ALIBABA CLOUD

阿里云

服务网格

流量管理

文档版本：20220617

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.使用ASM网关的流量路由功能

ASM网关提供了图形化创建目标规则和虚拟服务的功能，无需编写YAML文件，简化流量管理操作。本文介绍如何使用图形化的方式创建流量策略和路由策略。

前提条件

- 已创建ASM实例，且版本为专业版、企业版或旗舰版。具体操作，请参见创建ASM实例。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。
- 已部署入口网关服务。具体操作，请参见添加入口网关服务。
- 已部署应用到ASM实例的集群中。具体操作，请参见部署应用到ASM实例。
- 为命名空间注入Sidecar。具体操作，请参见多种方式灵活开启自动注入。
- 获取ASM网关地址。具体操作，请参见访问入口网关。

背景信息

本文以Reviews服务为例，使用图形化的方式设置服务的负载均衡、连接池等流量策略，流量策略创建成功后，将自动生成对应目标规则的YAML文件。以Bookinfo服务为例，使用图形化的方式为Bookinfo服务创建*/productpage*、*/login*等路由策略，从而可以通过*/productpage*等路径访问到Bookinfo服务。

创建流量策略

1. 导入上游服务。

   在ASM网关中导入服务，从而关联网关与服务。

   i. 登录ASM控制台。

   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。

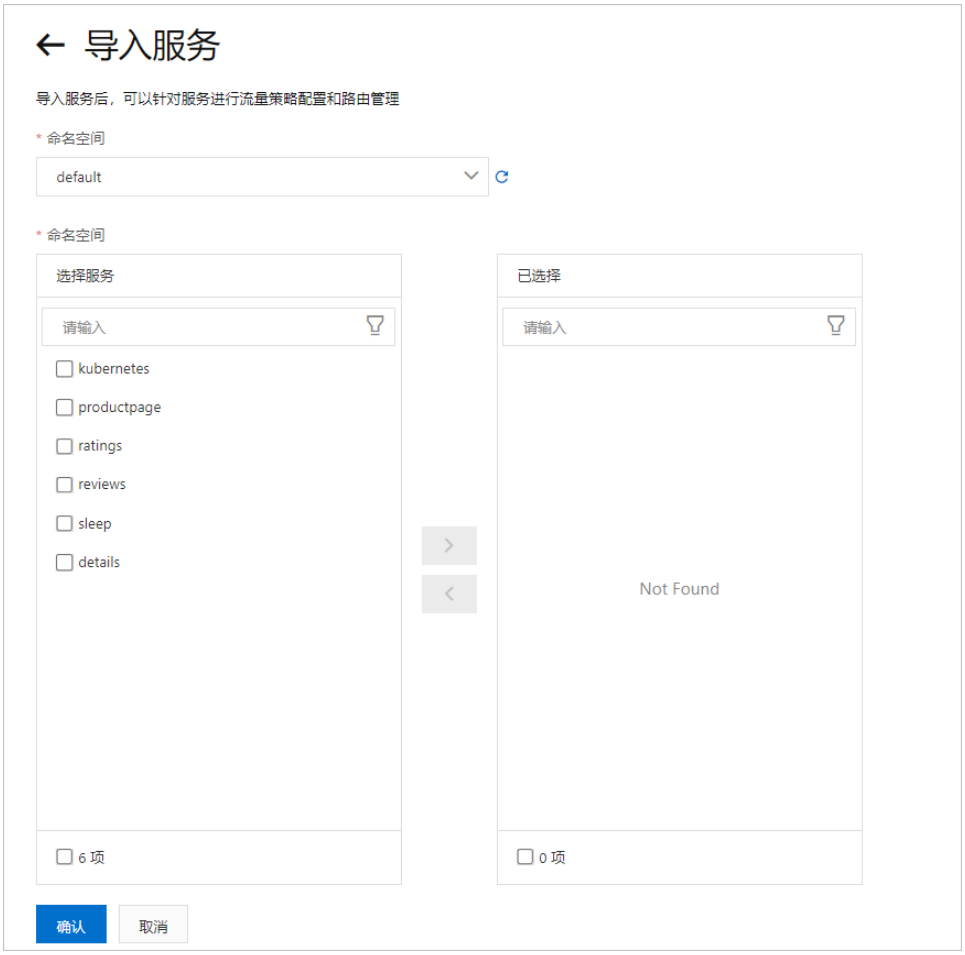
   iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。

   iv. 在网格详情页面左侧导航栏单击**ASM网关**，在右侧页面单击目标网关的名称。

   v. 在网关详情页面左侧导航栏单击**上游服务**。

   vi. 在网关上游服务页面单击**导入服务**。

vii. 在**导入服务**页面选择**命名空间**，选中reviews服务，单击图标，然后单击**确认**。



2. 对服务进行版本管理。

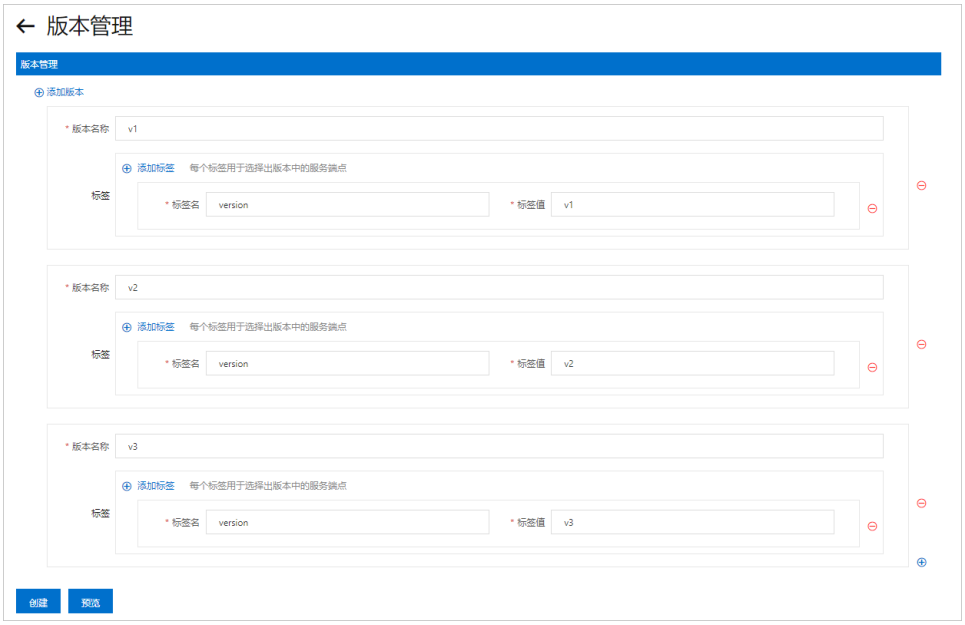
   按版本给服务的实例进行分组，本文以reviews服务为例，将reviews服务分为v1、v2、v3。

   i. 在**上游服务**页面单击reviews服务右侧操作列下的**版本管理**。

   ii. 在**版本管理**页面单击**添加版本**，输入**版本名称**为*v1*，单击**添加标签**，设置**标签名**为*version*，**标签值**为*v1*。

   iii. 单击**添加版本**，输入**版本名称**为*v2*，单击**添加标签**，设置**标签名**为*version*，**标签值**为*v2*。

iv. 再次单击**添加版本**，输入**版本名称**为*v3*，单击**添加标签**，设置**标签名**为*version*，**标签值**为*v3*，然后单击**创建**。



3. 创建流量策略。

i. 在**上游服务**页面单击reviews服务右侧**操作**列下的**流量策略**。

ii. 在**流量策略**页面单击**添加策略**，选择**版本流量策略**，选择v3版本，打开**负载均衡**开关，选择**简单模式**，设置**均衡方式**为**随机**，然后单击**创建**。

4. （可选）查看流量策略创建成功后生成的目标规则的YAML文件。

   在**上游服务**页面单击reviews服务右侧**操作**列下的**YAML**，在**预览**面板查看YAML文件。

   > **说明** 您也可以在网格详情页面左侧导航栏选择**流量管理** > **目标规则**，在**目标规则**页面单击目标规则右侧**操作**列下的**YAML**，在**编辑**面板查看生成的目标规则的YAML文件。

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: reviews
  namespace: default
  labels:
    provider: asm
spec:
  host: reviews
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
    - name: v3
      labels:
        version: v3
      trafficPolicy:
        loadBalancer:
          simple: RANDOM
```

创建路由策略

1. 导入上游服务。

   在ASM网关中导入服务，从而关联网关与服务。

   i. 登录ASM控制台。

   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。

   iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。

   iv. 在网格详情页面左侧导航栏单击**ASM网关**，在右侧页面单击目标网关的名称。

   v. 在网关详情页面左侧导航栏单击**上游服务**。

   vi. 在**上游服务**页面单击**导入服务**。

   vii. 在**导入服务**页面选择**命名空间**，选中productpage服务，单击图标，然后单击**确认**。

2. 创建路由策略。

   i. 创建*/productpage*路由。

      a. 在网关详情页面左侧导航栏单击**路由管理**，在右侧页面单击**创建**。

      b. 在**设置路由信息**配置向导中设置参数，单击**下一步**。

参数	说明
路由类型	选择**路由类型**，本文设置路由类型为http
命名空间	选择**命名空间**，本文设置命名空间为default。
路由名称	输入路由名称，本文设置为productpage-route。
描述	输入路由描述，本文设置为productpage路由。
优先级	路由配置存在优先级关系，优先级数字越小，表示优先级越高，如果URI被多个路由规则都能进行匹配，优先匹配高优先级的，本文设置为3。
匹配URL	设置URL匹配，只有符合要求的URL才能路由到目标服务。 打开**匹配URL**开关，设置**匹配方式**为**精确**，**匹配内容**为*/productpage*。 **说明** 您也可以单击**添加Header匹配规则**，设置Header匹配。

c. 在**设置路由目的地**配置向导中单击**添加路由目的地**，设置参数，单击**下一步**。

参数	说明
选择上游服务	设置路由策略生效的服务。本文选择*productpage*。
版本	设置路由策略生效的版本。
权重	设置路由的目标服务的流量权重。

d. 在**高级选项**配置向导中打开**故障注入**开关，打开**请求延时**开关，设置**延时值**为*4s*，**故障注入百分比**为*100*，单击**下一步**。

e. 在**路由配置确认**配置向导中确认配置的路由信息，确认无误后，单击**创建**。

ii. 创建*/static*路由。

a. 在**路由管理**单击**创建**。

b. 在**设置路由信息**配置向导中设置参数，单击**下一步**。

参数	说明
路由类型	选择**路由类型**，本文设置路由类型为http
命名空间	选择**命名空间**，本文设置命名空间为default。
路由名称	输入路由名称，本文设置为static。
描述	输入路由描述，本文设置为static资源。
优先级	路由配置存在优先级关系，优先级数字越小，表示优先级越高，如果URI被多个路由规则都能进行匹配，优先匹配高优先级的，本文设置为0。
匹配URL	设置URL匹配，只有符合要求的URL才能路由到目标服务。 打开**匹配URL**开关，设置**匹配方式**为**前缀**，**匹配内容**为*/static*。

c. 创建*/static*路由的后续步骤与创建*/productpage*路由相同，具体操作，请参见创建productpage路由 。

iii. 创建*/login*路由。

a. 在**路由管理**单击**创建**。

b. 在**设置路由信息**配置向导中设置参数，单击**下一步**。

参数	说明
路由类型	选择**路由类型**，本文设置路由类型为http
命名空间	选择**命名空间**，本文设置命名空间为default。
路由名称	输入路由名称，本文设置为login。
描述	输入路由描述，本文设置为登录请求路由。
优先级	路由配置存在优先级关系，优先级数字越小，表示优先级越高，如果URI被多个路由规则都能进行匹配，优先匹配高优先级的，本文设置为0。
匹配URL	设置URL匹配，只有符合要求的URL才能路由到目标服务。 打开**匹配URL**开关，设置**匹配方式**为**前缀**，**匹配内容**为*/login*。

c. 创建*/login*路由的后续步骤与创建*/productpage*路由相同，具体操作，请参见创建productpage路由 。

iv. 创建*/logout*路由。

a. 在**路由管理**单击**创建**。

b. 在**设置路由信息**配置向导中设置参数，单击**下一步**。

参数	说明
路由类型	选择**路由类型**，本文设置路由类型为http
命名空间	选择**命名空间**，本文设置命名空间为default。
路由名称	输入路由名称，本文设置为logout。
描述	输入路由描述，本文设置为登出路由。
优先级	路由配置存在优先级关系，优先级数字越小，表示优先级越高，如果URI被多个路由规则都能进行匹配，优先匹配高优先级的，本文设置为0。
匹配URL	设置URL匹配，只有符合要求的URL才能路由到目标服务。 打开**匹配URL**开关，设置**匹配方式**为**前缀**，**匹配内容**为*/logout*。

c. 创建*/logout*路由的后续步骤与创建*/productpage*路由相同，具体操作，请参见创建productpage路由 。

v. 创建*/api/v1/products*路由。

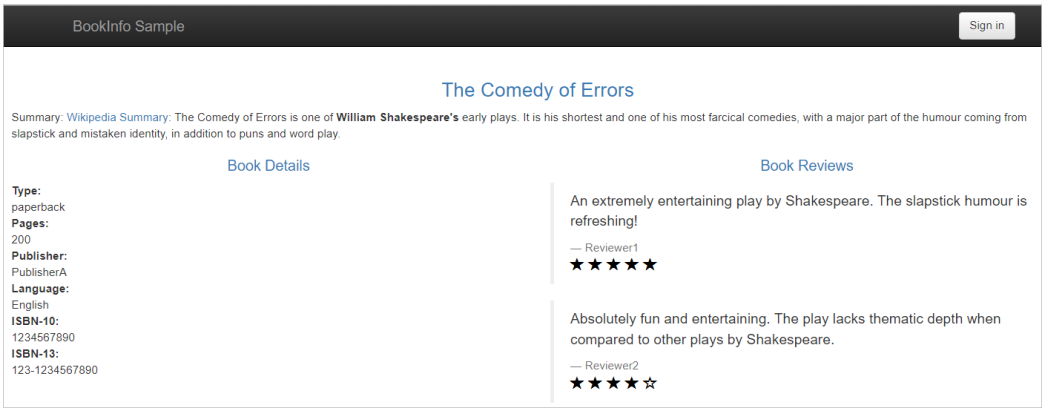
a. 在**路由管理**单击**创建**。

b. 在**设置路由信息**配置向导中设置参数，单击**下一步**。

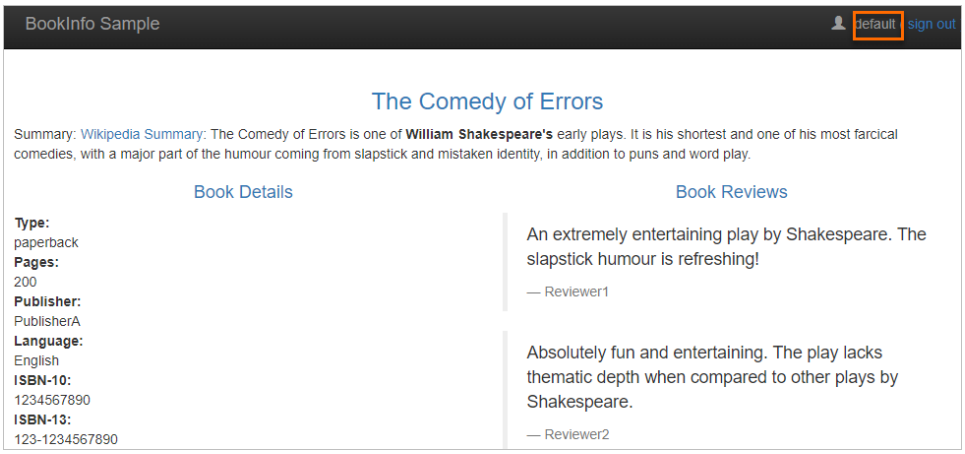
参数	说明
路由类型	选择**路由类型**，本文设置路由类型为http
命名空间	选择**命名空间**，本文设置命名空间为default。
路由名称	输入路由名称，本文设置为products-route。
描述	输入路由描述，本文设置为product信息。
优先级	路由配置存在优先级关系，优先级数字越小，表示优先级越高，如果URI被多个路由规则都能进行匹配，优先匹配高优先级的，本文设置为0。
匹配URL	设置URL匹配，只有符合要求的URL才能路由到目标服务。 打开**匹配URL**开关，设置**匹配方式**为**前缀**，**匹配内容**为*/api/v1/products*。

c. 创建*/api/v1/products*路由的后续步骤与创建*/productpage*路由相同，具体操作，请参见创建productpage路由 。

3. 验证路由策略是否生效。
   i. 在谷歌浏览器地址栏输入*http://{入口网关的IP地址}/productpage*。
   可以看到以下页面。



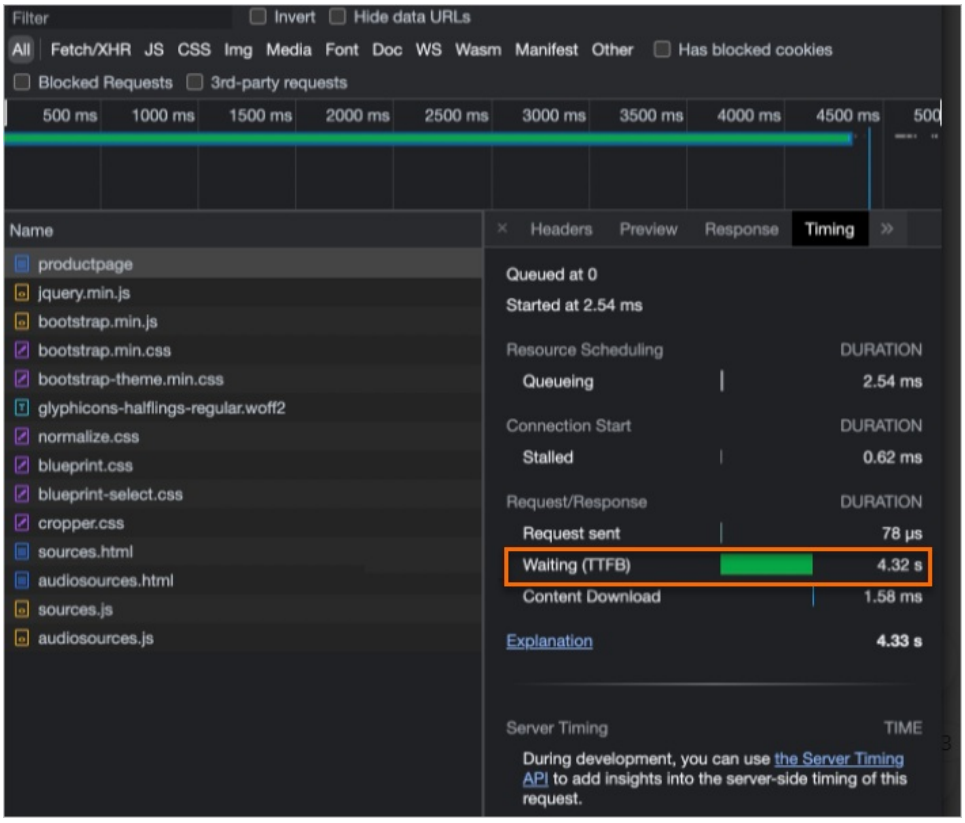
   ii. 在Bookinfo页面右上角单击**Sign in**。
   iii. 在**Please sign in**输入任意账号和密码，单击**Sign in**。
   可以看到能够登录到在Bookinfo。



   您还可以使用*/logout*、*/static*、*/api/v1/products*访问Bookinfo服务，这里将不一一验证。
   iv. 在谷歌浏览器右上角单击图标，选择**更多工具 > 开发者工具**。

v. 刷新地址http://{入口网关服务的IP地址}/productpage。

在**Network**页签下可以看到约4s的延迟。



4. （可选）查看路由策略创建成功后生成的虚拟服务的YAML文件。

在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，在**虚拟服务**页面单击目标虚拟服务右侧**操作**列下的**YAML**，在**编辑**面板查看生成的虚拟服务的YAML文件。

相关操作

查看服务详情

查看导入服务的是否注入Sidecar、地域等信息。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏单击**ASM网关**，在右侧页面单击目标网关的名称。
5. 在网关详情页面左侧导航栏单击**上游服务**。
6. 在**上游服务**页面单击目标服务右侧**操作**列下的**服务详情**。

在**服务详情**页面查看服务是否注入Sidecar、地域等信息。

路由下线

下线路由策略，使该路由策略失效。

> **说明** 您可以通过以下方式上线路由策略，使路由策略中重新生效：
>
> 在**路由管理**页面单击目标路由右侧**操作**列下的**上线**，在**确认**对话框单击**确定**。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏单击**ASM网关**，在右侧页面单击目标网关的名称。
5. 在网关详情页面左侧导航栏单击**路由管理**。
6. 在**路由管理**页面单击login路由右侧**操作**列下的**下线**。
7. 在**确认**对话框单击**确定**。
8. 验证路由下线是否成功。
   i. 在谷歌浏览器地址栏输入*http://{入口网关的IP地址}/productpage*。
   ii. 在Bookinfo页面右上角单击**Sign in**。
   iii. 在**Please sign in**输入任意账号和密码，单击**Sign in**。

可以看到页面显示找不到网页，说明login路由策略失效，路由下线成功。

2.通过ASM实现TCP应用流量迁移

服务网格ASM的流量管理功能可以实现应用的流量迁移。本文将通过示例介绍如何通过ASM实现TCP应用流量的迁移。

前提条件

- 已开通以下服务：
  - 容器服务
  - 负载均衡
- 已创建至少一个ACK集群。如果没有创建，请参见创建Kubernetes专有版集群和创建Kubernetes托管版集群。
- 已创建一个ASM实例，并已将ACK集群添加到ASM实例中，请参见创建ASM实例和添加集群到ASM实例。

背景信息

本文以Istio官方Task TCP-Traffic-Shifting为例来讲述如何实现在一个TCP服务的两个版本之间进行流量灰度切换。该Task中的服务是一个简单地Echo服务，在v1版本中，该服务在收到的数据在前面加上“one”并返回；在v2版本中，该服务在收到的数据前面加上“two”并返回。

步骤一：部署示例应用

1. 部署TCP- Echo应用的2个版本。
   i. 登录容器服务管理控制台。
   ii. 在控制台左侧导航栏中，单击**集群**。
   iii. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**应用管理**。
   iv. 在**无状态**页面**命名空间**下拉列表中选择命名空间。
   v. 单击右上方的**使用YAML创建资源**。
   vi. 设置**示例模板**为**自定义**，将下面的YAML模版粘贴到**模版**文本框中，单击**创建**。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tcp-echo-v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tcp-echo
      version: v1
  template:
    metadata:
      labels:
        app: tcp-echo
        version: v1
    spec:
      containers:
      - name: tcp-echo
        image: docker.io/istio/tcp-echo-server:1.1
        imagePullPolicy: IfNotPresent
        args: [ "9000", "one" ]
        ports:
        - containerPort: 9000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tcp-echo-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tcp-echo
      version: v2
  template:
    metadata:
      labels:
        app: tcp-echo
        version: v2
    spec:
      containers:
      - name: tcp-echo
        image: docker.io/istio/tcp-echo-server:1.1
        imagePullPolicy: IfNotPresent
        args: [ "9000", "two" ]
        ports:
        - containerPort: 9000
```

