开发人员

概览

概览

命名空间管理

应用生命周期管理

Kubernetes 工作负载管理

快速开始

Creating a simple application via image

Introduction

Important Notes

Prerequisites

Workflow Overview

Procedure

构建应用

Build application architecture

Introduction to the build application

Core components

核心概念

命名空间

创建应用

应用的操作

配置

实用指南

镜像

Overview of images

Understanding containers and images

Images

Image registry

Image repository

Image tags

Image IDs

Containers

实用指南

镜像仓库

介绍

原则与命名空间隔离

认证与授权

优势

应用场景

安装

升级

实用指南

Source to Image

概览

安装

升级

操作指南

实用指南

节点隔离策略

介绍

优势

适用场景

架构

操作指南

核心概念

权限

常见问题

常见问题

为什么导入命名空间时不应存在多个 ResourceQuota？

为什么导入命名空间时不应存在多个 LimitRange？

概览

灵雀云容器平台 提供统一的界面，通过 Web 控制台和 CLI（命令行界面）创建、编辑、删除和管理云原生应用。应用可以跨多个命名空间部署，并支持 RBAC 策略。

目录

- 命名空间管理
- 应用生命周期管理
  - 应用创建模式
  - 应用操作
  - 应用可观测性
- Kubernetes 工作负载管理

命名空间管理

命名空间为 Kubernetes 资源提供逻辑隔离。主要操作包括：

- 创建命名空间：定义资源配额和 Pod 安全准入策略。
- 导入命名空间：将现有 Kubernetes 命名空间导入到 灵雀云容器平台 中，实现与原生创建命名空间的完整平台能力一致性。

应用生命周期管理

灵雀云容器平台 支持端到端的生命周期管理，包括：

应用创建模式

在 灵雀云容器平台 中，应用可以通过多种方式创建。以下是一些常见方法：

- 从镜像创建：使用预构建的容器镜像创建自定义应用。此方法支持创建包含 `Deployments`、`Services`、`ConfigMaps` 及其他 Kubernetes 资源的完整应用。
- 从目录创建：灵雀云容器平台 提供应用目录，允许用户选择预定义的应用模板（Helm Charts 或 Operator Backed）进行创建。
- 从 YAML 创建：通过导入 YAML 文件，一步创建包含所有资源的自定义应用。
- 从代码创建：通过 Source to Image (S2I) 构建镜像。

应用操作

- 更新应用：更新应用的镜像版本、环境变量及其他配置，或导入现有 Kubernetes 资源进行集中管理。
- 导出应用：以 YAML、Kustomize 或 Helm Chart 格式导出应用，然后导入以在其他命名空间或集群中创建新的应用实例。
- 版本管理：支持自动或手动创建应用版本，出现问题时可一键回滚到指定版本，实现快速恢复。
- 删除应用：删除应用时，同时删除应用本身及其直接包含的所有 Kubernetes 资源。此外，此操作会断开应用与未直接包含在其定义中的其他 Kubernetes 资源的关联。

应用可观测性

为持续运营管理，平台提供日志、事件、监控等功能。

- 日志：支持查看当前运行 Pod 的实时日志，也提供容器重启前的日志。
- 事件：支持查看命名空间内所有资源的事件信息。
- 监控仪表盘：提供命名空间级监控仪表盘，包括专门针对应用、工作负载和 Pod 的视图，同时支持自定义监控仪表盘以满足特定运营需求。

Kubernetes 工作负载管理

支持核心工作负载类型：

- Deployments：管理无状态应用，支持滚动更新。
- StatefulSets：运行具有稳定网络 ID 的有状态应用。
- DaemonSets：部署节点级服务（例如日志采集器）。
- CronJobs：调度带重试策略的批处理作业。

快速开始

Creating a simple application via image

Introduction

Important Notes

Prerequisites

Workflow Overview

Procedure

Creating a simple application via image

> 本技术指南演示如何使用 Kubernetes 原生方法，在 灵雀云容器平台 中高效创建、管理和访问容器化应用。

目录

- Introduction
  - Use Cases
  - Time Commitment
- Important Notes
- Prerequisites
- Workflow Overview
- Procedure
  - Create namespace
  - Configure Image Repository
    - 方法一：通过工具链集成注册表
    - 方法二：外部注册表服务
  - Create application via Deployment
  - Expose Service via NodePort
  - Validate Application Accessibility

Introduction

Use Cases

- 新用户了解 Kubernetes 平台上基础应用创建流程
- 展示核心平台能力的实操演练，包括：
  - 项目/命名空间编排
  - 部署创建
  - 服务暴露模式
  - 应用可访问性验证

Time Commitment

预计完成时间：10-15 分钟

Important Notes

- 本技术指南聚焦关键参数，详尽配置请参考完整文档
- 所需权限：
  - 项目/命名空间创建
  - 镜像仓库接入
  - 工作负载部署

Prerequisites

- 具备 Kubernetes 架构及 灵雀云容器平台 平台基础概念
- 已按照平台搭建流程预配置项目

Workflow Overview

No.	Operation	Description
1	Create Namespace	建立资源隔离边界
2	Configure Image Repository	配置容器镜像来源
3	Create application via Deployment	创建 Deployment 工作负载
4	Expose Service via NodePort	配置 NodePort 类型服务
5	Validate Application Accessibility	测试访问端点连通性

Procedure

Create namespace

> 命名空间提供逻辑隔离，用于资源分组和配额管理。

Prerequisites

- 拥有创建、更新和删除命名空间权限（如管理员或项目管理员角色）
- kubectl 已配置集群访问权限

Creation Process

1. 登录后，进入 项目管理 > 命名空间
2. 选择 创建命名空间
3. 配置关键参数：

** 参数 **	说明
Cluster	目标集群，项目关联的集群之一
Namespace	唯一标识（自动加项目名前缀）

4. 默认资源限制下完成创建

Configure Image Repository

灵雀云容器平台 支持多种镜像来源策略：

方法一：通过工具链集成注册表

1. 进入 管理员 > 工具链 > 集成
2. 新建集成：

参数	要求
Name	唯一集成标识
API Endpoint	注册表服务地址（HTTP/HTTPS）
Secret	预先存在或新建的凭证

3. 将注册表分配至目标平台项目

方法二：外部注册表服务

- 使用公开可访问的注册表 URL
- 示例：`nginx:latest`

注册表访问与认证

集群成功拉取镜像需满足两点：

- 网络可达性：集群节点需具备访问容器注册表端点的出站网络权限。若连接失败，可能出现 `ImagePullBackOff` 或网络相关错误。
- 认证：若注册表需认证，需创建 `ImagePullSecret` 并在 Pod 或 Deployment 中引用。详细步骤见镜像拉取凭证文档。

Create application via Deployment

> Deployment 提供 Pod 副本集的声明式更新。

Creation Process

1. 在 容器平台视图中：
   - 使用命名空间选择器选择目标隔离边界
2. 进入 工作负载 > **Deployments**
3. 点击 创建 **Deployment**
4. 指定镜像来源：
   - 选择集成注册表 *或*
   - 输入外部镜像地址（如 `nginx:latest`）
5. 配置工作负载身份并启动

Management Operations

- 监控副本状态
- 查看事件和日志
- 检查 YAML 清单
- 分析资源指标和告警

Expose Service via NodePort

服务使 Pod 组具备网络访问能力。

Creation Process

1. 进入 网络 > 服务
2. 点击 创建服务，参数如下：

参数	值
Type	NodePort
Selector	目标 Deployment 名称

参数	值
Port Mapping	服务端口:容器端口 （如 8080:80 ）

3. 确认创建。

关键点

- 集群可见虚拟 IP
- NodePort 分配范围（30000-32767）

内部路由为工作负载提供统一 IP 或主机端口访问，实现服务发现。

1. 点击 网络 > 服务
2. 点击 创建服务
3. 根据下表配置 详情，其余参数保持默认。

参数	说明
Name	输入服务名称
Type	`NodePort`
Workload Name	选择之前创建的 `Deployment`
Port	服务端口：服务在集群内暴露的端口号，即 Port，例如 `8080`。 容器端口：服务端口映射的目标端口号（或名称），即 targetPort，例如 `80`。

4. 点击 创建，服务创建成功。

Validate Application Accessibility

Verification Method

1. 获取暴露端点信息：

   - 节点 **IP**：工作节点公网地址
   - **NodePort**：分配的外部端口

2. 构造访问 URL：`http://<Node_IP>:<NodePort>`

3. 预期结果：显示 Nginx 欢迎页面

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

构建应用

Build application architecture

Build application architecture

Introduction to the build application

Core components

核心概念

应用类型

Custom Applications

理解自定义应用

工作负载类型

理解参数

Overview

Core Concepts

使用场景和示例

CLI 示例及实际用法

最佳实践

常见问题排查

高级使用模式

理解环境变量

Overview

理解 Startup Commands

Overview

Core Concepts

使用场景和示例

CLI 示例与实际使用

最佳实践

资源单位说明

高级使用模式

命名空间

创建命名空间

了解命名空间

通过 Web 控制台创建命名空间

通过 **ac** CLI 在项目中创建命名空间

通过 **kubectl** 创建命名空间

导入 Namespace

Overview

Use Cases

Prerequisites

Procedure

资源配额

了解资源请求与

配额

硬件加速器资源

UID/GID 分配

启用 UID/GID 分配

验证 UID/GID 分配

更新命名空间

更新配额

更新容器 LimitRanges

更新 Pod Security Admission

删除/移除命名空间

删除命名空间

移除命名空间

创建应用

Creating applications from Imag

前提条件

操作步骤 1 - 工作负载

操作步骤 2 - 服务

操作步骤 3 - Ingress

应用管理操作

参考信息

Creating applications from Cha

注意事项

前提条件

操作步骤

状态分析参考

通过 YAML

注意事项

前提条件

操作步骤

Creating ap

Creating applications from Operator Backed

Creating applications by using CLI

Prerequisites

Procedure

Example

Reference

应用的操作与维护

Application Rollout

状态说明

原生应用

配置 HPA

了解水平 Pod 自

od 自

启动和停止原生应用

启动原生应用

停止原生应用

配置 VerticalPodAutoscaler (VPA)

了解 VerticalPodAutoscalers

nHI

Horiz

更新原生应用

导入资源

移除/批量移除资源

Horiz

...说明

导出应用

导出 Helm Chart

导出 YAML 到本地

导出 YAML 到代码仓库（Alpha）

更新和删除 Chart 应用

重要说明

前提条件

状态分析说明

应用版本管理

创建版本快照

回滚到历史版本

处理资源耗尽错误

Overview

配置驱逐策略

在节点配置中创建驱逐策略

驱逐信号

驱逐阈值

配置可调度资源

防止节点状态振荡

回收节点级资源

Pod 驱逐

服务质量与内存杀手（OOM Killer）

调度器与资源耗尽状态

示例场景

推荐实践

健康检查

理解健康检查

YAML 文件示例

通过 Web 控制台

探针失败排查

计算组件

Deployments

Understanding Deployments

DaemonSets

理解守护进程集

StatefulSets

理解 StatefulSe

Creating Deployments

Managing Deployments

使用 CLI 进行故障排查

创建守护进程集

管理守护进程集

创建 StatefulSe

管理 StatefulSe

Pods

理解 Pods

YAML 文件示例

使用 CLI 管理 Pod

使用 Web 控制台管理 Pod

示例

Containers

理解 Containers

理解 Ephemera

与 Containers 3

使用 Helm charts

使用 Helm charts

1. 了解 Helm

2 通过 CLI 以 Application 方式部署 Helm Charts

3 通过 UI 以 Application 方式部署 Helm Charts

配置

Configuring ConfigMap

Configuring Secrets

Understanding Config Maps
Config Map 限制
示例 ConfigMap
通过 Web 控制台创建 ConfigMap
通过 CLI 创建 ConfigMap
操作
通过 CLI 查看、编辑和删除
Pod 中使用 ConfigMap 的方式
ConfigMap 与 Secret 对比

理解 Secrets
创建 Opaque 类型的 Secret
创建容器镜像仓库类型的 Secret
创建 Basic Auth 类型的 Secret
创建 SSH-Auth 类型的 Secret
创建 TLS 类型的 Secret
通过 Web 控制台创建 Secret
在 Pod 中如何使用 Secret
后续操作
操作

应用可观测

监控面板

前提条件
命名空间级别监控面板
工作负载级别监控

Logs

Procedure

Events

操作步骤
事件记录解读

实用指南

设置定时任务触发规则

时间转换
编写 Crontab 表达式

添加 ImagePullSecrets 到 Servic

创建 ImagePullSecret
将 ImagePullSecret 添加到 ServiceAccour
验证新建 Pod 是否设置了 imagePullSecre

Service Ser

Overview
Features
Configuration a
User Guide (Fo

Build application architecture

目录

Introduction to the build application
Core components
Archon
Metis
Captain controller manager
Icarus

Introduction to the build application

Alauda Container Platform 是一个用于开发和运行容器化应用的平台。它旨在使应用及其支持的数据中心能够从少量机器和应用扩展到数千台机器，服务数百万客户。

基于 Kubernetes 构建，Alauda Container Platform 利用同样强大的技术，支持大规模电信、流媒体视频、游戏、银行及其他关键应用。这一基础使您能够将容器化应用扩展到混合环境——从本地基础设施到多云部署。

Core components

Archon

提供用于应用和资源管理操作的高级 API。作为控制平面组件，`Archon` 仅运行在 `global` 集群上，作为集群范围操作的中央管理接口。其 API 层支持对整个平台的应用、命名空间和基础设施资源进行声明式配置。

Metis

作为 `business clusters` 中的多功能控制器，提供关键的集群级操作：

- **Webhook** 管理：用于资源校验的 Admission webhook，包括资源配额和资源标签策略的执行。
- 状态同步：通过以下方式维护分布式组件间的一致性：
  - `Helm chart` 应用状态的调和
  - `Project quota` 的同步
  - `Application` 状态更新（写入 `Application.status`）

Captain controller manager

作为专门在 `global cluster` 上运行的 `Helm chart` 应用生命周期管理控制器，其职责包括：

- **Chart** 安装：协调跨集群的 `Helm chart` 部署
- 版本管理：处理 `Helm chart` 发布的无缝升级和回滚
- 卸载：彻底移除 `Helm chart` 应用及相关资源
- 发布跟踪：维护所有已部署 `Helm chart` 发布的状态和历史

Icarus

提供 `Container Platform` 的集中式基于 Web 的管理界面。作为展示层组件，`Icarus`：

- 提供全面的监控面板视图，用于集群健康状态监控
- 支持基于 GUI 的应用部署和管理流程
- 实现基于 **Kubernetes RBAC** 的多租户管理：
  - 通过命名空间隔离区分租户账户

  - 管理每个租户的资源访问权限
  - 提供租户特定的视图隔离
- 仅运行在 `global cluster` 上，作为多集群操作的统一控制点

Alauda Container Platform

核心概念

应用类型

Custom Applications

理解自定义应用

工作负载类型

理解参数

Overview

Core Concepts

使用场景和示例

CLI 示例及实际用法

最佳实践

常见问题排查

高级使用模式

理解环境变量

理解 Startup Commands

Overview

Core Concepts

使用场景和示例

CLI 示例与实际使用

最佳实践

高级使用模式

资源单位说明

应用类型

在平台的 **Container Platform > Applications** 中，可以创建以下类型的应用：

- 自定义应用：自定义应用表示由一个或多个相互关联的计算组件（如 Deployment 或 StatefulSet 等 Workloads）、内部网络配置（Services）及其他原生 Kubernetes 资源组成的完整业务应用。此类应用提供灵活的创建方式，支持直接通过 UI 编辑、YAML 编排及模板化部署，适用于开发、测试及生产环境。有关此应用类型的详细信息，请参阅 Custom Application。不同类型的原生应用可以通过以下方式创建：
  - 从镜像创建：使用现有容器镜像快速创建应用。
  - 从 YAML 创建：使用 YAML 配置文件创建应用。
  - 从代码创建：使用源代码创建应用。
- **Helm Chart** 应用：Helm Chart 应用允许您部署和管理以 Helm Chart 打包的应用。Helm Chart 是预配置的 Kubernetes 资源包，可以作为单个单元进行部署，简化复杂应用的安装和管理。有关此应用类型的详细信息，请参阅 Helm Chart Application
- **Operator** 支持的应用：Operator 支持的应用利用 Kubernetes Operator 的能力，实现复杂应用的生命周期自动化管理。通过部署由 Operator 支持的应用，您可以享受自动化的部署、扩缩容、升级和维护，因为 Operator 作为针对特定应用的智能控制器。有关此应用类型的详细信息，请参阅 Operator Backed Application。

Custom Applications

目录

理解自定义应用
核心能力
设计价值
自定义应用 CRD 架构设计
Application CRD 定义
ApplicationHistory 定义

理解自定义应用

自定义应用是一种基于原生 Kubernetes 资源（如 Deployment、Service、ConfigMap）构建的应用范式，严格遵循 Kubernetes 声明式 API 设计原则。用户可以通过标准 YAML 文件或直接调用 Kubernetes API 来定义和部署应用，实现对应用生命周期的细粒度控制。由镜像、代码和 YAML 等来源创建的应用均归类为 Alauda Container Platform 中的自定义应用。其设计核心在于平衡灵活性与标准化，适用于需要深度定制化管理的场景。

核心能力

1. 声明式 **API** 驱动管理

- 通过 Application CRD 将分散的资源（如 Deployment、Service、Ingress）聚合为逻辑应用单元，实现原子操作。

2. 应用级抽象与状态聚合

- 屏蔽底层资源细节（如 Pod 副本状态），开发者可直接通过 Application 资源监控整体应用健康状况（如就绪端点比例、版本一致性）。
- 支持跨组件依赖声明（如数据库服务必须先于应用服务启动），确保资源初始化顺序与协调。

3. 全生命周期治理

- 版本控制：跟踪历史配置，实现一键回滚至任意稳定状态。
- 依赖解析：自动识别并管理组件间版本兼容性（如 Service API 版本与 Ingress 控制器匹配）。

4. 增强的可观测性

- 聚合所有关联资源的状态指标（如 Deployment 可用副本数、Service 流量负载），通过统一监控面板提供全局视图。

设计价值

维度	价值主张
复杂度管理	将分散资源（如 Deployment、Service）封装为单一逻辑实体，降低认知和运维负担。
标准化	通过 Application CRD 统一应用描述标准，消除因 YAML 分散带来的管理混乱。
生态兼容性	无缝集成原生工具链（如 kubectl、Kubernetes Dashboard），支持 Helm Chart 扩展。
DevOps 效率	通过 GitOps 流水线（如 Argo CD）实现声明式交付，加速 CI/CD 自动化。

自定义应用 CRD 架构设计

自定义应用模块定义了两个核心 CRD 资源，构成应用管理的原子抽象单元：

维度	价值主张
Application	描述逻辑应用单元的元数据和组件拓扑，将 Deployment/Service 等资源聚合为单一实体。
ApplicationHistory	记录所有应用生命周期操作（创建/更新/回滚/删除）为版本化快照，与 Application CRD 紧密耦合，实现端到端变更可追溯。

Application CRD 定义

Application CRD 使用 `spec.componentKinds` 字段声明 Kubernetes 资源类型（如 Deployment、Service），支持跨资源生命周期管理。

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: applications.app.k8s.io
spec:
  group: app.k8s.io
  names:
    kind: Application
    listKind: ApplicationList
    plural: applications
    singular: application
  scope: Namespaced
  subresources:
    status: {}
  validation:
    openAPIV3Schema:
      properties:
        apiVersion:
          description: 'APIVersion defines the versioned schema of this r
epresentation
            of an object. Servers should convert recognized schemas to th
e latest
            internal value, and may reject unrecognized values. More inf
o: https://github.com/kubernetes/community/blob/master/contributors/deve
l/sig-architecture/api-conventions.md#resources'
          type: string
        kind:
          description: 'Kind is a string value representing the REST reso
urce this
            object represents. Servers may infer this from the endpoint t
he client
            submits requests to. Cannot be updated. In CamelCase. More in
fo: https://github.com/kubernetes/community/blob/master/contributors/deve
l/sig-architecture/api-conventions.md#types-kinds'
          type: string
        metadata:
          description: 'Metadata is a object value representing the metad
ata of the kubernetes resource.
            More info: https://github.com/kubernetes/community/blob/maste
r/contributors/devel/sig-architecture/api-conventions.md#metadata'
          type: object
        spec:
          properties:
```

```
            assemblyPhase:
              description: |
                The installer can set this field to indicate that the application's components
                are still being deployed ("Pending") or all are deployed already ("Succeeded"). When the
                application cannot be successfully assembled, the installer can set this field to "Failed".'
              type: string
            componentKinds:
              description: |
                This array of GroupKinds is used to indicate the types of resources that the
                application is composed of. As an example an Application that has a service and a deployment
                would set this field to [{"group":"core","kind": "Service"},{"group":"apps","kind":"Deployment"}]
              items:
                description: 'The item of the GroupKinds, with a structure like \"{"group":"core","kind": "Service"}\"'
                type: object
              type: array
            descriptor:
              properties:
                description:
                  description: 'A short, human readable textual description of the Application.'
                  type: string
                icons:
                  description: 'A list of icons for an application. Icon information includes the source, size, and mime type.'
                  items:
                    properties:
                      size:
                        description: 'The size of the icon.'
                        type: string
                      src:
                        description: 'The source of the icon.'
                        type: string
                      type:
                        description: 'The mime type of the icon.'
                        type: string
                    required:
                    - src
```

