网络

概览

概览

了解网络

核心网络层和组件

GatewayAPI

Service、Ingress 和 Gateway API 的比较

网络流量流向

商标声明

网络操作器

Alauda Container Platform 负载

了解 MetalLB Operator

选择 VIP 承载节点的算法

自定义资源定义

约束和限制

安装和卸载 MetalLB Operator

升级 MetalLB Operator

Alauda Container Platform Ingr

概述

安装

通过 Web 控制台配置

通过 YAML 配置

其他资源

Alauda Con

概述

通过 Envoy Gat

下一步

商标声明

商标声明

商标声明

ALB Operat

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

注意事项

使用 Web 控制台创建 AdminNetworkPolic

使用 CLI 创建 AdminNetworkPolicy 或 Bas

网络策略

通过 Web 控制

通过 CLI 创建 N

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

故障排除

商标声明

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

配置服务

为什么需要 Ser

ClusterIP 类型 S

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 Alauda Container Platform 负载均衡器

前提条件

配置 MetalLB 组件

配置 Gatewa

配置 GatewayAPI 策略

概述

前提条件

策略附加基础

策略附加摘要

在 Web Console 中创建 Policy

SecurityPolicy

BackendTLSPolicy

ClientTrafficPolicy

BackendTrafficPolicy

础知

ay

ay 设

Rou

相关任务
商标声明

配置 EnvoyProxy 资源配额
概述
配额模型
创建配额
配置 EnvoyProxy 请求
准入行为
监控 namespace 使用量
范围和限制
下一步
商标声明

配置 Alauda Container Platform 集群 DNS
概述
配置
商标声明

配置 GatewayAPI ListenerSet
概述
先决条件

配置 ALB
ALB

配置 Alauda Container Platform 节点 DNS 缓存
概述
主要功能

商标声明

网络可观测性

网络可观测性

实用指南

Ingress-Nginx 任务

前提条件

最大连接数

请求超时

会话亲和性（粘性会话）

修改标头

URL 重写

HSTS（HTTP Strict Transport Security）

速率限制

WAF

Forward-header 控制

HTTPS

保留源 IP

商标声明

Envoy Gateway 任务

概述

前提条件

高级任务

相关文档

更多配置

商标声明

Kube OVN

配置 Endpoint Health Checker

概述

主要功能

安装

工作原理

如何激活

卸载

商标声明

Gateway AP

概述

ACP 版本映射

商标声明

Soft Data C

前提条件

alb

任务：从 OC

简介

前提条件

基本 HTTP Rou

Route 超时

HTTP 严格传输

基于 Cookie 的

基于路径的路由

标头修改

连接限制

限流

IP 允许列表/阻止

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 透传
功能对比摘要
迁移策略
相关文档
商标声明

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

了解网络
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
Gateway API 用于 Ingress
Gateway API 概念
Service、Ingress 和 Gateway API 的比较
对于 L4（TCP/UDP）流量
LoadBalancer Service（推荐）
Gateway API TCP/UDP Route
建议
对于 L7（HTTP/HTTPS）流量
Ingress
Gateway API（推荐）
网络流量流向
商标声明

了解网络

在云原生环境中，网络的核心作用是充当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。

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：自动发现正在运行的微服务实例，并在这些实例之间智能分配流量。

实现：

- 服务发现
  - Pod 向中央注册表注册（例如 `etcd`）
  - Service 查询注册表以查找可用实例
  - Kubernetes `CoreDNS` 通过 Service 名称提供内置发现功能
- 负载均衡
  - Kubernetes `Service` 资源会获得虚拟 IP（VIP）
  - 充当后端 Pod（Endpoints）的内置负载均衡器
  - 在各实例之间均匀分配请求

流量管理与路由

职责：通过细粒度规则控制流量，从而支持高级部署模式。

实现：

- **Ingress**
  - 集群处理外部 HTTP/HTTPS 流量的“入口”
  - 基于主机名或路径的路由
  - SSL 终止

- **Service Mesh**（Istio、Linkerd）
  - 金丝雀发布：将一定比例的流量定向到新版本
  - 故障注入：模拟服务故障以进行韧性测试
  - 超时、重试与熔断：提高容错能力
  - 流量镜像：将生产流量复制到测试环境

网络安全

职责：为经过授权的服务通信强制实施“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**）
  - 加密并验证所有服务间通信
  - 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，使运维人员能够了解流量模式，并快速诊断连接或性能问题。

实现：

- 持续可视性
  - 使用 NetObserv 收集整个集群的网络流日志
  - 使用 Kubernetes 元数据丰富流记录
  - 存储并查询数据，以便进行故障排查和审计

- 按需故障排查
  - 使用 CLI 捕获流或数据包，以进行短期分析
  - 支持在 TUI 模式下实时检查，或执行后台捕获任务
  - 导出结果，供 Wireshark 等离线分析工具使用
- 典型使用场景
  - 识别被丢弃的连接、延迟热点和意外的流量路径
  - 验证服务间通信并检测异常
  - 结合汇总流和原始数据包调查事件

详细了解网络可观测性。

核心网络层和组件

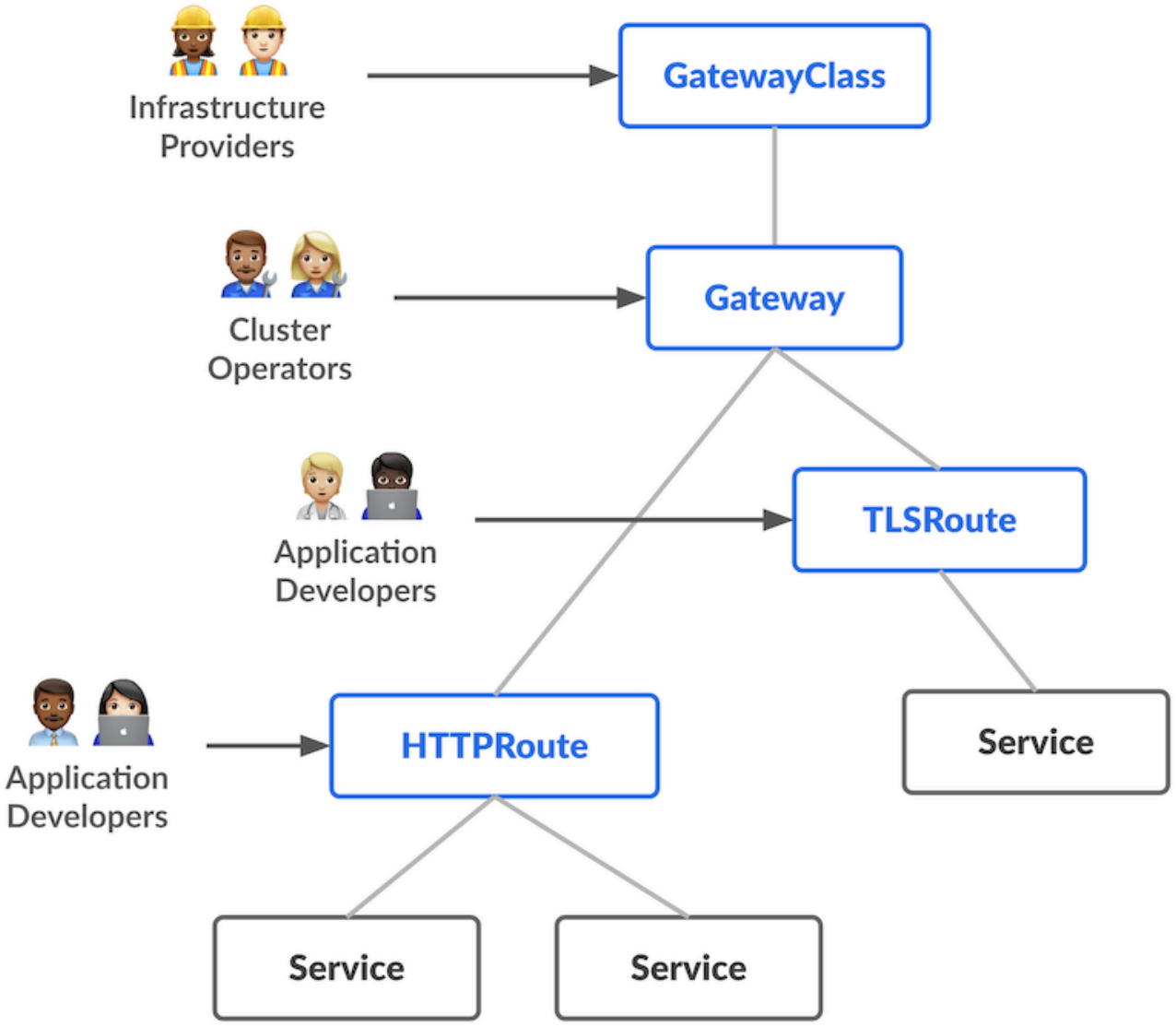
容器网络是一套为云原生应用设计的综合网络解决方案，可确保集群内东西向通信顺畅，并高效管理跨外部网络的南北向流量，同时提供必要的网络功能。它由以下核心组件组成：

- 用于管理集群内东西向流量的容器网络接口（CNI）。
- 用于管理 HTTPS Ingress 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，它还提供完善的网络安全和加密功能，确保通信安全。

GatewayAPI

Gateway API 是 Kubernetes 官方项目，专注于 Kubernetes 中的 L4 和 L7 路由。该项目代表了新一代 Kubernetes Ingress、负载均衡和 Service Mesh API。从一开始，它就被设计为通用、表达力强且面向角色。

整体资源模型聚焦于 3 种独立的人员角色，以及预期由这些角色管理的相应资源：



此 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 提供高级路由功能。

Gateway API 用于 Ingress

使用 Gateway API 管理 Ingress 流量时，Gateway 资源定义了一个访问入口，流量可以通过该入口在多个上下文之间进行路由——例如，从集群外部路由到集群内部（南北向流量）。

每个 Gateway 都与一个 GatewayClass 相关联，GatewayClass 描述了实际处理该 Gateway 流量的网关控制器类型；随后，单独的路由资源（例如 HTTPRoute）会与 Gateway 资源相关联。将这些不同的关注点拆分到不同资源中，是 Gateway API 面向角色特性的重要组成部分，同时也支持多种网关控制器（由 GatewayClass 资源表示）。

Gateway API 概念

以下设计目标推动了 Gateway API 概念的形成。这些目标体现了 Gateway 如何在当前 Ingress 等标准的基础上进行改进。

- 面向角色 - Gateway 由 API 资源组成，这些资源对使用和配置 Kubernetes Service 网络的组织角色进行建模。
- 可移植 - 这并不是一项改进，而是应当保持不变的特性。正如 Ingress 是一种具有众多实现的通用规范一样，Gateway API 也被设计为一种可移植规范，并由多种实现提供支持。
- 表达力强 - Gateway API 资源支持基于标头的匹配、流量权重等核心功能，以及过去只有通过 Ingress 自定义注解才能实现的其他功能。
- 可扩展 - Gateway API 允许在 API 的各个层中关联自定义资源。这使得 API 结构中的适当位置能够进行细粒度定制。

其他值得关注的功能包括：

- GatewayClass - GatewayClass 规范化了负载均衡实现的类型。这些类让用户能够通过 Kubernetes 资源模型轻松、明确地了解可用的功能类型。
- 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，这些功能支持共享负载均衡器和虚拟 IP。这使团队（即使跨 Namespace）也能够安全地共享基础设施，而无需直接协调。
- 类型化 Route 和类型化后端 - Gateway API 支持类型化的 Route 资源以及不同类型的后端。这使 API 能够灵活支持各种协议（如 HTTP 和 gRPC）以及各种后端目标（如 Kubernetes Service、存储桶或函数）。

有关 Gateway API 的更详细说明，请参阅 gateway-api 文档↗。

Service、Ingress 和 Gateway API 的比较

灵雀云容器平台 支持 Kubernetes 生态系统中的多种 Ingress 流量规范。 请比较 Service、Ingress 和 Gateway API，以选择符合您的工作负载和运维要求的流量入口模式。

对于 L4（TCP/UDP）流量

LoadBalancer 类型的 Service 和 Gateway API（TCP/UDP Route）都可以将 L4 流量暴露到外部。不过，它们在实现方式和性能特征方面存在显著差异。

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）
- 开销极小，性能接近原生水平

优势：

- 高性能、低延迟
- 更低的 CPU 和内存开销
- 经过实践验证且成熟的技术

Gateway API TCP/UDP Route

实现方式：用户空间代理

- 由 Envoy Gateway（或其他网关控制器）实现
- 流量必须在内核空间与用户空间之间往返传递
- 应用层会产生额外的处理开销

劣势：

- 与内核空间解决方案相比，性能有所下降
- 更高的资源消耗（CPU/内存）
- 用户空间上下文切换会带来额外延迟

建议

对于 **L4** 流量路由，我们建议使用 **LoadBalancer** 类型的 **Service**，因为其性能更高且资源开销更低。

目前，我们通过 MetalLB 支持 LoadBalancer 类型的 Service。

对于 L7（HTTP/HTTPS）流量

Ingress 和 Gateway API 都可以将 L7 流量暴露到外部，但二者的功能有所不同。

Ingress

Ingress 是 Kubernetes Community 采用的标准规范，推荐默认使用。 Ingress 由平台管理员管理的 ALB 实例处理。

Gateway API（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、更灵活且更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
   - 关注点明确分离：基础设施管理员（GatewayClass、Gateway）与 Application developer（Routes）
   - Ingress 将基础设施和应用关注点混合在单个资源中
2. 表达力更强的路由能力
   - 丰富的匹配规则：HTTP 标头、查询参数、方法等
   - Ingress 仅限于主机和路径匹配
   - 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 透传
   - Ingress 主要仅关注 HTTP/HTTPS
4. 可扩展性
   - 通过 CRD 提供类型安全的扩展点（例如 HTTPRoute 过滤器、策略附加）
   - Ingress 严重依赖供应商特定的注解，从而导致可移植性问题
5. 跨 **Namespace** 路由
   - Route 可以引用不同 Namespace 中的 Service（通过 ReferenceGrant）
   - Ingress 通常仅限于引用同一 Namespace 中的资源
6. 每个 **Gateway** 支持多个 **Listener**
   - 单个 Gateway 可以处理多个端口、协议和主机名

- Ingress 通常要求每种配置使用一个资源

7. 可移植性和标准化

- 与供应商无关的 API，并提供一致性测试
- 减少供应商锁定，并提高不同实现之间的互操作性
- Ingress 实现的功能和注解存在较大差异

8. 附加和选择模型

- Route 通过 `parentRefs` 显式附加到 Gateway Listener
- 关系更加清晰，也更易于排查问题
- Ingress 使用灵活性较低的 IngressClass

目前，我们通过 Envoy Gateway 支持 Gateway API。有关从 Ingress 迁移到 Gateway API 的信息，请参阅迁移指南↗

网络流量流向

通过 Alauda Container Platform 集群的网络流量流向示例。

```
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)
```

流量说明：

1. 客户端（浏览器 **/ curl**）。
   用户向 `https://www.example.com` 发送 HTTPS 请求。 客户端在第 7 层运行，并发起 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。
   在第 4 层（TCP/UDP）运行。 它将传入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 Controller**。
   在第 7 层（应用层）运行。 处理：
   - TLS 终止
   - 基于主机名和路径的路由
   - 策略和身份验证

   它将流量路由到匹配的 Kubernetes Service。
5. **[L4] Kubernetes Service**（虚拟 **IP**）
   充当集群内部的第 4 层负载均衡器。 根据选择器将请求分发到后端 Pod。
6. **Pod**（原生应用）
   应用运行并处理请求的最终目的地。 响应沿反向路径返回客户端。

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

MetalLB™ 是 The Linux Foundation 的商标。

CoreDNS™ 是 The Linux Foundation 的商标。

Alauda 是一家独立供应商。本产品与 The Linux Foundation 或 F5, Inc. 不存在关联，也未获其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

网络操作器

Alauda Container Platform 负载均衡器

Alauda Container Platform 负载均衡器

了解 MetalLB Operator

选择 VIP 承载节点的算法

自定义资源定义

约束和限制

安装和卸载 MetalLB Operator

升级 MetalLB Operator

商标声明

Alauda Container Platform Ingress Controller

Alauda Container Platform Ingress Controller

概述

安装

通过 Web 控制台配置

通过 YAML 配置

其他资源

商标声明

Alauda Container Platform Gateway API Controller

Alauda Container Platform Gateway API Controller

概述

通过 Envoy Gateway Operator 安装 Envoy Gateway

下一步

商标声明

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：

商标声明

Auth

基本概念

快速入门

相关 Ingress 注解

forward-auth

basic-auth

CR

ALB 特殊 Ingress 注解

Ingress-Nginx 身份验证相关的其他功能

注意：与 Ingress-Nginx 不兼容的部分

为 ALB 部署

方法 1：使用 L

方法 2：使用外

商标声明

ALB 性能选

小型生产环境

境

ALB 中的负载均衡会话亲和策略

概述

可用算法

配置方式

最佳实践

CORS

基本概念

CRD

Header 修改

基本概念

示例

商标声明

OTel

术语

前提条件

操作步骤

相关操作

附加说明

配置示例

商标声明

HTTP 重定向

基本概念

ModSecurity

术语

操作步骤

相关说明

ALB 与 Ingress-NGINX 注解兼容性

基本概念

支持的 ingress-nginx 注解

商标声明

ALB 监控

术语

操作步骤

监控指标

境

署建

使用

TCP/HTTP

Alauda Container Platform 负载均衡器

目录

了解 MetalLB Operator
术语
MetalLB 的高可用性原理
选择 VIP 承载节点的算法
计算公式
应用示例
自定义资源定义
IPAddressPool
L2Advertisement
BGPPeer
BGPAdvertisement
BFDProfile（可选）
约束和限制
安装和卸载 MetalLB Operator
配置组件放置
调整日志级别
升级 MetalLB Operator
商标声明

了解 MetalLB Operator

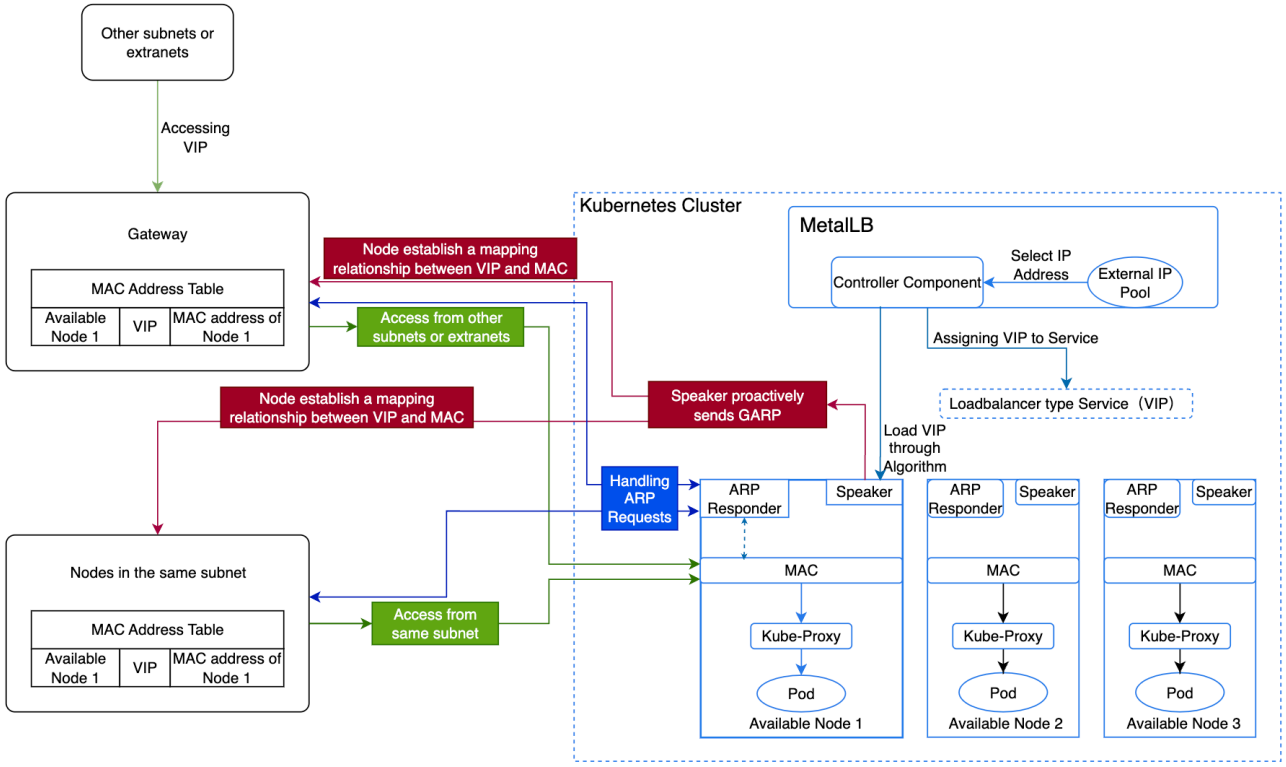
MetalLB Operator 为无法使用云负载均衡器服务的本地部署或裸金属环境提供 Kubernetes 原生负载均衡器实现。 它通过为 Service 资源分配外部 IP 地址，并通过第 2 层（ARP/NDP）或第 3 层（BGP）通告这些地址，使类型为 LoadBalancer 的 Service 资源能够正常工作。

Operator 会自动管理 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的服务提供统一的访问点。
ARP	地址解析协议（ARP）用于将网络层 IP 地址映射为数据链路层 MAC 地址。
GARP	免费 ARP（GARP）是一种特殊的 ARP 请求，用于通知网络中的其他节点某个 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，而是主动向网络发送信息。
ARP Responder	MetalLB 的一个组件，负责通过将 VIP 映射到节点的 MAC 地址来响应 ARP 请求。当节点需要与 VIP 通信时，会发送 ARP 请求，以获取与该 VIP 对应的 MAC 地址。每个可用节点都配有一个 ARP Responder，用于响应这些请求，将 VIP 映射到节点的 MAC 地址。
Controller	MetalLB 的一个组件，负责从外部地址池动态分配 VIP，用于 LoadBalancer 类型的内部路由。Controller 会监听集群中内部路由的创建和删除事件，并根据需要分配或释放 VIP。
Speaker	MetalLB 的一个组件，根据策略或算法确定节点是否应托管 VIP 并发送 GARP。它可确保节点之间达到一定程度的负载均衡；当某个节点不可用时，其他节点可以接管 VIP 并发送 GARP，从而实现高可用性。

MetalLB 的高可用性原理



默认情况下，平台使用 MetalLB 的 ARP 模式，其具体实现流程和原理如下：

- MetalLB 的 Controller 组件从外部地址池中选择一个 IP 地址，并将其分配给 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 来托管 VIP，然后由该节点转发流量。
- 该节点上的 Speaker 组件主动发送 GARP，在所有节点中建立 VIP 与 MAC 地址之间的映射关系。
  - 同一子网内的节点在学习到 VIP 与可用节点 MAC 地址之间的映射关系后，访问 VIP 时会直接与该节点通信。
  - 不同子网中的节点会先将流量路由到其子网的网关，再由网关将流量转发到托管 VIP 的节点。
- 当该节点发生故障时，MetalLB 会选择另一个 leader 来托管 VIP。随后发送 GARP 以刷新 Service IP 的 MAC 地址，从而确保高可用性。
- 流量到达节点后，Kube-Proxy 会将其转发到相应的 Pod。

选择 VIP 承载节点的算法

给定负载均衡器 IP 的“leader”（将要通告该 IP 的节点）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，同时考虑活跃的 speaker、外部流量策略、活跃端点、节点选择器等因素。
- 每个 speaker 执行相同的计算：获取“节点+VIP”元素哈希值的排序列表，如果当前 speaker 是列表中的第一项，则通告该 Service。

这样便无需记录哪个 speaker 负责通告给定 IP。

计算公式

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

注意：如果使用虚拟机，则虚拟机数量 = 外部地址池数量 * n。此处 n 必须大于 2，且最多允许一个节点发生故障。

- n-vip：表示 VIP 的数量。
- n-node：表示单个节点可以处理的 VIP 数量。

应用示例

如果一家公司有 10 个 VIP，每个可用节点可以处理 5 个 VIP，并且允许一个节点发生故障，那么该公司应如何规划外部地址池数量和可用节点数量？

分析：

总共需要两个外部地址池和四个可用节点。

- 每个可用节点最多可以处理 5 个 VIP，这意味着一个外部地址池可以容纳 5 个 VIP，因此 10 个 VIP 需要两个外部地址池。
- 允许一个节点发生故障意味着每个地址池必须包含一个承载 VIP 的节点和一个备用节点，因此两个外部地址池各需要两个可用节点。

自定义资源定义

MetalLB Operator 会在 metallb.io API 组下安装并管理多个 CRD。 这些 CRD 定义了 MetalLB 分配和通告 IP 地址的方式。

CRD	Kind	用途
`IPAddressPool`	`IPAddressPool`	定义可供负载均衡器 Service 使用的 IP 地址池。
`L2Advertisement`	`L2Advertisement`	配置如何使用 ARP/NDP（第 2 层）通告地址池中的 IP。
`BGPPeer`	`BGPPeer`	定义供 MetalLB 建立路由会话的 BGP 对等端（第 3 层）。
`BGPAdvertisement`	`BGPAdvertisement`	指定通过 BGP 通告哪些 IPAddressPool 以及所使用的策略。
`BFDProfile`	`BFDProfile`	可选；定义双向转发检测参数，以更快检测 BGP 故障。

IPAddressPool

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

```yaml
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 或范围。

2 默认情况下，MetalLB 会从任何已配置且有可用地址的地址池中分配 IP。这可能会导致将“昂贵”的地址分配给不需要这些地址的 Service。 若要阻止此行为，可以将 autoAssign 标志设置为 `false` ，以禁用地址池的自动分配。

L2Advertisement

用于第 2 层模式，在此模式下，MetalLB 通过 ARP/NDP 通告 IP。

```yaml
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: l2-adv
  namespace: metallb-system
spec:
  ipAddressPools: (1)
    - production-pool
```

(1) 将通告其 IP 的地址池列表。

BGPPeer

```yaml
apiVersion: metallb.io/v1beta1
kind: BGPPeer
metadata:
  name: peer1
  namespace: metallb-system
spec:
  peerAddress: 10.0.0.1 (1)
  peerASN: 64512 (2)
  myASN: 64513 (3)
  holdTime: 90s (4)
  keepaliveTime: 30s
  nodeSelectors: (5)
    - matchLabels:
        alertmanager: 'true'
```

(1) 对等路由器的 IP 地址。

(2) 对等端的 ASN。

(3) MetalLB 使用的 ASN。

(4) `holdTime` / `keepaliveTime`：BGP 会话的可选计时器。

(5) 可以在配置中使用对等端的 node-selectors 属性，将对等端限制到特定节点。

BGPAdvertisement

控制如何通过 BGP 通告地址池中的 IP。

```yaml
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 community 属性。

BFDProfile（可选）

用于定义 BGP 对等端的 BFD（双向转发检测）参数。

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

约束和限制

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 必须可供平台使用，且不能包含物理网络中已在使用的 IP，例如网关 IP。
- 不得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，确保同一个外部地址池中同时存在 IPv4 和 IPv6 地址，且两者的数量都大于 0。否则，双栈 LoadBalancer 类型的内部路由将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，无法成功部署 Alauda Container Platform Load Balancer 插件。

安装和卸载 **MetalLB Operator**

可以通过 Alauda Container Platform Web 控制台中的 Cluster Plugin 安装 Operator。

1. 前往 `Administrator` -> `Marketplace` -> `Cluster Plugin`
2. 输入关键字 `Alauda Container Platform Load Balancer`。
3. 点击 `⋮`，然后点击 安装。
4. 安装完成后，点击 `⋮`，然后点击 卸载。

配置组件放置

安装 Alauda Container Platform Load Balancer Cluster Plugin 后，直接在 `MetalLB` 自定义资源中配置组件放置，该资源位于 `metallb-system` 命名空间中。如果该资源不存在，Cluster Plugin 会在安装时创建；controller 和 speaker 的调度设置由此资源管理。

```yaml
apiVersion: metallb.io/v1beta1
kind: MetalLB
metadata:
  name: metallb
  namespace: metallb-system
spec:
  controllerNodeSelector:
    dedicated: ingress
  controllerTolerations:
    - key: dedicated
      operator: Equal
      value: ingress
      effect: NoSchedule
  nodeSelector:
    dedicated: ingress
  speakerTolerations:
    - key: dedicated
      operator: Equal
      value: ingress
      effect: NoSchedule
```

`controllerNodeSelector` 和 `controllerTolerations` 适用于 MetalLB controller Deployment。`nodeSelector` 和 `speakerTolerations` 适用于 MetalLB speaker DaemonSet。默认情况下，这些字段为空，因此 MetalLB 不会容忍任何额外的污点。

在 Layer 2 模式下，`L2Advertisement.spec.nodeSelectors` 配置决定哪些 speaker 可以通告 VIP。请使其与 MetalLB 组件放置以及 `IngressNginx.spec.controller.nodeSelector` 保持一致，以便 speaker 能够从运行 Ingress Nginx pod 的节点通告 VIP。

调整日志级别

在 `MetalLB` 自定义资源上设置 `spec.logLevel`，以调整 controller 和 speaker 使用的日志级别：

```yaml
spec:
  logLevel: warn
```

支持的值包括 `all`、`debug`、`info`、`warn`、`error` 和 `none`。默认值为 `info`。

升级 **MetalLB Operator**

Alauda Container Platform Load Balancer 是一个 `Core` 插件。它会随集群一起升级，不支持独立升级，因此无需为其下载或上传软件包。有关插件生命周期类型，请参阅 Core and Extensions。

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 或 F5, Inc. 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

Alauda Container Platform Ingress Controller

目录

概述
安装
通过 Web 控制台配置
通过 YAML 配置
Ingress 分片
创建处理所有命名空间中 Ingress 的 IngressNginx
创建处理特定命名空间中 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
其他资源
与官方 Chart 的默认值差异
商标声明

概述

Ingress Nginx Operator 用于简化 Ingress-Nginx 部署的高级管理。 此 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 会持续监控类型为 `ingressnginxes.ingress-nginx.alauda.io` (IngressNginx) 的自定义资源，并为每个已定义的 IngressNginx 资源自动创建对应的 Ingress-Nginx 实例。通过此架构，可以直接操作 IngressNginx 自定义资源，以控制 Ingress-Nginx 的功能和行为。

安装

1. 前往 `Administrator -> Marketplace -> OperatorHub`
2. 找到 `Alauda Container Platform Ingress Controller`，然后点击 `Install`

通过 **Web** 控制台配置

我们通过用户界面提供基本配置选项，并为你生成相应的 YAML。对于更复杂的配置，你可以直接编辑 YAML。

完成 Ingress Nginx Operator 的安装后：

1. 前往 `All Instances` 选项卡
2. 点击 `Create`，然后在显示的对话框中找到 IngressNginx Instance Type 部分，并点击 `Create`

字段	描述	**YAML** 路径
名称	Ingress Nginx 实例的名称	`.metadata.name`
命名空间	Ingress Nginx 实例的命名空间	`.metadata.namespace`
从节点	Ingress Nginx 部署的副本数	`.spec.controller.replicaCount`
资源	Ingress Nginx 部署的资源请求和限制	`.spec.controller.resources`

字段	描述	YAML 路径
服务类型	Ingress Nginx 的服务类型	`.spec.controller.service.type`
Ingress 范围	控制要处理哪些命名空间中的 Ingress 资源	`.spec.controller.scope.namespaceSelector`，请参阅 ingress-sharding

通过 YAML 配置

默认情况下，Ingress Nginx Operator 会在相应 IngressNginx 自定义资源所在的同一命名空间中部署 Ingress Nginx 实例。

Ingress 分片

默认情况下，对于显式指定其 IngressClass 名称或未指定任何 IngressClass 的 Ingress，Ingress Nginx 会处理所有命名空间中的 Ingress 资源。你可以使用 `.controller.scope.namespaceSelector` 将实例限制在特定命名空间中。例如，以下 IngressNginx 资源 `demo-scope` 仅处理带有标签 `cpaas.io/project=demo` 的命名空间中的 Ingress。

创建处理所有命名空间中 Ingress 的 IngressNginx

```yaml
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中 Ingress 的 IngressNginx

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

(1) 格式类似于 `$LABLE_KEY=$LABLE_VALUE`

通过 **LoadBalancer** 暴露

默认情况下，Ingress Controller 配置为 ClusterIP Service 类型。若要使用 LoadBalancer Service 在外部暴露 Ingress Controller，请应用以下配置：

> **NOTE**
>
> LoadBalancer Service 需要外部负载均衡器集成（云提供商 LB 或 MetalLB）来分配外部 IP。

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

使用 MetalLB 时指定 LoadBalancer VIP：

```yaml
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' # Desired VIP
        metallb.universe.tf/address-pool: 'pool-name' # MetalLB address p
ool
```

高级 **Controller** 部署设置

配置网络模式、副本、资源限制、节点选择和污点容忍：

```yaml
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
spec:
  controller:
    hostNetwork: false
    replicaCount: 1
    nodeSelector:
      dedicated: ingress
    tolerations:
      - key: dedicated
        operator: Equal
        value: ingress
        effect: NoSchedule
    resources:
      limits:
        cpu: 200m
        memory: 256Mi
      requests:
        cpu: 200m
        memory: 256Mi
```

`nodeSelector` 和 `tolerations` 设置控制 Ingress Nginx controller pod 的调度位置。默认情况下，Ingress Nginx controller 不容忍任何污点。

SSL 透传

启用 SSL 透传功能：

> **CAUTION**
>
> 启用 SSL 透传后，TLS 会在后端终止，因此 L7 功能（例如请求/响应标头操作、WAF、HTTP 到 HTTPS 重定向以及某些身份验证流程）不会应用于 controller 上的此类流量。

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

IPv4 和 IPv6 双栈支持

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

其他资源

`IngressNginx` 资源的 `.spec` 字段直接对应 Ingress Nginx 的 Helm Chart values，包括 `.spec.controller.nodeSelector` 和 `.spec.controller.tolerations`。有关其他配置选项，请参阅官方 Ingress NGINX 文档↗。

与官方 **Chart** 的默认值差异

1. 默认情况下，每个 IngressNginx 实例都会创建一个名称为 `$ns-$name` 且 controllerValue 为 `ingress-nginx.cpaas.io/$ns-$name` 的 IngressClass。可以使用 `.spec.controller.ingressClassResource.name` 和 `.spec.controller.ingressClassResource.controllerValue` 参数自定义这些值。
2. 默认情况下，`.spec.controller.service.type` 设置为 `ClusterIP`。
3. 默认情况下，`.spec.controller.watchIngressWithoutClass` 设置为 `true`，这意味着 controller 将处理未指定 IngressClass 的 Ingress 资源。

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 或 F5, Inc. 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

Alauda Container Platform Gateway API Controller

目录

概述
架构
相关自定义资源
Operator 相关的自定义资源
`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：使用推荐的默认值创建 EnvoyGatewayCtl
通过 YAML 配置
通过 EnvoyGatewayCtl 进行高级配置
示例：启用后端路由
下一步
商标声明

概述

使用 `envoy-gateway-operator` 通过平台管理的资源部署和管理 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` 将上游 `envoy-gateway` Helm chart 打包为 Operator，因此您可以通过 `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 chart 或渲染后的资源。

在 ACP 4.3 中，`envoy-gateway-operator` 打包了上游 `envoy-gateway` Helm chart 版本 `1.7.0`。

有关各 ACP 版本安装的 Gateway API CRD 版本，请参阅Gateway API CRD 版本。

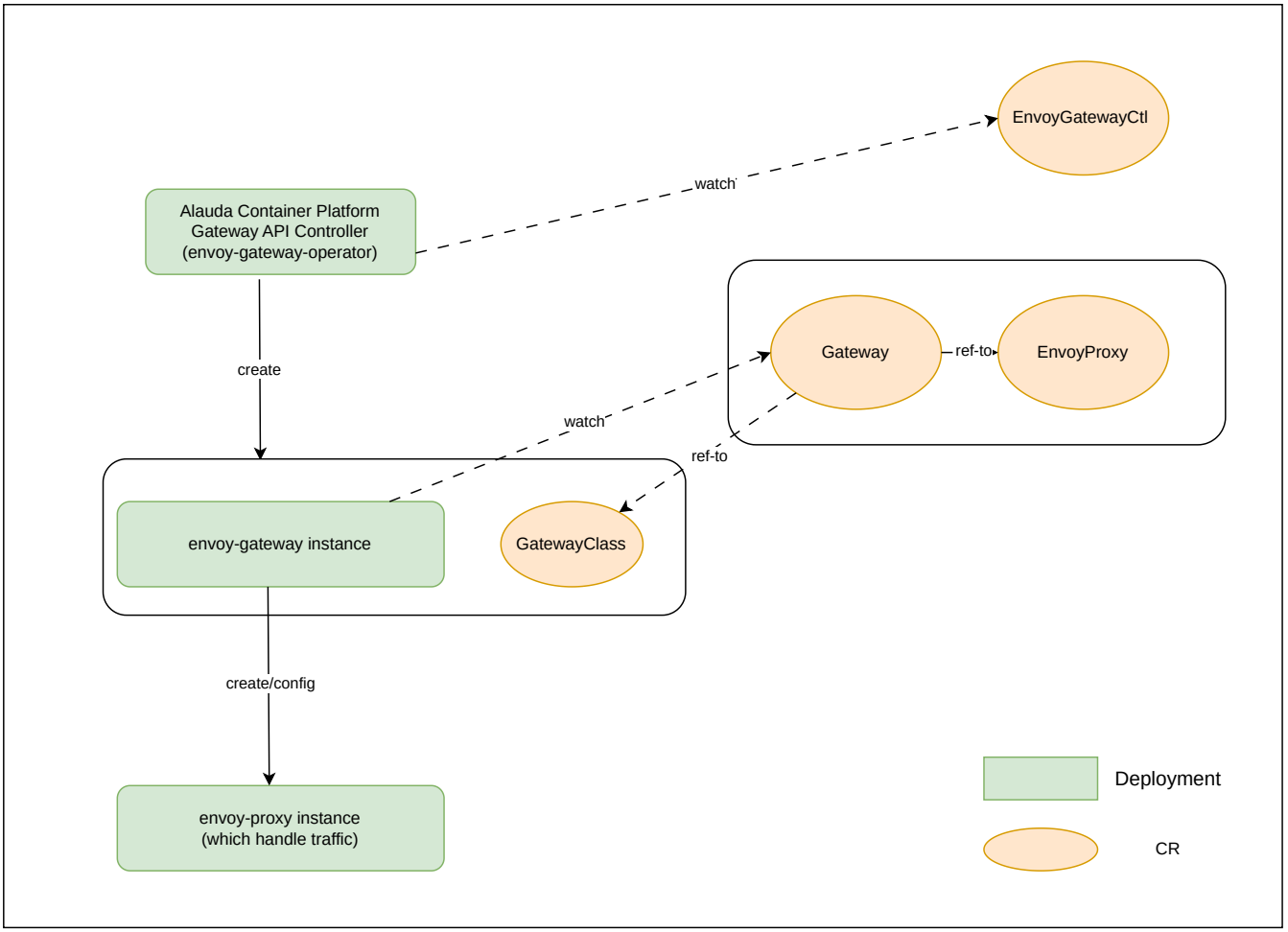
由于 Operator 由平台维护，因此也会随平台一同自动升级，从而减少手动维护 Envoy Gateway 安装的运维开销。

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

了解这些概念并创建 `EnvoyGatewayCtl` 后，请按照下一步中列出的顺序，继续配置 `Gateway`、`Route` 和 `Policy`。

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 `envoy-gateway instance` 并创建相应的 `GatewayClass`。`GatewayClass` 的名称遵循 `<namespace>-<name>` 模式。推荐的默认示例在 `envoy-gateway-operator` 命名空间中使用 `cpaas-default`，这也是 `Create EnvoyGatewayCtl` 页面上预先填充的默认配置。在此情况下，生成的 `GatewayClass` 为 `envoy-gateway-operator-cpaas-default`。每个 `EnvoyGatewayCtl` 的命名空间和名称组合都必须唯一，因此相应的 `GatewayClass` 名称也唯一。创建 `Gateway` 时，您需要选择适当的 `GatewayClass`，以确定由哪个 `envoy-gateway instance` 对其进行管理。
2. 创建引用此 `GatewayClass` 的 `Gateway` 后，`envoy-gateway instance` 将拥有该 `Gateway`。如果 `Gateway` 通过 `.spec.infrastructure.parametersRef` 引用了 `EnvoyProxy`，则会使用该 `EnvoyProxy` 控制底层 `envoy-proxy instance` 的部署方式。
3. 然后，`envoy-gateway instance` 会预配并管理相应的 `envoy-proxy instance`，后者作为数据平面，为该 `Gateway` 处理传入流量。
4. 您可以创建 `HTTPRoute`、`TCPRoute`、`UDPRoute`、`GRPCRoute` 或 `TLSRoute` 资源，并将其关联到 `Gateway`，以定义流量的匹配方式以及转发到后端 Services 的方式。您还可

以关联 `ClientTrafficPolicy` 或 `BackendTrafficPolicy` 等流量策略，以实现更高级的行为。

相关自定义资源

以下自定义资源涉及 Envoy Gateway 的部署和流量行为配置。

Operator 相关的自定义资源

自定义资源	描述
EnvoyGatewayCtl	控制特定 `envoy-gateway instance` 的部署。

对于典型用例，我们建议每个集群通过 `EnvoyGatewayCtl` 部署单个 `envoy-gateway instance`。

对于高级场景，`EnvoyGatewayCtl` 还允许您在同一集群中部署多个 `envoy-gateway instance` 资源，以加强控制和隔离，例如采用不同的部署模式↗。

`Gateway API` 相关的自定义资源

这些资源描述网关的路由语义，包括流量如何进入网关、请求如何匹配并转发到后端，以及流量处理期间应用哪些策略。

自定义资源	描述
GatewayClass ↗	告知 `envoy-gateway instance` 应管理哪些 `Gateway` 资源。
Gateway ↗	定义三个基本组件： 1. **GatewayClass** 引用 - 通过 `.spec.gatewayClassName` 引用控制此 Gateway 的 GatewayClass 2. **Listeners** 配置 - 通过 `.spec.listeners` 指定用于处理流量的端口、主机名和 TLS 证书 3. 基础设施参数 - 通过 `.spec.infrastructure.parametersRef` 引用 `EnvoyProxy`，配置副本、资源等部署细节
Route 资源 ↗	包括 `HTTPRoute`、`TCPRoute`、`UDPRoute`、`GRPCRoute` 和 `TLSRoute`。 这些资源定义四个基本组件：

自定义资源	描述
	1. 多个匹配规则集 - 用于匹配传入流量的各种条件 2. 后端目标 - 每个匹配规则集对应的目标服务 3. 流量策略 - 每条规则的配置，包括重试、超时、重定向等 4. **Gateway listeners** 关联 - 通过 `.spec.parentRefs` 定义，将路由连接到特定的 `Gateway` listeners
Policies	Policy 资源为 `Gateway` 和 `Route` 资源扩展了高级功能。Envoy Gateway 当前提供 `SecurityPolicy`、`BackendTLSPolicy`、`ClientTrafficPolicy` 和 `BackendTrafficPolicy`。

有关这些标准 Gateway API 资源的更多详情，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。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 `envoy-proxy instances` 的配置，例如副本、资源和调度约束

如需限制命名空间中 `EnvoyProxy` 资源声明的 CPU 和内存请求总量，请参阅配置 EnvoyProxy 资源配额。

在推荐的部署模式中，每个 `Gateway` 都会通过 `.spec.infrastructure.parametersRef` 引用专属的 `EnvoyProxy`。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名称和命名空间相同的配套 `EnvoyProxy` 资源。这种一对一映射让您可以通过更新相应的 `EnvoyProxy` 资源，控制 Gateway 的部署配置，例如副本、资源和调度。

通过 Envoy Gateway Operator 安装 Envoy Gateway

前提条件

继续操作前，请先查看上述概念和架构。

安装

步骤 1：安装 Envoy Gateway Operator

1. 导航至 `Administrator -> Marketplace -> OperatorHub`
2. 在 `Networking` 类别下找到 `Alauda Container Platform Gateway API Controller`，然后点击 `Install`

步骤 2：使用推荐的默认值创建 EnvoyGatewayCtl

以下默认值与本工作流中使用的推荐示例一致。

1. 导航至 `Administrator -> Marketplace -> OperatorHub`
2. 在 `Networking` 类别下找到 `Alauda Container Platform Gateway API Controller`，然后点击它打开详情页。
3. 导航至 `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 路径
Name	cpaas-default	Envoy Gateway 实例的名称	`.metadata.name`
Namespace	envoy-gateway-operator	Envoy Gateway 实例的命名空间。每个命名空间只能有一个 `EnvoyGatewayCtl`。	`.metadata.namespace`
Replica	1	Envoy Gateway 部署的副本数	`.spec.deployment.repli`

字段	默认值	描述	YAML 路径
Deploy Ns Mode	ControllerNamespace	确定 Gateway 部署安装在 Gateway 命名空间还是 controller 命名空间中。详情请参阅部署模式↗	`.spec.config.envoyGate`
Resources	1C1G	Envoy Gateway 部署的资源请求和限制	`.spec.deployment.resou`

建议：对于典型部署，请使用默认值。点击 **Create** 继续。

通过 YAML 配置

除了 Web Console 外，您还可以通过应用 YAML 创建 `EnvoyGatewayCtl`。

默认配置：

```yaml
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` 直接映射到上游 `envoy-gateway` Helm Chart 的值。当您需要修改 `EnvoyGatewayCtl` 时，请参阅上游 Chart 值文档，了解可用的配置选项。

有关可用配置选项的详细信息，请参阅 envoy-gateway-chart-values↗。

> **DANGER**
>
> 您只能通过 `EnvoyGatewayCtl` 资源配置 `envoy-gateway instance`。请勿直接修改 ConfigMaps、Deployments 等渲染后的资源，因为这些更改可能会在升级或协调过程中丢失。

