网络

操作指南

配置域名

域名自定义资源（CR）示例

通过 Web 控制台创建域名

通过 CLI 创建域名

多集群共享同一域名

后续操作

相关资源

创建证书

通过 Web 控制台创建证书

配置服务

为什么需要 Ser

ClusterIP 类型

配置 Ingresses

实现方式

Ingress 示例：

通过 Web 控制台创建 Ingress

配置子网

IP 分配规则

Calico 网络

Kube-OVN 网络

子网管理

配置 MetalLB

前提条件

通过 Web 控制台配置外部 IP 地址池

通过 Web 控制台配置 BGP Peers

配置 GatewayAPI 策略

概述

前提条件

策略附加基础

策略附加汇总

在 Web 控制台创建策略

SecurityPolicy
BackendTLSPolicy
ClientTrafficPolicy
BackendTrafficPolicy
相关任务

配置 GatewayAPI Route

概述
前提条件
配置
查看
下一步

配置 ALB

ALB
Frontend

配置 NodeLocal DNSCache

Overview
Key Features
Important Notes
Installation
How It Works
Configuration

配置 CoreDNS

Overview
Configuration

实用指南

Ingress-Nginx 任务

前提条件
最大连接数
请求超时
会话亲和性（粘性会话）
头部修改
URL 重写
HSTS（HTTP 严格传输安全）
限流
WAF
转发头控制
HTTPS

Envoy Gateway 任务

概述
前提条件
高级任务
相关文档
更多配置

配置 Endpoint Health Checker

Overview
Key Features
Installation

软数据中心

前提条件
操作步骤
验证

Kube OVN

alb

保留源 IP

How It Works

How To Activate

Uninstallation

任务：从 OC

介绍

前提条件

基础 HTTP Rou

路由超时

HTTP 严格传输

基于 Cookie 的

基于路径的路由

头部修改

连接限制

速率限制

IP 允许列表/阻

URL 重写

跨命名空间路由

默认 TLS 证书

使用自定义 CA

使用自定义证书

TLS 透传

功能对比总结

迁移策略

相关文档

故障排除

如何解决 **ARM** 环境下的节点间通

查找错误原因

操作指南

配置域名

- 域名自定义资源（CR）示例
- 通过 Web 控制台创建域名
- 通过 CLI 创建域名
- 多集群共享同一域名
- 后续操作
- 相关资源

创建证书

- 通过 Web 控制台创建证书

配置服务

- 为什么需要 Ser
- ClusterIP 类型

配置 Ingresses

- 实现方式
- Ingress 示例：
- 通过 Web 控制台创建 Ingress

配置子网

- IP 分配规则
- Calico 网络
- Kube-OVN 网络
- 子网管理

配置 MetalLB

- 前提条件
- 通过 Web 控制台配置外部 IP 地址池
- 通过 Web 控制台配置 BGP Peers

配置 GatewayAPI 策略

- 概述
- 前提条件
- 策略附加基础
- 策略附加汇总
- 在 Web 控制台创建策略
- SecurityPolicy
- BackendTLSPolicy

配置 GatewayAPI Route

- 概述

ClientTrafficPolicy

BackendTrafficPolicy

相关任务

前提条件

配置

查看

下一步

配置 ALB

ALB

Frontend

配置 NodeLocal DNSCache

Overview

Key Features

Important Notes

Installation

How It Works

Configuration

配置 CoreDNS

Overview

Configuration

配置域名

向平台添加域名资源，并为集群下的所有项目或特定项目下的资源分配域名。创建域名时，支持绑定证书。

> **NOTE**
>
> 平台上创建的域名需解析到集群的负载均衡地址后，才能通过域名访问。因此，您需要确保平台上添加的域名已成功注册，并且域名解析指向集群的负载均衡地址。

在平台上成功创建并分配的域名可以用于 **Container Platform** 的以下功能：

- 创建入站规则：网络管理 > 入站规则 > 创建入站规则
- 创建原生应用：应用管理 > 原生应用 > 创建原生应用 > 添加入站规则
- 为负载均衡添加监听端口：网络管理 > 负载均衡详情 > 添加监听端口

域名绑定证书后，应用开发人员在配置负载均衡和入站规则时，只需选择该域名，即可使用该域名自带的证书支持 https。

目录

域名自定义资源（CR）示例

通过 Web 控制台创建域名

通过 CLI 创建域名

多集群共享同一域名

  通过 Web 控制台配置

  通过 CLI 配置

后续操作

相关资源

域名自定义资源（CR）示例

```yaml
# test-domain.yaml
apiVersion: crd.alauda.io/v2
kind: Domain
metadata:
  name: '${random-unique-name}'
  annotations:
    cpaas.io/secret-ref: developer.test.cn-xfd8x 1
  labels:
    cluster.cpaas.io/name: global
    project.cpaas.io/name: demo
spec:
  name: developer.test.cn
  kind: full
```

1 如果启用证书，必须提前创建 LTS 类型的 Secret，`secret-ref` 是 Secret 名称。

通过 **Web** 控制台创建域名

1. 进入 管理员。
2. 在左侧导航栏点击 网络管理 > 域名。
3. 点击 创建域名。
4. 根据以下说明配置相关参数。

参数	说明
类型	• Domain：完整域名，例如 `developer.test.cn`。 • Wildcard Domain：带有通配符（*）的泛域名，例如 `*.test.cn`，包含域名 `test.cn` 下的所有子域名。
域名	根据选择的域名类型，输入完整域名或域名后缀。
分配集群	如果分配了集群，还需选择与该集群关联的项目，如集群下的所有项目。
证书	包含用于创建域名绑定证书的公钥（tls.crt）和私钥（tls.key）。证书分配的项目应与绑定的域名相同。 注意： • 不支持二进制文件导入。 • 绑定的证书需满足格式正确、在有效期内且为域名签发等条件。 • 创建绑定证书后，绑定证书的名称格式为：域名-随机字符。 • 创建绑定证书后，可在证书列表查看，但绑定证书的更新和删除仅支持在域名详情页操作。 • 创建绑定证书后，支持更新证书内容，但不支持替换为其他证书。

5. 点击 创建。

通过 CLI 创建域名

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

多集群共享同一域名

您可以通过在 global 集群中创建多个 Domain 资源，使用相同的 `spec.name` 值但不同的 `cluster.cpaas.io/name` 标签，实现同一域名在多个集群间共享。

通过 Web 控制台配置

1. 按照通过 Web 控制台创建域名的步骤操作。
2. 创建两个具有相同域名（例如 `app.example.com`）的域名资源。
3. 对每个域名，选择不同的分配集群（例如 cluster-a 和 cluster-b）。

通过 CLI 配置

在 global 集群中创建两个 Domain 自定义资源，`spec.name` 相同，但 `cluster.cpaas.io/name` 标签不同：

> **NOTE**
>
> 两个 Domain 资源的域名（`spec.name`）必须相同，但资源的 `metadata.name` 应唯一。两个资源均创建于 global 集群。

Cluster A 的域名：

```yaml
apiVersion: crd.alauda.io/v2
kind: Domain
metadata:
  name: '${random-unique-name}'
  labels:
    cluster.cpaas.io/name: cluster-a
    project.cpaas.io/name: project-a
spec:
  name: app.example.com  # 相同域名
  kind: full
```

Cluster B 的域名：

```yaml
apiVersion: crd.alauda.io/v2
kind: Domain
metadata:
  name: '${random-unique-name}'
  labels:
    cluster.cpaas.io/name: cluster-b
    project.cpaas.io/name: project-a
spec:
  name: app.example.com  # 与 Cluster A 相同的域名
  kind: full
```

后续操作

- 域名注册：如果创建的域名尚未注册，请先完成域名注册。
- 域名解析：如果域名未指向平台集群的负载均衡地址，请进行域名解析。

相关资源

- 配置证书

创建证书

平台管理员导入 TLS 证书并分配给指定项目后，具有相应项目权限的开发人员在使用入站规则和负载均衡功能时，可以使用平台管理员导入并分配的证书。随后，在证书过期等场景下，平台管理员可以集中更新证书。

> **NOTE**
>
> 证书功能当前不支持在公有云集群中使用。您可以根据需要在指定命名空间内创建 TLS 类型的 Secret 字典。

目录

通过 Web 控制台创建证书

通过 **Web** 控制台创建证书

1. 进入 **Administrator**。
2. 在左侧导航栏中，点击 **Network Management** > **Certificates**。
3. 点击 **Create Certificate**。
4. 参照以下说明配置相关参数。

参数	描述
Assign Project	• All Projects：将证书分配用于当前集群关联的所有项目。 • Specified Project：将证书分配用于指定项目。 • No Assignment：暂不分配项目。证书创建完成后，可通过 **Update Project** 操作更新可使用该证书的项目。
Public Key	指 tls.crt。导入公钥时，不支持二进制文件。
Private Key	指 tls.key。导入私钥时，不支持二进制文件。

5. 点击 **Create**。

配置服务

在 Kubernetes 中，Service 是一种用于暴露运行在集群中一个或多个 Pod 上的网络应用的方法。

目录

- 为什么需要 Service
- ClusterIP 类型 Service 示例：
- Headless 服务
- 通过 Web 控制台创建服务
- 通过 CLI 创建服务
- 示例：集群内访问应用
- 示例：集群外访问应用
- 示例：ExternalName 类型的 Service
- LoadBalancer 类型 Service 注解
  - AWS EKS 集群
  - 华为云 CCE 集群
  - Azure AKS 集群
  - Google GKE 集群
- 示例：使用 MetalLB BGP 和 Local Traffic Policy 的 LoadBalancer
  - 优势
  - 前提条件
  - 步骤
  - 关键配置点
    - externalTrafficPolicy: Local

使用 BGP 的 LoadBalancer

部署步骤

验证

为什么需要 Service

1. Pod 有自己的 IP，但：

- Pod IP 不稳定（如果 Pod 被重新创建，IP 会变化）。
- 直接访问 Pod 变得不可靠。

2. Service 通过提供以下功能解决了这个问题：

- 稳定的 IP 和 DNS 名称。
- 自动负载均衡到匹配的 Pod。

ClusterIP 类型 Service 示例：

```yaml
# simple-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ClusterIP 1
  selector: 2
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80 3
      targetPort: 80 4
```

1 可用的 type 值及其行为包括 `ClusterIP`、`NodePort`、`LoadBalancer`、`ExternalName`

2. Service 目标的 Pod 集合通常由你定义的 selector 决定。

3. `Service` 端口。

4. 将 Service 的 `targetPort` 绑定到 Pod 的 `containerPort`。此外，也可以引用 Pod 容器下的 `port.name`。

Headless 服务

有时你不需要负载均衡和单一的 Service IP。在这种情况下，可以创建所谓的 Headless 服务：

```yaml
spec:
  clusterIP: None
```

Headless 服务适用于：

- 你想发现单个 Pod 的 IP，而不仅仅是单一的服务 IP。
- 你需要直接连接到每个 Pod（例如，像 Cassandra 或 StatefulSets 这样的数据库）。
- 你使用 StatefulSets，其中每个 Pod 必须有稳定的 DNS 名称。

通过 Web 控制台创建服务

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Network** > **Services**。
3. 点击 **Create Service**。
4. 参考以下说明配置相关参数。

参数	说明
虚拟 **IP** 地址	如果启用，将为该 Service 分配一个 ClusterIP，可用于集群内的服务发现。 如果禁用，将创建一个 Headless 服务，通常用于 **StatefulSet**。

参数	说明
类型	• **ClusterIP**：在集群内部 IP 上暴露 Service。选择此值时，Service 只能从集群内部访问。 • **NodePort**：在每个节点的 IP 上通过静态端口（NodePort）暴露 Service。 • **ExternalName**：将 Service 映射到 externalName 字段的内容（例如，主机名 api.foo.bar.example）。 • **LoadBalancer**：通过外部负载均衡器对外暴露 Service。Kubernetes 本身不直接提供负载均衡组件，你需要自行提供，或者将 Kubernetes 集群与云提供商集成。
目标组件	• **Workload**：Service 会将请求转发到特定的工作负载，匹配标签如 `project.cpaas.io/name: projectname` 和 `service.cpaas.io/name: deployment-name`。 • **Virtualization**：Service 会将请求转发到特定的虚拟机或虚拟机组。 • **Label Selector**：Service 会将请求转发到带有指定标签的某类工作负载，例如 `environment: release`。
端口	用于配置该 Service 的端口映射。以下示例中，集群内其他 Pod 可以通过虚拟 IP（如果启用）和 TCP 端口 *80* 调用该 Service；访问请求将被转发到目标组件 Pod 的外部暴露的 TCP 端口 *6379* 或 *redis*。 • 协议：Service 使用的协议，支持的协议包括：`TCP`、`UDP`、`HTTP`、`HTTP2`、`HTTPS`、`gRPC`。 • **Service** 端口：Service 在集群内暴露的端口号，即 Port，例如 *80*。 • 容器端口：Service 端口映射到的目标端口号（或名称），即 targetPort，例如 *6379* 或 *redis*。 • **Service** 端口名称：会自动生成，格式为 `<protocol>-<service port>-<container port>`，例如：*tcp-80-6379* 或 *tcp-80-redis*。

参数	说明
会话亲和性	基于源 IP 地址（ClientIP）的会话亲和性。如果启用，来自同一 IP 地址的所有访问请求在负载均衡时会保持在同一服务器上，确保同一客户端的请求被转发到同一服务器处理。

5. 点击 **Create**。

通过 CLI 创建服务

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

基于已有的 deployment 资源 `my-app` 创建服务。

```
kubectl expose deployment my-app \
  --port=80 \
  --target-port=8080 \
  --name=test-service \
  --type=NodePort \
  -n p1-1
```

示例：集群内访问应用

```yaml
# access-internal-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
```

1. 应用该 YAML：

```
kubectl apply -f access-internal-demo.yaml
```

2. 启动另一个 Pod：

```
kubectl run test-pod --rm -it --image=busybox -- /bin/sh
```

3. 在 `test-pod` Pod 中访问 `nginx-clusterip` 服务：

```
wget -qO- http://nginx-clusterip
# 或使用 Kubernetes 自动创建的 DNS 记录：<service-name>.<namespace>.svc.cluster.local
wget -qO- http://nginx-clusterip.default.svc.cluster.local
```

你应该能看到包含 “Welcome to nginx!” 字样的 HTML 响应。

示例：集群外访问应用

```yaml
# access-external-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080
```

1. 应用该 YAML：

```
kubectl apply -f access-external-demo.yaml
```

2. 查看 Pods：

```
kubectl get pods -l app=nginx -o wide
```

3. curl 访问 Service：

```
curl http://{NodeIP}:{nodePort}
```

你应该能看到包含 “Welcome to nginx!” 字样的 HTML 响应。

当然，也可以通过创建 LoadBalancer 类型的 Service 从集群外访问应用。

注意：请提前配置 LoadBalancer 服务。

```yaml
# access-external-demo-with-loadbalancer.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-lb-service
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
```

1. 应用该 YAML：

   ```
   kubectl apply -f access-external-demo-with-loadbalancer.yaml
   ```

2. 获取外部 IP 地址：

   ```
   kubectl get svc nginx-lb-service
   ```

```
NAME            TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)
AGE
nginx-service   LoadBalancer   10.0.2.57        34.122.45.100   80:3000
5/TCP    30s
```

`EXTERNAL-IP` 即为你从浏览器访问的地址。

```
curl http://34.122.45.100
```

你应该能看到包含 “Welcome to nginx!” 字样的 HTML 响应。

如果 EXTERNAL-IP 显示为 `pending`，说明 LoadBalancer 服务尚未在集群中部署。

示例：ExternalName 类型的 Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-external-service
  namespace: default
spec:
  type: ExternalName
  externalName: example.com
```

1. 应用该 YAML：

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

2. 尝试在集群内的 Pod 中解析：

```
kubectl run test-pod --rm -it --image=busybox -- sh
```

然后执行：

```
nslookup my-external-service.default.svc.cluster.local
```

你会看到它解析为 `example.com`。

LoadBalancer 类型 Service 注解

AWS EKS 集群

有关 EKS LoadBalancer Service 注解的详细说明，请参阅 Annotation Usage Documentation ↗ 。

Key	Value	说明
service.beta.kubernetes.io/aws-load-balancer-type	external: 使用官方 AWS LoadBalancer Controller。	指定 LoadBalancer 类型的控制器。 注意：请提前联系平台管理员部署 AWS LoadBalancer Controller。
service.beta.kubernetes.io/aws-load-balancer-nlb-target-type	• instance：流量通过 NodePort 发送到 Pod。 • ip：流量直接路由到 Pod（集群必须使用 Amazon VPC CNI）。	指定流量如何到达 Pod。
service.beta.kubernetes.io/aws-load-balancer-scheme	• internal：私有网络。 • internet-facing：公网网络。	指定使用私有网络还是公网网络。

Key	Value	说明
service.beta.kubernetes.io/aws-load-balancer-ip-address-type	• IPv4 • dualstack	指定支持的 IP 地址栈。

华为云 CCE 集群

有关 CCE LoadBalancer Service 注解的详细说明，请参阅 Annotation Usage Documentation ↗ 。

Key	Value
kubernetes.io/elb.id	
kubernetes.io/elb.autocreate	示例：`{"type":"public","bandwidth_name":"cce-bandwidth-1551163379627","bandwidth_chargemode":"bandwidth","bandwi` `["cn-north-4b"],"l4_flavor_name":"L4_flavor.elb.s1.small"` 注意：请先阅读 填写说明 ↗ ，并根据需要调整示例参数。
kubernetes.io/elb.subnet-id	

Key	Value
kubernetes.io/elb.class	• union：共享负载均衡。 • performance：独享负载均衡，仅支持 Kubernetes 1.17 及以上版
kubernetes.io/elb.enterpriseID	

Azure AKS 集群

有关 AKS LoadBalancer Service 注解的详细说明，请参阅 Annotation Usage Documentation ↗ 。

Key	Value	说明
service.beta.kubernetes.io/azure-load-balancer-internal	• true：私有网络。 • false：公网网络。	指定使用私有网络还是公网网络。

Google GKE 集群

有关 GKE LoadBalancer Service 注解的详细说明，请参阅 Annotation Usage Documentation ↗ 。

Key	Value	说明
networking.gke.io/load-balancer-type	Internal	指定使用私有网络。
cloud.google.com/l4-rbs	enabled	默认为公网。如果配置此参数，流量将直接路由到 Pod。

示例：使用 **MetalLB BGP** 和 **Local Traffic Policy** 的 **LoadBalancer**

本示例演示如何使用 MetalLB BGP 模式配置 LoadBalancer 类型的 Service，并设置 `externalTrafficPolicy: Local`，实现无额外网络跳转的主动-主动负载均衡。

优势

- 主动**-**主动负载均衡：流量同时分布到多个节点
- 无额外网络跳转：直接路由到 Pod，无需节点中转转发
- 更好性能：`externalTrafficPolicy: Local` 保留源 IP，降低延迟
- 高可用性：BGP 路由公告确保流量到达健康节点

前提条件

配置 LoadBalancer Service 前，请确保：

1. 已部署 **MetalLB**：参见 创建外部 IP 地址池 了解安装方法
2. 已配置 **BGP Peer**：参见 创建 BGP Peers 了解 BGP 配置
3. 外部 **IP** 地址池：配置带有 BGPAdvertisement 的 IPAddressPool

步骤

使用 `externalTrafficPolicy: Local` 部署带 LoadBalancer 的应用：

```yaml
# nginx-loadbalancer-local-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer-local
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
```

关键配置点

externalTrafficPolicy: Local

`externalTrafficPolicy: Local` 设置带来多项好处：

- **保留源 IP**：客户端源 IP 得以保留，便于日志记录和安全策略
- **直接路由 Pod**：流量直接到达 Pod，无需节点级转发

使用 BGP 的 LoadBalancer

使用 MetalLB BGP 模式时：

- 路由由 BGPAdvertisement 中 nodeSelectors 指定的节点发布
- BGP Peer 接收路由公告并进行流量路由
- BGPPeer 和 BGPAdvertisement 的节点选择器需匹配，确保路由一致性

部署步骤

1. 部署应用：

   ```
   kubectl apply -f nginx-loadbalancer-local-demo.yaml
   ```

2. 验证 **LoadBalancer Service**：

   ```
   kubectl get svc nginx-loadbalancer-local
   ```

   预期输出：

   ```
   NAME                         TYPE           CLUSTER-IP      EXTERNAL-IP    P
   ORT(S)        AGE
   nginx-loadbalancer-local LoadBalancer   10.0.2.57       4.4.4.3        8
   0:30005/TCP   30s
   ```

3. 测试服务：

   ```
   curl http://4.4.4.3
   ```

验证

- 监控服务端点：`kubectl get endpoints nginx-loadbalancer-local`

- 检查服务状态：`kubectl describe svc nginx-loadbalancer-local`

配置 Ingresses

Ingress 规则（Kubernetes Ingress）将集群外部的 HTTP/HTTPS 路由暴露到内部路由（Kubernetes Service），从而实现对计算组件外部访问的控制。

创建一个 Ingress 来管理对 Service 的外部 HTTP/HTTPS 访问。

WARNING

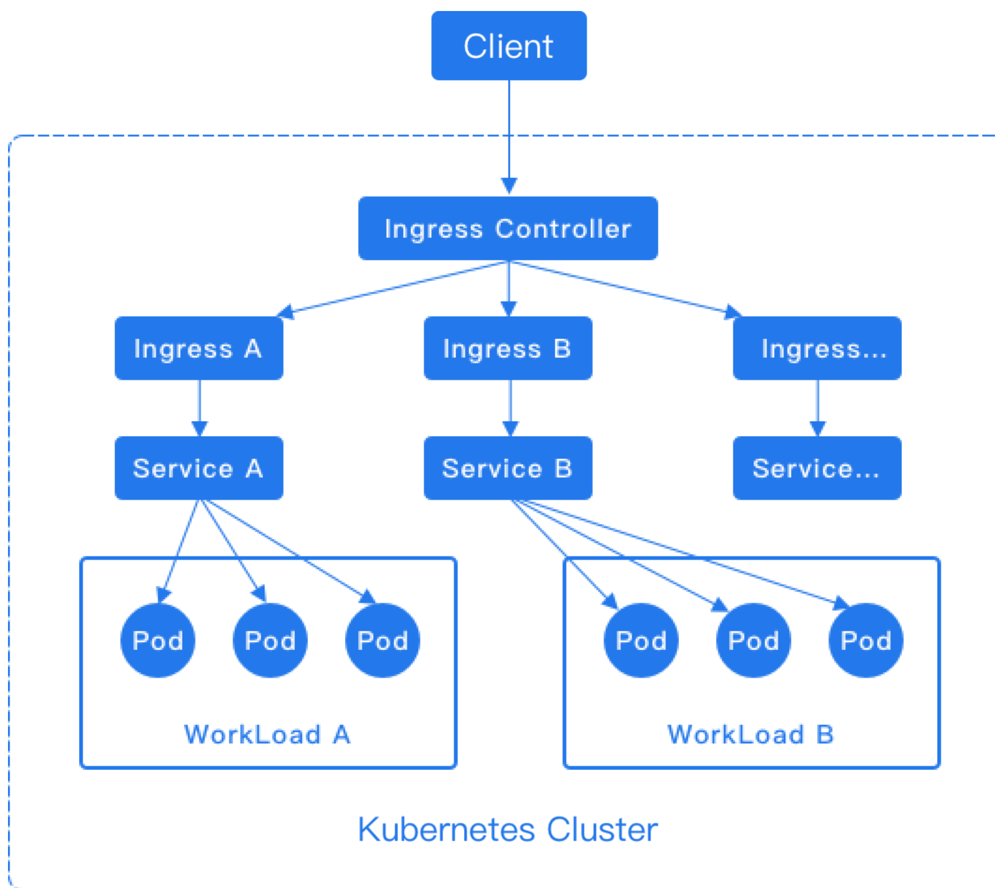
在同一命名空间内创建多个 ingress 时，不同的 ingress 不得 具有相同的 域名、协议 和 路径（即不允许重复的访问点）。

目录

- 实现方式
- Ingress 示例：
- 通过 Web 控制台创建 Ingress
- 通过 CLI 创建 Ingress

实现方式

Ingress 规则依赖于 Ingress Controller 的实现，Ingress Controller 负责监听 Ingress 和 Service 的变化。当创建新的 Ingress 后，Ingress Controller 收到请求时，会根据 Ingress 中的转发规则匹配，并将流量分发到指定的内部路由，如下图所示。



NOTE

对于 HTTP 协议，Ingress 仅支持 80 端口作为外部端口。对于 HTTPS 协议，Ingress 仅支持 443 端口作为外部端口。平台的负载均衡器会自动添加 80 和 443 监听端口。

- 通过 ingress-nginx-operator 安装 ingress-nginx 作为 ingress-controller
- 通过 alb-operator 安装 alb 作为 ingress-controller

Ingress 示例：

```yaml
# nginx-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ingress
  namespace: k-1
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: / # 1
spec:
  ingressClassName: nginx # 2
  rules:
    - host: demo.local # 3
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: nginx-service
                port:
                  number: 80
```

1. 更多配置请参考 nginx-configuration。
2. `nginx` 表示使用 `ingress-nginx` controller，`$alb_name` 表示使用 alb 作为 ingress controller。
3. 如果只想在本地运行 ingress，请提前配置 `hosts`。

通过 Web 控制台创建 Ingress

1. 访问 **Container Platform**。
2. 在左侧导航栏点击 **Network** > **Ingress**。
3. 点击 **Create Ingress**。
4. 参考以下说明配置相关参数。

参数	说明
Ingress Class	不同的 ingress controller 通过不同的 `IngressClass` 名称实现 Ingress。如果平台上存在多个 ingress controller，用户可以通过此选项选择使用哪一个。
域名	Host 可以是精确匹配（例如 `foo.bar.com`）或通配符（例如 `*.foo.com`）。可用的域名由平台管理员分配。
证书	TLS secret 或由平台管理员分配的证书。
匹配类型 和 路径	• **Prefix**：匹配路径前缀，例如 `/abcd` 可以匹配 `/abcd/efg` 或 `/abcde`。 • **Exact**：匹配精确路径，例如 `/abcd`。 • 实现特定：如果使用自定义 Ingress controller 管理 Ingress 规则，可以选择让 controller 决定。
Service	外部流量将转发到此 Service。
Service 端口	指定流量将转发到 Service 的哪个端口。

5. 点击 **Create**。

通过 CLI 创建 Ingress

```
kubectl apply -f nginx-ingress.yaml
```

配置子网

目录

- IP 分配规则
- Calico 网络
  - 约束与限制
  - Calico 网络子网示例自定义资源（CR）
  - 通过 Web 控制台创建 Calico 网络子网
  - 通过 CLI 创建 Calico 网络子网
  - 参考内容
- Kube-OVN 网络
  - Kube-OVN Overlay 网络子网示例自定义资源（CR）
  - 通过 Web 控制台创建 Kube-OVN Overlay 网络子网
  - 通过 CLI 创建 Kube-OVN Overlay 网络子网
  - Underlay 网络
  - 使用说明
  - 通过 Web 控制台添加桥接网络（可选）
  - 通过 CLI 添加桥接网络
  - 通过 Web 控制台添加 VLAN（可选）
  - 通过 CLI 添加 VLAN
  - Kube-OVN Underlay 网络子网示例自定义资源（CR）
  - 通过 Web 控制台创建 Kube-OVN Underlay 网络子网
  - 通过 CLI 创建 Kube-OVN Underlay 网络子网
  - 相关操作
- 子网管理

通过 Web 控制台更新网关

通过 CLI 更新网关

通过 Web 控制台更新保留 IP

通过 CLI 更新保留 IP

通过 Web 控制台分配项目

通过 CLI 分配项目

通过 Web 控制台分配命名空间

通过 CLI 分配命名空间

通过 Web 控制台扩展子网

通过 CLI 扩展子网

管理 Calico 网络

通过 Web 控制台删除子网

通过 CLI 删除子网

IP 分配规则

> **NOTE**
>
> 如果一个项目或命名空间被分配了多个子网，IP 地址将从其中一个子网随机选择。

- 项目分配：
  - 如果项目未绑定子网，则该项目下所有命名空间中的 Pods 只能使用默认子网的 IP 地址。如果默认子网的 IP 地址不足，Pods 将无法启动。
  - 如果项目绑定了子网，则该项目下所有命名空间中的 Pods 只能使用该特定子网的 IP 地址。
- 命名空间分配：
  - 如果命名空间未绑定子网，则该命名空间中的 Pods 只能使用默认子网的 IP 地址。如果默认子网的 IP 地址不足，Pods 将无法启动。
  - 如果命名空间绑定了子网，则该命名空间中的 Pods 只能使用该特定子网的 IP 地址。

Calico 网络

在 Calico 网络中创建子网，实现集群内资源更细粒度的网络隔离。

约束与限制

在 IPv6 集群环境中，Calico 网络中创建的子网默认使用 VXLAN 封装。VXLAN 封装所需端口与 IPIP 封装不同，需要确保 UDP 端口 4789 已开放。

Calico 网络子网示例自定义资源（CR）

```yaml
# test-calico-subnet.yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: test-calico
spec:
  cidrBlock: 10.1.1.1/24
  default: false # 1
  ipipMode: Always # 2
  natOutgoing: true # 3
  private: false
  protocol: Dual
  v4blockSize: 30
```

1. 当 `default` 为 true 时，使用 VXLAN 封装。
2. 详见封装模式参数和封装协议参数。
3. 详见出站流量 NAT 参数。

通过 Web 控制台创建 Calico 网络子网

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Network Management** > **Subnets**。
3. 点击 **Create Subnet**。
4. 按照以下说明配置相关参数。

参数	说明
CIDR	将子网分配给项目或命名空间后，该命名空间内的容器组将随机使用该 CIDR 范围内的 IP 进行通信。 注意：CIDR 与 BlockSize 的对应关系请参考参考内容。
封装协议	选择封装协议。双栈模式下不支持 **IPIP**。 • **IPIP**：使用 IPIP 协议实现跨段通信。 • **VXLAN (Alpha)**：使用 VXLAN 协议实现跨段通信。 • 无封装：通过路由转发直接连接。
封装模式	当封装协议为 **IPIP** 或 **VXLAN** 时，必须设置封装模式，默认为 **Always**。 • **Always**：始终启用 IPIP / VXLAN 隧道。 • **Cross Subnet**：仅当主机处于不同子网时启用 IPIP / VXLAN 隧道；同子网时通过路由转发直接连接。
出站流量 **NAT**	选择是否启用出站流量 NAT（网络地址转换），默认启用。 主要用于设置子网容器组访问外部网络时暴露的访问地址。 启用时，使用宿主机 IP 作为当前子网容器组的访问地址；未启用时，子网内容器组的 IP 将直接暴露给外部网络。

5. 点击 **Confirm**。
6. 在子网详情页，选择 **Actions** > **Allocate Project** / **Allocate Namespace**。
7. 完成配置后点击 **Allocate**。

通过 CLI 创建 Calico 网络子网

```
kubectl apply -f test-calico-subnet.yaml
```

参考内容

CIDR 与 blockSize 的动态匹配关系如下表所示。

CIDR	blockSize 大小	主机数量	单个 IP 池大小
`prefix<=16`	26	1024+	64
`16<prefix<=19`	27	256~1024	32
`prefix=20`	28	256	16
`prefix=21`	29	256	8
`prefix=22`	30	256	4
`prefix=23`	30	128	4
`prefix=24`	30	64	4
`prefix=25`	30	32	4
`prefix=26`	31	32	2
`prefix=27`	31	16	2
`prefix=28`	31	8	2
`prefix=29`	31	4	2
`prefix=30`	31	2	2
`prefix=31`	31	1	2

NOTE

不支持前缀大于 31 的子网配置。

Kube-OVN 网络

在 Kube-OVN Overlay 网络中创建子网，实现集群内资源更细粒度的网络隔离。

> **NOTE**
>
> 平台内置了用于节点与 Pods 通信的 **join** 子网，请避免 **join** 与新建子网之间的网段冲突。

Kube-OVN Overlay 网络子网示例自定义资源（CR）

```yaml
# test-overlay-subnet.yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: test-overlay-subnet
spec:
  default: false
  protocol: Dual
  cidrBlock: 10.1.0.0/23
  natOutgoing: true # 1
  excludeIps: # 2
    - 10.1.1.2
  gatewayType: distributed # 3
  gatewayNode: '' # 4
  private: false
  enableEcmp: false # 5
```

1. 详见出站流量 NAT 参数。
2. 详见保留 IP 参数。
3. 详见网关类型参数。可选值为 `distributed` 或 `centralized`。
4. 详见网关节点参数。
5. 详见 ECMP 参数。需联系管理员开启功能门控。

通过 Web 控制台创建 Kube-OVN Overlay 网络子网

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Network Management** > **Subnet**。
3. 点击 **Create Subnet**。
4. 按照以下说明配置相关参数。

参数	说明
网段	将子网分配给项目或命名空间后，该网段内的 IP 将随机分配给 Pods 使用。
保留 **IP**	设置的保留 IP 不会被自动分配。例如，可用作计算组件的 固定 **IP**。
网关类型	选择子网的网关类型以控制出站流量。 - **Distributed**：集群中每个宿主机都可作为当前宿主机上 Pods 的出站节点，实现分布式出口。 - **Centralized**：集群中所有 Pods 使用一个或多个特定宿主机作为出站节点，便于外部审计和防火墙控制。配置多个集中式 网关节点 可实现高可用。
ECMP (Alpha)	选择 **Centralized** 网关时，可使用 ECMP 功能。默认网关为主从模式，只有主网关处理流量。启用 ECMP（等价多路径路由）后，出站流量将通过多个等价路径路由至所有可用网关节点，提高网关总吞吐量。 注意：请提前开启 ECMP 相关功能。
网关节点	使用 **Centralized** 网关时，选择一个或多个特定宿主机作为网关节点。
出站流量 **NAT**	选择是否启用出站流量 NAT（网络地址转换），默认启用。 主要用于设置子网内 Pods 访问外部网络时暴露的访问地址。 启用时，使用宿主机 IP 作为当前子网 Pods 的访问地址；未启用时，子网内 Pods 的 IP 将直接暴露给外部网络，建议此时使用集中式网关。

5. 点击 **Confirm**。
6. 在子网详情页，选择 **Actions** > **Allocate Project** / **Namespace**。
7. 完成配置后点击 **Allocate**。

通过 **CLI** 创建 **Kube-OVN Overlay** 网络子网

