



应用服务网格 (CSM)

用户使用指南

天翼云科技有限公司

1. 简介

1.1 应用服务网格介绍

天翼云应用服务网格（CSM）提供兼容开源 istio 的服务网格能力，并基于开源实现做了功能增强，提供了丰富的流量治理、安全和可观测能力，大大降低开发和运维的负担。

1.2 产品优势

开源增强

100% 兼容 istio 服务网格，基于开源做了功能增强，包括集成了 OPA 策略引擎，提供强大的安全能力；实现了流量打标和全链路灰度能力，提供了丰富的流量治理能力；无缝对接天翼云容器引擎（CCSE），支持多集群统一治理。

流量治理

提供多集群治理能力，支持基于 istio VS、DR 资源的流量治理；基于开源扩展了本地限流功能以及流量打标和全链路灰度能力。

安全

一键开启 mTLS 安全链路，支持丰富的请求认证和授权策略；支持使用外部授权服务对网格内的服务进行访问授权；同时支持一键集成 OPA 策略引擎，支持丰富的访问策略。

无侵入

提供基于 webhook 的 sidecar 注入能力，sidecar 自动拦截业务流量实现丰富的服务治理能力，无需修改任何业务代码。

1.3 应用场景

多语言微服务统一治理

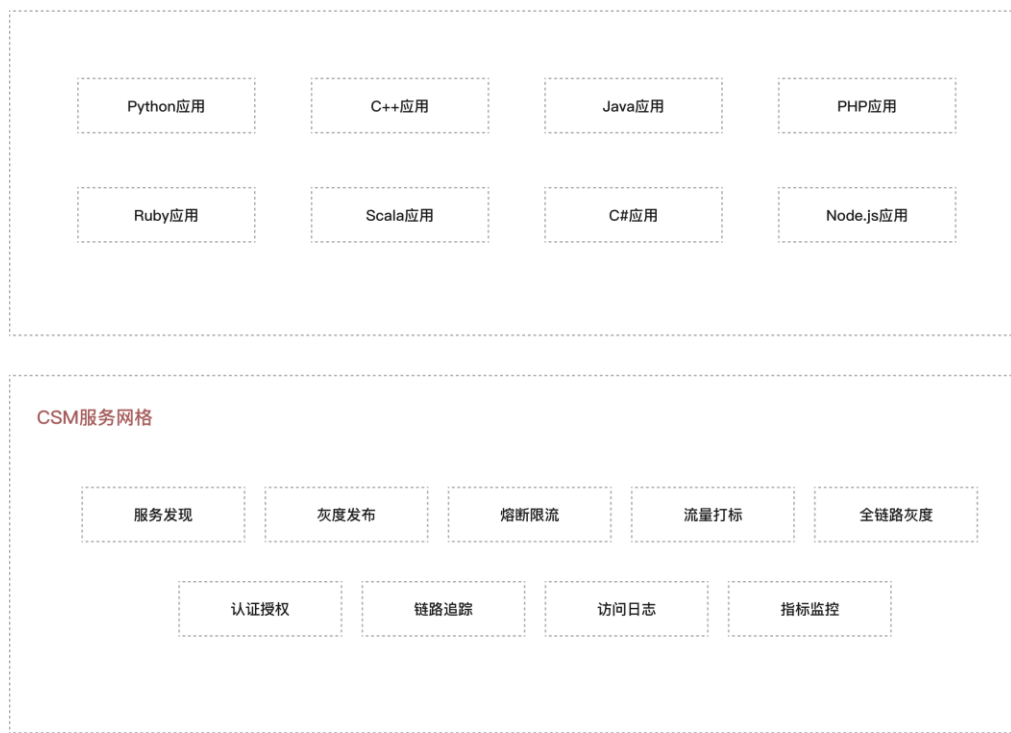
CSM 服务网格采用无侵入式的 sidecar 模式，提供了与语言无关的服务治理能力，无需修改业务代码即可实现对多语言应用的灰度发布、熔断限流、标签路由、全链路灰度等治理能力。

解决问题：

- (1) 服务治理能力与业务代码耦合问题，业务无需关注非业务的技术问题，提高业

务迭代效率。

- (2) 企业内部多语言业务互通问题，服务网格通过 sidecar 和统一控制面，屏蔽了语言和框架的差异，使得多语言框架应用之间的通信就像语言框架内部的通信一样简单。

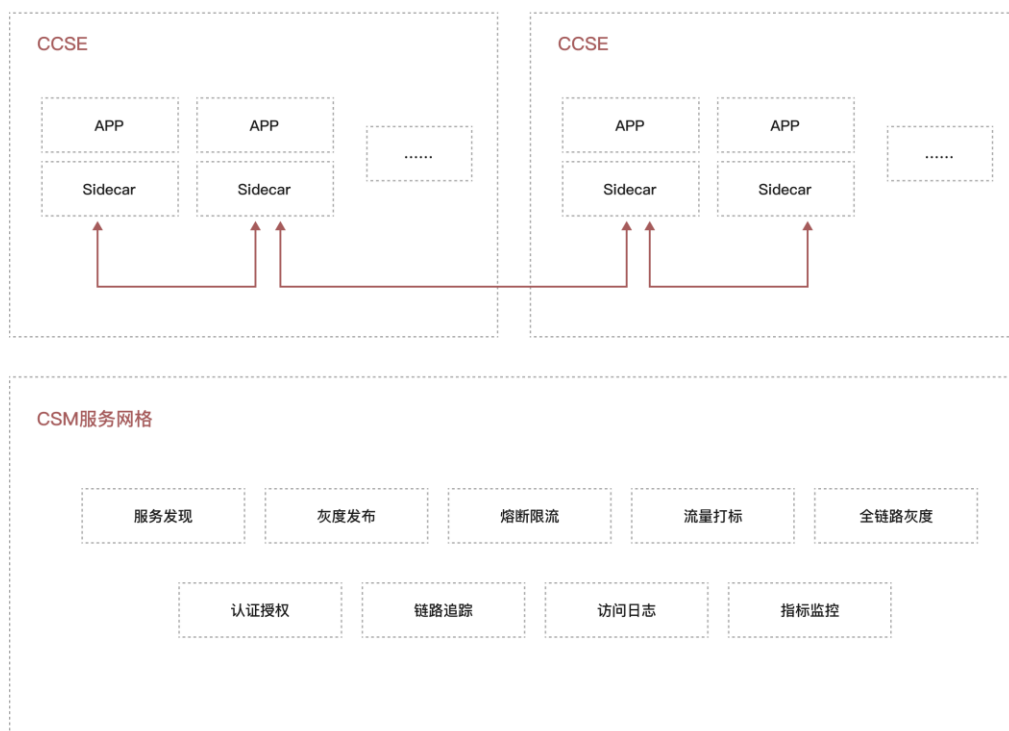


多集群统一服务治理

CSM 服务网格采用主从集群模式，主集群（CCSE）作为控制面部署集群，支持对多个从集群（CCSE）实行统一纳管，对多个 CCSE 集群上的服务进行统一治理。

解决问题：

- (1) 服务扩展问题：随着业务的发展，业务通过单个容器集群难以支撑时，CSM 服务网格可以在一个网格实例下实现对多个容器集群的统一治理。
- (2) 容灾问题：基于多集群和服务标签，通过服务网格的路由能力可以实现业务的多活容灾。



1.4 产品规格

版本	支持 pod 数量
基础版	50
标准版	1000
企业版	10000

1.5 基本术语

控制面 (Control plane)

控制面是整个服务网格的控制中枢，负责整个网格的管理，包括 sidecar 注入，服务发现和服务治理配置分发以及证书管理功能等；

数据面 (Data plane)

数据面是实际执行流量治理的代理服务器，在 istio 里面采用 Envoy 作为流量代理

服务器，Envoy 会拦截进出业务服务的流量并执行相应的流量治理策略。

sidecar 注入 (sidecar injection)

在业务 pod 内部增加一个 sidecar，用于实现流量拦截和代理功能，称为 sidecar 注入；在 K8s 的使用场景下，一般采用 webhook 机制实现业务无感知的 sidecar 注入。

虚拟服务 (Virtual Service)

虚拟服务是 istio 定义的流量治理对象，能够实现对指定服务配置路由规则的功能，匹配到该路由规则的请求将根据配置被转发到相应的目标。

目标规则 (Destination Rule)

目标规则是 istio 定义的用于管理流量转发目标服务的对象，比如可以使用目标规则设置要转发的目标服务的版本、超时重试策略、熔断策略等。

对等身份认证 (Peer Authentication)

在服务网格内，服务之间通信都要经过 sidecar，对等身份认证策略定义了 sidecar 之间的通信安全策略，可以是明文传输，或者强制 TLS 加密通信，或者两种都可以。

请求身份认证 (Request Authentication)

请求身份认证定义了服务对于收到的请求的身份认证策略，比如可以配置服务按照 jwt 策略对请求的身份进行认证，认证之后可以基于请求者的身份做进一步的访问授权策略。

授权策略 (Authorization Policy)

授权策略支持对请求是否有权访问当前资源进行限制，可以基于请求身份认证中获取的用户身份信息，或者请求的其他信息对请求的放通或者拒绝做限制。

1.6 使用限制

在使用 CSM 服务网格之前，您需要了解 CSM 服务网格的使用限制。当前使用限制主要有：

限制项	说明
非托管控制面	当前 CSM 服务网格采用非托管控制面，需要您提前开通专有版 CCSE 集群，用于部署控制面服务
控制面服务暴露	当前采用 ELB 暴露服务网格控制面服务，ELB 只提供私网 VPC IP 访问，如需开启公网访问需要您到 ELB 控制台给

	ELB 实例绑定弹性 IP
规格	当前仅提供基础版服务网格，支撑 pod 数量为 50

2. 快速入门

2.1 入门概述

本文介绍从服务网格使用入门的概述说明，具体步骤包括开通网格实例、部署应用、配置网关访问、体验流量治理能力等步骤，简要说明如下：

步骤	说明
创建网格实例	使用服务网格的服务治理能力之前首先要创建一个网关实例
部署应用	部署测试应用到网格实例中
配置网关访问	通过云原生网关实现外部访问网格内的服务
体验流量治理能力	使用 istio 资源体验服务网格的流量治理能力

2.2 创建 CSM 实例

前提条件：

1. 已开通 CCSE（容器云服务引擎），至少有一个 CCSE 集群实例

操作步骤：

使用 CSM 服务网格前需要先创建一个网格实例；从服务网格产品首页到或者网格控制台首页均有入口可以订购服务网格实例；订购服务网格前需要保证您已经开通了天翼云容器引擎 CCSE 实例，网格实例订购页如下：

< 订购服务网格

* 实例名称

最长40字符，只能包含小写字母、数字及分隔符("-")，且必须以小写字母开头，数字或小写字母结尾

版本类型

可支撑Pod规模50个

istio版本

计费模式

控制面部署

CCSE集群

istio控制面ELB

可观测性

启用链路追踪

启用监控指标

启用访问日志采集

启用控制面日志采集

高级选项

nacos注册服务

服务网格订购参数说明如下：

参数	说明
实例名称	用于标识一个网格实例
版本类型	产品规格，分为基础版（支持 50pod），标准版（支持 1000pod），企业版（支持 10000pod）；当前支持基础版
istio 版本号	天翼云服务网格基于 istio 实现，当前支持的 istio 版本为 1.16.2
计费模式	当前支持按需计费模式
控制面部署	服务网格控制面可以部署在租户的 CCSE 实例上或者以托管的方式部署在独立的资源上，当前支持部署在租户 CCSE 的方式

CCSE 集群	选择控制面部署的 CCSE 集群，需要您提前开通 CCSE 实例
istio 控制面 ELB	用于暴露服务网格控制面的 ELB 实例规格
启用链路追踪	启用链路追踪后，可以采集网格内服务访问链路数据并上报到后端存储
采样率	链路追踪采样率
启用监控指标	启用监控指标采集后，会采集网格内服务访问的指标数据，可以通过监控指标页面查看
启用访问日志采集	启用访问日志采集后，网格内的 sidecar 访问日志会被采集到 ALS 日志服务，可以通过 ALS 日志服务查看到日志详情；前提是您已经开通了 ALS 日志服务；
访问日志项目名	可以将访问日志采集到当前已有日志项目或者新建日志项目，新建日志项目名称规则为：istiod_{\$网格实例名称}_accesslog
启用控制面日志采集	启用控制面日志采集后，可以将服务网格控制面日志采集到 ALS 日志服务，可以通过 ALS 日志服务查看到控制面日志详情；前提是您已经开通了 ALS 日志服务；
控制面日志项目名	可以将控制面日志采集放到当前已有的日志项目或者新建日志项目，新建日志项目的规则为：istiod_{\$网格实例名称}_controlepanel
Nacos 注册服务	启用 Nacos 注册服务后，服务网格会通过 Nacos 提供的 MCP over xDS 端口获取到 Nacos 内注册的服务，网格内的服务可以访问这些服务地址
Nacos 注册服务地址	选择当前已经开通的 Nacos 实例

一个网格实例创建一般需要 5 分钟左右时间。

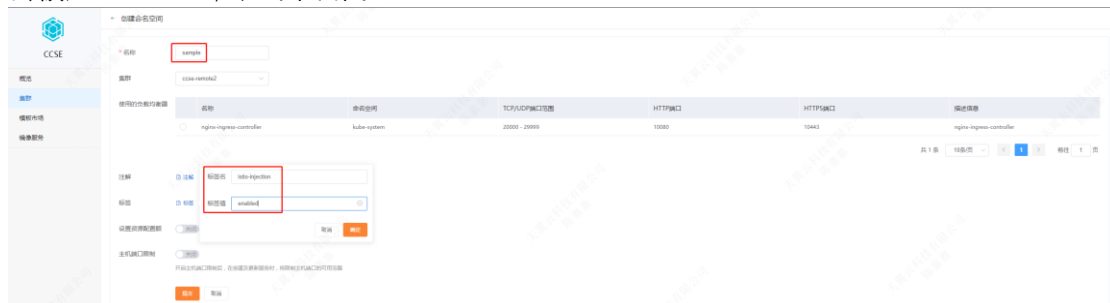
2.3 部署 bookinfo 应用到 CSM 实例

前提条件：

1. 已开通 CCSE（容器云服务引擎），至少有一个 CCSE 集群实例
2. 开通天翼云服务网格实例

操作步骤：

首先到 CCSE 控制台找到当前添加到服务网格的 CCSE 集群，创建测试应用部署的 sample 命名空间，同时给命名空间打上 istio-injection: enabled 的标签以保证该命名空间下的 pod 会被注入 sidecar，如下图所示：



使用如下 yaml 部署我们的 bookinfo 应用，注意根据当前集群所在的资源池替换镜像的地址（当前 CCSE 实例在内蒙 6 资源池，所以我们的镜像地址是 registry-vpc-nm6b-ccr.ctyun.cn:443）；如果您想要把演示应用部署到其他 namespace 也可以修改 yaml 里面的

namespace 字段实现, 当前使用的 yaml 如下 :

```
apiVersion: v1
kind: Service
metadata:
  name: details
  labels:
    withServiceMesh: "true"
    workloadKind: Deployment
    workloadName: details-v1
spec:
  ports:
    - port: 9080
      targetPort: 9080
      name: http
  selector:
    app: details
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: details-v1
  labels:
    withServiceMesh: "true"
spec:
  replicas: 1
  selector:
    matchLabels:
      name: details-v1
  template:
    metadata:
      labels:
        app: details
        version: v1
        name: details-v1
        source: CCSE
        csmAutoEnable: "on"
        "sidecar.istio.io/inject": "true"
    annotations:
      "sidecar.istio.io/inject": "true"
  spec:
    containers:
      - name: details
        image: 'registry-vpc-crs-huadong1.ctyun.cn/library/istio-examples-bookinfo-details-
v1:1.16.2'
        imagePullPolicy: IfNotPresent
    resources:
      limits:
        cpu: "200m"
        memory: "256Mi"
```

```
    requests:
      cpu: "50m"
      memory: "64Mi"
  ---
apiVersion: v1
kind: Service
metadata:
  name: ratings
  labels:
    withServiceMesh: "true"
    workloadKind: Deployment
    workloadName: ratings-v1
spec:
  ports:
    - port: 9080
      targetPort: 9080
      name: http
  selector:
    app: ratings
  ---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ratings-v1
  labels:
    withServiceMesh: "true"
spec:
  replicas: 1
  selector:
    matchLabels:
      name: ratings-v1
  template:
    metadata:
      labels:
        app: ratings
        version: v1
        name: ratings-v1
        source: CCSE
        "sidecar.istio.io/inject": "true"
        csmAutoEnable: "on"
    annotations:
      "sidecar.istio.io/inject": "true"
  spec:
    containers:
      - name: ratings
        image: 'registry-vpc-crs-huadong1.ctyun.cn/library/istio-examples-bookinfo-ratings-
v1:1.16.2'
        imagePullPolicy: IfNotPresent
        resources:
```

```
    limits:
      cpu: "200m"
      memory: "256Mi"
    requests:
      cpu: "50m"
      memory: "64Mi"
  ---
apiVersion: v1
kind: Service
metadata:
  name: reviews
  labels:
    withServiceMesh: "true"
    workloadKind: Deployment
    workloadName: reviews-v1
spec:
  ports:
    - port: 9080
      targetPort: 9080
      name: http
  selector:
    app: reviews
  ---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reviews-v1
  labels:
    withServiceMesh: "true"
spec:
  replicas: 1
  selector:
    matchLabels:
      name: reviews-v1
  template:
    metadata:
      labels:
        app: reviews
        version: v1
        name: reviews-v1
        source: CCSE
        "sidecar.istio.io/inject": "true"
        csmAutoEnable: "on"
    annotations:
      "sidecar.istio.io/inject": "true"
  spec:
    containers:
      - name: reviews
        image: registry-vpc-crs-huadong1.ctyun.cn/library/istio-examples-bookinfo-
```

```
reviews-v1:1.16.2'  
  imagePullPolicy: IfNotPresent  
  env:  
    - name: LOG_DIR  
      value: "/tmp/logs"  
  volumeMounts:  
    - name: tmp  
      mountPath: /tmp  
    - name: wlp-output  
      mountPath: /opt/ibm/wlp/output  
  volumes:  
    - name: wlp-output  
      emptyDir: {}  
    - name: tmp  
      emptyDir: {}  
---
```

```
apiVersion: v1  
kind: Service  
metadata:  
  name: productpage  
  labels:  
    withServiceMesh: "true"  
    workloadKind: Deployment  
    workloadName: productpage-v1  
spec:  
  ports:  
    - port: 9080  
      targetPort: 9080  
      name: http  
  selector:  
    app: productpage  
  type: NodePort  
---
```

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: productpage-v1  
  labels:  
    withServiceMesh: "true"  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      name: productpage-v1  
  template:  
    metadata:  
      labels:  
        app: productpage  
        version: v1
```

```

name: productpage-v1
source: CCSE
"sidecar.istio.io/inject": "true"
csmAutoEnable: "on"
annotations:
  "sidecar.istio.io/inject": "true"
spec:
  containers:
  - name: productpage
    image: 'registry-vpc-crs-huadong1.ctyun.cn/library/istio-examples-bookinfo-productpage-v1:1.16.2'
    imagePullPolicy: IfNotPresent
  resources:
    limits:
      cpu: "200m"
      memory: "256Mi"
    requests:
      cpu: "50m"
      memory: "64Mi"
  volumeMounts:
  - name: tmp
    mountPath: /tmp
  volumes:
  - name: tmp
    emptyDir: {}

```

在 CCSE 控制台选择我们要部署的 sample 命名空间，通过工作负载->无状态->新增 yaml，依次部署上面的 deployment；通过网络->服务->新增 yaml 依次部署上面的 Service 填入我们准备的 yaml 文件，保存即可，如下图所示：



部署完成后，可以在工作负载->无状态列表里看到 bookinfo 相关的部署信息，在网络->Service 菜单下可以看到刚部署的 Service 列表，如下图：



名称	类型	所属的工作负载	serviceip	端口	访问方式	创建时间	操作
details	ClusterIP	Deployment: details-v1	10.96.105.30	9080/TCP	集群内访问: details.sample:9080	2023-07-22 18:32:43	删除 刷新 查看详情
helloworld	ClusterIP	Deployment: helloworld-v2	10.96.79.82	5000/TCP	集群内访问: helloworld.sample:5000	2023-07-17 11:20:54	删除 刷新 查看详情
productpage	NodePort	Deployment: productpage-v1	10.96.29.120	9080:31686/TCP	集群内访问: productpage.sample:9080 集群外访问: 192.168.1.203:31686	2023-07-22 18:32:43	删除 刷新 查看详情
ratings	ClusterIP	Deployment: ratings-v1	10.96.73.176	9080/TCP	集群内访问: ratings.sample:9080	2023-07-22 18:32:43	删除 刷新 查看详情
reviews	ClusterIP	Deployment: reviews-v1	10.96.53.206	9080/TCP	集群内访问: reviews.sample:9080	2023-07-22 18:32:43	删除 刷新 查看详情

2.4 通过云原生网关访问 bookinfo 应用

前提条件：

1. 已开通 CCSE（容器云服务引擎），至少有一个 CCSE 集群实例
2. 开通天翼云服务网络实例
3. 开通天翼云微服务引擎，并在服务网格控制面集群同 VPC 内创建云原生网关实例

操作步骤：

我们在当前 CCSE 实例同 VPC 下预先开了一个云原生网关实例；通过云原生网关，我们可以从外部访问到 CCSE 内的服务，具体操作如下说明。

首先通过云原生网关实例内服务来源->创建来源，将当前 CCSE 集群添加为服务来源，如下：



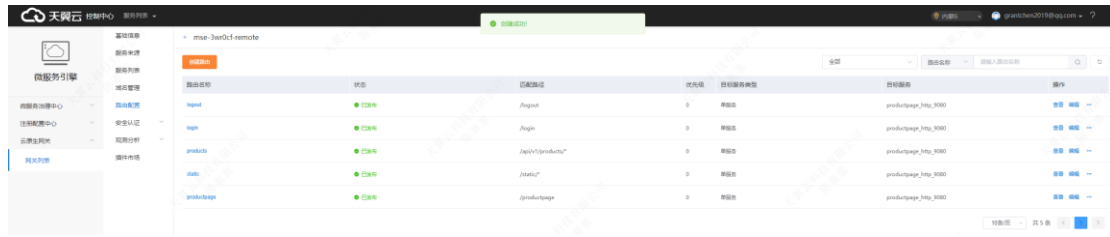
然后通过服务列表->创建服务，将 sample 命名空间下的 bookinfo 应用入口服务 productpage 添加到云原生网关内：



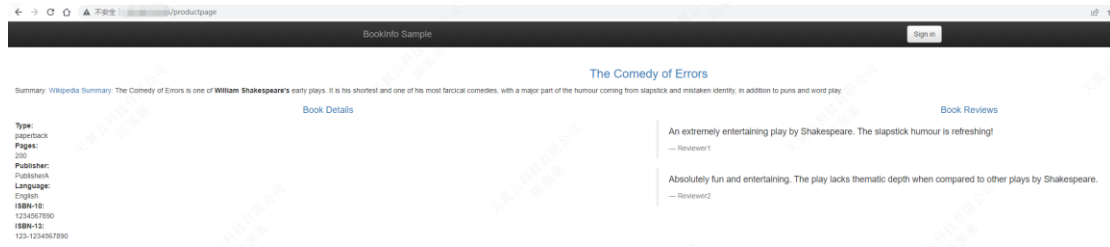
添加之后服务列表内可以看到 bookinfo 的 productpage 服务：

服务名称	服务地址	服务来源	请求协议	命名空间	最近更新时间	操作
productpage_http_9080	-	容器服务	http	sample	2023-09-05 09:36:38	删除 刷新 查看详情

在路由配置内，通过配置网关路由规则，将 bookinfo 的访问流量路由到 productpage 服务，需要添加的路由如下：



通过网绑定的 ELB 地址，我们现在就可以访问到 bookinfo 应用的首页了，如下图：



2.5 使用 istio 资源实现版本流量路由

前提条件：

1. 已开通 CCSE（容器云服务引擎），至少有一个 CCSE 集群实例
2. 开通天翼云服务网格实例
3. 开通天翼云微服务引擎，并在服务网格控制面集群同 VPC 内创建云原生网关实例

操作步骤：

这里以 bookinfo 应用里面的 reviews 服务为例，使用 istio 资源实现对 reviews 服务的多版本路由，首先到 CCSE 控制台部署 reviews 服务的 v2 和 v3 版本，yaml 如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reviews-v2
  labels:
    withServiceMesh: "true"
spec:
  replicas: 1
  selector:
    matchLabels:
      name: reviews-v2
  template:
    metadata:
      labels:
        app: reviews
        version: v2
        name: reviews-v2
        source: CCSE
        "sidecar.istio.io/inject": "true"
        csmAutoEnable: "on"
    annotations:
```

```
    "sidecar.istio.io/inject": "true"
spec:
  containers:
    - name: reviews
      image: 'registry-vpc-crs-huadong1.ctyun.cn/library/istio-examples-bookinfo-
reviews-v2:1.16.2'
      imagePullPolicy: IfNotPresent
      env:
        - name: LOG_DIR
          value: "/tmp/logs"
      volumeMounts:
        - name: tmp
          mountPath: /tmp
        - name: wlp-output
          mountPath: /opt/ibm/wlp/output
  volumes:
    - name: wlp-output
      emptyDir: {}
    - name: tmp
      emptyDir: {}
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reviews-v3
  labels:
    withServiceMesh: "true"
spec:
  replicas: 1
  selector:
    matchLabels:
      name: reviews-v3
  template:
    metadata:
      labels:
        app: reviews
        version: v3
        name: reviews-v3
        source: CCSE
        "sidecar.istio.io/inject": "true"
        csmAutoEnable: "on"
    annotations:
      "sidecar.istio.io/inject": "true"
  spec:
    containers:
      - name: reviews
        image: 'registry-vpc-crs-huadong1.ctyun.cn/library/istio-examples-bookinfo-
```



```

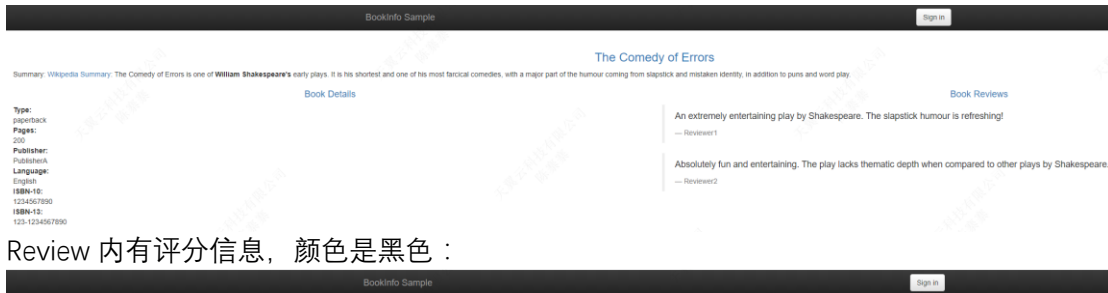
reviews-v3:1.16.2'
imagePullPolicy: IfNotPresent
env:
  - name: LOG_DIR
    value: "/tmp/logs"
volumeMounts:
  - name: tmp
    mountPath: /tmp
  - name: wlp-output
    mountPath: /opt/ibm/wlp/output
volumes:
  - name: wlp-output
    emptyDir: {}
  - name: tmp
    emptyDir: {}

```

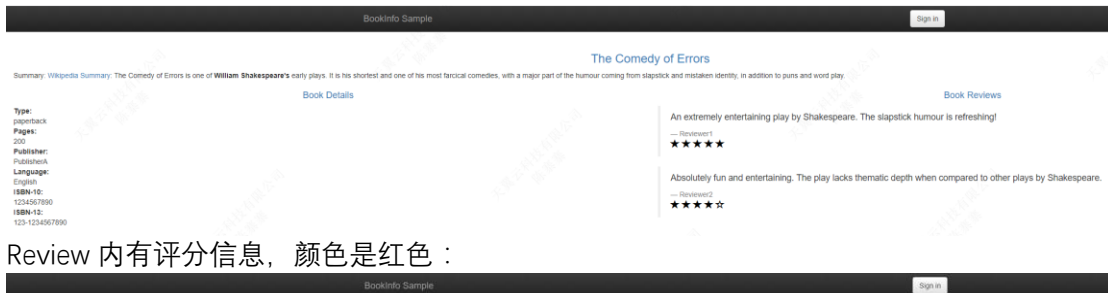
部署完成后，在 CCSE 控制台可以看到 reviews 服务有三个部署版本：

名称	类型	运行/期望Pod数量	镜像	创建时间	操作
reviews-v1	Deployment	3/3	registry-ops-mesh-ccscgpn.com/...	2023-07-22 18:57:10	去更新 重新部署 删除
reviews-v2	Deployment	3/3	docker.io/ibm/sample-helloworld...	2023-07-17 17:21:12	去更新 重新部署 删除
reviews-v3	Deployment	3/3	registry-ops-mesh-ccscgpn.com/...	2023-07-22 18:57:10	去更新 重新部署 删除
reviews-v1	Deployment	3/3	registry-ops-mesh-ccscgpn.com/...	2023-07-22 18:57:10	去更新 重新部署 删除
reviews-v2	Deployment	3/3	registry-ops-mesh-ccscgpn.com/...	2023-07-22 18:57:10	去更新 重新部署 删除
reviews-v3	Deployment	3/3	registry-ops-mesh-ccscgpn.com/...	2023-07-23 08:57:59	去更新 重新部署 删除
reviews-v2	Deployment	3/3	registry-ops-mesh-ccscgpn.com/...	2023-07-23 08:57:59	去更新 重新部署 删除