在**无状态**页面可以看到新创建的两个版本的TCP-Echo应用。

2. 创建一个服务，并将其对外暴露。
   i. 登录容器服务管理控制台。
   ii. 在控制台左侧导航栏中，单击**集群**。

iii. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**应用管理**。

iv. 在集群管理页左侧导航栏中，选择**网络 > 服务**。

v. 在**服务**页面**命名空间**下拉列表中选择命名空间，单击右上角的**创建**。

vi. 在**创建服务**对话框中，设置服务的相关信息，然后单击**创建**。

需要设置的参数如下所示：

- **名称**：设为tcp-echo。
- **类型**：选择服务类型，即服务访问的方式。支持**虚拟集群IP**、**节点端口**和**负载均衡**。

  > **说明** 您的服务类型为**虚拟集群IP**时，才能设置**实例间发现服务（Headless Service）**。您可以使用无头Service与其他服务发现机制进行接口，而不必与Kubernetes的实现捆绑在一起。

- **关联**：选择tcp-echo-v1。

  > **说明** 将从关联的Deployment中提取 `app` 这个标签作为Service的Selector，这决定了Kubernetes service将流量转发到哪个Deployment。由于tcp-echo-v1和tcp-echo-v2两个Deployment拥有相同的标签，即 `app:tcp-echo` ，因此可以关联任一Deployment。

- **外部流量策略**：可选值为Local或Cluster。

  > **说明** 您的服务类型为**节点端口**或**负载均衡**时，才能设置**外部流量策略**。

- **端口映射**：**名称**设为tcp；**服务端口**和**容器端口**设为9000；**协议**设为TCP。
- **注解**：为该服务添加一个注解（annotation），配置负载均衡的参数，例如设置 `service.beta.kubernetes.io/alicloud-loadbalancer-bandwidth:20` 表示将该服务的带宽峰值设置为20Mbit/s，从而控制服务的流量。更多参数请参见通过Annotation配置负载均衡。
- **标签**：您可为该服务添加一个标签，标识该服务。

在**服务（Service）**页面可以看到新创建的服务，type-echo。

步骤二：设置服务网格ASM的路由规则

通过设置服务网格的Istio网关、虚拟服务和目标规则，将流量全部指向tcp-echo服务的v1版本。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 创建服务网关。

   i. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，然后在右侧页面单击**使用YAML创建**。

ii. 从**命名空间**下拉列表中选择default，并在文本框中输入以下YAML文件内容，单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: tcp-echo-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 31400
      name: tcp
      protocol: TCP
    hosts:
    - "*"
```

5. 创建虚拟服务。

i. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。

ii. 从**命名空间**下拉列表中选择default，并在文本框中输入以下YAML文件内容，单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: tcp-echo
spec:
  hosts:
  - "*"
  gateways:
  - tcp-echo-gateway
  tcp:
  - match:
    - port: 31400
    route:
    - destination:
        host: tcp-echo
        port:
          number: 9000
        subset: v1
```

6. 创建目标规则。

i. 在网格详情页面左侧导航栏选择**流量管理 > 目标规则**，然后在右侧页面单击**使用YAML创建**。

ii. 从**命名空间**下拉列表中选择default，并在文本框中输入以下YAML文件内容，单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: tcp-echo-destination
spec:
  host: tcp-echo
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

步骤三：部署入口网关

在入口网关中，添加端口31400，并指向Istio网关的31400端口。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏单击**ASM网关**，然后在右侧页面单击**创建**。
5. 设置入口网关参数， 其他采用默认值。
   i. 从**部署集群**列表中选择要部署入口网关的集群。
   ii. 指定负载均衡的类型为**公网访问**。
   iii. 选择负载均衡。
   - 使用已有负载均衡：从已有负载均衡列表中选择。
   - 新建负载均衡：单击**新建负载均衡**，从下拉列表中选择所需的负载均衡规格。

   > **说明** 建议您为每个Kubernetes服务分配一个SLB。如果多个Kubernetes服务复用同一个SLB，存在以下风险和限制：
   > - 使用已有的SLB会强制覆盖已有监听，可能会导致您的应用不可访问。
   > - Kubernetes通过Service创建的SLB不能复用，只能复用您手动在控制台（或调用OpenAPI）创建的SLB。
   > - 复用同一个SLB的多个Service不能有相同的前端监听端口，否则会造成端口冲突。
   > - 复用SLB时，监听的名字以及虚拟服务器组的名字被Kubernetes作为唯一标识符。请勿修改监听和虚拟服务器组的名字。
   > - 不支持跨集群复用SLB。

   iv. 单击**添加端口**，将**名称**设为tcp，**服务端口**和**容器端口**设为31400。

   > **说明** 服务端口指的是整个网格对外暴露的端口，是外部访问使用的端口；而容器端口指的是从服务端口进来的流量所指向的Istio网关端口，所以容器端口需要与Istio网关一致。

   v. 单击**创建**。

步骤四：检查部署结果

通过Kubectl确认tcp-echo服务的流量指向是否符合预期。

1. 通过Kubectl客户端连接至Kubernetes集群。详情请参见步骤二：选择集群凭证类型。
2. 执行以下命令，获得服务的地址与端口。

```
export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o jso
npath='{.status.loadBalancer.ingress[0].ip}')
export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o jso
npath='{.spec.ports[?(@.name=="tcp")].port}')
```

3. 使用telnet指令向tcp-echo服务发起连接请求。

```
telnet $INGRESS_HOST $INGRESS_PORT
Trying xxx.xxx.xxx.xxx...
Connected to xxx.xxx.xxx.xxx.
Escape character is '^]'
```

4. 输入任意字符串，按Enter发送，返回的字符串前面带了“one”。这说明tcp-echo服务已经成功部署，且流量全部指向了tcp-echo-v1版本。

步骤五：按比例将流量路由到tcp-echo-v2

此处将20%的流量指向tcp-echo-v2版本，其余80%仍然打到tcp-echo-v1。

1. 修改ASM实例的虚拟服务配置。
   i. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**。
   ii. 在**虚拟服务**页面，单击tcp-echo所在**操作**列中的**YAML**。

iii. 在**编辑实例**页面的文本框中输入以下YAML文件内容，单击**确定**。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: tcp-echo
spec:
  hosts:
  - "*"
  gateways:
  - tcp-echo-gateway
  tcp:
  - match:
    - port: 31400
    route:
    - destination:
        host: tcp-echo
        port:
          number: 9000
        subset: v1
      weight: 80
    - destination:
        host: tcp-echo
        port:
          number: 9000
        subset: v2
      weight: 20
```

2. 执行以下命令，向tcp-echo服务发起10次请求。

```
 for i in {1..10}; do \
docker run -e INGRESS_HOST=$INGRESS_HOST -e INGRESS_PORT=$INGRESS_PORT -it --rm busyb
ox sh -c "(date; sleep 1) | nc $INGRESS_HOST $INGRESS_PORT"; \
done
one Mon Nov 12 23:38:45 UTC 2018
two Mon Nov 12 23:38:47 UTC 2018
one Mon Nov 12 23:38:50 UTC 2018
one Mon Nov 12 23:38:52 UTC 2018
one Mon Nov 12 23:38:55 UTC 2018
two Mon Nov 12 23:38:57 UTC 2018
one Mon Nov 12 23:39:00 UTC 2018
one Mon Nov 12 23:39:02 UTC 2018
one Mon Nov 12 23:39:05 UTC 2018
one Mon Nov 12 23:39:07 UTC 2018
```

根据以上请求的分发情况，可以看到20%的流量指向了tcp-echo-v2。

> **说明** 您的测试结果可能不一定总是10次中有2次打到tcp-echo-v2，但从较长时间范围的总体比例来看，一定是接近20% 的。

3.使用ASM为网格内gRPC服务实现负载均衡

在使用gRPC（基于HTTP/2）的Kubernetes服务时，到目标的单个连接将在一个Pod处终止。如果从客户端发送了多条消息，则所有消息将由该Pod处理，从而导致负载不均衡。本文通过示例介绍gRPC服务间负载不均衡的问题以及如何实现负载均衡。

背景信息

gRPC是一种基于HTTP/2的服务通信协议，使用基于Protocol Buffers（简称为PB）格式的服务定义。服务之间调用的数据可以被序列化为较小的二进制格式进行传输。使用gRPC，可以从*.proto*文件生成多种语言的代码，这也就使得gRPC成为了多语言微服务开发的最佳选择之一。

使用基于HTTP/1.1的RPC时，一个简单的TCP负载平衡器足以胜任，因为这些连接都是短暂的，客户端将尝试重新连接，不会保持与运行中的旧Pod之间的连接。但是使用基于HTTP/2的gRPC时，TCP连接保持打开状态，这样将保持连接到即将失效的Pod，亦或使集群失去平衡。

gRPC服务间调用的负载不均衡

通过一个以下示例可以看出在Kubernetes下gRPC服务间调用的负载不均衡问题。

前提条件：

- 已创建至少一个Kubernetes集群。
- 已设置通过kubectl连接该集群，详情参见通过kubectl工具连接集群。

操作步骤：

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，单击**命名空间与配额**。
5. 在**命名空间**页面单击右上方的**创建**，在**创建命名空间**对话框，输入命名空间的名称，例如grpc-nosidecar，单击**确定**。
6. 在命名空间grpc-nosidecar下部署gRPC服务的服务端istio-grpc-server。

   假设，待创建的描述文件为*istio-grpc-server.yaml*，请执行如下命令：

   ```
   kubectl apply -n grpc-nosidecar -f istio-grpc-server.yaml
   ```

   其中，*istio-grpc-server.yaml*文件的内容如下：

   ```
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: istio-grpc-server-v1
     labels:
       app: istio-grpc-server
       version: v1
   spec:
     replicas: 1
     selector:
       matchLabels:
   ```

```
      app: istio-grpc-server
      version: v1
  template:
    metadata:
      labels:
        app: istio-grpc-server
        version: v1
    spec:
      containers:
      - args:
        - --address=0.0.0.0:8080
        image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/istio-grpc-server
        imagePullPolicy: Always
        livenessProbe:
          exec:
            command:
            - /bin/grpc_health_probe
            - -addr=:8080
          initialDelaySeconds: 2
        name: istio-grpc-server
        ports:
        - containerPort: 8080
        readinessProbe:
          exec:
            command:
            - /bin/grpc_health_probe
            - -addr=:8080
          initialDelaySeconds: 2
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: istio-grpc-server-v2
  labels:
    app: istio-grpc-server
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: istio-grpc-server
      version: v2
  template:
    metadata:
      labels:
        app: istio-grpc-server
        version: v2
    spec:
      containers:
        - args:
            - --address=0.0.0.0:8080
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/istio-grpc-serv
er
          imagePullPolicy: Always
          livenessProbe:
```

```
livenessProbe:
  exec:
    command:
      - /bin/grpc_health_probe
      - -addr=:8080
  initialDelaySeconds: 2
name: istio-grpc-server
ports:
  - containerPort: 8080
readinessProbe:
  exec:
    command:
      - /bin/grpc_health_probe
      - -addr=:8080
  initialDelaySeconds: 2
---
apiVersion: v1
kind: Service
metadata:
  name: istio-grpc-server
  labels:
    app: istio-grpc-server
spec:
  ports:
  - name: grpc-backend
    port: 8080
    protocol: TCP
  selector:
    app: istio-grpc-server
  type: ClusterIP
---
```

7. 在命名空间grpc-nosidecar下部署gRPC服务的客户端istio-grpc-client。

   假设，待创建的描述文件为*istio-grpc-client.yaml*，请执行如下命令：

   ```
   kubectl apply -n grpc-nosidecar -f istio-grpc-client.yaml
   ```

   其中，*istio-grpc-client.yaml*文件的内容如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: istio-grpc-client
  labels:
    app: istio-grpc-client
spec:
  replicas: 1
  selector:
    matchLabels:
      app: istio-grpc-client
  template:
    metadata:
      labels:
        app: istio-grpc-client
    spec:
      containers:
      - image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/istio-grpc-client
        imagePullPolicy: Always
        command: ["/bin/sleep", "3650d"]
        name: istio-grpc-client
---
apiVersion: v1
kind: Service
metadata:
  name: istio-grpc-client
spec:
  ports:
  - name: grpc
    port: 8080
    protocol: TCP
  selector:
    app: istio-grpc-client
  type: ClusterIP
---
```

8. 执行以下命令，查看Pod运行状态：

```
kubectl get pod -n grpc-nosidecar
```

示例输出如下：

```
NAME                                    READY   STATUS    RESTARTS   AGE
istio-grpc-client-dd56bcb45-hvmjt       1/1     Running   0          95m
istio-grpc-server-v1-546d9876c4-j2p9r   1/1     Running   0          95m
istio-grpc-server-v2-66d9b8847-276bd    1/1     Running   0          95m
```

9. 执行以下命令，登录到客户端Pod容器。

```
kubectl exec -it -n grpc-nosidecar istio-grpc-client-dd56bcb45-hvmjt sh
```

10. 进入容器后，执行以下命令：

```
/bin/greeter-client --insecure=true --address=istio-grpc-server:8080 --repeat=100
```

可以看到所有的请求都指向了一个Pod：

```
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
2020/01/14 14:37:14 Hello world from istio-grpc-server-v2-66d9b8847-276bd
```

由此可见，从客户端发送了所有消息都由一个Pod处理，从而导致负载不均衡。

使用ASM实现负载均衡

下面将示例说明如何通过ASM实现负载均衡。

前提条件：

- 已创建至少一个服务网格ASM实例，并已经添加至少一个集群到该实例中。
- 已设置通过kubectl连接到该集群，详情参见通过kubectl工具连接集群。

操作步骤：

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，单击**命名空间与配额**。
5. 在**命名空间**页面单击右上方的**创建**，在**创建命名空间**对话框，输入命名空间的名称，例如grpc-sidecar，并新增标签 `istio-injection:enabled` ，单击**确定**。
6. 命名空间grpc-sidecar下部署gRPC服务的服务端istio-grpc-server。

   假设，待创建的描述文件为*istio-grpc-server.yaml*，请执行如下命令：

   ```
   kubectl apply -n grpc-sidecar -f istio-grpc-server.yaml
   ```

   *istio-grpc-server.yaml*文件的内容，参见上文示例。
7. 命名空间grpc-sidecar下部署gRPC服务的客户端istio-grpc-client。

   假设，待创建的描述文件为*istio-grpc-client.yaml*，请执行如下命令：

   ```
   kubectl apply -n grpc-sidecar -f istio-grpc-client.yaml
   ```

   *istio-grpc-client.yaml*文件的内容，参见上文示例。
8. 执行以下命令，查看Pod运行状态：

   ```
   kubectl get pod -n grpc-sidecar
   ```

   此时可以看到每个Pod中包含了2个容器，其中一个容器就是注入的Sidecar代理，示例输出如下：

```
 NAME                                     READY   STATUS    RESTARTS   AGE
istio-grpc-client-dd56bcb45-zhfsg        2/2     Running   0          1h15m
istio-grpc-server-v1-546d9876c4-tndsm    2/2     Running   0          1h15m
istio-grpc-server-v2-66d9b8847-99v62     2/2     Running   0          1h15m
```

9. 执行以下命令，登录到客户端Pod容器。

```
kubectl exec -it -n grpc-nosidecar istio-grpc-client-695f5fc66f-2brmv -c istio-grpc-c
lient sh
```

10. 进入容器后，执行以下命令：

```
/bin/greeter-client --insecure=true --address=istio-grpc-server:8080 --repeat=100
```

可以看到所有的请求分别指向了对应的2个Pod，比例接近于50：50，即负载均衡中的Round-Robin。

```
2020/01/14 14:53:16 Hello world from istio-grpc-server-v1-546d9876c4-tndsm
2020/01/14 14:53:16 Hello world from istio-grpc-server-v2-66d9b8847-99v62
2020/01/14 14:53:16 Hello world from istio-grpc-server-v1-546d9876c4-tndsm
2020/01/14 14:53:16 Hello world from istio-grpc-server-v2-66d9b8847-99v62
2020/01/14 14:53:16 Hello world from istio-grpc-server-v1-546d9876c4-tndsm
2020/01/14 14:53:16 Hello world from istio-grpc-server-v2-66d9b8847-99v62
2020/01/14 14:53:16 Hello world from istio-grpc-server-v1-546d9876c4-tndsm
2020/01/14 14:53:16 Hello world from istio-grpc-server-v2-66d9b8847-99v62
2020/01/14 14:53:16 Hello world from istio-grpc-server-v1-546d9876c4-tndsm
2020/01/14 14:53:16 Hello world from istio-grpc-server-v2-66d9b8847-99v62
2020/01/14 14:53:16 Hello world from istio-grpc-server-v1-546d9876c4-tndsm
2020/01/14 14:53:16 Hello world from istio-grpc-server-v2-66d9b8847-99v62
```

11. 配置服务网格ASM实例的控制平面。
    i. 登录ASM控制台。
    ii. 在左侧导航栏，选择**服务网格 > 网格管理**。
    iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
    iv. 在网格详情页面左侧导航栏选择**网格实例 > 全局命名空间**，然后在右侧页面单击**新建**。
    v. 在**新建命名空间**对话框设置命名空间名称为grpc-sidecar，单击**确定**。

vi. 使用以下内容，创建目标规则。具体操作，请参见管理目标规则。

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: dr-istio-grpc-server
spec:
  host: istio-grpc-server
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
    - name: v1
      labels:
        version: "v1"
    - name: v2
      labels:
        version: "v2"
```

vii. 使用以下内容创建虚拟服务，具体操作，请参见管理虚拟服务。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-istio-grpc-server
spec:
  hosts:
    - "istio-grpc-server"
  http:
    - match:
        - port: 8080
      route:
        - destination:
            host: istio-grpc-server
            subset: v1
          weight: 90
        - destination:
            host: istio-grpc-server
            subset: v2
          weight: 10
```

viii. 执行以下命令，登录到客户端Pod容器。

```
kubectl exec -it -n grpc-nosidecar istio-grpc-client-695f5fc66f-2brmv -c istio-gr
pc-client sh
```

ix. 进入容器后，执行以下命令：

```
/bin/greeter-client --insecure=true --address=istio-grpc-server:8080 --repeat=100
```

可以看到所有的请求分别指向了对应的2个Pod，但比例接近于90：10，并非默认的Round-Robin。

x. 在网格详情页面左侧导航栏单击**虚拟服务**，找到grpc-sidecar命名空间下名为vs-istio-grpc-server的虚拟服务，单击**YAML**，调整虚拟服务的权重，以查看调用的不同结果。

修改内容如下：

```
route:
   - destination:
       host: istio-grpc-server
       subset: v1
     weight: 0
   - destination:
       host: istio-grpc-server
       subset: v2
     weight: 100
```

xi. 通过Kubectl登录容器，执行以下命令：

```
/bin/greeter-client --insecure=true --address=istio-grpc-server:8080 --repeat=100
```

可以看到所有的请求分别指向了对应的1个Pod，即版本v2对应的Pod。

4.在ASM中通过EnvoyFilter添加HTTP响应头

在应用程序中添加HTTP响应头可以提高Web应用程序的安全性。本文介绍如何通过定义EnvoyFilter添加HTTP响应头。

前提条件

- 已创建一个ASM实例，并已将ACK集群添加到ASM实例中。具体操作，请参见创建ASM实例和添加集群到ASM实例。
- 通过kubectl连接ASM实例。
- 部署应用到ASM实例。
- 定义Istio资源。

背景信息

OWASP提供了最佳实践指南和编程框架，描述了如何使用安全响应头保护应用程序的安全。HTTP响应头的基准配置如下。

HTTP响应头基准配置

HTTP响应头	默认值	描述
Content-Security-Policy	frame-ancestors none;	防止其他网站进行Clickjacking攻击。
X-XSS-Protection	1;mode=block	激活浏览器的XSS过滤器（如果可用），检测到XSS时阻止渲染。
X-Content-Type-Options	Nosniff	禁用浏览器的内容嗅探。
Referrer-Policy	no-referrer	禁用自动发送引荐来源。
X-Download-Options	noopen	禁用旧版本IE中的自动打开下载功能。
X-DNS-Prefetch-Control	off	禁用对页面上的外部链接的推测性DNS解析。
Server	envoy	由Istio的入口网关自动设置。
X-Powered-by	无默认值	去掉该值来隐藏潜在易受攻击的应用程序服务器的名称和版本。

HTTP响应头	默认值	描述
Feature-Policy	camera 'none'; microphone 'none'; geolocation 'none'; encrypted-media 'none'; payment 'none'; speaker 'none'; usb 'none';	控制可以在浏览器中使用的功能和API。

以Bookinfo应用程序为例（详情请参见部署应用到ASM实例），通过Curl命令可以看到应用程序的HTTP响应头信息如下。

```
curl -I http://{入口网关服务的IP地址}/productpage
HTTP/1.1 200 OK
content-type: text/html; charset=utf-8
content-length: 5183
server: istio-envoy
date: Tue, 28 Jan 2020 08:15:21 GMT
x-envoy-upstream-service-time: 28
```

可以看到，默认情况下，示例应用程序的入口首页响应并没有包含HTTP响应头基准配置中所述的安全相关的HTTP响应头。通过Istio EnvoyFilter可以快速添加安全相关的HTTP响应头。