示例：启用后端路由

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

```yaml
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

Envoy Gateway operator 和 `EnvoyGatewayCtl` 就绪后，请按顺序继续执行以下任务：

- 配置 GatewayAPI Gateway
- 配置 GatewayAPI Route
- 配置 GatewayAPI Policy
- Envoy Gateway 相关任务

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 无关联，也未获得其认可或赞助。所有商标均为其各自所有者的财产，在此仅用于识别目的。

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：

商标声明

Auth

基本概念

快速入门

相关 Ingress 注解

forward-auth

basic-auth

CR

ALB 特殊 Ingress 注解

Ingress-Nginx 身份验证相关的其他功能

注意：与 Ingress-Nginx 不兼容的部分

为 ALB 部署

方法 1：使用 L

方法 2：使用外

商标声明

ALB 性能选

小型生产环境

境

境

署建

使用

ALB 中的负载均衡会话亲和策略

概述

可用算法

配置方式

最佳实践

HTTP 重定向

基本概念

CORS

基本概念

CRD

Header 修改

基本概念

示例

商标声明

ModSecurity

术语

操作步骤

相关说明

OTel

术语

前提条件

操作步骤

相关操作

附加说明

配置示例

商标声明

TCP/HTTP

ALB 与 Ingress-NGINX 注解兼容性

基本概念

支持的 ingress-nginx 注解

商标声明

ALB 监控

术语

操作步骤

监控指标

了解 ALB

> **WARNING**
>
> ALB 已弃用。请改用 ingress-nginx-operator 或 envoy-gateway。

ALB（Another Load Balancer）是由 OpenResty 驱动的 Kubernetes Gateway，融合了 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

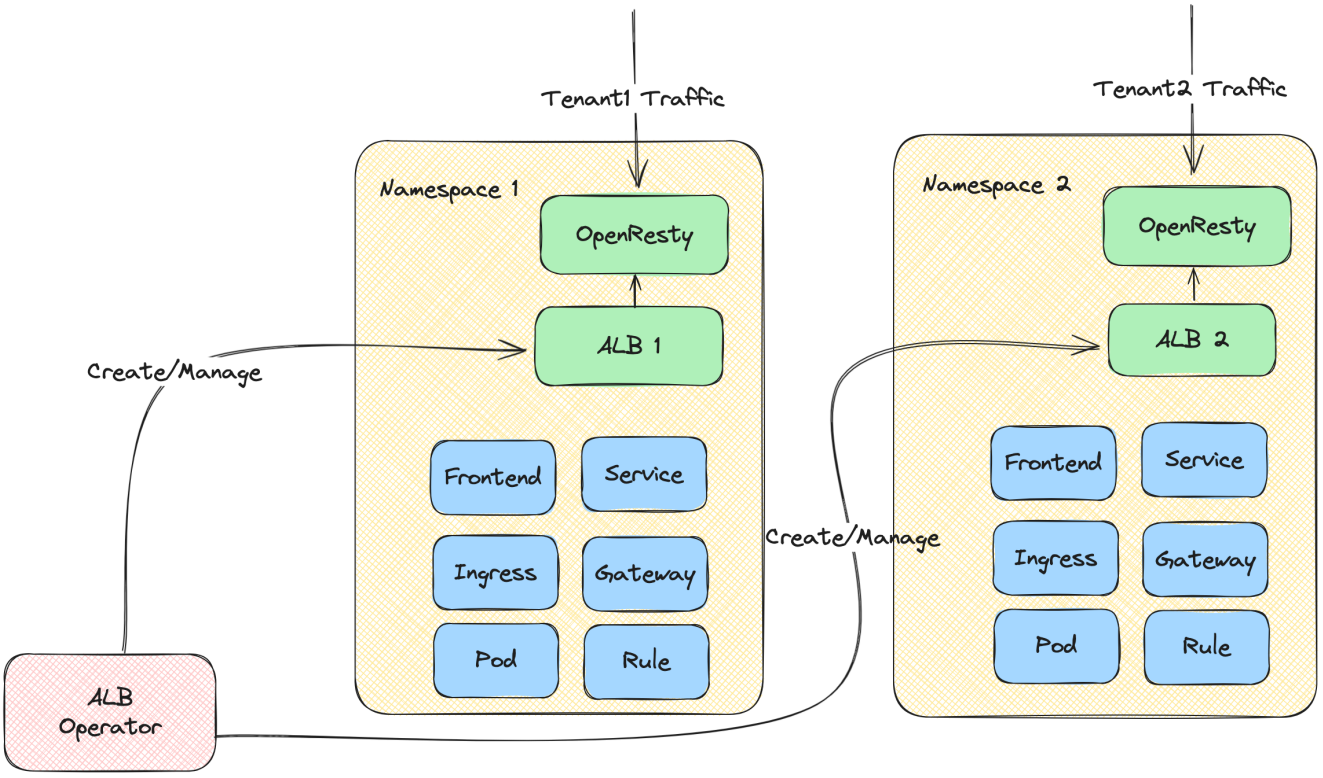
ALB Leader

项目

其他资源：

商标声明

核心组件



- **ALB Operator**：用于管理 ALB 实例生命周期的 operator。它会监视 ALB CR，并为不同租户创建或更新实例。
- **ALB Instance**：ALB 实例包括充当数据平面的 OpenResty 和充当控制平面的 Go controller。Go controller 会监视各种 CR（Ingress、Gateway、Rule 等），并将其转换为 ALB 专用的 DSL 规则。随后，OpenResty 使用这些 DSL 规则匹配和处理传入请求。

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，您可以通过 `curl http://${ip}` 访问该应用。

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

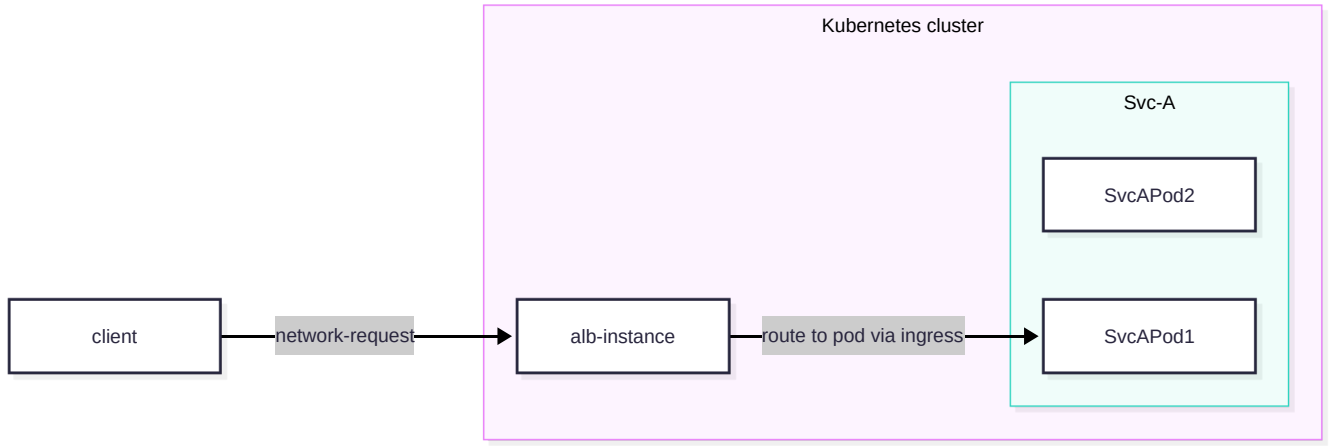
LoadBalancer 是现代云原生架构中的关键组件，可充当智能流量路由器和负载均衡器。

要了解 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相互关系：

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

这些组件协同工作，提供灵活而强大的流量管理能力。

下面将介绍这些概念如何在请求路径中协同工作。每个概念的详细介绍请参阅单独的文章。



在请求调用链中：

1. 客户端发送 HTTP/HTTPS/其他协议请求，最终请求将到达 **ALB** 的某个 **pod**，该 pod（一个 ALB 实例）将开始处理此请求。
2. 此 ALB 实例查找能够匹配此请求的规则。
3. 如有需要，根据规则修改、重定向或重写请求。
4. 从规则配置的服务中查找并选择一个 pod IP，然后将请求转发到该 pod。

Ingress

Ingress 是 Kubernetes 中的一种资源，用于描述应将请求发送到哪个服务。

Ingress Controller

一种理解 Ingress 资源并将请求代理到服务的程序。

ALB

ALB 是一个 Ingress controller。

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

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

ALB Instance

ALB 也是在集群中运行的 Deployment（一组 pod）。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 ALB 的 Frontend（FT）、Rule 和其他配置。

ALB-Operator

ALB-Operator 是部署在集群中的默认组件，是用于 ALB 的 operator。它会根据 ALB 资源为每个 ALB 创建、更新或删除 Deployment 及其他相关资源。

Frontend（缩写：FT）

FT 是由 ALB 自身定义的资源，用于表示 ALB 实例监听的端口。

FT 可以由 ALB-Leader 创建，也可以由用户手动创建。

ALB-Leader 创建 FT 的情况：

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

RULE

RULE 是由 ALB 自身定义的资源。
它与 Ingress 具有相同的作用，但更加具体。
一个 RULE 唯一关联一个 FT。

RULE 可以由 ALB-Leader 创建，也可以由用户手动创建。

ALB-Leader 创建 RULE 的情况：

1. 将 Ingress 同步到 RULE。

ALB Leader

在多个 ALB 实例中，其中一个将被选举为 leader。
leader 负责：

1. 将 Ingress 转换为 RULE。 我们将为 Ingress 中的每个路径创建一个 RULE。
2. 创建 Ingress 所需的 FT。 例如，如果 Ingress 有证书，我们将创建 FT 443（HTTPS）；如果 Ingress 没有证书，我们将创建 FT 80（HTTP）。

项目

从 ALB 的角度来看，项目是一组命名空间。

您可以在一个 ALB 中配置一个或多个项目。
当 ALB Leader 将 Ingress 转换为 RULE 时，它会忽略不属于该项目的命名空间中的 Ingress。

其他资源：

- 配置负载均衡器

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是一家独立的供应商。本产品与 The Linux Foundation 或 F5, Inc. 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处使用这些商标仅用于标识目的。

Auth

目录

基本概念
什么是 Auth
支持的 Auth 方法
Auth 配置方法
Auth 结果处理
快速入门
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
构造与 app-request 相关的注解
auth-response-headers
Cookie 处理
重定向登录相关配置
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 身份验证相关的其他功能
Global-Auth
No-Auth-Locations
注意：与 Ingress-Nginx 不兼容的部分
故障排除
商标声明

基本概念

什么是 **Auth**

Auth 是一种在请求到达实际服务之前执行身份验证的机制。它允许你在 ALB 层统一处理身份验证，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）
   - 将请求发送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，例如 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到身份验证服务
  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：简单的访问控制、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. 与已配置的身份验证信息进行比较
  4. 如果验证通过，则转发到后端服务

Auth 配置方法

1. 全局 **Auth**

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

2. 路径级 **Auth**

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

3. 禁用 **Auth**

- 为特定路径禁用 Auth
- 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）

快速入门

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-aut
h -o jsonpath='{.items[*].status.podIP}');echo $ALB_IP
```

配置 Secret 和 Ingress

```
# echo "Zm9vOiRhcHIxJHFJQ05aNjFRJDJpb29pSlZVQU1tcHJxMjU4L0NoUDE=" | base6
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 "Zm9vOiJhYXIi" | base64 -d # foo:bar
curl -v -X GET -H "Authorization: Basic Zm9vOmJhcg==" http://$ALB_IP:80/a
pp-file # should return 200
# wrong password
curl -v -X GET -H "Authorization: Basic XXXXOmJhcg==" http://$ALB_IP:80/a
pp-file # should return 401
```

相关 **Ingress** 注解

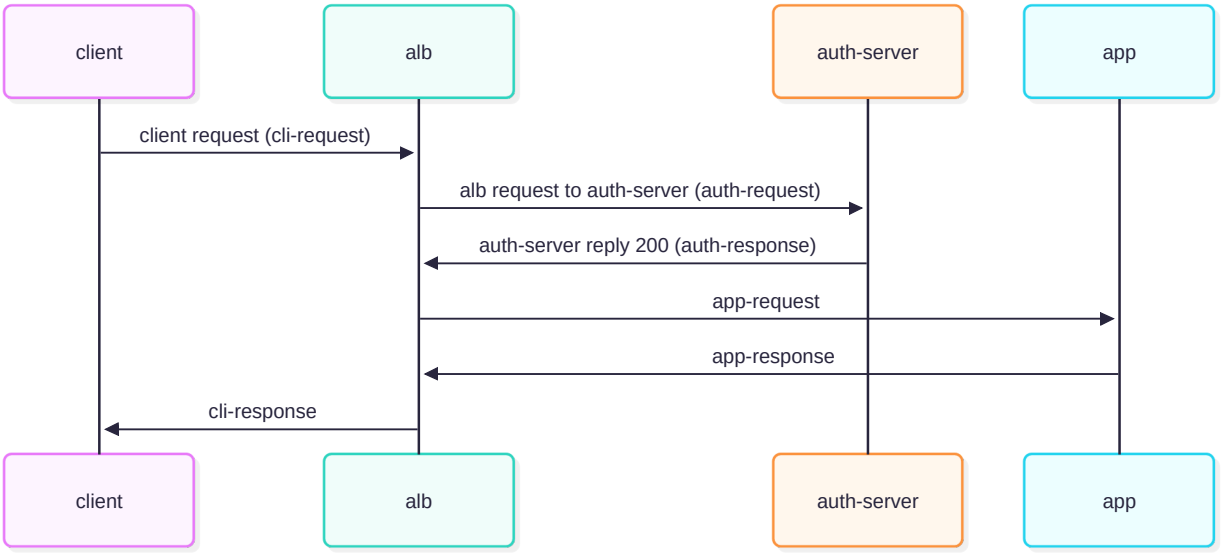
Ingress-nginx 定义了一系列用于配置身份验证流程详细信息的注解。下面列出了 ALB 支持的注解，其中“v”表示支持，“x”表示不支持。

	支持	类型	说明
forward-auth			通过发送 http 请求进行转发身份验证
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	

	支持	类型	说明
nginx.ingress.kubernetes.io/auth-always-set-cookie	v	boolean	
nginx.ingress.kubernetes.io/auth-snippet	x	string	
basic-auth			使用用户名和密码 Secret 进行身份验证
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" 或 "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保持连接行为
nginx.ingress.kubernetes.io/auth-keepalive	x	number	

	支持	类型	说明
nginx.ingress.kubernetes.io/auth-keepalive-share-vars	x	"true" 或 "false"	
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" 或 "false"	
nginx.ingress.kubernetes.io/auth-tls-match-cn	x	string	

forward-auth



相关注解：

- nginx.ingress.kubernetes.io/auth-url
- nginx.ingress.kubernetes.io/auth-method
- nginx.ingress.kubernetes.io/auth-signin
- nginx.ingress.kubernetes.io/auth-signin-redirect-param
- nginx.ingress.kubernetes.io/auth-response-headers
- nginx.ingress.kubernetes.io/auth-proxy-set-headers
- nginx.ingress.kubernetes.io/auth-request-redirect
- nginx.ingress.kubernetes.io/auth-always-set-cookie

这些注解描述了上图中对 auth-request、app-request 和 cli-response 所做的修改。

构造相关注解

auth-url

Auth-request 的 URL，值可以是变量。

auth-method

Auth-request 的方法。

auth-proxy-set-headers

值为 ConfigMap 引用，格式为 `ns/name`。 默认情况下，cli-request 中的所有标头都会发送到 auth-server。可以通过 proxy_set_header 配置其他标头。默认发送以下标头：

```
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;
```

构造与 **app-request** 相关的注解

auth-response-headers

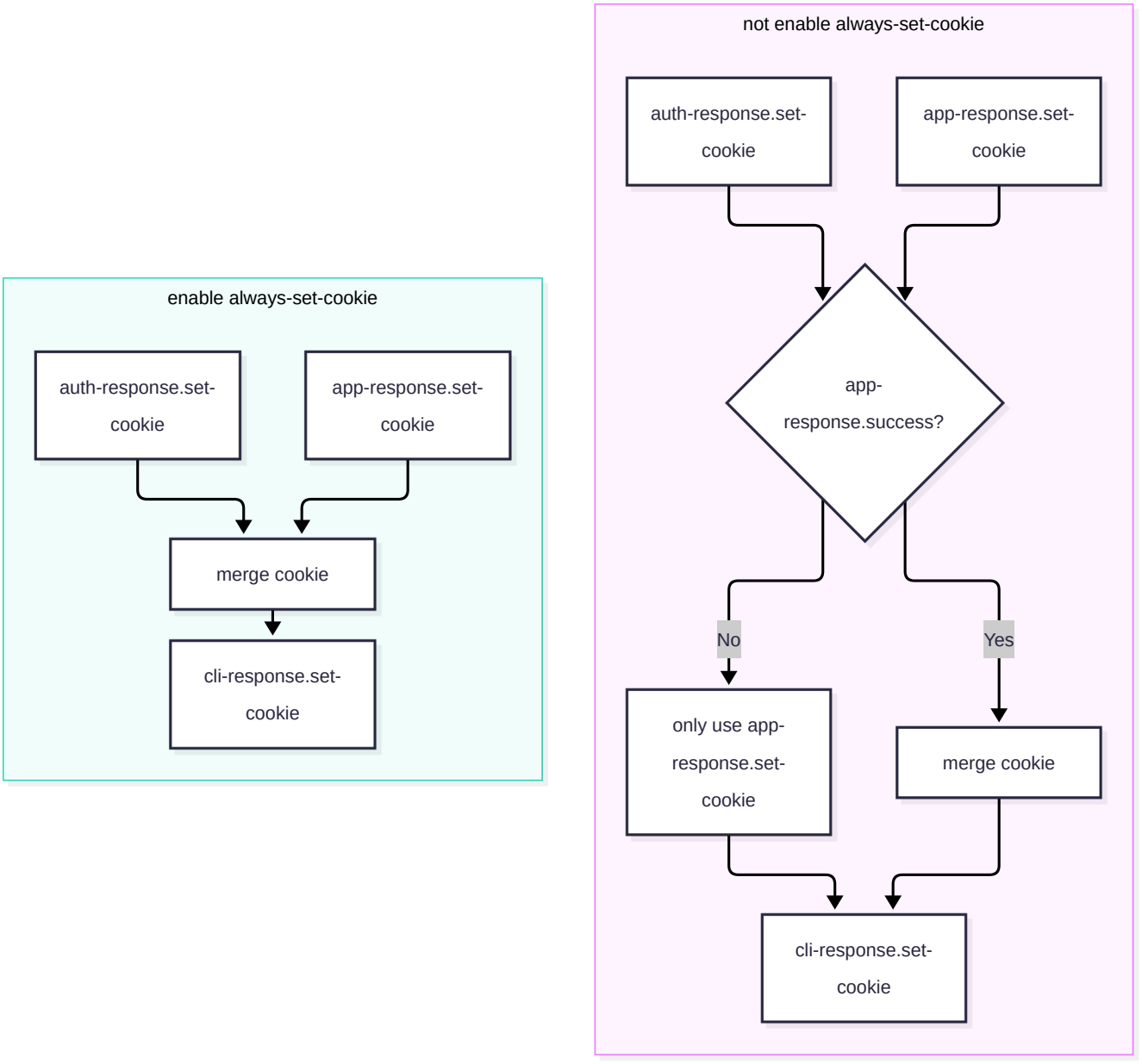
值为逗号分隔的字符串，用于将 auth-response 中的特定标头传递到 app-request。 例如：

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

ALB 发起 app-request 时，会包含 auth-response 标头中的 Remote-User 和 Remote-Name。

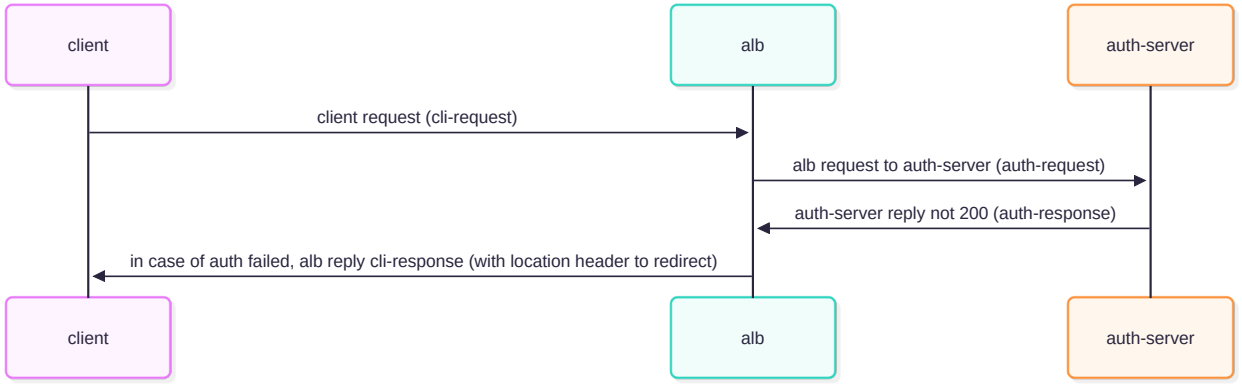
Cookie 处理

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



重定向登录相关配置

当 auth-server 返回 401 时，可以在 cli-response 中设置重定向标头，指示浏览器重定向到 auth-signin 指定的 URL 进行验证。



auth-signin

值为 URL，用于指定 cli-response 中的 location 标头。

auth-signin-redirect-param

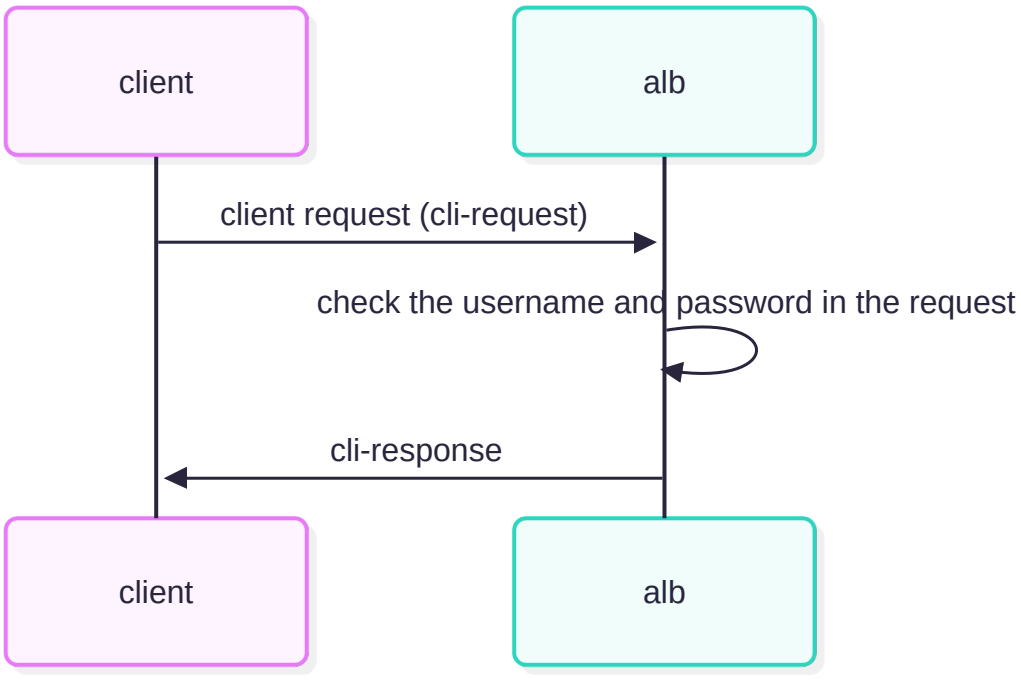
signin-url 中查询参数的名称，默认为 rd。 如果 signin-url 不包含指定的 `auth-signin-redirect-param` 参数名，alb 会自动添加该参数。参数值将设置为 `$pass_access_scheme://$http_host$escaped_request_uri`，用于记录原始请求 URL。

auth-request-redirect

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

basic-auth

basic-auth 是 RFC 7617 ↗ 中描述的身份验证过程。 交互过程如下：



auth-realm

受保护区域的描述↗ 即 cli-response 的 `WWW-Authenticate` 标头中的 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 中每个键表示一个用户名，对应的值为密码哈希（htpasswd 格式中不包含用户名）。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 新增了与身份验证相关的配置项，可在 ALB/Frontend/Rule CR 中进行配置。 运行时，ALB 会将 Ingress 上的注解转换为规则。

```yaml
auth:
  # Basic authentication configuration
  basic:
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-type: basic
    auth_type: 'basic'
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-realm
    realm: 'Restricted Access'
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-secret
    secret: 'ns/name'
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-secret-type
    secret_type: 'auth-map|auth-file'
  # Forward authentication configuration
  forward:
    #  boolean; corresponding to nginx.ingress.kubernetes.io/auth-always-set-cookie
    always_set_cookie: true
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-proxy-set-headers
    auth_headers_cm_ref: 'ns/name'
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-request-redirect
    auth_request_redirect: '/login'
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-method
    method: 'GET'
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-signin
    signin: '/signin'
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-signin-redirect-param
    signin_redirect_param: 'redirect_to'
    #  []string; corresponding to nginx.ingress.kubernetes.io/auth-response-headers
    upstream_headers:
      - 'X-User-ID'
      - 'X-User-Name'
      - 'X-User-Email'
    #  string; corresponding to nginx.ingress.kubernetes.io/auth-url
    url: 'http://auth-service/validate'
```

Auth 支持在以下位置进行配置：

- 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 的过程中，ALB 根据注解的前缀确定优先级。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`

这样可以处理与 ingress-nginx 的兼容性问题，并在指定的 Ingress 路径上配置身份验证。

Auth-Enable

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

ALB 新增的注解，用于指定是否为 Ingress 启用身份验证功能。

Ingress-Nginx 身份验证相关的其他功能

Global-Auth

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

No-Auth-Locations

在 ALB 中，可以在 Ingress 上配置注解 `alb.ingress.cpaas.io/auth-enable: "false"`，以禁用此 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

故障排除

1. 检查 ALB Pod 的 Nginx 容器日志
2. 检查返回内容中的 `X-ALB-ERR-REASON` 响应头

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是一家独立供应商。本产品与 F5, Inc. 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

为 ALB 部署高可用 VIP

ALB 的高可用性需要 VIP。获取 VIP 有两种方式。

目录

方法 1：使用 LoadBalancer 类型的 Service 提供 VIP

方法 2：使用外部 ALB 设备提供 VIP

商标声明

方法 1：使用 LoadBalancer 类型的 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。

使用此功能前，请确保集群支持 LoadBalancer Service。您可以使用平台内置的实现。设置方法请参阅配置 MetalLB。

MetalLB 就绪后，您可以向 `alb.spec.config.vip.lbSvcAnnotations` 添加以下注解，以调整 MetalLB 的行为。请参阅 ALB 网络配置。

注解	描述
`metallb.universe.tf/loadBalancerIPs`	以逗号分隔的 VIP 列表，将分配给 Service。多 VIP 或双栈场景请使用多个条目，例如：`192.0.2.10,2001:db8::10`。
`metallb.universe.tf/address-pool`	要从中分配地址的 MetalLB 地址池。

方法 2：使用外部 ALB 设备提供 VIP

- 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。如果要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。建议使用商用负载均衡器设备提供 VIP；如果没有，可使用纯软件数据中心 LB 解决方案（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，以缩短 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。 • 对于工作负载集群，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，建议设置为 2 秒。
检查间隔	健康检查的时间间隔，建议设置为 5 秒。
不健康阈值	连续失败达到指定次数后，将后端服务器的健康检查状态判定为失败，建议设置为 3 次。

商标声明

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

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

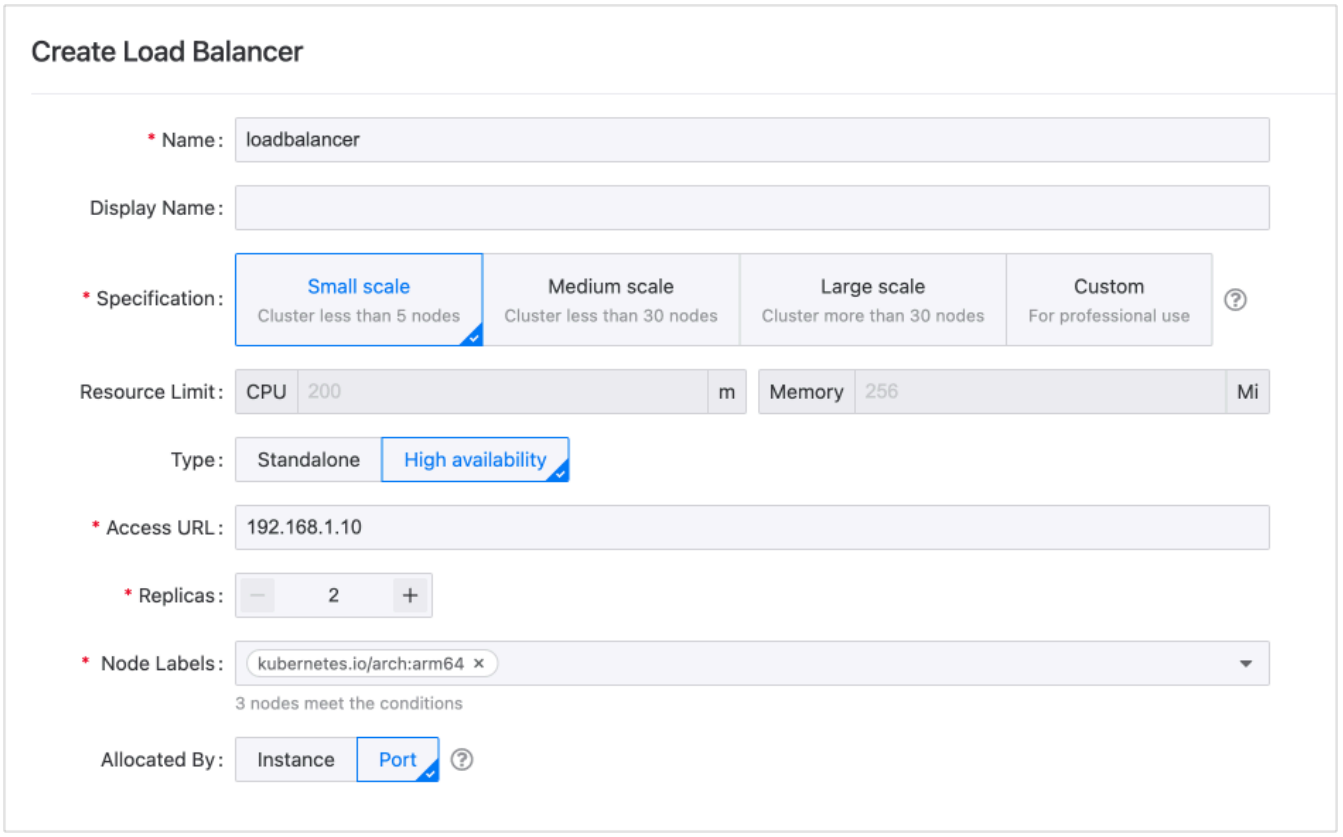
小型生产环境
中型生产环境
大型生产环境
特殊场景部署建议
负载均衡器使用模式选择

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。

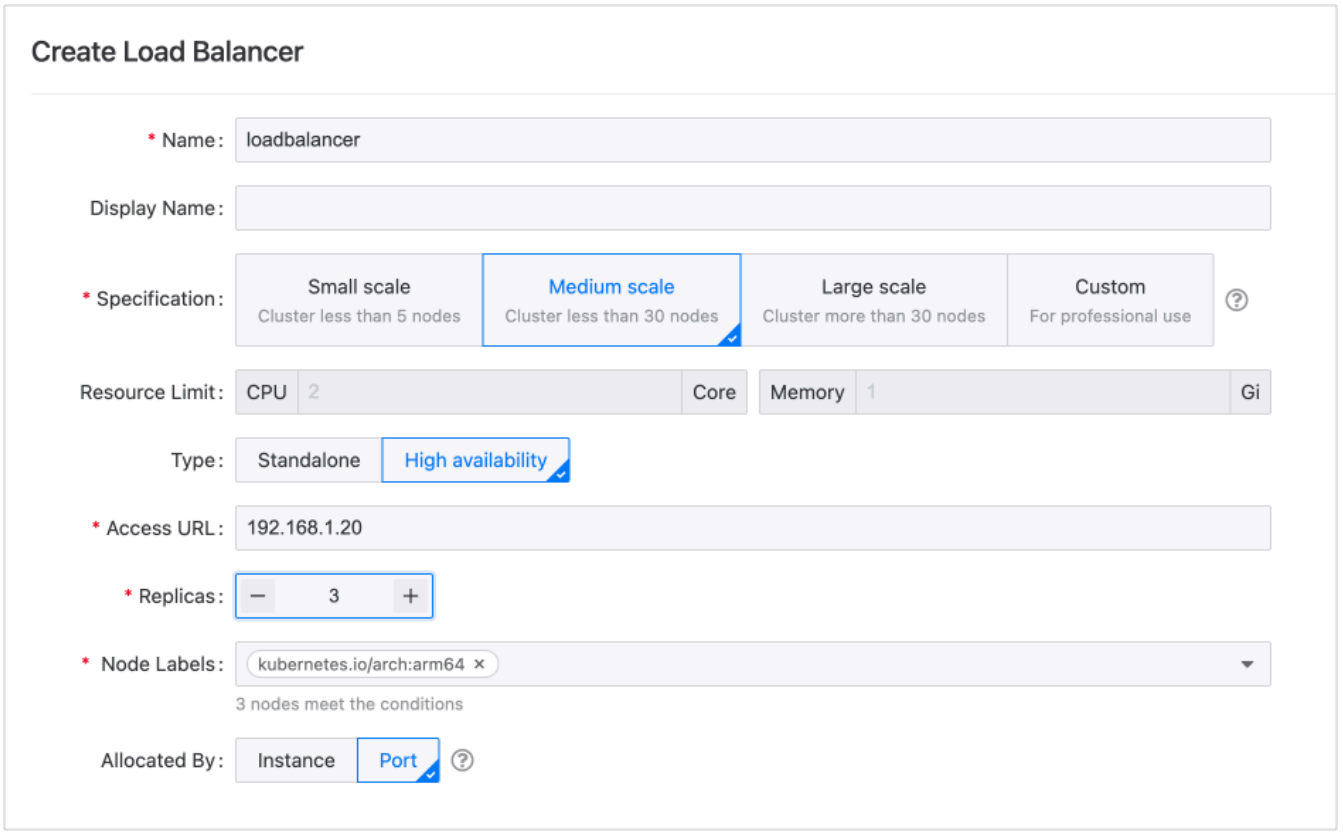


中型生产环境

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。

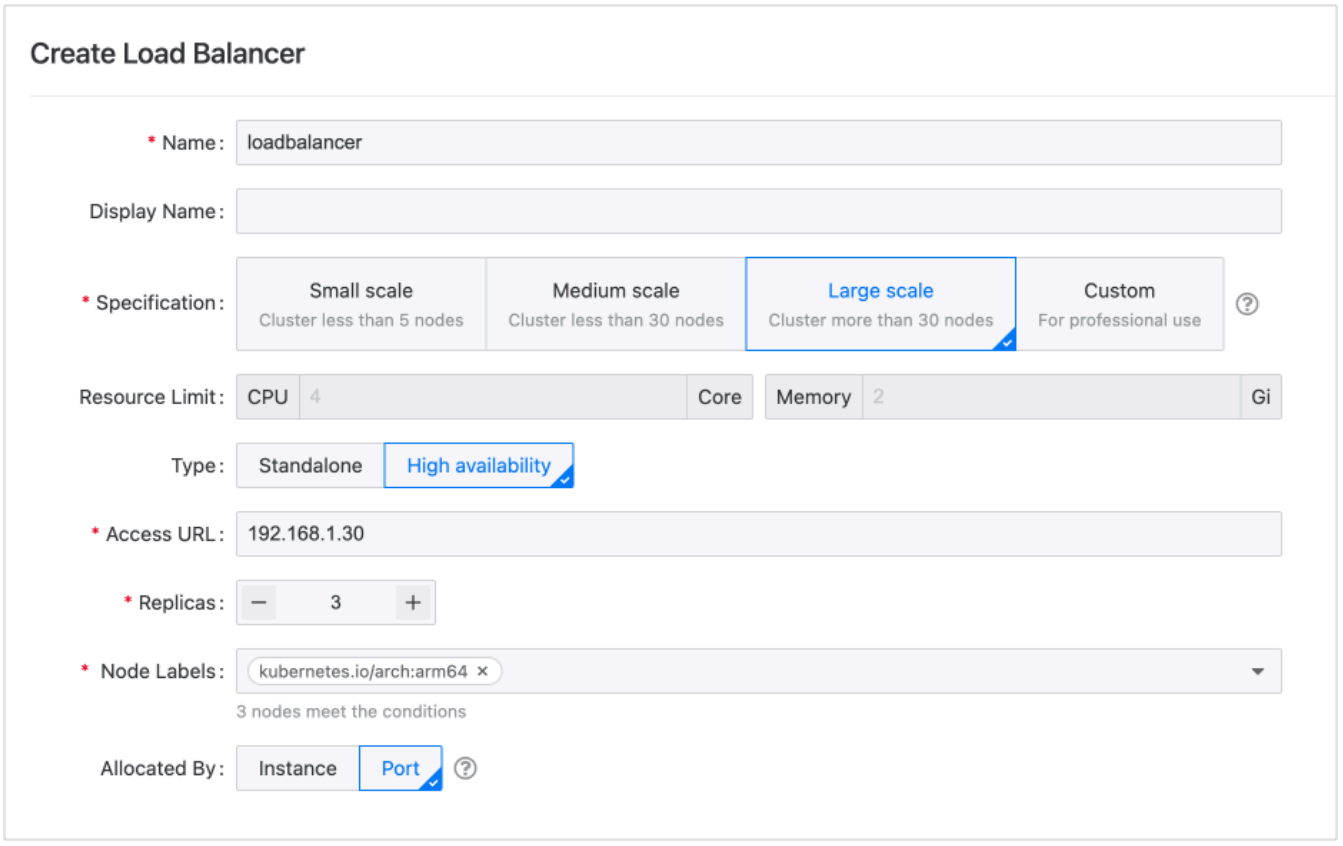


大型生产环境

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。



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 的亲和

基于 Header 的亲和

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、性能优化或负载均匀分布。

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 | header | 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

L4/L7 超时

目录

基本概念
CRD
超时的含义
Ingress 注解
端口级别超时
商标声明

基本概念

L4/L7 超时是 ALB 提供的一项功能，用于配置 L4/L7 代理的超时时间。

超时通过 Lua 脚本实现，修改超时配置时，Nginx 无需重新加载。

CRD

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

配置可以在以下位置进行：

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

超时的含义

超时有三种类型：

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。该超时时间设置在两次连续读取操作之间，而不是针对整个响应。如果在此时间内未收到任何数据，连接将关闭。
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，该超时时间设置在两次连续写入操作之间。如果在此时间内无法发送任何数据，连接将关闭。

Ingress 注解

注解	描述
nginx.ingress.kubernetes.io/proxy-connect-timeout	对应 CR 中的 proxy_connect_timeout_ms
nginx.ingress.kubernetes.io/proxy-read-timeout	对应 CR 中的 proxy_read_timeout_ms

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

端口级别超时

您可以直接在端口上配置超时，该配置用作 L4 超时。

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是一家独立的供应商。本产品与 F5, Inc. 没有关联，也未得到其认可或赞助。所有商标均归其各自所有者所有，此处仅用于识别目的。

HTTP 重定向

目录

基本概念
CRD
Ingress 注解
SSL 重定向
端口级别重定向
规则级别重定向
商标声明

基本概念

HTTP 重定向是 ALB 提供的一项功能。对于匹配规则的请求，ALB 会直接返回 30x HTTP 状态码，并使用 Location 标头指示客户端重定向到新的 URL。

ALB 支持在端口级别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
```

可在以下位置配置重定向：

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

Ingress 注解

注解	描述
nginx.ingress.kubernetes.io/permanent-redirect	对应 CR 中的 URL，默认将代码设置为 301
nginx.ingress.kubernetes.io/permanent-redirect-code	对应 CR 中的代码
nginx.ingress.kubernetes.io/temporal-redirect	对应 CR 中的 URL，默认将代码设置为 302
nginx.ingress.kubernetes.io/temporal-redirect-code	对应 CR 中的代码

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

SSL 重定向

1. SSL-redirect 和 force-ssl-redirect 的区别在于：只有当 Ingress 为相应域名配置了证书时，SSL-redirect 才会生效；而 force-ssl-redirect 无论是否配置证书都会生效。
2. 对于 HTTPS 端口，如果只配置了 SSL-redirect，则不会设置重定向。

端口级别重定向

在端口级别配置重定向时，发送到该端口的*所有请求*都会根据重定向配置进行重定向。

规则级别重定向

在规则级别配置重定向时，匹配该规则的请求会根据重定向配置进行重定向。

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是一家独立供应商。本产品与 F5, Inc. 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，在此仅用于标识目的。

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 修改

目录

基本概念

使用 annotations

示例

商标声明

基本概念

收到请求时，Header 修改功能允许在将请求转发到后端之前调整请求 Header。 同样，收到响应时，该功能允许在将响应返回给客户端之前调整响应 Header。

使用 annotations

目标	annotation key
ingress	`alb.ingress.cpaas.io/rewrite-request`, `alb.ingress.cpaas.io/rewrite-response`
rule	`alb.rule.cpaas.io/rewrite-request`, `alb.rule.cpaas.io/rewrite-response`

annotation 值是包含配置的 JSON 字符串。

```
type RewriteRequestConfig struct {
    Headers       map[string]string   `json:"headers,omitempty"`
// set header
    HeadersVar    map[string]string   `json:"headers_var,omitempty"`
// set header, which values are variable name
    HeadersRemove []string            `json:"headers_remove,omitempty"`
// remove header
    HeadersAdd    map[string][]string `json:"headers_add,omitempty"`
// add header, which values could be multiple
    HeadersAddVar map[string][]string `json:"headers_add_var,omitempty"`
// add header, which values could be multiple and which values are variab
le name
}
type RewriteResponseConfig struct {
    Headers       map[string]string   `json:"headers,omitempty"`
// set header
    HeadersRemove []string            `json:"headers_remove,omitempty"`
// remove header
    HeadersAdd    map[string][]string `json:"headers_add,omitempty"`
// add header, which values could be multiple
}
```

注意：在 `*_var` 映射中，键是 Header 名称，值是 ALB context 变量名称。 例如，向 Ingress 添加以下 annotation：

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

将把键 `x-my-host` 和请求 Host Header 的值添加到请求 Header 中。 变量名称请参阅 nginx 变量↗。

ALB 还提供其他变量：

变量名称	描述
`first_forward_or_remote_addr`	第一个转发地址或远程地址，默认值为 `remote_addr`
`first_forward`	第一个转发地址，默认值为空字符串

示例

要从 Cookie 添加 Authorization Header，可以使用：

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

要设置 HSTS，可以使用：

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

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是独立供应商。本产品与 F5, Inc. 无关联，也未获得其认可或赞助。所有商标均为其各自所有者的财产，在此仅用于标识目的。

URL 重写

目录

基本概念

配置

商标声明

基本概念

ALB 可以在将请求转发到后端之前重写请求 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

```yaml
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
```

通过规则

```yaml
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 `/abc`。

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是独立供应商。本产品与 F5, Inc. 没有关联，也未得到其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

ModSecurity

ModSecurity 是一种开源 Web 应用程序防火墙（WAF），旨在保护 Web 应用程序免受恶意攻击。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，可在 Ingress 级别单独进行配置。

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

配置示例

商标声明

术语

术语	说明
owasp-core-rules	OWASP 核心规则集是一套开源规则集，用于检测和防止常见的 Web 应用程序攻击。

操作步骤

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

方法一：添加注解

将以下注解添加到相应 YAML 文件的 metadata.annotations 字段中，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 ModSecur
nginx.ingress.kubernetes.io/enable-owasp-core-rules	bool	Ingress	启用 OWA 核心规则集
nginx.ingress.kubernetes.io/modsecurity-transaction-id	string	Ingress	用于标识每 请求的唯一 务 ID，便于 录日志和调 试。
nginx.ingress.kubernetes.io/modsecurity-snippet	string	Ingress, ALB, FT, Rule	允许用户指 自定义 ModSecur 置，以满足 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；设 `true` 以应用预定义 规则集。
alb.modsecurity.cpaas.io/cmref	string	Ingress	引用特定配置，例如， 过指定 ConfigMap 的 径 （`$ns/$name#$sect` 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, # Enable or disable ModSecurity
      'transactionId': '$xx', # Use ID from Nginx
      'useCoreRules': true, # Add modsecurity_rules_file /etc/nginx/owasp-modsecurity-crs/nginx-modsecurity.conf
      'useRecommend': true, # Add modsecurity_rules_file /etc/nginx/modsecurity/modsecurity.conf
      'cmRef': '$ns/$name#$section', # Add configuration from ConfigMap
    } }
```

3. 保存并应用配置文件。

相关说明

覆盖

如果未在 Rule 中配置 ModSecurity，它会尝试在 FT 中查找配置；如果 FT 中也没有配置，则会使用 ALB 中的配置。

配置示例

以下示例部署一个名为 `waf-alb` 的 ALB，以及一个名为 `hello` 的演示后端原生应用。此外，还会部署一个名为 `ing-waf-enable` 的 Ingress，其中定义了 `/waf-enable` 路由并配置 ModSecurity 规则。任何包含查询参数 `test` 且其值中包含字符串 `test` 的请求都将被阻止。

```
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
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/enable-modsecurity: "true"
    nginx.ingress.kubernetes.io/modsecurity-transaction-id: "$request_id"
    nginx.ingress.kubernetes.io/modsecurity-snippet: |
      SecRuleEngine On
      SecRule ARGS:test "@contains test" "id:1234,deny,log"
  name: ing-waf-enable
spec:
  ingressClassName: waf-alb
  rules:
    - http:
        paths:
          - backend:
              service:
                name: hello
                port:
                  number: 80
            path: /waf-enable
            pathType: ImplementationSpecific
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ing-waf-normal
spec:
  ingressClassName: waf-alb
```

```
  rules:
    - http:
        paths:
          - backend:
              service:
                name: hello
                port:
                  number: 80
            path: /waf-not-enable
            pathType: ImplementationSpecific
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
spec:
  replicas: 1
  selector:
    matchLabels:
      service.cpaas.io/name: hello
      service_name: hello
  template:
    metadata:
      labels:
        service.cpaas.io/name: hello
        service_name: hello
    spec:
      containers:
        - name: hello-world
          image: hashicorp/http-echo
          imagePullPolicy: IfNotPresent
---
apiVersion: v1
kind: Service
metadata:
  name: hello
spec:
  internalTrafficPolicy: Cluster
  ipFamilies:
    - IPv4
  ipFamilyPolicy: SingleStack
  ports:
    - name: http
      port: 80
```

```
      protocol: TCP
      targetPort: 5678
  selector:
    service_name: hello
  sessionAffinity: None
  type: ClusterIP
EOF
```

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是独立供应商。本产品与 F5, Inc. 无关联，未获得其认可或赞助。所有商标均为其各自所有者的财产，在此仅用于标识目的。

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）中的遥测数据收集、处理和导出提供与供应商无关的标准。它可以帮助开发人员更轻松地分析软件的性能和行为，从而促进应用问题的诊断和解决。

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
- 附加说明
  - 采样策略
  - 属性
- 配置示例
- 商标声明

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，用于跟踪分布式系统中的请求流；每个 Trace 由多个 Span 组成。
Span	Trace 中的独立操作或事件，包括开始时间、持续时间和其他相关信息。
OTel Server	能够接收和存储 Trace 数据的 OTel 服务器，例如 Jaeger、Prometheus 等。
Jaeger	一个开源分布式追踪系统，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 的键值对，用于提供额外的上下文信息。这包括 Resource Attributes 和 Span Attributes；详情请参阅 Attributes。
Sampler	用于确定是否对 Trace 进行采样和报告的策略组件。可以配置不同的采样策略，例如全量采样、按比例采样等。
ALB (Another Load Balancer)	一种在集群中的可用节点之间分配网络请求的软件或硬件设备；平台使用的负载均衡器（ALB）是一个第 7 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；同时还支持配置是否在 Ingress 层级提交 Trace。
FT (Frontend)	ALB 的端口配置，用于指定端口级别的配置。
Rule	端口（FT）上的路由规则，用于匹配特定路由。
HotROD (Rides on Demand)	Jaeger 提供的示例应用，用于演示分布式追踪的使用；详情请参阅 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；详情请参阅 hotrod-with-proxy↗。

前提条件

- 确保存在可正常运行的 **ALB**：创建或使用现有的 ALB。以下示例使用 `<otel-alb>` 作为 ALB 名称。有关创建 ALB 的操作步骤，请参阅 配置 ALB。

- 确保存在 **OTel** 数据报告服务器地址：下文将该地址称为 `<jaeger-server>`。

操作步骤

更新 **ALB** 配置

1. 在集群的控制平面节点上，使用 CLI 工具执行以下命令编辑 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。

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

   完成后的配置示例：

```
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，并将所有请求的 Trace 信息报告到 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，您可以更好地监控和调试应用的请求流，并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 在处理通过该 Ingress 的请求时启用 OpenTelemetry 功能，即收集并报告请求的 Trace 信息。设置为 `false` 或移除此 annotation 时，表示不收集或报告请求的 Trace 信息。

- 在 **Ingress** 上启用或禁用 **OTel** 信任
  OTel 信任用于确定 Ingress 是否信任并使用传入请求中的 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 会继续使用已有的 Trace 信息，有助于保持跨服务追踪的一致性，使整个请求链能够在分布式追踪系统中得到完整追踪和分析。设置为 `false` 时，将为请求生成新的追踪信息，这可能导致请求进入 Ingress 后被视为新追踪链的一部分，从而中断跨服务 Trace 的连续性。

- 在 **Ingress** 上添加不同的 **OTel** 配置
  此配置允许您为不同的 Ingress 资源自定义 OTel 的行为和数据导出方式，从而对每个服务的追踪策略或目标进行精细控制。
  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**：指定如何将收集的 Trace 数据发送到 OTel Collector（OTel 数据报告服务器）。
- **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**

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

在集群的控制平面节点上，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.versi
ons[2].schema.openAPIV3Schema.properties.spec.properties.config.propertie
s.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 数据报告服务器的地址，支持 http/https 协议和域名。
flags.hide_upstream_attrs	是否报告上游规则的信息。
flag.notrust_incoming_span	是否信任并使用传入请求中的 OTel Trace 信息（例如 trace ID）。
flags.report_http_request_header	是否报告请求标头。
flags.report_http_response_header	是否报告响应标头。

参数	说明
sampler.name	采样策略名称；有关详细信息，请参阅 采样策略。
sampler.options.fraction	采样率。
sampler.options.parent_name	parent_base 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 未进行配置，则 FT 将从 ALB 继承这些参数作为自身配置；也就是说，FT 继承 ALB 配置，而 Rule 可以同时从 ALB 和 FT 继承配置。

- **ALB**：ALB 上的配置通常是全局配置和默认配置。您可以在此配置 Collector 地址等全局参数，这些参数将由下层的 FT 和 Rule 继承。
- **FT**：FT 可以从 ALB 继承配置，即 FT 上未配置的某些 OTel 参数将使用 ALB 中的配置。但是，您也可以进一步细化 FT 的配置；例如，可以选择在 FT 上启用或禁用 OTel，而不影响其他 FT 或 ALB 的全局设置。
- **Rule**：Rule 可以同时从 ALB 和 FT 继承配置。但是，您也可以进一步细化 Rule 的配置；例如，特定 Rule 可以选择不信任传入的 OTel Trace 信息，或调整采样策略。

操作步骤

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

附加说明

采样策略

参数	说明
always on	始终报告所有追踪数据。

参数	说明
always off	从不报告追踪数据。
traceid-ratio	根据 `traceid` 决定是否报告。`traceparent` 的格式为 `xx-traceid-xx-flag`，其中 `traceid` 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 `fraction` 乘以 4294967295（即 (2^32-1)），则会报告。
parent-base	根据请求中 traceparent 的标志部分决定是否报告。当标志为 01 时，将会报告；例如：`curl -v "http://$ALB_IP/" -H 'traceparent: 00-xx-xx-01'`；当标志为 02 时，不会报告；例如：`curl -v "http://$ALB_IP/" -H 'traceparent: 00-xx-xx-02'`。

属性

- 资源属性
  默认报告这些属性。

参数	描述
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** 属性
  - 默认报告的属性：

参数	描述
http.status_code	状态码

参数	描述
http.request.resend_count	重试次数
alb.rule.rule_name	此请求匹配的 Rule 名称
alb.rule.source_type	此请求匹配的 Rule 类型，目前仅支持 Ingress
alb.rule.source_name	Ingress 的名称
alb.rule.source_ns	Ingress 所在的命名空间

- 默认报告的属性，但可以通过修改 flag.hide_upstream_attrs 字段将其排除：

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

- 默认不报告的属性，但可以通过修改 flag.report_http_request_header 字段进行报告：

参数	描述
`**http.request.header.<header>**`	请求标头

- 默认不报告的属性，但可以通过修改 flag.report_http_response_header 字段进行报告：

参数	描述
`**http.response.header.<header>**`	响应标头

配置示例

以下 YAML 配置会部署一个 ALB，并使用 Jaeger 作为 OTel server，以 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。

```yaml
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: theseedoaa/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 以及测试所需的所有 CR。

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

3. 执行以下命令获取 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"
```