```
kubectl apply -f test-overlay-subnet.yaml
```

Underlay 网络

在 Kube-OVN Underlay 网络中创建子网，不仅实现资源更细粒度的网络隔离，还能提供更优的性能体验。

> **INFO**
>
> Kube-OVN Underlay 中的容器网络需要物理网络支持。请参考最佳实践 准备 Kube-OVN Underlay 物理网络 以确保网络连通性。

使用说明

在 Kube-OVN Underlay 网络中创建子网的一般流程为：添加桥接网络 > 添加 VLAN > 创建子网。

1. 默认网卡名称。
2. 按节点配置网卡。

通过 **Web** 控制台添加桥接网络（可选）

```yaml
# test-provider-network.yaml
kind: ProviderNetwork
apiVersion: kubeovn.io/v1
metadata:
  name: test-provider-network
spec:
  defaultInterface: eth1 (1)
  customInterfaces: (2)
    - interface: eth2
      nodes:
        - node1
  excludeNodes:
    - node2
```

1. 默认网卡名称。
2. 按节点配置网卡。

桥接网络指桥接设备，将网卡绑定到桥接后，可转发容器网络流量，实现与物理网络的互通。

操作步骤：

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Network Management** > **Bridge Network**。
3. 点击 **Add Bridge Network**。
4. 根据以下说明配置相关参数。
   注意：

   - *目标 Pod* 指当前节点上调度的所有 Pods，或绑定特定子网的命名空间中调度到当前节点的 Pods，具体取决于桥接网络下子网的作用范围。
   - Underlay 子网的节点必须具备多块网卡，桥接网络使用的网卡必须专属分配给 Underlay，不能承载其他流量（如 SSH）。例如，若桥接网络有三个节点，计划分别使用 eth0、eth0、eth1 专属给 Underlay，则默认网卡可设置为 eth0，第三个节点的网卡可设置为 eth1。

参数	说明
默认网卡名称	目标 Pod 默认使用该网卡作为桥接网络卡，与物理网络互通。
按节点配置网卡	配置节点上的目标 Pod 将桥接到指定网卡，而非默认网卡。
排除节点	排除的节点上调度的所有 Pods 不会桥接到该节点的任何网卡。 注意：排除节点上的 Pods 无法与物理网络或跨节点容器网络通信，需避免调度相关 Pods 到这些节点。

5. 点击 **Add**。

通过 CLI 添加桥接网络

```
kubectl apply -f test-provider-network.yaml
```

通过 Web 控制台添加 VLAN（可选）

```
# test-vlan.yaml
kind: Vlan
apiVersion: kubeovn.io/v1
metadata:
  name: test-vlan
spec:
  id: 0 1
  provider: test-provider-network 2
```

1. VLAN ID。
2. 桥接网络引用。

平台预置了 **ovn-vlan** 虚拟局域网，连接到 **provider** 桥接网络。也可配置新的 VLAN 连接其他桥接网络，实现 VLAN 之间的网络隔离。

操作步骤：

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Network Management** > **VLAN**。
3. 点击 **Add VLAN**。
4. 根据以下说明配置相关参数。

参数	说明
VLAN ID	VLAN 的唯一标识，用于区分不同虚拟局域网。
桥接网络	VLAN 将连接到该桥接网络，实现与物理网络的互通。

5. 点击 **Add**。

通过 CLI 添加 VLAN

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

Kube-OVN Underlay 网络子网示例自定义资源（CR）

```yaml
# test-underlay-network.yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: test-underlay-network
spec:
  default: false
  protocol: Dual
  cidrBlock: 11.1.0.0/23
  gateway: 11.1.0.1
  excludeIps:
    - 11.1.0.3
  private: false
  allowSubnets: []
  vlan: test-vlan 1
  enableEcmp: false
```

1 VLAN 引用。

通过 **Web** 控制台创建 **Kube-OVN Underlay** 网络子网

> **NOTE**
>
> 平台也预置了用于 Overlay 传输模式下节点与 Pods 通信的 **join** 子网，该子网在 Underlay 传输模式下不使用，务必避免 **join** 与其他子网的 IP 段冲突。

操作步骤：

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Network Management** > **Subnet**。
3. 点击 **Create Subnet**。
4. 根据以下说明配置相关参数。

参数	说明
VLAN	子网所属的 VLAN。

参数	说明
子网	将子网分配给项目或命名空间后，物理子网内的 IP 将随机分配给 Pods 使用。
网关	上述子网内的物理网关。
保留 **IP**	指定的保留 IP 不会被自动分配。例如，可用作计算组件的 固定 **IP**。

5. 点击 **Confirm**。
6. 在子网详情页，选择 **Action** > **Assign Project** / **Namespace**。
7. 完成配置后点击 **Assign**。

通过 CLI 创建 Kube-OVN Underlay 网络子网

```
kubectl apply -f test-underlay-network.yaml
```

相关操作

当集群中同时存在 Underlay 和 Overlay 子网时，可根据需要配置Underlay 与 Overlay 子网自动互通。

子网管理

通过 Web 控制台更新网关

包括更改出站流量方式、网关节点和 NAT 配置。

1. 进入 **Administrator**。
2. 在左侧栏点击 **Network Management** > **Subnets**。
3. 点击子网名称。
4. 选择 **Action** > **Update Gateway**。
5. 更新参数配置，详情参考参数说明。

6. 点击 **OK**。

通过 CLI 更新网关

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {"op": "replace", "path": "/spec/gatewayType", "value": "centralized"},
  {"op": "replace", "path": "/spec/gatewayNode", "value": "192.168.66.21
0"},
  {"op": "replace", "path": "/spec/natOutgoing", "value": true},
  {"op": "replace", "path": "/spec/enableEcmp", "value": true}
]'
```

通过 Web 控制台更新保留 IP

网关 IP 不能从保留 IP 中移除，其他保留 IP 可自由编辑、删除或新增。

1. 进入 **Administrator**。
2. 在左侧栏点击 **Network Management** > **Subnets**。
3. 点击子网名称。
4. 选择 **Action** > **Update Reserved IP**。
5. 完成更新后点击 **Update**。

通过 CLI 更新保留 IP

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {
    "op": "replace",
    "path": "/spec/excludeIps",
    "value": ["10.1.0.1", "10.1.1.2", "10.1.1.4"]
  }
]'
```

通过 Web 控制台分配项目

将子网分配给特定项目，有助于团队更好地管理和隔离不同项目的网络流量，确保每个项目拥有充足的网络资源。

1. 进入 **Administrator**。
2. 在左侧栏点击 **Network Management** > **Subnets**。
3. 点击子网名称。
4. 选择 **Action** > **Assign Project**。
5. 添加或移除项目后，点击 **Assign**。

通过 CLI 分配项目

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {
    "op": "replace",
    "path": "/spec/namespaceSelectors",
    "value": [
      {
        "matchLabels": {
          "cpaas.io/project": "cong"
        }
      }
    ]
  }
]'
```

通过 Web 控制台分配命名空间

将子网分配给特定命名空间，实现更细粒度的网络隔离。

注意：分配过程会重建网关，出站数据包将被丢弃！请确保当前无业务应用访问外部集群。

1. 进入 **Administrator**。
2. 在左侧栏点击 **Network Management** > **Subnets**。
3. 点击子网名称。
4. 选择 **Action** > **Assign Namespace**。

5. 添加或移除命名空间后，点击 **Assign**。

通过 CLI 分配命名空间

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {
    "op": "replace",
    "path": "/spec/namespaces",
    "value": ["cert-manager"]
  }
]'
```

通过 Web 控制台扩展子网

当子网保留 IP 范围达到使用上限或即将耗尽时，可基于原子网范围进行扩展，不影响现有服务的正常运行。

1. 进入 **Administrator**。
2. 在左侧栏点击 **Network Management** > **Subnets**。
3. 点击子网名称。
4. 选择 **Action** > **Expand Subnet**。
5. 完成配置后点击 **Update**。

通过 CLI 扩展子网

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {
    "op": "replace",
    "path": "/spec/cidrBlock",
    "value": "10.1.0.0/22"
  }
]'
```

管理 Calico 网络

支持分配项目和命名空间，详情请参考分配项目和分配命名空间。

通过 Web 控制台删除子网

NOTE

- 删除子网时，如果仍有容器组使用该子网内的 IP，容器组可继续运行且 IP 不变，但无法进行网络通信。可重建容器组以使用默认子网的 IP，或为容器组所在命名空间分配新的子网使用。
- 默认子网不可删除。

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Network Management** > **Subnets**。
3. 点击 ⋮ > **Delete**，执行删除操作。

通过 CLI 删除子网

```
kubectl delete subnet test-overlay-subnet
```

配置 MetalLB

目录

前提条件
通过 Web 控制台配置外部 IP 地址池
通过 Web 控制台配置 BGP Peers
通过 CLI 使用 L2Advertisement 或 BGPAdvertisement 配置外部 IP 地址池
MetalLB 故障排除

前提条件

请确保您已阅读安装文档后再继续操作。

通过 Web 控制台配置外部 IP 地址池

1. 进入 **Administrator**。
2. 在左侧导航栏中，点击 **Network Management** > **External IP Address Pool**。
3. 点击 **Create External IP Address Pool**。
4. 按照以下说明配置相关参数。

参数	说明
Type	• L2：基于 MAC 地址的通信和转发，适用于需要简单快速二层交换的小规模或局域网，具有配置简单和低延迟的优势。 • BGP (Alpha)：基于 IP 地址的路由和转发，使用 BGP 协议交换路由信息，适用于需要跨多个自治系统复杂路由的大规模网络，具有高扩展性和可靠性的优势。
IP Resources	支持 CIDR 和 IP 范围格式输入。点击 **Add** 支持多条输入，示例如下： **CIDR**：`192.168.1.1/24`。 **IP** 范围：`192.168.2.1` ~ `192.168.2.255`。
Available Nodes	在 L2 模式下，可用节点是用于承载所有 VIP 流量的节点；在 BGP 模式下，可用节点是用于承载 VIP、与对等体建立 BGP 连接并对外宣布路由的节点。 • 节点名称：根据节点名称选择可用节点。 • 标签选择器：根据标签选择可用节点。 • 显示节点详情：以列表形式查看最终的可用节点。 注意： • 使用 BGP 类型时，可用节点为下一跳节点；确保所选可用节点是 BGP 连接节点的子集。 • 可单独配置标签选择器或节点名称选择可用节点；若同时配置，最终可用节点为两者的交集。
BGP Peers	选择 BGP 对等体；具体配置请参考 BGP Peers。

5. 点击 **Create**。

通过 Web 控制台配置 BGP Peers

1. 进入 **Administrator**。

2. 在左侧导航栏中，点击 **Network Management** > **BGP Peers**。

3. 点击 **Create BGP Peer**。

4. 按照以下说明配置参数。

参数	说明
Local AS Number	BGP 连接节点所在自治系统的 AS 号。 注意：若无特殊需求，建议使用 IBGP 配置，即本地 AS 号应与对等体 AS 号保持一致。
Peer AS Number	BGP 对等体所在自治系统的 AS 号。
Peer IP	BGP 对等体的 IP 地址，必须是能够建立 BGP 连接的有效 IP 地址。
Local IP	BGP 连接节点的 IP 地址。当 BGP 连接节点有多个 IP 时，选择指定的本地 IP 与对等体建立 BGP 连接。
Peer Port	BGP 对等体的端口号。
BGP-Connected Node	建立 BGP 连接的节点。若不配置此参数，则所有节点均建立 BGP 连接。
eBGP Multi-Hop	允许在非直接相连的 BGP 路由器之间建立 BGP 会话。启用此功能时，BGP 包的默认 TTL 值为 5，允许跨多个中间网络设备建立 BGP 对等关系，使网络设计更灵活。
RouterID	一个 32 位数字值（通常以点分十进制格式表示，类似 IPv4 地址格式），用于唯一标识 BGP 网络中的 BGP 路由器，通常用于建立 BGP 邻居关系、检测路由环路、选择最优路径以及排查网络问题。

5. 点击 **Create**。

通过 CLI 使用 L2Advertisement 或 BGPAdvertisement 配置外部 IP 地址池

```yaml
# ippool-with-L2advertisement.yaml
kind: IPAddressPool
apiVersion: metallb.io/v1beta1
metadata:
  name: test-ippool
  namespace: metallb-system
spec:
  addresses:
    - 13.1.1.1/24
  avoidBuggyIPs: true
---
kind: L2Advertisement
apiVersion: metallb.io/v1beta1
metadata:
  name: test-ippool
  namespace: metallb-system
spec:
  ipAddressPools:
    - test-ippool 1
  nodeSelectors:
    - matchLabels: {}
      matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
            - 192.168.66.210
```

BGP 模式

```yaml
# ippool-with-bgpadvertisement.yaml
kind: IPAddressPool
apiVersion: metallb.io/v1beta1
metadata:
  name: test-pool-bgp
  namespace: metallb-system
spec:
  addresses:
    - 4.4.4.3/23
  avoidBuggyIPs: true
---
kind: BGPAdvertisement
apiVersion: metallb.io/v1beta1
metadata:
  name: test-pool-bgp
  namespace: metallb-system
spec:
  ipAddressPools:
    - test-pool-bgp
  nodeSelectors:
    - matchLabels:
        alertmanager: 'true'
  peers:
    - test-bgp-example
```

```
kubectl apply -f ippool-with-L2advertisement.yaml -f ippool-with-bgpadvertisement.yaml
```

MetalLB 故障排除

现象	可能原因	解决方案
未分配外部 IP	无有效 IPAddressPool 或池配置错误	验证 IP 范围和命名空间
Pod CrashLoop	Speaker 或 Controller 缺少 RBAC 权限	检查 Operator 权限

现象	可能原因	解决方案
BGP 未建立	ASN 不匹配或对等体不可达	检查 `BGPPeer` 规范和网络路由
L2 不工作	VLAN 配置错误或 ARP 过滤	使用 `arping` 验证广播可达性

查看更多内容请访问 Troubleshooting MetalLB↗

配置 GatewayAPI Gateway

目录

- 概述
- 前提条件
- Gateway 基础知识
  - 什么是 Gateway
  - Gateway 暴露方式（Service Type）
    - LoadBalancer（推荐）
    - NodePort
    - ClusterIP
  - Listener 配置
    - 端口和协议
    - AllowRouteNS
    - TLS 配置
  - EnvoyProxy（部署配置）
    - 镜像仓库
- 创建 Gateway
  - 通过 Web 控制台
    - Listener 配置
  - 通过 YAML
    - 多种监听器类型的完整示例
- 查看 Gateway 详情
- 监听器与路由参考
  - 主机名

主机名交集规则

支持的路由类型

下一步

概述

本文档说明在 Envoy Gateway operator 和 `EnvoyGatewayCtl` 准备就绪后，如何配置 `Gateway`。
`Gateway` 定义流量如何进入网关，而配套的 `EnvoyProxy` 控制底层 Envoy 数据平面的部署方式。

在推荐的工作流程中，本文档位于 Envoy Gateway Operator 之后，Configure GatewayAPI Route 之前。

前提条件

请确保在继续之前已完成以下操作：

1. 阅读 Envoy Gateway Operator，了解基本概念和资源关系
2. 安装 Envoy Gateway operator 并创建 `EnvoyGatewayCtl`

> **NOTE**
>
> 本文档先介绍主要的 Gateway 概念，然后展示如何创建 `Gateway`。如果您已经熟悉这些概念，可以直接跳转到 创建 Gateway 部分。

Gateway 基础知识

什么是 Gateway

`Gateway` 是流量进入集群的入口点。它定义了外部请求如何被接收并路由到后端服务。
`Gateway` 主要定义：

- **Listeners**：定义网关监听的端口、协议和主机名
- **GatewayClass**：选择哪个 gateway controller 管理该 `Gateway`
- 基础设施引用：引用一个 `EnvoyProxy`，控制底层 Envoy 数据平面的部署方式

Gateway 暴露方式（Service Type）

Service Type 配置网关通过底层 Envoy Service 的暴露方式。共有三种模式：LoadBalancer、NodePort 和 ClusterIP。

在 YAML 中，该设置配置在配套的 `EnvoyProxy` 资源的 `.spec.provider.kubernetes.envoyService.type` 字段。

LoadBalancer（推荐）

优点是易用且具备高可用负载均衡能力。
使用 LoadBalancer 需要集群支持 LoadBalancer，可以通过 MetalLB 启用。

使用 MetalLB 时，可以通过服务注解指定静态虚拟 IP。在 Web 控制台中，使用 **Service Annotation** 字段：

```
metallb.universe.tf/address-pool: ADDRESS_POOL_NAME
# 或直接指定具体 IP
metallb.universe.tf/loadBalancerIPs: VIP_IP
```

详情请参见 如何在使用 MetalLB 时指定 VIP。

NodePort

优点是不依赖任何外部组件。

但使用 NodePort 有以下缺点：

- Kubernetes 会为 NodePort 分配与服务端口不同的端口号，访问时必须使用 NodePort 端口号，而非服务端口

- 服务可通过集群中任意节点 IP 访问，存在潜在安全风险

如何获取 NodePort 正确端口

在 Gateway 详情页，当 Service Type 为 NodePort 时，监听器列表会显示 NodePort 列，展示分配的端口号。也可以使用如下命令获取：

```
kubectl get svc -n ${ENVOYGATEWAYCTL_NS} -l gateway.envoyproxy.io/owning-gateway-name=${GATEWAY_NAME} -o=jsonpath="{.items[0].spec.ports[?(@.port==${PORT})].nodePort}"
```

输出即为 NodePort 端口号。

ClusterIP

如果不需要外部暴露，非常方便。

Listener 配置

Listener 定义网关监听的端口和协议。在 HTTP 或 HTTPS 协议下，不同主机名可视为不同监听器。

不能创建端口、协议或主机名冲突的监听器。

`Gateway` 中必须至少创建一个监听器。

端口和协议

每个监听器配置端口号和协议。支持的协议有：HTTP、HTTPS、TCP、UDP、TLS。

AllowRouteNS

默认情况下，Routes 只能附加到 `Same` 命名空间下的 Gateway。若需跨命名空间路由，使用 `Allowed Routes Namespace` 字段：

- `Same`：允许同命名空间的 Routes 附加到该监听器
- `All`：允许任意命名空间的 Routes 附加到该监听器

- `Selector`：允许符合选择器匹配的命名空间中的 Routes 附加到该监听器

在 ACP 中，项目通过命名空间上的标签标识，例如 `cpaas.io/project: <project-name>`。如果希望监听器仅被特定项目的 Routes 使用，选择 `Selector` 并匹配目标命名空间的项目标签。

监听器的 `Allowed Routes Namespace` 设置与协议共同决定了在路由 Web 控制台发布路由到监听器时可用的监听器列表，详见 附加到其他命名空间创建的 gateway。

TLS 配置

对于 HTTPS 和 TLS 协议，需要配置 TLS 设置。

TLS 模式：

TLS 模式	描述	是否需要证书
Terminate	Envoy 终止 TLS，解密流量后转发到后端服务	是，必须选择 TLS 证书
Passthrough	Envoy 直接透传加密的 TLS 流量到后端服务，不解密	否

> **NOTE**
>
> - HTTPS 协议仅支持 **Terminate** 模式
> - TLS 协议支持 **Terminate** 和 **Passthrough** 两种模式
> - TLS 监听器中，**Passthrough** 模式支持 TLSRoute
> - TLS 监听器中，**Terminate** 模式支持 TCPRoute

证书要求：

- 默认只能使用同命名空间下创建的 Secret
- Secret 类型必须为 `kubernetes.io/tls`，且包含 `tls.crt` 和 `tls.key` 键
- 跨命名空间使用 Secret，详见 使用其他命名空间创建的 Secret

EnvoyProxy（部署配置）

Envoy Gateway 使用 `EnvoyProxy` 资源控制网关的部署配置。建议为每个 Gateway 创建专用的 `EnvoyProxy` 资源，并通过 Gateway 的 `.spec.infrastructure.parametersRef` 字段引用。

通过 Web 控制台使用 `EnvoyGatewayCtl` 创建的 `GatewayClass` 创建 `Gateway` 时，控制台会自动创建同名同命名空间的配套 `EnvoyProxy` 资源。

通过 YAML 应用创建 `Gateway` 时，需自行确保 `Gateway` 的 `.spec.infrastructure.parametersRef` 与引用的 `EnvoyProxy` 资源保持一致。

此一对一映射方式提供更好的隔离性和更细粒度的部署配置控制，例如：

- 副本数
- 资源限制和请求
- 节点选择器
- 服务类型和注解
- 镜像仓库

镜像仓库

镜像仓库已为您的集群预配置默认值，除非必要，请勿修改。

其他部署配置方式，详见 deployment-mode ↗。

创建 Gateway

通过 Web 控制台

1. 进入 `Alauda Container Platform -> Networking -> Gateway -> Gateways`
2. 点击 `Create Gateway` 按钮
3. 在 `Create Gateway` 页面，选择由您的 `EnvoyGatewayCtl` 创建的 `GatewayClass`。在 Envoy Gateway Operator 推荐的默认示例中，该值为 `envoy-gateway-operator-cpaas-`

`default`。

页面显示以下配置选项：

字段	说明	YAML 路径
Name	Gateway 名称	gateway: `.metadata.name` envoyproxy: `.metadata.name`
GatewayClass	使用哪个 GatewayClass	gateway: `.spec.gatewayClassName`
Service Type	Service Type	envoyproxy: `.spec.provider.kubernetes.envoyService.ty`
Service Annotation	服务注解	envoyproxy: `.spec.provider.kubernetes.envoyService.an`
Resource Limits	部署资源限制	envoyproxy: `.spec.provider.kubernetes.envoyDeployment.container.`
Replicas	部署副本数	envoyproxy: `.spec.provider.kubernetes.envoyDeployment`
Node Labels	部署节点选择器	envoyproxy: `.spec.provider.kubernetes.envoyDeployment.nodeSelect`
Listener	Listener	gateway: `.spec.listeners`

WARNING

Web 控制台表单仅支持 `EnvoyGatewayCtl` 创建的 GatewayClasses。其他 GatewayClasses 请使用 YAML 编辑器。

NOTE

使用 `EnvoyGatewayCtl` 创建的 `GatewayClass` 时，Web 控制台会自动创建与 Gateway 同名同命名空间的 配套 EnvoyProxy 资源。

Listener 配置

创建或编辑监听器时，可以配置以下内容：

字段	说明	YAML 路径
Name	监听器名称	`.spec.listeners[].name`
Port	监听端口号	`.spec.listeners[].port`
Protocol	监听协议。选项：HTTP、HTTPS、TCP、UDP、TLS	`.spec.listeners[].protocol`
Hostname	可选。监听器的主机名	`.spec.listeners[].hostname`
Allowed Routes Namespace	控制哪些命名空间的路由可以附加到该监听器	`.spec.listeners[].allowedRoutes.namespaces.from`

HTTPS 协议配置

选择 **HTTPS** 协议时：

字段	说明	YAML 路径
Certificates	必填。选择包含 TLS 证书的 Kubernetes Secret	`.spec.listeners[].tls.certificateRefs`

> **NOTE**
>
> - HTTPS 协议仅支持 **Terminate** 模式
> - HTTPS 监听器必须选择证书
> - 默认只能使用同命名空间创建的 Secret

TLS 协议配置

选择 **TLS** 协议时：

配置字段

字段	说明	**YAML** 路径
TLS Mode	选择 TLS 模式。选项：Terminate、Passthrough。默认：Terminate	`.spec.listeners[].tls.mode`
TLS Certificate	仅在 TLS Mode 为 Terminate 时必填。选择包含 TLS 证书的 Secret	`.spec.listeners[].tls.certificateRefs`

TLS 模式详情，见 TLS 配置。

通过 YAML

如果未使用推荐的默认示例，请将 `envoy-gateway-operator-cpaas-default` 替换为您自己的 `GatewayClass`。

以下最小示例创建一个 HTTP `Gateway` 和一个专用的 `EnvoyProxy`。

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo
  namespace: demo
spec:
  infrastructure:
    parametersRef:
      group: gateway.envoyproxy.io
      kind: EnvoyProxy
      name: demo
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: http
      port: 80
      protocol: HTTP
      allowedRoutes:
        namespaces:
          from: Same
---
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    kubernetes:
      envoyService:
        type: ClusterIP
      envoyDeployment:
        replicas: 1
        container:
          imageRepository: registry.alauda.cn:60080/acp/envoyproxy/envoy
          resources:
            limits:
              cpu: '1'
              memory: 1Gi
            requests:
              cpu: '1'
              memory: 1Gi
    type: Kubernetes
```

多种监听器类型的完整示例

如果需要更多监听器类型，可使用以下完整示例：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo
  namespace: demo
spec:
  infrastructure: ①
    parametersRef:
      group: gateway.envoyproxy.io
      kind: EnvoyProxy
      name: demo
  gatewayClassName: envoy-gateway-operator-cpaas-default ②
  listeners: ③
    - name: http
      port: 80
      hostname: a.com ④
      protocol: HTTP ⑤
      allowedRoutes:
        namespaces:
          from: Same ⑥
    - name: https
      port: 443
      hostname: a.com
      protocol: HTTPS
      allowedRoutes:
        namespaces:
          from: Same
      tls: ⑦
        mode: Terminate
        certificateRefs:
          - name: demo-tls
    - name: tls-passthrough
      port: 8443
      hostname: tls.example.com
      protocol: TLS
      allowedRoutes:
        namespaces:
          from: Same
      tls:
        mode: Passthrough
    - name: tls-terminate
      port: 9443
      hostname: secure.example.com
```

```yaml
      protocol: TLS
      allowedRoutes:
        namespaces:
          from: Same
      tls:
        mode: Terminate
        certificateRefs:
          - name: demo-tls
    - name: tcp
      port: 8080
      protocol: TCP
      allowedRoutes:
        namespaces:
          from: Same
    - name: udp
      port: 8081
      protocol: UDP
      allowedRoutes:
        namespaces:
          from: Same
---
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo 8
  namespace: demo
spec:
  provider:
    kubernetes:
      envoyService:
        type: ClusterIP 9
      envoyDeployment:
        replicas: 1
        container:
          imageRepository: registry.alauda.cn:60080/acp/envoyproxy/envoy
10
          resources: 11
            limits:
              cpu: '1'
              memory: 1Gi
            requests:
              cpu: '1'
              memory: 1Gi
    type: Kubernetes 12
```

1. 参考 EnvoyProxy 资源 进行部署配置
2. 如有需要，替换 `envoy-gateway-operator-cpaas-default` 为您自己的 `GatewayClass`
3. `listeners` 定义流量如何进入网关
4. `hostname` 影响基于主机名的路由匹配监听器
5. `protocol` 决定可附加的路由类型
6. `allowedRoutes` 控制允许附加路由的命名空间
7. `tls` 配置 HTTPS 和 TLS 监听器的 TLS 终止或透传
8. `EnvoyProxy` 名称必须与 `.spec.infrastructure.parametersRef.name` 匹配
9. `envoyService.type` 控制网关暴露方式
10. 除非必要，保持 `imageRepository` 不变
11. `resources` 配置 Envoy 数据平面资源限制和请求
12. 保持 `provider.type` 为 `Kubernetes`

查看 Gateway 详情

在 Gateway 详情页，监听器列表显示以下信息：

列名	说明
Name	监听器名称
Protocol	监听器协议（HTTP、HTTPS、TCP、UDP、TLS）
Port	配置的端口号
NodePort	当 Service Type 为 NodePort 时显示，展示访问监听器的分配 NodePort 端口号。

NOTE

当 Gateway Service Type 为 **NodePort** 时，监听器列表会显示额外的 **NodePort** 列。访问网关时请使用 NodePort 端口号，而非服务端口。详情见 如何获取 NodePort 正确端口。

监听器与路由参考

附加 `Route` 资源到 `Gateway` 时，请遵循以下规则。

主机名

监听器中的主机名是同协议监听器的唯一标识。不能在 Gateway 中添加或更新冲突的监听器。

主机名交集规则

请求到达时，会匹配监听器主机名与路由主机名的交集。只有交集中的主机名用于路由流量。

监听器主机名	路由主机名	交集结果	示例
无主机名	无主机名	匹配所有主机	接受任意传入的 Host 头
无主机名	有主机名（如 `api.example.com`）	所有路由主机名	仅匹配 `api.example.com` 的请求
有主机名（如 `api.example.com`）	无主机名	所有监听器主机名	仅匹配 `api.example.com` 的请求
有主机名（如 `api.example.com`）	有完全匹配的主机名	完全匹配的主机名	仅匹配 `api.example.com` 的请求

监听器主机名	路由主机名	交集结果	示例
有通配符（如 `*.example.com`）	有匹配的主机名	匹配具体主机名	匹配 `api.example.com` 或 `web.example.com` 的请求
有主机名（如 `api.example.com`）	有不匹配的主机名	无交集 - 路由状态异常	路由无法处理流量

NOTE

通配符（`*`）表示后缀匹配。例如，`*.example.com` 匹配 `foo.example.com` 和 `bar.example.com`，但不匹配 `example.com`。

WARNING

无交集意味着路由状态异常，流量无法处理。

支持的路由类型

每个监听器根据其协议支持不同的路由类型：

监听器协议	支持的路由类型
HTTP	HTTPRoute, GRPCRoute
HTTPS	HTTPRoute, GRPCRoute
TLS（Passthrough 模式）	TLSRoute
TLS（Terminate 模式）	TCPRoute

监听器协议	支持的路由类型
TCP	TCPRoute
UDP	UDPRoute

配置路由时，确保路由类型与附加的监听器协议匹配。例如，不能将 `HTTPRoute` 附加到 `TCP` 监听器。

下一步

`Gateway` 准备就绪后，继续进行 配置 GatewayAPI Route。
如果在路由附加后需要高级流量控制，继续进行 配置 GatewayAPI Policy。

配置 GatewayAPI 策略

目录

- 概述
- 前提条件
- 策略附加基础
- 策略附加汇总
- 在 Web 控制台创建策略
- SecurityPolicy
  - 通过 Web 控制台配置
    - API Key Authentication
    - CORS 配置
  - 通过 YAML 配置
  - 参考
    - 功能
    - 工作原理
    - 注意事项
    - 官方文档
- BackendTLSPolicy
  - 通过 Web 控制台配置
  - 通过 YAML 配置
  - 参考
    - 功能
    - 注意事项
    - 官方文档

ClientTrafficPolicy
通过 Web 控制台配置
通过 YAML 配置
参考
功能
注意事项
官方文档
BackendTrafficPolicy
通过 Web 控制台配置
通过 YAML 配置
参考
功能
注意事项
官方文档
相关任务

概述

本文档说明在 `Gateway` 和 `Route` 资源准备好之后，如何配置策略资源。策略通过 `.spec.targetRefs` 使用策略附加模式，将额外的流量、安全和后端行为附加到支持的资源上。

在推荐的工作流程中，本文档位于配置 GatewayAPI 路由之后。

Envoy Gateway 当前提供四种策略类型：`SecurityPolicy`、`BackendTLSPolicy`、`ClientTrafficPolicy` 和 `BackendTrafficPolicy`。

前提条件

请确保在继续之前已完成以下操作：

1. 阅读配置 GatewayAPI Gateway和配置 GatewayAPI 路由
2. 创建策略将附加到的目标资源，例如 `Gateway`、`Route` 或 `Service`

策略附加基础

策略通过 `.spec.targetRefs` 附加到其他资源。

默认情况下，策略只能附加到同一命名空间中的资源。

对于 `Gateway` 目标，支持策略类型时可以使用 `sectionName` 来定位特定的监听器。对于 `Service` 目标，`sectionName` 指的是 Service 端口名称。

策略附加汇总

策略类型	目的	**Web 控制台支持**	**Gateway**	**HTTPRoute**	**GRPCRoute**
SecurityPolicy	身份验证、授权、CORS 及其他安全功能	API Key Auth, CORS	✅ (监听器名称 / `ALL`)	✅	✅
BackendTLSPolicy	Envoy 与后端服务之间的 TLS 配置	支持	❌	❌	❌

策略类型	目的	Web 控制台支持	Gateway	HTTPRoute	GRPCRoute
ClientTrafficPolicy	面向客户端的超时和连接行为控制	超时设置	✅ (监听器名称 / `ALL`)	❌	❌
BackendTrafficPolicy	后端超时和连接行为控制	超时设置	✅ (监听器名称 / `ALL`)	✅	✅

`sectionName` 用于定位 `Gateway` 上的特定监听器或 `Service` 上的特定端口。省略或设置为 `ALL` 时，策略应用于所有监听器或端口。

在 Web 控制台创建策略

所有策略类型均从同一入口创建：

1. 进入 `Alauda Container Platform -> Networking -> Gateway -> Policies`
2. 在 `Policy Type` 下拉框中选择所需的值
3. 点击 `Create Policy` 按钮

以下章节仅关注各策略类型特有的字段。

SecurityPolicy

通过 Web 控制台配置

通用字段（所有策略共享）：

字段	说明	YAML 路径
Policy Type	要创建的策略类型	`.kind`
Attach To	此策略适用的 Gateway API 资源。支持 Gateway、HTTPRoute 和 GRPCRoute。附加到 Gateway 时，可选择指定监听器名称或选择所有监听器。	`.spec.targetRefs`

SecurityPolicy 特有字段：

字段	说明	YAML 路径
Authorization Type	使用的身份验证/授权方式。支持多选：API Key Authentication、CORS 配置	`.spec.apiKeyAuth`, `.spec.cors`

API Key Authentication

字段	说明	YAML 路径
Secrets	包含用于身份验证的 API Key 的 Kubernetes secret	`.spec.apiKeyAuth.credentialRefs`
Extract From	指定从哪里提取 API Key（HTTP 头或查询参数）	`.spec.apiKeyAuth.extractFrom`

CORS 配置

字段	说明	YAML 路径
Allow Origins	允许的 CORS 请求来源列表	`.spec.cors.allowOrigins`
Allow Methods	允许的 HTTP 方法列表	`.spec.cors.allowMethods`