操作步骤

1. 执行以下命令，部署Istio服务条目。

```
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: addheader-into-ingressgateway
  namespace: istio-system
  labels:
    asm-system: 'true'
    provider: asm
spec:
  workloadSelector:
    # select by label in the same namespace
    labels:
      istio: ingressgateway
  configPatches:
    # The Envoy config you want to modify
  - applyTo: HTTP_FILTER
    match:
      context: GATEWAY
      proxy:
        proxyVersion: '^1\.9.*'
      listener:
```

```
        filterChain:
          filter:
            name: "envoy.filters.network.http_connection_manager"
            subFilter:
              name: "envoy.filters.http.router"
    patch:
      operation: INSERT_BEFORE
      value: # lua filter specification
        name: envoy.lua
        typed_config:
          "@type": "type.googleapis.com/envoy.extensions.filters.http.lua.v3.Lua"
          inlineCode: |-
            function envoy_on_response(response_handle)
                function hasFrameAncestors(rh)
                s = rh:headers():get("Content-Security-Policy");
                delimiter = ";";
                defined = false;
                for match in (s..delimiter):gmatch("(.-)"..delimiter) do
                    match = match:gsub("%s+", "");
                    if match:sub(1, 15)=="frame-ancestors" then
                    return true;
                    end
                end
                return false;
                end
                if not response_handle:headers():get("Content-Security-Policy") then
                csp = "frame-ancestors none;";
                response_handle:headers():add("Content-Security-Policy", csp);
                elseif response_handle:headers():get("Content-Security-Policy") then
                if not hasFrameAncestors(response_handle) then
                    csp = response_handle:headers():get("Content-Security-Policy");
                    csp = csp .. ";frame-ancestors none;";
                    response_handle:headers():replace("Content-Security-Policy", csp);
                end
                end
                if not response_handle:headers():get("X-Frame-Options") then
                response_handle:headers():add("X-Frame-Options", "deny");
                end
                if not response_handle:headers():get("X-XSS-Protection") then
                response_handle:headers():add("X-XSS-Protection", "1; mode=block");
                end
                if not response_handle:headers():get("X-Content-Type-Options") then
                response_handle:headers():add("X-Content-Type-Options", "nosniff");
                end
                if not response_handle:headers():get("Referrer-Policy") then
                response_handle:headers():add("Referrer-Policy", "no-referrer");
                end
                if not response_handle:headers():get("X-Download-Options") then
                response_handle:headers():add("X-Download-Options", "noopen");
                end
                if not response_handle:headers():get("X-DNS-Prefetch-Control") then
                response_handle:headers():add("X-DNS-Prefetch-Control", "off");
                end
                if not response_handle:headers():get("Feature-Policy") then
                response_handle:headers():add("Feature-Policy"
```

```
                response_handle:headers():add("Feature-Policy",
                                              "camera 'none';"..
                                              "microphone 'none';"..
                                              "geolocation 'none';"..
                                              "encrypted-media 'none';"..
                                              "payment 'none';"..
                                              "speaker 'none';"..
                                              "usb 'none';");
                end
                if response_handle:headers():get("X-Powered-By") then
                response_handle:headers():remove("X-Powered-By");
                end
            end
EOF
```

proxyVersion：设定为您当前的Istio版本。EnvoyFilter创建时需要设置proxyVersion来指定期望作用的Istio版本范围，EnvoyFilter配置中的一些字段存在Istio版本不兼容的可能性。不同Istio版本的EnvoyFilter内容不同：

- 如果您使用的Istio1.8及以下版本，根据版本替换proxyVersion字段。
- 如果您使用的Istio1.9及以上版本，根据版本替换proxyVersion字段，并且替换上述EnvoyFilter中的*envoy.http_connection_manager*为*envoy.filters.network.http_connection_manager*、*envoy.router*为*envoy.filters.http.router*、*type.googleapis.com/envoy.config.filter.http.lua.v2.Lua*为*type.googleapis.com/envoy.extensions.filters.http.lua.v3.Lua*。

2. 执行以下命令，验证HTTP响应头。

   通过以下Curl命令验证安全HTTP响应头是否添加成功。

```
curl -I http://{入口网关服务的IP地址}/productpage
HTTP/1.1 200 OK
content-type: text/html; charset=utf-8
content-length: 4183
server: istio-envoy
date: Tue, 28 Jan 2020 09:07:01 GMT
x-envoy-upstream-service-time: 17
content-security-policy: frame-ancestors none;
x-frame-options: deny
x-xss-protection: 1; mode=block
x-content-type-options: nosniff
referrer-policy: no-referrer
x-download-options: noopen
x-dns-prefetch-control: off
feature-policy: camera 'none';microphone 'none';geolocation 'none';encrypted-media 'none
';payment 'none';speaker 'none';usb 'none';
```

   可以看到示例应用程序的入口首页响应已经包含了HTTP响应头基准配置中所述的安全相关的HTTP响应头。

5.使用ASM实现基于位置的路由请求

在大规模服务场景下，成千上万个服务运行在不同的地域，这些服务需要相互调用来完成完整的功能。为了确保获得最佳性能，应当将流量路由到最近的服务，使得流量尽可能在同一个区域内，而不是只依赖于Kubernetes默认提供的轮询方式进行负载均衡。基于Istio的阿里云服务网格ASM产品提供了基于位置的路由能力，可以将流量路由到最靠近的容器。这样可以确保服务调用的低延迟，并尽量保持服务调用在同一区域内，降低流量费用。本文介绍如何在ASM内实现基于位置的路由请求，以提高性能并节省成本。

前提条件

- 创建ASM实例。
- 创建一个多可用区的托管版ACK集群，集群使用的节点分别位于同一地域下的不同可用区（本文示例中分别为北京地域下的可用区G和H），详情请参见创建Kubernetes托管版集群。

部署后端示例服务

在ACK集群中分别部署Nginx的两个版本（V1和V2），并通过Nodelabel将版本V1部署到可用区G，将版本V2部署到可用区H，操作步骤如下所示。

1. 在ACK集群中部署Nginx版本V1，对应的YAML如下。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mynginx-configmap-v1
  namespace: backend
data:
  default.conf: |-
    server {
        listen       80;
        server_name  localhost;
        #charset koi8-r;
        #access_log  /var/log/nginx/host.access.log  main;
        location / {
            return 200 'v1\n';
        }
    }
```

2. 通过Nodelabel将版本V1部署到可用区G，对应的YAML如下。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-v1
  namespace: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      version: v1
  template:
    metadata:
      labels:
        app: nginx
        version: v1
    spec:
      containers:
      - image: docker.io/nginx:1.15.9
        imagePullPolicy: IfNotPresent
        name: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: nginx-config
          mountPath: /etc/nginx/conf.d
          readOnly: true
      volumes:
      - name: nginx-config
        configMap:
          name: mynginx-configmap-v1
      nodeSelector:
        failure-domain.beta.kubernetes.io/zone: "cn-beijing-g"
```

3. 在ACK集群中部署Nginx版本V2，对应的YAML如下。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mynginx-configmap-v2
  namespace: backend
data:
  default.conf: |-
    server {
        listen       80;
        server_name  localhost;
        #charset koi8-r;
        #access_log  /var/log/nginx/host.access.log  main;
        location / {
            return 200 'v2\n';
        }
    }
```

4. 通过Nodelabel将版本V2部署到可用区H，对应的YAML如下。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-v2
  namespace: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      version: v2
  template:
    metadata:
      labels:
        app: nginx
        version: v2
    spec:
      containers:
      - image: docker.io/nginx:1.15.9
        imagePullPolicy: IfNotPresent
        name: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: nginx-config
          mountPath: /etc/nginx/conf.d
          readOnly: true
      volumes:
      - name: nginx-config
        configMap:
          name: mynginx-configmap-v2
      nodeSelector:
        failure-domain.beta.kubernetes.io/zone: "cn-beijing-h"
```

5. 部署后端示例应用服务，对应的YAML如下。

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: backend
  labels:
    app: nginx
spec:
  ports:
  - name: http
    port: 8000
    targetPort: 80
  selector:
    app: nginx
```

部署客户端示例服务

在ACK集群中分别部署2个客户端示例服务，并通过Nodelabel将版本V1部署到可用区G，将版本V2部署到可用区H，对应的操作步骤如下。

1. 在ACK集群中部署客户端示例服务，通过Nodelabel将版本V1部署到可用区G，对应的YAML如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sleep-cn-beijing-g
  namespace: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sleep
      version: v1
  template:
    metadata:
      labels:
        app: sleep
        version: v1
    spec:
      containers:
      - name: sleep
        image: tutum/curl
        command: ["/bin/sleep","infinity"]
        imagePullPolicy: IfNotPresent
      nodeSelector:
        failure-domain.beta.kubernetes.io/zone: "cn-beijing-g"
```

2. 在ACK集群中部署客户端示例服务，通过Nodelabel将版本V2部署到可用区H，对应的YAML如下所示。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sleep-cn-beijing-h
  namespace: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sleep
      version: v2
  template:
    metadata:
      labels:
        app: sleep
        version: v2
    spec:
      containers:
      - name: sleep
        image: tutum/curl
        command: ["/bin/sleep","infinity"]
        imagePullPolicy: IfNotPresent
      nodeSelector:
        failure-domain.beta.kubernetes.io/zone: "cn-beijing-h"
```

3. 部署客户端示例服务，对应的YAML如下。

```
apiVersion: v1
kind: Service
metadata:
  name: sleep
  namespace: backend
  labels:
    app: sleep
spec:
  ports:
  - name: http
    port: 80
  selector:
    app: sleep
```

4. 执行如下脚本，在2个Sleep Pod中访问后端示例服务。

```
echo "entering into the 1st container"
export SLEEP_ZONE_1=$(kubectl get pods -lapp=sleep,version=v1 -n backend  -o 'jsonpath={
.items[0].metadata.name}')
for i in {1..20}
do
  kubectl exec -it $SLEEP_ZONE_1 -c sleep -n backend -- sh -c 'curl  http://nginx.backen
d:8000'
done
echo "entering into the 2nd container"
export SLEEP_ZONE_2=$(kubectl get pods -lapp=sleep,version=v2 -n backend  -o 'jsonpath={
.items[0].metadata.name}')
for i in {1..20}
do
  kubectl exec -it $SLEEP_ZONE_2 -c sleep -n backend -- sh -c 'curl  http://nginx.backen
d:8000'
done
```

执行结果

通过执行结果可以看到后端示例服务以循环方式进行负载均衡。

```
entering into the 1st container
v2
v1
v1
v2
v2
v1
v1
v2
v2
v1
v1
v2
v2
v1
v2
v1
v1
v2
v2
v1
entering into the 2nd container
v2
v2
v1
v1
v2
v1
v2
v1
v2
v2
v1
v1
v2
v2
v1
v2
v1
v2
v1
v1
```

本地优先负载均衡

本地优先的负载均衡算法会优先访问同一可用区的Pod，并在本可用区Pod不可用时，调用其他可用区的Pod。

为了实现本地优先的负载均衡，您需要具有虚拟服务和带有异常策略的目标规则。离群策略检查Pod是否健康，并做出路由决策。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。

4. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。
5. 按照以下步骤定义虚拟服务，然后单击**创建**。
   i. 选择相应的**命名空间**。本文以选择backend命名空间为例。
   ii. 在文本框中，定义Istio虚拟服务，YAML定义如下所示。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: nginx
  namespace: backend
spec:
  hosts:
    - nginx
  http:
  - route:
    - destination:
        host: nginx
```

6. 在网格详情页面左侧导航栏选择**流量管理 > 目标规则**，然后在右侧页面单击**使用YAML创建**。
7. 按照以下步骤定义目标规则，然后单击**创建**。
   i. 选择相应的**命名空间**。本文以选择backend命名空间为例。
   ii. 在文本框中，定义Istio目标规则，YAML定义如下所示。

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: nginx
  namespace: backend
spec:
  host: nginx
  trafficPolicy:
    outlierDetection:
      consecutiveErrors: 7
      interval: 30s
      baseEjectionTime: 30s
```

8. 执行如下脚本，在2个Sleep Pod中再次访问后端示例服务。

```
echo "entering into the 1st container"
export SLEEP_ZONE_1=$(kubectl get pods -lapp=sleep,version=v1 -n backend  -o 'jsonpath={
.items[0].metadata.name}')
for i in {1..20}
do
  kubectl exec -it $SLEEP_ZONE_1 -c sleep -n backend -- sh -c 'curl  http://nginx.backen
d:8000'
done
echo "entering into the 2nd container"
export SLEEP_ZONE_2=$(kubectl get pods -lapp=sleep,version=v2 -n backend  -o 'jsonpath={
.items[0].metadata.name}')
for i in {1..20}
do
  kubectl exec -it $SLEEP_ZONE_2 -c sleep -n backend -- sh -c 'curl  http://nginx.backen
d:8000'
done
```

执行结果

通过执行结果可以看到以本地优先方式进行负载均衡，执行结果如下。

```
entering into the 1st container
v1
v1
v1
v1
v1
v1
v1
v2
v1
v1
v2
v1
v1
v1
v1
v1
v1
v1
v1
v1
entering into the 2nd container
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
```

局部加权的负载均衡

大多数情况下使用本地优先的负载均衡可以满足平衡的要求，但是有时候您可能需要将流量分成多个区域，如果所有请求都来自单个区域，则容易使一个区域超载。此时可以使用局部加权的负载均衡。

譬如设置如下的加权规则：

- 将从cn-beijing/cn-beijing-g的80%流量路由到cn-beijing/cn-beijing-g，20%的流量路由到cn-beijing/cn-beijing-h。
- 将从cn-beijing/cn-beijing-h的20%流量路由到cn-beijing/cn-beijing-g，80%的流量路由到cn-beijing/cn-beijing-h。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。
5. 按照以下步骤定义虚拟服务，然后单击**创建**。
   i. 选择相应的**命名空间**。本文以选择backend命名空间为例。
   ii. 在文本框中，定义Istio虚拟服务，YAML定义如下所示。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: nginx
  namespace: backend
spec:
  hosts:
    - nginx
  http:
  - route:
    - destination:
        host: nginx
```

6. 在网格详情页面左侧导航栏选择**流量管理 > 目标规则**，然后在右侧页面单击**使用YAML创建**。
7. 按照以下步骤定义目标规则，然后单击**创建**。
   i. 选择相应的**命名空间**。本文以选择backend命名空间为例。
   ii. 在文本框中，定义Istio目标规则，YAML定义如下所示。

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: nginx
  namespace: backend
spec:
  host: nginx
  trafficPolicy:
    outlierDetection:
      consecutiveErrors: 7
      interval: 30s
      baseEjectionTime: 30s
    loadBalancer:
      localityLbSetting:
        enabled: true
        distribute:
        - from: cn-beijing/cn-beijing-g/*
          to:
            "cn-beijing/cn-beijing-g/*": 80
            "cn-beijing/cn-beijing-h/*": 20
        - from: cn-beijing/cn-beijing-h/*
          to:
            "cn-beijing/cn-beijing-g/*": 20
            "cn-beijing/cn-beijing-h/*": 80
```

8. 执行如下脚本，在2个Sleep Pod中再次访问后端示例服务。

```
echo "entering into the 1st container"
export SLEEP_ZONE_1=$(kubectl get pods -lapp=sleep,version=v1 -n backend  -o 'jsonpath={
.items[0].metadata.name}')
for i in {1..20}
do
  kubectl exec -it $SLEEP_ZONE_1 -c sleep -n backend -- sh -c 'curl  http://nginx.backen
d:8000'
done
echo "entering into the 2nd container"
export SLEEP_ZONE_2=$(kubectl get pods -lapp=sleep,version=v2 -n backend  -o 'jsonpath={
.items[0].metadata.name}')
for i in {1..20}
do
  kubectl exec -it $SLEEP_ZONE_2 -c sleep -n backend -- sh -c 'curl  http://nginx.backen
d:8000'
done
```

执行结果

通过执行结果可以看到以局部加权比例进行负载均衡，执行结果如下。

```
entering into the 1st container
v1
v1
v1
v1
v2
v1
v1
v2
v1
v1
v1
v2
v1
v1
v1
v1
v1
v2
v1
v1
entering into the 2nd container
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v2
v1
v2
v1
v2
v2
```

6.通过入口网关访问网格内gRPC服务

服务网格ASM的流量管理功能支持通过入口网关访问内部的gRPC服务。本文通过示例介绍如何通过ASM入口网关访问内部gRPC服务，并在gRPC的两个版本之间进行流量切换。

前提条件

- 已创建ASM实例。具体操作，请参见创建ASM实例。
- 已创建ACK集群。具体操作，请参见创建Kubernetes托管版集群。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。
- 已部署入口网关服务。具体操作，请参见添加入口网关服务。
- 已部署应用到ASM实例的集群中。具体操作，请参见部署应用到ASM实例。
- 已创建ASM实例，且版本为专业版、企业版或旗舰版。具体操作，请参见创建ASM实例。

步骤一：部署示例应用

部署名为istio-grpc-server-v1和istio-grpc-server-v2的示例应用。

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，选择**工作负载 > 无状态**。
5. 在**无状态**页面**命名空间**下拉列表中选择命名空间，单击**使用YAML创建资源**。

   > **说明** 当前选中的命名空间应当已标注自动注入Sidecar，即包含istio-system=enabled标签。具体操作，请参见升级Sidecar代理。

6. 在创建页面将下面的YAML模版粘贴到模板文本框中，单击**创建**。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: istio-grpc-server-v1
  labels:
    app: istio-grpc-server
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: istio-grpc-server
      version: v1
  template:
    metadata:
      labels:
        app: istio-grpc-server
        version: v1
    spec:
```

```
      containers:
      - args:
        - --address=0.0.0.0:8080
        image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/istio-grpc-server
        imagePullPolicy: Always
        livenessProbe:
          exec:
            command:
            - /bin/grpc_health_probe
            - -addr=:8080
          initialDelaySeconds: 2
        name: istio-grpc-server
        ports:
        - containerPort: 8080
        readinessProbe:
          exec:
            command:
            - /bin/grpc_health_probe
            - -addr=:8080
          initialDelaySeconds: 2
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: istio-grpc-server-v2
  labels:
    app: istio-grpc-server
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: istio-grpc-server
      version: v2
  template:
    metadata:
      labels:
        app: istio-grpc-server
        version: v2
    spec:
      containers:
        - args:
            - --address=0.0.0.0:8080
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/istio-grpc-server
          imagePullPolicy: Always
          livenessProbe:
            exec:
              command:
                - /bin/grpc_health_probe
                - -addr=:8080
            initialDelaySeconds: 2
          name: istio-grpc-server
          ports:
            - containerPort: 8080
          readinessProbe:
```

```
            exec:
              command:
                - /bin/grpc_health_probe
                - -addr=:8080
            initialDelaySeconds: 2
---
apiVersion: v1
kind: Service
metadata:
  name: istio-grpc-server
  labels:
    app: istio-grpc-server
spec:
  ports:
  - name: grpc-backend
    port: 8080
    protocol: TCP
  selector:
    app: istio-grpc-server
  type: ClusterIP
---
```

步骤二：设置服务网格ASM的路由规则

设置服务网格的服务网关、虚拟服务和目标规则，将流量全部指向istio-grpc-server-v1。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 创建网关规则。
   i. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，然后在右侧页面单击**使用YAML创建**。
   ii. 在**创建**页面设置**命名空间**为**default**，将下面的YAML模板粘贴到文本框中，单击**确定**。

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: grpc-gateway
spec:
  selector:
    istio: ingressgateway # use Istio default gateway implementation
  servers:
  - port:
      number: 8080
      name: grpc
      protocol: GRPC
    hosts:
    - "*"
```

5. 创建目标规则。
   i. 在网格详情页面左侧导航栏选择**流量管理 > 目标规则**，然后在右侧页面单击**使用YAML创建**。

ii. 在**创建**页面设置**命名空间**为**default**，将下面的YAML模板粘贴到文本框中，单击**确定**。

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: dr-istio-grpc-server
spec:
  host: istio-grpc-server
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
    - name: v1
      labels:
        version: "v1"
    - name: v2
      labels:
        version: "v2"
```

6. 创建虚拟服务。

i. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。

ii. 在**创建**页面设置**命名空间**为**default**，将下面的YAML模板粘贴到文本框中，单击**确定**。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: grpc-vs
spec:
  hosts:
  - "*"
  gateways:
  - grpc-gateway
  http:
    - match:
        - port: 8080
      route:
        - destination:
            host: istio-grpc-server
            subset: v1
          weight: 100
        - destination:
            host: istio-grpc-server
            subset: v2
          weight: 0
```

步骤三：部署入口网关

在入口网关中，添加端口8080，并指向服务网关的8080端口。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏单击**ASM网关**。

5. 在**ASM网关**页面单击**创建**，在**创建**页面配置参数，其他采用默认值。

参数	配置项
部署集群	选择要部署入口网关的集群。
负载均衡类型	此处指定负载均衡的类型为**公网访问**。
负载均衡	选择负载均衡。 • 使用已有负载均衡：从已有负载均衡列表中选择。 • 新建负载均衡：单击**新建负载均衡**，从下拉列表中选择所需的负载均衡规格。 **说明** 建议您为每个Kubernetes服务分配一个SLB。如果多个Kubernetes服务复用同一个SLB，存在以下风险和限制： ◦ 使用已有的SLB会强制覆盖已有监听，可能会导致您的应用不可访问。 ◦ Kubernetes通过Service创建的SLB不能复用，只能复用您手动在控制台（或调用OpenAPI）创建的SLB。 ◦ 复用同一个SLB的多个Service不能有相同的前端监听端口，否则会造成端口冲突。 ◦ 复用SLB时，监听的名字以及虚拟服务器组的名字被Kubernetes作为唯一标识符。请勿修改监听和虚拟服务器组的名字。 ◦ 不支持跨集群复用SLB。
端口映射	单击**添加端口**，设置**名称**为tcp，**服务端口**和**容器端口**为8080。 **说明** 服务端口指的是整个网格对外暴露的端口，是外部访问使用的端口。而容器端口指的是从服务端口进来的流量所指向的服务网关端口。所以容器端口需要与服务网关端口一致。

6. 单击**确定**。

步骤四：运行gRPC客户端

1. 启动gRPC客户端。

```
docker run -d --name grpc-client registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/istio-grpc-client 365d
```

2. 登录到容器。

```
docker exec -it grpc-client sh
```

3. 访问网格内的gRPC服务。

```
/bin/greeter-client --insecure=true --address=<入口网关的IP地址>:8080 --repeat=100
```

返回以下结果，可以看到所有的请求都指向了istio-grpc-server-v1。

