

网络

概览

概览

- 理解网络
- 核心网络层和组件
- GatewayAPI
- Service、Ingress、Gateway API 的比较
- 网络流量流向

网络操作器

MetalLB Operator

- 了解 MetalLB Operator
- VIP 托管节点选择算法
- 自定义资源定义
- 约束与限制
- 安装与卸载 MetalLB Operator
- 升级 MetalLB Operator

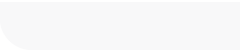
Ingress Nginx Operator

- Overview
- Installation
- Configuration Via Web Console
- Configuration Via YAML
- Additional Resources

Envoy Gate

- 概述
- 通过 Envoy Gat
- 下一步

ALB Operat



网络安全

理解网络策略 API	管理员网络策略	网络策略
介绍	说明	通过 Web 控制台
KUBE-OVN 对网络策略的评估	通过 Web 控制台创建 AdminNetworkPolicy	通过 CLI 创建 N
带作用域的 AdminNetworkPolicy 与带命名	通过 CLI 创建 AdminNetworkPolicy 或 Bas	参考

Ingress 和负载均衡

使用 Envoy Gateway 进行 Ingress 和负载均衡
概述
网络流量架构
配置路由
配置 Ingress 集群流量
负载均衡策略
TLS 和安全性
与 MetalLB 集成
故障排查

操作指南

配置域名

- 域名示例自定义资源（CR）
- 通过 Web 控制台创建域名
- 通过 CLI 创建域名
- 在多个集群中使用同一域名
- 后续操作
- 其他资源

创建证书

- 使用 Web 控制台创建证书

配置 Service

- 为什么需要 Service
- ClusterIP 类型

配置 Ingress

- 实现方式
- Ingress 示例：
- 使用 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 对等方

配置 GatewayAPI 策略

- 概述
- 前提条件
- Policy Attachment 基础
- Policy Attachment 概览
- 在 Web Console 中创建 Policy
- SecurityPolicy
- BackendTLSPolicy
- ClientTrafficPolicy
- BackendTrafficPolicy
- 相关任务

配置 GatewayAPI Route

- 概述
- 前提条件
- 配置
- 视图

配置 ALB

- ALB
- Frontend

配置 NodeLocal DNSCache

概述

主要特性

重要说明

安装

工作原理

配置

配置 CoreDNS

概述

配置

网络可观测性

网络可观测性

实用指南

Ingress-Nginx 的任务

前置条件

最大连接数

请求超时

会话亲和性（Sticky Sessions）

Header 修改

URL 重写

HSTS（HTTP Strict Transport Security）

限流

WAF

转发 Header 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

软数据中心 I

前提条件

操作步骤

验证

Kube OVN

配置 Endpoint Health Checker

概述

主要功能

alb

保留源 IP

安装

工作原理

如何启用

卸载

任务：从 **OC**

简介

前提条件

基本 HTTP Rou

Route 超时

HTTP Strict Tra

基于 Cookie 的

基于路径的路由

Header 修改

连接限制

速率限制

IP Allowlist/Bloc

URL 重写

跨命名空间 Rou

Ingress 的默认

使用自定义 CA

使用自定义证书

TLS Passthroug

功能对比摘要

迁移策略

相关文档

故障排除

如何解决 **ARM** 环境中的节点间通 查找错误原因

概览

目录

理解网络

- 服务发现与负载均衡

- 流量管理与路由

- 网络安全

- 可观测性

- 核心网络层和组件

- GatewayAPI

 - 用于 Ingress 的 Gateway API

 - Gateway API 概念

- Service、Ingress、Gateway API 的比较

 - 面向 L4 (TCP/UDP) 流量

 - LoadBalancer Service (推荐)

 - Gateway API TCP/UDP Routes

 - 建议

 - 面向 L7 (HTTP/HTTPS) 流量

 - Ingress

 - GatewayAPI (推荐)

- 网络流量流向

理解网络

在云原生环境中，网络的核心作用类似于中枢神经系统和循环系统。它是实现弹性、韧性和可观测性等关键应用特性的基础支撑。

云原生网络的职责可以分解为以下几个关键领域：

服务发现与负载均衡

职责：自动发现运行中的微服务实例，并在它们之间智能分配流量。

实现：

- 服务发现
 - Pod 向中心注册表（例如 `etcd`）注册
 - Service 查询注册表以查找可用实例
 - Kubernetes `CoreDNS` 通过 Service 名称提供内置发现能力
- 负载均衡
 - Kubernetes `Service` 资源获得虚拟 IP（VIP）
 - 作为内置负载均衡器，将流量转发到后端 Pod（Endpoints）
 - 在各个实例之间均匀分配请求

流量管理与路由

职责：通过精细化规则控制流量，以支持高级部署模式。

实现：

- **Ingress**
 - 集群面向外部 HTTP/HTTPS 流量的“入口网关”
 - 基于主机名/路径的路由
 - SSL 终止

- **Service Mesh** (Istio、Linkerd)
 - 金丝雀发布：将一定比例的流量导向新版本
 - 故障注入：模拟服务故障，用于韧性测试
 - 超时、重试与熔断：提升容错能力
 - 流量镜像：将生产流量复制到测试环境

网络安全

职责：为已授权的服务通信强制执行“零信任”安全模型。

实现：

- 网络策略
 - 类似防火墙的 Pod 通信规则
 - 定义哪些 Pod 可以相互通信
 - 示例：只有前端 Pod 可以访问后端数据库 Pod
- 双向 **TLS** (mTLS)
 - 对所有服务间通信进行加密和身份认证
 - 确保数据安全和身份验证
 - 广泛用于 Service Mesh

可观测性

职责：清晰呈现网络行为，帮助运维人员理解流量模式，并快速诊断连通性或性能问题。

实现：

- 持续可见性
 - 使用 NetObserve 收集集群范围的网络流日志
 - 使用 Kubernetes 元数据丰富流记录
 - 存储并查询数据，用于排障和审计

- 按需故障排查
 - 使用 CLI 进行流量或数据包抓取，开展短期分析
 - 支持在 TUI 模式下实时检查，或在后台执行抓取任务
 - 将结果导出到 Wireshark 等离线分析工具
- 典型用例
 - 识别连接丢失、延迟热点和异常流量路径
 - 验证服务间通信并检测异常
 - 结合汇总流量和原始数据包调查事件

了解更多关于[网络可观测性](#)。

核心网络层和组件

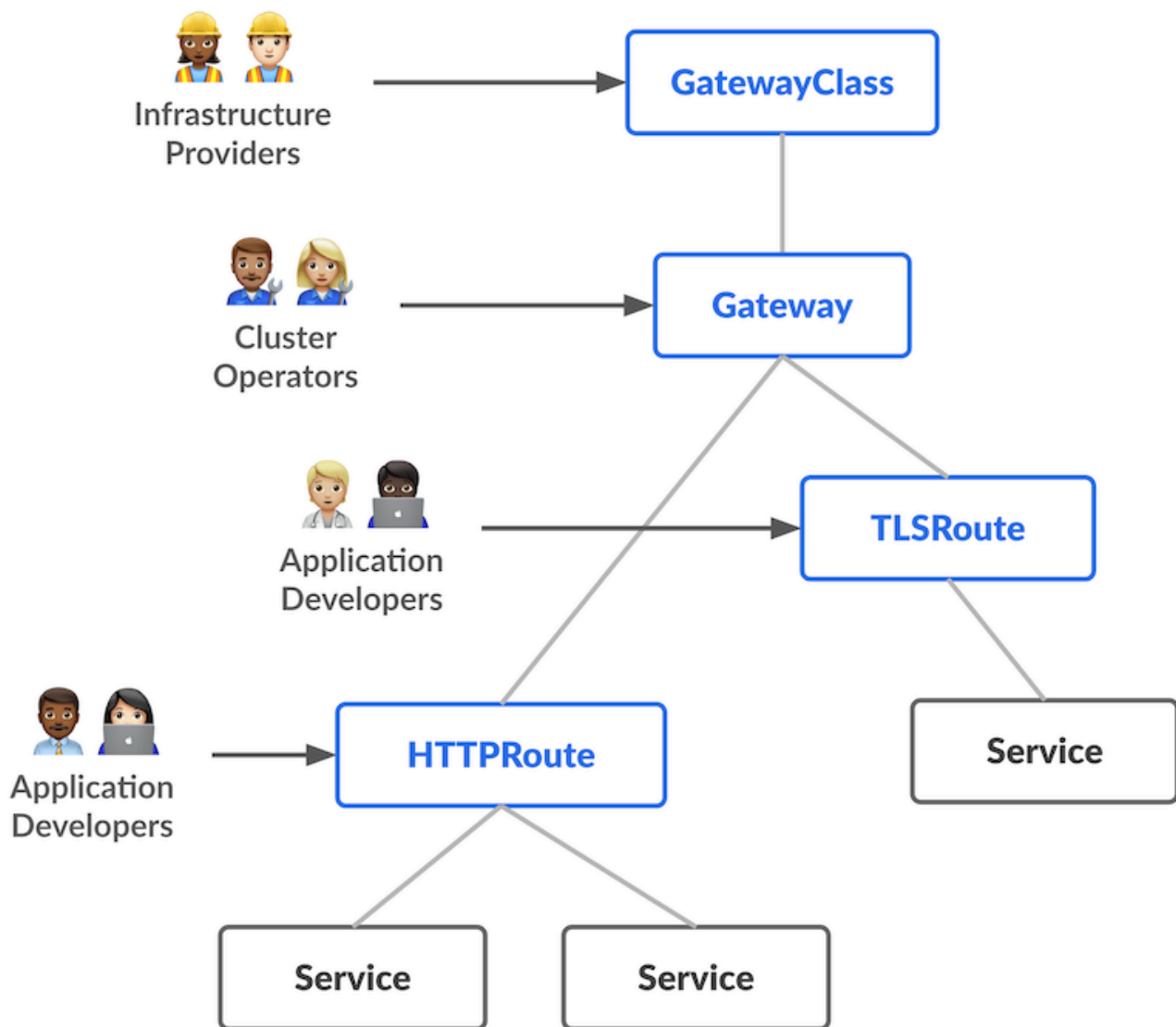
容器网络是一套面向云原生应用的综合性网络解决方案，旨在确保集群内的东西向通信无缝进行，并高效管理跨外部网络的南北向流量，同时提供必要的网络功能。它由以下核心组件组成：

- 用于管理集群内东西向流量的 Container Network Interfaces（CNIs）。
- 用于管理 HTTPS 入口流量的 Ingress Gateway Controller ALB（已弃用）、NGINX Ingress Controller 或 Envoy Gateway（推荐）。
- 用于处理 LoadBalancer 类型 Service 的 MetalLB。
- 此外，它还提供了强大的网络安全和加密功能，以确保通信安全。

GatewayAPI

Gateway API 是 Kubernetes 的官方项目，专注于 Kubernetes 中的 L4 和 L7 路由。该项目代表了 Kubernetes Ingress、负载均衡和 Service Mesh API 的下一代形态。从一开始，它就被设计为通用、富表达力且面向角色的。

整体资源模型聚焦于 3 个不同的角色及其对应的资源，这些角色预期管理这些资源：



该 API 中的大部分配置都包含在路由层中。这些特定于协议的资源（HTTPRoute、GRPCRoute 等）为 Ingress 和 Mesh 提供了高级路由能力。

用于 Ingress 的 Gateway API

在使用 Gateway API 管理入口流量时，Gateway 资源定义了一个流量访问点，流量可以在多个上下文之间进行路由——例如，从集群外部到集群内部（南北向流量）。

每个 Gateway 都关联一个 GatewayClass，它描述了将实际处理该 Gateway 流量的网关控制器类型；随后，各个路由资源（例如 HTTPRoute）会与 Gateway 资源关联。将这些不同职责拆分为独立资源，是 Gateway API 面向角色特性的重要组成部分，同时也允许存在多种网关控制器（由 GatewayClass 资源表示）。

Gateway API 概念

以下设计目标驱动了 Gateway API 的相关概念。这些目标展示了 Gateway 如何改进当前诸如 Ingress 之类的标准。

- 面向角色 - Gateway 由一组 API 资源组成，这些资源建模了使用和配置 Kubernetes Service 网络的组织角色。
- 可移植 - 这并不是一种改进，而是应当保持不变的特性。就像 Ingress 是一个具有众多实现的通用规范一样，Gateway API 也被设计为一个可移植的规范，并由多种实现支持。
- 富表达力 - Gateway API 资源支持诸如基于请求头的匹配、流量加权以及其他功能；这些能力在 Ingress 中只能通过自定义注解实现。
- 可扩展 - Gateway API 允许在 API 的不同层级关联自定义资源。这使得在 API 结构的合适位置进行细粒度定制成为可能。

其他一些值得注意的能力包括：

- GatewayClasses - GatewayClasses 将负载均衡实现类型正式化。这些 class 使用户能够轻松且明确地理解 Kubernetes 资源模型中可用的能力类型。
- 共享 Gateway 和跨 Namespace 支持 - 通过允许独立的 Route 资源附加到同一个 Gateway，它们可以共享负载均衡器和 VIP。这使得团队（即使跨 Namespace）也能在不直接协调的情况下安全地共享基础设施。
- 类型化 Routes 和类型化后端 - Gateway API 支持类型化 Route 资源，也支持不同类型的后端。这使得该 API 能够灵活支持多种协议（如 HTTP 和 gRPC）以及多种后端目标（如 Kubernetes Services、存储桶或函数）。

有关 Gateway API 的更详细说明，请参阅 [gateway-api documentation](#)。

Service、Ingress、Gateway API 的比较

灵雀云容器平台支持 Kubernetes 生态中的多种入口流量规范。比较 Service、Ingress 和 Gateway API，以选择适合你的工作负载和运维需求的流量入口模式。

面向 L4 (TCP/UDP) 流量

LoadBalancer 类型的 Service 和 Gateway API (TCP/UDP Routes) 都可以将四层流量暴露到外部。不过，它们在实现方式和性能特征上有显著差异。

LoadBalancer Service (推荐)

实现方式：内核空间转发

- 流量转发由 Linux 内核直接处理 (iptables/IPVS/eBPF)
- 开销最小，性能接近原生

优势：

- 高性能、低延迟
- 更低的 CPU 和内存开销
- 经过长期验证且成熟的技术

Gateway API TCP/UDP Routes

实现方式：用户空间代理

- 由 Envoy Gateway（或其他网关控制器）实现
- 流量必须从内核空间进入用户空间，再返回内核空间
- 在应用层引入额外处理开销

劣势：

- 相比内核空间方案性能下降
- 资源消耗更高 (CPU/内存)
- 由于用户空间上下文切换带来额外延迟

建议

我们建议将 **LoadBalancer** 类型的 **Service** 用于 **L4** 流量路由，因为它们具有更优的性能和更低的资源开销。

目前，我们通过 [MetalLB](#) 支持 LoadBalancer 类型的 Service。

面向 L7 (HTTP/HTTPS) 流量

虽然 Ingress、GatewayAPI 都可以将 L7 流量暴露到外部，但它们的能力各不相同。

Ingress

Ingress 是 Kubernetes 社区采用的标准规范，推荐作为默认使用方案。Ingress 由平台管理员管理的 ALB 实例处理。

GatewayAPI (推荐)

Gateway API 是 Kubernetes 的下一代路由标准，旨在解决 Ingress 的局限性，并提供更强大、更灵活且标准化的流量管理能力。与 Ingress 相比，Gateway API 具有以下优势：

1. 面向角色的设计

- 职责清晰分离：基础设施管理员（GatewayClass、Gateway）与应用开发人员（Routes）
- Ingress 将基础设施和应用职责混杂在单一资源中

2. 富表达力的路由能力

- 丰富的匹配规则：HTTP headers、查询参数、方法等
- Ingress 仅限于主机名和路径匹配
- 内置支持流量拆分、镜像和高级策略

3. 协议支持

- 原生支持 HTTP、HTTPS、TCP、UDP、gRPC 和 TLS passthrough
- Ingress 主要聚焦于 HTTP/HTTPS

4. 可扩展性

- 通过 CRD 提供类型安全的扩展点（例如 HTTPRoute filters、policy attachments）
- Ingress 严重依赖厂商特定的注解，容易导致可移植性问题

5. 跨 Namespace 路由

- Route 可以引用不同 Namespace 中的 Service（使用 ReferenceGrant）
- Ingress 通常仅限于同 Namespace 引用

6. 单个 Gateway 支持多个 Listener

- 一个 Gateway 可以处理多个端口、协议和主机名

- Ingress 通常每种配置都需要一个资源

7. 可移植性和标准化

- 中立于厂商的 API，并提供一致性测试
- 减少锁定，提高不同实现之间的互操作性
- Ingress 的实现能力和注解差异很大

8. 附加与选择模型

- Route 通过 `parentRefs` 显式附加到 Gateway listeners
- 关系更清晰，也更便于排障
- Ingress 使用 IngressClass，灵活性较低

目前，我们通过 [Envoygateway](#) 支持 GatewayApi。关于从 ingress 迁移到 gatewayapi，请参阅 [migration guide](#) ↗

网络流量流向

Alauda Container Platform 集群中的示例网络流量流向。

```
External Client (Browser / curl) https://www.example.com
|
▼
DNS Resolution (https://www.example.com) → IP (e.g. 34.23.88.11)
|
▼
External Load Balancer [L4]
|
▼
Envoy Gateway / Ingress NGINX Controller [L7]
|
▼
Kubernetes Service (ClusterIP) [L4]
|
▼
Pod (Application)
```

流向说明：

1. 客户端 (浏览器 / **curl**)

用户向 `https://www.example.com` 发送 HTTPS 请求。客户端工作在 L7，首先发起 DNS 查询。

2. **DNS** 解析

DNS 将域名解析为公共 IP 地址 (例如 `34.23.88.11`)。

3. **[L4]** 外部负载均衡器

工作在 L4 (TCP/UDP)。它会将入站连接转发到集群中的后端节点。示例：AWS NLB、GCP TCP LB、MetalLB。

4. **[L7]** **Envoy Gateway / Ingress** 控制器

工作在 L7 (应用层)。负责：

- TLS 终止
- 基于主机名和路径的路由
- 策略和身份认证

它将流量路由到匹配的 Kubernetes Service。

5. **[L4]** **Kubernetes Service** (虚拟 IP)

作为集群内部的四层负载均衡器运行。根据 selector 将请求分发到后端 Pod。

6. **Pod** (原生应用)

应用运行并处理请求的最终目的地。响应将沿着相反路径返回给客户端。

网络操作器

MetalLB Operator

MetalLB Operator

[了解 MetalLB Operator](#)

[VIP 托管节点选择算法](#)

[自定义资源定义](#)

[约束与限制](#)

[安装与卸载 MetalLB Operator](#)

[升级 MetalLB Operator](#)

Ingress Nginx Operator

Ingress Nginx Operator

[Overview](#)

[Installation](#)

[Configuration Via Web Console](#)

[Configuration Via YAML](#)

[Additional Resources](#)

Envoy Gateway Operator

Envoy Gateway Operator

- 概述
- 通过 Envoy Gateway Operator 安装 Envoy Gateway
- 下一步

ALB Operator

了解 ALB

- 核心组件
- 快速开始
- ALB、ALB Instance、Frontend/FT、Rule、ALB Leader
- 其他资源：

Auth

- 基本概念
- 快速开始
- 相关 Ingress 注解
- forward-auth
- basic-auth
- CR
- ALB 特殊 Ingress 注解
- Ingress-Nginx Auth 相关其他特性

为 ALB 部署

- 方法 1：使用 L
- 方法 2：使用外

在 ALB 中绑

- 对于集群内嵌 A
- 定义

ALB 性能选择的决策依据

- 小型生产环境
- 中型生产环境
- 大型生产环境
- 特殊场景部署建议
- 负载均衡器使用模式选择

ALB 中的负

- 概述

L4/L7 超时

- 基本概念

HTTP 重定向

- 基本概念
- CRD
- Ingress 注解
- 端口级重定向
- 规则级重定向

ModSecurity

- 术语
- 操作步骤
- 相关说明
- 配置示例

ALB 与 Ingress-NGINX 注解兼容性

- 基本概念
- 支持的 ingress-nginx 注解

Header Modification

- 基本概念

OTel

- 术语
- 前提条件
- 操作步骤

CORS

- 基本概念
- CRD

URL 重写

TCP/HTTP 代理

- 基本概念
- CRD

ALB 监控

- 术语
- 操作步骤
- 监控指标

MetalLB Operator

目录

了解 MetalLB Operator

术语

MetalLB 高可用原则

VIP 托管节点选择算法

计算公式

应用示例

自定义资源定义

IPAddressPool

L2Advertisement

BGPPeer

BGPAdvertisement

BFDDProfile (可选)

约束与限制

安装与卸载 MetalLB Operator

升级 MetalLB Operator

了解 MetalLB Operator

MetalLB Operator 为无法使用云负载均衡服务的本地或裸金属环境提供了 Kubernetes 原生的负载均衡器实现。

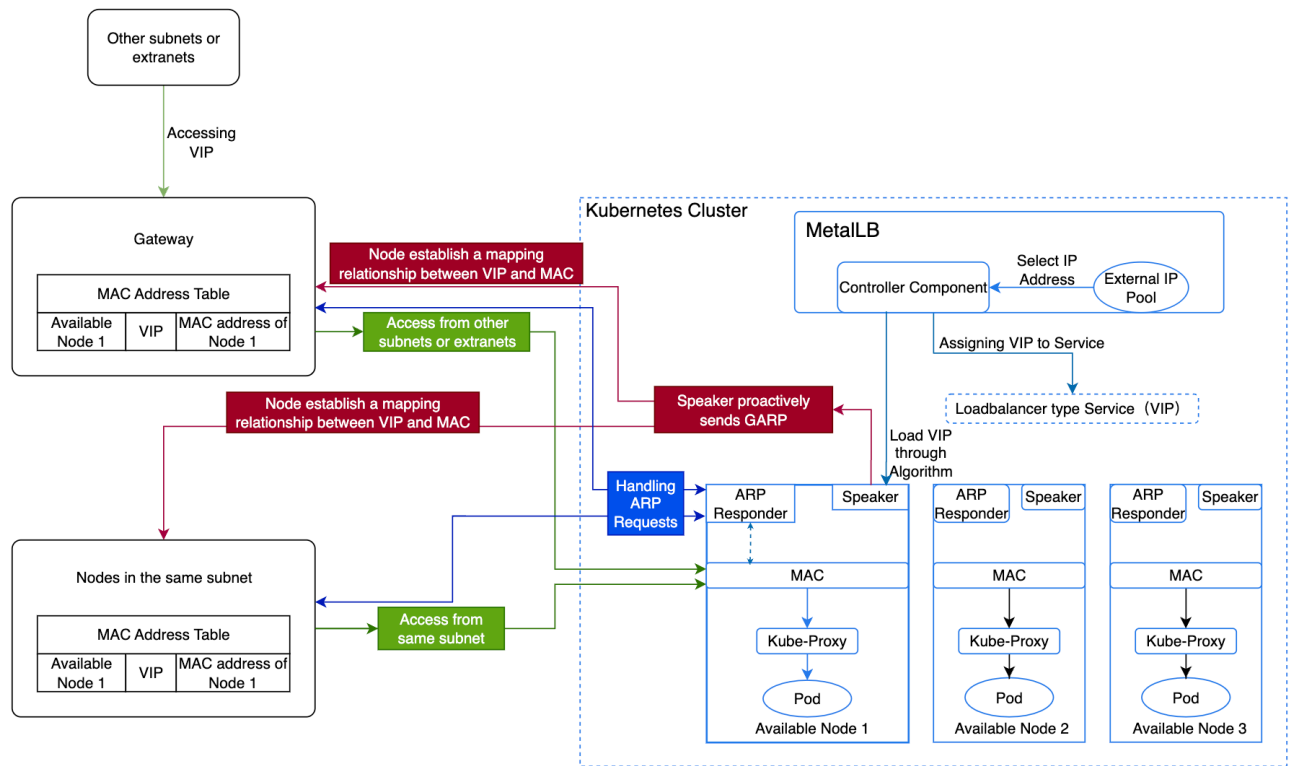
它通过为 LoadBalancer 类型的 Service 资源分配外部 IP 地址，并通过二层（ARP/NDP）或三层（BGP）进行地址通告，使这些服务能够正常工作。

该 Operator 自动化管理 MetalLB 组件的生命周期，包括安装、升级和配置同步。

术语

术语	说明
VIP	虚拟 IP 地址（VIP）是 MetalLB 为 LoadBalancer 类型的内部路由分配的 IP 地址，为外部流量访问集群内服务提供统一的访问入口。
ARP	地址解析协议（ARP）用于将网络层的 IP 地址映射到数据链路层的 MAC 地址。
GARP	Gratuitous ARP（GARP）是一种特殊的 ARP 请求，用于通知网络中其他节点某个 IP 地址与 MAC 地址的绑定关系。与普通 ARP 请求不同，GARP 不等待响应，而是主动将信息广播到网络中。
ARP Responder	MetalLB 的一个组件，负责响应 ARP 请求，将 VIP 映射到节点的 MAC 地址。当节点需要与 VIP 通信时，会发送 ARP 请求以获取对应的 MAC 地址。每个可用节点都有一个 ARP Responder 来响应这些请求，将 VIP 映射到该节点的 MAC 地址。
Controller	MetalLB 的一个组件，动态从外部地址池中分配 VIP 用于 LoadBalancer 类型的内部路由。Controller 监听集群中内部路由的创建和删除事件，按需分配或释放 VIP。
Speaker	MetalLB 的一个组件，根据策略或算法决定哪些节点应承载 VIP 并发送 GARP。它确保节点间的负载均衡，当某节点不可用时，其他节点可以接管 VIP 并发送 GARP，从而实现高可用。

MetalLB 高可用原则



平台默认使用 MetalLB 的 ARP 模式，具体实现流程和原理如下：

- MetalLB 的 Controller 组件从外部地址池选择一个 IP 地址，分配给 LoadBalancer 类型的内部路由作为 VIP。
- MetalLB 根据 [算法](#) 选择一个可用节点作为 leader 托管该 VIP，并转发流量。
- 该节点上的 Speaker 组件主动发送 GARP，在所有节点间建立 VIP 与 MAC 地址的映射关系。
 - 同一子网内的节点在获知 VIP 与可用节点 MAC 地址的映射后，访问 VIP 时会直接与该节点通信。
 - 不同子网的节点则先将流量路由到其子网的网关，再由网关转发到托管 VIP 的节点。
- 当该节点发生故障时，MetalLB 会选择另一个 leader 托管 VIP，并发送 GARP 刷新 Service IP 的 MAC 地址，从而保证高可用。
- 流量到达节点后，Kube-Proxy 将流量转发至对应的 Pod。

VIP 托管节点选择算法

给定负载均衡器 IP 的“leader”（即负责通告该 IP 的节点）的选举是无状态的，工作方式如下：

- 每个 Speaker 收集给定 IP 的潜在通告者列表，考虑活跃的 Speaker、外部流量策略、活跃的 endpoints、节点选择器等因素。
- 每个 Speaker 执行相同的计算：获取“节点+VIP”元素的哈希排序列表，若自身是列表中的第一个，则负责通告该服务。

这样无需记录哪个 Speaker 负责通告某个 IP。

计算公式

公式为：外部地址池数量 = $\text{ceil}(n\text{-vip} / n\text{-node})$ ，其中 ceil 表示向上取整。

注意：若使用虚拟机，则虚拟机数量 = 外部地址池数量 * n。这里 n 必须大于 2，允许最多一个节点故障。

- n-vip：表示 VIP 数量。
- n-node：表示单个节点可承载的 VIP 数量。

应用示例

某公司有 10 个 VIP，每个可用节点可承载 5 个 VIP，允许一个节点故障，如何规划外部地址池数量和可用节点数量？

分析：

需要两个外部地址池和四个可用节点。

- 每个可用节点最多承载 5 个 VIP，意味着一个外部地址池可容纳 5 个 VIP，10 个 VIP 需要两个外部地址池。
- 允许一个节点故障意味着每个地址池必须包含一个托管 VIP 的节点和一个备份节点，因此两个外部地址池各需两个可用节点。

自定义资源定义

MetalLB Operator 安装并管理 metallb.io API 组下的多个 CRD。
这些 CRD 定义了 MetalLB 如何分配和通告 IP 地址。

CRD	Kind	作用
IPAddressPool	IPAddressPool	定义负载均衡服务可用的 IP 地址池。
L2Advertisement	L2Advertisement	配置如何通过 ARP/NDP（二层）通告地址池中的 IP。
BGPPeer	BGPPeer	定义 MetalLB 用于建立路由会话的 BGP 对等体（三层）。
BGPAdvertisement	BGPAdvertisement	指定通过 BGP 通告哪些 IPAddressPool 及其策略。
BFDProfile	BFDProfile	可选；定义双向转发检测（BFD）参数，用于更快检测 BGP 故障。

IPAddressPool

定义 MetalLB 可分配给 `Service.type=LoadBalancer` 的 IP 范围。

```

apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: production-pool
  namespace: metallb-system
spec:
  addresses: ①
    - 192.168.10.100-192.168.10.150
  autoAssign: true ②

```

① 一个或多个 CIDR 或地址范围。

② 默认情况下，MetalLB 会从任何配置的地址池中分配空闲 IP，这可能导致“昂贵”的地址被分配给不需要的服务。

若要防止此行为，可以通过将 `autoAssign` 标志设置为 `false` 来禁用某个地址池的自动分配。

L2Advertisement

用于二层模式，MetalLB 通过 ARP/NDP 通告 IP。

```

apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: l2-adv
  namespace: metallb-system
spec:
  ipAddressPools: ❶
    - production-pool

```

- ❶ 要通告其 IP 的地址池列表。

BGPPeer

```

apiVersion: metallb.io/v1beta1
kind: BGPPeer
metadata:
  name: peer1
  namespace: metallb-system
spec:
  peerAddress: 10.0.0.1 ❶
  peerASN: 64512 ❷
  myASN: 64513 ❸
  holdTime: 90s ❹
  keepaliveTime: 30s
  nodeSelectors: ❺
    - matchLabels:
        alertmanager: 'true'

```

- ❶ 对等路由器的 IP 地址。
- ❷ 对等体的 ASN。
- ❸ MetalLB 使用的 ASN。
- ❹ `holdTime` / `keepaliveTime` : BGP 会话的可选定时器。
- ❺ 可通过配置中的 `node-selectors` 属性限制对等体只作用于特定节点。

BGPAdvertisement

控制如何通过 BGP 通告地址池中的 IP。

```

apiVersion: metallb.io/v1beta1
kind: BGPAdvertisement
metadata:
  name: bgp-adv
  namespace: metallb-system
spec:
  ipAddressPools: ①
    - production-pool
  aggregationLength: 32 ②
  communities: ③
    - no-export

```

- ① 要通告的地址池。
- ② 路由聚合前缀长度。
- ③ BGP 社区属性。

BFDProfile（可选）

用于定义 BGP 对等体的 BFD（双向转发检测）参数。

```

apiVersion: metallb.io/v1beta1
kind: BFDProfile
metadata:
  name: fast-failover
spec:
  receiveInterval: 300
  transmitInterval: 300
  detectMultiplier: 3

```

约束与限制

外部地址的 IP 资源必须满足以下条件：

- 外部地址池必须与可用节点在二层（L2）互联。
- IP 必须可被平台使用，且不能包含物理网络已使用的 IP，如网关 IP。
- 不能与集群使用的网络重叠，包括 Cluster CIDR、Service CIDR、子网等。

- 在双栈环境中，确保同一外部地址池中同时存在 IPv4 和 IPv6 地址，且两者数量均大于 0，否则双栈 LoadBalancer 类型的内部路由无法获取外部访问地址。
- 在 IPv6 环境中，节点的 DNS 必须支持 IPv6，否则 MetalLB 插件无法成功部署。

安装与卸载 MetalLB Operator

可通过 Alauda Container Platform Web 控制台的 Cluster Plugin 安装 Operator。

1. 进入 [Administrator](#) -> [Marketplace](#) -> [Cluster Plugin](#)
2. 输入关键词 [MetalLB](#)。
3. 点击 ，选择 **Install**。
4. 安装完成后，点击 ，选择 **Uninstall**。

升级 MetalLB Operator

通过 Alauda Container Platform Web 控制台上传包的方式升级 MetalLB Operator。

1. 进入 [Administrator](#) -> [Marketplace](#) -> [Upload Package](#)

了解更多关于 [Upload Package](#) 的内容。

Ingress Nginx Operator

目录

Overview

Installation

Configuration Via Web Console

Configuration Via YAML

Ingress Sharding

创建处理所有命名空间 Ingress 的 IngressNginx

创建处理特定命名空间 Ingress 的 IngressNginx

通过 LoadBalancer 暴露

高级 Controller 部署设置

SSL 透传

IPv4 和 IPv6 双栈支持

Additional Resources

与官方 Chart 默认值的差异

Overview

Ingress Nginx Operator 用于简化 Ingress-Nginx 部署的高级管理。

该 Operator 简化了 Ingress-Nginx 实例的部署、配置和维护流程。其运行架构如下：Ingress-Nginx Operator 持续监控类型为 `ingressnginxes.ingress-nginx.alauda.io`

(IngressNginx) 的自定义资源，并自动为每个定义的 IngressNginx 资源创建对应的 Ingress-Nginx 实例。该架构允许通过直接操作 IngressNginx 自定义资源来管理 Ingress-Nginx 的功能和行为。

Installation

1. 进入 `Administrator -> Marketplace -> OperatorHub`
2. 找到 `Alauda build of Ingress NGINX Controller` 并点击 `Install`

Configuration Via Web Console

我们通过用户界面提供基础配置选项，系统会为您生成对应的 YAML。对于更复杂的配置，您可以直接编辑 YAML。

完成 Ingress Nginx Operator 安装后：

1. 进入 `All Instances` 标签页
2. 点击 `Create`，在弹出的对话框中找到 IngressNginx 实例类型区域并点击 `Create`

字段	说明	YAML 路径
Name	Ingress Nginx 实例名称	<code>.metadata.name</code>
Namespace	Ingress Nginx 实例所在命名空间	<code>.metadata.namespace</code>
Replica	Ingress Nginx 部署的副本数	<code>.spec.controller.replicaCount</code>
Resources	Ingress Nginx 部署的资源请求和限制	<code>.spec.controller.resources</code>
Service Type	Ingress Nginx 的服务类型	<code>.spec.controller.service.type</code>

字段	说明	YAML 路径
Ingress Scope	控制处理哪些命名空间的 Ingress 资源	<code>.spec.controller.scope.namespaceSelector</code> ，详见 ingress-sharding

Configuration Via YAML

默认情况下，Ingress Nginx Operator 会在与对应 IngressNginx 自定义资源相同的命名空间中部署 Ingress Nginx 实例。

Ingress Sharding

默认情况下，Ingress Nginx 会处理所有命名空间中明确指定其 IngressClass 名称或未指定任何 IngressClass 的 Ingress 资源。您可以使用 `.controller.scope.namespaceSelector` 限制某个实例只处理特定命名空间。例如，下面的 IngressNginx 资源 `demo-scope` 只处理带有标签 `cpaas.io/project=demo` 的命名空间中的 ingress。

创建处理所有命名空间 Ingress 的 IngressNginx

```
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo-all
spec:
  controller:
    nodeSelector:
      kubernetes.io/os: linux
    replicaCount: 1
```

创建处理特定命名空间 Ingress 的 IngressNginx

```

apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo-scope
spec:
  controller:
    scope:
      namespaceSelector: 'cpaas.io/project=demo' ❶
      nodeSelector:
        kubernetes.io/os: linux
      replicaCount: 1

```

❶ 格式为 `$LABEL_KEY=$LABEL_VALUE`

通过 LoadBalancer 暴露

默认情况下，Ingress Controller 配置为 ClusterIP 类型服务。若要通过 LoadBalancer 服务将 Ingress Controller 暴露到外部，请应用如下配置：

NOTE

LoadBalancer 服务需要外部负载均衡器集成（云厂商 LB 或 MetalLB）以分配外部 IP。

```

apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
spec:
  controller:
    service:
      type: LoadBalancer

```

使用 MetalLB 指定 LoadBalancer 虚拟 IP：

```

apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
spec:
  controller:
    service:
      type: LoadBalancer
    annotations:
      metallb.universe.tf/loadBalancerIPs: '192.168.2.2' # 期望的 VIP
      metallb.universe.tf/address-pool: 'pool-name' # MetalLB 地址池

```

高级 Controller 部署设置

配置网络模式、副本数、资源限制和节点选择：

```

apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
spec:
  controller:
    hostNetwork: false
    replicaCount: 1
    nodeSelector:
      kubernetes.io/os: linux
    resources:
      limits:
        cpu: 200m
        memory: 256Mi
      requests:
        cpu: 200m
        memory: 256Mi

```

SSL 透传

启用 SSL 透传功能：

CAUTION

启用 SSL 透传后，TLS 终止发生在后端，因此 L7 功能（如请求/响应头操作、WAF、HTTP 到 HTTPS 重定向、部分认证流程）将不会对该流量在 controller 端生效。

```
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
spec:
  controller:
    extraArgs:
      enable-ssl-passthrough: ''
```

IPv4 和 IPv6 双栈支持

```
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
spec:
  controller:
    service:
      ipFamilyPolicy: PreferDualStack
```

Additional Resources

`IngressNginx` 资源的 `.spec` 字段直接对应 Ingress Nginx 的 Helm chart 值。更多配置选项请参考 [官方 Ingress NGINX 文档](#)。

与官方 Chart 默认值的差异

- 默认情况下，每个 IngressNginx 实例会创建名称为 `$ns-$name`，controllerValue 为 `ingress-nginx.cpaas.io/$ns-$name` 的 IngressClass。可通过

`.spec.controller.ingressClassResource.name` 和

`.spec.controller.ingressClassResource.controllerValue` 参数自定义。

2. 默认 `.spec.controller.service.type` 设置为 `ClusterIP`。

3. 默认 `.spec.controller.watchIngressWithoutClass` 设置为 `true`，表示 controller 会处理未指定 IngressClass 的 Ingress 资源。

Envoy Gateway Operator

目录

概述

架构

相关自定义资源

与 Operator 相关的自定义资源

与 `Gateway API` 相关的自定义资源

Envoy Gateway 部署相关的自定义资源

通过 Envoy Gateway Operator 安装 Envoy Gateway

前提条件

安装

步骤 1：安装 Envoy Gateway Operator

步骤 2：使用推荐默认值创建 EnvoyGatewayCtl

通过 YAML 进行配置

通过 EnvoyGatewayCtl 进行高级配置

示例：启用后端路由

下一步

概述

使用 `envoy-gateway-operator` 通过平台托管资源来部署和管理 Envoy Gateway。主要的自定义资源和运行时实例包括 `EnvoyGatewayCtl`、`GatewayClass`、`Gateway`、`EnvoyProxy`、`envoy-gateway instance` 和 `envoy-proxy instance`。

`envoy-gateway-operator` 会将上游 `envoy-gateway` Helm Chart 打包为一个 Operator，因此你可以通过 `EnvoyGatewayCtl` 自定义资源以声明式方式部署和管理 `envoy-gateway instance`，而无需直接管理该 Chart 或其渲染后的资源。

在 ACP 4.3 中，`envoy-gateway-operator` 打包的上游 `envoy-gateway` Helm Chart 版本为 `1.7.0`。

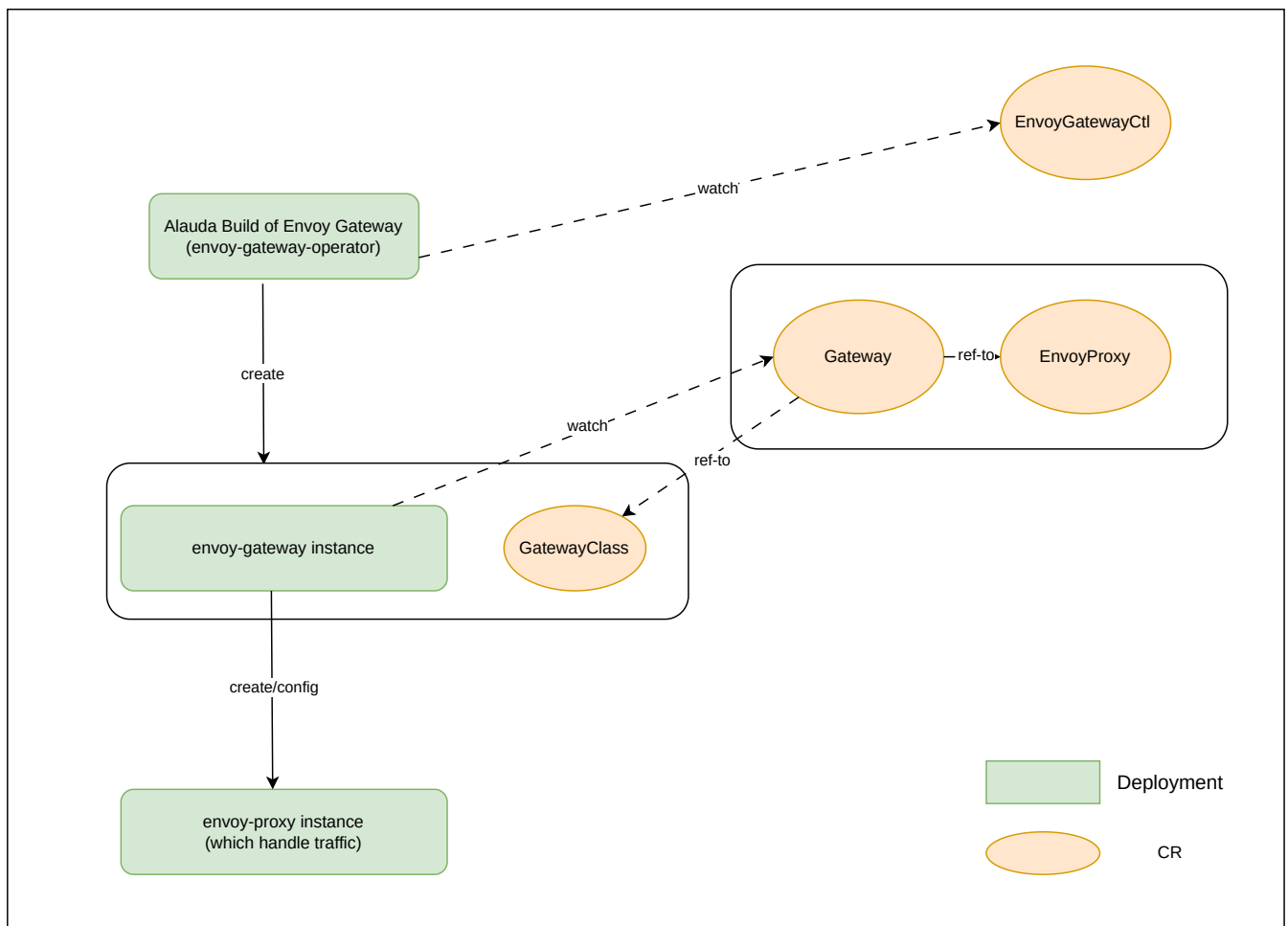
由于该 Operator 由平台维护，因此也会随着平台一起自动升级，这有助于减少手动维护 Envoy Gateway 安装所带来的运维开销。

创建 `EnvoyGatewayCtl` 作为通过平台部署和管理 Envoy Gateway 的入口点。

在理解这些概念并创建 `EnvoyGatewayCtl` 之后，请按照 [下一步](#) 中列出的顺序继续进行 `Gateway`、`Route` 和 `Policy` 配置。

架构

以下工作流展示了 `envoy-gateway-operator`、`EnvoyGatewayCtl`、`GatewayClass`、`Gateway` 和 `EnvoyProxy` 如何协同工作，以部署和管理 Envoy Gateway 及其数据平面。



1. 安装 `envoy-gateway-operator` 并创建 `EnvoyGatewayCtl` 后，operator 会部署一个 `envoy-gateway instance` 并创建对应的 `GatewayClass`。`GatewayClass` 名称遵循 `<namespace>-<name>` 模式。推荐的默认示例是在 `envoy-gateway-operator` 命名空间中使用 `cpaas-default`，这也是 `Create EnvoyGatewayCtl` 页面预填充的默认配置。在这种情况下，生成的 `GatewayClass` 为 `envoy-gateway-operator-cpaas-default`。每个 `EnvoyGatewayCtl` 都必须具有唯一的命名空间和名称组合，因此对应的 `GatewayClass` 名称也是唯一的。创建 `Gateway` 时，你需要选择合适的 `GatewayClass`，以确定由哪个 `envoy-gateway instance` 来管理它。
2. 当你创建一个引用此 `GatewayClass` 的 `Gateway` 时，`envoy-gateway instance` 会接管该 `Gateway`。如果该 `Gateway` 通过 `.spec.infrastructure.parametersRef` 引用了一个 `EnvoyProxy`，则会使用该 `EnvoyProxy` 来控制底层 `envoy-proxy instance` 的部署方式。
3. 随后，`envoy-gateway instance` 会调度并管理相应的 `envoy-proxy instance`，该实例作为数据平面，为该 `Gateway` 处理进入流量。
4. 你可以创建 `HTTPRoute`、`TCPRoute`、`UDPRoute`、`GRPCRoute` 或 `TLSRoute` 资源并将它们附加到 `Gateway`，以定义流量如何匹配并转发到后端 Services。你还可以附加诸如 `ClientTrafficPolicy` 或 `BackendTrafficPolicy` 等流量策略，以实现更高级的行为。

相关自定义资源

以下自定义资源参与 Envoy Gateway 的部署以及流量行为的配置。

与 Operator 相关的自定义资源

自定义资源	描述
EnvoyGatewayCtl	控制某个特定 <code>envoy-gateway instance</code> 的部署。

对于典型使用场景，我们建议每个集群通过 `EnvoyGatewayCtl` 部署一个 `envoy-gateway instance`。

对于高级场景，`EnvoyGatewayCtl` 还允许你在同一集群中部署多个 `envoy-gateway instance` 资源，以获得更强的控制力和隔离性，例如不同的 [deployment mode](#)。

与 Gateway API 相关的自定义资源

这些资源描述了 gateway 的路由语义，包括流量如何进入 gateway、请求如何匹配并转发到后端，以及在流量处理过程中应用哪些策略。

自定义资源	描述
GatewayClass	指示某个 <code>envoy-gateway instance</code> 应管理哪些 <code>Gateway</code> 资源。
Gateway	定义三个核心组件： GatewayClass 引用 - 通过 <code>.spec.gatewayClassName</code> 引用控制此 Gateway 的 GatewayClass Listeners 配置 - 通过 <code>.spec.listeners</code> 指定用于处理流量的端口、主机名和 TLS 证书 - 基础设施参数 - 通过引用 <code>EnvoyProxy</code> (<code>.spec.infrastructure.parametersRef</code>) 来配置副本数、资源等部署细节
Route resources	包括 <code>HTTPRoute</code>、<code>TCPRoute</code>、<code>UDPRoute</code>、<code>GRPCRoute</code> 和 <code>TLSRoute</code>。这些资源定义了四个核心组件： - 多个匹配规则集 - 用于匹配进入流量的各种条件 - 后端目标 - 每个匹配规则集对应的目标 Services - 流量策略 - 按规则配置重试、超时、重定向等

自定义资源	描述
	4. Gateway listeners 绑定 - 通过 <code>.spec.parentRefs</code> 将 route 连接到特定的 <code>Gateway</code> listener
Policies	Policy 资源为 <code>Gateway</code> 和 <code>Route</code> 资源扩展了高级能力。Envoy Gateway 当前提供 <code>SecurityPolicy</code> 、 <code>BackendTLSPolicy</code> 、 <code>ClientTrafficPolicy</code> 和 <code>BackendTrafficPolicy</code> 。

有关这些标准 Gateway API 资源的更多详细信息，请参阅 [Gateway API 概念](#) 和 [配置 GatewayAPI Policy](#)。

Envoy Gateway 部署相关的自定义资源

这些资源控制 Envoy Gateway 及其数据平面的部署方式。请使用以下自定义资源进行部署相关配置。

自定义资源	描述
EnvoyProxy	定义 <code>envoy-proxy instance</code> 的配置，例如副本数、资源以及调度约束

在推荐的部署模式中，每个 `Gateway` 都通过 `.spec.infrastructure.parametersRef` 引用其各自专用的 `EnvoyProxy`。当你在 Web Console 中使用由 `EnvoyGatewayCtl` 创建的 `GatewayClass` 创建 `Gateway` 时，控制台会自动创建一个同名、同命名空间的配套 `EnvoyProxy` 资源。这种一对一映射让你可以通过更新对应的 `EnvoyProxy` 资源来控制 Gateway 的部署配置，例如副本数、资源和调度。

通过 Envoy Gateway Operator 安装 Envoy Gateway

前提条件

在继续之前，请先阅读上文的概念和架构。

安装

步骤 1：安装 Envoy Gateway Operator

1. 导航到 `Administrator -> Marketplace -> OperatorHub`
2. 在 `Networking` 类别下找到 `Alauda build of Envoy Gateway`，然后点击 `Install`

步骤 2：使用推荐默认值创建 EnvoyGatewayCtl

以下默认值与本工作流中推荐的示例一致。

1. 导航到 `Administrator -> Marketplace -> OperatorHub`
2. 在 `Networking` 类别下找到 `Alauda build of Envoy Gateway`，然后点击它以打开详情页面。
3. 导航到 `All Instances` 选项卡
4. 点击 `Create`，然后在显示的对话框中，找到 `EnvoyGatewayCtl` 实例类型部分并点击 `Create`
5. `Create EnvoyGatewayCtl` 页面会显示以下配置选项：

字段	默认值	描述	YAML Path
名称	cpaas-default	Envoy Gateway 实例名称	<code>.metadata.name</code>
命名空间	envoy-gateway-operator	Envoy Gateway 实例的命名空间。每个命名空间只能有一个 <code>EnvoyGatewayCtl</code> 。	<code>.metadata.namespace</code>
从节点	1	Envoy Gateway 部署的从节点数量	<code>.spec.deployment.replicas</code>
部署 Namespace 模式	ControllerNamespace	确定 gateway 部署是安装在 gateway 命名空间还是 controller 命名空间中。详细信息请参见 deployment mode 。	<code>.spec.config.envoyGateway</code>