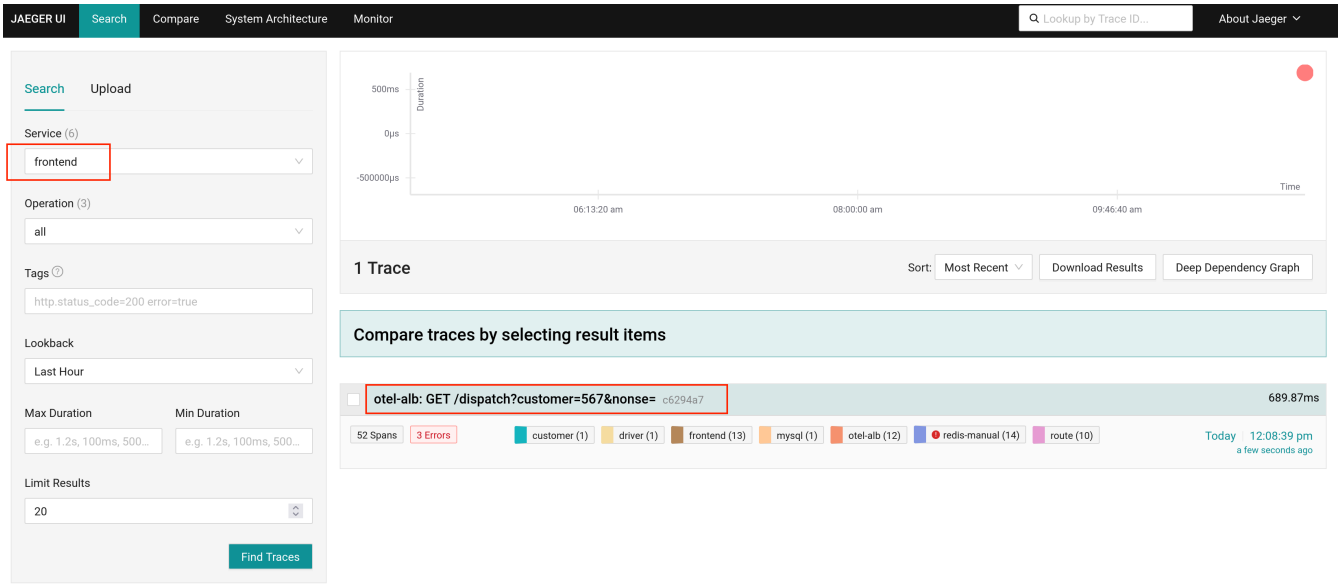
4. 执行以下命令获取 otel-alb 的访问地址。

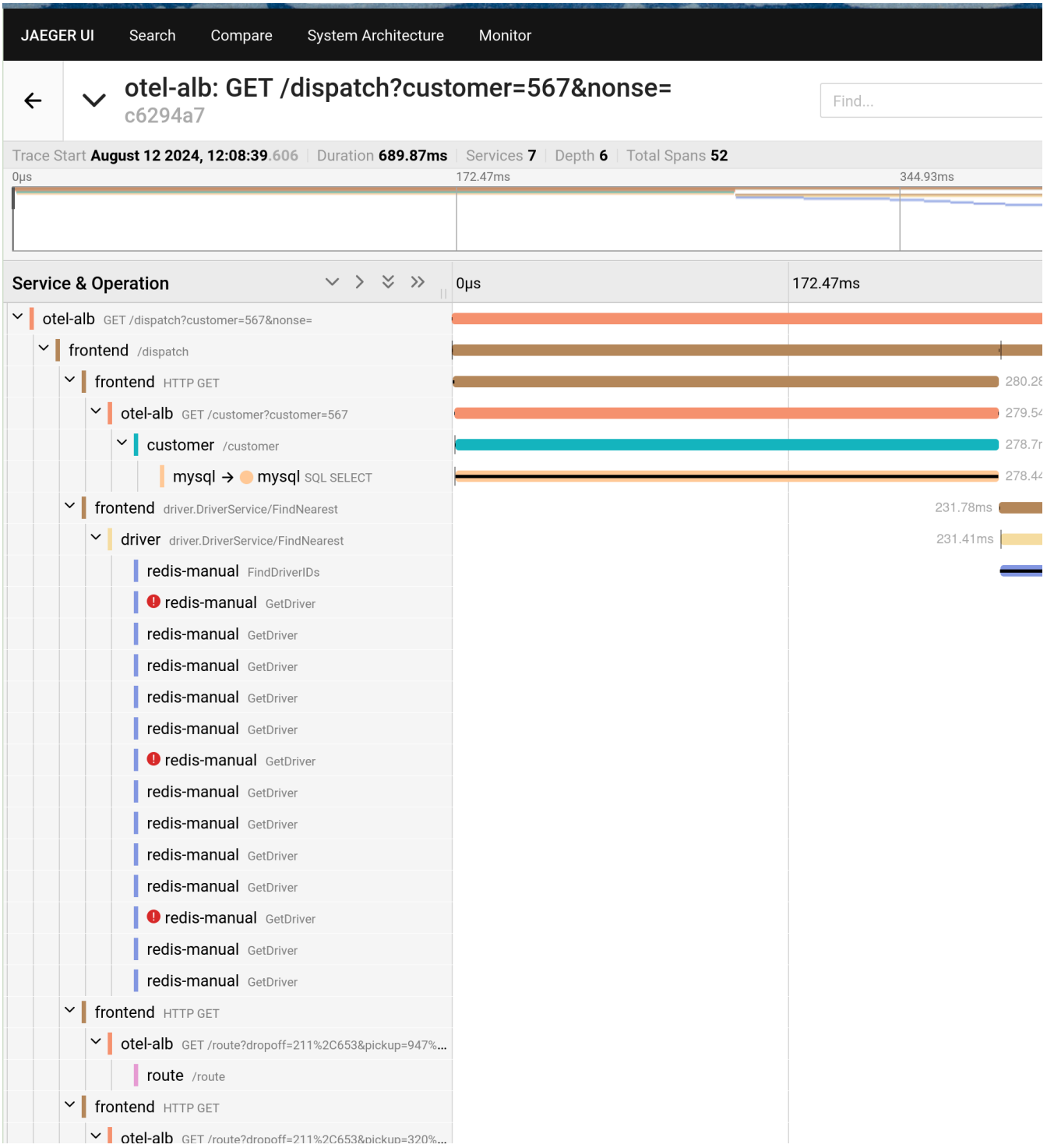
```
export ALB_IP=$(kubectl get po -A -o wide|grep otel-alb | awk '{print $7}');echo $ALB_IP
```

5. 执行以下命令通过 ALB 向 HotROD 发送请求。此时，ALB 将 Trace 报告到 Jaeger。

```
curl -v "http://<$ALB_IP>:80/dispatch?customer=567&nonse=" # Replace <$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，查看结果。


JAEGER UI
Search
Compare
System Architecture
Monitor
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
e.g. 1.2s, 100ms, 500...
Min Duration
e.g. 1.2s, 100ms, 500...
Limit Results
20
Find Traces
500ms
0µs
-500000µs
Duration
Time
06:13:20 am
08:00:00 am
09:46:40 am
1 Trace
Sort:
Most Recent
Download Results
Deep Dependency Graph
Compare traces by selecting result items
otel-alb: GET /dispatch?customer=567&nonse=
c6294a7
689.87ms
52 Spans
3 Errors
customer (1)
driver (1)
frontend (13)
mysql (1)
otel-alb (12)
redis-manual (14)
route (10)
Today 12:08:39 pm
a few seconds ago



商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是独立供应商。本产品与 F5, Inc. 不存在关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

TCP/HTTP Keepalive

目录

基本概念

CRD

商标声明

基本概念

1. ALB 支持在端口级别配置 keepalive。可以在前端进行配置。
2. Keepalive 位于客户端与 **ALB** 之间，而不是 **ALB** 与后端之间。
3. 它通过 Nginx 配置实现，当配置发生更改时，Nginx 需要并将自动重新加载。
4. TCP keepalive 和 HTTP keepalive 是两个不同的概念：
    1. **TCP keepalive** 是 TCP 协议的一项功能，在没有数据传输时定期发送探测数据包，以检查连接是否仍然有效。它有助于检测并清理失效连接。
    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，从而避免建立新连接的开销。通过减少延迟和资源使用量，这可以提升性能。

CRD

```
keepalive:
  properties:
    http:
      description: Downstream L7 keepalive
      properties:
        header_timeout:
          description: Keepalive header timeout. Default is not set.
          type: string
        requests:
          description: Keepalive requests. Default is 1000.
          type: integer
        timeout:
          description: Keepalive timeout. Default is 75s.
          type: string
      type: object
    tcp:
      description: TCPKeepAlive defines TCP keepalive parameters (SO_KEEP
ALIVE)
      properties:
        count:
          description: The TCP_KEEPCNT socket option.
          type: integer
        idle:
          description: The TCP_KEEPIDLE socket option.
          type: string
        interval:
          description: The TCP_KEEPINTVL socket option.
          type: string
      type: object
  type: object
```

只能在 Frontend `.spec.config.keepalive` 上进行配置。

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是独立供应商。本产品与 F5, Inc. 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

ALB 与 Ingress-NGINX 注解兼容性

WARNING

ALB 已弃用。如果需要使用 ingress-nginx 注解，请使用 ingress-nginx-operator。

基本概念

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

ALB 支持其中一部分注解。

支持的 ingress-nginx 注解

Name	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 规则可以配置 cookie 哈希
nginx.ingress.kubernetes.io/use-regex	bool	

Name	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" 或 "persistent"	o ingress 不支持。ALB 规则可以配置会话持久性
nginx.ingress.kubernetes.io/affinity-canary-behavior	"sticky" 或 "legacy"	o ingress 不支持。ALB 规则可以配置会话持久性
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" 或 "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" 或 "false"	x
nginx.ingress.kubernetes.io/auth-tls-match-cn	string	x
nginx.ingress.kubernetes.io/auth-url	string	v
nginx.ingress.kubernetes.io/auth-cache-key	string	x

Name	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-duration	string	x
nginx.ingress.kubernetes.io/auth-keepalive	number	x
nginx.ingress.kubernetes.io/auth-keepalive-share-vars	"true" 或 "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" 或 "false"	o auth
nginx.ingress.kubernetes.io/backend-protocol	string	v
nginx.ingress.kubernetes.io/canary	"true" 或 "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

Name	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 可以使用 ingress 的 default-backend
nginx.ingress.kubernetes.io/enable-cors	"true" 或 "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" 或 "false"	x
nginx.ingress.kubernetes.io/cors-max-age	number	x

Name	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" 或 "false"	v redirect
nginx.ingress.kubernetes.io/from-to-www-redirect	"true" 或 "false"	x
nginx.ingress.kubernetes.io/http2-push-preload	"true" 或 "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" 或 "false"	x
nginx.ingress.kubernetes.io/proxy-body-size	string	x

Name	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 timeout
nginx.ingress.kubernetes.io/proxy-send-timeout	number	v timeout
nginx.ingress.kubernetes.io/proxy-read-timeout	number	v timeout
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" 或 "1.1"	x
nginx.ingress.kubernetes.io/proxy-ssl-secret	string	x

Name	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" 或 "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" 或 "false"	x
nginx.ingress.kubernetes.io/session-cookie-change-on-failure	"true" 或 "false"	x
nginx.ingress.kubernetes.io/session-cookie-conditional-samesite-none	"true" 或 "false"	x
nginx.ingress.kubernetes.io/session-cookie-domain	string	x

Name	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" 或 "false"	v
nginx.ingress.kubernetes.io/ssl-passthrough	"true" 或 "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

Name	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" 或 "false"	x
nginx.ingress.kubernetes.io/connection-proxy-header	string	x
nginx.ingress.kubernetes.io/enable-access-log	"true" 或 "false"	o 默认启用 access_log，格式固定
nginx.ingress.kubernetes.io/enable-opentelemetry	"true" 或 "false"	v otel
nginx.ingress.kubernetes.io/opentelemetry-trust-incoming-span	"true" 或 "false"	v otel
nginx.ingress.kubernetes.io/enable-modsecurity	bool	v modsecurity
nginx.ingress.kubernetes.io/enable-owasp-core-rules	bool	v modsecurity

Name	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 modsecurity
nginx.ingress.kubernetes.io/modsecurity-snippet	string	v modsecurity
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

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是一家独立供应商。本产品与 F5, Inc. 无关联，也未获得其认可或赞助。所有商标均为其各自所有者的财产，在此仅用于标识目的。

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。

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

注意事项

使用 Web 控制台创建 AdminNetworkPolic

使用 CLI 创建 AdminNetworkPolicy 或 Bas

网络策略

通过 Web 控制

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

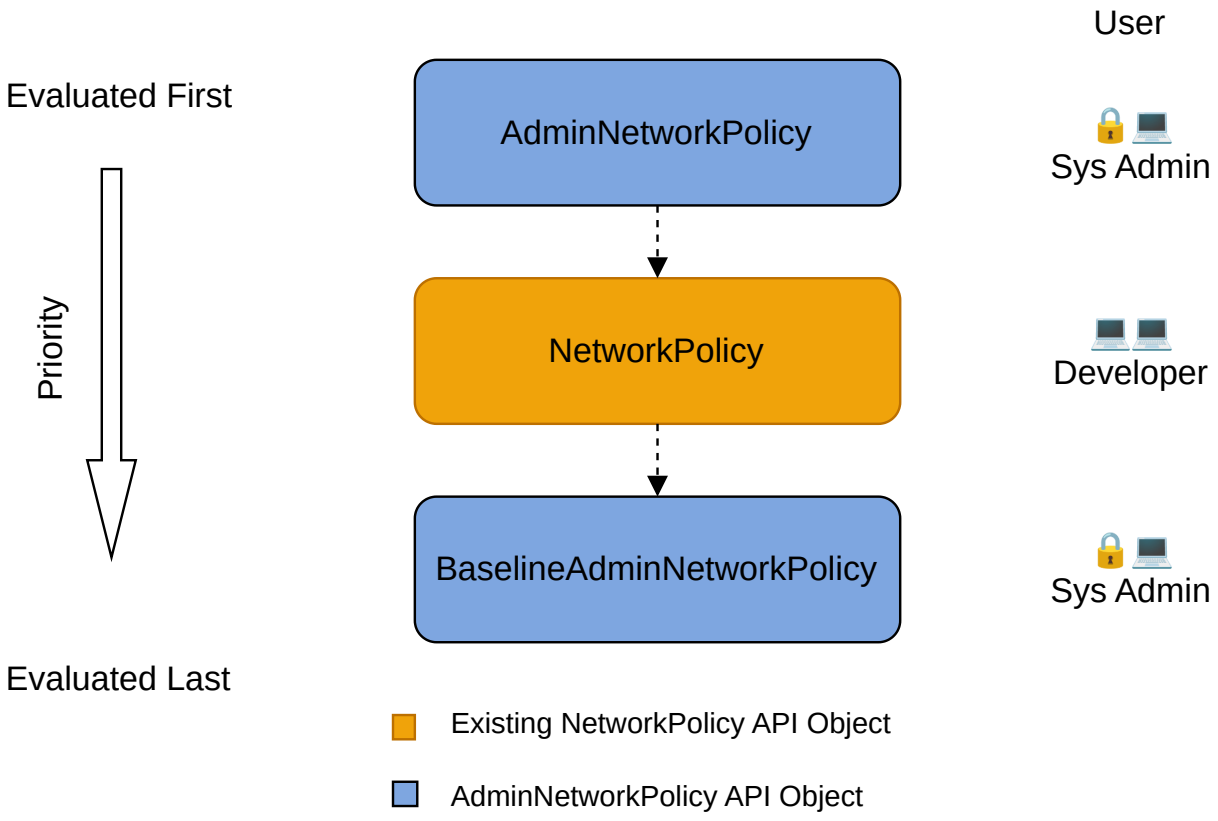
Kubernetes 风格集群中的网络安全涉及控制哪些工作负载（Pods、命名空间）可以通信以及如何通信。其核心是网络策略 API，支持分段（第 3/4 层）和流量控制。随着时间推移，更高级的 API 层出现，以支持集群范围的安全防护、多租户强制执行、零信任模型以及应用层（第 7 层）过滤。

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 `podSelector` 选定的命名空间内 Pods	由顶层 `subject` 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

平台现在为集群网络策略提供两个不同的 UI。旧版 UI 出于兼容性原因仍会保留，而新版 UI 更加灵活，并提供原生 YAML 编辑器。我们建议使用新版 UI。

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

目录

注意事项

使用 Web 控制台创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy

使用 CLI 创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy

注意事项

- 目前仅 Kube-OVN CNI 支持管理员网络策略。
- 在 Kube-OVN 网络模式下，此功能处于 Technology Preview 阶段。上游 API 仍为 `policy.networking.k8s.io/v1alpha1` 并在持续演进，因此请在其文档说明的限制范围内评估此功能，不要将其视为生产环境支持的功能。有关成熟度定义，请参阅版本和生命周期。

- 集群中只能存在一个 Baseline Admin Network Policy。

AdminNetworkPolicy

```yaml
# example-anp.yaml
apiVersion: policy.networking.k8s.io/v1alpha1
kind: AdminNetworkPolicy
metadata:
  name: example-anp
spec:
  priority: 3 (1)
  subject: (2)
    pods:
      namespaceSelector:
        matchLabels: {}
      podSelector:
        matchLabels:
          pod-template-hash: 55f66dd67d
  ingress:
    - name: ingress1
      action: Allow (3)
      ports:
        - portNumber:
            protocol: TCP
            port: 8090
      from: (4)
        - pods:
            namespaceSelector:
              matchLabels: {}
            podSelector:
              matchLabels:
                pod-template-hash: 55c84b59bb
  egress:
    - name: egress1
      action: Allow
      ports:
        - portNumber:
            protocol: TCP
            port: 8080
      to: (5)
        - networks:
            - 10.1.1.1/23
```

1. 数字越小，优先级越高。
2. `subject`：命名空间选择器和 Pod 选择器中最多只能指定一个。
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、跳过后续低优先级的集群网络策略，并继续由其他策略（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：

```yaml
# default.yaml
apiVersion: policy.networking.k8s.io/v1alpha1
kind: BaselineAdminNetworkPolicy
metadata:
  name: default (1)
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 (2)
      ports:
        - portNumber:
            protocol: TCP
            port: 8080
      to:
        - networks:
            - 3.3.3.3/23
```

(1) 集群中只能创建一个 metadata.name=`default` 的基线管理员网络策略。

(2) 可用值为 Allow、Deny。

使用 Web 控制台创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy

1. 转到管理员。
2. 在左侧导航栏中，单击网络 > 集群网络策略。
3. 单击创建管理员网络策略或配置基线管理员网络策略。
4. 按照以下说明完成相关配置。

区域	参数	描述
基本信息	名称	Admin Network Policy 或 Baseline Admin Network Policy 的名称。
	优先级	确定评估和应用策略的顺序。数值越小表示优先级越高。 注意：基线管理员网络策略没有优先级。
目标 Pod	命名空间选择器	以键值对形式输入目标命名空间的标签。如果未设置，该策略将应用于当前集群中的所有命名空间。指定后，该策略将仅应用于位于与这些选择器匹配的命名空间中的 Pod。
	当前策略影响的目标 Pod 预览	单击预览以查看此网络策略影响的目标 Pod。
	Pod 选择器	以键值对形式输入目标 Pod 的标签。如果未设置，该策略将应用于当前命名空间中的所有 Pod。

区域	参数		描述
Ingress	当前策略影响的目标 Pod 预览		单击预览以查看此网络策略影响的目标 Pod。
	流量操作		指定如何处理进入目标 Pod 的流量。包含三种模式：**Allow**（允许流量）、**Deny**（阻止流量）和 **Pass**（跳过所有优先级较低的管理员网络策略，使流量由 Network Policy 处理；如果不存在 Network Policy，则由 Baseline Admin Network Policy 处理）。 注意：基线管理员网络策略不支持 **Pass** 操作。
	规则 描述：如果在规则中添加了多个源，它们之间是逻辑或关系。	Pod 选择器	匹配集群中带有指定标签的命名空间或 Pod；只有匹配的 Pod 才能访问目标 Pod。可以单击预览查看当前规则影响的 Pod。 • 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，即从指定的命名空间中选择带有指定标签的 Pod。 • 如果未配置此项，则默认情况下，集群中所有命名空间内的所有 Pod 都可以访问目标 Pod。
		命名空间选择器	匹配当前命名空间中带有指定标签的 Pod；只有匹配的 Pod 才能访问目标 Pod。可以单击预览查看当前规则影响的 Pod。如果

区域	参数		描述
			未配置此项，则默认情况下，当前命名空间中的所有 Pod 都可以访问目标 Pod。
	端口		匹配指定协议和端口上的流量；可以添加数字端口或 Pod 上的端口名称。如果未配置此项，则会匹配所有端口。
Egress	规则	节点选择器	指定允许目标 Pod 访问哪些节点 IP。可以按节点标签选择节点，以控制 Pod 可访问的节点 IP。
	描述：如果在规则中添加了多个源，它们之间是逻辑或关系。	IP 范围	指定允许目标 Pod 连接的 CIDR 范围。如果未配置此项，则默认情况下，目标 Pod 可以连接到任意 IP。
		其他参数	与 Ingress 参数类似，具有相同的配置选项和行为。

使用 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：

```yaml
# 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

参考

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 `spec.policyTypes` 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 `spec.ingress`、`spec.egress` 和 `spec.policyTypes` 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
	规则 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。	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。
		当前集群内的 Pod	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。 • 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 *example_ip*/32 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 `spec.policyTypes` 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

故障排除

商标声明

使用 Envoy Gateway 进行 Ingress 和负载均衡

目录

概述
网络流架构
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（粘性会话）
TLS 和安全性
TLS 终止
TLS 直通
Gateway 与服务之间的 mTLS
与 MetalLB 集成
故障排除
商标声明

概述

Envoy Gateway 为 Kubernetes 集群提供第 7 层（L7）Ingress 和负载均衡功能。 它基于 Kubernetes Gateway API 构建，并基于 Envoy Proxy，支持高级 HTTP、HTTPS、gRPC 和 TCP 路由，同时提供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， 为传统 Ingress 对象提供了标准且可移植的替代方案。

网络流架构

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

- 外部 LoadBalancer 提供外部 IP，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（ClusterIP）将流量分发到各节点上的 Pod。

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

配置路由

定义 **GatewayClass**

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

创建 **Gateway**

```yaml
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
```

了解更多信息，请参阅配置 Gateway API Gateway。

创建 HTTP 路由

```yaml
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
```

了解更多信息，请参阅配置 Gateway API Route。

说明

- Gateway 为 HTTP/S 流量公开端口 80 和 443。
- HTTPRoute 根据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 支持多种外部流量入口模式，具体取决于您的基础设施。

环境	入口类型	示例
Cloud	`Service type=LoadBalancer`	云服务提供商分配公共 IP

环境	入口类型	示例
裸金属	MetalLB + `Service type=LoadBalancer`	MetalLB 分配外部 IP
边缘 / 内部	NodePort 或 hostNetwork	用于本地或私有访问

公开 Envoy Gateway 的示例 Service：

```yaml
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
```

您可以运行以下命令验证分配的外部 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（粘性会话）

Envoy 可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
```

详细了解 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 直通

用于端到端加密，Envoy 转发加密流量：

```
tls:
  mode: Passthrough
```

Gateway 与服务之间的 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 Service 的类型为 LoadBalancer。
3. MetalLB 会自动分配外部 IP。
4. 然后，DNS 记录可以将域名（例如 app.example.com）映射到分配的 IP。

故障排除

问题	可能原因	解决方案
Gateway 没有外部 IP	MetalLB 配置错误	检查 `Service` 类型和 MetalLB controller
HTTPRoute 被忽略	`parentRefs` 不匹配	确保使用正确的 `Gateway` 和命名空间
TLS 无法正常工作	未找到 Secret	验证 `certificateRefs` 路径
Gateway 返回 404	没有匹配的主机 / 路径	确认 `HTTPRoute` 规则
延迟突增	代理过载	扩展 Envoy Gateway 副本

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

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

配置服务

为什么需要 Ser

ClusterIP 类型 S

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 Alauda Container Platform 负载均衡器

前提条件

配置 MetalLB 组件

配置 Gatewa

配置 GatewayAPI 策略

概述

前提条件

策略附加基础

策略附加摘要

在 Web Console 中创建 Policy

SecurityPolicy

础知

ay

ay 设

Rou

BackendTLSPolicy
ClientTrafficPolicy
BackendTrafficPolicy
相关任务
商标声明

配置 GatewayAPI ListenerSet

概述
先决条件

配置 EnvoyProxy 资源配额

概述
配额模型
创建配额
配置 EnvoyProxy 请求
准入行为
监控 namespace 使用量
范围和限制
下一步
商标声明

配置 ALB

ALB

配置 Alauda Container Platform 节点 DNS 缓存

概述

配置 Alauda Container Platform 集群 DNS

概述
配置
商标声明

商标声明

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）

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

(1) 如果启用了证书，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 `developer.test.cn`。

参数	描述
	• Wildcard Domain：带通配符（*）的泛域名，例如 `*.test.cn`，包含域名 `test.cn` 下的所有子域名。
域名	根据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：

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

Cluster B 的域名：

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

配置服务

在 Kubernetes 中，Service 是一种用于暴露集群中作为一个或多个 Pod 运行的网络应用的方法。

目录

为什么需要 Service
ClusterIP 类型 Service 示例：
无头服务
使用 Web 控制台创建服务
使用 CLI 创建服务
示例：访问集群内的应用
示例：访问集群外的应用
示例：ExternalName 类型的服务
LoadBalancer 类型 Service 注解
AWS EKS 集群
Huawei Cloud CCE 集群
Azure AKS 集群
Google GKE 集群
示例：使用 MetalLB BGP 和本地流量策略的 LoadBalancer
优势
前提条件
步骤
关键配置点

externalTrafficPolicy: Local

使用 BGP 的 LoadBalancer

部署步骤

验证

商标声明

为什么需要 **Service**

1. Pod 拥有自己的 IP，但：
   - Pod IP 不稳定（Pod 重新创建时会发生变化）。
   - 直接访问 Pod 会变得不可靠。
2. Service 通过提供以下功能解决了这一问题：
   - 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：

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

1 可用的类型值及其行为分别为 `ClusterIP`、`NodePort`、`LoadBalancer`、`ExternalName`

2 Service 目标 Pod 的集合通常由您定义的 selector 确定。

3 `Service` 端口。

4 将 Service 的 `targetPort` 绑定到 Pod 的 `containerPort`。此外，您可以在 Pod 容器下引用 `port.name`。

无头服务

有时您不需要负载均衡和单个 Service IP。在这种情况下，您可以创建所谓的无头服务：

```
spec:
  clusterIP: None
```

无头服务适用于以下场景：

- 您希望发现单个 Pod IP，而不仅仅是单个 Service IP。
- 您需要直接连接到每个 Pod（例如，用于 Cassandra 等数据库或 StatefulSet）。
- 您使用的是 StatefulSet，其中每个 Pod 都必须具有稳定的 DNS 名称。

使用 **Web** 控制台创建服务

1. 转到 容器平台。
2. 在左侧导航栏中，单击 网络 > 服务。
3. 单击 创建服务。
4. 按照以下说明配置相关参数。

参数	描述
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，将创建无头服务，通常由 **StatefulSet** 使用。

参数	描述
类型	• **ClusterIP**：在集群内部 IP 上暴露 Service。选择此值后，Service 只能从集群内部访问。 • **NodePort**：在每个节点的 IP 上通过静态端口（NodePort）暴露 Service。 • **ExternalName**：将 Service 映射到 externalName 字段的内容（例如，主机名 api.foo.bar.example）。 • **LoadBalancer**：使用外部负载均衡器在集群外部暴露 Service。Kubernetes 不直接提供负载均衡组件；您必须自行提供，或者将 Kubernetes 集群与云服务提供商集成。
目标组件	• 工作负载：Service 会将请求转发到特定工作负载，该工作负载匹配类似 `project.cpaas.io/name: projectname` 和 `service.cpaas.io/name: deployment-name` 的标签。 • 虚拟化：Service 会将请求转发到特定虚拟机或虚拟机组。 • 标签选择器：Service 会将请求转发到具有指定标签的特定类型工作负载，例如 `environment: release`。

参数	描述
端口	用于配置此 Service 的端口映射。在以下示例中，集群内的其他 Pod 可以通过虚拟 IP（如果已启用）和 TCP 端口 *80* 调用此 Service；访问请求将转发到目标组件 Pod 对外暴露的 TCP 端口 *6379* 或 *redis*。 • 协议：Service 使用的协议，支持的协议包括：`TCP`、`UDP`、`HTTP`、`HTTP2`、`HTTPS`、`gRPC`。 • 服务端口：Service 在集群内暴露的服务端口号，即 Port，例如 *80*。 • 容器端口：服务端口映射到的目标端口号（或名称），即 targetPort，例如 *6379* 或 *redis*。 • 服务端口名称：将自动生成。格式为 `<protocol>-<service port>-<container port>`，例如：*tcp-80-6379* 或 *tcp-80-redis*。
会话亲和性	基于源 IP 地址（ClientIP）的会话亲和性。启用后，在负载均衡期间，来自同一 IP 地址的所有访问请求都将保持在同一服务器上，从而确保来自同一客户端的请求被转发到同一服务器进行处理。

5. 单击 创建。

使用 **CLI** 创建服务

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

基于现有的 Deployment 资源 `my-app` 创建服务。

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

示例：访问集群内的应用

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

1. 应用此 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` 服务：

```
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
```

你应会看到包含类似“Welcome to nginx!”文本的 HTML 响应。

示例：访问集群外的应用

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

1. 应用此 YAML：

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

2. 检查 Pod：

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

3. 使用 curl 访问 Service：

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

你应会看到包含类似“Welcome to nginx!”文本的 HTML 响应。

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

注意：请提前配置 LoadBalancer Service。

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

1. 应用此 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` 是你在浏览器中访问的地址。

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

你应会看到包含类似“Welcome to nginx!”文本的 HTML 响应。

如果 EXTERNAL-IP 为 `pending`，则表示 LoadBalancer Service 当前未部署在集群上。

示例：**ExternalName** 类型的服务

```yaml
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. 尝试在集群中的某个 Pod 内解析：

```
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`。

LoadBalancer 类型 Service 注解

AWS EKS 集群

有关 EKS LoadBalancer Service 注解的详细说明，请参阅注解使用文档↗。

Key	Value	Description
service.beta.kubernetes.io/aws-load-balancer-type	external：使用官方 AWS LoadBalancer Controller。	指定 LoadBalancer 类型使用的控制器。 注意：请提前联系平台管理员部署 AWS LoadBalancer Controller。
service.beta.kubernetes.io/aws-load-balancer-nlb-target-type	• instance：流量将通过 NodePort 发送到 Pod。 • ip：流量直接路由到 Pod（集群必须使用 Amazon VPC CNI）。	指定流量到达 Pod 的方式。
service.beta.kubernetes.io/aws-load-balancer-scheme	• internal：私有网络。 • internet-facing：公共网络。	指定使用私有网络还是公共网络。
service.beta.kubernetes.io/aws-load-balancer-ip-address-type	• IPv4 • dualstack	指定支持的 IP 地址栈。

Huawei Cloud CCE 集群

有关 CCE LoadBalancer Service 注解的详细说明，请参阅注解使用文档↗。

Key	Value
kubernetes.io/elb.id	

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

kubernetes.io/elb.subnet-id	
kubernetes.io/elb.class	• union：共享型负载均衡。 • performance：独享型负载均衡，仅支持 Kubernetes 1.17 及更高版
kubernetes.io/elb.enterpriseID	

Azure AKS 集群

有关 AKS LoadBalancer Service 注解的详细说明，请参阅注解使用文档↗。

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

Google GKE 集群

有关 GKE LoadBalancer Service 注解的详细说明，请参阅注解使用文档↗。

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

示例：使用 MetalLB BGP 和本地流量策略的 LoadBalancer

此示例演示如何使用 MetalLB BGP 模式和 `externalTrafficPolicy: Local` 配置 LoadBalancer Service，从而实现无需额外网络跳转的双活负载均衡。

优势

- 双活负载均衡：流量可同时分发到多个节点

- 无额外网络跳转：无需通过中间节点转发即可直接路由到 Pod
- 更高性能：`externalTrafficPolicy: Local` 可保留源 IP 并降低延迟
- 高可用性：BGP 路由通告确保流量能够到达正常运行的节点

前提条件

配置 LoadBalancer Service 前，请确保已完成以下准备：

1. 已部署 **MetalLB**：安装方法请参阅创建外部 IP 地址池
2. 已配置 **BGP Peer**：BGP 配置方法请参阅创建 BGP Peer
3. 外部 **IP** 地址池：配置带有 BGPAdvertisement 的 IPAddressPool

步骤

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

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

关键配置点

externalTrafficPolicy: Local

`externalTrafficPolicy: Local` 设置具有以下优势：

- 保留源 **IP**：保留客户端源 IP，从而支持正确的日志记录和安全策略
- 直接路由到 **Pod**：流量直接发送到 Pod，无需节点级转发

使用 **BGP** 的 **LoadBalancer**

使用 MetalLB BGP 模式时：

- 路由将从 BGPAdvertisement nodeSelectors 指定的节点进行通告
- BGP Peer 接收这些通告，并据此路由流量
- BGPPeer 与 BGPAdvertisement 之间的节点选择器保持一致，可确保路由一致性

部署步骤

1. 部署应用：

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

2. 验证 **LoadBalancer Service**：

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

   预期输出：

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

3. 测试服务：

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

验证

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

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

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是一家独立供应商。本产品与 The Linux Foundation 或 F5, Inc. 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，仅用于标识目的。

配置 Ingress

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

创建 Ingress 以管理对 Service 的外部 HTTP/HTTPS 访问。

> **WARNING**
>
> 在同一命名空间中创建多个 Ingress 时，不同 Ingress 不得具有相同的 **Domain**、**Protocol** 和 **Path**（即不允许存在重复的访问点）。

目录

实现方式

Ingress 示例：

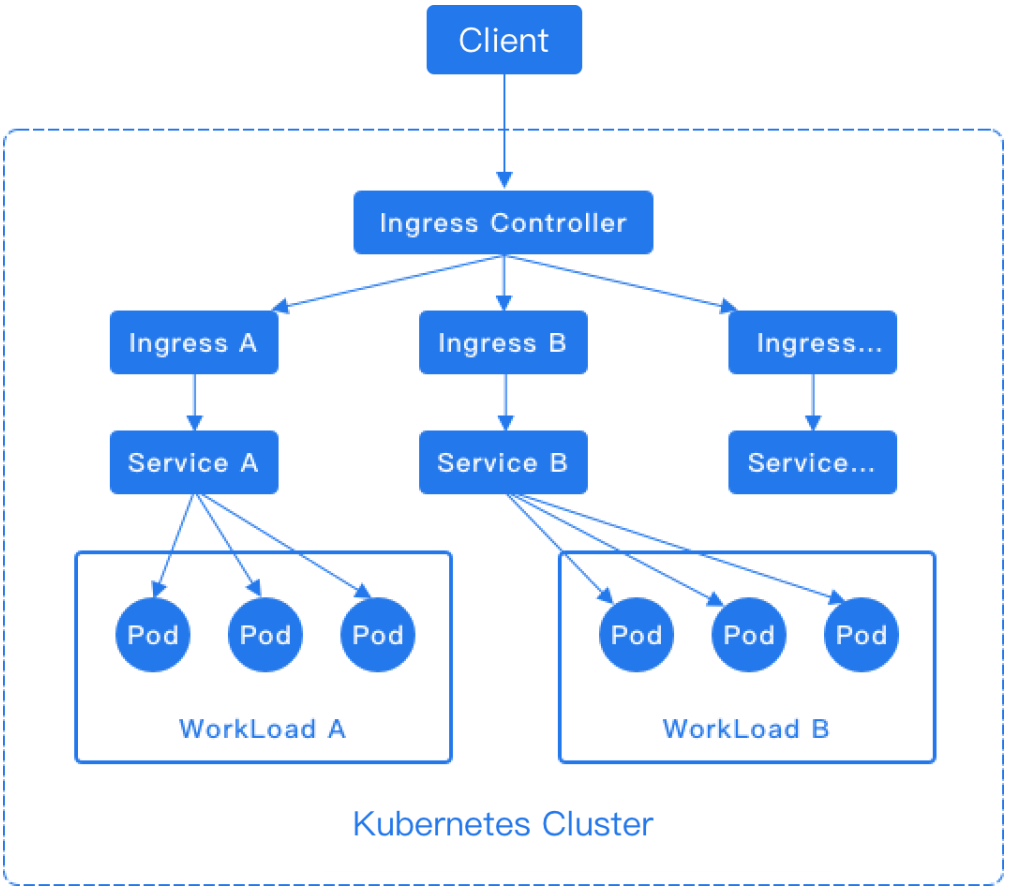
使用 Web 控制台创建 Ingress

使用 CLI 创建 Ingress

商标声明

实现方式

Ingress 规则取决于 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，会根据 Ingress 中的转发规则进行匹配，并将流量分发到指定的内部路由，如下图所示。



NOTE

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

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

Ingress 示例：

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

1 如需了解更多配置，请参阅 nginx-configuration↗。

2 `nginx` 用于使用 `ingress-nginx` controller，`$alb_name` 用于使用 alb 作为 ingress controller。

3 如果只希望在本地运行 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	Ingress 可以由具有不同 `IngressClass` 名称的不同 controller 实现。如果平台上有多个 Ingress controller 可用，用户可以通过此选项选择要使用的 controller。
Domain Name	Host 可以是精确匹配（例如 `foo.bar.com` ）或通配符（例如 `*.foo.com` ）。可用的域名由平台管理员分配。
Certificates	由平台管理员分配的 TLS secret 或证书。
Match Type 和 **Path**	• **Prefix**：匹配路径前缀，例如，`/abcd` 可以匹配 `/abcd/efg` 或 `/abcde` 。 • **Exact**：匹配精确路径，例如 `/abcd` 。 • **Implementation specific**：如果使用自定义 Ingress controller 管理 Ingress 规则，可以选择由该 controller 决定。
Service	外部流量将转发到此 Service。
Service Port	指定将流量转发到的 Service 端口。

5. 点击 **Create**。

使用 **CLI** 创建 **Ingress**

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

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是一家独立供应商。本产品与 F5, Inc. 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识。

配置子网

目录

IP 分配规则

Calico 网络

约束和限制

使用 Calico 网络的子网自定义资源（CR）示例

使用 Web 控制台在 Calico 网络中创建子网

使用 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 Subnet 自定义资源（CR）示例

使用 Web 控制台在 Kube-OVN Overlay 网络中创建子网

使用 CLI 在 Kube-OVN Overlay 网络中创建子网

Underlay 网络

使用说明

使用 Web 控制台添加网桥网络（可选）

使用 CLI 添加网桥网络

使用 Web 控制台添加 VLAN（可选）

使用 CLI 添加 VLAN

使用 Kube-OVN Underlay 网络的 Subnet 自定义资源（CR）示例

使用 Web 控制台在 Kube-OVN Underlay 网络中创建子网

使用 CLI 在 Kube-OVN Underlay 网络中创建子网

相关操作
子网管理
使用 Web 控制台更新网关
使用 CLI 更新网关
使用 Web 控制台更新保留 IP
使用 CLI 更新保留 IP
使用 Web 控制台分配项目
使用 CLI 分配项目
使用 Web 控制台分配命名空间
使用 CLI 分配命名空间
使用 Web 控制台扩展子网
使用 CLI 扩展子网
管理 Calico 网络
使用 Web 控制台删除子网
使用 CLI 删除子网

IP 分配规则

> **NOTE**
>
> 如果一个项目或命名空间分配了多个子网，将从其中一个子网中随机选择一个 IP 地址。

- 项目分配：
  - 如果项目未绑定子网，则该项目下所有命名空间中的 Pod 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，Pod 将无法启动。
  - 如果项目已绑定子网，则该项目下所有命名空间中的 Pod 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子网，则该命名空间中的 Pod 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，Pod 将无法启动。
- 如果命名空间已绑定子网，则该命名空间中的 Pod 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，以实现对集群内资源更细粒度的网络隔离。

约束和限制

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

使用 Calico 网络的子网自定义资源（CR）示例

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

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

使用 Web 控制台在 Calico 网络中创建子网

1. 前往管理员。
2. 在左侧导航栏中，单击网络 > 容器网络 > 子网。
3. 单击创建子网。
4. 参考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，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：有关 CIDR 与 BlockSize 的对应关系，请参阅参考内容。
封装协议	选择封装协议。双栈模式不支持 **IPIP**。 • **IPIP**：使用 IPIP 协议实现网段间通信。 • **VXLAN**（**Alpha**）：使用 VXLAN 协议实现网段间通信。 • 无封装：通过路由转发直接连接。
封装模式	当封装协议为 **IPIP** 或 **VXLAN** 时，必须设置封装模式，默认为始终。 • 始终：始终启用 IPIP / VXLAN 隧道。 • 跨子网：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，通过路由转发直接连接。
出站流量 **NAT**	选择是否启用出站流量 NAT（网络地址转换），默认启用。 该功能主要用于设置子网容器组访问外部网络时向外部网络公开的访问地址。 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，子网中容器组的 IP 将直接公开给外部网络。

5. 单击确认。
6. 在子网详情页面中，选择操作 > 分配项目 / 分配命名空间。
7. 完成配置，然后单击分配。

使用 **CLI** 在 **Calico** 网络中创建子网

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

参考内容

下表显示了 CIDR 与 blockSize 之间的动态匹配关系。

CIDR	blockSize 大小	主机数量	单个 IP 池的大小
`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，以实现集群中资源更细粒度的网络隔离。

> **NOTE**
>
> 平台内置了用于节点和 Pod 之间通信的 **join** 子网；请避免 **join** 与新创建的子网之间的网络段发生冲突。

使用 Kube-OVN Overlay 网络的 Subnet 自定义资源（CR）示例

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

1. 请参阅出站流量 NAT 参数。
2. 请参阅保留 IP 参数。
3. 请参阅网关类型参数。可用值为 `distributed` 或 `centralized`。

4. 请参阅网关节点参数。
5. 请参阅 ECMP 参数。使用此功能前，请联系管理员启用功能门控。

使用 **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 > 容器网络 > 子网。
3. 点击创建子网。
4. 参考以下说明配置相关参数。

参数	描述
网络段	将子网分配给项目或命名空间后，该网络段内的 IP 将随机分配给 Pod 使用。
保留 **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**。
网关类型	选择子网的网关类型，以控制出站流量。 - 分布式：集群中的每个主机都可以作为当前主机上 Pod 的出站节点，从而实现分布式出口。 - 集中式：集群中的所有 Pod 使用一个或多个特定主机作为出站节点，便于进行外部审计和防火墙控制。设置多个集中式网关节点可以实现高可用性。
ECMP（**Alpha**）	选择集中式网关时，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（等价多路径路由）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，从而提高网关的总吞吐量。 注意：请提前启用与 ECMP 相关的功能。
网关节点	使用集中式网关时，选择一个或多个特定主机作为网关节点。

参数	描述
出站流量 **NAT**	选择是否启用出站流量 NAT（网络地址转换）。默认情况下已启用。 主要用于设置子网中的 Pod 访问互联网时对外部网络暴露的访问地址。 启用出站流量 NAT 时，将使用主机 IP 作为当前子网中 Pod 的访问地址；未启用时，子网中 Pod 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. 在子网详情页面中，选择操作 > 分配项目 / 命名空间。
7. 完成配置并点击分配。

使用 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，不仅可以为资源提供更细粒度的网络隔离，还可以带来更好的性能体验。

> **INFO**
>
> Kube-OVN Underlay 中的容器网络需要物理网络提供支持。请参阅最佳实践准备 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：添加网桥网络 > 添加 VLAN > 创建子网。

1. 默认网卡名称。
2. 按节点配置网卡。

使用 Web 控制台添加网桥网络（可选）

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

1 默认网卡名称。

2 按节点配置网卡。

网桥网络是指网桥。将网卡绑定到网桥后，网桥可以转发容器网络流量，从而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 > 容器网络 > 网桥网络。
3. 点击添加网桥网络。
4. 根据以下说明配置相关参数。
   注意：

   - *目标 Pod* 是指调度到当前节点的所有 Pod，或调度到当前节点且位于绑定到特定子网的命名空间中的 Pod。这取决于网桥网络下子网的作用域。
   - Underlay 子网中的节点必须具有多个网卡，并且网桥网络使用的网卡必须专门分配给 Underlay，不能承载其他流量，例如 SSH。例如，如果网桥网络中的三个节点计划分别使用 eth0、eth0、eth1 专供 Underlay 使用，则默认网卡可以设置为 eth0，第三个节点的网卡可以设置为 eth1。

参数	描述
默认网卡名称	默认情况下，目标 Pod 将使用此网卡作为网桥网络卡，以实现与物理网络的互通。
按节点配置网卡	配置节点上的目标 Pod 将桥接到指定网卡，而不是默认网卡。
排除节点	排除节点后，调度到这些节点的所有 Pod 都不会桥接到这些节点上的任何网卡。 注意：排除节点上的 Pod 将无法与物理网络通信，也无法跨节点访问容器网络，请注意避免将相关 Pod 调度到这些节点。

5. 点击添加。

使用 **CLI** 添加网桥网络

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

使用 **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. 网桥网络引用。

平台预配置了一个 **ovn-vlan** 虚拟局域网，该网络将连接到 **provider** 网桥网络。也可以配置新的 VLAN 以连接到其他网桥网络，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 > 容器网络 > **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	此 VLAN 的唯一标识符，用于区分不同的虚拟局域网。
网桥网络	VLAN 将连接到此网桥网络，以实现与物理网络的互通。

5. 点击添加。

使用 **CLI** 添加 **VLAN**

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

使用 **Kube-OVN Underlay** 网络的 **Subnet** 自定义资源（**CR**）示例

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

(1) VLAN 引用。

使用 **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 **join** 子网，用于 Overlay 传输模式下节点和 Pod 之间的通信。在 Underlay 传输模式下不会使用此子网，因此必须避免 **join** 与其他子网之间的 IP 网段发生冲突。

操作步骤：

1. 进入管理员。
2. 在左侧导航栏中，点击网络 > 容器网络 > 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	子网所属的 VLAN。

参数	描述
子网	将子网分配给项目或命名空间后，物理子网中的 IP 将随机分配给 Pod 使用。
网关	上述子网中的物理网关。
保留 **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. 在子网详情页面中，选择操作 > 分配项目 / 命名空间。
7. 完成配置并点击分配。

使用 **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，可以根据需要配置Underlay 和 Overlay 子网之间的自动互通。

如果 Underlay 子网中的工作负载需要 IPv4 多播通信，请参阅为 Kube-OVN Underlay 网络配置 IPv4 多播。

子网管理

使用 **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. 前往管理员。
2. 在左侧边栏中，点击网络 > 容器网络 > 子网。
3. 点击子网名称。

4. 选择操作 > 更新网关。
5. 更新参数配置；详情请参阅参数说明。
6. 点击确定。

使用 CLI 更新网关

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

使用 Web 控制台更新保留 IP

无法从保留 IP 中移除网关 IP，但可以自由编辑、删除或添加其他保留 IP。

1. 前往管理员。
2. 在左侧边栏中，点击网络 > 容器网络 > 子网。
3. 点击子网名称。
4. 选择操作 > 更新保留 **IP**。
5. 完成更新后，点击更新。

使用 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"]
  }
]'
```

使用 Web 控制台分配项目

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

1. 前往管理员。
2. 在左侧边栏中，点击网络 > 容器网络 > 子网。
3. 点击子网名称。
4. 选择操作 > 分配项目。
5. 添加或移除项目后，点击分配。

使用 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"
        }
      }
    ]
  }
]'
```

使用 Web 控制台分配命名空间

将子网分配给特定命名空间可以实现更精细的网络隔离。

注意：分配过程将重建网关，并且出站数据包将被丢弃！请确保当前没有业务应用访问外部集群。

1. 前往管理员。
2. 在左侧边栏中，点击网络 > 容器网络 > 子网。

3. 点击子网名称。
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。

使用 CLI 分配命名空间

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

使用 Web 控制台扩展子网

当子网的保留 IP 范围达到使用上限或即将耗尽时，可以基于原始子网范围进行扩展，而不会影响现有服务的正常运行。

1. 前往管理员。
2. 在左侧边栏中，点击网络 > 容器网络 > 子网。
3. 点击子网名称。
4. 选择操作 > 扩展子网。
5. 完成配置并点击更新。

使用 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"
  }
]'
```

NOTE

扩展默认 **Pod** 子网

如果集群中的 Pod IP 地址即将耗尽，可以使用相同方式扩展默认 Pod 子网（`ovn-default`）。现有 Pod 会保留其地址，新 Pod 将从扩展后的范围中获取地址。

只能在保持相同网络基地址的情况下扩大子网（缩短网络前缀）；不能更改基地址或缩小范围。如果默认 Pod 子网无法就地扩展，例如更大的范围无法与当前基地址对齐，则应添加其他子网，并将其绑定到需要更多地址的命名空间。

集群创建后无法更改 Service CIDR。请在安装前规划 Service 网络大小。

管理 **Calico** 网络

支持分配项目和命名空间；详情请参阅项目分配和命名空间分配。

使用 **Web** 控制台删除子网

NOTE

- 删除子网时，如果仍有容器组使用子网中的 IP，容器组可以继续运行，且 IP 地址保持不变，但它们将无法通过网络进行通信。可以重建容器组以使用默认子网中的 IP，或将新子网分配给容器组所在的命名空间以供使用。
- 无法删除默认子网。

1. 前往管理员。
2. 在左侧导航栏中，点击网络 > 容器网络 > 子网。
3. 点击⋮> 删除，然后继续执行删除操作。

使用 **CLI** 删除子网

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

配置 Alauda Container Platform 负载均衡器

目录

前提条件
配置 MetalLB 组件
使用 Web 控制台配置外部 IP 池
使用 Web 控制台配置 BGP 对等节点
使用 CLI 配置带有 L2Advertisement 或 BGPAdvertisement 的外部 IP 池
监控 MetalLB
查看内置 MetalLB 监控面板
访问 MetalLB 指标端点
MetalLB 指标参考
MetalLB 故障排除
商标声明

前提条件

请确保在继续操作前已阅读安装文档。

配置 MetalLB 组件

安装集群插件后，配置 `MetalLB` 自定义资源的 `spec.controllerNodeSelector`、`spec.controllerTolerations`、`spec.nodeSelector` 和 `spec.speakerTolerations` 字段。有关配置示例以及与 `L2Advertisement.spec.nodeSelectors` 的关系，请参阅配置组件放置。

要调整 controller 和 speaker 的日志级别，请在同一资源上设置 `spec.logLevel`。有关支持的值，请参阅调整日志级别。

使用 Web 控制台配置外部 IP 池

1. 前往管理员。
2. 在左侧导航栏中，依次点击网络 > 外部 **IP** > 外部 **IP** 池。
3. 点击创建外部 **IP** 池。
4. 按照以下说明配置相应参数。

参数	描述
类型	• L2：基于 MAC 地址进行通信和转发，适用于需要简单、快速二层交换的小型网络或局域网，具有配置简单、延迟低等优势。 • BGP（Alpha）：基于 IP 地址进行路由和转发，并使用 BGP 协议交换路由信息，适用于需要在多个自治系统间进行复杂路由的大型网络，具有可扩展性强、可靠性高等优势。
IP 资源	支持以 CIDR 和 IP 范围格式输入。点击添加可添加多个条目，示例如下： **CIDR**：`192.168.1.1/24`。 **IP** 范围：`192.168.2.1` ~ `192.168.2.255`。
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、与对等节点建立 BGP 连接并向外部通告路由。 • 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的可用节点。

