timeout": "{...}",
  "multikill_timeout": "{...}"
}
```

- miss_timeout

  (Duration) Envoy统计 `server.watchdog_miss` 统计信息中的未响应线程的持续时间。如果没有指定，默认是200ms。

- megamiss_timeout

  (Duration) Envoy在 `server.watchdog_mega_miss` 统计信息中计算无响应线程的持续时间。如果未指定，则默认值为1000ms。

- kill_timeout

(Duration) 如果被监控的线程在这段时间内没有响应，则假定编程错误并终止整个Envoy进程。设置为0可禁用kill行为。如果未指定，则默认值为0（禁用）。

- multikill_timeout

  (Duration) 如果至少有两个被监控的线程，在这个持续时间内没有响应，则假定为真正的死锁并杀死整个Envoy进程。设置为0可禁用此行为。如果未指定，则默认值为0（禁用）。

Runtime

Runtime proto

Runtime架构概述。

```
{
  "symlink_root": "...",
  "subdirectory": "...",
  "override_subdirectory": "..."
}
```

- symlink_root

  (string, 必选) 该实现假定文件系统通过符号链接进行访问。并在切换到新文件目录时，使用原子链接交换。此参数指定符号链接的路径。Envoy将观察位置是否更改，并在发生时重新加载文件目录树。

- subdirectory

  (string) 指定要在根目录中加载的子目录。如果多个系统共享相同的更新机制，这会很有用。Envoy配置文件可以包含在一个专用的子目录中。

- override_subdirectory

  (string) 指定在根目录中加载的可选子目录。如果指定的目录存在，则此目录中的配置值将覆盖在主子目录中对应的值。当Envoy跨多种不同的服务器类型部署时，这很有用。有时为运行时配置提供每个服务集群目录是有用的。请参阅下面的如何使用覆盖目录。

RateLimitServiceConfig

RateLimitServiceConfig proto

速率限制架构概述。

```
{
  "cluster_name": "..."
}
```

```
```

- cluster_name

  (string, 必选) 指定承载速率限制服务的群集管理器名称。当需要进行速率限制服务请求时，客户端将连接到该群集，获取速率限制配置服务。

返回

- 上一级
- 首页目录

监听&监听发现

监听&监听发现

Listener架构概述

- Listener
- Listener.DrainType (Enum)
- Filter
- FilterChainMatch
- FilterChain

Listener

Listener proto

```
{
  "name": "...",
  "address": "{...}",
  "filter_chains": [],
  "use_original_dst": "{...}",
  "per_connection_buffer_limit_bytes": "{...}",
  "drain_type": "..."
}
```

- name

  (string) 配置该监听器的唯一名称。如果没有提供名称，Envoy将为监听器分配一个内部UUID。如果要通过LDS动态更新或删除侦听器，则必须提供唯一的名称。默认情况下，监听器名称的最大长度限制为60个字符。可以通过 `--max-obj-name-len` 命令行参数设置为所需的最大长度限制。

- address

  (Address, REQUIRED) 监听器应该监听的地址。一般来说，地址必须是唯一的，尽管这是由操作系统的根据绑定规则管理的。例如，多个监听器可以监听Linux上的0端口，因为实际的端口将被OS分配。

- filter_chains

  (FilterChain, REQUIRED) 这个监听器需要使用的过滤器列表。 `FilterChain` 在连接上使用特殊的 `FilterChainMatch` 。

  注意：在当前版本中，仅支持多个过滤器链，因此可以配置SNI。有关如何配置SNI以获取更多信息，请参阅FAQ条目。当配置多个过滤器链时，每个过滤器链必须具有相同的一组过滤器。如果过滤器不

同，配置将无法加载。将来，这种限制将被放宽，使得根据哪个过滤器链匹配（基于SNI或其他参数）可以使用不同的过滤器。

- use_original_dst

  (BoolValue) 在连接重定向如果使用 `iptables` ，则代理接收连接的端口可能与原始目标地址不同。当此标志设置为 `true` 时，监听器将重定向的连接切换到与原始目标地址关联的监听程序。如果没有与原始目标地址关联的监听器，则连接由该监听器的接收处理。默认为 `false` 。

- per_connection_buffer_limit_bytes

  (UInt32Value) 监听器的新连接读取和写入缓冲区大小的软限制。如果未指定，则使用默认值定义（1MB）。

- drain_type

  (Listener.DrainType) 监听器的级别，执行的逐出类型。

Listener.DrainType (Enum)

Listener.DrainType proto

- DEFAULT

  (DEFAULT) 在调用 `/healthcheck/fail` 管理端口（连同健康检查过滤器），监听器删除/修改以及热重启时，进行响应处理。

- MODIFY_ONLY

  针对监听器删除/修改和热启动而响应处理。此设置不包括健康检查失败。 如果Envoy仅当作入口和出口监听器，这个设置是可取的。

Filter

Filter proto

```
{
  "name": "...",
  "config": "{...}"
}
```

- name

  (string, REQUIRED) 要实例化的过滤器的名称。 该名称必须与支持的过滤器匹配。内置的过滤器

有：

- envoy.echo
- envoy.http_connection_manager
- envoy.mongo_proxy
- envoy.redis_proxy
- envoy.tcp_proxy

- config

  (Struct) 对应的过滤器配置，这取决于被实例化的过滤器。有关更多文档，请参阅支持的过滤器。

FilterChainMatch

FilterChainMatch proto

指定用于为监听器选择特定过滤器链的匹配条件。

```
{
  "sni_domains": []
}
```

- sni_domains

  (string) 如果非空，则考虑 `SNI` 域名。可能包含通配符前缀，例如 `*.example.com` 的。

  注意: 有关如何配置SNI以获取更多信息，请参阅FAQ条目。

FilterChain

FilterChain proto

过滤器链包含一组匹配条件，一个TLS上下文选项，一组过滤器和各种其他参数。

```
{
  "filter_chain_match": "{...}",
  "tls_context": "{...}",
  "filters": [],
  "use_proxy_proto": "{...}"
}
```

- filter_chain_match

  (FilterChainMatch) 将连接匹配到此过滤器链时使用的匹配规则。

- tls_context

(DownstreamTlsContext) 此过滤器链的TLS上下文配置。

- filters

  (Filter) 构成与监听器建立连接的过滤器链，包含各个网络过滤器的列表。当连接事件发生时，按顺序处理。注意：如果过滤器列表为空，则默认关闭连接。

- use_proxy_proto

  (BoolValue) 监听器是否应该在新连接上使用PROXY协议V1头。如果启用此选项，则监听器将假定该连接的远程地址是在标题中指定的地址。包括AWS ELB的一些负载平衡器支持此选项。如果该选项不存在或设置为`false`，Envoy将使用物理对等地址作为远程连接地址。

返回

- 上一级
- 首页目录

集群&集群发现

集群&集群发现

Cluster

Cluster proto

```
{
  "name": "...",
  "type": "...",
  "eds_cluster_config": "{...}",
  "connect_timeout": "{...}",
  "per_connection_buffer_limit_bytes": "{...}",
  "lb_policy": "...",
  "hosts": [],
  "health_checks": [],
  "max_requests_per_connection": "{...}",
  "circuit_breakers": "{...}",
  "tls_context": "{...}",
  "http_protocol_options": "{...}",
  "http2_protocol_options": "{...}",
  "dns_refresh_rate": "{...}",
  "dns_lookup_family": "...",
  "dns_resolvers": [],
  "outlier_detection": "{...}",
  "cleanup_interval": "{...}",
  "upstream_bind_config": "{...}",
  "lb_subset_config": "{...}",
  "ring_hash_lb_config": "{...}",
  "transport_socket": "{...}"
}
```

- name

  (string, REQUIRED) 提供在所有群集中必须唯一的群集名称。在发布统计信息时，会使用集群名称。在发布任何统计信息时，集群名称将被转换为 `_` 。默认情况下，群集名称的最大长度限制为60个字符。可通过 `--max-obj-name-len` 命令行参数，提高此上限。

- type

  (Cluster.DiscoveryType)) 用于解析群集的服务发现类型。

- eds_cluster_config

  (Cluster.EdsClusterConfig) 用于群集的EDS更新配置。

- connect_timeout

  (Duration) 连接到该群集中主机的超时时长。

- per_connection_buffer_limit_bytes

  (UInt32Value) 连接集群的读写缓冲区大小。如果未指定，则使用默认值（1MB）。

- lb_policy

  (Cluster.LbPolicy)) 在集群中选择主机时使用的负载平衡器类型。

- hosts

  (Address) 如果服务发现类型是 `STATIC` ， `STRICT_DNS` 或 `LOGICAL_DNS` ，则需要配置。

- health_checks

  (HealthCheck) 群集可选的健康检查配置。如果未指定配置，则不会执行健康检查，并且假定所有群集成员都将始终处于健康状态。

- max_requests_per_connection

  (UInt32Value) 可选，单个上游连接的最大请求数。HTTP/1.1和HTTP/2连接池都遵守此参数。如果没有指定，则没有限制。若此参数设置为1将有效地禁用保活状态的连接。

- circuit_breakers

(CircuitBreakers) 可选，集群熔断配置。

- tls_context

  (UpstreamTlsContext) 连接到上游群集的TLS配置。如果没有指定TLS配置，则新连接不会使用TLS。

- http_protocol_options

  (Http1ProtocolOptions) 处理HTTP1请求时的其他选项。

  只能设置 `http_protocol_options` 和 `http2_protocol_options` 其中一个配置。

- http2_protocol_options

  (Http2ProtocolOptions) 即使需要默认的HTTP2协议选项，也必须设置此字段，以便Envoy将在创建新的HTTP连接池时，假定上游支持HTTP/2。目前，Envoy仅支持上游连接的先验证。即使TLS与ALPN一起使用，也必须指定 `http2_protocol_options` 。另外，这允许HTTP/2通过纯文本连接。

  只能设置 `http_protocol_options` 和 `http2_protocol_options` 其中一个配置。

- dns_refresh_rate

  (Duration) 指定DNS刷新率，在群集类型为 `STRICT_DNS` 或 `LOGICAL_DNS` 时，则将此值用作群集的DNS刷新率。如果未指定此设置，则此值默认为5000。对于 `STRICT_DNS` 和 `LOGICAL_DNS` 以外的群集类型，此设置将被忽略。

- dns_lookup_family

  (Cluster.DnsLookupFamily)) DNS IP地址解析策略。 如果未指定此设置，则该值默认为 `V4_ONLY` 。

- dns_resolvers

  (Address) 如果指定了DNS解析程序，并且群集类型是 `STRICT_DNS` 或 `LOGICAL_DNS` ，则此值用于指定群集的dns解析程序。如果未指定此设置，则该值默认为使用 `/etc/resolv.conf` 配置的默认解析器。对于 `STRICT_DNS` 和 `LOGICAL_DNS` 以外的其他集群类型，此设置将被忽略。

- outlier_detection

  (Cluster.OutlierDetection) 如果指定，则会为此上游群集启用异常值检测。每个配置值都可以通过运

行时配置覆盖。

- cleanup_interval

  (Duration) 从集群类型 `ORIGINAL_DST` 中删除过期主机的时间间隔。如果主机在这段时间内没有被用作上游目的地，则认为它们是陈旧的。随着新的连接重定向到Envoy，新的主机将按需添加到原始目标集群，从而导致集群中的主机数量随着时间而增长。没有陈旧的主机（它们被主动用作目的地）被保存在群集中，从而允许与它们的连接保持打开状态，从而节省了打开新连接所花费的等待时间。如果未指定此设置，则该值默认为5000毫秒。对于 `ORIGINAL_DST` 以外的其他群集类型，此设置将被忽略。

- upstream_bind_config

  (BindConfig) 用于绑定新建立的上游连接的可选配置。这将覆盖 `bootstrap proto` 中指定的任何 `bind_config` 。如果地址和端口是空的，则不执行绑定。

- lb_subset_config

  (Cluster.LbSubsetConfig) 配置负载平衡子集。

- ring_hash_lb_config

  (Cluster.RingHashLbConfig) 可选，配置环哈希负载平衡策略。只能设置一个 `ring_hash_lb_config` 。

- transport_socket

  (TransportSocket) 请参阅 base.TransportSocket描述。

Cluster.EdsClusterConfig

Cluster.EdsClusterConfig proto

只有当集群发现类型是EDS时才有效。

```
{
  "eds_config": "{...}",
  "service_name": "..."
}
```

- eds_config

  (ConfigSource) 此群集的EDS更新源的配置。

- service_name

  (string) 可选，替代集群的名称，提供给EDS服务。这与集群名称没有同样的限制，即它可以是任意的长度。

Cluster.OutlierDetection

Cluster.OutlierDetection proto

有关异常值检测的更多信息，请参阅架构概述。

```
{
  "consecutive_5xx": "{...}",
  "interval": "{...}",
  "base_ejection_time": "{...}",
  "max_ejection_percent": "{...}",
  "enforcing_consecutive_5xx": "{...}",
  "enforcing_success_rate": "{...}",
  "success_rate_minimum_hosts": "{...}",
  "success_rate_request_volume": "{...}",
  "success_rate_stdev_factor": "{...}",
  "consecutive_gateway_failure": "{...}",
  "enforcing_consecutive_gateway_failure": "{...}"
}
```

- consecutive_5xx

  (UInt32Value) 发生连续5xx逐出主机之前，连续5xx响应的数量。默认为5。

- interval

  (Duration) 每次异常值分析扫描的时间间隔，这可能导致新抛出异常以及主机被重新添加到服务集群。默认为10000ms或10s。

- base_ejection_time

  (Duration) 主机弹出的基准时间。实际时间等于基本时间乘以主机被逐出的次数。默认为30000ms或30s。

- max_ejection_percent

  (UInt32Value) 由于异常检测而逐出的主机占上游群集的最大百分比。默认为10%。

- enforcing_consecutive_5xx

  (UInt32Value) 当通过连续5xx检测到异常状态时，主机实际被逐出的几率百分比。这个设置可以用来

禁止逐出或者缓慢地加速。默认为100。

- enforcing_success_rate

  (UInt32Value) 通过成功率统计检测到异常状态时，主机实际被逐出的几率百分比。这个设置可以用来禁止逐出或者缓慢地加速。默认为100。

- success_rate_minimum_hosts

  (UInt32Value) 必须具有足够的请求量来检测成功率异常值的群集中的主机数量。如果主机数量小于此设置，则不会为群集中的任何主机执行通过成功率统计信息的异常值检测。默认为5。

- success_rate_request_volume

  (UInt32Value) 在一个时间间隔内（如上述定义的时间间隔）必须收集的最小请求总数，以便将此主机包含在基于成功率的异常值检测中。如果低于此设置，则不会为该主机执行通过成功率统计的异常值检测。默认为100。

- success_rate_stdev_factor

  (UInt32Value) 这个因子被用来确定异常逐出成功率的阈值。逐出阈值是平均成功率与该因子与平均成功率的标准偏差的乘积之差：`mean - (stdev * success_rate_stdev_factor)`。这个因素除以一千得到一个两位小数值。也就是说，如果期望的因子是1.9，运行时间值应该是1900，默认为1900。

- consecutive_gateway_failure

  (UInt32Value) 逐出之前连续发生的连续网关故障数量，包括（502,503,504状态或连接错误，映射到其中一个状态代码）默认为5。

- enforcing_consecutive_gateway_failure

  (UInt32Value) 当通过连续的网关故障检测到异常状态时，主机实际被逐出的几率百分比。这个设置可以用来禁止逐出或者缓慢地加速。默认为0。

Cluster.LbSubsetConfig

Cluster.LbSubsetConfig proto

（可选）将此群集中的端口划分为由端口元数据定义的子集，并按路由和加权群集元数据进行选择。

