Menu

网络

介绍

介绍

- 使用限制

架构

理解 **Kube-OVN**

- 上游 OVN/OVS 组件
- 核心控制器和代理
- 监控、运维工具和扩展组件

理解 **ALB**

- 核心组件
- 快速开始
- ALB、ALB 实例、Frontend/FT、Rule、Ingress 和 Project 之间的关系
- ALB Leader
- 其他资源：

了解 **MetalLB**

- 术语
- MetalLB 高可用原理
- 选择 VIP 承载节点的算法
- 其他资源

核心概念

ALB 与 **Ingress-NGINX** 注解兼容性

- 基本概念
- 支持的 ingress-nginx 注解

Service、Ingress、Gateway API 与 ALB Rule 之间的比较

- 针对 L4（TCP/UDP）流量
- 针对 L7（HTTP/HTTPS）流量

GatewayAPI

- 通过 ALB

功能指南

创建服务

- 为什么需要 Service
- ClusterIP 类型 Service 示例：
- Headless Service（无头服务）
- 通过 Web 控制台创建服务
- 通过 CLI 创建服务
- 示例：集群内访问应用
- 示例：集群外访问应用
- 示例：ExternalName 类型 Service
- LoadBalancer 类型 Service 注解

创建 **Ingress**

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

创建域名

- 域名自定义资源（CR）示例
- 通过 Web 控制台创建域名
- 通过 CLI 创建域名
- 后续操作
- 相关资源

创建证书

- 通过 Web 控制台创建证书

创建外部 **IP** 地址池

- 前提条件
- 约束与限制
- 部署 MetalLB 插件
- IPAddressPool 自定义资源（CR）示例
- 通过 Web 控制台创建外部 IP 地址池
- 通过 CLI 创建外部 IP 地址池
- 查看告警策略

创建 **BGP Peers**

- 术语
- 前提条件
- BGPPeer 自定义资源（CR）示例
- 通过 Web 控制台创建 BGPPeer
- 通过 CLI 创建 BGPPeer

配置子网

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

配置网络策略

- 通过 Web 控制台创建 NetworkPolicy
- 通过 CLI 创建 NetworkPolicy
- 参考

创建 **Admin** 网络策略

- 注意事项
- 通过 Web 控制台创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy
- 通过 CLI 创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy
- 其他资源

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

- 安装 Multus CNI
- 创建子网
- 创建多网卡 Pod
- 验证双网卡创建
- 其他功能

配置集群网络策略

- 注意事项
- 操作步骤

配置 **Egress Gateway**

- 关于 Egress Gateway
- 注意事项
- 使用方法
- 配置参数
- 相关资源

网络可观测性

- 关于 DeepFlow
- 安装 DeepFlow
- 其他资源

配置 ALB 规则

- 介绍
- 动作
- 后端
- 创建规则
- Https
- Ingress

集群互联（Alpha）

支持配置网络模式为 Kube-OVN 的集群之间的集群互联，实现集群间 Pod 的相互访问。

- 前提条件
- 多节点 Kube-OVN 互联控制器构建
- 在 global 集群部署集群互联控制器
- 加入集群互联
- 相关操作

Endpoint Health Checker

- Overview
- Key Features
- Installation
- How It Works
- Uninstallation

NodeLocal DNSCache

- Overview
- Key Features
- Important Notes
- Installation
- How It Works
- Configuration

如何操作

准备 Kube-OVN Underlay 物理网络

- 使用说明
- 术语解释
- 环境要求
- 配置示例

软数据中心 LB 方案（Alpha）

- 前提条件
- 操作步骤
- 验证

Underlay 和 Overlay 子网的自动互联

- 操作步骤

通过集群插件安装 Ingress-Nginx

- 概述
- 安装
- 配置管理
- 性能调优
- 重要说明

通过 Ingress Nginx Operator 安装 Ingress-Nginx

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

Ingress-Nginx 的任务

- 前提条件
- 最大连接数
- 超时
- 会话保持
- 头部修改
- URL 重写
- HSTS（HTTP 严格传输安全）
- 速率限制
- WAF
- 转发头控制
- HTTPS

ALB

Calico 网络支持 WireGuard 加密

- 安装状态
- 术语
- 注意事项
- 先决条件
- 操作步骤
- 结果验证

Kube-OVN Overlay 网络支持 IPsec 加密

- 术语
- 注意事项
- 先决条件
- 操作步骤

故障排除

如何解决 ARM 环境中的节点间通信问题？

查找错误原因

介绍

容器网络是一种为云原生应用设计的综合网络解决方案，确保集群内的东西向通信顺畅，以及跨外部网络的南北向流量高效管理，同时提供关键的网络功能。它由以下核心组件组成：

- 用于集群内东西向流量管理的 Container Network Interfaces (CNIs)。
- 用于管理 HTTPS 入口流量的 Ingress Gateway Controller ALB。
- 用于处理 LoadBalancer 类型服务的 MetalLB。
- 此外，还提供强大的网络安全和加密功能，以确保通信安全。

目录

使用限制

使用限制

虽然容器网络提供了广泛的功能，但需注意以下限制：

- 底层网络要求

  某些底层网络功能，如 Kube-OVN Underlay Subnet、Egress IP 和 MetalLB，依赖于底层 L2 网络支持。 这些功能无法在公有云提供商及某些虚拟化环境（如 AWS 和 GCP）中使用。

凭借其多功能设计和全面的功能集，容器网络使组织能够构建、扩展和管理安全、可靠且高性能的容器化应用。

Menu

架构

理解 **Kube-OVN**

- 上游 OVN/OVS 组件
- 核心控制器和代理
- 监控、运维工具和扩展组件

理解 **ALB**

- 核心组件
- 快速开始
- ALB、ALB 实例、Frontend/FT、Rule、Ingress 和 Project 之间的关系
- ALB Leader
- 其他资源：

了解 MetalLB

- 术语
- MetalLB 高可用原理
- 选择 VIP 承载节点的算法
- 其他资源

理解 Kube-OVN

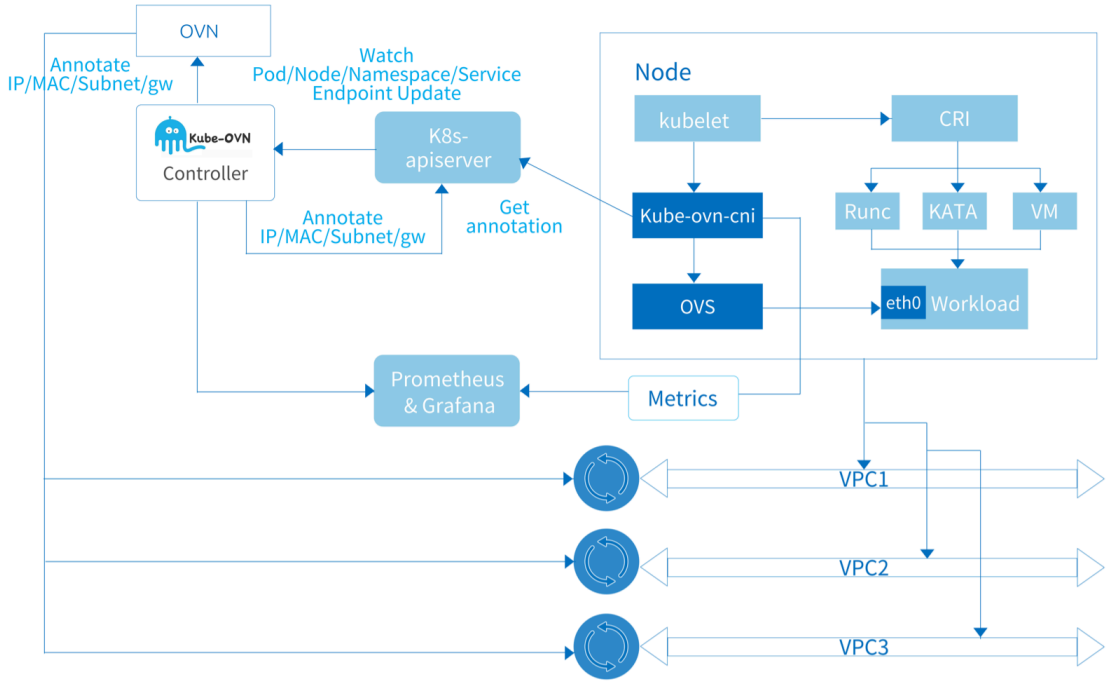
本文档描述了 Kube-OVN 的整体架构、各组件的功能以及它们之间的交互关系。

总体而言，Kube-OVN 充当 Kubernetes 和 OVN 之间的桥梁，将成熟的 SDN 与云原生相结合。 这意味着 Kube-OVN 不仅实现了 Kubernetes 下的网络规范，例如 CNI、Service 和 NetworkPolicy， 而且还将大量的 SDN 领域能力带入云原生，比如逻辑交换机、逻辑路由器、VPC、 网关、QoS、ACL 和流量镜像。

Kube-OVN 还保持了良好的开放性，以便与许多技术解决方案进行集成，如 Cilium、Submariner、Prometheus、KubeVirt 等。

Kube-OVN 的组件大致可以分为三类：

- 上游 OVN/OVS 组件。
- 核心控制器和代理。
- 监控、运维工具和扩展组件。



目录

上游 OVN/OVS 组件
ovn-central
ovs-ovn
核心控制器和代理
kube-ovn-controller
kube-ovn-cni
监控、运维工具和扩展组件
kube-ovn-speaker
kube-ovn-pinger
kube-ovn-monitor
kubectl-ko

上游 **OVN/OVS** 组件

这类组件来自 OVN/OVS 社区，经过特定修改以适应 Kube-OVN 使用场景。 OVN/OVS 本身是一个成熟的 SDN 系统，用于管理虚拟机和容器， 我们强烈建议对 Kube-OVN 实现感兴趣的用户先阅读 ovn-architecture(7) ↗ 以了解 OVN 是什么以及如何集成它。 Kube-OVN 使用 OVN 的北向接口来创建和协调虚拟网络，并将网络概念映射到 Kubernetes 中。

所有与 OVN/OVS 相关的组件都已打包为镜像，并准备在 Kubernetes 中运行。

ovn-central

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

- `ovn-nb`：保存虚拟网络配置，并为虚拟网络管理提供 API。`kube-ovn-controller` 主要与 `ovn-nb` 交互，以配置虚拟网络。
- `ovn-sb`：保存从 `ovn-nb` 的逻辑网络生成的逻辑流表，以及每个节点的实际物理网络状态。
- `ovn-northd`：将 `ovn-nb` 的虚拟网络转换为 `ovn-sb` 中的逻辑流表。

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

ovs-ovn

`ovs-ovn` 作为 DaemonSet 在每个节点上运行， `openvswitch` 、 `ovsdb` 和 `ovn-controller` 在 Pod 内部运行。 这些组件充当 `ovn-central` 的代理，以将逻辑流表转换为实际的网络配置。

核心控制器和代理

这一部分是 Kube-OVN 的核心组件，充当 OVN 和 Kubernetes 之间的桥梁，连接这两个系统，并在它们之间转换网络概念。 大多数核心功能在这些组件中实现。

kube-ovn-controller

此组件负责将 Kubernetes 中的所有资源转换为 OVN 资源，并充当整个 Kube-OVN 系统的控制平面。 `kube-ovn-controller` 监听与网络功能相关的所有资源的事件，并根据资源变化更新 OVN 中的逻辑网络。 监听的主要资源包括：

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

以 Pod 事件为例， `kube-ovn-controller` 监听 Pod 创建事件，通过内置的内存 IPAM 功能分配地址， 并调用 `ovn-central` 创建逻辑端口、静态路由和可能的 ACL 规则。 接下来， `kube-ovn-controller` 将分配的地址以及 CIDR、网关、路由等子网信息写入 Pod 的注解。 然后， `kube-ovn-cni` 读取这个注解，并用于配置本地网络。

kube-ovn-cni

此组件作为 DaemonSet 在每个节点上运行，实现 CNI 接口，并操作本地 OVS 配置本地网络。

该 DaemonSet 将 `kube-ovn` 二进制文件复制到每台机器上，作为 `kubelet` 和 `kube-ovn-cni` 之间的交互工具。 该二进制文件向 `kube-ovn-cni` 发送相应的 CNI 请求以进行进一步操作。 默认情况下，二进制文件将复制到 `/opt/cni/bin` 目录。

`kube-ovn-cni` 将配置特定的网络以执行适当的流量操作，其主要任务包括：

1. 配置 `ovn-controller` 和 `vswitchd`。
2. 处理 CNI Add/Del 请求：
    1. 创建或删除 veth 对，并绑定或解绑到 OVS 端口。
    2. 配置 OVS 端口。
    3. 更新宿主机的 iptables/ipset/route 规则。
3. 动态更新网络 QoS。
4. 创建并配置 `ovn0` NIC，以连接容器网络和宿主机网络。
5. 配置宿主机 NIC 以实现 Vlan/Underlay/EIP。
6. 动态配置跨集群网关。

监控、运维工具和扩展组件

这些组件提供监控、诊断和操作工具，以及扩展 Kube-OVN 核心网络能力的外部接口， 并简化日常操作和维护。

kube-ovn-speaker

此组件作为 DaemonSet 在特定标记的节点上运行，向外部发布路由， 允许通过 Pod IP 直接访问容器。

kube-ovn-pinger

此组件作为 DaemonSet 在每个节点上运行，用于收集 OVS 状态信息、节点网络质量、网络延迟等。

kube-ovn-monitor

此组件收集 OVN 状态信息和监控指标。

kubectl-ko

此组件是一个 kubectl 插件，可以快速执行常见操作。

理解 **ALB**

ALB（Another Load Balancer）是一个基于 OpenResty 的 Kubernetes Gateway，拥有来自 Alauda 多年生产环境的经验。

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

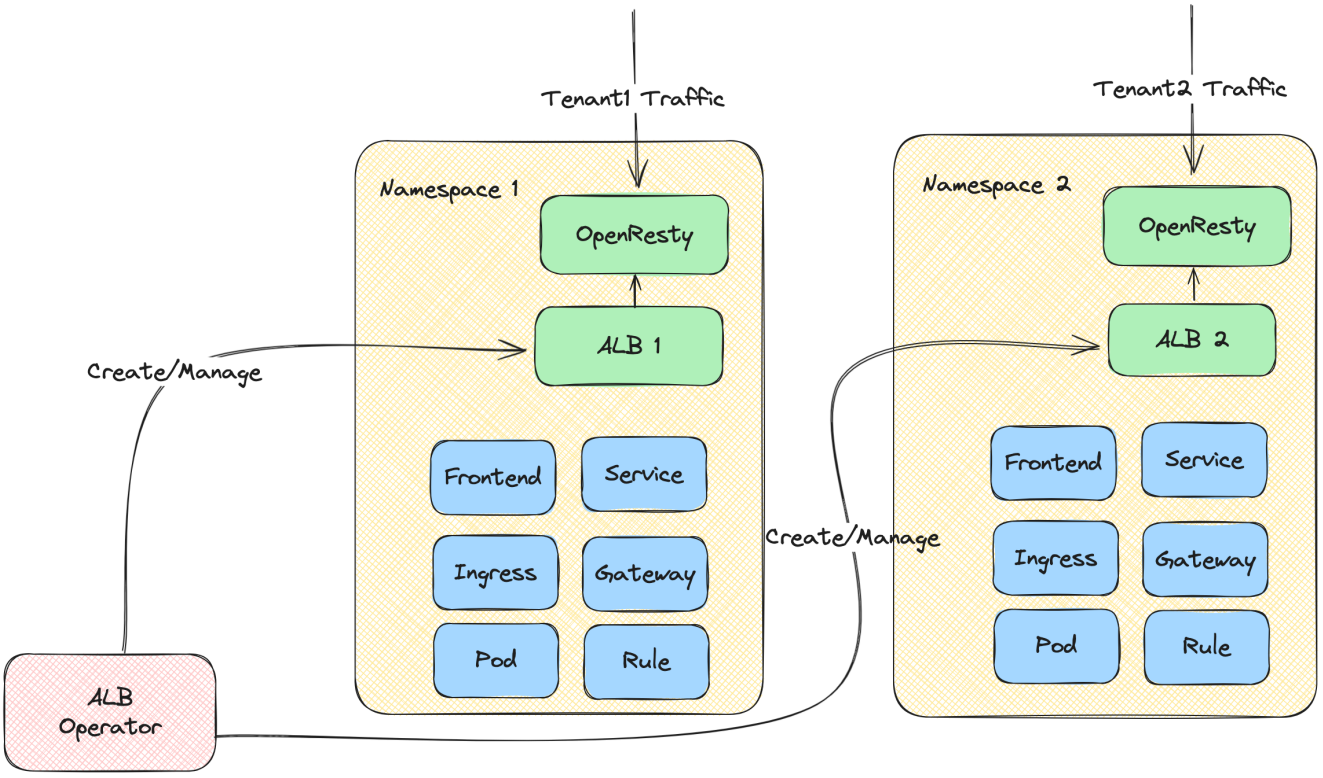
- Ingress
- Ingress Controller
- ALB
- ALB 实例
- ALB-Operator
- Frontend（简称 FT）
- RULE

ALB Leader

- Project

其他资源：

核心组件



- **ALB Operator**：管理 ALB 实例生命周期的 operator。它监控 ALB CR 并为不同租户创建/更新实例。
- **ALB Instance**：ALB 实例包含作为数据平面的 OpenResty 和作为控制平面的 Go 控制器。Go 控制器监控各种 CR（Ingress、Gateway、Rule 等），并将它们转换为 ALB 专用的 DSL 规则。OpenResty 使用这些 DSL 规则来匹配和处理传入请求。

快速开始

部署 **ALB Operator**

1. 创建一个集群。
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"  # 部署 alb 的节点 IP 地址
    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: docker.io/crccheck/hello-world:latest
        imagePullPolicy: IfNotPresent
---
apiVersion: v1
kind: Service
metadata:
  name: hello-world
  labels:
    k8s-app: hello-world
spec:
  ports:
  - name: http
    port: 80
    targetPort: 8000
  selector:
    k8s-app: hello-world
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-world
spec:
  rules:
```

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

现在你可以通过 `curl http://${ip}` 访问该应用。

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

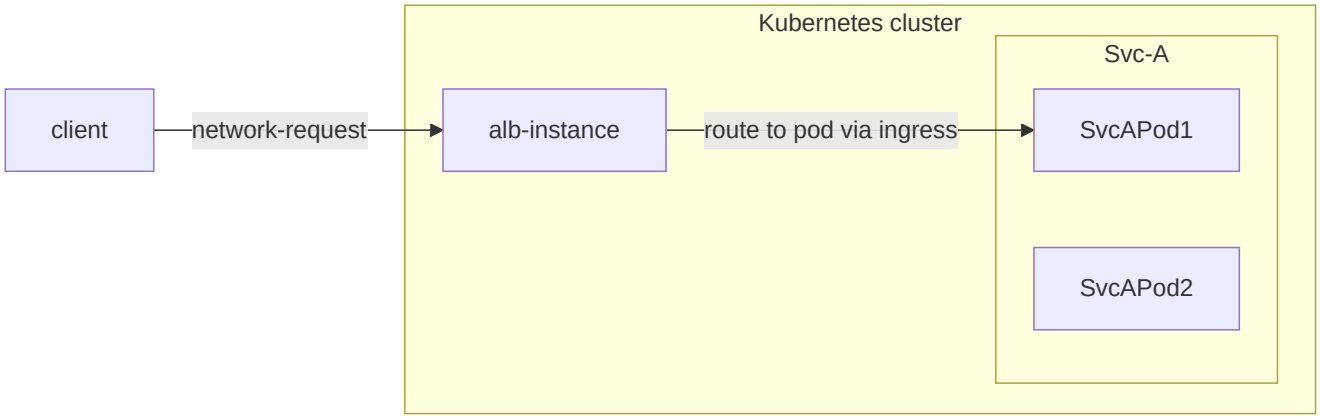
LoadBalancer 是现代云原生架构中的关键组件，作为智能流量路由和负载均衡器。

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

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

这些组件协同工作，实现灵活且强大的流量管理能力。

以下说明这些概念在请求路径中的协作方式。每个概念的详细介绍在单独文章中讲解。



在请求调用链中：

1. 客户端发送 HTTP/HTTPS/其他协议请求，最终请求会到达 **ALB** 的某个 **pod**，该 pod（即 ALB 实例）开始处理请求。
2. 该 ALB 实例找到一个能够匹配该请求的规则。
3. 如有需要，根据规则修改/重定向/重写请求。
4. 从规则配置的服务中找到并选择一个 pod IP，将请求转发给该 pod。

Ingress

Ingress 是 Kubernetes 中的资源，用于描述请求应发送到哪个服务。

Ingress Controller

理解 Ingress 资源并将请求代理到服务的程序。

ALB

ALB 是一种 Ingress controller。

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

ALB 由用户手动创建。每个 ALB 有自己的 IngressClass。创建 Ingress 时，可以使用 `.spec.ingressClassName` 字段指定由哪个 Ingress controller 处理该 Ingress。

ALB 实例

ALB 也是集群中运行的 Deployment（多个 pod）。每个 pod 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 及其他配置。

ALB-Operator

ALB-Operator 是集群中默认部署的组件，是 ALB 的 operator。它根据 ALB 资源创建/更新/删除 Deployment 及其他相关资源。

Frontend（简称 FT）

FT 是 ALB 自定义的资源，用于表示 ALB 实例监听的端口。

FT 可以由 ALB-Leader 或用户手动创建。

ALB-Leader 创建 FT 的情况：

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

RULE

RULE 是 ALB 自定义的资源。 它的作用类似于 Ingress，但更具体。 一个 RULE 唯一关联一个 FT。

RULE 可以由 ALB-Leader 或用户手动创建。

ALB-Leader 创建 RULE 的情况：

1. 将 Ingress 同步为 RULE。

ALB Leader

在多个 ALB 实例中，会选举出一个 leader。 Leader 负责：

1. 将 Ingress 转换为 Rules。 会为 Ingress 中的每个路径创建对应的 Rule。

2. 创建 Ingress 需要的 FT。 例如，Ingress 有证书则创建 FT 443（HTTPS），无证书则创建 FT 80（HTTP）。

Project

从 ALB 角度看，Project 是一组 namespace。

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

其他资源：

- 配置 Load Balancer

了解 MetalLB

目录

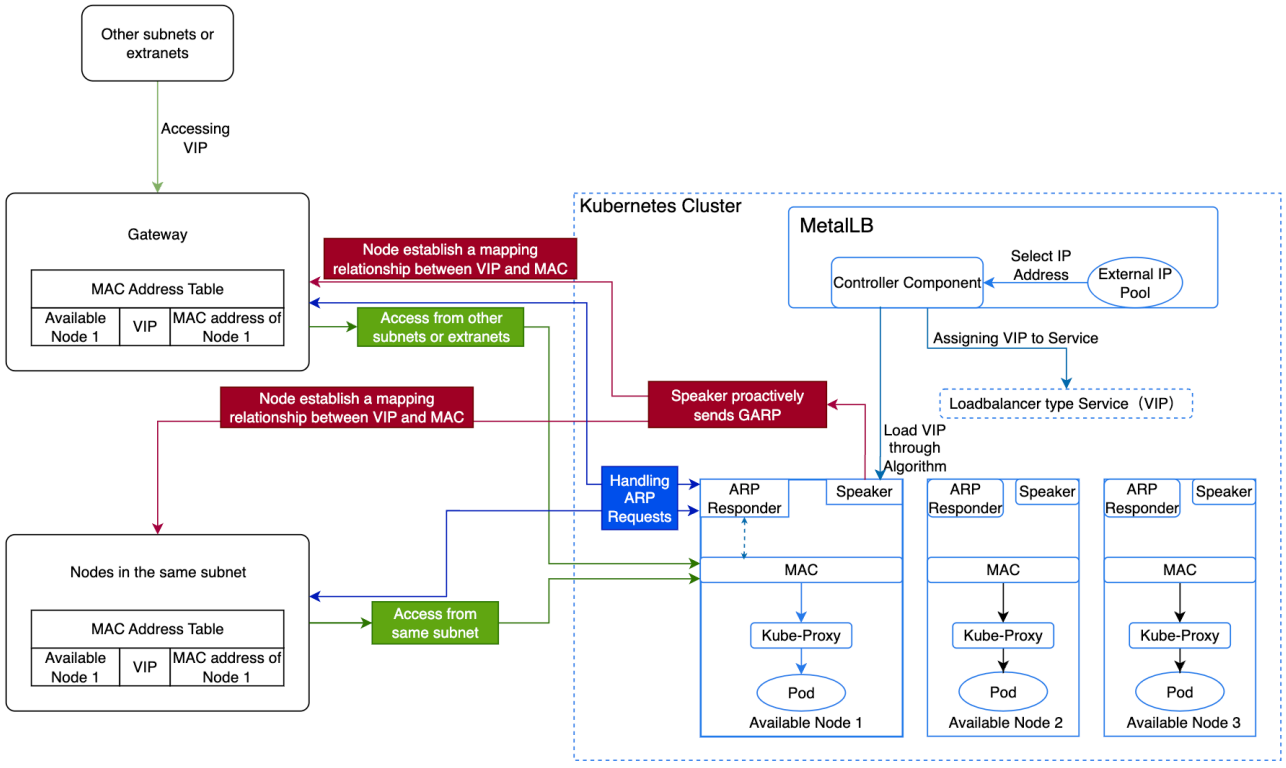
术语
MetalLB 高可用原理
选择 VIP 承载节点的算法
  计算公式
  应用示例
其他资源

术语

术语	描述
VIP	虚拟 IP 地址（VIP）是 MetalLB 为 LoadBalancer 类型的内部路由分配的 IP 地址，为外部流量访问集群内服务提供统一的访问入口。
ARP	地址解析协议（ARP）用于将网络层的 IP 地址映射到数据链路层的 MAC 地址。
GARP	免费 ARP（GARP）是一种特殊的 ARP 请求，用于通知网络中其他节点 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，而是主动在网络中发送信息。

术语	描述
ARP Responder	MetalLB 的一个组件，负责响应 ARP 请求，将 VIP 映射到节点的 MAC 地址。当节点需要与 VIP 通信时，会发送 ARP 请求以获取与 VIP 对应的 MAC 地址。每个可用节点都有一个 ARP Responder 来响应这些请求，将 VIP 映射到该节点的 MAC 地址。
Controller	MetalLB 的一个组件，负责从外部地址池动态分配 VIP 用于 LoadBalancer 类型的内部路由。Controller 监听集群中内部路由的创建和删除事件，以便按需分配或释放 VIP。
Speaker	MetalLB 的一个组件，根据策略或算法决定节点是否承载 VIP 并发送 GARP。它确保节点之间达到一定的负载平衡，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。

MetalLB 高可用原理



平台默认使用 MetalLB 的 ARP 模式，具体实现流程和原理如下：

- MetalLB 的 Controller 组件从外部地址池中选择一个 IP 地址，分配给 LoadBalancer 类型的内部路由作为 VIP。

- MetalLB 根据 算法选择一个可用节点作为 leader 来承载该 VIP，并转发流量。
- 该节点上的 Speaker 组件主动发送 GARP，在所有节点间建立 VIP 与 MAC 地址的映射关系。
  - 同一子网内的节点在获知 VIP 与可用节点 MAC 地址的映射后，访问 VIP 时会直接与该节点通信。
  - 不同子网的节点则先将流量路由到其子网的网关，再由网关转发到承载 VIP 的节点。
- 当该节点发生故障时，MetalLB 会选择另一个 leader 承载 VIP，并发送 GARP 刷新 Service IP 的 MAC 地址，从而保证高可用。
- 流量到达节点后，Kube-Proxy 会将流量转发到对应的 Pod。

选择 **VIP** 承载节点的算法

给定负载均衡器 IP 的“leader”（即将要广播该 IP 的节点）的选举是无状态的，工作方式如下：

- 每个 speaker 收集给定 IP 的潜在广播者列表，考虑活动的 speaker、外部流量策略、活动的 endpoints、节点选择器等因素。
- 每个 speaker 执行相同的计算：对“节点+VIP”元素的哈希值进行排序，获得一个有序列表，如果自己是列表中的第一个元素，则广播该服务。

这样就不需要记忆哪个 speaker 负责广播某个 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。这里 n 必须大于 2，允许最多一个节点故障。

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

应用示例

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

分析：

需要两个外部地址池和四个可用节点。

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

其他资源

- 创建外部 IP 地址池
- 创建 BGP Peers

Menu

核心概念

ALB 与 Ingress-NGINX 注解兼容性

- 基本概念
- 支持的 ingress-nginx 注解

Service、Ingress、Gateway API 与 ALB Rule 之间的比较

- 针对 L4（TCP/UDP）流量
- 针对 L7（HTTP/HTTPS）流量

GatewayAPI

- 通过 ALB

ALB 与 Ingress-NGINX 注解兼容性

目录

基本概念

支持的 ingress-nginx 注解

基本概念

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

ALB 支持其中部分注解。

支持的 ingress-nginx 注解

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

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

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

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

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

名称	类型	支持情况（**v** 支持 **x** 不支持 **o** 部分支持或可通过配置实现）
nginx.ingress.kubernetes.io/preserve-trailing-slash	"true" or "false"	x
nginx.ingress.kubernetes.io/proxy-body-size	string	x
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" or "1.1"	x

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

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

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

名称	类型	支持情况（**v** 支持 **x** 不支持 **o** 部分支持或可通过配置实现）
nginx.ingress.kubernetes.io/enable-modsecurity	bool	v modsecurity
nginx.ingress.kubernetes.io/enable-owasp-core-rules	bool	v modsecurity
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

Service、Ingress、Gateway API 与 ALB Rule 之间的比较

Alauda Container Platform 支持 Kubernetes 生态系统中的多种入口流量规范。
本文档对它们（Service、Ingress、Gateway API 和 ALB Rule）进行比较，帮助用户做出正确选择。

目录

针对 L4（TCP/UDP）流量

针对 L7（HTTP/HTTPS）流量

Ingress

GatewayAPI

ALB Rule

针对 L4（TCP/UDP）流量

LoadBalancer 类型的 Service、Gateway API 和 ALB Rule 都可以将 L4 流量暴露到外部。这里推荐使用 LoadBalancer 类型的 Service 方式。
Gateway API 和 ALB Rule 都是由 ALB 实现的，ALB 是一个用户空间代理，与 LoadBalancer 类型的 Service 相比，在处理 L4 流量时性能会显著下降。

针对 L7（HTTP/HTTPS）流量

Ingress、GatewayAPI 和 ALB Rule 都可以将 L7 流量暴露到外部，但它们在能力和隔离模型上有所不同。

Ingress

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

GatewayAPI

GatewayAPI 提供了更灵活的隔离模式，但其成熟度不及 Ingress。
通过使用 GatewayAPI，开发人员可以创建自己隔离的 ALB 实例来处理 GatewayAPI 规则。
因此，如果需要将 ALB 实例的创建和管理权限委托给开发人员，则需要选择使用 GatewayAPI。

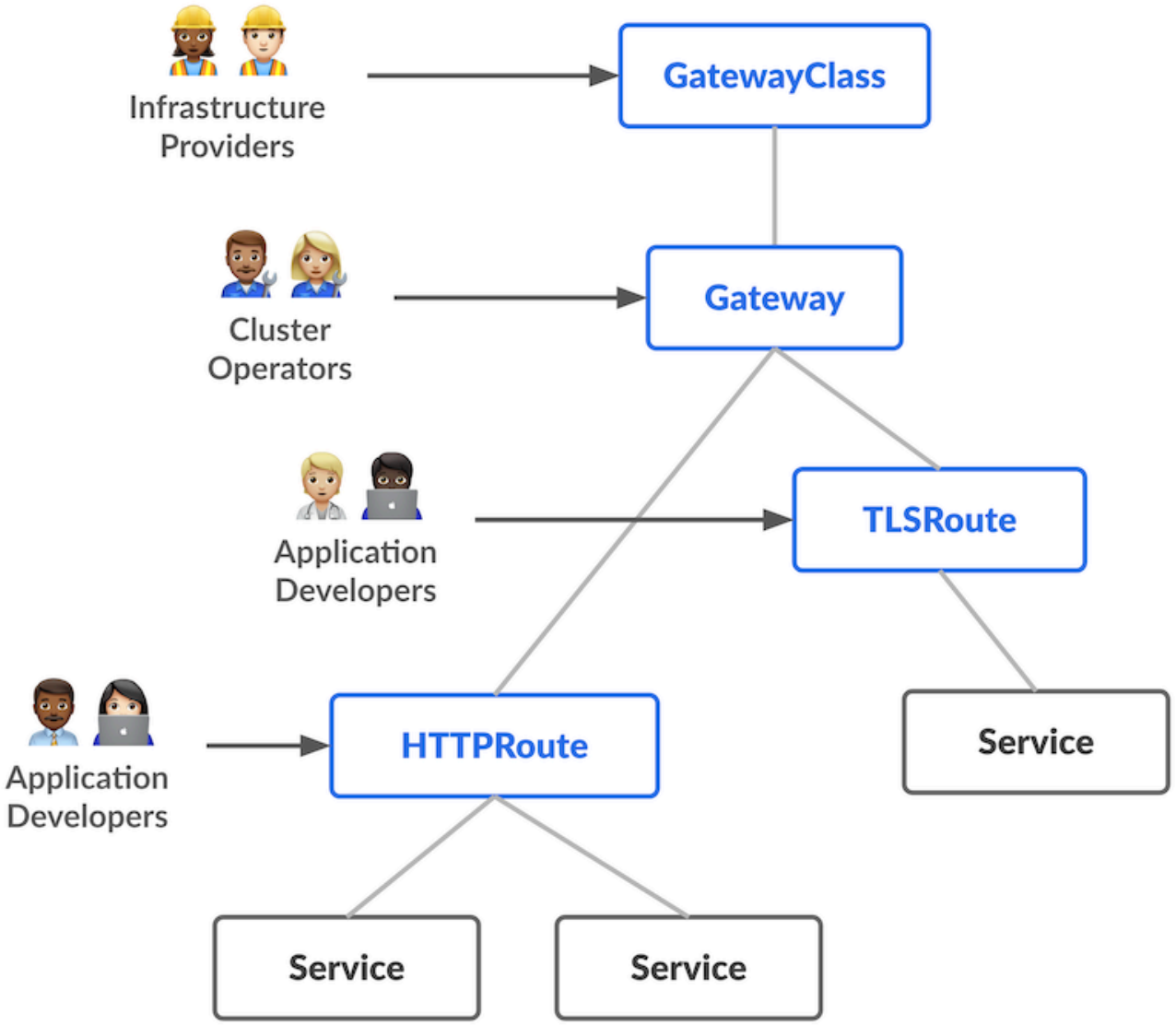
ALB Rule

ALB Rule（UI 中的 Load Balancer）提供了最灵活的流量匹配规则和最多的功能。实际上，Ingress 和 GatewayAPI 都是通过转换为 ALB Rule 来实现的。
然而，ALB Rule 比 Ingress 和 GatewayAPI 更复杂，且不是社区标准的 API。因此，建议仅在 Ingress 和 GatewayAPI 无法满足需求时使用。

GatewayAPI

GatewayAPI↗ 是一个官方的 Kubernetes 项目，专注于 Kubernetes 中的 L4 和 L7 路由。该项目代表了 Kubernetes Ingress、负载均衡和服务网格 API 的下一代。从一开始，它就被设计为通用、表达性强且面向角色的。

整体资源模型聚焦于 3 个不同的角色及其预期管理的相应资源：



目录

通过 **ALB**

ALB 支持 GatewayAPI。每个 Gateway 资源映射到一个 ALB 资源。 Listener 和路由由 ALB 直接处理；它们不会被转换为 `Frontend` 或 `Rule`。 通过 ALB 创建 GatewayAPI Gateway

Menu

功能指南

创建服务

- 为什么需要 Service
- ClusterIP 类型 Service 示例：
- Headless Service（无头服务）
- 通过 Web 控制台创建服务
- 通过 CLI 创建服务
- 示例：集群内访问应用
- 示例：集群外访问应用
- 示例：ExternalName 类型 Service
- LoadBalancer 类型 Service 注解

创建 **Ingress**

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

创建域名

- 域名自定义资源（CR）示例
- 通过 Web 控制台创建域名
- 通过 CLI 创建域名
- 后续操作
- 相关资源

创建证书

- 通过 Web 控制台创建证书

创建外部 **IP** 地址池

- 前提条件
- 约束与限制
- 部署 MetalLB 插件
- IPAddressPool 自定义资源（CR）示例
- 通过 Web 控制台创建外部 IP 地址池
- 通过 CLI 创建外部 IP 地址池
- 查看告警策略

创建 **BGP Peers**

- 术语
- 前提条件
- BGPPeer 自定义资源（CR）示例
- 通过 Web 控制台创建 BGPPeer
- 通过 CLI 创建 BGPPeer

配置子网

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

配置网络策略

- 通过 Web 控制台创建 NetworkPolicy
- 通过 CLI 创建 NetworkPolicy
- 参考

创建 **Admin** 网络策略

- 注意事项
- 通过 Web 控制台创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy
- 通过 CLI 创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy
- 其他资源

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

- 安装 Multus CNI
- 创建子网
- 创建多网卡 Pod
- 验证双网卡创建
- 其他功能

配置集群网络策略

- 注意事项
- 操作步骤

配置 **Egress Gateway**

- 关于 Egress Gateway
- 注意事项
- 使用方法
- 配置参数
- 相关资源

网络可观测性

- 关于 DeepFlow
- 安装 DeepFlow
- 其他资源

配置 **ALB** 规则

- 介绍
- 动作
- 后端
- 创建规则
- Https
- Ingress

集群互联（Alpha）

支持配置网络模式为 Kube-OVN 的集群之间的集群互联，实现集群间 Pod 的相互访问。

- 前提条件
- 多节点 Kube-OVN 互联控制器构建
- 在 global 集群部署集群互联控制器
- 加入集群互联
- 相关操作

Endpoint Health Checker

- Overview
- Key Features
- Installation
- How It Works
- Uninstallation

NodeLocal DNSCache

- Overview
- Key Features
- Important Notes
- Installation
- How It Works
- Configuration

创建服务

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

目录

为什么需要 Service

ClusterIP 类型 Service 示例：

Headless Service（无头服务）

通过 Web 控制台创建服务

通过 CLI 创建服务

示例：集群内访问应用

示例：集群外访问应用

示例：ExternalName 类型 Service

LoadBalancer 类型 Service 注解

AWS EKS 集群

华为云 CCE 集群

Azure AKS 集群

Google GKE 集群

为什么需要 **Service**

1. Pod 有自己的 IP，但：

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

2. Service 通过提供以下功能解决了这个问题：
   - 稳定的 IP 和 DNS 名称。
   - 自动负载均衡到匹配的 Pods。

ClusterIP 类型 Service 示例：

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

1. 可用的 type 值及其行为包括 `ClusterIP`、`NodePort`、`LoadBalancer`、`ExternalName`
2. Service 目标的 Pod 集合通常由你定义的 selector 决定。
3. `Service` 端口。
4. 将 Service 的 `targetPort` 绑定到 Pod 的 `containerPort`。此外，也可以引用 Pod 容器下的 `port.name`。

Headless Service（无头服务）

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

