



Compliance of  
tetrade

# Envoy 基础教程

从零开始学习 Envoy 网络代理

*The ultimate guide for Envoy Proxy beginners*

宋净超 (Jimmy Song) 编著

JIMMY SONG

# 目录

## Envoy 简介

前言	1.1
Envoy 简介概述	1.2
什么是 Envoy?	1.3

## HTTP 连接管理器 (HCM)

HTTP 连接管理器 (HCM) 概述	2.1
HTTP 连接管理器 (HCM) 介绍	2.2
HTTP 路由	2.3
请求匹配	2.4
流量分割	2.5
Header 操作	2.6
修改响应	2.7
生成请求 ID	2.8
超时	2.9
重试	2.10
请求镜像	2.11
速率限制简介	2.12
实验 1: 请求匹配	2.13
实验 2: 流量分割	2.14
实验 3: Header 操作	2.15
实验 4: 重试	2.16
实验 5: 局部速率限制	2.17
实验 6: 全局速率限制	2.18

## 集群

集群概述	3.1
服务发现	3.2
主动健康检查	3.3
异常点检测	3.4

断路器	3.5
负载均衡	3.6
实验 7：断路器	3.7

## 动态配置

动态配置概述	4.1
动态配置	4.2
来自文件系统的动态配置	4.3
来自控制平面的动态配置	4.4
实验 8：来自文件系统的动态配置	4.5

## 监听器子系统

监听器子系统概述	5.1
监听器过滤器	5.2
过滤器链匹配	5.3
HTTP 检查器监听器过滤器	5.4
原始目的地监听器过滤器	5.5
原始源监听器过滤器	5.6
代理协议监听器过滤器	5.7
TLS 检查器监听器过滤器	5.8
实验 9：原始目的地过滤器	5.9
实验 10：TLS 检查器过滤器	5.10
实验 11：匹配传输和应用协议	5.11

## 日志

日志概览	6.1
访问日志	6.2
配置访问记录器	6.3
访问日志过滤	6.4
Envoy 组件日志	6.5
实验 12：使用日志过滤器	6.6
实验 13：使用 gRPC 访问日志服务（ALS）记录日志	6.7
实验 14：将 Envoy 的日志发送到 Google Cloud Logging	6.8

## 管理接口

管理接口概览	7.1
启用管理接口	7.2
配置转储	7.3
统计	7.4
日志	7.5
集群	7.6
监听器和监听器的排空	7.7
分接式过滤器	7.8
健康检查	7.9
实验 15：使用 HTTP 分接式过滤器	7.10

## 扩展 Envoy

扩展 Envoy 概览	8.1
可扩展性概述	8.2
Lua 过滤器	8.3
WebAssembly (Wasm)	8.4
实验 16：使用 Lua 脚本扩展 Envoy	8.5
实验 17：使用 Wasm 和 Go 扩展 Envoy	8.6

# Envoy Handbook

Envoy 基础教程，本手册梳理了 Envoy 基础知识，适用于初学者，帮你快速掌握 Envoy 代理。



## 关于本书

Envoy 是一个开源的边缘和服务代理，专为云原生应用而设计。Envoy 与每个应用程序一起运行，通过提供网络相关的功能，如重试、超时、流量路由和镜像、TLS 终止等，以一种平台无关的方式抽象出网络。由于所有的网络流量都流经 Envoy 代理，因此很容易观察到流量和问题区域，调整性能，并准确定位延迟来源。

本书为 Tetratel 出品的《Envoy 基础教程》的文字内容，其配套的 8 节实验及 19 个测试，请访问 [Tetratel 学院](#)。

## 如何阅读本书

您可以使用以下方式阅读：

- [在线阅读](#) (建议)
- [下载 PDF](#)

注：PDF 非实时编译，内容可能落后于在线版本，建议在线阅读。

## 关于作者

宋净超 (Jimmy Song) , CNCF Ambassador, [云原生社区](#)创始人，个人网站 [jimmysong.io](http://jimmysong.io)。

## 许可证

您可以使用[署名 - 非商业性使用 - 相同方式共享 4.0 \(CC BY-NC-SA 4.0\)](#)协议共享。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-03-01 14:49:26

## Envoy 简介概述

在 Envoy 介绍章节中，你将了解 Envoy 的概况，并通过一个例子了解 Envoy 的构建模块。在本章结束时，你将通过运行 Envoy 的示例配置来了解 Envoy。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 15:53:20

## 什么是 Envoy?

IT 行业正在向微服务架构和云原生解决方案发展。由于使用不同的技术开发了成百上千的微服务，这些系统可能变得复杂，难以调试。

作为一个应用开发者，你考虑的是业务逻辑——购买产品或生成发票。然而，任何像这样的业务逻辑都会导致不同服务之间的多个服务调用。每个服务可能都有它的超时、重试逻辑和其他可能需要调整或微调的网络特定代码。

如果在任何时候最初的请求失败了，就很难通过多个服务来追踪，准确地指出失败发生的地方，了解请求为什么失败。是网络不可靠吗？是否需要调整重试或超时？或者是业务逻辑问题或错误？

服务可能使用不一致的跟踪和记录机制，使这种调试的复杂性增加。这些问题使你很难确定问题发生在哪，以及如何解决。如果你是一个应用程序开发人员，而调试网络问题不属于你的核心技能，那就更是如此。

将网络问题从应用程序堆栈中抽离出来，由另一个组件来处理网络部分，让调试网络问题变得更容易。这就是 Envoy 所做的事情。

在每个服务实例旁边都有一个 Envoy 实例在运行。这种类型的部署也被称为 **Sidecar 部署**。Envoy 的另一种模式是边缘代理，用于构建 API 网关。

Envoy 和应用程序形成一个原子实体，但仍然是独立的进程。应用程序处理业务逻辑，而 Envoy 则处理网络问题。

在发生故障的情况下，分离关注点可以更容易确定故障是来自应用程序还是网络。

为了帮助网络调试，Envoy 提供了以下高级功能。

## 进程外架构

Envoy 是一个独立的进程，旨在与每个应用程序一起运行——也就是我们前面提到的 Sidecar 部署模式。集中配置的 Envoy 的集合形成了一个透明的服务网格。

路由和其他网络功能的责任被推给了 Envoy。应用程序向一个虚拟地址（localhost）而不是真实地址（如公共 IP 地址或主机名）发送请求，不知道网络拓扑结构。应用程序不再承担路由的责任，因为该任务被委托给一个外部进程。

与其让应用程序管理其网络配置，不如在 Envoy 层面上独立于应用程序管理网络配置。在一个组织中，这可以使应用程序开发人员解放出来，专注于应用程序的业务逻辑。

Envoy 适用于任何编程语言。你可以用 Go、Java、C++ 或其他任何语言编写你的应用程序，而 Envoy 可以在它们之间架起桥梁。Envoy 的行为是相同的，无论应用程序的编程语言或它们运行的操作系统是什么。

Envoy 还可以在整个基础设施中透明地进行部署和升级。这与为每个单独的应用程序部署库升级相比，后者可能是非常痛苦和耗时的。

进程外架构是有益的，因为它使我们在不同的编程语言 / 应用堆栈中保持一致，我们可以免费获得独立的应用生命周期和所有的 Envoy 网络功能，而不必在每个应用中单独解决这些问题。

## L3/L4 过滤器结构

Envoy 是一个 L3/L4 网络代理，根据 IP 地址和 TCP 或 UDP 端口进行决策。它具有一个可插拔的过滤器链，可以编写你的过滤器来执行不同的 TCP/UDP 任务。

**过滤器链（Filter Chain）** 的想法借鉴了 Linux shell，即一个操作的输出被输送到另一个操作中。例如：

```
ls -l | grep "Envoy*.cc" | wc -l
```

Envoy 可以通过堆叠所需的过滤器来构建逻辑和行为，形成一个过滤器链。许多过滤器已经存在，并支持诸如原始 TCP 代理、UDP 代理、HTTP 代理、TLS 客户端认证等任务。Envoy 也是可扩展的，我们可以编写我们的过滤器。

## L7 过滤器结构

Envoy 支持一个额外的 HTTP L7 过滤器层。我们可以在 HTTP 连接管理子系统中插入 HTTP 过滤器，执行不同的任务，如缓冲、速率限制、路由 / 转发等。

## 一流的 HTTP/2 支持

Envoy 同时支持 HTTP/1.1 和 HTTP/2，并且可以作为一个透明的 HTTP/1.1 到 HTTP/2 的双向代理进行操作。这意味着任何 HTTP/1.1 和 HTTP/2 客户端和目标服务器的组合都可以被桥接起来。即使你的传统应用没有通过 HTTP/2 进行通信，如果你把它们部署在 Envoy 代理旁边，它们最终也会通过 HTTP/2 进行通信。

推荐在所有的服务间配置的 Envoy 使用 HTTP/2，以创建一个持久连接的网格，请求和响应可以在上面复用。

## HTTP 路由

当以 HTTP 模式操作并使用 REST 时，Envoy 支持路由子系统，能够根据路径、权限、内容类型和运行时间值来路由和重定向请求。在将 Envoy 作为构建 API 网关的前台 / 边缘代理时，这一功能非常有用，在构建服务网格（sidecar 部署模式）时，也可以利用这一功能。

## gRPC 准备就绪

Envoy 支持作为 gRPC 请求和响应的路由和负载均衡底层所需的所有 HTTP/2 功能。

gRPC 是一个开源的远程过程调用（RPC）系统，它使用 HTTP/2 进行传输，并将协议缓冲区作为接口描述语言（IDL），它提供的功能包括认证、双向流和流量控制、阻塞 / 非阻塞绑定，以及取消和超时。

## 服务发现和动态配置

我们可以使用静态配置文件来配置 Envoy，这些文件描述了服务间通信方式。

对于静态配置 Envoy 不现实的高级场景，Envoy 支持动态配置，在运行时自动重新加载配置。一组名为 xDS 的发现服务可以用来通过网络动态配置 Envoy，并为 Envoy 提供关于主机、集群 HTTP 路由、监听套接字和加密信息。

## 健康检查

负载均衡器有一个特点，那就是只将流量路由到健康和可用的上游服务。Envoy 支持健康检查子系统，对上游服务集群进行主动健康检查。然后，Envoy 使用服务发现和健康检查信息的组合来确定健康的负载均衡目标。Envoy 还可以通过异常点检测子系统支持被动健康检查。

## 高级负载均衡

Envoy 支持自动重试、断路、全局速率限制（使用外部速率限制服务）、影子请求（或流量镜像）、异常点检测和请求对冲。

## 前端 / 边缘代理支持

Envoy 的特点使其非常适合作为边缘代理运行。这些功能包括 TLS 终端、HTTP/1.1、HTTP/2 和 HTTP/3 支持，以及 HTTP L7 路由。

## TLS 终止

应用程序和代理的解耦使网格部署模型中所有服务之间的 TLS 终止（双向 TLS）成为可能。

## 一流的可观测性

为了便于观察，Envoy 会生成日志、指标和追踪。Envoy 目前支持 statsd（和兼容的提供者）作为所有子系统的统计。得益于可扩展性，我们也可以在需要时插入不同的统计提供商。

## HTTP/3 (Alpha)

Envoy 1.19.0 支持 HTTP/3 的上行和下行，并在 HTTP/1.1、HTTP/2 和 HTTP/3 之间进行双向转义。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 15:54:49

## HTTP 连接管理器（HCM）概述

在 HCM 章节中，我们将对 HTTP 连接管理器过滤器进行扩展。我们将学习过滤器的排序，以及 HTTP 路由和匹配如何工作。我们将向你展示如何分割流量、操作 Header 信息、配置超时、实现重试、请求镜像和速率限制。

在本章结束时，你将对 HCM 过滤器有一个很好的理解，以及如何路由和拆分 HTTP 流量，操纵 Header 等等。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 16:00:57

## HTTP 连接管理器 (HCM) 介绍

HCM 是一个网络级的过滤器，将原始字节转译成 HTTP 级别的消息和事件（例如，收到的 Header，收到的 Body 数据等）。

HCM 过滤器还处理标准的 HTTP 功能。它支持访问记录、请求 ID 生成和跟踪、Header 操作、路由表管理和统计等功能。

从协议的角度来看，HCM 原生支持 HTTP/1.1、WebSockets、HTTP/2 和 HTTP/3（仍在 Alpha 阶段）。

Envoy 代理被设计成一个 HTTP/2 复用代理，这体现在描述 Envoy 组件的术语中。

### HTTP/2 术语

在 HTTP/2 中，流是已建立的连接中的字节的双向流动。每个流可以携带一个或多个消息 (**message**)。消息是一个完整的帧 (**frame**) 序列，映射到一个 HTTP 请求或响应消息。最后，帧是 HTTP/2 中最小的通信单位。每个帧都包含一个帧头 (**frame header**)，它至少可以识别该帧所属的流。帧可以携带有关 HTTP Header、消息有效载荷等信息。

无论流来自哪个连接（HTTP/1.1、HTTP/2 或 HTTP/3），Envoy 都使用一个叫做 **编解码 API (codec PAI)** 的功能，将不同的线程协议翻译成流、请求、响应等协议无关模型。协议无关的模型意味着大多数 Envoy 代码不需要理解每个协议的具体内容。

## HTTP 过滤器

在 HCM 中，Envoy 支持一系列的 HTTP 过滤器。与监听器级别的过滤器不同，这些过滤器对 HTTP 级别的消息进行操作，而不知道底层协议（HTTP/1.1、HTTP/2 等）或复用能力。

有三种类型的 HTTP 过滤器。

- **解码器 (Decoder)**：当 HCM 对请求流的部分进行解码时调用。
- **编码器 (Encoder)**：当 HCM 对响应流的部分进行编码时调用。
- **解码器 / 编码器 (Decoder/Encoder)**：在两个路径上调用，解码和编码

下图解释了 Envoy 如何在请求和响应路径上调用不同的过滤器类型。

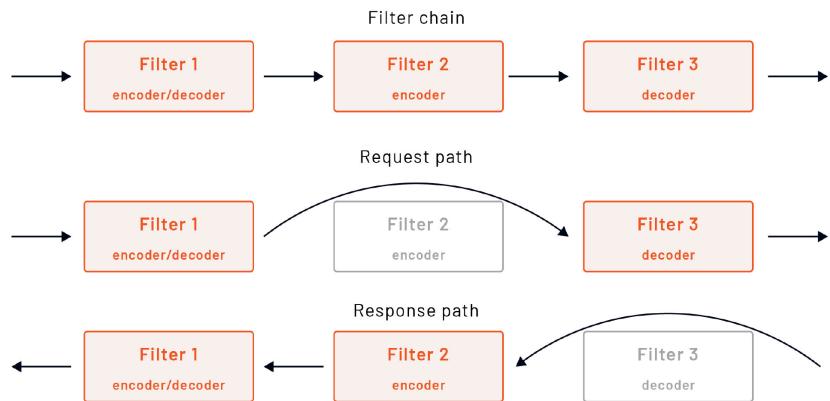


图 2.2.1：请求响应路径及 HTTP 过滤器

像网络过滤器一样，单个的 HTTP 过滤器可以停止或继续执行后续的过滤器，并在单个请求流的范围内相互分享状态。

## 数据共享

在高层次上，我们可以把过滤器之间的数据共享分成静态和动态。

静态包含 Envoy 加载配置时的任何不可变的数据集，它被分成三个部分。

### 1. 元数据

Envoy 的配置，如监听器、路由或集群，都包含一个 `metadata` 数据字段，存储键 / 值对。元数据允许我们存储特定过滤器的配置。这些值不能改变，并在所有请求 / 连接中共享。例如，元数据值在集群中使用子集选择器时被使用。

### 2. 类型化的元数据

类型化元数据不需要为每个流或请求将元数据转换为类型化的类对象，而是允许过滤器为特定的键注册一个一次性的转换逻辑。来自 xDS 的元数据在配置加载时被转换为类对象，过滤器可以在运行时请求类型化的版本，而不需要每次都转换。

### 3. HTTP 每路过滤器配置

与适用于所有虚拟主机的全局配置相比，我们还可以指定每个虚拟主机或路由的配置。每个路由的配置被嵌入到路由表中，可以在 `typed_per_filter_config` 字段下指定。

另一种分享数据的方式是使用动态状态。动态状态会在每个连接或 HTTP 流中产生，并且它可以被产生它的过滤器改变。名为 `StreamInfo` 的对象提供了一种从 map 上存储和检索类型对象的方法。

## 过滤器顺序

指定 HTTP 过滤器的顺序很重要。考虑一下下面的 HTTP 过滤器链。

```
http_filters:  
  - filter_1  
  - filter_2  
  - filter_3
```

一般来说，链中的最后一个过滤器通常是路由器过滤器。假设所有的过滤器都是解码器 / 编码器过滤器，HCM 在请求路径上调用它们的顺序是 `filter_1`、`filter_2`、`filter_3`。

在响应路径上，Envoy 只调用编码器过滤器，但顺序相反。由于这三个过滤器都是解码器 / 编码器过滤器，所以在响应路径上的顺序是 `filter_3`、`filter_2`、`filter_1`。

## 内置 HTTP 过滤器

Envoy 已经内置了几个 HTTP 过滤器，如 CORS、CSRF、健康检查、JWT 认证等。你可以[在这里](#)找到 HTTP 过滤器的完整列表。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:26:16

## HTTP 路由

前面提到的路由器过滤器（`envoy.filters.http.router`）就是实现 HTTP 转发的。路由器过滤器几乎被用于所有的 HTTP 代理方案中。路由器过滤器的主要工作是查看路由表，并对请求进行相应的路由（转发和重定向）。

路由器使用传入请求的信息（例如，`host` 或 `authority` 头），并通过虚拟主机和路由规则将其与上游集群相匹配。

所有配置的 HTTP 过滤器都使用包含路由表的路由配置（`route_config`）。尽管路由表的主要消费者将是路由器过滤器，但其他过滤器如果想根据请求的目的地做出任何决定，也可以访问它。

一组虚拟主机构成了路由配置。每个虚拟主机都有一个逻辑名称，一组可以根据请求头被路由到它的域，以及一组指定如何匹配请求并指出下一步要做什么的路由。

Envoy 还支持路由级别的优先级路由。每个优先级都有其连接池和断路设置。目前支持的两个优先级是 `DEFAULT` 和 `HIGH`。如果我们没有明确提供优先级，则默认为 `DEFAULT`。

这里有一个片段，显示了一个路由配置的例子。

```
route_config:  
  name: my_route_config # 用于统计的名称，与路由无关  
  virtual_hosts:  
    - name: bar_vhost  
      domains: ["bar.io"]  
      routes:  
        - match:  
          prefix: "/"  
          route:  
            priority: HIGH  
            cluster: bar_io  
    - name: foo_vhost  
      domains: ["foo.io"]  
      routes:  
        - match:  
          prefix: "/"  
          route:  
            cluster: foo_io  
        - match:  
          prefix: "/api"  
          route:  
            cluster: foo_io_api
```

当一个 HTTP 请求进来时，虚拟主机、域名和路由匹配依次发生。

1. `host` 或 `authority` 头被匹配到每个虚拟主机的 `domains` 字段中指定的值。例如，如果主机头被设置为 `foo.io`，则虚拟主机 `foo_vhost` 匹配。

2. 接下来会检查匹配的虚拟主机内 `routes` 下的条目。如果发现匹配，就不做进一步检查，而是选择一个集群。例如，如果我们匹配了 `foo.io` 虚拟主机，并且请求前缀是 `/api`，那么集群 `foo_io_api` 就被选中。
3. 如果提供，虚拟主机中的每个虚拟集群（`virtual_clusters`）都会被检查是否匹配。如果有匹配的，就使用一个虚拟集群，而不再进行进一步的虚拟集群检查。

虚拟集群是一种指定针对特定端点的重组词匹配规则的方式，并明确为匹配的请求生成统计信息。

虚拟主机的顺序以及每个主机内的路由都很重要。考虑下面的路由配置。

```
route_config:  
  virtual_hosts:  
    - name: hello_vhost  
      domains: ["hello.io"]  
      routes:  
        - match:  
          prefix: "/api"  
          route:  
            cluster: hello_io_api  
        - match:  
          prefix: "/api/v1"  
          route:  
            cluster: hello_io_api_v1
```

如果我们发送以下请求，哪个路由 / 集群被选中？

```
curl hello.io/api/v1
```

第一个设置集群 `hello_io_api` 的 路由被匹配。这是因为匹配是按照前缀的顺序进行评估的。然而，我们可能错误地期望前缀为 `/api/v1` 的路由被匹配。为了解决这个问题，我们可以调换路由的顺序，或者使用不同的匹配规则。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 16:05:53

## 请求匹配

本节将为你介绍 HTTP 连接管理器中的请求匹配。

### 路径匹配

我们只谈了一个使用 `前缀` 字段匹配前缀的匹配规则。下面的表格解释了其他支持的匹配规则。

规则名称	描述
<code>prefix</code>	前缀必须与 <code>path</code> 标头的开头相匹配。例如，前缀 <code>/api</code> 将匹配路径 <code>/api</code> 和 <code>/api/v1</code> ，而不是 <code>/</code> 。
<code>path</code>	路径必须与确切的 <code>path</code> 标头相匹配（没有查询字符串）。例如，路径 <code>/api</code> 将匹配路径 <code>/api</code> ，但不匹配 <code>/api/v1</code> 或 <code>/</code> 。
<code>safe_regex</code>	路径必须符合指定的正则表达式。例如，正则表达式 <code>^/products/\d+\$</code> 将匹配路径 <code>/products/123</code> 或 <code>/products/321</code> ，但不是 <code>/products/hello</code> 或 <code>/api/products/123</code> 。
<code>connect_matcher</code>	匹配器只匹配 CONNECT 请求（目前在 Alpha 中）。

默认情况下，前缀和路径匹配是大小写敏感的。要使其不区分大小写，我们可以将 `case_sensitive` 设置为 `false`。注意，这个设置不适用于 `safe_regex` 匹配。

### Header 匹配

另一种匹配请求的方法是指定一组 Header。路由器根据路由配置中所有指定的 Header 检查请求 Header。如果所有指定的头信息都存在于请求中，并且设置了相同的值，则进行匹配。

多个匹配规则可以应用于Header。

#### 范围匹配

`range_match` 检查请求 Header 的值是否在指定的以十进制为单位的整数范围内。该值可以包括一个可选的加号或减号，后面是数字。

为了使用范围匹配，我们指定范围的开始和结束。起始值是包含的，而终止值是不包含的（`[start, end]`）。

```
- match:
  prefix: "/"
  headers:
    - name: minor_version
      range_match:
        start: 1
        end: 11
```

上述范围匹配将匹配 `minor_version` 头的值，如果它被设置为 1 到 10 之间的任何数字。

### 存在匹配

`present_match` 检查传入的请求中是否存在一个特定的头。

```
- match:
  prefix: "/"
  headers:
    - name: debug
      present_match: true
```

如果我们设置了 `debug` 头，无论头的值是多少，上面的片段都会评估为 `true`。如果我们把 `present_match` 的值设为 `false`，我们就可以检查是否有 Header。

### 字符串匹配

`string_match` 允许我们通过前缀或后缀，使用正则表达式或检查该值是否包含一个特定的字符串，来准确匹配头的值。

```

- match:
  prefix: "/"
  headers:
    # 头部`regex_match`匹配所提供的正则表达式
    - name: regex_match
      string_match:
        safe_regex_match:
          google_re2: {}
          regex: "^v\\d+$"
    # Header `exact_match` 包含值`hello`。
    - name: exact_match
      string_match:
        exact:"hello"
    # 头部`prefix_match`以`api`开头。
    - name: prefix_match
      string_match:
        prefix:"api"
    # 头部`suffix_match`以`_1`结束
    - name: suffix_match
      string_match:
        suffix: "_1"
    # 头部`contains_match`包含值 "debug"
    - name: contains_match
      string_match:
        contains: "debug"

```

## 反转匹配

如果我们设置了 `invert_match`，匹配结果就会反转。

```

- match:
  prefix: "/"
  headers:
    - name: version
      range_match:
        start: 1
        end: 6
      invert_match: true

```

上面的片段将检查 `version` 头中的值是否在 1 和 5 之间；然而，由于我们添加了 `invert_match` 字段，它反转了结果，检查头中的值是否超出了这个范围。

`invert_match` 可以被其他匹配器使用。例如：

```

- match:
  prefix: "/"
  headers:
    - name: env
      contains_match: "test"
      invert_match: true

```

上面的片段将检查 `env` 头的值是否包含字符串 `test`。如果我们设置了 `env` 头，并且它不包括字符串 `test`，那么整个匹配的评估结果为真。

## 查询参数匹配

使用 `query_parameters` 字段，我们可以指定路由应该匹配的 URL 查询的参数。过滤器将检查来自 `path` 头的查询字符串，并将其与所提供的参数进行比较。

如果有一个以上的查询参数被指定，它们必须与规则相匹配，才能评估为真。

请考虑以下例子。

```
- match:
  prefix: "/"
  query_parameters:
  - name: env
    present_match: true
```

如果有一个名为 `env` 的查询参数被设置，上面的片段将评估为真。它没有说任何关于该值的事情。它只是检查它是否存在。例如，使用上述匹配器，下面的请求将被评估为真。

```
GET /hello?env=test
```

我们还可以使用字符串匹配器来检查查询参数的值。下表列出了字符串匹配的不同规则。

规则名称	描述
<code>exact</code>	必须与查询参数的精确值相匹配。
<code>prefix</code>	前缀必须符合查询参数值的开头。
<code>suffix</code>	后缀必须符合查询参数值的结尾。
<code>safe_regex</code>	查询参数值必须符合指定的正则表达式。
<code>contains</code>	检查查询参数值是否包含一个特定的字符串。

除了上述规则外，我们还可以使用 `ignore_case` 字段来指示精确、前缀或后缀匹配是否应该区分大小写。如果设置为 "true"，匹配就不区分大小写。

下面是另一个使用前缀规则进行不区分大小写的查询参数匹配的例子。

```
- match:
  prefix: "/"
  query_parameters:
  - name: env
    string_match:
      prefix: "env_"
      ignore_case: true
```

如果有一个名为 `env` 的查询参数，其值以 `env_` 开头，则上述内容将评估为真。例如，`env_staging` 和 `ENV_prod` 评估为真。

## gRPC 和 TLS 匹配器

我们可以在路上配置另外两个匹配器：gRPC 路由匹配器（`grpc`）和 TLS 上下文匹配器（`tls_context`）。

gRPC 匹配器将只在 gRPC 请求上匹配。路由器检查内容类型头的 `application/grpc` 和其他 `application/grpc+` 值，以确定该请求是否是 gRPC 请求。

例如：

```
- match:  
  prefix: "/"  
  grpc: {}
```

注意 gRPC 匹配器没有任何选项。

如果请求是 gRPC 请求，上面的片段将匹配路由。

同样，如果指定了 TLS 匹配器，它将根据提供的选项来匹配 TLS 上下文。在 `tls_context` 字段中，我们可以定义两个布尔值——`presented` 和 `validated`。`presented` 字段检查证书是否被出示。`validated` 字段检查证书是否被验证。

例如：

```
- match:  
  prefix: "/"  
  tls_context:  
    presented: true  
    validated: true
```

如果一个证书既被出示又被验证，上述匹配评估为真。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 16:07:24

## 流量分割

Envoy 支持在同一虚拟主机内将流量分割到不同的路由。我们可以在两个或多个上游集群之间分割流量。

有两种不同的方法。第一种是使用运行时对象中指定的百分比，第二种是使用加权集群。

## 使用运行时的百分比进行流量分割

使用运行时对象的百分比很适合于金丝雀发布或渐进式交付的场景。在这种情况下，我们想把流量从一个上游集群逐渐转移到另一个。

实现这一目标的方法是提供一个 `runtime_fraction` 配置。让我们用一个例子来解释使用运行时百分比的流量分割是如何进行的。

```
route_config:
  virtual_hosts:
    - name: hello_vhost
      domains: ["hello.io"]
      routes:
        - match:
            prefix: "/"
            runtime_fraction:
              default_value:
                numerator: 90
                denominator: HUNDRED
            route:
              cluster: hello_v1
        - match:
            prefix: "/"
            route:
              cluster: hello_v2
```

上述配置声明了两个版本的 hello 服务：`hello_v1` 和 `hello_v2`。

在第一个匹配中，我们通过指定分子（`90`）和分母（`HUNDRED`）来配置 `runtime_fraction` 字段。Envoy 使用分子和分母来计算最终的分数值。在这种情况下，最终值是  $90\%$  ( $90/100 = 0.9 = 90\%$ )。

Envoy 在  $[0, \text{分母}]$  范围内生成一个随机数（例如，在我们的案例中是  $[0, 100]$ ）。如果随机数小于分子值，路由器就会匹配该路由，并将流量发送到我们案例中的集群 `hello_v1`。

如果随机数大于分子值，Envoy 继续评估其余的匹配条件。由于我们有第二条路由的精确前缀匹配，所以它是匹配的，Envoy 会将流量发送到集群 `hello_v2`。一旦我们把分子值设为 0，所有随机数会大于分子值。因此，所有流量都会流向第二条路由。

我们也可以在运行时键中设置分子值。例如：

```

route_config:
  virtual_hosts:
    - name: hello_vhost
      domains: ["hello.io"]
      routes:
        - match:
            prefix: "/"
            runtime_fraction:
              default_value:
                numerator: 0
                denominator: HUNDRED
              runtime_key: routing.hello_io
          route:
            cluster: hello_v1
        - match:
            prefix: "/"
          route:
            cluster: hello_v2
...
layered_runtime:
  layers:
    - name: static_layer
      static_layer:
        routing.hello_io: 90

```

在这个例子中，我们指定了一个名为 `routing.hello_io` 的运行时键。我们可以在配置中的分层运行时字段下设置该键的值——这也可以从文件或通过运行时发现服务（RTDS）动态读取和更新。为了简单起见，我们在配置文件中直接设置。

当 Envoy 这次进行匹配时，它将看到提供了 `runtime_key`，并将使用该值而不是分子值。有了运行时键，我们就不必在配置中硬编码这个值了，我们可以让 Envoy 从一个单独的文件或 RTDS 中读取它。

当你有两个集群时，使用运行时百分比的方法效果很好。但是，当你想把流量分到两个以上的集群，或者你正在运行 A/B 测试或多变量测试方案时，它就会变得复杂。

## 使用加权集群进行流量分割

当你在两个或多个版本的服务之间分割流量时，加权集群的方法是理想的。在这种方法中，我们为多个上游集群分配了不同的权重。而带运行时百分比的方法使用了许多路由，我们只需要为加权集群提供一条路由。

我们将在下一个模块中进一步讨论上游集群。为了解释用加权集群进行的流量分割，我们可以把上游集群看成是流量可以被发送到的终端的集合。

我们在路由内指定多个加权集群（`weighted_clusters`），而不是设置一个集群（`cluster`）。

继续前面的例子，我们可以这样重写配置，以代替使用加权集群。

```

route_config:
  virtual_hosts:
    - name: hello_vhost
      domains: ["hello.io"]
      routes:
        - match:
          prefix: "/"
          route:
            weighted_clusters:
              clusters:
                - name: hello_v1
                  weight: 90
                - name: hello_v2
                  weight: 10

```

在加权的集群下，我们也可以设置 `runtime_key_prefix`，它将从运行时密钥配置中读取权重。注意，如果运行时密钥配置不在那里，Envoy 会使用每个集群旁边的权重。

```

route_config:
  virtual_hosts:
    - name: hello_vhost
      domains: ["hello.io"]
      routes:
        - match:
          prefix: "/"
          route:
            weighted_clusters:
              runtime_key_prefix: routing.hello_io
              clusters:
                - name: hello_v1
                  weight: 90
                - name: hello_v2
                  weight: 10
...
layered_runtime:
  layers:
    - name: static_layer
      static_layer:
        routing.hello_io.hello_v1: 90
        routing.hello_io.hello_v2: 10

```

权重代表 Envoy 发送给上游集群的流量的百分比。所有权重的总和必须是 100。然而，使用 `total_weight` 字段，我们可以控制所有权重之和必须等于的值。例如，下面的片段将 `total_weight` 设置为 15。

```
route_config:  
  virtual_hosts:  
    - name: hello_vhost  
      domains: ["hello.io"]  
      routes:  
        - match:  
          prefix: "/"  
        route:  
          weighted_clusters:  
            runtime_key_prefix: routing.hello_io  
            total_weight: 15  
            clusters:  
              - name: hello_v1  
                weight: 5  
              - name: hello_v2  
                weight: 5  
              - name: hello_v3  
                weight: 5
```

为了动态地控制权重，我们可以设置 `runtime_key_prefix`。路由器使用运行时密钥前缀值来构建与每个集群相关的运行时密钥。如果我们提供了运行时密钥前缀，路由器将检查 `runtime_key_prefix + "." + cluster_name` 的值，其中 `cluster_name` 表示集群数组中的条目（例如 `hello_v1`、`hello_v2`）。如果 Envoy 没有找到运行时密钥，它将使用配置中指定的值作为默认值。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 16:08:15

## Header 操作

HCM 支持在加权集群、路由、虚拟主机和 / 或全局配置层面操纵请求和响应头。

注意，我们不能直接从配置中修改所有的 Header，使用 Wasm 扩展的情况除外。然后，我们可以修改 `:authority header`，例如下面的情况。

不可变的头是伪头（前缀为 `:`，如 `:scheme`）和 `host` 头。此外，诸如 `:path` 和 `:authority` 这样的头信息可以通过 `prefix_rewrite`、`regex_rewrite` 和 `host_rewrite` 配置来间接修改。

Envoy 按照以下顺序对请求 / 响应应用这些头信息：

1. 加权的集群级头信息
2. 路由级 Header
3. 虚拟主机级 Header
4. 全局级 Header

这个顺序意味着 Envoy 可能会用更高层次（路由、虚拟主机或全局）配置的头来覆盖加权集群层次上设置的 Header。

在每一级，我们可以设置以下字段来添加 / 删除请求 / 响应头。

- `response_headers_to_add`：要添加到响应中的 Header 信息数组。
- `response_headers_to_remove`：要从响应中移除的 Header 信息数组。
- `request_headers_to_add`：要添加到请求中的 Header 信息数组。
- `request_headers_to_remove`：要从请求中删除的 Header 信息数组。

除了硬编码标头值之外，我们还可以使用变量来为标头添加动态值。变量名称以百分数符号（%）为分隔符。支持的变量名称包括

`%DOWNSTREAM_REMOTE_ADDRESS%`、`%UPSTREAM_REMOTE_ADDRESS%`、`%START_TIME%`、`%RESPONSE_FLAGS%` 和更多。你可以在[这里](#)找到完整的变量列表。

让我们看一个例子，它显示了如何在不同级别的请求 / 响应中添加 / 删除头信息。

```

route_config:
  response_headers_to_add:
    - header:
        key: "header_1"
        value: "some_value"
      # 如果为真（默认），它会将该值附加到现有值上。
      # 否则它将替换现有的值
      append: false
  response_headers_to_remove: "header_we_dont_need"
  virtual_hosts:
    - name: hello_vhost
      request_headers_to_add:
        - header:
            key: "v_host_header"
            value: "from_v_host"
      domains: ["hello.io"]
      routes:
        - match:
            prefix: "/"
          route:
            cluster: hello
          response_headers_to_add:
            - header:
                key: "route_header"
                value: "%DOWNSTREAM_REMOTE_ADDRESS%"
        - match:
            prefix: "/api"
          route:
            cluster: hello_api
          response_headers_to_add:
            - header:
                key: "api_route_header"
                value: "api-value"
            - header:
                key: "header_1"
                value: "this_will_be_overwritten"

