

# 什么是服务网格

服务网格是基于 Istio 开源技术构建的面向云原生应用的下一代服务网格。

服务网格是一种具备高性能、高易用性的全托管服务网格产品，通过提供完整的非侵入式的微服务治理方案，能够统一治理多云多集群的复杂环境，以基础设施的方式为用户提供服务流量治理、安全性治理、服务流量监控以及传统微服务（SpringCloud、Dubbo）功能。

DCE 5.0 的服务网格兼容社区原生 Istio 开源服务网格，提供原生 Istio 接入管理能力。在较高的层次上，服务网格有助于降低服务治理的复杂性，减轻开发运维团队的压力。

服务网格作为 DCE 5.0 产品的体系一员，无缝对接[容器管理](#)平台，可以为用户提供开箱即用的上手体验，并作为基础设施为[微服务引擎](#)提供容器微服务治理支持，方便用户通过单一平台对各类微服务系统做统一管理。

## 产品优势

DCE 5.0 服务网格相比其他产品具备以下优势：

- 简单易用

无需修改任何业务代码，也无需手动安装代理，只需开启服务网格功能，即可体验丰富的无侵入服务治理能力。

- 策略化的智能路由与弹性流量管理

支持为服务配置负载均衡、服务路由、故障注入、离群检测等治理规则。结合一站式治理系统；提供实时的、可视化的微服务流量管理；支持无侵入智能流量治理，应用无需任何改造，即可进行动态的智能路由和弹性流量管理。

- 权重、内容、TCP/IP 等路由规则。

- HTTP 会话保持，满足业务处理持续性诉求。
    - 限流、离群检测，实现服务间链路稳定、可靠。
    - 网络长连接管理降低资源损耗，提升网络吞吐量。
    - 服务安全认证：认证、鉴权、审计等，提供服务安全保障基石。
  - 图形化应用全景拓扑，流量治理可视化
- 服务网格提供了可视化的流量监控，包括链路信息、服务异常响应、超长响应时延等信息，通过图表以及拓扑等多样化形式全面展现业务运行情况。
- 服务网格可以结合应用运维管理、应用性能管理服务，提供详细的微服务级流量监控、异常响应流量报告以及调用链信息，能够更快速、更准确的定位问题。
- 性能增强，可靠性增强
- 服务网格控制面和数据面基于社区版本的基础上进行更可靠的加固和性能优化。
- 多云多集群、多基础设施
- 提供免运维的托管控制面，提供多云多集群的全局统一的服务治理、安全和服务运行监控能力，还提供对容器和虚拟机（VM）等多种基础设施的统一服务发现和管理。
- 协议扩展
- 支持 Dubbo 协议。
- 传统 SDK 集成
- 提供 Spring Cloud、Dubbo 等传统微服务 SDK 的集成解决方案，传统微服务 SDK 开发的业务无需大量代码改造即可快速迁移到云原生容网格运行环境上运行。

## 学习路径

服务网格的学习路径如下：

flowchart TD

```

install([安装部署])
install --> mesh[创建网格]
    subgraph mesh[创建网格]
        managed[托管网格]
        private[专有网格]
        external[外接网格]
    end

end

mesh --> cluster[纳管集群]

cluster --> inject[注入边车]

    subgraph inject[注入边车]
        namespace[命名空间注入]
        workload[工作负载注入]
    end
end

inject -.-> service[服务管理]
inject -.-> traffic[流量治理]
inject -.-> security[安全治理]
inject -.-> sidecar[边车管理]
inject -.-> watch[流量监控]
inject -.-> gateway[网格网关]
inject -.-> config[网格配置]
inject -.-> upgrade[版本升级]

service -.-> entry[服务条目<br>一键修复]
traffic -.-> virtual[虚拟服务<br>目标规则<br>网关规则]
security -.-> peer[对等身份认证<br>请求身份认证<br>授权策略]
sidecar -.-> sidecar[命名空间边车<br>工作负载边车<br>边车流量透传]
watch -.-> watch2[流量监控<br>流量拓扑]
config -.-> intercon[多云互联<br>Istio 资源<br>TLS 密钥]
upgrade -.-> upgrade1[Istio 升级<br>边车升级]

classDef plain fill:#ddd,stroke:#fff,stroke-width:1px,color:#000;
classDef k8s fill:#326ce5,stroke:#fff,stroke-width:1px,color:#fff;
classDef cluster fill:#fff,stroke:#bbb,stroke-width:1px,color:#326ce5;

class mesh plain

```

class install,service,gateway,traffic,watch,upgrade,security,entry,virtual,peer,cluster,sidecar,sidecar,watch2,managed,private,external,namespace,workload,upgrade1,config,intercon cluster

```
click install "https://docs.daocloud.io/mspider/install/install/"
click managed "https://docs.daocloud.io/mspider/user-guide/service-mesh/"
click private "https://docs.daocloud.io/mspider/user-guide/service-mesh/"
click external "https://docs.daocloud.io/mspider/user-guide/service-mesh/external-mesh/"
click cluster "https://docs.daocloud.io/mspider/user-guide/cluster-management/join-clus/"
click global "https://docs.daocloud.io/mspider/user-guide/sidecar-management/global-sidecar/"
click namespace "https://docs.daocloud.io/mspider/user-guide/sidecar-management/ns-sidecar/"
click workload "https://docs.daocloud.io/mspider/user-guide/sidecar-management/workload-sidecar/"

click gateway "https://docs.daocloud.io/mspider/user-guide/gateway-instance/create/"
click service "https://docs.daocloud.io/mspider/user-guide/service-list/"
click traffic "https://docs.daocloud.io/mspider/user-guide/traffic-governance/"
click security "https://docs.daocloud.io/mspider/user-guide/security/"
click watch "https://docs.daocloud.io/mspider/user-guide/traffic-monitor/"
click upgrade "https://docs.daocloud.io/mspider/user-guide/upgrade/istio-update/"
click entry "https://docs.daocloud.io/mspider/user-guide/service-list/service-entry/"
click virtual "https://docs.daocloud.io/mspider/user-guide/traffic-governance/virtual-service/"
click peer "https://docs.daocloud.io/mspider/user-guide/security/peer/"
click sidecar "https://docs.daocloud.io/mspider/user-guide/sidecar-management/ns-sidecar/"
click sidecar "https://docs.daocloud.io/mspider/user-guide/sidecar-management/passthrough/"
click watch2 "https://docs.daocloud.io/mspider/user-guide/traffic-monitor/conn-topo/"
click intercon "https://docs.daocloud.io/mspider/user-guide/multicluster/cluster-interconnect/"
```

## 常见问题和故障案例

- [服务网格常见问题](#)
- [创建网格时找不到所属集群](#)
- [创建网格时一直处于“创建中”，最终创建失败](#)
- [创建的网格异常，但无法删除网格](#)
- [托管网格纳管集群失败](#)
- [托管网格纳管集群时 istio-ingressgateway 异常](#)
- [网格空间无法正常解绑](#)
- [DCE 4.0 接入问题追踪](#)

- [命名空间边车配置与工作负载边车冲突](#)
- [托管网格多云互联异常](#)
- [边车占用大量内存](#)
- [创建网格时，集群列表存在未知集群](#)
- [托管网格 APIServer 证书过期解决办法](#)
- [服务网格中常见的 503 报错](#)
- [如何使集群中监听 localhost 的应用被其它 Pod 访问](#)

[下载 DCE 5.0](#) [安装 DCE 5.0](#) [申请社区免费体验](#)

## 功能总览

此处介绍服务网格支持的功能。

### 服务管理

- 服务注册与发现

支持服务注册与发现，支持服务实例的动态注册和注销。

- VM 服务注册

支持 VM 服务注册，支持一行命令完成 VM 服务的注册与纳管。

- 服务智能诊断

支持根据网格服务使用最佳实践，自动检查接入网格服务的配置情况，提供一键修复和手工修复等多种策略。

## 流量治理

- 七层连接池管理

支持配置 HTTP 最大请求数、最大重试次数、最大等待请求数、每次连接最大请求数以及连接最长空闲时间。

- 四层连接池管理

支持配置 TCP 最大连接数、连接超时时长、TCP 存活检测（包括空闲超时时长、最大探测数、探测间隔时长）。

- 离群检测

支持配置服务离群检测规则，包括实例被驱逐前的连续错误数、检查周期、基础隔离时间以及最大隔离实例比例。

- 重试

支持配置 HTTP 重试次数、重试超时时间以及重试条件。

- 超时

支持配置 HTTP 请求超时时间。

- 负载均衡

支持配置随机调度、轮询调度、最少连接和一致性哈希多种负载均衡算法。

- HTTP Header

可以灵活添加、修改和删除指定 HTTP Header，包括将 HTTP 请求转发到目标服务之前对 Header 的操作，以及将 HTTP 响应回复给客户端前对 Header 的操作。

- 故障注入

支持配置延时故障和中断故障。

## 安全治理

- 透明双向认证

支持界面配置服务间的双向认证，包括对等身份认证、请求身份认证。

- 细粒度访问授权

支持通过界面配置服务间的访问授权（后台 API 可以配置 Namespace 级别授权，授权将会给一个特定的接口）。

## 边车管理

- 边车注入

支持通过界面配置服务的边车注入策略，支持多维度的边车默认注入策略。

- 边车热升级

支持边车热升级，在控制面升级后，自动检测边车版本并给出升级建议，支持无缝热升级，业务不中断。

- 边车服务发现范围

支持通过自定义配置边车服务发现范围，根据不同业务场景，极大减少的边车资源消耗的压力。

## 可观测性

- 流量拓扑

支持查看网格应用流量拓扑，了解服务间依赖关系。

- 服务运行监控

支持查看服务访问信息，包括服务和服务各个版本的 QPS 和延时等指标。

- 访问日志

支持收集和检索服务的访问日志。

- 调用链

支持非侵入调用链埋点，并可以通过检索调用链数据进行问题定界定位。

## 多集群模式

- 多集群配置统一管理

支持多集群网格的网格全局配置、工作集群配置管理等；支持对不同集群进行不同粒度的边车注入策略，同时支持对数据面统一管理跨集群流量策略等。

- 可扩展性

支持一键接入、移除集群，支持接入集群时进行集群的接入检查，预防接入集群时的错误。

## 网格数据面微服务框架

- Spring Cloud

支持 Spring Cloud SDK 开发的服务无侵入式的接入网格，并统一管理。

- Dubbo 协议

支持 Dubbo SDK 开发的服务无侵入式的接入网格，并统一管理。

## 兼容性和扩展

- 版本兼容

API 完全兼容通用服务网格。

- 插件支持



支持 Tracing、Prometheus、Kiali、Grafana。

## 适用场景

服务网格适用的场景如下：

### 服务流量治理

流量治理是一个非常宽泛的话题，例如：

- 动态修改服务间访问的负载均衡策略，如根据某个请求特征做会话保持。
- 同一个服务有两个版本在线，将一部分流量切到某个版本上。
- 对服务进行保护，例如限制并发连接数、限制请求数、隔离故障服务实例等。
- 动态修改服务中的内容，或者模拟服务运行故障等。

服务网格可以提供非侵入的流量治理能力，无需修改任何业务代码就能实现这些服务治理功能。根据服务的协议，提供策略化、场景化的网络连接管理。可以根据需要对特定服务的特定端口配置不同的治理规则。

### 场景优势

- 重试

服务访问失败自动重试，从而提高总体访问成功率和质量。支持配置 HTTP 请求重试次数、重试超时时间和重试条件。

- 超时

服务访问超时自动处理，快速失败，从而避免资源锁定和请求卡顿。支持配置 HTTP 请求超时时间。

- 连接池（配置路径为：**流量治理** -> **目标规则**）。可以防止一个服务的失败级联影响到整个应用。

可以对四层协议配置 TCP：

- 最大连接数
- 连接超时时长
- TCP 存活检测
  - 空闲超时时长
  - 最大探测数
  - 探测间隔时长

还可以对七层协议配置 HTTP：

- 最大挂起请求数(HTTP/1.1)
- 最大重试数
- 后端最大请求数(HTTP/2)
- 空闲超时时长
- 每连接最大请求数
- 离群检测

通过离群检测配置实例被驱逐前的连续错误次数、驱逐间隔时长、最小驱逐时间、最大驱逐比例等参数，从而定期考察被访问的服务实例的工作情况。如果连续出现访问异常，则将服务实例标记为异常并进行隔离，在一段时间内不为其分配流量。过一段时间后，被隔离的服务实例会再次被解除隔离，尝试处理请求。如果还不正常，则被隔离更长的时间。从而实现异常服务实例的故障隔离和自动故障恢复。

- 负载均衡

配置各种负载均衡策略，如随机、轮询、最少连接，还可以配置一致性哈希将流量转发到特定的服务实例上。

- HTTP Header

灵活增加、修改和删除指定 HTTP Header，包括将 HTTP 请求转发到目标服务之前对 Header 进行操作。还可以在将 HTTP 响应回复给客户端前，对 Header 进行操作，以非侵入方式管理请求内容。

- 故障注入

通过对选定的服务注入中断故障、延时故障来构造故障场景，无需修改代码即可进行故障测试。

## 端到端的透明安全

众所周知，将传统的单体应用拆分为一个个微服务固然带来了各种好处，例如：更好的灵活性、可扩展性、重用性，但微服务也同样面临着特殊的安全需求：

- 为了抵御中间人攻击，需要用到流量加密。
- 为了提供灵活的服务访问控制，需要用到 TLS 和细粒度访问策略。
- 为了决定哪些人在哪些时间可以做哪些事，需要用到审计工具。

面对这些需求服务网格提供全面的安全解决方案，包括身份验证策略、透明的 TLS 加密以及授权和审计工具。

## 场景优势

- 非侵入安全

服务网格是以一种安全基础设施的方式向用户提供透明的安全能力，让不涉及安全问题的代码安全运行，让不太懂安全的人可以开发和运维安全的服务，不用修改业务代

码就能提供服务访问安全。应用服务网格提供了一个透明的分布式安全层，并提供了底层安全的通信通道，管理服务通信的认证、授权和加密，提供 Pod 到 Pod、服务到服务的通信安全。开发人员在这个安全基础设施层上只需专注于应用程序级别的安全性。

- 多集群安全

在多集群场景下服务网格提供了全局的服务访问安全。多个集群的网格共享一套根证书，给数据面的服务实例分发密钥和证书对，并定期替换密钥证书，根据需要撤销密钥证书。在服务间访问时，网格的数据面代理就会代理本地服务和对端进行双向认证、通道加密。这里的双向认证的服务双方可以来自两个不同的集群，从而做到跨集群的透明的端到端双向认证。

- 细粒度授权

在认证的基础上，就可以进行服务间的访问授权管理，可以控制某个服务，或者服务的一个特定接口进行授权管理。如只开放给特定的一个 Namespace 下的服务，或者开放给某个特定的服务。源服务和目标服务可以在不同的集群，甚至源服务的不同实例在不同的集群，目标服务的不同实例在不同的集群。

## 服务运行监控

运营容器化的基础设施带来了一系列新的挑战。我们需要增强容器、评估 API 端点的性能以及识别出基础设施中的有害部分。服务网格可在不修改代码的情况下实现 API 增强，并且不会带来服务延迟。

服务网格可以为网格内的所有服务通信进行遥测，这种遥测技术提供了服务行为的可观察性，有助于运营商对其应用程序进行故障排除、维护和优化，而不会给服务开发人员带来任何

额外负担。通过服务网格，运营商可以全面了解被监控的服务如何与其他服务以及组件本身进行交互。

## 场景优势

- 非侵入监控数据采集

在复杂应用的场景下，服务间的访问拓扑、调用链、监控等都是服务访问异常时进行定位定界的必要手段。服务网格技术的一项重要能力就是以应用非侵入的方式提供这些监控数据的采集，用户只需关注自己的业务开发，无需额外关注监控数据的生成。

- 丰富性能监控能力

基于网格生成服务访问数据，集成各种不同的性能监控服务，提供跨集群智能的服务运行管理。包括跨集群的服务调用链、服务访问拓扑和服务运行健康状态、通过跨集群的全局视图来关联服务间的访问状况等。

## 服务网格权限说明

[服务网格](#)支持以下几种用户角色：

- Admin
- Workspace Admin
- Workspace Editor
- Workspace Viewer

!!! info

自 DCE 5.0 的 [v0.6.0](../download/index.md) 起，全局管理模块支持为服务网格配置自定义角色，

即除了使用系统默认角色外，还可以在服务网格中自定义角色并授予不同权限。

# 一键诊断和修复

DCE 5.0 服务网格针对纳管的服务，内置了一键诊断和修复功能，可以通过图形化界面进行操作。

1. 进入某个服务网格后，点击 **服务管理** -> **服务列表**。在 **诊断配置** 列中，状态 **异常** 的

服务旁会出现 **诊断** 字样，点击 **诊断**。

诊断

诊断

2. 从右侧滑入一个诊断的弹窗，按照内置的 checklist 进行检查。已通过表示正常，未通过表示需要修复。

勾选未通过的条目，点击 **一键修复** 按钮。可以点击 **重新诊断** 刷新 checklist，通常会在几分钟内完成修复。

修复

修复

3. 成功修复之后，checklist 各项将变灰且全部显示为已通过，点击 **下一步**。

下一步

下一步

4. 列出了需要 **手工修复** 的检查项，您可以点击 **查看修复指导** 阅读对应的文档页面，手动修复检查项。

手工修复

手工修复

!!! note

对于以下系统命名空间中的服务，不建议使用一键修复：

分类	命名空间	作用
Istio 系统命名空间	istio-system	承载 Istio 控制平面组件及其资源
	istio-operator	部署和管理 Istio Operator
K8s 系统命名空间	kube-system	控制平面组件
	kube-public	集群配置和证书等
	kube-node-lease	监测和维护节点的活动
DCE 5.0 系统命名空间	amamba-system	应用工作台
	ghippo-system	全局管理
	insight-system	可观测性
	ipavo-system	首页仪表盘
	kairship-system	多云编排
	kant-system	云边协同
	kangaroo-system	镜像仓库
	kcoral-system	应用备份
	kubean-system	集群生命周期管理
	kpanda-system	容器管理
	local-path-storage	本地存储
	mcamel-system	中间件
	mspider-system	服务网格
	skoala-system	微服务引擎
	spidernet-system	网络模块

# 流量治理

流量治理为用户提供了三种资源配置，虚拟服务、目标规则、网关规则。通过配置相应规则可以实现路由、重定向、离群检测、分流等多项流量治理功能。用户可以通过向导或 YAML 形式创建、编辑治理策略。

- 虚拟服务主要用于对请求流量的路由定制规则，并可以对数据流做出分流、重定向、超时返回等处理
- 目标规则则更关注流量本身的治理，为请求流量提供更强大的负载均衡、连接存活探寻、离群检测等功能
- 网关规则为 Istio 网关提供服务在网关的暴露方式

在实际运用中，需要三类策略配合使用：

- 由虚拟服务来定义路由规则和描述满足条件的请求去哪里
- 目标规则则是定义子集和策略，描述到达目标的请求该怎么处理
- 如果有外部服务通信需求，则需要配置网关规则中的端口映射等细节来实现需求

在服务网格中，为用户提供了向导和 YAML 两种创建/编辑形式，用户可以根据个人习惯自由选择。

- 向导创建方式为用户提供简单直观的互动方式，一定程度上降低了用户的学习成本
- YAML 创建形式更适合资深用户，用户可以直接编写 YAML 文件创建治理策略，并且创建窗口中也为用户提供了较为常用的治理策略模板，提高用户编写速度

## 视频教程

- [如何将微服务接入 DCE 5.0 并治理南北向流量？](#)



- [如何借助服务网格治理传统微服务东西向流量？](#)

## 虚拟服务

在虚拟服务（VirtualService）中，可以通过多种匹配方式（端口、host、header 等）实现对不同的地域、用户请求做路由转发，分发至特定的服务版本中，并按权重比划分负载。

虚拟服务提供了 HTTP、TCP、TLS 三种协议的路由支持。

## 虚拟服务列表

虚拟服务列表展示了网格下的虚拟服务 CRD 信息，用户可以按命名空间查看，也可以基于作用范围、规则标签做 CRD 筛选，规则标签如下：

- HTTP 路由
- TCP 路由
- TLS 路由
- 重写
- 重定向
- 重试
- 超时
- 故障注入
- 代理服务
- 流量镜像

这些标签的字段配置，请参阅 [Istio 虚拟服务参数配置](#)。

虚拟服务提供了两种创建方式：图形向导创建和 YAML 创建。

## 图形向导创建步骤

通过图形向导创建的具体操作步骤如下（参阅[虚拟服务参数配置](#)）：

1. 在左侧导航栏点击 **流量治理** -> **虚拟服务**，点击右上角的 **创建** 按钮。

创建

创建

2. 在 **创建虚拟服务** 界面中，先确认并选择需要将虚拟服务创建到的命名空间、所属服务和应用范围后，点击 **下一步**。

创建虚拟服务

创建虚拟服务

3. 按屏幕提示分别配置 HTTP 路由、TLS 路由和 TCP 路由后，点击 **确定**。

路由配置

路由配置

4. 返回虚拟服务列表，屏幕提示创建成功。在虚拟服务列表右侧，点击操作一列的 **⋮**，  
可通过弹出菜单进行更多操作。

更多操作

更多操作

## YAML 创建

通过 YAML 创建的操作相对简单，用户可以点击 **YAML 创建** 按钮，进入创建页面直接编写 YAML，也可以使用页面内提供的模板简化编辑操作，编辑窗口会提供基本语法检查功能，帮助用户完成编写工作。以下是一个 YAML 示例：

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

```

metadata:
  annotations:
    kube.daocloud.io/cluster: dywtest3
    kube.daocloud.io/indexes: '{"activePolicies":["HTTP_ROUTE,RETRIES","cluster":"dywtest3","
createdAt":"2023-08-07T09:27:48Z","gateway":"nginx-gw/nginx-gwrule","gateways":["nginx-gw/ng
inx-gwrule"],"hosts":["www.nginx.app.com"],"is_deleted":"false","labels":"","name":"nginx-vs","
namespace":"nginx-gw"}'
  creationTimestamp: "2023-08-07T09:27:48Z"
  generation: 10
  managedFields:
    - apiVersion: networking.istio.io/v1beta1
      fieldsType: FieldsV1
      fieldsV1:
        f:metadata:
          f:annotations:
            .: {}
            f:kube.daocloud.io/cluster: {}
            f:kube.daocloud.io/indexes: {}
        f:spec:
          .: {}
          f:gateways: {}
          f:hosts: {}
          f:http: {}
      manager: cacheproxy
      operation: Update
      time: "2023-08-09T03:06:31Z"
  name: nginx-vs
  namespace: nginx-gw
  resourceVersion: "477662"
  uid: 446e8dcf-3c26-47ec-8754-997c21e4df17
spec:
  gateways:
    - nginx-gw/nginx-gwrule
  hosts:
    - www.nginx.app.com
  http:
    - match:
        - uri:
            prefix: /
        name: nginx-http
      retries:
        attempts: 2
        perTryTimeout: 5s
        retryOn: 5xx

```

```

route:
  - destination:
      host: nginx.nginx-test.svc.cluster.local
      port:
        number: 80
status: {}

```

## 概念介绍

- hosts

流量的目标主机。可以来自服务注册信息，服务条目（service entry），或用户自定义的服务域名。可以是带有通配符前缀的 DNS 名称，也可以是 IP 地址。根据所在平台情况，还可能使用短名称来代替 FQDN。这种场景下，短名称到 FQDN 的具体转换过程是要靠下层平台完成的。

一个主机名只能在一个 VirtualService 中定义。同一个 VirtualService 中可以用于控制多个 HTTP 和 TCP 端口的流量属性。

需要注意的是，当使用服务的短名称时（例如使用 `reviews`，而不是 `reviews.default.svc.cluster.local`），服务网格会根据规则所在的命名空间来处理这一名称，而非服务所在的命名空间。假设 `default` 命名空间的一条规则中包含了一个 `reviews` 的 host 引用，就会被视为 `reviews.default.svc.cluster.local`，而不会考虑 `reviews` 服务所在的命名空间。

为了避免可能的错误配置，建议使用 FQDN 来进行服务引用。hosts 字段对 HTTP 和 TCP 服务都是有效的。网格中的服务也就是在服务注册表中注册的服务，必须使用他们的注册名进行引用；只有 Gateway 定义的服务才可以使用 IP 地址。

示例：

```

spec:
  hosts:
    - ratings.prod.svc.cluster.local

```

- Gateways

通过将 VirtualService 绑定到同一 Host 的网关规则，可向网格外部暴露这些 Host。

网格使用默认保留字 mesh 指代网格中的所有 Sidecar。当这一字段被省略时，就会使用默认值 (mesh)，也就是针对网格中的所有 Sidecar 生效。如果为 gateways 字段设置了网关规则（可以有多个），就只会应用到声明的网关规则中。如果想同时对网关规则和服务生效，需要显式的将 mesh 加入 gateways 列表。

示例：

gateways:

- bookinfo-gateway
- mesh

- HTTP

有序规则列表。该字段包含了针对 HTTP 协议的所有路由配置功能，对名称前缀为 http- 、 http2- 、 grpc- 的服务端口，或者协议为 HTTP、HTTP2、GRPC 以及终结的 TLS，另外还有使用 HTTP、HTTP2 以及 GRPC 协议的 ServiceEntry 都是有效的。流量会使用匹配到的第一条规则。

HTTP 主要字段说明：

- Match  
匹配要激活的规则要满足的条件。单个匹配块内的所有条件都具有 AND 语义，而匹配块列表具有 OR 语义。如果任何一个匹配块成功，则匹配该规则。
- Route  
HTTP 规则可以重定向或转发（默认）流量。
- Redirect  
HTTP 规则可以重定向或转发（默认）流量。如果在规则中指定了流量通过选项，则将忽略路由/重定向。重定向原语可用于将 HTTP 301 重定向发送到其他 URI 或 Authority。
- Rewrite  
重写 HTTP URI 和 Authority header，重写不能与重定向原语一起使用。

- Fault

故障注入策略，适用于客户端的 HTTP 通信。如果在客户端启用了故障注入策略，则不会启用超时或重试。

- Mirror/MirrorPercent

将 HTTP 流量镜像到另一个目标，并可以设置镜像比例。

- TCP

一个针对透传 TCP 流量的有序路由列表。TCP 路由对所有 HTTP 和 TLS 之外的端口生效。进入流量会使用匹配到的第一条规则。

- TLS

一个有序列表，对应的是透传 TLS 和 HTTPS 流量。路由过程通常利用 ClientHello 消息中的 SNI 来完成。TLS 路由通常应用在 https-、tls- 前缀的平台服务端口，或者经 Gateway 透传的 HTTPS、TLS 协议端口，以及使用 HTTPS 或者 TLS 协议的 ServiceEntry 端口上。注意：没有关联 VirtualService 的 https- 或者 tls- 端口流量会被视为透传 TCP 流量。

TCP 协议和 TLS 的子字段相对简单，仅包含 match 和 route 两部分，并且与 HTTP 相似，不再累述。

## 参考资料

- [什么是虚拟服务？](#)
- [Istio 虚拟服务参数配置](#)

# 虚拟服务参数配置

通过功能界面可以知道，服务网格的路由配置主要分为三个部分：HTTP 路由、TLS 路由和 TCP 路由。本页介绍创建和编辑 **虚拟服务** 时不同路由规则下的具体参数配置。

## 基本配置

UI 元素	YAML 字段	描述
名称	metadata.name	必填。虚拟服务名称。格式要求：小写字母、数字和中划线 (-) 组成，必须以小写字母开头，以小写字母或数字结尾，最长 63 个字符。
命名空间	metadata.namespace	必填。虚拟服务所属命名空间。 同一个命名空间内，请求身份认证不可重名。
应用范围	spec gateways	必填。虚拟服务应用的范围，包含两类：- 指定网关规则（可添加多个），可用于对外暴露网格内部服务；- 对所有边车（-mesh）生效。
所属服务	spec hosts	必填。应用虚拟服务的服务对象，可包含三类：- 来自 Kubernetes 注册中心的注册服务；- 来自服务条目的注册服务；- 服务域名。

## 路由配置

- 支持对不同路由类型添加多条，每个类型中执行顺序由上至下，排名在前的路由规则优先生效
- 支持快速拖动排序
- 支持路由名称收起，方便界面编辑时突出当前路由配置的具体信息

## HTTP 路由

### 基本信息

UI 元素	YAML 字段	描述
路由名称	<code>spec.http.-name</code>	必填。http 路由名称。格式要求：小写字母、数字和中划线 (-) 组成，必须以小写字母开头，以小写字母或数字结尾，最长 63 个字符。

### 路由匹配规则

路由匹配规则：HTTP 路由用于将 HTTP 请求路由到不同的目标实例，您可以按照 URI、端口以及 Header 等条件进行匹配。例如，您可以将所有请求路径为 `/reviews` 的请求路由到 `reviews` 服务上，同时将所有请求路径为 `/details` 的请求路由到 `details` 服务上。

必填。YAML 字段为 `spec.http.-name.match`。

通过 URI 路径、端口等方式对请求做匹配，可添加多条规则，执行顺序由上至下，优先使用排名在前的匹配规则。

UI 元素	YAML 字段	描述
匹配 URI	<code>spec.http.-name.match.uri</code>	可选。对请求的 URI 路径做匹配，有三种匹配方式：- 精确（exact）：字段完全匹配- 前缀（prefix）：对字段前缀的匹配- 正则（regex）：基于 RE2 样式正则表达式的匹配



UI 元素	YAML 字段	描述
匹配端口	spec.http.-name.match.port	可选。对请求的端口做匹配。
匹配 header	spec.http.-name.match.header	必填。对请求的 http header 做匹配，同样支持三种匹配方式： <ul style="list-style-type: none"> <li>- 精确 (exact)：字段完全匹配</li> <li>- 前缀 (prefix)：对字段前缀的匹配</li> <li>- 正则 (regex)：基于 RE2 样式正则表达式的匹配</li> </ul>

## 路由目标/重定向规则

路由目标：路由目标用于将请求路由到不同的目标实例，您可以按照版本、权重等条件进行匹配。例如，您可以将所有请求路径为 /reviews 的请求路由到 reviews 服务的 v1 版本上，同时将所有请求路径为 /reviews 的请求路由到 reviews 服务的 v2 版本上。

**路由目标** 功能和 **重定向** 功能为互斥功能，一条 **HTTP 路由规则** 内仅可以二选一。

!!! note

当用户开启 **\_\_代理\_\_** 开关后，该项及相关内容置灰。

### 路由规则

UI 元素	YAML 字段	描述
路由目标	spec.http.-name.route.-destination	可选。已匹配请求的路由目标，可添加多条，优先执行排名在前的路由目标。

UI 元素	YAML 字段	描述
服务名称	spec.http.-name.route.-destination.host	必填。路由目标服务的名称或 IP。
版本	spec.http.-name.route.-destination.subset	可选。列表内容来自所选服务的可用 <b>目标规则</b> 。
权重	spec.http.-name.route.weight	可选。本条 <b>路由目标</b> 内各条所占流量的分配权重。所有 <b>路由目标</b> 的权重总和应为 100。
端口	spec.http.-name.route.-destination.port.number	可选。路由目标服务的端口。

## 重定向

UI 元素	YAML 字段	描述
重定向	spec.http.-name.redirect	可选。重定向用于将请求转发至其他路径。
重定向	spec.http.-name.redirect.uri	必填。新的访问地址路径（URI）。

UI 元素	YAML 字段	描述
重定向路径		
author ity	spec.http.-name.redirect.autho rity	可选。URI 路径中认证信息部分，通常 // 表示开始， / 表示结束。
端口	spec.http.-name.redirect.port.n umber	可选。重定向服务的端口号。
响应 码	spec.http.-name.redirect.redire ctCode	可选。指定响应码，当返回指定错误码时，执行重定向操作，默认不填为 301。

## 可选设置

另外还提供了 6 个可选设置，您可以根据实际需求启用或禁用。

### 重写

重写功能允许您修改请求或响应的头、URI 或主体等部分。例如，您可以使用重写规则将所有请求中的 URI 路径 “/foo” 重写为 “/bar”，或者将响应头中的 Cache-Control 标头添加到所有响应中。

UI 元素	YAML 字段	描述
重写	spec.http.-name.rewrite	可选。默认关闭   可以重现完整路径，也可以仅重写 http 前缀。
重写 路径	spec.http.-name.rewrite.uri	必填。新的访问地址路径（URI）。

UI 元素	YAML 字段	描述
authORITY	spec.http.-name.redirect.auth -name.redirect.auth	可选。URI 路径中认证信息部分, 通常 // 表示开始, / 表示结束。

## 超时

超时规则允许您设置请求的最长等待时间。如果请求在规定的时间内没有得到响应, 则该请求将失败。这样可以帮助您避免因某些请求处理较慢而导致其他请求失败的情况。

UI 元素	YAML 字段	描述
超时时长	spec.http.-name.time out	可选。用于定义向目标服务发起请求后可容忍的超时时长。 输入格式: 数字 + 单位 (s) 。

## 重试

重试规则允许您指定在发生故障时应该如何重试请求。例如, 您可以设置最大的重试次数和间隔时间, 以及要重试的错误类型。这样可以帮助您确保请求能够成功处理, 从而提高可靠性。

UI 元素	YAML 字段	描述
重试次数	spec.http.-name.retries spec.http.-name.retries.attempt s	可选。默认关闭。重试功能用于定义当请求反馈为异常时再次发起请求的尝试次数。 可选。一个反馈异常的请求的可重试次数, 重试间隔默认为 25ms。当重试和超时功能均开启时, 重试次数和重试超时的乘积与超时时长取最短者生

UI 元素	YAML 字段	描述
		效，因此实际的重试次数可能会小于设置值。
重试 超时	spec.http.-name.retries.perTryTimeout	可选。每次重试的超时时长，默认与超时功能的设置 (http.route.timeout) 相同。输入格式：数字 + 单位 (s) 。
重试 条件	spec.http.-name.retries.retryOn	<p>可选。允许重试的前提条件，该项下包含多个复选内容：- 5xx：当上游服务器返回 5xx 响应码或无响应（断开/重置/读取超时），envoy 将重试。 - refused-stream：当上游服务器用错误码 REFUSED_STREAM 重置数据流，envoy 将重试。 - gateway-error：仅针对 502、503、504 错误进行重试。 - retrievable-status-codes：当上游服务器的返回码与响应码或请求头中 x-envoy-retrievable-status-codes 定义相同，则重试。 - reset：当上游服务器无响应时（断开/重置/读取超时），将重试。 - connect-failure：当与上游服务器的连接失败（连接超时等）导致请求失败，将重试。 - retrievable-headers：当上游服务器响应码与重试策略中定义匹配或与 x-ENVIGET-retrievable-header-NAME 头匹配，将尝试重试。 - envoy-ratelimited：当存在报头 x-ENVISENT-ratelimited 时，将重试。 - retrievable-4xx：</p>

UI 元素	YAML 字段	描述
		当上游服务器返回 4xx 响应码时 (目前仅有 409) , envoy 将重试。

## 故障注入

故障注入规则可以模拟错误或延迟，以帮助测试和验证系统的健壮性。例如，您可以使用故障注入规则模拟后端服务返回的错误响应。

UI 元素	YAML 字段	描述
故障注入	spec.http.-name.fault	可选。默认关闭。故障注入功能用于在应用层向目标服务注入故障，并可提供“延迟”和“终止”两种故障类型；使用故障注入时，不能启用超时和重试功能。
延迟时长	spec.http.-name.fault.delay.fixedDelay	必填。请求响应可延迟的时长 输入格式：数字 + 单位 (s) 。
故障注入占比	spec.http.-name.fault.delay.percentage	可选。所有请求中故障注入比率，默认 100%。
终止响应码	spec.http.-name.fault.abort.httpStatus	必填。用于终止当前请求的 http 返回码。
故障注入占比	spec.http.-name.fault.abort.percentage	可选。所有请求中故障注入比率，默认 100%。

## 代理虚拟服务

代理服务规则可用于将请求路由到网格中的代理服务，而非直接路由到后端服务。这样可以帮您在不影响后端服务的情况下，在请求路径上添加处理逻辑。

YAML 字段为 `spec.http.-name.delegate`，默认关闭。

- 虚拟服务代理功能可以将路由配置拆分至主从两个虚拟服务中，由主虚拟服务完成基本设置和匹配规则，由代理虚拟服务完成具体路由规则。
- 开启代理功能后，仅主虚拟服务的路由匹配生效，代理虚拟服务的路由匹配规则不需设置。
- 代理虚拟服务的路由规则会和主虚拟服务路由规则合并执行。
- 代理功能不支持嵌套，仅主虚拟服务可以开启代理功能。
- 当虚拟服务有“路由目标/重定向”配置时，不可配置代理。

UI 元素	YAML 字段	描述
代理虚拟服务	<code>spec.http.-name.delegate.name</code>	必填。用于代理的从虚拟服务 注意：一个已配置代理项的虚拟服务不可做为代理，即不可嵌套代理；一个配置了 <code>spec.hosts</code> 字段的虚拟服务不可作为代理。
所属命名空间	<code>spec.http.-name.delegate.namespace</code>	可选。从虚拟服务所属命名空间，默认与主虚拟服务相同。

注意，若需要使用代理虚拟服务配置时，基础配置中的 **所属服务** 必须为空。

流量镜像

流量镜像规则可让您将流量复制到指定的目标服务中。这样可以帮助您在不影响主要请求路径的情况下，对流量进行抽样和分析，以便进行异常检测和故障排除。

UI 元素	YAML 字段	描述。
流量镜像	<code>spec.http.-name.mirror</code>	可选。默认关闭。用于将请求流量复制至

UI 元素	YAML 字段	描述。
镜像至服务	<code>y spec.http.-name.mirror.host</code>	其他目标服务。 必填。流量镜像的传输目标服务。
流量镜像占比	<code>n spec.http.-name.mirror.mirrorPercentage: value</code>	可选。复制的请求流量与原请求流量的比率，默认 100%。
服务版本	<code>n spec.http.-name.mirror.subset</code>	可选。服务版本列表内容来自当前目标服务的可用“目标规则”。

## TLS 路由

TLS 路由用于将加密的 TLS 流量路由到不同的目标实例。您可以根据 SNI 头部的值选择将流量路由到哪个目标实例。例如，您可以将所有 SNI 头部为 `example.com` 的请求路由到 `example` 服务上，同时将所有 SNI 头部为 `blog.example.com` 的请求路由到 `blog` 服务上。

- 可添加多条，执行顺序由上至下，排名在前的路由规则优先生效
- 多条 TLS 路由规则之间可以拖动排序
- 每条路由可收起，仅显示路由名称

## 路由匹配规则

YAML 字段为 `spec.tls.-name.match` 。

通过端口 (`-port`) 和 SNI (`-port.sniHosts`) 名称方式对请求做匹配，可添加多条规则，执行顺序由上至下，优先使用排名在前的匹配规则。



## 路由目标

UI 元素	YAML 字段	描述
添加路由目标	spec.tls.-name.route	必填。添加路由目标信息，可添加多条，执行顺序由上至下
服务名称	spec.tls.-name.route.-destination.host	必填。目标服务名称，下拉列表包含当前命名空间下启用 tls 协议的所有服务
端口	spec.tls.-name.route.-destination.port.number	可选。目标服务端口
服务版本	spec.tls.-name.route.-destination.subset	可选。服务版本列表内容来自当前目标服务的可用“目标规则”。
权重	spec.tls.-name.route.-destination.weight	可选。本条“tls 路由”规则内各个“路由目标”所占流量的分配权重，各条规则的权重总和应为 100。

## TCP 路由

TCP 路由用于将 TCP 流量路由到不同的目标实例。您可以根据端口号进行匹配。例如，您可以将所有 端口号为: 3306 流量路由到 db 服务上，同时将所有 端口号为:22 流量路由到 ssh 服务上。

## 路由匹配规则

UI 元素	YAML 字段	描述
添加路由匹配	spec.tcp.-name.match	可选。通过端口 (-port) 方式对请求做匹配，可添加

UI 元素	YAML 字段	描述
配规则		多条规则，执行顺序由上至下，优先使用排名在前的匹配规则。
端口	spec.tcp.-name.match.-port	必填。tcp 端口号。

## 路由目标

UI 元素	YAML 字段	描述
添加路由目标	spec.tcp.-name.route	必填。添加路由目标信息，可添加多条，执行顺序由上至下。
服务名称	spec.tcp.-name.route.-destination.host	必填。目标服务名称，下拉列表包含当前命名空间下可用 tcp 协议的所有服务。
端口	spec.tcp.-name.route.-destination.port.number	可选。目标服务端口。
服务版本	spec.tcp.-name.route.-destination.subset	可选。服务版本列表内容来自当前目标服务的可用“目标规则”。
权重	spec.tcp.-name.route.-destination.weight	可选。本条“TCP 路由”规则内各个“路由目标”所占流量的分配权重，各条规则的权重总和应为 100。

## 目标规则

目标规则（DestinationRule）同样是服务治理中重要的组成部分，目标规则通过端口、服务版本等方式对请求流量进行划分，并对各请求分流量订制 envoy 流量策略，应用到流量的

策略不仅有负载均衡，还有最小连接数、离群检测等。

## 字段概念介绍

几个重要字段如下：

- Host

使用 Kubernetes Service 的短名称。含义同 VirtualService 中 **destination** 的 **host** 字段一致。服务一定要存在于对应的服务注册中心中，否则会被忽略。

- LoadBalancer

默认情况下，Istio 使用轮询的负载均衡策略，实例池中的每个实例依次获取请求。Istio 同时支持如下的负载均衡模型，可以在 DestinationRule 中为流向某个特定服务或服务子集的流量指定这些模型。

- 随机：请求以随机的方式转到池中的实例。
- 权重：请求根据指定的百分比转到实例。
- 最少请求：请求被转到最少被访问的实例。

- Subsets

**subsets** 是服务端点的集合，可以用于 A/B 测试或者分版本路由等场景。可以将一个服务的流量切分成 N 份供客户端分场景使用。**name** 字段定义后主要供 VirtualService 里 **destination** 使用。每个子集都是在 **host** 对应服务的基础上基于一个或多个 **labels** 定义的，在 Kubernetes 中它是附加到像 Pod 这种对象上的键/值对。这些标签应用于 Kubernetes 服务的 Deployment 并作为元数据信息 (Metadata) 来识别不同的版本。

- OutlierDetection

离群检测是减少服务异常和降低服务延迟的一种设计模式，主要是无感的处理服务异常并保证不会发生级联甚至雪崩。如果在一定时间内服务累计发生错误的次数超过了

预先定义的阈值，就会将该错误的服务从负载均衡池中移除，并持续关注服务的健康状态，当服务回复正常后，又会将服务再移回到负载均衡池。

## 目标规则列表介绍

目标规则列表展示了网格下的目标规则 CRD 信息，并提供了目标规则生命期管理能力。用户可以基于规则名称、规则标签做 CRD 筛选，规则标签如下：

- 服务版本
- 负载均衡
- 地域负载均衡
- HTTP 连接池
- TCP 连接池
- 客户端 TLS
- 离群检测

### 目标规则列表

目标规则列表

## 操作步骤

服务网格提供了两种创建方式：图形向导创建和 YAML 创建。通过图形向导创建的具体操作步骤如下：

1. 在左侧导航栏点击 **流量治理** -> **目标规则**，点击右上角的 **创建** 按钮。

创建

创建

2. 在 **创建目标** 界面中，先进行基本配置后点击 **下一步** 。

创建目标

创建目标

3. 按屏幕提示选择策略类型，并配置对应的治理策略后，点击 **确定** 。

治理策略

治理策略

4. 返回目标规则列表，屏幕提示创建成功。

创建成功

创建成功

5. 在列表右侧，点击操作一列的 **⋮**，可通过弹出菜单进行更多操作。

更多操作

更多操作

YAML 创建方式与虚拟服务相似，您可以直接借助内置模板创建 YAML 文件，如下图所示。

YAML 创建

YAML 创建

以下是一个 目标规则的 YAML 示例：

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  annotations:
    kube.daocloud.io/cluster: dywtest3
    kube.daocloud.io/indexes: '{"activePolices":"","cluster":"dywtest3","createdAt":"2023-08-10T02:18:04Z","host":"kubernetes","is_deleted":"false","labels":"","name":"dr01","namespace":"default"}'
    creationTimestamp: "2023-08-10T02:18:04Z"
  generation: 1
  managedFields:
    - apiVersion: networking.istio.io/v1beta1
      fieldsType: FieldsV1
```