字段	默认值	描述	YAML Path
资源	1C1G	Envoy Gateway 部署的资源请求和限制	<code>.spec.deployment.resources</code>

建议：对于典型部署，请使用默认值。点击 **Create** 继续。

通过 YAML 进行配置

作为 Web Console 的替代方式，你可以通过应用 YAML 来创建 `EnvoyGatewayCtl`。

默认配置：

```
apiVersion: envoy-gateway.alauda.io/v1
kind: EnvoyGatewayCtl
metadata:
  name: cpaas-default
  namespace: envoy-gateway-operator
spec:
  config:
    envoyGateway:
      provider:
        kubernetes:
          deploy:
            type: ControllerNamespace
          type: Kubernetes
  deployment:
    replicas: 1
    resources:
      limits:
        cpu: '1'
        memory: 1024Mi
      requests:
        cpu: 100m
        memory: 256Mi
```

通过 EnvoyGatewayCtl 进行高级配置

`EnvoyGatewayCtl` 是用于配置 `envoy-gateway instance` 的自定义资源。其 `spec` 会直接映射到上游 `envoy-gateway` Helm Chart 的 values。当你需要修改 `EnvoyGatewayCtl` 时，请参考上游 Chart values 文档中的可用配置选项。

有关可用配置选项的详细信息，请参阅 [envoy-gateway-chart-values](#)。

DANGER

你应仅通过 `EnvoyGatewayCtl` 资源来配置 `envoy-gateway instance`。不要直接修改诸如 ConfigMaps、Deployments 等渲染后的资源，因为这些更改可能会在升级或 reconciliation 期间丢失。

示例：启用后端路由

以下示例展示了如何在 `EnvoyGatewayCtl` 的 spec 中启用后端路由：

```

apiVersion: envoy-gateway.alauda.io/v1
kind: EnvoyGatewayCtl
metadata:
  name: cpaas-default
  namespace: envoy-gateway-operator
spec:
  config:
    envoyGateway:
      extensionApis:
        enableBackend: true # Enable backend routing in EnvoyGatewayCtl n
ot the envoygateway configmap
      provider:
        kubernetes:
          deploy:
            type: ControllerNamespace
          type: Kubernetes
  deployment:
    replicas: 1
    resources:
      limits:
        cpu: '1'
        memory: 1024Mi
      requests:
        cpu: 100m
        memory: 256Mi

```

下一步

在 Envoy Gateway operator 和 `EnvoyGatewayCtl` 就绪后，请按以下顺序继续执行任务：

- [配置 GatewayAPI Gateway](#)
- [配置 GatewayAPI Route](#)
- [配置 GatewayAPI Policy](#)
- [Envoy Gateway 任务](#)

ALB Operator

了解 ALB

核心组件

快速开始

ALB、ALB Instance、Frontend/FT、Rule、

ALB Leader

其他资源：

Auth

基本概念

快速开始

相关 Ingress 注解

forward-auth

basic-auth

CR

ALB 特殊 Ingress 注解

Ingress-Nginx Auth 相关其他特性

为 ALB 部署

方法 1：使用 L

方法 2：使用外

在 ALB 中绑

对于集群内嵌 A
定义

ALB 性能选择的决策依据

小型生产环境

中型生产环境

大型生产环境

特殊场景部署建议

负载均衡器使用模式选择

ALB 中的负

概述

L4/L7 超时

基本概念

HTTP 重定向

基本概念

CRD

Ingress 注解

端口级重定向

CORS

基本概念

CRD

规则级重定向

Header Modification

基本概念

URL 重写

ModSecurity

术语

操作步骤

相关说明

配置示例

OTel

术语

前提条件

操作步骤

TCP/HTTP 代理

基本概念

CRD

ALB 与 Ingress-NGINX 注解兼容性

基本概念

支持的 ingress-nginx 注解

ALB 监控

术语

操作步骤

监控指标

[Alauda Container Platform](#) > [网络](#) > [网络操作器](#) > [ALB Operator](#) > 了解 ALB

了解 ALB

WARNING

ALB 已弃用。请改用 [ingress-nginx-operator](#) 或 [envoy-gateway](#)。

ALB (Another Load Balancer) 是一个由 OpenResty 驱动的 Kubernetes Gateway，凝聚了 Alauda 多年的生产实践经验。

目录

核心组件

快速开始

部署 ALB Operator

部署 ALB 实例

运行示例应用

ALB、ALB Instance、Frontend/FT、Rule、Ingress 和 Project 之间的关系

Ingress

Ingress Controller

ALB

ALB Instance

ALB-Operator

Frontend (缩写 : FT)

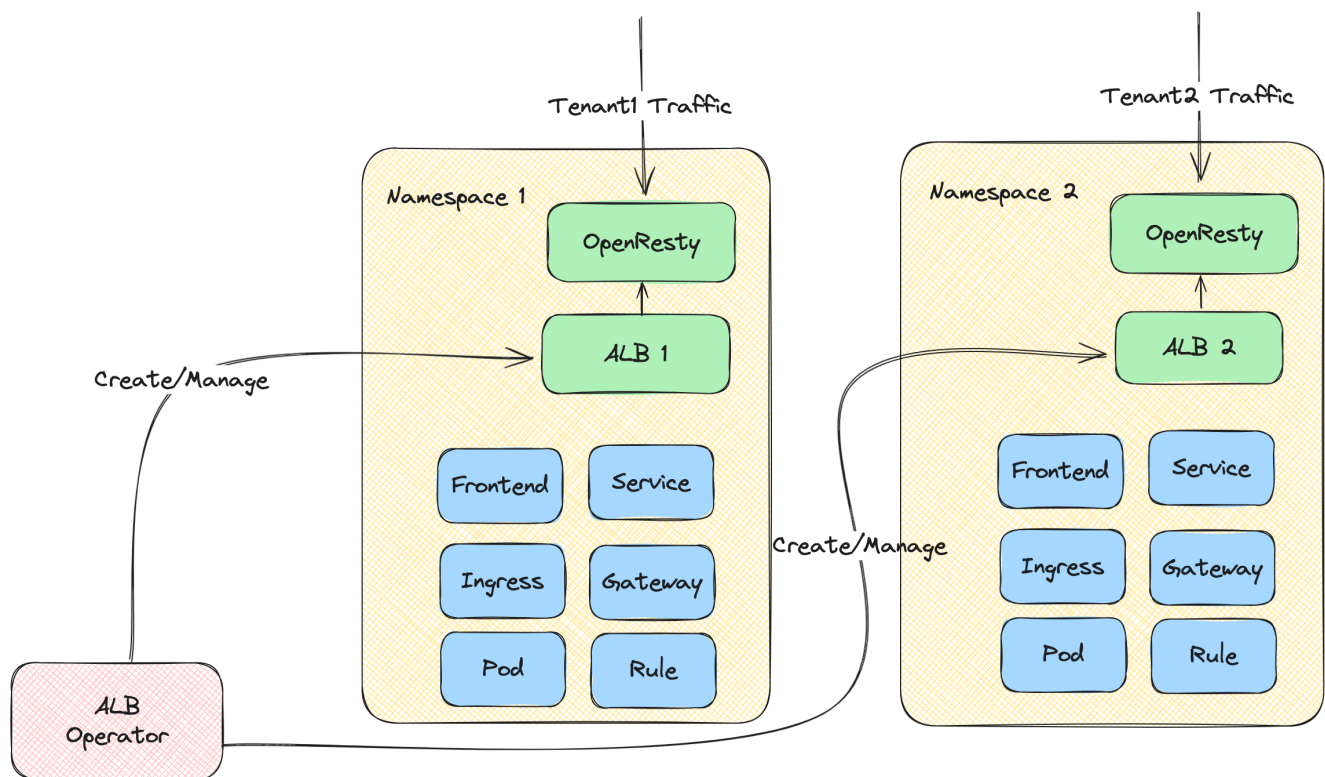
RULE

ALB Leader

Project

其他资源：

核心组件



- **ALB Operator**：负责管理 ALB 实例生命周期的 operator。它会监视 ALB CR，并为不同租户创建/更新实例。
- **ALB Instance**：一个 ALB 实例包括作为数据平面的 OpenResty，以及作为控制平面的 Go controller。Go controller 会监控各种 CR（Ingress、Gateway、Rule 等），并将它们转换为 ALB 专用的 DSL 规则。随后 OpenResty 使用这些 DSL 规则来匹配并处理传入请求。

快速开始

部署 ALB Operator

1. 创建集群。

2.

```
helm repo add alb https://alauda.github.io/alb/;helm repo update;helm search repo|grep alb
```
3.

```
helm install alb-operator alb/alauda-alb2
```

部署 ALB 实例

```
cat <<EOF | kubectl apply -f -
apiVersion: crd.alauda.io/v2beta1
kind: ALB2
metadata:
  name: alb-demo
  namespace: kube-system
spec:
  address: "172.20.0.5" # the ip address of node where alb been deployed
  type: "nginx"
  config:
    networkMode: host
    loadbalancerName: alb-demo
    projects:
      - ALL_ALL
    replicas: 1
EOF
```

运行示例应用



```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
  labels:
    k8s-app: hello-world
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: hello-world
  template:
    metadata:
      labels:
        k8s-app: hello-world
    spec:
      terminationGracePeriodSeconds: 60
      containers:
      - name: hello-world
        image: crccheck/hello-world:latest
        imagePullPolicy: IfNotPresent
---
apiVersion: v1
kind: Service
metadata:
  name: hello-world
  labels:
    k8s-app: hello-world
spec:
  ports:
  - name: http
    port: 80
    targetPort: 8000
  selector:
    k8s-app: hello-world
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-world
spec:
  rules:
```

```
- http:
  paths:
  - path: /
    pathType: Prefix
    backend:
      service:
        name: hello-world
        port:
          number: 80

EOF
```

现在你可以通过 `curl http://${ip}` 访问该应用。

ALB、ALB Instance、Frontend/FT、Rule、Ingress 和 Project 之间的关系

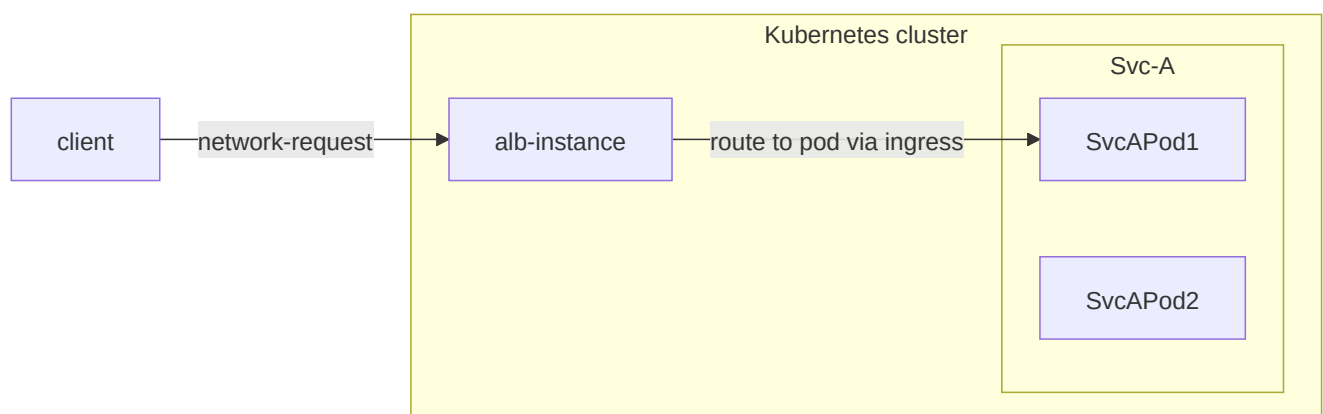
LoadBalancer 是现代云原生架构中的关键组件，充当智能流量路由器和负载均衡器。

要理解 ALB 在 Kubernetes 集群中的工作方式，我们需要了解几个核心概念及其关系：

- ALB 本身
- Frontend (FT)
- Rules
- Ingress 资源
- Projects

这些组件协同工作，以提供灵活且强大的流量管理能力。

下面将说明这些概念在请求路径中的协同方式。每个概念的详细介绍会在单独的文章中展开。



在请求调用链中：

1. 客户端发送一个 HTTP/HTTPS/其他协议请求，最终请求会到达 **ALB** 的一个 **pod**，该 pod（一个 ALB 实例）开始处理此请求。
2. 该 ALB 实例找到一条能够匹配此请求的规则。
3. 如有需要，根据该规则修改/重定向/重写请求。
4. 从该规则配置的 services 中找到并选择一个 pod IP，然后将请求转发到该 pod。

Ingress

Ingress 是 Kubernetes 中的一个资源，用于描述某个请求应发送到哪个 service。

Ingress Controller

一种理解 Ingress 资源并将请求代理到 service 的程序。

ALB

ALB 是一个 Ingress controller。

在 Kubernetes 集群中，我们使用 `alb2` 资源来操作 ALB。你可以使用 `kubectl get alb2 -A` 查看集群中所有的 ALB。

ALB 由用户手动创建。每个 ALB 都有自己的 IngressClass。当你创建 Ingress 时，可以使用 `.spec.ingressClassName` 字段来指定应由哪个 Ingress controller 处理该 Ingress。

ALB Instance

ALB 也可以是运行在集群中的一个 Deployment（由一组 pod 组成）。每个 pod 都称为一个 ALB 实例。

每个 ALB 实例独立处理请求，但所有实例共享属于同一个 ALB 的 Frontend（FT）、Rule 及其他配置。

ALB-Operator

ALB-Operator 是集群中默认部署的一个组件，是用于 ALB 的 operator。它会根据 ALB 资源为每个 ALB 创建/更新/删除 Deployment 及其他相关资源。

Frontend（缩写：FT）

FT 是 ALB 自身定义的一个资源。它用于表示 ALB 实例的监听端口。

FT 可以由 ALB-Leader 创建，也可以由用户手动创建。

ALB-Leader 创建 FT 的场景：

1. 如果 Ingress 有证书，则创建 FT 443 (HTTPS) 。
2. 如果 Ingress 没有证书，则创建 FT 80 (HTTP) 。

RULE

RULE 是 ALB 自身定义的一个资源。它与 Ingress 具有相同的角色，但更加具体。一个 RULE 会唯一关联到一个 FT。

RULE 可以由 ALB-Leader 创建，也可以由用户手动创建。

ALB-Leader 创建 RULE 的场景：

1. 将 Ingress 同步为 RULE。

ALB Leader

在多个 ALB 实例中，会选举出一个作为 leader。leader 负责：

1. 将 Ingress 转换为 Rules。我们会为 Ingress 中的每个 path 创建 Rule。
2. 创建 Ingress 所需的 FT。例如，如果 Ingress 有证书，则创建 FT 443 (HTTPS) ；如果 Ingress 没有证书，则创建 FT 80 (HTTP) 。

Project

从 ALB 的视角来看，Project 是一组 namespaces。

你可以在一个 ALB 中配置一个或多个 Project。当 ALB Leader 将 Ingress 转换为 Rules 时，它会忽略不属于该 Project 的 namespaces 中的 Ingress。

其他资源：

- [配置负载均衡器](#)

Auth

目录

基本概念

什么是 Auth

支持的 Auth 方法

Auth 配置方式

Auth 结果处理

快速开始

部署 ALB

配置 Secret 和 Ingress

验证

相关 Ingress 注解

forward-auth

构造相关注解

auth-url

auth-method

auth-proxy-set-headers

构造 app-request 相关注解

auth-response-headers

Cookie 处理

重定向相关配置

auth-signin

auth-signin-redirect-param

auth-request-redirect

basic-auth

auth-realm

auth-type

auth-secret

auth-secret-type

CR

ALB 特殊 Ingress 注解

Auth-Enable

Ingress-Nginx Auth 相关其他特性

Global-Auth

No-Auth-Locations

注意：与 Ingress-Nginx 不兼容的部分

故障排查

基本概念

什么是 Auth

Auth 是一种在请求到达实际服务之前进行身份验证的机制。它允许你在 ALB 级别统一处理身份验证，而无需在每个后端服务中实现身份验证逻辑。

支持的 Auth 方法

ALB 支持两种主要的身份验证方法：

1. Forward Auth（外部身份验证）

- 发送请求到外部身份验证服务以验证用户身份
 - 适用场景：需要复杂的身份验证逻辑，如 OAuth、SSO 等
-

- 工作流程：
 1. 用户请求到达 ALB
 2. ALB 将身份验证信息转发给身份验证服务
 3. 身份验证服务返回验证结果
 4. 根据身份验证结果决定是否允许访问后端服务

2. Basic Auth（基本身份验证）

- 基于用户名和密码的简单身份验证机制
- 适用场景：简单访问控制，开发环境保护
- 工作流程：
 1. 用户请求到达 ALB
 2. ALB 检查请求中的用户名和密码
 3. 与配置的身份验证信息进行比对
 4. 验证通过则转发到后端服务

Auth 配置方式

1. 全局 Auth

- 在 ALB 级别配置，适用于所有服务
- 在 ALB 或 FT CR 上配置

2. 路径级 Auth

- 在特定 Ingress 路径上配置
- 在特定 Rule 上配置
- 可以覆盖全局 Auth 配置

3. 禁用 Auth

- 针对特定路径禁用身份验证
- 在 Ingress 上通过注解配置：`alb.ingress.cpaas.io/auth-enable: "false"`
- 在 Rule 上通过 CR 配置

Auth 结果处理

- Auth 成功：请求将被转发到后端服务
- Auth 失败：返回 401 未授权错误
- 可以配置 Auth 失败后的重定向行为（适用于 Forward Auth）

快速开始

使用 ALB 配置 Basic Auth

部署 ALB

```
cat <<EOF | kubectl apply -f -
apiVersion: crd.alauda.io/v2
kind: ALB2
metadata:
  name: auth
  namespace: cpaas-system
spec:
  config:
    networkMode: container
    projects:
      - ALL_ALL
    replicas: 1
    vip:
      enableLbSvc: false
    type: nginx
EOF
export ALB_IP=$(kubectl get pods -n cpaas-system -l service_name=alb2-auth -o jsonpath='{.items[*].status.podIP}');echo $ALB_IP
```

配置 Secret 和 Ingress

```
# echo "Zm9vOiRhchIXJHFJQ05aNjFRJdJpb29pSlZVQU1tcHJxMjU4L0NoUDE=" | base64
4 -d # foo:$apr1$qICNZ61Q$2iooiJVUAMmprq258/ChP1
# openssl passwd -apr1 -salt qICNZ61Q bar # $apr1$qICNZ61Q$2iooiJVUAMmprq
258/ChP1
```

```
kubectl apply -f - <<'END'
```

```
apiVersion: v1
```

```
kind: Secret
```

```
metadata:
```

```
  name: auth-file
```

```
type: Opaque
```

```
data:
```

```
  auth: Zm9vOiRhchIXJHFJQ05aNjFRJdJpb29pSlZVQU1tcHJxMjU4L0NoUDE=
```

```
---
```

```
apiVersion: networking.k8s.io/v1
```

```
kind: Ingress
```

```
metadata:
```

```
  name: auth-file
```

```
  annotations:
```

```
    "nginx.ingress.kubernetes.io/auth-type": "basic"
```

```
    "nginx.ingress.kubernetes.io/auth-secret": "default/auth-file"
```

```
    "nginx.ingress.kubernetes.io/auth-secret-type": "auth-file"
```

```
spec:
```

```
  rules:
```

```
    - http:
```

```
      paths:
```

```
        - path: /app-file
```

```
          pathType: Prefix
```

```
          backend:
```

```
            service:
```

```
              name: app-server
```

```
              port:
```

```
                number: 80
```

```
END
```

验证

```
# echo "Zm9vOiJhYXJi" | base64 -d # foo:bar
curl -v -X GET -H "Authorization: Basic Zm9vOmJhcg==" http://$ALB_IP:80/a
pp-file # 应返回 200
# 错误密码
curl -v -X GET -H "Authorization: Basic XXXXOmJhcg==" http://$ALB_IP:80/a
pp-file # 应返回 401
```

相关 Ingress 注解

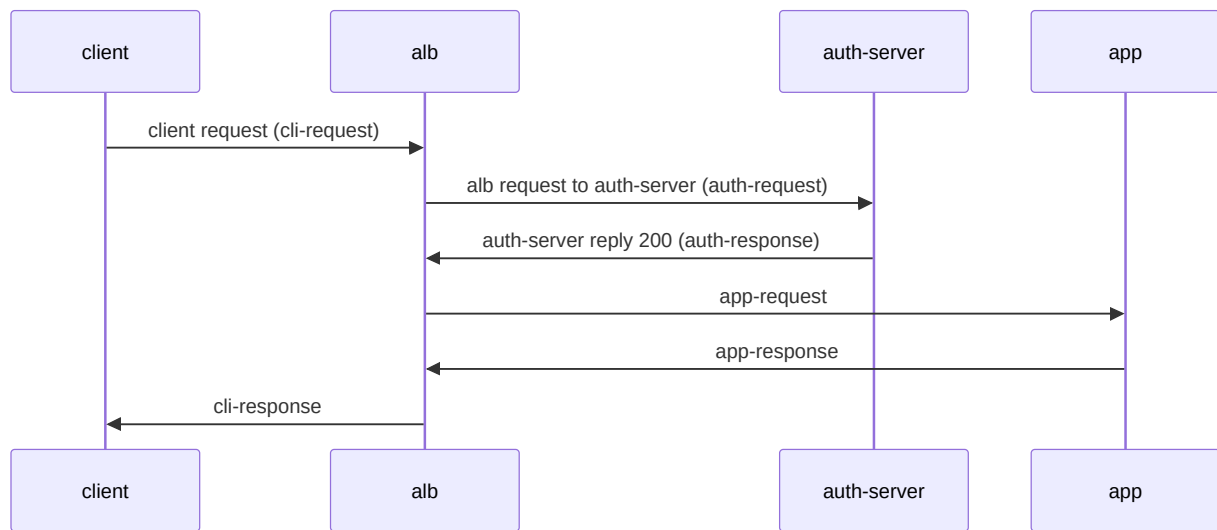
[Ingress-nginx](#) 定义了一系列注解用于配置身份验证过程的细节。以下是 ALB 支持的注解列表，其中“v”表示支持，“x”表示不支持。

	support	type	note
forward-auth			通过发送 http 请求实现的 forward auth
nginx.ingress.kubernetes.io/auth-url	v	string	
nginx.ingress.kubernetes.io/auth-method	v	string	
nginx.ingress.kubernetes.io/auth-signin	v	string	
nginx.ingress.kubernetes.io/auth-signin-redirect-param	v	string	
nginx.ingress.kubernetes.io/auth-response-headers	v	string	
nginx.ingress.kubernetes.io/auth-proxy-set-headers	v	string	
nginx.ingress.kubernetes.io/auth-request-redirect	v	string	

	support	type	note
nginx.ingress.kubernetes.io/auth-always-set-cookie	v	boolean	
nginx.ingress.kubernetes.io/auth-snippet	x	string	
basic-auth			基于用户名和密码的身份验证
nginx.ingress.kubernetes.io/auth-realm	v	string	
nginx.ingress.kubernetes.io/auth-secret	v	string	
nginx.ingress.kubernetes.io/auth-secret-type	v	string	
nginx.ingress.kubernetes.io/auth-type	-	"basic" or "digest"	basic: 支持 apr1 digest: 不支持
auth-cache			
nginx.ingress.kubernetes.io/auth-cache-key	x	string	
nginx.ingress.kubernetes.io/auth-cache-duration	x	string	
auth-keepalive			发送请求时保持连接。通过一系列注解指定 keepalive 行为
nginx.ingress.kubernetes.io/auth-keepalive	x	number	
nginx.ingress.kubernetes.io/auth-keepalive-share-vars	x	"true" or "false"	

	support	type	note
nginx.ingress.kubernetes.io/auth-keepalive-requests	x	number	
nginx.ingress.kubernetes.io/auth-keepalive-timeout	x	number	
auth-tls ↗			当请求为 https 时，额外验证证书。
nginx.ingress.kubernetes.io/auth-tls-secret	x	string	
nginx.ingress.kubernetes.io/auth-tls-verify-depth	x	number	
nginx.ingress.kubernetes.io/auth-tls-verify-client	x	string	
nginx.ingress.kubernetes.io/auth-tls-error-page	x	string	
nginx.ingress.kubernetes.io/auth-tls-pass-certificate-to-upstream	x	"true" or "false"	
nginx.ingress.kubernetes.io/auth-tls-match-cn	x	string	

forward-auth



相关注解：

- [nginx.ingress.kubernetes.io/auth-url](https://kubernetes.github.io/ingress-nginx/user-guide/auth-requests/)
- [nginx.ingress.kubernetes.io/auth-method](https://kubernetes.github.io/ingress-nginx/user-guide/auth-requests/#auth-method)
- [nginx.ingress.kubernetes.io/auth-signin](https://kubernetes.github.io/ingress-nginx/user-guide/auth-requests/#auth-signin)
- [nginx.ingress.kubernetes.io/auth-signin-redirect-param](https://kubernetes.github.io/ingress-nginx/user-guide/auth-requests/#auth-signin-redirect-param)
- [nginx.ingress.kubernetes.io/auth-response-headers](https://kubernetes.github.io/ingress-nginx/user-guide/auth-requests/#auth-response-headers)
- [nginx.ingress.kubernetes.io/auth-proxy-set-headers](https://kubernetes.github.io/ingress-nginx/user-guide/auth-requests/#auth-proxy-set-headers)
- [nginx.ingress.kubernetes.io/auth-request-redirect](https://kubernetes.github.io/ingress-nginx/user-guide/auth-requests/#auth-request-redirect)
- [nginx.ingress.kubernetes.io/auth-always-set-cookie](https://kubernetes.github.io/ingress-nginx/user-guide/auth-requests/#auth-always-set-cookie)

这些注解描述了上述流程中 auth-request、app-request 和 cli-response 的修改。

构造相关注解

auth-url

Auth-request 的 URL，值可以是变量。

auth-method

Auth-request 的请求方法。

auth-proxy-set-headers

值为格式为 `ns/name` 的 ConfigMap 引用。默认情况下，cli-request 的所有头部都会发送给 auth-server。可以通过 `proxy_set_header` 配置额外的头部。默认发送以下头部：

```
X-Original-URI      $request_uri;
X-Scheme            $pass_access_scheme;
X-Original-URL      $scheme://$http_host$request_uri;
X-Original-Method   $request_method;
X-Sent-From         "alb";
X-Real-IP           $remote_addr;
X-Forwarded-For     $proxy_add_x_forwarded_for;
X-Auth-Request-Redirect $request_uri;
```

构造 app-request 相关注解

auth-response-headers

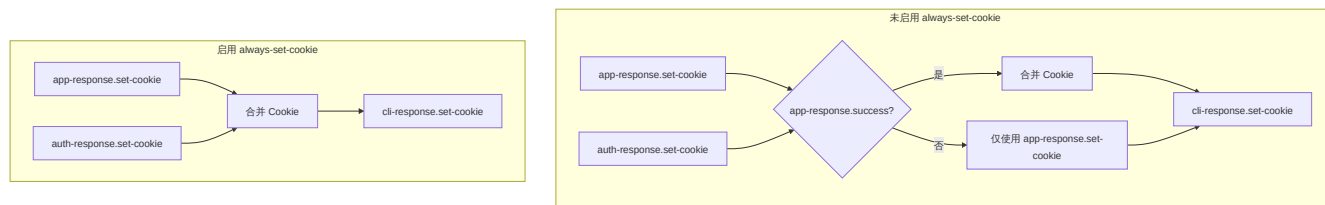
值为逗号分隔的字符串，允许将 auth-response 中的特定头部带到 app-request 中。示例：

```
nginx.ingress.kubernetes.io/auth-response-headers: Remote-User, Remote-Name
```

当 ALB 发起 app-request 时，会携带 auth-response 中的 Remote-User 和 Remote-Name 头部。

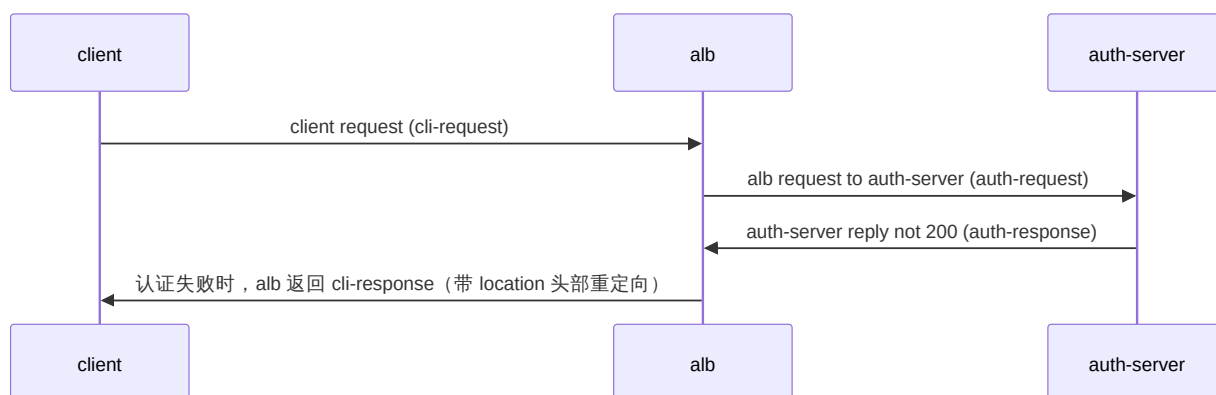
Cookie 处理

auth-response 和 app-response 都可以设置 Cookie。默认情况下，只有当 app-response.success 时，auth-response.set-cookie 才会合并到 cli-response.set-cookie 中。



重定向相关配置

当 auth-server 返回 401 时，可以在 cli-response 中设置重定向头部，指示浏览器跳转到 auth-signin 指定的 URL 进行验证。



auth-signin

值为 URL，指定 cli-response 中的 location 头部。

auth-signin-redirect-param

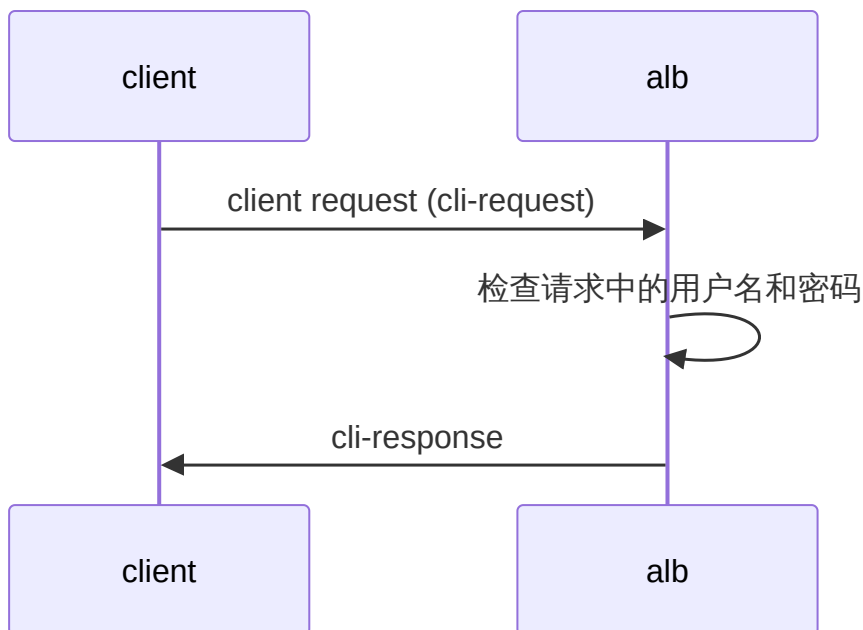
signin-url 中查询参数的名称，默认是 rd。如果 signin-url 不包含指定的 `auth-signin-redirect-param` 参数名，alb 会自动添加该参数。参数值设置为 `$pass_access_scheme://$http_host$escaped_request_uri`，用于记录原始请求 URL。

auth-request-redirect

在 auth-request 中设置 `x-auth-request-redirect` 头部。

basic-auth

basic-auth 是 [RFC 7617](#) 中描述的身份验证过程。交互流程如下：



auth-realm

[受保护区域的描述](#) 即 cli-response 中 `WWW-Authenticate` 头部的 realm 值。WWW-Authenticate: Basic realm="\$realm"

auth-type

身份验证方案的类型，目前仅支持 basic

auth-secret

用户名和密码的 Secret 引用，格式为 ns/name

auth-secret-type

Secret 支持两种类型：

1. auth-file：Secret 的 data 仅包含一个键 "auth"，其值为 Apache htpasswd 格式的字符串。

例如：

```
data:
  auth: "user1:$apr1$xyz..."
```

2. auth-map : Secret 的 data 中每个键代表一个用户名，对应的值是密码哈希（htpasswd 格式中不含用户名）。例如：

```
data:
  user1: "$apr1$xyz...."
  user2: "$apr1$abc...."
```

注意：目前仅支持使用 apr1 算法生成的 htpasswd 格式密码哈希。

CR

ALB CR 已添加与 Auth 相关的配置项，可在 ALB/Frontend/Rule CR 中配置。运行时，ALB 会将 Ingress 上的注解转换为规则。

auth:

基本身份验证配置

basic:

string; 对应 nginx.ingress.kubernetes.io/auth-type: basic

auth_type: 'basic'

string; 对应 nginx.ingress.kubernetes.io/auth-realm

realm: 'Restricted Access'

string; 对应 nginx.ingress.kubernetes.io/auth-secret

secret: 'ns/name'

string; 对应 nginx.ingress.kubernetes.io/auth-secret-type

secret_type: 'auth-map|auth-file'

Forward 身份验证配置

forward:

boolean; 对应 nginx.ingress.kubernetes.io/auth-always-set-cookie

always_set_cookie: true

string; 对应 nginx.ingress.kubernetes.io/auth-proxy-set-headers

auth_headers_cm_ref: 'ns/name'

string; 对应 nginx.ingress.kubernetes.io/auth-request-redirect

auth_request_redirect: '/login'

string; 对应 nginx.ingress.kubernetes.io/auth-method

method: 'GET'

string; 对应 nginx.ingress.kubernetes.io/auth-signin

signin: '/signin'

string; 对应 nginx.ingress.kubernetes.io/auth-signin-redirect-param

signin_redirect_param: 'redirect_to'

[]string; 对应 nginx.ingress.kubernetes.io/auth-response-headers

upstream_headers:

- 'X-User-ID'

- 'X-User-Name'

- 'X-User-Email'

string; 对应 nginx.ingress.kubernetes.io/auth-url

url: 'http://auth-service/validate'

Auth 支持在以下位置配置：

- Alb CR 的 `.spec.config.auth`
- Frontend CR 的 `.spec.config.auth`
- Rule CR 的 `.spec.config.auth`

继承顺序为 Alb > Frontend > Rule。如果子 CR 未配置，则使用父 CR 的配置。

ALB 特殊 Ingress 注解

在处理 Ingress 过程中，ALB 根据注解前缀确定优先级，优先级从高到低为：

- `index.$rule_index-$path_index.alb.ingress.cpaas.io`
- `alb.ingress.cpaas.io`
- `nginx.ingress.kubernetes.io`

这可以处理与 ingress-nginx 的兼容性问题，并在特定 Ingress 路径上指定 Auth 配置。

Auth-Enable

```
alb.ingress.cpaas.io/auth-enable: 'false'
```

ALB 新增的注解，用于指定是否启用该 Ingress 的身份验证功能。

Ingress-Nginx Auth 相关其他特性

Global-Auth

在 ingress-nginx 中，可以通过 ConfigMap 设置全局身份验证，相当于为所有 Ingress 配置身份验证。在 ALB 中，可以在 ALB2 和 FT CR 上配置身份验证，其下的规则会继承这些配置。

No-Auth-Locations

在 ALB 中，可以通过在 Ingress 上配置注解 `alb.ingress.cpaas.io/auth-enable: "false"` 来禁用该 Ingress 的身份验证功能。

注意：与 Ingress-Nginx 不兼容的部分

1. 不支持 auth-keepalive
2. 不支持 auth-snippet

3. 不支持 auth-cache
4. 不支持 auth-tls
5. Basic-auth 仅支持 basic , 不支持 digest
6. Basic-auth basic 仅支持 apr1 算法 , 不支持 bcrypt、sha256 等

故障排查

1. 检查 ALB pod 中 Nginx 容器日志
2. 检查返回中的 `X-ALB-ERR-REASON` 头部

Alauda Container Platform > 网络 > 网络操作器 > ALB Operator > 为 ALB 部署高可用 VIP

为 ALB 部署高可用 VIP

ALB 的高可用需要一个 VIP。获取 VIP 有两种方式。

目录

方法 1：使用 LoadBalancer 类型 Service 提供 VIP

方法 2：使用外部 ALB 设备提供 VIP

方法 1：使用 LoadBalancer 类型 Service 提供 VIP

在 `container` 网络模式下创建 ALB 时，系统会自动创建一个 LoadBalancer Service，为该 ALB 提供 VIP。

在使用此方式之前，请确保集群支持 LoadBalancer Services。你可以使用平台内置实现。有关配置，请参见 [配置 MetalLB](#)。

在 MetalLB 就绪后，你可以向 `alb.spec.config.vip.lbSvcAnnotations` 添加以下注解，以调整 MetalLB 的行为。参见 [ALB 网络配置](#)。

注解	描述
<code>metallb.universe.tf/loadBalancerIPs</code>	以逗号分隔的 VIP 列表，用于分配给 Service。对于多 VIP 或双栈场景，可使用多个条目，例如： <code>192.0.2.10,2001:db8::10</code> 。

注解	描述
<code>metallb.universe.tf/address-pool</code>	用于分配的 MetalLB 地址池。

方法 2：使用外部 ALB 设备提供 VIP

- 在部署前，请与网络工程师确认 ALB 服务的 IP 地址（公网 IP、私网 IP、VIP）或域名。如果你希望使用域名作为外部流量访问 ALB 的地址，需要提前申请域名并配置域名解析。建议使用商用 Load Balancer 设备提供 VIP；如果没有，也可以使用 [纯软件数据中心 LB 方案 \(Alpha\)](#)
- 根据业务场景，外部 ALB 需要为所有正在使用的端口配置健康检查，以减少 ALB 升级期间的停机时间。健康检查配置如下：

健康检查参数	描述
端口	- 对于全局集群，填写：11782。 - 对于业务集群，填写：1936。
协议	健康检查的协议类型，建议使用 TCP。
响应超时	接收健康检查响应所需的时间，建议配置为 2 秒。
检查间隔	健康检查的时间间隔，建议配置为 5 秒。
不健康阈值	后端服务器连续失败多少次后判定健康检查状态为失败，建议配置为 3 次。

[Alauda Container Platform](#) > [网络](#) > [网络操作器](#) > [ALB Operator](#) > 在 ALB 中绑定网卡

在 ALB 中绑定网卡

默认情况下，ALB 监听 ipv4 的 `0.0.0.0` 和 ipv6 的 `::`。在某些安全场景下，需要绑定到特定的网络接口卡（NIC）。

目录

[对于集群内嵌 ALB](#)

[对于用户自定义 ALB](#)

对于集群内嵌 ALB

默认情况下，每个集群都会部署一个内嵌 ALB。在 global 集群中，名称应为 'global-alb2'，而在其他集群中，应为 'cpaas-system'。

将 `$CLUSTER` 和 `$NIC` 替换为实际的集群和网卡。如果使用 Alive（Alauda Container Platform 虚拟 IP 管理），需要在 nic 列表中添加 `alive`。

```
kubectl annotate cluster -n cpaas-system $cluster cpaas.io/alb-bind-nic='{"nic":["$NIC","alive"]}'
```

默认情况下，ALB 在单栈集群中启用 IPv6。但使用 bindnic 时，指定的 NIC 可能没有 IPv6 地址。在这种情况下，ALB 仍会尝试绑定到 `::*`。作为解决方案，可以禁用 IPv6。

```
kubectl annotate cluster -n cpaas-system $cluster cpaas.io/alb-enable-ipv6='false'
```

对于用户自定义 **ALB**

```
kubectl patch alb2 -n cpaas-system $ALB -p '{"spec":{"config":{"enableIPv6":"false","bindNIC":{"\nic\":[\"$NIC\",\"alive\"]}}}}' --type=merge
```

ALB 性能选择的决策依据

针对平台提出的小型、中型、大型及自定义生产环境规格，以及实例和端口的资源分配方式，以下建议可供部署参考。

目录

[小型生产环境](#)

[中型生产环境](#)

[大型生产环境](#)

[特殊场景部署建议](#)

[负载均衡器使用模式选择](#)

小型生产环境

对于较小的业务规模，比如集群节点不超过 5 个，仅用于运行标准应用，单个负载均衡器即可满足需求。建议以高可用模式运行，至少部署 2 个副本，以保证环境的稳定性。

可以通过端口隔离方式对负载均衡器进行隔离，实现多个项目共享。

该规格在实验室环境下测得的峰值 QPS 约为每秒 300 次请求。

Create Load Balancer

* Name:

loadbalancer

Display Name:

* Specification:

Small scale
Cluster less than 5 nodes

Medium scale
Cluster less than 30 nodes

Large scale
Cluster more than 30 nodes

Custom
For professional use

Resource Limit:

CPU

200

m

Memory

256

Mi

Type:

Standalone

High availability

* Access URL:

192.168.1.10

* Replicas:

2

* Node Labels:

kubernetes.io/arch:arm64

3 nodes meet the conditions

Allocated By:

Instance

Port

中型生产环境

当业务规模达到一定程度，比如集群节点不超过 30 个，且需要处理高并发业务同时运行标准应用时，单个负载均衡器仍然足够。建议采用高可用模式，至少部署 3 个副本，以维持环境稳定。

可以采用端口隔离或实例分配方式，让多个项目共享负载均衡器。当然，也可以为核心项目专门创建新的负载均衡器。

该规格在实验室环境下测得的峰值 QPS 约为每秒 10,000 次请求。

Create Load Balancer

* Name:

loadbalancer

Display Name:

* Specification:

Small scale

Cluster less than 5 nodes

Medium scale

Cluster less than 30 nodes

Large scale

Cluster more than 30 nodes

Custom

For professional use

?

Resource Limit:

CPU

2

Core

Memory

1

Gi

Type:

Standalone

High availability

* Access URL:

192.168.1.20

* Replicas:

-

3

+

* Node Labels:

kubernetes.io/arch:arm64

x

3 nodes meet the conditions

Allocated By:

Instance

Port

?

大型生产环境

对于更大规模的业务，比如集群节点超过 30 个，且需要处理高并发业务及长连接数据，建议使用多个负载均衡器，每个均采用高可用类型，至少部署 3 个副本，以确保环境稳定。

可以通过端口隔离或实例分配方式实现多个项目共享负载均衡器，也可以为核心项目专门创建新的负载均衡器。

该规格在实验室环境下测得的峰值 QPS 约为每秒 20,000 次请求。

Create Load Balancer

* Name :

loadbalancer

Display Name :

* Specification :

Small scale

Cluster less than 5 nodes

Medium scale

Cluster less than 30 nodes

Large scale

Cluster more than 30 nodes

Custom

For professional use

?

Resource Limit :

CPU

4

Core

Memory

2

Gi

Type :

Standalone

High availability

* Access URL :

192.168.1.30

* Replicas :

-

3

+

* Node Labels :

kubernetes.io/arch:arm64

x

3 nodes meet the conditions

Allocated By :

Instance

Port

?