```
                  type: object
                type: array
              keywords:
                description: 'A list of keywords that identify the appl
ication.'
                items:
                  type: string
                type: array
              links:
                description: 'Links are a list of descriptive URLs inte
nded to be used to surface additional documentation, dashboards, etc.'
                items:
                  properties:
                    description:
                      description: 'The description of the link.'
                      type: string
                    url:
                      description: 'The url of the link.'
                      type: string
                  type: object
                type: array
              maintainers:
                description: 'A list of the maintainers of the Applicat
ion. Each maintainer has a
                  name, email, and URL. This field is meant for the dis
tributors of the Application
                  to indicate their identity and contact information.'
                items:
                  properties:
                    email:
                      description: 'The email of the maintainer.'
                      type: string
                    name:
                      description: 'The name of the maintainer.'
                      type: string
                    url:
                      description: 'The url to contact the maintainer.'
                      type: string
                  type: object
                type: array
              notes:
                description: 'Notes contain human readable snippets int
ended as a quick start
                  for the users of the Application. They may be plain t
```

```
ext or CommonMark markdown.'
                  type: string
                owners:
                  items:
                    properties:
                      email:
                        description: 'The email of the owner.'
                        type: string
                      name:
                        description: 'The name of the owner.'
                        type: string
                      url:
                        description: 'The url to contact the owner.'
                        type: string
                    type: object
                  type: array
                type:
                  description: 'The type of the application (e.g. WordPre
ss, MySQL, Cassandra).
                    You can have many applications of different names in
the same namespace.
                    They type field is used to indicate that they are all
the same type of application.'
                  type: string
                version:
                  description: 'A version indicator for the application
(e.g. 5.7 for MySQL version 5.7).'
                  type: string
              type: object
            info:
              description: 'Info contains human readable key-value pairs
for the Application.'
              items:
                properties:
                  name:
                    description: 'The name of the information.'
                    type: string
                  type:
                    description: 'The type of the information.'
                    type: string
                  value:
                    description: 'The value of the information.'
                    type: string
                  valueFrom:
```

```
                    description: 'The value reference from other resourc
e.'
                    properties:
                      configMapKeyRef:
                        description: 'The config map key reference.'
                        properties:
                          key:
                            type: string
                        type: object
                      ingressRef:
                        description: 'The ingress reference.'
                        properties:
                          host:
                            description: 'The host of the ingress referen
ce.'
                            type: string
                          path:
                            description: 'The path of the ingress referen
ce.'
                            type: string
                        type: object
                      secretKeyRef:
                        description: 'The secret key reference.'
                        properties:
                          key:
                            type: string
                        type: object
                      serviceRef:
                        description: 'The service reference.'
                        properties:
                          path:
                            description: 'The path of the service referen
ce.'
                            type: string
                          port:
                            description: 'The port of the service referen
ce.'
                            format: int32
                            type: integer
                        type: object
                      type:
                        type: string
                    type: object
                type: object
```

```
              type: array
            selector:
              description: 'The selector is used to match resources that
belong to the Application.
                All of the applications resources should have labels such
that they match this selector.
                Users should use the app.kubernetes.io/name label on all
components of the Application
                and set the selector to match this label. For instance,
                {"matchLabels": [{"app.kubernetes.io/name": "my-cool-ap
p"}]} should be used as the selector
                for an Application named "my-cool-app", and each componen
t should contain a label that matches.'
              type: object
          type: object
        status:
          description: 'The status summarizes the current state of the ob
ject.'
          properties:
            observedGeneration:
              description: 'The observedGeneration is the generation most
recently observed by the component
                responsible for acting upon changes to the desired state
of the resource.'
              format: int64
              type: integer
          type: object
  version: v1beta1
  versions:
  - name: v1beta1
    served: true
    storage: true
```

ApplicationHistory 定义

ApplicationHistory CRD 捕获所有生命周期操作（如创建、更新、回滚）为版本控制快照，并与 Application CRD 紧密集成，实现端到端审计追踪。

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: applicationhistories.app.k8s.io
spec:
  group: app.k8s.io
  names:
```

工作负载类型

除了通过 Applications 模块创建云原生应用外，也可以直接在 Container Platform > Workloads 中创建工作负载：

- Deployment：最常用的工作负载控制器，用于部署无状态应用。它确保指定数量的 Pod 副本在运行，支持滚动更新和回滚，适合无状态服务如 Web 服务器和 API。
- DaemonSet：确保 Pod 在集群中的每个节点（或特定节点）上运行。节点加入时自动创建 Pod，节点离开时自动删除 Pod。适合节点级任务，如日志代理和监控守护进程。
- StatefulSet：用于管理有状态应用的工作负载控制器。为每个 Pod 提供稳定的网络身份（主机名）和持久存储，确保即使在重新调度时数据也保持一致。适用于数据库、分布式缓存及其他有状态服务。
- CronJob：使用 cron 表达式管理基于时间的 Job。系统会在预定时间间隔自动创建 Job，适合定期任务，如备份、报表生成和清理作业。
- Job：用于运行有限任务的工作负载。它创建一个或多个 Pod，并确保指定数量的成功完成后终止。适合批处理、数据迁移及其他一次性操作。

除了通过 Web 控制台创建工作负载外，Kubernetes 还支持通过 CLI 工具直接管理更底层的资源：

- Pod：Kubernetes 中最小的可部署单元。Pod 可以包含一个或多个紧密耦合的容器，共享存储、网络和生命周期。Pod 通常由更高级别的控制器（如 Deployments）管理。
- Container：封装应用代码和依赖的标准化单元，确保跨环境一致执行。容器运行在 Pod 内，共享 Pod 的资源。

理解参数

目录

Overview

Core Concepts

什么是参数？

与容器镜像的关系

使用场景和示例

1. 应用配置

2. 不同环境部署

3. 数据库连接配置

CLI 示例及实际用法

使用 kubectl run

使用 kubectl create

复杂参数示例

带自定义配置的 Web 服务器

多参数应用示例

最佳实践

1. 参数设计原则

2. 安全注意事项

3. 配置管理

常见问题排查

1. 参数未被识别

2. 参数覆盖无效

3. 调试参数问题

高级使用模式

1. 使用 Init 容器实现条件参数
2. 使用 Helm 实现参数模板化

Overview

Kubernetes 中的参数指的是在运行时传递给容器的命令行参数。它们对应于 Kubernetes Pod 规范中的 `args` 字段，并覆盖容器镜像中定义的默认 CMD 参数。参数提供了一种灵活的方式来配置应用行为，而无需重新构建镜像。

Core Concepts

什么是参数？

参数是运行时传递的参数，具有以下特点：

- 覆盖容器镜像中的默认 CMD 指令
- 作为命令行参数传递给容器的主进程
- 允许动态配置应用行为
- 支持使用相同镜像进行不同配置的复用

与容器镜像的关系

在容器镜像术语中：

- **ENTRYPOINT**：定义可执行文件（对应 Kubernetes 的 `command`）
- **CMD**：提供默认参数（对应 Kubernetes 的 `args`）
- 参数：覆盖 CMD 参数，同时保留 ENTRYPOINT

```
# Dockerfile 示例
FROM nginx:alpine
ENTRYPOINT ["nginx"]
CMD ["-g", "daemon off;"]
```

```
# Kubernetes 覆盖示例
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: nginx
    image: nginx:alpine
    args: ["-g", "daemon off;", "-c", "/custom/nginx.conf"]
```

使用场景和示例

1. 应用配置

向应用传递配置选项：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  template:
    spec:
      containers:
      - name: app
        image: myapp:latest
        args:
        - "--port=8080"
        - "--log-level=info"
        - "--config=/etc/app/config.yaml"
```

2. 不同环境部署

针对不同环境使用不同参数：

```
# 开发环境
args: ["--debug", "--reload", "--port=3000"]

# 生产环境
args: ["--optimize", "--port=80", "--workers=4"]
```

3. 数据库连接配置

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: db-client
    image: postgres:13
    args:
    - "psql"
    - "-h"
    - "postgres.example.com"
    - "-p"
    - "5432"
    - "-U"
    - "myuser"
    - "-d"
    - "mydb"
```

CLI 示例及实际用法

使用 kubectl run

```
# 基本参数传递
kubectl run nginx --image=nginx:alpine --restart=Never -- -g "daemon of
f;" -c "/custom/nginx.conf"

# 多个参数
kubectl run myapp --image=myapp:latest --restart=Never -- --port=8080 --l
og-level=debug

# 交互式调试
kubectl run debug --image=ubuntu:20.04 --restart=Never -it -- /bin/bash
```

使用 **kubectl create**

```
# 创建带参数的 deployment
kubectl create deployment web --image=nginx:alpine --dry-run=client -o ya
ml > deployment.yaml

# 编辑生成的 YAML 添加 args:
# spec:
#   template:
#     spec:
#       containers:
#       - name: nginx
#         image: nginx:alpine
#         args: ["-g", "daemon off;", "-c", "/custom/nginx.conf"]

kubectl apply -f deployment.yaml
```

复杂参数示例

带自定义配置的 **Web** 服务器

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-custom
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-custom
  template:
    metadata:
      labels:
        app: nginx-custom
    spec:
      containers:
      - name: nginx
        image: nginx:1.21-alpine
        args:
        - "-g"
        - "daemon off;"
        - "-c"
        - "/etc/nginx/custom.conf"
        ports:
        - containerPort: 80
        volumeMounts:
        - name: config
          mountPath: /etc/nginx/custom.conf
          subPath: nginx.conf
      volumes:
      - name: config
        configMap:
          name: nginx-config
```

多参数应用示例

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp
spec:
  containers:
  - name: app
    image: mycompany/myapp:v1.2.3
    args:
    - "--server-port=8080"
    - "--database-url=postgresql://db:5432/mydb"
    - "--log-level=info"
    - "--metrics-enabled=true"
    - "--cache-size=256MB"
    - "--worker-threads=4"
```

最佳实践

1. 参数设计原则

- 使用有意义的参数名：如 `--port=8080`，而非 `-p 8080`
- 提供合理默认值：确保应用在无参数时也能正常工作
- 文档化所有参数：包含帮助文本和示例
- 验证输入：检查参数值并提供错误提示

2. 安全注意事项

```yaml
# ❌ 避免在参数中包含敏感数据
args: ["--api-key=secret123", "--password=mypass"]

# ✅ 使用环境变量传递密钥
env:
- name: API_KEY
  valueFrom:
    secretKeyRef:
      name: app-secrets
      key: api-key
args: ["--config-from-env"]
```

3. 配置管理

```yaml
# ✅ 参数与 ConfigMap 结合使用
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    args:
    - "--config=/etc/config/app.yaml"
    - "--log-level=info"
    volumeMounts:
    - name: config
      mountPath: /etc/config
  volumes:
  - name: config
    configMap:
      name: app-config
```

常见问题排查

1. 参数未被识别

```
# 查看容器日志
kubectl logs pod-name

# 常见错误：unknown flag
# 解决方案：检查参数语法及应用文档
```

2. 参数覆盖无效

```
# ❌ 错误示例：command 与 args 混用
command: ["myapp", "--port=8080"]
args: ["--log-level=debug"]

# ✅ 正确示例：仅使用 args 覆盖 CMD
args: ["--port=8080", "--log-level=debug"]
```

3. 调试参数问题

```
# 交互式运行容器测试参数
kubectl run debug --image=myapp:latest -it --rm --restart=Never -- /bin/sh

# 容器内手动测试命令
/app/myapp --port=8080 --log-level=debug
```

高级使用模式

1. 使用 Init 容器实现条件参数

```yaml
apiVersion: v1
kind: Pod
spec:
  initContainers:
  - name: config-generator
    image: busybox
    command: ['sh', '-c']
    args:
    - |
      if [ "$ENVIRONMENT" = "production" ]; then
        echo "--optimize --workers=8" > /shared/args
      else
        echo "--debug --reload" > /shared/args
      fi
    volumeMounts:
    - name: shared
      mountPath: /shared
  containers:
  - name: app
    image: myapp:latest
    command: ['sh', '-c']
    args: ['exec myapp $(cat /shared/args)']
    volumeMounts:
    - name: shared
      mountPath: /shared
  volumes:
  - name: shared
    emptyDir: {}
```

2. 使用 Helm 实现参数模板化

```yaml
# values.yaml
app:
  parameters:
    port: 8080
    logLevel: info
    workers: 4

# deployment.yaml 模板
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
      - name: app
        image: myapp:latest
        args:
        - "--port={{ .Values.app.parameters.port }}"
        - "--log-level={{ .Values.app.parameters.logLevel }}"
        - "--workers={{ .Values.app.parameters.workers }}"
```

参数为 Kubernetes 中容器化应用的配置提供了强大机制。通过正确理解和使用参数，您可以创建灵活、可复用且易维护的部署，满足不同环境和需求的适配。

理解环境变量

目录

- Overview
- Core Concepts
  - 什么是环境变量？
  - Kubernetes 中环境变量的来源
  - 环境变量优先级
- 使用场景和示例
  - 1. 应用配置
  - 2. 数据库配置
  - 3. 动态运行时信息
  - 4. 不同环境的配置
- CLI 示例及实用用法
  - 使用 kubectl run
  - 使用 kubectl create
  - 复杂环境变量示例
    - 带服务发现的微服务
    - 多容器 Pod 共享配置
- 最佳实践
  - 1. 安全最佳实践
  - 2. 配置组织
  - 3. 环境变量命名
  - 4. 默认值和校验

Overview

Kubernetes 中的环境变量是以键值对形式存在的配置数据，在容器运行时提供给容器使用。它们提供了一种灵活且安全的方式，将配置信息、密钥和运行时参数注入到应用程序中，而无需修改容器镜像或应用代码。

Core Concepts

什么是环境变量？

环境变量是：

- 容器内运行进程可访问的键值对
- 一种无需重建镜像的运行时配置机制
- 向应用程序传递配置信息的标准方式
- 通过任何编程语言的标准操作系统 API 访问

Kubernetes 中环境变量的来源

Kubernetes 支持多种环境变量来源：

来源类型	描述	使用场景
静态值	直接的键值对	简单配置
ConfigMap	引用 ConfigMap 中的键	非敏感配置
Secret	引用 Secret 中的键	敏感数据（密码、令牌）
字段引用	Pod/容器元数据	动态运行时信息
资源引用	资源请求/限制	资源感知配置

环境变量优先级

环境变量覆盖配置的顺序如下：

1. **Kubernetes** 环境变量（最高优先级）
2. 引用的 **ConfigMaps/Secrets**
3. **Dockerfile** 中的 **ENV** 指令
4. 应用默认值（最低优先级）

使用场景和示例

1. 应用配置

基础应用设置：

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: web-app
    image: myapp:latest
    env:
    - name: PORT
      value: "8080"
    - name: LOG_LEVEL
      value: "info"
    - name: ENVIRONMENT
      value: "production"
    - name: MAX_CONNECTIONS
      value: "100"
```

2. 数据库配置

使用 ConfigMaps 和 Secrets 配置数据库连接：

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  DB_HOST: "postgres.example.com"
  DB_PORT: "5432"
  DB_NAME: "myapp"
  DB_POOL_SIZE: "10"

---
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  DB_USER: bXl1c2Vy  # base64 编码的 "myuser"
  DB_PASSWORD: bXlwYXNzd29yZA==  # base64 编码的 "mypassword"

---
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    env:
    # 来自 ConfigMap
    - name: DB_HOST
      valueFrom:
        configMapKeyRef:
          name: db-config
          key: DB_HOST
    - name: DB_PORT
      valueFrom:
        configMapKeyRef:
          name: db-config
          key: DB_PORT
    - name: DB_NAME
      valueFrom:
        configMapKeyRef:
          name: db-config
```

```
        key: DB_NAME
  # 来自 Secret
  - name: DB_USER
    valueFrom:
      secretKeyRef:
        name: db-secret
        key: DB_USER
  - name: DB_PASSWORD
    valueFrom:
      secretKeyRef:
        name: db-secret
        key: DB_PASSWORD
```

3. 动态运行时信息

访问 Pod 和 Node 元数据：

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    env:
    # Pod 信息
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    # 资源信息
    - name: CPU_REQUEST
      valueFrom:
        resourceFieldRef:
          resource: requests.cpu
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          resource: limits.memory
```

4. 不同环境的配置

针对不同环境的配置示例：

```yaml
# 开发环境
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config-dev
data:
  DEBUG: "true"
  LOG_LEVEL: "debug"
  CACHE_TTL: "60"
  RATE_LIMIT: "1000"

---
# 生产环境
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config-prod
data:
  DEBUG: "false"
  LOG_LEVEL: "warn"
  CACHE_TTL: "3600"
  RATE_LIMIT: "100"

---
# 使用环境特定配置的部署
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  template:
    spec:
      containers:
      - name: app
        image: myapp:latest
        envFrom:
        - configMapRef:
            name: app-config-prod  # 开发时改为 app-config-dev
```

CLI 示例及实用用法

使用 kubectl run

```
# 直接设置环境变量
kubectl run myapp --image=nginx --env="PORT=8080" --env="DEBUG=true"

# 多个环境变量
kubectl run webapp --image=myapp:latest \
  --env="DATABASE_URL=postgresql://localhost:5432/mydb" \
  --env="REDIS_URL=redis://localhost:6379" \
  --env="LOG_LEVEL=info"

# 交互式 Pod 并设置环境变量
kubectl run debug --image=ubuntu:20.04 -it --rm \
  --env="TEST_VAR=hello" \
  --env="ANOTHER_VAR=world" \
  -- /bin/bash
```

使用 kubectl create

```
# 从字面值创建 ConfigMap
kubectl create configmap app-config \
  --from-literal=DATABASE_HOST=postgres.example.com \
  --from-literal=DATABASE_PORT=5432 \
  --from-literal=CACHE_SIZE=256MB

# 从文件创建 ConfigMap
echo "DEBUG=true" > app.env
echo "LOG_LEVEL=debug" >> app.env
kubectl create configmap app-env --from-env-file=app.env

# 为敏感数据创建 Secret
kubectl create secret generic db-secret \
  --from-literal=username=myuser \
  --from-literal=password=mypassword
```

复杂环境变量示例

带服务发现的微服务

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: service-config
data:
  USER_SERVICE_URL: "http://user-service:8080"
  ORDER_SERVICE_URL: "http://order-service:8080"
  PAYMENT_SERVICE_URL: "http://payment-service:8080"
  NOTIFICATION_SERVICE_URL: "http://notification-service:8080"

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-gateway
spec:
  template:
    spec:
      containers:
      - name: gateway
        image: api-gateway:latest
        env:
        - name: PORT
          value: "8080"
        - name: ENVIRONMENT
          value: "production"
        envFrom:
        - configMapRef:
            name: service-config
        - secretRef:
            name: api-keys
```

多容器 Pod 共享配置

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-app
spec:
  containers:
  # 主应用容器
  - name: app
    image: myapp:latest
    env:
    - name: ROLE
      value: "primary"
    - name: SHARED_SECRET
      valueFrom:
        secretKeyRef:
          name: shared-secret
          key: token
    envFrom:
    - configMapRef:
        name: shared-config

  # Sidecar 容器
  - name: sidecar
    image: sidecar:latest
    env:
    - name: ROLE
      value: "sidecar"
    - name: MAIN_APP_URL
      value: "http://localhost:8080"
    - name: SHARED_SECRET
      valueFrom:
        secretKeyRef:
          name: shared-secret
          key: token
    envFrom:
    - configMapRef:
        name: shared-config
```

最佳实践

1. 安全最佳实践

```yaml
# ✅ 对敏感数据使用 Secrets
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  api-key: <base64-encoded-value>
  database-password: <base64-encoded-value>

---
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    env:
    # ✅ 引用 Secrets
    - name: API_KEY
      valueFrom:
        secretKeyRef:
          name: app-secrets
          key: api-key
    # ❌ 避免硬编码敏感数据
    # - name: API_KEY
    #   value: "secret-api-key-123"
```

2. 配置组织

```yaml
# ✅ 按用途组织配置
apiVersion: v1
kind: ConfigMap
metadata:
  name: database-config
data:
  DB_HOST: "postgres.example.com"
  DB_PORT: "5432"
  DB_POOL_SIZE: "10"

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: cache-config
data:
  REDIS_HOST: "redis.example.com"
  REDIS_PORT: "6379"
  CACHE_TTL: "3600"

---
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    envFrom:
    - configMapRef:
        name: database-config
    - configMapRef:
        name: cache-config
```

3. 环境变量命名

```yaml
# ✅ 使用一致的命名规范
env:
- name: DATABASE_HOST       # 清晰且描述性强的名称
  value: "postgres.example.com"
- name: DATABASE_PORT       # 使用下划线分隔
  value: "5432"
- name: LOG_LEVEL          # 环境变量使用大写
  value: "info"
- name: FEATURE_FLAG_NEW_UI  # 相关变量使用前缀
  value: "true"

# ❌ 避免不清晰或不一致的命名
# - name: db                # 太短
# - name: databaseHost      # 命名风格不一致
# - name: log-level         # 分隔符不一致
```

4. 默认值和校验

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    env:
    - name: PORT
      value: "8080"          # 提供合理默认值
    - name: LOG_LEVEL
      value: "info"          # 默认安全值
    - name: TIMEOUT_SECONDS
      value: "30"            # 名称中包含单位
    - name: MAX_RETRIES
      value: "3"             # 限制重试次数
```