```

fieldsV1:
  f:spec:
    .: {}
    f:host: {}
    f:trafficPolicy:
      .: {}
      f:portLevelSettings: {}
  manager: cacheproxy
  operation: Update
  time: "2023-08-10T02:18:04Z"
name: dr01
namespace: default
resourceVersion: "708763"
uid: ff95ba70-7b92-4998-b6ba-9348d355d44c
spec:
  host: kubernetes
  trafficPolicy:
    portLevelSettings:
      - port:
          number: 9980
status: {}

```

## 策略介绍

## 地域负载均衡

地域负载均衡是 Istio 具备的一个基于工作负载部署所在的 Kubernetes 集群工作节点上的地域标签，用作流量转发优化的策略。配置方式主要有：流量分发规则（权重分布）和流量转移规则（故障转移）：

- 流量分发规则：主要配置源负载位置访问到目标负载位置在不同的区域之间的流量权重分配
- 流量转移规则：流量的故障转移一般需要配合离群检测功能使用，可以达到更及时检测工作负载故障进行流量转移

注意，地域标签是在网络成员集群的工作节点上的 label，注意检查节点的标签配置：

- 地区： topology.kubernetes.io/region
- 可用区域： topology.kubernetes.io/zone
- 分区： **topology.istio.io/subzone** 分区是 istio 特有的配置，以实现更细粒度划分

另外地域是根据分层顺序进行匹配排列的，不同 **region** 的 **zone** 是两个不同的可用区域。

详情请参阅 Istio 官方文档：[<https://istio.io/latest/zh/docs/tasks/traffic-management/locality-load-balancing/>](https://istio.io/latest/zh/docs/tasks/traffic-management/locality-load-balancing/)

## 网关规则

网关规则（Gateway）用于将服务暴露于网格之外，相较于 Kubernetes 的 ingress 对象，istio-gateway 增加了更多的功能：

- L4-L6 负载均衡
- 对外 mTLS
- SNI 的支持
- 其他 Istio 中已经实现的内部网络功能：Fault Injection、Traffic Shifting、Circuit Breaking、Mirroring

## 概念介绍

对于 L7 的支持，网关规则通过与虚拟服务配合实现。几个重要主字段如下：

- Selector  
选择用于南北流量的 istio 网关，可以使用多个，也可以与其他规则共用一个。
- Servers  
对外暴露服务的相关信息，包括 hosts（服务名称）、监听端口、协议类型等。
- TLS  
提供对外的 mTLS 协议配置，用户可以启用三种 TLS 模式，并可以自定义 CA 证书等操

作。

示例：

```
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
      - istio-grafana.frognew.com
```

## 操作步骤

服务网格提供了两种创建方式：向导和 YAML。通过向导创建的具体操作步骤如下：

1. 在左侧导航栏点击 **流量治理** -> **网关规则**，点击右上角的 **创建** 按钮。

创建

创建

2. 在 **创建网关规则** 界面中，配置基本信息，并根据需要添加服务端后点击 **确定**。

创建网关规则

创建网关规则

3. 返回网关规则列表，屏幕提示创建成功。

创建成功

创建成功

4. 在列表右侧，点击操作一列的 **⋮**，可通过弹出菜单进行更多操作。

更多操作

更多操作



# 流量故障转移

在 Istio 中，`localityLbSetting` 中的流量故障转移规则（Traffic Failover Rule）允许你配置当某个本地性区域（如区域、可用区）的服务实例不可用时，将流量故障转移到其他区域或可用区。这种机制可以提高服务的可用性，特别是在跨多个区域或可用区部署的情况下。

## localityLbSetting 的配置结构

流量故障转移规则切换可以在网格级别（网格概览 -> 治理信息）进行配置，也可以在服务级别进行配置。

`localityLbSetting` 通常在 `DestinationRule` 中定义，具体包括三个主要部分：

1. **enabled**：是否启用区域感知负载均衡。
2. **distribute**：控制流量如何在不同的区域或可用区之间分布。
3. **failover**：定义从一个区域或可用区失败后切换到其他区域或可用区的策略。

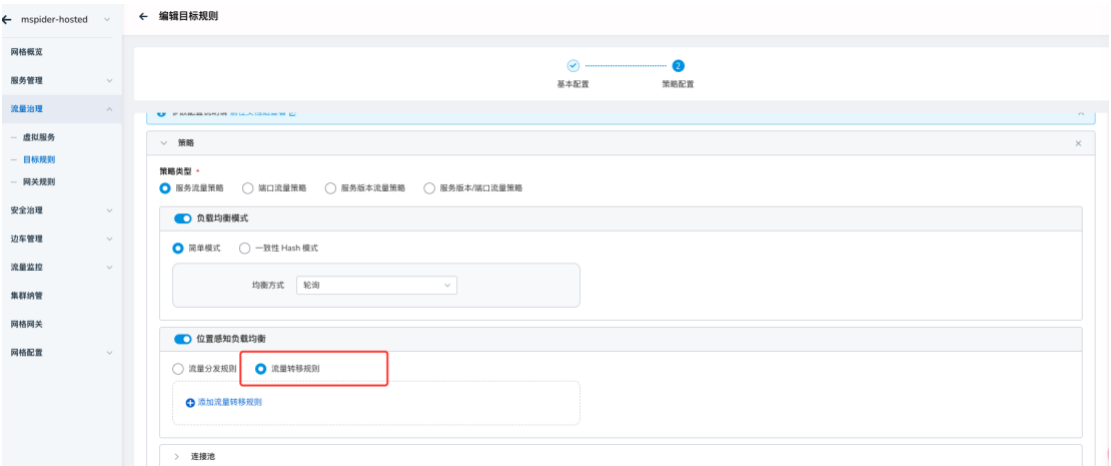
## 网格级别配置

在 **网格概览** 中，点击右上角的 ...，在弹出菜单中选择 **治理信息**：



网络级别

服务级别配置



服务级别

在实例级别、服务级别配置流量故障转移规则后，均需为每个服务单独配置「离群检测」（取决于具体服务触发离群的条件）。



## 离群检测

## 使用场景

1. **高可用性**：当一个区域的服务实例出现故障时，通过自动切换流量到其他区域，确保服务的连续性。
2. **灾难恢复**：在灾难或重大故障事件发生时，确保系统能够自动将流量重定向到其他可用的区域或数据中心。

## 小结

通过配置 `localityLbSetting` 中的 `failover` 规则，可以实现自动的区域切换，当某个区域出现故障时，流量会被自动切换到其他预先定义的区域。这种配置对于多区域部署的服务非常重要，有助于提升整体服务的可用性和鲁棒性。

## 安全治理

服务网格提供了一种[授权机制](#)和两种认证方式（[请求身份认证](#)和[对等身份认证](#)），用户可以在服务网格中通过向导和 YAML 编写两种方式创建、编辑资源文件，并可以针对网格全局、命名空间、工作负载三个层面创建规则。当资源创建成功后，Istiod 将转换为配置分发

至边车代理执行。

## 安全治理

安全治理

# 授权机制

服务网格提供的授权机制允许您控制对服务的访问。使用授权机制，您可以定义规则，指定允许哪些服务相互通信。这些规则可以基于各种因素，例如流量的源和目的地、使用的协议以及发出请求的客户端的身份。通过使用服务网格的授权机制，您可以确保只有经过授权的流量允许在您的服务网格中流动。

以下是如何使用服务网格的授权机制限制两个服务之间流量的示例：

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: my-auth-policy
spec:
  selector:
    matchLabels:
      app: my-service
  action: ALLOW
  rules:
    - from:
        - source:
            notNamespaces: ["my-namespace"]
      to:
        - operation:
            methods: ["GET"]
```

在此示例中，我们创建了一个 **AuthorizationPolicy** 资源，允许流量从除 my-namespace 以外的任何命名空间流向 my-service 服务，但仅限于 GET 请求。另请参见[图形界面的创建方式](#)。

## 请求身份验证

请求身份验证用于验证向服务发出请求的客户端的身份。

以下是如何使用请求级身份验证验证客户端身份的示例：

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: my-authn-policy
spec:
  selector:
    matchLabels:
      app: my-service
  jwtRules:
    - issuer: "https://my-auth-server.com"
      jwksUri: "https://my-auth-server.com/.well-known/jwks.json"
```

在此示例中，我们创建了一个 **RequestAuthentication** 资源，用于验证向 my-service 服务发出请求的客户端的身份。我们使用 JSON Web Token（JWT）来验证客户端，指定了发行者和用于验证令牌的公钥的位置。另请参见[图形界面的创建方式](#)。

## 对等身份验证

对等身份验证用于验证服务本身的身份。服务网格提供了几种对等身份验证选项，包括双向 TLS 身份验证和基于 JSON Web Token 的身份验证。

以下是如何使用双向 TLS 身份验证验证服务身份的示例：

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: my-peer-authn-policy
spec:
  selector:
    matchLabels:
      app: my-service
  mtls:
    mode: STRICT
```

在此示例中,我们创建了一个 **PeerAuthentication** 资源,要求 my-service 服务进行双向 TLS 身份验证。 我们使用 STRICT 模式,该模式要求客户端和服务端都提供有效的证书。 另请参见[图形界面的创建方式](#)。

更多说明,请参阅[服务网格身份和认证](#)。

## 对等身份认证

对等身份认证指的是在不对应用源码做侵入式修改的情况下,提供服务间的双向安全认证,同时密钥以及证书的创建、分发、轮转也都由系统自动完成,对用户透明,从而大大降低了安全配置管理的复杂度。

!!! note

启用对等身份认证后,相应的目标规则也需要开启 mTLS 模式,否则将无法访问。

一个对全网生效的严格 mTLS 策略。生效后,网格内部服务间访问将必须启用 mTLS。

YAML 示例:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "default"
  namespace: "istio-system"
spec:
  selector:
    matchLabels:
      app: reviews
  mtls:
    mode: STRICT
```

服务网格提供了两种创建方式:向导和 YAML。通过向导创建的具体操作步骤如下:

- 1.在左侧导航栏点击 **安全治理** -> **对等身份认证** , 点击右上角的 **创建** 按钮。

创建

创建

2. 在 **创建对等身份认证** 界面中，先进行基本配置后点击 **下一步**。

基本配置

基本配置

3. 按屏幕提示进行认证设置后，点击 **确定**。参阅 [mTLS 模式参数配置](#)。

认证设置

认证设置

4. 屏幕提示创建成功。

成功

成功

5. 在列表右侧，点击操作一列的 **⋮**，可通过弹出菜单进行更多操作。

更多操作

更多操作

!!! note

- 具体参数的配置，请参阅[对等身份认证参数配置](./params.md#\_2)
- 参阅[服务网格身份和认证](./mtls.md)。
- 更直观的操作演示，可参阅[视频教程](../../videos/mspider.md#\_5)

## 请求身份认证

当一个外部用户对网格内部服务发起请求时，可以采用这种认证模式。在这种模式下，使用

JSON Web Token (JWT) 实现请求加密。每个请求身份认证需要配置一个[授权策略](#)。

所有标签为 **app: httpbin** 的工作负载需要使用 JWT 认证。示例如下：

```
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
```

服务网格提供了两种创建方式：向导向导和 YAML。通过向导创建的具体操作步骤如下：

1. 在左侧导航栏点击 **安全治理** -> **请求身份认证**，点击右上角的 **创建** 按钮。

创建

创建

2. 在 **创建请求身份认证** 界面中，先进行基本配置后点击 **下一步**。

基本配置

基本配置

3. 按屏幕提示进行认证设置后，点击 **确定**，系统将验证所配置信息。参阅[认证设置的参数配置](#)。

认证设置

认证设置

4. 验证通过后，屏幕提示创建成功。

成功

成功

5. 在列表右侧，点击操作一列的 **⋮**，可通过弹出菜单进行更多操作。

更多操作

更多操作

!!! note



- 具体参数的配置，请参阅[请求身份认证参数配置](./params.md#\_5)。
- 参阅[服务网格身份和认证](./mtls.md)。
- 更直观的操作演示，可参阅[视频教程](../../videos/mspider.md#\_5)。

# 授权策略

授权策略类似于一种四层到七层的“防火墙”，它会像传统防火墙一样，对数据流进行分析和

匹配，然后执行相应的动作。 无论是来自内部还是外部的请求，都适用授权策略。

授权策略的参考 YAML 示例如下：

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: "ratings-viewer"
  namespace: default
spec:
  selector:
    matchLabels:
      app: ratings
  action: ALLOW
  rules:
    - from:
        - source:
            principals: ["cluster.local/ns/default/sa/bookinfo-reviews"]
      to:
        - operation:
            methods: ["GET"]
```

服务网格提供了两种创建方式：向导和 YAML。通过向导创建的具体操作步骤如下：

1. 在左侧导航栏点击 **安全治理** -> **授权策略** ， 点击右上角的 **创建** 按钮。

创建

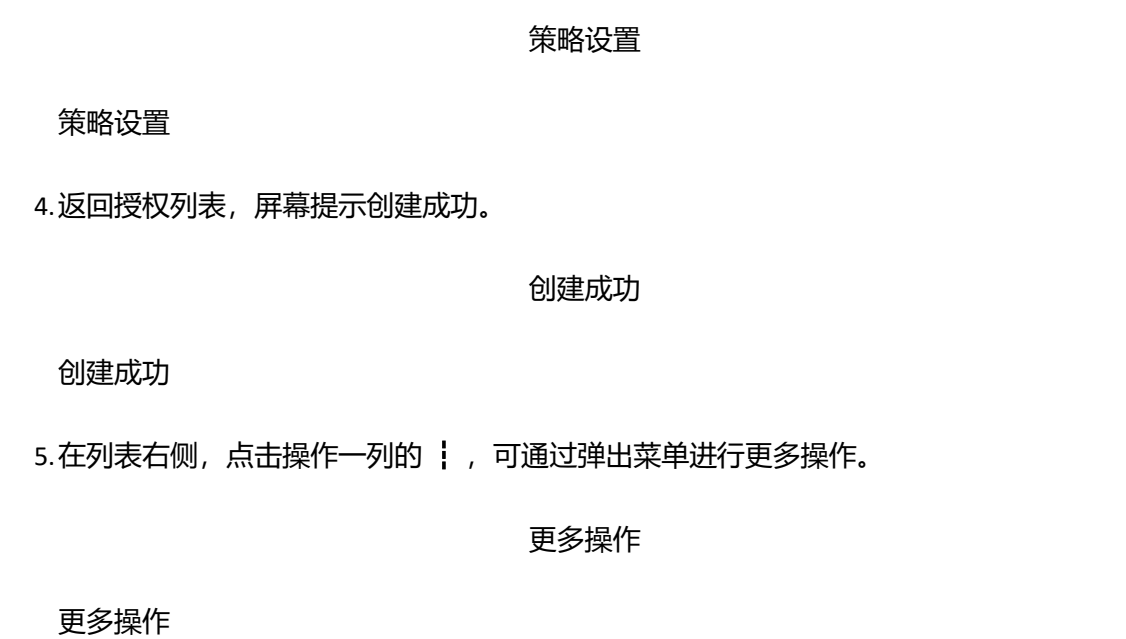
创建

2. 在 **创建授权策略** 界面中， 先进行基本配置后点击 **下一步** 。

基本配置

基本配置

3. 按屏幕提示进行策略设置后，点击 **确定** 。参阅[策略设置参数说明](#)。



!!! note

- 具体参数的配置，请参阅[\[授权策略参数配置\]\(../params.md#\\_8\)](#)。
- 更直观的操作演示，可参阅[\[视频教程\]\(../../videos/mspider.md\)](#)。

# 安全治理参数配置

本页介绍有关对等身份认证、请求身份认证、授权策略相关的参数配置。

## 对等身份认证

采用图形向导模式时，[对等身份认证](#)分为基本配置和认证设置两步，各参数说明如下。

## 基本配置

UI 项	YAML 字段	描述
名称	metadata.name	必填。对等身份认证名称，同一个命名空间下

UI 项	YAML 字段	描述
命名空间	metadata.namespace	不可重名。  必选。对等身份认证所属的命名空间。当选择网格的根命名空间时，将创建全局策略。全局策略仅能创建一个，需在界面中做检查，避免用户重复创建。
工作负载标签	spec.selector	可选。应用对等身份认证策略的工作负载选择标签，可添加多个标签，无需排序。
标签名称	spec.selector.matchLabels	必填。由小写字母、数字、连字符（-）、下划线（_）及小数点（.）组成
标签值	spec.selector.matchLabels.{标签名称}	必填。由小写字母、数字、连字符（-）、下划线（_）及小数点（.）组成

## 认证设置 - mTLS 模式

UI 项	YAML 字段	描述
mTLS 模式	spec.mTLS.mode	必填。用于设定命名空间的 mTLS 模式：- UNSET：继承父级选项。 否则视为 PERMISSIVE- PERMISSIVE：明文和 mTLS 连接- STRICT：仅 mTLS 连接- DISABLE：仅明文连接
为指定端口添加 mTLS 模式	spec.portLevelMtls	可选。针对指定端口设置 mTLS 规则，可添加多条规则，无需排序。- UNSET：继承父级选项。 否则视为 PERMISSIVE- PERMISSIVE：明文和 mTLS 连接- STRICT：仅 mTLS 连接- DISABLE：仅明文连接

## 请求身份认证

采用图形向导模式时，[请求身份认证](#)分为基本配置和认证设置两步，各参数说明如下。

## 基本配置

UI 项	YAML 字段	描述
名称	metadata.name	必填。请求身份认证名称，同一个命名空间下不可重名。
命名空间	metadata.namespace	必选。请求身份认证所属的命名空间。当选择网格的根命名空间时，将创建全局策略。全局策略仅能创建一个，需在界面中做检查，避免用户重复创建。同一个命名空间内，请求身份认证的名称不能重复。
工作负载标签	spec.selector	可选。应用请求身份认证策略的工作负载选择标签，可添加多条选择标签，无需排序。
标签名称	spec.selector.matchLabels	必填。由小写字母、数字、连字符 (-)、下划线 (_) 及小数点 (.)

UI 项	YAML 字段	描述
		组成
标签值	spec.selector.matchLabels.{标签名称}	必填。由小写字母、数字、连字符 (-)、下划线 (_) 及小数点 (.) 组成

## 认证设置

UI 项	YAML 字段	描述
添加 JWT 规则	spec.jwtRules	可选。用于用户请求认证的 JWT 规则，可添加多条规则。
Issuer	spec.jwtRules.issuers	必填。JSON Web Token (JWT) 签发人信息。
Audiences	spec.jwtRules.issuers.Audiences	可选。配置可访问的 audiences 列表, 如果为空, 将访问 service name。
jwtUri	spec.jwtRules.issuers.jwtUri	可选。JSON Web Key (JWK) 的 JSON 文件路径, 与 jwks 互斥, 二选一。例如 <https://www.googleapis.com/

UI 项	YAML 字段	描述
oauth2/v1/certs> jwks	spec.jwtRules.issuers.jwks	可选。JSON Web Key Set (JWKS) 文件内容，与 jwksUri 互斥，二选一。

更多请参阅 [OpenID Provider Metadata](#)。

## 授权策略

采用图形向导模式时，[授权策略](#)的创建分为 **基本配置** 和 **策略设置** 两步，各参数说明如下。

### 基本配置

UI 项	YAML 字段	描述
名称	metadata.name	必填。授权的策略名称。
命名空间	metadata.namespace	必选。授权策略所属命名空间，当选择网格根命名空间时，将创建全局策略，全局策略仅能创建一个，需在界面做检查，避免用户重复创建。同一个命名空间内，请求身份认证不可重名。
工作负载标签	spec.selector	可选。应用授权策略的

UI 项	YAML 字段	描述
		工作负载选择标签，可添加多条选择标签，无需排序。
标签名称	spec.selector.matchLabels	可选。由小写字母、数字、连字符 (-)、下划线 (_) 及小数点 (.) 组成。
标签值	spec.selector.matchLabels.{标签名称}	可选。由小写字母、数字、连字符 (-)、下划线 (_) 及小数点 (.) 组成。

## 策略设置

UI 项	YAML 字段	描述
策略动作	spec.action	可选。包含：- 允许 (allow) - 拒绝 (deny) - 审计 (audit) - 自定义 (custom) 选择自定义时，增加 provider 输入项。
Provider	spec.provider.name	必填。仅在 <b>策略动作</b>



UI 项	YAML 字段	描述
		选择为 <b>自定义</b> 时，才显示该输入框。
请求策略	spec.rules	可选。包含请求来源、请求操作、策略条件三部分，可添加多条，按顺序执行。
添加请求来源	spec.rules.-from	可选。请求来源可基于命名空间、IP 段等进行定义，可添加多条。各项参数参见下文 <a href="#">请求来源 Source</a> 。
添加请求操作	spec.rules.-to	可选。请求操作是对筛选出的请求执行的操作，例如发送至指定端口或主机，可添加多个操作。各项参数参见下文 <a href="#">请求操作 Operation</a> 。
添加策略条件	spec.rules.-when	必填。策略条件是一个可选设置，可以增加类似黑名单 (values)、白名单的限制条件

UI 项	YAML 字段	描述
		(notValues) , 可添加多个策略条件。各项参数参见下文 <a href="#">策略条件 Condition</a> 。

## 请求来源 Source

您可以增加请求来源（Source）。Source 指定一个请求的来源身份，对请求来源中的字段执行逻辑与运算。

例如，如果 Source 是：

- 主体为“admin”或“dev”
- 命名空间为“prod”或“test”
- 且 ip 不是“1.2.3.4”。

匹配的 YAML 内容为：

```
principals: ["admin", "dev"]
namespaces: ["prod", "test"]
notIpBlocks: ["1.2.3.4"]
```

具体字段说明如下：

Key 字段	类型	描述
principals	string[]	可选。从对等证书衍生的对等身份列表。对等身份格式为 "<TRUST_DOMAIN

Key 字段	类型	描述
		</ns><NAMESPACE>/sa/<SERVICE_ACCOUNT>" , 例如 "cluster.local/ns/default/sa/product-page" 。此字段要求启用 mTLS, 且等同于 source.principal 属性。如果不设置, 则允许所有主体。
notPrincipals	string[]	可选。对等身份的反向匹配列表。
requestPrincipals	string[]	可选。从 JWT 派生的请求身份列表。请求身份的格式为 "<ISS>/<SUB>" , 例如 "example.com/sub-1" 。此字段要求启用请求身份验证, 且等同于

Key 字段	类型	描述
		request.auth.principal 属性。如果不设置, 则允许所有请求主体。
notRequestPrincipals	string[]	可选。请求身份的反向匹配列表。
namespaces	string[]	可选。从对等证书衍生的命名空间。此字段要求启用 mTLS, 且等同于 source.namespace 属性。如果不设置, 则允许所有命名空间。
notNamespaces	string[]	可选。命名空间的反向匹配列表。
ipBlocks	string[]	可选。根据 IP 数据包来源地址进行填充的 IP 段列表。支持单个 IP (例如“1.2.3.4”) 和 CIDR (例如

Key 字段	类型	描述
		“1.2.3.0/24”)。这 等同于 source.ip 属性。如果不设 置，则允许所有 IP。
notIpBlocks	string[]	可选。IP 段的反 向匹配列表。
remoteIpBlocks	string[]	可选。根据 X-Forwarded-For 标头或代理协议 进行填充的 IP 段列表。要使用此 字段, 您必须在安 装 Istio 或在 ingress 网关使用 注解时在 meshConfig 下配 置 gatewayTopology 的 numTrustedProxie s 字段。支持单个

Key 字段	类型	描述
		IP (例如“1.2.3.4”) 和 CIDR (例如“1.2.3.0/24”)。这等同于 remote.ip 属性。如果不设置，则允许所有 IP。
notRemoteIpBlocks	string[]	可选。远程 IP 段的反向匹配列表。

## 请求操作 Operation

您可以增加请求操作 (Operation)。Operation 指定请求的操作，对操作中的字段执行逻辑与操作。

例如，以下操作将匹配：

- 主机后缀为“.example.com”
- 方法为“GET”或“HEAD”
- 补丁没有前缀“/admin”

```
hosts: ["*.example.com"]
methods: ["GET", "HEAD"]
notPaths: ["/admin*"]
```

Key 字段	类型	描述
hosts	string[]	可选。在 HTTP 请

Key 字段	类型	描述
		求中指定的主机
		列表。不区分大小写。如果不设置, 则允许所有主机。仅适用于 HTTP。
notHosts	string[]	可选。在 HTTP 请求中指定的主机反向匹配列表。不区分大小写。
ports	string[]	可选。连接中指定的端口列表。如果不设置, 则允许所有端口。
notPorts	string[]	可选。连接中所指定端口的反向匹配列表。
methods	string[]	可选。HTTP 请求中指定的方法列表。对于 gRPC 服务, 这将始终是“POST”。如果不设

Key 字段	类型	描述
		置, 则允许所有方法。仅适用于 HTTP。
notMethods	string[]	可选。HTTP 请求中所指定方法的反向匹配列表。
paths	string[]	可选。HTTP 请求中指定的路径列表。对于 gRPC 服务, 这将是 “/package.service/method” 格式的完全限定名称。如果不设置, 则允许所有路径。仅适用于 HTTP。
notPaths	string[]	可选。路径的反向匹配列表。

**在实际的操作情况中, 另外需要注意添加, 一些通用的 key**

- request.headers[User-Agent]
- request.auth.claims[iss]
- experimental.envoy.filters.network.mysql\_proxy[db.table]

更多关于 AuthorizationPolicy 的配置参数说明, 请参阅 [Istio Authorization Policy Conditions](#)



的说明。

## 策略条件 Condition

您还可以增加策略条件 (Condition)。Condition 指定其他必需的属性。

Key 字段	描述	支持的协议	Value 示例
request.headers	HTTP 请求头, 需要用 [] 括起来	HTTP only	["Mozilla/*"]
source.ip	源 IP 地址, 支持单个 IP 或 CIDR	HTTP and TCP	["10.1.2.3"]
remote.ip	由 X-Forwarded-For 请求头或代理协议确定的原始客户端 IP 地址, 支持单个 IP 或 CIDR	HTTP and TCP	["10.1.2.3", "10.2.0.0/16"]
source.namespace	源负载实例命名空间, 需启用双向	HTTP and TCP	["default"]

Key 字段	描述	支持的协议	Value 示例
source.principal	TLS 源负载的标识, 需启用双向 TLS	HTTP and TCP	["cluster.local/ns/default/sa/productpage"]
request.auth.principal	已认证过 principal 的请求	HTTP only	["accounts.my-svc.com/104958560606"]
request.auth.audiences	此身份验证信息的目标主体	HTTP only	["my-svc.com"]
request.auth.presenter	证书的颁发者	HTTP only	["123456789012.my-svc.com"]
request.auth.claims	Claims 来源于 JWT 。需要用 [] 括起来	HTTP only	["*@foo.com"]
destination.ip	目标 IP 地址, 支持单个 IP 或 CIDR	HTTP and TCP	["10.1.2.3", "10.2.0.0/16"]
destination.port	目标 IP 地址上的端口,	HTTP and TCP	["80", "443"]

Key 字段	描述	支持的协议	Value 示例
	必须在 [0, 65535] 范围内		
connection.sni	服务器名称指示, 需启用双向 TLS	HTTP and TCP	["www.example.com"]
experimental.envoy.filters.*	用于过滤器的实验性元数据匹配, 包装的值 [] 作为列表匹配	HTTP and TCP	["[update]"]

!!! note  
无法保证 `experimental.\*` 密钥向后的兼容性，可以随时将它们删除，但须谨慎操作。

# 服务网格身份和认证

身份 (Identity) 是任何安全基础架构的基本概念。在工作负载间开始通信时，通信双方必须交换包含身份信息的凭证以进行双向验证。在客户端，根据[安全命名](#)信息检查服务器的标识，以查看它是否是工作负载授权的运行程序。在服务器端，服务器可以根据授权策略确定客户端可以访问哪些信息，审计谁在什么时间访问了什么，根据他们使用的工作负载向客户收费，并拒绝任何未能支付账单的客户访问工作负载。

DCE 5.0 服务网格身份模型使用经典的 **service identity**（服务身份）来确定一个请求源端的身份。这种模型有极好的灵活性和粒度，可以用服务身份来标识人类用户、单个工作负载

或一组工作负载。 在没有服务身份的平台，服务网格可以使用其它可以对服务实例进行分组的身份，例如服务名称。

下面的列表展示了在不同平台上可以使用的服务身份：

- Kubernetes：Kubernetes 服务帐户
- GKE/GCE：GCP 服务帐户
- 本地（非 Kubernetes）：用户帐户、自定义服务帐户、服务名称、服务网格服务帐户或 GCP 服务帐户。自定义服务帐户引用现有服务帐户，就像客户的身份目录管理的身份一样。

## 身份和证书管理

服务网格 PKI (Public Key Infrastructure, 公钥基础设施) 使用 X.509 证书为每个工作负载都提供强大的身份标识。 **istio-agent** 与每个 Envoy 代理一起运行，与 **istiod** 一起协作来自动化的进行大规模密钥和证书轮换。下图显示了这个机制的运行流程。

workflow

服务网格通过以下流程提供密钥和证书：

1. **istiod** 提供 gRPC 服务以接受[证书签名请求](#) (CSRs) 。
2. **istio-agent** 在启动时创建私钥和 CSR，然后将 CSR 及其凭据发送到 **istiod** 进行签名。
3. **istiod** CA 验证 CSR 中携带的凭据，成功验证后签署 CSR 以生成证书。
4. 当工作负载启动时，Envoy 通过[秘密发现服务 \(SDS\)](#) API 向同容器内的 **istio-agent** 发送证书和密钥请求。
5. **istio-agent** 通过 Envoy SDS API 将从 **istiod** 收到的证书和密钥发送给 Envoy。
6. **istio-agent** 监控工作负载证书的过期时间。上述过程会定期重复进行证书和密钥轮换。

## 认证

服务网格提供两种类型的认证：

- **对等身份认证**：用于服务到服务的认证，以验证建立连接的客户端。服务网格提供[双向 TLS](#) 作为传输认证的全栈解决方案，无需更改服务代码就可以启用它。这个解决方案：

- 为每个服务提供强大的身份，表示其角色，以实现跨集群和云的互操作性。
- 保护服务到服务的通信。
- 提供密钥管理系统，以自动进行密钥和证书的生成、分发和轮换。

- **请求身份认证**：用于终端用户认证，以验证附加到请求的凭据。服务网格使用 JSON Web Token (JWT) 验证启用请求级认证，并使用自定义认证实现或任何 OpenID Connect 的认证实现（例如下面列举的）来简化的开发人员体验。

- [ORY Hydra](#)
- [Keycloak](#)
- [Auth0](#)
- [Firebase Auth](#)
- [Google Auth](#)

在所有情况下，服务网格都通过自定义 Kubernetes API 将认证策略存储在 **Istio config store**。Istiod 使每个代理保持最新状态，并在适当时提供密钥。此外，服务网格的认证机制支持宽容模式（permissive mode），以帮助您在强制实施前了解策略更改将如何影响您的安全状况。

## mTLS 认证

mTLS 全称为 Mutual Transport Layer Security，即双向传输层安全认证。mTLS 允许通信双方在 SSL/TLS 握手的初始连接期间进行相互认证。

DCE 5.0 服务网格通过客户端和服务端 [PEP](#)(Policy Enforcement Policy) 建立服务到服务的通信通道，PEP 被实现为 [Envoy 代理](#)。当一个工作负载使用 mTLS 认证向另一个工作负载发送请求时，该请求的处理方式如下：

1. 服务网格将出站流量从客户端重新路由到客户端的本地边车 Envoy。
2. 客户端 Envoy 与服务端 Envoy 开始双向 TLS 握手。在握手期间，客户端 Envoy 还做了[安全命名](#)检查，以验证服务器证书中显示的服务帐户是否被授权运行目标服务。
3. 客户端 Envoy 和服务端 Envoy 建立了一个双向的 TLS 连接，服务网格将流量从客户端 Envoy 转发到服务端 Envoy。
4. 服务端 Envoy 授权请求。如果获得授权，它将流量转发到通过本地 TCP 连接的后端服务。

服务网格将 **TLSv1\_2** 作为最低 TLS 版本为客户端和服务端配置了如下的加密套件：

- ECDHE-ECDSA-AES256-GCM-SHA384
- ECDHE-RSA-AES256-GCM-SHA384
- ECDHE-ECDSA-AES128-GCM-SHA256
- ECDHE-RSA-AES128-GCM-SHA256
- AES256-GCM-SHA384
- AES128-GCM-SHA256

## 宽容模式

服务网格双向 TLS 具有一个宽容模式 (permissive mode)，允许服务同时接受纯文本流量和双向 TLS 流量。这个功能极大地提升了双向 TLS 的入门体验。

在运维人员希望将服务移植到启用了双向 TLS 的服务网格上时，许多非服务网格客户端与非服务网格服务端之间的通信会产生问题。通常情况下，运维人员无法同时为所有客户端安装服务网格边车，甚至在某些客户端上没有这样做的权限。即使在服务端上安装了服务网格边车，运维人员也无法在不中断现有连接的情况下启用双向 TLS。

启用宽容模式后，服务可以同时接受纯文本和双向 TLS 流量。这个模式为入门提供了极大的灵活性。服务器中安装的服务网格边车立即接受双向 TLS 流量而不会打断现有的纯文本流量。因此，运维人员可以逐步安装和配置客户端服务网格边车发送双向 TLS 流量。一旦客户端配置完成，运维人员便可以将服务器端配置为仅 TLS 模式。

## 安全命名

服务器身份 (Server identity) 被编码在证书里，但服务名称 (service name) 通过服务发现或 DNS 被检索。安全命名信息将服务器身份映射到服务名称。身份 A 到服务名称 B 的映射表示“授权 A 运行服务 B”。控制平面监视 `apiserver`，生成安全命名映射，并将其安全地分发到 PEP。以下示例说明了为什么安全命名对身份验证至关重要。

假设运行服务 `datastore` 的合法服务器仅使用 `infra-team` 身份。恶意用户拥有 `test-team` 身份的证书和密钥。恶意用户打算模拟合法服务以检查从客户端发送的数据。恶意用户使用证书和 `test-team` 身份的密钥部署伪造服务器。假设恶意用户成功劫持（通过 DNS 欺骗、BGP/路由劫持、ARP 欺骗等）发送到 `datastore` 的流量并将其重定向到伪造的服务器。当客户端调用 `datastore` 服务时，它从服务器的证书中提取 `test-team` 身份，并用安全命名信息检查 `test-team` 是否被允许运行 `datastore`。客户端检测到 `test-team` 不允许运行 `datastore` 服务，认证失败。

请注意，对于非 HTTP/HTTPS 流量，安全命名不能保护其免于 DNS 欺骗，如攻击者劫持了 DNS 并修改了目的地 IP 地址。这是因为 TCP 流量不包含主机名信息，Envoy 只能依靠目的地 IP 地址进行路由，因此 Envoy 有可能将流量路由到劫持 IP 地址所在的服务上。这种 DNS 欺骗甚至可以在客户端 Envoy 接收到流量之前发生。

## 认证架构

您可以使用 **对等身份认证** 和 **请求身份认证** 策略为在服务网格中接收请求的工作负载指定认证要求。 网格运维人员使用 `.yaml` 文件来指定策略。部署后，策略将保存在 服务网格配置存储中。 服务网格控制器监视配置存储。

在任何的策略变更时，新策略都会转换为适当的配置，告知 PEP 如何执行所需的认证机制。控制平面可以获取公钥并将其附加到 JWT 验证的配置中。作为替代方案，Istiod 提供了 服务网格系统管理的密钥和证书的路径，并将它们安装到应用程序 Pod 用于双向 TLS。

服务网格异步发送配置到目标端点。代理收到配置后，新的认证要求会立即生效。

发送请求的客户端服务负责遵循必要的认证机制。对于 **请求身份认证**，应用程序负责获取 JWT 凭证并将其附加到请求。对于 **对等身份认证**，服务网格自动将两个 PEP 之间的所有流量升级为双向 TLS。如果认证策略禁用了双向 TLS 模式，则服务网格将继续在 PEP 之间使用纯文本。要覆盖此行为，请使用[目标规则](#)显式禁用双向 TLS 模式。

服务网格将这两种认证类型以及凭证中的其他声明（如果适用）输出到下一层：[授权](#)。

## 认证策略

本节中提供了更多服务网格认证策略方面的细节。正如[认证架构](#)中所说的，认证策略是对服务收到的请求生效的。要在双向 TLS 中指定客户端认证策略，需要在 `DestinationRule` 中设置 `TLSSettings`。

和其他的 服务网格配置一样，可以用 `.yaml` 文件的形式来编写认证策略。部署策略使用 `kubectl`。下面例子中的认证策略要求：与带有 `app: reviews` 标签的工作负载的传输层认证，必须使用双向 TLS：



```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-peer-policy"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: reviews
  mtls:
    mode: STRICT
```

## 策略存储

服务网格将网格范围的策略存储在根命名空间。这些策略使用一个空的 `selector` 应用到网格中的所有工作负载。具有命名空间范围的策略存储在相应的命名空间中。它们仅适用于其命名空间内的工作负载。如果您配置了 `selector` 字段，则认证策略仅适用于与您配置的条件匹配的工作负载。

Peer 和 **请求身份认证** 策略用 `kind` 字段区分，分别是 **PeerAuthentication** 和 **RequestAuthentication**。

## selector 字段

Peer 和 **请求身份认证** 策略使用 `selector` 字段来指定该策略适用的工作负载的标签。以下示例显示适用于带有 `app: product-page` 标签的工作负载的策略的 `selector` 字段：

```
selector:
  matchLabels:
    app: product-page
```

如果您没有为 `selector` 字段提供值，则服务网格会将策略与策略存储范围内的所有工作负载进行匹配。因此，`selector` 字段可帮助您指定策略的范围：

- 网格范围策略：为根命名空间指定的策略，不带或带有空的 `selector` 字段。

- 命名空间范围的策略：为非 root 命名空间指定的策略，不带有或带有空的 selector 字段。
- 特定于工作负载的策略：在常规命名空间中定义的策略，带有非空 selector 字段。

Peer 和 **请求身份认证** 策略对 selector 字段遵循相同的层次结构原则，但是服务网格以略微不同的方式组合和应用这些策略。

只能有一个网格范围的 **对等身份认证** 策略，每个命名空间也只能有一个命名空间范围的 **对等身份认证** 策略。当您为同一网格或命名空间配置多个网格范围或命名空间范围的 **对等身份认证** 策略时，服务网格会忽略较新的策略。当多个特定于工作负载的 **对等身份认证** 策略匹配时，服务网格将选择最旧的策略。

服务网格按照以下顺序为每个工作负载应用最窄的匹配策略：

1. 特定于工作负载的
2. 命名空间范围
3. 网格范围

服务网格可以将所有匹配的 **请求身份认证** 策略组合起来，就像它们来自单个 **请求身份认证** 策略一样。因此，您可以在网格或命名空间中配置多个网格范围或命名空间范围的策略。但是，避免使用多个网格范围或命名空间范围的 **请求身份认证** 策略仍然是一个好的实践。

## 对等身份认证

**对等身份认证** 策略指定服务网格对目标工作负载实施的双向 TLS 模式。支持以下模式：

- PERMISSIVE：工作负载接受双向 TLS 和纯文本流量。此模式在迁移因为没有边车而无法使用双向 TLS 的工作负载的过程中非常有用。一旦工作负载完成边车注入的迁移，应将模式切换为 STRICT。

- STRICT：工作负载仅接收双向 TLS 流量。
- DISABLE：禁用双向 TLS。从安全角度来看，除非您提供自己的安全解决方案，否则请勿使用此模式。
- UNSET：继承父作用域的模式。UNSET 模式的网格范围 **对等身份认证** 策略默认使用 **PERMISSIVE** 模式。

下面的 **对等身份认证** 策略要求命名空间 foo 中的所有工作负载都使用双向 TLS：

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-policy"
  namespace: "foo"
spec:
  mtls:
    mode: STRICT
```

对于特定于工作负载的 **对等身份认证** 策略，可以为不同的端口指定不同的双向 TLS 模式。

您只能将端口范围的双向 TLS 配置在工作负载声明过的端口上。以下示例为 app：

example-app 工作负载禁用了端口 80 上的双向 TLS，并对所有其他端口使用命名空间范

围的 **对等身份认证** 策略的双向 TLS 设置：

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-workload-policy"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: example-app
  portLevelMtls:
    80:
      mode: DISABLE
```