```
spec:
  clusterIP: None
```

无头服务适用于：

- 你想要发现单个 Pod 的 IP，而不仅仅是单一的服务 IP。
- 你需要直接连接到每个 Pod（例如，像 Cassandra 或 StatefulSet 这类数据库）。
- 你使用 StatefulSet，且每个 Pod 必须有稳定的 DNS 名称。

通过 **Web** 控制台创建服务

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

参数	说明
虚拟 **IP** 地址	如果启用，将为该 Service 分配一个 ClusterIP，可用于集群内的服务发现。如果禁用，则创建无头服务，通常用于 **StatefulSet**。
类型	• **ClusterIP**：在集群内部 IP 上暴露 Service。选择此值时，Service 只能从集群内部访问。 • **NodePort**：在每个节点的 IP 上以静态端口（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外部暴露 Service。Kubernetes 本身不直接提供负载均衡组件，你需要自行提供，或者将 Kubernetes 集群与云服务商集成。

参数	说明
目标组件	• **Workload**：Service 会将请求转发到特定的工作负载，匹配标签如 `project.cpaas.io/name: projectname` 和 `service.cpaas.io/name: deployment-name`。 • **Virtualization**：Service 会将请求转发到特定的虚拟机或虚拟机组。 • **Label Selector**：Service 会将请求转发到带有指定标签的某类工作负载，例如 `environment: release`。
端口	用于配置该 Service 的端口映射。以下示例中，集群内其他 Pod 可以通过虚拟 IP（如果启用）和 TCP 端口 *80* 调用该 Service；访问请求将被转发到目标组件 Pod 外部暴露的 TCP 端口 *6379* 或 *redis*。 • 协议：Service 使用的协议，支持的协议包括：`TCP`、`UDP`、`HTTP`、`HTTP2`、`HTTPS`、`gRPC`。 • **Service** 端口：Service 在集群内暴露的端口号，即 Port，例如 *80*。 • 容器端口：Service 端口映射到的目标端口号（或名称），即 targetPort，例如 *6379* 或 *redis*。 • **Service** 端口名称：会自动生成，格式为 `<protocol>-<service port>-<container port>`，例如：*tcp-80-6379* 或 *tcp-80-redis*。
会话亲和性	基于源 IP 地址（ClientIP）的会话亲和性。如果启用，来自同一 IP 地址的所有访问请求在负载均衡期间将保持在同一服务器上，确保同一客户端的请求被转发到同一服务器处理。

5. 点击 **Create**。

通过 CLI 创建服务

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

基于已有的部署资源 `my-app` 创建服务。

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

示例：集群内访问应用

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

1. 应用该 YAML：

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

2. 启动另一个 Pod：

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

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

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

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

示例：集群外访问应用

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

1. 应用该 YAML：

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

2. 查看 Pods：

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

3. curl 访问 Service：

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

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

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

注意：请提前配置 LoadBalancer 服务。

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

1. 应用该 YAML：

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

2. 获取外部 IP 地址：

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

```
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
nginx-service   LoadBalancer   10.0.2.57       34.122.45.100   80:30005/TCP   30s
```

`EXTERNAL-IP` 即为你从浏览器访问的地址。

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

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

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

示例：**ExternalName 类型 Service**

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

1. 应用该 YAML：

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

2. 在集群内的 Pod 中尝试解析：

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

然后执行：

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

你会看到它解析为 `example.com`。

LoadBalancer 类型 Service 注解

AWS EKS 集群

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

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

Key	Value	说明
	• dualstack	

华为云 CCE 集群

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

Key	
kubernetes.io/elb.id	

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

kubernetes.io/elb.subnet-id	

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

Azure AKS 集群

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

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

Google GKE 集群

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

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

创建 Ingress

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

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

WARNING

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

目录

实现方式

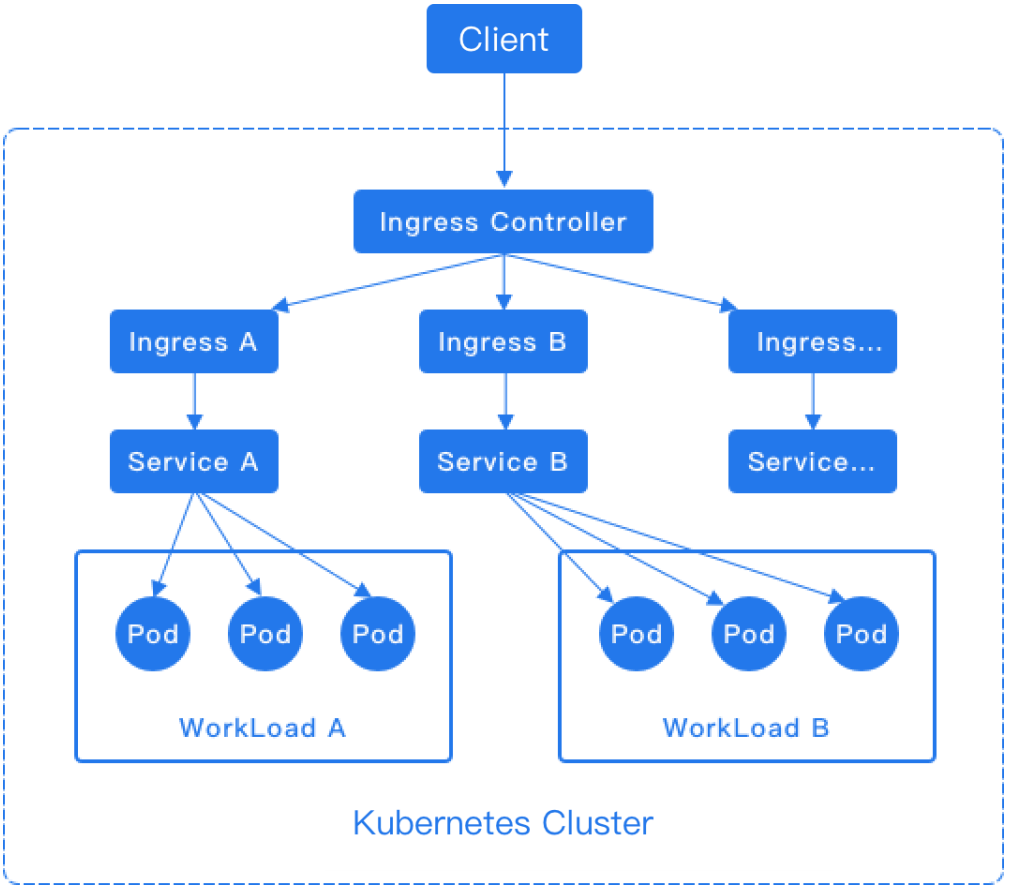
Ingress 示例：

通过 Web 控制台创建 Ingress

通过 CLI 创建 Ingress

实现方式

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



NOTE

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

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

Ingress 示例：

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

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

通过 Web 控制台创建 Ingress

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

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

5. 点击 **Create**。

通过 CLI 创建 Ingress

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

创建域名

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

NOTE

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

在平台上成功创建并分配的域名，可用于容器平台的以下功能：

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

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

目录

域名自定义资源（CR）示例

通过 Web 控制台创建域名

通过 CLI 创建域名

后续操作

相关资源

域名自定义资源（CR）示例

```yaml
# test-domain.yaml
apiVersion: crd.alauda.io/v2
kind: Domain
metadata:
  name: "000000000030755752601296 86e67ed4-917a-454a-8553-d55fc4030f81"
  annotations:
    cpaas.io/secret-ref: developer.test.cn-xfd8x (1)
  labels:
    cluster.cpaas.io/name: global
    project.cpaas.io/name: cong
spec:
  name: developer.test.cn
  kind: full
```

1. 若启用证书，需提前创建 LTS 类型的 Secret，`secret-ref` 为 Secret 名称。

通过 Web 控制台创建域名

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

参数	说明
类型	• Domain：完整域名，例如 `developer.test.cn`。 • Wildcard Domain：带有通配符（*）的泛域名，例如 `*.test.cn`，包含域名 `test.cn` 下的所有子域名。
域名	根据选择的域名类型，输入完整域名或域名后缀。

参数	说明
分配集群	若分配集群，还需选择与该集群关联的项目，如该集群下的所有项目。
证书	包含用于创建域名绑定证书的公钥（tls.crt）和私钥（tls.key）。证书分配的项目与绑定的域名相同。 注意： • 不支持二进制文件导入。 • 绑定的证书需满足格式正确、在有效期内且为该域名签发等条件。 • 创建绑定证书后，绑定证书的名称格式为：域名-随机字符。 • 创建绑定证书后，可在证书列表中查看，但绑定证书的更新和删除仅支持在域名详情页操作。 • 创建绑定证书后，支持更新证书内容，但不支持替换为其他证书。

5. 点击 创建。

通过 CLI 创建域名

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

后续操作

- 域名注册：若创建的域名尚未注册，请进行域名注册。
- 域名解析：若域名未指向平台集群的负载均衡地址，请进行域名解析。

相关资源

- 配置证书

创建证书

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

NOTE

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

目录

通过 Web 控制台创建证书

通过 **Web** 控制台创建证书

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

参数	说明
Assign Project	• 所有项目：将证书分配给当前集群关联的所有项目使用。 • 指定项目：将证书分配给指定项目使用。 • 不分配：暂不分配项目。证书创建完成后，可通过 **Update Project** 操作更新可使用该证书的项目。
Public Key	指 tls.crt。导入公钥时，不支持二进制文件。
Private Key	指 tls.key。导入私钥时，不支持二进制文件。

5. 点击 **Create**。

创建外部 IP 地址池

外部 IP 地址池是 MetalLB 用于获取 LoadBalancer 类型内部路由的外部访问 IP 的 IP 集合。

目录

前提条件

约束与限制

部署 MetalLB 插件

IPAddressPool 自定义资源（CR）示例

通过 Web 控制台创建外部 IP 地址池

通过 CLI 创建外部 IP 地址池

查看告警策略

前提条件

如果需要使用 BGP 类型的外部 IP 地址池，请联系管理员开启相关功能。

约束与限制

外部地址的 IP 资源必须满足以下条件：

- 外部地址池必须与可用节点实现二层（L2）互联。

- IP 必须可被平台使用，且不能包含物理网络已使用的 IP，如网关 IP。
- 不能与集群使用的网络重叠，包括 Cluster CIDR、Service CIDR、子网等。
- 在双栈环境中，确保同一外部地址池中同时存在 IPv4 和 IPv6 地址，且数量均大于 0，否则双栈 LoadBalancer 类型内部路由无法获取外部访问地址。
- 在 IPv6 环境中，节点的 DNS 必须支持 IPv6，否则 MetalLB 插件无法成功部署。

部署 **MetalLB** 插件

使用外部地址池依赖 MetalLB 插件。

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Marketplace** > **Cluster Plugin**。
3. 搜索 MetalLB，点击⋮右侧的 **MetalLB** > **Deploy**。
4. 等待部署状态显示 **Deployment Successful** 即完成部署。

IPAddressPool 自定义资源（**CR**）示例

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

BGP 模式：

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

1. IP 地址池引用说明。

INFO

问：什么是 `L2Advertisement` ？

答：

1. `L2Advertisement` 是 MetalLB 提供的自定义资源（CRD），用于控制在二层模式下通过 ARP（IPv4）或 NDP（IPv6）广播哪些 IP 地址池中的地址。

问：`L2Advertisement` 的作用是什么？

答：

1. 指定 IPAddressPool 中哪些 IP 地址需要进行二层广播（ARP/NDP 广播）；
2. 控制广播行为，防止 IP 冲突或跨网段广播；
3. 限制多网卡、多网络环境中的广播范围。

简而言之，它告诉 MetalLB：哪些 IP 可以广播，广播给谁（例如哪些节点）。

在二层模式下，如果未定义 `L2Advertisement`，MetalLB 不会广播任何地址。

问：MetalLB 中的 `BGPAdvertisement` 是什么？

答：

`BGPAdvertisement` 是 Kubernetes 的自定义资源定义（CRD），用于 MetalLB↗ 中，控制如何通过 BGP（边界网关协议）向外部网络广播 IP 地址范围（定义在 `IPAddressPool` 中）。

问：为什么 `BGPAdvertisement` 很重要？

答：

在 MetalLB 的 BGP 模式中，控制器通过 BGP 与外部路由器建立对等连接，并广播分配给 Kubernetes `Service` 对象的 IP。`BGPAdvertisement` 资源允许你：

- 控制广播哪些地址池
- 自定义路由广播设置，如：
  - 路由聚合
  - BGP 社区属性
  - 本地优先级（BGP 优先级）

如果未定义 `BGPAdvertisement`，即使配置了 BGP 对等体，MetalLB 也不会广播任何地址。

通过 Web 控制台创建外部 IP 地址池

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

参数	说明
Type	• L2：基于 MAC 地址的通信和转发，适用于需要简单快速二层交换的小规模或局域网，优点是配置简单、延迟低。

参数	说明
	• BGP（Alpha）：基于 IP 地址的路由和转发，使用 BGP 协议交换路由信息，适合需要跨多个自治系统复杂路由的大规模网络，优点是高扩展性和可靠性。
IP Resources	支持 CIDR 和 IP 范围格式输入。点击 **Add** 支持多条输入，示例如下： **CIDR**：`192.168.1.1/24`。 **IP 范围**：`192.168.2.1` ~ `192.168.2.255`。
Available Nodes	L2 模式下，可用节点是承载所有 VIP 流量的节点；BGP 模式下，可用节点是承载 VIP、与对等体建立 BGP 连接并对外宣布路由的节点。 • 节点名称：根据节点名称选择可用节点。 • 标签选择器：根据标签选择可用节点。 • 显示节点详情：以列表形式查看最终可用节点。 注意： • 使用 BGP 类型时，可用节点即下一跳节点；确保所选可用节点是 BGP 连接节点 的子集。 • 标签选择器和节点名称可单独配置，若同时配置，最终可用节点为两者的交集。
BGP Peers	选择 BGP 对等体；具体配置请参考 BGP Peers。

5. 点击 **Create**。

通过 CLI 创建外部 IP 地址池

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

查看告警策略

1. 进入 **Administrator**。

2. 在左侧导航栏点击 **Network Management** > **External IP Address Pool**。

3. 点击页面右上角的 **View Alarm Policy**，查看 MetalLB 的通用告警策略。

创建 BGP Peers

节点通过 BGP 协议建立连接，以便在不同 AS 之间或同一 AS 内交换路由信息。

目录

术语
前提条件
BGPPeer 自定义资源（CR）示例
通过 Web 控制台创建 BGPPeer
通过 CLI 创建 BGPPeer

术语

术语	说明
AS Number	AS 指由同一技术管理组织管理、使用统一路由策略的一组路由器。BGP 网络中的每个 AS 都分配有唯一的 AS 号，用于区分不同的 AS。AS 号分为 2 字节 AS 号和 4 字节 AS 号。 • 2 字节 AS 号范围为 1~65535，其中 1~64511 是互联网上注册的公共 AS 号，类似于公共 IP 地址；64512~65535 是私有 AS 号，类似于私有 IP 地址。 • 4 字节 AS 号范围为 1~4294967295。 支持 4 字节 AS 号的设备可以兼容支持 2 字节 AS 号的设备。

前提条件

请联系管理员启用相关功能。

BGPPeer 自定义资源（CR）示例

```
# test-bgb-example.yaml
apiVersion: metallb.io/v1beta2
kind: BGPPeer
metadata:
  name: example
  namespace: metallb-system
spec:
  myASN: 64512
  peerASN: 64512
  peerAddress: 172.30.0.3
  peerPort: 180
  nodeSelectors:
    - matchLabels:
        alertmanager: "true"
```

通过 Web 控制台创建 BGPPeer

1. 进入 **Administrator**。
2. 在左侧导航栏中，点击 **Network Management** > **BGP Peers**。
3. 点击 **Create BGP Peer**。
4. 参考以下说明配置参数。

参数	说明
Local AS Number	BGP 连接节点所在 AS 的 AS 号。

参数	说明
	注意：如无特殊需求，建议使用 IBGP 配置，即本地 AS 号应与对端 AS 号保持一致。
Peer AS Number	BGP 对端所在 AS 的 AS 号。
Peer IP	BGP 对端的 IP 地址，必须是能够建立 BGP 连接的有效 IP 地址。
Local IP	BGP 连接节点的 IP 地址。当 BGP 连接节点有多个 IP 时，选择指定的本地 IP 与对端建立 BGP 连接。
Peer Port	BGP 对端的端口号。
BGP Connected Node	建立 BGP 连接的节点。如果未配置此参数，则所有节点均会建立 BGP 连接。
eBGP Multi-Hop	允许 BGP 路由器之间建立非直接连接的 BGP 会话。启用该功能时，BGP 包的默认 TTL 值为 5，允许跨越多个中间网络设备建立 BGP 对等关系，使网络设计更灵活。
RouterID	一个 32 位数值（通常以点分十进制格式表示，类似 IPv4 地址格式），用于唯一标识 BGP 网络中的 BGP 路由器，通常用于建立 BGP 邻居关系、检测路由环路、选择最优路径及排查网络问题。

5. 点击 **Create**。

通过 **CLI** 创建 **BGPPeer**

```
kubectl apply -f test-bgb-example.yaml
```

配置子网

目录

IP 分配规则

Calico 网络

约束与限制

Calico 网络子网示例自定义资源（CR）

通过 Web 控制台创建 Calico 网络子网

通过 CLI 创建 Calico 网络子网

参考内容

Kube-OVN 网络

Kube-OVN Overlay 网络子网示例自定义资源（CR）

通过 Web 控制台创建 Kube-OVN Overlay 网络子网

通过 CLI 创建 Kube-OVN Overlay 网络子网

Underlay 网络

使用说明

通过 Web 控制台添加桥接网络（可选）

通过 CLI 添加桥接网络

通过 Web 控制台添加 VLAN（可选）

通过 CLI 添加 VLAN

Kube-OVN Underlay 网络子网示例自定义资源（CR）

通过 Web 控制台创建 Kube-OVN Underlay 网络子网

通过 CLI 创建 Kube-OVN Underlay 网络子网

相关操作

子网管理

通过 Web 控制台更新网关

通过 CLI 更新网关

通过 Web 控制台更新保留 IP

通过 CLI 更新保留 IP

通过 Web 控制台分配项目

通过 CLI 分配项目

通过 Web 控制台分配命名空间

通过 CLI 分配命名空间

通过 Web 控制台扩容子网

通过 CLI 扩容子网

管理 Calico 网络

通过 Web 控制台删除子网

通过 CLI 删除子网

IP 分配规则

NOTE

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

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

- 如果命名空间绑定了子网，则该命名空间中的 Pod 只能使用该特定子网的 IP 地址。

Calico 网络

在 Calico 网络中创建子网，实现集群内资源更细粒度的网络隔离。

约束与限制

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

Calico 网络子网示例自定义资源（CR）

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

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

通过 Web 控制台创建 Calico 网络子网

1. 进入 **Administrator**。

2. 在左侧导航栏点击 **Network Management** > **Subnets**。

3. 点击 **Create Subnet**。

4. 按照以下说明配置相关参数。

参数	说明
CIDR	将子网分配给项目或命名空间后，该命名空间内的容器组将随机使用该 CIDR 范围内的 IP 进行通信。 注意：CIDR 与 BlockSize 的对应关系请参考参考内容。
Encapsulation Protocol	选择封装协议。双栈模式下不支持 **IPIP**。 • **IPIP**：使用 IPIP 协议实现跨段通信。 • **VXLAN (Alpha)**：使用 VXLAN 协议实现跨段通信。 • **No Encapsulation**：通过路由转发直接连接。
Encapsulation Mode	当封装协议为 **IPIP** 或 **VXLAN** 时，必须设置封装模式，默认为 **Always**。 • **Always**：始终启用 IPIP / VXLAN 隧道。 • **Cross Subnet**：仅当主机处于不同子网时启用 IPIP / VXLAN 隧道；同子网时通过路由转发直接连接。
Outbound Traffic NAT	选择是否启用出站流量 NAT（网络地址转换），默认启用。 主要用于设置子网容器组访问外网时暴露的访问地址。 启用出站流量 NAT 时，当前子网容器组的访问地址为宿主机 IP；未启用时，子网内容器组的 IP 将直接暴露给外网。

5. 点击 **Confirm**。

6. 在子网详情页，选择 **Actions** > **Allocate Project / Allocate Namespace**。

7. 完成配置后点击 **Allocate**。

通过 CLI 创建 Calico 网络子网

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

参考内容

CIDR 与 blockSize 的动态匹配关系如下表所示。

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

NOTE

不支持前缀大于 31 的子网配置。

Kube-OVN 网络

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

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

Kube-OVN Overlay 网络子网示例自定义资源（CR）

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

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

通过 Web 控制台创建 Kube-OVN Overlay 网络子网

1. 进入 **Administrator**。

2. 在左侧导航栏点击 **Network Management** > **Subnet**。

3. 点击 **Create Subnet**。

4. 按照以下说明配置相关参数。

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

5. 点击 **Confirm**。

6. 在子网详情页，选择 **Actions** > **Allocate Project / Namespace**。

7. 完成配置后点击 **Allocate**。

通过 CLI 创建 Kube-OVN Overlay 网络子网

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

Underlay 网络

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

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

使用说明

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

1. 默认网卡名称。
2. 按节点配置网卡。

通过 Web 控制台添加桥接网络（可选）

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

1. 默认网卡名称。
2. 按节点配置网卡。

桥接网络指桥接设备，绑定网卡后可转发容器网络流量，实现与物理网络的互通。

操作步骤：

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

   注意：

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

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

5. 点击 **Add**。

通过 CLI 添加桥接网络

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

通过 Web 控制台添加 VLAN（可选）

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

1. VLAN ID。
2. 桥接网络引用。

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

操作步骤：

1. 进入 **Administrator**。

2. 在左侧导航栏点击 **Network Management** > **VLAN**。

3. 点击 **Add VLAN**。

4. 根据以下说明配置相关参数。

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

5. 点击 **Add**。

通过 CLI 添加 VLAN

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

Kube-OVN Underlay 网络子网示例自定义资源（CR）

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

1. VLAN 引用。

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

> **NOTE**
>
> 平台还预配置了用于 Overlay 传输模式下节点与 Pod 通信的 **join** 子网，该子网不会用于 Underlay 传输模式，务必避免 **join** 与其他子网的 IP 段冲突。

操作步骤：

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

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

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

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

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

相关操作

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

子网管理

通过 **Web** 控制台更新网关

包括更改出站流量方式、网关节点和 NAT 配置。

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

通过 **CLI** 更新网关