```

## 标准 Header

Envoy 在收到请求（解码）和向上游集群发送请求（编码）时，会操作一组头信息。

当使用裸露的 Envoy 配置将流量路由到单个集群时，在编码过程中会设置以下头信息。

```

':authority', 'localhost:10000'
':path', '/'
':method', 'GET'
':scheme', 'http'
'user-agent', 'curl/7.64.0'
'accept', '*/*'
'x-forwarded-proto', 'http'
'x-request-id', '14f0ac76-128d-4954-ad76-823c3544197e'
'x-envoy-expected-rq-timeout-ms', '15000'

```

在编码（响应）时，会发送一组不同的头信息。

```

':status', '200'
'x-powered-by', 'Express'
'content-type', 'text/html; charset=utf-8'
'content-length', '563'
'etag', 'W/"233-b+4UpNDb0tHFiEpLMsDEDK7iTTeI"'
'date', 'Fri, 16 Jul 2021 21:59:52 GMT'
'x-envoy-upstream-service-time', '2'
'server', 'envoy'

```

下表解释了 Envoy 在解码或编码过程中设置的不同头信息。

Header	描述
:scheme	设置并提供给过滤器，并转发到上游。(对于 HTTP/1, :scheme 头是由绝对 URL 或 x-forwarded-proto 头值设置的)。
user-agent	通常由客户端设置，但在启用 add_user_agent 时可以修改(仅当 Header 尚未设置时)。该值由 --service-cluster 命令行选项决定。
x-forwarded-proto	标准头，用于识别客户端用于连接到代理的协议。该值为 http 或 https。
x-request-id	Envoy 用来唯一地识别一个请求，也用于访问记录和跟踪。
x-envoy-expected-rq-timeout-ms	指定路由器期望请求完成的时间，单位是毫秒。这是从 x-envoy-upstream-rq-timeout-ms 头值中读取的(假设设置了 respect_expected_rq_timeout )或从路由超时设置中读取(默认为 15 秒)。
x-envoy-upstream-service-time	端点处理请求所花费的时间，以毫秒为单位，以及 Envoy 和上游主机之间的网络延迟。
server	设置为 server_name 字段中指定的值(默认为 envoy)。

根据不同的场景，Envoy 会设置或消费一系列其他头信息。当我们在课程的其余部分讨论这些场景和功能时，我们会引出不同的头信息。

## Header 清理

Header 清理是一个出于安全原因添加、删除或修改请求 Header 的过程。有一些头信息，Envoy 有可能会进行清理。

Header	描述
x-envoy-decorator-operation	覆盖由追踪机制产生的任何本地定义的跨度名称。
x-envoy-downstream-service-cluster	包含调用者的服务集群（对于外部请求则删除）。由 <code>-service-cluster</code> 命令行选项决定，要求 <code>user_agent</code> 设置为 <code>true</code> 。
x-envoy-downstream-service-node	和前面的头一样，数值由 <code>--service--node</code> 选项决定。
x-envoy-expected-rq-timeout-ms	指定路由器期望请求完成的时间，单位是毫秒。这是从 <code>x-envoy-upstream-rq-timeout-ms</code> 头值中读取的（假设设置了 <code>respect_expected_rq_timeout</code> ）或从路由超时设置中读取（默认为 15 秒）。
x-envoy-external-address	受信任的客户端地址（关于如何确定，详见下面的 XFF）。
x-envoy-force-trace	强制收集的追踪。
x-envoy-internal	如果请求是内部的，则设置为 "true"（关于如何确定的细节，见下面的 XFF）。
x-envoy-ip-tags	如果外部地址在 IP 标签中被定义，由 HTTP IP 标签过滤器设置。
x-envoy-max-retries	如果配置了重试策略，重试的最大次数。
x-envoy-retry-grpc-on	对特定 gRPC 状态代码的失败请求进行重试。
x-envoy-retry-on	指定重试策略。
x-envoy-upstream-alt-stat-name	Emist 上游响应代码 / 时间统计到一个双统计树。
x-envoy-upstream-rq-per-try-timeout-ms	设置路由请求的每次尝试超时。
x-envoy-upstream-rq-timeout-alt-response	如果存在，在请求超时的情况下设置一个 204 响应代码（而不是 504）。

Header	描述
<code>x-envoy-upstream-rq-timeout-ms</code>	覆盖路由配置超时。
<code>x-forwarded-client-certif</code>	表示一个请求流经的所有客户端 / 代理中的部分证书信息。
<code>x-forwarded-for</code>	表示 IP 地址请求通过了。更多细节见下面的 XFF。
<code>x-forwarded-proto</code>	设置来源协议 ( <code>http</code> 或 <code>https</code> )。
<code>x-request-id</code>	Envoy 用来唯一地识别一个请求。也用于访问日志和追踪。

是否对某个特定的头进行清理，取决于请求来自哪里。Envoy 通过查看 `x-forwarded-for` 头 (XFF) 和 `internal_address_config` 设置来确定请求是外部还是内部。

## XFF

XFF 或 `x-forwarded-for` 头表示请求在从客户端到服务器的途中所经过的 IP 地址。下游和上游服务之间的代理在代理请求之前将最近的客户的 IP 地址附加到 XFF 列表中。

Envoy 不会自动将 IP 地址附加到 XFF 中。只有当 `use_remote_address` (默认为 `false`) 被设置为 `true`，并且 `skip_xff_append` 被设置为 `false` 时，Envoy 才会追加该地址。

当 `use_remote_address` 被设置为 `true` 时，HCM 在确定来源是内部还是外部以及修改头信息时，会使用客户端连接的真实远程地址。这个值控制 Envoy 如何确定可信的客户端地址。

### 可信的客户端地址

可信的客户端地址是已知的第一个准确的源 IP 地址。向 Envoy 代理发出请求的下游节点的源 IP 地址被认为是正确的。

请注意，完整的 XFF 有时不能被信任，因为恶意的代理可以伪造它。然而，如果一个受信任的代理将最后一个地址放在 XFF 中，那么它就可以被信任。例如，如果我们看一下请求路径 `IP1 -> IP2 -> IP3 -> Envoy`，`IP3` 是 Envoy 会认为信任的节点。

Envoy 支持通过 `original_ip_detection_extensions` 字段设置的扩展，以帮助确定原始 IP 地址。目前，有两个扩展：`custom_header` 和 `xff`。

通过自定义头的扩展，我们可以提供一个包含原始下游远程地址的头名称。此外，我们还可以告诉 HCM 将检测到的地址视为可信地址。

通过 `xff` 扩展，我们可以指定从 `x-forwarded-for` 头的右侧开始的额外代理跳数来信任。如果我们将这个值设置为 `1` 还使用上面的例子，受信任的地址将是 `IP2` 和 `IP3`。

Envoy 使用可信的客户端地址来确定请求是内部还是外部。如果我们把 `use_remote_address` 设置为 `true`，那么如果请求不包含 XFF，并且直接下游节点与 Envoy 的连接有一个内部源地址，那么就认为是内部请求。Envoy 使用 [RFC1918](#) 或 [RFC4193](#) 来确定内部源地址。

如果我们把 `use_remote_address` 设置为 `false`（默认值），只有当 XFF 包含上述两个 RFC 定义的单一内部源地址时，请求才是内部的。

让我们看一个简单的例子，把 `use_remote_address` 设为 `true`，`skip_xff_append` 设为 `false`。

```
...
- filters:
  - name: envoy.filters.network.http_connection_manager
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.network
      use_remote_address: true
      skip_xff_append: false
...
```

如果我们从同一台机器向代理发送一个请求（即内部请求），发送到上游的头信息将是这样的。

```
':authority', 'localhost:10000'
':path', '/'
':method', 'GET'
':scheme', 'http'
'user-agent', 'curl/7.64.0'
'accept', '*/*'
'x-forwarded-for', '10.128.0.17'
'x-forwarded-proto', 'http'
'x-envoy-internal', 'true'
'x-request-id', '74513723-9bbd-4959-965a-861e2162555b'
'x-envoy-expected-rq-timeout-ms', '15000'
```

这些 Header 中的大部分与我们在标准 Header 例子中看到的相同。然而，增加了两个头——`x-forwarded-for` 和 `x-envoy-internal`。`x-forwarded-for` 将包含内部 IP 地址，而 `x-envoy-internal` 头将被设

置，因为我们用 XFF 来确定地址。我们不是通过解析 `x-forwarded-for` 头来确定请求是否是内部的，而是检查 `x-envoy-internal` 头的存在，以快速确定请求是内部还是外部的。

如果我们从该网络之外发送一个请求，即客户端和 Envoy 不在同一个节点上，以下头信息会被发送到 Envoy。

```
'authority', '35.224.50.133:10000'  
'path', '/'  
'method', 'GET'  
'scheme', 'http'  
'user-agent', 'curl/7.64.1'  
'accept', '*/*'  
'x-forwarded-for', '50.35.69.235'  
'x-forwarded-proto', 'http'  
'x-envoy-external-address', '50.35.69.235'  
'x-request-id', 'dc93fd48-1233-4220-9146-eac52435cdf2'  
'x-envoy-expected-rq-timeout-ms', '15000'
```

注意 `:authority` 的值是一个实际的 IP 地址，而不是 `localhost`。同样地，`x-forwarded-for` 头包含了被调用的 IP 地址。没有 `x-envoy-internal` 头，因为这个请求是外部的。然而，我们确实得到了一个新的头，叫做 `x-envoy-external-address`。Envoy 只为外部请求设置这个头。这个头可以在内部服务之间转发，并用于基于源客户端 IP 地址的分析。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 16:09:26

## 修改响应

HCM 支持修改和定制由 Envoy 返回的响应。请注意，这对上游返回的响应不起作用。

本地回复是由 Envoy 生成的响应。本地回复的工作原理是定义一组映射器（**mapper**），允许过滤和改变响应。例如，如果没有定义任何路由或上游集群，Envoy 会发送一个本地 HTTP 404。

每个映射器必须定义一个过滤器，将请求属性与指定值进行比较（例如，比较状态代码是否等于 403）。我们可以选择从多个过滤器来匹配状态代码、持续时间、Header、响应标志等。

除了过滤器字段，映射器还有新的状态代码（`status_code`）、正文（`body` 和 `body_format_override`）和 Header（`headers_to_add`）字段。例如，我们可以有一个匹配请求状态代码 403 的过滤器，然后将状态代码改为 500，更新正文，或添加 Header。

下面是一个将 HTTP 503 响应改写为 HTTP 401 的例子。注意，这指的是 Envoy 返回的状态代码。例如，如果上游不存在，Envoy 将返回一个 503。

```

...
- name: envoy.filters.network.http_connection_manager
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.filters.network.local_reply_config:
      mappers:
        - filter:
            status_code_filter:
              comparison:
                op: EQ
                value:
                  default_value: 503
                  runtime_key: some_key
            headers_to_add:
              - header:
                  key: "service"
                  value: "unavailable"
                  append: false
            status_code: 401
            body:
              inline_string: "Not allowed"

```

注意 `runtime_key` 字段是必须的。如果 Envoy 找不到运行时密钥，它就会返回到 `default_value`。

## 生成请求 ID

唯一的请求 ID 对于通过多个服务追踪请求、可视化请求流和精确定位延迟来源至关重要。

我们可以通过 `request_id_extension` 字段配置请求 ID 的生成方式。如果我们不提供任何配置，Envoy 会使用默认的扩展，称为 `UuidRequestIdConfig`。

默认扩展会生成一个唯一的标识符 (`UUID4`) 并填充到 `x-request-id` HTTP 头中。Envoy 使用 UUID 的第 14 个位点来确定跟踪的情况。

如果第 14 个比特位 (nibble) 被设置为 `9`，则应该进行追踪采样。如果设置为 `a`，应该是由于服务器端的覆盖 (`a`) 而强制追踪，如果设置为 `b`，应该是由客户端的请求 ID 加入而强制追踪。

之所以选择第 14 个位点，是因为它在设计上被固定为 `4`。因此，`4` 表示一个默认的 UUID 没有跟踪状态，例如 `7b674932-635d-4ceb-b907-12674f8c7267` (说明：第 14 比特位实际为第 13 个数字)。

我们在 `UuidRequestIdconfig` 中的两个配置选项是 `pack_trace_reason` 和 `use_request_id_for_trace_sampling`。

```
...
...
  route_config:
    name: local_route
  request_id_extension:
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.request_id.u
      pack_trace_reason: false
      use_request_id_for_trace_sampling: false
  http_filters:
    - name: envoy.filters.http.router
...
...
```

`pack_trace_reaseon` 是一个布尔值，控制实现是否改变 UUID 以包含上述的跟踪采样决定。默认值是 `true`。`use_request_id_for_trace_sampling` 设置是否使用 `x-request-id` 进行采样。默认值也是 `true`。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 16:11:04

## 超时

Envoy 支持许多可配置的超时，这取决于你使用代理的场景。

我们将在 HCM 部分看一下不同的可配置超时。请注意，其他过滤器和组件也有各自的超时时间，我们在此不做介绍。

在配置的较高层次上设置的一些超时——例如在 HCM 层次，可以覆盖较低层次上的配置，例如 HTTP 路由层次。

最著名的超时可能是请求超时。请求超时 (`request_timeout`) 指定了 Envoy 等待接收整个请求的时间（例如 `120s`）。当请求被启动时，该计时器被激活。当最后一个请求字节被发送到上游时，或者当响应被启动时，定时器将被停用。默认情况下，如果没有提供或设置为 0，则超时被禁用。

类似的超时称为 `idle_timeout`，表示如果没有活动流，下游或上游连接何时被终止。默认的空闲超时被设置为 1 小时。空闲超时可以在 HCM 配置的 `common_http_protocol_options` 中设置，如下所示。

```
...
filters:
- name: envoy.filters.network.http_connection_manager
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.filters.network.common_http_protocol_options:
      # 设置空闲超时为 10 分钟
      idle_timeout: 600s
...
...
```

为了配置上游连接的空闲超时，我们可以使用相同的字段 `common_http_protocol_options`，但在集群部分。

还有一个与 Header 有关的超时，叫做 `request_headers_timeout`。这个超时规定了 Envoy 等待接收请求头信息的时间（例如 `5s`）。该计时器在收到头信息的第一个字节时被激活。当收到头信息的最后一个字节时，该时间就会被停用。默认情况下，如果没有提供或设置为 0，则超时被禁用。

其他一些超时也可以设置，比如

```
  stream_idle_timeout、drain_timeout 和
  delayed_close_timeout。
```

接下来就是路由超时。如前所述，路由层面的超时可以覆盖 HCM 的超时和一些额外的超时。

路由 `timeout` 是指 Envoy 等待上游做出完整响应的时间。一旦收到整个下游请求，该计时器就开始计时。超时的默认值是 15 秒；但是，它与永不结束的响应（即流媒体）不兼容。在这种情况下，需要禁用超时，而应该使用 `stream_idle_timeout`。

我们可以使用 `idle_timeout` 字段来覆盖 HCM 层面上的 `stream_idle_timeout`。

我们还可以提到 `per_try_timeout` 设置。这个超时是与重试有关的，它为每次尝试指定一个超时。通常情况下，个别尝试应该使用比 `timeout` 域设置的值更短的超时。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 16:12:08

## 重试

我们可以在虚拟主机和路由层面定义重试策略。在虚拟主机级别设置的重试策略将适用于该虚拟主机的所有路由。如果在路由级别上定义了重试策略，它将优先于虚拟主机策略，并被单独处理——即路由级别的重试策略不会继承虚拟主机级别的重试策略的值。即使 Envoy 将重试策略独立处理，配置也是一样的。

除了在配置中设置重试策略外，我们还可以通过请求头（即 `x-envoy-retry-on` 头）进行配置。

在 Envoy 配置中，我们可以配置以下内容。

1. 最大重试次数：Envoy 将重试请求，重试次数最多为配置的最大值。  
指数退避算法是用于确定重试间隔的默认算法。另一种确定重试间隔的方法是通过 Header（例如 `x-envoy-upstream-rq-per-try-timeout-ms`）。所有重试也包含在整个请求超时中，即 `request_timeout` 配置设置。默认情况下，Envoy 将重试次数设置为一次。
2. 重试条件：我们可以根据不同的条件重试请求。例如，我们只能重试 `5xx` 响应代码，网关失败，`4xx` 响应代码，等等。
3. 重试预算：重试预算规定了与活动请求数有关的并发请求的限制。这可以帮助防止过大的重试流量。
4. 主机选择重试插件：重试期间的主机选择通常遵循与原始请求相同的过程。使用重试插件，我们可以改变这种行为，指定一个主机或优先级谓词，拒绝一个特定的主机，并导致重新尝试选择主机。

让我们看看几个关于如何定义重试策略的配置例子。我们使用 httpbin 并匹配返回 `500` 响应代码的 `/status/500` 路径。

```
route_config:
  name: 5xx_route
  virtual_hosts:
    - name: httpbin
      domains: [ "*" ]
      routes:
        - match:
            path: /status/500
          route:
            cluster: httpbin
            retry_policy:
              retry_on: "5xx"
              num_retries: 5
```

在 `retry_policy` 字段中，我们将重试条件（`retry_on`）设置为 `500`，这意味着我们只想在上游返回 HTTP `500` 的情况下重试（将会如此）。Envoy 将重试该请求五次。这可以通过 `num_retries` 字段进行配置。

如果我们运行 Envoy 并发送一个请求，该请求将失败（HTTP 500），并将创建以下日志条目：

```
[2021-07-26T18:43:29.515Z] "GET /status/500 HTTP/1.1" 500 URX 0
```

注意到 `500URX` 部分告诉我们，上游响应为 500，`URX` 响应标志意味着 Envoy 拒绝了该请求，因为达到了上游重试限制。

重试条件可以设置为一个或多个值，用逗号分隔，如下表所示。

重试条件 ( <code>retry_on</code> )	描述
<code>5xx</code>	在 <code>5xx</code> 响应代码或上游不响应时重试（包括 <code>connect-failure</code> 和 <code>refused-stream</code> ）。
<code>gatewayerror</code>	对 <code>502</code> 、 <code>503</code> 或响应 <code>504</code> 代码进行重试。
<code>reset</code>	如果上游根本没有回应，则重试。
<code>connect-failure</code>	如果由于与上游服务器的连接失败（例如，连接超时）而导致请求失败，则重试。
<code>envoy-ratelimited</code>	如果存在 <code>x-envoy-ratelimited</code> 头，则重试。
<code>retriable-4xx</code>	如果上游响应的是可收回的 <code>4xx</code> 响应代码（目前只有 <code>HTTP 409</code> ），则重试。
<code>refused-stream</code>	如果上游以 <code>REFUSED_STREAM</code> 错误代码重置流，则重试。
<code>retriable-status-codes</code>	如果上游响应的任何响应代码与 <code>x-envoy-retriable-status-codes</code> 头中定义的代码相匹配（例如，以逗号分隔的整数列表，例如 <code>"502,409"</code> ），则重试。
<code>retriable-header</code>	如果上游响应包括任何在 <code>x-envoy-retriable-header-names</code> 头中匹配的头信息，则重试。

除了控制 Envoy 重试请求的响应外，我们还可以配置重试时的主机选择逻辑。我们可以指定 Envoy 在选择重试的主机时使用的 `retry_host_predicate`。

我们可以跟踪之前尝试过的主机

(`envoy.retry_host_predicates.previous_host`)，如果它们已经被尝试过，就拒绝它们。或者，我们可以使用

`envoy.retry_host_predicates.canary_hosts` 拒绝任何标记为 canary 的主机（例如，任何标记为 `canary: true` 的主机）。

例如，这里是如何配置 `previous_hosts` 插件，以拒绝任何以前尝试过的主机，并重试最多 5 次的主机选择。

```
route_config:
  name: 5xx_route
  virtual_hosts:
  - name: httpbin
    domains: [ "*"]
    routes:
    - match:
        path: /status/500
      route:
        cluster: httpbin
        retry_policy:
          retry_host_predicate:
          - name: envoy.retry_host_predicates.previous_hosts
            host_selection_retry_max_attempts: 5
```

在集群中定义了多个端点，我们会看到每次重试都会发送到不同的主机上。

## 请求对冲

请求对冲背后的想法是同时向不同的主机发送多个请求，并使用首先响应的上游的结果。请注意，我们通常为幂等的请求配置这个功能，在这种情况下，多次进行相同的调用具有相同的效果。

我们可以通过指定一个对冲策略来配置请求的对冲。目前，Envoy 只在响应请求超时的情况下进行对冲。因此，当一个初始请求超时时，会发出一个重试请求，而不取消原来超时的请求。Envoy 将根据重试策略向下游返回第一个良好的响应。

可以通过设置 `hedge_on_per_try_timeout` 字段为 `true` 来配置对冲。就像重试策略一样，它可以在虚拟主机或路由级别上启用。

```
route_config:
  name: 5xx_route
  virtual_hosts:
  - name: httpbin
    domains: [ "*"]
    hedge_policy:
      hedge_on_per_try_timeout: true
    routes:
    - match:
      ...
    - ...
```

## 请求镜像

使用路由级别的请求镜像策略（`request_mirroring_policies`），我们可以配置 Envoy 将流量从一个集群镜像到另一个集群。

流量镜像或请求镜像是指当传入的请求以一个集群为目标时，将其复制并发送给第二个集群。镜像的请求是“发射并遗忘”的，这意味着 Envoy 在发送主集群的响应之前不会等待影子集群的响应。

请求镜像模式不会影响发送到主集群的流量，而且因为 Envoy 会收集影子集群的所有统计数据，所以这是一种有用的测试技术。

除了“发送并遗忘”之外，还要确保你所镜像的请求是空闲的。否则，镜像请求会扰乱你的服务与之对话的后端。

影子请求中的 `authority/host` 头信息将被添加 `-shadow` 字符串。

为了配置镜像策略，我们在要镜像流量的路由上使用 `request_mirror_policies` 字段。我们可以指定一个或多个镜像策略，以及我们想要镜像的流量的部分。

```
route_config:
  name: my_route
  virtual_hosts:
  - name: httpbin
    domains: [ "*"]
    routes:
    - match:
        prefix: /
      route:
        cluster: httpbin
        request_mirror_policies:
          cluster: mirror_httpbin
          runtime_fraction:
            default_value:
              numerator: 100
...
...
```

上述配置将 100% 地接收发送到集群 `httpbin` 的传入请求，并将其镜像到 `mirror_httpbin`。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 16:16:12

## 速率限制简介

速率限制是一种限制传入请求的策略。它规定了一个主机或客户端在一个特定的时间范围内发送多少次请求。一旦达到限制，例如每秒 100 个请求，我们就说发送请求的客户端受到速率限制。任何速率受限的请求都会被拒绝，并且永远不会到达上游服务。稍后，我们还将讨论可以使用的断路器，以及速率限制如何限制上游的负载并防止级联故障。

Envoy 支持全局（分布式）和局部（非分布式）的速率限制。

全局和局部速率限制的区别在于，我们要用全局速率限制来控制对一组在多个 Envoy 实例之间共享的上游的访问。例如，我们想对一个叫做多个 Envoy 代理的数据库的访问进行速率限制。另一方面，局部速率限制适用于每一个 Envoy 实例。

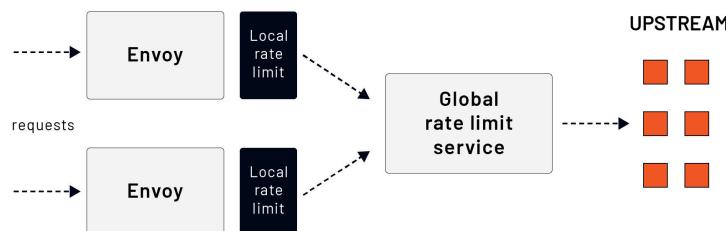


图 2.12.1：全局和局部速率限制

局部和全局速率限制可以一起使用，Envoy 分两个阶段应用它们。首先，应用局部速率限制，然后是全局速率限制。

我们将在接下来的章节中深入研究全局和局部速率限制，并解释这两种情况如何工作。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 16:16:42

## 实验 1：请求匹配

在这个实验中，我们将学习如何配置使用不同的方式来匹配请求。我们将使用 `direct_response`，我们还不会涉及任何 Envoy 集群。

### 路径匹配

以下是我们想放入配置中的规则：

1. 所有请求都需要来自 `hello.io` 域名（即在向代理发出请求时，我们将使用 `Host: hello.io` 标头）
2. 所有向路径 `/api` 发出的请求将返回字符串 `hello - path`
3. 所有向根路径（即 `/`）发出的请求将返回字符串 `hello - prefix`
4. 所有以 `/hello` 开头并在后面加上数字的请求（如 `/hello/1`，`/hello/523`）都应返回 `hello - regex` 字符串

让我们看一下配置：

```

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                stat_prefix: hello_service
              http_filters:
                - name: envoy.filters.http.router
                  route_config:
                    name: route
                    virtual_hosts:
                      - name: hello_vhost
                        domains: ["hello.io"]
                        routes:
                          - match:
                              path: "/api"
                              direct_response:
                                status: 200
                                body:
                                  inline_string: "hello - path"
                          - match:
                              safe_regex:
                                google_re2: {}
                                regex: ^/hello/\d+$
                              direct_response:
                                status: 200
                                body:
                                  inline_string: "hello - regex"
                          - match:
                              prefix: "/"
                              direct_response:
                                status: 200
                                body:
                                  inline_string: "hello - prefix"

```

由于我们只有一个域名，我们将使用一个单一的虚拟主机。域名数组将包含一个单一的域名：`hello.io`。这是 Envoy 要做的第一层匹配。

然后，虚拟主机将有多个路径匹配。首先，我们将使用路径匹配，因为我们想准确匹配 `/api` 路径。其次，我们使用 `^/hello/\d+$` 正则表达式来定义正则表达式匹配。最后，我们定义前缀匹配。注意，定义这些匹配的顺序很重要。如果我们把前缀匹配放在最前面，那么其余的匹配就不会被评估，因为前缀匹配永远是真的。

将上述 YAML 保存为 `2-lab-1-request-matching-1.yaml`，然后运行 `func-e run -c 2-lab-1-request-matching-1.yaml` 来启动 Envoy 代理。

从另一个单独的终端，我们可以进行一些测试调用。

```
$ curl -H "Host: hello.io" localhost:10000
hello - prefix

$ curl -H "Host: hello.io" localhost:10000/api
hello - path

$ curl -H "Host: hello.io" localhost:10000/hello/123
hello - regex
```

## 标头匹配

匹配传入请求的头信息可以与路径匹配相结合，以实现复杂的场景。在这个例子中，我们将使用前缀匹配和不同头信息匹配的组合。

让我们设想一些规则：

- 所有带头信息 `debug: 1` 的 POST 请求发送到 `/1`，返回 422 状态码
- 所有发送至 `/2` 的头为 `path` 且与正则表达式 `^/hello/\d+$` 相匹配的请求都会返回一个 200 状态码和消息 `regex`
- 所有将头名称 `priority` 设置为 1 到 5 之间的请求，发送到 `/3` 会返回一个 200 状态码和消息 `priority`
- 所有发送到 `/4` 的请求，如果存在 `test` 头，都会返回一个 500 状态码

以下是翻译成 Envoy 配置的上述规则：

```
static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                stat_prefix: hello_service
              http_filters:
                - name: envoy.filters.http.router
                  route_config:
                    name: route
                    virtual_hosts:
                      - name: vhost
                        domains: ["*"]
                        routes:
                          - match:
                              path: "/1"
                              headers:
                                - name: ":method"
                                  string_match:
                                    exact: POST
                                - name: "debug"
                                  string_match:
                                    exact: "1"
                              direct_response:
                                status: 422
                          - match:
                              path: "/2"
                              headers:
                                - name: "path"
                                  safe_regex_match:
                                    google_re2: {}
                                    regex: ^/hello/\d+$
                              direct_response:
                                status: 200
                                body:
                                  inline_string: "regex"
                          - match:
                              path: "/3"
                              headers:
                                - name: "priority"
                                  range_match:
                                    start: 1
                                    end: 6
                              direct_response:
                                status: 200
                                body:
                                  inline_string: "priority"
                          - match:
                              path: "/4"
                              headers:
                                - name: "test"
                                  present_match: true
                              direct_response:
                                status: 500
```

将上述 YAML 保存为 `2-lab-1-request-matching-2.yaml` 并运行

```
func-e run -c 2-lab-1-request-matching-2.yaml
```

让我们试着发送几个请求，测试一下规则：

```
$ curl -v -X POST -H "debug: 1" localhost:10000/1
...
> User-Agent: curl/7.64.0
> Accept: /*
> debug: 1
>
< HTTP/1.1 422 Unprocessable Entity

$ curl -H "path: /hello/123" localhost:10000/2
regex

$ curl -H "priority: 3" localhost:10000/3
priority

$ curl -v -H "test: tst" localhost:10000/4
...
> User-Agent: curl/7.64.0
> Accept: /*
> test: tst
>
< HTTP/1.1 500 Internal Server Error
```

## 查询参数匹配

与我们做路径和 Header 匹配的方式相同，我们也可以匹配特定的查询参数和它们的值。查询参数匹配支持与其他两项相同的匹配规则：匹配精确值、前缀和后缀，使用正则表达式，以及检查查询参数是否包含特定值。

让我们考虑配置中的以下情景：

- 所有发送到路径 `/1` 并带有查询参数 `test` 的请求都返回 422 状态码
- 所有发送到路径 `/2` 的查询参数为 `env` 的请求，其值以 `env_` 开头（忽略大小写），返回 200 状态代码
- 所有发送到路径 `/3` 的查询参数 `debug` 设置为 `true` 的请求，都返回 500 状态码

上述规则转化为以下 Envoy 配置：

```
static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                stat_prefix: hello_service
              http_filters:
                - name: envoy.filters.http.router
                  route_config:
                    name: route
                    virtual_hosts:
                      - name: vhost
                        domains: ["*"]
                        routes:
                          - match:
                              path: "/1"
                              query_parameters:
                                - name: test
                                  present_match: true
                            direct_response:
                              status: 422
                          - match:
                              path: "/2"
                              query_parameters:
                                - name: env
                                  string_match:
                                    prefix: env_
                                    ignore_case: true
                            direct_response:
                              status: 200
                          - match:
                              path: "/3"
                              query_parameters:
                                - name: debug
                                  string_match:
                                    exact: "true"
                            direct_response:
                              status: 500
```

将上述 YAML 保存为 `2-lab-1-request-matching-3.yaml` 并运行

```
func-e run -c 2-lab-1-request-matching-3.yaml
```

让我们试着发送几个请求，测试一下规则。

```
$ curl -v localhost:10000/1?test
...
> GET /1?test HTTP/1.1
> Host: localhost:10000
> User-Agent: curl/7.64.0
> Accept: /*
>
< HTTP/1.1 422 Unprocessable Entity

$ curl -v localhost:10000/2?env=eNv_prod
...
> GET /2?env=eNv_prod HTTP/1.1
> Host: localhost:10000
> User-Agent: curl/7.64.0
> Accept: /*
>
< HTTP/1.1 200 OK

$ curl -v localhost:10000/3?debug=true
...
> GET /3?debug=true HTTP/1.1
> Host: localhost:10000
> User-Agent: curl/7.64.0
> Accept: /*
>
< HTTP/1.1 500 Internal Server Error
```

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-03-04 16:13:30

## 实验 2：流量分割

在这个实验中，我们将学习如何使用运行时分数和加权集群来配置 Envoy 的流量分割。

### 使用运行时分数

当我们只有两个上游集群时，运行时分数是一种很好的流量分割方法。运行时分数的工作原理是提供一个运行时分数（例如分子和分母），代表我们想要路由到一个特定集群的流量的分数。然后，我们使用相同的条件（即，在我们的例子中相同的前缀）提供第二个匹配，但不同的上游集群。

让我们创建一个 Envoy 配置，对 70% 的流量返回状态为 201 的直接响应。其余的流量返回状态为 202。

```
static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                stat_prefix: hello_service
                http_filters:
                  - name: envoy.filters.http.router
                route_config:
                  virtual_hosts:
                    - name: hello_vhost
                      domains: ["*"]
                      routes:
                        - match:
                            prefix: "/"
                            runtime_fraction:
                              default_value:
                                numerator: 70
                                denominator: HUNDRED
                              runtime_key: routing.hello_io
                            direct_response:
                              status: 201
                              body:
                                inline_string: "v1"
                        - match:
                            prefix: "/"
                            direct_response:
                              status: 202
                              body:
                                inline_string: "v2"
```

将上述 Envoy 配置保存为 `2-lab-2-traffic-splitting-1.yaml`，并以该配置运行 Envoy。

```
func-e run -c 2-lab-2-traffic-splitting-1.yaml
```

在 Envoy 运行时，我们可以使用 `hey` 工具向代理发送 200 个请求。

```
$ hey http://localhost:10000
...
Status code distribution:
[201] 142 responses
[202] 58 responses
```

看一下状态代码的分布，我们会注意到，我们收到 HTTP 201 响应大概占 71%，其余的响应是 HTTP 202 响应。

## 使用加权集群

当我们有两个以上的上游集群，我们想把流量分给它们，我们可以使用加权集群的方法。在这里，我们单独给每个上游集群分配权重。我们在以前的方法中使用了多个匹配，而在加权集群中，我们将使用一个路由和多个加权集群。

对于这种方法，我们必须定义实际的上游集群。我们将运行 `httpbin` 镜像的三个实例。让我们在 3030、4040 和 5050 端口上运行三个不同的实例；我们将在 Envoy 配置中把它们称为 `instance_1`、`instance_2` 和 `instance_3`。

```
docker run -d -p 3030:80 kennethreitz/httpbin
docker run -d -p 4040:80 kennethreitz/httpbin
docker run -d -p 5050:80 kennethreitz/httpbin
```

一个上游集群可以通过以下片段来定义。

```
clusters:
- name: instance_1
  connect_timeout: 5s
  load_assignment:
    cluster_name: instance_1
  endpoints:
    - lb_endpoints:
      - endpoint:
          address:
            socket_address:
              address: 127.0.0.1
              port_value: 3030
```