上面的 **对等身份认证** 策略仅在有如下 Service 定义时工作，将流向 example-service 服务

的请求绑定到 example-app 工作负载的 80 端口

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
  namespace: foo
spec:
  ports:
    - name: http
      port: 8000
      protocol: TCP
      targetPort: 80
  selector:
    app: example-app
```

## 请求身份认证

**请求身份认证** 策略指定验证 JSON Web Token (JWT) 所需的值。这些值包括：

- token 在请求中的位置
- 请求的 issuer
- 公共 JSON Web Key Set (JWKS)

服务网格会根据 **请求身份认证** 策略中的规则检查提供的令牌（如果已提供），并拒绝令牌无效的请求。当请求不带有令牌时，默认将接受这些请求。要拒绝没有令牌的请求，请提供授权规则，该规则指定对特定操作（例如，路径或操作）的限制。

如果 **请求身份认证** 策略使用唯一的位置，则可以在这些策略中指定多个 JWT。当多个策略与一个工作负载匹配时，服务网格会将所有规则组合起来，就好像这些规则被指定为单个策略一样。此行为对于开发接受来自不同 JWT 提供者的工作负载时很有用。但是，不支持具有多个有效 JWT 的请求，因为此类请求的输出主体未被定义。

## Principal

使用 **对等身份认证** 策略和双向 TLS 时，服务网格将身份从 **对等身份认证** 提取到

source.principal 中。同样，当您使用 **请求身份认证** 策略时，服务网格会将 JWT 中的身份赋值给 request.auth.principal 。使用这些 principal 设置授权策略并作为遥测的输出。

## 更新认证策略

您可以随时更改认证策略，服务网格几乎实时将新策略推送到工作负载。但是，服务网格无法保证所有工作负载都同时收到新政策。以下建议有助于避免在更新认证策略时造成干扰：

- 将 **对等身份认证** 策略的模式从 **DISABLE** 更改为 **STRICT** 时，请使用 **PERMISSIVE** 模式来过渡，反之亦然。当所有工作负载成功切换到所需模式时，您可以将策略应用于最终模式。您可以使用 服务网格遥测技术来验证工作负载已成功切换。
- 将 **请求身份认证** 策略从一个 JWT 迁移到另一个 JWT 时，将新 JWT 的规则添加到该策略中，而不删除旧规则。这样，工作负载将接受两种类型的 JWT，当所有流量都切换到新的 JWT 时，您可以删除旧规则。但是，每个 JWT 必须使用不同的位置。

下一步：设置[对等身份认证](#)和[请求身份认证](#)

## 命名空间边车管理

您可以勾选任意集群下的多个命名空间，启用、禁用边车注入或清除策略。

注意：如果某工作负载的边车注入设置为禁用，则该工作负载不会随所属命名空间启用注入。

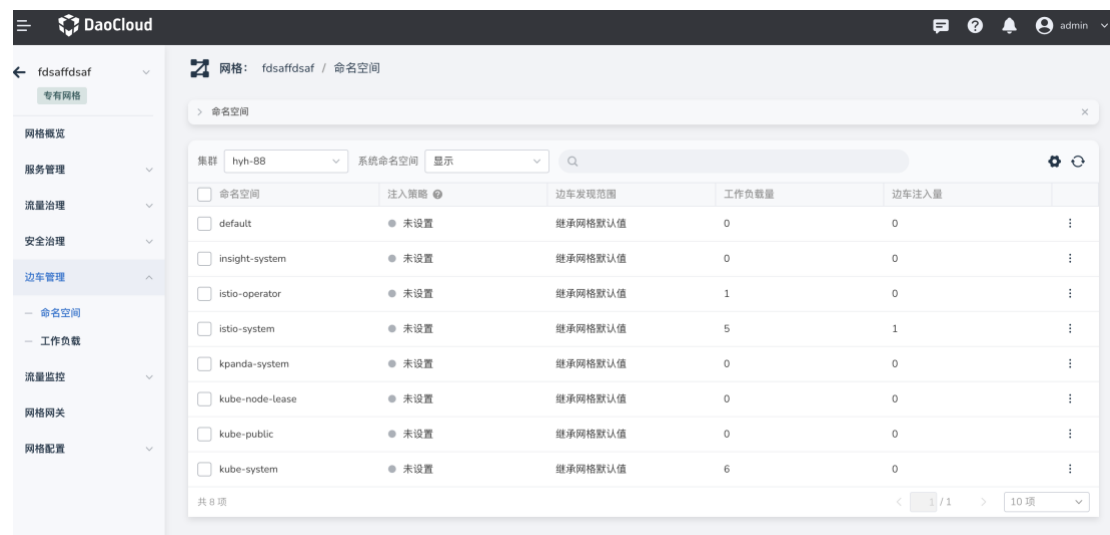
## 查看边车注入信息

在左侧导航栏中，点击 **边车管理** -> **命名空间**，可以查看对应网格下所有命名空间的边车

状态。

当命名空间较多时，可以按命名空间的名称进行排序，并通过搜索功能进行查找。 还可以选择是否显示以下系统命名空间：

- default
- insight-system
- istio-operator
- istio-system
- kpanda-system
- kube-node-lease
- kube-public
- kube-system



查看边车注入

## 启用边车注入

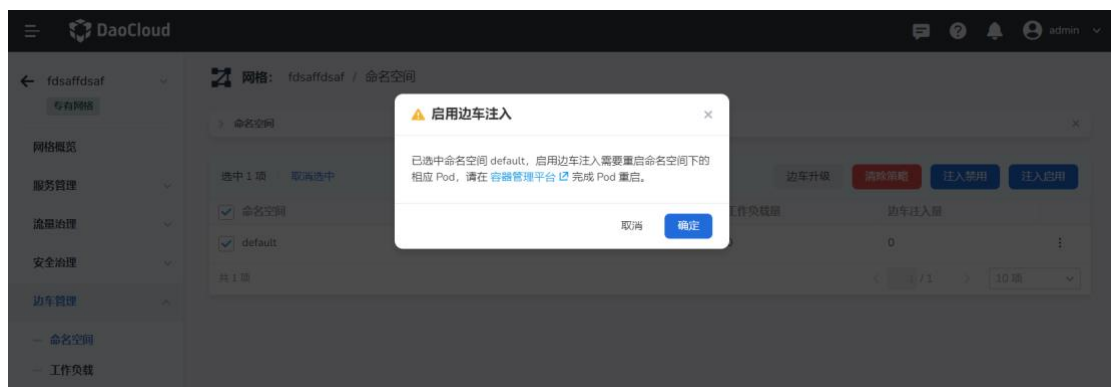
您可以勾选一个或多个命名空间，启用边车注入，具体步骤如下：

1. 勾选一个或多个未注入边车的命名空间，点击 **注入启用** ；



点击注入启用

2. 在弹出的对话框中，确认所选的命名空间，点击 **确定**。



确定

请按屏幕提示，重启对应的 Pod。

3. 返回命名空间的边车列表，可以看到刚刚所选命名空间的 **注入策略** 状态已变更为 **启用**。在用户完成工作负载的重启后，将完成边车注入，相关注入进度可查看 **边车注入量** 一列。



启用

## 禁用边车注入

您可以选择一个或多个命名空间，禁用边车注入，具体步骤如下：

1. 勾选一个或多个已启用边车注入的命名空间，点击 **注入禁用** ；



注入禁用

2. 在弹出的对话框中，确认所选的命名空间后，点击 **确定** 。



确定

请按屏幕提示，重启对应的 Pod。

3. 返回命名空间的边车列表，可以看到刚刚所选命名空间的 **注入策略** 状态已变更为 **禁用** 。 在用户完成工作负载的重启后，将完成边车禁用，相关卸载进度可查看 **边车注入量** 一列。





## 禁用

## 清除策略

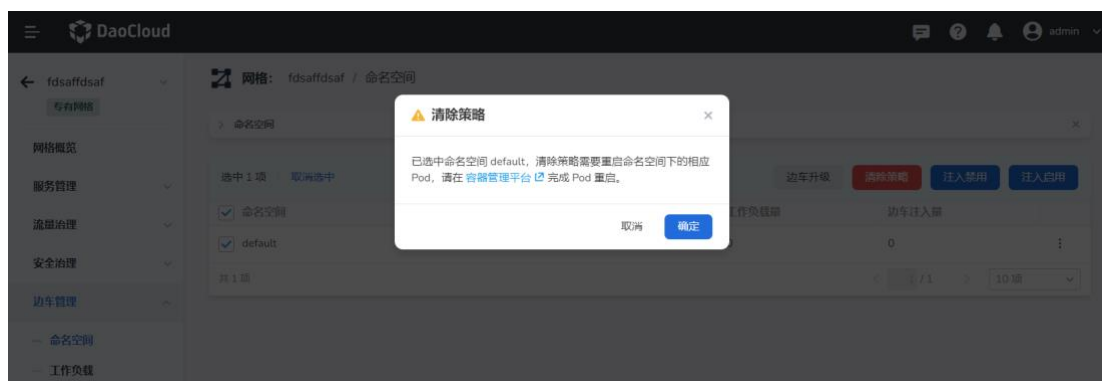
您可以选择一个或多个命名空间，清除命名空间层面的边车策略。清除后，命名空间下的工作负载的边车状态将仅受 **工作负载边车管理** 控制，具体步骤如下：

1. 勾选一个或多个已启用边车注入的命名空间，点击 **清除策略** 按钮；



## 清除策略

2. 在弹出的对话框中，确认所选的命名空间后，点击 **确定**。



确定

3. 返回命名空间的边车列表，可以看到刚刚所选命名空间的 **注入策略** 状态已变更为 **未设置**，此时用户可以在 **工作负载边车管理** 为特定工作负载设定边车注入策略。



未设置

下一步：[工作负载边车管理](#)

## 常见问题

- [命名空间边车配置与工作负载边车冲突](#)
- [边车占用大量内存](#)

## 工作负载边车管理

您可以对工作负载执行边车注入的查看、启用、禁用等操作，还可以为工作负载设置资源限

制。

## 查看边车注入信息

在左侧导航栏中，点击 **边车管理** -> **工作负载边车管理**，选择一个集群后，可以查看该集群下所有工作负载及其边车注入状态、所属命名空间、资源限制等信息。

### 工作负载边车列表

#### 工作负载边车列表


各列的含义如下：

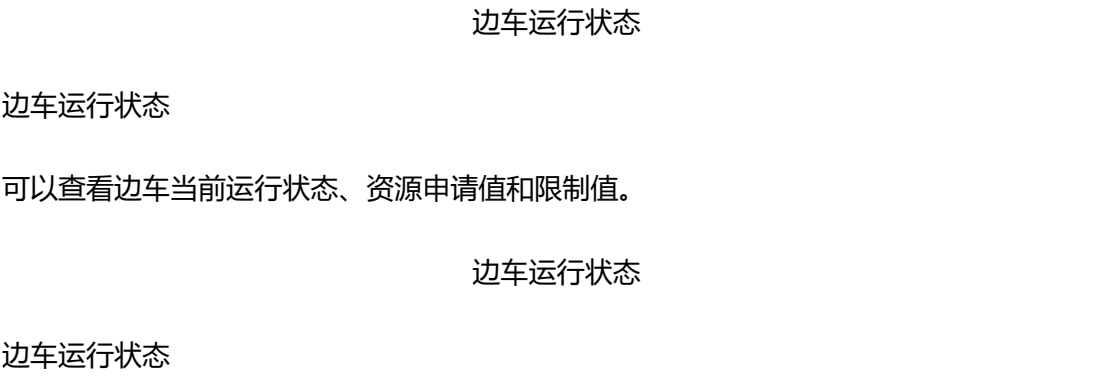
字段名称	含义
工作负载	所选集群下所有工作负载（不包含系统占用不可边车注入的工作负载）
状态	三个状态：已注入 - 已完成边车自动注入未注入 - 禁用边车自动注入 待重启 - 所述命名空间的 istio-injection 已发生变化，但相关 Pod 还未重启
命名空间	该工作负载所属命名空间；
服务	该工作负载相关服务，该项内容可能有多个，可采用扩缩列表，您点击后浮动窗显示所有内容；
已注入 Pod	该工作负载下 Pod 的注入情况；格式：已注入 Pod/所有可注入 Pod 如果工作负载的注入状态为 <b>已注入</b> ，但有部分 Pod 没有注入，例如 3/5，该项将高亮显示，提醒您有注入失败的 Pod，需及时处理
CPU 申请值/限制值	包含 <b>请求</b> 资源和 <b>限制</b> 资源两个数值，如果您未进行资源设置，该项为 <b>未设置</b> 。格式：请求 / 限制
内存申请值/限制	包含 <b>请求</b> 资源和 <b>限制</b> 资源两个数值，如果您未进行资源设置，该项

字段名称	含义
值	目为 <b>未设置</b> 。格式：请求 / 限制
操作	注入启用、清除策略、边车资源限制、查看边车状态和流量透传设置等  操作菜单

当工作负载较多时，可以对工作负载名称进行排序，并通过搜索功能查找当前集群下的目标工作负载。

## 查看边车运行状态

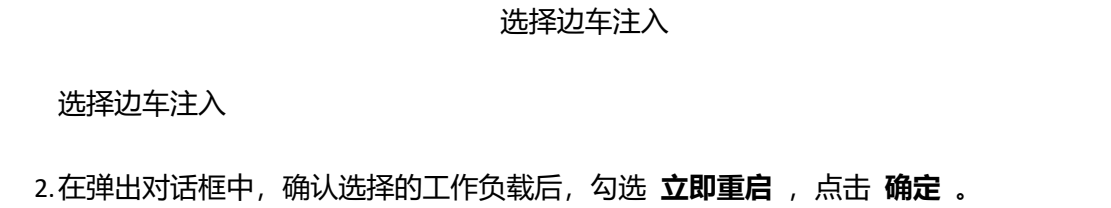
如果一个工作负载已注入边车,点击最后一列的  ,在弹出的菜单中选择 **查看边车状态** 。



## 启用边车注入

选择一个或多个工作负载后，可以启用边车自动注入功能。该操作会重启相关 Pod，因此请在执行该操作前确认 Pod 可以重启。具体操作步骤如下：

1. 选择一个或多个未启用边车注入的工作负载，点击右侧的 **注入启用** 。



2. 在弹出对话框中，确认选择的工作负载后，勾选 **立即重启** ，点击 **确定** 。

### 选择边车注入

#### 选择边车注入

3. 返回工作负载列表，所选工作负载的 **状态** 将发生变化，例如从 **未注入** 到 **已注入**。

在工作负载完成重启后，将完成边车注入，相关注入进度可查看 **已注入 Pod** 这一列。

### 边车注入成功

#### 边车注入成功

#### !!! note

如果工作负载所属命名空间已执行 \_\_注入启用/禁用\_\_ 操作但未重启工作负载，则该工作负载将无法执行新的边车相关操作。

需先完成重启，才能执行新的边车操作。

## 禁用边车注入

选择一个或多个工作负载后，可以禁用边车自动注入功能。该操作会重启相关 Pod，因此请

在执行该操作前确认 Pod 可以重启。具体操作步骤如下：

1. 选择一个或多个已启用边车注入的工作负载，点击右侧的 **注入禁用**。

### 选择边车禁用

#### 选择边车禁用

2. 在弹出对话框中，确认选择的工作负载数量是否正确，确认无误后，勾选 **立即重启**，点击 **确定**。

### 选择边车注入

#### 选择边车注入

3. 返回工作负载列表，可以看到所选工作负载的 **状态** 已变更为 **未注入**。相关卸载进度可查看 **已注入 Pod** 这一列。

边车禁用成功

边车禁用成功

## 边车资源限制

为了防止工作负载的资源被过量使用，可以为工作负载设置资源使用限制。该操作会重启相关 Pod，因此请在执行该操作前确认 Pod 可以重启。具体操作步骤如下：

1. 选择一个（或多个）已启用边车注入的工作负载，点击 **边车资源限制** 按钮。

边车资源限制

边车资源限制

2. 弹出对话框中，分别设置 CPU/内存的请求值与限制值。选中 **立即重启**，点击 **确定**。

选择边车注入

选择边车注入

3. 在工作负载边车管理列表中，可以看到指定工作负载的 **CPU 申请值/限制值** 和 **内存申请值/限制值** 内数据已更新。

## 边车升级

在 DCE 5.0 服务网格中，边车指的是 Envoy 代理，用于实现服务网格中的流量控制和路由规则等功能。边车升级是指将 Envoy 代理从旧版本升级到新版本。

需要升级边车的原因包括：

1. 安全更新：新版本可能修复了安全漏洞或其他安全问题，为了确保服务网格的安全性，需要将边车升级到最新版本。
2. 功能增强：新版本可能增加了一些新的功能或改进了一些功能，以提高服务网格的性能

和可靠性。

3. 错误修复：新版本可能修复了一些错误或 Bug，以提高服务网格的稳定性和可靠性。

4. 版本过时：随着时间的推移，旧版本的 Envoy 代理可能会变得过时并且不再受支持，

因此需要将其升级到最新版本以获得更好的支持和维护。

但是，在进行边车升级之前，需要进行充分的测试和验证，以确保升级过程不会对服务网格产生负面影响。

具体操作步骤，请参阅[边车版本升级](#)。

## 常见问题

- [命名空间边车配置与工作负载边车冲突](#)
- [边车占用大量内存](#)

## 边车流量透传

流量透传 (traffic passthrough) 指的是工作负载的全部或部分上游、下游流量不经边车转发，直接发送至工作负载本身。

流量透传在网格中的使用场景主要包括需要直接访问外部服务的情况，例如第三方 API 或云服务，这样可以避免将外部流量引入服务网格的管理之下；同时，流量透传也适用于简化网格配置，避免不必要的复杂性，尤其是在性能敏感的应用中，通过减少网络延迟来提高响应速度。此外，流量透传有助于兼容遗留系统或特定协议，确保它们正常工作，而不受网格的影响；在微服务架构中，它允许将某些服务直接暴露给客户端，而不经网格的流量管理，从而提供更灵活的服务访问方式。

DCE 5.0 服务网格实现了对工作负载出站/入站流量的边车透传可控，可针对特定端口、IP 段

实现拦截设置。

- 功能设置对象：工作负载
- 设置参数：端口、IP 段
- 流向：入站、出站

流量透传相关字段：

traffic.sidecar.istio.io/excludeInboundPorts

traffic.sidecar.istio.io/excludeOutboundPorts

traffic.sidecar.istio.io/excludeOutboundIPRanges

## 启用流量透传

本节说明如何在 DCE 图形界面上启用/禁用流量透传。

1. 进入某个网格，点击 **边车管理** -> **工作负载边车管理** 。

工作负载边车管理

工作负载边车管理

2. 点击某个负载右侧的 **⋮**，在弹出菜单中选择 **流量透传设置** 。

点击菜单项

点击菜单项

3. 设置流量透传的参数后，勾选 **立即重启工作负载**，点击 **确认变更** 。

流量透传设置

流量透传设置

- 入站：仅支持端口，即从外部访问网格内负载的端口
- 出站：可设置目标的端口或 IP 段

4. 如果设置无误，右上角将出现 **流量透传设置成功** 的提示消息。您还可以[查验流量透传](#)

[效果](#)。



## 成功设置

## 成功设置

5. 如果流量透传已启用，上述第 3 步的 **流量透传设置** 弹窗将显示设置的参数，可点击右侧的 **x**，勾选 **立即重启工作负载**，点击 **确认变更** 来禁用流量透传。

## 禁用流量透传

## 禁用流量透传

## 查验流量透传效果

在真实的网络集群中，查验流量透传前后的效果。

### 1. 准备工作

- 准备一个网络集群，例如 10.64.30.130
- 在命名空间中，配置工作负载 **helloworld**，并注入边车
- 启用流量透传，然后比对该负载的流量路由变化

## 工作负载

## 工作负载

### 2. 通过控制台或 ssh 登录到网络。

```
ssh root@10.64.30.130
```

### 3. 查看命名空间中的 svc，获取 clusterIP 和 Port：

```
$ kubectl get svc -n default
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
helloworld	ClusterIP	10.211.201.221	<none>	5000/TCP	39d
kubernetes	ClusterIP	10.211.0.1	<none>	443/TCP	62d
test-cv	NodePort	10.211.72.8	<none>	2222:30186/TCP	62d

### 4. 执行 curl 命令查看 helloworld 的流量路由：

=== “启用流量透传前”

```

```bash
curl -sSI 10.211.201.221:5000/hello
```
```none
HTTP/1.1 200 OK
content-type: text/html; charset=utf-8
content-length: 65
server: istio-envoy # (1)!
date: Tue, 07 Feb 2023 03:08:33 GMT
x-envoy-upstream-service-time: 100
x-envoy-decorator-operation: helloworld.default.svc.cluster.local:5000/*
```

```

1. 流量经过 istio-envoy，即边车的代理

### === “启用流量透传后”

```

```bash
curl -sSI 10.211.201.221:5000/hello
```
```none
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 65
Server: Werkzeug/0.12.2 Python/2.7.13 # (1)!
Date: Tue, 07 Feb 2023 03:08:10 GMT
```

```

1. 流量直接进入工作负载本身

## 更改边车日志级别

边车日志，用于记录工作负载的边车的工作情况，通过控制日志级别，可以控制边车日志的输出，从而减少日志的输出，减少日志的存储和传输。

在部署网格实例时，DCE 5.0 支持配置全局默认的边车日志级别，默认情况下所有工作负载的边车都会采用此配置。

## 边车日志级别配置

- 全局默认边车日志级别：默认情况下，全部边车日志的级别，可以在网格实例的边车信息中进行配置
- 临时修改工作负载边车日志级别：适用于单个工作负载的边车日志级别，可通过到工作负载的边车容器内临时修改

## 全局默认边车日志级别

!!! warning

全局默认边车日志级别，需由网格管理员进行修改。

1. 登录控制台，进入网格实例详情页，点击 **边车信息** 菜单，进入边车信息修改页面

image

image

2. 在边车信息修改页面，可以修改全局默认边车日志级别，修改完成后，点击 **保存** 按钮，

即可保存修改

image

image

## 临时修改工作负载边车日志级别

通常在临时分析问题，需要修改某个工作负载的边车日志级别，所以 DCE 5.0 服务网格支持到工作负载的边车容器内临时修改边车的日志级别。

这里需要使用 `kubectl` 来进行容器边车的配置更新，您需要访问集群控制台，并打开终端，

执行如下命令：

```
kubectl -n <namespace> exec -it <pod-name> -c istio-proxy -- curl -X POST localhost: 15000 /logging?level=<log level>
```

- <namespace> ：工作负载所在的命名空间
- <pod-name> ：工作负载的 Pod 名称

- <log level> :边车日志级别,可选值为:trace 、debug 、info 、warning 、error 、critical 、off 等
- istio-proxy : 边车容器名称, 不需要修改
- localhost: 15000 : 边车容器的监听地址, 不需要修改

如果需要将 **default** 命名空间下的 **productpage-v1-5b4f8f9b9f-8q9q2** 工作负载的边车日志

级别修改为 **debug** , 则执行如下命令:

```
kubectl -n default exec -it productpage-v1-5b4f8f9b9f-8q9q2 -c istio-proxy -- curl -X POST localhost: 15000/logging?level=debug
```

执行完成后, 可以在页面点击查看日志, 确认边车日志级别是否已经修改成功。

image

image

## 流量监控

流量监控图表采用了 Istio 的原生 Grafana 图表, 目前采用了以下 4 个图表板块:

- 全局监控: 网格全局的各类资源统计
- 性能监控: 网格控制面和数据面组件性能展示, 还包含基于组件版本的性能统计信息
- 服务监控: 网格内所有注入边车的服务性能统计信息, 并提供多种筛选方式, 您可以从命名空间、请求来源、服务版本等多个维度筛选显示内容
- 工作负载监控: 网格内所有注入边车的工作负载性能统计信息, 并提供多种筛选方式, 您可以从命名空间、所属服务、请求来源、服务版本等多个纬度筛选显示内容

还提供了流量拓扑, 监测不同命名空间、采集源的流量, 还增加了流量动画, 方便查看动态的流量流转情况。

traffic topology

traffic topology

有关更多信息, 请参阅 Istio 的 [Grafana Dashboard 说明](#)。

# 流量拓扑

DCE 5.0 服务网格提供了动态流转的流量拓扑功能。

在左侧导航栏中，点击 **流量监控** -> **流量拓扑**，您可以选择 **视图方式**、**命名空间**、采集源、时间来查看服务的拓扑关系。

流量拓扑

流量拓扑

## 显示设置

共有 3 个选项：

- 命名空间边界：按命名空间分区显示服务
- 显示空闲节点
- 开启动画：展示流量流转的动态方向

显示设置

显示设置

服务拓扑图支持平移、缩放等操作。

## 图例

点击左下角的 **图例** 按钮，可以查看当前线条、圆圈、颜色所代表的含义。

图例

图例

服务用圆圈表示，圆圈的颜色表示了该服务的健康状态：

- 健康（灰色）：错误率 = 0 并且 延时不超过 100 ms
- 警告（橙色）：0 < 错误率 ≤ 5% 或 100 ms < 延时 ≤ 200 ms
- 异常（红色）：错误率 > 5% 或延时 > 200 ms
- 未知（虚线）：未获得任何指标数据

## 服务指标数据信息

点击任意服务，会弹出一个侧边栏，基于协议类型展示服务相关指标：

- HTTP 协议：错误率（%）、请求速率（RPM）、平均延时（ms）
- TCP 协议：连接数（个）、接收吞吐量（B/S）、发送吞吐量（B/S）
- 治理信息：查看治理的虚拟服务、目标服务、网关等

侧边栏

侧边栏

## 监控看板指标说明

本文用于补充介绍，网格实例中流量监控中的各个监控看板的指标的含义，方便用户查询和定义

## 全局监控

全局监控

全局监控

| 分类 | 参数    | 参数介绍                       | 计算方式               |
|----|-------|----------------------------|--------------------|
| 常规 | 全局请求总 | “Global Request Volume” 代表 | OPS/s = 总操作数 / 时间段 |

| 分类 | 参数               | 参数介绍  | 计算方式   |
|----|------------------|---|--|
|    | 量                | 整个服务网格中的请求流量总量, Ops/s 标识每秒执行的操作或请求的数量。在 Istio 中, 这个度量通常用来评估整个服务网格的流量  | (秒)  |
|    | 全局成功率 (非 5xx 响应) |   | 全局成功率 (非响应率 5xx) = $\frac{(\text{总请求} - \text{5xx 响应})}{\text{总请求}} * 100$           |
|    | 4xxs             | 4xxs 是客户端错误, 表示请求可能有问题 (例如, 404 表示未找到)                                | ops/s = 每秒收到的 4xx 响应的数量  |
|    | 5xxs             | 5xxs 是服务器错误, 表示服务器未能正确处理有效请求 (例如, 500 表示内部服务器错误)                      | ops/s = 每秒收到的 5xx 响应的数量  |
|    | 虚拟服务             | 统计一段时间范围内服务网格内的虚拟服务的净增长情况, 通过比较添加和删除事件的数量, 可以了解服务的动态变化, 并能有助于识别潜在的问题。 | $\max(\text{Virtual Services \&\& add}) - \max(\text{Virtual Services \&\& delete})$ |
|    | 目标规则             | 统计一段时间范围内服务网格内的目标规则 (Destination                                      | $\max(\text{Destination Rule \&\& add}) - \max(\text{Destination Rule \&\& delete})$ |

| 分类 | 参数     | 参数介绍   | 计算方式   |
|----|--------|--|--|
|    |        | Rule) 的净增长情况, 通过比较添加和删除事件的数量, 可以了解服务的动态变化, 并可能有助于识别潜在的问题。                              |  |
|    | 网关规则   | 统计一段时间范围内服务网格内的网关规则 (Gateway) 的净增长情况, 通过比较添加和删除事件的数量, 可以了解服务的动态变化, 并可能有助于识别潜在的问题。      | $\max(\text{Gateway \&\& and}) - \max(\text{Gateway \&\& delete})$                 |
|    | 工作负载条目 | 统计一段时间内 WorkloadEntry 的增长趋势, 通过观察 Workload 的变化, 可以快速了解服务的动态变化, 并可能有助于识别潜在的问题。          | $\max(\text{WorkloadEntry \&\& and}) - \max(\text{WorkloadEntry \&\& delete})$     |
|    | 服务条目   | 统计一段时间内 Service Entries 的增长趋势, 通过观察 Service Entries 的变化, 可以快速了解服务的动态变化, 并可能有助于识别潜在的问题。 | $\max(\text{Service Entries \&\& and}) - \max(\text{Service Entries \&\& delete})$ |
|    | 对等认证策  | 统计一段时间内  | $\max(\text{PeerAuthentication Policies \&\& and}) -$                              |



| 分类             | 参数      | 参数介绍   | 计算方式   |
|----------------|---------|--|--|
|                | 略       | PeerAuthentication Policies 的增长趋势, 通过观察 PeerAuthentication Policies 的变化, 可以快速了解服务的动态变化, 并可能有助于识别潜在的问题。               | $\max(\text{PeerAuthentication Policies} \&\& \text{delete})$  |
|                | 请求认证策略  | 统计一段时间内 RequestAuthentication Policies 的增长趋势, 通过观察 RequestAuthentication Policies 的变化, 可以快速了解服务的动态变化, 并可能有助于识别潜在的问题。 | $\max(\text{RequestAuthentication Policies} \&\& \text{and}) - \max(\text{RequestAuthentication Policies} \&\& \text{delete})$ |
|                | 授权策略    | 统计一段时间内 Authorization Policies 的增长趋势, 通过观察 Authorization Policies 的变化, 可以快速了解服务的动态变化, 并可能有助于识别潜在的问题。                 | $\max(\text{Authorization Policies} \&\& \text{and}) - \max(\text{Authorization Policies} \&\& \text{delete})$                 |
| HTTP/GRPC 工作负载 | Service | 服务名称, 名称组合服务所在的命名空间和 Kubernetes 的 Service 访问信息   |  |
|                | 工作负载    | 工作负载名称, 名称组合工作负载   |  |

| 分类       | 参数      | 参数介绍  | 计算方式  |
|----------|---------|---|---|
|          |         | 载所在的命名空间  |   |
|          | 请求量     | 请求的数量, 表示特定时间段内的请求总数  | <code>sum(rate(istio_requests_total{...}[1m]))</code>             |
|          | P50 延迟  | 中位数延迟, 50%的请求在的请求延迟在这个时间以下完成                                      | <code>histogram_quantile(0.50, sum(rate(...)))</code>             |
|          | P90 延迟  | 90%的请求在的请求延迟在这个时间以下完成   | <code>histogram_quantile(0.90, sum(rate(...)))</code>             |
|          | P99 延迟  | 99%的请求在的请求延迟在这个时间以下完成   | <code>histogram_quantile(0.99, sum(rate(...)))</code>             |
|          | 成功率     | 成功率, 在查询时间段内成功响应 (响应状态码不等于 5xx) 的请求所占的百分比,                        | <code>sum(rate(... response_code!~"5.*")) / sum(rate(...))</code> |
| TCP 工作负载 | Service | 工作负载名称, 从 Istio 提供的指标中获取 destination_service 标签, 这个标签包含了服务的名称。    | destination_workload  |
|          | 工作负载    | 工作负载名称, 从 Istio 提供的指标中获取 destination_workload 标签, 这个标签包含了工作负载的名称。 | destination_service   |
|          | 发送字节数   | 每秒发送的字节量  | 通过 istio_tcp_sent_bytes_total                                     |

| 分类             | 参数    | 参数介绍  | 计算方式   |
|----------------|-------|---|--|
|                |       |   | 计算累计的 TCP 字节, 使用<br><br>rate 计算对应的发送速率   |
|                | 接收字节数 | 每秒接收的字节量  | 通过<br><br>istio_tcp_received_bytes_total<br><br>计算累计的 TCP 字节, 使用<br><br>rate 计算对应的发送速率 |
| 基于版本的 Istio 组件 |       | Istio 组件构建版本的可视化展示, 展示各个组件的版本分布, 以及它们在不同集群中的部署情况。这对于了解 Istio 部署的健康状况和一致性非常有用。 | sum(istio_build{mesh_id="\$mesh"}) by (component, tag, mesh_cluster)                   |

## 性能监控

### 性能监控

#### 性能监控

| 分类       | 参数            | 参数介绍  | 计算方式  |
|----------|---------------|---|---|
| VCPU 使用量 | vCPU / 1k rps | 展示 Istio 每千次请求 (1k rps) 所消耗的虚拟 CPU (vCPU) 资源, 主要查询了 istio-ingressgateway 和 istio-proxy。为了保障查询效率, Istio 限制仅当 | (sum(irate(container_cpu_usage_seconds_total{namespace!="istio-system",container="istio-proxy"}[1m]))/(round(sum(irate(istio_requests_total[1m])), 0.001)/1000))/(sum(irate(istio_requests_total{source_workload="istio-ingressgateway"}[1m])) > bool 10) |

| 分类             | 参数             | 参数介绍  | 计算方式   |
|----------------|----------------|---|--|
|                |                | istio-ingressgateway 请求大于 10 时才进行 istio-proxy 统计。 |  |
|                | vCPU           | 展示 Istio 中虚拟 CPU (vCPU) 的整体使用情况                   |  |
| 内存和数<br>据      | 内存用量           | 展示 Istio 系统组件中的内存使用情况, 统计单位为 bytes                | $\frac{\text{sum}(\text{container\_memory\_working\_set\_bytes}\{\text{pod}=\sim\text{"istio-ingressgateway-."}\})}{\text{count}(\text{container\_memory\_working\_set\_bytes}\{\text{pod}=\sim\text{"istio-ingressgateway-.", container!=\text{"POD"}\})}$  |
|                | 数据传输量<br>B/s   | 展示 Istio 系统组件的每秒传输的字节数, 统计单位为 Bps                 | $\frac{\text{sum}(\text{irate}(\text{istio\_response\_bytes\_sum}\{\text{source\_workload}=\text{"istio-ingressgateway"}, \text{reporter}=\sim\text{"$reporter"}, \text{destination\_mesh\_id}=\text{"$mesh"}\}[1\text{m}]))}{\text{sum}(\text{istio\_build}\{\text{mesh\_id}=\text{"$mesh"}\}) \text{ by } (\text{component}, \text{tag}, \text{mesh\_cluster})}$ |
| Istio 组<br>件版本 | Istio 组件<br>版本 | 展示 Istio 组件的版本统计信息, 图例格式将包括组件名称、标签和网格集群。          | $\frac{\text{sum}(\text{container\_memory\_working\_set\_bytes}\{\text{container}=\text{"istio-proxy"}\})}{\text{vCPU}}$   |

| 分类 | 参数 | 参数介绍 | 计算方式  |
|----|----|------|---|
|    |    |      | <p>“istio-proxy”的 CPU 使用秒数的</p> <p>速率总和  </p> <p><code>sum(rate(container_cpu_usage_seconds_total{container="istio-proxy"}[1m]))    </code></p> <p>  磁盘   展示边车</p> <p>代理 (Proxy) 资源使用的磁盘</p> <p>统计信息,展示容器“istio-proxy”</p> <p>的文件系统使用字节的总和  </p> <p><code>sum(container_fs_usage_bytes{container="istio-proxy"})  </code></p> <p>  Istiod 资源使用率   内存</p> <p>  展示 Istiod 服务的内存使用</p> <p>情况,提供了一个全面的视图:</p> <p>- 总计:Istiod 服务在 Kubernetes</p> <p>中的总内存使用量 - 容器内</p> <p>存: Istiod 服务在 Kubernetes 中</p> <p>每个容器的内存使用量 包括虚</p> <p>拟内存、常驻内存、堆内存和栈</p> <p>内存等不同类型的内存使用情</p> <p>况。   总计 (Total (k8s)) :</p> <p><code>"sum(container_memory_working_set_bytes{container=~\"discovery\" istio-proxy\"},</code></p> <p><code>pod=~\"istiod-.*\")\" 容器内存</code></p> |

分类      参数      参数介绍

计算方式

(({ container }) (k8s)) :

```
"container_memory_working_set_bytes{container=~\"discovery\|istio-proxy\",
pod=~\"istiod-.*\"}" | |
```

| vCPU                      | 展示

Istiod 服务的虚拟 CPU (vCPU)

使用情况, 提供了一个全面的视

图: - 总计: 显示 Istiod 服务

在 Kubernetes 中的总 CPU 使用

率 - 容器 CPU 使用率: 显示

Istiod 服务在 Kubernetes 中每个

容器的 CPU 使用率 - Pilot: 显示

Istiod 的 pilot 组件的 CPU 使用情

况 | 总计 (Total (k8s)) :

```
"sum(rate(container_cpu_usage_seconds_total{container=~\"disc
overy\|istio-proxy\",
```

```
pod=~\"istiod-.*\"}[1m]))" 容
```

器 CPU 使用率 ({ container })

(k8s)) :

```
"sum(rate(container_cpu_usage_seconds_total{container=~\"disc
overy\|istio-proxy\",
```

```
pod=~\"istiod-.*\"}[1m])) by
```

```
(container)" pilot:
```

```
"irate(process_cpu_seconds_total{app=~\"istiod\"}[1m]))" | |
```

| 分类 | 参数         | 参数介绍                         | 计算方式   |
|----|------------|------------------------------|--|
|    |            |                              | 磁盘   展示每个集群中 Istio 组件的磁盘使用情况，特别是与 discovery 和 istio-proxy 容器相关的文件系统使用情况。   |
|    | Goroutines | 展示每个集群中 Istio 组件的 Go 协程数量的趋势 | $\text{sum}(\text{process\_open\_fds}\{\text{mesh\_id}=\text{"\$mesh"}, \text{app}=\text{"istiod"}\}) \text{ by } (\text{mesh\_cluster})$ $\text{container\_fs\_usage\_bytes}\{\text{container}=\sim\text{"discovery istio-proxy"}, \text{pod}=\sim\text{"istiod-.*"}\}$ $\text{sum}(\text{go\_goroutines}\{\text{mesh\_id}=\text{"\$mesh"}, \text{app}=\text{"istiod"}\}) \text{ by } (\text{mesh\_cluster})$ |

## 服务监控

### 服务监控

#### 服务监控

| 分类 | 参数     | 参数介绍   | 计算方式  |
|----|--------|--|---|
| 常规 | 客户端请求量 | 展示当前服务的客户端的每 5 分钟的操作量, 如果结果为空值是会展示为 N/A; 阈值数值超过 80%时会展示为红色 | $\text{round}(\text{sum}(\text{irate}(\text{istio\_requests\_total}\{\text{reporter}=\text{"\$reporter"}, \text{destination\_mesh\_id}=\text{"\$mesh"}, \text{destination\_service}=\text{"\$service"}\}[5\text{m}]]), 0.001)    $   客户端成功率 (非 5xx 响应)   展示当前服务的客户端的每 5 分钟的成功请求率, 并且提供了一种可视化方式来快速识别 |

分类

参数

参数介绍

计算方式

潜在的问题或趋势。 |

```
sum(irate(istio_requests_total{re
porter=~"$reporter",destination_
mesh_id="$mesh",destination_s
ervice=~"$service",response_cod
e!~"5.*"}[5m])) /
sum(irate(istio_requests_total{re
porter="$reporter",destination_mesh_id="$m
esh",destination_service="$service"}[5m]
))) | | | | 客户
```

端请求时延

| 展示当前服务的客户端的请

求用时情况，定义了三个目标，

用于计算 P50、P90 和 P99 的

持续时间。表达式分别计算了

50%、90% 和 99% 的持续时间

百分位数。 | 示例：

```
(histogram_quantile(0.50,
sum(irate(istio_request_duration
_milliseconds_bucket{reporter=~
\"$reporter\",destination_mesh_i
d=\"$mesh\",destination_service=
~\"$service\"}[1m])) by (le)) /
1000) or
histogram_quantile(0.50,
sum(irate(istio_request_duration
_seconds_bucket{reporter=\"$report
er\",destination_mesh_id=\"$mesh\",destination_
service=\"$service\"}[1m])) by (le)) |
```

| | TCP 接收

字节数



| 分类                | 参数 | 参数介绍   | 计算方式   |
|-------------------|----|--|--|
|                   |    |  | <p>  展示当前服务在 1 分钟内</p> <p>TCP 接收字节的即时速率之和,</p> <p>如果匹配到 "null", 结果文本为</p> <p>"N/A", 单位 Bps  </p> <p>"sum(irate(istio_tcp_received_bytes_total{reporter=~\"\$reporter\",destination_mesh_id=\"\$mesh\",destination_service=~\"\$service\"}[1m]))"</p> <p>round(sum(irate(istio_requests_total{reporter="destination",destination_mesh_id="\$mesh",destination_service=~"\$service"}[5m])), 0.001)</p>   |
| 服务的请求量            |    | 展示当前服务的请求量, 按照时间展示趋势图, 如果匹配到 "null", 结果文本为 "N/A", 单位 Ops                         |  |
| 服务器成功率 (非 5xx 响应) |    | 展示当前服务的非 5xx 响应的成功率, 阈值设置表明, 95%以下的成功率标记为红色, 99%以下的成功率标记为橙色, 100%为绿色 (百分比取值 2 位) | <p>sum(irate(istio_requests_total{reporter="destination",destination_mesh_id="\$mesh",destination_service=~"\$service",response_code!~"5."}[5m])) / sum(irate(istio_requests_total{reporter="destination",destination_mesh_id="\$mesh",destination_service=~"\$service"}[5m]))  </p> <p>  服务端请求延时</p> <p>  展示当前服务的服务器请求的延时, 通过计算不同分位数的持续时间来提供对服务性能的深入了解。三个目标表达式分别计算了中位数、90%、和 99% 的请求的延时, 以提供从中位数到</p> |

分类      参数      参数介绍

计算方式

高端的性能概览 | - histogram\_quantile -

istio\_request\_duration\_milliseconds\_bucket -

istio\_request\_duration\_seconds\_bucket | |

| TCP 发送字节数

| 展示当前服务在 1 分钟内 TCP 发送字节的

即时速率之和, 如果匹配到 "null", 结果文本为

"N/A", 单位 Bps |

|| 客户端工作负载 | 基于源和响应码的传入

请求 | 展示按源工作负载和

响应代码分类的传入请求, 能够可视化地展示

各种工作负载之间的交互情况, 分别对具有和

不具有互相认证 TLS 的连接进行计算, 以提供对

连接安全性的洞察。可以清晰地了解请求如何

在不同的源和目的地之间分布。 |

|| | 基于源的传入成功率

(非 5xx 响应) | 展示按源工作负载和

命名空间分类的传入成功率, 其中成功率是非

5xx 响应的百分比, 分别对具有和不具有互相认

证 TLS 的连接进行计算, 以提供对连接成功率的

视图。 |

```
sum(irate(istio_requests_total{reporter=~"$reporter",destination_mesh_id="$mesh",connection_security_policy="mutual_tls",destination_service=~"$service",response_code!="5xx"}))
```

分类

参数

参数介绍

计算方式

5.", source\_workload=~"\$srcwl",  
source\_workload\_namespace=~"  
\$srcns"}[5m])) by  
(source\_workload,  
source\_workload\_namespace) /  
sum(irate(istio\_requests\_total{re  
porter=~"\$reporter",destination\_  
mesh\_id="\$mesh",  
connection\_security\_policy="mu  
tual\_tls",  
destination\_service=~"\$service",  
source\_workload=~"\$srcwl",  
source\_workload\_namespace=~"  
\$srcns"}[5m])) by  
(source\_workload,  
source\_workload\_namespace) |  
  
| 基于源的

传入请求时延

| 展示按源工作负载和命名空

间的请求耗时，分别计算了

P50、P90、P95 和 P99 等分位



数不同的请求持续时间。注意，



将分位值除以 1000 转为秒


作为单位。 | 示例：

(histogram\_quantile(0.50,  
sum(irate(istio\_request\_duration\_  
\_milliseconds\_bucket{reporter=~  
"\$reporter",destination\_mesh\_id  
=~"\$mesh",  
connection\_security\_policy="mu  
tual\_tls",  
destination\_service=~"\$service",  
source\_workload=~"\$srcwl",  
source\_workload\_namespace=~"

| 分类                | 参数 | 参数介绍   | 计算方式   |
|-------------------|----|--|--|
|                   |    |  | $\frac{\sum(\text{irate}(\text{istio\_request\_duration\_seconds\_bucket}\{\text{reporter}=\sim\text{"\$reporter"},\text{destination\_mesh\_id}=\sim\text{"\$mesh"},\text{connection\_security\_policy}=\sim\text{"mutual\_tls"},\text{destination\_service}=\sim\text{"\$service"},\text{source\_workload}=\sim\text{"\$srcwl"},\text{source\_workload\_namespace}=\sim\text{"\$srcns"}\}[1m]))}{(\text{source\_workload},\text{source\_workload\_namespace},\text{le})} \times 1000$ |
| 基于源的传入请求大小        |    | 展示按源工作负载传入的请求大小，分别计算了 P50、P90、P95 和 P99 等分位数不同的请求大小。   |  |
| 基于源的响应大小          |    | 展示按源工作负载 (source workload) 展示了在 P50、P90、P95、P99 (🔒 mTLS): 表示在启用了相互 TLS (mTLS) 的情况下的响应大小的百分位数 |  |
| 接收来自传入 TCP 连接的字节数 |    | 展示当前服务通过 TCP 连接收到的字节数，并且展示在 mutual TLS 连接 (标记为   |  |

| 分类 | 参数      | 参数介绍   | 计算方式 |
|----|---------|--|------|
|    |         |  mutual TLS) 和非 mutual TLS 连接 |      |
|    |         | 下从 TCP 连接收到的字节数的   |      |
|    |         | 情况   |      |
|    |         | 发送到传入  |      |
|    | 字节数     | TCP 连接的  |      |
|    |         | 送的字节数, 并且展示在   |      |
|    |         | mutual TLS 连接 (标记为   |      |
|    |         |  mutual TLS) 和非 mutual TLS 连接 |      |
|    |         | 下从 TCP 连接收到的字节数的   |      |
|    |         | 情况   |      |
|    |         | 服务工作   |      |
|    |         | 基于目标负  |      |
| 负载 | 码和响应码   | 码分类的传入请求, 能够可视化  |      |
|    |         | 的传入请求  |      |
|    |         | 地展示各种工作负载之间的交  |      |
|    |         | 互情况, 分别对具有和不具有互  |      |
|    |         | 相认证 TLS 的连接进行计算, 以   |      |
|    |         | 提供对连接安全性的洞察。可以   |      |
|    |         | 清晰地了解请求如何在不同的  |      |
|    |         | 源和目的地之间分布。   |      |
|    | 功率 (非   | 展示按目的工作负载和命名空  |      |
|    |         | 间分类的传入成功率, 其中成功  |      |
|    |         | 率是非 5xx 响应的百分比, 分别   |      |
|    |         | 对具有和不具有互相认证 TLS 的  |      |
|    | 5xx 响应) |  |      |
|    |         |  |      |
|    |         |  |      |
|    |         |  |      |

| 分类 | 参数                | 参数介绍   | 计算方式 |
|----|-------------------|--|------|
|    |                   | 连接进行计算, 以提供对连接成<br>功率的视图。  |      |
|    | 基于服务负载的传入请求时延     | 展示按目的工作负载和命名空间的请求耗时, 分别计算了 P50、P90、P95 和 P99 等分位数不同的请求持续时间。注意, 将分位值除以 1000 转为秒作为单位。  |      |
|    | 基于服务负载的传入请求大小     | 展示按目的工作负载传入的请求大小, 分别计算了 P50、P90、P95 和 P99 等分位数不同的请求大小。   |      |
|    | 基于服务负载的响应大小       | 展示按目的工作负载 (source workload) 展示了在 P50、P90、P95、P99 (  mTLS): 表示在启用了相互 TLS (mTLS) 的情况下的响应大小的百分位数 |      |
|    | 接收来自传入 TCP 连接的字节数 | 展示当前服务通过 TCP 连接收到的字节数, 并且展示在 mutual TLS 连接 (标记为  mTLS) 和非 mutual TLS 连接                       |      |

| 分类 | 参数               | 参数介绍  | 计算方式 |
|----|------------------|---|------|
|    |                  | 下从 TCP 连接收到的字节数的情况  |      |
|    | 发送到传入 TCP 连接的字节数 | 展示当前服务通过 TCP 连接发送的字节数，并且展示在 mutual TLS 连接（标记为  mTLS）和非 mutual TLS 连接下从 TCP 连接收到的字节数的情况 |      |