理解 Startup Commands

目录

Overview

Core Concepts

什么是 Startup Commands？

与 Dockerfile 和参数的关系

Command 与 Args 的交互

使用场景和示例

1. 自定义应用启动

2. 调试和故障排查

3. 初始化脚本

4. 多用途镜像

CLI 示例与实际使用

使用 kubectl run

使用 kubectl create job

复杂启动命令示例

多步骤初始化

条件启动逻辑

最佳实践

1. 信号处理与优雅关闭

2. 错误处理与日志记录

3. 安全性考虑

4. 资源管理

高级使用模式

1. 带自定义命令的 Init Containers
2. 使用不同命令的 Sidecar 容器
3. 带自定义命令的 Job 模式

Overview

Kubernetes 中的 startup commands 定义了容器启动时运行的主要可执行文件。它们对应于 Kubernetes Pod 规范中的 `command` 字段，并覆盖容器镜像中定义的默认 ENTRYPOINT 指令。startup commands 提供了对容器内部运行进程的完全控制。

Core Concepts

什么是 **Startup Commands**？

Startup commands 是：

- 容器启动时运行的主要可执行文件
- 覆盖容器镜像中的 ENTRYPOINT 指令
- 定义容器内的主进程（PID 1）
- 与参数（args）配合使用，形成完整的命令行

与 **Dockerfile** 和参数的关系

理解 Dockerfile 指令与 Kubernetes 字段之间的关系：

Dockerfile	**Kubernetes**	作用
ENTRYPOINT	`command`	定义可执行文件
CMD	`args`	提供默认参数

```dockerfile
# Dockerfile 示例
FROM ubuntu:20.04
ENTRYPOINT ["/usr/bin/myapp"]
CMD ["--config=/etc/default.conf"]
```

```yaml
# Kubernetes 覆盖示例
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: myapp
    image: myapp:latest
    command: ["/usr/bin/myapp"]
    args: ["--config=/etc/custom.conf", "--debug"]
```

Command 与 Args 的交互

场景	容器镜像	Kubernetes 规范	最终命令
默认	ENTRYPOINT + CMD	（无）	ENTRYPOINT + CMD
仅覆盖 args	ENTRYPOINT + CMD	`args: ["new-args"]`	ENTRYPOINT + new-args
仅覆盖 command	ENTRYPOINT + CMD	`command: ["new-cmd"]`	new-cmd
同时覆盖 command 和 args	ENTRYPOINT + CMD	`command: ["new-cmd"]` `args: ["new-args"]`	new-cmd + new-args

使用场景和示例

1. 自定义应用启动

使用相同基础镜像运行不同的应用：

```
apiVersion: v1
kind: Pod
metadata:
  name: web-server
spec:
  containers:
  - name: nginx
    image: ubuntu:20.04
    command: ["/usr/sbin/nginx"]
    args: ["-g", "daemon off;", "-c", "/etc/nginx/nginx.conf"]
```

2. 调试和故障排查

覆盖默认命令以启动 shell 进行调试：

```
apiVersion: v1
kind: Pod
metadata:
  name: debug-pod
spec:
  containers:
  - name: debug
    image: myapp:latest
    command: ["/bin/bash"]
    args: ["-c", "sleep 3600"]
```

3. 初始化脚本

在启动主应用之前运行自定义初始化：

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/sh"]
    args:
    - "-c"
    - |
      echo "Initializing application..."
      /scripts/init.sh
      echo "Starting main application..."
      exec /usr/bin/myapp --config=/etc/app.conf
```

4. 多用途镜像

使用同一镜像实现不同用途：

```yaml
# Web 服务器
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  template:
    spec:
      containers:
      - name: web
        image: myapp:latest
        command: ["/usr/bin/myapp"]
        args: ["server", "--port=8080"]

---
# 后台工作进程
apiVersion: apps/v1
kind: Deployment
metadata:
  name: worker
spec:
  template:
    spec:
      containers:
      - name: worker
        image: myapp:latest
        command: ["/usr/bin/myapp"]
        args: ["worker", "--queue=tasks"]

---
# 数据库迁移
apiVersion: batch/v1
kind: Job
metadata:
  name: migrate
spec:
  template:
    spec:
      containers:
      - name: migrate
        image: myapp:latest
        command: ["/usr/bin/myapp"]
        args: ["migrate", "--up"]
```

```
restartPolicy: Never
```

CLI 示例与实际使用

使用 kubectl run

```
# 完全覆盖命令
kubectl run debug --image=nginx:alpine --command -- /bin/sh -c "sleep 360
0"

# 运行交互式 shell
kubectl run -it debug --image=ubuntu:20.04 --restart=Never --command -- /
bin/bash

# 自定义应用启动
kubectl run myapp --image=myapp:latest --command -- /usr/local/bin/start.
sh --config=/etc/app.conf

# 一次性任务
kubectl run task --image=busybox --restart=Never --command -- /bin/sh -c
"echo 'Task completed'"
```

使用 kubectl create job

```
# 创建带自定义命令的 job
kubectl create job backup --image=postgres:13 --dry-run=client -o yaml --
pg_dump -h db.example.com mydb > backup.yaml

# 应用 job
kubectl apply -f backup.yaml
```

复杂启动命令示例

多步骤初始化

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: complex-init
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/bash"]
    args:
    - "-c"
    - |
      set -e
      echo "Step 1: Checking dependencies..."
      /scripts/check-deps.sh

      echo "Step 2: Setting up configuration..."
      /scripts/setup-config.sh

      echo "Step 3: Running database migrations..."
      /scripts/migrate.sh

      echo "Step 4: Starting application..."
      exec /usr/bin/myapp --config=/etc/app/config.yaml
    volumeMounts:
    - name: scripts
      mountPath: /scripts
    - name: config
      mountPath: /etc/app
  volumes:
  - name: scripts
    configMap:
      name: init-scripts
      defaultMode: 0755
  - name: config
    configMap:
      name: app-config
```

条件启动逻辑

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: conditional-app
spec:
  template:
    spec:
      containers:
      - name: app
        image: myapp:latest
        command: ["/bin/sh"]
        args:
        - "-c"
        - |
          if [ "$APP_MODE" = "worker" ]; then
            exec /usr/bin/myapp worker --queue=$QUEUE_NAME
          elif [ "$APP_MODE" = "scheduler" ]; then
            exec /usr/bin/myapp scheduler --interval=60
          else
            exec /usr/bin/myapp server --port=8080
          fi
        env:
        - name: APP_MODE
          value: "server"
        - name: QUEUE_NAME
          value: "default"
```

最佳实践

1. 信号处理与优雅关闭

```yaml
# ✅ 正确的信号处理
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/bash"]
    args:
    - "-c"
    - |
      # 捕获 SIGTERM 实现优雅关闭
      trap 'echo "Received SIGTERM, shutting down gracefully..."; kill -TERM $PID; wait $PID' TERM

      # 后台启动主应用
      /usr/bin/myapp --config=/etc/app.conf &
      PID=$!

      # 等待进程结束
      wait $PID
```

2. 错误处理与日志记录

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/bash"]
    args:
    - "-c"
    - |
      set -euo pipefail  # 出错、未定义变量或管道失败时退出

      log() {
        echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" >&2
      }

      log "Starting application initialization..."

      if ! /scripts/health-check.sh; then
        log "ERROR: Health check failed"
        exit 1
      fi

      log "Starting main application..."
      exec /usr/bin/myapp --config=/etc/app.conf
```

3. 安全性考虑

```yaml
# ✅ 以非 root 用户运行
apiVersion: v1
kind: Pod
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 1000
  containers:
  - name: app
    image: myapp:latest
    command: ["/usr/bin/myapp"]
    args: ["--config=/etc/app.conf"]
    securityContext:
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
      capabilities:
        drop:
        - ALL
```

4. 资源管理

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/usr/bin/myapp"]
    args: ["--config=/etc/app.conf"]
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

高级使用模式

1. 带自定义命令的 Init Containers

```yaml
apiVersion: v1
kind: Pod
spec:
  initContainers:
  - name: setup
    image: busybox
    command: ["/bin/sh"]
    args:
    - "-c"
    - |
      echo "Setting up shared data..."
      mkdir -p /shared/data
      echo "Setup complete" > /shared/data/status
    volumeMounts:
    - name: shared-data
      mountPath: /shared
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/sh"]
    args:
    - "-c"
    - |
      while [ ! -f /shared/data/status ]; do
        echo "Waiting for setup to complete..."
        sleep 1
      done
      echo "Starting application..."
      exec /usr/bin/myapp
    volumeMounts:
    - name: shared-data
      mountPath: /shared
  volumes:
  - name: shared-data
    emptyDir: {}
```

2. 使用不同命令的 Sidecar 容器

```yaml
apiVersion: v1
kind: Pod
spec:
  containers:
  # 主应用
  - name: app
    image: myapp:latest
    command: ["/usr/bin/myapp"]
    args: ["--config=/etc/app.conf"]

  # 日志收集 sidecar
  - name: log-shipper
    image: fluent/fluent-bit:latest
    command: ["/fluent-bit/bin/fluent-bit"]
    args: ["--config=/fluent-bit/etc/fluent-bit.conf"]

  # 指标导出 sidecar
  - name: metrics
    image: prom/node-exporter:latest
    command: ["/bin/node_exporter"]
    args: ["--path.rootfs=/host"]
```

3. 带自定义命令的 Job 模式

```yaml
# 备份 job
apiVersion: batch/v1
kind: Job
metadata:
  name: database-backup
spec:
  template:
    spec:
      containers:
      - name: backup
        image: postgres:13
        command: ["/bin/bash"]
        args:
        - "-c"
        - |
          set -e
          echo "Starting backup at $(date)"
          pg_dump -h $DB_HOST -U $DB_USER $DB_NAME > /backup/dump-$(date
+%Y%m%d-%H%M%S).sql
          echo "Backup completed at $(date)"
        env:
        - name: DB_HOST
          value: "postgres.example.com"
        - name: DB_USER
          value: "backup_user"
        - name: DB_NAME
          value: "myapp"
        volumeMounts:
        - name: backup-storage
          mountPath: /backup
      restartPolicy: Never
      volumes:
      - name: backup-storage
        persistentVolumeClaim:
          claimName: backup-pvc
```

Startup commands 在 Kubernetes 中提供了对容器执行的完全控制。通过理解如何正确配置和使用 startup commands，您可以创建灵活、可维护且健壮的容器化应用，以满足您的具体需求。

资源单位说明

- CPU：可选单位为：core、m（millicore）。其中 1 core = 1000 m。
- Memory：可选单位为：Mi（1 MiB = 2^20 字节）、Gi（1 GiB = 2^30 字节）。其中 1 Gi = 1024 Mi。
- Virtual GPU（可选）：该参数仅在集群中存在 GPU 资源时生效。虚拟 GPU 核数；100 个虚拟核等于 1 个物理 GPU 核。支持正整数。
- Video Memory（可选）：该参数仅在集群中存在 GPU 资源时生效。虚拟 GPU 显存；1 单位显存等于 256 Mi。支持正整数。

Alauda Container Platform

命名空间

创建命名空间

了解命名空间

通过 Web 控制台创建命名空间

通过 **ac** CLI 在项目中创建命名空间

通过 **kubectl** 创建命名空间

导入 Namespace

Overview

Use Cases

Prerequisites

Procedure

资源配额

了解资源请求与

配额

硬件加速器资源

UID/GID 分配

启用 UID/GID 分配

验证 UID/GID 分配

更新命名空间

更新配额

更新容器 LimitRanges

更新 Pod Security Admission

删除/移除命名空间

删除命名空间

移除命名空间

创建命名空间

目录

了解命名空间

通过 Web 控制台创建命名空间

通过 **ac** CLI 在项目中创建命名空间

通过 **kubectl** 创建命名空间

- YAML 文件示例
- 通过 YAML 文件创建
- 直接通过命令行创建

了解命名空间

参考官方 Kubernetes 文档：Namespaces ↗

> 在 Kubernetes 中，命名空间提供了一种在单个集群内隔离资源组的机制。资源名称在命名空间内必须唯一，但不同命名空间之间可以重复。基于命名空间的作用域仅适用于有命名空间的对象（例如 Deployments、Services 等），不适用于集群范围的对象（例如 StorageClass、Nodes、PersistentVolumes 等）。

通过 **Web** 控制台创建命名空间

> 在与项目关联的集群内，创建一个新的命名空间，该命名空间需符合项目可用资源配额。 新命名空间运行在项目分配的资源配额范围内（如 CPU、内存），且命名空间中的所有资源必须位于关联的集群内。

1. 在 项目管理 视图中，点击要创建命名空间的 *项目名称*。
2. 在左侧导航栏中，点击 **Namespaces** > **Namespaces**。
3. 点击 创建命名空间。
4. 配置 基本信息。

参数	说明
Cluster	选择与项目关联的集群，用于承载该命名空间。
Namespace	命名空间名称必须包含一个必填前缀，即项目名称。

5. （可选）配置资源配额。
每当为命名空间内的容器指定计算或存储资源的限制（limits），或每当新增 Pod 或 PVC 时，都会消耗此处设置的配额。
注意：

- 命名空间的资源配额继承自项目在集群中的分配配额。某资源类型的最大允许配额不得超过项目剩余可用配额。如果某资源的可用配额为 0，则会阻止命名空间创建。请联系平台管理员调整配额。
- **GPU** 配额配置要求：
  - 仅当集群中配置了 GPU 资源时，才可配置 GPU 配额（vGPU 或 pGPU）。
  - 使用 vGPU 时，也可设置内存配额。

  GPU 单位定义：

  - **vGPU** 单位：100 个虚拟 GPU 单位（vGPU）= 1 个物理 GPU 核心（pGPU）。
    - 注意：pGPU 单位仅以整数计数（例如 1 pGPU = 1 核心 = 100 vGPU）。
  - 内存单位：
    - 1 个内存单位 = 256 MiB。
    - 1 GiB = 4 个内存单位（1024 MiB = 4 × 256 MiB）。

- 默认配额行为：
  - 若未指定某资源类型的配额，则默认不设限。
  - 这意味着命名空间可以使用项目分配的该类型所有可用资源，无需显式限制。

配额参数说明

类别	配额类型	数值及单位	说明
存储资源配额	全部	Gi	该命名空间内所有 Persistent Volume Claims（PVC）请求的存储容量总和不得超过此值。
	存储类		该命名空间内所有与所选 StorageClass 关联的 Persistent Volume Claims（PVC）请求的存储容量总和不得超过此值。 注意：请提前将 StorageClass 分配给命名空间所属项目。
扩展资源	从配置字典（ConfigMap）中获取；详情请参见扩展资源配额说明。	-	若无对应配置字典，则不显示此类别。
其他配额	输入自定义配额；具体输入规则请参见其他配额说明。	-	为避免资源重复问题，以下值不可作为配额类型： • limits.cpu • limits.memory • requests.cpu • requests.memory

类别	配额类型	数值及单位	说明
			• pods • cpu • memory

6. （可选）配置 容器限制范围，详情请参见限制范围。
7. （可选）配置 **Pod** 安全准入，具体详情请参见Pod 安全准入。
8. （可选）在 更多配置 区域，为当前命名空间添加标签和注解。
   提示：可以通过标签定义命名空间的属性，或通过注解为命名空间补充额外信息；两者均可用于过滤和排序命名空间。
9. 点击 创建。

通过 ac CLI 在项目中创建命名空间

使用 `ac` 命令行工具在项目中创建命名空间，执行以下命令：

```
# 在指定项目和集群中创建命名空间
ac adm new-project-namespace <namespace-name> --project <project-name> --cluster <cluster-name>
```

通过 kubectl 创建命名空间

YAML 文件示例

```
example-namespace.yaml
```

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: example
  labels:
    pod-security.kubernetes.io/audit: baseline # Option, to ensure security, it is recommended to choose the baseline or restricted mode.
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/warn: baseline
```

example-resourcequota.yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example-resourcequota
  namespace: example
spec:
  hard:
    limits.cpu: '20'
    limits.memory: 20Gi
    pods: '500'
    requests.cpu: '2'
    requests.memory: 2Gi
```

example-limitrange.yaml

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: example-limitrange
  namespace: example
spec:
  limits:
    - default:
        cpu: 100m
        memory: 100Mi
      defaultRequest:
        cpu: 50m
        memory: 50Mi
      max:
        cpu: 1000m
        memory: 1000Mi
      type: Container
```

通过 YAML 文件创建

```
kubectl apply -f example-namespace.yaml
kubectl apply -f example-resourcequota.yaml
kubectl apply -f example-limitrange.yaml
```

直接通过命令行创建

```
kubectl create namespace example
kubectl create resourcequota example-resourcequota --namespace=example --hard=limits.cpu=20,limits.memory=20Gi,pods=500
kubectl create limitrange example-limitrange --namespace=example --default='cpu=100m,memory=100Mi' --default-request='cpu=50m,memory=50Mi' --max='cpu=1000m,memory=1000Mi'
```

导入 Namespace

目录

Overview
Use Cases
Prerequisites
Procedure

Overview

Namespace 生命周期管理能力：

- 跨集群 Namespace 导入：将 Namespace 导入到一个 Project 中，实现对平台所提供的所有 Kubernetes 集群中 Namespace 的集中管理。这样为管理员提供了跨分布式环境的统一资源治理和监控能力。

Namespace 解绑：

- 解绑 Namespace 功能允许您将 Namespace 与当前 Project 解除关联，重置其关联状态，以便后续重新分配或清理。
- 将 Namespace 导入到 Project 中后，该 Namespace 具备与平台上原生创建的 Namespace 等效的能力，包括继承 Project 级别的策略（如资源配额）、统一监控和集中治理控制。

重要说明：

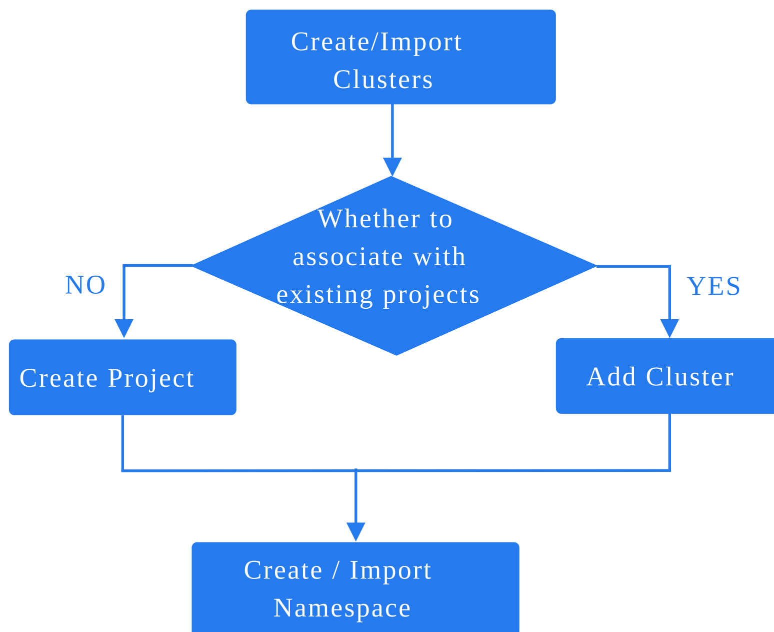
- 一个 Namespace 在任意时刻只能关联到一个 Project。

- 如果 Namespace 已经关联到某个 Project，必须先解除与该 Project 的关联，才能导入或重新分配到其他 Project。

Use Cases

Namespace 管理的常见用例包括：

- 当新 **Kubernetes** 集群 连接到平台时，可以使用 导入 **Namespace** 功能，将其已有的 **Kubernetes Namespace** 关联到某个 Project。只需选择目标 Project 和集群，即可开始导入。此操作赋予该 **Project** 对这些 **Namespace** 的治理权限，包括资源配额、监控和策略执行。



- 已从一个 **Project** 解绑的 **Namespace**，可以通过 导入 **Namespace** 功能无缝地重新关联到另一个 Project，实现持续的集中治理。
- 当前未被任何 **Project** 管理的 Namespace（例如通过集群级脚本创建的）必须使用 导入 **Namespace** 功能关联到目标 **Project**，以启用平台级治理，包括可视化和集中管理。

Prerequisites

- 该 Namespace 当前未被平台中任何已有 Project 管理。
- Namespace 只能导入到已关联其目标 Kubernetes 集群的 Project 中。如果不存在此类 Project，则需先创建一个与该集群关联的 Project。

Procedure

1. 在 **Project** 管理 中，点击要导入 Namespace 的 ***Project*** *名称*。
2. 进入 **Namespaces** > **Namespaces**。
3. 点击 创建 **Namespace** 旁的 下拉按钮，选择 导入 **Namespace**。
4. 参考创建 Namespace文档了解参数配置详情。
5. 点击 导入。

资源配额

参考官方 Kubernetes 文档：Resource Quotas

目录

了解资源请求与限制
配额
资源配额
YAML 文件示例
使用 CLI 创建资源配额
存储配额
硬件加速器资源配额
其他配额

了解资源请求与限制

用于限制特定命名空间可用的资源。该命名空间内所有 Pod 的总资源使用量（不包括处于 `Terminating` 状态的 Pod）不得超过配额。

资源请求：定义容器所需的最小资源（如 CPU、内存），指导 Kubernetes 调度器将 Pod 安排到具有足够容量的节点上。

资源限制：定义容器可使用的最大资源，防止资源耗尽，确保集群稳定。

配额

资源配额

> 如果某资源标记为 `Unlimited`，则不强制执行显式配额，但使用量不能超过集群的可用容量。

资源配额跟踪命名空间内的累计资源消耗（如容器限制、新建 Pod 或 PVC）。

支持的配额类型

字段	描述
资源请求	命名空间内所有 Pod 的总请求资源： • CPU • 内存
资源限制	命名空间内所有 Pod 的总限制资源： • CPU • 内存
Pod 数量	命名空间允许的最大 Pod 数量。

注意：

- 命名空间配额来源于项目分配的集群资源。如果任何资源的可用配额为 0，命名空间创建将失败。请联系管理员。
- `Unlimited` 表示该命名空间可使用项目剩余的该资源类型的集群资源。

YAML 文件示例