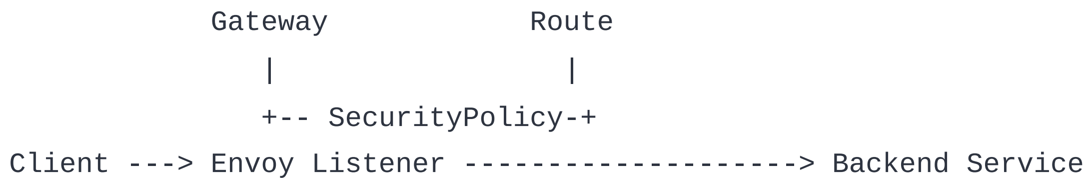
字段	说明	YAML 路径
Allow Headers	允许的 CORS 请求头列表	`.spec.cors.allowHeaders`
Expose Headers	响应中暴露给客户端的头列表	`.spec.cors.exposeHeaders`
Max Age	CORS 预检响应的缓存时间	`.spec.cors.maxAge`
Allow Credentials	是否允许 CORS 请求携带凭据	`.spec.cors.allowCredentials`

通过 YAML 配置

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: demo-security-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: demo
  apiKeyAuth:
    credentialRefs:
      - group: ""
        kind: Secret
        name: demo
        namespace: demo
    extractFrom:
      - headers:
          - authorization
  cors:
    allowOrigins:
      - "https://example.com"
    allowMethods:
      - GET
      - POST
    allowHeaders:
      - "Content-Type"
    exposeHeaders:
      - "X-Custom-Header"
    maxAge: "1h"
    allowCredentials: true
```

参考

SecurityPolicy 用于配置 Gateway 和路由的身份验证、授权及其他安全相关功能。它提供声明式方式，在请求到达后端应用之前验证请求，从而保护服务。



功能

- 身份验证：使用多种方式验证客户端身份（API Key、JWT、OIDC、Basic Auth）
- 授权：基于验证的凭据控制资源访问
- **CORS** 配置：管理跨域资源共享策略

工作原理

1. 创建包含所需身份验证/授权规则的 SecurityPolicy
2. 将其附加到特定的 Gateway、HTTPRoute 或 GRPCRoute
3. Envoy Gateway 根据策略验证传入请求
4. 合法请求转发至后端服务；非法请求返回相应 HTTP 状态码拒绝访问

注意事项

1. Web 控制台当前支持配置 **API Key Authentication** 和 **CORS**。其他身份验证方式及高级安全功能需使用 YAML 配置。
2. 每个路由只能关联一个 SecurityPolicy。
3. 如果 SecurityPolicy 引用的 secret 无值，则所有请求都会被拒绝，返回 `401 Unauthorized`。
4. Web 控制台中，默认 `Extract From` 字段为 `header`，`Header Name` 字段为 `authorization`。
5. 可在 Web 控制台的路由拓扑标签页查看附加到路由的策略。

官方文档

- SecurityPolicy 规范 ↗
- API Key Authentication ↗

BackendTLSPolicy

通过 Web 控制台配置

通用字段：

字段	说明	YAML 路径
Policy Type	要创建的策略类型	`.kind`
Attach To	此策略适用的 Service。必须指定 Service 名称和端口名称（`sectionName`）。	`.spec.targetRefs`

BackendTLSPolicy 特有字段：

字段	说明	YAML 路径
Hostname	必填。Envoy 连接后端服务时使用的 SNI（服务器名称指示）	`.spec.validation.hostname`
Subject Alternative Names	可选。用于后端 HTTPS 响应验证。未指定时，默认使用 hostname 值。	`.spec.validation.subjectAltNames`
Validation Type	验证后端 TLS 证书的方法。选项：`CACertificateRefs`（使用自定义 CA 证书）、`WellKnownCACertificates`（使用系统 CA 证书）	`.spec.validation`

CACertificateRefs 配置：

字段	说明	YAML 路径
CA Certificate Secret	包含 CA 证书的 Kubernetes secret。该 secret 必须包含 `ca.crt` 键，内容为 PEM 编码的 TLS 证书。	`.spec.validation.cACertificateRefs`

NOTE

创建或选择 CA 证书 secret 时：

- secret 类型必须适合 CA 证书
- 键必须为 `ca.crt`
- 可导入证书文件，文件必须以 `-----BEGIN CERTIFICATE-----` 开头，以 `-----END CERTIFICATE-----` 结尾
- 导入格式无效的证书时，会显示错误信息 “must contain PEM-encoded TLS certificates”
- 选择无 `ca.crt` 键的现有 secret 时，会显示错误信息 “must have ca.crt key”

通过 YAML 配置

```yaml
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: BackendTLSPolicy
metadata:
  name: demo-backend-tls-policy
  namespace: demo
spec:
  targetRefs:
    - group: ""
      kind: Service
      name: demo-backend
      namespace: demo
      sectionName: https-port
  validation:
    hostname: backend.example.com
    subjectAltNames:
      - backend.example.com
    cACertificateRefs:
      - group: ""
        kind: Secret
        name: backend-ca
        namespace: demo
```

参考

BackendTLSPolicy 控制 Envoy Gateway 与后端服务之间的 TLS 配置。允许您配置：

- **SNI**（服务器名称指示）：建立 TLS 连接时使用的主机名
- 证书验证：如何验证后端服务器证书
- **CA** 证书：用于验证后端证书的自定义 CA 证书

功能

- 配置与后端服务连接的 TLS 设置
- 支持自定义 CA 证书或系统知名 CA 证书
- 配置 SNI 以完成正确的 TLS 握手

注意事项

1. `targetRefs` 中的 `sectionName` 对应 Service 的端口名称。
2. 使用 `WellKnownCACertificates` 时，使用系统默认 CA 证书进行验证。
3. hostname 为必填项，作为 Envoy 连接后端时的 SNI 值。

官方文档

- BackendTLSPolicy 规范↗

ClientTrafficPolicy

通过 Web 控制台配置

通用字段：

字段	说明	YAML 路径
Policy Type	要创建的策略类型	`.kind`
Attach To	此策略适用的 Gateway。可选择指定监听器名称或选择所有监听器。	`.spec.targetRefs`

超时配置（选项）：

字段	说明	YAML 路径
TCP Idle Timeout	TCP 连接的空闲超时时间。空闲定义为上下游连接	`.spec.settings.timeout.tcp.idleTimeout`

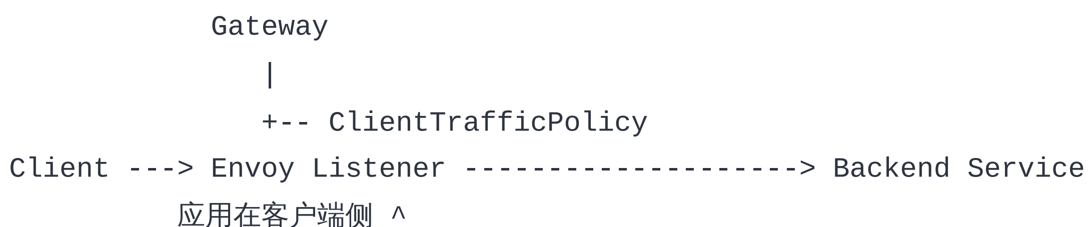
字段	说明	YAML 路径
	均无数据发送或接收的时间段。默认值：1 小时。	
HTTP Request Received Timeout	Envoy 等待完整请求接收的时间。计时从请求开始，到请求最后一个字节发送到上游或响应开始时停止。默认值：1 小时。	`.spec.settings.timeout.http.requestReceivedTimeout`
HTTP Idle Timeout	HTTP 连接的空闲超时时间。空闲定义为连接中无活动请求的时间段。默认值：无限制。	`.spec.settings.timeout.http.idleTimeout`
HTTP Stream Idle Timeout	流空闲超时时间，定义流在无上下游活动时可存在的最长时间。默认值：5 分钟。	`.spec.settings.timeout.http.streamIdleTimeout`

通过 YAML 配置

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: ClientTrafficPolicy
metadata:
  name: demo-client-traffic-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: https
  settings:
    timeout:
      tcp:
        idleTimeout: "30m"
      http:
        requestReceivedTimeout: "60s"
        idleTimeout: "5m"
        streamIdleTimeout: "30s"
```

参考

ClientTrafficPolicy 控制客户端到 Envoy Gateway 的连接行为。提供细粒度控制：



- **TCP** 设置：连接级别的超时和保持活动设置
- **HTTP** 设置：请求/响应超时及 HTTP 协议行为

功能

- 配置 TCP 连接空闲超时
- 控制 HTTP 请求接收超时
- 设置 HTTP 连接空闲超时

- 配置 HTTP 流空闲超时

注意事项

1. 超时值以持续时间字符串表示（如 "30s"、"5m"、"1h"）。

官方文档

- ClientTrafficPolicy 规范

BackendTrafficPolicy

通过 Web 控制台配置

通用字段：

字段	说明	YAML 路径
Policy Type	要创建的策略类型	`.kind`
Attach To	此策略适用的 Gateway API 资源。支持 Gateway、HTTPRoute、GRPCRoute、TCPRoute、UDPRoute 和 TLSRoute。附加到 Gateway 时，可选择指定监听器名称或选择 `ALL`。YAML 中通过 `.spec.targetRefs` 的 `sectionName` 配置。	`.spec.targetRefs`

超时配置（选项）：

字段	说明	YAML 路径
TCP Connection Timeout	网络连接建立超时，包括 TCP 和 TLS 握手。默认值：10 秒。	`.spec.settings.timeout.tcp.connectionTimeout`

字段	说明	YAML 路径
HTTP Connection Idle Timeout	HTTP 连接空闲超时。空闲定义为连接中无活动请求的时间段。默认值：1 小时。	`.spec.settings.timeout.http.connectionIdleTimeout`
HTTP Max Connection Duration	HTTP 连接的最大持续时间。默认值：无限制。	`.spec.settings.timeout.http.maxConnectionDuration`
HTTP Request Timeout	从上游接收完整响应的超时时间。默认值：15 秒。支持设置为无限制。	`.spec.settings.timeout.http.requestTimeout`

通过 YAML 配置

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: BackendTrafficPolicy
metadata:
  name: demo-backend-traffic-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: demo
  settings:
    timeout:
      tcp:
        connectionTimeout: "5s"
      http:
        connectionIdleTimeout: "30m"
        maxConnectionDuration: "1h"
        requestTimeout: "30s"
```

参考

BackendTrafficPolicy 控制 Envoy Gateway 到后端服务的连接行为。提供细粒度控制：

- **TCP** 设置：连接建立超时
- **HTTP** 设置：连接持续时间、空闲超时和请求超时

功能

- 配置 TCP 连接建立超时
- 控制 HTTP 连接空闲超时
- 设置最大 HTTP 连接持续时间

- 配置 HTTP 请求超时

注意事项

1. 超时值以持续时间字符串表示（如 "30s"、"5m"、"1h"）。
2. `requestTimeout` 字段支持设置为 "unlimited" 以禁用超时。

官方文档

- BackendTrafficPolicy 规范↗

相关任务

策略附加完成后，请继续参考 Envoy Gateway 任务获取更多操作示例和高级配置任务。

配置 GatewayAPI Route

目录

- 概述
- 前提条件
- 配置
  - 通过 Web 控制台配置
    - 创建 HTTPRoute
    - 创建 TCP/UDP Route
    - 创建 GRPCRoute
    - 创建 TLSRoute
  - 通过 YAML 配置
  - 路由字段参考
    - 发布到监听器
    - 后端
    - 主机名
    - 规则
    - HTTPRoute 参考
    - GRPCRoute 匹配和过滤器参考
    - TLSRoute 参考
- 查看
  - 拓扑
- 下一步
- 相关任务

概述

本文档说明了在 `Gateway` 就绪后如何配置 `Route` 资源。`Route` 绑定到一个或多个网关监听器，并定义匹配的流量如何转发到后端服务。

在推荐的工作流程中，本文档位于配置 GatewayAPI Gateway之后，配置 GatewayAPI Policy 之前。

除了创建和更新操作外，本文档还介绍了 ACP Web 控制台提供的额外路由查看功能。

前提条件

请确保在继续之前完成以下操作：

1. 阅读配置 GatewayAPI Gateway，了解监听器、绑定规则和 `EnvoyProxy`
2. 创建一个 `Gateway`，供您的 `Route` 绑定使用

> **NOTE**
>
> 本文档先分别介绍每种路由类型，然后提供 YAML 示例，最后在共享参考部分说明通用路由概念。

配置

`Route` 绑定到 `Gateway` 上的一个或多个监听器。您可以选择的监听器取决于路由类型、监听器协议以及监听器允许的路由命名空间设置。

通过 Web 控制台配置

1. 进入 `Alauda Container Platform -> Networking -> Gateway -> Routes`
2. 点击 `Create Route` 按钮
3. 选择路由类型（HTTPRoute、TCPRoute、UDPRoute、GRPCRoute 或 TLSRoute）

创建 HTTPRoute

字段	说明	YAML 路径
Publish to Listener	发布到监听器	`.spec.parentRefs`
Hostnames	主机名	`.spec.hostnames`
Matches	匹配规则	`.spec.rules[].matches`
Filters	过滤器	`.spec.rules[].filters`
Backend Instance	后端	`.spec.rules[].backendRefs`
Options	选项	`.spec.rules[].filters`, `.spec.rules[].timeouts`, `.spec.rules[].retry`, `.spec.rules[].sessionPersistence`

选项配置

Options 字段允许您配置高级流量管理设置：

选项	说明	YAML 路径
Session Affinity	会话保持	`.spec.rules[].sessionPersistence`
Timeout	超时设置	`.spec.rules[].timeouts`
Retry	重试策略	`.spec.rules[].retry`

创建 TCP/UDP Route

字段	说明	YAML 路径
Publish to Listener	发布到监听器	`.spec.parentRefs`
Backend Instance	后端	`.spec.rules[].backendRefs`

创建 GRPCRoute

字段	说明	YAML 路径
Publish to Listener	发布到监听器	`.spec.parentRefs`
Hostnames	主机名	`.spec.hostnames`
Matches	grpc 匹配	`.spec.rules[].matches`
Filters	grpc 过滤器	`.spec.rules[].filters`
Backend Instance	后端	`.spec.rules[].backendRefs`

创建 TLSRoute

字段	说明	YAML 路径
Publish to Listener	发布到监听器	`.spec.parentRefs`
Hostnames	主机名（可选）	`.spec.hostnames`
Backend Instance	后端	`.spec.rules[].backendRefs`

通过 YAML 配置

以下最小示例创建了一个 `HTTPRoute`，绑定到 `demo` `Gateway` 的 `https` 监听器，并将匹配流量转发到后端 `Service`。

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: demo-443
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: https
  rules:
    - matches:
        - path:
            type: Exact
            value: /a
      backendRefs:
        - group: ''
          kind: Service
          name: echo-resty
          namespace: demo-space
          port: 80
          weight: 100
```

如果您需要更多路由类型和高级 HTTPRoute 选项，请使用以下完整示例：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: demo-443
  namespace: demo
spec:
  hostnames: 1
    - example.com
  parentRefs: 2
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: https
  rules: 3
    - backendRefs: 4
        - group: ''
          kind: Service
          name: echo-resty
          namespace: demo-space
          port: 80
          weight: 100
      filters: [] 5
      matches: 6
        - path:
            type: Exact
            value: /a
      timeouts: 7
        request: '30s'
        backendRequest: '10s'
      retry: 8
        codes:
          - 503
        attempts: 3
        backoff: '100ms'
      sessionPersistence: 9
        type: Cookie
        sessionName: a
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TCPRoute
metadata:
  name: tcp
  namespace: demo-space
```

```yaml
spec:
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: tcp
  rules:
    - backendRefs:
        - group: ''
          kind: Service
          name: echo-resty
          port: 80
          weight: 100
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: UDPRoute
metadata:
  name: udp
  namespace: demo
spec:
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      namespace: demo
      sectionName: udp
  rules:
    - backendRefs:
        - group: ''
          kind: Service
          name: echo-resty
          namespace: demo
          port: 53
          weight: 100
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: GRPCRoute
metadata:
  name: grpc
  namespace: demo
spec:
  hostnames:
    - grpc.example.com
  parentRefs:
```

```yaml
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: https
  rules:
    - matches:
        - method:
            type: service
            value: myservice
      filters:
        - type: RequestHeaderModifier
          requestHeaderModifier:
            set:
              - name: x-custom-header
                value: custom-value
      backendRefs:
        - group: ''
          kind: Service
          name: grpc-service
          port: 50051
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TLSRoute
metadata:
  name: tls
  namespace: demo
spec:
  hostnames:
    - tls.example.com
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: tls
  rules:
    - backendRefs:
        - group: ''
          kind: Service
          name: tls-backend
          port: 443
```

1. 主机名
2. 发布到监听器

3 规则

4 后端

5 过滤器

6 匹配

7 选项

8 超时

9 重试

10 会话保持

路由字段参考

每个路由都是由 `GatewayAPI` 规范定义的 CR。有关每种路由类型的字段和配置选项的详细信息，请参阅官方文档：

- HTTPRoute 规范
- TCPRoute 规范
- UDPRoute 规范
- GRPCRoute 规范
- TLSRoute 规范

发布到监听器

在 **Web** 控制台中

在 Web 控制台中，您可以选择多个监听器发布路由。可用的监听器候选项基于以下条件过滤：

- 用户权限：您必须有访问该网关命名空间的权限（项目必须包含该命名空间）。
- 路由命名空间白名单：网关监听器允许的路由命名空间必须包含路由所在的命名空间。
- 路由类型匹配：路由的类型（HTTPRoute、GRPCRoute 等）必须匹配监听器允许的路由类型。

对于更复杂的跨命名空间场景，请参阅绑定到另一个命名空间创建的网关。

在 **YAML** 中

- `sectionName` 是监听器名称。
- 路由只能绑定到支持其特定类型的监听器。
- 默认情况下，路由只能绑定到与 `Gateway` 同一命名空间的监听器。

跨命名空间绑定请参阅绑定到另一个命名空间创建的网关。

后端

定义匹配请求应转发到的目标服务。

每个服务可以有一个 `weight` 字段，用于指定流量分配比例。

主机名

`hostnames` 字段由 `HTTPRoute`、`GRPCRoute` 和 `TLSRoute` 支持。`TCPRoute` 和 `UDPRoute` 不使用此字段。

`hostnames` 是字符串数组，遵循主机名交集规则。

规则

每个路由可以包含多个规则。每条规则由以下部分组成：

匹配

定义请求被此规则路由的条件。

一条规则可以有多个匹配：

- 每个匹配包含多个条件（如路径、头部、查询参数、方法）
- 条件内部使用与（**AND**）逻辑（全部满足）
- 匹配之间使用或（**OR**）逻辑（任一匹配满足即可）

示例：如果匹配1要求 `path=/api` 且 `header=v1`，匹配2要求 `query=test`，则请求满足 `(path=/api 且 header=v1)` 或 `(query=test)` 之一即被路由。

匹配结构在路由类型间通用，但支持的匹配条件依路由类型不同而异。例如，`HTTPRoute` 和 `GRPCRoute` 支持不同的匹配条件集。

过滤器

指定对请求或响应应用的转换或修改。

过滤器概念在路由类型间通用，但支持的过滤器类型依路由类型不同而异。

HTTPRoute 参考

以下匹配条件、过滤器类型和高级选项用于 `HTTPRoute`。

匹配条件类型

对象	方法	值类型	说明	值要求
Path				
	`Exact`	路径（字符串）	精确匹配 URL 路径，区分大小写。即对 /abc 的精确匹配只匹配 /abc，不匹配 /abc/、/Abc 或 /abcd。	必须以 `/` 开头，不允许连续 `//`。
	`PathPrefix`	路径（字符串）	基于 URL 路径前缀匹配，按 `/` 分割。匹配区分大小写，按路径元素逐个匹配。例如，路径 /abc、/abc/ 和 /abc/def 都匹配前缀 /abc，但 /abcd 不匹配。	必须以 `/` 开头，不允许连续 `//`。
	`RegularExpression`	路径（字	正则表达式引擎：RE2。	例如 `/api/v1/.*`。

对象	方法	值类型	说明	值要求
		符串）		
Header				
	`Exact`	名称（头部键）+值	精确匹配头部值。	
	`RegularExpression`	名称（头部键）+值	正则表达式引擎：RE2。	
QueryParam				
	`Exact`	名称（参数键）+值	精确匹配查询参数值。	参数值长度：1-1024 字符
	`RegularExpression`	名称（参数键）+值	正则表达式引擎：RE2。	
Method	-	方法名称	HTTP 方法匹配。	`GET`、`HEAD`、`POST`、`PUT`、`DELETE`、`CONNECT`、`OPTIONS`、`TRACE`、`PATCH`

匹配条件参考

条件类型	官方文档
Path	HTTPPathMatch ↗
Headers	HTTPHeaderMatch ↗
QueryParams	HTTPQueryParamMatch ↗
Method	HTTPMethod ↗

过滤器类型

类型	方法	值类型	说明
RequestHeaderModifier	`Set`	名称（字符串）+值（字符串）	用给定名称和值覆盖请求头
	`Add`	名称（字符串）+值（字符串）	向请求头添加值，追加到现有值
	`Remove`	[]string	从请求中移除指定头部（不区分大小写）
ResponseHeaderModifier	`Set`	名称（字符串）+值（字符串）	用给定名称和值覆盖响应头
	`Add`	名称（字符串）+值（字符串）	向响应头添加值，追加到现有值
	`Remove`	[]string	从响应中移除指定头部（不区分大小写）

类型	方法	值类型	说明
RequestRedirect	`Scheme`	字符串	Location 头的协议（http/https）
	`Hostname`	PreciseHostname	Location 头的主机名
	`ReplaceFullPath`	字符串	替换整个请求路径
	`ReplacePrefixMatch`	字符串	替换匹配的路径前缀
	`Port`	PortNumber	Location 头的端口
	`StatusCode`	整数	HTTP 重定向状态码
URLRewrite	`Hostname`	PreciseHostname	请求中重写的主机名
	`ReplaceFullPath`	字符串	替换整个请求路径
	`ReplacePrefixMatch`	字符串	替换匹配的路径前缀
CORS	`AllowOrigins`	[]string	允许的 CORS 请求来源列表

类型	方法	值类型	说明
	`AllowMethods`	[]HTTPMethod	允许的 HTTP 方法列表
	`AllowHeaders`	[]string	允许的 CORS 请求头列表
	`ExposeHeaders`	[]string	响应中暴露给客户端的头部列表
	`MaxAge`	Duration	CORS 预检响应的缓存时间
	`AllowCredentials`	bool	是否允许 CORS 请求携带凭据

注意：

- `RequestRedirect` 和 `URLRewrite` 不能在同一规则中同时使用
- `ReplacePrefixMatch` 仅兼容 `PathPrefix` 类型的 HTTPRouteMatch
- 头部名称根据 RFC 7230 不区分大小写
- 同一头部的多个值必须使用 RFC 7230 逗号分隔格式

过滤器参考

过滤器类型	官方文档
RequestHeaderModifier	HTTPHeaderFilter
ResponseHeaderModifier	HTTPHeaderFilter
RequestRedirect	HTTPRequestRedirectFilter

过滤器类型	官方文档
URLRewrite	HTTPURLRewriteFilter ↗
CORS	HTTPCORSFilter ↗
RequestMirror	HTTPRequestMirrorFilter ↗
HTTPExternalAuthFilter	HTTPExternalAuthFilter ↗

选项

Options 部分为 `HTTPRoute` 提供了高级流量管理功能，包括超时、重试和会话保持设置。

超时

字段	说明	**YAML 路径**
请求超时	网关在接收客户端完整请求后，完成 HTTP 响应的最大时长。选项：默认（使用默认超时，通常为 15 秒）、无限制（设置为 "0s" 表示无超时）、自定义。	`.spec.rules[].timeouts.request`
后端请求超时	单次网关到后端调用的最大时长，从网关开始发送请求到接收完整后端响应。选项：默认（使用实现特定默认值）、无限制（设置为 "0s"）、自定义。	`.spec.rules[].timeouts.backendRequest`

NOTE

- 请求超时从接收完整客户端请求后开始计时，涵盖整个事务，可能包含多次后端调用（如重试时）。
- 指定时，后端请求超时必须小于等于请求超时。
- 选择“默认”时，字段设为 nil（使用实现默认）。

- 选择“无限制”时，字段设为 "0s"（最大可能值）。

字段	规范
`.spec.rules[].timeouts`	HTTPRouteTimeouts ↗

重试

字段	说明	YAML 路径
状态码	触发重试的 HTTP 状态码（如 503、502）。值范围：400-599。	`.spec.rules[].retry.codes`
尝试次数	重试次数。	`.spec.rules[].retry.attempts`
回退时间	重试前等待时间（如 "100ms"、"1s"）。	`.spec.rules[].retry.backoff`

NOTE

- 默认情况下，重试禁用。如果未配置或留空重试字段，网关不会重试失败请求。
- 必须显式配置重试次数和重试条件，才能启用重试功能。
- 在 Web 控制台配置重试时，若删除所有重试配置项，字段设为 nil。

字段	规范
`.spec.rules[].retry`	HTTPRouteRetry ↗

会话保持

配置会话亲和性，确保来自同一客户端的请求路由到相同后端。

字段	说明	YAML 路径
类型	会话保持类型。选项：Cookie、Header。	`.spec.rules[].sessionPersistence.type`
会话名称	用于会话跟踪的 Cookie 或 Header 名称。	`.spec.rules[].sessionPersistence.sessionName`

字段	规范
`.spec.rules[].sessionPersistence`	SessionPersistence ↗

GRPCRoute 匹配和过滤器参考

以下匹配条件和过滤器类型用于 `GRPCRoute`。

GRPCRoute 匹配

`GRPCRoute` 支持以下匹配类型：

对象	方法	值类型	说明
Method	-	类型（service/method）+值	匹配 gRPC 方法。类型可为 `service`（匹配服务名）或 `method`（匹配方法名）。
Headers	`Exact`	名称（头部键）+值	精确匹配头部值。
	`RegularExpression`	名称（头部键）+值	正则表达式引擎：RE2。

GRPCRoute 过滤器

`GRPCRoute` 仅支持 `RequestHeaderModifier` 过滤器：

类型	方法	值类型	说明	值要求
RequestHeaderModifier	`Set`	名称（字符串）+值（字符串）	用给定名称和值覆盖请求头	最多 16 项，值长度：1-4096 字符
	`Add`	名称（字符串）+值（字符串）	向请求头添加值，追加到现有值	最多 16 项，值长度：1-4096 字符
	`Remove`	[]string	从请求中移除指定头部（不区分大小写）	最多 16 项

> **NOTE**
>
> `GRPCRoute` 不支持超时、重试或会话保持等选项。

TLSRoute 参考

以下行为特定于 `TLSRoute`。

> **NOTE**
>
> - `TLSRoute` 的主机名是可选的。如果监听器有主机名但 `TLSRoute` 没有，则 `TLSRoute` 自动继承监听器的主机名。
> - `TLSRoute` 只能绑定到协议为 `TLS` 且模式为 `Passthrough` 的监听器。

查看

拓扑

以下功能是 ACP Web 控制台提供的额外查看能力。

拓扑标签页提供路由及其关联资源的可视化表示。它显示所有绑定到路由的策略，以及它们依赖的资源，如 `SecurityPolicy` 引用的 secret。

此功能目前仅支持 `HTTPRoute`。

下一步

路由绑定到监听器后，如需高级流量或安全策略，请继续配置配置 GatewayAPI Policy。更多操作示例请参阅 Envoy Gateway 任务。

相关任务

- 如何绑定到其他命名空间创建的监听器

配置 ALB

WARNING

ALB 已被废弃。请改用 ingress-nginx-operator 或 envoy-gateway。

目录

- ALB
  - 前提条件
  - 配置 ALB
    - 资源相关配置
    - 网络配置
    - 项目配置
    - 调整配置
  - ALB 操作
    - 创建
    - 更新
    - 删除
- Frontend
  - 前提条件
  - 配置 Frontend
  - Frontend 操作
    - 创建
  - 后续操作
  - 相关操作

Rule

前提条件

使用 dslx 和优先级匹配请求

dslx

优先级

动作

后端

后端协议

服务组与会话保持策略

Rule 操作

使用 Web 控制台

使用 CLI

Https

Ingress 中的证书注解

Rule 中的证书

TLS 模式

Ingress

Ingress 同步

日志与监控

查看日志

监控指标

ALB

ALB 是表示负载均衡器的自定义资源。alb-operator 默认嵌入所有集群中，监听 ALB 资源的创建/更新/删除操作，并据此创建相应的 Deployment 和 Service。

对于每个 ALB，会有一个对应的 Deployment 监听所有附加到该 ALB 的 Frontends 和 Rules，并根据这些配置将请求路由到后端。

前提条件

负载均衡器 的高可用需要 VIP。请参考 配置 VIP。

配置 ALB

ALB 配置包含三部分。

```yaml
# test-alb.yaml
apiVersion: crd.alauda.io/v2beta1
kind: ALB2
metadata:
  name: alb-demo
  namespace: cpaas-system
spec:
  address: 192.168.66.215
  config:
    vip:
      enableLbSvc: false
      lbSvcAnnotations: {}
    networkMode: host
    nodeSelector:
      cpu-model.node.kubevirt.io/Nehalem: 'true'
    replicas: 1
    resources:
      alb:
        limits:
          cpu: 200m
          memory: 256Mi
        requests:
          cpu: 200m
          memory: 256Mi
      limits:
        cpu: 200m
        memory: 256Mi
      requests:
        cpu: 200m
        memory: 256Mi
    projects:
      - ALL_ALL
  type: nginx
```

资源相关配置

资源相关字段描述 alb 的部署配置。

字段	类型	描述
`.spec.config.nodeSelector`	map[string]string	alb 的节点选择器
`.spec.config.replicas`	int，可选，默认 3	alb 的副本数
`.spec.config.resources.limits`	k8s 容器资源， 可选	alb 中 nginx 容器的资源限制
`.spec.config.resources.requests`	k8s 容器资源， 可选	alb 中 nginx 容器的资源请求
`.spec.config.resources.alb.limits`	k8s 容器资源， 可选	alb 中 alb 容器的资源限制
`.spec.config.resources.alb.requests`	k8s 容器资源， 可选	alb 中 alb 容器的资源请求
`.spec.config.antiAffinityKey`	string，可选，默认 local	k8s 反亲和性键

网络配置

网络字段描述如何访问 ALB。例如，在 `host` 模式下，alb 使用 hostnetwork，可通过节点 IP 访问 ALB。

字段	类型	描述
`.spec.config.networkMode`	string: `host` 或 `container`，可选，默认 `host`	在 `container` 模式下，operator 会创建 LoadBalancer Service，并使用其地址作为 ALB 地址。
`.spec.address`	string，必填	可以手动指定 alb 的地址

字段	类型	描述
`.spec.config.vip.enableLbSvc`	bool，可选	在 `container` 模式下自动为 true。
`.spec.config.vip.lbSvcAnnotations`	map[string]string，可选	LoadBalancer Service 的额外注解

项目配置

字段	类型
`.spec.config.projects`	[]string，必填
`.spec.config.portProjects`	string，可选
`.spec.config.enablePortProject`	bool，可选

将 ALB 添加到项目意味着：

1. 在 Web UI 中，只有该项目内的用户可以查找和配置此 ALB。
2. 该 ALB 会处理属于该项目的 ingress 资源。请参考 ingress-sync。
3. 在 Web UI 中，项目 X 创建的规则无法在项目 Y 下查找或配置。

如果启用端口项目并为项目分配端口范围，则意味着：

1. 不能创建不属于该项目分配端口范围的端口。

调整配置

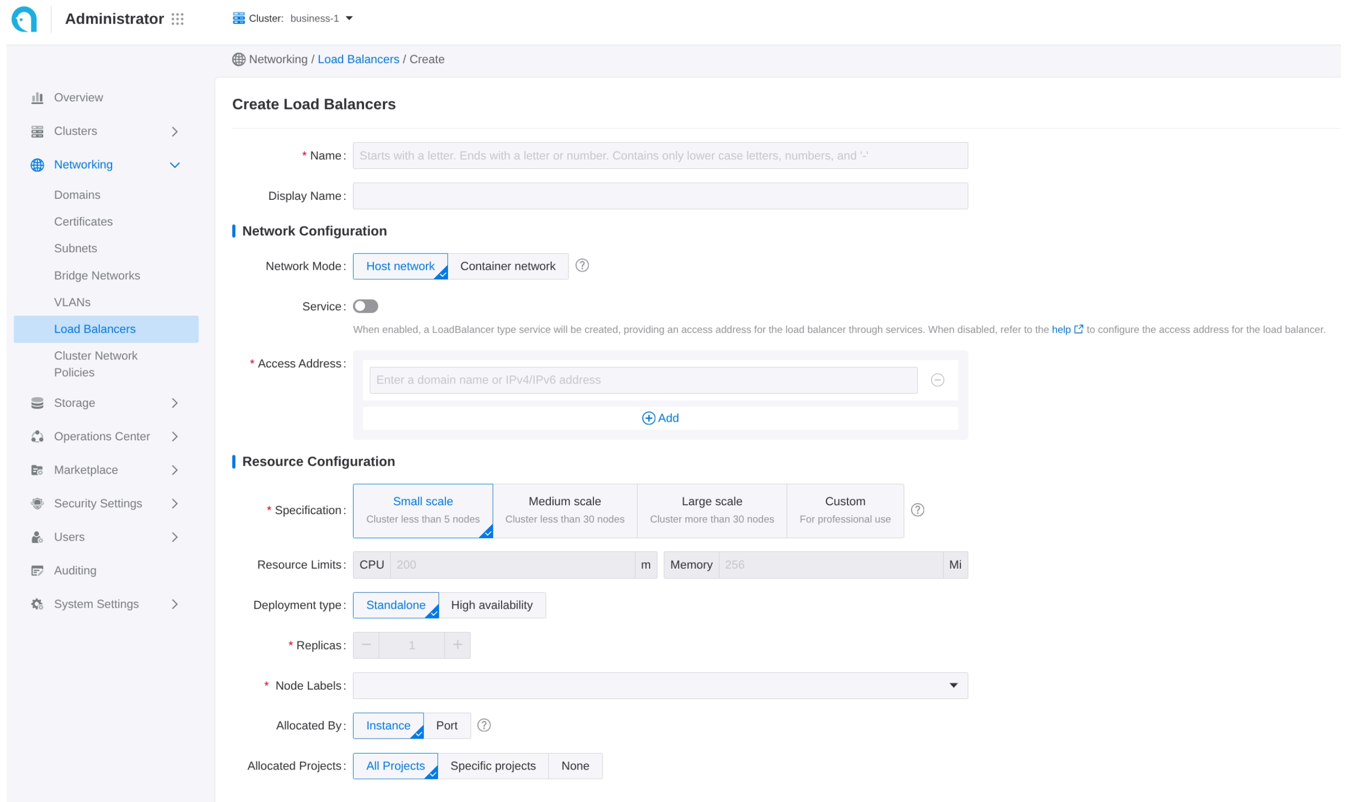
alb cr 中有一些全局配置可以调整。

- 绑定网卡
- Ingress 同步

ALB 操作

创建

使用 **Web** 控制台



Web UI 中暴露了一些常用配置。创建负载均衡器的步骤：

1. 进入 **Administrator**。
2. 左侧边栏点击 **Network Management** > **Load Balancer**。
3. 点击 **Create Load Balancer**。

Web UI 中每个输入项对应 CR 的字段：

参数	描述
Assigned Address	`.spec.address`
Allocated By	`Instance` 表示项目模式，可选择项目； `port` 表示端口项目模式，创建 alb 后可分配端口范围

使用 **CLI**