工作负载监控

工作负载监控

工作负载监控

| 分类 | 参数              | 参数介绍  | 计算方式  |
|----|-----------------|---|---|
| 常规 | 传入请求量           | 展示当前工作负载的传入请求量，单位为 Ops，如果接收到的数据为空（null），则会显示“N/A”                         | 计算过去 5 分钟内的传入请求总数，其中包括特定的目的地工作负载、命名空间和集群。                         |
|    | 传入成功率（非 5xx 响应） | 展示当前工作负载的每 5 分钟的成功请求率（不含 5xx 的请求），并且提供了一种可视化方式来快速识别潜在的问题或趋势。如果成功率低于 95%，将 | 使用了两个分母和分子的查询来计算非 5xx 响应的百分比。分子计算与特定服务相关的非 5xx 响应，分母计算与该服务相关的所有请求 |

| 分类        | 参数   | 参数介绍  | 计算方式  |
|-----------|------|---|---|
|           |      | 显示为红色, 如果低于 99%,<br><br>则显示为橙色  |   |
|           | 请求时延 | 展示当前工作负载的请求的时延, 分别显示 P50、P90 和 P99 的请求持续时间, 可以用于观察系统的性能, 并快速识别潜在的瓶颈或延迟问题。                                   | $\frac{\text{histogram\_quantile}(0.50, \text{sum}(\text{irate}(\text{istio\_request\_duration\_milliseconds\_bucket}\{\text{reporter}=\sim\text{\$reporter}, \text{destination\_mesh\_id}=\sim\text{\$mesh}, \text{destination\_workload}=\sim\text{\$workload}, \text{destination\_cluster}=\sim\text{\$dstcluster}\}, \text{destination\_workload\_namespace}=\sim\text{\$namespace}\})[1m]))}{1000} \text{ or } \frac{\text{histogram\_quantile}(0.50, \text{sum}(\text{irate}(\text{istio\_request\_duration\_seconds\_bucket}\{\text{reporter}=\sim\text{\$reporter}, \text{destination\_mesh\_id}=\sim\text{\$mesh}, \text{destination\_workload}=\sim\text{\$workload}, \text{destination\_cluster}=\sim\text{\$dstcluster}\}, \text{destination\_workload\_namespace}=\sim\text{\$namespace}\})[1m]))}{1000} \text{ by (le)}}$ |
| TCP 服务端流量 |      | 展示当前工作负载的 TCP 服务器流量, 注重统计作为目标工作负载、命名空间和集群, 即作为 TCP 服务器, 有助于了解网络负载和可能的瓶颈; 并以每秒字节 (Bps) 为单位显示 TCP 发送和接收的字节总数。 | $\frac{\text{destination\_workload\_namespace}=\sim\text{\$namespace}, \text{destination\_workload}=\sim\text{\$workload}, \text{destination\_cluster}=\sim\text{\$dstcluster}}{\text{TCP 客户端流量}}$ <p>  展示当前工作负载的 TCP 客户端的流量, 注重统计作为源工作负载、命名空间和集群, 即作为 TCP 服务器, 有助于了解网</p>  |



| 分类 | 参数 | 参数介绍 | 计算方式   |
|----|----|------|--|
|    |    |      | <p>络负载和可能的瓶颈; 并以每秒</p> <p>字节 (Bps) 为单位显示 TCP 发</p> <p>送和接收的字节总数。  </p> <p>source_workload_namespace=~"<br/>\$namespace",<br/>source_workload=~"\$workload"</p> <p>   入站工作负载   基于源和</p> <p>响应码的传入请求</p> <p>  展示当前工作负载中, 根据源</p> <p>工作负载和响应代码分类的传</p> <p>入请求, 计算使用和不使用双向</p> <p>TLS 连接的传入请求速率, 并按</p> <p>源工作负载、源命名空间和响应</p> <p>代码分类   示例:</p> <pre>round(sum(irate(istio_requests_t otal{connection_security_policy! ="mutual_tls", destination_workload_namespac e=~"\$namespace", destination_workload=~"\$worklo ad", destination_cluster=~"\$dstcluste r", reporter=~"\$reporter",destinatio n_mesh_id="\$mesh", source_workload=~"\$srcwl", source_workload_namespace=~" \$srcns"}[5m])) by (source_workload, source_workload_namespace, response_code), 0.001)</pre> |

| 分类                   | 参数   | 参数介绍 | 计算方式   |
|----------------------|--|------|--|
| 基于源的传入成功率 (非 5xx 响应) | 展示传入成功请求的比率 (非 5xx 响应), 分别按使用和不使用双向 TLS 的连接进行分类, 并进一步按源工作负载和源命名空间分组。   |      | $\frac{\text{sum}(\text{irate}(\text{istio\_requests\_total}\{\text{reporter}=\sim\text{"\$reporter"},\text{destination\_mesh\_id}=\text{"\$mesh"},\text{connection\_security\_policy}=\text{"mutual\_tls"},\text{destination\_workload\_namespace}=\sim\text{"\$namespace"},\text{destination\_workload}=\sim\text{"\$workload"},\text{destination\_cluster}=\sim\text{"\$dstcluster"},\text{response\_code}!\sim\text{"5.*"},\text{source\_workload}=\sim\text{"\$srcwl"},\text{source\_workload\_namespace}=\sim\text{"\$srcns"}\}[5m]))\text{ by }(\text{source\_workload},\text{source\_workload\_namespace})}{\text{sum}(\text{irate}(\text{istio\_requests\_total}\{\text{reporter}=\sim\text{"\$reporter"},\text{destination\_mesh\_id}=\text{"\$mesh"},\text{connection\_security\_policy}=\text{"mutual\_tls"},\text{destination\_workload\_namespace}=\sim\text{"\$namespace"},\text{destination\_workload}=\sim\text{"\$workload"},\text{destination\_cluster}=\sim\text{"\$dstcluster"},\text{source\_workload}=\sim\text{"\$srcwl"},\text{source\_workload\_namespace}=\sim\text{"\$srcns"}\}[5m]))\text{ by }(\text{source\_workload},\text{source\_workload\_namespace})}$ |
| 基于源的传入请求时延           | 展示使用双向 TLS (标志为  mTLS) 和未使用双向 TLS 的源工作负载的请求持续时间。通过各种百分位数 P50、P90 和 P99, 可以更好地理解不同工作 |      |  |

| 分类     | 参数                | 参数介绍  | 计算方式  |
|--------|-------------------|---|---|
|        |                   | 负载下的性能表现  |   |
|        | 基于源的传入请求大小        | 展示来自不同来源的请求大小的图表, 增加了用于计算不同分位数 P50、P90、P95 和 P99 的请求大小。   |   |
|        | 基于源的响应大小          | 展示响应大小的线型图表, 可用于监控来自不同来源的响应大小, 使用了 histogram_quantile 函数计算不同分位数的响应字节大小 (例如 P50、P90、P95 和 P99) 。 |   |
|        | 接收来自传入 TCP 连接的字节数 | 展示与传入 TCP 连接相关的字节接收情况, 分别表示与 mTLS 和非 mTLS 的连接有关的数据。   |   |
|        | 发送到传入 TCP 连接的字节数  | 展示对发送到传入 TCP 连接的字节的可视化表示。通过与之前的图表结合, 可以提供完整的对 TCP 连接的监控视图, 同时比较启用和未启用 mTLS 的连接                  |   |
| 出站工作负载 | 基于目标和响应码的传出请求     | 展示按目的地和响应码的传出请求, 基于   | $\text{round}(\text{sum}(\text{irate}(\text{istio\_requests\_total}\{\text{destination\_principal}=\sim\text{"spiffe.*"}, \text{source\_workload\_namespace}=\sim\text{"$ |

| 分类 | 参数                              | 参数介绍   | 计算方式   |
|----|---------------------------------|--|--|
|    | 出请求                             | istio_requests_total 指标, , 分别表示与 mTLS 和非 mTLS 的请求量数据。                                  | \$namespace",<br>source_workload=~"\$workload",<br>reporter="source",<br>destination_mesh_id="\$mesh",<br>destination_service=~"\$dstsvc"}[5m])) by (destination_service, response_code), 0.001) |
|    | 基于目标的<br>传出成功率<br>(非 5xx<br>响应) | 展示传出成功请求的比率 (非 5xx 响应) , 分别按使用和不使用双向 TLS 的连接进行分类, 并进一步按源工作负载和源命名空间分组。                  |  |
|    | 基于目标的<br>传出请求时<br>延             | 展示对应请求目标传出的请求时长的百分位, 支持 50、P90、P95 和 P99。同时支持 mTLS 和非 mTLS 的安全策略, 单位: 秒/毫秒。            |  |
|    | 基于目标的<br>传出请求大<br>小             | 展示在使用 mTLS 和不使用时传出的请求大小, 支持支持 50、P90、P95 和 P99。表达式中使用了与 Istio、工作负载、命名空间和目标服务相关的标签进行过滤。 |  |
|    | 基于目标的                           | 展示传出不同目的地的响应大  |  |

| 分类 | 参数                | 参数介绍   | 计算方式  |
|----|-------------------|--|---|
|    | 响应大小              | 小, 分别计算了 P50、P90、P95 和 P99 的值, 有的与 mutual_tls 的连接安全策略相关联。                  |   |
|    | 传出 TCP 连接发送的字节数   | 展示通过 TCP 连接发出的字节。通过对比 mTLS (双向 TLS) 和非 mTLS 连接, 深入了解不同安全策略下的字节发送情况。        |   |
|    | 接收来自传出 TCP 连接的字节数 | 展示通过目的服务发出的外部 TCP 连接接收的字节量, 分别统计通过 mutual TLS 和不通过 mutual TLS 进行保护的连接的字节总量 | 示例:<br><br>round(sum(irate(istio_tcp_sent_bytes_total{reporter="source", destination_mesh_id="\$mesh", connection_security_policy!="mutual_tls", source_workload_namespace=~"\$namespace", source_workload=~"\$workload", destination_service=~"\$dstsvc"}[1m])) by (destination_service), 0.001) |

## 监控图表阅读指南

服务网格基于 Istio 开源的 Grafana Dashboard, 为了每一个服务网格实例自动生成了流量监控看板。

主要根据以下几个维度进行呈现:

- 全局监控: 主要展示网格实例的策略数量、服务与工作负载等全部的资源情况。

- 性能监控：重点展示网格实例的系统资源的性能情况、以及 Sidecar 资源消耗资源等。
- 服务监控：以服务作为主视角，展示服务的流量和工作负载的状态。
- 工作负载监控：以工作负载作为主视角，展示工作负载的资源消耗、业务负载等数据。

## 图表阅读指南

以性能监控为例，图表中提供了各个指标的监控数据，展示形式主要以数值、图表、表单等来呈现。

### 性能监控

#### 性能监控

图表上方的搜索条提供了监控看板的一些辅助检索字段，基于看板的不同，可用的检索字段也略有不同。

- 时间范围（通用）
- 刷新按钮（通用）
- 命名空间
- 服务名称
- 工作负载
- ...

## 全屏查看指标

点击对应指标图表的标题，可以展开更多的指标操作，其中第一个就是方法查看，可通过键盘 **ESC** 返回。

view menu  
view menu  
view big  
view big

## 导出指标数据

如果希望查看图表对应的原始数据，可以点击指标图表的标题，然后选择 `Inspect > Data`，就可以用数据表的形式查看对应的数据。

`inspect data`

此时，我们可以进行数据的导出，支持 CSV 和 Excel 两种格式

`data`

## 查看指标的计算表达式

如果想要详细的了解指标的计算公式，可以查看指标图表的 Json 结构体，在其中会给出指标的具体计算方式，操作路径如下：

`json`

## 集群纳管

服务网格的集群纳管页面为用户提供了集群接入、移除功能。

**集群接入是一个集群加入网格管理边界的第一步。**

完成接入的集群将出现在边车管理的集群列表中，然后用户可以对该集群的命名空间、工作负载启用边车注入，实现对集群的聚合治理管控。

该功能仅在 **托管网格** 有效，在 **专有网格** 和 **外接网格** 模式下未开放该功能。具体参见[服务网格分类](#)。托管网格采用单控制面多集群的方式，实现对多集群统一获取服务注册信息、下发治理策略，各托管集群的控制面组件仅负责接受转发来自唯一管理集群控制面的策略，不执行任何本地创建的策略。

# 添加集群

当用户创建一个托管网格后，还未接入任何托管的集群，此时网格处于 **未就绪** 状态，用户需要添加一个或多个集群。

具体操作步骤如下：

1. 在左侧导航栏点击 **集群纳管**，点击 **添加集群** 按钮。或者在网格列表中，点击最右侧的 ... 按钮，在弹出菜单中选择 **添加集群**。

添加集群按钮

添加集群按钮

2. 在弹出的集群列表中，勾选待加入的集群后，点击 **确定** 按钮，完成集群的添加操作。

添加集群页面

添加集群页面

!!! note

1. 仅 **\*\*托管网格\*\*** 可以添加集群，专有网格无法添加集群。
2. 仅可添加状态为 **\_\_运行中\_\_** 的集群至网格，其他状态的集群无法选中，也无法添加。
3. 集群添加的过程会持续几秒，请关注网格列表中的 **\_\_集群统计\_\_** 信息，了解正常集群的数量变化。

# 移除集群

如果想要从托管网格中移除一个集群，可以按照以下步骤移除集群。

1. 在左侧导航栏点击 **集群纳管**，在集群列表右侧，点击 **移除** 按钮。或者在网格列表中，点击最右侧的 ... 按钮，在弹出菜单中选择 **移除**。

移除集群



## 移除集群

2. 在弹出窗口中，确认信息无误后，点击 **确定**。

## 移除集群

## 移除集群

移除集群，需要完成一些前置操作，例如：

- 卸载已注入的边车。
- 清除集群相关的网格网关。
- 其他前置操作，请按屏幕提示操作。

!!! warning

移除集群，将无法通过网格集中管理集群，另外集群日志等相关信息可能会丢失，请谨慎操作。

# 管理网格网关

## 网关介绍

网格网关模块，提供了网格网关实例的生命周期管理，包含创建、更新、删除等管理能力，用户可以在该页面管理当前网格内的全部托管的网关实例。

网格网关分为 Ingress（入站）和 Egress（出站）两类。

- Ingress 网关用于定义服务网格内应用的流量入口，所有进入服务网格内应用的流量应  
对经过 Ingress 网关。
- Egress 网关用于定义了网格内应用的流量出口，可以让外部服务的流量统一经过  
Egress 网关，从而实现更精确的流量控制。

网关实例运行的也是 Envoy，但与边车不同，网格以具体的实例单独运行。

# 创建网格网关

创建网格网关的操作步骤如下。

1. 在左侧导航栏中点击 **网格网关**，进入网关列表，点击右上角的 **创建** 按钮。

创建

创建

2. 在 **创建网格网关** 窗口中，按提示配置各项参数后点击 **确定**。

创建 创建

3. 返回网格网关列表，屏幕右上角提示创建成功。刚创建的网格网关状态为 **创建中**。

创建

创建

4. 几秒后刷新页面，状态将变为 **运行中**，表示网关成功配置。点击列表右侧的 **⋮**，可以执行编辑和删除操作。

创建

创建

!!! info

如有任何异常，请查看异常提示尝试解决问题。

更多参阅[视频教程](../../videos/mspider.md)。

# 删除网格网关

如果想要删除一个或多个网关，可以参照本文来操作。

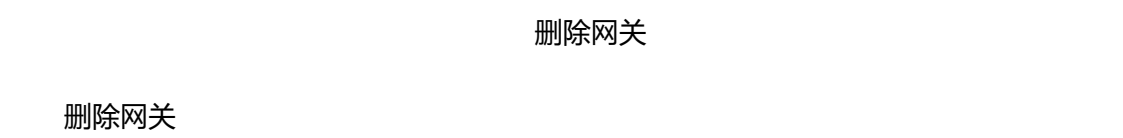
## 删除一个网关

!!! caution

删除网关之前，需要检查相应的 Gateway 和 VirtualService 资源，否则会有无效配置，导致流量异常。

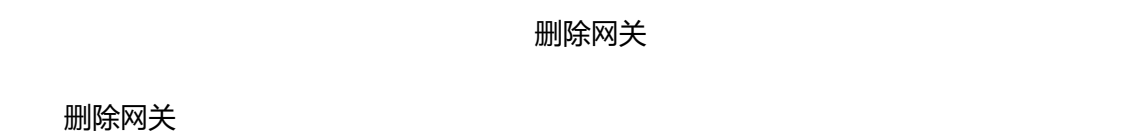
以下操作步骤是推荐的操作，可以防止误删除。

1. 进入某个网格后，在左侧导航栏点击 **网格网关**，在列表右侧点击 **⋮** 按钮，在弹出菜单中选择 **删除**。



你也可以勾选某个网关前的复选框后，点击 **删除** 按钮。

2. 在弹出窗口中，确认信息无误后，点击 **确定**，该网关将被删除。



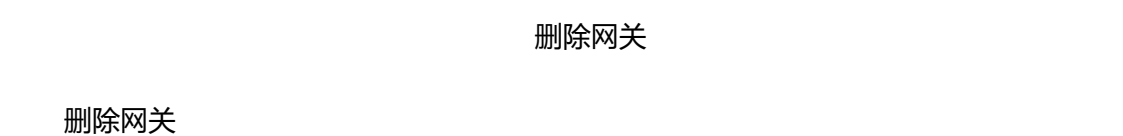
!!! warning

删除网关后，该网关相关的信息将会丢失，请谨慎操作。

## 批量删除

如果确实需要一次性删除多个网关，可以采用此项操作。

1. 在网关列表中勾选多个网关后，点击右上角的 **删除** 按钮。



2. 在弹出窗口中，确认信息无误后，点击 **确定**，选中的网关将全部被删除。

!!! warning

如非必要，请勿使用批量删除功能，删除的网关无法恢复，请谨慎操作。

## 外接网格的网关管理

外接网格的部署和生命周期管理是由用户自行负责的。网格网关的安装方式取决于具体的网

格部署方式，您可以按照以下相应方式来安装网格网关。

## 使用 IstioOperator 方式

这种方法需要依赖已经安装和部署完成的 Istiod。您可以按照以下步骤执行操作：

首先，创建对应的 **ingress** 或 **egress** 配置文件：

```
yaml title="ingress.yaml" apiVersion: install.istio.io/v1alpha1 kind: IstioOperator metadata:
name: ingress spec:   profile: empty # (1)!   components:     ingressGateways:     -
name: istio-ingressgateway     namespace: istio-ingress     enabled: true     label:
istio: ingressgateway # (2)!   values:     gateways:     istio-ingressgateway:
injectionTemplate: gateway # (3)!
```

1. 不安装 CRD 或控制平面
2. 为网关设置唯一标签。这是确保 Gateway 可以选择此工作负载所必需的。
3. 启用网关注入

然后，使用标准的 **istioctl** 命令进行安装：

```
kubectl create namespace istio-ingress
istioctl install -f ingress.yaml
```

## 使用 Helm 方式

### 标准 Kubernetes 集群

对于通过 Helm 部署和管理的网格，建议仍然使用 Helm 安装和维护网格网关。

在使用 Helm 安装网格网关之前，您可以查看支持的配置值以确认参数配置：

```
helm show values istio/gateway
```

然后，使用 **helm install** 命令进行安装：

```
kubectl create namespace istio-ingress
helm install istio-ingressgateway istio/gateway -n istio-ingress
```

## OpenShift 集群

对于 OpenShift 集群,其参数配置与标准 Kubernetes 集群不同,Istio 官方提供了推荐配置:

```
helm install istio-ingressgateway istio/gateway -n istio-ingress -f manifests/charts/gateway/openshift-values.yaml
```

## 使用 YAML 方式

使用 YAML 单独安装网格网关,这种方式不会影响应用的部署,但需要用户自行管理网关的生命周期。

## 创建网关工作负载

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: istio-ingressgateway
  namespace: istio-ingress
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  template:
    metadata:
      annotations:
        inject.istio.io/templates: gateway # (1)!
      labels:
        istio: ingressgateway # (2)!
        sidecar.istio.io/inject: "true" # (3)!
    spec:
      securityContext: # (4)!
        sysctls:
          - name: net.ipv4.ip_unprivileged_port_start
            value: "0"
      containers:
        - name: istio-proxy
          image: auto # (5)!
          securityContext: # (6)!
            capabilities:
```

```
drop:
- ALL
runAsUser: 1337
runAsGroup: 1337
```

1. 选择网关注入模板（而不是默认的 Sidecar 模板）
2. 为网关设置唯一标签。这是确保 Gateway 可以选择此工作负载所必需的
3. 启用网关注入。如果后续连接到修订版的控制平面，请替换为 `istio.io/rev: revision-name`
4. 允许绑定到所有端口（例如 80 和 443）
5. 每次 Pod 启动时，该镜像都会自动更新。
6. 放弃所有 `privilege` 特权，允许以非 `root` 身份运行

## 添加相应权限

```
# 设置 Role 以允许读取 TLS 凭据
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: istio-ingressgateway-sds
  namespace: istio-ingress
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
- --
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: istio-ingressgateway-sds
  namespace: istio-ingress
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: istio-ingressgateway-sds
subjects:
- kind: ServiceAccount
  name: default
```

## 创建网关服务

```
apiVersion: v1
kind: Service
metadata:
  name: istio-ingressgateway
  namespace: istio-ingress
spec:
  type: LoadBalancer
  selector:
    istio: ingressgateway
  ports:
    - port: 80
      name: http
    - port: 443
      name: https
```

自行管理网关工作负载将大大增加维护工作的复杂性,包括管理网关的生命周期和配置管理

等。因此,建议用户使用 Helm 或 IstioOperator 的方式进行安装和管理。

## 更多安装方式

Istio 提供了丰富的部署方式,上述仅列举了几种主流的部署方式。如果您使用的方式与上

述方式不同,请参阅 Istio 文档以获取更多详细信息:

- Istio Operator 安装: <https://istio.io/docs/setup/install/operator/>
- Gateway 资源: <https://istio.io/docs/reference/config/networking/gateway/>

## 多云网络互联

多云网络互联是在多集群模式下,多集群之间网络互不相通(Pod 无法直接建立通讯)的情

况下提供的一套解决方案,可以快速地将多集群的网络连通,实现跨集群的 Pod 互相访问。

## 使用场景

如果您的环境满足以下条件，可尝试请使用多云网络互联功能：

- 1.使用的是托管网格，至少包含了 2 个或以上的集群
- 2.托管网格的工作集群之间的网络存在无法直接连通的情况（工作集群 Pod CIDR 规划冲突、集群网络不在同一数据中心等）

!!! note

多云网络互联仅适用于托管网格。

## 名词解释与使用说明

- 网络分组：
  - 用于辨别网格内集群的网络拓扑关系。
  - 如果某些集群的网络能够直接互通（跨集群的 Pod IP 没有冲突且能够直接连通），那么这些集群应该被放在一个网络分组里，否则将属于不同的网络分组。
  - 网络分组和集群的网络 CNI 实现类型没有明确的关联关系。
- 东西网关：
  - 用于不同网络分组间通信互联，一个网络分组内可以创建多个。
  - 用户可以在网络分组下任意集群创建东西网关，当移动集群至其他网络分组时，东西网关也会随之迁移。
  - 东西网关所用负载均衡 IP 通常有集群所在的云平台自动分配，也可以由用户配置设定。
- 基本设置：创建用于东西网关的 **网关规则** 配置，该配置将开放 15443 端口用于网络分组间的通信，用户可以在 **网关规则** 列表中查看 CRD 文件详细内容，但不建议



修改，可能会导致网络通信失败。

- 网络分组列表：

- 展示当前创建的网络分组及分组下集群、东西网关信息；
- 在该列表下用户可以对网络分组、分组下集群及东西网关执行增删操作，并可以在网络分组间迁移集群。

- 互联列表：

- 包含至少一个东西网关的网络分组可以加入互联列表，加入互联的网络分组可以和其他网络分组通信。
- 已加入互联的网络分组可以修改负载均衡 IP，但如需修改其他配置（例如增删集群、东西网关），需要先移除 **网络分组互联** 列表。
- 加入互联和移除互联会引起网络控制面的重启，建议谨慎操作。

## 操作流程

建议操作流程如下图所示

操作流程

操作流程

## 操作步骤

1. 进入某个托管网络，点击启用开关，在成功启用后将自动创建用于东西网关的 **网关规则**。点击 **创建网络分组** 按钮，至少网络分组添加至少一个工作集群。

创建

创建

2. 为网络分组填入名称，并添加至少一个集群。

### 创建对话框

#### 创建对话框

##### !!! note

1. 网络分组名称创建后无法修改。
  2. 注意同一网络分组内的集群需确保处于同一网络类型并可以互通，否则可能造成互联失败。
3. 网络分组内添加或移出集群。

创建网络分组后，即可为分组内添加更多的集群，**同一分组的集群需确保处于同一网络类型并可以互通，否则可能造成互联失败。**

### 添加集群

#### 添加集群

添加完成后即可在网络分组下看到多个集群。


### 集群列表

#### 集群列表

##### !!! note

1. 网络分组中至少要保留一个集群，如需移除所有集群，请删除网络分组
  2. 网络分组内的成员集群发生变化是一件非常重要的事情，请慎重操作
  3. 变更后会在规则变更后再生效前会存在网络延迟，建议在业务低峰期进行操作
  4. 跨网络分组内的成员集群发生变化后，会涉及到工作负载 \_\_Pod\_\_ 的边车需要重启后流量才会生效
4. 创建/删除东西网关。

东西网关用于网络分组间通信，可以在分组内的任一集群上创建一个或多个网关。点击

一个集群右侧的 ，选择 **编辑东西网关**：

### 编辑东西网关

#### 编辑东西网关

创建配置项如下：

- **负载均衡注释** 为可选设置，部分云平台会以注释方式提供负载均衡 IP 分配，请参考云平台提供的技术文档；
- **东西网关副本数** 默认为 1，如果需提高网关可用性，可创建多个副本；点击 **确定** ；

### 网关参数

### 网关参数

选择一个东西网关，点击操作中的 **删除东西网关** ，可以删除该网关，如图所示：

### 删除网关

### 删除网关

!!! note

每个网络分组至少需要创建一个东西网关，否则无法加入互联。

## 5. 网络分组互联

1. 如果要将一个网络分组加入互联，点击 **加入互联** 按钮 。

### 加入互联

### 加入互联

2. 勾选一个网络分组，点击 **下一步** 。

### 选择集群

### 选择集群

3. 选择可用的 **东西网关地址** ，点击 **确定** 。

### 选择东西网关地址

### 选择东西网关地址

4. 在互联列表中可以看到新加入的网络分组，列表内的分组之间彼此建立了通信。

**!!! note**

- 没有东西网关的网络分组无法加入互联
- 加入分组互联操作会引起网络控制面重启，建议加入互联前确保网络分组配置无误

## 其他操作

### 1.更新东西网关地址。

用户可以对处于互联状态的网络分组增删东西网关地址。在互联列表中勾选一个分组，

点击 **更新东西网关地址** 。

更新东西网关地址

更新东西网关地址

更新地址后点击 **确定** 。

确定

确定

### 2.移出互联

如果网络分组不再需要和其他网络分组建立互联，或需要调整分组内集群及东西网关设

置，可以移出互联。 在网络分组互联中，选择需要移除的网络分组，点击 **移出互**

**联** 按钮，出现下图对话框。

移出互联

移出互联

经过二次确认，网络分组即可移出互联列表。

**!!! warn**

- 网络分组内的集群是可以互相通信的，无法直接通讯的集群需要依靠网络分组的东西向网关进行通讯，因此需要加入互联
- 如果要修改某个网络分组内的集群信息，强烈建议先将集群移出网络互联，修改完成后再加入互联，减少对整体网络互联的影响

### 3.删除网络分组

在需要删除的网络分组操作下拉框中点击 **删除网络分组**，即可删除所选网络分组。

删除分组

删除分组

#### 4. 关闭 **多云网络互联** 功能

在顶部 **基本设置** 区域，点击 **启用** 状态的滑块，将弹出关闭多云互联的前提条件：

关闭互联

关闭互联

先移除网络分组：

创建

创建

在文本框输入 **关闭基本设置** 进行确认后，即可关闭多云互联功能：

创建

创建

!!! note

- 关闭网络互联后，由于跨集群的服务地址信息已被缓存，会存在一定的生效延迟，一般在 1-2 分钟之内；特殊服务需手工重启网格组件才会生效

## 网格下多云网络流量分流配置

本文讲述如何在多云网络中为工作负载配置不同的流量。

前置条件：

- 服务 **helloworld** 运行于网格 **hosted-mesh** 的命名空间 **helloworld** 下
- 开启多云网络互联
- 网格提供 **ingressgateway** 网关实例

配置步骤：

- 1. 基于所属集群，对请求流量做权重分流；

流程

流程

- 2. 为两个集群的 **helloworld** 工作负载分别添加标签：

| 所属集群         | 标签      | 值  |
|--------------|---------|----|
| yl-cluster10 | version | v1 |
| yl-cluster20 | version | v2 |

添加标签

添加标签

- 3. 在左侧导航栏点击 **边车管理** -> **工作负载边车管理**，为两个集群的 **helloworld** 工作负载注入边车。

集群 01

集群 01

集群 02

集群 02

- 4. 在左侧导航栏点击 **流量治理** -> **目标规则** -> **创建**，创建两个服务版本。

服务版本

服务版本

对应的 YAML 如下：

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: helloworld-version
  namespace: helloworld
spec:
```

```

host: helloworld
subsets:
  - labels:
      version: v1
      name: v1
  - labels:
      version: v2
      name: v2

```

5. 在左侧导航栏点击 **流量治理** -> **网关规则** -> **创建**，创建网关规则。

## 网关规则

### 网关规则

对应的 YAML 如下：

```

apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: helloworld-gateway
  namespace: helloworld
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - hello.m.daocloud
      port:
        name: http
        number: 80
        protocol: http

```

6. 在左侧导航栏点击 **流量治理** -> **虚拟服务** -> **创建**，创建路由规则，基于权重比例把

流量分流至两个集群：

## 虚拟服务

### 虚拟服务

对应的 YAML 如下：

```

apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:

```

```

name: helloworld-version
namespace: helloworld
spec:
  gateways:
    - helloworld-gateway
  hosts:
    - helloworld.helloworld.svc.cluster.local
  http:
    - match:
        - uri:
            exact: /hello
      name: helloworld-routes
      route:
        - destination:
            host: helloworld
            port:
              number: 5000
            subset: v1
            weight: 30
        - destination:
            host: helloworld
            port:
              number: 5000
            subset: v2
            weight: 70

```

7.在左侧导航栏点击 **网格配置** -> **多云网络互联** ，开启多云网络互联。

多云网络互联

多云网络互联

8.通过 JMeter 发起 1000 次 get 请求，设置断言

JMeter01

JMeter01

JMeter02

JMeter02

9.查看请求聚合报告（设置断言 helloworld v2） ，成功率 70%，异常率比例接近 30%。

报告

报告



# 网络连通性检测

在现代云原生架构中，企业越来越多地采用多集群、多网络的部署模式以增强应用的弹性和可用性。然而，这种复杂的架构也带来了网络连通性的问题。在托管网格模式下，特别是多网络模式，确保集群之间的网络连通性变得尤为重要。为了解决这个问题，平台提供一种轻量易操作的方式来帮助用户快速检测集群之间的连通性。

!!! note

在托管网格模式下，如果是多网络模式，需要开启[多云互联配置](cluster-interconnect.md)。

主要功能如下：

- 轻量级检测代理：

在每个集群中部署轻量级检测代理，这些代理可以相互通信，并通过 HTTP/HTTPS 请求验证连接。

- 自动化检测流程：

通过点击按钮，用户可以发起检测请求，工具会自动遍历所有集群并进行连通性检测。

- 详细的结果：

会生成详细的检测结果，包括每个集群之间的连通性状态。如果检测到问题，结果中会提供可能的原因和解决建议。

## Istio 资源管理

**Istio 资源管理** 页面按资源类型列出了 Istio 的所有资源，为用户提供了各类资源的展示、创建、编辑、删除等能力。

img

img

## UI 操作示例

用户可以用 YAML 形式对资源进行增删改查操作。此处是一个 telemetry 遥测资源的创建/删除示例。

1. 在左侧导航栏中点击 **网格配置** -> **Istio 资源管理**，点击右上角的 **YAML 创建** 按钮。

img

img

2. 在 YAML 创建页面中，输入正确的 YAML 语句后点击 **确定**。

img

```
apiVersion: telemetry.istio.io/v1alpha1
kind: Telemetry
metadata:
  name: namespace-metrics
  namespace: default
spec:
  metrics: # (1)!
  - providers:
    - name: prometheus
    overrides:
    - tagOverrides:
      request_method:
        value: "request.method"
      request_host:
        value: "request.host"
```

1. 未指定选择算符，应用到命名空间中的所有工作负载

3. 返回资源列表，点击操作一列的 **⋮** 按钮，可以从弹出菜单中选择编辑和删除等更多操作。

img

img

## Istio 资源类型

Istio 常见的资源类型如下：

# 流量治理

## 资源类型

## 描述

DestinationRule (目标规则)

目标规则是服务治理中重要的组成部分, 目标规则通过端口、服务版本等方式对请求流量进行划分, 并对各请求分流量订制 Envoy 流量策略, 应用到流量的策略不仅有负载均衡, 还有最小连接数、离群检测等。

EnvoyFilter

该资源提供了对 Envoy 配置的能力, 可以定义新的过滤器、监听器。使用该资源时需谨慎, 错误配置可能会对网格环境的稳定性造成较大影响。注意: - EnvoyFilter 可以配置在 Istio 根目录 (对所有工作负载生效) 或某个工作负载 (使用工作负载选择标签); - 当多个 EnvoyFilter 作用于同一个工作负载时, 会按创建顺序优先执行;

Gateway (网关规则)

网关规则用于定义网格边缘的负

| 资源类型          | 描述  |
|---------------|---|
|               | <p>载均衡器, 用于将服务暴露于网格之外, 或提供内部服务的对外访问。相较于 k8s 的 ingress 对象, istio 的网关规则增加了更多的功能:   L4-L6 负载均衡   对外 mTLS   SNI 的支持   其他 istio 中已经实现的内部网络功能: Fault Injection, Traffic Shifting, Circuit Breaking, Mirroring 对于 L7 的支持, 网关规则通过与虚拟服务配合实现。</p> |
| ProxyConfig   | <p>用于暴露代理的配置选项, 例如: 代理的线程数。该资源为可选资源, 如不创建, 系统将使用内建默认值; 注意: - ProxyConfig 中任何配置变更需要重启相关工作负载才会生效; - 作用于网格或命名空间的 ProxyConfig 不可以包含任何工作负载选择标签, 否则将仅作用于选定的工作负载;</p>  |
| WorkloadEntry | <p>该资源提供了对非 k8s 标准工作</p>  |

## 资源类型

## 描述

ServiceEntry (服务条目)

负载的描述，例如：运行于虚拟机的工作负载。WorkloadEntry 需要与服务条目配合使用，由服务条目通过筛选器建立服务与工作负载之间的对应关系；

服务条目允许向 Istio 的内部服务注册表中添加其他条目，以便网格中自动发现的服务可以访问/路由到这些手动指定的服务。服务条目描述服务的属性包括：DNS 名称、VIP、端口、协议、端点等。这些服务可以是网格外部的(例如:web API) 或网格内部无法自动注册的服务（例如：一个数据库，几个 VM）。

SideCar

Sidecar 用于描述边车对工作负载实例之间的流量转发配置。默认情况下,Istio 将支持转发网格中所有工作负载实例之间的通信，并接受与工作负载相关的所有端口流量。

注意： - 根命名空间下的 SideCar

## 资源类型

## 描述

VirtualService (虚拟服务)

对所有没有配置 SideCar 的命名空间及工作负载生效； - 任意命名空间或工作负载如果存在多个 SideCar，将被定义为行为未定义（无生效资源）；

在虚拟服务中, 提供了 HTTP、TCP、TLS 三种协议的路由支持, 可以通过多种匹配方式（端口、host、header 等）实现对不同的地域、用户请求做路由转发, 分发至特定的服务版本中, 按权重比划分负载, 并提供了故障注入、流量镜像等多种治理工具。

WorkloadGroup

WorkloadGroup 描述了工作负载实例的集合, 定义了工作负载实例引导代理的规范细节, 包括元数据和标识。WorkloadGroup 仅用于非 Kubernetes 工作负载, 模拟 Kubernetes 中边车注入和部署方式, 引导 Istio 代理,

## 安全治理

### 资源类型

### 描述

AuthorizationPolicy（授权策略）

授权策略类似于一种四层到七层的“防火墙”，它会像传统防火墙一样，对数据流进行分析和匹配，然后执行相应的动作。无论是来自外部的请求，或是网格内服务间的请求，都适用授权策略。

PeerAuthentication（对等身份认证）

对等身份认证提供服务间的双向安全认证，同时密钥以及证书的创建、分发、轮转也都由系统自动完成，对用户透明，从而大大降低了安全配置管理的复杂度。

RequestAuthentication（请求身份认证）

请求身份认证用于外部用户对网格内部服务发起请求的认证，用户使用 jwt 实现请求加密；每个请求身份认证需要配置一个授权策略

## 代理扩展

### 资源类型

### 描述

WasmPlugin

WasmPlugin 通过 WebAssembly 过滤器为

| 资源类型 | 描述   |
|------|--|
|      | Istio 代理提供扩展功能，其提供的过滤器能力可以和 Istio 内部过滤器形成复杂交互关系。 |

## 系统设置

| 资源类型      | 描述   |
|-----------|--|
| Telemetry | 该资源定义了如何为网格内的工作负载生成遥测，为 Istio 提供了指标、日志、全链路追踪三项可观测性工具的配置。<br><br>注意：在 Istio 根目录创建并且不包含工作负载选择器的遥测资源将对网格全局生效。 |

## TLS 密钥管理

密钥（Secret）是以键/值对形式保存的、包含证书、私钥、凭证等敏感信息的对象，用于服务间的链路 TLS 加密通信。服务网格提供了界面化的 TLS 密钥管理功能。

## 准备凭证和私钥

以 helloworld 为例创建一个证书和一个私钥。

```
openssl req -out example_certs1/helloworld.example.com.csr -newkey rsa: 2048 -nodes -keyout example_certs1/helloworld.example.com.key -subj "/CN=helloworld.example.com/O=helloworld organization"
openssl x509 -req -sha256 -days 365 -CA example_certs1/example.com.crt -CAkey example_cert
```



```
s1/example.com.key -set_serial 1 -in example_certs1/helloworld.example.com.csr -out example_certs1/helloworld.example.com.crt
```

生成的证书类似如下：

```
- ----BEGIN CERTIFICATE REQUEST-----
MIICiDCCAXACAQAwwQzEfMB0GA1UEAwwWaGVsbG93b3JsZC5leGFtcGxlLmNvbTEg
MB4GA1UECgwXaGVsbG93b3JsZCBvcmdhbml6YXRpb24wggeiMA0GCSqGSIb3DQEB
AQUAA4IBDwAwggEKAoIBAQDHLmX2uLbgWyrGC1/FMVPCTOoOfnM0kKCQyAEbYxqX
HJV4CQ3V7UlyEIdj/w0QK+eY8dD8QVKKLiX4DCNYM7Rv/X2Jltw0GHG6788VstGy
1tvNv9u/wsgHV7J0ybxn+iElgppLTKiLjuqZv/8HPNvE/CcvGmPbH2depd5nvYxq
kTNhYU1T8wPfSPSRoPZncwqjwnFy+IkjWzO/NBYYtVDU21VAuDjmqF8oO6cFSulm
OG0GDKaE0eXkYnRQsfssZHtGSEgb/R6feZNhIa4DajHYiGu37qqdS+gPodzGtRGI
ryc2gsXJNynwISpw78ne3GCgvCVshHV1a9+XaWo0nkN9AgMBAAGgADANBgkqhkiG
9w0BAQsFAAOCAQEAZiDs4ICVbmZhscdIE/vJINWLL77dAa2nstFQHrck0rINdqDF
YYF84J8OjNpaHkIzWQ+icKJ9j6FjdBkmCvfpd9nMnihe6XTbfNNEx4hyGCg1My5x
XmeKsd/4U5oJCle3qb0U/KDsIAbdv859DnSGHnf3rwx22Fnallw2bnbyIkRXqIPO
QjCoOu9Yor2sxfDw+gVzdUnsKVQYRII4RPJrBL+DWs9uUu/LrfZx8VHyGQSSNaMY
ivThfuaDOSIWR1Rqd6+z13/nmJrV0HJWwOk18Of5O/Cqb9sx5aduML9zmd2hvpv5a
qCLeUusdlSW12hVnudaP7TM3RE7R6r1TwwXE2As==
- ----END CERTIFICATE REQUEST-----
```

生成的私钥类似如下：

```
- ----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQDHLmX2uLbgWyrG
C1/FMVPCTOoOfnM0kKCQyAEbYxqXHJV4CQ3V7UlyEIdj/w0QK+eY8dD8QVKKLiX4
DCNYM7Rv/X2Jltw0GHG6788VstGy1tvNv9u/wsgHV7J0ybxn+iElgppLTKiLjuqZ
v/8HPNvE/CcvGmPbH2depd5nvYxqkTNhYU1T8wPfSPSRoPZncwqjwnFy+IkjWzO/
NBYYtVDU21VAuDjmqF8oO6cFSulmOG0GDKaE0eXkYnRQsfssZHtGSEgb/R6feZNh
Ia4DajHYiGu37qqdS+gPodzGtRGIryc2gsXJNynwISpw78ne3GCgvCVshHV1a9+X
aWo0nkN9AgMBAAGgEASMH/oeRljx5BRxUo7dnUjZA7M9TgTDj9M2qiXOud79IW
HELSt3V5/SB1GPK+QMAKXq0efM8x5iRu43EsCTL7oGC02u4xkSfNh03SEK/BqOQy
dcDCABShG23EXEDLhAM20Yoj75gXjseumng6LN7kP9sURnYSPyP+letulelByX6Y
is/bKKGcs4L5txH6KpMJL8hOW/6nK1AREeDvmd4Y98CoU+OC2R9ThR2FFRXgdYxh
2xCZ68+XQ1bUf2+24YI9/73iPvKRPe9zZKPhMak1kd7Lnuy3DEXAvjhpPs2eqiJ9
J7n6DkDn/fAMN6yvkm07vBbWS8AV421iKX+rf7b3AQKBgQDjm4Zg+/NXj+egrJio
Qt7Ufklp1dGje60OfKSZj7F3pBh/R1+HHmXh0YHOvDy0OEBUY9h97XyYHIBvPMf
fnyr2S0czk/HpcS6Tnfj9QaVjtQdZ8h7Sf+3uPWHgqv+gBJarA5fQZSgNRezEBo
5mqh+wXvzQioTrm3E1nBYUxw8QKBgQDgBxoD0DrLgypIiqR95i4lyf4YI5MzguR
F2MLFkQ6QWdMfzZnh7Cs/CdqUD3weZTqPFom3zDe1NFHG5xnGrdIoKzUMZ5JVg8t
xznpcqk7mPnm8ghGZ6R515SF1515iEyy15ct1v+nKcjRNHmYe0d5tIWRDrfGcXJu
DfC73CT7TQKBgGJlqFSCI0vbeds9Mi3r2+XLgoS3o1nSWqrKgrTHAUy/Dz/l8Iu
OPmhxknV1taAKOPTB/JHjGVr/5x0u3w/x1DaHsA3BxG8vN/0jNuyzdfU2Cil9mol
QN+AmtKrT/uMIpeaAPe47gS4IBWioa02/Z1rz9MrhLSM44caXFBV6R9hAoGAaUgN
GsOuDdw7b9HwEdat00aFKjT1xZx92pK1Eg3JzJLWB+Y03Byxk+oAX/8LzMpmiFoK
-----END PRIVATE KEY-----
```

```
iAAVyHK9UyyPqQiuH+yarDIRUCbIBN9+wM3cfyMaNkWCTAwDCNueSdX/41SBrv6Q
ZpOGm7mQTXjauCUfZvvGVXBUP2crPrtAahjALHUCgYEAxGvVH0I6Uphpf4gGqqcm
lge0R6nGtPeKn6/Nh2Y3sHVxGvByFkuRXqZ2gSw6etleSJhQLLcvZm/K6FZu0sMQ
C4jSbYtJupL8x4uSCEH5X/na/6AwUfmwA0P7jGuKKhHgsHpH8OHly6eHonM7pzn
WQuRxaN+FuKE3RiKHFXkslks=
- ----END PRIVATE KEY-----
```

