网络

介绍

介绍

优势

应用场景

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

了解 ALB

核心组件

快速开始

ALB 常见概念

ALB、ALB 实例、Frontend/FT、Rule、In

ALB Leader

其他资源：

了解 MetalL

术语

MetalLB 的高可

MetalLB 选择 V

外部地址池与节

其他资源

核心概念

认证

问题描述

意译结果

基本概念

快速入门

相关 Ingress 注释

转发认证

基本认证

CR

ALB 特殊的 Ingress 注释

Ingress-Nginx 认证相关的其他功能

注意：与 Ingress-Nginx 不兼容的部分

故障排除

Ingress-nginx 注解兼容性

基本概念

支持的 ingress-nginx 注解

TCP/HTTP

基本概念

CRD

ModSecurity

术语

操作步骤

相关说明

配置示例

不同 Ingress

针对 L4（TCP/

针对 L7（HTTP

L4/L7 超时

基本概念

CRD

超时的含义

HTTP 重定向

GatewayAPI

OTel

术语

先决条件

步骤

相关操作

附加说明

配置示例

功能指南

创建服务

创建 Ingress

配置网关

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 类型的 Service
LoadBalancer 类型 Service 注解

实现方式
前提条件
Ingress 示例：
使用 Web 控制台创建 Ingress
使用 CLI 创建 Ingress

术语
先决条件
示例网关和 Alb
通过 Web 控制
通过 CLI 创建网
查看平台创建的
更新网关

创建域名

示例域名自定义资源 (CR)
通过 Web 控制台创建域名
通过 CLI 创建域名

创建证书

通过 Web 控制台创建证书

配置子网

IP 分配规则
Calico 网络
Kube-OVN 网络
子网管理

通过 CLI 创建外部 IP 地址池

配置网络策略

通过 Web 控制
通过 CLI 创建 N
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 Bas
其他资源

配置集群网络

创建 BGP 对等体

术语
先决条件
示例 BGPPeer 自定义资源 (CR)
通过 Web 控制台创建 BGPPeer

通过 CLI 创建 BGPPeer

如何操作

为 ALB 部署高可用 VIP

方法 1：使用 LoadBalancer 类型的内部路
方法 2：使用外部负载均衡设备提供 VIP

软件数据中心负载均衡方案（Alp

先决条件
操作步骤
验证

准备 Kube-O

使用说明
术语解释
环境要求
配置示例

Underlay 和 Overlay 子网的自动

操作步骤

使用 OAuth Proxy 配合 ALB

Overview
Procedure
Result

创建 Gatewa

部署 MetalLB
设置 Pod 安全策

配置负载均衡器

前提条件
ALB2 自定义资源（CR）示例
通过 Web 控制台创建负载均衡器
通过 CLI 创建负载均衡器
通过 Web 控制台更新负载均衡器
通过 Web 控制台删除负载均衡器
通过 CLI 删除负载均衡器
配置监听端口（Frontend）
前提条件
Frontend 自定义资源（CR）示例
通过 Web 控制台创建监听端口（Frontend
通过 CLI 创建监听端口（Frontend）
后续操作
相关操作
Rule 自定义资源（CR）示例

将 IPv6 流量转发到集群内的 IPv4 地址

配置方法
结果验证

特殊场景部署建
负载均衡器使用

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

Kube-OVN

术语
注意事项
先决条件

通过 Web 控制台创建规则
通过 CLI 创建规则
日志与监控
查看日志
监控指标
其他资源

结果验证

ALB 监控

术语
操作步骤
监控指标

操作步骤

故障排除

如何解决 **ARM** 环境中的节点间通

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，确保集群内无缝的东西向通信以及高效的南北向流量管理，同时提供基本的网络功能。它由以下核心组件组成：

- 为集群内的东西向流量管理提供的容器网络接口 (CNIs)。
- 管理 HTTPS 入口流量的 Ingress Gateway 控制器 ALB。
- 处理 LoadBalancer 类型服务的 MetalLB。
- 此外，它提供强大的网络安全和加密功能，以确保安全通信。

目录

优势

容器网络提供以下核心优势：

- 灵活的网络管理

  支持多种 CNI 的容器网络支持覆盖、底层和路由模式，为适应不同的网络环境提供灵活性。它还提供精细化的 IP 分配和强大的出口管理。作为 Kube-OVN 的创始团队，我们在构建和维护大规模网络方面拥有丰富的实践经验，确保可靠和高性能的连接。

- 隔离、多租户和入口网关的 **API** 灵活性

  通过 ALB 操作员，可以在一个集群中创建和管理多个 ALB 实例。每个租户可以拥有一组专用的 ALB 实例作为入口网关，确保有效的隔离和资源管理。此外，用户可以根据其偏好和运营需求灵活选择 Ingress 和 Gateway API，确保无缝流量管理和增强的灵活性。作为 ALB 的创始团队，我们可以保证解决方案的强大和可扩展性。

- 全面的网络安全

  容器网络提供多层次的安全框架，以确保各个层面的保护。在 CNI 层面，我们支持多种安全策略模型，包括 NetworkPolicy 和 AdminNetworkPolicy，以强制实施细粒度的网络访问控制。为了安全的数据传输，网络融入了强大的流量加密技术。在 Ingress Gateway 层面，我们提供高级安全机制，例如 TLS 终端和对 ModSecurity 的支持，为面向外部的应用提供全面保护。借助内置的网络策略强制执行、加密和流量监控，确保防止未授权访问并符合安全标准。

应用场景

容器网络特别适用于以下场景：

- 东西向流量管理

  利用 CNIs 在集群内提供高效的 Pod 之间通信，同时支持覆盖和底层网络模式，以满足不同的部署需求。

- 南北向流量控制

  使用 ALB 作为入口网关控制器来管理外部 HTTPS 流量，提供灵活的 API 选择和对不同团队的多租户隔离能力。

- 负载均衡服务的暴露

  利用 MetalLB 为 LoadBalancer 类型服务提供高可用性，允许通过虚拟 IP 地址可靠地访问集群服务。

- 网络安全和加密

  通过 NetworkPolicy、AdminNetworkPolicy 和流量加密实现全面的安全，以确保网络基础设施中安全通信。

使用限制

尽管容器网络提供广泛的功能，但以下限制应注意：

- 底层网络要求

  一些底层网络功能，如 Kube-OVN 底层子网、出口 IP 和 MetalLB，需要底层 L2 网络支持。这些功能在公共云提供商和某些虚拟化环境（如 AWS 和 GCP）中无法使用。

凭借其多样化的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、高性能的容器化应用。

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

了解 ALB

核心组件

快速开始

ALB 常见概念

ALB、ALB 实例、Frontend/FT、Rule、In

ALB Leader

其他资源：

了解 MetalL

术语

MetalLB 的高可

MetalLB 选择 V

外部地址池与节

其他资源

理解 Kube-OVN

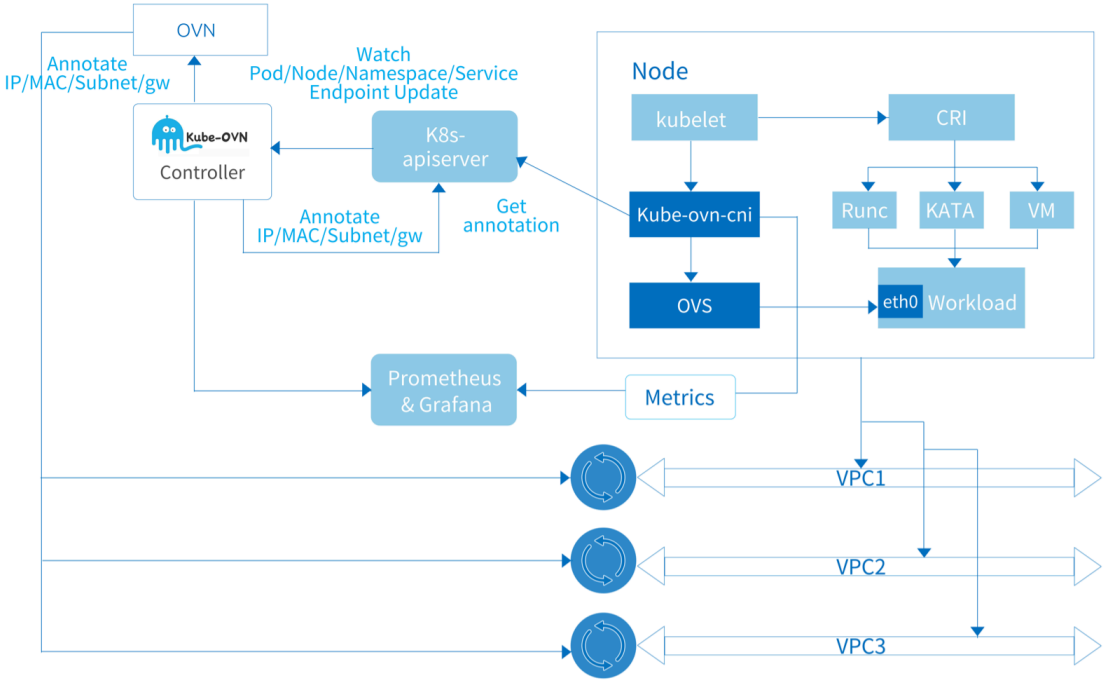
本文档描述了 Kube-OVN 的整体架构、各组件的功能以及它们之间的交互关系。

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

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

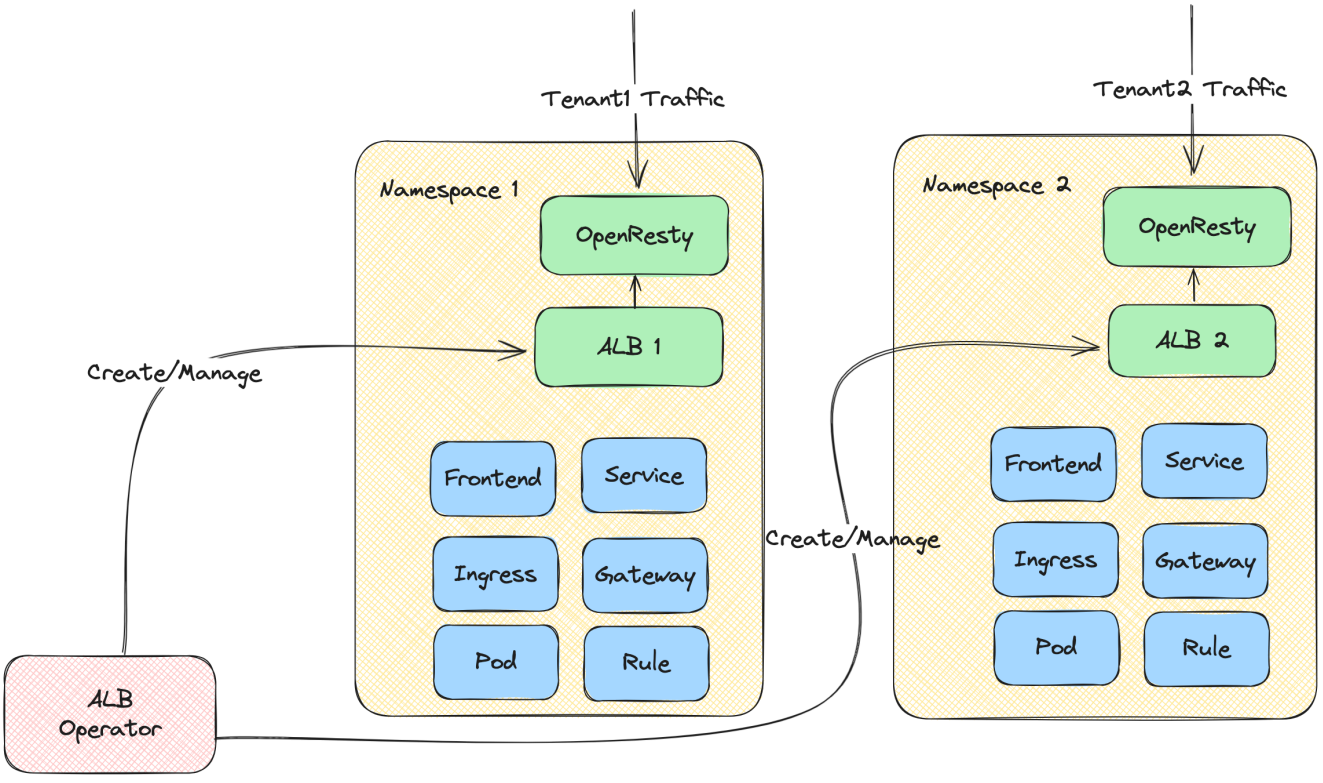
此组件是一个 kubectl 插件，可以快速执行常见操作。

了解 ALB

ALB（Another Load Balancer）是一个基于 OpenResty 的 Kubernetes Gateway，拥有 Alauda 多年生产环境经验的支持。

目录

核心组件



- **ALB Operator**：管理 ALB 实例生命周期的 operator，负责监听 ALB CR 并为不同租户创建和更新 ALB 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），并将其转换为 ALB 特定的 DSL 规则。OpenResty 使用这些 DSL 规则匹配并处理传入请求。

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"  # ALB 部署所在节点的 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 常见概念

以下定义了 ALB 中的常见概念。

Auth

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

了解更多关于 ALB Auth。

网络模式

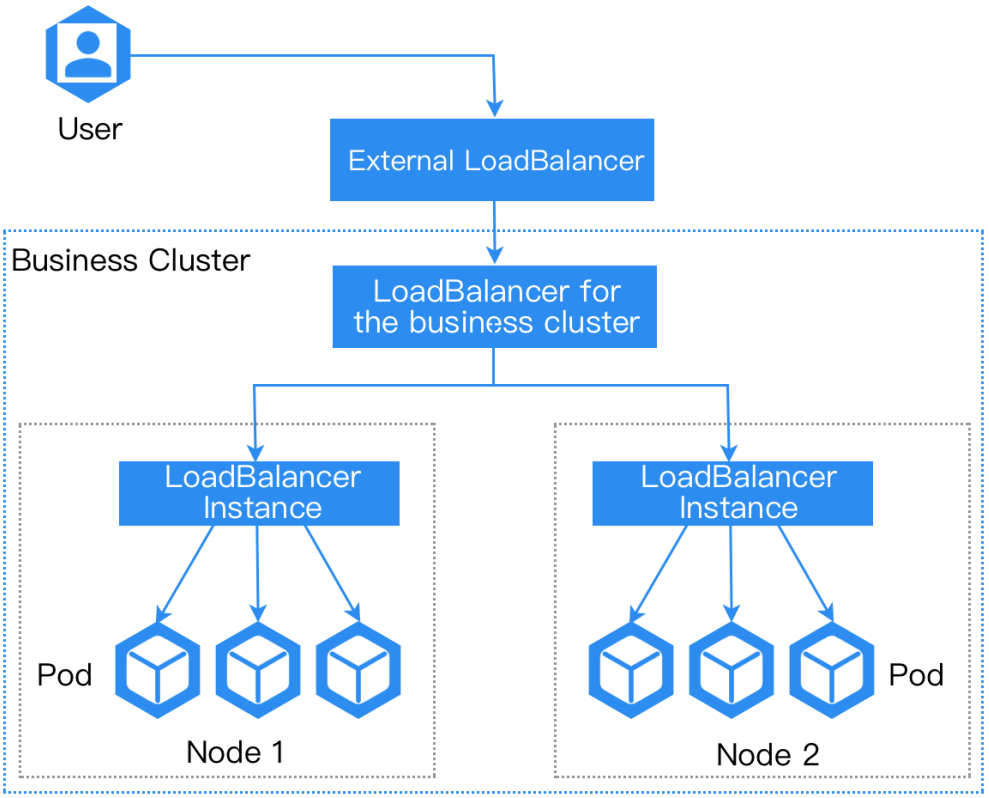
ALB 实例可以部署在两种模式下：host 网络模式和容器网络模式。

Host 网络模式

直接使用节点的网络栈，与节点共享 IP 地址和端口。

在此模式下，负载均衡实例直接绑定节点端口，无需端口映射或类似的容器网络封装转换。

> **NOTE**
>
> 为避免端口冲突，单个节点上只允许部署一个 ALB 实例。



在 host-network 模式下，ALB 实例默认监听节点的所有网卡。

优点：

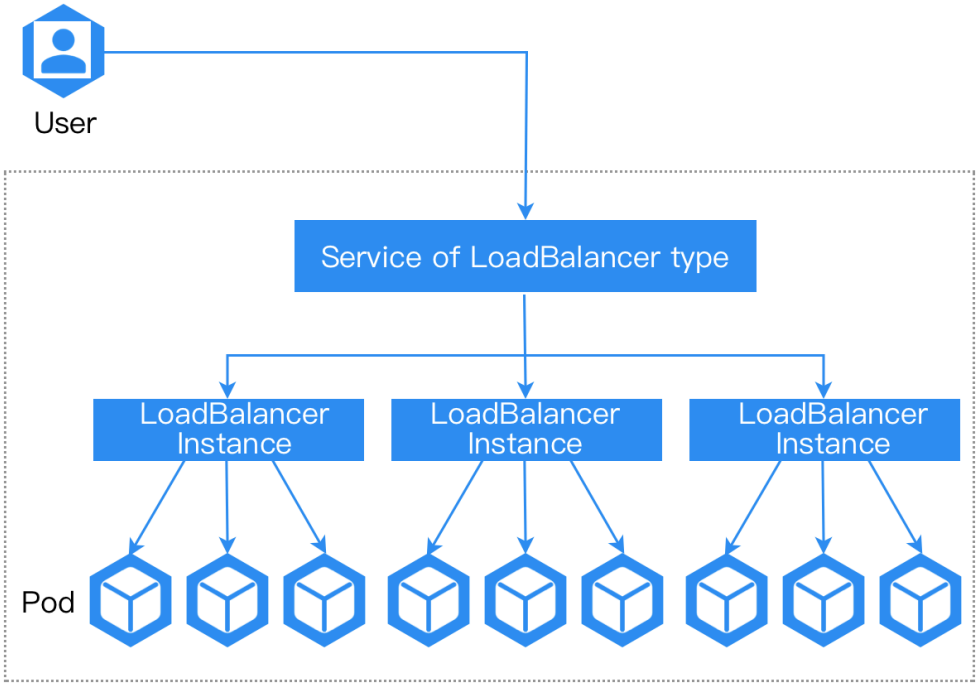
1. 最佳网络性能。
2. 可通过节点 IP 访问。

缺点：

1. 单个节点只允许部署一个 ALB 实例。
2. 端口可能与其他进程冲突。

容器网络模式

与 host 网络模式不同，容器网络模式通过容器网络部署 ALB。



优点：

1. 支持单节点部署多个 ALB 实例。
2. ALB 集成 MetalLB，可为 ALB 提供 VIP。
3. 端口不会与其他进程冲突。

缺点：

1. 性能略低。
2. 必须通过 LoadBalancer 服务访问 ALB。

Frontend

定义了一个名为 frontend（简称 ft）的资源，用于声明所有 ALB 应监听的端口。

每个 frontend 对应负载均衡器（LB）上的一个监听端口。Frontend 通过标签与 ALB 关联。

```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:
  backendProtocol: "http"
  certificate_name: "" (3)
  port: 80
  protocol: http (4)
  serviceGroup: (5)
    services:
      - name: hello-world
        namespace: default
        port: 80
        weight: 100 (6)
```

1. 必填，指明该 Frontend 所属的 ALB 实例。
2. 格式为 `$alb_name-$port`。
3. 格式为 `$secret_ns/$secret_name`。
4. Frontend 自身的协议。
   - `http|https|grpc|grpcs` 用于 L7 代理。
   - `tcp|udp` 用于 L4 代理。
5. 对于 L4 代理，`serviceGroup` 是必填；对于 L7 代理，`serviceGroup` 是可选的。当请求到达时，ALB 会先尝试匹配与该 Frontend 关联的规则，只有当请求不匹配任何规则时，才会将请求转发到 Frontend 配置中指定的默认 `serviceGroup`。
6. `weight` 配置适用于轮询和加权轮询调度算法。

> **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。

其他资源

- L4/L7 超时
- Keepalive

Rules

定义了一个名为 rule 的资源，用于描述 ALB 实例如何处理七层请求。

Rule 可配置复杂的流量匹配和分发模式。流量到达时，根据内部规则进行匹配并执行相应的转发，同时提供如 cors、url 重写等附加功能。

```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: kube-system
spec:
  backendProtocol: "" (3)
  certificate_name: "" (4)
  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"
enableCORS: false
priority: 4 5
serviceGroup: 6
  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. 同 Frontend。
4. 同 Frontend。
5. 数值越小优先级越高。
6. 同 Frontend。

dslx

dslx 是一种领域特定语言，用于描述匹配条件。

例如，下面的规则匹配满足以下所有条件的请求：

- URL 以 /app-a 或 /app-b 开头
- 请求方法为 POST
- URL 参数 group 的值为 vip
- Host 以 .app.com 结尾
- Header 中 LOCATION 的值为 east-1 或 east-2
- 存在名为 uid 的 cookie
- 源 IP 在 1.1.1.1 到 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"
```

项目隔离

对于 rule，默认启用项目隔离，每个用户只能看到自己项目的规则。

项目模式

一个 ALB 可以被多个项目共享，这些项目可以共同管理该 ALB。ALB 的所有端口对这些项目均可见。

端口项目模式

ALB 的某个端口可以属于不同项目，该部署模式称为端口项目模式。管理员需要指定每个项目可使用的端口段，项目用户只能在该端口段内创建端口，并且只能看到该端口段内的端口。