```
kubectl patch subnet test-overlay-subnet --type=json -p='[
  {"op": "replace", "path": "/spec/gatewayType", "value": "centralized"},
  {"op": "replace", "path": "/spec/gatewayNode", "value": "192.168.66.210"},
  {"op": "replace", "path": "/spec/natOutgoing", "value": true},
  {"op": "replace", "path": "/spec/enableEcmp", "value": true}
]'
```

通过 Web 控制台更新保留 IP

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

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

通过 CLI 更新保留 IP

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

通过 Web 控制台分配项目

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

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

通过 CLI 分配项目

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

通过 Web 控制台分配命名空间

将子网分配给特定命名空间，实现更细粒度的网络隔离。

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

1. 进入 **Administrator**。
2. 在左侧边栏点击 **Network Management** > **Subnets**。
3. 点击子网名称。
4. 选择 **Action** > **Assign Namespace**。
5. 添加或移除命名空间后，点击 **Assign**。

通过 CLI 分配命名空间

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

通过 Web 控制台扩容子网

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

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

通过 CLI 扩容子网

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

管理 Calico 网络

支持项目和命名空间分配，详情请参考项目分配和命名空间分配。

通过 **Web** 控制台删除子网

NOTE

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

1. 进入 **Administrator**。

2. 在左侧导航栏点击 **Network Management** > **Subnets**。

3. 点击⋮> **Delete**，执行删除操作。

通过 **CLI** 删除子网

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

创建网络策略

INFO

平台现提供两种不同的网络策略 UI。旧版 UI 为兼容性维护而保留，新版 UI 更加灵活，并提供原生 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` 功能门控，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

网络策略仅适用于容器流量。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 选择器		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		阻止所有到目标 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。
	规则	当前命名空间	匹配当前命名空间内指定标签的 Pod；只有匹配的 Pod 可以访问目标 Pod。您可以点击 **Preview**

区域	参数		说明
	说明：规则中添加多个来源时，它们之间是逻辑上的 或 关系。	内的 Pod	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，并可排除不允许访问的 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 *example_ip*/32 形式添加排除项，以排除单个 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		阻止所有到目标 Pod 的出口流量。 注意: • 如果在 YAML 的 `spec.policyTypes` 字段中添加了 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↗。

创建 Admin 网络策略

INFO

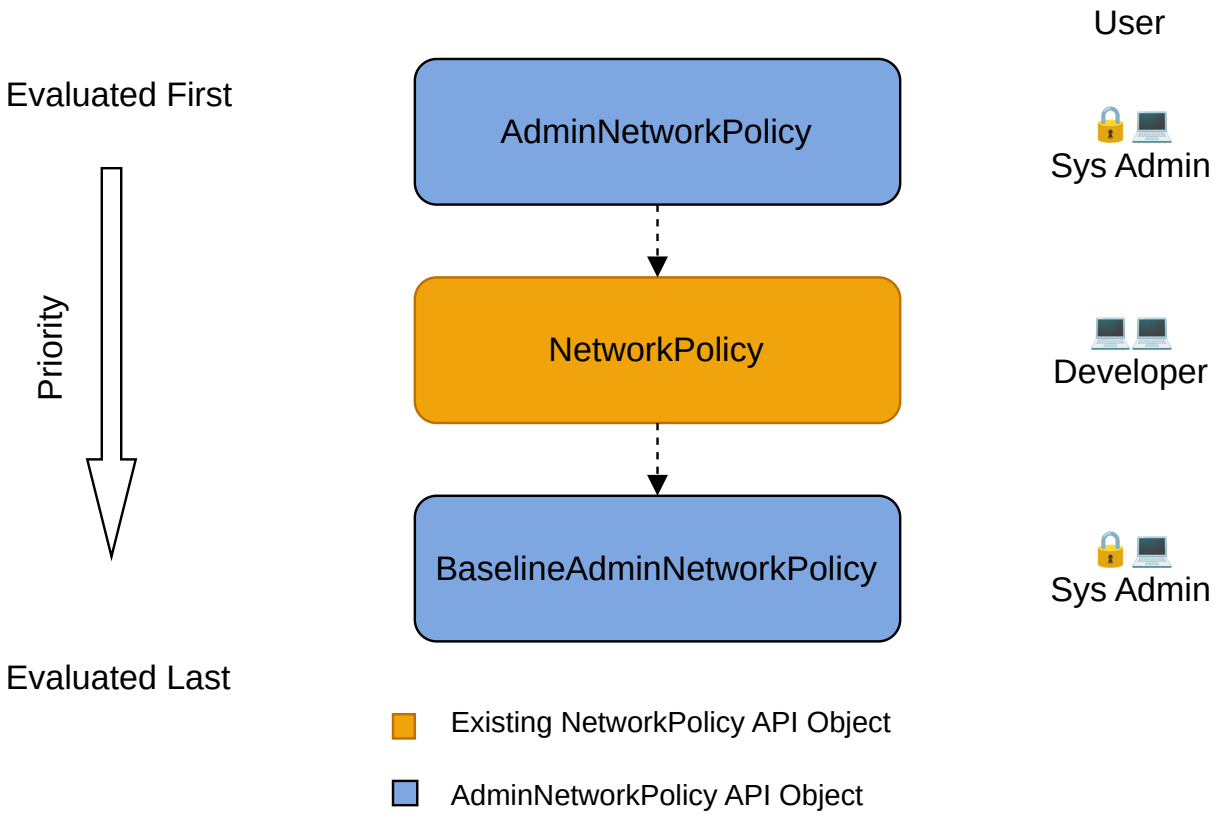
平台目前提供了两种不同的集群网络策略 UI。旧版 UI 为兼容性保留，新版 UI 更加灵活，并提供了原生 YAML 编辑器。我们推荐使用新版 UI。

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

新版集群网络策略采用 Kubernetes 社区的 Admin Network Policy↗ 标准设计，提供更灵活的配置方式和丰富的配置选项。

当多个网络策略同时生效时，遵循严格的优先级顺序：Admin Network Policy 优先于 Network Policy，Network Policy 优先于 Baseline Admin Network Policy。

操作步骤如下：



目录

注意事项
通过 Web 控制台创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy
通过 CLI 创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy
其他资源

注意事项

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

AdminNetworkPolicy 示例

```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`：最多只能指定 namespace selector 或 pod selector 中的一个。

3. `action`：可选值为 Allow、Deny 和 Pass。 Allow 表示允许流量访问，Deny 表示拒绝流量访问，Pass 表示允许流量并跳过后续优先级较低的集群网络策略，由其他策略（NetworkPolicy 和 BaselineAdminNetworkPolicy）继续处理该流量。
4. 可选值包括 Namespace Selector、Pod Selector。
5. 可选值包括 Namespace Selector、Pod Selector、Node Selector、IP Block。

BaselineAdminNetworkPolicy 示例：

```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 为 `default` 的 baseline admin 网络策略。

2. 可选值为 Allow、Deny。

通过 **Web** 控制台创建 **AdminNetworkPolicy** 或 **BaselineAdminNetworkPolicy**

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Network** > **Cluster Network Policies**。
3. 点击 **Create Admin Network Policies** 或 **Configure the Baseline Admin Network Policy**。
4. 按照以下说明完成相关配置。

区域	参数	说明
基本信息	名称	Admin Network Policy 或 Baseline Admin Network Policy 的名称。
	优先级	决定策略的评估和应用顺序，数值越小优先级越高。 注意：Baseline admin 网络策略无优先级设置。
目标 Pod	Namespace Selector	以键值对形式输入目标 Namespace 的标签，若不设置，则策略应用于当前集群所有 Namespace。指定后，策略仅应用于匹配这些选择器的 Namespace 内的 Pod。
	当前策略影响的目标 Pod 预览	点击 预览 查看该网络策略影响的目标 Pod。
	Pod Selector	以键值对形式输入目标 Pod 的标签，若不设置，则策略应用于当前 Namespace 下所有 Pod。
	当前策略影响的目标 Pod 预览	点击 预览 查看该网络策略影响的目标 Pod。

区域	参数		说明
Ingress	流量动作		指定如何处理流入目标 Pod 的流量，有三种模式：**Allow**（允许流量）、**Deny**（拒绝流量）、**Pass**（跳过所有优先级较低的 admin 网络策略，由 Network Policy 或无 Network Policy 时由 Baseline Admin Network Policy 处理流量）。 注意：Baseline admin 网络策略不支持动作 **Pass**。
	规则 说明：规则中添加多个来源时，它们之间是逻辑 或 关系。	Pod Selector	匹配集群中带有指定标签的 Namespace 或 Pod，只有匹配的 Pod 能访问目标 Pod。可点击 预览 查看当前规则影响的 Pod。 • 若同时配置了 namespace 和 Pod 选择器，则取交集，即从指定的 Namespace 中选择带有指定标签的 Pod。 • 若未配置此项，默认允许集群所有 Namespace 中的所有 Pod 访问目标 Pod。
		Namespace Selector	匹配当前 Namespace 中带有指定标签的 Pod，只有匹配的 Pod 能访问目标 Pod。可点击 预览 查看当前规则影响的 Pod。若未配置此项，默认允许当前 Namespace 中所有 Pod 访问目标 Pod。
	端口		匹配指定协议和端口的流量；可添加数字端口或 Pod 上的端口名称。若未配置此项，则匹配所有端口。

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

通过 CLI 创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy

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

其他资源

- 配置集群网络策略

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

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

目录

安装 Multus CNI

部署 Multus CNI 插件

创建子网

创建多网卡 Pod

验证双网卡创建

其他功能

固定 IP

额外路由

安装 **Multus CNI**

部署 **Multus CNI** 插件

1. 进入 管理员。

2. 在左侧导航栏点击 **Marketplace** > **Cluster Plugins**。

3. 在搜索框输入 “multus” 查找 Multus CNI 插件。

4. 在列表中找到 **"Alauda Container Platform Networking for Multus"** 插件。

5. 点击插件条目旁的三点（⋮），选择 安装。

6. 插件将部署到您的集群中，您可以在 状态 列监控安装进度。

> **NOTE**
>
> Multus CNI 插件作为其他 CNI 插件与 Kubernetes 之间的中间件，使 Pod 支持多个网络接口。

创建子网

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

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

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

创建后，应用该资源：

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

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

NOTE

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

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

创建后，应用该资源：

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

创建多网卡 Pod

按照以下示例创建 Pod。

NOTE

- `metadata.annotations` 中必须包含键值对 `k8s.v1.cni.cncf.io/networks=default/attachnet`，其中值的格式为 `<NAMESPACE>/<NAME>`，`<NAMESPACE>` 和 `<NAME>` 分别是 NetworkAttachmentDefinition CR 的命名空间和名称。

- 若 Pod 需要三个网络接口，配置 `k8s.v1.cni.cncf.io/networks` 的值为 `default/attachnet,default/attachnet2`。

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

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

验证双网卡创建

使用以下命令验证双网卡是否创建成功：

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

其他功能

固定 IP

- 主网卡（第一个接口）：若需固定主网卡 IP，方法与单网卡固定 IP 相同，在 Pod 上添加注解 `ovn.kubernetes.io/ip_address=<IP>`。
- 次网卡（第二个或其他接口）：基本方法与主网卡类似，区别是注解键中的 `ovn` 替换为对应 NetworkAttachmentDefinition 的 provider。例如：`attachnet.default.ovn.kubernetes.io/ip_address=172.170.0.101`。

额外路由

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

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

配置集群网络策略

集群网络策略负责管理项目级别的访问控制规则。启用该功能后，不同项目之间默认相互隔离，不同项目中的计算组件无法通过网络互相访问。可以通过添加项目间访问或使用 **IP** 地址访问规则实现通信。

配置完成后，集群网络策略将同步到集群下的命名空间中，可在容器平台的网络策略功能模块中查看。

目录

注意事项

操作步骤

注意事项

- 集群网络策略的生效依赖于集群所使用的网络插件是否支持网络策略。
  - Kube-OVN 和 Calico 支持网络策略。
  - Flannel 不支持网络策略。
  - 访问集群或使用自定义网络插件时，可参考相关文档确认支持情况。
- 该功能在 Kube-OVN 网络模式下处于 Alpha 版本成熟度。

操作步骤

1. 进入 管理员。

2. 在左侧导航栏点击 网络 > 集群网络策略。

3. 点击 立即配置。

4. 按照以下说明完成相关配置。

配置项	说明
项目间完全隔离	是否开启项目间完全隔离开关，默认开启，点击可关闭。开启后，当前集群内所有项目之间实现网络隔离，其他资源不允许访问集群内任何项目（例如外部 IP、负载均衡器）。不影响项目访问集群外部资源。
单项目访问	该参数仅在开启项目间完全隔离开关时生效。 配置单向访问的源项目和目标项目。 点击添加新增配置记录，支持多条。 在源项目下拉框中选择将访问目标项目的项目或选择所有项目；在目标项目下拉框中选择被访问的目标项目。
IP 段访问	该参数仅在开启项目间完全隔离开关时生效。 配置单向访问的具体**IP/**段和目标项目。 点击添加新增配置记录，支持多条。 在源 **IP** 段输入框中填写访问目标项目的 IP 或 CIDR 段；在目标项目下拉框中选择被访问的目标项目。

5. 点击 配置。

配置 **Egress Gateway**

目录

关于 Egress Gateway

实现细节

注意事项

使用方法

创建 Network Attachment Definition

创建 VPC Egress Gateway

启用基于 BFD 的高可用

配置参数

VPC BFD Port

VPC Egress Gateway

相关资源

关于 **Egress Gateway**

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

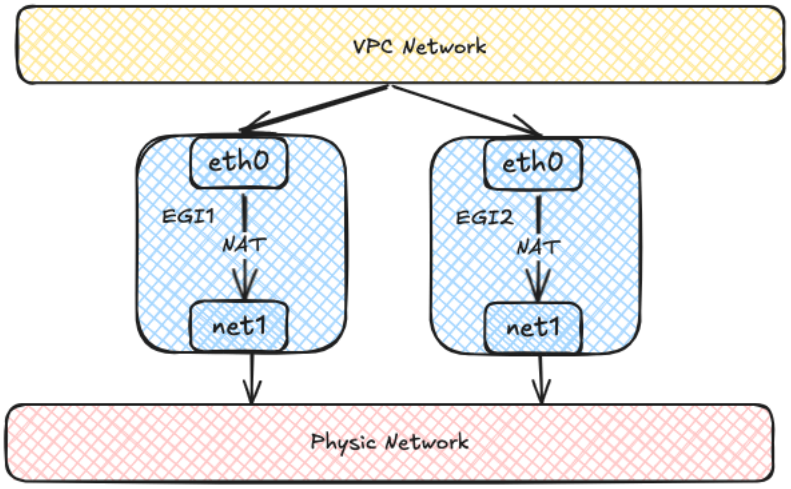
- 通过 ECMP 实现 Active-Active 高可用，支持水平吞吐量扩展
- 通过 BFD 实现快速故障切换（<1秒）
- 支持 IPv6 和双栈
- 通过 NamespaceSelector 和 PodSelector 实现细粒度路由控制
- 通过 NodeSelector 实现 Egress Gateway 的灵活调度

同时，Egress Gateway 具有以下限制：

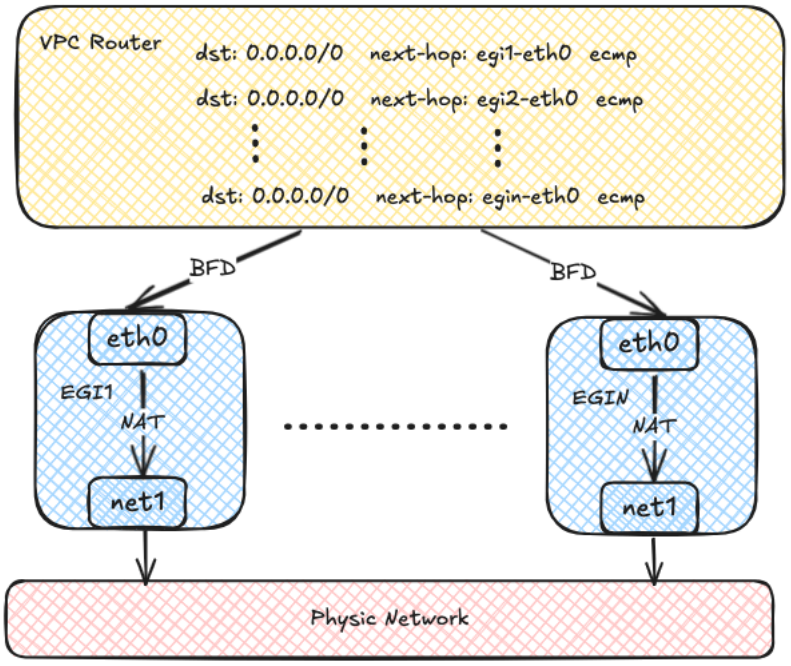
- 使用 macvlan 作为底层网络连接，要求底层网络支持 Underlay
- 多实例 Gateway 模式下，需要多个 Egress IP
- 目前仅支持 SNAT，不支持 EIP 和 DNAT
- 目前不支持记录源地址转换关系

实现细节

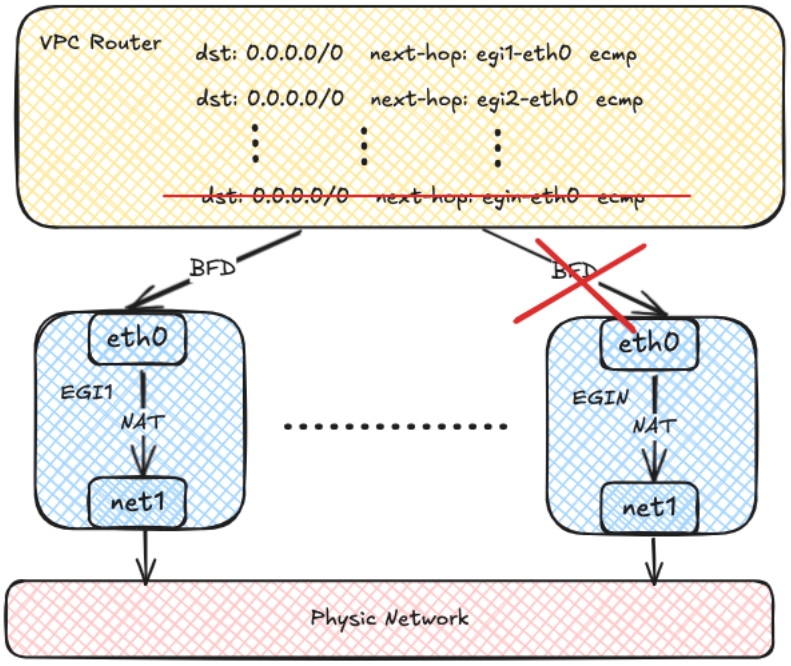
每个 Egress Gateway 由多个具有多个网络接口的 Pod 组成。 每个 Pod 有两个网络接口：一个加入虚拟网络，用于 VPC 内通信， 另一个通过 Macvlan 连接到底层物理网络，用于外部网络通信。 虚拟网络流量最终通过 Egress Gateway 实例内的 NAT 访问外部网络。



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



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



注意事项

- 仅 Kube-OVN CNI 支持 Egress Gateway。
- Egress Gateway 需要 Multus-CNI。

使用方法

创建 **Network Attachment Definition**

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

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

1. 连接外部网络的宿主机接口。
2. Provider 名称，格式为 `<network attachment definition name>.<namespace>`。
3. 用于标识外部网络的 Provider 名称，必须与 NetworkAttachmentDefinition 中一致。

> 提示
>
> 你可以使用任意 CNI 插件创建 Network Attachment Definition 来访问对应网络。

创建 VPC Egress Gateway

创建 VPC Egress Gateway 资源，示例如下：

```yaml
apiVersion: kubeovn.io/v1
kind: VpcEgressGateway
metadata:
  name: gateway1
  namespace: default (1)
spec:
  replicas: 1 (2)
  externalSubnet: macvlan1 (3)
  nodeSelector: (4)
    - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
            - kube-ovn-worker
            - kube-ovn-worker2
  selectors: (5)
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: default
  policies: (6)
    - snat: true (7)
      subnets: (8)
        - subnet1
    - snat: false
      ipBlocks: (9)
        - 10.18.0.0/16
```

1. 创建 VPC Egress Gateway 实例的命名空间。
2. VPC Egress Gateway 实例的副本数。
3. 连接外部网络的外部子网。
4. VPC Egress Gateway 适用的节点选择器。
5. VPC Egress Gateway 适用的命名空间和 Pod 选择器。
6. VPC Egress Gateway 的策略，包括 SNAT 及适用的子网/IP 段。
7. 是否为该策略启用 SNAT。
8. 策略适用的子网。
9. 策略适用的 IP 段。

上述资源会在 default 命名空间下创建名为 *gateway1* 的 VPC Egress Gateway， 以下 Pod 会通过 *macvlan1* 子网访问外部网络：

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

> 注意
>
> 匹配 *.spec.selectors* 的 Pod 始终启用 SNAT 访问外部网络。

创建完成后，查看 VPC Egress Gateway 资源：

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

查看更多信息：

```
kubectl get veg gateway1 -o wide
NAME       VPC           REPLICAS   BFD ENABLED   EXTERNAL SUBNET   PHASE       READY   INTERNAL IPS     EXTERNAL IPS      WORKING NODES
AGE
gateway1   ovn-cluster   1          false         macvlan1          Completed   true    ["10.16.0.12"]   ["172.17.0.11"]   ["kube-ovn-
worker"]   82s
```

查看工作负载：

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

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

查看 Pod 内的 IP 地址、路由和 iptables 规则：

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

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

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

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

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

在 Gateway Pod 中抓包验证网络流量：

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

OVN Logical Router 上自动创建路由策略：

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

1. VPC Egress Gateway 用于转发 *.spec.policies* 指定 Pod 流量的逻辑路由器策略。
2. VPC Egress Gateway 用于转发 *.spec.selectors* 指定 Pod 流量的逻辑路由器策略。

如果需要启用负载均衡，修改 *.spec.replicas*，示例如下：

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

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

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

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

启用基于 **BFD** 的高可用

基于 BFD 的高可用依赖于 VPC BFD LRP 功能， 因此需要修改 VPC 资源以启用 BFD Port。 以下示例为默认 VPC 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，必须是有效且不与其他 IP/子网冲突的地址。
3. 用于选择运行 BFD Port 的节点的节点选择器，BFD Port 绑定选中节点的 OVN HA Chassis Group，以 Active/Backup 模式工作。

启用 BFD Port 后，会在对应的 OVN Logical Router 上自动创建专用的 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。

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

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

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

查看 VPC Egress Gateway 信息：

```
$ kubectl get veg gateway2 -o wide
NAME       VPC    REPLICAS   BFD ENABLED   EXTERNAL SUBNET   PHASE       READY   INTERNAL IPS                    EXTERNAL IPS
WORKING NODES                           AGE
gateway2   vpc1   2          true          macvlan           Completed   true    ["10.16.0.102","10.16.0.103"]
["172.17.0.13","172.17.0.14"]   ["kube-ovn-worker","kube-ovn-worker2"]   58s

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

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

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

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

查看 BFD 连接状态：

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

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

注意

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

配置参数

VPC BFD Port

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

VPC Egress Gateway

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

字段			类型	可选	默认值	描述	
internalSubnet			string	是	默认 VPC 内的子网名称。	用于访问内网/外网的子网名称。	
externalSubnet				否	-		
internalIPs			string 数组	是	-	用于访问内网/外网的 IP 地址。支持 IPv4 栈。 指定的 IP 数量不得少于 *replicas*。 建议设置为 *<replicas> + 1*，避免 Pod 创	
externalIPs							
bfd	enabled		boolean	是	false	BFD 配置。	是否为 Egr 启用 BFD。
	minRX		integer/int32	是	1000		BFD 的 mir 单位毫秒。
	minTX						
	multiplier		integer/int32	是	3		BFD 乘数。
policies	snat		boolean	是	false	出口策略。	是否启用 SNAT/MAS
	ipBlocks		string 数组	是	-		适用的 IP 段 支持 IPv4 和
	subnets		string 数组	是	-		适用的 VPC 支持 IPv4、 子网。
selectors	namespaceSelector	matchLabels	object	是	-	通过命名空间选择器和 Pod 选择器配置出口策略。 匹配的 Pod 会应用 SNAT/MASQUERADE。	命名空间选择器。空标签选择器匹配所有命名空间。
		matchExpressions	object 数组	是	-		
	podSelector	matchLabels	object	是	-		Pod 选择器。空标签选择器匹配所有 Pod。
		matchExpressions	object 数组	是	-		

字段			类型	可选	默认值	描述	
nodeSelector	matchLabels		object	是	-	用于选择承载工作负载部署的节点的节点选择器。 工作负载（Deployment/Pod）将在选中节点上运行。	键值对映射
	matchExpressions		object 数组	是	-		标签选择器要求间为 A
	matchFields		object 数组	是	-		字段选择器要求间为 A
trafficPolicy			string	是	Cluster	仅在启用 **BFD** 时生效。 可选值：*Cluster*/*Local*。 设置为 *Local* 时，出口流量会优先重定向行的 VPC Egress Gateway 实例，若实例定向到其他实例。	

相关资源

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

网络可观测性

目录

关于 DeepFlow
- 什么是 DeepFlow
- 使用 eBPF 技术
- 软件架构

安装 DeepFlow
- 介绍
  - 内核要求
  - 部署拓扑
- 准备工作
  - 存储类
  - 软件包
- 安装
- 访问 Grafana Web UI

其他资源

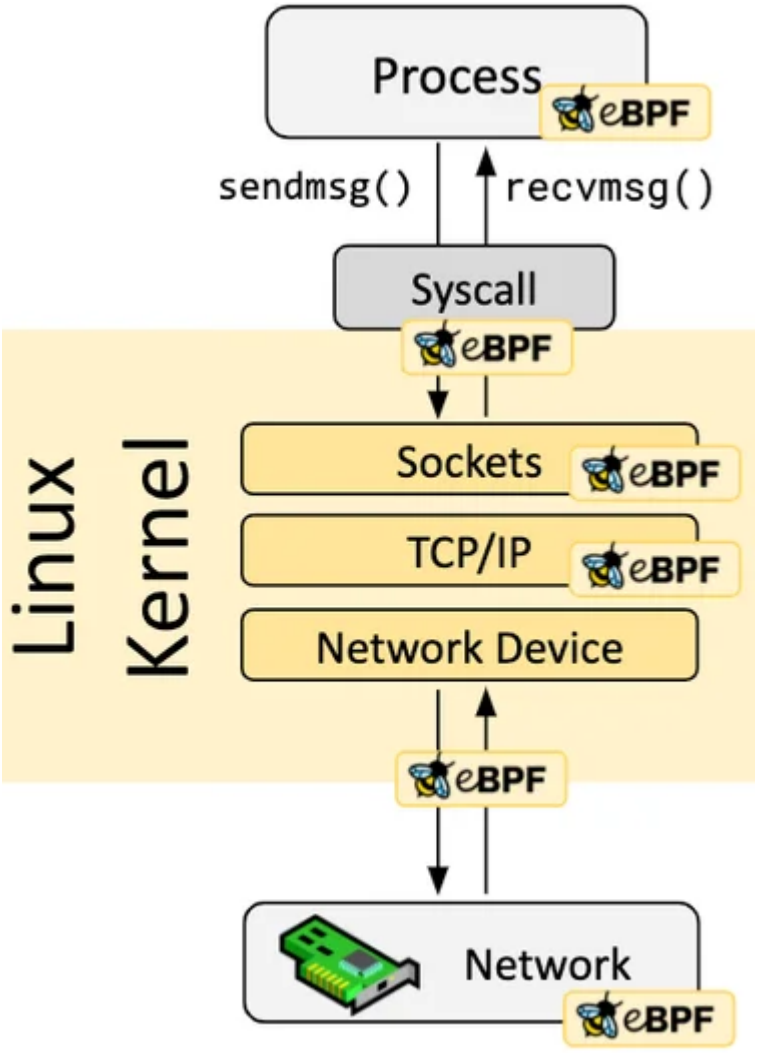
关于 **DeepFlow**

什么是 **DeepFlow**

DeepFlow 开源项目旨在为复杂的云原生和 AI 应用提供深度可观测性。
DeepFlow 通过 eBPF 实现了零代码数据采集，涵盖指标、分布式追踪、请求日志和函数性能分析，并进一步集成了 SmartEncoding，实现全栈关联和高效访问所有可观测数据。借助 DeepFlow，云原生和 AI 应用自动获得深度可观测性，免除了开发者持续在代码中埋点的繁重负担，为 DevOps/SRE 团队提供从代码到基础设施的全面监控和诊断能力。