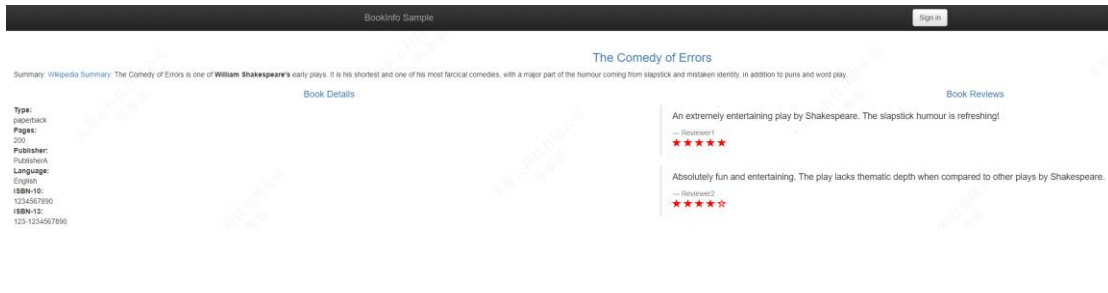
通过云原生网关持续访问 bookinfo 应用可以看到前端页面在三种效果内跳变，分别对应 reviews 服务的三个版本，如下图：
Review 内没有评分：



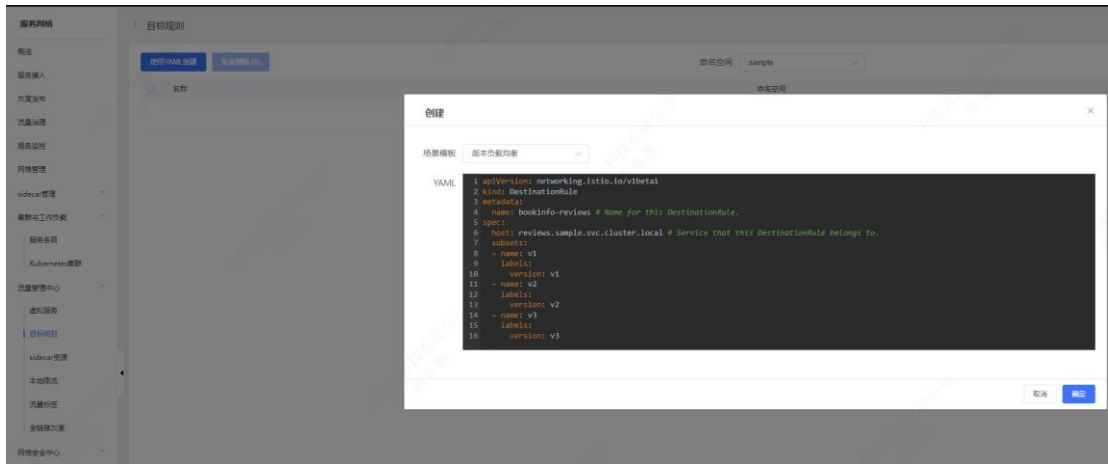
Review 内有评分信息，颜色是黑色：



Review 内有评分信息，颜色是红色：



部署完多个版本的服务之后，我们通过定义目标规则（DestinationRule）为 reviews 服务在服务网格内定义多个版本；在网格控制台进入流量管理中心->目标规则，选择 sample 命名空间，使用 yaml 创建，选择版本负载均衡模板，基于 reviews pod 的 version 标签为 reviews 服务定义三个版本，配置如下：



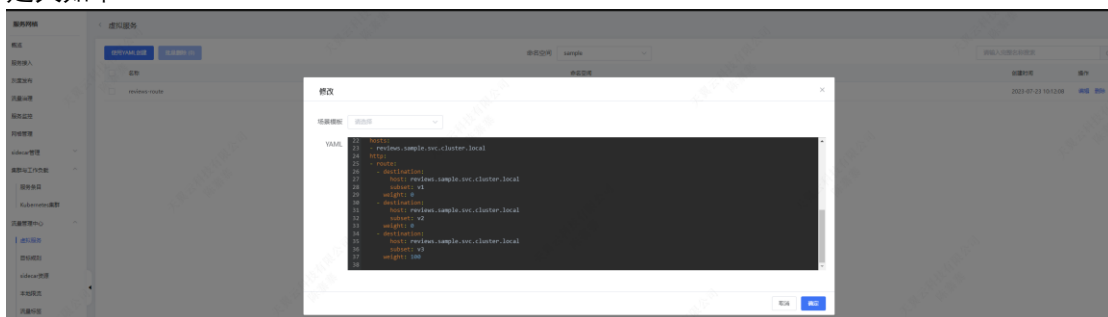
部分字段说明如下：

字段	说明
Metadata.name	目标规则的名称，namespace 内唯一
Spec.host	该目标规则匹配的服务名
Spec.subsets[].name	目标服务子集的名称
Spec.subsets[].labels	目标服务子集匹配的 pod 标签

配置完成后可以看到当前配置的目标规则列表：



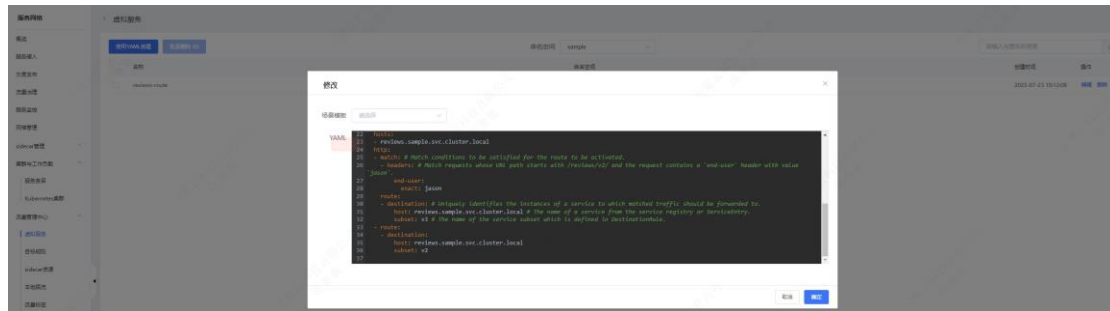
下一步，我们为通过虚拟服务（VirtualService）为 reviews 服务定义路由规则，比如定义只访问 v3 版本的 reviews 服务，可以在流量管理中心->虚拟服务，选择 sample 命名空间，使用 yaml 创建，基于 DestinationRule 定义的三个 subset，分配 100%流量到 v3 版本，具体定义如下：



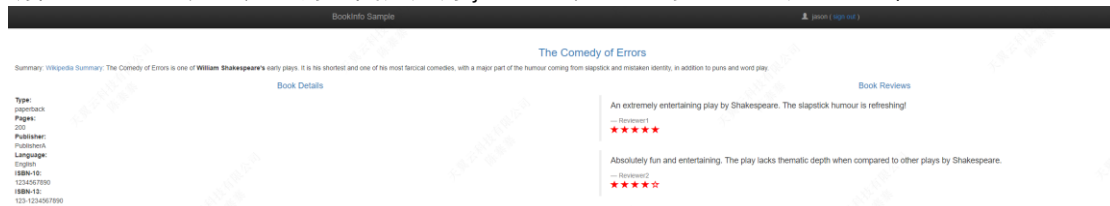
多次访问 bookinfo 应用可以看到前端样式无变化，reviews 部分是带打分，且是红色的：



除了按比例在多个版本随机分配流量之外，还可以按照 http 匹配规则实现自定义的流量路由策略，比如匹配指定头部的访问路由到 v3，其他请求路由到 v2，可以按照如下方式修改虚拟服务定义：



访问 bookinfo 页面，当前登陆用户为 jason 时，看到的 reviews 是 v3 版本：



切换到另外一个登陆用户 jack 时，看到的 reviews 是 v2 版本：



3. 产品功能

3.1 开通

从服务网格产品首页，选择【公测开通】即可进入网格订购页，如下图：

< 订购服务网格

* 实例名称

最长40字符，只能包含小写字母、数字及分隔符("-")，且必须以小写字母开头，数字或小写字母结尾

版本类型

可支撑Pod规模50个

istio版本

计费模式

控制面部署

CCSE集群

istio控制面ELB

可观测性

启用链路追踪

启用监控指标

启用访问日志采集

启用控制面日志采集

高级选项

nacos注册服务

服务网格订购参数说明如下：

参数	说明
实例名称	用于标识一个网格实例
版本类型	产品规格，分为基础版（支持 50pod），标准版（支持 1000pod），企业版（支持 10000pod）；当前支持基础版
istio 版本号	天翼云服务网格基于 istio 实现，当前支持的 istio 版本为 1.16.2
计费模式	当前支持按需计费模式
控制面部署	服务网格控制面可以部署在租户的 CCSE 实例上或者以托管的方式部署在独立的资源上，当前支持部署在租户 CCSE 的方式

CCSE 集群	选择控制面部署的 CCSE 集群，需要您提前开通 CCSE 实例
istio 控制面 ELB	用于暴露服务网格控制面的 ELB 实例规格
启用链路追踪	启用链路追踪后，可以采集网格内服务访问链路数据并上报到后端存储
采样率	链路追踪采样率
启用监控指标	启用监控指标采集后，会采集网格内服务访问的指标数据，可以通过监控指标页面查看
启用访问日志采集	启用访问日志采集后，网格内的 sidecar 访问日志会被采集到 ALS 日志服务，可以通过 ALS 日志服务查看到日志详情；前提是您已经开通了 ALS 日志服务；
访问日志项目名	可以将访问日志采集到当前已有日志项目或者新建日志项目，新建日志项目名称规则为：istiod_\${网格实例名称}_accesslog
启用控制面日志采集	启用控制面日志采集后，可以将服务网格控制面日志采集到 ALS 日志服务，可以通过 ALS 日志服务查看到控制面日志详情；前提是您已经开通了 ALS 日志服务；
控制面日志项目名	可以将控制面日志采集放到当前已有的日志项目或者新建日志项目，新建日志项目的规则为：istiod_\${网格实例名称}_controlepanel
Nacos 注册服务	启用 Nacos 注册服务后，服务网格会通过 Nacos 提供的 MCP over xDS 端口获取到 Nacos 内注册的服务，网格内的服务可以访问这些服务地址
Nacos 注册服务地址	选择当前已经开通的 Nacos 实例

一个网格实例创建一般需要 5 分钟左右时间。

3.2 退订

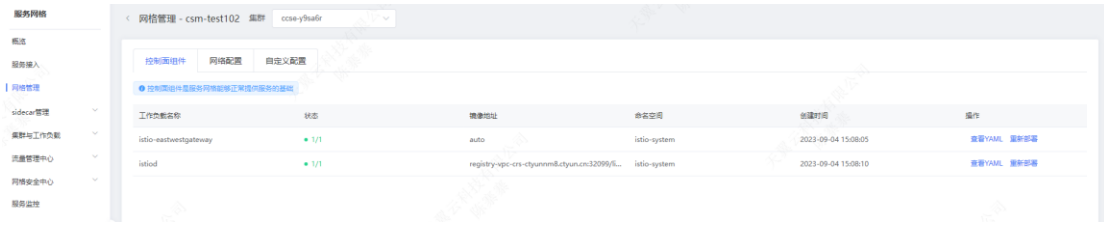
进入网格实例列表页可以看到当前已经开通的网格实例，点击实例右侧按钮，选择删除即可退订网格，如下图所示：



3.3 网格管理

3.3.1 控制面组件

该菜单下展示了控制面部署的组件信息，如下图：



组件说明如下：

组件	说明
istio-eastwestgateway	服务网格东西网关，用于暴露服务网格控制面服务
istiod	服务网格控制面服务

3.3.2 网络配置

网络配置页面展示了当前网格的基础配置，包括 istio configmap，sidecar 注入配置，sidecar 注入 webhook 配置，如下图：



您可以通过查看 yaml 了解网格当前的配置。

3.3.3 自定义配置

当前支持链路追踪、指标采集、控制面&数据面日志采集及 OPA 功能开关，如下图：



配置说明如下：

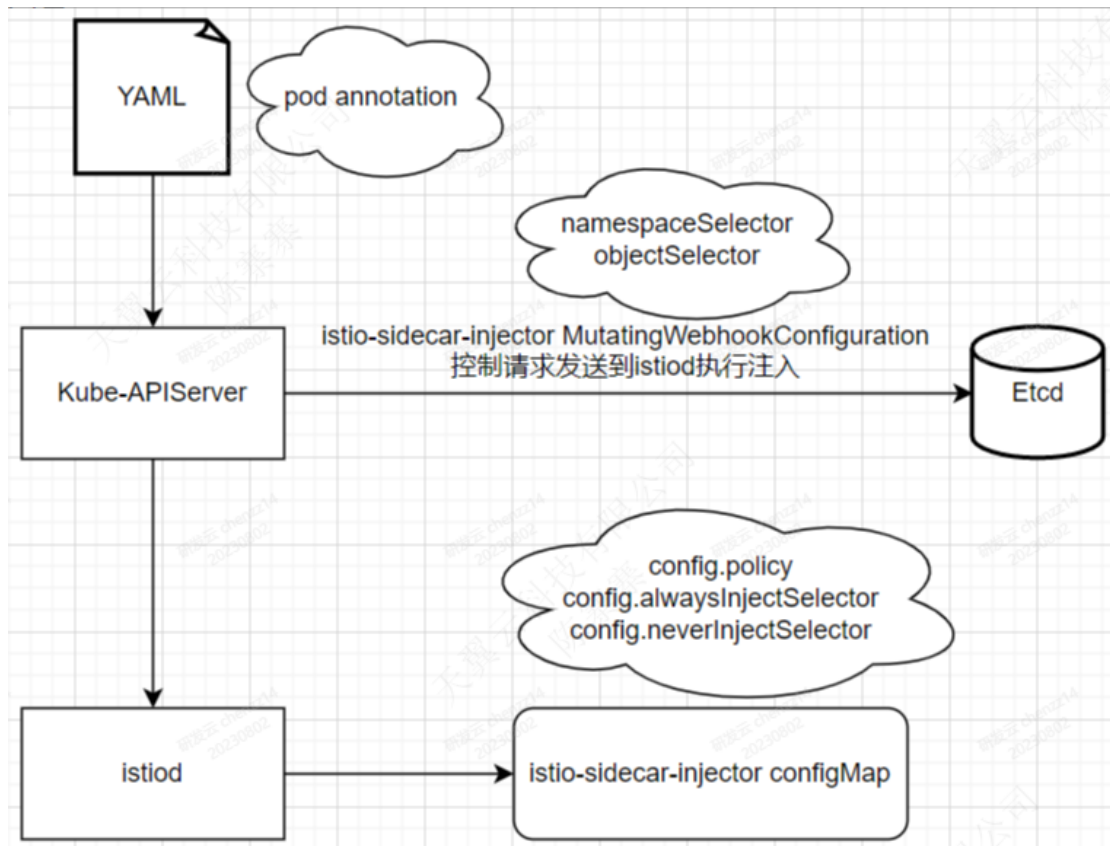
配置	说明
启用链路追踪	需要您提前开启 ARMS 监控服务，同时支持配置链路追踪采样率
开启 prometheus 指标监控	需要您提前开启 ARMS 监控服务
启用访问日志采集	您提前开启 ALS 日志服务，sidecar 访问日志将采集到指定的日志服务 project
启用控制面日志采集	您提前开启 ALS 日志服务，控制面日志将采集到指定的日志服务 project
启用 OPA	启用 OPA 功能后，将会在网格内安装 OPA 相关的控制面服务，后续可以在业务中集成 OPA 策略引擎

3.4 sidecar 管理

3.4.1 注入策略配置

CSM 服务网格基于 sidecar 技术实现业务无感知的流量治理能力，当前支持基于 K8s webhook 技术的 sidecar 注入能力，下图是 sidecar 注入过程：

- (1) pod 创建请求发送到 K8s api server 之后，根据预先定义的 mutatingwebhookconfiguration 与当前 pod 创建请求进行匹配，如果匹配则，K8s api server 向 istio 的控制面发送 webhook 请求
- (2) istio 控制面根据配置向 pod 的 yaml 里注入 sidecar 容器，返回给 api server
- (3) api server 基于 istio 控制面注入后的 yaml 创建 pod，从而实现了 sidecar 的注入



sidecar 注入策略受多种配置影响，有 pod 所在命名空间的标签、pod 注解、自动注入匹配、特殊匹配规则等，天翼云服务网格 CSM 控制台 sidecar 管理->注入策略配置菜单提供的注入策略配置能力，如下图所示：



配置说明如下：

配置	说明
Config.policy	是否开启自动注入功能，可选值为 enabled 或者 disabled，默认为 enabled

AlwaysInjectSelector	注入 sidecar 的 Pod 标签选择器，多个标签是 and 关系；被选中的 pod 总是会注入 sidecar
NeverInjectSelector	不注入 sidecar 的 Pod 标签选择器，多个标签是 and 关系；被选中的 pod 永远不注入 sidecar；优先级高于 AlwaysInjectSelector

Pod 是否会被注入 sidecar 还和 pod 所在命名空间以及 pod 自身的注解有关，这两种因子叠加上面三个配置参数，最终的注入策略生效逻辑如下：

命名空间标签 (istio-injection)	Pod 注解 (sidecar.istio.io/inject)	自动注入开关 (config.polic y)	NeverInjectSelector	AlwaysInjectSelector	结果
enabled	true	×	×	×	注入
enabled	false	×	×	×	不注入
enabled	未设置	×	匹配	×	不注入
enabled	未设置	enabled	不匹配	×	注入
enabled	未设置	disabled	×	不匹配	不注入
enabled	未设置	disabled	不匹配	匹配	注入
disabled	×	×	×	×	不注入

未设置	true	×	×	×	不注入
未设置	false	×	×	×	不注入
未设置	未设置	×	×	×	不注入

3.4.2 sidecar 代理配置

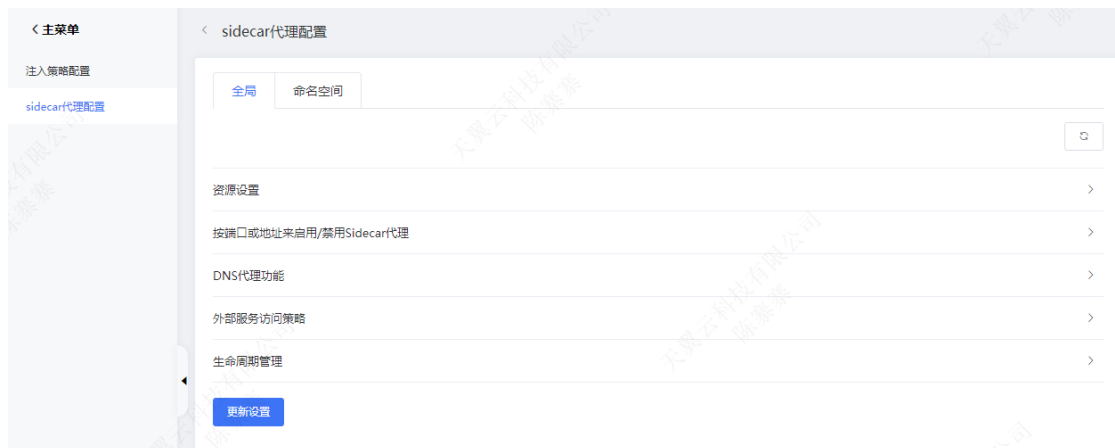
sidecar 注入 pod 之后会代理业务容器进出的流量，从而实现丰富的流量治理、可观测等能力；sidecar 本身作为一个代理服务器支持灵活的策略配置，包括代理的资源消耗限制、生命周期管理、流量拦截策略、外部访问策略等。

sidecar 代理配置生效范围及应用

sidecar 代理配置支持不同的生效范围，包括全局、命名空间、工作负载以及 pod 级，优先级依次升高；CSM 服务网格当前支持全局和命名空间级别的代理配置。全局代理配置将在全局生效，在没有其他配置的情况下，所有注入的 sidecar 都将采用这个配置；命名空间的代理配置将在指定的命名空间生效，该命名空间下注入的 sidecar，在没有指定 pod 自定义配置的情况下将全部采用该命名空间代理配置；您可以通过给 pod 打注解实现 pod 级别的代理配置。

sidecar 代理配置具体操作

在 sidecar 管理->sidecar 代理配置可以看到全局和命名空间代理配置，如下图所示：



配置项说明如下：

配置项	说明
istio 代理资源	sidecar 代理容器运行时最小需要使用的 CPU 和内存资源以及最大能申请到的 CPU 和内存资源
istio-init 容器资源	初始化容器运行时最小需要使用的 CPU 和内存资源以及最大能申请到的 CPU 和内存资源
拦截对外访问的地址范围	拦截对这些地址的 outbound 访问
不拦截对外访问的地址范围	不拦截对这部分地址的 outbound 访问
设置端口使入流量经过 sidecar 代理	拦截对这些端口的 inbound 访问
设置端口使出流量经过 sidecar 代理	拦截对这些端口的 outbound 访问
设置端口使入流量免于经过 sidecar 代理	不拦截对这些端口的 inbound 访问
设置端口使出流量免于经过 sidecar 代理	不拦截对这些端口的 outbound 访问
启用 DNS 代理功能	启用 DNS 代理功能后，sidecar 将拦截 DNS 解析请求，如果 sidecar 本地有 DNS 结果缓存，则直接返回结果；否则 sidecar 会将 DNS 请求转发出去
外部访问策略	定义了 sidecar 访问外部服务的策略，有两个选项，配置为 REGISTRY_ONLY 时，sidecar 只会访问网格内已注册的服务，拒绝访问外部服务；配置为 ALLOW_ANY 时，不管服务有没有注册到网格内，sidecar 都可以访问。您可以通过将外部服务注册为 ServiceEntry，在 REGISTRY_ONLY 模式下也可以访问到这些服务。
sidecar 代理终止等待时长	服务下线之后由于服务注册信息同步延迟以及部分存量请求仍在处理中等因素影响，不能直接停止服务，sidecar 作为中间的代理也不能立即停止服务，否则可能造成部分请求失败；该配置项定义了 sidecar

	容器在收到退出信号后继续等待的时长，默认为 5 秒，您可以根据自己业务的需求调整该配置。
sidecar 代理生命周期	<p>支持以 json 形式配置 sidecar 代理的生命周期，主要包括两个配置能力：</p> <p>postStart：sidecar 容器启动后执行的动作</p> <p>preStop：sidecar 代理容器停止前执行的动作</p> <p>例如下面的配置定义了 sidecar 容器启动后会等待 pilot agent 启动；sidecar 容器停止前会等待 10 秒</p> <pre> { "postStart": { "exec": { "command": ["pilot-agent", "wait"] } }, "preStop": { "exec": { "command": ["/bin/sh", "-c", "sleep 10"] } } } </pre>

对于 pod 级的 sidecar 代理配置，CSM 服务网格兼容 istio 自带的注解方式实现 sidecar 代理配置，注解说明如下表：

注解	说明
proxy.istio.io/config	覆盖 sidecar 代理配置
sidecar.istio.io/agentLogLevel	指定 pilot-agent 的日志级别
sidecar.istio.io/controlPlaneAuthPolicy	指定 sidecar 和控制面连接的认证方式，NONE 表示无认证，MUTUAL_TLS

	表示采用双向 TLS 认证
sidecar.istio.io/discoveryAddress	指定 sidecar xDS 连接地址
sidecar.istio.io/extraStatTags	从 sidecar 遥测数据中额外提取的标签
sidecar.istio.io/inject	是否注入 sidecar
sidecar.istio.io/interceptionMode	定义 sidecar 拦截流量的模式, REDIRECT 或者 TPROXY
sidecar.istio.io/logLevel	sidecar 代理的日志级别
sidecar.istio.io/proxyCPU	sidecar 请求的 CPU
sidecar.istio.io/proxyCPULimit	sidecar CPU 限制
sidecar.istio.io/proxyMemory	sidecar 请求的内存
sidecar.istio.io/proxyMemoryLimit	sidecar 内存限制
traffic.sidecar.istio.io/excludeInboundPorts	不拦截入流量方向的端口
traffic.sidecar.istio.io/excludeOutboundPorts	不拦截出流量方向的端口
traffic.sidecar.istio.io/includeOutboundIPRanges	拦截出流量方向的 ip 段
traffic.sidecar.istio.io/excludeOutboundIPRanges	不拦截出流量方向的 ip 段
traffic.sidecar.istio.io/includeInboundPorts	拦截入流量方向的端口
traffic.sidecar.istio.io/includeOutboundPorts	拦截出流量方向的端口

3.5 集群与工作负载

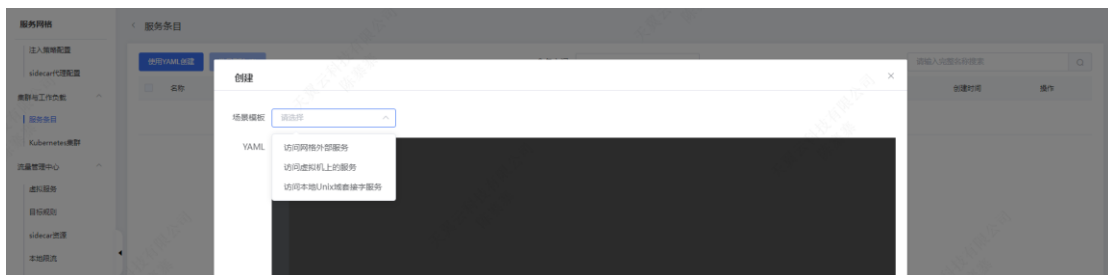
3.5.1 服务条目管理

在服务网格内部服务需要访问的外部的场景下，可以使用服务条目 (**ServiceEntry**) 定义外部服务的访问信息，这样网格内的服务也可以访问外部的服务。服务条目描述了服务的域名、协议、端口、节点列表等信息，

定义服务条目时，需要注意端口不要与 **sidecar** 的端口冲突，下面的表格梳理了当前 **sidecar** 占用的端口：

端口	协议	说明
15000	TCP	sidecar 管理端口
15001	TCP	sidecar 出口代理端口
15006	TCP	sidecar 入口代理端口
15020	HTTP	Prometheus 指标采集端口
15021	HTTP	监控检查端口
15090	HTTP	Prometheus 指标采集端口

通过集群与工作负载菜单下的服务条目可以管理网格内的外部服务；当前提供了三种模板用于创建服务条目，如下图：



访问网格外部的 `www.ctyun.cn` 服务可以按照下面的配置：

```
apiVersion: networking.istio.io/v1beta1
```

```
kind: ServiceEntry
```

```
metadata:
```

```
  name: external-svc-https # Name for this ServiceEntry.
```

```
spec:
```

```
  hosts: # External services that mesh can access.
```

```
  - www.ctyun.cn
```

```
  location: MESH_EXTERNAL # Location can be MESH_EXTERNAL or MESH_INTERNAL.
```

```
  ports: #
```

```
  - number: 443
```

```
    name: https
```

protocol: TLS # Access external services over TLS.

resolution: DNS # Resolve external services' IP address by DNS.