ALB、ALB 实例、Frontend/FT、Rule、Ingress 和项目之间的关系

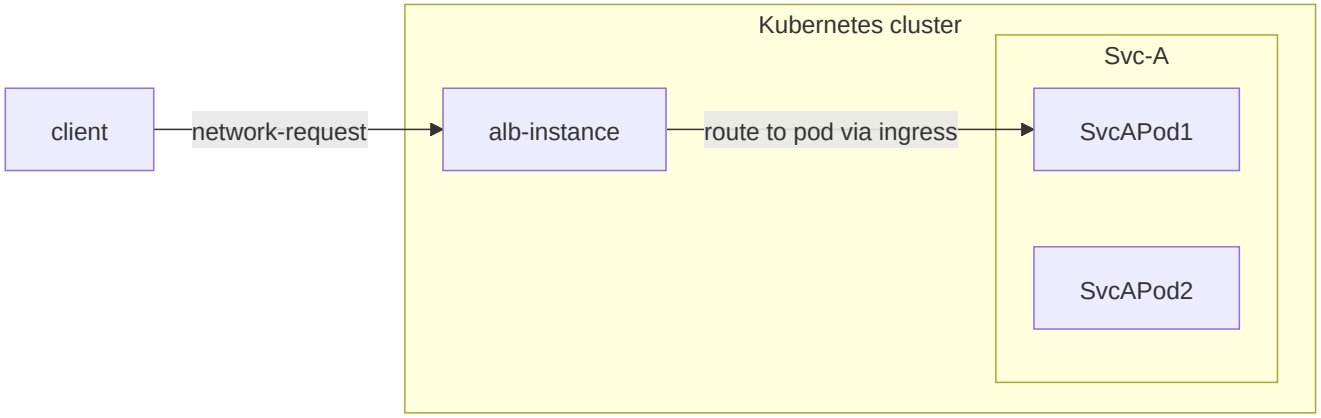
LoadBalancer 是现代云原生架构中的关键组件，作为智能流量路由和负载均衡器。

要理解 ALB 在 Kubernetes 集群中的工作原理，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。

下面介绍这些概念如何协同工作，以及它们在请求调用链中的角色。每个概念的详细介绍将在其他文章中展开。



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. ALB 实例查找匹配该请求的规则。
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，用于描述请求应转发到哪个服务。

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，可以通过 `.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 组成）。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，创建 FT 443（HTTPS）。
2. 如果 Ingress 无证书，创建 FT 80（HTTP）。

RULE

RULE 是 ALB 自定义的资源，作用类似于 Ingress，但更具体。

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. 同步 Ingress 到 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。为 Ingress 中的每个路径创建 Rule。
2. 创建 Ingress 需要的 FT。例如，Ingress 有证书时创建 FT 443（HTTPS），无证书时创建 FT 80（HTTP）。

项目

从 ALB 角度看，项目是一组命名空间。

你可以在 ALB 中配置一个或多个项目。

当 ALB Leader 将 Ingress 转换为 Rules 时，会忽略不属于项目的命名空间中的 Ingress。

其他资源：

- 配置 Load Balancer

了解 MetalLB

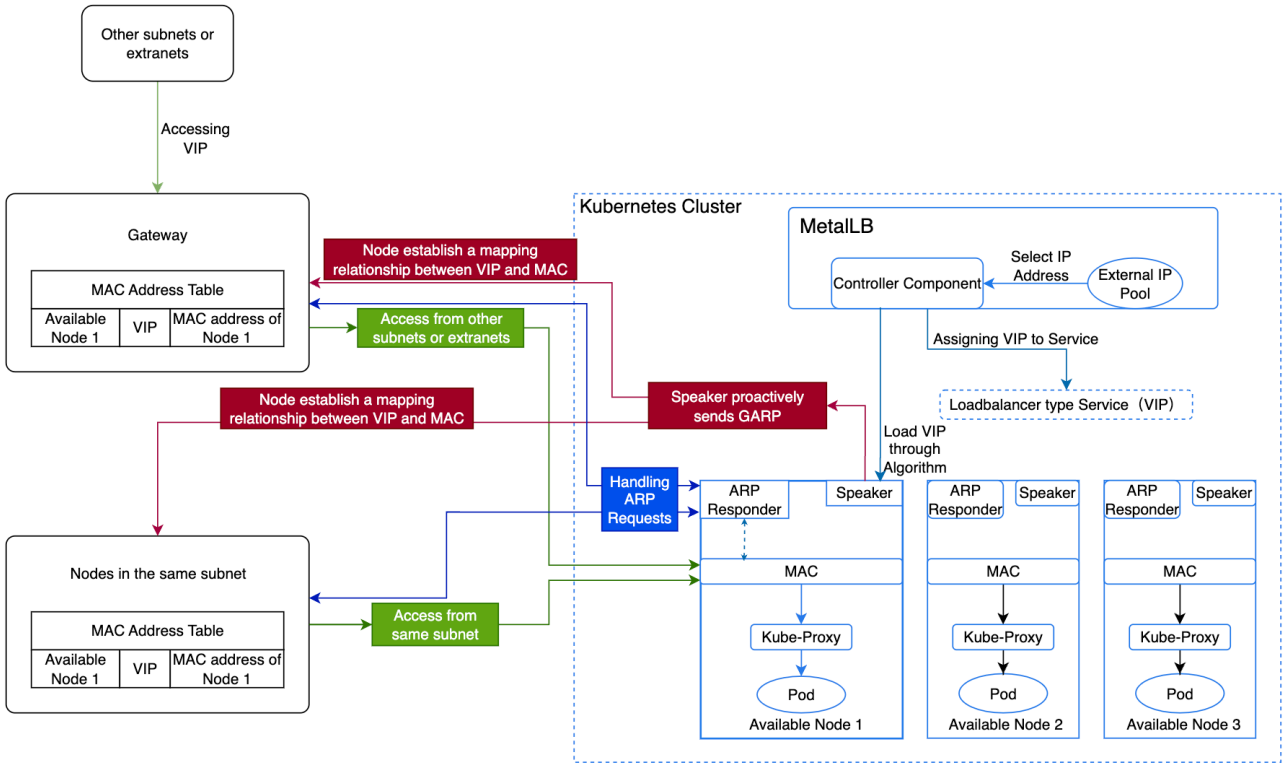
目录

术语

术语	说明
VIP	虚拟 IP 地址（VIP）是 MetalLB 为 LoadBalancer 类型内部路由分配的 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，而是主动将信息发送到网络中。

术语	说明
ARP Responder	MetalLB 的一个组件，负责通过将 VIP 映射到节点的 MAC 地址来响应 ARP 请求。当节点需要与 VIP 通信时，会发送 ARP 请求以获取对应的 MAC 地址。每个可用节点都有一个 ARP Responder，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。它确保节点间达到一定的负载均衡，当某个节点不可用时，其他节点可以接管 VIP 并发送 GARP，从而实现高可用。

MetalLB 的高可用原理



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本子网的网关，再由网关转发到承载 VIP 的节点。
- 当该节点发生故障时，MetalLB 会选择另一个可用节点承载 VIP，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。

MetalLB 选择 VIP 承载节点的算法

MetalLB 对所有对应外部地址池的可用节点与 VIP 进行哈希，并根据特定算法排序，选择第一个可用节点作为 VIP 的承载节点。

外部地址池与节点数量

创建外部地址池并添加可用节点。所有可用节点保持备份关系，即只有承载 VIP 的节点可以转发流量，需承担该外部地址池中所有 VIP 的全部流量。

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：若使用虚拟机，则虚拟机数量 = 外部地址池数量 * n。此处 n 必须大于 2，最多允许一个节点故障。

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

应用示例

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

分析：

需要两个外部地址池和四个可用节点。

- 每个可用节点最多承载 5 个 VIP，意味着一个外部地址池可容纳 5 个 VIP，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需包含一个承载 VIP 的节点和一个备份节点，因此两个外部地址池各需两个可用节点。

其他资源

- 创建外部 IP 地址池
- 创建 BGP Peers

核心概念

认证

问题描述

意译结果

基本概念

快速入门

相关 Ingress 注释

转发认证

基本认证

CR

ALB 特殊的 Ingress 注释

Ingress-Nginx 认证相关的其他功能

注意：与 Ingress-Nginx 不兼容的部分

故障排除

Ingress-nginx 注解兼容性

基本概念

支持的 ingress-nginx 注解

ModSecurity

术语

操作步骤

相关说明

配置示例

L4/L7 超时

基本概念

CRD

超时的含义

GatewayAPI

TCP/HTTP

基本概念

CRD

不同 Ingress

针对 L4（TCP/

针对 L7（HTTP

HTTP 重定向

OTel

术语

先决条件

步骤

相关操作

附加说明

配置示例

目录

问题描述

1. 不符合中文表达习惯：

- "认证是一种机制，用于在请求到达实际服务之前进行身份验证。" 这句话可以更简洁地表达为“认证是一种在请求到达实际服务之前进行身份验证的机制。”
- "它允许您在 ALB 层统一处理认证，而无需在每个后端服务中实现认证逻辑。" 这句话可以简化为“它允许在 ALB 层统一处理认证，无需在每个后端服务中实现认证逻辑。”

2. 语句不通顺：

- "可以配置认证失败后的重定向行为（适用于转发认证）" 这句话可以更清晰地表达为“可以配置认证失败后的重定向行为，适用于转发认证。”

3. 晦涩难懂：

- "auth-response 和 app-response 都可以设置 cookie。" 这句话可以更明确地说明“auth-response 和 app-response 都可以设置 cookie，默认情况下，只有当 app-

response.success 为真时，auth-response.set-cookie 才会合并至 cli-response.set-cookie。”

意译结果

认证

基本概念

什么是认证

认证是一种在请求到达实际服务之前进行身份验证的机制。它允许在 ALB 层统一处理认证，无需在每个后端服务中实现认证逻辑。

支持的认证方法

ALB 支持两种主要的认证方法：

1. 转发认证（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，例如 OAuth、SSO 等。
- 工作流程：

    1. 用户请求到达 ALB
    2. ALB 将认证信息转发至认证服务
    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. 基本认证（**Basic Authentication**）

- 基于用户名和密码的简单认证机制。
- 适用场景：简单的访问控制、开发环境保护。

- 工作流程：

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

认证配置方法

1. 全局认证

   - 在 ALB 层进行配置，适用于所有服务
   - 在 ALB 或 FT CR 中进行配置

2. 路径级别认证

   - 在特定 Ingress 路径进行配置
   - 在特定规则上进行配置
   - 可以覆盖全局认证配置

3. 禁用认证

   - 针对特定路径禁用认证
   - 通过注释进行配置：`alb.ingress.cpaas.io/auth-enable: "false"`
   - 在规则中使用结合 CR 进行配置

认证结果处理

- 认证成功：请求将被转发至后端服务。
- 认证失败：返回 401 未授权错误。
- 可以配置认证失败后的重定向行为，适用于转发认证。

快速入门

使用 ALB 配置基本认证

部署 ALB

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

配置密钥和 Ingress

```
# echo "Zm9vOiRhcHIxJHFJQ05aNjFRJDJpb29pSlZVQU1tcHJxMjU4L0NoUDE=" | base6
4 -d #  foo:$apr1$qICNZ61Q$2iooiJVUAMmprq258/ChP1
# openssl passwd -apr1 -salt qICNZ61Q bar # $apr1$qICNZ61Q$2iooiJVUAMmprq
258/ChP1

kubectl apply -f - <<'END'
apiVersion: v1
kind: Secret
metadata:
  name: auth-file
type: Opaque
data:
  auth: Zm9vOiRhcHIxJHFJQ05aNjFRJDJpb29pSlZVQU1tcHJxMjU4L0NoUDE=
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: auth-file
  annotations:
    "nginx.ingress.kubernetes.io/auth-type":  "basic"
    "nginx.ingress.kubernetes.io/auth-secret":  "default/auth-file"
    "nginx.ingress.kubernetes.io/auth-secret-type": "auth-file"
spec:
  rules:
  - http:
      paths:
      - path: /app-file
        pathType: Prefix
        backend:
          service:
            name: app-server
            port:
              number: 80
END
```

验证

```
# echo "Zm9vOiJhYXIi" | base64 -d # foo:bar
curl -v -X GET -H "Authorization: Basic Zm9vOmJhcg==" http://$ALB_IP:80/app-file # 应该返回 200
# 错误的密码
curl -v -X GET -H "Authorization: Basic XXXXOmJhcg==" http://$ALB_IP:80/app-file # 应该返回 401
```

相关 **Ingress** 注释

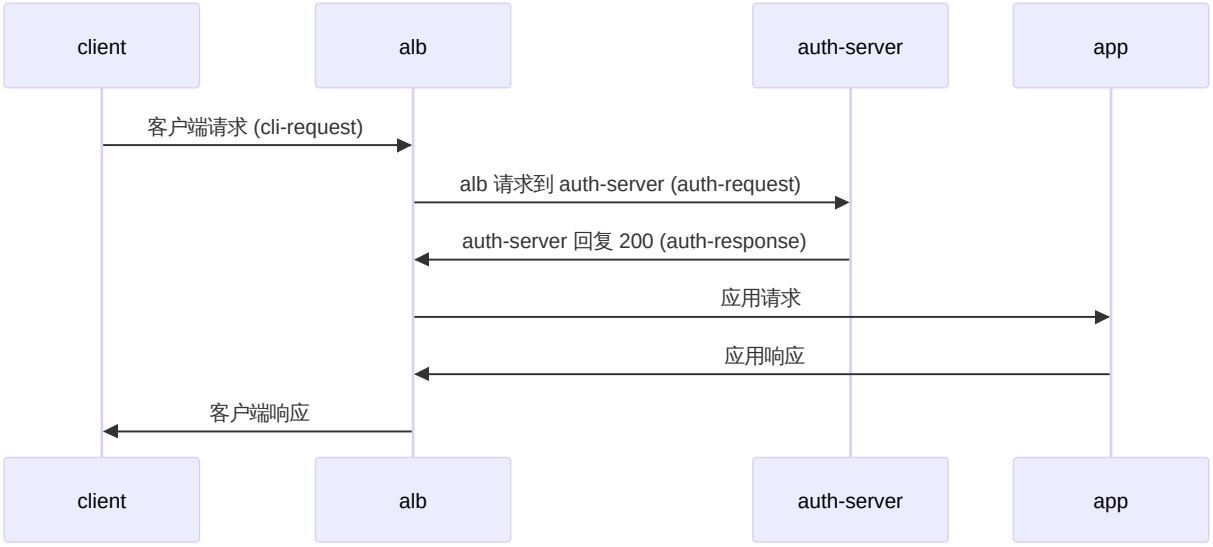
Ingress-nginx 定义了一系列注释以配置认证过程的具体细节。以下是 ALB 支持的注释列表，其中"v"表示支持，"x"表示不支持。

注释	支持	类型	备注
forward-auth			通过发送 http 请求进行转发认证
nginx.ingress.kubernetes.io/auth-url	v	string	
nginx.ingress.kubernetes.io/auth-method	v	string	
nginx.ingress.kubernetes.io/auth-signin	v	string	
nginx.ingress.kubernetes.io/auth-signin-redirect-param	v	string	
nginx.ingress.kubernetes.io/auth-response-headers	v	string	
nginx.ingress.kubernetes.io/auth-proxy-set-headers	v	string	
nginx.ingress.kubernetes.io/auth-request-redirect	v	string	
nginx.ingress.kubernetes.io/auth-always-set-cookie	v	boolean	

注释	支持	类型	备注
nginx.ingress.kubernetes.io/auth-snippet	x	string	
basic-auth			通过用户名和密码的秘密进行认证
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			在发送请求时保持活跃，通过一系列注释指定保持活跃的行为
nginx.ingress.kubernetes.io/auth-keepalive	x	number	
nginx.ingress.kubernetes.io/auth-keepalive-share-vars	x	"true" or "false"	
nginx.ingress.kubernetes.io/auth-keepalive-requests	x	number	
nginx.ingress.kubernetes.io/auth-keepalive-timeout	x	number	

注释	支持	类型	备注
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	

转发认证



相关注释：

- 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

这些注释描述了在上述图中对 auth-request、app-request 和 cli-response 所做的修改。

构建相关注释

auth-url

auth-request 的 URL，值可以是变量。

auth-method

auth-request 的方法。

auth-proxy-set-headers

值为格式为 `ns/name` 的 ConfigMap 引用。 默认情况下，所有来自 cli-request 的头部将发送到 auth-server，可通过 proxy_set_header 配置附加头部。以下头部默认会被发送：

```
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;
```

构建应用请求相关注释

auth-response-headers

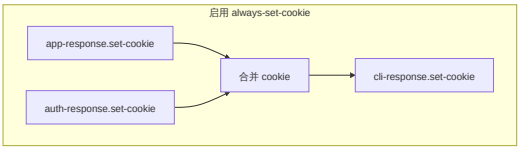
值为以逗号分隔的字符串，允许我们将特定头部从 auth-response 带入 app-request。 示例：

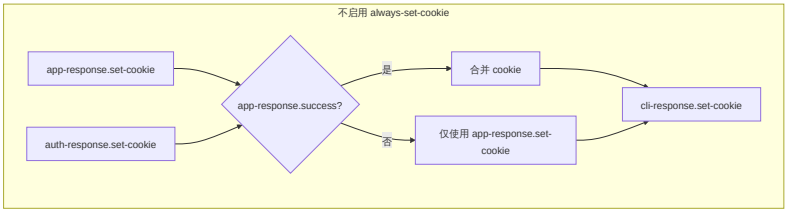
```
nginx.ingress.kubernetes.io/auth-response-headers: Remote-User,Remote-Nam
e
```

当 ALB 发起 app-request 时，Remote-User 和 Remote-Name 会包含在 auth-response 的头部中。

cookie 处理

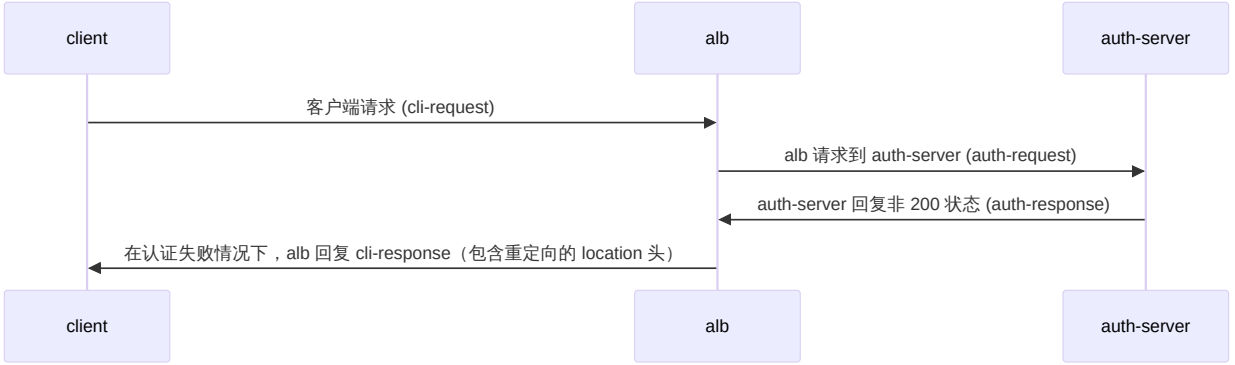
auth-response 和 app-response 都可以设置 cookie。默认情况下，只有当 app-response.success 为真时，auth-response.set-cookie 才会合并至 cli-response.set-cookie。





重定向签到相关配置

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



auth-signin

值是一个 URL，指定 cli-response 中的 location 头。

auth-signin-redirect-param

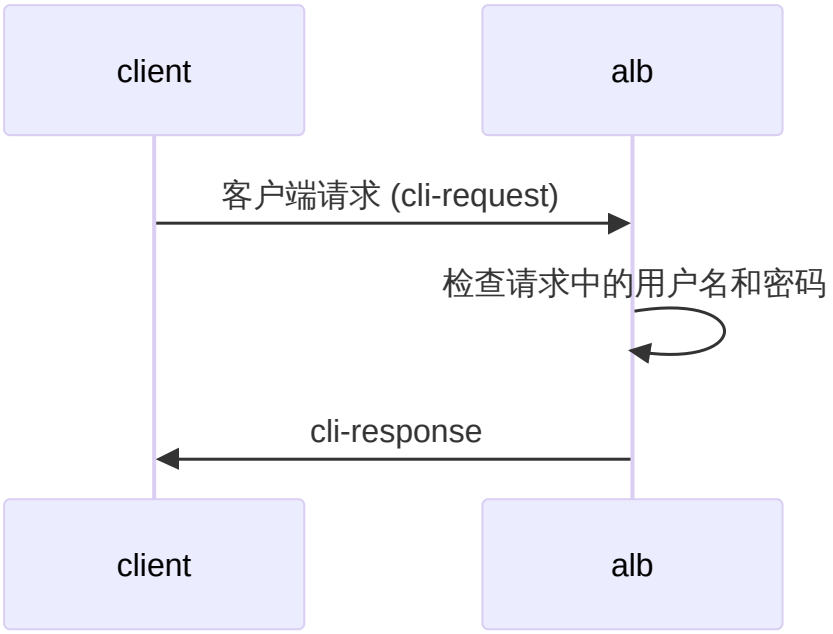
签名 URL 中查询参数的名称，默认为 rd。 如果签名 URL 不包含指定参数名的 `auth-signin-redirect-param`，alb 将自动添加该参数。参数值将设置为 `$pass_access_scheme://$http_host$escaped_request_uri`，用于记录原始请求 URL。

auth-request-redirect

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

基本认证

基本认证是 RFC 7617↗ 中描述的认证过程。 交互过程如下：



auth-realm

受保护区域的描述↗ 即 cli-response 的 `WWW-Authenticate` 头中的 realm 值。 WWW-Authenticate: Basic realm="$realm"

auth-type

认证方案的类型，目前仅支持基本认证。

auth-secret

用户名和密码的秘密引用，格式为 ns/name。

auth-secret-type

密钥支持两种类型：

1. auth-file：密钥的数据仅包含一个键 "auth"，其值为 Apache htpasswd 格式的字符串。例如：

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

2. auth-map：密钥的数据中的每个键代表一个用户名，且相应的值是密码哈希（不含用户名的 htpasswd 格式）。例如：

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

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

CR

ALB CR 已添加与认证相关的配置项，可以在 ALB/Frontend/Rule CR 上配置。 在运行时，ALB 会将 Ingress 上的注释转换为规则。

```yaml
auth:
  # 基本认证配置
  basic:
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-type: basic
    auth_type: "basic"
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-realm
    realm: "受限访问"
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-secret
    secret: "ns/name"
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-secret-type
    secret_type: "auth-map|auth-file"
  # 转发认证配置
  forward:
    #  布尔值；对应 nginx.ingress.kubernetes.io/auth-always-set-cookie
    always_set_cookie: true
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-proxy-set-headers
    auth_headers_cm_ref: "ns/name"
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-request-redirect
    auth_request_redirect: "/login"
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-method
    method: "GET"
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-signin
    signin: "/signin"
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-signin-redirect-param
    signin_redirect_param: "redirect_to"
    #  []字符串；对应 nginx.ingress.kubernetes.io/auth-response-headers
    upstream_headers:
      - "X-User-ID"
      - "X-User-Name"
      - "X-User-Email"
    #  字符串；对应 nginx.ingress.kubernetes.io/auth-url
    url: "http://auth-service/validate"
```