特殊场景部署建议

场景	部署建议
功能测试	建议部署单实例负载均衡器。
测试环境	如果测试环境符合上述小型或中型定义，使用单点负载均衡器即可。负载均衡器实例可被多个项目共享。
核心应用	建议为核心应用专门使用独立的负载均衡器。
大规模数据传输	由于负载均衡器本身的内存消耗较小，即使是大型规格，保留 2Gi 内存也足够。但如果业务涉及大规模数据传输，导致内存消耗显著，应相应增加负载均衡器的内存分配。 建议在自定义规格场景中逐步扩展负载均衡器内存，密切监控内存使用情况，最终确定合理的内存大小以保证使用率合理。

负载均衡器使用模式选择

使用模式	优势	劣势
(推荐) 将负载均衡器作为实例资源分配给单个项目	• 管理相对简单。 • 每个项目拥有独立负载均衡器，确保规则隔离和资源分离，互不干扰。	在主机网络模式下，集群必须拥有较多可用节点供负载均衡器使用，导致资源消耗较高。
将负载均衡器作为实例资源分配给多个项目	管理相对简单。	由于所有被分配项目对负载均衡器实例拥有完全权限，当某项目配置负载均衡器端口和规则时，可能出现以下情况： • 该项目配置的规则可能影响其他项目。 • 负载均衡器配置误操作可能更改其他项目设置。 • 某业务的流量请求可能影响负载均衡器实例的整体可用性。
通过端口动态分配负载均衡器资源，不同项目使用不同端口	项目间规则隔离，确保互不干扰。	• 管理复杂度增加。平台管理员需主动规划和分配项目端口，并配置外部服务映射。 • 基于端口的分配成熟度较低，目前使用客户较少，功能需进一步完善。 • 资源冲突风险。所有使用同一负载均衡器的服务可能面临单个服务影响整个负载均衡器的场景。

ALB 中的负载均衡会话亲和策略

本指南介绍了 ALB 中可用的各种负载均衡算法，以及如何配置它们以优化应用原生应用 Pod 之间的流量分布。

目录

概述

可用算法

- 轮询（默认）

- 源 IP 哈希

- 基于 Cookie 的亲

- 基于 Header 的亲

- EWMA（指数加权移动平均）

配置方式

1. 使用 Ingress 注解

2. 使用 ALB Frontend/Rule 自定义资源

最佳实践

概述

ALB 支持多种负载均衡算法，用于将传入流量分配到后端 Pod。算法的选择取决于您的应用需求，例如会话持久性、性能优化或负载均衡分布。

可用算法

轮询（默认）

- 策略：`rr`
- 描述：按顺序循环将请求分发到所有可用的 Pod。
- 适用场景：适用于无状态原生应用，每个请求都可以由任意 Pod 处理。

源 IP 哈希

- 策略：`sip-hash`
- 描述：始终将来自相同源 IP 地址的请求路由到同一个 Pod。
- 行为：如果存在 X-Forwarded-For 头，则使用其中第一个 IP，否则使用客户端源 IP。
- 适用场景：需要基于客户端 IP 的会话持久性时使用。

基于 Cookie 的亲和

- 策略：`cookie`
- 属性：`cookie-name`（默认为 `JSESSIONID`）
- 描述：将具有相同 cookie 值的请求路由到同一个 Pod。
- 行为：
 - 如果指定的 cookie 不存在，ALB 会在响应中添加该 cookie
 - Cookie 格式：`timestamp.worker_pid.random_number`
- 适用场景：适合需要基于 cookie 实现会话粘性的应用。

基于 Header 的亲和

- 策略：`header`
- 属性：`header-name`
- 描述：将具有相同 header 值的请求路由到同一个 Pod。

- 适用场景: 适合需要基于特定 HTTP 头进行路由的应用。

EWMA (指数加权移动平均)

- 策略: `ewma`
- 描述: 基于 Pod 响应时间, 使用“二选一”算法 (Power of Two Choices, P2C) 结合 EWMA 进行流量路由。
- 行为:
 - 为每个 Pod 维护基于响应时间的 EWMA 分数
 - 将流量路由到 EWMA 分数较低的 Pod
 - 分数随时间呈指数衰减
- 适用场景: 适用于对延迟敏感的应用
- 参考: [Twitter Finagle EWMA Documentation](#) ↗

配置方式

1. 使用 Ingress 注解

```
annotations:  
  alb.ingress.cpaas.io/session-affinity-policy: '<algorithm>' # rr | sip-  
hash | cookie | header | ewma  
  alb.ingress.cpaas.io/session-affinity-attribute: '<attribute>' # cookie  
和 header 策略必填
```

2. 使用 ALB Frontend/Rule 自定义资源

```
spec:  
  serviceGroup:  
    session_affinity_policy: '<algorithm>' # rr | sip-hash | cookie | hea  
der | ewma  
    session_affinity_attribute: '<attribute>' # cookie 和 header 策略必填
```

最佳实践

- 对于请求模式均匀的无状态原生应用，选择轮询算法
- 需要基于客户端 IP 实现会话持久性时，使用源 IP 哈希
- 对于需要会话粘性的 Web 原生应用，采用基于 Cookie 的亲和策略
- 对于响应时间波动较大的服务，考虑使用 EWMA 以优化延迟

[Alauda Container Platform](#) > [网络](#) > [网络操作器](#) > [ALB Operator](#) > [L4/L7 超时](#)

L4/L7 超时

目录

| [基本概念](#)

CRD

超时的含义

Ingress 注解

端口级超时

基本概念

L4/L7 超时是 ALB 提供的一个功能，用于配置 L4/L7 代理的超时时间。

超时通过 Lua 脚本实现，修改时 不需要重新加载 Nginx。

CRD


```
timeout:
  properties:
    proxy_connect_timeout_ms:
      type: integer
    proxy_read_timeout_ms:
      type: integer
    proxy_send_timeout_ms:
      type: integer
  type: object
```

配置可以设置在：

- Frontend: `.spec.config.timeout`
- Rule: `.spec.config.timeout`

超时的含义

超时分为三种类型：

1. **proxy_connect_timeout_ms**：定义与上游服务器建立连接的超时时间。如果在此时间内无法建立连接，请求将失败。
2. **proxy_read_timeout_ms**：定义从上游服务器读取响应的超时时间。该超时是两次连续读取操作之间的时间间隔，而非整个响应的总时间。如果在此时间内未接收到数据，连接将被关闭。
3. **proxy_send_timeout_ms**：定义向上游服务器发送请求的超时时间。与读取超时类似，该超时是两次连续写入操作之间的时间间隔。如果在此时间内无法发送数据，连接将被关闭。

Ingress 注解

注解	说明
<code>nginx.ingress.kubernetes.io/proxy-connect-timeout</code>	对应 CR 中的 <code>proxy_connect_timeout_ms</code>
<code>nginx.ingress.kubernetes.io/proxy-read-timeout</code>	对应 CR 中的 <code>proxy_read_timeout_ms</code>

注解	说明
nginx.ingress.kubernetes.io/proxy-send-timeout	对应 CR 中的 proxy_send_timeout_ms

端口级超时

可以直接在端口上配置超时，用作 L4 超时。

HTTP 重定向

目录

基本概念

CRD

Ingress 注解

SSL-Redirect

端口级重定向

规则级重定向

基本概念

HTTP 重定向是 ALB 提供的一项功能。它会直接返回一个 30x 的 HTTP 状态码给匹配规则的请求。Location 头部将用于指示客户端重定向到新的 URL。

ALB 支持在端口和规则级别配置重定向。

CRD

```
redirect:
  properties:
    code:
      type: integer
    host:
      type: string
    port:
      type: integer
    prefix_match:
      type: string
    replace_prefix:
      type: string
    scheme:
      type: string
    url:
      type: string
  type: object
```

重定向可以配置在：

- 前端：`.spec.config.redirect`
- 规则：`.spec.config.redirect`

Ingress 注解

注解	说明
<code>nginx.ingress.kubernetes.io/permanent-redirect</code>	对应 CR 中的 URL，默认会将 code 设置为 301
<code>nginx.ingress.kubernetes.io/permanent-redirect-code</code>	对应 CR 中的 code
<code>nginx.ingress.kubernetes.io/temporal-redirect</code>	对应 CR 中的 URL，默认会将 code 设置为 302
<code>nginx.ingress.kubernetes.io/temporal-redirect-code</code>	对应 CR 中的 code

注解	说明
nginx.ingress.kubernetes.io/ssl-redirect	对应 CR 中的 scheme，默认会将 scheme 设置为 HTTPS
nginx.ingress.kubernetes.io/force-ssl-redirect	对应 CR 中的 scheme，默认会将 scheme 设置为 HTTPS

SSL-Redirect

1. SSL-redirect 和 force-ssl-redirect 的区别在于，SSL-redirect 只有在 ingress 对应域名有证书时才生效，而 force-ssl-redirect 无论是否有证书都会生效。
2. 对于 HTTPS 端口，如果只配置了 SSL-redirect，则不会设置重定向。

端口级重定向

当在端口级别配置重定向时，*所有*访问该端口的请求都会根据重定向配置进行重定向。

规则级重定向

当在规则级别配置重定向时，*匹配*该规则请求将根据重定向配置进行重定向。

[Alauda Container Platform](#) > [网络](#) > [网络操作器](#) > [ALB Operator](#) > [CORS](#)

CORS

目录

[基本概念](#)

[CRD](#)

基本概念

[CORS](#) （跨域资源共享）是一种机制，允许网页上的资源（例如字体、JavaScript 等）从其源域之外的其他域请求。

CRD

enableCORS:

description: enableCORS 是是否启用跨域的开关,
当 EnableCORS 为 false 时, alb2 会将信息传递给后端服务器,
由后端服务器决定是否允许跨域

type: boolean

corsAllowHeaders:

description: corsAllowHeaders 定义 cors 允许的请求头,
当 enableCORS 为 true 时, 多个请求头用逗号分隔

type: string

corsAllowOrigin:

description: corsAllowOrigin 定义 cors 允许的源,
当 enableCORS 为 true 时, 多个源用逗号分隔

type: string

可在规则的 `.spec` 上进行配置。

Header Modification

目录

基本概念

使用注解

示例

基本概念

当接收到请求时，header 修改允许在转发到后端之前调整请求头。同样地，当接收到响应时，允许在返回给客户端之前调整响应头。

使用注解

目标	注解键
ingress	<code>alb.ingress.cpaas.io/rewrite-request</code> , <code>alb.ingress.cpaas.io/rewrite-response</code>
rule	<code>alb.rule.cpaas.io/rewrite-request</code> , <code>alb.rule.cpaas.io/rewrite-response</code>

注解值是包含配置的 JSON 字符串。


```

type RewriteRequestConfig struct {
    Headers      map[string]string `json:"headers,omitempty"`
    // 设置 header
    HeadersVar    map[string]string `json:"headers_var,omitempty"`
    // 设置 header, 值为变量名
    HeadersRemove []string          `json:"headers_remove,omitempty"`
    // 移除 header
    HeadersAdd     map[string][]string `json:"headers_add,omitempty"`
    // 添加 header, 值可以是多个
    HeadersAddVar  map[string][]string `json:"headers_add_var,omitempty"`
    // 添加 header, 值可以是多个且为变量名
}

type RewriteResponseConfig struct {
    Headers      map[string]string `json:"headers,omitempty"`
    // 设置 header
    HeadersRemove []string          `json:"headers_remove,omitempty"`
    // 移除 header
    HeadersAdd     map[string][]string `json:"headers_add,omitempty"`
    // 添加 header, 值可以是多个
}

```

注意：在 `*_var` 映射中，键是 header 名称，值是 ALB 上下文变量名。例如，向一个 Ingress 添加如下注解：

```

alb.ingress.cpaas.io/rewrite-request: '{
    "headers_var": {
        "x-my-host": "http_host"
    }
}'

```

将会添加键为 `x-my-host`，值为请求 host header 的请求头。你可以参考 [nginx variable](#) 获取变量名。

ALB 提供了额外的变量：

变量名	描述
<code>first_forward_or_remote_addr</code>	第一个转发地址或远程地址，默认是 <code>remote_addr</code>

变量名	描述
<code>first_forward</code>	第一个转发地址，默认是空字符串

示例

要从 cookie 中添加 Authorization 头，可以使用：

```
alb.ingress.cpaas.io/rewrite-request: '{"headers_var":{"Authorization":"cookie_auth_token"}}'
```

要设置 HSTS，可以使用：

```
alb.rule.cpaas.io/rewrite-response: |
  { "headers": { "Strict-Transport-Security": "max-age=63072000; includeSubDomains; preload"} }
```

[Alauda Container Platform](#) > [网络](#) > [网络操作器](#) > [ALB Operator](#) > URL 重写

URL 重写

目录

| [基本概念](#)

[配置](#)

基本概念

ALB 可以在将请求转发到后端之前重写请求的 URL。您可以使用正则表达式捕获组来重写 URL。

配置

通过 ingress 注解

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  nginx.ingress.kubernetes.io/rewrite-target: /$2
  name: demo
spec:
  ingressClassName: alb
  rules:
    - http:
        paths:
          - backend:
              service:
                name: frontend
                port:
                  number: 8080
              path: /(prefix-x)(/|$)(.*)
              pathType: ImplementationSpecific

```

通过 rule

```

apiVersion: crd.alauda.io/v1
kind: Rule
metadata:
  labels:
    alb2.cpaas.io/frontend: alb-00080
    alb2.cpaas.io/name: alb
  name: demo
  namespace: cpaas-system
spec:
  dslx:
    - type: URL
      values:
        - - REGEX
          - ^/(prefix-x)(/|$)(.*)
      rewrite_base: /(prefix-x)(/|$)(.*)
      rewrite_target: /$3

```

示例：客户端请求 `/prefix-x/abc`；后端接收到 `/abc`。

ModSecurity

ModSecurity 是一个开源的 Web 应用防火墙（WAF），旨在保护 Web 应用免受恶意攻击。它由开源社区维护，支持多种编程语言和 Web 服务器。平台负载均衡器（ALB）支持配置 ModSecurity，允许在 Ingress 级别进行个性化配置。

目录

术语

操作步骤

方法一：添加注解

方法二：配置 CR

相关说明

覆盖规则

配置示例

术语

术语	说明
owasp-core-rules	OWASP Core Rule Set 是一个开源规则集，用于检测和防止常见的 Web 应用攻击。

操作步骤

通过向对应资源的 YAML 文件添加注解或配置 CR 来配置 ModSecurity。

方法一：添加注解

在对应 YAML 文件的 metadata.annotations 字段中添加以下注解以配置 ModSecurity。

- Ingress-Nginx 兼容注解

注解	类型	适用对象	说明
nginx.ingress.kubernetes.io/enable-modsecurity	bool	Ingress	启用 ModSecurity
nginx.ingress.kubernetes.io/enable-owasp-core-rules	bool	Ingress	启用 OWA Core Rule Set。
nginx.ingress.kubernetes.io/modsecurity-transaction-id	string	Ingress	用于标识每个请求的唯一事务 ID，便于记录和调试。
nginx.ingress.kubernetes.io/modsecurity-snippet	string	Ingress, ALB, FT, Rule	允许用户自定义 ModSecurity 配置，以满足特定的安全需求。

- ALB 特殊注解

注解	类型	适用对象	说明
alb.modsecurity.cpaas.io/use-recommend	bool	Ingress	启用或禁用推荐的 ModSecurity 规则；该 <code>true</code> 时应用预定义规则集。
alb.modsecurity.cpaas.io/cmref	string	Ingress	引用特定配置，例如定义 ConfigMap 的引用（ <code>\$ns/\$name#\$section</code> ）加载自定义安全配置。

方法二：配置 CR

1. 打开需要配置的 ALB、FT 或 Rule 配置文件。
2. 根据需要在 spec.config 下添加以下字段。

```
{ 'modsecurity': {
  'enable': true, # 启用或禁用 ModSecurity
  'transactionId': '$xx', # 使用来自 Nginx 的 ID
  'useCoreRules': true, # 添加 modsecurity_rules_file /etc/nginx/owasp-modsecurity-crs/nginx-modsecurity.conf
  'useRecommend': true, # 添加 modsecurity_rules_file /etc/nginx/modsecurity/modsecurity.conf
  'cmRef': '$ns/$name#$section', # 从 ConfigMap 添加配置
} }
```

3. 保存并应用配置文件。

相关说明

覆盖规则

如果 Rule 中未配置 ModSecurity，则会尝试从 FT 中查找配置；如果 FT 中也无配置，则使用 ALB 中的配置。

配置示例

以下示例部署了一个名为 `waf-alb` 的 ALB 和一个名为 `hello` 的演示后端应用。同时部署了一个名为 `ing-waf-enable` 的 Ingress，定义了 `/waf-enable` 路由并配置了 ModSecurity 规则。任何包含查询参数 `test` 且其值中包含字符串 `test` 的请求都会被阻止。


```
cat <<EOF | kubectl apply -f -
apiVersion: crd.alauda.io/v2
kind: ALB2
metadata:
  name: waf-alb
  namespace: cpaas-system
spec:
  config:
    loadbalancerName: waf-alb
    projects:
      - ALL_ALL
    replicas: 1
    type: nginx
  ---
apiVersion: networking.k8s.io/v1
```

OTel

OpenTelemetry (OTel) 是一个开源项目，旨在为分布式系统（例如微服务架构）中的 telemetry 数据采集、处理和导出提供一个供应商无关的标准。它可以帮助开发者更轻松地分析软件的性能和行为，从而便于诊断和解决应用问题。

目录

术语

前提条件

操作步骤

更新 ALB 配置

相关操作

在 Ingress 中配置 OTel

在应用中使用 OTel

继承

补充说明

Sampling Strategies

Attributes

配置示例

术语

术语	说明
Trace	提交到 OTel Server 的数据，是一组相关事件或操作的集合，用于跟踪分布式系统中请求的流转；每个 Trace 由多个 Span 组成。
Span	Trace 中的一个独立操作或事件，包含开始时间、持续时间以及其他相关信息。
OTel Server	能够接收和存储 Trace 数据的 OTel 服务器，例如 Jaeger、Prometheus 等。
Jaeger	一个开源的分布式 tracing 系统，用于监控和排查微服务架构，支持与 OpenTelemetry 集成。
Attributes	附加到 Trace 或 Span 上的键值对，用于提供额外的上下文信息。包括 Resource Attributes 和 Span Attributes；更多信息请参见 Attributes 。
Sampler	决定是否对 Trace 进行采样并上报的策略组件。可以配置不同的采样策略，例如全量采样、比例采样等。
ALB (Another Load Balancer)	一种在集群内可用节点之间分发网络请求的软件或硬件设备；平台中使用的负载均衡器（ALB）是一个七层软件负载均衡器，可以配置为使用 OTel 监控流量。ALB 支持将 Trace 提交到指定的 Collector，并允许使用不同的采样策略；同时也支持配置是否在 Ingress 层提交 Trace。
FT (Frontend)	ALB 的端口配置，用于指定端口级配置。
Rule	端口（FT）上的路由规则，用于匹配特定路由。
HotROD (Rides on Demand)	Jaeger 提供的一个示例应用，用于演示 distributed tracing 的使用；更多详情请参见 Hot R.O.D. - Rides on Demand 。
hotrod-with-proxy	通过环境变量指定 HotROD 内部微服务的地址；更多详情请参见 hotrod-with-proxy 。

前提条件

- 确保存在可用的 **ALB**：创建或使用现有的 ALB。以下示例中使用 `<otel-alb>` 作为 ALB 名称。有关创建 ALB 的说明，请参见 [配置 ALB](#)。
- 确保存在 **OTel** 数据上报服务器地址：此地址后文将称为 `<jaeger-server>`。

操作步骤

更新 ALB 配置

1. 在集群的主节点上，使用 CLI 工具执行以下命令以编辑 ALB 配置。

```
kubectl edit alb2 -n cpaas-system <otel-alb> # Replace <otel-alb> with the actual ALB name
```

2. 在 `spec.config` 部分下添加以下字段。

```
otel:
  enable: true
  exporter:
    collector:
      address: '<jaeger-server>' # Replace <jaeger-server> with the actual OTel data reporting server address
      request_timeout: 1000
```

完成后的示例配置：

```
spec:
  address: 192.168.1.1
  config:
    otel:
      enable: true
      exporter:
        collector:
          address: "http://jaeger.default.svc.cluster.local:4318"
          request_timeout: 1000
    antiAffinityKey: system
    defaultSSLCert: cpaas-system/cpaas-system
    defaultSSLStrategy: Both
    gateway:
      ...
  type: nginx
```

3. 执行以下命令保存更新。更新后，ALB 默认启用 OpenTelemetry，且所有请求的 Trace 信息都会上报到 Jaeger Server。

```
:wq
```

相关操作

在 Ingress 中配置 OTel

- 在 Ingress 上启用或禁用 OTel

通过配置是否在 Ingress 上启用 OTel，可以更好地监控和调试应用的请求流，通过跟踪请求在不同服务之间的传播来识别性能瓶颈或错误。

操作步骤

在 Ingress 的 metadata.annotations 字段下添加以下配置：

```
nginx.ingress.kubernetes.io/enable-opentelemetry: "true"
```

参数说明：

- **nginx.ingress.kubernetes.io/enable-opentelemetry**：设置为 `true` 时，表示 Ingress controller 在处理经过此 Ingress 的请求时启用 OpenTelemetry 功能，即会采集并上报请求的 Trace 信息。设置为 `false` 或删除该注解时，表示不会采集或上报请求的 Trace 信息。
- 在 Ingress 上启用或禁用 OTel Trust

OTel Trust 决定 Ingress 是否信任并使用入站请求中的 Trace 信息（例如 trace ID）。

操作步骤

在 Ingress 的 metadata.annotations 字段下添加以下配置：

```
nginx.ingress.kubernetes.io/opentelemetry-trust-incoming-span: "true"
```

参数说明：

- **nginx.ingress.kubernetes.io/opentelemetry-trust-incoming-span**：设置为 `true` 时，Ingress 会继续使用已存在的 Trace 信息，有助于保持跨服务 tracing 的一致性，使

整个请求链路能够在分布式 tracing 系统中被完整追踪和分析。设置为 `false` 时，将为请求生成新的 tracing 信息，这可能会导致请求在进入 Ingress 后被视为新的 tracing 链路的一部分，从而中断跨服务 trace 的连续性。

- 在 Ingress 上添加不同的 OTel 配置

此配置允许您针对不同的 Ingress 资源自定义 OTel 的行为和数据导出方式，从而对每个服务的 tracing 策略或目标进行精细化控制。

操作步骤

在 Ingress 的 `metadata.annotations` 字段下添加以下配置：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    alb.ingress.cpaas.io/otel: >
      {
        "enable": true,
        "exporter": {
          "collector": {
            "address": "<jaeger-server>", # Replace <jaeger-server
            > with the actual OTel data reporting server address, e.g., "address":
            "http://128.0.0.1:4318"
            "request_timeout": 1000
          }
        }
      }
    }
```

参数说明：

- **exporter**：指定如何将采集到的 Trace 数据发送到 OTel Collector（OTel 数据上报服务器）。
- **address**：指定 OTel Collector 的地址。
- **request_timeout**：指定请求超时时间。

在应用中使用 OTel

以下配置展示了完整的 OTel 配置结构，可用于定义如何在应用中启用和使用 OTel 功能。

在集群主节点上，使用 CLI 工具执行以下命令以获取完整的 OTel 配置结构。

```
kubectl get crd alaudaloadbalancer2.crd.alauda.io -o json|jq ".spec.versions[2].schema.openAPIV3Schema.properties.spec.properties.config.properties.otel"
```

回显结果：

```
{
  "otel": {
    "enable": true
  },
  "exporter": {
    "collector": {
      "address": ""
    },
  },
  "flags": {
    "hide_upstream_attrs": false
    "notrust_incoming_span": false
    "report_http_request_header": false
    "report_http_response_header": false
  },
  "sampler": {
    "name": "",
    "options": {
      "fraction": ""
      "parent_name": ""
    },
  },
},
}
```

参数说明：

参数	说明
otel.enable	是否启用 OTel 功能。
exporter.collector.address	OTel 数据上报服务器的地址，支持 http/https 协议和域名。
flags.hide_upstream_attrs	是否上报上游规则的信息。

参数	说明
<code>flag.notrust_incoming_span</code>	是否信任并使用入站请求中的 OTel Trace 信息（例如 trace ID）。
<code>flags.report_http_request_header</code>	是否上报请求头。
<code>flags.report_http_response_header</code>	是否上报响应头。
<code>sampler.name</code>	采样策略名称；详情请参见 Sampling Strategies 。
<code>sampler.options.fraction</code>	采样率。
<code>sampler.options.parent_name</code>	<code>parent_base</code> 采样策略的父策略。

继承

默认情况下，如果 ALB 配置了某些 OTel 参数而 FT 未配置，则 FT 会继承 ALB 的参数作为自身配置；也就是说，FT 继承 ALB 的配置，而 Rule 可以同时继承 ALB 和 FT 的配置。

- **ALB**：ALB 上的配置通常是全局和默认配置。您可以在这里配置诸如 Collector 地址之类的全局参数，这些参数会被下层的 FT 和 Rule 继承。
- **FT**：FT 可以继承 ALB 的配置，这意味着 FT 上未配置的某些 OTel 参数将使用 ALB 中的配置。不过，FT 也可以进一步细化；例如，您可以选择仅在 FT 上有选择地启用或禁用 OTel，而不影响其他 FT 或 ALB 的全局设置。
- **Rule**：Rule 可以继承 ALB 和 FT 的配置。不过，Rule 也可以进一步细化；例如，某个特定 Rule 可以选择不信任传入的 OTel Trace 信息，或者调整采样策略。

操作步骤

通过在 ALB、FT 和 Rule 的 YAML 文件中配置 `spec.config.otel` 字段，您可以添加与 OTel 相关的配置。

补充说明

Sampling Strategies

参数	说明
always on	始终上报所有 tracing 数据。
always off	永不上报 tracing 数据。
traceid-ratio	根据 <code>traceid</code> 决定是否上报。 <code>traceparent</code> 的格式为 <code>xx-traceid-xx-flag</code> ，其中 <code>traceid</code> 的前 16 个字符表示一个 32 位十六进制整数。如果该整数小于 <code>fraction</code> 乘以 4294967295（即 $2^{32}-1$ ），则会上报。
parent-base	根据请求中 <code>traceparent</code> 的 <code>flag</code> 部分决定是否上报。当 <code>flag</code> 为 01 时会上报；例如： <code>curl -v "http://\$ALB_IP/" -H 'traceparent: 00-xx-xx-01'</code> ；当 <code>flag</code> 为 02 时则不上报；例如： <code>curl -v "http://\$ALB_IP/" -H 'traceparent: 00-xx-xx-02'</code> 。

Attributes

- Resource Attributes**

这些属性默认会上报。

参数	说明
hostname	ALB Pod 的主机名
service.name	ALB 的名称
service.namespace	ALB 所在的命名空间
service.type	默认为 ALB
service.instance.id	ALB Pod 的名称

- Span Attributes**

- 默认上报的属性：

参数	说明
http.status_code	状态码
http.request.resend_count	重试次数
alb.rule.rule_name	此请求匹配到的规则名称
alb.rule.source_type	此请求匹配到的规则类型，当前仅为 Ingress
alb.rule.source_name	Ingress 的名称
alb.rule.source_ns	Ingress 所在的命名空间

- 默认上报，但可通过修改 `flag.hide_upstream_attrs` 字段排除的属性：

参数	说明
alb.upstream.svc_name	流量转发到的 Service（内部路由）名称
alb.upstream.svc_ns	被转发到的 Service（内部路由）所在的命名空间
alb.upstream.peer	转发到的 Pod 的 IP 地址和端口

- 默认不报告，但可通过修改 `flag.report_http_request_header` 字段进行上报的属性：

参数	说明
<code>**http.request.header.<header>**</code>	请求头

- 默认不报告，但可通过修改 `flag.report_http_response_header` 字段进行上报的属性：

参数	说明
<code>**http.response.header.<header>**</code>	响应头

配置示例

以下 YAML 配置会部署一个 ALB，并使用 Jaeger 作为 OTel server，使用 Hotrod-proxy 作为演示后端。通过配置 Ingress 规则，当客户端请求 ALB 时，流量将被转发到 HotROD。此外，HotROD 内部微服务之间的通信也会通过 ALB 进行路由。

1. 将以下 YAML 保存为名为 all.yaml 的文件。


```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: hotrod
spec:
  replicas: 1
  selector:
    matchLabels:
      service.cpaas.io/name: hotrod
      service_name: hotrod
  template:
    metadata:
      labels:
        service.cpaas.io/name: hotrod
        service_name: hotrod
    spec:
      containers:
        - name: hotrod
          env:
            - name: PROXY_PORT
              value: '80'
            - name: PROXY_ADDR
              value: 'otel-alb.default.svc.cluster.local:'
            - name: OTEL_EXPORTER_OTLP_ENDPOINT
              value: 'http://jaeger.default.svc.cluster.local:4318'
          image: theseedaaa/hotrod-with-proxy:latest
          imagePullPolicy: IfNotPresent
          command: ['/bin/hotrod', 'all', '-v']
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hotrod-frontend
spec:
  ingressClassName: otel-alb
  rules:
    - http:
        paths:
          - backend:
              service:
                name: hotrod
                port:
                  number: 8080

```

```

        path: /dispatch
        pathType: ImplementationSpecific
      - backend:
          service:
            name: hotrod
            port:
              number: 8080
          path: /frontend
          pathType: ImplementationSpecific
    ---
  apiVersion: networking.k8s.io/v1
  kind: Ingress
  metadata:
    name: hotrod-customer
  spec:
    ingressClassName: otel-alb
    rules:
      - http:
          paths:
            - backend:
                service:
                  name: hotrod
                  port:
                    number: 8081
                path: /customer
                pathType: ImplementationSpecific
    ---
  apiVersion: networking.k8s.io/v1
  kind: Ingress
  metadata:
    name: hotrod-route
  spec:
    ingressClassName: otel-alb
    rules:
      - http:
          paths:
            - backend:
                service:
                  name: hotrod
                  port:
                    number: 8083
                path: /route
                pathType: ImplementationSpecific
    ---

```

```
apiVersion: v1
kind: Service
metadata:
  name: hotrod
spec:
  internalTrafficPolicy: Cluster
  ipFamilies:
    - IPv4
  ipFamilyPolicy: SingleStack
  ports:
    - name: frontend
      port: 8080
      protocol: TCP
      targetPort: 8080
    - name: customer
      port: 8081
      protocol: TCP
      targetPort: 8081
    - name: router
      port: 8083
      protocol: TCP
      targetPort: 8083
  selector:
    service_name: hotrod
  sessionAffinity: None
  type: ClusterIP
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jaeger
spec:
  replicas: 1
  selector:
    matchLabels:
      service.cpaas.io/name: jaeger
      service_name: jaeger
  template:
    metadata:
      labels:
        service.cpaas.io/name: jaeger
        service_name: jaeger
    spec:
      containers:
```



```

    - name: jaeger
      env:
        - name: LOG_LEVEL
          value: debug
      image: jaegertracing/all-in-one:1.58.1
      imagePullPolicy: IfNotPresent
      hostNetwork: true
      tolerations:
        - operator: Exists
  ---
apiVersion: v1
kind: Service
metadata:
  name: jaeger
spec:
  internalTrafficPolicy: Cluster
  ipFamilies:
    - IPv4
  ipFamilyPolicy: SingleStack
  ports:
    - name: http
      port: 4318
      protocol: TCP
      targetPort: 4318
  selector:
    service_name: jaeger
  sessionAffinity: None
  type: ClusterIP
  ---
apiVersion: crd.alauda.io/v2
kind: ALB2
metadata:
  name: otel-alb
spec:
  config:
    loadbalancerName: otel-alb
  otel:
    enable: true
    exporter:
      collector:
        address: 'http://jaeger.default.svc.cluster.local:4318'
        request_timeout: 1000
  projects:
    - ALL_ALL

```

```
replicas: 1
resources:
  alb:
    limits:
      cpu: 200m
      memory: 2Gi
    requests:
      cpu: 50m
      memory: 128Mi
  limits:
    cpu: '1'
    memory: 1Gi
  requests:
    cpu: 50m
    memory: 128Mi
type: nginx
```

2. 在 CLI 工具中执行以下命令，部署 Jaeger、ALB、HotROD 以及所有测试所需的 CR。

```
kubectl apply ./all.yaml
```

3. 执行以下命令以获取 Jaeger 的访问地址。

```
export JAEGER_IP=$(kubectl get po -A -o wide |grep jaeger | awk '{print $7}');echo "http://$JAEGER_IP:16686"
```

4. 执行以下命令以获取 otel-alb 的访问地址。

```
export ALB_IP=$(kubectl get po -A -o wide|grep otel-alb | awk '{print $7}');echo $ALB_IP
```

5. 执行以下命令通过 ALB 向 HotROD 发送请求。此时，ALB 会将 Trace 上报到 Jaeger。

```
curl -v "http://<$ALB_IP>:80/dispatch?customer=567&nonce=" # Replace <$ALB_IP> in the command with the access address of otel-alb obtained in the previous procedure
```

6. 打开在 [步骤 3](#) 中获取的 Jaeger 访问地址以查看结果。

JAEGER UI

Search

Compare

System Architecture

Monitor

Q

Lookup by Trace ID.

About Jaeger

Search

Upload

Service (6)

frontend

Operation (3)

all

Tags

http.status_code=200 error=true

Lookback

Last Hour

Max Duration

Min Duration

e.g. 1.2s, 100ms, 500...

e.g. 1.2s, 100ms, 500...

Limit Results

20

Find Traces

Duration

500ms

0µs

-500000µs

06:13:20 am

08:00:00 am

09:46:40 am

Time

1 Trace

Sort: Most RecentDownload ResultsDeep Dependency Graph

Compare traces by selecting result items

otel-alb: GET /dispatch?customer=567&nonce=c6294a7689.87ms

52 Spans3 Errorscustomer (1)driver (1)frontend (13)mysql (1)otel-alb (12)redis-manual (14)route (10)Today 12:08:39 pm a few seconds ago

otel-alb: GET /dispatch?customer=567&nonse=c6294a7

Trace Start **August 12, 2024, 12:08:39.606** | Duration **689.87ms** | Services **7** | Depth **6** | Total Spans **52**

0µs 172.47ms 344.93ms

Service & Operation

- otel-alb GET /dispatch?customer=567&nonse=c6294a7
 - frontend /dispatch
 - frontend HTTP GET
 - otel-alb GET /customer?customer=567
 - customer /customer
 - mysql → mysql SQL SELECT
 - frontend driver.DriverService/FindNearest
 - driver driver.DriverService/FindNearest
 - redis-manual FindDriverIDs
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - redis-manual GetDriver
 - frontend HTTP GET
 - otel-alb GET /route?dropoff=211%2C653&pickup=947%2C653
 - route /route
 - frontend HTTP GET
 - otel-alb GET /route?dropoff=211%2C653&pickup=320%2C653

0µs 172.47ms

280.28ms 279.54ms 278.71ms 278.44ms 231.78ms 231.41ms

TCP/HTTP Keepalive

目录

基本概念

CRD

基本概念

1. ALB 支持在端口级别配置 keepalive，配置项可在 frontend 上设置。
2. Keepalive 是客户端与 **ALB** 之间的连接保持，而非 **ALB** 与后端之间。
3. 通过 Nginx 配置实现，且当配置变更时，Nginx 需要且会自动重载。
4. TCP keepalive 和 HTTP keepalive 是两个不同的概念：
 1. **TCP keepalive** 是 TCP 协议的特性，当没有数据传输时，发送周期性的探测包以检测连接是否仍然存活，有助于发现并清理死连接。
 2. **HTTP keepalive**（也称为持久连接）允许多个 HTTP 请求复用同一个 TCP 连接，避免了建立新连接的开销，从而通过减少延迟和资源使用提升性能。

CRD

```
keepalive:
  properties:
    http:
      description: 下游 L7 keepalive
      properties:
        header_timeout:
          description: Keepalive header 超时时间。默认不设置。
          type: string
        requests:
          description: Keepalive 请求数。默认是 1000。
          type: integer
        timeout:
          description: Keepalive 超时时间。默认是 75s。
          type: string
      type: object
    tcp:
      description: TCPKeepAlive 定义 TCP keepalive 参数 (SO_KEEPALIVE)
      properties:
        count:
          description: TCP_KEEPCNT 套接字选项。
          type: integer
        idle:
          description: TCP_KEEPIDLE 套接字选项。
          type: string
        interval:
          description: TCP_KEEPINTVL 套接字选项。
          type: string
      type: object
    type: object
```

仅能在 Frontend 的 `.spec.config.keepalive` 上配置。

ALB 与 Ingress-NGINX 注解兼容性

WARNING

ALB 已被废弃。如果您需要使用 ingress-nginx 注解，请使用 [ingress-nginx-operator](#)。

基本概念

ingress-nginx 是 Kubernetes 中常用的 Ingress Controller，定义了许多注解以实现官方 ingress 定义之外的各种功能。

ALB 支持部分这些注解。

支持的 ingress-nginx 注解

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持或可通过配置实现)
nginx.ingress.kubernetes.io/app-root	string	x
nginx.ingress.kubernetes.io/affinity	cookie	o ingress 不支持。 alb 规则可配置 cookie hash
nginx.ingress.kubernetes.io/use-regex	bool	

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持或可通过配置实现)
nginx.ingress.kubernetes.io/affinity-mode	"balanced" or "persistent"	o ingress 不支持。 alb 规则可配置 session persistence
nginx.ingress.kubernetes.io/affinity-canary-behavior	"sticky" or "legacy"	o ingress 不支持。 alb 规则可配置 session persistence
nginx.ingress.kubernetes.io/auth-realm	string	v auth
nginx.ingress.kubernetes.io/auth-secret	string	v auth
nginx.ingress.kubernetes.io/auth-secret-type	string	v auth
nginx.ingress.kubernetes.io/auth-type	"basic" or "digest"	v auth
nginx.ingress.kubernetes.io/auth-tls-secret	string	x
nginx.ingress.kubernetes.io/auth-tls-verify-depth	number	x
nginx.ingress.kubernetes.io/auth-tls-verify-client	string	x
nginx.ingress.kubernetes.io/auth-tls-error-page	string	x
nginx.ingress.kubernetes.io/auth-tls-pass-certificate-to-upstream	"true" or "false"	x
nginx.ingress.kubernetes.io/auth-tls-match-cn	string	x
nginx.ingress.kubernetes.io/auth-url	string	v

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持 或可通过配置实现)
nginx.ingress.kubernetes.io/auth-cache-key	string	x
nginx.ingress.kubernetes.io/auth-cache-duration	string	x
nginx.ingress.kubernetes.io/auth-keepalive	number	x
nginx.ingress.kubernetes.io/auth-keepalive-share-vars	"true" or "false"	x
nginx.ingress.kubernetes.io/auth-keepalive-requests	number	x
nginx.ingress.kubernetes.io/auth-keepalive-timeout	number	x
nginx.ingress.kubernetes.io/auth-proxy-set-headers	string	v
nginx.ingress.kubernetes.io/auth-snippet	string	x
nginx.ingress.kubernetes.io/enable-global-auth	"true" or "false"	o auth
nginx.ingress.kubernetes.io/backend-protocol	string	v
nginx.ingress.kubernetes.io/canary	"true" or "false"	x
nginx.ingress.kubernetes.io/canary-by-header	string	x
nginx.ingress.kubernetes.io/canary-by-header-value	string	x
nginx.ingress.kubernetes.io/canary-by-header-pattern	string	x
nginx.ingress.kubernetes.io/canary-by-cookie	string	x

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持或可通过配置实现)
nginx.ingress.kubernetes.io/canary-weight	number	x
nginx.ingress.kubernetes.io/canary-weight-total	number	x
nginx.ingress.kubernetes.io/client-body-buffer-size	string	x
nginx.ingress.kubernetes.io/configuration-snippet	string	x
nginx.ingress.kubernetes.io/custom-http-errors	[]int	x
nginx.ingress.kubernetes.io/custom-headers	string	o
nginx.ingress.kubernetes.io/default-backend	string	o 可使用 ingress 的 default-backend
nginx.ingress.kubernetes.io/enable-cors	"true" or "false"	v
nginx.ingress.kubernetes.io/cors-allow-origin	string	v
nginx.ingress.kubernetes.io/cors-allow-methods	string	v
nginx.ingress.kubernetes.io/cors-allow-headers	string	v
nginx.ingress.kubernetes.io/cors-expose-headers	string	x
nginx.ingress.kubernetes.io/cors-allow-credentials	"true" or "false"	x
nginx.ingress.kubernetes.io/cors-max-age	number	x

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持或可通过配置实现)
nginx.ingress.kubernetes.io/force-ssl-redirect	"true" or "false"	v redirect
nginx.ingress.kubernetes.io/from-to-www-redirect	"true" or "false"	x
nginx.ingress.kubernetes.io/http2-push-preload	"true" or "false"	x
nginx.ingress.kubernetes.io/limit-connections	number	x
nginx.ingress.kubernetes.io/limit-rps	number	x
nginx.ingress.kubernetes.io/global-rate-limit	number	x
nginx.ingress.kubernetes.io/global-rate-limit-window	duration	x
nginx.ingress.kubernetes.io/global-rate-limit-key	string	x
nginx.ingress.kubernetes.io/global-rate-limit-ignored-cidrs	string	x
nginx.ingress.kubernetes.io/permanent-redirect	string	v redirect
nginx.ingress.kubernetes.io/permanent-redirect-code	number	v redirect
nginx.ingress.kubernetes.io/temporal-redirect	string	v redirect
nginx.ingress.kubernetes.io/preserve-trailing-slash	"true" or "false"	x
nginx.ingress.kubernetes.io/proxy-body-size	string	x

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持 或可通过配置实现)
nginx.ingress.kubernetes.io/proxy-cookie-domain	string	x
nginx.ingress.kubernetes.io/proxy-cookie-path	string	x
nginx.ingress.kubernetes.io/proxy-connect-timeout	number	v <u>timeout</u>
nginx.ingress.kubernetes.io/proxy-send-timeout	number	v <u>timeout</u>
nginx.ingress.kubernetes.io/proxy-read-timeout	number	v <u>timeout</u>
nginx.ingress.kubernetes.io/proxy-next-upstream	string	x
nginx.ingress.kubernetes.io/proxy-next-upstream-timeout	number	x
nginx.ingress.kubernetes.io/proxy-next-upstream-tries	number	x
nginx.ingress.kubernetes.io/proxy-request-buffering	string	x
nginx.ingress.kubernetes.io/proxy-redirect-from	string	x
nginx.ingress.kubernetes.io/proxy-redirect-to	string	x
nginx.ingress.kubernetes.io/proxy-http-version	"1.0" or "1.1"	x
nginx.ingress.kubernetes.io/proxy-ssl-secret	string	x

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持 或可通过配置实现)
nginx.ingress.kubernetes.io/proxy-ssl-ciphers	string	x
nginx.ingress.kubernetes.io/proxy-ssl-name	string	x
nginx.ingress.kubernetes.io/proxy-ssl-protocols	string	x
nginx.ingress.kubernetes.io/proxy-ssl-verify	string	x
nginx.ingress.kubernetes.io/proxy-ssl-verify-depth	number	x
nginx.ingress.kubernetes.io/proxy-ssl-server-name	string	x
nginx.ingress.kubernetes.io/enable-rewrite-log	"true" or "false"	x
nginx.ingress.kubernetes.io/rewrite-target	URI	v
nginx.ingress.kubernetes.io/satisfy	string	x
nginx.ingress.kubernetes.io/server-alias	string	x
nginx.ingress.kubernetes.io/server-snippet	string	x
nginx.ingress.kubernetes.io/service-upstream	"true" or "false"	x
nginx.ingress.kubernetes.io/session-cookie-change-on-failure	"true" or "false"	x
nginx.ingress.kubernetes.io/session-cookie-conditional-samesite-none	"true" or "false"	x
nginx.ingress.kubernetes.io/session-cookie-domain	string	x

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持或可通过配置实现)
nginx.ingress.kubernetes.io/session-cookie-expires	string	x
nginx.ingress.kubernetes.io/session-cookie-max-age	string	x
nginx.ingress.kubernetes.io/session-cookie-name	string	x
nginx.ingress.kubernetes.io/session-cookie-path	string	x
nginx.ingress.kubernetes.io/session-cookie-samesite	string	x
nginx.ingress.kubernetes.io/session-cookie-secure	string	x
nginx.ingress.kubernetes.io/ssl-redirect	"true" or "false"	v
nginx.ingress.kubernetes.io/ssl-passthrough	"true" or "false"	x
nginx.ingress.kubernetes.io/stream-snippet	string	x
nginx.ingress.kubernetes.io/upstream-hash-by	string	x
nginx.ingress.kubernetes.io/x-forwarded-prefix	string	x
nginx.ingress.kubernetes.io/load-balance	string	x
nginx.ingress.kubernetes.io/upstream-vhost	string	v
nginx.ingress.kubernetes.io/denylist-source-range	CIDR	o 可通过 modsecurity 实现类似效果

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持 或可通过配置实现)
nginx.ingress.kubernetes.io/whitelist-source-range	CIDR	o 可通过 modsecurity 实现类似效果
nginx.ingress.kubernetes.io/proxy-buffering	string	x
nginx.ingress.kubernetes.io/proxy-buffers-number	number	x
nginx.ingress.kubernetes.io/proxy-buffer-size	string	x
nginx.ingress.kubernetes.io/proxy-max-temp-file-size	string	x
nginx.ingress.kubernetes.io/ssl-ciphers	string	x
nginx.ingress.kubernetes.io/ssl-prefer-server-ciphers	"true" or "false"	x
nginx.ingress.kubernetes.io/connection-proxy-header	string	x
nginx.ingress.kubernetes.io/enable-access-log	"true" or "false"	o 默认启用 access_log，格式固定
nginx.ingress.kubernetes.io/enable-opentelemetry	"true" or "false"	v otel
nginx.ingress.kubernetes.io/opentelemetry-trust-incoming-span	"true" or "false"	v otel
nginx.ingress.kubernetes.io/enable-modsecurity	bool	v modsecurity
nginx.ingress.kubernetes.io/enable-owasp-core-rules	bool	v modsecurity

名称	类型	支持情况 (v 支持 x 不支持 o 部分支持 或可通过配置实现)
nginx.ingress.kubernetes.io/modsecurity-transaction-id	string	v <u>modsecurity</u>
nginx.ingress.kubernetes.io/modsecurity-snippet	string	v <u>modsecurity</u>
nginx.ingress.kubernetes.io/mirror-request-body	string	x
nginx.ingress.kubernetes.io/mirror-target	string	x
nginx.ingress.kubernetes.io/mirror-host	string	x

[Alauda Container Platform](#) > [网络](#) > [网络操作器](#) > [ALB Operator](#) > ALB 监控

ALB 监控

目录

术语

操作步骤

监控指标

ALB 流量监控

ALB 资源使用

Ingress、HTTPRoute、Rule 流量监控

术语

术语	描述
ALB	平台自研的第七层负载均衡器。

操作步骤

1. 进入 管理员。
2. 在左侧导航栏，点击 运维中心 > 监控 > 监控面板。

3. 点击页面顶部的 集群，切换到要监控的集群。
4. 点击页面右上角的 切换。
5. 通过以下两种方式进入 **ALB** 状态 监控面板：
 - 方式一：点击 **container-platform** 卡片，展开监控目录，然后点击 **ALB** 状态 名称进入监控面板。需要时可将该监控面板设置为主面板。
 - 方式二：在搜索框输入关键词（如 alb）进行搜索，然后点击 **ALB** 状态 名称进入监控面板。需要时可将该监控面板设置为主面板。
6. 通过面板查看各项监控指标。
 - 选择要监控的命名空间：点击页面顶部的 **namespace**，选择要监控的命名空间，默认为全部，即监控所有命名空间。
 - 选择要监控的 **ALB**：点击页面顶部的 **name**，选择要监控的 ALB，默认为全部，即监控所有 ALB。

监控指标

展示所选 ALB 在 最近 **5** 分钟 内的总流量、资源使用情况、Ingress（入站规则）、HTTPRoute（HTTPRoute 类型的路由规则）和 Rule（既非 Ingress 也非 HTTPRoute 的规则）的监控指标。

注意：所有数据均为 最近 **5** 分钟 内采集的监控数据。

ALB 流量监控

监控指标	描述
活动连接数	所选 ALB 上的活动连接数。
每秒请求数	所选 ALB 每秒接收的请求总数。
错误率	所选 ALB 每秒发生的 4XX（如 404）和 5XX 错误请求的比例。
延迟	所选 ALB 上请求的平均延迟。

ALB 资源使用

监控指标	描述
CPU 使用率	所选 ALB 的 CPU 使用率。
内存使用率	所选 ALB 的内存使用率。
网络接收/发送	所选 ALB 的网络 I/O 吞吐量。
磁盘读写速率	所选 ALB 的磁盘 I/O 吞吐量。

Ingress、HTTPRoute、Rule 流量监控

监控指标	描述
QPS （每秒查询数）	所选 ALB 上 Ingress/HTTPRoute/Rule 每秒接收的请求数，默认单位为 req/s。
请求 BPS （每秒字节数）	所选 ALB 上 Ingress/HTTPRoute/Rule 每秒接收请求的总字节数。
响应 BPS （每秒字节数）	所选 ALB 上 Ingress/HTTPRoute/Rule 发送响应的总字节数。
错误率	所选 ALB 上 Ingress/HTTPRoute/Rule 处理请求时发生错误的百分比。
P50、P90、P99	所选 ALB 上请求的响应时间，具体为中位响应时间。表示 50%、90% 和 99% 的请求响应时间小于或等于该值。 注意：P50、P90 和 P99 的原理是将采集的数据从小到大排序，取第 50%、90% 和 99% 位置的数据值；因此，采集的 50%、90% 和 99% 的数据都低于该值。百分位数有助于分析数据分布并识别各种极端情况。
上游 P50 、上游 P90 、上游 P99	上游服务的请求响应时间。表示发送到上游服务的请求中，50%、90% 和 99% 的响应时间小于或等于该值。

[Alauda Container Platform](#) > [网络](#) > 网络安全

网络安全

理解网络策略 [API](#)

介绍

KUBE-OVN 对网络策略的评估

带作用域的 AdminNetworkPolicy 与带命名

管理员网络策略

说明

通过 Web 控制台创建 AdminNetworkPolicy

通过 CLI 创建 AdminNetworkPolicy 或 Bas

网络策略

通过 Web 控制台

通过 CLI 创建 N

参考

理解网络策略 API

目录

介绍

KUBE-OVN 对网络策略的评估

带作用域的 AdminNetworkPolicy 与带命名空间作用域的 NetworkPolicy 有何区别？

介绍

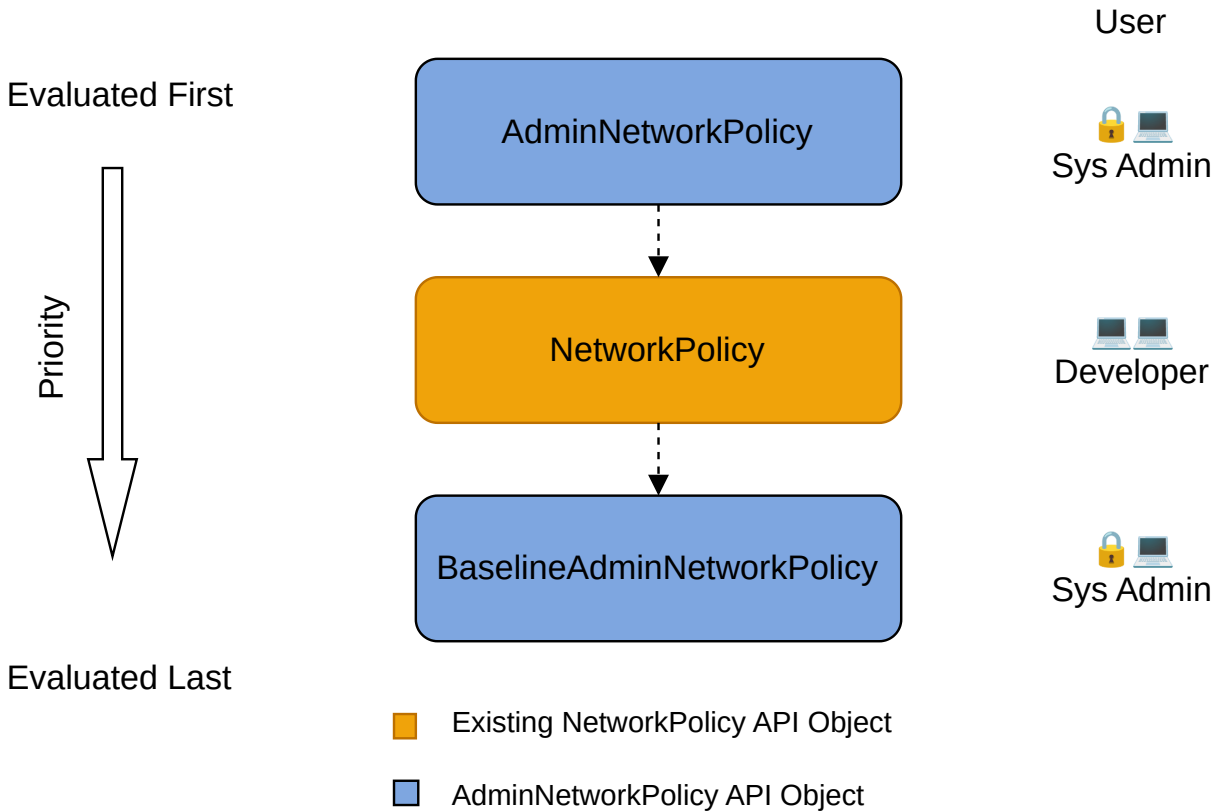
Kubernetes 风格集群中的网络安全涉及控制哪些工作负载（Pods、命名空间）可以通信以及如何通信。其核心是网络策略 API，支持分段（第 3/4 层）和流量控制。随着时间推移，更高级的 API 层出现，以支持集群范围的安全防护、多租户强制执行、零信任模型以及应用层（第 7 层）过滤。

本文重点介绍 API 模型 v1alpha1：AdminNetworkPolicy（ANP）和 BaselineAdminNetworkPolicy（BANP），它们在网络策略 API 中的位置，与标准 NetworkPolicy 的关系，以及如何在规划未来版本的同时，今天使用它们。

KUBE-OVN 对网络策略的评估

当应用多个网络策略时，它们遵循严格的优先级顺序：Admin Network Policy 优先于 Network Policy，后者又优先于 Baseline Admin Network Policy。

操作步骤如下：



带作用域的 **AdminNetworkPolicy** 与带命名空间作用域的 **NetworkPolicy** 有何区别？

维度	命名空间作用域的 NetworkPolicy	集群作用域的 AdminNetworkPolicy
作用域	仅适用于单个命名空间	适用于整个集群
受众 / 所有权	应用开发人员或命名空间所有者	平台管理员或集群管理员
主要用途	工作负载级别的微分段（L3/L4）	强制执行全局或跨命名空间访问规则
控制目标	由 <code>podSelector</code> 选定的命名空间内 Pods	由顶层 <code>subject</code> 选定的命名空间或 Pods
策略优先级	中等优先级 — 在管理员级别规则之后评估	优先级更高 — 覆盖命名空间策略

维度	命名空间作用域的 NetworkPolicy	集群作用域的 AdminNetworkPolicy
典型用例	允许前端 → 后端，限制数据库访问等	限制租户间流量，强制出口限制，设置全局保护
可见性	仅影响创建它的命名空间	可同时影响多个命名空间
跨命名空间规则	仅通过 namespaceSelector 间接允许	一等公民的跨命名空间流量控制
风险面	配置错误影响单个命名空间	配置错误可能影响整个集群流量

管理员网络策略

INFO

平台现提供两种不同的集群网络策略 UI。旧版 UI 为兼容性保留，新版 UI 更加灵活，并提供原生 YAML 编辑器。建议使用新版 UI。

请联系平台管理员启用 `cluster-network-policy` 和 `cluster-network-policy-next` 功能门控，以访问新版 UI。

目录

说明

通过 Web 控制台创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy

通过 CLI 创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy

说明

- 目前仅 Kube-OVN CNI 支持管理员网络策略。
- 在 Kube-OVN 网络模式下，该功能处于 Alpha 版本阶段。
- 集群中只能存在一个 Baseline Admin Network Policy。

