# 1.0 Envoy简介

在Envoy介绍模块中，你将了解Envoy的概况，并通过一个例子了解Envoy的构建模块。在本模块结束时，你将通过运行Envoy的示例配置来了解Envoy。

# 1.1 什么是Envoy？

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它，准确地指出失败发生的地方，并了解它为什么失败。是网络不可靠吗？我们是否需要调整重试或超时？或者是业务逻辑问题或错误？

使这种调试的复杂性增加的是，服务可能使用不一致的跟踪和记录机制。这些问题使你很难确定问题，问题发生在哪里，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。

让调试这些网络问题变得更容易的是，将网络问题从应用程序堆栈中推出，由另一个组件来处理网络部分。这就是Envoy所能做的。

在它的一种部署模式中，我们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挎包部署模式。集中配置的Envoy的集合形成了一个透明的网状结构。

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。它的行为是相同的，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。

**滤波链**来自一个壳的想法，即一个操作的输出被输送到另一个操作中。比如说。

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。

推荐的服务对服务的配置在所有Envoy之间使用HTTP/2，以创建一个持久连接的网状结构，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支持一个路由子系统，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（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以及如何与它们进行通信。

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、监听套接字和加密材料的信息。届时，Envoy将尝试优雅地耗尽所有连接。

## 健康检查

与负载均衡器相关的一个特点是只将流量路由到健康和可用的上游服务。Envoy支持一个健康检查子系统，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还可以通过离群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）、请求影子（或流量镜像）、离群检测和请求对冲。

## 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状部署模型中所有服务之间的TLS终止（相互TLS）成为可能。

## 一流的可观察性

为了便于观察，Envoy会生成日志、指标和跟踪。Envoy目前支持 [statsd](https://github.com/etsy/statsd)(和兼容的提供者）作为所有子系统的统计汇。由于可扩展性，我们也可以在需要时插入不同的统计提供者。

## 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之间进行双向翻译。

# 1.2 Envoy的构建模块

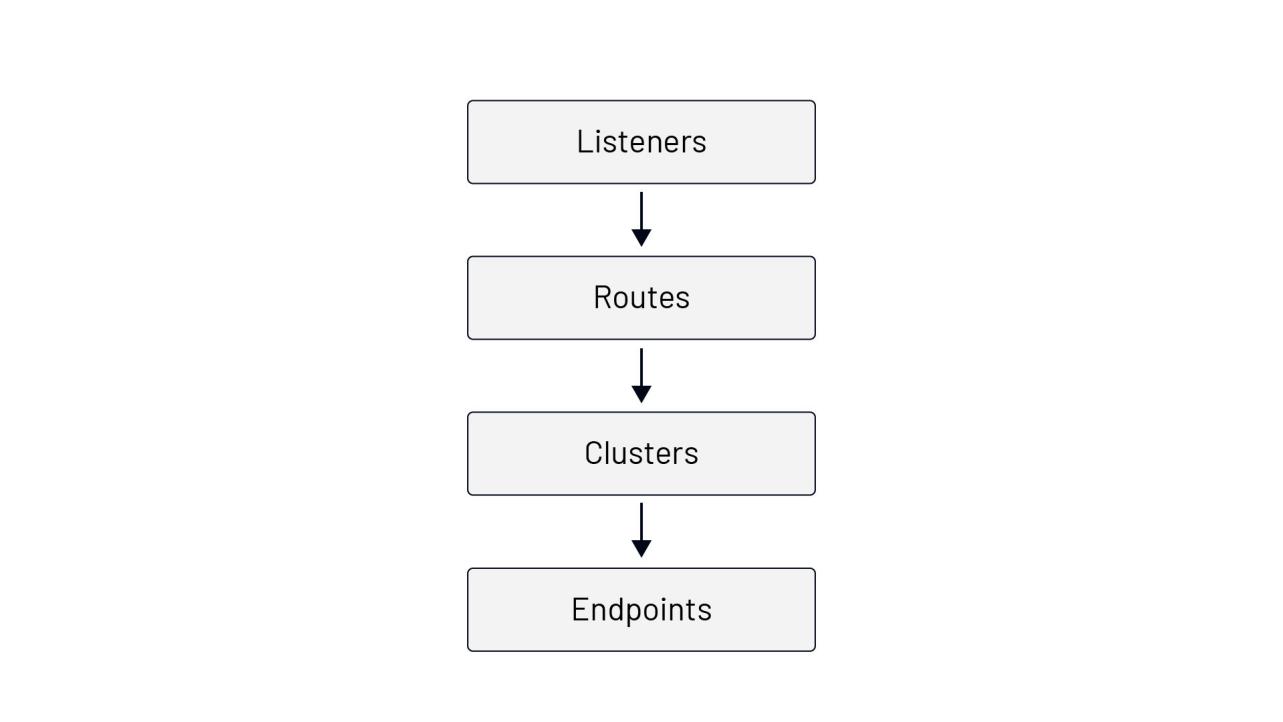
在这一课中，我们将解释Envoy的基本构建模块。

Envoy配置的根被称为引导配置。它包含了一些字段，我们可以在这里提供静态或动态的资源和高级别的Envoy配置（例如，Envoy实例名称、运行时配置、启用管理界面等等）。

为了开始学习，我们将主要关注静态资源，在课程的后面，我们将介绍如何配置动态资源。

Envoy输出许多统计数据，这取决于启用的组件和它们的配置。我们会在整个课程中提到不同的统计信息，在课程后面的专门模块中，我们会更多地讨论统计信息。

下图显示了通过这些概念的请求流。



这一切都从**监听器**开始。Envoy暴露的监听器是命名的网络位置，可以是一个IP地址和一个端口，也可以是一个Unix域套接字路径。Envoy通过监听器接收连接和请求。考虑一下下面的Envoy配置。

static\_resources:  
 listeners:  
 - name: listener\_0  
 地址。  
 socket\_address:  
 address: 0.0.0.0  
 port\_value: 10000  
 filter\_chains:[{}]

通过上面的Envoy配置，我们在地址和0.0.0.0端口10000上声明了一个名为listener\_0的监听器。这意味着Envoy正在监听0.0.0.0:10000的传入请求。

每个听众都有不同的部分需要配置。然而，唯一需要的设置是地址。上述配置是有效的，你可以用它来运行Envoy -- 尽管它没有用，因为所有的连接都会被关闭。

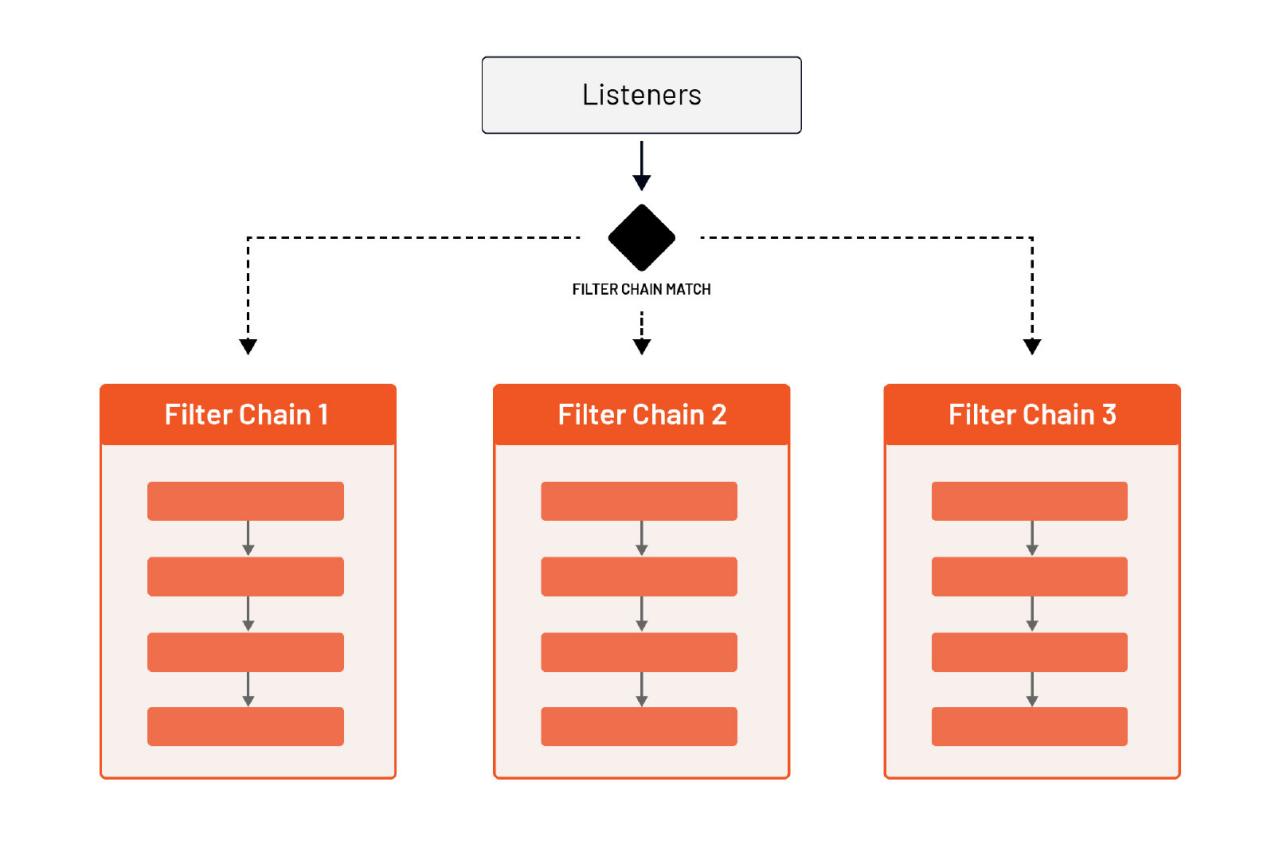
我们让filter\_chains字段为空，因为在接收数据包后不需要额外的操作。

为了进入下一个构件（路由），我们需要创建一个或多个网络过滤器链（filter\_chains），至少要有一个过滤器。

网络过滤器通常对数据包的有效载荷进行操作，查看有效载荷并对其进行解析。例如，Postgres网络过滤器解析数据包的主体，检查数据库操作的种类或其携带的结果。

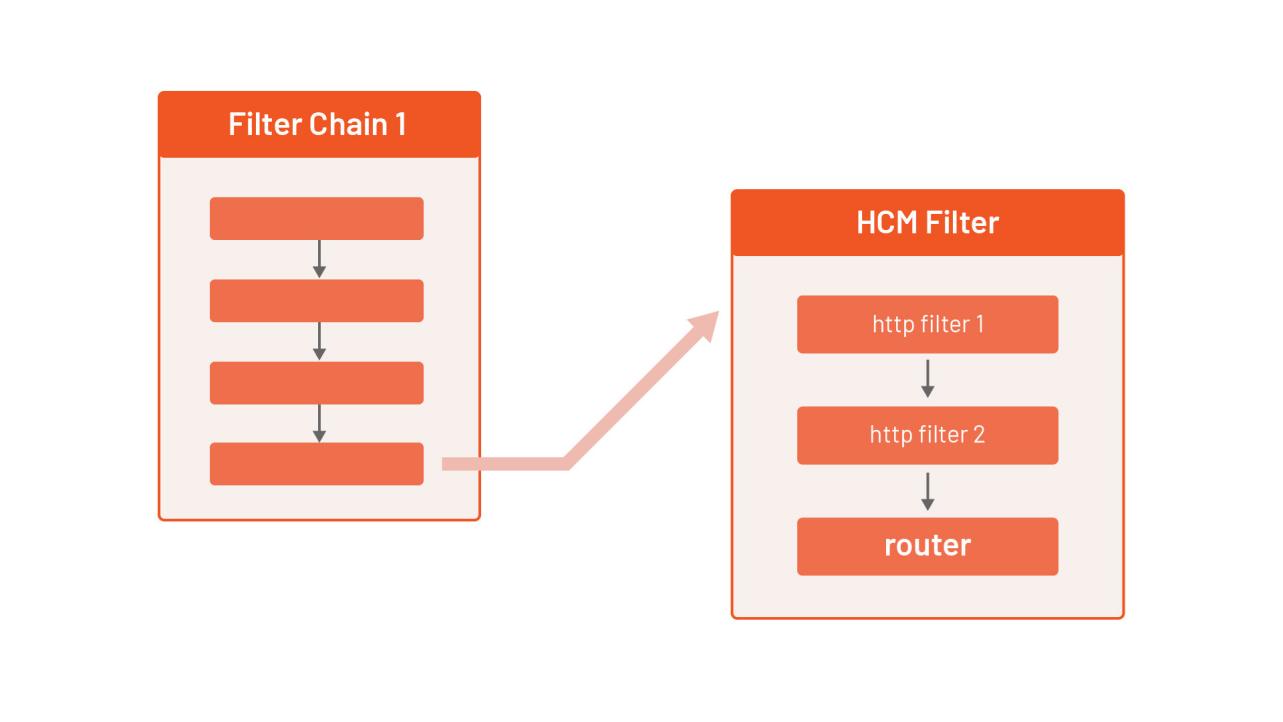
Envoy定义了三类过滤器：听众过滤器、网络过滤器和HTTP过滤器。听众过滤器在收到数据包后立即启动，通常对数据包的头信息进行操作。听众过滤器的例子是代理听众过滤器（提取PROXY协议头），或TLS检查器听众过滤器（检查流量是否为TLS，如果是，则从TLS握手中提取数据）。

每个通过监听器进来的请求可以流经多个过滤器。我们还可以写一个配置，根据传入的请求或连接属性选择不同的过滤器链。



一个特殊的、内置的网络过滤器被称为**HTTP连接管理器**过滤器或**HCM**。HCM过滤器能够将原始字节转换为HTTP级别的消息。它可以处理访问日志，生成请求ID，操作头信息，管理路由表，并收集统计数据。我们将在以后的课程中对HCM进行更详细的介绍。

就像我们可以为每个监听器定义多个网络过滤器（其中一个是HCM）一样，Envoy也支持在HCM过滤器中定义多个HTTP级过滤器。我们可以在名为http\_filters的字段下定义这些HTTP过滤器。



HTTP过滤器链中的最后一个过滤器必须是路由器过滤器（envoy.filters.HTTP.router）。路由器过滤器负责执行路由任务。这最终把我们带到了第二个构件--**路由**。

我们在HCM过滤器的route\_config字段下定义路由配置。在路由配置中，我们可以通过查看元数据（URI、头文件......）来匹配传入的请求，并在此基础上，定义流量的发送位置。

路由配置中的顶级元素是一个虚拟主机。每个虚拟主机都有一个名字，在发布统计数据时使用（不用于路由），还有一组被路由到它的域。

让我们考虑下面的路由配置和域的集合。

route\_config。  
 name: my\_route\_config  
 virtual\_hosts:  
 - name: tetrate\_host  
 域名。["tetrate.io"]  
 路线。  
 ...  
 - name: test\_hosts  
 域。["test.tetrate.io", "qa.tetrate.io"]  
 路由。  
 ...

如果传入请求的目的地是 tetrate.io（即 Host/Authority 标头被设置为其中一个值），则 tetrate\_hosts 虚拟主机中定义的路由将得到处理。

同样，如果Host/Authority标头包含test.tetrate.io或qa.tetrate.io，test\_hosts虚拟主机下的路由将被处理。使用这种设计，我们可以用一个监听器（0.0.0.0:10000）来处理多个顶级域。

如果你在数组中指定多个域，搜索顺序如下。

1. 精确的域名（例如：tetrate.io）。
2. 后缀域名通配符（如\*.tetrate.io）。
3. 前缀域名通配符（例如：tetrate.\*）。
4. 匹配任何域的特殊通配符（\*）。

在Envoy匹配域名后，是时候处理所选虚拟主机中的路由字段了。这是我们指定如何匹配一个请求，以及接下来如何处理该请求（例如，重定向、转发、重写、发送直接响应等）的地方。

我们来看看一个例子。

static\_resources:  
 listeners:  
 - name: listener\_0  
 地址。  
 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.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: hello\_world\_service  
 http\_filters:  
 - name: envoy.filters.http.router  
 route\_config:  
 name: my\_first\_route  
 virtual\_hosts:  
 - name: direct\_response\_service  
 域名。["\*"]  
 路线。  
 - 匹配。  
 prefix:"/"  
 direct\_response:  
 status: 200  
 body:  
 inline\_string:"yay"

配置的顶部部分与我们之前看到的一样。我们已经添加了HCM过滤器、统计前缀（hello\_world\_service）、单个HTTP过滤器（路由器）和路由配置。

在虚拟主机内，我们要匹配任何域名。在路由下，我们匹配前缀（/），然后我们可以发送一个响应。

当涉及到匹配请求时，我们有多种选择。

|  |  |  |
| --- | --- | --- |
| 航线匹配 | 描述 | 例子 |
| 前缀 | 前缀必须与:path头的开头相符。 | /hello与hello.com/hello、hello.com/helloworld和hello.com/hello/v1匹配。 |
| 路 | 路径必须与:path头完全匹配。 | /hello 匹配 hello.com/hello，但不匹配 hello.com/helloworld 或 hello.com/hello/v1 |
| safe\_regex | 所提供的重码必须与:path头匹配。 | /\例如，hello.com/123，但不能匹配hello.com/hello或hello.com/54321。 |
| 连接\_matcher | 匹配器只匹配CONNECT请求 |  |

一旦Envoy将请求与路由相匹配，我们就可以对其进行路由、重定向或返回一个直接响应。在这个例子中，我们通过direct\_response配置字段使用**直接响应**。

你可以把上述配置保存到envoy-direct-response.yaml中。

我们将使用一个名为 [func-e](https://func-e.io/)func-e允许我们选择和使用不同的Envoy版本。

我们可以通过运行以下命令下载func-e CLI。

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

现在我们用我们创建的配置运行Envoy。

func-e run -c envoy-direct-response.yaml

一旦Envoy启动，我们就可以向localhost:10000发送一个请求，以获得我们配置的直接响应。

$ curl localhost:10000  
yay

同样，如果我们添加一个不同的主机头（例如-H "Host: hello.com"）将得到相同的响应，因为hello.com主机与虚拟主机中定义的域相匹配。

在大多数情况下，从配置中直接发送响应是一个很好的功能，但我们会有一组端点或主机，我们将流量路由到这些端点或主机。在Envoy中做到这一点的方法是通过定义**集群**。