```
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:18:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
```

步骤五：按比例将流量路由到v2

将40%的流量指向istio-grpc-server-v2，其余60%的流量仍然指向istio-grpc-server-v1。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**。
5. 在**虚拟服务**页面单击grpc-vs**操作**列的**YAML**。
6. 在**编辑**对话框中更新以下YAML内容，单击**确定**。

```
....
      route:
        - destination:
            host: istio-grpc-server
            subset: v1
          weight: 60
        - destination:
            host: istio-grpc-server
            subset: v2
          weight: 40
```

7. 登录到容器，执行以下命令，访问网格内的gRPC服务。

```
/bin/greeter-client --insecure=true --address=<入口网关的IP地址>:8080 --repeat=100
```

返回以下结果，可以看到40%的流量指向了istio-grpc-server-v2。

> **说明** 您的测试结果不一定总是100次中有40次指向istio-grpc-server-v2，但从总体比例来看，一定是接近40% 的。

```
2020/09/11 03:34:51 Hello world from istio-grpc-server-v2-665c4cf57d-h74lw
2020/09/11 03:34:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:34:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:34:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:34:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:34:51 Hello world from istio-grpc-server-v2-665c4cf57d-h74lw
2020/09/11 03:34:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:34:51 Hello world from istio-grpc-server-v1-dbdd97cc-n85lw
2020/09/11 03:34:51 Hello world from istio-grpc-server-v2-665c4cf57d-h74lw
2020/09/11 03:34:51 Hello world from istio-grpc-server-v2-665c4cf57d-h74lw
```

7.通过入口网关访问网格内WebSocket服务

WebSocket是一种网络传输协议，可在单个TCP连接上进行全双工通信，位于OSI模型的应用层。WebSocket允许服务端主动向客户端推送数据。遵守WebSocket协议的服务即为WebSocket服务。本文通过示例介绍如何通过ASM入口网关访问网格内的WebSocket服务。

前提条件

- 已创建ASM实例。具体操作，请参见创建ASM实例。
- 已创建ACK集群。具体操作，请参见创建Kubernetes托管版集群。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。
- 已部署入口网关服务。具体操作，请参见添加入口网关服务。
- 已部署应用到ASM实例的集群中。具体操作，请参见部署应用到ASM实例。
- 已创建ASM实例，且版本为专业版、企业版或旗舰版。具体操作，请参见创建ASM实例。

步骤一：部署示例应用

1. 通过kubectl连接集群。具体操作，请参见通过kubectl工具连接集群。
2. 使用以下内容，创建名为*tornado*的YAML文件。

```
apiVersion: v1
kind: Service
metadata:
  name: tornado
  labels:
    app: tornado
    service: tornado
spec:
  ports:
  - port: 8888
    name: http
  selector:
    app: tornado
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tornado
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tornado
      version: v1
  template:
    metadata:
      labels:
        app: tornado
        version: v1
    spec:
      containers:
      - name: tornado
        image: hiroakis/tornado-websocket-example
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8888
---
```

3. 执行以下命令，创建tornado应用。

```
kubectl apply -f tornado.yaml
```

步骤二：设置路由规则

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 创建网关规则。
   i. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，然后在右侧页面单击**使用YAML创建**。

ii. 设置**命名空间**为**default**，将以下内容复制到文本框中，单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: tornado-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

设置`number`为`80`，使WebSocket服务通过80端口接收传入或传出的HTTP或TCP连接。

5. 创建虚拟服务。
   i. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。
   ii. 设置**命名空间**为**default**，将以下内容复制到文本框中，单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: tornado
spec:
  hosts:
  - "*"
  gateways:
  - tornado-gateway
  http:
  - match:
    - uri:
        prefix: /
    route:
    - destination:
        host: tornado
      weight: 100
```

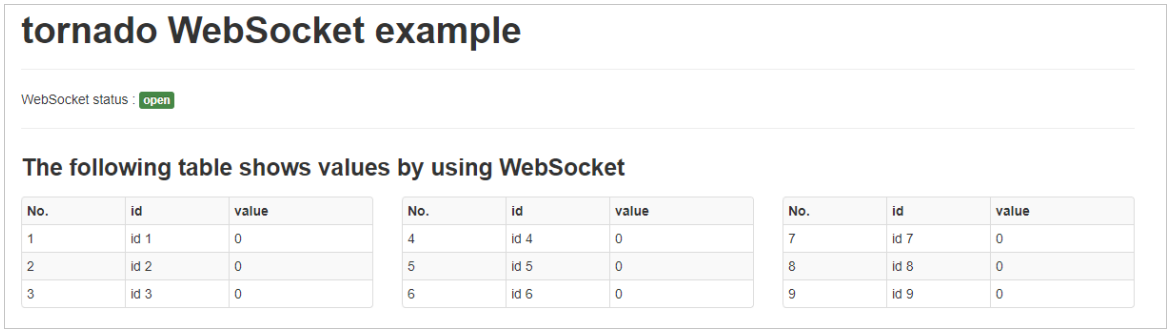
设置`hosts`为`*`，表示任意请求都可以访问WebSocket服务。

步骤三：获取入口网关的地址

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，选择**网络 > 服务**。
5. 在**服务**页面顶部设置**命名空间**为**istio-system**，查看istio-ingressgateway**外部端点**列下端口为80的IP地址。

步骤四：验证通过入口网关访问WebSocket服务

1. 在四个不同类型的浏览器中输入*http://<入口网关地址>*。



2. 分别执行以下命令，请求WebSocket服务。

```
curl "http://<入口网关地址>/api?id=8&value=300"
```

```
curl "http://<入口网关地址>/api?id=5&value=600"
```

```
curl "http://<入口网关地址>/api?id=1&value=200"
```

```
curl "http://<入口网关地址>/api?id=3&value=290"
```

可以看到，四个浏览器的WebSocket服务页面数据同时被更新，且页面显示结果一致。

8.在ASM中使用WebSocket协议访问服务

WebSocket是基于RFC 6455标准，用于客户端和服务器端之间进行双向通信的协议。Istio Sidecar Proxy提供了开箱即用方式使用WebSocket协议，便于您使用WebSocket协议实现服务访问。本文介绍如何在ASM中使用WebSocket over HTTP/1.1和HTTP/2协议访问服务。

前提条件

- 已创建一个ASM实例，并已将ACK集群添加到ASM实例中。具体操作，请参见创建ASM实例和添加集群到ASM实例。
- 已为命名空间注入Sidecar，具体操作，请参见多种方式灵活开启自动注入。本文以default命名空间为例。

背景信息

WebSocket与HTTP协议不同，WebSocket协议使用HTTP Upgrade标头来建立各方之间的连接。Istio无法识别WebSocket协议，但Istio Sidecar Proxy提供了开箱即用方式支持WebSocket协议。关于Istio支持WebSocket协议详细介绍，请参见HTTP upgrades、HTTP connection manager和Protocol Selection。

您可以在ASM中使用WebSocket over HTTP/1.1和HTTP/2协议访问服务，区别如下：

- WebSocket over HTTP/1.1协议：一次请求和响应，建立一个连接，用完关闭连接。
- WebSocket over HTTP/2协议：多个请求可同时在一个连接上并行执行，某个请求任务耗时严重，不会影响到其它连接的正常执行。

部署WebSocket客户端和服务端

1. 通过kubectl工具连接集群。
2. 部署WebSocket服务端。

   本文使用WebSocket社区提供的Python库中WebSocket服务端为例。如果您想修改WebSocket服务端的部署配置，请参见容器化应用程序。

   i. 使用以下内容，创建*websockets-server.yaml*。

```
apiVersion: v1
kind: Service
metadata:
  name: websockets-server
  labels:
    app: websockets-server
spec:
  type: ClusterIP
  ports:
    - port: 8080
      targetPort: 80
      name: http-websocket
  selector:
    app: websockets-server
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: websockets-server
  labels:
    app: websockets-server
spec:
  selector:
    matchLabels:
      app: websockets-server
  template:
    metadata:
      labels:
        app: websockets-server
    spec:
      containers:
      - name: websockets-test
        image: registry.cn-beijing.aliyuncs.com/aliacs-app-catalog/istio-websockets-
test:1.0
        ports:
        - containerPort: 80
```

ii. 在default命名空间下部署WebSocket服务端。

```
kubectl apply -f websockets-server.yaml -n default
```

3. 部署WebSocket客户端。

本文使用以下Dockerfile创建WebSocket客户端镜像，*websockets-client.yaml*文件中使用已经构建完成的WebSocket客户端镜像，您可以直接部署到集群中。

```
FROM python:3.9-alpine
RUN pip3 install websockets
```

i. 使用以下内容，创建*websockets-client.yaml*。

```
apiVersion: v1
kind: Service
metadata:
  name: websockets-client
  labels:
    app: websockets-client
spec:
  type: ClusterIP
  ports:
    - port: 8080
      targetPort: 80
      name: http-websockets-client
  selector:
    app: websockets-client
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: websockets-client-sleep
  labels:
    app: websockets-client
spec:
  selector:
    matchLabels:
      app: websockets-client
  template:
    metadata:
      labels:
        app: websockets-client
    spec:
      containers:
      - name: websockets-client
        image: registry.cn-beijing.aliyuncs.com/aliacs-app-catalog/istio-websockets-client-test:1.0
        command: ["sleep", "14d"]
```

ii. 在default命名空间下部署WebSocket客户端。

```
kubectl apply -f websockets-client.yaml -n default
```

使用WebSocket over HTTP/1.1协议

使用WebSocket客户端向服务端发送了多个请求，Sidecar Proxy日志只会包含一个WebSocket连接的访问日志条目。 Envoy将WebSocket连接视为TCP字节流，并且代理只能理解HTTP升级请求或响应，因此Envoy只会在连接关闭后创建该访问日志条目，并且也不会看到每个TCP请求的任何消息。

1. 在客户端的容器组中打开Shell命令行窗口。
   i. 登录容器服务管理控制台。
   ii. 在控制台左侧导航栏中，单击**集群**。
   iii. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
   iv. 在集群管理页左侧导航栏中，选择**工作负载 > 容器组**。

v. 在**容器组**页面单击websocket-client右侧**操作**列下**终端**，单击**容器：websocket-client**。

vi. 执行以下命令，打开Shell命令行窗口。

```
kubectl exec -it -n <namespace> websockets-client-sleep...  -c websockets-client -- sh
```

2. 执行以下命令，访问WebSocket服务端。

```
python3 -m websockets ws://websockets-server.<namespace>.svc.cluster.local:8080
```

预期输出：

```
Connected to ws://websockets-server.default.svc.cluster.local:8080.
```

输入hello和word，可以看到返回hello和word。

```
> hello
< hello
> world
< world
Connection closed: 1000 (OK).
```

3. 查看Sidecar Proxy日志。
   - 查看WebSocket客户端的Sidecar Proxy日志。
     a. 在**容器组**页面单击WebSocket客户端容器名称。
     b. 单击**日志**页签，设置**容器**为**istio-proxy**。

        可以看到日志中包含使用WebSocket over HTTP/1.1协议记录。

        ```
        {...."upstream_host":"10.208.0.105:80","bytes_sent":23,"protocol":"HTTP/1.1",....
        }
        ```

   - 查看WebSocket服务端的Sidecar Proxy日志。
     a. 在**容器组**页面单击WebSocket服务端容器名称。
     b. 单击**日志**页签，设置**容器**为**istio-proxy**。

        可以看到日志中包含使用WebSocket over HTTP/1.1协议记录。

        ```
        {...."downstream_local_address":"10.208.0.105:80","upstream_local_address":"127.0.**.**:53983","protocol":"HTTP/1.1",....}
        ```

使用WebSocket over HTTP/2协议

低于1.12版本的Istio使用WebSocket over HTTP/2协议，将不能正常连接到WebSocket服务器端，并返回503错误。

如果您已经为低于1.12版本的Istio设置HTTP/2升级，并发生报错，您可以通过在目标规则中定义h2UpgradePolicy为*DO_NOT_UPGRADE*的方式，使低于1.12版本的Istio可以正常使用WebSocket over HTTP/1.1协议。

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  labels:
    provider: asm
  name: websockets-server
spec:
  host: websockets-server
  trafficPolicy:
    connectionPool:
      http:
        h2UpgradePolicy: DO_NOT_UPGRADE
```

Istio 1.12或以上版本使用WebSocket over HTTP/2协议的操作如下：

1. 创建目标规则。
   i. 登录ASM控制台。
   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。
   iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
   iv. 在网格详情页面左侧导航栏选择**流量管理 > 目标规则**，然后在右侧页面单击**使用YAML创建**。
   v. 设置**命名空间**为default，复制以下内容到文本框中，然后单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  labels:
    provider: asm
  name: websockets-server
spec:
  host: websockets-server
  trafficPolicy:
    connectionPool:
      http:
        h2UpgradePolicy: UPGRADE
```

   h2UpgradePolicy：设置为*UPGRADE*，表示为websocket s-server启用HTTP/2协议。

2. 创建EnvoyFilter。

   默认情况下，WebSocket不支持HTTP/2协议，但Envoy却支持WebSocket在HTTP/2流上进行隧道传输，以便在整个部署过程中可以使用统一的HTTP/2网络。您可以通过EnvoyFilter针对目标工作负载设置 `allow_connect: true` ，从而使WebSocket服务端允许HTTP/2协议连接。

i. 创建Envoy过滤器模板。

a. 登录ASM控制台。

b. 在左侧导航栏，选择**服务网格 > 网格管理**。

c. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。

d. 在网格详情页面左侧导航栏选择**插件中心 > 插件市场**。

e. 在**插件市场**页面选择**设置allow_connect为true允许升级的协议连接**，单击**应用此模板**。

f. 在**填充模板参数**配置向导中选择**SIDECAR_INBOUND**，单击**下一步**。

g. 输入模板名称，在页面底部单击**确定**。

ii. 绑定工作负载。

a. 在网格详情页面左侧导航栏选择**插件中心 > Envoy过滤器模板**。

b. 在**Envoy过滤器模板**页面下单击目标模板右侧**操作**列下的**编辑模板**。

c. 单击**绑定工作负载**页签，单击**选定工作负载绑定**。

d. 在**选定工作负载绑定**对话框设置**命名空间**为default，**工作负载**类型为**Deployment**，在**未绑定区域**单击websockets-server**操作**列下的**绑定**，单击**确定**。

绑定工作负载后，ASM会自动创建Envoy过滤器，Envoy过滤器的YAML内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: h2-upgrade-wss
  labels:
    asm-system: 'true'
    provider: asm
spec:
  workloadSelector:
    labels:
      app: websockets-server
  configPatches:
  - applyTo: NETWORK_FILTER
    match:
      context: SIDECAR_INBOUND
      proxy:
        proxyVersion: '^1\.*.*'
      listener:
        filterChain:
          filter:
            name: "envoy.filters.network.http_connection_manager"
    patch:
      operation: MERGE
      value:
        typed_config:
          '@type': type.googleapis.com/envoy.extensions.filters.network.http_con
nection_manager.v3.HttpConnectionManager
          http2_protocol_options:
            allow_connect: true
```

3. 执行以下命令，访问WebSocket服务端。

```
python3 -m websockets ws://websockets-server.<namespace>.svc.cluster.local:8080
```

预期输出：

```
Connected to ws://websockets-server.default.svc.cluster.local:8080.
```

输入hello和word，可以看到返回hello和word。

```
> hello
< hello
> world
< world
Connection closed: 1000 (OK).
```

4. 查看Sidecar Proxy日志。
   - 查看WebSocket客户端的Sidecar Proxy日志。
     a. 在**容器组**页面单击WebSocket客户端容器名称。
     b. 单击**日志**页签，设置**容器**为**istio-proxy**。

        可以看到日志中包含使用WebSocket over HTTP/1.1协议记录，说明客户端发起请求使用的是HTTP/1.1协议。

        ```
        {...."authority":"websockets-server.default.svc.cluster.local:8080","upstream_service_time":null,"protocol":"HTTP/1.1",....}
        ```

   - 查看WebSocket服务端的Sidecar Proxy日志。
     a. 在**容器组**页面单击WebSocket服务端容器名称。
     b. 单击**日志**页签，设置**容器**为**istio-proxy**。

        可以看到日志中包含使用WebSocket over HTTP/2协议记录，说明WebSocket服务端的接收到的协议已经升级为HTTP/2。

        ```
        {...."method":"GET","upstream_local_address":"127.0.**.**:34477","protocol":"HTTP/2",....}
        ```

相关文档

- 通过入口网关访问网格内WebSocket服务

9.在ASM中使用VirtualService的Delegate能力

通过VirtualService定义路由规则时，如果您的服务包含很多微服务，或者您的服务包含很多路由规则，会存在维护一个庞大的VirtualService的问题。ASM引入了Delegate机制，将服务的路由规则进行拆分，降低路由规则变更带来的风险。本文以Bookinfo服务为例，演示如何使用多个VirtualService定义Bookinfo服务的路由规则。

前提条件

- 已创建ASM实例。具体操作，请参见创建ASM实例。

  > **说明** ASM实例必须为v1.8.6.4-g814070fe-aliyun或者以上版本。

- 已创建ACK集群。具体操作，请参见创建Kubernetes托管版集群。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。
- 已部署入口网关服务。具体操作，请参见添加入口网关服务。
- 已部署应用到ASM实例的集群中。具体操作，请参见部署应用到ASM实例。

背景信息

在服务网格ASM中，通过VirtualService定义路由规则，控制流量路由到服务上的各种行为 。实际使用过程中，会遇到维护一个庞大的VirtualService的问题。服务的路由升级都要修改这个VirtualService规则，同时升级经常会导致服务路由更新冲突，路由配置冗余和耦合。任何的规则配置错误，都会影响数据平面集群下的服务，甚至影响所有的服务访问。

ASM通过扩展VirtualService引入了Delegate机制，使服务的路由规则无需耦合在一个VirtualService中。您可以将VirtualService拆分为主VirtualService和子VirtualService。主VirtualService定义了服务的总规则，子VirtualService定义了服务的详细路由规则。主VirtualService由管理员统一维护，子VirtualService由服务维护者进行维护，可以大大降低服务路由规则变更带来的风险，提高服务独立部署和升级的效率。

注意事项

- ASM不支持对Delegate进行嵌套，只允许在主VirtualService设置Delegate参数，例如主VirtualService和子VirtualService都设置了Delegate参数，则该主VirtualService和子VirtualService将不会生效。
- 子VirtualService的HTTPMatchRequest为主VirtualService的子集，否则会发生冲突，HTTPRoute将不会生效。
- 当主VirtualService的Route和Redirect为空时，才可以指定Delegate。子VirtualService的Hosts必须为空。子VirtualService的路由规则会和主VirtualService的路由规则合并。

步骤一：设置网关规则

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，然后在右侧页面单击**使用YAML创建**。
5. 选择**命名空间**，并将以下内容复制到文本框中，单击**创建**。本文以default命名空间为例。

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

设置 `number` 为 `80` ，使Bookinfo服务通过80端口接收传入或传出的HTTP连接。

步骤二：设置主虚拟服务

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。
5. 选择**命名空间**，并将以下内容复制到文本框中，单击**创建**。本文以default命名空间为例。

   以下内容创建了vs-1和vs-2的 `delegate` ，其中vs-1要求请求中必须包含 `/log` 才能访问Bookinfo服务，vs-2要求请求中必须包含 `/` 才能访问Bookinfo服务。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: bookinfo
  namespace: default
spec:
  gateways:
    - bookinfo-gateway
  hosts:
    - '*'
  http:
    - delegate:
        name: vs-1
        namespace: ns1
      match:
        - uri:
            prefix: /log
    - delegate:
        name: vs-2
        namespace: ns1
      match:
        - uri:
            prefix: /
```

delegate下的参数解释：

- name：delegate的名称。
- namespace：delegate的命名空间。

步骤三：设置子虚拟服务

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。
5. 选择**命名空间**，并将以下内容复制到文本框中，单击**创建**。本文以ns1命名空间为例。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: vs-1
  namespace: ns1
spec:
  http:
    - match:
        - uri:
            exact: /login
        - uri:
            exact: /logout
      route:
        - destination:
            host: productpage.default.svc.cluster.local
            port:
              number: 9080
```

   - metadata：与主VirtualService的delegate参数保持一致，用于绑定delegate参数。本文与vs-1的delegate参数一致，表示绑定vs-1的delegate参数。
   - match：设置请求的过滤条件。本文设置 `uri` 为 `exact: /login` 和 `exact: /logout` ，表示可以登录和登出Bookinfo服务。
6. 重复执行上述步骤，选择**命名空间**，并将以下内容复制到文本框中，单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: vs-2
  namespace: ns1
spec:
  http:
    - match:
        - uri:
            exact: /productpage
        - uri:
            prefix: /static
        - uri:
            prefix: /api/v1/products
      route:
        - destination:
            host: productpage.default.svc.cluster.local
            port:
              number: 9080
```

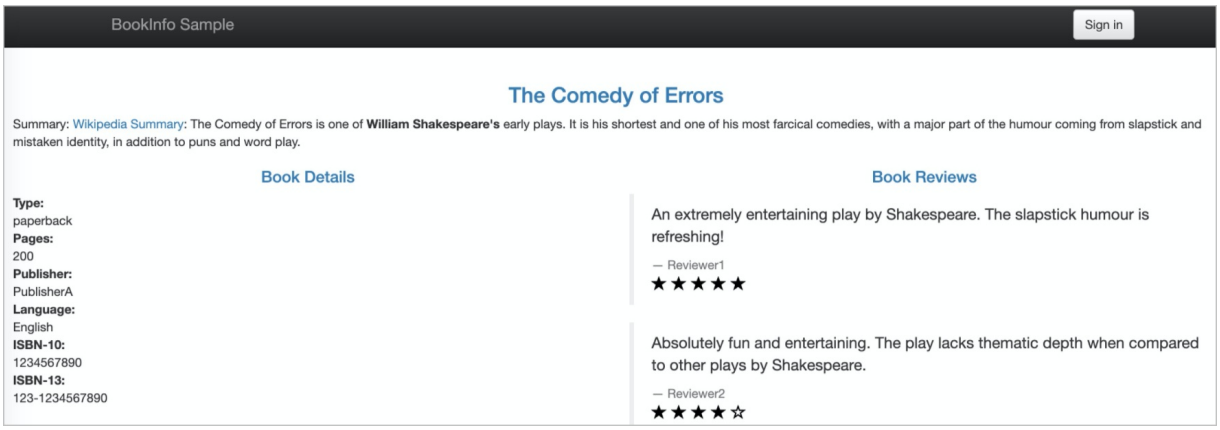
- metadata：与主VirtualService的delegate参数保持一致，用于绑定delegate参数。本文与vs-2的delegate参数一致，表示绑定vs-2的delegate。
- match：设置请求的过滤条件。本文设置请求中必须包含/productpage、/static、/api/v1/products参数。