```
{
  "fallback_policy": "...",
  "default_subset": "{...}",
```

```
  "subset_selectors": []
}
```

- fallback_policy

  (Cluster.LbSubsetConfig.LbSubsetFallbackPolicy) 选定路由的元数据没有响应的端口子集匹配时使用的行为。该值默认为NO_FALLBACK。

- default_subset

  (Struct) 如果 `fallback_policy` 为 `DEFAULT_SUBSET` ，则指定在回退期间使用的端点的默认子集。将 `default_subset` 中的每个字段与 `envoy.lb` 命名空间下的匹配 `LbEndpoint.Metadata进行比较。没有主机匹配是有效的，在这种情况下，行为与` NO_FALLBACK `的` fallback_policy`相同。

- subset_selectors

  (Cluster.LbSubsetConfig.LbSubsetSelector) 对于每个条目，遍历 `LbEndpoint.Metadata` 的 `envoy.lb` 命名空间，并为每个唯一的key和value组合创建一个子集。例如：

  ```
  { "subset_selectors": [
    { "keys": [ "version" ] },
    { "keys": [ "stage", "hardware_type" ] }
  ]}
  ```

  当来自所选路由和加权群集的元数据包含与子集的元数据相同的key和value时，匹配子集。相同的主机可能出现在多个子集中。

Cluster.LbSubsetConfig.LbSubsetSelector
Cluster.LbSubsetConfig.LbSubsetSelector proto

子集的规格

```
{
  "keys": []
}
```

- keys

  (string) 与加权的群集元数据匹配的key列表。

Cluster.LbSubsetConfig.LbSubsetFallbackPolicy (Enum)
Cluster.LbSubsetConfig.LbSubsetFallbackPolicy proto

如果选择 `NO_FALLBACK` ，则会报告等效于没有健康主机的结果。如果选择了 `ANY_ENDPOINT` ，则可能会返回任何群集端点（取决于策略，健康检查等）。如果选择 `DEFAULT_SUBSET` ，则在匹配来自 `default_subset` 字段的值的端口上执行负载平衡。

- NO_FALLBACK

  (DEFAULT)
- ANY_ENDPOINT
- DEFAULT_SUBSET

Cluster.RingHashLbConfig

Cluster.RingHashLbConfig proto

RingHash负载均衡策略的具体配置。

```
{
  "minimum_ring_size": "{...}",
  "deprecated_v1": "{...}"
}
```

- minimum_ring_size

  (UInt64Value) 最小哈希环大小，即总的虚拟节点。更大的尺寸将提供更好的请求分布，因为集群中的每个主机将具有更多的虚拟节点。默认为1024.在主机总数大于最小值的情况下，每个主机将被分配一个虚拟节点。

- deprecated_v1

  (Cluster.RingHashLbConfig.DeprecatedV1) 已弃用的v1配置。

Cluster.RingHashLbConfig.DeprecatedV1

Cluster.RingHashLbConfig.DeprecatedV1 proto

```
{
  "use_std_hash": "{...}"
}
```

- use_std_hash

  (BoolValue) 默认为 `true` ，这意味着 `std::hash` 用于将主机散列到 `ketama` 环上。 `std::hash` 可能因平台而异。为此，Envoy默认最终会使用xxHash。该字段用于迁移目的，最终将被弃用。现在将其设置为 `false` 以使用 `xxHash` 。

Cluster.DiscoveryType (Enum)
Cluster.DiscoveryType proto

有关每种类型的解释，请参阅服务发现类型。

- STATIC

  (DEFAULT) 静态发现类型

- STRICT_DNS

  严格的DNS发现类型

- LOGICAL_DNS

  逻辑DNS发现类型

- EDS

  服务发现类型

- ORIGINAL_DST

  原始目标发现类型

Cluster.LbPolicy (Enum)
Cluster.LbPolicy proto

有关每种类型的信息，请参阅负载平衡器体系架构概述部分。

- ROUND_ROBIN

  (DEFAULT) 轮循负载平衡策略

- LEAST_REQUEST

  最小请求负载平衡策略

- RING_HASH

  环形散列负载平衡策略

- RANDOM

随机负载平衡策略

- ORIGINAL_DST_LB

  原始目标负载平衡策略

Cluster.DnsLookupFamily (Enum)
Cluster.DnsLookupFamily proto

当选择V4_ONLY时，DNS解析器将仅执行IPv4系列中的地址查找。如果选择V6_ONLY，则DNS解析程序将仅执行IPv6系列中的地址查找。如果指定了AUTO，则DNS解析器将首先执行IPv6系列中的地址查找，然后回退到IPv4系列中的地址查找。对于 `STRICT_DNS` 和 `LOGICAL_DNS` 以外的集群类型，此设置将被忽略。

- AUTO

  (DEFAULT)
- V4_ONLY
- V6_ONLY

UpstreamBindConfig
UpstreamBindConfig proto

包含Envoy的可扩展地址结构，在与上游建立连接时绑定。

```
{
  "source_address": "{...}"
}
```

- source_address

  (Address) 建立上游连接时，Envoy应该绑定的地址。

CircuitBreakers
CircuitBreakers proto

可以为每个定义的优先级单独指定断路设置。

```
{
  "thresholds": []
}
```

- thresholds

  (CircuitBreakers.Thresholds) 如果使用相同的 `RoutingPriority` 定义多个阈值，则使用列表中的第一个阈值。如果给定的 `RoutingPriority` 没有定义 `Thresholds` ，则使用默认值。

CircuitBreakers.Thresholds

CircuitBreakers.Thresholds proto

为RoutingPriority定义断路阈值设置。

```
{
  "priority": "...",
  "max_connections": "{...}",
  "max_pending_requests": "{...}",
  "max_requests": "{...}",
  "max_retries": "{...}"
}
```

- priority

  (RoutingPriority) 设置指定断路器的 `RoutingPriority` 。

- max_connections

  (UInt32Value) Envoy将对上游群集进行的最大连接数。如果未指定，则默认值为1024。

- max_pending_requests

  (UInt32Value) Envoy将允许上游集群的最大待处理请求数。如果未指定，则默认值为1024。

- max_requests

  (UInt32Value) Envoy将对上游群集执行的最大并行请求数。如果未指定，则默认值为1024。

- max_retries

  (UInt32Value) Envoy允许上游集群执行的最大并行重试次数。如果未指定，则默认值为3。

返回

- 上一级
- 首页目录

服务发现

服务发现

- LbEndpoint
- LocalityLbEndpoints
- ClusterLoadAssignment
- ClusterLoadAssignment.Policy

LbEndpoint

LbEndpoint proto

EndPoint是指Envoy可以将流量路由到的端口。

```
{
  "endpoint": "{...}",
  "metadata": "{...}",
  "load_balancing_weight": "{...}"
}
```

- endpoint

  (Endpoint) 上游主机标识符

- metadata

  (Metadata) 负载均衡端口元数据是指可用于为请求选择群集中的端口。过滤器名称应该指定为 `envoy.lb` 。一个bool类型的键值对例子为 `canary` ，提供上游主机的可选 `canary` 状态。这可以在路由的 `ForwardAction` `metadata_match` 字段中匹配，以在集群负载平衡中考虑的端口子集。

- load_balancing_weight

  (UInt32Value) 上游主机的可选负载平衡权重，范围为1~128. 在某些内置Envoy负载平衡器中，会使用负载平衡权重。端口的负载平衡权重除以本地所有端口权重的总和，以获得端口的所占通信量百分比。然后这个百分比再由来自 `LocalityLbEndpoints` 的端点的本地负载平衡权重加权。如果没有指定，每个主机被假定在一个地方具有相同的权重。

  注意：当前128的限制是有些随意，但是考虑当前实现的性能问题而使用，若解决这个问题是可以取消这个限制。

LocalityLbEndpoints

LocalityLbEndpoints proto

一组端口归属一个局部负载均衡器。一个本地可以有多个局部负载均衡器，通常只有在不同的组之间需要负载平衡权重或不同的优先级时，才会使用多个组。

```
{
  "locality": "{...}",
  "lb_endpoints": [],
  "load_balancing_weight": "{...}",
  "priority": "..."
}
```

- locality

  (Locality) 标识上游主机运行的位置。

- lb_endpoints

  (LbEndpoint) 所属的端口组配置。

- load_balancing_weight

  (UInt32Value) 可选：配置每个端口组的优先级、`region`、`zone`、`sub_zone`权重，范围1~128。一个端口组的负载均衡权重除以相同优先级的所有权重之和，获得该端口组的承载业务有效百分比。

  必须为相同优先级的所有局部端口组指定权重，若不指定，则认为每个局部在群集中具有相同的权重。

  注意：当前128的限制是有些随意，但是考虑当前实现的性能问题而使用，若解决这个问题是可以取消这个限制。

- priority

  (uint32) 可选：此局部负载均衡端口组的优先级。如果未指定，则默认为最高优先级（0）。

  一般情况下，Envoy只会选择最高优先级（0）的端口。如果该优先级的所有端口不可用/不健康，Envoy将快速故障恢复至下一个最高优先级的端口组。

  优先级应该配置为从0（最高）到N（最低），中间没有间隔。

ClusterLoadAssignment

ClusterLoadAssignment proto

从RDS中获取的路由，映射到单个集群或者使用RDS集群权重实现跨集群的流量拆分。

使用EDS，每个集群都有独立的LB进行处理，将在集群内的产生局部LB，即在一个局部主机之间具有更细粒度的LB。对于给定的集群，主机的有效权重是其 `load_balancing_weight` 乘以其局部的LB权重 `load_balancing_weight` 。

```
{
  "cluster_name": "...",
  "endpoints": [],
  "policy": "{...}"
}
```

- cluster_name

  (string, REQUIRED) 集群的名称。如果在集群EdsClusterConfig中指定，则是 `service_name` 值。

- endpoints

  (LocalityLbEndpoints) 对应的局部端口组列表。

- policy

  (ClusterLoadAssignment.Policy) 负载均衡策略设置。

ClusterLoadAssignment.Policy
ClusterLoadAssignment.Policy proto

负载平衡策略设置。

```
{
  "drop_overload": "..."
}
```

- drop_overload

  (double) 应该丢弃的流量（0-100）的百分比。如果上游主机无法从down机中恢复，或者无法进行自动调整，则需要对上游主机进行保护，因此需要修整整个传入流量以保护它们。

  注意：v2 API与v1 API存在差异，在v1中被称为维护模式。

返回
- 上一级
- 首页目录

健康检查

健康检查

健康检查架构概述。如果为集群配置了健康检查，则会发出相应的统计信息。详见请参考统计相关文档。

- HealthCheck
- HealthCheck.Payload
- HealthCheck.HttpHealthCheck
- HealthCheck.TcpHealthCheck
- HealthCheck.RedisHealthCheck

HealthCheck

HealthCheck proto

```
{
  "timeout": "{...}",
  "interval": "{...}",
  "interval_jitter": "{...}",
  "unhealthy_threshold": "{...}",
  "healthy_threshold": "{...}",
  "reuse_connection": "{...}",
  "http_health_check": "{...}",
  "tcp_health_check": "{...}",
  "redis_health_check": "{...}"
}
```

- timeout

  (Duration) 等待健康检查响应的时间。如果达到超时，则尝试健康检查将被视为失败。

- interval

  (Duration) 每次尝试健康检查之间的时间间隔。

- interval_jitter

  (Duration) 可选，抖动量（以毫秒为单位）。如果指定，在Envoy内部将抖动量叠加到时间间隔上。

- unhealthy_threshold

  (UInt32Value) 在主机被标记为不健康之前，需要进行不健康的健康检查次数。请注意，对于http运行健康检查，如果主机以503响应，此阈值将被忽略，并且主机立即被视为不健康。

- healthy_threshold

  (UInt32Value) 主机在标记为健康之前所需的健康检查次数。请注意，在启动过程中，只需要一次成功的健康检查即可将主机标记为健康状态。

- reuse_connection

  (BoolValue) 健康检查之间复用健康检查连接。默认是 `true` 。

- http_health_check

  (HealthCheck.HttpHealthCheck) HTTP健康检查。

- tcp_health_check

  (HealthCheck.TcpHealthCheck) TCP健康检查

- redis_health_check

  (HealthCheck.RedisHealthCheck) Redis健康检查

  注意：只能在 **`http_health_check`** ， **`tcp_health_check`** ， **`redis_health_check`** 选其中一个进行设置。

HealthCheck.Payload

HealthCheck.Payload proto

描述载荷中有效载荷的字节编码。

```
{
  "text": "..."
}
```

- text

  (string, REQUIRED) 十六进制编码载荷，例如， “000000FF” 。准确地说，必须设置一个文本。

HealthCheck.HttpHealthCheck

HealthCheck.HttpHealthCheck proto

```
{
  "path": "...",
  "service_name": "..."
}
```

- path

  (string, REQUIRED) 指定运行健康检查期间，所请求的HTTP路径。例如 `/healthcheck`。

- service_name

  (string) 可选，服务名称参数，用于验证运行状况检查的群集的身份。请参阅架构概述以获取更多信息。

HealthCheck.TcpHealthCheck

HealthCheck.TcpHealthCheck proto