认证可针对以下进行配置：

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

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

ALB 特殊的 Ingress 注释

在处理 Ingress 的过程中，ALB 根据注释的前缀确定优先级。 优先级从高到低为：

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

这可以解决与 Ingress-nginx 的兼容性问题，并在特定的 Ingress 路径上指定认证配置。

auth-enable

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

ALB 添加的新注释，用于指定是否为 Ingress 启用认证功能。

Ingress-Nginx 认证相关的其他功能

global-auth

在 Ingress-nginx 中，您可以通过 ConfigMap 设置全局认证。这相当于为所有 Ingress 配置认证。在 ALB 中，您可以在 ALB2 和 FT CRs 上配置认证。它们下面的规则将继承这些配置。

no-auth-locations

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

注意：与 Ingress-Nginx 不兼容的部分

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

3. 不支持 auth-cache
4. 不支持 auth-tls
5. 基本认证仅支持 basic，不支持 digest
6. 基本认证仅支持 apr1 算法，不支持 bcrypt sha256 等。

故障排除

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

Ingress-nginx 注解兼容性

目录

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。

支持的 ingress-nginx 注解

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

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

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-keepalive-share-vars	"true" or "false"	x
nginx.ingress.kubernetes.io/auth-keepalive-requests	number	x
nginx.ingress.kubernetes.io/auth-keepalive-timeout	number	x
nginx.ingress.kubernetes.io/auth-proxy-set-headers	string	v
nginx.ingress.kubernetes.io/auth-snippet	string	x
nginx.ingress.kubernetes.io/enable-global-auth	"true" or "false"	o auth
nginx.ingress.kubernetes.io/backend-protocol	string	v
nginx.ingress.kubernetes.io/canary	"true" or "false"	x
nginx.ingress.kubernetes.io/canary-by-header	string	x
nginx.ingress.kubernetes.io/canary-by-header-value	string	x
nginx.ingress.kubernetes.io/canary-by-header-pattern	string	x
nginx.ingress.kubernetes.io/canary-by-cookie	string	x
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

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/configuration-snippet	string	x
nginx.ingress.kubernetes.io/custom-http-errors	[]int	x
nginx.ingress.kubernetes.io/custom-headers	string	o
nginx.ingress.kubernetes.io/default-backend	string	o 可使用 ingress 的 default-backend
nginx.ingress.kubernetes.io/enable-cors	"true" or "false"	v
nginx.ingress.kubernetes.io/cors-allow-origin	string	v
nginx.ingress.kubernetes.io/cors-allow-methods	string	v
nginx.ingress.kubernetes.io/cors-allow-headers	string	v
nginx.ingress.kubernetes.io/cors-expose-headers	string	x
nginx.ingress.kubernetes.io/cors-allow-credentials	"true" or "false"	x
nginx.ingress.kubernetes.io/cors-max-age	number	x
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

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/limit-rps	number	x
nginx.ingress.kubernetes.io/global-rate-limit	number	x
nginx.ingress.kubernetes.io/global-rate-limit-window	duration	x
nginx.ingress.kubernetes.io/global-rate-limit-key	string	x
nginx.ingress.kubernetes.io/global-rate-limit-ignored-cidrs	string	x
nginx.ingress.kubernetes.io/permanent-redirect	string	v redirect
nginx.ingress.kubernetes.io/permanent-redirect-code	number	v redirect
nginx.ingress.kubernetes.io/temporal-redirect	string	v redirect
nginx.ingress.kubernetes.io/preserve-trailing-slash	"true" or "false"	x
nginx.ingress.kubernetes.io/proxy-body-size	string	x
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

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-next-upstream-timeout	number	x
nginx.ingress.kubernetes.io/proxy-next-upstream-tries	number	x
nginx.ingress.kubernetes.io/proxy-request-buffering	string	x
nginx.ingress.kubernetes.io/proxy-redirect-from	string	x
nginx.ingress.kubernetes.io/proxy-redirect-to	string	x
nginx.ingress.kubernetes.io/proxy-http-version	"1.0" or "1.1"	x
nginx.ingress.kubernetes.io/proxy-ssl-secret	string	x
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

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/server-alias	string	x
nginx.ingress.kubernetes.io/server-snippet	string	x
nginx.ingress.kubernetes.io/service-upstream	"true" or "false"	x
nginx.ingress.kubernetes.io/session-cookie-change-on-failure	"true" or "false"	x
nginx.ingress.kubernetes.io/session-cookie-conditional-samesite-none	"true" or "false"	x
nginx.ingress.kubernetes.io/session-cookie-domain	string	x
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

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/stream-snippet	string	x
nginx.ingress.kubernetes.io/upstream-hash-by	string	x
nginx.ingress.kubernetes.io/x-forwarded-prefix	string	x
nginx.ingress.kubernetes.io/load-balance	string	x
nginx.ingress.kubernetes.io/upstream-vhost	string	v
nginx.ingress.kubernetes.io/denylist-source-range	CIDR	o 可通过 modsecurity 实现类似效果
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

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-opentelemetry	"true" or "false"	v otel
nginx.ingress.kubernetes.io/opentelemetry-trust-incoming-span	"true" or "false"	v otel
nginx.ingress.kubernetes.io/enable-modsecurity	bool	v modsecurity
nginx.ingress.kubernetes.io/enable-owasp-core-rules	bool	v modsecurity
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

TCP/HTTP 保持连接

目录

基本概念

1. ALB 支持在端口级别进行保持连接配置。该配置可以在前端进行。

2. 保持连接只在 客户端与 **ALB** 之间，而不是 **ALB** 与后端之间 实现。

3. 该功能通过 Nginx 配置实现，并且当配置发生变化时，Nginx 需要并会自动重载。

4. TCP 保持连接和 HTTP 保持连接是两个不同的概念：

    1. **TCP** 保持连接 是 TCP 协议的一个特性，在没有数据传输时定期发送探测数据包，以检查连接是否仍然存活。它有助于检测并清理死连接。
    2. **HTTP** 保持连接（也称为持久连接）允许多个 HTTP 请求复用同一 TCP 连接，从而避免建立新连接的开销。这通过减少延迟和资源使用提高了性能。

CRD

```
keepalive:
  properties:
    http:
      description: 下游 L7 保持连接
      properties:
        header_timeout:
          description: 保持连接头超时。默认未设置。
          type: string
        requests:
          description: 保持连接请求。默认值为 1000。
          type: integer
        timeout:
          description: 保持连接超时。默认值为 75s。
          type: string
      type: object
    tcp:
      description: TCPKeepAlive 定义 TCP 保持连接参数（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
```

只能在 Frontend `spec.config.keepalive` 配置。

ModSecurity

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

目录

术语

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

操作步骤

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

方法一：添加注解

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

- **Ingress-Nginx** 兼容注解

注解	类型	适用对象	说明
nginx.ingress.kubernetes.io/enable-modsecurity	bool	Ingress	启用 ModSecurity。
nginx.ingress.kubernetes.io/enable-owasp-core-rules	bool	Ingress	启用 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，以满足特定的安全需求。

- **ALB** 特殊注解

注解	类型	适用对象	说明
alb.modsecurity.cpaas.io/use-recommend	bool	Ingress	启用或禁用推荐的 ModSecurity 规则；设置为 `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，则会尝试从 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
```

不同 Ingress 方式的比较

Alauda Container Platform 支持 Kubernetes 生态系统中的多种 ingress 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

LoadBalancer 类型的 Service、Gateway API 和 ALB Rules 都可以对外暴露 L4 流量。这里推荐使用 LoadBalancer 类型的 Service 方式。 Gateway API 和 ALB Rules 都是由 ALB 实现的，ALB 是一个用户空间代理，在处理 L4 流量时，其性能相比 LoadBalancer 类型的 Service 会显著下降。

针对 L7（HTTP/HTTPS）流量

Ingress、GatewayAPI 和 ALB Rules 都可以对外暴露 L7 流量，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，推荐作为默认使用方式。 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的隔离 ALB 实例来处理 GatewayAPI 规则。 因此，如果您需要将 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 Rules 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。

HTTP 重定向

目录

基本概念

HTTP 重定向是 ALB 提供的一项功能。它将直接为匹配规则的请求返回一个 30x 的 HTTP 状态码。Location 头将用于指示客户端重定向到新 URL。ALB 支持在端口和规则级别进行重定向配置。

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
```

重定向可以在以下地方进行配置：

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

Ingress 注解

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

注解	描述
nginx.ingress.kubernetes.io/ssl-redirect	对应于 CR 中的方案，默认将方案设置为 HTTPS
nginx.ingress.kubernetes.io/force-ssl-redirect	对应于 CR 中的方案，默认将方案设置为 HTTPS

SSL 重定向

1. ssl-redirect 和 force-ssl-redirect 的不同之处在于，ssl-redirect 仅在 Ingress 拥有相应域的证书时生效，而 force-ssl-redirect 无论是否存在证书均生效。
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

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

超时通过 Lua 脚本实现，并且在更改后 不需要重载 Nginx。

CRD

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

配置可以在以下位置进行设置：

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

超时的含义

超时分为三种类型：

1. **proxy_connect_timeout_ms**: 定义与上游服务器建立连接的超时时间。如果在此时间内无法建立连接，请求将失败。
2. **proxy_read_timeout_ms**: 定义从上游服务器读取响应的超时时间。该超时时间是在两次连续的读取操作之间设定的，而非针对整个响应。如果在此时间内未收到数据，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。与读取超时相似，该超时是在两次连续的写入操作之间设定的。如果在此时间内无法发送数据，连接将被关闭。

Ingress 注解

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

注解	描述
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，其用作 L4 超时。

GatewayAPI

GatewayAPI↗ 是 Kubernetes ingress 的一项新标准。

ALB 也支持 GatewayAPI。每个 Gateway 资源都会被转换为一个 ALB 资源。

Listener 和 Router 将直接在 ALB 中处理，不会被转换为 `Frontend` 和 `Rule`。

OTel

OpenTelemetry（OTel）是一个开源项目，旨在为在分布式系统（如微服务架构）中收集、处理和导出遥测数据提供一个供所有供应商使用的中立标准。它帮助开发者更轻松地分析软件的性能和行为，从而促进应用程序问题的诊断和解决。

目录

术语

术语	说明
Trace	提交到 OTel 服务器的数据，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动；每个 Trace 由多个 Span 组成。
Span	Trace 中的一个独立操作或事件，包括开始时间、持续时间和其他相关信息。
OTel Server	能够接收和存储 Trace 数据的 OTel 服务器，如 Jaeger、Prometheus 等。
Jaeger	一种开源分布式追踪系统，用于监控和排查微服务架构，支持与 OpenTelemetry 的集成。
Attributes	附加到 Trace 或 Span 的键值对，以提供额外的上下文信息。包括资源属性和 Span 属性；有关更多信息，请参见 Attributes。
Sampler	一种策略组件，用于决定是否对 Trace 进行采样和报告。可以配置不同的采样策略，如全量采样、比例采样等。
ALB（另一个负载均衡器）	分配网络请求到集群中可用节点的软件或硬件设备；平台使用的负载均衡器（ALB）是一个第七层软件负载均衡器，能够配置监控流量并使用 OTel。ALB 支持将 Traces 提交到指定的收集器，并允许配置不同的采样策略；它还支持配置是否在 Ingress 层提交 Traces。
FT（前端）	ALB 的端口配置，指定端口级别的配置。
Rule	在端口（FT）上的路由规则，用于匹配特定路由。
HotROD（按需乘车）	Jaeger 提供的示例应用程序，用于演示分布式追踪的使用；有关更多详细信息，请参考 Hot R.O.D. - Rides on Demand ↗。
hotrod-with-proxy	通过环境变量指定 HotROD 内部微服务的地址；有关更多详细信息，请参考 hotrod-with-proxy ↗。

先决条件

- 确保操作性的 **ALB** 存在：创建或使用现有的 ALB，本文件中 ALB 的名称用 `<otel-alb>` 替换。有关创建 ALB 的说明，请参考 Creating Load Balancer。
- 确保存在 **OTel** 数据报告服务器地址：该地址在此后称为 `<jaeger-server>`。

步骤

更新 ALB 配置

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

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

2. 在 `spec.config` 部分下添加以下字段。

```
otel:
  enable: true
  exporter:
    collector:
      address: "<jaeger-server>" # 将 <jaeger-server> 替换为实际的 OTel 数据报告服务器地址
      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. 执行以下命令以保存更新。更新后，ALB 将默认启用 OpenTelemetry，所有请求 Trace 信息将被报告到 Jaeger 服务器。

```
:wq
```

相关操作

在 **Ingress** 中配置 **OTel**

- 启用或禁用 **Ingress** 上的 **OTel**

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

  步骤

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

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

  参数说明：

  - **nginx.ingress.kubernetes.io/enable-opentelemetry**：当设置为 `true` 时，表示 Ingress 控制器在处理请求时启用 OpenTelemetry 功能，这意味着请求 Trace 信息将被收集并报告。当设置为 `false` 或移除此注释时，表示请求 Trace 信息将不被收集或报告。

- 启用或禁用 **Ingress** 上的 **OTel Trust**

  OTel Trust 决定 Ingress 是否信任并使用来自传入请求的 Trace 信息（例如 trace ID）。

  步骤

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

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

参数说明：

- **nginx.ingress.kubernetes.io/opentelemetry-trust-incoming-span**：当设置为 `true` 时，Ingress 将继续使用已经存在的 Trace 信息，帮助保持跨服务追踪的一致性，允许在分布式追踪系统中完整地追踪和分析整个请求链。当设置为 `false` 时，将为请求生成新的追踪信息，这可能导致请求在进入 Ingress 后被视为新的追踪链的一部分，从而中断跨服务的追踪连续性。

- 在 **Ingress** 上添加不同的 **OTel** 配置

  此配置允许您自定义 OTel 的行为和数据导出方法，以便对不同的 Ingress 资源进行细粒度的追踪策略或目标控制。

  步骤

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

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

  参数说明：

  - **exporter**：指定收集的 Trace 数据如何发送到 OTel 收集器（OTel 数据报告服务器）。
  - **address**：指定 OTel 收集器的地址。
  - **request_timeout**：指定请求超时时间。

在应用程序中使用 **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程序中启用和使用 OTel 功能。

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

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

输出结果：

```
{
    "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	描述
otel.enable	是否启用 OTel 功能。
exporter.collector.address	OTel 数据报告服务器的地址，支持 http/https 协议和域名。
flags.hide_upstream_attrs	是否报告关于上游规则的信息。
flag.notrust_incoming_span	是否信任并使用来自传入请求的 OTel Trace 信息（例如 trace ID）。
flags.report_http_request_header	是否报告请求头。
flags.report_http_response_header	是否报告响应头。
sampler.name	采样策略名称；有关详细信息，请参见 Sampling Strategies。
sampler.options.fraction	采样率。
sampler.options.parent_name	父级基于采样策略的父策略。

继承

默认情况下，如果 ALB 配置了某些 OTel 参数且 FT 未配置，则 FT 将从 ALB 继承参数作为其配置；即 FT 继承 ALB 的配置，Rules 可以从 ALB 和 FT 两者继承配置。

- **ALB**：ALB 上的配置通常是全局的和默认的。在此可以配置例如 Collector 地址等全局参数，这些参数将被下级 FT 和 Rules 继承。
- **FT**：FT 可以从 ALB 继承配置，这意味着未在 FT 中配置的某些 OTel 参数将使用来自 ALB 的配置。然而，FT 也可以进一步细化；例如，您可以选择在 FT 上选择性地启用或禁用 OTel，而不影响其他 FT 或 ALB 的全局设置。
- **Rule**：Rule 可以从 ALB 和 FT 两者继承配置。然而，Rule 也可以进一步细化；例如，特定的 Rule 可以选择不信任传入的 OTel Trace 信息或调整采样策略。

步骤

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

附加说明

采样策略

参数	说明
always on	始终报告所有追踪数据。
always off	从不报告追踪数据。
traceid-ratio	根据 `traceid` 决定是否报告。`traceparent` 的格式为 `xx-traceid-xx-flag`，其中 `traceid` 的前 16 个字符代表一个 32 位的十六进制整数。如果此整数小于 `fraction` 乘以 4294967295（即 (2^32-1)），则将被报告。
parent-base	根据请求中 `traceparent` 的标志部分决定是否报告。当标志为 01 时，将被报告；例如：`curl -v "http://$ALB_IP/" -H 'traceparent: 00-xx-xx-01'`；当标志为 02 时，将不会被报告；例如：`curl -v "http://$ALB_IP/" -H 'traceparent: 00-xx-xx-02'`。

属性

- 资源属性

  默认情况下报告这些属性。

参数	说明
hostname	ALB Pod 的主机名
service.name	ALB 的名称

参数	说明
service.namespace	ALB 所在的命名空间
service.type	默认是 ALB
service.instance.id	ALB Pod 的名称

- **Span** 属性
  - 默认情况下报告的属性：

参数	说明
http.status_code	状态码
http.request.resend_count	重试计数
alb.rule.rule_name	此请求匹配的规则名称
alb.rule.source_type	此请求匹配的规则类型，目前仅 Ingress
alb.rule.source_name	Ingress 的名称
alb.rule.source_ns	Ingress 所在的命名空间

  - 默认情况下报告但可以通过修改 flag.hide_upstream_attrs 字段排除的属性：

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

  - 默认情况下未报告但可以通过修改 flag.report_http_request_header 字段来报告的属性：

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

- 默认情况下未报告但可以通过修改 flag.report_http_response_header 字段来报告的属性：

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 并使用 Jaeger 作为 OTel 服务器，并以 Hotrod-proxy 作为演示后端。通过配置 Ingress 规则，当客户端请求 ALB 时，流量将转发到 HotROD。此外，HotROD 内部微服务之间的通信也通过 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以部署 Jaeger、ALB、HotROD 及所有必要的 CR 进行测试。