参数	描述
	注意： • 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。 • 可以分别配置标签选择器或节点名称来选择可用节点；如果同时配置两者，最终的可用节点将是二者的交集。
BGP 对等节点	选择 BGP 对等节点；具体配置请参阅 BGP 对等节点。

5. 点击创建。

使用 **Web** 控制台配置 **BGP** 对等节点

1. 转到管理员。
2. 在左侧导航栏中，点击网络 > 外部 **IP** > **BGP** 对等节点。
3. 点击创建 **BGP** 对等节点。
4. 参考以下说明配置相应参数。

参数	描述
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 编号应与对等节点 AS 编号保持一致。
对等节点 **AS** 编号	BGP 对等节点所在 AS 的 AS 编号。
对等节点 **IP**	BGP 对等节点的 IP 地址，必须是能够建立 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 地址时，选择指定的本地 IP，以便与对等节点建立 BGP 连接。

参数	描述
对等节点端口	BGP 对等节点的端口号。
BGP 连接节点	建立 BGP 连接的节点。如果未配置此参数，则所有节点都会建立 BGP 连接。
eBGP 多跳	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，允许跨多个中间网络设备建立 BGP 对等关系，使网络设计更加灵活。
RouterID	用于在 BGP 网络中唯一标识 BGP 路由器的 32 位数值，通常采用类似 IPv4 地址格式的点分十进制格式表示，一般用于建立 BGP 邻居关系、检测路由环路、选择最佳路径以及排查网络问题。

5. 点击创建。

> 注意： 使用 BGP 模式时，请允许每个 BGP 对等节点与 MetalLB speaker 节点之间的 TCP 端口 179 通信。如果配置了自定义对等节点端口，则改为允许配置的端口通信。

使用 **CLI** 配置带有 **L2Advertisement** 或 **BGPAdvertisement** 的外部 **IP** 池

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

BGP 模式

```
# 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-bgpadver
tisement.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查看 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 **MetalLB** 监控面板

1. 转到运维中心。

2. 在左侧导航栏中，点击监控 > 监控面板。
3. 搜索 **MetalLB**，然后打开监控面板。

内置 MetalLB 监控面板包括以下图表：

图表	描述
已公开的 **Service**	MetalLB 当前已通告的 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。
已通告的前缀	MetalLB 当前已通告的 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` 命名空间中的监控 Service 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 监控 Service：

```
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 '^metallb_'
```

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，则表示组件指标端点可用。

MetalLB 指标参考

内置 MetalLB 监控面板使用以下指标：

指标	源组件	描述	使用场景
`metallb_speaker_announced`	Speaker	指示 LoadBalancer Service 是否由某个 speaker 节点通告。	检查哪些通告 LoadBala Service。
`metallb_allocator_addresses_total`	Controller	地址池中的 IP 地址总数。	计算地址容量和可数量。
`metallb_allocator_addresses_in_use_total`	Controller	地址池中已分配的 IP 地址数量。	计算已使数量和地使用趋势
`metallb_bgp_session_up`	Speaker / BGP exporter	指示 BGP 会话是否已建立。	检查 BGI 运行状况
`metallb_bgp_announced_prefixes_total`	Speaker / BGP exporter	MetalLB 通告的 BGP 前缀数量。	检查路由状态。

MetalLB 故障排除

症状	可能原因	解决方案
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
Pod 反复崩溃重启	缺少 Speaker 或 Controller RBAC	检查 operator 权限
BGP 未建立	ASN 不匹配或对等端不可达	检查 `BGPPeer` spec 和网络路由
L2 无法正常工作	VLAN 配置错误或 ARP 过滤	使用 `arping` 验证广播可达性

如需了解更多信息，请参阅 MetalLB 故障排除↗

商标声明

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 不存在关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

配置 GatewayAPI Gateway

目录

概述
前提条件
Gateway 基础知识
什么是 Gateway
Gateway 暴露（Service Type）
LoadBalancer（推荐）
NodePort
ClusterIP
Listener 配置
允许 ListenerSet
端口和协议
AllowRouteNS
TLS 配置
EnvoyProxy（部署配置）
推荐资源规格
镜像仓库
创建 Gateway
通过 Web 控制台
Listener 配置
通过 YAML

包含多种 Listener 类型的完整示例
查看 Gateway 详情
Listener 和 Route 参考
主机名
主机名交集规则
支持的 Route 类型
下一步
商标声明

概述

在 Envoy Gateway operator 和 `EnvoyGatewayCtl` 就绪后，配置一个 `Gateway`。 `Gateway` 定义流量如何进入 Gateway，而配套的 `EnvoyProxy` 控制底层 Envoy data plane 的部署方式。

在推荐的工作流中，请在 Envoy Gateway Operator 之后、配置 GatewayAPI Route 之前配置 `Gateway`。

要公开由 Gateway 命名空间外部拥有的其他 listener，请参阅配置 Gateway API ListenerSet。

前提条件

继续操作前，请确保已完成以下操作：

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**
>
> 创建 `Gateway` 前，请先了解 Gateway 的主要概念。如果您已经熟悉这些概念，可以跳转到创建 Gateway部分。

Gateway 基础知识

什么是 Gateway

`Gateway` 是进入集群的流量入口。它定义如何接收外部请求并将其路由到后端服务。
`Gateway` 主要定义：

- **Listener**：定义 Gateway 监听的端口、协议和主机名
- **GatewayClass**：选择管理此 `Gateway` 的 Gateway controller
- **Infrastructure reference**：引用一个控制底层 Envoy data plane 部署方式的 `EnvoyProxy`

Gateway 暴露（Service Type）

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

在 YAML 中，此设置配置于配套的 `EnvoyProxy` 资源中的 `.spec.provider.kubernetes.envoyService.type`。

LoadBalancer（推荐）

其优势在于易于使用，并具备高可用负载均衡能力。 要使用 LoadBalancer，集群必须支持 LoadBalancer，可以通过 MetalLB 启用。

使用 MetalLB 时，可以通过 Service annotations 指定静态 VIP。在 Web Console 中，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
```

有关更多详细信息，请参阅使用 MetalLB 指定 VIP 的方法。

NodePort

其优势在于不需要任何外部依赖。

但是，使用 NodePort 存在以下缺点：

- 使用 NodePort 时，Kubernetes 分配的 NodePort 端口号与 Service 自身的端口不同。访问时必须使用 NodePort 端口号，而不是 Service 端口。
- 可以通过集群中任意节点的 IP 地址访问该 Service，这可能带来潜在的安全风险。

使用 **NodePort** 时如何获取正确的端口

在 Gateway 详情页面中，当 Service Type 为 NodePort 时，listener 列表会显示 NodePort 列，其中包含分配的端口号。您也可以使用以下命令：

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

输出结果即为 NodePort。

ClusterIP

如果不需要对外暴露，ClusterIP 非常方便。

Listener 配置

Listener 定义 Gateway 监听的端口和协议。对于 HTTP 或 HTTPS 协议，不同的主机名可以视为不同的 listener。

不能创建端口、协议或主机名冲突的 listener。

必须在 `Gateway` 中至少创建一个 listener。

允许 **ListenerSet**

Gateway 可以接受来自同一命名空间或其他命名空间中 ListenerSet 的额外 listener。配置 `spec.allowedListeners.namespaces` 以控制接受哪些 ListenerSet 命名空间：

值	行为
`None`	拒绝所有 ListenerSet。这是默认设置。
`Same`	接受 Gateway 命名空间中的 ListenerSet。
`Selector`	接受与 `selector` 匹配的命名空间中的 ListenerSet。
`All`	接受所有命名空间中的 ListenerSet。

对于 `Selector`，请在 `.spec.allowedListeners.namespaces.selector` 下配置标签选择器。有关更多信息，请参阅 配置 GatewayAPI ListenerSet。

端口和协议

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

AllowRouteNS

默认情况下，Route 只能附加到 `Same` 命名空间中的 Gateway。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 listener。
- `All`：允许任意命名空间中的 Route 附加到此 listener。
- `Selector`：允许与选择器匹配的命名空间中的 Route 附加到此 listener。

在 ACP 中，项目通过命名空间上的标签标识，例如 `cpaas.io/project: <project-name>`。如果希望 listener 仅供特定项目中的 Route 使用，请使用 `Selector`，并匹配目标命名空间上的项目标签。

Listener 的 `Allowed Routes Namespace` 设置及其协议共同决定了在 Route Web Console 中将 Route 发布到 listener 时可用的 listener。

有关更多信息，请参阅附加到在其他命名空间中创建的 Gateway。

TLS 配置

对于 HTTPS 和 TLS 协议，需要配置 TLS 设置。

TLS 模式：

TLS 模式	描述	是否需要证书
Terminate	Envoy 终止 TLS，并在转发到后端服务前解密流量	是，必须选择 TLS 证书
Passthrough	Envoy 不解密加密的 TLS 流量，直接将其传递到后端服务	否

NOTE

- HTTPS 协议仅支持 **Terminate** 模式
- TLS 协议同时支持 **Terminate** 和 **Passthrough** 模式
- **Passthrough** 模式的 TLS listener 支持 TLSRoute
- **Terminate** 模式的 TLS listener 支持 TCPRoute

证书要求：

- 默认情况下，只能使用在同一命名空间中创建的 Secret
- Secret 必须为 `kubernetes.io/tls` 类型，并包含 `tls.crt` 和 `tls.key` 键
- 对于跨命名空间 Secret，请参阅使用在其他命名空间中创建的 Secret

EnvoyProxy（部署配置）

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

当您在 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名称和命名空间相同的配套 `EnvoyProxy` 资源。

通过应用 YAML 创建 `Gateway` 时，您需要负责确保 `Gateway` `.spec.infrastructure.parametersRef` 以及所引用的 `EnvoyProxy` 资源保持一致。

这种一对一映射方式可以更好地隔离部署配置，并提供更细粒度的控制，例如：

- 副本数
- 资源限制和请求
- 节点选择器
- Service 类型和 annotations
- 镜像仓库

要限制某个命名空间中的 Envoy data plane 请求总量，请在该命名空间中创建一个 `EnvoyProxyResourceQuota`。配额会评估该配套的 `EnvoyProxy` 以及命名空间中的其他 `EnvoyProxy` 资源。

推荐资源规格

对于典型的 HTTP 和 HTTPS 南北向流量，单个 `envoy-proxy` Pod 可使用以下值作为起始配置。这些建议适用于 `.spec.provider.kubernetes.envoyDeployment.container.resources`。

每个 `envoy-proxy` **Pod 的峰值 QPS**	推荐 **CPU**	推荐内存
4,000	2 核	1Gi
8,000	4 核	2Gi
12,000	8 核	4Gi

NOTE

- 这些值是起始配置，并非硬性限制。请根据实际流量特征进行验证。
- TLS 终止、启用的策略、Route 复杂度、请求和响应大小以及长连接都可能增加资源使用量。
- 使用多个副本时，应根据每个副本实际处理的 QPS 进行扩展，而不只是依据 Gateway 的总 QPS。

镜像仓库

镜像仓库已预配置为集群的默认值。除非必要，否则不要修改。

有关其他部署配置方法，请参阅 deployment-mode ↗ 。

创建 Gateway

通过 Web 控制台

1. 导航至 `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，该对象为 `envoy-gateway-operator-cpaas-default`。

页面会显示以下配置选项：

字段	描述	YAML 路径
Name	Gateway 名称	gateway: `.metadata.name` envoyproxy: `.metadata.name`
GatewayClass	要使用的 GatewayClass	gateway: `.spec.gatewayClassName`
Service Type	Service Type	envoyproxy: `.spec.provider.kubernetes.envoyService.typ`
Service Annotation	Service 注解	envoyproxy: `.spec.provider.kubernetes.envoyService.ann`
Resource Limits	Deployment 资源限制	envoyproxy: `.spec.provider.kubernetes.envoyDeployment.container.r`
Replicas	Deployment 副本数	envoyproxy: `.spec.provider.kubernetes.envoyDeployment.`

字段	描述	YAML 路径
Node Labels	Deployment 节点选择器	envoyproxy: `.spec.provider.kubernetes.envoyDeployment.nodeSelecto`
Listener	Listener	gateway: `.spec.listeners`

WARNING

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

NOTE

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

Listener 配置

创建或编辑 Listener 时，可以配置以下内容：

字段	描述	YAML 路径
Name	Listener 的名称	`.spec.listeners[].name`
Port	Listener 监听的端口号	`.spec.listeners[].port`
Protocol	Listener 使用的协议。选项：HTTP、HTTPS、TCP、UDP、TLS	`.spec.listeners[].protocol`
Hostname	可选。Listener 的主机名	`.spec.listeners[].hostname`

字段	描述	YAML 路径
Allowed Routes Namespace	控制哪些命名空间可以将路由附加到此 Listener	`.spec.listeners[].allowedRoutes.namespaces.from`

HTTPS 协议配置

选择 **HTTPS** 作为协议时：

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

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

TLS 协议配置

选择 **TLS** 作为协议时：

配置字段

字段	描述	YAML 路径
TLS Mode	选择 TLS 模式。选项：Terminate、Passthrough。默认值：Terminate	`.spec.listeners[].tls.mode`
TLS Certificate	仅当 TLS Mode 为 Terminate 时必填。选择包含 TLS 证书的 secret。	`.spec.listeners[].tls.certificateRefs`

有关 TLS 模式的详细信息，请参阅 TLS 配置。

通过 YAML

如果不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`。

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

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

包含多种 **Listener** 类型的完整示例

如果需要更多 Listener 类型，请使用以下完整示例：

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

```
      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. 用于部署配置的 EnvoyProxy 资源
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` 影响基于主机的路由与 Listener 的匹配方式
5. `protocol` 确定哪些路由类型可以附加到 Listener
6. `allowedRoutes` 控制哪些命名空间可以附加路由
7. `tls` 为 HTTPS 和 TLS Listener 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 暴露方式
10. 除非环境确有需要，否则请保持 `imageRepository` 不变
11. `resources` 配置 Envoy 数据平面的资源限制和请求
12. 将 `provider.type` 保持为 `Kubernetes`

查看 **Gateway** 详情

在 Gateway 详情页面中，Listener 列表显示以下信息：

列	描述
Name	Listener 名称
Protocol	Listener 协议（HTTP、HTTPS、TCP、UDP、TLS）
Port	配置的端口号
NodePort	当 Service Type 为 NodePort 时显示。显示用于访问 Listener 的已分配 NodePort 端口号。

NOTE

当 Gateway Service Type 为 **NodePort** 时，Listener 列表会额外显示 **NodePort** 列。访问 Gateway 时，请使用 NodePort 值，而不是 Service 端口。更多详细信息，请参阅使用 NodePort 时如何获取正确的端口。

Listener 和 Route 参考

将 `Route` 资源附加到 `Gateway` 时，请遵循以下规则。

主机名

Listener 中的主机名是具有相同协议的 Listener 的唯一标识符。无法在 Gateway 中添加或更新冲突的 Listener。

主机名交集规则

请求到达时，系统会将其与 Listener 主机名和 Route 主机名的交集进行匹配。只有交集中的主机名才会用于路由流量。

Listener 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	接受任何传入的主机标头
无主机名	有主机名（例如 `api.example.com`）	所有 Route 主机名	仅匹配包含 `api.example.com` 的请求
有主机名（例如 `api.example.com`）	无主机名	所有 Listener 主机名	仅匹配包含 `api.example.com` 的请求
有主机名（例如 `api.example.com`）	有匹配的精确主机名	精确匹配的主机名	仅匹配包含 `api.example.com` 的请求

Listener 主机名	**Route** 主机名	交集结果	示例
有通配符（例如 `*.example.com`）	有匹配的主机名	匹配的特定主机名	匹配包含 `api.example.com` 或 `web.example.com` 的请求
有主机名（例如 `api.example.com`）	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` 可以匹配 `foo.example.com` 和 `bar.example.com`，但无法匹配 `example.com`。

WARNING

无交集意味着 Route 状态会变为异常，无法处理流量。

支持的 **Route** 类型

每个 Listener 根据其协议支持不同的 Route 类型：

Listener 协议	支持的 **Route** 类型
HTTP	HTTPRoute、GRPCRoute
HTTPS	HTTPRoute、GRPCRoute
TLS（`Passthrough` 模式）	TLSRoute
TLS（`Terminate` 模式）	TCPRoute
TCP	TCPRoute
UDP	UDPRoute

配置 Route 时，请确保其协议与要附加到的 Listener 的协议匹配。例如，不能将 `HTTPRoute` 附加到 `TCP` Listener。

下一步

`Gateway` 就绪后，请继续阅读配置 GatewayAPI Route。如果附加 Route 后需要进行高级流量控制，请继续阅读配置 GatewayAPI Policy。 如需在命名空间级别限制 Envoy 数据平面资源，请参阅 配置 EnvoyProxy 资源配额。

商标声明

Envoy® 是 Linux Foundation 的注册商标。

MetalLB™ 是 Linux Foundation 的商标。

Alauda 是独立供应商。本产品与 Linux Foundation 无关，亦未获得其认可或赞助。所有商标均为其各自所有者的财产，在此仅用于识别目的。

配置 GatewayAPI 策略

目录

概述
前提条件
策略附加基础
策略附加摘要
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
注意事项
官方文档
BackendTLSPolicy
通过 Web Console 配置
通过 YAML 配置
参考
功能

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

概述

本文档介绍如何在 `Gateway` 和 `Route` 资源准备就绪后配置策略资源。策略通过 `.spec.targetRefs` 使用策略附加模式，将额外的流量、安全和后端行为附加到受支持的资源上。

在推荐的工作流中，本文档位于配置 GatewayAPI 路由之后。

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

前提条件

继续操作前，请确保已完成以下操作：

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

策略附加基础

策略通过 `.spec.targetRefs` 附加到其他资源。

默认情况下，策略只能附加到同一命名空间中的资源。

对于 `Gateway` 目标，当策略类型支持时，可以使用 `sectionName` 指定目标监听器。对于 `Service` 目标，`sectionName` 指 Service 端口名称。

策略附加摘要

策略类型	用途	**Web Console** 支持	**Gateway**	**HTTPRoute**
SecurityPolicy	身份验证、授权、CORS 及其他安全功能	API Key 身份验证、CORS	✅（监听器名称 / `ALL`）	✅
BackendTLSPolicy	Envoy 与后端服务之间的 TLS 配置	支持	❌	❌
ClientTrafficPolicy	面向客户端的超时和连接行为	超时设置	✅（监听器名称 / `ALL`）	❌
BackendTrafficPolicy	后端超时和连接行为	超时设置	✅（监听器名称 / `ALL`）	✅

`sectionName` 用于指定 `Gateway` 上的特定监听器，或 `Service` 上的特定端口。如果省略或 设置为 `ALL`，策略将应用于所有监听器或端口。

有关 ListenerSet 策略行为和版本限制，请参阅配置 Gateway API ListenerSet。

在 Web Console 中创建 Policy

所有 Policy 类型都从同一入口创建：

1. 导航至 `Alauda Container Platform -> Networking -> Gateway -> Policies`
2. 在 `Policy Type` 下拉菜单中选择所需值
3. 点击 `Create Policy` 按钮

以下部分仅介绍各 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 类型	要创建的 Policy 类型	`.kind`
附加到	此 Policy 应用到的 Gateway API 资源。支持 Gateway、HTTPRoute 和 GRPCRoute。附加到 Gateway 时，可以选择性地指定 listener 名称，或选择所有 listener。	`.spec.targetRefs`

SecurityPolicy 特有字段：

字段	描述	YAML 路径
授权类型	要使用的身份验证/授权方法。支持多选：API Key Authentication、CORS Configuration	`.spec.apiKeyAuth`, `.spec.cors`

API Key Authentication

字段	描述	YAML 路径
Secrets	包含用于身份验证的 API Key 的 Kubernetes Secret	`.spec.apiKeyAuth.credentialRefs`
提取位置	指定 API Key 的提取位置（HTTP header 或查询参数）	`.spec.apiKeyAuth.extractFrom`

CORS Configuration

字段	描述	YAML 路径
Allow Origins	CORS 请求允许的来源列表	`.spec.cors.allowOrigins`
Allow Methods	允许的 HTTP 方法列表	`.spec.cors.allowMethods`
Allow Headers	CORS 请求中允许的 header 列表	`.spec.cors.allowHeaders`
Expose Headers	在响应中向客户端公开的 header 列表	`.spec.cors.exposeHeaders`
Max Age	CORS 预检响应的缓存时长	`.spec.cors.maxAge`
Allow Credentials	是否允许在 CORS 请求中携带凭据	`.spec.cors.allowCredentials`

通过 YAML 配置

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

参考

SecurityPolicy 用于为 Gateway 和 Route 配置身份验证、授权及其他安全相关功能。它提供了一种声明式方式，通过在请求到达后端应用程序之前验证传入请求来保护服务。

```
            Gateway            Route
               |                 |
               +-- SecurityPolicy-+
Client ---> Envoy Listener --------------------> Backend Service
```

功能

- 身份验证：使用各种方法（API Key、JWT、OIDC、Basic Auth）验证客户端身份
- 授权：根据已验证的凭据控制对资源的访问
- **CORS** 配置：管理跨域资源共享策略

工作原理

1. 创建包含所需身份验证/授权规则的 SecurityPolicy
2. 将其附加到特定的 Gateway、HTTPRoute 或 GRPCRoute
3. Envoy Gateway 根据 Policy 验证传入请求
4. 有效请求将转发到后端服务；无效请求将被拒绝，并返回相应的 HTTP 状态码

注意事项

1. Web Console 当前支持配置 **API Key Authentication** 和 **CORS**。对于其他身份验证方法和高级安全功能，需要使用 YAML 配置。
2. 每个 Route 只能关联一个 SecurityPolicy。
3. 如果 SecurityPolicy 引用了不包含任何值的 Secret，则对附加到该 Route 的所有请求都将返回 `401 Unauthorized` 并被拒绝。
4. 在 Web Console 中，`Extract From` 字段默认设置为 `header`，`Header Name` 字段默认设置为 `authorization`。
5. 在 Web Console 中，导航至 Route 的拓扑选项卡，即可查看附加到 Route 的 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 类型	要创建的 Policy 类型	`.kind`
附加到	此 Policy 应用到的 Service。必须指定 Service 名称和端口名称（`sectionName`）。	`.spec.targetRefs`

BackendTLSPolicy 特定字段：

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）	`.spec.validation.hostname`
Subject Alternative Names	可选。用于验证后端 HTTPS 响应。如果未指定，则默认使用 hostname 值。	`.spec.validation.subjectAltNames`
验证类型	验证后端 TLS 证书的方法。选项：`CACertificateRefs`（使用自定义 CA 证书）、`WellKnownCACertificates`（使用系统 CA 证书）	`.spec.validation`

CACertificateRefs 配置：

字段	描述	YAML 路径
CA 证书 Secret	包含 CA 证书的 Kubernetes Secret。该 Secret 必须包含 `ca.crt` 密钥，其中存储 PEM 编码的 TLS 证书。	`.spec.validation.cACertificateRefs`

NOTE

创建或选择 CA 证书 Secret 时：

- Secret 类型必须适用于 CA 证书
- 密钥必须为 `ca.crt`
- 可以导入证书文件，该文件必须以 `-----BEGIN CERTIFICATE-----` 开头并以 `-----END CERTIFICATE-----` 结尾
- 导入无效证书格式时，将显示错误消息“must contain PEM-encoded TLS certificates”
- 选择不包含 `ca.crt` 密钥的现有 Secret 时，将显示错误消息“must have ca.crt key”

通过 YAML 配置

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

参考

BackendTLSPolicy 控制 Envoy Gateway 与后端服务之间的 TLS 配置。它允许你配置：

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

- **SNI**（**Server Name Indication**）：与后端建立 TLS 连接时使用的主机名
- 证书验证：验证后端服务器证书的方式
- **CA** 证书：用于验证后端证书的自定义 CA 证书

功能

- 配置连接到后端服务时的 TLS 设置
- 支持自定义 CA 证书或系统已知 CA 证书
- 配置 SNI 以完成正确的 TLS 握手

注意事项

1. `targetRefs` 中的 `sectionName` 对应 Service 的端口名称。
2. 使用 `WellKnownCACertificates` 时，将使用系统的默认 CA 证书进行验证。
3. 主机名为必填项，Envoy 连接后端时将其用作 SNI 值。

官方文档

- BackendTLSPolicy 规范↗

ClientTrafficPolicy

通过 Web 控制台配置

通用字段：

字段	描述	YAML 路径
策略类型	要创建的策略类型	`.kind`
附加到	此策略应用到的 Gateway。您可以选择指定监听器名称，或选择所有监听器。	`.spec.targetRefs`

超时配置（可选）：

字段	描述	YAML 路径
TCP 空闲超时	TCP 连接的空闲超时时间。空闲时间是指上游或下游连接均未发送或	`.spec.settings.timeout.tcp.idleTimeout`

字段	描述	YAML 路径
	接收任何字节的时间段。默认值：1 小时。	
HTTP 请求接收超时	Envoy 等待完整接收请求的时长。此计时器在请求开始时启动，并在请求的最后一个字节发送到上游或响应开始时停止。默认值：1 小时。	`.spec.settings.timeout.http.requestReceivedTimeout`
HTTP 空闲超时	HTTP 连接的空闲超时时间。空闲时间是指连接中没有活动请求的时间段。默认值：无限制。	`.spec.settings.timeout.http.idleTimeout`
HTTP 流空闲超时	流空闲超时定义了流在没有任何上游或下游活动的情况下可以存在的时长。默认值：5 分钟。	`.spec.settings.timeout.http.streamIdleTimeout`

通过 YAML 配置

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

参考

ClientTrafficPolicy 控制客户端到 Envoy Gateway 的连接行为。它可以对以下内容进行细粒度控制：

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

- **TCP** 设置：连接级别的超时和 keepalive 设置
- **HTTP** 设置：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 流空闲超时

注意事项

1. 超时值以时长字符串表示（例如 "30s"、"5m"、"1h"）。

官方文档

- ClientTrafficPolicy 规范↗

BackendTrafficPolicy

通过 Web Console 配置

常用字段：

字段	描述	YAML 路径
Policy Type	要创建的策略类型	`.kind`
Attach To	此策略适用的 Gateway API 资源。支持 Gateway、HTTPRoute、GRPCRoute、TCPRoute、UDPRoute 和 TLSRoute。附加到 Gateway 时，可以选择指定 listener 名称，或选择 `ALL` listeners。在 YAML 中，通过 `.spec.targetRefs` 中的 `sectionName` 进行配置。	`.spec.targetRefs`

超时配置（选项）：

字段	描述	YAML 路径
TCP Connection Timeout	建立网络连接的超时时间，包括 TCP 和 TLS 握手。默认值：10 秒。	`.spec.settings.timeout.tcp.connectionTimeout`

字段	描述	YAML 路径
HTTP Connection Idle Timeout	HTTP 连接的空闲超时时间。空闲时间定义为连接中没有活动请求的时间段。默认值：1 小时。	`.spec.settings.timeout.http.connectionIdleTimeout`
HTTP Max Connection Duration	HTTP 连接的最长持续时间。默认值：无限制。	`.spec.settings.timeout.http.maxConnectionDuration`
HTTP Request Timeout	从上游接收完整响应之前的最长等待时间。默认值：15 秒。支持设置为无限制。	`.spec.settings.timeout.http.requestTimeout`

通过 YAML 配置

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

参考

BackendTrafficPolicy 控制从 Envoy Gateway 到后端服务的连接行为。它提供以下方面的细粒度控制：

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

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

功能

- 配置 TCP 连接建立超时
- 控制 HTTP 连接空闲超时
- 设置 HTTP 连接最长持续时间

- 配置 HTTP 请求超时

注意事项

1. 超时值以时长字符串表示（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

策略附加后，请继续阅读 Envoy Gateway 任务，了解更多操作示例和高级配置任务。

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 无关联，也未获得其认可或赞助。所有商标均为其各自所有者的财产，在此仅用于标识目的。

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
发布到侦听器
后端
Hostnames
Rules
HTTPRoute 参考
GRPCRoute 匹配和过滤器参考
TLSRoute 参考
视图
拓扑
下一步

相关任务

商标声明

概述

本文介绍如何在 `Gateway` 就绪后配置 `Route` 资源。 `Route` 会附加到一个或多个网关侦听器，并定义如何将匹配的流量转发到后端服务。

在推荐的工作流中，本文位于配置 GatewayAPI Gateway 之后，以及配置 GatewayAPI Policy 之前。

除了创建和更新操作外，本文还介绍 ACP Web Console 提供的其他 Route 查看功能。

前提条件

请确保在继续操作前已完成以下事项：

1. 阅读配置 GatewayAPI Gateway 以了解侦听器、附加规则和 `EnvoyProxy`
2. 创建一个 `Gateway` ，以供 `Route` 附加到该对象

> **NOTE**
>
> 本文首先分别介绍每种 Route 类型，然后提供 YAML 示例，最后在共享参考部分说明通用的 Route 概念。

配置

`Route` 会附加到 `Gateway` 上的一个或多个侦听器。可选择的侦听器取决于 Route 类型、侦听器协议以及侦听器允许的 Route 命名空间设置。

Route 也可以附加到外部 `ListenerSet` 侦听器。请参阅配置 Gateway API ListenerSet。

通过 Web Console 配置

1. 导航到 `Alauda Container Platform -> Networking -> Gateway -> Routes`
2. 点击 `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
发布到侦听器	发布到侦听器	`.spec.parentRefs`
Hostnames	Hostnames	`.spec.hostnames`
Matches	Matches	`.spec.rules[].matches`
Filters	Filters	`.spec.rules[].filters`
后端实例	后端	`.spec.rules[].backendRefs`
Options	Options	`.spec.rules[].filters`, `.spec.rules[].timeouts`, `.spec.rules[].retry`, `.spec.rules[].sessionPersistence`

Options 配置

Options 字段用于配置高级流量管理设置：

选项	描述	YAML 路径
会话亲和性	会话持久性	`.spec.rules[].sessionPersistence`
超时	超时设置	`.spec.rules[].timeouts`
重试	重试策略	`.spec.rules[].retry`

创建 TCP/UDP Route

字段	描述	YAML 路径
发布到侦听器	发布到侦听器	`.spec.parentRefs`
后端实例	后端	`.spec.rules[].backendRefs`

创建 GRPCRoute

字段	描述	YAML 路径
发布到侦听器	发布到侦听器	`.spec.parentRefs`
Hostnames	Hostnames	`.spec.hostnames`
Matches	gRPC Matches	`.spec.rules[].matches`
Filters	gRPC Filters	`.spec.rules[].filters`
后端实例	后端	`.spec.rules[].backendRefs`

创建 TLSRoute

字段	描述	YAML 路径
发布到侦听器	发布到侦听器	`.spec.parentRefs`
Hostnames	Hostnames（可选）	`.spec.hostnames`
后端实例	后端	`.spec.rules[].backendRefs`

通过 YAML 配置

以下最小示例创建一个 `HTTPRoute`，将其附加到 `demo` `Gateway` 的 `https` 侦听器，并将匹配的流量转发到后端 `Service`。

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

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

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

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

```
    - 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 发布到侦听器

3. Rules
4. 后端
5. Filters
6. Matches
7. Options
8. 超时
9. 重试
10. 会话持久性

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 规范↗
- TCPRoute 规范↗
- UDPRoute 规范↗
- GRPCRoute 规范↗
- TLSRoute 规范↗

发布到侦听器

在 **Web Console** 中

在 Web Console 中，可以选择多个侦听器来发布 Route。可用的侦听器候选项会根据以下条件进行筛选：

- 用户权限：您必须有权访问网关的命名空间（项目必须包含此命名空间）。
- **Route** 命名空间允许列表：网关侦听器允许的 Route 命名空间必须包含 Route 的命名空间。
- **Route** 类型匹配：Route 的类型（HTTPRoute、GRPCRoute 等）必须与侦听器允许的 Route 类型匹配。

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网关。

附加到 **ListenerSet**

Web Console 还会列出已接受的 ListenerSet 侦听器。发布 Route 时，选择 ListenerSet 和侦听器名称。ListenerSet 命名空间、父 Gateway 许可、侦听器协议以及侦听器 `allowedRoutes` 必须允许此次附加。

在 **YAML** 中

- `sectionName` 是侦听器名称。
- Route 只能附加到支持其特定类型的侦听器。
- 默认情况下，Route 只能附加到 `Gateway` 与其位于同一命名空间的侦听器。

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网关。

要在 YAML 中附加到 ListenerSet，请将 `parentRefs[].kind` 设置为 `ListenerSet`，将 `name` 和 `namespace` 设置为 ListenerSet，并将 `sectionName` 设置为侦听器名称：

```yaml
parentRefs:
  - group: gateway.networking.k8s.io
    kind: ListenerSet
    namespace: app-demo
    name: app-listeners
    sectionName: app-http
```

有关更多信息，请参阅配置 GatewayAPI ListenerSet。

后端

定义应将匹配请求转发到的目标服务。

每个服务都可以包含 `weight` 字段，用于指定应路由到该服务的流量比例。

Hostnames

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

`hostnames` 是字符串数组，遵循Hostname 交集规则。

Rules

每个 Route 可以包含多个规则。每条规则由以下组件组成：

Matches

定义请求必须满足的条件，满足这些条件后请求才会由此规则进行路由。

一条规则可以包含多个 Match：

- 每个 Match 由多个条件组成（例如路径、标头、查询参数和方法）
- 单个 Match 内部的条件使用 **AND** 逻辑（必须全部满足）
- 各个 Match 之间使用 **OR** 逻辑（满足任意一个 Match 即可）

示例：如果 Match-1 要求 `path=/api` AND `header=v1`，而 Match-2 要求 `query=test`，则当请求匹配 `(path=/api AND header=v1)` OR `(query=test)` 中的任意一个时，该请求就会被路由。

不同 Route 类型使用通用的 Match 结构，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。

不同 Route 类型使用通用的 Filter 概念，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 条件类型

对象	方法	值类型	描述	值要求
路径				

对象	方法	值类型	描述	值要求
	`Exact`	路径（字符串）	精确匹配 URL 路径，且区分大小写。这意味着对 /abc 的精确路径匹配只会匹配指向 /abc 的请求，不会匹配 /abc/、/Abc 或 /abcd。	必须以 `/` 开头能包含连续的 `//`。
	`PathPrefix`	路径（字符串）	根据以 / 分隔的 URL 路径前缀进行匹配。匹配区分大小写，并且逐个路径元素进行。例如，路径 /abc、/abc/ 和 /abc/def 都会匹配前缀 /abc，但路径 /abcd 不会匹配。	必须以 `/` 开头能包含连续的 `//`。
	`RegularExpression`	路径（字符串）	正则表达式引擎：RE2。	例如，`/api/v1/`
标头				
	`Exact`	名称（标头键）+ 值	精确匹配标头值。	
	`RegularExpression`	名称（标头键）+ 值	正则表达式引擎：RE2。	
查询参数				
	`Exact`	名称（参数键）+ 值	精确匹配查询参数值。	参数值：1-1024 字符

对象	方法	值类型	描述	值要求
	`RegularExpression`	名称（参数键）+ 值	正则表达式引擎：RE2。	
方法	-	方法名称	HTTP 方法匹配。	`GET`、`HEAD`、`POST`、`PUT`、`DELETE`、`CONNECT`、`OPTIONS`、`TRACE`、`PAT`

匹配条件参考

条件类型	官方文档
路径	HTTPPathMatch ↗
标头	HTTPHeaderMatch ↗
查询参数	HTTPQueryParamMatch ↗
方法	HTTPMethod ↗

过滤器类型

类型	方法	值类型	描述
RequestHeaderModifier	`Set`	名称（字符串）+ 值（字符串）	使用给定的名称和值覆盖请求标头
	`Add`	名称（字符串）+ 值（字符串）	将标头添加到请求，并追加到现有值
	`Remove`	[]string	从请求中移除指定的标头（不区分大小写）

类型	方法	值类型	描述
ResponseHeaderModifier	`Set`	名称（字符串）+ 值（字符串）	使用给定的名称和值覆盖响应标头
	`Add`	名称（字符串）+ 值（字符串）	将标头添加到响应，并追加到现有值
	`Remove`	[]string	从响应中移除指定的标头（不区分大小写）
RequestRedirect	`Scheme`	string	Location 标头的协议（http/https）
	`Hostname`	PreciseHostname	Location 标头的主机名
	`ReplaceFullPath`	string	替换整个请求路径
	`ReplacePrefixMatch`	string	替换匹配的路径前缀
	`Port`	PortNumber	Location 标头的端口
	`StatusCode`	int	HTTP 重定向状态码
URLRewrite	`Hostname`	PreciseHostname	要在请求中重写的主机名

类型	方法	值类型	描述
	`ReplaceFullPath`	string	替换整个请求路径
	`ReplacePrefixMatch`	string	替换匹配的路径前缀
CORS	`AllowOrigins`	[]string	CORS 请求允许的源列表
	`AllowMethods`	[]HTTPMethod	允许的 HTTP 方法列表
	`AllowHeaders`	[]string	CORS 请求中允许的标头列表
	`ExposeHeaders`	[]string	响应中向客户端公开的标头列表
	`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` 不能在同一规则中同时使用
- `ReplacePrefixMatch` 仅与 `PathPrefix` HTTPRouteMatch 兼容
- 根据 RFC 7230，标头名称不区分大小写
- 同一标头的多个值必须使用 RFC 7230 逗号分隔格式

过滤器参考

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

选项

选项部分为 `HTTPRoute` 提供高级流量管理功能，包括超时、重试和会话持久性设置。

超时

字段	描述	**YAML** 路径
请求超时	网关在从客户端接收完整请求后完成 HTTP 响应的最大时长。选项：默认（使用默认超时，通常为 15 秒）、无限制（设置为“0s”以移除超时）、自定义。	`.spec.rules[].timeouts.request`
后端请求超时	单次网关到后端调用的最大时长，从网关开始发送请求到接收完整后端响应为止。选项：默认（使用特定于实现的默认值）、无限制（设置为“0s”）、自定义。	`.spec.rules[].timeouts.backendRequest`

NOTE

- 请求超时在接收完整客户端请求后开始计时，并涵盖整个事务；如果发生重试，该事务可能包括多次后端调用。
- 指定后端请求超时时，该值必须 ≤ 请求超时。
- 选择“默认”时，该字段设置为 nil（使用实现默认值）。
- 选择“无限制”时，该字段设置为“0s”（最大可能值）。

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

重试

字段	描述	YAML 路径
状态码	触发重试的 HTTP 状态码（例如 503、502）。取值范围：400-599。	`.spec.rules[].retry.codes`
次数	重试次数。	`.spec.rules[].retry.attempts`
退避	重试前的等待时间（例如“100ms”、“1s”）。	`.spec.rules[].retry.backoff`

NOTE

- 默认情况下，重试已禁用。如果未配置重试字段或该字段为空，网关将不会重试任何失败的请求。
- 必须明确配置重试次数和重试条件，才能启用重试功能。
- 在 Web 控制台中配置重试时，如果移除所有重试配置项，该字段将设置为 nil。

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

会话持久性

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后端。

字段	描述	YAML 路径
类型	会话持久性类型。选项：Cookie、Header。	`.spec.rules[].sessionPersistence.type`
会话名称	用于会话跟踪的 cookie 或 header 的名称。	`.spec.rules[].sessionPersistence.sessionName`

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

GRPCRoute 匹配和过滤器参考

以下匹配条件和过滤器类型由 `GRPCRoute` 使用。

GRPCRoute 匹配

`GRPCRoute` 支持以下匹配类型：

对象	方法	值类型	描述
Method	-	type（服务/方法）+ value	匹配 gRPC 方法。类型可以是 `service`（匹配服务名称）或 `method`（匹配方法名称）。
Headers	`Exact`	name（header 键）+ value	精确匹配 header 值。

对象	方法	值类型	描述
	`RegularExpression`	name（header 键）+ value	正则表达式引擎：RE2。

GRPCRoute 过滤器

`GRPCRoute` 仅支持 `RequestHeaderModifier` 过滤器：

类型	方法	值类型	描述	值要求
RequestHeaderModifier	`Set`	name（字符串）+ value（字符串）	使用给定名称和值覆盖请求 header	最多 16 项，值：1-4096 字符
	`Add`	name（字符串）+ value（字符串）	将 header 添加到请求，追加到现有值	最多 16 项，值：1-4096 字符
	`Remove`	[]string	从请求中移除指定的 header（不区分大小写）	最多 16 项

NOTE

`GRPCRoute` 不支持超时、重试或会话持久性等选项。

TLSRoute 参考

以下行为特定于 `TLSRoute`。

NOTE

- `TLSRoute` 主机名是可选的。如果监听器具有主机名，但 `TLSRoute` 没有，则 `TLSRoute` 会自动继承监听器的主机名。

- `TLSRoute` 只能附加到 `Passthrough` 模式下使用 `TLS` 协议的监听器。

视图

拓扑

以下功能是 ACP Web Console 提供的其他查看功能。

拓扑选项卡以可视化方式呈现路由及其关联资源。它会显示附加到路由的所有策略，以及其依赖资源，例如 `SecurityPolicy` 引用的 secret。

此功能目前仅适用于 `HTTPRoute`。

下一步

将路由附加到监听器后，如果需要高级流量或安全策略，请继续配置 GatewayAPI 策略。如需了解更多操作示例，请参阅 Envoy Gateway 任务。

相关任务

- 如何附加到在其他命名空间中创建的监听器

商标声明

Envoy® 是 Linux Foundation 的注册商标。

Alauda 是独立供应商。本产品与 Linux Foundation 无关，也未获得其认可或赞助。所有商标均为其各自所有者的财产，此处仅用于标识目的。

配置 GatewayAPI ListenerSet

目录

概述
先决条件
配置
通过 Web Console 配置
通过 YAML 配置
附加 Route
Policy 行为
查看 ListenerSet 详情
下一步
相关任务
商标声明

概述

配置一个 `ListenerSet`，为现有的 `Gateway` 提供其他监听器。ListenerSet 归其所在的命名空间所有，并通过 `spec.parentRef` 附加到 Gateway。

ACP 同时支持直接在 `Gateway.spec.listeners` 中定义的监听器，以及在 ListenerSet 中定义的监听器。Gateway 必须显式允许 ListenerSet；默认情况下不接受任何 ListenerSet。

先决条件

继续操作前，请确保已完成以下事项：

1. 阅读配置 GatewayAPI Gateway，了解 Gateway 监听器和监听器附加规则。
2. 创建 Gateway，并配置其 ListenerSet 命名空间允许范围。

配置

通过 Web Console 配置

1. 导航至 `Alauda Container Platform -> Networking -> Gateway -> ListenerSets`。
2. 单击 `Create ListenerSet` 按钮。
3. 选择 ListenerSet 命名空间及其父 Gateway。
4. 添加一个或多个监听器，并配置其允许的 Route 命名空间。

ListenerSet 表单包含以下字段：

字段	描述	YAML 路径
名称	ListenerSet 的名称	`.metadata.name`
命名空间	拥有 ListenerSet 的命名空间	`.metadata.namespace`
父 Gateway	接收监听器的 Gateway	`.spec.parentRef`
监听器	端口、协议、主机名、TLS 以及允许的 Route 命名空间	`.spec.listeners[]`

通过 YAML 配置

首先配置父 Gateway，以允许来自所需命名空间的 ListenerSet。请参阅在 Gateway 上允许 ListenerSet。

创建一个 ListenerSet，并使用 `spec.parentRef` 指向该 Gateway：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: ListenerSet
metadata:
  name: app-listeners
  namespace: app-demo
spec:
  parentRef:
    name: edge
    namespace: gateway-system
  listeners:
    - name: app-http
      protocol: HTTP
      port: 8080
      allowedRoutes:
        namespaces:
          from: Same
```

监听器名称只需在 ListenerSet 内唯一。只有当父 Gateway 的 `allowedListeners` 规则包含该命名空间时，其他命名空间中的 ListenerSet 才会被接受。

附加 **Route**

Route 可以附加到 Gateway 监听器或 ListenerSet 监听器。将父引用类型设置为 `ListenerSet`，并使用 `sectionName` 指定所选监听器：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: app-route
  namespace: app-demo
spec:
  parentRefs:
    - group: gateway.networking.k8s.io
      kind: ListenerSet
      namespace: app-demo
      name: app-listeners
      sectionName: app-http
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /app
      backendRefs:
        - name: app-service
          port: 80
```

ListenerSet 监听器的 `allowedRoutes` 控制哪些 Route 命名空间可以附加。所选监听器的协议和主机名也必须与 Route 兼容。有关完整的 Route 工作流，请参阅将 Route 附加到 ListenerSet。

Policy 行为

不包含 `sectionName` 的 Gateway 级 Policy 会应用于父 Gateway 的合并监听器配置，包括已附加的 ListenerSet 监听器。

在 ACP Envoy Gateway `v1.8.0` 行中，以下 Policy 支持以 Gateway 为目标并可选使用 `sectionName`，但不支持直接以 `ListenerSet` 为目标：

- `SecurityPolicy`
- `ClientTrafficPolicy`
- `BackendTrafficPolicy`
- `EnvoyExtensionPolicy`

`EnvoyPatchPolicy` 以 Gateway 或 GatewayClass 为目标，并不是 ListenerSet 专用的 Policy 入口点。如果行为应覆盖 Gateway 及其已附加的 ListenerSet 监听器，请使用 Gateway 级 Policy。如果行为属于特定 Route，请使用 Route 级 Policy。

查看 ListenerSet 详情

在 ListenerSet 详情页面中，查看父 Gateway、监听器配置和状态条件。排查附加问题时，请检查 Gateway 的 `status.attachedListenerSets`、ListenerSet 的 `Accepted` 和 `Programmed` 条件，以及每个监听器的状态。

下一步

ListenerSet 就绪后，继续执行配置 GatewayAPI Route 以附加 Route。如果需要流量或安全 Policy，请继续执行配置 GatewayAPI Policy。

相关任务

- 在 Gateway 上允许 ListenerSet
- 将 Route 附加到 ListenerSet

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

Alauda 是一家独立供应商。本产品与 The Linux Foundation 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

配置 EnvoyProxy 资源配额

目录

概述

配额模型

创建配额

配置 EnvoyProxy 请求

准入行为

监控 namespace 使用量

范围和限制

下一步

商标声明

概述

使用 `EnvoyProxyResourceQuota` 限制一个命名空间中由 `EnvoyProxy` 资源声明的 CPU 和内存请求总量。该配额适用于 `EnvoyProxy` 所声明的 Envoy 数据平面配置；它不会替代 Kubernetes `ResourceQuota`，也不会直接测量生成的 Envoy Deployment。

此功能在 ACP 4.5 及更高版本中可用。该功能由 Envoy Gateway Operator 提供，并使用 `envoy-gateway.alauda.io/v1alpha1` API 组。

NOTE

`Gateway` 和 `EnvoyProxy` 是独立的资源。建议为每个 `Gateway` 使用一个专用的 `EnvoyProxy`，并通过 `.spec.infrastructure.parametersRef` 由 `Gateway` 引用该资源。使用 ACP Web Console 时，控制台可以将配套的 `EnvoyProxy` 与 `Gateway` 一起创建；应用 YAML 时，请显式创建并引用这两个资源。 配额适用于 `Gateway` 命名空间中的 `EnvoyProxy` 资源。有关更多信息，请参阅 配置 GatewayAPI Gateway。

配额模型

`EnvoyProxyResourceQuota` 是命名空间级资源。配额仅适用于同一命名空间中的 `EnvoyProxy` 资源。首个版本支持以下硬限制：

资源	含义
`requests.cpu`	Envoy 数据平面从节点的 CPU 请求总量
`requests.memory`	Envoy 数据平面从节点的内存请求总量

配额资源同时报告配置的硬限制和当前命名空间的使用量：

- `status.hard` 与 `spec.hard` 保持一致。
- `status.used` 是命名空间中 `EnvoyProxy` 资源的使用量总和。

如果一个命名空间包含多个配额对象，则每个对象的 `status.used` 都会报告相同的命名空间总量。不要将多个配额对象的 `status.used` 值相加。其命名空间中的每个 `EnvoyProxy` 请求都必须满足所有配额对象的要求。

创建配额

通过应用 namespace 级的 `EnvoyProxyResourceQuota` 资源来创建配额。以下示例将该 namespace 中 Envoy data plane 从节点的 CPU 请求限制为 4 个核心，将内存请求限制为 8 GiB：

```yaml
apiVersion: envoy-gateway.alauda.io/v1alpha1
kind: EnvoyProxyResourceQuota
metadata:
  name: gateway-quota
  namespace: tenant-a
spec:
  hard:
    requests.cpu: "4"
    requests.memory: 8Gi
```

```bash
kubectl apply -f gateway-quota.yaml
```

当必须同时检查不同策略时，可以在一个 namespace 中创建多个配额。只有当其产生的使用量在每个配额的硬限制以内时，`EnvoyProxy` 才会被准入。

配置 EnvoyProxy 请求

配额会统计 `EnvoyProxy` 资源中的以下字段：

EnvoyProxy 字段	含义
`.spec.provider.kubernetes.envoyDeployment.replicas`	Envoy plane 量
`.spec.provider.kubernetes.envoyDeployment.container.resources.requests.cpu`	每个从 CPU 请
`.spec.provider.kubernetes.envoyDeployment.container.resources.requests.memory`	每个从 内存请

例如，以下 `EnvoyProxy` 会从配额中消耗 `750m` CPU 和 `768Mi` 内存：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: gateway-proxy
  namespace: tenant-a
spec:
  provider:
    type: Kubernetes
    kubernetes:
      envoyDeployment:
        replicas: 3
        container:
          resources:
            requests:
              cpu: 250m
              memory: 256Mi
```

使用量计算如下：