```
{
  "send": "{...}",
  "receive": []
}
```

- send

  (HealthCheck.Payload) 若有效载荷为空，意味着仅做连接的健康检查。

- receive

  (HealthCheck.Payload) 当检查响应时，执行模糊匹配，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。

HealthCheck.RedisHealthCheck

HealthCheck.RedisHealthCheck proto

```
{}
```

返回

- 上一级
- 首页目录

HTTP路由管理&发现

HTTP路由管理和路由发现（RDS）

- RouteConfiguration
- VirtualHost
- VirtualHost.TlsRequirementType (Enum)
- Route
- WeightedCluster
- WeightedCluster.ClusterWeight
- RouteMatch
- CorsPolicy
- RouteAction
- RouteAction.RetryPolicy
- RouteAction.RequestMirrorPolicy
- RouteAction.HashPolicy
- RouteAction.HashPolicy.Header
- RouteAction.HashPolicy.Cookie
- RouteAction.HashPolicy.ConnectionProperties
- RouteAction.ClusterNotFoundResponseCode (Enum)
- RedirectAction
- RedirectAction.RedirectResponseCode (Enum)
- Decorator
- VirtualCluster
- RateLimit
- RateLimit.Action
- RateLimit.Action.SourceCluster
- RateLimit.Action.DestinationCluster
- RateLimit.Action.RequestHeaders
- RateLimit.Action.RemoteAddress
- RateLimit.Action.GenericKey
- RateLimit.Action.HeaderValueMatch
- HeaderMatcher

RouteConfiguration

RouteConfiguration proto

- 路由架构概述
- HTTP路由过滤器

```
{
  "name": "...",
  "virtual_hosts": [],
  "internal_only_headers": [],
  "response_headers_to_add": [],
  "response_headers_to_remove": [],
  "request_headers_to_add": [],
  "validate_clusters": "{...}"
}
```

- name

  (string) 路由配置的名称。例如，它可能会匹配filter.network.Rds中的`route_config_name`。

- virtual_hosts

  (VirtualHost) 一组虚拟主机组成的路由表。

- internal_only_headers

  (string) （可选）指定做为内部使用的HTTP头部字段列表。如果在外部请求中找到它们，则会在过滤器调用之前清除它们。有关更多信息，请参见x-envoy-internal。

- response_headers_to_add

  (HeaderValueOption) 指定在连接管理器编码时，为每个响应的增加HTTP头部字段列表。在这个级别指定的头部将位于内部的VirtualHost或RouteAction的头部之后。

- response_headers_to_remove

  (string) 指定在连接管理器编码每个响应中需要删除的HTTP头部字段列表。

- request_headers_to_add

  (HeaderValueOption) 指定在HTTP连接管理器路由的每个请求时，需要添加的HTTP头部字段列表。在这个级别指定的头部将位于内部的VirtualHost或RouteAction的头部之后。有关更多信息，请参阅自定义请求的头部字段文档。

- validate_clusters

  (BoolValue) 可选的bool类型，指定路由表引用的集群是否由集群管理器验证。如果设置为true，并且路由引用了不存在的集群，则路由表将不会加载。如果设置为false，并且路由引用不存在的集群，则路由表将加载，如果在运行时选择路由，则路由器过滤器将返回404。如果路由表是通过route_config选项静态定义的，则此配置默认为true。如果路由表是通过rds选项动态加载的，则此设

置默认为false。用户可以在某些情况下覆盖此默认行为（例如，当使用带有静态路由表的CDS时）。

VirtualHost

VirtualHost proto

在路由配置的顶层选项有个虚拟主机。每个虚拟主机都有一个逻辑名称以及一组根据传入请求的主机头路由到它的域。这允许单个监听端口多个顶级域服务。一旦基于域选择了虚拟主机，就会根据顺序处理那些路由匹配到哪个上游集群，并且是否执行重定向。

```
{
  "name": "...",
  "domains": [],
  "routes": [],
  "require_tls": "...",
  "virtual_clusters": [],
  "rate_limits": [],
  "request_headers_to_add": [],
  "response_headers_to_add": [],
  "response_headers_to_remove": [],
  "cors": "{...}"
}
```

- name

  (string, REQUIRED) 虚拟主机的逻辑名称。用于某些统计信息，但与路由无关。

- domains

  (string, REQUIRED) 将与此虚拟主机相匹配的域（主机名）列表。支持通配符，例如：支持“.foo.com”或“-bar.foo.com”。

  注意：通配符将不匹配空字符串。例如“-bar.foo.com”将匹配“baz-bar.foo.com”，但不匹配“-bar.foo.com”。此外，还可以使用特殊的条目“”来匹配任何主机名。整个路由配置中只有一台虚拟主机可以匹配“*”。域名在所有虚拟主机中必须是唯一的，否则配置将无法加载。

- routes

  (Route) 将按顺序匹配传入请求的路由列表。第一个匹配的路由将被使用。

- require_tls

  (VirtualHost.TlsRequirementType) 指定虚拟主机所提供的TLS类型。如果未指定此选项，则虚拟主机不需要TLS。

- virtual_clusters

(VirtualCluster) 为此虚拟主机定义的虚拟集群列表。虚拟集群用于进行其他统计信息收集。

- rate_limits

  (RateLimit) 指定将应用于虚拟主机的一组限速配置。

- request_headers_to_add

  (HeaderValueOption) 指定添加到由此虚拟主机处理的每个请求的HTTP头部列表。在此级别指定的头部将应用于内部的RouteAction头部之后和封装RouteConfiguration的头部之前。有关更多信息，请参阅自定义请求头的文档。

- response_headers_to_add

  (HeaderValueOption) 指定添加到由此虚拟主机处理的每个响应的HTTP头部列表。在此级别指定的头部将应用于内部的RouteAction头部之后和封装RouteConfiguration的头部之前。

- response_headers_to_remove

  (string) 指定由此虚拟主机处理的每个响应中删除的HTTP头部列表。

- cors

  (CorsPolicy) 表示虚拟主机具有CORS策略。

VirtualHost.TlsRequirementType (Enum)

VirtualHost.TlsRequirementType proto

- NONE

  (DEFAULT) 虚拟主机没有TLS要求。

- EXTERNAL_ONLY

  来自外部请求必须使用TLS。如果请求是来自外部的，并且没有使用TLS，则将发送301重定向，告诉客户端使用HTTPS。

- ALL

  所有请求都必须使用TLS。 如果请求没有使用TLS，则会发送301重定向，通知客户端使用HTTPS。

Route

Route proto

路由既包括匹配那些请求，以及接下来需要执行的行为（例如，重定向，转发，重写等）。

注意：Envoy通过头匹配支持HTTP方法的路由。

```
{
  "match": "{...}",
  "route": "{...}",
  "redirect": "{...}",
  "metadata": "{...}",
  "decorator": "{...}"
}
```

- match

  (RouteMatch, REQUIRED) 路由匹配参数。

- route

  (RouteAction) 将请求路由到某个上游群集。

- redirect

  (RedirectAction) 返回一个重定向。

  注意：路由和重定向必须选择其中一个设置。

- metadata

  (Metadata) 元数据字段可用于提供有关路由的其他信息。 它可以用于配置，统计和日志记录。元数据应该在与之对应的滤器命名空间下。例如，如果此元数据用于路由器过滤器，则应将过滤器名称指定为 `envoy.router`。

- decorator

  (Decorator) 匹配路由的标识符。

WeightedCluster

WeightedCluster proto

与指定单个上游群集作为请求目标的群集字段相比，`weighted_clusters`选项允许指定多个上游群集以及制定要转发给每个群集的流量百分比的权重。路由器将根据权重选择上游集群。

```
{
  "clusters": [],
  "runtime_key_prefix": "..."
}
```

- clusters

  (WeightedCluster.ClusterWeight, REQUIRED) 指定与路由关联的一个或多个上游群集。

- runtime_key_prefix

  (string) 指定应该用于构造与每个集群关联的运行时key/value前缀。当指定 `runtime_key_prefix` 时，路由器将在 `runtime_key_prefix+"."+cluster[i].name` 下查找与每个上游集群相关的权重，其中 `cluster[i]` 表示集群组的某个集群。如果群集的运行时key不存在，则配置文件中指定的值将用作默认权重。有关键名称如何映射到底层实现，请参阅运行时文档。

WeightedCluster.ClusterWeight

WeightedCluster.ClusterWeight proto

```
{
  "name": "...",
  "weight": "{...}",
  "metadata_match": "{...}"
}
```

- name

  (string, REQUIRED) 上游群集的名称。群集必须存在于群集管理器配置中。

- weight

  (UInt32Value) 一个0~100之间的整数，当请求与路由匹配时，选择上游集群的权重。集群组中所有集群的权重总和加起来必须为100。

- metadata_match

  (Metadata) （可选）端口元数据匹配条件，将仅考虑上游集群中具有与在 `metadata_match` 中设置的元数据匹配的端口，过滤器名称应该指定为 `envoy.lb` 。

RouteMatch

RouteMatch proto

```
{
  "prefix": "...",
  "path": "...",
  "regex": "...",
  "case_sensitive": "{...}",
  "runtime": "{...}",
  "headers": []
}
```

- prefix

  (string) 如果指定，则路由采用匹配前缀规则，这意味着前缀必须匹配 `:path` 头部分。

- path

  (string) 如果指定，路由是一个精确的路径规则，意味着一旦查询字符串被移除，路径必须与 `:path` 头完全匹配。

- regex

  (string) 如果指定，则路由采用正则表达式规则，这意味着一旦查询字符串被移除，正则表达式必须匹配 `:path` 头。整个路径（不含查询字符串）必须匹配正则表达式。如果只有 `:path` 头的部分与正则表达式匹配，则实际规则为不匹配。这里定义了正则表达式语法。

  实例:

  - 正则表达式 `/b[io]t` 匹配路径 `/bit`
  - 正则表达式 `/b[io]t` 匹配路径 `/bot`
  - 正则表达式 `/b[io]t` 不匹配路径 `/bite`
  - 正则表达式 `/b[io]t` 不匹配路径 `/bit/bot`

  注意：只能选择 `prefix`，`path`，`regex` 其中一个设置。

- case_sensitive

  (BoolValue) 表示前缀/路径匹配是否不区分大小写。默认值是true。

- runtime

  (RuntimeUInt32) 指定当前路由匹配另外运行时的key。设置一个0-100之间的整数，每当路由匹配时，将会产生一个0-99之间的随机数。如果该随机数<=当前设置的值（首先检查）或者该key不存在，则默认行为，则匹配该路径（假定所有路径都与路由匹配）。可以在运行时逐步进行路由的变更，而无需完整的部署配置。请参阅流量转移文档以获取详细的说明。

- headers

  (HeaderMatcher) 指定路由匹配的一组头部字段。路由器将检查请求的头域中与该配置中的头部字段。如果路由配置中的所有头部字段都在请求头部匹配相同的值（或者路由配置的key存在，而value没有明确），则匹配将发生。

CorsPolicy

CorsPolicy proto

```
{
  "allow_origin": [],
  "allow_methods": "...",
  "allow_headers": "...",
  "expose_headers": "...",
  "max_age": "...",
  "allow_credentials": "{...}",
  "enabled": "{...}"
}
```

- allow_origin

  (string) 指定将被允许执行CORS请求的来源。

- allow_methods

  (string) 指定 `access-control-allow-methods` 头部的内容。

- allow_headers

  (string) 指定 `access-control-allow-headers` 头部的内容。

- expose_headers

  (string) 指定 `access-control-expose-headers` 头部的内容。

- max_age

  (string) 指定 `access-control-max-age` 头部的内容。

- allow_credentials

  (BoolValue) 指定资源是否允许凭据。

- enabled

(BoolValue) 指定是否启用CORS。默认为true。只在路由上有效。

RouteAction

RouteAction proto