## 创建密钥

1. 进入某个网格后，在左侧导航栏点击 **网格配置** -> **TLS 密钥管理**，点击 **创建** 按钮。

点击创建

点击创建

2. 填写名称，选择命名空间，填入刚创建的凭证和私钥，根据情况添加标签后，点击 **确定**。

填写

填写

3. 屏幕提示创建成功，点击右侧的 **！** 按钮，可以执行编辑、YAML 编辑以及删除等操作。

填写

填写

本例创建成功后的密钥 YAML 如下：

```
metadata:
  name: secret001
  namespace: default
  uid: 1b43fc33-7898-40f9-9868-b65570f35a3d
  resourceVersion: '450493'
  creationTimestamp: '2023-04-20T07:09:33Z'
  labels:
    mspider.io/managed: 'true'
    test: teston0422
  annotations:
    ckube.daocloud.io/indexes: >-
    {"cluster":"nicole-hosted-mesh-hosted","createdAt":"2023-04-20T07:09:33Z","is_deleted":"false","labels":{"mspider.io/managed=true"},"test=teston0422"},"name":"secret001","namespace":"def
```

```

    ault", "type": "kubernetes.io/tls"}
    kube.doacloud.io/cluster: nicole-hosted-mesh-hosted
    managedFields:
      - manager: cacheproxy
        operation: Update
        apiVersion: v1
        time: '2023-04-20T07:09:33Z'
        fieldsType: FieldsV1
        fieldsV1:
          f:data:
            .: {}
          f:tls.crt: {}
          f:tls.key: {}
          f:metadata:
            f:labels:
              .: {}
            f:mspider.io/managed: {}
            f:test: {}
          f:type: {}
    data:
      tls.crt: >-
        TUIJQ2IEQ0NBWEFDQVFBd1F6RWZNQjBHQTfVRUF3d1dhR1ZzYkc5M2IzSnNaQzVsZU
        dGdGNHeGxMbU52YIRFZwpNQjRHQTfVRUNnd1hhR1ZzYkc5M2IzSnNaQ0J2Y21kaGJtbDZZW
        FJwYjI0d2dnRWINQTBHQ1Nxr1NJYjNEUUVCCkFRVUFBNELCRHdBd2dnRUtBb0ICQVFESEExt
        WDJ1TGJnV3lyR0MxL0ZNVIBDVG9Pb0ZuTTBtS0NReUFFYlI4cVgKSEpWNENRM1Y3VWx5R
        Ulkai93MFFLK2VZOGREOFFWS0tMaVg0RENOWU03UnYvWDJKbHR3MEDIRzY3ODhWc3RH
        eQoxdHZOdjl1L3dzZ0hWN0oweWJ4bitpRWxncHBMVEtpTGP1cVp2LzhIUE52RS9DY3ZHbVBIS
        DJkZXBkNW52WXhxCmtUTmhZVTFUOHdQZlNQ1JvUFpuY3dxanduRnkrSWtqV3pPL05CWV1
        0VkrVMjFWQXVESm1xRjhvTzZjRlN1bG0KT0cwR0RLYUuWZVhrWW5SUvNmC3NaSHRHU0
        VnYi9SNmZlWk5oSWE0RGFqSFpR3UzN3FxFZMrZ1BvZHPHdFJHSQpyeWMyZ3NYSk55d25JU
        3B3NzhuZTNHq2d2Q1ZzaEhWMWE5K1hhV28wbmtOOUFnTUJBQUdnQURBTkna3Foa2IHCjI3
        MEJBUXNGQUFPQ0FRRUFAaURzNEIDVmjTmBhSGtJendRK2ljS0o5ajZGamRCa21DdmZwZDluTW5
        paGU2WFRiZk5ORXg0aHlHq2cxTXk1eApYbWVLC2QvNFU1b0pDbGUzcWIwVS9LRHNJQWJk
        djg1OURuU0dlbmYzcnd4MjJGbmFJbHcyYm5ieUlrUlhxSVBPClFqQ29PdTIzY3Iyc3hmRHcrZ1ZaZ
        FVuc0tWUVlySUk0U1BKckJMK0RXczl1VXUvTHJmWng4Vkh5R1FTU05hTVkKaXZUaGZ1YUR
        PU0lXUjFScWQ2K3oxMy9ubUpYVjBISld3T2sxOE9mNU8vQ3FiOXN4NWfkdU1MOXptZDJocH
        Y1YQpxQ0xVXVZzGxTVzEyaFZudWRhUDdUTTNSRTdSNnIxVHd3WEUyQT09s
      tls.key: >-
        TUIJRXZRSUJBREFOQmdrcWhraUc5dzBCQVFFRkFBU0NCS2N3Z2dTakFnRUFBb0ICQVF
        ESEExtWDJ1TGJnV3lyRwpDMS9GTvZQq1RvT29Gbk0wa0tDUXIBRWJZeHFYSEpWNENRM1Y
        3VWx5RUlkai93MFFLK2VZOGREOFFWS0tMaVg0CkRDTIINN1J2L1gySmx0dzBHSEc2Nzg4Vn
        N0R3kxdHZOdjl1L3dzZ0hWN0oweWJ4bitpRWxncHBMVEtpTGP1cVoKdi84SFBODkUvQ2N2R2l
        QYkgyZGVwZDVudll4cWtUTmhZVTFUOHdQZlNQ1JvUFpuY3dxanduRnkrSWtqV3pPLwpOQll

```

ZdFZEVtIxVkf1REptcUY4b082Y0ZTdWxtT0cwR0RLYUuWZVhrWW5SUVNmc3NaSHRHU0Vn  
 Yi9SNmZiWk5oCklhNERhakhZaUd1MzdxWRTK2dQb2R6R3RSR0IyeWMYz3NYSk55d25JU3B3  
 NzhuZTNHQ2d2Q1ZzaEhWMWE5K1gKYVdvMG5rTjIBZ01CQUFFQ2dnRUFTTWgvb2VsbGp4N  
 UJSeFVvN2RuVWpaQTdNOVRnVERqOU0ycWlYT3VkNzIjVwpIRUxTdDNWNS9TQmxHUGsrU  
 U1hS1hxMGVmTTh4NWISdTQzRXNDVEw3b0dDMDJ1NHhrU2ZOaDAzU0VrL0JxT1F5CmRjRE  
 NBQINoRzIzRVhFRExoQU0yMFlvajc1Z1hqc2V1bW5nNkxON2tQOXNVUm5ZU1B5UCtsZXR1b  
 GVJQnlYNlkKaXMvYktLR2NzNEw1dHhINktwTUpMOGhPVy82bksxQVJFZURWbWQ0WTK4Q2  
 9VK09DMII5VGhSMkZGUlhnZFl4aAoyeENaNjgrWFEYyIVmMisYnFIJOS83M2lQdktSUGU5elpL  
 UGhNYWsx2Q3TG51eTNERVhBdmpocFBzMMVxaUo5Cko3bjZEa0RuL2ZBTU42eXZrbTA3dkJi  
 V1M4QVY0MjFpS1grcmY3YjNBuUcZ1FEam00Wmcrl05YaitlZ3JKaW8KUXQ3VWZJa2xwMW  
 RHamU2ME9mS1NaajdGM3BCaC9SMStISG1YaDBZSE92RHkwT0VCVvk5aDk3WHIzSEICdlBN  
 ZgpmbnlyMlMwY3prL0hwY1M2VG5GajlRYVZqdFFkVjhoN1NmKzN1UFdIZ3F2dStnQkphckE1Z  
 lFaU2dOUmV6RUJvCjVtcWgrd1h2elFpb1RybTNFMW5CWVV4dzhs0JnUURnQnhvRDBEcckne  
 XBJaXFSOTvpNGx5ZjRZSTVNemd1ck8KRjJNTEZrUTZRV2RNZnpabmg3Q3MvQ2RxVUQzd2V  
 aVHFQRm9tM3pEZTFORkhHNXhuR3JkSW9LeIVNWjVKVkc4dAp4em5wY2txN21Qbm04Z2hH  
 WjZSNTE1U0ZsNWw1aUV5eTE1Y3QxdituS2NqUk5IbVlIMGQ1dElXUkRyZkdjWEp1CkRmQzcz  
 Q1Q3VFFLQmdHSmpscUZTQ0kwdmJlZHM5TWkzcjlrWEExbn1MzbzFuU1dxcctncIRIQXVZL0R6L  
 2w4SXUKT1BtaHhrblYxdGFBS09QVEIvSkhqR1ZyLzV4MHUzdy94MURhSHNBm0J4Rzh2Ti8wa  
 k51eXpkZiUyQ2lsOW1vbApRTitBbXRLclQvdU1JcGVhQVBINDdnUzRJQldpb2EwMi9aMXJ6OU1  
 yaExTTTQ0Y2FYRkIjWNlI5aEFvR0FhVWdOCkdzT3VEZHc3Yjld0VkyXQwMGFGS2pUMXhae  
 DkycEsxRWczSnpKTFdCK1kwM0J5eGsrB0FYlzhMek1wbWIGb0sKaUFBVnlISzlVeXlQcVFpdUgr  
 eWFyREISVUNibEJOOST3TTNjZnlNYU5rV0NUQXdeQ051ZVNkWC80MVNCcnY2UQpacE9HbT  
 dtUVRyAmf1Q1VmWnZ2R1ZYQIVQMmNyUHJ0QWFoakFMSFVDZ1lFQXhHdlZIMEk2VXBoc  
 GY0Z0dxcWNtCjFnZTBSNm5HdFBIS242L05oMlkzc0hWeEd2QnlGa3VSWHFammdTdzZldGxlU0  
 poUUXMY3ZabS9LNkZadTBzTVEKQzRqU2JZVEp1cHVMOHg0dVNDRUglWC9uYS82QXdVZ  
 m13QTBQN2pHdUtlLaEhnc0hwSDhPSGx5NmVib25NN3B6bgpXUXVSeGFuK0Z1S0UzUmlLSEZ  
 Ya3Nsaz0s=

type: [kubernetes.io/tls](https://kubernetes.io/tls)

## 使用场景

创建好的 TLS 密钥可用于：

1. **目标规则**。添加策略配置时，启用客户端 TLS 后，可以在以下 2 种模式中选择创建

好的密钥：

- 简单模式
- 双向模式

### 目标规则

## 目标规则

2. **网关规则**。启用服务端 TLS 模式后，可以在以下 3 种模式中选择创建好的密钥：

- 简单模式
- 双向模式
- Istio 双向模式

## 网关规则

## 网关规则

# 服务网格组件资源自定义

本文介绍如何对通过[容器管理](#)配置网格的[控制面组件](#)资源。

## 前提条件

集群已被服务网格纳管，网格组件已正常安装； 登录账号具有全局服务集群及工作集群中

命名空间 istio-system 的 **admin** 或 **editor** 权限；

## 自定义操作

以托管网格下工作集群上 **istiod** 为例，具体操作如下：

1. 服务网格下查看托管网格 nicole-dsm-mesh 的接入集群为 nicole-dsm-c2，如下图所示。

### 接入集群

### 接入集群

2. 点击集群名称，跳转至 **容器管理** 模块中集群页面，点击进入 **工作负载** -> **无状态负**

**载** 页面查找 **istiod** ；

查找 istiod

查找 istiod

3. 点击工作负载名称进入 **容器配置** -> **基本信息** 标签页；

查看配额

查看配额

4. 点击编辑按钮，修改 CPU 和内存配额，点击 **下一步**、**确定**。

修改配额

修改配额

5. 查看该工作负载下的 Pod 资源信息，可见已变更。

确认配额

确认配额

## 组件资源弹性伸缩

用户可以在[容器管理](#)对服务网格的[控制面组件](#)实现弹性伸缩策略，目前提供了三种弹性伸缩方式：

- 指标收缩（HPA）
- 定时收缩（CronHPA）
- 垂直伸缩（VPA）

用户可以根据需求选择合适的弹性伸缩策略。下面以指标收缩（HPA）为例，介绍创建弹性伸缩策略的方法。

## 前提条件

确保集群已安装 helm 应用 **Metrics Server**，可参考[安装 metrics-server 插件](#)

环境依赖

环境依赖

## 创建策略

以专有集群的 **istiod** 为例，具体操作如下：

1. 在[容器管理]中选择对应集群，点击进入 **工作负载** -> **无状态负载** 页面查找 **istiod**；

查找 istiod

查找 istiod

2. 点击工作负载名称进入 **弹性伸缩** 标签页；

标签页

标签页

3. 点击 **编辑** 按钮，配置弹性伸缩策略参数；

- 策略名称：输入弹性伸缩策略的名称，请注意名称最长 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾，

例如 **hpa-my-dep**。

- 命名空间：负载所在的命名空间。
- 工作负载：执行弹性伸缩的工作负载对象。
- 目标 CPU 利用率：工作负载资源下 Pod 的 CPU 使用率。计算方式为：工作负载下所有的 Pod 资源/工作负载的请求 (request) 值。当实际 CPU 用量大于/小于目标值时，系统自动减少/增加 Pod 副本数量。

- 目标内存用量：工作负载资源下的 Pod 的内存用量。当实际内存用量大于/小于目标值时，系统自动减少/增加 Pod 副本数量。
- 副本范围：Pod 副本数的弹性伸缩范围。默认区间为 1 - 10。

编辑页

编辑页

4. 点击 **确定** 完成编辑，此时新的策略已生效。

## 更多弹性伸缩配置

请参考：

- [创建 HPA 弹性伸缩策略](#)
- [创建 VPA 弹性伸缩策略](#)

## OCP 环境接入服务网格配置方法

OCP (OpenShift Container Platform) 是 Red Hat 推出的容器平台。

本页说明服务网格接入 OCP 平台的操作步骤。

## SCC 安全策略设置

在 Openshift 集群中，为服务网格增加命名空间的 **privileged** 用户权限，以 istio-operator、

istio-system 两个命名空间为例：

```
oc adm policy add-scc-to-user privileged system: serviceaccount:istio-operator:istio-operator
oc adm policy add-scc-to-user privileged system: serviceaccount:istio-system:istio-system
```



## 接入 Openshift 集群

创建一个网格，接入 Openshift 集群。返回网格列表，发现 Openshift 集群已接入成功。

集群接入成功

集群接入成功

但后端会报错：

COMMIT

```
2022-10-27T07:06:50.610621Z info Running command: iptables-restore --noflush /tmp/iptables-rules-1666854410610268141.txt1105821213
```

```
2022-10-27T07:06:50.616716Z error Command error output: xtables parameter problem: iptables-restore: unable to initialize table 'nat'
```

Error occurred at line: 1

Try `iptables-restore -h` or `iptables-restore --help` for more information.

```
2022-10-27T07:06:50.616746Z error Failed to execute: iptables-restore --noflush /tmp/iptables-rules-1666854410610268141.txt1105821213, exit status 2
```

通过以下几步消除错误。

## OCP 激活 iptables

### 修改 YAML

参考以下 YAML，根据实际环境修改部署：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: dsm-init
  namespace: openshift-sdn
spec:
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: dsm-init
  template:
    metadata:
      labels:
```

```

    app: dsm-init
    type: infra
spec:
  containers:
  - command:
    - /bin/sh
    - -c
    - |
      #!/bin/sh
      set -x
      iptables -t nat -A OUTPUT -m tcp -p tcp -m owner ! --gid-owner 1337 -j REDIR
ECT --to-ports 15006
      iptables -t nat -D OUTPUT -m tcp -p tcp -m owner ! --gid-owner 1337 -j REDIR
ECT --to-ports 15006
      while true; do sleep 100d; done
    image: release.daocloud.io/mspider/proxyv2:1.15.0 # (I)!
    name: dsm-init
  resources:
    requests:
      cpu: 100m
      memory: 20Mi
  securityContext:
    privileged: true
  dnsPolicy: ClusterFirst
  hostNetwork: true
  hostPID: true
  nodeSelector:
    kubernetes.io/os: linux
  priorityClassName: system-node-critical
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: sdn
  serviceAccountName: sdn

```

### 1. 修改 proxy 的镜像地址

## 添加参数

在 globalmesh YAML 中添加以下一行参数：

```
istio.custom_params.components.cni.enabled: "true"
```

!!! note

OpenShift 4.1 以上版本不再使用 iptables，转而使用 nftables。

因此需要安装 istio CNI 插件，否则在边车注入时会出现如下错误，即无法执行 iptables-restore 命令。

```
```none
istio iptables-restore: unable to initialize table 'nat'
```
```

## 部署 istio-cni

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  components:
    cni:
      enabled: true
      namespace: istio-system
  values:
    sidecarInjectorWebhook:
      injectedAnnotations:
        k8s.v1.cni.cncf.io/networks: istio-cni
    cni:
      excludeNamespaces:
        - istio-system
      psp_cluster_role: enabled
      cniBinDir: /var/lib/cni/bin
      cniConfDir: /etc/cni/multus/net.d
      cniConfFileName: istio-cni.conf
      chained: false
```

## 兼容性测试报告 - OpenShift

基于 OpenShift，测试服务网格在创建专有网格时的兼容性。

本次测试报告以 OpenShift 4.11 为例，详细阐述了服务网格接入 OpenShift 专有网格模式时的兼容性测试过程。

## 测试目的

验证 DCE 5.0 服务网络上创建基于原生 istio, 从控制面/数据面聚合 OpenShift 单集群模式的专有网络。

## 测试环境

## 服务网格环境

基于 demo-dev.daocloud.io

## OpenShift 环境

准备了 OpenShift 3 个节点:

| HostName                      | IP           | Kubernetes<br>版本 | Crictl<br>版本 | OS 内核版本                      | OS<br>Release |
|-------------------------------|--------------|------------------|--------------|------------------------------|---------------|
| master01.mspider.openshift.io | 10.6.157.111 | v1.24.0+3882f8f  | 1.24.2       | 4.18.0-372.26.1.el8_6.x86_64 | rhcos         |
| master02.mspider.openshift.io | 10.6.157.112 | v1.24.0+3882f8f  | 1.24.2       | 4.18.0-372.26.1.el8_6.x86_64 | rhcos         |
| master03.mspider.openshift.io | 10.6.157.113 | v1.24.0+3882f8f  | 1.24.2       | 4.18.0-372.26.1.el8_6.x86_64 | rhcos         |

查看 3 个节点的状况:

```
oc get nodes -owide
```

输出类似于:

| NAME                          | STATUS                          | ROLES                         | AGE                          | VERSION                                    |              |
|-------------------------------|---------------------------------|-------------------------------|------------------------------|--|--------------|
| INTERNAL-IP                   | EXTERNAL-IP                     | OS-IMAGE                      |                              |  |              |
| KERNEL-VERSION                |                                 | CONTAINER-RUNTIME             |                              |  |              |
| master01.mspider.openshift.io | Ready                           | master.worker                 | 7d3h                         | v1.24.0+3882f8f                            | 10.6.157.111 |
| <none>                        | Red Hat Enterprise Linux CoreOS | 411.86.202209211811-0 (Ootpa) | 4.18.0-372.26.1.el8_6.x86_64 | cri-o: //1.24.2-7.rhaos4.11.gitca400e0.el8 |              |
| master02.mspider.openshift.io | Ready                           | master.worker                 | 7d2h                         | v1.24.0+3882f8f                            | 10.6.157.112 |
| <none>                        | Red Hat Enterprise Linux CoreOS | 411.86.202209211811-0 (Ootpa) | 4.18.0-372.26.1.el8_6.x86_64 | cri-o: //1.24.2-7.rhaos4.11.gitca400e0.el8 |              |

```
master03.mspider.openshift.io Ready    master.worker    7d2h   v1.24.0+3882f8f    10.6.157.113
<none>    Red Hat Enterprise Linux CoreOS 411.86.202209211811-0 (Ootpa) 4.18.0-372.2
6.1.el8_6.x86_64   cri-o: //1.24.2-7.rhaos4.11.gitca400e0.el8
```

## 测试过程

## 通过容器管理接入 OpenShift

获取 OpenShift 集群的 kubeconfig (~/.kube/config), 在 demo-dev 环境的容器管理中接入集群。

```
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: https://10.6.157.130:6443
    name: mspider
contexts:
- context:
    cluster: mspider
    user: admin
    name: admin
current-context: admin
preferences: {}
users:
- name: admin
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURStLS0tCk1JSURaekNDQWsrZ
0F3SUJBZ0lJV0wzelp0MFErbGt3RFFZSkvWklodmNOQVFFTEJRQXdOakVTTUJBR0ExVUUKQ
3hNSmIzQmxibk5vYVdaME1TQXdIZl1IEVIFRREV4ZGhaRzFwYmkxcmRXSmxZMj1lWm1sbkxY
TnBaMjVsY2pBZQpGdzB5TWpFd01qWXhNekE1TWpaYUZ3MHPNakV3TWpNeE16QTVNalPhT
URBeEZ6QVZCZ05WQkFvVERuTjVjM1JsCmJlUcHRZWE4wWlhKek1SVXdFd1IEVIFRREV3eHp
lWE4wWlcwNl1XUnRhVzR3Z2dFaU1BMEdDU3FHU0liM0RRRUlKQVVFVQUE0SUJEd0F3Z2dFS
0FvSUJBUNtZ2RkUNCTCtGcmRmQmVseGU2dkNnbWpZMXJqN09rV1dDTHVUVVWVJWgoy
aWg1NHBWUVByUzVkQVEvbUVaME1NcTVBSmQ4dE1maWpVcFlhNkU4d1pmeU0wZEM0d3h
0empXWnZvQWhSejZUCjFuRDFvd1RsZndJMDJjZ0lkZUpJSiJEa2tsV3JQUjVRZU0wbytlLdWYrNl
dFUDVSSW82bWVxVEtuUHBuMmRyaWwKQUxPSF1vQWFiaklqVG51YUM2QWNISzdyN1Vob
GtQQ2pRK3dKdVI1cld5eW9HVFBTmlc4MVd6Mm9UNi9UQ05UVQpYTDhncWFoa2dlWGpYZkl
0bzYweVVPnNv3K1dMNk1oc0NhVlpXZG1TaDVRMzZZdTISZEFKazduaHlVR3JkNXRYCmplR
m9LM0gyZjcyemwrOGdrcTdlVjNVYjRPy2pyU2VUUTlVNIY5NjhwQzcxQWdNqkFBR2pmekI5T
UE0R0ExVWQKRHdFQi93UUVBd0lGb0RBZEJnTlZlU1VFRmpBVUJnZ3JCZ0VGQlFjREFRWU1
Ld1ICQlFVSEF3SXdeQVIEVlIwVApBUUgVqkFjd0FEQWRCZ05WSFE0RUZnUVVEdW1uRWkx
WjRlLytlejJXcEY4bTBLODhGcE13SHdZRFZSMGpCQmd3CkZvQVVBMT1VjTVdYYjh5aE54TVgr
```

OUhBcWpneTBYMnN3RFFZSktvWklodmNOQVFFTEJRQRnZ0VCQUYvL1BvM3MKSwp5N2tva1p2OU1LUDJvMm9ubWdhY0owNklIay9VbTVIbC8rMVE5WmhDeDJWSDNtWnIyME5kQ0M5Rm03OFovZwpoK3IDV9ihbGRXTnAxUXhyaWJUbgpsecmNsek55L20vYWRmMUhKSGoram1mdEJFeCtmNkVEdFFQZ3BUZUICbK01CnhzMDFSUldwVTBKVNvsYU5RclphNHUyYVZnc3h3djYzdEEyeXlzb1pxbGFqWktNZjU3TEhNTjBKdE10K3FyTDMKS1p1R1pqblVvcDk3SFh0VDVBZ01jU1R1V0UyME9QR0tldFVvTVI3cldza2pvSIU5VWNSelBTQ1FGTWZCam53VQpNLzNRM0IYTmNVay9mSDdxZk1DV2pGL29sNWNXMnhZOTN4ekF0bTVFWVpUei9VRk4wOEFzS09qMS9hRWVBaHII CkFhclV5UHBEYtIwUkpJWT0KLS0tLS1FTkQgQ0VSVEIGSUNBVEU0tLS0tLQo=

**client-key-data:** LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSU1Fb3dJQkFBS0NBUUUVBcG9IU2FnZ1MvaGEzWHdYcGNYdXJ3b0pvMk5hNCt6cEZsZ2k3azFlaUd kb29lZUtWCIVENjB1WFFFUDVoR2REREt1UUNYZkxUSDRvMUtXSUFsUE1HWDhqTk hRdU1NYmM0MW1iNkFJVWMradzlhTUUKNVg4Q05ObklDSFhpU0NVUTVKSlZxejB1VUhqTktQaXJuL3UxaEQrVV NLT3BucWt5cHo2WjluYTRwUUN6aDBmdwpHbTR5STA1N21ndWdlSGl1NisxSVpaRHdvMFBzQ 2JrZWExc3NxQmt6MHRsdK5WczlxRSt2MHdqVTFGeS9JS21vClpJSgw0MTN5TGFPdE1sSXVyc1 BsaStqSWJBbWxXVm5aa29lVU4rbUx2VvhRQ1pPNTRjbEJxM2ViVjQzaGFDdHgKOW4rOXE1Zn ZJSkt1M2xkMUcrRG5JNjBuazBOditsZmV2S1F1OVFJREFRQUJBb0lCQUJ5dHo2Z2pxK0hIMTky dQpEdj1VNWNpaTNadzduN0RsNEladkNwL2RRcXhoUHDkL1YyaHk1SDNzMWE1K2MrUWZZM HRxSnExOEZkR1h0RzU1CjRINHVlaFZsYjZpOW9xNW5EaVJsQTN5MzRMZG1BQjdPN1ZENkI wOURFNGtoaE5BWVVraU1TK1VxcWNZQ2IKTysKQVJHVk1UYU9IT1JHRERrZnUzSEMvcEhN OFJDNI3dzJxV1BCazZJKzJWbHdGZTNDMUZGTGpHL0IxQkFPaUZkYQpibUxkMU85cEt4QV MvODIBSjZy d0RBY1MwWk0zaDRTTkl5LzIvcVY0MHRHUFFhaG5RWEJKcGpNRy9ZQWV0U3 JkCIVES3VnSkF6azVvT2Q0Z1pVSINzZFYvK3lCQlJlc2Zta3FpTTdxQXArZVpzbkxUUERYZy8zc DICcXd2eHJUKzcKb3BGaFVTRUNnWUVBeVEyZfZaRHZHcWljTzFXSVF1VFZYbFBsdjd4dTl2a klic1pxZWV3K1F1VXBnYkFoOXFQMApwbDJsaGtFR1VIZjZzeUdnb3BUTnRaci9IYzVCUXRSbzJ Gbz14Z25QZkxTQ21ZMS9FbTQyd3UzUmhJNVNZOU1lCmZWYVFTeS9vRTRrQ3Y1ZmpTSmM2 dkZ1aHVYTI1FWkhraWRmSHFValFDazduRTERyY2lxT01YNWNDZ1lFQTFBTSsKVitXWXIvb0JSa EVpck9vbVF1U1l1d1lRcHhKaVF1UFBzUk9jR3VkrUFxU2V6SVVRV2RSTERETE16cHhuUHG QgpGMHRRqnY4VmhrMk9uS0dRazAweVZnTlhnOHlvMnBINk8zQ0w1QUdiMTdPMIBvWmJ4Mk ozc1JNa3Fuc015US92CkJVbUJ6YXVTTWEwVXprREFoU0xHaU01V253dDNMOEpEaUE0ZWQx TUNnWUFWUFAzd3l3V25FRWFvc2VsWi82aFcKZFPcZ2g3eEZGSIMvdHZBS2Z4MDRnc2Uycm 05NENXdIBvemJZRHNObstiV2U5Sjl1Y1QrcHZuelNuallncENXTApMVVUzUIZRSXZWektjYnNK ckdEV2lKN0IYT3h4SIJxMmI4MkFVOGcxUUJUdFBsTkJHTkNZa3lscldMYmw3RDV6CkhUczNN N3plU2VWNUUzR0s1Nm50Y3dLQmdRRExNQVNheHE4cjBFVINPbS9xR2tuckNcEWIrVGNTZD VyMmtsQmwKVys3YkZkTm1KbUhPanFSYUF3eWR0em9lVvdUZDI1SnZXZENXcC9lZ0RFeG1N SzFYanI5MEVhWfk2ZGJXRUYzcQpRM1crWWWhCU2pLaFhpZnNCdm93Smg5ZzNEdEQxRFRFO DI2TFJBdUtNZTEyYVFoS0FSaERSNGpiQUhJUHDvSINLCkcWwDFnd0tCZ0J3cERINmJWMGc2a EQzN1VVOStpQnBTblVnWWNzLzZBVlJ0R0k5RDhreFFIb1JESnNWT2ZVV1QKaERDWDf2aUF pU0hEUStmSz1IZUhOWV5k5Z05iRmdRakpDdlAxbHFRYU9IZnltUGliWUo5OVgxWnZ5eWlqNjJkb Qo1a3gvMnhwOGJPanM0U2hCY21JcGc5TjFZODI1UWtpTlJ4MEl0NUdFRWJweFNXQ1U4TGEy Ci0tLS0tRU5EIFJTQSBQUklWQVRFIEtFWS0tLS0tCg==

在容器管理中查看接入集群的状态：

接入 openshift

接入 openshift

## 创建专有网格

- 1.登录 global-cluster 节点，查看服务网格同步容器管理接入集群的状态。

查看集群接入状态

查看集群接入状态

- 2.完成创建网格时的配置后，点击 **确定** 。

对于控制面集群，请选择通过容器管理接入的 openshift4-mspider 集群

创建网格

创建网格

- 3.OpenShift 集群成功接入网格

成功接入网格

成功接入网格

- 4.查看各个组件是否健康

组件状况

组件状况

组件状况

组件状况

## 接入时遇到的问题

- 1.提示错误 **istio-operator RS CreateFailed**

RS CreateFailed

RS CreateFailed

原因分析：OpenShift（安全上下文约束）SCC 对创建 Pod 有权限限制

解决办法：增加 SCC 用户命名空间权限 “privileged”

```
oc adm policy add-scc-to-user privileged -z istio-operator
oc adm policy add-scc-to-user privileged -z istio-system
oc adm policy add-scc-to-group privileged system:authenticated
```

重启 istio-operator rs 解决问题。

## 2. 提示错误 **message: no running Istio pods in “istio-system”**

原因分析：字面提示是 istio-system 命名空间下缺少 istiod/istio-ingressgateway service

pod。实际是网格全局边车资源限制中的内存资源参数格式错误；正确格式为 500Mi

资源参数格式

资源参数格式

解决办法：修改 yaml 文件，添加 Mi 单位

```
kubectrl edit gm -n mspider-system openshift -oyaml
```

修改单位

修改单位

## 3. 注入边车时提示 **pod Init: CrashLoopBackOff**

激活 ocp iptables：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: dsm-init
  namespace: openshift-sdn
spec:
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: dsm-init
  template:
    metadata:
      labels:
```



```

    app: dsm-init
    type: infra
spec:
  containers:
  - command:
    - /bin/sh
    - -c
    - |
      #!/bin/sh
      set -x
      iptables -t nat -A OUTPUT -m tcp -p tcp -m owner ! --gid-owner 1337 -j
      REDIRECT --to-ports 15006
      iptables -t nat -D OUTPUT -m tcp -p tcp -m owner ! --gid-owner 1337 -j
      REDIRECT --to-ports 15006
      while true; do sleep 100d; done
    image: release.daocloud.io/mspider/proxyv2:1.15.0
    name: dsm-init
  resources:
    requests:
      cpu: 100m
      memory: 20Mi
  securityContext:
    privileged: true
  dnsPolicy: ClusterFirst
  hostNetwork: true
  hostPID: true
  nodeSelector:
    kubernetes.io/os: linux
  priorityClassName: system-node-critical
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: sdn
  serviceAccountName: sdn

```

运行命令：

```
kubectrl edit gm -n mspider-system openshift -oyaml
```

添加： istio.custom\_params.components.cni.enabled: "true"

## 部署 bookinfo 应用测试

创建 bookinfo namespace，部署 bookinfo 应用时需要对命名空间添加 SCC。

```
oc adm policy add-scc-to-user privileged -z bookinfo
```

### 1. 启用命名空间边车注入

启用命名空间边车注入

启用命名空间边车注入

查看命名空间边车

查看命名空间边车

### 2. 终端查看 bookinfo 命名空间 Labels

```
oc describe ns bookinfo |grep "istio-injection-enabled"
```

屏幕输出类似于：

Labels: istio-injection-enabled

### 3. 部署 bookinfo 应用

```
oc apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/bookinfo/platform/kube/bookinfo.yaml -n bookinfo
```

部署 bookinfo

部署 bookinfo

```
oc get pod -n bookinfo
```

## 网格多云部署

本页说明如何在多云环境下部署服务网格。

## 前置条件

## 集群要求

- 1. 集群类型与版本**：明确当前集群的类型与版本，确保后续安装的服务网格能够在该集群中正常运行。

2. **提供可靠的 IP**：控制面集群必须提供一个可靠的 IP 给其他数据面集群来访问控制面。
3. **授权**：加入网格的集群需要提供一个拥有足够权限的远程密钥，允许 Mspider 对集群进行安装组件以及 Istio 控制面访问其他集群的 API Server。

## 多集群区域规划

在 Istio 中，Region、Zone 和 SubZone 是用于维护多集群部署中的服务可用性的概念。具体来说：

- **Region** 表示一个大区域，通常用于表示整个云提供商的数据中心区域；在 Kubernetes 中，标签 `topology.kubernetes.io/region` 确定节点的区域。
- **Zone** 表示一个小区域，通常用于表示数据中心中的一个子区域；在 Kubernetes 中，标签 `topology.kubernetes.io/zone` 确定节点的区域。
- **SubZone** 则是一个更小的区域，用于表示 Zone 中的一个更小的部分。Kubernetes 中不存在分区概念，因此 Istio 引入了自定义节点标签 [topology.istio.io/subzone](https://topology.istio.io/subzone) 来定义分区。

这些概念的作用主要在于帮助 Istio 管理不同区域间的服务可用性。例如，在多集群部署中，如果一个服务在 Zone A 中出现故障，Istio 可以通过配置自动将服务流量转移到 Zone B，从而保证服务可用性。

配置方式是通过在集群的 **每个节点** 添加相应的 Label：

| 区域      | Label                                      |
|---------|--|
| Region  | <code>topology.kubernetes.io/region</code> |
| Zone    | <code>topology.kubernetes.io/zone</code>   |
| SubZone | <code>topology.istio.io/subzone</code>     |

添加 Label 可以通过容器管理平台找到对应的集群，给相应节点配置 Label

image-20221214101221559

image-20221214101221559

image-20221214101633044

image-20221214101633044

## 网络规划

### 网络基础信息

- 网络 ID
- 网络版本
- 网络集群

### 网络规划

确认集群之间网络状态，根据不同状态配置 这部分主要在于多网络模式两部分的配置：

- 规划网络 ID
- 东西网络的部署与配置
- 如何将网络控制面暴露给其他工作集群。

网络模式有如下两种：

- 单网络模式 > 明确集群之间 Pod 网络是否能够直接通。 > 如果 Pod 网络能够直接通讯，证明是同网络模式，但是要注意如果其中 **Pod 网络出现冲突**，就需要选择不同网络模式
- 多网络模式 > 如果集群之间网络不通，需要给集群划分 **网络 ID**，并且需要不同网络区域的集群安装东西网关， > 并且配置相关配置，具体操作步骤在下面章节[不同网络模式的网络组件安装与配置](#)中。

# 规划表单

将上述所提到的集群与网络相关规划汇聚成一张表单，便于用户更加方便的参考。

## 集群规划

| 集群名                | 集   | 集群 pod 子网       | pod 网络 | 集群节点 & 网络区域                               | 集    | Master IP        |
|--------------------|-----|-----------------|--------|---|------|------------------|
|                    | 群   | (podSubnet)     | 互通关系   |   | 群    |                  |
|                    | 类   |                 |        |   | 版    |                  |
|                    | 型   |                 |        |   | 本    |                  |
| mdemo-c<br>luster1 | 标   | “10.201.0.0/16” | -      | master：<br><br>region1/zone1/subzone<br>1 | 1.25 | 10.64.3<br>0.130 |
| mdemo-c<br>luster2 | 标   | “10.202.0.0/16” | -      | master：<br><br>region1/zone2/subzone<br>2 | 1.25 | 10.6.23<br>0.5   |
| mdemo-c<br>luster3 | 标   | “10.100.0.0/16” | -      | master：<br><br>region1/zone3/subzone<br>3 | 1.25 | 10.64.3<br>0.131 |
|                    | 准   |                 |        |   |      |                  |
|                    | k8s |                 |        |   |      |                  |
|                    | 准   |                 |        |   |      |                  |
|                    | k8s |                 |        |   |      |                  |

## 网络规划

|       |                         |
|-------|-------------------------|
| 配置项   | 值                       |
| 网络 ID | mdemo-mesh              |
| 网络模式  | 托管模式                    |
| 网络模式  | 多网络模式（需要安装东西网关，规划网络 ID） |

|      |  |
|------|--|
| 配置项  | 值  |
| 网格版本 | 1.15.3-mspider                               |
| 托管集群 | mdemo-cluster1                               |
| 工作集群 | mdemo-cluster1、mdemo-cluster2、mdemo-cluster3 |

## 网络规划

由于上面的表单所知，集群之间不存在网络互通的情况，因此网格为多网络模式，需要规划

如下配置：

| 集群名            | 集群网格角色    | 集群网络标识 (网络 ID) | hosted-istiod LB IP | eastwest LB IP | ingress LB IP |
|----------------|-----------|----------------|---------------------|----------------|---------------|
| mdemo-cluster1 | 托管集群，工作集群 | network-c1     | 10.64.30.71         | 10.64.30.73    | 10.64.30.72   |
| mdemo-cluster2 | 工作集群      | network-c2     | -                   | 10.6.136.29    | -             |
| mdemo-cluster3 | 工作集群      | network-c3     | -                   | 10.64.30.77    | -             |

## 接入集群以及组件准备

用户需要准备符合要求的集群，集群可以新创建（创建集群也可以采用容器管理平台的创建能力），也可以是已经存在的集群。

但是后续网格所需的集群都必须将其接入容器管理平台。

## 接入集群

如果不是通过容器管理平台创建的集群，例如已经存在的集群，或者通过自定义方式（类似 kubeadm 或 Kind 集群）创建的集群都需要将集群接入容器管理平台。

接入集群

接入集群

接入集群

接入集群

## 确认可观测组件（可选）

网格的关键能力可观测性，其中需要关键的观测性组件便是 Insight Agent，因此如果需要拥有网格的观测能力，需要安装其组件。

容器管理平台创建集群方式创建的集群将默认安装 Insight Agent 组件。

其他方式需要在容器管理界面中，找到本集群中的 **Helm 应用**，选择 **insight-agent** 安装。

Helm 应用

Helm 应用

安装 insight

安装 insight

安装 insight

安装 insight

## 网格部署

通过服务网格创建网格，并且将规划的集群加入到对应的网格中。

## 创建网格

首先在网格管理页面 -> **创建网格**：

## 创建网格

### 创建网格

创建网格的具体参数如图显示：

1. 选择托管网格：多云环境下，只有托管网格模式能够纳管多集群
2. 输入唯一的网格名称
3. 按照前置条件环境，选择预选的符合要求的网格版本
4. 选择托管控制面所在的集群
5. 负载均衡 IP：该参数为暴露控制面的 Istiod 所需要的参数，需要预先准备
6. 镜像仓库：在私有云中，需要将网格所需的镜像上传仓库，公有云建议填入 `release.daocloud.io/mspider`

### 上传镜像

### 上传镜像

网格处于创建中，需要等待网格创建完成后，状态由 **创建中** 转变成 **运行中**；

## 暴露网格托管控制面 Hosted Istiod

### 确认托管网格控制面服务

确保网格状态正常后，观察控制面集群 `mdemo-cluster1` 的 `istio-system` 下面的 Service 是否都成功绑定了 LoadBalancer IP。

### 绑定 lb ip

### 绑定 lb ip

发现托管网格控制面的服务 `istiod-mdemo-cluster-hosted-lb` 的没有分配 LoadBalancer IP 需要额外处理。



## 分配 EXTERNAL IP

在不同的环境申请或者分配 LoadBalancer IP 的方式不同，尤其是公有云环境，需要根据公有云厂商所提供的方式去创建 LoadBalancer IP。

本文演示 demo 采用了 metallb 的方式，给所属 LoadBalancer Service 分配 IP，相关部署与配置参考 [Metallb 安装配置](#) 部分。

部署完 metallb 以后，再次[确认托管网格控制面服务](#)。

## 验证托管控制面 Istiod EXTERNAL IP 是否通畅

在非托管集群环境中验证托管控制面 Istiod，本次实践通过 curl 的方式验证，如果返回 400

错误，基本可以判定网络已经打通：