```
CPU usage = replicas × envoyDeployment.container.resources.requests.cpu
Memory usage = replicas × envoyDeployment.container.resources.requests.me
mory
```

如果省略某个字段，配额将使用以下默认值：

字段	默认值
`replicas`	`1`
`requests.cpu`	`100m`
`requests.memory`	`512Mi`

配额不会统计资源限制，也不会检查由 Envoy Gateway 生成的最终 Deployment。

准入行为

当 namespace 中存在一个或多个配额时，Envoy Gateway Operator 会在 `EnvoyProxy` `CREATE` 和 `UPDATE` 请求存储之前对其进行验证：

1. 根据该 namespace 中的 `EnvoyProxy` 资源计算其现有使用量。
2. 对于更新操作，使用新对象的使用量替换旧对象的使用量。
3. 根据该 namespace 中的每个配额检查产生的 CPU 和内存使用量。
4. 如果任何硬限制将被超出，则拒绝该请求。错误信息会标识配额、资源、当前使用量、请求使用量和硬限制。

删除 `EnvoyProxy` 后，控制器重新调和配额时会减少报告的 `status.used`。不包含 `EnvoyProxyResourceQuota` 对象的 namespace 不会施加此 EnvoyProxy 特定的准入限制。

> **WARNING**
>
> 准入是一种尽力而为的防护措施。并发的 `CREATE` 或 `UPDATE` 请求会根据请求时观察到的使用量进行评估，并不会以原子方式预留资源。在需要严格隔离时，请预留足够的余量。

监控 namespace 使用量

配额状态是 namespace 级总量的事实来源。列出配额以查看硬限制和已使用值：

```
kubectl get envoyproxyresourcequota -n tenant-a
```

输出包含以下列：

列	来源	含义
`CPU Hard`	`.status.hard.requests.cpu`	此配额的 CPU 硬限制
`CPU Used`	`.status.used.requests.cpu`	namespace 中 EnvoyProxy 资源计入的 CPU 请求总量
`Memory Hard`	`.status.hard.requests.memory`	此配额的内存硬限制

列	来源	含义
`Memory Used`	`.status.used.requests.memory`	namespace 中 EnvoyProxy 资源计入的内存请求总量

如需查看完整值，请将某个配额资源检查为 YAML：

```
kubectl get envoyproxyresourcequota gateway-quota -n tenant-a -o yaml
```

配额或 `EnvoyProxy` 发生变化后，控制器会异步更新 `status.hard` 和 `status.used`。当一个 namespace 包含多个配额时，仅显示一次 namespace 使用量，并分别显示每个配额的硬限制；不要对重复的 `CPU Used` 或 `Memory Used` 值求和。

范围和限制

- 此功能从 ACP 4.5 起可用。
- 仅支持 `requests.cpu` 和 `requests.memory`。
- 配额具有 namespace 作用域，仅适用于该 namespace 中的 `EnvoyProxy` 资源。
- 配额独立于 Kubernetes `ResourceQuota`；不会读取或更新 `ResourceQuota.status.used`。
- 不会限制资源 `limits`。
- 不会跟踪最终的 Envoy Deployment，也不会强制执行 Deployment 准入策略。
- 不支持跨 namespace 配额、特定于 Gateway 或 GatewayClass 的组，也不支持标签选择器。

下一步

创建配额后，配置相应的 Gateway 及其专用的 `EnvoyProxy`。若要了解 Envoy Gateway Operator 如何部署这些资源，请参阅 Envoy Gateway Operator。

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

配置 ALB

WARNING

自 ACP 4.4 起，ALB 已弃用。无法创建新的 ALB 实例，且无法再通过 Web 控制台使用 ALB。在从早期版本升级的集群上，现有 ALB 实例会继续运行，并且仍可通过 API 管理，因此可以将其流量迁移到 Envoy Gateway 或 Ingress NGINX。迁移完成后，请移除 ALB 实例。

目录

- ALB
  - 前提条件
  - 配置 ALB
    - 资源相关配置
    - 网络配置
    - 项目配置
    - 调整配置
  - ALB 操作
    - 创建
    - 更新
    - 删除
- 前端
  - 前置条件
  - 配置 Frontend

Frontend 操作
创建
后续操作
相关操作
Rule
前提条件
使用 dslx 和优先级匹配请求
dslx
优先级
Action
Backend
后端协议
Service Group 和会话亲和性策略
Rule 操作
使用 web console
使用 CLI
Https
Ingress 中的证书注解
Rule 中的证书
TLS 模式
Ingress
Ingress 同步
日志与监控
查看日志
监控指标
商标声明

ALB

ALB 是表示负载均衡器的自定义资源。在从早期版本升级的集群上，alb-operator 会监视 ALB 资源的创建、更新和删除操作，并据此创建相应的部署和服务。

每个 ALB 都有一个对应的 Deployment，该 Deployment 会监视所有关联到此 ALB 的 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性需要 VIP。请参阅配置 VIP。

配置 **ALB**

ALB 配置分为三个部分。

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

资源相关配置

资源相关字段描述 alb 的部署配置。

字段	类型	描述
`.spec.config.nodeSelector`	map[string]string	alb 的节点选择器

字段	类型	描述
`.spec.config.replicas`	int,optional default 3	alb 的副本数
`.spec.config.resources.limits`	k8s container-resource,optional	alb 的 nginx 容器限制
`.spec.config.resources.requests`	k8s container-resource,optional	alb 的 nginx 容器请求
`.spec.config.resources.alb.limits`	k8s container-resource,optional	alb 容器的限制
`.spec.config.resources.alb.requests`	k8s container-resource,optional	alb 容器的请求
`.spec.config.antiAffinityKey`	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，您可以通过节点 IP 访问 ALB。

字段	类型	描述
`.spec.config.networkMode`	string: `host` or `container`, optional, default `host`	在 `container` 模式下，operator 会创建一个 LoadBalancer Service，并将其地址用作 ALB 地址。
`.spec.address`	string,required	您可以手动指定 alb 的地址
`.spec.config.vip.enableLbSvc`	bool, optional	在 `container` 模式下自动为 true。
`.spec.config.vip.lbSvcAnnotations`	map[string]string, optional	LoadBalancer Service 的额外注解。

项目配置

字段	类型
`.spec.config.projects`	[]string,required
`.spec.config.portProjects`	string,optional
`.spec.config.enablePortProject`	bool,optional

将 ALB 添加到项目意味着：

1. 在 Web UI 中，只有指定项目中的用户才能找到并配置此 ALB。
2. 此 ALB 将处理属于此项目的 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 下找到或配置。

如果启用端口项目并为项目分配端口范围，则意味着：

1. 您无法创建不属于分配给该项目的端口范围的端口。

调整配置

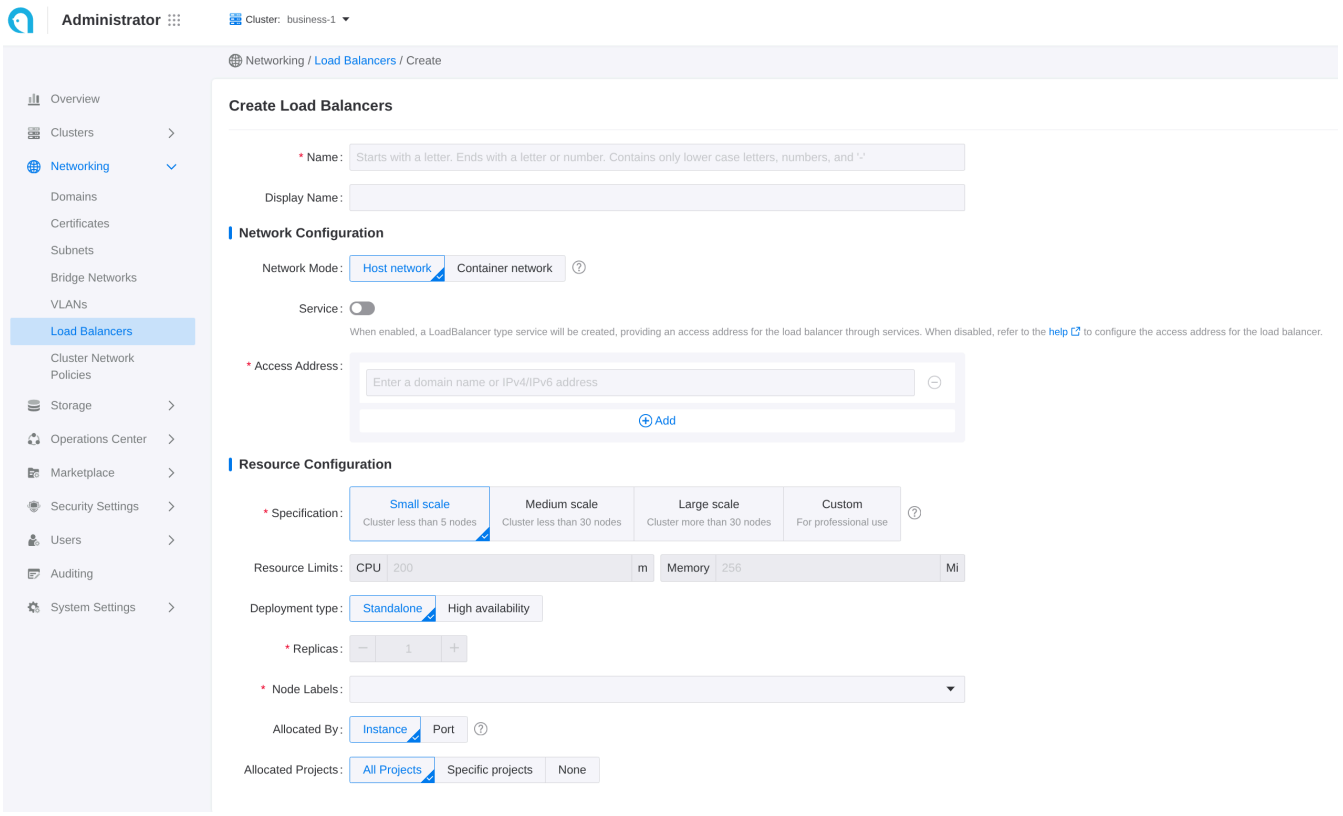
alb cr 中有一些可调整的全局配置。

- Ingress 同步

ALB 操作

创建

使用 **Web** 控制台。



Web UI 中提供了一些常见配置。请按照以下步骤创建负载均衡器：

1. 进入管理员。
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
分配的地址	`.spec.address`
分配方式	`Instance` 表示项目模式，您可以在下方选择项目；port 表示 port-project 模式，创建 alb 后可以分配端口范围

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服务中断 3 至 5 分钟。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. 根据需要更新网络和资源配置。
   - 请根据业务需求合理设置规格。您也可以参阅相关的如何合理分配 CPU 和内存资源以获取指导。
   - 内部路由仅支持从已禁用状态更新为已启用状态。
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一并删除，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
```

前端

Frontend 是一种自定义资源，用于定义 ALB 的监听端口和协议。 支持的协议：L7（http|https|grpc|grpcs）和 L4（tcp|udp）。

- 在 L4 Proxy 中，使用 frontend 直接配置后端服务。
- 在 L7 Proxy 中，使用 frontend 配置监听端口，并使用 rule 配置后端服务。

如果需要添加 HTTPS 监听端口，还应联系管理员为当前项目分配 TLS 证书以进行加密。

前置条件

请先创建一个 ALB。

配置 **Frontend**

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

(1) alb label：必填，表示此 `Frontend` 所属的 ALB 实例。

(2) frontend name：格式为 `$alb_name-$port`。

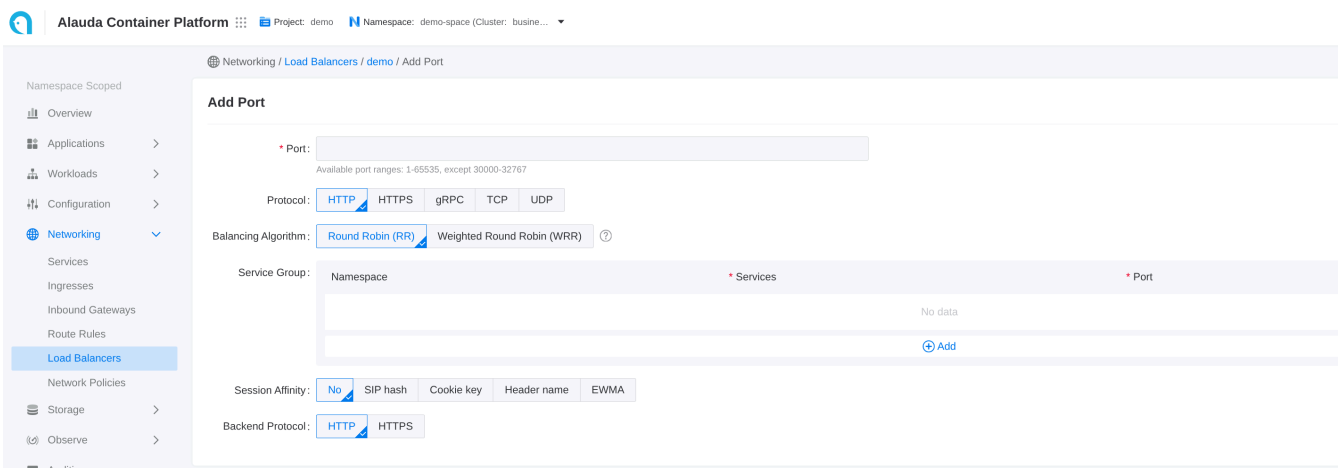
(3) port：监听的端口。

4. protocol：此端口使用的协议。
   - L7 协议 https|http|grpcs|grpc，L4 协议 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，后端协议默认为 gRPC，不支持会话持久性。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 流量转发到后端服务。
   - 如果使用 Google GKE 集群，同一容器网络类型的负载均衡器不能同时使用 TCP 和 UDP 监听协议。
5. certificate_name：grpcs 和 https 协议所使用的默认证书，格式为 `$secret_ns/$secret_name`。
6. backendProtocol：后端服务使用的协议。
7. 默认 `serviceGroup`：
   - L4 proxy：必填。ALB 直接将流量转发到默认服务组。
   - L7 proxy：选填。ALB 首先匹配此 Frontend 上的规则；如果没有匹配项，则回退到默认的 `serviceGroup`。
8. session_affinity_policy

Frontend 操作

创建

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，进入详情页。
4. 点击添加端口。

Web UI 中的每个输入项对应 CR 的一个字段

参数	描述
会话亲和性	`.spec.serviceGroup.session_affinity_policy`

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，还可以设置更多样化的后端服务匹配规则。负载均衡器会先根据设置的规则匹配相应的后端服务；如果规则匹配失败，则会匹配上述内部路由组对应的后端服务。

相关操作

可以点击列表页右侧的⋮图标，或点击详情页右上角的操作，根据需要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为端口，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 是一种自定义资源（CR），用于定义 ALB 如何匹配和处理传入请求。

ALB 处理的 Ingress 可以自动转换为规则。

前提条件

- 配置 ALB
- 配置 Frontend

下面是一个示例规则，可帮助你快速了解如何使用规则。

> **NOTE**
>
> 必须通过标签将 rule 关联到 frontend 和 alb。

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

(1) 必填，指示此规则所属的 `Frontend`。

(2) 必填，指示此规则所属的 ALB。

(3) backendProtocol

4. certificate_name
5. dslx
6. 数值越小，优先级越高。
7. 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专用语言。 例如，以下规则会匹配满足所有以下条件的请求：

- url 以 /app-a 或 /app-b 开头
- method 为 post
- url 参数的 group 为 vip
- host 为 *.app.com
- header 的 location 为 east-1 或 east-2
- 包含名为 uid 的 cookie
- 源 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'
```

优先级

优先级是一个介于 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，可以使用以下注解格式：

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

对于规则，只需在 `.spec.priority` 中直接设置一个整数值作为优先级。

Action

请求匹配规则后，可以对该请求应用以下操作：

功能	说明	链接
超时	配置请求的超时设置。	timeout
重定向	将传入请求重定向到指定 URL。	redirect
CORS	为原生应用启用跨源资源共享（CORS）。	cors
Header 修改	允许修改请求或响应的 Header。	header modification
URL 重写	转发传入请求前重写其 URL。	url-rewrite
WAF	集成 Web Application Firewall（WAF），增强安全性。	waf
OTEL	启用 OpenTelemetry（OTEL）以进行分布式追踪和监控。	otel
Keepalive	启用或禁用原生应用的 keepalive 功能。	keepalive

Backend

后端协议

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

Service Group 和会话亲和性策略

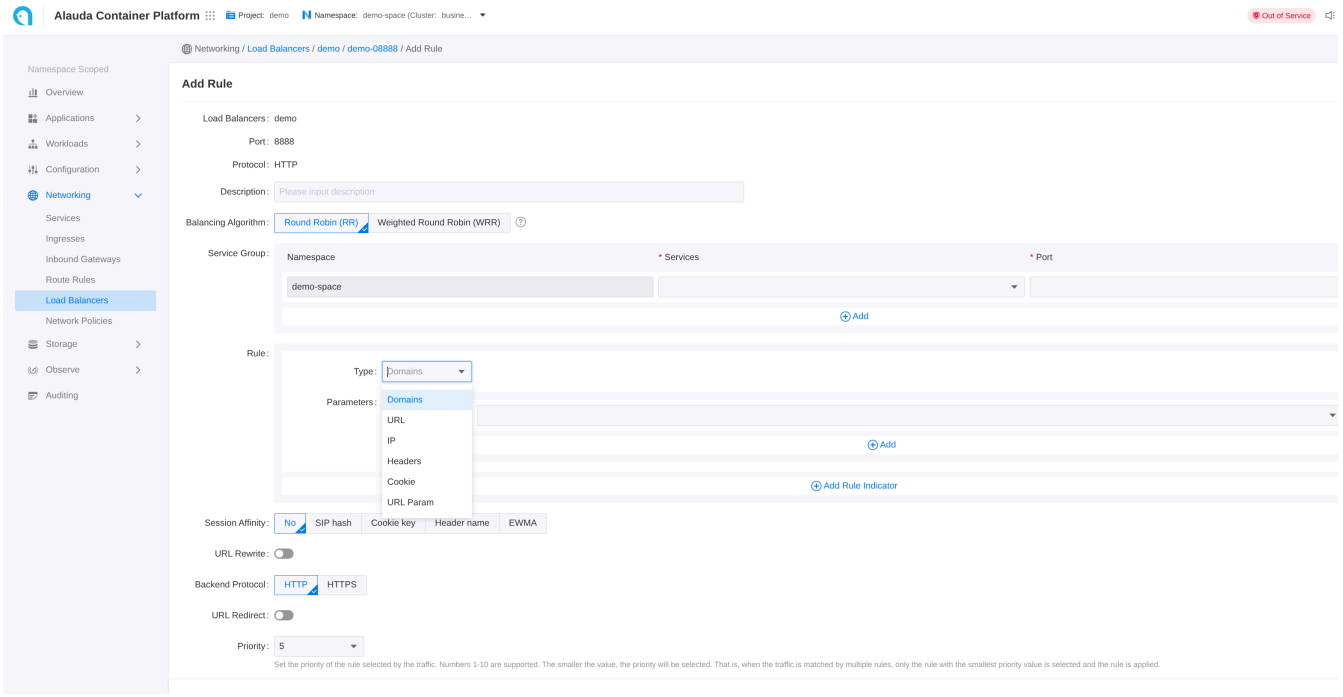
可以在一个规则中配置一个或多个服务。

默认情况下，ALB 使用轮询（RR）算法在后端服务之间分配请求。不过，也可以为各个服务分配权重，或选择其他负载均衡算法。

详情请参阅负载均衡算法。

Rule 操作

使用 web console



1. 进入容器平台。
2. 在左侧导航栏中点击网络 > 负载均衡。
3. 点击负载均衡器的名称。
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。

webui 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 协议（ft）为 HTTPS 或 GRPCS，也可以将规则配置为使用 HTTPS。

可以在规则或 ingress 中指定证书，以匹配相应端口的证书。

支持终止 TLS；如果后端协议为 HTTPS，也可以进行重新加密。不过，不能为与后端服务的通信指定证书。

Ingress 中的证书注解

可以通过注解跨 namespace 引用证书。

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

Rule 中的证书

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

TLS 模式

Edge 模式

在 Edge 模式下，客户端通过 HTTPS 与 ALB 通信，而 ALB 使用 HTTP 协议与后端服务通信。 要实现此配置：

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

重新加密模式

在重新加密模式下，客户端通过 HTTPS 与 ALB 通信，而 ALB 使用 HTTPS 协议与后端服务通信。 要实现此配置：

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，并处理同一项目中的 Ingress。

当 ingress namespace 带有类似 `cpaas.io/project: demo` 的标签时，表示该 ingress 属于 `demo` 项目。

在 `.spec.config.projects` 配置中将项目名称设为 `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` 目前仅支持 `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 提供了一种使用默认证书的策略。

可以使用以下设置配置 ALB 自定义资源：

- `.spec.config.defaultSSLStrategy`：定义未配置证书的 ingress 的 SSL 策略
- `.spec.config.defaultSSLCert`：按 `$secret_ns/$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**：在 HTTPS 端口上使用默认证书创建规则

日志与监控

结合日志和监控数据，您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，从容器视角查看负载均衡器运行时的日志。

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，从节点视角查看负载均衡器的指标趋势信息。

   - 使用率：负载均衡器在当前节点上对 CPU 和内存的实时使用情况。
   - 吞吐量：负载均衡器实例的整体入站和出站流量。

有关监控指标的更多详细信息，请参阅 ALB 监控。

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 或 F5, Inc. 不存在关联，也未获得其认可或赞助。所有商标均为其各自所有者的财产，此处仅用于识别目的。

配置 Alauda Container Platform 节点 DNS 缓存

目录

概述
主要功能
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
已知风险和临时解决方法
风险 1：主机网络端口冲突
风险 2：节点级 DNS 依赖
风险 3：通过节点 IP 收集指标
商标声明

概述

NodeLocal DNSCache 是一种集群插件，通过在集群节点上运行 DNS 缓存代理来提升集群 DNS 性能。此插件通过在每个节点上本地缓存 DNS 响应，减少 DNS 查询延迟并降低中心 DNS 服务的负载，从而提升集群稳定性。

主要功能

- 本地 **DNS** 缓存：在每个节点上本地缓存 DNS 响应，以减少查询延迟
- 性能提升：显著减少应用程序的 DNS 查找时间

重要说明

WARNING

部署注意事项：

1. **Kube-OVN Underlay** 模式：此插件不支持在 Kube-OVN Underlay 模式下部署。若进行部署，可能会导致 DNS 查询失败。
2. **Kubelet** 重启：部署此插件会导致 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，但只会对新创建的 Pod 生效。当 CNI 为 Kube-OVN 时，需要手动向 kube-ovn-controller 添加参数 `"--node-local-dns-ip=(本地 DNS 缓存服务器的 IP 地址)"`。
4. **NetworkPolicy** 配置：如果集群中配置了 NetworkPolicy，则需要在 networkPolicy 中额外允许节点 CIDR 和 nodeLocalDNSIP 的入方向和出方向，以确保正常通信。
5. 通过重建进行集群升级：如果通过重建节点（重新置备）来升级集群，Kubelet 配置更改将会丢失。要使 NodeLocal DNS 配置在升级过程中保持持久化，需要在集群模板的以下三个位置，将参数 `--cluster-dns` 添加到 `kubeletExtraArgs` 中：
   - `KubeadmControlPlane` → `initConfiguration` → `nodeRegistration` → `kubeletExtraArgs`
   - `KubeadmControlPlane` → `joinConfiguration` → `nodeRegistration` → `kubeletExtraArgs`
   - `KubeadmConfigTemplate` → `template` → `spec` → `joinConfiguration` → `nodeRegistration` → `kubeletExtraArgs`

将以下参数添加到上述每个 `kubeletExtraArgs` 部分：

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

WARNING

4.2.x 升级说明

将此插件从低于 4.2.0 的版本（不包括 4.2.0）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` 资源对应的 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：此兼容性问题已在 4.3 及更高版本中解决，因此升级到 4.3+ 时无需手动干预。

安装

通过 **Marketplace** 安装

1. 导航至 管理员 > **Marketplace** > 集群插件。
2. 在插件列表中搜索 "**Alauda Container Platform Node DNS Cache**"。
3. 单击 安装，打开安装配置页面。
4. 配置所需参数：

参数	描述	示例值
IP	节点本地 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，最好使用	169.254.20.10

参数	描述	示例值
	169.254.20.10。对于 IPv6，建议使用 fd00::/8 范围内的地址，最好使用 fd00::10。	

5. 查看部署说明，并确保您的环境满足要求。
6. 单击 安装 完成安装。
7. 等待插件状态变为 "就绪"。

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：如果您的集群启用了 NetworkPolicy，则必须配置适当的规则，以允许 DNS 流量访问 NodeLocal DNSCache。如果没有这些规则，Pod 可能无法解析 DNS 查询。

使用 NetworkPolicy 时，请确保允许以下 DNS 流量：

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-cache
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 169.254.20.10/32 # NodeLocal DNS IP 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
```

已知风险和临时解决方法

请将此功能视为尚未达到生产就绪状态的 Alpha 功能。建议在测试或预生产环境中使用。对于生产集群或类似生产环境的集群，启用 NodeLocal DNSCache 前请评估以下风险。如果这些风险对于您的工作负载不可接受，请继续直接使用 CoreDNS。

在插件安装页面公开相关参数之前，仅在目标环境中完成验证后，才能使用以下临时解决方法。

风险 1：主机网络端口冲突

`node-cache` Pod 在主机网络命名空间中运行。其健康检查端点使用主机网络端口，可能与使用相同端口的工作负载或节点级进程发生冲突。例如，默认健康检查端点监听端口 `8080`。

安装前，请确认目标节点上的工作负载不依赖相同的主机网络端口，例如端口 `8080`：

```
ss -lntp | grep ':8080'
```

编辑 NodeLocal DNSCache ConfigMap：

```
kubectl -n <namespace> edit cm <configmap-name>
```

将 Corefile `health` 端口更改为相同的未使用端口，例如：

```
health 127.0.0.1:18080
```

编辑 NodeLocal DNSCache DaemonSet：

```
kubectl -n <namespace> edit ds <daemonset-name>
```

将 `node-cache` 容器探针端口更改为相同端口：

```
livenessProbe:
  httpGet:
    host: 127.0.0.1
    path: /health
    port: 18080
```

等待 DaemonSet 滚动更新完成：

```
kubectl -n <namespace> rollout status ds/<daemonset-name>
```

如果当前插件版本未在安装页面公开健康检查端口参数，请使用站点级补丁，确保 Corefile `health` 端口与 DaemonSet 探针端口保持一致。

> **WARNING**
>
> 重要：这是临时的站点补丁，不是持久化的插件配置。插件升级、重新安装或平台协调可能会重新生成 ConfigMap 和 DaemonSet，并覆盖您的自定义端口设置。如果在这些操作后仍需要使用非默认健康检查端口，请重新应用补丁，然后再将插件视为就绪。

风险 2：节点级 DNS 依赖

NodeLocal DNSCache 生效后，节点上的 Pod 会使用本地 `node-cache` Pod 进行 DNS 解析。如果该 Pod 在插件升级、驱逐或崩溃期间不可用，则同一节点上 Pod 的 DNS 解析可能失败。请为 `node-cache` 就绪状态配置监控和告警。

如果 DNS 解析已经失败，且之前未配置备用 DNS 服务器，请暂时将 kubelet `cluster-dns` 指向 CoreDNS ClusterIP，然后重新创建受影响的 Pod。

您可以同时使用 NodeLocal DNSCache IP 和 CoreDNS ClusterIP 配置 kubelet `cluster-dns`。在需要进行更改的每个节点上，编辑 kubelet 参数文件：

```
vi /var/lib/kubelet/kubeadm-flags.env
```

将 `--cluster-dns` 从单个 NodeLocal DNSCache IP 更改为同时包含 NodeLocal DNSCache IP 和 CoreDNS ClusterIP：

```
--cluster-dns=169.254.20.10,10.96.0.10
```

保存文件后重启 kubelet：

```
systemctl restart kubelet
```

此更改仅影响新创建的 Pod。请重新创建受影响的 Pod，使其 `/etc/resolv.conf` 重新生成，例如：

```
kubectl -n <namespace> rollout restart deployment/<deployment-name>
```

当 NodeLocal DNSCache 不可用时，可以使用 CoreDNS 作为备用，但这不是透明故障转移。对于 glibc resolver，在使用下一个 DNS 服务器之前，DNS 故障转移可能会在多个 `search` 查询期间等待超时。基于 musl 的镜像（例如 Alpine Linux）通常不会出现此延迟，因为 musl 可以并行查询名称服务器。

如果工作负载对 DNS 解析延迟敏感，请针对该工作负载调整 resolver `timeout` 和 `attempts`。例如：

```
spec:
  dnsConfig:
    options:
      - name: timeout
        value: "1"
      - name: attempts
        value: "1"
```

WARNING

重要：编辑节点上的 kubelet 文件同样属于临时的站点更改。节点重建、重新预配或任何会重新生成 kubelet 配置的工作流，都可能完全丢弃多 DNS 设置。如果在这些操作后仍需要 CoreDNS 备用配置，请重新应用 kubelet 更改，然后再次重新创建受影响的 Pod。

风险 3：通过节点 IP 收集指标

如果外部监控无法通过节点 IP 收集 `node-cache` 指标，请编辑 NodeLocal DNSCache ConfigMap：

```
kubectl -n <namespace> edit cm <configmap-name>
```

在 Corefile 中，从 `prometheus` 指令中移除 IP 绑定。请勿更改 DNS 服务端口。

```
prometheus 169.254.20.10:9253
```

```
prometheus :9253
```

重启 `node-cache` Pod，或滚动更新 DaemonSet，使更新后的 Corefile 生效。

> **WARNING**
>
> 重要：这同样是临时的站点补丁，不是持久化的插件配置。如果直接修补生成的 ConfigMap，插件升级、重新安装或平台协调可能会覆盖 `prometheus` 绑定，并将指标端点恢复为之前的默认值。如果仍需要通过节点 IP 收集外部指标，请在这些操作后重新应用更改。

商标声明

CoreDNS™ 是 The Linux Foundation 的商标。

Kubernetes® 是 The Linux Foundation 在美国及其他国家/地区的注册商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 无关联，也未获其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识。

配置 Alauda Container Platform 集群 DNS

目录

概述
配置
主机别名
节点选择器
节点容忍度
商标声明

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. 导航至 管理员 > **Marketplace** > 集群插件。
2. 搜索 "**Alauda Container Platform** 集群 **DNS**"，然后点击 更新。
3. 配置以下参数：

主机别名

配置自定义 DNS 解析条目。

参数	描述
IP	要解析的 IP 地址
域名	域名（以空格分隔） 示例：`example.com test.example.com`

节点选择器

指定 CoreDNS Pod 应在哪些节点上运行。

参数	描述
标签键	要在节点上匹配的标签键
标签值	要在节点上匹配的标签值

节点容忍度

允许 CoreDNS Pod 调度到带有污点的节点上。

参数	描述
键	要容忍的污点键
值	污点值（可选）
类型	`NoSchedule`、`PreferNoSchedule` 或 `NoExecute`

4. 点击 更新 以应用配置。

商标声明

CoreDNS™ 是 The Linux Foundation 的商标。

Alauda 是独立的供应商。本产品与 The Linux Foundation 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处使用这些商标仅用于识别目的。

网络可观测性

网络可观测性提供持续的集群范围网络流量收集以及按需的数据包或流捕获，以便排查故障。它使用 eBPF 检查流量行为、识别性能问题，并调查工作负载之间的连接问题。

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 任务

Ingress-Nginx 任务

前提条件

最大连接数

请求超时

会话亲和性（粘性会话）

修改标头

URL 重写

HSTS（HTTP Strict Transport Security）

速率限制

WAF

Forward-header 控制

HTTPS

保留源 IP

商标声明

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
商标声明

Gateway API CRD 版本

Gateway API CRD 版本

概述
ACP 版本映射
商标声明

Soft Data Center LB Solution（Alpha）

Soft Data Center LB Solution（Alpha）

前提条件
操作步骤
验证
商标声明

Kube OVN

了解 Kube-OVN CNI

网络模式和范围

架构和组件

逻辑网络

网络变更如何成为流

流量路径

高可用性

检查集群中的组件

后续步骤

准备 Kube-OVN Underlay 物理网

使用说明

术语说明

环境要求

配置示例

Underlay 和

操作步骤

启用 u2oIntercc

Kube-OVN

集群互联（Alpha）

支持在网络模式为 Kube-OVN 的集群之间配置集群互联，以实

前提条件

配置 Egress Gateway

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

已验证容量指南

可能中断流量的操作

其他资源

配置集中式网关

使用 Label 选择器指定网关节点

配置 Kube-

开始之前

安装 Multus CN

创建辅助网络

创建具有多个网

配置 IPPool

说明

注意事项

商标声明

配置 MTU

默认 MTU 行为

自定义 MTU 设置

配置 BGP

安装 `kube-ovn`

发布 Pod 和 Su

为 Kube-OVN Underlay 网络配置 IPv4 多播

开始之前

启用 Multicast Snooping

查看配置结果

关闭 Multicast Snooping

配置 Endpoint Health Checker

配置 Endpoint Health Checker

概述

主要功能

安装

工作原理

如何激活

卸载

商标声明

alb

ALB 的任务

如何为 alb-operator 设置 NodeSelector 和 Tolerations

如何为 alb 设置 NodeSelector 和 Tolerations

商标声明

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 严格传输安全（HSTS）

基于 Cookie 的会话亲和性

基于路径的路由

标头修改

连接限制

限流

IP 允许列表/阻止列表

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 重新加密

使用自定义证书的边缘终止

TLS 透传

功能对比摘要

迁移策略

相关文档

商标声明

Ingress-Nginx 任务

目录

前提条件
最大连接数
请求超时
会话亲和性（粘性会话）
修改标头
URL 重写
HSTS（HTTP Strict Transport Security）
速率限制
  限制特定路径
  配置拒绝状态码
WAF
Forward-header 控制
HTTPS
  TLS 重新加密并验证后端证书
  TLS 边缘终止
  直通
  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
如何配置
通过启用 externalTrafficPolicy=Local 的 MetalLB
工作原理
如何配置
商标声明

前提条件

安装 ingress-nginx

最大连接数

Max-Worker-Connections ↗

请求超时

配置请求超时 ↗

会话亲和性（粘性会话）

配置粘性会话 ↗

修改标头

操作	链接
在请求中设置标头	proxy-set-header ↗

操作	链接
在请求中移除标头	在请求中设置一个空标头
在响应中设置标头	使用 more-set-header ↗ 指令的 configuration-snippets ↗
在响应中移除标头	hide-headers ↗

URL 重写

rewrite ↗

HSTS（HTTP Strict Transport Security）

配置 HSTS ↗

速率限制

Ingress NGINX 支持通过 Ingress 注解限制请求速率和并发连接数。完整的注解参考请参阅速率限制 ↗。

限制特定路径

注解应用于 Ingress 对象。如果只需要限制一个路径（例如登录端点），请将该路径放在单独的 Ingress 对象中，以免其他路径受到相同注解的限制。

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: login-rate-limit
  namespace: demo
  annotations:
    nginx.ingress.kubernetes.io/limit-rps: "10"
    nginx.ingress.kubernetes.io/limit-burst-multiplier: "2"
    nginx.ingress.kubernetes.io/limit-connections: "20"
spec:
  ingressClassName: demo-nginx
  rules:
    - host: app.example.com
      http:
        paths:
          - path: /auth/login
            pathType: Exact
            backend:
              service:
                name: app
                port:
                  number: 80
```

- `limit-rps` 限制每个 Controller Pod 处理的每个客户端 IP 的每秒请求数。
- `limit-burst-multiplier` 允许在配置速率之外额外突发请求。突发大小根据 `limit-rps` 和此乘数计算。
- `limit-connections` 限制每个 Controller Pod 处理的每个客户端 IP 的并发连接数。

这些限制在每个 Controller Pod 内本地生效。存在多个副本时，整个集群的有效阈值取决于流量分布和副本数，而不是一个共享计数器。在依赖基于 IP 的限制之前，请保留真实客户端 IP。

配置拒绝状态码

`limit-req-status-code` 和 `limit-conn-status-code` 是适用于整个 Controller 的设置。请在 `IngressNginx` 资源上配置它们，而不要编辑生成的 ConfigMap：

```yaml
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo-nginx
  namespace: ingress-nginx-operator
spec:
  controller:
    config:
      limit-req-status-code: "429"
      limit-conn-status-code: "429"
```

更改这些值会影响由该 Controller 实例处理的每个 Ingress。当其他应用程序无法接受相同的拒绝状态码时，请使用专用 Controller。

WAF

modsecurity ↗

Forward-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

后端协议 ↗

直通

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，原始客户端 IP 地址可能会因 NAT（网络地址转换）而丢失。保留源 IP 对以下方面很重要：

- 访问控制和安全策略
- 准确的日志记录和分析
- 按客户端进行速率限制
- 基于地理位置的路由

通过 HAProxy Proxy Protocol

工作原理

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

流量流向：

1. 客户端连接到 HAProxy 负载均衡器
2. HAProxy 将包含原始客户端 IP 的 PROXY protocol 标头添加到连接前
3. Ingress-Nginx 接收连接并解析 PROXY protocol 标头
4. Ingress-Nginx 从标头中提取真实客户端 IP
5. 后端应用在 `X-Forwarded-For` 和 `X-Real-IP` 标头中接收到正确的客户端 IP

优势：

- 适用于支持 PROXY protocol 的任何负载均衡器（HAProxy、AWS NLB 等）
- 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，所有到 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）

如何配置

配置 HAProxy 负载均衡器以发送 PROXY protocol 标头，然后部署启用了 proxy-protocol 支持的 ingress-nginx：

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

有关更多详细信息，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，因此可以让 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。

通过启用 **externalTrafficPolicy=Local** 的 **MetalLB**

工作原理

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

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

流量流向：

1. 客户端连接到 MetalLB 虚拟 IP
2. MetalLB 将流量直接路由到运行 Ingress-Nginx Pod 的节点
3. 流量在不执行 SNAT 的情况下直接到达本地 Ingress-Nginx Pod
4. Ingress-Nginx 获取真实客户端 IP
5. 后端应用在标头中接收到正确的客户端 IP

优势：

- 配置简单，无需额外协议
- Kubernetes 原生功能
- 延迟更低（没有额外的代理跳转）

注意事项：

- 负载分布不均：流量只能发送到具有本地 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 确保两者一致）
- 运行状况检查行为：如果所有本地 Pod 都不健康，该节点会从负载均衡中完全移除
- 带污点的节点：如果选定的节点具有 `NoSchedule` 污点，请为 Ingress-Nginx 和 MetalLB 配置匹配的容忍度

如何配置

使用 `externalTrafficPolicy: Local` 部署 ingress-nginx，并确保 Pod 放置位置与 MetalLB 配置一致：

```yaml
apiVersion: ingress-nginx.alauda.io/v1
kind: IngressNginx
metadata:
  name: demo
  namespace: ingress-nginx-operator
spec:
  controller:
    service:
      type: LoadBalancer                    # Use MetalLB to provision a LoadBalancer service
      externalTrafficPolicy: Local          # Preserve source IP by routing traffic only to local pods
      annotations:
        metallb.universe.tf/address-pool: demo-pool  # Specify the MetalLB IP address pool to use
    nodeSelector:                           # Schedule pods only on nodes matching these labels. This selector must match the MetalLB L2Advertisement node selector
      dedicated: ingress
    tolerations:
      - key: dedicated
        operator: Equal
        value: ingress
        effect: NoSchedule
```

重要事项：

- `nodeSelector` 必须与 MetalLB `L2Advertisement` 选择的节点匹配，以便将 Ingress-Nginx Pod 调度到能够接收 MetalLB 流量的节点上。
- 在 MetalLB 自定义资源上配置相同的节点选择和容忍度：对 speaker 使用 `spec.nodeSelector` 和 `spec.speakerTolerations`，对 controller 使用 `spec.controllerNodeSelector` 和 `spec.controllerTolerations`。默认情况下，MetalLB 不容忍其他污点。

有关更多详细信息，请参阅 externalTrafficPolicy 文档↗。

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 或 F5, Inc. 不存在关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，本文仅将其用于标识目的。

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 Listener

如何使用在另一个命名空间中创建的证书

如何使用 SSL Passthrough

如何更改最低 TLS 版本

使用 NodePort Service 时如何指定 NodePort

使用 MetalLB 时如何指定 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：通过 useListenerPortAsContainerPort 直接使用特权端口

如何从集群内部访问 LoadBalancer VIP

如何配置 HSTS

如何配置自定义错误页面

相关文档

更多配置

商标声明

概述

完成主要配置流程（operator、网关、路由和策略）后，可执行以下 Envoy Gateway 高级任务。

在 Gateway API 中应用配置更改时，主要有三种可用方式：

1. 修改标准 **Gateway API** 资源：直接编辑 `Gateway`、`HTTPRoute`、`TCPRoute`、`UDPRoute` 和其他核心资源。
2. 通过 **PolicyAttachment** 附加策略：使用 `SecurityPolicy`、`ClientTrafficPolicy`、`BackendTrafficPolicy` 和其他策略资源扩展网关和路由行为。
3. 配置全局设置：修改 `EnvoyGatewayCtl`，以更改 `envoy-gateway` 实例行为或其他影响所有网关的全局设置。

高级任务和特殊场景包括跨命名空间路由、可观测性配置、部署自定义和故障排除。

前提条件

1. 配置 EnvoyGatewayCtl
2. 配置网关
3. 配置路由
4. 配置 GatewayAPI 策略

高级任务

OpenTelemetry (OTel)

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

如何附加到在其他命名空间中创建的 **Listener**

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

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: example-gateway
spec:
  listeners:
    - name: http-80
      protocol: HTTP
      port: 80
      allowedRoutes:
        namespaces:
          from: All # 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 la
bel team=frontend
```

有关详情，请参阅跨命名空间路由↗。

如何使用在另一个命名空间中创建的证书

若要使用在另一个命名空间中创建的证书，请在存储证书的命名空间中创建 `ReferenceGrant`。请按照跨命名空间证书引用↗和 referencegrant↗ 中的说明操作。

> **NOTE**
>
> 不能指定单个 `secret` 资源；必须允许整个命名空间

如何使用 SSL Passthrough

请按照以下说明操作：

- tls↗
- tls-passthrough↗

如何更改最低 TLS 版本

请按照自定义网关 TLS 参数↗中的说明操作。

```
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` 字段对应 clienttlssettings↗。 如果除了最低 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。

使用 NodePort Service 时如何指定 NodePort

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。

有两种方式：

按照从 svc 端口获取 nodeport中的说明手动获取 NodePort 分配值。

在 `EnvoyProxy` 配置中手动指定 NodePort，而不是让 Kubernetes 自动分配。

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

1 使用 patch 字段修改生成的 Service 资源，以指定 NodePort

> **NOTE**
>
> NodePort 只能位于特定范围内，通常为 `30000-32767`。如果希望 Gateway Listener 端口与 NodePort 保持一致，Listener 端口也必须位于 NodePort 范围内。

使用 MetalLB 时如何指定 VIP

使用 MetalLB 作为 LoadBalancer 提供程序时，可以通过 Service 注解为 Gateway Service 指定静态 VIP。

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

1. 在 `envoyService.annotations` 字段中添加 MetalLB 注解
2. 指定用于分配 IP 的地址池名称
3. 或者指定一个特定 IP 地址（必须位于地址池范围内）

可用注解：

注解	说明
`metallb.universe.tf/address-pool`	选择用于分配 IP 的地址池
`metallb.universe.tf/loadBalancerIPs`	指定一个特定 IP 地址（支持多个 IP，以逗号分隔）

NOTE

- 指定的 IP 必须位于已配置的 MetalLB 地址池中
- 指定 VIP 之前，请确保已正确安装和配置 MetalLB

- 有关 MetalLB 配置，请参阅配置 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`。

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo
spec:
  infrastructure:
    parametersRef:
      group: gateway.envoyproxy.io
      kind: EnvoyProxy
      name: demo
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: http
      port: 80
      protocol: HTTP
---
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
spec:
  provider:
    kubernetes:
      envoyDeployment:
        replicas: 1
        patch:
          type: StrategicMerge
          value:
            spec:
              template:
                metadata:
                  annotations:
                    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` 可让 Envoy 代理 Pod 直接使用主机网络命名空间。这在以下场景中很有用：

- 提升网络性能
- 通过节点 IP 直接访问网关

注意事项：

- 使用 hostNetwork 的 Pod 会直接绑定到主机的网络接口
- 如果多个 Pod 尝试在同一节点上使用相同端口，可能会发生端口冲突
- 由于 Pod 共享主机的网络命名空间，安全隔离性会降低
- 应使用 `nodeSelector` 或 `affinity` 规则控制 Pod 的放置位置，避免端口冲突

根据是否要直接使用特权端口（< 1024），配置 `hostNetwork` 有两种方式：

方法 **1**：使用端口偏移（默认，推荐）

这是默认且推荐的方式。Envoy Gateway 会自动为特权端口增加 10000 的偏移量，从而无需特殊权限。

优点：

- 无需特殊权限或功能
- 以非 root 用户身份运行且无需额外权限，因此更安全
- 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：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: demo
  namespace: demo
spec:
  provider:
    type: Kubernetes
    kubernetes:
      envoyDeployment:
        patch:
          type: StrategicMerge
          value:
            spec:
              template:
                spec:
                  hostNetwork: true                    # Enable host netw
ork mode
                  dnsPolicy: ClusterFirstWithHostNet   # Required for pro
per DNS resolution
        pod:
          nodeSelector:                                # Recommended: con
trol pod placement to avoid conflict
            kubernetes.io/hostname: "demo"
```

访问：

- 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：通过 **useListenerPortAsContainerPort** 直接使用特权端口

此方法允许 Envoy 直接绑定到特权端口（< 1024），例如端口 80 和 443。

优点：

- 可直接使用标准端口（80 和 443）
- 与预期使用标准端口的客户端兼容性更好

缺点：

- 需要 NET_BIND_SERVICE 功能
- 与端口偏移方式相比，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：

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

如何从集群内部访问 **LoadBalancer VIP**

默认情况下，Envoy Gateway 创建的 LoadBalancer Service 使用 `externalTrafficPolicy: Local`。此策略会保留客户端源 IP 地址，但有一项重要限制：来自没有 Envoy Gateway Pod 的集群节点的请求会失败，因为流量不会转发到其他节点。

解决方案 **1**：使用 **Service ClusterIP**（建议用于集群内访问）

对于在集群内运行的应用，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**：更改为集群流量策略

如果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/exter
nalTrafficPolicy", "value": "Cluster"}
]'
```

如何配置 **HSTS**

HTTP Strict Transport Security (HSTS) 会告知浏览器仅通过 HTTPS 与某个主机名交互。ACP 使用 `ClientTrafficPolicy` `.spec.headers.lateResponseHeaders` 在 Gateway 上配置 HSTS。Gateway Listener 会在完成其他所有标头修改后应用延迟响应标头，因此每个 Gateway 响应都会包含 `Strict-Transport-Security` 标头，包括 Envoy 自身生成的响应（例如请求不匹配任何路由时返回的 404）。路由级别的 `ResponseHeaderModifier` 过滤器无法覆盖此类响应。

此功能适用于 ACP 4.5 及更高版本。

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: ClientTrafficPolicy
metadata:
  name: hsts-policy
  namespace: default
spec:
  targetRef:
    group: gateway.networking.k8s.io
    kind: Gateway
    name: eg
  headers:
    lateResponseHeaders:
      set:
      - name: Strict-Transport-Security
        value: "max-age=31536000; includeSubDomains"
EOF
```

验证普通响应以及主机不匹配时 Envoy 生成的 404 响应中的标头：

```
curl -sk -v --resolve www.example.com:443:<gateway-ip> https://www.exampl
e.com/ 2>&1 | grep -i strict-transport
curl -sk -v https://<gateway-ip>/ 2>&1 | grep -E "^< HTTP|strict-transpor
t"
```

注意：

- 使用 `set` 而不是 `add`，以免后端降低标头值的安全级别。`addIfAbsent` 和 `remove` 也可在 `lateResponseHeaders` 中使用。
- HSTS 不会将 HTTP 流量重定向到 HTTPS；它只会告知已通过 HTTPS 收到此标头的浏览器继续使用 HTTPS。如有需要，也请在路由上配置 HTTP 到 HTTPS 的重定向。
- 浏览器会将此决定缓存 `max-age` 秒，在此期间无法从服务器撤销。正式部署一年期限前，请先使用较短的 `max-age`（例如 `300`）进行验证。
- 有关完整 API，请参阅官方 Envoy Gateway HTTP 响应标头文档↗。

如何配置自定义错误页面

默认情况下，Gateway 会提供 Envoy 生成的纯文本错误响应正文（例如 503 时的 `no healthy upstream`），并原样传递后端错误响应。可通过两种机制返回自定义错误页面；两者均适用于 ACP 4.5 及更高版本：

- `BackendTrafficPolicy` `.spec.responseOverride[]` 会重写与状态码匹配的响应。
- `HTTPRouteFilter` `.spec.directResponse` 会为路由规则返回固定响应，而不联系任何后端。

`responseOverride` 可按状态码自定义错误页面。正文可以是内联字符串，也可以是 `ConfigMap` 的 `response.body` 键：

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: BackendTrafficPolicy
metadata:
  name: error-page-policy
  namespace: default
spec:
  targetRef:
    group: gateway.networking.k8s.io
    kind: HTTPRoute
    name: backend
  responseOverride:
  - match:
      statusCodes:
      - type: Value
        value: 404
    response:
      contentType: text/html
      body:
        type: Inline
        inline: "<html><body><h1>Page not found</h1></body></html>"
  - match:
      statusCodes:
      - type: Value
        value: 500
    response:
      contentType: application/json
      body:
        type: ValueRef
        valueRef:
          group: ""
          kind: ConfigMap
          name: error-page-config
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: error-page-config
  namespace: default
data:
  response.body: '{"error": "Internal Server Error"}'
EOF
```

每条规则的可选字段 `source` 用于控制重写哪些响应：`All`（默认值）会重写后端和 Envoy 生成的响应；`Backend` 仅重写来自后端的响应；`Local` 仅重写 Envoy 自身生成的响应，例如路由匹配成功但后端没有就绪端点时产生的 503。使用具有相同状态码的 `Local` 和 `Backend` 规则，为网关错误和后端错误提供不同页面。规则还可以重写状态码（`response.statusCode`），或从另一个 URL 获取响应（`redirect`）。

`directResponse` 可以在不联系任何后端的情况下响应路径。通过 `ExtensionRef` 将此过滤器应用于 HTTPRoute 规则；`HTTPRouteFilter` 必须与 HTTPRoute 位于同一命名空间，且正文大小不能超过 4096 字节：

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: backend
  namespace: default
spec:
  parentRefs:
  - name: eg
  hostnames:
  - www.example.com
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /maintenance
    filters:
    - type: ExtensionRef
      extensionRef:
        group: gateway.envoyproxy.io
        kind: HTTPRouteFilter
        name: maintenance-page
  - backendRefs:
    - name: backend
      port: 3000
---
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: HTTPRouteFilter
metadata:
  name: maintenance-page
  namespace: default
spec:
  directResponse:
    contentType: text/html
    statusCode: 503
    body:
      type: Inline
      inline: "<html><body><h1>Under maintenance</h1></body></html>"
EOF
```