```
{
  "cluster": "...",
  "cluster_header": "...",
  "weighted_clusters": "{...}",
  "cluster_not_found_response_code": "...",
  "metadata_match": "{...}",
  "prefix_rewrite": "...",
  "host_rewrite": "...",
  "auto_host_rewrite": "{...}",
  "timeout": "{...}",
  "retry_policy": "{...}",
  "request_mirror_policy": "{...}",
  "priority": "...",
  "request_headers_to_add": [],
  "response_headers_to_add": [],
  "response_headers_to_remove": [],
  "rate_limits": [],
  "include_vh_rate_limits": "{...}",
  "hash_policy": [],
  "use_websocket": "{...}",
  "cors": "{...}"
}
```

- cluster

  (string) 指示请求路由到的上游群集。

  注意：只能选择 `cluster`，`cluster_header`，`weighted_clusters` 其中一个设置。

- cluster_header

  (string) Envoy将通过从请求头中读取以 `cluster_header` 命名的HTTP头的值来明确要路由的群集。如果没有找到引用的群集，Envoy将返回一个404响应。

  注意：在内部，Envoy始终使用HTTP/2的 `:authority` 头来表示HTTP/1的 `Host` 头。因此，如果试图在使用 `Host` 匹配，则使用 `:authority` 替代。

  注意：只能选择 `cluster`，`cluster_header`，`weighted_clusters` 其中一个设置。

- weighted_clusters

  (WeightedCluster) 可以为该路由指定多个上游群集。根据每个群集的权重，将请求路由到其中一个上游群集。请参阅流量拆分以获取详细说明。

注意：只能选择 `cluster`，`cluster_header`，`weighted_clusters` 其中一个设置。

- cluster_not_found_response_code

  (RouteAction.ClusterNotFoundResponseCode) 未找到配置的群集时使用的HTTP状态码。默认响应码是503，服务不可用。

- metadata_match

  (Metadata)（可选）端口元数据匹配条件，将仅考虑上游集群中具有与在 `metadata_match` 中设置的元数据匹配的端口，过滤器名称应该指定为 `envoy.lb` 。

- prefix_rewrite

  (string) 表示在转发过程中，匹配的前缀（或路径）重写的值。此选项允许应用程序的URLs使用不同的路径（通过反向代理层公开的路径）。

- host_rewrite

  (string) 表示在转发过程中，主机头将重写的值。

  注意：只能选择 `host_rewrite`，`auto_host_rewrite` 其中一个设置。

- auto_host_rewrite

  (BoolValue) 表示在转发过程中，主机头将与群集管理器选择的上游主机的主机名交换。此选项仅适用于目标群集的类型为 `strict_dns` 或 `logical_dns` 。对于其他群集类型设置为true将无效。

  注意：只能选择 `host_rewrite`，`auto_host_rewrite` 其中一个设置。

- timeout

  (Duration) 指定路由的超时时间。如果未指定，则默认值为15秒。

  注意：该超时包括所有重试。 另请参阅x-envoy-upstream-rq-timeout-ms，x-envoy-upstream-rq-per-try-timeout-ms和重试概述。

- retry_policy

  (RouteAction.RetryPolicy) 表示该路由具有重试策略。

- request_mirror_policy

(RouteAction.RequestMirrorPolicy) 表示路由有请求镜像策略。

- priority

  (RoutingPriority) 可选，指定路由的优先级。

- request_headers_to_add

  (HeaderValueOption) 指定将被添加到匹配此路由请求的一组头。在这个级别指定的头部将添加在 `VirtualHost` 和 `RouteConfiguration` 的头部之前。有关更多信息，请参阅自定义请求头的文档。

- response_headers_to_add

  (HeaderValueOption) 指定一组头，这些头将被添加到匹配此路由的请求响应中。在这个级别指定的头部将添加在 `VirtualHost` 和 `RouteConfiguration` 的头部之前。

- response_headers_to_remove

  (string) 指定一个HTTP头的列表，将从匹配该路由的请求响应中删除。

- rate_limits

  (RateLimit) 指定可应用于该路由的一组速率限制配置。

- include_vh_rate_limits

  (BoolValue) 指定速率限制过滤器是否应包含虚拟主机速率限制。默认情况下，如果路由配置的速率限制，虚拟主机的 `rate_limits` 不适用于请求。

- hash_policy

  (RouteAction.HashPolicy) 指定用于环哈希负载平衡的哈希策略列表。每个散列策略都是单独评估的，并且组合结果用于路由请求。组合的方法是确定性的，使得相同的散列策略列表将产生相同的散列。由于散列策略检查请求的特定部分，因此可能无法产生散列（即散列头不存在）。如果（且仅当）所有配置的散列策略未能生成散列，则不会为该路由生成散列。在这种情况下的行为与没有指定哈希策略（即环哈希负载平衡器将随机选择后端）相同。

- use_websocket

  (BoolValue) 表示是否允许连接到该路由的HTTP/1.1客户端升级到WebSocket连接。默认值是false。

注意：如果设置为true，Envoy会希望匹配这个路由的第一个请求包含WebSocket升级头。如果头不存在，连接将被拒绝。如果设置为true，Envoy将在客户端和上游服务器之间设置纯TCP代理。因此，拒绝WebSocket升级请求的上游服务器也需要负责关闭相应的连接。在此之前，Envoy将继续将数据从客户端代理到上游服务器。若该路由允许websocket升级，则不支持重定向，超时和重试。

- cors

  (CorsPolicy) 表示该路由具有CORS策略。

RouteAction.RetryPolicy

RouteAction.RetryPolicy proto

HTTP重试架构概述。

```
{
  "retry_on": "...",
  "num_retries": "{...}",
  "per_try_timeout": "{...}"
}
```

- retry_on

  (string) 指定重试触发的条件。这些与 `x-envoy-retry-on` 和 `x-envoy-retry-grpc-on` 记录的具有相同效果。

- num_retries

  (UInt32Value) 指定允许的重试次数。此参数是可选的，默认值为1。这些与 `x-envoy-max-retries` 记录的具有相同效果。

- per_try_timeout

  (Duration) 指定每个重试尝试的超时时间，值为非零。该参数是可选的。为 `x-envoy-upstream-rq-per-try-timeout-ms` 记录的相同条件适用。

  注意：如果未指定，Envoy将使用全局路由超时请求。因此，当使用基于5xx的重试策略时，超时请求将不会被重试，因为总超时时长已经耗尽。

RouteAction.RequestMirrorPolicy

RouteAction.RequestMirrorPolicy proto

路由器能够将流量从一个集群映射到另一个集群。目前的实现是“fire and forget”，这意味着在返回来自主集群的响应之前，Envoy不会等待 `shadow` 集群作出响应。所有正常的统计数据都会被收集用于

`shadow`集群，使得该功能对测试有用。

在遮蔽期间，`host/authority`头部被改变，以便附加`-shadow`。这对于日志记录很有用。例如，`cluster1`变为`cluster1-shadow`。

```
{
  "cluster": "...",
  "runtime_key": "..."
}
```

- cluster

  (string, REQUIRED) 指定请求将被镜像到的群集。群集必须存在于群集管理器配置中。

- runtime_key

  (string) 如果未指定，则将对目标群集的所有请求进行镜像。如果指定，Envoy将查找运行时key以获取请求的镜像百分比。有效值为0到10000，支持0.01%的增幅。如果在配置中指定了运行时的key，但在运行过程中不存在，则默认为0，因此请求的0%将被镜像。

RouteAction.HashPolicy

RouteAction.HashPolicy proto

如果上游群集使用散列负载平衡器，则指定路由的散列策略。

```
{
  "header": "{...}",
  "cookie": "{...}",
  "connection_properties": "{...}"
}
```

- header

  (RouteAction.HashPolicy.Header) Header哈希策略。

- cookie

  (RouteAction.HashPolicy.Cookie) Cookie哈希策略。

- connection_properties

  (RouteAction.HashPolicy.ConnectionProperties) 连接属性的哈希策略。

  注意：只能选择`header`，`cookie`，`connection_properties`其中一个设置。

RouteAction.HashPolicy.Header

RouteAction.HashPolicy.Header proto

```
{
  "header_name": "..."
}
```

- header_name

  (string, REQUIRED) 将用于获取哈希key的请求头部名称。如果请求头不存在，则不会产生散列。

RouteAction.HashPolicy.Cookie

RouteAction.HashPolicy.Cookie proto

Envoy支持两种类型的Cookie关联：

1. 被动：Envoy需要一个存在于cookie头部的cookie，并对其值进行哈希处理。
2. 产生：根据请求发送到的端口，Envoy会根据客户端响应中的第一个请求生成并设置一个有效期（TTL）的cookie。然后，客户端将在随后的所有请求中进行携带这个散列以确保这些请求被发送到相同的端口。Cookie是通过散列源、目标端口和地址生成的，以便同一连接上的多个独立的HTTP2流将独立地接收相同的cookie，即使它们同时到达Enovy。

```
{
  "name": "...",
  "ttl": "{...}"
}
```

- name

  (string, REQUIRED) 将用于获取散列key的cookie的名称。 如果cookie不存在，并且下面的ttl没有设置，则不会产生散列。

- ttl

  (Duration) 若指定，在cookie不存在的情况下，会生成一个带有TTL的cookie。

RouteAction.HashPolicy.ConnectionProperties

RouteAction.HashPolicy.ConnectionProperties proto

```
{
  "source_ip": "..."
}
```

- source_ip

  (bool) 哈希源IP地址。

RouteAction.ClusterNotFoundResponseCode (Enum)
RouteAction.ClusterNotFoundResponseCode proto

- SERVICE_UNAVAILABLE

  (DEFAULT) HTTP状态码 - 503服务不可用。

- NOT_FOUND

  HTTP状态码 - 404未找到。

RedirectAction
RedirectAction proto

```
{
  "host_redirect": "...",
  "path_redirect": "...",
  "response_code": "..."
}
```

- host_redirect

  (string) 该主机的URL部分将与此值交换。

- path_redirect

  (string) 路径的URL部分将与此值交换。

- response_code

  (RedirectAction.RedirectResponseCode) 在重定向响应中使用的HTTP状态代码。默认响应码是MOVED_PERMANENTLY（301）。

RedirectAction.RedirectResponseCode (Enum)
RedirectAction.RedirectResponseCode proto

- MOVED_PERMANENTLY

  (DEFAULT) 永久转移的HTTP状态码 - 301。

- FOUND

  Found HTTP状态码 - 302。

- SEE_OTHER

  See Other HTTP状态码 - 303。

- TEMPORARY_REDIRECT

  临时重定向HTTP状态码 - 307。

- PERMANENT_REDIRECT

  永久重定向HTTP状态码 - 308。

Decorator

Decorator proto

```
{
  "operation": "..."
}
```

- operation

  (string, REQUIRED) 与此路由匹配的请求相关联的操作名称。如果已启用跟踪，则将使用此信息作为为此请求记录的span名称。

  注意：对于入口（入站）请求或出站（出站）响应，该值可能被 `x-envoy-decorator-operation` 头覆盖。

VirtualCluster

VirtualCluster proto

虚拟集群是一种针对某些重要端口，指定正则表达式匹配规则的方法，例如为匹配的请求显式生成统计信息。当做前缀/路径匹配时这很有用，Envoy并不总是知道应用程序认为是一个端口。因此，Envoy不可能统一发送每个端口的统计数据。然而，系统往往具有高度关联的端口，他们希望获得“完美”的统计数据。虚拟集群统计是完美的，因为它们在下游散发，因此包含网络级别的故障。

虚拟群集统计信息的文档。

注意：虚拟群集是一个有用的工具，但我们不建议为每个应用程序端口设置一个虚拟群集。 这既不容易维护，而且匹配和统计输出也是需要代价的。

```
{
  "pattern": "...",
  "name": "...",
  "method": "..."
}
```

- pattern

  (string, REQUIRED) 指定用于匹配请求的正则表达式模式。 请求的整个路径必须匹配正则表达式。所使用的正则表达式语法在这里定义。

  示例：

  - 正则表达式 `/rides/d+` 匹配 `/rides/0` 路径
  - 正则表达式 `/rides/d+` 匹配 `/rides/123` 路径
  - 正则表达式 `/rides/d+` 不匹配 `/rides/123/456` 路径

- name

  (string, REQUIRED) 指定虚拟群集的名称。发布统计信息时会使用虚拟群集名称和虚拟主机名称。统计信息由路由器过滤器发出，并记录在此处。

- method

  (RequestMethod)（可选）指定要匹配的HTTP方法。例如GET，PUT等

RateLimit

RateLimit proto

全局速率限制架构概述。

```
{
  "stage": "{...}",
  "disable_key": "...",
  "actions": []
}
```

- stage

  (UInt32Value) 指在过滤器中设置的阶段。速率限制配置仅适用于具有相同阶段编号的过滤器。默认的阶段编号是0。

  注意：过滤器支持0~10范围的阶段编号。

- disable_key

  (string) 在运行时禁用此速率限制的key配置。

- actions

  (RateLimit.Action, REQUIRED) 此速率限制应用相关的操作列表。顺序很重要，因为按顺序处理操作的，描述符是通过在该顺序中附加描述符条目来组成的。如果某个操作无法添加描述符条目，则不会为该配置生成描述符。请参阅相应的操作文档。

RateLimit.Action

RateLimit.Action proto

```
{
  "source_cluster": "{...}",
  "destination_cluster": "{...}",
  "request_headers": "{...}",
  "remote_address": "{...}",
  "generic_key": "{...}",
  "header_value_match": "{...}"
}
```

- source_cluster

  (RateLimit.Action.SourceCluster) 基于源群集的速率限制。

- destination_cluster

  (RateLimit.Action.DestinationCluster) 基于目标群集的速率限制。

- request_headers

  (RateLimit.Action.RequestHeaders) 基于请求头的速率限制。

- remote_address

  (RateLimit.Action.RemoteAddress) 基于远程地址的速率限制。

- generic_key

  (RateLimit.Action.GenericKey) 基于通用密钥的速率限制。

- header_value_match

(RateLimit.Action.HeaderValueMatch) 请求头的内容匹配的速率限制。

必须正确设置 `source_cluster` , `destination_cluster` , `request_headers` , `remote_address` , `generic_key` , `header_value_match` 其中的一个。

RateLimit.Action.SourceCluster

RateLimit.Action.SourceCluster proto

以下描述符条目被追加到描述符中：

```
("source_cluster", "<local service cluster>")
```

其中 `<local service cluster>` 源自 `--service-cluster` 选项。

```
{}
```

RateLimit.Action.DestinationCluster

RateLimit.Action.DestinationCluster proto

以下描述符条目被追加到描述符中：

```
("destination_cluster", "<routed target cluster>")
```

一旦请求与路由表规则匹配，则选择以下路由表配置设置之一路由集群：

- cluster：表示要到达的上游集群。
- weighted_clusters：从一组具有权重属性的集群组中随机选择一个集群。
- cluster_header：指示请求的头中选择所包含目标群集。

```
{}
```

RateLimit.Action.RequestHeaders

RateLimit.Action.RequestHeaders proto

当一个头包含一个与 `header_name` 匹配的关键字时，附加下面的描述符条目：

```
("<descriptor_key>", "<header_value_queried_from_header>")
```

```
{
  "header_name": "...",
```

```
  "descriptor_key": "..."
}
```

- header_name

  (string, REQUIRED) 从匹配请求的头部名称。头的值用于填充 `descriptor_key` 的描述符条目的值。

- descriptor_key

  (string, REQUIRED) 在描述符条目中使用的key。

RateLimit.Action.RemoteAddress

RateLimit.Action.RemoteAddress proto

以下描述符条目被追加到描述符中，并使用来自x-forwarded-for的可信地址填充：

```
("remote_address", "<trusted address from x-forwarded-for>")
```

```
{}
```

RateLimit.Action.GenericKey

RateLimit.Action.GenericKey proto

以下描述符条目被追加到描述符中：

```
("generic_key", "<descriptor_value>")
```

```
{
  "descriptor_value": "..."
}
```

- descriptor_value

  (string, REQUIRED) 描述符条目中使用的值。

RateLimit.Action.HeaderValueMatch

RateLimit.Action.HeaderValueMatch proto

以下描述符条目被追加到描述符中：

```
("header_match", "<descriptor_value>")
```

```
{
  "descriptor_value": "...",
  "expect_match": "{...}",
  "headers": []
}
```

- descriptor_value

  (string, REQUIRED) 描述符条目中使用的值。

- expect_match

  (BoolValue) 如果设置为true，则该操作将在请求与头部匹配时附加描述符条目。如果设置为false，则该操作将在请求不匹配头部时附加描述符条目。默认值是true。

- headers

  (HeaderMatcher, REQUIRED) 指定速率限制操作应匹配的一组头部字段。该动作将检查请求的头部与配置中的所有指定头。如果配置中的所有头都存在于具有相同值的请求中（或者如果值字段不在配置中则基于存在），则匹配将发生。

HeaderMatcher

HeaderMatcher proto

注意：

1. 在内部，Envoy始终使用HTTP/2的`:authority`头来表示HTTP/1`Host`头。因此，如果试图在主机上匹配，则匹配`:authority`。
2. 要使用HTTP方法进行路由，请使用HTTP/2特殊的`:method`头。这适用于HTTP/1和HTTP/2，做为Envoy标准化头。例如。