```yaml
# example-resourcequota.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example-resourcequota
  namespace: <example>
spec:
  hard:
    limits.cpu: "20"
    limits.memory: 20Gi
    pods: "500"
    requests.cpu: "2"
    requests.memory: 2Gi
```

使用 **CLI** 创建资源配额

通过 **YAML** 文件创建

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

直接通过命令行创建

```
kubectl create resourcequota example-resourcequota --namespace=<example>
--hard=limits.cpu=20,limits.memory=20Gi,pods=500
```

存储配额

配额类型：

- 全部：命名空间内 PVC 的总存储容量。
- 存储类：特定存储类的 PVC 总存储容量。

注意：确保存储类已预先分配给包含该命名空间的项目。

硬件加速器资源配额

当安装了 `Alauda Build of Hami` 或 `NVIDIA GPU Device Plugin` 后，您可以使用关于硬件加速器的扩展资源配额。

参考 Alauda Build of Hami 和 Alauda Build of NVIDIA GPU Device Plugin。

其他配额

自定义配额名称的格式必须符合以下规范：

- 如果自定义配额名称不包含斜杠（/）：必须以字母或数字开头和结尾，可以包含字母、数字、连字符（-）、下划线（_）或点（.），形成一个合格名称，最大长度为 63 个字符。
- 如果自定义配额名称包含斜杠（/）：名称分为两部分：前缀和名称，格式为：prefix/name。前缀必须是有效的 DNS 子域名，名称必须符合合格名称的规则。
- DNS 子域名：
  - 标签：必须以小写字母或数字开头和结尾，可以包含连字符（-），但不能全部由连字符组成，最大长度为 63 个字符。
  - 子域名：扩展标签规则，允许多个标签通过点（.）连接形成子域名，最大长度为 253 个字符。

Limit Range

目录

- 理解 Limit Range
- 使用 CLI 创建 Limit Range
  - YAML 文件示例
  - 通过 YAML 文件创建
  - 直接通过命令行创建

理解 Limit Range

参考官方 Kubernetes 文档：Limit Ranges

使用 Kubernetes LimitRange 作为准入控制器是在容器或 **Pod** 级别进行资源限制。它为在创建或更新 LimitRange 后创建的容器或 Pod 设置默认请求值、限制值和最大值，同时持续监控容器使用情况，确保命名空间内没有资源超过定义的最大值。

> 容器的资源请求是资源限制与集群超额配置之间的比例。资源请求值作为调度器调度容器时的参考和标准。调度器会检查每个节点的可用资源（总资源 - 节点上已调度 Pod 中容器资源请求的总和）。如果新 Pod 容器的资源请求总和超过节点剩余可用资源，则该 Pod 不会被调度到该节点。

LimitRange 是一种准入控制器：

- 它为所有未设置计算资源需求的容器应用默认请求和限制值。

- 它跟踪使用情况，确保不超过命名空间中任何 LimitRange 定义的资源最大值和比例。

包括以下配置

资源	字段
CPU	• 默认请求 • 限制 • 最大值
内存	• 默认请求 • 限制 • 最大值

使用 **CLI** 创建 **Limit Range**

YAML 文件示例

```yaml
# example-limitrange.yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: example-limitrange
  namespace: example
spec:
  limits:
    - default:
        cpu: 100m
        memory: 100Mi
      defaultRequest:
        cpu: 50m
        memory: 50Mi
      max:
        cpu: 1000m
        memory: 1000Mi
      type: Container
```

通过 YAML 文件创建

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

直接通过命令行创建

```
kubectl create limitrange example-limitrange --namespace=example --default='cpu=100m,memory=100Mi' --default-request='cpu=50m,memory=50Mi' --max='cpu=1000m,memory=1000Mi'
```

Pod Security Policies

ACP 支持 Kubernetes Pod Security Admission (PSA) 和 Kyverno Policy，以帮助在集群中运行的 Pods 强制执行安全标准。

目录

- Pod Security Admission
  - 安全模式
  - 安全标准
  - 配置
    - 命名空间标签
  - 免除
- Kyverno Policy
  - 前提条件
  - 应用 Kyverno 策略
  - Web 控制台
  - CLI

Pod Security Admission

参考 Kubernetes 官方文档：Pod Security Admission

Pod Security Admission (PSA) 是 Kubernetes 的一个准入控制器，通过验证 Pod 规范是否符合预定义标准，在命名空间级别强制执行安全策略。

安全模式

PSA 定义了三种模式来控制如何处理策略违规：

模式	行为	使用场景
Enforce	拒绝创建/修改不合规的 Pods。	需要严格安全执行的生产环境。
Audit	允许创建 Pod，但在审计日志中记录违规。	监控和分析安全事件，不阻止工作负载。
Warn	允许创建 Pod，但向客户端返回违规警告。	测试环境或策略调整的过渡阶段。

关键说明：

- **Enforce** 仅作用于 Pods（例如拒绝 Pods，但允许非 Pod 资源如 Deployments）。
- **Audit** 和 **Warn** 作用于 Pods 及其控制器（例如 Deployments）。

安全标准

PSA 定义了三种安全标准来限制 Pod 权限：

标准	描述	主要限制
Privileged	不受限制的访问，适用于受信任的工作负载（如系统组件）。	不验证 `securityContext` 字段。
Baseline	最小限制，防止已知的权限提升。	阻止 `hostNetwork`、`hostPID`、特权容器和不受限制的 `hostPath` 卷。
Restricted	最严格的策略，执行安全最佳实践。	要求： - `runAsNonRoot: true` - `seccompProfile.type: RuntimeDefault` - 丢弃 Linux 能力。

配置

命名空间标签

为命名空间应用标签以定义 PSA 策略。

YAML 文件示例

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: example-namespace
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/audit: baseline
    pod-security.kubernetes.io/warn: baseline
```

CLI 命令

```
# 第一步：更新 Pod Admission 标签
kubectl label namespace <namespace-name> \
  pod-security.kubernetes.io/enforce=baseline \
  pod-security.kubernetes.io/audit=restricted \
  --overwrite

# 第二步：验证标签
kubectl get namespace <namespace-name> --show-labels
```

免除

免除特定用户、命名空间或运行时类的 PSA 检查。

示例配置：

```yaml
apiVersion: pod-security.admission.config.k8s.io/v1
kind: PodSecurityConfiguration
exemptions:
  usernames: ['admin']
  runtimeClasses: ['nvidia']
  namespaces: ['kube-system']
```

Kyverno Policy

ACP 提供多个示例用于创建 Pod 安全的 Kyverno 策略。示例包括：

- **Restricted**：Restricted 阻止访问所有主机特性，要求 Pod 使用分配给命名空间的 UID 和 SELinux 上下文运行。
- **Restricted-v2**：Restricted-v2 阻止访问所有主机特性，要求 Pod 使用分配给命名空间的 UID 和 SELinux 上下文运行。这是最严格的策略，默认用于认证用户。基于传统的 'restricted'，它还要求丢弃所有能力，不允许权限提升的二进制文件。若未设置 seccomp 配置文件，则默认使用 runtime/default，否则必须使用该 seccomp 配置文件。
- **Anyuid**：Anyuid 提供了 restricted 策略的所有功能，但允许用户以任意 UID 和 GID 运行。
- **Hostaccess**：Hostaccess 允许访问所有主机命名空间，但仍要求 Pod 使用分配给命名空间的 UID 和 SELinux 上下文运行。警告：此策略允许访问主机命名空间、文件系统和 PID，仅应由受信任的 Pod 使用，谨慎授权。
- **Hostmount-anyuid**：Hostmount-anyuid 提供了 restricted 策略的所有功能，但允许 Pod 使用主机挂载和任意 UID 运行。主要用于持久卷回收器。警告：此策略允许以任意 UID（包括 UID 0）访问主机文件系统，谨慎授权。
- **Hostnetwork**：Hostnetwork 允许使用主机网络和主机端口，但仍要求 Pod 使用分配给命名空间的 UID 和 SELinux 上下文运行。
- **Hostnetwork-v2**：Hostnetwork-v2 允许使用主机网络和主机端口，但仍要求 Pod 使用分配给命名空间的 UID 和 SELinux 上下文运行。基于传统的 'hostnetwork' 策略，它还要求丢弃所有能力，不允许权限提升的二进制文件。若未设置 seccomp 配置文件，则默认使用 runtime/default，否则必须使用该 seccomp 配置文件。
- **Node-exporter**：Node-exporter 策略用于 Prometheus 节点导出器。
- **Nonroot**：Nonroot 提供了 restricted 策略的所有功能，但允许用户以任意非 root UID 运行。用户必须指定 UID，或由容器运行时的清单指定。
- **Nonroot-v2**：Nonroot-v2 提供了 restricted 策略的所有功能，但允许用户以任意非 root UID 运行。用户必须指定 UID，或由容器运行时的清单指定。基于传统的 'nonroot' 策略，它还要求丢弃所有能力，不允许权限提升的二进制文件。若未设置 seccomp 配置文件，则默认使用 runtime/default，否则必须使用该 seccomp 配置文件。
- **Privileged**：Privileged 允许访问所有特权和主机特性，并能以任意用户、任意组、任意 fsGroup 以及任意 SELinux 上下文运行。警告：这是最宽松的策略，仅应由集群管理员使用，谨慎授权。

> **NOTICE**
>
> **Restricted** 策略不等同于 Kubernetes Pod Security Admission 的 'restricted' 标准。 如果您想使用 Kyverno **Restricted** 策略而非 Kubernetes Pod Security Admission 的 'restricted' 标准，可能需要更改您的 Pod 安全配置。

前提条件

- 安装 Alauda Container Platform Compliance for Kyverno，参考文档：document。
- 在 ACP featuregate 设置中启用功能门控 `namespace-resource-manage`。

应用 Kyverno 策略

Web 控制台

1. 在 ACP 控制台，导航至 **Container Platform**，选择要应用安全策略的命名空间。
2. 进入 **Advanced** > **Resources**。
3. 搜索资源类型名称为 **Policy**，资源组为 **kyverno.io** 的资源。
4. 选择版本 "v2beta1"，点击 **Create** 创建新的 Kyverno Policy。
5. 在 **Create Resource** 对话框中，选择 **Samples** 标签页。
6. 选择所需的 Pod 安全策略示例（例如 `Restricted`），然后点击 **Try**。
7. 审核并根据需要修改策略 YAML，点击 **Update** 应用策略。

CLI

1. 登录到要应用安全策略的 Kubernetes 集群。
2. 运行以下命令从示例资源创建 Kyverno Policy：

```
$ kubectl get consoleyamlsamples.console.alauda.io restricted-policy -o
template --template={{.spec.yaml}}|kubectl apply -f -
$ kubectl get policies.kyverno.io
NAME          ADMISSION    BACKGROUND    READY    AGE    MESSAGE
restricted    true         true          True     1m    Ready
```

可用的示例资源包括：

- restricted-policy
- restrictedv2-policy
- anyuid-policy
- hostaccess-policy
- hostmount-anyuid-policy
- hostnetwork-policy
- hostnetwork-v2-policy
- node-exporter-policy
- nonroot-policy
- nonroot-v2-policy
- privileged-policy

UID/GID 分配

在 Kubernetes 中，每个 Pod 都以特定的用户 ID（UID）和组 ID（GID）运行，以确保安全性和适当的访问控制。默认情况下，Pod 可能以 root 用户（UID 0）身份运行，这可能带来安全风险。为了增强安全性，建议为 Pod 分配非 root 的 UID 和 GID。

ACP 允许自动为命名空间分配特定的 UID 和 GID 范围，以确保该命名空间内的所有 Pod 都以指定的用户和组 ID 运行。

目录

- 启用 UID/GID 分配
- 验证 UID/GID 分配
  - UID/GID 范围
  - 验证 Pod 的 UID/GID

启用 UID/GID 分配

要为命名空间启用 UID/GID 分配，请按照以下步骤操作：

1. 进入 项目管理。
2. 在左侧导航栏点击 命名空间。
3. 点击目标命名空间。
4. 点击 操作 > 更新 **Pod** 安全策略。
5. 将 执行 选项值更改为 受限，点击 更新。

6. 点击 标签 旁的编辑图标，添加键为 `security.cpaas.io/enabled`，值为 `true` 的标签，点击 更新。（要禁用，删除此标签或将值设置为 `false`。）
7. 点击 保存。

验证 UID/GID 分配

UID/GID 范围

在命名空间详情页面，可以在 注解 中查看分配的 UID 和 GID 范围。

security.cpaas.io/uid-range 注解指定了该命名空间内 Pod 可分配的 UID/GID 范围，例如 **security.cpaas.io/uid-range=1000002000-1000011999**，表示 uid/gid 范围是从 1000002000 到 1000011999。

验证 Pod 的 UID/GID

如果 Pod 在 `securityContext` 中未指定 `runAsUser` 和 `fsGroup`，平台将自动分配该命名空间 UID 范围中的第一个值。

1. 在命名空间中创建一个 Pod，YAML 配置如下：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: uid-gid-test-pod
spec:
  containers:
  - name: test-container
    image: busybox
    command: ["sleep", "3600"]
```

2. Pod 创建后，获取 Pod 的 yaml 以检查分配的 UID 和 GID：

```
kubectl get pod uid-gid-test-pod -n <namespace-name> -o yaml
```

Pod 的 YAML 会在 `securityContext` 部分显示分配的 UID 和 GID：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: uid-gid-test-pod
spec:
  containers:
  - name: test-container
    image: busybox
    command: ["sleep", "3600"]
    securityContext:
      runAsUser: 1000000
  securityContext:
    fsGroup: 1000000
```

如果 Pod 在 securityContext 中指定了 runAsUser 和 fsGroup，平台会验证指定的 UID/GID 是否在分配范围内。如果不在范围内，Pod 创建将失败。

1. 在命名空间中创建一个 Pod，YAML 配置如下：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: uid-gid-test-pod-invalid
spec:
  containers:
  - name: test-container
    image: busybox
    command: ["sleep", "3600"]
    securityContext:
      runAsUser: 2000000  # 无效的 UID，超出分配范围
  securityContext:
    fsGroup: 2000000  # 无效的 GID，超出分配范围
```

2. 应用该 YAML 后，Pod 创建会失败，并显示错误信息，提示指定的 UID/GID 超出分配范围。

超售比

目录

理解命名空间资源超售比

CRD 定义

使用 CLI 创建超售比

使用 Web 控制台创建/更新超售比

注意事项

操作步骤

理解命名空间资源超售比

灵雀云容器平台 允许您为每个命名空间设置资源超售比（CPU 和内存）。该设置管理该命名空间内容器的限制（最大使用量）与请求（保证的最小值）之间的关系，从而优化资源利用率。

通过配置该比率，您可以确保用户定义的容器限制和请求保持在合理范围内，提高整个集群的资源效率。

关键概念

- Limits：容器可使用的最大资源。超过限制可能导致 CPU 限流或内存被终止。
- Requests：容器所需的保证最小资源。Kubernetes 根据请求来调度容器。
- Overcommit Ratio：限制 / 请求。该设置定义了命名空间内此比率的可接受范围，平衡资源保障与防止过度消耗。

核心能力

- 通过设置合适的超售比，提升命名空间内资源密度和应用稳定性，管理资源限制与请求之间的平衡。

示例

> 假设命名空间超售比设置为 2，创建应用时指定 CPU 限制为 4c，则对应的 CPU 请求值计算如下：

CPU 请求 = CPU 限制 / 超售比。因此，CPU 请求为 4c / 2 = 2c。

CRD 定义

```yaml
# example-namespace-overcommit.yaml
apiVersion: resource.alauda.io/v1
kind: NamespaceResourceRatio
metadata:
  namespace: example
  name: example-namespace-overcommit
spec:
  cpu: 3 # 缺失该字段表示继承集群超售比；0 表示不限制。
  memory: 4 # 缺失该字段表示继承集群超售比；0 表示不限制。
status:
  clusterCPU: 2 # 集群超售比
  clusterMemory: 3
```

使用 CLI 创建超售比

```
kubectl apply -f example-namespace-overcommit.yaml
```

使用 Web 控制台创建/更新超售比

允许调整命名空间的超售比，以管理资源限制与请求之间的比例，确保容器资源分配保持在定义范围内，提升集群资源利用率。

注意事项

如果集群使用节点虚拟化（如虚拟节点），请在为虚拟机配置超售比之前，先在集群/命名空间层面关闭超售。

操作步骤

1. 进入项目管理，导航至命名空间 > 命名空间列表。
2. 点击目标*命名空间名称*。
3. 点击操作 > 更新超售比。
4. 选择合适的超售比配置方式，为命名空间设置 CPU 或内存超售比。

参数	说明
继承集群配置	• 命名空间继承集群的超售比。 • 示例：若集群 CPU/内存超售比为 4，则命名空间默认为 4。 • 容器请求 = 限制 / 集群超售比。 • 若未设置限制，则使用命名空间的默认容器配额。
自定义	• 设置命名空间专属的超售比（整数且大于 1）。 • 示例：集群超售比为 4，命名空间超售比为 2 → 请求 = 限制 / 2。 • 留空表示禁用该命名空间的超售。

5. 点击更新。

注意：更改仅对新创建的 Pod 生效，已有 Pod 保持原有请求，直到重建。

管理 Namespace 成员

目录

- 导入成员
  - 约束与限制
  - 前提条件
  - 操作步骤
- 添加成员
  - 操作步骤
- 移除成员
  - 操作步骤

导入成员

平台支持批量将成员导入到 Namespace，并分配 Namespace Administrator 或 Developer 等角色以授予相应权限。

约束与限制

- 成员只能从该 Namespace 所属项目的 项目成员 中导入。
- 平台不支持导入系统默认创建的管理员用户或当前激活用户。

前提条件

要将用户导入为 Namespace 成员，必须先将其添加到该 Namespace 所属的项目中。

操作步骤

1. 进入 项目管理，点击包含待导入成员的 *项目名称*。
2. 导航至 **Namespaces** > **Namespaces**。
3. 点击待导入成员所在的 ***Namespace*** *名称*。
4. 在 **Namespace Members** 标签页，点击 导入成员。
5. 按照以下操作步骤，将列表中的全部或部分用户导入到 Namespace。

> **TIP**
>
> 可通过对话框右上角的下拉框选择用户组，并在用户名搜索框中输入用户名进行模糊搜索。

- 将列表中所有用户作为 Namespace 成员导入，并批量分配角色。

  1. 点击对话框底部 设置角色 项右侧的下拉框，选择要分配的角色名称。
  2. 点击 全部导入。

- 从列表中导入一个或多个用户作为 Namespace 成员。

  1. 勾选用户名/显示名前的复选框，选择一个或多个用户。
  2. 点击对话框底部 设置角色 项右侧的下拉框，选择要分配给所选用户的角色名称。
  3. 点击 导入。

添加成员

当平台已添加 OICD 类型的 IDP 后，可以将 OIDC 用户添加为 Namespace 成员。

您可以将符合输入要求的有效 OIDC 账号添加为 Namespace 角色，并为用户分配相应的 Namespace 角色。

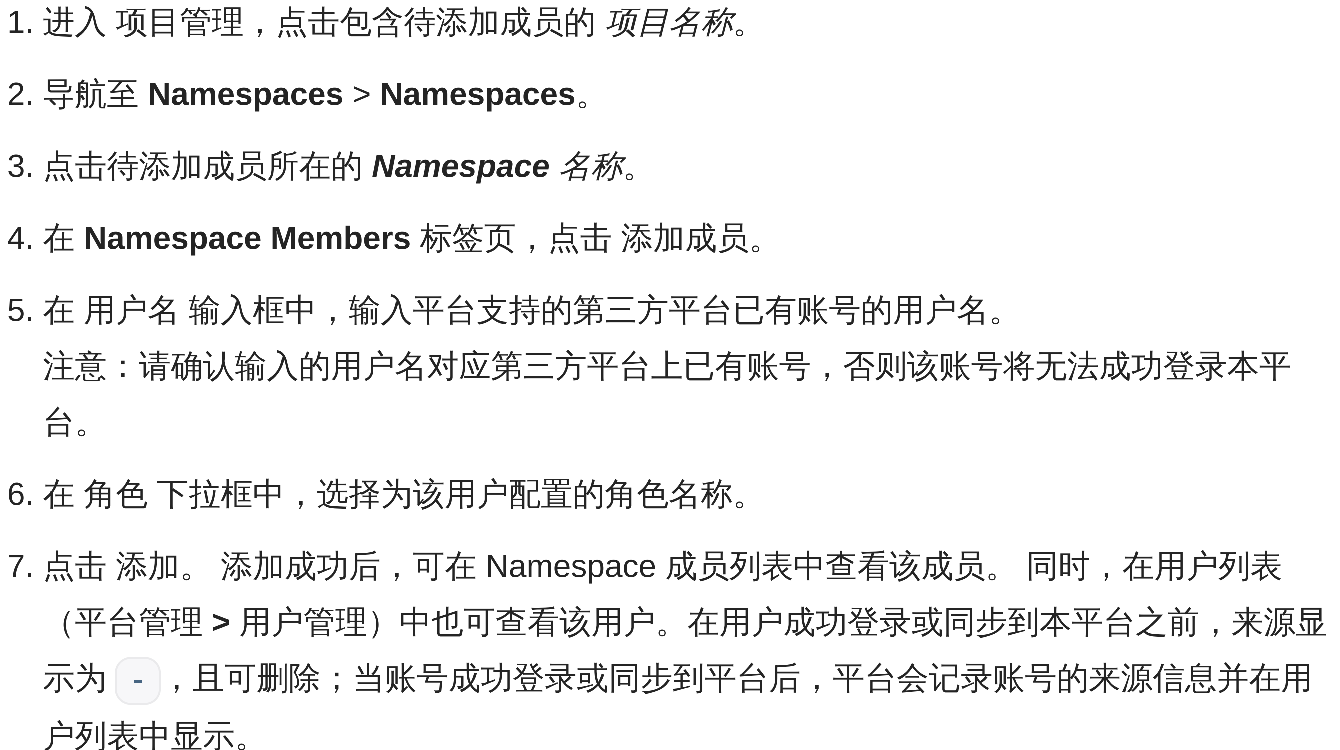
注意：添加成员时，系统不会验证账号的有效性，请确保所添加的账号有效，否则这些账号将无法成功登录平台。

有效的 **OIDC** 账号包括：通过平台配置的 IDP 的 OIDC 身份认证服务中有效的账号，包括已成功登录平台和未登录平台的账号。

前提条件

平台已添加 **OICD** 类型的 IDP。

操作步骤



1. 进入 项目管理，点击包含待添加成员的 *项目名称*。
2. 导航至 **Namespaces** > **Namespaces**。
3. 点击待添加成员所在的 ***Namespace*** *名称*。
4. 在 **Namespace Members** 标签页，点击 添加成员。
5. 在 用户名 输入框中，输入平台支持的第三方平台已有账号的用户名。
   注意：请确认输入的用户名对应第三方平台上已有账号，否则该账号将无法成功登录本平台。
6. 在 角色 下拉框中，选择为该用户配置的角色名称。
7. 点击 添加。 添加成功后，可在 Namespace 成员列表中查看该成员。 同时，在用户列表（平台管理 **>** 用户管理）中也可查看该用户。在用户成功登录或同步到本平台之前，来源显示为 `-`，且可删除；当账号成功登录或同步到平台后，平台会记录账号的来源信息并在用户列表中显示。

移除成员

移除指定的 Namespace 成员并删除其关联角色，以撤销其 Namespace 权限。

操作步骤

1. 进入 项目管理，点击包含待移除成员的 *项目名称*。
2. 导航至 **Namespaces** > **Namespaces**。
3. 点击待移除成员所在的 ***Namespace*** *名称*。
4. 在 **Namespace Members** 标签页，点击待移除成员记录右侧的⋮> 移除。
5. 点击 移除。

更新命名空间

目录

- 更新配额
  - 通过 Web 控制台更新资源配额
  - 通过 CLI 更新资源配额
- 更新容器 LimitRanges
  - 通过 Web 控制台更新 LimitRange
  - 通过 CLI 更新 LimitRange
- 更新 Pod Security Admission
  - 通过 Web 控制台更新 Pod Security Admission
  - 通过 CLI 更新 Pod Security Admission

更新配额

> Resource Quota

通过 Web 控制台更新资源配额

1. 进入 **Project Management**，在左侧边栏导航至 **Namespaces** > **Namespace** 列表。
2. 点击目标 ***namespace name***。
3. 点击 **Actions** > **Update Quota**。
4. 调整资源配额（CPU、Memory、Pods 等），然后点击 **Update**。

通过 CLI 更新资源配额

> Resource Quota YAML file example