```
kubectl apply -f test-alb.yaml -n cpaas-system
```

更新

使用 **Web** 控制台

> **NOTE**
>
> 更新负载均衡器会导致 3 到 5 分钟的服务中断，请选择合适时间进行操作！

1. 进入 **Administrator**。
2. 左侧导航栏点击 **Network Management** > **Load Balancer**。
3. 点击 ⋮ > **Update**。
4. 根据需要更新网络和资源配置。
    - 请根据业务需求合理设置规格。也可参考相关文档 如何合理分配 CPU 和内存资源。
    - 内部路由 仅支持从 **Disabled** 状态更新为 **Enabled** 状态。
5. 点击 **Update**。

删除

使用 **Web** 控制台

> **NOTE**
>
> 删除负载均衡器后，相关端口和规则也会被删除且无法恢复。

1. 进入 **Administrator**。
2. 左侧导航栏点击 **Network Management** > **Load Balancer**。
3. 点击 ⋮ > **Delete**，并确认。

使用 **CLI**

```
kubectl delete alb2 alb-demo -n cpaas-system
```

Frontend

Frontend 是定义 ALB 监听端口和协议的自定义资源。 支持协议：L7（http|https|grpc|grpcs）和 L4（tcp|udp）。

- L4 代理直接用 frontend 配置后端服务。
- L7 代理用 frontend 配置监听端口，用 rule 配置后端服务。

如果需要添加 HTTPS 监听端口，还需联系管理员为当前项目分配 TLS 证书以实现加密。

前提条件

先创建 ALB。

配置 Frontend

```yaml
# alb-frontend-demo.yaml
apiVersion: crd.alauda.io/v1
kind: Frontend
metadata:
  labels:
    alb2.cpaas.io/name: alb-demo ①
  name: alb-demo-00080 ②
  namespace: cpaas-system
spec:
  port: 80 ③
  protocol: http ④
  certificate_name: '' ⑤
  backendProtocol: 'http' ⑥
  serviceGroup: ⑦
    session_affinity_policy: '' ⑧
    services:
      - name: hello-world
        namespace: default
        port: 80
        weight: 100
```

① alb 标签：必填，指明该 `Frontend` 所属的 ALB 实例。

② frontend 名称：格式为 `$alb_name-$port`。

③ port：监听的端口。

④ protocol：该端口使用的协议。

- L7 协议 https|http|grpcs|grpc 和 L4 协议 tcp|udp。
- 选择 HTTPS 时必须添加证书；gRPC 协议添加证书为可选。
- 选择 gRPC 协议时，后端协议默认为 gRPC，不支持会话保持。若为 gRPC 协议设置证书，负载均衡器会卸载 gRPC 证书并将未加密的 gRPC 流量转发至后端服务。
- 使用 Google GKE 集群时，同一 容器网络类型 的负载均衡器不能同时拥有 TCP 和 UDP 监听协议。

⑤ certificate_name：用于 grpcs 和 https 协议的默认证书，格式为 `$secret_ns/$secret_name`。

⑥ backendProtocol：后端服务使用的协议。

⑦ 默认 `serviceGroup`：

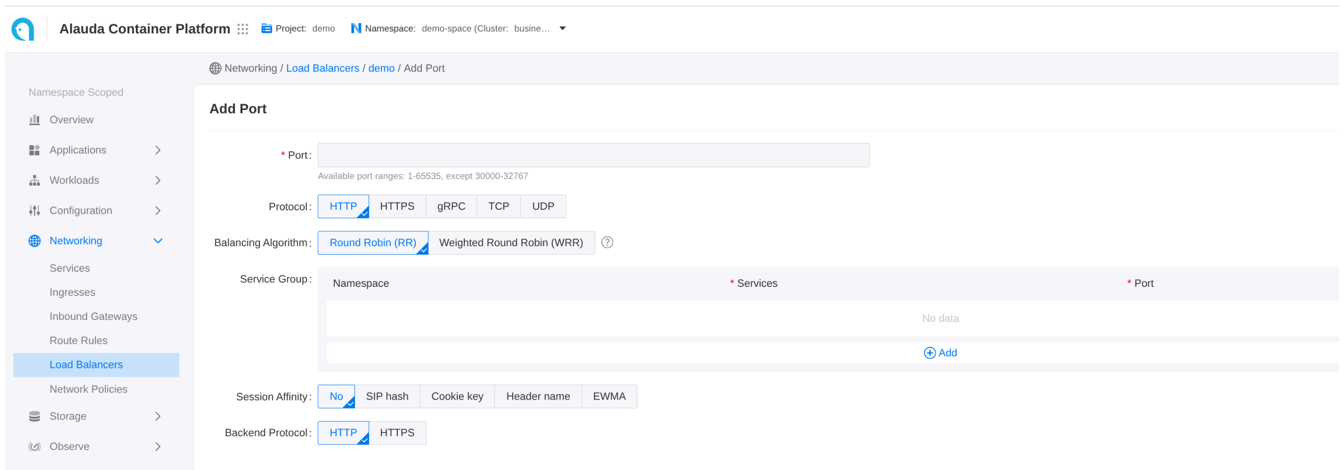
- L4 代理：必填。ALB 直接将流量转发到默认服务组。
- L7 代理：可选。ALB 先匹配该 Frontend 上的 Rules，若无匹配则回退到默认 `serviceGroup`。

8 session_affinity_policy

Frontend 操作

创建

使用 Web 控制台



1. 进入 **Container Platform**。
2. 左侧导航栏点击 **Network** > **Load Balancing**。
3. 点击负载均衡器名称进入详情页。
4. 点击 **Add Port**。

Web UI 中每个输入项对应 CR 的字段：

参数	描述
Session Affinity	`.spec.serviceGroup.session_affinity_policy`

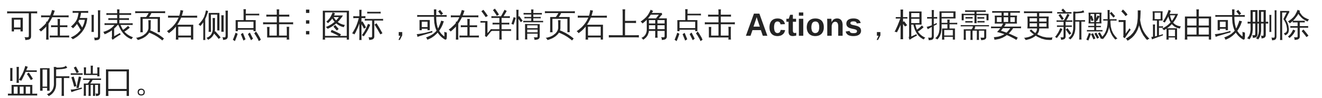
使用 CLI

```
kubectl apply -f alb-frontend-demo.yaml -n cpaas-system
```

后续操作

对于 HTTP、gRPC 和 HTTPS 端口的流量，除了默认的内部路由组外，还可以设置更多不同的后端服务匹配 规则。负载均衡器会优先根据规则匹配对应后端服务，规则匹配失败时再匹配上述内部路由组对应的后端服务。

相关操作



可在列表页右侧点击 ⋮ 图标，或在详情页右上角点击 **Actions**，根据需要更新默认路由或删除监听端口。

> **NOTE**
>
> 若负载均衡器的资源分配方式为 **Port**，则只有管理员可在 **Administrator** 视图中删除相关监听端口。

Rule

Rule 是定义 ALB 如何匹配和处理请求的自定义资源。

ALB 处理的 Ingress 可以 自动转换为规则。

前提条件

- 配置 ALB
- 配置 Frontend

下面是一个示例规则，帮助快速了解规则的使用。

> **NOTE**
>
> 规则必须通过标签附加到 frontend 和 alb。

```yaml
apiVersion: crd.alauda.io/v1
kind: Rule
metadata:
  labels:
    alb2.cpaas.io/frontend: alb-demo-00080 # 1
    alb2.cpaas.io/name: alb-demo # 2
  name: alb-demo-00080-test
  namespace: cpaas-system
spec:
  backendProtocol: 'https' # 3
  certificate_name: 'a/b' # 4
  dslx: # 5
    - type: URL
      values:
        - - STARTS_WITH
          - /
  priority: 4 # 6
  serviceGroup: # 7
    services:
      - name: hello-world
        namespace: default
        port: 80
        weight: 100
```

1. 必填，指明该规则所属的 `Frontend`。
2. 必填，指明该规则所属的 ALB。
3. backendProtocol
4. certificate_name
5. dslx
6. 数字越小优先级越高。
7. serviceGroup

使用 **dslx** 和优先级匹配请求

dslx

DSLX 是用于描述匹配条件的领域专用语言。 例如，下面的规则匹配满足 全部 以下条件的请求：

- url 以 /app-a 或 /app-b 开头
- 请求方法为 post
- url 参数 group 为 vip
- host 以 *.app.com 结尾
- header 中 location 为 east-1 或 east-2
- 有名为 uid 的 cookie
- 源 IP 在 1.1.1.1-1.1.1.100 范围内

```yaml
dslx:
  - type: METHOD
    values:
      - - EQ
        - POST
  - type: URL
    values:
      - - STARTS_WITH
        - /app-a
      - - STARTS_WITH
        - /app-b
  - type: PARAM
    key: group
    values:
      - - EQ
        - vip
  - type: HOST
    values:
      - - ENDS_WITH
        - .app.com
  - type: HEADER
    key: LOCATION
    values:
      - - IN
        - east-1
        - east-2
  - type: COOKIE
    key: uid
    values:
      - - EXIST
  - type: SRC_IP
    values:
      - - RANGE
        - '1.1.1.1'
        - '1.1.1.100'
```

优先级

优先级为 0 到 10 的整数，数字越小优先级越高。Ingress 中配置规则优先级可使用如下注解格式：

```
# alb.cpaas.io/ingress-rule-priority-$rule_index-$path_index
alb.cpaas.io/ingress-rule-priority-0-0: '10'
```

规则中直接在 `.spec.priority` 使用整数设置优先级。

动作

请求匹配规则后，可以对请求应用以下动作：

功能	描述	链接
Timeout	配置请求超时设置	timeout
Redirect	将请求重定向到指定 URL	redirect
CORS	启用跨域资源共享（CORS）	cors
Header Modification	允许修改请求或响应头	header modification
URL Rewrite	转发前重写请求 URL	url-rewrite
WAF	集成 Web 应用防火墙（WAF）	waf
OTEL	启用 OpenTelemetry (OTEL) 分布式追踪与监控	otel
Keepalive	启用或禁用应用的 keepalive 功能	keepalive

后端

后端协议

默认后端协议为 HTTP。若需使用 TLS 重新加密，可配置为 HTTPS。

服务组与会话保持策略

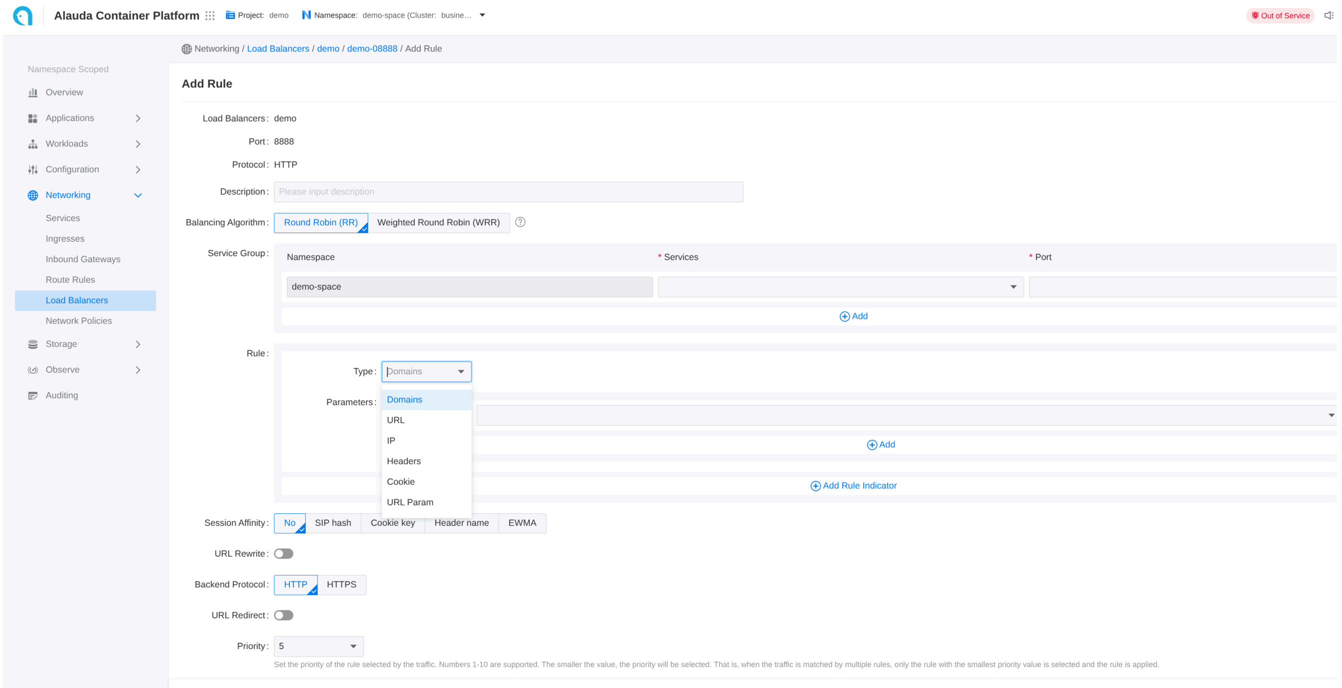
规则中可配置一个或多个服务。

默认情况下，ALB 使用轮询（RR）算法在后端服务间分发请求。也可为单个服务分配权重或选择其他负载均衡算法。

详情请参考 负载均衡算法。

Rule 操作

使用 Web 控制台



1. 进入 **Container Platform**。
2. 左侧导航栏点击 **Network** > **Load Balancing**。
3. 点击负载均衡器名称。
4. 选择监听端口名称。
5. 点击 **Add Rule**。
6. 参考下述描述配置相关参数。
7. 点击 **Add**。

Web UI 中每个输入项对应 CR 的字段。

使用 CLI

```
kubectl apply -f alb-rule-demo.yaml -n cpaas-system
```

Https

若 frontend 协议（ft）为 HTTPS 或 GRPCS，规则也可配置为使用 HTTPS。

可在规则或 ingress 中指定证书，以匹配该端口的证书。

支持终止（Termination），若后端协议为 HTTPS，则支持重新加密。但不能为与后端服务通信指定证书。

Ingress 中的证书注解

证书可通过注解跨命名空间引用。

```
alb.networking.cpaas.io/tls: qq.com=cpaas-system/dex.tls,qq1.com=cpaas-sy
stem/dex1.tls
```

Rule 中的证书

在 `.spec.certificate_name` 中，格式为 `$secret_namespace/$secret_name`。

TLS 模式

Edge 模式

Edge 模式下，客户端与 ALB 使用 HTTPS 通信，ALB 与后端服务使用 HTTP 协议。 实现方式：

1. 创建使用 https 协议的 ft
2. 创建后端协议为 http 的 rule，并通过 `.spec.certificate_name` 指定证书

重新加密模式

重新加密模式下，客户端与 ALB 使用 HTTPS 通信，ALB 与后端服务使用 HTTPS 协议。 实现方式：

1. 创建使用 https 协议的 ft
2. 创建后端协议为 https 的 rule，并通过 `.spec.certificate_name` 指定证书

Ingress

Ingress 同步

每个 ALB 创建一个同名的 IngressClass，处理同项目内的 ingress。

当 ingress 命名空间带有标签 `cpaas.io/project: demo`，表示该 ingress 属于 `demo` 项目。

ALB 的 `.spec.config.projects` 配置中包含项目名 `demo` 时，会自动将这些 ingress 转换为规则。

> **NOTE**
>
> ALB 监听 ingress 并自动创建 `Frontend` 或 Rule。`source` 字段定义如下：
>
> 1. `spec.source.type` 当前仅支持 `ingress`。
> 2. `spec.source.name` 为 ingress 名称。
> 3. `spec.source.namespace` 为 ingress 命名空间。

SSL 策略

对于未配置证书的 ingress，ALB 提供使用默认证书的策略。

可在 ALB 自定义资源中配置：

- `.spec.config.defaultSSLStrategy`：定义无证书 ingress 的 SSL 策略
- `.spec.config.defaultSSLCert`：设置默认证书，格式为 `$secret_ns/$secret_name`

可用的 SSL 策略：

- **Never**：不在 HTTPS 端口创建规则（默认行为）
- **Always**：使用默认证书在 HTTPS 端口创建规则

日志与监控

结合日志和监控数据，可快速定位和解决负载均衡器问题。

查看日志

1. 进入 **Administrator**。
2. 左侧导航栏点击 **Network Management** > **Load Balancer**。
3. 点击 *负载均衡器名称*。
4. 在 **Logs** 标签页，从容器视角查看负载均衡器运行日志。

监控指标

> **NOTE**
>
> 负载均衡器所在集群必须部署监控服务。

1. 进入 **Administrator**。
2. 左侧导航栏点击 **Network Management** > **Load Balancer**。
3. 点击 *负载均衡器名称*。
4. 在 **Monitoring** 标签页，从节点视角查看负载均衡器的指标趋势信息。
    - 使用率：负载均衡器在当前节点的 CPU 和内存实时使用情况。
    - 吞吐量：负载均衡器实例的整体进出流量。

更多监控指标详情请参考 ALB 监控。

配置 NodeLocal DNSCache

目录

Overview
Key Features
Important Notes
Installation
通过 Marketplace 安装
How It Works
架构
Configuration
网络策略配置

Overview

NodeLocal DNSCache 是一个集群插件，通过在集群节点上运行 DNS 缓存代理来提升集群 DNS 性能。该插件通过在每个节点本地缓存 DNS 响应，减少对中央 DNS 服务的负载，从而降低 DNS 查询延迟并提升集群稳定性。

Key Features

- 本地 **DNS** 缓存：在每个节点本地缓存 DNS 响应，减少查询延迟
- 性能提升：显著缩短应用的 DNS 查询时间

Important Notes

WARNING

部署注意事项：

1. **Kube-OVN Underlay** 模式：该插件不支持在 Kube-OVN Underlay 模式下部署，若部署可能导致 DNS 查询失败。
2. **Kubelet** 重启：部署该插件会导致 kubelet 重启。
3. **Pod** 需重启生效：插件部署成功后不会影响正在运行的 Pod，仅对新创建的 Pod 生效。当 CNI 为 Kube-OVN 时，需要手动在 kube-ovn-controller 中添加参数 "--node-local-dns-ip=(本地 DNS 缓存服务器的 IP 地址)"。
4. **NetworkPolicy** 配置：如果集群配置了 NetworkPolicy，需要在 networkPolicy 中额外允许 node CIDR 和 nodeLocalDNSIP 的双向通信，确保正常通信。
5. **MicroOS** 集群升级：MicroOS 集群通过重建集群方式升级，会导致 kubelet 配置丢失。为使 NodeLocal DNS 配置在升级后保持持久，需要在集群模板的以下三个位置的 `kubeletExtraArgs` 中添加 `--cluster-dns` 参数：
   - `KubeadmControlPlane` → `initConfiguration` → `nodeRegistration` → `kubeletExtraArgs`
   - `KubeadmControlPlane` → `joinConfiguration` → `nodeRegistration` → `kubeletExtraArgs`
   - `KubeadmConfigTemplate` → `template` → `spec` → `joinConfiguration` → `nodeRegistration` → `kubeletExtraArgs`

   在上述每个 `kubeletExtraArgs` 中添加如下参数：

   ```
   cluster-dns: "<NodeLocal_DNS_IP>"  # 例如 169.254.20.10
   ```

WARNING

4.2.x 升级注意事项

当从低于 4.2.0（不含 4.2.0）版本升级到 4.2.x 版本时，由于 ResourcePatch 兼容性问题，需要执行以下步骤：

升级前：

- 记录 kube-ovn-controller ResourcePatch 配置中的 `--node-local-dns-ip` 参数值
- 删除 `deploy/kube-ovn-controller` 资源的 ResourcePatch

升级后：

- 手动将记录的 `--node-local-dns-ip` 参数添加回 kube-ovn-controller 配置

注意：该兼容性问题在 4.3 及以上版本已修复，升级到 4.3+ 无需手动干预。

Installation

通过 Marketplace 安装

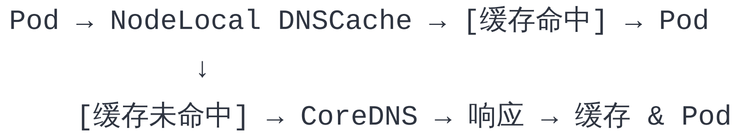
1. 进入 管理员 > **Marketplace** > 集群插件。
2. 在插件列表中搜索 "**Alauda Build of NodeLocal DNSCache**"。
3. 点击 安装，打开安装配置页面。
4. 配置所需参数：

参数	说明	示例值
IP	节点本地 DNS 缓存服务器的 IP 地址。IPv4 推荐使用 169.254.0.0/16 范围内的地址，优选 169.254.20.10。IPv6 推荐使用 fd00::/8 范围内的地址，优选 fd00::10。	169.254.20.10

5. 查看部署注意事项，确保环境满足要求。
6. 点击 安装 完成安装。
7. 等待插件状态变为 "**Ready**"。

How It Works

架构



```
Pod → NodeLocal DNSCache → [缓存命中] → Pod
          ↓
    [缓存未命中] → CoreDNS → 响应 → 缓存 & Pod
```

Configuration

网络策略配置

> 重要：如果集群启用了 NetworkPolicy，必须配置相应规则允许 DNS 流量访问 NodeLocal DNSCache。否则 Pod 可能无法解析 DNS 查询。

使用 NetworkPolicy 时，确保允许以下 DNS 流量：

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-cache
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 169.254.20.10/32 # NodeLocal DNS IP 地址
      ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
  egress:
    - to:
        - ipBlock:
            cidr: 169.254.20.10/32 # NodeLocal DNS IP 地址
      ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
```

配置 CoreDNS

目录

Overview
Configuration
Host Alias
Node Selectors
Node Tolerations

Overview

CoreDNS 是 Kubernetes 集群的默认 DNS 服务。本指南介绍如何为 CoreDNS 配置主机别名、节点选择器和容忍度。

Configuration

1. 进入 管理员 > **Marketplace** > 集群插件。
2. 搜索“**Alauda Build of CoreDNS**”，然后点击 更新。
3. 配置以下参数：

Host Alias

配置自定义 DNS 解析条目。

Parameter	Description
IP	需要解析的 IP 地址
Domains	域名（以空格分隔） 示例：`example.com test.example.com`

Node Selectors

指定 CoreDNS Pod 应运行的节点。

Parameter	Description
Label Key	用于匹配节点的标签键
Label Value	用于匹配节点的标签值

Node Tolerations

允许 CoreDNS Pod 调度到带有污点的节点上。

Parameter	Description
Key	需要容忍的污点键
Value	污点值（可选）
Type	`NoSchedule`、`PreferNoSchedule` 或 `NoExecute`

4. 点击 更新 以应用配置。

实用指南

Ingress-Nginx 任务

Ingress-Nginx 任务

前提条件

最大连接数

请求超时

会话亲和性（粘性会话）

头部修改

URL 重写

HSTS（HTTP 严格传输安全）

限流

WAF

转发头控制

HTTPS

保留源 IP

Envoy Gateway 任务

Envoy Gateway 任务

概述

前提条件

高级任务

相关文档

更多配置

软数据中心 LB 方案（Alpha）

软数据中心 LB 方案（Alpha）

前提条件

操作步骤

验证

Kube OVN

了解 Kube-OVN CNI

上游 OVN/OVS 组件

核心 Controller 和 Agent

监控、运维工具及扩展组件

准备 Kube-OVN Underlay 物理网

使用说明

术语说明

环境要求

配置示例

Kube-OVN U

Overview

前提条件

配置步骤

集群互联（Alpha）

支持配置基于 Kube-OVN 网络模式的集群

前提条件

构建多节点 Kube-OVN 互联控制器

在 global 集群中部署集群互联控制器

加入集群互联

配置 Kube-OVN 网络以支持 Pod 多网卡（Alpha

安装 Multus CNI

创建子网

创建多网卡 Pod

验证双网卡创建

相关操作

其他功能

配置集中式网关

配置 IPPool

配置 MTU

默认 MTU 行为

自定义 MTU 设置

Underlay 和 Overlay 子网的自动互联

weight: 13

操作步骤

启用 u2oInterconnection 的 Underlay 子网间隔离

配置 Endpoint Health Checker

配置 Endpoint Health Checker

Overview

Key Features

Installation

How It Works

How To Activate

Uninstallation

alb

Tasks for ALB

如何为 alb-operator 设置 NodeSelector 和 Tolerations

如何为 alb 设置 NodeSelector 和 Tolerations

任务：从 OCP Route 迁移到 GatewayAPI Route

任务：从 OCP Route 迁移到 GatewayAPI Route

介绍

前提条件

基础 HTTP Route

路由超时

HTTP 严格传输安全 (HSTS)

基于 Cookie 的会话亲和

基于路径的路由

头部修改

连接限制

速率限制

IP 允许列表/阻止列表

URL 重写

跨命名空间路由准入

默认 TLS 证书用于 Ingress

使用自定义 CA 的 TLS 重新加密

使用自定义证书的边缘终止

TLS 透传

功能对比总结

迁移策略

相关文档

Ingress-Nginx 任务

目录

- 前提条件
- 最大连接数
- 请求超时
- 会话亲和性（粘性会话）
- 头部修改
- URL 重写
- HSTS（HTTP 严格传输安全）
- 限流
- WAF
- 转发头控制
- HTTPS
  - TLS 重新加密并验证后端证书
  - TLS 边缘终止
  - 透传
  - 默认证书
  - 在 IngressNginx 中添加 Pod 注解
- 保留源 IP
  - 通过 HAProxy Proxy Protocol
    - 工作原理
    - 配置方法
  - 通过 MetalLB 配合 externalTrafficPolicy=Local
    - 工作原理

配置方法

前提条件

安装 ingress-nginx

最大连接数

Max-Worker-Connections

请求超时

配置请求超时

会话亲和性（粘性会话）

配置粘性会话

头部修改

操作	链接
在请求中设置头部	proxy-set-header
在请求中移除头部	在请求中设置空头部
在响应中设置头部	使用 configuration-snippets 配合 more-set-header 指令
在响应中移除头部	hide-headers

URL 重写

rewrite ↗

HSTS（HTTP 严格传输安全）

配置 HSTS ↗

限流

配置限流 ↗

WAF

modsecurity ↗

转发头控制

x-forwarded-prefix-header ↗

HTTPS

TLS 重新加密并验证后端证书

验证后端 https 证书 ↗

TLS 边缘终止

backend protocol ↗

透传

ssl-passthrough ↗

默认证书

使用以下 yaml 部署带有默认证书的 ingress-nginx

```yaml
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
spec:
  controller:
    extraArgs:
      default-ssl-certificate: $DEFAULT_CERT_NAMESPACE/$DEFAULT_CERT_NAME
```

请参考 default-ssl-certificate ↗

在 **IngressNginx** 中添加 **Pod** 注解

添加 pod 注解

保留源 **IP**

当流量经过负载均衡器或代理时，原始客户端 IP 地址可能因 NAT（网络地址转换）而丢失。保留源 IP 对以下场景非常重要：

- 访问控制和安全策略
- 准确的日志记录和分析
- 基于客户端的限流
- 基于地理位置的路由

通过 **HAProxy Proxy Protocol**

工作原理

PROXY protocol 是一种网络协议，用于在代理 TCP 连接时保留客户端连接信息。它通过在 TCP 连接前添加一个包含原始源 IP 和端口的头部来实现。

流量流程：

1. 客户端连接到 HAProxy 负载均衡器
2. HAProxy 在连接中添加带有原始客户端 IP 的 PROXY protocol 头部
3. Ingress-Nginx 接收连接并解析 PROXY protocol 头部
4. Ingress-Nginx 从头部提取真实客户端 IP
5. 后端应用在 `X-Forwarded-For` 和 `X-Real-IP` 头部中接收到正确的客户端 IP

优点：

- 适用于任何支持 PROXY protocol 的负载均衡器（如 HAProxy、AWS NLB 等）
- 跨多个代理层保留源 IP
- 不影响路由或节点选择

注意事项：

- 负载均衡器和 Ingress-Nginx 都必须配置使用 PROXY protocol
- 一旦启用，所有到 Ingress-Nginx 的流量必须使用 PROXY protocol（混合使用会导致连接失败）

配置方法

配置 HAProxy 负载均衡器发送 PROXY protocol 头部，然后部署启用 proxy-protocol 支持的 ingress-nginx：

```yaml
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
  namespace: ingress-nginx-operator
spec:
  controller:
    config:
      use-proxy-protocol: "true" # 启用 proxy-protocol 支持
```

```
frontend tcp_front_80
    bind *:80
    mode tcp
    default_backend ingress_tcp_80

frontend tcp_front_443
    bind *:443
    mode tcp
    default_backend ingress_tcp_443

backend ingress_tcp_80
    mode tcp
    balance roundrobin
    server node1 192.168.133.46:80 check send-proxy-v2

backend ingress_tcp_443
    mode tcp
    balance roundrobin
    server node1 192.168.133.46:443 check send-proxy-v2
```

更多详情请参见 PROXY protocol 文档↗。

注意：HAProxy 可以使用 TCP 模式转发流量，无需处理 TLS 证书。由于 PROXY protocol 在 TCP 层工作，您可以让 Ingress-Nginx 直接处理 HTTPS 终止和证书管理，无需在 HAProxy 中配置证书。

通过 MetalLB 配合 externalTrafficPolicy=Local

工作原理

使用 Kubernetes Service 类型为 `LoadBalancer` 时，默认行为（`externalTrafficPolicy: Cluster`）会执行源 NAT，将客户端 IP 替换为节点 IP。设置 `externalTrafficPolicy: Local` 通过以下方式保留源 IP：

1. 直接路由：流量仅路由到接收流量的同一节点上的 Pod
2. 无 **SNAT**：kube-proxy 不执行源 NAT，保留原始客户端 IP
3. 健康检查：只有具有健康本地 Pod 的节点才会被包含在负载均衡池中

流量流程：

1. 客户端连接到 MetalLB 虚拟 IP
2. MetalLB 直接将流量路由到有 Ingress-Nginx Pod 的节点
3. 流量直接到达本地 Ingress-Nginx Pod，无 SNAT
4. Ingress-Nginx 获取真实客户端 IP
5. 后端应用在头部中接收正确的客户端 IP

优点：

- 配置简单，无需额外协议
- Kubernetes 原生特性
- 延迟低（无额外代理跳转）

注意事项：

- 负载分布不均：流量只能到达有本地 Pod 的节点，可能导致负载不均衡
- **Pod** 调度：Ingress-Nginx Pod 必须调度到 MetalLB 可路由的节点（使用 nodeSelector 保证一致）
- 健康检查行为：如果所有本地 Pod 不健康，节点将完全从负载均衡中移除

配置方法

部署带有 `externalTrafficPolicy: Local` 的 ingress-nginx，并确保 Pod 调度与 MetalLB 配置一致：

```yaml
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
  namespace: ingress-nginx-operator
spec:
  controller:
    service:
      type: LoadBalancer                    # 使用 MetalLB 提供 LoadBalancer 服务
      externalTrafficPolicy: Local          # 通过仅路由到本地 Pod 保留源 IP
      annotations:
        metallb.universe.tf/address-pool: demo-pool  # 指定 MetalLB 使用的 IP 地址池
    nodeSelector:                           # 仅在匹配这些标签的节点上调度 Pod。此选择器必须与 MetalLB 地址池的节点选择器匹配
      ingress-nginx: "true"
```

重要：`nodeSelector` 必须与 MetalLB 地址池配置中的节点匹配，确保 Ingress-Nginx Pod 调度到 MetalLB 能路由到的节点。

更多详情请参见 externalTrafficPolicy 文档↗。

Envoy Gateway 任务

目录

- 概述
- 前提条件
- 高级任务
  - OpenTelemetry (OTel)
  - 如何附加到其他命名空间创建的 Listener
  - 如何使用其他命名空间创建的证书
  - 如何使用 SSL 透传
  - 如何修改最低 TLS 版本
  - 如何在使用 NodePort 服务时指定 NodePort
  - 如何在使用 MetalLB 时指定 VIP
  - 如何在 Envoy Gateway 中添加 Pod 注解
  - 如何为 `envoy-gateway-operator` 设置 NodeSelector 和 Tolerations
  - 如何为 `envoy-gateway` 设置 NodeSelector 和 Tolerations
  - 如何为 `envoy-proxy` 设置 NodeSelector 和 Tolerations
  - 如何在 `envoy-proxy` 中使用 `hostNetwork`
    - 方式一：使用端口偏移（默认，推荐）
    - 方式二：使用特权端口，启用 useListenerPortAsContainerPort
  - 如何从集群内部访问 LoadBalancer VIP
- 相关文档
- 更多配置

概述

本文档扩展了主配置路径（operator → gateway → route → policy），介绍了超出之前文档涵盖的标准资源配置之外的高级任务。

在 Gateway API 中应用配置变更时，主要有三种方式：

1. 修改标准 **Gateway API** 资源：直接编辑 `Gateway`、`HTTPRoute`、`TCPRoute`、`UDPRoute` 等核心资源。
2. 通过 **PolicyAttachment** 附加策略：使用 `SecurityPolicy`、`ClientTrafficPolicy`、`BackendTrafficPolicy` 等策略资源扩展 Gateway 和 Route 的行为。
3. 配置全局设置：修改 `EnvoyGatewayCtl`，改变 `envoy-gateway` 实例行为或影响所有网关的其他全局设置。