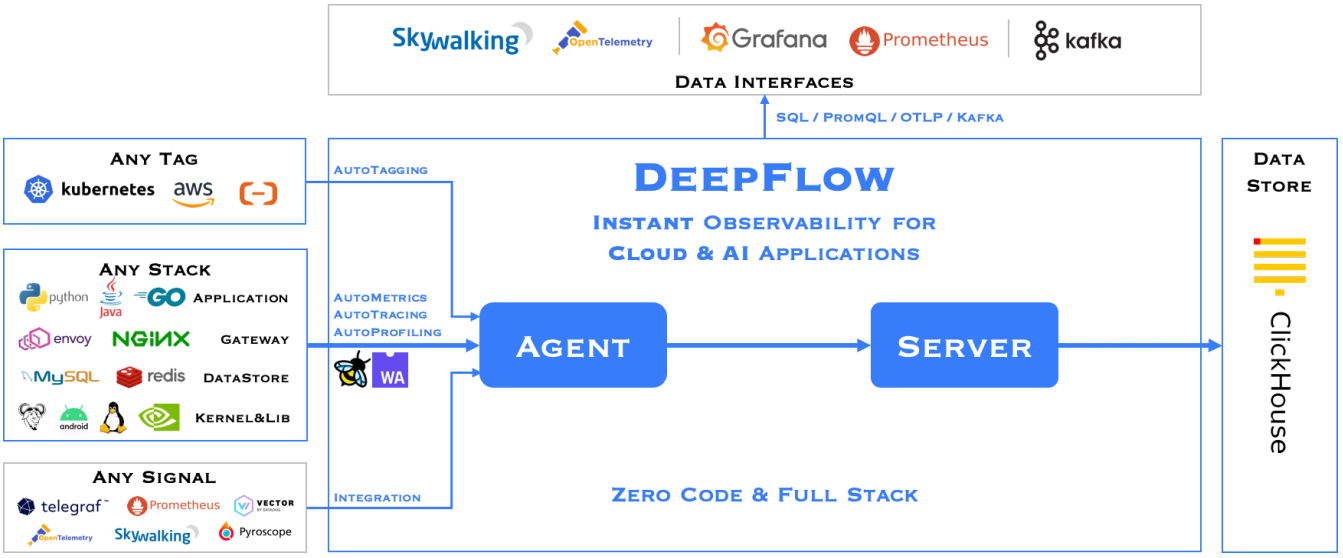
使用 **eBPF** 技术

假设您对 eBPF 有基本了解，它是一种通过在沙箱中运行程序来扩展内核功能的安全高效技术，相较于传统修改内核源码和编写内核模块的方式，是一项革命性创新。eBPF 程序是事件驱动的，当内核或用户程序经过 eBPF Hook 时，会执行加载在该 Hook 点的对应 eBPF 程序。Linux 内核预定义了一系列常用的 Hook 点，也可以通过 kprobe 和 uprobe 技术动态添加内核和应用程序中的自定义 Hook 点。
得益于即时编译（JIT）技术，eBPF 代码的执行效率可媲美原生内核代码和内核模块。
得益于验证机制，eBPF 代码运行安全，不会导致内核崩溃或进入死循环。



软件架构

DeepFlow 由两个组件组成，Agent 和 Server。Agent 运行在每个 K8s 节点、传统主机和云主机上，负责采集主机上所有应用进程的 AutoMetrics 和 AutoTracing 数据。Server 运行在 K8s 集群中，提供 Agent 管理、标签注入、数据摄取和查询服务。



安装 DeepFlow

介绍

内核要求

DeepFlow 中的 eBPF 功能（AutoTracing、AutoProfiling）对内核版本有以下要求：

架构	发行版	内核版本	kprobe	Golang uprobe	OpenSSL uprobe	perf
X86	CentOS 7.9	3.10.0 1	Y	Y 2	Y 2	Y
	RedHat 7.6	3.10.0 1	Y	Y 2	Y 2	Y
	*	4.9-4.13				Y
		4.14 3	Y	Y 2		Y

架构	发行版	内核版本	kprobe	Golang uprobe	OpenSSL uprobe	perf
		4.15	Y	Y [2]		Y
		4.16	Y	Y		Y
		4.17+	Y	Y	Y	Y
ARM	CentOS 8	4.18	Y	Y	Y	Y
	EulerOS	5.10+	Y	Y	Y	Y
	KylinOS V10 SP2	4.19.90-25.24+	Y	Y	Y	Y
	KylinOS V10 SP3	4.19.90-52.24+	Y	Y	Y	Y
	其他发行版	5.8+	Y	Y	Y	Y

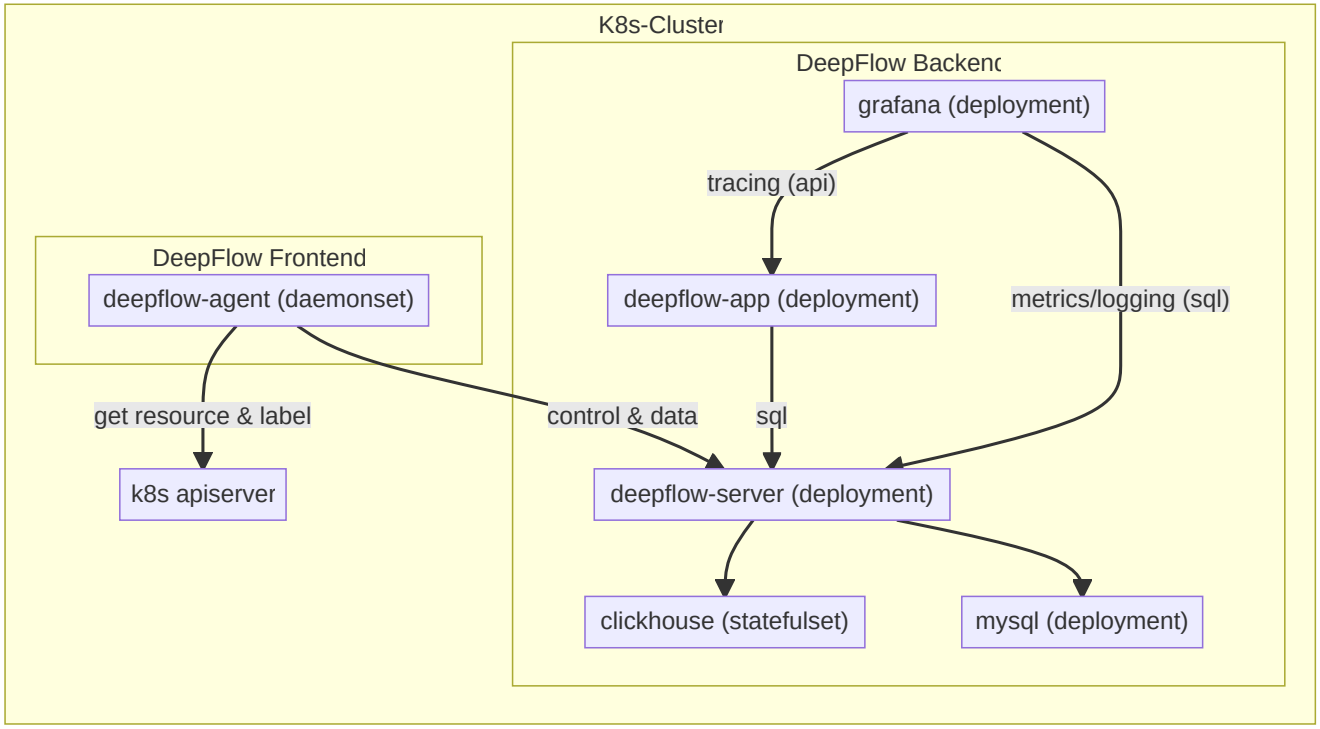
内核版本附加说明：

1. CentOS 7.9 和 RedHat 7.6 在 3.10 内核中回移植了一些 eBPF 功能 (opens new window)。在这两个发行版中，DeepFlow 支持的具体内核版本如下（依赖的 Hook 点）：
   - 3.10.0-957.el7.x86_64
   - 3.10.0-1062.el7.x86_64
   - 3.10.0-1127.el7.x86_64
   - 3.10.0-1160.el7.x86_64
2. 容器内的 Golang/OpenSSL 进程不支持。
3. 在内核版本 4.14 中，tracepoint 不能被多个 eBPF 程序附加（例如，两个或更多 deepflow-agent 不能同时运行），其他版本不存在此问题。

> 注意
>
> RedHat 声明：

> Red Hat Enterprise Linux 7.6 中的 eBPF 作为技术预览提供，
> 因此不具备完整支持，不适合生产环境部署。
> 其主要目的是获得更广泛的曝光，并可能在未来实现完整支持。
> Red Hat Enterprise Linux 7.6 中的 eBPF 仅启用追踪功能，
> 允许将 eBPF 程序附加到探针、tracepoint 和 perf 事件。

部署拓扑



准备工作

存储类

DeepFlow 中的 MySQL 和 ClickHouse 需要由 Storage Class 提供的持久卷存储。

有关存储配置的更多信息，请参见存储文档。

软件包

下载 **DeepFlow** 软件包

访问 Custom Portal 下载 DeepFlow 软件包。

如果您无法访问 Custom Portal，请联系技术支持。

上传软件包到平台

使用 violet 工具将软件包发布到平台。

有关该工具的详细使用说明，请参见CLI。

安装

1. 进入 管理员 > **Marketplace** > 集群插件。
2. 在插件列表中搜索“**Alauda Container Platform Observability with DeepFlow**”。
3. 点击 安装，打开安装配置页面。
4. 根据需要填写配置参数。各参数详细说明请参见下表。
5. 等待插件状态变为 已安装。

表格：配置参数

参数	是否可选	说明
Replicas	否	ClickHouse 服务器和 DeepFlow 服务器的副本数。建议设置为大于等于 3 的奇数，以保证高可用性。
Storage Class	是	用于为 MySQL 和 ClickHouse 创建持久卷的 Storage Class。未设置时使用默认 Storage Class。
MySQL Storage Size	否	MySQL 持久卷大小。
ClickHouse Storage Size	否	ClickHouse 存储卷大小。
ClickHouse Data Storage Size	否	ClickHouse 数据存储大小。
Username	否	Grafana Web 控制台用户名。
Password	否	Grafana Web 控制台密码。强烈建议首次登录后修改该密码。

参数	是否可选	说明
Confirm Password	否	确认 Grafana Web 控制台密码。
Ingress Class Name	是	用于创建 Grafana Web 控制台 Ingress 的 Ingress Class 名称。未设置时使用默认 Ingress Class。
Ingress Path	否	Grafana Web 控制台的 Ingress 服务路径。
Ingress TLS Secret Name	是	Grafana Web 控制台 Ingress 使用的 TLS Secret 名称。
Ingress Hosts	是	Grafana Web 控制台 Ingress 使用的主机列表。
Agent Group Configuration	否	默认 DeepFlow Agent 组的配置。

访问 **Grafana Web UI**

您可以通过 Ingress 配置中指定的主机和服务路径访问 Grafana Web UI，使用用户名和密码登录。

> 注意
>
> 强烈建议首次登录后修改密码。

其他资源

- DeepFlow - 云与 AI 应用的即时可观测性↗
- DeepFlow Agent 配置↗
- eBPF - 介绍、教程与社区资源↗

配置 ALB 规则

目录

介绍
- 什么是规则？
- 前提条件
- 规则快速演示
- 使用 dslx 和优先级匹配请求
  - dslx
  - 优先级

动作

后端
- 后端协议
- 服务组和会话亲和策略

创建规则
- 使用 Web 控制台
- 使用 CLI

Https
- Ingress 中的证书注解
- 规则中的证书
- TLS 模式
  - Edge 模式
  - 重新加密模式

Ingress
- ingress 同步

ssl 策略

介绍

什么是规则？

规则是一个自定义资源（CR），定义了 ALB 如何匹配和处理传入请求。

由 ALB 处理的 Ingress 可以自动转换为规则。

前提条件

安装 alb 和 ft

规则快速演示

下面是一个演示规则，帮助你快速了解如何使用规则。

> **NOTE**
>
> 规则必须通过标签关联到 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，指明该规则所属的 `Frontend`。
2. 必填，指明该规则所属的 ALB。
3. backendProtocol
4. certificate_name
5. dslx
6. 数值越小，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特定语言。 例如，下面的规则匹配满足所有以下条件的请求：

- url 以 /app-a 或 /app-b 开头
- method 是 post
- url 参数 group 的值是 vip
- host 是 *.app.com
- header 中 location 的值是 east-1 或 east-2
- 有名为 uid 的 cookie
- 源 IP 在 1.1.1.1-1.1.1.100 范围内

```
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，数值越小优先级越高。要在 ingress 中配置规则优先级，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，直接在 `.spec.priority` 中设置整数值即可。

动作

请求匹配规则后，可以对请求应用以下动作：

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

后端

后端协议

默认情况下，后端协议设置为 HTTP。如果需要使用 TLS 重新加密，可以配置为 HTTPS。

服务组和会话亲和策略

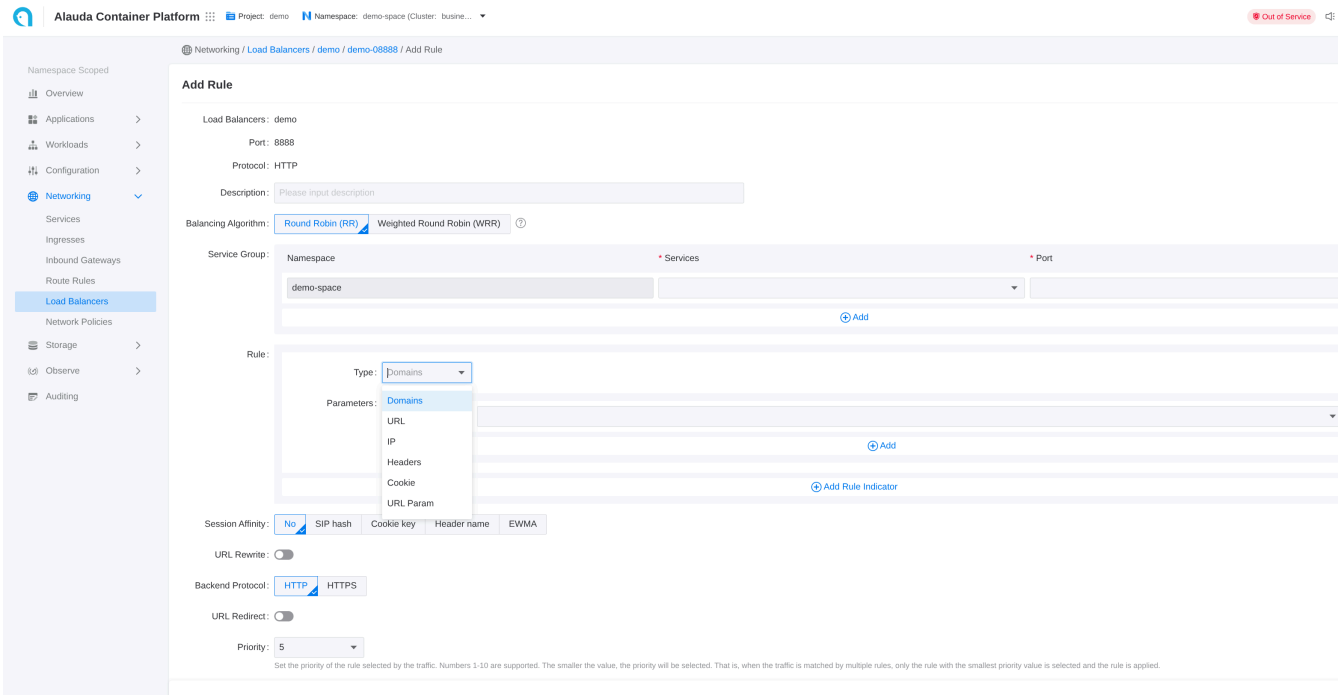
规则中可以配置一个或多个服务。

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

详情请参考负载均衡算法。

创建规则

使用 **Web** 控制台



1. 进入 **Container Platform**。
2. 点击左侧导航栏的 **Network** > **Load Balancing**。
3. 点击负载均衡器名称。
4. 点击监听端口名称。
5. 点击 **Add Rule**。
6. 参考以下说明配置相关参数。
7. 点击 **Add**。

Web UI 上的每个输入项对应 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）是 HTTPS 或 GRPCS，规则也可以配置为使用 HTTPS。 你可以在规则或 ingress 中指定证书，以匹配该端口的证书。

支持终止（termination），如果后端协议是 HTTPS，也支持重新加密（re-encryption）。但不能为与后端服务通信指定证书。

Ingress 中的证书注解

证书可以通过注解跨命名空间引用。

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

规则中的证书

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

TLS 模式

Edge 模式

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

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

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

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

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

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

> **NOTE**
>
> ALB 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 命名空间。

ssl 策略

对于未配置证书的 ingress，ALB 提供使用默认证书的策略。

你可以通过以下配置设置 ALB 自定义资源：

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

可用的 SSL 策略：

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

集群互联（**Alpha**）

支持配置网络模式相同为 Kube-OVN 的集群之间的集群互联，使集群内的 Pod 能够相互访问。集群互联控制器是 Kube-OVN 提供的扩展组件，负责收集不同集群间的网络信息，并通过下发路由实现多个集群网络的互联。

目录

前提条件

多节点 Kube-OVN 互联控制器构建

- Deploy 部署
- Docker 和 Containerd 部署

在 global 集群部署集群互联控制器

加入集群互联

相关操作

- 更新互联集群的网关节点信息
- 退出集群互联
- 清理互联集群残留
- 卸载互联集群
- 配置集群网关高可用

前提条件

- 不同集群的子网 CIDR 不能相互重叠。

- 需要有一组机器，能够被每个集群的 kube-ovn-controller 通过 IP 访问，用于部署跨集群互联的控制器。
- 每个集群需要有一组机器，能够被 kube-ovn-controller 通过 IP 访问，用作后续的网关节点。
- 该功能仅支持默认 VPC，用户自定义 VPC 无法使用互联功能。

多节点 Kube-OVN 互联控制器构建

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

Deploy 部署

注意：该部署方式支持平台 v3.16.0 及以上版本。

操作步骤

1. 在集群主节点执行以下命令获取 install-ic-server.sh 安装脚本。

```
wget https://github.com/kubeovn/kube-ovn/blob/release-1.12/dist/images/install-ic-server.sh
```

2. 打开当前目录下的脚本文件，按如下修改参数。

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

   修改后的参数配置如下：

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

3. 保存脚本文件后，执行以下命令运行脚本。

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

Docker 和 Containerd 部署

1. 选择任意集群中的 三个及以上节点 部署互联控制器。此处以准备三台节点为例。

2. 任选一台节点作为 Leader，根据不同部署方式执行以下命令。

   注意：配置前请检查 `/etc` 目录下是否存在 ovn 目录，如无则执行 `mkdir /etc/ovn` 创建。

   - **Docker** 部署命令

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

     - Leader 节点命令：

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

     - 其他两台节点命令：

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

- **Containerd** 部署命令

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

  - Leader 节点命令：

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

  - 其他两台节点命令：

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

在 **global** 集群部署集群互联控制器

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

注意：为保证正确运行，global 上名为 ovn-ic 的 ConfigMap 不允许被修改。如需更改参数，请先删除 ConfigMap 并正确重新配置后再应用。

```
cat << EOF |kubectl apply -f -
apiVersion: v1
kind: ConfigMap
metadata:
  name: ovn-ic
  namespace: cpaas-system
data:
  ic-db-host: "192.168.39.22,192.168.39.24,192.168.39.37"   # 集群互联控制器所在节点地
址，此处为控制器部署的三台节点的本地 IP
  ic-nb-port: "6645"            # 集群互联控制器 nb 端口，默认 6645
  ic-sb-port: "6646"            # 集群互联控制器 sb 端口，默认 6646
EOF
```

加入集群互联

将网络模式为 Kube-OVN 的集群加入集群互联。

前提条件

集群中 已创建的子网、**ovn-default** 及 加入的子网 不得与集群互联组中任何集群网段冲突。

操作步骤

1. 在左侧导航栏点击 集群 > 集群的集群。
2. 点击要加入集群互联的 *集群* 名称。
3. 在右上角点击 操作 > 集群互联。
4. 点击 加入集群互联。
5. 选择该集群的网关节点。
6. 点击 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. 在左侧导航栏点击 集群 > 集群的集群。
2. 点击需更新网关节点信息的 *集群名称*。
3. 在右上角点击 操作 > 集群互联。
4. 点击需更新网关节点信息的集群对应的 更新网关节点。
5. 重新选择该集群的网关节点。

6. 点击 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，退出后集群 Pod 与外部集群 Pod 断开连接。

操作步骤

1. 在左侧导航栏点击 集群 > 集群的集群。
2. 点击要退出的 *集群* 名称。
3. 在右上角点击 操作 > 集群互联。
4. 点击要退出的集群对应的 退出集群互联。
5. 正确输入集群名称。
6. 点击 退出。

清理互联集群残留

当集群未退出互联集群即被删除时，控制器上可能残留部分数据。若尝试使用这些节点重新创建集群并加入互联集群，可能会失败。可在控制器（kube-ovn-controller）的 `/var/log/ovn/ovn-ic.log` 日志中查看详细错误信息，部分错误信息示例如下：

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

操作步骤

1. 对要加入的集群执行退出互联集群操作。
2. 在容器或 Pod 中执行清理脚本。

   可直接在 ovn-ic-db 容器或 ovn-ic-controller Pod 中执行清理脚本，任选其一：

   方法一：在 **ovn-ic-db** 容器中执行

   - 进入 ovn-ic-db 容器，执行以下命令进行清理操作。

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

然后执行以下清理命令之一：

- 使用原集群名称执行清理操作，替换 *<cluster-name>* 为 原集群名称：

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

- 使用原集群中任意节点名称执行清理操作，替换 *<node-name>* 为 原集群中任意节点名称：

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

方法二：在 **ovn-ic-controller Pod** 中执行

- 进入 ovn-ic-controller Pod，执行以下命令进行清理操作。

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

然后执行以下清理命令之一：

- 使用原集群名称执行清理操作，替换 *<cluster-name>* 为 原集群名称：

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

- 使用原集群中任意节点名称执行清理操作，替换 *<node-name>* 为 原集群中任意节点名称：

  ```
  ./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，有两种具体退出方式，根据需求选择。

   - 删除业务集群中名为 ovn-ic-config 的 ConfigMap，执行以下命令。

     ```
     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. 登录控制器部署节点，删除控制器。

   - Docker 命令：

     ```
     docker stop ovn-ic-db
     docker 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，若需配置集群网关高可用，可执行以下步骤：

1. 登录需改造为高可用网关的集群，执行以下命令将 `enable-ic` 字段修改为 `false`。

   注意：将 `enable-ic` 字段改为 `false` 会中断集群互联，直到再次设置为 `true`。

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

2. 修改网关节点配置，更新 `gw-nodes` 字段，网关节点之间用英文逗号分隔；同时将 `enable-ic` 字段改为 `true`。

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

   # 配置示例
   apiVersion: v1
   data:
     auto-route: "true"
     az-name: docker
     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，执行 `ovn-nbctl lrp-get-gateway-chassis {当前集群名称}-ts` 命令，验证配置是否生效。

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

# 返回示例。此处 100 和 99 为优先级值，值越大对应的网关节点优先级越高。
docker-ts-71292a21-131d-492a-9f0c-0611af458950 100
docker-ts-1de7ee15-f372-4ab9-8c85-e54d61ea18f1 99
```

Endpoint Health Checker

目录

Overview
Key Features
Installation
Install via Marketplace
How It Works
Health Check Mechanism
Core Functionality
Health Check Process
Activation Methods
Uninstallation

Overview

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

Key Features

- 自动健康监控：持续监控服务端点的健康状态
- 负载均衡器集成：自动将不健康的端点从负载均衡轮换中移除
- 服务可用性：确保流量仅导向健康且可用的端点

Installation

Install via Marketplace

1. 进入 管理员 > **Marketplace** > 集群插件。
2. 在插件列表中搜索“**Alauda Container Platform Endpoint Health Checker**”。
3. 点击 安装，打开安装配置页面。
4. 等待插件状态变为“**Ready**”。

How It Works

Health Check Mechanism

Endpoint Health Checker 是一个专用的健康监控组件，确保只有健康的端点接收流量。它通过监控服务端点并自动管理其可用状态来实现。

Core Functionality

Endpoint Health Checker 的工作原理：

1. 服务发现：识别配置了健康监控的服务和 Pod
2. **Pod** 健康监控：监控支撑服务端点的 Pod 的就绪和存活探针状态
3. 主动健康检查：使用可配置的标准执行主动健康评估：
   - **TCP** 连接检查：建立 TCP 连接以验证端口可访问性
   - **HTTP/HTTPS** 响应验证：发送 HTTP 请求并验证响应码和内容

4. 端点管理：自动将不健康的端点从服务端点列表中移除，防止流量路由到故障实例

Health Check Process

健康检查流程包括：

- 探针集成：利用 Kubernetes 的就绪和存活探针结果作为初步健康指标
- 网络连通性：向目标端点端口发送 TCP 或 HTTP 数据包以验证可访问性
- 响应验证：评估响应状态、时长和内容以判断端点健康状况
- 自动故障转移：将无响应或失败的端点从负载均衡轮换中移除

Activation Methods

健康检查可通过两种方式激活：

1. **Pod** 级注解（推荐）：

   在 Deployment 的 Pod 模板中添加注解：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-demo
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx-demo
  template:
    metadata:
      labels:
        app: nginx-demo
      annotations:
        endpoint-health-checker.io/enabled: "true"
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          ports:
            - containerPort: 80
          livenessProbe:
            tcpSocket:
              port: 80
            initialDelaySeconds: 15
            periodSeconds: 10
          readinessProbe:
            tcpSocket:
              port: 80
            initialDelaySeconds: 5
            periodSeconds: 5
```