集群是一组接受流量的类似上游主机。这可以是你的服务所监听的主机或IP地址的列表。

例如，假设我们的hello world服务是在127.0.0.0:8000上监听。然后，我们可以用一个单一的端点创建一个集群，像这样。

集群。  
- name: hello\_world\_service  
 load\_assignment:  
 cluster\_name: hello\_world\_service  
 端点。  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: 127.0.0.1  
 port\_value: 8000

集群的定义与听众的定义在同一级别，使用集群字段。我们在路由配置中引用集群时，以及在排放统计数据时，都会使用集群。该名称在所有集群中必须是唯一的。

在load\_assignment字段下，我们可以定义要进行负载均衡的端点列表，以及负载均衡策略设置。

Envoy支持多种负载均衡算法（round-robin, Maglev, least-request, random），这些算法是由静态引导配置、DNS、动态xDS（CDS和EDS服务）以及主动/被动健康检查共同配置的。如果我们没有通过lb\_policy字段明确地设置负载均衡算法，它默认为round-robin。

端点字段定义了一组属于特定地域的端点。使用可选的定位字段，我们可以指定上游主机的运行位置，然后在负载均衡过程中使用（即，将请求代理到离调用者更近的端点）。

添加新的端点指示负载均衡器在一个以上的接收者之间分配流量。通常情况下，负载均衡器对所有端点一视同仁，但集群定义允许在端点内建立一个层次结构。

例如，端点可以有一个**权重**属性，这将指示负载均衡器与其他端点相比，向这些端点发送更多/更少的流量。

另一种层次结构类型是基于**地域性的**，通常用于定义故障转移架构。这种层次结构允许我们定义地理上比较接近的 "首选 "端点，以及在 "首选 "端点变得不健康的情况下应该使用的 "备份 "端点。

由于我们只有一个端点，所以我们还没有设置定位。在lb\_endpoints字段下，可以定义Envoy可以路由流量的实际端点。

集群配置也是我们可以配置以下可选功能的地方。

* 主动健康检查（health\_checks）。
* 断路器 (circuit\_breakers)
* 异常值检测（outlier\_detection）。
* 在处理上游的HTTP请求时有额外的协议选项
* 一组可选的网络过滤器，应用于所有出站连接等。

和监听器的地址一样，端点地址可以是一个套接字地址，也可以是一个Unix域套接字。在我们的例子中，我们使用一个套接字地址，并在127.0.0.1:8000为我们的服务定义端点。一旦选择了端点，请求就会被代理到该端点的上游。

让我们看看我们定义的集群是如何与其他配置结合起来的。

static\_resources:  
 listeners:  
 - name: listener\_0  
 地址。  
 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.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: hello\_world\_service  
 http\_filters:  
 - name: envoy.filters.http.router  
 route\_config:  
 name: my\_first\_route  
 virtual\_hosts:  
 - name: direct\_response\_service  
 域名。["\*"]  
 路线。  
 - 匹配。  
 prefix:"/"  
 航线。  
 cluster: hello\_world\_service  
 集群。  
 - name: hello\_world\_service  
 connect\_timeout:5s  
 load\_assignment:  
 cluster\_name: hello\_world\_service  
 端点。  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: 127.0.0.1  
 port\_value: 8000

我们已经添加了集群配置，我们没有使用direct\_response，而是使用路由字段并指定集群名称。

为了尝试这种配置，让我们在8000.NET端口启动一个hello-world Docker镜像。

docker run -dit -p 8000:3000 gcr.io/tetratelabs/hello-world:1.0.0

我们可以向127.0.0.1:8000发送一个请求，以检查我们是否得到 "Hello World "的响应。

接下来，让我们把上述Envoy配置保存到envoy-clusters.yaml中，并启动Envoy代理。

func-e run -c envoy-cluster.yaml

当Envoy代理启动时，向0.0.0.0:10000发送一个请求，让Envoy代理请求到hello world端点。

$ curl -v 0.0.0.0:10000  
...  
> get / http/1.1  
> 主机：localhost:10000  
> 用户代理: curl/7.64.0  
> 接受。\*/\*  
>  
< http/1.1 200 ok  
< 日期: Wed, 30 Jun 2021 23:53:47 GMT  
< 内容长度：11  
< 内容类型: text/plain; charset=utf-8  
< x-envoy-upstream-service-time: 0  
< 服务器：envoy  
<  
\* 与主机localhost的连接#0保持不变  
你好，世界

# 1.3 实验室的先决条件

从冗长的输出中，我们会注意到由Envoy代理设置的响应头x-envoy-upstream-service-time和server: envoy。

在课程中，我们将利用实际的实验室来实践我们所学的知识。

所有实验室都是在Linux Debian 10上开发和测试的，安装了以下工具。

* [Docker](https://docker.com)
* [Python 3](https://www.python.org/downloads/)
* [func-e](https://func-e.io/)
* [hey](https://github.com/rakyll/hey)

# 2.0 HTTP连接管理器(HCM)

在HCM模块中，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、操作头信息、配置超时、实现重试、请求镜像和速率限制。

在本模块结束时，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，操纵头文件等等。

# 2.1 HTTP连接管理器（HCM）介绍

HCM是一个网络级的过滤器，将原始字节翻译成HTTP级别的消息和事件（例如，收到的头文件，收到的主体数据等）。

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、头操作、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*。一个消息是一个完整的*帧*序列，映射到一个HTTP请求或响应消息。最后，一个*帧*是HTTP/2中最小的通信单位。每个帧都包含一个帧头，它至少可以识别该帧所属的流。帧可以携带有关HTTP头、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的*功能，将不同的线程协议翻译成流、请求、响应等的协议无关模型。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。

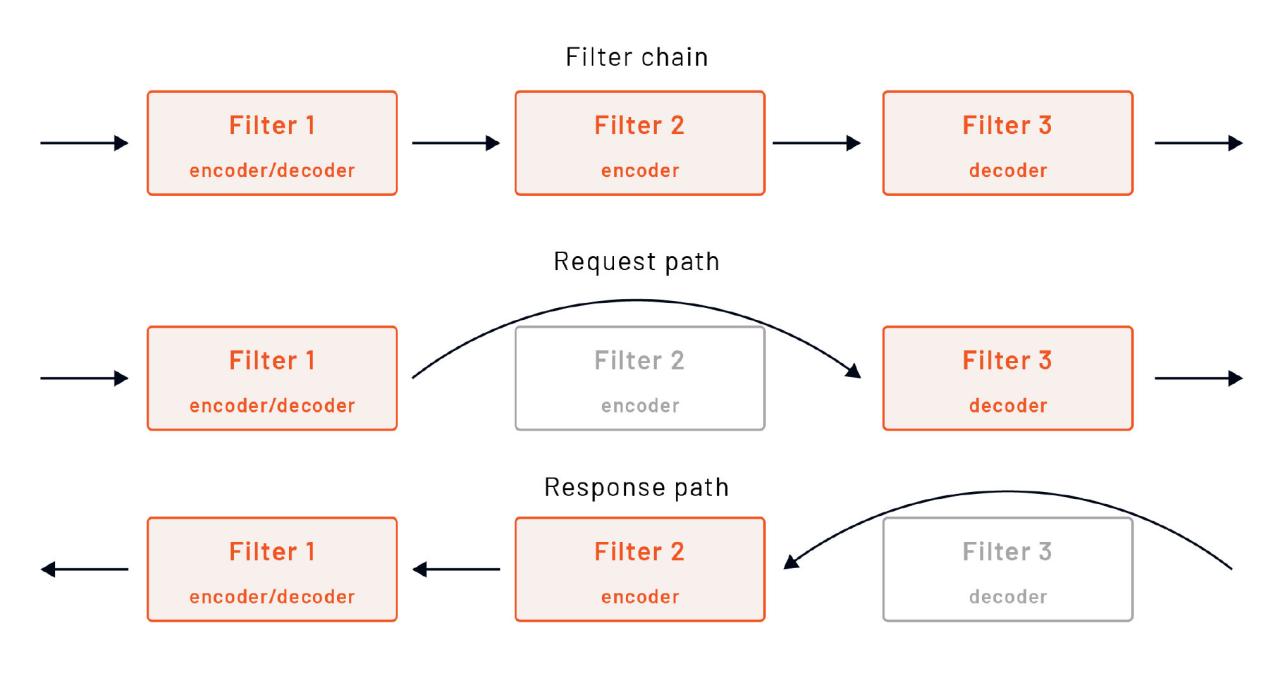
## HTTP过滤器

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：当HCM对请求流的部分进行解码时调用。
* 编码器：当HCM对响应流的部分进行编码时调用。
* 解码器/编码器：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。



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**状态。

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，都包含一个元数据字段，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（它们是不可改变的），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。一个叫做StreamInfo的对象提供了一种从地图上存储和检索类型对象的方法。

### 过滤器订购

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过滤器\_1，过滤器\_2，过滤器\_3。

在响应路径上，Envoy只调用编码器滤波器，但顺序相反。由于这三个滤波器都是解码器/编码器滤波器，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。你可以找到HTTP过滤器的完整列表 [here](https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters#config-http-filters).

# 2.2 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，主机或授权头），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  
 域名。["bar.io"]  
 路由。  
 - 匹配。  
 前缀。"/"  
 航线。  
 优先级：高  
 集群: bar\_io  
 - 名称：foo\_vhost  
 网域。["foo.io"]  
 路线。  
 - 匹配。  
 前缀。"/"  
 路线。  
 cluster: foo\_io  
 - 匹配。  
 前缀。"/api"  
 路由。  
 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. 主机或授权头被匹配到每个虚拟主机的域字段中指定的值。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路由下的条目。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  
 域名。["hello.io"]  
 路由。  
 - 匹配。  
 prefix:"/api"  
 路由。  
 cluster: hello\_io\_api  
 - 匹配。  
 prefix:"/api/v1"  
 路由。  
 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。

# 2.3 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 |
| 前缀 | 前缀必须与路径标头的开头相匹配。例如，前缀/api将匹配路径/api和/api/v1，而不是/。 |
| 路 | 路径必须与确切的路径标题相匹配（没有查询字符串）。例如，路径/api将匹配路径/api，但不匹配/api/v1或/。 |
| safe\_regex | 路径必须符合指定的正则表达式。例如，正则表达式^/products/\d+$将匹配路径/products/123或/products/321，但不是/products/hello或/api/products/123。 |
| 连接\_matcher | 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。

## 标题匹配

另一种匹配请求的方法是指定一组头文件。路由器根据路由配置中所有指定的头文件检查请求头文件。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标题。

**范围匹配**

range\_match检查请求头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。起始值是包容的，而范围的末端是排他的（[start, end)）。

- 匹配。  
 前缀。"/"  
 headers:  
 - name: minor\_version  
 range\_match:  
 start: 1  
 结束。 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。

**目前的比赛**

present\_match检查传入的请求中是否存在一个特定的头。

- 匹配。  
 前缀。"/"  
 headers:  
 - name: debug  
 present\_match: true

如果我们设置了调试头，无论头的值是多少，上面的片段都会评估为真。如果我们把present\_match的值设为false，我们就可以检查是否有头文件。

**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。

- 匹配。  
 前缀。"/"  
 头信息。  
 # 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`后缀\_match`以`\_1`结束  
 - name: suffix\_match  
 string\_match:  
 后缀。"\_1"  
 # 头部`contains\_match`包含值 "debug"  
 - name: contains\_match  
 string\_match:  
 包含。"debug"

**倒置匹配**

如果我们设置了invert\_match，匹配结果就会倒置。

- 匹配。  
 前缀。"/"  
 headers:  
 - 名称：版本  
 range\_match:   
 start: 1  
 结束。 6  
 invert\_match: true

上面的片段将检查版本头中的值是否在1和5之间；然而，由于我们添加了invert\_match字段，它颠倒了结果，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。比如说。

- 匹配。  
 前缀。"/"  
 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测试。如果我们设置了env头，并且它不包括字符串测试，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。过滤器将检查来自路径头的查询字符串，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。

- 匹配。  
 前缀。"/"  
 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 |
| 确实 | 必须与查询参数的精确值相匹配。 |
| 前缀 | 前缀必须符合查询参数值的开头。 |
| 后缀 | 后缀必须符合查询参数值的结尾。 |
| safe\_regex | 查询参数值必须符合指定的正则表达式。 |
| 包含 | 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。

- 匹配。  
 前缀。"/"  
 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

我们可以在路由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。

比如说。

- 匹配。  
 前缀。"/"  
 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。呈现字段检查证书是否被呈现。验证字段检查证书是否被验证。

比如说。

- 匹配。  
 前缀。"/"  
 tls\_context:  
 呈现: true  
 验证: true

如果一个证书既被出示又被验证，上述匹配评估为真。

# 2.4 交通分流

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，第二种是使用加权群组。

## 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运行时间\_分数的配置。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  
 域名。["hello.io"]  
 路由。  
 - 匹配。  
 prefix:"/"  
 runtime\_fraction:  
 default\_value:  
 numerator: 90  
 分母。HUNDRED  
 路线。  
 cluster: hello\_v1  
 - 匹配。  
 prefix:"/"  
 路由。  
 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，分母]范围内生成一个随机数（例如，在我们的案例中是[0，100]）。如果随机数小于分母值，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。

如果随机数大于分母，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，没有一个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母值。比如说。

route\_config:  
 virtual\_hosts:  
 - name: hello\_vhost  
 域名。["hello.io"]  
 路由。  
 - 匹配。  
 prefix:"/"  
 runtime\_fraction:  
 default\_value:  
 numerator: 0  
 分母。HUNDRED  
 runtime\_key: routing.hello\_io  
 路由。  
 cluster: hello\_v1  
 - 匹配。  
 prefix:"/"  
 路由。  
 cluster: hello\_v2  
...  
layered\_runtime:  
 层。  
 - 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，它将看到提供了运行时键，并将使用该值而不是分母值。有了运行时密钥，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。

## 使用加权群组进行交通分流

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簇（weighted\_clusters），而不是设置一个簇（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  
 域名。["hello.io"]  
 路由。  
 - 匹配。  
 prefix:"/"  
 航线。  
 weighted\_clusters:  
 集群。  
 - name: hello\_v1  
 权重。 90  
 - name: hello\_v2  
 权重。 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  
 域名。["hello.io"]  
 路由。  
 - 匹配。  
 prefix:"/"  
 航线。  
 weighted\_clusters:  
 runtime\_key\_prefix: routing.hello\_io  
 集群。  
 - name: hello\_v1  
 权重。 90  
 - name: hello\_v2  
 weight: 10  
...  
layered\_runtime:  
 层数。  
 - 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  
 域名。["hello.io"]  
 路由。  
 - 匹配。  
 prefix:"/"  
 航线。  
 weighted\_clusters:  
 runtime\_key\_prefix: routing.hello\_io  
 total\_weight: 15  
 集群。  
 - name: hello\_v1  
 权重。 5  
 - name: hello\_v2  
 权重。 5  
 - name: hello\_v3  
 权重。 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。

# 2.5 头部操作

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头文件。例外的情况是，如果我们使用Wasm扩展。然后，我们可以修改 :authority header，例如。

不可变的头是伪头（前缀为:，如:scheme）和主机头。此外，诸如 :path 和 :authority 这样的头信息可以通过前缀重写、重码重写和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. 路由级头文件
3. 虚拟主机级标头
4. 全局级标题

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头。

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: 要添加到响应中的头信息数组。
* response\_headers\_to\_remove: 要从响应中移除的头信息数组。
* request\_headers\_to\_add: 要添加到请求中的头信息数组。
* request\_headers\_to\_remove: 要从请求中删除的头信息数组。

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（%）为界。支持的变量名称包括%DOWNSTREAM\_REMOTE\_ADDRESS%、%UPSTREAM\_REMOTE\_ADDRESS%、%START\_TIME%、%RESPONSE\_FLAGS%和更多。你可以找到完整的变量列表 [here](https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_conn_man/headers#custom-request-response-headers).