AdminNetworkPolicy


```
# example-anp.yaml
apiVersion: policy.networking.k8s.io/v1alpha1
kind: AdminNetworkPolicy
metadata:
  name: example-anp
spec:
  priority: 3 ❶
  subject: ❷
    pods:
      namespaceSelector:
        matchLabels: {}
      podSelector:
        matchLabels:
          pod-template-hash: 55f66dd67d
  ingress:
    - name: ingress1
      action: Allow ❸
      ports:
        - portNumber:
            protocol: TCP
            port: 8090
      from: ❹
        - pods:
            namespaceSelector:
              matchLabels: {}
            podSelector:
              matchLabels:
                pod-template-hash: 55c84b59bb
  egress:
    - name: egress1
      action: Allow
      ports:
        - portNumber:
            protocol: TCP
            port: 8080
      to: ❺
        - networks:
            - 10.1.1.1/23
```

❶ 数字越小，优先级越高。

❷ `subject`：最多只能指定命名空间选择器或 Pod 选择器中的一个。

③ `action` : 可用值为 Allow、Deny 和 Pass。Allow 表示允许流量访问，Deny 表示拒绝流量访问，Pass 表示允许流量并跳过后续优先级较低的集群网络策略，继续由其他策略（NetworkPolicy 和 BaselineAdminNetworkpolicy）处理流量。

④ 可用值为 Namespace Selector、Pod Selector。

⑤ 可用值为 Namespace Selector、Pod Selector、Node Selector、IP Block。

BaselineAdminNetworkpolicy:

```
# default.yaml
apiVersion: policy.networking.k8s.io/v1alpha1
kind: BaselineAdminNetworkPolicy
metadata:
  name: default ❶
spec:
  subject:
    pods:
      namespaceSelector:
        matchLabels: {}
      podSelector:
        matchLabels:
          pod-template-hash: 55c84b59bb
  ingress:
    - name: ingress1
      action: Allow
      ports:
        - portNumber:
            protocol: TCP
            port: 8888
      from:
        - pods:
            namespaceSelector:
              matchLabels: {}
            podSelector:
              matchLabels:
                pod-template-hash: 55f66dd67d
  egress:
    - name: egress1
      action: Allow ❷
      ports:
        - portNumber:
            protocol: TCP
            port: 8080
      to:
        - networks:
            - 3.3.3.3/23
```

❶ 集群中只能创建一个 metadata.name= default 的 baseline admin network policy。

❷ 可用值为 Allow、Deny。

通过 Web 控制台创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy

- 1. 进入 管理员 页面。
- 2. 在左侧导航栏点击 网络 > 集群网络策略。
- 3. 点击 创建管理员网络策略 或 配置 **Baseline** 管理员网络策略。
- 4. 按照以下说明完成相关配置。

区域	参数	说明
基本信息	名称	管理员网络策略或 Baseline 管理员网络策略的名称。
	优先级	决定策略的评估和应用顺序。数值越小优先级越高。 注意：Baseline 管理员网络策略无优先级。
目标 Pod	命名空间选择器	以键值对形式填写目标命名空间的标签。若不设置，策略将应用于当前集群的所有命名空间。指定后，策略仅应用于匹配这些选择器的命名空间中的 Pod。
	当前策略影响的目标 Pod 预览	点击 预览 查看该网络策略影响的目标 Pod。
	Pod 选择器	以键值对形式填写目标 Pod 的标签。若不设置，策略将应用于当前命名空间的所有 Pod。
	当前策略影响的目标 Pod 预览	点击 预览 查看该网络策略影响的目标 Pod。
入口流量	流量动作	指定如何处理目标 Pod 的入口流量。共有三种模式： Allow

区域	参数	说明
		(允许流量)、Deny (拒绝流量)、Pass (跳过所有优先级较低的管理员网络策略，允许流量由 Network Policy 处理，若无 Network Policy 则由 Baseline Admin Network Policy 处理)。 注意：Baseline 管理员网络策略不支持 Pass 动作。
	Pod 选择器	匹配集群中具有指定标签的命名空间或 Pod；只有匹配的 Pod 能访问目标 Pod。可点击 预览 查看当前规则影响的 Pod。 - 若同时配置了命名空间和 Pod 选择器，则取交集，即从指定命名空间中选择具有指定标签的 Pod。 - 若未配置此项，则默认允许集群中所有命名空间的所有 Pod 访问目标 Pod。
	命名空间选择器	匹配当前命名空间中具有指定标签的 Pod；只有匹配的 Pod 能访问目标 Pod。可点击 预览 查看当前规则影响的 Pod。若未配置此项，则默认允许当前命名空间的所有 Pod 访问目标 Pod。
	端口	匹配指定协议和端口的流量；可添加数字端口或 Pod 上的端口名称。若未配置此项，则匹配所有端口。

区域	参数	说明
出口流量	规则	节点选择器 指定目标 Pod 允许访问的节点 IP。可通过节点标签选择节点，控制 Pod 可访问的节点 IP。
	说明：规则中添加多个来源时，它们之间是逻辑 或 关系。	IP 范围 指定目标 Pod 允许连接的 CIDR 范围。若未配置此项，则默认允许目标 Pod 连接任意 IP。
		其他参数 与入口流量参数类似，配置选项和行为相同。

通过 CLI 创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy

```
kubectl apply -f example-anp.yaml -f default.yaml
```

网络策略

INFO

平台目前提供了两种不同的网络策略 UI。旧版 UI 为兼容性保留，新版 UI 更加灵活且提供了原生的 YAML 编辑器。我们推荐使用新版。

请联系平台管理员开启 `network-policy-next` 功能门以访问新版 UI。

NetworkPolicy 是一个命名空间范围的 Kubernetes 资源，由 CNI 插件实现。通过网络策略，您可以控制 Pod 的网络流量，实现网络隔离，降低攻击风险。

默认情况下，所有 Pod 可以自由通信，允许来自任何来源的入口和出口流量。当应用 NetworkPolicy 后，目标 Pod 只接受符合策略规范的流量。

WARNING

网络策略仅作用于容器流量，不影响以 **hostNetwork** 模式运行的 Pod。

示例 NetworkPolicy：

```
# example-network-policy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: example
  namespace: demo-1
  annotations:
    cpaas.io/display-name: test
spec:
  podSelector:
    matchLabels:
      pod-template-hash: 55c84b59bb
  ingress:
    - ports:
        - protocol: TCP
          port: 8989
      from: ①
        - podSelector:
            matchLabels:
              kubevirt.io/vm: test
  egress:
    - ports:
        - protocol: TCP
          port: 80
      to:
        - ipBlock:
            cidr: 192.168.66.221/23
            except: []
  policyTypes:
    - Ingress
    - Egress
```

① from 和 to 对等体支持 namespaceSelector、podSelector、ipBlock

目录

通过 Web 控制台创建 NetworkPolicy

通过 CLI 创建 NetworkPolicy

通过 Web 控制台创建 NetworkPolicy

- 1. 进入 **Container Platform**。
- 2. 在左侧导航栏点击 **Network > Network Policies**。
- 3. 点击 **Create Network Policy**。
- 4. 参考以下说明完成相关配置。

区域	参数	说明
目标 Pod	Pod Selector	以键值对形式输入目标 Pod 的标签；若不设置，则应用于当前命名空间内所有 Pod。
	当前策略影响的目标 Pod 预览	点击 Preview 查看当前网络策略影响的目标 Pod。
入口流量	阻止所有入口流量	阻止所有进入目标 Pod 的入口流量。 注意： • 如果在 YAML 的 <code>spec.policyTypes</code> 字段中添加了 Ingress，但未配置具体规则，切换回表单时会自动勾选阻止所有入口流量 选项。 • 如果同时删除了 YAML 中的 <code>spec.ingress</code>、<code>spec.egress</code> 和 <code>spec.policyTypes</code> 字段，切换回表单时也会自动勾选 阻止所有入口流量 选项。

区域	参数	说明
	当前命名空间内的 Pod	匹配当前命名空间内具有指定标签的 Pod；只有匹配的 Pod 可以访问目标 Pod。您可以点击 Preview 查看当前规则影响的 Pod。若未配置此项，默认允许当前命名空间内所有 Pod 访问目标 Pod。
	规则	匹配集群内具有指定标签的命名空间或 Pod；只有匹配的 Pod 可以访问目标 Pod。您可以点击 Preview 查看当前规则影响的 Pod。
	说明: 如果规则中添加多个来源，它们之间是逻辑上的或 关系。	- 如果同时配置了命名空间选择器和 Pod 选择器，则取两者的交集，即从指定命名空间中选择具有指定标签的 Pod。 - 若未配置此项，默认允许集群内所有命名空间的 Pod 访问目标 Pod。
	IP 范围	输入允许访问目标 Pod 的 CIDR，可以排除不允许访问的 CIDR 范围。若未配置此项，任何流量均可访问目标 Pod。 说明: 可以通过添加形如 <i>example_ip/32</i> 的排除项来排除单个 IP 地址。
	端口	匹配指定协议和端口的流量；可添加数字端口或 Pod 上的端口名称。若未配置此项，则匹配所有端口。
出口流量	阻止所有出口流量	阻止所有目标 Pod 的出口流量。 注意： 如果在 YAML 的 <code>spec.policyTypes</code> 字段中添

区域	参数	说明
		加了 Egress，但未配置具体规则，切换回表单时会自动勾选阻止所有出口流量 选项。
	其他参数	与 入口流量 参数类似，此处不再赘述。

5. 点击 **Create**。

通过 CLI 创建 NetworkPolicy

```
kubectl apply -f example-network-policy.yaml
```

参考

如需更多详情，请查阅官方文档 [Network Policies](#)。

Ingress 和负载均衡

使用 [Envoy Gateway](#) 进行 Ingress 和负载均衡

概述

网络流量架构

配置路由

配置 Ingress 集群流量

负载均衡策略

TLS 和安全性

与 MetalLB 集成

故障排查

使用 Envoy Gateway 进行 Ingress 和负载均衡

目录

概述

网络流量架构

配置路由

定义 GatewayClass

创建 Gateway

创建 HTTP 路由

配置 Ingress 集群流量

负载均衡策略

默认轮询

加权负载均衡

会话亲和性 (Sticky Sessions)

TLS 和安全性

TLS 终止

TLS 透传

Gateway 与 Service 之间的 mTLS

与 MetalLB 集成

故障排查

概述

Envoy Gateway 为 Kubernetes 集群提供第 7 层 (L7) Ingress 和负载均衡。它基于 Kubernetes Gateway API 构建，并建立在 Envoy Proxy 之上，支持高级的 HTTP、HTTPS、gRPC 和 TCP 路由，同时具备完整的可观测性和策略控制能力。

Envoy Gateway 通过定义 GatewayClass、Gateway 和 `Route` 资源，统一了 Ingress 流量管理模型，提供了一种标准、可移植的替代方案，以取代传统的 Ingress 对象。

网络流量架构



- 外部负载均衡器提供一个外部 IP，并将流量映射到集群中。
- Envoy Gateway 终止 TLS、应用路由规则，并将流量转发到后端服务。
- Service（虚拟 IP）在各节点上的 Pod 之间分发流量。

该架构以完全符合 Gateway API 的数据平面取代了“Ingress Controller”。

配置路由

定义 GatewayClass

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: envoy
spec:
  controllerName: gateway.envoyproxy.io/gatewayclass-controller
```

创建 Gateway

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: public-gateway
  namespace: gateway-system
spec:
  gatewayClassName: envoy
  listeners:
    - name: http
      protocol: HTTP
      port: 80
      allowedRoutes:
        namespaces:
          from: Same
    - name: https
      protocol: HTTPS
      port: 443
      tls:
        mode: Terminate
        certificateRefs:
          - kind: Secret
            name: site-cert
```

了解更多 [配置 Gateway API Gateway](#)。

创建 HTTP 路由

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: web-route
  namespace: default
spec:
  parentRefs:
    - name: public-gateway
      namespace: gateway-system
  hostnames:
    - 'app.example.com'
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /
      backendRefs:
        - name: web-service
          port: 8080
```

了解更多 [配置 Gateway API Route](#)。

说明

- Gateway 为 HTTP/S 流量暴露 80 和 443 端口。
- HTTPRoute 基于主机名和路径定义路由规则。
- 多个 HTTPRoute 对象可以共享同一个 Gateway。

配置 Ingress 集群流量

Envoy Gateway 会根据你的基础设施支持多种外部流量入口模式。

环境	入口类型	示例
云环境	Service type=LoadBalancer	云提供商分配公共 IP
裸金属	MetalLB + Service type=LoadBalancer	MetalLB 分配外部 IP

环境	入口类型	示例
边缘 / 内部	NodePort 或 hostNetwork	用于本地或私有访问

暴露 Envoy Gateway 的示例 Service :

```
apiVersion: v1
kind: Service
metadata:
  name: envoy-gateway
  namespace: gateway-system
spec:
  type: LoadBalancer
  selector:
    app: envoy
  ports:
    - name: http
      port: 80
      targetPort: 8080
    - name: https
      port: 443
      targetPort: 8443
```

你可以通过运行以下命令验证分配到的外部 IP :

```
kubectl get svc envoy-gateway -n gateway-system
```

负载均衡策略

Envoy Gateway 通过其后端引用提供灵活的负载均衡方法。

默认轮询

```
backendRefs:  
  - name: app-v1  
    port: 8080  
  - name: app-v2  
    port: 8080
```

加权负载均衡

```
backendRefs:  
  - name: app-v1  
    port: 8080  
    weight: 80  
  - name: app-v2  
    port: 8080  
    weight: 20
```

会话亲和性 (Sticky Sessions)

Envoy 通过策略配置支持基于 cookie 或源 IP 的会话亲和性：

```
apiVersion: gateway.envoyproxy.io/v1alpha1  
kind: BackendTrafficPolicy  
metadata:  
  name: web-policy  
  namespace: default  
spec:  
  targetRef:  
    group: gateway.networking.k8s.io  
    kind: HTTPRoute  
    name: web-route  
  sessionAffinity:  
    type: Cookie  
    cookie:  
      name: session_id  
      ttl: 3600s
```

了解更多关于 [BackendTrafficPolicy](#)。

TLS 和安全性

TLS 终止

在 Gateway 监听器级别处理：

```
tls:
  mode: Terminate
  certificateRefs:
    - kind: Secret
      name: site-cert
```

TLS 透传

用于端到端加密场景，此时 Envoy 会转发加密流量：

```
tls:
  mode: Passthrough
```

Gateway 与 Service 之间的 mTLS

高级策略 CRD 可启用 mTLS 或客户端证书验证。

与 MetalLB 集成

在没有云负载均衡器的裸金属集群中：

1. 配置 [MetalLB IPAddressPool](#) 和 L2Advertisement。
2. 确保 Envoy Gateway 的 Service 类型为 LoadBalancer。
3. MetalLB 会自动分配一个外部 IP。
4. 然后可以通过 DNS 记录将域名（例如 app.example.com）映射到该 IP。

故障排查

问题	可能原因	解决方案
Gateway 没有外部 IP	MetalLB 配置错误	检查 <code>Service</code> 类型和 MetalLB 控制器
HTTPRoute 被忽略	<code>parentRefs</code> 不匹配	确保 <code>Gateway</code> 和命名空间正确
TLS 无法工作	找不到 Secret	验证 <code>certificateRefs</code> 路径
Gateway 返回 404	没有匹配的主机名/路径	确认 <code>HTTPRoute</code> 规则
延迟尖峰	代理过载	扩展 Envoy Gateway 副本数

操作指南

配置域名

域名示例自定义资源（CR）

通过 Web 控制台创建域名

通过 CLI 创建域名

在多个集群中使用同一域名

后续操作

其他资源

创建证书

使用 Web 控制台创建证书

配置 Service

为什么需要 Service

ClusterIP 类型

配置 Ingress

实现方式

Ingress 示例：

使用 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 对等方

配置 GatewayAPI 策略

概述

前提条件

Policy Attachment 基础

Policy Attachment 概览

在 Web Console 中创建 Policy

SecurityPolicy

R R

ewi

础

ay

ay i

oute

BackendTLSPolicy

ClientTrafficPolicy

BackendTrafficPolicy

相关任务

[配置 GatewayAPI Route](#)

概述

前提条件

配置

视图

下一步

[配置 ALB](#)

ALB

Frontend

[配置 NodeLocal DNSCache](#)

概述

主要特性

重要说明

安装

工作原理

配置

[配置 CoreDNS](#)

概述

配置

配置域名

将域名资源添加到平台，并为集群下的所有项目或特定项目下的资源分配可用域名。创建域名时，支持绑定证书。

NOTE

在平台上创建的域名需要先解析到集群的负载均衡地址，才能通过域名访问。因此，你需要确保在平台上添加的域名已经成功注册，并且域名已解析到集群的负载均衡地址。

在平台上成功创建并分配的域名可用于 **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

当域名绑定证书后，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，即可使用域名自带的证书来支持 https。

目录

域名示例自定义资源（CR）

通过 Web 控制台创建域名

通过 CLI 创建域名

在多个集群中使用同一域名

通过 Web 控制台配置

[通过 CLI 配置](#)[后续操作](#)[其他资源](#)

域名示例自定义资源 (CR)

```
# 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 如果启用了证书，则必须提前创建 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. 在左侧导航栏中，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	描述
类型	Domain：完整域名，例如 <code>developer.test.cn</code>。

参数	描述
	Wildcard Domain：带通配符（*）的泛域名，例如 <code>*.test.cn</code>，包含域名 <code>test.cn</code> 下的所有子域名。
域名	根据所选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，还需要选择与该集群关联的项目，例如该集群关联的所有项目。
证书	包括用于创建域名绑定证书的公钥（tls.crt）和私钥（tls.key）。分配给证书的项目与绑定的域名相同。 说明： - 不支持二进制文件导入。 - 绑定的证书应满足格式正确、在有效期内、已针对该域名签名等条件。 - 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，可以在证书列表中查看该证书，但仅支持在域名详情页面更新和删除绑定证书。 - 创建绑定证书后，支持更新证书内容，但不支持替换其他证书。

5. 点击 创建。

通过 CLI 创建域名

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

在多个集群中使用同一域名

你可以通过在 global 集群中创建独立的 Domain 资源来配置同一域名在多个集群中使用，这些资源的 `spec.name` 值相同，但 `cluster.cpaas.io/name` 标签不同。

通过 Web 控制台配置

1. 按照[通过 Web 控制台创建域名](#)中的步骤操作。
2. 创建两个 相同域名（例如 `app.example.com`）的域名资源。
3. 为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，它们的 `spec.name` 相同，但 `cluster.cpaas.io/name` 标签不同：

NOTE

两个 Domain 资源中的域名（`spec.name`）必须相同，但资源的 `metadata.name` 应唯一。两个资源都创建在 global 集群中。

Cluster A 的域名：

```
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 # Same domain name
  kind: full
```

Cluster B 的域名：

```
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 # Same domain name as Cluster A
  kind: full
```

后续操作

- 域名注册：如果创建的域名尚未注册，请先注册该域名。
- 域名解析：如果域名尚未指向平台集群的负载均衡地址，请执行域名解析。

其他资源

- [配置证书](#)

创建证书

在平台管理员导入 TLS 证书并将其分配给指定项目后，具备相应项目权限的开发者在使用入口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，在证书到期等场景下，平台管理员可以集中更新证书。

NOTE

当前公有云集群暂不支持使用证书功能。您可以根据需要在指定 namespace 中创建 TLS 类型的 secret 字典。

目录

[使用 Web 控制台创建证书](#)

使用 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. 根据以下说明配置相关参数。

参数	说明
Assign Project	• All Projects：将证书分配给与当前集群关联的所有项目使用。 • Specified Project：将证书分配给指定项目使用。 • No Assignment：暂不分配项目。证书创建完成后，您可以通过 Update Project 操作更新可使用该证书的项目。
Public Key	指的是 tls.crt。导入公钥时，不支持二进制文件。
Private Key	指的是 tls.key。导入私钥时，不支持二进制文件。

5. 点击 **Create**。

配置 Service

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

目录

[为什么需要 Service](#)

ClusterIP 类型 Service 示例：

Headless Services

使用 Web 控制台创建 service

使用 CLI 创建 service

示例：访问集群内的应用

示例：访问集群外的应用

ExternalName 类型的 Service 示例

负载均衡器类型 Service 注解

AWS EKS 集群

华为云 CCE 集群

Azure AKS 集群

Google GKE 集群

使用 MetalLB BGP 和 Local Traffic Policy 的 LoadBalancer 示例

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（Pod 重新创建后会发生变化）。
- 直接访问 Pod 的可靠性较差。

2. Service 通过以下方式解决这一问题：

- 提供稳定的 IP 和 DNS 名称。
- 自动将流量负载均衡到匹配的 Pod。

ClusterIP 类型 Service 示例：

```
# 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 可用的类型值及其行为为 `ClusterIP`、`NodePort`、`LoadBalancer`、`ExternalName`

- 2 Service 所针对的 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，你还可以引用 pod 容器下的 `port.name`。

Headless Services

有时你并不需要负载均衡和单个 Service IP。在这种情况下，你可以创建所谓的 Headless Service：

```
spec:  
  clusterIP: None
```

Headless Service 适用于以下场景：

- 你希望发现单个 Pod IP，而不仅仅是一个 Service IP。
- 你需要直接连接到每个 Pod（例如 Cassandra 或 StatefulSet 这类数据库）。
- 你正在使用 StatefulSet，并且每个 Pod 都必须拥有稳定的 DNS 名称。

使用 Web 控制台创建 service

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 **Network > Services**。
3. 单击 **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，将为此 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 Service，通常由 StatefulSet 使用。

参数	说明
类型	• ClusterIP : 将 Service 暴露在集群内部 IP 上。选择此值后，Service 只能从集群内部访问。 • NodePort : 将 Service 暴露到每个 Node 的 IP 上的一个静态端口 (NodePort) 。 • ExternalName : 将 Service 映射到 externalName 字段的内容（例如主机名 api.foo.bar.example）。 • LoadBalancer : 使用外部负载均衡器将 Service 暴露到外部。 Kubernetes 不直接提供负载均衡组件；你必须自行提供，或者将你的 Kubernetes 集群与云服务商集成。
目标组件	• Workload : Service 将把请求转发到一个特定的 workload，该 workload 匹配诸如 <code>project.cpaas.io/name: projectname</code> 和 <code>service.cpaas.io/name: deployment-name</code> 这样的标签。 • Virtualization : Service 将把请求转发到一个特定的虚拟机或虚拟机组。 • Label Selector : Service 将把请求转发到具有指定标签的某一类 workload，例如 <code>environment: release</code> 。

参数	说明
端口	用于配置此 Service 的端口映射。在以下示例中，集群中的其他 Pod 可以通过虚拟 IP（如果已启用）和 TCP 端口 80 调用该 Service；访问请求将被转发到目标组件的 Pod 对外暴露的 TCP 端口 6379 或 <i>redis</i>。 • Protocol：Service 使用的协议，支持的协议包括：<code>TCP</code>、<code>UDP</code>、<code>HTTP</code>、<code>HTTP2</code>、<code>HTTPS</code>、<code>gRPC</code>。 • Service Port：Service 在集群内暴露的端口号，即 Port，例如 80。 • Container Port：service 端口映射到的目标端口号（或名称），即 targetPort，例如 6379 或 <i>redis</i>。 • Service Port Name：将自动生成，格式为 <code><protocol>-<service port>-<container port></code>，例如：<code>tcp-80-6379</code> 或 <code>tcp-80-redis</code>。
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期间都会保留在同一台服务器上，确保来自同一客户端的请求被转发到同一台服务器进行处理。

5. 单击 **Create**。

使用 CLI 创建 service

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

基于现有的 `my-app` deployment 资源创建一个 service。

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

示例：访问集群内的应用

```
# 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. 应用此 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` service :

```
wget -qO- http://nginx-clusterip
# or using DNS records created automatically by Kubernetes: <service-name>.<namespace>.svc.cluster.local
wget -qO- http://nginx-clusterip.default.svc.cluster.local
```

你应该会看到一个包含 "Welcome to nginx!" 等文本的 HTML 响应。

示例：访问集群外的应用

```
# 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. 应用此 YAML :

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

2. 查看 Pods :

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

3. curl Service :

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

你应该会看到一个包含 "Welcome to nginx!" 等文本的 HTML 响应。

当然，也可以通过创建一个类型为 LoadBalancer 的 Service，从集群外访问该应用。

注意：请先配置 LoadBalancer service。

```
# 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. 应用此 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 是你从浏览器访问的地址。

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

你应该会看到一个包含 "Welcome to nginx!" 等文本的 HTML 响应。

如果 EXTERNAL-IP 为 **pending**，则说明该 LoadBalancer service 当前未部署在集群中。

ExternalName 类型的 Service 示例

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

1. 应用此 YAML：

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

2. 尝试在集群中的 Pod 内进行解析：

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

然后：

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


你会看到它解析为 `example.com`。

负载均衡器类型 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	Description
service.beta.kubernetes.io/aws-load-balancer-type	external: 使用官方 AWS LoadBalancer Controller。	指定 LoadBalancer 类型使用的 controller。 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：流量将通过 NodePort 发送到 pods。 - ip：流量将直接路由到 pods（集群必须使用 Amazon VPC CNI）。	指定流量到达 pods 的方式。
service.beta.kubernetes.io/aws-load-balancer-scheme	- internal：私有网络。 - internet-facing：公网。	指定使用私有网络还是公网。
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：<pre>{"type": "public", "bandwidth_name": "cce-bandwidth-1551163379627", "bandwidth_chargemode": "bandwidth", "bandwidth_spec": [{"cn-north-4b"}, {"l4_flavor_name": "L4_flavor.elb.s1.small"}]</pre> 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	
kubernetes.io/elb.class	- union：共享负载均衡。 - performance：独享负载均衡，仅支持 Kubernetes 1.17 及以上版本
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	Description
service.beta.kubernetes.io/azure-load-balancer-internal	• true：私有网络。 • false：公网。	指定使用私有网络还是公网。

Google GKE 集群

有关 GKE LoadBalancer Service 注解的详细说明，请参阅 [Annotation Usage Documentation](#)。

Key	Value	Description
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 pods。

使用 MetalLB BGP 和 Local Traffic Policy 的 LoadBalancer 示例

此示例演示如何使用 MetalLB BGP 模式和 `externalTrafficPolicy: Local` 配置一个 LoadBalancer Service，以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 pods，不经过中间节点转发
- 更好的性能：`externalTrafficPolicy: Local` 可保留源 IP 并降低延迟
- 高可用性：BGP 路由通告可确保流量到达健康节点

前提条件

在配置 LoadBalancer Service 之前，请确保你已完成以下准备：

1. 已部署 **MetalLB**：安装说明请参阅 [创建外部 IP 地址池](#)
2. 已配置 **BGP Peer**：BGP 配置请参阅 [创建 BGP Peer](#)
3. 外部 **IP** 地址池：配置带有 BGPAdvertisement 的 IPAddressPool

步骤

使用 `externalTrafficPolicy: Local` 部署你的应用并创建 LoadBalancer Service：

```
# 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` 设置提供以下优势：

- 保留源 IP：保留客户端源 IP，便于正确记录日志并应用安全策略
- 直接 Pod 路由：流量直接到达 Pod，而不会经过节点级转发

使用 BGP 的 LoadBalancer

在 MetalLB 的 BGP 模式下：

- 路由由 BGPAdvertisement 中指定的节点进行通告
- BGP Peer 会接收这些通告并据此路由流量
- BGPPeer 与 BGPAdvertisement 之间的节点选择器对齐可确保路由一致

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
nginx-loadbalancer-local	LoadBalancer	10.0.2.57	4.4.4.3	8

3. 测试 service：

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

验证

- 监控 service endpoints：`kubectl get endpoints nginx-loadbalancer-local`

- 检查 **service** 状态：`kubectl describe svc nginx-loadbalancer-local`

配置 Ingress

Ingress 规则（Kubernetes Ingress）将来自集群外部的 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，用于管理对某个 Service 的外部 HTTP/HTTPS 访问。

WARNING

在同一 namespace 中创建多个 ingress 时，不同的 ingress **MUST NOT** 具有相同的 **Domain**、**Protocol** 和 **Path**（即，不允许重复的访问点）。

目录

实现方式

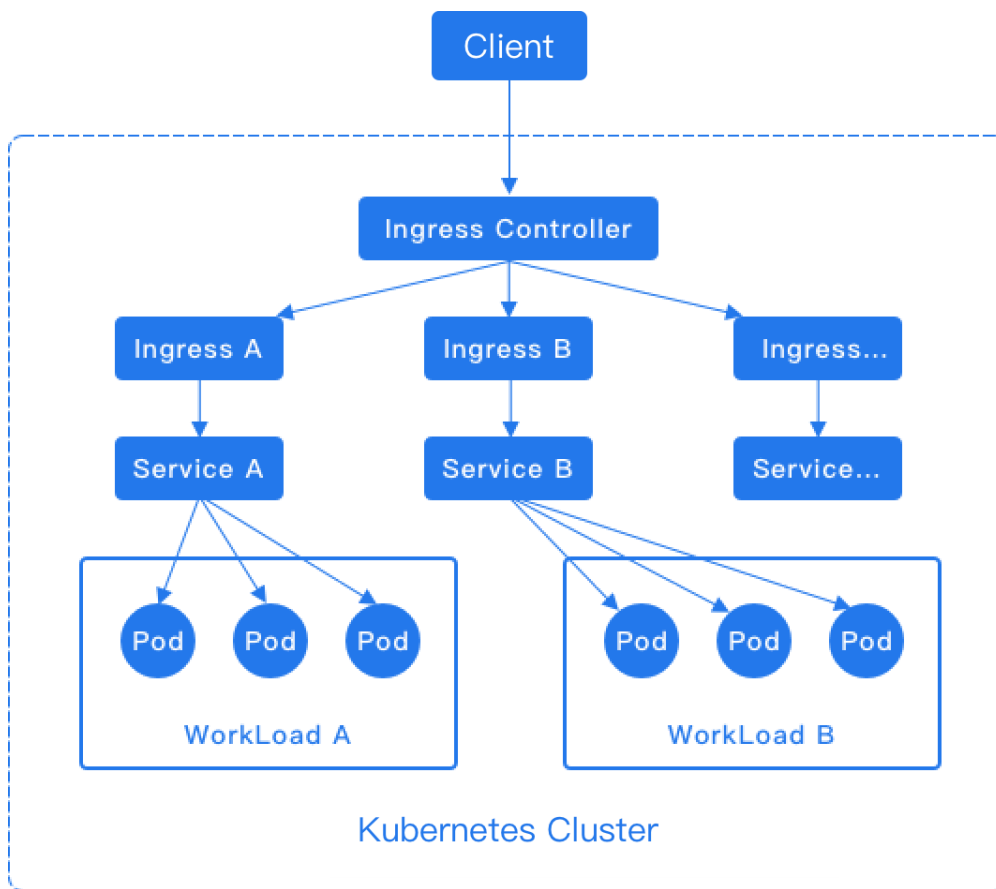
Ingress 示例：

使用 Web 控制台创建 Ingress

使用 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 的变更。创建新的 Ingress 后，当 Ingress Controller 收到请求时，它会匹配 Ingress 中的转发规则，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](#) 将 `ingress-nginx` 安装为 `ingress-controller`
- 通过 [alb-operator](#) 将 `alb` 安装为 `ingress-controller`

Ingress 示例：

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

❶ 如需查看更多配置，请参考 [nginx-configuration](#)。

❷ `nginx` 表示使用 `ingress-nginx` controller，`$alb_name` 表示使用 alb 作为 ingress controller。

❸ 如果只想在本地运行 ingress，请提前配置 `hosts`。

使用 Web 控制台创建 Ingress

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

参数	描述
Ingress Class	不同的 controller 可以通过不同的 <code>IngressClass</code> 名称来实现 Ingress。 如果平台上有多个 ingress controller，可通过此选项选择要使用的 controller。
Domain Name	主机名可以是精确匹配（例如 <code>foo.bar.com</code> ），也可以是通配符（例如 <code>*.foo.com</code> ）。可用的域名由平台管理员分配。
Certificates	由平台管理员分配的 TLS secret 或 Certificates。
Match Type 和 Path	• Prefix：匹配路径前缀，例如 <code>/abcd</code> 可以匹配 <code>/abcd/efg</code> 或 <code>/abcde</code>。 • Exact：匹配精确路径，例如 <code>/abcd</code>。 • Implementation specific：如果你使用自定义 Ingress controller 来管理 Ingress 规则，可以由 controller 自行决定。
Service	外部流量将转发到该 Service。
Service Port	指定将流量转发到哪个 Service 端口。

5. 点击 **Create**。

使用 CLI 创建 Ingress

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

配置子网

目录

IP 分配规则

Calico 网络

限制和约束

使用 Calico 网络的子网自定义资源（CR）示例

通过 Web 控制台在 Calico 网络中创建子网

通过 CLI 在 Calico 网络中创建子网

参考内容

Kube-OVN 网络

使用 Kube-OVN Overlay 网络的子网自定义资源（CR）示例

通过 Web 控制台在 Kube-OVN Overlay 网络中创建子网

通过 CLI 在 Kube-OVN Overlay 网络中创建子网

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

使用 Kube-OVN Underlay 网络的子网自定义资源（CR）示例

通过 Web 控制台在 Kube-OVN Underlay 网络中创建子网

通过 CLI 在 Kube-OVN Underlay 网络中创建子网

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

如果项目或命名空间分配了多个子网，则会从其中一个子网中随机选择一个 IP 地址。

- 项目分配：
 - 如果项目未绑定到子网，则该项目下所有命名空间中的 Pods 只能使用默认子网中的 IP 地址。如果默认子网中的 IP 地址不足，Pods 将无法启动。
 - 如果项目已绑定到子网，则该项目下所有命名空间中的 Pods 只能使用该特定子网中的 IP 地址。
- 命名空间分配：

- 如果命名空间未绑定到子网，则该命名空间中的 Pods 只能使用默认子网中的 IP 地址。如果默认子网中的 IP 地址不足，Pods 将无法启动。
- 如果命名空间已绑定到子网，则该命名空间中的 Pods 只能使用该特定子网中的 IP 地址。

Calico 网络

在 Calico 网络中创建子网，以实现集群内资源更细粒度的网络隔离。

限制和约束

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

使用 Calico 网络的子网自定义资源（CR）示例

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

- ❶ 当 `default` 为 true 时，使用 VXLAN 封装。
- ❷ 请参见 Encapsulation Mode 参数和 Encapsulation Protocol 参数。
- ❸ 请参见 Outbound Traffic NAT 参数。

通过 Web 控制台在 Calico 网络中创建子网

1. 进入 管理员。
2. 在左侧导航栏中，点击 网络管理 > 子网。
3. 点击 创建子网。
4. 按照以下说明配置相关参数。

参数	描述
CIDR	将子网分配给项目或命名空间后，该命名空间中的容器组将随机使用此 CIDR 范围内的 IP 地址进行通信。 注意：关于 CIDR 与 BlockSize 的对应关系，请参见 参考内容。
Encapsulation Protocol	选择封装协议。双栈模式下不支持 IPIP。 • IPIP：使用 IPIP 协议实现分段间通信。 • VXLAN (Alpha)：使用 VXLAN 协议实现分段间通信。 • No Encapsulation：通过路由转发直接连接。
Encapsulation Mode	当封装协议为 IPIP 或 VXLAN 时，必须设置封装模式，默认值为 Always。 • Always：始终启用 IPIP / VXLAN 隧道。 • Cross Subnet：仅当主机位于不同子网时启用 IPIP / VXLAN 隧道；当主机位于同一子网时，通过路由转发直连。
Outbound Traffic NAT	选择是否启用出站流量 NAT（Network Address Translation），默认已启用。 它主要用于设置子网容器组访问外部网络时对外暴露的访问地址。 启用出站流量 NAT 时，将使用主机 IP 作为当前子网容器组的访问地址；未启用时，子网中的容器组 IP 将直接暴露给外部网络。

5. 点击 确认。
6. 在子网详情页中，选择 操作 > 分配项目 / 分配命名空间。
7. 完成配置后，点击 分配。

通过 CLI 在 Calico 网络中创建子网

```
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，以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络段冲突。

使用 Kube-OVN Overlay 网络的子网自定义资源（CR）示例

```
# 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 请参见 Outbound Traffic NAT 参数。
- ② 请参见 Reserved IP 参数。
- ③ 请参见 Gateway Type 参数。可用值为 `distributed` 或 `centralized`。
- ④ 请参见 Gateway Nodes 参数。

- 5 请参见 ECMP 参数。前提是已联系管理员启用该 feature gate。

通过 Web 控制台在 Kube-OVN Overlay 网络中创建子网

1. 进入 管理员。
2. 在左侧导航栏中，点击 网络管理 > 子网。
3. 点击 创建子网。
4. 按照以下说明配置相关参数。

参数	描述
Network Segment	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。
Reserved IP	设置的保留 IP 不会被自动分配。例如，可将其用作计算组件的固定 IP。
Gateway Type	选择子网的网关类型以控制出站流量。 - Distributed：集群中的每台主机都可作为当前主机上 Pods 的出站节点，实现分布式出口。 - Centralized：集群中的所有 Pods 使用一个或多个特定主机作为出站节点，便于外部审计和防火墙控制。设置多个集中式 gateway nodes 可实现高可用。
ECMP (Alpha)	在选择 Centralized 网关时，可以使用 ECMP 功能。默认情况下，网关以主从模式运行，仅主网关处理流量。启用 ECMP (Equal-Cost Multipath Routing) 后，出站流量将通过多条等价路径路由到所有可用的网关节点，从而提升网关总吞吐量。 注意：请提前启用与 ECMP 相关的功能。
Gateway Nodes	使用 Centralized 网关时，选择一个或多个特定主机作为网关节点。
Outbound Traffic NAT	选择是否启用出站流量 NAT (Network Address Translation)，默认已启用。 它主要用于设置 Pods 在子网中访问互联网时对外暴露的访问地址。 启用出站流量 NAT 时，将使用主机 IP 作为当前子网中 Pods 的访问地址；

参数	描述
	未启用时，子网内 Pods 的 IP 将直接暴露给外部网络。在这种情况下，建议使用集中式网关。

5. 点击 确认。
6. 在子网详情页中，选择 操作 > 分配项目 / 命名空间。
7. 完成配置后，点击 分配。

通过 CLI 在 Kube-OVN Overlay 网络中创建子网

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

Underlay 网络

在 Kube-OVN Underlay 网络中创建子网，不仅可以对资源进行更细粒度的网络隔离，还能提供更好的性能体验。

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）

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

- ❶ 默认网卡名称。
- ❷ 按节点配置网卡。

桥接网络是指一个 bridge，网卡绑定到该 bridge 后，可以转发容器网络流量，实现与物理网络的互通。

操作步骤：

1. 进入 管理员。
2. 在左侧导航栏中，点击 网络管理 > 桥接网络。
3. 点击 添加桥接网络。
4. 根据以下说明配置相关参数。

注意：

- 目标 Pod 指调度到当前节点上的所有 Pods，或命名空间绑定到特定子网且调度到当前节点上的 Pods。这取决于桥接网络下子网的范围。
- Underlay 子网中的节点必须具备多个网卡，且桥接网络使用的网卡必须专用于 Underlay，不能承载其他流量，例如 SSH。比如，桥接网络有三台节点，计划将 eth0、eth0、eth1 专用于 Underlay，那么默认网卡可设置为 eth0，第三台节点的网卡可设置为 eth1。

参数	描述
默认网卡名称	默认情况下，目标 Pod 将使用该网卡作为桥接网卡与物理网络互通。
按节点配置网卡	配置节点上的目标 Pods 将桥接到指定网卡，而不是默认网卡。
排除节点	当节点被排除后，调度到这些节点上的所有 Pods 都不会桥接到这些节点上的任何网卡。 注意：排除节点上的 Pods 将无法与物理网络或跨节点容器网络通信，应谨慎避免将相关 Pods 调度到这些节点上。

5. 点击 添加。

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。

平台预配置了 **ovn-vlan** 虚拟局域网，它将连接到 **provider** 桥接网络。您也可以配置新的 VLAN 连接到其他桥接网络，从而实现 VLAN 之间的网络隔离。

操作步骤：

1. 进入 管理员。
2. 在左侧导航栏中，点击 网络管理 > **VLAN**。
3. 点击 添加 **VLAN**。
4. 根据以下说明配置相关参数。

参数	描述
VLAN ID	该 VLAN 的唯一标识符，用于区分不同的虚拟局域网。
Bridge Network	该 VLAN 将连接到此桥接网络，以便与物理网络互通。

5. 点击 添加。

通过 CLI 添加 VLAN

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

使用 Kube-OVN Underlay 网络的子网自定义资源（CR）示例

```
# 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 控制台在 Kube-OVN Underlay 网络中创建子网

NOTE

平台还预配置了一个用于 Overlay 传输模式下节点与 Pods 通信的 **join** 子网。该子网不会在 Underlay 传输模式下使用，因此务必避免 **join** 与其他子网之间的 IP 段冲突。

操作步骤：

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

参数	描述
VLAN	该子网所属的 VLAN。

参数	描述
Subnet	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。
Gateway	上述子网内的物理网关。
Reserved IP	指定的保留 IP 不会被自动分配。例如，可将其用作计算组件的固定 IP。

- 5. 点击 确认。
- 6. 在子网详情页中，选择 操作 > 分配项目 / 命名空间。
- 7. 完成配置后，点击 分配。

通过 CLI 在 Kube-OVN Underlay 网络中创建子网

```
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 子网之间自动互通](#)。

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 管理员。
- 2. 在左侧侧边栏中，点击 网络管理 > 子网。
- 3. 点击该子网名称。
- 4. 选择 操作 > 更新网关。

- 更新参数配置；详细信息请参见 [参数说明](#)。
- 点击 确定。

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": "192.168.66.210"},
  {"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，而其他保留 IP 可自由编辑、删除或新增。

- 进入 管理员。
- 在左侧侧边栏中，点击 网络管理 > 子网。
- 点击该子网名称。
- 选择 操作 > 更新保留 IP。
- 完成更新后，点击 更新。

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，确保每个项目都有足够的网络资源。

1. 进入 管理员。
2. 在左侧侧边栏中，点击 网络管理 > 子网。
3. 点击该子网名称。
4. 选择 操作 > 分配项目。
5. 添加或移除项目后，点击 分配。

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，可实现更细粒度的网络隔离。

注意：分配过程中会重建网关，出站数据包将被丢弃！请确保当前没有业务应用正在访问外部集群。

1. 进入 管理员。
2. 在左侧侧边栏中，点击 网络管理 > 子网。
3. 点击该子网名称。
4. 选择 操作 > 分配命名空间。

5. 添加或移除命名空间后，点击 [分配](#)。

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

当子网的保留 IP 范围接近使用上限或即将耗尽时，可以在不影响现有服务正常运行的情况下，基于原始子网范围进行扩展。

1. 进入 [管理员](#)。
2. 在左侧侧边栏中，点击 [网络管理](#) > [子网](#)。
3. 点击该子网名称。
4. 选择 [操作](#) > [扩展子网](#)。
5. 完成配置后，点击 [更新](#)。

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"
  }
]'
```

扩展默认 Pod 子网

如果集群中的 Pod IP 地址不足，可以按相同方式扩展默认 Pod 子网（`ovn-default`）。现有 Pods 会保留原有地址，新 Pods 将从扩展后的地址范围中获取地址。

子网只能在保持相同网络基地址的前提下扩展（即网络前缀变短）；不能修改基地址，也不能缩小范围。如果默认 Pod 子网无法原地扩展，例如更大的范围与当前基地址不对齐，则应新增一个子网，并将其绑定到需要更多地址的命名空间。

Service CIDR 在集群创建后无法更改。请在安装前规划好 Service 网络规模。

管理 Calico 网络

支持分配项目和命名空间；详情请参见 [项目分配](#) 和 [命名空间分配](#)。

通过 Web 控制台删除子网

NOTE

- 删除子网后，如果仍有容器组在使用该子网内的 IP，容器组可继续运行且 IP 地址保持不变，但将无法进行网络通信。可以重建容器组以使用默认子网中的 IP，或为容器组所在的命名空间重新分配新的子网后再使用。
- 默认子网不能删除。

1. 进入 管理员。
2. 在左侧导航栏中，点击 网络管理 > 子网。
3. 点击 ⋮ > 删除，并继续完成删除操作。

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 对等方

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

监控 MetalLB

查看内置的 MetalLB 监控面板

访问 MetalLB 指标端点

MetalLB 指标参考

MetalLB 故障排查

前提条件

在继续之前，请确保您已阅读 [安装](#) 文档。

通过 Web 控制台配置外部 IP 地址池

1. 进入 管理员。
2. 在左侧导航栏中，单击 网络管理 > 外部 IP 地址池。

- 单击 **创建外部 IP 地址池**。
- 参考以下说明配置某些参数。

参数	说明
类型	- L2：基于 MAC 地址进行通信和转发，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、延迟低的优势。 - BGP (Alpha)：基于 IP 地址进行路由和转发，使用 BGP 协议交换路由信息，适用于需要跨多个自治系统进行复杂路由的大规模网络，具有高扩展性和高可靠性的优势。
IP 资源	支持以 CIDR 和 IP 范围格式输入。单击 添加 可支持多个条目，例如： CIDR: <code>192.168.1.1/24</code> . IP Range: <code>192.168.2.1</code> ~ <code>192.168.2.255</code> .
可用节点	在 L2 模式下，可用节点用于承载所有 VIP 流量；在 BGP 模式下，可用节点用于承载 VIP、与对等方建立 BGP 连接并向外部发布路由。 - 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可用节点。 注意： - 当使用 BGP 类型时，可用节点为下一跳节点；请确保所选可用节点是 BGP 连接节点的子集。 - 您可以单独配置标签选择器或节点名称来选择可用节点；如果两者同时配置，则最终可用节点为两者的交集。
BGP 对等方	选择 BGP 对等方；具体配置请参见 BGP 对等方 。

- 单击 **创建**。

通过 Web 控制台配置 BGP 对等方

1. 进入 管理员。
2. 在左侧导航栏中，单击 网络管理 > **BGP** 对等方。
3. 单击 创建 **BGP** 对等方。
4. 参考以下说明配置参数。

参数	说明
本地 AS 编号	BGP 连接节点所在 AS 的 AS 编号。 注意：如果没有特殊要求，建议使用 IBGP 配置，即本地 AS 编号应与对端 AS 编号保持一致。
对端 AS 编号	BGP 对等方所在 AS 的 AS 编号。
对端 IP	BGP 对等方的 IP 地址，必须是可用于建立 BGP 连接的有效 IP 地址。
本地 IP	BGP 连接节点的 IP 地址。当 BGP 连接节点具有多个 IP 时，请选择指定的本地 IP 与对端建立 BGP 连接。
对端端口	BGP 对等方的端口号。
BGP 连接节点	建立 BGP 连接的节点。如果未配置此参数，则所有节点都将建立 BGP 连接。
eBGP Multi-Hop	允许在未直接连接的 BGP 路由器之间建立 BGP 会话。启用此功能后，BGP 数据包的默认 TTL 值为 5，从而可以在多个中间网络设备之间建立 BGP 对等关系，使网络设计更加灵活。
RouterID	一个 32 位数值（通常以点分十进制格式表示，类似于 IPv4 地址格式），用于在 BGP 网络中唯一标识一个 BGP 路由器，一般用于建立 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. 单击 创建。

注意：当您使用 BGP 模式时，请允许每个 BGP 对等方与 MetalLB speaker 节点之间的 TCP 端口 179 通信。如果您配置了自定义对端端口，请改为允许所配置的端口。

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

```
# 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


```
# 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
```

监控 MetalLB

配置 MetalLB 后，您可以使用内置监控面板和监控指标来观察 IP 地址分配和 BGP 会话状态。

注意：内置 MetalLB 监控面板在 ACP 4.4 中可用。

查看内置的 MetalLB 监控面板

1. 进入 运维中心。

2. 在左侧导航栏中，单击 监控 > 监控面板。
3. 搜索 **MetalLB**，然后打开该监控面板。

内置 MetalLB 监控面板包含以下图表：

图表	说明
已暴露的 Services	当前由 MetalLB 广播的 LoadBalancer Service 数量。
可用 IP	所有地址池中仍可用的 IP 地址数量。
已建立的 BGP 会话	当前已建立的 BGP 会话数量。
已广播的前缀	当前由 MetalLB 广播的 BGP 前缀数量。
地址池使用趋势	所选地址池的 IP 地址使用趋势。
所选地址池容量	所选地址池中已使用和可用的 IP 地址容量。

访问 MetalLB 指标端点

MetalLB 通过 `metallb-system` 命名空间中的 monitor Services 暴露 controller 和 speaker 指标。您可以从集群内部访问这些指标，以确认组件是否正确暴露指标。

检查 MetalLB monitor Services：

```
kubectl get svc -n metallb-system | grep monitor
```

检查 Service 端点：

```
kubectl get endpointslice -n metallb-system -l kubernetes.io/service-name=controller-monitor-service
kubectl get endpointslice -n metallb-system -l kubernetes.io/service-name=speaker-monitor-service
```

端点地址默认使用端口 `17472`。

从集群内部查询 controller 指标端点：

```
curl -s http://<controller-endpoint-address>:17472/metrics | grep '^metal  
lb_'
```

从集群内部查询 speaker 指标端点：