```
{
  "name": ":method",
  "value": "POST"
}
```

```
{
  "name": "...",
  "value": "...",
  "regex": "{...}"
}
```

- name

  (string, REQUIRED) 指定请求的头部名称。

- value

  (string) 指定头部的值。如果值不存在，则将匹配具有头部的请求，而不关心具体的值。

- regex

  (BoolValue) 指定匹配头部值是否采用正则表达式。默认为false。整个请求头的值必须与正则表达式匹配。如果只有请求头的值部分与正则表达式相匹配，则规定不匹配。这里定义了值字段中使用的正则表达式语法。

  示例：

  - 正则表达式 `d{3}` 匹配值 `123`
  - 正则表达式 `d{3}` 不匹配值 `1234`
  - 正则表达式 `d{3}` 不匹配值 `123.456`

返回

- 上一级
- 首页目录

TLS配置

TLS配置

- DataSource
- TlsParameters
- TlsParameters.TlsProtocol (Enum)
- TlsCertificate
- TlsSessionTicketKeys
- CertificateValidationContext
- CommonTlsContext
- UpstreamTlsContext
- DownstreamTlsContext

DataSource

DataSource proto

```
{
  "filename": "..."
}
```

- filename

  (string, REQUIRED) 本地文件系统数据源。必须设置一个文件名。

TlsParameters

TlsParameters proto

```
{
  "tls_minimum_protocol_version": "...",
  "tls_maximum_protocol_version": "...",
  "cipher_suites": [],
  "ecdh_curves": []
}
```

- tls_minimum_protocol_version

  (TlsParameters.TlsProtocol) 容许的最小TLS协议版本。

- tls_maximum_protocol_version

  (TlsParameters.TlsProtocol) 容许的最大TLS协议版本。

- cipher_suites

  (string) 如果指定，则TLS监听器将仅支持指定的密码套件。如果未指定，则默认列表：

  `
  [ECDHE-ECDSA-AES128-GCM-SHA256|ECDHE-ECDSA-CHACHA20-POLY1305] [ECDHE-RSA-AES128-GCM-SHA256|ECDHE-RSA-CHACHA20-POLY1305]
  ECDHE-ECDSA-AES128-SHA256
  ECDHE-RSA-AES128-SHA256
  ECDHE-ECDSA-AES128-SHA
  ECDHE-RSA-AES128-SHA
  AES128-GCM-SHA256
  AES128-SHA256
  AES128-SHA
  ECDHE-ECDSA-AES256-GCM-SHA384
  ECDHE-RSA-AES256-GCM-SHA384
  ECDHE-ECDSA-AES256-SHA384
  ECDHE-RSA-AES256-SHA384
  ECDHE-ECDSA-AES256-SHA
  ECDHE-RSA-AES256-SHA
  AES256-GCM-SHA384
  AES256-SHA256
  AES256-SHA

  > 将会被使用。

- ecdh_curves

  (string) 如果指定，TLS连接将只支持指定的ECDH密钥交换。如果未指定，将使用默认（X25519，P-256）。

TlsParameters.TlsProtocol (Enum)

TlsParameters.TlsProtocol proto

- TLS_AUTO

  (DEFAULT) Envoy将选择最佳的TLS版本。

- TLSv1_0

TLS 1.0

- TLSv1_1

  TLS 1.1
- TLSv1_2

  TLS 1.2
- TLSv1_3

  TLS 1.3

TlsCertificate

TlsCertificate proto

```
{
  "certificate_chain": "{...}",
  "private_key": "{...}"
}
```

- certificate_chain

  (DataSource) TLS证书链。

- private_key

  (DataSource) TLS私钥。

TlsSessionTicketKeys

TlsSessionTicketKeys proto

```
{
  "keys": []
}
```

- keys

  (DataSource, REQUIRED)
  TLS会话的加解密的密钥。数组中的第一个密钥将作所有新会话加密的上下文。所有密钥都将用来解密所收到的凭证。这允许通过，先放置新的密钥在前面，第二个是旧的密钥，用于实现密钥轮换。

  如果未指定session_ticket_keys，那么TLS库仍将支持通过故障恢复单个会话，但会使用内部生成和

管理的密钥，因此会话不能在热重启或不同的主机上恢复。

每个密钥必须包含完全80字节的密码安全随机数据。例如，`openssl rand 80` 的输出。

注意：使用此功能需要考虑严重的安全风险。即使使用了支持完美前向保密的密码，对密钥的不正确处理也可能导致连接的保密性丧失。有关讨论，请参阅讨论。为了最大限度地降低风险，您必须：

- 保持会话凭证密钥至少与TLS证书私钥一样安全。
- 至少每天轮换会话凭证密钥，最好每小时轮换一次。
- 始终使用密码安全的随机数据源生成密钥。

CertificateValidationContext

CertificateValidationContext proto

```
{
  "trusted_ca": "{...}",
  "verify_certificate_hash": [],
  "verify_subject_alt_name": []
}
```

- trusted_ca

  (DataSource) TLS证书数据包含颁发机构证书，提供用于验证客户端的证书。如果未指定并且提供了客户端证书，则不会进行验证。默认情况下，校验客户端证书是可选的，除非还指定了其中一个附加选项（`require_client_certificate`，`verify_certificate_hash`或`verify_subject_alt_name`）。

- verify_certificate_hash

  (string) 如果指定，Envoy将验证（pin）所提供证书的十六进制编码的SHA-256散列。

- verify_subject_alt_name

  (string) 可选，标题替代名称列表。如果指定，Envoy将验证证书的标题替代名称是否与指定其中一个值匹配。

CommonTlsContext

CommonTlsContext proto

客户端和服务器TLS上下文共享。

```
{
  "tls_params": "{...}",
  "tls_certificates": [],
```

```
  "validation_context": "{...}",
  "alpn_protocols": []
}
```

- tls_params

  (TlsParameters) TLS协议版本，算法套件等。

- tls_certificates

  (TlsCertificate) 多个TLS证书可以与相同的上下文关联。 例如：为了同时允许RSA和ECDSA证书，可以配置两个TLS证书。

  注意：虽然这是一个列表，但是目前只支持单个证书。这将在未来放宽。

- validation_context

  (CertificateValidationContext) 如何验证对等证书。

- alpn_protocols

  (string) 提供监听器应该公开的ALPN协议列表。实际上，这可能会被设置为两个值之一（有关更多信息，请参阅HTTP连接管理器中的codec_type参数）：

  - “h2,http/1.1” 如果监听器要同时支持HTTP/2和HTTP/1.1。
  - “http/1.1” 如果监听器只支持HTTP/1.1。

  这个参数没有默认值。如果为空，Envoy将不会暴露ALPN。

UpstreamTlsContext

UpstreamTlsContext proto

```
{
  "common_tls_context": "{...}",
  "sni": "..."
}
```

- common_tls_context

  (CommonTlsContext) 常见的TLS上下文设置。

- sni

(string) 创建TLS后端连接时使用的SNI字符串。

DownstreamTlsContext

DownstreamTlsContext proto

```
{
  "common_tls_context": "{...}",
  "require_client_certificate": "{...}",
  "session_ticket_keys": "{...}"
}
```

- common_tls_context

  (CommonTlsContext) 常见的TLS上下文设置。

- require_client_certificate

  (BoolValue) 如果指定，Envoy将拒绝没有有效客户端证书的连接。

- session_ticket_keys

  (TlsSessionTicketKeys) TLS会话凭证密钥设置。

  注意：只有一个 `session_ticket_keys` 可被设置。

返回

- 上一级
- 首页目录

通用的类型

通用类型

Locality

Locality proto

标识Envoy运行所在的位置或上游主机运行所在的位置。

```
{
  "region": "...",
  "zone": "...",
  "sub_zone": "..."
}
```

- region

  (string) 当前属于哪个区域

- zone

  (string) AWS上的可用区域（AZ），GCP区域等

- sub_zone

  (string) 当用于上游主机的位置时，该字段进一步将区域分成更小的子区域，从而可以独立地进行负

载平衡。

Node

Node proto

标识特定的Envoy实例。节点标识符将呈现给管理服务器，管理服务器可以使用该标识符来区分每个Envoy服务的配置。

```
{
  "id": "...",
  "cluster": "...",
  "metadata": "{...}",
  "locality": "{...}",
  "build_version": "..."
}
```

- id

  (string) Envoy节点的标识符。

- cluster

  (string) Envoy节点所属的群集。

- metadata

  (Struct) 所在节点的扩展元数据，Envoy将直接传递给管理服务器。

- locality

  (Locality) 指定Envoy实例的运行位置。

- build_version

  (string) 在金丝雀（灰度发布）期间管理服务器，知道哪个版本的Envoy正在进行测试。这将由Envoy在管理服务器RPC中设置。

Endpoint

Endpoint proto

上游主机标识符

```
{
  "address": "{...}"
}
```

- address

  (Address)

Metadata

Metadata proto

元数据用于监听器的匹配场景，为路由、端口、过滤器链等提供了额外的输入。它的结构是从过滤器名称（反向DNS格式）到特定的过滤器元数据的映射。过滤器的元数据key/value将合并在连接并发生请求时处理，同一个key的更新值将覆盖旧值。

元数据的使用示例，在HTTP连接管理追加附加信息，将体现在`envoy.http_connection_manager.access_log`命名空间。

为了实现负载平衡，元数据提供了一种集群端口子集的方法。端口匹配关联的元数据对象，路由匹配关联的元数据对象。当前有一些定义的元数据用于此目的：

- `{"envoy.lb": {"canary": <bool> }}` 这表明了一个端口的`canary`状态，并且用于头部（x-envoy-upstream-canary）和统计处理。

```
{
  "filter_metadata": "{...}"
}
```

- filter_metadata

  (map<string, Struct>) key是反向DNS过滤器名称，例如`com.acme.widget`。命名空间`envoy.*`保留给Envoy内置过滤器使用。

RuntimeUInt32

RuntimeUInt32 proto

若没有指定时，则在运行时生成的uint32默认值。

```
{
  "default_value": "...",
  "runtime_key": "..."
}
```

- default_value

  (uint32) 默认值，运行时没有可用的值时。

- runtime_key

  (string, REQUIRED) 运行时，通过key以获取相应的value。如果定义，则使用此值。

HeaderValue

HeaderValue proto

Header键值对。

```
{
  "key": "...",
  "value": "..."
}
```

- key

  (string) Header Key.

- value

  (string) Header Value.

  HTTP访问日志记录的格式说明符可以在此处应用，但未知的Header值将替换为空字符串而不是`-`。

HeaderValueOption

HeaderValueOption proto

Header键值对追加控制选项。

```
{
  "header": "{...}",
  "append": "{...}"
}
```

- header

  (HeaderValue) 控制选项所应用的Header键值对。

- append

  (BoolValue) 是否添加的开关，如果为true（默认值），则该值将附加到现有值。

ApiConfigSource

ApiConfigSource proto

API配置源。这标识了Envoy将用来获取xDS的API类型和群集。

```
{
  "api_type": "...",
  "cluster_name": [],
  "refresh_delay": "{...}"
}
```

- api_type

  (ApiConfigSource.ApiType) API类型。

- cluster_name

  (string, REQUIRED) 可以提供多个群集名称。如果定义了大于1个集群，如果发生任何类型的故障，则将循环切换。

- refresh_delay

  (Duration) 对于REST API，连续轮询之间的间隔。

ApiConfigSource.ApiType (Enum)

ApiConfigSource.ApiType proto

可以通过REST API或gRPC获取。

- REST_LEGACY

  (DEFAULT) REST-JSON对应于传统v1 API。

- REST

  REST-JSON v2 API，将使用v2 protos规范的JSON编码。

- GRPC

  gRPC v2 API。

AggregatedConfigSource

AggregatedConfigSource proto

聚合发现服务（ADS）选项。这目前是空的，但在ConfigSource中设置时可以用来指定要使用ADS。

```
{}
```

ConfigSource

ConfigSource proto

监听器，集群，路由，端口等配置可以从文件系统或xDS API源获取。使用 `inotify` 监视文件系统配置以进行更新。

```
{
  "path": "...",
  "api_config_source": "{...}",
  "ads": "{...}"
}
```

- path

  (string) 配置从文件系统路径来源来更新配置。

- api_config_source

  (ApiConfigSource) API配置源。

- ads

  (AggregatedConfigSource) 配置使用ADS将做为配置源。将使用引导程序配置中的ADS API配置源。

  注意：必须选择 `path` , `api_config_source` , `ads` 其中一个选项配置。

TransportSocket

TransportSocket proto

监听器和集群中传输套接字的配置。如果配置为空，则将根据 `tls_context` 的平台和现有的来选择默认的传输套接字实现和配置。

```
{
  "name": "...",
  "config": "{...}"
}
```

- name

  (string, REQUIRED) 要实例化的传输套接字的名称。该名称必须匹配支持的传输套接字实现。

- config

  (Struct) 具体将要被实例化的传输套接字配置。请参阅支持的传输套接字实现以获取更多信息。

RoutingPriority (Enum)
RoutingPriority proto

Envoy在路由和虚拟集群级别，都支持上游优先级路由。当前的实现是针对每个优先级别，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，两个物理连接也将被用于上游主机。将来，Envoy可能会支持真正的HTTP/2优先级，而不是单个上行连接。

- DEFAULT

  (DEFAULT)

- HIGH

RequestMethod (Enum)
RequestMethod proto

HTTP请求方法

- METHOD_UNSPECIFIED

  (DEFAULT)
- GET
- HEAD
- POST
- PUT
- DELETE
- CONNECT
- OPTIONS
- TRACE

返回
- 上一级
- 首页目录

网络地址

网络地址

注意：v2 API差异：现有 `.proto` 地址结构。

- Pipe
- SocketAddress
- SocketAddress.Protocol (Enum)
- BindConfig
- Address
- CidrRange

Pipe

Pipe proto