```
hosted_istiod_ip="10.64.30.71"
curl -I "${hosted_istiod_ip}: 443"
# HTTP/1.0 400 Bad Request
```

## 确认并且配置网格托管控制面 Istiod 参数

### 1. 获取托管网格控制面服务 EXTERNAL IP

在网格 **mdemo-mesh** 控制面集群 **mdemo-cluster1** 中去确认托管网格控制面服务

**istiod-mdemo-mesh-hosted-lb** 已经分配 LoadBalancer IP 以后，并且记录其 IP，示

例如下：

确认

确认

确认托管网格控制面服务 **istiod-mdemo-mesh-hosted-lb** EXTERNAL-IP 为 **10.64.30.72** 。

### 2. 手动配置网格托管控制面 Istiod 参数

首先，在容器管理平台进入全局控制面集群 **kpanda-global-cluster**（如果无法确认相关集群的位置，可以询问相应负责人或者通过[查询全局服务集群](#)）

- **自定义资源模块** 搜索资源 **GlobalMesh**
- 接下来在 **mspider-system** 找到对应的网格 **mdemo-mesh**
- 然后编辑 YAML
- 在 YAML 中 `.spec.ownerConfig.controlPlaneParams` 字段增加 `istio.custom_params.values.global.remotePilotAddress` 参数；
- 其值为上文中记录的 **istiod-mdemo-mesh-hosted-lb** EXTERNAL-IP 地址：  
**10.64.30.72** 。

增加参数

增加参数

增加参数

增加参数

增加参数

增加参数

## 添加工作集群

在服务网络的图形界面上添加集群。

- 1.等待网格控制面创建成功后，选中对应网格，进入网格管理页面 -> **集群纳管** -> **添加**

**集群**：

添加集群

添加集群

2. 选中所需的工作集群后，等待集群安装网格组件完成；

安装组件

安装组件

3. 在接入过程中，集群状态会由 **接入中** 转变成 **接入成功**：

接入成功

接入成功

## 检测多云控制面是否正常

由于当前工作集群与网格控制面集群 Pod 网络不同，需要通过上文[暴露网格托管控制面 Hosted Istiod](#) 部分将控制面的 Istiod 暴露给公网。

验证工作集群 Istio 相关组件是否能运行正常，需要在工作集群中检查 **istio-system** 命名空间下的 **istio-ingressgateway** 是否能够正常运行：

验证

验证

## 不同网络模式的网格组件安装与配置

在本部分主要分为两个部分：

1. 给所有工作集群配置 **网络 ID**
2. 在所有网络不互通的集群中安装东西网关

这里先提到一个问题：为什么需要安装东西网关呢？由于工作集群之间 Pod 网络无法直达，因此服务跨集群通讯时也会出现网络问题，Istio 提供了一个解决方案就是东西网关。当目标服务位于不同网络时，其流量将会转发到目标集群的东西网关，东西网关将解析请求，将

请求转发到真正的目标服务。

由上面对东西网关的原理理解以后，又有一个新的问题，Istio 如何区分服务在什么网络中呢？Istio 要求用户每个工作集群安装 Istio 时，显示的定义 **网络 ID**，这也是第一步分存在的原因。

## 手动为工作集群配置 网络 ID

由于工作集群网络的不同，需要手动给每个工作集群配置 **网络 ID**。如果在实际环境中，集群之间 Pod 网络能够相互直达，就可以配置成同一个 **网络 ID**。

让我们开始配置 **网络 ID**，其具体流程如下：

1. 首先进入全局控制面集群 **kpanda-global-cluster**（如果无法确认相关集群的位置，可以询问相应负责人或者通过[查询全局服务集群](#)）

2. 然后在 **自定义资源模块** -> 搜索资源 **MeshCluster**

3. **mspider-system** 命名空间下找到加入网格的工作集群，本次案例的工作集群有：

**mdemo-cluster2** 、 **mdemo-cluster3**

4. 以 **mdemo-cluster2** 为例，编辑 YAML

- 找到字段 `.spec.meshedParams[].params`，给其中参数列增加 **网络 ID** 字段
- 参数列的注意事项：
  - 需要确认 `global.meshID: mdemo-mesh` 是否为同一个网格 ID
  - 需要确认集群角色 `global.clusterRole: HOSTED_WORKLOAD_CLUSTER` 是否为工作集群
- 添加参数 `istio.custom_params.values.global.network`，其值按照最初的[规划表](#)中的网络 ID：**network-c2**

编辑 YAML

编辑 YAML

编辑 YAML

编辑 YAML

编辑 YAML

编辑 YAML

重复上述步骤，给所有工作集群（**mdemo-cluster1**、**mdemo-cluster2**、**mdemo-cluster3**）加上 **网络 ID**。

## 为工作集群的 istio-system 标识 网络 ID

进入容器管理平台，进入对应的工作集群：**mdemo-cluster1**、**mdemo-cluster2**、**mdemo-cluster3** 的命名空间添加网络的标签。

- 标签 Key：**topology.istio.io/network**
- 标签 value：**\${CLUSTER\_NET}**

下面以 **mdemo-cluster3** 为例，找到 **命名空间**，选中 **istio-system** -> **修改标签**。

标识网络 ID

标识网络 ID

标识网络 ID

标识网络 ID

# 手动安装东西网关

## 创建网关实例

确认所有工作集群中的 Istio 相关组件都就绪以后，开始安装东西网关。

在工作集群中通过 IstioOperator 资源安装东西网关，东西网关的 YAML 如下：

!!! note

一定要根据当前集群的 \_\_网络 ID\_\_ 修改参数。

```
```bash
linenums="1" # cluster1 CLUSTER_NET_ID=network-c1 CLUSTER=mdemo-cluster1
LB_IP=10.64.30.73
```

## cluster2

```
CLUSTER_NET_ID=network-c2 CLUSTER=mdemo-cluster2 LB_IP=10.6.136.29
```

## cluster3

```
CLUSTER_NET_ID=network-c3 CLUSTER=mdemo-cluster3 LB_IP=10.64.30.77
MESH_ID=mdemo-mesh HUB=release-ci.daocloud.io/mspider ISTIO_VERSION=1.15.3-mspider
cat < eastwest-iop.yaml apiVersion: install.istio.io/v1alpha1 kind: IstioOperator metadata: name:
eastwest namespace: istio-system spec: components: ingressGateways: - enabled: true k8s: env: -
name: ISTIO_META_REQUESTED_NETWORK_VIEW value: ${CLUSTER_NET_ID} # 修改为当前集
群的网络 ID service: loadBalancerIP: ${LB_IP} # 修改为本集群为东西网关规划的 LB IP ports:
- name: tls port: 15443 targetPort: 15443 type: LoadBalancer label: app: istio-eastwestgateway
istio: eastwestgateway topology.istio.io/network: ${CLUSTER_NET_ID} # 修改为当前集群的网络
ID name: istio-eastwestgateway profile: empty tag: ${ISTIO_VERSION} values: gateways:
istio_ingressgateway: injectionTemplate: gateway global: network: ${CLUSTER_NET_ID} # 修改为
当前集群的网络 ID hub: ${HUB} # 可选，如果无法翻墙或者私有仓库，请修改 meshID:
${MESH_ID} # 修改为当前集群的 网格 ID (Mesh ID) multiCluster: clusterName: ${CLUSTER}
# 修改为当前 EOF
```

其创建方式为：

1. 在容器管理平台进入相应的工作集群
2. \_\_自定义资源\_\_ 模块搜索 \_\_IstioOperator\_\_
3. 选中 \_\_istio-system\_\_ 命名空间
4. 点击 \_\_创建 YAML\_\_

![创建网关实例](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/create-ew.png)

![创建网关实例](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/create-ew2.png)

### ### 创建东西网关 Gateway 资源

在网格的 \_\_网关规则\_\_ 中创建规则：

```
```yaml
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: cross-network-gateway
  namespace: istio-system
spec:
  selector:
    istio: eastwestgateway
  servers:
    - hosts:
        - "*.local"
      port:
        name: tls
        number: 15443
        protocol: TLS
      tls:
        mode: AUTO_PASSTHROUGH
```

## 设置网格全局网络配置

在安装完东西网关以及网关的解析规则以后，需要再所有集群中声明网格中的东西网关配置。

在 容器管理平台进入全局控制面集群 **kpanda-global-cluster** （如果无法确认相关集群的位

置，可以询问相应负责人或者通过[查询全局服务集群](#))

- 在 **自定义资源** 部分搜索资源 **GlobalMesh**
- 接下来在 **mSpider-system** 找到对应的网格 **mdemo-mesh**
- 然后编辑 YAML

在 YAML 中 `.spec.ownerConfig.controlPlaneParams` 字段增加一系列

`istio.custom_params.values.global.meshNetworks` 参数

*# !! 这两行配置缺一不可*

*# 格式: istio.custom\_params.values.global.meshNetworks.\${CLUSTER\_NET\_ID}.gateways[0].address*

*# istio.custom\_params.values.global.meshNetworks.\${CLUSTER\_NET\_ID}.gateways[0].port*

`istio.custom_params.values.global.meshNetworks.network-c1.gateways[0].address: 10.64.30.73 # (1)!`

`istio.custom_params.values.global.meshNetworks.network-c1.gateways[0].port: '15443' # (2)!`

`istio.custom_params.values.global.meshNetworks.network-c2.gateways[0].address: 10.6.136.29 # (3)!`

`istio.custom_params.values.global.meshNetworks.network-c2.gateways[0].port: '15443' # (4)!`

`istio.custom_params.values.global.meshNetworks.network-c3.gateways[0].address: 10.64.30.77 # (5)!`

`istio.custom_params.values.global.meshNetworks.network-c3.gateways[0].port: '15443' # (6)!`

1.cluster1

2.cluster3 东西网关端口

3.cluster2

4.cluster2 东西网格端口

5.cluster3

6.cluster3 东西网关端口

增加参数

增加参数



## 网络连通性 demo 应用与验证

### 部署 demo

主要是两个应用：[helloworld](#) 与 [sleep](#) (该两个 demo 属于 Istio 提供的测试应用)

集群部署情况：

```
集群 | helloworld 与版本 | sleep --|-----|--- mdemo-cluster1 | :heart: VERSION=vc1
| :heart: mdemo-cluster1 | :heart: VERSION=vc2 | :heart: mdemo-cluster1 | :heart: VERSION=vc3
| :heart:
```

### 容器管理平台部署 demo

推荐使用容器管理平台创建对应工作负载与应用，在容器管理平台找到对应集群，进入【控制台】执行下面操作。

下面以 mdemo-cluster1 部署 helloworld vc1 为例：

其中每个集群需要注意的点：

- 镜像地址：
  - helloworld: docker.m.daocloud.io/istio/examples-helloworld-v1
  - Sleep: curlimages/curl
- **helloworld** 工作负载增加对应 label
  - app: helloworld
  - version: \${VERSION}
- **helloworld** 工作负载增加对应的版本**环境变量**
  - SERVICE\_VERSION: \${VERSION}

部署 demo

部署 demo

部署 demo

部署 demo

部署 demo

部署 demo

部署 demo

部署 demo

## 命令行部署 demo

部署过程中需要用到的配置文件分别有：

- [gen-helloworld.sh](#)
- [sleep.yaml](#)

```
``bash linenums="1" # ----- cluster1 ----- kubectl create namespace sample
kubectl label namespace sample istio-injection=enabled
```

## deploy helloworld vc1

```
bash gen-helloworld.sh --version vc1 | sed -e "s/docker.io/docker.m.daocloud.io/" | kubectl apply
-f - -n sample
```

## deploy sleep

```
kubectl apply -f sleep.yaml -n sample
```

## ----- cluster2 -----

```
kubectl create namespace sample kubectl label namespace sample istio-injection=enabled
```

## deploy helloworld vc2

```
bash gen-helloworld.sh --version vc2 | sed -e "s/docker.io/docker.m.daocloud.io/" | kubectl apply
-f - -n sample
```

## deploy sleep

```
kubectl apply -f sleep.yaml -n sample
```

## ——— cluster3 ———

```
kubectl create namespace sample kubectl label namespace sample istio-injection=enabled
```

## deploy helloworld vc3

```
bash gen-helloworld.sh --version vc3 | sed -e "s/docker.io/docker.m.daocloud.io/" | kubectl apply -f - -n sample
```

## deploy sleep

```
kubectl apply -f sleep.yaml -n sample
```

### 验证 demo 集群网络

```
```bash linenums="1"
```

```
# 任意选择一个集群执行
```

```
while true; do kubectl exec -n sample -c sleep \
    "$(kubectl get pod -n sample -l app=sleep -o jsonpath='{.items[0].metadata.name}')" \
    -- curl -sS helloworld.sample:5000/hello; done
```

# 预期结果会轮询三个集群的不同版本，结果如下：

```
# Hello version: vc2, instance: helloworld-vc2-5d67bc48d8-2tpxd
# Hello version: vc2, instance: helloworld-vc2-5d67bc48d8-2tpxd
# Hello version: vc2, instance: helloworld-vc2-5d67bc48d8-2tpxd
# Hello version: vc2, instance: helloworld-vc2-5d67bc48d8-2tpxd
# Hello version: vc1, instance: helloworld-vc1-846f79cc4d-2d8kd
# Hello version: vc1, instance: helloworld-vc1-846f79cc4d-2d8kd
# Hello version: vc1, instance: helloworld-vc1-846f79cc4d-2d8kd
# Hello version: vc3, instance: helloworld-vc3-55b7b5869f-trl8v
# Hello version: vc1, instance: helloworld-vc1-846f79cc4d-2d8kd
# Hello version: vc3, instance: helloworld-vc3-55b7b5869f-trl8v
# Hello version: vc2, instance: helloworld-vc2-5d67bc48d8-2tpxd
```

```
# Hello version: vc1, instance: helloworld-vc1-846f79cc4d-2d8kd
# Hello version: vc2, instance: helloworld-vc2-5d67bc48d8-2tpxd
```

## 拓展

## 其他创建集群方式

### 通过容器管理创建集群

集群创建可以存在多种方式，推荐使用容器管理中的创建集群功能，但是用户可以选择其他创建方式，本文提供的其他方案可以参考拓展章节的[其他创建集群方式](#)

创建集群

创建集群

创建集群

创建集群

创建集群

创建集群

创建集群

创建集群

可以灵活的选择集群需要拓展的组件，网格的可观测能力必须依赖 Insight-agent

安装 insight

安装 insight

如果集群需要定义更多的集群高级配置，可以在本步骤添加。

集群高级配置

集群高级配置

创建集群需要等待 30 分钟左右。

等待

等待

## 通过 kubeadm 创建集群

```
kubeadm init --image-repository registry.aliyuncs.com/google_containers \
--apiserver-advertise-address=10.64.30.131 \
--service-cidr=10.111.0.0/16 \
--pod-network-cidr=10.100.0.0/16 \
--cri-socket /var/run/cri-dockerd.sock
```

## 创建 kind 集群

```
```bash
linenums="1" cat < kind-cluster1.yaml kind: Cluster apiVersion: kind.x-k8s.io/v1alpha4
networking: podSubnet: "10.102.0.0/16" # 安装网络规划阶段，定义规划的 pod 子网

serviceSubnet: "10.122.0.0/16" # 安装网络规划阶段，定义规划的 service 子网
apiServerAddress: 10.6.136.22 nodes: - role: control-plane image:
kindest/node:v1.25.3@sha256:f52781bc0d7a19fb6c405c2af83abfeb311f130707a0e219175677e
366cc45d1 extraPortMappings: - containerPort: 35443 # 如果无法申请 LoadBalancer 可以临
时采用 nodePort 方式 hostPort: 35443 # set the bind address on the host
EOF cat < kind-cluster2.yaml kind: Cluster apiVersion: kind.x-k8s.io/v1alpha4 networking:
podSubnet: "10.103.0.0/16" # 安装网络规划阶段，定义规划的 pod 子网 serviceSubnet:
"10.133.0.0/16" # 安装网络规划阶段，定义规划的 service 子网 apiServerAddress:
10.6.136.22 nodes: - role: control-plane image:
kindest/node:v1.25.3@sha256:f52781bc0d7a19fb6c405c2af83abfeb311f130707a0e219175677e
366cc45d1 extraPortMappings: - containerPort: 35443 # 如果无法申请 LoadBalancer 可以临
时采用 nodePort 方式 hostPort: 35444 # set the bind address on the host
EOF
kind create cluster --config kind-cluster1.yaml --name mdemo-kcluster1 kind create cluster --
config kind-cluster2.yaml --name mdemo-kcluster2
```

### ### Metallb 安装配置

#### #### demo 集群 metallb 网络池规划记录

集群名	IP 池	IP 分配情况
-----	-----	-----
mdemo-cluster1	10.64.30.71-10.64.30.73	-
mdemo-cluster2	10.6.136.25-10.6.136.29	-
mdemo-cluster3	10.64.30.75-10.64.30.77	-

#### #### 安装

#### #### 容器管理平台 Helm 安装

推荐使用容器管理平台中 \_\_Helm 应用\_\_ -> \_\_Helm 模板\_\_ -> 找到 metallb -> \_\_安装\_\_。

![安装 metallb](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/install-metallb-from-helm.png)

![安装 metallb](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/install-metallb-from-helm1.png)

#### ##### 手动安装

参阅 [MetalLB 官方文档](https://metallb.org/installation/)。

注意：如果集群的 CNI 使用的是 calico,你需要禁用 calico 的 BGP 模式，否则会影响 MetalLB 的正常工作。

```
``bash linenums="1"
```

```
# 如果 kube-proxy 使用的是 IPVS 模式，你需要启用 staticARP
```

```
# see what changes would be made, returns nonzero returncode if different
```

```
kubectl get configmap kube-proxy -n kube-system -o yaml | \
```

```
sed -e "s/strictARP: false/strictARP: true/" | \
```

```
kubectl diff -f - -n kube-system
```

```
# actually apply the changes, returns nonzero returncode on errors only
```

```
kubectl get configmap kube-proxy -n kube-system -o yaml | \
```

```
sed -e "s/strictARP: false/strictARP: true/" | \
```

```
kubectl apply -f - -n kube-system
```

```
# 部署 metallb
```

```
kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.13.7/config/manifests/metall
```

b-native.yaml

# 检查 pod 运行状态

```
kubectl get pods -n metallb-system
```

# 配置 metallb

```
cat << EOF | kubectl apply -f -
```

```
apiVersion: metallb.io/v1beta1
```

```
kind: IPAddressPool
```

```
metadata:
```

```
  name: first-pool
```

```
  namespace: metallb-system
```

```
spec:
```

```
  addresses:
```

```
    - 10.64.30.75-10.64.30.77 # 根据规划修改
```

```
EOF
```

```
cat << EOF | kubectl apply -f -
```

```
apiVersion: metallb.io/v1beta1
```

```
kind: L2Advertisement
```

```
metadata:
```

```
  name: example
```

```
  namespace: metallb-system
```

```
spec:
```

```
  ipAddressPools:
```

```
    - first-pool
```

```
  interfaces:
```

```
    - enp1s0
```

```
    # - ens192 # 根据不同机器的网卡命名不同，请修改
```

```
EOF
```

## 增加给对应服务增加指定 IP

```
kubectl annotate service -n istio-system istiod-mdemo-mesh-hosted-lb metallb.universe.tf/address-pool='first-pool'
```

## 验证

```
``bash
linenums="1" kubectl create deploy nginx --image docker.m.daocloud.io/nginx:latest --port 80 -n default
kubectl expose deployment nginx --name nginx-lb --port 8080 --target-port 80 --type LoadBalancer -n default
```

# 获取对应服务 EXTERNAL-IP

```
kubectl get svc -n default # NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE # kubernetes
ClusterIP 10.212.0.1 443/TCP 25h # nginx-lb LoadBalancer 10.212.249.64 10.6.230.71 8080:
31881/TCP 10s
```

```
curl -I 10.6.230.71: 8080 # HTTP/1.1 200 OK # Server: nginx/1.21.6 # Date: Wed, 02 Mar 2022
15:31:15 GMT # Content-Type: text/html # Content-Length: 615 # Last-Modified: Tue, 25 Jan
2022 15:03:52 GMT # Connection: keep-alive # ETag: "61f01158-267" # Accept-Ranges: bytes
```

### 查询全局服务集群

通过容器管理的集群列表界面，通过搜索 \_\_集群角色：全局服务集群\_\_。

![全局服务集群](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/get-kpanda-global-cluster.png)

## # 多集群网络通讯原理与架构分析

近年来，多集群架构已经成为“老生常谈”。我们喜欢高可用，喜欢异地多可用区，而多集群架构天生就具备了这样的能力。

另一方面我们也希望通过多集群混合云来降低成本，利用到不同集群各自的优势和特性，以便使用不同集群的最新技术与支持现实中越来越复杂的云上业务。

最为本质的问题时单集群模式下存在承载能力上限，并且集群故障时，无法快速故障转移。

就是因为这些种种原因，多集群几乎成为了云计算的新潮流。

随着新架构的诞生，新的难题也接踵而来，尤其是在多集群场景下，其中业务的治理、监控、网络通讯也面临着严重的考验。

这时不得不提到服务网格，该技术从理论上来说对于多云多集群场景下，从设计基因中就自带了解决上面问题的能力。

为什么说服务网格能够支持复杂的多云场景呢？

服务网格将自身能力下沉到基础设施层面，因此从理论上来说，在多云环境中，它天然能够感知自己云上业务的各种信息，比如说服务地址、服务之间链路关系等网络信息，网格将感知的信息聚会成一个完整的多云网络拓扑，在一个已知且能控制的多云拓扑中，用户能够从高纬度的观测并且控制业务通讯。

上述都只理论分析，再多的理论分析也不如一个实践论证，因此我们将自身服务网格产品（Mspider）到底如何支持多云环境进行一个由浅入深的整体剖析。

## ## 多云服务网格的优势



首先，我们回到最初的起点，我们为什么要选择多云服务网格？

所有的技术都是为了解决问题而存在，不具有优势与解决问题的技术，是虚假且没有意义的。因此我们简单介绍一下相对于单集群场景下，多云场景下的服务网格能够给我们带来哪些优势？

#### ### 故障隔离与故障转移

在单集群场景下，业务服务 A 出现问题时，我们通过将业务流量转移到同集群的另一个业务实例上。

但是如果集群出现故障时，这时候会变成多个业务同时故障，这时候如何将该集群的所有业务转移到另一个集群上，这是已经非常费时且复杂地。

但是实际地生产环境下，每分每秒是非常昂贵。

但是多云场景下，集群维度的故障转移是基本能力，用户能够快速转移其所有业务的流量。用户只需要定义集群之间故障转移的策略，例如 A 集群故障时，将业务流量转移到集群 B 中。

#### ### 地域感知与故障转移

当多云环境下，对云上的地域进行定义；在业务健康时，可以让业务之间通讯以就近原则，发生故障时，可以快速转移至最近的地域环境中。

地域感知对于多云的重要性不言而喻。

因为现实世界、实际地域区域会影响网络通讯，中间有时间空间资源等多种因素。

例如当中国北京的集群都故障时，随机将业务转移到美国，但是实际环境中中国上海的集群空闲在那里，这肯定是不合理的。

#### ### 多维度团队或项目隔离

虽然随着多云的体量扩大，但是团队的复杂度并没有增加，团队可以将多云划分成不同的区域、不同的集群、不同业务项目。

团队拥有多云能力的情况下，可以只需要管理自己相关的集群与业务。

#### ### 多云维度的观测能力

单集群模式下，用户只能知道当前集群内的业务运行情况，每个集群之间的业务通讯时不透明且独立的。用户无法感知到集群外的业务通讯链路。

但是多云场景下，只要属于同一个网格，网格能将同一网络下不同集群之间的通讯情况进行聚合分析，将其绘制成一张完整的多云网络拓扑。

好比原先的单集群只是一张 2D 的照片，现在多云场景下，服务网格能够提供一张 3D 立体的拓扑网络。

这不仅仅是范围的扩大，而是维度层次上的增加。

#### ## 多云下网络通讯原理

在了解了多云场景下巨大的优势，我们回归现实，多云场景有伴随着哪些问题？

其中哪个问题最为根本？

首先我们从字面上理解一下这个意思，多个云环境？多个云厂商？

多个云集群？这里的多个，就意味着存在不同。

而且在多云世界里，网络通讯是一个非常硬核的问题，云厂商之间能够直接通讯吗？

多集群之间如何通讯无阻？

在单集群的情况，所以的工作负载都在同一网络上（集群内部网络），但是多集群的场景下，每个集群可能存在多种因素导致网络无法互通，这里列举常见的几种情况：

- 地域问题导致集群实际网络之间存在网络隔离
- 初始化集群时，保持默认配置导致集群子网都一样，因此集群与集群之间网络会存在 IP 冲突。

以上几种情况下的多集群之间网络通讯是存在问题的，为了保持集群内部网络的独立性以及集群网络之间网络打通，Istio 提供了两种网络模型，用户可以根据集群状态与用户需求自由选择不同的网络模型。

### ### 单一网络模型

顾名思义，在单一网络的情况下，多集群之间的网络实例能给直接通讯，这时候不需要网络进行任何操作，服务之间网络通讯能够直达，其架构图如下：

![架构图](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/single-net.png)

单一网络模型的优势是非常明显的，用户通讯时，无需复杂的配置和中转，因此网络解析也更加直观，因此也能带来更高的网络性能。

在这时不得不提的是，有的用户会问，如果拥有多个集群，并且当时多集群之间网络存在冲突，但是又想要多集群之间网络选择单一网络模型时如何做？在这里简单提供一些网络改造需要解决的问题：

- 子网冲突问题：需要解决集群实例网络与集群服务网络的子网冲突问题
- 网络隔离问题：集群之间网络可能由于地域、安全等多种因素造成网络隔离的现状，用户可以选择通过隧道或直接路由，解决跨多个集群的 Pod IP 路由。

### ### 多网络模型

在多网络模型中，不同网络中的工作负载实例不能够直接通讯，因此服务网格提供了一种解决方案，让业务实例通过一个或多个 Mesh 网关相互访问，从而解决工作实例网络不能互相通讯的问题。

在服务通讯时，网格会根据网络分区进行服务发现，当请求服务与目标服务不属于同一网络分区时，服务请求将会转发给目标分区的东西网关，通过东西网关路由到真实的目标服务。

![服务请求分发](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster)

r/images/multi-net.png)

### ## 多云服务网格架构的抉择

在上述网络方面的思考以后，用户可能对服务网格自身架构存在好奇，例如服务网格控制面组件位置在哪里？

如何运行多集群模式？

这架构拥有哪些优势或者缺陷？

让我们具体分析现有服务网格提供的架构，现在主要有如下三种架构：

- 单集群单控制面
- 多集群单控制面
- 多集群多控制面

#### ### 单集群单控制面

在最简单的情况下，可以在单一集群上使用控制平面运行网格。

![控制平面运行网格](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multi-cluster/images/singlecluster-primary.png)

#### ### 多集群单控制面

多集群部署也可以共享控制平面实例。

在这种情况下，在核心控制集群部署核心的共享控制面实例，其网格核心的通讯策略都由 **共享控制面实例** 控制。

在从集群中，其实也存在一个从控制面实例，该实例其核心能力是 **控制本集群的边车生命周期管理**。

![共享控制面实例](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/multi-primary.png)

![控制本集群边车](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/multi-primary-control.png)

思考一个问题，从集群的边车如何与共享控制面建立连接？其回答时共享控制面需要暴露自身控制面服务。

在单一网络模型中主集群的共享控制平面可以通过稳定 IP(例如集群 IP)访问。

但是在多网络模型的多集群中，需要通过 Mesh 网关对外 **暴露** 共享控制平面。

在实际的生产环境，可以根据实际的云供应商或者内部网络的规划灵活选择；

例如内部负载均衡器，避免将控制面暴露到公共网络中，因为其存在潜在的安全隐患。

共享控制面在上面描述中都是属于某一个网格集群中,其实其远程共享控制面也可以部署在网格外的集群中,这样能够让网格的控制面与数据面在物理上实现隔离,避免主集群的控制面与数据面同时出现问题,可以分散风险。

![远程共享部署](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/remote-primary.png)

### ### 多集群多控制面

为了获得高可用性,您可以在多个集群、区或地域之间部署多个控制平面实例。

在具有多个共享控制面的多集群场景下,每个控制面都属于某个集群,共享控制面接受用户定义的配置(例如 `__Service__`、`__ServiceEntry__`、`__DestinationRule__` 等)来则自身集群的 `Kubernetes API Server`。因此每个主控制面集群都拥有独立的配置来源。

因此这里存在一个问题,多个控制面集群如何同步配置?

这是一个很复杂的问题,需要增加另外的同步操作。

在大型的生产环境可能需要配合 `CI/CD` 工具一起自动化该过程,实现配置同步。

![image-20221209083658951](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/multi-multi.png)

在这种多控制面模型下的在部署难度与配置复杂度上需要付出很大代价。

但是也能收获更高的回报,其核心优势在于:

- **\*\*高可用\*\***: 如果控制平面不可用,故障范围仅限于由该控制平面管理的群集中的工作负载。
- **\*\*配置隔离\*\***: 您可以在一个群集,区域或地区中进行配置更改,而不会影响其他群集。
- **\*\*控制面分区滚动发布\*\***: 能够在不同的控制面区域内,对网格控制面进行滚动发布,包括金丝雀发布网格控制面,其影响范围也只是该控制面所在的区域。

### ## 多云环境下的服务发现

了解了服务网格控制面架构以后,这时候我们不得不提到服务网格数据面的通讯与运行原理,理解这些原理能提高我们在实际运行服务网格遇到问题的解决能力。其中服务网格数据面通讯里面最为核心的技术便是服务发现能力。

在单集群模式下,服务网格控制面会从集群注册中心获取所有注册在集群中的服务,然后将集群注册中心获取到的服务信息聚合成一个服务端点列表,向每个边车提供。

在多群集场景下的服务发现能力更加复杂,因为集群之间存在注册中心隔离,网格为了运行多集群采用的解决方案是服务网格控制平面会观察并且记录每个集群中的注册中心的服务,

然后将集群中的 **同命名空间下** 的 **同名服务** 进行聚合，最后梳理成一张完整的 **网格服务端点列表**，然后下发给网格内的边车。因此服务才能具有发现其他集群服务端点的能力。

因此网格控制面为了能够访问其控制面的多集群的注册中心，在部署阶段需要从集群 **授权远程密钥** 给所属的控制面集群。

授权以后控制面才能对多集群进行服务发现，因此才具有后续的跨集群负载均衡能力。

**[授权远程密钥]**(<https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/multi-discovery.png>)

在多集群网格部署模型下，多集群服务默认的策略是：每个集群均衡负载。

但是在复杂且庞大的生产环境中，其实很多服务只需要在某些区域进行流量通讯，这时候就可以采用本地优先负载均衡策略（具体方式为参考 **[Istio 官方提到的 Locality Load Balancing]**(<https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/>））。

在某些场景下我们会发现，跨集群的流量负载能力并不是频繁操作，时时刻刻的跨集群负载也不是我们所期望的。

例如说我们只是需要进行蓝绿部署，其版本位于不同的集群中；

例如当某个集群的某个服务出现故障时，能够切换到备用集群的服务中。

这些场景都是偶发的。发生频率很低，但是又是刚需。

这时候我们怎么样才能通过网格实现呢？在 Istio 服务网格下有两种方式：

1. 我们可以不交换集群的 **API Server** 远程密钥，这样的话集群只能进行自身集群内的服务发现。

如果需要进行跨集群流量负载，可以通过 **ServiceEntry** 配合外部负载器的方式实现。

**[多副本恢复]**(<https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/multi-dup-discovery.png>)

2. 通过配置 **VirtualService** 与 **DestinationRule** 策略，禁用多集群之间的流量负载。

## ## 服务网格技术方案

上述几个章节我们简单围绕服务网格三个核心维度（网络模型、控制面架构、数据面服务发现）分析其相关原理与优势。

但是这时候我们会发现在网络模型和控制面架构部分介绍时都存在多种方案的情形，那么我们如何选择具体那种方案呢？

DaoCloud 深耕在服务网格产品多年的，拥有实际落地客户，以及从单集群演变到多集群客户解决方案的经验，我们产品内部也经过了多种方案的验证与实践，采集多位客户的场景，最后将我们的多集群服务网格产品的架构选型方案如下：自由的网络模型 + 多集群单控制面。

### ### 网络模型

在选择网络模型落地时，其实并不是非 A 即 B。

比如说用户需要更高效果，其打通部分集群之间网络成本在用户容忍范围内，用户可以选择部分集群网络为单一网络模式，在无法打通集群网络的集群中选择多网络模式。

每种模型都有自己独特的优势与代价，那么我能不能同时拥有呢？

这个答案是肯定的，我们都要。所以我们提出一个全能网络模型的概念，用户可以根据自己的需要选择网络模型：

- 单一网络
- 多网络
- 单一网络 + 多网络

这样丰富的网络模型方案，让我们能够更加贴合用户需求解决实际问题。

### ### 多集群单控制面

到这一部分，很多人会提问为什么选择多集群单控制面模式呢？从上述文章看起来多集群多控制面提供的能力不是更加丰富吗？

这里不得不提到实现多集群多控制面在落地的时候存在一个非常痛苦的点：多控制面从部署到运用到维护的成本非常高。

我们怎么得出这个结论的呢？在早期我们内部选择了多集群多控制面的验证，但是在这个过程中发现多控制面存在一些弊端：

- 策略复杂高隐患：
  - 配置复杂：不同的控制面需要不同的策略，虽然换个角度看是独立的控制面策略，但是在多集群场景下，集群肯定是简单的一个两个，应用的总数也会随着集群的扩大变得更加复杂，非常容易发生部分集群忘记配置，或者更改配置时需要重复配置多套相同的策略。
  - 策略冲突：多套控制面的策略非常多，用户需要准确掌握每个策略，不然很容易造成策略冲突。
- 控制面部署与升级维护成本高：当存在多套控制面时，我们需要维护多套控制面的部署与升级。
- 污染多个控制面集群：由于网络需要聚会多集群命名空间与服务，会污染控制面集群。让我举例说明：当集群 A 拥有命名空间 N1、N2；集群 B 拥有命名空间 N3 时；且集群 A、B 都控制面集群时，集群 A 需要新增 N3，集群 B 需要新增 N1、N2。当多集群与集群中业务到达一定体量时，这是非常恐怖的。

但是在多集群单控制面模式下能够大部分避免上面存在的问题，尤其是其中治理策略复杂这块，所有的策略将会统一在一个集群中，因此其中配置难度与管理难度将大大降低。

### ### 服务网格架构

接下来让我们来了解一下服务网格的实际架构：

![托管网格](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/mspider-hosted-mesh.png)

在原生多集群单控制面下其实并不能解决上文提到的污染多个控制面集群问题，但是我们仔细看上面的架构，大家会发现在这部分架构与社区提供的方案有一个很大的不同，在控制面集群中多了一套服务网格组件，这一套组件的最大的特色是提供了一个虚拟集群，该集群与网格控制面连接，并且目的是将网格资源与控制面集群的资源进行隔离，避免产生脏资源，这时候就能完美解决治理资源污染集群的问题，并且避免用户误操作删除控制面策略资源。

总结一下服务网格技术选型方案的优势：

- **更简单的管理**：将整个服务网格的管理集中在一个控制平面中，可以帮助组织更方便、更快速地管理整个服务网格。
- **低复杂度的配置与更好的性能**：通过将整个服务网格的所有数据都统一在一个控制平面中，可以更快速地处理服务的路由和治理策略，从而提高整个服务网格的性能。
- **更强的安全性**：在单控制面多集群模式中，我们可以更好地控制对整个服务网格的访问，并使用更严格的安全策略来保护其数据。
- **支持跨集群负载与容灾能力**：能够灵活的配置跨集群流量负载与容灾策略。
- **策略配置统一高效**：不需要配置复杂的控制面配置同步与合并问题，尤其是控制面越多，其业务服务的体量越大，其同步和合并一定会是一个非常复杂且存在风险的存在。
- **灵活的控制面与数据面隔离**：服务网格允许用户将网格控制面与数据面集群进行隔离，并且在用户资源有限的情况下也支持集群同时兼具控制面与数据面在同一集群下。
- **网络模式灵活切换**：服务网格默认场景是多集群位于单网络，但是多网络模式也能够支持部署。
- **网格资源隔离**：在开源模式下，控制面集群需要管理所有集群的网格策略，这时候会存在污染控制面集群的命名空间，或者部分网格资源会受到集群本身操作影响（例如某个命名空间被删除）。

## ## 参考

本文部分图片来自 [Istio 官网](https://istio.io/latest/docs/)与 [DaoCloud 文档网站](https://docs.daocloud.io/)。

参阅 [Istio 部署模型](https://istio.io/latest/docs/ops/deployment/deployment-models/)。

## # 多集群灰度发布

多集群灰度发布的实现逻辑是，通过在不同集群部署不同的业务应用，然后通过服务网格配置相应的策略实现业务版本之间流量调整，然后根据运行情况下线版本。

前置准备，参考网格多云部署文档，搭建网格基础设施。

## 创建 demo 应用并且验证流量

### 集群部署 v1 版本 helloworld

首先选择一个命名空间（gray-demo），并且在网格页面开启该命名空间边车注入。

在应用工作台部署应用，在这里我们以 istio 的 helloworld 为例。

![部署应用](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/create-demo.png)

选择对应集群（mdemo-cluster2）与命名空间，并且配置工作负载基本信息。

![配置工作负载](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/create-demo1.png)

- 选择镜像：docker.m.daocloud.io/istio/examples-helloworld-v1
- 版本：latest

配置服务信息：

- 访问类型：集群内访问
- 端口配置
  - 协议：TCP
  - 名称：http-5000
  - 容器端口：5000
  - 服务端口：5000

![访问类型](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/create-demo2.png)

![端口配置](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/create-demo4.png)

为了区分不同版本的工作负载，需要在 \_\_容器管理平台\_\_ 找到对应的工作负载，点击 \_\_标签与注解\_\_，给容器组标签添加键值对：

"version": "v1"

![标签与注解](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/add-labels.png)



![添加键值对](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/add-labels1.png)

### 集群部署 v2 版本 helloworld

> 注意服务命名与命名空间必须相同。

首先选择上面一致命名空间（gray-demo），并且在网格页面开启该命名空间边车注入。

部署流程与上面一样，其主要区别是：

- 镜像发生了变化：`docker.m.daocloud.io/istio/examples-helloworld-v2`
- 在容器平台给相应 **容器组标签** 加上 label "version": "v2"

## 部署灰度应用策略

### 多集群目标规则

首先创建 DestinationRule，通过定义 SubSet 定义不同集群的业务版本。

其标签键值对为上文中添加的容器组标签：\_\_version: <VERSION>\_\_。

策略：必须开启 Istio 双向 TLS

![开启双向 TLS](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/demo-dr.png)

![开启双向 TLS](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/demo-dr1.png)

必须开启 **Istio 双向** TLS 模式

![开启双向 TLS](https://docs.daocloud.io/daocloud-docs-images/docs/mspider/user-guide/multicluster/images/demo-dr2.png)

目标规则 YAML 如下：

```
```yaml
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: helloworld
  namespace: gray-demo
spec:
```

```

host: helloworld
subsets:
  - labels:
      version: v1
      name: v1
      trafficPolicy:
        tls:
          mode: ISTIO_MUTUAL
  - labels:
      version: v2
      name: v2
      trafficPolicy:
        tls:
          mode: ISTIO_MUTUAL
trafficPolicy:
  tls:
    mode: ISTIO_MUTUAL

```

## 通过 ingress 暴露服务

首先需要创建一条网关规则：

创建一条网关规则

创建一条网关规则

网关规则 YAML 如下：

```

apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: gray-demo-gateway
  namespace: gray-demo
spec:
  selector:
    istio: ingressgateway # (I)!
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "*"

```

1. use istio default controller

然后配置访问服务所需的虚拟服务规则。

配置虚拟服务规则

配置虚拟服务规则

配置虚拟服务规则

配置虚拟服务规则

虚拟服务 YAML 如下：

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: gray-demo-vs
  namespace: gray-demo
spec:
  gateways:
    - gray-demo-gateway
  hosts:
    - "*"
  http:
    - name: http-5000
      route:
        - destination:
            host: helloworld.gray-demo.svc.cluster.local
            port:
              number: 5000
            subset: v1
          weight: 80
        - destination:
            host: helloworld.gray-demo.svc.cluster.local
            port:
              number: 5000
            subset: v2
          weight: 20
```

## 验证

INGRESS\_LB\_IP 是指 Ingress 网格负载均衡地址，可以在容器管理平台中查看，如果没有有效的负载均衡 IP 可以通过 NodePort 方式访问。

## 验证

### 验证

(由于容器管理平台界面无法直接查看外部 IP，因此使用控制台的能力查看)

在浏览器中访问：`http://${INGRESS_LB_IP}/hello`

确认其 v1 与 v2 版本的访问比例是否与上面策略的比例为 8:2

## 确认

### 确认

# 服务网格故障排查

本文将持续统计和梳理服务网格过程可能因环境或操作不规范引起的报错，以及服务网格使用过程中遇到某些报错的问题分析、解决方案。若遇到服务网格的使用问题，请优先查看此排障手册。

## 创建网格

服务网格支持 3 种模式：

- 托管模式：控制面和数据面分开，创建一个虚拟集群用来保存托管的 Istio CRD 资源  
创建托管网格时，会在托管控制面集群创建托管控制面虚拟集群 API Server 用来保存 Istio CRD 资源；原则上一套 DCE 5.0 环境仅允许有一个托管网格。DCE 5.0 的全局服务集群含有全局管理所依赖的 Istio 相关组件，不允许使用该集群创建托管模式网格，demo 时可以使用外接网格创建。
- 专有模式：控制面和数据面部署在一个集群
- 外接模式：所属集群必须含有 Istio 相关组件

创建网格时常见的故障案例有：

- [创建网格时找不到所属集群](#)
- [创建网格时一直处于“创建中”，最终创建失败](#)
- [创建的网格异常，但无法删除网格](#)
- [托管网格纳管集群失败](#)
- [托管网格纳管集群时 istio-ingressgateway 异常](#)
- [网格空间无法正常解绑](#)
- [DCE 4.0 接入问题追踪](#)
- [命名空间边车配置与工作负载边车冲突](#)
- [托管网格多云互联异常](#)
- [边车占用大量内存](#)
- [创建网格时，集群列表存在未知集群](#)
- [托管网格 APIServer 证书过期解决办法](#)

## 其他常见问题

- [服务网格中常见的 503 报错](#)
- [如何使集群中监听 localhost 的应用被其它 Pod 访问](#)

# 创建网格时找不到所属集群

## 原因分析

主要原因在于所属集群的 **MeshCluster** 状态不是 “CLUSTER\_RUNNING”，可登录全局服务集

群查看所属集群的 MeshCluster CRD 状态。

```
kubectl get meshcluster -n mspider-system
```

造成这种问题的情况有几种：

## 情况 1

如果先前未使用该所属集群创建过网格，执行上述命令未发现该所属集群的 meshcluster CRD， 可能是因为 gsc-controller 从容器管理（kpanda）同步集群异常。

## 情况 2

针对已创建的网格移除集群，执行上述命令。发现该集群的 meshcluster 状态可能处于如下状态之一：

- MANAGED\_RECONCILING
- MANAGED\_SUCCEEDED
- MANAGED\_EVICTING
- MANAGED\_FAILED

原因可能是正在清理资源或者存在 meshconfig 未清理干净。

## 解决方案

1. 针对情况 1，只需要重启全局服务集群的 gsc-controller 即可。

```
kubectl -n mspider-system delete pod $(kubectl -n mspider-system get pod -l app=mspider-gsc-controller -o 'jsonpath={.items.metadata.name}')
```

2. 针对情况 2，由于环境可能未清理干净。请确保所属集群(控制面集群)控制面组件清理干净，否则影响下一次网格创建。

1. 删除未清理干净导致状态异常的 meshconfig

```
kubectl delete meshcluster -n mspider-system ${clustername}
```

2. 重启 gsc-controller 重新同步集群

```
kubectl delete po -n mspider-system ${gsc-coontroller-xxxxxxxxxx}
```

# 创建网格时一直处于“创建中”

## 问题描述

mcpc-ckube-remote pod 一直 **ContainerCreating** 。 mcpc-remote-kube-api-server configmap 等待很长时间没有创建。

故障截图

故障截图

## 日志

### 1. 查看 Pod 日志

```
kubectl describe pod mspider-mcpc-ckube-remote-5447c5bcfc-25t7t -n istio-system
```

Events:

Type	Reason	Age	From	Message
Normal	Scheduled	18m	default-scheduler	Successfully assigned istio-system/mspider-mcpc-ckube-remote-5447c5bcfc-25t7t to yl-cluster1
Warning	FailedMount	6m59s	kubelet	Unable to attach or mount volumes: unmounted volumes=[remote-kube-api-server], unattached volumes=[remote-kube-api-server kube-api-access-ljnsc ckube-config]: timed out waiting for the condition
Warning	FailedMount	2m23s (x5 over 16m)	kubelet	Unable to attach or mount volumes: unmounted volumes=[remote-kube-api-server], unattached volumes=[ckube-config remote-kube-api-server kube-api-access-ljnsc]: timed out waiting for the condition
Warning	FailedMount	105s (x16 over 18m)	kubelet	MountVolume.SetUp failed for volume "remote-kube-api-server" : configmap "mspider-mcpc-remote-kube-api-server" not found
Warning	FailedMount	5s (x2 over 13m)	kubelet	Unable to attach or mount volumes: unmounted volumes=[remote-kube-api-server], unattached volumes=[kube-api-access-ljnsc ckube-config remote-kube-api-server]: timed out waiting for the condition

### 2. 查看 gsc controller 日志

??? note “点击查看详细日志”