```
curl -s http://<speaker-endpoint-address>:17472/metrics | grep '^metallb  
_'
```

如果命令返回 MetalLB 指标，则说明该组件的指标端点可用。

MetalLB 指标参考

内置 MetalLB 监控面板使用以下指标：

指标	来源组件	说明	使用场景
<code>metallb_speaker_announced</code>	Speaker	表示是否从某个 speaker 节点广播了 LoadBalancer Service。	检查哪些广播了 LoadBalancer Service。
<code>metallb_allocator_addresses_total</code>	Controller	地址池中的 IP 地址总数。	计算地址容量和可数。
<code>metallb_allocator_addresses_in_use_total</code>	Controller	已从地址池分配的 IP 地址数量。	计算已使用和地址池趋势。
<code>metallb_bgp_session_up</code>	Speaker / BGP exporter	表示 BGP 会话是否已建立。	检查 BGP 健康状态
<code>metallb_bgp_announced_prefixes_total</code>	Speaker / BGP exporter	MetalLB 广播的 BGP 前缀数量。	检查路由状态。

MetalLB 故障排查

现象	可能原因	解决方法
未分配外部 IP	没有有效的 IPAddressPool，或地址池配置错误	验证 IP 范围和命名空间
Pods 发生 CrashLoop	Speaker 或 Controller 缺少 RBAC 权限	检查 Operator 权限
BGP 未建立	AS 编号不匹配或对端不可达	检查 <code>BGPPeer</code> 规范和网络路由
L2 不工作	VLAN 错误或 ARP 过滤	使用 <code>arping</code> 验证广播可达性

查看更多 [MetalLB 故障排查](#) ↗

配置 GatewayAPI Gateway

目录

概述

前提条件

Gateway 基础

什么是 Gateway

Gateway 暴露方式 (Service Type)

LoadBalancer (推荐)

NodePort

ClusterIP

监听器配置

端口和协议

AllowRouteNS

TLS 配置

EnvoyProxy (部署配置)

推荐的资源规格

镜像仓库

创建 Gateway

通过 Web Console

监听器配置

通过 YAML

包含多种监听器类型的完整示例

[查看 Gateway 详情](#)

[监听器与 Route 参考](#)

[主机名](#)

[主机名交集规则](#)

[支持的 Route 类型](#)

[下一步](#)

概述

在 Envoy Gateway operator 和 `EnvoyGatewayCtl` 准备就绪后，配置一个 `Gateway`。

`Gateway` 定义流量如何进入 gateway，而配套的 `EnvoyProxy` 则控制底层 Envoy data plane 的部署方式。

在推荐的工作流程中，请在 [Envoy Gateway Operator](#) 之后、[配置 GatewayAPI Route](#) 之前配置 `Gateway`。

前提条件

请确保在继续之前已完成以下步骤：

1. 阅读 [Envoy Gateway Operator](#)，以了解基本概念和资源关系
2. 安装 Envoy Gateway operator 并创建一个 `EnvoyGatewayCtl`

NOTE

在创建 `Gateway` 之前，请先了解主要的 Gateway 概念。如果你已经熟悉这些概念，可以直接跳到 [创建 Gateway](#) 部分。

Gateway 基础

什么是 Gateway

Gateway 是流量进入集群的入口点。它定义外部请求如何接收并路由到后端服务。**Gateway** 主要定义以下内容：

- 监听器：定义 gateway 监听的端口、协议和主机名
- **GatewayClass**：选择由哪个 gateway controller 管理该 **Gateway**
- 基础设施引用：引用一个 **EnvoyProxy**，用于控制底层 Envoy data plane 的部署方式

Gateway 暴露方式 (Service Type)

Service Type 用于配置 gateway 如何通过底层 Envoy Service 暴露。共有三种模式：LoadBalancer、NodePort 和 ClusterIP。

在 YAML 中，此设置配置在配套的 **EnvoyProxy** 资源中，位置为 `.spec.provider.kubernetes.envoyService.type`。

LoadBalancer (推荐)

其优点是易用，并具备高可用的负载均衡能力。要使用 LoadBalancer，集群必须支持 LoadBalancer，可通过 [MetalLB](#) 启用。

使用 MetalLB 时，你可以通过 service 注解指定静态 VIP。在 Web Console 中，请使用 **Service Annotation** 字段：

```
metallb.universe.tf/address-pool: ADDRESS_POOL_NAME
# Or specify a specific IP directly
metallb.universe.tf/loadBalancerIPs: VIP_IP
```

更多详情请参见 [如何在使用 MetalLB 时指定 VIP](#)。

NodePort

其优点是不需要任何外部依赖。

不过，使用 NodePort 有以下缺点：

- 使用 NodePort 时，Kubernetes 分配的 NodePort 端口号与 service 自身端口不同。访问时必须使用 NodePort 端口号，而不是 service 端口。
- 该 service 可通过集群中任意节点 IP 地址访问，可能带来潜在的安全风险。

使用 NodePort 时如何获取正确端口

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

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

输出结果即为 NodePort。

ClusterIP

如果你不需要对外暴露，这种方式非常方便。

监听器配置

监听器定义 gateway 监听的端口和协议。在 HTTP 或 HTTPS 协议中，不同的主机名可以视为不同的监听器。

你不能创建端口、协议或主机名冲突的监听器。

你必须在 **Gateway** 中至少创建一个监听器。

端口和协议

每个监听器都通过端口号和协议进行配置。支持的协议包括：HTTP、HTTPS、TCP、UDP、TLS。

AllowRouteNS

默认情况下，Route 只能附加到 **Same** 命名空间中的 Gateway。若要允许跨命名空间路由，请使用 **Allowed Routes Namespace** 字段：

- **Same**：允许同一命名空间中的 Route 附加到此监听器。
- **All**：允许任意命名空间中的 Route 附加到此监听器。
- **Selector**：允许与选择器匹配的命名空间中的 Route 附加到此监听器。

在 ACP 中，一个项目通过命名空间上的标签来标识，例如 `cpaas.io/project: <project-name>`。如果你希望某个监听器仅供特定项目的 Route 使用，请使用 **Selector** 并匹配目标命名空间上的项目标签。

监听器的 **Allowed Routes Namespace** 设置与其协议共同决定了当你将 route 发布到监听器时，Route Web Console 中哪些监听器可用。

更多信息请参见 [attach to gateway created in other ns](#)。

TLS 配置

对于 HTTPS 和 TLS 协议，你需要配置 TLS 设置。

TLS 模式：

TLS 模式	描述	是否需要证书
Terminate	Envoy 终止 TLS，并在转发到后端服务之前解密流量	是，必须选择 TLS 证书
Passthrough	Envoy 直接将加密的 TLS 流量转发给后端服务，不进行解密	否

NOTE

- HTTPS 协议仅支持 **Terminate** 模式
- TLS 协议同时支持 **Terminate** 和 **Passthrough** 模式
- 采用 **Passthrough** 模式的 TLS 监听器支持 TLSRoute
- 采用 **Terminate** 模式的 TLS 监听器支持 TCPRoute

证书要求：

- 默认情况下，你只能使用在同一命名空间中创建的 secret
- secret 必须是 `kubernetes.io/tls` 类型，并包含 `tls.crt` 和 `tls.key` 键
- 关于跨命名空间 secret，请参见 [use secret created in other ns](#)

EnvoyProxy (部署配置)

Envoy Gateway 使用 `EnvoyProxy` 资源来控制 gateway 的部署配置。我们建议为每个 Gateway 创建一个专用的 `EnvoyProxy` 资源，并通过 Gateway 的 `.spec.infrastructure.parametersRef` 字段进行引用。

当你在 Web Console 中使用由 `EnvoyGatewayCtl` 创建的 `GatewayClass` 创建 `Gateway` 时，控制台会自动创建一个与其同名、同命名空间的配套 `EnvoyProxy` 资源。

当你通过应用 YAML 创建 `Gateway` 时，你需要自行保证 `Gateway` 的 `.spec.infrastructure.parametersRef` 与所引用的 `EnvoyProxy` 资源保持一致。

这种一对一映射方式可以提供更好的隔离性，并对以下部署配置提供更细粒度的控制：

- 从节点数
- 资源限制和请求
- 节点选择器
- service 类型和注解
- 镜像仓库

推荐的资源规格

对于典型的 HTTP 和 HTTPS 南北向流量，可将以下值作为单个 `envoy-proxy` Pod 的起始参考值。这些建议适用于

`.spec.provider.kubernetes.envoyDeployment.container.resources`。

每个 <code>envoy-proxy</code> Pod 的峰值 QPS	推荐 CPU	推荐内存
4,000	2 核	1Gi
8,000	4 核	2Gi

每个 <code>envoy-proxy</code> Pod 的峰值 QPS	推荐 CPU	推荐内存
12,000	8 核	4Gi

NOTE

- 这些值是起始参考值，不是硬性限制。请结合你的实际流量特征进行验证。
- TLS 终止、已启用的策略、路由复杂度、请求和响应大小，以及长连接都可能增加资源消耗。
- 使用多个从节点时，应根据每个从节点实际处理的 QPS 进行扩缩容，而不仅仅是依据 Gateway 的总 QPS。

镜像仓库

镜像仓库已根据你的集群的默认值预先配置。除非必要，请不要修改它。

有关其他部署配置方式，请参见 [deployment-mode](#)。

创建 Gateway

通过 Web Console

1. 导航到 `Alauda Container Platform -> Networking -> Gateway -> Gateways`
2. 点击 `Create Gateway` 按钮
3. 在 `Create Gateway` 页面中，选择由你的 `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`。

该页面显示以下配置项：

字段	描述	YAML 路径
名称	Gateway 名称	gateway: <code>.metadata.name</code> envoyproxy: <code>.metadata.name</code>

字段	描述	YAML 路径
GatewayClass	要使用的 GatewayClass	gateway: <code>.spec.gatewayClassName</code>
Service Type	Service Type	envoyproxy: <code>.spec.provider.kubernetes.envoyService.type</code>
Service Annotation	Service 注解	envoyproxy: <code>.spec.provider.kubernetes.envoyService.annotations</code>
资源限制	部署资源限制	envoyproxy: <code>.spec.provider.kubernetes.envoyDeployment.containerResourceLimits</code>
从节点数	部署从节点数	envoyproxy: <code>.spec.provider.kubernetes.envoyDeployment.replicas</code>
节点标签	部署节点选择器	envoyproxy: <code>.spec.provider.kubernetes.envoyDeployment.nodeSelector</code>
监听器	监听器	gateway: <code>.spec.listeners</code>

WARNING

Web Console 表单仅支持由 `EnvoyGatewayCtl` 创建的 GatewayClass。对于其他 GatewayClass，请使用 YAML 编辑器。

NOTE

当使用 `EnvoyGatewayCtl` 创建的 `GatewayClass` 时，Web Console 会自动创建一个与 Gateway 名称和命名空间匹配的配套 EnvoyProxy 资源。

监听器配置

在创建或编辑监听器时，你可以配置以下内容：

字段	描述	YAML 路径
名称	监听器名称	<code>.spec.listeners[].name</code>

字段	描述	YAML 路径
端口	监听器监听的端口号	<code>.spec.listeners[].port</code>
协议	监听器的协议。 可选项： HTTP、 HTTPS、TCP、 UDP、TLS	<code>.spec.listeners[].protocol</code>
主机名	可选。监听器的主机名	<code>.spec.listeners[].hostname</code>
允许的 Route 命名空间	控制哪些命名空间可以向此监听器附加 route	<code>.spec.listeners[].allowedRoutes.namespaces.from</code>

HTTPS 协议配置

当选择 **HTTPS** 作为协议时：

字段	描述	YAML 路径
证书	必填。选择包含 TLS 证书的 Kubernetes secret。	<code>.spec.listeners[].tls.certificateRefs</code>

NOTE

- HTTPS 协议仅支持 **Terminate** 模式
- HTTPS 监听器必须选择证书
- 默认情况下，你只能使用在同一命名空间中创建的 secret

TLS 协议配置

当选择 **TLS** 作为协议时：

配置字段

字段	描述	YAML 路径
TLS 模式	选择 TLS 模式。可选项： Terminate、Passthrough。 默认值：Terminate	<code>.spec.listeners[].tls.mode</code>
TLS 证书	仅当 TLS 模式为 Terminate 时必填。选择包含 TLS 证书 的 secret。	<code>.spec.listeners[].tls.certificateRefs</code>

有关 TLS 模式的详细信息，请参见 [TLS 配置](#)。

通过 YAML

如果你没有使用推荐的默认示例，请将 `envoy-gateway-operator-cpaas-default` 替换为你自己的 `EnvoyGatewayCtl` 创建的 `GatewayClass`。

下面这个最小示例创建了一个 HTTP `Gateway` 和一个专用 `EnvoyProxy`。

```

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

```

包含多种监听器类型的完整示例

如果你需要更多监听器类型，请使用以下完整示例：


```

apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo
  namespace: demo
spec:
  infrastructure: ❶
    parametersRef:
      group: gateway.envoyproxy.io
      kind: EnvoyProxy
      name: demo
  gatewayClassName: envoy-gateway-operator-cpaas-default ❷
  listeners: ❸
    - name: http
      port: 80
      hostname: a.com ❹
      protocol: HTTP ❺
      allowedRoutes:
        namespaces:
          from: Same ❻
    - name: https
      port: 443
      hostname: a.com
      protocol: HTTPS
      allowedRoutes:
        namespaces:
          from: Same
    - name: tls ❼
      mode: Terminate
      certificateRefs:
        - name: demo-tls
    - name: tls-passthrough
      port: 8443
      hostname: tls.example.com
      protocol: TLS
      allowedRoutes:
        namespaces:
          from: Same
    - name: tls-terminate
      port: 9443
      hostname: secure.example.com

```

```

    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 resource](#) 进行部署配置
- 2 如有需要，将 `envoy-gateway-operator-cpaas-default` 替换为你自己的 `GatewayClass`
- 3 `listeners` 定义流量如何进入 gateway
- 4 `hostname` 会影响基于主机名的路由如何匹配监听器
- 5 `protocol` 决定哪些 route 类型可以附加到监听器
- 6 `allowedRoutes` 控制哪些命名空间可以附加 route
- 7 `tls` 为 HTTPS 和 TLS 监听器配置 TLS 终止或透传
- 8 `EnvoyProxy` 名称必须与 `.spec.infrastructure.parametersRef.name` 匹配
- 9 `envoyService.type` 控制 gateway 的暴露方式
- 10 除非你的环境确实需要，否则请保持 `imageRepository` 不变
- 11 `resources` 配置 Envoy data plane 的资源限制和请求
- 12 将 `provider.type` 保持为 `Kubernetes`

查看 Gateway 详情

在 Gateway 详情页中，监听器列表会显示以下信息：

列	描述
名称	监听器名称
协议	监听器协议（HTTP、HTTPS、TCP、UDP、TLS）
端口	已配置的端口号
NodePort	当 Service Type 为 NodePort 时显示。显示用于访问该监听器的已分配 NodePort 端口号。

NOTE

当 Gateway Service Type 为 **NodePort** 时，监听器列表会额外显示 **NodePort** 列。访问 gateway 时请使用 NodePort 值，而不是 service 端口。更多详情请参见 [使用 NodePort 时如何获取正确端口](#)。

监听器与 Route 参考

将 `Route` 资源附加到 `Gateway` 时，请使用以下规则。

主机名

监听器中的主机名是相同协议下监听器的唯一标识符。你不能在 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，会将其与监听器的主机名和 Route 的主机名做交集匹配。只有交集中的主机名才会用于流量路由。

监听器主机名	Route 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 header
无主机名	有主机名（例如 <code>api.example.com</code> ）	所有 Route 主机名	仅匹配带有 <code>api.example.com</code> 的请求
有主机名（例如 <code>api.example.com</code> ）	无主机名	所有监听器主机名	仅匹配带有 <code>api.example.com</code> 的请求
有主机名（例如 <code>api.example.com</code> ）	有完全匹配的主机名	精确匹配主机名	仅匹配带有 <code>api.example.com</code> 的请求
有通配符（例如 <code>*.example.com</code> ）	有匹配的主机名	匹配具体主机名	匹配 <code>api.example.com</code> 或

监听器主机名	Route 主机名	交集结果	示例
			<code>web.example.com</code> 的请求
有主机名（例如 <code>api.example.com</code> ）	有不匹配的主机名	无交集 - Route 状态异常	Route 无法处理流量

NOTE

通配符（`*`）执行后缀匹配。例如，`*.example.com` 可匹配 `foo.example.com` 和 `bar.example.com`，但不能匹配 `example.com`。

WARNING

无交集意味着 Route 状态会变为异常，并且无法处理流量。

支持的 Route 类型

每个监听器会根据其协议支持不同的 Route 类型：

监听器协议	支持的 Route 类型
HTTP	HTTPRoute, GRPCRoute
HTTPS	HTTPRoute, GRPCRoute
TLS (<code>Passthrough</code> 模式)	TLSRoute
TLS (<code>Terminate</code> 模式)	TCPRoute
TCP	TCPRoute
UDP	UDPRoute

在配置 route 时，请确保它们与所附加监听器的协议相匹配。例如，你不能将 `HTTPRoute` 附加到 `TCP` 监听器。

下一步

在 `Gateway` 准备就绪后，请继续阅读 [配置 GatewayAPI Route](#)。如果在 route 附加后你还需要高级流量控制，请继续阅读 [配置 GatewayAPI Policy](#)。

配置 GatewayAPI Policy

目录

概述

前提条件

Policy Attachment 基础

Policy Attachment 概览

在 Web Console 中创建 Policy

SecurityPolicy

通过 Web Console 配置

API Key Authentication

CORS Configuration

通过 YAML 配置

参考

功能

工作原理

说明

官方文档

BackendTLSPolicy

通过 Web Console 配置

通过 YAML 配置

参考

功能

[说明](#)[官方文档](#)

ClientTrafficPolicy

[通过 Web Console 配置](#)[通过 YAML 配置](#)[参考](#)[功能](#)[说明](#)[官方文档](#)

BackendTrafficPolicy

[通过 Web Console 配置](#)[通过 YAML 配置](#)[参考](#)[功能](#)[说明](#)[官方文档](#)[相关任务](#)

概述

本文档说明在 [Gateway](#) 和 [Route](#) 资源就绪后，如何配置 policy 资源。Policy 通过 `.spec.targetRefs` 使用 Policy Attachment 模式，将额外的流量、安全和后端行为附加到受支持的资源上。

在推荐的工作流中，本文档位于 [配置 GatewayAPI Route](#) 之后。

Envoy Gateway 目前提供四种 policy 类型：[SecurityPolicy](#)、[BackendTLSPolicy](#)、[ClientTrafficPolicy](#) 和 [BackendTrafficPolicy](#)。

前提条件

在继续之前，请确保你已完成以下操作：

- 1. 阅读 [配置 GatewayAPI Gateway](#) 和 [配置 GatewayAPI Route](#)
- 2. 创建 policy 将要附加到的目标资源，例如 `Gateway`、`Route` 或 `Service`

Policy Attachment 基础

Policy 通过 `.spec.targetRefs` 附加到其他资源。

默认情况下，policy 只能附加到同一命名空间中的资源。

对于 `Gateway` 目标，当 policy 类型支持时，可以使用 `sectionName` 指定某个特定的 listener。对于 `Service` 目标，`sectionName` 表示 Service 端口名称。

Policy Attachment 概览

Policy 类型	目的	Web Console 支持	Gateway	HTTPRoute
SecurityPolicy	认证、授权、CORS 和其他安全功能	API Key Auth、CORS	✓ (listener name / <code>ALL</code>)	✓
BackendTLSPolicy	Envoy 与后端服务之间的 TLS 配置	Supported	✗	✗
ClientTrafficPolicy	面向客户端的超时和连接行为	Timeout settings	✓ (listener name / <code>ALL</code>)	✗
BackendTrafficPolicy	后端超时和连接行为	Timeout settings	✓ (listener name / <code>ALL</code>)	✓

`sectionName` 用于指定 `Gateway` 上的特定 listener，或 `Service` 上的特定端口。若省略，或设置为 `ALL`，则该 policy 适用于所有 listener 或端口。

在 Web Console 中创建 Policy

所有 policy 类型都从同一个入口创建：

1. 导航到 `Alauda Container Platform -> Networking -> Gateway -> Policies`
2. 在 `Policy Type` 下拉列表中选择所需值
3. 点击 `Create Policy` 按钮

以下章节仅关注各 policy 类型特有的字段。

SecurityPolicy

通过 Web Console 配置

通用字段（所有 policy 共享）：

字段	描述	YAML 路径
Policy Type	要创建的 policy 类型	<code>.kind</code>
Attach To	该 policy 适用的 Gateway API 资源。支持 Gateway、HTTPRoute 和 GRPCRoute。附加到 Gateway 时，你可以选择性地指定 listener 名称或选择所有 listener。	<code>.spec.targetRefs</code>

SecurityPolicy 特有字段：

字段	描述	YAML 路径
Authorization Type	要使用的认证/授权方法。支持多选：API Key Authentication、CORS Configuration	<code>.spec.apiKeyAuth</code> , <code>.spec.cors</code>

API Key Authentication

字段	描述	YAML 路径
Secrets	包含用于认证的 API key 的 Kubernetes secret	<code>.spec.apiKeyAuth.credentialRefs</code>
Extract From	指定从何处提取 API key (HTTP headers 或查询参数)	<code>.spec.apiKeyAuth.extractFrom</code>

CORS Configuration

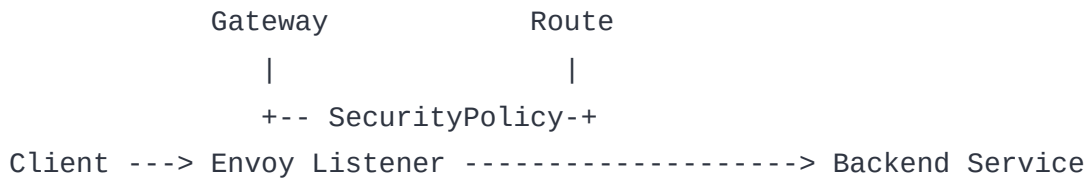
字段	描述	YAML 路径
Allow Origins	允许用于 CORS 请求的 origin 列表	<code>.spec.cors.allowOrigins</code>
Allow Methods	允许的 HTTP method 列表	<code>.spec.cors.allowMethods</code>
Allow Headers	CORS 请求中允许的 header 列表	<code>.spec.cors.allowHeaders</code>
Expose Headers	在响应中向客户端暴露的 header 列表	<code>.spec.cors.exposeHeaders</code>
Max Age	CORS 预检响应的缓存时长	<code>.spec.cors.maxAge</code>
Allow Credentials	是否允许在 CORS 请求中携带凭据	<code>.spec.cors.allowCredentials</code>

通过 YAML 配置

```
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 用于为你的 Gateway 和 Route 配置认证、授权以及其他安全相关功能。它提供了一种声明式方式，可在请求到达后端应用之前验证传入请求，从而保护你的服务。



功能

- 认证：使用多种方法验证客户端身份（API Key、JWT、OIDC、Basic Auth）
- 授权：基于已验证的凭据控制对资源的访问
- **CORS Configuration**：管理跨域资源共享策略

工作原理

1. 使用你需要的认证/授权规则创建 SecurityPolicy
2. 将其附加到特定的 Gateway、HTTPRoute 或 GRPCRoute
3. Envoy Gateway 根据该 policy 验证传入请求
4. 合法请求将转发到后端服务；非法请求将返回相应的 HTTP 状态码并被拒绝

说明

1. Web Console 当前支持配置 **API Key Authentication** 和 **CORS**。对于其他认证方式和高级安全功能，需要使用 YAML 配置。
2. 每个 Route 只能关联一个 SecurityPolicy。
3. 如果 SecurityPolicy 引用了一个没有值的 secret，则所有发送到该附加 route 的请求都会被拒绝，并返回 `401 Unauthorized`。
4. 在 Web Console 中，默认情况下，`Extract From` 字段设置为 `header`，`Header Name` 字段设置为 `authorization`。
5. 你可以通过在 Web Console 中导航到 [Route 的 topology 选项卡](#) 来查看某个 route 关联了哪些 policy。

官方文档

- [SecurityPolicy 规范](#) ↗
- [API Key Authentication](#) ↗

BackendTLSPolicy

通过 Web Console 配置

通用字段：

字段	描述	YAML 路径
Policy Type	要创建的 policy 类型	<code>.kind</code>
Attach To	该 policy 适用的 Service。你必须指定 Service 名称和端口名称（ <code>sectionName</code> ）。	<code>.spec.targetRefs</code>

BackendTLSPolicy 特有字段：

字段	描述	YAML 路径
Hostname	必填。Envoy 连接后端服务时使用的 SNI（Server Name Indication）	<code>.spec.validation.hostname</code>
Subject Alternative Names	可选。用于后端 HTTPS 响应验证。若未指定，则默认使用 hostname 值。	<code>.spec.validation.subjectAltNames</code>
Validation Type	用于验证后端 TLS 证书的方法。 可选项： <code>CACertificateRefs</code> （使用自定义 CA 证书）、 <code>WellKnownCACertificates</code> （使用系统 CA 证书）	<code>.spec.validation</code>

CACertificateRefs 配置：

字段	描述	YAML 路径
CA Certificate Secret	包含 CA 证书的 Kubernetes secret。该 secret 必须具有 <code>ca.crt</code> 键，且其中包含 PEM 编码的 TLS 证书。	<code>.spec.validation.cACertificateRefs</code>

NOTE

在创建或选择 CA 证书 secret 时：

- secret 类型必须适用于 CA 证书
- 键必须为 `ca.crt`
- 你可以导入证书文件，该文件必须以 `-----BEGIN CERTIFICATE-----` 开头，并以 `-----END CERTIFICATE-----` 结尾
- 当导入无效的证书格式时，将显示错误信息 "must contain PEM-encoded TLS certificates"
- 当选择一个不存在 `ca.crt` 键的现有 secret 时，将显示错误信息 "must have ca.crt key"

通过 YAML 配置


```

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。它允许你配置：

```

Service
|
+-- BackendTLSPolicy
Client ---> Envoy Listener -----> Backend Service Port
                                   applies here ^

```

- **SNI (Server Name Indication)** : 建立到后端的 TLS 连接时使用的 hostname
- **Certificate Validation** : 如何验证后端服务器证书
- **CA Certificates** : 用于验证后端证书的自定义 CA 证书

功能

- 为连接到后端服务的连接配置 TLS 设置
- 支持自定义 CA 证书或系统已知 CA 证书
- 提供正确 TLS 握手所需的 SNI 配置

说明

1. `targetRefs` 中的 `sectionName` 对应于 Service 的端口名称。
2. 使用 `WellKnownCACertificates` 时，验证将使用系统默认 CA 证书。
3. `hostname` 为必填项，并在 Envoy 连接后端时用作 SNI 值。

官方文档

- [BackendTLSPolicy 规范](#)

ClientTrafficPolicy

通过 Web Console 配置

通用字段：

字段	描述	YAML 路径
Policy Type	要创建的 policy 类型	<code>.kind</code>
Attach To	该 policy 适用的 Gateway。你可以选择性地指定 listener 名称或选择所有 listener。	<code>.spec.targetRefs</code>

超时配置（选项）：

字段	描述	YAML 路径
TCP Idle Timeout	TCP 连接的空闲超时。空闲时间定义为在上游或下游连接上均未发送	<code>.spec.settings.timeout.tcp.idleTimeout</code>

字段	描述	YAML 路径
	或接收任何字节的一段时间。默认值：1 小时。	
HTTP Request Received Timeout	Envoy 等待完整请求接收的持续时间。该计时器在请求开始时启动，并在请求最后一个字节被发送到上游或响应开始时停止。默认值：1 小时。	<code>.spec.settings.timeout.http.requestReceivedTimeout</code>
HTTP Idle Timeout	HTTP 连接的空闲超时。空闲时间定义为连接中没有活动请求的一段时间。默认值：无限。	<code>.spec.settings.timeout.http.idleTimeout</code>
HTTP Stream Idle Timeout	stream 空闲超时定义为一个 stream 在没有任何上游或下游活动时存在的时长。默认值：5 分钟。	<code>.spec.settings.timeout.http.streamIdleTimeout</code>

通过 YAML 配置

```

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 控制来自客户端到 Envoy Gateway 的连接行为。它提供了对以下内容的细粒度控制：

```

      Gateway
      |
      +--- ClientTrafficPolicy
Client ---> Envoy Listener -----> Backend Service
      applies on the client-facing side ^

```

- **TCP Settings** : 连接级超时和 keepalive 设置
- **HTTP Settings** : 请求/响应超时和 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 stream 空闲超时

说明

1. 超时值以持续时间字符串指定（例如 `"30s"`、`"5m"`、`"1h"`）。

官方文档

- [ClientTrafficPolicy 规范](#)

BackendTrafficPolicy

通过 Web Console 配置

通用字段：

字段	描述	YAML 路径
Policy Type	要创建的 policy 类型	<code>.kind</code>
Attach To	该 policy 适用的 Gateway API 资源。支持 Gateway、HTTPRoute、GRPCRoute、TCPRoute、UDPRoute 和 TLSRoute。附加到 Gateway 时，你可以选择性地指定 listener 名称或选择 <code>ALL</code> listeners。在 YAML 中，这是通过 <code>.spec.targetRefs</code> 中的 <code>sectionName</code> 进行配置的。	<code>.spec.targetRefs</code>

超时配置（选项）：

字段	描述	YAML 路径
TCP Connection Timeout	网络连接建立的超时，包括 TCP 和 TLS 握手。默认值：10 秒。	<code>.spec.settings.timeout.tcp.connectionTimeout</code>

字段	描述	YAML 路径
HTTP Connection Idle Timeout	HTTP 连接的空闲超时。空闲时间定义为连接中没有活动请求的一段时间。默认值：1 小时。	<code>.spec.settings.timeout.http.connectionIdleTimeout</code>
HTTP Max Connection Duration	HTTP 连接的最长持续时间。默认值：无限。	<code>.spec.settings.timeout.http.maxConnectionDuration</code>
HTTP Request Timeout	从上游接收完整响应所需的时间。默认值：15 秒。支持设置为无限。	<code>.spec.settings.timeout.http.requestTimeout</code>

通过 YAML 配置

```

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 控制从 Envoy Gateway 到后端服务的连接行为。它提供了对以下内容的细粒度控制：

```

      Gateway / Route
      |
      +-- BackendTrafficPolicy
Client ---> Envoy Listener -----> Backend Service
                                   applies here ^

```

- **TCP Settings**：连接建立超时
- **HTTP Settings**：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 HTTP 连接最长持续时间

- 配置 HTTP 请求超时

说明

1. 超时值以持续时间字符串指定（例如 `"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

在附加 policy 之后，继续查看 [Envoy Gateway 的任务](#)，了解更多 运维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 Console 配置

创建 HTTPRoute

创建 TCP/UDP Route

创建 GRPCRoute

创建 TLSRoute

通过 YAML 配置

Route 字段参考

发布到监听器

Backend

Hostnames

Rules

HTTPRoute 参考

GRPCRoute Match 和 Filter 参考

TLSRoute 参考

视图

Topology

下一步

概述

本文档说明在 `Gateway` 准备就绪后如何配置 `Route` 资源。`Route` 会附加到一个或多个 gateway 监听器，并定义匹配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 [Configure GatewayAPI Gateway](#) 之后、[Configure GatewayAPI Policy](#) 之前。

除创建和更新操作外，本文档还介绍了 ACP Web Console 提供的额外 route 查看能力。

前提条件

在继续之前，请确保已完成以下步骤：

1. 阅读 [Configure GatewayAPI Gateway](#)，了解监听器、附加规则和 `EnvoyProxy`
2. 创建一个 `Route` 要附加到的 `Gateway`

NOTE

本文档先分别介绍每种 route 类型，然后提供 YAML 示例，最后在共享参考部分解释通用的 route 概念。

配置

`Route` 会附加到 `Gateway` 上的一个或多个监听器。可选择的监听器取决于 route 类型、监听器协议以及监听器允许的 route 命名空间设置。

通过 Web Console 配置

- 1. 导航到 `Alauda Container Platform -> Networking -> Gateway -> Routes`
- 2. 单击 `Create Route` 按钮
- 3. 选择 route 类型 (HTTPRoute、TCPRoute、UDPRoute、GRPCRoute 或 TLSRoute)

创建 HTTPRoute

字段	描述	YAML 路径
发布到监听器	<code>publish to listener</code>	<code>.spec.parentRefs</code>
Hostnames	<code>hostnames</code>	<code>.spec.hostnames</code>
Matches	<code>matches</code>	<code>.spec.rules[].matches</code>
Filters	<code>filters</code>	<code>.spec.rules[].filters</code>
Backend 实例	<code>backend</code>	<code>.spec.rules[].backendRefs</code>
Options	<code>options</code>	<code>.spec.rules[].filters</code> , <code>.spec.rules[].timeouts</code> , <code>.spec.rules[].retry</code> , <code>.spec.rules[].sessionPersistence</code>

Options 配置

Options 字段允许你配置高级流量管理设置：

选项	描述	YAML 路径
会话亲和性	<code>session persistence</code>	<code>.spec.rules[].sessionPersistence</code>
超时	<code>timeout settings</code>	<code>.spec.rules[].timeouts</code>
重试	<code>retry policy</code>	<code>.spec.rules[].retry</code>

创建 TCP/UDP Route

字段	描述	YAML 路径
发布到监听器	publish to listener	<code>.spec.parentRefs</code>
Backend 实例	backend	<code>.spec.rules[].backendRefs</code>

创建 GRPCRoute

字段	描述	YAML 路径
发布到监听器	publish to listener	<code>.spec.parentRefs</code>
Hostnames	hostnames	<code>.spec.hostnames</code>
Matches	grpc matches	<code>.spec.rules[].matches</code>
Filters	grpc filters	<code>.spec.rules[].filters</code>
Backend 实例	backend	<code>.spec.rules[].backendRefs</code>

创建 TLSRoute

字段	描述	YAML 路径
发布到监听器	publish to listener	<code>.spec.parentRefs</code>
Hostnames	hostnames (可选)	<code>.spec.hostnames</code>
Backend 实例	backend	<code>.spec.rules[].backendRefs</code>

通过 YAML 配置

以下最小示例创建一个 `HTTPRoute`，它附加到 `demo` `Gateway` 的 `https` 监听器，并将匹配到的流量转发到后端 `Service`。

```
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
```

如果你需要更多 route 类型和高级 HTTPRoute 选项，请使用以下完整示例：


```

apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: demo-443
  namespace: demo
spec:
  hostnames: ❶
    - example.com
  parentRefs: ❷
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo
      sectionName: https
  rules: ❸
    - backendRefs: ❹
        - group: ''
          kind: Service
          name: echo-resty
          namespace: demo-space
          port: 80
          weight: 100
      filters: [] ❺
      matches: ❻
        - path:
            type: Exact
            value: /a
      timeouts: ❼
        request: '30s'
        backendRequest: '10s'
      retry: ❽
        codes:
          - 503
        attempts: 3
        backoff: '100ms'
      sessionPersistence: ❾
        type: Cookie
        sessionName: a
  ---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TCPRoute
metadata:
  name: tcp
  namespace: demo-space

```

```
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:
```



```

- 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 Hostnames

2 Publish to listener

- 3 Rules
- 4 Backend
- 5 Filters
- 6 Matches
- 7 Options
- 8 Timeout
- 9 Retry
- 10 Session Persistence

Route 字段参考

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

- [HTTPRoute specification](#) ↗
- [TCPRoute specification](#) ↗
- [UDPRoute specification](#) ↗
- [GRPCRoute specification](#) ↗
- [TLSRoute specification](#) ↗

发布到监听器

在 Web Console 中

在 Web Console 中，你可以选择多个监听器来发布 route。可用的监听器候选项会根据以下条件进行过滤：

- **用户权限**：你必须有权访问 gateway 的命名空间（项目必须包含该命名空间）。
- **Route 命名空间 allowlist**：[gateway 监听器允许的 route 命名空间](#)必须包含 route 的命名空间。
- **Route kind 匹配**：route 的 kind（HTTPRoute、GRPCRoute 等）必须与监听器允许的 route kinds 匹配。

对于更复杂的跨命名空间场景，请参阅[附加到在其他命名空间中创建的 gateway](#)。

在 YAML 中

- `sectionName` 是监听器名称。
- Route 只能附加到[支持其特定 kind 的监听器](#)。
- 默认情况下，route 只能附加到 `Gateway` 位于同一命名空间的监听器。

对于跨命名空间附加，请参阅[附加到在其他命名空间中创建的 gateway](#)。

Backend

定义应将匹配请求转发到的目标服务。

每个 service 都可以有一个 `weight` 字段，用于指定路由到该 service 的流量比例。

Hostnames

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

`hostnames` 是一个字符串数组。它遵循 [Hostname Intersection Rules](#)。

Rules

每个 route 可以包含多个规则。每个规则由以下组件组成：

Matches

定义请求必须满足的条件，才能由该规则路由。

一个规则可以有多个 match：

- 每个 match 由多个条件组成（例如 path、headers、query parameters、method）
- match 内部的条件使用 **AND** 逻辑（必须全部满足）
- match 之间使用 **OR** 逻辑（任意一个 match 即可满足规则）

示例：如果 Match-1 要求 `path=/api` AND `header=v1`，而 Match-2 要求 `query=test`，那么当请求满足 `(path=/api AND header=v1)` OR `(query=test)` 时会被路由。

match 结构在不同 route 类型之间是通用的，但支持的 match 条件取决于 route 类型。例如，`HTTPRoute` 和 `GRPCRoute` 支持不同的 match 条件集合。

Filters

指定要应用于请求或响应的转换或修改。

filter 概念在不同 route 类型之间是通用的，但支持的 filter 类型取决于 route 类型。

HTTPRoute 参考

以下 match 条件、filter 类型和高级选项用于 `HTTPRoute`。

Match Condition Types

对象	方法	值类型	描述	值要求
Path	<code>Exact</code>	path (string)	精确匹配 URL path，且区分大小写。这意味着对 <code>/abc</code> 的精确 path 匹配只会匹配到 <code>/abc</code> ，不会匹配 <code>/abc/</code> 、 <code>/Abc</code> 或 <code>/abcd</code> 。	必须以 <code>/</code> 开头能出现连续的 <code>//</code> 。
	<code>PathPrefix</code>	path (string)	基于按 <code>/</code> 分隔的 URL path 前缀进行匹配。匹配区分大小写，并且按逐个 path element 的方式进行。例如， <code>/abc</code> 、 <code>/abc/</code> 和 <code>/abc/def</code> 都会匹配前缀 <code>/abc</code> ，但 <code>/abcd</code> 不会。	必须以 <code>/</code> 开头能出现连续的 <code>//</code> 。

对象	方法	值类型	描述	值要求
Header	RegularExpression	path (string)	正则引擎： RE2。	例如， <code>/api/v1/</code>
	Exact	name (header key) + value	精确匹配 header 值。	
	RegularExpression	name (header key) + value	正则引擎： RE2。	
QueryParam	Exact	name (param key) + value	精确匹配 query parameter 值。	参数值：1-102 字符
	RegularExpression	name (param key) + value	正则引擎： RE2。	
	-	method name	HTTP method 匹配。	<code>GET</code> 、 <code>HEAD</code> <code>POST</code> 、 <code>PUT</code> <code>DELETE</code> 、 <code>CONNECT</code> 、 <code>OPTIONS</code> 、 <code>TRACE</code> 、 <code>PAT</code>

Match Condition References

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

Filter Types

类型	方法	值类型	描述
RequestHeaderModifier	Set	name (string) + value (string)	使用给定的名称和值覆盖 request header
	Add	name (string) + value (string)	向 request 添加 header，并追加到现有值
	Remove	[]string	从 request 中移除指定的 headers（不区分大小写）
ResponseHeaderModifier	Set	name (string) + value (string)	使用给定的名称和值覆盖 response header
	Add	name (string) + value (string)	向 response 添加 header，并追加到现有值
	Remove	[]string	从 response 中移除指定的 headers（不区分大小写）
RequestRedirect	Scheme	string	Location header 的 scheme (http/https)
	Hostname	PreciseHostname	Location header 的 hostname
	ReplaceFullPath	string	替换整个 request path

类型	方法	值类型	描述
	ReplacePrefixMatch	string	替换匹配到的 path 前缀
	Port	PortNumber	Location header 的端口
	StatusCode	int	HTTP 重定向状态码
URLRewrite	Hostname	PreciseHostname	要在 request 中重写的 hostname
	ReplaceFullPath	string	替换整个 request path
	ReplacePrefixMatch	string	替换匹配到的 path 前缀
CORS	AllowOrigins	[]string	CORS 请求允许的 origin 列表
	AllowMethods	[]HTTPMethod	允许的 HTTP methods 列表
	AllowHeaders	[]string	CORS 请求允许的 headers 列表

类型	方法	值类型	描述
	ExposeHeaders	[]string	在响应中暴露给客户端的 headers 列表
	MaxAge	Duration	CORS 预检响应的缓存时长
	AllowCredentials	bool	是否允许 CORS 请求携带凭据

注意：

- RequestRedirect 和 URLRewrite 不能在同一个 rule 中一起使用
- ReplacePrefixMatch 仅与 PathPrefix HTTPRouteMatch 兼容
- 根据 RFC 7230，header 名称不区分大小写
- 同一 header 的多个值必须使用 RFC 7230 以逗号分隔的格式

Filter References

Filter 类型	官方文档
RequestHeaderModifier	HTTPHeaderFilter ↗
ResponseHeaderModifier	HTTPHeaderFilter ↗
RequestRedirect	HTTPRequestRedirectFilter ↗
URLRewrite	HTTPURLRewriteFilter ↗
CORS	HTTPCORSFilter ↗
RequestMirror	HTTPRequestMirrorFilter ↗
HTTPExternalAuthFilter	HTTPExternalAuthFilter ↗

Options

Options 部分为 `HTTPRoute` 提供高级流量管理能力，包括超时、重试和会话保持设置。

Timeouts

字段	描述	YAML 路径
Request Timeout	gateway 在从客户端接收完整 request 后，完成 HTTP response 的最长持续时间。选项：Default（使用默认超时，通常为 15 秒）、Unlimited（设置为 "0s" 以移除超时）、Custom。	<code>.spec.rules[].timeouts.request</code>
Backend Request Timeout	单次 gateway 到 backend 调用的最长持续时间，即从 gateway 开始发送 request 到收到完整 backend response 为止。选项：Default（使用实现特定的默认值）、Unlimited（设置为 "0s"）、Custom。	<code>.spec.rules[].timeouts.backendRequest</code>

NOTE

- Request Timeout 在收到完整 client request 后开始计时，并覆盖完整事务；如果发生重试，可能包含多次 backend 调用。
- 在指定时，Backend Request Timeout 必须小于或等于 Request Timeout。
- 选择 "Default" 时，该字段会被设置为 nil（使用实现默认值）。
- 选择 "Unlimited" 时，该字段会被设置为 "0s"（可达到的最大值）。

字段	规范
<code>.spec.rules[].timeouts</code>	HTTPRouteTimeouts

Retry

字段	描述	YAML 路径
Status Codes	触发重试的 HTTP 状态码（例如 503、502）。取值范围：400-599。	<code>.spec.rules[].retry.codes</code>
Attempts	重试次数。	<code>.spec.rules[].retry.attempts</code>
Backoff	重试前的等待时间（例如 "100ms"、"1s"）。	<code>.spec.rules[].retry.backoff</code>

NOTE

- 默认情况下，重试是禁用的。如果未配置 `retry` 字段或其为空，gateway 将不会重试任何失败的请求。
- 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 Console 中配置重试时，如果你移除了所有重试配置项，该字段会被设置为 `nil`。

字段	规范
<code>.spec.rules[].retry</code>	HTTPRouteRetry

Session Persistence

配置会话亲和性设置，以确保来自同一客户端的请求路由到同一个后端。

字段	描述	YAML 路径
Type	会话保持类型。选项：Cookie、Header。	<code>.spec.rules[].sessionPersistence.type</code>
Session Name	用于会话跟踪的 cookie 或 header 名称。	<code>.spec.rules[].sessionPersistence.sessionName</code>

字段	规范
<code>.spec.rules[].sessionPersistence</code>	SessionPersistence ↗

GRPCRoute Match 和 Filter 参考

以下 match 条件和 filter 类型用于 `GRPCRoute`。

GRPCRoute Matches

`GRPCRoute` 支持以下 match 类型：

对象	方法	值类型	描述
Method	-	type (service/method) + value	匹配 gRPC method。Type 可以是 <code>service</code> （匹配 service 名称）或 <code>method</code> （匹配 method 名称）。
Headers	<code>Exact</code>	name (header key) + value	精确匹配 header 值。
	<code>RegularExpression</code>	name (header key) + value	正则引擎：RE2。

GRPCRoute Filters

`GRPCRoute` 仅支持 `RequestHeaderModifier` filter：

类型	方法	值类型	描述	值要求
RequestHeaderModifier	<code>Set</code>	name (string) + value (string)	使用给定的名称和值覆盖 request header	最多 16 项，值：1-4096 字符
	<code>Add</code>	name (string) + value (string)	向 request 添加 header，并追加到现有值	最多 16 项，值：1-4096 字符

类型	方法	值类型	描述	值要求
	Remove	[]string	从 request 中移除指定的 headers（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 Options。

TLSRoute 参考

以下行为是 [TLSRoute](#) 特有的。

NOTE

- [TLSRoute](#) 的 hostnames 是可选的。如果监听器有 hostname，但 [TLSRoute](#) 没有，则 [TLSRoute](#) 会自动继承监听器的 hostname。
- [TLSRoute](#) 只能附加到处于 [Passthrough](#) 模式的 [TLS](#) 协议监听器。

视图

Topology

以下功能是 ACP Web Console 提供的额外查看能力。

Topology 选项卡以可视化方式展示 route 及其关联资源。它会显示附加到该 route 的所有 policy，以及它们的依赖资源，例如被 [SecurityPolicy](#) 引用的 secrets。

此功能当前仅适用于 [HTTPRoute](#)。

下一步

在 route 已附加到监听器后，如果你需要高级流量或安全策略，请继续阅读 [Configure GatewayAPI Policy](#)。更多操作示例请参阅 [Tasks for Envoy Gateway](#)。

相关任务

- [How to attach to listener created in other namespace](#)

配置 ALB

WARNING

ALB 已废弃。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

可调整配置

ALB 上的操作

创建

更新

删除

Frontend

前提条件

配置 Frontend

Frontend 上的操作

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

priority

动作

后端

backend protocol

Service Group 和 Session Affinity Policy

Rule 上的操作

使用 Web 控制台

使用 CLI

Https

Ingress 中的证书 Annotation

Rule 中的证书

TLS 模式

Ingress

Ingress 同步

日志和监控

查看日志

监控指标

ALB

ALB 是一个表示负载均衡器的自定义资源。默认嵌入到所有集群中的 alb-operator 会监听对 ALB 资源的创建、更新和删除操作，并据此创建相应的 Deployment 和 Service。

对于每个 ALB，都有一个对应的 Deployment，它会监听附加到该 ALB 的所有 Frontend 和 Rule，并根据这些配置将请求路由到后端。

前提条件

负载均衡器 的高可用需要一个 VIP。请参阅 [配置 VIP](#)。

配置 ALB

ALB 配置由三部分组成。


```
# 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

资源相关字段描述了 alb 的部署配置。

字段	类型	描述
<code>.spec.config.nodeSelector</code>	<code>map[string]string</code>	alb 的节点选择器

字段	类型	描述
<code>.spec.config.replicas</code>	int, optional default 3	alb 的副本数
<code>.spec.config.resources.limits</code>	k8s container-resource, optional	alb 的 nginx 容器限制
<code>.spec.config.resources.requests</code>	k8s container-resource, optional	alb 的 nginx 容器请求
<code>.spec.config.resources.alb.limits</code>	k8s container-resource, optional	alb 的 alb 容器限制
<code>.spec.config.resources.alb.requests</code>	k8s container-resource, optional	alb 的 alb 容器请求
<code>.spec.config.antiAffinityKey</code>	string, optional default local	k8s antiAffinityKey

网络配置

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

字段	类型	描述
<code>.spec.config.networkMode</code>	string: <code>host</code> or <code>container</code> , optional, default <code>host</code>	在 <code>container</code> 模式下，operator 会创建一个 LoadBalancer Service，并使用其地址作为 ALB 地址。
<code>.spec.address</code>	string, required	你可以手动指定 alb 的地址
<code>.spec.config.vip.enableLbSvc</code>	bool, optional	在 <code>container</code> 模式下会自动设为 true。
<code>.spec.config.vip.lbSvcAnnotations</code>	map[string]string, optional	LoadBalancer Service 的额外 annotations。

项目配置

字段	类型
<code>.spec.config.projects</code>	<code>[]string,required</code>
<code>.spec.config.portProjects</code>	<code>string,optional</code>
<code>.spec.config.enablePortProject</code>	<code>bool,optional</code>

将 ALB 添加到某个项目中意味着：

1. 在 Web UI 中，只有该项目内的用户才能找到并配置这个 ALB。
2. 该 ALB 将处理属于该项目的 ingress 资源。请参阅 [ingress-sync](#)。
3. 在 Web UI 中，在项目 X 中创建的规则无法在项目 Y 下被找到或配置。

如果你启用了端口项目并为项目分配了一个端口范围，这意味着：

1. 你不能创建不属于该项目分配端口范围的端口。

可调整配置

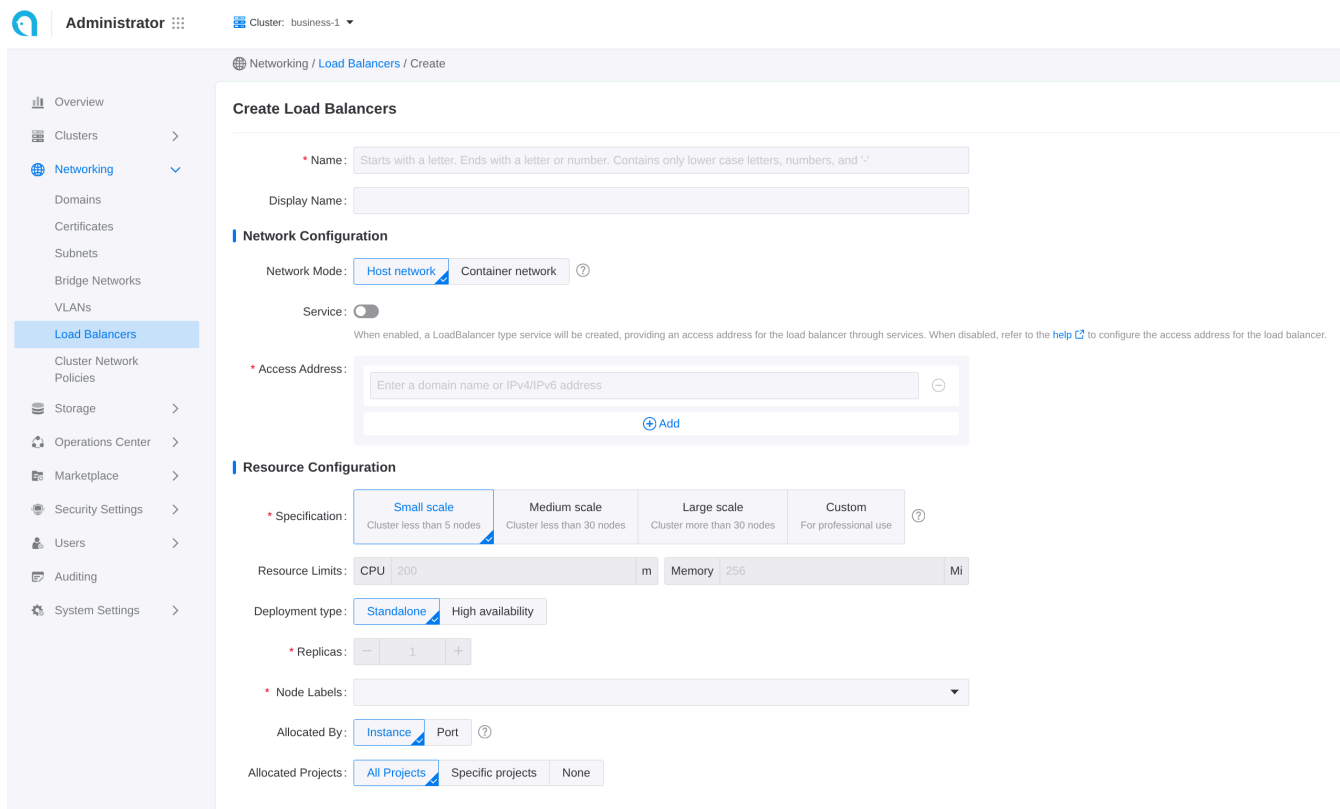
在 alb cr 中有一些全局配置可以调整。

- [绑定 NIC](#)
- [Ingress 同步](#)

ALB 上的操作

创建

使用 **Web** 控制台



Create Load Balancers

* Name: Starts with a letter. Ends with a letter or number. Contains only lower case letters, numbers, and '-'

Display Name:

Network Configuration

Network Mode: **Host network** Container network ?