```
# 第一步：编辑命名空间配额
kubectl edit resourcequota <quota-name> -n <namespace-name>

# 第二步：验证更改
kubectl get resourcequota <quota-name> -n <namespace-name> -o yaml
```

更新容器 LimitRanges

> Limit Range

通过 Web 控制台更新 LimitRange

1. 进入 **Project Management** 视图，在左侧边栏导航至 **Namespaces** > **Namespace** 列表。
2. 点击目标 ***namespace name***。
3. 点击 **Actions** > **Update Container LimitRange**。
4. 调整容器限制范围（`defaultRequest`、`default`、`max`），然后点击 **Update**。

通过 CLI 更新 LimitRange

> Limit Range YAML file example

```
# 第一步：编辑 LimitRange
kubectl edit limitrange <limitrange-name> -n <namespace-name>

# 第二步：验证更改
kubectl get limitrange <limitrange-name> -n <namespace-name> -o yaml
```

更新 Pod Security Admission

> Pod Security Admission

通过 Web 控制台更新 Pod Security Admission

1. 进入 **Project Management** 视图，在左侧边栏导航至 **Namespaces** > **Namespace** 列表。
2. 点击目标 ***namespace name***。
3. 点击 **Actions** > **Update Pod Security Admission**。
4. 调整安全标准（`enforce`、`audit`、`warn`），然后点击 **Update**。

通过 CLI 更新 Pod Security Admission

> Update Pod Security Admission CLI command

删除/移除命名空间

您可以选择永久删除命名空间，或将其从当前项目中移除。

目录

删除命名空间
移除命名空间

删除命名空间

删除命名空间：永久删除命名空间及其内所有资源（例如 Pods、Services、ConfigMaps）。此操作不可撤销，并会释放分配的资源配额。

```
kubectl delete namespace <namespace-name>
```

移除命名空间

移除命名空间：将命名空间从当前项目中移除，但不删除其资源。命名空间仍保留在集群中，可通过导入命名空间导入到其他项目。

NOTE

- 此功能仅限于 灵雀云容器平台。

- Kubernetes 原生不支持从项目中“移除”命名空间。

```
kubectl label namespace <namespace-name> cpaas.io/project- --overwrite
```

创建应用

Creating applications from Imag

前提条件

操作步骤 1 - 工作负载

操作步骤 2 - 服务

操作步骤 3 - Ingress

应用管理操作

参考信息

Creating applications from Cha

注意事项

前提条件

操作步骤

状态分析参考

通过 YAML

注意事项

前提条件

操作步骤

Creating ap

Creating applications from Operator Backed

Creating applications by using CLI

Prerequisites

Procedure

Example

Reference

Creating applications from Image

目录

- 前提条件
- 操作步骤 1 - 工作负载
  - 工作负载 1 - 配置基本信息
  - 工作负载 2 - 配置 Pod
  - 工作负载 3 - 配置容器
- 操作步骤 2 - 服务
- 操作步骤 3 - Ingress
- 应用管理操作
- 参考信息
  - 存储卷挂载说明
  - 健康检查参数
    - 通用参数
    - 协议特定参数

前提条件

获取镜像地址。镜像来源可以是平台管理员通过工具链集成的镜像仓库，也可以是第三方平台的镜像仓库。

- 对于前者，管理员通常会将镜像仓库分配给您的项目，您可以使用其中的镜像。如果找不到所需的镜像仓库，请联系管理员进行分配。

- 如果是第三方平台的镜像仓库，请确保当前集群可以直接拉取该镜像。

操作步骤 1 - 工作负载

1. 在 **Container Platform** 中，左侧边栏导航至 **Applications** > **Applications**。
2. 点击 **Create**。
3. 选择 **Create from Image** 作为创建方式。
4. 选择或输入镜像，点击 **Confirm**。

> **INFO**
>
> 注意：使用来自 Web 控制台集成的镜像仓库中的镜像时，可以通过 **Already Integrated** 过滤镜像。集成项目名称，例如 images (registry-projectname)，其中包含该 Web 控制台中的项目名 projectname 以及镜像仓库中的项目名 containers。
>
> 使用私有镜像仓库的镜像时，需要配置对应的镜像拉取凭证。详情请参见 为 ServiceAccount 添加 ImagePullSecrets。

6. 参考以下说明配置相关参数。

工作负载 1 - 配置基本信息

在 **Workload** > **Basic Info** 部分，配置工作负载的声明式参数

参数	说明
Model	根据需要选择工作负载类型： • **Deployment**：详细参数说明请参见 创建 Deployment。 • **DaemonSet**：详细参数说明请参见 创建 DaemonSet。 • **StatefulSet**：详细参数说明请参见 创建 StatefulSet。
Replicas	定义 Deployment 中 Pod 副本的期望数量（默认：`1`）。根据工作负载需求进行调整。

参数	说明
More > **Update Strategy**	配置 `rollingUpdate` 策略，实现零停机部署： 最大扩容数 (`maxSurge`)： • 更新期间允许超过期望副本数的最大 Pod 数量。 • 支持绝对值（如 `2` ）或百分比（如 `20%` ）。 • 百分比计算方式：`ceil(当前副本数 × 百分比)`。 • 示例：10 个副本时，4.1 → `5`。 最大不可用数 (`maxUnavailable`)： • 更新期间允许不可用的最大 Pod 数量。 • 百分比值不可超过 `100%`。 • 百分比计算方式：`floor(当前副本数 × 百分比)`。 • 示例：10 个副本时，4.9 → `4`。 注意事项： 1. 默认值：未显式设置时，`maxSurge=1`，`maxUnavailable=1`。 2. 非运行状态的 **Pod**（如 `Pending` / `CrashLoopBackOff`）视为不可用。 3. 同时限制： • `maxSurge` 和 `maxUnavailable` 不能同时为 `0` 或 `0%`。 • 若两者百分比均计算为 `0`，Kubernetes 会强制设置 `maxUnavailable=1` 以保证更新进度。 示例： 对于 10 个副本的 Deployment： • `maxSurge=2` → 更新期间总 Pod 数为 `10 + 2 = 12`。 • `maxUnavailable=3` → 最小可用 Pod 数为 `10 - 3 = 7`。 • 确保在允许受控滚动的同时保持可用性。

工作负载 2 - 配置 Pod

注意：在混合架构集群中部署单架构镜像时，请确保为 Pod 调度配置合适的 节点亲和规则。

1. 在 **Pod** 部分，配置容器运行时参数及生命周期管理：

参数	说明
Volumes	挂载持久卷到容器。支持的卷类型包括 `PVC`、`ConfigMap`、`Secret`、`emptyDir`、`hostPath` 等。具体实现详情请参见 存储卷挂载说明。
Image Credential	仅在从第三方镜像仓库（通过手动输入镜像 URL）拉取镜像时必需。 注意：平台集成的镜像仓库镜像会自动继承相关凭证。
More > **Close Grace Period**	Pod 接收到终止信号后允许的优雅关闭时间（默认：`30s`）。 - 在此期间，Pod 会完成正在处理的请求并释放资源。 - 设置为 `0` 会强制立即删除（发送 SIGKILL），可能导致请求中断。

2. 节点亲和规则

参数	说明
More > **Node Selector**	限制 Pod 调度到具有特定标签的节点（例如 `kubernetes.io/os: linux`）。 Node Selector: acp.cpaas.io/node-group-share-mode:Share × Found 1 matched nodes in current cluster
More > **Affinity**	基于已有 Pod 定义细粒度调度规则。 **Pod** 亲和类型： • **Pod** 亲和：将新 Pod 调度到托管特定 Pod 的节点（同拓扑域）。 • **Pod** 反亲和：避免新 Pod 与特定 Pod 共置于同一节点。 执行模式： • **RequiredDuringSchedulingIgnoredDuringExecution**：仅当规则满足时调度 Pod。 • **PreferredDuringSchedulingIgnoredDuringExecution**：优先选择满足规则的节点，但允许例外。

参数	说明
	配置字段： • `topologyKey`：定义拓扑域的节点标签（默认：`kubernetes.io/hostname`）。 • `labelSelector`：通过标签查询过滤目标 Pod。

3. 网络配置

- Kube-OVN

参数	说明
带宽限制	对 Pod 网络流量实施 QoS： • 出站速率限制：最大出站流量速率（例如 `10Mbps`）。 • 入站速率限制：最大入站流量速率。
子网	从预定义的子网池分配 IP。如未指定，使用命名空间默认子网。
静态 **IP** 地址	绑定持久 IP 地址到 Pod： • 多个 Deployment 中的 Pod 可声明相同 IP，但同一时间仅允许一个 Pod 使用。 • 关键：静态 IP 数量必须 ≥ Pod 副本数。

- Calico

参数	说明
静态 **IP** 地址	分配固定 IP，严格唯一： • 每个 IP 在集群中只能绑定给一个 **Pod**。 • 关键：静态 IP 数量必须 ≥ Pod 副本数。

工作负载 3 - 配置容器

1. 在 **Container** 部分，参考以下说明配置相关信息。

参数	说明
资源请求与限制	• **Requests**：容器运行所需的最小 CPU/内存。 • **Limits**：容器运行时允许的最大 CPU/内存。单位定义请参见 资源单位。 命名空间超售比： • 无超售比： 若存在命名空间资源配额，容器请求/限制继承命名空间默认值（可修改）。 无命名空间配额时，无默认值，自定义请求。 • 有超售比： 请求自动计算为 `Limits / 超售比`（不可修改）。 约束条件： • 请求 ≤ 限制 ≤ 命名空间配额最大值。 • 超售比变更需重建 Pod 生效。 • 超售比启用时禁用手动请求配置。 • 无命名空间配额时无容器资源限制。
扩展资源	配置集群可用的扩展资源（如 vGPU、pGPU）。
卷挂载	持久存储配置。详见 存储卷挂载说明。 操作： • 已有 Pod 卷：点击 **Add** • 无 Pod 卷：点击 **Add & Mount** 参数： • `mountPath`：容器文件系统路径（如 `/data`）

参数	说明
	• `subPath`：卷内相对文件/目录路径。 对于 `ConfigMap` / `Secret`：选择具体键 • `readOnly`：只读挂载（默认读写） 详见 Kubernetes 卷。
端口	暴露容器端口。 示例：暴露 TCP 端口 `6379`，名称为 `redis`。 字段： • `protocol`：TCP/UDP • `Port`：暴露端口（如 `6379`） • `name`：符合 DNS 规范的标识符（如 `redis`）
启动命令与参数	覆盖默认 ENTRYPOINT/CMD： 示例 **1**：执行 `top -b` - **Command**：`["top", "-b"]` - 或 Command：`["top"]`，Args：`["-b"]` 示例 **2**：输出 `$MESSAGE`： `/bin/sh -c "while true; do echo $(MESSAGE); sleep 10; done"` 详见 定义命令。
More > 环境变量	• 静态值：直接键值对 • 动态值：引用 ConfigMap/Secret 键、Pod 字段（`fieldRef`）、资源指标（`resourceFieldRef`） 注意：环境变量会覆盖镜像或配置文件中的设置。
More > 引用 **ConfigMap**	将整个 ConfigMap/Secret 注入为环境变量。支持的 Secret 类型：`Opaque`、`kubernetes.io/basic-auth`。
More > 健康检查	• 存活探针：检测容器健康（失败则重启）

参数	说明
	• 就绪探针：检测服务可用性（失败则从端点移除） 详见 健康检查参数。
More > 日志文件	配置日志路径： - 默认：收集 `stdout` - 文件模式：如 `/var/log/*.log` 要求： • 存储驱动 `overlay2`：默认支持 • `devicemapper`：需手动挂载 EmptyDir 到日志目录 • Windows 节点：确保父目录已挂载（如 `c:/a` 对应 `c:/a/b/c/*.log`）
More > 排除日志文件	排除特定日志收集（如 `/var/log/aaa.log`）。
More > 停止前执行命令	容器终止前执行命令。 示例：`echo "stop"` 注意：命令执行时间必须短于 Pod 的 `terminationGracePeriodSeconds`。

2. 点击右上角 **Add Container** 或 **Add Init Container**。
   参见 Init Containers ↗。 Init Container：
   1. 在应用容器启动前执行（顺序执行）。
   2. 完成后释放资源。
   3. 允许删除条件：
      - Pod 中有多个应用容器且至少有一个 Init Container。
      - 单应用容器 Pod 不允许删除 Init Container。
3. 点击 **Create**。

操作步骤 2 - 服务

参数	说明
Service	Kubernetes **Service**，为集群中运行的应用暴露单一外部访问端点，即使工作负载分布在多个后端。具体参数说明请参见 创建 Service。 注意：应用下创建的内部路由默认名称前缀为计算组件名称。若计算组件类型（部署模式）为 StatefulSet，建议不要更改内部路由（工作负载）默认名称，否则可能导致工作负载访问异常。

操作步骤 3 - Ingress

参数	说明
Ingress	Kubernetes **Ingress**，通过协议感知的配置机制，使 HTTP（或 HTTPS）网络服务可用，支持 URI、主机名、路径等 Web 概念。Ingress 允许基于 Kubernetes API 定义的规则将流量映射到不同后端。详细参数说明请参见 创建 Ingress。 注意：应用下创建 Ingress 时，所使用的 **Service** 必须是当前应用下创建的资源，且确保该 **Service** 关联应用下的工作负载，否则工作负载的服务发现和访问将失败。

7. 点击 **Create**。

应用管理操作

修改应用配置时，可使用以下任一方式：

1. 点击应用列表右侧的竖向省略号（⋮）。
2. 在应用详情页右上角选择 **Actions**。

操作	说明
Update	• 更新：仅修改目标工作负载，使用其定义的 更新策略（以 Deployment 策略为例），保留现有副本数和滚动配置。

<table>
<tr><th>操作</th><th>说明</th></tr>
<tr><td></td><td>

强制更新：触发全应用滚动更新，使用各组件的更新策略。
1. 适用场景：

批量配置变更需立即在集群内传播（如作为环境变量引用的 ConfigMap/Secret 更新）。
关键安全更新需协调组件重启。

2. 警告注意：

大规模重启可能导致短暂服务降级。
生产环境使用前需验证业务连续性。

网络影响：

Ingress 规则删除：若满足条件，外部访问仍通过 <code>LB_IP:NodePort</code> 可用：
1) LoadBalancer Service 使用默认端口。
2) 存在引用应用组件的存活路由规则。
完全终止外部访问需删除 Service。
Service 删除：导致应用组件网络连接不可逆丢失。关联 Ingress 规则失效，尽管 API 对象仍存在。

</td></tr>
<tr><td>Delete</td><td>

级联删除：
1. 删除所有子资源，包括 Deployment、Service 和 Ingress 规则。
2. Persistent Volume Claim (PVC) 按 StorageClass 中定义的保留策略处理。
删除前检查清单：
1. 确认相关 Service 无活跃流量。
2. 确认有状态组件数据已备份。
3. 使用 <code>kubectl describe ownerReferences</code> 检查依赖资源关系。

</td></tr>
</table>

参考信息

存储卷挂载说明

<table>
<tr><th>类型</th><th>用途</th></tr>
<tr><td>Persistent Volume Claim</td><td>绑定已有的 PVC 以申请持久存储。

注意：仅可选择已绑定（关联 PV）的 PVC。未绑定 PVC 会导致 Pod 创建失败。</td></tr>
<tr><td>ConfigMap</td><td>将完整或部分 ConfigMap 数据挂载为文件：
完整 ConfigMap：在挂载路径下创建以键名命名的文件
子路径选择：挂载特定键（如 <code>my.cnf</code>）</td></tr>
<tr><td>Secret</td><td>将完整或部分 Secret 数据挂载为文件：
完整 Secret：在挂载路径下创建以键名命名的文件
子路径选择：挂载特定键（如 <code>tls.crt</code>）</td></tr>
<tr><td>Ephemeral Volumes</td><td>集群动态提供的临时卷，具备：
动态供应
生命周期与 Pod 绑定
支持声明式配置
使用场景：临时数据存储。详见 临时卷</td></tr>
<tr><td>Empty Directory</td><td>Pod 内容器间共享的临时存储：
- Pod 启动时在节点创建
- Pod 删除时清理

使用场景：容器间文件共享、临时数据存储。详见 EmptyDir</td></tr>
<tr><td>Host Path</td><td>挂载宿主机目录（必须以 <code>/</code> 开头，如 <code>/volumepath</code>）。</td></tr>
</table>

健康检查参数

通用参数

参数	说明
Initial Delay	探针启动前的宽限时间（秒）。默认值：`300`。
Period	探针间隔时间（1-120秒）。默认值：`60`。
Timeout	探针超时时间（1-300秒）。默认值：`30`。
Success Threshold	标记为健康所需的连续成功次数。默认值：`0`。
Failure Threshold	触发动作的最大连续失败次数： - `0`：禁用失败触发动作 - 默认：连续失败 5 次触发容器重启。

协议特定参数

参数	适用协议	说明
Protocol	HTTP/HTTPS	健康检查协议
Port	HTTP/HTTPS/TCP	探测目标容器端口
Path	HTTP/HTTPS	端点路径（如 `/healthz`）
HTTP Headers	HTTP/HTTPS	自定义请求头（添加键值对）
Command	EXEC	容器可执行的检查命令（如 `sh -c "curl -I localhost:8080 | grep OK"`）。 注意：需转义特殊字符并测试命令有效性。

Creating applications from Chart

基于 Helm Chart 代表了一种原生应用的部署模式。Helm Chart 是一组定义 Kubernetes 资源的文件集合，旨在打包应用并支持版本控制的应用分发。这使得环境切换变得无缝，例如从开发环境迁移到生产环境。

目录

注意事项

前提条件

操作步骤

状态分析参考

注意事项

当集群中同时包含 Linux 和 Windows 节点时，必须配置明确的节点选择以防止调度冲突。示例：