```
{
  "path": "..."
}
```

- path

  (string, REQUIRED) Unix域套接字路径。

SocketAddress

SocketAddress proto

```
{
  "protocol": "...",
  "address": "...",
  "port_value": "...",
  "named_port": "...",
  "resolver_name": "..."
}
```

- protocol

  (SocketAddress.Protocol)

- address

  (string) 套接字的地址。监听器将绑定到该地址或出站连接地址。若配置为一个空的地址，意味着将

绑定到0.0.0.0或::。在连接之后，仍然可以通过 `FilterChainMatch` 中的匹配前缀/后缀来区分地址。对于群集，可以是通过DNS解析的IP或主机名。如果是主机名，除非需要默认（即DNS）解析，否则应该设置resolver_name。

- port_value

  (uint32)

- named_port

  (string) 这只有在下面指定了 `resolver_name` 并且指定的解析器能够进行命名的端口解析时才有效。

  注意： `port_value` 、 `named_port` 必须选其中一个设置。

- resolver_name

  (string) 解析器的名称。这一定是在Envoy注册的。如果这是空的，则应用依赖于上下文的默认值。如果地址是主机名，则应该设置DNS以外的解决方案。如果地址是一个具体的IP地址，则不会发生解析。

SocketAddress.Protocol (Enum)

SocketAddress.Protocol proto

- TCP

  (DEFAULT)

BindConfig

BindConfig proto

```
{
  "source_address": "{...}"
}
```

- source_address

  (SocketAddress, REQUIRED) 创建套接字时绑定的地址。

Address

Address proto

指定逻辑或物理地址和端口，这些地址和端口用于告诉Envoy绑定/监听的地址，连接到上游并查找相应的

管理服务器。

```
{
  "socket_address": "{...}",
  "pipe": "{...}"
}
```

- socket_address

  (SocketAddress)

- pipe

  (Pipe)

  注意：必须设置一个正确的socket_address或者Pipe。

CidrRange

CidrRange proto

CidrRange指定一个IP地址和前缀长度来构造一个CIDR范围的子网掩码。

```
{
  "address_prefix": "...",
  "prefix_len": "{...}"
}
```

- address_prefix

  (string, REQUIRED) IPv4或IPv6地址，例如 192.0.0.0或2001:db8::。

- prefix_len

  (UInt32Value) 前缀的长度，例如：0或者32。

返回

- 上一级
- 首页目录

协议选项

HTTP协议选项

- Http1ProtocolOptions
- Http2ProtocolOptions

Http1ProtocolOptions

Http1ProtocolOptions proto

```
{
  "allow_absolute_url": "{...}"
}
```

- allow_absolute_url

  (BoolValue) 在请求中使用绝对URL处理HTTP请求。这些请求通常由客户端发送到转发/显式代理。这允许客户端将envoy配置为他们的HTTP代理。例如，在Unix中，这通常是通过设置 `http_proxy` 环境变量来完成的。

Http2ProtocolOptions

Http2ProtocolOptions proto

```
{
  "hpack_table_size": "{...}",
  "max_concurrent_streams": "{...}",
  "initial_stream_window_size": "{...}",
  "initial_connection_window_size": "{...}"
}
```

- hpack_table_size

  (UInt32Value) 允许编码器使用动态HPACK表的最大表大小（以八位字节为单位）。有效值范围从0到4294967295（2^32-1），默认值为4096。0表示禁用头部压缩。

- max_concurrent_streams

  (UInt32Value) 在一个HTTP/2连接上允许最大流大小。有效值范围从1到2147483647（2^31-1），默认值为2147483647。

- initial_stream_window_size

(UInt32Value) 这个字段也可以作为Envoy在HTTP/2编解码缓冲区中缓冲每个流的字节限制。一旦缓冲区到达这个值，将触发停止数据流发送到编解码缓冲区。

- initial_connection_window_size

  (UInt32Value) 与 `initial_stream_window_size` 类似，但是用于连接级流量控制窗口。目前，这与 `initial_stream_window_size` 具有相同的最小/最大/默认值。

返回

- 上一级
- 首页目录

发现API

通用发现服务接口

- DiscoveryRequest
- DiscoveryResponse

DiscoveryRequest

DiscoveryRequest proto

发现请求：是指在通过某些API，为Envoy节点请求一组相同类型的带版本标签的资源。

```
{
  "version_info": "...",
  "node": "{...}",
  "resource_names": [],
  "type_url": "...",
  "response_nonce": "..."
}
```

- version_info

  (string) 请求消息所携带的版本信息，将是最近成功处理的响应中收到的版本信息，第一个请求中`version_info`为空。在收到响应之前不会发送新的请求，直到Envoy实例准备好ACK/NACK新配置为止。ACK/NACK分别通过返回应用的新API配置版本或先前的API配置版本来进行。每个`type_url`（见下文）都有一个独立的版本信息。

- node

  (Node) 发出请求的节点信息

- resource_names

  (string) 要订阅的资源列表，群集名称列表或路由配置名称。如果为空，则返回该API的所有资源。LDS/CDS期望`resource_names`为空，因为这是Envoy实例的全局资源。LDS和CDS响应将意味着需要通过EDS/RDS获取的一些资源，这些资源将会在`resource_names`中列出。

- type_url

  (string) 正在请求的资源的类型，例如“type.googleapis.com/envoy.api.v2.ClusterLoadAssignment”。 在单独的xDS API（例如CDS，LDS等）的请求中，资源类型不可见，但对于ADS是必需的。

- response_nonce

  (string) 对应于DiscoveryResponse的nonce的ACK/NACK。请参阅关于 `version_info` 和 DiscoveryResponse nonce上述的讨论。如果 `nonce` 不可用，则这可能是空的。以支持启动或非流xDS的实现。

DiscoveryResponse

DiscoveryResponse proto

```
{
  "version_info": "...",
  "resources": [],
  "type_url": "...",
  "nonce": "..."
}
```

- version_info

  (string) 响应数据的版本信息

- resources

  (Any) 响应资源。关于这些资源的类型，取决于被调用的API。

- type_url

  (string) 资源的URL。如果资源非空，任何消息中的 `type_url` 必须与资源保持一致。这可以有效地识别在ADS上混合提供的xDS API。

- nonce

  (string) 对于基于gRPC的订阅，nonce提供了一种方法来显式确认以下 `DiscoveryRequest` 中的特定 `DiscoveryResponse` 。在此发现响应之前，Envoy可能会向流管理服务器发送其他消息，以便在响应发送时未经处理。nonce允许管理服务器忽略前一版本的任何进一步发现请求，直到当前的发现请求。nonce是可选的，对于基于非流的xDS实现，不是必需的。

返回

- 上一级
- 首页目录

限速组件

通用的限速组件

- RateLimitDescriptor
- RateLimitDescriptor.Entry

RateLimitDescriptor

RateLimitDescriptor proto

`RateLimitDescriptor` 是服务所使用的分层条目列表，用于确定最终的速率限制key和整体限制。这里有一些使用“Envoy”为域名的例子。

```
["authenticated": "false"], ["remote_address": "10.0.0.1"]
```

功能：限制所有IP地址为 `10.0.0.1` 未经身份验证的流量。该配置key为 `remote_address` 使用默认限制。如果希望提高 `10.0.0.1` 的限制或完全阻止，可以直接在配置中指定。

```
["authenticated": "false"], ["path": "/foo/bar"]
```

它做什么：在全局范围内为一个特定的路径（或前缀，如果在服务中配置的方式）限制所有未经身份验证的请求。

```
["authenticated": "false"], ["path": "/foo/bar"], ["remote_address": "10.0.0.1"]
```

功能：将未经验证的流量限制为特定IP地址和路径。像（1）我们可以提高/阻止特定的IP地址，如果我们想要一个覆盖配置。

```
["authenticated": "true"], ["client_id": "foo"]
```

它做什么：限制一个经过身份验证的客户端“foo”的所有流量。

```
["authenticated": "true"], ["client_id": "foo"], ["path": "/foo/bar"]
```

它做什么：限制流量到一个经过验证的客户端的特定路径“foo”

API背后的想法是，如果需要，(1)/(2)/(3)和(4)/(5)可以在1个请求中发送。这使得构建具有通用复杂后端应用场景的成为可能。

```
{
  "entries": []
}
```

- entries

  (RateLimitDescriptor.Entry, REQUIRED) 描述符条目列表。

RateLimitDescriptor.Entry

RateLimitDescriptor.Entry proto

```
{
  "key": "...",
  "value": "..."
}
```

- key

  (string, REQUIRED) key描述符。

- value

  (string, REQUIRED) value描述符。

返回

- 上一级
- 首页目录

过滤器

过滤器

- 网络过滤器
  - TLS客户端身份认证
  - HTTP连接管理
  - Mongo代理
  - 速率限制
  - Redis代理
  - TCP代理
- HTTP过滤器
  - 缓存
  - 故障注入
  - 健康检查
  - Lua
  - 速率限制
  - 路由
  - gRPC-JSON转码器
- 常见访问日志类型
  - filter.accesslog.AccessLog
  - filter.accesslog.AccessLogFilter
  - filter.accesslog.ComparisonFilter
  - filter.accesslog.ComparisonFilter.Op (Enum)
  - filter.accesslog.StatusCodeFilter
  - filter.accesslog.DurationFilter
  - filter.accesslog.NotHealthCheckFilter
  - filter.accesslog.TraceableFilter
  - filter.accesslog.RuntimeFilter
  - filter.accesslog.AndFilter
  - filter.accesslog.OrFilter
  - filter.accesslog.FileAccessLog
- 常见故障注入类型
  - filter.FaultDelay
  - filter.FaultDelay.FaultDelayType (Enum)

返回

- 上一级
- 首页目录

网络过滤器

网络过滤器

- TLS客户端身份认证
  - filter.network.ClientSSLAuth
- HTTP连接管理
  - filter.network.HttpConnectionManager
  - filter.network.HttpConnectionManager.Tracing
  - filter.network.HttpConnectionManager.Tracing.OperationName (Enum)
  - filter.network.HttpConnectionManager.SetCurrentClientCertDetails
  - filter.network.HttpConnectionManager.CodecType (Enum)
  - filter.network.HttpConnectionManager.ForwardClientCertDetails (Enum)
  - filter.network.Rds
  - filter.network.HttpFilter
- Mongo代理
  - filter.network.MongoProxy
- 速率限制
  - filter.network.RateLimit
- Redis代理
  - filter.network.RedisProxy
  - filter.network.RedisProxy.ConnPoolSettings
- TCP代理
  - filter.network.TcpProxy
  - filter.network.TcpProxy.DeprecatedV1
  - filter.network.TcpProxy.DeprecatedV1.TCPRoute

返回

- 上一级
- 首页目录

TLS客户端身份认证

TLS客户端身份认证

- filter.network.ClientSSLAuth

TLS客户端身份认证配置概述

filter.network.ClientSSLAuth

filter.network.ClientSSLAuth proto

```
{
  "auth_api_cluster": "...",
  "stat_prefix": "...",
  "refresh_delay": "{...}",
  "ip_white_list": []
}
```

- auth_api_cluster

  (string, REQUIRED) 运行身份验证服务的群集管理器群集。过滤器将每60秒连接到该服务以获取主体列表。该服务必须支持期望的REST API。

- stat_prefix

  (string, REQUIRED) 发布统计信息时所使用的前缀。

- refresh_delay

  (Duration) 刷新验证服务主体时间间隔（以毫秒为单位）。默认是60000（60s）。实际的刷新时间是这个值加上0到 `refresh_delay_ms` 之间的随机抖动值。

- ip_white_list

  (CidrRange) 一个可选的IP地址和子网掩码列表，提供白名单列表给供过器使用。如果没有提供列表，则没有IP白名单。

返回

- 上一级
- 首页目录

HTTP连接管理

HTTP连接管理

- filter.network.HttpConnectionManager
- filter.network.HttpConnectionManager.Tracing
- filter.network.HttpConnectionManager.Tracing.OperationName (Enum)
- filter.network.HttpConnectionManager.SetCurrentClientCertDetails
- filter.network.HttpConnectionManager.CodecType (Enum)
- filter.network.HttpConnectionManager.ForwardClientCertDetails (Enum)
- filter.network.Rds
- filter.network.HttpFilter

HTTP连接管理配置概述。

filter.network.HttpConnectionManager

filter.network.HttpConnectionManager proto