```none

```
time="2022-12-27T08:08:10Z" level=error msg="unable to get livez for cluster hosted-mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:51674->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func1()" file="mesh-cluster.go:191"
```

```
time="2022-12-27T08:08:10Z" level=error msg="cluster hosted-mesh-hosted reconcile a pi-server-healthy error: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:51674->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1212"
```

```
time="2022-12-27T08:08:13Z" level=error msg="unable to get livez for cluster hosted-mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:35842->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func1()" file="mesh-cluster.go:191"
```

```
time="2022-12-27T08:08:13Z" level=error msg="cluster hosted-mesh-hosted reconcile a pi-server-healthy error: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:35842->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1212"
```

```
time="2022-12-27T08:08:16Z" level=error msg="unable to get livez for cluster hosted-mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:35874->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func1()" file="mesh-cluster.go:191"
```

```
time="2022-12-27T08:08:16Z" level=error msg="cluster hosted-mesh-hosted reconcile a pi-server-healthy error: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:35874->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1212"
```

```
time="2022-12-27T08:08:19Z" level=error msg="unable to get livez for cluster hosted-mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:35902->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func1()" file="mesh-cluster.go:191"
```

```
time="2022-12-27T08:08:19Z" level=error msg="cluster hosted-mesh-hosted reconcile a pi-server-healthy error: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:35902->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1212"
```

```
time="2022-12-27T08:08:22Z" level=error msg="unable to get livez for cluster hosted-
```



```

mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:35940->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func1()" file="mesh-cluster.go:191"
time="2022-12-27T08:08:22Z" level=error msg="cluster hosted-mesh-hosted reconcile a pi-server-healthy error: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:35940->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1212"
time="2022-12-27T08:08:23Z" level=error msg="cluster cluster1-141 reconcile component-status error: deployment: istio-system/mspider-mcpc-mcpc-controller, error: {type: Available, reason: Deployment does not have minimum availability.;type: Progressing, reason: ReplicaSet \"mspider-mcpc-mcpc-controller-d7b76b945\" has timed out progressing.};deployment: istio-system/mspider-mcpc-ckube-remote, error: {type: Available, reason: Deployment does not have minimum availability.;type: Progressing, reason: ReplicaSet \"mspider-mcpc-ckube-remote-5447c5bfc\" has timed out progressing.}" func="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1212"
time="2022-12-27T08:08:23Z" level=info msg="get mesh hosted-mesh's address: 0.0.0.0:30527" func="hosted-apiserver-proxy.(*proxyServer).GetMeshPort()" file="proxy.go:87"
time="2022-12-27T08:08:23Z" level=error msg="cluster cluster1-141 reconcile component-status error: deployment: istio-system/mspider-mcpc-mcpc-controller, error: {type: Available, reason: Deployment does not have minimum availability.;type: Progressing, reason: ReplicaSet \"mspider-mcpc-mcpc-controller-d7b76b945\" has timed out progressing.};deployment: istio-system/mspider-mcpc-ckube-remote, error: {type: Available, reason: Deployment does not have minimum availability.;type: Progressing, reason: ReplicaSet \"mspider-mcpc-ckube-remote-5447c5bfc\" has timed out progressing.}" func="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1212"
time="2022-12-27T08:08:25Z" level=error msg="unable to get livez for cluster hosted-mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:52050->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func1()" file="mesh-cluster.go:191"
time="2022-12-27T08:08:25Z" level=error msg="cluster hosted-mesh-hosted reconcile a pi-server-healthy error: client rate limiter Wait returned an error: context deadline exceeded - error from a previous attempt: read tcp 192.188.110.245:52050->10.105.106.29:6443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1212"
1.672128505418613e+09 ERROR Reconciler error {"controller": "meshcluster", "controllerGroup": "discovery.mspider.io", "controllerKind": "MeshCluster", "MeshCluster": {"name": "hosted-mesh-hosted", "namespace": "mspider-system"}, "namespace": "mspider-system", "name": "hosted-mesh-hosted", "reconcileID": "e3583462-c271-4e58-ae00-18ba923362b8", "error": "Operation cannot be fulfilled on meshclusters.discovery.mspider.io \"hosted-mesh-hosted\": the object has been modified; please apply your changes to the latest version and try again"}

```

```

sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).reconcileHandler
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:326
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).processNextWorkItem
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:273
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).Start.func2.2
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:234
time="2022-12-27T08:08:25Z" level=error msg="update cluster status error: Operation
cannot be fulfilled on meshclusters.discovery.mspider.io \"hosted-mesh-hosted\": the obje
ct has been modified; please apply your changes to the latest version and try again" f
unc="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1241"
time="2022-12-27T08:08:28Z" level=error msg="unable to get livez for cluster hosted-
mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - err
or from a previous attempt: EOF" func="mesh-cluster.(*Reconciler).checkAPIServerHealt
hy.func1()" file="mesh-cluster.go:191"
time="2022-12-27T08:08:28Z" level=error msg="cluster hosted-mesh-hosted reconcile a
pi-server-healthy error: client rate limiter Wait returned an error: context deadline exce
eded - error from a previous attempt: EOF" func="mesh-cluster.(*Reconciler).Reconcile()
" file="mesh-cluster.go:1212"
time="2022-12-27T08:08:31Z" level=error msg="unable to get livez for cluster hosted-
mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - err
or from a previous attempt: read tcp 192.188.110.245:52104->10.105.106.29:6443: read:
connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func
1()" file="mesh-cluster.go:191"
time="2022-12-27T08:08:31Z" level=error msg="cluster hosted-mesh-hosted reconcile a
pi-server-healthy error: client rate limiter Wait returned an error: context deadline exce
eded - error from a previous attempt: read tcp 192.188.110.245:52104->10.105.106.29:6
443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="
mesh-cluster.go:1212"
1.6721285114885132e+09 ERROR Reconciler error {"controller": "meshclu
ster", "controllerGroup": "discovery.mspider.io", "controllerKind": "MeshCluster", "Mesh
Cluster": {"name": "hosted-mesh-hosted", "namespace": "mspider-system"}, "namespace": "m
spider-system", "name": "hosted-mesh-hosted", "reconcileID": "b3abe28d-8ec4-4a69-8ecb-
1e69e2f8c12d", "error": "Operation cannot be fulfilled on meshclusters.discovery.mspider.
io \"hosted-mesh-hosted\": the object has been modified; please apply your changes to
the latest version and try again"}
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).reconcileHandler
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:326
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).processNextWorkItem
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:273

```

```

sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).Start.func2.2
time="2022-12-27T08:08:31Z" level=error msg="update cluster status error: Operation
cannot be fulfilled on meshclusters.discovery.mspider.io \"hosted-mesh-hosted\": the obje
ct has been modified; please apply your changes to the latest version and try again" f
unc="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1241"
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:234
time="2022-12-27T08:08:34Z" level=error msg="unable to get livez for cluster hosted-
mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - err
or from a previous attempt: read tcp 192.188.110.245:49646->10.105.106.29:6443: read:
connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func
1()" file="mesh-cluster.go:191"
time="2022-12-27T08:08:34Z" level=error msg="cluster hosted-mesh-hosted reconcile a
pi-server-healthy error: client rate limiter Wait returned an error: context deadline exce
eded - error from a previous attempt: read tcp 192.188.110.245:49646->10.105.106.29:6
443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="
mesh-cluster.go:1212"
time="2022-12-27T08:08:37Z" level=error msg="unable to get livez for cluster hosted-
mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - err
or from a previous attempt: read tcp 192.188.110.245:49656->10.105.106.29:6443: read:
connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func
1()" file="mesh-cluster.go:191"
time="2022-12-27T08:08:37Z" level=error msg="cluster hosted-mesh-hosted reconcile a
pi-server-healthy error: client rate limiter Wait returned an error: context deadline exce
eded - error from a previous attempt: read tcp 192.188.110.245:49656->10.105.106.29:6
443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="
mesh-cluster.go:1212"
time="2022-12-27T08:08:37Z" level=error msg="update cluster status error: Operation
cannot be fulfilled on meshclusters.discovery.mspider.io \"hosted-mesh-hosted\": the obje
ct has been modified; please apply your changes to the latest version and try again" f
unc="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1241"
1.6721285175748258e+09 ERROR Reconciler error {"controller": "meshclu
ster", "controllerGroup": "discovery.mspider.io", "controllerKind": "MeshCluster", "Mesh
Cluster": {"name": "hosted-mesh-hosted", "namespace": "mspider-system"}, "namespace": "m
spider-system", "name": "hosted-mesh-hosted", "reconcileID": "4cc3c5be-dbe1-4bdc-a606-
70ff2b7bc3f2", "error": "Operation cannot be fulfilled on meshclusters.discovery.mspider.
io \"hosted-mesh-hosted\": the object has been modified; please apply your changes to
the latest version and try again"}
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).reconcileHandler
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:326
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).processNextWorkItem
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:273

```

```

sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).Start.func2.2
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:234
time="2022-12-27T08:08:39Z" level=error msg="mesh hosted-mesh reconcile remote-co
nfig error: create mesh hosted-mesh hosted ns error: an error on the server (\\"unknown
\\") has prevented the request from succeeding (post namespaces)" func="global-mesh.(*
Reconciler).Reconcile()" file="global-mesh.go:1198"
time="2022-12-27T08:08:40Z" level=error msg="unable to get livez for cluster hosted-
mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - err
or from a previous attempt: read tcp 192.188.110.245:49676->10.105.106.29:6443: read:
connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func
1()" file="mesh-cluster.go:191"
time="2022-12-27T08:08:40Z" level=error msg="cluster hosted-mesh-hosted reconcile a
pi-server-healthy error: client rate limiter Wait returned an error: context deadline exce
ded - error from a previous attempt: read tcp 192.188.110.245:49676->10.105.106.29:6
443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="
mesh-cluster.go:1212"
time="2022-12-27T08:08:40Z" level=error msg="update cluster status error: Operation
cannot be fulfilled on meshclusters.discovery.mspider.io \\"hosted-mesh-hosted\\": the obje
ct has been modified; please apply your changes to the latest version and try again" f
unc="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1241"
1.6721285206091342e+09 ERROR Reconciler error {"controller": "meshclu
ster", "controllerGroup": "discovery.mspider.io", "controllerKind": "MeshCluster", "Mesh
Cluster": {"name": "hosted-mesh-hosted", "namespace": "mspider-system"}, "namespace": "m
spider-system", "name": "hosted-mesh-hosted", "reconcileID": "31adec56-ca5b-4636-a74e-
3ce4ae8b07af", "error": "Operation cannot be fulfilled on meshclusters.discovery.mspider.
io \\"hosted-mesh-hosted\\": the object has been modified; please apply your changes to
the latest version and try again"}
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).reconcileHandler
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:326
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).processNextWorkItem
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:273
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).Start.func2.2
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:234
time="2022-12-27T08:08:43Z" level=error msg="unable to get livez for cluster hosted-
mesh-hosted: client rate limiter Wait returned an error: context deadline exceeded - err
or from a previous attempt: read tcp 192.188.110.245:47316->10.105.106.29:6443: read:
connection reset by peer" func="mesh-cluster.(*Reconciler).checkAPIServerHealthy.func
1()" file="mesh-cluster.go:191"
time="2022-12-27T08:08:43Z" level=error msg="cluster hosted-mesh-hosted reconcile a
pi-server-healthy error: client rate limiter Wait returned an error: context deadline exce

```

```

eded - error from a previous attempt: read tcp 192.188.110.245:47316->10.105.106.29:6
443: read: connection reset by peer" func="mesh-cluster.(*Reconciler).Reconcile()" file="
mesh-cluster.go:1212"
time="2022-12-27T08:08:43Z" level=error msg="update cluster status error: Operation
cannot be fulfilled on meshclusters.discovery.mspider.io \"hosted-mesh-hosted\": the obje
ct has been modified; please apply your changes to the latest version and try again" f
unc="mesh-cluster.(*Reconciler).Reconcile()" file="mesh-cluster.go:1241"
1.6721285236628819e+09 ERROR Reconciler error {"controller": "meshclu
ster", "controllerGroup": "discovery.mspider.io", "controllerKind": "MeshCluster", "Mesh
Cluster": {"name": "hosted-mesh-hosted", "namespace": "mspider-system"}, "namespace": "m
spider-system", "name": "hosted-mesh-hosted", "reconcileID": "908883a2-78a3-48f0-8a91-
4f963903ae19", "error": "Operation cannot be fulfilled on meshclusters.discovery.mspider.
io \"hosted-mesh-hosted\": the object has been modified; please apply your changes to
the latest version and try again"}
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).reconcileHandler
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:326
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).processNextWorkItem
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:273
sigs.k8s.io/controller-runtime/pkg/internal/controller.(*Controller).Start.func2.2
/go/pkg/mod/sigs.k8s.io/controller-runtime@v0.13.0/pkg/internal/controller/contro
ller.go:234
...

```

## 原因分析

1. 情况 1：托管网格由于控制面集群没有提前部署 **StorageClass** 导致无法创建高可用 ETCD。

xxxxx-etcd-0 一直 pending, etcd pvc 无法绑定 sc 导致 pvc pending, 进而导致 etcd pod 无法绑定 pvc。可以尝试以下步骤解决问题：

1. 部署 hwameistor 或者 localPath
2. 删除 istio-system 命名空间下 pending 的 pvc
3. 重启一下 xxx-etcd-0 pod, 等待即可

!!! note

使用 hwameistor 完成部署后, 每个节点必须存在一个空盘。您需要创建 LDC, 然

后检查 LSN LocalStorage\_PoolHDD。

2.情况 2：托管网格 istiod-xxxx-hosted-xxxx 组件异常

3.情况 3：mspider-mcpc-ckube-remote-xxxx 组件异常，describe 出现如下报错：

```
Normal    Scheduled    18m                                default-scheduler    Successfully assigned
istio-system/mspider-mcpc-ckube-remote-5447c5bcfc-25t7t to yl-cluster20
Warning   FailedMount  6m59s                                kubelet              Unable to attach
or mount volumes: unmounted volumes=[remote-kube-api-server], unattached volumes=
[remote-kube-api-server kube-api-access-ljnccs ckube-config]: timed out waiting for the c
ondition
Warning   FailedMount  2m23s (x5 over 16m) kubelet              Unable to attach
or mount volumes: unmounted volumes=[remote-kube-api-server], unattached volumes=
[ckube-config remote-kube-api-server kube-api-access-ljnccs]: timed out waiting for the c
ondition
Warning   FailedMount  105s (x16 over 18m) kubelet              MountVolume.Set
Up failed for volume "remote-kube-api-server" : configmap "mspider-mcpc-remote-kube-
api-server" not found
Warning   FailedMount  5s (x2 over 13m)    kubelet              Unable to attach
or mount volumes: unmounted volumes=[remote-kube-api-server], unattached volumes=[k
ube-api-access-ljnccs ckube-config remote-kube-api-server]: timed out waiting for the con
dition
```

4.情况 4：inotify watcher limit problems, remote-ckube 组件日志

panic: too many open files

## 解决方案

1.情况 1：控制面集群提前部署 sc。

2.情况 2：该组件异常可能控制面集群未部署 metalLB 导致网络不通，

istiod-xxxx-hosed-lb 无法分配 endpoint。可在 addon 中为该集群部署 metalLB。

3.情况 3：在移除原有托管网格后的环境中，再次创建托管网格的情况下，容易出现控制

面还没有及时下发导致 “mspider-mcpc-remote-kube-api-server” ConfigMap 未及时

创建。可以重启一下全局服务集群 gsc controller：

```
kubectrl -n mspider-system delete pod $(kubectrl -n mspider-system get pod -l app=mspider-
gsc-controller -o 'jsonpath={.items.metadata.name}')
```

#### 4.情况 4: 修改 fs.inotify.max\_user\_instances = 65535

## 创建的网格异常但无法删除

### 原因分析

网格处于失败状态, 无法点击网格实例。由于该网格中纳管了集群、创建了网格网关实例、或启用了边车注入, 导致移除网格时检测总是失败, 所以无法被正常删除。

### 解决方案

建议排查具体网格失败的原因并解决, 如果想要强制删除, 请执行以下操作:

#### 1. 禁用纳管集群的边车注入

##### 1. 禁用命名空间边车自动注入。

在 **容器管理** 中, 选择该集群 → **命名空间** → **修改标签** → 移除  
istio-injection: enabled 标签, 重启该命名空间下的所有 Pod。

移除标签

移除标签

##### 2. 禁用工作负载边车注入:

在 **容器管理** 中, 选择该集群 → **工作负载** → **无状态负载** → **标签与注解**  
→ 移除 sidecar.istio.io/inject: true 标签。

禁用边车注入

禁用边车注入

#### 2. 删除创建的网格网关实例。

### 3. 移除集群。

在 **容器管理** 中，选择全局服务集群，自定义资源搜索 `globalmeshes.discovery.mspider.io`。在 `mspider-system` 命名空间下选择要移除集群的网格，编辑 YAML：

编辑 yaml

编辑 yaml

### 4. 返回服务网格，删除该网格实例。

## 托管网格纳管集群失败

### 问题

- 托管网格接入新集群时 `istio-ingressgateway` 无法正常工作。

??? note “点击查看完整的错误日志”

```
```none
[root@dce88 metallb]# kubectl describe pod istio-ingressgateway-b8b597c59-7gnwr -n istio-system
Name:          istio-ingressgateway-b8b597c59-7gnwr
Namespace:     istio-system
Priority:       0
Node:          dce88/10.6.113.100
Start Time:    Fri, 24 Feb 2023 10:51:22 +0800
Labels:        app=istio-ingressgateway
               chart=gateways
               heritage=Tiller
               install.operator.istio.io/owning-resource=unknown
               istio=ingressgateway
               istio.io/rev=default
               operator.istio.io/component=IngressGateways
               pod-template-hash=b8b597c59
               release=istio
               service.istio.io/canonical-name=istio-ingressgateway
```



```

        service.istio.io/canonical-revision=latest
        sidecar.istio.io/inject=false
Annotations:  prometheus.io/path: /stats/prometheus
               prometheus.io/port: 15020
               prometheus.io/scrape: true
               sidecar.istio.io/inject: false
Status:       Running
IP:           10.244.0.17
IPs:
IP:           10.244.0.17
Controlled By: ReplicaSet/istio-ingressgateway-b8b597c59
Containers:
  istio-proxy:
    Container ID:  docker://40ef710d84433b108b58e8d286b46d6fde3f5ac2f6b7e80765d
61aafb2d269da
    Image:         10.16.10.120/release.daocloud.io/mspider/proxyv2:1.16.1-mspider
    Image ID:      docker-pullable://10.16.10.120/release.daocloud.io/mspider/proxyv2
@sha256:4ac80ff62af5ebe7ebb5cb75d1b1f6791b55b56e1b45a0547197b26d9674fd22
    Ports:         15021/TCP, 8080/TCP, 8443/TCP, 15090/TCP
    Host Ports:    0/TCP, 0/TCP, 0/TCP, 0/TCP
    Args:
      proxy
      router
      --domain
      $(POD_NAMESPACE).svc.cluster.local
      --proxyLogLevel=warning
      --proxyComponentLogLevel=misc:error
      --log_output_level=default:info
    State:         Running
      Started:      Fri, 24 Feb 2023 10:51:25 +0800
    Ready:         False
    Restart Count:  0
    Limits:
      cpu:          2
      memory:       1Gi
    Requests:
      cpu:          100m
      memory:       128Mi
    Readiness:      http-get http://:15021/healthz/ready delay=1s timeout=1s period=2s #s
uccess=1 #failure=30
    Environment:
      JWT_POLICY:    third-party-jwt
      PILOT_CERT_PROVIDER: istiod
      CA_ADDR:       istiod.istio-system.svc:15012

```

```

NODE_NAME: (v1:spec.nodeName)
POD_NAME: istio-ingressgateway-b8b597c59-7gnwr
(v1:metadata.name)
POD_NAMESPACE: istio-system (v1:metadata.namespace)
INSTANCE_IP: (v1:status.podIP)
HOST_IP: (v1:status.hostIP)
SERVICE_ACCOUNT: (v1:spec.serviceAccountName)
ISTIO_META_WORKLOAD_NAME: istio-ingressgateway
ISTIO_META_OWNER: kubernetes://apis/apps/v1/namespaces/i
stio-system/deployments/istio-ingressgateway
ISTIO_META_MESH_ID: fupan-mesh
TRUST_DOMAIN: cluster.local
ISTIO_META_UNPRIVILEGED_POD: true
ISTIO_META_DNS_AUTO_ALLOCATE: true
ISTIO_META_DNS_CAPTURE: true
ISTIO_META_CLUSTER_ID: futeest-2

```

Mounts:

```

/etc/istio/config from config-volume (rw)
/etc/istio/ingressgateway-ca-certs from ingressgateway-ca-certs (ro)
/etc/istio/ingressgateway-certs from ingressgateway-certs (ro)
/etc/istio/pod from podinfo (rw)
/etc/istio/proxy from istio-envoy (rw)
/var/lib/istio/data from istio-data (rw)
/var/run/secrets/credential-uds from credential-socket (rw)
/var/run/secrets/istio from istiod-ca-cert (rw)
/var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-hpgkx (ro)
/var/run/secrets/tokens from istio-token (ro)
/var/run/secrets/workload-spiffe-credentials from workload-certs (rw)
/var/run/secrets/workload-spiffe-uds from workload-socket (rw)

```

Conditions:

Type	Status
Initialized	True
Ready	False
ContainersReady	False
PodScheduled	True

Volumes:

workload-socket:

```

Type: EmptyDir (a temporary directory that shares a pod's lifetime)
Medium:
SizeLimit: <unset>

```

credential-socket:

```

Type: EmptyDir (a temporary directory that shares a pod's lifetime)
Medium:
SizeLimit: <unset>

```

```

workload-certs:
  Type:      EmptyDir (a temporary directory that shares a pod's lifetime)
  Medium:
  SizeLimit: <unset>
istiod-ca-cert:
  Type:      ConfigMap (a volume populated by a ConfigMap)
  Name:      istio-ca-root-cert
  Optional:  false
podinfo:
  Type:      DownwardAPI (a volume populated by information about the pod)
  Items:
    metadata.labels -> labels
    metadata.annotations -> annotations
istio-envoy:
  Type:      EmptyDir (a temporary directory that shares a pod's lifetime)
  Medium:
  SizeLimit: <unset>
istio-data:
  Type:      EmptyDir (a temporary directory that shares a pod's lifetime)
  Medium:
  SizeLimit: <unset>
istio-token:
  Type:      Projected (a volume that contains injected data from
multiple sources)
  TokenExpirationSeconds: 43200
config-volume:
  Type:      ConfigMap (a volume populated by a ConfigMap)
  Name:      istio
  Optional:  true
ingressgateway-certs:
  Type:      Secret (a volume populated by a Secret)
  SecretName: istio-ingressgateway-certs
  Optional:  true
ingressgateway-ca-certs:
  Type:      Secret (a volume populated by a Secret)
  SecretName: istio-ingressgateway-ca-certs
  Optional:  true
kube-api-access-hpgkx:
  Type:      Projected (a volume that contains injected data from
multiple sources)
  TokenExpirationSeconds: 3607
  ConfigMapName:      kube-root-ca.crt
  ConfigMapOptional:  <nil>
  DownwardAPI:        true

```

```

QoS Class:                               Burstable
Node-Selectors:                           <none>
Tolerations:                             node.kubernetes.io/not-ready:NoExecute op=Exists for
300s
   node.kubernetes.io/unreachable:NoExecute op=Exists f
or 300s
Events:
Type    Reason      Age           From          Message
----    -
Normal  Scheduled   63s          default-scheduler Successfully assigne
d istio-system/istio-ingressgateway-b8b597c59-7gnwr to dce88
Normal  Pulled      62s          kubelet       Container image "
10.16.10.120/release.daocloud.io/mspider/proxyv2:1.16.1-mspider" already present on mac
hine
Normal  Created     60s          kubelet       Created container
istio-proxy
Normal  Started     60s          kubelet       Started container i
stio-proxy
Warning Unhealthy   19s (x22 over 59s) kubelet       Readiness probe fai
led: Get "http://10.244.0.17:15021/healthz/ready": dial tcp 10.244.0.17:15021: connect: c
onnection refused
...

```

- 托管网格的控制面和数据面一起部署的情况下，istiod-remote ep ip 分配错误。

托管集群当作工作负载集群接入托管网格时，istiod-remote ep ip 分配为 metalLB IP，应

该为 PodIP (mspider-mcpc-ckube-remote-xxx)

istiod-remote

istiod-remote

## 原因分析

1. 情况一：被纳管集群没有安装 metalLB 导致集群网络不通，ingressgateway 无法正常分

配 endpoint 一直 CrashLoopBackoff

2. 情况二：istio-remote 组件 endpoint 分配错误，执行 kubectl get ep -n istio-system 查看

详情

## 解决方案

1. 情况一：addon 部署 metalLB

2. 情况二：检查

1. 控制面集群 istio-remote ep: istio- $\{\text{meshID}\}$ -hosted  $\{\text{podIP}\}$ :15012/15017

2. 工作负载集群 istio-remote ep: istiod-hosted-mesh-hosted-lb 的  $\{\text{loadBalancerIP}\}$ :15012/15017

# 托管网格纳管集群时 istio-ingressgateway 异常

## 情况分析

当托管网格纳管工作负载集群时，常常会出现 **istio-ingressgateway** 组件不健康。

如下图：

不健康

不健康

首先查看 istio-ingressgateway 日志报错信息：

```
2023-04-25T12: 18:04.657568Z    info    JWT policy is third-party-jwt
2023-04-25T12: 18:04.657573Z    info    using credential fetcher of JWT type in cluster.local trust domain
2023-04-25T12: 18:06.658680Z    info    Workload SDS socket not found. Starting Istio SDS Server
2023-04-25T12: 18:06.658716Z    info    CA Endpoint istiod.istio-system.svc:15012, provider Citadel
2023-04-25T12: 18:06.658747Z    info    Using CA istiod.istio-system.svc:15012 cert with certs: var/run/secrets/istio/root-cert.pem
2023-04-25T12: 18:06.667981Z    info    Opening status port 15020
2023-04-25T12: 18:06.694111Z    info    ads All caches have been synced up in 2.053719558s, marking server ready
```

```

2023-04-25T12: 18:06.694864Z      info    xdsproxy    Initializing with upstream address
"istiod-remote.istio-system.svc:15012" and cluster "yw55"
2023-04-25T12: 18:06.696385Z      info    sds         Starting SDS grpc server
2023-04-25T12: 18:06.696568Z      info    starting    starting Http service at 127.0.0.1:15004
2023-04-25T12:18:06.698950Z      info    Pilot SAN:  [istiod-remote.istio-system.svc]
2023-04-25T12: 18:06.705118Z      info    Starting proxy agent
2023-04-25T12: 18:06.705177Z      info    starting
2023-04-25T12:18:06.705214Z      info    Envoy command: [-c etc/istio/proxy/envoy-rev.json --
restart-epoch 0 --drain-time-s 45 --parent-shutdown-time-s 60 --local-address-ip-version v4 --file-f
lush-interval-msec 1000 --log-format %Y-%m-%dT%T.%fZ %l envoy %n %
v -l warning --component-log-level misc:error]
2023-04-25T12: 18:07.696708Z      info    cache       generated new workload certificate      l
atency=1.001557215s ttl=23h59m59.303308657s
2023-04-25T12: 18:07.696756Z      info    cache       Root cert has changed, start rotating root
cert
2023-04-25T12:18:07.696785Z      info    ads         XDS: Incremental Pushing:0 ConnectedEnd
points:0 Version:
2023-04-25T12: 18:07.696896Z      info    cache       returned workload trust anchor from cache
ttl=23h59m59.303107754s
2023-04-25T12:19:07.664759Z      warning envoy config    StreamAggregatedResources gRPC
config stream to xds-grpc closed since 40s ago: 14, connection error: desc = "transport: Error
while dialing dial tcp 10.233.48.75:15012: i/o timeout"
2023-04-25T12:19:29.530922Z      warning envoy config    StreamAggregatedResources gRPC
config stream to xds-grpc closed since 62s ago: 14, connection error: desc = "transport: Error
while dialing dial tcp 10.233.48.75:15012: i/o timeout"
2023-04-25T12:19:51.228936Z      warning envoy config    StreamAggregatedResources gRPC
config stream to xds-grpc closed since 84s ago: 14, connection error: desc = "transport: Error
while dialing dial tcp 10.233.48.75:15012: i/o timeout"
2023-04-25T12:20:11.732449Z      warning envoy config    StreamAggregatedResources gRPC
config stream to xds-grpc closed since 104s ago: 14, connection error: desc = "transport: Error
while dialing dial tcp 10.233.48.75:15012: i/o timeout"
2023-04-25T12:20:41.426914Z      warning envoy config    StreamAggregatedResources gRPC
config stream to xds-grpc closed since 134s ago: 14, connection error: desc = "transport: Error
while dialing dial tcp 10.233.48.75:15012: i/o timeout"
2023-04-25T12:21:15.199447Z      warning envoy config    StreamAggregatedResources gRPC
config stream to xds-grpc closed since 168s ago: 14, connection error: desc = "transport: Error
while dialing dial tcp 10.233.48.75:15012: i/o timeout"

```

上面报错信息显示连接 10.233.48.75: 15012 即 **istiod-remote** 的 service ip:15012 timeout !!!。

此时我们查看 **istio-system** 命名空间下 **istiod-remote** 的 endpoint 。

```
kubectl get ep -n istio-system
```

NAME	ENDPOINTS
	AGE
istio-eastwestgateway	36s
istio-ingressgateway	10m
istiod	10.233.97.220: 15012,10.233.97.220:15010,10.233.97.220:15017 + 1 more... 10m
istiod-remote	10.233.95.141: 15012,10.233.95.141:15017 10m

这里可以看出 **istio-remote** 分配的 **endpoint** 地址是 istiod-remote 10.233.95.141:15012,10.233.95.141:15017 。如下图：

timeout

timeout

!!! note

工作负载集群接入托管网格时，istiod-remote 的 endpoint 地址分配的应该是 istiod-xxxx-hosted-lb service 的 loadBalancer IP:15012，而这里却分配成控制面集群的 istiod-xxxx-hosted-xxxx Pod IP。

## 解决方案

更新网格 **基本信息** 配置中的控制面地址：

1. 登录控制面集群执行以下命令获取这个地址：

```
kubectl get svc -n istio-system istiod-ywistio-hosted-lb -o "jsonpath={.status.loadBalancer.ingress[0].ip}"
```

获取地址

获取地址

2. 点击右侧菜单，选择 **编辑基本信息** 。

基本信息

基本信息

3. 填写控制面地址。

填写地址

填写地址

4. 再次查看工作负载集群的 `istio-ingressgateway`，发现此时已经正常。

查看集群状态

查看集群状态

5. 查看 `istiod-remote endpoint` 信息也正常。

查看信息

查看信息

## 托管网格 APIServer 证书过期处理办法

### 问题现象

为了安全起见，托管网格的证书的有效期限仅为一年，我们需要定期重新生成证书以确保集群服务正常。

如果在界面发现网格状态异常，且查看控制面集群的 `hosted-apiserver` 日志，发现类似以下的信息：

```
x509: certificate has expired or is not yet valid  
MspiderHostedKubeAPICertExpiration
```

则表示证书已过期或即将过期，需要更换。

### 影响范围

证书过期不会影响业务正常运行，但是会影响策略下发、应用新建或重启等操作，需要及时更换。



## 修复方案

对于已经安装的网格，我们需要手动处理证书更新的过程。

首先，根据下面的 yaml，替换其中所有的 MESH\_ID 为网格 ID（界面上的名字，如

hosted-demo）。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: MESH_ID-hosted-apiserver-certs-renew
  namespace: istio-system
spec:
  parallelism: 1
  completions: 1
  template:
    spec:
      serviceAccountName: mspider-mcpc
      restartPolicy: Never
      volumes:
        - name: etcd-certs
          secret:
            secretName: MESH_ID-etcd-certs
        - name: kube-certs
          secret:
            secretName: MESH_ID-kube-certs
      containers:
        - name: init-certs
          image: release.daocloud.io/mspider/self-hosted-apiserver:0.0.13
          imagePullPolicy: IfNotPresent
          env:
            - name: MESH_ID
              value: MESH_ID
            - name: KUBE_CERT_SECRET
              value: MESH_ID-kube-certs
            - name: ETCD_CERT_SECRET
              value: MESH_ID-etcd-certs
            - name: KUBECONFIG_SECRET
              value: MESH_ID-apiserver-admin-kubeconfig
            - name: EXT_SANS
              value: MESH_ID-hosted-apiserver,MESH_ID-hosted-apiserver.istio-system,MESH_ID-hosted-apiserver.istio-system.svc,MESH_ID-hosted-apiserver.istio-system.svc.cluster.local
          command:
```

```

- bash
- -c
volumeMounts:
  - name: etcd-certs
    mountPath: /etc/kubernetes/pki/etcd
  - name: kube-certs
    mountPath: /etc/kubernetes/pki
args:
- |-
  set -ex
  cd /etc/kubernetes/pki
  if [ ! -f ca.crt ]; then
    echo "ca.crt not found"
    exit 1
  fi
  d=$(mktemp -d)
  cp -Lrf /etc/kubernetes/pki/* ${d}
  cd ${d}
  # renew certs
  kubeadm certs renew all --cert-dir ${d}
  files=$(for a in $(find . -maxdepth 1 -type f); do echo -n " --from-file $a "; done)

e)
  cat > /tmp/secret-patch.json <<EOF
  {"data": $(kubectrl create secret generic ${KUBE_CERT_SECRET} ${files} --dry-run -o jsonpath='{.data}')}
  EOF
  kubectrl patch secrets ${KUBE_CERT_SECRET} --type merge --patch-file /tmp/secret-patch.json

  files=$(for a in $(find etcd -maxdepth 1 -type f); do echo -n " --from-file $a "; done)

done)
  cat > /tmp/secret-patch.json <<EOF
  {"data": $(kubectrl create secret generic ${ETCD_CERT_SECRET} ${files} --dry-run -o jsonpath='{.data}')}
  EOF
  kubectrl patch secrets ${ETCD_CERT_SECRET} --type merge --patch-file /tmp/secret-patch.json

  kubeadm init phase kubeconfig admin --cert-dir ${d} --kubeconfig-dir ${d}
  cp admin.conf config
  sed -i 's#server: .*#server: https://MESH_ID-hosted-apiserver:6443#g' config
  cat > /tmp/secret-patch.json <<EOF
  {"data": $(kubectrl create secret generic ${KUBECONFIG_SECRET} --from-file config --dry-run -o jsonpath='{.data}')}

```

EOF

```
kubectl patch secrets ${KUBECONFIG_SECRET} --type merge --patch-file /tmp/secrets-patch.json
```

其次，在托管网格控制面集群（可在网格列表中查看）中，创建这个 Job，等待执行成功。

成功后需要重启 istio-system 命名空间下的 istiod、hosted-apiserver、etcd、ckube-remote 组件。

**重启这些组件不会影响业务正常服务。**

## 验证

- 界面网格状态
- 删除创建网格相关资源能够正常工作

# 跨集群互联问题

本页说明服务网格中跨集群互联相关的问题及其解决办法。

## 跨集群服务存在访问卡顿 10s 的现象

故障案例：托管网格有 2 个集群，集群均存在相同测试服务，通过入口网关访问测试服务，但会出现时不时卡顿 10s。

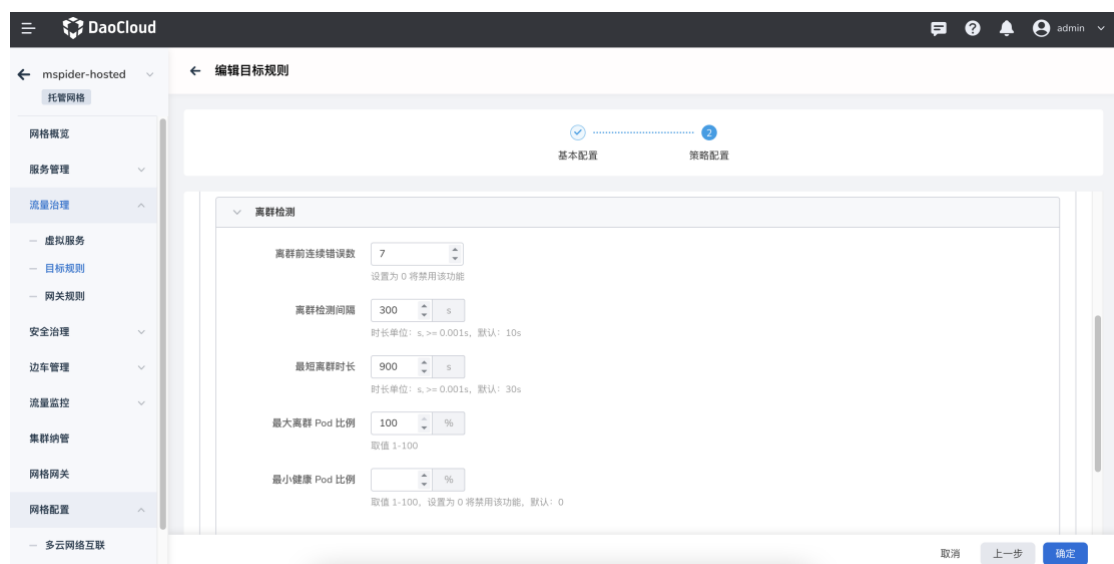
## 原因分析

1. 集群已纳管，服务已被发现，但是因为未开启多云互联，网络未打通，请求访问失败后继续访问本集群的测试服务，所以会出现一段时间卡顿
2. 集群已开启互联，互联集群创建在同一网络分组，但互联集群 Pod 间的通讯网络未打通

3. 东西网关状态异常
4. 部分集群宕机。多云互联实现了工作集群之间的网络打通，但是多云互联不会解决单个服务异常的策略，所以需要配置离群实例摘除策略。

## 解决方案

1. 开启多云互联
2. 创建多个网络分组，将集群放在不同分组，重启所有 Pod
3. 确定东西网关异常的原因，修复
4. 在目标规则中，启用离群检测策略：



### 离群检测

配置成功后，当出现集群宕机时，会自动摘除宕机集群的实例，不会出现卡顿现象。

## 网络的流量只打到部分集群上的测试服务

故障案例：托管网络有 2 个集群，已开启多云互联，成功配置后；通过入口网关持续访问测试服务，流量只打到一部分集群的测试服务。

## 原因分析

1. 部分测试服务状态异常，检查服务状态
2. 部分测试服务未注入边车，检查服务边车注入状态
3. 部分测试服务的配置不正确，检查服务配置，如 svc 的端口、端口名称等
4. 测试服务创建后才开启的多云互联

## 解决方案

1. 检查服务异常原因，让服务状态恢复正常
2. 注入边车
3. 所有测试服务的 svc 配置一致，可通过 **服务管理** -> **服务列表** 的诊断功能协助观察

服务诊断

服务诊断

4. 重启所有网关，包括自建以及数据面集群的南北以及东西网关

服务诊断

服务诊断

## 网格空间无法正常解绑

### 问题描述

- 网格类型：托管网格
- Istio 版本：0.16.1-mspider

[全局管理](#)的工作空间改变名称后，网格空间无法正常解绑； 同样在全局管理中解除空间绑

定后，服务网格依然显示绑定而且无法解绑，网格空间中被全局管理删除的空间无法解除绑定。

unbind error

unbind error

## 分析

服务网格对工作空间做了缓存，需要清除缓存中的脏数据。

## 解决办法

- 1.如果是托管集群，进入对应网格实例所部署的 `$ClusterName`，找到 `mesh-hosted-apiserver`  
`/kpanda/clusters/$ClusterName/namespaces/istio-system/pods/mesh-hosted-apiserver-0/containers`

如果是专有网格模式，直接在相应集群中操作。

- 2.使用以下命令移除对应 namespace 上的 annotation：

```
kubectl annotate ns $namespace controller.mspider.io/workspace-id- controller.mspider.io/workspace-name-
```

## DCE 4.0 接入问题追踪

本页列出一些服务网格接入 DCE 4.0 时常见的问题。

### LimitRange 问题

### 问题描述

- 1.接入 DCE 4.0 时报错。

```
message: 'unable to retrieve Pods: Unauthorized'
```

- 2.控制面集群。

提示以下报错：

```
mispider-mcpc-mcpc-controller-5bd6d54c4-df6kg CrashLoopBackOff
```

查看控制面的日志：

```
kubectl logs -n istio-system mispider-mcpc-mcpc-controller-5bd6d54c4-df6kg
time="2022-10-19T06:35:49Z" level=error msg="Unable to get kube configs of multi clusters: Get \"http://mispider-mcpc-ckube-remote/api/v1/namespaces/istio-system/configmaps/mispider-mcpc-remote-kube-api-server?resourceVersion=dsm-cluster-dce4-mispider\": dial tcp: lookup mispider-mcpc-ckube-remote: i/o timeout" func="cmholder.NewConfigHolder()" file="holder.go:52"
panic: unable to get kube configs of multi clusters: Get "http://mispider-mcpc-ckube-remote/api/v1/namespaces/istio-system/configmaps/mispider-mcpc-remote-kube-api-server?resourceVersion=dsm-cluster-dce4-mispider": dial tcp: lookup mispider-mcpc-ckube-remote: i/o timeout

goroutine 1 [running]:
main.main()
    /app/cmd/control-plane/mcpc/main.go:62 +0x694
```

3.rs: istio-operator-\*\*\* 报错：

```
message: 'pods "istio-operator-5fbbf5bbd-hf2q2" is forbidden: memory max limit to request ratio per Container is 1, but provided ratio is 2.000000'
```

## 解决办法

需要手动在 istio-operator 和 istio-system 中设置 limit range，将超配比例设置成 0。

执行以下命令查看 istio-operator 命名空间的 limit range：

```
kubectl describe limits -n istio-operator dce-default-limit-range
```

执行以下命令查看 istio-system 命名空间的 limit range：

```
kubectl describe limits -n istio-system dce-default-limit-range
```

## istiod 和 ingressgateway 一直处于 ContainerRunning 状态

原因分析：DCE 4.0 所使用的 Kubernetes 版本为 1.18，相对新版服务网格，其版本太低了。

表现形式 01：istio-managed-istio-hosted 一直无法启动，提示 istio-token 的 Configmap 不

存在。

需要手动为网格实例的 CR 中 **GlobalMesh** 添加对应的参数：

`istio.custom_params.values.global.jwtPolicy: first-party-jwt`。

`params`

`params`

!!! tip

1. DCE 4.0 在接入新版服务网格之前，需提前部署 coreDNS。
2. `__GlobalMesh__` 配置是在 DCE5 的全局服务集群，而不是在接入集群。

## 命名空间边车与工作负载边车配置冲突

### 现象描述

修改命名空间边车策略后，马上进行边车注入，Pod 不会重启生效。

### 原因分析

**Namespace** 的配置是，当前命名空间内 Sidecar 的默认边车注入策略；Pod 启用时会根据当前命名空间策略，自动进行边车注入。注入行为发生在启动节点；当 Pod 在运行时修改命名空间边车策略。

为保证生产环境的稳定性，Istio 不会自动重启 Pod，需要用户手动重启 Pod。

### 解决方案

- 需要手工重启 Pod，请根据实际业务情况谨慎操作，建议提前做好规划。
- 通过 `kubectl rollout restart deployment -n` 命令重启 Pod。



# 边车占用大量内存

造成这种情况的情况有几种：

## 情况 1

未开启边车发现范围的命名空间隔离功能。边车缓存了网格所有服务的信息，当网格集群规模较大，服务发现规则较多，就会占用大量内存。建议开启，在创建网格时或 **网格概览** -> **编辑边车** 信息里面选择。针对跨命名空间访问的情况，在 **边车管理** -> **命名空间**，对单个命名空间进行配置。

边车发现范围

边车发现范围

边车发现范围

边车发现范围

## 情况 2

经过边车的流量规模较大，请求延迟高，响应体文本大，都会占用更多的内存，可以通过监控拓扑查看确认情况。

边车发现范围

边车发现范围

## 情况 3

边车内存泄漏。通过监控组件查看，比如 DCE 5.0 组件 Insight 等。若集群规模流量未变化，边车内存占用不断上升，则可能是内存泄漏，联系我们以定位问题。

内存占用查看

内存占用查看

## 创建网格实例时，集群列表存在未知的集群

造成这种问题的原因有几种：

- **原因**：未知集群卸载不久，集群信息同步任务还未触发。

**分析**：可能是未知集群的网格实例还处于卸载中的状态，等待网格实例卸载完成。

**解决办法**：此时只需自动触发集群信息同步即可。

- **原因**：集群卸载有残留。

**解决办法**：确定未知集群已不存在，手动删除残留集群信息，登录全局服务集群

kpanda-global-cluster，执行以下命令：

```
kubecttl -n mspider-system get mc  
kubecttl -n mspider-system delete mc jy-test
```

内存占用查看

内存占用查看

## 服务网格中常见的 503 报错

本文介绍服务网格中常见的 503 报错场景及解决办法。

## 偶发 503

# 使用自定义监控指标，每当配置变更时，日志监控发现少量请求出现 503

## 问题原因

自定义指标功能的逻辑是通过生成一个对应的 EnvoyFilter 来进行 istio.stats 的配置更新。

该配置在 Envoy Listener 级别生效，即通过 LDS 同步生效。Envoy 在应用 Listener 级别的配置时，需要断开已有连接。对应的在途请求因为连接被 Reset 或 Close 导致出现 503。