```
spec:
    spec:
      nodeSelector:
        kubernetes.io/os: linux
```

前提条件

如果模板来自某个应用并引用了相关资源（例如 secret 字典），请确保待引用的资源在当前命名空间中已存在，方可进行应用部署。

操作步骤

1. 在 **Container Platform** 中，导航至左侧边栏的 **Applications** > **Applications**。
2. 点击 **Create**。
3. 选择 **Create from Catalog** 作为创建方式。
4. 选择一个 Chart 并配置参数，选择 Chart 后配置所需参数，如 `resources.requests`、`resources.limits` 以及与 Chart 紧密相关的其他参数。
5. 点击 **Create**。

网页控制台将跳转至 **Application** > [**Native Applications**] 详情页。该过程需要一定时间，请耐心等待。如操作失败，请根据界面提示完成操作。

状态分析参考

点击 *应用名称* 可显示 Chart 的详细状态分析信息。

类型	原因

Initialized	表示 Chart 模板下载状态。
	• **True**：表示 Chart 模板已成功下载。
	• **False**：表示 Chart 模板下载失败；可在消息栏查看具体失败原因。
	  ◦ `ChartLoadFailed`：Chart 模板下载失败。
	  ◦ `InitializeFailed`：Chart 下载前初始化过程出现异常。

Validated	表示 Chart 模板的用户权限、依赖关系及其他校验状态。 • **True**：表示所有校验均通过。 • **False**：表示存在未通过的校验；可在消息栏查看具体失败原因。

类型	原因
	• `DependenciesCheckFailed`：Chart 依赖检查失败。 • `PermissionCheckFailed`：当前用户缺少对某些资源的操作权限。 • `ConsistentNamespaceCheckFailed`：通过原生应用模板部署应用时，Chart 包含需要跨命名空间部署的资源。
Synced	表示 Chart 模板的部署状态。 • **True**：表示 Chart 模板已成功部署。 • **False**：表示 Chart 模板部署失败；原因栏显示 `ChartSyncFailed`，可在消息栏查看具体失败原因。

WARNING

- 如果模板引用了跨命名空间资源，请联系管理员协助创建。之后，您可以正常在网页控制台进行 Chart 应用的更新和删除。
- 如果模板引用了集群级资源（例如 StorageClasses），建议联系管理员协助创建。

通过 YAML 创建应用

如果您熟悉 YAML 语法，并且更倾向于在表单或预定义模板之外定义配置，可以选择一键 YAML 创建方式。此方法可以更灵活地配置云原生应用的基础信息和资源。

目录

注意事项

前提条件

操作步骤

注意事项

当集群中同时存在 Linux 和 Windows 节点时，为防止应用调度到不兼容的节点，必须配置节点选择。例如：

```
spec:
    spec:
      nodeSelector:
        kubernetes.io/os: linux
```

前提条件

确保 YAML 中定义的镜像可以在当前集群内拉取。您可以使用 `podman pull` 命令进行验证。

操作步骤

1. 进入 **Container Platform**，导航至 **Application** > **Applications**。
2. 点击 **Create**。
3. 选择 **Create from YAML**。
4. 完成配置后点击 **Create**。
5. 可在详情页查看对应的 **Deployment**。

Creating applications from Code

通过代码创建应用是使用 Source to Image(S2I) 技术实现的。S2I 是一个自动化框架，用于直接从源代码构建容器器镜像。这种方法标准化并自动化了应用构建过程，使开发人员能够专注于源代码开发，而无需担心容器化的细节。

目录

Prerequisites
Procedure

Prerequisites

- 完成 Alauda Container Platform Builds 的安装

Procedure

1. 进入 **Container Platform**，导航至 **Application** > **Applications**。
2. 点击 **Create**。
3. 选择 **Create from Code**。
4. 有关详细的参数描述，请参见 Managing applications created from Code
5. 完成参数输入后，点击 **Create**。

6. 可在 **Detail Information** 页面查看对应的部署情况。

Creating applications from Operator Backed

目录

Understanding Operator Backed Application
核心能力
Operator Backed Application CRD
通过 Web 控制台创建 Operator Backed 应用
故障排查

Understanding Operator Backed Application

Operator 是基于 Kubernetes 自定义控制器和自定义资源定义（CRD）构建的扩展机制，旨在自动化复杂应用的完整生命周期管理。在 Alauda Container Platform 中，Operator Backed Application 指通过预集成或用户自定义的 Operator 创建的应用实例，其操作流程由 Operator Lifecycle Manager (OLM) 管理，涵盖安装、升级、依赖关系解析和访问控制等关键流程。

核心能力

1. 复杂操作自动化：Operator 克服了 Kubernetes 原生资源（如 Deployment、StatefulSet）在管理有状态应用时的局限性，解决分布式协调、持久存储和版本滚动升级等复杂问题。例如：Operator 编码的逻辑可实现数据库集群故障切换、跨节点数据一致性和备份恢复的自主操作。

2. 声明式、状态驱动架构：Operator 通过基于 YAML 的声明式 API 定义期望的应用状态（如 spec.replicas: 5），持续调和实际状态与声明状态，实现自愈能力。与 GitOps 工具（如 Argo CD）深度集成，确保环境配置一致性。
3. 智能生命周期管理：
   - 滚动升级与回滚：OLM 的 Subscription 对象订阅更新通道（如 stable、alpha），触发 Operator 及其管理应用的自动版本迭代。
   - 依赖关系解析：Operator 动态识别运行时依赖（如特定存储驱动、CNI 插件），确保部署成功。
4. 标准化生态集成：OLM 标准化 Operator 包（Bundle）和分发渠道，实现从 OperatorHub 或私有仓库一键部署生产级应用（如 Etcd）。企业增强：Alauda Container Platform 扩展 RBAC 策略和多集群分发能力，满足企业合规需求。

Operator Backed Application CRD

该 Operator 设计与实现充分借鉴开源社区标准和方案，其自定义资源定义（CRD）设计融合了 Kubernetes 生态中成熟的最佳实践和架构模式。CRD 设计参考资料：

1. CatalogSource ↗ ：定义集群可用的 Operator 包来源，如 OperatorHub 或自定义 Operator 仓库。
2. ClusterServiceVersion (CSV) ↗ ：Operator 的核心元数据定义，包含名称、版本、提供的 API、所需权限、安装策略及详细生命周期管理信息。
3. InstallPlan ↗ ：安装 Operator 的实际执行计划，由 OLM 根据 Subscription 和 CSV 自动生成，详细列出创建 Operator 及其依赖资源的具体步骤。
4. OperatorGroup ↗ ：定义 Operator 提供服务并调和资源的目标命名空间集合，同时限制 Operator 的 RBAC 权限范围。
5. Subscription ↗ ：用于声明用户希望在集群中安装和跟踪的特定 Operator，包括 Operator 名称、目标通道（如 stable、alpha）和更新策略。OLM 通过 Subscription 创建和管理 Operator 的安装及升级。

通过 Web 控制台创建 Operator Backed 应用

1. 在 **Container Platform** 中，左侧导航栏进入 **Applications** > **Applications**。

2. 点击 **Create**。
3. 选择 **Create from Catalog** 作为创建方式。
4. 选择一个 Operator Backed 实例并配置 **Custom Resource Parameters**。选择 Operator 管理的应用实例，并在 CR 清单中配置其自定义资源（CR）规格，包括：
   - `spec.resources.limits`（容器级资源限制）。
   - `spec.resourceQuota`（Operator 定义的配额策略）。以及其他 CR 相关参数，如 `spec.replicas`、`spec.storage.className` 等。
5. 点击 **Create**。

Web 控制台将跳转至 **Applications** > **Operator Backed Apps** 页面。

> **INFO**
>
> 注意： Kubernetes 资源创建过程需要异步调和，完成时间视集群状态可能需数分钟。

故障排查

若资源创建失败：

1. 查看控制器调和错误：

```
kubectl get events --field-selector involvedObject.kind=<Your-Custom-Resource> --sort-by=.metadata.creationTimestamp
```

2. 验证 API 资源是否可用：

```
kubectl api-resources | grep <Your-Resource-Type>
```

3. 在确认 CRD/Operator 就绪后重试创建：

```
kubectl apply -f your-resource-manifest.yaml
```

Creating applications by using CLI

`kubectl` 是与 Kubernetes 集群交互的主要命令行接口（CLI）。它作为 Kubernetes API Server 的客户端——一个 RESTful HTTP API，作为控制平面的编程接口。所有 Kubernetes 操作都通过 API 端点暴露，`kubectl` 本质上将 CLI 命令转换为相应的 API 请求，以管理集群资源和应用工作负载（Deployments、StatefulSets 等）。

CLI 工具通过智能解析输入的工件（镜像，或 Chart 等）来促进应用部署，并生成相应的 Kubernetes API 对象。生成的资源根据输入类型不同而有所差异：

- **Image**：直接创建 Deployment。
- **Chart**：实例化 Helm Chart 中定义的所有对象。

目录

- Prerequisites
- Procedure
- Example
  - YAML
  - kubectl commands
- Reference

Prerequisites

已安装 **Alauda Container Platform Web Terminal** 插件，并启用 web-cli 开关。

Procedure

1. 在 **Container Platform** 中，点击右下角的终端图标。
2. 等待会话初始化（1-3 秒）。
3. 在交互式 shell 中执行 kubectl 命令：

   ```
   kubectl get pods -n ${CURRENT_NAMESPACE}
   ```

4. 查看实时命令输出

Example

YAML

```yaml
# webapp.yaml
apiVersion: app.k8s.io/v1beta1
kind: Application
metadata:
  name: webapp
spec:
  componentKinds:
    - group: apps
      kind: Deployment
    - group: ""
      kind: Service
  descriptor: {}

# webapp-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
  labels:
    app: webapp
    env: prod
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
        tier: frontend
    spec:
      containers:
      - name: webapp
        image: nginx:1.25-alpine
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: "100m"
            memory: "128Mi"
          limits:
            cpu: "250m"
```

```yaml
            memory: "256Mi"
---
# webapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  selector:
    app: webapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

kubectl commands

```
kubectl apply -f webapp.yaml -n {CURRENT_NAMESPACE}
kubectl apply -f webapp-deployment.yaml -n {CURRENT_NAMESPACE}
kubectl apply -f webapp-service.yaml -n {CURRENT_NAMESPACE}
```

Reference

- **Conceptual Guide**: kubectl Overview
- **Syntax Reference**: kubectl Cheat Sheet
- **Command Manual**: kubectl Commands

应用的操作与维护

Application Rollout

安装 Alauda Container Platform

前提条件

安装 Alauda Container Platform Argo Rollo

原生应用蓝绿发布

蓝绿发布的优势

使用 Argo Rollouts 进行蓝绿发布

前提条件

操作步骤

应用灰度发布

灰度发布的优势

使用 Argo Rollo

前提条件

操作步骤

状态说明

状态说明

原生应用

Deployment

配置 HPA

配置 HPA

了解水平 Pod 自动扩缩器

前提条件

创建水平 Pod 自动扩缩器

计算规则

启动和停止原生应用

启动和停止原生应用

启动原生应用

停止原生应用

配置 VerticalPodAutoscaler (VPA)

配置 VerticalPodAutoscaler (VPA)

了解 VerticalPodAutoscalers

前提条件

创建 VerticalPodAutoscaler

后续操作

配置 CronHPA

配置 CronHPA

了解 Cron Horizontal Pod Autoscaler

前提条件

创建 Cron Horizontal Pod Autoscaler

调度规则说明

更新原生应用

更新原生应用

导入资源

移除/批量移除资源

导出应用

导出应用

导出 Helm Chart

导出 YAML 到本地

导出 YAML 到代码仓库（Alpha）

更新和删除 **Chart** 应用

更新和删除 **Chart** 应用

重要说明

前提条件

状态分析说明

应用版本管理

应用版本管理

创建版本快照

回滚到历史版本

删除原生应用

删除原生应用

处理资源耗尽错误

处理资源耗尽错误

Overview

配置驱逐策略

在节点配置中创建驱逐策略

驱逐信号

驱逐阈值

配置可调度资源

防止节点状态振荡

回收节点级资源

Pod 驱逐

服务质量与内存杀手（OOM Killer）

调度器与资源耗尽状态

示例场景

推荐实践

健康检查

健康检查

理解健康检查

YAML 文件示例

通过 Web 控制台配置健康检查参数

探针失败排查

Application Rollout

安装 Alauda Container Platform

前提条件

安装 Alauda Container Platform Argo Rollo

原生应用蓝绿发布

蓝绿发布的优势

使用 Argo Rollouts 进行蓝绿发布

前提条件

操作步骤

应用灰度发

灰度发布的优势

使用 Argo Rollo

前提条件

操作步骤

安装 Alauda Container Platform Argo Rollouts

目录

前提条件

安装 Alauda Container Platform Argo Rollouts

操作步骤

前提条件

1. 下载 与您的平台架构对应的 **Alauda Container Platform Argo Rollouts** 集群插件安装包。
2. 使用 Upload Packages 机制上传安装包。
3. 使用集群插件机制安装安装包到集群。

> **INFO**
>
> Upload Packages： 进入 **Administrator** > **Marketplace** > **Upload Packages** 页面。 点击右侧的 **Help Document** 获取如何发布集群插件到集群的操作说明。更多详情请参考 CLI。

安装 Alauda Container Platform Argo Rollouts

操作步骤

2. 点击 **Marketplace** > **Cluster Plugins**，进入 **Cluster Plugins** 列表页面。
3. 找到 **Alauda Container Platform Argo Rollouts** 集群插件，点击 **Install**，进入 **Install Alauda Container Platform Argo Rollouts Plugin** 页面。
4. 直接点击 **Install** 即可完成 **Alauda Container Platform Argo Rollouts** 集群插件的安装。

Alauda Container Platform

原生应用蓝绿发布

在现代软件开发中，部署应用的新版本是开发周期中的关键部分。然而，将更新发布到生产环境可能存在风险，因为即使是很小的问题也可能导致显著的停机时间和收入损失。蓝绿发布是一种部署策略，它通过确保应用的新版本可以零停机时间部署来缓解这种风险。

蓝绿发布是一种部署策略，其中会设置两个相同的环境："蓝色"环境和"绿色"环境。蓝色环境是生产环境，应用的当前运行版本就在这里运行；绿色环境是非生产环境，新版本的应用会部署到这里。

当应用的新版本准备好部署时，它会被部署到绿色环境中。新版本部署并测试完成后，流量会切换到绿色环境，使其成为新的生产环境。随后，蓝色环境会变为非生产环境，未来版本的应用可以部署到这里。

目录

蓝绿发布的优势
使用 Argo Rollouts 进行蓝绿发布
前提条件
操作步骤
- 创建 Deployment
- 创建 Blue Service
- 验证 Blue Deployment
- 验证到 Blue 的流量路由
- 创建 Rollout
- 验证 Rollouts
- 准备 Green Deployment
- 将 Rollout 晋升到 Green

蓝绿发布的优势

- 零停机时间：蓝绿发布允许新版本的应用以零停机时间部署，因为流量会无缝地从蓝色环境切换到绿色环境。
- 易于回滚：如果应用的新版本出现问题，回滚到前一个版本很容易，因为蓝色环境仍然可用。
- 降低风险：通过使用蓝绿发布，部署应用新版本的风险会显著降低。这是因为新版本可以先在绿色环境中部署和测试，然后再将流量从蓝色环境切换过来。这使得可以进行充分测试，并减少生产环境中出现问题的可能性。
- 提高可靠性：通过使用蓝绿发布，应用的可靠性会提高。这是因为蓝色环境始终可用，而且绿色环境中的任何问题都可以被快速识别和解决，而不会影响用户。
- 灵活性：蓝绿发布为部署过程提供了灵活性。一个应用的多个版本可以并排部署，便于进行测试和实验。

使用 **Argo Rollouts** 进行蓝绿发布

Argo Rollouts 是一个 Kubernetes 控制器和一组 CRD，可为 Kubernetes 提供蓝绿发布、金丝雀发布、金丝雀分析、实验以及渐进式交付等高级部署能力。

Argo Rollouts 可以（可选地）与 ingress controllers 和 service meshes 集成，利用它们的流量整形能力，在更新期间逐步将流量切换到新版本。此外，Rollouts 还可以查询并解释来自各种提供方的指标，以验证关键 KPI，并在更新期间驱动自动晋升或回滚。

借助 Argo Rollouts，你可以在 Alauda Container Platform（ACP）集群上自动化蓝绿发布。典型流程包括：

1. 定义 Rollout 资源来管理不同的应用版本。
2. 配置 Kubernetes Service，在蓝色（当前）和绿色（新）环境之间路由流量。
3. 将新版本部署到绿色环境。
4. 验证和测试新版本。
5. 通过切换流量，将绿色环境晋升为生产环境。

这种方式可以最大限度地减少停机时间，并实现受控且安全的发布。

> 关键概念：
>
> - **Rollout**：Kubernetes 中的一个自定义资源定义（CRD），用于替代标准的 Deployment 资源，从而实现蓝绿发布、金丝雀发布等高级部署控制。

前提条件

1. ACP（Alauda Container Platform）。
2. 由 ACP 管理的 Kubernetes 集群。
3. 集群中已安装 Argo Rollouts。
4. Argo Rollouts kubectl 插件。
5. 一个用于创建 namespace 的 project。
6. 集群中用于部署应用的 namespace。

操作步骤

1 创建 Deployment

首先定义应用的“蓝色”版本。这是用户当前访问的版本。创建一个 Kubernetes Deployment，并设置合适的副本数、容器镜像版本（例如 `hello:1.23.1`）以及正确的标签，例如 `app=web`。

使用以下 YAML：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: hello:1.23.1
          ports:
            - containerPort: 80
```

YAML 字段说明：

- `apiVersion`：用于创建该资源的 Kubernetes API 版本。
- `kind`：指定这是一个 Deployment 资源。
- `metadata.name`：Deployment 的名称。
- `spec.replicas`：期望的 pod 副本数量。
- `spec.selector.matchLabels`：定义 Deployment 如何查找要管理的 pod。
- `template.metadata.labels`：应用到 pod 上的标签，由 Service 用于选择它们。
- `spec.containers`：在每个 pod 中运行的容器。
- `containers.name`：容器名称。
- `containers.image`：要运行的容器镜像。
- `containers.ports.containerPort`：容器暴露的端口。

使用 `kubectl` 应用该配置：

```
kubectl apply -f deployment.yaml
```

这会设置生产环境。

另外，你也可以使用 Helm Chart 来创建 Deployment 和 Service。

2 创建 Blue Service

创建一个 Kubernetes `Service` 来暴露蓝色 Deployment。该 Service 会根据匹配标签将流量转发到蓝色 pod。最初，Service 的 selector 目标是带有 `app=web` 标签的 pod。

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

YAML 字段说明：

- `apiVersion`：用于创建该 Service 的 Kubernetes API 版本。
- `kind`：指定该资源是一个 Service。
- `metadata.name`：Service 的名称。
- `spec.selector`：根据标签识别要路由流量的 pod。
- `ports.protocol`：使用的协议（TCP）。
- `ports.port`：Service 暴露的端口。
- `ports.targetPort`：流量被转发到容器上的端口。

使用以下命令应用它：

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

这使外部可以访问蓝色 Deployment。

3 验证 Blue Deployment

通过列出 pod 来确认 `blue` Deployment 运行正常：

```
kubectl get pods -l app=web
```

检查所有预期的副本数（2）是否都处于 `Running` 状态。这可以确保应用已准备好提供流量服务。

4 验证到 Blue 的流量路由

确保 `web` Service 正确地将流量转发到蓝色 Deployment。使用以下命令：

```
kubectl describe service web | grep Endpoints
```

输出应列出蓝色 pod 的 IP 地址。这些就是接收流量的端点。

5 创建 Rollout

接下来，创建使用 `BlueGreen` 策略的 Argo Rollouts `Rollout` 资源。

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: rollout-bluegreen
spec:
  replicas: 2
  revisionHistoryLimit: 2
  selector:
    matchLabels:
      app: web
  workloadRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web
    scaleDown: onsuccess
  strategy:
    blueGreen:
      activeService: web
      autoPromotionEnabled: false
```

YAML 字段说明：

- `spec.selector`：pod 的标签选择器。当前选中的 pod 所属的现有 ReplicaSet 都会受此 Rollout 影响。它必须与 pod 模板的标签匹配。
- `workloadRef`：指定要应用到 Rollout 的工作负载引用和缩容策略。
  - `scaleDown`：指定在迁移到 Rollout 后，是否缩减工作负载（Deployment）。可选项如下：
    - `"never"`：Deployment 不会被缩减。
    - `"onsuccess"`：当 Rollout 变为 healthy 后，Deployment 会被缩减。
    - `"progressively"`：随着 Rollout 扩容，Deployment 会同步缩容。如果 Rollout 失败，Deployment 会重新扩容。
- `strategy`：Rollout 策略，支持 `BlueGreen` 和 `Canary` 策略。
  - `blueGreen`：`BlueGreen` Rollout 策略定义。
    - `activeService`：指定在晋升时使用新模板哈希更新的 Service。此字段是 blueGreen 更新策略的必填项。

- `autoPromotionEnabled`：通过在晋升前立即暂停 Rollout 来禁用新版本的自动晋升。如果省略，则默认行为是在 ReplicaSet 完全 ready/available 后立即晋升新版本。可使用以下命令恢复 Rollout：`kubectl argo rollouts promote ROLLOUT`

使用以下命令应用它：

```
kubectl apply -f rollout.yaml
```

这会为该 Deployment 设置使用 `BlueGreen` 策略的 Rollouts。

6 验证 Rollouts

创建 `Rollout` 后，Argo Rollouts 会使用与 Deployment 相同的模板创建一个新的 ReplicaSet。当新 ReplicaSet 的 pod 处于 healthy 状态时，Deployment 会被缩减为 0。