```
{
  "codec_type": "...",
  "stat_prefix": "...",
  "rds": "{...}",
  "route_config": "{...}",
  "http_filters": [],
  "add_user_agent": "{...}",
  "tracing": "{...}",
  "http_protocol_options": "{...}",
  "http2_protocol_options": "{...}",
  "server_name": "...",
  "idle_timeout": "{...}",
  "drain_timeout": "{...}",
  "access_log": [],
  "use_remote_address": "{...}",
  "generate_request_id": "{...}",
  "forward_client_cert_details": "...",
  "set_current_client_cert_details": "{...}"
}
```

- codec_type

  (filter.network.HttpConnectionManager.CodecType) 应用与连接管理器的编解码器类型。

- stat_prefix

  (string, REQUIRED) 连接管理器发布的统计信息所使用的前缀。有关更多信息，请参阅统计文档。

- rds

(filter.network.Rds) 通过RDS API动态加载连接管理器的路由表。

必须正确设置 `rds` , `route_config` 其中一个。

- route_config

  (RouteConfiguration) 在此属性中指定静态的连接管理器的路由表。

  必须正确设置 `rds` , `route_config` 其中一个。

- http_filters

  (filter.network.HttpFilter) 构成连接管理器请求的过滤器链，包括各个HTTP过滤器的列表。当请求发生时，将按照顺序处理过滤器。

- add_user_agent

  (BoolValue) 连接管理器是否处理user-agent和x-envoy-downstream-service-cluster头。有关更多信息，请参阅相应的链接。默认为false。

- tracing

  (filter.network.HttpConnectionManager.Tracing) 是否定义对象，决定连接管理器是否将跟踪数据发送到已配置的跟踪服务程序中。

- http_protocol_options

  (Http1ProtocolOptions) 传递给HTTP/1编解码器，额外的HTTP/1设置选项。

- http2_protocol_options

  (Http2ProtocolOptions) 额外的HTTP/2设置选项，直接传递给HTTP/2编解码器。

- server_name

  (string) 连接管理器将在响应头中写入响应的服务名。如果未设置，则默认为Enovy。

- idle_timeout

  (Duration) 由连接管理器管理的连接空闲超时时长。在没有活动的请求时，持续的时间超过设定的阈值，则认为连接超时。如果没有设置，则没有空闲超时。当达到空闲超时后，连接将被关闭。如果连接是HTTP/2连接，则在关闭连接之前会发生顺序排空。看drain_timeout。

- drain_timeout

  (Duration) Envoy将在发送HTTP/2“关闭通知”（GOAWAY帧与最大流ID）和最终GOAWAY帧之间等待的时间。这是为了让Envoy支持与最后GOAWAY帧竞争的新的流处理，所提供的宽限期。在这个宽限期间，Envoy将继续接受新的流。在宽限期之后，最终GOAWAY帧被发送，Envoy将开始拒绝新的流。在连接遇到空闲超时或通用服务器耗尽时都会发生排空。如果未指定此选项，则默认宽限期为5000毫秒（5秒）。

- access_log

  (filter.accesslog.AccessLog) 从连接管理器发出的HTTP访问日志的配置。

- use_remote_address

  (BoolValue) 如果设置为true，连接管理器将在确定内部和外部源以及操作各种头部时使用客户端连接的真实远程地址。如果设置为false或不存在，连接管理器将使用`x-forwarded-for` HTTP头。有关更多信息，请参阅x-forwarded-for，x-envoy-internal和x-envoy-external-address的文档。

- generate_request_id

  (BoolValue) 连接管理器是否会自动生成x-request-id头，如果该头不存在。默认为true。生成一个随机的UUID4（性能代价比较大），所以在高吞吐量的情况下，这个功能是不需要的，它可以被禁用。

- forward_client_cert_details

  (filter.network.HttpConnectionManager.ForwardClientCertDetails) 如何处理x-forward-client-cert（XFCC）HTTP头。

- set_current_client_cert_details

  (filter.network.HttpConnectionManager.SetCurrentClientCertDetails) 只有在forward_client_cert_details为`APPEND_FORWARD`或`SANITIZE_SET`且客户端连接为`mTLS`时，此字段才有效。它指定要转发的客户端证书中的字段。请注意，在x-forwarded-client-cert头中，始终设置`Hash`，并在客户端证书显示SAN值时始终设置`By`。

filter.network.HttpConnectionManager.Tracing

filter.network.HttpConnectionManager.Tracing proto

```
{
  "operation_name": "...",
  "request_headers_for_tags": []
```

```
}
```

- operation_name

  (filter.network.HttpConnectionManager.Tracing.OperationName) span名将由此字段生成。

- request_headers_for_tags

  (string) 用于为活动span创建标签的标题名称列表。该标题名用于填充span标记名，标题值用于填充span标记值。如果指定头的名称出现在请求头中，则会创建该标签。

filter.network.HttpConnectionManager.Tracing.OperationName (Enum)
filter.network.HttpConnectionManager.Tracing.OperationName proto

- INGRESS
  (DEFAULT) 标记HTTP监听器用于入站/入口请求。

- EGRESS
  标记HTTP监听器用于出站/出口请求。

filter.network.HttpConnectionManager.SetCurrentClientCertDetails (Enum)
filter.network.HttpConnectionManager.SetCurrentClientCertDetails proto

```
{
  "subject": "{...}",
  "san": "{...}"
}
```

- subject

  (BoolValue) 是否转发客户端证书的标题。默认为false。

- san

  (BoolValue) 是否转发客户端证书的SAN。默认为false。

filter.network.HttpConnectionManager.CodecType (Enum)
filter.network.HttpConnectionManager.CodecType proto

- AUTO

  (DEFAULT) 对于每个新的连接，连接管理器将自行决定使用哪个编解码器。此模式支持TLS监听器的ALPN以及监听器明文的协议。如果ALPN数据可用，则优选，否则使用协议解析。在几乎所有情况

下，这是优选的设置。

- HTTP1

  连接管理器将假定客户端使用HTTP/1.1协议。

- HTTP2

  连接管理器将假定客户端使用HTTP/2（Envoy不需要通过TLS发送HTTP/2或者使用ALPN，事先预知的）。

filter.network.HttpConnectionManager.ForwardClientCertDetails (Enum)

filter.network.HttpConnectionManager.ForwardClientCertDetails proto

如何处理x-forward-client-cert（XFCC）HTTP头。

- SANITIZE

  (DEFAULT) 不要将XFCC头部发送到下一跳。这是默认值。

- FORWARD_ONLY

  当客户端连接是mTLS（Mutual TLS）时，转发请求中的XFCC头。

- APPEND_FORWARD

  当客户端连接是mTLS时，将客户端证书信息附加到请求的XFCC头并转发它。

- SANITIZE_SET

  当客户端连接是mTLS时，使用客户端证书信息重置XFCC头，并将其发送到下一个跃点。

- ALWAYS_FORWARD_ONLY

  始终在请求中转发XFCC头，而不管客户端连接是否为mTLS。

filter.network.Rds

filter.network.Rds proto

```
{
  "config_source": "{...}",
  "route_config_name": "..."
}
```

- config_source

  (ConfigSource, REQUIRED) RDS的配置源描述符。

- route_config_name

  (string, REQUIRED) 配置的路由名称。这个名字将被传递给 `RDS API` 。这允许配置多个HTTP监听器（和关联的HTTP连接管理器）使用不同的路由配置。

filter.network.HttpFilter

filter.network.HttpFilter proto

```
{
  "name": "...",
  "config": "{...}"
}
```

- name

  (string, REQUIRED) 要实例化的过滤器的名称。该名称必须与支持的过滤器匹配。内置的过滤器有：

  - envoy.buffer
  - envoy.cors
  - envoy.fault
  - envoy.http_dynamo_filter
  - envoy.grpc_http1_bridge
  - envoy.grpc_json_transcoder
  - envoy.grpc_web
  - envoy.health_check
  - envoy.lua
  - envoy.rate_limit
  - envoy.router

- config

  (Struct) 指定的过滤器配置，这取决于被实例化的过滤器。有关更多文档，请参阅支持的过滤器。

返回

- 上一级
- 首页目录

Mongo代理

Mongo代理

MongoDB配置参考。

filter.network.MongoProxy

filter.network.MongoProxy proto

```
{
  "stat_prefix": "...",
  "access_log": "...",
  "delay": "{...}"
}
```

- stat_prefix

  (string, REQUIRED) 发布统计信息时使用的前缀（提升易读性）。

- access_log

  (string) 可选路径，用于记录Mongo的访问日志。如果未指定访问日志路径，则不会写入访问日志。请注意，访问日志也受运行时配置控制。

- delay

  (filter.FaultDelay) 在代理Mongo操作之前注入一个固定的延迟。延迟适用于以下MongoDB操作：`Query`，`Insert`，`GetMore`和`KillCursors`。一旦进行一个有效延迟时，在定时器触发之前，在所有数据输入的时间也将作为延迟的一部分。

返回

- 上一级
- 首页目录

速率限制

速率限制

速率限制配置参考。

filter.network.RateLimit

filter.network.RateLimit proto

```
{
  "stat_prefix": "...",
  "domain": "...",
  "descriptors": [],
  "timeout": "{...}"
}
```

- stat_prefix

  (string, REQUIRED) 发布统计信息时使用的前缀。

- domain

  (string, REQUIRED) 用于速率限制服务请求中的域。

- descriptors

  (RateLimitDescriptor, REQUIRED) 用于速率限制服务请求中的速率限制描述符列表。

- timeout

  (Duration) 速率限制服务RPC的超时时间（以毫秒为单位）。如果未设置，则默认为20ms。

返回

- 上一级
- 首页目录

Redis代理

Redis代理

Redis代理的配置参考。

- filter.network.RedisProxy
- filter.network.RedisProxy.ConnPoolSettings

filter.network.RedisProxy

filter.network.RedisProxy proto

```
{
  "stat_prefix": "...",
  "cluster": "...",
  "settings": "{...}"
}
```

- stat_prefix

  (string, REQUIRED) 发布统计信息时使用的前缀。

- cluster

  (string, REQUIRED) 集群管理器的集群名称。有关配置支持群集的建议，请参阅架构概述的配置部分。

- settings

  (filter.network.RedisProxy.ConnPoolSettings, REQUIRED) 连接到上游集群的网络连接池设置。

filter.network.RedisProxy.ConnPoolSettings

filter.network.RedisProxy.ConnPoolSettings proto

Redis连接池设置。

```
{
  "op_timeout": "{...}"
}
```

- op_timeout

(Duration) 每个操作的超时时长（单位：毫秒）。在第一个命令写入后端连接时启动定时器。从Redis收到的每个响应都会重置定时器，因为它表示下一个命令正在由后端处理。这种行为的唯一例外是到后端的连接尚未建立。在这种情况下，群集上的连接超时将得到控制，直到连接准备就绪。

返回

- 上一级
- 首页目录

TCP代理

TCP代理

TCP代理配置概述。

- filter.network.TcpProxy
- filter.network.TcpProxy.DeprecatedV1
- filter.network.TcpProxy.DeprecatedV1.TCPRoute

filter.network.TcpProxy

filter.network.TcpProxy proto

```
{
  "stat_prefix": "...",
  "cluster": "...",
  "access_log": [],
  "deprecated_v1": "{...}",
  "max_connect_attempts": "{...}"
}
```

- stat_prefix

  (string, REQUIRED) 发布统计信息时使用的前缀。

- cluster

  (string) 要连接到的上游群集。

  注意：一旦在监听器中实现了完整的过滤器链匹配，此字段将成为配置目标集群的唯一方法。所有其他匹配将通过过滤器链匹配规则完成。当不需要其他匹配规则时，此配置非常简单，该字段仍可用于选择群集。否则，需要配置deprecated_v1才能在此期间使用更复杂的路由。

- access_log

  (filter.accesslog.AccessLog) 由此TCP代理发出的访问日志的配置。

- deprecated_v1

  (filter.network.TcpProxy.DeprecatedV1) 使用不推荐使用v1格式的TCP代理过滤器配置。这是复杂路由所必需的，直到实现了监听器中的过滤器链匹配为止。

- max_connect_attempts

(UInt32Value) 在放弃建连之前将尝试的最大失败连接数。如果没有指定参数，则将尝试1次连接。

filter.network.TcpProxy.DeprecatedV1

filter.network.TcpProxy.DeprecatedV1 proto

使用v1格式的TCP代理过滤器配置，直到Envoy能够在监听器级别匹配源/目的地（称为过滤器链匹配）。

```
{
  "routes": []
}
```

- routes

  (filter.network.TcpProxy.DeprecatedV1.TCPRoute, REQUIRED) 过滤器的路由表。所有的过滤器实例都必须有一个路由表，即使它是空的。

filter.network.TcpProxy.DeprecatedV1.TCPRoute

filter.network.TcpProxy.DeprecatedV1.TCPRoute proto

TCP代理的路由是由一组可选的标准L4和一个集群的名称组成。如果下游连接符合所有指定的条件，则路由中的集群将用于相应的上游连接。按照指定的顺序尝试路由，直到找到匹配项。如果找不到匹配，则连接关闭。并没有规定路由始终有效的并且总是产生一个匹配。

```
{
  "cluster": "...",
  "destination_ip_list": [],
  "destination_ports": "...",
  "source_ip_list": [],
  "source_ports": "..."
}
```

- cluster

  (string, REQUIRED) 下游网络连接符合匹配条件时需要连接的群集。

- destination_ip_list

  (CidrRange) 可选的IP地址子网列表，格式为“ip_address/xx”。如果下游连接的目标IP地址，至少被包含在一个指定的子网中，则符合匹配条件。如果未指定参数或列表为空，则将忽略目标IP地址。如果连接已被重定向，则下游连接的目标IP地址可能与代理正在监听的地址不同。

- destination_ports

(string) 可选的字符串，包含通过逗号分隔的端口号列表，或者端口范围。如果下游连接的目标端口，至少一个被包含在指定范围内，则符合匹配条件。如果未指定参数，则将忽略目标端口。如果连接已被重定向，下游连接的目标端口地址可能与代理正在监听的端口不同。

- source_ip_list

  (CidrRange) 可选的IP地址子网列表，格式为“ip_address/xx”。如果下游连接的源IP地址，至少一个被包含在指定的子网中，则符合匹配条件。如果未指定参数或列表为空，则将忽略源IP地址。

- source_ports

  (string) 可选字符串，包含以逗号分隔的端口号列表，或者端口范围。如果下游连接的源端口，至少一个被包含在指定范围内，则满足匹配条件。如果未指定参数，则源端口将被忽略。

返回

- 上一级
- 首页目录

HTTP过滤器

HTTP过滤器

- 缓存
  - filter.http.Buffer
- 故障注入
  - filter.http.FaultAbort
  - filter.http.HTTPFault
- 健康检查
  - filter.http.HealthCheck
- Lua
  - filter.http.Lua
- 速率限制
  - filter.http.RateLimit
- 路由
  - filter.http.Router
- gRPC-JSON转码器
  - filter.http.GrpcJsonTranscoder
  - filter.http.GrpcJsonTranscoder.PrintOptions

返回

- 上一级
- 首页目录

缓存

缓存

缓存配置参考。

filter.http.Buffer

filter.http.Buffer proto

```
{
  "max_request_bytes": "{...}",
  "max_request_time": "{...}"
}
```

- max_request_bytes

  (UInt32Value) 在连接管理器停止缓冲并返回413响应之前，过滤器将缓冲的最大请求大小。

- max_request_time

  (Duration) 过滤器在返回408响应之前，等待完整请求的最长秒数。

返回

- 上一级
- 首页目录

故障注入

故障注入

故障注入配置参考。

filter.http.FaultAbort

filter.http.FaultAbort proto

```
{
  "percent": "...",
  "http_status": "..."
}
```

- percent

  (uint32) 一个介于0到100之间的整数，表示请求/操作/连接通过下面的状态码中止的百分比。

- http_status

  (uint32) 用于中止HTTP请求的HTTP状态码。

  注意：必须设置正确的http_status。

filter.http.HTTPFault

filter.http.HTTPFault proto