步骤四：获取ASM网关地址

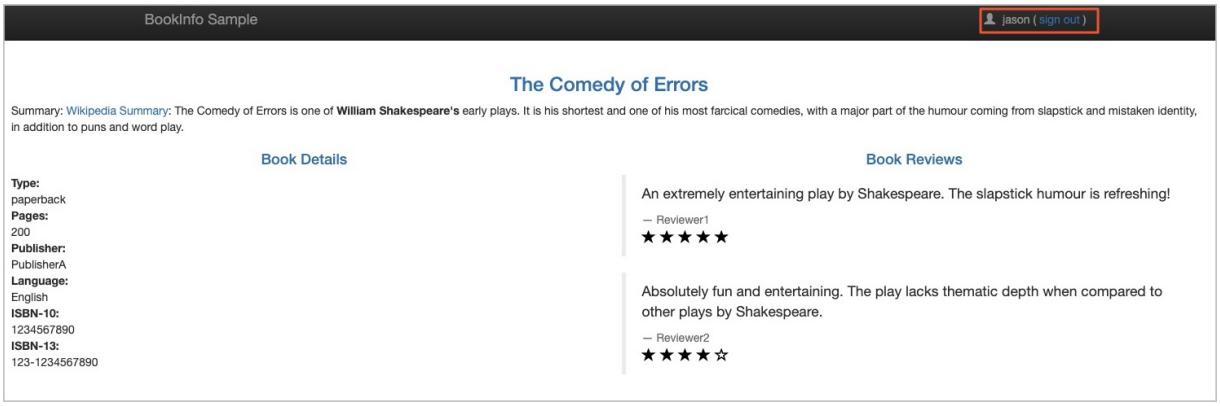
1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，选择**网络 > 服务**。
5. 在**服务**页面顶部设置**命名空间**为istio-system，查看istio-ingressgateway**外部端点**列下端口为80的IP地址。

结果验证

在浏览器中输入*http://<ASM网关地址>/productpage*，显示以下页面。说明请求中包含 `/productpage` 参数访问Bookinfo服务成功，即vs-2的VirtualService设置成功。



单击右上角的Sign in，在弹出对话框输入账号密码，登录Bookinfo服务。显示如下页面，说明登录Bookinfo服务成功，表示vs-1的VirtualService设置成功。



10.使用ASM本地限流功能

在大促等场景下，瞬间洪峰流量会使系统超出最大负载，调用大量堆积，导致整个调用链路卡死。ASM提供了本地限流功能，支持对网关和服务进行流量限制，达到保护系统的目的。本文介绍如何使用ASM本地限流功能。

前提条件

- 已创建ASM实例，且ASM实例要符合以下要求：
  - 如果您使用的是ASM商业版（专业版），要求ASM商业版（专业版）为v1.11.5.30或以上版本。关于升级ASM实例的具体操作，请参见升级ASM实例。
  - 如果您使用的是ASM标准版，ASM标准版仅支持Istio原生方式配置本地限流功能，且要求ASM标准版为v1.9或以上版本。不同Istio版本需参考相应版本文档，关于最新的Istio版本配置本地限流功能的具体操作，请参见Enabling Rate Limits using Envoy。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。
- 已部署入口网关服务。具体操作，请参见添加入口网关服务。
- 创建Bookinfo服务和Nginx。具体操作，请参见部署应用到ASM实例。本文将Bookinfo部署在default命名空间，将Nginx部署在foo命名空间。
- 使用以下内容创建ASM网关。具体操作，请参见添加入口网关服务。本文将ASM网关部署在istio-system命名空间。

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: bookinfo-gateway
  namespace: default
spec:
  selector:
    istio: ingressgateway
  servers:
  - hosts:
    - bf2.example.com
    port:
      name: http
      number: 80
      protocol: http
```

- 使用以下内容创建虚拟服务。具体操作，请参见管理虚拟服务。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: bookinfo
  namespace: default
spec:
  gateways:
  - bookinfo-gateway
  hosts:
  - bf2.example.com
  http:
  - match:
    - uri:
        exact: /productpage
    - uri:
        prefix: /static
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        prefix: /api/v1/products
    name: productpage-route-name1
    route:
    - destination:
        host: productpage
        port:
          number: 9080
  - match:
    - uri:
        prefix: /nginx
    name: nginx-route-name1
    rewrite:
      uri: /
    route:
    - destination:
        host: nginx.foo.svc.cluster.local
        port:
          number: 80
```

- 安装流量加压工具。具体操作，请参见hey。

ASMLocalRateLimiter声明式配置介绍

ASM通过CRD ASMLocalRateLimiter实现本地限流的声明式配置。配置Spec说明如下：

字段	类型	说明	是否必须

字段	类型	说明	是否必须
workloadSelector	map<string, string>	通过一个或多个标签，指明限流配置生效的一组特定的Pod或VM。标签搜索的范围限制在资源所在的配置命名空间。更多信息，请参见Workload Selector。	Yes
isGateway	bool	默认为 `false` 。	No
configs	LocalRateLimiterConfig[]	本地限流配置，可配置多条。	Yes

LocalRateLimiterConfig属性字段

字段	类型	说明	是否必须
name	string	单条限流配置的名称。	No
match	RatelimitMatch	匹配条件。	No
limit	LimitConfig	限流阈值配置。	No

RatelimitMatch属性字段

字段	类型	说明	是否必须
vhost	VirtualHostMatch	VirtualHost匹配条件。	No

LimitConfig属性字段

字段	类型	说明	是否必须
fill_interval	Duration	令牌填充时间单位，例如 `seconds: 1` 或者 `nanos: 1000` 。nanos表示纳秒。	No
quota	int	令牌数量，必须为整数，例如：1000。	No

VirtualHostMatch属性字段

字段	类型	说明	是否必须
name	string	匹配的 `VirtulHost` 名称。	No
port	int	匹配的请求端口。	No

字段	类型	说明	是否必须
route	RouteMatch	匹配的请求接口对应的路由名称。	No

RouteMatch属性字段

字段	类型	说明	是否必须
name_match	string	匹配的路由名称，对应 `VirtualService` 下的单条路由名称。	No
header_match	HeaderMatcher[]	匹配服务请求的header，支持配置多个。	No

HeaderMatcher属性字段

字段	类型	说明	是否必须
name	string	header名称。	No
regex_match	string	正则表达式匹配。	No
exact_match	string	精确匹配。	No
prefix_match	string	前缀匹配，以什么开头进行匹配。	No
suffix_match	string	后缀匹配，以什么结尾进行匹配。	No
present_match	bool	如果配置为 `true` ，则不关心该header value的具体取值，只需要header存在即可。	No
invert_match	bool	默认为 `false` ，如果设置为 `true` ，则正则匹配结果取反。	No

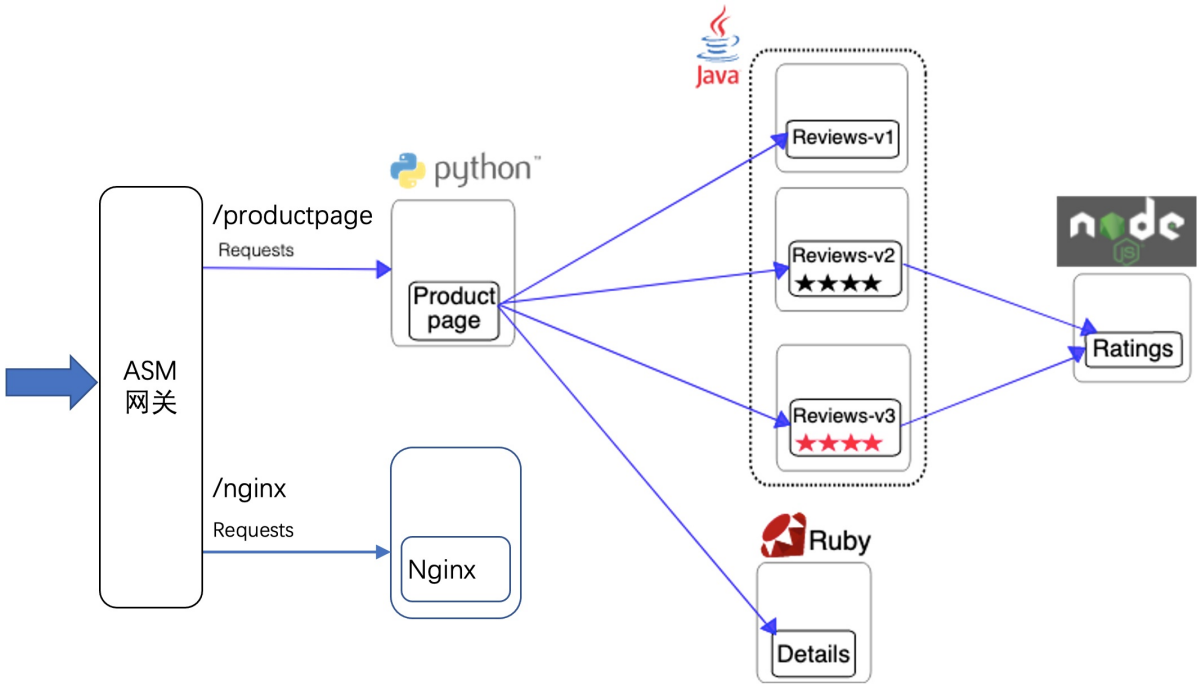
适用对象

ASM本地限流功能适用于ASM网关和应用服务（注入了Sidecar）。

> **说明** 场景示例涉及的配置文件，您可以通过此文件下载。

场景示例说明

本文以Bookinfo和Nginx为例介绍网关和服务限流的具体使用场景。Nginx将单独部署在foo命名空间，用于验证限流的开启范围。



网关限流

对网关进行限流，从流量入口处进行限流，防止下游服务被压垮。

使用kubectl连接ASM，具体操作，请参见通过kubectl连接ASM实例。然后使用以下内容，创建ASMLocalRateLimiter。

> **注意** 以下配置中的 `limite.quota` 只针对单个网关实例生效，若网关有n个实例，test1该路由对应的后端服务限流阈值则为n * quota，若调整了后端服务实例个数，需要对应调整限流阈值。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMLocalRateLimiter
metadata:
  name: for-api-test
  namespace: default
spec:
  workloadSelector:
    labels:
      app: istio-ingressgateway
  isGateway: true
  configs:
    - match:
        vhost:
          name: "www.example1.com"    ## 如果gateway中配置了多个host域名，填最后一个即可。
          port: 80
          route:
            name_match: "test1"    ##VirtualService路由配置中对应route的name,若VirtualService路由配置下没有对应name的路由，则不生效。
      limit:
         fill_interval:
            seconds: 1
         quota: 10
    - match:
        vhost:
          name: "www.example2.com"
          port: 80
          route:
            name_match: "test1"
      limit:
         fill_interval:
            seconds: 1
         quota: 100
```

- workloadSelector：用于匹配生效的Pod或虚拟机。本文为了使ASMLocalRateLimiter作用于网关，设置为*istio-ingressgateway*。
- isGateway：是否作用于网关，本示例设置为*true*。
- fill_interval下的seconds：令牌填充时间。
- quota：令牌发放个数。

  本文设置seconds为*1*，quota为*100*，表示1s内发放100个令牌，即网关1s内最多处理100个请求。

网关限流场景示例

场景一：对单个接口配置限流规则

对`bf2.example.com:80`这个VirtualHost下的`productpage-route-name1`路由设定限流配置，使得访问`productpage-route-name1`路由下的*/productpage*、*/static*、*/login*、*/logout*接口都将受到流量限制。

1. 使用kubectl连接ASM，具体操作，请参见通过kubectl连接ASM实例。
2. 创建ASMLocalRateLimiter。

i. 使用以下内容，创建*asmlocalratelimiter-test-gw.yaml*。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMLocalRateLimiter
metadata:
  name: ingressgateway
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      app: istio-ingressgateway
  isGateway: true
  configs:
  - limit:
      fill_interval:
        seconds: 1
      quota: 10
    match:
      vhost:
        name: bf2.example.com
        port: 80
        route:
          name_match: productpage-route-name1  ## 该名称需要和VirtualService下的route name一致。
```

ii. 执行以下命令，创建ASMLocalRateLimiter。

```
kubectl apply -f asmlocalratelimiter-test-gw.yaml
```

3. 在hey工具中执行以下命令，持续产生压力流量。

```
hey -host bf2.example.com -c 10 -n 100000 http://<ASM网关IP>/productpage
```

```
hey -host bf2.example.com -c 10 -n 100000 http://<ASM网关IP>/nginx
```

4. 执行以下命令，访问Bookinfo服务的*/productpage*接口。

```
curl -H 'host: bf2.example.com'  http://<ASM网关IP>/productpage -v
```

预期输出：

```
< HTTP/1.1 429 Too Many Requests
< Content-Length: 18
< Content-Type: text/plain
< Date: Thu, 13 Jan 2022 03:03:09 GMT
< Server: istio-envoy
<
local_rate_limited
```

可以看到访问Bookinfo服务受到限流。

5. 执行以下命令，访问Bookinfo服务的*/nginx*接口。

```
curl -H 'host: bf2.example.com'  http://${ASM_GATEWAY_IP}/nginx  -v
```

返回结果中没有429，说明访问未限流。

场景二：对网关域名和端口的全局配置限流规则

对 `bf2.example.com:80` 这个virtualHost设定限流配置，使得访问Bookinfo服务下的*/productpage*、*/static*、*/login*、*/logout*、*/nginx*接口都将收到流量限制。

1. 使用kubectl连接ASM，具体操作，请参见通过kubectl连接ASM实例。
2. 创建ASMLocalRateLimiter。
   i. 使用以下内容，创建*asmlocalratelimiter-test-gw-global.yaml*。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMLocalRateLimiter
metadata:
  name: ingressgateway
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      app: istio-ingressgateway
  isGateway: true
  configs:
    - match:
        vhost:
          name: "bf2.example.com"
          port: 80
      limit:
         fill_interval:
            seconds: 1
         quota: 10
```

   ii. 执行以下命令，创建ASMLocalRateLimiter。

```
kubectl apply -f asmlocalratelimiter-test-gw-global.yaml
```

3. 在hey工具中执行以下命令，持续产生压力流量。

```
hey -host bf2.example.com -c 10 -n 100000 http://${ASM_GATEWAY_IP}/nginx
```

4. 执行以下命令，访问Bookinfo服务的*/nginx*接口。

```
curl -H 'host: bf2.example.com'  http://${ASM_GATEWAY_IP}/nginx -v
```

   可以看到，返回 `HTTP/1.1 429 Too Many Requests` ，说明访问Bookinfo服务的*/nginx*接口受到限流。

场景三：删除限流配置后，访问恢复

1. 使用kubectl连接ASM，具体操作，请参见通过kubectl连接ASM实例。
2. 执行以下命令，删除限流配置文件。

```
kubectl delete -f asmlocalratelimiter-test-gw.yaml
```

```
kubectl delete -f asmlocalratelimiter-test-gw-global.yaml
```

3. 执行以下命令，访问Bookinfo服务的*/nginx*接口。

```
curl -H 'host: bf2.example.com'  http://${ASM_GATEWAY_IP}/nginx -v
```

返回结果中没有429，说明访问未限流。

应用服务限流

对服务进行限流，限制服务一定时间内处理请求的数量。

使用kubectl连接ASM，具体操作，请参见通过kubectl连接ASM实例。然后使用以下内容，创建ASMLocalRateLimiter。

> **说明** 针对应用服务进行配置，若服务工作负载有n个实例，对应整体服务限流阈值为n * quota_per_instance。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMLocalRateLimiter
metadata:
  name: reviews-v3-local-ratelimiter
  namespace: default
spec:
  workloadSelector:
    labels:
      app: reviews
      version: v3
....
```

若服务有多个版本，可以通过 `WorkLoadSelector label` 定义生效的Deployment范围，例如以上表示仅针对reviews的v3版本配置限流。

应用服务限流场景示例

> **说明** 应用服务限流生效前提需要应用服务注入了Sidecar，在操作应用服务限流配置之前，请务必移除网关相关限流配置，避免对应用服务限流配置产生影响。

场景一：对reviews服务配置限流规则

1. 使用kubectl连接ASM，具体操作，请参见通过kubectl连接ASM实例。
2. 创建ASMLocalRateLimiter。

i. 使用以下内容，创建*asmlocalratelimiter-test-reviews.yaml*。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMLocalRateLimiter
metadata:
  name: reviews
  namespace: default
spec:
  workloadSelector:
    labels:
      app: reviews
  configs:
    - match:
        vhost:
          name: "*"
          port: 9080
          route:
            header_match:
            - name: ":path"
              prefix_match: "/"
      limit:
         fill_interval:
            seconds: 1
         quota: 10
```

以上配置是针对reviews服务的9080端口，然后通过 `header_match` 配置匹配条件，匹配以/开头的请求。

ii. 执行以下命令，创建ASMLocalRateLimiter。

```
kubectl apply -f asmlocalratelimiter-test-reviews.yaml
```

3. 在hey工具中执行以下命令，持续产生压力流量。

```
hey -host bf2.example.com -c 10 -n 100000 http://<ASM网关IP>/productpage
```

4. 执行以下命令，查看productpage sidecar 日志。

```
kubectl logs -f productpage-v1-b84f8bfdd-wgxlc  -c istio-proxy
```

预期输出：

```
[2022-01-14T07:56:13.086Z] "GET /reviews/0 HTTP/1.1" 429 - via_upstream - "-" 0 18 1 1 "
-" "hey/0.0.1" "9295da56-9a6b-9476-b662-1cbd61a82898" "reviews:9080" "10.180.0.190:9080"
outbound|9080|v1|reviews.default.svc.cluster.local 10.180.0.196:36702 192.168.195.113:90
80 10.180.0.196:33522 - -
[2022-01-14T07:56:13.091Z] "GET /reviews/0 HTTP/1.1" 429 - via_upstream - "-" 0 18 0 0 "
-" "hey/0.0.1" "9295da56-9a6b-9476-b662-1cbd61a82898" "reviews:9080" "10.180.0.190:9080"
outbound|9080|v1|reviews.default.svc.cluster.local 10.180.0.196:36702 192.168.195.113:90
80 10.180.0.196:33528 - -
[2022-01-14T07:56:13.051Z] "GET /details/0 HTTP/1.1" 200 - via_upstream - "-" 0 178 41 1
"-" "hey/0.0.1" "061d3542-52a7-9511-b217-7fdf9ee9a1dd" "details:9080" "10.180.0.160:9080
" outbound|9080||details.default.svc.cluster.local 10.180.0.196:58724 192.168.127.75:908
0 10.180.0.196:57754 - default
[2022-01-14T07:56:13.095Z] "GET /reviews/0 HTTP/1.1" 429 - via_upstream - "-" 0 18 0 0 "
-" "hey/0.0.1" "061d3542-52a7-9511-b217-7fdf9ee9a1dd" "reviews:9080" "10.180.0.190:9080"
outbound|9080|v1|reviews.default.svc.cluster.local 10.180.0.196:36702 192.168.195.113:90
80 10.180.0.196:33534 - -
```

可以以看到访问 `reviews:9080` 返回结果中包含429，说明访问受到限流。

场景二：对review v3版本配置限流规则

1. 使用kubectl连接ASM，具体操作，请参见通过kubectl连接ASM实例。
2. 创建ASMLocalRateLimiter。
   i. 使用以下内容，创建asmlocalratelimiter-test-reviews-only-v3.yaml。

```yaml
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMLocalRateLimiter
metadata:
  name: reviews
  namespace: default
spec:
  workloadSelector:
    labels:
      app: reviews
      version: v3
  configs:
    - match:
        vhost:
          name: "*"
          port: 9080
          route:
            header_match:
            - name: ":path"
              prefix_match: "/"
      limit:
         fill_interval:
            seconds: 1
         quota: 10
```

以上配置是针对reviews v3服务的9080端口，然后通过 `header_match` 配置匹配条件，匹配/开头的请求。

ii. 执行以下命令，创建ASMLocalRateLimiter。

```
kubectl apply -f asmlocalratelimiter-test-reviews-only-v3.yaml
```

3. 在hey工具中执行以下命令，持续产生压力流量。

```
hey -host bf2.example.com -c 10 -n 100000 http://<ASM网关IP>/productpage
```

4. 执行以下命令，查看reviews-v1 、reviews-v2、reviews-v3对应Pod下Sidecar的日志。

```
kubectl logs -f ${your-reviews-pod-name}  -c istio-proxy
```

可以看到只有reviews-v3下access_log日志对应请求的返回包含429， reviews-v2和reviews-v3对应请求的返回是正常的200。说明只有reviews-v3服务受到限流，reviews-v1和reviews-v2服务未受到限流。

11.使用SLB优雅下线功能避免流量损失

ASM网关在缩容或滚动重启时，会删除网关Pod而导致少量流量损失。启用SLB连接优雅下线功能后，即使删除网关Pod，现有连接在一定时间内仍能正常传输，流量将不会有损失。本文介绍如何使用SLB连接优雅下线功能。

前提条件

- 已创建ASM实例，且版本为企业版或旗舰版。具体操作，请参见创建ASM实例。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。

步骤一：启用SLB优雅下线功能

您可以根据实际情况在新建网关时或为已有网关启用SLB优雅下线功能。

新建网关时启用SLB优雅下线功能

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏单击**ASM网关**，然后在右侧页面单击**创建**。
5. 在**创建**页面选择集群，设置**负载均衡类型**为**公网访问**，在**新建负载均衡**下选择负载均衡规格，设置**网关副本数**为*10*，其他参数采用默认设置。
6. 单击**高级选项**，选中**SLB优雅下线**，输入连接超时时间，单击**创建**。

为已有网关启用SLB优雅下线功能

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏单击**ASM网关**。
5. 在**ASM网关**页面单击目标网关的名称。
6. 在**网关详情**页面**高级选项**区域单击**SLB优雅下线**右侧的图标，选中**SLB优雅下线**，输入连接超时时间，单击**确认**。

步骤二：部署示例应用

1. 通过kubectl工具连接集群。
2. 使用以下内容创建*httpbin.yaml*。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: httpbin
---
apiVersion: v1
kind: Service
metadata:
  name: httpbin
  labels:
    app: httpbin
    service: httpbin
spec:
  ports:
  - name: http
    port: 8000
    targetPort: 80
  selector:
    app: httpbin
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpbin
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpbin
      version: v1
  template:
    metadata:
      labels:
        app: httpbin
        version: v1
    spec:
      serviceAccountName: httpbin
      containers:
      - image: docker.io/kennethreitz/httpbin
        imagePullPolicy: IfNotPresent
        name: httpbin
        ports:
        - containerPort: 80
```