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

3. 执行以下命令以获取 Jaeger 的访问地址。

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

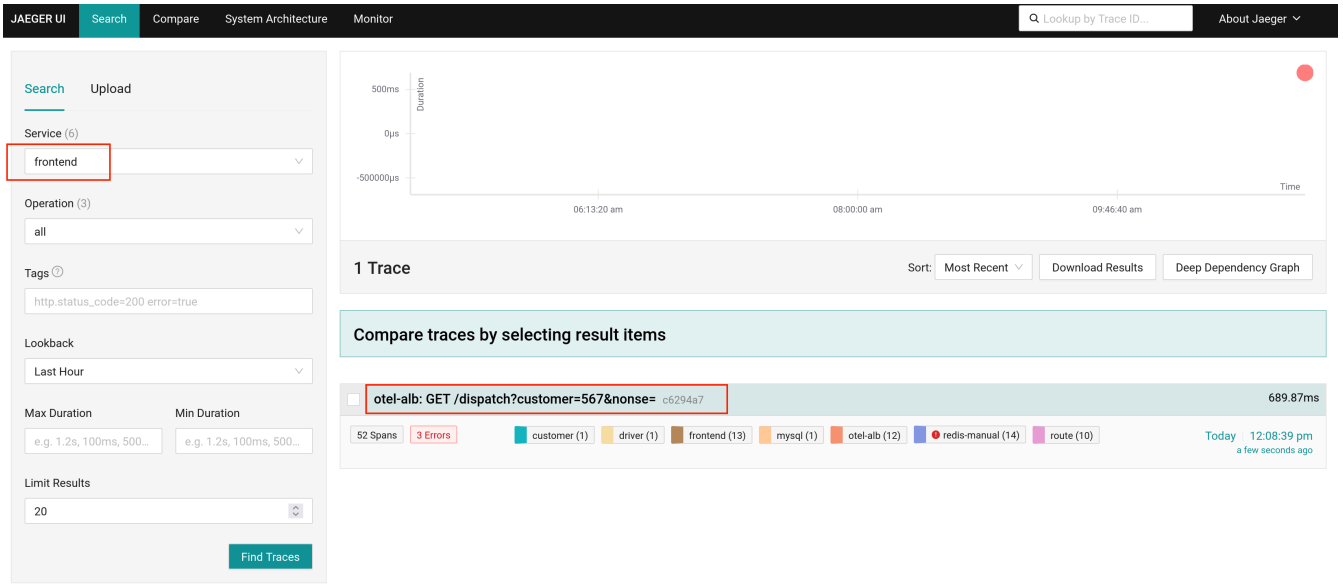
4. 执行以下命令以获取 otel-alb 的访问地址。

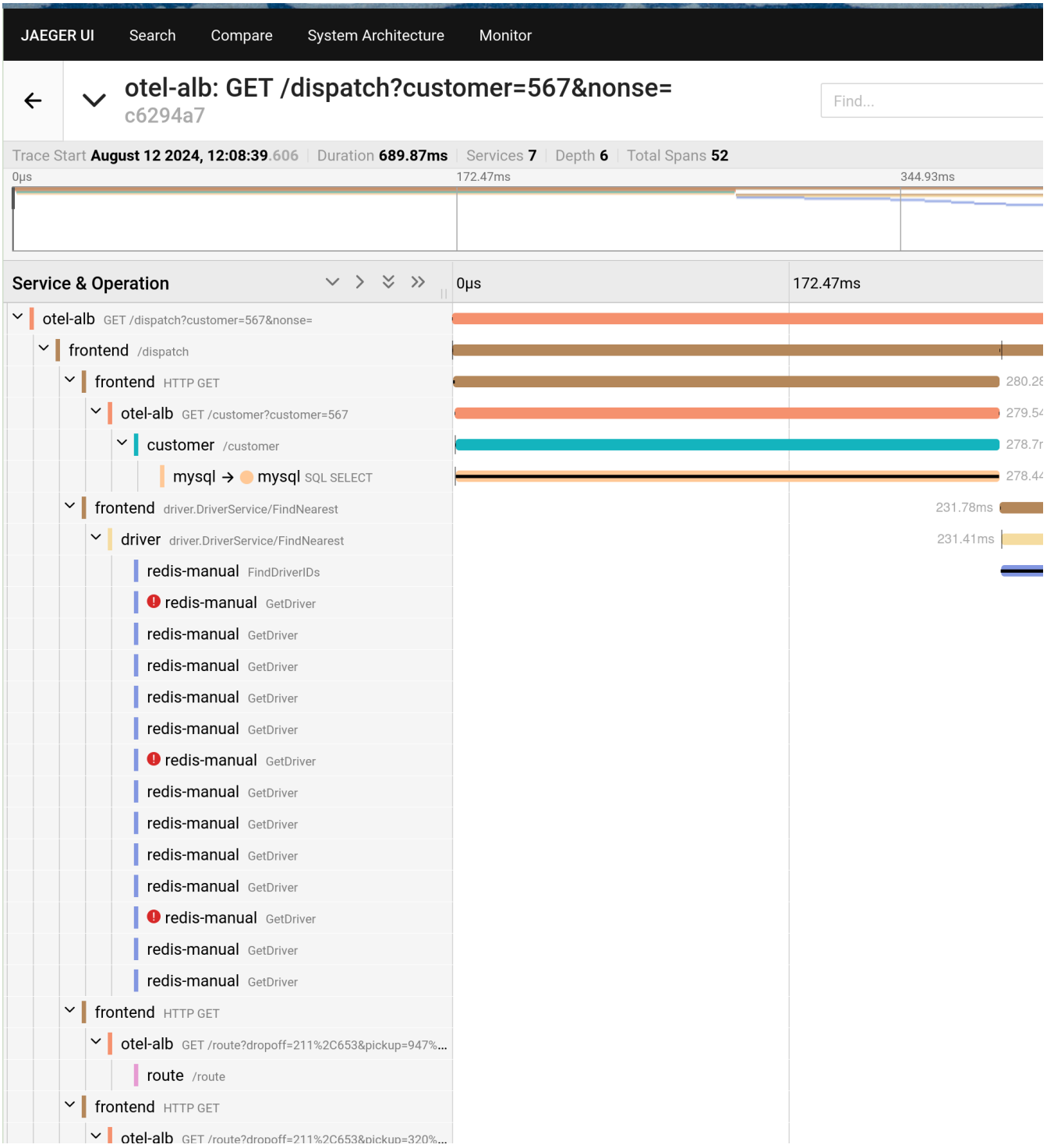
```
export ALB_IP=$(kubectl get po -A -o wide|grep otel-alb | awk '{print $7}');echo $ALB_IP
```

5. 执行以下命令以通过 ALB 向 HotROD 发送请求。此时，ALB 将报告 Trace 到 Jaeger。

```
curl -v "http://<$ALB_IP>:80/dispatch?customer=567&nonse=" # 将命令中的 <$ALB_IP> 替换为之前步骤中获取的 otel-alb 的访问地址
```

6. 打开在 步骤 3 中获得的 Jaeger 访问地址以查看结果。


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%...

Alauda Container Platform

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 类型的 Service

LoadBalancer 类型 Service 注解

创建 Ingress

实现方式

前提条件

Ingress 示例：

使用 Web 控制台创建 Ingress

使用 CLI 创建 Ingress

配置网关

术语

先决条件

示例网关和 Alb

通过 Web 控制

通过 CLI 创建网

查看平台创建的

更新网关

创建域名

示例域名自定义资源 (CR)

通过 Web 控制台创建域名

通过 CLI 创建域名

创建证书

通过 Web 控制台创建证书

建路

配置子网

IP 分配规则

Calico 网络

Kube-OVN 网络

子网管理

通过 CLI 创建外部 IP 地址池

配置网络策略

通过 Web 控制

通过 CLI 创建 N

参考

创建 **Admin** 网络策略

创建 **BGP** 对等体

配置集群网络

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**

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 集合通常由您定义的 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）

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

```
spec:
  clusterIP: None
```

无头服务适用于：

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

通过 **Web** 控制台创建服务

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

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

参数	说明
类型	• **ClusterIP**：在集群内部 IP 上暴露 Service。选择此值时，Service 只能从集群内部访问。 • **NodePort**：在每个节点的 IP 上以静态端口（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，您必须自行提供，或者将 Kubernetes 集群与云提供商集成。
目标组件	• **Workload**：Service 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 将请求转发到特定的虚拟机或虚拟机组。 • **Label Selector**：Service 将请求转发到具有指定标签的某类工作负载，例如 `environment: release`。
端口	用于配置此 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* 调用此 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：自动生成，格式为 `<protocol>-<service port>-<container port>`，例如：*tcp-80-6379* 或 *tcp-80-redis*。

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

5. 点击 **Create**。

通过 **CLI** 创建服务

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

基于已有的 deployment 资源 `my-app` 创建服务。

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

示例：集群内访问应用

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

1. 应用此 YAML：

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

2. 启动另一个 Pod：

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

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

```
wget -qO- http://nginx-clusterip
# 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
```

您应该能看到包含 “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. 应用此 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}
```

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

当然，也可以通过创建类型为 LoadBalancer 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. 应用此 YAML：

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

2. 获取外部 IP 地址：

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

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

`EXTERNAL-IP` 即为您从浏览器访问的地址。

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

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

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

示例：**ExternalName** 类型的 **Service**

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

1. 应用此 YAML：

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

2. 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
```

您会看到解析结果为 `example.com`。

LoadBalancer 类型 Service 注解

AWS EKS 集群

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

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

Key	Value	说明
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 • dualstack	指定支持的 IP 地址栈。

华为云 CCE 集群

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

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

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

Azure AKS 集群

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

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

Google GKE 集群

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

Key	Value	说明
networking.gke.io/load-balancer-type	Internal	指定使用私有网络。
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 Pod。

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），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。

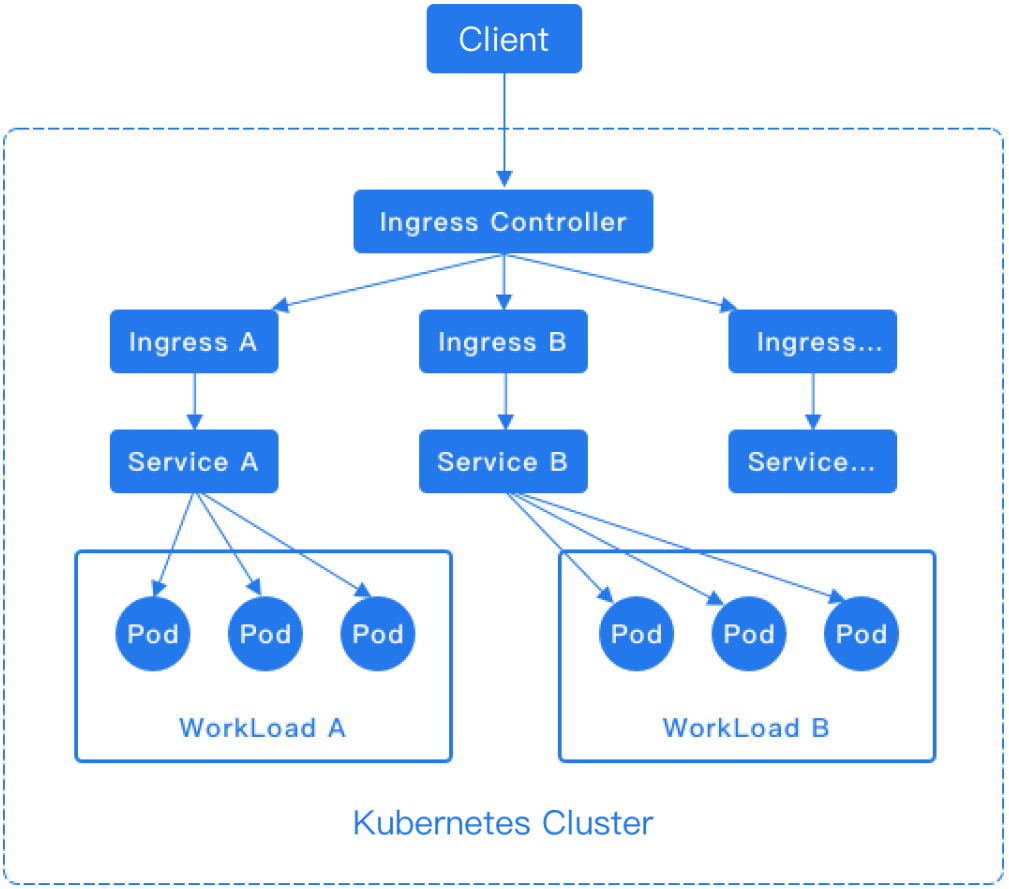
> **WARNING**
>
> 在同一命名空间内创建多个 ingress 时，不同的 ingress 不得 具有相同的 域名、协议 和 路径（即不允许重复的访问入口）。

目录

实现方式

Ingress 规则依赖于 Ingress Controller 的实现，Ingress Controller 负责监听 Ingress 和 Service 的变化。在创建新的 Ingress 规则后，Ingress Controller 内部会自动生成与该 Ingress 规则匹

配的转发规则。当 Ingress Controller 收到请求时，会根据 Ingress 规则匹配转发规则，并将流量分发到指定的内部路由，如下图所示。



NOTE

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

快速开始

接下来，我们将使用社区版本的 Ingress-NGINX，演示如何使用 NGINX controller 访问您自己的应用。

1. 部署 `Ingress-NGINX` controller。

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.12.2/deploy/static/provider/cloud/deploy.yaml
```

使用该命令后会自动创建以下资源：

类型	名称	描述
`Namespace`	`ingress-nginx`	用于隔离 Controller 的资源
`ServiceAccount`	`ingress-nginx`	Controller 使用的服务账号
`ClusterRole`	`ingress-nginx`	集群范围权限
`ClusterRoleBinding`	`ingress-nginx`	将 ClusterRole 绑定到服务账号
`ConfigMap`	`ingress-nginx-controller`	配置 Controller 行为（如日志级别、代理超时等）
`ValidatingWebhookConfig`	`ingress-nginx-admission`	用于验证 Ingress 配置合法性的 webhook（可选）
`Service` (TCP/UDP)	`ingress-nginx-controller`	类型默认为 `LoadBalancer`，可更改为 `NodePort`。
`Deployment`	`ingress-nginx-controller`	
`Pod`	`ingress-nginx-controller-xxx`	
`Role` / `RoleBinding`	admission 相关	支持 webhook
`Job`	`ingress-nginx-admission-create`	webhook 注册

如果想更改默认的镜像仓库地址，可以先用 `curl` 下载 YAML 文件，修改后再应用该 YAML 文件。

```
curl -O https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.12.2/deploy/static/provider/cloud/deploy.yaml
```

等待 `ingress-nginx-controller-xxx` Pod 运行。

2. 本地测试

- 创建一个简单的 Web 服务器及其对应的 Service：

```
kubectl create deployment demo --image=nginx --port=80
kubectl expose deployment demo
```

- 创建一个 ingress 资源。此示例使用映射到 `localhost` 的主机：

```
kubectl create ingress demo-localhost --class=nginx \
  --rule="demo.local/*=demo:80"
```

- 将本地端口转发到 ingress controller：

```
kubectl port-forward --namespace=ingress-nginx service/ingress-nginx-
controller 8080:80
```

- 使用 curl 访问您的部署：

```
curl --resolve demo.local:8080:127.0.0.1 http://demo.local:8080
```

注意：此参数临时将域名 demo.local 解析到 IP 127.0.0.1，并用于端口 8080。当您访问 http://demo.local:8080 ↗ 时，实际上访问的是 http://127.0.0.1:8080 ↗。 另外，您应配置 `hosts` 文件：

```
echo "127.0.0.1 demo.local" | sudo tee -a /etc/hosts
```

最终您应该看到包含 “Welcome to nginx!” 字样的 HTML 响应。

然后您就可以访问网站 `http://demo.local:8080/`。

INFO

`ingress-nginx-controller` 默认类型为 `LoadBalancer`，如果 `EXTERNAL-IP` 字段显示为 `pending`，说明您的 Kubernetes 集群无法自动创建负载均衡器。

如果您使用的云服务商支持通过（特定云服务商的）annotations 指定 Service 的负载均衡器 IP 地址，建议切换为该方式。

3. 在线测试

   当您的 `ingress-nginx-controller`（LoadBalancer 类型的 Service）存在 `EXTERNAL-IP` 时，您可以创建 ingress 资源。以下示例假设您已为 `www.developer.io` 设置了 DNS 记录：

   ```
   kubectl create ingress demo --class=nginx \
     --rule="www.developer.io/*=demo:80"
   ```

   您可以访问 `http://www.developer.io` 来查看相同的输出。

前提条件

- 当前命名空间内必须存在可用的 **Service**。
- 请与管理员确认已为当前命名空间关联的项目分配了可用的域名。
- 若需通过 HTTPS 访问域名，需先将 HTTPS 证书保存为 TLS secret。

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. 使用 `ingress-nginx` controller。
3. 如果只想在本地运行 ingress，请提前配置 `hosts`。

使用 **Web** 控制台创建 **Ingress**

1. 访问 容器平台。
2. 在左侧导航栏点击 网络 > **Ingress**。
3. 点击 创建 **Ingress**。
4. 参考以下说明配置相关参数。

参数	说明
Ingress Class	不同的 ingress controller 可能有不同的 `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	主机可以是精确匹配（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：如果您使用自定义 Ingress controller 管理 Ingress 规则，可以选择由 controller 决定。
Service	外部流量将转发到此 Service。
Service 端口	指定流量将转发到 Service 的哪个端口。

5. 点击 创建。

使用 CLI 创建 Ingress

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

> **NOTE**
>
> 如果 ingress 没有指定 Ingress Class，所有分配给该项目的 ALB 实例都会处理该 ingress。

配置网关

入站网关（Gateway）是从网关类（Gateway Class）部署的实例。它创建监听器以捕获指定域名和端口上的外部流量。结合路由规则，它可以将指定的外部流量路由到相应的后端实例。

创建入站网关以实现更细粒度的网络资源分配。

目录

术语

资源名称	概述	使用说明
网关类	在标准网关 API 文档中，网关类被定义为创建网关的模板。不同的模板可以为不同的业务场景创建入站网关，从而促进快速的流量管理。	平台包括专用的网关类。
入站网关	入站网关对应于特定的资源实例，用户可以独占使用该入站网关的所有监听和计算资源。它是对监听器有效的路由规则的配置。当网关检测到外部流量时，将根据路由规则将其分配到后端实例。	可以视为负载均衡器实例。
路由规则	路由规则定义了一系列从网关到服务的流量分配指南。当前网关 API 中支持的标准路由规则类型包括 HTTPRoute、TCPRoute、UDPRoute 等。	平台当前支持监听 HTTP、HTTPS、TCP 和 UDP 协议。

先决条件

平台管理员必须确保集群支持 LoadBalancer 类型的内部路由。对于公有云集群，必须安装 LoadBalancer 服务控制器。在非公有云集群中，平台提供外部地址池功能，允许 LoadBalancer 类型的内部路由在配置完成后自动从外部地址池中获取 IP 以供外部访问。

示例网关和 **Alb2** 自定义资源（**CR**）

```yaml
# demo-gateway.yaml
apiVersion: gateway.networking.k8s.io/v1beta1
kind: Gateway
metadata:
  namespace: k-1
  name: test
  annotations:
    cpaas.io/display-name: ces
    listeners.cpaas.io/creationTimestamp: '["2025-05-26T02:05:56.135Z"]'
    listeners.cpaas.io/display-name: '[""]'
  labels:
    alb.cpaas.io/alb-ref: test-o93q7
spec:
  gatewayClassName: exclusive-gateway (1)
  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 (2)
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 3
```

1. 请参见下面的网关类介绍。
2. `alb2` 名称格式为 `{gatewayName}-{random}`。
3. `gateway` 名称。

通过 Web 控制台创建网关

1. 进入 容器平台。
2. 在左侧导航栏中，点击 网络 > 入站网关。
3. 点击 创建入站网关。
4. 根据以下说明配置特定参数。

参数	描述
名称	入站网关的名称。
网关类	网关类定义了网关的行为，类似于存储类（StorageClasses）的概念；它是一个集群资源。 专用：入站网关将对应于特定的资源实例，用户可以利用该网关的所有监听和计算资源。
规格	您可以根据需要选择推荐的使用场景或自定义资源限制。
访问地址	入站网关的地址，默认情况下自动获取。
内部路由注释	用于声明 LoadBalancer 类型内部路由的配置或能力。有关特定注释信息，请参阅 LoadBalancer 类型内部路由注释说明。

5. 点击 创建。

通过 **CLI** 创建网关

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

查看平台创建的资源

入站网关创建后，平台会自动创建许多资源。请勿删除以下资源。

默认创建的资源	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*
服务账户	*name-lb-random-serviceaccount*

更新网关

NOTE

更新入站网关将导致 3-5 分钟的服务中断。请在适当的时间进行此操作。

通过 **Web** 控制台更新网关

1. 访问 容器平台。
2. 在左侧导航栏中，点击 网络 > 入站网关。
3. 点击⋮> 更新。
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 更新。

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将其转发到后端实例。

先决条件

- 如果需要监控 HTTP 协议，请提前联系管理员准备 域名。
- 如果需要监控 HTTPS 协议，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. 在左侧导航栏中，点击 网络 > 入站网关。
2. 点击 *入站网关名称*。
3. 点击 添加监听器。
4. 根据以下说明配置特定参数。

参数	描述
监听器协议和端口	当前支持监控 HTTP、HTTPS、TCP 和 UDP 协议，您可以自定义输入要监控的端口，例如：`80`。 注意： • 当端口相同时，HTTP、HTTPS 和 TCP 监听器类型不能共存；只能选择其中一种协议。 • 当使用 HTTP 或 HTTPS 协议时，如果端口相同，域名必须不同。
域名	选择当前命名空间中可用的域名，用于监控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 创建。

通过 **CLI** 添加监听器

```
kubectl patch gateway test \
  -n k-1 \
  --type=merge \
  -p '{
    "metadata": {
      "annotations": {
        "listeners.cpaas.io/creationTimestamp": "[\"2025-05-26T02:05:56.1
35Z\",\"2025-05-26T03:33:52.431Z\"]",
        "listeners.cpaas.io/display-name": "[\"\",\"\" ]"
      }
    },
    "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）。它们将网关监控的网络流量暴露给集群的内部路由（Kubernetes Service），促进路由转发策略。关键区别在于它们针对不同的服务对象：入站规则服务于 Ingress Controller，而路由规则服务于 Ingress Gateway。

一旦在 Ingress Gateway 中设置了监听，网关将实时监控来自指定域名和端口的流量。路由规则可以根据需要将入站流量转发到后端实例。

示例 **HTTPRoute** 自定义资源（**CR**）

```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. 可用的类型有：`HTTPRoute`、`TCPRoute`、`UDPRoute`。
2. `Gateway` 监听器名称。

> **NOTE**
>
> 如果 HTTPRoute 类型路由规则中没有匹配 **Path** 对象的规则，将自动添加一个匹配规则，模式为 PathPrefix，值为 /。

通过 **Web** 控制台创建路由

1. 访问 容器平台。
2. 在左侧导航栏中，点击 网络 > 路由规则。
3. 点击 创建路由规则。
4. 按照以下说明配置一些参数。

参数	描述
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	在左侧选择框中，选择已创建的 **Ingress Gateway**，在右侧选择框中，选择已创建的 **Listener**。平台将把创建的路由规则发布到下面的监听器，使网关能够将捕获的流量转发到指定的后端实例。 注意：不允许将路由规则发布到端口为 **11782** 的监听器或已挂载 TCP 或 UDP 路由的监听器。
匹配	您可以添加一个或多个匹配规则，以捕获符合要求的流量。例如，捕获指定路径的流量、捕获指定方法的流量等。 注意：

参数	描述
	• 点击 添加；当添加多个路由规则时，规则之间的关系为 'AND'，所有规则必须匹配才能生效。 • 点击 添加匹配；当添加多组路由规则时，组之间的关系为 'OR'，任意组匹配均可生效。 • TCPRoute 和 UDPRoute 不支持配置匹配规则。 • 当匹配对象为 **path**，且匹配方法为 **Exact** 或 **PathPrefix** 时，输入的 **value** 必须以 "/" 开头，并禁止使用 "//"、"/./"、"/../"、"%2f"、"%2F"、"#"、"/.."、"/." 等字符。
操作	您可以添加一个或多个操作来处理捕获的流量。 • 头部：HTTP 消息的头部包含许多元数据，提供有关请求或响应的附加信息。通过修改头部字段，服务器可以影响请求和响应的处理方式。 • 重定向：匹配的 URL 将以指定的方式处理，然后请求将重新发起。 • 重写：匹配的 URL 将以指定的方式处理，然后请求将重定向到不同的资源路径或文件名。 注意： • 点击 添加；当添加多个操作规则时，平台将根据规则的显示顺序依次执行所有操作。 • TCPRoute 和 UDPRoute 不支持配置操作规则。 • 在同一路由规则内，不能有多个 **Header** 类型的操作具有相同的 **value**。 • 在同一路由规则内，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**。 • 如果希望使用 **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中选择的内部路由和端口转发到后端实例。您还可以设置权重，权重值越高，被轮询的概率越大。 提示：权重旁边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 创建。

通过 **CLI** 创建路由

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

创建域名

向平台添加域名资源，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**
>
> 在平台上创建的域名必须解析到集群的负载均衡地址后，才能通过该域名进行访问。因此，您需要确保在平台上添加的域名已成功注册，并且域名解析到集群的负载均衡地址。

在平台上成功创建并分配的域名可以在 容器平台 的以下功能中使用：

- 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
- 添加监听端口以支持负载均衡：网络管理 > 负载均衡器详情 > 添加监听端口

一旦域名绑定到证书，应用开发者可以在配置负载均衡器和入站规则时直接选择域名，从而使用与域名一起提供的证书以支持 https。

目录

示例域名自定义资源 (CR)