使用以下命令确保 pod 运行正常：

```
kubectl argo rollouts get rollout rollout-bluegreen
Name:            rollout-bluegreen
Namespace:       default
Status:          ✔ Healthy
Strategy:        BlueGreen
Images:          hello:1.23.1 (stable, active)
Replicas:
  Desired:       2
  Current:       2
  Updated:       2
  Ready:         2
  Available:     2

NAME                                        KIND        STATUS
AGE  INFO
⟳ rollout-bluegreen                          Rollout     ✔ Health
y  95s
└──# revision:1
   └──⧉ rollout-bluegreen-595d4567cc          ReplicaSet  ✔ Healthy
18s  stable,active
      ├──□ rollout-bluegreen-595d4567cc-mc769  Pod         ✔ Running
8s   ready:1/1
      └──□ rollout-bluegreen-595d4567cc-zdc5x  Pod         ✔ Running
8s   ready:1/1
```

Service `web` 会将流量转发到 Rollouts 创建的 pod。使用以下命令：

```
kubectl describe service web | grep Endpoints
```

7 准备 Green Deployment

接下来，将应用的新版本准备为绿色部署。使用新的镜像版本更新 Deployment `web`（例如 `hello:1.23.2`）。

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: hello:1.23.2
          ports:
            - containerPort: 80
```

YAML 字段说明：

- 与原始 Deployment 相同，唯一的区别是：
  - `containers.image`：更新为新的镜像版本。

使用以下命令应用它：

```
kubectl apply -f deployment.yaml
```

这会为测试设置新版本的应用。

Rollouts 会创建一个新的 ReplicaSet 来管理绿色 pod，而流量仍然会转发到蓝色 pod。使用以下命令进行验证：

```
kubectl argo rollouts get rollout rollout-bluegreen
Name:             rollout-bluegreen
Namespace:        default
Status:           ॥ Paused
Message:          BlueGreenPause
Strategy:         BlueGreen
Images:           hello:1.23.1 (stable, active)
                  hello:1.23.2
Replicas:
  Desired:        2
  Current:        4
  Updated:        2
  Ready:          2
  Available:      2

NAME                                             KIND        STATUS
AGE  INFO
⟳ rollout-bluegreen                               Rollout     ॥ Paused
14m
├──# revision:2
│  └──⧉ rollout-bluegreen-776b688d57             ReplicaSet  ✔ Healthy
24s
│     ├──□ rollout-bluegreen-776b688d57-kxr66  Pod         ✔ Running
23s  ready:1/1
│     └──□ rollout-bluegreen-776b688d57-vv7t7  Pod         ✔ Running
23s  ready:1/1
└──# revision:1
   └──⧉ rollout-bluegreen-595d4567cc             ReplicaSet  ✔ Healthy
12m  stable,active
      ├──□ rollout-bluegreen-595d4567cc-mc769  Pod         ✔ Running
12m  ready:1/1
      └──□ rollout-bluegreen-595d4567cc-zdc5x  Pod         ✔ Running
12m  ready:1/1
```

目前有 4 个 pod 在运行，分别是蓝色和绿色版本。而 active service 仍然指向蓝色版本，Rollout 流程处于暂停状态。

如果你使用 Helm Chart 来部署应用，请使用 helm 工具将应用升级到绿色版本。

8 将 Rollout 晋升到 Green

当绿色版本准备就绪后，晋升 Rollout 以将流量切换到绿色 pod。使用以下命令：

```
kubectl argo rollouts promote rollout-bluegreen
```

要验证 Rollout 是否已完成：

```
kubectl argo rollouts get rollout rollout-bluegreen
Name:              rollout-bluegreen
Namespace:         default
Status:            ✔ Healthy
Strategy:          BlueGreen
Images:            hello:1.23.2 (stable, active)
Replicas:
  Desired:         2
  Current:         2
  Updated:         2
  Ready:           2
  Available:       2

NAME                                           KIND          STATUS
AGE   INFO
⟳ rollout-bluegreen                             Rollout       ✔ Health
y       3h2m
├──# revision:2
│  └──⧉ rollout-bluegreen-776b688d57             ReplicaSet   ✔ Healthy
168m  stable,active
│     ├──□ rollout-bluegreen-776b688d57-kxr66  Pod          ✔ Running
168m  ready:1/1
│     └──□ rollout-bluegreen-776b688d57-vv7t7  Pod          ✔ Running
168m  ready:1/1
└──# revision:1
   └──⧉ rollout-bluegreen-595d4567cc             ReplicaSet   • ScaledD
own   3h1m
      ├──□ rollout-bluegreen-595d4567cc-mc769  Pod          ◌ Terminat
ing  3h    ready:1/1
      └──□ rollout-bluegreen-595d4567cc-zdc5x  Pod          ◌ Terminat
ing  3h    ready:1/1
```

如果 active `Images` 已更新为 `hello:1.23.2`，并且蓝色 ReplicaSet 已缩减到 0，则表示 Rollout 已完成。

应用灰度发布

Canary Deployment 是一种渐进式发布策略，它会将新的应用版本逐步引入到一小部分用户或流量中。这种增量式发布方式允许团队在全面部署之前监控系统行为、收集指标并确保稳定性。该方法能够显著降低风险，尤其是在生产环境中。

Argo Rollouts 是一个 Kubernetes 原生的渐进式交付控制器，可支持高级部署策略。它通过提供 Canary、Blue-Green Deployments、Analysis Runs、Experimentation 和 Automated Rollbacks 等功能扩展了 Kubernetes 能力。它与可观测性栈集成，用于基于指标的健康检查，并提供基于 CLI 和监控面板的应用交付控制。

> 关键概念：
>
> - **Rollout**：Kubernetes 中的一个 Custom Resource Definition（CRD），用于替代标准的 Deployment 资源，从而支持 blue-green、canary deployment 等高级部署控制。
> - **Canary Steps**：一系列增量式流量切换操作，例如先将 25% 的流量，再将 50% 的流量导向新版本。
> - **Pause Steps**：在进入下一个 canary step 之前引入等待间隔，以便进行手动或自动验证。

目录

灰度发布的优势

使用 Argo Rollouts 进行灰度发布

前提条件

操作步骤

- 创建 Deployment
- 创建 Stable Service

创建 Canary Service

创建 Gateway

DNS 配置

创建 HTTPRoute

访问 Stable Service

创建 Rollout

验证 Rollout

准备 Canary Deployment

晋级 Rollout

中止 Rollout（可选）

灰度发布的优势

- 风险缓解：通过先将变更部署到少量服务器上，你可以在全面发布前发现并处理问题，从而将对用户的影响降到最低。
- 增量式发布：这种方式允许逐步暴露新功能，有助于你有效监控性能和用户反馈。
- 实时反馈：灰度发布可以在真实环境条件下，立即提供新版本性能和稳定性的洞察。
- 灵活性：你可以根据性能指标调整部署过程。这使得你能够按需暂停或回滚，实现动态发布。
- 成本效益：与 blue/green deployments 不同，canary deployments 不需要单独的环境，因此资源利用率更高。

使用 **Argo Rollouts** 进行灰度发布

Argo Rollouts 支持 canary deployment 策略来发布 Deployment，并通过 Gateway API Plugin 控制流量。在 ACP 中，你可以使用 ALB 作为 Gateway API Provider 来实现 Argo Rollouts 的流量控制。

前提条件

1. 集群中已安装带有 Gateway API plugin 的 Argo Rollouts。
2. Argo Rollouts kubectl plugin（可从 here 安装）。
3. 一个用于创建 namespace 的 project。
4. 集群中已部署 ALB，并已分配给该 project。
5. 集群中一个用于部署应用的 namespace。

操作步骤

1 创建 Deployment

首先定义应用的“stable”版本。这是用户当前访问的版本。创建一个 Kubernetes Deployment，并设置合适的副本数、容器镜像版本（例如 `hello:1.23.1`）以及正确的标签，例如 `app=web`。

使用以下 YAML：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: hello:1.23.1
          ports:
            - containerPort: 80
```

YAML 字段说明：

- `apiVersion`：用于创建该资源的 Kubernetes API 版本。
- `kind`：指定这是一个 Deployment 资源。
- `metadata.name`：Deployment 的名称。
- `spec.replicas`：期望的 Pod 副本数。
- `spec.selector.matchLabels`：定义 Deployment 如何查找要管理的 Pod。
- `template.metadata.labels`：应用到 Pod 的标签，供 Service 选择。
- `spec.containers`：每个 Pod 中运行的容器。
- `containers.name`：容器名称。
- `containers.image`：要运行的容器镜像。
- `containers.ports.containerPort`：容器暴露的端口。

使用 `kubectl` 应用配置：

```
kubectl apply -f deployment.yaml
```

这会设置生产环境。

另外，你也可以使用 Helm Chart 来创建这些 deployments 和 services。

2 创建 Stable Service

创建一个 Kubernetes `Service` 来暴露 stable deployment。该 Service 将根据匹配的标签，将流量转发到 stable 版本的 Pod。初始时，Service selector 目标是带有 `app=web` 标签的 Pod。

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-stable
spec:
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

YAML 字段说明：

- `apiVersion`：用于创建该 Service 的 Kubernetes API 版本。
- `kind`：指定该资源是一个 Service。
- `metadata.name`：Service 的名称。
- `spec.selector`：根据标签标识要将流量路由到哪些 Pod。
- `ports.protocol`：使用的协议（TCP）。
- `ports.port`：Service 暴露的端口。
- `ports.targetPort`：流量被转发到容器上的端口。

使用以下命令应用：

```
kubectl apply -f web-stable-service.yaml
```

这样即可从集群外访问 stable deployment。

3 创建 Canary Service

创建一个 Kubernetes `Service` 来暴露 canary deployment。该 Service 将根据匹配的标签，将流量转发到 canary 版本的 Pod。初始时，Service selector 目标是带有 `app=web` 标签的 Pod。

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-canary
spec:
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

YAML 字段说明：

- `apiVersion`：用于创建该 Service 的 Kubernetes API 版本。
- `kind`：指定该资源是一个 Service。
- `metadata.name`：Service 的名称。
- `spec.selector`：根据标签标识要将流量路由到哪些 Pod。
- `ports.protocol`：使用的协议（TCP）。
- `ports.port`：Service 暴露的端口。
- `ports.targetPort`：流量被转发到容器上的端口。

使用以下命令应用：

```
kubectl apply -f web-canary-service.yaml
```

这样即可从集群外访问 canary deployment。

4 创建 Gateway

使用 `example.com` 作为访问服务的域名，创建 Gateway 以通过该域名暴露服务：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: default
spec:
  gatewayClassName: exclusive-gateway
  listeners:
  - allowedRoutes:
      namespaces:
        from: All
    name: gateway-metric
    port: 11782
    protocol: TCP
  - allowedRoutes:
      namespaces:
        from: All
    hostname: example.com
    name: web
    port: 80
    protocol: HTTP
```

使用以下命令：

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

Gateway 将分配一个外部 IP 地址，请从 Gateway 资源中 `status.addresses` 里类型为 `IPAddress` 的条目获取该 IP 地址。

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: default
...
status:
  addresses:
  - type: IPAddress
    value: 192.168.134.30
```

5 DNS 配置

在你的 DNS 服务器中配置该域名，将域名解析到 Gateway 的 IP 地址。使用以下命令验证 DNS 解析：

```
nslookup example.com
Server:         192.168.16.19
Address:        192.168.16.19#53

Non-authoritative answer:
Name:   example.com
Address: 192.168.134.30
```

它应该返回 Gateway 的地址。

6 创建 HTTPRoute

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: web
spec:
  hostnames:
  - example.com
  parentRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: default
    namespace: default
    sectionName: web
  rules:
  - backendRefs:
    - group: ""
      kind: Service
      name: web-canary
      namespace: default
      port: 80
      weight: 0
    - group: ""
      kind: Service
      name: web-stable
      namespace: default
      port: 80
      weight: 100
    matches:
    - path:
        type: PathPrefix
        value: /
```

使用以下命令：

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

7 访问 Stable Service

在集群外，使用以下命令通过域名访问服务：

```
curl http://example.com
```

或者，你也可以在浏览器中访问 `http://example.com`。

8 创建 Rollout

接下来，创建 Argo Rollouts 的 `Rollout` 资源，并使用 `Canary` 策略。

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: rollout-canary
spec:
  minReadySeconds: 30
  replicas: 2
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: web
  strategy:
    canary:
      canaryService: web-canary
      maxSurge: 25%
      maxUnavailable: 0
      stableService: web-stable
      steps:
      - setWeight: 50
      - pause: {}
      - setWeight: 100
      trafficRouting:
        plugins:
          argoproj-labs/gatewayAPI:
            httpRoute: web
            namespace: default
  workloadRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web
    scaleDown: onsuccess
```

YAML 字段说明：

- `spec.selector`：Pod 的标签选择器。此选择器所选中的现有 ReplicaSet 将受到该 Rollout 的影响。它必须与 Pod 模板的标签匹配。
- `workloadRef`：指定 workload 引用以及要应用于 Rollout 的缩容策略。
- `scaleDown`：指定在迁移到 Rollout 之后，workload（Deployment）是否缩容。可选项如下：
  - `"never"`：Deployment 不会缩容。
  - `"onsuccess"`：当 Rollout 变为健康状态后，Deployment 会缩容。
  - `"progressively"`：随着 Rollout 扩容，Deployment 会同步缩容。如果 Rollout 失败，Deployment 将重新扩容。
- `strategy`：Rollout 策略，支持 `BlueGreen` 和 `Canary` 策略。
- `canary`：`Canary` Rollout 策略定义。
  - `canaryService`：指向一个 Service 的引用，控制器将更新该 Service 以选择 canary Pod。流量路由时必填。
  - `stableService`：指向一个 Service 的引用，控制器将更新该 Service 以选择 stable Pod。流量路由时必填。
  - `steps`：定义更新 canary 时要执行的一系列步骤。首次部署 Rollout 时会跳过这些步骤。
    - `setWeight`：设置 canary ReplicaSet 的流量比例。
    - `pause`：无限期或按指定时长暂停 Rollout。支持的单位：s、m、h。`{}` 表示无限期。
    - `plugin`：执行已配置的插件，此处配置的是 `gatewayAPI` 插件。

使用以下命令应用：

```
kubectl apply -f rollout.yaml
```

这会为该 Deployment 设置 `Canary` 策略的 Rollout。它会先将权重设为 50，并等待晋级。50% 的流量将转发到 canary service。完成 Rollout 晋级后，权重将设为 100，100% 的流量将转发到 canary service。最终，canary service 将成为 stable service。

9 验证 Rollout

创建 `Rollout` 后，Argo Rollouts 会基于 Deployment 的相同模板创建一个新的 ReplicaSet。当新 ReplicaSet 的 Pod 处于健康状态时，Deployment 会缩容到 0。

使用以下命令确保 Pod 运行正常：

```
kubectl argo rollouts get rollout rollout-canary
Name:             rollout-canary
Namespace:        default
Status:           ✔ Healthy
Strategy:         Canary
Step:           9/9
SetWeight:      100
ActualWeight:   100
Images:           hello:1.23.1 (stable)
Replicas:
Desired:        2
Current:        2
Updated:        2
Ready:          2
Available:      2

NAME                                          KIND        STATUS     AGE
INFO
⟳ rollout-canary                                 Rollout      ✔ Healthy
32s
└──# revision:1
  └──⧉ rollout-canary-5c9d79697b           ReplicaSet  ✔ Healthy   32
s  stable
     ├──□ rollout-canary-5c9d79697b-fh78d  Pod         ✔ Running   32s
ready:1/1
     └──□ rollout-canary-5c9d79697b-rrbtj  Pod         ✔ Running   32s
ready:1/1
```

10 准备 Canary Deployment

接下来，将应用的新版本准备为 green deployment。使用新的镜像版本更新 Deployment `web` （例如 `hello:1.23.2` ）。使用以下命令：

```
kubectl patch deployment web -p '{"spec":{"template":{"spec":{"containers":[{"name":"web","image":"hello:1.23.2"}]}}}}'
```

这会为测试准备新的应用版本。

Rollouts 会创建一个新的 ReplicaSet 来管理 canary Pod，并且 50% 的流量会转发到 canary Pod。使用以下命令进行验证：

```
kubectl argo rollouts get rollout rollout-canary
Name:              rollout-canary
Namespace:         default
Status:            ॥ Paused
Message:           CanaryPauseStep
Strategy:          Canary
Step:            1/3
SetWeight:       50
ActualWeight:    50
Images:            hello:1.23.1 (stable)
                 hello:1.23.2 (canary)
Replicas:
Desired:         2
Current:         3
Updated:         1
Ready:           3
Available:       3

NAME                                          KIND        STATUS     AGE
INFO
⟳ rollout-canary                                 Rollout      ॥ Paused   9
5s
├──# revision:2
│  └──⧉ rollout-canary-5898765588           ReplicaSet  ✔ Healthy   4
6s  canary
│     └──□ rollout-canary-5898765588-ls5jk  Pod         ✔ Running   4
5s  ready:1/1
└──# revision:1
   └──⧉ rollout-canary-5c9d79697b           ReplicaSet  ✔ Healthy   95
s  stable
      ├──□ rollout-canary-5c9d79697b-fk269  Pod         ✔ Running   94s
ready:1/1
      └──□ rollout-canary-5c9d79697b-wkmcn  Pod         ✔ Running   94s
ready:1/1
```

当前有 3 个 Pod 在运行，分别是 stable 和 canary 版本。此时权重为 50，50% 的流量会转发到 canary service。Rollout 流程已暂停，等待晋级。

如果你使用 Helm Chart 部署应用，则使用 Helm 工具将应用升级到 canary 版本。

访问 `http://example.com` 时，50% 的流量将转发到 canary service。你应该会从该 URL 收到不同的响应。

11 晋级 Rollout

当 canary 版本测试通过后，你可以晋级 Rollout，将所有流量切换到 canary Pod。使用以下命令：

```
kubectl argo rollouts promote rollout-canary
```

验证 Rollout 是否完成：

```
kubectl argo rollouts get rollout rollout-canary
Name:            rollout-canary
Namespace:       default
Status:          ✔ Healthy
Strategy:        Canary
Step:          3/3
SetWeight:     100
ActualWeight:  100
Images:          hello:1.23.2 (stable)
Replicas:
Desired:       2
Current:       2
Updated:       2
Ready:         2
Available:     2

NAME                                          KIND        STATUS
AGE    INFO
⟳ rollout-canary                                Rollout     ✔ Healthy
8m42s
├──# revision:2
│  └──⧉ rollout-canary-5898765588           ReplicaSet  ✔ Healthy
7m53s  stable
│     ├──□ rollout-canary-5898765588-ls5jk  Pod         ✔ Running
7m52s  ready:1/1
│     └──□ rollout-canary-5898765588-dkfwg  Pod         ✔ Running
68s    ready:1/1
└──# revision:1
   └──⧉ rollout-canary-5c9d79697b           ReplicaSet  • ScaledDown
8m42s
      ├──□ rollout-canary-5c9d79697b-fk269  Pod         ◌ Terminating
8m41s  ready:1/1
      └──□ rollout-canary-5c9d79697b-wkmcn  Pod         ◌ Terminating
8m41s  ready:1/1
```

如果 stable 的 `Images` 已更新为 `hello:1.23.2`，并且 revision 1 的 ReplicaSet 已缩容为 0，则说明 Rollout 已完成。

访问 `http://example.com` 时，100% 的流量将转发到 canary service。

12 中止 Rollout（可选）

如果你在 Rollout 过程中发现 canary 版本存在问题，可以中止该流程，将所有流量切换回 stable service。使用以下命令：

```
kubectl argo rollouts abort rollout-canary
```

验证结果：

```
kubectl argo rollouts get rollout rollout-canary
Name:            rollout-demo
Namespace:       default
Status:          ✖ Degraded
Message:         RolloutAborted: Rollout aborted update to revision 3
Strategy:        Canary
Step:          0/3
SetWeight:     0
ActualWeight:  0
Images:          hello:1.23.1 (stable)
Replicas:
Desired:       2
Current:       2
Updated:       0
Ready:         2
Available:     2

NAME                                     KIND        STATUS        A
GE  INFO
⟳ rollout-canary                            Rollout       ✖ Degraded
18m
├──# revision:3
│  └──⧉ rollout-canary-5c9d79697b           ReplicaSet  • ScaledDown
18m  canary,delay:passed
└──# revision:2
   └──⧉ rollout-canary-5898765588           ReplicaSet  ✔ Healthy
17m  stable
      ├──□ rollout-canary-5898765588-ls5jk  Pod         ✔ Running
17m  ready:1/1
      └──□ rollout-canary-5898765588-dkfwg  Pod         ✔ Running
10m  ready:1/1
```

访问 `http://example.com` 时，100% 的流量将转发到 stable service。

状态说明

目录

原生应用
Deployment

原生应用

原生应用的状态及其对应含义如下。状态后面的数字表示计算组件的数量。

状态	含义
运行中	所有计算组件均正常运行。
部分运行中	部分计算组件正在运行，而其他计算组件已停止。
已停止	所有计算组件均已停止。
处理中	至少有一个计算组件处于待处理状态。
无计算组件	该应用下没有计算组件。
失败	部署失败。

注意：同样，计算组件状态中的数字表示容器组的数量。

Deployment

- Running：所有 Pod 均正常运行。
- Processing：存在未处于运行状态的 Pod。
- Stopped：所有 Pod 已停止。
- Failed：Deployment 失败。

配置 HPA

HPA（Horizontal Pod Autoscaler，水平 Pod 自动扩缩器）根据预设的策略和指标，自动上下调整 Pod 的数量，使应用能够应对突发的业务流量峰值，同时在低流量时段优化资源利用率。

目录

- 了解水平 Pod 自动扩缩器
  - HPA 是如何工作的？
  - 支持的指标
- 前提条件
- 创建水平 Pod 自动扩缩器
  - 使用 CLI
  - 使用 Web 控制台
  - 使用自定义指标进行 HPA
    - 需求
    - 传统（核心指标）HPA
    - 自定义指标 HPA
    - 触发条件定义
    - 自定义指标 HPA 兼容性
    - autoscaling/v2beta2 的更新
- 计算规则