让我们创建 Envoy 配置，将 50% 的流量分给 `instance_1`，30% 分给 `instance_2`，20% 分给 `instance_3`。

我们还将启用管理接口来检索指标，这些指标将显示向不同集群发出的请求数量。

```
static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: listener_http
                http_filters:
                  - name: envoy.filters.http.router
                    route_config:
                      name: route
                      virtual_hosts:
                        - name: vh
                          domains: ["*"]
                          routes:
                            - match:
                                prefix: "/"
                              route:
                                weighted_clusters:
                                  clusters:
                                    - name: instance_1
                                      weight: 50
                                    - name: instance_2
                                      weight: 30
                                    - name: instance_3
                                      weight: 20
                            clusters:
                              - name: instance_1
                                connect_timeout: 5s
                                load_assignment:
                                  cluster_name: instance_1
                                  endpoints:
                                    - lb_endpoints:
                                        - endpoint:
                                            address:
                                              socket_address:
                                                address: 127.0.0.1
                                                port_value: 3030
                                    - name: instance_2
                                      connect_timeout: 5s
                                      load_assignment:
                                        cluster_name: instance_1
                                        endpoints:
                                          - lb_endpoints:
                                              - endpoint:
                                                  address:
                                                    socket_address:
                                                      address: 127.0.0.1
                                                      port_value: 4040
                                    - name: instance_3
                                      connect_timeout: 5s
                                      load_assignment:
                                        cluster_name: instance_1
                                        endpoints:
                                          - lb_endpoints:
                                              - endpoint:
                                                  address:
                                                    socket_address:
                                                      address: 127.0.0.1
```

```
    port_value: 5050
  admin:
    address:
      socket_address:
        address: 127.0.0.1
        port_value: 9901
```

将上述 YAML 保存为 `2-lab-2-traffic-splitting-2.yaml` 并运行代理程序：`func-e run -c 2-lab-2-traffic-splitting-2.yaml`。

一旦代理运行，我们将使用 `hey` 来发送 200 个请求。

```
hey http://localhost:10000
```

来自 `hey` 的响应不会帮助我们确定分割，因为每个上游集群的响应都是 HTTP 200。

要想看到流量的分割，请在 `http://localhost:9901/stats/prometheus` 上打开统计表。在指标列表中，寻找 `envoy_cluster_external_upstream_rq` 指标，该指标计算外部上游请求的数量。我们应该看到与此类似的分割。

```
# TYPE envoy_cluster_external_upstream_rq counter
envoy_cluster_external_upstream_rq{envoy_response_code="200",env
envoy_cluster_external_upstream_rq{envoy_response_code="200",env
envoy_cluster_external_upstream_rq{envoy_response_code="200",env
```

如果我们计算一下百分比，我们会发现它们与我们在配置中设置的百分比相对应。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:14:18

## 实验 3：Header 操作

在这个实验中，我们将学习如何在不同的配置级别上操作请求和响应头。

我们将使用一个单一的例子，它的作用如下：

- 对所有的请求添加一个响应头 `lab: 3`
- 在虚拟主机上添加一个请求头 `vh: one`
- 为 `/json` 路由匹配添加一个名为 `json` 响应头。响应头有来自请求头 `hello` 的值

我们将只有一个名为 `single_cluster` 的上游集群，监听端口为 `3030`。让我们运行监听该端口的 `httpbin` 容器：

```
docker run -d -p 3030:80 kennethreitz/httpbin
```

让我们创建遵循上述规则的 Envoy 配置：

```

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                stat_prefix: hello_service
              http_filters:
                - name: envoy.filters.http.router
                  route_config:
                    response_headers_to_add:
                      - header:
                          key: "lab"
                          value: "3"
            virtual_hosts:
              - name: vh_1
                request_headers_to_add:
                  - header:
                      key: vh
                      value: "one"
                domains: ["*"]
            routes:
              - match:
                  prefix: "/json"
                  route:
                    cluster: single_cluster
                    response_headers_to_add:
                      - header:
                          key: "json"
                          value: "%REQ(hello)%"
              - match:
                  prefix: "/"
                  route:
                    cluster: single_cluster
  clusters:
    - name: single_cluster
      connect_timeout: 5s
      load_assignment:
        cluster_name: single_cluster
      endpoints:
        - lb_endpoints:
          - endpoint:
              address:
                socket_address:
                  address: 127.0.0.1
                  port_value: 3030

```

将上述 YAML 保存为 `2-lab-3-header-manipulation-1.yaml` 并运行 Envoy 代理:

```
func-e run -c 2-lab-3-header-manipulation-1.yaml
```

让我们先对 `/headers` 做一个简单的请求, 这将匹配 `/` 前缀:

```
$ curl -v localhost:10000/headers
...
< x-envoy-upstream-service-time: 2
< lab: 3
<
{
  "headers": {
    "Accept": "*/*",
    "Host": "localhost:10000",
    "User-Agent": "curl/7.64.0",
    "Vh": "one",
    "X-Envoy-Expected-Rq-Timeout-Ms": "15000"
  }
}
```

我们会注意到响应 Header `lab: 3` 被设置了。这来自路由配置，并将被添加到我们的所有请求中。

在响应中，我们可以看到 `Vh: one` 头（注意这个大写字母来自 `httpbin` 代码）被添加到请求中。`httpbin` 的响应显示了所收到的头信息（即请求头信息）。

让我们试着向 `/json` 路径发出请求。发送到该路径上的 `httpbin` 的请求将返回一个 JSON 样本。此外，这一次我们将包括一个名为 `hello: world` 的请求头。

```
$ curl -v -H "hello: world" localhost:10000/json
> GET /json HTTP/1.1
> Host: localhost:10000
> User-Agent: curl/7.64.0
> Accept: /*
> hello: world
>
< HTTP/1.1 200 OK
< server: envoy
< json: world
< lab: 3
...
...
```

注意这次我们设置的请求头（`hello: world`），在响应路径上，我们看到 `json: world` 头，它的值来自我们设置的请求头。同样地，`lab: 3` 响应头被设置。请求头 `Vh: one` 也同时被设置，但这次我们看不到它，因为它没有被输出。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-03-04 16:16:30

## 实验 4：重试

在这个实验中，我们将学习如何配置不同的重试策略。我们将使用 `httpbin` Docker 镜像，因为我们可以向不同的路径（例如 `/status/[statuscode]`）发送请求，而 `httpbin` 会以该状态码进行响应。

确保你有 `httpbin` 容器在 3030 端口监听。

```
docker run -d -p 3030:80 kennethreitz/httpbin
```

让我们创建 Envoy 配置，定义一个关于 `5xx` 响应的简单重试策略。我们还将启用管理接口，这样我们就可以在指标中看到重试的报告。

```
static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: hello_service
              http_filters:
                - name: envoy.filters.http.router
                  route_config:
                    virtual_hosts:
                      - name: httpbin
                        domains: ["*"]
                        routes:
                          - match:
                              prefix: "/"
                            route:
                              cluster: httpbin
                              retry_policy:
                                retry_on: "5xx"
                                num_retries: 5
            clusters:
              - name: httpbin
                connect_timeout: 5s
                load_assignment:
                  cluster_name: single_cluster
                  endpoints:
                    - lb_endpoints:
                        - endpoint:
                            address:
                              socket_address:
                                address: 127.0.0.1
                                port_value: 3030
                admin:
                  address:
                    socket_address:
                      address: 127.0.0.1
                      port_value: 9901
```

将上述 YAML 保存为 `2-lab-4-retries-1.yaml`，然后运行 Envoy 代理：

```
func-e run -c 2-lab-4-retries-1.yaml
```

让我们向 `/status/500` 路径发送一个单一请求。

```
$ curl -v localhost:10000/status/500
...
< HTTP/1.1 500 Internal Server Error
< server: envoy
...
< content-length: 0
< x-envoy-upstream-service-time: 276
```

正如预期的那样，我们收到了一个 500 响应。另外，注意到 `x-envoy-upstream-service-time`（上游主机处理该请求所花费的时间，以毫秒为单位）比我们发送 `/status/200` 请求时要大得多。

```
$ curl localhost:10000/status/200
...
< HTTP/1.1 200 OK
< server: envoy
...
< content-length: 0
< x-envoy-upstream-service-time: 2
```

这是因为 Envoy 执行了重试，但最后还是失败了。同样，如果我们在管理界面 (<http://localhost:9901/stats/prometheus>) 上打开统计页面，我们会发现代表重试次数的指标 (`envoy_cluster_retry_upstream_rq`) 的数值为 5。

```
# TYPE envoy_cluster_retry_upstream_rq counter
envoy_cluster_retry_upstream_rq{envoy_response_code="500",envoy_
```

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-03-04 16:17:20

## 实验 5：局部速率限制

在这个实验中，我们将学习如何配置一个局部速率限制器。我们将使用运行在 3030 端口的 `httpbin` 容器。

```
docker run -d -p 3030:80 kennethreitz/httpbin
```

让我们创建一个有五个令牌的速率限制器。每隔 30 秒，速率限制器就会向桶里补充 5 个令牌。

```

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: ingress_http
              route_config:
                name: local_route
                virtual_hosts:
                  - name: instance_1
                    domains: ["*"]
                    routes:
                      - match:
                          prefix: /status
                        route:
                          cluster: instance_1
                      - match:
                          prefix: /headers
                        route:
                          cluster: instance_1
              typed_per_filter_config:
                envoy.filters.http.local_ratelimit:
                  "@type": type.googleapis.com/envoy.extension
                  stat_prefix: headers_route
                  token_bucket:
                    max_tokens: 5
                    tokens_per_fill: 5
                    fill_interval: 30s
                    filter_enabled:
                      default_value:
                        numerator: 100
                        denominator: HUNDRED
                    filter_enforced:
                      default_value:
                        numerator: 100
                        denominator: HUNDRED
                    response_headers_to_add:
                      - append: false
                        header:
                          key: x-rate-limited
                          value: OH_NO
              http_filters:
                - name: envoy.filters.http.local_ratelimit
                  typed_config:
                    "@type": type.googleapis.com/envoy.extensions.filter
                    stat_prefix: httpbin_rate_limiter
                - name: envoy.filters.http.router
  clusters:
    - name: instance_1
      connect_timeout: 0.25s
      type: STATIC
      lb_policy: ROUND_ROBIN
      load_assignment:
        cluster_name: instance_1
        endpoints:
          - lb_endpoints:
              - endpoint:
                  address:

```

```
    socket_address:  
      address: 127.0.0.1  
      port_value: 3030  
  admin:  
    address:  
      socket_address:  
        address: 127.0.0.1  
        port_value: 9901
```

将上述 YAML 保存为 `2-lab-5-local-rate-limiter-1.yaml`，用 `func-e run -c 2-lab-5-local-rate-limiter-1.yaml` 运行 Envoy。

上述配置为路由 `/headers` 启用了一个局部速率限制器。此外，一旦达到速率限制，我们将在响应中添加一个头信息 (`x-rate-limited`)。

如果我们在 30 秒内向 `http://localhost:10000/headers` 发出超过 5 个请求，我们会得到 HTTP 429 的响应。

```
$ curl -v localhost:10000/headers  
...  
> GET /headers HTTP/1.1  
> Host: localhost:10000  
> User-Agent: curl/7.64.0  
> Accept: */*  
>  
< HTTP/1.1 429 Too Many Requests  
< x-rate-limited: OH_NO  
...  
local_rate_limited
```

另外，注意到 Envoy 设置的 `x-rate-limited` 头。

一旦我们被限制了速率，我们将不得不等待 30 秒，让速率限制器再次用令牌把桶填满。我们也可以尝试向 `/status/200` 发出请求，你会发现我们不会在这个路径上受到速率限制。

如果我们打开统计页面 (`localhost:9901/stats/prometheus`)，我们会发现限速指标是使用我们配置的 `headers_route_rate_limiter` 统计前缀记录的。

```
# TYPE envoy_headers_route_http_local_rate_limit_enabled counter  
envoy_headers_route_http_local_rate_limit_enabled{} 13  
  
# TYPE envoy_headers_route_http_local_rate_limit_enforced counter  
envoy_headers_route_http_local_rate_limit_enforced{} 8  
  
# TYPE envoy_headers_route_http_local_rate_limit_ok counter  
envoy_headers_route_http_local_rate_limit_ok{} 5  
  
# TYPE envoy_headers_route_http_local_rate_limit_rate_limited counter  
envoy_headers_route_http_local_rate_limit_rate_limited{} 8
```

## 实验 6：全局速率限制

在这个实验中，我们将学习如何配置一个全局速率限制器。我们将使用[速率限制器服务](#)和一个 Redis 实例来跟踪令牌。我们将使用 Docker Compose 来运行 Redis 和速率限制器服务容器。

让我们首先创建速率限制器服务的配置。

```
domain: my_domain
descriptors:
- key: generic_key
  value: instance_1
  descriptors:
    - key: header_match
      value: get_request
      rate_limit:
        unit: MINUTE
        requests_per_unit: 5
```

我们指定一个通用键（`instance_1`）和一个名为 `header_match` 的描述符，速率限制为 5 个请求 / 分钟。

将上述文件保存到 `/config/r1-config.yaml` 文件中。

现在我们可以运行 Docker Compose 文件，它将启动 Redis 和速率限制器服务。

```
version: "3"
services:
  redis:
    image: redis:alpine
    expose:
      - 6379
    ports:
      - 6379:6379
    networks:
      - ratelimit-network

# Rate limit service configuration
ratelimit:
  image: envoyproxy/ratelimit:bd46f11b
  command: /bin/ratelimit
  ports:
    - 10001:8081
    - 6070:6070
  depends_on:
    - redis
  networks:
    - ratelimit-network
  volumes:
    - $PWD/config:/data/config/config
  environment:
    - USE_STATSD=false
    - LOG_LEVEL=debug
    - REDIS_SOCKET_TYPE=tcp
    - REDIS_URL=redis:6379
    - RUNTIME_ROOT=/data
    - RUNTIME_SUBDIRECTORY=config

networks:
  ratelimit-network:
```

将上述文件保存为 `rl-docker-compose.yaml`，并使用下面的命令启动所有容器：

```
$ docker-compose -f rl-docker-compose.yaml up
```

为了确保速率限制器服务正确读取配置，我们可以检查容器的输出或使用速率限制器服务的调试端口。

```
$ curl localhost:6070/rlconfig
my_domain.generic_key_instance_1.header_match_get_request: unit=
```

随着速率限制器和 Redis 的启动和运行，我们可以启动 `httpbin` 容器。

```
docker run -d -p 3030:80 kennethreitz/httpbin
```

接下来，我们将创建 Envoy 配置，定义速率限制动作。我们将设置描述符 `instance_1` 和 `get_request`，只要有 GET 请求被发送到 `httpbin`。

在 `http_filters` 下，我们通过指定域名（`my_domain`）和指向 Envoy 可以用来到达速率限制服务的集群来配置 `ratelimit` 过滤器。

```

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: ingress_http
              route_config:
                name: local_route
                virtual_hosts:
                  - name: namespace.local_service
                    domains: ["*"]
                    routes:
                      - match:
                          prefix: /
                        route:
                          cluster: instance_1
                          rate_limits:
                            - actions:
                                - generic_key:
                                    descriptor_value: instance_1
                                - header_value_match:
                                    descriptor_value: get_request
                                    headers:
                                      - name: ":method"
                                        exact_match: GET
                          http_filters:
                            - name: envoy.filters.http.ratelimit
                              typed_config:
                                "@type": type.googleapis.com/envoy.extensions.filters.
                                  domain: my_domain
                                  enable_x_ratelimit_headers: DRAFT_VERSION_03
                                  rate_limit_service:
                                    transport_api_version: V3
                                    grpc_service:
                                      envoy_grpc:
                                        cluster_name: rate-limit
                            - name: envoy.filters.http.router
            clusters:
              - name: instance_1
                connect_timeout: 0.25s
                type: STATIC
                lb_policy: ROUND_ROBIN
                load_assignment:
                  cluster_name: instance_1
                  endpoints:
                    - lb_endpoints:
                        - endpoint:
                            address:
                              socket_address:
                                address: 127.0.0.1
                                port_value: 3030
              - name: rate-limit
                connect_timeout: 1s
                type: STATIC
                lb_policy: ROUND_ROBIN
                protocol_selection: USE_CONFIGURED_PROTOCOL
                http2_protocol_options: {}
                load_assignment:

```

```
cluster_name: rate-limit
endpoints:
- lb_endpoints:
  - endpoint:
    address:
      socket_address:
        address: 127.0.0.1
        port_value: 10001
admin:
address:
socket_address:
address: 127.0.0.1
port_value: 9901
```

将上述 YAML 保存为 `2-lab-6-global-rate-limiter-1.yaml`，并使用  
`func-e run -c 2-lab-6-global-rate-limiter-1.yaml` 运行代理。

我们现在可以发送五个以上的请求，我们会得到速率限制。

```
$ curl -v localhost:10000
...
< HTTP/1.1 429 Too Many Requests
< x-envoy-ratelimited: true
< x-ratelimit-limit: 5, 5;w=60
< x-ratelimit-remaining: 0
< x-ratelimit-reset: 25
...
```

我们收到了 429 响应，以及表明我们受到速率限制的响应头；在受到速率限制之前我们可以发出多少个请求 (`x-ratelimit-remaining`) 以及速率限制何时重置 (`x-ratelimit-reset`)。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:19:00

## 集群概述

在本章中，我们将学习集群以及如何管理它们。在 Envoy 的集群配置部分，我们可以配置一些功能，如负载均衡、健康检查、连接池、异常点检测等。

在本章结束时，你将了解集群和端点是如何工作的，以及如何配置负载均衡策略、异常点检测和断路。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:45:37

## 服务发现

集群可以在配置文件中静态配置，也可以通过集群发现服务（CDS）API 动态配置。每个集群都是一个端点的集合，Envoy 需要解析这些端点来发送流量。

解析端点的过程被称为 **服务发现**。

### 什么是端点？

集群是一个识别特定主机的端点的集合。每个端点都有以下属性。

#### 地址 ( `address` )

该地址代表上游主机地址。地址的形式取决于集群的类型。对于 STATIC 或 EDS 集群类型，地址应该是一个 IP，而对于 LOGICAL 或 STRICT DNS 集群类型，地址应该是一个通过 DNS 解析的主机名。

#### 主机名 ( `hostname` )

一个与端点相关的主机名。注意，主机名不用于路由或解析地址。它与端点相关联，可用于任何需要主机名的功能，如自动主机重写。

#### 健康检查配置 ( `health_check_config` )

可选的健康检查配置用于健康检查器联系健康检查主机。该配置包含主机名和可以联系到主机以执行健康检查的端口。注意，这个配置只适用于启用了主动健康检查的上游集群。

## 服务发现类型

有五种支持的服务发现类型，我们将更详细地介绍。

### 静态 ( `STATIC` )

静态服务发现类型是最简单的。在配置中，我们为集群中的每个主机指定一个已解析的网络名称。例如：

```

clusters:
- name: my_cluster_name
  type: STATIC
  load_assignment:
    cluster_name: my_service_name
    endpoints:
      - lb_endpoints:
        - endpoint:
          address:
            socket_address:
              address: 127.0.0.1
              port_value: 8080

```

注意，如果我们不提供类型，默認為 `STATIC`。

## 严格的 DNS ( `STRICT_DNS` )

通过严格的 DNS，Envoy 不断地、异步地解析集群中定义的 DNS 端点。如果 DNS 查询返回多个 IP 地址，Envoy 假定它们是集群的一部分，并在它们之间进行负载均衡。同样，如果 DNS 查询返回 0 个主机，Envoy 就认为集群没有任何主机。

关于健康检查的说明——如果多个 DNS 名称解析到同一个 IP 地址，则不共享健康检查。这可能会给上游主机造成不必要的负担，因为 Envoy 会对同一个 IP 地址进行多次健康检查（跨越不同的 DNS 名称）。

当 `respect_dns_ttl` 字段被启用时，我们可以使用 `dns_refresh_rate` 控制 DNS 名称的连续解析。如果不指定，DNS 刷新率默认为 5000ms。另一个设置 (`dns_failure_refresh_rate`) 控制故障时的刷新频率。如果没有提供，Envoy 使用 `dns_refresh_rate`。

下面是一个 `STRICT_DNS` 服务发现类型的例子。

```

clusters:
- name: my_cluster_name
  type: STRICT_DNS
  load_assignment:
    cluster_name: my_service_name
    endpoints:
      - lb_endpoints:
        - endpoint:
          address:
            socket_address:
              address: my-service
              port_value: 8080

```

## 逻辑 DNS ( `LOGICAL_DNS` )

逻辑 DNS 服务发现与严格 DNS 类似，它使用异步解析机制。然而，它只使用需要启动新连接时返回的第一个 IP 地址。

因此，一个逻辑连接池可能包含与各种不同上游主机的物理连接。这些连接永远不会耗尽，即使在 DNS 解析返回零主机的情况下。

### 什么是连接池？

集群中的每个端点将有一个或多个连接池。例如，根据所支持的上游协议，每个协议可能有一个连接池分配。Envoy 中的每个工作线程也为每个集群维护其连接池。例如，如果 Envoy 有两个线程和一个同时支持 HTTP/1 和 HTTP/2 的集群，将至少有四个连接池。连接池的方式是基于底层线程协议的。对于 HTTP/1.1，连接池根据需要获取端点的连接（最多到断路限制）。当请求变得可用时，它们就被绑定到连接上。当使用 HTTP/2 时，连接池在一个连接上复用多个请求，最多到 `max_concurrent_streams` 和 `max_requests_per_connections` 指定的限制。HTTP/2 连接池建立尽可能多的连接，以满足请求。

逻辑 DNS 的一个典型用例是用于大规模网络服务。通常使用轮询 DNS，它们在每次查询时返回多个 IP 地址的不同结果。如果我们使用严格的 DNS 解析，Envoy 会认为集群端点在每次内部解析时都会改变，并会耗尽连接池。使用逻辑 DNS，连接将保持存活，直到它们被循环。

与严格的 DNS 一样，逻辑 DNS 也使用 `respect_dns_ttl` 和 `dns_refresh_rate` 字段来配置 DNS 刷新率。

```
clusters:
- name: my_cluster_name
  type: LOGICAL_DNS
  load_assignment:
    cluster_name: my_service_name
  endpoints:
    - lb_endpoints:
      - endpoint:
          address:
            socket_address:
              address: my-service
              port_value: 8080
```

## 端点发现服务（EDS）

Envoy 可以使用端点发现服务来获取集群的端点。通常情况下，这是首选的服务发现机制。Envoy 获得每个上游主机的显式知识（即不需要通过 DNS 解析的负载均衡器进行路由）。每个端点都可以携带额外的属性，可以告知 Envoy 负载均衡的权重和金丝雀状态区，等等。

```
clusters:
- name: my_cluster_name
  type: EDS
  eds_cluster_config:
    eds_config:
      ...
      ...
```

我们会在动态配置和 xDS 一章中更详细地解释动态配置。

## 原始目的地 ( `ORIGINAL_DST` )

当与 Envoy 的连接通过 iptables REDIRECT 或 TPROXY 目标或与代理协议的连接时，我们使用原来的目标集群类型。

在这种情况下，请求被转发到重定向元数据（例如，使用 `x-envoy-original-dst-host` 头）地址的上游主机，而无需任何配置或上游主机发现。

当上游主机的连接闲置时间超过 `cleanup_interval` 字段中指定的时间（默认为 5000 毫秒）时，这些连接会被汇集起来并被刷新。

```
clusters:  
- name: original_dst_cluster  
  type: ORIGINAL_DST  
  lb_policy: ORIGINAL_DST_LB
```

ORIGINAL\_DST 集群类型可以使用的唯一负载均衡策略是 ORIGINAL\_DST\_LB 策略。

除了上述服务发现机制外，Envoy 还支持自定义集群发现机制。我们可以使用 `cluster_type` 字段配置自定义的发现机制。

Envoy 支持两种类型的健康检查，主动和被动。我们可以同时使用这两种类型的健康检查。在主动健康检查中，Envoy 定期向端点发送请求以检查其状态。使用被动健康检查，Envoy 监测端点如何响应连接。它使 Envoy 甚至在主动健康检查将其标记为不健康之前就能检测到一个不健康的端点。Envoy 的被动健康检查是通过异常点检测实现的。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 16:21:46

# 主动健康检查

Envoy 支持端点上不同的主动健康检查方法。HTTP、TCP、gRPC 和 Redis 健康检查。健康检查方法可以为每个集群单独配置。我们可以通过集群配置中的 `health_checks` 字段来配置健康检查。

无论选择哪种健康检查方法，都需要定义几个常见的配置设置。

**超时** (`timeout`) 表示分配给等待健康检查响应的时间。如果在这个字段中指定的时间值内没有达到响应，健康检查尝试将被视为失败。**间隔** (`internal`) 指定健康检查之间的时间节奏。例如，5 秒的间隔将每 5 秒触发一次健康检查。

其他两个必要的设置可用于确定一个特定的端点何时被认为是健康或不健康的。`healthy_threshold` 指定在一个端点被标记为健康之前所需的 "健康" 检查（例如，HTTP 200 响应）的数量。`unhealthy_threshold` 的作用与此相同，但是对于 "不健康" 的健康检查，它指定了在一个端点被标记为不健康之前所需的不健康检查的数量。

## 1. HTTP 健康检查

Envoy 向端点发送一个 HTTP 请求。如果端点回应的是 HTTP 200，Envoy 认为它是健康的。200 响应是默认的响应，被认为是健康响应。使用 `expected_statuses` 字段，我们可以通过提供一个被认为是健康的 HTTP 状态的范围来进行自定义。

如果端点以 HTTP 503 响应，`unhealthy_threshold` 被忽略，并且端点立即被认为是不健康的。

```
clusters:
- name: my_cluster_name
  health_checks:
    - timeout: 1s
      interval: 0.25s
      unhealthy_threshold: 5
      healthy_threshold: 2
      http_health_check:
        path: "/health"
        expected_statuses:
          - start: 200
            end: 299
...

```

例如，上面的片段定义了一个 HTTP 健康检查，Envoy 将向集群中的端点发送一个 `/health` 路径的 HTTP 请求。Envoy 每隔 0.25s (`internal`) 发送一次请求，在超时前等待 1s (`timeout`)。要被认为是健康的，端点必须以 200 和 299 之间的状态 (`expected_statuses`) 响应两次 (`healthy_threshold`)。端点需要

以任何其他状态代码响应五次（`unhealthy_threshold`）才能被认为是不是健康的。此外，如果端点以 HTTP 503 响应，它将立即被视为不健康（`unhealthy_threshold` 设置被忽略）。

## 2. TCP 健康检查

我们指定一个 Hex 编码的有效载荷（例如：`68656C6C6F`），并将其发送给终端。如果我们设置了一个空的有效载荷，Envoy 将进行仅连接的健康检查，它只尝试连接到端点，如果连接成功就认为是成功的。

除了被发送的有效载荷外，我们还需要指定响应。Envoy 将对响应进行模糊匹配，如果响应与请求匹配，则认为该端点是健康的。

```
clusters:
- name: my_cluster_name
  health_checks:
    - timeout: 1s
      interval: 0.25s
      unhealthy_threshold: 1
      healthy_threshold: 1
      tcp_health_check:
        send:
          text: "68656C6C6F"
        receive:
          - text: "68656C6C6F"
...

```

## 3. gRPC 健康检查

本健康检查遵循 `grpc.health.v1.Health` 健康检查协议。查看 [GRPC 健康检查协议文档](#) 以了解更多关于其工作方式的信息。

我们可以设置的两个可选的配置值是 `service_name` 和 `authority`。服务名称是设置在 `grpc.health.v1.Health` 的 `HealthCheckRequest` 的 `service` 字段中的值。授权是 `:authority` 头的值。如果它是空的，Envoy 会使用集群的名称。

```
clusters:
- name: my_cluster_name
  health_checks:
    - timeout: 1s
      interval: 0.25s
      unhealthy_threshold: 1
      healthy_threshold: 1
      grpc_health_check: {}
...

```

## 4. Redis 健康检查

Redis 健康检查向端点发送一个 Redis PING 命令，并期待一个 PONG 响应。如果上游的 Redis 端点回应的不是 PONG，就会立即导致健康检查失败。我们也可以指定一个 `key`，Envoy 会执行 `EXIST <key>` 命令，

而不是 PING 命令。如果 Redis 的返回值是 0 (即密钥不存在) , 那么该端点就是健康的。任何其他响应都被视为失败。

```

clusters:
- name: my_cluster_name
  health_checks:
    - timeout: 1s
      interval: 0.25s
      unhealthy_threshold: 1
      healthy_threshold: 1
      redis_health_check:
        key: "maintenance"
...

```

上面的例子检查键 "维护" (如 `EXIST maintenance` ) , 如果键不存在, 健康检查就通过。

## HTTP 健康检查过滤器

HTTP 健康检查过滤器可以用来限制产生的健康检查流量。过滤器可以在不同的操作模式下运行, 控制流量是否被传递给本地服务 (即不传递或传递) 。

### 1. 非穿透模式

当以非穿透模式运行时, 健康检查请求永远不会被发送到本地服务。

Envoy 会以 HTTP 200 或 HTTP 503 进行响应, 这取决于服务器当前的耗尽状态。

非穿透模式的一个变种是, 如果上游集群中至少有指定比例的端点可用, 则返回 HTTP 200。端点的百分比可以用

`cluster_min_healthy_percentages` 字段来配置。

```

...
  pass_through_mode: false
  cluster_min_healthy_percentages:
    value: 15
...

```

### 2. 穿透模式

在穿透模式下, Envoy 将每个健康检查请求传递给本地服务。该服务可以用 HTTP 200 或 HTTP 503 来响应。

穿透模式的另一个设置是使用缓存。Envoy 将健康检查请求传递给服务, 并将结果缓存一段时间 ( `cache_time` ) 。任何后续的健康检查请求将使用缓存起来的值。一旦缓存失效, 下一个健康检查请求会再次传递给服务。

```
...
  pass_through_mode: true
  cache_time: 5m
...
```

上面的片段启用了穿透模式，缓存在 5 分钟内到期。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 22:44:18

## 异常点检测

第二种类型的健康检查被称为被动健康检查。异常点检测（**outlier detection**）是一种被动的健康检查形式。说它 "被动" 是因为 Envoy 没有 "主动" 发送任何请求来确定端点的健康状况。相反，Envoy 观察不同端点的性能，以确定它们是否健康。如果端点被认为是不健康的，它们就会被移除或从健康负载均衡池中弹出。

端点的性能是通过连续失败、时间成功率、延迟等来确定的。

为了使异常点检测发挥作用，我们需要过滤器来报告错误、超时和重置。目前，有四个过滤器支持异常点检测。HTTP 路由器、TCP 代理、Redis 代理和 Thrift 代理。

检测到的错误根据起源点分为两类。

### 1. 来自外部的错误

这些错误是针对事务的，发生在上游服务器上，是对收到的请求的回应。这些错误是在 Envoy 成功连接到上游主机后产生的。例如，端点响应的是 HTTP 500。

### 2. 本地产生的错误

Envoy 产生这些错误是为了应对中断或阻止与上游主机通信的事件，例如超时、TCP 重置、无法连接到指定端口等。

这些错误也取决于过滤器的类型。例如，HTTP 路由器过滤器可以检测两种错误。相反，TCP 代理过滤器不理解 TCP 层以上的任何协议，只报告本地产生的错误。

在配置中，我们可以指定是否可以区分本地和外部产生的错误（使用 `split_external_local_origin_errors` 字段）。这允许我们通过单独的计数器跟踪错误，并配置异常点检测，对本地产生的错误做出反应，而忽略外部产生的错误，反之亦然。默认模式错误将不被分割（即 `split_external_local_origin_errors` 为 `false`）。

## 端点弹出

当一个端点被确定为异常点时，Envoy 将检查它是否需要从健康负载均衡池中弹出。如果没有端点被弹出，Envoy 会立即弹出异常（不健康的）端点。否则，它会检查 `max_ejection_percent` 设置，确保被弹出的端点数量低于配置的阈值。如果超过 `max_ejection_percent` 的主机已经被弹出，该端点就不会被弹出了。

每个端点被弹出的时间是预先确定的。我们可以使用 `base_ejection_time` 值来配置弹出时间。这个值要乘以端点连续被弹出的次数。如果端点继续失败，它们被弹出的时间会越来越长。这里的第二个设置叫做 `max_ejection_time`。它控制端点被弹出的最长时间——也就是说，端点被弹出的最长时间在 `max_ejection_time` 值中被指定。

Envoy 在 `internal` 字段中指定的间隔时间内检查每个端点的健康状况。每检查一次端点是否健康，弹出的倍数就会被递减。经历弹出时间后，端点会自动返回到健康的负载均衡池中。

现在我们了解了异常点检测和端点弹出的基本知识，让我们看看不同的异常点检测方法。

## 检测类型

Envoy 支持以下五种异常点检测类型。

## 1. 连续的 5xx

这种检测类型考虑到了所有产生的错误。Envoy 内部将非 HTTP 过滤器产生的任何错误映射为 HTTP 5xx 代码。

当错误类型被分割时，该检测类型只计算外部产生的错误，忽略本地产生的错误。如果端点是一个 HTTP 服务器，只考虑 5xx 类型的错误。

如果一个端点返回一定数量的 5xx 错误，该端点会被弹出。`consecutive 5xx` 值控制连续 5xx 错误的数量。

```
clusters:
- name: my_cluster_name
  outlier_detection:
    interval: 5s
    base_ejection_time: 15s
    max_ejection_time: 50s
    max_ejection_percent: 30
    consecutive_5xx: 10
    ...

```

上述异常点检测，一旦它失败 10 次，将弹出一个失败的端点。失败的端点会被弹出 15 秒（`base_ejection_time`）。在多次弹出的情况下，单个端点被弹出的最长时间是 50 秒（`max_ejection_time`）。在一个失败的端点被弹出之前，Envoy 会检查是否有超过 30% 的端点已经被弹出（`max_ejection_percent`），并决定是否弹出这个失败的端点。

## 2. 连续的网关故障

连续网关故障类型与连续 5xx 类型类似。它将 5xx 错误的一个子集，称为“网关错误”（如 502、503 或 504 状态代码）和本地源故障，如超时、TCP 复位等。

这种检测类型考虑了分隔模式下的网关错误，并且只由 HTTP 过滤器支持。连续错误的数量可通过 `contrieable_gateway_failure` 字段进行配置。

```
clusters:
- name: my_cluster_name
  outlier_detection:
    interval: 5s
    base_ejection_time: 15s
    max_ejection_time: 50s
    max_ejection_percent: 30
    consecutive_gateway_failure: 10
  ...