验证重写后的页面：

```
curl -s -w '\n%{http_code}\n' --resolve www.example.com:80:<gateway-ip> h
ttp://www.example.com/missing-page
```

注意：

- `responseOverride` 仅适用于经过目标 HTTPRoute 的响应。完全没有匹配到路由的请求会收到默认的 Envoy 404，且不会应用任何规则。
- 直接响应由 Envoy 自身生成，因此同一 HTTPRoute 上具有匹配状态码（例如 503，且使用 `source: Local` 或默认的 `source: All`）的 `responseOverride` 规则会重写该直接响应。如果同时使用这两种机制，请将直接响应配置在单独的 HTTPRoute 上。
- 有关完整 API，请参阅 Envoy Gateway 官方文档中的响应覆盖↗和直接响应↗。

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 任务↗

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是一家独立供应商。本产品与 The Linux Foundation 无关，也未获得其认可或赞助。所有商标均为其各自所有者的财产，在此仅用于标识目的。

Gateway API CRD 版本

目录

概述

ACP 版本映射

ACP 3.16.x - 4.1.x

ACP 4.2.1 - 4.2.x

ACP 4.3.x

ACP 4.4.x

商标声明

概述

ACP 在平台生命周期中维护并安装标准 Gateway API 自定义资源定义（CRD）。Envoy Gateway Operator 在 `gateway.envoyproxy.io` API 组中管理 Envoy Gateway 扩展 CRD；它不负责 `gateway.networking.k8s.io` API 组中的标准 Gateway API CRD。

ACP 使用 Gateway API Experimental 通道中的 Gateway API CRD。此通道包含标准 Gateway API 资源和实验性 Gateway API 资源，例如 `BackendTLSPolicy` 和 `ListenerSet`。

请勿在 ACP 管理的集群中安装第二套 Gateway API CRD。创建或升级 `Gateway`、Route、Policy 和 ListenerSet 资源时， 请使用 ACP 版本提供的 CRD。

ACP 3.16 是首个包含 Gateway API CRD 的版本。下表记录了 ACP 为每个版本线维护的 CRD。

ACP 版本映射

每个版本线表格每行列出一个 CRD。`Type` 用于标识 CRD 来自 Gateway API 项目还是 Envoy Gateway。`Version` 包含提供服务的版本，`Storage version` 包含用于持久化 资源的版本，`Bundle version` 用于标识 Gateway API CRD 包。`N/A` 表示该 CRD 不属于 Gateway API 包。

ACP 3.16.x - 4.1.x

类型	CRD	版本	存储版本	包版本
Gateway API	`GatewayClass`	v1alpha2, v1, v1beta1	v1beta1	v1.0.0
Gateway API	`Gateway`	v1alpha2, v1, v1beta1	v1beta1	v1.0.0
Gateway API	`GRPCRoute`	v1alpha2	v1alpha2	v1.0.0
Gateway API	`HTTPRoute`	v1alpha2, v1, v1beta1	v1beta1	v1.0.0
Gateway API	`ReferenceGrant`	v1alpha2, v1beta1	v1beta1	v1.0.0
Gateway API	`TCPRoute`	v1alpha2	v1alpha2	v1.0.0
Gateway API	`TLSRoute`	v1alpha2	v1alpha2	v1.0.0
Gateway API	`UDPRoute`	v1alpha2	v1alpha2	v1.0.0
Gateway API	`BackendTLSPolicy`	v1alpha2	v1alpha2	v1.0.0

更改：

- Gateway API CRD 首次在 ACP 3.16 中引入。
- 此版本线包含初始标准 Gateway API CRD 集，不包含 ACP Gateway API 扩展 CRD。

ACP 4.2.1 - 4.2.x

类型	CRD	版本	存储版本	包版本
Gateway API	`GatewayClass`	v1alpha2, v1, v1beta1	v1	v1.3.0
Gateway API	`Gateway`	v1alpha2, v1, v1beta1	v1	v1.3.0
Gateway API	`GRPCRoute`	v1alpha2, v1	v1	v1.3.0
Gateway API	`HTTPRoute`	v1alpha2, v1, v1beta1	v1	v1.3.0
Gateway API	`ReferenceGrant`	v1alpha2, v1beta1	v1beta1	v1.3.0
Gateway API	`TCPRoute`	v1alpha2	v1alpha2	v1.3.0
Gateway API	`TLSRoute`	v1alpha2	v1alpha2	v1.3.0
Gateway API	`UDPRoute`	v1alpha2	v1alpha2	v1.3.0
Gateway API	`BackendTLSPolicy`	v1alpha2, v1alpha3	v1alpha3	v1.3.0
Gateway API	`XBackendTrafficPolicy`	v1alpha1	v1alpha1	v1.3.0
Gateway API	`XListenerSet`	v1alpha1	v1alpha1	v1.3.0

更改：

- Gateway API CRD 包更新为 v1.3.0。
- `GatewayClass`、`Gateway`、`GRPCRoute` 和 `HTTPRoute` 的存储版本更改为 `v1`。
- `BackendTLSPolicy` 添加 `v1alpha3`，并将存储版本更改为 `v1alpha3`。

- 添加 `XBackendTrafficPolicy` 和 `XListenerSet`。

ACP 4.3.x

类型	CRD	版本	存储版本	包版本
Gateway API	`GatewayClass`	v1alpha2, v1, v1beta1	v1	v1.4.1
Gateway API	`Gateway`	v1alpha2, v1, v1beta1	v1	v1.4.1
Gateway API	`GRPCRoute`	v1alpha2, v1	v1	v1.4.1
Gateway API	`HTTPRoute`	v1alpha2, v1, v1beta1	v1	v1.4.1
Gateway API	`ReferenceGrant`	v1alpha2, v1beta1	v1beta1	v1.4.1
Gateway API	`TCPRoute`	v1alpha2	v1alpha2	v1.4.1
Gateway API	`TLSRoute`	v1alpha2, v1alpha3	v1alpha3	v1.4.1
Gateway API	`UDPRoute`	v1alpha2	v1alpha2	v1.4.1
Gateway API	`BackendTLSPolicy`	v1alpha2, v1alpha3, v1	v1	v1.4.1
Gateway API	`XBackendTrafficPolicy`	v1alpha1	v1alpha1	v1.4.1
Gateway API	`XListenerSet`	v1alpha1	v1alpha1	v1.4.1
Gateway API	`XMesh`	v1alpha1	v1alpha1	v1.4.1
Envoy Gateway	`Backend`	v1alpha1	v1alpha1	N/A

类型	CRD	版本	存储版本	包版本
Envoy Gateway	`BackendTrafficPolicy`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`ClientTrafficPolicy`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`EnvoyExtensionPolicy`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`EnvoyPatchPolicy`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`EnvoyProxy`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`HTTPRouteFilter`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`SecurityPolicy`	v1alpha1	v1alpha1	N/A

更改：

- Gateway API CRD 包更新为 v1.4.1。
- `BackendTLSPolicy` 添加提供服务的 `v1`，并将存储版本更改为 `v1`。
- `TLSRoute` 添加提供服务的 `v1alpha3`，并将存储版本更改为 `v1alpha3`。
- 添加 `XMesh`。
- Envoy Gateway CRD 作为 `v1alpha1` 资源包含在内。

ACP 4.4.x

类型	CRD	版本	存储版本	包版本
Gateway API	`GatewayClass`	v1alpha2, v1, v1beta1	v1	v1.5.1

类型	CRD	版本	存储版本	包版本
Gateway API	`Gateway`	v1alpha2, v1, v1beta1	v1	v1.5.1
Gateway API	`GRPCRoute`	v1alpha2, v1	v1	v1.5.1
Gateway API	`HTTPRoute`	v1alpha2, v1, v1beta1	v1	v1.5.1
Gateway API	`ReferenceGrant`	v1, v1alpha2, v1beta1	v1beta1	v1.5.1
Gateway API	`TCPRoute`	v1alpha2	v1alpha2	v1.5.1
Gateway API	`TLSRoute`	v1, v1alpha2, v1alpha3	v1	v1.5.1
Gateway API	`UDPRoute`	v1alpha2	v1alpha2	v1.5.1
Gateway API	`BackendTLSPolicy`	v1, v1alpha2, v1alpha3	v1	v1.5.1
Gateway API	`ListenerSet`	v1	v1	v1.5.1
Gateway API	`XBackendTrafficPolicy`	v1alpha1	v1alpha1	v1.5.1
Gateway API	`XListenerSet`	v1alpha1	v1alpha1	v1.4.1
Gateway API	`XMesh`	v1alpha1	v1alpha1	v1.5.1
Envoy Gateway	`Backend`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`BackendTrafficPolicy`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`ClientTrafficPolicy`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`EnvoyExtensionPolicy`	v1alpha1	v1alpha1	N/A

类型	CRD	版本	存储版本	包版本
Envoy Gateway	`EnvoyPatchPolicy`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`EnvoyProxy`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`HTTPRouteFilter`	v1alpha1	v1alpha1	N/A
Envoy Gateway	`SecurityPolicy`	v1alpha1	v1alpha1	N/A

更改：

- 标准 Gateway API CRD 包更新为 v1.5.1。
- 添加 `ListenerSet`。
- `ReferenceGrant` 添加提供服务的 `v1`。
- `TLSRoute` 添加提供服务的 `v1`，并将存储版本更改为 `v1`。
- `BackendTLSPolicy` 提供服务的版本为 `v1`，并将其用于存储。
- Envoy Gateway CRD 仍为 `v1alpha1`。

有关配置工作流，请参阅 配置 GatewayAPI Gateway、 配置 GatewayAPI Route 和 配置 GatewayAPI Policy。

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

Alauda 是独立的供应商。本产品与 The Linux Foundation 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处使用商标仅用于标识目的。

Soft Data Center LB Solution（Alpha）

通过在集群外部创建高可用负载均衡器，部署纯软件数据中心负载均衡器（LB），为多个 ALB 提供负载均衡能力，确保业务稳定运行。支持仅配置 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

商标声明

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，以缩短 LB 向异常后端节点转发流量所需的时间。
2. 在外部 LB 的所有主机节点上预安装以下软件（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` 在启动时启动。
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；平台已包含此镜像。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。本文以 `build-harbor.alauda.cn/tkestack/keepalived:v3.16.0-beta.3.g598ce923` 为例。

   - 在 global 集群中执行 `kubectl get prdb base -o json | jq .spec.registry.address`，获取镜像仓库地址参数。
   - 在安装包解压后的目录中执行 `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 主机节点上分别执行一次，并且主机节点的 `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. 创建 Keepalived 配置文件的文件夹。

```
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/` 文件夹中，文件命名为 `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 chec
k script
      check_timeout: 3  # optional, default 3: check script timeout per
iod
      name: "vip-1" # Identifier for this instance, can only contain al
phanumeric characters and hyphens, cannot start with a hyphen
      peer: [ "192.168.128.116", "192.168.128.75" ] # Keepalived node I
P, actual generated keepalived.conf will remove all IPs on the interfac
e.
      kube_lock:
        kubecfgs: # The kube-config list used by kube-lock will sequent
ially 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 control plane 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 t
he 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 sequent
ially 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::19
2:168:129:100" ]
      ports:
        - port: 80
          virtual_server_config: |
            delay_loop 10
            lb_algo rr
            lb_kind NAT
            protocol TCP
          raw_check: |
            TCP_CHECK {
                connect_timeout 1
                connect_port 1936
            }
```

6. 在业务集群中执行以下命令，检查配置文件中的证书到期日期，确保该证书仍然有效。证书到期后，LB 功能将不可用，需要联系平台管理员更新证书。

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

7. 将 Kubernetes 集群中三个控制平面节点上的 `/etc/kubernetes/admin.conf` 文件复制到外部 LB 节点的 `/etc/keepalived/kubecfg` 文件夹中，并为文件添加索引命名，例如 `kubecfg01.conf`，然后将这三个文件中的 `apiserver` 节点地址修改为 Kubernetes 集群中节点的实际地址。
   注意：平台证书更新后，需要再次执行此步骤，覆盖原有文件。

8. 检查证书的有效性。

   1. 将业务集群控制平面节点上的 `/usr/bin/kubectl` 复制到 LB 节点。
   2. 执行 `chmod +x /usr/bin/kubectl`，授予执行权限。
   3. 执行以下命令，确认证书有效性。

```
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，则表示证书有效。

```
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/ke
epalived:/live/cfg build-harbor.alauda.cn/tkestack/keepalived:v3.16.0-b
eta.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. 运行命令 `ipvsadm -ln` 查看 IPVS 规则，你将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 InAc
tConn
TCP  192.168.128.118:80 rr
  -> 192.168.129.100:80           Masq    1      0          0
  -> 192.168.138.100:80           Masq    1      0          0
  -> 192.168.143.192:80           Masq    1      0          0
TCP  [2004::192:168:128:118]:80 rr
  -> [2004::192:168:129:100]:80   Masq    1      0          0
  -> [2004::192:168:138:100]:80   Masq    1      0          0
  -> [2004::192:168:143:192]:80   Masq    1      0          0
```

2. 关闭 VIP 所在的 LB 节点，并测试 IPv4 和 IPv6 的 VIP 是否都能成功迁移到另一个节点，通常在 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="htt
p://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at <a href="http://nginx.com/">nginx.co
m</a>.</p>

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

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

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

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

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

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是一家独立供应商。本产品与 F5, Inc. 无关联，也未获得其认可或赞助。所有商标均为其各自所有者的财产，在此仅用于识别目的。

Kube OVN

了解 Kube-OVN CNI

网络模式和范围

架构和组件

逻辑网络

网络变更如何成为流

流量路径

高可用性

检查集群中的组件

后续步骤

准备 Kube-OVN Underlay 物理网

使用说明

术语说明

环境要求

配置示例

Underlay 和

操作步骤

启用 u2oIntercc

Kube-OVN

集群互联（Alpha）

支持在网络模式为 Kube-OVN 的集群之间配置集群互联，以实

前提条件

配置 Egress Gateway

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

已验证容量指南

可能中断流量的操作

其他资源

配置 Kube-

开始之前

安装 Multus CN

创建辅助网络

创建具有多个网

配置集中式网关

使用 Label 选择器指定网关节点

配置 IPPool

说明
注意事项
商标声明

配置 MTU

默认 MTU 行为
自定义 MTU 设置

配置 BGP

安装 kube-ovn
发布 Pod 和 Su

为 Kube-OVN Underlay 网络配置 IPv4 多播

开始之前
启用 Multicast Snooping
查看配置结果
关闭 Multicast Snooping

了解 Kube-OVN CNI

Kube-OVN 使用 Open Virtual Network (OVN) 和 Open vSwitch (OVS) 连接 Kubernetes Pod。它负责分配 Pod 地址、构建逻辑网络，以及实现网络策略和 Service 负载均衡。

本页介绍 Kube-OVN 架构、逻辑网络、配置更新和常见流量路径。此外，还提供组件检查示例和配置指南链接。

目录

网络模式和范围
架构和组件
控制平面组件
kube-ovn-controller
ovn-central
节点组件
ovs-ovn
kube-ovn-cni
监控和可选组件
kube-ovn-pinger
kube-ovn-monitor
kube-ovn-speaker
组件部署
逻辑网络
逻辑交换机和 Subnet

IP 地址分配和 Subnet 选择
join 交换机和节点接口
覆盖封装
网关模式
网络隔离和访问控制
网络变更如何成为流
流量路径
同一节点上的 Pod 到 Pod
跨节点的 Pod 到 Pod
Pod 到 Service
Pod 到节点或 API server
Pod 到外部网络
高可用性
检查集群中的组件
列出工作负载
查找 Pod 和容器
检查滚动更新状态和事件
资源使用量和限制
kubectl-ko
后续步骤

网络模式和范围

Kube-OVN 支持 Overlay 和 Underlay 子网。网络模式决定 Pod 流量如何到达其他节点和物理网络。

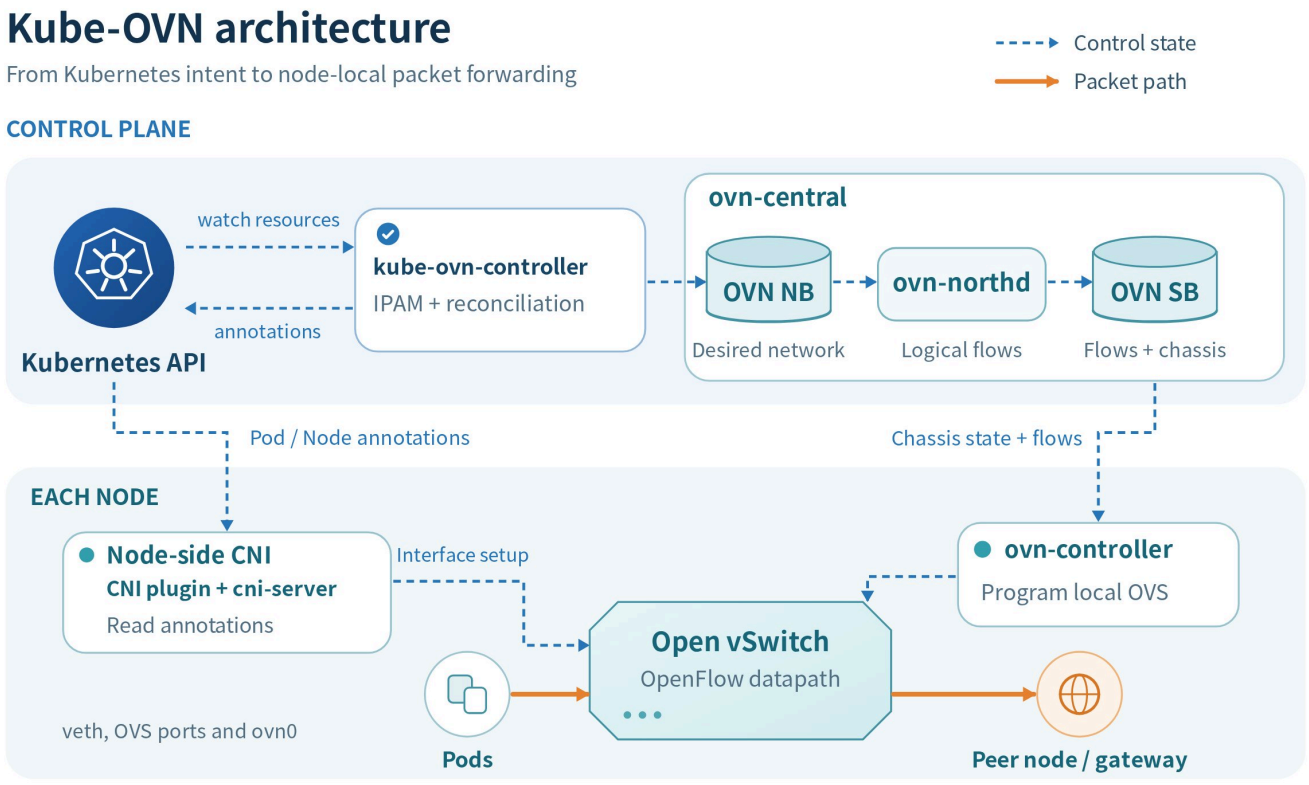
网络模式	**Pod** 的连接方式	规划事项
Overlay	OVS 使用隧道在节点网络上转发跨节点的 Pod 流量。默认隧道类型为 Geneve。	Pod CIDR、节点间连通性、隧道流量和 MTU。

网络模式	**Pod** 的连接方式	规划事项
Underlay	子网通过 VLAN 和 ProviderNetwork（桥接网络）连接到物理网络。该物理网络上的流量不会经过 Overlay 隧道。	物理 IP 分配、节点接口、VLAN 连通性和物理网关。请参阅准备 Underlay 物理网络。

本页中的流量路径示意图描述的是默认 VPC 中使用 Geneve 隧道的 Overlay 网络。Service 示意图还假设已为控制器和源子网启用 OVN Service 负载均衡。示例中显示的名称和 CIDR 作为默认值，可能与你的集群配置不同；请与管理员确认子网设置和已安装的 Kube-OVN 配置。

架构和组件

控制平面维护所需的网络状态，并将其转换为转发规则。节点组件连接 Pod 接口，并在 OVS 中实现这些规则。监控和可选组件提供运行状况信息及外部集成功能。



OVN renders the logical network into local flows. OVS forwards packets on each node.

控制平面组件

kube-ovn-controller

`kube-ovn-controller` Deployment 监视与网络相关的 Kubernetes 资源，包括 Pod、Services、EndpointSlices、Nodes、NetworkPolicies、VPC、Subnets、VLANs 和 ProviderNetworks。

它为 Pod 分配地址，将逻辑端口以及所需的路由或策略写入 OVN northbound 数据库，并在 Pod 注解中记录已分配的网络信息，供节点智能体使用。

ovn-central

`ovn-central` Deployment 在其 `ovn-central` 容器中运行 OVN 控制平面服务：

- `ovn-nb`：northbound 数据库存储所需的逻辑网络，包括交换机、路由器、端口、ACL 和负载均衡器。
- `ovn-sb`：southbound 数据库存储逻辑流、数据路径和端口绑定，以及 chassis 状态。
- `ovn-northd`：将 northbound 配置转换为逻辑流及 southbound 数据库使用的相关记录。

有关上游实现细节，请参阅 OVN 架构↗。有关组件故障的影响，请参阅高可用性。

节点组件

ovs-ovn

`ovs-ovn` DaemonSet 在每个符合条件的 Linux 节点上运行一个 `openvswitch` 容器。该容器包含本地 OVS 数据库、`ovn-controller` 和 `ovs-vswitchd`。OVS 数据库存储网桥和接口配置，`ovn-controller` 编写转发规则，`ovs-vswitchd` 则通过 OVS 数据路径管理数据包转发。

kube-ovn-cni

`kube-ovn-cni` DaemonSet 在每个符合条件的 Linux 节点上运行一个 `cni-server` 容器。其 `install-cni` init 容器会将 `kube-ovn` CNI 插件及支持二进制文件安装到主机的 CNI 二进制文件目录（默认为 `/opt/cni/bin`），并默认将 CNI 配置写入 `/etc/cni/net.d`。容器运行时会调用 `kube-ovn` 插件，该插件向本地 `cni-server` 发送 CNI 请求。

节点侧 CNI 插件和 `cni-server` 协同完成以下工作：

- 在 CNI ADD 期间配置 Pod 接口、IP 地址和路由，并在 CNI DEL 期间移除相应的连接。

- 将 Pod 接口连接到 OVS 端口，并维护其本地网络配置。

`cni-server` 守护进程还通过以下方式维护节点网络：

- 持续协调主机网关过滤、NAT 和路由规则。
- 创建并配置 `ovn0`，将主机网络连接到默认 VPC。

其他节点网络配置有各自的要求。请参阅准备 Underlay 物理网络和集群互联（Alpha）。

监控和可选组件

kube-ovn-pinger

`kube-ovn-pinger` DaemonSet 在每个符合条件的 Linux 节点上运行一个 `pinger` 容器，用于收集 OVS 状态信息，并探测网络连通性和延迟。

kube-ovn-monitor

`kube-ovn-monitor` Deployment 收集 OVN 状态信息和指标。在标准部署中，它运行于为 OVN 中央服务选定的节点上。

kube-ovn-speaker

此可选 DaemonSet 运行于为 BGP 通告选定的节点上，并将路由发布到外部网络，使外部网络可以直接通过 Pod IP 访问。有关安装和 BGP 选项，请参阅配置 BGP。

组件部署

下表介绍 OVN 控制平面与 ACP 安装在同一集群中的标准 ACP 安装。默认情况下，Kubernetes 工作负载使用 `kube-system` 命名空间。如果你的安装使用其他命名空间，或使用单独托管的 OVN 控制平面，请使用实际的命名空间和工作负载配置。

组件	**Kubernetes** 工作负载	部署位置	主容器
`ovn-central`	Deployment	为 OVN 中央服务选定的节点，默认使用 `kube-ovn/role=master`。	`ovn-central`

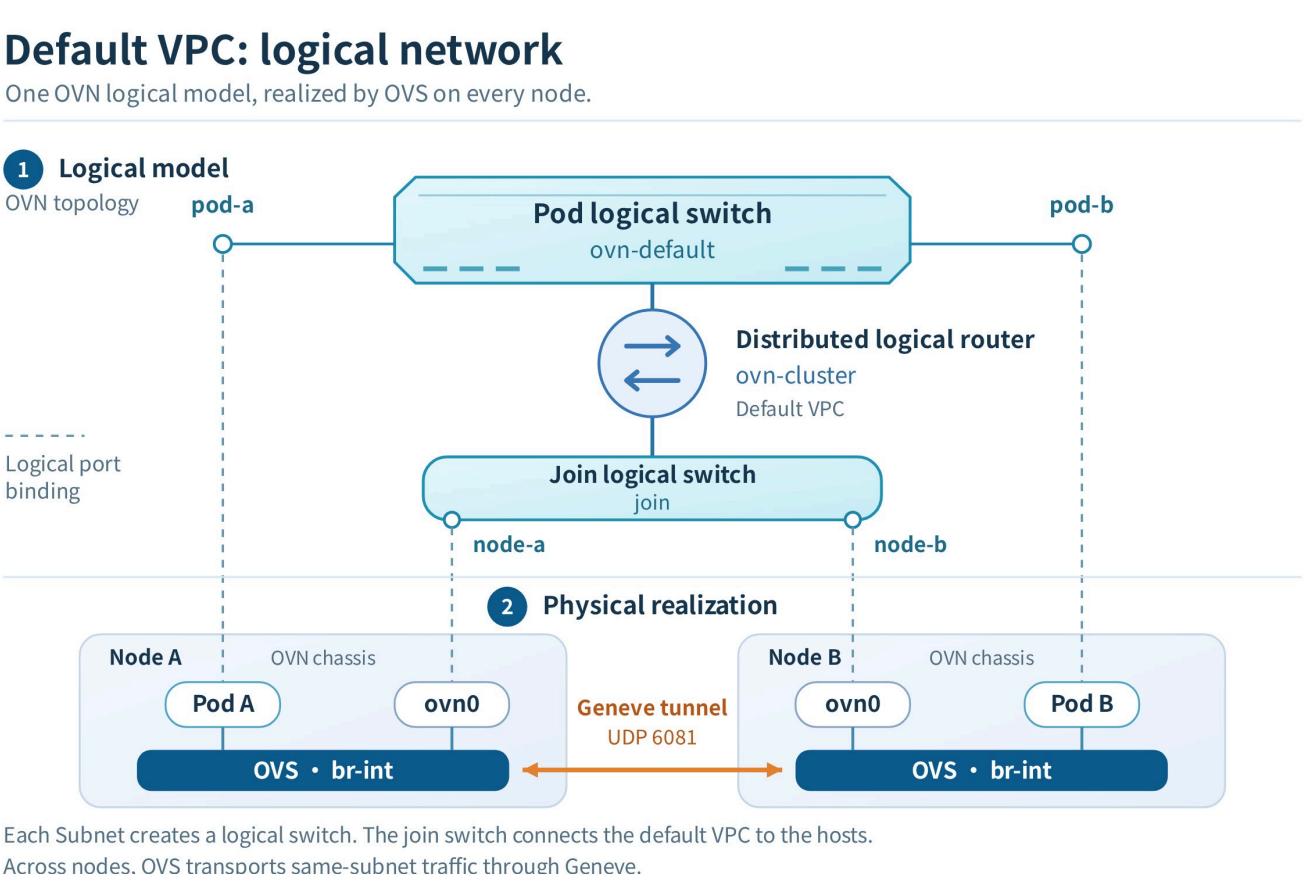
组件	**Kubernetes** 工作负载	部署位置	主容器
`kube-ovn-controller`	Deployment	由调度器选定的符合条件的 Linux 节点；副本分布在各节点上。默认不限制在控制平面节点上。	`kube-ovn-controller`
`ovs-ovn`	DaemonSet	每个符合条件的 Linux 节点。	`openvswitch`
`kube-ovn-cni`	DaemonSet	每个符合条件的 Linux 节点。	`cni-server`
`kube-ovn-pinger`	DaemonSet	每个符合条件的 Linux 节点。	`pinger`
`kube-ovn-monitor`	Deployment	为 OVN 中央服务选定的节点，默认使用 `kube-ovn/role=master`。	`kube-ovn-monitor`
`kube-ovn-speaker`（可选）	DaemonSet	为 BGP 通告选定的节点，示例清单中使用 `ovn.kubernetes.io/bgp=true`。	`kube-ovn-speaker`
`kubectl-ko`	Kubectl 插件	管理员的工作站或其他可访问集群的主机。	不适用

节点选择器、亲和性和容忍度决定哪些节点符合条件。Deployment 副本数取决于安装方式；DaemonSet 会在每个符合条件的节点上放置一个 Pod。可选的 speaker 按照配置 BGP 中的说明单独安装。

有关命令和实际示例输出，请参阅检查集群中的组件。

逻辑网络

OVN 将集群网络建模为逻辑交换机和逻辑路由器，Kube-OVN 则将 Kubernetes 资源映射到这些组件上。VPC 会根据其逻辑路由配置将 Subnet 组织在一起。下图展示了默认 VPC，包括将其连接到主机的 `join` 交换机。自定义 VPC 拥有自己的逻辑路由器，并且需要单独配置路由和外部连接。有关资源字段，请参阅 VPC API 参考。



Each Subnet creates a logical switch. The join switch connects the default VPC to the hosts. Across nodes, OVS transports same-subnet traffic through Geneve.

在此默认 VPC 覆盖网络中，Pod 连接到其 Subnet 的逻辑交换机，Subnet 网关将该交换机连接到 VPC 逻辑路由器，而 `join` 将路由器连接到节点的 `ovn0` 端口。OVS 在节点上实现逻辑交换和路由。跨节点覆盖流量使用隧道；发往主机或外部网络的流量通过相应的节点接口离开覆盖网络。以下各节将介绍其中的每个部分。

逻辑交换机和 Subnet

每个由 OVN 管理的 Subnet 都由一个逻辑交换机承载。连接到 Subnet 的 Pod 会成为逻辑交换机端口。Namespace 可以拥有自己的 Subnet，也可以与其他 Namespace 共享一个 Subnet。

`kube-ovn-controller` 从选定的 Subnet 分配 Pod IP 地址。可以为 Pod 指定 MAC 地址，也可以由 IPAM 生成。控制器会在 Pod 注解中记录已分配的 IP 和 MAC 地址，以及 Subnet CIDR、网关和逻辑交换机。节点侧 CNI 组件会结合这些信息及其他路由配置，配置 Pod 接口和路由。

默认 Subnet `ovn-default` 在标准默认 VPC 部署中承载 Pod 网络。自定义 VPC 拥有自己的逻辑路由器和 Subnet。为一个 Pod 连接多个接口是单独的配置，详见 Pod 多网络接口。

IP 地址分配和 Subnet 选择

Kube-OVN 会从为每个 Pod 接口选择的 Subnet 中分配地址。对于主网络，平台的项目或 Namespace 绑定决定可用的 Subnet，除非 Pod 显式选择了 Subnet。如果没有显式选择或绑定，Pod 将使用其 VPC 的默认 Subnet。有关平台分配规则，请参阅 配置 Subnet。

规划地址时，请考虑以下事项：

- 可用地址：Subnet CIDR 定义地址范围。`excludeIps` 中的保留地址不会自动分配；保留地址并不会将其分配给 Pod。
- **IP** 池：IPPool 将分配范围限制为 Subnet 内的一组地址。Namespace 或 Pod 设置会选择该地址池。在符合条件的地址池中地址耗尽，并不意味着 Pod 可以使用 Subnet 中其他位置的剩余地址。
- 地址耗尽：如果没有可用的符合条件的地址，Pod 网络设置将无法完成。请检查 Pod 事件，以及其选定 Subnet 或 IPPool 中的可用地址。
- **Pod** 替换：替换 Pod 不一定会通过自动分配获得之前 Pod 的地址。如果应用需要稳定的访问地址，请使用 Service。

请选择与集群配置匹配的 CIDR 和地址族。检查其是否与节点网络、Service CIDR、`join` 以及 Pod 必须访问的外部网络发生冲突。对于 Pod 的辅助接口，请按照 Pod 多网络接口 中的独立 Subnet 和网络附加配置进行操作。

join 交换机和节点接口

节点不属于 Pod Subnet。Kube-OVN 通过专用逻辑交换机 `join`（默认 CIDR 为 `100.64.0.0/16`）以及节点上名为 `ovn0` 的 OVS 内部端口，将每个节点连接到覆盖网络。

`ovn0` 将主机网络连接到默认 VPC。`kube-ovn-controller` 从 `join` Subnet 为节点分配地址，并将其记录在 Node 注解中；`kube-ovn-cni` 读取这些注解以创建和配置接口。下文的 Pod 到节点的路径 介绍了流量如何到达节点或该节点上的 API server 端点。

覆盖封装

在默认 Geneve 网络中，不同节点上的 OVS 实例通过隧道连接。`ovn-controller` 安装转发规则，OVS 将跨节点覆盖流量封装在 Geneve（UDP 6081）中，并将其发送到远程节点的封装地址。节点网络必须允许参与通信的节点之间传输此流量。有关由此产生的封装开销，请参阅 配置 MTU。

对于 underlay 流量，VLAN Subnet 通过 `localnet` 端口和 ProviderNetwork 的节点接口连接到物理网络。underlay 与 overlay Subnet 之间的连接需要适当的路由配置；请参阅 underlay 和 overlay Subnet 的自动互联。

网关模式

对于默认 VPC 中的 overlay Subnet，`gatewayType` 选择用于外部出口的节点。对于此 VPC 中除 `join` 之外的 Subnet，控制器会将省略的 `gatewayType` 设置为 `distributed`。

逻辑路由器使用策略路由，将这些流量定向到所选节点在 `join` 交换机上的地址：

- 分布式：每个节点都充当运行在其上的 Pod 的网关，因此南北向流量会通过本地节点退出。
- 集中式：外部出口会定向到选定的网关节点。默认情况下，多个网关节点使用主备模式；ECMP 需要 `spec.enableEcmp: true`。

这些模式控制外部出口；Pod 之间的流量或发往 Service 的流量不会自动通过集中式网关。请参阅下文的外部访问路径以及配置集中式网关。

网络隔离和访问控制

Namespace 或 Subnet 分配决定工作负载所属的位置以及其接收地址的方式。仅凭此分配，并不会阻止工作负载访问其他 Namespace 或 Subnet 中的工作负载。同一 VPC 中的 Subnet 可以通过其逻辑路由器进行通信，前提是路由和访问控制规则允许这样做。

请针对相应目的使用以下控制项：

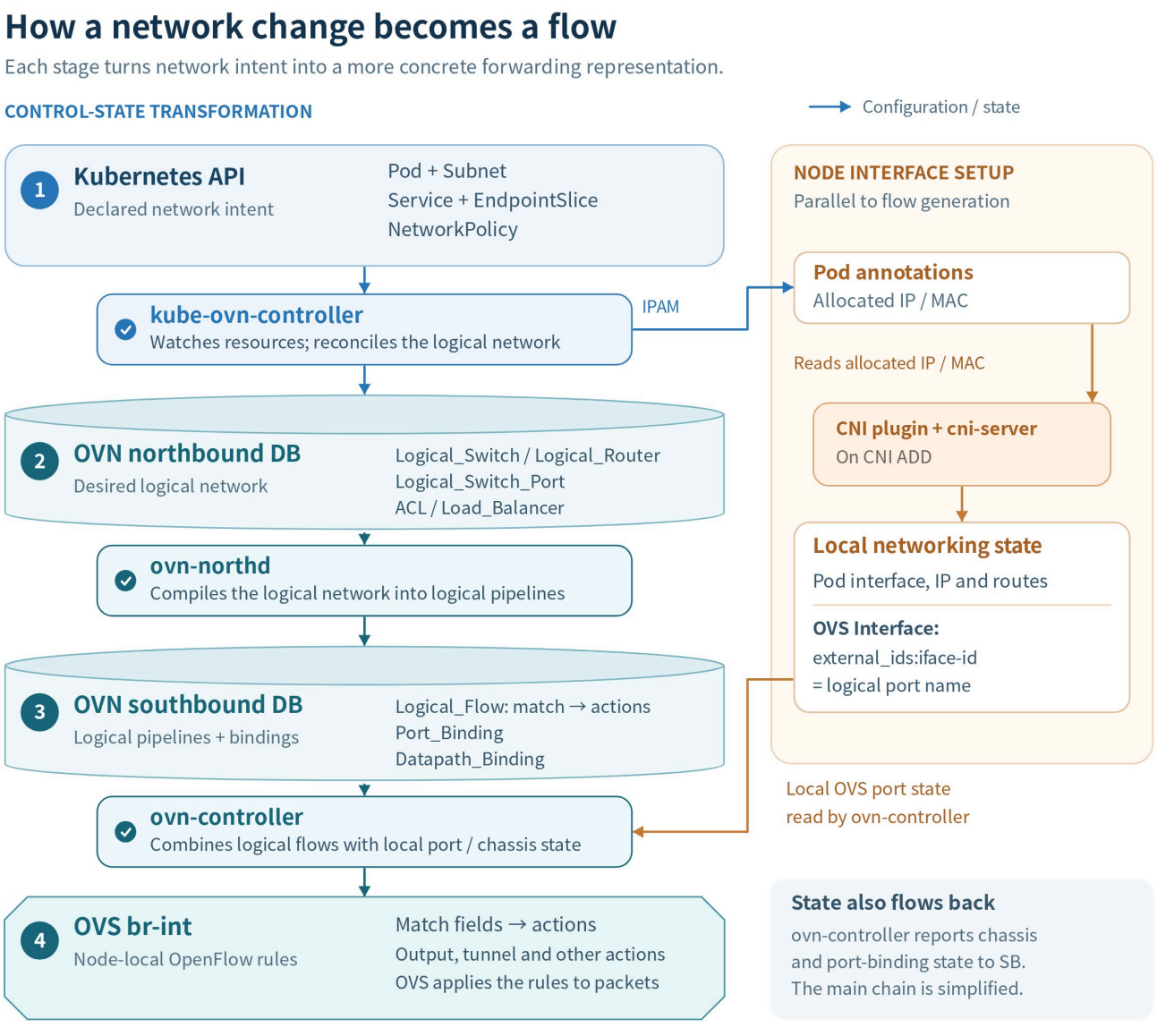
控制项	目的
VPC	在单独的逻辑路由配置下组织 Subnet。在定义 VPC 之间的隔离时，请考虑已配置的路由、网关、物理网络以及其他 Pod 接口。
NetworkPolicy	选择 Pod 并控制其入站或出站流量。策略可以使用 Pod 和 Namespace 选择器或 IP 范围。
Subnet 隔离和 ACL	在 Subnet 的逻辑网络层应用访问控制规则。诊断连接问题时，请同时检查这些规则和 NetworkPolicy。

对于标准 NetworkPolicy，入站和出站隔离会分别进行评估。当某个方向处于隔离状态时，该方向上的流量必须获得选择该 Pod 的策略允许；一个连接必须同时满足两端适用的规则。如果配置了集群范围的策略或 Subnet ACL，其操作和评估顺序也会影响结果。请参阅 Network Policy 和 Network Policy API。

使用 `hostNetwork` 的 Pod 共享节点网络，并且在 Subnet 上没有普通的 Pod 网络接口。请勿使用普通 Pod NetworkPolicy 假设来描述其隔离情况；应检查节点的访问控制以及已记录的策略限制。

网络变更如何成为流

Kubernetes 资源描述所需的网络状态。Kube-OVN 和 OVN 会将此配置转换为逐步细化的表示形式，直到每个节点都拥有转发数据包所需的 OpenFlow 规则。



Logical flows do not map one-to-one to OpenFlow rules. These arrows carry state, not application traffic.

箭头表示配置和状态更新。数据包转发在节点上的 OVS 中进行。

1. 从 **Kubernetes** 资源到逻辑网络对象。 `kube-ovn-controller` 监视 Pod、Subnet、Service 和 NetworkPolicy 等资源，并将其协调为 OVN northbound 数据库（`ovn-nb`）中的对象：逻辑交换机、路由器、端口、ACL 和负载均衡器描述所需的连接。
2. 从逻辑网络对象到逻辑流。 `ovn-northd` 读取 northbound 配置，并在 OVN southbound 数据库（`ovn-sb`）中构建逻辑处理管道。 `Logical_Flow` 记录描述匹配条件和操作；`Datapath_Binding` 和 `Port_Binding` 记录标识逻辑数据路径和端口。
3. 从逻辑流到节点本地 **OpenFlow** 规则。 每个节点上的 `ovn-controller` 将 southbound 信息与其 chassis 标识和本地 OVS 接口状态结合起来。它会在本地 OVS 集成网桥 `br-int` 中安装 OpenFlow 规则，使 OVS 能够在该节点上实现逻辑网络。

Pod 接口设置会提供此过程所需的本地状态。创建 Pod 期间，节点侧 CNI 插件与 `cni-server` 协作，使用分配的 IP 和 MAC 地址配置 Pod 接口和 OVS 端口。OVS 接口的 `external_ids:iface-id` 标识对应的 OVN 逻辑端口，使 `ovn-controller` 能够将该端口与节点关联起来。 `ovn-controller` 还会将 chassis 和端口绑定状态报告回 southbound 数据库。

管理员可以使用 kubectl-ko 检查这些层。

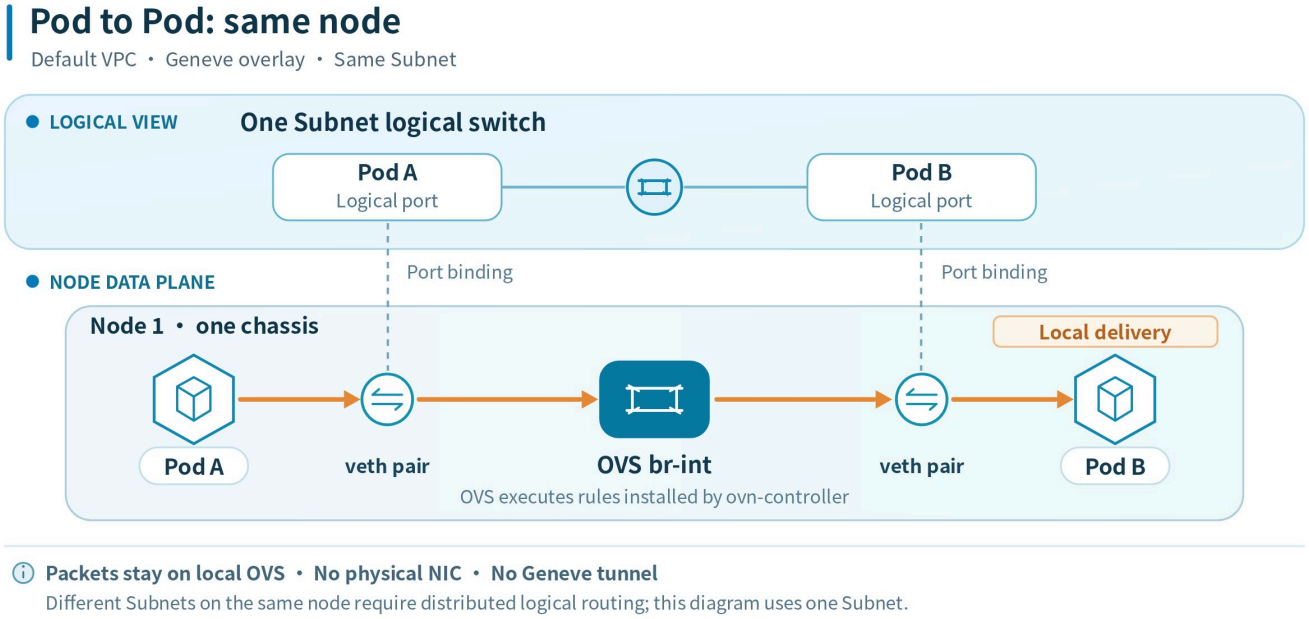
流量路径

下图描述了使用 Geneve 隧道的默认 VPC 覆盖网络。OVS 执行由 `ovn-controller` 安装的逻辑交换、路由和负载均衡规则。NetworkPolicy 和 Subnet ACL 可以允许或阻止这些路径上的流量。

底层网络和自定义 VPC 出口具有不同的拓扑。解读这些图时，请参考部署所采用的网络模式和范围。

同一节点上的 **Pod** 到 **Pod**

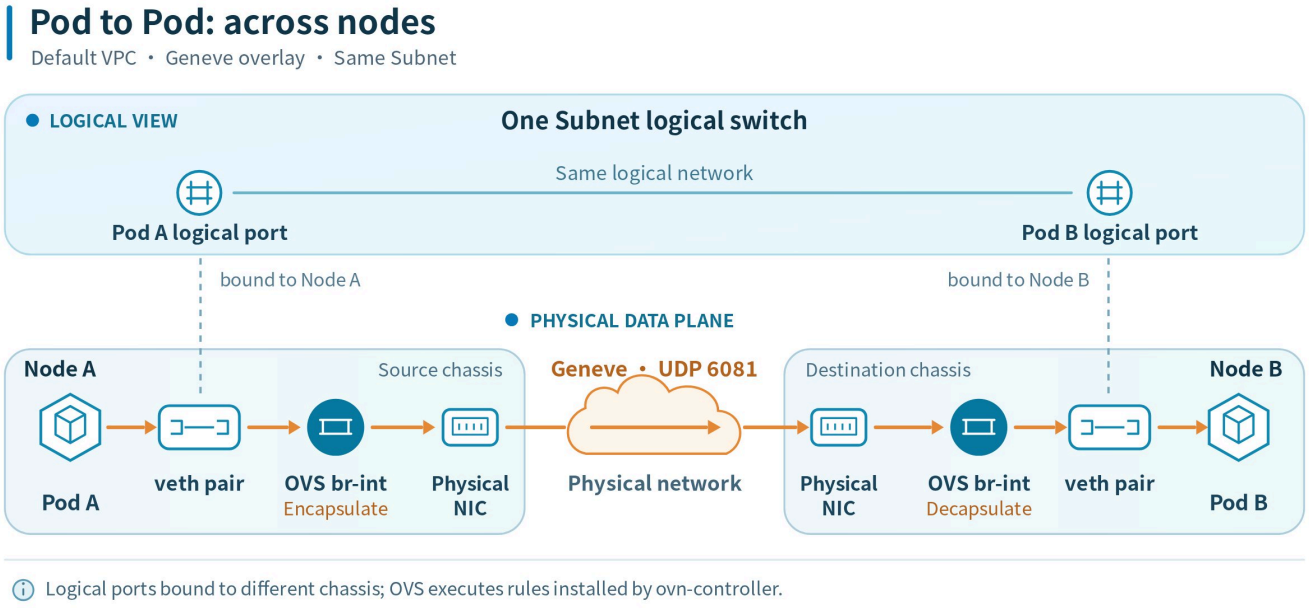
对于同一节点上同一 Subnet 中的两个 Pod，数据包会从源 Pod 接口经过本地 OVS 网桥 `br-int`，到达目标 Pod 接口。逻辑交换机提供转发规则；数据包不会经过物理 NIC 或 Geneve 隧道。



如果 Pod 属于同一 VPC 中的不同 Subnet，则由 VPC 逻辑路由器提供路由规则。如果两个 Pod 位于同一节点上，OVS 仍会在本地处理流量。

跨节点的 **Pod** 到 **Pod**

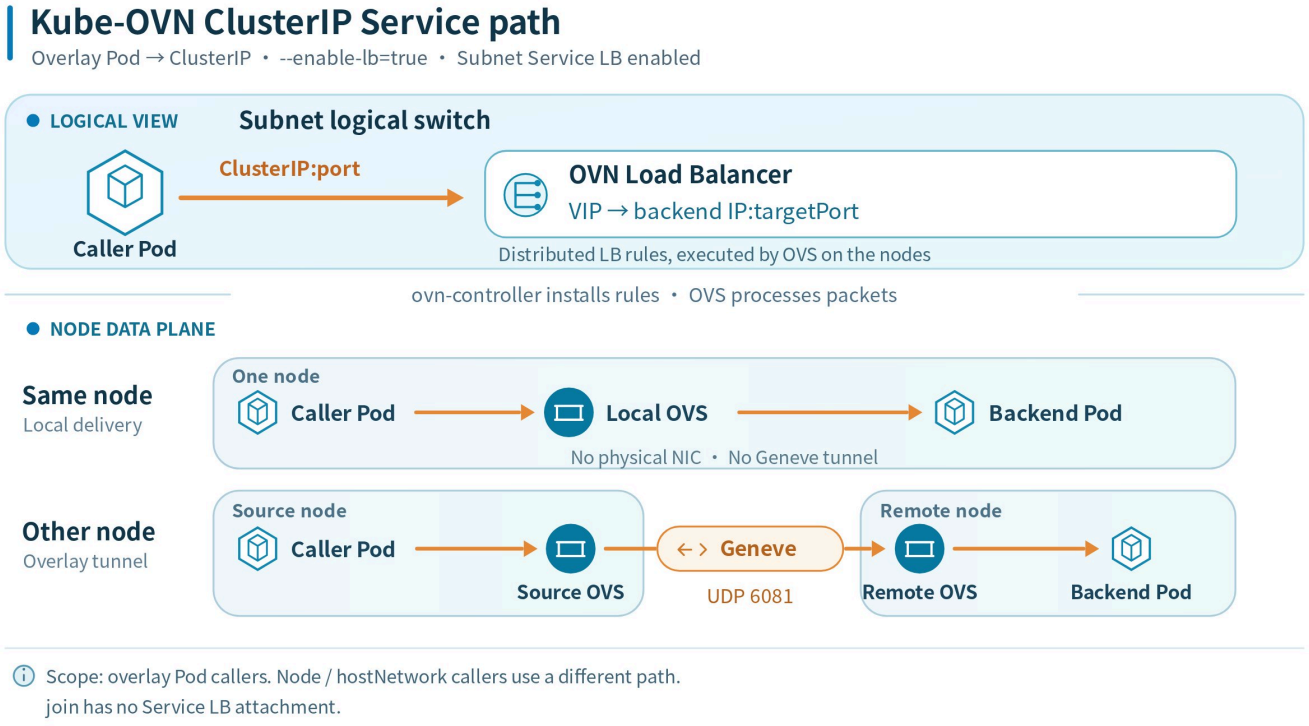
对于位于不同节点、但属于同一 Subnet 的 Pod，逻辑交换机会连接它们的逻辑端口，这些端口分别绑定到不同的节点 chassis。源 OVS 会将数据包封装到 Geneve 中，通过节点网络发送；目标 OVS 会对其解封装，并将其交付给目标 Pod。



如果 Pod 属于同一 VPC 中的不同 Subnet，逻辑路由器也会在 Subnet 之间路由数据包。跨节点的覆盖网络传输仍会使用 OVS 实例之间的 Geneve 隧道。

Pod 到 Service

为控制器和 Subnet 启用 OVN Service 负载均衡后，`kube-ovn-controller` 会在符合条件的 Subnet 逻辑交换机上维护 OVN 负载均衡器。OVS 会将 Service 虚拟 IP 和端口转换为选定后端端点的 IP 和目标端口。对于 Pod 后端，之后的转发会遵循上文介绍的同节点或跨节点路径。



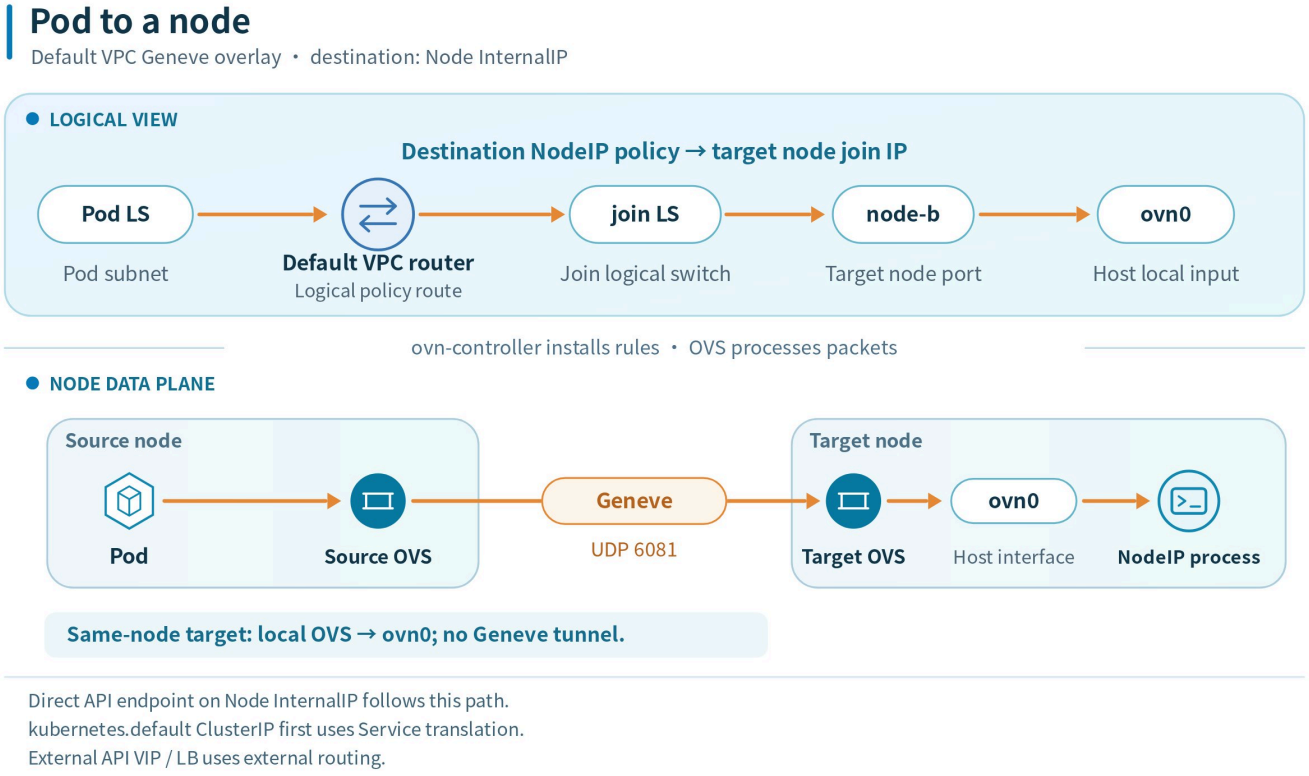
`join` 交换机不会关联 OVN Service 负载均衡器。该图展示了使用 Pod 后端的普通覆盖网络 Pod 调用方。其他 Service 访问场景还需考虑以下条件：

访问场景	注意事项
普通覆盖网络 Pod 访问虚拟 IP	仅当控制器和源 Subnet 均启用了 OVN Service 负载均衡时，此图才适用。禁用该功能会改变 Service 流量的处理方式；请与管理员确认集群的 Service 实现方式。
节点或 `hostNetwork` Pod 访问虚拟 IP	流量源自主机网络，具体取决于节点上配置的 Service 实现方式和路由。流量不会从图中所示的 Pod 逻辑端口进入。
外部客户端访问主机端口或负载均衡器	流量进入集群的方式取决于所配置的暴露机制。网关模式和流量策略可能会影响可达性和源 IP 保留。
Pod 访问无头 Service	没有虚拟 IP 可供转换。对于 Pod 端点，客户端会使用 Pod 到 Pod 路径连接到端点地址。

有关创建和暴露 Service 的信息，请参阅配置 Service。将外部访问与集中式网关结合使用前，请先查看其主机端口和 hostport 限制。

Pod 到节点或 API server

对于发往节点内部 IP 的流量，默认 VPC 逻辑路由器会使用目标策略路由，将数据包导向该节点在 `join` 交换机上的地址。数据包通过目标主机的 `ovn0` 接口到达目标主机。发往远程节点的流量会在 OVS 实例之间通过 Geneve 传输；发往本地节点的流量则留在本地。



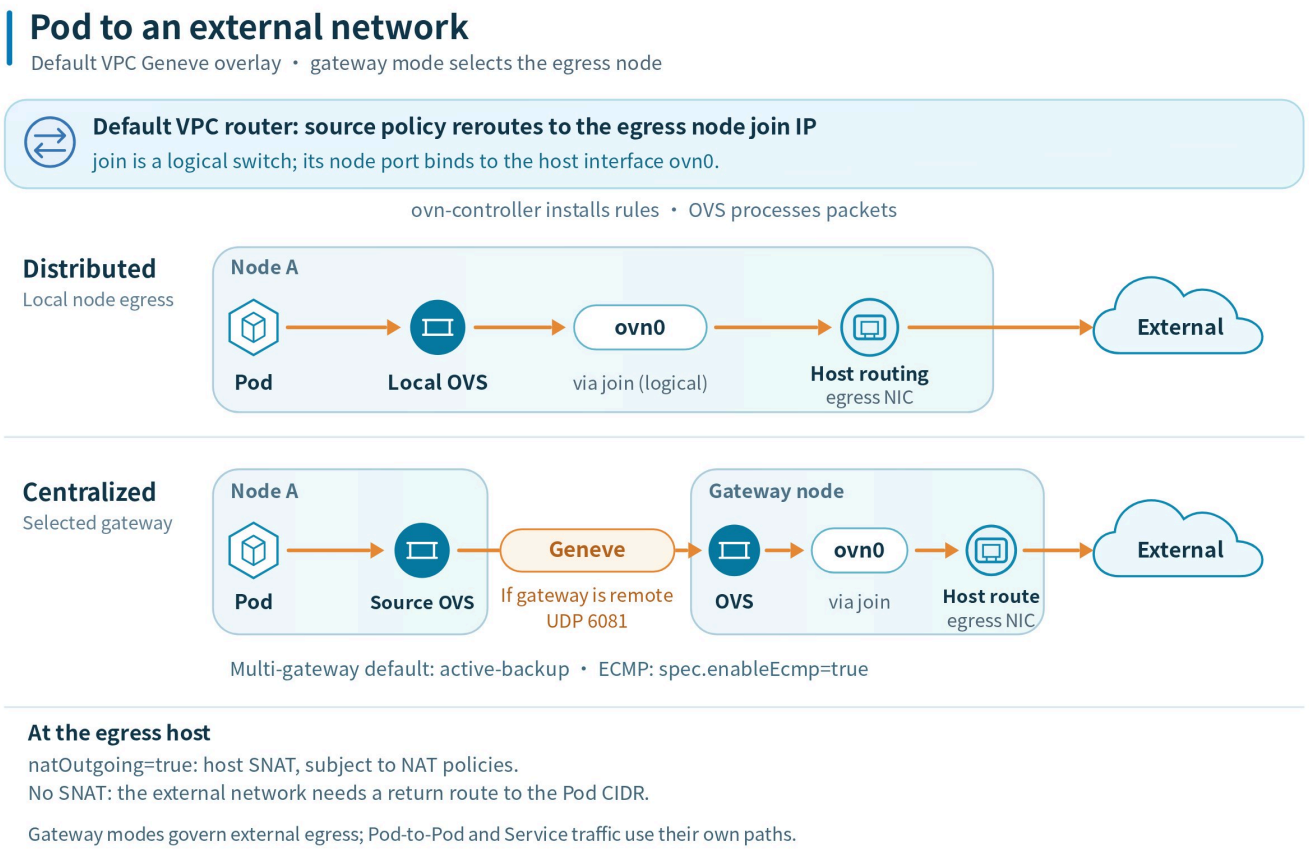
直接访问节点内部 IP 上的 API server 端点时，流量会遵循此节点访问路径。通过 `kubernetes` Service 虚拟 IP 访问时，首先会将 Service 地址转换为 API server 端点；之后的转发会遵循发往该端点的路由。通过外部 VIP 或负载均衡器暴露的 API 端点则遵循外部路由路径。

Pod 到外部网络

默认 VPC 逻辑路由器会使用源策略路由，选择出口节点在 `join` 交换机上的地址。数据包通过 `ovn0` 到达该节点，然后遵循主机的路由规则，经由出口 NIC 到达外部网络：

- 分布式网关：流量通过 Pod 所在的本地节点离开。

- 集中式网关：流量通过选定的网关节点离开。如果该网关位于远程节点，OVS 会通过 Geneve 将数据包传输到该节点。有关 active-backup 和 ECMP 行为，请参阅网关模式。



使用 `natOutgoing: true` 时，匹配的外部流量会在出口主机上进行源 NAT，具体取决于所配置的 NAT 策略。如果数据包离开出口主机时仍带有原始 Pod 源地址，则上游网络需要配置一条返回 Pod CIDR 的路由。有关网关配置，请参阅配置集中式网关。有关通过专用网关 Pod 为特定工作负载提供出口，请参阅出口网关及其与集中式网关的比较。

高可用性

在高可用部署中，`ovn-central` 使用 Raft 复制 northbound 和 southbound 数据库。每个数据库都需要其成员中的大多数可访问，才能提交更新。数据库可用性只是网络可用性的一部分：控制器、节点网络以及任何选定的网关也必须正常工作。

故障	对工作负载的影响
`kube-ovn-controller` 不可用	已有的转发规则可以继续使用，但在控制器可用前，无法协调新建的 Pod 网络、Service 端点或策略等变更。

故障	对工作负载的影响
OVN 数据库不可用	在节点数据平面和所需路径保持完整的情况下，已安装的规则可以继续转发流量。新的或变更后的网络配置无法完全应用。
节点 OVS 组件重启，或节点故障	经过该节点的流量可能中断，包括其 Pod 的流量以及该节点所承载的任何网关角色的流量。仅依靠数据库复制无法避免此类中断。
集中式网关节点故障	外部连接取决于配置的网关冗余和故障转移模式。故障转移期间连接可能中断；请参阅集中式网关行为。

当 `ovn-controller` 必须在重启后重新加载网络状态时，它需要访问 southbound 数据库，然后配置本地 OVS。`ovs-vswitchd` 不会直接读取 southbound 数据库。节点的接口和转发规则恢复期间，连接可能会中断。

检查集群中的组件

使用为目标集群配置的 `kubectl`，以及拥有读取这些资源权限的账号。以下示例输出来自运行 Kube-OVN `v1.16.13` 的三节点测试集群。副本数、Pod 名称、节点名称、IP 地址、运行时长和事件因集群而异；请将这些示例用于理解输出，而不要将其视为必须遵循的配置值。

以下长输出已缩减为下文讨论的关键字段和代表性行。在摘录中，`...` 表示省略的行；运行命令会返回完整输出。

列出工作负载