Service: ☐ When enabled, a LoadBalancer type service will be created, providing an access address for the load balancer through services. When disabled, refer to the [help](#) to configure the access address for the load balancer.

* Access Address: Enter a domain name or IPv4/IPv6 address [Add](#)

Resource Configuration

* Specification: **Small scale** Medium scale Large scale Custom ?
Cluster less than 5 nodes Cluster less than 30 nodes Cluster more than 30 nodes For professional use

Resource Limits: CPU 200 m Memory 256 Mi

Deployment type: **Standalone** High availability

* Replicas: - 1 +

* Node Labels:

Allocated By: **Instance** Port ?

Allocated Projects: **All Projects** Specific projects None

Web UI 中公开了一些常见配置。按照以下步骤创建负载均衡器：

1. 导航到 管理员。
2. 在左侧边栏中，点击 网络管理 > 负载均衡器。
3. 点击 创建负载均衡器。

Web UI 中的每个输入项都对应 CR 的一个字段：

参数	描述
Assigned Address	<code>.spec.address</code>
Allocated By	<code>Instance</code> 表示项目模式，你可以在下方选择项目； <code>port</code> 表示端口项目模式，你可以在创建 alb 后分配端口范围。

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。请为此操作选择合适的时间！

1. 进入 管理员。
2. 在左侧导航栏中，点击 网络管理 > 负载均衡器。
3. 点击 ⋮ > 更新。
4. 按需更新网络和资源配置。
 - 请根据业务需求合理设置规格。你也可以参考相关的 [如何合理分配 CPU 和内存资源](#) 作为指导。
 - 内部路由 仅支持从 **Disabled** 状态更新为 **Enabled** 状态。
5. 点击 更新。

删除

使用 Web 控制台

NOTE

删除负载均衡器后，关联的端口和规则也会被删除，且无法恢复。

1. 进入 管理员。
2. 在左侧导航栏中，点击 网络管理 > 负载均衡器。
3. 点击 ⋮ > 删除，并确认。

使用 CLI

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

Frontend

Frontend 是一个定义 ALB 监听端口和协议的自定义资源。

支持的协议：L7（http|https|grpc|grpcs）和 L4（tcp|udp）。

- 在 L4 Proxy 中，使用 Frontend 直接配置后端 Service。
- 在 L7 Proxy 中，使用 Frontend 配置监听端口，并使用 [rule](#) 配置后端 Service。

如果你需要添加 HTTPS 监听端口，还应联系管理员，为当前项目分配一个 TLS 证书用于加密。

前提条件

先创建一个 ALB。

配置 Frontend

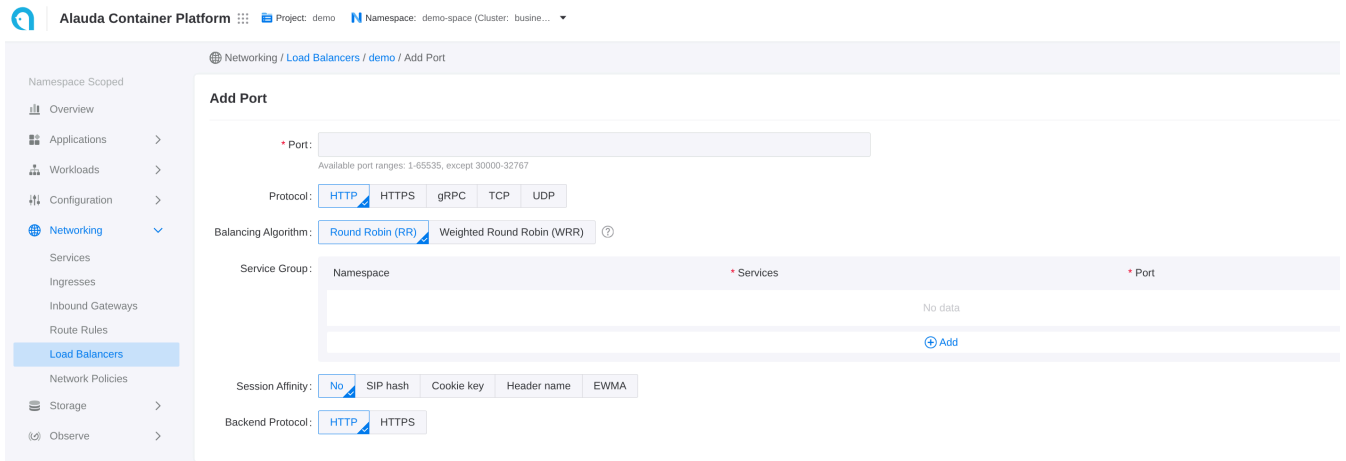
```
# alb-frontend-demo.yaml
apiVersion: crd.alauda.io/v1
kind: Frontend
metadata:
  labels:
    alb2.cpaas.io/name: alb-demo ❶
  name: alb-demo-00080 ❷
  namespace: cpaas-system
spec:
  port: 80 ❸
  protocol: http ❹
  certificate_name: '' ❺
  backendProtocol: 'http' ❻
  serviceGroup: ❼
    session_affinity_policy: '' ❽
  services:
    - name: hello-world
      namespace: default
      port: 80
      weight: 100
```

- 1 alb label : 必填, 表示该 `Frontend` 所属的 ALB 实例。
- 2 frontend name : 格式为 `$alb_name-$port` 。
- 3 port : 监听的端口。
- 4 protocol : 该端口使用的协议。
 - L7 协议为 https|http|grpcs|grpc , L4 协议为 tcp|udp。
 - 选择 HTTPS 时, 必须添加证书; gRPC 协议可选添加证书。
 - 选择 gRPC 协议时, 后端协议默认也是 gRPC, 不支持会话保持。如果为 gRPC 协议设置了证书, 负载均衡器将卸载 gRPC 证书, 并将未加密的 gRPC 流量转发到后端 Service。
 - 如果使用 Google GKE 集群, 同一 **container network type** 的负载均衡器不能同时具有 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 Service 使用的协议。
- 7 默认 `serviceGroup` :
 - L4 proxy : 必填。ALB 直接将流量转发到默认 service group。
 - L7 proxy : 可选。ALB 会先匹配该 Frontend 上的 Rules ; 如果都不匹配, 则回退到默认的 `serviceGroup` 。
- 8 `session_affinity_policy`

Frontend 上的操作

创建

使用 **Web** 控制台



1. 前往 容器平台。
2. 在左侧导航栏中，点击 网络 > 负载均衡。
3. 点击负载均衡器名称，进入详情页。
4. 点击 添加端口。

Web UI 中的每个输入项都对应 CR 的一个字段

参数	描述
Session Affinity	<code>.spec.serviceGroup.session_affinity_policy</code>

使用 CLI

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

后续操作

对于来自 HTTP、gRPC 和 HTTPS 端口的流量，除了默认的内部路由组外，你还可以设置更多不同的后端 Service 匹配 [规则](#)。负载均衡器会先根据已设置的规则匹配对应的后端 Service；如果规则匹配失败，则会继续匹配前述内部路由组对应的后端 Service。

相关操作

你可以点击列表页右侧的  图标，或在详情页右上角点击 操作，按需更新默认路由或删除监听端口。

NOTE

如果负载均衡器的资源分配方式是 **Port**，则只有管理员可以在 管理员 视图中删除相关监听端口。

Rule

Rule 是一个定义 ALB 如何匹配和处理传入请求的 Custom Resource（CR）。

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 Rule，帮助你快速了解如何使用规则。

NOTE

rule 必须通过 label 附加到 frontend 和 alb 上。

```

apiVersion: crd.alauda.io/v1
kind: Rule
metadata:
  labels:
    alb2.cpaas.io/frontend: alb-demo-00080 ❶
    alb2.cpaas.io/name: alb-demo ❷
  name: alb-demo-00080-test
  namespace: cpaas-system
spec:
  backendProtocol: 'https' ❸
  certificate_name: 'a/b' ❹
  dslx: ❺
  - type: URL
    values:
      - - STARTS_WITH
      - /
  priority: 4 ❻
  serviceGroup: ❷
  services:
    - name: hello-world
      namespace: default
      port: 80
      weight: 100

```

- ❶ 必填，表示该 Rule 所属的 `Frontend`。
- ❷ 必填，表示该 Rule 所属的 ALB。
- ❸ `backendProtocol`
- ❹ `certificate_name`
- ❺ `dslx`
- ❻ 数值越小，优先级越高。
- ❼ `serviceGroup`

使用 **dslx** 和优先级匹配请求

dslx

DSLX 是一种用于描述匹配条件的领域特定语言。

例如，下面的规则会匹配满足以下全部条件的请求：

- url 以 /app-a 或 /app-b 开头
- method 是 post
- url param 的 group 为 vip
- host 是 *.app.com
- header 的 location 是 east-1 或 east-2
- cookie 名称为 uid
- source IP 来自 1.1.1.1-1.1.1.100

```

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'

```

priority

Priority 是一个范围为 0 到 10 的整数，数值越小表示优先级越高。要在 ingress 中配置规则的优先级，可以使用以下 annotation 格式：

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

对于规则，只需直接在 `.spec.priority` 中设置一个整数值即可。

动作

当请求匹配到某条规则后，你可以对该请求应用以下动作：

功能	描述	链接
Timeout	配置请求的超时设置。	timeout
Redirect	将传入请求重定向到指定 URL。	redirect
CORS	为应用启用跨域资源共享（CORS）。	cors
Header Modification	允许修改请求或响应头。	header modification
URL Rewrite	在转发前重写传入请求的 URL。	url-rewrite
WAF	集成 Web Application Firewall（WAF）以增强安全性。	waf
OTEL	启用 OpenTelemetry（OTEL）用于分布式追踪和监控。	otel
Keepalive	启用或禁用应用的 keepalive 功能。	keepalive

后端

backend protocol

默认情况下，后端协议设置为 HTTP。如果你想使用 TLS 重新加密，可以将其配置为 HTTPS。

Service Group 和 Session Affinity Policy

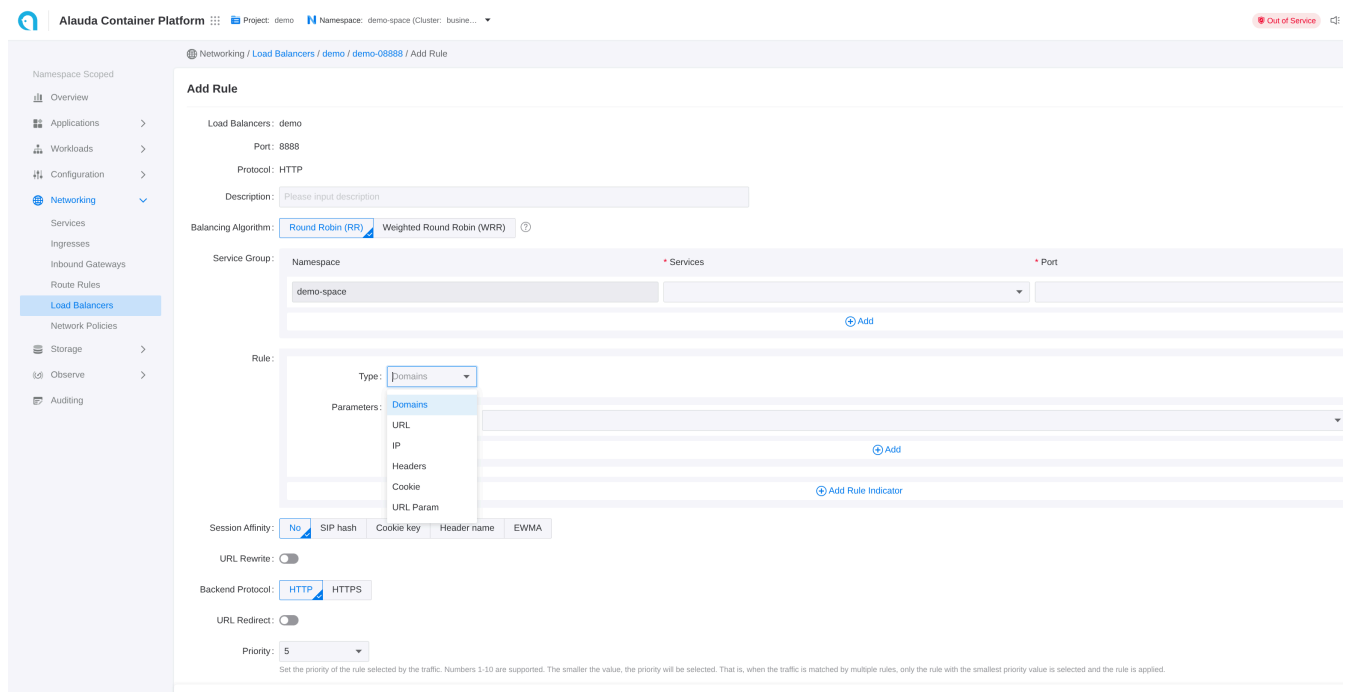
你可以在一条规则中配置一个或多个 Service。

默认情况下，ALB 使用轮询（RR）算法在后端 Service 之间分发请求。不过，你可以为单个 Service 分配权重，或选择其他负载均衡算法。

有关更多详情，请参阅 [负载均衡算法](#)。

Rule 上的操作

使用 Web 控制台



1. 进入 容器平台。
2. 在左侧导航栏中，点击 网络 > 负载均衡。
3. 点击负载均衡器名称。
4. 选择监听端口名称。
5. 点击 添加规则。
6. 参考以下说明配置相关参数。
7. 点击 添加。

Web UI 中的每个输入项都对应 CR 的一个字段

使用 CLI

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

Https

如果 frontend (ft) 协议是 HTTPS 或 GRPCS，规则也可以配置为使用 HTTPS。

你可以在规则或 ingress 中指定证书，以匹配该特定端口的证书。

支持终止，并且如果后端协议是 HTTPS，则可以进行重新加密。但是，你不能为与后端 Service 的通信指定证书。

Ingress 中的证书 Annotation

可以通过 annotation 跨 namespace 引用证书。

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

在 edge 模式下，客户端通过 HTTPS 与 ALB 通信，而 ALB 使用 HTTP 协议与后端 Service 通信。

要实现这一点：

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

Re-encrypt 模式

在 re-encrypt 模式下，客户端通过 HTTPS 与 ALB 通信，而 ALB 使用 HTTPS 协议与后端 Service 通信。

要实现这一点：

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

Ingress

Ingress 同步

每个 ALB 都会创建一个同名的 IngressClass，并处理同一项目内的 ingress。

当 ingress namespace 带有类似 `cpaas.io/project: demo` 的 label 时，表示该 ingress 属于 `demo` 项目。

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

NOTE

ALB 会监听 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` 是 ingress 名称。
3. `spec.source.namespace` 是 ingress namespace。

SSL 策略

对于未配置证书的 ingress，ALB 提供了使用默认证书的策略。

你可以使用以下设置配置 ALB 自定义资源：

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

可用的 SSL 策略：

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

日志和监控

结合日志和监控数据，你可以快速识别并解决负载均衡器问题。

查看日志

1. 进入 管理员。
2. 在左侧导航栏中，点击 网络管理 > 负载均衡器。
3. 点击 负载均衡器名称。
4. 在 日志 选项卡中，从容器的视角查看负载均衡器运行时日志。

监控指标

NOTE

负载均衡器所在的集群必须部署监控服务。

1. 进入 管理员。
2. 在左侧导航栏中，点击 网络管理 > 负载均衡器。
3. 点击 负载均衡器名称。
4. 在 监控 选项卡中，从节点的视角查看负载均衡器的指标趋势信息。
 - **Usage Rate**：当前节点上负载均衡器对 CPU 和内存的实时使用率。
 - **Throughput**：负载均衡器实例的整体进出流量。

有关监控指标的更多详细信息，请参阅 [ALB 监控](#)。

配置 NodeLocal DNSCache

目录

概述

主要特性

重要说明

安装

通过 Marketplace 安装

工作原理

架构

配置

网络策略配置

概述

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。该插件会在每个节点本地缓存 DNS 响应，从而减少 DNS 查询延迟，并降低中心 DNS 服务的负载，提升集群稳定性。

主要特性

- 本地 **DNS** 缓存：在每个节点本地缓存 DNS 响应，以降低查询延迟
- 性能提升：显著缩短应用程序的 DNS 查询时间

重要说明

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。如果部署，可能会导致 DNS 查询失败。
2. **kubelet** 重启：部署该插件会导致 kubelet 重启。
3. 需要重启 **Pod**：插件成功部署后，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，需要手动将参数 "--node-local-dns-ip=(本地 DNS 缓存服务器的 IP 地址)" 添加到 kube-ovn-controller 中。
4. **NetworkPolicy** 配置：如果集群中配置了 NetworkPolicy，需要在 networkPolicy 中额外允许 node CIDR 和 nodeLocalDNSIP 的双向流量，确保通信正常。
5. 通过重建进行集群升级：如果集群通过重建节点（重新部署）进行升级，kubelet 配置变更将会丢失。为了使 NodeLocal DNS 配置在升级后保持持久化，需要在集群模板中的以下三个位置将 `--cluster-dns` 参数添加到 `kubeletExtraArgs` 中：

- KubeadmControlPlane → initConfiguration → nodeRegistration → kubeletExtraArgs
- KubeadmControlPlane → joinConfiguration → nodeRegistration → kubeletExtraArgs
- KubeadmConfigTemplate → template → spec → joinConfiguration → nodeRegistration → kubeletExtraArgs

在上述每个 `kubeletExtraArgs` 部分中添加以下参数：

```
cluster-dns: "<NodeLocal_DNS_IP>" # e.g., 169.254.20.10
```

WARNING

4.2.x 升级说明

当将该插件从低于 4.2.0 的版本（不包括 4.2.0 本身）升级到 4.2.x 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：

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

注意：该兼容性问题已在 4.3 及以上版本中解决，因此升级到 4.3+ 时无需手动干预。

安装

通过 Marketplace 安装

1. 导航到 **Administrator > Marketplace > Cluster Plugins**。
2. 在插件列表中搜索 "**Alauda Build of NodeLocal DNSCache**"。
3. 点击 **Install** 打开安装配置页面。
4. 配置所需参数：

参数	描述	示例值
IP	node local DNS 缓存服务器的 IP 地址。对于 IPv4，建议使用 169.254.0.0/16 范围内的地址，优先推荐 169.254.20.10。对于 IPv6，建议使用 fd00::/8 范围内的地址，优先推荐 fd00::10。	169.254.20.10

5. 查看部署说明，并确保您的环境满足要求。
6. 点击 **Install** 完成安装。

7. 等待插件状态变为 **"Ready"**。

工作原理

架构

```
Pod → NodeLocal DNSCache → [Cache Hit] → Pod
      ↓
      [Cache Miss] → CoreDNS → Response → Cache & Pod
```

配置

网络策略配置

重要：如果集群启用了 NetworkPolicy，必须配置正确的规则，以允许 DNS 流量访问 NodeLocal DNSCache。没有这些规则，Pod 可能无法解析 DNS 查询。

使用 NetworkPolicy 时，请确保允许以下 DNS 流量：

```
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 address
      ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
  egress:
    - to:
        - ipBlock:
            cidr: 169.254.20.10/32 # NodeLocal DNS IP address
      ports:
        - protocol: UDP
          port: 53
        - protocol: TCP
          port: 53
```

配置 CoreDNS

目录

概述

配置

主机别名

节点选择器

节点容忍度

概述

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。

配置

1. 导航到 **Administrator > Marketplace > Cluster Plugins**。
2. 搜索 "**Alauda Build of CoreDNS**" 并单击 **Update**。
3. 配置以下参数：

主机别名

配置自定义 DNS 解析条目。

参数	说明
IP	要解析的 IP 地址
Domains	域名（以空格分隔） 示例： <code>example.com test.example.com</code>

节点选择器

指定 CoreDNS Pod 应运行在哪些节点上。

参数	说明
Label Key	要在节点上匹配的标签键
Label Value	要在节点上匹配的标签值

节点容忍度

允许将 CoreDNS Pod 调度到带有 taint 的节点上。

参数	说明
Key	要容忍的 taint 键
Value	taint 值（可选）
Type	<code>NoSchedule</code> 、 <code>PreferNoSchedule</code> 或 <code>NoExecute</code>

4. 单击 **Update** 以应用配置。

网络可观测性

网络可观测性提供集群范围的持续网络流量收集，以及按需进行数据包或流的捕获以便排查问题。它使用 eBPF 来检查流量行为、识别性能问题，并调查工作负载之间的连接性问题。

Note

因为 Alauda Container Platform Network Observability Operator 的发版周期与灵雀云容器平台不同，所以 Alauda Container Platform Network Observability Operator 的文档现在作为独立的文档站点托管在 [Alauda Container Platform Network Observability Operator](#)。

实用指南

Ingress-Nginx 的任务

Ingress-Nginx 的任务

前置条件

最大连接数

请求超时

会话亲和性 (Sticky Sessions)

Header 修改

URL 重写

HSTS (HTTP Strict Transport Security)

限流

WAF

转发 Header 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 解决方案（Alpha）

软数据中心 LB 解决方案（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 和智能体
- 监控、运维工具和扩展组件

准备 Kube-OVN Underlay 物理网络 Kube-OVN 安装

- 使用说明
- 术语说明
- 环境要求
- 配置示例
- 概述
- 前提条件
- 配置步骤

集群互联（Alpha）

支持配置使用 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

配置 Kube-OVN 网络以支持 Pod 多网络接口

- 开始之前
- 安装 Multus CNI

在 global 集群中部署集群互联控制器

加入集群互联

相关操作

创建辅助网络

创建具有多个网络接口的 Pod

验证网络接口

可选配置

配置 IPPool

说明

注意事项

配置 MTU

默认 MTU 行为

自定义 MTU 设置

配置 BGP

Underlay 和 Overlay 子网的自动互联

操作步骤

启用 u2oInterconnection 后的 Underlay 子网隔离

配置 Endpoint Health Checker

配置 Endpoint Health Checker

概述

主要功能

安装

工作原理

如何启用

卸载

alb

ALB 任务

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

简介

前提条件

基本 HTTP Route

Route 超时

HTTP Strict Transport Security (HSTS)

基于 Cookie 的会话亲和性

基于路径的路由

Header 修改

连接限制

速率限制

IP Allowlist/Blocklist

URL 重写

跨命名空间 Route 接纳

Ingress 的默认 TLS 证书

使用自定义 CA 的 TLS Re-encrypt

使用自定义证书的 Edge Termination

TLS Passthrough

功能对比摘要

[迁移策略](#)

[相关文档](#)

Ingress-Nginx 的任务

目录

前置条件

最大连接数

请求超时

会话亲和性 (Sticky Sessions)

Header 修改

URL 重写

HSTS (HTTP Strict Transport Security)

限流

WAF

转发 Header 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

通过带有 externalTrafficPolicy=Local 的 MetalLB

工作原理

配置方法

前置条件

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（Sticky Sessions）

[配置 Sticky Sessions](#) ↗

Header 修改

action	link
在请求中设置 header	proxy-set-header ↗
在请求中移除 header	在请求中设置一个空 header
在响应中设置 header	带有 more-set-header ↗ 指令的 configuration-snippets ↗

action	link
在响应中移除 header	hide-headers ↗

URL 重写

[rewrite ↗](#)

HSTS (HTTP Strict Transport Security)

[配置 HSTS ↗](#)

限流

[配置限流 ↗](#)

WAF

[modsecurity ↗](#)

转发 Header 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

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

请参阅 [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，由于 NAT（Network Address Translation，网络地址转换），原始客户端 IP 地址可能会丢失。保留源 IP 对以下场景很重要：

- 访问控制和安全策略
- 准确的日志记录和分析
- 按客户端进行限流

- 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 header 来携带原始源 IP 和端口。

流量流程：

1. 客户端连接到 HAProxy 负载均衡器
2. HAProxy 在连接前添加包含原始客户端 IP 的 PROXY protocol header
3. Ingress-Nginx 接收连接并解析 PROXY protocol header
4. Ingress-Nginx 从 header 中提取真实客户端 IP
5. 后端应用在 `X-Forwarded-For` 和 `X-Real-IP` headers 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（HAProxy、AWS NLB 等）
- 可在多个代理层之间保留源 IP
- 不影响路由或节点选择

注意事项：

- 负载均衡器和 Ingress-Nginx 都必须配置为使用 PROXY protocol
- 启用后，所有流向 Ingress-Nginx 的流量都必须使用 PROXY protocol（混合使用 PROXY 和非 PROXY 流量会导致连接失败）

配置方法

配置 HAProxy 负载均衡器以发送 PROXY protocol headers，然后部署启用了 proxy-protocol 支持的 ingress-nginx：

```

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

```

```

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

```

更多详情请参阅 [PROXY protocol 文档](#)。

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

通过带有 `externalTrafficPolicy=Local` 的 MetalLB

工作原理

当使用 `type: LoadBalancer` 的 Kubernetes Service 时，默认行为（`externalTrafficPolicy: Cluster`）会执行源 NAT，从而将客户端 IP 替换为节点的 IP。将 `externalTrafficPolicy: Local` 设置为通过以下方式保留源 IP：

1. 直接路由：流量只会被路由到接收该流量的同一节点上的 Pod
2. 不进行 **SNAT**：kube-proxy 不执行源 NAT，从而保留原始客户端 IP
3. 健康检查：只有本地 Pod 健康的节点才会被纳入负载均衡器池

流量流程：

1. 客户端连接到 MetalLB 虚拟 IP
2. MetalLB 将流量直接路由到具有 Ingress-Nginx Pod 的节点
3. 流量直接到达本地 Ingress-Nginx Pod，不经过 SNAT
4. Ingress-Nginx 看到真实的客户端 IP
5. 后端应用在 headers 中接收到正确的客户端 IP

优点：

- 配置简单，不需要额外协议
- 原生 Kubernetes 功能
- 更低的延迟（没有额外的代理跳转）

注意事项：

- 负载分布不均：流量只能进入拥有本地 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 可以路由到的节点上（使用 nodeSelector 确保对齐）
- 健康检查行为：如果所有本地 Pod 都不健康，该节点会被完全移出负载均衡

配置方法

部署一个设置了 `externalTrafficPolicy: Local` 的 ingress-nginx，并确保 Pod 放置与 MetalLB 配置一致：

```
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
  namespace: ingress-nginx-operator
spec:
  controller:
    service:
      type: LoadBalancer # Use MetalLB to provision a
LoadBalancer service
      externalTrafficPolicy: Local # Preserve source IP by routi
ng traffic only to local pods
      annotations:
        metallb.universe.tf/address-pool: demo-pool # Specify the Metall
B IP address pool to use
      nodeSelector: # Schedule pods only on nodes
matching these labels. This selector must match the MetalLB address poo
l's node selector
        ingress-nginx: "true"
```

重要： `nodeSelector` 必须与 MetalLB 地址池配置中的节点匹配，以确保 Ingress-Nginx Pod 被调度到能够接收来自 MetalLB 流量的节点上。

更多详情，请参阅 [externalTrafficPolicy 文档](#)。

Envoy Gateway 任务

目录

概述

前提条件

高级任务

OpenTelemetry (OTel)

如何附加到在其他命名空间中创建的监听器

如何使用在其他命名空间中创建的证书

如何使用 SSL Passthrough

如何更改最小 TLS 版本

如何在使用 NodePort Service 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`

方法 1：使用端口偏移（默认，推荐）

方法 2：使用 `useListenerPortAsContainerPort` 直接使用特权端口

如何从集群内部访问 LoadBalancer VIP

相关文档

更多配置

概述

在完成主要配置路径后，请使用这些高级 Envoy Gateway 任务：operator、gateway、route 和 policy。

在 Gateway API 中应用配置更改时，主要有三种可用方法：

1. 修改标准 **Gateway API** 资源：直接编辑 `Gateway`、`HTTPRoute`、`TCPRoute`、`UDPRoute` 以及其他核心资源。
2. 通过 **PolicyAttachment** 附加 **policy**：使用 `SecurityPolicy`、`ClientTrafficPolicy`、`BackendTrafficPolicy` 以及其他 policy 资源来扩展 Gateway 和 Route 的行为。
3. 配置全局设置：修改 `EnvoyGatewayCtl` 以更改 `envoy-gateway` 实例行为，或影响所有 gateway 的其他全局设置。

高级任务和特殊场景包括跨命名空间路由、可观测性设置、部署自定义和故障排查。

前提条件

1. [配置 EnvoyGatewayCtl](#)
2. [配置 Gateway](#)
3. [配置 Route](#)
4. [配置 GatewayAPI Policy](#)

高级任务

OpenTelemetry (OTel)

请按照 [OpenTelemetry 集成](#) 中的说明进行操作，但请使用 `EnvoyGatewayCtl` 修改 `envoy-gateway-config`。

如何附加到在其他命名空间中创建的监听器

在 Gateway 的监听器配置中，你需要指定允许哪些命名空间向其附加 Routes。

```
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 # without limit
    - name: http-81
      protocol: HTTP
      port: 81
      allowedRoutes:
        namespaces:
          from: Same # only allow routes in the same namespace
    - name: http-82
      protocol: HTTP
      port: 82
      allowedRoutes:
        namespaces:
          from: Selector
          selector:
            matchLabels:
              team: frontend # only allow routes in the namespace with label team=frontend
```

有关更多详细信息，请参阅 [跨命名空间路由](#)。

如何使用在其他命名空间中创建的证书

要使用在其他命名空间中创建的证书，请在存储证书的命名空间中创建一个

`ReferenceGrant`。请按照 [cross-namespace-certificate-references](#) 和 [referencegrant](#) 中的说明进行操作。

NOTE

你不能指定单个 `secret` 资源；必须允许整个命名空间

如何使用 SSL Passthrough

请按照以下说明进行操作：

- [tls](#) ↗
- [tls-passthrough](#) ↗

如何更改最小 TLS 版本

请按照 [customize-gateway-tls-parameters](#) ↗ 中的说明进行操作

```
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](#) ↗。如果你除了最小 TLS 版本之外还需要自定义密码套件，请参阅同一上游任务和 API 参考。

如何在使用 NodePort Service 时指定 NodePort

在使用 NodePort service 时，Kubernetes 会为每个 service 端口分配一个 NodePort 值。通过节点 IP 访问 service 时，请使用分配的 NodePort，而不是 service 端口。

有两种方法：

按照 [get nodeport from svc port](#) 手动获取 NodePort 分配。

在 `EnvoyProxy` 配置中手动指定 NodePort，而不是让 Kubernetes 自动分配。

```
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，以指定 NodePort

NOTE

NodePort 只能位于特定范围内，通常是 `30000-32767`。如果你希望 Gateway 监听器端口和 NodePort 保持一致，那么监听器端口也必须位于 NodePort 范围内。

如何在使用 MetalLB 时指定 VIP

当使用 MetalLB 作为 LoadBalancer 提供程序时，你可以通过 service 注解为 Gateway service 指定一个静态 VIP。

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

- ❶ 在 `envoyService.annotations` 字段中添加 MetalLB 注解
- ❷ 指定用于分配 IP 的地址池名称
- ❸ 或者指定一个特定的 IP 地址（必须位于地址池范围内）

可用注解：

注解	描述
<code>metallb.universe.tf/address-pool</code>	选择用于分配 IP 的地址池
<code>metallb.universe.tf/loadBalancerIPs</code>	指定一个特定的 IP 地址（支持多个 IP，使用逗号分隔）

NOTE

- 指定的 IP 必须位于已配置的 MetalLB 地址池中
- 在指定 VIP 之前，请确保 MetalLB 已正确安装和配置
- 有关 MetalLB 配置，请参阅 [Configure MetalLB](#)

如何在 **Envoy Gateway** 中添加 **Pod** 注解

在 `EnvoyProxy` 的

```
.spec.provider.kubernetes.envoyDeployment.patch.value.spec.template.metadata.annotations
```

中设置 `annotations`。

```

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:
                    your-annotation-key: 'your-annotation-value'
            container:
              imageRepository: registry.alauda.cn:60080/acp/envoyproxy/envoy
      type: Kubernetes

```

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

更新 `Subscription` 资源。

```
# example of nodeSelector and tolerations
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。

```
# in default $NAME=cpaas-default and $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 Pods 直接使用宿主机网络命名空间。这在以下场景中很有用：

- 获得更好的网络性能
- 通过节点 IP 直接访问 gateway

注意事项：

- 使用 `hostNetwork` 的 Pods 会直接绑定到宿主机的网络接口
- 如果多个 Pods 尝试在同一节点上使用同一端口，可能会发生端口冲突
- 由于 Pods 共享宿主机的网络命名空间，安全隔离会降低
- 你应该使用 `nodeSelector` 或 `affinity` 规则来控制 Pod 调度位置并避免端口冲突

根据你是否希望直接使用特权端口（< 1024），有两种配置 `hostNetwork` 的方法：

方法 1：使用端口偏移（默认，推荐）

这是默认且推荐的方法。Envoy Gateway 会自动为特权端口加上 10000 的偏移量，以避免需要特殊权限。

优点：

- 不需要特殊权限或能力
- 更安全，因为它以非 root 用户身份运行，不需要额外权限
- 配置更简单
- 开箱即用

缺点：

- 客户端必须使用偏移后的端口（10080、10443）

配置：

```

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 # Enable host network mode
                  dnsPolicy: ClusterFirstWithHostNet # Required for proper DNS resolution
            pod:
              nodeSelector: # Recommended: control pod placement to avoid conflict
                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

方法 2：使用 `useListenerPortAsContainerPort` 直接使用特权端口

这种方法允许 Envoy 直接绑定特权端口（< 1024），例如端口 80 和 443。

优点：

- 可以直接使用标准端口（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
- 与端口偏移方法相比，安全性略低
- 配置更复杂

配置：

```

apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    type: Kubernetes
    kubernetes:
      useListenerPortAsContainerPort: true # Disable port offset
      envoyDeployment:
        patch:
          type: StrategicMerge
          value:
            spec:
              template:
                spec:
                  hostNetwork: true
                  dnsPolicy: ClusterFirstWithHostNet
                  containers:
                    - name: envoy
                      command:
                        - /usr/local/bin/envoy-with-cap # use envoy with f
ilecap

                      securityContext:
                        capabilities:
                          add:
                            - NET_BIND_SERVICE # Required for binding to pri
vileged ports
                  pod:
                    nodeSelector:
                                                                # Recommended: con
trol pod placement to avoid conflict
                    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 会创建 `externalTrafficPolicy: Local` 的 LoadBalancer service。此策略可保留客户端源 IP 地址，但有一个重要限制：来自没有 Envoy Gateway Pods 的集群节点的请求将会失败，因为流量不会转发到其他节点。

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

对于在集群内部运行的应用，请使用 service ClusterIP，而不是 LoadBalancer VIP。这样可以完全避免路由限制。

解决方案 2：更改为 **Cluster** 流量策略

如果你需要从任意集群节点访问 LoadBalancer VIP，请将 `externalTrafficPolicy` 更改为 `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 Policy](#)

更多配置

请参阅 [EnvoyGateway Tasks](#) ↗

软数据中心 LB 解决方案 (Alpha)

通过在集群外创建一个高可用负载均衡器，部署纯软件的数据中心负载均衡器 (LB)，为多个 ALB 提供负载均衡能力，以确保业务稳定运行。它支持仅配置 IPv4、仅配置 IPv6，或同时配置 IPv4 和 IPv6 双栈。

目录

[前提条件](#)[操作步骤](#)[验证](#)

前提条件

1. 准备两个或更多主机节点作为 LB。建议在 LB 节点上安装 Ubuntu 22.04 操作系统，以减少 LB 将流量转发到异常后端节点所需的时间。
2. 在外部 LB 的所有主机节点上预先安装以下软件（本章以两个外部 LB 主机节点为例）：
 - `ipvsadm`
 - `container-runtime such as containerd`
3. 确保每个主机的 `container-runtime` 在开机时启动。
4. 确保每个主机节点的时钟已同步。

5. 准备用于启动外部 LB 服务的 Keepalived 镜像；平台已包含该镜像。镜像地址格式如下：

`<image repository address>/tkestack/keepalived:<version suffix>`。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。
- 在解压安装包的目录中执行 `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：以下操作必须在每个外部 LB 主机节点上执行一次，且主机节点的 `hostname` 不能重复。

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

```
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` 中。


```
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. 为 Keepalived 配置文件创建一个文件夹。

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

5. 根据以下文件中的注释修改配置项，并将其保存到 `/etc/keepalived/` 目录下，文件名为 `alive.yaml`。



instances:

```
- vip: # Multiple VIPs can be configured
    vip: 192.168.128.118 # VIPs must be different
    id: 20 # Each VIP's ID must be unique, optional
    interface: "eth0"
    check_interval: 1 # optional, default 1: interval to execute check script
    check_timeout: 3 # optional, default 3: check script timeout period
    name: "vip-1" # Identifier for this instance, can only contain alphanumeric characters and hyphens, cannot start with a hyphen
    peer: [ "192.168.128.116", "192.168.128.75" ] # Keepalived node IP, actual generated keepalived.conf will remove all IPs on the interface.

    kube_lock:
        kubecfgs: # The kube-config list used by kube-lock will sequentially attempt these kubecfgs for leader election in Keepalived
            - "/live/cfg/kubecfg/kubecfg01.conf"
            - "/live/cfg/kubecfg/kubecfg02.conf"
            - "/live/cfg/kubecfg/kubecfg03.conf"
        ipvs: # Configuration for option IPVS
            ips: [ "192.168.143.192", "192.168.138.100", "192.168.129.100" ] # IPVS backend, change k8s master node IP to ALB node's node IP
            ports: # Configure health check logic for each port on the VIP
                - port: 80 # The port on the virtual server must match the real server's port
                    virtual_server_config: |
                        delay_loop 10 # Interval for performing health checks on the real server
                        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: # The kube-config list used by kube-lock will sequentially attempt these kubecfgs for leader election in Keepalived
- "/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::192:168:129:100" ]
  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. 在业务集群中执行以下命令，检查配置文件中的证书过期时间，确保该证书仍然有效。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-certificate-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` 目录下，并按索引命名，例如 `kubecfg01.conf`，同时将这三个文件中的 `apiserver` 节点地址修改为 Kubernetes 集群的实际节点地址。

注意：平台证书更新后，需要重新执行此步骤，并覆盖原有文件。

8. 检查证书有效性。

1. 将业务集群 Master 节点上的 `/usr/bin/kubect1` 复制到 LB 节点。
2. 执行 `chmod +x /usr/bin/kubect1` 授予执行权限。
3. 执行以下命令确认证书是否有效。

```
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
```

如果返回以下结果，则证书有效。

```
kubectl --kubeconfig=/etc/keepalived/kubecfg/kubecfg01.conf get node
## Output
```

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
## Output
```

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
## Output
```

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 镜像上传到外部 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` 的节点上运行以下命令：`sysctl -w net.ipv4.conf.all.arp_accept=1`。

验证

1. 运行 `ipvsadm -ln` 命令查看 IPVS 规则，可以看到适用于业务集群 ALB 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 InAct
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 是否都能成功迁移到另一节点，通常会在 20 秒内完成。
3. 在非 LB 节点上使用 `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="http://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>
```

```
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="http://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 和智能体

监控、运维工具和扩展组件

准备 Kube-OVN Underlay 物理网络 Kube-OVN 网络

使用说明

术语说明

环境要求

配置示例

概述

前提条件

配置步骤

集群互联 (Alpha)

支持配置使用 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

相关操作

配置 Kube-OVN 网络以支持 Pod 多网络接口

开始之前

安装 Multus CNI

创建辅助网络

创建具有多个网络接口的 Pod

验证网络接口

可选配置

配置 IPPool

说明

注意事项

配置 MTU

默认 MTU 行为

自定义 MTU 选择

配置 BGP

Underlay 和 Overlay 子网的自动互联

操作步骤

启用 u2oInterconnection 后的 Underlay 子网隔离

了解 Kube-OVN CNI

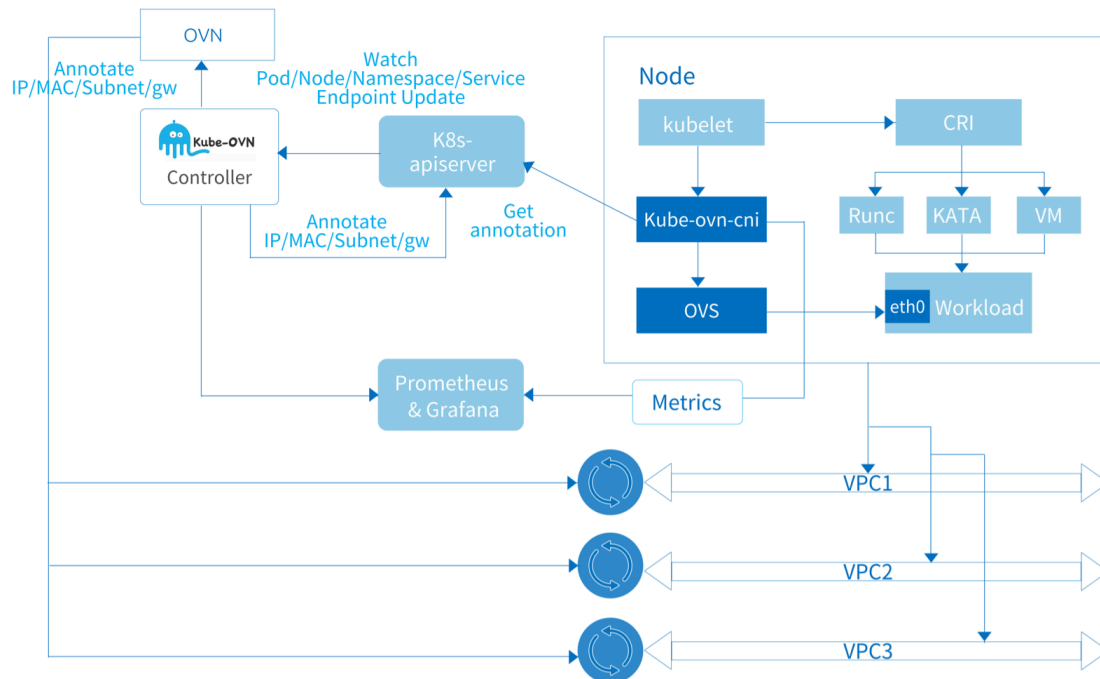
Kube-OVN 提供了连接 Kubernetes 网络与 OVN 的 CNI 架构、组件和交互。

总体而言，Kube-OVN 充当 Kubernetes 与 OVN 之间的桥梁，将成熟的 SDN 与 Cloud Native 结合起来。这意味着 Kube-OVN 不仅实现了 Kubernetes 下的网络规范，例如 CNI、Service 和 Networkpolicy，还将大量 SDN 领域的能力带入云原生，例如逻辑交换机、逻辑路由器、VPC、网关、QoS、ACL 和流量镜像。

Kube-OVN 也保持了良好的开放性，可与许多技术方案集成，例如 Cilium、Submariner、Prometheus、KubeVirt 等。

Kube-OVN 的组件大致可分为三类。

- 上游 OVN/OVS 组件。
- 核心 Controller 和智能体。
- 监控、运维工具和扩展组件。



目录

上游 OVN/OVS 组件

ovn-central

ovs-ovn

核心 Controller 和智能体

kube-ovn-controller

kube-ovn-cni

监控、运维工具和扩展组件

kube-ovn-speaker

kube-ovn-pinger

kube-ovn-monitor

kubectl-ko

上游 OVN/OVS 组件

这类组件来自 OVN/OVS Community，并针对 Kube-OVN 的使用场景做了特定修改。

OVN/OVS 本身是一个成熟的 SDN 系统，用于管理虚拟机和容器，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 是什么以及如何与之集成。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 相关的组件都已打包为镜像，可直接在 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 API。`kube-ovn-controller` 主要会与 `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` 实例会通过 Raft 协议同步数据，以确保高可用。

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。

资源限制

在大型集群中，`ovs-ovn` DaemonSet 中的 `openvswitch` 容器可能会超出其默认资源限制：

- 内存压力 — 容器会被 `OOMKilled`，Pod 重启（退出码 137）。
- CPU 压力 — 在 Pod 创建/删除突发、较大的逻辑流更新或高包速率等高负载情况下，容器可能触及 CPU 限制。`ovn-controller` 和 `ovs-vswitchd` 会被限流——Pod 不会被杀死，但 CPU 资源不足可能导致偶发丢包（数据平面无法跟上），同时 Pod 网络启动变慢、转发延迟增加。

当出现上述任一情况时，请提高相关限制：

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

在 `openvswitch` 容器的 `resources` 配置块中，提高 `limits.memory` / `limits.cpu`（以及对应的 `requests`）。对于长期运行的集群，请在 kube-ovn Helm values 中的 `ovs-ovn.resources` 下设置相同的值，以便升级后仍然保留这些更改。

核心 Controller 和智能体

这部分是 Kube-OVN 的核心组件，作为 OVN 与 Kubernetes 之间的桥梁，连接两个系统并在其间转换网络概念。大多数核心功能都由这些组件实现。

kube-ovn-controller

该组件负责将 Kubernetes 中的所有资源转换为 OVN 资源，并作为整个 Kube-OVN 系统的控制平面。`kube-ovn-controller` 会监听所有与网络功能相关的资源事件，并根据资源变化更新 OVN 中的逻辑网络。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` 会监听 Pod 创建事件，通过内置的内存型 IPAM 功能分配地址，并调用 `ovn-central` 创建逻辑端口、静态路由以及可能的 ACL 规则。接下来，`kube-ovn-controller` 会将分配的地址以及 CIDR、gateway、route 等 subnet 信息写入 Pod 的注解。随后，`kube-ovn-cni` 会读取该注解，并据此配置本地网络。

kube-ovn-cni

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

这个 DaemonSet 会将 `kube-ovn` 二进制文件复制到每台机器上，作为 `kubelet` 与 `kube-ovn-cni` 之间交互的工具。该二进制文件会将对应的 CNI 请求发送给 `kube-ovn-cni` 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` 会配置特定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 pair，并绑定或解绑到 OVS 端口。
 2. 配置 OVS 端口。
 3. 更新主机 iptables/ipset/route 规则。
3. 动态更新网络 QoS。
4. 创建并配置 `ovn0` NIC，以连接容器网络和主机网络。
5. 配置主机 NIC，以实现 Vlan/Underlay/EIP。
6. 动态配置集群间网关。

监控、运维工具和扩展组件

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

kube-ovn-speaker

该组件是一个运行在带有特定标签节点上的 DaemonSet，用于向外部发布路由，使外部可以通过 Pod IP 直接访问容器。有关部署和 BGP 选项，请参见 [Configure BGP](#)。

kube-ovn-pinger

该组件是一个运行在每个节点上的 DaemonSet，用于收集 OVS 状态信息、节点网络质量、网络延迟等。

kube-ovn-monitor

该组件用于收集 OVN 状态信息和监控指标。

kubectl-ko

该组件是一个 kubectl 插件，可快速执行常见操作。

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

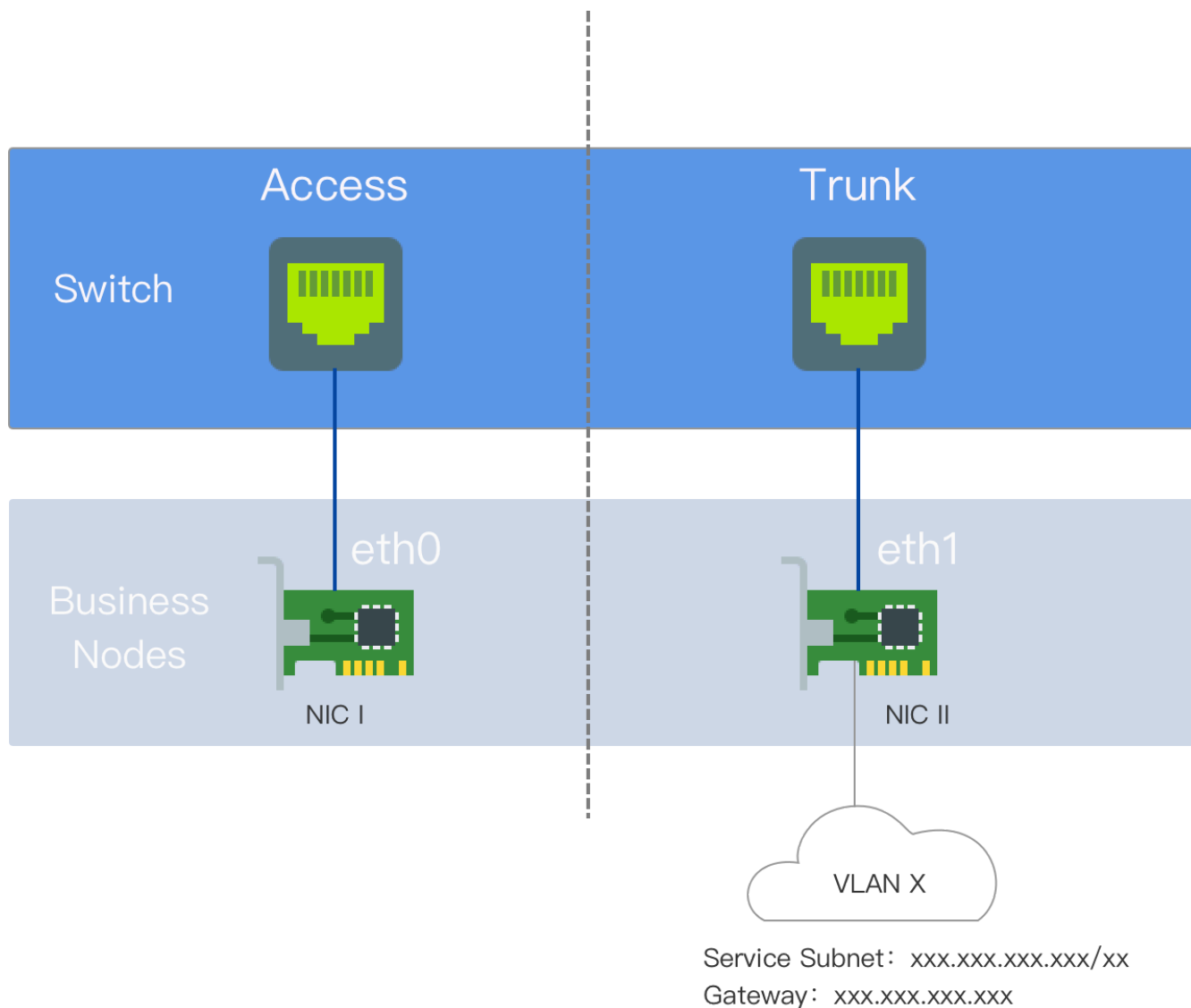
[使用说明](#)[术语说明](#)[环境要求](#)[配置示例](#)[交换机配置](#)[检查网络连通性](#)[平台配置](#)

使用说明

Kube-OVN Underlay 需要部署在多网卡（NIC）环境中，且 Underlay 子网必须独占一块网卡。该网卡上不能承载其他类型的流量，例如 SSH；这类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，并建议网卡速率至少为 **10 Gbps** 或更高（例如 10 Gbps、25 Gbps、40 Gbps）。

- 网卡一：具有默认路由、已配置 IP 地址、与外部交换机接口互联且配置为 Access 模式的网卡。
- 网卡二：没有默认路由且未配置 IP 地址、与外部交换机接口互联且配置为 Trunk 模式的网卡。Underlay 子网仅使用网卡二。



术语说明

VLAN (Virtual Local Area Network) 是一种将局域网在逻辑上划分为多个网段 (或更小的 LAN) 的技术, 以便于虚拟工作组之间进行数据交换。

VLAN 技术的出现使管理员能够根据实际应用需求, 将同一物理局域网中的不同用户在逻辑上划分到不同的广播域中。每个 VLAN 都由一组具有相似需求的计算机工作站组成, 并具备与物理形成的 LAN 相同的属性。由于 VLAN 是逻辑划分而非物理划分, 因此同一 VLAN 中的工作站不必局限于同一物理区域, 它们可以跨越不同的物理 LAN 段存在。

VLAN 的主要优点包括：

- 端口隔离。即使在同一交换机上，不同 VLAN 中的端口也无法相互通信。一个物理交换机可以充当多个逻辑交换机。这通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，从而消除了广播信息带来的不安全性。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性。
- 灵活管理。当需要更改用户的网络归属时，无需更换端口或网线，只需更改软件配置即可。

环境要求

在 Underlay 模式下，Kube-OVN 会将物理 NIC 桥接到 OVS，并通过该物理 NIC 直接将报文发送到外部网络。二层/三层转发能力依赖底层网络设备。相应的网关、VLAN 和安全策略需要在底层网络设备上预先配置。

- 网络配置要求
 - Kube-OVN 在启动容器时会通过 ICMP 协议检查网关连通性；底层网关必须能够响应 ICMP 请求。
 - 对于服务访问流量，Pods 会首先将报文发送到网关，而网关必须具备将报文转发回本地子网的能力。
 - 当交换机或桥开启 Hairpin 功能时，必须关闭 **Hairpin**。如果使用 VMware 虚拟机环境，请将 VMware 主机上的 **Net.ReversePathFwdCheckPromisc** 设置为 **1**，此时无需关闭 Hairpin。
 - 作为桥接的 NIC 不能是 **Linux Bridge**。
 - NIC 绑定模式支持 Mode 0 (balance-rr)、Mode 1 (active-backup)、Mode 4 (802.3ad)、Mode 6 (balance-alb)，建议使用 0 或 1。其他绑定模式尚未测试，请谨慎使用。
- IaaS（虚拟化）层配置要求
 - 对于 OpenStack 虚拟机环境，需要禁用对应网络端口的 **PortSecurity**。
 - 对于 VMware 的 vSwitch 网络，**MAC Address Changes**、**Forged Transmits** 和 **Promiscuous Mode Operation** 都必须设置为 **Accept**。

- 对于 AWS、GCE 和 Alibaba Cloud 等公有云，由于不支持用户自定义 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器。网卡一是具有默认路由的网卡；网卡二没有默认路由且未配置 IP 地址，仅用于 Underlay 子网。网卡二与外部交换机互联。

- 在交换机侧，与网卡二相连的接口应配置为 Trunk 模式，以允许相应 VLAN 通过。
- 在对应的 vlan-interface 接口上配置集群子网的网关地址。如果需要双栈，也可以同时配置 IPv6 网关地址。
- 如果网关位于防火墙后方，需要允许节点访问 cluster-cidr 网络。
- 服务器网卡无需额外配置。

交换机配置

配置 VLAN 接口：

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

配置与网卡二相连的接口：