3. 执行以下命令，部署httpbin应用。

```
kubectl apply -f httpbin.yaml -n default
```

步骤三：创建虚拟服务和网关规则

1. 创建虚拟服务。
   i. 登录ASM控制台。
   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。

iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。

iv. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。

v. 选择**命名空间**，选择任意**场景模板**，将以下内容替换**YAML**文本框中内容，然后单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: httpbin
  namespace: default
spec:
  gateways:
    - httpbin-gateway
  hosts:
    - '*'
  http:
    - route:
        - destination:
            host: httpbin
            port:
              number: 8000
```

2. 创建网关规则。

i. 登录ASM控制台。

ii. 在左侧导航栏，选择**服务网格 > 网格管理**。

iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。

iv. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，然后在右侧页面单击**使用YAML创建**。

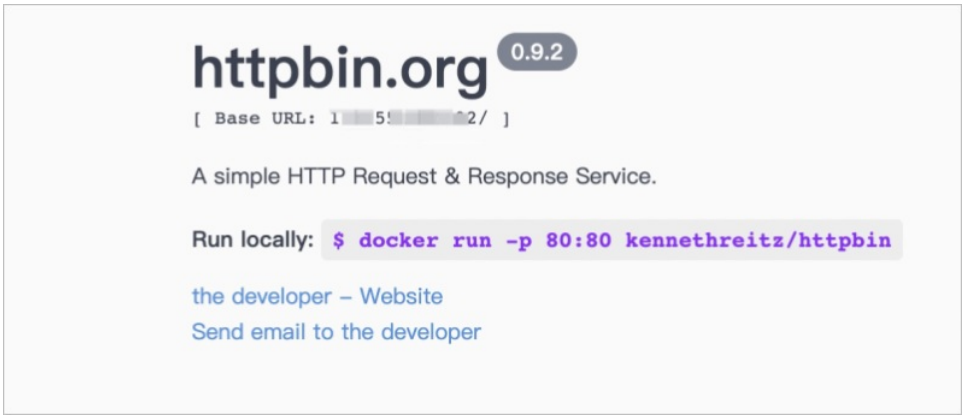
v. 选择**命名空间**，选择任意**场景模板**，将以下内容替换**YAML**文本框中内容，然后单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: httpbin-gateway
  namespace: default
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - '*'
      port:
        name: http
        number: 80
        protocol: HTTP
```

3. 验证路由配置是否成功。

i. 获取ASM网关地址，具体操作，请参见添加入口网关服务。

ii. 在浏览器地址栏中输入*http://<ASM网关地址>*。



显示如上页面，说明路由配置成功。

步骤四：验证启用SLB连接优雅下线功能后的效果

1. 根据您使用的操作系统，安装并下载轻量级压测工具hey的对应版本。具体操作，请参见hey。

   以下为各个操作系统对应的hey版本：

   - Linux 64-bit：hey-release.s3.us-east-2.amazonaws.com/hey_linux_amd64
   - Mac 64-bit：hey-release.s3.us-east-2.amazonaws.com/hey_darwin_amd64
   - Windows 64-bit：hey-release.s3.us-east-2.amazonaws.com/hey_windows_amd64

2. 缩容ASM网关。

   i. 登录ASM控制台。

   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。

   iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。

   iv. 在网格详情页面左侧导航栏单击**ASM网关**。

   v. 在**ASM网关**页面单击目标网关右侧**操作**列下的**查看YAML**。

   vi. 在**编辑**对话框设置replicaCount的参数值为*1*，然后单击**确定**。

```
33    readinessProbe: {}
34    replicaCount: 1
35    resources:
36      limits:
37        cpu: '2'
38        memory: 4G
39      requests:
40        cpu: 200m
41        memory: 256Mi
```

3. 查看启用SLB连接优雅下线功能后的流量损失情况。

   执行以下命令，以200并发，50000请求数请求httpbin应用。

```
hey -c 200 -n 50000  -disable-keepalive   http://{ASM网关IP}/
```

   预期输出：

```
.............
Status code distribution:
  [200] 50000 responses
```

可以看到，50000个请求全部返回200状态码，也就是说50000个请求都访问成功，流量没有损失。

如果在没有启用SLB优雅下线功能情况下，在ASM网关Pod缩容时，以200并发，50000请求数请求httpbin应用。

```
Status code distribution:
  [200] 49747 responses
Error distribution:
  [253] Get "http://47.55.2xx.xx": dial tcp 47.55.2xx.xx:80: connect: connection refused
```

50000个请求中仅49747个请求返回200状态码，也就是说仅49747个请求访问成功，流量有少量的损失。

12.流量打标和标签路由

ASM支持通过TrafficLabel CRD设置流量标签，然后根据流量标签将流量路由到不同的工作负载。本文介绍流量标签和标签路由的功能。

背景信息

流量打标又称作流量染色，指对符合一定流量特征的请求打上具体的染色标记。Istio通过透明拦截的方式劫持了应用流量，基于此技术背景，Sidecar可以对应用流量进行染色标记。

ASM商业版（专业版）扩展了一个新的TrafficLabel CRD，通过该CRD来定义具体的流量染色逻辑，实现命名空间、工作负载及接口级别的流量打标。

注意事项

仅ASM商业版（专业版）v1.10.5.40及以上版本支持流量打标和标签路由功能。

如何对工作负载进行流量打标

方式一：按照命名空间进行流量打标

对命名空间下的所有应用进行流量打标。

1. 通过kubectl连接ASM实例。
2. 使用以下内容，创建example1.yaml。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: TrafficLabel
metadata:
  name: example1
  namespace: default
spec:
  rules:
  - labels:
      - name: userdefinelabel1
        valueFrom:
        - $getContext(x-request-id)
        - $localLabel
    attachTo:
    - opentracing
    # 表示生效的协议，空为都不生效，*为都生效。
    protocols: "*"
  hosts: # 表示生效的服务。
  - "*"
```

参数	描述
namespace	该CRD生效的命名空间。
labels参数下name	标签名称。

参数	描述
labels参数下valueFrom	标签值，取值： 说明 valueFrom的参数值采用自然顺序的优先级，例如以上配置表示优先从*getContext(x-request-id)*获取标签值，当通过*getContext*变量获取不到才会从*localLabel*变量获取标签值。 ○ *getContext(x-request-id)*：从流量的上下文获取标签值。 ○ *localLabel*：从工作负载的业务Pod标签`ASM_TRAFFIC_TAG`获取标签值。
attachTo	取值为*opentracing*，对应HTTP或GRPC协议，意味着会将该流量标识添加到header下。

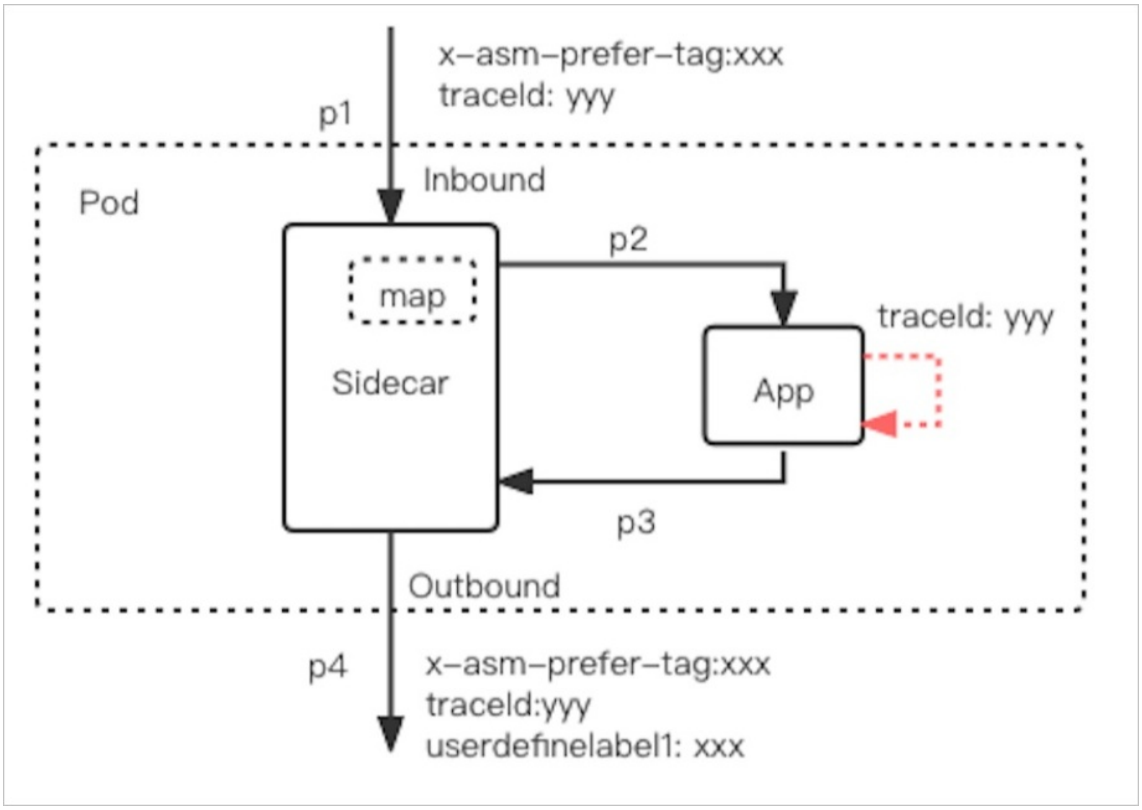
getContext(xxx)和localLabel两个变量详细说明如下：

- getContext(x-request-id)

  getContext(x-request-id)是个函数型变量，标明流量颜色需要从流量的上下文中获取，其中x-request-id为该函数参数，指代Trace ID。不同的Trace系统采用的Trace ID不同。

  说明 getContext仅针对Sidecar生效。

  Sidecar包含入口和出口两种类型的流量，流量打标其实指的是对出口流量进行打标。



ASM商业版（专业版）的Sidecar默认试图通过x-asm-prefer-tag header从入口流量中获取流量tag，并以Trace ID为key记录到上下文 `map<traceId, tag>` 中。当业务容器发起的出口请求时，Sidecar会通过Trace ID查找上下文map。若找到关联tag，则Sidecar会附带x-asm-prefer-tag header（默认）以及TrafficLabel CRD定义的标签名userdefinelabel1。

> **说明** 业务应用采用不同的Trace系统则会有不同Trace ID。更多信息，请参见Tracing。

若不定义getContext函数下的参数，Sidecar默认采用x-request-id为Trace ID。Trace ID需要业务应用进行Header propagation或Context propagation。若Header Propagation逻辑缺失，则出口流量会因为找不到关联的Trace ID key，Sidecar将无法从 `map<traceId,tag>` 找到对应的流量tag。

若业务应用有对接Trace系统，则Trace SDK会帮助业务代码进行Header propagation或Context propagation，例如采用接入ARMS进行trace。若业务有对接ARMS Prometheus，则只需要在TrafficLabel CRD中修改valueFrom为*getContext(x-b3-traceid)*，其中参数*x-b3-traceid*为Zipkin系统的Trace header。

- localLabel

  localLabel是从Sidecar对应的业务POD标签ASM_TRAFFIC_TAG获取流量标识，并对出口流量打标。以下工作负载定义了ASM_TRAFFIC_TAG为*test*。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: productpage-v1
  labels:
    app: productpage
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: productpage
      version: v1
  template:
    metadata:
      annotations:
        sidecar.istio.io/logLevel: debug
      labels:
        app: productpage
        version: v1
        ASM_TRAFFIC_TAG: test
    spec:
      serviceAccountName: bookinfo-productpage
      containers:
      - name: productpage
        image: docker.io/istio/examples-bookinfo-productpage-v1:1.16.2
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
        volumeMounts:
        - name: tmp
          mountPath: /tmp
      volumes:
      - name: tmp
        emptyDir: {}
```

3. 执行以下命令，使CRD生效。

```
kubectl apply -f example1.yaml
```

方式二：按照工作负载进行流量打标

对某个命名空间下某个应用进行流量打标。

1. 通过kubectl连接ASM实例。
2. 使用以下内容，创建example2.yaml。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: TrafficLabel
metadata:
  name: example2
  namespace: workshop
spec:
  workloadSelector:
    labels:
      app: test
  rules:
  - labels:
      - name: userdefinelabel1
        valueFrom:
        - $localLabel
        - $getContext(x-request-id)
    attachTo:
    - opentracing
    protocols: "*"
  hosts:
  - "*"
```

参数	描述
namespace	该CRD生效的命名空间。
labels参数下name	标签名称。
labels参数下valueFrom	标签值，取值： **说明** valueFrom具有优先级，以上配置表示优先从*getContext(x-request-id)*获取标签值，当通过*getContext*变量获取不到才会从localLabel变量获取标签值。 - *getContext(x-request-id)*：从流量的上下文获取标签值。 - *localLabel*：从工作负载的业务Pod标签`ASM_TRAFFIC_TAG`获取标签值。
workloadSelector参数下的app	匹配工作负载的标签，例如本文的CRD只作用于带有test标签的工作负载。
attachTo	取值为*opentracing*，对应HTTP或GRPC协议，意味着会将该流量标识添加到header下。

3. 执行以下命令，使CRD生效。

```
kubectl apply -f example2.yaml
```

方式三：按照接口进行流量打标

对某个命名空间下某个应用接口进行流量打标。

1. 通过kubectl连接ASM实例。
2. 使用以下内容，创建example3.yaml。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: TrafficLabel
metadata:
  name: example3
  namespace: workshop
spec:
  workloadSelector:
    labels:
      app: test
  rules:
  - match:
    - headers:
          end-user: "jason"
      uri:
        prefix: "/test"
      labels:
      - name: userdefinelabel1
        valueFrom:
        - $getContext(x-b3-traceid)
    attachTo:
    - opentracing
    protocols: "*"
  hosts:
    - "*"
```

参数	描述
namespace	该CRD生效的命名空间。
workloadSelector参数下的app	匹配工作负载的标签，例如本文的CRD只作用于带有`app: test`标签的工作负载。
match参数下headers	请求头，本文设置请求头为`jason`**用户**。
match参数下prefix	请求URL，描述特定接口请求，本文设置为带有*/test*的请求URL。
labels参数下name	标签名称。

参数	描述
labels参数下valueFrom	标签值，取值： **说明** valueFrom具有优先级，以上配置表示优先从*getContext(x-request-id)*获取标签值，当通过*getContext*变量获取不到才会从localLabel变量获取标签值。 - *getContext(x-request-id)*：从流量的上下文获取标签值。 - *localLabel*：从工作负载的业务Pod标签`ASM_TRAFFIC_TAG`获取标签值。
attachTo	取值为*opentracing*，对应HTTP或GRPC协议，意味着会将该流量标识添加到header下。

3. 执行以下命令，使CRD生效。

```
kubectl apply -f example3.yaml
```

基于流量标签进行标签路由

通过TrafficLabel定义了流量标签userdefinelabel1，其取值为test1、test2、test3等，您还需要创建DestinationRule和VirtualService，将流量根据标签路由到对应工作负载。

1. 使用以下内容，创建*example4.yaml*。

   使用以下内容将productpage分为test1 、test2 、test3等组。

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: productpage
spec:
  host: productpage
  subsets:
  - name: test1
    labels:
      version: test1
  - name: test2
    labels:
      version: test2
  - name: test3
    labels:
      version: test3
  ...
  - name: testn
    labels:
      version: testn
  - name: base
    labels:
      version: base
```

2. 执行以下命令，创建DestinationRule。

```
kubectl apply -f example4.yaml
```

3. 使用以下内容，创建*example5.yaml*。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: example-app
  namespace: default
spec:
  hosts:
    - example
  http:
  - match:
    - headers:
        userdefinelabel1:
         exact: test1
    route:
    - destination:
        host: example
        subset: test1
  - match:
    - headers:
        userdefinelabel1:
         exact: test2
    route:
    - destination:
        host: example
        subset: test2
  - match:
    - headers:
        userdefinelabel1:
         exact: test3
    route:
    - destination:
        host: example
        subset: test3
  - route:
    - destination:
        host: example
        subset: base
```

headers和subset决定了将流量根据标签路由到对应工作负载，例如本文将带有值为test1，名称为userdefinelabel1的标签的流量路由到test1组别的工作负载。参数解释如下：

参数	描述
headers参数下userdefinelabel1	流量标签名称。
headers参数下exact	流量标签值。
subset	路由到的目标工作负载的组别。

4. 执行以下命令，创建VirtualService。

```
kubectl apply -f example5.yaml
```

其他操作：简化VirtualService变量配置方式

若工作负载的版本有很多个的时候，VirtualService配置会比较臃肿，您可以使用以下内容简化VirtualService变量配置方式，并且对流量降级。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: example-app
  namespace: default
spec:
  hosts:
    - example
  http:
  - route:
    - destination:
        host: example
        subset: $userdefinelabel1
      fallback:
        case: noinstances|noavailabled
        target:
          host: example
          subset: v1
```

其中 `fallback` 配置 `spec` 说明如下：

参数	描述
case	以竖线（|）分隔的字符串，取值： • *noinstances*：缺失实例 • *noavailabled*：后端实例不可用 • *noisolationunit*：后端分组不存在
target	路由的目标服务。本文定义了当缺少实例或者实例不可用时，将流量理由到v1组别的工作负载。

相关文档

- 基于ASM商业版实现全链路灰度发布

13.通过AHAS对应用进行流量控制

服务网格ASM配合应用高可用服务AHAS支持对部署在服务网格内的应用进行流量控制，避免应用被瞬时的流量高峰冲垮，从而保障应用的高可用性。本文介绍如何通过AHAS对网格实例中应用进行流量控制。

前提条件

- 已创建一个ASM商业版（专业版）实例，且实例版本>=1.12.4。具体操作，请参见创建ASM实例。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。
- 已部署入口网关服务。具体操作，请参见添加入口网关服务。
- 已为default命名空间开启自动注入。具体操作，请参见多种方式灵活开启自动注入。
- 已开通AHAS免费版和专业版。具体操作，请参见开通页面和计费概述。

步骤一：在ASM中启用AHAS流控功能

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在**基本信息**页面右上角单击**功能设置**。
5. 在**功能设置更新**面板单击**展开高级选项**，选中**AHAS流控**，然后单击**确定**。

步骤二：部署演示应用Bookinfo

1. 通过kubectl工具连接集群。
2. 在集群中部署Bookinfo应用。
   i. 使用以下内容，创建*bookinfo.yaml*。
   展开此项，查看YAML详细内容

```
# Details service
apiVersion: v1
kind: Service
metadata:
  name: details
  labels:
    app: details
    service: details
spec:
  ports:
  - port: 9080
    name: http
  selector:
    app: details
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bookinfo-details
  labels:
    account: details
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: details-v1
  labels:
    app: details
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: details
      version: v1
  template:
    metadata:
      labels:
        app: details
        version: v1
    spec:
      serviceAccountName: bookinfo-details
      containers:
      - name: details
        image: docker.io/istio/examples-bookinfo-details-v1:1.16.2
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
        securityContext:
          runAsUser: 1000
---
# Ratings service
apiVersion: v1
kind: Service
metadata:
  name: ratings
  labels:
    app: ratings
    service: ratings
spec:
  ports:
  - port: 9080
    name: http
  selector:
    app: ratings
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bookinfo-ratings
  labels:
    account: ratings
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ratings-v1
```

```
  labels:
    app: ratings
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ratings
      version: v1
  template:
    metadata:
      labels:
        app: ratings
        version: v1
    spec:
      serviceAccountName: bookinfo-ratings
      containers:
      - name: ratings
        image: docker.io/istio/examples-bookinfo-ratings-v1:1.16.2
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
        securityContext:
          runAsUser: 1000
---
# Reviews service
apiVersion: v1
kind: Service
metadata:
  name: reviews
  labels:
    app: reviews
    service: reviews
spec:
  ports:
  - port: 9080
    name: http
  selector:
    app: reviews
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bookinfo-reviews
  labels:
    account: reviews
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reviews-v1
  labels:
    app: reviews
    version: v1
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: reviews
      version: v1
  template:
    metadata:
      labels:
        app: reviews
        version: v1
    spec:
      serviceAccountName: bookinfo-reviews
      containers:
      - name: reviews
        image: docker.io/istio/examples-bookinfo-reviews-v1:1.16.2
        imagePullPolicy: IfNotPresent
        env:
        - name: LOG_DIR
          value: "/tmp/logs"
        ports:
        - containerPort: 9080
        volumeMounts:
        - name: tmp
          mountPath: /tmp
        - name: wlp-output
          mountPath: /opt/ibm/wlp/output
        securityContext:
          runAsUser: 1000
      volumes:
      - name: wlp-output
        emptyDir: {}
      - name: tmp
        emptyDir: {}
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reviews-v2
  labels:
    app: reviews
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: reviews
      version: v2
  template:
    metadata:
      labels:
        app: reviews
        version: v2
    spec:
      serviceAccountName: bookinfo-reviews
```

```
      serviceAccountName: bookinfo-reviews
      containers:
      - name: reviews
        image: docker.io/istio/examples-bookinfo-reviews-v2:1.16.2
        imagePullPolicy: IfNotPresent
        env:
        - name: LOG_DIR
          value: "/tmp/logs"
        ports:
        - containerPort: 9080
        volumeMounts:
        - name: tmp
          mountPath: /tmp
        - name: wlp-output
          mountPath: /opt/ibm/wlp/output
        securityContext:
          runAsUser: 1000
      volumes:
      - name: wlp-output
        emptyDir: {}
      - name: tmp
        emptyDir: {}
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reviews-v3
  labels:
    app: reviews
    version: v3
spec:
  replicas: 1
  selector:
    matchLabels:
      app: reviews
      version: v3
  template:
    metadata:
      labels:
        app: reviews
        version: v3
    spec:
      serviceAccountName: bookinfo-reviews
      containers:
      - name: reviews
        image: docker.io/istio/examples-bookinfo-reviews-v3:1.16.2
        imagePullPolicy: IfNotPresent
        env:
        - name: LOG_DIR
          value: "/tmp/logs"
        ports:
        - containerPort: 9080
        volumeMounts:
        - name: tmp
          mountPath: /tmp
```