2. **Pod 级 readinessGates**（旧版）：

   为旧版本配置 Pod 的 readinessGates：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-demo-legacy
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx-demo-legacy
  template:
    metadata:
      labels:
        app: nginx-demo-legacy
    spec:
      readinessGates:
        - conditionType: "endpointHealthCheckSuccess"
      containers:
        - name: nginx
          image: nginx:alpine
          ports:
            - containerPort: 80
          livenessProbe:
            tcpSocket:
              port: 80
            initialDelaySeconds: 15
            periodSeconds: 10
          readinessProbe:
            tcpSocket:
              port: 80
            initialDelaySeconds: 5
            periodSeconds: 5
```

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

Uninstallation

卸载 Endpoint Health Checker：

1. 进入 管理员 > **Marketplace** > 集群插件。

2. 找到已安装的“**Endpoint Health Checker**”插件。

3. 点击选项菜单，选择 卸载。

4. 按提示确认卸载。

NodeLocal DNSCache

目录

Overview
Key Features
Important Notes
Installation
Install via Marketplace
How It Works
Architecture
Configuration
Network Policy Configuration

Overview

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

Key Features

- 本地 **DNS** 缓存：在每个节点本地缓存 DNS 响应，减少查询延迟

- 性能提升：显著缩短应用的 DNS 查询时间

Important Notes

WARNING

部署注意事项：

1. **Kube-OVN Underlay** 模式：该插件不支持在 Kube-OVN Underlay 模式下部署，若部署可能导致 DNS 查询失败。
2. **Kubelet** 重启：部署该插件会导致 kubelet 重启。
3. **Pod** 重启要求：插件成功部署后不会影响正在运行的 Pod，仅对新创建的 Pod 生效。当 CNI 为 Kube-OVN 时，需要手动在 kube-ovn-controller 中添加参数 "--node-local-dns-ip=(本地 DNS 缓存服务器的 IP 地址)"。
4. **NetworkPolicy** 配置：如果集群配置了 NetworkPolicy，需要在 networkPolicy 中额外允许 node CIDR 和 nodeLocalDNSIP 的双向通信，以确保正常通信。

Installation

Install via Marketplace

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

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

5. 查看部署注意事项，确保环境满足要求。

6. 点击 安装 完成安装。

7. 等待插件状态变为 "**Ready**"。

How It Works

Architecture

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

Configuration

Network Policy Configuration

> 重要：如果集群启用了 NetworkPolicy，必须配置合适的规则以允许 DNS 流量访问 NodeLocal DNSCache。否则 Pod 可能无法解析 DNS 查询。

使用 NetworkPolicy 时，确保允许以下 DNS 流量：

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

Menu

如何操作

准备 **Kube-OVN Underlay** 物理网络

准备 **Kube-OVN Underlay** 物理网络

- 使用说明
- 术语解释
- 环境要求
- 配置示例

软数据中心 **LB** 方案（**Alpha**）

软数据中心 **LB** 方案（**Alpha**）

- 前提条件
- 操作步骤
- 验证

Underlay 和 Overlay 子网的自动互联

Underlay 和 Overlay 子网的自动互联

- 操作步骤

通过集群插件安装 Ingress-Nginx

通过集群插件安装 Ingress-Nginx

- 概述
- 安装
- 配置管理
- 性能调优
- 重要说明

通过 Ingress Nginx Operator 安装 Ingress-Nginx

通过 **Ingress Nginx Operator** 安装 **Ingress-Nginx**

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

Ingress-Nginx 的任务

Ingress-Nginx 的任务

- 前提条件
- 最大连接数
- 超时
- 会话保持
- 头部修改
- URL 重写
- HSTS（HTTP 严格传输安全）
- 速率限制
- WAF
- 转发头控制
- HTTPS

ALB

Auth

- 基本概念
- 快速开始
- 相关 Ingress 注解
- forward-auth
- basic-auth
- CR
- ALB 特殊 Ingress 注解
- Ingress-Nginx Auth 相关其他特性
- 注意：与 Ingress-Nginx 不兼容的部分
- 故障排查

部署 ALB 的高可用 VIP

- 方式一：使用 LoadBalancer 类型的 Service 提供 VIP
- 方式二：使用外部 ALB 设备提供 VIP

Header Modification

- 基本概念
- 示例

HTTP 重定向

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

L4/L7 超时

- 基本概念
- CRD
- 超时的含义
- Ingress 注解
- 端口级超时

ModSecurity

- 术语
- 操作步骤
- 相关说明
- 配置示例

TCP/HTTP Keepalive

- 基本概念
- CRD

使用 OAuth Proxy 配合 ALB

- Overview
- Procedure
- Result

通过 ALB 配置 GatewayApi Gateway

- 术语
- 前提条件
- Gateway 和 Alb2 自定义资源（CR）示例
- 通过 Web 控制台创建 Gateway
- 通过 CLI 创建 Gateway
- 查看平台创建的资源
- 更新 Gateway
- 通过 Web 控制台更新 Gateway
- 添加监听器
- 通过 Web 控制台添加监听器
- 通过 CLI 添加监听器
- 创建路由规则
- HTTPRoute 自定义资源（CR）示例
- 通过 Web 控制台创建路由规则
- 通过 CLI 创建路由规则

在 ALB 中绑定网卡

- 对于集群内嵌 ALB
- 对于用户自定义 ALB

ALB 性能选择决策

- 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

部署 ALB

- ALB
- 监听端口（Frontend）
- 日志与监控

通过 ALB 将 IPv6 流量转发到集群内的 IPv4 地址

- 配置方法
- 结果验证

OTel

- 术语
- 前提条件
- 操作步骤
- 相关操作
- 附加说明
- 配置示例

ALB 监控

- 术语
- 操作步骤
- 监控指标

CORS

- 基本概念
- CRD

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

URL 重写

- 基本概念
- 配置

Calico 网络支持 WireGuard 加密

Calico 网络支持 WireGuard 加密

- 安装状态
- 术语
- 注意事项
- 先决条件
- 操作步骤
- 结果验证

Kube-OVN Overlay 网络支持 IPsec 加密

Kube-OVN Overlay 网络支持 IPsec 加密

- 术语
- 注意事项
- 先决条件
- 操作步骤

准备 **Kube-OVN Underlay** 物理网络

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

目录

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

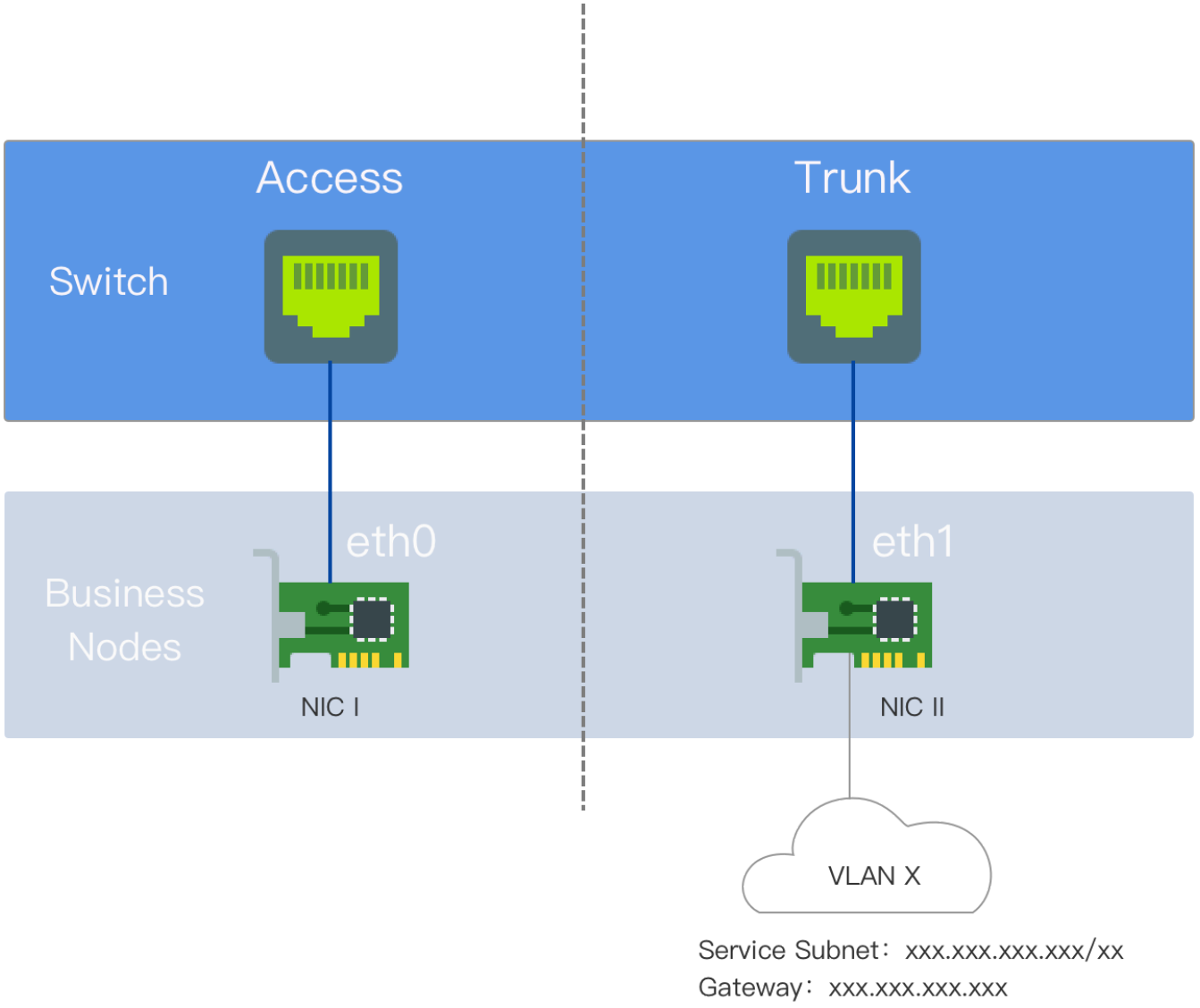
使用说明

Kube-OVN Underlay 需要多网卡 (NIC) 的部署，Underlay 子网必须专用一个 NIC。该 NIC 上不得有其他类型流量，如 SSH；其他流量应使用其他 NIC。

在使用之前，请确保节点服务器至少具备 双 **NIC** 环境，并建议 NIC 速度 至少为 **10 Gbps** 或更高（例如，10 Gbps、25 Gbps、40 Gbps）。

- NIC 一：具有默认路由的 NIC，配置有 IP 地址，与外部交换机接口互连，设置为访问模式。

- NIC 二：没有默认路由且未配置 IP 地址的 NIC，与外部交换机接口互连，设置为干道模式。Underlay 子网专用 NIC 二。



术语解释

VLAN（虚拟局域网）是一种技术，它逻辑上将局域网划分为多个段（或较小的 LAN），以便利虚拟工作组的数据交换。

VLAN 技术的出现使管理员能够根据实际应用需求，逻辑上将同一物理局域网中的不同用户划分为不同的广播域。每个 VLAN 由一组拥有相似需求的计算机工作站组成，并具有与物理形成的 LAN 相同的特性。由于 VLAN 是逻辑划分而非物理划分，因此同一 VLAN 内的工作站并不局限于相同的物理区域；它们可以存在于不同的物理 LAN 段中。

VLAN 的主要优点包括：

- 端口分段。即使在同一交换机上，不同 VLAN 的端口之间也无法相互通信。物理交换机能够作为多个逻辑交换机工作。这通常用于控制不同部门和站点之间的相互访问。
- 网络安全。不同的 VLAN 之间无法直接通信，消除了广播信息的安全隐患。VLAN 内的广播和单播流量不会转发到其他 VLAN，助于控制流量、减少设备投资、简化网络管理并提高网络安全性。
- 灵活管理。当更改用户的网络隶属关系时，无需更换端口或电缆；只需进行软件配置更改。

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 直接发送数据包到外部。L2/L3 转发能力依赖于底层网络设备。相应的网关、VLAN 和安全策略需要在底层网络设备上提前配置。

- 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，Pods 将首先向网关发送数据包，网关必须具备将数据包转发回本地子网的能力。
  - 当交换机或桥接启用了 Hairpin 功能时，必须禁用 **Hairpin**。如果使用 VMware 虚拟机环境，将 VMware 主机上的 **Net.ReversePathFwdCheckPromisc** 设置为 **1**，Hairpin 不需要禁用。
  - 桥接 NIC 不能 是 **Linux** 桥。
  - NIC 绑定模式支持模式 0（balance-rr）、模式 1（active-backup）、模式 4（802.3ad）、模式 6（balance-alb），推荐使用 0 或 1。其他绑定模式未经过测试，请谨慎使用。
- **IaaS**（虚拟化）层配置要求
  - 对于 OpenStack 虚拟机环境，相应网络端口的 **PortSecurity** 需要禁用。

- 对于 VMware 的 vSwitch 网络，**MAC** 地址更改、伪造发送和 混杂模式操作必须全部设置为 接受。
- 对于 AWS、GCE 和阿里云等公共云，由于缺乏用户定义的 MAC 地址能力，无法支持 Underlay 模式网络。

配置示例

本示例中的节点是双 NIC 物理机器。NIC 一是具有默认路由的 NIC；NIC 二是没有默认路由且未配置 IP 地址的 NIC，专用于 Underlay 子网。NIC 二与外部交换机互连。

- 在交换机端，连接到 NIC 二 的接口应配置为干道模式，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，还可以同时配置 IPv6 网关地址。
- 如果网关位于防火墙后，则必须允许节点到 cluster-cidr 网络的访问。
- 服务器 NIC 不需要配置。

交换机配置

配置 VLAN 接口：

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

配置连接到 NIC 二 的接口：

```
#
interface Ten-GigabitEthernet1/0/19
  port link mode bridge
  port link-type trunk  // 配置接口为干道模式
  undo port trunk permit vlan 1
  port trunk permit vlan 74  // 允许相应 VLAN 通过
#
```

检查网络连通性

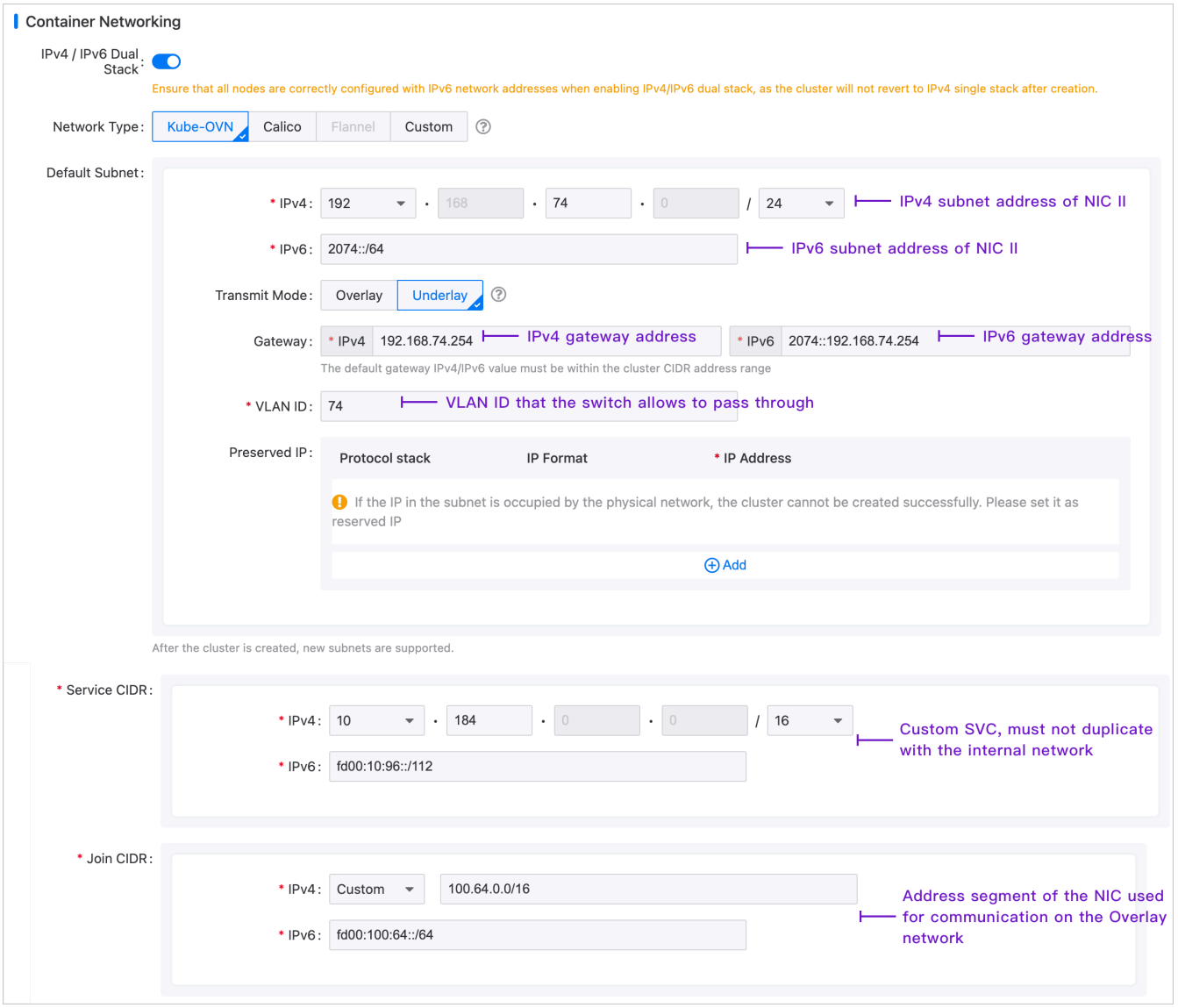
测试 NIC 二 是否能够与网关地址通信：

```
ip link add ens224.74 link ens224 type vlan id 74  // NIC 名称为 ens224, VLAN ID 为 74
ip link set ens224.74 up
ip addr add 192.168.74.200/24 dev ens224.74  // 在 Underlay 子网内选择一个测试地址, 这里为 192.168.74.200/24
ping 192.168.74.254  // 如果能够 ping 通网关, 确认物理环境符合部署要求
ip addr del 192.168.74.200/24 dev ens224.74  // 测试后删除测试地址
ip link del ens224.74  // 测试后删除子接口
```

平台配置

在左侧导航栏中，点击 集群管理 > 集群，然后点击 创建集群。有关具体配置步骤，请参阅 创建集群 文档，容器网络配置见下图所示。

注意：加入子网在 Underlay 环境中没有实际意义，主要用于后续创建 Overlay 子网，提供节点和容器组之间通信所需的 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

软数据中心 LB 方案（Alpha）

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

目录

前提条件

操作步骤

验证

前提条件

1. 准备两个或以上主机节点作为 LB，建议 LB 节点安装 Ubuntu 22.04 操作系统，以减少 LB 转发流量到异常后端节点的时间。
2. 在所有外部 LB 主机节点预先安装以下软件（本章以两个外部 LB 主机节点为例）：
   - `ipvsadm`
   - `Docker (20.10.7)`
3. 确保每个主机的 Docker 服务开机启动，执行命令：`sudo systemctl enable docker.service`。
4. 确保每个主机节点的时钟同步。

5. 准备 Keepalived 镜像，用于启动外部 LB 服务；平台已包含该镜像。镜像地址格式如下：`<image repository address>/tkestack/keepalived:<version suffix>`。不同版本的版本后缀可能略有差异。可通过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。
   - 在安装包解压目录执行 `cat ./installer/res/artifacts.json |grep keepalived -C 2|grep tag|awk '{print $2}'|awk -F '"' '{print $2}'` 获取 版本后缀。

操作步骤

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

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

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

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

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

3. 执行 `reboot` 命令重启。

4. 创建 Keepalived 配置文件夹。

```
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，并保存到 `/etc/keepalived/` 文件夹，文件名为 `alive.yaml`。

```
instances:
  - vip: # 可配置多个 VIP
      vip: 192.168.128.118 # VIP 必须不同
      id: 20 # 每个 VIP 的 ID 必须唯一，可选
      interface: "eth0"
      check_interval: 1 # 可选，默认 1：执行检查脚本的间隔
      check_timeout: 3  # 可选，默认 3：检查脚本超时时间
      name: "vip-1" # 此实例的标识符，只能包含字母数字和连字符，且不能以连字符开头
      peer: [ "192.168.128.116", "192.168.128.75" ] # Keepalived 节点 IP，实际生成的
keepalived.conf 会移除接口上的所有 IP https://github.com/osixia/docker-
keepalived/issues/33
      kube_lock:
        kubecfgs: # kube-lock 使用的 kube-config 列表，会依次尝试这些 kubecfg 进行
Keepalived 的 leader 选举
          - "/live/cfg/kubecfg/kubecfg01.conf"
          - "/live/cfg/kubecfg/kubecfg02.conf"
          - "/live/cfg/kubecfg/kubecfg03.conf"
    ipvs: # IPVS 选项配置
      ips: [ "192.168.143.192", "192.168.138.100","192.168.129.100" ] # IPVS 后端，将
k8s master 节点 IP 改为 ALB 节点的节点 IP
      ports: # 配置 VIP 上每个端口的健康检查逻辑
        - port: 80 # 虚拟服务器上的端口必须与真实服务器端口一致
          virtual_server_config: |
            delay_loop 10  # 对真实服务器执行健康检查的间隔
            lb_algo rr
            lb_kind NAT
            protocol TCP
          raw_check: |
            TCP_CHECK {
                connect_timeout 10
                connect_port 1936
            }
  - vip:
      vip: 2004::192:168:128:118
      id: 102
      interface: "eth0"
      peer: [ "2004::192:168:128:75","2004::192:168:128:116" ]
      kube_lock:
        kubecfgs: # kube-lock 使用的 kube-config 列表，会依次尝试这些 kubecfg 进行
Keepalived 的 leader 选举
          - "/live/cfg/kubecfg/kubecfg01.conf"
          - "/live/cfg/kubecfg/kubecfg02.conf"
          - "/live/cfg/kubecfg/kubecfg03.conf"
```

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

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

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

7. 将 Kubernetes 集群中三个 Master 节点的 `/etc/kubernetes/admin.conf` 文件复制到外部 LB 节点的 `/etc/keepalived/kubecfg` 文件夹，命名带索引，如 `kubecfg01.conf`，并修改这三个文件中的 `apiserver` 节点地址为 Kubernetes 集群的实际节点地址。

   注意：平台证书更新后，需要重新执行此步骤，覆盖原文件。

8. 检查证书有效性。

   1. 将业务集群 Master 节点的 `/usr/bin/kubectl` 复制到 LB 节点。

   2. 执行 `chmod +x /usr/bin/kubectl` 赋予执行权限。

   3. 执行以下命令确认证书有效。

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

      若返回如下结果，则证书有效。

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

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

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

9. 将 Keepalived 镜像上传到外部 LB 节点，并使用 Docker 运行 Keepalived。

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

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

验证

1. 执行命令 `ipvsadm -ln` 查看 IPVS 规则，可见适用于业务集群 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 InActConn
TCP  192.168.128.118:80 rr
  -> 192.168.129.100:80           Masq    1      0          0
  -> 192.168.138.100:80           Masq    1      0          0
  -> 192.168.143.192:80           Masq    1      0          0
TCP  [2004::192:168:128:118]:80 rr
  -> [2004::192:168:129:100]:80   Masq    1      0          0
  -> [2004::192:168:138:100]:80   Masq    1      0          0
  -> [2004::192:168:143:192]:80   Masq    1      0          0
```

2. 关闭 VIP 所在的 LB 节点，测试 IPv4 和 IPv6 的 VIP 是否能成功迁移到其他节点，通常在 20 秒内完成。

3. 在非 LB 节点使用 `curl` 命令测试与 VIP 的通信是否正常。

```
curl 192.168.128.118

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

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

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

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

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

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

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

Underlay 和 Overlay 子网的自动互联

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

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

目录

操作步骤

操作步骤

1. 进入 管理员。
2. 在左侧导航栏点击 集群管理 > 资源管理。
3. 输入 **Subnet** 过滤资源对象。
4. 在需要修改的 Underlay 子网旁点击⋮> 更新。
5. 修改 YAML 文件，在 `Spec` 中添加字段 `u2oInterconnection: true`。
6. 点击 更新。

注意：Underlay 子网中已有的计算组件需要重新创建，变更才能生效。

通过集群插件安装 Ingress-Nginx

WARNING

Ingress-Nginx 集群插件已被废弃。请改用 ingress-nginx-operator。

目录

概述

安装

配置管理

- 更新配置
- 常见配置场景
  - 通过 LoadBalancer 暴露
  - MetalLB 集成
  - 高级 Controller 部署设置
  - SSL 透传
  - IPv6 单栈支持

性能调优

- 资源分配建议
  - 小规模（< 300 QPS）
  - 中等规模（< 10,000 QPS）
  - 大规模（< 20,000 QPS）
  - 高性能（无限制）

重要说明

- 限制

版本兼容性

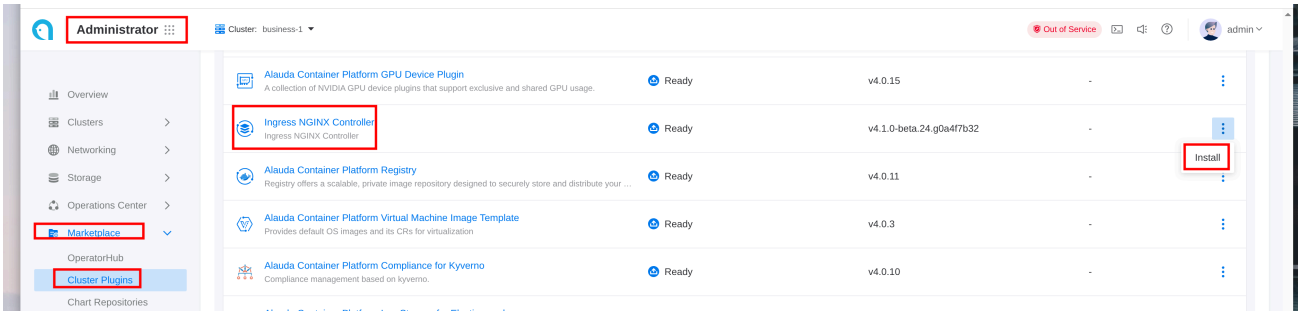
相关资源

概述

NGINX Ingress Controller 作为集群插件部署在 `cpaas-system` 命名空间中。本指南涵盖了在 Kubernetes 集群中安装、配置及管理 Ingress NGINX Controller 的最佳实践。

安装

1. 进入 `管理员 -> Marketplace -> 集群插件`
2. 找到 Ingress NGINX 插件并点击 `安装`



配置管理

更新配置

1. 配置 kubectl 使用 全局 集群上下文
2. 获取集群对应的 ModuleInfo 名称：

```
kubectl get minfo | grep ingress | grep $CLUSTER_NAME
```

3. 编辑 ModuleInfo 配置：

```
kubectl edit minfo $MINFO
```

其中 `spec.config` 部分对应 Ingress NGINX Helm Chart 的 values。

常见配置场景

通过 **LoadBalancer** 暴露

使用 LoadBalancer 类型的 Service 暴露 Ingress Controller：

```
spec:
  config:
    controller:
      service:
        type: LoadBalancer
```

MetalLB 集成

使用 MetalLB 时指定 LoadBalancer VIP：

```
spec:
  config:
    controller:
      service:
        annotations:
          metallb.universe.tf/loadBalancerIPs: "192.168.2.2"  # 期望的 VIP
          metallb.universe.tf/address-pool: "pool-name"        # MetalLB 地址池
```

高级 **Controller** 部署设置

配置网络模式、副本数、资源限制及节点选择：

```
spec:
  config:
    controller:
      hostNetwork: false
      replicaCount: 1
      nodeSelector:
        kubernetes.io/os: linux
      resources:
        limits:
          cpu: 200m
          memory: 256Mi
        requests:
          cpu: 200m
          memory: 256Mi
```

SSL 透传

启用 SSL 透传功能：

```
spec:
  config:
    controller:
      extraArgs:
        enable-ssl-passthrough: ""
```

IPv6 单栈支持

配置仅使用 IPv6：

```
spec:
  config:
    controller:
      service:
        ipFamilies:
          - IPv6
```

性能调优

资源分配建议

小规模（**< 300 QPS**）

```
spec:
  config:
    controller:
      resources:
        limits:
          cpu: 200m
          memory: 256Mi
        requests:
          cpu: 200m
          memory: 256Mi
```

中等规模（**< 10,000 QPS**）

```
spec:
  config:
    controller:
      resources:
        limits:
          cpu: "2"
          memory: 1Gi
        requests:
          cpu: "2"
          memory: 1Gi
```

大规模（**< 20,000 QPS**）

```yaml
spec:
  config:
    controller:
      resources:
        limits:
          cpu: "4"
          memory: 2Gi
        requests:
          cpu: "4"
          memory: 2Gi
```

高性能（无限制）

最大性能且无 CPU 限制：

```yaml
spec:
  config:
    controller:
      resources:
        requests:
          cpu: "4"
          memory: 2Gi
```

重要说明

限制

- 每个集群仅支持一个 Ingress NGINX Controller 实例

版本兼容性

当前版本对应关系：
ACP Ingress-NGINX 4.1.x 对应官方 Chart 4.12.2（controller v1.12.3）

相关资源

tasks for ingress-nginx

官方 Ingress NGINX Chart↗

官方 Ingress NGINX 文档↗

通过 Ingress Nginx Operator 安装 Ingress-Nginx

目录

Overview

Installation

Configuration Via Web Console

Configuration Via YAML

Ingress Sharding

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

相关

与官方 Chart 默认值的差异

Overview

Ingress Nginx Operator 用于简化 Ingress-Nginx 部署的高级管理。该 Operator 简化了 Ingress-Nginx 实例的部署、配置和维护流程。其运行架构如下：Ingress-Nginx Operator 持续监控类型为 `ingressnginxes.ingress-nginx.alauda.io`（IngressNginx）的自定义资源，并自动为

每个定义的 IngressNginx 资源创建对应的 Ingress-Nginx 实例。该架构允许直接操作 IngressNginx 自定义资源来管理 Ingress-Nginx 的功能和行为。

Installation

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

Configuration Via Web Console

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

完成 Ingress Nginx Operator 安装后：

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

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

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

Configuration Via YAML

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

Ingress Sharding

默认情况下，Ingress Nginx 会处理所有命名空间中明确指定其 IngressClass 名称或未指定任何 IngressClass 的 Ingress 资源。您可以使用 `.controller.scope.namespaceSelector` 限制实例只处理特定命名空间。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. 格式为 `$LABLE_KEY=$LABLE_VALUE`

通过 **LoadBalancer** 暴露

默认情况下，Ingress Controller 配置为 ClusterIP 类型的服务。若要通过 LoadBalancer 服务将 Ingress Controller 暴露到外部，请应用以下配置：

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

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

使用 MetalLB 指定 LoadBalancer 虚拟 IP：

```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"  # 期望的 VIP
        metallb.universe.tf/address-pool: "pool-name"        # MetalLB 地址池
```

高级 Controller 部署设置

配置网络模式、副本数、资源限制和节点选择：

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

SSL 透传

启用 SSL 透传功能：

CAUTION

启用 SSL 透传时，TLS 在后端终止，因此 L7 功能（如请求/响应头操作、WAF、HTTP 到 HTTPS 重定向、部分认证流程）不会应用于该流量。

```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
```

相关

`IngressNginx` 资源的 `.spec` 字段直接对应 Ingress Nginx Helm chart 的 values。更多配置选项请参考 官方 Ingress NGINX 文档↗。

与官方 Chart 默认值的差异

1. 默认情况下，每个 IngressNginx 实例会创建一个名称为 `$ns-$name` 的 IngressClass，controllerValue 为 `ingress-nginx.cpaas.io/$ns-$name`。这些值可通过 `.spec.ingressClassResource.name` 和 `.spec.ingressClassResource.controllerValue` 参数自定义。

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`，意味着控制器会处理未指定 IngressClass 的 Ingress 资源。

Ingress-Nginx 的任务

目录

前提条件
最大连接数
超时
会话保持
头部修改
URL 重写
HSTS（HTTP 严格传输安全）
速率限制
WAF
转发头控制
HTTPS
TLS 重新加密并验证后端证书
TLS 边缘终止
透传
默认证书

前提条件

安装 ingress-nginx

最大连接数

max-worker-connections ↗

超时

配置超时 ↗

会话保持

配置会话保持 ↗

头部修改

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

URL 重写

rewrite ↗

HSTS（HTTP 严格传输安全）

配置 HSTS ↗

速率限制

配置速率限制 ↗

WAF

modsecurity ↗

转发头控制

x-forwarded-prefix-header ↗

HTTPS

TLS 重新加密并验证后端证书

验证后端 HTTPS 证书 ↗

TLS 边缘终止

后端协议 ↗

透传

ssl-passthrough ↗

默认证书

default-ssl-certificate ↗

Menu

ALB

Auth

- 基本概念
- 快速开始
- 相关 Ingress 注解
- forward-auth
- basic-auth
- CR
- ALB 特殊 Ingress 注解
- Ingress-Nginx Auth 相关其他特性
- 注意：与 Ingress-Nginx 不兼容的部分
- 故障排查

部署 ALB 的高可用 VIP

- 方式一：使用 LoadBalancer 类型的 Service 提供 VIP
- 方式二：使用外部 ALB 设备提供 VIP

Header Modification

- 基本概念
- 示例

HTTP 重定向

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

L4/L7 超时

- 基本概念
- CRD
- 超时的含义
- Ingress 注解
- 端口级超时

ModSecurity

- 术语
- 操作步骤
- 相关说明
- 配置示例

TCP/HTTP Keepalive

- 基本概念
- CRD

使用 OAuth Proxy 配合 ALB

- Overview
- Procedure
- Result

通过 ALB 配置 GatewayApi Gateway

- 术语
- 前提条件
- Gateway 和 Alb2 自定义资源（CR）示例
- 通过 Web 控制台创建 Gateway
- 通过 CLI 创建 Gateway
- 查看平台创建的资源
- 更新 Gateway
- 通过 Web 控制台更新 Gateway
- 添加监听器
- 通过 Web 控制台添加监听器
- 通过 CLI 添加监听器
- 创建路由规则
- HTTPRoute 自定义资源（CR）示例
- 通过 Web 控制台创建路由规则
- 通过 CLI 创建路由规则

在 ALB 中绑定网卡

- 对于集群内嵌 ALB
- 对于用户自定义 ALB

ALB 性能选择决策

- 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

部署 ALB

- ALB
- 监听端口（Frontend）
- 日志与监控

通过 ALB 将 IPv6 流量转发到集群内的 IPv4 地址

- 配置方法
- 结果验证

OTel

- 术语
- 前提条件
- 操作步骤
- 相关操作
- 附加说明
- 配置示例

ALB 监控

- 术语
- 操作步骤
- 监控指标

CORS

- 基本概念
- CRD

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

URL 重写

- 基本概念
- 配置

Auth

目录

基本概念

什么是 Auth

支持的 Auth 方式

Auth 配置方式

Auth 结果处理

快速开始

部署 ALB

配置 Secret 和 Ingress

验证

相关 Ingress 注解

forward-auth

构造相关注解

auth-url

auth-method

auth-proxy-set-headers

构造 app-request 相关注解

auth-response-headers

Cookie 处理

重定向签名相关配置

auth-signin

auth-signin-redirect-param

auth-request-redirect

basic-auth
auth-realm
auth-type
auth-secret
auth-secret-type
CR
ALB 特殊 Ingress 注解
Auth-Enable
Ingress-Nginx Auth 相关其他特性
Global-Auth
No-Auth-Locations
注意：与 Ingress-Nginx 不兼容的部分
故障排查

基本概念

什么是 **Auth**

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

支持的 **Auth** 方式

ALB 支持两种主要的身份验证方式：

1. **Forward Auth**（外部认证）
   - 向外部认证服务发送请求以验证用户身份
   - 适用场景：需要复杂的认证逻辑，如 OAuth、SSO 等
   - 工作流程：
     1. 用户请求到达 ALB

2. ALB 将认证信息转发给认证服务
3. 认证服务返回验证结果
4. 根据认证结果决定是否允许访问后端服务

2. **Basic Auth**（基本认证）

- 基于用户名和密码的简单认证机制
- 适用场景：简单访问控制，开发环境保护
- 工作流程：

1. 用户请求到达 ALB
2. ALB 检查请求中的用户名和密码
3. 与配置的认证信息进行比对
4. 验证通过则转发到后端服务

Auth 配置方式

1. 全局 **Auth**

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

2. 路径级别 **Auth**

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

3. 禁用 **Auth**

- 针对特定路径禁用 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：请求将被转发到后端服务
- Auth 失败：返回 401 unauthorized 错误
- 可以配置认证失败后的重定向行为（适用于 Forward Auth）