```
#
interface Ten-GigabitEthernet1/0/19
    port link mode bridge
    port link-type trunk    // Configure the interface to Trunk mode
    undo port trunk permit vlan 1
    port trunk permit vlan 74    // Allow the corresponding VLAN to pass through
#
```

检查网络连通性

测试网卡二是否可以与网关地址通信：

```
ip link add ens224.74 link ens224 type vlan id 74 // The NIC name is ens
224, and the VLAN ID is 74
ip link set ens224.74 up
ip addr add 192.168.74.200/24 dev ens224.74 // Select a test address wit
hin the Underlay subnet, here it's 192.168.74.200/24
ping 192.168.74.254 // If able to ping the gateway, it confirms that the
physical environment meets deployment requirements
ip addr del 192.168.74.200/24 dev ens224.74 // Delete the test address a
fter testing
ip link del ens224.74 // Delete the sub-interface after testing
```

平台配置

在左侧导航栏中，点击 **Cluster Management > Cluster**，然后点击 **Create Cluster**。关于集群创建路径的选择，请参见 [Clusters Overview](#)，容器网络配置如下图所示。

注意：Join 子网在 Underlay 环境中没有实际意义，主要用于后续创建 Overlay 子网，为节点与容器组之间的通信提供所需的 IP 地址范围。

Container Networking

IPv4 / IPv6 Dual Stack: ☒

Ensure that all nodes are correctly configured with IPv6 network addresses when enabling IPv4/IPv6 dual stack, as the cluster will not revert to IPv4 single stack after creation.

Network Type:

Kube-OVN Calico Flannel Custom ?

Default Subnet:

* IPv4: 192 . 168 . 74 . 0 / 24

* IPv6: 2074::/64

Transmit Mode:

Overlay Underlay ?

Gateway:

* IPv4 192.168.74.254

* IPv6 2074::192.168.74.254

* VLAN ID: 74

Preserved IP:

Protocol stack	IP Format	* IP Address
! If the IP in the subnet is occupied by the physical network, the cluster cannot be created successfully. Please set it as reserved IP		
+ Add		

IPv4 subnet address of NIC II

IPv6 subnet address of NIC II

IPv4 gateway address

IPv6 gateway address

VLAN ID that the switch allows to pass through

The default gateway IPv4/IPv6 value must be within the cluster CIDR address range

After the cluster is created, new subnets are supported.

* Service CIDR:

* IPv4: 10 . 184 . 0 . 0 / 16

* IPv6: fd00:10:96::/112

Custom SVC, must not duplicate with the internal network

* Join CIDR:

* IPv4: Custom 100.64.0.0/16

* IPv6: fd00:100:64::/64

Address segment of the NIC used for communication on the Overlay network

Kube-OVN Underlay + MetalLB 负载均衡器 Service 配置

目录

概述

前提条件

环境要求

流量流向

配置步骤

1. 使用 VLAN 子接口配置 ProviderNetwork
2. 配置 Kube-OVN Controller 参数
3. 配置 Underlay 子网外部地址功能
4. 创建 MetalLB 外部地址池
5. 创建示例应用和 LoadBalancer Service
6. 验证配置
7. 迁移现有 Service

概述

此方案用于将 MetalLB L2 模式与 Kube-OVN Underlay 网络集成。它允许用户将 Underlay 子网 IP 用作 MetalLB 负载均衡器 Service 的 VIP，并将流量直接转发到后端业务 Pods。

⚠ 关键：LoadBalancer VIP 和后端 Pod IP 必须位于同一个 **Underlay** 子网中。

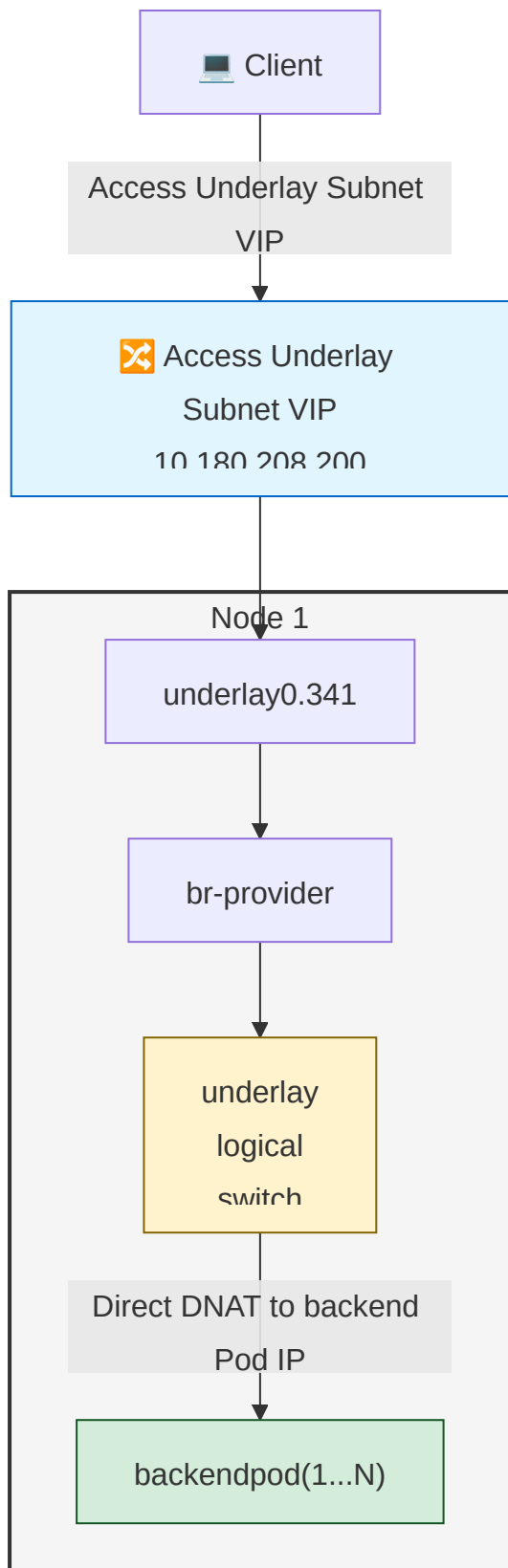
前提条件

环境要求

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

流量流向

流量图：



配置步骤

1. 使用 VLAN 子接口配置 ProviderNetwork

重要：必须使用 VLAN 子接口。

配置 Kube-OVN Underlay 网络以自动创建 VLAN 子接口：

```
apiVersion: kubeovn.io/v1
kind: ProviderNetwork
metadata:
  name: provider
spec:
  defaultInterface: underlay0.341
  autoCreateVlanSubinterfaces: true # Automatically creates VLAN sub-int
    erfaces (e.g., underlay0.341) if only parent interface (underlay0) exists
    # If customInterfaces are used, each interface MUST also be the VLAN su
    b-interface
    # (parent interface name + VLAN suffix, e.g., underlay0.341), not the p
    arent interface
    # customInterfaces:
    #   - interface: underlay0.341
    #     nodes:
    #       - node1

---
apiVersion: kubeovn.io/v1
kind: Vlan
metadata:
  name: ovn-vlan
spec:
  id: 0 # Use 0 because autoCreateVlanSubinterfaces creates the VLAN s
    ub-interface (underlay0.341) which handles VLAN tagging, not Kube-OVN dir
    ectly
  provider: provider
status:
  subnets:
    - ovn-default
```

⚠ 重要：如果 `ProviderNetwork` 通过 `customInterfaces` 指定了按节点的接口，则每个 `customInterfaces[].interface` 也必须使用 **VLAN** 子接口（父接口名称 + VLAN 后

缀，例如 `underlay0.341`），而不能使用父接口（例如 `underlay0`）。如果在 `customInterfaces` 中保留父接口，MetalLB 负载均衡器流量将无法到达后端 Pods。

 **警告：**单独修改 `ProviderNetwork` 或 `Vlan` 资源时，Underlay 网络连通性将会中断。只有在这两个资源都完全配置并保持同步后，网络连通性才会恢复。请在维护窗口期间安排配置变更，以尽量减少服务中断。

2. 配置 Kube-OVN Controller 参数

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

使用 **Web** 控制台：

1. 导航到 管理员 > **Marketplace** > 集群插件，然后搜索 `ovn`，找到 **Alauda Container Platform Networking for Kube-OVN**
2. 在插件行中，单击操作菜单（竖向⋮），然后选择 **更新** 以打开配置对话框
3. 配置以下设置：
 - **Skip CT for Dst LPort IPs**：否
 - **Enable OVN LB Local**：是

3. 配置 Underlay 子网外部地址功能

编辑 Underlay 子网，为 LoadBalancer 使用保留一段 IP 地址范围：

重要：外部地址池 IP 必须位于 Underlay 子网内。

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

```

apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: underlay-subnet
spec:
  enableExternalLBAddress: true      # Indicates this subnet has IP range for LB service VIP
  excludeIps:
    - 10.180.208.200..10.180.208.220 # Reserve IP range for external address pool

```

4. 创建 MetalLB 外部地址池

```

# 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 subnet IP range
  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 # Optional: Interface for ARP broadcasting; use bridge interface (br-*) instead of physical interface
  nodeSelectors: []

```

部署地址池：

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

5. 创建示例应用和 LoadBalancer Service

```
# 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
  # To use specific IPPool: add annotation `metallb.io/address-pool: acp-
  underlay-pool`
  # To use fixed IP: set `spec.loadBalancerIP: 10.180.208.201`
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local # **IMPORTANT**: Required for preserving
  source IP and enabling direct Pod routing
  selector:
    app: backend
  ports:
    - port: 80
      targetPort: 80
```

部署应用：

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

6. 验证配置

```
# Check service status
kubectl get svc backend-lb-service -o wide

# Test external access
curl http://10.180.208.200
```

7. 迁移现有 Service

对于使用旧地址池（仅节点子网）的现有 Service，您可以将其迁移到新的 Underlay 地址池：

```
# Add annotation to migrate existing service
kubectl annotate service <existing-service-name> metallb.io/address-pool=
acp-underlay-pool --overwrite

# Verify the service has been assigned a new IP from the Underlay pool
kubectl get svc <existing-service-name> -o wide
```

对于新 **Service**，直接添加注解：

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

```
# Check service status
kubectl get svc backend-lb-service -o wide

# Test external access
curl http://10.180.208.200
```


集群互联 (Alpha)

它支持配置网络模式与 Kube-OVN 相同的集群之间的集群互联，使集群中的 Pod 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。
- 需要有一组能够被各集群的 kube-ovn-controller 通过 IP 访问的机器，用于部署跨集群互联的控制器。
- 每个集群都需要存在一组可被 kube-ovn-controller 通过 IP 访问的机器，后续将用作网关节点。
- 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 及更高版本支持）、Podman 部署和 Containerd 部署。

Deploy 部署

注意：平台 v3.16.0 及更高版本支持此部署方式。

操作步骤

1. 在集群主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 image repository address>"    ## For example: REGISTRY="registry.alauda.cn:60080/acp/"  
VERSION="<Kube-OVN version>"    ## For example: VERSION="v1.9.25"
```

3. 保存脚本文件，并使用以下命令执行脚本。

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

Podman 和 Containerd 部署

1. 选择任意集群中的 三个或以上节点 来部署互联控制器。以下示例准备了三个节点。

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，请先检查 `/etc` 下是否存在 `ovn` 目录。如果不存在，请使用 `mkdir /etc/ovn` 命令创建。

- 容器部署命令 注意：执行 `podman images | grep ovn` 命令以获取 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="<IP address of the current node>" \    ## For example:
  -e LOCAL_IP="192.168.39.37"
  -e NODE_IPS="<IP addresses of all nodes, separated by commas>" \
  ## For example: -e NODE_IPS="192.168.39.22,192.168.39.24,192.168.3
  9.37"
  <image repository address> bash start-ic-db.sh    ## For example:
  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="<IP address of the current node>" \   ## For example:
-e LOCAL_IP="192.168.39.24"
-e LEADER_IP="<IP address of the Leader node>" \   ## For example:
-e LEADER_IP="192.168.39.37"
-e NODE_IPS="<IP addresses of all nodes, separated by commas>" \
## For example: -e NODE_IPS="192.168.39.22,192.168.39.24,192.168.3
9.37"
<image repository address> bash start-ic-db.sh   ## For example: 1
92.168.39.10:60080/acp/kube-ovn:v1.8.8  bash start-ic-db.sh
```

- **Containerd 部署命令**

注意：执行 `crictl images | grep ovn` 命令以获取 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=<IP addresses of all nodes, separated by commas>" \
  ## For example: --env="NODE_IPS="192.168.178.97,192.168.181.93,192.168.177.192""
--env="LOCAL_IP=<IP address of the current node>" \  ## For example: --env="LOCAL_IP="192.168.178.97""
<image repository address> ovn-ic-db bash start-ic-db.sh  ## For example: 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=<IP addresses of all nodes, separated by commas>" \
  ## For example: --env="NODE_IPS="192.168.178.97,192.168.181.93,192.168.177.192"" \
--env="LOCAL_IP=<IP address of the current node>" \  ## For example: --env="LOCAL_IP="192.168.181.93""
--env="LEADER_IP=<IP address of the Leader node>" \  ## For example: --env="LEADER_IP="192.168.178.97""
<image repository address> ovn-ic-db bash start-ic-db.sh  ## For example: 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 集群的任意控制节点上，按照注释替换以下参数，并执行以下命令创建 ConfigMap 资源。

注意：为确保正常运行，global 上名为 ovn-ic 的 ConfigMap 不允许被修改。如需修改任何参数，请先删除该 ConfigMap，正确重新配置后再重新应用该 ConfigMap。

```
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" # Address of
the node where the cluster interconnect controller is located, in this ca
se, the local IP of the three nodes where the controller is deployed
  ic-nb-port: "6645" # Cluster Interconnect Controller nb por
t, default 6645
  ic-sb-port: "6646" # Cluster Interconnect Controller sb por
t, default 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

集群中已创建的子网、ovn-default 和 join 子网不能与集群互联组中的任何集群网段冲突。

操作步骤

1. 在左侧导航栏中，单击 **Clusters > Cluster of clusters**。
2. 单击要加入集群互联的 **cluster** 名称。
3. 在右上角，单击 **Options > Cluster Interconnect**。

4. 单击 **Join the cluster interconnect**。
5. 为该集群选择一个网关节点。
6. 单击 **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，单击 **Clusters > Cluster of clusters**。
2. 单击需要更新网关节点信息的 **Cluster name**。
3. 在右上角，单击 **Operations > Cluster Interconnect**。
4. 对需要更新网关节点信息的集群，单击 **Update Gateway Node**。
5. 重新选择该集群的网关节点。
6. 单击 **Update**。

退出集群互联

已加入集群互联组的集群退出集群互联后，集群 Pod 将与外部集群 Pod 断开连接。

操作步骤

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

清理互联集群残留

当集群在未退出互联集群的情况下被删除时，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，可能会失败。你可以在控制器（kube-ovn-controller）的

`/var/log/ovn/ovn-ic.log` 日志中查看详细错误信息。部分错误信息可能包括：

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

操作步骤

1. 对要加入的集群执行[退出互联集群](#)。

2. 在容器或 Pod 中执行清理脚本。

你可以直接在 `ovn-ic-db` 容器或 `ovn-ic-controller` Pod 中执行清理脚本。以下两种方式任选其一：

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：

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

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

- 使用原始集群中任意节点的名称执行清理操作。将 `<node-name>` 替换为原始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-head
ers) -- /bin/bash
```

然后执行以下任一清理命令：

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

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

- 使用原始集群中任意节点的名称执行清理操作。将 `<node-name>` 替换为原始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

- 退出互联集群。具体退出方式有两种，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。

- 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 -l ovn
-nb-leader=true -o custom-columns=NAME:.metadata.name --no-headers) --
/bin/bash
```

- 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` 字段改为 `false`。

注意：将 `enable-ic` 字段改为 `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
```

```
# Configuration example
```

```
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 {current cluster name}-ts` 命令，验证配置是否生效。

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

```
# Return to the display example. In this case, the values of 100 and 99
are the priority, and the larger the value, the higher the priority of
the corresponding gateway node to be used.
```

```
az1-ts-71292a21-131d-492a-9f0c-0611af458950 100
```

```
az1-ts-1de7ee15-f372-4ab9-8c85-e54d61ea18f1 99
```

配置 Egress Gateway

目录

概述

Egress Gateway 与 Centralized Gateway 的对比

Egress Gateway 的工作原理

开始之前

配置流程

步骤 1：准备外部网络附加

步骤 2：创建 VPC Egress Gateway

步骤 3：验证网关

1. 检查资源状态
2. 检查网关 Pod 内部的网络
3. 确认流量转发
4. 确认 OVN 路由策略

可选：启用多副本负载均衡

可选：配置带宽限制

可选：启用基于 BFD 的高可用

1. 在 VPC 上启用 BFD Port
2. 在 VPC Egress Gateway 上启用 BFD
3. 验证 BFD 状态

可能中断流量的操作

其他资源

概述

Egress Gateway，也称为 VPC Egress Gateway，为 overlay 网络中的 Pod 提供稳定的出站地址。它会先将选定的工作负载路由到专用网关 Pod，然后流量再离开集群。

在以下场景中使用 Egress Gateway：

- 为特定工作负载提供稳定的源 IP 地址
- 以工作负载级别进行出站控制，而不是子网级别控制
- 通过水平扩展获得更高吞吐量
- 更快地处理出站流量故障切换

主要能力：

- 通过 ECMP 实现 Active-Active 高可用，并支持水平吞吐扩展
- 通过 BFD 实现快速故障切换，通常可在 1 秒内完成
- 支持 IPv4、IPv6 和双栈环境
- 通过 NamespaceSelector 和 PodSelector 实现细粒度流量匹配
- 通过 node selector 和 toleration 实现灵活调度

当前限制：

- 多副本部署需要多个 egress IP
- 不保留源 NAT 映射记录

Egress Gateway 与 Centralized Gateway 的对比

将 Egress Gateway 与 Centralized Gateway 进行对比，以选择最适合你场景的网关模型。有关 centralized 模式的详细信息，请参阅 [配置 Centralized Gateway](#)。

维度	Egress Gateway	Centralized Gateway
粒度	通过 <code>selectors</code> 和 <code>policies</code> (Namespace/Pod/Subnet/IPBlock) 进行细粒度控制。	在子网级别应用 (<code>gatewayType: centralized</code>)。
出站地址来源	使用带有外部子网 IP 的专用网关 Pod。	使用指定的网关节点；在 <code>natOutgoing: true</code> 时，出站流量使用节点 IP。
数据路径	流量先路由到 VPC Egress Gateway Pod，然后 SNAT 到外部网络。	流量路由到指定节点 (<code>ovn0</code>)，再由主机路由/NAT 规则转发。
HA 和故障切换	Active-Active ECMP；支持 BFD 快速故障切换（亚秒级）。	支持 ECMP 和主备模式；故障切换通常比基于 BFD 的 Egress Gateway 更慢。
调度控制	支持为网关工作负载配置节点调度控制 (<code>nodeSelector</code> 、 <code>tolerations</code>)。	通过 <code>gatewayNode</code> 或 <code>gatewayNodeSelectors</code> 选择网关节点。
典型使用场景	租户级或工作负载级出站隔离、细粒度策略控制、更高性能和更快的故障切换。	更简单的子网级固定源出站、审计、IP 白名单和防火墙源 IP 管理。

选择建议：

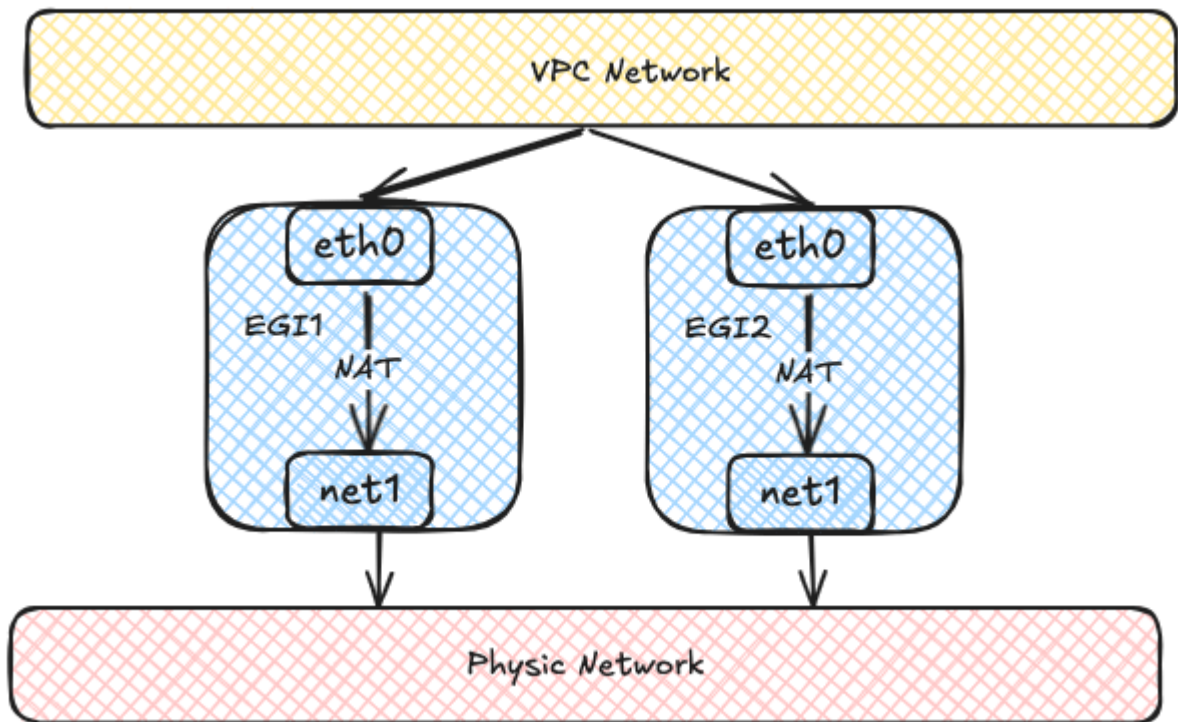
- 当你需要工作负载级策略控制、可扩展吞吐量和快速故障切换时，选择 **Egress Gateway**。
- 当子网级固定出站和简单运维已足够时，选择 **Centralized Gateway**。

Egress Gateway 的工作原理

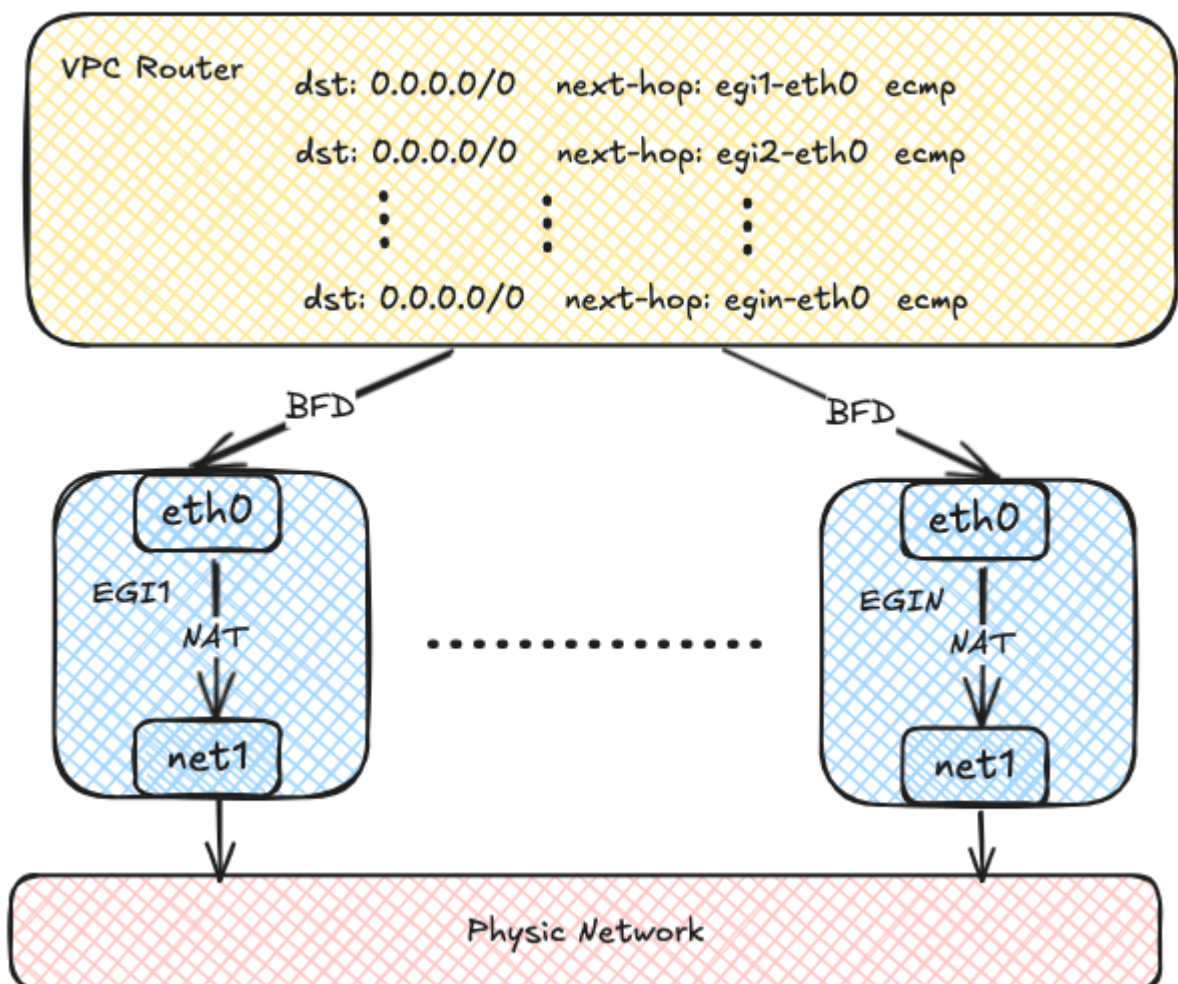
Egress Gateway 以一个或多个 Pod 的形式运行。每个 Pod 使用两个网络接口：

- 一个接口连接集群内的 overlay 网络。
- 另一个接口连接外部 underlay 网络。

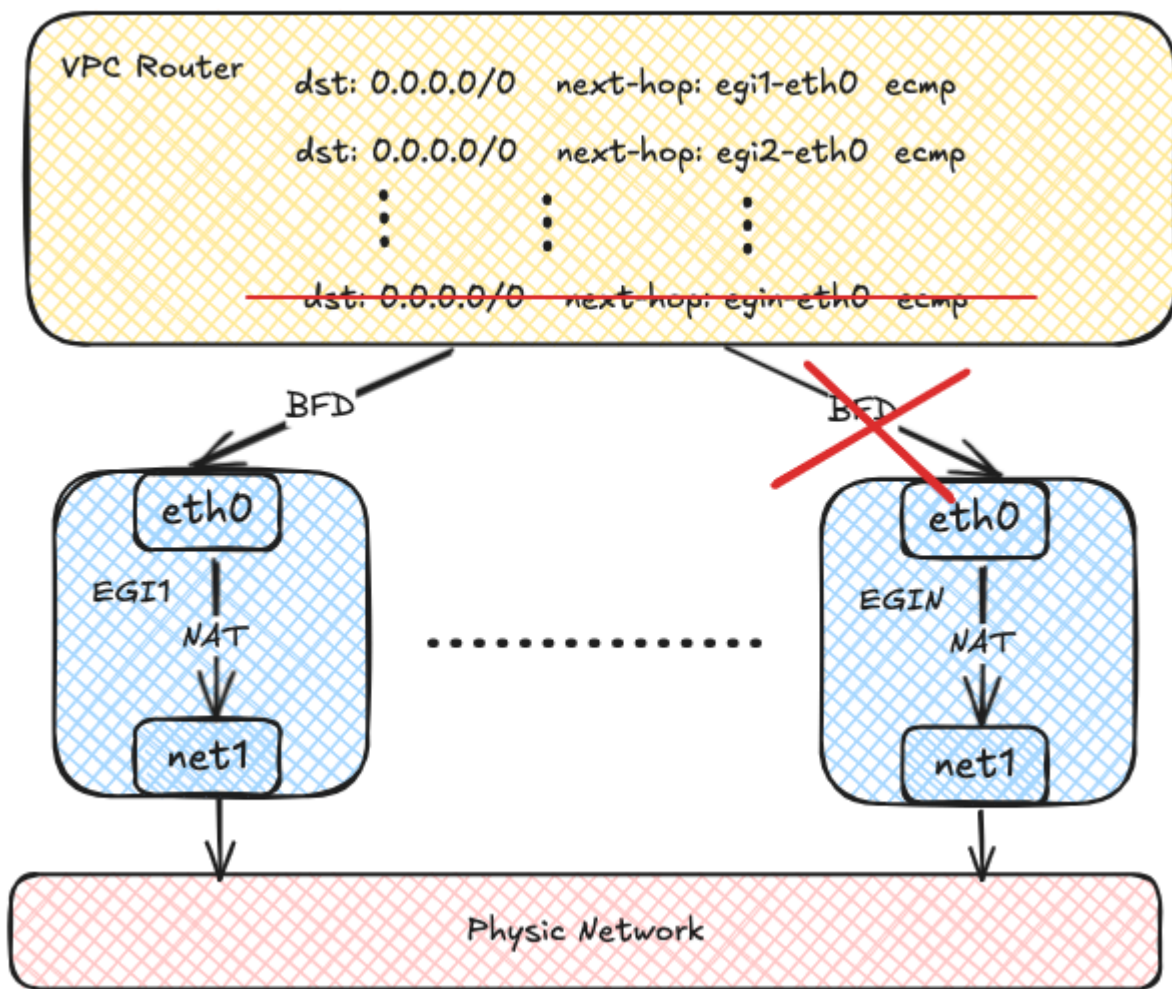
来自所选 Pod 的流量会先转发到网关 Pod，然后通过 underlay 接口发送到外部网络。



每个 Egress Gateway 实例都会在 OVN 路由表中注册其地址。当 overlay 网络中的 Pod 访问外部网络时，OVN 会使用源地址哈希在多个网关实例之间分发流量。这提供了负载均衡，并允许随着实例数量增加而水平扩展吞吐量。



OVN 可以使用 BFD 探测多个网关实例。如果某个实例失败，OVN 会将对应路由标记为不可用，并快速将流量重定向到健康的实例。



开始之前

在开始之前，请确保满足以下前提条件：

- 集群中已经安装 *Multus CNI Plugin* **MUST**。
- 已完成外部 underlay 网络、VLAN 和 bridge network 的规划并可用。
- 你已确定哪些工作负载应使用该网关，以及它们是否需要 SNAT。

要安装 *Multus CNI Plugin*，请参阅 [部署 Multus CNI Plugin](#)。

配置流程

按以下顺序配置 Egress Gateway：

1. 准备外部网络附加，包括子网和 Network Attachment Definition。
2. 创建 VPC Egress Gateway 资源，并指定外部子网和策略。
3. 验证资源状态、路由和流量转发。
4. 可选地扩展副本并启用基于 BFD 的快速故障切换。

步骤 1：准备外部网络附加

Egress Gateway 使用多个接口同时连接内部网络和外部网络。在创建网关之前，请准备以下资源：

- 一个外部子网
- 该子网对应的 Network Attachment Definition (NAD)

NOTE

NetworkAttachmentDefinition 名称 **MUST NOT** 包含点号（.）。每个 NetworkAttachmentDefinition 只能用于一个子网。不要在多个子网之间重复使用同一个 NetworkAttachmentDefinition。

以下示例使用 Kube-OVN underlay 子网作为外部网络。

NOTE

此示例假设用于 underlay 网络的 bridge network 和 VLAN 资源 `external-vlan` 已经创建。

```

apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: underlay-ext
  namespace: default
spec:
  config: |-
    {
      "cniVersion": "0.3.0",
      "type": "kube-ovn", ❶
      "server_socket": "/run/openvswitch/kube-ovn-daemon.sock", ❷
      "provider": "underlay-ext.default.ovn" ❸
    }
  ---
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: underlay-ext
spec:
  protocol: IPv4
  provider: underlay-ext.default.ovn ❹
  cidrBlock: 172.17.0.0/16 ❺
  gateway: 172.17.0.1 ❻
  vlan: external-vlan ❼
  excludeIps: ❽
    - 172.17.0.11..172.17.0.20

```

- ❶ secondary network 使用的 Kube-OVN CNI plugin。
- ❷ CNI plugin 使用的 Kube-OVN daemon socket。
- ❸ 形式为 `<network attachment definition name>.<namespace>.ovn` 的 provider 名称。
- ❹ 子网使用的 provider。此值 **MUST** 与 NetworkAttachmentDefinition 中的 provider 保持一致。
- ❺ 外部 underlay 网络的 CIDR。
- ❻ 外部 underlay 网络的 gateway。
- ❼ underlay 子网使用的 VLAN 资源。
- ❽ 从自动分配中排除的 IP 范围。有关详细信息，请参阅 [使用 Kube-OVN Underlay 网络的子网自定义资源（CR）示例](#)。

TIP

在使用 underlay 子网之前，请确保物理网络、VLAN 和 bridge network 已正确准备。有关环境规划的详细信息，请参阅 [准备 Kube-OVN Underlay 物理网络](#)。

步骤 2：创建 VPC Egress Gateway

创建 VPC Egress Gateway 资源，并定义哪些工作负载应使用它。


```

apiVersion: kubeovn.io/v1
kind: VpcEgressGateway
metadata:
  name: gateway1
  namespace: default ❶
spec:
  replicas: 1 ❷
  internalSubnet: ovn-default ❸
  externalSubnet: underlay-ext ❹
  externalIPs: ❺
    - 172.17.0.11
    - 172.17.0.12
  resources: ❻
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 200m
      memory: 256Mi
      ephemeral-storage: 2Gi
  nodeSelector: ❼
    - matchExpressions:
      - key: kubernetes.io/hostname
        operator: In
        values:
          - node1
          - node2
  tolerations: ❸
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
  selectors: ❹
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: ns1
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: ns2
    podSelector:
      matchLabels:
        app: myapp
  policies: ❿
    - snat: true ⓫

```

```

subnets: 12
  - subnet1
- snat: false
ipBlocks: 13
  - 10.18.0.0/16

```

- 1 创建 VPC Egress Gateway 实例的 namespace。
- 2 VPC Egress Gateway 实例数量。
- 3 连接内部网络的 internal subnet。该子网 **MUST** 是同一 VPC 中的 overlay 子网，并且有足够的可用 IP 供网关实例使用。如果未指定，网关 Pod 将使用 VPC 的默认 internal subnet。
- 4 连接外部网络的 external subnet。
- 5 网关 Pod 在 underlay 网络上使用的 external IP。每个网关实例都会从该列表中分配一个 IP。这些 IP **MUST** 位于外部子网的 CIDR 内，并且应包含在子网的 `excludeIps` 范围中。建议预留 `.spec.replicas + 1` 个 IP，以便在边缘情况下网关 Pod 仍能获取到 IP。
- 6 每个 VPC Egress Gateway 实例的资源 requests 和 limits。如果未指定，将应用 VPC Egress Gateway controller 中定义的默认资源 requests 和 limits。
- 7 用于调度 VPC Egress Gateway 实例的 node selector。
- 8 用于调度 VPC Egress Gateway 实例的 toleration。
- 9 用于选择通过 VPC Egress Gateway 访问外部网络的 Pod 的 Namespace selector 和 Pod selector。
- 10 VPC Egress Gateway 的策略，包括要应用的 SNAT 和 subnets/ipBlocks。
- 11 是否为该策略启用 SNAT。
- 12 该策略适用的子网。
- 13 该策略适用的 IP block。

此示例在 `default` namespace 中创建一个名为 `gateway1` 的 VPC Egress Gateway。匹配这些 selector 和 policy 的流量会通过外部子网 `underlay-ext` 转发。在此示例中，包括：

- `ns1` namespace 中的 Pod
- 带有标签 `app: myapp` 的 `ns2` namespace 中的 Pod
- 与 `subnet1` 子网相关的流量
- 与 CIDR `10.18.0.0/16` 相关的流量

NOTE

匹配 `.spec.selectors` 的 Pod 始终会被网关执行 SNAT。

步骤 3：验证网关

创建网关后，确认它已准备就绪并按预期转发流量。

1. 检查资源状态

先检查基础资源状态：

```
$ kubectl get veg gateway1
```

NAME	VPC	REPLICAS	BFD ENABLED	EXTERNAL SUBNET	PHASE
gateway1	ovn-cluster	1	false	underlay-ext	Completed

然后检查详细的网关信息：

```
kubectl get veg gateway1 -o wide
```

NAME	VPC	REPLICAS	BFD ENABLED	EXTERNAL SUBNET	PHASE
gateway1	ovn-cluster	1	false	underlay-ext	Completed

READY	INTERNAL IPS	EXTERNAL IPS	WORKING NODES	AGE
true	["10.16.0.12"]	["172.17.0.11"]	["node1"]	82s

最后，验证网关工作负载是否正在运行：

```
$ kubectl get deployment -n default -l ovn.kubernetes.io/vpc-egress-gateway=gateway1
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
gateway1	1/1	1	1	4m40s


```
$ kubectl get pod -n default -l ovn.kubernetes.io/vpc-egress-gateway=gateway1 -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP
gateway1-b9f8b4448-76lhm	1/1	Running	0	4m48s	10.16.0.1
2 node1	<none>	<none>			

2. 检查网关 Pod 内部的网络

检查网关 Pod 内的 IP 地址、路由条目和 iptables 规则：


```

$ kubectl exec -n default 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 st
ate 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 s
tate 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 -n default 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 -n default 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 -n default gateway1-b9f8b4448-76lhm -c gateway -- ip route

```

```
show table 1000
default via 10.16.0.1 dev eth0

$ kubectl exec -n default 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
```

3. 确认流量转发

在网关 Pod 中抓包，以确认流量已通过网关转发：

```
$ kubectl exec -n default gateway1-b9f8b4448-76lhm -c gateway -- 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], protocol 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], protocol 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], protocol 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], protocol ICMP (1), length 84)
    172.17.0.1 > 10.17.0.9: ICMP echo reply, id 37989, seq 0, length 64
```

4. 确认 OVN 路由策略

会为所选流量自动创建 OVN Logical Router policies :

```
$ 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 1
00.64.0.4
30000                                ip4.dst == 172.18.0.3    reroute 1
00.64.0.3
30000                                ip4.dst == 172.18.0.4    reroute 1
00.64.0.2
29100                                ip4.src == $VEG.8ca38ae7da18.ipv4  reroute 1
0.16.0.12 ①
29100                                ip4.src == $VEG.8ca38ae7da18_ip4  reroute 1
0.16.0.12 ②
29000 ip4.src == $ovn.default.kube.ovn.control.plane_ip4  reroute 1
00.64.0.3
29000 ip4.src == $ovn.default.kube.ovn.worker2_ip4  reroute 1
00.64.0.2
29000 ip4.src == $ovn.default.kube.ovn.worker_ip4  reroute 1
00.64.0.4
29000 ip4.src == $subnet1.kube.ovn.control.plane_ip4  reroute 1
00.64.0.3
29000 ip4.src == $subnet1.kube.ovn.worker2_ip4  reroute 1
00.64.0.2
29000 ip4.src == $subnet1.kube.ovn.worker_ip4  reroute 1
00.64.0.4

```

- ① VPC Egress Gateway 用于转发与 *.spec.policies* 匹配流量的 Logical Router Policy。
- ② VPC Egress Gateway 用于转发与 *.spec.selectors* 匹配流量的 Logical Router Policy。

可选：启用多副本负载均衡

NOTE

在扩展副本之前，请确保在外部子网中准备了足够的外部 IP，并将它们指定到 *.spec.externalIPs* 中。

要启用 ECMP 负载均衡并水平扩展吞吐量，请增加 *.spec.replicas*：


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

```
$ kubectl get veg -n default gateway1
```

NAME	VPC	REPLICAS	BFD	ENABLED	EXTERNAL SUBNET	PHASE
gateway1	ovn-cluster	2	false		underlay-ext	Completed
	AGE					
	true	39m				

```
$ kubectl get pod -n default -l ovn.kubernetes.io/vpc-egress-gateway=gateway1 -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP
gateway1-b9f8b4448-76lhm	1/1	Running	0	40m	10.16.0.12
node1	<none>	<none>			
gateway1-b9f8b4448-zd4dl	1/1	Running	0	64s	10.16.0.13
node2	<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	1
00.64.0.4				
30000		ip4.dst == 172.18.0.3	reroute	1
00.64.0.3				
30000		ip4.dst == 172.18.0.4	reroute	1
00.64.0.2				
29100		ip4.src == \$VEG.8ca38ae7da18.ipv4	reroute	1
0.16.0.12, 10.16.0.13				
29100		ip4.src == \$VEG.8ca38ae7da18_ip4	reroute	1
0.16.0.12, 10.16.0.13				
29000		ip4.src == \$ovn.default.kube.ovn.control.plane_ip4	reroute	1
00.64.0.3				
29000		ip4.src == \$ovn.default.kube.ovn.worker2_ip4	reroute	1
00.64.0.2				
29000		ip4.src == \$ovn.default.kube.ovn.worker_ip4	reroute	1
00.64.0.4				
29000		ip4.src == \$subnet1.kube.ovn.control.plane_ip4	reroute	1
00.64.0.3				
29000		ip4.src == \$subnet1.kube.ovn.worker2_ip4	reroute	1
00.64.0.2				

```
29000 ip4.src == $subnet1.kube.ovn.worker_ip4 reroute 1
00.64.0.4
```

可选：配置带宽限制

要限制所选 namespace 或工作负载的带宽，请设置 `.spec.bandwidth.ingress` 和 `.spec.bandwidth.egress`。每个字段都接受以下格式之一：

- 非负整数或数字字符串，按 Mbps 解释。例如，`2000` 和 `"2000"` 都表示 2000 Mbps。
- 带有 `M`、`Mi`、`G` 或 `Gi` 后缀的 Kubernetes quantity 字符串，按 bit/s 解释。带有后缀时支持小数值。该值会向上取整到最接近的整数 Mbps。例如，`1Gi` 会变为 1074 Mbps。

将某个字段省略或设置为 `0`，即可使该方向不受限。限制适用于每个网关实例。

NOTE

如果 `.spec.replicas` 大于 1，每个副本都会获得所配置的限制。Quantity 字符串需要支持它们的 VPC Egress Gateway CRD 和 controller 版本。在滚动升级期间，请继续使用整数 Mbps 值，直到 Kube-OVN 升级完成。

以下示例将所选 namespace 的 ingress 带宽限制为 2 Gbps，egress 带宽限制为 4 Gbps：


```
apiVersion: kubeovn.io/v1
kind: VpcEgressGateway
metadata:
  name: gateway-bandwidth
  namespace: default
spec:
  replicas: 1
  internalSubnet: ovn-default
  externalSubnet: underlay-ext
  externalIPs:
    - 172.17.0.21
  bandwidth:
    ingress: 2G
    egress: 4G
  selectors:
    - namespaceSelector:
        matchExpressions:
          - key: kubernetes.io/metadata.name
            operator: In
            values:
              - ecommerce
              - ecommerce-test
```

在此示例中，方向是相对于所选 namespace 或工作负载而言的：`ingress` 限制从外部网络到它们的流量，`egress` 限制从它们到外部网络的流量。

可选：启用基于 **BFD** 的高可用

基于 BFD 的故障切换依赖于 VPC BFD LRP。请按以下顺序启用。

1. 在 **VPC** 上启用 **BFD Port**

首先，在 VPC 上启用 BFD Port：