```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. 如果启用了证书，必须提前创建一个 LTS 类型的 Secret。`secret-ref` 是密钥名称。

通过 Web 控制台创建域名

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

参数	描述
类型	• 域名：完整的域名，例如 `developer.test.cn`。 • 泛域名：带有通配符 (*) 字符的域名，例如 `*.test.cn`，包括域名 `test.cn` 下的所有子域名。

参数	描述
域名	根据所选的域名类型输入完整的域名或域名后缀。
分配集群	如果分配了集群，还需要选择与所分配集群相关联的项目，例如与该集群相关的所有项目。
证书	包括用于创建绑定到域名的证书的公钥 (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。 • 绑定的证书应满足正确格式、在有效期内、并且对域名进行了签名等条件。 • 创建绑定证书后，绑定证书的名称格式为：域名 - 随机字符。 • 创建绑定证书后，可以在证书列表中查看，但只支持在域名详情页面上更新和删除绑定证书。 • 创建绑定证书后，支持更新证书内容，但不支持替换其他证书。

5. 点击 创建。

通过 **CLI** 创建域名

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

后续操作

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

其他资源

- 配置证书

创建证书

在平台管理员导入 TLS 证书并将其分配给指定项目后，具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，在证书过期等场景中，平台管理员可以集中更新证书。

> **NOTE**
>
> 当前不支持在公有云集群中使用证书功能。您可以在指定的命名空间内根据需要创建 TLS 类型的 Secret 字典。

目录

通过 Web 控制台创建证书

1. 进入 平台管理。
2. 在左侧导航栏中，点击 网络管理 > 证书。
3. 点击 创建证书。
4. 请参考以下说明配置相关参数。

参数	描述
分配项目	• 所有项目：将证书分配用于当前集群关联的所有项目。 • 指定项目：将证书分配用于指定项目。 • 不分配：暂不分配项目。证书创建完成后，您可以通过 更新项目 操作更新可以使用该证书的项目。
公钥	这指的是 tls.crt。在导入公钥时，不支持二进制文件。
私钥	这指的是 tls.key。在导入私钥时，不支持二进制文件。

5. 点击 创建。

创建外部 IP 地址池

外部 IP 地址池是一组 IP，MetalLB 利用这些 IP 来获取负载均衡器类型内部路由的外部访问 IP。

目录

前提条件

如果您需要使用 BGP 类型的外部 IP 地址池，请联系管理员以启用相关功能。

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以第二层（L2）互联。

- IP 必须可以被平台使用，且不能包括物理网络中已使用的 IP，例如网关 IP。
- 必须与集群使用的网络没有重叠，包括集群 CIDR、服务 CIDR、子网等。
- 在双栈环境中，确保同时在同一外部地址池中存在 IPv4 和 IPv6 地址，且它们的数量都大于 0。否则，双栈负载均衡器类型的内部路由将无法获取外部访问地址。
- 在 IPv6 环境中，节点的 DNS 必须支持 IPv6；否则，MetalLB 插件无法成功部署。

部署 **MetalLB** 插件

使用外部地址池依赖于 MetalLB 插件。

1. 进入 平台管理。
2. 在左侧导航栏中，点击 市场 > 集群插件。
3. 搜索 MetalLB，点击 **MetalLB** 右侧的⋮> 部署。
4. 等待部署状态显示 部署成功 以完成部署。

示例 **IPAddressPool** 自定义资源 **(CR)**

```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 池参考。

INFO

Q: 什么是 `L2Advertisement` ?

A:

1. `L2Advertisement` 是 MetalLB 提供的自定义资源 (CRD)，用于控制哪些 IP 地址池的地址应通过 ARP (IPv4) 或 NDP (IPv6) 在第二层模式下广播。

Q: `L2Advertisement` 的目的是什么？

A:

1. 指定在 IPAddressPool 中哪些 IP 地址进行 L2 广播 (ARP/NDP 广告)；
2. 控制广播行为以防止 IP 冲突或跨段广播；

3. 在多 NIC、多网络环境中限制广播范围。

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）。

在第二层模式下，如果未定义 `L2Advertisement`，MetalLB 将不会广播任何地址。

Q: MetalLB 中的 `BGPAdvertisement` 是什么？

A:

`BGPAdvertisement` 是 Kubernetes 自定义资源定义 (CRD)，用于 MetalLB↗，这是一个用于裸金属 Kubernetes 集群的负载均衡器实现。它控制如何通过 BGP (边界网关协议) 将 IP 地址范围（在 `IPAddressPool` 中定义）通告给外部网络。

Q: `BGPAdvertisement` 重要吗？

A:

在 MetalLB 的 BGP 模式下，控制器与外部路由器通过 BGP 建立对等关系，并通告分配给 Kubernetes `Service` 对象的 IP。`BGPAdvertisement` 资源允许您：

- 控制哪些地址池被通告
- 自定义路由通告设置，例如：
  - 路由聚合
  - BGP 社区
  - 本地优先级 (BGP 优先级)

如果未定义 `BGPAdvertisement`，即使您已配置 BGP 对等体，MetalLB 也不会通告任何地址。

通过 **Web** 控制台创建外部 **IP** 地址池

1. 进入 平台管理。

2. 在左侧导航栏中，点击 网络管理 > 外部 **IP** 地址池。

3. 点击 创建外部 **IP** 地址池。

4. 根据以下说明配置某些参数。

参数	描述
类型	• L2：基于 MAC 地址的通信和转发，适合需要简单快速的第二层交换的小型或局域网络，具有配置简单和延迟低的优点。 • BGP（Alpha）：基于 IP 地址的路由和转发，使用 BGP 协议交换路由信息，适合需要在多个自治系统中进行复杂路由的大型网络，具有高可扩展性和可靠性的优点。
IP 资源	支持 CIDR 和 IP 范围格式的输入。点击 添加 支持多个条目，示例如下： **CIDR**：`192.168.1.1/24`。 **IP** 范围：`192.168.2.1` ~ `192.168.2.255`。
可用节点	在 L2 模式下，可用节点是用于承载所有 VIP 流量的节点；在 BGP 模式下，可用节点是用于承载 VIP，与对等方建立 BGP 连接并向外通告路由的节点。 • 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：以列表格式查看最终可用节点。 注意： • 使用 BGP 类型时，可用节点是下一跳节点；确保所选可用节点是 BGP 连接节点 的子集。 • 您可以单独配置标签选择器或节点名称以选择可用节点；如果同时配置了两者，则最终可用节点是两者的交集。
BGP 对等体	选择 BGP 对等体；有关具体配置，请参考 BGP 对等体。

5. 点击 创建。

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 平台管理。

2. 在左侧导航栏中，点击 网络管理 > 外部 **IP** 地址池。

3. 点击页面右上角的 查看告警策略 以查看 MetalLB 的通用告警策略。

配置子网

目录

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，将会随机从其中一个子网中选择一个 IP 地址。

- 项目分配：
  - 如果一个项目没有绑定到子网，则该项目下所有命名空间中的 Pods 只能使用默认子网中的 IP 地址。如果默认子网中的 IP 地址不足，Pods 将无法启动。
  - 如果一个项目绑定到子网，则该项目下所有命名空间中的 Pods 只能使用该特定子网中的 IP 地址。
- 命名空间分配：
  - 如果一个命名空间没有绑定到子网，则该命名空间中的 Pods 只能使用默认子网中的 IP 地址。如果默认子网中的 IP 地址不足，Pods 将无法启动。

- 如果一个命名空间绑定到子网，则该命名空间中的 Pods 只能使用该特定子网中的 IP 地址。

Calico 网络

在 Calico 网络中创建子网，以实现集群内资源的更细粒度的网络隔离。

约束和限制

在 IPv6 集群环境中，默认情况下，在 Calico 网络中创建的子网使用 VXLAN 封装。所需的 VXLAN 封装端口与 IPIP 封装的端口不同。您需要确保 UDP 端口 4789 是开放的。

使用 Calico 网络的示例子网自定义资源 (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. 请参见封装模式参数和封装协议参数。
3. 请参见出站流量 NAT 参数。

通过 Web 控制台在 Calico 网络中创建子网

1. 转到 平台管理。

2. 在左侧导航栏中，单击 网络管理 > 子网。

3. 单击 创建子网。

4. 根据以下说明配置相关参数。

参数	描述
CIDR	在将子网分配给项目或命名空间后，命名空间内的容器组将随机使用此 CIDR 内的 IP 地址进行通信。 注意：有关 CIDR 和 BlockSize 之间的对应关系，请参见 参考内容。
封装协议	选择封装协议。**IPIP** 在双栈模式下不受支持。 • **IPIP**：使用 IPIP 协议实现跨段通信。 • **VXLAN (Alpha)**：使用 VXLAN 协议实现跨段通信。 • 无封装：通过路由转发直接连接。
封装模式	当封装协议为 **IPIP** 或 **VXLAN** 时，必须设置封装模式，默认为 **Always**。 • **Always**：始终启用 IPIP / VXLAN 隧道。 • 跨子网：仅当主机在不同子网时启用 IPIP / VXLAN 隧道；当主机在同一子网时通过路由转发直接连接。
出站流量 **NAT**	选择是否启用出站流量 NAT（网络地址转换），默认启用。 主要用于设置子网容器组访问外部网络时暴露给外部网络的访问地址。 当启用出站流量 NAT 时，主机 IP 将作为当前子网容器组的访问地址；当未启用时，子网内容器组的 IP 将直接暴露给外部网络。

5. 单击 确认。

6. 在子网详细信息页面，选择 操作 > 分配项目 / 分配命名空间。

7. 完成配置并单击 分配。

通过 **CLI** 在 **Calico** 网络中创建子网

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

参考内容

CIDR 和 blockSize 之间的动态匹配关系如下表所示。

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

NOTE

不支持前缀大于 31 的子网配置。

Kube-OVN 网络

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

> **NOTE**
>
> 平台内置了 **join** 子网，用于节点和 Pods 之间的通信；请避免 **join** 和新创建的子网之间的网络段冲突。

使用 Kube-OVN Overlay 网络的示例子网自定义资源 (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. 请参见出站流量 NAT 参数。
2. 请参见保留 IP 参数。
3. 请参见网关类型参数。可用值为 `distributed` 或 `centralized`。
4. 请参见网关节点参数。

5. 请参见 ECMP 参数。前提是您联系管理员启用该功能。

通过 **Web** 控制台在 **Kube-OVN Overlay** 网络中创建子网

1. 转到 平台管理。

2. 在左侧导航栏中，单击 网络管理 > 子网。

3. 单击 创建子网。

4. 根据以下说明配置相关参数。

参数	描述
网络段	在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 Pods 使用。
保留 **IP**	设置的保留 IP 将不会被自动分配。例如，可以用作计算组件的 固定 **IP** 地址。
网关类型	选择子网的网关类型以控制出站流量。 - 分布式：集群中的每个主机都可以作为当前主机上 Pods 的出站节点，实现分布式出站。 - 集中式：集群中的所有 Pods 使用一个或多个特定主机作为出站节点，便于外部审计和防火墙控制。设置多个集中式 网关节点 可以实现高可用性。
ECMP (Alpha)	选择 集中式 网关时，可以使用 ECMP 功能。默认情况下，网关以主从模式运行，只有主网关处理流量。启用 ECMP（等成本多路径路由）后，出站流量将通过多个等成本路径路由到所有可用的网关节点，从而增加网关的总吞吐量。 注意：请提前启用与 ECMP 相关的功能。
网关节点	使用 集中式 网关时，选择一个或多个特定主机作为网关节点。
出站流量 **NAT**	选择是否启用出站流量 NAT（网络地址转换）。默认情况下启用。 主要用于设置 Pods 访问互联网时暴露给外部网络的访问地址。 当启用出站流量 NAT 时，主机 IP 将作为当前子网内 Pods 的访问地址；当未启用时，子网内 Pods 的 IP 将直接暴露给外部网络。在这种情况下，建议使用集中式网关。

5. 单击 确认。

6. 在子网详细信息页面，选择 操作 > 分配项目 / 命名空间。

7. 完成配置并单击 分配。

通过 CLI 在 Kube-OVN Overlay 网络中创建子网

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

下层网络

在 Kube-OVN 下层网络中创建子网，不仅可以实现资源的更细粒度网络隔离，还可以提供更好的性能体验。

> **INFO**
>
> Kube-OVN 下层的容器网络需要物理网络的支持。请参阅最佳实践 准备 Kube-OVN 下层物理网络 以确保网络连接。

使用说明

在 Kube-OVN 下层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. 默认网络卡名称。
2. 按节点配置网络卡。

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. 默认网络卡名称。
2. 按节点配置网络卡。

桥接网络是指一个桥接，在将网络卡绑定到桥接后，可以转发容器网络流量，实现与物理网络的互通。

操作步骤：

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

   注意：

   - *目标 Pod* 是指调度在当前节点上的所有 Pods 或调度到当前节点的绑定到特定子网的命名空间中的 Pods。这取决于桥接网络下子网的范围。
   - 下层子网中的节点必须具有多个网络卡，桥接网络使用的网络卡必须专门分配给下层，不能承载其他流量，例如 SSH。例如，如果桥接网络有三个节点计划使用 eth0、eth0、eth1 专门用于下层，则默认网络卡可以设置为 eth0，节点三的网络卡可以设置为 eth1。

参数	描述
默认网络卡名称	默认情况下，目标 Pod 将使用此作为与物理网络互通的桥接网络卡。
按节点配置网络卡	在配置的节点上，目标 Pods 将桥接到指定的网络卡，而不是默认网络卡。
排除节点	当排除节点时，所有调度到这些节点的 Pods 将不会桥接到这些节点上的任何网络卡。 注意：排除节点上的 Pods 将无法与物理网络或跨节点容器网络进行通信，需注意避免将相关 Pods 调度到这些节点。

5. 单击 添加。

通过 CLI 添加桥接网络

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

通过 Web 控制台添加 VLAN（可选）

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

1. VLAN ID。
2. 桥接网络引用。

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

操作步骤：

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

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

5. 单击 添加。

通过 CLI 添加 VLAN

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

使用 Kube-OVN 下层网络的示例子网自定义资源 (CR)

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

1. VLAN 引用。

通过 **Web** 控制台在 **Kube-OVN** 下层网络中创建子网

> **NOTE**
>
> 平台还预配置了 **join** 子网，用于在 Overlay 传输模式下节点和 Pods 之间的通信。此子网在下层传输模式下将不被使用，因此必须避免 **join** 和其他子网之间的 IP 段冲突。

操作步骤：

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

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

5. 单击 确认。

6. 在子网详细信息页面，选择 操作 > 分配项目 / 命名空间。

7. 完成配置并单击 分配。

通过 CLI 在 Kube-OVN 下层网络中创建子网

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

相关操作

当集群中同时存在下层和 Overlay 子网时，您可以根据需要配置 下层和 Overlay 子网之间的自动互通。

子网管理

通过 Web 控制台更新网关

这包括更改出站流量方式、网关节点和 NAT 配置。

1. 转到 平台管理。

2. 在左侧边栏中，单击 网络管理 > 子网。

3. 单击子网的名称。

4. 选择 操作 > 更新网关。

5. 更新参数配置；有关详细信息，请参见 参数描述。

6. 单击 确定。

通过 CLI 更新网关

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

通过 Web 控制台更新保留 IP

网关 IP 不能从保留 IP 中删除，而其他保留 IP 可以自由编辑、删除或添加。

1. 转到 平台管理。

2. 在左侧边栏中，单击 网络管理 > 子网。

3. 单击子网的名称。

4. 选择 操作 > 更新保留 **IP**。

5. 更新完成后，单击 更新。

通过 CLI 更新保留 IP

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

通过 Web 控制台分配项目

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

1. 转到 平台管理。
2. 在左侧边栏中，单击 网络管理 > 子网。
3. 单击子网的名称。
4. 选择 操作 > 分配项目。
5. 添加或移除项目后，单击 分配。

通过 CLI 分配项目

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

通过 Web 控制台分配命名空间

将子网分配给特定命名空间可以实现更细粒度的网络隔离。

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

1. 转到 平台管理。
2. 在左侧边栏中，单击 网络管理 > 子网。
3. 单击子网的名称。
4. 选择 操作 > 分配命名空间。
5. 添加或移除命名空间后，单击 分配。

通过 CLI 分配命名空间

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

通过 Web 控制台扩展子网

当子网的保留 IP 范围达到使用限制或即将耗尽时，可以根据原子网范围进行扩展，而不会影响现有服务的正常运行。

1. 转到 平台管理。
2. 在左侧边栏中，单击 网络管理 > 子网。
3. 单击子网的名称。
4. 选择 操作 > 扩展子网。

5. 完成配置并单击 更新。

通过 CLI 扩展子网

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

管理 Calico 网络

支持分配项目和命名空间；有关详细信息，请参见 项目分配 和 命名空间分配。

通过 Web 控制台删除子网

NOTE

- 当子网被删除时，如果仍有容器组使用子网内的 IP，容器组可以继续运行，IP 地址将保持不变，但它们将无法通过网络进行通信。容器组可以重建以使用默认子网内的 IP，或为容器组所在的命名空间分配新的子网以供使用。
- 默认子网无法删除。

1. 转到 平台管理。

2. 在左侧导航栏中，单击 网络管理 > 子网。

3. 单击⋮> 删除，并继续进行删除。

通过 CLI 删除子网

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

创建网络策略

> **INFO**
>
> 平台现在提供了两种不同的网络策略 UI。旧版 UI 为了兼容性仍然保留，而新版 UI 更加灵活，并提供了原生的 YAML 编辑器。我们推荐使用新版 UI。
>
> 请联系平台管理员启用 `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站和出站流量。 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

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；如果未设置，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。
入站	阻止所有入站流量		阻止所有流向目标 Pod 的入站流量。 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 阻止所有入站流量 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 阻止所有入站流量 选项。
	规则 说明：如果规则中添加了多个来源，它们	当前命名空间内的 Pod	匹配当前命名空间内具有指定标签的 Pod；只有匹配的 Pod 可以访问目标 Pod。您可以点击 **Preview** 查看当前规则影响的 Pod。如果未配置此项，默认允许当前命名空间内所有 Pod 访问目标 Pod。

区域	参数		说明
	之间是逻辑上的 或 关系。	当前集群内的 Pod	匹配集群内具有指定标签的命名空间或 Pod；只有匹配的 Pod 可以访问目标 Pod。您可以点击 **Preview** 查看当前规则影响的 Pod。 • 如果同时配置了命名空间选择器和 Pod 选择器，则取两者的交集，即从指定命名空间中选择具有指定标签的 Pod。 • 如果未配置此项，默认允许集群内所有命名空间的所有 Pod 访问目标 Pod。
		IP 范围	输入允许访问目标 Pod 的 CIDR，可以排除不允许访问的 CIDR 范围。如果未配置此项，则允许任何流量访问目标 Pod。 说明: 可以通过添加 *example_ip*/32 的形式排除单个 IP 地址。
	端口		匹配指定协议和端口的流量；可以添加数字端口或 Pod 上的端口名称。如果未配置此项，则匹配所有端口。
出站	阻止所有出站流量		阻止所有从目标 Pod 发出的出站流量。 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 阻止所有出站流量 选项。
	其他参数		与 入站 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

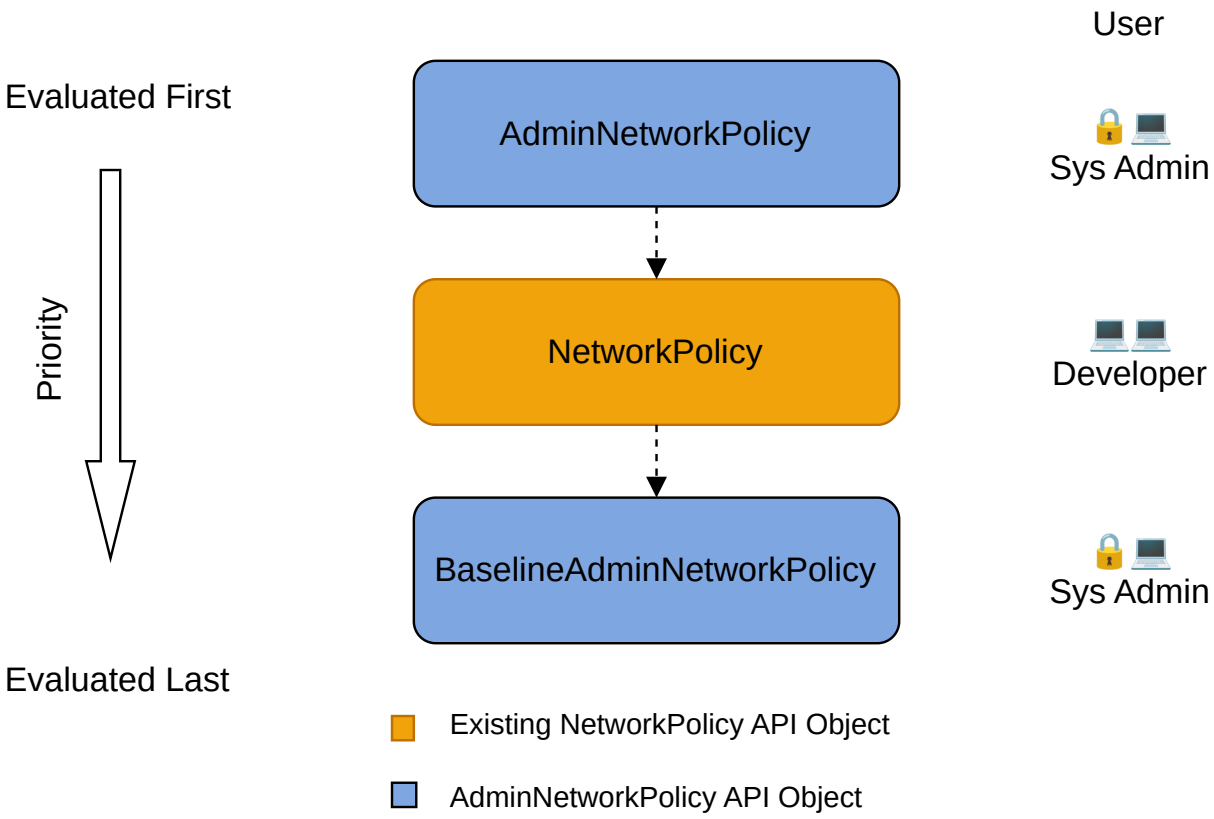
平台现提供两种不同的集群网络策略 UI。旧版 UI 为兼容性保留，新版 UI 更加灵活，并提供原生 YAML 编辑器。我们推荐使用新版 UI。

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 community 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

- 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. 数值越小，优先级越高。

2. `subject`：最多只能指定 namespace selector 或 pod selector 中的一种。
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. 可选值为 Namespace Selector、Pod Selector。
5. 可选值为 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=`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 平台管理。

2. 在左侧导航栏点击 网络 > 集群网络策略。

3. 点击 创建 **Admin** 网络策略 或 配置 **Baseline Admin** 网络策略。

4. 按照以下说明完成相关配置。

区域	参数	说明
基本信息	名称	Admin 网络策略或 Baseline Admin 网络策略的名称。
	优先级	决定策略的评估和应用顺序，数值越低优先级越高。 注意：Baseline admin 网络策略无优先级。
目标 Pod	Namespace Selector	以键值对形式填写目标 Namespace 的标签，若不设置，则策略作用于当前集群所有 Namespace。设置后，策略仅作用于匹配这些选择器的 Namespace 中的 Pod。
	当前策略影响的目标 Pod 预览	点击 预览 查看该网络策略影响的目标 Pod。
	Pod Selector	以键值对形式填写目标 Pod 的标签，若不设置，则策略作用于当前 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。
入口流量	流量动作	指定如何处理目标 Pod 的入口流量。共有三种模式：**Allow**（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，允许流量由 Network Policy 处理，若

区域	参数		说明
			无 Network Policy，则由 Baseline Admin Network Policy 处理）。 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，它们之间为逻辑 或 关系。	Pod Selector	匹配集群中指定标签的 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 中选择具有指定标签的 Pod。 • 若未配置此项，默认允许集群中所有 Namespace 的所有 Pod 访问目标 Pod。
		Namespace Selector	匹配当前 Namespace 中指定标签的 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。
出口流量	规则 说明：规则中添加多个来源时，它们之间为逻辑 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	与入口流量参数类似，配置选项和行为相同。

通过 CLI 创建 AdminNetworkPolicy 或 BaselineAdminNetworkPolicy

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

其他资源

- 配置集群网络策略

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相互访问。可以通过添加单项目访问或**IP** 段访问规则实现通信。

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

目录

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. 进入平台管理。

2. 在左侧导航栏点击网络管理 > 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，实现当前集群内所有项目之间的网络隔离，其他资源不允许访问集群内任一项目（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，支持多条记录。 在源项目下拉框中选择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，支持多条记录。 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。

创建 BGP 对等体

节点建立连接以交换路由信息，不论是在不同的 AS 之间还是同一 AS 内部，通过 BGP 协议进行通信。

目录

术语

术语	说明
AS 编号	AS 指由同一技术管理机构管理的一组路由器，它们使用统一的路由策略。在 BGP 网络中，每个 AS 被分配一个唯一的 AS 编号以将其与其他 AS 区分开。AS 编号分为 2 字节 AS 编号和 4 字节 AS 编号。 • 2 字节 AS 编号的范围是 1~65535，其中 1~64511 是在 Internet 上注册的公共 AS 编号，类似于公共 IP 地址；64512~65535 是私有 AS 编号，类似于私有 IP 地址。 • 4 字节 AS 编号的范围是 1~4294967295。 支持 4 字节 AS 编号的设备可以与支持 2 字节 AS 编号的设备兼容。

先决条件

请联系管理员以启用相关功能。

示例 **BGPPeer** 自定义资源 **(CR)**