如果我们在虚机上部署了一组 MongoDB 的实例，但是并没有注册到网络上，可以按照如下的配置在网格内访问 MongoDB：

apiVersion: networking.istio.io/v1beta1

kind: ServiceEntry

metadata:

name: external-svc-mongocluster # Name for this ServiceEntry.

spec:

hosts:

- mymongodb.somedomain # Not used.

addresses:

- 192.192.192.192/24 # VIPs

ports:

- number: 27018

name: mongodb

protocol: MONGO

location: MESH_EXTERNAL # Location can be MESH_EXTERNAL or MESH_INTERNAL.

resolution: STATIC # Access service by endpoints' static IP address.

endpoints:

- address: 2.2.2.2

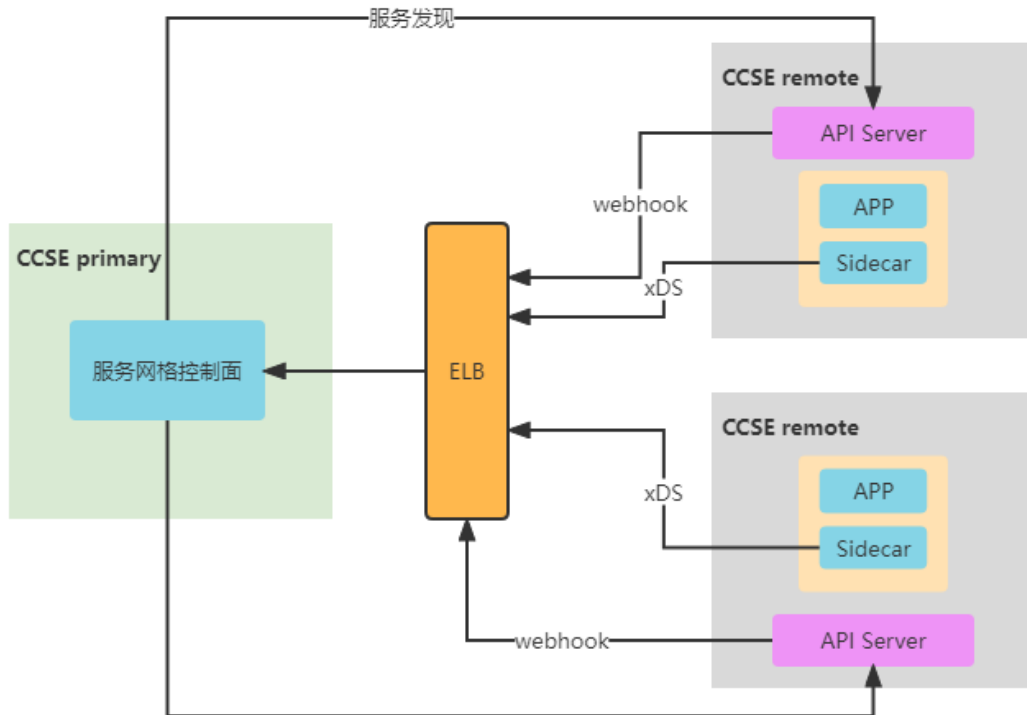
- address: 3.3.3.3

注意：

服务条目里面的端口协议必须是 HTTP|HTTPS|GRPC|HTTP2|MONGO|TCP|TLS 里面的值。

3.5.2 Kubernetes 集群管理

CSM 服务网格支持多集群架构，在开通网格实例时选择的 CCSE 集群我们称为主集群 (primary)，服务网格的控制面会部署在主集群上，后续可以添加同 VPC 下的其他 CCSE 集群到网格内，我们称为从集群 (remote)，整体架构如下图：



添加集群

选择 集群与工作负载 -> kubernetes 集群 菜单，可以看到当前网格内的集群列表：



在添加从集群之前，首先确定主集群控制面是否已经完成暴露（可以被从集群连接、使用），选择右侧 查看集群状态选项：



如果在开通网格之后超过半小时控制面服务暴露还未完成，请联系客服解决。

当前已经完成主集群控制面暴露，可以通过左上角的 添加从集群 按钮添加同 vpc 内的其他集群。

删除集群

当前只支持删除从集群，选择 集群与工作负载 -> kubernetes 集群 菜单，可以看到当前网格内的集群列表，在从集群右侧操作栏可以看到删除选项，点击删除即可。

多集群管理部署示例

前置条件

1. 已开通服务网格实例，控制面部署在主集群，我们称为 C1
2. 在网格中添加另外一个集群，我们称为 C2

部署应用

1. 在集群 C1 中创建命名空间 bookinfo

```
kubectl create ns bookinfo
```

```
kubectl label ns bookinfo istio-injection=enabled
```

部署 bookinfo 应用的 productpage、details、rating 以及 reviews（版本 v1）服务（yaml 参考快速入门）

部署完成后，C1 集群 pod 如下：

```
root@ccse-vmnqatbh-1# kubectl get po -n bookinfo -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE                NOMINATED NODE   READINESS GATES
details-v1-78fbd6c4d5-bhx84         2/2     Running   0           30m   192.168.66.193  ccse-vm6dvlval     <none>            <none>
productpage-v1-7d78657c55-j4t42     2/2     Running   0           30m   192.168.64.229  ccse-vm6dvlval     <none>            <none>
ratings-v1-7c46556d89-tlxj2        2/2     Running   0           30m   192.168.67.122  ccse-vm6dvlval     <none>            <none>
reviews-v1-6f56dbfc6b-nm9nl         2/2     Running   0           30m   192.168.65.218  ccse-vm6dvlval     <none>            <none>
sleep-7fb478946b-44t5x             2/2     Running   0           9m3s  192.168.65.206  ccse-vm6dvlval     <none>            <none>
```

2. 在集群 C2 中创建命名空间 bookinfo

```
kubectl create ns bookinfo
```

```
kubectl label ns bookinfo istio-injection=enabled
```

部署 reviews 服务的 v2 和 v3 版本（yaml 参考快速入门）

部署完成后，C2 集群 pod 如下：

```
root@ccse-vmchuanvtx-1# kubectl get po -n bookinfo -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE                NOMINATED NODE   READINESS GATES
reviews-v2-7d9b5b44b7-j64s9         2/2     Running   0           27m   192.168.128.173  ccse-vmrhqch6qo    <none>            <none>
reviews-v3-b96c79f68-kq2w1         2/2     Running   0           27m   192.168.128.231  ccse-vmrhqch6qo    <none>            <none>
```

3. 在 C1 中部署 sleep 应用，验证访问

```
apiVersion: v1
```

```
kind: ServiceAccount
```

metadata:

name: sleep

apiVersion: v1

kind: Service

metadata:

name: sleep

labels:

app: sleep

service: sleep

spec:

ports:

- port: 80

name: http

selector:

app: sleep

apiVersion: apps/v1

kind: Deployment

metadata:

name: sleep

spec:

replicas: 1

selector:

matchLabels:

app: sleep

template:

metadata:

labels:

app: sleep

```

spec:
  terminationGracePeriodSeconds: 0
  serviceAccountName: sleep

  containers:
  - name: sleep
    image: registry-vpc-crs-huadong1.ctyun.cn/library/curl
    command: ["/bin/sleep", "infinity"]
    imagePullPolicy: IfNotPresent

  volumeMounts:
  - mountPath: /etc/sleep/tls
    name: secret-volume

  volumes:
  - name: secret-volume

  secret:
    secretName: sleep-secret
    optional: true

---

```

通过 sleep 应用多次访问 productpage 服务：

```
kubectl exec -it -n bookinfo sleep-7fb478946b-44t5x -c istio-proxy -- curl
http://productpage:9080/productpage -svo/dev/null
```

4. 查看 productpage sidecar 日志：

```
kubectl logs productpage-v1-7d78657c55-j4t42 -n bookinfo -c istio-proxy | grep
reviews
```

可以看到 productpage 服务访问到了 reviews 服务的三个版本的 pod

5. 配置 DestinationRule 、 VirtualService 资源，限制只能访问 reviews 的 v3 版本

```

apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews

```

```
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v3
    labels:
      version: v3
version: v3
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v3
weight: 100
```

再次多次访问 productpage 服务，查看 productpage sidecar 日志如下，可以看到总是访问到 reviews 的 v3 版本：

```
192.168.64.229:50584 10.96.174.231:9080 192.168.64.229:45460 - - 0 425 263 263 "-" curl/7.81.0 "f13011fb-6244-4bf7-b028-0e74cc7c44c5" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
cluster.local 192.168.64.229:50574 10.96.174.231:9080 192.168.64.229:45460 - - 0 425 43 43 "-" curl/7.81.0 "bae3e4e1-d06c-4b05-82d7-e792b3310cfc" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
2023-10-18T07:42:12.4202Z GET /reviews/0 HTTP/1.1 200 via_upstream - - 0 425 75 75 "-" curl/7.81.0 "3f0eb2ec-b60a-4f40-a5d9-a9958bed88b5" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
cluster.local 192.168.64.229:50574 10.96.174.231:9080 192.168.64.229:45472 - - 0 425 50 50 "-" curl/7.81.0 "299a7ced-7ab9-4399-a1e1-57473c711f13" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
2023-10-18T07:42:13.5332Z GET /reviews/0 HTTP/1.1 200 via_upstream - - 0 425 36 35 "-" curl/7.81.0 "f8b4f3c7-38cb-4bf8-83cd-6cd00219e215" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
cluster.local 192.168.64.229:50584 10.96.174.231:9080 192.168.64.229:45502 - - 0 425 29 29 "-" curl/7.81.0 "79ed484e-1f07-4841-9006-378fa77fcfb" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
cluster.local 192.168.64.229:50574 10.96.174.231:9080 192.168.64.229:46710 - - 0 425 39 38 "-" curl/7.81.0 "e75797da-656e-46f1-a5f6-23a9441b9c9a" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
2023-10-18T07:42:13.6422Z GET /reviews/0 HTTP/1.1 200 via_upstream - - 0 425 34 34 "-" curl/7.81.0 "c542a4c5-1bde-4d4a-a866-f889ae231ec2" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
cluster.local 192.168.64.229:50584 10.96.174.231:9080 192.168.64.229:48724 - - 0 425 36 36 "-" curl/7.81.0 "b944d250-2222-4154-9435-aa2a2220cbb8" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
cluster.local 192.168.64.229:50574 10.96.174.231:9080 192.168.64.229:48794 - - 0 425 35 34 "-" curl/7.81.0 "d35fc1b3-cc40-4f6a-8485-1aaa4fb05920" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
2023-10-18T07:42:13.6922Z GET /reviews/0 HTTP/1.1 200 via_upstream - - 0 425 126 125 "-" curl/7.81.0 "b8a90c8e-2a90-966f-bfaf-89eaa17edbd8" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
cluster.local 192.168.64.229:50584 10.96.174.231:9080 192.168.64.229:48748 - - 0 425 40 39 "-" curl/7.81.0 "9cb8529-2ba1-4cd1-81cd-9e60cc9911e5" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
cluster.local 192.168.64.229:50584 10.96.174.231:9080 192.168.64.229:48750 - - 0 425 40 39 "-" curl/7.81.0 "9cb8529-2ba1-4cd1-81cd-9e60cc9911e5" "reviews:9080" 192.168.128.231:9080 outbound|9080|v3|reviews.bookinfo.svc.c
```

3.6 流量管理中心

3.6.1 虚拟服务

虚拟服务 (VirtualService) 是服务网格的关键资源；虚拟服务定义了一组路由规则，并与请求匹配，根据匹配的结果将流量路由到响应的目标服务。本文介绍虚拟服务的基本管理以及 CRD 说明。

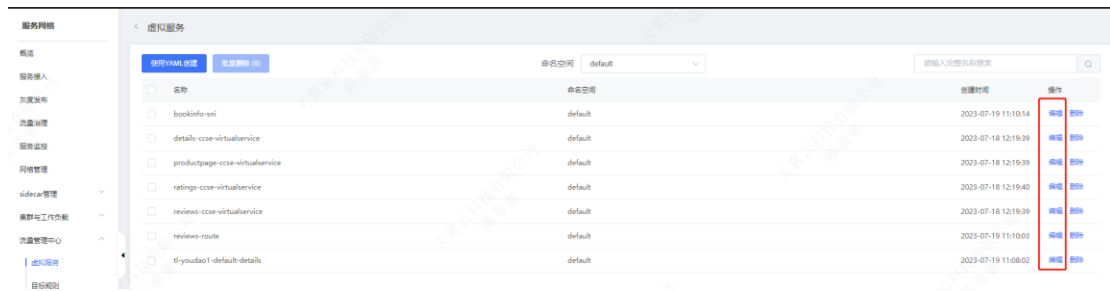
创建虚拟服务

1. 进入网格实例页，选择菜单栏 流量管理中心 -> 虚拟服务
2. 确认虚拟服务所在命名空间，选择 使用 yamll 创建
3. 当前已经定义了一些路由模板，可以选择对应模板，按照自己的路由需求进行修改，然后提交



修改虚拟服务

1. 进入服务网格实例页，选择菜单栏 流量管理中心 -> 虚拟服务
2. 选择相应的命名空间，列表页会展示当前命名空间下所有的虚拟服务定义
3. 选择操作栏下的编辑选项，可以对已经创建的虚拟服务进行编辑修改



删除虚拟服务

1. 进入服务网格实例页，选择菜单栏 流量管理中心 -> 虚拟服务
2. 选择相应的命名空间，列表页会展示当前命名空间下所有的虚拟服务定义
3. 选择操作栏下的删除选项，可以删除已经创建的虚拟服务



虚拟服务配置资源配置示例及关键字段说明

下面的虚拟服务配置将对 reviews 服务的访问默认转发到 reviews 服务的 v1 版本，如果 uri 匹配到/test 前缀，则将请求转发到 reviews 服务的 v2 版本；reviews 服务的两个版本使用另外一个目标规则定义。

```
apiVersion: networking.istio.io/v1beta1
```

```
kind: VirtualService
```

```
metadata:
```

```
  name: reviews-route
```

```
spec:
```

```
  hosts:
```

```
    - reviews.prod.svc.cluster.local
```

```
  http:
```

```
    - name: "reviews-v2-routes"
```

```
      match:
```

```
        - uri:
```

```

    prefix: "/test"
  route:
  - destination:
      host: reviews.prod.svc.cluster.local
      subset: v2
  - name: "reviews-v1-route"
    route:
  - destination:
      host: reviews.prod.svc.cluster.local
      subset: v1

```

关联的目标规则配置中定义了 reviews 服务的两个子集，分别对应 v1 和 v2 版本：

```

apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: reviews-destination
spec:
  host: reviews.prod.svc.cluster.local
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2

```

VirtualService 字段说明

字段	类型	必选	说明
hosts	[]string	否	流量转发规则匹配的主机名称，可以是 DNS 域名（支持前缀通配匹配）、ip 或者在 K8s 里可以是服务

			的名称（推荐使用 FQDN）。Hosts 字段适用于 tcp 和 http 流量路由规则，对于引用网格内部服务的规则，hosts 字段必须是域名的形式，ip 形式的 hosts 只能用到 Gateway 资源对象中
gateways	[]string	否	当前路由规则绑定的 Gateway 对象列表，可以是 gateway-ns/gateway-name 的形式，如果不加命名空间前缀，默认引用当前命名空间下的网关
http	[]HTTPRoute	否	针对 http 流量的路由规则列表，适用于 HTTP/HTTP2/gRPC 协议
tls	[]TLSRoute	否	针对 TLS 和 HTTPS 协议的路由规则列表，通常基于 TLS 连接过程中的 sni 信息进行路由
tcp	[]TCPRoute	否	针对非 HTTP 和 TLS 协议的所有请求的路由规则
exportTo	[]string	否	定义了当前虚拟服务对哪些命名空间暴露，用于实现可见性控制；不定义的话默认对所有命名空间可见

HTTPRoute 定义了对 HTTP1.1、HTTP2、gRPC 的路由规则，字段说明如下：

字段	类型	必选	说明
name	string	No	路由名称，会被记录到访问日志中，主要用于定位问题用
match	[]HTTPMatchRequest	No	路由匹配规则列表，一个 match 下面的多个匹配规则之间是 and 关系，必须同时满足；match 之间是 or 的关系
route	[]HTTPRouteDestination	No	路由配置，可以直接返回结果、重定向请求或者将请求转发到其他服务
redirect	HTTPRedirect	No	用于返回 301 重定向
directResponse	HTTPDirectResponse	No	用于返回固定的响应，在 route 和 redirect 为空的时候可以配置该字段

delegate	Delegate	No	指定用于委托 HTTPRoute 的虚拟服务，当 route 和 redirect 为空时可以设置，委托的虚拟服务会和当前规则合并 注意：1. 当前只支持一层委托 2. 委托的 HTTPMatchRequest 必须是根的严格子集，否则会有冲突，HTTPRoute 将不会生效
rewrite	HTTPRewrite	No	重写 uri 或者 authority 头部，不能与 redirect 一起使用；重写发生在转发请求之前
timeout	Duration	No	http 请求的超时时间
retries	HTTPRetry	No	http 请求重试策略
fault	HTTPFaultInjection	No	客户端的故障注入策略（重试和超时策略将不生效）
mirror	Destination	No	将流量镜像一份转发到指定的目标服务；该行为遵循尽最大努力原则，sidecar 将不等待镜像结果返回
mirrorPercentage	Percent	No	和 mirror 字段配合使用，定义镜像流量比例，默认为 100%
corsPolicy	CorsPolicy	No	跨域策略配置
headers	Headers	No	Header 操作策略

TLSRoute 定义了对 TLS 和 HTTPS 流量的路由规则，主要基于 SNI 路由，配置参数：

字段	类型	必选	说明
match	[]TLSMatchAttributes	Yes	TLS 匹配规则，一个 match 内的多个规则是 and 关系，多个 match 之间是 or

			的关系
route	[]RouteDestination	No	流量转发目标

TCPRoute 描述了对 TCP 流量的匹配和转发规则，主要是基于端口的流量转发，配置字段包括：

字段	类型	必选	说明
match	[]L4MatchAttributes	Yes	TCP 流量匹配规则，一个 match 内的多个规则是 and 关系，多个 match 之间是 or 的关系
route	[]RouteDestination	No	流量转发目标

HTTPMatchRequest

HTTPMatchRequest 配置定义了对 http 请求的一系列的匹配规则，具体字段如下：

字段	类型	必选	说明
name	string	No	标识一个匹配规则，会被记录到访问日志中，主要用于 debug
uri	StringMatch	No	Uri 的匹配规则，当前支持三种匹配，分别是精确匹配、前缀匹配和正则匹配；uri 匹配支持大小写敏感或不敏感匹配，可以通过 ignore_uri_case 字段配置
scheme	StringMatch	No	Uri 的 scheme 匹配，支持精确、前缀和正则匹配；大小写敏感
method	StringMatch	No	HTTP 方法匹配，支持精确、前缀和正则匹配；大小写敏感
authority	StringMatch	No	HTTP Authority 匹配，支持精确、前缀和正则匹配；大小写敏感

headers	map<string, StringMatch>	No	HTTP 头部匹配，针对每个头部支持定义匹配规则，支持精确、前缀和正则匹配；大小写敏感；uri, scheme, method, authority 几个头部的匹配会被忽略
port	uint32	No	目标主机的服务端口
sourceLabels	map<string, string>	No	请求源工作负载应当具备的标签
gateways	[]string	No	当前匹配规则适用的网关列表，会覆盖虚拟服务级别的网关列表
queryParams	map<string, StringMatch>	No	请求参数匹配，针对每个头部支持定义匹配规则，支持精确、前缀和正则匹配；大小写敏感
ignoreUriCase	bool	No	Uri 匹配是否需要大小写敏感
withoutHeaders	map<string, StringMatch>	No	和 headers 一样是对头部的匹配，匹配的效果刚好相反：如果 header 匹配了，则该请求不会应用该规则的处理
sourceNamespace	string	No	限制请求源工作负载所在的命名空间
statPrefix	string	No	相关监控指标的前缀

HTTPRouteDestination 定义了 HTTP 路由转发规则的目标服务

字段	类型	必选	说明
destination	Destination	Yes	要转发的目标服务信息
weight	int32	No	当前目标服务的权重
headers	Headers	No	头部操作规则

RouteDestination 定义了四层转发的目标服务

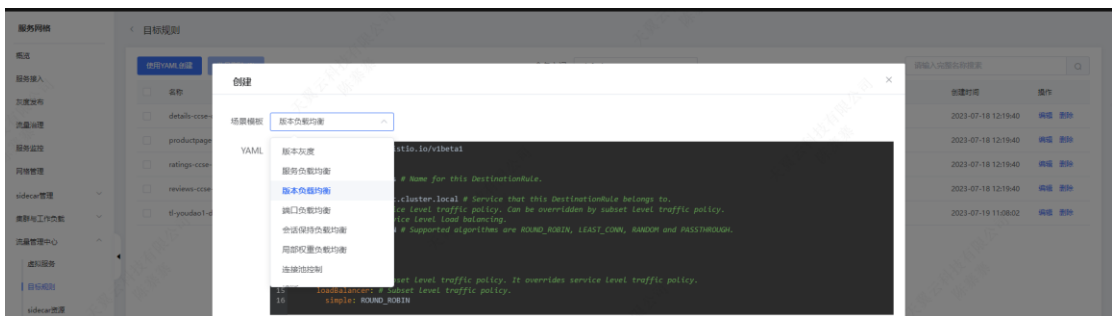
字段	类型	必选	说明
destination	Destination	Yes	要转发的目标服务信息
weight	int32	No	当前目标服务的权重

3.6.2 目标规则

目标规则（DestinationRule）定义了服务分组以及请求选定了路由之后，sidecar 对上游服务请求的行为，例如负载均衡、连接池大小、故障节点剔除等策略；本文说明如何创建、修改、删除目标规则

创建目标规则

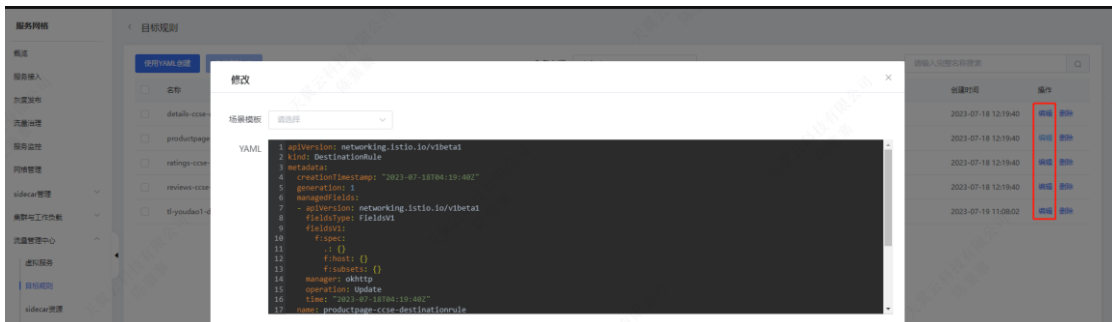
1. 登陆服务网格控制台，选择流量管理中心 -> 目标规则 菜单，选择相应的命名空间
2. 页面左上角选择使用 YAML 创建，当前提供了不同的场景配置模板



3. 可以基于模板修改，或者根据自己的需求编辑配置

修改目标规则

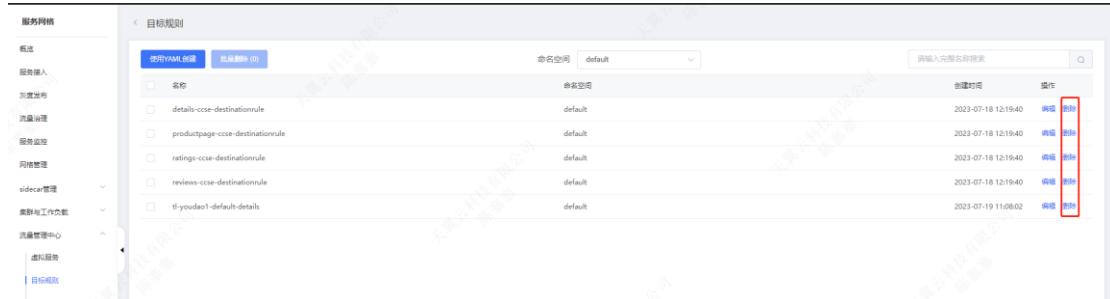
1. 进入服务网格控制台，选择流量管理中心 -> 目标规则菜单，选择相应的命名空间
2. 页面会展示当前所选命名空间下的目标规则列表，右侧操作栏选择编辑操作



3. 修改完成后提交即可

删除目标规则

1. 进入服务网格控制台，选择流量管理中心 -> 目标规则菜单，选择相应的命名空间
2. 页面会展示当前所选命名空间下的目标规则列表，右侧操作栏选择删除操作



目标规则资源配置示例及关键字段说明

下面目标规则定义了 bookinfo ratings 服务的默认负载均衡策略为 LEAST_REQUEST，同时定义了一个版本子集 testversion，对应服务的标签是 version: v3；对于 testversion 版本定义了独立的负载均衡策略是 ROUND_ROBIN

```
apiVersion: networking.istio.io/v1beta1
```

```
kind: DestinationRule
```

```
metadata:
```

```
  name: bookinfo-ratings
```

```
spec:
```

```
  host: ratings.prod.svc.cluster.local
```

```
  trafficPolicy:
```

```
    loadBalancer:
```

```
      simple: LEAST_REQUEST
```

```
  subsets:
```

```
    - name: testversion
```

```
      labels:
```

```
        version: v3
```

```
      trafficPolicy:
```

```
        loadBalancer:
```

simple: ROUND_ROBIN

上面的例子可以看到目标规则可以定义服务的负载均衡策略、版本分组策略等，具体的配置说明如下：

DestinationRule

字段	类型	必选	说明
host	string	Yes	该目标规则应用的服务名称
trafficPolicy	TrafficPolicy	No	流量策略（负载均衡、连接池、故障节点剔除等）
subsets	[]Subset	No	服务子集（流量策略可以在服务子集粒度定义，并覆盖全局策略）
exportTo	[]string	No	定义了当前目标规则暴露给哪些命名空间，用于控制目标规则的可见性，默认所有命名空间可见
workloadSelector	WorkloadSelector	No	选择当前目标规则应用的工作负载

TrafficPolicy

定义了请求后端服务具体的流量策略（服务级或者端口级策略）

字段	类型	必选	说明
loadBalancer	LoadBalancerSettings	No	负载均衡算法
connectionPool	ConnectionPoolSettings	No	对上游服务的连接池配置
outlierDetection	OutlierDetection	No	故障节点剔除策略
tls	ClientTLSSettings	No	请求上游服务的 tls 策略
portLevelSettings	[]PortTrafficPolicy	No	端口级流量策略
tunnel	TunnelSettings	No	请求目标服务的隧道配置；只能应用在 TCP 和 TLS 路由场景，不能用于 HTTP

Subset

服务子集通过标签选择器在服务内选定一个子集，目标规则内可以定义多个子集；通过再虚拟服务中根据匹配规则引用不同的子集，可以实现 AB 实验、灰度发布等功能。子集内可以定义独立的流量策略（覆盖服务级策略），子集配置说明如下：

字段	类型	必选	说明
name	string	Yes	子集名称，在虚拟服务里通过名称引用指定子集
labels	map<string, string>	No	子集标签过滤规则，匹配这些标签的服务节点才会被纳入当前子集
trafficPolicy	TrafficPolicy	No	子集的流量策略集成自服务级策略，子集内定义的流量策略将覆盖服务级策略

LoadBalancerSettings

定义了请求后端服务的负载均衡策略，字段说明如下：

字段	类型	必选	说明
simple	SimpleLB（枚举）	No	可选值为 UNSPECIFIED, RANDOM, PASSTHROUGH, ROUND_ROBIN, LEAST_REQUEST, LEAST_CONN
consistentHash	ConsistentHashLB（枚举）	No	基于 HTTP 头部、Cookie 等属性提供亲和性负载均衡配置
localityLbSetting	LocalityLoadBalancerSetting	No	基于请求源和目标的地域的调度策略
warmupDurationSecs	Duration	No	服务预热时间配置，用于新的节点加入并且需要预热的场景；当前只支持 ROUND_ROBIN 和 LEAST_REQUEST 负载均衡策

			略
--	--	--	---

ConnectionPoolSettings

对上游服务的连接池设置，应用在上游服务的每个节点；配置说明如下：

字段	类型	必选	说明
tcp	TCPSettings	No	定义了 TCP 的最大连接数，连接超时时间，TCPKeepAlive 等配置
http	HTTPSettings	No	定义了 HTTP 连接池相关配置

OutlierDetection

该配置实现了剔除上游故障节点（熔断）的功能，支持对 TCP 和 HTTP 服务进行熔断，主要基于连接超时、失败，HTTP 状态码等统计失败情况进行熔断，具体配置如下：

字段	类型	必选	说明
splitExternalLocalOriginErrors	bool	No	是否将本次检测到的错误纳入熔断统计，默认为 false；设置为 true 时，consecutive_local_origin_failure 值将被纳入错误统计
consecutiveLocalOriginFailures	UInt32Value	No	剔除异常节点前本地检测到的连续错误次数
consecutiveGatewayErrors	UInt32Value	No	剔除异常节点前检测到的连续网关错误次数（HTTP 502、503、504）
consecutive5xxErrors	UInt32Value	No	剔除异常节点前检测到的连续 5XX 次数
interval	Duration	No	异常情况统计时间窗口（必须大于 1ms，默认为 10s）
baseEjectionTime	Duration	No	故障节点被剔除的最短时间，默认为 30s

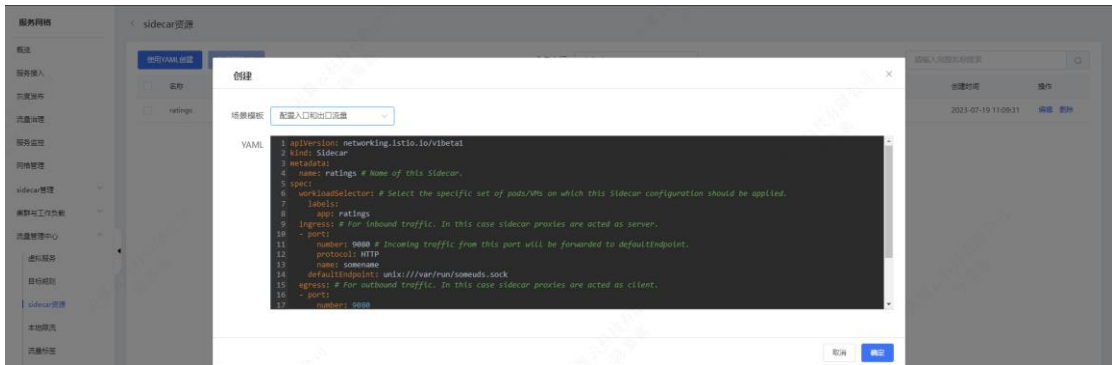
maxEjectionPercent	int32	No	上游所有节点中最大可以剔除多少比例的节点
minHealthPercent	int32	No	故障节点剔除策略生效的最低健康节点比例，监控节点比例低于该值时，故障剔除策略将失效

3.6.3 Sidecar 资源

默认情况下，服务网格内的 **sidecar** 可以访问到网格内的所有服务，也可以被网格内所有其他服务访问到。Sidecar 资源可以实现对网格内的 **sidecar** 更细粒度的配置，包括开放的端口、协议，可以访问的服务等。

Sidecar 资源创建

1. 登陆服务网格控制台，选择流量管理中心 -> Sidecar 资源 菜单，选择相应的命名空间
2. 选择使用 YAML 创建，目前提供了配置模板，可以基于模板修改或者自己编辑配置