```
        - name: wlp-output
          mountPath: /opt/ibm/wlp/output
        securityContext:
          runAsUser: 1000
      volumes:
      - name: wlp-output
        emptyDir: {}
      - name: tmp
        emptyDir: {}
---
# Productpage services
apiVersion: v1
kind: Service
metadata:
  name: productpage
  labels:
    app: productpage
    service: productpage
spec:
  ports:
  - port: 9080
    name: http
  selector:
    app: productpage
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bookinfo-productpage
  labels:
    account: productpage
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: productpage-v1
  labels:
    app: productpage
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: productpage
      version: v1
  template:
    metadata:
      labels:
        app: productpage
        version: v1
    spec:
      serviceAccountName: bookinfo-productpage
      containers:
      - name: productpage
```

```
        image: docker.io/istio/examples-bookinfo-productpage-v1:1.16.2
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
        volumeMounts:
        - name: tmp
          mountPath: /tmp
        securityContext:
          runAsUser: 1000
      volumes:
      - name: tmp
        emptyDir: {}
---
```

ii. 执行以下命令，将Bookinfo应用部署到集群中。

> **说明** `DATA_PLANE_KUBECONFIG` 表示ASM实例集群的Kubeconfig保存在本地的路径。

```
kubectl --kubeconfig=${DATA_PLANE_KUBECONFIG} apply -f bookinfo.yaml
```

3. 通过kubectl连接ASM实例。
4. 在ASM中创建网关规则和虚拟服务。
   i. 使用以下内容，创建*bookinfo-gateway.yam*。

   展开此项，查看YAML详细内容

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
  - "*"
  gateways:
  - bookinfo-gateway
  http:
  - match:
    - uri:
        exact: /productpage
    - uri:
        prefix: /static
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        prefix: /api/v1/products
    route:
    - destination:
        host: productpage
        port:
          number: 9080
```

ii. 执行以下命令，创建网关规则和虚拟服务。

```
kubectl --kubeconfig=${ASM_KUBECONFIG} apply -f bookinfo-gateway.yaml
```

步骤三：开启AHAS流控

1. 使用以下内容，创建*envoyfilter-ahas.yaml*。

   展开此项，查看YAML详细内容

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  labels:
    provider: "asm"
    asm-system: "true"
  name: ahas-filter
  namespace: default
spec:
  configPatches:
    - applyTo: HTTP_FILTER
      match:
        proxy:
          proxyVersion: "^1\\.12\\.[4-9]\\-[0-9]+"
          metadata:
            ASM_EDITION: "Pro"
        context: SIDECAR_INBOUND
        listener:
          filterChain:
            filter:
              name: "envoy.filters.network.http_connection_manager"
              subFilter:
                name: "envoy.filters.http.router"
      patch:
        operation: INSERT_BEFORE
        value:
          name: "com.aliyun.ahas"
    - applyTo: HTTP_FILTER
      match:
        proxy:
          proxyVersion: "^1\\.12\\.[4-9]\\-[0-9]+"
          metadata:
            ASM_EDITION: "Pro"
        context: SIDECAR_OUTBOUND
        listener:
          filterChain:
            filter:
              name: "envoy.filters.network.http_connection_manager"
              subFilter:
                name: "envoy.filters.http.router"
      patch:
        operation: INSERT_BEFORE
        value:
          name: "com.aliyun.ahas"
```

2. 执行以下命令，创建EnvoyFilter以开启AHAS流控功能。

```
kubectl --kubeconfig=${ASM_KUBECONFIG} apply -f envoyfilter-ahas.yaml
```

3. 验证开启AHAS流控功能是否成功。
   i. 登录ASM控制台。
   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。

iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。

iv. 在网格详情页面左侧导航栏选择**AHAS流控 > 应用防护**。

在**应用防护**页面可以看到接入的服务，说明开启AHAS流控功能成功。

步骤四：配置限流规则

为了便于触发流控，本示例中添加一个1QPS的流控规则，即1秒内只允许一次访问。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**AHAS流控 > 应用防护**。
5. 在**应用防护**页面单击default_productpage-v1应用卡片。
6. 在左侧导航栏单击**接口详情**。
7. 在**接口详情**页面单击productpage接口卡片右上角+图标。

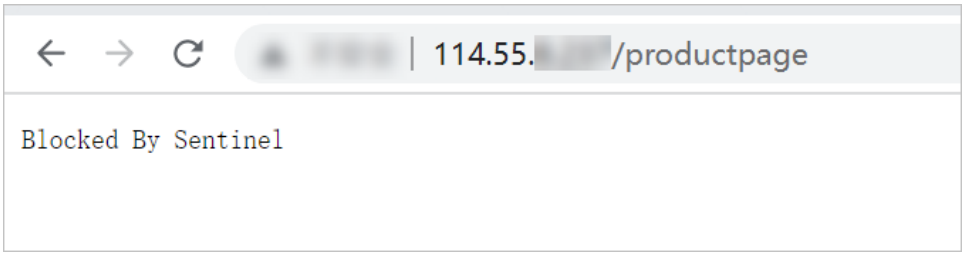
> **说明** 如果该接口已配置限流规则，还需要增加限流规则，您需要单击资源卡片右上角的图标。

8. 在**新建规则**对话框设置**单机QPS阈值**为0.1，单击**下一步**。
9. 设置**关联行为**为**默认行为**，单击**下一步**，然后单击**新增**。

   默认行为表示触发限流规则后返回Blocked by Sentinel的提示文本。您还可以自定义更多的关联行为，具体操作，请参见配置Web行为。

步骤五：触发限流规则

在浏览器中输入*http://{入口网关地址}/productpage*，短时间内连续刷新2次，访问Productpage应用。可以看到Productpage页面返回Blocked By Sentinel，说明限流规则生效。



相关操作

执行以下命令，删除EnvoyFilter，即可删除AHAS限流服务。

```
kubectl --kubeconfig=${ASM_KUBECONFIG} delete envoyfilter <EnvoyFilterName>
```

其中 `EnvoyFilterName` 是步骤2创建的EnvoyFilter名称。

14.使用ASMAdaptiveConcurrency实现自适应并发控制（Beta）

ASMAdaptiveConcurrency能够根据采样的请求数据动态调整允许的并发的数量，当并发数量超过服务可承受范围时，会拒绝请求，达到保护服务的目的。本文介绍如何使用ASMAdaptiveConcurrency实现自适应并发控制。

前提条件

- 已创建高级版或旗舰版ASM实例，且实例为1.12.4.19或以上版本。具体操作，请参见创建ASM实例。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。

背景信息

通常情况下，希望服务在超出负载能力时能拒绝超额的请求，从而防止产生其他连锁反应。您可以使用服务网格的DestinationRule配置基础的熔断能力，但这要求必须给出一个触发熔断的阈值，例如具体的pending requests数量，使得服务网格数据平面在网络访问超出熔断配置时能够拒绝请求。在实际应用场景下，准确地估算服务的负载能力是较为困难的。

ASMAdaptiveConcurrency采用自适应并发控制算法通过定期对比采样Latency和测定的理想Latency之间的差异及一系列计算对并发限制进行动态调整，从而尽可能使得并发限制数量在服务可承受的范围附近，同时拒绝超出该限制的请求（拒绝请求时会返回 `HTTP 503` 及错误信息 `reached concurrency limit` ）。

由于周期性的MinRTT统计进行期间会将连接数限制在较小值（min_concurrency），建议您在启用了AdaptiveConcurrency服务后，同时使用DestinationRule为服务启用重试功能，使得在minRTT计算期间被拒绝的请求能够通过Sidecar的重试功能尽可能地成功返回结果。

步骤一：部署示例应用

部署testserver和gotest应用，设置testserver应用的负载能力为并发处理500个请求，超过并发数量的请求会被排队处理。每个请求的处理时间是1000ms。设置gotest应用的一个副本一次能发起200个请求。

1. 通过kubectl工具连接集群。
2. 部署testserver应用。

i. 使用以下内容，创建*testserver.yaml*文件。

展开查看详细内容

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: testserver
  name: testserver
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: testserver
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: testserver
    spec:
      containers:
      - args:
        - -m
        - "500"
        - -t
        - "1000"
        command:
        - /usr/local/bin/limited-concurrency-http-server
        image: registry.cn-hangzhou.aliyuncs.com/acs/asm-limited-concurrency-http-server:v0.1.1-gee0b08f-aliyun
        imagePullPolicy: IfNotPresent
        name: testserver
        ports:
        - containerPort: 8080
          protocol: TCP
```

您可以使用 `-m` 指定应用能够承受的并发数量， `-t` 指定请求处理时间。

ii. 执行以下命令，部署testserver应用。

```
kubectl apply -f testserver.yaml
```

3. 部署testserver的Service。

i. 使用以下内容，创建*testservice.yaml*文件。

展开查看详细内容

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: testserver
  name: testserver
  namespace: default
spec:
  internalTrafficPolicy: Cluster
  ipFamilies:
    - IPv4
  ipFamilyPolicy: SingleStack
  ports:
    - name: http
      port: 8080
      protocol: TCP
      targetPort: 8080
    - name: metrics
      port: 15020
      protocol: TCP
      targetPort: 15020
  selector:
    app: testserver
  type: ClusterIP
```

ii. 执行以下命令，部署testserver的Service。

```
kubectl apply -f testservice.yaml
```

4. 部署gotest应用。

i. 使用以下内容，创建*gotest.yaml*文件。

展开查看详细内容

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: gotest
  name: gotest
  namespace: default
spec:
  replicas: 0
  selector:
    matchLabels:
      app: gotest
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: gotest
    spec:
      containers:
      - args:
        - -c
        - "200"
        - -n
        - "10000"
        - -u
        - testserver:8080
        command:
        - /root/go-stress-testing-linux
        image: xocoder/go-stress-testing-linux:v0.1
        imagePullPolicy: Always
        name: gotest
        resources:
          limits:
            cpu: 500m
```

ii. 执行以下命令，部署gotest应用。

```
kubectl apply -f gotest.yaml
```

步骤二：创建ASMAdaptiveConcurrency

1. 通过kubectl连接ASM实例。
2. 使用以下内容，创建*adaptiveconcurrency.yaml*。

   展开查看详细内容

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMAdaptiveConcurrency
metadata:
    name: sample-adaptive-concurrency
    namespace: default
spec:
    workload_selector:
        labels:
            app: testserver
    sample_aggregate_percentile:
        value: 60
    concurrency_limit_params:
        max_concurrency_limit: 500
        concurrency_update_interval: 15s
    min_rtt_calc_params:
        interval: 60s
        request_count: 100
        jitter:
            value: 15
        min_concurrency: 50
        buffer:
            value: 25
```

参数	类型	描述	是否必填
workload_selector	WorkloadSelector	工作负载选择器，声明如何匹配生效的工作负载。	是
labels	map	列出需要匹配的工作负载（Pod）的Labels，用于匹配工作负载。	是
sample_aggregate_percentile	Percent	采样百分位，将采样请求的该百分位值作为SampleRTT参与计算。	是
value	int	百分比数，有效值为0~100。	是
concurrency_limit_params	ConcurrencyLimitParams	并发限制相关配置。	是
max_concurrency_limit	int	最大并发限制数。默认值1000。	否
concurrency_update_interval	duration	最大并发计算间隔，例如60s。	是
min_rtt_calc_params	MinRTTCalcParams	计算MinRTT的相关配置。	是
interval	duration	理想round-trip time计算间隔，例如120s。	否
request_count	int	统计多少请求的数据用作计算理想round-trip time。默认值为50。	否

参数	类型	描述	是否必填
jitter	Percent	计算理想round-trip time时，增加随机延迟的百分比，例如，interval为120s时，jitter为50，则间隔时间为ramdom（120, 120 + （120 * 50%））。默认值为15。	否
min_concurrency	int	计算理想round-trip time时的并发数量，同时也是控制器初始的并发数值。该值的取值应当远低于服务的瓶颈，从而使得服务在该并发下可以以最理想的延迟返回。默认值为3。	否
buffer	Percent	合理延迟波动范围（百分比），例如延迟在100ms上下时，10%浮动属于合理范围，则这个值设置为10。默认值为25。	否

3. 执行以下命令，创建ASMAdaptiveConcurrency。

```
kubectl apply -f adaptiveconcurrency.yaml
```

步骤三：启用ARMS Prometheus监控

为了对并发控制器的运行状态有直观的理解，以及便于对参数调优，您可以将自适应并发控制的相关Metrics输出到ARMS Prometheus，使用ARMS Prometheus监控查看并发控制器的运行状态。

1. 启用ARMS Prometheus监控，具体操作，请参见ARMS Prometheus监控。
2. 在集群中配置ServiceMonitor，使得ARMS Prometheu可以获取到testserver数据。
   i. 使用以下内容，创建*servicemonitor.yaml*。

      展开查看详细内容

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: testserver-envoy-metrics
  namespace: default
spec:
  endpoints:
    - interval: 5s
      path: /stats/prometheus
      port: metrics
  namespaceSelector:
    any: true
  selector:
    matchLabels:
      app: testserver
```

   ii. 执行以下命令，创建ServiceMonitor。

```
kubectl apply -f servicemonitor.yaml
```

步骤四：验证ASMAdaptiveConcurrency的自适应并发控制是否成功

1. 设置gotest的副本数为5。

   一个gotest应用副本将发起200个请求，5个副本则将发起1000个请求。

   i. 登录容器服务管理控制台。
   ii. 在控制台左侧导航栏中，单击**集群**。
   iii. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
   iv. 在集群管理页左侧导航栏中，选择**工作负载 > 无状态**。
   v. 在**无状态**页面设置**命名空间**为default，在gotest应用右侧**操作**列下选择**更多 > 查看YAML**。
   vi. 在**编辑YAML**对话框设置replicas为*5*，单击**更新**。

2. 在应用实时监控服务ARMS导入Dashboard，查看并发控制器的运行状态。

   i. 登录容器服务管理控制台。
   ii. 在控制台左侧导航栏中，单击**集群**。
   iii. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
   iv. 在集群管理页左侧导航栏中，选择**运维管理 > Prometheus监控**。
   v. 在**Prometheus监控**页面右上角单击**跳转到应用实时监控服务ARMS**。
   vi. 在**大盘列表**页面单击任意大盘的名称。
   vii. 在左侧导航栏选择 **> Import**。
   viii. 在**Import via panel json**文本框中输入以下内容，单击**Load**。

   展开查看详细内容

```
{
  "annotations": {
    "list": [
      {
        "builtIn": 1,
        "datasource": "-- Grafana --",
        "enable": true,
        "hide": true,
        "iconColor": "rgba(0, 211, 255, 1)",
        "name": "Annotations & Alerts",
        "type": "dashboard"
      }
    ]
  },
  "description": "monitoring ASM Adaptive Concurrency",
  "editable": true,
  "gnetId": 6693,
  "graphTooltip": 0,
  "id": 3239002,
  "iteration": 1651922323976,
  "links": [],
  "panels": [
    {
      "aliasColors": {},
      "bars": false,
      "dashLength": 10,
      "dashes": false,
      "datasource": "$cluster"
```

```
      "datasource": "$cluster",
      "fieldConfig": {
        "defaults": {
          "custom": {}
        },
        "overrides": []
      },
      "fill": 1,
      "fillGradient": 0,
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 0,
        "y": 0
      },
      "hiddenSeries": false,
      "id": 22,
      "legend": {
        "avg": false,
        "current": false,
        "max": false,
        "min": false,
        "show": true,
        "total": false,
        "values": false
      },
      "lines": true,
      "linewidth": 1,
      "nullPointMode": "null",
      "options": {
        "alertThreshold": true
      },
      "percentage": false,
      "pluginVersion": "7.4.0-pre",
      "pointradius": 2,
      "points": false,
      "renderer": "flot",
      "seriesOverrides": [],
      "spaceLength": 10,
      "stack": false,
      "steppedLine": false,
      "targets": [
        {
          "expr": "envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_controller_rq_blocked{service=\"$service\", pod=\"$pod\"}",
          "interval": "",
          "legendFormat": "{{service}}-{{pod}}",
          "queryType": "randomWalk",
          "refId": "A"
        }
      ],
      "thresholds": [],
      "timeFrom": null,
      "timeRegions": [],
      "timeShift": null,
```

```
      "title": "RqBlocked",
      "tooltip": {
        "shared": true,
        "sort": 0,
        "value_type": "individual"
      },
      "type": "graph",
      "xaxis": {
        "buckets": null,
        "mode": "time",
        "name": null,
        "show": true,
        "values": []
      },
      "yaxes": [
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        },
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        }
      ],
      "yaxis": {
        "align": false,
        "alignLevel": null
      }
    },
    {
      "aliasColors": {},
      "bars": false,
      "dashLength": 10,
      "dashes": false,
      "datasource": "$cluster",
      "fieldConfig": {
        "defaults": {
          "custom": {}
        },
        "overrides": []
      },
      "fill": 1,
      "fillGradient": 0,
      "gridPos": {
        "h": 8,
        "w": 12,
```

```
        "x": 12,
        "y": 0
      },
      "hiddenSeries": false,
      "id": 24,
      "legend": {
        "avg": false,
        "current": false,
        "max": false,
        "min": false,
        "show": true,
        "total": false,
        "values": false
      },
      "lines": true,
      "linewidth": 1,
      "nullPointMode": "null",
      "options": {
        "alertThreshold": true
      },
      "percentage": false,
      "pluginVersion": "7.4.0-pre",
      "pointradius": 2,
      "points": false,
      "renderer": "flot",
      "seriesOverrides": [],
      "spaceLength": 10,
      "stack": false,
      "steppedLine": false,
      "targets": [
        {
          "expr": "envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_controller_burst_queue_size{service=\"$service\", pod=\"$pod\"}",
          "format": "time_series",
          "interval": "",
          "legendFormat": "{{service}}-{{pod}}",
          "queryType": "randomWalk",
          "refId": "A"
        }
      ],
      "thresholds": [],
      "timeFrom": null,
      "timeRegions": [],
      "timeShift": null,
      "title": "HeadRoom",
      "tooltip": {
        "shared": true,
        "sort": 0,
        "value_type": "individual"
      },
      "type": "graph",
      "xaxis": {
        "buckets": null,
        "mode": "time",
        "name": null,
```

```
        "name": null,
        "show": true,
        "values": []
      },
      "yaxes": [
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        },
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        }
      ],
      "yaxis": {
        "align": false,
        "alignLevel": null
      }
    },
    {
      "aliasColors": {},
      "bars": false,
      "dashLength": 10,
      "dashes": false,
      "datasource": "$cluster",
      "fieldConfig": {
        "defaults": {
          "custom": {}
        },
        "overrides": []
      },
      "fill": 1,
      "fillGradient": 0,
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 0,
        "y": 8
      },
      "hiddenSeries": false,
      "id": 26,
      "legend": {
        "avg": false,
        "current": false,
        "max": false,
        "min": false,
        "show": true,
```

```
        "total": false,
        "values": false
      },
      "lines": true,
      "linewidth": 1,
      "nullPointMode": "null",
      "options": {
        "alertThreshold": true
      },
      "percentage": false,
      "pluginVersion": "7.4.0-pre",
      "pointradius": 2,
      "points": false,
      "renderer": "flot",
      "seriesOverrides": [],
      "spaceLength": 10,
      "stack": false,
      "steppedLine": false,
      "targets": [
        {
          "expr": "envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_con
troller_concurrency_limit{service=\"$service\",pod=\"$pod\"}",
          "interval": "",
          "legendFormat": "{{service}}-{{pod}}",
          "queryType": "randomWalk",
          "refId": "A"
        }
      ],
      "thresholds": [],
      "timeFrom": null,
      "timeRegions": [],
      "timeShift": null,
      "title": "ConcurrencyLimit",
      "tooltip": {
        "shared": true,
        "sort": 0,
        "value_type": "individual"
      },
      "type": "graph",
      "xaxis": {
        "buckets": null,
        "mode": "time",
        "name": null,
        "show": true,
        "values": []
      },
      "yaxes": [
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
```

```
        },
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        }
      ],
      "yaxis": {
        "align": false,
        "alignLevel": null
      }
    },
    {
      "aliasColors": {},
      "bars": false,
      "dashLength": 10,
      "dashes": false,
      "datasource": "$cluster",
      "fieldConfig": {
        "defaults": {
          "custom": {}
        },
        "overrides": []
      },
      "fill": 1,
      "fillGradient": 0,
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 12,
        "y": 8
      },
      "hiddenSeries": false,
      "id": 28,
      "legend": {
        "avg": false,
        "current": false,
        "max": false,
        "min": false,
        "show": true,
        "total": false,
        "values": false
      },
      "lines": true,
      "linewidth": 1,
      "nullPointMode": "null",
      "options": {
        "alertThreshold": true
      },
      "percentage": false,
      "pluginVersion": "7.4.0-pre",
      "pointradius": 2,
```

```
      "points": false,
      "renderer": "flot",
      "seriesOverrides": [],
      "spaceLength": 10,
      "stack": false,
      "steppedLine": false,
      "targets": [
        {
          "expr": "envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_con
troller_gradient{service=\"$service\",pod=\"$pod\"}",
          "interval": "",
          "legendFormat": "{{service}}-{{pod}}",
          "queryType": "randomWalk",
          "refId": "A"
        }
      ],
      "thresholds": [],
      "timeFrom": null,
      "timeRegions": [],
      "timeShift": null,
      "title": "Gradient",
      "tooltip": {
        "shared": true,
        "sort": 0,
        "value_type": "individual"
      },
      "type": "graph",
      "xaxis": {
        "buckets": null,
        "mode": "time",
        "name": null,
        "show": true,
        "values": []
      },
      "yaxes": [
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        },
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        }
      ],
      "yaxis": {
        "align": false,
```

```
        "alignLevel": null
      }
    },
    {
      "aliasColors": {},
      "bars": false,
      "dashLength": 10,
      "dashes": false,
      "datasource": "$cluster",
      "fieldConfig": {
        "defaults": {
          "custom": {}
        },
        "overrides": []
      },
      "fill": 1,
      "fillGradient": 0,
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 0,
        "y": 16
      },
      "hiddenSeries": false,
      "id": 32,
      "legend": {
        "avg": false,
        "current": false,
        "max": false,
        "min": false,
        "show": true,
        "total": false,
        "values": false
      },
      "lines": true,
      "linewidth": 1,
      "nullPointMode": "null",
      "options": {
        "alertThreshold": true
      },
      "percentage": false,
      "pluginVersion": "7.4.0-pre",
      "pointradius": 2,
      "points": false,
      "renderer": "flot",
      "seriesOverrides": [],
      "spaceLength": 10,
      "stack": false,
      "steppedLine": false,
      "targets": [
        {
          "expr": "envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_con
troller_min_rtt_msecs{service=\"$service\",pod=\"$pod\"}",
          "interval": "",
          "legendFormat": "{{service}}-{{pod}}"
```

```
            "legendFormat": "{{service}}-{{pod}}",
            "queryType": "randomWalk",
            "refId": "A"
          }
        ],
        "thresholds": [],
        "timeFrom": null,
        "timeRegions": [],
        "timeShift": null,
        "title": "MinRTT(msec)",
        "tooltip": {
          "shared": true,
          "sort": 0,
          "value_type": "individual"
        },
        "type": "graph",
        "xaxis": {
          "buckets": null,
          "mode": "time",
          "name": null,
          "show": true,
          "values": []
        },
        "yaxes": [
          {
            "format": "ms",
            "label": null,
            "logBase": 1,
            "max": null,
            "min": null,
            "show": true
          },
          {
            "format": "short",
            "label": null,
            "logBase": 1,
            "max": null,
            "min": null,
            "show": true
          }
        ],
        "yaxis": {
          "align": false,
          "alignLevel": null
        }
      },
      {
        "aliasColors": {},
        "bars": false,
        "dashLength": 10,
        "dashes": false,
        "datasource": "$cluster",
        "fieldConfig": {
          "defaults": {
            "custom": {}
```

```
        },
        "overrides": []
      },
      "fill": 1,
      "fillGradient": 0,
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 12,
        "y": 16
      },
      "hiddenSeries": false,
      "id": 34,
      "legend": {
        "avg": false,
        "current": false,
        "max": false,
        "min": false,
        "show": true,
        "total": false,
        "values": false
      },
      "lines": true,
      "linewidth": 1,
      "nullPointMode": "null",
      "options": {
        "alertThreshold": true
      },
      "percentage": false,
      "pluginVersion": "7.4.0-pre",
      "pointradius": 2,
      "points": false,
      "renderer": "flot",
      "seriesOverrides": [],
      "spaceLength": 10,
      "stack": false,
      "steppedLine": false,
      "targets": [
        {
          "expr": "envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_con
troller_sample_rtt_msecs{service=\"$service\",pod=\"$pod\"}",
          "interval": "",
          "legendFormat": "{{service}}-{{pod}}",
          "queryType": "randomWalk",
          "refId": "A"
        }
      ],
      "thresholds": [],
      "timeFrom": null,
      "timeRegions": [],
      "timeShift": null,
      "title": "SampleRTT(msec)",
      "tooltip": {
        "shared": true,
```

```
        "sort": 0,
        "value_type": "individual"
      },
      "type": "graph",
      "xaxis": {
        "buckets": null,
        "mode": "time",
        "name": null,
        "show": true,
        "values": []
      },
      "yaxes": [
        {
          "format": "ms",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        },
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        }
      ],
      "yaxis": {
        "align": false,
        "alignLevel": null
      }
    },
    {
      "aliasColors": {},
      "bars": false,
      "dashLength": 10,
      "dashes": false,
      "datasource": "test-adaptive-concurrency_1217520382582089",
      "fieldConfig": {
        "defaults": {
          "custom": {}
        },
        "overrides": []
      },
      "fill": 1,
      "fillGradient": 0,
      "gridPos": {
        "h": 8,
        "w": 12,
        "x": 0,
        "y": 24
      },
      "hiddenSeries": false,
```

```
      "id": 30,
      "legend": {
        "avg": false,
        "current": false,
        "max": false,
        "min": false,
        "show": true,
        "total": false,
        "values": false
      },
      "lines": true,
      "linewidth": 1,
      "nullPointMode": "null",
      "options": {
        "alertThreshold": true
      },
      "percentage": false,
      "pluginVersion": "7.4.0-pre",
      "pointradius": 2,
      "points": false,
      "renderer": "flot",
      "seriesOverrides": [],
      "spaceLength": 10,
      "stack": false,
      "steppedLine": false,
      "targets": [
        {
          "expr": "envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_con
troller_min_rtt_calculation_active{service=\"$service\",pod=\"$pod\"}",
          "interval": "",
          "legendFormat": "{{service}}-{{pod}}",
          "queryType": "randomWalk",
          "refId": "A"
        }
      ],
      "thresholds": [],
      "timeFrom": null,
      "timeRegions": [],
      "timeShift": null,
      "title": "MinRTTCalc",
      "tooltip": {
        "shared": true,
        "sort": 0,
        "value_type": "individual"
      },
      "type": "graph",
      "xaxis": {
        "buckets": null,
        "mode": "time",
        "name": null,
        "show": true,
        "values": []
      },
      "yaxes": [
```

```
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        },
        {
          "format": "short",
          "label": null,
          "logBase": 1,
          "max": null,
          "min": null,
          "show": true
        }
      ],
      "yaxis": {
        "align": false,
        "alignLevel": null
      }
    }
  ],
  "refresh": "5s",
  "schemaVersion": 26,
  "style": "dark",
  "tags": [],
  "templating": {
    "list": [
      {
        "current": {
          "selected": true,
          "text": "edas120_1217520382582089",
          "value": "edas120_1217520382582089"
        },
        "error": null,
        "hide": 0,
        "includeAll": false,
        "label": null,
        "multi": false,
        "name": "cluster",
        "options": [],
        "query": "prometheus",
        "queryValue": "",
        "refresh": 1,
        "regex": "",
        "skipUrlSync": false,
        "type": "datasource"
      },
      {
        "allValue": null,
        "current": {
          "isNone": true,
          "selected": false,
          "text": "None"
```