本文档聚焦于主配置路径之外的高级任务和特殊场景，包括跨命名空间路由、可观测性设置、部署定制和故障排查。

前提条件

1. 配置 EnvoyGatewayCtl
2. 配置 Gateway
3. 配置 Route
4. 配置 GatewayAPI 策略

高级任务

OpenTelemetry (OTel)

请按照 OpenTelemetry 集成 ↗ 中的说明操作，但使用 `EnvoyGatewayCtl` 修改 `envoy-gateway-config`。

如何附加到其他命名空间创建的 **Listener**

在 Gateway 的 listener 配置中，需要指定允许哪些命名空间将 Routes 附加到该 Listener。

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: example-gateway
spec:
  listeners:
    - name: http-80
      protocol: HTTP
      port: 80
      allowedRoutes:
        namespaces:
          from: All # 不限制
    - name: http-81
      protocol: HTTP
      port: 81
      allowedRoutes:
        namespaces:
          from: Same # 仅允许同命名空间的 Routes
    - name: http-82
      protocol: HTTP
      port: 82
      allowedRoutes:
        namespaces:
          from: Selector
          selector:
            matchLabels:
              team: frontend # 仅允许带有标签 team=frontend 的命名空间中的 Rou
tes
```

详情请参阅 跨命名空间路由↗。

如何使用其他命名空间创建的证书

若要使用其他命名空间创建的证书，需要在证书所在命名空间创建 `ReferenceGrant`。请参考 跨命名空间证书引用↗ 和 referencegrant↗ 的说明。

NOTE

不能指定单个 `secret` 资源，必须允许整个命名空间

如何使用 **SSL** 透传

请参考以下文档：

- tls
- tls-passthrough

如何修改最低 **TLS** 版本

请参考 自定义 Gateway TLS 参数

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: ClientTrafficPolicy
metadata:
  name: enforce-tls-13
  namespace: default
spec:
  targetRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: eg
  tls:
    minVersion: "1.3"
EOF
```

`ClientTrafficPolicy` 中的 `.spec.tls` 字段是 clienttlssettings。如果除了最低 TLS 版本外还需要自定义密码套件，请参考同一上游任务和 API 参考。

如何在使用 **NodePort** 服务时指定 **NodePort**

使用 NodePort 服务时，Kubernetes 会为每个服务端口分配一个 NodePort。通过节点 IP 访问服务时，应使用分配的 NodePort，而非服务端口。

有两种方式：

手动获取 NodePort 分配，参考 从 svc 端口获取 nodeport。

在 `EnvoyProxy` 配置中手动指定 NodePort，而非让 Kubernetes 自动分配。

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
spec:
  ipFamily: DualStack
  provider:
    kubernetes:
      envoyDeployment:
        container:
          imageRepository: registry.alauda.cn:60080/acp/envoyproxy/envoy
      envoyService:
        patch: # 1
          type: StrategicMerge
          value:
            spec:
              ports:
                - nodeport: 31888
                  port: 80
        type: NodePort
    type: Kubernetes
```

1. 使用 patch 字段对生成的 Service 资源进行补丁，指定 NodePort

> **NOTE**
>
> NodePort 只能在特定范围内，通常为 `30000-32767`。若希望 Gateway listener 端口与 NodePort 一致，listener 端口也必须在 NodePort 范围内。

如何在使用 **MetalLB** 时指定 **VIP**

使用 MetalLB 作为 LoadBalancer 提供者时，可以通过 Service 注解为 Gateway 服务指定静态 VIP。

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    type: Kubernetes
    kubernetes:
      envoyService:
        type: LoadBalancer
        annotations: # 1
          metallb.universe.tf/address-pool: production # 2
          metallb.universe.tf/loadBalancerIPs: VIP_IP # 3
```

1. 在 `envoyService.annotations` 字段添加 MetalLB 注解
2. 指定地址池名称以从中分配 IP
3. 或指定具体 IP 地址（必须在地址池范围内）

可用注解：

注解	说明
`metallb.universe.tf/address-pool`	选择用于分配 IP 的地址池
`metallb.universe.tf/loadBalancerIPs`	指定具体 IP 地址（支持多个，逗号分隔）

> **NOTE**
>
> - 指定的 IP 必须在已配置的 MetalLB 地址池内
> - 确保 MetalLB 已正确安装和配置后再指定 VIP
> - 有关 MetalLB 配置，请参阅 配置 MetalLB

如何在 Envoy Gateway 中添加 Pod 注解

添加 Pod 注解

如何为 `envoy-gateway-operator` 设置 NodeSelector 和 Tolerations

更新 `Subscription` 资源。

```
# nodeSelector 和 tolerations 示例
kubectl patch subscription envoy-gateway-operator  -n envoy-gateway-opera
tor   --type='merge' -p '
{
  "spec": {
    "config": {
      "nodeSelector": {
        "node-role.kubernetes.io/infra": ""
      },
      "tolerations": [
        {
          "effect": "NoSchedule",
          "key": "node-role.kubernetes.io/infra",
          "operator": "Equal",
          "value": "reserved"
        }
      ]
    }
  }
}'
```

如何为 `envoy-gateway` 设置 NodeSelector 和 Tolerations

更新 `EnvoyGatewayCtl` 资源。

```
# 默认情况下 $NAME=cpaas-default, $NS=envoy-gateway-operator
kubectl patch envoygatewayctl $NAME -n $NS --type='merge' -p '
{
  "spec": {
    "deployment": {
      "pod": {
        "nodeSelector": {
          "node-role.kubernetes.io/infra": ""
        },
        "tolerations": [
          {
            "effect": "NoSchedule",
            "key": "node-role.kubernetes.io/infra",
            "operator": "Equal",
            "value": "reserved"
          }
        ]
      }
    }
  }
}'
```

如何为 `envoy-proxy` 设置 NodeSelector 和 Tolerations

更新 `EnvoyProxy` 资源。

```
kubectl patch envoyproxy $NAME -n $NS --type='merge' -p '
{
  "spec": {
    "provider": {
      "kubernetes": {
        "envoyDeployment": {
          "pod": {
            "nodeSelector": {
              "node-role.kubernetes.io/infra": ""
            },
            "tolerations": [
              {
                "effect": "NoSchedule",
                "key": "node-role.kubernetes.io/infra",
                "operator": "Equal",
                "value": "reserved"
              }
            ]
          }
        }
      }
    }
  }
}'
```

如何在 `envoy-proxy` 中使用 `hostNetwork`

使用 `hostNetwork: true` 允许 Envoy proxy Pod 直接使用宿主机网络命名空间。适用于：

- 提升网络性能
- 通过节点 IP 直接访问网关

注意事项：

- 使用 hostNetwork 的 Pod 会直接绑定宿主机网络接口
- 若多个 Pod 在同一节点使用相同端口，可能发生端口冲突
- 安全隔离降低，Pod 共享宿主机网络命名空间
- 建议使用 `nodeSelector` 或 `affinity` 规则控制 Pod 调度，避免端口冲突

配置 `hostNetwork` 有两种方式，取决于是否需要直接使用特权端口（< 1024）：

方式一：使用端口偏移（默认，推荐）

这是默认且推荐的方式。Envoy Gateway 会自动对特权端口加上 10000 的偏移，避免需要特殊权限。

优点：

- 无需特殊权限或能力
- 更安全，作为非 root 用户运行，无额外权限
- 配置简单
- 开箱即用

缺点：

- 客户端需使用偏移端口（10080、10443）

配置示例：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    type: Kubernetes
    kubernetes:
      envoyDeployment:
        patch:
          type: StrategicMerge
          value:
            spec:
              template:
                spec:
                  hostNetwork: true                      # 启用 hostNetwork 模式
                  dnsPolicy: ClusterFirstWithHostNet    # 保证 DNS 正确解析
        pod:
          nodeSelector:                                  # 推荐：控制 Pod 调度避免冲突
            kubernetes.io/hostname: "demo"
```

访问方式：

- 端口 80 → 通过 `http://<node-ip>:10080` 访问
- 端口 443 → 通过 `https://<node-ip>:10443` 访问

方式二：使用特权端口，启用 useListenerPortAsContainerPort

此方式允许 Envoy 直接绑定特权端口（< 1024），如 80 和 443。

优点：

- 可直接使用标准端口 80 和 443
- 与期望标准端口的客户端兼容性更好

缺点：

- 需要 NET_BIND_SERVICE 能力
- 安全性较端口偏移方式略低
- 配置更复杂

配置示例：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    type: Kubernetes
    kubernetes:
      useListenerPortAsContainerPort: true  # 关闭端口偏移
      envoyDeployment:
        patch:
          type: StrategicMerge
          value:
            spec:
              template:
                spec:
                  hostNetwork: true
                  dnsPolicy: ClusterFirstWithHostNet
                  containers:
                  - name: envoy
                    command:
                      - /usr/local/bin/envoy-with-cap  # 使用带 filecap 的
envoy
                    securityContext:
                      capabilities:
                        add:
                        - NET_BIND_SERVICE  # 绑定特权端口所需
        pod:
          nodeSelector:                             # 推荐：控制 Pod 调度
避免冲突
            kubernetes.io/hostname: "demo"
```

访问方式：

- 端口 80 → 通过 `http://<node-ip>:80` 访问
- 端口 443 → 通过 `https://<node-ip>:443` 访问

如何从集群内部访问 LoadBalancer VIP

默认情况下，Envoy Gateway 创建的 LoadBalancer 服务使用 `externalTrafficPolicy: Local`。该策略保留客户端源 IP，但存在重要限制：来自没有 Envoy Gateway Pod 的集群节点的请求会失败，因为流量不会转发到其他节点。

解决方案 **1**：使用 **Service ClusterIP**（推荐集群内访问）

集群内运行的应用应使用服务的 ClusterIP，而非 LoadBalancer VIP，避免路由限制。

解决方案 **2**：修改为 **Cluster** 流量策略

若需从任意集群节点访问 LoadBalancer VIP，可将 `externalTrafficPolicy` 改为 `Cluster`：

```
kubectl patch envoyproxy $GATEWAY_NAME -n $GATEWAY_NS --type='json' -p='[
  {"op": "replace", "path": "/spec/provider/kubernetes/envoyService/externalTrafficPolicy", "value": "Cluster"}
]'
```

相关文档

- 配置 EnvoyGatewayCtl
- 配置 Gateway
- 配置 Route
- 配置 GatewayAPI 策略

更多配置

请参阅 EnvoyGateway 任务 ↗

软数据中心 LB 方案（Alpha）

通过在集群外创建高可用负载均衡器，部署纯软件数据中心负载均衡器（LB），为多个 ALB 提供负载均衡能力，确保业务稳定运行。支持仅 IPv4、仅 IPv6 或 IPv4 和 IPv6 双栈配置。

目录

前提条件
操作步骤
验证

前提条件

1. 准备两个或以上主机节点作为 LB，建议 LB 节点安装 Ubuntu 22.04 操作系统，以减少 LB 转发流量到异常后端节点的时间。
2. 在所有外部 LB 主机节点预先安装以下软件（本章以两个外部 LB 主机节点为例）：
    - `ipvsadm`
    - `container-runtime`，如 containerd
3. 确保每个主机的 `container-runtime` 开机自启动。
4. 确保每个主机节点的时钟同步。
5. 准备 Keepalived 镜像，用于启动外部 LB 服务；平台已包含该镜像。镜像地址格式如下：`<镜像仓库地址>/tkestack/keepalived:<版本后缀>`。不同版本的版本后缀可能略有差异。可

按以下方式获取镜像仓库地址和版本后缀。本文档以 `build-harbor.alauda.cn/tkestack/keepalived:v3.16.0-beta.3.g598ce923` 为例。

- 在 global 集群中执行 `kubectl get prdb base -o json | jq .spec.registry.address` 获取 镜像仓库地址 参数。
- 在安装包解压目录执行 `cat ./installer/res/artifacts.json |grep keepalived -C 2|grep tag|awk '{print $2}'|awk -F '"' '{print $2}'` 获取 版本后缀。

操作步骤

注意：以下操作需在每个外部 LB 主机节点执行一次，且主机节点的 `hostname` 不得重复。

1. 将以下配置信息添加到文件 `/etc/modules-load.d/alive.kmod.conf`。

```
ip_vs
ip_vs_rr
ip_vs_wrr
ip_vs_sh
nf_conntrack_ipv4
nf_conntrack
ip6t_MASQUERADE
nf_nat_masquerade_ipv6
ip6table_nat
nf_conntrack_ipv6
nf_defrag_ipv6
nf_nat_ipv6
ip6_tables
```

2. 将以下配置信息添加到文件 `/etc/sysctl.d/alive.sysctl.conf`。

```
net.ipv4.ip_forward = 1
net.ipv4.conf.all.arp_accept = 1
net.ipv4.vs.conntrack = 1
net.ipv4.vs.conn_reuse_mode = 0
net.ipv4.vs.expire_nodest_conn = 1
net.ipv4.vs.expire_quiescent_template = 1
net.ipv6.conf.all.forwarding=1
```

3. 使用 `reboot` 命令重启。

4. 创建 Keepalived 配置文件夹。

```
mkdir -p /etc/keepalived
mkdir -p /etc/keepalived/kubecfg
```

5. 按照以下文件中的注释修改配置项，并保存至 `/etc/keepalived/` 文件夹，文件名为 `alive.yaml`。

```
instances:
  - vip: # 可配置多个 VIP
      vip: 192.168.128.118 # VIP 必须不同
      id: 20 # 每个 VIP 的 ID 必须唯一，非必填
      interface: "eth0"
      check_interval: 1 # 非必填，默认 1：执行检查脚本间隔
      check_timeout: 3  # 非必填，默认 3：检查脚本超时时间
      name: "vip-1" # 实例标识，只能包含字母数字和短横线，不能以短横线开头
      peer: [ "192.168.128.116", "192.168.128.75" ] # Keepalived 节点 IP，实际生成的 keepalived.conf 会移除接口上的所有 IP。
      kube_lock:
        kubecfgs: # kube-lock 使用的 kube-config 列表，会依次尝试这些 kubecfg 进行 Keepalived 选主
          - "/live/cfg/kubecfg/kubecfg01.conf"
          - "/live/cfg/kubecfg/kubecfg02.conf"
          - "/live/cfg/kubecfg/kubecfg03.conf"
    ipvs: # option IPVS 配置
      ips: [ "192.168.143.192", "192.168.138.100","192.168.129.100" ] # IPVS 后端，将 k8s master 节点 IP 改为 ALB 节点的节点 IP
      ports: # 配置 VIP 上每个端口的健康检查逻辑
        - port: 80 # 虚拟服务器端口必须与真实服务器端口一致
          virtual_server_config: |
            delay_loop 10  # 对真实服务器执行健康检查的间隔
            lb_algo rr
            lb_kind NAT
            protocol TCP
          raw_check: |
            TCP_CHECK {
                connect_timeout 10
                connect_port 1936
            }
  - vip:
      vip: 2004::192:168:128:118
      id: 102
      interface: "eth0"
      peer: [ "2004::192:168:128:75","2004::192:168:128:116" ]
      kube_lock:
        kubecfgs: # kube-lock 使用的 kube-config 列表，会依次尝试这些 kubecfg 进行 Keepalived 选主
          - "/live/cfg/kubecfg/kubecfg01.conf"
          - "/live/cfg/kubecfg/kubecfg02.conf"
          - "/live/cfg/kubecfg/kubecfg03.conf"
    ipvs:
```

```
      ips: [ "2004::192:168:143:192","2004::192:168:138:100","2004::19
2:168:129:100" ]
      ports:
        - port: 80
          virtual_server_config: |
            delay_loop 10
            lb_algo rr
            lb_kind NAT
            protocol TCP
          raw_check: |
            TCP_CHECK {
                connect_timeout 1
                connect_port 1936
            }
```

6. 在业务集群执行以下命令，检查配置文件中证书的过期时间，确保证书仍有效。证书过期后 LB 功能将不可用，需要联系平台管理员更新证书。

```
openssl x509 -in <(cat /etc/kubernetes/admin.conf | grep client-certifi
cate-data | awk '{print $NF}' | base64 -d ) -noout -dates
```

7. 将 Kubernetes 集群中三个 Master 节点的 `/etc/kubernetes/admin.conf` 文件复制到外部 LB 节点的 `/etc/keepalived/kubecfg` 文件夹，命名带索引，如 `kubecfg01.conf`，并修改这三个文件中的 `apiserver` 节点地址为 Kubernetes 集群的实际节点地址。
   注意：平台证书更新后需重新执行此步骤，覆盖原文件。

8. 检查证书有效性。

   1. 将业务集群 Master 节点的 `/usr/bin/kubectl` 复制到 LB 节点。
   2. 执行 `chmod +x /usr/bin/kubectl` 赋予执行权限。
   3. 执行以下命令确认证书有效。

   ```
   kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg01.conf get node
   kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg02.conf get node
   kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg03.conf get node
   ```

   若返回如下结果，证书有效。

```
kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg01.conf get node
## 输出
NAME              STATUS   ROLES                  AGE     VERSION
192.168.129.100   Ready    <none>                 7d22h   v1.25.6
192.168.134.167   Ready    control-plane,master   7d22h   v1.25.6
192.168.138.100   Ready    <none>                 7d22h   v1.25.6
192.168.143.116   Ready    control-plane,master   7d22h   v1.25.6
192.168.143.192   Ready    <none>                 7d22h   v1.25.6
192.168.143.79    Ready    control-plane,master   7d22h   v1.25.6

kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg02.conf get node
## 输出
NAME              STATUS   ROLES                  AGE     VERSION
192.168.129.100   Ready    <none>                 7d22h   v1.25.6
192.168.134.167   Ready    control-plane,master   7d22h   v1.25.6
192.168.138.100   Ready    <none>                 7d22h   v1.25.6
192.168.143.116   Ready    control-plane,master   7d22h   v1.25.6
192.168.143.192   Ready    <none>                 7d22h   v1.25.6
192.168.143.79    Ready    control-plane,master   7d22h   v1.25.6

kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg03.conf get node
## 输出
NAME              STATUS   ROLES                  AGE     VERSION
192.168.129.100   Ready    <none>                 7d22h   v1.25.6
192.168.134.167   Ready    control-plane,master   7d22h   v1.25.6
192.168.138.100   Ready    <none>                 7d22h   v1.25.6
192.168.143.116   Ready    control-plane,master   7d22h   v1.25.6
192.168.143.192   Ready    <none>                 7d22h   v1.25.6
192.168.143.79    Ready    control-plane,master   7d22h   v1.25.6
```

9. 将 Keepalived 镜像上传至外部 LB 节点，并使用 nerdctl 运行 Keepalived。

```
nerdctl run -dt --restart=always --privileged --network=host -v /etc/keepalived:/live/cfg build-harbor.alauda.cn/tkestack/keepalived:v3.16.0-beta.3.g598ce923
```

10. 在访问 `keepalived` 的节点执行命令：`sysctl -w net.ipv4.conf.all.arp_accept=1`。

验证

1. 执行命令 `ipvsadm -ln` 查看 IPVS 规则，可见业务集群 ALB 适用的 IPv4 和 IPv6 规则。

```
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight        ActiveConn InAc
tConn
TCP  192.168.128.118:80 rr
  -> 192.168.129.100:80           Masq    1      0          0
  -> 192.168.138.100:80           Masq    1      0          0
  -> 192.168.143.192:80           Masq    1      0          0
TCP  [2004::192:168:128:118]:80 rr
  -> [2004::192:168:129:100]:80   Masq    1      0          0
  -> [2004::192:168:138:100]:80   Masq    1      0          0
  -> [2004::192:168:143:192]:80   Masq    1      0          0
```

2. 关闭 VIP 所在的 LB 节点，测试 IPv4 和 IPv6 的 VIP 是否能成功迁移到其他节点，通常在 20 秒内完成。
3. 在非 LB 节点使用 `curl` 命令测试与 VIP 的通信是否正常。

```
curl 192.168.128.118

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed
and working. Further configuration is required.</p>

<p>For online documentation and support please refer to <a href="htt
p://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at <a href="http://nginx.com/">nginx.co
m</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

```
curl -6 [2004::192:168:128:118]:80 -g

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed
and working. Further configuration is required.</p>

<p>For online documentation and support please refer to <a href="htt
p://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at<a href="http://nginx.com/">nginx.com
</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Kube OVN

了解 Kube-OVN CNI

上游 OVN/OVS 组件

核心 Controller 和 Agent

监控、运维工具及扩展组件

准备 Kube-OVN Underlay 物理网

使用说明

术语说明

环境要求

配置示例

Kube-OVN U

Overview

前提条件

配置步骤

集群互联（Alpha）

支持配置基于 Kube-OVN 网络模式的集群ﾌ

前提条件

构建多节点 Kube-OVN 互联控制器

在 global 集群中部署集群互联控制器

加入集群互联

相关操作

配置 Kube-OVN 网络以支持 Pod 多网卡（Alpha

安装 Multus CNI

创建子网

创建多网卡 Pod

验证双网卡创建

其他功能

配置集中式网关

配置 IPPool

配置 MTU

默认 MTU 行为

自定义 MTU 设置

Underlay 和 Overlay 子网的自动互联

weight: 13

操作步骤

启用 u2oInterconnection 的 Underlay 子网间隔离

了解 Kube-OVN CNI

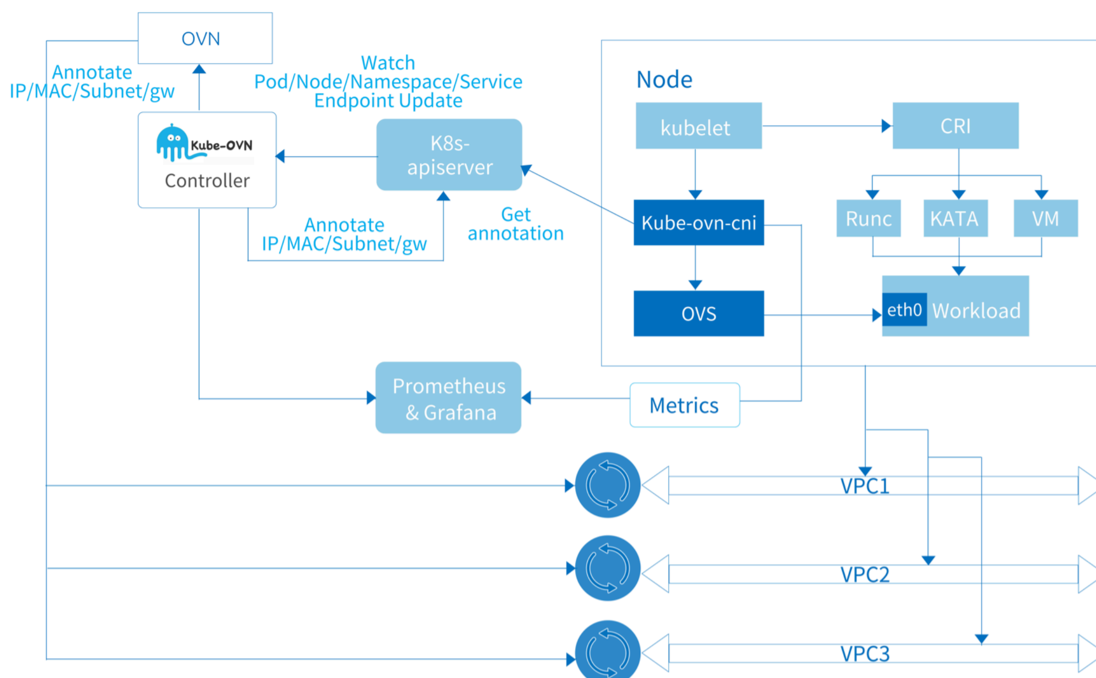
本文档介绍了 Kube-OVN 的整体架构、各组件的功能及其相互作用。

总体来说，Kube-OVN 作为 Kubernetes 与 OVN 之间的桥梁，结合了成熟的 SDN 与云原生技术。 这意味着 Kube-OVN 不仅实现了 Kubernetes 下的网络规范，如 CNI、Service 和 NetworkPolicy， 还将大量 SDN 领域的能力带入云原生环境，如逻辑交换机、逻辑路由器、VPC、 网关、QoS、ACL 以及流量镜像。

Kube-OVN 还保持良好的开放性，能够与多种技术方案集成，如 Cilium、Submariner、Prometheus、KubeVirt 等。

Kube-OVN 的组件大致可以分为三类：

- 上游 OVN/OVS 组件。
- 核心 Controller 和 Agent。
- 监控、运维工具及扩展组件。



目录

- 上游 OVN/OVS 组件
  - ovn-central
  - ovs-ovn
- 核心 Controller 和 Agent
  - kube-ovn-controller
  - kube-ovn-cni
- 监控、运维工具及扩展组件
  - kube-ovn-speaker
  - kube-ovn-pinger
  - kube-ovn-monitor
  - kubectl-ko

上游 **OVN/OVS** 组件

此类组件来自 OVN/OVS 社区，针对 Kube-OVN 的使用场景进行了特定修改。 OVN/OVS 本身是一个成熟的用于管理虚拟机和容器的 SDN 系统， 我们强烈建议对 Kube-OVN 实现感兴趣的用户先阅读 ovn-architecture(7) 以了解 OVN 是什么以及如何与其集成。 Kube-OVN 使用 OVN 的 northbound 接口来创建和协调虚拟网络，并将网络概念映射到 Kubernetes 中。

所有与 OVN/OVS 相关的组件均已打包成镜像，准备在 Kubernetes 中运行。

ovn-central

`ovn-central` Deployment 运行 OVN 的控制平面组件，包括 `ovn-nb`、 `ovn-sb` 和 `ovn-northd`。

- `ovn-nb`：保存虚拟网络配置并提供虚拟网络管理的 API。 `kube-ovn-controller` 主要与 `ovn-nb` 交互以配置虚拟网络。

- `ovn-sb`：保存由 `ovn-nb` 的逻辑网络生成的逻辑流表，以及每个节点的实际物理网络状态。
- `ovn-northd`：将 `ovn-nb` 的虚拟网络转换为 `ovn-sb` 中的逻辑流表。

多个 `ovn-central` 实例通过 Raft 协议同步数据以确保高可用性。

ovs-ovn

`ovs-ovn` 以 DaemonSet 形式运行在每个节点上，Pod 内运行 `openvswitch`、`ovsdb` 和 `ovn-controller`。 这些组件作为 `ovn-central` 的智能体，将逻辑流表转换为真实的网络配置。

核心 Controller 和 Agent

这部分是 Kube-OVN 的核心组件，作为 OVN 与 Kubernetes 之间的桥梁，连接两个系统并转换网络概念。 大部分核心功能都在这些组件中实现。

kube-ovn-controller

该组件负责将 Kubernetes 内的所有资源翻译为 OVN 资源，并作为整个 Kube-OVN 系统的控制平面。 `kube-ovn-controller` 监听所有与网络功能相关资源的事件，并根据资源变化更新 OVN 内的逻辑网络。 主要监听的资源包括：

Pod、Service、Endpoint、Node、NetworkPolicy、VPC、Subnet、Vlan、ProviderNetwork。

以 Pod 事件为例，`kube-ovn-controller` 监听 Pod 创建事件，通过内置的内存 IPAM 功能分配地址， 并调用 `ovn-central` 创建逻辑端口、静态路由及可能的 ACL 规则。 随后， `kube-ovn-controller` 将分配的地址和子网信息（如 CIDR、网关、路由等）写入 Pod 的注解中， 该注解随后被 `kube-ovn-cni` 读取并用于配置本地网络。

kube-ovn-cni

该组件以 DaemonSet 形式运行在每个节点，实现 CNI 接口，操作本地 OVS 配置本地网络。

该 DaemonSet 会将 `kube-ovn` 二进制文件复制到每台机器，作为 `kubelet` 与 `kube-ovn-cni` 交互的工具。 该二进制文件会向 `kube-ovn-cni` 发送相应的 CNI 请求以进行后续操作。默认情况下，该二进制文件会被复制到 `/opt/cni/bin` 目录。

`kube-ovn-cni` 会配置具体网络以执行相应的流量操作， 主要任务包括：

1. 配置 `ovn-controller` 和 `vswitchd`。
2. 处理 CNI Add/Del 请求：
    1. 创建或删除 veth 对并绑定或解绑到 OVS 端口。
    2. 配置 OVS 端口。
    3. 更新主机的 iptables/ipset/路由规则。
3. 动态更新网络 QoS。
4. 创建并配置 `ovn0` 网卡以连接容器网络和主机网络。
5. 配置主机网卡以实现 Vlan/Underlay/EIP。
6. 动态配置集群间网关。

监控、运维工具及扩展组件

这些组件提供监控、诊断、运维工具和外部接口，以扩展 Kube-OVN 的核心网络能力， 并简化日常运维工作。

kube-ovn-speaker

该组件以 DaemonSet 形式运行在特定标签的节点上，发布路由到外部， 允许外部通过 Pod IP 直接访问容器。

kube-ovn-pinger

该组件以 DaemonSet 形式运行在每个节点，收集 OVS 状态信息、节点网络质量、网络延迟等。

kube-ovn-monitor

该组件收集 OVN 状态信息及监控指标。

kubectl-ko

该组件是一个 kubectl 插件，可快速执行常用操作。

准备 Kube-OVN Underlay 物理网络

Kube-OVN Underlay 传输模式下的容器网络依赖于物理网络支持。在部署 Kube-OVN Underlay 网络之前，请与网络管理员协作，提前规划并完成物理网络的相关配置，确保网络连通性。

目录

- 使用说明
- 术语说明
- 环境要求
- 配置示例
  - 交换机配置
  - 检查网络连通性
  - 平台配置

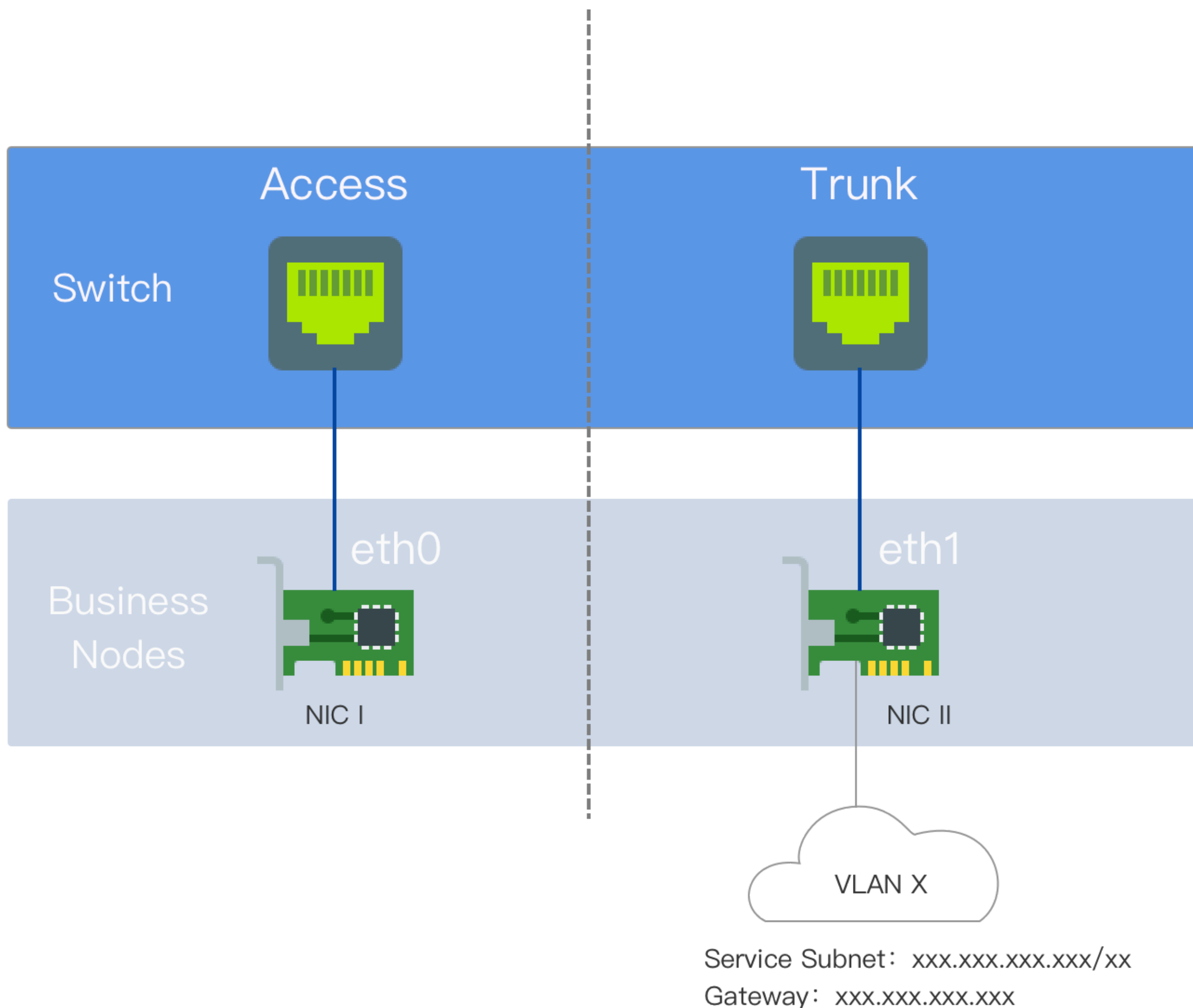
使用说明

Kube-OVN Underlay 需要多网卡（NIC）部署，且 Underlay 子网必须专用一块网卡，该网卡上不能有其他类型的流量（如 SSH），其他流量应使用其他网卡。

使用前，请确保节点服务器至少具备双网卡环境，且建议网卡速率至少为 **10 Gbps** 或更高（如 10 Gbps、25 Gbps、40 Gbps）。

- 网卡一：带默认路由的网卡，配置有 IP 地址，与外部交换机接口互联，交换机端口设置为 Access 模式。

- 网卡二：无默认路由且未配置 IP 地址的网卡，与外部交换机接口互联，交换机端口设置为 Trunk 模式。Underlay 子网专用网卡二。



术语说明

VLAN（虚拟局域网）是一种将局域网逻辑划分为多个段（或更小的局域网）以便虚拟工作组数据交换的技术。

VLAN 技术的出现使管理员能够根据实际应用需求，将同一物理局域网内的不同用户逻辑划分为不同的广播域。每个 VLAN 由一组具有相似需求的计算机工作站组成，具备与物理形成的局域网相同的属性。由于 VLAN 是逻辑划分而非物理划分，同一 VLAN 内的工作站不必局限于同一物理区域，可以分布在不同的物理局域网段。