## 解决办法

上游 Server 主动 Close 连接时，您看到的 503 并非上游 Server 发送，而是客户端边车因为上游连接主动断开，由本地返回的响应。

Istio 默认的重试配置中未包含“上游 Server 主动 Close 连接”的情况。EnvoyProxy 的重试条件中，Reset 符合这种情况对应的触发条件。因此，您需要为对应服务的路由配置 retry Policy。在 VirtualService 下的 retry Policy 配置包含 Reset 的触发条件。

配置示例如下。该配置仅针对 Ratings 服务生效。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: ratings-route
spec:
  hosts:
    - ratings.prod.svc.cluster.local
  http:
    - route:
        - destination:
            host: ratings.prod.svc.cluster.local
            subset: v1
```

```

retries:
  attempts: 2
  retryOn: connect-failure,refused-stream,unavailable,cancelled,retriable-status-codes,reset,50
3

```

## 相关 FAQ

### 为什么 Istio 默认的重试机制未生效？

Istio 默认重试发生的条件如下。默认会重试 2 次。该场景未在重试条件内，因此未生效。

```

"retry_policy":
{
  "retry_on": "connect-failure,refused-stream,unavailable,cancelled,retriable-status-codes",
  "num_retries": 2,
  "retry_host_predicate":
    [{ "name": "envoy.retry_host_predicates.previous_hosts" }],
  "host_selection_retry_max_attempts": "5",
  "retriable_status_codes": [503],
}

```

- **connect-failure**：连接失败。
- **refused-stream**：当 HTTP2 Stream 流返回 REFUSED\_STREAM 错误码。
- **unavailable**：当 gRPC 请求返回 unavailable 错误码。
- **cancelled**：当 gRPC 请求返回 cancelled 错误码。
- **retriable-status-codes**：当请求返回的 status\_code 和 retriable\_status\_codes 配置下定义的错误码匹配。

关于最新版本的 EnvoyProxy 完整重试条件，请参见以下文档。

- HTTP 已有的重试条件配置（包含 HTTP2 和 HTTP3）：[Router](#)
- gRPC 独有的重试条件配置：[x-envoy-retry-grpc-on](#)

## 偶发 503，没有规律，在此过程中并未发生配置变更

503 偶尔出现，但是在流量密集时，会持续出现。通常出现在边车的 Inbound 侧。

## 问题原因

Envoy 的空闲连接保持时间和应用不匹配。Envoy 空闲连接时间默认为 1 小时。

- Envoy 空闲连接时间过长，应用相对较短：

应用已经结束空闲连接，但是 Envoy 认为没有结束。此时如果有新的连接，就会报 503UC。

- Envoy 空闲连接时间过短，应用相对较长：

这种情况不会导致 503。Envoy 认为之前的连接已经被关闭，因此会直接新建一个连接。

## 解决办法

### 方案一：在 DestinationRule 中配置 idleTimeout

造成该问题的原因就是 idleTimeout 不匹配，因此在 DestinationRule 中配置 idleTimeout 属于比较根本的解决办法。

如果配置了 idleTimeout，在 Outbound 和 Inbound 两侧都会生效，即 Outbound 和 Inbound 的 Sidecar 都会存在 idleTimeout 的配置。如果客户端没有 Sidecar，idleTimeout 也会生效，并能够有效减少 503。

配置建议：此配置与业务相关，太短会导致连接数过高。建议您配置为略短于业务应用真正的 idleTimeout 时间。

### 方案二：在 VirtualService 中配置重试

重试会重建连接，可以解决此问题。具体操作，请参见[场景一](#)的解决办法。

!!! important

该操作为非幂等的请求，重试存在较大的风险，请谨慎操作。

## 边车生命周期相关

### 问题原因

边车和业务容器生命周期导致，常发生于 Pod 重启。

### 解决办法

具体操作，请参见[边车生命周期](#)。

## 必定 503

### 应用监听 localhost

#### 问题原因

当集群中的应用监听 localhost 网络地址时，如果 localhost 是本地地址，会导致集群中的其他 Pod 无法对其进行正常访问。

#### 解决办法

您可以通过对外暴露应用服务解决此问题。具体操作，请参见[如何使集群中监听 localhost 的应用被其他 Pod 访问](#)。

## 启用边车后，健康检查总是失败，报错 503

### 问题原因

在服务网格开启 mTLS 后，kubelet 向 Pod 发送的健康检查请求被边车拦截，而 kubelet 没

有对应的 TLS 证书，导致健康检查失败。

## 解决办法

您可以通过配置端口健康检查流量免于经过边车代理解决此问题。

# 如何使集群中监听 localhost 的应用被其它 Pod 访问

本文介绍如何在应用监听 localhost 的情况下，通过配置边车资源，使监听 localhost 的应用可以被集群中其它 Pod 通过 Service 访问。

## 问题现象

当部署在集群中的应用监听 localhost 时，即使通过 Service 暴露应用的服务端口，该服务也无法被集群中的其他 Pod 访问。

不同语言的应用监听 localhost 示例如下：

- Golang：net.Listen("tcp", "localhost: 8080")
- Node.js：http.createServer().listen(8080, "localhost")
- Python：socket.socket().bind(("localhost", 8083))

## 问题原因

当集群中应用监听 localhost 网络地址时，由于 localhost 是本地地址，集群中的其它 Pod 对其访问不通是正常现象。

## 解决办法

您可以任选以下方式，对外暴露应用服务。

- 方式一：修改应用监听的网络地址

如果您希望应用提供的服务对外暴露，建议修改应用代码，将应用监听的网络地址从

`localhost` 改为 `0.0.0.0`。

- 方式二：使用服务网格暴露监听 `localhost` 的服务

如果您不希望修改应用代码，同时需要将监听 `localhost` 的应用暴露给集群中的其它

Pod，可以在创建边车时进行配置。

请您按照实际情况对以下字段进行替换。

字段	说明
<code>{namespace}</code>	替换为应用部署所在的命名空间。
<code>{container_port}</code>	替换为应用监听 <code>localhost</code> 的容器端口。
<code>{port}</code>	替换为应用的 Service 端口。
<code>{key} : {value}</code>	替换为选中应用 Pod 的标签。

```
apiVersion: networking.istio.io/v1beta1
kind: Sidecar
metadata:
  name: localhost-access
  namespace: { namespace }
spec:
  ingress:
    - defaultEndpoint: "127.0.0.1:{container_port}"
      port:
        name: tcp
        number: { port }
        protocol: TCP
  workloadSelector:
    labels:
      { key }: { value }
```



# 外部服务无法访问排障

在配置服务网格的外部服务访问时，可能会遇到服务无法正常连通外部服务的情况。 为了帮助您快速定位和解决问题，以下是详细的故障排除步骤。

## 背景说明

在使用服务网格进行微服务治理时，**Sidecar** 容器会接管服务的所有入站（Inbound）和出站（Outbound）流量请求。 这意味着，当应用程序需要访问非网格纳管的集群内 API、数据库等外部服务时，流量都会经过 Sidecar 代理进行处理。

为此，服务网格提供了 **多种出口流量转发策略** 的能力，参考[如何选择出口流量策略](#)，即可以选择合适的出口流量策略。

如果在配置后，出现了服务无法正常访问外部服务的情况，可以按照以下步骤进行排查。

## 1. 检查外部服务本身是否正常

首先，确认外部服务是否正常运行。

### 验证方法：

- 在不经过服务网格的情况下，直接从集群内的其他服务或节点访问外部服务，确认其可用性。
- 使用工具如 curl、telnet 或数据库客户端直接连接外部服务，查看是否可以正常响应。

## 2. 检查未注入 Sidecar 时是否正常

接着，确认问题是否由于 Sidecar 代理引起。

### • 验证方法：

- 在网格实例的边车管理中,找到对应的工作负载,然后临时禁用 Sidecar 注入。
- 然后观察未注入 Sidecar 的服务是否可以正常访问外部服务。
- **分析:** 如果未注入 Sidecar 时,服务可以正常访问外部服务,说明问题可能出在 Sidecar 代理或者网格配置上。

### 3. 检查出口流量策略的配置

确保已经正确配置了外部流量策略,网格实例创建后默认为注册服务,仅允许访问在网格注册过的服务。

如果服务未注册到网格或者是集群外部服务,可以根据出口网络策略的管理规范进行调整。

- 的确需要使用注册服务的转发模式,可以通过创建 **服务目录 (ServiceEntry)** 通过白名单的形式开放
- 也可以调整为全部服务的转发模式,这样后续非注册到网格的服务也可以开放网络访问,减少运维成本和故障

### 4. 检查服务端口协议

确保服务端口的协议配置正确,尤其是在 ServiceEntry 和 VirtualService 中。

- **检查内容:**
  - 确认端口协议 (如 HTTP、TCP、TLS) 与外部服务实际使用的协议一致。
  - 对于使用非 HTTP 协议的服务,如 MySQL、Redis,应该使用适当的协议类型。
- **验证方法:**
  - 查看 ServiceEntry 配置,确认 ports 字段中的 protocol 是否正确。
  - 检查是否需要启用 **透传模式 (Pass-Through)**,以便绕过 Sidecar 代理。

## 5. 查看 Sidecar 日志和监控

通过查看 Sidecar 代理的日志，可以获取更多的故障信息。

- 验证方法：

- 使用 `kubectl logs` 命令查看 Sidecar 容器（通常为 `istio-proxy`）的日志：  

```
kubectl logs [pod-name] -c istio-proxy
```
- 观察是否有连接被拒绝、超时等错误信息。

## 6. 检查网络策略和防火墙

在某些情况下，集群网络策略或防火墙可能会阻止出站流量。

**验证方法：**

- 检查 Kubernetes 的 **NetworkPolicy** 配置，确认是否允许出站流量到外部服务。
- 检查云提供商或物理防火墙的配置，确保未阻止相关端口和协议。

## 7. 确认 TLS 和证书配置

如果外部服务需要 TLS 加密，需要确认证书和加密设置正确。

**验证方法：**

- 检查是否需要配置 **DestinationRule**，指定正确的 TLS 模式。
- 确认证书是否有效，是否被信任。
- 在 Sidecar 中配置适当的证书信任配置，如 `TLSContext`。

## 专业术语解释

- **ServiceEntry**：Istio 中用于将外部服务纳入服务网格控制的配置对象，允许对外部服务应用 Istio 的流量管理和策略控制。

- **External Traffic Policy:** 用于控制网格中服务对外部服务的出站流量策略，决定了服务如何处理来自外部或发送到外部的流量。
- **Sidecar:** 与应用程序容器共同驻留的代理容器，通常是 Envoy 代理。在服务网格中用于拦截和处理服务的入站和出站流量，实现负载均衡、服务发现和安全等功能。
- **Deployment/Pod:** Kubernetes 中的工作负载对象，Deployment 管理一组 Pod 的部署和生命周期，Pod 是 Kubernetes 中最小的可部署单元，包含一个或多个容器。

## 总结

通过上述故障排除步骤，您可以系统地排查服务无法访问外部服务的问题。关键在于逐步验证各个可能的原因，从外部服务的可用性、Sidecar 的影响、配置的正确性，到网络策略和安全设置，最终找到问题的根源并加以解决。

# 工作集群中 istio-init 容器启动失败

## 错误日志

```
error Command error output: xtables parameter problem: iptables-restore: unable to initialize table 'nat'
```

## 问题原因

在服务网格中，如果将网格实例 Istio 部署在一些特殊的操作系统（如 Red Hat 的某些发行版本），通常与底层操作系统和网络配置有关，造成这个问题的原因有：

- 缺失 iptables 模块
- iptables 的 nat 表需要内核模块支持。如果环境上 Red Hat 操作系统上缺少相关模块

(如 `xt_nat`、`iptables_nat`)，就会导致这个问题。

- 防火墙冲突

OpenShift 和 Istio 都依赖 `iptables` 来管理网络规则。如果系统中启用了 `firewalld` 或其他网络工具（如 `nftables`），可能与 `iptables` 发生冲突。

- `nftables` 与 `iptables` 的不兼容

Red Hat 8 和更新版本中，`iptables` 默认是通过 `nftables` 后端实现的。如果某些配置使用了传统 `iptables` 而非 `nftables`，就会导致类似问题。

## 解决方法

服务网格提供了对应的 `istio-os-init` 的 `Daemonset`，负责在每个集群的节点中激活所需要的内核模块。

!!! note

仅小部分操作系统有限制，所以这个功能在工作集群中默认不会被部署（已确认 OpenShift 因为 Red Hat 系统限制，需要手工处理）。

从全局服务管理集群中在 `istio-system` 命名空间下将 `DaemonSet` 资源 `istio-os-init` 复制并同样部署到 `istio-system`。

```
kind: DaemonSet
apiVersion: apps/v1
metadata:
  name: istio-os-init
  namespace: istio-system
  labels:
    app: istio-os-init
spec:
  selector:
    matchLabels:
      app: istio-os-init
  template:
    metadata:
      labels:
        app: istio-os-init
    spec:
```

```
volumes:
  - name: host
    hostPath:
      path: /
      type: ""
initContainers:
  - name: fix-modprobe
    image: docker.m.daocloud.io/istio/proxyv2:1.16.1
    command:
      - chroot
    args:
      - /host
      - sh
      - "-c"
      - >-
        set -ex

        # Ensure istio required modprobe

        modprobe -v -a --first-time nf_nat xt_REDIRECT xt_conntrack
        xt_owner xt_tcpudp iptable_nat iptable_mangle iptable_raw || echo
        "Istio required basic modprobe done"

        # Load Ipv6 mods # TODO for config for ipv6

        modprobe -v -a --first-time ip6table_nat ip6table_mangle
        ip6table_raw || echo "Istio required ipv6 modprobe done"

        # Load TPROXY mods # TODO for config for TPROXY

        # modprobe -v -a --first-time xt_connmark xt_mark || echo "Istio
        required TPROXY modprobe done"
resources:
  limits:
    cpu: 100m
    memory: 50Mi
  requests:
    cpu: 10m
    memory: 10Mi
volumeMounts:
  - name: host
    mountPath: /host
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
```

```
imagePullPolicy: IfNotPresent
securityContext:
  privileged: true
containers:
- name: sleep
  image: docker.m.daocloud.io/istio/proxyv2:1.16.1
  command:
    - sleep
    - 100000d
  resources:
    limits:
      cpu: 100m
      memory: 50Mi
    requests:
      cpu: 10m
      memory: 10Mi
  terminationMessagePath: /dev/termination-log
  terminationMessagePolicy: File
  imagePullPolicy: IfNotPresent
restartPolicy: Always
terminationGracePeriodSeconds: 30
dnsPolicy: ClusterFirst
securityContext: {}
schedulerName: default-scheduler
updateStrategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 0
revisionHistoryLimit: 10
```

也可以直接复制上面的 YAML 部署到网格实例所在的集群中。

## 使用网格完成定向服务访问限制

在业务场景中，我们需要限制只允许服务来访问某些特定的服务，这里可以组合 Istio 能力来进行统一的管理。

在 Istio 中，我们可以使用 **Egress** 来控制服务对外访问的流量，同时也可以使用 **Service Entry** 来控制网格外服务，结合授权策略 (Authorized Policy) 来控制服务对外访问的权限。

本文将介绍如何使用 Egress 和授权策略来控制服务对外访问的流量和权限。

## 准备工作

首先，您需要确保你的网格属于正常的状态，如果您还没有安装 Istio，请参考[创建网格](#)。

## 网格启用仅出口流量

配置网格启用仅出口流量，请修改网格的治理信息，查看下方的截图介绍，注意修改了之后，

我们的服务对集群外的访问需要配合 **Service Entry** 来使用。



## 创建一个 Egress 网关



## 准备测试应用

可以使用任意应用进行测试，在后续步骤中，我们会通过 `kubect exec pod` 进入 pod 内进行网络访问测试，建议至少保证应用内有 `curl` 命令即可。

这里是用了一个简单的 **bookinfo** 来做示例，您也可以使用其他应用。

另外，需要保证应用的 **Pod** 被成功注入了 **sidecar**，这个可以在网格的界面中查看对应服务的状态。





## 规则配置

下方规则均给出示例展示，您可以在服务网格的界面中进行创建，用 **YAML** 的方式展示，方便于理解资源的定义。

### 创建 Service Entry

这里我们首先创建一个允许的出口访问地址，这里我们使用了 **baidu** 的地址，可以安装如图进行操作



### 创建 Virtual Service



### 创建 网关规则

注意使用 **ISTIO\_MUTAL**，这样才可以用授权策略



### 创建网关规则的 DR



### 创建 baidu 的 DR

让所有流量走 **HTTPS**



image

## 启用授权策略

image

image

## 功能测试

### 从示例应用的 Pod 中访问 baidu

可以成功看到访问的结果是正常的，这是因为我们在网格中启用了出口流量，并且限制了来源使用的服务。

image

image

### 从其他应用的 Pod 中访问 baidu

此时因为有限定来源服务，所以对于从其他服务发起的咨询，会被拒绝。

image

image

## 支持接入自定义工作负载类型

DCE 5.0 提供了增强的服务网格模块，能够部署在 Kubernetes 集群中，并且自动纳管集群内的服务，实现流量控制、服务发现、流量监控和故障熔断等功能。

DCE 5.0 服务网格默认支持注入 Deployment、DaemonSet 和 StatefulSet 类型的工作负载，可直接在工作负载页面执行 Sidecar 注入动作，将服务加入到网格中。

然而在实际的生产业务中，可能会因为集群发行版本的问题，导致出现特殊的工作负载类型，传统服务网格的能力对此类特殊工作负载类型无法支持治理能力。

DCE 5.0 服务网格提供的 Istio 版本中已对此能力进行了增强，用户仅需简单的配置即可完成对特殊工作负载类型的治理。

## 网格控制面启用自定义工作负载能力

通过标准方式，`helm upgrade` 升级的方式，为网格控制面模块增加对应的特殊工作负载类型。

### 备份参数

# 首先，备份现有的网格 Helm 参数配置  
`helm -n mspider-system get values mspider > mspider.yaml`

### 更新配置

编辑上述备份的 `mspider.yaml`，并追加自定义工作负载类型的配置，如果存在多个配置类型，可以增加多个：

```
yaml title="mspider.yaml" global:      # 以 DeploymentConfig 为例      # --- add start ---
custom_workloads:                      - localized_name: # (1)!                en-US: DeploymentConfig
zh-CN: 自定义工作负载类型              name: deploymentconfigs            path:
pod_template: .spec.template # (2)!                replicas: .spec.replicas # (3)!
status_ready_replicas: .status.availableReplicas # (4)!                resource_schema:                group:
apps.openshift.io # (5)!                kind: DeploymentConfig                resource:
deploymentconfigs                      version: v1 # (6)!      # --- add end ---      debug: true      # ...
```

1. 中英文必须要写
2. 定义工作负载 Pod 内容
3. 定义工作负载副本数
4. 定义健康的副本数
5. 自定义工作负载的 CRD 的属组

## 6. 自定义工作负载的 CRD 的版本

使用 Helm 更新 mspider :

# 添加 repo, 如果不存在的话

```
helm repo add mspider https://release.daocloud.io/chartrepo/mspider
```

```
helm repo update mspider
```

# 执行更新

```
export VERSION=$(helm list -n mspider-system | grep "mspider" | awk '{print $NF}')
```

```
helm upgrade --install --create-namespace \
```

```
- n mspider-system mspider mspider/mspider \
```

```
- -cleanup-on-fail \
```

```
- -version=$VERSION \
```

```
- -set global.imageRegistry=release.daocloud.io/mspider \
```

```
- f mspider.yaml
```

更新网格控制面的工作负载:

# 执行更新命令在 kpanda-global-cluster

```
kubectl -n mspider-system rollout restart deployment mspider-api-service mspider-ckube-remote
```

```
mspider-gsc-controller mspider-ckube mspider-work-api
```

## 为指定的网格实例添加自定义工作负载类型

在我们为网格全局控制面启动了自定义工作负载能力之后, 我们只需要在对应的网格控制面

的实例中启用对应的自定义工作负载类型即可

# 这里仍旧是 kpanda-global-cluster 操作

```
[root@ globalcluster]# kubectl -n mspider-system get globalmesh
```

NAME	MODE	OWNERCLUSTER	DEPLOYNAMESPACE	PHASE
MESHVERSION				

local	EXTERNAL	kpanda-global-cluster	istio-system	SUCCEEDED	1.16.1
-------	----------	-----------------------	--------------	-----------	--------

test-ce	HOSTED	dsm01	istio-system	SUCCEEDED	1.17.1-
---------	--------	-------	--------------	-----------	---------

mspider

# 编辑需要启用的网格实例的 CR 配置

```
[root@ globalcluster]# kubectl -n mspider-system edit globalmesh test-ce
```

```
apiVersion: discovery.mspider.io/v3alpha1
```

```
kind: GlobalMesh
```

```
metadata:
```

```
  finalizers:
```

```
    - gsc-controller
```

```
  generation: 31
```

```

    name: test-ce
    ...
spec:
  clusters:
    - dsm01
  hub: release.daocloud.io/mspider
  mode: HOSTED
  ownerCluster: dsm01
  ownerConfig:
    controlPlaneParams:
      global.high_available: "true"
      global.istio_version: 1.17.1-mspider
      ...
    controlPlaneParamsStruct: # (1)!
      # --- add start ---
      global:
        custom_workloads:
          - localized_name:
              en-US: DeploymentConfig
              zh-CN: DeploymentConfig
            name: deploymentconfigs
            path:
              pod_template: .spec.template
              replicas: .spec.replicas
              status_ready_replicas: .status.availableReplicas
            resource_schema:
              group: apps.openshift.io
              kind: DeploymentConfig
              resource: deploymentconfigs
              version: v1
            # --- end ---
      istio:
        custom_params:
          values:
            sidecarInjectorWebhook:
              injectedAnnotations:
                k8s.v1.cni.cncf.io/networks: default/istio-cni
        deployNamespace: istio-system

```

## 1. 注意找到这一行

网格实例的 CR 修改成功，注意网格控制面所在集群的控制面服务

# 这里需要在网格控制面所在的集群操作

```
[root@ meshcontrolcluster]#kubectl -n istio-system rollout restart deployment mspider-mcpc-ckub
e-remote mspider-mcpc-mcpc-controller mspider-mcpc-reg-proxy test-ce-hosted-apserver
```

## 示例应用

在 OCP 中，支持一个新的工作负载 **DeploymentConfig**，本文以此为例演示如何成功支持纳管此工作负载。

## DeploymentConfig

```
yaml title="dc-nginx.yaml" apiVersion: apps.openshift.io/v1 kind: DeploymentConfig metadata:
name: nginx-deployment-samzong spec:      replicas: 1      selector:      app:
nginx-app-samzong      template:      metadata:      labels:      app:
nginx-app-samzong      spec:      containers:      - image: nginx:latest
imagePullPolicy: Always      name: nginx-samzong      ports:      -
containerPort: 80      protocol: TCP
```

使用上面的 yaml 创建一个名为 **nginx-deployment-samzong** 的应用，然后创建关联的 svc：

```
yaml title="dc-nginx-svc.yaml" apiVersion: v1 kind: Service metadata:      name: my-service spec:
selector:      app: nginx-app-samzong      ports:      - port: 80      protocol: TCP
targetPort: 80
```

这是一个标准的 Kubernetes 服务，我们通过 **app: nginx-app-samzong** 来绑定到预先创建的

**DeploymentConfig**。

```
kubectl -n NS_NAME apply -f dc-nginx.yaml dc-nginx-svc.yaml
```

## 效果

在工作负载成功启动之后，可以在 **边车管理** 中查看工作负载。默认为未注入，我们可以手工注入。

image

image

在服务列表可以看到对应的服务，此时服务的工作负载也是正常在运行的，我们可以增加对应的策略来提供对服务的访问。

image

image

## 结语

通过上面的操作，我们可以看到，通过简单的配置，就可以支持自定义的工作负载类型。这样就可以支持更多的工作负载类型，让用户可以更加灵活地使用 DCE 5.0 服务网格。

## 优化 Sidecar 资源占用问题

在 Istio 的设计理念中，Sidecar 默认缓存了全部集群服务的信息，这导致 Sidecar 占用的资源会很高，特别是当业务 Pod 成百上千时更为严重。

graph LR

init[减少 Sidecar 资源消耗] -.70%.> ns[Namespace]

init -.20%.> vs[vs/dr/ServiceEntry]

ns --> nss[使用 NS 级别的 Sidecar]

ns --> topo[依赖 NS 之间的访问拓控制] --> health1[配合健康检查]

vs --> export[增加 exportTo] --> health2[配合健康检查]

vs --> limit[依赖服务级别的限制]

```
classDef plain fill: #ddd,stroke:#fff,stroke-width:1px,color:#000;
```

```
classDef k8s fill: #326ce5,stroke:#fff,stroke-width:1px,color:#fff;
```

```
classDef cluster fill: #fff,stroke:#bbb,stroke-width:1px,color:#326ce5;
```

```
class ns,vs cluster;
```

```
class nss,topo,health1,health2,export,limit plain;
```

```
class init k8s
```

为了缓解资源占用问题，可以通过以下设置来缓解。

## 技术方案

### 1.Namespace 级别的 Sidecar 限制

```

apiVersion: networking.istio.io/v1beta1
kind: Sidecar
metadata:
  name: restrict-access-sidecar
  namespace: namespace1
spec:
  egress:
    - hosts:
        - namespace2/*
        - namespace3/*

```

以上 YAML 限制了 namespace1 下的服务，仅可以访问到 namespace2 和 namespace3 的服务。

为此需要在集群管理的命名空间中，支持增加 NS 可访问的其他 NS 白名单。

此时会有几个问题，需要考虑到：

- 启用白名单机制，是否有全局开关
- 修改白名单是否会导致 Sidecar 更新，配置更新是否需要重启
- 如果在应用访问出现问题时，是否有对应的健康检查提示
- 增加类似 NS-group 的方式（从工作空间读取）自动完成 Sidecar NS 的双向互访配置（全局开关）

## 2. 在 Istio 资源之上增加对应的 exportTo

通过 NS 下 Sidecar 访问控制，实现还是 NS 级别的限制；如果需要缩减资源消耗，可以在 Istio 资源中增加对应的 **exportTo** 的配置，声明该资源可以在哪些命名空间下访问。

这样的方式会带来较高的配置成本；如果要做，一定要考虑对应的批量配置的功能：

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my-virtual-service
  namespace: my-namespace
spec:

```



```
exportTo:
- "namespace1"
- "namespace2"
hosts:
- "*"
http:
- route:
  - destination:
    host: my-service
```

## 实际演示

- 准备好 2 个服务，在不同的命名空间： NS-a、NS-b
- 确保 2 个服务都成功注入 Sidecar
- 创建一个 Sidecar 资源，YAML 内容参考如下

sidecar

sidecar

apiVersion: networking.istio.io/v1beta1

kind: Sidecar

metadata:

name: restrict-access-sidecar

namespace: default # (1)!

spec:

egress:

- hosts:

- webstore-demo/\* # (2)!

1. current namespace

2. allow current namespace request this namespace service

yaml

yaml

- 访问效果

访问效果

访问效果

# 基于多云互联网格同名服务访问及流量的精准控制案例

本文中介绍的均为基于客户的真实业务应用演化的案例分析

开启多云互联的网格基础环境后业务访问流量不跨集群的需求

## 环境

- 集群：A、B
- 命名空间：NS1、NS2
- 服务：svcA, svcB ... （若干服务）

## 背景

在 A、B 两个集群的 NS1 和 NS2 中，都存在若干相同名称的服务（svcA, svcB ...），其他

服务都调用 svcA，如下图：

若干服务

若干服务

两个集群中相同名称的服务并不是同一业务逻辑，比如：集群 A 中 NS1 中的 svcA 和集群 B 中 NS1 中 svcA，不是同一个业务服务，只是名称和为止对等，业务逻辑不通，其他服务亦然。

## 场景 1：正常双活灾备

如果集群 A 和集群 B 中的服务都注入 Istio Sidecar，如下：

注入 sidecar

注入 sidecar

在经过其他特殊配置的情况下，两个集群的流量互通，相关服务互为灾备方案。举个例子，集群 A 中的 svcB 可以访问到集群 A 中的 svcA，也可以访问到集群 B 中的 svcA。

**结论：这是典型的利用网格多云能力的灾备场景，但是在本案例中，由于两个集群中对等服务的业务逻辑不通，就会产生业务问题。**

## 场景 2：通过边车限制跨级群通讯

其中一个集群的服务注入 Istio Sidecar，另一个集群的服务不注入 Istio Sidecar，如下：

不注入 sidecar

不注入 sidecar

这种情况下无需额外配置，两个集群的流量不会互通，比如集群 B 中的 svcB 只会访问到集群 B 中的 svcA，但是不会访问到集群 A 中的 svcA，通过 Istio 官方 Demo 的模拟结果也可以证实：

demo

demo

而集群 A 中的服务没有注入 Sidecar，所以也无法通过 Istio 能力访问到集群 B 中的服务。

反之亦然，**结论：简单的说就是一个集群的服务注入 Sidecar，另一个集群的服务不注入 Sidecar，默认情况下不会产生跨集群流量，在本案例中由于两个集群对等服务的业务逻辑不通，这样的结果符合业务需求。**

## 场景 3：通过流量规则限制跨级群通讯

如果集群 A 和集群 B 中的服务都注入 Istio Sidecar，但是还需要双方独立流量不进行跨级

群通讯，比如网格环境中有些业务需要跨级群灾备，由于资源有限，有些业务应用也要部署在这样的基础多云互联环境下，但是不希望这些应用跨级群通讯。

首先答案是可行，需要特殊配置路由规则，限制远程集群的流量：

限制流量

限制流量

规则生效情况下，服务只会访问到本机群的上游服务：

访问上游服务

访问上游服务

即使本机群的上游服务不可达，也不会访问到远程集群：

上游服务不可达

上游服务不可达

**结论：通过 DR 精准限制流量也可以在完全注入 Sidecar 的情况下限制跨级群访问，所以在本案例中也符合业务预期，可以精准控制某些业务流量的跨级群通讯，从而保证业务逻辑正常。**

## 如何选择出口流量策略

随着服务网格（Service Mesh）在微服务架构中的广泛应用，如何有效管理和治理外部服务的访问成为了一个关键问题。本指南将介绍如何在 Istio 环境中，如何管理网格内服务对外部服务的访问。

!!! note

外部服务：一般指服务网格外的服务、未注入边车的集群内服务、集群外服务。

## 前言

在服务网格中，出口流量策略支持了两种转发模式：

- 注册服务：边车仅转发目标为注册服务或服务条目中注册服务的出站流量（默认）
- 所有服务：边车转发所有出站流量

当转发模式配置为注册服务时，可以配合 Egress 实现精细化的管理。

## 接合 Egress 精细化管理

**Egress 流量** 是指从服务网格内部发出的，前往外部服务的网络流量。在 Istio 中，通过配置 **Egress 策略**，可以对这些出站流量进行控制和管理，包括哪些服务可以访问外部资源，如何访问，以及访问哪些具体的外部服务。

通过精细化的 Egress 流量治理策略，运维人员可以有效地管理网格内服务对外部服务的访问情况，提升安全性，并减少不必要的网络开销。

使用示例：[使用网格完成定向服务访问限制](#)

## 开放所有服务

在复杂的网络结构中，集群内的服务往往需要访问各种外部服务，如第三方 API、数据库、消息队列等。倘若对出站网络策略设定过于严格，可能会增加排障和运维的成本。服务在访问外部资源时，可能需要进行额外的配置，或者在网络受限时发生访问失败的问题。

## 出站流量策略的建议

为了解决上述问题，建议根据实际需求，合理设置出站流量策略。

## 开放所有外部服务访问



image

如果您的应用对外部服务没有严格的访问限制，可以将网格的出站流量策略设置为 **允许访问所有外部服务**。这样，网格内注入了边车（Sidecar）的服务在需要访问外部服务时，无需进行复杂的配置，减少了运维的复杂度。

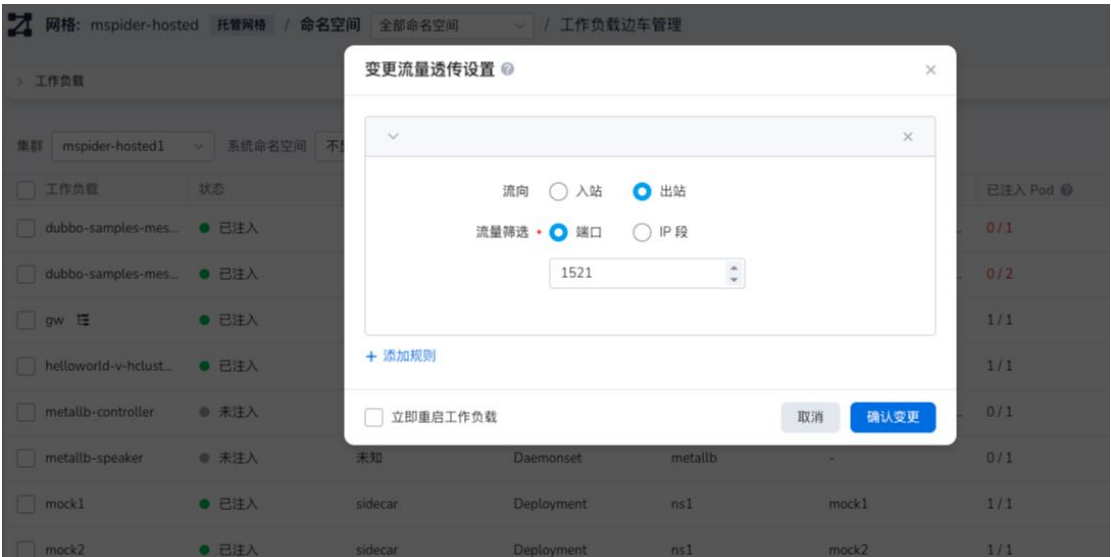
**边车（Sidecar）**：在服务网格中，边车是一种特殊的代理（通常是 Envoy 代理），与应用容器一起运行。它负责拦截和处理服务的进站和出站流量，实现服务网络的可观察性和控制能力。

## 针对特定协议的优化

对于常见的 HTTP/HTTPS 外部服务，通常开放所有服务即可满足需求。然而，对于 **MySQL**、**Redis** 等需要保持长连接的数据库服务，通过边车代理可能会引入一定的性能损耗。原因是边车代理需要解析和处理每一个请求，而长连接的协议对延迟和连接稳定性较为敏感。

为此，建议在边车治理策略中，对这些需要长连接的出站流量启用 **透传（Pass-Through）** 模式，直接绕过边车代理，减少性能影响。

**透传（Pass-Through）模式**：在 Istio 中，透传模式允许某些流量直接从应用程序发送到目标服务，而不经边车代理的处理。这样可以降低延迟，提升性能。



image

## 总结

通过合理设置服务网络的出站流量策略，可以有效地满足服务对外部资源的访问需求，降低运维和配置的复杂度。对于没有严格外部访问限制的环境，开放所有外部服务的访问是简化运维的有效方式。而对于需要长连接的数据库服务，启用透传模式可以减少边车代理对性能的影响。

## 托管网格工作集群资源同步能力

托管网格会将其 Istio 相关资源存储在一个独立的虚拟集群中，避免对工作集群的资源进行污染以及保障资源的独立与安全性，但是这里就会引发一个问题，工作集群的相关网格资源（Istio CRD）将也会失去作用。

用户需要采用开源的一些网格组件时，往往将 Istio 资源存储在工作集群中，DCE 5.0 服务网格为此提供了一个可选的高级功能：**工作集群资源同步**

!!! note

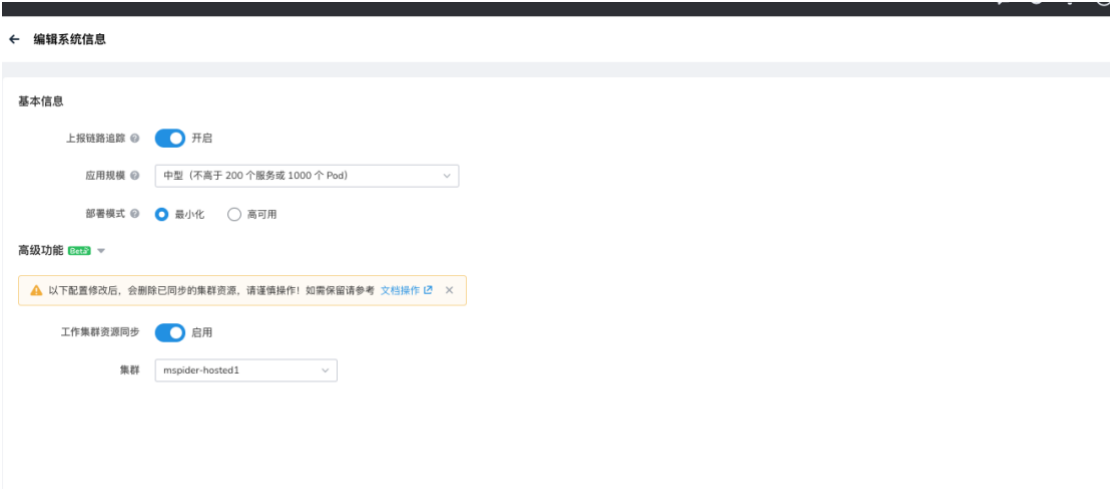
用户需要自行保证工作集群 Istio 资源的准确性，以及是否与当前网格托管资源冲突问题

如果工作集群资源与托管资源命名冲突，工作资源将不会同步

## 开启/关闭

在网格实例列表，点击某个托管网格右侧的 ... ，在弹出菜单中选择 **编辑基本信息** ，可以

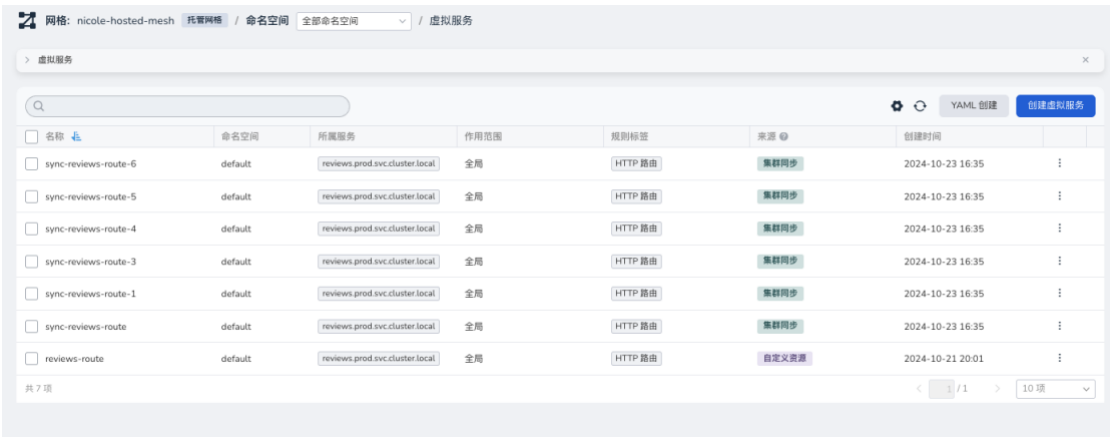
开启/关闭 **工作集群资源同步**



on/off

## 资源列表

在某个托管网格的详情页面，可以通过来源区分不同的资源：



resources



## 自动移除未管理资源

!!! warning

关闭 **\*\*工作集群资源同步\*\*** 功能或者 **\*\*更换工作集群\*\*** 时,将会 **\*\*自动删除\*\*** 集群同步的资源。

可以通过关闭自动删除能力, 或者手动修改部分资源来源, 在更改配置时不被移除。

## 关闭自动删除能力

如需关闭自动删除能力, 可以在全局服务管理集群中修改自定义资源 **GlobalMesh** 找到对应的网格, 在 YAML 中添加以下参数:

`global.enabled_resources_synchronizer_auto_remove: "false"`

```
200   routerIssuers: mesh-mspider-hosted-certs
201   spec:
202     clusters:
203       - mspider-hosted1
204       - mspider-hosted2
205     hub: release-ci.daocloud.io/mspider
206     mode: HOSTED
207     ownerCluster: mspider-hosted1
208     ownerConfig:
209       controlPlaneParams:
210         global.enabled_resources_synchronizer_auto_remove: 'false'
211         global.enabled_resources_synchronizer: 'false'
212         global.high_available: 'false'
213         global.istio_version: 1.21.1-mspider
214         global.mesh_capacity: S200P1000
```

disable

## 手动修改资源来源

如果用户想要修改资源来源, 可以在托管网格的控制面集群, 找到虚拟机集群的 API Server

实例: [meshName]-hosted-apiserver, 然后进入容器的 Shell 或控制台, 执行如下操作:

kubectl label vs [resourceName] -n [ns] mspider.io/origin-cluster- mspider.io/synced-from-

示例效果如下:

```

root@nicole-hosted-mesh-hosted-apiserver-6b96f9cb7b-lnzx8:/# kubectl get vs -n default
NAME          GATEWAYS   HOSTS   AGE
reviews-route [reviews.prod.svc.cluster.local] 44h
sync-reviews-route [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-1 [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-3 [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-4 [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-5 [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-6 [reviews.prod.svc.cluster.local] 16m
root@nicole-hosted-mesh-hosted-apiserver-6b96f9cb7b-lnzx8:/# kubectl get vs -n default sync-reviews-route-3 -oyaml
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  creationTimestamp: "2024-10-23T08:35:50Z"
  generation: 1
  labels:
    mspider.io/istio-synced-hash: 0QISNvim
    mspider.io/origin-cluster: nicole-k1-v28-a25
    mspider.io/synced-from: worker-synchronizer
  name: sync-reviews-route-3
  namespace: default
  resourceVersion: "902492"
  uid: 54f75110-24ef-4a88-a24d-768114f0cd98
spec:
  hosts:
  - reviews.prod.svc.cluster.local
  http:
  - name: reviews-route
    route:
    - destination:
        host: reviews.prod.svc.cluster.local
        port:
            number: 8080
root@nicole-hosted-mesh-hosted-apiserver-6b96f9cb7b-lnzx8:/# kubectl label vs sync-reviews-route-3 -n default mspider.io/origin-cluster- mspider.io/synced-from-
virtualservice.networking.istio.io/sync-reviews-route-3 unlabeled
root@nicole-hosted-mesh-hosted-apiserver-6b96f9cb7b-lnzx8:/#

```

### example 1

```

root@nicole-hosted-mesh-hosted-apiserver-6b96f9cb7b-lnzx8:/# kubectl get vs -n default
NAME          GATEWAYS   HOSTS   AGE
reviews-route [reviews.prod.svc.cluster.local] 44h
sync-reviews-route [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-1 [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-3 [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-4 [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-5 [reviews.prod.svc.cluster.local] 16m
sync-reviews-route-6 [reviews.prod.svc.cluster.local] 16m
root@nicole-hosted-mesh-hosted-apiserver-6b96f9cb7b-lnzx8:/# kubectl get vs -n default sync-reviews-route-3 -oyaml
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  creationTimestamp: "2024-10-23T08:35:50Z"
  generation: 1
  labels:
    mspider.io/istio-synced-hash: 0QISNvim
    mspider.io/origin-cluster: nicole-k1-v28-a25
    mspider.io/synced-from: worker-synchronizer
  name: sync-reviews-route-3
  namespace: default
  resourceVersion: "902492"
  uid: 54f75110-24ef-4a88-a24d-768114f0cd98
spec:
  hosts:
  - reviews.prod.svc.cluster.local
  http:
  - name: reviews-route
    route:
    - destination:
        host: reviews.prod.svc.cluster.local
        port:
            number: 8080
root@nicole-hosted-mesh-hosted-apiserver-6b96f9cb7b-lnzx8:/# kubectl label vs sync-reviews-route-3 -n default mspider.io/origin-cluster- mspider.io/synced-from-
virtualservice.networking.istio.io/sync-reviews-route-3 unlabeled
root@nicole-hosted-mesh-hosted-apiserver-6b96f9cb7b-lnzx8:/#

```

### example 2