```

### 3. 连续的本地源失败

这种类型只在分隔模式下启用 (`split_external_local_origin_errors` 为 `true`)，它只考虑本地产生的错误。连续失败的数量可以通过 `contriable_local_origin_failure` 字段进行配置。如果未提供，默认为 5。

```
clusters:  
- name: my_cluster_name  
  outlier_detection:  
    interval: 5s  
    base_ejection_time: 15s  
    max_ejection_time: 50s  
    max_ejection_percent: 30  
    consecutive_local_origin_failure: 10  
    ...
```

#### 4. 成功率

成功率异常点检测汇总了集群中每个端点的成功率数据。基于成功率，它将在给定的时间间隔内弹出端点。在默认模式下，所有的错误都被考虑，而在分隔模式下，外部和本地产生的错误被分别处理。

通过 `success_rate_request_volume` 值，我们可以设置最小请求量。如果请求量小于该字段中指定的请求量，将不计算该主机的成功率。同样地，我们可以使用 `success_rate_minimum_hosts` 来设置具有最小要求的请求量的端点数量。如果具有最小要求的请求量的端点数量少于 `success_rate_minimum_hosts` 中设置的值，Envoy 将不会进行异常点检测。

`success_rate_stdev_factor` 用于确定弹出阈值。弹出阈值是平均成功率和该系数与平均成功率标准差的乘积之间的差。

平均值 - (stdev \* success\_rate\_stdev\_factor)

这个系数被除以一千，得到一个双数。也就是说，如果想要的系数是1.9，那么运行时间值应该是1900。

## 5. 故障率

故障率异常点检测与成功率类似。不同的是，它不依赖于整个集群的平均成功率。相反，它将该值与用户在 `failure_percentage_threshold` 字段中配置的阈值进行比较。如果某个主机的故障率大于或等于这个值，该主机就会被弹出。

可以使用 `failure_percentage_minimum_hosts` 和  
`failure_percentage_request_volume` 配置最小主机和请求量。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 22:45:49

## 断路器

断路是一种重要的模式，可以帮助服务的弹性。断路模式通过控制和管理对故障服务的访问来防止额外的故障。它允许我们快速失败，并尽快向下游反馈。

让我们看一个定义断路的片段。

```
...
  clusters:
    - name: my_cluster_name
  ...
  circuit_breakers:
    thresholds:
      - priority: DEFAULT
        max_connections: 1000
      - priority: HIGH
        max_requests: 2000
  ...
...
```

我们可以为每条路由的优先级分别配置断路器的阈值。例如，较高优先级的路由应该有比默认优先级更高的阈值。如果超过了任何阈值，断路器就会断开，下游主机就会收到 HTTP 503 响应。

我们可以用多种选项来配置断路器。

### 1. 最大连接数（`max_connections`）

指定 Envoy 与集群中所有端点的最大连接数。如果超过这个数字，断路器会断开，并增加集群的 `upstream_cx_overflow` 指标。默认值是 1024。

### 2. 最大的排队请求（`max_pending_requests`）

指定在等待就绪的连接池连接时被排队的最大请求数。当超过该阈值时，Envoy 会增加集群的 `upstream_rq_pending_overflow` 统计。默认值是 1024。

### 3. 最大请求（`max_requests`）

指定 Envoy 向集群中所有端点发出的最大并行请求数。默认值是 1024。

### 4. 最大重试（`max_retries`）

指定 Envoy 允许给集群中所有终端的最大并行重试次数。默认值是 3，如果这个断路器溢出，`upstream_rq_retry_overflow` 计数器就会递增。

另外，我们可以将断路器与重试预算（`retry_budget`）相结合。通过指定重试预算，我们可以将并发重试限制在活动请求的数量上。

Copyright © 2017-2022 | 基于 CC 4.0 协议 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 22:45:01

## 负载均衡

负载均衡是一种在单个上游集群的多个端点之间分配流量的方式。在众多端点之间分配流量的原因是为了最好地利用可用资源。

为了实现资源的最有效利用，Envoy 提供了不同的负载均衡策略，可以分为两组：**全局负载均衡**和**分布式负载均衡**。不同的是，在全局负载均衡中，我们使用单一的控制平面来决定端点之间的流量分配。Envoy 决定负载如何分配（例如，使用主动健康检查、分区感知路由、负载均衡策略）。

在多个端点之间分配负载的技术之一是**一致性哈希**。服务器使用请求的一部分来创建一个哈希值来选择一个端点。在模数散列中，哈希值被认为是一个巨大的数字。为了得到发送请求的端点索引，我们将哈希值与可用端点的数量取余数（`index=hash % endpointCount`）。如果端点的数量是稳定的，这种方法效果很好。然而，如果端点被添加或删除（即它们不健康，我们扩大或缩小它们的规模，等等），大多数请求将在一个与以前不同的端点上结束。

一致性哈希是一种方法，每个端点根据某些属性被分配多个有价值的值。然后，每个请求被分配到具有最接近哈希值的端点。这种方法的价值在于，当我们添加或删除端点时，大多数请求最终会被分配到与之前相同的端点。拥有这种“粘性”是有帮助的，因为它不会干扰端点持有的任何缓存。

## 负载均衡策略

Envoy 使用其中一个负载均衡策略来选择一个端点发送流量。负载均衡策略是可配置的，可以为每个上游集群分别指定。请注意，负载均衡只在健康的端点上执行。如果没有定义主动或被动的健康检查，则假定所有端点都是健康的。

我们可以使用 `lb_policy` 字段和其他针对所选策略的字段来配置负载均衡策略。

### 加权轮询（默认）

加权轮询（`ROUND_ROBIN`）以轮询顺序选择端点。如果端点是加权的，那么就会使用加权的轮询顺序。这个策略给我们提供了一个可预测的请求在所有端点的分布。权重较高的端点将在轮转中出现得更频繁，以实现有效的加权。

### 加权的最小请求

加权最小请求（`LEAST_REQUEST`）算法取决于分配给端点的权重。

如果所有的端点权重相等，算法会随机选择 N 个可用的端点（`choice_count`），并挑选出活动请求最少的一个。

如果端点的权重不相等，该算法就会转入一种模式，即使用加权的循环计划，其中的权重是根据选择时端点的请求负载动态调整。

以下公式用于动态计算权重。

```
weight = load_balancing_weight / (active_requests + 1)^active_re
```

`active_request_bias` 是可配置的（默认为 1.0）。主动请求偏差越大，主动请求就越积极地降低有效权重。

如果 `active_request_bias` 被设置为 0，那么算法的行为就像轮询一样，在挑选时忽略了活动请求数。

我们可以使用 `least_request_lb_config` 字段来设置加权最小请求的可选配置。

```
...
lb_policy: LEAST_REQUEST
least_request_lb_config:
  choice_count: 5
  active_request_bias: 0.5
...
```

## 环形哈希

环形哈希（或模数散列）算法（`RING_HASH`）实现了对端点的一致性哈希。每个端点地址（默认设置）都被散列并映射到一个环上。Envoy 通过散列一些请求属性，并在环上顺时针找到最近的对应端点，将请求路由到一个端点。哈希键默认为端点地址；然而，它可以使用 `hash_key` 字段改变为任何其他属性。

我们可以通过指定最小（`minimum_ring_size`）和最大（`maximum_ring_size`）的环形哈希算法，并使用统计量（`min_hashes_per_host` 和 `max_hashes_per_host`）来确保良好的分布。环越大，请求的分布就越能反映出所需的权重。最小环大小默认为 1024 个条目（限制在 8M 个条目），而最大环大小默认为 8M（限制在 8M）。

我们可以使用 `ring_hash_lb_config` 字段设置环形哈希的可选配置。

```
...
    lb_policy: RING_HASH
    ring_hash_lb_config:
        minimum_ring_size: 2000
        maximum_ring_size: 10000
...

```

## Maglev

与环形哈希算法一样，maglev（MAGLEV）算法也实现了对端点的一致性哈希。该算法产生一个查找表，允许在一个恒定的时间内找到一个项目。Maglev的设计是为了比环形哈希算法的查找速度更快，并且使用更少的内存。你可以在下面这篇文章中阅读更多关于它的内容 [Maglev: A Fast and Reliable Software Network Load Balancer](#)。

我们可以使用 `maglev_lb_config` 字段来设置 maglev 算法的可选配置。

```
...
    lb_policy: MAGLEV
    maglev_lb_config:
        table_size: 69997
...

```

默认的表大小是 65537，但它可以被设置为任何素数，只要它不大于 5000011。

## 原始目的地

原始目的地（ORIGINAL\_DESTINATION）是一个特殊用途的负载均衡器，只能与原始目的地集群一起使用。我们在谈到原始目的地集群类型时已经提到了原始目的地负载均衡器。

## 随机

顾名思义，随机（RANDOM）算法会挑选一个可用的随机端点。如果你没有配置主动健康检查策略，随机算法的表现比轮询算法更好。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 16:23:50

## 实验 7：断路器

在这个实验中，我们将演示如何使用断路器。我们将运行一个 Python HTTP 服务器和一个 Envoy 代理在它前面。要启动在 8000 端口监听的 Python 服务器，请运行：

```
python3 -m http.server 8000
```

接下来，我们将用下面的断路器创建 Envoy 配置：

```
...
  circuit_breakers:
    thresholds:
      max_connections: 20
      max_requests: 100
      max_pending_requests: 20
```

因此，如果我们超过了 20 个连接或 100 个请求或 20 个待处理请求，断路器就会断开。下面是完整的 Envoy 配置：

```

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: listener_http
              http_filters:
                - name: envoy.filters.http.router
                  route_config:
                    name: route
                    virtual_hosts:
                      - name: vh
                        domains: ["*"]
                        routes:
                          - match:
                              prefix: "/"
                            route:
                              cluster: python_server
            clusters:
              - name: python_server
                connect_timeout: 5s
                circuit_breakers:
                  thresholds:
                    max_connections: 20
                    max_requests: 100
                    max_pending_requests: 20
                load_assignment:
                  cluster_name: python_server
                  endpoints:
                    - lb_endpoints:
                        - endpoint:
                            address:
                              socket_address:
                                address: 127.0.0.1
                                port_value: 8000
                admin:
                  address:
                    socket_address:
                      address: 127.0.0.1
                      port_value: 9901

```

将上述配置保存为 `3-lab-1-circuit-breaker.yaml`，然后运行 Envoy 代理。

```
func-e run -c 3-lab-1-circuit-breaker.yaml
```

为了向代理发送多个并发请求，我们将使用一个名为 `hey` 的工具。默认情况下，`hey` 运行 50 个并发，发送 200 个请求，所以我们在请求 `http://localhost:1000`，甚至不需要传入任何参数。

```
hey http://localhost:10000
...
Status code distribution:
[200] 104 responses
[503] 96 responses
```

`hey` 将输出许多统计数字，但我们感兴趣的是状态码的分布。它显示我们在收到了 104 个 HTTP 200 响应，在 96 个 HTTP 503 响应——这就是断路器断开的地方。

我们可以使用 Envoy 的管理接口（运行在 9901 端口）来查看详细的指标，比如说：

```
...
envoy_cluster_upstream_cx_overflow{envoy_cluster_name="python_se
envoy_cluster_upstream_rq_pending_overflow{envoy_cluster_name="p
```

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:19:32

## 动态配置概述

在本章中，我们将学习如何使用动态配置来配置 Envoy 代理。到现在为止，我们一直在使用静态配置。本章将教我们如何在运行时从文件系统或通过网络使用发现服务为单个资源提供配置。

在本章结束时，你将了解静态和动态配置之间的区别。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 15:31:36

## 动态配置

Envoy 的强大功能之一是支持动态配置。到现在为止，我们一直在使用静态配置。我们使用 `static_resources` 字段将监听器、集群、路由和其他资源指定为静态资源。

当使用动态配置时，我们不需要重新启动 Envoy 进程就可以生效。相反，Envoy 通过从磁盘或网络上的文件读取配置，动态地重新加载配置。动态配置使用所谓的**发现服务 API**，指向配置的特定部分。这些 API 也被统称为 **xDS**。当使用 xDS 时，Envoy 调用外部基于 gRPC/REST 的配置供应商，这些供应商实现了发现服务 API 来检索配置。

外部基于 gRPC/REST 的配置提供者也被称为**控制平面**。当使用磁盘上的文件时，我们不需要控制平面。Envoy 提供了控制平面的 Golang 实现，但是 Java 和其他控制平面的实现也可以使用。

Envoy 内部有多个发现服务 API。所有这些在下表中都有描述。

发现服务名称	描述
监听器发现服务 (LDS)	使用 LDS, Envoy 可以在运行时发现监听器，包括所有的过滤器栈、HTTP 过滤器和对 RDS 的引用。
扩展配置发现服务 (ECDS)	使用 ECDS, Envoy 可以独立于监听器获取扩展配置（例如，HTTP 过滤器配置）。
路由发现服务 (RDS)	使用 RDS, 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。与 EDS 和 CDS 相结合，我们可以实现复杂的路由拓扑结构。
虚拟主机发现服务 (VHDS)	使用 VHDS 允许 Envoy 从路由配置中单独请求虚拟主机。当路由配置中有大量的虚拟主机时，就可以使用这个功能。
宽泛路由发现服务 (SRDS)	使用 SRDS，我们可以把路由表分解成多个部分。当我们有大的路由表时，就可以使用这个 API。
集群发现服务 (CDS)	使用 CDS, Envoy 可以发现上游集群。Envoy 将通过排空和重新连接所有现有的连接池来优雅地添加、更新或删除集群。Envoy 在初始化时不必知道所有的集群，因为我们可以在以后使用 CDS 配置它们。
端点发现服务 (EDS)	使用 EDS, Envoy 可以发现上游集群的成员。
秘密发现服务 (SDS)	使用 SDS, Envoy 可以为其监听器发现秘密（证书和私钥，TLS 会话密钥），并为对等的证书验证逻辑进行配置。
运行时发现服务 (RTDS)	使用 RTDS, Envoy 可以动态地发现运行时层。

### 聚合发现服务 (ADS)

表中的发现服务是独立的，有不同的 gRPC/REST 服务名称。使用聚合发现服务 (ADS)，我们可以使用一个单一的 gRPC 服务，在一个 gRPC 流中支持所有的资源类型（监听器、路由、集群...）。ADS 还能确保不同资源的更新顺序正确。请注意，ADS 只支持 gRPC。如果没有 ADS，我们就需要协调其他 gRPC 流来实现正确的更新顺序。

### 增量 gRPC xDS

每次我们发送资源更新时，我们必须包括所有的资源。例如，每次 RDS 更新必须包含每条路由。如果我们不包括一个路由，Envoy 会认为该路由已被删除。这样做更新会导致很高的带宽和计算成本，特别是当有大量的资源在网络上被发送时。Envoy 支持 xDS 的 delta 变体，我们可以只包括我们想添加 / 删除 / 更新的资源，以改善这种情况。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 15:30:49

## 来自文件系统的动态配置

动态提供配置另一种方式是通过指向文件系统上的文件。为了使动态配置发挥作用，我们需要在 `node` 字段下提供信息。如果我们可能有多个 Envoy 代理指向相同的配置文件，那么 `node` 字段是用来识别一个特定的 Envoy 实例。

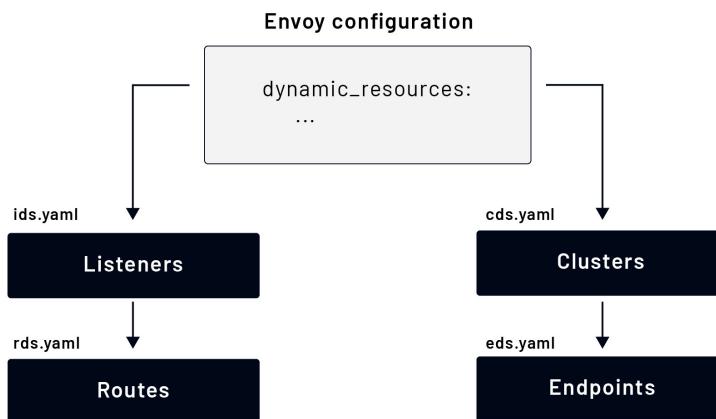


图 4.3.1：动态配置

为了指向动态资源，我们可以使用 `dynamic_resources` 字段来告诉 Envoy 在哪里可以找到特定资源的动态配置。例如：

```

node:
  cluster: my-cluster
  id: some-id

dynamic_resources:
  lds_config:
    path: /etc/envoy/lds.yaml
  cds_config:
    path: /etc/envoy/cds.yaml
  
```

上面的片段是一个有效的 Envoy 配置。如果我们把 LDS 和 CDS 作为静态资源来提供，它们的单独配置将非常相似。唯一不同的是，我们必须指定资源类型和版本信息。下面是 CDS 配置的一个片段。

```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.config.cluster.v3.Cluster
  name: instance_1
  connect_timeout: 5s
  load_assignment:
    cluster_name: instance_1
    endpoints:
      - lb_endpoints:
          - endpoint:
              address:
                socket_address:
                  address: 127.0.0.1
                  port_value: 3030
```

如果我们想使用 EDS 为集群提供端点，我们可以这样写上面的配置。

```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.config.cluster.v3.Cluster
  name: instance_1
  type: EDS
  eds_cluster_config:
    eds_config:
      path: /etc/envoy/eds.yaml
```

另外，注意我们已经把集群的类型设置为 `EDS`。EDS 的配置会是这样的。

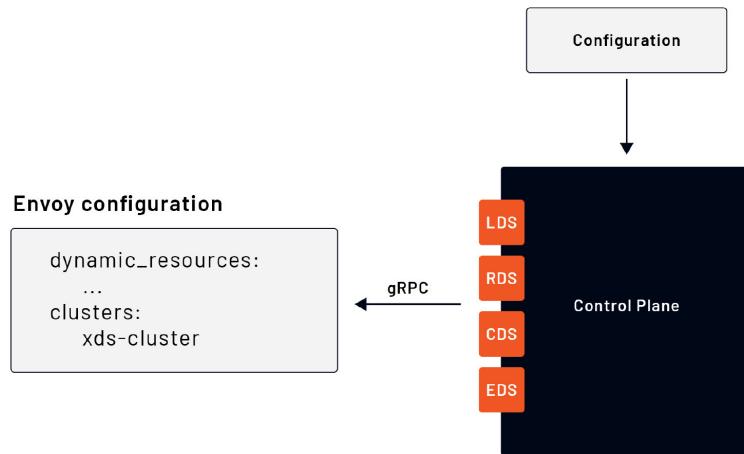
```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.config.endpoint.v3.ClusterL
  cluster_name: instance_1
  endpoints:
    - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: 127.0.0.1
                port_value: 3030
```

当任何一个文件被更新时，Envoy 会自动重新加载配置。如果配置无效，Envoy 会输出错误，但会保持现有（工作）配置的运行。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:28:48

## 来自控制平面的动态配置

使用控制平面来更新 Envoy 比使用文件系统的配置更复杂。我们必须创建自己的控制平面，实现发现服务接口。[这里](#)有一个 xDS 服务器实现的简单例子。这个例子显示了如何实现不同的发现服务，并运行 Envoy 连接的 gRPC 服务器的实例来检索配置。



Envoy 方面的动态配置与文件系统的配置类似。这一次，不同的是，我们提供了实现发现服务的 gRPC 服务器的位置。我们通过静态资源指定一个集群来做到这一点。

```
...
dynamic_resources:
  lds_config:
    resource_api_version: V3
    api_config_source:
      api_type: GRPC
      transport_api_version: V3
    grpc_services:
      - envoy_grpc:
          cluster_name: xds_cluster
  cds_config:
    resource_api_version: V3
    api_config_source:
      api_type: GRPC
      transport_api_version: V3
    grpc_services:
      - envoy_grpc:
          cluster_name: xds_cluster

static_resources:
  clusters:
    - name: xds_cluster
      type: STATIC
      load_assignment:
        cluster_name: xds_cluster
      endpoints:
        - lb_endpoints:
          - endpoint:
              address:
                socket_address:
                  address: 127.0.0.1
                  port_value: 9090
```

控制平面不需要在 Envoy 概念上操作。它可以抽象出配置。它也可以使用图形用户界面或不同的 YAML、XML 或任何其他配置文件来收集用户的输入。重要的是，无论高级别配置是如何进入控制平面的，它都需要被翻译成 Envoy xDS API。

例如，Istio 是 Envoy 代理机群的控制平面，可以通过各种自定义资源定义（VirtualService、Gateway、DestinationRule...）进行配置。除了上层配置外，在 Istio 中，Kubernetes 环境和集群内运行的服务也被用来作为生成 Envoy 配置的输入。上层配置和环境中发现的服务可以一起作为控制平面的输入。控制平面可以接受这些输入，将其转化为 Envoy 可读的配置，并通过 gRPC 将其发送给 Envoy 实例。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 15:51:19

## 实验 8：来自文件系统的动态配置

我们将在这个实验中创建一个动态的 Envoy 配置，并通过单独的配置文件配置监听器、集群、路由和端点。

让我们从最小的 Envoy 配置开始。

```
node:
  cluster: cluster-1
  id: envoy-instance-1
dynamic_resources:
  lds_config:
    path: ./lds.yaml
  cds_config:
    path: ./cds.yaml
admin:
  address:
    socket_address:
      address: 127.0.0.1
      port_value: 9901
  access_log:
    - name: envoy.access_loggers.file
      typed_config:
        "@type": type.googleapis.com/envoy.extensions.access_logge
```

将上述 YAML 保存到 `envoy-proxy-1.yaml` 文件。我们还需要创建空的（暂时的）`cds.yaml` 和 `lds.yaml` 文件。

```
touch {cds,lds}.yaml
```

我们现在可以用这个配置来运行 Envoy 代理：`func-e run -c envoy-proxy-1.yaml`。如果我们看一下生成的配置（比如 `localhost:9901/config_dump`），我们会发现它是空的，因为我们没有提供任何监听器或集群。

接下来让我们创建监听器和路由配置。

```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.config.listener.v3.Listener
  name: listener_0
  address:
    socket_address:
      address: 0.0.0.0
      port_value: 10000
  filter_chains:
- filters:
  - name: envoy.filters.network.http_connection_manager
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.network.listener_http
      stat_prefix: listener_http
    http_filters:
    - name: envoy.filters.http.router
      rds:
        route_config_name: route_config_1
        config_source:
          path: ./rds.yaml
```

创建一个空的 `rds.yaml` 文件（`touch rds.yaml`），并将上述 YAML 保存为 `lds.yaml`。因为 Envoy 只关注文件路径的移动，所以保存文件不会触发配置重载。为了触发重载，让我们覆盖 `lds.yaml` 文件。

```
mv lds.yaml tmp; mv tmp lds.yaml
```

上述命令触发了重载，我们应该从 Envoy 得到以下日志条目：

```
[2021-09-07 19:04:06.710][2113][info][upstream] [source/server/1]
```

同样，如果我们向 `localhost:10000` 发送请求，我们会得到一个 HTTP 404。

接下来让我们创建 `rds.yaml` 的内容。

```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.config.route.v3.RouteConfig
  name: route_config_1
  virtual_hosts:
- name: vh
  domains: ["*"]
  routes:
- match:
  prefix: "/headers"
  route:
    cluster: instance_1
```

强制重载：

```
mv rds.yaml tmp; mv tmp rds.yaml
```

最后，我们还需要对集群进行配置。在这之前，让我们运行一个 httpbin 容器。

```
docker run -d -p 5050:80 kennethreitz/httpbin
```

现在我们更新集群（`cds.yaml`）并强制重载（`mv cds.yaml tmp; mv tmp cds.yaml`）：

```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.config.cluster.v3.Cluster
  name: instance_1
  connect_timeout: 5s
  load_assignment:
    cluster_name: instance_1
    endpoints:
    - lb_endpoints:
      - endpoint:
        address:
          socket_address:
            address: 127.0.0.1
            port_value: 5050
```

当 Envoy 更新配置时，我们会得到以下日志条目：

```
$ [2021-09-07 19:09:15.582][2113][info][upstream] [source/common
```

现在我们可以提出请求并验证流量是否到达集群中定义的端点。

```
$ curl localhost:10000/headers
{
  "headers": {
    "Accept": "*/*",
    "Host": "localhost:10000",
    "User-Agent": "curl/7.64.0",
    "X-Envoy-Expected-Rq-Timeout-Ms": "15000"
  }
}
```

注意我们是如何将端点与集群配置在同一个文件中的。我们可以通过单独定义端点（`eds.yaml`）将两者分开。

让我们从创建 `eds.yaml` 文件开始：

```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.config.endpoint.v3.Cluster
  cluster_name: instance_1
  endpoints:
    - lb_endpoints:
      - endpoint:
        address:
          socket_address:
            address: 127.0.0.1
            port_value: 5050
```

将上述 YAML 保存为 `eds.yaml`。

为了使用这个端点文件，我们需要更新集群（`cds.yaml`）以读取 `eds.yaml` 中的端点。

```
version_info: "0"
resources:
- "@type": type.googleapis.com/envoy.config.cluster.v3.Cluster
  name: instance_1
  connect_timeout: 5s
  type: EDS
  eds_cluster_config:
    eds_config:
      path: ./eds.yaml
```

通过运行 `mv cds.yaml tmp; mv tmp cds.yaml` 来强制重载。Envoy 会重新加载配置，我们就可以像以前一样向 `localhost:10000/headers` 发送请求。现在的区别是，不同的配置在不同的文件中，可以分别更新。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-03-04 16:20:04

## 监听器子系统概述

在监听器子系统章节中，我们将学习 Envoy 代理的监听器子系统。我们将介绍过滤器、过滤器链匹配、以及不同的监听器过滤器。

在本章节结束时，你将了解监听器子系统的不同部分，并知道过滤器和过滤器链匹配是如何工作的。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 16:41:36

## 监听器过滤器

正如在介绍章节中提到的，监听器子系统处理下游或传入的请求处理。监听器子系统负责传入的请求和对客户端的响应路径。除了定义 Envoy 对传入请求进行“监听”的地址和端口外，我们还可以选择对每个监听器进行监听过滤器的配置。

不要把监听器过滤器和我们前面讨论的网络过滤器链和 L3/L4 过滤器混淆起来。Envoy 在处理网络级过滤器之前先处理监听器过滤器，如下图所示。

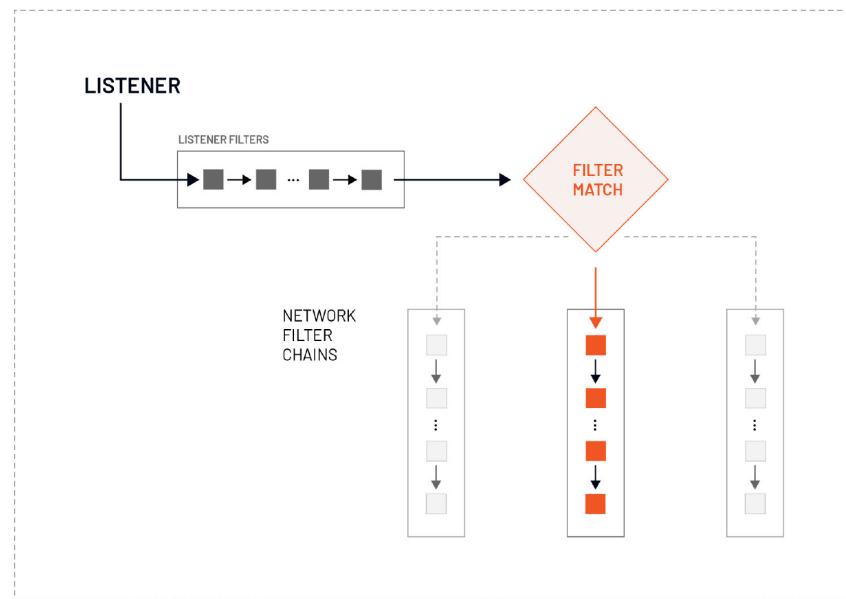


图 5.2.1：监听器过滤器

请注意，在没有任何监听器过滤器的情况下操作 Envoy 也是常见的。

Envoy 会在网络级过滤器之前处理监听器过滤器。我们可以在监听器过滤器中操作连接元数据，通常是为了影响后来的过滤器或集群如何处理连接。

监听器过滤器对新接受的套接字进行操作，并可以停止或随后继续执行进一步的过滤器。监听器过滤器的顺序很重要，因为 Envoy 在监听器接受套接字后，在创建连接前，会按顺序处理这些过滤器。

我们可以使用监听器过滤器的结果来进行过滤器匹配，并选择一个合适的网络过滤器链。例如，我们可以使用 HTTP 检查器监听器过滤器来确定 HTTP 协议（HTTP/1.1 或 HTTP/2）。基于这个结果，我们就可以选择并运行不同的网络过滤器链。

Copyright © 2017-2022 | 基于 CC 4.0 协议 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 16:54:36

## 过滤器链匹配

过滤器链匹配允许我们指定为监听器选择特定过滤器链的标准。

我们可以在配置中定义多个过滤器链，然后根据目标端口、服务器名称、协议和其他属性来选择和执行它们。例如，我们可以检查哪个主机名正在连接，然后选择不同的过滤器链。如果主机名 `hello.com` 连接，我们可以选择一个过滤器链来呈现该特定主机名的证书。

在 Envoy 开始过滤器匹配之前，它需要有一些由监听器过滤器从接收的数据包中提取的数据。之后，Envoy 要选择一个特定的过滤器链，必须满足所有的匹配条件。例如，如果我们对主机名和端口进行匹配，这两个值都需要匹配，Envoy 才能选择该过滤器链。

匹配顺序如下：

1. 目的地端口（当使用 `use_original_dst` 时）
2. 目的地 IP 地址
3. 服务器名称（TLS 协议的 SNI）
4. 传输协议
5. 应用协议（TLS 协议的 ALPN）
6. 直接连接的源 IP 地址（这在我们使用覆盖源地址的过滤器时与源 IP 地址不同，例如，代理协议监听器过滤器）
7. 来源类型（例如，任何、本地或外部网络）
8. 源 IP 地址
9. 来源端口

具体标准，如服务器名称 / SNI 或 IP 地址，也允许使用范围或通配符。如果在多个过滤器链中使用通配符标准，最具体的值将被匹配。

例如，对于 `www.hello.com` 从最具体到最不具体的匹配顺序是这样的。

1. `www.hello.com`
2. `*.hello.com`
3. `*.com`
4. 任何没有服务器名称标准的过滤器链

下面是一个例子，说明我们如何使用不同的属性配置过滤器链匹配。

```
filter_chains:
- filter_chain_match:
  server_names:
    - "*.hello.com"
  filters:
  ...
- filter_chain_match:
  source_prefix_ranges:
    - address_prefix: 192.0.0.1
      prefix_len: 32
  filters:
  ...
- filter_chain_match:
  transport_protocol: tls
  filters:
  ...
```

让我们假设一个 TLS 请求从 IP 地址进来，并且 `192.0.0.1` SNI 设置为 `v1.hello.com`。记住这个顺序，第一个满足所有条件的过滤器链匹配是服务器名称匹配（`v1.hello.com`）。因此，Envoy 会执行该匹配下的过滤器。

但是，如果请求是从 IP `192.0.0.1` 进来的，那就不是 TLS，而且 SNI 也不符合 `*.hello.com` 的要求。Envoy 将执行第二个过滤器链——与特定 IP 地址相匹配的那个。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 16:59:38

## HTTP 检查器监听器过滤器

### HTTP 检查器监听器过滤器

( `envoy.filters.listener.http_inspector` ) 允许我们检测应用协议是否是 HTTP。如果协议不是 HTTP，监听器过滤器将通过该数据包。

如果应用协议被确定为 HTTP，它也会检测相应的 HTTP 协议（如 HTTP/1.x 或 HTTP/2）。

我们可以使用过滤器链匹配中的 `application_protocols` 字段来检查 HTTP 检查过滤器的结果。

让我们考虑下面的片段。

```
...
  listener_filters:
    - name: envoy.filters.listener.http_inspector
      typed_config:
        "@type": type.googleapis.com/envoy.extensions.filters.li
      filter_chains:
        - filter_chain_match:
            application_protocols: ["h2"]
            filters:
              - name: my_http2_filter
              ...
        - filter_chain_match:
            application_protocols: ["http/1.1"]
            filters:
              - name: my_http1_filter
...