```
kubectl get deployments -n kube-system \
  ovn-central kube-ovn-controller kube-ovn-monitor
```

示例输出：

```
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
ovn-central           3/3     3            3           52m
kube-ovn-controller   3/3     3            3           52m
kube-ovn-monitor      1/1     1            1           52m
```

```
kubectl get daemonsets -n kube-system \
  ovs-ovn kube-ovn-cni kube-ovn-pinger
```

示例输出：

```
NAME              DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NO
DE SELECTOR             AGE
ovs-ovn           3         3         3       3            3           ku
bernetes.io/os=linux   52m
kube-ovn-cni      3         3         3       3            3           ku
bernetes.io/os=linux   52m
kube-ovn-pinger   3         3         3       3            3           ku
bernetes.io/os=linux   52m
```

将 Deployment 的 `READY` 列与其请求的副本数进行比较。对于每个 DaemonSet，请比较 `DESIRED`、`CURRENT` 和 `READY`；成功完成滚动更新后，符合条件的节点上的就绪数量应与期望数量一致。在此示例中，`ovn-central` 和 `kube-ovn-controller` 各有三个就绪副本，`kube-ovn-monitor` 有一个，且每个 DaemonSet 在全部三个符合条件的节点上都有一个就绪 Pod。

查找 **Pod** 和容器

列出 Pod，查看其实际调度位置和运行时状态：

```
kubectl get pods -n kube-system \
  -l 'app in (ovn-central,kube-ovn-controller,ovs,kube-ovn-cni,kube-ovn-p
inger,kube-ovn-monitor,kube-ovn-speaker)' \
  -o wide
```

示例输出（摘录；每个组件显示一个 Pod）：

```
NAME                                    READY   STATUS    RESTARTS   AGE
IP           NODE                     NOMINATED NODE   READINESS GATES
kube-ovn-cni-7ctr7                      1/1     Running   0          52m
10.64.0.33   ctyun-k8s3-20261014-c    <none>           <none>
...
kube-ovn-controller-78b79f69bf-6lcn4    1/1     Running   0          52m
10.64.0.73   ctyun-k8s3-20261014-a    <none>           <none>
...
kube-ovn-monitor-56d9f6cd47-47q5k       1/1     Running   0          52m
10.64.0.5    ctyun-k8s3-20261014-b    <none>           <none>
kube-ovn-pinger-j77zk                   1/1     Running   0          52m
10.16.0.5    ctyun-k8s3-20261014-b    <none>           <none>
...
ovn-central-6c784885b4-2flx8            1/1     Running   0          52m
10.64.0.73   ctyun-k8s3-20261014-a    <none>           <none>
...
ovs-ovn-bbgjn                           1/1     Running   0          52m
10.64.0.33   ctyun-k8s3-20261014-c    <none>           <none>
...
```

`NODE` 列显示每个 Pod 的运行位置。`READY`、`STATUS` 和 `RESTARTS` 有助于识别尚未就绪或正在重启的 Pod。`ovs-ovn` 的 Pod 选择器是 `app=ovs`；此列表中的其他组件使用其组件名称作为 `app` 的值。未安装的可选组件不会有匹配的 Pod；此示例集群未安装 `kube-ovn-speaker`。

例如，列出 OVN 和 OVS Pod 中的容器：

```
kubectl get pods -n kube-system -l 'app in (ovn-central,ovs)' \
  -o 'custom-columns=POD:.metadata.name,NODE:.spec.nodeName,CONTAINERS:.s
pec.containers[*].name'
```

示例输出（摘录；每个工作负载显示一个 Pod）：

```
POD                             NODE                     CONTAINERS
ovn-central-6c784885b4-2flx8    ctyun-k8s3-20261014-a    ovn-central
...
ovs-ovn-bbgjn                   ctyun-k8s3-20261014-c    openvswitch
...
```

对于标准部署，`CONTAINERS` 列会显示数据库 Pod 使用 `ovn-central`，OVS Pod 使用 `openvswitch`。NB、SB 和 `ovn-northd` 作为进程运行在 `ovn-central` 容器中；节点 OVS 数据库、`ovs-vswitchd` 和 `ovn-controller` 运行在 `openvswitch` 中。它们不是单独的 Kubernetes Pod 或容器。

检查滚动更新状态和事件

若要检查滚动更新详细信息和工作负载事件，请描述该工作负载。例如：

```
kubectl describe deployment -n kube-system ovn-central
```

示例输出（摘录）：

```
Name:                   ovn-central
Namespace:              kube-system
...
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0
unavailable
...
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
...
```

在此示例中，三个副本均可用，且 `Available` 和 `Progressing` 条件均为 `True`。

```
kubectl describe daemonset -n kube-system kube-ovn-cni
```

示例输出（摘录；省略重复的创建事件）：

```
Name:           kube-ovn-cni
Namespace:      kube-system
...
Desired Number of Nodes Scheduled: 3
Current Number of Nodes Scheduled: 3
...
Number of Nodes Scheduled with Available Pods: 3
Number of Nodes Misscheduled: 0
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
...
Events:
  Type    Reason            Age   From                  Message
  ----    ------            ----  ----                  -------
  Normal  SuccessfulCreate  52m   daemonset-controller  Created pod: kube
-ovn-cni-7ctr7
...
```

此处，期望、当前和可用节点数量均为三个，且没有 Pod 处于等待或失败状态。显示的 `SuccessfulCreate` 事件记录了一个 Pod 的创建。事件是临时的，因此较早创建的工作负载可能会显示 `Events: <none>`；请同时检查当前就绪状态和事件。

这些查询会显示部署和就绪状态。若要检查特定网络连接的行为，请参阅流量路径和相关配置指南。

资源使用量和限制

内存不足可能导致 `openvswitch` 容器处于 `OOMKilled` 状态；CPU 限流可能会延迟网络配置和数据包处理。出现这些症状时，请让集群管理员检查资源使用情况。

ACP Helm Chart 使用 `ovs-ovn.requests` 和 `ovs-ovn.limits` 设置资源。其升级模板会从现有的 `ovs-ovn` DaemonSet 读取 CPU 和内存值，因此管理员在更改或升级配置时，还必须核实该 DaemonSet 上实际生效的资源设置。

kubectl-ko

此 kubectl 插件可让集群管理员检查 OVN 数据库和节点 OVS 状态。它封装了 `ovn-nbctl`、`ovn-sbctl`、`ovs-vsctl` 和 `ovs-ofctl` 等命令。请使用适用于已安装 Kube-OVN 版本的插件版本，并确保拥有访问其组件所需的权限。

后续步骤

任务	指南
选择 Pod 网络并规划其地址范围	配置子网和配置 IPPool
控制哪些工作负载可以相互通信	网络策略
限制 Pod 的入口和出口带宽（QoS）	配置 Pod 网络带宽限制
提供稳定的应用访问地址	配置 Services
选择出站网关	配置集中式网关和配置 Egress Gateway
将 Pod 连接到物理网络	准备底层物理网络和配置底层子网
连接底层子网和覆盖网络子网	自动连接底层网络和覆盖网络
为 Pod 添加另一个网络接口	Pod 多网络接口
考虑隧道开销	配置 MTU

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

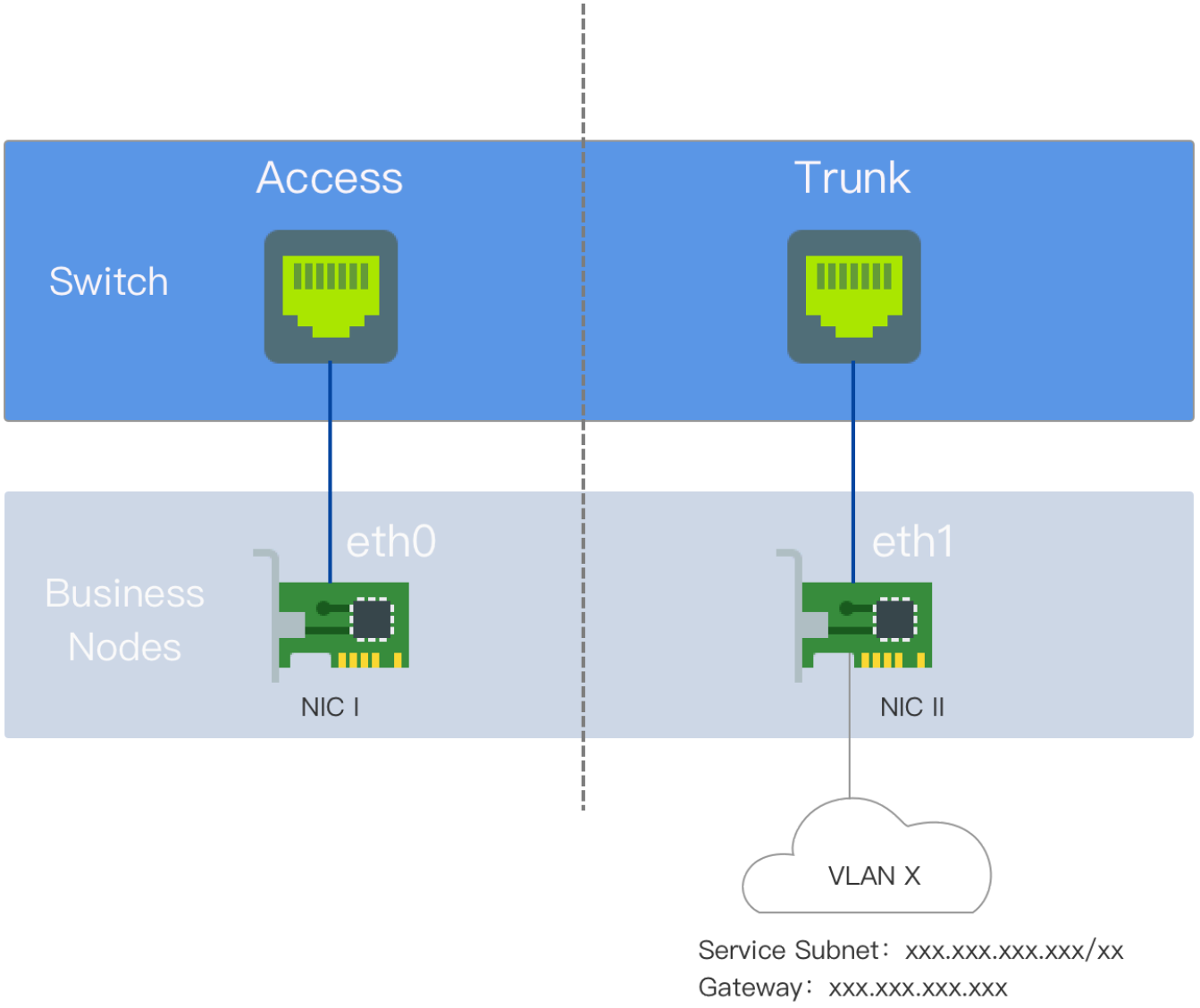
使用说明
术语说明
环境要求
配置示例
交换机配置
检查网络连通性
平台配置

使用说明

Kube-OVN Underlay 要求使用多块网络接口卡（NIC）进行部署，且 Underlay 子网必须专用一块 NIC。该 NIC 上不应承载 SSH 等其他类型的流量；这些流量应使用其他 NIC。

使用前，请确保节点服务器至少具备双 **NIC** 环境，并建议 NIC 速率至少为 **10 Gbps** 或更高（例如 10 Gbps、25 Gbps、40 Gbps）。

- NIC One：配置默认路由并设置 IP 地址的 NIC，与外部交换机接口互连，该接口设置为 Access 模式。
- NIC Two：未配置默认路由且未设置 IP 地址的 NIC，与外部交换机接口互连，该接口设置为 Trunk 模式。Underlay 子网专用 NIC Two。



术语说明

VLAN（Virtual Local Area Network，虚拟局域网）是一种将局域网逻辑划分为多个网段（或更小的 LAN），以便虚拟工作组进行数据交换的技术。

VLAN 技术的出现，使管理员能够根据实际应用需求，将同一物理局域网中的不同用户逻辑划分到不同的广播域中。每个 VLAN 由一组具有相似需求的计算机工作站组成，并具备与物理构建的 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 中的工作站不受限于同一物理区域；它们可以跨越不同的物理 LAN 网段存在。

VLAN 的主要优势包括：

- 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。
- 网络安全性。不同 VLAN 无法直接通信，从而消除了广播信息带来的不安全因素。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、简化网络管理并提高网络安全性。
- 灵活管理。更改用户的网络归属时，无需更换端口或网线；只需修改软件配置即可。

环境要求

在 Underlay 模式下，Kube-OVN 将物理 NIC 桥接到 OVS，并通过该物理 NIC 直接向外部发送数据包。L2/L3 转发能力依赖底层网络设备。相应的网关、VLAN 和安全策略需要预先在底层网络设备上配置。

- 网络配置要求
  - Kube-OVN 在启动容器时通过 ICMP 协议检查网关的连通性；底层网关必须响应 ICMP 请求。
  - 对于服务访问流量，Pod 会先将数据包发送到网关，网关必须具备将数据包转发回本地子网的能力。
  - 当交换机或网桥启用了 Hairpin 功能时，必须禁用 **Hairpin**。如果使用 VMware 虚拟机环境，请将 VMware 主机上的 **Net.ReversePathFwdCheckPromisc** 设置为 **1**，此时无需禁用 Hairpin。
  - 桥接 NIC 不能是 **Linux Bridge**。
  - NIC bonding 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 bonding 模式尚未经过测试，请谨慎使用。
  - 如果工作负载需要 IPv4 多播通信，则物理网络必须允许所需的 IPv4 多播组和 IGMP 流量。有关配置详情，请参阅为 Kube-OVN Underlay 网络配置 IPv4 多播。
- **IaaS**（虚拟化）层配置要求
  - 对于 OpenStack VM 环境，需要禁用相应网络端口的 **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，由于不具备用户自定义 MAC 地址的能力，因此不支持 Underlay 模式网络。

配置示例

本示例中的节点是双 NIC 物理机。NIC One 是配置默认路由的 NIC；NIC Two 是未配置默认路由且未设置 IP 地址的 NIC，专用于 Underlay 子网。NIC Two 与外部交换机互连。

- 在交换机侧，与 NIC Two 连接的接口应配置为 Trunk 模式，以允许相应的 VLAN 通过。
- 在相应的 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之后，则必须允许从节点到 cluster-cidr 网络的访问。
- 无需配置服务器 NIC。

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 NIC Two 连接的接口：

```
#
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 thro
ugh
#
```

检查网络连通性

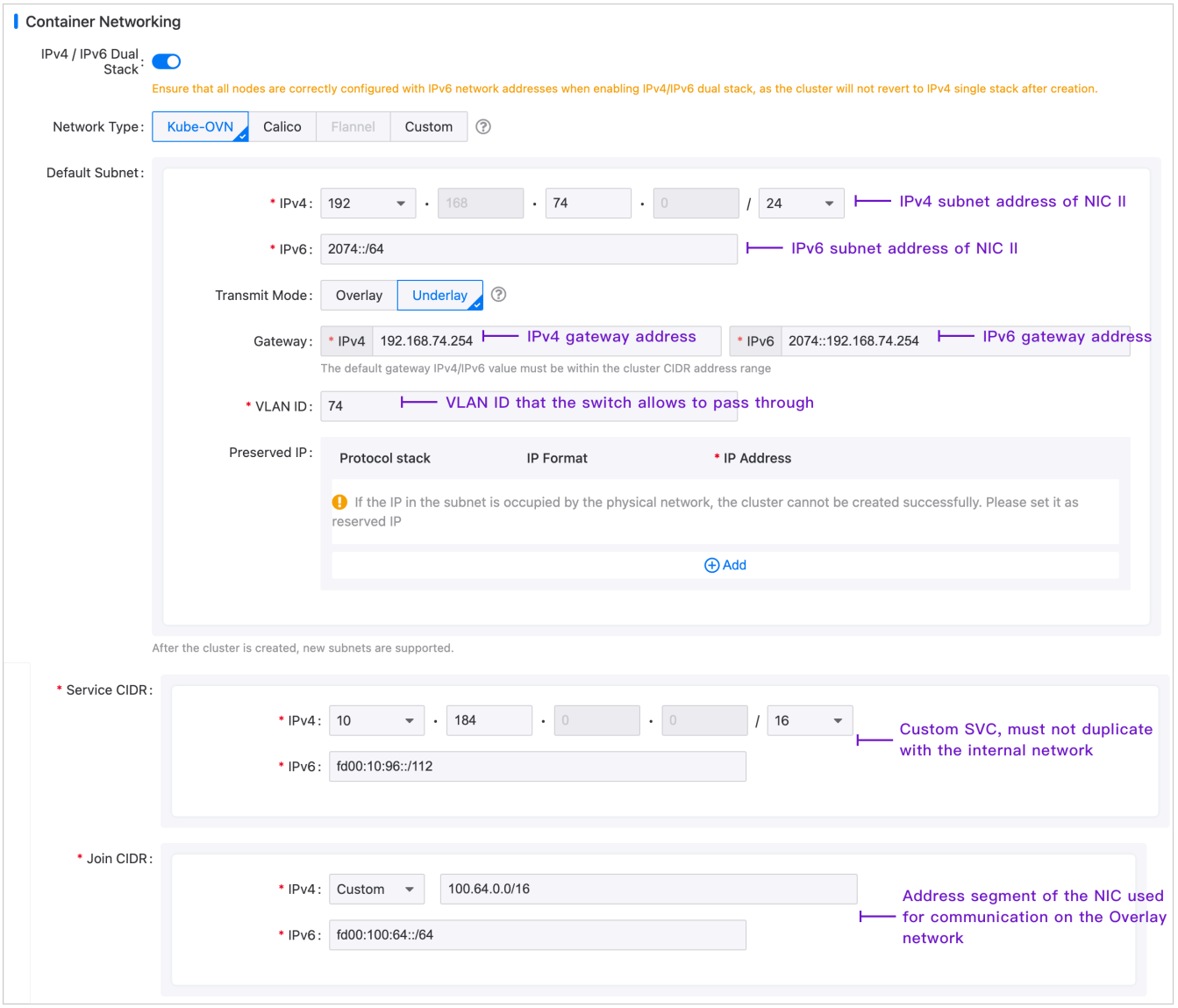
测试 NIC Two 是否可以与网关地址通信：

```
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，单击集群管理 **>** 集群，然后单击创建集群。有关集群创建路径选择，请参阅集群概览，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，主要用于之后创建 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
IPv4 subnet address of NIC II
* IPv6: 2074::/64
IPv6 subnet address of NIC II
Transmit Mode: Overlay Underlay
Gateway: * IPv4 192.168.74.254
IPv4 gateway address
* IPv6 2074::192.168.74.254
IPv6 gateway address
The default gateway IPv4/IPv6 value must be within the cluster CIDR address range
* VLAN ID: 74
VLAN ID that the switch allows to pass through
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
After the cluster is created, new subnets are supported.
* Service CIDR:
* IPv4: 10 . 184 . 0 . 0 / 16
Custom SVC, must not duplicate with the internal network
* IPv6: fd00:10:96::/112
* Join CIDR:
* IPv4: Custom 100.64.0.0/16
Address segment of the NIC used for communication on the Overlay network
* IPv6: fd00:100:64::/64

Underlay 和 Overlay 子网自动互连

如果集群同时具有 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 子网之间的自动互连，您可以手动修改 Underlay 子网的 YAML 文件。配置完成后，Kube-OVN 还将使用额外的 Underlay IP 连接 Underlay 子网和 ovn-cluster 逻辑路由器，并设置相应的路由规则以启用互连。

目录

操作步骤

可选：将 U2O 路由限制为 Overlay CIDR

启用 u2oInterconnection 的 Underlay 子网之间的隔离

步骤 1：配置 kube-ovn-controller

步骤 2：配置子网隔离

操作步骤

1. 前往 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** 以筛选资源对象。

4. 点击要修改的 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：需要重新创建 Underlay 子网中现有的计算组件，才能使更改生效。

可选：将 U2O 路由限制为 Overlay CIDR

若只希望将发往 Overlay 子网 CIDR 的流量通过 U2O 网关路由，请在同一 Underlay 子网上启用 `u2oFeatures.overlayOnlyRouting`。发往其他目标的流量将继续使用物理网关。

```
spec:
  u2oInterconnection: true
  u2oFeatures:
    overlayOnlyRouting: true
```

此选项要求 `u2oInterconnection: true`。

启用 u2oInterconnection 的 Underlay 子网之间的隔离

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

如果需要隔离两个都启用 `u2oInterconnection` 的 Underlay 子网，必须先配置 kube-ovn-controller 参数，然后配置子网隔离。

步骤 1：配置 kube-ovn-controller

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

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

添加或修改以下参数：

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

CAUTION

`--ls-ct-skip-dst-lport-ips` 控制是否跳过发往逻辑端口 IP 的流量的连接跟踪（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，但可能会对性能造成轻微影响。

但是，对于使用基于 ACL 的隔离的 Underlay 子网，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`，这需要 conntrack 状态；如果未启用，系统无法将回复识别为“相关”流量，因此会将其丢弃。

步骤 2：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 logica
l 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
`action`	要执行的操作：`allow`、`drop` 或 `allow-related`
`direction`	流量方向：`to-lport`（入口）或 `from-lport`（出口）
`match`	使用 L2-L4 字段和布尔运算符的 OVN 匹配表达式
`priority`	规则优先级（值越高越先评估；建议范围：1002-1899）

NOTE

- `acls` 字段提供基于优先级的规则评估，比标准 Kubernetes NetworkPolicy 更加灵活。
- 使用 `to-lport` 方向时，`ip4.src` 指传入流量的源 IP。
- 建议的优先级范围：`1002` 到 `1899`，以避免与系统默认 ACL 规则冲突。

Kube-OVN Underlay + MetalLB 负载均衡服务配置

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. 创建示例应用和负载均衡服务
6. 验证配置
7. 迁移现有服务

商标声明

概述

此方案解决了 MetalLB L2 模式与 Kube-OVN Underlay 网络的集成问题。用户可以将 Underlay 子网 IP 用作 MetalLB 负载均衡服务 VIP，并将流量直接转发到后端业务 Pod。

> ⚠️ 关键：负载均衡 VIP 和后端 Pod IP 必须位于同一 **Underlay** 子网中。

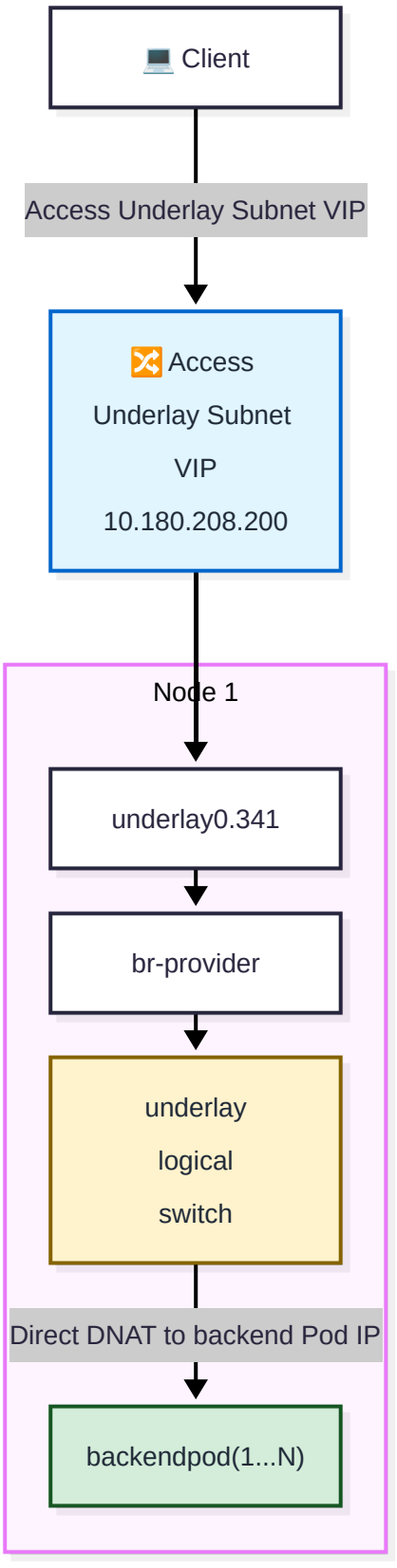
前提条件

环境要求

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

流量流向

流量示意图：


💻 Client
Access Underlay Subnet VIP
🔀 Access
Underlay Subnet
VIP
10.180.208.200
Node 1
underlay0.341
br-provider
underlay
logical
switch
Direct DNAT to backend Pod IP
backendpod(1...N)

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` 指定了每个节点的接口，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`）。在 `customInterfaces` 中保留父接口会阻止 MetalLB 负载均衡流量到达后端 Pod。

> ⚠️ 警告：单独修改 `ProviderNetwork` 或 `Vlan` 资源时，Underlay 网络连接将会中断。只有在这两个资源均完成配置并保持同步后，网络连接才会恢复。请在维护窗口期间计划配置更改，以尽量减少服务中断。

2. 配置 Kube-OVN Controller 参数

为 Kube-OVN Controller 配置负载均衡功能所需的参数：

使用 **Web Console**：

1. 导航至 管理员 > **Marketplace** > 集群插件，然后搜索 `ovn` 以找到 **Alauda Container Platform Networking**
2. 在插件行中，点击操作菜单（垂直⋮），然后选择 更新 以打开配置对话框
3. 配置以下设置：
   - 跳过目标 **LPort IP** 的 **CT**：否
   - 启用 **OVN LB Local**：是

3. 配置 Underlay 子网外部地址功能

编辑 Underlay 子网，为负载均衡使用预留 IP 范围：

> 重要：外部地址池 IP 必须位于 Underlay 子网内。

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

```yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: underlay-subnet
spec:
  enableExternalLBAddress: true       # Indicates this subnet has IP rang
e for LB service VIP
  excludeIps:
  - 10.180.208.200..10.180.208.220    # Reserve IP range for external add
ress pool
```

4. 创建 MetalLB 外部地址池

```yaml
# underlay-ippool.yaml
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: acp-underlay-pool
  namespace: metallb-system
spec:
  addresses:
    - 10.180.208.200-10.180.208.220  # Underlay 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. 创建示例应用和负载均衡服务

```yaml
# application-with-loadbalancer.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-app
  labels:
    app: backend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
          image: nginx:1.25
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: backend-lb-service
  # 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. 迁移现有服务

对于使用旧地址池（仅节点子网）的现有服务，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
```

对于新服务，直接添加注解：

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

```bash
# Check service status
kubectl get svc backend-lb-service -o wide

# Test external access
curl http://10.180.208.200
```

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是一家独立供应商。本产品与 The Linux Foundation 或 F5, Inc. 无关联，也未获得其认可或赞助。所有商标均为其各自所有者的财产，此处仅用于标识。

集群互联（Alpha）

支持在网络模式相同且均为 Kube-OVN 的集群之间配置集群互联，以实现集群中 Pod 的相互访问。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
已构建多节点 Kube-OVN 互联控制器
部署 Deployment
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。
- 需要准备一组可由各集群的 kube-ovn-controller 通过 IP 访问的机器，用于部署跨集群互联的控制器。
- 每个集群都需要准备一组可由该集群的 kube-ovn-controller 通过 IP 访问的机器，后续将这些机器用作跨集群互联的网关节点。
- 此功能仅适用于默认 VPC，用户自定义 VPC 无法使用互联功能。

已构建多节点 Kube-OVN 互联控制器

有三种可用的部署方式：部署 Deployment（平台 v3.16.0 及更高版本支持）、Podman 部署和 Containerd 部署。

部署 Deployment

注意：平台 v3.16.0 及更高版本支持此部署方式。

操作步骤

1. 在集群的一个控制平面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. 打开当前目录中的脚本文件，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。

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

Podman 和 Containerd 部署

1. 选择任意集群中的三个或更多节点来部署 Interconnected Controller。本例准备了三个节点。
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，请检查 `/etc` 下是否存在 ovn 目录。如果不存在，请使用命令 `mkdir /etc/ovn` 创建该目录。

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

     - Leader 节点的命令：

       ```
       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="192.168.39.37"
       -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
       ```

     - 其他两个节点的命令：

```
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="192.168.39.22,192.168.39.24,192.168.39.37"
<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
```

- **Containerd** 部署命令

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

  - Leader 节点的命令：

```
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=rbin
d:rw" \
--mount="type=bind,src=/var/log/ovn,dst=/var/log/ovn,options=rbin
d:rw" \
--env="NODE_IPS=<IP addresses of all nodes, separated by commas>"
\   ## For example: --env="NODE_IPS="192.168.178.97,192.168.181.9
3,192.168.177.192""
--env="LOCAL_IP=<IP address of the current node>" \   ## For examp
le: --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 b
ash start-ic-db.sh
```

- 其他两个节点的命令：

```
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=rbin
d:rw" \
--mount="type=bind,src=/var/log/ovn,dst=/var/log/ovn,options=rbin
d:rw" \
--env="NODE_IPS=<IP addresses of all nodes, separated by commas>"
\   ## For example: --env="NODE_IPS="192.168.178.97,192.168.181.9
3,192.168.177.192"" \
--env="LOCAL_IP=<IP address of the current node>" \   ## For examp
le: --env="LOCAL_IP="192.168.181.93""
--env="LEADER_IP=<IP address of the Leader node>" \   ## For examp
le: --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 b
ash start-ic-db.sh
```

在 **Global** 集群中部署集群互联控制器

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

注意：为确保正常运行，不允许修改 global 上名为 ovn-ic 的 ConfigMap。如果需要更改任何参数，请删除该 ConfigMap，正确重新配置后再应用 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

集群中的已创建子网、**ovn-default** 和加入子网不得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，单击集群 > 集群的集群。
2. 单击要加入集群互联的*集群*名称。
3. 在右上角单击选项 > 集群互联。

4. 单击加入集群互联。
5. 为集群选择一个网关节点。
6. 单击加入。

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，单击集群 > 集群的集群。
2. 单击要更新网关节点信息的*集群名称*。
3. 在右上角单击操作 > 集群互联。
4. 对于要更新网关节点信息的集群，单击更新网关节点。
5. 重新为集群选择网关节点。
6. 单击更新。

退出集群互联

已加入集群互联组的集群退出集群互联。退出后，该集群的 Pod 将与外部集群的 Pod 断开连接。

操作步骤

1. 在左侧导航栏中，单击集群 > 集群的集群。
2. 单击要停用的*集群*名称。
3. 在右上角单击选项 > 集群互联。
4. 对于要退出的集群，单击退出集群互联。
5. 正确输入集群名称。
6. 单击退出。

清理互联集群残留

如果删除集群时未退出集群互联，控制器上可能会残留一些数据。当您尝试使用这些节点再次创建集群并加入集群互联时，可能会失败。您可以在控制器（kube-ovn-controller）的 `/var/log/ovn/ovn-ic.log` 日志中查看详细的错误信息。部分错误消息可能包括：

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

操作步骤

1. 对要加入的集群退出互联集群。
2. 在容器或 Pod 中执行清理脚本。
   您可以直接在 ovn-ic-db 容器或 ovn-ic-controller Pod 中执行清理脚本。请选择以下任一方法：
   方法 **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：
     - 使用原始集群的名称执行清理操作。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>
       ```

   方法 **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：

- 使用原始集群的名称执行清理操作。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 至步骤 3 需要在所有已加入互联集群的业务集群上执行。

操作步骤

1. 退出互联集群。具体有两种退出方法， 请根据需要选择其中一种。
   - 删除业务集群中名为 ovn-ic-config 的 ConfigMap。使用以下命令。

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

   - 通过平台操作退出互联集群。
2. 使用以下命令进入 ovn-central 的 Leader Pod。

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

3. 使用以下命令清理 ts 逻辑交换机。

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

4. 登录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

加入集群互联后，如需将集群网关配置为高可用，请执行以下步骤：

1. 登录需要转换为高可用网关的集群，并执行以下命令，将 `enable-ic` 字段更改为 `false`。
   注意：将 `enable-ic` 字段更改为 `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. 更新 `gw-nodes` 字段来修改网关节点配置，并使用英文逗号分隔网关节点；同时将 `enable-ic` 字段更改为 `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. 进入集群 ovn-central 中的 Pod，并执行 `ovn-nbctl lrp-get-gateway-chassis {current cluster name}-ts` 命令，验证配置是否已生效。

```
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：准备外部网络附加配置

步骤 2：创建 VPC Egress Gateway

步骤 3：验证 Gateway

1. 检查资源状态

2. 检查 gateway 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 的高可用性

1. 在 VPC 上启用 BFD Port

2. 在 VPC Egress Gateway 上启用 BFD

3. 验证 BFD 状态

已验证容量指南

可能中断流量的操作

其他资源

概述

Egress Gateway 也称为 VPC Egress Gateway，可为 overlay 网络中的 Pod 提供稳定的出站地址。 它会在流量离开集群前，将选定的工作负载通过专用 gateway Pod 进行路由。

在以下场景中可以使用 Egress Gateway：

- 为特定工作负载提供稳定的源 IP 地址
- 实现工作负载级别的出站控制，而不是子网级别的控制
- 通过水平扩展提高吞吐量
- 加快出站流量的故障转移

主要功能：

- 通过 ECMP 实现 Active-Active 高可用，并支持水平扩展吞吐量
- 通过 BFD 实现快速故障转移，通常耗时不到 1 秒
- 支持 IPv4、IPv6 和双栈环境
- 通过 NamespaceSelector 和 PodSelector 实现细粒度流量匹配
- 通过节点选择器和容忍度实现灵活调度

当前限制：

- 多副本部署需要多个出站 IP
- 不会保留源 NAT 映射记录

Egress Gateway 与 Centralized Gateway 的对比

请将 Egress Gateway 与 Centralized Gateway 进行比较，以选择最适合您场景的 gateway 模式。 有关集中式模式的详细信息，请参阅配置 Centralized Gateway。

维度	Egress Gateway	Centralized Gateway
粒度	通过 `selectors` 和 `policies`（Namespace/Pod/Subnet/IPBlock）进行细粒度控制。	在子网级别应用（`gatewayType: centralized`）。
出站地址来源	使用具有外部子网 IP 的专用 gateway Pod。	使用指定的 gateway 节点；使用 `natOutgoing: true` 时，出站流量使用节点 IP。
数据路径	流量被路由到 VPC Egress Gateway Pod，然后通过 SNAT 转换到外部网络。	流量被路由到指定节点（`ovn0`），然后由主机路由/NAT 规则转发。
高可用与故障转移	Active-Active ECMP；支持通过 BFD 实现快速故障转移（亚秒级）。	支持 ECMP 和主备模式；故障转移通常比基于 BFD 的 Egress Gateway 更慢。
调度控制	支持对 gateway 工作负载进行节点调度控制（`nodeSelector`、`tolerations`）。	Gateway 节点由 `gatewayNode` 或 `gatewayNodeSelectors` 选择。
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转移。	更简单的子网级固定源出站、审计、IP 允许列表以及防火墙源 IP 管理。

选择建议：

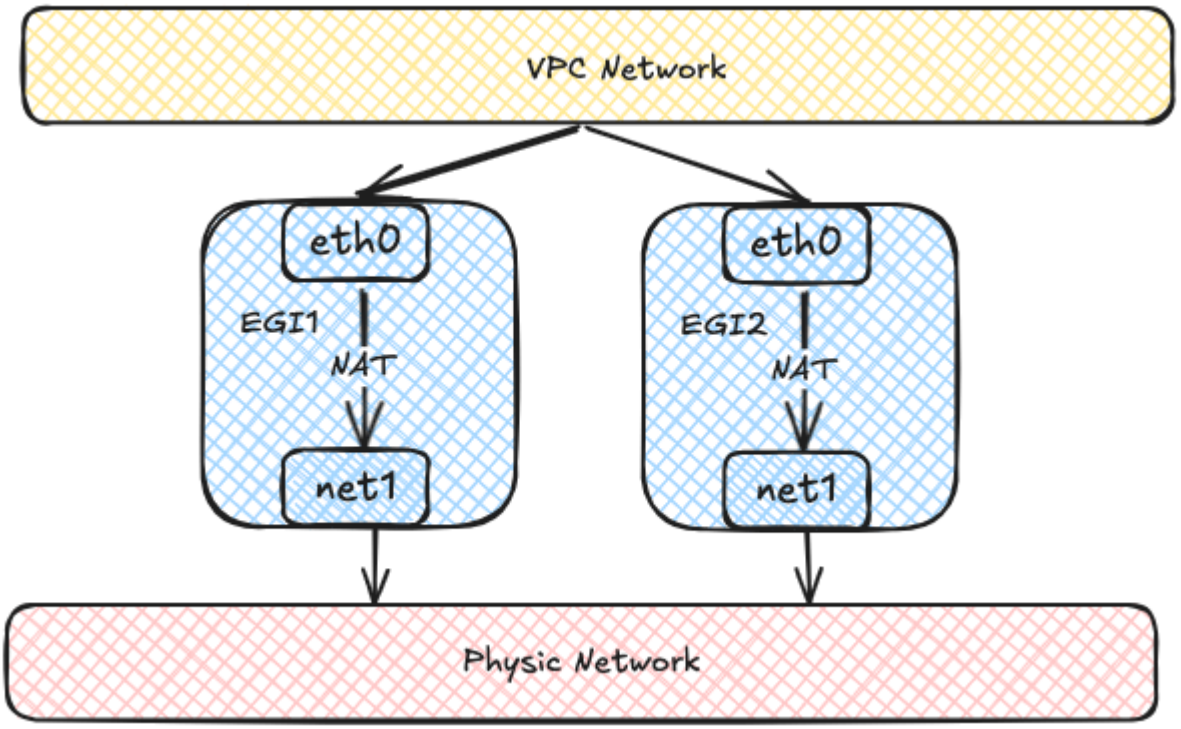
- 如果需要工作负载级别的策略控制、可扩展的吞吐量和快速故障转移，请选择 **Egress Gateway**。
- 如果子网级别的固定出站和简单运维即可满足需求，请选择 **Centralized Gateway**。