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"  
 值。"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"  
 值。"from\_v\_host"  
 域。["hello.io"]  
 路线。  
 - 匹配。  
 prefix:"/"  
 航线。  
 cluster: hello  
 response\_headers\_to\_add:  
 - header:   
 key: "route\_header"  
 值。"%downstream\_remote\_address%"  
 - 匹配。  
 前缀。"/api"  
 路线。  
 cluster: hello\_api  
 response\_headers\_to\_add:  
 - header:   
 key: "api\_route\_header"  
 值。"api-value"  
 - header:  
 key: "header\_1"  
 值。"this\_will\_be\_overwritten"

## 标准标头

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。

':权限', 'localhost:10000'  
':路径', '/'  
':方法', 'GET  
':方案', 'http'  
':user-agent', 'curl/7.64.0'.  
'接受'，'\*/\*'  
'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'.  
'内容类型', 'text/html; charset=utf-8'  
'content-length', '563'.  
'etag', 'W/"233-b+4UpNDbOtHFiEpLMsDEDK7iTeI"'  
'date', 'Fri, 16 Jul 2021 21:59:52 GMT'.  
'x-envoy-upstream-service-time', '2  
'服务器', 'envoy'

下表解释了Envoy在解码或编码过程中设置的不同头信息。

|  |  |
| --- | --- |
| 标题 | 描述 |
| :计划 | 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-forwaded-proto头值设置的) |
| 用户-代理 | 通常由客户端设置，但在启用 add\_user\_agent 时可以修改（仅当头文件尚未设置时）。该值由--服务集群命令行选项决定。 |
| x-forwarded-proto | 标准头，用于识别客户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。 |
| 服务器 | 设置为server\_name字段中指定的值（默认为envoy）。 |

根据不同的场景，Envoy会设置或消耗一系列其他头信息。当我们在课程的其余部分讨论这些场景和功能时，我们会叫出不同的头信息。

## 标头消毒

头部消毒是一个过程，包括出于安全原因添加、删除或修改请求头。有一些头信息，Envoy有可能会进行消毒处理。

|  |  |
| --- | --- |
| 标题 | 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-service-cluster命令行选项决定，要求user\_agent设置为true。 |
| x-envoy-downstream-service-node | 和前面的头一样，数值由--服务--节点选项决定。 |
| 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-external-address | 受信任的客户地址（关于如何确定，详见下面的XFF）。 |
| x-envoy-force-trace | 强制收集的痕迹。 |
| 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 | 指定重试政策。 |
| 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）。 |
| x-envoy-upstream-rq-timeout-ms | 覆盖路由配置超时。 |
| x-forwarded-client-certif | 表示一个请求流经的所有客户/代理中的部分证书信息。 |
| x-forwarded-for | 表示IP地址请求通过了。更多细节见下面的XFF。 |
| x-forwarded-proto | 设置来源协议（http或https）。 |
| x-request-id | 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消毒，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-forwaded-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机构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**。

**受信任的客户地址**

一个受信任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会认为准确的节点。

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并1使用与上面相同的例子，受信任的地址将是IP2和IP3。

Envoy使用受信任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](https://datatracker.ietf.org/doc/html/rfc1918)或 [RFC4193](https://datatracker.ietf.org/doc/html/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。

...  
- 过滤器。  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 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。

':权限', 'localhost:10000'  
':路径', '/'  
':方法', 'GET  
':方案', 'http'  
':user-agent', 'curl/7.64.0'.  
'接受', '\*/\*'  
'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' 。

这些标头中的大部分与我们在标准标头例子中看到的相同。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，以下头信息会被发送到上游。

':authority', '35.224.50.133:10000'  
':路径', '/'  
':方法', 'GET  
':方案', 'http'  
':user-agent', 'curl/7.64.1'.  
'接受'，'\*/\*'  
'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，而不只是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，并用于基于起源客户端IP地址的分析。

# 2.6 回复修改

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**，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 [multiple filters](https://www.envoyproxy.io/docs/envoy/latest/api-v3/config/accesslog/v3/accesslog.proto#envoy-v3-api-msg-config-accesslog-v3-accesslogfilter)来匹配状态代码、持续时间、头文件、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）和头文件（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，或添加头文件。

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.http\_connection\_manager.v3.HttpConnectionManager  
 local\_reply\_config:  
 mappers:  
 - filter:  
 status\_code\_filter:  
 比较。  
 op:EQ  
 值。  
 default\_value: 503  
 runtime\_key: some\_key  
 headers\_to\_add:  
 - header:  
 key: "service"  
 值。"不可用"  
 附加: false  
 status\_code:401  
 body:  
 inline\_string:"不允许"

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。

# 2.7 请求ID的生成

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指出延迟来源至关重要。

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，应该是由于客户端的请求ID加入而强制追踪。

之所以选择第14个nibble，是因为它在4设计上被固定为。因此，表示4一个默认的UUID，没有跟踪状态，例如7b674932-635d-\*\*4\*\*ceb-b907-12674f8c7267。

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.uuid.v3.UuidRequestIdConfig  
 pack\_trace\_reason: false  
 use\_request\_id\_for\_trace\_sampling: false  
 http\_filters:  
 - name: envoy.filters.http.router  
...

打包跟踪原因是一个布尔值，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。

# 2.8 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--可以在较低层次上覆盖，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，如下图所示。

...  
过滤器。  
- name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 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。

还有一个与头文件有关的超时，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5秒）。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。

如果我们往下走，下一站就是路由超时。如前所述，路线层面的超时可以覆盖HCM的超时和一些额外的超时。

路由超时是指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超时域设置的值更短的超时。

# 2.9 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头文件（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。

1. 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。

1. 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。这可以帮助防止重试流量对流量的贡献。

1. 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  
 域名。["\*"]  
 路线。  
 - 匹配。  
 路径。/status/500  
 路由。  
 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 0 269 269 "-" "curl/7.64.0" "1ae9ffe2-21f2-43f7-ab80-79be4a95d6d4" "localhost:10000" "127.0.0.1:5000"

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（retry\_on）。 | 描述 |
| 5xx | 在5xx响应代码或上游不响应时重试（包括connect-failure和refused-stream）。 |
| 闸门错误 | 对 502、 503、 或响应504代码进行重试。 |
| 重置 | 如果上游根本没有回应，则重试。 |
| 连接失败 | 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。 |
| Envoy-ratelimited | 如果存在x-envoy-ratelimited头，则重试。 |
| 可检索-4xx | 如果上游响应的是可收回的4xx响应代码（目前409只有HTTP），则重试。 |
| 拒绝流 | 如果上游以REFUSED\_STREAM错误代码重置流，则重试。 |
| 可恢复的状态代码 | 如果上游响应的任何响应代码与x-envoy-retriable-status-codes头中定义的代码相匹配（例如，以逗号分隔的整数列表，例如 "502,409"），则重试。 |
| 可检索的头文件 | 如果上游响应包括任何在x-envoy-retriable-header-names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 predicate 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  
 域名。["\*"]  
 路线。  
 - 匹配。  
 路径。/status/500  
 路由。  
 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。

## 要求进行套期保值

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，我们通常为同位素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  
 域名。["\*"]  
 hedge\_policy:  
 hedge\_on\_per\_try\_timeout: true  
 路线。  
 - 匹配。  
 ...

# 2.10 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将流量从一个集群影射到另一个集群。

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 "发射并忘记 "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 "fire and forget "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。

阴影请求中的授权/主机头信息将被添加-阴影字符串。

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  
 域名。["\*"]  
 路线。  
 - 匹配。  
 前缀。/  
 路由。  
 cluster: httpbin  
 request\_mirror\_policies:  
 cluster: mirror\_httpbin  
 runtime\_fraction:  
 default\_value:  
 分子。 100  
 ...

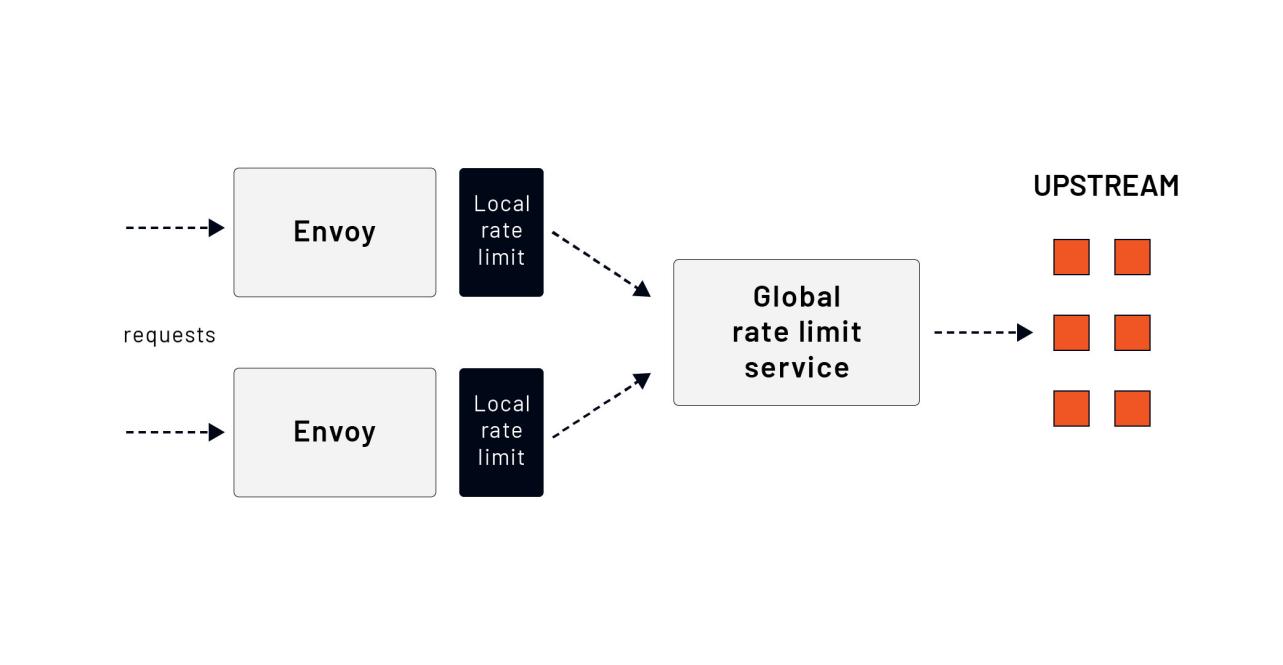
上述配置将100%地接收发送到集群httpbin的传入请求，并将其镜像到镜像\_httpbin。

# 2.11速率限制

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，我们就说进行呼叫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。

全局和本地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**上流的访问。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，本地速率限制适用于每一个Envoy实例。

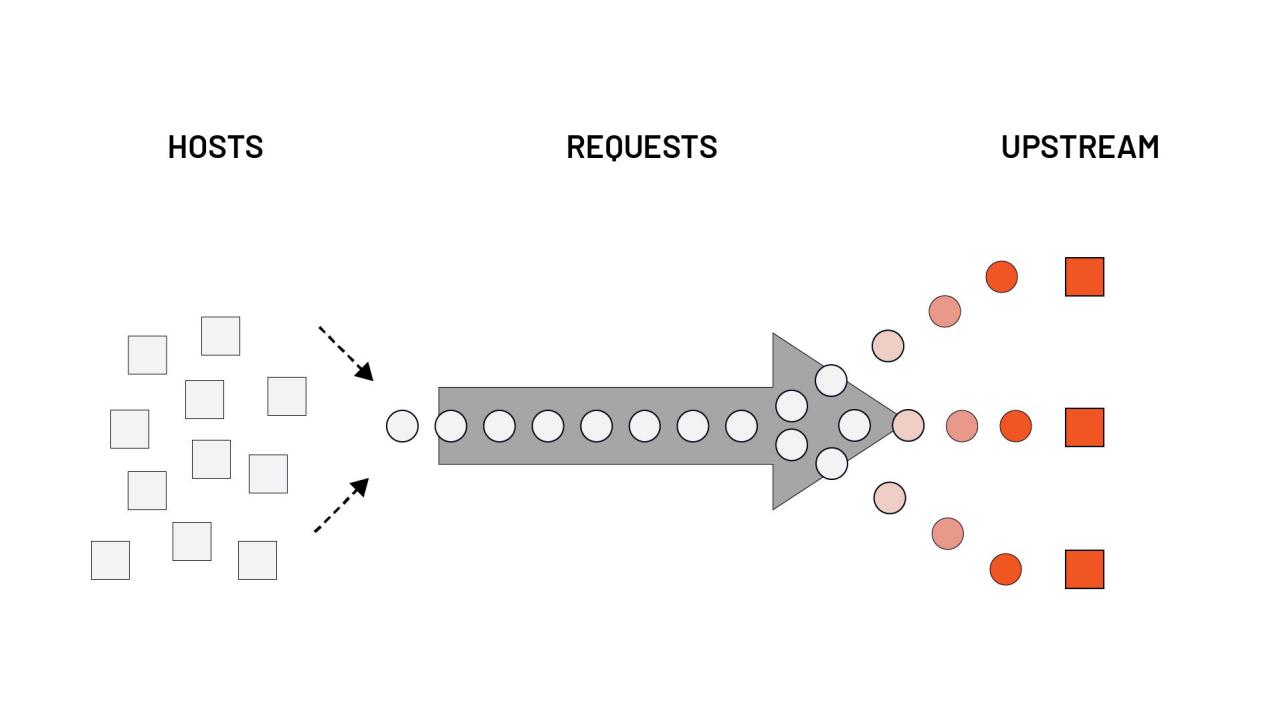


本地和全球费率限制都可以一起使用，Envoy分两个阶段应用它们。首先，应用本地速率限制，然后是全球速率限制。

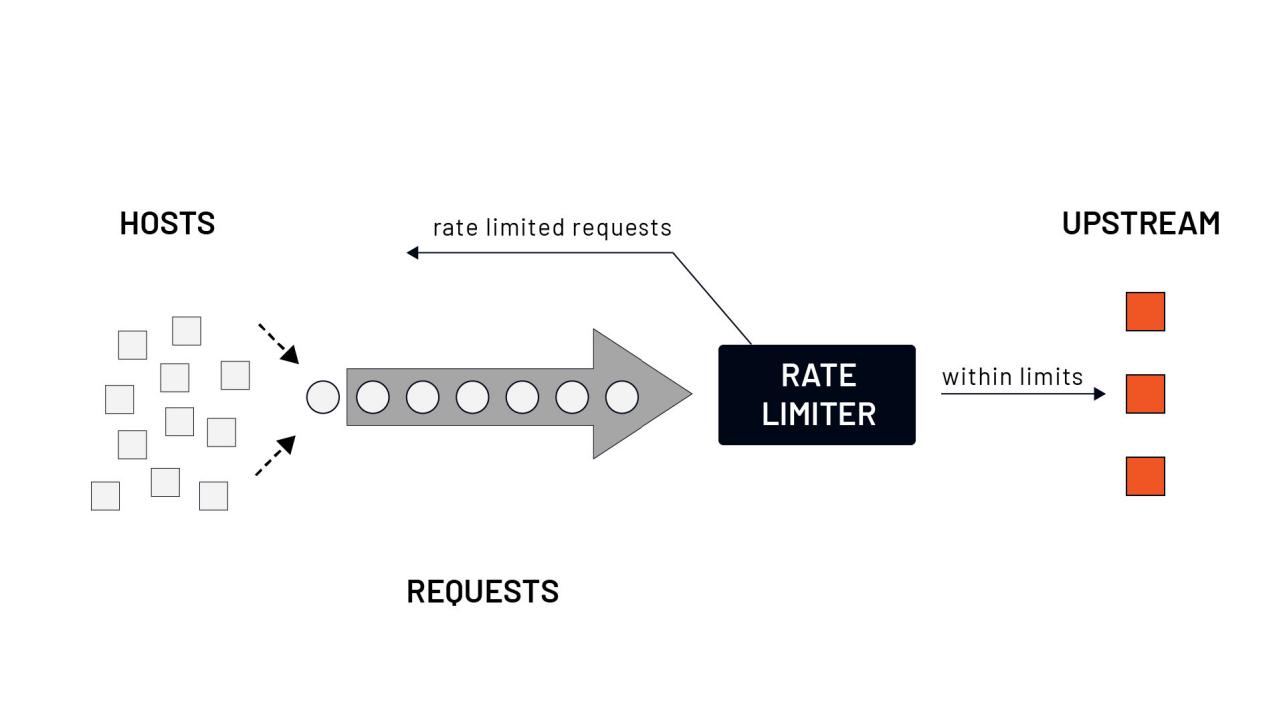
我们将在接下来的章节中深入研究全局和局部速率限制，并解释这两种情况如何工作。

# 2.12 全球速率限制

当许多主机向少数上游服务器发送请求，且平均延迟较低时，全局或分布式速率限制很有用。



由于主机不能快速处理这些请求，请求就会变得滞后。在这种情况下，众多的下游主机可以压倒少数的上游主机。全局速率限制器有助于防止串联故障。



Envoy与任何实现定义的RPC/IDL协议的外部速率限制服务集成。该服务的参考实现使用Go、gRPC和Redis作为其后端，可用于 [here](https://github.com/envoyproxy/ratelimit).

Envoy调用外部速率限制服务（例如，在Redis中存储统计信息并跟踪请求），以得到该请求是否应该被速率限制的答案。

使用外部速率限制服务，我们可以将限制应用于一组服务，或者，如果我们谈论的是服务网，则应用于网中的所有服务。我们可以控制进入网状结构的请求数量，作为一个整体。

为了控制单个服务层面的请求率，我们可以使用本地速率限制器。本地速率限制器允许我们对每个服务有单独的速率限制。本地和全局速率限制通常是一起使用的。

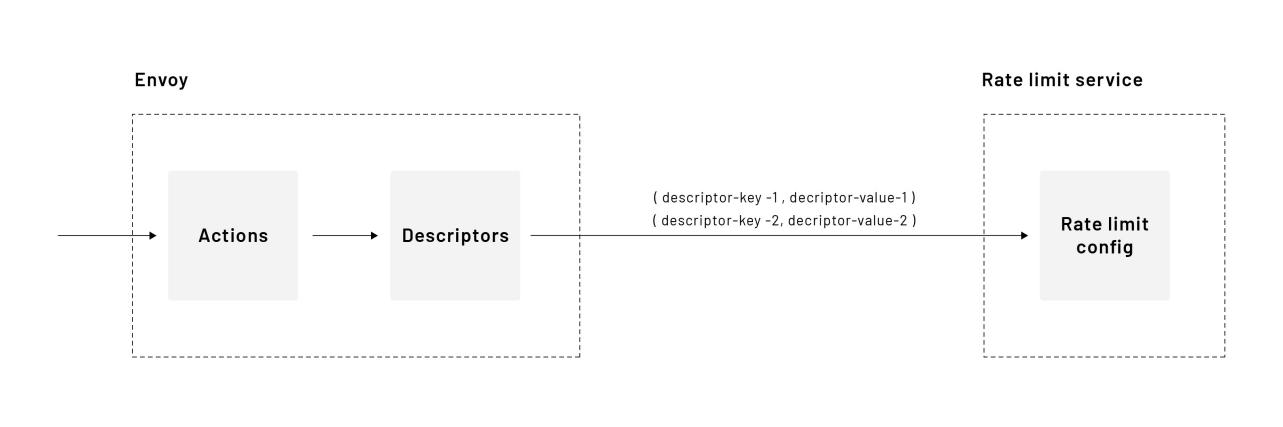
## 配置全局速率限制

在配置全局速率限制时，我们必须设置两个部分--客户端（Envoy）和服务器端（速率限制服务）。

我们可以在Envoy侧将速率限制服务配置为**网络级速率限制过滤器**或**HTTP级速率限制过滤器**。

当在网络层面上使用速率限制过滤器时，Envoy对我们配置了过滤器的监听器的每个新连接调用速率限制服务。同样，使用HTTP级别的速率限制过滤器，Envoy对安装了过滤器的监听器上的每个新请求调用速率限制服务，**并且**路由表指定应调用全球速率限制服务。所有到目标上游集群的请求以及从发起集群到目标集群的请求都可以被限制速率。

在配置速率限制服务之前，我们需要解释**行动**、**描述符**（键/值对）和描述**符列表的**概念。



在路由或虚拟主机层面的Envoy配置中，我们定义了一组**动作**。每个动作都包含一个速率限制动作的列表。让我们考虑下面的例子，在虚拟主机级别上定义速率限制。

rate\_limits:  
- 行动。  
 - header\_value\_match:  
 descriptor\_value: get\_request  
 头信息。  
 - name: :method  
 prefix\_match:GET  
 - header\_value\_match:  
 Descriptionor\_value: path  
 头信息。  
 - name: :path  
 prefix\_match:/api  
- 行动。  
 - header\_value\_match:  
 描述符\_值：post\_request  
 头信息。  
 - name: :method  
 prefix\_match:POST  
- 行动。  
 - header\_value\_match:  
 描述符\_值: get\_request  
 头信息。  
 - name: :method  
 prefix\_match:GET