```

我们在 `listener_filters` 字段下添加了 `http_inspector` 过滤器来检查连接并确定应用协议。如果 HTTP 协议是 HTTP/2 (`h2c`)，Envoy 会匹配第一个网络过滤器链（以 `my_http2_filter` 开始）。

另外，如果下游的 HTTP 协议是 HTTP/1.1 (`http/1.1`)，Envoy 会匹配第二个过滤器链，并从名为 `my_http1_filter` 的过滤器开始运行过滤器链。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 17:02:06

## 原始目的地监听器过滤器

原始目的地过滤器（`envoy.filters.listener.original_dst`）会读取 `SO_ORIGINAL_DST` 套接字选项。当一个连接被 `iptables REDIRECT` 或 `TPROXY` 目标（如果 `transparent` 选项被设置）重定向时，这个选项被设置。该过滤器可用于与 `ORIGINAL_DST` 类型的集群连接。

当使用 `ORIGINAL_DST` 集群类型时，请求会被转发到由重定向元数据寻址的上游主机，而不做任何主机发现。因此，在集群中定义任何端点都是没有意义的，因为端点是从原始数据包中提取的，并不是由负载均衡器选择。

我们可以将 Envoy 作为一个通用代理，使用这种集群类型将所有请求转发到原始目的地。

要使用 `ORIGINAL_DST` 集群，流量需要通过 `iptables REDIRECT` 或 `TPROXY` 目标到达 Envoy。

```
...
  listener_filters:
    - name: envoy.filters.listener.original_dst
      typed_config:
        "@type": type.googleapis.com/envoy.extensions.filters.listening...
  ...
  clusters:
    - name: original_dst_cluster
      connect_timeout: 5s
      type: ORIGINAL_DST
      lb_policy: CLUSTER_PROVIDED
```

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 17:04:21

## 原始源监听器过滤器

原始源过滤器（`envoy.filters.listener.original_src`）在 Envoy 的上游（接收 Envoy 请求的主机）一侧复制了连接的下游（连接到 Envoy 的主机）的远程地址。

例如，如果我们用 `10.0.0.1` 发送请求给 Envoy 连接到上游，源 IP 是 `10.0.0.1`。这个地址是由代理协议过滤器决定的（接下来解释），或者它可以来自于可信的 HTTP Header。

```
- name: envoy.filters.listener.original_src
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.filters.liste
    mark: 100
```

该过滤器还允许我们在上游连接的套接字上设置 `SO_MARK` 选项。`SO_MARK` 选项用于标记通过套接字发送的每个数据包，并允许我们做基于标记的路由（我们可以在以后匹配标记）。

上面的片段将该标记设置为 100。使用这个标记，我们可以确保非本地地址在绑定到原始源地址时可以通过 Envoy 代理路由回来。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 17:10:01

## 代理协议监听器过滤器

代理协议监听器过滤器（`envoy.filters.listener.proxy_protocol`）增加了对 [HAProxy 代理协议](#) 的支持。

代理使用其 IP 堆栈连接到远程服务器，并丢失初始连接的源和目的地信息。PROXY 协议允许我们在不丢失客户端信息的情况下链接代理。该协议定义了一种在主 TCP 流之前通过 TCP 通信连接的元数据的方式。元数据包括源 IP 地址。

使用这个过滤器，Envoy 可以从 PROXY 协议中获取元数据，并将其传播到 `x-forwarded-for` 头中。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 17:11:51

## TLS 检查器监听器过滤器

[TLS 监听器](#)过滤器让我们可以检测到传输的是 TLS 还是明文。如果传输是 TLS，它会检测服务器名称指示（SNI）和 / 或客户端的应用层协议协商（ALPN）。

### 什么是 SNI？

SNI 或服务器名称指示（Server Name Indication）是对 TLS 协议的扩展，它告诉我们在 TLS 握手过程的开始，哪个主机名正在连接。我们可以使用 SNI 在同一个 IP 地址和端口上提供多个 HTTPS 服务（使用不同的证书）。如果客户端以主机名 "hello.com" 进行连接，服务器可以出示该主机名的证书。同样地，如果客户以 "example.com" 连接，服务器就会提供该证书。

### 什么是 ALPN？

ALPN 或应用层协议协商是对 TLS 协议的扩展，它允许应用层协商应该在安全连接上执行哪种协议，而无需进行额外的往返请求。使用 ALPN，我们可以确定客户端使用的是 HTTP/1.1 还是 HTTP/2。

我们可以使用 SNI 和 ALPN 值来匹配过滤器链，使用

`server_names`（对于 SNI）和 / 或 `application_protocols`（对于 ALPN）字段。

下面的片段显示了我们如何使用 `application_protocols` 和 `server_names` 来执行不同的过滤器链。

```
...
  listener_filters:
    - name: "envoy.filters.listener.tls_inspector"
      typed_config:
        "@type": type.googleapis.com/envoy.extensions.filters.
        filter_chains:
          - filter_chain_match:
              application_protocols: ["h2c"]
            filters:
              - name: some_filter
              ...
          - filter_chain_match:
              server_names: "something.hello.com"
            transport_socket:
              ...
            filters:
              - name: another_filter
              ...
...
```

## 实验 9：原始目的地过滤器

在这个实验中，我们将学习如何配置原始目的地过滤器。要做到这一点，我们需要启用 IP 转发，然后更新 `iptables` 规则，以捕获所有流量并将其重定向到 Envoy 正在监听的端口。

我们将使用一个 Linux 虚拟机，而不是 Google Cloud Shell。

让我们从启用 IP 转发开始。

```
# 启用IP转发功能  
sudo sysctl -w net.ipv4.ip_forward=1
```

接下来，我们需要配置 `iptables` 来捕获所有发送到 80 端口的流量，并将其重定向到 10000 端口。Envoy 代理将在 10000 端口进行监听。

首先，我们需要确定我们将在 `iptables` 命令中使用的网络接口名称。我们可以使用 `ip link show` 命令列出网络接口。例如：

```
jimmy@instance-1:~$ ip link show  
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNK  
2: ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc mq st
```

输出结果告诉我们，我们有两个网络接口：环回接口和一个名为 `ens4` 的接口。这是我们将在 `iptables` 命令中使用的接口名称。

```
# 捕获所有来自外部的80端口的流量并将其重定向到10000端口  
sudo iptables -t nat -A PREROUTING -i ens4 -p tcp --dport 80 -j
```

最后，我们将运行另一条 `iptables` 命令，防止从虚拟机发出请求时出现路由循环。设置这个规则将允许我们从虚拟机上运行 `curl tetrade.io`，并且仍然被重定向到 10000 端口。

```
# 使我们能够从同一个实例中运行curl（即防止路由循环）。  
sudo iptables -t nat -A OUTPUT -p tcp -m owner !--uid-owner root
```

在修改了 `iptables` 规则后，我们可以创建以下 Envoy 配置。

```

static_resources:
  listeners:
    - name: inbound
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      listener_filters:
        - name: envoy.filters.listener.original_dst
          typed_config:
            "@type": type.googleapis.com/envoy.extensions.filter
filter_chains:
  - filters:
    - name: envoy.filters.network.http_connection_manager
      typed_config:
        "@type": type.googleapis.com/envoy.extensions.filt
        stat_prefix: ingress_http
      access_log:
        - name: envoy.access_loggers.file
          typed_config:
            "@type": type.googleapis.com/envoy.extensions.
            path: ./envoy.log
    http_filters:
      - name: envoy.filters.http.router
    route_config:
      virtual_hosts:
        - name: proxy
          domains: ["*"]
          routes:
            - match:
              prefix: "/"
              route:
                cluster: original_dst_cluster
  clusters:
    - name: original_dst_cluster
      type: ORIGINAL_DST
      connect_timeout: 5s
      lb_policy: CLUSTER_PROVIDED
      original_dst_lb_config:
        use_http_header: true

```

这个配置看起来与我们已经看到的配置相似。我们在

`listenener_filters` 中添加了 `original_dst` 过滤器，启用了对一个文件的访问日志，并将所有流量路由到一个叫做 `original_dst_cluster` 的集群。这个集群的类型设置为 `ORIGINAL_DST`，将请求发送到原始目的地。

此外，我们将 `use_http_header` 字段设置为 `true`。当设置为 `true` 时，我们可以使用 `x-envoy-original-dst-host` 头来覆盖目标地址。请注意，这个标头默认情况下是没有经过处理的，所以启用它允许将流量路由到任意的主机，这可能会产生安全问题。我们在这里只是把它作为一个例子。

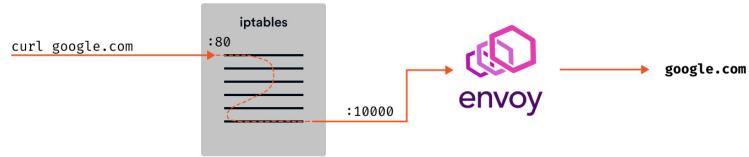


图 5.9.1：原始DST过滤器

对于透明代理的情况，这就是我们所需要的。我们不希望做任何解析。我们希望将请求代理到原始目的地。

将上述 YAML 保存为 `5-lab-1-originaldst.yaml`。

为了运行它，我们将使用 `func-e CLI`。让我们在虚拟机上安装 CLI。

```
curl https://func-e.io/install.sh | sudo bash -s -- -b /usr/loca
```

现在我们可以用我们创建的配置运行 Envoy 代理。

```
sudo func-e run -c 5-lab-1-originaldst.yaml
```

注意，在这种情况下，我们用 `sudo` 运行 `func-e`，所以我们可以用同一台机器来测试代理，防止路由循环（见第二条 `iptables` 规则）。

我们可以向 `tetratel.io` 发送一个请求，如果我们查看 `envoy.log` 文件，我们会看到以下条目。

```
[2021-07-07T21:22:57.294Z] "GET / HTTP/1.1" 301 - 0 227 34 34 "-
```

日志条目显示，`iptables` 捕获了该请求，并将其重定向到 Envoy 正在监听的端口 `10000`。然后，Envoy 将该请求代理到原来的目的地。

我们也可以从虚拟机的外部提出请求。从第二个终端：这一次，我们使用 `Google Cloud Shell`，而且我们不在虚拟机中。我们可以向虚拟机的 IP 地址发送请求，并提供 `x-envoy-original-dst-host` 头，我们希望 Envoy 将请求发送给该 IP 地址。

我在这个例子中使用 `google.com`。要获得 IP 地址，你可以运行 `nslookup google.com` 并使用该命令中的 IP 地址。

```
$ curl -H "x-envoy-original-dst-host: 74.125.199.139" [vm-ip-add<HTML><HEAD><meta http-equiv="content-type" content="text/html; c<TITLE>301 Moved</TITLE></HEAD><BODY><H1>301 Moved</H1>The document has moved<A href="http://www.google.com/">here</A>.</BODY></HTML>
```

你会注意到响应被代理到了 `google.com`。我们也可以检查虚拟机上的 `envoy.log` 来查看日志条目。

要清理 `iptables` 规则并禁用 IP 转发，请运行：

```
# 禁用IP转发功能
sudo sysctl -w net.ipv4.ip_forward=0

# 从nat表中删除所有规则
sudo iptables -t nat -F
```

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:27:59

## 实验 10：TLS 检查器过滤器

在这个实验中，我们将看一个例子，说明我们如何设置 Envoy，以便在一个 IP 地址上用不同的证书为多个网站服务。

我们将使用自签名的证书进行测试，但如果使用真实签名的证书，其过程是相同的。

使用 `openssl`，我们将在 `certs` 文件夹中为 `www.hello.com` 和 `www.example.com` 创建自签名证书。

```
$ mkdir certs && cd certs  
$ openssl req -nodes -new -x509 -keyout www_hello_com.key -out w  
$ openssl req -nodes -new -x509 -keyout www_example_com.key -out
```

对于每个通用名称，我们最终会有两个文件：私钥和证书（例如，`www_example_com.key` 和 `www_example_com.cert`）。

我们将为每个过滤链的匹配分别配置 `transport_socket` 字段。下面是一个如何定义 TLS 传输套接字并提供密钥和证书的片段。

```
...  
    transport_socket:  
        name: envoy.transport_sockets.tls  
        typed_config:  
            "@type": type.googleapis.com/envoy.extensions.transpor  
            common_tls_context:  
                tls_certificates:  
                    - certificate_chain:  
                        filename: certs/www_hello_com.cert  
                    private_key:  
                        filename: certs/www_hello_com.key  
...  
...
```

因为我们想根据 SNI 使用不同的证书，我们将在 TLS 监听器中添加一个 TLS 检查器过滤器，使用 `filter_chain_match` 和 `server_names` 字段来根据 SNI 进行匹配。

下面是两个过滤链匹配部分：注意每个过滤链匹配都有自己的 `transport_socket`，有指向证书和密钥文件的指针。

```
listener_filters:
  - name: "envoy.filters.listener.tls_inspector"
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.
filter_chains:
  - filter_chain_match:
    server_names: "www.example.com"
    filters:
      transport_socket:
        name: envoy.transport_sockets.tls
      ...
      http:filters:
      ...
  - filter_chain_match:
    server_names: "www.hello.com"
    filters:
      transport_socket:
        name: envoy.transport_sockets.tls
      ...
      http:filters:
      ...
  ...
```

你可以在 `5-lab-2-tls_match.yaml` 文件中找到完整的配置，用 `func-e run -c 5-lab-2-tls_match.yaml` 运行它。由于我们将只使用 `openssl` 进行连接，我们不需要集群端点的运行。

为了检查 SNI 匹配是否正常工作，我们可以使用 `openssl` 并连接到提供服务器名称的 Envoy 监听器。例如：

```
$ openssl s_client -connect 0.0.0.0:443 -servername www.example.
CONNECTED(00000003)
depth=0 C = US, ST = Washington, L = Seattle, O = Example LLC, O
verify error:num=18:self signed certificate
verify return:1
depth=0 C = US, ST = Washington, L = Seattle, O = Example LLC, O
verify return:1
---
Certificate chain
  0 s:C = US, ST = Washington, L = Seattle, O = Example LLC, OU =
    i:C = US, ST = Washington, L = Seattle, O = Example LLC, OU =
  ...
  ...
```

该命令将根据所提供的服务器名称返回正确的对等证书。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:33:12

## 实验 11：匹配传输和应用协议

在这个实验中，我们将学习如何使用 TLS 检查器过滤器来选择一个特定的过滤器链。单个过滤器链将根据 `transport_protocol` 和 `application_protocol` 将流量分配到不同的上游集群。

我们将为我们的上游主机使用 `mendhak/http-https-echo` Docker 镜像。这些容器可以被配置为监听 HTTP/HTTPS 并将响应回传。

我们将运行镜像的三个实例来代表非 TLS HTTP、TLS HTTP/1.1 和 TLS HTTP/2 协议。

```
# non-TLS HTTP
docker run -dit -p 8080:8080 -t mendhak/http-https-echo:18

# TLS HTTP1.1
docker run -dit -e HTTPS_PORT=443 -p 443:443 -t mendhak/http-htt

# TLS HTTP2
docker run -dit -e HTTPS_PORT=8443 -p 8443:8443 -t mendhak/http-
```

为了确保这三个容器都在运行，我们可以用 curl 发送几个请求，看看是否得到了回应。从输出中，我们还可以检查主机名是否与实际的 Docker 容器 ID 相符。

```
# non-TLS HTTP
$ curl http://localhost:8080
{
  "path": "/",
  "headers": {
    "host": "localhost:8080",
    "user-agent": "curl/7.64.0",
    "accept": "*/*"
  },
  "method": "GET",
  "body": "",
  "fresh": false,
  "hostname": "localhost",
  "ip": "::ffff:172.18.0.1",
  "ips": [],
  "protocol": "http",
  "query": {},
  "subdomains": [],
  "xhr": false,
  "os": {
    "hostname": "100db0dce742"
  },
  "connection": {}
}
```

注意在向其他两个容器发送请求时，我们将使用 `-k` 标志来告诉 curl 跳过对服务器 TLS 证书的验证。此外，我们可以使用 `--http1.1` 和 `--http2` 标志来发送 HTTP1.1 或 HTTP2 请求。

```

# HTTP1.1
$ curl -k --http1.1 https://localhost:443
{
  "path": "/",
  "headers": {
    "host": "localhost",
    "user-agent": "curl/7.64.0",
    "accept": "*/*"
  },
  "method": "GET",
  "body": "",
  "fresh": false,
  "hostname": "localhost",
  "ip": "::ffff:172.18.0.1",
  "ips": [],
  "protocol": "https",
  "query": {},
  "subdomains": [],
  "xhr": false,
  "os": {
    "hostname": "51afc40f7506"
  },
  "connection": {
    "servername": "localhost"
  }
}

$ curl -k --http2 https://localhost:8443
{
  "path": "/",
  "headers": {
    "host": "localhost:8443",
    "user-agent": "curl/7.64.0",
    "accept": "*/*"
  },
  "method": "GET",
  "body": "",
  "fresh": false,
  "hostname": "localhost",
  "ip": "::ffff:172.18.0.1",
  "ips": [],
  "protocol": "https",
  "query": {},
  "subdomains": [],
  "xhr": false,
  "os": {
    "hostname": "40e7143e6a55"
  },
  "connection": {
    "servername": "localhost"
  }
}

```

一旦我们验证了容器的正确运行，我们就可以创建 Envoy 配置。我们将使用 `tls_inspector` 和 `filter_chain_match` 字段来检查传输协议是否是 TLS，以及应用协议是否是 HTTP1.1 (`http/1.1`) 或 HTTP2 (`h2`)。基于这些信息，我们会有不同的集群，将流量转发到上游主机 (Docker 容器)。记住 HTTP 运行在端口 `8080`，TLS HTTP/1.1 运行在端口 `443`，TLS HTTP2 运行在端口 `8443`。

```
static_resources:
  listeners:
  - address:
      socket_address:
        address: 0.0.0.0
        port_value: 10000
    listener_filters:
    - name: "envoy.filters.listener.tls_inspector"
      typed_config:
        "@type": type.googleapis.com/envoy.extensions.filters.listener.TlsInspectorFilter
    filter_chains:
    - filter_chain_match:
        # Match TLS and HTTP2
        transport_protocol: tls
        application_protocols: [h2]
      filters:
      - name: envoy.filters.network.tcp_proxy
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.TcpProxyFilter
          cluster: service-tls-http2
          stat_prefix: https_passthrough
    - filter_chain_match:
        # Match TLS and HTTP1.1
        transport_protocol: tls
        application_protocols: [http/1.1]
      filters:
      - name: envoy.filters.network.tcp_proxy
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.TcpProxyFilter
          cluster: service-tls-http1.1
          stat_prefix: https_passthrough
    - filter_chain_match:
        # No matches here, go to HTTP upstream
      filters:
      - name: envoy.filters.network.tcp_proxy
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.TcpProxyFilter
          cluster: service-http
          stat_prefix: ingress_http
  clusters:
  - name: service-tls-http2
    type: STRICT_DNS
    lb_policy: ROUND_ROBIN
    load_assignment:
      cluster_name: service-tls-http2
      endpoints:
      - lb_endpoints:
          - endpoint:
              address:
                socket_address:
                  address: 127.0.0.1
                  port_value: 8443
    - name: service-tls-http1.1
      type: STRICT_DNS
      lb_policy: ROUND_ROBIN
      load_assignment:
        cluster_name: service-tls-http1.1
        endpoints:
        - lb_endpoints:
            - endpoint:
                address:
                  socket_address:
                    address: 127.0.0.1
                    port_value: 443
```

```
- name: service-http
  type: STRICT_DNS
  lb_policy: ROUND_ROBIN
  load_assignment:
    cluster_name: service-http
    endpoints:
      - lb_endpoints:
          - endpoint:
              address:
                socket_address:
                  address: 127.0.0.1
                  port_value: 8080
  admin:
    address:
      socket_address:
        address: 0.0.0.0
        port_value: 9901
```

将上述 YAML 保存为 `tls.yaml`，并使用 `func-e run -c tls.yaml` 运行它。

为了测试这一点，我们可以像以前一样发出类似的 curl 请求，并检查主机名是否与正在运行的 Docker 容器相符。

```
$ curl http://localhost:10000 | jq '.os.hostname'
"100db0dce742"

$ curl -k --http1.1 https://localhost:10000 | jq '.os.hostname'
"51afc40f7506"

$ curl -k --http2 https://localhost:10000 | jq '.os.hostname'
"40e7143e6a55"
```

另外，我们可以检查各个容器的日志，看看请求是否被正确发送。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:33:39

## 日志概览

在日志章节中，我们将学习 Envoy 中不同类型的日志和方法。

无论是作为流量网关还是作为服务网格中的 Sidecar，Envoy 都处于独特的位置，可以揭示你的网络中正在发生的事情。日志记录是了解系统状态的重要方式，无论是分析、审计还是故障排除。日志记录也有一个数量问题，有可能会泄露秘密。

在本章结束时，你将了解 Envoy 中存在哪些日志选项，包括如何对日志进行结构化和过滤，以及它们可以被写到哪里。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 17:29:29

## 访问日志

本节将为你讲解 Envoy 中的访问日志配置。

### 什么是访问日志？

每当你打开浏览器访问谷歌或其他网站时，另一边的服务器就会收集你的访问信息。具体来说，它在收集和储存你从服务器上请求的网页数据。在大多数情况下，这些数据包括来源（即主机信息）、请求网页的日期和时间、请求属性（方法、路径、Header、正文等）、服务器返回的状态、请求的大小等等。所有这些数据通常被存储在称为**访问日志（access log）**的文本文件中。

通常，来自网络服务器或代理的访问日志条目遵循标准化的通用日志格式。不同的代理和服务器可以使用自己的默认访问日志格式。Envoy 有其默认的日志格式。我们可以自定义默认格式，并配置它，使其以与其他服务器（如 Apache 或 NGINX）有相同的格式写出日志。有了相同的访问日志格式，我们就可以把不同的服务器放在一起使用，用一个工具把数据记录和分析结合起来。

本模块将解释访问日志在 Envoy 中是如何工作的，以及如何配置和定制。

### 捕获和读取访问日志

我们可以配置捕获任何向 Envoy 代理发出的访问请求，并将其写入所谓的访问日志。让我们看看几个访问日志条目的例子。

```
[2021-11-01T20:37:45.204Z] "GET / HTTP/1.1" 200 - 0 3 0 - "-" "c  
[2021-11-01T21:08:18.274Z] "POST /hello HTTP/1.1" 200 - 0 3 0 - "  
[2021-11-01T21:09:42.717Z] "GET /test HTTP/1.1" 404 NR 0 0 0 - "
```

输出包含三个不同的日志条目，并遵循相同的默认日志格式。默认的日志格式看起来像这样。

```
%START_TIME% "%REQ(:METHOD)% %REQ(X-ENVOY-ORIGINAL-PATH?:PATH)  
%RESPONSE_CODE% %RESPONSE_FLAGS% %BYTES_RECEIVED% %BYTES_SENT% %  
%RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)% \"%REQ(X-FORWARDED-FOR)%\"  
\"%REQ(X-REQUEST-ID)%\" \"%REQ(:AUTHORITY)%\" \"%UPSTREAM_HOST%\"
```

诸如 `%RESPONSE_FLAGS%`、`%REQ(:METHOD)%` 等值被称为**命令操作符（command operator）**。

### 命令操作符

命令操作符提取相关数据并插入到 TCP 和 HTTP 的日志条目中。如果这些值没有设置或不可用（例如，TCP 中的 RESPONSE\_CODE），日志将包含字符 `-`（或 JSON 日志的 `"-"`）。

每个命令操作符都以字符 `%` 开始和结束，例如，`%START_TIME%`。如果命令操作符接受任何参数，那么我们可以在括号内提供这些参数。例如，如果我们想使用 `START_TIME` 命令操作符只记录日、月、年，那么我们可以通过在括号中指定这些值来进行配置：`%START_TIME(%d-%m-%Y)%`。

让我们来看看不同的命令操作符。我们试图根据它们的共同属性将它们归入单独的表格。

命令操作符	描述	
START_TIME	请求开始时间，包括毫秒。	%ST/ ^%)%
PROTOCOL	协议 (HTTP/1.1、HTTP/2 或 HTTP/3)	%PF
RESPONSE_CODE	HTTP 响应代码。如果下游客户端断开连接，响应代码将被设置为 0。	%RE <sup>C</sup>
RESPONSE_CODE_DETAIL	关于 HTTP 响应的额外信息（例如，谁设置的以及为什么设置）。	RES <sup>I</sup>
CONNECTION_TERMINATION_DETAILS	提供关于 Envoy 因 L4 原因终止连接的额外信息。	%COT
ROUTE_NAME	路由的名称。	%R <sup>O</sup>
CONNECTION_ID	下游连接的一个标识符。它可以用来交叉引用多个日志汇聚中的 TCP 访问日志，或交叉引用同一连接的基于计时器的报告。该标识符在一个执行过程中很可能是唯一的，但在多个实例或重新启动之间可能会重复。	%COT

命令操作符	描述	
GRPC_STATUS	gRPC 状态代码，包括文本信息和一个数字。	%GRPC_STATUS
HOSTNAME	系统主机名。	%HOSTNAME
LOCAL_REPLY_BODY	被 Envoy 拒绝的请求的正文。	%LOCAL_REPLY_BODY
FILTER_CHAIN_NAME	下游连接的网络过滤器链名称。	%FILTER_CHAIN_NAME

## 大小

该组包含所有代表大小的命令操作符——从请求和响应头字节到接收和发送的字节。

命令操作符	描述	
REQUEST_HEADER_BYTES	请求头的未压缩字节。	%REQ
RESPONSE_HEADERS_BYTES	响应Header的未压缩字节数。	%RES
RESPONSE_TRAILERS_BYTES	响应trailer的未压缩字节。	%RES
BYTES_SENT	为HTTP发送的正文字节和为TCP发送的连接上的下游字节。	%BYT
BYTES RECEIVED	收到的正文字节数。	%BYT
UPSTREAM_WIRE_BYTES_SENT	由HTTP流向上游发送的总字节数。	%UPS
UPSTREAM_WIRE_BYTES RECEIVED	从上游HTTP流收到的字节总数。	%ups

命令操作符	描述	
UPSTREAM_HEADER_BYTES_SENT	由 HTTP 流向上游发送的头字节的数量。	%UPS
UPSTREAM_HEADER_BYTES_RECEIVED	HTTP 流从上游收到的头字节的数量。	%UPS
DOWNSTREAM_WIRE_BYTES_SENT	HTTP 流向下游发送的总字节数。	%down
DOWNSTREAM_WIRE_BYTES_RECEIVED	HTTP 流从下游收到的字节总数。	%down
DOWNSTREAM_HEADER_BYTES_SENT	由 HTTP 流向下游发送的头字节的数量。	%DOWN
DOWNSTREAM_HEADER_BYTES_RECEIVED	HTTP 流从下游收到的头字节的数量。	%down

## 时长

命令操作符	描述	示例
DURATION	从开始时间到最后一个字节输出，请求的总持续时间（以毫秒为单位）。	%DURATION%
REQUEST_DURATION	从开始时间到收到下游请求的最后一个字节，请求的总持续时间（以毫秒计）。	%REQUEST_DURATION%
RESPONSE_DURATION	从开始时间到从上游主机读取的第一个字节，请求的总持续时间（以毫秒计）。	RESPONSE_DURATION

命令操作符	描述	示例
RESPONSE_TX_DURATION	从上游主机读取的第一个字节到下游发送的最后一个字节，请求的总时间（以毫秒为单位）。	%RESPONSE_TX_DURATION%

## 响应标志

`RESPONSE_FLAGS` 命令操作符包含关于响应或连接的额外细节。下面的列表显示了 HTTP 和 TCP 连接的响应标志的值和它们的含义。

### HTTP 和 TCP

- UH: 除了 503 响应代码外，在一个上游集群中没有健康的上游主机。
- UF: 除了 503 响应代码外，还有上游连接失败。
- UO: 上游溢出（断路），此外还有 503 响应代码。
- NR: 除了 404 响应代码外，没有为给定的请求配置路由，或者没有匹配的下游连接的过滤器链。
- URX: 请求被拒绝是因为达到了上游重试限制（HTTP）或最大连接尝试（TCP）。
- NC: 未找到上游集群。
- DT: 当一个请求或连接超过 `max_connection_duration` 或 `max_downstream_connection_duration`。

### 仅限 HTTP

- DC: 下游连接终止。
- LH: 除了 503 响应代码，本地服务的健康检查请求失败。
- UT: 除 504 响应代码外的上行请求超时。
- LR: 除了 503 响应代码外，连接本地重置。
- UR: 除 503 响应代码外的上游远程复位。
- UC: 除 503 响应代码外的上游连接终止。
- DI: 请求处理被延迟了一段通过故障注入指定的时间。
- FI: 该请求被中止，并有一个通过故障注入指定的响应代码。

- RL: 除了 429 响应代码外, 该请求还被 HTTP 速率限制过滤器在本地进行了速率限制。
- UAEX: 该请求被外部授权服务拒绝。
- RLSE: 请求被拒绝, 因为速率限制服务中存在错误。
- IH: 该请求被拒绝, 因为除了 400 响应代码外, 它还为一个严格检查的头设置了一个无效的值。
- SI: 除 408 响应代码外, 流空闲超时。
- DPE: 下游请求有一个 HTTP 协议错误。
- UPE: 上游响应有一个 HTTP 协议错误。
- UMSDR: 上游请求达到最大流时长。
- OM: 过载管理器终止了该请求。

## 上游信息

命令操作符	
UPSTREAM_HOST	上游主机 URL 或
UPSTREAM_CLUSTER	上游主机所属的 <code>envoy.reloadable_configs</code> 。如果 <code>envoy.reloadable_configs</code> 被启用, 那么如果上游连接失败, 则会从该配置重新加载。
UPSTREAM_LOCAL_ADDRESS	上游连接的本地地址和端口。
UPSTREAM_TRANSPORT_FAILURE_REASON	如果由于传输套接字失败导致上游连接失败, 则返回失败原因。

## 下游信息

命令描述符
DOWNSTREAM_REMOTE_ADDRESS
DOWNSTREAM_REMOTE_ADDRESS_WITHOUT_PORT
DOWNSTREAM_DIRECT_REMOTE_ADDRESS
DOWNSTREAM_DIRECT_REMOTE_ADDRESS_WITHOUT_PORT
DOWNSTREAM_DIRECT_REMOTE_ADDRESS_WITHOUT_PORT
DOWNSTREAM_LOCAL_ADDRESS_WITHOUT_PORT
DOWNSTREAM_LOCAL_PORT

## Header 和 Trailer

`REQ`、`RESP` 和 `TRAILER` 命令操作符允许我们提取请求、响应和 `Trailer header` 的信息，并将其纳入日志。

`Trailer` 是一个响应首部，允许发送方在分块发送的消息后面添加额外的元信息，这些元信息可能是随着消息主体的发送动态生成的，比如消息的完整性校验，消息的数字签名，或者消息经过处理之后的最终状态等。

命令操作符	描述	示例
REQ (X?Y):Z	HTTP 请求头，其中 X 是主要的 HTTP 头，Y 是备选的 HTTP 头，Z 是一个可选的参数，表示最长为 Z 个字符的字符串截断。如果头信息 X 的值没有被设置，那么将使用请求头信息 Y。如果任何一个头都不存在，- 将出现在日志中。	%REQ(HELLO?BYE):5% 包括头信息 hello 的值。如果没有设置，则使用 Header bye 的值。它将值截断为 5 个字符。
RESP (X?Y):Z	与 REQ 相同，但取自 HTTP 响应头。	%RESP(HELLO?BYE):5% 包括头信息 hello 的值。如果没有设置，则使用头条 bye 的值。它将值截断为 5 个字符。
TRAILER (X?Y):Z	与 REQ 相同，但取自 HTTP 响应 Trailer。	%TRAILER(HELLO?BYE):5% 包括头信息 hello 的值。如果没有设置，则使用头条 bye 的值。它将该值截断为 5 个字符。

## 元数据

命令操作符	
DYNAMIC_METADATA(NAMESPACE:KEY*:Z)	动态元数据信息，是设置元数据时使用的一个可选的查询键。Z 是分隔的嵌套键。Z 是字符串截断，长度如， my_filter。{"json_object": {"\$": "my_filter", "key": "value"}, "key": "value"}。数据可以用 %DYNAMIC_METADATA 进行记录。要记录写 %DYNAMIC_METADATA
CLUSTER_METADATA(NAMESPACE:KEY*:Z)	上游集群元数据信息，是设置元数据时使用的一个可选的查询键。KEY 是命名空间中选择指定由 : 分隔的参数，表示字符串字符。
FILTER_STATE(KEY:F):Z	过滤器状态信息，是设置过滤器状态对象的。filter proto 将被记录为一个序列化的 proto 是为一个 protobuf 调用的参数，表示 FilterState 序列化。如果设置了 filter，那么过滤器状态对象将被序列化为 JSON 字符串。Z 是字符串的截断，长度

## TLS

命令操作符	描述
REQUESTED_SERVER_NAME	在 SSL 连接套接字上名称指示 (SNI) 设置值。
DOWNSTREAM_LOCAL_URI_SAN	用于建立下游 TLS 证书的 SAN 中存在的。
DOWNSTREAM_PEER_URI_SAN	用于建立下游 TLS 证书 SAN 中存在的。
DOWNSTREAM_LOCAL_SUBJECT	用于建立下游 TLS 证书中存在的主题。
DOWNSTREAM_PEER_SUBJECT	用于建立下游 TLS 证书中的主题。
DOWNSTREAM_PEER_ISSUER	用于建立下游 TLS 证书中存在的签发者。
DOWNSTREAM_TLS_SESSION_ID	已建立的下游 TLS 会话 ID。 %DOWNSTREAM_TLS_SESSION_ID
DOWNSTREAM_TLS_CIPHER	用于建立下游 TLS 会话集的 OpenSSL 名称。
DOWNSTREAM_TLS_VERSION	用于建立下游 TLS 会话版本 (TLSv1.2 或 TLSv1.3)。
DOWNSTREAM_PEER_FINGERPRINT_256	用于建立下游 TLS 证书的十六进制编码指纹。
DOWNSTREAM_PEER_FINGERPRINT_1	用于建立下游 TLS 证书的十六进制编码指纹。
DOWNSTREAM_PEER_SERIAL	用于建立下游 TLS 证书的序列号。
DOWNSTREAM_PEER_CERT	用于建立下游 TLS 安全编码的 PEM 格式证书。
DOWNSTREAM_PEER_CERT_V_START	用于建立下游 TLS 证书的有效期开始日。
DOWNSTREAM_PEER_CERT_V_END	用于建立下游 TLS 证书的有效期结束日。

Copyright © 2017-2022 | 基于 CC 4.0 协议 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 17:48:02

## 配置访问记录器

我们可以在 HTTP 或 TCP 过滤器级别和监听器级别上配置访问记录器。

我们还可以配置多个具有不同日志格式和日志沉积的访问日志。**日志沉积**

**(log sink)** 是一个抽象的术语，指的是日志写入的位置，例如，写入控制台 (stdout、stderr) 、文件或网络服务。

要配置多个访问日志的情况是，我们想在控制台（标准输出）中看到高级信息，并将完整的请求细节写入磁盘上的文件。用于配置访问记录器的字段被称为 `access_log`。

让我们看看在 HTTP 连接管理器 (HCM) 层面上启用访问日志到标准输出 (`StdoutAccessLog`) 的例子。

```
- filters:  
  - name: envoy.filters.network.http_connection_manager  
    typed_config:  
      "@type": type.googleapis.com/envoy.extensions.filters.network.  
      stat_prefix: ingress_http  
      access_log:  
        - name: envoy.access_loggers.stdout  
          typed_config:  
            "@type": type.googleapis.com/envoy.extensions.access_1
```

Envoy 的目标是拥有可移植和可扩展的配置：类型化的配置。这样做的一个副作用是配置的名字很冗长。例如，为了启用访问日志，我们找到 HTTP 配置类型的名称，然后找到对应于控制台的类型 (`StdoutAccessLog`)。

`StdoutAccessLog` 配置将日志条目写到标准输出（控制台）。其他支持的访问日志沉积有以下几种：

- 文件 (`FileAccessLog`)
- gRPC (`HttpGrpcAccessLogConfig` 和 `TcpGrpcAccessLogConfig`)
- 标准错误 (`StderrAccessLog`)
- Wasm (`WasmAccessLog`)
- Open Telemetry

文件访问日志允许我们将日志条目写到配置中指定的文件中。例如：

```

- filters:
  - name: envoy.filters.network.http_connection_manager
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.network
      stat_prefix: ingress_http
      access_log:
        - name: envoy.access_loggers.file
          typed_config:
            "@type": type.googleapis.com/envoy.extensions.access_log
            path: ./envoy-access-logs.log

```

注意名称（`envoy.access_loggers.file`）和类型（`file.v3.FileAccessLog`）的变化。此外，我们还提供了我们希望 Envoy 存储访问日志的路径。

gRPC 访问日志沉积将日志发送到 HTTP 或 TCP gRPC 日志服务。为了使用 gRPC 日志沉积，我们必须建立一个 gRPC 服务器，其端点要实现 `MetricsService`，特别是 `StreamMetrics` 函数。然后，Envoy 可以连接到 gRPC 服务器并将日志发送给它。

在此之前，我们提到了默认的访问日志格式，它是由不同的命令操作符组成的。

```

[%start_time%] "%req(:method)%req(x-envoy-original-path?:path)%%
%response_code% %response_flags% %bytes_received% %bytes_sent% %
%resp(x-envoy-upstream-service-time)% \"%req(x-forwarded-for)%\" \"%req(x-request-id)%\" \"%req(:authority)%\" \"%upstream_host%"

```

日志条目的格式是可配置的，可以使用 `log_format` 字段进行修改。使用 `log_format`，我们可以配置日志条目包括哪些值，并指定我们是否需要纯文本或 JSON 格式的日志。

例如，我们只想记录开始时间、响应代码和用户代理。我们会这样配置它。

```

- filters:
  - name: envoy.filters.network.http_connection_manager
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.network
      stat_prefix: ingress_http
      access_log:
        - name: envoy.access_loggers.stdout
          typed_config:
            "@type": type.googleapis.com/envoy.extensions.access_log
            log_format:
              text_format_source:
                inline_string: "%START_TIME% %RESPONSE_CODE% %REQ(

```

一个使用上述格式的日志条目样本看起来是这样的：

```
2021-11-01T21:32:27.170Z 404 curl/7.64.0
```

同样，如果我们希望日志是 JSON 等结构化格式，我们也可以不提供文本格式，而是设置 JSON 格式字符串。

为了使用 JSON 格式，我们必须提供一个格式字典，而不是像纯文本格式那样提供一个单一的字符串。

下面是一个使用相同的日志格式的例子，但用 JSON 写日志条目来代替。

```
- filter:
  - name: envoy.filters.network.http_connection_manager
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.net
      stat_prefix: ingress_http
      access_log:
        - name: envoy.access_loggers.stdout
          typed_config:
            "@type": type.googleapis.com/envoy.extensions.access_l
            log_format:
              json_format:
                start_time: "%START_TIME%"
                response_code: "%response_code%"
                user_agent: "%req(user-agent)%"
```

上述片段将产生以下日志条目。

```
{"user_agent": "curl/7.64.0", "response_code": 404, "start_time": "20
```