快速开始

使用 ALB 配置 Basic Auth

部署 **ALB**

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

配置 **Secret** 和 **Ingress**

```
# echo "Zm9vOiRhcHIxJHFJQ05aNjFRJDJpb29pSlZVQU1tcHJxMjU4L0NoUDE=" | base64 -d #
foo:$apr1$qICNZ61Q$2iooiJVUAMmprq258/ChP1
# openssl passwd -apr1 -salt qICNZ61Q bar # $apr1$qICNZ61Q$2iooiJVUAMmprq258/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://$ALB_IP:80/app-file # 应返回 200
# 错误密码
curl -v -X GET -H "Authorization: Basic XXXXOmJhcg==" http://$ALB_IP:80/app-file # 应返回 401
```

相关 **Ingress** 注解

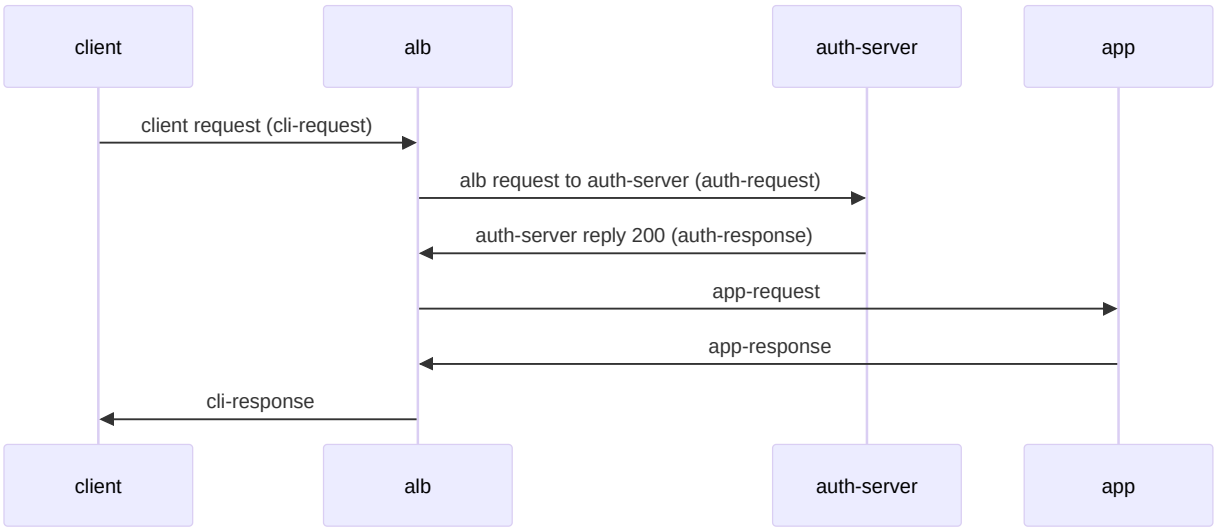
Ingress-nginx 定义了一系列注解，用于配置认证过程的具体细节。以下是 ALB 支持的注解列表，其中 "v" 表示支持，"x" 表示不支持。

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

	support	type	说明
nginx.ingress.kubernetes.io/auth-always-set-cookie	v	boolean	
nginx.ingress.kubernetes.io/auth-snippet	x	string	
basic-auth			通过用户名和密码 secret 实现认证
nginx.ingress.kubernetes.io/auth-realm	v	string	
nginx.ingress.kubernetes.io/auth-secret	v	string	
nginx.ingress.kubernetes.io/auth-secret-type	v	string	
nginx.ingress.kubernetes.io/auth-type	-	"basic" or "digest"	basic: 支持 apr1 **digest:** 不支持
auth-cache			
nginx.ingress.kubernetes.io/auth-cache-key	x	string	
nginx.ingress.kubernetes.io/auth-cache-duration	x	string	
auth-keepalive			发送请求时保持长连接。通过一系列注解指定 keepalive 行为
nginx.ingress.kubernetes.io/auth-keepalive	x	number	
nginx.ingress.kubernetes.io/auth-keepalive-share-vars	x	"true" or "false"	

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

forward-auth



相关注解：

- nginx.ingress.kubernetes.io/auth-url
- 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

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

构造相关注解

auth-url

auth-request 的 URL，值可以是变量。

auth-method

auth-request 的请求方法。

auth-proxy-set-headers

值为 ConfigMap 引用，格式为 `ns/name`。 默认情况下，cli-request 的所有请求头都会发送给 auth-server。可以通过 proxy_set_header 配置额外的请求头。默认发送的请求头如下：

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

构造 **app-request** 相关注解

auth-response-headers

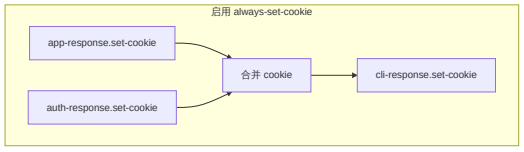
值为逗号分隔的字符串，允许将 auth-response 中的特定请求头带入 app-request。 示例：

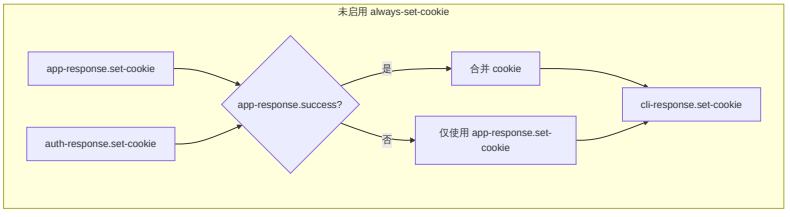
```
nginx.ingress.kubernetes.io/auth-response-headers: Remote-User,Remote-Name
```

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

Cookie 处理

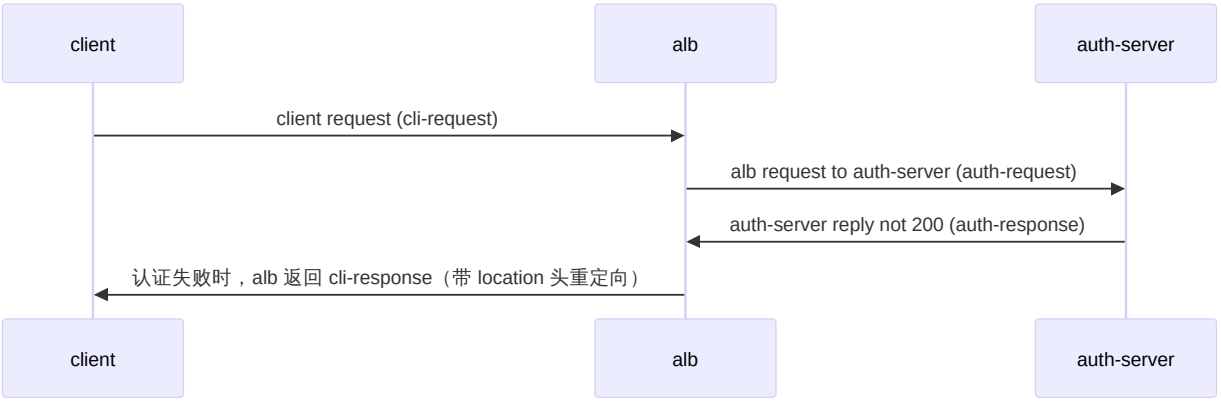
auth-response 和 app-response 都可以设置 cookie。默认情况下，仅当 app-response.success 时，auth-response.set-cookie 会合并到 cli-response.set-cookie 中。





重定向签名相关配置

当 auth-server 返回 401 时，可以在 cli-response 中设置 location 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，指定 cli-response 中的 location 头。

auth-signin-redirect-param

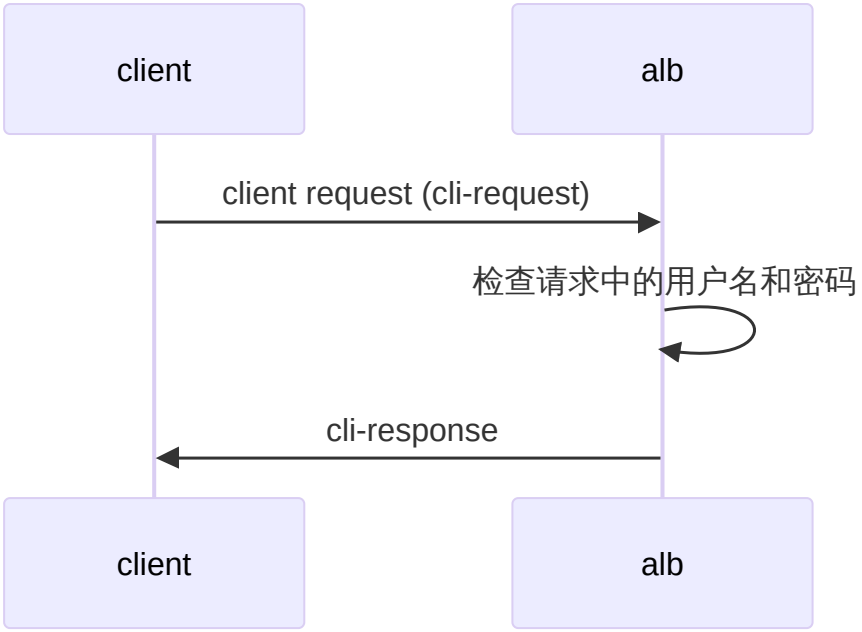
signin-url 中的查询参数名称，默认是 rd。 如果 signin-url 中不包含 `auth-signin-redirect-param` 指定的参数名，alb 会自动添加该参数。参数值设置为 `$pass_access_scheme://$http_host$escaped_request_uri`，用于记录原始请求 URL。

auth-request-redirect

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

basic-auth

basic-auth 是 RFC 7617↗ 中描述的认证过程。 交互流程如下：



auth-realm

受保护区域的描述↗ 即 cli-response 中 `WWW-Authenticate` 头的 realm 值。 示例：WWW-Authenticate: Basic realm="$realm"

auth-type

认证方案的类型，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 中每个键表示用户名，对应的值是密码哈希（htpasswd 格式中不包含用户名）。例如：

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

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

CR

ALB CR 已新增与 auth 相关的配置项，可在 ALB/Frontend/Rule CR 上配置。 运行时，ALB 会将 Ingress 上的注解转换为规则。

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

Auth 支持在以下位置配置：

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

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

ALB 特殊 Ingress 注解

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

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

这可以解决与 ingress-nginx 的兼容性问题，并支持在特定 Ingress 路径上指定 auth 配置。

Auth-Enable

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

ALB 新增的注解，用于指定是否启用该 Ingress 的认证功能。

Ingress-Nginx Auth 相关其他特性

Global-Auth

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

No-Auth-Locations

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

注意：与 Ingress-Nginx 不兼容的部分

1. 不支持 auth-keepalive

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

故障排查

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

部署 ALB 的高可用 VIP

ALB 的高可用需要一个 VIP，有两种方式获取 VIP。

目录

方式一：使用 LoadBalancer 类型的 Service 提供 VIP

方式二：使用外部 ALB 设备提供 VIP

方式一：使用 LoadBalancer 类型的 Service 提供 VIP

在以 `container` 网络模式创建 ALB 时，系统会自动创建一个 LoadBalancer 类型的 Service，为该 ALB 提供 VIP。

使用前请确保集群支持 LoadBalancer Service，可以使用平台内置实现。具体配置请参见 配置 MetalLB。

MetalLB 准备好后，可以在 `alb.spec.config.vip.lbSvcAnnotations` 添加以下注解来调整 MetalLB 行为。详见 ALB 网络配置。

注解	说明
`metallb.universe.tf/loadBalancerIPs`	分配给 Service 的 VIP，多个用逗号分隔，支持多 VIP 或双栈，例如：`192.0.2.10,2001:db8::10`。

注解	说明
`metallb.universe.tf/address-pool`	MetalLB 分配地址的地址池。

方式二：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。如果想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，否则可以使用纯软件数据中心 LB 方案（Alpha）。

- 根据业务场景，外部 ALB 需要对所有使用的端口配置健康检查，以减少 ALB 升级时的停机时间。健康检查配置如下：

健康检查参数	说明
端口	• 全局集群填写：11782。 • 业务集群填写：1936。
协议	健康检查的协议类型，建议使用 TCP。
响应超时	接收健康检查响应所需时间，建议配置为 2 秒。
检查间隔	健康检查的时间间隔，建议配置为 5 秒。
不健康阈值	连续失败次数，达到该次数后判定后端服务器健康检查失败，建议配置为 3 次。

Header Modification

目录

基本概念
使用注解
示例

基本概念

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

使用注解

目标	注解键
ingress	`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`

注解值是包含配置的 JSON 字符串。

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

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

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

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

ALB 提供了额外的变量：

变量名	描述
`first_forward_or_remote_addr`	第一个转发地址或远程地址，默认是 `remote_addr`
`first_forward`	第一个转发地址，默认是空字符串

示例

要从 cookie 中添加 Authorization 头，可以使用：

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

要设置 HSTS，可以使用：

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

HTTP 重定向

目录

基本概念

CRD

Ingress 注解

SSL-Redirect

端口级别重定向

规则级别重定向

基本概念

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

ALB 支持在端口和规则级别配置重定向。

CRD

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

重定向可以配置在：

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

Ingress 注解

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

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-Redirect

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

端口级别重定向

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

规则级别重定向

当在规则级别配置重定向时，匹配该规则的请求将根据重定向配置进行重定向。

L4/L7 超时

目录

基本概念

CRD

超时的含义

Ingress 注解

端口级超时

基本概念

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

超时通过 Lua 脚本实现，修改时 不需要重新加载 Nginx。

CRD

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

配置可以应用于：

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

超时的含义

超时分为三种类型：

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

Ingress 注解

注解	说明
nginx.ingress.kubernetes.io/proxy-connect-timeout	对应 CR 中的 proxy_connect_timeout_ms

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

端口级超时

您可以直接在端口上配置超时，该配置作为 L4 超时使用。

ModSecurity

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

目录

术语
操作步骤
方法一：添加注解
方法二：配置 CR
相关说明
覆盖规则
配置示例

术语

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

操作步骤

通过在对应资源的 YAML 文件中添加注解或配置 CR 来配置 ModSecurity。

方法一：添加注解

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

- **Ingress-Nginx** 兼容注解

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

- **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；设置

注解	类型	适用对象	说明
			为 `true` 以应用预定义的安全规则集。
alb.modsecurity.cpaas.io/cmref	string	Ingress	引用特定配置，例如通过指定 ConfigMap 的引用路径（`$ns/$name#$section`）加载自定义安全配置。

方法二：配置 CR

1. 打开需要配置的 ALB、FT 或 Rule 配置文件。

2. 根据需要在 spec.config 下添加以下字段。

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

3. 保存并应用配置文件。

相关说明

覆盖规则

如果 Rule 中未配置 ModSecurity，则会尝试在 FT 中查找配置；如果 FT 中也没有配置，则使用 ALB 中的配置。

配置示例

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

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

TCP/HTTP Keepalive

目录

基本概念

CRD

基本概念

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

CRD

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

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

使用 OAuth Proxy 配合 ALB

目录

Overview

Procedure

Result

Overview

本文档演示如何使用 OAuth Proxy 配合 ALB 实现外部认证。

Procedure

按照以下步骤使用该功能：

1. 部署 kind

```
kind create cluster --name alb-auth --image=kindest/node:v1.28.0
kind get kubeconfig --name=alb-auth > ~/.kube/config
```

2. 部署 alb

```
helm repo add alb https://alauda.github.io/alb/;helm repo update;helm search repo|grep alb
helm install alb-operator alb/alauda-alb2
alb_ip=$(docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' alb-auth-control-plane)
echo $alb_ip
cat <<EOF | kubectl apply -f -
apiVersion: crd.alauda.io/v2
kind: ALB2
metadata:
    name: alb-auth
spec:
    address: "$alb_ip"
    type: "nginx"
    config:
        networkMode: host
        loadbalancerName: alb-demo
        projects:
        - ALL_ALL
        replicas: 1
EOF
```

3. 部署测试应用

- 创建 github oauth app ↗

> 注意此步骤中会获取 `$GITHUB_CLIENT_ID` 和 `$GITHUB_CLIENT_SECRET`，需要将其设置为环境变量

- 配置 dns

> 此处使用 echo.com 作为应用域名，auth.alb.echo.com 和 alb.echo.com

- 部署 oauth-proxy

> oauth2-proxy 需要访问 github，可能需要设置 HTTPS_PROXY 环境变量

```
COOKIE_SECRET=$(python -c 'import os,base64;
print(base64.urlsafe_b64encode(os.urandom(32)).decode())')
OAUTH2_PROXY_IMAGE="quay.io/oauth2-proxy/oauth2-proxy:v7.7.1"
kind load docker-image $OAUTH2_PROXY_IMAGE --name alb-auth
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: oauth2-proxy
  name: oauth2-proxy
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: oauth2-proxy
  template:
    metadata:
      labels:
        k8s-app: oauth2-proxy
    spec:
      containers:
        - args:
            - --http-address=0.0.0.0:4180
            - --redirect-url=http://auth.alb.echo.com/oauth2/callback
            - --provider=github
            - --whitelist-domain=.alb.echo.com
            - --email-domain=*
            - --upstream=file:///dev/null
            - --cookie-domain=.alb.echo.com
            - --cookie-secure=false
            - --reverse-proxy=true
          env:
            - name: OAUTH2_PROXY_CLIENT_ID
              value: $GITHUB_CLIENT_ID
            - name: OAUTH2_PROXY_CLIENT_SECRET
              value: $GITHUB_CLIENT_SECRET
            - name: OAUTH2_PROXY_COOKIE_SECRET
              value: $COOKIE_SECRET
          image: $OAUTH2_PROXY_IMAGE
          imagePullPolicy: IfNotPresent
          name: oauth2-proxy
          ports:
```

```
            - containerPort: 4180
              name: http
              protocol: TCP
            - containerPort: 44180
              name: metrics
              protocol: TCP
---
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: oauth2-proxy
  name: oauth2-proxy
spec:
 ports:
 - appProtocol: http
   name: http
   port: 80
   protocol: TCP
   targetPort: http
 - appProtocol: http
   name: metrics
   port: 44180
   protocol: TCP
   targetPort: metrics
 selector:
   k8s-app: oauth2-proxy
EOF
```

4. 配置 ingress

   我们将配置两个 ingress，auth.alb.echo.com 和 alb.echo.com

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/auth-url: "https://auth.alb.echo.com/oauth2/auth"
    nginx.ingress.kubernetes.io/auth-signin: "https://auth.alb.echo.com/oauth2/start?
rd=http://\$host\$request_uri"
  name: echo-resty
spec:
  ingressClassName: alb-auth
  rules:
    - host: alb.echo.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: echo-resty
                port:
                  number: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: oauth2-proxy
spec:
  ingressClassName: alb-auth
  rules:
    - host: auth.alb.echo.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: oauth2-proxy
                port:
                  number: 80
EOF
```

Result

- 操作完成后，将部署 alb、oauth-proxy 和测试应用。
- 访问 alb.echo.com 后，会被重定向到 github 认证页面，验证通过后即可看到应用输出。

通过 ALB 配置 GatewayApi Gateway

入站网关（Gateway）是从 Gateway Class 创建的实例。它创建监听器以捕获指定域名和端口的外部流量。结合路由规则，可以将指定的外部流量路由到对应的后端实例。

创建入站网关以实现更细粒度的网络资源分配。

目录

术语

前提条件

Gateway 和 Alb2 自定义资源（CR）示例

通过 Web 控制台创建 Gateway

通过 CLI 创建 Gateway

查看平台创建的资源

更新 Gateway

通过 Web 控制台更新 Gateway

添加监听器

前提条件

通过 Web 控制台添加监听器

通过 CLI 添加监听器

创建路由规则

HTTPRoute 自定义资源（CR）示例

通过 Web 控制台创建路由规则

通过 CLI 创建路由规则

术语

资源名称	概述	使用说明
Gateway Class	在标准 Gateway API 文档中，Gateway Class 被定义为创建网关的模板。不同的模板可创建适用于不同业务场景的入站网关，便于快速流量管理。	平台内置专用的 Gateway Class。
入站网关	入站网关对应具体的资源实例，用户可独占使用该入站网关的所有监听和计算资源。它是对监听器生效的路由规则配置。当网关检测到外部流量时，会根据路由规则分发到后端实例。	可视为负载均衡器实例。
路由规则	路由规则定义了从网关到服务的流量分发一系列准则。Gateway API 当前标准支持的路由规则类型包括 HTTPRoute、TCPRoute、UDPRoute 等。	平台当前支持监听 HTTP、HTTPS、TCP 和 UDP 协议。

前提条件

平台管理员需确保集群支持 LoadBalancer 类型的内部路由。对于公有云集群，必须安装 LoadBalancer Service Controller。在非公有云集群，平台提供外部地址池功能，配置完成后，LoadBalancer 类型的内部路由可自动从外部地址池获取 IP 以供外部访问。

Gateway 和 **Alb2** 自定义资源（**CR**）示例

```yaml
# demo-gateway.yaml
apiVersion: gateway.networking.k8s.io/v1beta1
kind: Gateway
metadata:
  namespace: k-1
  name: test
  annotations:
    cpaas.io/display-name: ces
  labels:
    alb.cpaas.io/alb-ref: test-o93q7 (1)
spec:
  gatewayClassName: exclusive-gateway (2)
  listeners:
    - allowedRoutes:
        namespaces:
          from: All
      name: gateway-metric
      protocol: TCP
      port: 11782
---
apiVersion: crd.alauda.io/v2beta1
kind: ALB2
metadata:
  namespace: k-1
  name: test-o93q7 (3)
spec:
  type: nginx
  config:
    enableAlb: false
    networkMode: container
    resources:
      limits:
        cpu: 200m
        memory: 256Mi
      requests:
        cpu: 200m
        memory: 256Mi
    vip:
      enableLbSvc: true
      lbSvcAnnotations: {}
    gateway:
      mode: standalone
      name: test (4)
```

1. 此网关使用的 ALB。
2. 见下方 Gateway Class 介绍。
3. `ALB2` 名称格式：`{gatewayName}-{random}`。
4. Gateway 名称。

通过 **Web** 控制台创建 **Gateway**

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Network** > **Inbound Gateway**。
3. 点击 **Create Inbound Gateway**。
4. 参考以下说明配置具体参数。

参数	说明
名称	入站网关的名称。
Gateway Class	Gateway Class 定义了网关的行为，类似于存储类（StorageClasses）的概念；它是集群资源。 **Dedicated**：入站网关对应特定资源实例，用户可使用该网关的所有监听和计算资源。
规格	可根据需求选择推荐使用场景或自定义资源限制。
访问地址	入站网关的地址，默认自动获取。
内部路由注解	用于声明 LoadBalancer 类型内部路由的配置或能力。具体注解信息请参考LoadBalancer 类型内部路由注解说明。

5. 点击 **Create**。

通过 **CLI** 创建 **Gateway**

```
kubectl apply -f demo-gateway.yaml
```

查看平台创建的资源

入站网关创建后，平台会自动创建大量资源。请勿删除以下资源。

默认创建资源	名称
ALB2 类型资源	*name-lb-random*
Deployment	*name-lb-random*
内部路由	• *name-lb-random* • *name-lb-random-lb-random*
配置字典	• *name-lb-random-port-info* • *name-lb-random*
Service Account	*name-lb-random-serviceaccount*

更新 **Gateway**

NOTE

更新入站网关会导致 3-5 分钟的服务中断，请选择合适时间进行操作。

通过 **Web** 控制台更新 **Gateway**

1. 进入 **Container Platform**。

2. 在左侧导航栏点击 **Network** > **Inbound Gateway**。

3. 点击⋮> **Update**。

4. 根据需要更新入站网关配置。

   注意：请根据业务需求合理设置规格。

5. 点击 **Update**。

添加监听器

监控指定域名下的流量，并根据绑定的路由规则转发到后端实例。

前提条件

- 需要监控 HTTP 协议时，请提前联系管理员准备 域名。
- 需要监控 HTTPS 协议时，请提前联系管理员准备 域名 和 证书。

通过 **Web** 控制台添加监听器

1. 在左侧导航栏点击 **Network** > **Inbound Gateway**。

2. 点击 *入站网关名称*。

3. 点击 **Add Listener**。

4. 参考以下说明配置具体参数。

参数	说明
监听协议和端口	当前支持监听 HTTP、HTTPS、TCP 和 UDP 协议，可自定义输入监听端口，例如：`80`。 注意：

参数	说明
	• 端口相同时，HTTP、HTTPS 和 TCP 监听类型不能共存，只能选择其中一种协议。 • 使用 HTTP 或 HTTPS 协议时，端口相同，域名必须不同。
域名	选择当前命名空间内可用的域名，用于监听访问该域名的网络流量。 提示：TCP 和 UDP 协议不支持选择域名。

5. 点击 **Create**。

通过 **CLI** 添加监听器

```
kubectl patch gateway test \
  -n k-1 \
  --type=merge \
  -p '{
    "spec": {
      "listeners": [
        {
          "allowedRoutes": {
            "namespaces": {
              "from": "All"
            }
          },
          "name": "gateway-metric",
          "protocol": "TCP",
          "port": 11782
        },
        {
          "allowedRoutes": {
            "namespaces": {
              "from": "All"
            }
          },
          "name": "demo-listener",
          "protocol": "HTTP",
          "port": 8088,
          "hostname": "developer.test.cn"
        }
      ]
    }
  }'
```

创建路由规则

路由规则为入站流量提供路由策略，类似于入站规则（Kubernetes Ingress）。它们将网关监听到的网络流量暴露给集群内部路由（Kubernetes Service），便于实现路由转发策略。关键区别在于它们面向的服务对象不同：入站规则服务于 Ingress Controller，而路由规则服务于 Ingress Gateway。

Ingress Gateway 设置监听后，会实时监控指定域名和端口的流量。路由规则可将流量按需转发到后端实例。

HTTPRoute 自定义资源（CR）示例

```yaml
# example-httproute.yaml
apiVersion: gateway.networking.k8s.io/v1beta1
kind: HTTPRoute (1)
metadata:
  namespace: k-1
  name: example-http-route
  annotations:
    cpaas.io/display-name: ""
spec:
  hostnames:
    - developer.test.cn
  parentRefs:
    - kind: Gateway
      namespace: k-1
      name: test
      sectionName: demo-listener (2)
  rules:
    - matches:
        - path:
            type: Exact
            value: "/demo"
      filters: []
      backendRefs:
        - kind: Service
          name: test-service
          namespace: k-1
          port: 80
          weight: 100
```

1. 可用类型包括：`HTTPRoute`、`TCPRoute`、`UDPRoute`。

2. `Gateway` 监听器名称。

NOTE

如果 HTTPRoute 类型路由规则中 **Path** 对象无匹配规则，将自动添加一个 PathPrefix 模式且值为 / 的匹配规则。

通过 **Web** 控制台创建路由规则

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Network** > **Route Rules**。
3. 点击 **Create Route Rule**。
4. 按照以下说明配置部分参数。

参数	说明
路由类型	当前支持的路由类型有：HTTPRoute、TCPRoute、UDPRoute。 提示： HTTPRoute 支持发布到 HTTP 和 HTTPS 协议监听器。
发布到监听器	左侧选择框选择已创建的 **Ingress Gateway**，右侧选择框选择已创建的 **Listener**。平台会将创建的路由规则发布到该监听器下，使网关将捕获的流量转发到指定后端实例。 注意： 不允许发布路由规则到端口为 **11782** 或已挂载 TCP 或 UDP 路由的监听器。
匹配	可添加一个或多个匹配规则以捕获符合要求的流量，例如：捕获指定 **Path** 的流量、捕获指定方法的流量等。 注意： • 点击 **Add**；添加多个路由规则时，规则间关系为“与（AND）”，需全部匹配才生效。 • 点击 **Add Match**；添加多组路由规则时，组间关系为“或（OR）”，任一组匹配即可生效。 • TCPRoute 和 UDPRoute 不支持配置匹配规则。

参数	说明
	• 匹配对象为 **path**，匹配方式为 **Exact** 或 **PathPrefix** 时，输入的 **value** 必须以“/”开头，且不允许包含“//”、“/./”、“/../”、“%2f”、“%2F”、“#”、“/..”、“/.” 等字符。
动作	可添加一个或多个动作处理捕获的流量。 • Header：HTTP 消息头包含大量元数据，提供请求或响应的附加信息。通过修改头字段，服务器可影响请求和响应的处理方式。 • Redirect：匹配的 URL 将按指定方式处理后，重新发起请求。 • Rewrite：匹配的 URL 将按指定方式处理后，重定向到不同的资源路径或文件名。 注意： • 点击 **Add**；添加多个动作规则时，平台将按规则显示顺序依次执行所有动作。 • TCPRoute 和 UDPRoute 不支持配置动作规则。 • 同一路由规则内，不允许存在多个相同 **value** 的 **Header** 类型动作。 • 同一路由规则内，只能存在一种类型的 **Redirect** 或 **Rewrite**，且只能存在一种模式的 **FullPath** 或 **PrefixPath**。 • 若需使用 **PrefixPath** 操作，请先添加一个 **PathPrefix** 模式的匹配规则。
后端实例	规则生效后，将根据当前命名空间内选择的内部路由和端口转发到后端实例。可设置权重，权重值越高，被轮询的概率越大。 提示： 权重旁的百分比表示转发到该实例的概率，计算方式为当前权重值与所有权重值之和的比值。

5. 点击 **Create**。

通过 **CLI** 创建路由规则

```
kubectl apply -f example-httproute.yaml
```

在 ALB 中绑定网卡

默认情况下，ALB 监听 IPv4 地址 `0.0.0.0` 和 IPv6 地址 `::`。在某些安全场景下，需要将其绑定到特定的网络接口卡（NIC）。

目录

对于集群内嵌 ALB

对于用户自定义 ALB

对于集群内嵌 ALB

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

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

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

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

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

对于用户自定义 **ALB**