Sidecar 资源修改

1. 登陆服务网格控制台，选择流量管理中心 -> Sidecar 资源 菜单，选择相应的命名空间
2. 在 Sidecar 资源列表右侧操作栏可以看到编辑按钮，可以对已创建的 Sidecar 资源进行编辑修改

Sidecar 资源删除

1. 登陆服务网格控制台，选择流量管理中心 -> Sidecar 资源 菜单，选择相应的命名空间
2. 在 Sidecar 资源列表右侧操作栏可以看到删除按钮，可以删除已创建的 Sidecar 资源

下面的例子在 prod-us1 命名空间下定义了名为 default 的 Sidecar 配置，对于被选中的 **sidecar**，只允许访问 prod-us1、prod-apis 和 istio-system 命名空间下的服务

```

apiVersion: networking.istio.io/v1beta1

kind: Sidecar

metadata:
  name: default
  namespace: prod-us1

spec:
  egress:
    - hosts:
      - "prod-us1/*"
      - "prod-apis/*"
      - "istio-system/*"
  
```

Sidecar 配置说明如下：

字段	类型	必选	说明
workloadSelector	WorkloadSelector	No	Sidecar 资源应用到 pod 的选择器
ingress	[]IstioIngressListener	No	sidecar 入流量配置
egress	[]IstioEgressListener	No	sidecar 出流量配置
outboundTrafficPolicy	OutboundTrafficPolicy	No	外部访问策略，可选项是只允许访问网格注册中心的服务 (REGISTRY_ONLY) 或者允许访问所有服务 (ALLOW_ANY)

IstioIngressListener

定义 **sidecar** 入流量监听器的配置，具体配置项如下：

字段	类型	必选	说明
----	----	----	----

port	Port	Yes	监听器端口配置
bind	string	No	监听器绑定的 ip
captureMode	CaptureMode	No	监听器流量拦截策略
defaultEndpoint	string	Yes	流量转发的默认目标地址，可以是 ip 端口或者 Unix domain socket
tls	ServerTLSSettings	No	外部请求访问的 TLS 卸载相关的配置

IstioEgressListener

定义了出流量的监听器配置，具体说明如下：

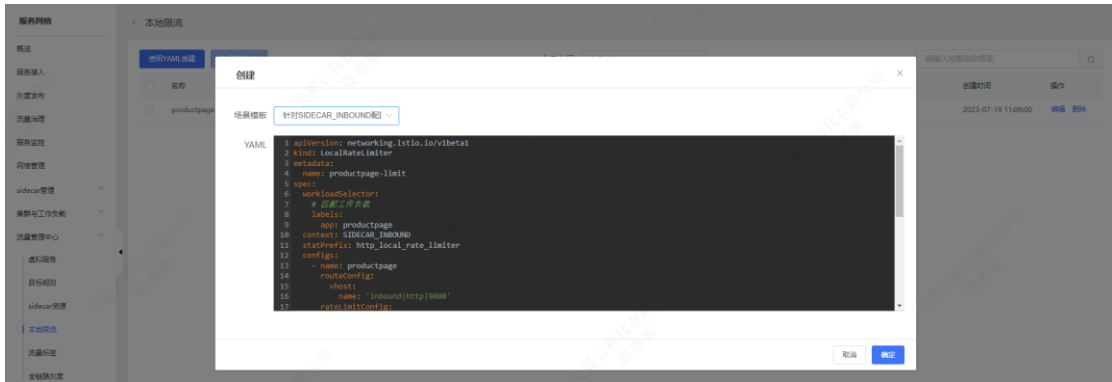
字段	类型	必选	说明
port	Port	No	监听器端口
bind	string	No	监听器地址，可以是 IP 或者 unix domain socket
captureMode	CaptureMode	No	流量拦截策略
hosts	[]string	Yes	当前 sidecar 可以访问的外部服务列表，采用 namespace/dnsName 格式

3.6.4 本地限流

在业务流量突增的场景下，为了保护系统不被冲垮，一般会采用限流实现对后端系统的保护。CSM 服务网格提供了本地限流能力，轻松实现服务限流，下面介绍本地限流功能的操作说明。

创建本地限流策略

1. 登陆服务网格控制台，选择流量管理中心->本地限流，选择本地限流要生效的命名空间
2. 选择使用 YAML 创建，当前提供了 SIDECAR_INBOUND 限流配置的模板



3. 根据模板修改配置，保存即可

修改本地限流策略

1. 登陆服务网格控制台，选择流量管理中心->本地限流，选择本地限流要生效的命名空间
2. 本地限流页面默认展示当前命名空间下的所有限流策略，在操作栏选择编辑操作修改限流配置



删除本地限流策略

1. 登陆服务网格控制台，选择流量管理中心->本地限流，选择本地限流要生效的命名空间
2. 本地限流页面默认展示当前命名空间下的所有限流策略，在操作栏选择删除操作删除限流配置



本地限流策略配置说明

CSM 服务网格采用自定义资源形式实现本地限流配置，下面的限流策略配置匹配 app 是 productpage 的工作负载，针对 productpage 应用的 SIDECAR_INBOUND 流量进行限流；采用令牌桶算法限流，最大 token 数量为 10，token 填充间隔为 60 秒，每次填充 10 个 token；限流策略针对所有流量都打开并强制执行；如果请求被限流，在应答头部里会增加 x-local-rate-limit: true

```
apiVersion: networking.istio.io/v1beta1
```

```
kind: LocalRateLimiter
```

```
metadata:
```

```
  name: productpage-limit
```

```
spec:
```

```
  workloadSelector:
```

```
    # 匹配工作负载
```

```
  labels:
```

```
    app: productpage
```

```
  context: SIDECAR_INBOUND
```

```
  statPrefix: http_local_rate_limiter
```

```
  configs:
```

```
    - name: productpage
```

```
  routeConfig:
```

```
    vhost:
```

```
      name: 'inbound|http|9080'
```

```
  rateLimitConfig:
```

```
    tokenBucket:
```

maxTokens: 10
 tokensPerFill: 10
 fillInterval: 60s
 filterEnabled:
 runtimeKey: local_rate_limit_enabled
 defaultValue:
 numerator: 100
 denominator: HUNDRED
 filterEnforced:
 runtimeKey: local_rate_limit_enforced
 defaultValue:
 numerator: 100
 denominator: HUNDRED
 responseHeadersToAdd:
 - appendAction: OVERWRITE_IF_EXISTS_OR_ADD
 header:
 key: x-local-rate-limit
 value: 'true'

LocalRateLimiter 详细配置说明：

字段	类型	必选	说明
workload_selector	WorkloadSelector	No	基于标签选择本地限流配置生效的工作负载
context	PatchContext	Yes	本地限流生效的 context，枚举值，支持 ANY（所有 context），SIDECAR_INBOUND（sidecar 入流量方向），SIDECAR_OUTBOUND（sidecar 出流量方向），GATEWAY（网关）；一般情况下，限流发生在服务接收端 SIDECAR_INBOUND 方向或者在网关进行统一限流

stat_prefix	string	Yes	Prometheus 指标前缀
configs	[]LocalRateLimitConfig	No	限流配置

LocalRateLimitConfig 配置说明

该配置定义了具体的限流参数，如下表：

字段	类型	必选	说明
name	string	No	限流配置的名字
route_config	RouteConfigurationMatch	No	限流策略匹配的路由配置
rate_limit_config	LocalRateLimit	No	限流算法相关参数

RouteConfigurationMatch 配置：

字段	类型	必选	说明
vhost	VirtualHostMatch	Yes	匹配路由中的虚拟主机，将限流策略应用到对应的虚拟主机

VirtualHostMatch 配置：

字段	类型	必选	说明
name	string	Yes	匹配的虚拟主机的名字
route	RouteMatch	No	匹配虚拟主机下的特定路由

RouteMatch

字段	类型	必选	说明
name	string	Yes	路由匹配名称
action	Action	No	路由匹配操作，枚举值，支持 ROUTE（转发），REDIRECT（重定向），DIRECT_RESPONSE（直接返回），或者 ANY（任意）

LocalRateLimit 定义了限流算法相关参数，具体如下：

字段	类型	必选	说明
token_bucket	TokenBucket	Yes	令牌桶算法参数
filter_enabled	RuntimeFractionalPercent	No	使用限流策略的流量比例，不强制执行
filter_enforced	RuntimeFractionalPercent	No	在使用限流策略的流量中强制执行限流的比例
request_headers_to_add_when_not_enforced	[]HeaderValueOption	No	被限流但是未被强制执行的请求转发时添加的头部
response_headers_to_add	[]HeaderValueOption	No	被执行限流的请求添加的应答头部
descriptors	[]LocalRateLimitDescriptor	No	限流描述符列

			表
local_rate_limit_per_downstream_connection	bool	No	设置为 false 时, token bucket 在 sidecar 线程间共享; 设置为 true 时, sidecar 为每个连接都创建一个 token bucket
enable_x_ratelimit_headers	XRateLimitHeadersRFCVersion	No	RFC X-RateLimit 标准版本头部定义
vh_rate_limits	VhRateLimitsOptions	No	路由级限流是否需要包含虚拟服务级限流配置

3.6.5 流量标签

在 CSM 服务网格中可以通过自定义的资源 (TrafficLabel) 实现对请求流量的打标, 进而可以基于流量的标签进行单服务及全链路的路由、策略控制等。本文介绍流量打标原理, 流

量标签管理，流量标签配置示例及详细说明。

流量打标原理

流量打标就是要给流量打上标签，进而可以对流量进行路由等策略配置。当前 CSM 服务网格支持从请求头部或者工作负载本身的标签中取值作为流量的标签；同时还支持在调用链中透传流量标签信息，用于实现全链路的标签路由策略。

下面是一个流量标签的配置示例，定义了一个流量标签 `userdefinelabel1`，会取当前 pod 的 `CSM_TRAFFIC_TAG` 标签值作为 `userdefinelabel1` 的值添加到请求头部里面；同时还定义了另外一个 `userdefinelabel2` 标签，该标签定义优先从请求的 `x-traffic-tag` 头部取值，同时使用 Envoy 生成的 `x-request-id` 作为上下文 key，在整个调用链路中透传该标签；如果打标失败了，会按照 `valueFrom` 数组中下一个定义继续取值，取到第一个非空值时停止。

```
apiVersion: networking.istio.io/v1beta1
kind: TrafficLabel
metadata:
  name: traffic-label-example
  namespace: demo
spec:
  rule:
    labels:
      - name: userdefinelabel1
        valueFrom:
          - $getLabel(CSM_TRAFFIC_TAG)
      - name: userdefinelabel2
        valueFrom:
          - $getInboundRequestHeaderWithContext(x-traffic-tag, x-request-id)
          - $getLabel(CSM_TRAFFIC_TAG)
```

当前 CSM 服务网格支持的流量打标方式有以下几种：

函数	说明	示例
<code>getInboundRequestHeader(headerName)</code>	从 Inbound 方向的请求 header 中取值；只应用于	请求 headerName : v1

	istio ingress 网关	打标后： userdefinelabel: v1
getInboundRequestHeaderWithContext(headerName, contextKey)	从 Inbound 方向的请求 header 取值，同时从 contextKey 头部取值作为流量标签在上下文透传的 key，该头部可以是用于实现链路追踪上下文透传的头部，比如 x-request-id、x-b3-traceid 或者您的业务中使用的自定义头部；该函数只适用于 sidecar	请求： contextID: 1234abcd headerName: v1 打标后： contextID: 1234abcd headerName: v1 userdefinelabel: v1
getOutboundRequestHeader(headerName)	从 outbound 方向的请求头部取值，只适用于 sidecar	请求 headerName： v1 打标后： userdefinelabel: v1
getLabel(labelName)	从 pod 的标签取值；适用于 istio ingress 网关和 sidecar	Pod 标签： labelName: v1 打标后： userdefinelabel: v1

流量标签管理

创建流量标签

1. 通过服务网格控制台 流量管理中心 -> 流量标签菜单进入流量标签管理页面
2. 选择命名空间，选择左上角使用 YAML 创建
3. 基于模板编辑流量标签定义，保存即可

修改流量标签

1. 通过服务网格控制台 流量管理中心 -> 流量标签菜单进入流量标签管理页面
2. 选择命名空间，列表页会展示当前命名空间下定义的 TrafficLabel 对象
3. 选择操作栏的编辑，修改定义，保存即可

删除流量标签

1. 通过服务网格控制台 流量管理中心 -> 流量标签菜单进入流量标签管理页面
2. 选择命名空间，列表页会展示当前命名空间下定义的 TrafficLabel 对象
3. 选择操作栏的删除，即可删除已定义的 TrafficLabel 对象

TrafficLabel 配置说明：

字段	类型	必选	说明
workload_selector	WorkloadSelector	No	工作负载选择器，选择当前流量标签生效的工作负载
rule	TrafficLabelRule	No	流量标签定义

TrafficLabelRule

流量标签的详细定义

字段	类型	必选	说明
labels	[]Label	No	流量标签定义

Label

字段	类型	必选	说明
name	string	No	流量标签名称, 该标签会被添加到请求头部
value_from	[]string	No	流量标签取值, value_from 为数组, 按照顺序取到的第一个非空值作为该标签的取值

流量标签使用示例

准备工作

首先部署测试应用 app1 和 app2, 调用关系为 ingress -> app1-> app2

```
apiVersion: v1
kind: Service
metadata:
  name: app1
labels:
  app: app1
  service: app1
spec:
  ports:
    - port: 8000
      name: http
  selector:
    app: app1
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1-base
  namespace: demo
labels:
  app: app1
  CSM_TRAFFIC_TAG: base
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app1
      name: app1
      CSM_TRAFFIC_TAG: base
template:
  metadata:
    labels:
      app: app1
```

```
    name: app1
    source: CCSE
    CSM_TRAFFIC_TAG: base
spec:
  containers:
  - name: default
    image: registry-vpc-crs-huadong1.ctyun.cn/library/trace-demo:v1.0.0
    imagePullPolicy: IfNotPresent
    env:
    - name: version
      value: base
    - name: app
      value: app1
    - name: upstream_url
      value: "http://app2:8000/"
    ports:
    - containerPort: 8000
---
apiVersion: v1
kind: Service
metadata:
  name: app2
  labels:
    app: app2
    service: app2
spec:
  ports:
  - port: 8000
    name: http
  selector:
    app: app2
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2-base
  namespace: demo
  labels:
    app: app2
    CSM_TRAFFIC_TAG: base
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app2
      name: app2
    CSM_TRAFFIC_TAG: base
  template:
    metadata:
```

```
labels:
  app: app2
  name: app2
  source: CCSE
  CSM_TRAFFIC_TAG: base
spec:
  containers:
  - name: default
    image: registry-vpc-crs-huadong1.ctyun.cn/library/trace-demo:v1.0.0
    imagePullPolicy: IfNotPresent
    env:
    - name: version
      value: base
    - name: app
      value: app2
  ports:
  - containerPort: 8000
```

部署 Gateway 和 VirtualService 资源

```
apiVersion: networking.istio.io/v1beta1
```

```
kind: Gateway
```

```
metadata:
```

```
  name: trace-demo-gateway
```

```
spec:
```

```
  selector:
```

```
    istio: ingressgateway
```

```
  servers:
```

```
  - hosts:
```

```
    - "*"
```

```
    port:
```

```
      name: http
```

```
      number: 80
```

```
      protocol: HTTP
```

```
---
```

```
apiVersion: networking.istio.io/v1beta1
```

```
kind: VirtualService
```

```
metadata:
  name: vs-gateway-app1
spec:
  gateways:
  - trace-demo-gateway
  hosts:
  - "*"
  http:
  - route:
    - destination:
        host: app1
```

Ingress gateway 流量打标

Ingress 网关侧对流量打标之后我们看通过查看上游的 app1 服务收到的请求头部确认流量打标是否成功，首先定义流量标签策略：

```
apiVersion: networking.istio.io/v1beta1
kind: TrafficLabel
metadata:
  name: traffic-label-example
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      istio: ingressgateway
  rule:
    labels:
      - name: label-from-header
        valueFrom:
          - $getInboundRequestHeader(app-version-tag)
      - name: label-from-pod
        valueFrom:
          - $getLabel(app)
```

上面的配置在 Ingress 网关处定义了两个标签，label-from-header 会从 app-version-tag 请求头部取值打标，label-from-pod 从 pod 的 app 标签取值打标。我们通过 Ingress 网关访问 app1 服务，并带上 app-version-tag: ccccc，查看 app1 的日志如下：


```

023/08/21 03:46:52 request header [X-Envoy-Attempt-Count] => [[1]]
023/08/21 03:46:52 request header [Traceparent] => [[00-00000000000000000000000000000000-12-01]]
023/08/21 03:46:52 request header [X-Forwarded-Client-Cert] => [[By=spiffe://cluster.local/ns/default/sa/default;
://cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account]]
023/08/21 03:46:52 request header [X-B3-Traceid] => [[b'00000000000000000000000000000000']]
023/08/21 03:46:52 request header [X-B3-Spanid] => [[2f'00000000000000000000000000000000']]
023/08/21 03:46:52 request header [Label-From-Header] => [[cccc]]
023/08/21 03:46:52 request header [Label-From-Pod] => [[istio-ingressgateway]]
023/08/21 03:46:52 request header [App-Version-Tag] => [[cccc]]
023/08/21 03:46:52 request header [X-Request-Id] => [[1a'00000000000000000000000000000000']]
023/08/21 03:46:52 request header [X-B3-Parentspanid] => [[d2d'00000000000000000000000000000000']]
023/08/21 03:46:52 request header [X-B3-Sampled] => [[1]]
023/08/21 03:46:52 request header [User-Agent] => [[curl/7.88.1]]
023/08/21 03:46:52 request header [X-Forwarded-Proto] => [[http]]

```

Sidecar 流量打标

首先定义 sidecar 流量打标规则，label1 从 app-version-tag 头部取值，同时使用 trace-ctx-id 字段做链路传递；label2 去 outbound 方向的 trace-ctx-id 字段；label3 取应用的 app 标签值：

apiVersion: [networking.istio.io/v1beta1](https://kubernetes.io/docs/concepts/containers/apiserver-endpoints/)

kind: TrafficLabel

metadata:

name: traffic-label-example

namespace: default

spec:

rule:

labels:

- name: label1

valueFrom:

- \$getInboundRequestHeaderWithContext(app-version-tag, trace-ctx-id)

- name: label2

valueFrom:

- \$getOutboundRequestHeader(trace-ctx-id)

- name: label3

valueFrom:

- \$getLabel(app)

请求 Ingress 网关带上两个头部 'app-version-tag: ccccc' 和 'trace-ctx-id: 1234'，查看 app2 的日志，可以看到预期中的流量标签：

```

2023/08/21 06:10:23 request header [X-B3-Sampled] => [[1]]
2023/08/21 06:10:23 request header [X-Forwarded-Proto] => [[http]]
2023/08/21 06:10:23 request header [X-Request-Id] => [[3'00000000000000000000000000000000']]
2023/08/21 06:10:23 request header [App-Version-Tag] => [[cccc]]
2023/08/21 06:10:23 request header [Label3] => [[app2]]
2023/08/21 06:10:23 request header [X-B3-Parentspanid] => [[b'00000000000000000000000000000000']]
2023/08/21 06:10:23 request header [User-Agent] => [[Go-http-client/1.1]]
2023/08/21 06:10:23 request header [Label1] => [[cccc]]
2023/08/21 06:10:23 request header [Label2] => [[1234]]
2023/08/21 06:10:23 request header [X-Envoy-Attempt-Count] => [[1]]
2023/08/21 06:10:23 request header [X-Forwarded-Client-Cert] => [[By=spiffe://cluster.local/ns
e://cluster.local/ns/default/sa/default]]

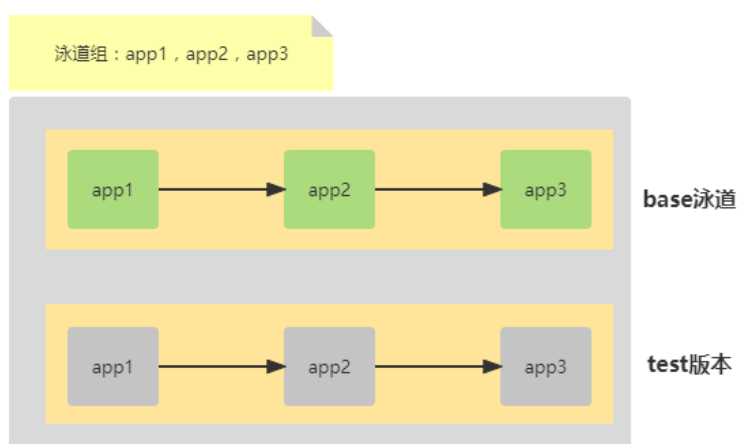
```

3.6.6 全链路灰度

在微服务架构中，一次业务迭代可能涉及到多个微服务，业务上线前为了测试和验证搭建一套测试环境往往是一个比较头疼的问题，测试过程中跟其他迭代之间可能有相互影响或者冲突，导致迭代效率低下。全链路灰度能力将灰度和全链路进行结合，允许对一个调用链路上的微服务统一进行灰度，通过流量标签功能，实现在整个调用链透传流量标签；然后基于该标签配置对每个微服务的治理策略，即可实现对调用链路的统一治理。

泳道及泳道组

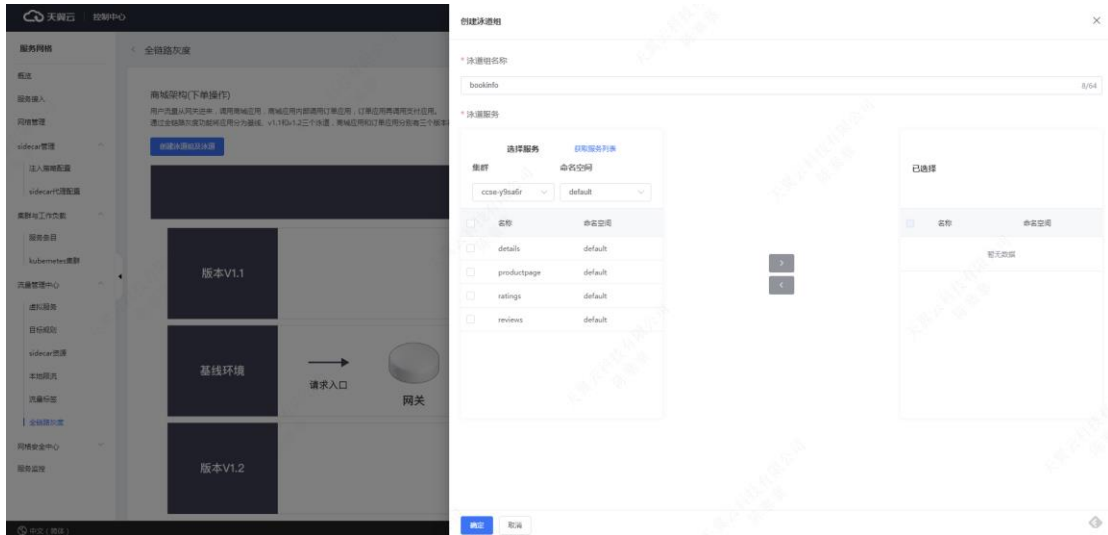
针对全链路灰度场景，我们首先定义了泳道组和泳道的概念，其中泳道组包含一组需要进行全链路灰度的微服务；泳道是泳道组内的概念，每个泳道代表一个虚拟的流量通道，全链路灰度时期望流量只在某个泳道内流动，如下图所示：



泳道组管理

从服务网格控制台选择 流量管理中心 -> 流量泳道 -> 创建泳道组及泳道

选择集群和命名空间，选择要添加到泳道组的服务，如下图：



配置参数说明：

配置	说明
泳道组名称	唯一标识一个泳道组
集群	泳道组服务所在的集群
命名空间	泳道组所在的集群命名空间
服务列表	泳道组包含的服务

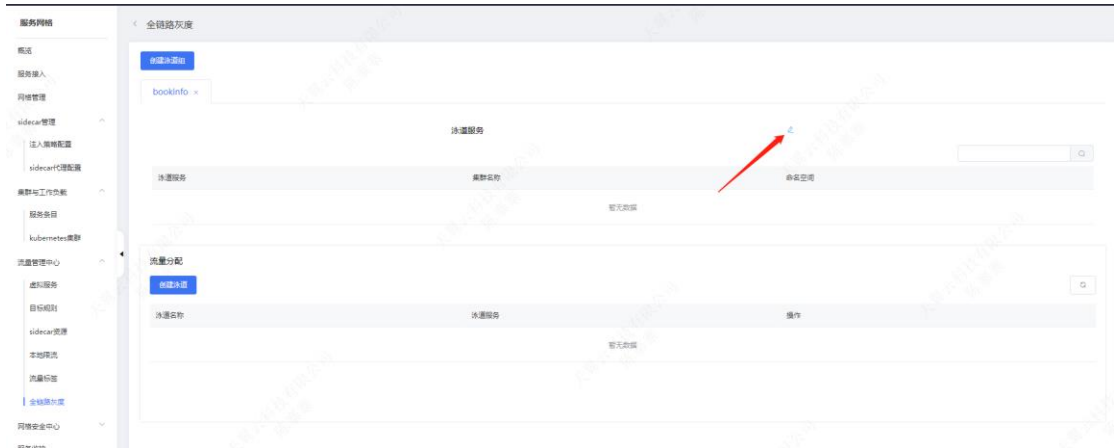
在创建泳道组后，后台会创建一些资源说明：

资源名称	类型	说明
trafficlabel-\${命名空间}	TrafficLabel	用于实现流量打标，默认从工作负载的 CSM_TRAFFIC_TAG 标签取值，赋值给 userdefinelabel 标签；可以根据业务需求参考流量标签章节修改流量打标逻辑
tl-\${泳道组名称}-\${命名空间}-\${泳道组服务名称}	VirtualService	基于上面的流量标签定义实现对泳道组内服务的路由控制，默认基于流量标签和泳道名称进行匹配，匹配成功则路由到对应泳道
tl-\${泳道组名称}-\${命名空间}	DestinationRule	用于实现泳道分组定义

\${泳道组服务名称}		
-------------	--	--

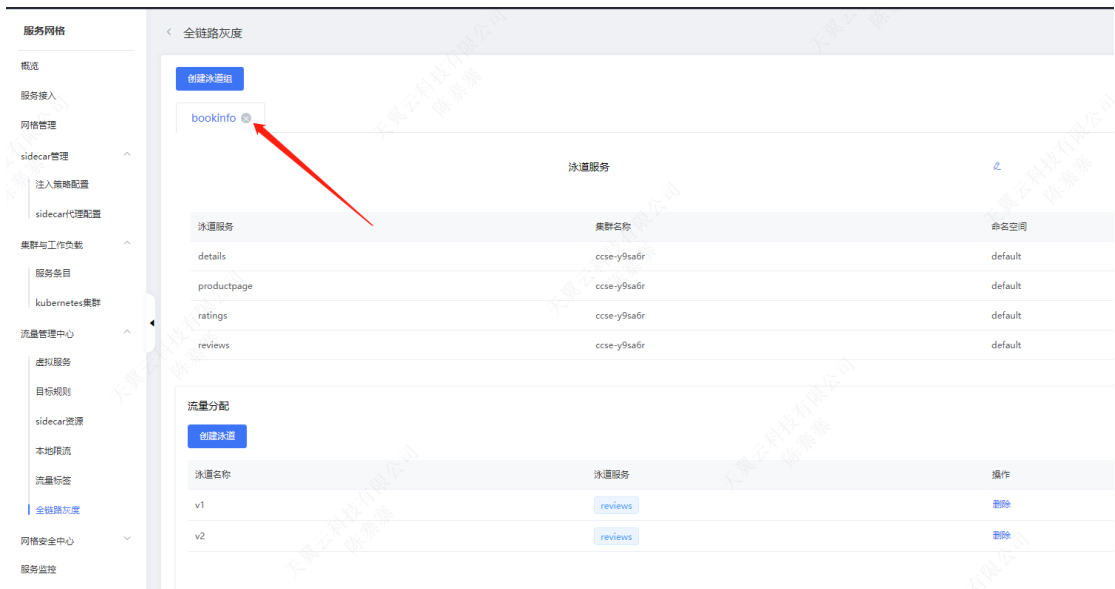
编辑泳道组

您可以在泳道组页面修改泳道包含的服务，如下图：



删除泳道组

点击泳道组 tab 的删除按钮即可



创建泳道

在当前泳道组内可以定义多个泳道，选择 创建泳道，填写泳道名称并选择泳道内的服务即可；需要注意：

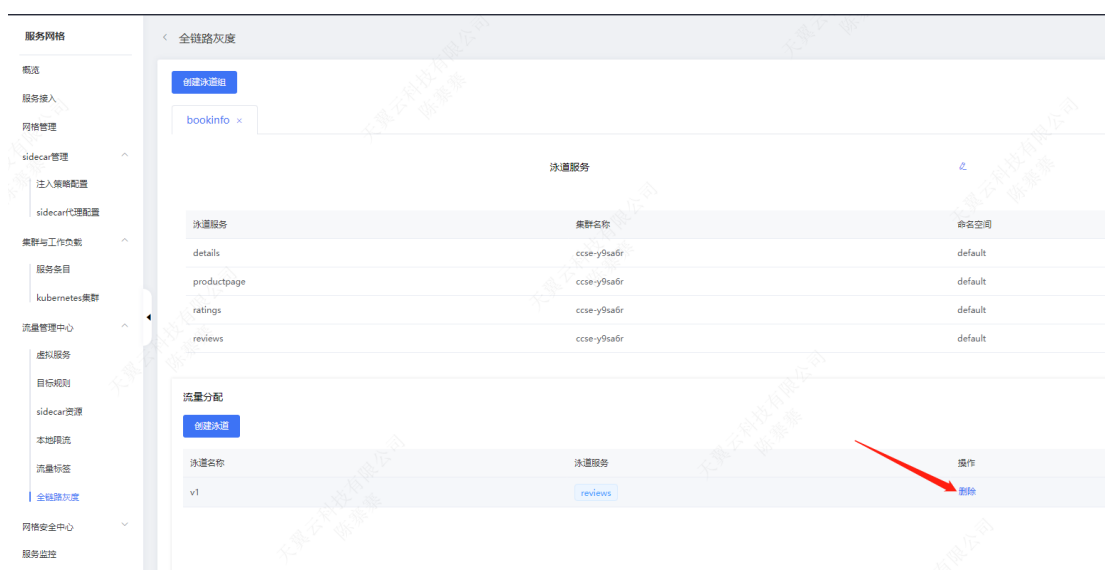
1. 泳道名称最终用于和流量标签进行匹配

- 默认按照服务的 CSM_TRAFFIC_TAG 标签对服务进行分组，该标签值需要和泳道名称保持一致



删除泳道

在泳道组内的泳道列表操作栏，选择删除操作即可



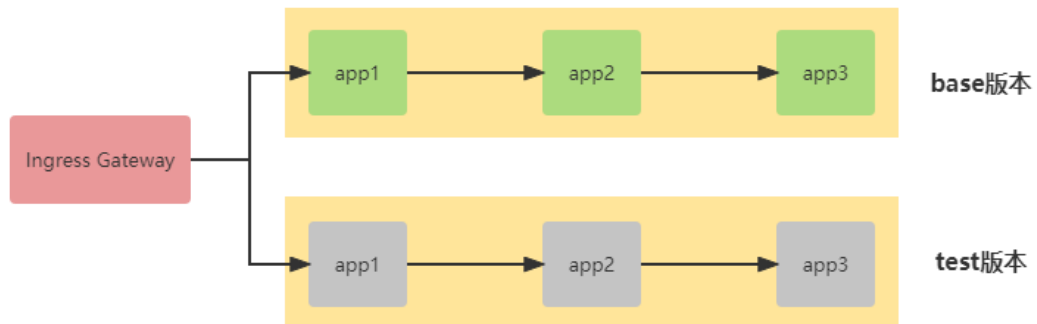
全链路灰度验证

全链路灰度是基于流量标签功能，在微服务整个调用链中进行统一的流量打标和治理，从而实现全链路灰度能力。本文介绍如何实现全链路灰度发布能力。

微服务调用关系

微服务架构中有三个服务，app1、app2、app3，调用关系如下图，入口由 Ingress 网关做流量分发，要实现全链路灰度，预期是可以实现微服务之间只在在 base 版本内和 test 版