上面的片段定义了三个独立的动作，其中包含速率限制动作。Envoy将尝试将请求与速率限制动作相匹配，并生成描述符发送到速率限制服务。如果Envoy不能将任何一个费率限制动作与请求相匹配，则不会创建描述符--即所有费率限制动作都必须相匹配。

例如，如果我们收到一个到/api的GET请求，第一个动作与两个速率限制动作相匹配；因此会创建一个如下描述符。

("header\_match": "get\_request"), ("header\_match": "path")

第二个动作是不会匹配的。然而，最后一个也会匹配。因此，Envoy将向速率限制服务发送以下描述符。

("header\_match": "get\_request"), ("header\_match": "path")  
("header\_match": "get\_request")

让我们来看看另一个满足以下要求的客户端配置的例子。

* 对/users的POST请求被限制在每分钟10个请求。
* 对/users的请求被限制在每分钟20个请求。
* 带有dev: true头的/api请求被限制在每秒10个请求的速率。
* 向/api发出的带有dev: false头的请求被限制在每秒5个请求。
* 对/api的任何其他请求都没有速率限制

请注意，这次我们是在路由层面上定义速率限制。

航线。  
- 匹配。  
 prefix:"/users"  
 路由。  
 cluster: some\_cluster  
 rate\_limits:  
 - 行动。  
 - generic\_key:  
 descriptor\_value: users  
 - header\_value\_match:  
 描述符\_值: post\_request  
 headers:  
 - name: ":method"  
 exact\_match:POST  
 - 行动。  
 - generic\_key:  
 描述符\_值: users  
- 匹配。  
 prefix:"/api"  
 路线。  
 cluster: some\_cluster  
 rate\_limits:  
 - 行动。  
 - generic\_key:  
 descriptor\_value: api  
 - request\_headers:  
 header\_name: dev  
 描述符\_key: dev\_request  
...  
http\_filters:  
- name: envoy.filters.http.ratelimit  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.http.ratelimit.v3.RateLimit  
 domain: some\_domain  
 enable\_x\_ratelimit\_headers:draft\_version\_03  
 rate\_limit\_service:  
 transport\_api\_version:V3  
 grpc\_service:  
 envoy\_grpc:  
 cluster\_name: rate-limit-cluster

上述配置包含每条路由的rate\_limits配置和envoy. filters.http.ratelimit过滤器配置。过滤器的配置指向速率限制服务的上游集群。我们还设置了域名（domain）和 enabled\_x\_ratelimit\_headers 字段，指定我们要使用 x-ratelimit headers。域名允许我们按任意的域名来隔离一组速率限制配置。

如果我们看一下路由中的速率限制配置，注意到我们是如何拆分行动以匹配我们想要设置的不同速率限制的。例如，我们有一个带有api通用密钥和请求头的动作。然而，在同一个配置中，我们也有一个只设置了通用密钥的动作。这使得我们可以根据这些动作配置不同的速率限制。

让我们把行动翻译成描述符。

GET /users --> ("generic\_key": "users")  
  
POST /users --> ("generic\_key": "users"), ("header\_match": "post\_request")  
  
GET /api   
dev: some\_header\_value --> ("generic\_key": "api"), ("dev\_request": "some\_header\_value")

header\_match和request\_headers之间的区别是，对于后者，我们可以根据特定的头信息值来创建速率限制（例如，dev: true或dev: something，因为头信息的值成为描述符的一部分）。

在速率限制服务方面，我们需要开发一个配置，根据Envoy发送的描述符来指定速率限制。

例如，如果我们向/users发送一个GET请求，Envoy会向速率限制服务发送以下描述符。("generic\_key": "users")。然而，如果我们发送一个POST请求，描述符列表看起来像这样。

("generic\_key": "users"), ("header\_match": "post\_request")

速率限制服务配置是分层次的，允许匹配嵌套描述符。让我们看看上述描述符的速率限制服务配置会是什么样子。

领域: some\_domain  
描述符。  
- key: generic\_key  
 值: users  
 rate\_limit:  
 单位。MINUTE  
 requests\_per\_unit: 20  
 descriptors:  
 - key: header\_match  
 值: post\_request  
 rate\_limit:  
 单位。MINUTE  
 requests\_per\_unit: 10  
- key: generic\_key  
 值: api  
 描述符。  
 - key: dev\_request  
 值: true  
 rate\_limit:  
 单位。SECOND  
 requests\_per\_unit: 10  
 - key: dev\_request  
 值: false  
 rate\_limit:  
 单位。SECOND  
 requests\_per\_unit: 5

我们之前在Envoy方面的配置中提到了域名值。现在我们可以看看如何使用域名。我们可以在整个代理机群中使用相同的描述符名称，但要用域名来分隔它们。

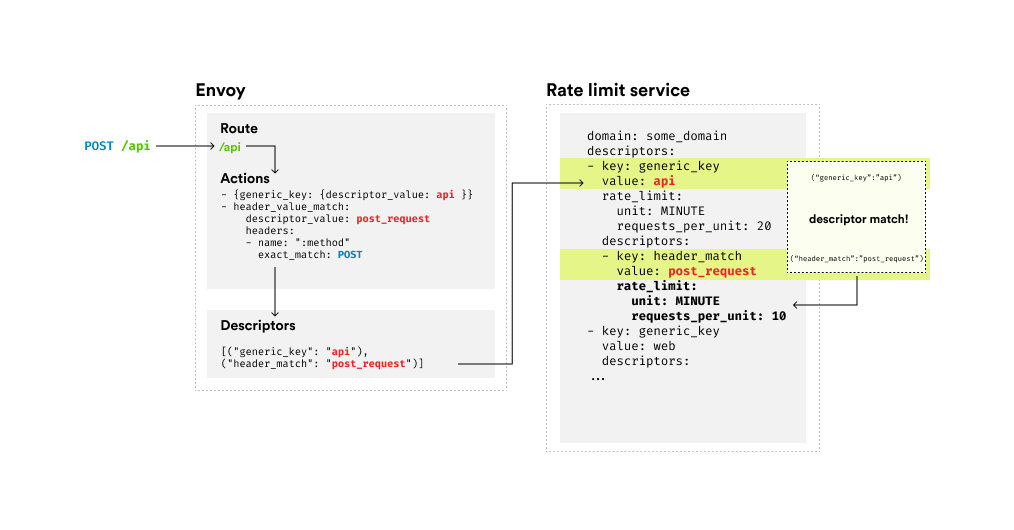
让我们看看速率限制服务上的匹配对不同请求是如何工作的。

|  |  |  |  |
| --- | --- | --- | --- |
| 收到的请求 | 生成的描述符 | 费率限制 | 解释 |
| GET /users | ("generic\_key": "users") | 20 req/min | key users与配置中的第一层相匹配。由于配置中的第二层（header\_match）没有包括在描述符中，所以使用了user键的速率限制。 |
| POST /users | ("generic\_key": "users"), ("header\_match": "post\_request") | 10 req/min | 发送的描述符和header\_match一样匹配用户，所以使用header\_match描述符下的速率限制。 |
| GET /api | ("generic\_key": "api") | 无费率限制 | 我们只有api描述符的第一级匹配。然而，并没有配置速率限制。为了执行速率限制，我们需要第二级描述符，这些描述符只有在传入的请求中存在头文件dev时才会被设置。 |
| GET /api dev: true | ("generic\_key": "api"), ("dev\_request": "true") | 10 req/秒 | 列表中的第二个描述符与配置中的第二层相匹配（即我们匹配api，然后也匹配dev\_request：真值）。 |
| GET /api dev: false | ("generic\_key": "api"), ("dev\_request": "false") | 5次/秒 | 列表中的第二个描述符与配置中的第二层相匹配（即我们匹配api，然后也匹配dev\_request：真值）。 |
| GET /api 开发商：你好 | ("generic\_key": "api"), ("dev\_request": "hello") | 无费率限制 | 列表中的第二个描述符与配置中的任何二级描述符都不匹配。 |

除了我们在上面的例子中使用的动作外，下表显示了我们可以用来创建描述符的其他动作。

|  |  |
| --- | --- |
| 行动名称 | 描述 |
| 来源\_集群 | 源集群的速率限制 |
| 目的地\_群集 | 目的地集群的速率限制 |
| request\_headers | 对请求头的速率限制 |
| 远程\_地址 | 远程地址的速率限制 |
| 通用钥匙 | 对一个通用钥匙的速率限制 |
| header\_value\_match | 对请求头的存在进行速率限制 |
| 元数据 | 元数据的速率限制 |

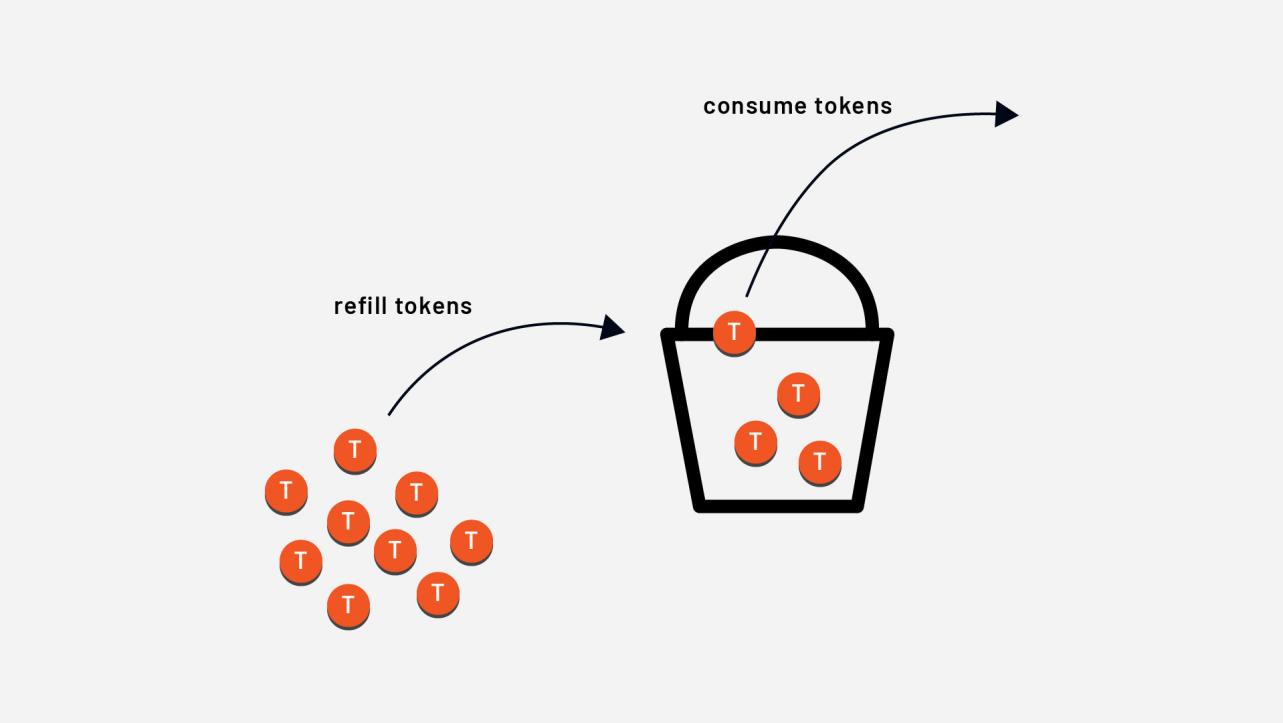
下图总结了这些操作与描述符和速率限制服务上的实际速率限制配置的关系。



# 2.13 本地速率限制

本地速率限制过滤器对过滤器链处理的传入连接应用一个**令牌桶**速率限制。

代币桶算法的基础是代币在桶中的比喻。桶里的代币以一个固定的速度被重新填满。每次收到一个请求或连接时，我们都会检查桶里是否还有任何代币。如果有，就从桶中取出一个代币，然后处理该请求。如果没有剩余的代币，该请求就会被放弃（即速率限制）。



本地速率限制可以在听众层面或虚拟主机或路由层面进行全局配置，就像全局速率限制一样。我们还可以在同一配置中结合全局和本地速率限制。

token\_bucket指定了过滤器处理的请求所使用的配置。它包括桶可以容纳的最大数量（max\_tokens），在每个填充间隔（tokens\_per\_refill）中增加的代币数量，以及填充内部（fill\_interval）。

下面是一个最多可以容纳5000个代币的桶的配置实例。每隔30秒，100个tokens被添加到桶中。这个桶的容量永远不会超过5000个tokens。

token\_bucket:  
 max\_tokens: 5000  
 tokens\_per\_fill: 100  
 fill\_interval:30s

为了控制令牌桶是在所有工作者之间共享（即每个Envoy进程）还是按连接使用，我们可以设置local\_rate\_limit\_per\_downstream\_connection字段。默认值是false，这意味着速率限制被应用于每个Envoy进程。

控制是否启用或强制执行某一部分请求的速率限制的两个设置被称为 filter\_enabled 和 filter\_enforced。这两个值在默认情况下都设置为0%。

速率限制可以被启用，但不一定对一部分请求强制执行。例如，我们可以对50%的请求启用速率限制。然后，在这50%的请求中，我们可以强制执行速率限制。



以下配置对所有传入的请求启用并执行速率限制。

token\_bucket:  
 max\_tokens: 5000  
 tokens\_per\_fill: 100  
 fill\_interval:30s  
filter\_enabled:  
 default\_value:  
 numerator: 100  
 分母。HUNDRED  
filter\_enforced:  
 default\_value:  
 分子。 100  
 分母。百分之百

我们还可以为限制速率的请求添加请求和响应头信息。我们可以在request\_headers\_to\_add\_when\_not\_enforced字段中提供一个头信息列表，Envoy将为每个转发到上游的限速请求添加一个请求头信息。请注意，这只会在过滤器启用但未强制执行时发生。

对于响应头信息，我们可以使用 response\_headers\_to\_add 字段。我们可以提供一个头文件的列表，这些头文件将被添加到已被限制速率的请求的响应中。这只有在过滤器被启用或完全强制执行时才会发生。

如果我们在前面的例子的基础上，这里有一个例子，说明如何在所有速率有限的请求中添加特定的响应头。

token\_bucket:  
 max\_tokens: 5000  
 tokens\_per\_fill: 100  
 fill\_interval:30s  
filter\_enabled:  
 default\_value:  
 numerator: 100  
 分母。HUNDRED  
filter\_enforced:  
 default\_value:  
 分子。 100  
 分母。百分之百  
response\_headers\_to\_add:  
 - append: false  
 header:  
 key: x-local-rate-limit  
 value: 'true

我们可以配置本地速率限制器，使所有虚拟主机和路由共享相同的令牌桶。为了在全局范围内启用本地速率限制过滤器（不要与全局速率限制过滤器混淆），我们可以在http\_filters列表中为其提供配置。

比如说。

...  
http\_filters:  
- name: envoy.filters.http.local\_ratelimit  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.http.local\_ratelimit.v3.LocalRateLimit  
 stat\_prefix: http\_local\_rate\_limiter  
 token\_bucket:  
 max\_tokens: 10000  
 ...  
- name: envoy.filters.http.router  
...

如果我们想启用每个路由的本地速率限制，我们仍然需要将过滤器添加到http\_filters列表中，而不需要任何配置。然后，在路由配置中，我们可以使用typed\_per\_filter\_config并指定本地速率限制的过滤器配置。

比如说。

...  
route\_config:  
 name: my\_route  
 virtual\_hosts。  
 - name: my\_service  
 域名。["\*"]  
 路线。  
 - 匹配。  
 前缀。/  
 路线。  
 cluster: some\_cluster  
 typed\_per\_filter\_config:  
 envoy.filters.http.local\_ratelimit:  
 "@type": type.googleapis.com/envoy.extensions. filters.http.local\_ratelimit.v3.LocalRateLimit  
 token\_bucket:  
 max\_tokens: 10000  
 tokens\_per\_fill: 1000  
 fill\_interval: 1s  
 filter\_enabled:  
 default\_value:  
 分子。100  
 分母。HUNDRED  
 filter\_enforced:  
 default\_value:  
 分子。100  
 分母。HUNDRED  
http\_filters:  
- name: envoy.filters.http.local\_ratelimit  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.http.local\_ratelimit.v3.LocalRateLimit  
 stat\_prefix: http\_local\_rate\_limiter  
- name: envoy.filters.http.router

上述配置在http\_filter列表中加载了本地速率限制过滤器。我们在路由配置中使用typed\_per\_filter\_config来配置它，并以envoy.filters.http.local\_ratelimit的名字来引用这个过滤器。

## 使用描述符进行局部速率限制

就像我们在做全局速率限制时使用描述符一样，我们也可以把它们用于本地每条路由的速率限制。我们需要配置两个部分：路由上的操作和本地速率限制过滤器配置中的描述符列表。

我们可以用为全局速率限制定义动作的方式为局部速率限制定义动作。

...  
route\_config:  
 name: my\_route  
 virtual\_hosts。  
 - name: my\_service  
 域名。["\*"]  
 路线。  
 - 匹配。  
 前缀。/  
 路线。  
 cluster: some\_cluster  
 rate\_limits:  
 - 行动。  
 - header\_value\_match:  
 descriptor\_value: post\_request  
 headers:  
 - name: ":method"  
 exact\_match:POST  
 - header\_value\_match:  
 描述符\_值: get\_request  
 头信息。  
 - name: ":method"  
 精确匹配。GET  
...

第二部分是编写配置以匹配生成的描述符，并提供令牌桶信息。这在描述符字段下的速率限制过滤器配置中得到完成。

比如说。

typed\_per\_filter\_config:  
 envoy.filters.http.local\_ratelimit:  
 "@type": type.googleapis.com/envoy.extensions. filters.http.local\_ratelimit.v3.LocalRateLimit  
 stat\_prefix: some\_stat\_prefix  
 token\_bucket:  
 max\_tokens: 1000  
 tokens\_per\_fill: 1000  
 fill\_interval: 60s  
 filter\_enabled:  
 ...  
 filter\_enforced:  
 ...  
 描述符。  
 - 条目。  
 - key: header\_match  
 值：post\_request  
 token\_bucket:  
 max\_tokens: 20  
 tokens\_per\_fill: 5  
 fill\_interval:30s  
 - 条目。  
 - key: header\_match  
 value: get\_request  
 token\_bucket:  
 max\_tokens: 50  
 tokens\_per\_fill: 5  
 fill\_interval: 20s  
...