某些命令操作符，如 `FILTER_STATE` 或 `DYNAMIC_METADATA`，可能产生嵌套的 JSON 日志条目。

日志格式也可以使用通过 `formatters` 字段指定的 formatter 插件。当前版本中有两个已知的格式化插件：元数据

`(envoy.formatter.metadata)` 和无查询请求  
`(envoy.formatter.req_without_query)` 扩展。

元数据格式化扩展实现了 METADATA 命令操作符，允许我们输出不同类型的元数据（DYNAMIC、CLUSTER 或 ROUTE）。

同样，`req_without_query` 格式化允许我们使用 `REQ_WITHOUT_QUERY` 命令操作符，其工作方式与 `REQ` 命令操作符相同，但会删除查询字符串。该命令操作符用于避免将任何敏感信息记录到访问日志中。

下面是一个如何提供格式化器以及如何在 `inline_string` 中使用它的例子。

```
- filters:
  - name: envoy.filters.network.http_connection_manager
    typed_config:
      "@type": type.googleapis.com/envoy.extensions.filters.network.ingress_http
      stat_prefix: ingress_http
      access_log:
        - name: envoy.access_loggers.stdout
          typed_config:
            "@type": type.googleapis.com/envoy.extensions.access_log.formatters:
              - name: envoy.formatter.req_without_query
                typed_config:
                  "@type": type.googleapis.com/envoy.extensions.formatters.access_log_without_query
```

上述配置中的这个请求 `curl localhost:10000/?hello=1234` 会产生一个不包括查询参数（`hello=1234`）的日志条目。

```
[2021-11-01t21:48:55.941z] get / http/1.1
```

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 18:02:04

## 访问日志过滤

Envoy 中访问日志的另一个特点是可以指定过滤器，决定是否需要写入访问日志。例如，我们可以有一个访问日志过滤器，只记录 500 状态代码，只记录超过 5 秒的请求，等等。下表显示了支持的访问日志过滤器。

访问日志过滤器名称	描述
<code>status_code_filter</code>	对状态代码值进行过滤。
<code>duration_filter</code>	对总的请求持续时间进行过滤，单位为毫秒。
<code>not_health_check_filter</code>	对非健康检查请求的过滤。
<code>traceable_filter</code>	对可追踪的请求进行过滤。
<code>runtimetime_filter</code>	对请求进行随机抽样的过滤器。
<code>and_filter</code>	对过滤器列表中每个过滤器的结果进行逻辑 "和" 运算。过滤器是按顺序进行评估的。
<code>or_filter</code>	对过滤器列表中每个过滤器的结果进行逻辑 "或" 运算。过滤器是按顺序进行评估的。
<code>header_filter</code>	根据请求头的存在或值来过滤请求。
<code>response_flag_filter</code>	过滤那些收到设置了 Envoy 响应标志的响应的请求。
<code>grpc_status_filter</code>	根据响应状态过滤 gRPC 请求。
<code>extension_filter</code>	使用一个在运行时静态注册的扩展过滤器。
<code>metadata_filter</code>	基于匹配的动态元数据的过滤器。

每个过滤器都有不同的属性，我们可以选择设置。这里有一个片段，显示了如何使用状态代码、Header 和一个 `and` 过滤器。

```
...
access_log:
- name: envoy.access_loggers.stdout
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.access_loggers
  filter:
    and_filter:
      filters:
        header_filter:
          header:
            name: ":method"
            string_match:
              exact: "GET"
        status_code_filter:
          comparison:
            op: GE
            value:
              default_value: 400
...
...
```

上面的片段为所有响应代码大于或等于 400 的 GET 请求写了一条日志条目到标准输出。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 18:04:11

## Envoy 组件日志

到目前为止，我们已经谈到了向 Envoy 发送请求时产生的日志。然而，Envoy 也会在启动时和执行过程中产生日志。

我们可以在每次运行 Envoy 时看到 Envoy 组件的日志。

```
...
[2021-11-03 17:22:43.361][1678][info][main] [source/server/serve
[2021-11-03 17:22:43.361][1678][info][main] [source/server/serve
[2021-11-03 17:22:43.361][1678][info][main] [source/server/serve
...
...
```

组件日志的默认格式字符串是 `[%Y-%m-%d %T.%e][%t][%l][%n] [%g:%#]`  
`%v`。格式字符串的第一部分代表日期和时间，然后是线程 ID（`%t`）、消息的日志级别（`%l`）、记录器名称（`%n`）、源文件的相对路径和行号（`%g:%#`），以及实际的日志消息（`%v`）。

在启动 Envoy 时，我们可以使用 `--log-format` 命令行选项来定制格式。例如，如果我们想记录时间记录器名称、源函数名称和日志信息，那么我们可以这样写格式字符串：`[%T.%e][%n][%! ] %v`。

然后，在启动 Envoy 时，我们可以设置格式字符串，如下所示。

```
func-e run -c someconfig.yaml --log-format '[%T.%e][%n][%! ] %v'
```

如果我们使用格式字符串，日志条目看起来像这样：

```
[17:43:15.963][main][initialize] response trailer map: 160 byte
[17:43:15.965][main][createRuntime] runtime: {}
[17:43:15.965][main][initialize] No admin address given, so no admin endpoint
[17:43:15.966][config][initializeTracers] loading tracing config
[17:43:15.966][config][initialize] loading 0 static secret(s)
[17:43:15.966][config][initialize] loading 0 cluster(s)
[17:43:15.966][config][initialize] loading 1 listener(s)
[17:43:15.969][config][initializeStatsConfig] loading stats config
[17:43:15.969][runtime][onRtdsReady] RTDS has finished initialization
[17:43:15.969][upstream][maybeFinishInitialize] cm init: all clusters initialized
[17:43:15.969][main][onRuntimeReady] there is no configured limit
[17:43:15.970][main][operator()] all clusters initialized. initialize
[17:43:15.970][config][startWorkers] all dependencies initialized
[17:43:15.971][main][run] starting main dispatch loop
```

Envoy 具有多个日志记录器，对于每个日志记录器（例如 `main`、`config`、`http...`），我们可以控制日志记录级别（`info`、`debug`、`trace`）。如果我们启用 Envoy 管理界面并向 `/logging` 路径发送请求，就可以查看所有活动的日志记录器的名称。另一种查看所有可用日志的方法是通过[源码](#)。

下面是 `/logging` 终端的默认输出的样子。

```
active loggers:  
  admin: info  
  alternate_protocols_cache: info  
  aws: info  
  assert: info  
  backtrace: info  
  cache_filter: info  
  client: info  
  config: info  
  connection: info  
  conn_handler: info  
  decompression: info  
  dns: info  
  dubbo: info  
  envoy_bug: info  
  ext_authz: info  
  rocketmq: info  
  file: info  
  filter: info  
  forward_proxy: info  
  grpc: info  
  hc: info  
  health_checker: info  
  http: info  
  http2: info  
  hystrix: info  
  init: info  
  io: info  
  jwt: info  
  kafka: info  
  key_value_store: info  
  lua: info  
  main: info  
  matcher: info  
  misc: info  
  mongo: info  
  quic: info  
  quic_stream: info  
  pool: info  
  rbac: info  
  redis: info  
  router: info  
  runtime: info  
  stats: info  
  secret: info  
  tap: info  
  testing: info  
  thrift: info  
  tracing: info  
  upstream: info  
  udp: info  
  wasm: info
```

请注意，每个日志记录器的默认日志级别都被设置为 `info`。其他的日志级别有以下几种：

- trace
- debug
- info

- warning/warn
- error
- critical
- off

为了配置日志级别，我们可以使用 `--log-level` 选项或 `--component-log-level` 来分别控制每个组件的日志级别。组件的日志级别可以用 `log_name:log_level` 格式来写。如果我们要为多个组件设置日志级别，那么就用逗号来分隔它们。例如：  
如：`upstream:critical,secret:error,router:trace`。

例如，要将 `main` 日志级别设置为 `trace`，`config` 日志级别设置为 `error`，并关闭所有其他日志记录器，我们可以键入以下内容。

```
func-e run -c someconfig.yaml --log-level off --component-log-le
```

默认情况下，所有 Envoy 应用程序的日志都写到标准错误（stderr）。要改变这一点，我们可以使用 `--log-path` 选项提供一个输出文件。

```
func-e run -c someconfig.yaml --log-path app-logs.log
```

在其中一个试验中，我们还将展示如何配置 Envoy，以便将应用日志写入谷歌云操作套件（以前称为 Stackdriver）。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all rights reserved. Updated at 2022-02-28 18:06:04

## 实验 12：使用日志过滤器

在这个实验中，我们将学习使用日志过滤器，根据一些请求属性，只记录某些请求。

让我们想出几个我们想用过滤器来实现的日志要求。

- 将所有带有 HTTP 404 状态码的请求记录到一个名为 `not_found.log` 的日志文件中。
- 将所有带有 `env=debug` 头值的请求记录到一个名为 `debug.log` 的日志文件中。
- 将所有 POST 请求记录到标准输出

基于这些要求，我们将有两个访问记录器记录到文件（`not_found.log` 和 `debug.log`）和一个写到 `stdout` 的访问记录器。

`access_log` 字段是一个数组，我们可以在它下面定义多个记录器。在各个日志记录器里面，我们可以使用 `filter` 字段来指定何时将字符串写到日志中。

对于第一个要求，我们将使用状态码过滤器，而对于第二个要求，我们将使用 Header 过滤器。下面是配置的样子。

```

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: ingress_http
            access_log:
              - name: envoy.access_loggers.file
                typed_config:
                  "@type": type.googleapis.com/envoy.extensions.acce
                    path: ./debug.log
              filter:
                header_filter:
                  header:
                    name: env
                    string_match:
                      exact: debug
              - name: envoy.access_loggers.file
                typed_config:
                  "@type": type.googleapis.com/envoy.extensions.acce
                    path: ./not_found.log
              filter:
                status_code_filter:
                  comparison:
                    value:
                      default_value: 404
                      runtime_key: ingress_http_status_code_filter
            - name: envoy.access_loggers.stdout
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.acce
              filter:
                header_filter:
                  header:
                    name: ":method"
                    string_match:
                      exact: POST
  http_filters:
    - name: envoy.filters.http.router
      route_config:
        name: my_first_route
        virtual_hosts:
          - name: direct_response_service
            domains: ["*"]
            routes:
              - match:
                  prefix: "/404"
                  direct_response:
                    status: 404
                    body:
                      inline_string: "404"
              - match:
                  prefix: "/"
                  direct_response:
                    status: 200
                    body:
                      inline_string: "200"

```

将上述 YAML 保存为 `6-lab-1-logging-filters.yaml` 并在后台运行 Envoy。

```
func-e run -c 6-lab-1-logging-filters.yaml &
```

在 Envoy 运行的情况下，如果我们发送一个请求到 `http://localhost:10000`，那么我们会注意到没有任何东西被写入标准输出或日志文件。

接下来让我们试试 POST 请求。

```
$ curl -X POST localhost:10000
[2021-11-03T21:52:36.398Z] "POST / HTTP/1.1" 200 - 0 3 0 - "-" "
```

你会注意到，日志条目被写到了配置中定义的标准输出。

接下来，让我们发送一个头信息 `env: debug` 与以下请求。

```
curl -H "env: debug" localhost:10000
```

像第一个例子一样，没有任何东西会被写入标准输出（这不是一个 POST 请求）。然而，如果我们在 `debug.log` 文件中查看，那么我们会看到日志条目。

```
$ cat debug.log
[2021-11-03T21:54:49.357Z] "GET / HTTP/1.1" 200 - 0 3 0 - "-" "c
```

同样地，让我们向 `/404` 发送一个请求，并查看 `not_found.log` 文件。

```
$ curl localhost:10000/404
404

$ cat not_found.log
[2021-11-03T21:55:37.891Z] "GET /404 HTTP/1.1" 404 - 0 3 0 - "-" "
```

在满足多个过滤条件的情况下（例如，我们有一个 POST 请求，我们将请求发送到 `/404`），在这种情况下，日志将被写入标准输出和 `not_found.log`。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:34:02

## 实验 13：使用 gRPC 访问日志服务 (ALS) 记录日志

本实验涵盖了如何配置 Envoy 使用独立的 gRPC 访问日志服务 (ALS)。我们将使用一个[基本的 gRPC 服务器](#)，它实现了 `StreamAccessLogs` 函数，并将收到的日志从 Envoy 输出到标准输出。

让我们先把 ALS 服务器作为一个 Docker 容器来运行。

```
docker run -dit -p 5000:5000 gcr.io/tetratelabs/envoy-als:0.1.0
```

ALS 服务器默认监听端口为 5000，所以如果我们看一下 Docker 容器的日志，它们应该类似于下面的内容。

```
$ docker logs [container-id]
Creating new ALS server
2021/11/05 20:24:03 Listening on :5000
```

输出告诉我们，ALS 正在监听 5000 端口。

在 Envoy 配置中，有两个要求需要配置。首先是使用 `access_log` 字段的访问日志和一个名为 `HttpGrpcAccessLogConfig` 的日志类型。其次，在访问日志配置中，我们必须引用 gRPC 服务器。为此定义一个 Envoy 集群。

下面是配置记录器并指向名为 `grpc_als_cluster` 的 Envoy 集群的片段。

```
...
access_log:
- name: envoy.access_loggers.http_grpc
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.access_loggers
    common_config:
      log_name: "mygrpclog"
      transport_api_version: V3
      grpc_service:
        envoy_grpc:
          cluster_name: grpc_als_cluster
...
...
```

下一个片段是集群配置，在这一点上我们应该已经很熟悉了。在我们的例子中，我们在同一台机器上运行 gRPC 服务器，端口为 5000。

```
...
clusters:
- name: grpc_als_cluster
  connect_timeout: 5s
  type: STRICT_DNS
  http2_protocol_options: {}
  load_assignment:
    cluster_name: grpc_als_cluster
    endpoints:
      - lb_endpoints:
        - endpoint:
          address:
            socket_address:
              address: 127.0.0.1
              port_value: 5000
...
...
```

让我们把这两块放在一起，得出一个使用 gRPC 访问日志服务的 Envoy 配置示例。

```

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: ingress_http
              access_log:
                - name: envoy.access_loggers.http_grpc
                  typed_config:
                    "@type": type.googleapis.com/envoy.extensions.acce
                      common_config:
                        log_name: "mygrpclog"
                        transport_api_version: V3
                      grpc_service:
                        envoy_grpc:
                          cluster_name: grpc_als_cluster
            http_filters:
              - name: envoy.filters.http.router
                route_config:
                  name: my_first_route
                  virtual_hosts:
                    - name: direct_response_service
                      domains: ["*"]
                      routes:
                        - match:
                            prefix: "/404"
                            direct_response:
                              status: 404
                              body:
                                inline_string: "404"
                        - match:
                            prefix: "/"
                            direct_response:
                              status: 200
                              body:
                                inline_string: "200"
            clusters:
              - name: grpc_als_cluster
                connect_timeout: 5s
                type: STRICT_DNS
                http2_protocol_options: {}
                load_assignment:
                  cluster_name: grpc_als_cluster
                  endpoints:
                    - lb_endpoints:
                      - endpoint:
                          address:
                            socket_address:
                              address: 127.0.0.1
                              port_value: 5000

```

将上述 YAML 保存为 `6-lab-2-grpc-als.yaml`，并使用 `func-e` 启动 Envoy 代理。

```
func-e run -c 6-lab-2-grpc-als.yaml &
```

我们正在后台运行 Docker 容器和 Envoy，所以我们现在可以用 `curl` 向 Envoy 代理发送几个请求。

```
$ curl localhost:10000  
200
```

代理的回应是 `200`，因为那是我们在配置中所定义的。你会注意到没有任何日志输出到标准输出，这是预期的。

要看到这些日志，我们必须看一下 Docker 容器的日志。你可以使用 `docker ps` 来获取容器 ID，然后运行 `logs` 命令。

```
$ docker logs 96f  
Creating new ALS server  
2021/11/05 20:24:03 Listening on :5000  
2021/11/05 20:33:52 Received value  
2021/11/05 20:33:52 {"identifier": {"node": {"userAgentName": "envo  
...  
...
```

然后我们会注意到从 Envoy 代理发送到我们 gRPC 服务器的日志条目。gRPC 服务器中的代码很简单，只是将收到的值转换为一个字符串并输出。

下面是完整的 `StreamAccessLogs` 函数的样子。

```
func (s *server) StreamAccessLogs(stream v3.AccessLogService_Str  
for {  
    in, err := stream.Recv()  
    log.Println("Received value")  
    if err == io.EOF {  
        return nil  
    }  
    if err != nil {  
        return err  
    }  
    str, _ := s.marshaler.MarshalToString(in)  
    log.Println(str)  
}
```

在这一点上，我们可以从收到的数据流中解析具体的数值，并决定如何格式化它们以及将它们发送到哪里。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:34:28

## 实验 14：将 Envoy 的日志发送到 Google Cloud Logging

在这个实验中，我们将学习如何将 Envoy 应用日志发送到 Google Cloud Logging。我们将在 GCP 中运行的虚拟机（VM）实例上运行 Envoy，配置 Envoy 实例，将应用日志发送到 GCP 中的云日志。配置 Envoy 实例将允许我们在日志资源管理器中查看 Envoy 的日志，获得日志分析，并使用其他谷歌云功能。

为了使用谷歌云的日志收集、分析和其他工具，我们需要安装云日志代理（Ops Agent）。

在这个演示中，我们将在一个单独的虚拟机上安装 Ops 代理。另外，请注意，其他云供应商可能使用不同的日志工具和服务。

### 安装 Ops 代理

在 Google Cloud，在你的地区创建一个新的虚拟机实例。一旦创建了虚拟机，我们就可以通过 SSH 进入该实例并安装 Ops 代理。

在虚拟机实例中，运行以下命令来安装 Ops 代理。

```
curl -sS0 https://dl.google.com/cloudagents/add-google-cloud-ops
sudo bash add-google-cloud-ops-agent-repo.sh --also-install
```

安装 Ops 代理的另一个选择是按照以下步骤进行。

1. 从 GCP 的导航页面，选择**监测**。
2. 在“**监控**”导航页面，选择“**仪表盘**”。
3. 在仪表板表中，找到并点击**虚拟机实例**。
4. 选择一个没有安装代理的实例旁边的复选框（例如，代理栏显示未检测到）。
5. 点击**安装代理**按钮，在打开的窗口中，点击**在 Cloud Shell 中运行**按钮，开始安装。

安装代理的命令将在 Cloud Shell 中打开。你需要做的最后一件事是按回车键开始安装。

下面是成功安装的命令和输出在 Cloud Shell 中的样子。

```
$ :> agents_to_install.csv && \
→ echo '"projects/envoy-project/zones/us-west1-a/instances/envoy
→ curl -sSO https://dl.google.com/cloudagents/mass-provision-goo
→ python3 mass-provision-google-cloud-ops-agents.py --file agent
2021-11-03T19:04:31.577710Z Processing instance: projects/peterj
-----Getting output-----
Progress: |=====
Instance: projects/envoy-project/zones/us-west1-a/instances/envo

SUCCEEDED: [1/1] (100.0%)
FAILED: [0/1] (0.0%)
COMPLETED: [1/1] (100.0%)

See script log file: ./google_cloud_ops_agent_provisioning/20211
```

随着安装的进展，虚拟机实例仪表板中的代理列将显示待定。一旦代理安装完成，该值将变为 **Ops Agent**，这表明 Ops Agent 已成功安装。

现在我们可以通过 SSH 进入虚拟机实例，安装 func-e（用于运行 Envoy），创建一个基本的 Envoy 配置，并运行它，这样 Envoy 的应用日志就会被发送到 GCP 的云端日志。

## 安装 func-e

要在虚拟机上安装 func-e，请运行：

```
curl https://func-e.io/install.sh | sudo bash -s -- -b /usr/loca
```

我们可以运行 `func-e --version` 来检查安装是否成功。

## 发送 Envoy 应用日志到云端日志

让我们创建一个我们将在本实验中使用的原始 Envoy 配置。

```
static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                stat_prefix: ingress_http
              http_filters:
                - name: envoy.filters.http.router
                  route_config:
                    name: my_first_route
                    virtual_hosts:
                      - name: direct_response_service
                        domains: ["*"]
                        routes:
                          - match:
                              prefix: "/"
                            direct_response:
                              status: 200
                              body:
                                inline_string: "200"
```

将上述 YAML 保存为 `6-lab-3-gcp-logging.yaml`。

让我们来配置 Ops Agent，为 Envoy 创建一个新的接收器，描述如何检索日志。

```
logging:
  receivers:
    envoy:
      type: files
      include_paths:
        - /var/log/envoy.log
  service:
    pipelines:
      default_pipeline:
        receivers: [envoy]
```

将上述内容保存到虚拟机实例上的 `/etc/google-cloud-ops-agent/config.yaml` 文件中。要重新启动 Ops Agent，请运行 `sudo service google-cloud-ops-agent restart`。

在 Ops Agent 使用新配置的情况下，我们可以运行 Envoy，并告诉它把日志写到 `/var/log/envoy.log` 文件中，代理会在那里接收。

```
sudo func-e run -c 6-lab-3-gcp-logging.yaml --log-path /var/log/
```

接下来，我们可以点击日志，然后点击 GCP 中的日志资源管理器，查看虚拟机上运行的 Envoy 实例的日志。



The screenshot shows a list of log entries from the GCP Cloud Logging interface. The logs are timestamped at 21:16:08.961 on March 4, 2022. They are categorized under the 'envoy/source/server' namespace and show various initialization and configuration steps for an Envoy instance. Key log messages include:

- "[main] starting main dispatch loop"
- "[main] dependencies initialized, starting workers"
- "[main] all clusters initialized, initializing init manager"
- "[main] there is no configured limit to the number of allowed active connections. Set a limit via the r...
- "[upstream]m int: all clusters initialized"
- "[upstream]RTDS has finished initialization"
- "[config]loading static configuration"
- "[config]loading 1 listener(s)"
- "[config]loading 0 cluster(s)"
- "[config]loading 0 static secret(s)"
- "[config]loading tracing configuration"
- "[main]No admin address given, so no admin HTTP server started."
- "[main]runtime: {}"
- "[main] response trailer map: 168 bytes, grpc-message,grpc-status"

图 6.8.1: GCP的日志资源管理器中的 Envoy 日志

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-03-04 16:35:04

## 管理接口

在管理接口章节中，我们将学习 Envoy 所暴露的管理接口，以及我们可以用来检索配置和统计的不同端点，以及执行其他管理任务。

在本章结束时，你将了解如何启用管理接口以及我们可以通过它执行的不同任务。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 19:48:05

## 启用管理接口

在整个课程中，我们已经多次提到了管理接口。Envoy 暴露了一个管理接口，允许我们修改 Envoy，并获得一个视图和查询指标和配置。

管理接口由一个具有多个端点的 REST API 和一个简单的用户界面组成，如下图所示。

Command	Description
<code>certs</code>	print certs on machine
<code>clusters</code>	upstream cluster status
<code>config_dump</code>	dump current Envoy configs (experimental)
<code>contention</code>	dump current Envoy mutex contention stats (if enabled)
<code>cpuprofiler</code>	enable/disable the CPU profiler
<code>drain_listeners</code>	drain listeners
<code>healthcheck/fail</code>	cause the server to fail health checks
<code>healthcheck/ok</code>	cause the server to pass health checks
<code>heapprofiler</code>	enable/disable the heap profiler
<code>help</code>	print out list of admin commands
<code>hot_restart_version</code>	print the hot restart compatibility version
<code>init_dump</code>	dump current Envoy init manager information (experimental)
<code>listeners</code>	print listener info
<code>logging</code>	query/change logging levels
<code>memory</code>	print current allocation/heap usage
<code>quitquitquit</code>	exit the server
<code>ready</code>	print server state, return 200 if LIVE, otherwise return 503
<code>reopen_logs</code>	reopen access logs
<code>reset_counters</code>	reset all counters to zero
<code>runtime</code>	print runtime values
<code>runtime_modify</code>	modify runtime values
<code>server_info</code>	print server version/status information
<code>stats</code>	print server stats
<code>stats/prometheus</code>	print server stats in prometheus format
<code>stats/recentlookups</code>	Show recent stat-name lookups
<code>stats/recentlookups/clear</code>	clear list of stat-name lookups and counter
<code>stats/recentlookups/disable</code>	disable recording of reset stat-name lookup names
<code>stats/recentlookups/enable</code>	enable recording of reset stat-name lookup names

图 7.2.1：Envoy 管理接口

管理接口必须使用 `admin` 字段明确启用。例如：

```
admin:
  address:
    socket_address:
      address: 127.0.0.1
      port_value: 9901
```

在启用管理接口时要小心。任何有权限进入管理接口的人都可以进行破坏性的操作，比如关闭服务器（`/quitquit` 端点）。我们还可能让他们访问私人信息（指标、集群名称、证书信息等）。目前（Envoy 1.20 版本），管理端点是不安全的，也没有办法配置认证或 TLS。有一个工作项目正在进行中，它将限制只有受信任的 IP 和客户端证书才能访问，以确保传输安全。

在这项工作完成之前，应该只允许通过安全网络访问管理接口，而且只允许从连接到该安全网络的主机访问。我们可以选择只允许通过 `localhost` 访问管理接口，如上面的配置中所示。另外，如果你决定允许从远程主机访问，那么请确保你也设置了防火墙规则。

在接下来的课程中，我们将更详细地了解管理接口的不同功能。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 19:57:12

## 配置转储

`/config_dump` 端点是一种快速的方法，可以将当前加载的 Envoy 配置显示为 JSON 序列化的 proto 消息。

Envoy 输出以下组件的配置，并按照下面的顺序排列。

- 自举 (bootstrap)
- 集群 (clusters)
- 端点 (endpoints)
- 监听器 (listeners)
- 范围路由 (scoped routes)
- 路由 (routes)
- 秘密 (secrets)

## 包括 EDS 配置

为了输出端点发现服务 (EDS) 的配置，我们可以在查询中加入 `?include_eds` 参数。

## 筛选输出

同样，我们可以通过提供我们想要包括的资源和一个掩码来过滤输出，以返回一个字段的子集。

例如，为了只输出静态集群配置，我们可以在资源查询参数中使用 `static_clusters` 字段，从 `ClustersConfigDump proto` 在 `resource` 查询参数中使用。

```
$ curl localhost:9901/config_dump?resource=static_clusters
{
  "configs": [
    {
      "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfigDu
      "cluster": {
        "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluste
        "name": "instance_1",
      },
      ...
    },
    {
      "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfigDu
      "cluster": {
        "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluste
        "name": "instance_2",
      },
      ...
    }
  ]
}
```

## 使用 `mask` 参数

为了进一步缩小输出范围，我们可以在 `mask` 参数中指定该字段。例如，只显示每个集群的 `connect_timeout` 值。

```
$ curl localhost:9901/config_dump?resource=static_clusters&mask=
{
  "configs": [
    {
      "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfig"
      "cluster": [
        "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluster"
        "connect_timeout": "5s"
      ]
    },
    {
      "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfig"
      "cluster": [
        "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluster"
        "connect_timeout": "5s"
      ]
    },
    {
      "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfig"
      "cluster": [
        "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluster"
        "connect_timeout": "1s"
      ]
    }
  ]
}
```

## 使用正则表达式

另一个过滤选项是指定一个正则表达式来匹配加载的配置的名称。例如，要输出所有名称字段与正则表达式 `.*listener.*` 相匹配的监听器，我们可以这样写。

```
$ curl localhost:9901/config_dump?resource=static_clusters&name_
{
  "configs": [
    {
      "@type": "type.googleapis.com/envoy.admin.v3.ListenersConfig"
      "listener": [
        "@type": "type.googleapis.com/envoy.config.listener.v3.Listener"
        "name": "listener_0",
        "address": [
          "socket_address": [
            "address": "0.0.0.0",
            "port_value": 10000
          ]
        ],
        "filter_chains": [
          {}
        ]
      ],
      "last_updated": "2021-11-15T20:06:51.208Z"
    }
  ]
}
```

同样，`/init_dump` 端点列出了各种 Envoy 组件的未就绪目标的当前信息。和配置转储一样，我们可以使用 `mask` 查询参数来过滤特定字段。

## 证书

`/certs` 输出所有加载的 TLS 证书。数据包括证书文件名、序列号、主题候补名称和到期前天数。结果是 JSON 格式的，遵循 `admin.v3.Certificates` proto。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 20:04:36

## 统计

管理接口的统计输出的主要端点是通过 `/stats` 端点访问的。这个输入通常是用来调试的。我们可以通过向 `/stats` 端点发送请求或从管理接口访问同一路径来访问该端点。

该端点支持使用 `filter` 查询参数和正则表达式来过滤返回的统计资料。

另一个过滤输出的维度是使用 `usedonly` 查询参数。当使用时，它将只输出 Envoy 更新过的统计数据。例如，至少增加过一次的计数器，至少改变过一次的仪表，以及至少增加过一次的直方图。

默认情况下，统计信息是以 StatsD 格式写入的。每条统计信息都写在单独的一行中，统计信息的名称（例如，`cluster_manager.active_clusters`）后面是统计信息的值（例如，`15`）。

例如：

```
...
cluster_manager.active_clusters.15
cluster_manager.cluster_added:3
cluster_manager.cluster_modified:4
...
```

`format` 查询参数控制输出格式。设置为 `json` 将以 JSON 格式输出统计信息。如果我们想以编程方式访问和解析统计信息，通常会使用这种格式。

第二种格式是 Prometheus 格式（例如，`format=prometheus`）。这个选项以 Prometheus 格式格式化状态，可以用来与 Prometheus 服务器集成。另外，我们也可以使用 `/stats/prometheus` 端点来获得同样的输出。

## 内存

`/memory` 端点将输出当前内存分配和堆的使用情况，单位为字节。下面是 `/stats` 端点打印出来的信息的一个子集。

```
$ curl localhost:9901/memory
{
  "allocated": "5845672",
  "heap_size": "10485760",
  "pageheap_unmapped": "0",
  "pageheap_free": "3186688",
  "total_thread_cache": "80064",
  "total_physical_bytes": "12699350"
}
```

## 重置计数器

向 `/reset_counters` 发送一个 POST 请求，将所有计数器重置为零。注意，这不会重置或放弃任何发送到 statsd 的数据。它只影响到 `/stats` 端点的输出。在调试过程中可以使用 `/stats` 端点和 `/reset_counters` 端点。

## 服务器信息和状态

`/server_info` 端点输出运行中的 Envoy 服务器的信息。这包括版本、状态、配置路径、日志级别信息、正常运行时间、节点信息等。

该 [admin.v3.ServerInfo](#) proto 解释了由端点返回的不同字段。

`/ready` 端点返回一个字符串和一个错误代码，反映 Envoy 的状态。如果 Envoy 是活的，并准备好接受连接，那么它返回 HTTP 200 和字符串 `LIVE`。否则，输出将是一个 HTTP 503。这个端点可以作为准备就绪检查。

`/runtime` 端点以 JSON 格式输出所有运行时值。输出包括活动的运行时覆盖层列表和每个键的层值堆栈。这些值也可以通过向 `/runtime_modify` 端点发送 POST 请求并指定键 / 值对来修改。例如，`POST /runtime_modify?my_key_1=somevalue`。

`/hot_restart_version` 端点，加上 `--hot-restart-version` 标志，可以用来确定新的二进制文件和运行中的二进制文件是否热重启兼容。

热重启是指 Envoy 能够 "热" 或 "实时" 重启自己。这意味着 Envoy 可以完全重新加载自己（和配置）而不放弃任何现有的连接。

## Hystrix 事件流

`/hystrix_event_stream` 端点的目的是作为流源用于 [Hystrix 仪表盘](#)。向该端点发送请求将触发来自 Envoy 的统计流，其格式是 Hystrix 仪表盘所期望的。

注意，我们必须在引导配置中配置 Hystrix 统计同步，以使端点工作。

例如：

```
stats_sinks:  
- name: envoy.stat_sinks.hystrix  
  typed_config:  
    "@type": type.googleapis.com/envoy.config.metrics.v3.Hystr  
    num_buckets: 10
```

## 争用

如果启用了互斥追踪功能，`/contention` 端点会转储当前 Envoy 互斥内容的统计信息。

## CPU 和堆分析器

我们可以使用 `/cpuprofiler` 和 `/heapprofiler` 端点来启用或禁用 CPU / 堆分析器。注意，这需要用 gperftools 编译 Envoy。Envoy 的 GitHub 资源库有[文档](#)说明。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 20:12:39

## 日志

/logging 端点启用或禁用特定组件或所有记录器的不同日志级别。

要列出所有的记录器，我们可以向 /logging 端点发送一个 POST 请求。

```
$ curl -X POST localhost:9901/logging
active loggers:
  admin: info
  alternate_protocols_cache: info
  aws: info
  assert: info
  backtrace: info
  cache_filter: info
  client: info
  config: info
  ...
```

输出将包含记录器的名称和每个记录器的日志级别。要改变所有活动日志记录器的日志级别，我们可以使用 level 参数。例如，我们可以运行下面的程序，将所有日志记录器的日志记录级别改为 debug 。

```
$ curl -X POST localhost:9901/logging?level=debug
active loggers:
  admin: debug
  alternate_protocols_cache: debug
  aws: debug
  assert: debug
  backtrace: debug
  cache_filter: debug
  client: debug
  config: debug
  ...
```

要改变某个日志记录器的级别，我们可以用日志记录器的名称替换 level 查询参数名称。例如，要将 admin 日志记录器级别改为 warning ，我们可以运行以下程序。

```
$ curl -X POST localhost:9901/logging?admin=warning
active loggers:
  admin: warning
  alternate_protocols_cache: info
  aws: info
  assert: info
  backtrace: info
  cache_filter: info
  client: info
  config: info
```

为了触发所有访问日志的重新开放，我们可以向 /reopen\_logs 端点发送一个 POST 请求。

Copyright © 2017-2022 | 基于 CC 4.0 协议 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 20:14:32

## 集群

集群端点（`/clusters`）将显示配置的集群列表，并包括以下信息：

- 每个主机的统计数据
- 每个主机的健康状态
- 断路器设置
- 每个主机的权重和位置信息

这里的主机指的是每个被发现的属于上游集群的主机。

下面的片段显示了信息的模样（注意，输出是经过裁剪的）。

```
{
  "cluster_statuses": [
    {
      "name": "api_google_com",
      "host_statuses": [
        {
          "address": {
            "socket_address": {
              "address": "10.0.0.1",
              "port_value": 8080
            }
          },
          "stats": [
            {
              "value": "23",
              "name": "cx_total"
            },
            {
              "name": "rq_error"
            },
            {
              "value": "51",
              "name": "rq_success"
            },
            ...
          ],
          "health_status": {
            "eds_health_status": "HEALTHY"
          },
          "weight": 1,
          "locality": {}
        }
      ],
      "circuit_breakers": {
        "thresholds": [
          {
            "max_connections": 1024,
            "max_pending_requests": 1024,
            "max_requests": 1024,
            "max_retries": 3
          },
          {
            "priority": "HIGH",
            "max_connections": 1024,
            "max_pending_requests": 1024,
            "max_requests": 1024,
            "max_retries": 3
          }
        ]
      },
      "observability_name": "api_google_com"
    },
    ...
  ]
}
```