```
{
  "delay": "{...}",
  "abort": "{...}",
  "upstream_cluster": "...",
  "headers": [],
  "downstream_nodes": []
}
```

- delay

  (filter.FaultDelay) 如果指定，过滤器将根据配置的值注入延迟。必须指定中止或延迟。

- abort

  (filter.http.FaultAbort) 如果指定，过滤器将根据配置的值中止请求。必须指定中止或延迟。

- upstream_cluster

(string) 指定过滤器所匹配的（目标）上游群集的名称。故障注入将仅限于特定上游群集的请求。

- headers

  (HeaderMatcher) 指定过滤器应匹配的一组头部键值。故障注入过滤器支持根据配置中指定的一组头部匹配请求，来应用故障注入。实际故障注入的概率依赖与百分比字段的值。过滤器会根据配置中的所指定头部来检查请求。如果配置中的所有头部名称以及相应的值都存在于请求中（若没有配置头部的值，则也认为存在），则匹配将发生。

- downstream_nodes

  (string) 针对指定的下游主机列表进行注入故障。如果未设置此设置，则会为所有下游节点注入故障。下游节点名称取自HTTP的 `x-envoy-downstream-service-node` 头，并与下游节点列表进行比较。

返回

- 上一级
- 首页目录

健康检查

健康检查

健康检查配置概述。

filter.http.HealthCheck

filter.http.HealthCheck proto

```
{
  "pass_through_mode": "{...}",
  "endpoint": "...",
  "cache_time": "{...}"
}
```

- pass_through_mode

  (BoolValue, REQUIRED) 指定过滤器是否在传递模式下运行。

- endpoint

  (string, REQUIRED) 传入应被视为健康检查的HTTP端口。例如 `/healthcheck` 。

- cache_time

  (Duration) 如果在传递模式下运行，则过滤器需要缓存上游响应的时间（以毫秒为单位）。

返回

- 上一级
- 首页目录

Lua

Lua
Lua配置概述

filter.http.Lua
filter.http.Lua proto

```
{
  "inline_code": "..."
}
```

- inline_code

  (string, REQUIRED) Envoy将执行的Lua代码。这可以是一个非常小的脚本，如果需要，可以从磁盘进一步加载代码。请注意，如果使用JSON配置，则代码必须能正确转义。YAML配置可能更容易阅读，因为YAML支持多行字符串，所以复杂的脚本可以很容易地内嵌在配置中表示。

返回

- 上一级
- 首页目录

速率限制

速率限制

速率限制配置参考

filter.http.RateLimit

filter.http.RateLimit proto

```
{
  "domain": "...",
  "stage": "...",
  "request_type": "...",
  "timeout": "{...}"
}
```

- domain

  (string, REQUIRED) 需要调用速率限制服务时的域。

- stage

  (uint32) 指定要应用于的相应阶段的速率限制配置编号。如果未设置，则默认为0。

  注意：过滤器支持0到10的阶段编号。

- request_type

  (string) 过滤器应该适用的请求类型。支持内部的，外部的或两者同时。如果将 `x-envoy-internal` 设置为true，则将请求视为内部请求。如果 `x-envoy-internal` 未设置或为false，则请求被视为外部。过滤器默认为两者同时，它将应用于所有的请求类型。

- timeout

  (Duration) 速率限制服务RPC的超时时间（以毫秒为单位）。如果未设置，则默认为20ms。

返回

- 上一级
- 首页目录

路由

路由

路由配置参考

filter.http.Router

filter.http.Router proto

```
{
  "dynamic_stats": "{...}",
  "start_child_span": "...",
  "upstream_log": []
}
```

- dynamic_stats

  (BoolValue) 路由器是否生成动态集群统计信息。默认为true。可以在高性能场景下禁用。

- start_child_span

  (bool) 是否为出口路由请求启动子span。在其他过滤器（`auth`，`ratelimit`等）进行出站请求并且子span的根位于同一入口父span的情况下，这可能很有用。默认为false。

- upstream_log

  (filter.accesslog.AccessLog) 配置由路由器发出的HTTP上游访问日志。上游日志的配置方式与访问日志相同，但是每个日志条目代表上游请求。若配置了重试，则路由器可以为每个下游（入站）请求做出多个上游请求。

返回

- 上一级
- 首页目录

gRPC-JSON转码器

gRPC-JSON 转码器

gRPC-JSON 转码器配置参考。

filter.http.GrpcJsonTranscoder

filter.http.GrpcJsonTranscoder proto

```
{
  "proto_descriptor": "...",
  "services": [],
  "print_options": "{...}"
}
```

- proto_descriptor

  (string, REQUIRED) 为gRPC服务提供二进制 `protobuf` 描述符集合。描述符集合必须包含服务中使用的所有类型。确保为 `protoc` ，并使用 `--include_import` 选项。

  要为gRPC服务生成一个 `protobuf` 描述符集，在运行 `protoc` 之前，还需要从Github中克隆google apis库，因为在 `include` 路径中需要 `annotations.proto` 。

  ```
  git clone https://github.com/googleapis/googleapis
  GOOGLEAPIS_DIR=<your-local-googleapis-folder>
  ```

  然后运行protoc从 `bookstore.proto` 生成描述符：

  ```
  protoc -I$(GOOGLEAPIS_DIR) -I. --include_imports --include_source_info \
  --descriptor_set_out=proto.pb test/proto/bookstore.proto
  ```

  如果您有许多原始源文件，则可以使用通过这个命令来传递所有文件。

- services

  (string, REQUIRED) 提供将要转换的服务名的字符串列表。如果服务名在 `proto_descriptor` 中不存在，Envoy则启动失败。 `proto_descriptor` 可能包含比这里指定的服务名称更多的服务，但是它们不会被转换。

- print_options

  (filter.http.GrpcJsonTranscoder.PrintOptions) 响应JSON的控制选项。这些选项直接传递给 JsonPrintOptions。

filter.http.GrpcJsonTranscoder.PrintOptions

filter.http.GrpcJsonTranscoder.PrintOptions proto

```
{
  "add_whitespace": "...",
  "always_print_primitive_fields": "...",
  "always_print_enums_as_ints": "...",
  "preserve_proto_field_names": "..."
}
```

- add_whitespace

  (bool) 是否添加空格，换行符和缩进以使输出的JSON易于阅读。默认为false。

- always_print_primitive_fields

  (bool) 是否始终打印原始字段。默认情况下，具有默认值的原始字段将在JSON输出中被省略。例如，设置为0的int32字段将被省略。将此标志设置为true，将覆盖默认行为并打印原始字段，而不考虑其值。默认为false。

- always_print_enums_as_ints

  (bool) 是否始终打印枚举作为整数。默认情况下，它们呈现为字符串。默认为false。

- preserve_proto_field_names

  (bool) 是否保留原始字段名称。默认情况下，`protobuf`将使用`json_name`选项生成JSON字段名称，或者按照下面的顺序生成较低的骆驼风格的大小写。设置此标志将保留原始字段名称。默认为false。

返回

- 上一级
- 首页目录

常见访问日志类型

常见访问日志类型

Envoy访问日志记录的是在一段固定的时间内通过Envoy进行入站的交互，典型场景包括单个请求/响应交互（例如HTTP），流（例如通过HTTP 2/gRPC）或连接代理（例如TCP等）。访问日志包含协议指定的`protobuf`消息中定义的字段。

除非另有明确声明，否则所有字段都描述Envoy与连接的客户端之间的下游交互。描述上游交互的字段将在其名称中明确包含上游。

- filter.accesslog.AccessLog
- filter.accesslog.AccessLogFilter
- filter.accesslog.ComparisonFilter
- filter.accesslog.ComparisonFilter.Op (Enum)
- filter.accesslog.StatusCodeFilter
- filter.accesslog.DurationFilter
- filter.accesslog.NotHealthCheckFilter
- filter.accesslog.TraceableFilter
- filter.accesslog.RuntimeFilter
- filter.accesslog.AndFilter
- filter.accesslog.OrFilter
- filter.accesslog.FileAccessLog

filter.accesslog.AccessLog

filter.accesslog.AccessLog proto

```
{
  "name": "...",
  "filter": "{...}",
  "config": "{...}"
}
```

- name

  (string) 要实例化的访问日志实现的名称。 该名称必须与静态注册的访问日志匹配。当前的内置记录器为：1） “envoy.file_access_log”

- filter

  (filter.accesslog.AccessLogFilter) 在写入访问日志时需要使用的过滤器。

- config

  (Struct) 依赖实例化访问日志的自定义配置。内置的配置包括：1）`“envoy.file_access_log”：FileAccessLog`

filter.accesslog.AccessLogFilter

filter.accesslog.AccessLogFilter proto

```
{
  "status_code_filter": "{...}",
  "duration_filter": "{...}",
  "not_health_check_filter": "{...}",
  "traceable_filter": "{...}",
  "runtime_filter": "{...}",
  "and_filter": "{...}",
  "or_filter": "{...}"
}
```

注意：必须正确设置 `status_code_filter` , `duration_filter` , `not_health_check_filter` , `traceable_filter` , `runtime_filter` , `and_filter` , `or_filter` 其中一个。

- status_code_filter

  (filter.accesslog.StatusCodeFilter) 状态码过滤器。

- duration_filter

  (filter.accesslog.DurationFilter) 时长过滤器。

- not_health_check_filter

  (filter.accesslog.NotHealthCheckFilter) 不健康检查过滤器。

- traceable_filter

  (filter.accesslog.TraceableFilter) 可追踪过滤器。

- runtime_filter

  (filter.accesslog.RuntimeFilter) 运行时过滤器。

- and_filter

  (filter.accesslog.AndFilter) 于过滤器。

- or_filter

  (filter.accesslog.OrFilter) 或过滤器。

filter.accesslog.ComparisonFilter
filter.accesslog.ComparisonFilter proto

整数比较过滤器。

```
{
  "op": "...",
  "value": "{...}"
}
```

- op

  (filter.accesslog.ComparisonFilter.Op) 比较运算符。

- value

  (RuntimeUInt32) 与之比较的值。

filter.accesslog.ComparisonFilter.Op (Enum)
filter.accesslog.ComparisonFilter.Op proto

- EQ

  (DEFAULT) 相等 `=`

- GE

  大于等于 `>=`

filter.accesslog.StatusCodeFilter
filter.accesslog.StatusCodeFilter proto

HTTP响应/状态代码的过滤器。

```
{
  "comparison": "{...}"
}
```

- comparison

(filter.accesslog.ComparisonFilter, REQUIRED) 对比。

filter.accesslog.DurationFilter
filter.accesslog.DurationFilter proto

请求持续总时间，以毫秒为单位过滤。

```
{
  "comparison": "{...}"
}
```

- comparison

  (filter.accesslog.ComparisonFilter, REQUIRED) 对比。

filter.accesslog.NotHealthCheckFilter
filter.accesslog.NotHealthCheckFilter proto

筛选不健康检查请求。由健康检查过滤器标记。

```
{}
```

filter.accesslog.TraceableFilter
filter.accesslog.TraceableFilter proto

筛选可追踪的请求。请参阅跟踪概述，以获取有关请求如何可跟踪的更多信息。

```
{}
```

filter.accesslog.RuntimeFilter
filter.accesslog.RuntimeFilter proto

过滤器用于随机抽样请求。在 `x-request-id` 头部存在的情况下采样抽取。如果存在 `x-request-id` ，则过滤器将根据运行时Key/value和从 `x-request-id` 提取值并在多个主机上持续采样。如果缺失，过滤器将根据运行时Key/value随机抽样。

```
{
  "runtime_key": "..."
}
```

- runtime_key

(string, REQUIRED) 运行时key，以获取要采样的请求的百分比。此运行时值控制在0-100范围内，默认为0。

filter.accesslog.AndFilter

filter.accesslog.AndFilter proto

对过滤器中每个过滤器的结果执行逻辑“和”运算。过滤器将按顺序进行评估，如果其中一个返回false，则过滤器立即返回false。

```
{
  "filters": []
}
```

- filters

  (filter.accesslog.AccessLogFilter, REQUIRED)

filter.accesslog.OrFilter

filter.accesslog.OrFilter proto

对每个单独的过滤器的结果执行逻辑“或”操作。过滤器将按顺序进行评估，如果其中一个返回true，则过滤器立即返回true。

```
{
  "filters": []
}
```

- filters

  (filter.accesslog.AccessLogFilter, REQUIRED)

filter.accesslog.FileAccessLog

filter.accesslog.FileAccessLog proto

将日志条目直接写入文件的 `AccessLog` 的自定义配置。内置配置为 `envoy.file_access_log AccessLog` 。

```
{
  "path": "...",
  "format": "..."
}
```

- path

  (string, REQUIRED) 要写入访问日志条目的本地文件的路径。

- format

  (string) 访问日志的格式。Envoy支持自定义访问日志格式以及默认格式。

返回

- 上一级
- 首页目录

常见故障注入类型

常见故障注入类型

filter.FaultDelay

filter.FaultDelay proto

延迟故障注入适用于在HTTP/gRPC/Mongo/Redis的操作中，提供延迟或延迟TCP连接的代理。

```
{
  "type": "...",
  "percent": "...",
  "fixed_delay": "{...}"
}
```

- type

  (filter.FaultDelay.FaultDelayType) 要使用的延迟类型（固定|指数|..）。目前只支持固定延时（`step function`）。

- percent

  (uint32) 0到100之间的整数，表示将被注入延迟的操作/连接请求的百分比。

- fixed_delay

  (Duration) 在向上游转发操作之前添加固定延迟。有关JSON/YAML持续时间映射，请参阅连接。对于HTTP/Mongo/Redis，指定的延迟将在新的请求/操作之前被注入。对于TCP连接，连接上游的代理将在指定的时间段延迟。如果type是FIXED，则这是必需的。

  注意：fixed_delay必须被设置。

filter.FaultDelay.FaultDelayType (Enum)

filter.FaultDelay.FaultDelayType proto

- FIXED

  (DEFAULT) 固定延迟（步进功能）。

返回

- 上一级
- 首页目录

FAQ

FAQ

返回

- 首页目录

Envoy有多快？

Envoy有多快？

TBD

返回

- 上一级
- 首页目录

我在哪里获得二进制文件？

我在哪里获得二进制文件？

TBD

返回

- 上一级
- 首页目录

我如何设置SNI？

我如何设置SNI？

TBD

返回

- 上一级
- 首页目录

如何设置区域感知路由？

如何设置区域感知路由？

TBD

返回

- 上一级
- 首页目录

我如何设置Zipkin跟踪？

我如何设置Zipkin跟踪？

TBD

返回

- 上一级
- 首页目录