VLAN 的主要优点包括：

- 端口隔离。即使在同一交换机上，不同 VLAN 的端口之间也无法通信。物理交换机可以作为多个逻辑交换机使用。常用于控制网络中不同部门和站点之间的相互访问。
- 网络安全。不同 VLAN 之间不能直接通信，消除了广播信息的不安全性。VLAN 内的广播和单播流量不会转发到其他 VLAN，有助于控制流量、减少设备投入、简化网络管理并提升网络安全。
- 灵活管理。更改用户网络归属时，无需更换端口或线缆，仅需软件配置变更。

环境要求

在 Underlay 模式下，Kube-OVN 将物理网卡桥接到 OVS，通过该物理网卡直接向外发送数据包。L2/L3 转发能力依赖底层网络设备，需在底层网络设备上预先配置对应的网关、VLAN 和安全策略。

- 网络配置要求
  - Kube-OVN 启动容器时通过 ICMP 协议检测网关连通性，底层网关必须响应 ICMP 请求。
  - 对于服务访问流量，Pod 会先将数据包发送到网关，网关必须具备将数据包转发回本地子网的能力。
  - 当交换机或桥接启用 Hairpin 功能时，必须禁用 **Hairpin**。若使用 VMware 虚拟机环境，需在 VMware 主机上将 **Net.ReversePathFwdCheckPromisc** 设置为 **1**，此时无需禁用 Hairpin。
  - 桥接的网卡不能是 **Linux Bridge**。
  - 网卡绑定模式支持 Mode 0（balance-rr）、Mode 1（active-backup）、Mode 4（802.3ad）、Mode 6（balance-alb），推荐使用 0 或 1。其他绑定模式未测试，请谨慎使用。
- **IaaS**（虚拟化）层配置要求
  - OpenStack 虚拟机环境需关闭对应网络端口的 **PortSecurity**。
  - VMware vSwitch 网络需将 **MAC Address Changes**、**Forged Transmits** 和 **Promiscuous Mode Operation** 均设置为 **Accept**。
  - 公有云如 AWS、GCE、阿里云等因不支持用户自定义 MAC 地址，无法支持 Underlay 模式网络。

配置示例

本示例中的节点为双网卡物理机。网卡一为带默认路由的网卡；网卡二为无默认路由且未配置 IP 地址的网卡，专用于 Underlay 子网，网卡二与外部交换机互联。

- 交换机侧，连接网卡二的接口应配置为 Trunk 模式，允许对应 VLAN 通过。
- 在对应 vlan-interface 接口上配置集群子网的网关地址。如需双栈，可同时配置 IPv6 网关地址。
- 若网关位于防火墙后，需允许节点访问 cluster-cidr 网络。
- 服务器网卡无需配置。

交换机配置

配置 VLAN 接口：

```
#
interface Vlan-interface74
  ip address 192.168.74.254 255.255.255.0   //IPv4 网关地址
  ipv6 address 2074::192:168:74:254/64  //IPv6 网关地址
#
```

配置连接网卡二的接口：

```
#
interface Ten-GigabitEthernet1/0/19
  port link mode bridge
  port link-type trunk  // 配置接口为 Trunk 模式
  undo port trunk permit vlan 1
  port trunk permit vlan 74  // 允许对应 VLAN 通过
#
```

检查网络连通性

测试网卡二是否能与网关地址通信：

```
ip link add ens224.74 link ens224 type vlan id 74  // 网卡名为 ens224, VLAN ID 为 74
ip link set ens224.74 up
ip addr add 192.168.74.200/24 dev ens224.74  // 选择 Underlay 子网内的测试地址, 此处为 192.168.74.200/24
ping 192.168.74.254  // 能 ping 通网关则说明物理环境满足部署要求
ip addr del 192.168.74.200/24 dev ens224.74  // 测试完成后删除测试地址
ip link del ens224.74  // 测试完成后删除子接口
```

平台配置

在左侧导航栏点击 **Cluster Management > Cluster**，然后点击 **Create Cluster**。具体配置流程请参考 Create Cluster 文档，容器网络配置示例如下图所示。

注意：Join 子网在 Underlay 环境中无实际意义，主要用于后续创建 Overlay 子网，提供节点与容器组间通信所需的 IP 地址段。

Kube-OVN Underlay + MetalLB LoadBalancer 服务配置

目录

- Overview
- 前提条件
  - 环境要求
  - 流量流程
- 配置步骤
  - 1. 配置带 VLAN 子接口的 ProviderNetwork
  - 2. 配置 Kube-OVN Controller 参数
  - 3. 配置 Underlay 子网的外部地址功能
  - 4. 创建 MetalLB 外部地址池
  - 5. 创建示例应用及 LoadBalancer 服务
  - 6. 验证配置
  - 7. 迁移现有服务

Overview

该方案解决了 MetalLB L2 模式与 Kube-OVN Underlay 网络的集成问题。它允许用户使用 Underlay 子网 IP 作为 MetalLB LoadBalancer 服务的 VIP，流量直接转发到后端业务 Pod。

> ⚠️ 关键：LoadBalancer VIP 与后端 Pod IP 必须在同一 **Underlay** 子网内。

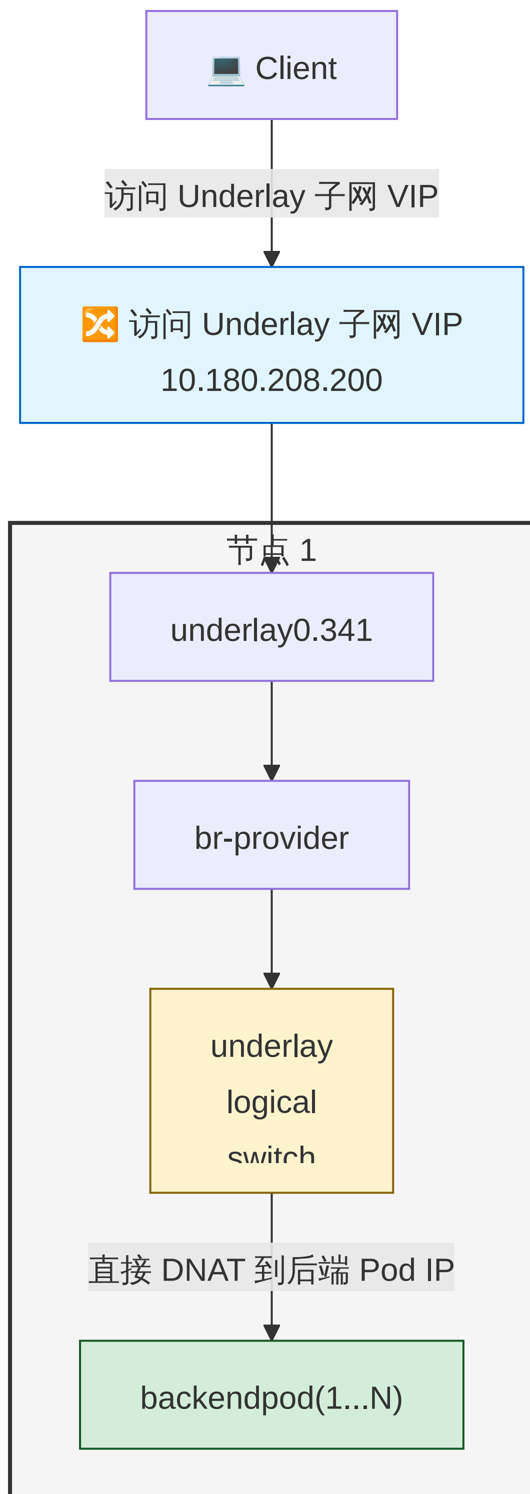
前提条件

环境要求

- **ACP** 版本：>= 4.2.2
- **IP** 支持：仅支持 IPv4（当前不支持 IPv6）

流量流程

流量示意图：



配置步骤

1. 配置带 VLAN 子接口的 ProviderNetwork

重要：必须使用 VLAN 子接口。

配置 Kube-OVN Underlay 网络自动创建 VLAN 子接口：

```yaml
apiVersion: kubeovn.io/v1
kind: ProviderNetwork
metadata:
  name: provider
spec:
  defaultInterface: underlay0.341
  autoCreateVlanSubinterfaces: true  # 如果只有父接口（underlay0），则自动创建 VLAN 子接口（如 underlay0.341）

---
apiVersion: kubeovn.io/v1
kind: Vlan
metadata:
  name: ovn-vlan
spec:
  id: 0    # 使用 0 是因为 autoCreateVlanSubinterfaces 创建了 VLAN 子接口（underlay0.341），负责 VLAN 标记，非 Kube-OVN 直接处理
  provider: provider
status:
  subnets:
  - ovn-default
```

> ⚠️ 警告：单独修改 `ProviderNetwork` 或 `Vlan` 资源时，Underlay 网络连通性会中断。只有当两个资源均配置完成并同步后，网络连通性才会恢复。请在维护窗口内规划配置变更以减少服务中断。

2. 配置 Kube-OVN Controller 参数

为 LoadBalancer 功能配置 Kube-OVN controller 所需参数：

通过 **Web** 控制台操作：

1. 进入 管理员 > **Marketplace** > 集群插件，搜索 `ovn`，找到 **Alauda Container Platform Networking for Kube-OVN**
2. 在插件行点击操作菜单（竖直 ⋮），选择 更新 打开配置对话框
3. 配置以下参数：

- **Skip CT for Dst LPort IPs**：否
- **Enable OVN LB Local**：是

3. 配置 Underlay 子网的外部地址功能

编辑 Underlay 子网，预留 IP 段用于 LoadBalancer 使用：

> 重要：外部地址池 IP 必须在 Underlay 子网内。

修改 Underlay 子网参数 `spec.enableExternalLBAddress: true`：

```yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: underlay-subnet
spec:
  enableExternalLBAddress: true       # 表示该子网有用于 LB 服务 VIP 的 IP 段
  excludeIps:
  - 10.180.208.200..10.180.208.220    # 预留 IP 段作为外部地址池
```

4. 创建 MetalLB 外部地址池

```yaml
# underlay-ippool.yaml
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: acp-underlay-pool
  namespace: metallb-system
spec:
  addresses:
    - 10.180.208.200-10.180.208.220  # Underlay 子网 IP 段
  avoidBuggyIPs: true
  autoAssign: true
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: acp-underlay-pool
  namespace: metallb-system
spec:
  ipAddressPools:
    - acp-underlay-pool
  interfaces:
    - br-provider  # 可选：用于 ARP 广播的接口，建议使用桥接接口（br-*），而非物理接口
  nodeSelectors: []
```

部署地址池：

```
kubectl apply -f underlay-ippool.yaml
```

5. 创建示例应用及 LoadBalancer 服务

```yaml
# application-with-loadbalancer.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-app
  labels:
    app: backend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: backend-lb-service
  # 若使用指定 IPPool：添加注解 `metallb.io/address-pool: acp-underlay-pool`
  # 若使用固定 IP：设置 `spec.loadBalancerIP: 10.180.208.201`
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local  # **重要**：用于保留源 IP 并启用直连 Pod 路由
  selector:
    app: backend
  ports:
    - port: 80
      targetPort: 80
```

部署应用：

```
kubectl apply -f application-with-loadbalancer.yaml
```

6. 验证配置

```
# 查看服务状态
kubectl get svc backend-lb-service -o wide

# 测试外部访问
curl http://10.180.208.200
```

7. 迁移现有服务

对于使用旧地址池（仅节点子网）的现有服务，可迁移至新的 Underlay 地址池：

```
# 为现有服务添加注解以迁移
kubectl annotate service <existing-service-name> metallb.io/address-pool=
acp-underlay-pool --overwrite

# 验证服务已分配新的 Underlay 地址池 IP
kubectl get svc <existing-service-name> -o wide
```

对于新服务，直接添加注解：

```
apiVersion: v1
kind: Service
metadata:
  name: backend-lb-service
  annotations:
    metallb.io/address-pool: acp-underlay-pool  # 使用 Underlay 地址池
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  selector:
    app: backend
  ports:
    - port: 80
      targetPort: 80
```

```
# 查看服务状态
kubectl get svc backend-lb-service -o wide

# 测试外部访问
curl http://10.180.208.200
```

集群互联（Alpha）

它支持配置网络模式与 Kube-OVN 相同的集群之间的互联，使集群中的 Pods 可以相互访问。Cluster Interconnect Controller 是 Kube-OVN 提供的扩展组件，负责收集不同集群之间的网络信息，并通过下发路由连接多个集群的网络。

目录

- 前提条件
- 构建多节点 Kube-OVN 互联控制器
  - Deploy 部署
  - Podman 和 Containerd 部署
- 在 global 集群中部署集群互联控制器
- 加入集群互联
- 相关操作
  - 更新互联集群的网关节点信息
  - 退出集群互联
  - 清理互联集群残留
  - 卸载互联集群
  - 配置集群网关高可用

前提条件

- 不同集群的子网 CIDR 不能相互重叠。

- 需要有一组可被各集群的 kube-ovn-controller 通过 IP 访问的机器，用于部署跨集群互联控制器。
- 每个集群都需要存在一组可被 kube-ovn-controller 通过 IP 访问的机器，以便后续作为网关节点使用。
- 此功能仅适用于默认 VPC，用户自定义 VPC 不能使用互联功能。

构建多节点 Kube-OVN 互联控制器

提供三种部署方式：Deploy 部署（平台 v3.16.0 及以上版本支持）、Podman 部署和 Containerd 部署。

Deploy 部署

注意：平台 v3.16.0 及以上版本支持此部署方式。

操作步骤

1. 在集群 Master 节点上执行以下命令，从 kube-ovn-controller Pod 中获取 `install-ic-server.sh` 安装脚本。

```
kubectl -n kube-system cp $(kubectl get pods -n kube-system -l app=kube-ovn-controller -o custom-columns=NAME:.metadata.name --no-headers | head -1):/kube-ovn/install-ic-server.sh ./install-ic-server.sh
```

2. 打开当前目录下的脚本文件，并按如下方式修改参数。

```
REGISTRY="kubeovn"
VERSION=""
```

修改后的参数配置如下：

```
REGISTRY="<Kube-OVN 镜像仓库地址>"   ## 例如：REGISTRY="registry.alauda.cn:60080/acp/"
VERSION="<Kube-OVN 版本>"   ## 例如：VERSION="v1.9.25"
```

3. 保存脚本文件，并使用以下命令执行。

```
sh install-ic-server.sh
```

Podman 和 Containerd 部署

1. 在任意集群中选择三个或更多节点部署互联控制器。此示例中准备了三个节点。
2. 选择任意节点作为 Leader，并根据不同的部署方式执行以下命令。
   注意：配置前，请检查 `/etc` 下是否存在 ovn 目录。如果不存在，请使用命令 `mkdir /etc/ovn` 创建。

- 容器部署命令 注意：执行命令 `podman images | grep ovn` 获取 Kube-OVN 镜像地址。

  - Leader 节点命令：

    ```
    podman run \
    --name=ovn-ic-db \
    -d \
    --env "ENABLE_OVN_LEADER_CHECK=false" \
    --network=host \
    --restart=always \
    --privileged=true \
    -v /etc/ovn/:/etc/ovn \
    -v /var/run/ovn:/var/run/ovn \
    -v /var/log/ovn:/var/log/ovn \
    -e LOCAL_IP="<当前节点的 IP 地址>" \    ## 例如：-e LOCAL_IP="192.168.39.37"
    -e NODE_IPS="<所有节点的 IP 地址，逗号分隔>" \    ## 例如：-e NODE_IPS="192.168.39.22,192.168.39.24,192.168.39.37"
    <镜像仓库地址> bash start-ic-db.sh    ## 例如：192.168.39.10:60080/acp/kube-ovn:v1.8.8 bash start-ic-db.sh
    ```

  - 另外两个节点的命令：

```
podman run \
--name=ovn-ic-db \
-d \
--env "ENABLE_OVN_LEADER_CHECK=false" \
--network=host \
--restart=always \
--privileged=true \
-v /etc/ovn/:/etc/ovn \
-v /var/run/ovn:/var/run/ovn \
-v /var/log/ovn:/var/log/ovn \
-e LOCAL_IP="<当前节点的 IP 地址>" \   ## 例如：-e LOCAL_IP="192.168.39.24"
-e LEADER_IP="<Leader 节点的 IP 地址>" \  ## 例如：-e LEADER_IP="192.168.39.37"
-e NODE_IPS="<所有节点的 IP 地址，逗号分隔>" \   ## 例如：-e NODE_IPS="192.168.39.22,192.168.39.24,192.168.39.37"
<镜像仓库地址> bash start-ic-db.sh   ## 例如：192.168.39.10:60080/acp/kube-ovn:v1.8.8  bash start-ic-db.sh
```

- **Containerd** 部署命令

  注意：执行命令 `crictl images | grep ovn` 获取 Kube-OVN 镜像地址。

  - Leader 节点命令：

```
ctr -n k8s.io run \
-d \
--env "ENABLE_OVN_LEADER_CHECK=false" \
--net-host \
--privileged \
--mount="type=bind,src=/etc/ovn/,dst=/etc/ovn,options=rbind:rw" \
--mount="type=bind,src=/var/run/ovn,dst=/var/run/ovn,options=rbind:rw" \
--mount="type=bind,src=/var/log/ovn,dst=/var/log/ovn,options=rbind:rw" \
--env="NODE_IPS=<所有节点的 IP 地址，逗号分隔>" \   ## 例如：--env="NODE_IPS="192.168.178.97,192.168.181.93,192.168.177.192""
--env="LOCAL_IP=<当前节点的 IP 地址>" \   ## 例如：--env="LOCAL_IP="192.168.178.97""
<镜像仓库地址> ovn-ic-db bash start-ic-db.sh   ## 例如：registry.alauda.cn:60080/acp/kube-ovn:v1.9.25 ovn-ic-db bash start-ic-db.sh
```

- 另外两个节点的命令：

```
ctr -n k8s.io run \
-d \
--env "ENABLE_OVN_LEADER_CHECK=false" \
--net-host \
--privileged \
--mount="type=bind,src=/etc/ovn/,dst=/etc/ovn,options=rbind:rw" \
--mount="type=bind,src=/var/run/ovn,dst=/var/run/ovn,options=rbind:rw" \
--mount="type=bind,src=/var/log/ovn,dst=/var/log/ovn,options=rbind:rw" \
--env="NODE_IPS=<所有节点的 IP 地址，逗号分隔>" \   ## 例如：--env="NODE_IPS="192.168.178.97,192.168.181.93,192.168.177.192"" \
--env="LOCAL_IP=<当前节点的 IP 地址>" \   ## 例如：--env="LOCAL_IP="192.168.181.93""
--env="LEADER_IP=<Leader 节点的 IP 地址>" \   ## 例如：--env="LEADER_IP="192.168.178.97""
<镜像仓库地址> ovn-ic-db bash start-ic-db.sh   ## 例如：registry.alauda.cn:60080/acp/kube-ovn:v1.9.25 ovn-ic-db bash start-ic-db.sh
```

在 global 集群中部署集群互联控制器

在 global 的任意控制节点上，根据注释替换以下参数，并执行以下命令创建 ConfigMap 资源。

注意：为确保正常运行，不允许修改 global 中名为 ovn-ic 的 ConfigMap。如需修改任何参数，请先删除该 ConfigMap，再重新正确配置后应用。

```
cat << EOF |kubectl apply -f -
apiVersion: v1
kind: ConfigMap
metadata:
  name: ovn-ic
  namespace: cpaas-system
data:
  ic-db-host: "192.168.39.22,192.168.39.24,192.168.39.37"   # 集群互联控制器所在节点的地址，此处为部署控制器的三个节点的本地 IP
  ic-nb-port: "6645"            # 集群互联控制器 nb 端口，默认 6645
  ic-sb-port: "6646"            # 集群互联控制器 sb 端口，默认 6646
EOF
```

加入集群互联

将网络模式为 Kube-OVN 的集群加入集群互联。

前提条件

集群中的已创建子网、**ovn-default** 和 **join** 子网 与集群互联组中的任何集群网段都不冲突。

操作步骤

1. 在左侧导航栏中，点击 **Clusters** > **Cluster of clusters**。
2. 点击要加入集群互联的 *集群* 名称。
3. 在右上角，点击 **Options** > **Cluster Interconnect**。
4. 点击 **Join the cluster interconnect**。
5. 为该集群选择一个网关节点。
6. 点击 **Join**。

相关操作

更新互联集群的网关节点信息

更新已加入集群互联组的集群网关节点信息。

操作步骤

1. 在左侧导航栏中，点击 **Clusters** > **Cluster of clusters**。
2. 点击需要更新网关节点信息的 *集群名称*。
3. 在右上角，点击 **Operations** > **Cluster Interconnect**。
4. 点击要更新网关节点信息的集群的 **Update Gateway Node**。
5. 重新为该集群选择网关节点。
6. 点击 **Update**。

退出集群互联

已加入集群互联组的集群退出集群互联，退出后会断开该集群 Pod 与外部集群 Pod 的连接。

操作步骤

1. 在左侧导航栏中，点击 **Clusters** > **Cluster of clusters**。
2. 点击要下线的 *集群* 名称。
3. 在右上角，点击 **Options** > **Cluster Interconnect**。
4. 点击要退出的集群的 **Exit cluster interconnection**。
5. 正确输入集群名称。
6. 点击 **Exit**。

清理互联集群残留

当集群未退出互联集群就被删除时，控制器上可能会残留部分数据。当再次使用这些节点创建集群并加入互联集群时，可能会出现失败。可以查看控制器（kube-ovn-controller）`/var/log/ovn/ovn-ic.log` 日志中的详细错误信息。部分错误信息可能包括：

```
transaction error: {"details":"Transaction causes multiple rows in xxxxxx"}
```

操作步骤

1. 退出互联集群。
2. 在容器或 Pod 中执行清理脚本。
   可以直接在 ovn-ic-db 容器或 ovn-ic-controller Pod 中执行清理脚本。请任选以下一种方式：
   方式 **1**：在 **ovn-ic-db** 容器中执行

   - 进入 ovn-ic-db 容器，并使用以下命令执行清理操作。

     ```
     ctr -n k8s.io task exec -t --exec-id ovn-ic-db ovn-ic-db /bin/bash
     ```

     然后执行以下任一清理命令：

     - 使用原集群名称执行清理操作。将 *<cluster-name>* 替换为原集群的名称：

       ```
       ./clean-ic-az-db.sh <cluster-name>
       ```

     - 使用原集群中任意节点名称执行清理操作。将 *<node-name>* 替换为原集群中任意节点的名称：

       ```
       ./clean-ic-az-db.sh <node-name>
       ```

   方式 **2**：在 **ovn-ic-controller Pod** 中执行

   - 进入 ovn-ic-controller Pod，并使用以下命令执行清理操作。

     ```
     kubectl -n kube-system exec -ti $(kubectl get pods -n kube-system -l app=ovn-ic-controller -o custom-columns=NAME:.metadata.name --no-headers) -- /bin/bash
     ```

     然后执行以下任一清理命令：

     - 使用原集群名称执行清理操作。将 *<cluster-name>* 替换为原集群的名称：

       ```
       ./clean-ic-az-db.sh <cluster-name>
       ```

     - 使用原集群中任意节点名称执行清理操作。将 *<node-name>* 替换为原集群中任意节点的名称：

```
./clean-ic-az-db.sh <node-name>
```

卸载互联集群

注意：步骤 1 到 步骤 3 需要在所有已加入互联集群的业务集群上执行。

操作步骤

1. 退出互联集群。具体有两种退出方式，请根据需要选择一种。

   - 删除业务集群中名为 ovn-ic-config 的 ConfigMap。使用以下命令。

     ```
     kubectl -n kube-system delete cm ovn-ic-config
     ```

   - 通过平台操作退出互联集群。

2. 使用以下命令进入 ovn-central 的 Leader Pod。

   ```
   kubectl -n kube-system exec -ti $(kubectl get pods -n kube-system -lovn
   -nb-leader=true -o custom-columns=NAME:.metadata.name --no-headers) --
   /bin/bash
   ```

3. 使用以下命令清理 ts 逻辑交换机。

   ```
   ovn-nbctl ls-del ts
   ```

4. 登录到部署控制器的节点并删除控制器。

   - Podman 命令：

     ```
     podman stop ovn-ic-db
     podman rm ovn-ic-db
     ```

   - Containerd 命令：

```
ctr -n k8s.io task kill ovn-ic-db
ctr -n k8s.io containers rm ovn-ic-db
```

5. 使用以下命令删除 global 集群中名为 ovn-ic 的 ConfigMap。

```
kubectl delete cm ovn-ic -n cpaas-system
```

配置集群网关高可用

在加入集群互联后，如需将集群网关配置为高可用，可以执行以下步骤：

1. 登录到需要转换为高可用网关的集群，并执行以下命令，将 `enable-ic` 字段修改为 `false`。
   注意：将 `enable-ic` 字段修改为 `false` 会中断集群互联，直到再次将其设置为 `true`。

   ```
   kubectl edit cm ovn-ic-config -n kube-system
   ```

2. 通过更新 `gw-nodes` 字段修改网关节点配置，并使用英文逗号分隔网关节点；同时将 `enable-ic` 字段改为 `true`。

```
kubectl edit cm ovn-ic-config -n kube-system

# 配置示例
apiVersion: v1
data:
  auto-route: "true"
  az-name: az1
  enable-ic: "true"
  gw-nodes: 192.168.188.234,192.168.189.54
  ic-db-host: 192.168.178.97
  ic-nb-port: "6645"
  ic-sb-port: "6646"
kind: ConfigMap
metadata:
  creationTimestamp: "2023-06-13T08:01:16Z"
  name: ovn-ic-config
  namespace: kube-system
  resourceVersion: "99671"
  uid: 6163790a-ad9d-4d07-ba82-195b11244983
```

3. 进入 ovn-central 集群中的 Pod，执行 `ovn-nbctl lrp-get-gateway-chassis {当前集群名称}-ts` 命令，验证配置是否生效。

```
ovn-nbctl lrp-get-gateway-chassis az1-ts

# 返回显示示例。其中 100 和 99 的值为优先级，值越大，对应网关节点被使用的优先级越高。
az1-ts-71292a21-131d-492a-9f0c-0611af458950 100
az1-ts-1de7ee15-f372-4ab9-8c85-e54d61ea18f1 99
```

配置 Egress Gateway

目录

- 关于 Egress Gateway
  - 实现细节
- 前提条件
- 使用方法
  - 创建 Network Attachment Definition
  - 创建 VPC Egress Gateway
  - 启用基于 BFD 的高可用
- 配置参数
  - VPC BFD Port
  - VPC Egress Gateway
- 注意事项
- 相关资源

关于 Egress Gateway

Egress Gateway 用于控制 Pod 的外部网络访问，使用一组静态地址，具有以下特性：

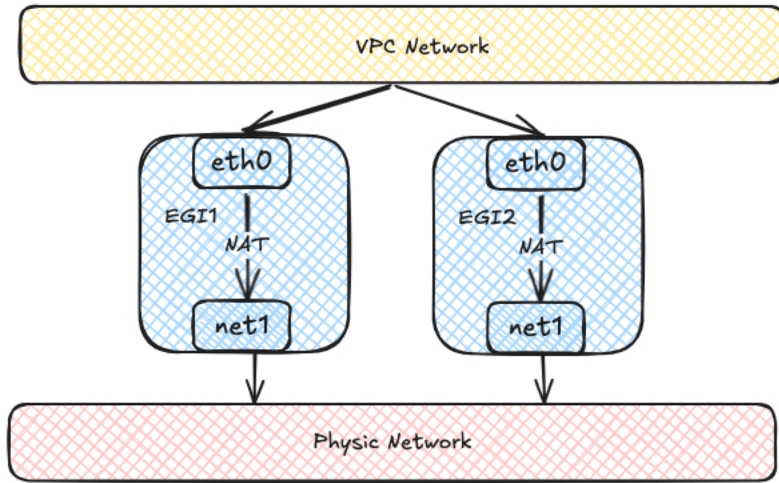
- 通过 ECMP 实现 Active-Active 高可用，支持水平吞吐量扩展
- 通过 BFD 实现快速故障切换（<1秒）
- 支持 IPv6 和双栈
- 通过 NamespaceSelector 和 PodSelector 实现细粒度路由控制
- 通过 NodeSelector 实现 Egress Gateway 的灵活调度

同时，Egress Gateway 具有以下限制：

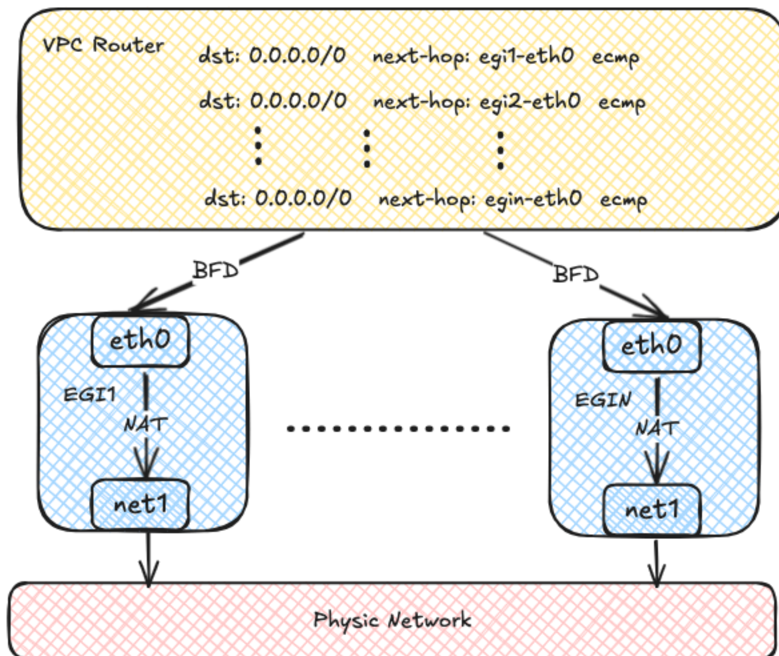
- 使用 macvlan 作为底层网络连接，要求底层网络支持 Underlay
- 多实例 Gateway 模式下需要多个 Egress IP
- 目前仅支持 SNAT，不支持 EIP 和 DNAT
- 目前不支持记录源地址转换关系

实现细节

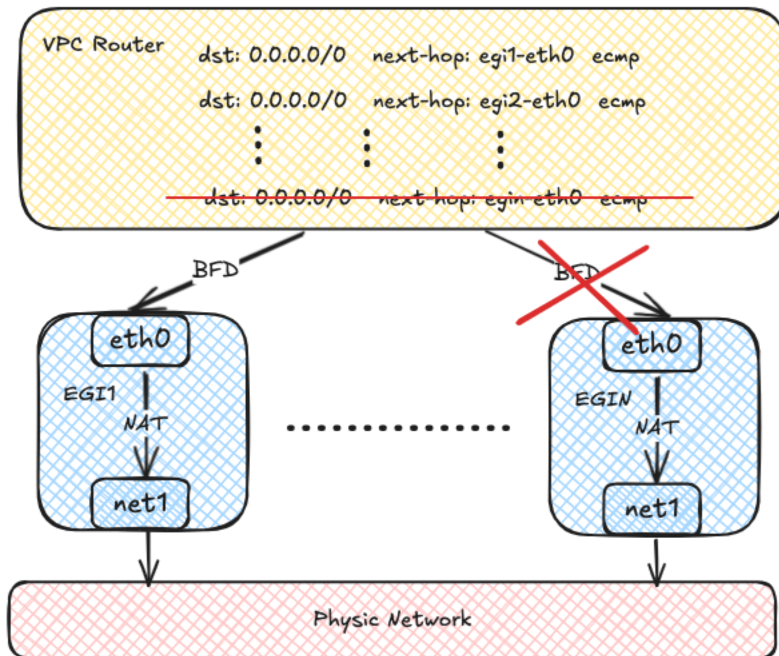
每个 Egress Gateway 由多个具有多个网络接口的 Pod 组成。 每个 Pod 有两个网络接口：一个加入虚拟网络，用于 VPC 内通信， 另一个通过 Macvlan 连接到底层物理网络，用于外部网络通信。 虚拟网络流量最终通过 Egress Gateway 实例内的 NAT 访问外部网络。



每个 Egress Gateway 实例在 OVN 路由表中注册其地址。 当 VPC 内的 Pod 需要访问外部网络时， OVN 使用源地址哈希将流量转发到多个 Egress Gateway 实例地址， 实现负载均衡。随着 Egress Gateway 实例数量增加， 吞吐量也能水平扩展。



OVN 使用 BFD 协议探测多个 Egress Gateway 实例。 当某个 Egress Gateway 实例故障时，OVN 将对应路由标记为不可用， 实现快速故障检测和恢复。



前提条件

集群中 必须 安装 *Alauda Container Platform Networking for Multus*，才能使用 Egress Gateway。

关于 *Alauda Container Platform Networking for Multus* 的安装，请参考 安装 Multus CNI。

使用方法

创建 **Network Attachment Definition**

Egress Gateway 使用多个 NIC 同时访问内网和外网， 因此需要创建一个 Network Attachment Definition 以连接外部网络。 下面是使用 macvlan 插件并由 Kube-OVN 提供 IPAM 的示例：

```yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: eth1
  namespace: default
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "macvlan",
    "master": "eth1", (1)
    "mode": "bridge",
    "ipam": {
      "type": "kube-ovn",
      "server_socket": "/run/openvswitch/kube-ovn-daemon.sock",
      "provider": "eth1.default" (2)
    }
  }'
---
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: macvlan1
spec:
  protocol: IPv4
  provider: eth1.default (3)
  cidrBlock: 172.17.0.0/16 (4)
  gateway: 172.17.0.1 (5)
  excludeIps: (6)
    - 172.17.0.2..172.17.0.10
```

1. 连接外部网络的宿主机接口。
2. provider 名称，格式为 `<network attachment definition name>.<namespace>`。
3. 用于标识外部网络的 provider 名称，必须与 NetworkAttachmentDefinition 中一致。
4. 外部网络的 CIDR。
5. 外部网络的网关。
6. 排除自动分配的 IP 范围。详情请参考 Kube-OVN Overlay 网络示例 Subnet CR。