```

apiVersion: kubeovn.io/v1
kind: Vpc
metadata:
  name: ovn-cluster
spec:
  bfdPort:
    enabled: true ❶
    ip: 10.255.255.255 ❷
    nodeSelector: ❸
      matchLabels:
        kubernetes.io/os: linux

```

- ❶ 是否启用 BFD Port。
- ❷ BFD Port 的 IP 地址，**MUST** 是一个有效 IP 地址，且不得与任何其他 IP/Subnet 冲突。
- ❸ 用于选择以 Active-Backup 模式运行 BFD Port 的节点的 node selector。

TIP

默认情况下存在 Vpc 资源 *ovn-cluster*。你可以直接编辑它以启用 BFD Port。

启用 BFD Port 后，会在 OVN Logical Router 上自动创建一个专用的 BFD LRP：

```

$ kubectl ko nbctl show ovn-cluster
router 0c1d1e8f-4c86-4d96-88b2-c4171c7ff824 (ovn-cluster)
  port bfd@ovn-cluster ❶
    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"]

```

- ❶ 在 OVN Logical Router 上创建的 BFD Port。

2. 在 VPC Egress Gateway 上启用 BFD

然后通过将 `.spec.bfd.enabled` 设置为 `true` 来在 VPC Egress Gateway 上启用 BFD :

```
apiVersion: kubeovn.io/v1
kind: VpcEgressGateway
metadata:
  name: gateway2
  namespace: default
spec:
  vpc: ovn-cluster ①
  replicas: 2
  internalSubnet: ovn-default ②
  externalSubnet: underlay-ext ③
  externalIPs: ④
    - 172.17.0.11
    - 172.17.0.12
    - 172.17.0.13
  bfd:
    enabled: true ⑤
    minRX: 100 ⑥
    minTX: 100 ⑦
    multiplier: 5 ⑧
  policies:
    - snat: true
      ipBlocks:
        - 10.18.0.0/16
```

- ① Egress Gateway 所属的 VPC。
- ② Egress Gateway 实例连接到的 internal subnet。
- ③ Egress Gateway 实例连接到的 external subnet。
- ④ 分配给 Egress Gateway 实例的 external IP。
- ⑤ 是否为 Egress Gateway 启用 BFD。
- ⑥ BFD 的最小接收间隔，单位为毫秒。
- ⑦ BFD 的最小发送间隔，单位为毫秒。
- ⑧ BFD 的 multiplier，用于确定在判定故障前允许丢失的报文数。

此示例创建一个名为 `gateway2` 的 VPC Egress Gateway，包含两个副本并启用 BFD。如果某个实例失败，BFD session 会断开，OVN 会将该路由标记为不可用，并将流量重定向到健康的实例。

故障切换检测时间取决于 BFD 设置。使用以下公式： $break\ time = (multiplier + 1) * max(minRX, minTX)$ 。在此示例配置下，故障切换检测大约为 500-600 ms。

在网络质量较差或链路抖动频繁的环境中，100 ms 的 BFD 间隔可能过于敏感。你可以将 `.spec.bfd.minRX` 和 `.spec.bfd.minTX` 增加到 500-1000 ms，以减少误判故障和频繁切换。提高这些值也会增加故障切换检测时间，因此请根据稳定性和故障检测延迟之间的需求平衡进行调整。

NOTE

在故障切换期间，现有连接可能会中断并需要重新建立。新的连接仍然可以正常建立。

3. 验证 BFD 状态

检查 VPC Egress Gateway 状态：


```
$ kubectl get veg -n default gateway2 -o wide
```

NAME	VPC	REPLICAS	BFD ENABLED	EXTERNAL SUBNET	PHASE
gateway2	ovn-cluster	2	true	underlay-ext	Completed
INTERNAL IPS	EXTERNAL IPS				
["10.16.0.12", "10.16.0.13"]	["172.17.0.11", "172.17.0.12"]				
KING NODES	AGE				
["node1", "node2"]	58s				

```
$ kubectl get pod -n default -l ovn.kubernetes.io/vpc-egress-gateway=gateway2 -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP
gateway2-fcc6b8b87-8lgvx	1/1	Running	0	2m18s	10.16.0.13
node2	<none>	<none>			
gateway2-fcc6b8b87-wmww6	1/1	Running	0	2m18s	10.16.0.12
node1	<none>	<none>			

```
$ kubectl ko nbctl lr-policy-list ovn-cluster
```

Routing Policies

Policy	Match	Action	Priority
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	1
00.64.0.4			
30000	ip4.dst == 172.18.0.3	reroute	1
00.64.0.3			
30000	ip4.dst == 172.18.0.4	reroute	1
00.64.0.2			
29100	ip4.src == \$VEG.8ca38ae7da18.ipv4	reroute	1
0.16.0.12, 10.16.0.13	bfd		
29100	ip4.src == \$VEG.8ca38ae7da18_ip4	reroute	1
0.16.0.12, 10.16.0.13	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	1
00.64.0.3			
29000	ip4.src == \$ovn.default.kube.ovn.worker2_ip4	reroute	1
00.64.0.2			
29000	ip4.src == \$ovn.default.kube.ovn.worker_ip4	reroute	1
00.64.0.4			
29000	ip4.src == \$subnet1.kube.ovn.control.plane_ip4	reroute	1
00.64.0.3			
29000	ip4.src == \$subnet1.kube.ovn.worker2_ip4	reroute	1

```
00.64.0.2
    29000          ip4.src == $subnet1.kube.ovn.worker_ip4  reroute  1
00.64.0.4

$ kubectl ko nbctl list bfd
 _uuid           : 223ede10-9169-4c7d-9524-a546e24bfab5
detect_mult      : 5
dst_ip           : "10.16.0.12"
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           : "10.16.0.13"
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
```

然后检查 BFD sessions :

```
$ kubectl exec -n default gateway2-fcc6b8b87-8lgvx -c bfdd -- bfdd-control status
There are 1 sessions:
Session 1
id=1 local=10.16.0.13 (p) remote=10.255.255.255 state=Up

$ kubectl exec -n default gateway2-fcc6b8b87-wmww6 -c bfdd -- bfdd-control status
There are 1 sessions:
Session 1
id=1 local=10.16.0.12 (p) remote=10.255.255.255 state=Up
```

NOTE

如果所有网关实例都处于下线状态，由 VPC Egress Gateway 处理的出站流量将被丢弃。

可能中断流量的操作

以下操作可能会短暂中断出站流量，因为它们会删除或重建网关实例：

1. 更改副本数量
2. 更改配置，例如 internal 或 external IP、node selector 或 BFD 设置
3. 如果未指定 `.spec.image`，则升级或降级 Kube-OVN
4. 手动删除 Egress Gateway Pod

其他资源

- [RFC 5880 - 双向转发检测 \(BFD\)](#) ↗

配置 Kube-OVN 网络以支持 Pod 多网络接口

本指南介绍如何配置 Kube-OVN，以便 Pod 可以使用多个网络接口。

借助 Multus CNI 和 Kube-OVN，你可以为 Pod 挂载辅助网络，并通过 Kube-OVN Subnet 和 IPAM 管理该网络的 IP 地址。

TIP

- 主网络接口和辅助网络接口都支持 IPv4、IPv6 和双栈网络。
- 辅助网络同时支持 underlay 和 overlay 网络。

目录

开始之前

安装 Multus CNI

部署 Multus CNI 插件

创建辅助网络

创建具有多个网络接口的 Pod

验证网络接口

方法 1：检查 Pod 注解

方法 2：从 Pod 内部检查网络接口

方法 3：检查 Kube-OVN IP 资源

可选配置

固定 IP

额外路由

为引用同一个 NetworkAttachmentDefinition 的所有 Pod 配置额外路由

为特定 Pod/工作负载配置额外路由

在 Pod 路由表中验证额外路由

开始之前

请确保满足以下条件：

- 集群使用 Kube-OVN 作为网络插件。
- 你具有安装集群插件和创建 Kubernetes 资源的权限。
- 已配置 `kubectl`，并且可以访问目标集群。

安装 Multus CNI

部署 Multus CNI 插件

1. 导航到 管理员 > **Marketplace** > 集群插件。
2. 搜索 "multus"，并选择 "**Alauda Container Platform Networking for Multus**"。
3. 如果插件尚未安装，点击右侧的三点菜单 (⋮) 并选择 安装。
4. 等待插件状态变为 "Installed"。

NOTE

Multus CNI 插件充当其他 CNI 插件与 Kubernetes 之间的中间件，使 Pod 能够拥有多个网络接口。

创建辅助网络

要为 Pod 添加辅助网络接口，首先为该网络创建一个 NetworkAttachmentDefinition。

示例：`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。

NetworkAttachmentDefinition 名称中 绝不能 包含点号（`.`）。每个 NetworkAttachmentDefinition 只能被一个子网使用。不要在多个子网之间重复使用同一个 NetworkAttachmentDefinition。

如果你使用 underlay 辅助网络，请将子网中的 `spec.vlan` 设置为目标 VLAN CR 名称。

```
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"
    }
```

应用资源：

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

然后为辅助网络接口创建一个 Kube-OVN 子网。

示例：`subnet.yml`

NOTE

`spec.provider` 必须与 NetworkAttachmentDefinition 中的 provider 保持一致。

```
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: subnet1
spec:
  cidrBlock: 172.22.0.0/16
  provider: attachnet.default.ovn
```

应用资源：

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

创建具有多个网络接口的 Pod

在辅助网络准备就绪后，创建一个 Pod，并在 `metadata.annotations` 中引用该 NetworkAttachmentDefinition。

NOTE

- 注解 `k8s.v1.cni.cncf.io/networks` 使用格式 `<NAMESPACE>/<NAME>`，其中 `<NAMESPACE>` 和 `<NAME>` 分别是对应 NetworkAttachmentDefinition 的命名空间和名称。
- 如需挂载多个辅助网络，请用逗号分隔指定多个值，例如 `default/attachnet,default/attachnet2`。

```
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
```

对于需要多个网络接口的工作负载，你也可以在 Deployment 或 StatefulSet 等更高级别控制器的 Pod 模板中指定网络挂载注解。示例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
      annotations:
        k8s.v1.cni.cncf.io/networks: default/attachnet
    spec:
      containers:
        - name: web
          image: nginx:latest
          ports:
            - containerPort: 80
```

验证网络接口

Pod 运行后，请使用以下方法验证结果。

方法 1：检查 Pod 注解

运行以下命令：

```
kubectl get pod pod1 -o yaml
```

检查输出中的相关注解：

metadata:

annotations:

```
attachnet.default.ovn.kubernetes.io/allocated: "true"
attachnet.default.ovn.kubernetes.io/cidr: 172.22.0.0/16
attachnet.default.ovn.kubernetes.io/gateway: 172.22.0.1
attachnet.default.ovn.kubernetes.io/ip_address: 172.22.0.23
attachnet.default.ovn.kubernetes.io/logical_router: ovn-cluster
attachnet.default.ovn.kubernetes.io/logical_switch: subnet1
attachnet.default.ovn.kubernetes.io/mac_address: 02:66:77:88:99:aa
attachnet.default.ovn.kubernetes.io/pod_nic_type: veth-pair
attachnet.default.ovn.kubernetes.io/routed: "true"
```

k8s.v1.cni.cncf.io/network-status: |-

```
[{
  "name": "kube-ovn",
  "interface": "eth0",
  "ips": [
    "10.24.0.18/16"
  ],
  "mac": "02:11:22:33:44:55",
  "default": true,
  "dns": {},
  "gateway": [
    "10.24.0.1"
  ]
},{
  "name": "default/attachnet",
  "interface": "net1",
  "ips": [
    "172.22.0.23"
  ],
  "mac": "02:66:77:88:99:aa",
  "dns": {}
}]
```

```
k8s.v1.cni.cncf.io/networks: default/attachnet
ovn.kubernetes.io/allocated: "true"
ovn.kubernetes.io/cidr: 10.24.0.0/16
ovn.kubernetes.io/gateway: 10.24.0.1
ovn.kubernetes.io/ip_address: 10.24.0.18/16
ovn.kubernetes.io/logical_router: ovn-cluster
ovn.kubernetes.io/logical_switch: ovn-default
ovn.kubernetes.io/mac_address: 02:11:22:33:44:55
ovn.kubernetes.io/pod_nic_type: veth-pair
ovn.kubernetes.io/routed: "true"
```


注解 `k8s.v1.cni.cncf.io/network-status` 会显示该 Pod 所有网络接口的状态，包括主接口和辅助接口。在本示例中，Pod 有两个网络接口：`eth0` 是主接口，`net1` 是辅助接口。

方法 2：从 Pod 内部检查网络接口

运行以下任一命令：

```
kubectl exec pod1 -- ifconfig
```

```
kubectl exec pod1 -- ip addr show
```

NOTE

可用命令取决于容器镜像。如果镜像中某个命令不可用，请使用另一个命令。

`ip addr show` 的示例输出：

```

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 02:11:22:33:44:55 brd ff:ff:ff:ff:ff:ff
    inet 10.24.0.18/16 brd 10.24.255.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::11:22ff:fe33:4455/64 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 02:66:77:88:99:aa brd ff:ff:ff:ff:ff:ff
    inet 172.22.0.23/16 brd 172.22.255.255 scope global net1
        valid_lft forever preferred_lft forever
    inet6 fe80::66:77ff:fe88:99aa/64 scope link
        valid_lft forever preferred_lft forever

```

方法 3：检查 Kube-OVN IP 资源

运行以下命令，验证两个接口的 IP 分配记录：

```
kubectl get ip | grep -w pod1.default
```

示例输出：

NAME	V4IP	V6IP	MAC
pod1.default	10.24.0.18		02:11:22:33:44:55
node-1 ovn-default			
pod1.default.attachnet.default.ovn	172.22.0.23		02:66:77:88:99:aa
node-1 subnet1			

可选配置

固定 IP

- 主网络接口：如需为主接口分配固定 IP 地址，请为 Pod 添加注解
`ovn.kubernetes.io/ip_address=<IP>`。
- 辅助网络接口：如需为辅助接口分配固定 IP 地址，请将注解键中的 `ovn` 替换为对应的 NetworkAttachmentDefinition provider。示例：

```
attachnet.default.ovn.kubernetes.io/ip_address=172.22.0.101
```

TIP

对于 Deployment 或 StatefulSet 等工作负载，你可以在 Pod 模板中指定固定 IP 注解。

额外路由

你可以为引用同一个 NetworkAttachmentDefinition 的所有 Pod，或者为特定 Pod/工作负载配置额外路由。

为引用同一个 **NetworkAttachmentDefinition** 的所有 **Pod** 配置额外路由

要为引用同一个 NetworkAttachmentDefinition 的所有辅助网络接口配置额外路由，请在 NetworkAttachmentDefinition 配置中添加 `routes` 字段。示例：

```
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": "192.168.10.0/24",
        "gw": "172.22.0.101"
      }, {
        "dst": "192.168.20.0/24",
        "gw": "172.22.0.102"
      }]
    }
```

NOTE

请将路由网关值替换为在你的环境中可从辅助网络访问到的有效地址。

为特定 Pod/工作负载配置额外路由

要为特定辅助网络接口配置额外路由，请为 Pod 添加注解

`attachnet.default.ovn.kubernetes.io/routes`。示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  annotations:
    k8s.v1.cni.cncf.io/networks: default/attachnet
    attachnet.default.ovn.kubernetes.io/routes: |-
      [{
        "dst": "192.168.10.0/24",
        "gw": "172.22.0.101"
      }]
spec:
  containers:
    - name: web
      image: nginx:latest
      ports:
        - containerPort: 80
```

对于 Deployment 或 StatefulSet 等工作负载，你也可以在 Pod 模板中指定 routes 注解。示例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
      annotations:
        k8s.v1.cni.cncf.io/networks: default/attachnet
        attachnet.default.ovn.kubernetes.io/routes: |-
          [{
            "dst": "192.168.10.0/24",
            "gw": "172.22.0.101"
          }]
    spec:
      containers:
        - name: web
          image: nginx:latest
          ports:
            - containerPort: 80
```

在 Pod 路由表中验证额外路由

配置路由后，请在 Pod 内部验证实际生效的路由条目：

```
kubectl exec pod1 -- ip route show
```

或者，如果容器镜像中没有 `ip`，请使用 `route -n`：

```
kubectl exec pod1 -- route -n
```

`ip route show` 的示例输出：

```
default via 10.24.0.1 dev eth0
10.24.0.0/16 dev eth0 proto kernel scope link src 10.24.0.2
172.22.0.0/16 dev net1 proto kernel scope link src 172.22.0.101
192.168.10.0/24 via 172.22.0.101 dev net1
```

如果输出中没有出现相应的目的地址和网关路由，请检查 `NetworkAttachmentDefinition` 或 Pod 注解中的 `routes` 配置，然后重新创建 Pod 以重新应用 CNI 配置。

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。这在网络审计、IP 白名单和防火墙规则管理等安全运维场景中尤其有用，因为这些场景需要识别并控制来自特定源 IP 的流量。在集中式网关模式下，来自 Pods 的所有出站流量都会经由指定的网关节点转发，从而实现统一的网络策略强制执行和监控。

NOTE

集中式子网下的 Pods 不能通过 `hostport` 或 `externalTrafficPolicy: Local` 的 NodePort 类型 Service 访问。

如果你希望子网内的流量以固定 IP 访问外部网络，用于审计和白名单等安全运维场景，可以将 Subnet 中的网关类型设置为集中式。在集中式网关模式下，来自 Pods 访问外部网络的数据包会先被路由到特定节点的 `ovn0` 网卡，然后通过主机的路由规则向外转发。当 `natOutgoing` 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`。


```
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
```

- 如果集中式网关需要为出站网络指定某台机器的特定网卡，`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` spec 字段，用于将 ECMP 开关迁移到子网级别。你可以根据不同的子网设置是否启用 ECMP 模式。`kube-ovn-controller` deployment 中的 `enable-ecmp` 参数已不再使用。在升级到 v1.12.0 后，之前版本的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，5s 内检测故障，并在 5s-10s 内完成故障切换，在此期间部分流量可能会失败。

在集中式网关主备模式下，故障切换基于 Node Ready 状态，在断电情况下可能需要几分钟才能完成切换。

目录

使用 Label 选择器指定网关节点

使用 Label 选择器指定网关节点

除了直接指定节点名称外，你还可以使用 `gatewayNodeSelectors` 通过 label 选择器动态选择网关节点。这种方式更加灵活，尤其适用于节点名称不固定，或者需要根据 label 动态选择网关的场景。

NOTE

- 如果 `gatewayNode` 不为空，则优先使用 `gatewayNode`，并忽略 `gatewayNodeSelectors`。
- 多个选择器之间采用 OR 逻辑进行评估——匹配任意一个选择器的节点都会成为网关节点。
- 当节点 label 发生变化时，系统会自动更新网关节点列表。

```
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
```

[Alauda Container Platform](#) > [网络](#) > [实用指南](#) > [Kube OVN](#) > [配置 IPPool](#)

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：

```

apiVersion: kubeovn.io/v1
kind: IPPool
metadata:
  name: pool-1
spec:
  subnet: ovn-default ❶
  ips: ❷
  - "10.16.0.201"
  - "10.16.0.210/30"
  - "10.16.0.220..10.16.0.230"
  namespaces: ❸
  - ns-1

```

- ❶ IP 池所属的子网。
- ❷ IP 范围。支持的格式：`<IP>`、`<CIDR>` 和 `<IP1>..<IP2>`。同时支持 IPv4 和 IPv6。
- ❸ 可选，IP 池绑定的命名空间。已绑定命名空间中的 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即可。

当在已绑定命名空间中创建 Pod 时，其 IP 将从相应的 IP 池中分配。

NOTE

在将 IP 池绑定到命名空间之前，请确保该命名空间仅绑定了 IP 池所属的子网。

你也可以通过 annotation 将 IP 池分配给 Pod：

```
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 workload，请在 Deployment、StatefulSet 等的 Pod 模板中使用 annotation：

```
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. 允许使用超出子网范围的 IP 地址，但这些 IP 不会生效。
3. 属于同一子网的不同 IP 池不能具有重叠的 IP 范围。

4. 可以根据需要随时更新 `.spec.ips` 字段，任何更改都会立即生效。
5. IP 池会继承子网的保留 IP。在从 IP 池中随机分配 IP 地址时，会跳过 IP 池范围内的保留 IP。
6. 在从子网中随机分配 IP 地址时，将排除该子网中所有 IP 池的 IP 范围。
7. 多个 IP 池可以绑定到同一个命名空间。

配置 MTU

在 Kube-OVN 中，MTU（Maximum Transmission Unit，最大传输单元）设置对于确保最佳网络性能并避免数据包分片至关重要。MTU 定义了可在网络上传输的数据包最大大小。

默认情况下，Kube-OVN 会检测底层物理网络接口的 MTU，并据此设置虚拟网络接口的 MTU。不过，在某些场景下，你可能需要为 Kube-OVN 的网络组件自定义 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 会自动检测宿主机上物理网络接口的 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 以适配 VXLAN 或 Geneve 封装开销。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
	IPv6	物理网络接口 MTU - 120
VXLAN	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 会设置为与物理网络接口 MTU 一致。

自定义 MTU 设置

你可以针对覆盖网络全局自定义 MTU，也可以按子网单独配置。

WARNING

不正确地调整 MTU 设置可能会导致网络性能问题，包括数据包丢失和分片。请在进行更改之前确保 MTU 设置与底层网络基础设施兼容。

增加 MTU 时的关键注意事项

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

为什么需要这样做？

OVS 内部端口（例如 `ovn0`）会自动将与 `br-int` 桥连接的所有接口中的最小 **MTU** 作为自己的 MTU。该行为在以下场景中会带来潜在问题：

1. 你为新 Pod 配置了更大的 MTU 值
2. 一些现有 Pod 仍然使用原先较小的 MTU
3. 由于最小 MTU 规则，`ovn0` 接口仍保留较小的 MTU
4. 当数据包超过 `ovn0` 的 MTU 时，使用更大 MTU 的 Pod 发出的流量会被丢弃

解决方案：在提高 MTU 设置后，重新创建所有 Pod，以确保整个网络路径上的 MTU 配置一致。

NOTE

MTU 配置变更只会对新创建的 Pod 生效。现有 Pod 会保留原始 MTU 设置，直到它们被重新创建。建议在修改 MTU 设置时同时规划 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，默认情况下 MTU 仍与物理网络接口 MTU 保持一致。

按子网配置 MTU

你也可以通过在子网配置中指定 `mtu` 字段来为单个子网设置 MTU。例如，要将某个特定子网的 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 设置。

Alauda Container Platform > 网络 > 实用指南 > Kube OVN > 配置 BGP

配置 BGP

Kube-OVN 可以使用 BGP 将 Pod 和 Subnet 路由通告到外部网络。要使用此功能，请在选定的（或所有）节点上安装 `kube-ovn-speaker`，并为需要从集群外部访问的 Pod 或 Subnet 添加 `ovn.kubernetes.io/bgp` 注解。

`kube-ovn-speaker` 使用 [GoBGP](#) 发布路由，并将下一跳设置为自身。

目录

安装 `kube-ovn-speaker`

发布 Pod 和 Subnet 路由

通告策略

BGP 高级选项

安装 `kube-ovn-speaker`

运行 `kube-ovn-speaker` 的节点必须能够承载返回流量。请为要部署 speaker 的节点添加标签：

```
kubectl label nodes speaker-node-1 ovn.kubernetes.io/bgp=true
kubectl label nodes speaker-node-2 ovn.kubernetes.io/bgp=true
```

NOTE

当运行多个 `kube-ovn-speaker` 实例时，每个实例都会向上游发布路由。上游路由器必须支持多路径 ECMP。

从与你的集群 Kube-OVN 版本匹配的 tag 或 branch 下载 `speaker.yaml`（与 `kube-ovn` 镜像 tag 保持一致），例如：

```
wget https://raw.githubusercontent.com/kubeovn/kube-ovn/release-1.15/yamls/speaker.yaml
```

根据需要调整清单中的 BGP 参数。

如果你只有一个上游对等体：

```
- --neighbor-address=10.32.32.254
- --neighbor-ipv6-address=2409:AB00:AB00:2000::AFB:8AFE
- --neighbor-as=65030
- --cluster-as=65000
```

如果你有一对交换机：

```
- --neighbor-address=10.32.32.252,10.32.32.253
- --neighbor-ipv6-address=2409:AB00:AB00:2000::AFB:8AFC,2409:AB00:AB00:2000::AFB:8AFD
- --neighbor-as=65030
- --cluster-as=65000
```

- `neighbor-address`：BGP 对等体地址，通常为路由器网关。
- `neighbor-as`：BGP 对等体的自治系统号。
- `cluster-as`：容器网络的自治系统号。

应用该清单：

```
kubectl apply -f speaker.yaml
```

发布 Pod 和 Subnet 路由

对于 Subnet 的 BGP 导出，请将该 Subnet 上的 `natOutgoing` 设置为 `false`，以便 Pod IP 可以直接在底层网络上使用。

使用注解启用路由通告：

```
kubectl annotate pod sample ovn.kubernetes.io/bgp=true
kubectl annotate subnet ovn-default ovn.kubernetes.io/bgp=true
```

删除这些注解以停止通告：

```
kubectl annotate pod sample ovn.kubernetes.io/bgp-
kubectl annotate subnet ovn-default ovn.kubernetes.io/bgp-
```

有关注解使用不同值时的行为，请参见 [通告策略](#)。

通告策略

`kube-ovn-speaker` 支持两种策略：

- **Cluster**：即使某个节点上没有运行具有该 IP 的 Pod，也会由每个 speaker 通告 Pod IP 和 Subnet CIDR。外部流量可以进入任何运行 speaker 的节点，然后在集群内部转发。可能会产生额外跳数。Pod 和 Subnet 的默认策略均为此项。
- **Local**：仅由实际承载这些 Pod 的节点上的 speaker 通告 Pod IP（或其 Subnet 被标记为 BGP 的 Pod）。外部流量会落到工作负载运行的节点，从而缩短路径。

NOTE

对于 **Local** 策略，通常需要在每个节点上部署 `kube-ovn-speaker`。如果 Pod 调度到没有 speaker 的节点上，其 IP 将不会被通告。

可通过 `ovn.kubernetes.io/bgp` 注解为单个 Pod 或 Subnet 覆盖该策略：

- `ovn.kubernetes.io/bgp=cluster`，或默认的 `ovn.kubernetes.io/bgp=yes`，选择 **Cluster** 策略。
- `ovn.kubernetes.io/bgp=local` 选择 **Local** 策略。

BGP 高级选项

`kube-ovn-speaker` 为复杂环境支持以下额外参数：

- `auth-password`：BGP 对等体的密码。
- `holdtime`：BGP 保持时间；在此时间间隔内没有流量的邻居将被移除（默认 90 秒）。
- `graceful-restart`：启用 BGP Graceful Restart。
- `graceful-restart-time`：Graceful Restart 时间（RFC 4724 第 3 节）。
- `graceful-restart-deferral-time`：Graceful Restart 延迟时间（RFC 4724 第 4.1 节）。
- `passivemode`：speaker 仅接受连接（被动模式）。
- `ebgp-multihop`：EBGP 多跳对等体的 TTL（默认 1）。

Underlay 和 Overlay 子网的自动互联

如果一个集群同时具有 Underlay 和 Overlay 子网，默认情况下，Overlay 子网中的 Pod 可以通过使用 NAT 的网关访问 Underlay 子网中 Pod 的 IP。但是，Underlay 子网中的 Pod 需要配置节点路由才能访问 Overlay 子网中的 Pod。

要实现 Underlay 和 Overlay 子网之间的自动互联，可以手动修改 Underlay 子网的 YAML 文件。配置完成后，Kube-OVN 还会使用一个额外的 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。

目录

操作步骤

可选：将 U2O 路由限制为 Overlay CIDR

启用 u2oInterconnection 后的 Underlay 子网隔离

第 1 步：配置 kube-ovn-controller

第 2 步：配置子网隔离

操作步骤

1. 进入 管理员。
2. 在左侧导航栏中，点击 集群管理 > 资源管理。
3. 输入 **Subnet** 以过滤资源对象。

4. 点击待修改的 Underlay 子网旁边的：`>` 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 子网中现有的计算组件需要重新创建后，变更才会生效。

可选：将 U2O 路由限制为 Overlay CIDR

如果只想让目标地址属于 Overlay 子网 CIDR 的流量通过 U2O 网关路由，可以在同一个 Underlay 子网上启用 `u2oFeatures.overlayOnlyRouting`。发往其他目的地的流量仍将继续使用物理网关。

```
spec:
  u2oInterconnection: true
  u2oFeatures:
    overlayOnlyRouting: true
```

此选项要求启用 `u2oInterconnection: true`。

启用 u2oInterconnection 后的 Underlay 子网隔离

当多个 Underlay 子网都启用了 `u2oInterconnection: true` 时，它们之间的流量将不再经过物理网关，而是直接通过内部 OVN 网络路由。

如果你需要在两个都启用了 `u2oInterconnection` 的 Underlay 子网之间进行隔离，则必须先配置 kube-ovn-controller 参数，然后再配置子网隔离。

第 1 步：配置 kube-ovn-controller

修改 kube-ovn-controller Deployment，以禁用对目标逻辑端口 IP 的 connection tracking 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` 控制是否跳过发往逻辑端口 IP 的流量的 connection tracking (conntrack)。默认值为 `true`，会跳过 conntrack 以提升性能。将其设置为 `false` 不会影响功能，但可能会对性能产生轻微影响。

不过，对于基于 ACL 的隔离的 Underlay 子网，你必须将其设置为 `false`。否则，网关到 Pod 的流量将失败（例如，ping 请求到达 Pod，但回复被丢弃），因为 ACL 隔离使用 `allow-related`，它需要 conntrack 状态；如果没有 conntrack，就无法将回复识别为“related”，从而被丢弃。

第 2 步：配置子网隔离

使用以下参数配置子网：

```
spec:
  u2oInterconnection: true
  acls:
    - action: drop
      direction: to-lport # Ingress direction (traffic entering the logical port)
      match: ip4.src == 172.20.0.0/16
      priority: 1002
    - action: drop
      direction: to-lport # Ingress direction
      match: ip4.src == 192.50.0.0/16
      priority: 1002
```

ACL 参数：

参数	描述
<code>action</code>	要执行的操作： <code>allow</code> 、 <code>drop</code> 或 <code>allow-related</code>
<code>direction</code>	流量方向： <code>to-lport</code> （入方向）或 <code>from-lport</code> （出方向）
<code>match</code>	使用 L2-L4 字段和布尔运算符的 OVN 匹配表达式
<code>priority</code>	规则优先级（数值越大越先评估；推荐范围：1002-1899）

NOTE

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

配置 Endpoint Health Checker

目录

概述

主要功能

安装

通过 Marketplace 安装

工作原理

健康检查机制

核心功能

健康检查流程

性能提升

如何启用

Pod 级注解（推荐）

对于 ALB

对于 IngressNginx

对于 EnvoyGateway

对于自定义部署

Pod 级 readinessGates（旧版）

卸载

概述

Endpoint Health Checker 是一个集群插件，用于监控和管理 k8s 集群中 service endpoint 的健康状态。它会自动从 service 中移除不健康的 endpoint，确保流量仅路由到健康的实例，从而提升整体服务的可靠性和可用性。

主要功能

- 自动健康监控：持续监控 k8s 集群中 service endpoint 的健康状态
- 负载均衡集成：自动从 service 中移除不健康的 endpoint
- 服务可用性：确保流量仅定向到健康、可用的 endpoint
- 快速故障切换：将节点断电期间的 endpoint 切换时间从 40 秒缩短到 10 秒

安装

通过 Marketplace 安装

1. 导航到 管理员 > **Marketplace** > 集群插件。
2. 在插件列表中搜索 "**Alauda Container Platform Endpoint Health Checker**"。
3. 单击 安装，打开安装配置页面。
4. 在部署配置对话框中，你可以选择性地配置以下参数：

参数	描述
节点选择器	配置标签选择器，以指定 Endpoint Health Checker 组件应运行在哪些节点上。单击 添加 可添加多个标签键值对。
节点容忍度	配置容忍度，以允许将 Endpoint Health Checker 组件调度到具有特定污点的节点上。单击 添加 可添加多个包含 Key、Value 和 Type 的容忍度。

5. 单击 安装 部署插件。
6. 等待插件状态变为 "**Ready**"。

工作原理

健康检查机制

Endpoint Health Checker 是一个专用的健康监控组件，可确保只有健康的 endpoint 才能接收流量。它通过监控 service endpoint，并自动管理其可用性状态来运行。

核心功能

Endpoint Health Checker 的工作方式如下：

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：监控为 service endpoint 提供后端支持的 pod 的 readiness 和 liveness 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** 连通性检查：建立 TCP 连接以验证端口可访问性
4. **Endpoint** 管理：自动从 service endpoint 列表中移除不健康的 endpoint，以防止流量路由到失效实例

健康检查流程

健康检查流程包括：

- 探针集成：利用 Kubernetes readiness 和 liveness 探针结果作为初始健康指标
- 网络连通性：向目标 endpoint 端口发送 TCP 数据包以验证可访问性
- 响应校验：评估响应状态、时延和内容，以确定 endpoint 的健康状况
- 自动故障切换：从 service endpoint 列表中移除无响应或失败的 endpoint

性能提升

- 旧方式：依赖 kubelet 心跳检测，最多有 40 秒延迟
- 当前方式：主动 endpoint 健康检查，检测和切换时间为 10 秒
- 提升：显著提高了 ALB + MetalLB 环境中节点故障期间的服务可用性

如何启用

健康检查可通过两种方式启用：

Pod 级注解（推荐）

对于 ALB

为 `ALB2` 设置 `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": "true"}'
    name: demo-alb
spec:
  config:
    loadbalancerName: demo-alb
    nodeSelector:
      ingress: 'true'
    replicas: 1
  type: nginx
```

对于 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'
```

对于 EnvoyGateway

按照 [如何在 Envoy Gateway 中添加 Pod 注解](#) 中的说明进行操作，将注解设置为 `endpoint-health-checker.io/enabled: 'true'`。

对于自定义部署

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


```
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)

为旧版本在 pod spec 中配置 readinessGates :

```

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。对于新的部署，建议使用 pod 注解 `endpoint-health-checker.io/enabled: 'true'`。

卸载

要卸载 Endpoint Health Checker：

1. 导航到 管理员 > **Marketplace** > 集群插件。
2. 找到已安装的 "**Endpoint Health Checker**" 插件。
3. 单击选项菜单并选择 卸载。
4. 在提示时确认卸载。

[Alauda Container Platform](#) > [网络](#) > [实用指南](#) > [alb](#) > ALB 任务

ALB 任务

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

```
# example of nodeSelector and tolerations
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

```
kubect1 patch alb2 $NAME -n $MS --type='merge' -p '{
  "metadata": {
    "annotations": {
      "alb.cpaas.io/toleration": "[{\"key\": \"node-role.kubernetes.io/infra\", \"operator\": \"Equal\", \"value\": \"reserved\", \"effect\": \"NoSchedule\"}]"
    }
  },
  "spec": {
    "config": {
      "nodeSelector": {
        "node-role.kubernetes.io/infra": ""
      }
    }
  }
}'
```

任务：从 OCP Route 迁移到 GatewayAPI Route

目录

简介

前提条件

基本 HTTP Route

 OCP Route 配置

 Gateway API 配置

Route 超时

 OCP Route 配置

 Gateway API 配置

HTTP Strict Transport Security (HSTS)

 OCP Route 配置

 Gateway API 配置

基于 Cookie 的会话亲和性

 OCP Route 配置

 Gateway API 配置

基于路径的路由

 OCP Route 配置

 Gateway API 配置

Header 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 Allowlist/Blocklist

OCP Route 配置

Gateway API 配置

URL 重写

OCP Route 配置

Gateway API 配置

跨命名空间 Route 接纳

OCP Route 配置

Gateway API 配置

Ingress 的默认 TLS 证书

OCP Route 配置

Gateway API 配置

使用自定义 CA 的 TLS Re-encrypt

OCP Route 配置

Gateway API 配置

使用自定义证书的 Edge Termination

OCP Route 配置

Gateway API 配置

TLS Passthrough

OCP Route 配置

Gateway API 配置

功能对比摘要

迁移策略

相关文档

简介

使用 Envoy Gateway 将 OpenShift Container Platform (OCP) Routes 迁移到 Kubernetes Gateway API HTTPRoutes。使用下面按功能划分的映射，将每个 OCP Route 功能转换为等效的 Gateway API 配置。

前提条件

1. [配置 EnvoyGatewayCtl](#)
2. [配置 Gateway](#)
3. 对 [Gateway API Routes](#) 有基本了解

基本 HTTP Route

OCP Route 配置

在 OCP 中，基本 HTTP route 使用 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 该 route 的 hostname
- ② 后端 service 引用
- ③ 后端 service 上的目标端口

Gateway API 配置

在 Gateway API 中，对应的配置使用 HTTPRoute：

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames: ❶
  - example.com
  parentRefs: ❷
  - name: demo-gateway
    namespace: demo
  rules:
    - backendRefs: ❸
      - name: example-service
        port: 8080
```

- ❶ 此 route 接受的 hostnames（相当于 OCP Route 的 `host`）
- ❷ 对 Gateway listener 的引用
- ❸ 后端 service 和端口

Route 超时

OCP Route 配置

在 OCP 中，超时通过注解进行配置：

```

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 请求的通用超时
- ② 隧道连接 (WebSocket、HTTP/2 等) 的超时

Gateway API 配置

在 Gateway API 中，超时在 HTTPRoute 规则中配置：

```

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 [请求超时](#)。

NOTE

Gateway API 没有专门针对隧道连接的单独超时。 `request` 超时适用于所有连接类型。

HTTP Strict Transport Security (HSTS)

OCP Route 配置

在 OCP 中，HSTS 通过 edge terminated 或 re-encrypt route 上的注解进行配置：

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

1 HSTS header 配置

Gateway API 配置

在 Gateway API 中，HSTS 通过响应 header 修改进行配置：

```

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 将 HSTS header 添加到响应中

更多详情请参见 [HTTP Header Modification](#)。

NOTE

不同于 OCP 的 `requiredHSTSPolicies` 集群级强制执行，Gateway API 需要按 route 显式配置。Gateway API 中没有全局 HSTS policy 强制机制。

基于 Cookie 的会话亲和性

OCP Route 配置

在 OCP 中，会话亲和性通过注解进行配置：

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/balance: source ❶
    haproxy.router.openshift.io/disable_cookies: "false" ❷
    router.openshift.io/cookie_name: my-cookie ❸
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

- ❶ 负载均衡算法
- ❷ 启用基于 cookie 的会话亲和性
- ❸ 用于会话持久化的 cookie 名称

Gateway API 配置

在 Gateway API 中，会话持久化在 HTTPRoute 规则中配置：

```
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`（跨浏览器会话持久存在）或 `Session`（浏览器关闭时失效）

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：

```

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 用于匹配请求的路径前缀

Gateway API 配置

在 Gateway API 中，路径匹配在 HTTPRoute 规则中配置：

```

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（相当于 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](#)。

Header 修改

OCP Route 配置

在 OCP 中，header 修改使用注解：

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/response-set-header: X-Custom-Header:valu
e ①
    haproxy.router.openshift.io/request-set-header: X-Request-Header:valu
e ②
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

- ① 设置响应 headers
- ② 设置请求 headers

Gateway API 配置

在 Gateway API 中，header 修改使用 filters 配置：

```

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 ❶
          requestHeaderModifier:
            add:
              - name: X-Request-Header
                value: value
        - type: ResponseHeaderModifier ❷
          responseHeaderModifier:
            add:
              - name: X-Custom-Header
                value: value
      backendRefs:
        - name: example-service
          port: 8080

```

❶ 请求 header 修改

❷ 响应 header 修改

更多详情请参见 [HTTP Header Modification](#)。

连接限制

OCP Route 配置

在 OCP 中，连接限制通过注解进行配置：

```

apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/pod-concurrent-connections: "100" ❶
spec:
  host: example.com
  to:
    kind: Service
    name: example-service

```

❶ 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 进行配置：

```

apiVersion: gateway.envoyproxy.io/v1alpha1
kind: ClientTrafficPolicy
metadata:
  name: connection-limit-policy
  namespace: demo
spec:
  targetRefs: ❶
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo-gateway
  connection: ❷
    connectionLimit:
      value: 100

```

❶ 将 policy 附加到 Gateway

❷ 连接限制配置

更多详情请参见 [连接限制](#)。

NOTE

不同于 OCP 按每个后端 pod 限制连接数，Gateway API 的连接限制应用于 Gateway 级别，并在各个 Envoy proxy 实例之间分配。

速率限制

OCP Route 配置

在 OCP 中，速率限制使用注解：

```
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: "10"
    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 进行配置：

```
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，并支持本地和全局两种速率限制机制。

IP Allowlist/Blocklist

OCP Route 配置

在 OCP 中，IP allowlisting 使用注解：

```

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 allowlist 配置

Gateway API 配置

在 Gateway API 中，IP filtering 通过带有 authorization 规则的 SecurityPolicy 实现：

```

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 policy 附加到 HTTPRoute

② 默认拒绝的 authorization 配置

③ 使用 CIDR 表示法的 IP allowlist

对于 IP blocklist (denylist) , 将 `defaultAction: Allow` 并使用 `action: Deny` :

```

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 拒绝来自特定 IP 的流量

更多详情请参见 [IP Allowlist/Denylist](#)。

NOTE

如果你的 Gateway 位于负载均衡器或 proxy 之后，请确保使用 ClientTrafficPolicy 正确配置客户端 IP 检测。

URL 重写

OCP Route 配置

在 OCP 中，URL 重写使用注解：

```
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 filters 配置：


```
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 filter

更多详情请参见 [URL Rewrite](#)。

跨命名空间 Route 接纳

OCP Route 配置

在 OCP 中，route admission policy 控制不同命名空间中的 routes 是否可以声明相同的 hostname。这通过 Ingress Operator 在集群级别配置。

Gateway API 配置

在 Gateway API 中，跨命名空间访问在 Gateway listener 级别进行控制：

```
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 # Allow routes from all namespaces
```

① 配置哪些命名空间可以附加 routes

`allowedRoutes.namespaces.from` 的可选值：

- **Same**：仅允许与 Gateway 位于同一命名空间中的 routes
- **All**：允许来自任意命名空间的 routes
- **Selector**：允许来自匹配 label selector 的命名空间中的 routes

更多详情请参见 [Cross-Namespace Routing](#)。

NOTE

不同于 OCP 的集群级接纳 policy，Gateway API 在单个 Gateway listener 级别进行控制，从而提供更细粒度的控制能力。

Ingress 的默认 TLS 证书

OCP Route 配置

在 OCP 中，没有 TLS 配置的 routes 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 listener 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 该 listener 的默认 TLS 证书

附加到此 listener 且没有特定 TLS 配置的任何 HTTPRoute 都将使用该默认证书。

使用自定义 CA 的 TLS Re-encrypt

OCP Route 配置

在 OCP 中，re-encrypt routes 会在 router 处终止 TLS，并在验证后重新加密到后端：

```
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 Re-encrypt 终止模式
- 2 用于验证后端的 CA certificate

Gateway API 配置

在 Gateway API 中，后端 TLS 验证通过 BackendTLSPolicy 进行配置：


```

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 ❶
        certificateRefs:
          - name: frontend-tls
---
apiVersion: gateway.networking.k8s.io/v1
kind: BackendTLSPolicy
metadata:
  name: backend-tls-policy
  namespace: demo
spec:
  targetRefs: ❷
    - group: ""
      kind: Service
      name: example-service
  validation: ❸
    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 将 policy 应用于后端 Service
- 3 后端 TLS 验证配置

更多详情请参见 [Backend TLS](#)。

NOTE

CA certificate 必须存储在 ConfigMap 中，而不是 Secret 中。

使用自定义证书的 Edge Termination

OCP Route 配置

在 OCP 中，edge termination 使用 route 的 TLS 配置：

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: edge ❶
    certificate: | ❷
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    key: | ❸
      -----BEGIN PRIVATE KEY-----
      ...
      -----END PRIVATE KEY-----
```

- ❶ Edge termination 模式
- ❷ TLS certificate
- ❸ TLS private key

Gateway API 配置

在 Gateway API 中，TLS termination 在 Gateway listener 上配置：


```

apiVersion: v1
kind: Secret
metadata:
  name: example-tls
  namespace: demo
type: kubernetes.io/tls ❶
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 ❷
      port: 443
      tls:
        mode: Terminate ❸
        certificateRefs: ❹
          - 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 protocol
- 3 Terminate TLS 模式
- 4 对 TLS secret 的引用

TLS Passthrough

OCP Route 配置

在 OCP 中，passthrough routes 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 Passthrough 终止模式

Gateway API 配置

在 Gateway API 中，TLS passthrough 使用 TLSRoute：

```

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 ❶
      port: 443
      tls:
        mode: Passthrough ❷
  ---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TLSRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
      sectionName: tls-passthrough
  hostnames: ❸
    - example.com
  rules:
    - backendRefs:
        - name: example-service
          port: 8443

```

- ❶ TLS protocol
- ❷ Passthrough 模式
- ❸ 用于路由的 SNI hostname

更多详情请参见 [TLS Passthrough](#) 和 [TLSRoute specification](#)。

NOTE

TLSRoute 使用 SNI (Server Name Indication) 进行路由决策，而不终止 TLS。这意味着：

- 无法访问 HTTP headers 或请求内容来做路由决策

- 无法应用任何 L7 filters（header 修改、URL 重写等）
- 路由完全基于 SNI hostname

功能对比摘要

功能	OCP Route	Gateway API
基本 HTTP 路由	Route spec	HTTPRoute
基于路径的路由	<code>.spec.path</code>	<code>.spec.rules[].matches[].path</code>
超时	注解	HTTPRoute <code>.spec.rules[].timeouts</code>
HSTS	注解	ResponseHeaderModifier filter
会话亲和性	注解	HTTPRoute <code>.spec.rules[].sessionPersistence</code>
Header 修改	注解	RequestHeaderModifier/ResponseHeaderModifier filters
连接限制	注解	ClientTrafficPolicy
速率限制	注解	BackendTrafficPolicy

功能	OCP Route	Gateway API
IP Allowlist/Blocklist	注解	带有 authorization 的 SecurityPolicy
URL 重写	注解	URLRewrite filter
跨命名空间	集群级 policy	Gateway <code>.spec.listeners[].allowedRoutes</code>
默认 TLS 证书	controller 级	Gateway listener TLS
TLS Re-encrypt	<code>.spec.tls.termination: reencrypt</code>	BackendTLSPolicy
TLS Edge	<code>.spec.tls.termination: edge</code>	带有 HTTPS 的 Gateway listener TLS
TLS Passthrough	<code>.spec.tls.termination: passthrough</code>	具有 Passthrough 模式的 TLSRoute

迁移策略

从 OCP Routes 迁移到 Gateway API 时：

- 从 **Gateway** 开始：创建一个带有适合你使用场景的 listener 的 Gateway 资源
 - 将 Gateway 部署在同一命名空间或专用的 gateway 命名空间中
 - 为你需要的所有 protocol 配置 listeners (HTTP、HTTPS、TLS、TCP、UDP)
 - 确保 Gateway service 已创建并具有可访问的 endpoint
- 将 **Routes** 转换为 **HTTPRoutes**：将每个 OCP Route 迁移为一个 HTTPRoute，先从简单 routes 开始

- 从非生产环境或低流量 routes 开始，以将风险降到最低
- 验证 hostname 和 path 匹配配置
- 在添加高级功能之前先测试基本连通性

3. 应用 **Policies**：对于需要 policies（BackendTrafficPolicy、SecurityPolicy、ClientTrafficPolicy）的功能，在基本 route 工作后再创建并附加它们

- 一次添加一个 policy 并验证功能
- 监控任何意外行为或性能影响

4. 逐步测试：在进入下一个 route 之前，先验证每一步迁移

- 监控：检查 Gateway 和 Route 的 status conditions，查看是否有错误
- 功能测试：使用 curl 或自动化测试验证所有 routes 是否按预期工作
- 性能测试：将响应时间和吞吐量与 OCP Routes 进行比较
- 验证检查：
 - 验证 TLS certificates 是否正确应用
 - 测试会话亲和性和负载均衡行为
 - 验证速率限制和 security policies
 - 检查 header 修改和 URL 重写

5. 更新 **DNS**：验证完成后，更新 DNS 记录以指向新的 Gateway service

- 双运行期间（推荐）：在保持 OCP Routes 仍然启用以便回滚的同时，将部分流量指向新的 Gateway
- 使用加权 DNS 或金丝雀部署逐步切换流量
- 在切换期间监控错误率、延迟和应用指标

6. 回滚流程：如果迁移后发现问题

- 将 DNS 回退到指向 OCP Routes
- 在 DNS 切换后至少保留 OCP Routes 24-48 小时
- 记录导致问题的任何配置差异
- 如需回滚，为相关方准备沟通计划

相关文档

- [配置 GatewayAPI Gateway](#)
- [配置 GatewayAPI Route](#)
- [配置 GatewayAPI Policy](#)
- [Envoy Gateway 的任务](#)



[Alauda Container Platform](#) > [网络](#) > 故障排除

故障排除

如何解决 **ARM** 环境中的节点间通 查找错误原因

如何解决 ARM 环境中的节点间通信问题？

在使用较低版本的 kernel 和某些国产网卡时，启用 Checksum Offload 后可能会出现网卡校验和计算错误的问题。这会导致 Kube-OVN Overlay 网络中的节点间通信失败。具体解决方案如下：

- 方案 1：升级 **kernel** 版本。建议将 kernel 版本升级到 4.19.90-25.16.v2101 或更高版本。
- 方案 2：禁用 **Checksum Offload**。如果无法立即升级 kernel 版本，并且出现节点间通信问题，可以使用以下命令禁用物理网卡的 Checksum Offload。

```
ethtool -K eth0 tx off
```

查找错误原因

错误请求的响应头中的 `X-ALB-ERR-REASON` 字段将指示错误原因。

错误原因可能是：

```
InvalidBalancer : no balancer found for xx # it means no endpoint found f
or the service
BackendError : read xxx byte data from backend # it means the backend did
give response, the error code is not cause by alb.
InvalidUpstream : no rule match # it means the request does not match any
rule, so alb return 404.
```