```
kubectl patch alb2 -n cpaas-system $ALB -p '{"spec":{"config":
{"enableIPV6":"false","bindNIC":"{\"nic\":[\"$NIC\",\"alive\"]}"}}}' --type=merge
```

ALB 性能选择决策

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

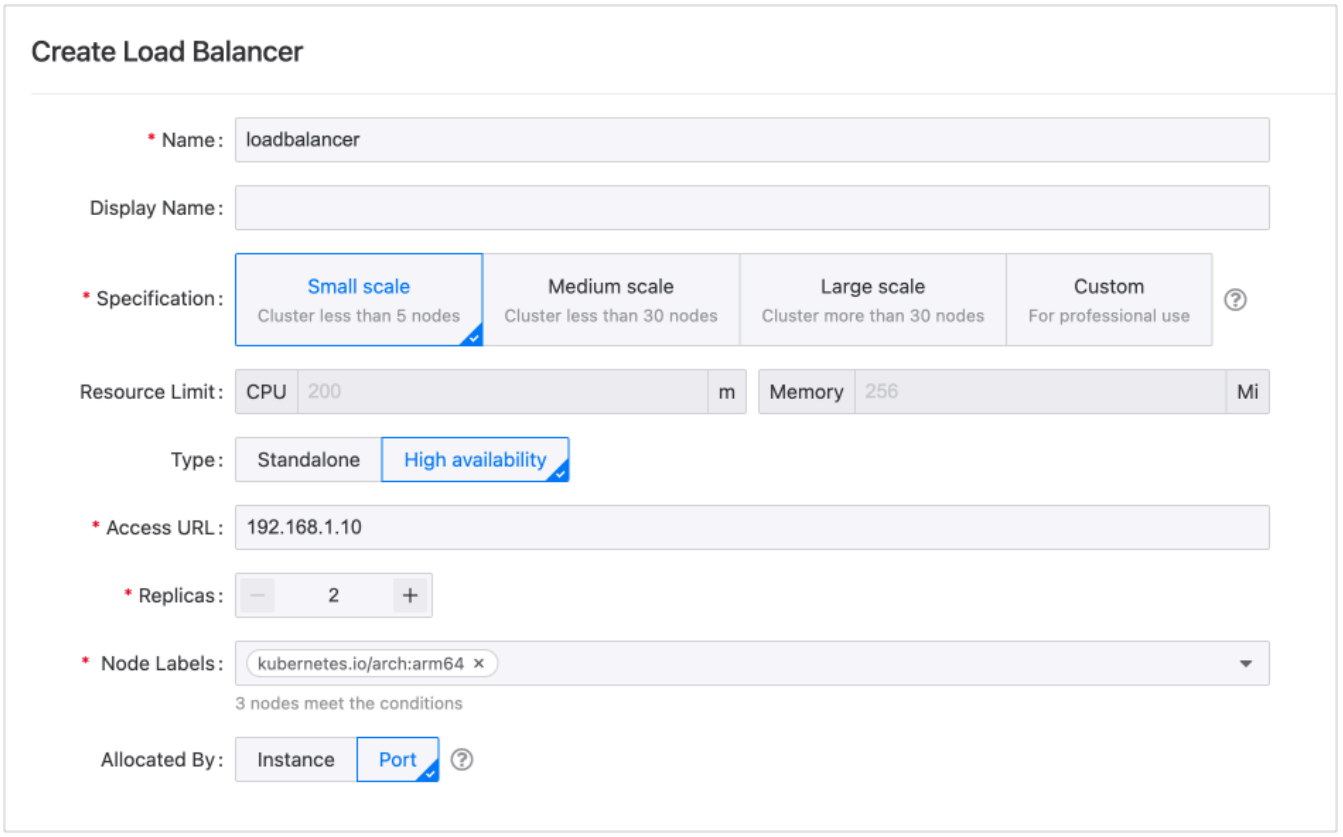
负载均衡器使用模式选择

小型生产环境

对于较小的业务规模，例如集群节点不超过 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，允许多个项目共享使用。

该规格在实验室环境下测得的峰值 QPS 约为每秒 300 次请求。

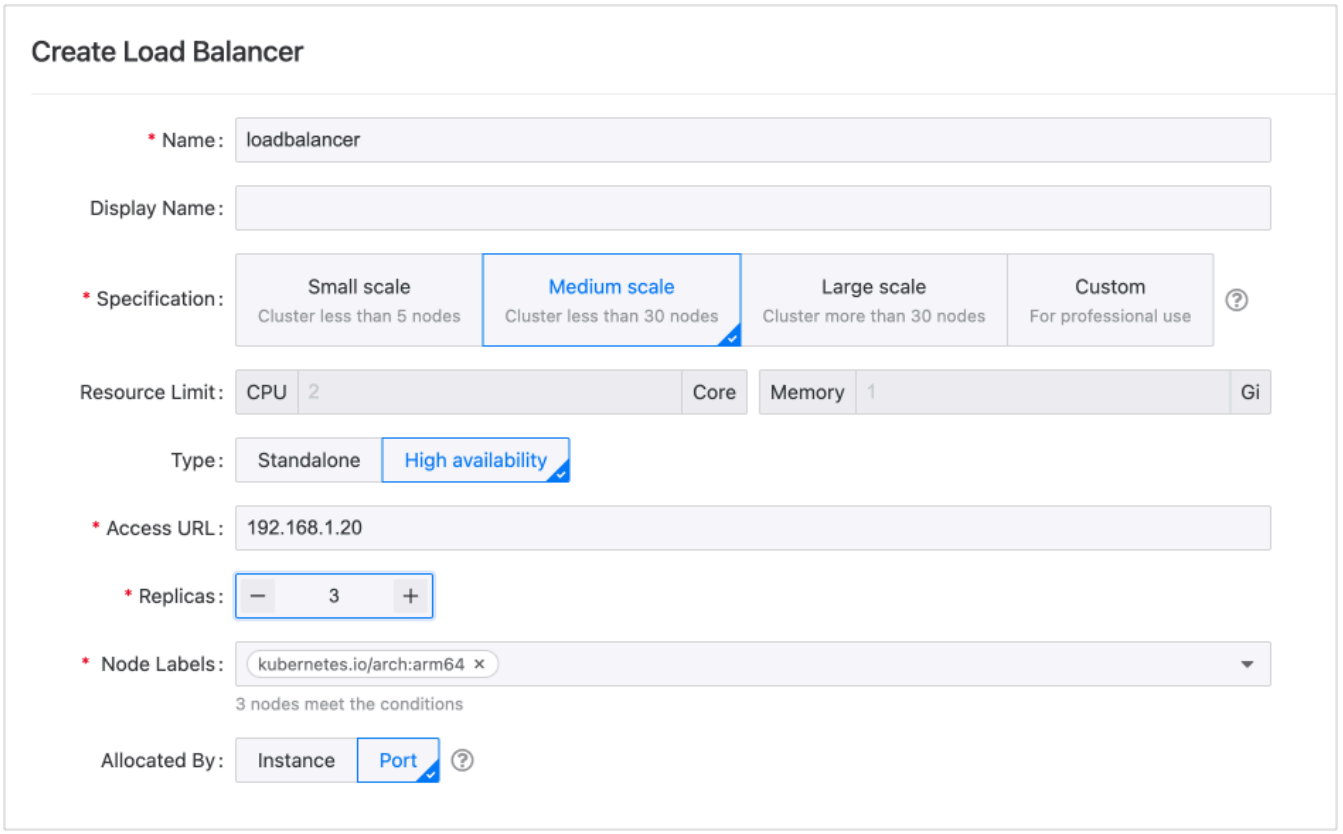


中型生产环境

当业务规模达到一定程度，例如集群节点不超过 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，以维持环境的稳定性。

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。

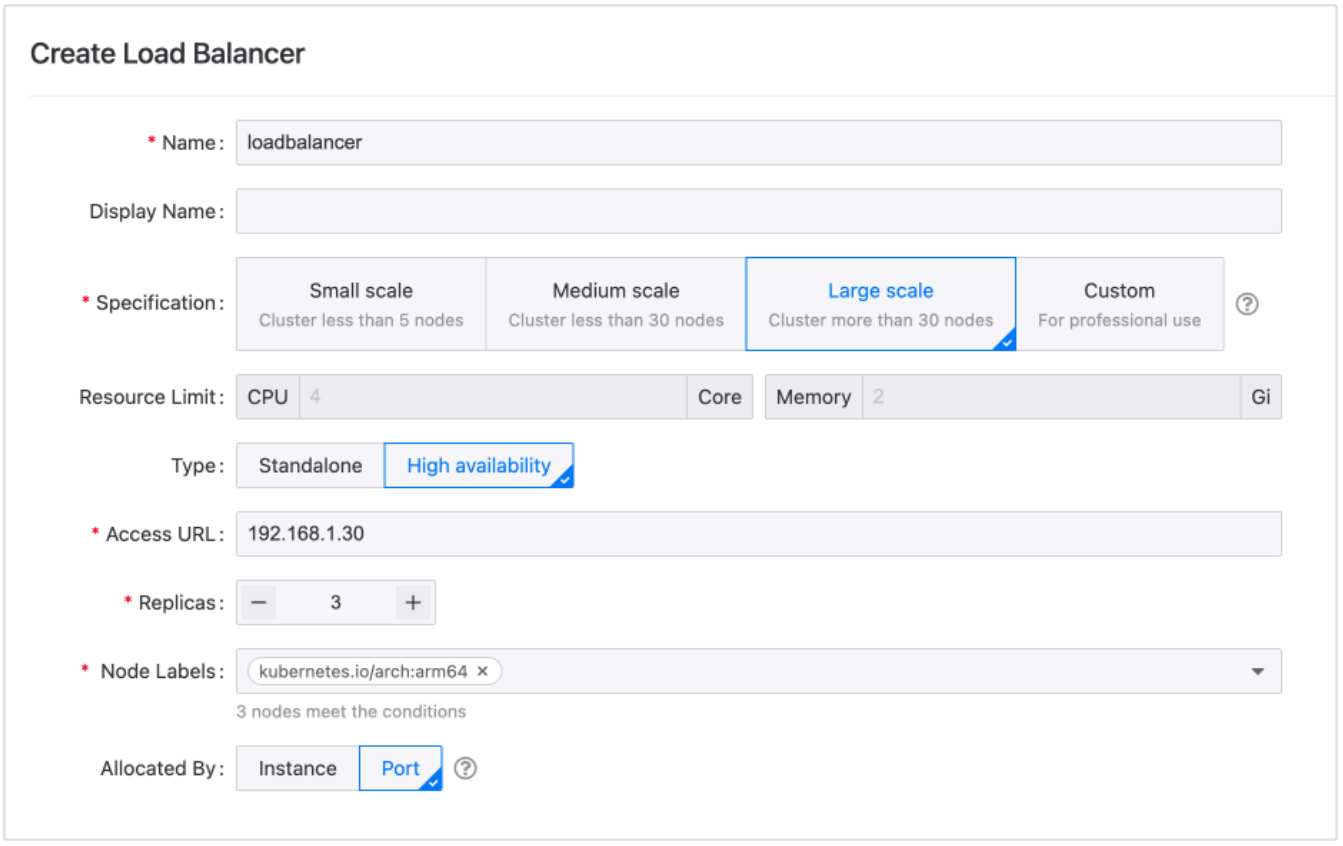


大型生产环境

对于更大规模的业务，例如集群节点超过 30 个，且需要处理高并发业务以及长连接数据，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，以确保环境的稳定性。

可以通过 端口 隔离或 实例 分配方式对负载均衡器进行隔离，供多个项目共享。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内存消耗较小，即使是 大型 规格也只需预留 2Gi 内存即可。但若业务涉及大规模数据传输，导致内存消耗较大，则应相应增加负载均衡器的内存分配。 建议在 自定义 规格场景下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。	由于所有分配的项目均拥有负载均衡器实例的完全权限，当某项目配置负载均衡器的端口和规则时，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。	• 管理复杂度提升。平台管理员需主动规划并分配项目端口，配置外部服务映射。 • 基于端口的分配成熟度较低，目前使用的客户较少，功能仍需进一步完善。 • 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。

部署 **ALB**

目录

ALB
- 前提条件
- 配置 ALB
  - 资源配置
  - 网络配置
  - 项目配置
  - 调整配置
- ALB 操作
  - 创建
  - 更新
  - 删除

监听端口（Frontend）
- 前提条件
- 配置 Frontend
- Frontend 操作
  - 创建
- 后续操作
- 相关操作

日志与监控
- 查看日志
- 监控指标

ALB

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

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

前提条件

负载均衡器 的高可用性需要 VIP。请参考 配置 VIP。

配置 ALB

ALB 配置包含三部分。

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

资源配置

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，可选，默认 3	alb 的副本数
`.spec.config.resources.limits`	k8s 容器资源，可选	alb 中 nginx 容器的资源限制
`.spec.config.resources.requests`	k8s 容器资源，可选	alb 中 nginx 容器的资源请求
`.spec.config.resources.alb.limits`	k8s 容器资源，可选	alb 中 alb 容器的资源限制
`.spec.config.resources.alb.requests`	k8s 容器资源，可选	alb 中 alb 容器的资源请求
`.spec.config.antiAffinityKey`	string，可选，默认 local	k8s 反亲和性键

网络配置

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

字段	类型	描述
`.spec.config.networkMode`	string: `host` 或 `container`，可选，默认 `host`	在 `container` 模式下，operator 会创建一个 LoadBalancer Service，并使用其地址作为 ALB 地址。
`.spec.address`	string，必填	可以手动指定 alb 的地址
`.spec.config.vip.enableLbSvc`	bool，可选	在 `container` 模式下自动为 true。
`.spec.config.vip.lbSvcAnnotations`	map[string]string，可选	LoadBalancer Service 的额外注解。

项目配置

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

将 ALB 添加到项目意味着：

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

如果启用端口项目并为项目分配端口范围，则意味着：

1. 不能创建不属于该项目分配端口范围的端口。

调整配置

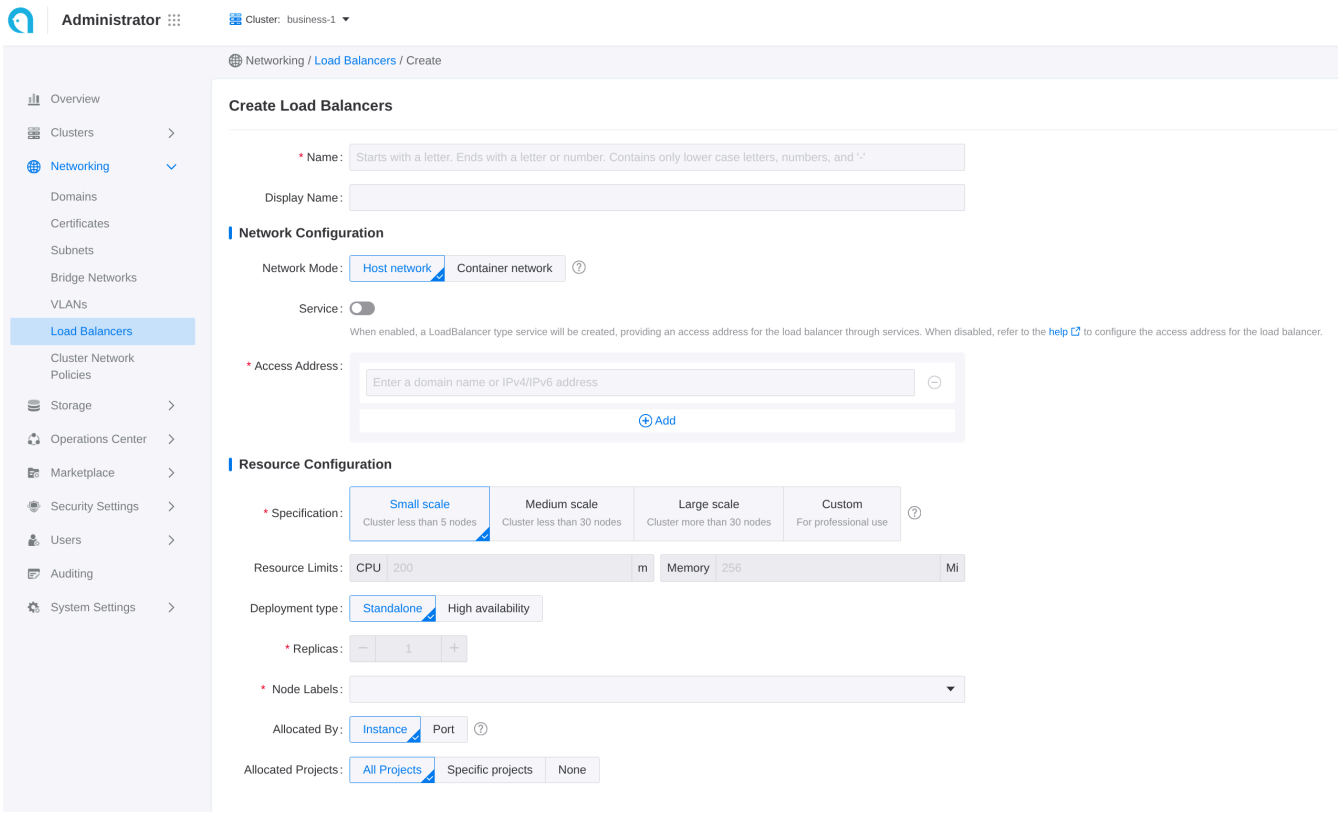
alb cr 中有一些全局配置可以调整。

- bind-nic
- ingress-sync

ALB 操作

创建

使用 **Web** 控制台



Web UI 中暴露了一些常用配置。创建负载均衡器的步骤如下：

1. 进入 **Administrator**。

2. 在左侧边栏点击 **Network Management** > **Load Balancer**。

3. 点击 **Create Load Balancer**。

Web UI 中的每个输入项对应 CR 的一个字段：

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

使用 **CLI**

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

更新

使用 **Web** 控制台

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

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

删除

使用 **Web** 控制台

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

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

使用 **CLI**

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

监听端口（Frontend）

Frontend 是定义 ALB 监听端口和协议的自定义资源。 支持的协议：L7（http|https|grpc|grpcs）和 L4（tcp|udp）。 L4 代理直接使用 frontend 配置后端服务。 L7 代理使用 frontend 配置监听端口，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 证书以实现加密。

前提条件

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 标签：必填，表示该 `Frontend` 所属的 ALB 实例。

2. frontend 名称：格式为 `$alb_name-$port`。

3. port：监听的端口。

4. protocol：该端口使用的协议。

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

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

6. backendProtocol：后端服务使用的协议。

7. 默认 `serviceGroup`：

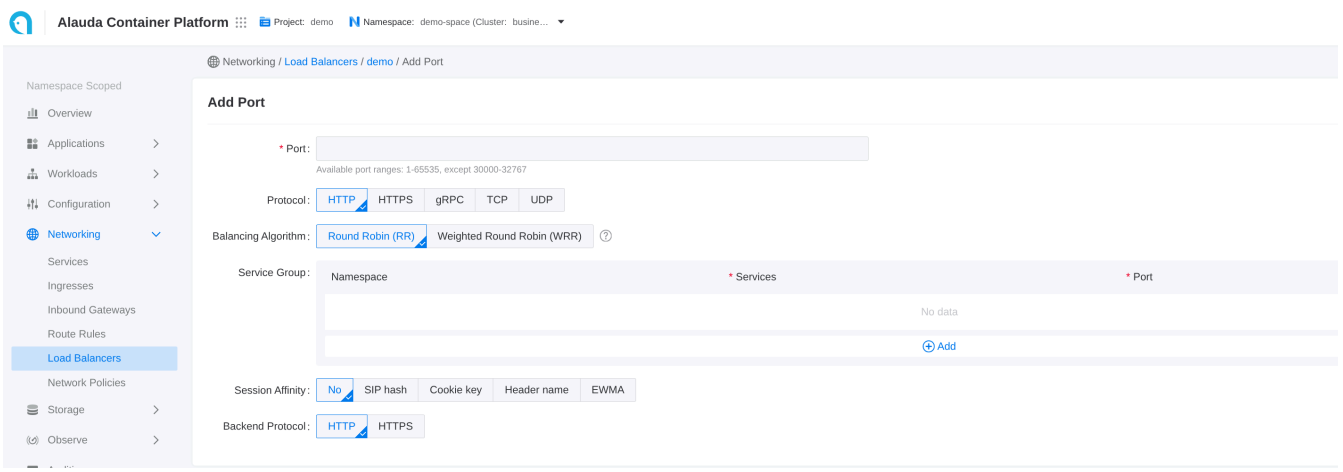
   - L4 代理：必填。ALB 直接将流量转发到默认服务组。
   - L7 代理：可选。ALB 先匹配该 Frontend 上的 Rules；若无匹配，则回退到默认 `serviceGroup`。

8. session_affinity_policy

Frontend 操作

创建

使用 **Web** 控制台



1. 进入 **Container Platform**。

2. 在左侧导航栏点击 **Network** > **Load Balancing**。

3. 点击负载均衡器名称进入详情页。

4. 点击 **Add Port**。

Web UI 中的每个输入项对应 CR 的字段：

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

使用 **CLI**

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

后续操作

对于 HTTP、gRPC 和 HTTPS 端口的流量，除了默认的内部路由组外，还可以设置更多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对应的后端服务；若规则匹配失败，则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，或详情页右上角的 **Actions**，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 **Port**，只有管理员可以在 **Administrator** 视图中删除相关监听端口。

日志与监控

结合日志和监控数据，可以快速定位和解决负载均衡器问题。

查看日志

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

监控指标

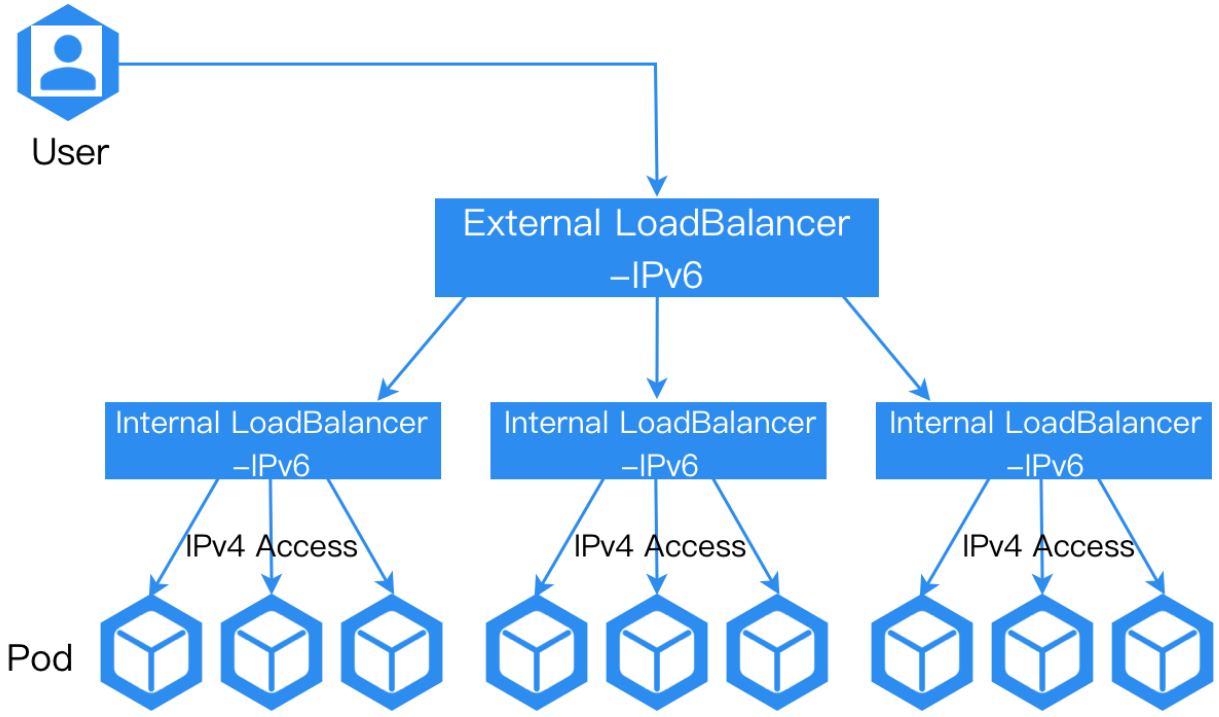
> **NOTE**
>
> 负载均衡器所在集群必须部署监控服务。

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

更多监控指标的详细信息请参考 ALB 监控。

通过 ALB 将 IPv6 流量转发到集群内的 IPv4 地址

通过为集群配置外部负载均衡器，我们可以将 IPv6 流量转发到集群内的 IPv4 地址。这使我们能够在现有的 IPv4 网络上引入 IPv6 功能，为系统架构提供更大的灵活性和可扩展性，更好地满足多样化的网络需求。



目录

配置方法

结果验证

配置方法

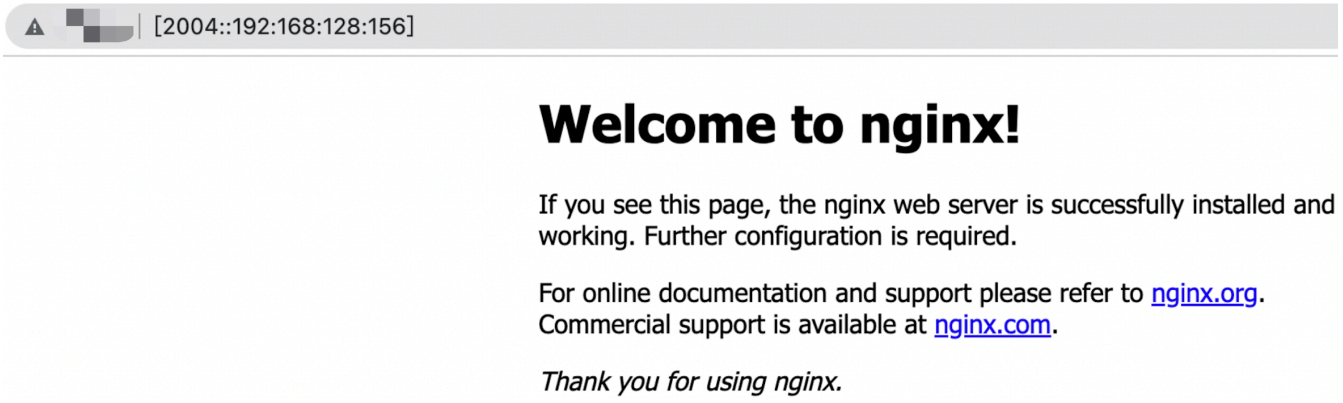
1. 配置负载均衡器所在节点的 IPv6 地址。

2. 确保外部负载均衡器具有 IPv6 地址，并确保访问负载均衡器 IPv6 地址的流量能够转发到负载均衡器所在节点的 IPv6 地址。

完成上述配置后，挂载在负载均衡器上的 IPv4 服务即可通过负载均衡器提供外部 IPv6 访问能力。

结果验证

配置完成后，访问外部负载均衡器的 IPv6 地址应能正常访问应用。



OTel

OpenTelemetry (OTel) 是一个开源项目，旨在为分布式系统（如微服务架构）中的遥测数据收集、处理和导出提供供应商中立的标准。它帮助开发者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
  - 继承关系
- 附加说明
  - 采样策略
  - Attributes
- 配置示例

术语

术语	说明
Trace	提交给 OTel Server 的数据，是一组相关事件或操作的集合，用于跟踪分布式系统中请求的流转；每个 Trace 由多个 Span 组成。
Span	Trace 中的独立操作或事件，包含开始时间、持续时间及其他相关信息。
OTel Server	能够接收和存储 Trace 数据的 OTel 服务器，如 Jaeger、Prometheus 等。
Jaeger	一个开源的分布式追踪系统，用于监控和排查微服务架构，支持与 OpenTelemetry 集成。
Attributes	附加在 Trace 或 Span 上的键值对，用于提供额外的上下文信息，包括 Resource Attributes 和 Span Attributes；详见 Attributes。
Sampler	决定是否采样并上报 Trace 的策略组件。可配置不同的采样策略，如全采样、比例采样等。
ALB (Another Load Balancer)	在集群中分发网络请求到可用节点的软件或硬件设备；平台中使用的负载均衡器（ALB）是七层软件负载均衡器，可配置为通过 OTel 监控流量。ALB 支持将 Trace 提交到指定的 Collector，并允许不同采样策略；还支持配置是否在 Ingress 级别提交 Trace。
FT (Frontend)	ALB 的端口配置，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；详情参见 Hot R.O.D. - Rides on Demand↗。
hotrod-with-proxy	通过环境变量指定 HotROD 内部微服务地址；详情参见 hotrod-with-proxy↗。

前提条件

- 确保存在可用的 **ALB**：创建或使用已有的 ALB，本文档中 ALB 名称用 `<otel-alb>` 代替。有关创建 ALB 的操作步骤，请参见 Deploy ALB。
- 确保存在 **OTel** 数据上报服务器地址：该地址以下称为 `<jaeger-server>`。

操作步骤

更新 **ALB** 配置

1. 在集群的 Master 节点，使用 CLI 工具执行以下命令编辑 ALB 配置。

```
kubectl edit alb2 -n cpaas-system <otel-alb> # 将 <otel-alb> 替换为实际的 ALB 名称
```

2. 在 `spec.config` 部分添加以下字段。

```
otel:
  enable: true
  exporter:
    collector:
      address: "<jaeger-server>" # 将 <jaeger-server> 替换为实际的 OTel 数据上报服务器地址
      request_timeout: 1000
```

完成后的示例配置：

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

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

```
:wq
```

相关操作

在 **Ingress** 中配置 **OTel**

- 启用或禁用 **Ingress** 上的 **OTel**

  通过配置是否启用 Ingress 上的 OTel，可以更好地监控和调试应用的请求流，追踪请求在不同服务间的传播，识别性能瓶颈或错误。

  操作步骤

  在 Ingress 的 metadata.annotations 字段下添加如下配置：

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

参数说明：

- **nginx.ingress.kubernetes.io/enable-opentelemetry**：设置为 `true` 表示 Ingress 控制器在处理该 Ingress 的请求时启用 OpenTelemetry 功能，即会收集并上报请求的 Trace 信息。设置为 `false` 或移除此注解则表示不收集或上报请求 Trace 信息。

- 启用或禁用 **Ingress** 上的 **OTel Trust**

  OTel Trust 决定 Ingress 是否信任并使用来自请求的 Trace 信息（如 trace ID）。

  操作步骤

  在 Ingress 的 metadata.annotations 字段下添加如下配置：

  ```
  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，使整个请求链路能在分布式追踪系统中完整追踪和分析。设置为 `false` 时，会为请求生成新的追踪信息，可能导致请求进入 Ingress 后被视为新的追踪链，打断跨服务追踪的连续性。