```yaml
# 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 平台管理。

2. 在左侧导航栏中，点击 网络管理 > **BGP** 对等体。

3. 点击 创建 **BGP** 对等体。

4. 请参考以下说明配置参数。

参数	描述
本地 **AS** 编号	BGP 连接节点所在 AS 的 AS 编号。

参数	描述
	注意: 如果没有特殊要求，建议使用 IBGP 配置，即本地 AS 编号应与对等体 AS 编号一致。
对等体 **AS** 编号	BGP 对等体所在 AS 的 AS 编号。
对等体 **IP**	BGP 对等体的 IP 地址，必须是一个有效的 IP 地址，可以建立 BGP 连接。
本地 **IP**	BGP 连接节点的 IP 地址。当 BGP 连接节点有多个 IP 时，选择指定的本地 IP 与对等体建立 BGP 连接。
对等体端口	BGP 对等体的端口号。
BGP 连接节点	建立 BGP 连接的节点。如果未配置此参数，则所有节点将建立 BGP 连接。
eBGP 多跳	允许在未直接连接的 BGP 路由器之间建立 BGP 会话。当启用此功能时，BGP 数据包的默认 TTL 值为 5，允许在多个中间网络设备之间建立 BGP 对等关系，使网络设计更加灵活。
RouterID	一个 32 位的数字值（通常以点分十进制格式表示，类似于 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、选择最佳路径以及故障排除网络问题。

5. 点击 创建。

通过 **CLI** 创建 **BGPPeer**

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

如何操作

为 ALB 部署高可用 VIP

方法 1：使用 LoadBalancer 类型的内部路
方法 2：使用外部负载均衡设备提供 VIP

软件数据中心负载均衡方案（Alp

先决条件
操作步骤
验证

准备 Kube-O

使用说明
术语解释
环境要求
配置示例

Underlay 和 Overlay 子网的自动

操作步骤

使用 OAuth Proxy 配合 ALB

Overview
Procedure
Result

创建 Gatewa

部署 MetalLB
设置 Pod 安全策

配置负载均衡器

前提条件
ALB2 自定义资源（CR）示例
通过 Web 控制台创建负载均衡器
通过 CLI 创建负载均衡器
通过 Web 控制台更新负载均衡器
通过 Web 控制台删除负载均衡器
通过 CLI 删除负载均衡器
配置监听端口（Frontend）
前提条件
Frontend 自定义资源（CR）示例
通过 Web 控制台创建监听端口（Frontend
通过 CLI 创建监听端口（Frontend）

将 IPv6 流量转发到集群内的 IPv4 地址

配置方法
结果验证

特殊场景部署建
负载均衡器使用

Calico 网络支持 WireGuard 加密

安装状态
术语
注意事项
先决条件

Kube-OVN

术语
注意事项

为 ALB 部署高可用 VIP

负载均衡器的高可用性需要一个 VIP。获取 VIP 的方式有两种。

目录

方法 1：使用 LoadBalancer 类型的内部路由提供 VIP

在创建负载均衡器时，启用 内部路由选项，系统会自动创建一个 LoadBalancer 类型的内部路由为负载均衡器提供 VIP。在使用之前，请确保当前集群支持 LoadBalancer 类型的内部路由。您可以使用平台内置的 LoadBalancer 内部路由实现，具体配置请参考 External Address Pool；如果 内部路由选项被禁用，则需要为负载均衡器配置访问地址。

方法 2：使用外部负载均衡设备提供 VIP

- 部署前，请与网络工程师确认负载均衡服务的 IP 地址（公网 IP、私网 IP、VIP）或域名。如果您希望使用域名作为外部流量访问负载均衡器的地址，则需要提前申请域名并配置域名解析。建议使用商业负载均衡设备提供 VIP；如果没有，则可以使用 纯软件数据中心 LB 解决方案（Alpha）。

- 根据业务场景，外部负载均衡器需要为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	描述
端口	• 对于全球集群，填写：11782。 • 对于业务集群，填写：1936。
协议	健康检查的协议类型，建议使用 TCP。
响应超时	接收健康检查响应所需的时间，建议配置为 2 秒。
检查间隔	健康检查的时间间隔，建议配置为 5 秒。
不健康阈值	确定后端服务器健康检查状态失败所需的连续失败次数，建议配置为 3 次。

软件数据中心负载均衡方案（Alpha）

通过创建一个高度可用的负载均衡器在集群外部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，以确保业务操作的稳定。它支持仅 IPv4、仅 IPv6 或 IPv4 和 IPv6 双栈的配置。

目录

先决条件

1. 准备两个或更多主机节点作为 LB。建议在 LB 节点上安装 Ubuntu 22.04 操作系统，以减少 LB 将流量转发到异常后端节点的时间。
2. 在所有外部 LB 主机节点上预安装以下软件（本章以两个外部 LB 主机节点为例）：
   - `ipvsadm`
   - `Docker (20.10.7)`
3. 确保每个主机的 Docker 服务在启动时启用，可以使用以下命令：`sudo systemctl enable docker.service`。
4. 确保每个主机节点的时钟已同步。

5. 准备 Keepalived 的镜像，用于启动外部 LB 服务；该平台已经包含此镜像。镜像地址格式为：`<image repository address>/tkestack/keepalived:<version suffix>`。版本后缀可能在不同版本间略有不同。您可以通过以下方式获取镜像仓库地址和版本后缀。本文档以 `build-harbor.alauda.cn/tkestack/keepalived:v3.16.0-beta.3.g598ce923` 为例。

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

操作步骤

注意：以下操作必须在每个外部 LB 主机节点上执行一次，并且主机节点的 `hostname` 不能重复。

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

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

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

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

3. 使用 `reboot` 命令重启。

4. 创建 Keepalived 配置文件的文件夹。

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

5. 根据以下文件中的注释修改配置项，并将其保存在 `/etc/keepalived/` 文件夹中，文件命名为 `alive.yaml`。

```
instances:
  - vip: # 可以配置多个 VIP
      vip: 192.168.128.118 # VIP 必须不同
      id: 20 # 每个 VIP 的 ID 必须唯一，选填
      interface: "eth0"
      check_interval: 1 # 选填，默认 1：执行检查脚本的间隔
      check_timeout: 3  # 选填，默认 3：检查脚本的超时时间
      name: "vip-1" # 此实例的标识符，仅能包含字母数字字符和连字符，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 会删除接口上的所有 IP https://github.com/osixia/docker-keepalived/issues/33
      kube_lock:
        kubecfgs: # kube-lock 使用的 kube-config 列表，Keepalived 中将按顺序尝试这些 kubecfgs 进行领导者选举
          - "/live/cfg/kubecfg/kubecfg01.conf"
          - "/live/cfg/kubecfg/kubecfg02.conf"
          - "/live/cfg/kubecfg/kubecfg03.conf"
    ipvs: # 选项 IPVS 的配置
      ips: [ "192.168.143.192", "192.168.138.100","192.168.129.100" ] # IPVS 后端，将 k8s 主节点 IP 改为 ALB 节点的节点 IP
      ports: # 为 VIP 上的每个端口配置健康检查逻辑
        - port: 80 # 虚拟服务器上的端口必须与真实服务器的端口匹配
          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，Keepalived 中将按顺序尝试这些 kubecfgs 进行领导者选举
          - "/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::19
2:168:129:100" ]
      ports:
        - port: 80
          virtual_server_config: |
            delay_loop 10
            lb_algo rr
            lb_kind NAT
            protocol TCP
          raw_check: |
            TCP_CHECK {
                connect_timeout 1
                connect_port 1936
            }
```

6. 在业务集群中执行以下命令检查配置文件中的证书到期日期，确保证书仍然有效。证书过期后，LB 功能将变得不可用，需联系平台管理员进行证书更新。

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

7. 从 Kubernetes 集群中的三个主节点复制 `/etc/kubernetes/admin.conf` 文件到外部 LB 节点的 `/etc/keepalived/kubecfg` 文件夹中，以索引命名，例如 `kubecfg01.conf`，并在这三个文件中将 `apiserver` 节点地址修改为 Kubernetes 集群的实际节点地址。

   注意：平台证书更新后，此步骤需重新执行，覆盖原始文件。

8. 检查证书的有效性。

   1. 从业务集群的主节点复制 `/usr/bin/kubectl` 到 LB 节点。

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

   3. 执行以下命令确认证书的有效性。

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

如果返回以下结果，则证书有效。

```
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/kee
palived:/live/cfg build-harbor.alauda.cn/tkestack/keepalived:v3.16.0-be
ta.3.g598ce923
```

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

验证

1. 运行命令 `ipvsadm -ln` 查看 IPVS 规则，您将会看到适用于业务集群 ALB 的 IPv4 和 IPv6 规则。

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

2. 关闭 VIP 所在的 LB 节点，测试 IPv4 和 IPv6 的 VIP 能否成功迁移到另一个节点，通常在 20 秒内。

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

```
curl 192.168.128.118

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

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

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

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

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

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

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

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

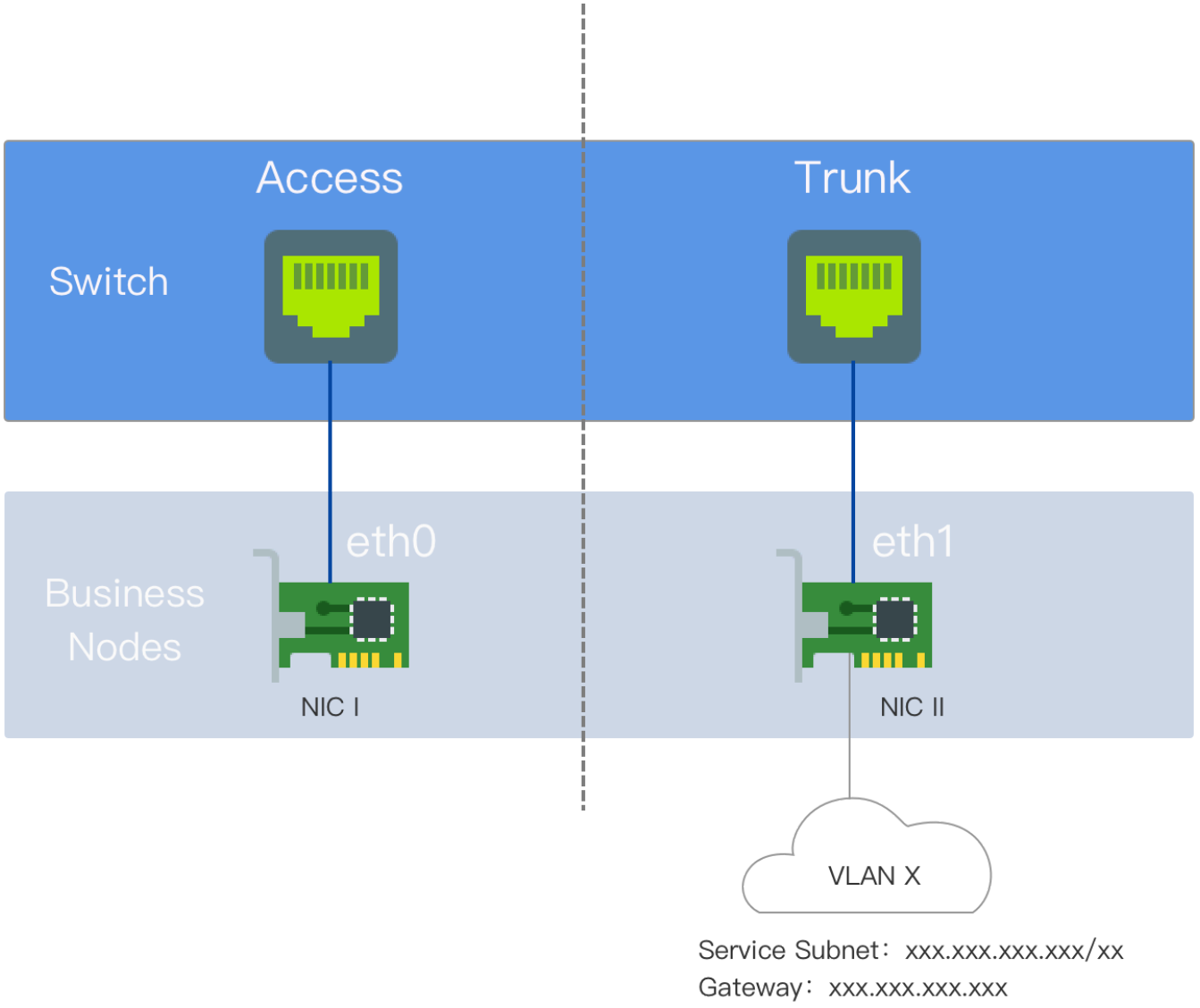
使用说明

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

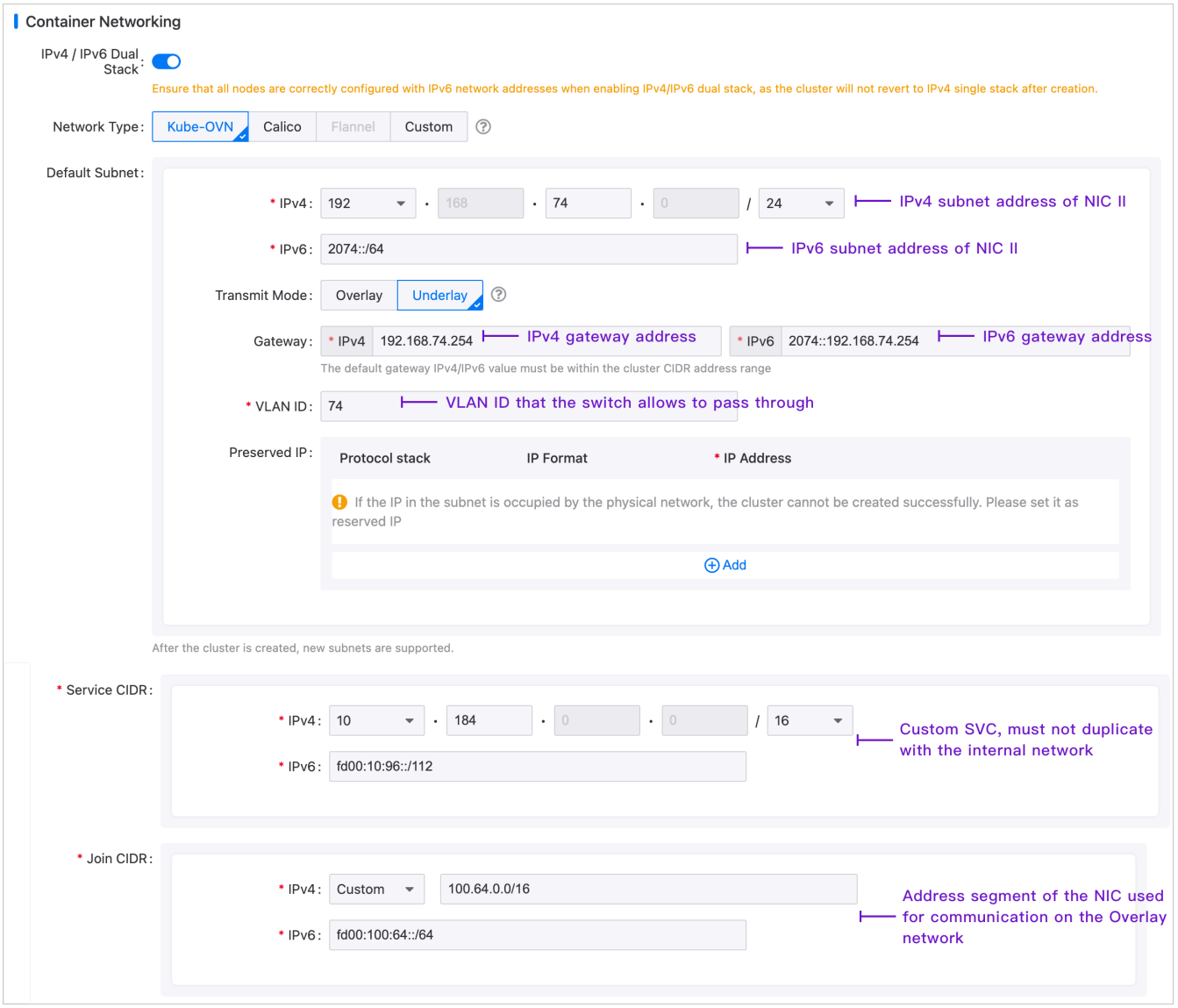
测试 NIC 二 是否能够与网关地址通信：