本内调用，实现虚拟泳道。



前提条件：

1. 开通 CCSE（容器云服务引擎）
2. 开通服务网格实例

操作步骤：

首先部署 base 版本应用：

```
apiVersion: v1
kind: Service
metadata:
  name: app1
  labels:
    app: app1
    service: app1
spec:
  ports:
    - port: 8000
      name: http
  selector:
    app: app1
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1-base
  namespace: demo
  labels:
    app: app1
    CSM_TRAFFIC_TAG: base
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
  app: app1
  name: app1
  CSM_TRAFFIC_TAG: base
template:
  metadata:
    labels:
      app: app1
      name: app1
      source: CCSE
      CSM_TRAFFIC_TAG: base
  spec:
    containers:
      - name: default
        image: registry-vpc-crs-huadong1.ctyun.cn/library/trace-demo:v1.0.0
        imagePullPolicy: IfNotPresent
        env:
          - name: version
            value: base
          - name: app
            value: app1
          - name: upstream_url
            value: "http://app2:8000/"
        ports:
          - containerPort: 8000
---
apiVersion: v1
kind: Service
metadata:
  name: app2
  labels:
    app: app2
    service: app2
spec:
  ports:
    - port: 8000
      name: http
  selector:
    app: app2
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2-base
  namespace: demo
  labels:
    app: app2
    CSM_TRAFFIC_TAG: base
spec:
  replicas: 1
```

```
selector:
  matchLabels:
    app: app2
    name: app2
    CSM_TRAFFIC_TAG: base
template:
  metadata:
    labels:
      app: app2
      name: app2
      source: CCSE
      CSM_TRAFFIC_TAG: base
  spec:
    containers:
      - name: default
        image: registry-vpc-crs-huadong1.ctyun.cn/library/trace-demo:v1.0.0
        imagePullPolicy: IfNotPresent
        env:
          - name: version
            value: base
          - name: app
            value: app2
          - name: upstream_url
            value: "http://app3:8000/"
        ports:
          - containerPort: 8000
```

```
apiVersion: v1
kind: Service
metadata:
  name: app3
  labels:
    app: app3
    service: app3
```

```
spec:
  ports:
    - port: 8000
      name: http
  selector:
    app: app3
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app3-base
  namespace: demo
  labels:
    app: app3
    CSM_TRAFFIC_TAG: base
```



```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app3
      name: app3
      CSM_TRAFFIC_TAG: base
  template:
    metadata:
      labels:
        app: app3
        name: app3
        source: CCSE
        CSM_TRAFFIC_TAG: base
    spec:
      containers:
        - name: default
          image: registry-vpc-crs-huadong1.ctyun.cn/library/trace-demo:v1.0.0
          imagePullPolicy: IfNotPresent
          env:
            - name: version
              value: base
            - name: app
              value: app3
          ports:
            - containerPort: 8000
```

部署 test 版本应用 :

```
apiVersion: v1
kind: Service
metadata:
  name: app1
  labels:
    app: app1
    service: app1
spec:
  ports:
    - port: 8000
      name: http
  selector:
    app: app1
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1-test
  namespace: demo
```

```
labels:
  app: app1
  CSM_TRAFFIC_TAG: test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app1
      name: app1
      CSM_TRAFFIC_TAG: test
  template:
    metadata:
      labels:
        app: app1
        name: app1
        source: CCSE
        CSM_TRAFFIC_TAG: test
    spec:
      containers:
        - name: default
          image: registry-vpc-crs-huadong1.ctyun.cn/library/trace-demo:v1.0.0
          imagePullPolicy: IfNotPresent
          env:
            - name: version
              value: test
            - name: app
              value: app1
            - name: upstream_url
              value: "http://app2:8000/"
          ports:
            - containerPort: 8000
---
apiVersion: v1
kind: Service
metadata:
  name: app2
  labels:
    app: app2
    service: app2
spec:
  ports:
    - port: 8000
      name: http
  selector:
    app: app2
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
name: app2-test
namespace: demo
labels:
  app: app2
  CSM_TRAFFIC_TAG: test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app2
      name: app2
      CSM_TRAFFIC_TAG: test
  template:
    metadata:
      labels:
        app: app2
        name: app2
        source: CCSE
        CSM_TRAFFIC_TAG: test
    spec:
      containers:
        - name: default
          image: registry-vpc-crs-huadong1.ctyun.cn/library/trace-demo:v1.0.0
          imagePullPolicy: IfNotPresent
          env:
            - name: version
              value: test
            - name: app
              value: app2
            - name: upstream_url
              value: "http://app3:8000/"
          ports:
            - containerPort: 8000
---
apiVersion: v1
kind: Service
metadata:
  name: app3
  labels:
    app: app3
    service: app3
spec:
  ports:
    - port: 8000
      name: http
  selector:
    app: app3
---
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: app3-test
  namespace: demo
  labels:
    app: app3
    CSM_TRAFFIC_TAG: test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app3
      name: app3
      CSM_TRAFFIC_TAG: test
  template:
    metadata:
      labels:
        app: app3
        name: app3
        source: CCSE
        CSM_TRAFFIC_TAG: test
    spec:
      containers:
        - name: default
          image: registry-vpc-crs-huadong1.ctyun.cn/library/trace-demo:v1.0.0
          imagePullPolicy: IfNotPresent
          env:
            - name: version
              value: test
            - name: app
              value: app3
          ports:
            - containerPort: 8000
```

部署 Gateway 和虚拟路由资源：

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: trace-demo-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - "*"
      port:
        name: http
```

```

    number: 80
    protocol: HTTP
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: vs-gateway-app1
spec:
  gateways:
  - trace-demo-gateway
  hosts:
  - "*"
  http:
  - route:
    - destination:
        host: app1

```

确定网关的访问入口地址，设置为 GATEWAY_HOST 环境变量，执行如下语句访问：

```
for i in {1..100}; do curl http://${GATEWAY_HOST}/; echo; sleep 1; done;
```

```

└─$ for i in {1..100}; do curl http://${GATEWAY_HOST}/; echo; sleep 1; done;
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: base, ip: 10.1.0.204)
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: test, ip: 10.1.0.207)
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: base, ip: 10.1.0.204)
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: test, ip: 10.1.0.207)
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: test, ip: 10.1.0.207)
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: test, ip: 10.1.0.207)
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: base, ip: 10.1.0.204)
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: base, ip: 10.1.0.204)
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)

```

可以看到访问在两个版本之间跳变，微服务之间的调用也是在版本间穿插进行，此时的全链路没有统一的路由策略。

配置流量标签

在 Ingress 网关处，使用 app-version-tag 请求头部给流量打上标签，用于从 Ingress 网关到 app1 的路由控制；从 app-version-tag 头部给流过各个应用的流量打标，通过 trace-ctx-id 字段在调用链上下游传递流量标签：

```

apiVersion: networking.istio.io/v1beta1
kind: TrafficLabel
metadata:
  name: traffic-label-example
  namespace: istio-system
spec:
  workloadSelector:

```

```
labels:
  istio: ingressgateway
rule:
  labels:
  - name: version-label
    valueFrom:
    - $getInboundRequestHeader(app-version-tag)
---
apiVersion: networking.istio.io/v1beta1
kind: TrafficLabel
metadata:
  name: traffic-label-example
  namespace: default
spec:
  rule:
  labels:
  - name: version-label
    valueFrom:
    - $getInboundRequestHeaderWithContext(app-version-tag, trace-ctx-id)
```

部署目标规则和虚拟服务策略

通过目标规则给 base 和 test 版本分别定义 subset，通过虚拟服务定义基于 version-label 的路由策略：

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: dr-app1
spec:
  host: app1
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
  - labels:
    version: base
    name: base
  - labels:
    version: test
    name: test
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: dr-app2
spec:
  host: app2
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
```

```
subsets:
  - labels:
      version: base
      name: base
  - labels:
      version: test
      name: test
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: dr-app3
spec:
  host: app3
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
    - labels:
        version: base
        name: base
    - labels:
        version: test
        name: test
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: vs-gateway-app1
spec:
  gateways:
    - trace-demo-gateway
  hosts:
    - "*"
  http:
    - match:
        - headers:
            version-label:
              exact: base
      route:
        - destination:
            host: app1
            subset: base
    - match:
        - headers:
            version-label:
              exact: test
      route:
        - destination:
```

```
    host: app1
    subset: test
  - route:
    - destination:
      host: app1
  ---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: vs-app2
spec:
  hosts:
  - app2
  http:
  - match:
    - headers:
      version-label:
        exact: base
    route:
    - destination:
      host: app2
      subset: base
    - match:
      - headers:
        version-label:
          exact: test
      route:
      - destination:
        host: app2
        subset: test
    - route:
      - destination:
        host: app2
  ---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: vs-app3
spec:
  hosts:
  - app3
  http:
  - match:
    - headers:
      version-label:
        exact: base
    route:
    - destination:
      host: app3
```


- subset: base
- match:
- headers:
 - version-label:
 - exact: test
- route:
- destination:
 - host: app3
 - subset: test
- route:
 - destination:
 - host: app3

请求带上 app-version-tag: base, 同时指定请求级别的 trace-ctx-id :
for i in {1..200}; do curl -H "app-version-tag: base" -H "trace-ctx-id: trace-\$i"
[http://\\${GATEWAY_HOST}](http://${GATEWAY_HOST}); echo; sleep 1; done;

可以看到请求只在 base 版本之间调用

```
for i in {1..200}; do curl -H "app-version-tag: base" -H "trace-ctx-id: trace-$i" http://${GATEWAY_HOST}; echo; sleep 1; done;  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)  
[app1] (version: base, ip: 10.1.0.202) -> [app2] (version: base, ip: 10.1.0.203) -> [app3] (version: base, ip: 10.1.0.204)
```

修改头部 app-version-tag: test, 可以看到请求只在 test 版本之间调用 :

```
for i in {1..200}; do curl -H "app-version-tag: test" -H "trace-ctx-id: trace-$i" http://${GATEWAY_HOST}; echo; sleep 1; done;  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)  
[app1] (version: test, ip: 10.1.0.205) -> [app2] (version: test, ip: 10.1.0.206) -> [app3] (version: test, ip: 10.1.0.207)
```

3.6.7 多协议治理

gRPC 协议治理

gRPC 是谷歌开发的远程过程调用框架 (RPC), 有多语言的实现, 底层采用 HTTP2 作为传输协议; 由于 HTTP2 采用长连接机制, 在负载均衡的场景下可能导致负载的不平衡, 本文介绍负载不均衡的场景以及如何通过服务网格实现负载均衡。

前提条件：

1. 开通 CCSE（容器云服务引擎）
2. 开通服务网格实例

操作步骤：

部署 gRPC server 和 client 应用

apiVersion: apps/v1

kind: Deployment

metadata:

name: grpc-server-v1

labels:

app: grpc-server

version: v1

spec:

replicas: 1

selector:

matchLabels:

app: grpc-server

version: v1

template:

metadata:

labels:

app: grpc-server

version: v1

spec:

containers:

- args:

- --address=0.0.0.0:8080

image: registry-vpc-crs-huadong1.ctyun.cn/library/grpc-server

imagePullPolicy: Always

```
name: grpc-server
ports:
- containerPort: 8080
```

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
name: grpc-server-v2
```

```
labels:
```

```
app: grpc-server
```

```
version: v2
```

```
spec:
```

```
replicas: 1
```

```
selector:
```

```
matchLabels:
```

```
app: grpc-server
```

```
version: v2
```

```
template:
```

```
metadata:
```

```
labels:
```

```
app: grpc-server
```

```
version: v2
```

```
spec:
```

```
containers:
```

```
- args:
```

```
- --address=0.0.0.0:8080
```

```
image: registry-vpc-crs-huadong1.ctyun.cn/library/grpc-server
```

```
imagePullPolicy: Always
```

```
name: grpc-server
```

```
ports:
```

- containerPort: 8080

apiVersion: v1

kind: Service

metadata:

name: grpc-server

labels:

app: grpc-server

spec:

ports:

- name: grpc-backend

port: 8080

protocol: TCP

selector:

app: grpc-server

type: ClusterIP

apiVersion: apps/v1

kind: Deployment

metadata:

name: grpc-client

labels:

app: grpc-client

spec:

replicas: 1

selector:

matchLabels:

app: grpc-client

template:

metadata:

labels:

app: grpc-client

"sidecar.istio.io/inject": "true"

spec:

containers:

- image: registry-vpc-crs-huadong1.ctyun.cn/library/grpc-client

imagePullPolicy: Always

command: ["/bin/sleep", "3650d"]

name: grpc-client

部署之后的 pod 列表（一个 client，两个版本的 server）：

```
[root@ecs-ymogqzhp ~]# kubectl get po -n grpc
NAME                                READY   STATUS    RESTARTS   AGE
grpc-client-b7499b9c-45d2s          1/1     Running   0           12m
grpc-server-v1-76bfb999dd-d65wn     1/1     Running   0           12m
grpc-server-v2-67b7d8f479-px64j    1/1     Running   0           12m
```

通过 client 访问 server，可以看到总是访问服务端的同一个实例

kubectl exec -it grpc-client-b7499b9c-45d2s -n grpc -- /bin/greeter-client --insecure=true --address=grpc-server:8080 --repeat=10

```
[root@ecs-ymogqzhp ~]# kubectl exec -it grpc-client-b7499b9c-45d2s -n grpc -- /bin/greeter-client --insecure=true --address=grpc-server:8080 --repeat=10
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
2023/10/18 08:30:08 Hello world from grpc-server-v2-67b7d8f479-px64j
```

为 grpc client 注入 sidecar（打上标签"sidecar.istio.io/inject": "true"），重新部署 grpc-client 之后可以看到 pod 列表如下：

```
[root@ecs-ymogqzhp ~]# kubectl get po -n grpc
NAME                                READY   STATUS    RESTARTS   AGE
grpc-client-5676ff6f68-r5t2f        2/2     Running   0           2m38s
grpc-server-v1-76bfb999dd-db55b     1/1     Running   0           2m38s
grpc-server-v2-67b7d8f479-smwkg     1/1     Running   0           2m38s
```

再次通过 grpc-client 访问 grpc-server 可以看到请求交替访问两个版本的 grpc-server：

```
[ -]# kubectl exec -it grpc-client-5676ff6f68-r5t2f -n grpc -- /bin/greeter-client --insecure=true --address=grpc-server:8080 --repeat=10
2023/10/18 08:51:02 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:51:02 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:51:02 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:51:02 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:51:02 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:51:02 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:51:02 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:51:02 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:51:02 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:51:02 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:51:02 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:51:02 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:51:02 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:51:02 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:51:02 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:51:02 Hello world from grpc-server-v2-67b7d8f479-smwkg
```

部署流量治理策略使 70%的流量访问 v2 版本的 grpc-server,30%的流量访问 v1 版本的 grpc-server

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

name: dr-grpc-server

spec:

host: grpc-server

trafficPolicy:

loadBalancer:

simple: ROUND_ROBIN

subsets:

- name: v1

labels:

version: "v1"

- name: v2

labels:

version: "v2"

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: vs-grpc-server

spec:

hosts:

- "grpc-server"

http:

- match:

- port: 8080

route:

- destination:

host: grpc-server

subset: v1

weight: 30

- destination:

host: grpc-server

subset: v2

weight: 70

再次访问可以看到请求在 grpc-server 的两个版本之间不再是交替访问，而是大概按照 7:3 的比例访问：

```
2023/10/18 08:55:38 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:55:38 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:55:38 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:55:38 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:55:38 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:55:38 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:55:38 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:55:38 Hello world from grpc-server-v2-67b7d8f479-smwkg
2023/10/18 08:55:38 Hello world from grpc-server-v1-76bfb999dd-db55b
2023/10/18 08:55:38 Hello world from grpc-server-v1-76bfb999dd-db55b
```

3.6.8 高可用能力构建

在服务网格中使用本地限流

前提条件：

1. 开通 CCSE（容器云服务引擎）
2. 开通服务网格实例

操作步骤：

我们以 bookinfo 应用为例演示本地限流能力，首先在 default 命名空间部署 bookinfo 应用

```
[root@elk ~]# kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
details-v1-548fc9d755-vlwxq        2/2    Running   0           17m
productpage-v1-6cd599ddb6-ffrm6    2/2    Running   0           17m
ratings-v1-5748667fcf-tvzp2       2/2    Running   0           17m
reviews-v1-85b598c597-kwhcb       2/2    Running   0           17m
[root@elk ~]#
```

然后在 default 命名空间部署针对 productpage 服务的限流策略，token 数量最大为 5，每 2 秒重新填充 5 个 token：

```
apiVersion: networking.istio.io/v1beta1
```

```
kind: LocalRateLimiter
```

```
metadata:
```

```
  name: productpage-limit
```

```
spec:
```

```
  workloadSelector:
```

```
    # 匹配工作负载
```

```
  labels:
```

```
    app: productpage
```

```
context: SIDECAR_INBOUND
```

```
statPrefix: http_local_rate_limiter
```

```
configs:
```

```
- name: productpage
```

```
  routeConfig:
```

```
    vhost:
```

```
      name: 'inbound|http|9080'
```

```
  rateLimitConfig:
```

```
    tokenBucket:
```

```
      maxTokens: 5
```

```
      tokensPerFill: 5
```

```
      fillInterval: 2s
```

```
  filterEnabled:
```

```
    runtimeKey: local_rate_limit_enabled
```



```

defaultValue:
  numerator: 100
  denominator: HUNDRED
filterEnforced:
  runtimeKey: local_rate_limit_enforced
  defaultValue:
    numerator: 100
    denominator: HUNDRED
responseHeadersToAdd:
  - appendAction: OVERWRITE_IF_EXISTS_OR_ADD
    header:
      key: x-local-rate-limit
      value: 'true'

```

我们使用 go-stress-testing (<https://github.com/link1st/go-stress-testing>) 工具验证效果, 如下图所示, 结果状态码中有 200 和 429 (请求被限流); 200 的个数每 2 秒增加 5 个, 符合我们设定的 2 秒重新填充 5 个 token 的设定; 429 状态码在持续增加;

```

[ ]# go-stress-testing -c 10 -n 100000 -u http://192.168.1.44:9080/productpage
开始启动 并发数:10 请求数:100000 请求参数:
request:
form:http
url:http://192.168.1.44:9080/productpage
method:GET
headers:map[Content-Type:application/x-www-form-urlencoded; charset=utf-8]
data:
verify:statusCode
timeout:30s
debug:false
http2.0: false
keepalive: false
maxCon:1

```

耗时	并发数	成功数	失败数	qps	最长耗时	最短耗时	平均耗时	下载字节	字节每秒	状态码
1s	10	10	4135	10.39	245.21	0.89	962.85	116,260	116,257	200:10;429:4135
2s	10	10	8897	5.23	245.21	0.88	1911.24	201,976	100,986	200:10;429:8897
3s	10	15	13140	5.23	245.21	0.85	1910.65	299,265	99,753	200:15;429:13140
4s	10	15	14490	4.04	245.21	0.85	2473.45	323,565	80,878	200:15;429:14490
5s	10	20	15293	4.34	245.21	0.85	2303.46	358,934	71,786	200:20;429:15293
6s	10	20	17020	3.61	245.21	0.85	2772.14	390,020	64,993	200:20;429:17020
7s	10	25	18628	3.85	245.21	0.85	2599.16	439,879	62,828	200:25;429:18628
8s	10	25	20257	3.36	245.21	0.85	2977.27	469,201	58,640	200:25;429:20257
9s	10	30	21803	3.58	245.21	0.85	2796.60	517,944	57,547	200:30;429:21803
10s	10	30	23430	3.21	245.21	0.85	3111.14	547,230	54,722	200:30;429:23430
11s	10	35	24867	3.40	245.21	0.85	2938.44	594,011	54,000	200:35;429:24867

单独请求 productpage 服务可以看到请求被限流的情况：

```
[root@localhost ~]# curl http://192.168.1.44:9080/productpage -sv
* About to connect() to 192.168.1.44 port 9080 (#0)
*   Trying 192.168.1.44...
* Connected to 192.168.1.44 (192.168.1.44) port 9080 (#0)
> GET /productpage HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 192.168.1.44:9080
> Accept: */*
>
< HTTP/1.1 429 Too Many Requests
< x-local-rate-limit: true
< content-length: 18
< content-type: text/plain
< date: Tue, 08 Aug 2023 07:35:40 GMT
< server: istio-envoy
< x-envoy-decorator-operation: productpage.default.svc.cluster.local:9080/*
<
* Connection #0 to host 192.168.1.44 left intact
```

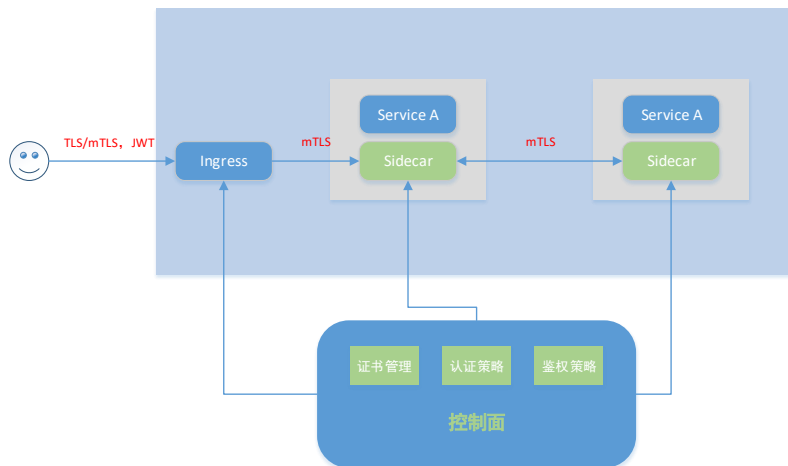
3.7 网络安全中心

3.7.1 网络安全概述

单体应用发展成微服务架构带来了诸多便利，业务迭代更加敏捷，扩展性更好，同时为服务架构也带了新的安全考虑，如：

1. 微服务之间通信安全问题，如何防止中间人攻击
 2. 微服务之间的访问控制问题，对于敏感服务和信息，如何限制微服务之间的访问
 3. 微服务之间访问频繁且关系复杂，如何追溯服务之间的调用情况，需要具备审计能力
- 服务网格的安全机制提供了零信任的安全机制，很好地解决了以上问题。

服务网络安全架构



服务身份及证书管理

身份是安全的基础，只有给每个服务赋予一个身份才能在服务相互访问时对各方的身份进行认证以及对操作进行鉴权。服务网格使用证书作为网格内工作负载的身份标识，服务网格负载网格内工作负载的身份证书颁发、轮换等，为网格安全提供基础能力。

身份认证

服务网格提供两种认证机制：

对等身份认证：用于 sidecar 代理之间通信时的身份认证，解决 sidecar 之间身份认证和通信加密问题，支持基于工作负载证书的 mTLS 双向认证

请求身份认证：用于外部请求进入网格内的身份认证，支持 JWT 及 OIDC 的认证方式

授权

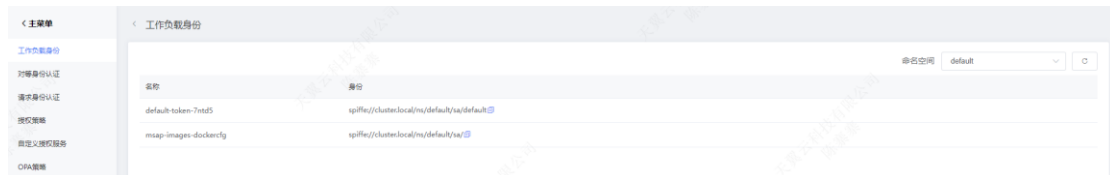
在微服务通信中，确定了双方的身份之后，我们还需要对客户端是否有权访问服务端的资源进行权限检查。当前服务网格支持全局、命名空间和工作负载级别的授权策略控制，同时支持集成外部授权服务以及 OPA 策略引擎，提供丰富的授权能力。

3.7.2 工作负载身份

CSM 服务网格中的工作负载都有一个身份信息，这个信息主要用于网格内服务之间通信时实现身份认证和基于身份的访问授权。工作负载身份信息由服务网格自动生成和维护，基于 Service Account 实现；CSM 服务网格中的工作负载身份信息符合 SPIFFE 标准，格式如下：

```
spiffe://{trust-domain}/ns/{namespace}/sa/{service-account}
```

在服务网格控制台 网格安全中心 -> 工作负载身份 菜单下，选择 namespace 可以看到该命名空间下的工作负载身份信息，如下图：



3.7.3 对等身份认证

对等身份认证（PeerAuthentication）策略定义了 sidecar 之间通信的 TLS 模式，当前支持三种模式：

PERMISSIVE：宽松模式，该模式下，sidecar 将接受明文和双向 TLS 加密通信

STRICT：严格模式，该模式下，sidecar 只接受双向 TLS 加密通信

DISABLE：禁用双向 TLS；一般情况下不应该使用该模式

如下 PeerAuthentication 策略定义了访问 foo 命名空间下的服务都必须采用双向 TLS 认证模

式

apiVersion: security.istio.io/v1beta1

kind: PeerAuthentication

metadata:

name: default

namespace: foo

spec:

mtls:

mode: STRICT

创建对等身份认证

1. 从服务网格控制台选择菜单 网络安全中心 -> 对等身份认证
2. 选择命名空间，默认展示当前命名空间下的对等身份认证策略，选择左上角 创建对等身份认证
3. 根据模板编辑，保存即可

修改对等身份认证

1. 从服务网格控制台选择菜单 网络安全中心 -> 对等身份认证
2. 选择命名空间，默认展示当前命名空间下的对等身份认证策略，选择操作栏的编辑功能
3. 编辑保存即可

删除对等身份认证

1. 从服务网格控制台选择菜单 网络安全中心 -> 对等身份认证
2. 选择命名空间，默认展示当前命名空间下的对等身份认证策略，选择操作栏的删除功能删除指定的策略配置

PeerAuthentication 配置说明：

字段	类型	必选	说明
selector	WorkloadSelector	No	工作负载选择器，根据标签选择策略生效的工作负载
mtls	MutualTLS	No	Mtls 配置，MutualTLS.mode 可选值： UNSET：未定义，默认继承父层级配置（命名空间级或者全局），如果父层级不存在则默认为 PERMISSIVE DISABLE：禁用 mTLS 认证，采用明文传输 PERMISSIVE：同时支持 mTLS 和明文 STRICT：只支持 mTLS 模式
portLevelMtls	map<uint32, MutualTLS>	No	端口级对等身份认证策略

3.7.4 请求身份认证

请求身份认证 (RequestAuthentication) 定义了终端用户请求网格服务时候的身份认证策略；如果认证信息非法，请求将被拦截；默认情况下，如果不带身份认证信息，请求将被放过，可以配置特定授权策略要求请求的身份信息不能为空来拦截身份认证信息为空的请求。

请求身份认证配置粒度

请求身份认证策略支持三种配置粒度，全局、命名空间和工作负载级；当策略的命名空间为系统命名空间时（默认为 istio-system），策略为全局生效；当策略命名空间不是系统命名空间，且没有选择工作负载，策略将只在当前命名空间生效并覆盖全局策略；当策略在非系统命名空间，且选择了工作负载，则只对指定工作负载生效。

下面的配置中 RequestAuthentication 定义了 foo 命名空间下匹配 app: httpbin 标签的工作负载使用的 jwt 认证策略，同时 AuthorizationPolicy 定义了对请求来源的匹配，要求认证信息不能为空

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  jwtRules:
    - issuer: "issuer-foo"
      jwksUri: https://example.com/.well-known/jwks.json
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  rules:
    - from:
      - source:
          requestPrincipals: ["*"]
```