对于所有的POST请求（即（"header\_match": "post\_request"）），桶被设置为20个令牌，它每30秒重新填充5个令牌。对于所有的GET请求，该桶最多可以容纳50个令牌，每20秒重新填充5个令牌。

# 2.14 速率限制的统计

无论是使用全局还是局部的速率限制，Envoy都会发出下表中描述的指标。我们可以在配置过滤器时使用stat\_prefix字段来设置统计信息的前缀。

当使用本地速率限制器时，每个度量名称的前缀是<stat\_prefix>.http\_local\_rate\_limit.<metric\_name>，当使用全局速率限制器时，前缀是cluster.<route\_target\_cluster>.ratelimit.<metric\_name>。

|  |  |  |
| --- | --- | --- |
| 速率限制器 | 公制名称 | 描述 |
| 当地 | 启用 | 速率限制器被调用的请求总数 |
| 本地/全球 | 好的 | 来自代币桶的低于限制的响应总数 |
| 当地 | 速率有限 | 没有可用令牌的答复总数（但不一定强制执行） |
| 当地 | 强制执行 | 有速率限制的请求总数（例如，返回HTTP 429）。 |
| 全球 | 超限 | 费率限制服务的超限答复总数 |
| 全球 | 错误 | 与费率限制服务联系的错误总数 |
| 全球 | 允许的失败模式 | 属于错误但由于 failure\_mode\_deny 设置而被允许的请求总数 |

# 3.0群组

在这个模块中，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、离群检测等。

在本模块结束时，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、异常值检测和断路。

# 3.1 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。每个集群是一个端点的集合，Envoy需要解析这些端点来发送流量。

解决端点的过程被称为**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 (地址)**

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 -- 让我们更详细地看看它们。

### 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决的网络名称。比如说。

集群。  
 - name: my\_cluster\_name  
 类型。STATIC  
 load\_assignment:  
 cluster\_name: my\_service\_name  
 端点。  
 - 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，它默认为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。

集群。  
 - name: my\_cluster\_name  
 类型。STRICT\_DNS  
 load\_assignment:  
 cluster\_name: my\_service\_name  
 端点。  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 地址: 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。通常使用轮回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，连接将保持活力，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。

集群。  
 - name: my\_cluster\_name  
 类型。LOGICAL\_DNS  
 load\_assignment:  
 cluster\_name: my\_service\_name  
 端点。  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 地址: 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。

集群。  
 - name: my\_cluster\_name  
 类型。EDS  
 eds\_cluster\_config:  
 eds\_config:  
 ...

我们在[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。

集群。  
 - name: original\_dst\_cluster  
 类型。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离群检测实现的。

# 3.2 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**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 "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。

集群。  
 - name: my\_cluster\_name  
 health\_checks:  
 - timeout:1s  
 间隔：0.25s  
 unhealthy\_threshold: 5  
 healthy\_threshold: 2  
 http\_health\_check:  
 路径。"/health"  
 expected\_statuses:  
 - start: 200  
 结束。 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（间隔）发送一次请求，在超时前等待1s（超时）。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）。

1. 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。

集群。  
 - name: my\_cluster\_name  
 health\_checks:  
 - timeout:1s  
 间隔：0.25s  
 unhealthy\_threshold: 1  
 healthy\_threshold: 1  
 tcp\_health\_check:  
 send:  
 text:"68656C6C6F"  
 接收。  
 - text:"68656C6C6F"  
 ...

1. gRPC健康检查

本健康检查遵循 [grpc.health.v1.Health](https://github.com/grpc/grpc/blob/master/src/proto/grpc/health/v1/health.proto)健康检查协议。查看 [GRPC health checking protocol document](https://github.com/grpc/grpc/blob/master/doc/health-checking.md)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服务名称和权限。服务名称是设置在grpc.health.v1.Health的HealthCheckRequest的服务字段中的值。授权是:authority头的值。如果它是空的，Envoy会使用集群的名称。

集群。  
 - name: my\_cluster\_name  
 health\_checks:  
 - timeout:1s  
 间隔：0.25s  
 unhealthy\_threshold: 1  
 healthy\_threshold: 1  
 grpc\_health\_check:{}  
 ...

1. 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键，Envoy会执行EXIST <键>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。

集群。  
 - name: my\_cluster\_name  
 health\_checks:  
 - timeout:1s  
 间隔：0.25s  
 unhealthy\_threshold: 1  
 healthy\_threshold: 1  
 redis\_health\_check:  
 key: "维护"  
 ...

上面的例子检查键 "维护"（如EXIST维护），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. 非穿透式模式

当以非直通模式运行时，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。

非直通模式的一个变种是，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:  
 值: 15  
...

1. 透过模式

在直通模式下，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。

通行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  
...

上面的片段启用了直通模式，缓存在5分钟内到期。

# 3.3 离群点检测

第二种类型的健康检查被称为**被动健康检查**。被称为**离群点检测的**过程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。

为了使离群值检测发挥作用，我们需要过滤器来报告错误、超时和重置。目前，有四个过滤器支持离群检测。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

这些错误是针对交易的，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。

1. 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，并配置离群点检测，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。默认模式是错误不被分割（即 split\_external\_local\_origin\_errors为false）。

## 端点射出

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会立即弹出离群（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，他们会被弹出的时间越来越长。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在间隔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。剔除时间过后，端点会自动返回到健康的负载均衡池中。

现在我们了解了离群点检测和端点弹出的基本知识，让我们看看不同的离群点检测方法。

## 检测类型

Envoy支持以下五种异常值检测类型。

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。

集群。  
 - name: my\_cluster\_name  
 outlier\_detection:  
 间隔。5s  
 base\_ejection\_time: 15s  
 最大弹射时间: 50s  
 最大弹射率。 30  
 连续\_5xx。 10  
 ...

上述离群检测将弹出一个失败的端点，一旦它失败10次。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。

1. 连续的网关故障

连续网关故障类型与连续5xx类型类似。它考虑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裂模式下的网关错误，并且只由HTTP过滤器支持。连续错误的数量可通过contriable\_gateway\_failure字段进行配置。

集群。  
 - name: my\_cluster\_name  
 outlier\_detection:  
 间隔。5s  
 base\_ejection\_time: 15s  
 最大弹射时间: 50s  
 最大弹射率。 30  
 连续\_网关\_失败。 10  
 ...

1. 连续的局部起源失败

这种类型只在分割模式下启用（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。如果没有提供，它默认为5。

集群。  
 - name: my\_cluster\_name  
 outlier\_detection:  
 间隔。5s  
 base\_ejection\_time: 15s  
 最大弹射时间: 50s  
 最大弹射率。 30  
 连续\_本地\_源头\_失败。 10  
 ...

1. 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，而在分割模式下，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将不会进行离群检测。

成功率\_stdev\_因子用于确定弹出阈值。弹射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。

1. 失败率

失败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。

# 3.4 断路器

断路是一种重要的模式，可以帮助服务的恢复性。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，并尽快向下游施加背压。

让我们看一个定义断路的片段。

...  
 集群。  
 - name: my\_cluster\_name  
 ...  
 circuit\_breakers:  
 thresholds:  
 - 优先级。DEFAULT  
 max\_connections:1000  
 - 优先级：高  
 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，断路器就会跳闸，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，断路器会跳闸，并增加集群的上游\_cx\_overflow指标。默认值是1024。

1. 最大的未决请求（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。

1. 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。

1. 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，上游\_rq\_retry\_overflow计数器就会递增。

作为选择，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。

# 3.5 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被称为**一致散列**。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，散列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，我们将散列值与可用端点的数量相乘（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。

一致性散列是一种方法，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。

## 负载均衡政策

Envoy使用其中一个负载均衡策略来选择一个端点来发送流量。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。

### 加权轮回制（默认）

加权轮回（ROUND\_ROBIN）以轮回顺序选择端点。如果端点是加权的，那么就会使用加权的轮回时间表。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，即使用加权的循环时间表，其中的权重是根据选择时端点的请求负荷动态调整。

以下公式用于动态计算权重。

weight = load\_balancing\_weight / (active\_requests + 1)^active\_request\_bias

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回一样，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:最短请求时间（LEAST\_REQUEST  
 least\_request\_lb\_config:  
 choice\_count:5  
 active\_request\_bias: 0.5  
...

### 环状散列

环形散列（或模数散列）算法（RING\_HASH）实现了对端点的一致散列。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。散列键默认为端点地址；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。环的大小越大，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。戒指\_HASH  
 ring\_hash\_lb\_config:  
 最小环数: 2000  
 最大环数: 10000  
...

### 磁悬浮

与环形散列算法一样，磁悬浮（MAGLEV）算法也实现了对端点的一致散列。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。磁悬浮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。你可以在文章中阅读更多关于它的内容 [Maglev: A Fast and Reliable Software Network Load Balancer](https://dgryski.medium.com/consistent-hashing-algorithmic-tradeoffs-ef6b8e2fcae8).

我们可以使用maglev\_lb\_config字段来设置磁悬浮算法的可选配置。

...  
 lb\_policy:磁悬浮  
 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。

### 原定目的地

原始目的地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回算法更好。

# 4.0 动态配置概述

在本模块中，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。本模块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。

在本模块结束时，你将了解静态和动态配置之间的区别。

# 4.1 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被使用。

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 |
| 听众发现服务（LDS） | 使用LDS，Envoy可以在运行时发现监听器，包括所有的过滤器堆栈、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）。 |
| 路线发现服务（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。 |
| 范围广泛的路由发现服务（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可以为其听众发现秘密（证书和私钥，TLS会话票密钥），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）。**

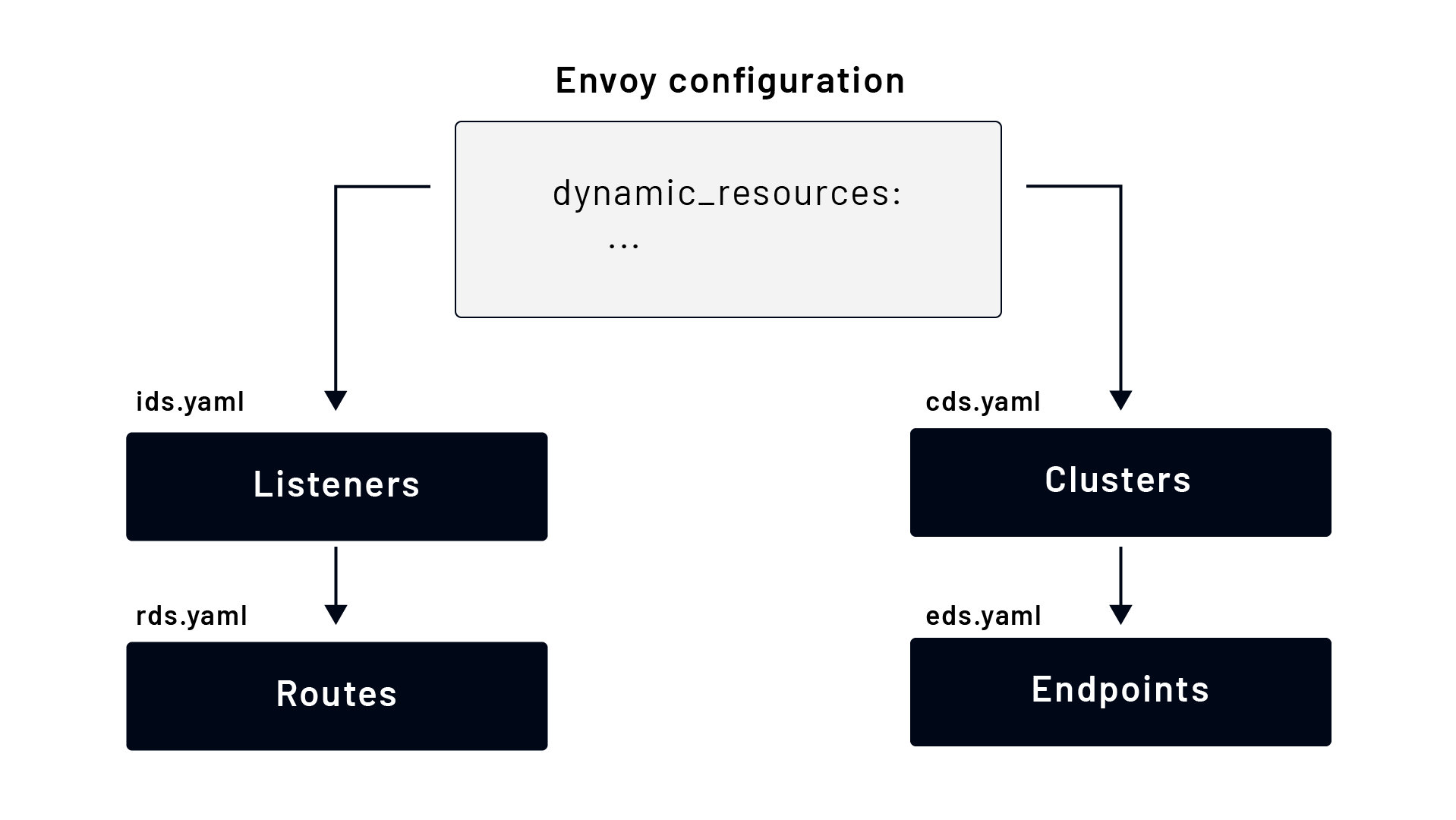
表中的发现服务是独立的，有不同的gRPC/REST服务名称。使用聚合发现服务（ADS），我们可以使用一个单一的gRPC服务，在一个gRPC流中支持所有的资源类型（监听器、路由、集群...）。ADS还能确保不同资源的更新顺序正确。请注意，ADS只支持gRPC。如果没有ADS，我们就需要协调其他gRPC流来实现正确的更新顺序。

**Delta 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。

# 4.2 来自文件系统的动态配置

动态提供配置的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，我们需要在节点字段下提供信息。如果我们可能有多个Envoy代理指向相同的配置文件，那么节点字段是用来识别一个特定的Envoy实例。



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。比如说。

节点。  
 cluster: my-cluster  
 id: some-id  
  
dynamic\_resources:  
 lds\_config:  
 path:/etc/envoy/lds.yaml  
 cds\_config:  
 路径。/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"  
资源。  
- "@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"  
资源。  
- "@type": type.googleapis.com/envoy.config.cluster.v3.Cluster  
 name: instance\_1  
 类型。EDS  
 eds\_cluster\_config:  
 eds\_config:  
 path:/etc/envoy/eds.yaml

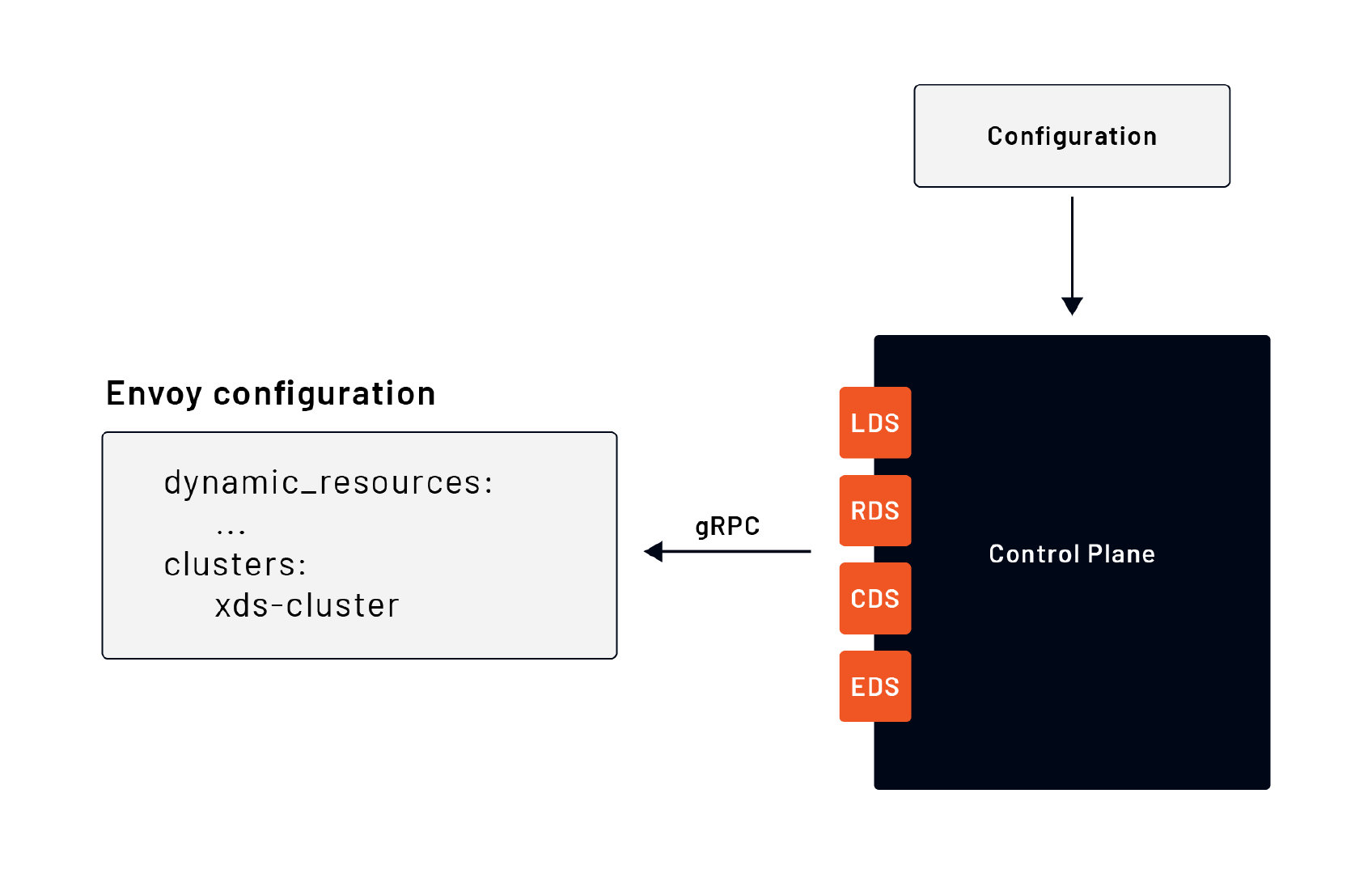
另外，注意我们已经把集群的类型设置为EDS。EDS的配置会是这样的。

version\_info:"0"  
资源。  
- "@type": type.googleapis.com/envoy.config.endpoint.v3.ClusterLoadAssignment  
 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。