```
ip link add ens224.74 link ens224 type vlan id 74  // NIC 名称为 ens224, VL
AN 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

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 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

1. 进入 平台管理。
2. 在左侧导航栏点击 集群管理 > 资源管理。
3. 输入 **Subnet** 以筛选资源对象。
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。

使用 OAuth Proxy 配合 ALB

目录

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 s
earch repo|grep alb
helm install alb-operator alb/alauda-alb2
alb_ip=$(docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddr
ess}}{{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↗

> 注意此步骤中会获得 `$GITHUB_CLIENT_ID` 和 `$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_b64en
code(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/oa
uth2/auth"
    nginx.ingress.kubernetes.io/auth-signin: "https://auth.alb.echo.co
m/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 以及测试应用。
- 访问 alb.echo.com 后，会被重定向到 github 认证页面，认证通过后即可看到应用的输出。

创建 GatewayAPI Gateway

GatewayAPI 是 Kubernetes 的一个新 API，它提供了一种更灵活和可扩展的方式来管理入口流量。它允许您以更声明式的方式定义路由规则、流量策略和其他配置。 本文档提供了在 Alauda Container Platform Kubernetes 集群中创建 GatewayAPI Gateway 的分步指南。

前置要求

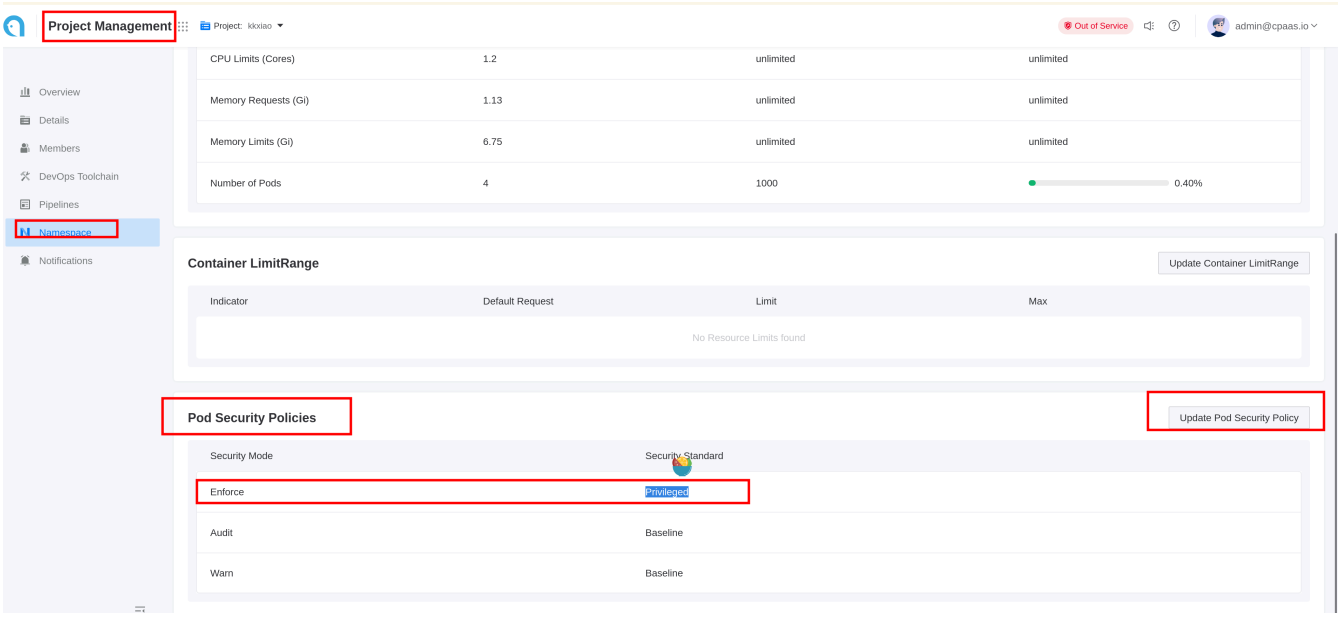
目录

部署 MetalLB

GatewayAPI Gateway 需要使用 MetalLB 来分配 IP 地址。请参考 创建 MetalLB 来部署 MetalLB。

设置 Pod 安全策略为特权模式

如果您希望部署 Gateway 的命名空间是通过 UI 创建的，则需要将其 Pod 安全策略（PSP）更新为特权模式。



操作步骤

1. 进入 平台管理。

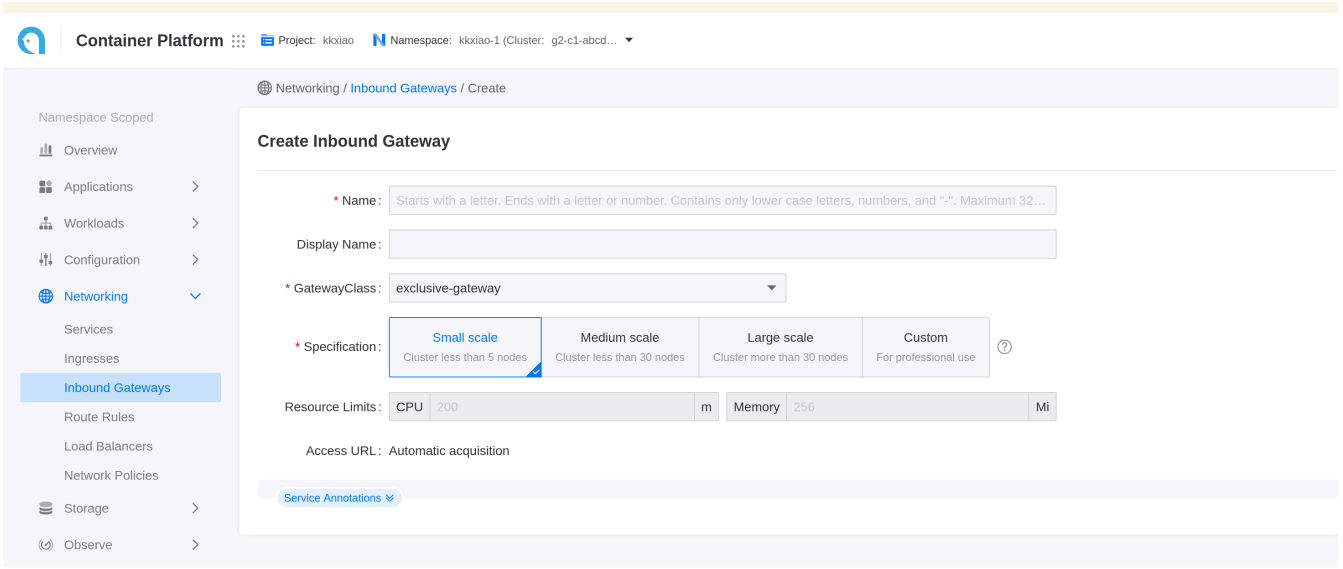
2. 在左侧导航栏中，点击 网络管理 > 入口网关。

3. 点击 创建入口网关。

4. 按照以下说明完成网络配置：

参数	描述
名称	Gateway 的名称。
GatewayClass	Alauda Container Platform 提供的内置 `exclusive-gateway`，由 ALB 支持。它将创建一个容器网络模式的 ALB，以实现 GatewayAPI Gateway 的规范。
规格	请根据业务需求合理设置规格。您也可以参考 如何合理分配 CPU 和内存资源 进行指导。

5. 点击 创建。创建过程可能需要一些时间，请耐心等待。


Container Platform
Project: kkxiao
Namespace: kkxiao-1 (Cluster: g2-c1-abcd...
Networking / Inbound Gateways / Create
Namespace Scoped
Overview
Applications
Workloads
Configuration
Networking
Services
Ingresses
Inbound Gateways
Route Rules
Load Balancers
Network Policies
Storage
Observe
Create Inbound Gateway
* Name:
Starts with a letter. Ends with a letter or number. Contains only lower case letters, numbers, and "-". Maximum 32...
Display Name:
* GatewayClass:
exclusive-gateway
* Specification:
Small scale
Cluster less than 5 nodes
Medium scale
Cluster less than 30 nodes
Large scale
Cluster more than 30 nodes
Custom
For professional use
Resource Limits:
CPU
200
m
Memory
256
Mi
Access URL:
Automatic acquisition
Service Annotations

配置负载均衡器

负载均衡器是一种将流量分发到容器实例的服务。通过利用负载均衡功能，它会自动分配访问流量到计算组件，并转发到这些组件的容器实例。负载均衡可以提升计算组件的容错能力，扩展这些组件的外部服务能力，并增强应用的可用性。

平台管理员可以为平台上的任意集群创建单点或高可用负载均衡器，并统一管理和分配负载均衡器资源。例如，可以将负载均衡分配给项目，确保只有具有相应项目权限的用户才能使用负载均衡。

本节相关概念说明请参考下表。

参数	说明
Load Balancer	一种将网络请求分发到集群中可用节点的软件或硬件设备。平台中使用的负载均衡器是七层（Layer 7）软件负载均衡器。
VIP	虚拟IP地址（Virtual IP Address），指不对应具体计算机或具体网卡的IP地址。当负载均衡器为高可用类型时，访问地址应为VIP。

目录

前提条件

Load Balancer 的高可用需要 VIP，详情请参考配置 VIP。

ALB2 自定义资源（CR）示例

```yaml
# test-alb.yaml
apiVersion: crd.alauda.io/v2beta1
kind: ALB2
metadata:
  name: alb-demo
  namespace: cpaas-system
  annotations:
    cpaas.io/display-name: ""
spec:
  address: 192.168.66.215
  config:
    vip: 1
      enableLbSvc: false
      lbSvcAnnotations: {}
    networkMode: host 2
    enablePortProject: false 3
    nodeSelector:
      cpu-model.node.kubevirt.io/Nehalem: "true"
    projects: 4
      - ALL_ALL
    replicas: 1
    resources: 5
      limits:
        cpu: 200m
        memory: 256Mi
      requests:
        cpu: 200m
        memory: 256Mi
  type: nginx
```

1. 当 `enableLbSvc` 为 true 时，会为负载均衡器的访问地址创建一个内部 LoadBalancer 类型服务。 `lbSvcAnnotations` 配置参考 LoadBalancer 类型服务注解。
2. 查看下方网络模式配置。
3. 查看下方资源分配方式。
4. 查看下方分配项目。
5. 查看下方规格说明。

通过 **Web** 控制台创建负载均衡器

1. 进入 平台管理。

2. 在左侧导航栏点击 网络管理 > 负载均衡器。

3. 点击 创建负载均衡器。

4. 按照以下说明完成网络配置。

参数	说明
网络模式	• **Host** 网络模式：单节点上只允许部署一个负载均衡器副本，多个服务共享一个 ALB，网络性能更优。 • 容器网络模式：单节点上可部署多个负载均衡器副本，满足每个服务独立 ALB 的需求，网络性能略逊。
服务及注解（**Alpha**）	• 服务：启用时会为负载均衡器访问地址创建一个内部 LoadBalancer 类型服务。使用前请确保当前集群支持 LoadBalancer 类型服务。可实现平台内置的 LoadBalancer 类型服务；禁用时需为负载均衡器配置外部地址池。 • 注解：用于声明内部 LoadBalancer 类型路由的配置或能力，具体请参考内部 LoadBalancer 类型服务注解。
访问地址	负载均衡的访问地址，即负载均衡器实例的服务地址。负载均衡器创建成功后，可通过该地址访问。 • Host 网络模式下，请根据实际情况填写，可为域名或 IP 地址（内网 IP、外网 IP、VIP）。 • 容器网络模式下，地址会自动获取。

5. 按照以下说明完成资源配置。

参数	说明
规格	请根据业务需求合理设置规格，也可参考如何合理分配 CPU 和内存资源进行参考。
部署类型	• 单点：负载均衡器容器组部署在单个节点，若机器故障可能导致负载均衡器不可用风险。 • 高可用：负载均衡器多个容器组部署在对应数量的节点上，通常为 3 个，满足大业务量负载均衡需求并具备应急灾备能力。
副本数	副本数即负载均衡器容器组数量。 提示：为保证负载均衡器高可用，建议副本数不少于 3 个。
节点标签	通过标签筛选节点部署负载均衡器。 提示： • 建议满足要求的节点数大于负载均衡器副本数。 • 同一标签键只能选择一个（若选择多个则无匹配主机）。
资源分配方式	• 实例：可为项目提供负载均衡器实例可监听的 1-65535 范围内任意端口。 • 端口（**Alpha**）：仅能分配指定范围内端口给项目使用，端口资源有限时可实现更细粒度资源控制。

分配项目	• 当资源分配方式为实例时，负载均衡器可分配给当前集群关联的所有项目或指定项目，分配项目中所有命名空间的所有 Pod 均可接收负载均衡器分发的请求。
	  • 所有项目：分配负载均衡器供当前集群关联的所有项目使用。
	  • 指定项目（**Alpha**）：点击指定项目下拉框，勾选项目名称左侧复选框，选择一个或多个项目，分配负载均衡器供指定项目使用。
	  提示：可在下拉框输入项目名称进行筛选。
	  • 不分配（**Alpha**）：暂不分配任何项目，负载均衡器创建后可通过更新项目操作更新已创建负载均衡器的分配项目参数。

参数	说明
	• 当资源分配方式为端口时，无需配置此项，创建负载均衡器后请手动分配端口信息。

6. 点击 创建，创建过程需要一定时间，请耐心等待。

通过 CLI 创建负载均衡器

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

通过 Web 控制台更新负载均衡器

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

1. 进入 平台管理。

2. 在左侧导航栏点击 网络管理 > 负载均衡器。

3. 点击⋮> 更新。

4. 根据需要更新网络和资源配置。

   - 请根据业务需求合理设置规格，也可参考相关如何合理分配 CPU 和内存资源。
   - 内部路由 仅支持从禁用状态更新为启用状态。

5. 点击 更新。

通过 Web 控制台删除负载均衡器

> **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 平台管理。
2. 在左侧导航栏点击 网络管理 > 负载均衡器。
3. 点击⋮> 删除，并确认。

通过 CLI 删除负载均衡器

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

配置监听端口（Frontend）

负载均衡器支持通过监听端口及对应协议接收客户端连接请求，协议包括 HTTPS、HTTP、gRPC、TCP 和 UDP。

前提条件

若需添加 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。

Frontend 自定义资源（CR）示例

```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:
  backendProtocol: "http"
  certificate_name: "" (3)
  port: 80
  protocol: http (4)
  serviceGroup: (5)
    services:
      - name: hello-world
        namespace: default
        port: 80
        weight: 100 (6)
```

1. 必填，表示该 `Frontend` 所属的 ALB 实例。
2. 格式为 `$alb_name-$port`。
3. 格式为 `$secret_ns/$secret_name`。
4. 该 `Frontend` 本身的协议。
   - `http|https|grpc|grpcs` 用于七层代理。
   - `tcp|udp` 用于四层代理。
5. 四层代理时，`serviceGroup` 必填；七层代理时，`serviceGroup` 可选。当请求到达时，ALB 会先尝试匹配与该 `Frontend` 关联的规则，只有请求未匹配任何规则时，才会转发到 `Frontend` 配置的默认 `serviceGroup`。
6. `weight` 配置适用于轮询和加权轮询调度算法。

> **NOTE**
>
> ALB 监听 ingress 并自动创建 `Frontend` 或 Rule。`source` 字段定义如下：
>
> 1. `spec.source.type` 当前仅支持 `ingress`。

2. `spec.source.name` 为 ingress 名称。
3. `spec.source.namespace` 为 ingress 命名空间。

通过 **Web** 控制台创建监听端口（**Frontend**）

1. 进入 容器平台。
2. 在左侧导航栏点击 网络 > 负载均衡。
3. 点击负载均衡器名称进入详情页。
4. 点击 添加监听端口。
5. 参考以下说明配置相关参数。

参数	说明
协议	支持的协议包括 HTTPS、HTTP、gRPC、TCP 和 UDP。选择 HTTPS 时必须添加证书；gRPC 协议添加证书为可选。 注意： • 选择 gRPC 协议时，后端协议默认为 gRPC，不支持会话保持。 • 为 gRPC 协议设置证书时，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。 • 若使用 Google GKE 集群，同一容器网络类型的负载均衡器不能同时存在 TCP 和 UDP 监听协议。
内部路由组	- 负载均衡算法为轮询（**RR**）时，流量按内部路由组顺序分发到内部路由端口。 **-** 负载均衡算法为加权轮询（**WRR**）时，权重值越高的内部路由被选中概率越大，流量按配置的**weight**计算概率分配到内部路由端口。 提示：概率计算为当前权重值与所有权重值之和的比值。
会话保持	始终将特定请求转发到上述内部路由组对应的后端服务。

参数	说明
	特定请求包括（任选其一）： • 源地址哈希：来自同一 IP 地址的所有请求。 注意：公有云环境中，源地址常变，可能导致同一客户端请求源 IP 不同，源地址哈希效果不佳。 • Cookie 键：携带指定 Cookie 的请求。 • Header 名称：携带指定 Header 的请求。
后端协议	用于转发流量至后端服务的协议。例如，转发到后端 Kubernetes 或 dex 服务时需选择 HTTPS 协议。

6. 点击 确定。

通过 **CLI** 创建监听端口（**Frontend**）

```
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，除默认内部路由组外，还可设置更多丰富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后端服务，规则匹配失败时再匹配上述内部路由组对应的后端服务。

相关操作

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

NOTE

若负载均衡器资源分配方式为端口，仅管理员可在平台管理视图中删除相关监听端口。

配置规则

为 HTTPS、HTTP 和 gRPC 协议的监听端口添加转发规则。负载均衡器将根据规则匹配后端服务。

NOTE

TCP 和 UDP 协议不支持添加转发规则。

Rule 自定义资源（**CR**）示例

```yaml
# alb-rule-demo.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: "" (3)
  certificate_name: "" (4)
  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"
  enableCORS: false
  priority: 4 5
  serviceGroup: 6
    services:
      - name: hello-world
        namespace: default
        port: 80
        weight: 100
```

1. 必填，表示该规则所属的 `Frontend`。
2. 必填，表示该规则所属的 ALB。
3. 同 `Frontend`。
4. 同 `Frontend`。
5. 数值越小优先级越高。
6. 同 `Frontend`。

dslx

dslx 是一种领域专用语言，用于描述匹配条件。

例如，以下规则匹配满足所有条件的请求：

- URL 以 /app-a 或 /app-b 开头
- 请求方法为 POST
- URL 参数 group 等于 vip
- Host 以 .app.com 结尾
- Header 中 LOCATION 值为 east-1 或 east-2
- 存在名为 uid 的 Cookie
- 源 IP 地址在 1.1.1.1 到 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"
```

通过 **Web** 控制台创建规则

1. 进入 容器平台。

2. 在左侧导航栏点击 网络 > 负载均衡。

3. 点击负载均衡器名称。

4. 点击监听端口名称。

5. 点击 添加规则。

6. 参考以下说明配置相关参数。

参数	说明
内部路由组	- 负载均衡算法为轮询（**RR**）时，访问流量按内部路由组顺序分发到内部路由端口。 **-** 负载均衡算法为加权轮询（**WRR**）时，内部路由权重值越高被轮询概率越大，访问流量按配置的**weight**计算概率分配到内部路由端口。 提示：概率计算为当前权重值与所有权重值之和的比值。

参数	说明
规则	负载均衡器匹配后端服务的条件，包括规则指标及其值。不同规则指标间关系为“且”。 • 域名：支持添加通配符域名和精确域名。同优先级下，若同时存在通配符和精确域名规则，优先匹配精确域名转发规则。 • URL：**RegEx** 对应以 `/` 开头的 URL 正则表达式；**StartsWith** 对应以 `/` 开头的 URL 前缀。 • IP：**Equal** 对应具体 IP 地址；**Range** 对应 IP 地址范围。 • Header：除输入 Header 键外，还需设置匹配规则。**Equal** 对应 Header 具体值；**Range** 对应 Header 值范围；**RegEx** 对应 Header 正则表达式。 • Cookie：除输入 Cookie 键外，还需设置匹配规则。**Equal** 对应 Cookie 具体值。 • URL 参数：匹配规则中，**Equal** 对应具体 URL 参数；**Range** 对应 URL 参数范围。 • 服务名：服务名指使用 gRPC 协议的服务名。使用 gRPC 协议时可配置此项，允许根据提供的服务名转发流量，例如 `/helloworld.Greeter`。
会话保持	始终将特定访问请求转发到上述内部路由组对应的后端服务。 特定访问请求包括（任选其一）： • 源地址哈希：来自同一 IP 地址的所有访问请求。 • Cookie 键：携带指定 Cookie 的访问请求。 • Header 名称：携带指定 Header 的访问请求。

参数	说明
URL 重写	将访问地址重写为平台后端服务地址。此功能需配置 URL 的 **StartsWith** 规则指标，且重写地址（rewrite-target）必须以 / 开头。 例如：设置域名为 *bar.example.com*，URL 起始路径为 `/`，启用 **URL** 重写 功能并设置重写地址为 */test*，访问 *bar.example.com* 时会将 URL 重写为 *bar.example.com/test*。
后端协议	用于转发访问流量至后端服务的协议。例如：转发至后端 Kubernetes 或 dex 服务时选择 HTTPS 协议。
重定向	将访问流量转发到新的重定向地址，而非内部路由组对应的后端服务。 例如：原访问地址页面升级或更新，为避免用户收到 404 或 503 错误页面，可通过配置将流量重定向至新地址。 • HTTP 状态码：浏览器重定向至新地址前呈现给用户的状态码。 • 重定向地址：输入相对地址（如 */index.html*）时，转发目标为 *负载均衡器地址/index.html*；输入绝对地址（如 *https://www.example.com*）时，转发目标为输入地址。
规则优先级	规则匹配优先级：共 10 级，1 最高，默认优先级为 5。 当多个规则同时满足时，选择优先级更高的规则应用；若优先级相同，系统使用默认匹配规则。
跨域资源共享（**CORS**）	CORS（跨域资源共享）是一种机制，利用额外的 HTTP 头部指示浏览器允许运行在一个源（域）上的 Web 应用访问来自不同源服务器的指定资源。 当资源请求另一个与自身域、协议或端口不同的服务器资源时，会发起跨域 HTTP 请求。
允许的源	用于指定允许访问的源。 • *****：允许任意源请求。 • 域名：允许当前域请求。