创建请求身份认证

1. 进入网格控制台，选择 网络安全中心 -> 请求身份认证菜单
2. 选择命名空间，列表页会展示当前命名空间下的请求身份认证策略
3. 选择创建

修改请求身份认证

1. 进入网格控制台，选择 网络安全中心 -> 请求身份认证菜单
2. 选择命名空间，列表页会展示当前命名空间下的请求身份认证策略，选择操作栏的编辑选项，修改策略，保存即可

删除请求身份认证

1. 进入网格控制台，选择 网络安全中心 -> 请求身份认证菜单
2. 选择命名空间，列表页会展示当前命名空间下的请求身份认证策略，选择操作栏的删除选项，删除选定的策略即可

RequestAuthentication 配置说明：

字段	类型	必选	说明
selector	WorkloadSelector	No	工作负载选择器，选择策略生效的工作负载
jwtRules	[]JWTRule	No	JWT 配置列表，表示当前工作负载所支持的 jwt 配置（不支持一个请求带多个 jwt 认证信息的情况）

JWTRule

字段	类型	必选	说明
issuer	string	Yes	Jwt 签发方
audiences	[]string	No	Jwt 接收方列表，jwt 中包含其中任意一个接收者都可以通过验证；
jwtUri	string	No	验证 jwt 签名的公钥 URL，jwksUri 和 jwks 只能有一个生效
jwks	string	No	验证 jwt 签名的公钥，jwksUri 和 jwks 只能有一个生效
fromHeaders	[]JWTHeader	No	提取 Jwt 信息的头部字段
fromParams	[]string	No	提取 Jwt 信息的 query 字段
outputPayloadToHeader	string	No	向后端透传 payload 的头部字段名称
forwardOriginalToken	bool	No	设置为 true 时，原始的 jwt 将被透传到后端
outputClaimToHeaders	[]ClaimToHeader	No	定义 payload 里面的字段透传到后端的头部信息

3.7.5 授权策略

授权策略 (AuthorizationPolicy) 提供全局、命名空间级工作负载级别的访问控制，支持工作负载之间及终端用户对网格内的工作负载的访问控制策略。

授权策略配置粒度

授权策略支持三种配置粒度，全局、命名空间和工作负载级；当策略的命名空间为系统命名空间时（默认为 istio-system），策略为全局生效；当策略命名空间不是系统命名空间，且没有选择工作负载，策略将只在当前命名空间生效并覆盖全局策略；当策略在非系统命名空间，且选择了工作负载，则只对指定工作负载生效。

下面的配置定义了对 foo 命名空间下的服务的授权策略：

1. 请求方的 service account 是 cluster.local/ns/default/sa/sleep 或者命名空间是 test
2. 只能对 foo 命名空间下的 GET /info*接口或者 POST /data 接口
3. 请求必须经过 jwt 认证，且 iss 必须是 <https://accounts.google.com>

其他情况全部拒绝访问

apiVersion: security.istio.io/v1beta1

kind: AuthorizationPolicy

metadata:

name: httpbin

namespace: foo

spec:

action: ALLOW

rules:

- from:

- source:

principals: ["cluster.local/ns/default/sa/sleep"]

- source:

namespaces: ["test"]

to:

- operation:

methods: ["GET"]

paths: ["/info*"]

- operation:

methods: ["POST"]

paths: ["/data"]

when:

- key: request.auth.claims[iss]

values: ["https://accounts.google.com"]

创建授权策略

1. 进入服务网格控制台，选择 网格安全中心 -> 授权策略，选择命名空间，列表页默认展示当前命名空间下的授权策略
2. 选择左上角 使用 YAML 创建，选择配置模板，编辑保存策略即可

修改授权策略

1. 进入服务网格控制台，选择 网格安全中心 -> 授权策略，选择命名空间，列表页默认展示当前命名空间下的授权策略
2. 选择右侧操作栏的修改，编辑保存策略即可

删除授权策略

1. 进入服务网格控制台，选择 网格安全中心 -> 授权策略，选择命名空间，列表页默认展示当前命名空间下的授权策略
2. 选择右侧操作栏的删除，即可删除选中的策略

AuthorizationPolicy 配置说明：

字段	类型	必选	说明
selector	WorkloadSelector	No	工作负载选择器，选择策略生效的工作负载
rules	[]Rule	No	一组匹配规则，至少匹配其中一个才认为是匹配；

			规则列表为空时，默认不匹配；当 Action 为 ALLOW 时，实际行为是所有的请求都被拒绝
action	Action	No	授权策略的行为，支持以下可选值： ALLOW：如果匹配则允许访问，action 默认值 DENY：如果匹配则拒绝访问 AUDIT：匹配的请求将被审计 CUSTOM：使用外部授权服务对请求进行授权
provider	ExtensionProvider (oneof)	No	指定外部授权服务，需要跟 action=CUSTOM 时一起使用；引用 MeshConfig 中的 extension provider

Rule：定义具体的授权规则

字段	类型	必选	说明
from	[]From	No	请求源匹配
to	[]To	No	对目标的操作规则
when	[]Condition	No	其他匹配条件

Source：对应 From.source 字段，定义请求源的匹配规则，所有条件是 and 关系（需要同时满足）

字段	类型	必选	说明
principals	[]string	No	匹配请求源端的 SPIFFE 信息，需要开启 mTLS
notPrincipals	[]string	No	反向匹配请求源端的 SPIFFE 信息
requestPrincipals	[]string	No	匹配请求源 jwt 信息，格式为<ISS>/<SUB>，需要开启 RequestAuthentication 策略
notRequestPrincipals	[]string	No	反向匹配请求源 jwt 信息
namespaces	[]string	No	匹配请求源的命名空间，需要开启 mTLS
notNamespaces	[]string	No	反向匹配请求源的命名空间
ipBlocks	[]string	No	匹配请求源的 ip
notIpBlocks	[]string	No	反向匹配请求源的 ip
remoteIpBlocks	[]string	No	匹配请求的 X-Forwarded-For 头部
notRemoteIpBlocks	[]string	No	反向匹配请求的 X-Forwarded-For 头部

Operation：对应 To.operation 字段，定义了请求对目标端的操作信息

字段	类型	必选	说明
hosts	[]string	No	匹配请求的 HTTP host，仅用于 HTTP 服务
notHosts	[]string	No	反向匹配请求的 HTTP host
ports	[]string	No	匹配目标端口
notPorts	[]string	No	反向匹配目标端口
methods	[]string	No	匹配请求方法，仅用于 HTTP 服务
notMethods	[]string	No	反向匹配请求方法
paths	[]string	No	匹配请求路径，仅用于 HTTP 服务；对于 gRPC 服务，路径为“/package.service/method”这种形式
notPaths	[]string	No	反向匹配请求路径

Condition：定义了其他匹配条件

字段	类型	必选	说明
key	string	Yes	匹配的 key

values	[]string	No	Value 匹配列表, values 和 notValues 至少配置一个
notValues	[]string	No	Value 反向匹配列表, values 和 notValues 至少配置一个

注：存在多个授权策略的情况下优先匹配 action=CUSTOM 的策略，其次匹配 action=DENY 的策略，最后匹配 action=ALLOW 的策略，如果中间匹配到拒绝的场景，则拒绝请求，否则继续执行后续策略。

3.7.6 自定义授权服务

CSM 服务网格支持接入第三方鉴权服务，用于网格内的服务鉴权，当前支持定义 HTTP 协议和 gRPC 协议的自定义鉴权服务。

添加自定义授权服务

1. 从服务网格控制台 选择 网络安全中心 -> 自定义授权服务菜单，选择创建
2. 填写参数，提交即可

参数说明如下：

配置	说明
协议	调用外部授权服务的协议，当前支持 HTTP 和 gRPC
名称	唯一标识一个外部授权服务
服务地址	外部授权服务的地址
服务端口	外部授权服务的端口号
超时时间	调用外部授权服务时的超时时间，单位秒

针对 HTTP 协议的外部授权服务支持以下额外配置选项：

配置	说明
鉴权服务不可用时放行请求	打开此开关后，访问鉴权服务出现异常时将放行请求
在鉴权请求中携带 header	将请求中指定的头部带到鉴权服务
在鉴权请求中新增 header	在发往鉴权服务的请求中新增头部
鉴权通过时覆盖 Header	使用鉴权请求 Response 中 Header 覆盖发往目标服务的请求中的 Header
鉴权失败时覆盖 Header	使用鉴权请求 Response 中 Header 覆盖 Response 中的 Header
在鉴权请求中携带请求 Body	打开此开关后，将在访问鉴权服务时携带请求 Body
鉴权请求携带 Body 的最大长度(Byte)	限制发往鉴权服务的 Body 大小
允许将不完整消息发送至鉴权服务	在请求 Body 超出最大长度限制时，若不启用该选项则将拒绝请求并返回 HTTP 413

针对 gRPC 协议的外部授权服务支持以下额外配置选项：

配置	说明
鉴权服务不可用时放行请求	打开此开关后，访问鉴权服务出现异常时将放行请求
在鉴权请求中携带请求 Body	打开此开关后，将在访问鉴权服务时携带请求 Body

鉴权请求携带 Body 的最大长度(Byte)	限制发往鉴权服务的 Body 大小
允许将不完整消息发送至鉴权服务	在请求 Body 超出最大长度限制时，若不启用该选项则将拒绝请求并返回 HTTP 413
鉴权请求 Body 编码方式及存放位置	当前支持两种选项： UTF8 String(Body)：授权请求将编码为 UTF8 字符串放到请求 body 字段 RawBytes(RawBody)：授权请求将编码为原始字节串放到请求的 raw_body 字段

修改自定义授权服务

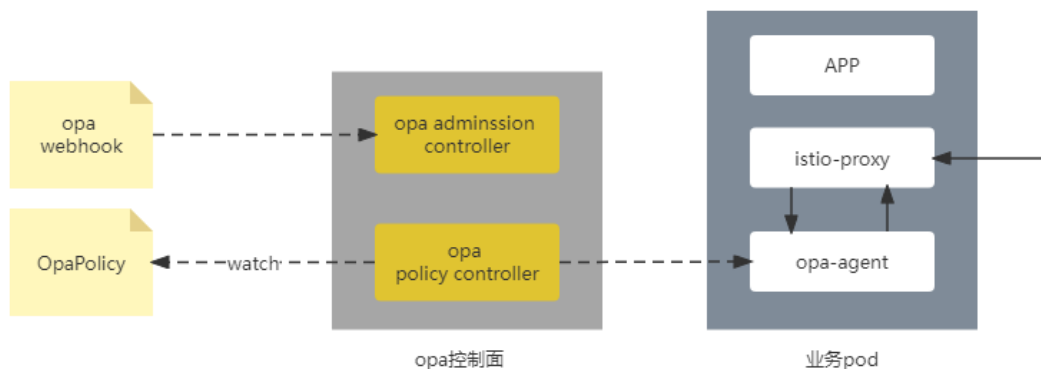
1. 从服务网格控制台 选择 网络安全中心 -> 自定义授权服务菜单，默认会展示当前定义的所有外部授权服务
2. 选择要编辑的服务编辑即可

删除自定义授权服务

1. 从服务网格控制台 选择 网络安全中心 -> 自定义授权服务菜单，默认会展示当前定义的所有外部授权服务
2. 选择要编辑的服务删除即可

3.7.7 OPA 策略

OPA (Open Policy Agent) 是一个通用的策略引擎，通过声明式的 Rego 语言实现丰富的授权策略。CSM 服务网格提供一键集成 OPA 引擎功能，整体架构如下：

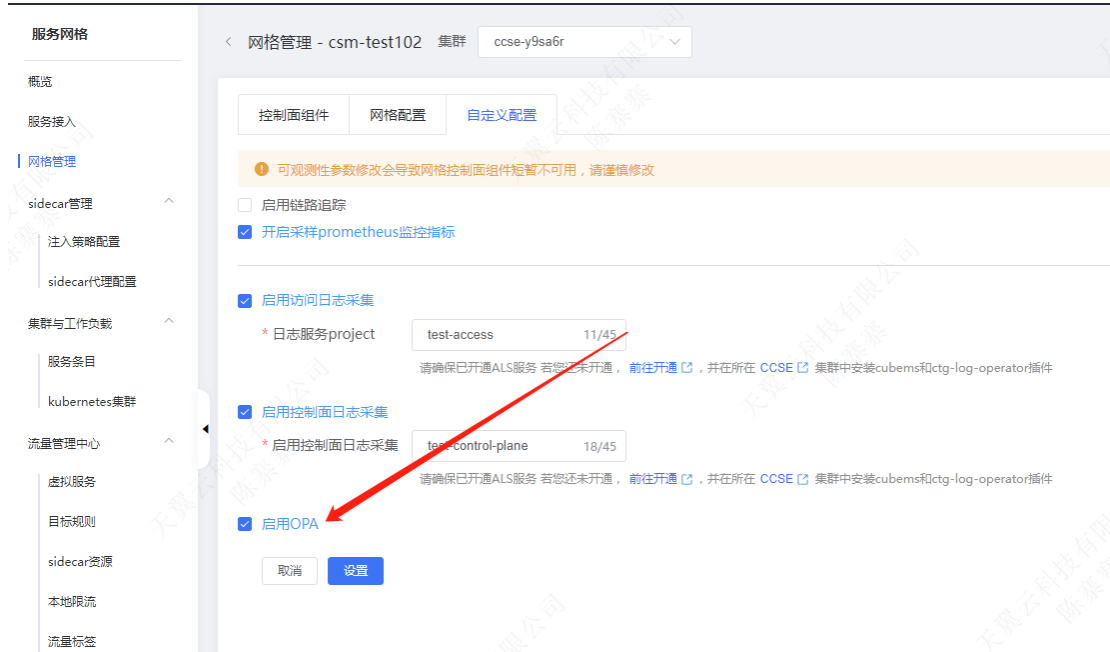


其中 OPA 控制面包括 OPA webhook 服务，用于实现 OPA sidecar 的注入；OPA policy controller 监听 OPA 策略 CRD，用于实现 OPA 策略分发；业务 pod 内除了业务容器和 istio-proxy 之外还会注入 OPA sidecar，外部请求 pod 时流量被 istio-proxy 拦截，根据定义的授权策略请求 OPA sidecar，实现对请求的授权。

注意：OPA sidecar 当前使用 8181 端口管理 OPA 策略，使用 9191 端口提供外部鉴权服务，业务 pod 注意避开这两个端口

开启 OPA 功能

使用 OPA 功能时首先要打开开关，进入服务网格控制台，选择网络管理 -> 自定义配置，勾选 启用 OPA 即可



关闭 OPA 功能

进入服务网格控制台，选择网络管理 -> 自定义配置，取消勾选 启用 OPA 即可

进入服务网格控制台，选择网络管理 -> OPA 开关，关闭即可

OPA 策略管理

CSM 服务网格采用自定义 CRD (OpaPolicy) 方式管理 OPA 策略，主要包括 OPA 策略生效的工作负载选择以及 OPA 策略 (Rego)，下面的配置对 default 命名空间下标签包含 version: v1 的工作负载生效，策略中定义了 guest 和 admin 两个角色，分别给两个角色定义了访问权限，同时还定义了两个用户 bob 和 alice，分别赋予了特定角色；策略执行时会从请求中解析出用户，结合用户的权限和请求信息对访问进行鉴权。

```
apiVersion: opacontroller.k8s.io/v1
kind: OpaPolicy
metadata:
  name: object-policy
  namespace: default
spec:
  policy: |-
```

```

package istio.authz
import input.attributes.request.http as http_request
import input.parsed_path

allow {
  roles_for_user[r]
  required_roles[r]
}

roles_for_user[r] {
  r := user_roles[user_name][_]
}

required_roles[r] {
  perm := role_perms[r][_]
  perm.method = http_request.method
  perm.path = http_request.path
}

user_name = parsed {
  [_, encoded] := split(http_request.headers.authorization, " ")
  [parsed, _] := split(base64url.decode(encoded), ":")
}

user_roles = {
  "alice": ["guest"],
  "bob": ["admin"]
}

role_perms = {
  "guest": [
    {"method": "GET", "path": "/productpage"},
  ],
  "admin": [
    {"method": "GET", "path": "/productpage"},
    {"method": "GET", "path": "/api/v1/products"},
  ],
}

workloadSelector:
  labels:
    version: v1

```

OpaPolicy 配置说明

字段	类型	必选	说明
----	----	----	----

workloadSelect or	WorkloadSelect or	N o	选择策略生效的工作负载
policy	string	N o	Rego 策略，配置语法参考： https://www.openpolicyagent.org/docs/latest/policy-language/

创建 OPA 策略

1. 进入服务网格控制台，选择 网络安全中心 -> OPA 策略 菜单，选择要配置的命名空间
2. 选择使用 YAML 创建

修改 OPA 策略

1. 进入服务网格控制台，选择 网络安全中心 -> OPA 策略 菜单，选择要配置的命名空间
2. 列表页会展示当前命名空间下的 OPA 策略，找到要修改的策略，选择右侧操作栏的修改，编辑提交即可

删除 OPA 策略

1. 进入服务网格控制台，选择 网络安全中心 -> OPA 策略 菜单，选择要配置的命名空间
2. 列表页会展示当前命名空间下的 OPA 策略，找到要修改的策略，选择右侧操作栏的删除即可

3.7.8 授权控制示例

使用自定义授权服务

前提条件：

1. 开通 CCSE（容器云服务引擎）
2. 开通服务网格实例

操作步骤：

1. 创建测试命名空间

```
kubectl create ns foo
```

 # 打开 sidecar 自动注入

```
kubectl label ns foo istio-injection=enabled
```

2. 部署 sleep 和 httpbin 应用

apiVersion: v1

kind: ServiceAccount

metadata:

name: sleep

apiVersion: v1

kind: Service

metadata:

name: sleep

labels:

app: sleep

service: sleep

spec:

ports:

- port: 80

name: http

selector:

app: sleep

apiVersion: apps/v1

kind: Deployment

metadata:

name: sleep

spec:

replicas: 1

selector:

matchLabels:

app: sleep

template:

metadata:

labels:

app: sleep

spec:

terminationGracePeriodSeconds: 0

serviceAccountName: sleep

containers:

- name: sleep

image: registry-vpc-crs-huadong1.ctyun.cn/library/curl

command: ["/bin/sleep", "infinity"]

imagePullPolicy: IfNotPresent

volumeMounts:

- mountPath: /etc/sleep/tls

name: secret-volume

volumes:

- name: secret-volume

secret:

secretName: sleep-secret

optional: true

3. 部署外部授权服务

apiVersion: v1

kind: Service

metadata:

name: ext-authz

labels:

app: ext-authz

spec:

ports:

- name: http

port: 8000

targetPort: 8000

- name: grpc

port: 9000

targetPort: 9000

selector:

app: ext-authz

apiVersion: apps/v1

kind: Deployment

metadata:

name: ext-authz

spec:

replicas: 1

selector:

matchLabels:

app: ext-authz

template:

metadata:

labels:

app: ext-authz

spec:

containers:

- image: registry-vpc-crs-huadong1.ctyun.cn/library/ext-authz:1.16.2

imagePullPolicy: IfNotPresent

name: ext-authz

ports:

- containerPort: 8000

- containerPort: 9000

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: registry-vpc-crs-huadong1.ctyun.cn/library/httpbin
        imagePullPolicy: IfNotPresent
        name: httpbin
        ports:
        - containerPort: 80

```

部署完成后，在 foo 命名空间下看到 3 个服务

```

[~] kubectl get po -n foo

```

NAME	READY	STATUS	RESTARTS	AGE
ext-authz-74b6fd5584-1fkcx	2/2	Running	0	36m
httpbin-698cc5f69-n4mv2	2/2	Running	0	49m
sleep-9454cc476-p8rfk	2/2	Running	0	49m

不使用授权策略的情况下，验证从 sleep 应用访问 httpbin 应用没有被拦截（返回状态码 200）：

```

kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})"
-c sleep -n foo -- curl http://httpbin.foo:8000/ip -s -o /dev/null -w "%{http_code}\n"

```


查看外部授权服务已经启动，HTTP 和 gRPC 授权服务分别监听 8000 和 9000 端口：

```
kubectl logs "$(kubectl get pod -l app=ext-authz -n foo -o jsonpath={.items..metadata.name})" -n foo -c ext-authz
```

```
2023/08/14 11:26:54 Starting HTTP server at [::]:8000
```

```
2023/08/14 11:26:54 Starting gRPC server at [::]:9000
```

4. 添加外部授权服务

将上面的 HTTP 和 gRPC 服务添加到服务网格的外部授权服务内

5. 定义授权策略&验证访问

定义如下授权策略，对 httpbin 应用的/headers 路径的请求将被转发到第三方授权服务进行验证：

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: ext-authz
spec:
  selector:
    matchLabels:
      app: httpbin
  action: CUSTOM
  provider:
    # The provider name must match the extension provider defined in the mesh
    # config.
    # You can also replace this with sample-ext-authz-http to test the other external
    # authorizer definition.
    name: sample-ext-authz-grpc
  rules:
    # The rules specify when to trigger the external authorizer.
    - to:
      - operation:
        paths: ["/headers"]
```

从 sleep 应用请求 httpbin 的/headers 路径，由于请求头带了"x-ext-authz: deny"，请求被拦截：

```
kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})"
-c sleep -n foo -- curl "http://httpbin.foo:8000/headers" -H "x-ext-authz: deny" -s
denied by ext_authz for not found header `x-ext-authz: allow` in the request
```

修改请求，带上"x-ext-authz: allow"头部，请求放行：

```
kubectl exec "$(kubectl get pod -l app=sleep -n foo -o jsonpath={.items..metadata.name})"
-c sleep -n foo -- curl "http://httpbin.foo:8000/headers" -H "x-ext-authz: allow" -s
```

```
{
  "headers": {
    "Accept": "*/*",
```

```

"Host": "httpbin.foo:8000",
"User-Agent": "curl/8.2.1",
"X-B3-Parentspanid": "eb42a8165099e7db",
"X-B3-Sampled": "1",
"X-B3-Spanid": "cd6540ba1cfd9e8c",
"X-B3-Traceid": "e7e15cc10dc66630eb42a8165099e7db",
"X-Envoy-Attempt-Count": "1",
"X-Ext-Authz": "allow",
"X-Ext-Authz-Additional-Header-Override": "grpc-additional-header-override-value",
"X-Ext-Authz-Check-Received": "source:{address:{socket_address:{address:\"10.1.0.25\"
port_value:37654}} principal:\"spiffe://cluster.local/ns/foo/sa/sleep\"}
destination:{address:{socket_address:{address:\"10.1.0.24\" port_value:80}}
principal:\"spiffe://cluster.local/ns/foo/sa/httpbin\"} request:{time:{seconds:1692015383
nanos:990298000} http:{id:\"7462237371770661564\" method:\"GET\"
headers:{key:\":authority\" value:\"httpbin.foo:8000\"} headers:{key:\":method\"
value:\"GET\"} headers:{key:\":path\" value:\"/headers\"} headers:{key:\":scheme\"
value:\"http\"} headers:{key:\":accept\" value:\"*/.*\"} headers:{key:\":user-agent\"
value:\"curl/8.2.1\"} headers:{key:\":x-b3-sampled\" value:\"1\"} headers:{key:\":x-b3-
spanid\" value:\"eb42a8165099e7db\"} headers:{key:\":x-b3-traceid\"
value:\"e7e15cc10dc66630eb42a8165099e7db\"} headers:{key:\":x-envoy-attempt-count\"
value:\"1\"} headers:{key:\":x-ext-authz\" value:\"allow\"} headers:{key:\":x-forwarded-client-
cert\"
value:\"By=spiffe://cluster.local/ns/foo/sa/httpbin;Hash=c32db24acfa670a8bbe46f0897ebb7
0b9ccc0e630ee32afcd1ec037d6616e6c5;Subject=\\\"\\\";URI=spiffe://cluster.local/ns/foo/sa
/sleep\"} headers:{key:\":x-forwarded-proto\" value:\"http\"} headers:{key:\":x-request-id\"
value:\"aba7b68c-6bee-9d58-b163-7454075c6ece\"} path:\"/headers\"
host:\"httpbin.foo:8000\" scheme:\"http\" protocol:\"HTTP/1.1\"}} metadata_context:{},
"X-Ext-Authz-Check-Result": "allowed",
"X-Forwarded-Client-Cert":
"By=spiffe://cluster.local/ns/foo/sa/httpbin;Hash=c32db24acfa670a8bbe46f0897ebb70b9cc
c0e630ee32afcd1ec037d6616e6c5;Subject=\\\"\\\";URI=spiffe://cluster.local/ns/foo/sa/sleep"
}
}

```

使用 OPA 策略引擎实现访问控制

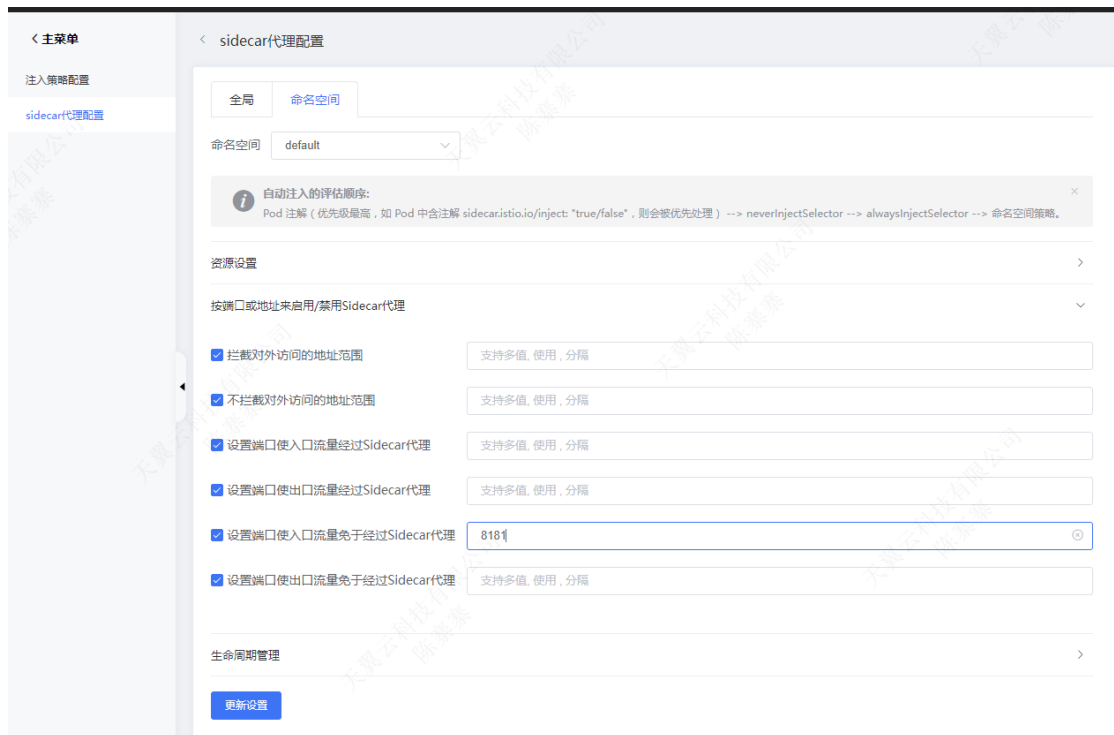
前提条件：

1. 开通 CCSE（容器云服务引擎）
2. 开通服务网格实例

操作步骤：

1. 在服务网格控制台->网络管理->OPA 功能开关菜单下打开 OPA 开关，主要是安装 OPA 控制面组件
2. 在 sidecar 管理->sidecar 代理配置菜单下选择命名空间 tab，选择我们验证功能要用的命名空间，这里选择 default，配置 sidecar 入流量不拦截 8181 端口（OPA agent 的配

置端口)