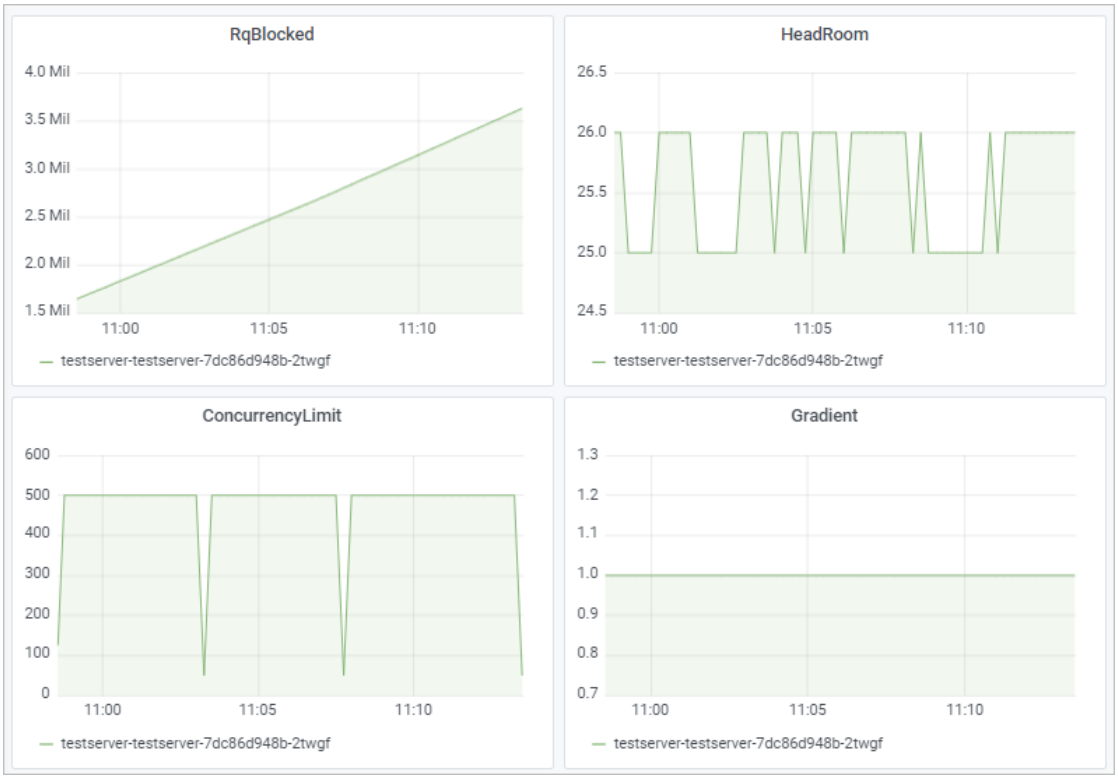
```
          "text": "None",
          "value": ""
        },
        "datasource": "$cluster",
        "definition": "label_values(envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_controller_burst_queue_size,service)",
        "error": null,
        "hide": 0,
        "includeAll": false,
        "label": null,
        "multi": false,
        "name": "service",
        "options": [],
        "query": "label_values(envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_controller_burst_queue_size,service)",
        "refresh": 2,
        "regex": "",
        "skipUrlSync": false,
        "sort": 1,
        "tagValuesQuery": "",
        "tags": [],
        "tagsQuery": "",
        "type": "query",
        "useTags": false
      },
      {
        "allValue": null,
        "current": {
          "selected": false,
          "text": "All",
          "value": "$__all"
        },
        "datasource": "$cluster",
        "definition": "label_values(envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_controller_concurrency_limit, pod)",
        "error": null,
        "hide": 0,
        "includeAll": true,
        "label": null,
        "multi": true,
        "name": "pod",
        "options": [],
        "query": "label_values(envoy_http_inbound_0_0_0_0_8080_adaptive_concurrency_gradient_controller_concurrency_limit, pod)",
        "refresh": 2,
        "regex": "",
        "skipUrlSync": false,
        "sort": 0,
        "tagValuesQuery": "",
        "tags": [],
        "tagsQuery": "",
        "type": "query",
        "useTags": false
      }
    ]
```

```
  },
  "time": {
    "from": "now-15m",
    "to": "now"
  },
  "timepicker": {
    "refresh_intervals": [
      "5s",
      "10s",
      "30s",
      "1m",
      "5m",
      "15m",
      "30m",
      "1h",
      "2h",
      "1d"
    ],
    "time_options": [
      "5m",
      "15m",
      "1h",
      "6h",
      "12h",
      "24h",
      "2d",
      "7d",
      "30d"
    ]
  },
  "timezone": "",
  "title": "ASM Adaptive Concurrency",
  "uid": "000000084",
  "version": 3
}
```

ix. 在Options页面单击Import。

> **说明** 如果Options页面提示大盘UID冲突，您需要手动修改大盘UID，然后单击Import。

x. 在Dashboard中选择集群，设置**Service**为testserver，**Pod**为ALL。



可以看到，gotest应用向testserver应用发起了1000个请求，但是testserver应用始终限制收到的并发数在500以内。说明使用ASMASMAdaptiveConcurrency的自适应并发控制成功。

15.管理Spring Cloud服务

您可以将Spring Cloud业务应用接入ASM，从而可以使用云原生化的服务治理能力，不需要业务做任何代码修改，即可管理Spring Cloud业务服务。本文介绍如何使用ASM管理Spring Cloud服务。

前提条件

- 已创建ASM实例。具体操作，请参见创建ASM实例。
- 已创建ACK集群。具体操作，请参见创建Kubernetes托管版集群。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。

背景信息

Spring Cloud是一个标准，有不同的实现，比如Spring Cloud Netflix、Spring Cloud Alibaba、Spring Cloud Consul 等。不同的Spring Cloud 实现对于ASM 来说核心区别主要在于采用了不同的服务发现，ASM 针对这些不同的Spring Cloud 版本迁移支持列表如下：

Spring Cloud版本	服务发现	迁移支持 （零代码修改）
Spring Cloud Alibaba	MSE Nacos （阿里云上产品）	支持
Spring Cloud Alibaba	Nacos（自建）	支持
Spring Cloud Netflix	Eureka	暂不支持，建议采用Nacos注册中心
Spring Cloud Consul	Consul	暂不支持，建议采用Nacos注册中心
Spring Cloud Zookeeper	Zookeeper	暂不支持，建议采用Nacos注册中心

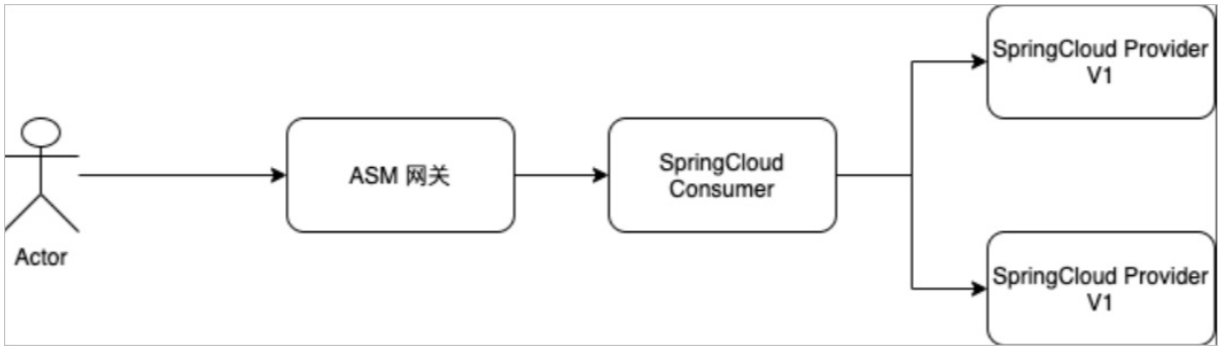
Demo介绍

本文部署的Spring Cloud服务可以通过此nacos-example下载。

Spring Cloud服务包含Consumer服务和Provider服务，其中Provider有v1和v2两个版本，并且都注册到Nacos注册中心。Consumer从Nacos注册中心同步Provider服务地址进行负载均衡发起请求，其中Consumer暴露一个8080端口，提供了一个echo接口，对应逻辑是将请求转发给Provider，并输出Provider返回的结果，不同的Provider版本返回结果不同：

- Provider v1版本收到echo请求会返回 `Hello Nacos Discovery From v1xxx` 。
- Provider v2版本收到echo请求会返回 `Hello Nacos Discovery From v2xxx` 。

其中返回结果中 `.xxx` 为echo接口对应的具体参数，例如请求*/echo/world*发送到Provider v1版本，则会返回 `Hello Nacos Discovery From v1world` 。



步骤一：配置ServiceEntry和EnvoyFilter

您需要在ASM中配置ServiceEntry和EnvoyFilter，使ASM可以管理Spring Cloud服务。

1. 通过kubectl连接ASM实例。
2. 创建ServiceEntry。
   i. 使用以下内容，创建*external-nacos-svc.yaml*。

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: external-nacos-svc
spec:
  hosts:
  - "NACOS_SERVER_HOST"  ## 需要替换为你的Nacos Server HOST，例如"mse-xxx-p.nacos-ans.
mse.aliyuncs.com"。
  location: MESH_EXTERNAL
  ports:
  - number: 8848
    name: http
  resolution: DNS
```

   其中8848为Nacos的默认端口，如果您是自建的Nacos Server且对端口有修改，则number参数也需要对应修改。

   ii. 执行以下命令，创建ServiceEntry。

```
kubectl apply -f external-nacos-svc.yaml
```

3. 创建EnvoyFilter。
   i. 使用以下内容，创建*external-envoyfilter.yaml*。

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  labels:
    provider: "asm"
    asm-system: "true"
  name: nacos-subscribe-lua
  namespace: istio-system
spec:
  configPatches:
    # The first patch adds the lua filter to the listener/http connection manager
  - applyTo: HTTP_FILTER
    match:
      proxy:
        proxyVersion: "^1.*"
      context: SIDECAR_OUTBOUND
      listener:
        portNumber: 8848
        filterChain:
          filter:
            name: "envoy.filters.network.http_connection_manager"
            subFilter:
              name: "envoy.filters.http.router"
```

```
    patch:
      operation: INSERT_BEFORE
      value: # lua filter specification
       name: envoy.lua
       typed_config:
          "@type": "type.googleapis.com/envoy.extensions.filters.http.lua.v3.Lua"
          inlineCode: |
             -- copyright: ASM (Alibaba Cloud ServiceMesh)
             function envoy_on_request(request_handle)
               local request_headers = request_handle:headers()
               -- /nacos/v1/ns/instance/list?healthyOnly=false&namespaceId=public&cl
ientIP=11.122.63.81&serviceName=DEFAULT_GROUP%40%40service-provider&udpPort=53174&en
coding=UTF-8
               local path = request_headers:get(":path")
               if string.match(path,"^/nacos/v1/ns/instance/list") then
                 local servicename = string.gsub(path,".*&serviceName.-%%40([%w.\\_\
\-]+)&.*","%1")
                 request_handle:streamInfo():dynamicMetadata():set("context", "reque
st.path", path)
                 request_handle:streamInfo():dynamicMetadata():set("context", "reque
st.servicename", servicename)
                 request_handle:logInfo("subscribe for serviceName: " .. servicename
)
               else
                 request_handle:streamInfo():dynamicMetadata():set("context", "reque
st.path", "")
               end
             end
             function envoy_on_response(response_handle)
               local request_path = response_handle:streamInfo():dynamicMetadata():g
et("context")["request.path"]
               if request_path == "" then
                  return
               end
               local servicename = response_handle:streamInfo():dynamicMetadata():ge
t("context")["request.servicename"]
               response_handle:logInfo("modified response ip to serviceName:" .. ser
vicename)
               local bodyObject = response_handle:body(true)
               local body= bodyObject:getBytes(0,bodyObject:length())
               body = string.gsub(body,"%s+","")
               body = string.gsub(body,"(ip\":\")(%d+.%d+.%d+.%d+)","%1"..servicenam
e)
               response_handle:body():setBytes(body)
             end
```

ii. 执行以下命令，创建EnvoyFilter。

```
kubectl apply -f external-envoyfilter.yaml
```

步骤二：在ACK创建Spring Cloud服务

> **说明**
>
> - 因为需要拦截注册流程，EnvoyFilter需要先于业务工作负载Deployment之前创建。若某些业务Deployment先于EnvoyFilter创建，您需要滚动更新该业务Deployment。
> - 业务服务需要创建Kubernetes Service资源，并且需要有Cluster IP。

1. 通过kubectl工具连接集群。
2. 执行以下命令，创建Spring Cloud服务。

```
export NACOS_ADDRESS=xxxx # xxxx为MSE或自建的Nacos地址，建议使用VPC内网地址。
wget https://alibabacloudservicemesh.oss-cn-beijing.aliyuncs.com/asm-labs/springcloud/demo.yaml -O demo.yaml
sed -e "s/NACOS_SERVER_CLUSTERIP/$NACOS_ADDRESS/g" demo.yaml |kubectl apply -f -
```

   关于创建Nacos引擎的具体操作，请参见创建Nacos引擎。

3. 执行以下命令，查看Spring Cloud服务。

```
kubectl get pods
```

   预期输出：

```
consumer-bdd464654-jn8q7       2/2     Running     0          25h
provider-v1-66bc67fb6d-46pgl   2/2     Running     0          25h
provider-v2-76568c45f6-85z87   2/2     Running     0          25h
```

步骤三：创建网关规则和虚拟服务

1. 通过kubectl连接ASM实例。
2. 创建网关规则。
   i. 使用以下内容，创建*test-gateway.yaml*。

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: test-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

   ii. 执行以下命令，创建网关规则。

```
kubectl apply -f test-gateway.yaml
```

3. 创建虚拟服务。

i. 使用以下内容，创建*consumer.yaml*。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: consumer
spec:
  hosts:
  - "*"
  gateways:
  - test-gateway
  http:
  - match:
    - uri:
        prefix: /
    route:
    - destination:
        host: consumer.default.svc.cluster.local
        port:
          number: 8080
```

ii. 执行以下命令，创建虚拟服务。

```
kubectl apply -f consumer.yaml
```

步骤四：验证ASM是否管理Spring Cloud服务成功

1. 查看Ingress Gateway的IP地址。
   i. 登录ASM控制台。
   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。
   iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
   iv. 在网格管理页面左侧导航栏单击**ASM网关**。
   在ASM网关页面查看Ingress Gateway **Kubernetes服务**列下的IP地址。
2. 执行以下命令，通过Ingress Gateway向Spring Cloud Consumer服务发起请求。

```
curl <Ingress Gateway的IP地址>/echo/world
```

预期输出：

```
Hello Nacos Discovery From v1world
Hello Nacos Discovery From v2world
Hello Nacos Discovery From v1world
Hello Nacos Discovery From v2world
```

可以看到Provider默认在v1、v2版本间轮询访问。

3. 创建目标规则和虚拟服务。

i. 使用以下内容，创建*service-provider.yaml*。

```
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: service-provider
spec:
  host: service-provider
  subsets:
  - name: v1
    labels:
      label: v1
  - name: v2
    labels:
      label: v2
```

ii. 执行以下命令，创建目标规则。

```
kubectl apply -f service-provider.yaml
```

iii. 使用以下内容，创建*service-provider1.yaml*。

以下虚拟服务定义了*/echo/hello*的请求将被路由到Provider v1版本，其他请求将被路由到Provider v2版本。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: service-provider
spec:
  hosts:
  - service-provider
  http:
  - name: "hello-v1"
    match:
    - uri:
        prefix: "/echo/hello"
    route:
    - destination:
        host: service-provider
        subset: v1
  - name: "default"
    route:
    - destination:
        host: service-provider
        subset: v2
```

iv. 执行以下命令，创建虚拟服务。

```
kubectl apply -f service-provider1.yaml
```

4. 执行以下命令，向Spring Cloud Consumer服务发起请求。

```
curl <Ingress Gateway的IP地址>/echo/hello
```

预期输出：

```
Hello Nacos Discovery From v1hello
Hello Nacos Discovery From v1hello
```

可以看到/echo/hello请求都被路由到Provider v1版本，其他请求则会路由到Provider v2版本。说明Spring Cloud 流量被Istio接管，并可以支持使用Istio方式配置相关路由规则，管理Spring Cloud服务成功。

FAQ

为什么我参照该文档，自己部署的SpringCloud业务服务不生效呢？

1. 请检查是否开启了针对Nacos端口或者IP的流量拦截
2. 因为需要拦截注册流程，EnvoyFilter需要先于业务工作负载Deployment之前创建。若某些业务Deployment先于EnvoyFilter创建，您需要滚动更新该业务Deployment。
3. 请检查业务服务是否创建了Kubernetes Service资源，并且Type类型为Cluster IP。
4. Nacos Client SDK版本需低于2.0版本，因为Nacos 2.0+ Client SDK 采用GRPC和服务端建立，当前方案不适用。