参数	说明
允许的头部	用于指定 CORS 中允许的 HTTP 请求头，避免不必要的预检请求，提高请求效率。示例条目如下： 注意：其他常用或自定义请求头未逐一列出，请根据实际情况填写。 • **Origin**：表示请求来源，即发送请求的域。 • **Authorization**：用于指定请求的授权信息，通常用于身份验证，如 Basic Authentication 或 Token。 • **Content-Type**：用于指定请求/响应的内容类型，如 application/json、application/x-www-form-urlencoded 等。 • **Accept**：用于指定客户端可接受的内容类型，通常用于客户端希望接收特定类型响应时。

7. 点击 添加。

通过 **CLI** 创建规则

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

日志与监控

结合可视化日志和监控数据，可快速定位和解决负载均衡器的问题或故障。

查看日志

1. 进入 平台管理。

2. 在左侧导航栏点击 网络管理 > 负载均衡器。

3. 点击 *负载均衡器名称*。

4. 在 日志 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 平台管理。

2. 在左侧导航栏点击 网络管理 > 负载均衡器。

3. 点击 *负载均衡器名称*。

4. 在 监控 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：当前节点负载均衡器 CPU 和内存的实时使用情况。
   - 吞吐量：负载均衡器实例的整体进出流量。

其他资源

- ALB 监控

如何合理分配 CPU 和内存资源

针对平台提出的 小型、中型、大型 和 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。

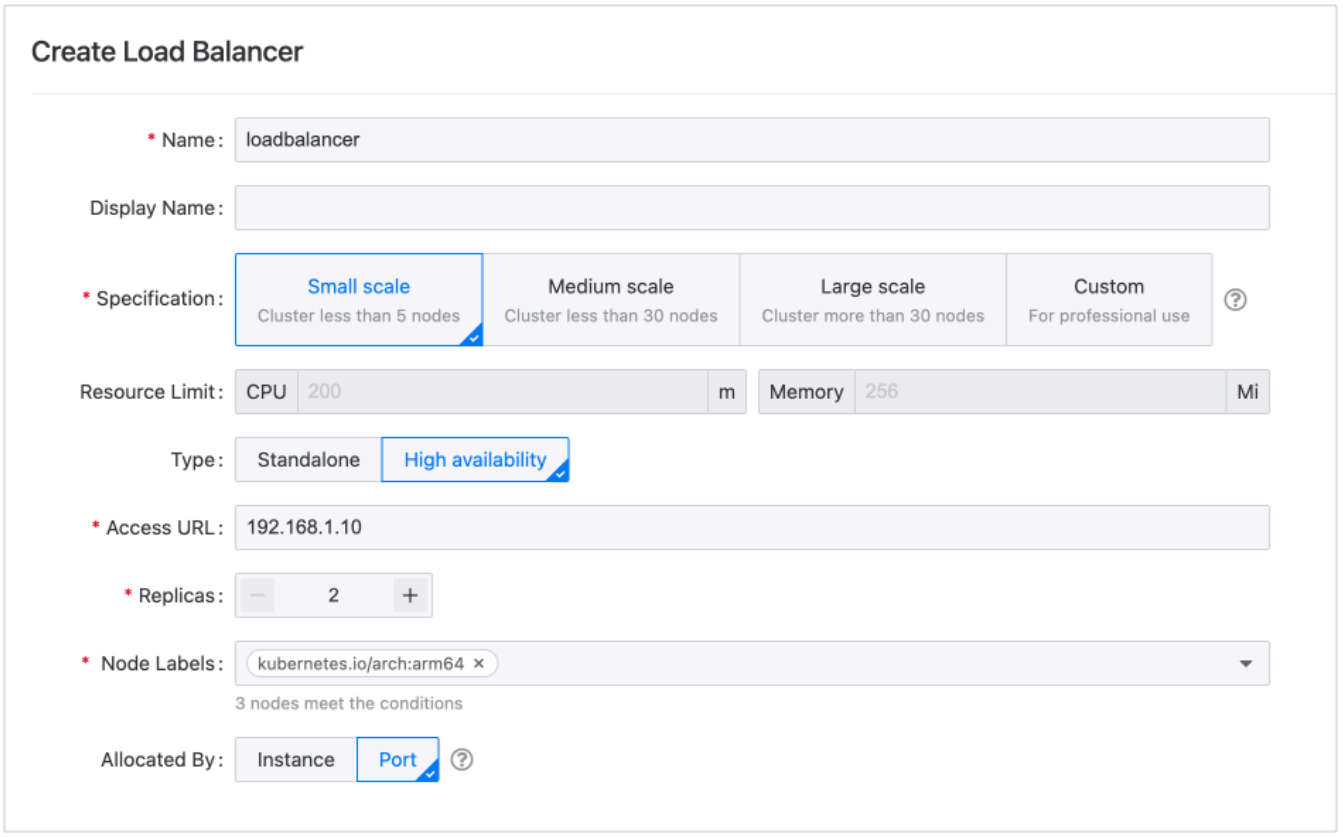
目录

小型生产环境

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

可以通过 端口 隔离方式对负载均衡器进行隔离，实现多个项目共享。

该规格在实验室环境下测得的峰值 QPS 约为每秒 300 次请求。

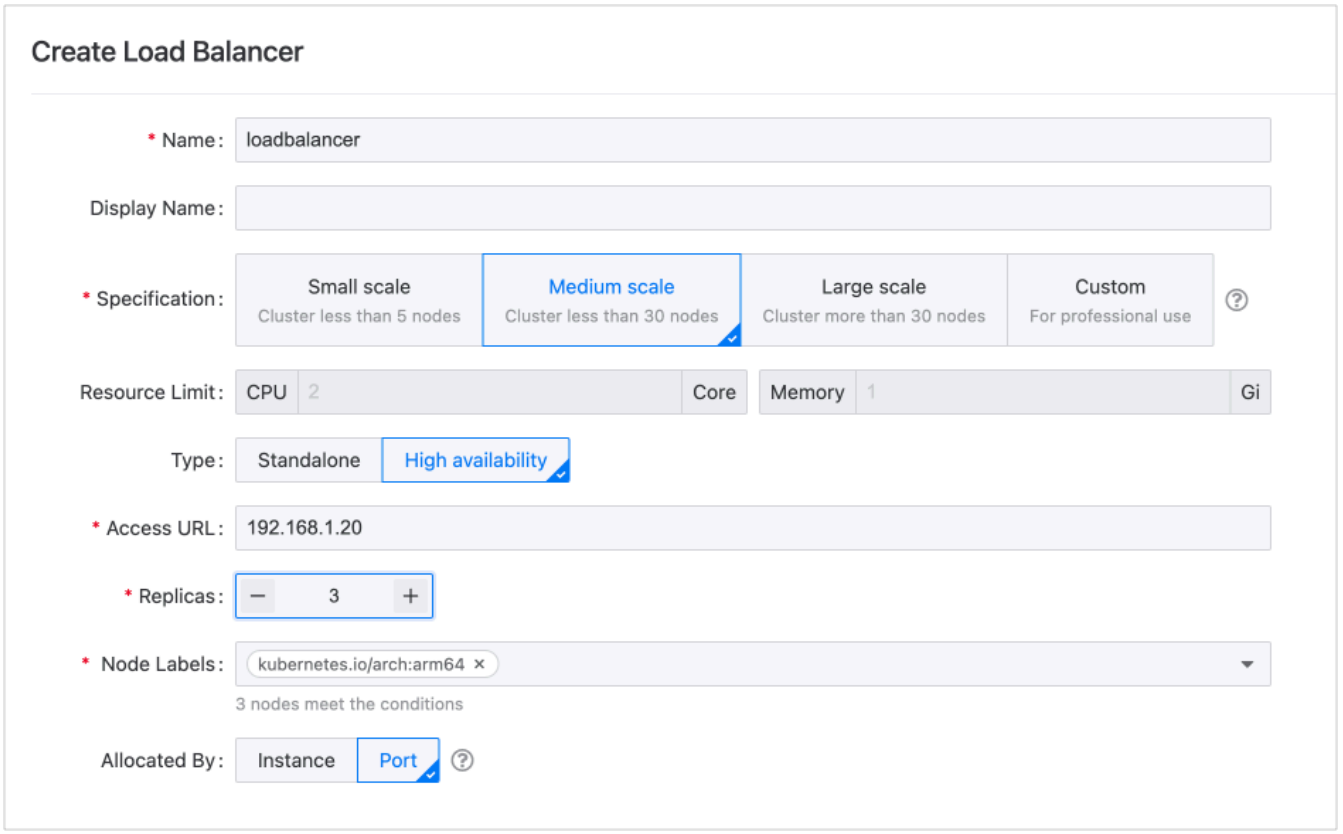


中型生产环境

当业务量达到一定规模，比如集群中节点不超过 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。

可以使用 端口 隔离或 实例 分配方式，实现多个项目共享负载均衡器。当然，也可以为核心项目新建专用负载均衡器。

该规格在实验室环境下测得的峰值 QPS 约为每秒 10,000 次请求。

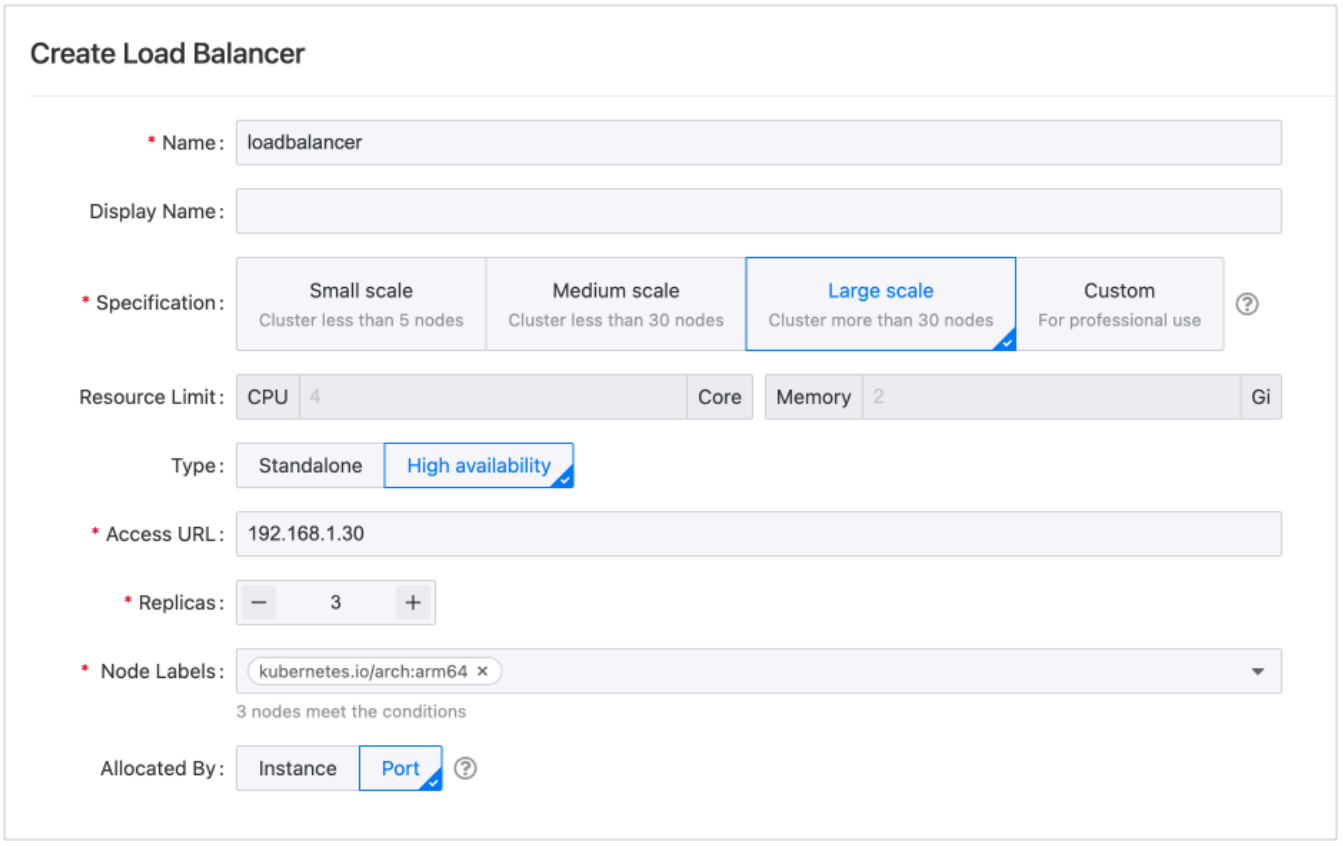


大型生产环境

对于更大规模的业务，比如集群中节点超过 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，实现多个项目共享。也可以为核心项目新建专用负载均衡器。

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

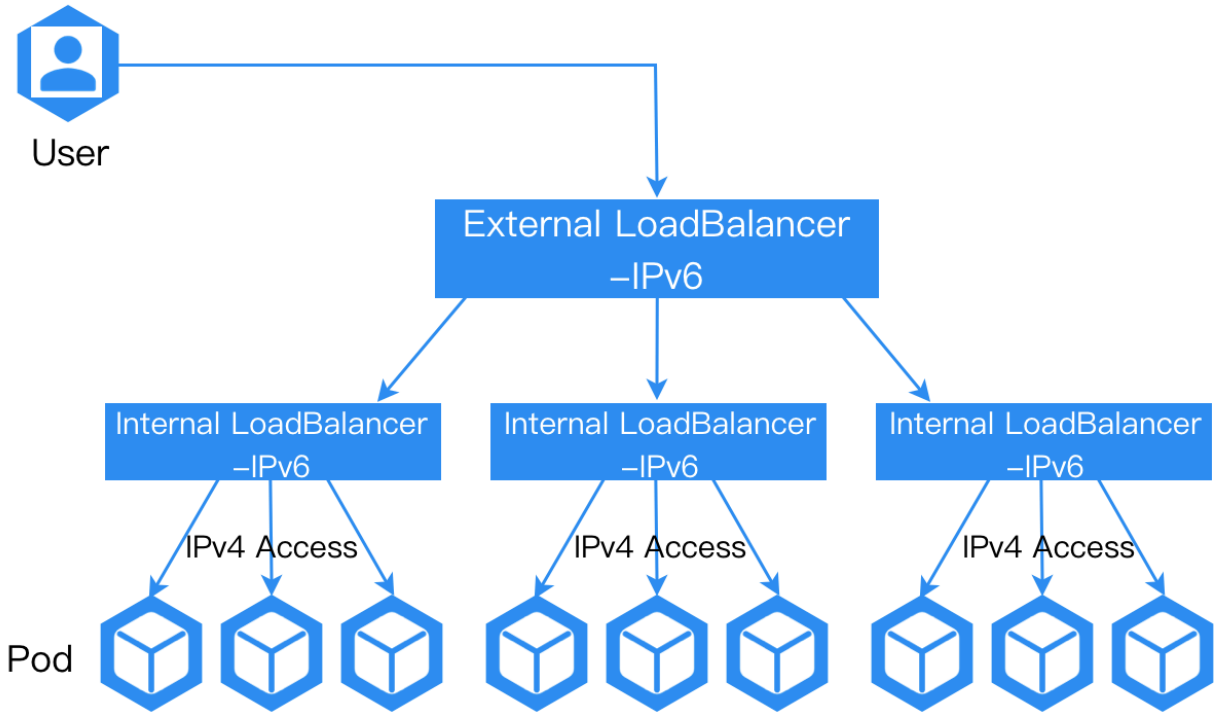
场景	部署建议
功能测试	建议部署 单实例 负载均衡器。
测试环境	如果测试环境符合上述 小型 或 中型 的定义，使用 单点 负载均衡器即可。负载均衡器 实例 可被 多个项目 共享。
核心应用	建议为核心应用专门使用特定的负载均衡器。
大规模数据传输	由于负载均衡器本身的内存消耗较小，即使是 大型 规格，预留 2Gi 内存也足够。但如果业务需要传输大规模数据，会导致内存消耗显著，应相应增加负载均衡器的内存分配。 建议在 自定义 规格场景中逐步扩展负载均衡器内存，密切监控内存使用情况，最终确定合理使用率下的可接受内存大小。

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对负载均衡器实例拥有完全权限，当某个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。	• 管理复杂度增加。平台管理员需主动规划和分配项目端口，并配置外部服务映射。 • 基于端口的分配成熟度较低，目前使用客户较少，功能仍需进一步完善。 • 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情况。

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

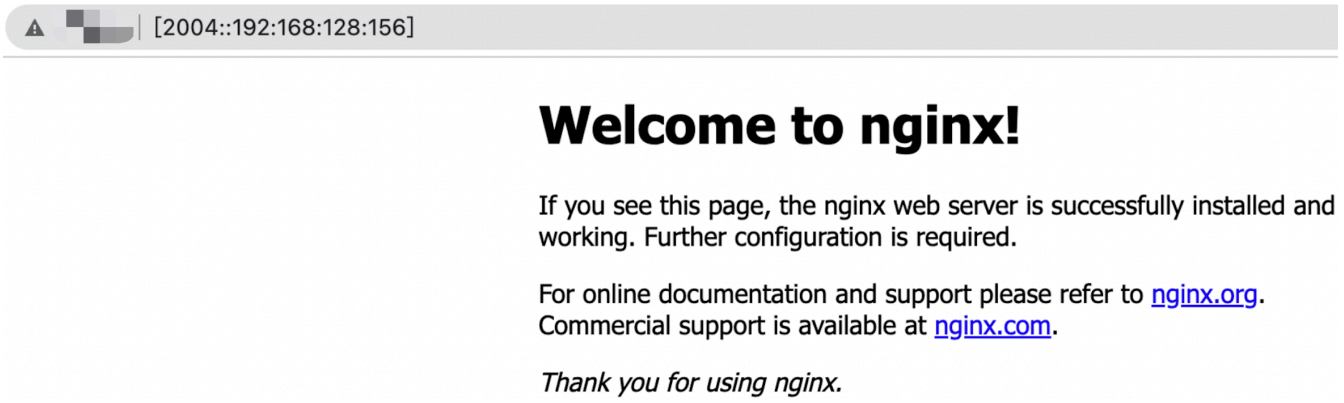
1. 配置负载均衡器所在节点的 IPv6 地址。

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。



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

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 '{"spe
    c":{"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。

       ```yaml
       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/xxxxxxxxxxxx/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

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/bas
h
```

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

        ```yaml
        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
      ```

ALB 监控

目录

术语

术语	描述
ALB	平台自研的七层负载均衡器。

操作步骤

1. 前往 平台管理。
2. 在左侧导航栏中，点击 操作中心 > 监控 > 监控仪表盘。
3. 点击页面顶部的 集群 切换到您想要监控的集群。

4. 点击页面右上角的 切换。

5. 您可以通过以下两种方式进入 **ALB** 状态 监控仪表盘：

   - 方法 1：点击 **container-platform** 卡片以展开监控目录，然后点击 **ALB** 状态 名称以进入监控仪表盘。如果需要，您可以将此监控仪表盘设置为主仪表盘。
   - 方法 2：在搜索框中输入关键字（例如，alb）进行搜索，然后点击 **ALB** 状态 名称以进入监控仪表盘。如果需要，您可以将此监控仪表盘设置为主仪表盘。

6. 通过仪表盘查看各种监控指标。

   - 选择要监控的命名空间：点击页面顶部的 命名空间 来选择要监控的命名空间，默认为全部，即监控所有命名空间。
   - 选择要监控的 **ALB**：点击页面顶部的 名称 来选择要监控的 ALB，默认为全部，即监控所有 ALB。

监控指标

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

说明：所有数据都是在 最近 **5** 分钟 收集的监控数据。

ALB 流量监控

监控指标	描述
活跃连接数	所选 ALB 上的活跃连接数。
每秒请求数	所选 ALB 每秒收到的请求总数。
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 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 每秒收到的请求数量，默认单位为 req/s。
请求 **BPS**（每秒字节数）	所选 ALB 上 Ingress/HTTPRoute/Rule 每秒收到的请求总大小。
响应 **BPS**（每秒字节数）	所选 ALB 上 Ingress/HTTPRoute/Rule 每秒发送的响应总大小。
错误率	所选 ALB 上 Ingress/HTTPRoute/Rule 处理请求时发生的错误百分比。
P50、**P90**、**P99**	所选 ALB 上请求的响应时间，具体为中位响应时间。这意味着 50%、90% 和 99% 的请求的响应时间小于或等于该值。 说明：P50、P90 和 P99 的原则是将收集的数据从小到大排序，并在 50%、90% 和 99% 的位置取值，因此，50%、90% 和 99% 的收集数据在此值以下。分位数有助于分析数据的分布情况及识别各种极端情况。
上游 **P50**、上游 **P90**、上游 **P99**	上游服务的请求响应时间。表示发送到上游服务的请求中，50%、90% 和 99% 的请求的响应时间小于或等于该值。

故障排除

如何解决 **ARM** 环境中的节点间通　查找错误原因

如何解决 ARM 环境中的节点间通信问题？

在使用较低版本的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
```

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。
```