3. 在 default 命名空间下部署以下 yaml，用于 OPA agent 相关功能的准备；其中 opa-istio-config 这个 config map 定义了 OPA agent 启动的配置；opa-policy 定义了默认的 OPA 策略；ext-authz 这个 EnvoyFilter 定义了当前命名空间下的外部授权策略，请求进入 sidecar 之后会执行这个外部授权插件，调用 OPA agent 服务

```
apiVersion: v1

kind: ConfigMap

metadata:
  name: opa-istio-config

data:
  config.yaml: |
    plugins:
      envoy_ext_authz_grpc:
        addr: :9191
        path: istio/authz/allow
    decision_logs:
      console: true
  ---
apiVersion: v1
```

```

kind: ConfigMap
metadata:
  name: opa-policy
data:
  policy.rego: |
    package istio.authz

    import input.attributes.request.http as http_request
    import input.parsed_path

    default allow = false

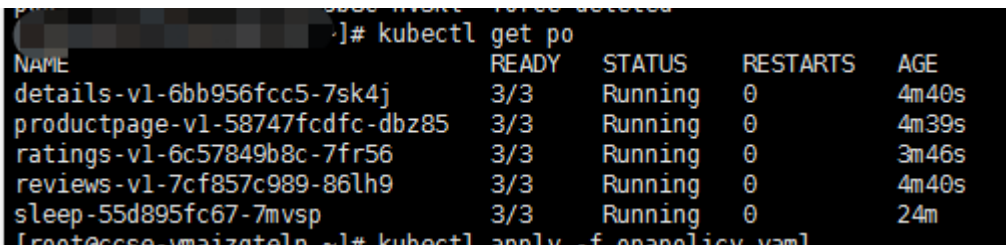
```

4. 给 default 命名空间打上标签，自动注入 istio sidecar 和 OPA sidecar

```
kubectl label namespace default opa-istio-injection="enabled"
```

```
kubectl label namespace default istio-injection="enabled"
```

5. 部署 bookinfo 应用和 sleep 应用，可以看到 pod 里除了业务容器之外还有两个容器，分别是 istio sidecar 和 OPA sidecar，OPA sidecar 用于实现外部授权服务



```

[roo@ccse-vmazoteln ~]# kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
details-v1-6bb956fcc5-7sk4j        3/3     Running   0           4m40s
productpage-v1-58747fcdcf-dbz85    3/3     Running   0           4m39s
ratings-v1-6c57849b8c-7fr56       3/3     Running   0           3m46s
reviews-v1-7cf857c989-86lh9       3/3     Running   0           4m40s
sleep-55d895fc67-7mvsp            3/3     Running   0           24m

```

6. 最后，我们需要定义外部授权服务和授权策略 (AuthorizationPolicy)，其中外部授权服务就是我们注入的 OPA sidecar，授权服务的地址总是 127.0.0.1:9191；为了能够对 OPA sidecar 授权服务寻址，我们还需要定义一个 ServiceEntry，如下：

```

apiVersion: networking.istio.io/v1beta1
kind: ServiceEntry
metadata:
  name: ext-authz
  namespace: opa-istio
spec:
  hosts:

```

- "ext-authz.opa-istio.internal"

ports:

- number: 9191

name: grpc

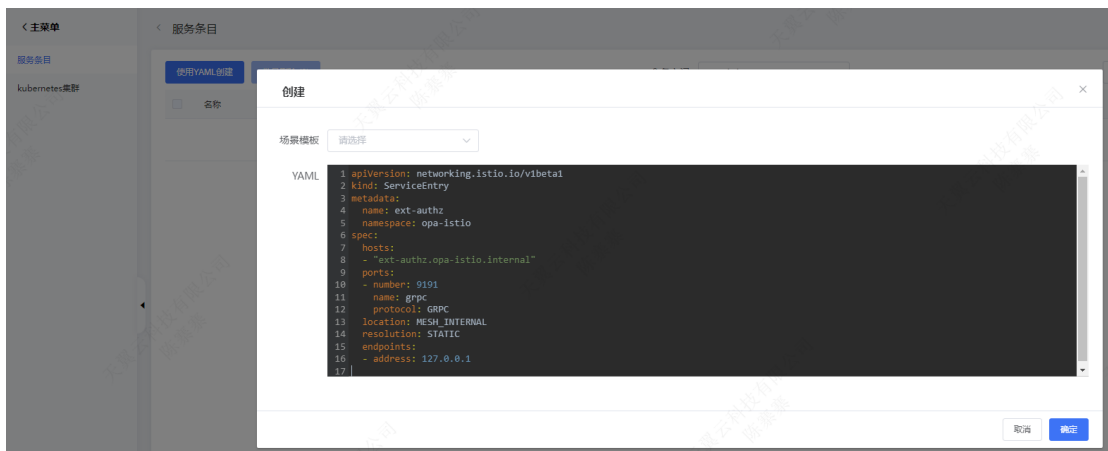
protocol: GRPC

location: MESH_INTERNAL

resolution: STATIC

endpoints:

- address: 127.0.0.1



添加外部授权服务，可以根据业务请求 body 大小调整相关参数配置，如下：



定义 AuthorizationPolicy 授权策略，注意引用的外部授权服务需要和上面定义的外部授权服务名称一致，可以根据业务的需要执行 OPA 外部授权策略，如下示例对匹配标签 app: productpage 的：

```
apiVersion: security.istio.io/v1beta1

kind: AuthorizationPolicy

metadata:
  name: ext-authz

spec:
  selector:
    matchLabels:
      app: productpage
  action: CUSTOM
  provider:
    # The provider name must match the extension provider defined in the mesh
    config.
    # You can also replace this with sample-ext-authz-http to test the other external
    authorizer definition.
    name: opa-ext-authz-grpc

  rules:
    # The rules specify when to trigger the external authorizer.
    - to:
      - operation:
        paths: ["/*"]
```

7. 上面的配置中，我们通过 workloadSelector 指定对 productpage 应用进行访问授权，我们从 sleep 应用发起请求，访问 productpage，此时采用默认 OPA 策略，请求总是被拒绝

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name} -n
default) -c istio-proxy -n default -- curl
http://productpage.default:9080/api/v1/products --user bob:password -o /dev/null -s
-w '%{http_code}\n'

403
```

8. 配置 OPA 策略：以下 OPA 策略定义了 guest 和 admin 角色，guest 和 admin 可以使用 GET 方法访问/productpage 路径，admin 另外还可以使用 GET 方法访问

/api/v1/products 路径；外部访问时会先解析出用户名，获取用户角色，进一步获取用户访问权限，并与请求进行比对，满足条件则放过，否则拦截

```
apiVersion: opacontroller.k8s.io/v1
kind: OpaPolicy
metadata:
  name: object-policy
  namespace: default
spec:
  policy: |-
    package istio.authz

    import input.attributes.request.http as http_request
    import input.parsed_path

    allow {
      roles_for_user[r]
      required_roles[r]
    }

    roles_for_user[r] {
      r := user_roles[user_name][_]
    }

    required_roles[r] {
      perm := role_perms[r][_]
      perm.method = http_request.method
      perm.path = http_request.path
    }

    user_name = parsed {
      [_, encoded] := split(http_request.headers.authorization, " ")
```

```

    [parsed, _] := split(base64url.decode(encoded), ":")
  }

  user_roles = {
    "alice": ["guest"],
    "bob": ["admin"]
  }

  role_perms = {
    "guest": [
      {"method": "GET", "path": "/productpage"},
    ],
    "admin": [
      {"method": "GET", "path": "/productpage"},
      {"method": "GET", "path": "/api/v1/products"},
    ],
  }

  workloadSelector:

  labels:

  version: v1

```

9. 验证

以 bob 的身份访问 `/api/v1/products` 和 `/productpage`，由于 bob 是 admin 权限，两个路径都可以访问，返回 200

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name} -n
default) -c istio-proxy -n default -- curl http://productpage.default:9080/api/v1/products -
-user bob:password -o /dev/null -s -w '%{http_code}\n'
```

n'

200

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name} -n
default) -c istio-proxy -n default -- curl http://productpage.default:9080/productpage --
user bob:password -o /dev/null -s -w '%{http_code}\n'
```


200

以 alice 的身份访问/api/v1/products 和/productpage, 由于 alice 是 guest 权限, 所以对 /api/v1/products 的访问会被拒绝, 返回 403 ; 对/productpage 的访问可以通过, 返回 200

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name} -n default) -c istio-proxy -n default -- curl http://productpage.default:9080/api/v1/products -user alice:password -o /dev/null -s -w '%{http_code}\n'
```

403

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name} -n default) -c istio-proxy -n default -- curl http://productpage.default:9080/productpage --user alice:password -o /dev/null -s -w '%{http_code}\n'
```

200

10. 修改 OPA 策略, 给 alice 加上 admin 权限

```
user_roles = {
  "alice": ["guest", "admin"],
  "bob": ["admin"]
}
```

重新验证以 alice 身份访问/api/v1/products, 返回 200

```
kubectl exec $(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name} -n default) -c istio-proxy -n default -- curl http://productpage.default:9080/api/v1/products --user alice:password -o /dev/null -s -w '%{http_code}\n'
```

200

使用 JWT 认证授权

服务网格中使用 JWT 实现对请求的身份认证, 进一步可以配置授权策略, 限制对请求的授权。

首先部署测试应用

参考以上示例部署 sleep 和 httpbin 应用, pod 列表如下:

```
[~]# kubectl get po -n bookinfo
NAME                READY   STATUS    RESTARTS   AGE
httpbin-85d76b4bb6-5rjtk  2/2     Running   0           2m1s
sleep-7fb478946b-44t5x  2/2     Running   0           4h49m
```

配置 JWT 认证策略:

```
apiVersion: security.istio.io/v1beta1
```

```
kind: RequestAuthentication
```

```
metadata:
```

```
name: "jwt-example"
namespace: bookinfo
spec:
  selector:
    matchLabels:
      app: httpbin
  jwtRules:
    - issuer: "testing@secure.istio.io"
      jwks: '{ "keys": [ { "e": "AQAB", "kid": "DHFbpolUqrY8t2zpA2qXfCmr5VO5ZEr4RzHU_-
envvQ", "kty": "RSA", "n": "xAE7eB6qugXyCAG3yhh7pkDkT65pHymX-
P7Kflupjf59vsdo91bSP9C8H07pSAGQO1MV_xFj9VswgsCg4R6otmg5PV2He95IzdHtOcU5DX
lg_pbhLdKXbi66GIVeK6ABZOUW3WYtnNHD-91gVuoeJT_DwtGGcp4ignkgXfkiEm4sw-
4sfb4qdt5oLbyVpmW6x9cfa7vs2WTfURiCrBoUqgBo_-
4WTiULmmHSGZHOjzwa8WtrtOQGSAFjlbno85jp6MnGGZPYZbDAa_b3y5u-
YpW7ypZrvD8BgtKVjgtQgZHLAGezMt0ua3DRrWnKqTZ0BJ_EyxOGuHJrLsn00fnMQ"}]}'
```

使用非法的 JWT 访问，可以看到返回了 401 错误：

```
kubectl exec "$(kubectl get pod -l app=sleep -n bookinfo -o
jsonpath={.items..metadata.name})" -c sleep -n bookinfo -- curl
"http://httpbin:8000/headers" -sS -o /dev/null -H "Authorization: Bearer invalidToken" -w
"%{http_code}\n"
401
```

不带 JWT 头部时会放过请求（返回 200）：

```
kubectl exec "$(kubectl get pod -l app=sleep -n bookinfo -o
jsonpath={.items..metadata.name})" -c sleep -n bookinfo -- curl
"http://httpbin:8000/headers" -sS -o /dev/null -w "%{http_code}\n"
200
```

创建授权策略，要求请求带有合法的 JWT 才允许访问：

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt
  namespace: bookinfo
spec:
  selector:
    matchLabels:
      app: httpbin
  action: ALLOW
  rules:
    - from:
      - source:
          requestPrincipals: ["testing@secure.istio.io/testing@secure.istio.io"]
```

再次不带 JWT 访问返回了 403：

```
kubectl exec "$(kubectl get pod -l app=sleep -n bookinfo -o
jsonpath={.items..metadata.name})" -c sleep -n bookinfo -- curl
"http://httpbin:8000/headers" -sS -o /dev/null -w "%{http_code}\n"
403
```

使用已经生成好的 token 验证访问，可以看到返回了 200 状态码：

```
TOKEN=
eyJhbGciOiJIUzI1NiIsImtpZCI6IkkRIRmJwb0lVcXJZOHQyenBBMnFYZkNtcjVWTzVaRXI0UnplV
V8tZW52dEiLCJ0eXAiOiJKV1QiLCJkaW50eXN0aW8uW8iLCJzZWVudWVlOiJ0ZX
MTUzMjM4OTcwMCwiaXNzIjoiaGVzZGluZ0BzZWN1cmUuaXN0aW8uW8iLCJzZWVudWVlOiJ0ZX
N0aW5nQHNIY3VyZS5pc3Rpbj5pbyJ9.CfNnxWP2tcnR9q0vxyxweaF3ovQYHYZl82hAUsn21b
wQd9zP7c-LS9qd_vpdLG4Tn1A15NxfCjp5f7QNBUo-
KC9PJqYpgGbaXhaGx7bEdFWjcwv3nZzvc7M__ZpaCERdwU7igUmJqYGBYQ51vr2njU9ZimyK
kfDe3axcyiBzde7G6dabliUosJwKOPcKIWPccCgefSj_GNfwlip3-SsFdIR7BtbVUcqR-yv-
XOxJ3Uc1MI0tz3uMiiZcyPV7sNCU4KRnemRIMHVOfuvHsU60_GhGbiSFzgPTAa9WTltbnarTbx
udb_YEOx12JiwYToeX0DCPb43W1tzlBxgm8NxUg
  kubectl exec "$(kubectl get pod -l app=sleep -n bookinfo -o
jsonpath={.items..metadata.name})" -c sleep -n bookinfo -- curl
"http://httpbin:8000/headers" -H "Authorization: Bearer $TOKEN" -sS -o /dev/null -w
"%{http_code}\n"
  200
```

修改授权策略，只允许 JWT 信息中 groups 字段包含 group1 的时候才允许访问，策略如下：

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt
  namespace: bookinfo
spec:
  selector:
    matchLabels:
      app: httpbin
  action: ALLOW
  rules:
  - from:
    - source:
      requestPrincipals: ["testing@secure.istio.io/testing@secure.istio.io"]
    when:
      - key: request.auth.claims[groups]
        values: ["group1"]
```

使用上述 TOKEN 访问返回 403：

```
kubectl exec "$(kubectl get pod -l app=sleep -n bookinfo -o
jsonpath={.items..metadata.name})" -c sleep -n bookinfo -- curl
"http://httpbin:8000/headers" -H "Authorization: Bearer $TOKEN" -sS -o /dev/null -w
"%{http_code}\n"
  403
```

更新 TOKEN

```
TOKEN=
eyJhbGciOiJIUzI1NiIsImtpZCI6IkkRIRmJwb0lVcXJZOHQyenBBMnFYZkNtcjVWTzVaRXI0UnplV
V8tZW52dEiLCJ0eXAiOiJKV1QiLCJkaW50eXN0aW8uW8iLCJzZWVudWVlOiJ0ZX
MTUzMjM4OTcwMCwiaXNzIjoiaGVzZGluZ0BzZWN1cmUuaXN0aW8uW8iLCJzZWVudWVlOiJ0ZX
N0aW5nQHNIY3VyZS5pc3Rpbj5pbyJ9.CfNnxWP2tcnR9q0vxyxweaF3ovQYHYZl82hAUsn21b
wQd9zP7c-LS9qd_vpdLG4Tn1A15NxfCjp5f7QNBUo-
KC9PJqYpgGbaXhaGx7bEdFWjcwv3nZzvc7M__ZpaCERdwU7igUmJqYGBYQ51vr2njU9ZimyK
kfDe3axcyiBzde7G6dabliUosJwKOPcKIWPccCgefSj_GNfwlip3-SsFdIR7BtbVUcqR-yv-
XOxJ3Uc1MI0tz3uMiiZcyPV7sNCU4KRnemRIMHVOfuvHsU60_GhGbiSFzgPTAa9WTltbnarTbx
udb_YEOx12JiwYToeX0DCPb43W1tzlBxgm8NxUg
```

```
Rpby5pbyIsInNjb3BlIjpbInNjb3BIMSlInNjb3BIMiJdLCJzdWl0Ij0ZXN0aW5nQHNIY3VyZS5pc
3RpbY5pbyJ9.EdJnEZSH6X8hcyEii7c8H5lnhgjB5dwo07M5oheC8Xz8mOilyg--
AHCFWHybM48reunF--
oGaG6IXVngCEpVF0_P5DwsUoBgpPmK1JOaKN6_pe9sh0ZwTtdgK_RP01Pul7kUdbOTlkuUi2
AO-
qUyOm7Art2POzo36DLQIUxv8Ad7NBOqfQaKjE9ndaPWT7aexUsBHxmgjGbz1SyLH879f7uH
YPbPKIpHU6P9S-
DaKnGLaEchnoKnov7ajhrEhGXAQRukhDPKUHO9L30oPlr5IJIIEQfHYtt6IZvINUGeLUcif3wpry1
R5tBXRicx2sXMq7LyuDremDbcNy_iE76Upg
```

再次访问返回 200 :

```
kubectl exec "$(kubectl get pod -l app=sleep -n bookinfo -o
jsonpath={.items..metadata.name})" -c sleep -n bookinfo -- curl
"http://httpbin:8000/headers" -H "Authorization: Bearer $TOKEN" -sS -o /dev/null -w
"%{http_code}\n"
200
```

3.8 服务监控

可观测概述

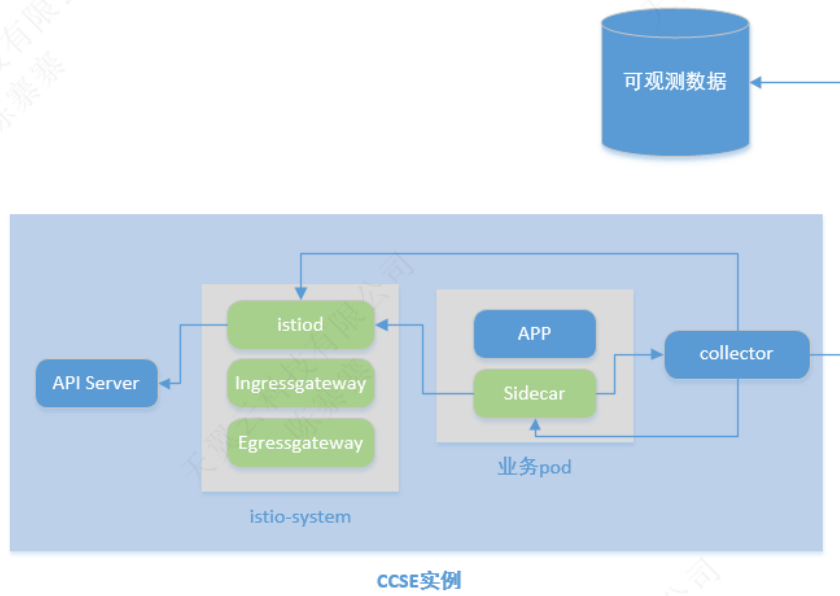
服务网格中的可观测主要包括指标监控、日志和调用链追踪，可观测能力对于分布式微服务系统至关重要，它可以帮助我们快速了解系统运行警效，定位系统问题，发现系统瓶颈。

天翼云服务网格中目前具备指标监控及控制面和数据面日志监控功能，指标监控主要包括：

控制面指标：当前主要用于监控网格实例控制面运行情况，以及告警等功能，当前仅用于后端监控，未开放给客户查看

业务指标监控：数据面 sidecar 采集到的指标，可以用于查看服务间调用的请求数、耗时等数据，提供监控看板查看

控制面&数据面日志：如果您开通了 ALS 日志服务并且在服务网格开启了控制面和数据面的日志采集，可以将控制面和数据面 sidecar 的日志采集到 ALS 日志服务，当前可以到 ALS 日志服务控制台选择相应的日志项目查看日志情况



指标监控

指标监控页面展示了网格内服务的访问数据，需要选择集群、命名空间和服务名称，可以查看服务的相关访问指标



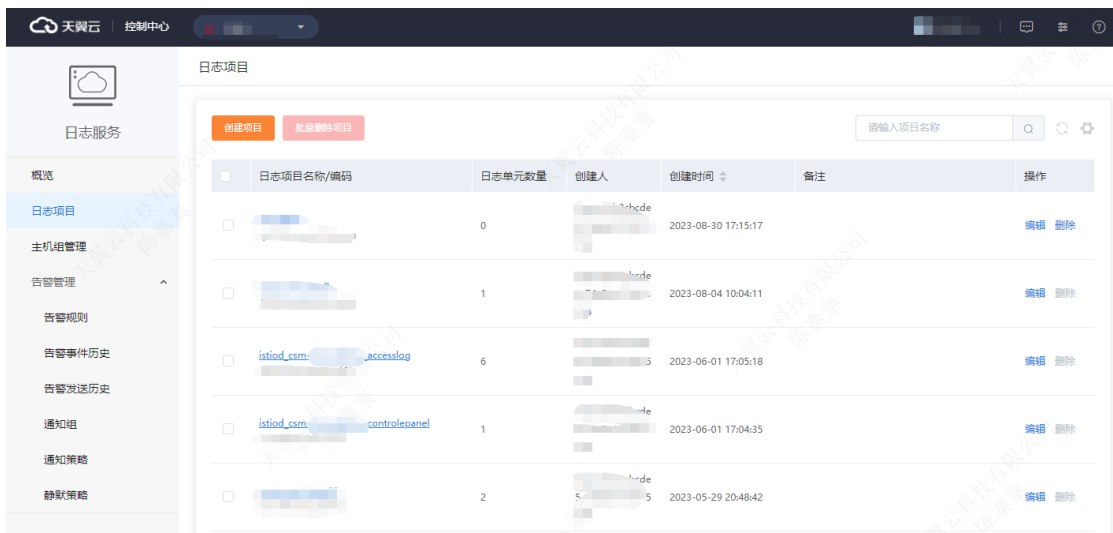
指标说明如下：

指标	说明
被调用次数	调用者视角调用当前所选服务的次数
被调用平均时延	调用者视角调用当前所选服务的平均耗时
被调用成功率	调用者视角调用当前服务的成功率（非 5XX 认为是调用成功）
收到请求总次数	当前服务视角收到的总请求数

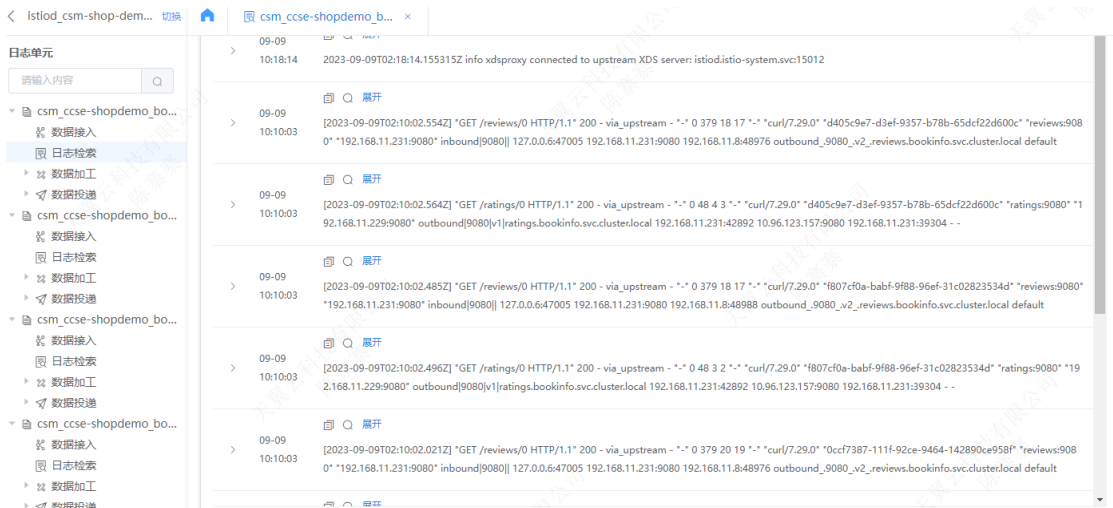
响应 4xx 码统计	调用者视角调用当前服务返回 4XX 的数量
响应 5xx 码统计	调用者视角调用当前服务返回 5XX 的数量
响应平均时延	当前服务视角被调用的平均耗时
响应成功率	当前服务视角被调用响应的成功率（非 5XX 认为是调用成功）
请求 Ops	当前服务视角每秒被调用的次数的时序图
请求成功率	调用者视角请求成功率的时序图（返回非 5XX 认为是成功）

日志

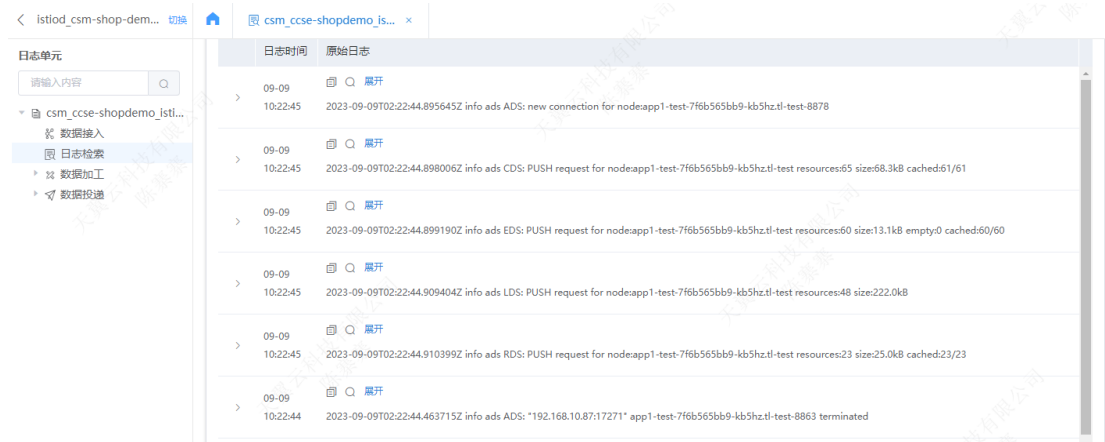
进入天翼云 ALS 日志服务控制台，选择日志项目，可以看到跟网格实例相关的日志项目，其中以 accesslog 结尾的是网格数据面日志项目，以 controplane 结尾的是网格控制面日志



选择数据面日志可以看到 istio sidecar 的访问日志，如下：



选择控制面日志可以看到控制面的日志：



4. 最佳实践

4.1 CSM 服务网格配置建议

本文介绍 CSM 服务网格相关资源配置的建议，避免因配置错误或者不合理导致的不符合预期的行为。

VirtualService

1. 虚拟服务资源需要配置到目标服务所在的命名空间下，不建议跨命名空间配置虚拟服务
2. 一个特定 host 的路由规则只定义在一个虚拟服务资源下，同 host 存在多个虚拟服务资源时将只会生效最新创建的一个
3. 虚拟服务资源下的 host（虚拟服务对应的 host 或者路由目标中的 host）建议配置成 FQDN（fully qualified domain name）；如果使用短名，istio 会根据虚拟服务资源定义的命名空间补全 host，可能与实际的服务名不一致。
4. 对于一个服务，同时存在 host 模糊匹配和精准匹配的虚拟服务时，将以精准匹配的定义为准
5. istio 会在没有匹配到路由规则时默认访问目标服务的所有子集，但是这样没有执行任何流量治理策略；所以我们建议总是为服务定义一个默认路由，并通过目标规则定义流量策略，这样可以充分利用 Envoy 的流量治理能力，保证服务的访问总是在掌控之中

DestinationRule

1. 业务访问中使用目标规则时的查找顺序是：
 - a) 先查找客户端的命名空间中是否有要访问的服务的目标规则定义
 - b) 再查找服务端命名空间中是否有要访问的服务的目标规则定义
 - c) 最后查找根命名空间中的目标规则定义

对于一次访问，如果目标规则没有定义在上面三个地方中，目标规则将不会生效，这里建议每个服务在自己所在的命名空间下定义目标规则

2. 目标规则中的 host 也建议使用 FQDN，避免不必要的错配
3. 对于一个服务，同时存在 host 模糊匹配和精准匹配的目标规则时，将以精准匹配的定义为准
4. 目标规则中可以针对某个服务的子集定义负载均衡、连接池、异常检测等策略，这些策略只有在虚拟服务中显式引用了该目标子集所产生的访问下才会生效，否则不生效

sidecar

1. sidecar 资源定义了 sidecar 代理的配置，每个命名空间只能有一个不带 workloadSelector 的 sidecar 资源，定义了该命名空间下默认的 sidecar 配置
2. 根命名空间下定义一个不带 workloadSelector 的 sidecar 资源，将对网格所有的命名空间生效；如果有命名空间级别的 sidecar 资源，则它的优先级更高

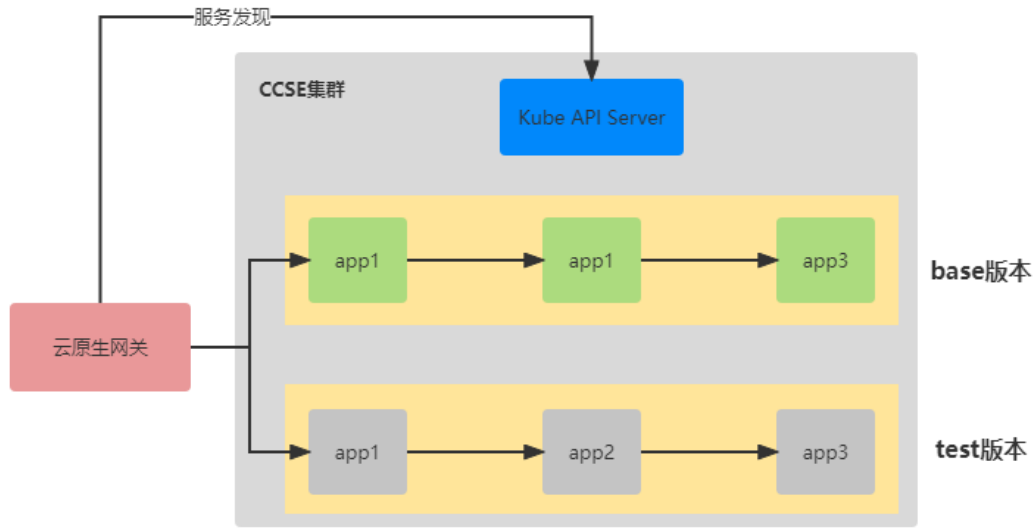
TrafficLabel

1. 如果有一组服务要做全链路灰度，可以按照服务所在的命名空间划分，每个命名空间下创建一个 TrafficLabel 资源，定义所需要的流量标签
2. Ingress 网关要使用 TrafficLabel 可以根据 Ingress 所在的命名空间和标签信息，使用工作负载选择器指定要绑定的网关
3. 对单个服务进行流量打标的场景，使用工作负载选择器选择对应的工作负载
4. 一个工作负载上的流量标签只定义在一个 TrafficLabel 资源上

4.2 使用云原生网关和 CSM 服务网格实现全链路灰度发布

在流量标签和全链路灰度功能说明中，我们可以看到使用 TrafficLabel 实现从 Ingress 网关到业务的全链路灰度能力。除了 Ingress 网关之外，天翼云还提供全托管式的云原生网关产品，具备更好的性能、扩展性和高可用能力，这里介绍如何使用我们的云原生网关作为访问入口的场景下实现全链路灰度。

服务架构



云原生网关作为微服务的访问入口，支持把 CCSE 作为服务发现来源。我们在 CCSE 部署 base 和 test 两个版本的微服务，通过云原生网关的标签路由能力实现对微服务入口应用 app1 的路由调度，通过流量标签实现微服务链路中的路由调度。这里我们还是使用 app-version-tag 头部实现全链路流量控制，下面介绍具体操作步骤（应用部署参考全链路灰度部分，这里不再赘述）

云原生网关服务来源配置

首先在当前 CSM 服务网格同 vpc 下开通一个云原生网关实例，开通完成后，进入云原生网关实例页面，选择服务来源菜单 -> 添加服务来源，选择 CCSE，在下拉列表中选择开通了服务网格并部署了应用的 CCSE 集群，如下图：



添加服务到云原生网关

基于新添加的 CCSE 服务来源，我们添加服务到云原生网关用于路由访问；选择服务管理菜单 -> 添加服务 -> CCSE 服务来源，选择服务部署的命名空间，可以看到我们已经部署的 app1、app2、app3 应用，这里我们只需要添加入口的 app1 应用即可



配置服务标签

在服务列表页可以看到新添加的 app1 服务，选择管理 进入服务管理页面，最下方可以看到版本管理。云原生网关会获取服务的标签信息，并根据标签配置不同的服务子集（类似服务网格中目标规则中的 subset）。我们根据服务的 version 标签配置 base 和 test 两个版本，如下图：

服务版本

版本名称	标签名	标签值	实例数/比例	操作
base	CSM_TRAFFIC_TAG	base	1(50.00%)	编辑 删除
test	CSM_TRAFFIC_TAG	test	1(50.00%)	编辑 删除

云原生网关配置路由规则

在云原生网关控制台选择路由管理 -> 添加路由，配置 app-version-tag 头部匹配 base 时路由到 app1 的 base 版本，匹配 test 时路由到 app1 的 test 版本，如下图：

创建路由

方法 (Method)

Method匹配值,可多选,不填则匹配所有的HTTP方法

优先级

如果不同路由包含相同uri,值越大优先级越高,路由将被优先匹配,默认为0

请求头 (Header) + 添加请求头

Header Key	条件	值	操作
app-version-tag	等于	base	

请求参数 (Query) + 添加请求参数

Cookie + 添加Cookie参数

目标服务

单服务 多服务 标签路由 Mock 重定向

目标服务	版本	权重 (%)	操作
app1_http_8000	base	100	

+ 添加目标服务

创建路由
✕

方法 (Method)

Method匹配值, 可多选, 不填则匹配所有的HTTP方法

优先级

如果不同路由包含相同uri, 值越大优先级越高, 路由将被优先匹配, 默认为0

请求头 (Header) + 添加请求头

Header Key	条件	值	操作
app-version-tag	等于	test	🗑

请求参数 (Query) + 添加请求参数

Cookie + 添加Cookie参数

目标服务

单服务
 多服务
 标签路由
 Mock
 重定向

目标服务	版本	权重 (%)	操作
app1_http_8000	test	100	🗑

+ 添加目标服务

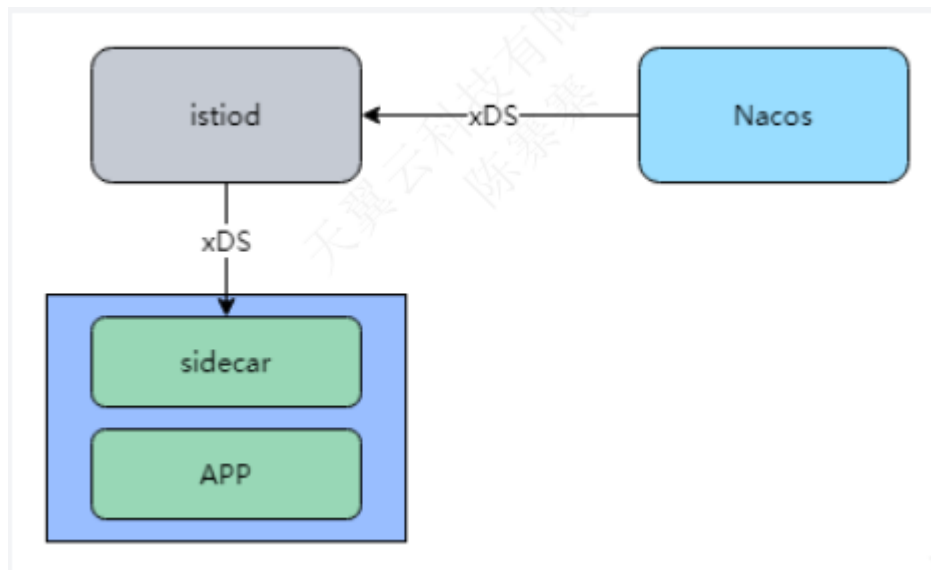
再次访问应用可以看到带上 app-version-tag: test 头部访问总是请求到 app1 的 test 版本, 带上 app-version-tag: base 头部访问总是请求到 app1 的 base 版本; 但是还是会访问到 app2 和 app3 的多版本

配置流量标签

我们需要再部署针对应用的 TrafficLabel 以及虚拟服务和目标规则资源, 此部分参考全链路灰度部分, 这里不需要部署针对 Ingress 网关的流量标签资源。

4.3 CSM 服务网格对接 Nacos 注册中心

天翼云 CSM 服务网格对接 Nacos 架构如下图所示, 控制面通过 xDS 协议从 nacos 发现服务, 通过 xDS 协议下发到数据面中



服务网格对接 Nacos 注册中心前需要您已经订购了微服务引擎注册配置中心实例（Nacos 内核）

开通网格实例时添加 nacos 注册中心

在网格开通页面可以选择添加 nacos 注册服务，选择已经开通的同 vpc 下的 nacos 注册中心实例即可

访问 Nacos 中注册的服务

访问服务网格中从 Nacos 注册中心发现的服务格式如下：

`服务名}.${nacos 分组名}.${nacos 命名空间 id}.nacos`

4.4 使用 SDK 操作 istio 资源

Go 语言版本

准备工作

1. 创建服务网格实例
2. 到 CCSE 控制台找到服务网格控制面部署所在的集群，创建命名空间 demo
3. 获取网格控制面 CCSE 集群的 kubeconfig 信息，保存到本地文件中
4. 准备一个 VirtualService 资源，保存到 vs.yaml 文件中

使用 go 在 default 命名空间创建 VirtualService 资源并列举 default 命名空间下所有的 VirtualService 资源，代码示例：