为了获得 JSON 输出，我们可以在发出请求或在浏览器中打开 URL 时附加 `?format=json`。

## 主机统计

输出包括每个主机的统计数据，如下表所解释。

指标名称	描述
<code>cx_total</code>	连接总数
<code>cx_active</code>	有效连接总数
<code>cx_connect_fail</code>	连接失败总数
<code>rq_total</code>	请求总数
<code>rq_timeout</code>	超时的请求总数
<code>rq_success</code>	有非 5xx 响应的请求总数
<code>rq_error</code>	有 5xx 响应的请求总数
<code>rq_active</code>	有效请求总数

## 主机健康状况

主机的健康状况在 `health_status` 字段下报告。健康状态中的值取决于健康检查是否被启用。假设启用了主动和被动（断路器）健康检查，该表显示了可能包含在 `health_status` 字段中的布尔字段。

字段名称	描述
<code>failed_active_health_check</code>	真, 如果该主机目前未能通过主动健康检查。
<code>failed_outlier_check</code>	真, 如果该宿主目前被认为是一个异常, 并已被弹出。
<code>failed_active_degraded_check</code>	如果主机目前通过主动健康检查被标记为降级, 则为真。
<code>pending_dynamic_removal</code>	如果主机已经从服务发现中移除, 但由于主动健康检查正在稳定, 则为真。
<code>pending_active_hc</code>	真, 如果该主机尚未被健康检查。
<code>excluded_via_immediate_hc_fail</code>	真, 如果该主机应被排除在恐慌、溢出等计算之外, 因为它被明确地通过协议信号从轮换中取出, 并且不打算被路由到。
<code>active_hc_timeout</code>	真, 如果主机由于超时而导致活动健康检查失败。
<code>eds_health_status</code>	默认情况下, 设置为 <code>healthy</code> (如果不使用 EDS)。否则, 它也可以被设置为 <code>unhealthy</code> 或 <code>degraded</code> 。

请注意, 表中的字段只有在设置为真时才会被报告。例如, 如果主机是健康的, 那么健康状态将看起来像这样。

```
"Health_status":{  
    "eds_health_status":"HEALTHY"  
}
```

如果配置了主动健康检查, 而主机是失败的, 那么状态将看起来像这样。

```
"Health_status":{  
    "failed_active_health_check": true,  
    "eds_health_status":"HEALTHY"  
}
```

## 监听器和监听器排空

`/listeners` 端点列出了所有配置的监听器。这包括名称以及每个监听器的地址和监听的端口。

例如：

```
$ curl localhost:9901/listeners
http_8080::0.0.0.0:8080
http_hello_world_9090::0.0.0.0:9090
```

对于 JSON 输出，我们可以在 URL 上附加 `?format=json`。

```
$ curl localhost:9901/listeners?format=json
{
  "listener_statuses": [
    {
      "name": "http_8080",
      "local_address": {
        "socket_address": {
          "address": "0.0.0.0",
          "port_value": 8080
        }
      }
    },
    {
      "name": "http_hello_world_9090",
      "local_address": {
        "socket_address": {
          "address": "0.0.0.0",
          "port_value": 9090
        }
      }
    }
  ]
}
```

## 监听器排空

发生排空 (draining) 的一个典型场景是在热重启排空期间。它涉及到在 Envoy 进程关闭之前，通过指示监听器停止接受传入的请求来减少打开连接的数量。

默认情况下，如果我们关闭 Envoy，所有的连接都会立即关闭。要进行优雅的关闭（即不关闭现有的连接），我们可以使用 `/drain_listeners` 端点，并加入一个可选的 `graceful` 查询参数。

Envoy 根据通过 `--drain-time-s` 和 `--drain-strategy` 指定的配置来排空连接。

如果没有提供，排空时间默认为 10 分钟（600 秒）。该值指定了 Envoy 将排空连接的时间——即在关闭它们之前等待多久。

排空策略参数决定了排空序列中的行为（例如，在热重启期间），连接是通过发送 "Connection:CLOSE"（HTTP/1.1）或 GOAWAY 帧（HTTP/2）。

有两种支持的策略：渐进（默认）和立即。当使用渐进策略时，随着排空时间的推移，排空的请求的百分比慢慢增加到 100%。即时策略将使所有的请求在排空序列开始后立即排空。

排空是按监听器进行的。然而，它必须在网络过滤器层面得到支持。目前支持优雅排空的过滤器是 Redis、Mongo 和 HTTP 连接管理器。

端点的另一个选项是使用 `inboundonly` 查询参数（例如，`/drain_listeners?inboundonly`）排空所有入站监听器的能力。这使用监听器上的 `traffic_direction` 字段来确定流量方向。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 20:27:24

## 分接式过滤器

分接式过滤器（Tap Filter）的目的是根据一些匹配的属性来记录 HTTP 流量。有两种方法来配置分接式过滤器。

1. 使用 Envoy 配置里面的 `static_config` 字段
2. 使用 `admin_config` 字段并指定配置 ID。

不同的是，我们在静态配置（`static_config`）中一次性提供所有东西——匹配配置和输出配置。当使用管理配置（`admin_config`）时，我们只提供配置 ID，然后在运行时使用 `/tap` 管理端点来配置过滤器。

正如我们所提到的，过滤器的配置被分成两部分：匹配配置和输出配置。

我们可以用匹配配置指定匹配谓词，告诉分接式过滤器要分接哪些请求并写入配置的输出。

例如，下面的片段显示了如何使用 `any_match` 来匹配所有的请求，无论其属性如何。

```
common_config:
  static_config:
    match:
      any_match: true
...
```

我们也有一个选项，可以在请求和响应 Header、Trailer 和正文上进行匹配。

## Header/Trailer 匹配

Header/Trailer 匹配器使用 `HttpHeadersMatch` proto，在这里我们指定一个头数组来匹配。例如，这个片段匹配任何请求头 `my-header` 被精确设置为 `hello` 的请求。

```
common_config:
  static_config:
    match:
      http_request_headers_match:
        headers:
          name: "my-header"
          string_match:
            exact: "hello"
...

```

请注意，在 `string_match` 中，我们可以使用其他匹配器（例如 `prefix`、`suffix`、`safe_regex`），正如前面解释的那样。

## 正文匹配

通用请求和响应正文 (body) 匹配使用 `HttpGenericBodyMatch` 来指定字符串或二进制匹配。顾名思义，字符串匹配 (`string_match`) 是在 HTTP 正文中寻找一个字符串，而二进制匹配 (`binary_match`) 是在 HTTP 正文中寻找一串字节的位置。

例如，如果响应体包含字符串 `hello`，则下面的片段可以匹配。

```
common_config:
  static_config:
    match:
      http_response_generic_body_match:
        patterns:
          string_match: "hello"
...

```

## 匹配谓词

我们可以用 `or_match`、`and_match` 和 `not_match` 等匹配谓词来组合多个 Header、Trailer 和正文匹配器。

`or_match` 和 `and_match` 使用 `MatchSet` 原语，描述逻辑 OR 或逻辑 AND。我们在匹配集内的 `rules` 字段中指定构成一个集合的规则列表。

下面的例子显示了如何使用 `and_match` 来确保响应体包含 `hello` 这个词，以及请求头 `my-header` 被设置为 `hello`。

```
common_config:
  static_config:
    match:
      and_match:
        rules:
          - http_response_generic_body_match:
              patterns:
                - string_match: "hello"
          - http_request_headers_match:
              headers:
                name: "my-header"
                string_match:
                  exact: "hello"
...

```

如果我们想实现逻辑 OR，那么我们可以用 `or_match` 字段替换 `and_match` 字段。字段内的配置将保持不变，因为两个字段都使用 `MatchSet` proto。

让我们使用与之前相同的例子来说明 `not_match` 是如何工作的。假设我们想过滤所有没有设置头信息 `my-header: hello` 的请求，以及响应体不包括 `hello` 这个字符串的请求。

下面是我们如何写这个配置。

```
common_config:
  static_config:
    match:
      not_match:
        and_match:
          rules:
            - http_response_generic_body_match:
                patterns:
                  - string_match: "hello"
            - http_request_headers_match:
                headers:
                  name: "my-header"
                  string_match:
                    exact: "hello"
...

```

`not_match` 字段和父 `match` 字段一样使用 `MatchPredicate` 原语。匹配字段是一个递归结构，它允许我们创建复杂的嵌套匹配配置。

这里要提到的最后一个字段是 `any_match`。这是一个布尔字段，当设置为 `true` 时，将总是匹配。

## 输出配置

一旦请求被过滤出来，我们需要告诉过滤器将输出写入哪里。目前，我们可以配置一个单一的输出沉积。

下面是一个输出配置示例。

```
...
output_config:
  sinks:
    - format: JSON_BODY_AS_STRING
      file_per_tap:
        path_prefix: tap
...

```

使用 `file_per_tap`，我们指定要为每个被监听的数据流输出一个文件。`path_prefix` 指定了输出文件的前缀。文件用以下格式命名：

`<path_prefix>_<id>.<pb | json>`

`id` 代表一个标识符，使我们能够区分流实例的记录跟踪。文件扩展名（`pb` 或 `json`）取决于格式选择。

捕获输出的第二个选项是使用 `streaming_admin` 字段。这指定了 `/tap` 管理端点将流式传输被捕获的输出。请注意，要使用 `/tap` 管理端点进行输出，还必须使用 `admin_config` 字段配置分接式过滤器。如果我们静态地配置了分接式过滤器，我们就不会使用 `/tap` 端点来获取输出。

## 格式选择

我们有多种输出格式的选项，指定消息的书写方式。让我们看看不同的格式，从默认格式开始，`JSON_BODY_AS_BYTES`。

`JSON_BODY_AS_BYTES` 输出格式将消息输出为 JSON，任何响应的 body 数据将在 `as_bytes` 字段中，其中包含 base64 编码的字符串。

例如，下面是分接输出的示例。

```
{
  "http_buffered_trace": {
    "request": {
      "headers": [
        {
          "key": ":authority",
          "value": "localhost:10000"
        },
        {
          "key": ":path",
          "value": "/"
        },
        {
          "key": ":method",
          "value": "GET"
        },
        {
          "key": ":scheme",
          "value": "http"
        },
        {
          "key": "user-agent",
          "value": "curl/7.64.0"
        },
        {
          "key": "accept",
          "value": "*/*"
        },
        {
          "key": "my-header",
          "value": "hello"
        },
        {
          "key": "x-forwarded-proto",
          "value": "http"
        },
        {
          "key": "x-request-id",
          "value": "67e3e8ac-429a-42fb-945b-ec25927fdcc1"
        }
      ],
      "trailers": []
    },
    "response": {
      "headers": [
        {
          "key": ":status",
          "value": "200"
        },
        {
          "key": "content-length",
          "value": "5"
        },
        {
          "key": "content-type",
          "value": "text/plain"
        },
        {
          "key": "date",
          "value": "Mon, 29 Nov 2021 19:31:43 GMT"
        },
        {
          "key": "server",
          "value": "envoy"
        }
      ]
    }
  }
}
```

```

        ],
        "body": {
            "truncated": false,
            "as_bytes": "aGVsbG8="
        },
        "trailers": []
    }
}

```

注意 `body` 中的 `as_bytes` 字段。该值是 `body` 数据的 base64 编码表示（本例中为 `hello`）。

第二种输出格式是 `JSON_BODY_AS_STRING`。与之前的格式不同的是，在 `JSON_BODY_AS_STRING` 中，`body` 数据是以字符串的形式写在 `as_string` 字段中。当我们知道 `body` 是人类可读的，并且不需要对数据进行 base64 编码时，这种格式很有用。

```

...
"body": {
    "truncated": false,
    "as_string": "hello"
},
...

```

其他三种格式类型是 `PROTO_BINARY`、`PROTO_BINARY_LENGTH_DELIMITED` 和 `PROTO_TEXT`。

`PROTO_BINARY` 格式以二进制 proto 格式写入输出。这种格式不是自限性的，这意味着如果分接写了多个没有任何长度信息的二进制消息，那么数据流将没有用处。如果我们在每个文件中写一个消息，那么输出格式将更容易解析。

我们也可以使用 `PROTO_BINARY_LENGTH_DELIMITED` 格式，其中消息被写成序列元组。每个元组是消息长度（编码为 32 位 protobuf varint 类型），后面是二进制消息。

最后，我们还可以使用 `PROTO_TEXT` 格式，在这种格式下，输出结果以下面的 protobuf 格式写入。

```
http_buffered_trace {
    request {
        headers {
            key: ":authority"
            value: "localhost:10000"
        }
        headers {
            key: ":path"
            value: "/"
        }
        headers {
            key: ":method"
            value: "GET"
        }
        headers {
            key: ":scheme"
            value: "http"
        }
        headers {
            key: "user-agent"
            value: "curl/7.64.0"
        }
        headers {
            key: "accept"
            value: "*/*"
        }
        headers {
            key: "debug"
            value: "true"
        }
        headers {
            key: "x-forwarded-proto"
            value: "http"
        }
        headers {
            key: "x-request-id"
            value: "af6e0879-e057-4efc-83e4-846ff4d46efe"
        }
    }
    response {
        headers {
            key: ":status"
            value: "500"
        }
        headers {
            key: "content-length"
            value: "5"
        }
        headers {
            key: "content-type"
            value: "text/plain"
        }
        headers {
            key: "date"
            value: "Mon, 29 Nov 2021 22:32:40 GMT"
        }
        headers {
            key: "server"
            value: "envoy"
        }
        body {
            as_bytes: "hello"
        }
    }
}
```

## 静态配置分接式过滤器

我们把匹配的配置和输出配置（使用 `file_per_tap` 字段）结合起来，静态地配置分接式过滤器。

下面是一个通过静态配置来配置分接式过滤器的片段。

```
- name: envoy.filters.http.tap
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.filters.http.t
    common_config:
      static_config:
        match_config:
          any_match: true
        output_config:
          sinks:
            - format: JSON_BODY_AS_STRING
              file_per_tap:
                path_prefix: my-tap
```

上述配置将匹配所有的请求，并将输出写入带有 `my-tap` 前缀的文件名中。

## 使用 `/tap` 端点配置分接式过滤器

为了使用 `/tap` 端点，我们必须在分接式过滤器配置中指定

`admin_config` 和 `config_id`。

```
- name: envoy.filters.http.tap
  typed_config:
    "@type": type.googleapis.com/envoy.extensions. filters.http.
    common_config:
      admin_config:
        config_id: my_tap_config_id
```

一旦指定，我们就可以向 `/tap` 端点发送 POST 请求以配置分接式过滤器。例如，下面是配置 `my_tap_config_id` 名称所引用的分接式过滤器的 POST 正文。

```
config_id: my_tap_config_id
tap_config:
  match_config:
    any_match: true
  output_config:
    sinks:
      - streaming_admin:{}
```

我们指定匹配配置的格式等同于我们为静态提供的配置所设置的格式。

使用管理配置和 `/tap` 端点的明显优势是，我们可以在运行时更新匹配配置，而且不需要重新启动 Envoy 代理。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 20:37:33

## 健康检查

`/healthcheck/fail` 可用于失败的入站健康检查。端点

`/healthcheck/ok` 用于恢复失败的端点。

这两个端点都需要使用 HTTP 健康检查过滤器。我们可能会在关闭服务器前或进行完全重启时使用它来排空服务器。当调用失败的健康检查选项时，所有的健康检查都将失败，无论其配置如何。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 20:47:10

## 实验 15：使用 HTTP 分接式过滤器

在这个实验中，我们将展示如何使用和配置 HTTP 分接式过滤器。我们将配置一个 `/error` 路由，它返回一个直接的响应，其主体为 `error` 值。然后，在分接式过滤器中，我们将配置匹配器，以匹配任何响应主体包含 `error` 字符串和请求头 `debug: true` 的请求。如果这两个条件都为真，那么我们就分接该请求并将输出写入一个前缀为 `tap_debug` 的文件。

让我们从创建匹配配置开始。我们将使用两个匹配器，一个用来匹配请求头（`http_request_headers_match`），另一个用来匹配响应体（`http_response_generic_body_match`）。我们将用逻辑上的 AND 来组合这两个条件。

下面是匹配配置的样子。

```
- name: envoy.filters.http.tap
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.filters.http.t
    common_config:
      static_config:
        match:
          and_match:
            rules:
              - http_request_headers_match:
                  headers:
                    name: debug
                    string_match:
                      exact: "true"
              - http_response_generic_body_match:
                  patterns:
                    - string_match: error
```

我们将使用 `JSON_BODY_AS_STRING` 格式，并将输出写入以 `tap_debug` 为前缀的文件。

```
output_config:
  sinks:
    - format: JSON_BODY_AS_STRING
      file_per_tap:
        path_prefix: tap_debug
```

让我们把这两块放在一起，创建一个完整的配置。我们将使用 `direct_response`，所以我们不需要设置或运行任何额外的服务。

```

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: ingress_http
                http_filters:
                  - name: envoy.filters.http.tap
                    typed_config:
                      "@type": type.googleapis.com/envoy.extensions.filters.
                        common_config:
                          static_config:
                            match:
                              and_match:
                                rules:
                                  - http_request_headers_match:
                                      headers:
                                        name: debug
                                        string_match:
                                          exact: "true"
                                  - http_response_generic_body_match:
                                      patterns:
                                        - string_match: error
                output_config:
                  sinks:
                    - format: JSON_BODY_AS_STRING
                      file_per_tap:
                        path_prefix: tap_debug
            - name: envoy.filters.http.router
          route_config:
            name: local_route
            virtual_hosts:
              - name: local_service
                domains: ["*"]
                routes:
                  - match:
                      path: "/"
                      direct_response:
                        status: 200
                        body:
                          inline_string: hello
                  - match:
                      path: "/error"
                      direct_response:
                        status: 500
                        body:
                          inline_string: error

```

将上述 YAML 保存为 `7-lab-1-tap-filter-1.yaml` 文件，并使用 func-e CLI 运行它：

```
func-e run -c 7-lab-1-tap-filter-1.yaml &
```

如果我们发送一个请求到 `http://localhost:10000`，我们会收到符合第一条路由的响应（HTTP 200 和 body `hello`）。这个请求不会被分接，因为我们没有提供任何头信息，响应体也没有包含 `error` 值。

让我们试着设置 `debug` 头并向 `/error` 端点发送一个请求。

```
$ curl -H "debug: true" localhost:10000/error  
error
```

这一次，在同一个文件夹中创建了一个包含被挖掘的请求内容的 JSON 文件。下面是该文件的内容应该是这样的。

```
{
  "http_buffered_trace": {
    "request": {
      "headers": [
        {
          "key": ":authority",
          "value": "localhost:10000"
        },
        {
          "key": ":path",
          "value": "/error"
        },
        {
          "key": ":method",
          "value": "GET"
        },
        {
          "key": ":scheme",
          "value": "http"
        },
        {
          "key": "user-agent",
          "value": "curl/7.64.0"
        },
        {
          "key": "accept",
          "value": "*/*"
        },
        {
          "key": "debug",
          "value": "true"
        },
        {
          "key": "x-forwarded-proto",
          "value": "http"
        },
        {
          "key": "x-request-id",
          "value": "4855ee5d-7798-4c50-8692-a6989e72ca9b"
        }
      ],
      "trailers": []
    },
    "response": {
      "headers": [
        {
          "key": ":status",
          "value": "500"
        },
        {
          "key": "content-length",
          "value": "5"
        },
        {
          "key": "content-type",
          "value": "text/plain"
        },
        {
          "key": "date",
          "value": "Mon, 29 Nov 2021 22:38:32 GMT"
        },
        {
          "key": "server",
          "value": "envoy"
        }
      ]
    }
  }
}
```

```

        ],
        "body": {
            "truncated": false,
            "as_string": "error"
        },
        "trailers": []
    }
}

```

输出显示了所有的请求头和 Trailer，以及我们收到的响应。

我们将在下一个例子中使用同样的场景，但我们将使用 `/tap` 管理端点来实现它。

首先，让我们创建 Envoy 配置。

```

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: ingress_http
                http_filters:
                  - name: envoy.filters.http.tap
                    typed_config:
                      "@type": type.googleapis.com/envoy.extensions.filt
                        common_config:
                          admin_config:
                            config_id: my_tap_id
            - name: envoy.filters.http.router
              route_config:
                name: local_route
                virtual_hosts:
                  - name: local_service
                    domains: [ "*" ]
                    routes:
                      - match:
                          path: "/"
                        direct_response:
                          status: 200
                          body:
                            inline_string: hello
                      - match:
                          path: "/error"
                        direct_response:
                          status: 500
                          body:
                            inline_string: error
      admin:
        address:
          socket_address:
            address: 0.0.0.0
            port_value: 9901

```

这一次，我们使用 `admin_config` 字段并指定配置 ID。此外，我们要使管理接口使用 `/tap` 端点。

将上述 YAML 保存为 `7-lab-1-tap-filter-2.yaml`，并使用 func-e CLI 运行它。

```
func-e run -c 7-lab-1-tap-filter-2.yaml &
```

如果我们尝试向 `/` 和 `error` 路径发送请求，我们会得到预期的响应。我们必须用 tap 配置向 `/tap` 端点发送一个 POST 请求，以启用请求分接。

让我们使用这个符合任何请求的 tap 配置。

```
{
  "config_id": "my_tap_id",
  "tap_config": {
    "match": {
      "any_match": true
    },
    "output_config": {
      "sinks": [
        {
          "streaming_admin": {}
        }
      ]
    }
  }
}
```

注意，我们提供的是与我们在 Envoy 配置中定义的 ID 相匹配的配置 ID。如果我们提供了一个无效的配置 ID，那么在向 `/tap` 端点发送 POST 请求时，我们会得到一个错误。

```
Unknown config id 'some_tap_id'. No extension has registered wit
```

我们还使用 `streaming_admin` 字段作为输出汇，这意味着如果 `/tap` 的 POST 请求被接受，那么 Envoy 将流式处理序列化的 JSON 信息，直到我们终止请求。

让我们把上述 JSON 保存到 `tap-config-any.json`，然后用 cURL 向 `/tap` 端点发送一个 POST 请求。

```
curl -X POST -d @tap-config-any.json http://localhost:9901/tap
```

我们将打开第二个终端窗口，向 `localhost:10000` 发送一个 cURL 请求，以测试配置。由于我们对所有的请求都进行了匹配，我们将在第一个终端窗口中看到流式分接的输出。

```
{  
    "http_buffered_trace": {  
        "request": {  
            "headers": [  
                {  
                    "key": ":authority",  
                    "value": "localhost:10000"  
                },  
                {  
                    "key": ":path",  
                    "value": "/"  
                },  
                {  
                    "key": ":method",  
                    "value": "POST"  
                },  
                {  
                    "key": ":scheme",  
                    "value": "http"  
                },  
                {  
                    "key": "user-agent",  
                    "value": "curl/7.64.0"  
                },  
                {  
                    "key": "accept",  
                    "value": "*/*"  
                },  
                {  
                    "key": "content-length",  
                    "value": "198"  
                },  
                {  
                    "key": "content-type",  
                    "value": "application/x-www-form-urlencoded"  
                },  
                {  
                    "key": "x-forwarded-proto",  
                    "value": "http"  
                },  
                {  
                    "key": "x-request-id",  
                    "value": "59ca4c38-6112-444d-9b64-ff30e1326338"  
                }  
            ],  
            "trailers": []  
        },  
        "response": {  
            "headers": [  
                {  
                    "key": ":status",  
                    "value": "200"  
                },  
                {  
                    "key": "content-length",  
                    "value": "5"  
                },  
                {  
                    "key": "content-type",  
                    "value": "text/plain"  
                },  
                {  
                    "key": "date",  
                    "value": "Mon, 29 Nov 2021 23:09:25 GMT"  
                },  
            ]  
        }  
    }  
}
```

```
{  
    "key": "server",  
    "value": "envoy"  
},  
{  
    "key": "connection",  
    "value": "close"  
}  
],  
"body": {  
    "truncated": false,  
    "as_bytes": "aGVsbG8="  
},  
"trailers": []  
}  
}
```

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-03-04 16:35:45

## 扩展 Envoy 概览

在本章节中，我们将了解扩展 Envoy 的不同方法。

我们将更详细地介绍使用 Lua 和 Wasm 过滤器来扩展 Envoy 的功能。

在本章结束时，你将了解扩展 Envoy 的不同方法以及如何使用 Lua 和 Wasm 过滤器。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](#) all right reserved. Updated at 2022-02-28 20:48:00

## 可扩展性概述

扩展 Envoy 的一种方式是实现不同的过滤器，处理或增强请求。这些过滤器可以生成统计数据，翻译协议，修改请求，等等。

HTTP 过滤器就是一个例子，比如外部的 authz 过滤器和其他内置于 Envoy 二进制的过滤器。

此外，我们还可以编写我们的过滤器，让 Envoy 动态地加载和运行。我们可以通过正确的顺序声明来决定我们要在过滤器链中的哪个位置运行过滤器。

我们有几个选择来扩展 Envoy。默认情况下，Envoy 过滤器是用 C++ 编写的。但是，我们可以用 Lua 脚本编写，或者使用 WebAssembly (WASM) 来开发其他编程语言的 Envoy 过滤器。

请注意，与 C++ 过滤器相比，Lua 和 Wasm 过滤器的 API 是有限的。

### 1. 原生 C++ API

第一个选择是编写原生 C++ 过滤器，然后将其与 Envoy 打包。这就需要我们重新编译 Envoy，并维护我们的版本。如果我们试图解决复杂或高性能的用例，采取这种方式是有意义的。

### 2. Lua 过滤器

第二个选择是使用 Lua 脚本。在 Envoy 中有一个 HTTP 过滤器，允许我们定义一个 Lua 脚本，无论是内联还是外部文件，并在请求和响应流程中执行。

### 3. Wasm 过滤器

最后一个选项是基于 Wasm 的过滤器。我们用这个选项把过滤器写成一个单独的 Wasm 模块，Envoy 在运行时动态加载它。

在接下来的模块中，我们将学习更多关于 Lua 和 Wasm 过滤器的知识。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-03-01 09:40:39

## Lua 过滤器

Envoy 具有一个内置的 HTTP Lua 过滤器，允许在请求和响应流中运行 Lua 脚本。Lua 是一种可嵌入的脚本语言，主要在嵌入式系统和游戏中流行。Envoy 使用 [LuaJIT](#) (Lua 的即时编译器) 作为运行时。LuaJIT 支持的最高 Lua 脚本版本是 5.1，其中一些功能来自 5.2。

在运行时，Envoy 为每个工作线程创建一个 Lua 环境。正因为如此，没有真正意义上的全局数据。任何在加载时创建和填充的全局数据都可以从每个独立的工作线程中看到。

Lua 脚本是以同步风格的 coroutines 运行的，即使它们可能执行复杂的异步任务。这使得它更容易编写。Envoy 通过一组 API 执行所有的网络 / 异步处理。当一个异步任务被调用时，Envoy 会暂停脚本的执行，等到异步操作完成就会恢复。

我们不应该从脚本中执行任何阻塞性操作，因为这会影响 Envoy 的性能。我们应该只使用 Envoy 的 API 来进行所有的 IO 操作。

我们可以使用 Lua 脚本修改和 / 或检查请求和响应头、正文和 Trailer。我们还可以对上游主机进行出站异步 HTTP 调用，或者执行直接响应，跳过任何进一步的过滤器迭代。例如，在 Lua 脚本中，我们可以进行上游的 HTTP 调用并直接响应，而不继续执行其他过滤器。

## 如何配置 Lua 过滤器

Lua 脚本可以使用 `inline_code` 字段进行内联定义，或者使用过滤器上的 `source_codes` 字段引用本地文件。

```
name: envoy.filters.http.lua
typed_config:
  "@type": type.googleapis.com/envoy.extensions.filters.http.lua
  inline_code: |
    -- Called on the request path.
    function envoy_on_request(request_handle)
      -- Do something.
    end
    -- Called on the response path.
    function envoy_on_response(response_handle)
      -- Do something.
    end
  source_codes:
    myscript.lua:
      filename: /scripts/myscript.lua
```

Envoy 将上述脚本视为全局脚本，对每一个 HTTP 请求都会执行它。在每个脚本中可以定义两个全局函数。

```
function envoy_on_request(request_handle)
end
```

和

```
function envoy_on_response(response_handle)
end
```

`envoy_on_request` 函数在请求路径上被调用，而 `envoy_on_response` 脚本则在响应路径上被调用。每个函数都接收一个句柄，该句柄有不同的定义方法。脚本可以包含响应或请求函数，也可以包含两者。

我们也有一个选项，可以在虚拟主机、路由或加权集群级别上按路由禁用或改写脚本。

使用 `typed_per_filter_config` 字段来禁用或引用主机、路由或加权集群层面上的现有 Lua 脚本。例如，下面是如何使用 `typed_per_filter_config` 来引用一个现有的脚本（例如：`some-script.lua`）。

```
typed_per_filter_config:
  envoy.filters.http.lua:
    "@type": type.googleapis.com/envoy.extensions.filters.http.
    name: some-script.lua
```

同样地，我们可以这样定义 `source_code` 和 `inline_string` 字段，而不是指定 `name` 字段。

```
typed_per_filter_config:
  envoy.filters.http.lua:
    "@type": type.googleapis.com/envoy.extensions.filters.http.1
    source_code:
      inline_string: |
        function envoy_on_response(response_handle)
          -- Do something on response.
        end
```

## 流处理 API

我们在前面提到，`request_handle` 和 `response_handle` 流句柄会被传递给全局 `request` 和 `response` 函数。

在流句柄上可用的方法包括 `headers`、`body`、`metadata`、各种日志方法（如

`logTrace`、`logInfo`、`logDebug ...`）、`httpCall`、`connection` 等等。你可以在 [Lua 过滤器源码](#) 中找到完整的方法列表。

除了流对象外，API 还支持以下对象：

- [Header 对象](#) (由 `headers()` 方法返回)
- 缓冲区对象 (由 `body()` 方法返回)。
- [动态元数据对象](#) (由 `metadata()` 方法返回)
- [Stream 信息对象](#) (由 `streamInfo()` 方法返回)
- 连接对象 (通过 `connection()` 方法返回)
- [SSL 连接信息对象](#) (由连接对象的 `ssl()` 方法返回)

你会在那里看到如何使用 Lua 实验中的一些对象和方法。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-02-28 20:56:39

## WebAssembly (Wasm)

Wasm 是一种可执行代码的可移植二进制格式，依赖于一个开放的标准。它允许开发人员用自己喜欢的编程语言编写，然后将代码编译成 **Wasm** 模块。

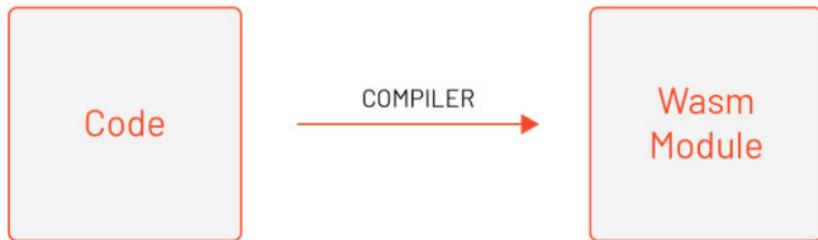


图 8.4.1：代码编译成 *wasm*

Wasm 模块与主机环境隔离，并在一个称为虚拟机（**VM**）的内存安全沙盒中执行。Wasm 模块使用一个 API 与主机环境进行通信。

Wasm 的主要目标是在网页上实现高性能应用。例如，假设我们要用 Javascript 构建一个网页应用程序。我们可以用 Go（或其他语言）写一些，并将其编译成一个二进制文件，即 Wasm 模块。然后，我们可以在与 Javascript 网页应用程序相同的沙盒中运行已编译的 Wasm 模块。

最初，Wasm 被设计为在网络浏览器中运行。然而，我们可以将虚拟机嵌入到其他主机应用程序中，并执行它们。这就是 Envoy 的作用！

Envoy 嵌入了 V8 虚拟机的一个子集。V8 是一个用 C++ 编写的高性能 JavaScript 和 WebAssembly 引擎，它被用于 Chrome 和 Node.js 等。

我们在本课程的前面提到，Envoy 使用多线程模式运行。这意味着有一个主线程，负责处理配置更新和执行全局任务。

除了主线程之外，还有负责代理单个 HTTP 请求和 TCP 连接的 worker 线程。这些 worker 线程被设计为相互独立。例如，处理一个 HTTP 请求的 worker 线程不会受到其他处理其他请求的 worker 线程的影响。

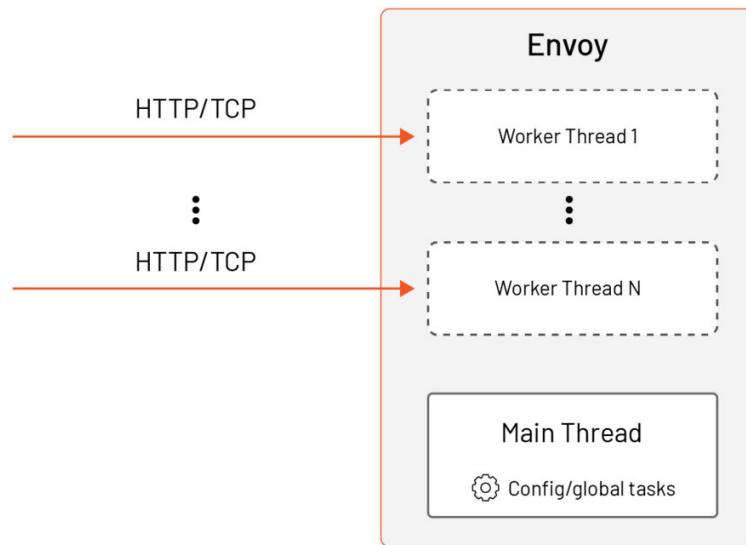


图 8.4.2: Envoy 线程

每个线程拥有自己的资源副本，包括 Wasm 虚拟机。这样做的原因是为了避免任何昂贵的跨线程同步，即实现更高的内存使用率。

Envoy 在运行时将每个独特的 Wasm 模块（所有 \*.wasm 文件）加载到一个独特的 Wasm VM。由于 Wasm VM 不是线程安全的（即，多个线程必须同步访问一个 Wasm VM），Envoy 为每个将执行扩展的线程创建一个单独的 Wasm VM 副本。因此，每个线程可能同时有多个 Wasm VM 在使用。

## Proxy-Wasm

我们将使用的 SDK 允许我们编写 Wasm 扩展，这些扩展是 HTTP 过滤器，网络过滤器，或称为 **Wasm 服务** 的专用扩展类型。这些扩展在 Wasm 虚拟机内的 worker 线程（HTTP 过滤器，网络过滤器）或主线程（Wasm 服务）上执行。正如我们提到的，这些线程是独立的，它们本质上不知道其他线程上发生的请求处理。

HTTP 过滤器是处理 HTTP 协议的，它对 HTTP Header、body 等进行操作。同样，网络过滤器处理 TCP 协议，对数据帧和连接进行操作。我们 also 可以说，这两种插件类型是无状态的。