Egress Gateway 的工作原理

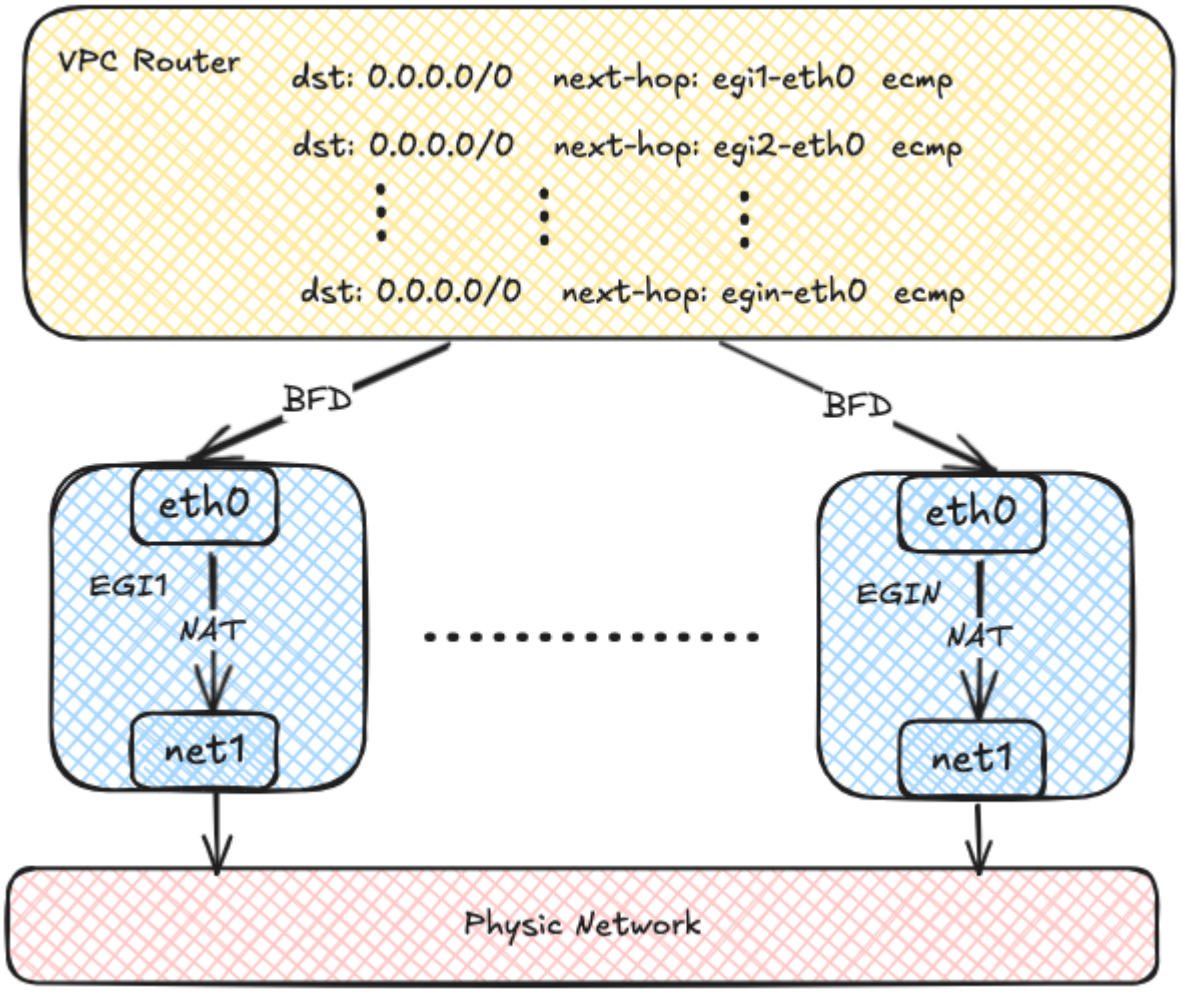
Egress Gateway 以一个或多个 Pod 的形式运行。每个 Pod 使用两个网络接口：

- 一个接口连接到集群内部的 overlay 网络。
- 另一个接口连接到外部 underlay 网络。

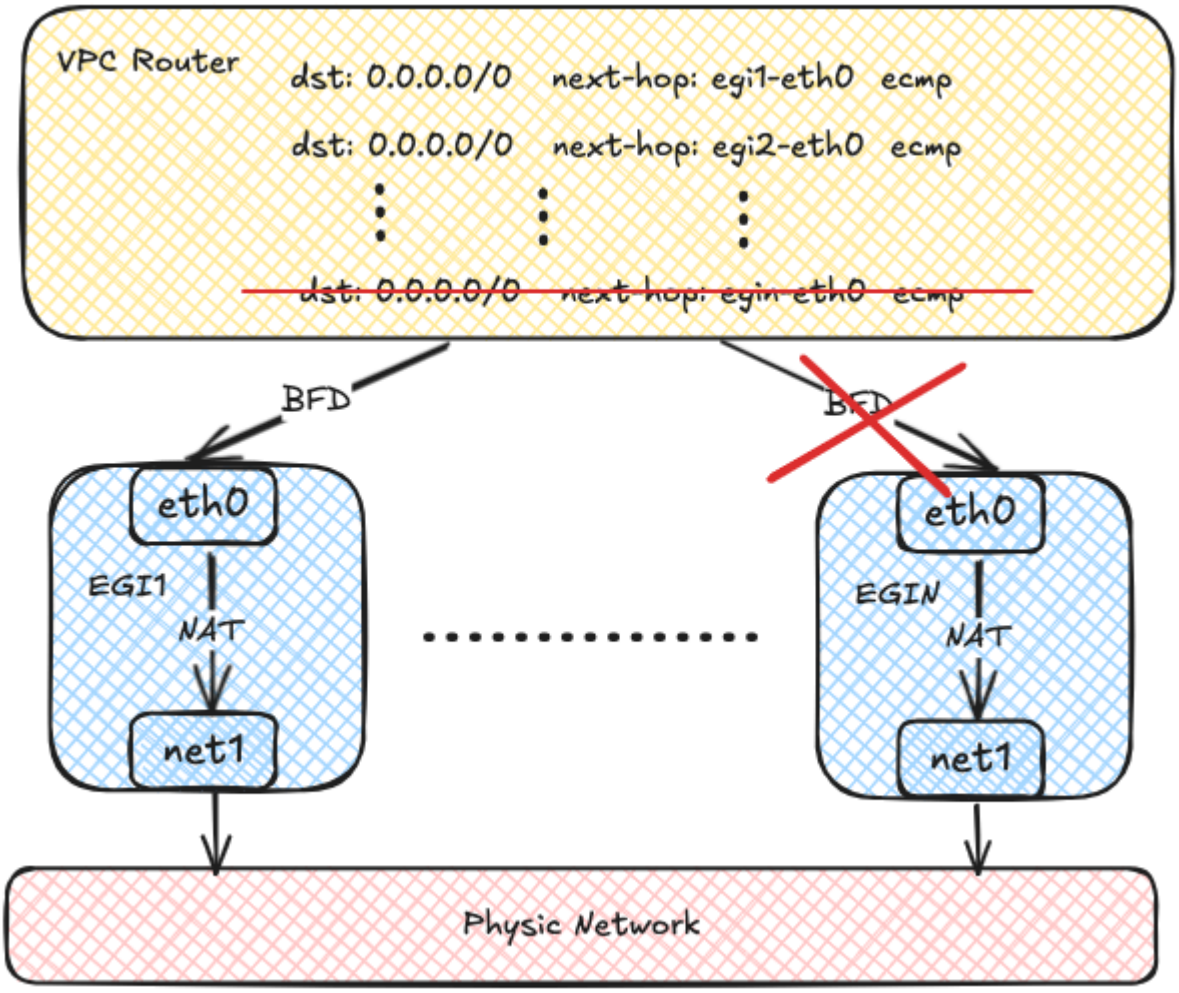
来自选定 Pod 的流量首先转发到 gateway Pod，然后通过 underlay 接口发送到外部网络。



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 使用源地址哈希将流量分发到多个 gateway 实例。 这不仅可以实现负载均衡，还能在添加更多实例时横向扩展吞吐量。



OVN 可以使用 BFD 探测多个 gateway 实例。 如果某个实例发生故障，OVN 会将相应路由标记为不可用，并快速将流量重定向到正常运行的实例。



开始之前

开始之前，请确保满足以下前提条件：

- *Multus CNI Plugin* 必须已安装在集群中。
- 外部 underlay 网络、VLAN 和 bridge 网络已完成规划并可用。
- 您已确定哪些工作负载应使用该 gateway，以及它们是否需要 SNAT。

如需安装 *Multus CNI Plugin*，请参阅部署 Multus CNI Plugin。

配置流程

请按以下顺序配置 Egress Gateway：

1. 准备外部网络附加配置，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：扩展副本数量并启用基于 BFD 的快速故障转移。

步骤 1：准备外部网络附加配置

Egress Gateway 使用多个接口连接内部和外部网络。 创建 gateway 之前，请准备以下资源：

- 一个外部子网
- 用于该子网的 Network Attachment Definition (NAD)

> **NOTE**
>
> NetworkAttachmentDefinition 名称不得包含点号（`.`）。 每个 NetworkAttachmentDefinition 只能由一个子网使用。请勿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**
>
> 此示例假设已为 underlay 网络创建 bridge 网络和 VLAN 资源 `external-vlan`。

```yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: underlay-ext
  namespace: default
spec:
  config: |-
    {
      "cniVersion": "0.3.0", 1
      "type": "kube-ovn", 2
      "server_socket": "/run/openvswitch/kube-ovn-daemon.sock", 3
      "provider": "underlay-ext.default.ovn"
    }
---
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: underlay-ext
spec:
  protocol: IPv4
  provider: underlay-ext.default.ovn 4
  cidrBlock: 172.17.0.0/16 5
  gateway: 172.17.0.1 6
  vlan: external-vlan 7
  excludeIps: 8
    - 172.17.0.11..172.17.0.20
```

1 为辅助网络使用 Kube-OVN CNI plugin。

2 CNI plugin 使用的 Kube-OVN daemon socket。

3 格式为 `<network attachment definition name>.<namespace>.ovn` 的 provider 名称。

4 子网使用的 provider。此值必须与 NetworkAttachmentDefinition 中的 provider 匹配。

5 外部 underlay 网络的 CIDR。

6 外部 underlay 网络的 gateway。

7 underlay 子网使用的 VLAN 资源。

8 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 Subnet 自定义资源 (CR) 示例。

TIP

使用 underlay 子网之前，请确保已正确准备物理网络、VLAN 和 bridge 网络。 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，并定义应使用该资源的工作负载。

```yaml
apiVersion: kubeovn.io/v1
kind: VpcEgressGateway
metadata:
  name: gateway1
  namespace: default (1)
spec:
  replicas: 1 (2)
  internalSubnet: ovn-default (3)
  externalSubnet: underlay-ext (4)
  externalIPs: (5)
    - 172.17.0.11
    - 172.17.0.12
  resources: (6)
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 200m
      memory: 256Mi
      ephemeral-storage: 2Gi
  nodeSelector: (7)
    - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
            - node1
            - node2
  tolerations: (8)
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
  selectors: (9)
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: ns1
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: ns2
      podSelector:
        matchLabels:
          app: myapp
  policies: (10)
    - snat: true (11)
```

```
    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 实例的数量。

3 连接内部网络的内部子网。该子网必须是同一 VPC 中的 overlay 子网，并且必须有足够的可用 IP 为 gateway 实例分配。 如果未指定，gateway Pod 将使用 VPC 的默认内部子网。

4 连接外部网络的外部子网。

5 gateway Pod 在 underlay 网络上使用的外部 IP。每个 gateway 实例都会从此列表中分配一个 IP。 这些 IP 必须位于外部子网的 CIDR 范围内，并且应包含在该子网的 `excludeIps` 范围内。 建议预留 *.spec.replicas + 1* 个 IP，以便 gateway Pod 在边缘情况下仍能获取 IP。

6 每个 VPC Egress Gateway 实例的资源请求和限制。 如果未指定，将应用 VPC Egress Gateway controller 中定义的默认资源请求和限制。

7 用于调度 VPC Egress Gateway 实例的节点选择器。

8 用于调度 VPC Egress Gateway 实例的容忍度。

9 用于选择通过 VPC Egress Gateway 访问外部网络的 Pod 的 Namespace 选择器和 Pod 选择器。

10 VPC Egress Gateway 的策略，包括要应用的 SNAT 和子网/ipBlocks。

11 是否为该策略启用 SNAT。

12 策略适用的子网。

13 策略适用的 IP block。

此示例在 `default` Namespace 中创建名为 *gateway1* 的 VPC Egress Gateway。 匹配选择器和策略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
- *ns2* Namespace 中带有标签 `app: myapp` 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 始终由 gateway 执行 SNAT。

步骤 3：验证 Gateway

创建 gateway 后，请确认其已就绪，并且正在按预期转发流量。

1. 检查资源状态

首先检查基本资源状态：

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

然后检查详细的 gateway 信息：

```
kubectl get veg gateway1 -o wide
NAME       VPC           REPLICAS   BFD ENABLED   EXTERNAL SUBNET   PHASE
READY   INTERNAL IPS     EXTERNAL IPS      WORKING NODES   AGE
gateway1   ovn-cluster   1          false         underlay-ext      Compl
eted   true    ["10.16.0.12"]   ["172.17.0.11"]   ["node1"]       82s
```

最后，确认 gateway 工作负载正在运行：

```
$ kubectl get deployment -n default -l ovn.kubernetes.io/vpc-egress-gatew
ay=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=gate
way1 -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP
NODE    NOMINATED NODE   READINESS GATES
gateway1-b9f8b4448-76lhm   1/1     Running   0          4m48s   10.16.0.1
2   node1   <none>           <none>
```

2. 检查 **gateway Pod** 内部的网络

检查 gateway Pod 内部的 IP 地址、路由条目和 iptables 规则：

```
$ kubectl exec -n default gateway1-b9f8b4448-76lhm -c gateway -- ip addre
ss 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 gateway Pod 中抓取数据包，以确认流量正在通过 gateway 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), snapsh
ot length 262144 bytes
06:50:58.936528 eth0  In  ifindex 17 92:26:b8:9e:f2:1c ethertype IPv4 (0x
0800), length 104: (tos 0x0, ttl 63, id 30481, offset 0, flags [DF], prot
o 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 (0x0
800), length 104: (tos 0x0, ttl 62, id 30481, offset 0, flags [DF], proto
ICMP (1), length 84)
    172.17.0.11 > 172.17.0.1: ICMP echo request, id 39449, seq 0, length
64
06:50:58.936613 net1  In  ifindex 2 02:42:39:79:7f:08 ethertype IPv4 (0x0
800), length 104: (tos 0x0, ttl 64, id 26701, offset 0, flags [none], pro
to 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 (0x
0800), length 104: (tos 0x0, ttl 63, id 26701, offset 0, flags [none], pr
oto 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

OVN 会自动为选定流量创建 Logical Router 策略：

```
$ kubectl ko nbctl lr-policy-list ovn-cluster
Routing Policies
     31000                             ip4.dst == 10.16.0.0/16   allow
     31000                             ip4.dst == 10.17.0.0/16   allow
     31000                           ip4.dst == 100.64.0.0/16   allow
     30000                              ip4.dst == 172.18.0.2  reroute  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 用于转发匹配 *.spec.policies* 的流量的 Logical Router Policy。

2 VPC Egress Gateway 用于转发匹配 *.spec.selectors* 的流量的 Logical Router Policy。

可选：启用多副本负载均衡

> **NOTE**
>
> 扩展副本数量之前，请确保在外部子网中准备足够的外部 IP，并将其指定在 *.spec.externalIPs* 中。

要启用 ECMP 负载均衡并横向扩展吞吐量，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
READY   AGE
gateway1   ovn-cluster   2          false         underlay-ext      Compl
eted   true    39m

$ kubectl get pod -n default -l ovn.kubernetes.io/vpc-egress-gateway=gate
way1 -o wide
NAME                        READY   STATUS    RESTARTS   AGE   IP
NODE    NOMINATED NODE   READINESS GATES
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

要限制选定 Namespace 或工作负载的带宽，请设置 *.spec.bandwidth.ingress* 和 *.spec.bandwidth.egress*。 每个字段接受以下格式之一：

- 非负整数或数字字符串，按 Mbps 解释。例如，`2000` 和 `"2000"` 均表示 2000 Mbps。
- 带有 `M`、`Mi`、`G` 或 `Gi` 后缀的 Kubernetes 数量字符串，按每秒 bit 数解释。存在后缀时支持十进制值。该值会向上取整到最接近的整数 Mbps。例如，`1Gi` 会变为 1074 Mbps。

省略字段或将其设置为 `0`，即可不限制相应方向的带宽。限制应用于每个 gateway 实例。

> **NOTE**
>
> 如果 *.spec.replicas* 大于 1，则每个副本都会获得配置的限制。 数量字符串需要使用支持该格式的 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。

以下示例将选定 Namespace 的 ingress 带宽限制为 2 Gbps，将 egress 带宽限制为 4 Gbps：

```yaml
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，方向是相对于选定 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** 的高可用性

基于 BFD 的故障转移依赖于 VPC BFD LRP。 请按以下顺序启用它。

1. 在 **VPC** 上启用 **BFD Port**

首先，在 VPC 上启用 BFD Port：

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

(1) 是否启用 BFD Port。

(2) BFD Port 的 IP 地址，该地址必须是有效 IP 地址，且不得与任何其他 IP/子网冲突。

(3) 用于选择在 Active-Backup 模式下运行 BFD Port 的节点的节点选择器。

> **TIP**
>
> Vpc 资源 *ovn-cluster* 默认存在。您可以直接编辑该资源以启用 BFD Port。

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

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

(1) 在 OVN Logical Router 上创建的 BFD Port。

2. 在 VPC Egress Gateway 上启用 BFD

然后，将 *.spec.bfd.enabled* 设置为 *true*，在 VPC Egress Gateway 上启用 BFD：

```yaml
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 实例连接的内部子网。

(3) Egress Gateway 实例连接的外部子网。

(4) 分配给 Egress Gateway 实例的外部 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 的乘数，用于确定在声明故障前允许丢失的数据包数量。

此示例创建一个名为 *gateway2*、具有两个副本并启用 BFD 的 VPC Egress Gateway。 如果某个实例发生故障，BFD 会话将断开，OVN 会将路由标记为不可用，并将流量重定向到正常运行的实例。

故障转移检测时间取决于 BFD 设置。 请使用以下公式：*故障时间 = (乘数 + 1) * max(minRX, minTX)*。 使用此示例配置时，故障转移检测时间约为 500-600 毫秒。

在网络质量较差或链路频繁抖动的环境中，100 毫秒的 BFD 间隔可能过于敏感。 您可以将 *.spec.bfd.minRX* 和 *.spec.bfd.minTX* 增加到 500-1000 毫秒，以减少误报和频繁故障转移。 增加这些值也会延长故障转移检测时间，因此请根据稳定性与故障检测延迟之间所需的平衡进行调整。

> **NOTE**
>
> 现有连接可能会在故障转移期间中断，并需要重新连接。新连接仍可正常建立。

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
READY   INTERNAL IPS                EXTERNAL IPS                   WOR
KING NODES       AGE
gateway2   ovn-cluster   2          true          underlay-ext      Compl
eted   true    ["10.16.0.12","10.16.0.13"]   ["172.17.0.11","172.17.0.1
2"]   ["node1","node2"]   58s

$ kubectl get pod -n default -l ovn.kubernetes.io/vpc-egress-gateway=gate
way2 -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP
NODE    NOMINATED NODE   READINESS GATES
gateway2-fcc6b8b87-8lgvx   1/1     Running   0          2m18s   10.16.0.1
3   node2   <none>           <none>
gateway2-fcc6b8b87-wmww6   1/1     Running   0          2m18s   10.16.0.1
2   node1   <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  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="defau
lt/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="defau
lt/gateway2"}
logical_port        : "bfd@ovn-cluster"
min_rx              : 100
min_tx              : 100
options             : {}
status              : up
```

然后检查 BFD 会话：

```
$ kubectl exec -n default gateway2-fcc6b8b87-8lgvx -c bfdd -- bfdd-contro
l 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-contro
l 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 gateway 实例都已停止，VPC Egress Gateway 处理的出站流量将被丢弃。

已验证容量指南

以下结果在 ACP 4.3.2 和 Kube-OVN v1.15.13 上进行了验证：每个 VPC Egress Gateway 配置两个副本，bfdd CPU 请求/限制为 `50m/200m`。
请将其作为容量规划指南，而不是严格的产品限制。

NOTE

已验证容量仅供参考。实际容量可能因硬件性能和集群负载而有所不同。

BFD 设置	已验证规模	建议
`1000ms / 1000ms x 5`	100 个 gateway / 200 个 gateway Pod	适用于较小规模的部署。
`1500ms / 1500ms x 5`	150 个 gateway / 300 个 gateway Pod	使用 1500 ms 探测间隔，并将检测倍数设置为 5。
`3000ms / 3000ms x 5`	300 个 gateway / 600 个 gateway Pod	适用于中等规模。
`4000ms / 4000ms x 5`	400 个 gateway / 800 个 gateway Pod	使用此设置时应预留容量余量。
`5000ms / 5000ms x 5`	500 个 gateway / 1000 个 gateway Pod	建议用于 500-gateway 部署。

对于配备约 32 个 vCPU 和 96 GiB 内存的专用 Egress Gateway 节点，`500ms / 500ms x3` BFD 设置以及 bfdd `50m/200m` CPU 请求/限制经过验证，每个节点最多支持约 200 个 gateway Pod。

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新创建 gateway 实例：

1. 更改副本数
2. 更改配置，例如内部或外部 IP、节点选择器或 BFD 设置
3. 升级或降级 Kube-OVN（未指定 *.spec.image* 时）
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，使 Pod 能够使用多个网络接口。

借助 Multus CNI 和 Kube-OVN，您可以为 Pod 附加一个辅助网络，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均支持 IPv4、IPv6 和双栈网络。
- 辅助网络支持 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 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
商标声明

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。
- 您拥有安装集群插件和创建 Kubernetes 资源的权限。
- `kubectl` 已配置且可以访问目标集群。

安装 **Multus CNI**

部署 **Multus CNI** 插件

1. 前往 管理员 > **Marketplace** > 集群插件。
2. 搜索“multus”，然后选择 **“Alauda Container Platform Multi-Network”**。
3. 如果尚未安装该插件，请点击右侧的三点菜单 (⋮)，然后选择 安装。
4. 等待插件状态变为“已安装”。

> **NOTE**
>
> Multus CNI 插件作为其他 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**
>
> 配置中的 provider 格式为 `<NAME>.<NAMESPACE>.ovn`，其中 `<NAME>` 和 `<NAMESPACE>` 分别是此 NetworkAttachmentDefinition CR 的名称和命名空间。
>
> NetworkAttachmentDefinition 名称不得包含点号 (`.`)。 每个 NetworkAttachmentDefinition 只能用于一个子网。请勿在多个子网之间重复使用同一个 NetworkAttachmentDefinition。
>
> 如果使用 underlay 辅助网络，请将 Subnet 中的 `spec.vlan` 设置为目标 VLAN CR 名称。

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

应用此资源：

```bash
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。

```yaml
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: subnet1
spec:
  cidrBlock: 172.22.0.0/16
  provider: attachnet.default.ovn
```

应用此资源：

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

创建具有多个网络接口的 Pod

辅助网络就绪后，创建一个 Pod，并在 `metadata.annotations` 中引用 NetworkAttachmentDefinition。

NOTE

- 注解 `k8s.v1.cni.cncf.io/networks` 使用 `<NAMESPACE>/<NAME>` 格式，其中 `<NAMESPACE>` 和 `<NAME>` 分别是对应 NetworkAttachmentDefinition 的命名空间和名称。
- 要附加多个辅助网络，请指定以逗号分隔的多个值，例如 `default/attachnet,default/attachnet2`。

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

对于需要多个网络接口的工作负载，也可以在更高级别控制器（如 Deployment 或 StatefulSet）的 Pod 模板中指定网络附加注解。示例：

```yaml
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，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` 显示此 Pod 的所有网络接口状态，包括主接口和辅助接口。 在此示例中，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

运行以下命令之一：

```
kubectl exec pod1 -- ifconfig
```

```
kubectl exec pod1 -- ip addr show
```

> **NOTE**
>
> 具体可用的命令取决于容器镜像。如果镜像中不包含其中一个命令，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
NODE    SUBNET
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：要为主接口分配固定 IP 地址，请将注解 `ovn.kubernetes.io/ip_address=<IP>` 添加到 Pod。
- 辅助网络接口：要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，可以在 Pod 模板中指定固定 IP 注解。

额外路由

可以为引用同一 NetworkAttachmentDefinition 的所有 Pod，或为特定 Pod/工作负载配置额外路由。

为引用同一 **NetworkAttachmentDefinition** 的所有 **Pod** 配置额外路由

要为引用同一 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您环境中可从辅助网络访问的有效地址。

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，请将注解 `attachnet.default.ovn.kubernetes.io/routes` 添加到 Pod。示例：

```yaml
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，也可以在 Pod 模板中指定路由注解。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，在 Pod 内部验证生效的路由条目：

```
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
```

如果输出中未显示对应的目标和网关路由，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。

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是独立供应商。本产品与 F5, Inc. 无关联，亦未获得其认可或赞助。所有商标均归其各自所有者所有，仅用于标识目的。

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。

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

- 如果集中式网关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。

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

配置 IPPool

IPPool 是比 Subnet 更细粒度的 IPAM 管理单元。 您可以通过 IPPool 将子网段细分为多个单元，并将每个单元绑定到一个或多个命名空间。

目录

说明

创建 IPPool

使用 IPPool

注意事项

商标声明

说明

创建 IPPool

以下是一个示例：

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

(1) IPPool 所属的子网。

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

(3) IPPool 绑定的可选命名空间。绑定命名空间中的 Pod 只会从绑定的 IPPool 中获取 IP，而不会从子网中的其他范围获取 IP。

使用 IPPool

要从 IPPool 中随机分配 IP，只需将 IPPool 绑定到所需的命名空间。 创建绑定命名空间中的 Pod 时，其 IP 将从相应的 IPPool 中分配。

> **NOTE**
>
> 将 IPPool 绑定到命名空间之前，请确保只有 IPPool 所属的子网绑定到了该命名空间。

您还可以通过 annotation 将 IPPool 分配给 Pod：

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

对于工作负载，请在 Deployment、StatefulSet 等资源的 Pod 模板中使用 annotation：

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

注意事项

1. 为确保与 *Fixed Addresses* 功能兼容，IPPool 的名称不能是 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 IPPool 不能具有重叠的 IP 范围。

4. `.spec.ips` 字段可根据需要随时更新。任何更改都会立即生效。
5. IPPool 会继承子网的预留 IP。从 IPPool 中随机分配 IP 地址时，将跳过 IPPool 范围内的预留 IP。
6. 从子网中随机分配 IP 地址时，将排除该子网中所有 IPPool 的 IP 范围。
7. 同一命名空间可以绑定多个 IPPool。

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是独立供应商。本产品与 F5, Inc. 没有任何关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处使用这些商标仅用于识别目的。

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。

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）。

为 Kube-OVN Underlay 网络配置 IPv4 多播

本指南介绍如何为 Kube-OVN Underlay 子网中的工作负载启用 IPv4 多播通信。为子网启用 multicast snooping 后，Kube-OVN 会在对应的 OVN 逻辑交换机上配置 multicast snooping 和 multicast querier。

此配置按子网单独生效。它不会在不同的三层网络之间配置 multicast routing，也不会配置特定厂商的物理交换机。

目录

开始之前

启用 Multicast Snooping

使用 Web 控制台

查看配置结果

关闭 Multicast Snooping

开始之前

请确保满足以下条件：

- **ACP version**：4.2
- 目标子网是一个 Kube-OVN Underlay 子网，且 `spec.protocol` 设置为 `IPv4`。IPv6 和双栈子网不在本指南范围内。

- 物理网络已按照 准备 Kube-OVN Underlay 物理网络 完成准备。
- 该子网至少有一个可用 IP 地址。启用 multicast snooping 后，Kube-OVN 会为该子网创建一个 multicast querier。该 querier 会使用子网地址池中的一个可用 IP 地址以及一个自动生成的 MAC 地址。
- 物理网络允许应用所使用的 multicast group 和 IGMP 流量。

如果物理交换机启用了 IGMP snooping，请与网络管理员协作，确认可以在连接集群节点和外部接收端的 VLAN 端口上学习到 multicast 成员关系。不同厂商物理交换机的 querier 和 IGMP snooping 行为可能有所不同。

启用 **Multicast Snooping**

使用 **Web** 控制台

1. 进入 管理员。
2. 从顶部的 **Cluster** 下拉列表中选择目标集群。
3. 在左侧导航栏中，展开 **Clusters**，然后单击 **Resources**。
4. 在资源搜索框中输入 子网。展开 **Correlated with Cluster**，然后选择 子网 (`kubeovn.io`)。
5. 单击目标 Underlay 子网旁边的操作菜单（⋮），然后选择 **Update**。
6. 在 `spec` 中添加 `enableMulticastSnoop: true`：

```
spec:
  enableMulticastSnoop: true
```

7. 单击 **Update**。

对于每个需要承载 multicast 流量的 Underlay 子网，重复上述步骤。

查看配置结果

更新完成后，在 Web 控制台中查看目标子网的 YAML，并验证以下字段：

- `spec.enableMulticastSnoop` 为 `true`，表示已启用 multicast snooping。
- `status.mcastQuerierIP` 和 `status.mcastQuerierMAC` 有值，表示已创建 multicast querier。

如果 querier IP 或 MAC 地址为空，请确保该子网有可用 IP 地址。

关闭 **Multicast Snooping**

在 Web 控制台中打开目标子网的 YAML，将 `spec.enableMulticastSnoop` 设置为 `false`，然后单击 **Update**。

更新完成后，Kube-OVN 会释放 querier IP 和 MAC 地址，`status.mcastQuerierIP` 和 `status.mcastQuerierMAC` 也会变为空。

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
性能改进
如何激活
Pod 级别注解（推荐）
对于 ALB
对于 IngressNginx
对于 EnvoyGateway
对于自定义部署
Pod 级别 readinessGates（旧版）
卸载
商标声明

概述

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

主要功能

- 自动健康监控：持续监控 k8s 集群中服务端点的健康状态
- 负载均衡器集成：自动从服务中移除不健康的端点
- 服务可用性：确保流量仅定向到健康且可用的端点
- 快速故障转移：在节点断电期间，将端点切换时间从 40s 缩短至 10s

安装

通过 **Marketplace** 安装

1. 导航至 管理员 > **Marketplace** > 集群插件。
2. 在插件列表中搜索 "**Alauda Container Platform Endpoint Health Checker**"。
3. 点击 安装，打开安装配置页面。
4. 在部署配置对话框中，您可以选择配置以下参数：

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。点击 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 Endpoint Health Checker 组件调度到具有特定污点的节点上。点击 添加 可添加多个包含键、值和类型的容忍度。

5. 点击 安装 以部署插件。
6. 等待插件状态变为 "就绪"。

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 才会接收流量。它通过监控服务 Endpoint 并自动管理其可用状态来实现这一目标。

核心功能

Endpoint Health Checker 的工作方式如下：

1. 服务发现：识别集群中配置为进行健康监控的服务和 Pod。
2. **Pod** 健康监控：监控为服务 Endpoint 提供支持的 Pod 的就绪探针和存活探针状态。
3. 主动健康检查：使用可配置的条件执行主动健康评估：
   - **TCP** 连接检查：建立 TCP 连接以验证端口是否可访问。
4. **Endpoint** 管理：自动从服务 Endpoint 列表中移除不健康的 Endpoint，防止流量路由到故障实例。

健康检查流程

健康检查流程包括：

- 探针集成：使用 Kubernetes 就绪探针和存活探针的结果作为初始健康指标。
- 网络连接：向目标 Endpoint 端口发送 TCP 数据包，以验证其可访问性。
- 响应验证：评估响应状态、响应时间和内容，以确定 Endpoint 的健康状况。
- 自动故障转移：从服务 Endpoint 列表中移除无响应或故障的 Endpoint。

性能改进

- 之前的方法：依赖 kubelet 心跳检测，延迟最长可达 40 秒。
- 当前的方法：主动执行 Endpoint 健康检查，检测和切换时间为 10 秒。
- 改进效果：显著提升 ALB + MetalLB 环境在节点故障期间的服务可用性。

如何激活

可以通过以下两种方法激活健康检查：

Pod 级别注解（推荐）

对于 ALB

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

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

对于 IngressNginx

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

```yaml
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 注解中的说明操作，并将注解设置为 `endpoint-health-checker.io/enabled: 'true'`。

对于自定义部署

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

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

Pod 级别 readinessGates（旧版）

对于旧版本，请在 Pod spec 中配置 readinessGates：

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

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

卸载

要卸载 Endpoint Health Checker：

1. 导航至 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，然后选择 卸载。
4. 根据提示确认卸载。

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

MetalLB™ 是 The Linux Foundation 的商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 或 F5, Inc. 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处使用这些商标仅用于标识目的。

ALB 的任务

目录

如何为 alb-operator 设置 NodeSelector 和 Tolerations

如何为 alb 设置 NodeSelector 和 Tolerations

商标声明

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

```
kubectl 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": ""
      }
    }
  }
}'
```

商标声明

NGINX® 是 F5, Inc. 在美国及其他国家/地区的注册商标。

Alauda 是一家独立供应商。本产品与 F5, Inc. 无关，也未获得其认可或赞助。所有商标均为其各自所有者的财产，此处仅用于标识目的。

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 严格传输安全（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
标头修改
OCP Route 配置

Gateway API 配置

连接限制

OCP Route 配置

Gateway API 配置

限流

OCP Route 配置

Gateway API 配置

IP 允许列表/阻止列表

OCP Route 配置

Gateway API 配置

URL 重写

OCP Route 配置

Gateway API 配置

跨命名空间 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 重新加密

OCP Route 配置

Gateway API 配置

使用自定义证书的边缘终止

OCP Route 配置

Gateway API 配置

TLS 透传

OCP Route 配置

Gateway API 配置

功能对比摘要

迁移策略

相关文档

商标声明

简介

使用 Envoy Gateway 将 OpenShift Container Platform (OCP) Routes 迁移到 Kubernetes Gateway API HTTPRoutes。使用以下按功能划分的映射，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. 基本了解 Gateway API Routes

基本 **HTTP Route**

OCP Route 配置

在 OCP 中，使用 Route 资源创建基本 HTTP route：

```yaml
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 route 的主机名

2 后端服务引用

3 后端服务上的目标端口

Gateway API 配置

在 Gateway API 中，使用 HTTPRoute 配置等效功能：

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

1 此 route 接受的主机名（等效于 OCP Route 的 `host`）

2 对 Gateway listener 的引用

3 后端服务和端口

Route 超时

OCP Route 配置

在 OCP 中，使用注解配置超时：

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

(1) HTTP 请求的常规超时

(2) 隧道连接的超时（WebSocket、HTTP/2 等）

Gateway API 配置

在 Gateway API 中，在 HTTPRoute 规则中配置超时：

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

(1) 请求超时配置

有关更多详细信息，请参阅请求超时↗。

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

HTTP 严格传输安全（HSTS）

OCP Route 配置

在 OCP 中，使用边缘终止或重新加密 route 上的注解配置 HSTS：

```
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 标头配置

Gateway API 配置

在 Gateway API 中，使用响应标头修改配置 HSTS：

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

(1) 向响应添加 HSTS 标头

有关更多详细信息，请参阅 HTTP 标头修改↗。

> **NOTE**
>
> 与 OCP 的 `requiredHSTSPolicies` 集群范围强制机制不同，Gateway API 要求针对每个 route 进行显式配置。Gateway API 没有全局 HSTS 策略强制机制。ACP 通过 Envoy Gateway 的 `ClientTrafficPolicy` 扩展提供 Gateway 范围的 HSTS；有关更多信息，请参阅如何配置 HSTS。

基于 Cookie 的会话亲和性

OCP Route 配置

在 OCP 中，使用 annotations 配置会话亲和性：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/balance: source (1)
    haproxy.router.openshift.io/disable_cookies: "false" (2)
    router.openshift.io/cookie_name: my-cookie (3)
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

(1) 负载均衡算法

(2) 启用基于 Cookie 的会话亲和性

(3) 用于会话持久性的 Cookie 名称

Gateway API 配置

在 Gateway API 中，在 HTTPRoute 规则中配置会话持久性：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - backendRefs:
        - name: example-service
          port: 8080
      sessionPersistence: 1
        type: Cookie
        sessionName: my-cookie
        cookieConfig:
          lifetimeType: Permanent 2
```

1 会话持久性配置

2 Cookie 生命周期类型：`Permanent`（在浏览器会话之间保持）或 `Session`（浏览器关闭时过期）

有关更多详细信息，请参阅会话亲和性/粘性会话↗。

基于路径的路由

OCP Route 配置

在 OCP 中，基于路径的路由使用 `path` 字段：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  path: /api (1)
  to:
    kind: Service
    name: example-service
```

(1) 用于匹配请求的路径前缀

Gateway API 配置

在 Gateway API 中，在 HTTPRoute 规则中配置路径匹配：

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

(1) 路径匹配配置

Gateway API 支持多种匹配类型：

- `PathPrefix`：匹配路径前缀（等效于 OCP 的默认行为）
- `Exact`：匹配确切路径
- `RegularExpression`：使用正则表达式进行匹配

有关更多详细信息，请参阅 HTTPRoute 匹配。

标头修改

OCP Route 配置

在 OCP 中，使用 annotations 修改标头：

```
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) 设置响应标头

(2) 设置请求标头

Gateway API 配置

在 Gateway API 中，使用过滤器配置标头修改：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - filters:
        - type: RequestHeaderModifier (1)
          requestHeaderModifier:
            add:
              - name: X-Request-Header
                value: value
        - type: ResponseHeaderModifier (2)
          responseHeaderModifier:
            add:
              - name: X-Custom-Header
                value: value
      backendRefs:
        - name: example-service
          port: 8080
```

(1) 请求标头修改

(2) 响应标头修改

有关更多详细信息，请参阅 HTTP 标头修改↗。

连接限制

OCP Route 配置

在 OCP 中，使用 annotations 配置连接限制：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/pod-concurrent-connections: "100" (1)
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

(1) 每个后端 pod 的最大并发连接数

Gateway API 配置

在 Gateway API 中，使用附加到 Gateway 的 ClientTrafficPolicy 配置连接限制：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: ClientTrafficPolicy
metadata:
  name: connection-limit-policy
  namespace: demo
spec:
  targetRefs: (1)
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: demo-gateway
  connection: (2)
    connectionLimit:
      value: 100
```

(1) 将策略附加到 Gateway

(2) 连接限制配置

有关更多详细信息，请参阅连接限制↗。

NOTE

与 OCP 的每个后端 pod 限制不同，Gateway API 的连接限制应用于 Gateway 级别，并分布在各个 Envoy 代理实例之间。

限流

OCP Route 配置

在 OCP 中，限流使用注解进行配置：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/rate-limit-connections: "true" 1
    haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp: "1
0"
    haproxy.router.openshift.io/rate-limit-connections.rate-http: "100"
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

1 限流配置

Gateway API 配置

在 Gateway API 中，使用 BackendTrafficPolicy 配置限流：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: BackendTrafficPolicy
metadata:
  name: rate-limit-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: example-route
  rateLimit: 1
    type: Local
    local:
      rules:
        - limit:
            requests: 100
            unit: Second
```

1 限流配置

有关更多详细信息，请参阅限流↗。

> **NOTE**
>
> Gateway API 限流比 OCP Route 注解更加灵活，同时支持本地限流和全局限流机制。

IP 允许列表/阻止列表

OCP Route 配置

在 OCP 中，IP 允许列表使用注解进行配置：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/ip_allowlist: 192.168.1.0/24 10.0.0.1 1
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
```

1 IP 允许列表配置

Gateway API 配置

在 Gateway API 中，使用带有授权规则的 SecurityPolicy 实现 IP 过滤：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: ip-filter-policy
  namespace: demo
spec:
  targetRefs: 1
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: example-route
  authorization: 2
    defaultAction: Deny
    rules:
      - action: Allow
        principal:
          clientCIDRs: 3
            - 192.168.1.0/24
            - 10.0.0.1/32
```

1 将策略附加到 HTTPRoute

2 配置默认拒绝的授权规则

3 使用 CIDR 表示法配置 IP 允许列表

对于 IP 阻止列表（拒绝列表），设置 `defaultAction: Allow` 并使用 `action: Deny`：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: ip-blocklist-policy
  namespace: demo
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: example-route
  authorization:
    defaultAction: Allow (1)
    rules:
      - action: Deny (2)
        principal:
          clientCIDRs:
            - 192.168.100.0/24
```

1 默认操作允许所有流量

2 拒绝来自特定 IP 的流量

有关更多详细信息，请参阅IP 允许列表/拒绝列表↗。

> **NOTE**
>
> 如果 Gateway 位于负载均衡器或代理之后，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 重写使用注解进行配置：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
  annotations:
    haproxy.router.openshift.io/rewrite-target: /new-path (1)
spec:
  host: example.com
  path: /old-path
  to:
    kind: Service
    name: example-service
```

(1) 重写目标路径

Gateway API 配置

在 Gateway API 中，使用过滤器配置 URL 重写：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  hostnames:
    - example.com
  parentRefs:
    - name: demo-gateway
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /old-path
      filters:
        - type: URLRewrite (1)
          urlRewrite:
            path:
              type: ReplacePrefixMatch
              replacePrefixMatch: /new-path
      backendRefs:
        - name: example-service
          port: 8080
```

(1) URL 重写过滤器

有关更多详细信息，请参阅URL 重写↗。

跨命名空间 **Route** 接纳

OCP Route 配置

在 OCP 中，Route 接纳策略控制不同命名空间中的 Route 是否可以声明相同的主机名。此配置通过 Ingress Operator 在集群级别进行。

Gateway API 配置

在 Gateway API 中，跨命名空间访问在 Gateway 监听器级别进行控制：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: gateway-ns
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: http
      protocol: HTTP
      port: 80
      allowedRoutes: 1
        namespaces:
          from: All  # Allow routes from all namespaces
```

1 配置哪些命名空间可以附加 Route

`allowedRoutes.namespaces.from` 的选项：

- `Same`：仅允许与 Gateway 位于同一命名空间中的 Route
- `All`：允许来自任意命名空间的 Route
- `Selector`：允许来自与标签选择器匹配的命名空间的 Route

有关更多详细信息，请参阅跨命名空间路由。

> **NOTE**
>
> 与 OCP 的集群范围接纳策略不同，Gateway API 在单个 Gateway 监听器级别进行控制，因此能够提供更精细的控制。

Ingress 的默认 TLS 证书

OCP Route 配置

在 OCP 中，没有 TLS 配置的 Route 可以使用在 Ingress Controller 级别配置的默认证书。

Gateway API 配置

在 Gateway API 中，在 Gateway 监听器上配置默认 TLS 证书：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      tls: 1
        mode: Terminate
        certificateRefs:
          - name: default-tls-cert
```

1 监听器的默认 TLS 证书

附加到此监听器且未配置特定 TLS 的任何 HTTPRoute 都将使用此默认证书。

使用自定义 CA 的 TLS 重新加密

OCP Route 配置

在 OCP 中，重新加密 Route 会在路由器处终止 TLS，并在验证后对后端重新加密：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: reencrypt (1)
    destinationCACertificate: | (2)
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
```

(1) 重新加密终止模式

(2) 用于验证后端的 CA 证书

Gateway API 配置

在 Gateway API 中，使用 BackendTLSPolicy 配置后端 TLS 验证：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      tls:
        mode: Terminate (1)
        certificateRefs:
          - name: frontend-tls
---
apiVersion: gateway.networking.k8s.io/v1
kind: BackendTLSPolicy
metadata:
  name: backend-tls-policy
  namespace: demo
spec:
  targetRefs: (2)
    - group: ""
      kind: Service
      name: example-service
  validation: (3)
    caCertificateRefs:
      - name: backend-ca-cert
        kind: ConfigMap
    hostname: backend.example.com
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
  hostnames:
    - example.com
  rules:
```

```
    - backendRefs:
        - name: example-service
          port: 8443
```

1. 在 Gateway 处终止 TLS
2. 将策略应用于后端 Service
3. 后端 TLS 验证配置

有关更多详细信息，请参阅后端 TLS↗。

> **NOTE**
>
> CA 证书必须存储在 ConfigMap 中，而不是 Secret 中。

使用自定义证书的边缘终止

OCP Route 配置

在 OCP 中，边缘终止使用 Route 的 TLS 配置：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: edge (1)
    certificate: | (2)
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    key: | (3)
      -----BEGIN PRIVATE KEY-----
      ...
      -----END PRIVATE KEY-----
```

(1) 边缘终止模式

(2) TLS 证书

(3) TLS 私钥

Gateway API 配置

在 Gateway API 中，在 Gateway 监听器上配置 TLS 终止：

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: example-tls
  namespace: demo
type: kubernetes.io/tls (1)
data:
  tls.crt: <base64-encoded-cert>
  tls.key: <base64-encoded-key>
---
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: https
      protocol: HTTPS (2)
      port: 443
      tls:
        mode: Terminate (3)
        certificateRefs: (4)
          - name: example-tls
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
      sectionName: https
  hostnames:
    - example.com
  rules:
    - backendRefs:
        - name: example-service
          port: 8080
```

1. TLS Secret 类型
2. HTTPS 协议
3. 终止 TLS 模式
4. 对 TLS Secret 的引用

TLS 透传

OCP Route 配置

在 OCP 中，透传 Route 不会终止 TLS：

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: example-route
spec:
  host: example.com
  to:
    kind: Service
    name: example-service
  tls:
    termination: passthrough (1)
```

1. 透传终止模式

Gateway API 配置

在 Gateway API 中，TLS 透传使用 TLSRoute：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: demo-gateway
  namespace: demo
spec:
  gatewayClassName: envoy-gateway-operator-cpaas-default
  listeners:
    - name: tls-passthrough
      protocol: TLS (1)
      port: 443
      tls:
        mode: Passthrough (2)
---
apiVersion: gateway.networking.k8s.io/v1alpha2
kind: TLSRoute
metadata:
  name: example-route
  namespace: demo
spec:
  parentRefs:
    - name: demo-gateway
      sectionName: tls-passthrough
  hostnames: (3)
    - example.com
  rules:
    - backendRefs:
        - name: example-service
          port: 8443
```

(1) TLS 协议

(2) 透传模式

(3) 用于路由的 SNI 主机名

有关更多详细信息，请参阅TLS 透传和TLSRoute 规范↗。

> **NOTE**
>
> TLSRoute 使用 SNI（服务器名称指示）进行路由决策，而不会终止 TLS。这意味着：
>
> - 无法访问用于路由决策的 HTTP 标头或请求内容

- 无法应用 L7 过滤器（标头修改、URL 重写等）
- 路由完全基于 SNI 主机名

功能对比摘要

功能	**OCP Route**	**Gateway API**
基本 HTTP 路由	Route 规范	HTTPRoute
基于路径的路由	`.spec.path`	`.spec.rules[].matches[].path`
超时	注解	HTTPRoute `.spec.rules[].timeouts`
HSTS	注解	ResponseHeaderModifier 过滤器
会话亲和性	注解	HTTPRoute `.spec.rules[].sessionPersistence`
标头修改	注解	RequestHeaderModifier/ResponseHeaderModifier 过滤器
连接限制	注解	ClientTrafficPolicy
速率限制	注解	BackendTrafficPolicy

功能	OCP Route	Gateway API
IP 允许列表/阻止列表	注解	带授权的 SecurityPolicy
URL 重写	注解	URLRewrite 过滤器
跨命名空间	集群级策略	Gateway `.spec.listeners[].allowedRoutes`
默认 TLS 证书	控制器级别	Gateway 侦听器 TLS
TLS 重新加密	`.spec.tls.termination: reencrypt`	BackendTLSPolicy
TLS 边缘终止	`.spec.tls.termination: edge`	使用 HTTPS 的 Gateway 侦听器 TLS
TLS 透传	`.spec.tls.termination: passthrough`	采用透传模式的 TLSRoute

迁移策略

从 OCP Route 迁移到 Gateway API 时：

1. 从 **Gateway** 开始：根据您的使用场景，创建具有适当侦听器的 Gateway 资源
   - 将 Gateway 部署在同一命名空间中，或部署在专用的 gateway 命名空间中
   - 为所需的所有协议（HTTP、HTTPS、TLS、TCP、UDP）配置侦听器
   - 确保已创建 Gateway service，并且具有可访问的端点
2. 将 **Route** 转换为 **HTTPRoute**：将每个 OCP Route 迁移为 HTTPRoute，从简单的路由开始
   - 从非生产或低流量路由开始，以最大程度地降低风险

- 验证主机名和路径匹配配置
- 在添加高级功能之前，测试基本连接

3. 应用策略：对于需要策略（BackendTrafficPolicy、SecurityPolicy、ClientTrafficPolicy）的功能，请在基本路由正常工作后创建并附加这些策略

- 一次添加一个策略，并验证其功能
- 监控是否存在任何意外行为或性能影响

4. 增量测试：在继续迁移下一个路由之前，验证每个迁移步骤

- 监控：检查 Gateway 和 Route 的状态条件，确认是否存在错误
- 功能测试：使用 curl 或自动化测试验证所有路由是否按预期工作
- 性能测试：将响应时间和吞吐量与 OCP Route 进行比较
- 验证检查：
  - 验证 TLS 证书是否已正确应用
  - 测试会话亲和性和负载均衡行为
  - 验证速率限制和安全策略
  - 检查标头修改和 URL 重写

5. 更新 **DNS**：验证完成后，更新 DNS 条目，使其指向新的 Gateway service

- 双重运行期间（推荐）：将部分流量指向新的 Gateway，同时保持 OCP Route 处于活动状态，以便支持回滚
- 使用加权 DNS 或金丝雀部署逐步转移流量
- 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 恢复为指向 OCP Route
- DNS 切换后至少保持 OCP Route 活动 24–48 小时
- 记录导致问题的任何配置差异
- 如果需要回滚，为利益相关者制定沟通计划

相关文档

- 配置 GatewayAPI Gateway
- 配置 GatewayAPI Route
- 配置 GatewayAPI Policy
- Envoy Gateway 任务

商标声明

Envoy® 是 The Linux Foundation 的注册商标。

Alauda 是独立供应商。本产品与 The Linux Foundation 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

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.
```