```
package main
```

```
import (
    "context"
    "flag"
    "gopkg.in/yaml.v2"
    "istio.io/client-go/pkg/apis/networking/v1beta1"
    "istio.io/client-go/pkg/clientset/versioned"
```

```

    v1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/tools/clientcmd"
    "log"
    "os"
)

var (
    ns      = flag.String("namespace", "demo", "namespace to deploy")
    vsFile  = flag.String("vs", "vs.yaml", "virtual service yaml")
    kubeConf = flag.String("kubeconfig", "", "kube config file")
)

func main() {
    flag.Parse()
    rc, err := clientcmd.BuildConfigFromFlags("", *kubeConf)
    if err != nil {
        panic(err)
    }
    clientset, err := versioned.NewForConfig(rc)
    if err != nil {
        panic(err)
    }
    bytes, err := os.ReadFile(*vsFile)
    if err != nil {
        panic(err)
    }
    vs := &v1beta1.VirtualService{}
    err = yaml.Unmarshal(bytes, vs)
    if err != nil {
        panic(err)
    }
    log.Println(vs)
    _, err = clientset.NetworkingV1beta1().VirtualServices(*ns).Create(context.TODO(), vs,
v1.CreateOptions{})
    if err != nil {
        panic(err)
    }

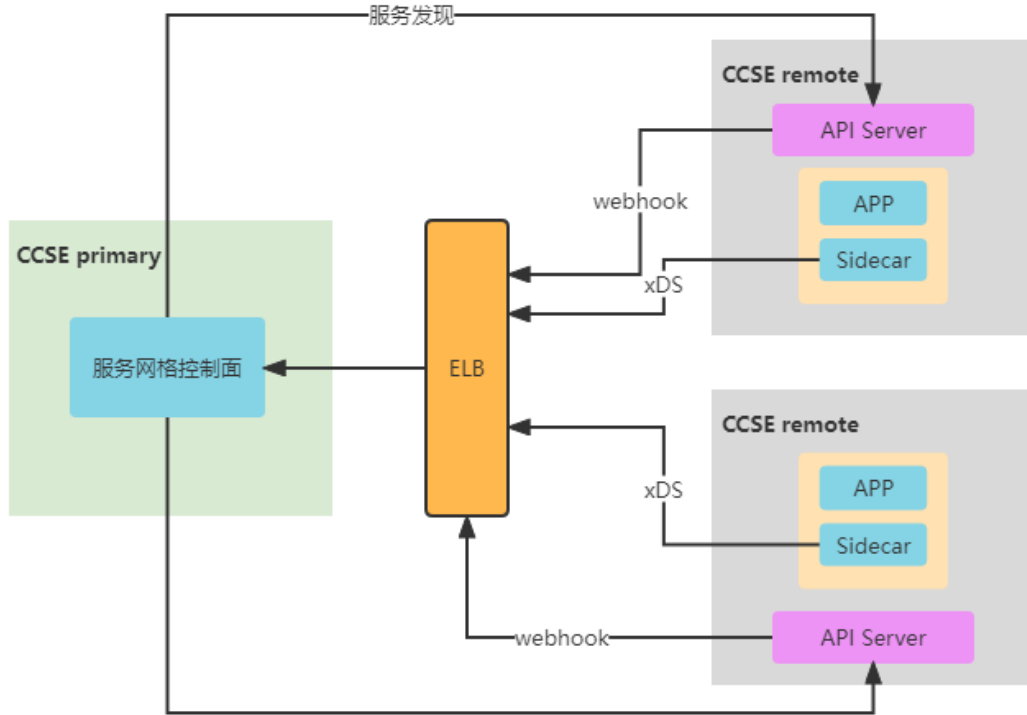
    vsList, err := clientset.NetworkingV1beta1().VirtualServices(*ns).List(context.TODO(),
v1.ListOptions{})
    if err != nil {
        panic(err)
    }
    for _, vsItem := range vsList.Items {
        log.Println(vsItem)
    }
}

```

5. 常见问题

5.1 CSM 服务网格控制面部署在哪里？

当前开通 CSM 服务网格需要您先开通 CCSE 实例，开通服务网格时需要您先选择已经开通的 CCSE 集群，用于部署服务网格控制面。CSM 服务网格架构图如下：



当前 CSM 服务网格支持多集群模式，主集群（Primary）用于部署网格控制面，通过 ELB 向其他从集群（Remote）暴露控制面服务（包括 xDS、webhook 等接口）。

5.2 CSM 服务网格开通失败，提示 xx 资源已存在

由于 CSM 服务网格控制面部署在租户的 CCSE 集群上，开通过程中会主集群安装 CRD，部署相关控制面服务等，如果您在自己的 CCSE 集群上已经安装了相同的资源，可能导致资源冲突，需要您确认是否可以删除当前 CCSE 集群上的冲突资源是否可以删除，确认删除冲突资源后再重新开通服务网格。

CSM 服务网格安装过程中需要新增的 CRD 如下：

```
authorizationpolicies.security.istio.io
customproxyconfigs.networking.istio.io
destinationrules.networking.istio.io
envoyfilters.networking.istio.io
gateways.networking.istio.io
istiooperators.install.istio.io
localratelimiters.networking.istio.io
peerauthentications.security.istio.io
proxyconfigs.networking.istio.io
```

requestauthentications.security.istio.io
 serviceentries.networking.istio.io
 sidecars.networking.istio.io
 telemetries.telemetry.istio.io
 trafficlabels.networking.istio.io
 virtualservices.networking.istio.io
 wasmplugins.extensions.istio.io
 workloadentries.networking.istio.io
 workloadgroups.networking.istio.io

新增资源部署如下：

apiVersion	Kind	Namespace	Name
apps/v1	Deployment	istio-system	istio-eastwestgateway
networking.istio.io/v1alpha3	Gateway	istio-system	istiod-gateway
networking.istio.io/v1alpha3	VirtualService	istio-system	istiod-vs
rbac.authorization.k8s.io/v1	ClusterRole	istio-system	istio-eastwestgateway-sds
rbac.authorization.k8s.io/v1	ClusterRoleBinding	istio-system	istio-eastwestgateway-sds
v1	Service	istio-system	istio-eastwestgateway
v1	ServiceAccount	istio-system	istio-eastwestgateway-service-account
rbac.authorization.k8s.io/v1	ClusterRole	istio-system	istio-operator
rbac.authorization.k8s.io/v1	ClusterRoleBinding	istio-system	istio-operator
install.istio.io/v1alpha1	IstioOperator	istio-system	istiocontrolplane
apps/v1	Deployment	istio-system	istio-operator
v1	Service	istio-system	istio-operator
v1	ServiceAccount	istio-system	istio-operator
v1	Endpoints	opa-istio	admission-controller
v1	Service	opa-istio	admission-

			controller
admissionregistration.k8s.io/v1	MutatingWebhookConfiguration	无	opa-istio-admission-controller

5.3 如何删除 CCSE 集群上处于 Terminating 状态的命名空间？

CCSE 集群中如果已经安装了服务网格可能存在 istio-system 命名空间，如果要开通服务网格需要先清理命名空间，您可能会遇到删除命名空间后，命名空间一直处于 Terminating 状态，且无法删除。

原因：可能是因为命名空间下有其他资源，或者没有资源的情况也可能一直卡死

解决方案：

1. `kubectl get namespace <terminating-namespace> -o json > ns.json`
2. 编辑 ns.json，将 spec.finalizers 设为空数组
3. 执行 `kubectl proxy`，启动一个 kube api server 本地代理
4. 另开一个窗口执行命令更新命名空间的 finalizers

```
curl -k -H "Content-Type: application/json" -X PUT --data-binary @ns.json
http://127.0.0.1:8001/api/v1/namespaces/<terminating-namespace>/finalize
```

a) opa-istio 容器为什么启动失败？

在需要使用 OPA 功能的命名空间打上 `opa-istio-injection="enabled"` 标签后，服务网格内会注入 opa-istio 容器，用于实现鉴权；opa-istio 容器启动当前依赖同命名空间下部署两个 config map，如下：

opa-istio 的配置文件，指定了 grpc 鉴权服务的端口、路径等

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: opa-istio-config
data:
  config.yaml: |
    plugins:
      envoy_ext_authz_grpc:
        addr: :9191
        path: istio/authz/allow
    decision_logs:
      console: true
```

opa-istio 默认的 OPA 策略，默认 istio/authz/allow 路径的鉴权结果是 false（拦截）

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: opa-policy
data:
  policy.rego: |
    package istio.authz

    import input.attributes.request.http as http_request
    import input.parsed_path

    default allow = false
```

在需要使用 OPA 功能的命名空间下创建这两个 config map 再启动 pod 即可。

b) OPA 鉴权规则为什么不生效？

OPA 功能的流程如下图所示：



1. 首先需要确保您已经给业务 pod 注入了 istio sidecar，它是执行 OPA 授权策略的地方
2. CSM 服务网格中采用注入 OPA sidecar 的方式集成 OPA 授权能力，所以需要您确认业务 pod 里注入了 OPA sidecar，具体可以通过给 pod 所在的命名空间打上 opa-istio-injection="enabled" 标签
3. 您需要在注入了 OPA sidecar 之后在同命名空间下定义 OpaPolicy，OpaPolicy 的 WorkloadSelector 需要选中要执行 OPA 授权的工作负载
4. 此时我们的 OPA sidecar 中已经配置了响应的 OPA 策略，我们还需要通过定义 CUSTOM 类型的 AuthorizationPolicy，将客户端请求的流量转发到 OPA sidecar 做授权；因此我们还需要定义外部授权服务（网络安全中心-自定义授权服务）需

要定义 ServiceEntry 提供外部授权服务的域名解析能力。

因此当遇到 OPA 策略不生效的问题时，您可以按照以下顺序进行确认：

1. 是否开启了 OPA 功能（在网格管理-自定义配置菜单，主要包括部署 OPA 控制面、webhook 等服务）
2. 是否注入了 istio sidecar，具体可以检查 namespace 是否有 istio 注入标签，或者 pod 是否有 istio sidecar 注入相关的注解
3. 是否定义了外部授权服务，以及依赖的 ServiceEntry
4. 是否定义了 CUSTOM 类型的 AuthorizationPolicy，将应用流量引导到 OPA sidecar

5.6 为什么加入网格的 pod 停止时会有被调或者主调失败的情况？

问题：

加入到网格的 pod 会注入 sidecar，当 pod 停止时可能会有当前 pod 正在处理中的请求或者当前 pod 调用外部的请求失败。

原因分析：

pod 注入 sidecar 之后，istio-proxy 作为代理会拦截 pod 的入流量（外部请求当前 pod）和出流量（当前 pod 请求外部），在 pod 停止时，istio-proxy 也会在一段时间内退出，如果这段时间内存量的入和出方向的请求没有处理完，则请求可能失败。

解决方案：

1. 修改 sidecar 代理终止等待时长
 - (1) 进入服务网格控制台，选择 sidecar 管理 -> sidecar 代理配置
 - (2) 选择要配置的命名空间，根据业务需求调整 sidecar 代理终止等待时长

2. 修改 sidecar 生命周期管理策略

sidecar 生命周期管理策略可以配置 sidecar 在启动后或者停止前执行的一些操作，比如可以在 pod 停止后基于一些信号决定什么时候停止 sidecar，保障业务请求都处理完成，配置步骤：

- (1) 进入服务网格控制台，选择 sidecar 管理 -> sidecar 代理配置

(2) 选择要配置的命名空间，根据业务需求调整 preStop 和 postStart 配置

5.7 为什么目标规则配置不生效？

问题：

我们在 default 命名空间部署 bookinfo 应用，我们在另外一个命名空间 foo 配置虚拟服务和目标规则，将各个服务的流量全部路由到 v1 版本，如下图所示：

Bookinfo 应用：

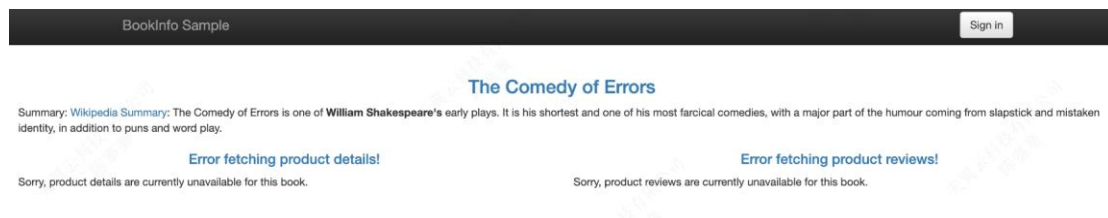
```
samples kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
details-v1-7c7dbcb4b5-pfpwg        2/2     Running   0           11s
productpage-v1-6c5c9c9d9-8cr7j     2/2     Running   0           11s
ratings-v1-844796bf85-lqv4j       2/2     Running   0           11s
reviews-v1-5cf854487-tsrxl        2/2     Running   0           11s
reviews-v2-955b74755-lrqs4        2/2     Running   0           11s
reviews-v3-797fc48bc9-5hvmb       2/2     Running   0           11s
```

在 foo 命名空间定义的虚拟服务和目标规则：

```
samples kubectl get vs -n foo
NAME          GATEWAYS          HOSTS          AGE
details       ["details.default.svc.cluster.local"] 2m55s
productpage   ["productpage.default.svc.cluster.local"] 2m55s
ratings       ["ratings.default.svc.cluster.local"] 2m55s
reviews       ["reviews.default.svc.cluster.local"] 2m55s

samples kubectl get dr -n foo
NAME          HOST          AGE
details       details.default.svc.cluster.local 25s
productpage   productpage.default.svc.cluster.local 26s
ratings       ratings.default.svc.cluster.local 26s
reviews       reviews.default.svc.cluster.local 26s
```

此时通过 Ingress gateway 访问 bookinfo 应用显示对 details 和 reviews 服务的调用失败了



原因：

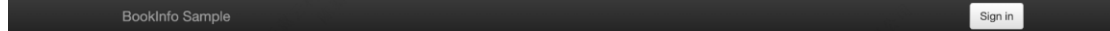
查看 productpage 日志可以看到，找不到上游的 details 和 reviews 服务 (NC)

```
[2023-08-15T11:44:18.974Z] "GET /details/0 HTTP/1.1" 503 NC cluster_not_found "-" 0 0 0 "-" Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.0.0 Safari/537.36 "50210a99-786d-926e-9184-7af432ec4d52" "details:9080" "-" - - 10.100.47.59:9080 10.1.0.56:56438 - -
[2023-08-15T11:44:18.978Z] "GET /reviews/0 HTTP/1.1" 503 NC cluster_not_found "-" 0 0 0 "-" Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.0.0 Safari/537.36 "50210a99-786d-926e-9184-7af432ec4d52" "reviews:9080" "-" - - 10.97.227.1:9080 10.1.0.56:51844 - -
```

原因在于目标规则的定义没有生效，我们在虚拟服务里定义的规则需要路由到 v1 版本，但是没有找到目标规则对 v1 版本的定义；这里建议您把目标规则定义到目标服务所在的命名空间或者根命名空间（默认 istio-system）。

在 default 命名空间重新定义了目标规则之后，服务可以正常访问了：

```
samples kubectl get dr
NAME          HOST                                     AGE
details      details.default.svc.cluster.local     6s
productpage  productpage.default.svc.cluster.local 6s
ratings      ratings.default.svc.cluster.local     6s
reviews      reviews.default.svc.cluster.local     6s
```



5.8 多集群的命名空间是什么关系？

当前 CSM 服务网格支持多集群架构，控制面采用非托管模式，部署在租户的 CCSE 集群上。多集群中部署网格控制面的集群称为主集群，添加的其他 CCSE 集群称为从集群，允许服务网格中多集群的命名空间不一致的情况。主集群命名空间关注数据面管理以及控制面配置，从集群命名空间只关注数据面管理。

sidecar 自动注入

当前 CSM 服务网格支持基于命名空间标签的 sidecar 注入，给命名空间打上 istio-injection: enabled 标签后可以打开命名空间 sidecar 自动注入功能；当前主从集群的命名空间自动注入功能是独立配置的，在某个集群上给命名空间打上自动注入标签只影响当前集群上该命名空间 pod 的自动注入，不影响其他集群上同命名空间的自动注入。

服务网格 CRD 配置

CSM 服务网格通过在主集群定义 CRD，进一步由控制面服务消费和并下发到所有集群。因此对于一些有命名空间语义的 CRD（如 VirtualService，DestinationRule 等），也就只能定义在主集群的命名空间中，并且对从集群中存在的相同命名空间生效。对于从集群中存在而主集群中不存在的命名空间，无法定义和生效相关的 CRD。

5.9 服务网格对于 Service 的定义是否有要求？

CSM 服务网格支持包括 HTTP、HTTPS、gRPC、TCP 等流量的代理，默认会自动检测协议类型。对于无法识别的协议，将被当做 TCP 或者 UDP 流量处理。本文介绍网格中的服务端口定义规范。

端口协议定义

方式 1：通过端口名定义

服务的端口必须有名字，命名满足[协议]-[后缀]格式，协议字段可以是 http、http2、grpc、mongo、redis 等，主要用于服务网格的路由特性，例如 name: http2-foo 或者不加后缀 name: http 都是合法的格式，但是 name: http2foo 是非法的。如果端口名称不按照上述格式定义，该端口上的流量将被当做 TCP 流量或者 UDP（显示指定 UDP 端口的情况）。

方式 2：通过服务端口的 appProtocol 字段定义

当前 CCSE 集群 Service 端口支持使用 appProtocol 字段显式指定协议

服务共享工作负载的情况

多个 Service 选中相同工作负载的情况下，Service 中同一个端口的协议必须一致。

5.10 sidecar 内存持续升高有什么解决办法？

问题现象

pod 中 istio sidecar 容器监控中发现内存消耗持续升高

原因分析

istio 采用 envoy 作为数据面代理，动态内存消耗主要有以下方面：

原因	说明
网格配置	默认情况下网格中的配置和服务发现信息会全量同步到所有 sidecar，当网格中的配置增多或者 pod 数量增多时，将导致 sidecar 代理的内存升高
动态请求消耗的内存	Envoy 中会为请求申请 buffer 用于缓存请求及应答信息，当请求量和消息比较大的时候会增加内存消耗
可观测指标发散	当指标的 tag 较多时将会在内存中产生较多副本，增加内存消耗
HTTP2 流量控制相关	Sidecar 实现 HTTP2 编解码中有流级别及连接级别的缓存字节数限制，默认为 256MB，当 sidecar 处理能力不足将导致数据在内存中累积

解决：

对于网格配置导致的 sidecar 内存升高，可以通过限定资源的可见范围较少配置扩散，如 VirtualService、DestinationRule 的 exportTo 字段指定配置分发的命名空间；通过配置 Sidecar 资源，限定当前服务可见的服务发现范围等。

对于动态请求导致的内存消耗，考虑减少 sidecar 代理的流量。默认情况下 sidecar 会拦截所有 inbound 和 outbound 的流量，可以根据业务需要对不需要代理的端口取消流量拦截；

具体可以通过 sidecar 管理-sidecar 代理配置菜单下配置全局及命名空间级别的流量拦截策略。

对于指标发散和 HTTP2 缓冲配置导致的内存升高，我们将在后续的版本中提供优化方案。

5.11 为什么开通服务网格之后，网格控制面组件启动失败？

问题现象

开通服务网格后控制面服务没有正常启动，报错信息如下：

```
message: 'Internal error occurred: failed calling webhook "cosign-webhook.kube-system.svc": failed to call webhook: Post "https://cosign-webhook.kube-system.svc:443/handlepod?timeout=30s": x509: certificate is valid for cosign-webhook, cosign-webhook.default, cosign-webhook.default.svc, not cosign-webhook.kube-system.svc'
```

问题原因

由于在 CCSE 集群开启了 cube-sign 插件，该插件会对集群中部署的镜像做签名验证，提升系统安全性。服务网格控制面组件镜像不会使用 cube-sign 签名，因此部署会因为签名校验而失败。

解决

关闭 CCSE 集群 cube-sign 插件，重新部署网格控制面服务。

5.12 为什么注入了 sidecar，但是流量拦截没生效？

问题现象

服务网格中部署的 pod 注入了 sidecar，但是流量拦截不生效

问题分析

查看 pod 注入的 sidecar 配置如下：

```
initContainers:
- args:
  - istio-iptables
  - -p
  - "15001"
  - -z
  - "15006"
  - -u
  - "1337"
  - -m
  - REDIRECT
  - -i
  - ""
  - -x
  - ""
  - -b
  - ""
  - -d
  - 15090,15021,15020
```

看到初始化容器进行 iptables 配置是，-i 和-b 参数都是空字符串参数含义如下：

参数	说明
i	逗号分隔的 ip 集合（支持 CIDR），表示 outbound 流量会被重定向到 sidecar 的目标 ip 集合，*表示所有 outbound 流量都会被重定向到 sidecar，空字符串表示所有流量都不会被重定向到 sidecar
b	逗号分隔的端口列表，表示会被重定向到 sidecar 的 inbound 端口，*表示所有端口都会被重定向，空字符串表示所有流量都不会被重定向

按照以上配置，i 和 b 参数都是空字符串，所有的 inbound 和 outbound 流量都不会经过 sidecar，也无法获得网格的治理能力。

问题解决

在 sidecar 管理-sidecar 代理配置按照全局或者命名空间粒度配置 sidecar 的拦截行为，确认符合业务需求。