Envoy 还支持有状态的场景。例如，你可以编写一个扩展，将请求数据、日志或指标等统计信息在多个请求之间进行汇总——这意味着跨越了许多 worker 线程。对于这种情况，我们会使用 Wasm 服务类型。Wasm 服务类型运行在单个虚拟机上；这个虚拟机只有一个实例，它运行在 Envoy 主线程上。你可以用它来汇总无状态过滤器的指标或日志。

下图显示了 Wasm 服务扩展是如何在主线程上执行的，而不是 HTTP 或网络过滤器，后者是在 worker 线程上执行。

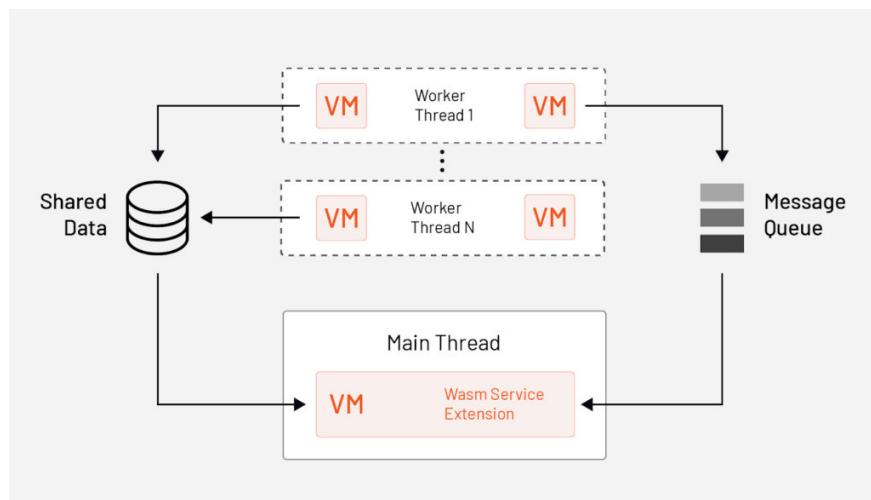


图 8.4.3: API

事实上，Wasm 服务扩展是在主线程上执行的，并不影响请求延迟。另一方面，网络或 HTTP 过滤器会影响延迟。

图中显示了在主线程上运行的 Wasm 服务扩展，它使用消息队列 API 订阅队列并接收由运行在 worker 线程上的 HTTP 过滤器或网络过滤器发送的消息。然后，Wasm 服务扩展可以聚合从 worker 线程上收到的数据。

Wasm 服务扩展并不是持久化数据的唯一方法。你也可以调用 HTTP 或 gRPC API。此外，我们可以使用定时器 API 在请求之外执行行动。

我们提到的 API、消息队列、定时器和共享数据都是由 [Proxy-Wasm](#) 提供的。

Proxy-Wasm 是一个代理无关的 ABI（应用二进制接口）标准，它规定了代理（我们的主机）和 Wasm 模块如何互动。这些互动是以函数和回调的形式实现的。

Proxy-Wasm 中的 API 与代理无关，这意味着它们可以与 Envoy 代理以及任何其他代理（例如 [MOSN](#)）一起实现 Proxy-Wasm 标准。这使得你的 Wasm 过滤器可以在不同的代理之间移植，而且它们并不局限于 Envoy。

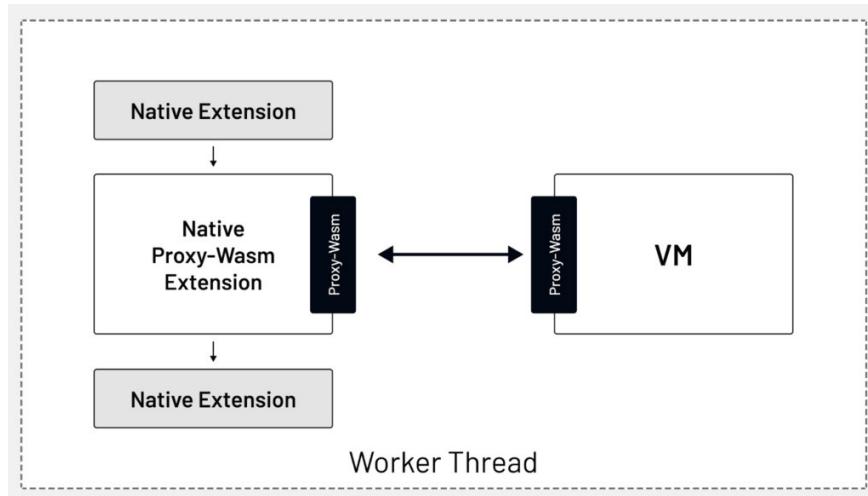


图 8.4.4: Proxy-Wasm

当请求进入 Envoy 时，它们会经过不同的过滤器链，被过滤器处理，在链中的某个点，请求数据会流经本地 Proxy-Wasm 扩展。

这个扩展使用 Proxy-Wasm 接口与运行在虚拟机内的扩展通信。过滤器处理完数据后，该链就会继续，或停止，这取决于从扩展返回的结果。

基于 Proxy-Wasm 规范，我们可以使用一些特定语言的 SDK 实现来编写扩展。

在其中一个实验中，我们将使用 [Go SDK for Proxy-Wasm](#) 来编写 Go 中的 Proxy-Wasm 插件。

[TinyGo](#) 是一个用于嵌入式系统和 WebAssembly 的编译器。它不支持使用所有的标准 Go 包。例如，不支持一些标准包，如 `net` 和其他。

你还可以选择使用 Assembly Script、C++、Rust 或 Zig。

## 配置 Wasm 扩展

Envoy 中的通用 Wasm 扩展配置看起来像这样。

```

- name: envoy.filters.http.wasm
  typed_config:
    "@type": type.googleapis.com/udpa.type.v1.TypedStruct
    type_url: type.googleapis.com/envoy.extensions.filters.http.
    value:
      config:
        vm_config:
          vm_id: "my_vm"
          runtime: "envoy.wasm.runtime.v8"
          configuration:
            "@type": type.googleapis.com/google.protobuf.StringV
            value: '{"plugin-config": "some-value"}'
        code:
          local:
            filename: "my-plugin.wasm"
        configuration:
          "@type": type.googleapis.com/google.protobuf.StringVal
          value: '{"vm-wide-config": "some-value"}'

```

`vm_config` 字段用于指定 Wasm 虚拟机、运行时，以及我们要执行的 `.wasm` 扩展的实际指针。

`vm_id` 字段在虚拟机之间进行通信时使用。然后这个 ID 可以用来通过共享数据 API 和队列在虚拟机之间共享数据。请注意，要在多个插件中重用虚拟机，你必须使用相同的 `vm_id`、运行时、配置和代码。

下一个项目是 `runtime`。这通常被设置为 `envoy.wasm.runtime.v8`。例如，如果我们用 Envoy 编译 Wasm 扩展，我们会在这里使用 `null` 运行时。其他选项是 Wasm micro runtime、Wasm VM 或 Wasmtime；不过，这些在官方 Envoy 构建中都没有启用。

`vm_config` 字段下的配置是用来配置虚拟机本身的。除了虚拟机 ID 和运行时外，另一个重要的部分是 `code` 字段。

`code` 字段是我们引用编译后的 Wasm 扩展的地方。这可以是一个指向本地文件的指针（例如，`/etc/envoy/my-plugin.wasm`）或一个远程位置（例如，`https://wasm.example.com/my-plugin.wasm`）。

`configuration` 文件，一个在 `vm_config` 下，另一个在 `config` 层，用于为虚拟机和插件提供配置。然后当虚拟机或插件启动时，可以从 Wasm 扩展代码中读取这些值。

要运行一个 Wasm 服务插件，我们必须在 `bootstrap_extensions` 字段中定义配置，并将 `singleton` 布尔字段的值设置为真。

```

bootstrap_extensions:
- name: envoy.bootstrap.wasm
  typed_config:
    "@type": type.googleapis.com/envoy.extensions.wasm.3.WasmSer
    singleton: true
    config:
      vm_config:{ ...}

```

# 开发 Wasm 扩展 - Proxy-Wasm Go SDK API

在开发 Wasm 扩展时，我们将学习上下文、hostcall API 和入口点。

## 上下文

上下文是 Proxy-Wasm SDK 中的一个接口集合，并与我们前面解释的概念相匹配。

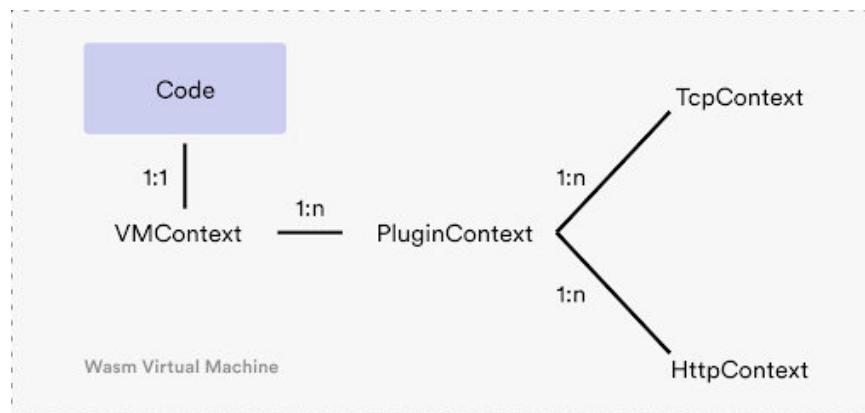


图 8.4.5：上下文

例如，每个虚拟机中都有一个 `VMContext`，可以有一个或多个 `PluginContexts`。这意味着我们可以在同一个虚拟机上下文中运行不同的插件（即使用同一个 `vm_id` 时）。每个 `PluginContext` 对应于一个插件实例。那就是 `TcpContext` (TCP 网络过滤器) 或 `HttpContext` (HTTP 过滤器)。

`VMContext` 接口定义了两个函数：`OnVMStart` 函数和 `NewPluginContext` 函数。

```

type VMContext interface {
    OnVMStart(vmConfigurationSize int) OnVMStartStatus
    NewPluginContext(contextID uint32) PluginContext
}

```

顾名思义，`OnVMStart` 在虚拟机创建后被调用。在这个函数中，我们可以使用 `GetVMConfiguration` hostcall 检索可选的虚拟机配置。这个函数的目的是执行任何虚拟机范围的初始化。

作为开发者，我们需要实现 `NewPluginContext` 函数，在该函数中我们创建一个 `PluginContext` 的实例。

`PluginContext` 接口定义了与 `VMContext` 类似的功能。下面是这个接口。

```
type PluginContext interface {
    OnPluginStart(pluginConfigurationSize int) OnPluginStartStatus
    OnPluginDone() bool

    OnQueueReady(queueID uint32)
    OnTick()

    NewTcpContext(contextID uint32) TcpContext
    NewHttpContext(contextID uint32) HttpContext
}
```

`OnPluginStart` 函数与我们前面提到的 `OnVMStart` 函数类似。它在插件被创建时被调用。在这个函数中，我们也可以使用 `GetPluginConfiguration` API 来检索插件的特定配置。我们还必须实现 `NewTcpContext` 或 `NewHttpContext`，在代理中响应 HTTP/TCP 流时被调用。这个上下文还包含一些其他的函数，用于设置队列（`OnQueueReady`）或在流处理的同时做异步任务（`onTick`）。

参考 [Proxy Wasm Go SDK Github 仓库](#) 中的 `context.go` 文件，以获得最新的接口定义。

## Hostcall API

[这里](#) 实现的 hostcall API，为我们提供了与 Wasm 插件的 Envoy 代理互动的方法。

hostcall API 定义了读取配置的方法；设置共享队列并执行队列操作；调度 HTTP 调用，从请求和响应流中检索 Header、Trailer 和正文并操作这些值；配置指标；以及更多。

## 入口点

插件的入口点是 `main` 函数。Envoy 创建了虚拟机，在它试图创建 `VMContext` 之前，它调用了 `main` 函数。在典型的实现中，我们把 `SetVMContext` 方法称为 `main` 函数。

```
func main() {
    proxywasm.SetVMContext(&myVMContext{})
}

type myVMContext struct { ... }

var _ types.VMContext = &myVMContext{}
```

## 实验 16：使用 Lua 脚本扩展 Envoy

在这个实验中，我们将编写一个 Lua 脚本，为响应头添加一个头，并使用一个文件中定义的全局脚本。

我们将创建一个 Envoy 配置和一个 Lua 脚本，在响应句柄上添加一个头。由于我们不会使用请求路径，所以我们不需要定义 `envoy_on_request` 函数。响应函数看起来像这样。

```
function envoy_on_response(response_handle)
    response_handle:headers():add("hello", "world")
end
```

我们在从 `headers()` 函数返回的 `header` 对象上调用 `add(<header-name>, <header-value>)` 函数。

让我们在 Envoy 配置中内联定义这个脚本。为了简化配置，我们将使用 `direct_response`，而不是集群。

```
static_resources:
  listeners:
    - name: main
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: ingress_http
                route_config:
                  name: some_route
                  virtual_hosts:
                    - name: some_service
                      domains:
                        - "*"
                      routes:
                        - match:
                            prefix: "/"
                          direct_response:
                            status: 200
                            body:
                              inline_string: "200"
              http_filters:
                - name: envoy.filters.http.lua
                  typed_config:
                    "@type": type.googleapis.com/envoy.extensions.filters.
                      inline_code: |
                        function envoy_on_response(response_handle)
                          response_handle:headers():add("hello", "world")
                        end
            - name: envoy.filters.http.router
```

将上述 YAML 保存为 `8-lab-1-lua-script.yaml` 并运行它。

```
func-e run -c 8-Lab-1-lua-script.yaml &
```

为了测试这个功能，我们可以向 `localhost:10000` 发送一个请求，并检查响应头。

```
$ curl -v localhost:10000
...
> GET / HTTP/1.1
> Host: localhost:10000
> User-Agent: curl/7.64.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-length: 3
< content-type: text/plain
< hello: world
< date: Tue, 23 Nov 2021 21:37:01 GMT
< server: envoy
<
200
```

输出应该包括我们在其他标准头文件中添加的 `hello: world` header。

让我们来看看一个更复杂的情况。对于传入的请求，我们要检查它们是否有一个叫做 `my-request-id` 的头，如果这个头不存在，那么我们要为所有 GET 请求添加一个 `my-request-id` 头。

因为我们想在 `envoy_on_response` 函数中检查方法和一个头，我们将使用动态元数据在 `envoy_on_request` 函数中存储这些值。然后，在响应函数中，我们可以读取元数据，检查头是否被设置，方法是否为 GET，并添加 `my-request-id` 头。

下面是代码的样子。

```
function envoy_on_request(request_handle)
    local headers = request_handle:headers()
    local metadata = request_handle:streamInfo():dynamicMetadata()
    metadata:set("envoy.filters.http.lua", "requestInfo", {
        requestId = headers:get("my-request-id"),
        method = headers:get(":method"),
    })
end
function envoy_on_response(response_handle)
    local requestInfoObj = response_handle:streamInfo():dynamicMet

    local requestId = requestInfoObj.requestId
    local method = requestInfoObj.method
    if (requestId == nil or requestId == '') and (method == 'GET')
        response_handle:logInfo("Adding request ID header")
        response_handle:headers():add("my-request-id", "some_id_here")
    end
end
```

注意，目前我们使用 `some_id_here` 作为 `my-request-id` 的值，以后我们会创建一个函数，为我们生成一个 ID。下面是完整的 Envoy 配置的样子。

```

static_resources:
  listeners:
    - name: main
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: ingress_http
                route_config:
                  name: some_route
                  virtual_hosts:
                    - name: some_service
                      domains:
                        - "*"
                      routes:
                        - match:
                            prefix: "/"
                          direct_response:
                            status: 200
                            body:
                              inline_string: "200"
            http_filters:
              - name: envoy.filters.http.lua
                typed_config:
                  "@type": type.googleapis.com/envoy.extensions.filters.
                    inline_code: |
                      function envoy_on_request(request_handle)
                        local headers = request_handle:headers()
                        local metadata = request_handle:streamInfo():d
                        metadata:set("envoy.filters.http.lua", "reques
                          requestId = headers:get("my-request-id"),
                          method = headers:get(":method"),
                        })
                      end
                      function envoy_on_response(response_handle)
                        local requestInfoObj = response_handle:streamI
                        local requestId = requestInfoObj.requestId
                        local method = requestInfoObj.method
                        if (requestId == nil or requestId == '') and (
                          response_handle:logInfo("Adding request ID h
                            response_handle:headers():add("my-request-id
                          end
                        end
              - name: envoy.filters.http.router

```

将上述 YAML 保存为 `8-lab-1-lua-script-1.yaml` 并运行它。

```
func-e run -c 8-Lab-1-lua-script-1.yaml &
```

让我们试一试几种情况。首先，我们将发送一个没有设置 `my-request-id` 头的 GET 请求。

```
$ curl -v localhost:10000
...
[2021-11-23 22:59:35.932][2258][info][lua] [source/extensions/filters/http/lua]
< HTTP/1.1 200 OK
< content-length: 3
< content-type: text/plain
< my-request-id: some_id_here
< date: Tue, 23 Nov 2021 22:59:35 GMT
< server: envoy
<
* Connection #0 to host Localhost left intact
200
```

我们知道 Lua 代码运行了，因为我们看到了日志条目和 `my-request-id` 头的设置。

让我们试着发送一个 POST 请求。

```
$ curl -X POST -v localhost:10000
...
> POST / HTTP/1.1
> Host: localhost:10000
> User-Agent: curl/7.64.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-length: 3
< content-type: text/plain
< date: Mon, 29 Nov 2021 23:49:58 GMT
< server: envoy
```

注意到头信息中没有包括 `my-request-id` 头信息。最后，让我们也尝试发送一个 GET 请求，但也提供 `my-request-id` 头。在这种情况下，`my-request-id` 头信息也不应该被包含在响应中。

```
$ curl -v -H "my-request-id: something" localhost:10000
...
> GET / HTTP/1.1
> Host: localhost:10000
> User-Agent: curl/7.64.0
> Accept: */*
> my-request-id: something
>
< HTTP/1.1 200 OK
< content-length: 3
< content-type: text/plain
< date: Mon, 29 Nov 2021 23:51:57 GMT
< server: envoy
<
* Connection #0 to host Localhost left intact
```

作为最后的练习，我们将创建一个单独的 `.lua` 脚本，生成一个简单的随机字符串，可以用于请求 ID。我们将加载该脚本，然后在响应函数中调用它来获得请求 ID。

让我们创建一个 `library.lua` 文件，内容如下。

```
LIBRARY = {}

function LIBRARY.RandomString()
    local result = ""
    for i = 1, 24 do
        result = result .. string.char(math.random(97, 122))
    end
    return result
end

return LIBRARY
```

我们正在声明一个名为 `LIBRARY` 的表和一个名为 `RandomString` 的函数。

将上述 Lua 脚本保存为一个名为 `library.lua` 的文件，并将其放在你的 Envoy 进程将要运行的同一个文件夹中。

Luajit 运行时在进程的工作目录和 `/usr/local/share/lua/5.1` 文件夹中寻找 Lua 模块。

我们在 Envoy 配置中的现有代码将基本保持不变。我们只需要加载 `library.lua` 和调用 `RandomString` 函数。

这是更新后 Envoy 配置。

```

static_resources:
  listeners:
    - name: main
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.
                  stat_prefix: ingress_http
            route_config:
              name: some_route
              virtual_hosts:
                - name: some_service
                  domains:
                    - "*"
                  routes:
                    - match:
                        prefix: "/"
                      direct_response:
                        status: 200
                        body:
                          inline_string: "200"
            http_filters:
              - name: envoy.filters.http.lua
                typed_config:
                  "@type": type.googleapis.com/envoy.extensions.filters.
                    inline_code: |
                      local library = require("library")
                      function envoy_on_request(request_handle)
                        local headers = request_handle:headers()
                        local metadata = request_handle:streamInfo():d
                        metadata:set("envoy.filters.http.lua", "reques
                          requestId = headers:get("my-request-id"),
                          method = headers:get(":method"),
                        })
                      end
                      function envoy_on_response(response_handle)
                        local requestInfoObj = response_handle:streamI
                        local requestId = requestInfoObj.requestId
                        local method = requestInfoObj.method
                        if (requestId == nil or requestId == '') and (
                          response_handle:logInfo("Adding request ID h
                            response_handle:headers():add("my-request-id
                          end
                        end
                      end
        - name: envoy.filters.http.router

```

将上述 YAML 保存为 `8-lab-1-lua-script-2.yaml`，然后用 `func-e` 运行它。

为了试用它，让我们向 `localhost:10000` 发送一个请求。

```
$ curl -v localhost:10000
...
[2021-11-23 23:14:18.206][2526][info][lua] [source/extensions/fi
< HTTP/1.1 200 OK
< content-length: 3
< content-type: text/plain
< my-request-id: usptcritlocbzsezhjmroule
< date: Tue, 23 Nov 2021 23:14:18 GMT
< server: envoy
<
* Connection #0 to host Localhost Left intact
200
```

输出将包括 `my-request-id` 和我们生成的、从 `library.lua` 文件调用的随机字符串。

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-03-04 16:36:07

## 实验 17：使用 Wasm 和 Go 扩展 Envoy

在这个实验中，我们将使用 [TinyGo](#)、[proxy-wasm-go-sdk](#) 和 [func-e CLI](#) 来构建和测试一个 Envoy Wasm 扩展。

我们将写一个简单的 Wasm 模块，为响应头添加一个头。稍后，我们将展示如何读取配置和添加自定义指标。我们将使用 Golang 并使用 TinyGo 编译器进行编译。

### 安装 TinyGo

让我们下载并安装 TinyGo。

```
 wget https://github.com/tinygo-org/tinygo/releases/download/v0.2  
 sudo dpkg -i tinygo_0.21.0_amd64.deb
```

你可以运行 `tinygo version` 来检查安装是否成功。

```
$ tinygo version  
tinygo version 0.21.0 linux/amd64 (using go version go1.17.2 and
```

### 为 Wasm 模块搭建脚手架

我们将首先为我们的扩展创建一个新的文件夹，初始化 Go 模块，并下载 SDK 依赖。

```
$ mkdir header-filter && cd header-filter  
$ go mod init header-filter  
$ go mod edit -require=github.com/tetratelabs/proxy-wasm-go-sdk@  
$ go mod download github.com/tetratelabs/proxy-wasm-go-sdk
```

接下来，让我们创建 `main.go` 文件，其中有我们 WASM 扩展的代码。

```

package main

import (
    "github.com/tetratelabs/proxy-wasm-go-sdk/proxywasm"
    "github.com/tetratelabs/proxy-wasm-go-sdk/proxywasm/types"
)

func main() {
    proxywasm.SetVMContext(&vmContext{})
}

type vmContext struct {
    // Embed the default VM context here,
    // so that we don't need to reimplement all the methods.
    types.DefaultVMContext
}

// Override types.DefaultVMContext.
func (*vmContext) NewPluginContext(contextID uint32) types.PluginContext {
    return &pluginContext{}
}

type pluginContext struct {
    // Embed the default plugin context here,
    // so that we don't need to reimplement all the methods.
    types.DefaultPluginContext
}

// Override types.DefaultPluginContext.
func (*pluginContext) NewHttpContext(contextID uint32) types.HttpContext {
    return &httpHeaders{contextID: contextID}
}

type httpHeaders struct {
    // Embed the default http context here,
    // so that we don't need to reimplement all the methods.
    types.DefaultHttpContext
    contextID uint32
}

func (ctx *httpHeaders) OnHttpRequestHeaders(numHeaders int, end bool) {
    proxywasm.LogInfo("OnHttpRequestHeaders")
    return types.ActionContinue
}

func (ctx *httpHeaders) OnHttpResponseHeaders(numHeaders int, end bool) {
    proxywasm.LogInfo("OnHttpResponseHeaders")
    return types.ActionContinue
}

func (ctx *httpHeaders) OnHttpStreamDone() {
    proxywasm.LogInfof("%d finished", ctx.contextID)
}

```

将上述内容保存在一个名为 `main.go` 的文件中。

让我们建立过滤器，检查是否一切正常。

```
tinygo build -o main.wasm -scheduler=none -target=wasi main.go
```

构建命令应该成功运行并生成一个名为 `main.wasm` 的文件。

我们将使用 `func-e` 来运行一个本地 Envoy 实例来测试我们构建的扩展。

首先，我们需要一个 Envoy 配置，它将配置扩展。

```

static_resources:
  listeners:
    - name: main
      address:
        socket_address:
          address: 0.0.0.0
          port_value: 10000
      filter_chains:
        - filters:
            - name: envoy.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.fi
                stat_prefix: ingress_http
                codec_type: auto
                route_config:
                  name: local_route
                  virtual_hosts:
                    - name: local_service
                      domains:
                        - "*"
                      routes:
                        - match:
                            prefix: "/"
                          direct_response:
                            status: 200
                            body:
                              inline_string: "hello world\n"
              http_filters:
                - name: envoy.filters.http.wasm
                  typed_config:
                    "@type": type.googleapis.com/udpa.type.v1.
                    type_url: type.googleapis.com/envoy.extens
                    value:
                      config:
                        vm_config:
                          runtime: "envoy.wasm.runtime.v8"
                          code:
                            local:
                              filename: "main.wasm"
                - name: envoy.filters.http.router
  admin:
    address:
      socket_address:
        address: 127.0.0.1
        port_value: 9901

```

将上述内容保存到 `8-lab-2-wasm-config.yaml` 文件。

Envoy 的配置在 10000 端口设置了一个监听器，返回一个直接响应 (HTTP 200)，正文是 `hello world`。在 `http_filters` 部分，我们配置了 `envoy.filters.http.wasm` 过滤器，并引用了我们之前建立的本地 WASM 文件 (`main.wasm`)。

让我们在后台用这个配置运行 Envoy。

```
func-e run -c 8-Lab-2-wasm-config.yaml &
```

Envoy 实例的启动应该没有任何问题。一旦启动，我们就可以向 Envoy 监听的端口（10000）发送一个请求。

```
$ curl -v localhost:10000
...
< HTTP/1.1 200 OK
< content-length: 13
< content-type: text/plain
< my-new-header: some-value-here
< date: Mon, 22 Jun 2021 17:02:31 GMT
< server: envoy
<
hello world
```

输出显示了两个日志条目：一个来自 `OnHttpRequestHeaders` 处理器，第二个来自 `OnHttpResponseHeaders` 处理器。最后一行是过滤器中的直接响应配置所返回的响应示例。

你可以通过用 `fg` 把进程带到前台，然后按 `CTRL+C` 停止代理。

## 在 HTTP 响应上设置附加头信息

让我们打开 `main.go` 文件，在响应头信息中添加一个头信息。我们将更新 `OnHttpResponseHeaders` 函数来做到这一点。

我们将调用 `AddHttpResponseHeader` 函数来添加一个新的头。更新 `OnHttpResponseHeaders` 函数，使其看起来像这样。

```
func (ctx *httpHeaders) OnHttpResponseHeaders(numHeaders int, en proxywasm.LogInfo("OnHttpResponseHeaders"))
    err := proxywasm.AddHttpResponseHeader("my-new-header", "some-
if err != nil {
    proxywasm.LogCriticalf(" failed to add response header: %v",
}
return types.ActionContinue
}
```

让我们重新建立扩展。

```
tinygo build -o main.wasm -scheduler=none -target=wasi main.go
```

现在我们可以用更新后的扩展来重新运行 Envoy 代理。

```
func-e run -c 8-Lab-2-wasm-config.yaml &
```

现在，如果我们再次发送一个请求（确保添加 `-v` 标志），我们将看到被添加到响应中的头。

```
$ curl -v localhost:10000
...
< HTTP/1.1 200 OK
< content-length: 13
< content-type: text/plain
< my-new-header: some-value-here
< date: Mon, 22 Jun 2021 17:02:31 GMT
< server: envoy
<
hello world
```

## 从配置中读取数值

在代码中硬编码这样的值从来不是一个好主意。让我们看看我们如何读取额外的头文件。

1. 将 `additionalHeaders` 和 `contextID` 添加到 `pluginContext` 结构体中：

```
type pluginContext struct {
    // 在这里嵌入默认的插件上下文。
    // 这样我们就不需要重新实现所有的方法了。
    types.DefaultPluginContext
    additionalHeaders map[string]string
    contextID uint32
}
```

2. 更新 `NewPluginContext` 函数以初始化数值。

```
func (*vmContext) NewPluginContext(contextID uint32) types.PluginContext {
    return &pluginContext{contextID: contextID, additionalHeaders: map[string]string{}}
}
```

3. 在 `OnPluginStart` 函数中，我们现在可以从 Envoy 配置中读入值，并将键 / 值对存储在 `extrapperHeaders` 映射中。

```

func (ctx *pluginContext) OnPluginStart(pluginConfigurationSize
    // Get the plugin configuration
    config, err := proxywasm.GetPluginConfiguration()
    if err != nil && err != types.ErrorStatusNotFound {
        proxywasm.LogCriticalf("failed to load config: %v", err)
        return types.OnPluginStartStatusFailed
    }

    // Read the config
    scanner := bufio.NewScanner(bytes.NewReader(config))
    for scanner.Scan() {
        line := scanner.Text()
        if strings.HasPrefix(line, "#") {
            continue
        }
        // Each line in the config is in the "key=value" format
        if tokens := strings.Split(scanner.Text(), "="); len(tokens) == 2 {
            ctx.additionalHeaders[tokens[0]] = tokens[1]
        }
    }
    return types.OnPluginStartStatusOK
}

```

为了访问我们设置的配置值，我们需要在初始化 HTTP 上下文时将该地图添加到 HTTP 上下文中。要做到这一点，我们需要先更新 `httpheaders` 结构。

```

type httpHeaders struct {
    // 在这里嵌入默认的http上下文。
    // 这样我们就不需要重新实现所有的方法了。
    types.DefaultHttpContext
    contextID uint32
    additionalHeaders map[string]string
}

```

然后，在 `NewHttpContext` 函数中，我们可以用来自插件上下文的附加 Header map 来实例化 `httpHeaders`。

```

func (ctx *pluginContext) NewHttpContext(contextID uint32) types
    return &httpHeaders{contextID: contextID, additionalHeaders: c
}

```

最后，为了设置 Header，我们修改了 `OnHttpResponseHeaders` 函数，遍历 `extraHeaders` 映射，并为每个项目调用 `AddHttpResponseHeader`。

```

func (ctx *httpHeaders) OnHttpResponseHeaders(numHeaders int, en
proxywasm.LogInfo("OnHttpResponseHeaders")

for key, value := range ctx.additionalHeaders {
    if err := proxywasm.AddHttpResponseHeader(key, value); err !=
        proxywasm.LogCriticalf("failed to add header: %v", err)
        return types.ActionPause
}
proxywasm.LogInfof("header set: %s=%s", key, value)
}

return types.ActionContinue
}

```

让我们再次重建这个扩展。

```
tinygo build -o main.wasm -scheduler=none -target=wasi main.go
```

另外，让我们更新配置文件，在过滤器配置（`configuration` 字段）中包括额外的头信息。

```

- name: envoy.filters.http.wasm
  typed_config:
    "@type": type.googleapis.com/udpa.type.v1.TypedStruct
    type_url: type.googleapis.com/envoy.extensions.filters.http.
    value:
      config:
        vm_config:
          runtime: "envoy.wasm.runtime.v8"
          code:
            local:
              filename: "main.wasm"
# ADD THESE LINES
  configuration:
    "@type": type.googleapis.com/google.protobuf.StringVal
    value: |
      header_1=somevalue
      header_2=secondvalue

```

随着过滤器的更新，我们可以重新运行代理。当你发送一个请求时，你会注意到我们在过滤器配置中设置的头信息被添加为响应头信息。

```

$ curl -v localhost:10000
...
< HTTP/1.1 200 OK
< content-length: 13
< content-type: text/plain
< header_1: somevalue
< header_2: secondvalue
< date: Mon, 22 Jun 2021 17:54:53 GMT
< server: envoy
...

```

## 添加一个指标

让我们添加另一个功能——计数器，每次有一个叫 `hello` 的 请求头被设置时都会增加。

首先，让我们更新 `pluginContext` 以包括 `helloHeaderCounter`。

```
type pluginContext struct {
    // 在这里嵌入默认的插件上下文。
    // 这样我们就不需要重新实现所有的方法了。
    types.DefaultPluginContext
    additionalHeaders map[string]string
    contextID uint32
    // 添加这一行
    helloHeaderCounter proxywasm.MetricCounter
}
```

有了结构中的计数器指标，我们现在可以在 `NewPluginContext` 函数中创建它。我们将调用头信息 `hello_header_counter`。

```
func (*vmContext) NewPluginContext(contextID uint32) types.PluginContext {
    return &pluginContext{contextID: contextID, additionalHeaders: map[string]string{}}
}
```

由于我们要检查传入的请求头以决定是否增加计数器，我们需要将 `helloHeaderCounter` 也添加到 `httpHeaders` 结构中。

```
type httpHeaders struct {
    // 在这里嵌入默认的http上下文。
    // 这样我们就不需要重新实现所有的方法了。
    types.DefaultHttpContext
    contextID uint32
    additionalHeaders map[string]string
    // 添加这一行
    helloHeaderCounter proxywasm.MetricCounter
}
```

另外，我们需要从 `pluginContext` 中获取计数器，并在创建新的 HTTP 上下文时设置它。

```
// 覆盖 types.DefaultPluginContext
func (ctx *pluginContext) NewHttpContext(contextID uint32) types.HttpContext {
    return &httpHeaders{contextID: contextID, additionalHeaders: map[string]string{}}
}
```

现在，我们已经将 `helloHeaderCounter` 一直输送到 `httpHeaders` 中，我们可以在 `OnHttpRequestHeaders` 函数中使用它。

```
func (ctx *httpHeaders) OnHttpRequestHeaders(numHeaders int, end
proxywasm.LogInfo("OnHttpRequestHeaders"))

_, err := proxywasm.GetHttpRequestHeader("hello")
if err != nil {
    // 如果头没有被设置，则忽略
    return types.ActionContinue
}

ctx.helloHeaderCounter.Increment(1)
proxywasm.LogInfo("hello_header_counter incremented")
返回 types.ActionContinue
}
```

在这里，我们要检查 `\"hello\"` 请求头是否被定义（注意，我们并不关心头的值），如果它被定义，我们就在计数器实例上调用 `Increment` 函数。否则，我们将忽略它，如果我们从 `GetHttpRequestHeader` 调用中得到一个错误，则返回 `ActionContinue`。

让我们再次重建这个扩展。

```
tinygo build -o main.wasm -scheduler=none -target=wasi main.go
```

然后重新运行 Envoy 代理。像这样发出几个请求。

```
curl -H "hello: something" localhost:10000
```

你会注意到像这样的日志 Envoy 日志条目。

```
wasm log: hello_header_counter incremented
```

你也可以使用 9901 端口的管理地址来检查指标是否被跟踪。

```
$ curl localhost:9901/stats/prometheus | grep hello
# TYPE envoy_hello_header_counter counter
envoy_hello_header_counter{} 1
```

Copyright © 2017-2022 | 基于 [CC 4.0 协议](#) 发布 | [jimmysong.io](http://jimmysong.io) all right reserved. Updated at 2022-03-04 16:51:42