# 4.3 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。我们必须创建我们的控制平面，实现发现服务接口。有一个xDS服务器实现的简单例子 [here](https://github.com/envoyproxy/go-control-plane/tree/main/internal/example).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:  
 - 名称: xds\_cluster  
 类型。STATIC  
 load\_assignment。  
 cluster\_name: xds\_cluster  
 端点。  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 地址。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。重要的部分是，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。除了高层配置外，在 [Istio](https://istio.io)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。高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。

# 5.0 听众子系统

在监听器子系统模块中，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。

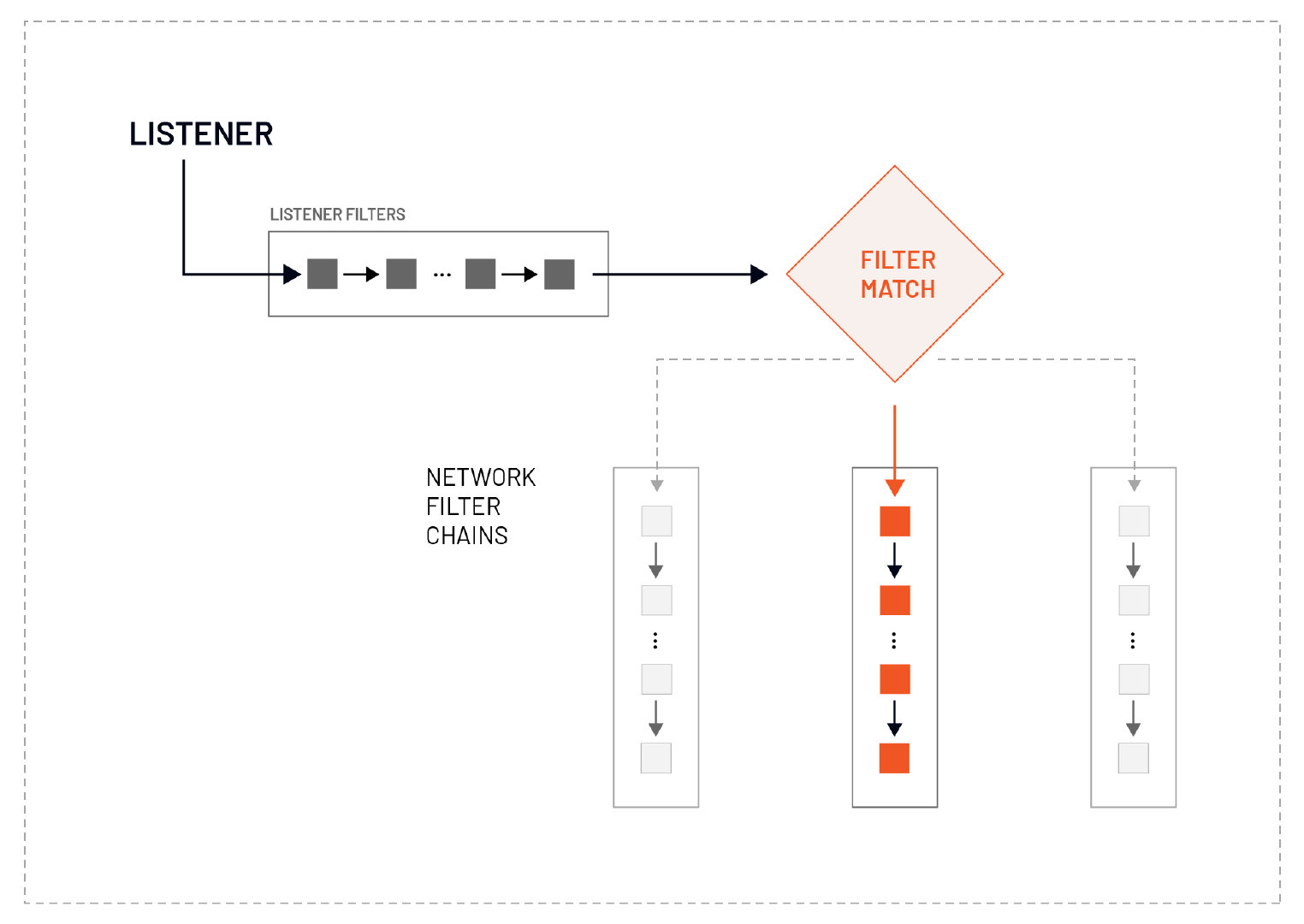
在本模块结束时，你将了解听众子系统的不同部分，并知道过滤器和过滤器链匹配是如何工作的。

# 5.1 听众过滤器

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。听众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。

我们不要把听众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。

请注意，在没有任何听众过滤器的情况下操作Envoy并不是不典型的。



Envoy会在网络级过滤器之前处理监听器过滤器。我们可以在听众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。

听众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。听众过滤器的顺序很重要，因为Envoy在听众接受套接字后，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检查器听众过滤器来确定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。

# 5.2 滤波链匹配

过滤链匹配允许我们指定为听众选择特定过滤链的标准。

我们可以在配置中定义多个过滤链，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链。如果主机名hello.com连接，我们可以选择一个过滤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才能选择该过滤链。

财产匹配顺序如下。

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. 运输协议
5. 应用协议（TLS协议的ALPN）。
6. 直接连接的源IP地址（这只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链匹配。

filter\_chains:  
- filter\_chain\_match:  
 server\_names:  
 - "\*.hello.com"  
 过滤器。  
 ...  
- filter\_chain\_match:  
 source\_prefix\_ranges:  
 - address\_prefix: 192.0.0.1  
 prefix\_len: 32  
 滤波器。  
 ...  
- filter\_chain\_match:  
 transport\_protocol: tls  
 过滤器。  
 ...

让我们假设一个TLS请求从IP地址进来，并且192.0.0.1SNI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IP192.0.0.1进来的，那就不是TLS，而且SNI也不符合\*.hello.com的要求。Envoy将执行第二个过滤链--与特定IP地址相匹配的那个。

# 5.3 HTTP检查器监听器过滤器

的 [HTTP inspector listener filter](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/http_inspector)(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.listener.http\_inspector.v3.HttpInspector  
 filter\_chains:  
 - filter\_chain\_match:  
 application\_protocols:["h2"]  
 过滤器。  
 - name: my\_http2\_filter  
 ...   
 - filter\_chain\_match:  
 application\_protocols:["http/1.1"]  
 过滤器。  
 - 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。

# 5.4 原始目的地监听器过滤器

该 [Original destination](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/original_dst_filter)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（如果透明选项被设置）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.listener.original\_dst.v3.OriginalDst  
...  
集群。  
 - name: original\_dst\_cluster  
 connect\_timeout:5s  
 类型。ORIGNAL\_DST  
 lb\_policy:cluster\_provided

# 5.5 原始源听众过滤器

的 [original source filter](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/original_src_filter)(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 ，Envoy连接到上游，源IP10.0.0.1是 。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头。

- name: envoy.filters.listener.original\_src  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.listener.original\_src.v3.OriginalSrc  
 标记。 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。

# 5.6 代理协议监听器过滤器

该 [Proxy protocol](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/proxy_protocol)听众过滤器（envoy.filters.listener.proxy\_protocol）增加了对 [HAProxy proxy protocol](https://www.haproxy.org/download/1.9/doc/proxy-protocol.txt).

代理人使用他们的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连锁代理。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，例如。

# 5.7 TLS检查器监听器过滤器

听众过滤器让我们可以检测到传输是TLS还是明文。 [TLS inspector](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/tls_inspector)听众过滤器允许我们检测传输是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。使用ALPN，我们可以确定客户是否在说HTTP/1.1或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来执行不同的过滤链。

...  
 listener\_filters:  
 - name: "envoy.filters.listener.tls\_inspector"  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.listener.tls\_inspector.v3.TlsInspector  
 filter\_chains:  
 - filter\_chain\_match:  
 application\_protocols:["h2c"]  
 过滤器。  
 - name: some\_filter  
 ...   
 - filter\_chain\_match:  
 server\_names:"something.hello.com"  
 transport\_socket:  
 ...  
 过滤器。  
 - name: another\_filter  
...

# 6.0 日志

在日志模块中，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中的副车，Envoy都处于独特的地位，可以揭示你的网络中正在发生的事情。了解的一个常见途径是日志记录，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。

在本模块结束时，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。

# 6.1 访问记录

## 什么是访问记录？

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）、你请求网页的日期和时间、请求属性（方法、路径、头文件、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**访问日志的**文本文件中。

通常，来自网络服务器或代理机构的访问日志条目遵循标准化的通用日志格式。不同的代理机构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）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它。

## 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 - "-" "curl/7.64.0" "9c08a41b-805f-42c0-bb17-40ec50a3377a" "localhost:10000" "-"  
[2021-11-01T21:08:18.274Z] "POST /hello HTTP/1.1" 200 - 0 3 0 - "-" "curl/7.64.0" "6a593d31-c9ac-453a-80e9-ab805d07ae20" "localhost:10000" "-"  
[2021-11-01T21:09:42.717Z] "GET /test HTTP/1.1" 404 NR 0 0 0 - "-" "curl/7.64.0" "1acc3559-50eb-463c-ae21-686fe34abbe8" "localhost:10000" "-"

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)%%protocol%"  
%response\_code% %response\_flags% %bytes\_received% %bytes\_sent% %duration% %。  
%resp(x-envoy-upstream-service-time)% "%req(x-forwarded-for)%" "%req(user-agent)%"  
"%req(x-request-id)%" "%req(:authority)%" "%upstream\_host%"

诸如%RESPONSE\_FLAGS%、%REQ(:METHOD)%等值被称为**命令操作符**。

### 指挥员

命令操作者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。

|  |  |  |
| --- | --- | --- |
| 指挥员 | 描述 | 例子 |
| 启用时间（START\_TIME | 请求开始时间，包括毫秒。 | %START\_TIME(%Y/%m/%dT%H:%M:%S%z ^%)% |
| 协议 | 协议（HTTP/1.1、HTTP/2或HTTP/3）。 | ％PROTOCOL％ |
| 响应\_CODE | HTTP响应代码。响应代码被设置为如果0下游客户端断开连接。 | %回复\_代码% |
| 响应\_代码\_细节 | 关于HTTP响应的额外信息（例如，谁设置的以及为什么）。 | 响应\_代码\_细节%。 |
| 连接终止的细节 | 提供关于Envoy因L4原因终止连接的额外信息。 | %connection\_termination\_details%。 |
| ROUTE\_NAME | 航线的名称。 | %ROUTE\_NAME% |
| 连接点\_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。 | %connection\_id% |
| 呼叫中心： 呼叫中心： 呼叫中心： 呼叫中心：GRPC\_STATUS | gRPC状态代码，包括文本信息和一个数字。 | %GRPC\_STATUS% |
| 帐号：HOSTNAME | 系统主机名。 | %HOSTNAME% |
| 本地\_回复\_正文 | 被Envoy拒绝的请求的正文。 | %local\_reply\_body% |
| 过滤链名称 | 下游连接的网络过滤链名称。 | %filter\_chain\_name% |

#### 尺寸

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。

|  |  |  |
| --- | --- | --- |
| 指挥员 | 描述 | 例子 |
| request\_header\_bytes | 请求头的未压缩字节。 | %request\_header\_bytes% |
| 响应\_headers\_bytes | 响应头文件的未压缩字节数。 | %response\_headers\_bytes％。 |
| 响应\_trailers\_bytes | 响应拖车的未压缩字节。 | %响应\_trailers\_bytes%。 |
| 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。 | %BYTES\_SENT% |
| BYTES\_RECEIVED | 收到的正文字节数。 | %bytes\_received% |
| 上游\_wire\_bytes\_sent | 由HTTP流向上游发送的总字节数。 | %upstream\_wire\_bytes\_sent％。 |
| 上游\_wire\_bytes\_received | 从上游HTTP流收到的字节总数。 | %upstream\_wire\_bytes\_received％。 |
| 上游\_头条\_bytes\_sent | 由HTTP流向上游发送的头字节的数量。 | %upstream\_header\_bytes\_sent％。 |
| 上游\_头条\_bytes\_received | HTTP流从上游收到的头字节的数量。 | %upstream\_header\_bytes\_received% |
| 下游\_wire\_bytes\_sent | HTTP流向下游发送的总字节数。 | %downstream\_wire\_bytes\_sent% |
| 下游\_wire\_bytes\_received | HTTP流从下游收到的字节总数。 | %downstream\_wire\_bytes\_received% |
| 下游\_头条\_bytes\_sent | 由HTTP流向下游发送的头字节的数量。 | %downstream\_header\_bytes\_sent%的数据。 |
| 下游\_头条\_bytes\_received | HTTP流从下游收到的头字节的数量。 | %downstream\_header\_bytes\_received% |

#### 寿命

|  |  |  |
| --- | --- | --- |
| 指挥员 | 描述 | 例子 |
| 持续时间 | 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% |
| 请求时间 | 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% |
| request\_tx\_duration | 从开始时间到上游发送的最后一个字节，请求的总持续时间（以毫秒计）。 | %request\_tx\_duration％。 |
| 响应时间 | 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）。 | 响应的时间百分比 |
| 响应\_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）。 | %响应\_tx\_duration%。 |

#### 回应标志

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。
* 统一通信。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

|  |  |  |
| --- | --- | --- |
| 指挥员 | 描述 | 例子 |
| UPSTREAM\_HOST | 上游主机URL或TCP连接的tcp://ip:端口。 | %upstream\_host% |
| 上游\_集群 | 上游主机所属的上游集群。如果运行时特性 envoy.reloadable\_features.use\_observable\_cluster\_name 被启用，那么如果提供了 alt\_stat\_name 就会被使用。 | %upstream\_cluster% |
| 上游\_本地\_地址 | 上游连接的本地地址。如果是一个IP地址，那么它包括地址和端口。 | %upstream\_local\_address％。 |
| 上游\_运输\_故障\_原因 | 如果由于传输套接字导致连接失败，则提供来自传输套接字的失败原因。 | %upstream\_transport\_failure\_reason%（上游传输失败）。 |

#### 下游信息

|  |  |  |
| --- | --- | --- |
| 指挥员 | 描述 | 例子 |
| 下游\_远端\_地址 | 下游连接的远程地址。如果是一个IP地址，那么它包括地址和端口。 | %下游\_远端\_地址％。 |
| 下游\_远程地址\_无端口 | 下游连接的远程地址。如果是一个IP地址，那么它只包括地址。 | %downstream\_remote\_address\_without\_port%（无端口）。 |
| 下游\_直接\_远程\_地址 | 下游连接的直接远程地址。如果是一个IP地址，那么它包括地址和端口。 | %downstream\_direct\_remote\_address%（下游直接地址）。 |
| 下游\_直接远程地址\_没有端口的地址 | 下游连接的直接远程地址。如果是一个IP地址，那么它只包括地址。 | %downstream\_direct\_remote\_address\_without\_port%(顺流而下) |
| 下游\_本地\_地址 | 下游连接的本地地址。如果它是一个IP地址，那么它包括地址和端口。如果原始连接是由iptables REDIRECT重定向的，那么这个值代表由原始目标过滤器恢复的原始目标地址。如果由iptables TPROXY重定向，并且监听器的透明选项被设置为 "true"，那么这个值代表原始目标地址和端口。 | %downstream\_local\_address％。 |
| 下游\_本地地址\_不含端口 | 与DOWNSTREAM\_LOCAL\_ADDRESS相同，如果该地址是一个IP地址，则不包括端口。 | %downstream\_local\_address\_without\_port%（不含端口）。 |
| 下游\_本地\_端口 | 与DOWNSTREAM\_LOCAL\_ADDRESS\_WITHOUT\_PORT类似，但只提取DOWNSTREAM\_LOCAL\_ADDRESS的端口部分。 | %downstream\_local\_port% |

#### 车头和拖车

REQ、RESP和TRAILER命令操作符允许我们提取请求、响应和拖车头信息，并将其纳入日志。

|  |  |  |
| --- | --- | --- |
| 指挥员 | 描述 | 例子 |
| 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，则使用头条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响应跟踪器 | %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

|  |  |  |
| --- | --- | --- |
| 指挥员 | 描述 | 例子 |
| dynamic\_metadata(namespace:key\*):z | 动态元数据信息，其中NAMESPACE是设置元数据时使用的过滤器。KEY是命名空间中的一个可选的查找键，可以选择指定用:分隔的嵌套键。Z是一个可选的参数，表示字符串截断，长度不超过Z个字符。 | 例如，my\_filter。{"my\_key":"hello", "json\_object":{"some\_key":"foo"}}元数据可以用%DYNAMIC\_METADATA(my\_filter)%进行记录。要记录一个特定的键，我们可以写%DYNAMIC\_METADATA(my\_filter:my\_key)%。 |
| cluster\_metadata(namespace:key\*):z | 上游集群元数据信息，其中NAMESPACE是设置元数据时使用的过滤器命名空间，KEY是命名空间中一个可选的查找键，可选择指定由:分隔的嵌套键。Z是一个可选的参数，表示字符串截断，长度不超过Z个字符。 | 见DYNAMIC\_METADATA的例子。 |
| filter\_state(key:f):z | 过滤器状态信息，其中的KEY是需要查询过滤器状态对象的。如果可能的话，序列化的proto将被记录为一个JSON字符串。如果序列化的proto是未知的，那么它将被记录为一个protobuf调试字符串。F是一个可选的参数，表示FilterState使用哪种方法进行序列化。如果设置了PLAIN，那么过滤器状态对象将被序列化为一个非结构化的字符串。如果设置了TYPED或者没有提供F，那么过滤器状态对象将被序列化为一个JSON字符串。Z是一个可选的参数，表示字符串的截断，长度不超过Z个字符。 | %FILTER\_STATE(my\_key:PLAIN):10% |