> **TIP**
>
> 你可以使用任何 CNI 插件创建 Network Attachment Definition 来访问对应网络。 上述示例的替代方案是使用 Kube-OVN 的下层子网而非 macvlan。

创建 VPC Egress Gateway

创建 VPC Egress Gateway 资源，示例如下：

```yaml
apiVersion: kubeovn.io/v1
kind: VpcEgressGateway
metadata:
  name: gateway1
  namespace: default ①
spec:
  replicas: 1 ②
  externalSubnet: macvlan1 ③
  resources: ④
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 200m
      memory: 256Mi
      ephemeral-storage: 2Gi
  nodeSelector: ⑤
    - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
            - kube-ovn-worker
            - kube-ovn-worker2
  tolerations: ⑥
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
  selectors: ⑦
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: default
  policies: ⑧
    - snat: true ⑨
      subnets: ⑩
        - subnet1
    - snat: false
      ipBlocks: ⑪
        - 10.18.0.0/16
```

① 创建 VPC Egress Gateway 实例的命名空间。

② VPC Egress Gateway 实例的副本数。

③ 连接外部网络的外部子网。

④ 每个 VPC Egress Gateway 实例的资源请求和限制，未指定时使用控制器默认值。

⑤ 用于调度 VPC Egress Gateway 实例的节点选择器。

⑥ 用于调度 VPC Egress Gateway 实例的容忍度。

⑦ 用于选择通过 VPC Egress Gateway 访问外部网络的 Pod 的命名空间选择器和 Pod 选择器。

⑧ VPC Egress Gateway 的策略，包括 SNAT 及应用的子网/IP 块。

⑨ 是否启用该策略的 SNAT。

⑩ 策略应用的子网。

⑪ 策略应用的 IP 块。

上述资源创建了一个名为 *gateway1* 的 VPC Egress Gateway，位于 default 命名空间， 以下 Pod 将通过 *macvlan1* 子网访问外部网络：

- default 命名空间中的 Pod
- *subnet1* 子网下的 Pod
- IP 属于 CIDR *10.18.0.0/16* 的 Pod

NOTE

匹配 *.spec.selectors* 的 Pod 将启用 SNAT 访问外部网络。

创建完成后，查看 VPC Egress Gateway 资源：

```
$ kubectl get veg gateway1
NAME        VPC           REPLICAS    BFD ENABLED    EXTERNAL SUBNET    PHASE        READY    AGE
gateway1    ovn-cluster   1           false          macvlan1           Completed    true     13s
```

查看更多信息：

```
kubectl get veg gateway1 -o wide
NAME        VPC           REPLICAS    BFD ENABLED    EXTERNAL SUBNET    PHASE        READY    INTERNAL IPS     EXTERNAL IP
S      WORKING NODES         AGE
gateway1    ovn-cluster   1           false          macvlan1           Completed    true     ["10.16.0.12"]   ["172.17.0.
11"]   ["kube-ovn-worker"]   82s
```

查看工作负载：

```
$ kubectl get deployment -l ovn.kubernetes.io/vpc-egress-gateway=gateway1
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
gateway1    1/1     1            1           4m40s

$ kubectl get pod -l ovn.kubernetes.io/vpc-egress-gateway=gateway1 -o wide
NAME                        READY   STATUS     RESTARTS   AGE     IP           NODE              NOMINATED NODE   READ
INESS GATES
gateway1-b9f8b4448-76lhm    1/1     Running    0          4m48s   10.16.0.12   kube-ovn-worker   <none>           <non
e>
```

查看 Pod 中的 IP 地址、路由和 iptables 规则：

```
$ kubectl exec gateway1-b9f8b4448-76lhm -c gateway -- ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: net1@if13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 62:d8:71:90:7b:86 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.11/16 brd 172.17.255.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::60d8:71ff:fe90:7b86/64 scope link
       valid_lft forever preferred_lft forever
17: eth0@if18: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc noqueue state UP group default
    link/ether 36:7c:6b:c7:82:6b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.16.0.12/16 brd 10.16.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::347c:6bff:fec7:826b/64 scope link
       valid_lft forever preferred_lft forever

$ kubectl exec gateway1-b9f8b4448-76lhm -c gateway -- ip rule show
0:      from all lookup local
1001:   from all iif eth0 lookup default
1002:   from all iif net1 lookup 1000
1003:   from 10.16.0.12 iif lo lookup 1000
1004:   from 172.17.0.11 iif lo lookup default
32766:  from all lookup main
32767:  from all lookup default

$ kubectl exec gateway1-b9f8b4448-76lhm -c gateway -- ip route show
default via 172.17.0.1 dev net1
10.16.0.0/16 dev eth0 proto kernel scope link src 10.16.0.12
10.17.0.0/16 via 10.16.0.1 dev eth0
10.18.0.0/16 via 10.16.0.1 dev eth0
172.17.0.0/16 dev net1 proto kernel scope link src 172.17.0.11

$ kubectl exec gateway1-b9f8b4448-76lhm -c gateway -- ip route show table 1000
default via 10.16.0.1 dev eth0

$ kubectl exec gateway1-b9f8b4448-76lhm -c gateway -- iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N VEG-MASQUERADE
-A PREROUTING -i eth0 -j MARK --set-xmark 0x4000/0x4000
-A POSTROUTING -d 10.18.0.0/16 -j RETURN
-A POSTROUTING -s 10.18.0.0/16 -j RETURN
-A POSTROUTING -j VEG-MASQUERADE
-A VEG-MASQUERADE -j MARK --set-xmark 0x0/0xffffffff
-A VEG-MASQUERADE -j MASQUERADE --random-fully
```

在 Gateway Pod 中抓包验证网络流量：

```
$ kubectl exec -ti gateway1-b9f8b4448-76lhm -c gateway -- bash
nobody@gateway1-b9f8b4448-76lhm:/kube-ovn$ tcpdump -i any -nnve icmp and host 172.17.0.1
tcpdump: data link type LINUX_SLL2
tcpdump: listening on any, link-type LINUX_SLL2 (Linux cooked v2), snapshot length 262144 bytes
06:50:58.936528 eth0  In  ifindex 17 92:26:b8:9e:f2:1c ethertype IPv4 (0x0800), length 104: (tos 0x0, ttl 63, id 30481, offset 0, flags [DF], proto ICMP (1), length 84)
    10.17.0.9 > 172.17.0.1: ICMP echo request, id 37989, seq 0, length 64
06:50:58.936574 net1  Out ifindex 2 62:d8:71:90:7b:86 ethertype IPv4 (0x0800), length 104: (tos 0x0, ttl 62, id 30481, offset 0, flags [DF], proto ICMP (1), length 84)
    172.17.0.11 > 172.17.0.1: ICMP echo request, id 39449, seq 0, length 64
06:50:58.936613 net1  In  ifindex 2 02:42:39:79:7f:08 ethertype IPv4 (0x0800), length 104: (tos 0x0, ttl 64, id 26701, offset 0, flags [none], proto ICMP (1), length 84)
    172.17.0.1 > 172.17.0.11: ICMP echo reply, id 39449, seq 0, length 64
06:50:58.936621 eth0  Out ifindex 17 36:7c:6b:c7:82:6b ethertype IPv4 (0x0800), length 104: (tos 0x0, ttl 63, id 26701, offset 0, flags [none], proto ICMP (1), length 84)
    172.17.0.1 > 10.17.0.9: ICMP echo reply, id 37989, seq 0, length 64
```

OVN Logical Router 上自动创建路由策略：

```
$ kubectl ko nbctl lr-policy-list ovn-cluster
Routing Policies
     31000                            ip4.dst == 10.16.0.0/16   allow
     31000                            ip4.dst == 10.17.0.0/16   allow
     31000                           ip4.dst == 100.64.0.0/16   allow
     30000                              ip4.dst == 172.18.0.2  reroute  100.64.0.4
     30000                              ip4.dst == 172.18.0.3  reroute  100.64.0.3
     30000                              ip4.dst == 172.18.0.4  reroute  100.64.0.2
     29100                  ip4.src == $VEG.8ca38ae7da18.ipv4  reroute  10.16.0.12 (1)
     29100                   ip4.src == $VEG.8ca38ae7da18_ip4  reroute  10.16.0.12 (2)
     29000 ip4.src == $ovn.default.kube.ovn.control.plane_ip4  reroute  100.64.0.3
     29000       ip4.src == $ovn.default.kube.ovn.worker2_ip4  reroute  100.64.0.2
     29000        ip4.src == $ovn.default.kube.ovn.worker_ip4  reroute  100.64.0.4
     29000     ip4.src == $subnet1.kube.ovn.control.plane_ip4  reroute  100.64.0.3
     29000           ip4.src == $subnet1.kube.ovn.worker2_ip4  reroute  100.64.0.2
     29000            ip4.src == $subnet1.kube.ovn.worker_ip4  reroute  100.64.0.4
```

1. VPC Egress Gateway 用于转发 *.spec.policies* 指定 Pod 流量的 Logical Router 策略。
2. VPC Egress Gateway 用于转发 *.spec.selectors* 指定 Pod 流量的 Logical Router 策略。

若需启用负载均衡，修改 *.spec.replicas*，示例如下：

```
$ kubectl scale veg gateway1 --replicas=2
vpcegressgateway.kubeovn.io/gateway1 scaled

$ kubectl get veg gateway1
NAME       VPC           REPLICAS   BFD ENABLED   EXTERNAL SUBNET   PHASE       READY   AGE
gateway1   ovn-cluster   2          false         macvlan           Completed   true    39m

$ kubectl get pod -l ovn.kubernetes.io/vpc-egress-gateway=gateway1 -o wide
NAME                        READY   STATUS    RESTARTS   AGE   IP           NODE               NOMINATED NODE   READI
NESS GATES
gateway1-b9f8b4448-76lhm    1/1     Running   0          40m   10.16.0.12   kube-ovn-worker    <none>           <none
>
gateway1-b9f8b4448-zd4dl    1/1     Running   0          64s   10.16.0.13   kube-ovn-worker2   <none>           <none
>

$ kubectl ko nbctl lr-policy-list ovn-cluster
Routing Policies
     31000                            ip4.dst == 10.16.0.0/16    allow
     31000                            ip4.dst == 10.17.0.0/16    allow
     31000                           ip4.dst == 100.64.0.0/16    allow
     30000                              ip4.dst == 172.18.0.2  reroute  100.64.0.4
     30000                              ip4.dst == 172.18.0.3  reroute  100.64.0.3
     30000                              ip4.dst == 172.18.0.4  reroute  100.64.0.2
     29100                  ip4.src == $VEG.8ca38ae7da18.ipv4  reroute  10.16.0.12, 10.16.0.13
     29100                   ip4.src == $VEG.8ca38ae7da18_ip4  reroute  10.16.0.12, 10.16.0.13
     29000 ip4.src == $ovn.default.kube.ovn.control.plane_ip4  reroute  100.64.0.3
     29000       ip4.src == $ovn.default.kube.ovn.worker2_ip4  reroute  100.64.0.2
     29000        ip4.src == $ovn.default.kube.ovn.worker_ip4  reroute  100.64.0.4
     29000     ip4.src == $subnet1.kube.ovn.control.plane_ip4  reroute  100.64.0.3
     29000           ip4.src == $subnet1.kube.ovn.worker2_ip4  reroute  100.64.0.2
     29000            ip4.src == $subnet1.kube.ovn.worker_ip4  reroute  100.64.0.4
```

启用基于 **BFD** 的高可用

基于 BFD 的高可用依赖于 VPC BFD LRP 功能， 因此需要修改 VPC 资源以启用 BFD Port。 下面是为默认 VPC 启用 BFD Port 的示例：

```yaml
apiVersion: kubeovn.io/v1
kind: Vpc
metadata:
  name: ovn-cluster
spec:
  bfdPort:
    enabled: true # 1
    ip: 10.255.255.255 # 2
    nodeSelector: # 3
      matchLabels:
        kubernetes.io/os: linux
```

1. 是否启用 BFD Port。
2. BFD Port 的 IP 地址，必须是有效且不与其他 IP/子网冲突的地址。
3. 用于选择运行 BFD Port 的节点的节点选择器，BFD Port 绑定选中节点的 OVN HA Chassis Group，以 Active/Backup 模式工作。

启用 BFD Port 后，OVN Logical Router 上会自动创建专用于 BFD 的 LRP：

```
$ kubectl ko nbctl show ovn-cluster
router 0c1d1e8f-4c86-4d96-88b2-c4171c7ff824 (ovn-cluster)
    port bfd@ovn-cluster (1)
        mac: "8e:51:4b:16:3c:90"
        networks: ["10.255.255.255"]
    port ovn-cluster-join
        mac: "d2:21:17:71:77:70"
        networks: ["100.64.0.1/16"]
    port ovn-cluster-ovn-default
        mac: "d6:a3:f5:31:cd:89"
        networks: ["10.16.0.1/16"]
    port ovn-cluster-subnet1
        mac: "4a:09:aa:96:bb:f5"
        networks: ["10.17.0.1/16"]
```

1. OVN Logical Router 上创建的 BFD Port。

之后，在 VPC Egress Gateway 中将 *.spec.bfd.enabled* 设置为 *true*，示例如下：

```yaml
apiVersion: kubeovn.io/v1
kind: VpcEgressGateway
metadata:
  name: gateway2
  namespace: default
spec:
  vpc: ovn-cluster (1)
  replicas: 2
  internalSubnet: ovn-default (2)
  externalSubnet: macvlan1 (3)
  bfd:
    enabled: true (4)
    minRX: 100 (5)
    minTX: 100 (6)
    multiplier: 5 (7)
  policies:
    - snat: true
      ipBlocks:
        - 10.18.0.0/16
```

1. Egress Gateway 所属的 VPC。
2. Egress Gateway 实例连接的内部子网。
3. Egress Gateway 实例连接的外部子网。
4. 是否为 Egress Gateway 启用 BFD。
5. BFD 的最小接收间隔，单位毫秒。
6. BFD 的最小发送间隔，单位毫秒。
7. BFD 的乘数，决定丢包多少次后判定故障。

示例中创建了一个名为 *gateway2* 的 VPC Egress Gateway，副本数为 2，且启用了 BFD。 当某个实例故障时，BFD 会话断开，OVN 会快速检测故障并停止转发流量到故障实例， 所有流量将转发到健康实例，确保外部网络访问不中断。

故障切换时间取决于 BFD 配置， 计算公式为：*故障切换时间 = (multiplier + 1) * max(minRX, minTX)*。 本例中约为 500~600 毫秒。

> **NOTE**
>
> 故障切换期间，现有连接可能会中断，需要重新建立连接，但新连接不会受影响，可正常建立。

查看 VPC Egress Gateway 信息：

```
$ kubectl get veg gateway2 -o wide
NAME       VPC    REPLICAS   BFD ENABLED   EXTERNAL SUBNET   PHASE       READY   INTERNAL IPS                    EXT
ERNAL IPS                    WORKING NODES                           AGE
gateway2   vpc1   2          true          macvlan           Completed   true    ["10.16.0.102","10.16.0.103"]   ["1
72.17.0.13","172.17.0.14"]   ["kube-ovn-worker","kube-ovn-worker2"]   58s

$ kubectl get pod -l ovn.kubernetes.io/vpc-egress-gateway=gateway2 -o wide
NAME                          READY   STATUS    RESTARTS   AGE     IP            NODE               NOMINATED NODE   RE
ADINESS GATES
gateway2-fcc6b8b87-8lgvx   1/1     Running   0          2m18s   10.16.0.103   kube-ovn-worker2   <none>           <n
one>
gateway2-fcc6b8b87-wmww6   1/1     Running   0          2m18s   10.16.0.102   kube-ovn-worker    <none>           <n
one>

$ kubectl ko nbctl lr-policy-list ovn-cluster
Routing Policies
     31000                            ip4.dst == 10.16.0.0/16    allow
     31000                            ip4.dst == 10.17.0.0/16    allow
     31000                           ip4.dst == 100.64.0.0/16    allow
     30000                              ip4.dst == 172.18.0.2  reroute  100.64.0.4
     30000                              ip4.dst == 172.18.0.3  reroute  100.64.0.3
     30000                              ip4.dst == 172.18.0.4  reroute  100.64.0.2
     29100                  ip4.src == $VEG.8ca38ae7da18.ipv4  reroute  10.16.0.102, 10.16.0.103  bfd
     29100                   ip4.src == $VEG.8ca38ae7da18_ip4  reroute  10.16.0.102, 10.16.0.103  bfd
     29090                  ip4.src == $VEG.8ca38ae7da18.ipv4     drop
     29090                   ip4.src == $VEG.8ca38ae7da18_ip4     drop
     29000 ip4.src == $ovn.default.kube.ovn.control.plane_ip4  reroute  100.64.0.3
     29000       ip4.src == $ovn.default.kube.ovn.worker2_ip4  reroute  100.64.0.2
     29000        ip4.src == $ovn.default.kube.ovn.worker_ip4  reroute  100.64.0.4
     29000     ip4.src == $subnet1.kube.ovn.control.plane_ip4  reroute  100.64.0.3
     29000           ip4.src == $subnet1.kube.ovn.worker2_ip4  reroute  100.64.0.2
     29000            ip4.src == $subnet1.kube.ovn.worker_ip4  reroute  100.64.0.4

$ kubectl ko nbctl list bfd
_uuid               : 223ede10-9169-4c7d-9524-a546e24bfab5
detect_mult         : 5
dst_ip              : "10.16.0.102"
external_ids        : {af="4", vendor=kube-ovn, vpc-egress-gateway="default/gateway2"}
logical_port        : "bfd@ovn-cluster"
min_rx              : 100
min_tx              : 100
options             : {}
status              : up

_uuid               : b050c75e-2462-470b-b89c-7bd38889b758
detect_mult         : 5
dst_ip              : "10.16.0.103"
external_ids        : {af="4", vendor=kube-ovn, vpc-egress-gateway="default/gateway2"}
logical_port        : "bfd@ovn-cluster"
min_rx              : 100
min_tx              : 100
options             : {}
status              : up
```

查看 BFD 连接状态：

```
$ kubectl exec gateway2-fcc6b8b87-8lgvx -c bfdd -- bfdd-control status
There are 1 sessions:
Session 1
 id=1 local=10.16.0.103 (p) remote=10.255.255.255 state=Up

$ kubectl exec gateway2-fcc6b8b87-wmww6 -c bfdd -- bfdd-control status
There are 1 sessions:
Session 1
 id=1 local=10.16.0.102 (p) remote=10.255.255.255 state=Up
```

NOTE

如果所有网关实例均不可用，应用了 VPC Egress Gateway 的出口流量将被丢弃。

配置参数

VPC BFD Port

<table>
<tr><th colspan="2">字段</th><th>类型</th><th>可选</th><th>默认值</th><th colspan="2">描述</th><th>示例</th></tr>
<tr><td colspan="2">enabled</td><td>boolean</td><td>是</td><td>false</td><td colspan="2">是否启用 BFD Port。</td><td>true</td></tr>
<tr><td colspan="2" rowspan="3">ip</td><td rowspan="3">string</td><td rowspan="3">否</td><td rowspan="3">-</td><td colspan="2" rowspan="3">BFD Port 使用的 IP 地址。
必须不与其他地址冲突。支持 IPv4、IPv6 和双栈。</td><td>169.255.255.255</td></tr>
<tr><td>fdff::1</td></tr>
<tr><td>169.255.255.255,fdff::1</td></tr>
<tr><td rowspan="2">nodeSelector</td><td>matchLabels</td><td>object</td><td>是</td><td>-</td><td rowspan="2">用于选择承载 BFD Port 的节点的标签选择器。
BFD Port 绑定选中节点的 OVN HA Chassis Group，以 Active/Backup 模式工作。
若未指定，Kube-OVN 会自动选择最多三个节点。
可通过执行 kubectl ko nbctl list ha_chassis_group 查看所有 OVN HA Chassis Group 资源。</td><td>键值对映射。</td><td>-</td></tr>
<tr><td>matchExpressions</td><td>object 数组</td><td>是</td><td>-</td><td>标签选择器需求列表，需求间为 AND 关系。</td><td>-</td></tr>
</table>

VPC Egress Gateway

字段	类型	可选	默认值	描述
vpc	string	是	默认 VPC 名称 (ovn-cluster)	VPC 名称。
replicas	integer/int32	是	1	副本数。
prefix	string	是	-	工作负载部署名称的不可变前缀。

字段			类型	可选	默认值	描述		
image			string	是	-	工作负载部署使用的镜像。		
internalSubnet			string	是	默认 VPC 内子网名称。	用于访问内外网的子网名称。		
externalSubnet				否	-			
internalIPs			string 数组	是	-	用于访问内外网的 IP 地址。支持 IPv4、IPv6 和双 指定的 IP 数量不得少于 *replicas*。 建议设置为 *<replicas> + 1*，避免 Pod 创建异常的情况。		
externalIPs								
bfd	enabled		boolean	是	false	BFD 配置。	是否为 Egress Ga 启用 BFD。	
	minRX		integer/int32	是	1000		BFD 的 minRX/mi 单位毫秒。	
	minTX							
	multiplier		integer/int32	是	3		BFD 乘数。	
policies	snat		boolean	是	false	出口策略。	是否启用 SNAT/MASQUER	
	ipBlocks		string 数组	是	-		应用网关的 IP 段。 支持 IPv4 和 IPv6	
	subnets		string 数组	是	-		应用网关的 VPC 称。 支持 IPv4、IPv6 子网。	
selectors	namespaceSelector	matchLabels	object	是	-	通过命名空间选择器和 Pod 选择器配置出口策略。 匹配的 Pod 会应用 SNAT/MASQUERADE。	命名空间选择器，空标签选择器匹配所有命名空间。	
		matchExpressions	object 数组	是	-			A
	podSelector	matchLabels	object	是	-		Pod 选择器，空标签选择器	
		matchExpressions	object 数组	是	-			

字段		类型	可选	默认值	描述		
						匹配所有 Pod。	求 表 求 A 系
nodeSelector	matchLabels	object	是	-	用于选择承载工作负载部署的节点的节点选择器。 工作负载（Deployment/Pod）将在选中节点上运行。	键值对映射。	
	matchExpressions	object 数组	是	-		标签选择器需求列 需求间为 AND 关	
	matchFields	object 数组	是	-		字段选择器需求列 需求间为 AND 关	
tolerations	key	string	是	-	用于调度 VPC Egress Gateway 实例的容忍度。	key 是容忍的污点 表示匹配所有污点 若 key 为空，oper 必须为 Exists；此 示匹配所有值和所 键。	
	operator	string	是	Equal		operator 表示键与 关系。有效值为 E 和 Equal。 Exists 相当于值的 符，允许 Pod 容忍 所有污点。	
	value	string	是	-		value 是容忍的污 若 operator 为 Exi value 应为空，否 通字符串。	
	effect	string	是	-		effect 表示匹配的 果，空表示匹配所 果。 指定时允许值为 NoSchedule、 PreferNoSchedule NoExecute。	
	tolerationSeconds	integer	是	-		tolerationSeconds 容忍 NoExecute 效 点的时间（秒）。 默认不设置，表示 容忍（不驱逐）。	

字段	类型	可选	默认值	描述
				系统将 0 和负值视 （立即驱逐）。
trafficPolicy	string	是	Cluster	仅在启用 **BFD** 时生效。 可选值：*Cluster*/*Local*。 设置为 *Local* 时，出口流量优先重定向到同节点运 VPC Egress Gateway 实例，若实例不可用，则重 到其他实例。

注意事项

任何导致 Egress Gateway 实例被删除/重建的操作都可能触发出口流量的临时故障切换，包括但不限于：

1. 修改副本数；
2. 修改部分配置，如内部/外部 IP、节点选择器、BFD 配置等；
3. 升级/降级 Kube-OVN（当未指定 *.spec.image* 时）；
4. 手动删除 Egress Gateway 实例 Pod。

相关资源

- Egress Gateway - Kube-OVN 文档↗
- RFC 5880 - 双向转发检测 (BFD)↗

配置 **Kube-OVN** 网络以支持 **Pod** 多网卡（**Alpha**）

通过使用 Multus CNI，您可以为 Pod 添加多个不同网络的网络接口。使用 Kube-OVN 网络的 Subnet 和 IP CRD 进行高级 IP 管理，实现子网管理、IP 预留、随机分配、固定分配等功能。

目录

- 安装 Multus CNI
  - 部署 Multus CNI 插件
- 创建子网
- 创建多网卡 Pod
- 验证双网卡创建
- 其他功能
  - 固定 IP
  - 额外路由

安装 **Multus CNI**

部署 **Multus CNI** 插件

1. 进入 管理员。
2. 在左侧导航栏点击 **Marketplace** > **Cluster Plugins**。

3. 在搜索栏输入 “multus” 查找 Multus CNI 插件。
4. 在列表中找到 **"Alauda Container Platform Networking for Multus"** 插件。
5. 点击插件条目旁的三点（⋮），选择 安装。
6. 插件将部署到您的集群中，您可以在 状态 列监控安装进度。

> **NOTE**
>
> Multus CNI 插件作为其他 CNI 插件和 Kubernetes 之间的中间件，使 Pod 能够拥有多个网络接口。

创建子网

根据以下示例创建 attachnet 子网：`network-attachment-definition.yml`。

> **NOTE**
>
> config 中的 provider 格式为 `<NAME>.<NAMESPACE>.ovn`，其中 `<NAME>` 和 `<NAMESPACE>` 分别是该 NetworkAttachmentDefinition CR 的名称和命名空间。

```yaml
apiVersion: 'k8s.cni.cncf.io/v1'
kind: NetworkAttachmentDefinition
metadata:
  name: attachnet
  namespace: default
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "kube-ovn",
    "server_socket": "/run/openvswitch/kube-ovn-daemon.sock",
    "provider": "attachnet.default.ovn"
    }'
```

创建后，应用该资源：

```
kubectl apply -f network-attachment-definition.yml
```

使用以下示例创建第二个网络接口的 Kube-OVN 子网：`subnet.yml`。

NOTE

- `spec.provider` 必须与 NetworkAttachmentDefinition 中的 provider 保持一致。
- 如果需要使用 Underlay 子网，设置子网的 `spec.vlan` 为您想使用的 VLAN CR 名称。根据需要配置其他子网参数。

```yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: subnet1
spec:
  cidrBlock: 172.170.0.0/16
  provider: attachnet.default.ovn
```

创建后，应用该资源：

```
kubectl apply -f subnet.yml
```

创建多网卡 Pod

根据以下示例创建 Pod。

NOTE

- `metadata.annotations` 必须包含键值对 `k8s.v1.cni.cncf.io/networks=default/attachnet`，其中值的格式为 `<NAMESPACE>/<NAME>`，`<NAMESPACE>` 和 `<NAME>` 分别是 NetworkAttachmentDefinition CR 的命名空间和名称。
- 如果 Pod 需要三个网络接口，配置 `k8s.v1.cni.cncf.io/networks` 的值为 `default/attachnet,default/attachnet2`。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  annotations:
    k8s.v1.cni.cncf.io/networks: default/attachnet
spec:
  containers:
    - name: web
      image: nginx:latest
      ports:
        - containerPort: 80
```

Pod 创建成功后，使用命令 `kubectl exec pod1 -- ip a` 查看 Pod 的 IP 地址。

验证双网卡创建

使用以下命令验证双网卡是否创建成功：

```
kubectl exec pod1 -- ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen
1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
151: eth0@if152: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1400 qdisc
noqueue state UP
    link/ether a6:3c:d8:ae:83:06 brd ff:ff:ff:ff:ff:ff
    inet 10.3.0.8/16 brd 10.3.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::a43c:d8ff:feae:8306/64 scope link
       valid_lft forever preferred_lft forever
153: net1@if154: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1400 qdisc
noqueue state UP
    link/ether 0a:36:08:01:dc:df brd ff:ff:ff:ff:ff:ff
    inet 172.170.0.3/16 brd 172.170.255.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::836:8ff:fe01:dcdf/64 scope link
       valid_lft forever preferred_lft forever
```

其他功能

固定 IP

- 主网卡（第一个网卡）：如果需要固定主网卡的 IP，方法与单网卡固定 IP 相同。为 Pod 添加注解 `ovn.kubernetes.io/ip_address=<IP>`。
- 次网卡（第二个或其他网卡）：基本方法与主网卡类似，不同之处在于注解键中的 `ovn` 替换为对应的 NetworkAttachmentDefinition provider。例如：`attachnet.default.ovn.kubernetes.io/ip_address=172.170.0.101`。

额外路由

从 1.8.0 版本开始，Kube-OVN 支持为次网卡配置额外路由。使用该功能时，在 NetworkAttachmentDefinition 的 config 中添加 `routes` 字段，填写需要配置的路由。例如：

```yaml
apiVersion: 'k8s.cni.cncf.io/v1'
kind: NetworkAttachmentDefinition
metadata:
  name: attachnet
  namespace: default
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "kube-ovn",
    "server_socket": "/run/openvswitch/kube-ovn-daemon.sock",
    "provider": "attachnet.default.ovn",
    "routes": [{
      "dst": "19.10.0.0/16"
    }, {
      "dst": "19.20.0.0/16",
      "gw": "19.10.0.1"
    }]
  }'
```

配置集中式网关

集中式网关允许子网内的 Pods 使用固定 IP 访问外部网络。这对于网络审计、IP 白名单和防火墙规则管理等安全操作尤为重要，因为您需要识别和控制来自特定源 IP 的流量。在集中式网关模式下，所有来自 Pods 的出站流量都会通过指定的网关节点路由，实现集中式的网络策略执行和监控。

> **NOTE**
>
> 集中子网下的 Pods 无法通过 `hostport` 或带有 `externalTrafficPolicy: Local` 的 NodePort 类型 Service 访问。

如果您希望子网内的流量使用固定 IP 访问外部网络，以便进行审计和白名单等安全操作， 可以将子网中的网关类型设置为 centralized。 在集中式网关模式下，Pods 访问外部网络的报文首先会被路由到特定节点的 `ovn0` 网卡， 然后通过主机的路由规则出站。 当 `natOutgoing` 设置为 `true` 时，Pod 访问外部网络时将使用特定节点的 IP。

集中式网关示例如下，其中 `gatewayType` 字段为 `centralized` ， `gatewayNode` 为 Kubernetes 中特定机器的 NodeName。

```yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: centralized
spec:
  protocol: IPv4
  cidrBlock: 10.166.0.0/16
  default: false
  excludeIps:
  - 10.166.0.1
  gateway: 10.166.0.1
  gatewayType: centralized
  gatewayNode: "node1,node2"
  natOutgoing: true
```

- 如果集中式网关想指定机器的特定网卡用于出站网络， `gatewayNode` 格式可以改为 `kube-ovn-worker:172.18.0.2, kube-ovn-control-plane:172.18.0.3`。
- 自 Kube-OVN v1.12.0 起，subnet crd 定义中新增了 `enableEcmp` 字段，用于将 ECMP 开关迁移到子网级别。您可以根据不同子网设置是否启用 ECMP 模式。 `kube-ovn-controller` 部署中的 `enable-ecmp` 参数不再使用。升级到 v1.12.0 版本后，子网开关会自动继承原全局开关参数的值。

NOTE

在集中式网关 ECMP 模式下，kube-ovn-controller 通过 ping 主动探测节点状态，5 秒内检测故障，5-10 秒内完成切换，期间可能会有部分流量失败。

在集中式网关主备模式下，切换基于节点 Ready 状态，断电情况下可能需要几分钟完成切换。

目录

使用标签选择器指定网关节点

使用标签选择器指定网关节点

除了直接指定节点名称外，您还可以使用 `gatewayNodeSelectors` 通过标签选择器动态选择网关节点。 这种方式更灵活，尤其适用于节点名称不固定或需要基于标签动态选择网关的场景。

NOTE

- 如果 `gatewayNode` 不为空，则优先使用 `gatewayNode`，忽略 `gatewayNodeSelectors`。
- 多个选择器之间采用 OR 逻辑——匹配任意选择器的节点都会成为网关节点。
- 当节点标签发生变化时，系统会自动更新网关节点列表。

```yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: centralized-selector
spec:
  protocol: IPv4
  cidrBlock: 10.166.0.0/16
  default: false
  excludeIps:
  - 10.166.0.1
  gateway: 10.166.0.1
  gatewayType: centralized
  gatewayNodeSelectors:
    - matchLabels:
        role: gateway
    - matchExpressions:
        - key: node-type
          operator: In
          values: ["gateway", "egress"]
  natOutgoing: true
```

配置 IPPool

IPPool 是比 Subnet 更细粒度的 IPAM 管理单元。
你可以通过 IPPool 将子网段细分为多个单元，每个单元绑定一个或多个命名空间。

目录

说明
创建 IPPool
使用 IPPool
注意事项

说明

创建 IPPool

示例如下：

```yaml
apiVersion: kubeovn.io/v1
kind: IPPool
metadata:
  name: pool-1
spec:
  subnet: ovn-default 1
  ips: 2
  - "10.16.0.201"
  - "10.16.0.210/30"
  - "10.16.0.220..10.16.0.230"
  namespaces: 3
  - ns-1
```

1 IP 池所属的子网。

2 IP 范围。支持的格式：*<IP>*、*<CIDR>* 和 *<IP1>..<IP2>*。支持 IPv4 和 IPv6。

3 可选，IP 池绑定的命名空间。绑定命名空间中的 Pod 只会从绑定的池中获取 IP，而不会从子网中的其他范围获取。

使用 IPPool

要从 IP 池中随机分配 IP，只需将 IP 池绑定到目标命名空间。
当绑定命名空间中的 Pod 创建时，其 IP 将从对应的 IP 池中分配。

> **NOTE**
>
> 在将 IP 池绑定到命名空间之前，请确保该命名空间只绑定了 IP 池所属的子网。

你也可以通过注解为 Pod 指定 IP 池：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
  annotations:
    ovn.kubernetes.io/ip_pool: pool-1
spec:
  containers:
  - name: web
    image: nginx:latest
```