- 在 **Ingress** 上添加不同的 **OTel** 配置

  该配置允许针对不同的 Ingress 资源自定义 OTel 的行为和数据导出方式，实现对各服务追踪策略或目标的精细化控制。

  操作步骤

  在 Ingress 的 metadata.annotations 字段下添加如下配置：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    alb.ingress.cpaas.io/otel: >
      {
         "enable": true,
         "exporter": {
             "collector": {
                 "address": "<jaeger-server>", # 将 <jaeger-server> 替换为实际的 OTel
数据上报服务器地址，例如 "address": "http://128.0.0.1:4318"
                 "request_timeout": 1000
             }
         }
      }
```

参数说明：

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

在应用中使用 **OTel**

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

在集群 Master 节点，使用 CLI 工具执行以下命令获取完整的 OTel 配置结构。

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

回显结果：

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

参数说明：

参数	说明
otel.enable	是否启用 OTel 功能。
exporter.collector.address	OTel 数据上报服务器地址，支持 http/https 协议及域名。
flags.hide_upstream_attrs	是否上报上游规则相关信息。
flag.notrust_incoming_span	是否信任并使用来自请求的 OTel Trace 信息（如 trace ID）。
flags.report_http_request_header	是否上报请求头。
flags.report_http_response_header	是否上报响应头。

参数	说明
sampler.name	采样策略名称；详情见 Sampling Strategies。
sampler.options.fraction	采样率。
sampler.options.parent_name	parent_base 采样策略的父策略。

继承关系

默认情况下，如果 ALB 配置了某些 OTel 参数且 FT 未配置，则 FT 会继承 ALB 的参数作为自身配置；即 FT 继承 ALB 配置，而 Rule 可以继承 ALB 和 FT 的配置。

- **ALB**：ALB 上的配置通常是全局和默认的，可配置全局参数如 Collector 地址，供下层 FT 和 Rule 继承。
- **FT**：FT 可以继承 ALB 的配置，未配置的 OTel 参数将使用 ALB 配置。但 FT 也可进一步细化，例如选择性地启用或禁用 FT 上的 OTel，不影响其他 FT 或 ALB 的全局设置。
- **Rule**：Rule 可继承 ALB 和 FT 的配置，也可进一步细化，例如某条 Rule 可选择不信任传入的 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，即可添加 OTel 相关配置。

附加说明

采样策略

参数	说明
always on	始终上报所有追踪数据。

参数	说明
always off	从不上报追踪数据。
traceid-ratio	根据 `traceid` 决定是否上报。`traceparent` 格式为 `xx-traceid-xx-flag`，其中 traceid 的前 16 个字符代表一个 32 位十六进制整数。若该整数小于 `fraction` 乘以 4294967295（即 (2^32-1)），则上报。
parent-base	根据请求中 traceparent 的 flag 部分决定是否上报。flag 为 01 时上报，例如：`curl -v "http://$ALB_IP/" -H 'traceparent: 00-xx-xx-01'`；flag 为 02 时不上报，例如：`curl -v "http://$ALB_IP/" -H 'traceparent: 00-xx-xx-02'`。

Attributes

- 资源属性（**Resource Attributes**）

  这些属性默认上报。

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

- **Span** 属性

  - 默认上报的属性：

参数	说明
http.status_code	状态码

参数	说明
http.request.resend_count	重试次数
alb.rule.rule_name	本次请求匹配的规则名称
alb.rule.source_type	本次请求匹配的规则类型，当前仅支持 Ingress
alb.rule.source_name	Ingress 名称
alb.rule.source_ns	Ingress 所在命名空间

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

参数	说明
alb.upstream.svc_name	流量转发的 Service（内网路由）名称
alb.upstream.svc_ns	被转发的 Service（内网路由）所在命名空间
alb.upstream.peer	被转发 Pod 的 IP 地址和端口

- 默认不上报但可通过修改 flag.report_http_request_header 字段上报的属性：

参数	说明
`**http.request.header.<header>**`	请求头

- 默认不上报但可通过修改 flag.report_http_response_header 字段上报的属性：

参数	说明
`**http.response.header.<header>**`	响应头

配置示例

以下 YAML 配置部署一个 ALB，使用 Jaeger 作为 OTel 服务器，Hotrod-proxy 作为示范后端。通过配置 Ingress 规则，当客户端请求 ALB 时，流量将转发到 HotROD。同时，HotROD 内部

微服务间的通信也通过 ALB 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 及所有测试所需的 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"
```

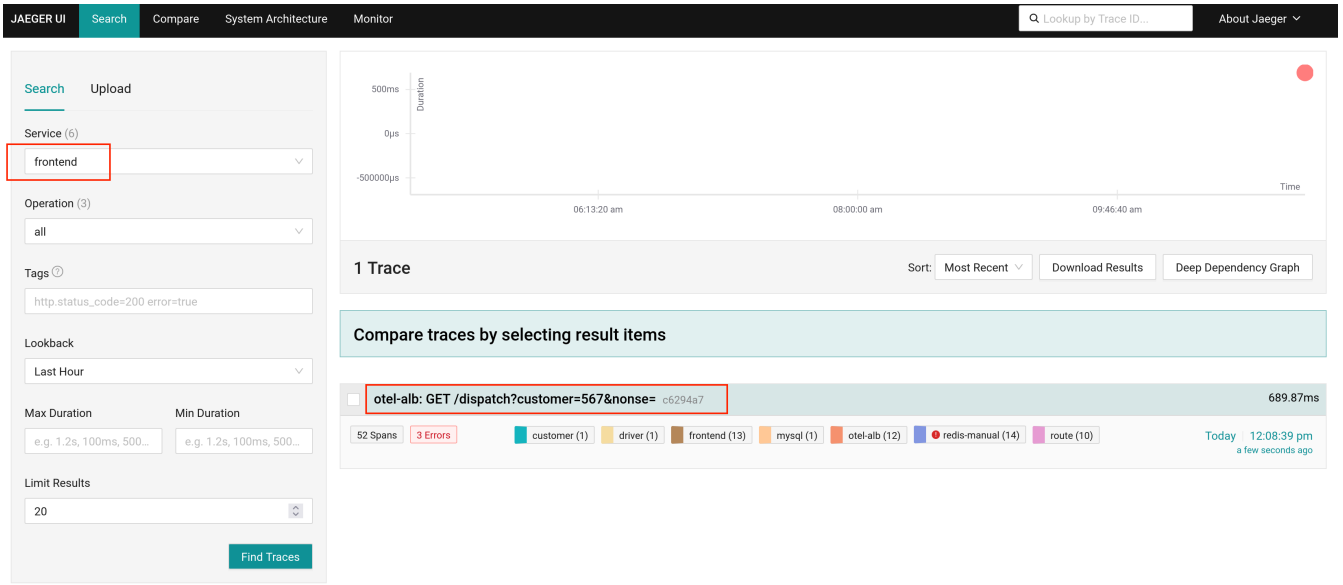
4. 执行以下命令获取 otel-alb 的访问地址。

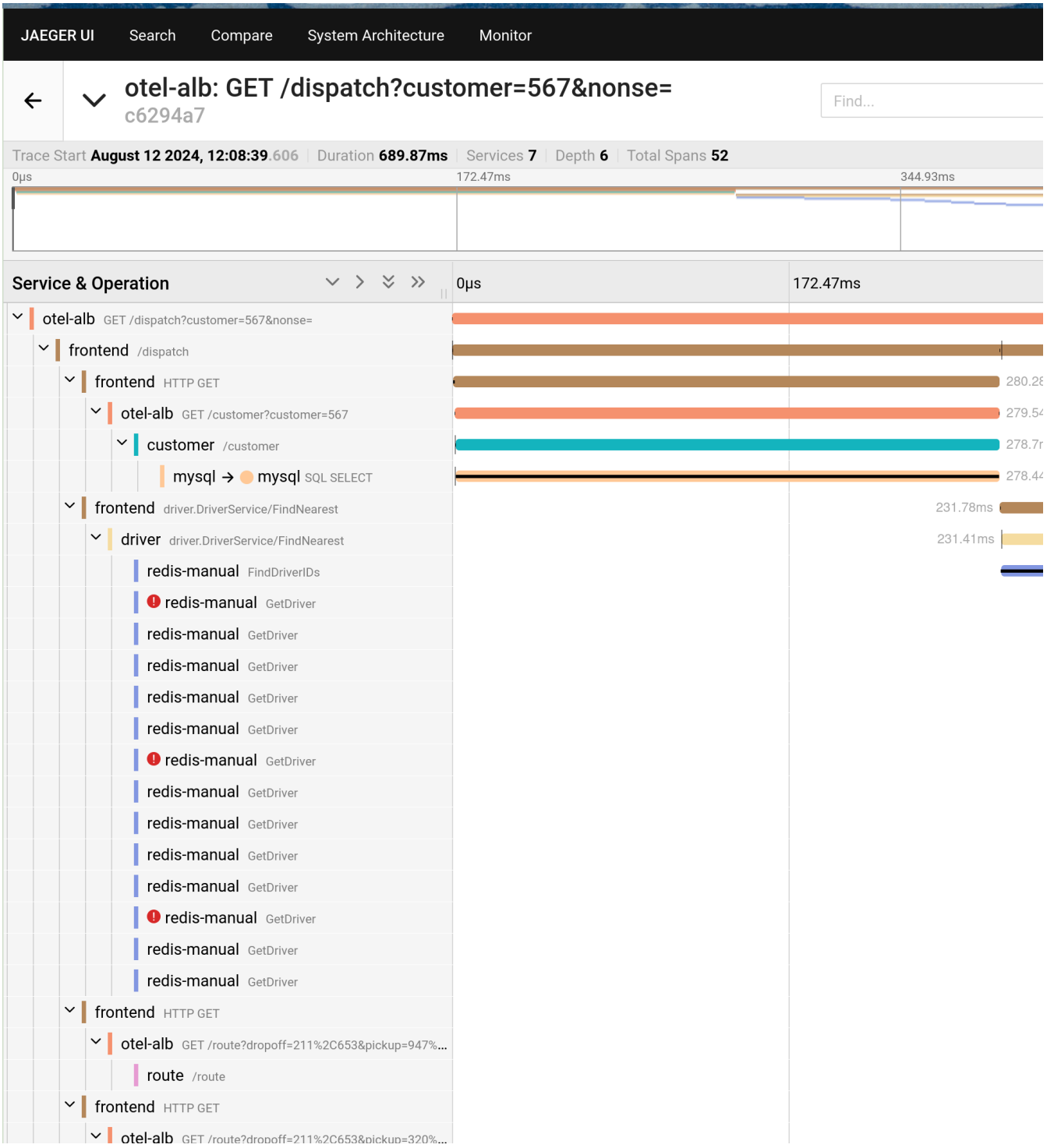
```
export ALB_IP=$(kubectl get po -A -o wide|grep otel-alb | awk '{print $7}');echo
$ALB_IP
```

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

```
curl -v "http://<$ALB_IP>:80/dispatch?customer=567&nonse=" # 将命令中的 <$ALB_IP> 替换
为上一步获取的 otel-alb 访问地址
```

6. 打开第 3 步 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


JAEGER UI
Search
Compare
System Architecture
Monitor
otel-alb: GET /dispatch?customer=567&nonse=
c6294a7
Find...
Trace Start August 12 2024, 12:08:39.606
Duration 689.87ms
Services 7
Depth 6
Total Spans 52
0µs
172.47ms
344.93ms
Service & Operation
0µs
172.47ms
otel-alb GET /dispatch?customer=567&nonse=
frontend /dispatch
frontend HTTP GET
otel-alb GET /customer?customer=567
customer /customer
mysql → mysql SQL SELECT
frontend driver.DriverService/FindNearest
231.78ms
driver driver.DriverService/FindNearest
231.41ms
redis-manual FindDriverIDs
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
redis-manual GetDriver
frontend HTTP GET
otel-alb GET /route?dropoff=211%2C653&pickup=947%...
route /route
frontend HTTP GET
otel-alb GET /route?dropoff=211%2C653&pickup=320%...

ALB 监控

目录

术语

操作步骤

监控指标

ALB 流量监控

ALB 资源使用情况

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. 你可以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。需要时可以将该监控面板设置为主面板。
- 方式二：在搜索框输入关键词（例如 alb）进行搜索，然后点击 **ALB** 状态 名称进入监控面板。需要时可以将该监控面板设置为主面板。

6. 通过监控面板查看各项监控指标。

- 选择监控的命名空间：点击页面顶部的 命名空间，选择要监控的命名空间，默认全部，表示监控所有命名空间。
- 选择监控的 **ALB**：点击页面顶部的 名称，选择要监控的 ALB，默认全部，表示监控所有 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 资源使用情况

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 每秒接收的请求总大小。
响应 **BPS**（每秒字节数）	所选 ALB 上 Ingress/HTTPRoute/Rule 发送的响应总大小。
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，取位于 50%、90% 和 99% 位置的数据值；因此，采集的 50%、90% 和 99% 的数据均低于该值。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。

监控指标	描述
上游 **P99**	

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）从资源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 允许的来源,
    当 enableCORS 为 true 时, 多个来源用逗号分隔
  type: string
```

它可以在规则的 `.spec` 上进行配置。

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 之间的流量分配。

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。
- 使用场景：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：将来自相同源 IP 地址的请求始终路由到同一个 Pod。
- 行为：如果存在 X-Forwarded-For 头，则使用该头中的第一个 IP，否则使用客户端的源 IP。
- 使用场景：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`（默认值为 `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 会将其添加到响应中
  - Cookie 格式为：`timestamp.worker_pid.random_number`
- 使用场景：适用于需要基于 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。
- 使用场景：适用于需要基于特定 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
- 使用场景：适合对延迟敏感的应用
- 参考：Twitter Finagle EWMA 文档↗

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 的会话持久性时，使用源 IP 哈希
- 对于需要会话粘性的 Web 应用，采用基于 Cookie 的亲和
- 对于响应时间波动较大的服务，考虑使用 EWMA 以优化延迟

URL 重写

目录

基本概念

配置

基本概念

ALB 可以在将请求转发到后端之前重写请求 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 rule

```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` ；后端接收到 `/abc` 。

Calico 网络支持 WireGuard 加密

Calico 支持对 IPv4 和 IPv6 流量进行 WireGuard 加密，可以通过 FelixConfiguration 资源中的参数独立启用。

目录

- 安装状态
  - 默认安装
  - 非默认安装
- 术语
- 注意事项
- 先决条件
- 操作步骤
- 结果验证
  - IPv4 流量验证

安装状态

默认安装

操作系统	内核版本
Linux	5.6 及以上版本默认安装

操作系统	内核版本
Ubuntu 20.04	5.4.0-135-generic
Kylin Linux Advanced Server V10 - SP3	4.19.90-52.22.v2207.ky10.x86_64

非默认安装

操作系统	内核版本
openEuler	4.18.0-147.5.2.13.h996.eulerosv2r10.x86_64
CentOS 7	3.10.0-1160.el7.x86_64
Redhat 8.7	4.18.0-425.3.1.el8.x86_64
Kylin Linux Advanced Server V10 - SP2	4.19.90-24.4.v2101.ky10.x86_64
Kylin Linux Advanced Server V10 - SP1	4.19.90-23.8.v2101.ky10.x86_64
Kylin Linux Advanced Server V10	4.19.90-11.ky10.x86_64

术语

术语	说明
wireguardEnabled	启用 IPv4 流量在 IPv4 下层网络上进行加密。
wireguardEnabledV6	启用 IPv6 流量在 IPv6 下层网络上进行加密。

注意事项

1. 使用 Calico 网络插件时，请确保 `natOutgoing` 参数设置为 `true`，以支持 WireGuard 加密。默认情况下，创建集群时该参数已正确配置，无需额外配置。

2. WireGuard 支持对 IPv4 和 IPv6 流量进行加密；如果需要对两种类型的流量进行加密，则必须分别进行配置。有关详细的参数配置，请参阅 Felix 配置文档↗，配置 `wireguardEnabled` 和 `wireguardEnabledV6` 两个参数。

3. 如果 WireGuard 未默认安装，请参阅 WireGuard 安装指南↗ 进行手动安装，尽管在某些情况下可能会出现 WireGuard 模块的手动安装失败。

4. 跨节点的容器间流量将被加密，包括从一个主机到另一个主机的网络流量；但是，在同一节点的 Pods 之间及 Pod 与其宿主节点之间的通信将不会被加密。

先决条件

- 必须事先在集群中的所有节点上安装 WireGuard。有关详情，请参阅 WireGuard 安装文档↗。未安装 WireGuard 的节点不支持加密。

操作步骤

1. 启用或禁用 IPv4 和 IPv6 加密。

   注意：以下命令必须在节点所在的 Master 节点的 CLI 工具中执行。

   - 仅启用 IPv4 加密

     ```
     kubectl patch felixconfiguration default --type='merge' -p '{"spec":
     {"wireguardEnabled":true}}'
     ```

   - 仅启用 IPv6 加密

     ```
     kubectl patch felixconfiguration default --type='merge' -p '{"spec":
     {"wireguardEnabledV6":true}}'
     ```

- 启用 IPv4 和 IPv6 加密

```
kubectl patch felixconfiguration default --type='merge' -p '{"spec":
{"wireguardEnabled":true,"wireguardEnabledV6":true}}'
```

- 同时禁用 IPv4 和 IPv6 加密
  - 方法 1：在 CLI 工具中执行命令以禁用加密。

```
kubectl patch felixconfiguration default --type='merge' -p '{"spec":
{"wireguardEnabled":false,"wireguardEnabledV6":false}}'
```

  - 方法 2：修改 felixconfiguration 配置文件以禁用加密。
    1. 执行以下命令以打开 felixconfiguration 配置文件。

```
kubectl get felixconfiguration -o yaml default
```

    2. 将 `wireguardEnabled` 和 `wireguardEnabledV6` 参数设置为 false 以禁用 WireGuard 加密。

```
apiVersion: crd.projectcalico.org/v1
kind: FelixConfiguration
metadata:
  annotations:
    projectcalico.org/metadata: '{"uid":"f5facabd-8304-46d6-81c1-
f1816235b487","creationTimestamp":"2024-08-06T03:46:51Z"}'
  generation: 2
  name: default
  resourceVersion: "890216"
spec:
  bpfLogLevel: ""
  floatingIPs: Disabled
  logSeverityScreen: Info
  reportingInterval: 0s
  wireguardEnabled: false        # 改为 true 以启用 IPv4 加密
  wireguardEnabledV6: false     # 改为 true 以启用 IPv6 加密
```

2. 完成 Calico WireGuard 加密配置后，执行以下命令确认 WireGuard 加密状态。如果 IPv4 和 IPv6 加密均已启用，`Status` 字段下存在 `wireguardPublicKey` 或 `wireguardPublicKeyV6` 表示成功激活；如果 IPv4 和 IPv6 加密均已禁用，则这些字段不会包含 `wireguardPublicKey` 或 `wireguardPublicKeyV6`，表示成功停用。

```
calicoctl get node <NODE-NAME> -o yaml # 将 <NODE-NAME> 替换为节点名称。
```

输出：

```
Status:
    wireguardPublicKey: L/MUP9+Yxx/xxxxxxxxxxx/xxxxxxxxxx =
```

结果验证

本文使用 IPv4 流量验证作为示例；IPv6 流量验证与 IPv4 类似，此处不再重复。

IPv4 流量验证

1. 配置 WireGuard 加密后，检查路由信息，节点之间的流量优先使用 wireguard.cali 接口进行消息转发。

```
root@test:~# ip rule   # 查看当前路由规则
     0: from all lookup local
     99:  not from all fwmark 0x100000/0x100000 lookup 1    # 对于所有未标记为
0x100000 的数据包，使用路由表 1 进行路由查找
     32766:  from all lookup main
     32767 :  from all lookup default

root@test:~# ip route show table 1    # 显示表 1 的路由条目。
    10.3.138.0 dev wireguard.cali scope link
    10.3.138.0/26 dev wireguard.cali scope link
    throw 10.3.231.192
    10.3.236.128 dev wireguard.cali scope link     # 到达 IP 地址 10.3.236.128 的流量
将通过 wireguard.cali 接口发送
    10.3.236.128/26 dev wireguard.cali scope link
    throw 10.10.10.124/30
    10.10.10.200/30 dev wireguard.cali scope link
    throw 10.10.20.124/30
    10.10.20.200/30 dev wireguard.cali scope link
    throw
    10.13.138.0 dev wireguard.cali scope link
    10.13.138.0/26 dev wireguard.cali scope link
    throw 10.13.231.192/26
    10.13.236.128 dev wireguard.cali scope link
    10.13.236.128/26 dev wireguard.cali scope link

root@test:~# ip r get 10.10.10.202    # 当前节点到目标 IP 地址 10.10.10.202 的路由路径
    10.10.10.202 dev wireguard.cali table 1 src 10.10.10.127 uid 0  cache   # 当从当前
节点访问目标 IP 地址 10.10.10.202 时，数据包将通过 wireguard.cali 接口发送，使用路由表
1，源地址将设置为 10.10.10.127

root@test:~# ip route    # 显示主路由表
    default via 192.168.128.1 dev eth0 proto static
    10.3.138.0/26 via 10.3.138.0 dev vxlan.
    blackhole 10.3.231.193
    10.3.231.194
    10.3.231.195
    10.3.231.196
    10.3.231.197
    3.231.192/26 proto 80
    dev cali8dcd31cIdOO scope link
    dev cali3012b5b29b scope link
    dev calibeefea2ff87 scope link
    dev cali2b27d5e4053 scope link
```

```
    dev cali1a35dbdd639 scope link
    calico on link
```

2. 在节点上捕获数据包以观察跨节点流量。

```
root@test:~# ip a s wireguard.cali    # 查看 wireguard.cali 网络接口的详细信息
    30: wireguard.cali: <POINTOPOINT,NOARP,UP,LOWER_UP> mtu 1440 qdisc noqueue state
UNKNOWN group default qlen 1000
    link/none
    inet 10.10.10.127/32 scope global wireguard.cali   # wireguard.cali 接口分配的 IP
地址为 10.10.10.127
    valid_lft forever preferred_lft forever

root@test:~# tcpdump -i wireguard.cali -nnve icmp   # 捕获并显示通过 wireguard.cali 的
ICMP 数据包
    tcpdump: listening on wireguard.cali, link-type RAW (Raw IP), capture size 262144
bytes
    08:58:36.987559 ip: (tos 0x0, ttl 63, id 29731, offset 0, flags [DF], proto ICMP
(1), length 84)
    10.10.10.125 > 10.10.10.202: ICMP echo request, id 1110, seq 0, length 64
    08:58:36.988683 ip: (tos 0x0, ttl 63, id 1800, offset 0, flags [none], proto ICMP
(1), length 84)
    10.10.10.202 > 10.10.10.125: ICMP echo reply, id 1110, seq 0, length 64
    2 packets captured
    2 packets received by filter
    0 packets dropped by kernel
```

3. 测试表明 IPv4 类型流量通过 wireguard.cali 接口转发。

Kube-OVN Overlay 网络支持 IPsec 加密

本文档提供了在 Kube-OVN Overlay 网络中启用和禁用 IPsec 加密隧道流量的详细指南。由于 OVN 隧道流量通过物理路由器和交换机传输，这些设备可能位于不受信任的公共网络中或面临攻击风险，因此启用 IPsec 加密可以有效防止流量数据被监控或篡改。

目录

术语
注意事项
先决条件
操作步骤
启用 IPsec
禁用 IPsec

术语

术语	解释
IPsec	一种用于保护和验证通过互联网传输的数据的协议和技术。它在 IP 层提供安全通信，主要用于创建虚拟私人网络（VPN）以及保护 IP 数据包的传输。IPsec 主要通过以下方法确保数据安全： • 数据加密：通过加密技术，IPsec 可以确保数据在传输过程中不被窃取或篡改。常见的加密算法包括 AES、3DES 等。

术语	解释
	• 数据完整性检查：IPsec 使用哈希函数（例如 SHA-1、SHA-256）验证数据的完整性，确保数据在传输过程中未被修改。 • 身份验证：IPsec 可以使用多种方法（例如预共享密钥、数字证书）验证通信双方的身份，以防止未经授权的访问。 • 密钥管理：IPsec 使用互联网密钥交换（IKE）协议进行加密密钥的协商与管理。

注意事项

- 启用 IPsec 可能会导致网络中断几秒钟。
- 如果内核版本为 3.10.0-1160.el7.x86_64，启用 Kube-OVN 的 IPsec 功能可能会遇到兼容性问题。

先决条件

请执行以下命令检查当前操作系统内核是否支持 IPsec 相关模块。如果输出显示所有与 XFRM 相关的模块为 `y` 或 `m`，则表示支持 IPsec。

```
cat /boot/config-$(uname -r) | grep CONFIG_XFRM
```

输出：

```
CONFIG_XFRM_ALGO=y
CONFIG_XFRM_USER=y
CONFIG_XFRM_SUB_POLICY=y
CONFIG_XFRM_MIGRATE=y
CONFIG_XFRM_STATISTICS=y
CONFIG_XFRM_IPCOMP=m
```

操作步骤

注意：除非另有说明，以下命令必须在集群主节点的 CLI 工具中执行。

启用 IPsec

1. 修改 kube-ovn-controller 的配置文件。
    1. 执行以下命令编辑 kube-ovn-controller 的 YAML 配置文件。

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

    2. 按照以下说明修改指定字段。

        ```
        spec:
          template:
            spec:
              containers:
                - args:
                    - --enable-ovn-ipsec=true  # 添加此字段
                  securityContext:
                    runAsUser: 0  # 将值更改为 0
        ```

        字段说明：

        - **spec.template.spec.containers[0].args**：在此字段下添加 `- --enable-ovn-ipsec=true`。
        - **spec.template.spec.containers[0].securityContext.runAsUser**：将此字段的值更改为 0。

    3. 保存更改。
2. 修改 kube-ovn-cni 的配置文件。
    1. 执行以下命令编辑 kube-ovn-cni 的 YAML 配置文件。

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

2. 按照以下说明修改指定字段。

```
spec:
  template:
    spec:
      containers:
        - args:
            - --enable-ovn-ipsec=true  # 添加此字段
          volumeMounts:  # 添加挂载路径，将名为 ovs-ipsec-keys 的卷挂载到容器
            - mountPath: /etc/ovs_ipsec_keys
              name: ovs-ipsec-keys
      volumes:  # 添加类型为 hostPath 的名为 ovs-ipsec-keys 的卷
        - name: ovs-ipsec-keys
          hostPath:
            path: /etc/origin/ovs_ipsec_keys
```

字段说明：

- **spec.template.spec.containers[0].args**：在此字段下添加 `- --enable-ovn-ipsec=true`。
- **spec.template.spec.containers[0].volumeMounts**：添加挂载路径，并将名为 ovs-ipsec-keys 的卷挂载到容器。
- **spec.template.spec.volumes**：在此字段下添加类型为 hostPath 的名为 ovs-ipsec-keys 的卷。

3. 保存更改。

3. 验证功能是否成功启用。

1. 执行以下命令进入 kube-ovn-cni Pod。

```
kubectl exec -it -n kube-system $(kubectl get pods -n kube-system -l app=kube-ovn-cni -o=jsonpath='{.items[0].metadata.name}') -- /bin/bash
```

2. 执行以下命令检查安全关联连接的数量。如果有（节点数量 - 1）个连接处于活动状态，表示启用成功。

```
ipsec status | grep "Security"
```

输出：

```
Security Associations (2 up, 0 connecting):  # 由于该集群中有 3 个节点，可以看到连接数量为 2 个活跃
```

禁用 IPsec

1. 修改 kube-ovn-controller 的配置文件。
    1. 执行以下命令编辑 kube-ovn-controller 的 YAML 配置文件。

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

    2. 按照以下说明修改指定字段。

        ```yaml
        spec:
          template:
            spec:
              containers:
                - args:
                    - --enable-ovn-ipsec=false  # 更改为 false
                  securityContext:
                    runAsUser: 65534  # 将值更改为 65534
        ```

        字段说明：

        - **spec.template.spec.containers[0].args**：将此字段 `enable-ovn-ipsec` 的值更改为 false。
        - **spec.template.spec.containers[0].securityContext.runAsUser**：将此字段的值更改为 65534。

    3. 保存更改。
2. 修改 kube-ovn-cni 的配置文件。

1. 执行以下命令编辑 kube-ovn-cni 的 YAML 配置文件。

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

2. 按照以下说明修改指定字段。

- 修改前的配置

```
spec:
  template:
    spec:
      containers:
        - args:
            - --enable-ovn-ipsec=true # 更改为 false
          volumeMounts: # 移除名为 ovs-ipsec-keys 的挂载路径
            - mountPath: /etc/ovs_ipsec_keys
              name: ovs-ipsec-keys
      volumes: # 移除名为 ovs-ipsec-keys 的卷, 类型为 hostPath
        - name: ovs-ipsec-keys
          hostPath:
            path: /etc/origin/ovs_ipsec_keys
```

字段说明：

- **spec.template.spec.containers[0].args**：将此字段 `enable-ovn-ipsec` 的值更改为 false。
- **spec.template.spec.containers[0].volumeMounts**：移除此字段下名为 ovs-ipsec-keys 的挂载路径。
- **spec.template.spec.volumes**：移除此字段下名为 ovs-ipsec-keys 的卷，类型为 hostPath。

- 修改后的配置

```
spec:
  template:
    spec:
      containers:
        - args:
            - --enable-ovn-ipsec=false
          volumeMounts:
      volumes:
```

3. 保存更改。

3. 验证功能是否成功禁用。

   1. 执行以下命令进入 kube-ovn-cni Pod。

      ```
      kubectl exec -it -n kube-system $(kubectl get pods -n kube-system -l app=kube-ovn-cni -o=jsonpath='{.items[0].metadata.name}') -- /bin/bash
      ```

   2. 执行以下命令检查连接状态。如果没有输出，表示禁用成功。

      ```
      ipsec status
      ```

Menu

故障排除

如何解决 **ARM** 环境中的节点间通信问题？

查找错误原因

Menu

如何解决 **ARM** 环境中的节点间通信问题？

在使用较低版本的内核和某些国内网卡时，可能会出现网络卡在启用校验和卸载后计算校验和不正确的情况。这可能导致 Kube-OVN Overlay 网络中节点间的通信失败。具体解决方案如下：

- 解决方案 **1**：升级内核版本。建议将内核版本升级到 4.19.90-25.16.v2101 或更高版本。
- 解决方案 **2**：禁用校验和卸载。如果无法立即升级内核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校验和卸载。

```
ethtool -K eth0 tx off
```

Menu

查找错误原因

错误请求的响应头中的 `X-ALB-ERR-REASON` 字段将指示错误的原因。

错误原因可能是：

```
InvalidBalancer : 未找到xx的负载均衡器 # 表示未找到服务的端点
BackendError : 从后端读取xxx字节数据 # 表示后端确实返回了响应，错误代码不是由alb引起的。
InvalidUpstream : 没有规则匹配 # 表示请求不匹配任何规则，因此alb返回404。
```