#### TLS

|  |  |  |
| --- | --- | --- |
| 指挥员 | 描述 | 例子 |
| 请求的服务器名称 | 在SSL连接套接字上为服务器名称指示（SNI）设置的字符串值。 | %要求的服务器名称%。 |
| 下游\_local\_uri\_san | 用于建立下游TLS连接的本地证书的SAN中存在的URI。 | %downstream\_local\_uri\_san%。 |
| 下游\_peer\_uri\_san | 用于建立下游TLS连接的同行证书SAN中存在的URI。 | %downstream\_peer\_uri\_san%。 |
| 下游\_本地\_主题 | 用于建立下游TLS连接的本地证书中存在的主题。 | %下游\_本地\_主题%。 |
| 下游\_同行\_主题 | 用于建立下游TLS连接的同伴证书中的主题。 | %下游\_同行\_主题%。 |
| 下游\_同行\_发行商 | 用于建立下游TLS连接的同行证书中存在的签发者。 | %下游\_同行\_发行商% |
| downstream\_tls\_session\_id | 已建立的下游TLS连接的会话ID。 | %downstream\_tls\_session\_id%。 |
| 下游\_tls\_cipher | 用于建立下游TLS连接的密码集的OpenSSL名称。 | %downstream\_tls\_cipher％。 |
| 下游\_tls\_version | 用于建立下游TLS连接的TLS版本（TLSv1.2或TLSv1.3）。 | %downstream\_tls\_version% |
| 下游\_同行\_指纹\_256 | 用于建立下游TLS连接的客户证书的十六进制编码的SHA256指纹。 | %downstream\_peer\_fingerprint\_256% |
| 下游\_同行\_指纹\_1 | 用于建立下游TLS连接的客户证书的十六进制编码的SHA1指纹。 | %downstream\_peer\_fingerprint\_1% |
| 下游\_peer\_serial | 用于建立下游TLS连接的客户证书的序列号。 | %downstream\_peer\_serial% |
| 下游\_同行\_证书 | 用于建立下游TLS连接的URL-安全编码的PEM格式的客户证书。 | %downstream\_peer\_cert％。 |
| 下游\_peer\_cert\_v\_start | 用于建立下游TLS连接的客户证书的有效期开始日期。 | %downstream\_peer\_cert\_v\_start%。可以像START\_TIME一样定制 |
| 下游\_peer\_cert\_v\_end | 用于建立下游TLS连接的客户证书的有效期结束日期。 | %downstream\_peer\_cert\_v\_end%。可以像START\_TIME一样定制 |

# 6.2 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日志器。我们还可以配置多个具有不同日志格式和日志汇的访问日志。**日志**汇是一个抽象的术语，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。

我们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。

- 过滤器。  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog

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汇有以下几种。

* 文件 (FileAccessLog)
* gRPC（HttpGrpcAccessLogConfig和TcpGrpcAccessLogConfig）。
* 标准错误（StderrAccessLog）。
* 废物 (WasmAccessLog)
* 开放式遥测

文件访问日志允许我们将日志条目写到我们在配置中指定的文件中。比如说。

- 过滤器。  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.file  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.file.3.FileAccessLog  
 路径: ./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访问日志汇将日志发送到HTTP或TCP gRPC日志服务。为了使用gRPC日志汇，我们必须建立一个gRPC服务器，其端点要实现 [MetricsService](https://github.com/envoyproxy/envoy/blob/main/api/envoy/service/metrics/v3/metrics_service.proto)，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。

早些时候，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)%%protocol%"  
%response\_code% %response\_flags% %bytes\_received% %bytes\_sent% %duration% %。  
%resp(x-envoy-upstream-service-time)% "%req(x-forwarded-for)%" "%req(user-agent)%"  
"%req(x-request-id)%" "%req(:authority)%" "%upstream\_host%"

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。

比方说，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。

- 过滤器。  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog  
 log\_format:  
 text\_format\_source:  
 inline\_string:"%start\_time% %response\_code% %req（user-agent）%"

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。

- 过滤器。  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog  
 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":"2021-11-01T21:37:59.979Z"}

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。

- 过滤器。  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog  
 log\_format:  
 text\_format\_source:  
 inline\_string:"[%start\_time%] %req(:method)%req\_without\_query(x-envoy-original-path?:path)%protocol%"  
 formatters:  
 - name: envoy.formatter.req\_without\_query  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.formatter.req\_without\_query.v3.ReqWithoutQuery

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

# 6.3 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 |
| status\_code\_filter | 对状态代码值进行过滤。 |
| 持续时间\_过滤器 | 对总的请求持续时间进行过滤，单位为毫秒。 |
| 不是健康检查过滤器 | 对非健康检查请求的过滤。 |
| 可追踪的过滤器 | 对可追踪的请求进行过滤。 |
| 运行时间\_过滤器 | 对请求进行随机抽样的过滤器。 |
| and\_filter | 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。 |
| or\_filter | 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。 |
| header\_filter | 根据请求头的存在或值来过滤请求。 |
| 响应\_flag\_filter | 过滤那些收到设置了Envoy响应标志的响应的请求。 |
| 状况过滤器（grpc\_status\_filter | 根据响应状态过滤gRPC请求。 |
| 延长线\_过滤器 | 使用一个在运行时静态注册的扩展过滤器。 |
| metadata\_filter | 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、标题和一个*和*过滤器。

...  
access\_log:  
- name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog  
 过滤器。  
 and\_filter:  
 filters:  
 header\_filter:  
 header:  
 name: ":method"  
 string\_match:  
 准确。"GET"  
 status\_code\_filter:  
 比较。  
 op:GE  
 值。  
 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。

# 6.4 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/server.cc:368] 初始化历时0（基础id=0，热重启版本=11.104）。  
[2021-11-03 17:22:43.361][1678][info][main] [source/server/server.cc:370] 静态链接扩展。  
[2021-11-03 17:22:43.361][1678][info][main] [source/server/server.cc:372] envoy. filters.network: envoy.client\_ssl\_auth, envoy.echo, envoy.ext\_authz, envoy. filters.network.client\_ssl\_auth  
...

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] 响应拖车地图。160字节：grpc-message,grpc-status  
[17:43:15.965][main][createRuntime] runtime:{}  
[17:43:15.965][main][initialize] 没有给出管理员地址，所以没有启动管理员HTTP服务器。  
[17:43:15.966][config][initializeTracers] 载入跟踪配置  
[17:43:15.966][config][initialize]加载0个静态秘密（s）。  
[17:43:15.966][config][initialize]加载0个集群(s)。  
[17:43:15.966][config][initialize]加载1个监听器(s)  
[17:43:15.969][config][initializeStatsConfig] 载入统计配置  
[17:43:15.969][runtime][onRtdsReady] RTDS已经完成初始化。  
[17:43:15.969][上游][maybeFinishInitialize] cm init：所有集群已初始化  
[17:43:15.969][main][onRuntimeReady] 没有配置允许活动连接数的限制。通过运行时键overload.global\_downstream\_max\_connections设置一个限制。  
[17:43:15.970][main][operator()] 所有集群都已初始化。初始化初始管理器  
[17:43:15.970][config][startWorkers]所有依赖关系已初始化。  
[17:43:15.971][main][run] 启动主调度循环。

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 [source code](https://github.com/envoyproxy/envoy/blob/82261f5a401418df13626ca3fa52fa65fea10c81//source/common/common/logger.h).

下面是/logging终端的默认输出的样子。

活跃的记录者。  
 admin: info  
 alternate\_protocols\_cache: info  
 aws: info  
 assert: info  
 backtrace: info  
 cache\_filter: info  
 client: info  
 config: info  
 connection：信息  
 conn\_handler：信息  
 解压：信息  
 dns：信息  
 dubbo：信息  
 envoy\_bug：信息  
 ext\_authz: info  
 rocketmq：信息  
 file: 信息  
 filter: info  
 forward\_proxy：信息  
 grpc：信息  
 hc：信息  
 health\_checker：信息  
 http：info  
 http2: info  
 hystrix：信息  
 init: info  
 io：信息  
 jwt：信息  
 kafka：信息  
 key\_value\_store: info  
 lua: info  
 main：信息  
 匹配器：信息  
 杂项：信息  
 mongo：信息  
 quic：信息  
 quic\_stream：信息  
 pool：信息  
 rbac：信息  
 redis：信息  
 router：信息  
 runtime：信息  
 stats: 信息  
 secret：信息  
 tap：信息  
 测试：信息  
 thrift: 信息  
 追踪：信息  
 upstream: info  
 udp：信息  
 wasm: 信息

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。

* 痕迹
* 调试
* 信息
* 警告/警告
* 错误
* 关键
* 关闭

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的格式来写。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：上游：关键，秘密：错误，路由器：跟踪。

例如，要将主日志级别设置为跟踪，配置日志级别设置为错误，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-level main:trace, config:error

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

在其中一个实验室中，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）。

# 7.0 管理界面

在管理界面模块中，我们将学习Envoy所暴露的管理界面，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。

在本模块结束时，你将了解如何启用管理界面以及我们可以通过它执行的不同任务。

# 7.1 启用管理界面

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

管理界面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。



管理界面必须使用管理字段明确启用。比如说。

管理员。  
 地址。  
 socket\_address:  
 地址。 127.0.0.1  
 port\_value: 9901

在启用管理界面时要小心。任何有权限进入管理界面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界面，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本地主机访问管理界面，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界面的不同功能。

# 7.2 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
* 集群
* 端点
* 听众
* 范围内的路线
* 航线
* 秘密

### 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](https://www.envoyproxy.io/docs/envoy/latest/api-v3/admin/v3/config_dump.proto#envoy-v3-api-msg-admin-v3-clustersconfigdump)在资源查询参数中使用。

$ curl localhost:9901/config\_dump? resource=static\_clusters  
{  
"configs":[  
 {  
 "@type":"type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster" 。  
 "集群"。{  
 "@type":"type.googleapis.com/envoy.config.cluster.v3.Cluster"。  
 "name":"instance\_1"。  
 },  
 ...  
 {  
 "@type":"type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster"。  
 "集群"。{  
 "@type":"type.googleapis.com/envoy.config.cluster.v3.Cluster"。  
 "name":"instance\_2",  
...

#### 使用掩码参数

为了进一步缩小输出范围，我们可以在掩码参数中指定该字段。例如，只显示每个集群的connect\_timeout值。

$ curl localhost:9901/config\_dump? resource=static\_clusters&mask=cluster.connect\_timeout  
{  
"configs":[  
 {  
 "@type":"type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster"。  
 "集群"。{  
 "@type":"type.googleapis.com/envoy.config.cluster.v3.Cluster"。  
 "connect\_timeout":"5s"  
 }  
 },  
 {  
 "@type":"type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster"。  
 "集群"。{  
 "@type":"type.googleapis.com/envoy.config.cluster.v3.Cluster"。  
 "connect\_timeout":"5s"  
 }  
 },  
 {  
 "@type":"type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster"。  
 "集群"。{  
 "@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，我们可以写如下。

$ curl localhost:9901/config\_dump? resource=static\_clusters&name\_regex=.\*listener.\*  
  
{  
"configs":[  
 {  
 "@type":"type.googleapis.com/envoy.admin.3.ListenersConfigDump.StaticListener"。  
 "listener":{  
 "@type":"type.googleapis.com/envoy.config.listener.v3.Listener",  
 "name":"listener\_0",  
 "地址":{  
 "socket\_address":{  
 "地址":"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，它遵循了 [admin.v3.Certificates proto](https://www.envoyproxy.io/docs/envoy/latest/api-v3/admin/v3/certs.proto#envoy-v3-api-msg-admin-v3-certificates).

# 7.3 统计数据

管理界面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界面访问同一路径来访问该端点。

该端点支持使用过滤器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）。

比如说。

...  
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。

## 记忆

/memory端点将输出当前内存分配和堆的使用情况，单位为字节。它是/stats端点打印出来的信息的一个子集。

$ curl localhost:9901/memory  
{  
"已分配"。"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](https://www.envoyproxy.io/docs/envoy/latest/api-v3/admin/v3/server_info.proto#envoy-v3-api-msg-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 dashboard](https://github.com/Netflix-Skunkworks/hystrix-dashboard/wiki).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。

比如说。

stats\_sinks:   
 - name: envoy.stat\_sinks.hystrix  
 typed\_config:  
 "@type": type.googleapis.com/envoy.config.metrics.v3.HystrixSink  
 num\_buckets: 10

## 争论

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 [documentation](https://github.com/envoyproxy/envoy/blob/main/bazel/PPROF.md)关于如何做到这一点。

# 7.4 记录

/logging端点启用或禁用特定组件或所有记录器的不同记录级别。

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  
活动的记录器。  
 admin: info  
 alternate\_protocols\_cache: info  
 aws: info  
 assert: info  
 backtrace: info  
 cache\_filter: info  
 client: info  
 config: info  
...

输出将包含记录仪的名称和每个记录仪的记录级别。要改变所有活动日志记录器的记录级别，我们可以使用级别参数。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调试。

$ curl -X POST localhost:9901/logging?level=debug  
活动的记录器。  
 admin: debug  
 alternate\_protocols\_cache: debug  
 aws：debug  
 assert: debug  
 回溯：调试  
 cache\_filter: debug  
 客户端: 调试  
 配置：调试  
...

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级别查询参数名称。例如，要将管理员日志记录器级别改为警告，我们可以运行以下程序。

$ curl -X POST localhost:9901/logging?admin=warning  
活动的记录器。  
 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。

# 7.5 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
* 每个主机的重量和位置信息

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，输出是经过修剪的）。

{  
"cluster\_statuses"。[  
 {  
 "name":"api\_google\_com",  
 "host\_statuses":[  
 {  
 "地址"。{  
 "socket\_address":{  
 "地址":"10.0.0.1",  
 "port\_value": 8080  
 }  
 },  
 "统计资料":[  
 {  
 "值":"23",  
 "name":"cx\_total"  
 },  
 {  
 "name":"rq\_error"  
 },  
 {  
 "值":"51",  
 "名称":"rq\_success"  
 },  
 ...  
 ],  
 "health\_status":{  
 "eds\_health\_status":"健康"  
 },  
 "体重":1,  
 "地区":{}  
 }  
 ],  
 "circuit\_breakers":{  
 "阈值":[  
 {  
 "max\_connections":1024,  
 "max\_pending\_requests":1024,  
 "max\_requests":1024,  
 "max\_retries": 3  
 },  
 {  
 "优先级":"高"。  
 "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。

|  |  |
| --- | --- |
| 公制名称 | 描述 |
| cx\_total | 连接总数 |
| cx\_active | 有效连接总数 |
| cx\_connect\_fail | 连接失败总数 |
| rq\_total | 请求总数 |
| rq\_timeout | 过时的请求总数 |
| 胜利 | 有非5xx响应的请求总数 |
| rq\_error | 有5xx响应的请求总数 |
| rq\_active | 有效请求总数 |

## 宿主健康状况

主机的健康状况在health\_status字段下被报告。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。

|  |  |
| --- | --- |
| 领域名称 | 描述 |
| 失败的主动健康检查 | 真，如果该主机目前未能通过主动健康检查。 |
| 失败的检查 | 是的，如果该宿主目前被认为是一个异类，并已被弹出。 |
| failed\_active\_degraded\_check - 如果主机目前通过主动健康检查被标记为降级，则为真。 |  |
| pending\_dynamic\_removal - 如果主机已经从服务发现中移除，但由于主动健康检查而正在稳定，则为真。 |  |
| 待定\_活动\_hc | 真，如果该主机尚未被健康检查。 |
| 被排除在外的\_通过即时的\_hc\_fail | 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。 |
| active\_hc\_timeout | 真，如果主机由于超时而导致活动HC失败。 |
| 健康状况 | 默认情况下，设置为健康（如果不使用EDS）。否则，它也可以被设置为不健康或退化。 |

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":"健康"  
}

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":"健康"  
}

# 7.6 倾听者和倾听者的消耗

/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。

比如说。

$ 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":{  
 "地址":"0.0.0.0",  
 "port\_value": 8080  
 }  
 }  
 },  
 {  
 "name":"http\_hello\_world\_9090",  
 "local\_address":{  
 "socket\_address":{  
 "地址":"0.0.0.0",  
 "port\_value": 9090  
 }  
 }  
 }  
]  
}

## 听众排水

发生排水的一个典型场景是在热重启排水期间。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开放连接的数量。

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，并加入一个可选的优雅查询参数。

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，耗尽时间默认为10分钟（600秒）。该值指定了Envoy将耗尽连接的时间--即在关闭它们之前等待。

耗尽策略参数决定了耗尽序列中的行为（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，随着排泄时间的推移，鼓励排泄的请求的百分比增加到100%。即时策略将使所有的请求在排水序列开始后立即排水。

排水是按听众进行的。然而，它必须在网络过滤器层面得到支持。目前支持优雅排水的过滤器是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）耗尽所有入站监听器的能力。这使用听众上的traffic\_direction字段来确定流量方向。

# 7.7 分接式过滤器

分接过滤器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过滤器。(1) 使用Envoy配置里面的static\_config字段，或者(2) 使用admin\_config字段并指定配置ID。不同的是，我们在静态配置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:  
 匹配。  
 any\_match: true  
...

我们也有一个选项，可以在请求和响应头、拖车和正文上进行匹配。

## 标题/拖车匹配

头部/拖车匹配器使用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:  
 匹配。  
 http\_request\_headers\_match。  
 headers:  
 name: "my-header"  
 string\_match:  
 exact:"你好"  
...

请注意，在string\_match中，我们可以使用其他匹配器（例如前缀、后缀、safe\_regex），正如前面解释的那样。

## 身体匹配

通用请求和响应体匹配使用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:  
 匹配。  
 http\_response\_generic\_body\_match。  
 模式。  
 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标题、预告片和正文匹配器。

or\_match和and\_match使用MatchSet原语，描述逻辑OR或逻辑AND。我们在匹配集内的规则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:  
 匹配。  
 and\_match:  
 规则。  
 - http\_response\_generic\_body\_match。  
 模式。  
 - string\_match:"hello"  
 - http\_request\_headers\_match:  
 headers:  
 name: "my-header"  
 string\_match:  
 准确。"你好"  
...

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:  
 匹配。  
 not\_match:  
 and\_match:  
 规则。  
 - http\_response\_generic\_body\_match。  
 模式。  
 - string\_match:"hello"  
 - http\_request\_headers\_match:  
 headers:  
 name: "my-header"  
 string\_match:  
 准确。"你好"  
...

not\_match 字段和父匹配字段一样使用 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 "真 "时，将总是匹配。

## 输出配置

一旦请求被挖掘出来，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汇。

下面是一个输出配置样本的样子。

...  
output\_config:  
 汇。  
 - 格式。json\_body\_as\_string  
 file\_per\_tap:  
 path\_prefix: tap  
...