对于工作负载，可在 Deployment、StatefulSet 等的 Pod 模板中使用注解：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
      annotations:
        ovn.kubernetes.io/ip_pool: pool-1
    spec:
      containers:
      - name: web
        image: nginx:latest
```

注意事项

1. 为确保与 *Fixed Addresses* 功能兼容，IP 池名称不能是 IP 地址。
2. 允许 IP 地址超出子网范围，但这些 IP 不会生效。
3. 同一子网下的不同 IP 池不能有重叠的 IP 范围。

4. `.spec.ips` 字段可随时更新，修改后立即生效。
5. IP 池会继承子网的保留 IP。随机分配 IP 池中的 IP 时，会跳过 IP 池范围内的保留 IP。
6. 从子网随机分配 IP 时，会排除子网中所有 IP 池的 IP 范围。
7. 可以将多个 IP 池绑定到同一个命名空间。

配置 MTU

在 Kube-OVN 中，MTU（最大传输单元）设置对于确保网络性能的最佳状态和避免数据包分片至关重要。MTU 定义了网络中可传输的最大数据包大小。

默认情况下，Kube-OVN 会检测底层物理网络接口的 MTU，并相应地为虚拟网络接口设置 MTU。 但是，在某些场景下，您可能需要自定义 Kube-OVN 网络组件的 MTU 设置。

目录

默认 MTU 行为

自定义 MTU 设置

全局 MTU 配置

按子网配置 MTU

默认 MTU 行为

Kube-OVN 会自动检测主机物理网络接口的 MTU， 并相应地为 Pod 和 OVS 接口设置 MTU。

在覆盖网络中，Kube-OVN 会降低 MTU 以适应 VXLAN 或 Geneve 封装的开销。 MTU 的计算方式如下：

封装类型	隧道 **IP** 版本	**MTU** 计算方式
Geneve	IPv4	物理网络接口 MTU - 100

封装类型	隧道 IP 版本	MTU 计算方式
VXLAN	IPv6	物理网络接口 MTU - 120
	IPv4	物理网络接口 MTU - 50
	IPv6	物理网络接口 MTU - 70

在底层网络中，MTU 设置为与物理网络接口 MTU 相匹配。

自定义 MTU 设置

MTU 设置可以针对覆盖网络进行全局自定义，也可以针对单个子网进行配置。

WARNING

错误调整 MTU 设置可能导致网络性能问题， 包括数据包丢失和分片。请确保 MTU 设置与您的底层网络基础设施兼容后再进行更改。

重要注意事项：增加 **MTU** 时

当将 **MTU** 从较小值增加到较大值时，必须重启所有 **Pod**，以确保新 MTU 在整个集群中统一生效。

为什么需要这样做？

OVS 内部端口（如 `ovn0`）会自动采用连接到 `br-int` 桥的所有接口中 最小的 **MTU** 作为自身 MTU。此行为在以下场景中可能导致问题：

1. 您为新 Pod 配置了更大的 MTU 值
2. 部分现有 Pod 仍使用原来的较小 MTU
3. 由于最小 MTU 规则，`ovn0` 接口保持较小的 MTU
4. 来自较大 MTU Pod 的流量在数据包超过 `ovn0` MTU 时被丢弃

解决方案： 增加 MTU 设置后，重新创建所有 Pod，确保整个网络路径中的 MTU 配置一致。

NOTE

MTU 配置更改仅对新创建的 Pod 生效。 现有 Pod 会保留原有的 MTU 设置，直到它们被重新创建。 建议在更改 MTU 设置时规划 Pod 重启。

全局 MTU 配置

要为覆盖网络设置全局 MTU，可以修改 `kube-ovn-cni` DaemonSet 的命令参数。 例如，要将全局 MTU 设置为 1400，可以按如下方式更新 DaemonSet：

```
kubectl -n kube-system edit ds kube-ovn-cni
```

然后，在容器参数中添加或更新 `--mtu=1400` 参数：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
    kubernetes.io/description: |
      This daemon set launches the kube-ovn cni daemon.
  name: kube-ovn-cni
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app: kube-ovn-cni
  template:
    metadata:
      labels:
        app: kube-ovn-cni
        component: network
        type: infra
    spec:
      containers:
      - name: cni-server
        args:
        - --mtu=1400
```

> **NOTE**
>
> 全局 MTU 设置仅影响覆盖网络。对于底层网络， MTU 默认保持与物理网络接口 MTU 一致。

按子网配置 MTU

您也可以通过在子网配置中指定 `mtu` 字段，为单个子网设置 MTU。 例如，要将某个子网的 MTU 设置为 1450，可以使用以下命令：

```
kubectl patch subnet my-subnet --type merge -p '{"spec": {"mtu": 1450}}'
```

此设置将覆盖该子网的全局 MTU 配置。

目录

weight: 13
操作步骤
启用 u2oInterconnection 的 Underlay 子网间隔离
第 1 步：配置 kube-ovn-controller
第 2 步：配置子网隔离

weight: 13

Underlay 和 Overlay 子网的自动互联

如果集群同时存在 Underlay 和 Overlay 子网，默认情况下，Overlay 子网下的 Pod 可以通过网关使用 NAT 访问 Underlay 子网中 Pod 的 IP。但 Underlay 子网中的 Pod 需要配置节点路由才能访问 Overlay 子网中的 Pod。

为了实现 Underlay 和 Overlay 子网的自动互联，可以手动修改 Underlay 子网的 YAML 文件。配置完成后，Kube-OVN 会使用额外的 Underlay IP 连接 Underlay 子网和 ovn-cluster 逻辑路由器，并设置相应的路由规则以实现互联。

操作步骤

1. 进入 管理员。

2. 在左侧导航栏点击 集群管理 > 资源管理。
3. 输入 **Subnet** 进行资源对象筛选。
4. 点击待修改的 Underlay 子网旁的 ⋮ > 更新。
5. 修改 YAML 文件，在 `Spec` 中添加字段 `u2oInterconnection: true`。
6. 点击 更新。

注意：Underlay 子网中已有的计算组件需要重新创建，变更才能生效。

启用 u2oInterconnection 的 Underlay 子网间隔离

当多个 Underlay 子网启用 `u2oInterconnection: true` 后，它们之间的流量不再经过物理网关，而是通过内部 OVN 网络直接路由。

如果需要在启用 `u2oInterconnection` 的两个 Underlay 子网之间实现隔离，必须先配置 kube-ovn-controller 参数，然后配置子网隔离。

第 1 步：配置 kube-ovn-controller

修改 kube-ovn-controller Deployment，禁用目标逻辑端口 IP 的连接跟踪跳过：

```
kubectl edit deployment kube-ovn-controller -n kube-system
```

添加或修改以下参数：

```
spec:
  template:
    spec:
      containers:
      - name: kube-ovn-controller
        args:
        - --ls-ct-skip-dst-lport-ips=false
```

CAUTION

`--ls-ct-skip-dst-lport-ips` 控制是否跳过目标逻辑端口 IP 的连接跟踪（conntrack）。默认值为 `true`，跳过 conntrack 以提升性能。设置为 `false` 不影响功能，但可能略微影响性能。

然而，对于基于 ACL 隔离的 Underlay 子网，必须设置为 `false`。否则，网关到 Pod 的流量会失败（例如 ping 请求能到达 Pod，但回复被丢弃），因为 ACL 隔离使用了 `allow-related`，需要 conntrack 状态；没有 conntrack，回复无法被识别为“相关”，因此被丢弃。

第 2 步：配置子网隔离

对子网配置以下参数：

```yaml
spec:
  u2oInterconnection: true
  acls:
  - action: drop
    direction: to-lport   # 入方向（流量进入逻辑端口）
    match: ip4.src == 172.20.0.0/16
    priority: 1002
  - action: drop
    direction: to-lport   # 入方向
    match: ip4.src == 192.50.0.0/16
    priority: 1002
```

ACL 参数说明：

参数	说明
`action`	动作：`allow`、`drop` 或 `allow-related`
`direction`	流量方向：`to-lport`（入方向）或 `from-lport`（出方向）
`match`	OVN 匹配表达式，使用 L2-L4 字段和布尔运算符
`priority`	规则优先级（值越大优先级越高，推荐范围：1002-1899）

NOTE

- `acls` 字段支持基于优先级的规则评估，比标准 Kubernetes NetworkPolicy 更灵活。
- 使用 `to-lport` 方向时，`ip4.src` 表示入方向流量的源 IP。
- 推荐优先级范围：`1002` 到 `1899`，避免与系统默认 ACL 规则冲突。

配置 Endpoint Health Checker

目录

- Overview
- Key Features
- Installation
  - 通过 Marketplace 安装
- How It Works
  - 健康检查机制
    - 核心功能
    - 健康检查流程
    - 性能提升
- How To Activate
  - Pod 级注解（推荐）
    - 针对 ALB
    - 针对 IngressNginx
    - 针对 EnvoyGateway
    - 针对自定义 Deployment
  - Pod 级 readinessGates（旧版）
- Uninstallation

Overview

Endpoint Health Checker 是一个集群插件，旨在监控和管理 k8s 集群中服务端点的健康状态。它会自动将不健康的端点从服务中移除，确保流量仅路由到健康实例，从而提升整体服务的可靠性和可用性。

Key Features

- 自动健康监控：持续监控 k8s 集群中服务端点的健康状态
- 负载均衡集成：自动将不健康端点从服务中移除
- 服务可用性：确保流量仅导向健康且可用的端点
- 快速故障切换：在节点断电时将端点切换时间从 40 秒缩短至 10 秒

Installation

通过 **Marketplace** 安装

1. 进入 管理员 > **Marketplace** > 集群插件。
2. 在插件列表中搜索“**Alauda Container Platform Endpoint Health Checker**”。
3. 点击 安装 打开安装配置页面。
4. 在部署配置对话框中，可选配置以下参数：

参数	描述
节点选择器	配置标签选择器，指定 Endpoint Health Checker 组件应运行在哪些节点上。点击 添加 可添加多个标签键值对。
节点容忍度	配置容忍度，允许 Endpoint Health Checker 组件调度到带有特定污点的节点上。点击 添加 可添加多个包含 Key、Value 和类型的容忍度。

5. 点击 安装 部署插件。
6. 等待插件状态变为“**Ready**”。

How It Works

健康检查机制

Endpoint Health Checker 是专门的健康监控组件，确保只有健康的端点接收流量。它通过监控服务端点并自动管理其可用性状态来实现。

核心功能

Endpoint Health Checker 的工作流程：

1. 服务发现：识别集群中配置了健康监控的服务和 Pod。
2. **Pod** 健康监控：监控支撑服务端点的 Pod 的就绪和存活探针状态。
3. 主动健康检查：使用可配置的标准执行主动健康评估：
    - **TCP** 连接检查：建立 TCP 连接以验证端口可访问性。
4. 端点管理：自动将不健康端点从服务端点列表中移除，防止流量路由到故障实例。

健康检查流程

健康检查流程包括：

- 探针集成：利用 Kubernetes 的就绪和存活探针结果作为初步健康指标。
- 网络连通性：向目标端点端口发送 TCP 包以验证可访问性。
- 响应验证：评估响应状态、时长和内容以确定端点健康状况。
- 自动故障切换：将无响应或失败的端点从服务端点列表中移除。

性能提升

- 之前方法：依赖 kubelet 心跳检测，延迟最长达 40 秒。
- 当前方法：主动端点健康检查，检测和切换时间为 10 秒。
- 提升效果：显著提升 ALB + MetalLB 环境中节点故障时的服务可用性。

How To Activate

健康检查可通过以下两种方式激活：

Pod 级注解（推荐）

针对 ALB

设置 `ALB2` 的 `alb.cpaas.io/pod-annotations` 注解

```
apiVersion: crd.alauda.io/v2
kind: ALB2
metadata:
  annotations:
    alb.cpaas.io/pod-annotations: '{"endpoint-health-checker.io/enable
d":"true"}'
  name: demo-alb
spec:
  config:
    loadbalancerName: demo-alb
    nodeSelector:
      ingress: 'true'
    replicas: 1
  type: nginx
```

针对 IngressNginx

1. 安装 ingress-nginx
2. 在 `IngressNginx` 的 `.spec.controller.podAnnotations` 中设置 `podAnnotations`。

```
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
  namespace: ingress-nginx-operator
spec:
  controller:
    replicaCount: 1
    podAnnotations:
      endpoint-health-checker.io/enabled: 'true'
```

针对 **EnvoyGateway**

1. 安装 envoy-gateway-operator
2. 在 `EnvoyProxy` 的 `.spec.provider.kubernetes.envoyDeployment.patch.value.spec.template.metadata.annotations` 中设置 `annotations`。

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo
spec:
  infrastructure:
    parametersRef:
      group: gateway.envoyproxy.io
      kind: EnvoyProxy
      name: demo
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: http
      port: 80
      protocol: HTTP
---
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
spec:
  provider:
    kubernetes:
      envoyDeployment:
        replicas: 1
        patch:
          type: StrategicMerge
          value:
            spec:
              template:
                metadata:
                  annotations:
                    endpoint-health-checker.io/enabled: 'true'
        container:
          imageRepository: registry.alauda.cn:60080/acp/envoyproxy/envo
y
    type: Kubernetes
```

针对自定义 Deployment

在 `Deployment` 的 `.spec.template.metadata.annotations` 中设置 `annotations`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
      annotations:
        endpoint-health-checker.io/enabled: 'true'
    spec:
      containers:
        - name: container
          ports:
            - containerPort: 8080
          livenessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 15
            periodSeconds: 10
          readinessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5
```

Pod 级 readinessGates（旧版）

为旧版本配置 Pod 的 readinessGates：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-legacy
  namespace: cpaas-system
spec:
  replicas: 3
  selector:
    matchLabels:
      app: pod-legacy
  template:
    metadata:
      labels:
        app: pod-legacy
    spec:
      readinessGates:
        - conditionType: 'endpointHealthCheckSuccess'
      containers:
        - name: container
          image: your-image:latest
          ports:
            - containerPort: 8080
          livenessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 15
            periodSeconds: 10
          readinessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5
```

> 注意：readinessGates 配置来自旧版本，建议新部署使用 Pod 注解 `endpoint-health-checker.io/enabled: 'true'`。

Uninstallation

卸载 Endpoint Health Checker：

1. 进入 管理员 > **Marketplace** > 集群插件。
2. 找到已安装的“**Endpoint Health Checker**”插件。
3. 点击选项菜单，选择 卸载。
4. 按提示确认卸载。

Tasks for ALB

目录

如何为 alb-operator 设置 NodeSelector 和 Tolerations

如何为 alb 设置 NodeSelector 和 Tolerations

如何为 alb-operator 设置 NodeSelector 和 Tolerations

更新 `deployment` 资源

```
# nodeSelector 和 tolerations 示例
kubectl patch subscription ingress-nginx-operator -n ingress-nginx-operat
or    --type='merge' -p '
{
  "spec": {
    "config": {
      "nodeSelector": {
        "node-role.kubernetes.io/infra": ""
      },
      "tolerations": [
        {
          "effect": "NoSchedule",
          "key": "node-role.kubernetes.io/infra",
          "operator": "Equal",
          "value": "reserved"
        }
      ]
    }
  }
}'
```

如何为 alb 设置 NodeSelector 和 Tolerations

更新 `alb` 资源

```
kubectl patch alb2 $NAME -n $MS --type='merge' -p '
{
  "metadata": {
    "annotations": {
      "alb.cpaas.io/toleration": "[{\"key\":\"node-role.kubernetes.io/inf
ra\",\"operator\":\"Equal\",\"value\":\"reserved\",\"effect\":\"NoSchedul
e\"}]"
    }
  },
  "spec": {
    "config": {
      "nodeSelector": {
        "node-role.kubernetes.io/infra": ""
      }
    }
  }
}'
```

任务：从 OCP Route 迁移到 GatewayAPI Route

目录

介绍

前提条件

基础 HTTP Route

- OCP Route 配置
- Gateway API 配置

路由超时

- OCP Route 配置
- Gateway API 配置

HTTP 严格传输安全 (HSTS)

- OCP Route 配置
- Gateway API 配置

基于 Cookie 的会话亲和

- OCP Route 配置
- Gateway API 配置

基于路径的路由

- OCP Route 配置
- Gateway API 配置

头部修改

- OCP Route 配置
- Gateway API 配置

连接限制
OCP Route 配置
Gateway API 配置
速率限制
OCP Route 配置
Gateway API 配置
IP 允许列表/阻止列表
OCP Route 配置
Gateway API 配置
URL 重写
OCP Route 配置
Gateway API 配置
跨命名空间路由准入
OCP Route 配置
Gateway API 配置
默认 TLS 证书用于 Ingress
OCP Route 配置
Gateway API 配置
使用自定义 CA 的 TLS 重新加密
OCP Route 配置
Gateway API 配置
使用自定义证书的边缘终止
OCP Route 配置
Gateway API 配置
TLS 透传
OCP Route 配置
Gateway API 配置
功能对比总结
迁移策略
相关文档

介绍

本指南提供了从 OpenShift Container Platform (OCP) Routes 迁移到 Kubernetes Gateway API HTTPRoutes（配合 Envoy Gateway）的详细步骤。每个章节涵盖了特定的 OCP Route 功能及其在 Gateway API 中的等效配置。

前提条件

1. 配置 EnvoyGatewayCtl
2. 配置 Gateway
3. 基本了解 Gateway API Routes

基础 HTTP Route

OCP Route 配置

在 OCP 中，基础 HTTP 路由通过 Route 资源创建：

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  namespace: demo
spec:
  host: example.com 1
  to: 2
    kind: Service
    name: example-service
  port: 3
    targetPort: 8080
```

1. 路由的主机名
2. 后端服务引用
3. 后端服务的目标端口

Gateway API 配置

在 Gateway API 中，等效配置使用 HTTPRoute：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames: (1)
    - example.com
  parentRefs: (2)
    - name: demo-gateway
      namespace: demo
  rules:
    - backendRefs: (3)
        - name: example-service
          port: 8080
```

1. 此路由接受的主机名（等同于 OCP Route 的 `host`）
2. Gateway 监听器引用
3. 后端服务及端口

路由超时

OCP Route 配置

在 OCP 中，超时通过注解配置：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/timeout: 30s # 1
    haproxy.router.openshift.io/timeout-tunnel: 1h # 2
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

1. HTTP 请求的一般超时
2. 隧道连接（WebSocket、HTTP/2 等）的超时

Gateway API 配置

在 Gateway API 中，超时配置在 HTTPRoute 规则中：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - backendRefs:
        - name: example-service
          port: 8080
      timeouts: # 1
        request: 30s
        backendRequest: 25s
```

1. 请求超时配置

更多详情请参见 Request Timeouts。

> **NOTE**
>
> Gateway API 没有专门针对隧道连接的超时，`request` 超时适用于所有连接类型。

HTTP 严格传输安全 (HSTS)

OCP Route 配置

在 OCP 中，HSTS 通过边缘终止或重新加密路由的注解配置：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/hsts_header: max-age=31536000;includeSubDomains;preload # 1
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: edge
```

1. HSTS 头配置

Gateway API 配置

在 Gateway API 中，HSTS 通过响应头修改配置：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - filters: # (1)
        - type: ResponseHeaderModifier
          responseHeaderModifier:
            add:
              - name: Strict-Transport-Security
                value: max-age=31536000;includeSubDomains;preload
      backendRefs:
        - name: example-service
          port: 8080
```

1. 向响应添加 HSTS 头

更多详情请参见 HTTP Header Modification。

> **NOTE**
>
> 与 OCP 的 `requiredHSTSPolicies` 集群范围强制不同，Gateway API 需要在每个路由上显式配置，且没有全局 HSTS 策略强制机制。

基于 Cookie 的会话亲和

OCP Route 配置

在 OCP 中，会话亲和通过注解配置：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/balance: source ①
    haproxy.router.openshift.io/disable_cookies: "false" ②
    router.openshift.io/cookie_name: my-cookie ③
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

① 负载均衡算法

② 启用基于 Cookie 的会话亲和

③ 会话持久化的 Cookie 名称

Gateway API 配置

在 Gateway API 中，会话持久化配置在 HTTPRoute 规则中：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - backendRefs:
        - name: example-service
          port: 8080
      sessionPersistence: # 1
        type: Cookie
        sessionName: my-cookie
        cookieConfig:
          lifetimeType: Permanent # 2
```

1. 会话持久化配置
2. Cookie 生命周期类型：`Permanent`（跨浏览器会话持久）或 `Session`（浏览器关闭时过期）

更多详情请参见 Session Affinity/Sticky Sessions。

基于路径的路由

OCP Route 配置

在 OCP 中，基于路径的路由使用 `path` 字段：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  path: /api 1
  to:
    kind: Service
    name: example-service
```

1 匹配请求的路径前缀

Gateway API 配置

在 Gateway API 中，路径匹配配置在 HTTPRoute 规则中：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - matches: 1
        - path:
            type: PathPrefix
            value: /api
      backendRefs:
        - name: example-service
          port: 8080
```

1 路径匹配配置

Gateway API 支持多种匹配类型：

- `PathPrefix`：匹配路径前缀（等同于 OCP 的默认行为）
- `Exact`：精确匹配路径
- `RegularExpression`：使用正则表达式匹配

更多详情请参见 HTTPRoute Matches。

头部修改

OCP Route 配置

在 OCP 中，头部修改通过注解实现：

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/response-set-header: X-Custom-Header:value 1
    haproxy.router.openshift.io/request-set-header: X-Request-Header:value 2
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

1 设置响应头

2 设置请求头

Gateway API 配置

在 Gateway API 中，头部修改通过过滤器配置：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - filters:
        - type: RequestHeaderModifier # 1
          requestHeaderModifier:
            add:
              - name: X-Request-Header
                value: value
        - type: ResponseHeaderModifier # 2
          responseHeaderModifier:
            add:
              - name: X-Custom-Header
                value: value
      backendRefs:
        - name: example-service
          port: 8080
```

1. 请求头修改
2. 响应头修改

更多详情请参见 HTTP Header Modification。

连接限制

OCP Route 配置

在 OCP 中，连接限制通过注解配置：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/pod-concurrent-connections: "100" # 1
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

1 每个后端 Pod 的最大并发连接数

Gateway API 配置

在 Gateway API 中，连接限制通过附加到 Gateway 的 ClientTrafficPolicy 配置：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: ClientTrafficPolicy
metadata:
  name: connection-limit-policy
  namespace: demo
spec:
  targetRefs: # 1
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo-gateway
  connection: # 2
    connectionLimit:
      value: 100
```

1 将策略附加到 Gateway

2 连接限制配置

更多详情请参见 Connection Limit。

> **NOTE**

与 OCP 针对每个后端 Pod 的限制不同，Gateway API 的连接限制应用于 Gateway 级别，并在 Envoy 代理实例间分布。

速率限制

OCP Route 配置

在 OCP 中，速率限制通过注解配置：

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/rate-limit-connections: "true" 1
    haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp: "1
0"
    haproxy.router.openshift.io/rate-limit-connections.rate-http: "100"
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

1 速率限制配置

Gateway API 配置

在 Gateway API 中，速率限制通过 BackendTrafficPolicy 配置：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: BackendTrafficPolicy
metadata:
  name: rate-limit-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: example-route
  rateLimit: # 1
    type: Local
    local:
      rules:
        - limit:
            requests: 100
            unit: Second
```

1. 速率限制配置

更多详情请参见 Rate Limiting。

> **NOTE**
>
> Gateway API 的速率限制比 OCP Route 注解更灵活，支持本地和全局速率限制机制。

IP 允许列表/阻止列表

OCP Route 配置

在 OCP 中，IP 允许列表通过注解配置：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/ip_allowlist: 192.168.1.0/24 10.0.0.1 ①
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

① IP 允许列表配置

Gateway API 配置

在 Gateway API 中，IP 过滤通过 SecurityPolicy 的授权规则实现：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: ip-filter-policy
  namespace: demo
spec:
  targetRefs: ①
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: example-route
  authorization: ②
    defaultAction: Deny
    rules:
      - action: Allow
        principal:
          clientCIDRs: ③
            - 192.168.1.0/24
            - 10.0.0.1/32
```

① 将策略附加到 HTTPRoute

② 授权配置，默认拒绝

3. 使用 CIDR 表示的 IP 允许列表

对于 IP 阻止列表（拒绝列表），设置 `defaultAction: Allow` 并使用 `action: Deny`：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: ip-blocklist-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: example-route
  authorization:
    defaultAction: Allow # 1
    rules:
      - action: Deny # 2
        principal:
          clientCIDRs:
            - 192.168.100.0/24
```

1. 默认动作允许所有流量
2. 拒绝特定 IP 的流量

更多详情请参见 IP Allowlist/Denylist。

> **NOTE**
>
> 如果 Gateway 位于负载均衡器或代理后面，请确保通过 ClientTrafficPolicy 正确配置客户端 IP 检测。

URL 重写

OCP Route 配置

在 OCP 中，URL 重写通过注解实现：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/rewrite-target: /new-path (1)
spec:
  host: example.com
  path: /old-path
  to:
    kind: Service
    name: example-service
```

(1) 重写目标路径

Gateway API 配置

在 Gateway API 中，URL 重写通过过滤器配置：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /old-path
      filters:
        - type: URLRewrite # 1
          urlRewrite:
            path:
              type: ReplacePrefixMatch
              replacePrefixMatch: /new-path
      backendRefs:
        - name: example-service
          port: 8080
```

1. URL 重写过滤器

更多详情请参见 URL Rewrite ↗。

跨命名空间路由准入

OCP Route 配置

在 OCP 中，路由准入策略控制不同命名空间的路由是否可以声明相同主机名，通过 Ingress Operator 在集群级别配置。

Gateway API 配置

在 Gateway API 中，跨命名空间访问控制在 Gateway 监听器级别配置：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: gateway-ns
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: http
      protocol: HTTP
      port: 80
      allowedRoutes: # 1
        namespaces:
          from: All  # 允许所有命名空间的路由
```

1. 配置允许附加路由的命名空间范围

`allowedRoutes.namespaces.from` 的选项：

- `Same`：仅允许与 Gateway 同命名空间的路由
- `All`：允许任意命名空间的路由
- `Selector`：允许匹配标签选择器的命名空间的路由

更多详情请参见 Cross-Namespace Routing。

> **NOTE**
>
> 与 OCP 的集群范围准入策略不同，Gateway API 在每个 Gateway 监听器级别控制，提供更细粒度的权限管理。

默认 TLS 证书用于 Ingress

OCP Route 配置

在 OCP 中，未配置 TLS 的路由可以使用在 Ingress Controller 级别配置的默认证书。

Gateway API 配置

在 Gateway API 中，在 Gateway 监听器上配置默认 TLS 证书：

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      tls: (1)
        mode: Terminate
        certificateRefs:
          - name: default-tls-cert
```

(1) 监听器的默认 TLS 证书

任何附加到该监听器且未指定 TLS 配置的 HTTPRoute 都将使用此默认证书。

使用自定义 CA 的 TLS 重新加密

OCP Route 配置

在 OCP 中，重新加密路由在路由器处终止 TLS，并对后端进行重新加密和验证：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: reencrypt ①
    destinationCACertificate: | ②
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
```

① 重新加密终止模式

② 用于验证后端的 CA 证书

Gateway API 配置

在 Gateway API 中，后端 TLS 验证通过 BackendTLSPolicy 配置：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      tls:
        mode: Terminate 1
        certificateRefs:
          - name: frontend-tls
---
apiVersion: gateway.networking.k8s.io/v1
kind: BackendTLSPolicy
metadata:
  name: backend-tls-policy
  namespace: demo
spec:
  targetRefs: 2
    - group: ""
      kind: Service
      name: example-service
  validation: 3
    caCertificateRefs:
      - name: backend-ca-cert
        kind: ConfigMap
    hostname: backend.example.com
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
  hostnames:
    - example.com
  rules:
```

```
    - backendRefs:
        - name: example-service
          port: 8443
```

1. 在 Gateway 处终止 TLS
2. 应用到后端 Service 的策略
3. 后端 TLS 验证配置

更多详情请参见 Backend TLS。

> **NOTE**
>
> CA 证书必须存储在 ConfigMap 中，不能使用 Secret。

使用自定义证书的边缘终止

OCP Route 配置

在 OCP 中，边缘终止通过路由的 TLS 配置实现：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: edge (1)
    certificate: | (2)
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    key: | (3)
      -----BEGIN PRIVATE KEY-----
      ...
      -----END PRIVATE KEY-----
```

1. 边缘终止模式
2. TLS 证书
3. TLS 私钥

Gateway API 配置

在 Gateway API 中，TLS 终止配置在 Gateway 监听器上：

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: example-tls
  namespace: demo
type: kubernetes.io/tls ①
data:
  tls.crt: <base64-encoded-cert>
  tls.key: <base64-encoded-key>
---
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: https
      protocol: HTTPS ②
      port: 443
      tls:
        mode: Terminate ③
        certificateRefs: ④
          - name: example-tls
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
      sectionName: https
  hostnames:
    - example.com
  rules:
    - backendRefs:
        - name: example-service
          port: 8080
```

1. TLS Secret 类型
2. HTTPS 协议
3. TLS 终止模式
4. 引用 TLS Secret

TLS 透传

OCP Route 配置

在 OCP 中，透传路由不终止 TLS：

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: passthrough (1)
```

1. 透传终止模式

Gateway API 配置

在 Gateway API 中，TLS 透传使用 TLSRoute：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: tls-passthrough
      protocol: TLS # 1
      port: 443
      tls:
        mode: Passthrough # 2
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TLSRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
      sectionName: tls-passthrough
  hostnames: # 3
    - example.com
  rules:
    - backendRefs:
        - name: example-service
          port: 8443
```

1. TLS 协议
2. 透传模式
3. 用于路由的 SNI 主机名

更多详情请参见 TLS Passthrough 和 TLSRoute 规范。

> **NOTE**
>
> TLSRoute 使用 SNI（服务器名称指示）进行路由决策，不终止 TLS。这意味着：
>
> - 无法访问 HTTP 头或请求内容进行路由决策

- 无法应用 L7 过滤器（头部修改、URL 重写等）
- 路由纯粹基于 SNI 主机名

功能对比总结

功能	OCP Route	Gateway API	备
基础 HTTP 路由	Route spec	HTTPRoute	直
基于路径的路由	`.spec.path`	`.spec.rules[].matches[].path`	G A 多
超时	注解	HTTPRoute `.spec.rules[].timeouts`	G A 超
HSTS	注解	ResponseHeaderModifier 过滤器	G A 强
会话亲和	注解	HTTPRoute `.spec.rules[].sessionPersistence`	G A 持
头部修改	注解	RequestHeaderModifier/ResponseHeaderModifier 过滤器	G A 过
连接限制	注解	ClientTrafficPolicy	应 G 另 代
速率限制	注解	BackendTrafficPolicy	G A

功能	OCP Route	Gateway API	备
IP 允许/阻止列表	注解	SecurityPolicy 带授权规则	原 cl
URL 重写	注解	URLRewrite 过滤器	G A 重
跨命名空间	集群级策略	Gateway `.spec.listeners[].allowedRoutes`	G A 器
默认 TLS 证书	Controller 级别	Gateway 监听器 TLS	逐 置
TLS 重新加密	`.spec.tls.termination: reencrypt`	BackendTLSPolicy	需 策
TLS 边缘终止	`.spec.tls.termination: edge`	Gateway 监听器 TLS + HTTPS	证 S
TLS 透传	`.spec.tls.termination: passthrough`	TLSRoute + Passthrough 模式	使 T 君 H

迁移策略

从 OCP Routes 迁移到 Gateway API 时：

1. 先创建 **Gateway**：创建包含适用监听器的 Gateway 资源

- 部署 Gateway 于相同命名空间或专用网关命名空间
- 配置所需的所有协议监听器（HTTP、HTTPS、TLS、TCP、UDP）

- 确保 Gateway 服务已创建且可访问

2. 转换 **Routes** 为 **HTTPRoutes**：逐个迁移 OCP Route 为 HTTPRoute，从简单路由开始
    - 先迁移非生产或低流量路由以降低风险
    - 验证主机名和路径匹配配置
    - 在添加高级功能前测试基本连通性
3. 应用策略：针对需要策略的功能（BackendTrafficPolicy、SecurityPolicy、ClientTrafficPolicy），在基础路由正常后创建并附加
    - 每次添加一个策略并验证功能
    - 监控是否有异常行为或性能影响
4. 逐步测试：验证每一步迁移后再进行下一条路由迁移
    - 监控：检查 Gateway 和 Route 状态条件是否有错误
    - 功能测试：使用 curl 或自动化测试验证所有路由正常
    - 性能测试：对比响应时间和吞吐量与 OCP Routes
    - 验证检查：
        - 确认 TLS 证书正确应用
        - 测试会话亲和和负载均衡行为
        - 验证速率限制和安全策略
        - 检查头部修改和 URL 重写
5. 更新 **DNS**：验证无误后，更新 DNS 指向新的 Gateway 服务
    - 双运行期（推荐）：部分流量指向新 Gateway，同时保持 OCP Routes 活跃以便回滚
    - 通过加权 DNS 或金丝雀发布逐步切换流量
    - 切换期间监控错误率、延迟和应用指标
6. 回滚方案：迁移后若发现问题
    - 将 DNS 指回 OCP Routes
    - DNS 切换后至少保持 OCP Routes 活跃 24-48 小时
    - 记录导致问题的配置差异
    - 如需回滚，制定相关沟通计划

相关文档

- 配置 GatewayAPI Gateway
- 配置 GatewayAPI Route
- 配置 GatewayAPI 策略
- Envoy Gateway 任务

故障排除

如何解决 **ARM** 环境下的节点间通 查找错误原因

如何解决 **ARM** 环境下的节点间通信问题？

在使用较低内核版本和某些国产网卡时，开启 Checksum Offload 后可能出现网卡计算校验和错误的问题，导致 Kube-OVN Overlay 网络中节点间通信失败。具体解决方案如下：

- 方案一：升级内核版本。建议将内核版本升级至 4.19.90-25.16.v2101 或更高版本。
- 方案二：关闭 **Checksum Offload**。如果暂时无法升级内核版本且出现节点间通信问题，可以通过以下命令关闭物理网卡的 Checksum Offload。

  ```
  ethtool -K eth0 tx off
  ```

查找错误原因

错误请求响应头中的 `X-ALB-ERR-REASON` 字段将指示错误的原因。

错误原因可能是：

```
InvalidBalancer : no balancer found for xx # 表示未找到该服务的任何 endpoint
BackendError : read xxx byte data from backend # 表示后端确实返回了响应，错误代码不是由 alb 引起的。
InvalidUpstream : no rule match # 表示请求未匹配任何规则，因此 alb 返回了 404。
```