使用file\_per\_tap，我们指定要为每个被窃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管理员端点进行输出，还必须使用admin\_config字段配置分接过滤器。如果我们静态地配置了分接过滤器，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，任何响应的主体数据将在as\_bytes字段中，其中包含base64编码的字符串。

例如，下面是攻丝输出的情况。

{  
"http\_buffered\_trace"。{  
 "request":{  
 "headers":[  
 {  
 "key":":authority",  
 "值":"localhost:10000"  
 },  
 {  
 "key":":path",  
 "值":"/"  
 },  
 {  
 "key":":方法",  
 "值":"GET"  
 },  
 {  
 "key":":方案",  
 "值":"http"  
 },  
 {  
 "key":"user-agent",  
 "值":"curl/7.64.0"  
 },  
 {  
 "key":"接受"。  
 "值":"\*/\*"  
 },  
 {  
 "key":"my-header",  
 "值":"你好"  
 },  
 {  
 "key":"x-forwarded-proto",  
 "值":"http"  
 },  
 {  
 "key":"x-request-id",  
 "值":"67e3e8ac-429a-42fb-945b-ec25927fdcc1"  
 }  
 ],  
 "跟踪器"。[]  
 },  
 "响应"。{  
 "头信息"。[  
 {  
 "key":":status",  
 "值":"200"  
 },  
 {  
 "key":"content-length",  
 "值":"5"  
 },  
 {  
 "key":"内容类型"。  
 "值":"text/plain"  
 },  
 {  
 "key":"date",  
 "值":"Mon, 29 Nov 2021 19:31:43 GMT"  
 },  
 {  
 "key":"server",  
 "值":"envoy"  
 }  
 ],  
 "body":{  
 "truncated": false,  
 "as\_bytes":"aGVsbG8="  
 },  
 "拖车":[]  
 }  
}  
}

注意正文中的as\_bytes字段。该值是主体数据的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，主体数据是以字符串的形式写在as\_string字段中。当我们知道主体是人类可读的，并且不需要对数据进行base64编码时，这种格式很有用。

...  
 "身体"。{  
 "truncated": false,  
 "as\_string":"你好"  
 },  
...

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，这意味着如果水槽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 {  
 请求 {  
 headers {  
 key: ":authority"  
 值。"localhost:10000"  
 }  
 headers {  
 key: ":path"  
 值。"/"  
 }  
 headers {  
 key: ":method"  
 值。"GET"  
 }  
 headers {  
 key: ":scheme"  
 值。"http"  
 }  
 headers {  
 key: "user-agent"  
 值。"curl/7.64.0"  
 }  
 headers {  
 key: "接受"  
 值。"\*/\*"  
 }  
 headers {  
 key: "debug"  
 值。"true"  
 }  
 headers {  
 key: "x-forwarded-proto"  
 值。"http"  
 }  
 headers {  
 key: "x-request-id"  
 值。"af6e0879-e057-4efc-83e4-846ff4d46efe"  
 }  
 }  
 响应 {  
 headers {  
 key: ":status"  
 值。"500"  
 }  
 headers {  
 key: "content-length"  
 值。"5"  
 }  
 headers {  
 key: "content-type"  
 值。"text/plain"  
 }  
 headers {  
 key: "date"  
 值。"Mon, 29 Nov 2021 22:32:40 GMT"  
 }  
 headers {  
 key: "server"  
 值。"envoy"  
 }  
 body {  
 as\_bytes:"你好"  
 }  
 }  
}

## 静态配置分接滤波器

我们把匹配的配置和输出配置（使用file\_per\_tap字段）结合起来，静态地配置分接过滤器。

下面是一个通过静态配置来配置分接过滤器的片段。

- name: envoy.filters.http.tap  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.http.tap.v3.Tap  
 common\_config:  
 static\_config:  
 match\_config:  
 any\_match: true  
 output\_config:  
 sinks:  
 - 格式。json\_body\_as\_string  
 file\_per\_tap:  
 path\_prefx: 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端点配置分接过滤器

为了使用/tap端点，我们必须在分接过滤器配置中指定admin\_config和config\_id。

- name: envoy.filters.http.tap  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.http.tap.v3.Tap  
 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请求以配置分接过滤器。例如，下面是配置my\_tap\_config\_id名称所引用的分接过滤器的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。

# 7.8 健康检查

/healthcheck/fail可用于失败的入站健康检查。端点/healthcheck/ok用于恢复失败端点的效果。

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耗尽服务器。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。

# 8.0 扩展Envoy

在本模块中，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。

在本模块结束时，你将了解扩展Envoy的不同方法以及如何使用Lua和Wasm过滤器。

# 8.1 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。

过滤器的一个例子是HTTP过滤器，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. 本地C++ API

第一个选择是编写本地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。

1. 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。

1. 瓦斯姆过滤器

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。

# 8.2 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](https://luajit.org/)(Lua的即时编译器）作为运行时间。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，一旦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Envoys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、正文和预告片。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。

名称： envoy.filters.http.lua  
typed\_config:  
 "@type": type.googleapis.com/envoy.extensions. filters.http.lua.v3.Lua  
 inline\_code:|  
 -- 在请求路径上调用。  
 函数 envoy\_on\_request(request\_handle)  
 -- 做一些事情。  
 结束  
 -- 在响应路径上被调用。  
 function envoy\_on\_response(response\_handle)  
 -- 做一些事情。  
 结束  
 source\_codes:  
 myscript.lua:  
 文件名。/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。

函数 envoy\_on\_request(request\_handle)  
结束

和

函数 envoy\_on\_response(response\_handle)  
结束

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.lua.v3.LuaPerRoute  
 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.lua.v3.LuaPerRoute  
 source\_code:  
 inline\_string:|  
 函数 envoy\_on\_response(response\_handle)  
 -- 在响应时做一些事情。  
 结束

## 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头文件、正文、元数据、各种日志方法（如logTrace、logInfo、logDebug...）、httpCall、连接等等。你可以在 "流处理 "中找到完整的方法列表。 [Lua filter source code](https://github.com/envoyproxy/envoy/blob/d79a3ab49f1aa522d0a465385425e3e00c8db147/source/extensions/filters/http/lua/lua_filter.h#L151).

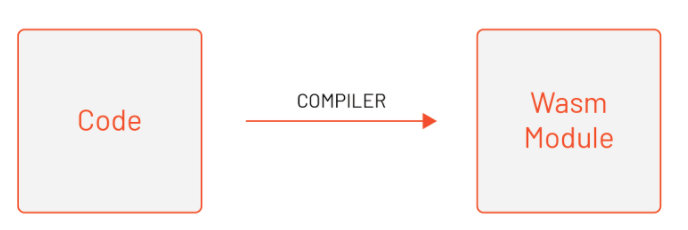
除了流对象外，API还支持以下对象。

* [Header object](https://github.com/envoyproxy/envoy/blob/55fc06b43082064cf7551d8dbc08a0e30e2c2f40/source/extensions/filters/http/lua/wrappers.h#L46)(由headers()方法返回)
* 缓冲区对象（由body()方法返回）。
* [Dynamic metadata object](https://github.com/envoyproxy/envoy/blob/55fc06b43082064cf7551d8dbc08a0e30e2c2f40/source/extensions/filters/http/lua/wrappers.h#L151)(由metadata()方法返回)
* [Stream info object](https://github.com/envoyproxy/envoy/blob/55fc06b43082064cf7551d8dbc08a0e30e2c2f40/source/extensions/filters/http/lua/wrappers.h#L199)(由streamInfo()方法返回)
* 连接对象（通过connection()方法返回）。
* [SSL connection info object](https://github.com/envoyproxy/envoy/blob/0fae6970ddaf93f024908ba304bbd2b34e997a51/source/extensions/filters/common/lua/wrappers.h#L124)(由连接对象的ssl()方法返回)

我们将看到如何使用Lua实验室中的一些对象和方法。

## 8.3 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。它允许开发人员用他们喜欢的编程语言编写，然后将代码编译成**Wasm模块**。



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。

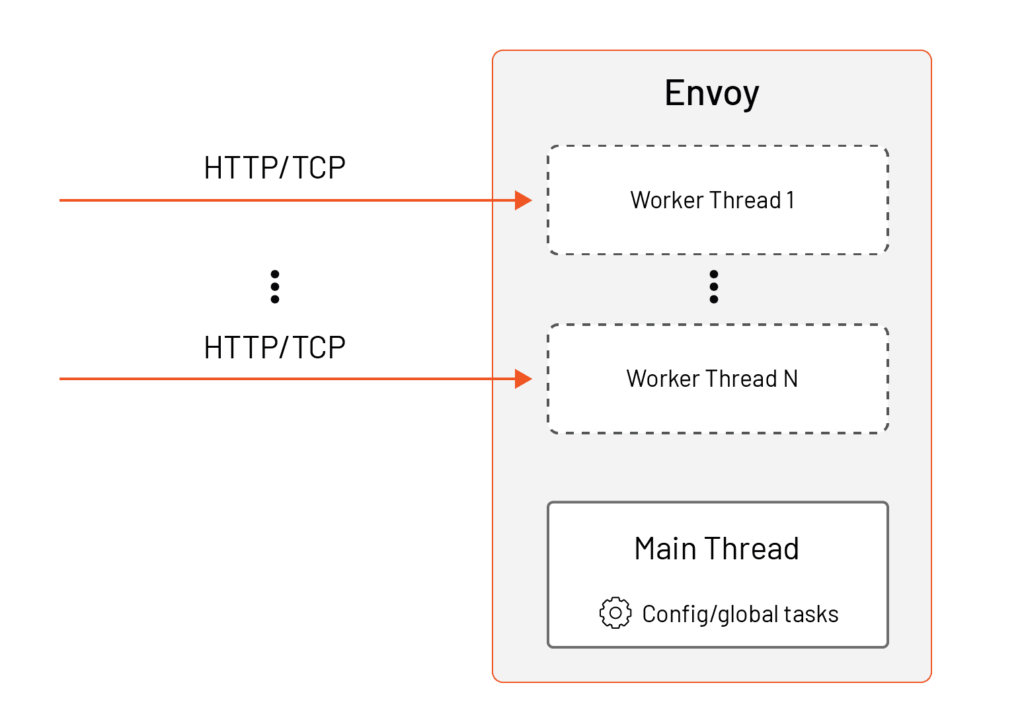
Wasm的主要目标是在网页上实现高性能应用。例如，假设我们要用Javascript构建一个网页应用程序。我们可以用Go（或其他语言）写一些，并将其编译成一个二进制文件，即Wasm模块。然后，我们可以在与Javascript网页应用程序相同的沙盒中运行已编译的Wasm模块。

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连接的工作线程。这些工作线程被设计为相互独立。例如，处理一个HTTP请求的工人线程不会受到其他处理其他请求的工人线程的影响。



每个线程拥有它的资源副本，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。

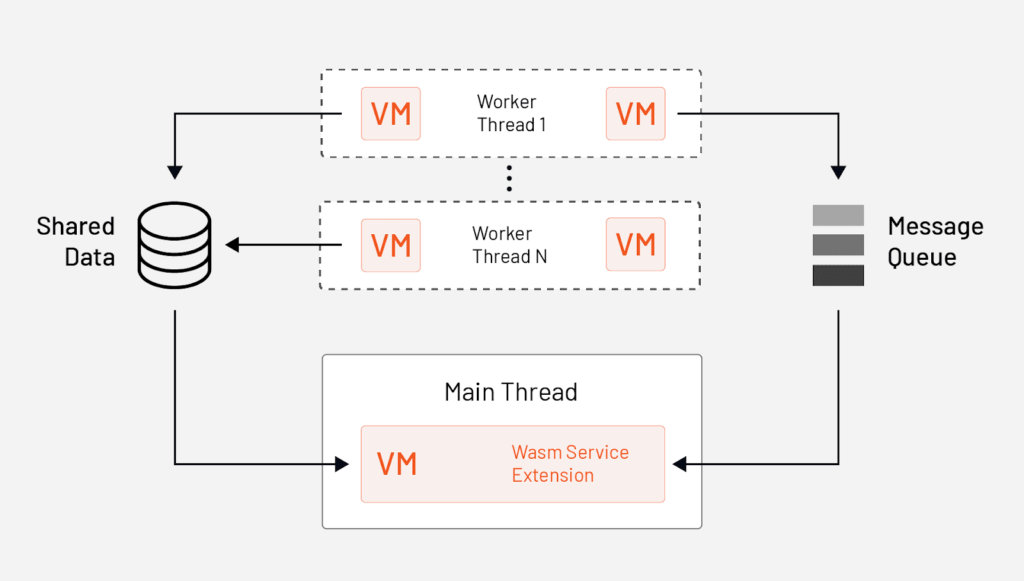
## 催产剂-Wasm

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虚拟机内的工作线程（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头、主体等进行操作。同样，网络过滤器处理TCP协议，对数据帧和连接进行操作。我们也可以说，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--这基本上意味着跨越了许多工作线程。对于这种情况，我们会使用Wasm服务类型。Wasm服务类型运行在一个单子虚拟机上；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，后者是在工作线程上执行。



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。

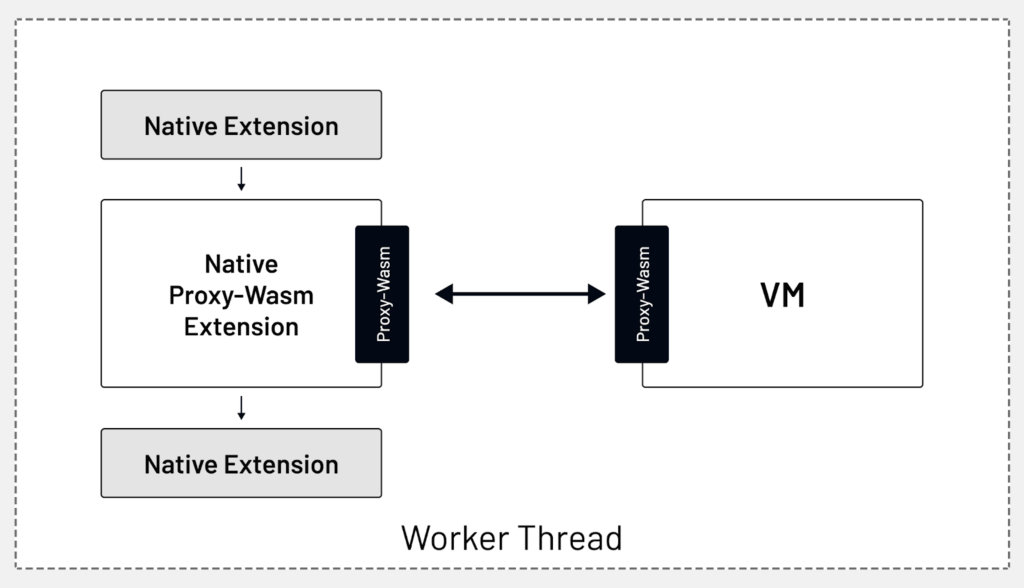
图中显示了在主线程上运行的Wasm服务扩展，它使用消息队列API订阅队列并接收由运行在工作线程上的HTTP过滤器或网络过滤器发送的消息。然后，Wasm服务扩展可以聚合从工作线程上收到的数据。

Wasm服务扩展并不是持久化数据的唯一方法。你也可以呼出HTTP或gRPC APIs。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一个叫做 [Proxy-Wasm](https://github.com/proxy-wasm).

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：Envoy）一起工作。[MOSN](https://github.com/mosn/mosn)，例如）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。



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对话。 一旦过滤器处理了数据，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](https://github.com/tetratelabs/proxy-wasm-go-sdk)来编写Go中的Proxy-Wasm插件。

[TinyGo](https://tinygo.org/)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。

你还可以选择使用汇编脚本、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。

- 名称： envoy. filters.http.wasm  
 typed\_config:  
 "@type": type.googleapis.com/udpa.type.v1.TypedStruct  
 type\_url: type.googleapis.com/envoy.extensions. filters.http.wasm.v3.Wasm  
 值。  
 config:  
 vm\_config:  
 vm\_id:"my\_vm"  
 runtime:"envoy.wasm.runtime.v8"  
 配置。  
 "@type": type.googleapis.com/google.protobuf.StringValue  
 值：'{"插件-配置"。"some-value"}'  
 代码。  
 本地。  
 文件名： "my-plugin.wasm"  
 配置。  
 "@类型"： type.googleapis.com/google.protobuf.StringValue  
 值：'{"vm-wide-config":"some-value"}'

vm\_config字段用于指定Wasm虚拟机、运行时间，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、运行时间、配置和代码。

下一个项目是运行时间。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和运行时间外，另一个重要的部分是代码字段。

代码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）。

配置文件，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，并将单子布尔字段的值设置为真。

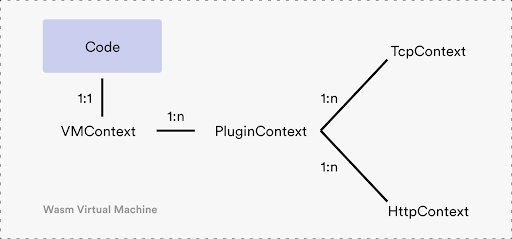
bootstrap\_extensions。  
- name: envoy.bootstrap.wasm  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.wasm.3.WasmService  
 singleton: true  
 配置。  
 vm\_config:{ ...}

## 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。

### 背景

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。



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 接口 {  
 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 接口 {  
 OnPluginStart(pluginConfigurationSize int) OnPluginStartStatus  
 OnPluginDone() bool  
  
 OnQueueReady(queueID uint32)  
 控件(OnTick)  
  
 NewTcpContext(contextID uint32) TcpContext  
 NewHttpContext(contextID uint32) HttpContext  
}

OnPluginStart函数与我们前面提到的OnVMStart函数相类似。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）。

中的context.go文件，以获得最新的接口定义。 [Proxy Wasm Go SDK Github repository](https://github.com/tetratelabs/proxy-wasm-go-sdk/blob/main/proxywasm/types/context.go)获取最新的接口定义。

### 主机呼叫API

实现的hostcall API [here](https://github.com/tetratelabs/proxy-wasm-go-sdk/blob/main/proxywasm/hostcall.go)，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头文件、拖车和正文并操作这些值；配置指标；以及更多。

### 入境点

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{}.