了解水平 Pod 自动扩缩器

您可以创建一个水平 Pod 自动扩缩器，指定希望运行的 Pod 最小和最大数量，以及 Pod 应达到的 CPU 利用率或内存利用率目标。

创建水平 Pod 自动扩缩器后，平台开始查询 Pod 上的 CPU 和/或内存资源指标。当这些指标可用时，水平 Pod 自动扩缩器计算当前指标利用率与期望指标利用率的比值，并据此进行扩缩容。查询和扩缩容操作以固定间隔进行，但指标可用通常需要一到两分钟。

对于 replication controller，此扩缩容直接对应 replication controller 的副本数。对于 deployment 配置，扩缩容直接对应 deployment 配置的副本数。注意，自动扩缩仅适用于处于 Complete 阶段的最新 deployment。

平台会自动考虑资源情况，避免在资源峰值（如启动时）期间进行不必要的自动扩缩。处于未就绪状态的 Pod 在扩容时视为 0 CPU 使用率，缩容时会被忽略。无已知指标的 Pod 在扩容时视为 0% CPU 使用率，缩容时视为 100% CPU 使用率。这样可以使 HPA 决策更稳定。要使用此功能，必须配置 readiness 检查以判断新 Pod 是否已准备好使用。

HPA 是如何工作的？

水平 Pod 自动扩缩器（HPA）扩展了 Pod 自动扩缩的概念。HPA 允许您创建和管理一组负载均衡的节点。当 CPU 或内存达到设定阈值时，HPA 会自动增加或减少 Pod 数量。

HPA 作为一个控制循环运行，默认同步周期为 15 秒。在此期间，controller manager 会根据 HPA 配置查询 CPU、内存利用率或两者。controller manager 从资源指标 API 获取每个被 HPA 目标的 Pod 的资源利用率指标，如 CPU 或内存。

如果设置了利用率目标，controller 会计算每个 Pod 容器对应资源请求的利用率百分比。然后对所有目标 Pod 的利用率取平均，生成一个比率，用于调整期望副本数。

支持的指标

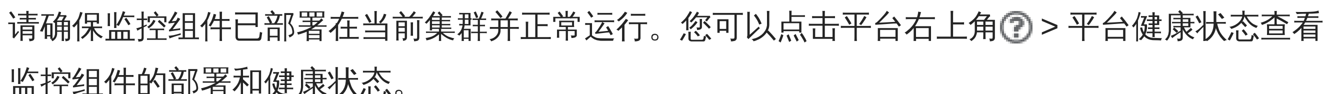
水平 Pod 自动扩缩器支持以下指标：

指标	描述
CPU 利用率	使用的 CPU 核数。可用于计算 Pod 请求 CPU 的百分比。
内存利用率	使用的内存量。可用于计算 Pod 请求内存的百分比。

指标	描述
网络入流量	进入 Pod 的网络流量，单位为 KiB/s。
网络出流量	从 Pod 发送出去的网络流量，单位为 KiB/s。
存储读取流量	从存储读取的数据量，单位为 KiB/s。
存储写入流量	写入存储的数据量，单位为 KiB/s。

> 重要提示：对于基于内存的自动扩缩，内存使用必须与副本数成比例地增减。一般来说：
>
> - 副本数增加时，每个 Pod 的内存（工作集）使用应整体下降。
> - 副本数减少时，每个 Pod 的内存使用应整体上升。
> - 请使用平台检查应用的内存行为，确保应用满足这些要求后再使用基于内存的自动扩缩。

前提条件



请确保监控组件已部署在当前集群并正常运行。您可以点击平台右上角 ? > 平台健康状态查看监控组件的部署和健康状态。

创建水平 **Pod** 自动扩缩器

使用 **CLI**

您可以通过命令行界面定义 YAML 文件并使用 `kubectl create` 命令创建水平 Pod 自动扩缩器。以下示例展示了对 Deployment 对象的自动扩缩。初始部署需要 3 个 Pod，HPA 对象将最小副本数提升到 5。如果 Pod 的 CPU 使用率达到 75%，Pod 数量将增加到 7：

1. 创建名为 `hpa.yaml` 的 YAML 文件，内容如下：

```yaml
apiVersion: autoscaling/v2 ①
kind: HorizontalPodAutoscaler ②
metadata:
  name: hpa-demo ③
  namespace: default
spec:
  maxReplicas: 7 ④
  minReplicas: 3 ⑤
  scaleTargetRef:
    apiVersion: apps/v1 ⑥
    kind: Deployment ⑦
    name: deployment-demo ⑧
  targetCPUUtilizationPercentage: 75 ⑨
```

① 使用 autoscaling/v2 API。

② HPA 资源的名称。

③ 需要扩缩的 deployment 名称。

④ 最大副本数。

⑤ 最小副本数。

⑥ 指定扩缩对象的 API 版本。

⑦ 指定对象类型，必须是 Deployment、ReplicaSet 或 StatefulSet。

⑧ HPA 应用的目标资源。

⑨ 触发扩缩的目标 CPU 利用率百分比。

2. 应用 YAML 文件创建 HPA：

```
kubectl create -f hpa.yaml
```

示例输出：

```
horizontalpodautoscaler.autoscaling/hpa-demo created
```

3. 创建 HPA 后，您可以运行以下命令查看 deployment 的新状态：

```
kubectl get deployment deployment-demo
```

示例输出：

```
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-demo   5/5     5            5           3m
```

4. 您也可以查看 HPA 的状态：

```
kubectl get hpa hpa-demo
```

示例输出：

```
NAME         REFERENCE                    TARGETS    MINPODS    MAXPODS
REPLICAS    AGE
hpa-demo    Deployment/deployment-demo   0%/75%     3          7
3           2m
```

使用 **Web** 控制台

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Workloads** > **Deployments**。
3. 点击 ***Deployment*** *名称*。
4. 向下滚动到 弹性伸缩 区域，点击右侧的 更新。
5. 选择 水平伸缩 并完成策略配置。

参数	描述
Pod 数量	部署成功后，需要评估对应已知和常规业务量变化的最小 **Pod** 数量，以及在高业务压力下命名空间配额可支持的最大 **Pod** 数量。最大或最小 Pod 数量可在设置后更改，建议先通过性能测试推导更准确的值，并在使用过程中持续调整以满足业务需求。
触发策略	列出对业务变化敏感的指标及其目标阈值，以触发扩容或缩容动作。 例如，设置 *CPU 利用率 = 60%*，一旦 CPU 利用率偏离 60%，平台将根据当前部署资源不足或过剩情况自动调整 Pod 数量。

参数	描述
	注意：指标类型包括内置指标和自定义指标。自定义指标仅适用于原生应用中的部署，且需先添加自定义指标。
扩缩步长（**Alpha**）	对于有特定扩缩速率要求的业务，可通过指定扩容步长或缩容步长逐步适应业务量变化。 缩容步长可自定义稳定窗口，默认为 300 秒，意味着必须等待 300 秒后才执行缩容操作。

6. 点击 更新。

使用自定义指标进行 HPA

自定义指标 HPA 扩展了原有的 HorizontalPodAutoscaler，支持除 CPU 和内存利用率之外的更多指标。

需求

- kube-controller-manager：horizontal-pod-autoscaler-use-rest-clients=true
- 预先安装 metrics-server
- Prometheus
- custom-metrics-api

传统（核心指标）HPA

传统 HPA 支持 CPU 利用率和内存指标动态调整 Pod 实例数，示例如下：

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-app-nginx
  namespace: test-namespace
spec:
  maxReplicas: 1
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-app-nginx
  targetCPUUtilizationPercentage: 50
```

该 YAML 中，`scaleTargetRef` 指定扩缩的工作负载对象，`targetCPUUtilizationPercentage` 指定 CPU 利用率触发指标。

自定义指标 HPA

使用自定义指标需要安装 prometheus-operator 和 custom-metrics-api。安装后，custom-metrics-api 提供大量自定义指标资源：

```
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "namespaces/go_memstats_heap_sys_bytes",
      "singularName": "",
      "namespaced": false,
      "kind": "MetricValueList",
      "verbs": ["get"]
    },
    {
      "name": "jobs.batch/go_memstats_last_gc_time_seconds",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": ["get"]
    },
    {
      "name": "pods/go_memstats_frees",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": ["get"]
    }
  ]
}
```

这些资源均为 MetricValueList 的子资源。您可以通过 Prometheus 创建规则来创建或维护子资源。自定义指标的 HPA YAML 格式与传统 HPA 不同：

```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: demo
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: demo
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Pods
      pods:
        metricName: metric-demo
        targetAverageValue: 10
```

示例中，`scaleTargetRef` 指定工作负载。

触发条件定义

- `metrics` 为数组类型，支持多个指标
- `metric type` 可为：Object（描述 k8s 资源）、Pods（描述每个 Pod 的指标）、Resources（内置 k8s 指标：CPU、内存）、External（通常为集群外部指标）
- 若自定义指标非由 Prometheus 提供，需通过在 Prometheus 创建规则等一系列操作新增指标

指标的主要结构如下：

```
{
      "describedObject": {  # 描述对象（Pod）
        "kind": "Pod",
        "namespace": "monitoring",
        "name": "nginx-788f78d959-fd6n9",
        "apiVersion": "/v1"
      },
      "metricName": "metric-demo",
      "timestamp": "2020-02-5T04:26:01Z",
      "value": "50"
}
```

该指标数据由 Prometheus 采集并更新。

自定义指标 HPA 兼容性

自定义指标 HPA YAML 实际兼容原有核心指标（CPU），写法示例如下：

```yaml
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: nginx
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: 80
    - type: Resource
      resource:
        name: memory
        targetAverageValue: 200Mi
```

- `targetAverageValue` 为每个 Pod 的平均值

- `targetAverageUtilization` 为根据直接值计算的利用率

算法参考：

```
replicas = ceil(sum(CurrentPodsCPUUtilization) / Target)
```

autoscaling/v2beta2 的更新

autoscaling/v2beta2 支持内存利用率：

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
  namespace: default
spec:
  minReplicas: 1
  maxReplicas: 3
  metrics:
    - resource:
        name: cpu
        target:
          averageUtilization: 70
          type: Utilization
      type: Resource
    - resource:
        name: memory
        target:
          averageUtilization:
          type: Utilization
      type: Resource
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
```

变化：`targetAverageUtilization` 和 `targetAverageValue` 改为 `target`，并转换为 `xxxValue` 和 `type` 的组合：

- `xxxValue`：AverageValue（平均值）、AverageUtilization（平均利用率）、Value（直接值）
- `type`：Utilization（利用率）、AverageValue（平均值）

注意：

- 对于 **CPU** 利用率 和 内存利用率 指标，只有实际值超出目标阈值 ±10% 范围时才触发自动扩缩。
- 缩容可能影响正在进行的业务，请谨慎操作。

计算规则

当业务指标变化时，平台会根据以下规则自动计算与业务量匹配的目标 Pod 数，并进行调整。如业务指标持续波动，数值会调整到设置的最小 **Pod** 数量或最大 **Pod** 数量。

- 单策略目标 Pod 数：ceil[(sum(实际指标值)/指标阈值)]。即所有 Pod 的实际指标值之和除以指标阈值后向上取整。例如：当前有 3 个 Pod，CPU 利用率分别为 80%、80%、90%，设定的 CPU 利用率阈值为 60%。根据公式，Pod 数将自动调整为：
ceil[(80%+80%+90%)/60%] = ceil 4.1 = 5 个 Pod。
注意：
  - 若计算出的目标 Pod 数超过设置的最大 **Pod** 数（例如 *4*），平台只会扩容到 4 个 Pod。若调整最大 Pod 数后指标仍持续偏高，可能需要采用其他扩缩方法，如增加命名空间 Pod 配额或添加硬件资源。
  - 若计算出的目标 Pod 数（上述示例中为 *5*）小于根据扩容步长调整后的 Pod 数（例如 *10*），平台只会扩容到 5 个 Pod。
- 多策略目标 Pod 数：取各策略计算结果中的最大值。

启动和停止原生应用

目录

启动原生应用
停止原生应用

启动原生应用

1. 访问 **Container Platform**。
2. 在左侧导航栏中，点击 **Application** > **Applications**。
3. 点击原生应用名称。
4. 点击 **Start**。

停止原生应用

1. 访问 **Container Platform**。
2. 在左侧导航栏中，点击 **Application** > **Applications**。
3. 点击原生应用名称。
4. 点击 **Stop**。
5. 阅读提示信息，确认无误后，点击 **Stop**。

配置 VerticalPodAutoscaler (VPA)

对于无状态和有状态应用，VerticalPodAutoscaler (VPA) 会根据您的业务需求自动推荐并可选地应用更合适的 CPU 和内存资源限制，确保 Pod 拥有足够的资源，同时提升集群资源利用率。

目录

- 了解 VerticalPodAutoscalers
  - VPA 是如何工作的？
  - 支持的功能
- 前提条件
  - 安装 Vertical Pod Autoscaler 插件
- 创建 VerticalPodAutoscaler
  - 使用 CLI
  - 使用 Web 控制台
  - 高级 VPA 配置
    - 更新策略选项
    - 容器策略选项
- 后续操作

了解 VerticalPodAutoscalers

您可以创建一个 VerticalPodAutoscaler，根据 Pod 的历史使用模式推荐或自动更新其 CPU 和内存资源请求与限制。

创建 VerticalPodAutoscaler 后，平台开始监控 Pod 的 CPU 和内存资源使用情况。当收集到足够数据时，VerticalPodAutoscaler 会基于观察到的使用模式计算推荐的资源值。根据配置的更新模式，VPA 可以自动应用这些推荐，或仅提供推荐供手动应用。

VPA 通过分析 Pod 的资源使用情况并基于此分析提出建议，帮助确保 Pod 拥有所需资源而不会过度配置，从而实现集群资源的更高效利用。

VPA 是如何工作的？

VerticalPodAutoscaler (VPA) 扩展了 Pod 资源优化的概念。VPA 监控 Pod 的资源使用情况，并基于观察到的使用模式提供 CPU 和内存请求的推荐。

VPA 持续监控 Pod 的资源使用情况，并随着新数据的到来不断更新其推荐。VPA 可运行于以下模式：

- **Off**：VPA 仅提供推荐，不自动应用。
- **Manual Adjustment**：您可以根据 VPA 推荐手动调整资源配置。

> 重要提示：弹性伸缩可以实现 Pod 的水平或垂直伸缩。当资源充足时，弹性伸缩效果良好，但当集群资源不足时，可能导致 Pod 处于 Pending 状态。因此，请确保集群有足够资源或合理的配额，或者配置告警以监控伸缩情况。

支持的功能

VerticalPodAutoscaler 基于历史使用模式提供资源推荐，帮助您优化 Pod 的 CPU 和内存配置。

> 重要提示：手动应用 VPA 推荐时会触发 Pod 重建，可能导致应用短暂中断。建议在生产工作负载的维护窗口期间应用推荐。

前提条件

- 请确保当前集群已部署监控组件且运行正常。您可以点击平台右上角 ⓘ > 平台健康状态 查看监控组件的部署和健康状况。
- 集群中必须安装 Alauda Container Platform Vertical Pod Autoscaler 集群插件。

安装 Vertical Pod Autoscaler 插件

使用 VPA 前，需先安装 Vertical Pod Autoscaler 集群插件：

1. 登录并进入 **Administrators** 页面。
2. 点击 **Marketplace** > **Cluster Plugins**，进入 **Cluster Plugins** 列表页面。
3. 找到 Alauda Container Platform Vertical Pod Autoscaler 集群插件，点击安装，进入安装页面。

创建 VerticalPodAutoscaler

使用 CLI

您可以通过命令行界面定义 YAML 文件并使用 `kubectl create` 命令创建 VerticalPodAutoscaler。以下示例展示了针对 Deployment 对象的垂直 Pod 自动伸缩：

1. 创建名为 `vpa.yaml` 的 YAML 文件，内容如下：

```yaml
apiVersion: autoscaling.k8s.io/v1 1
kind: VerticalPodAutoscaler 2
metadata:
  name: my-deployment-vpa 3
  namespace: default
spec:
  targetRef:
    apiVersion: apps/v1 4
    kind: Deployment 5
    name: my-deployment 6
  updatePolicy:
    updateMode: 'Off' 7
  resourcePolicy: 8
    containerPolicies:
      - containerName: '*' 9
        mode: 'Auto' 10
```

1. 使用 autoscaling.k8s.io/v1 API。
2. VPA 的名称。
3. 指定目标工作负载对象。VPA 使用工作负载的选择器查找需要调整资源的 Pod。支持的工作负载类型包括 DaemonSet、Deployment、ReplicaSet、StatefulSet、ReplicationController、Job 和 CronJob。
4. 指定要伸缩对象的 API 版本。
5. 指定对象类型。
6. VPA 应用的目标资源。
7. 定义 VPA 如何应用推荐的更新策略。updateMode 可选：
   - Auto：创建 Pod 时自动设置资源请求，并更新当前 Pod 至推荐资源请求。目前等同于“Recreate”。此模式可能导致应用停机。未来支持就地 Pod 资源更新后，“Auto” 模式将采用该更新机制。
   - Recreate：创建 Pod 时自动设置资源请求，并驱逐当前 Pod 以更新至推荐资源请求。不使用就地更新。
   - Initial：仅在创建 Pod 时设置资源请求，之后不做修改。
   - Off：不自动修改 Pod 资源请求，仅在 VPA 对象中提供推荐。
8. 资源策略，可为不同容器设置具体策略。例如，将容器模式设为“Auto”表示会计算该容器的推荐，而“Off”表示不计算推荐。

9. 应用于 Pod 中所有容器的策略。
10. 设置模式为 Auto 或 Off。Auto 表示为该容器生成推荐，Off 表示不生成推荐。

2. 应用 YAML 文件创建 VPA：

```
kubectl create -f vpa.yaml
```

示例输出：

```
verticalpodautoscaler.autoscaling.k8s.io/my-deployment-vpa created
```

3. 创建 VPA 后，可通过以下命令查看推荐：

```
kubectl describe vpa my-deployment-vpa
```

示例输出（部分）：

```
Status:
  Recommendation:
    Container Recommendations:
      Container Name:  my-container
      Lower Bound:
        Cpu:      100m
        Memory:   262144k
      Target:
        Cpu:      200m
        Memory:   524288k
      Upper Bound:
        Cpu:      300m
        Memory:   786432k
```

使用 **Web** 控制台

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Workloads** > **Deployments**。
3. 点击 ***Deployment** 名称*。
4. 向下滚动至 弹性伸缩 区域，点击右侧的 更新。

5. 选择 垂直伸缩 并配置伸缩规则。

参数	说明
伸缩模式	目前支持 手动伸缩 模式，通过分析历史资源使用情况提供推荐资源配置，您可以根据推荐值手动调整。调整会导致 Pod 重建和重启，请选择合适时间以避免影响运行中的应用。 通常 Pod 运行超过 8 天后，推荐值更为准确。 注意，当集群资源不足时，伸缩可能导致 Pod 处于 Pending 状态。请确保集群资源充足或配额合理，或配置告警监控伸缩情况。
目标容器	默认为工作负载的第一个容器。您可以根据需要选择为一个或多个容器启用资源限制推荐。

6. 点击 更新。

高级 VPA 配置

更新策略选项

- `updateMode: "Off"` - VPA 仅提供推荐，不自动应用。您可根据需要手动应用。
- `updateMode: "Auto"` - 创建 Pod 时自动设置资源请求，并更新当前 Pod 至推荐值。目前等同于“Recreate”。
- `updateMode: "Recreate"` - 创建 Pod 时自动设置资源请求，并驱逐当前 Pod 以更新至推荐值。
- `updateMode: "Initial"` - 仅在创建 Pod 时设置资源请求，之后不做修改。
- `minReplicas: <number>` - 最小副本数。确保在 Updater 驱逐 Pod 时至少保持该数量的 Pod 可用。必须大于 0。

容器策略选项

- `containerName: "*"` - 应用于 Pod 中所有容器。
- `mode: "Auto"` - 自动为容器生成推荐。
- `mode: "Off"` - 不为容器生成推荐。

注意：

- VPA 推荐基于历史使用数据，通常需要 Pod 运行数天后推荐才准确。
- 在 Auto 模式应用 VPA 推荐时会触发 Pod 重建，可能导致应用短暂中断。

后续操作

配置 VPA 后，可在 弹性伸缩 区域查看目标容器的 CPU 和内存资源限制推荐值。在 容器 区域选择目标容器标签，点击 资源限制 右侧图标，根据推荐值更新资源限制。

配置 CronHPA

对于具有周期性业务波动的无状态应用，CronHPA（Cron Horizontal Pod Autoscaler）支持根据您设置的时间策略调整 Pod 数量，使您能够根据可预测的业务模式优化资源使用。

目录

了解 Cron Horizontal Pod Autoscaler
CronHPA 如何工作？
前提条件
创建 Cron Horizontal Pod Autoscaler
使用 CLI
使用 Web 控制台
调度规则说明

了解 Cron Horizontal Pod Autoscaler

您可以创建一个 cron horizontal pod autoscaler，按照计划指定在特定时间运行的 Pod 数量，从而为可预测的流量模式做准备，或在非高峰时段减少资源使用。

创建 cron horizontal pod autoscaler 后，平台开始监控该计划，并在指定时间自动调整 Pod 数量。此基于时间的扩缩容独立于资源利用率指标，非常适合具有已知使用模式的应用。

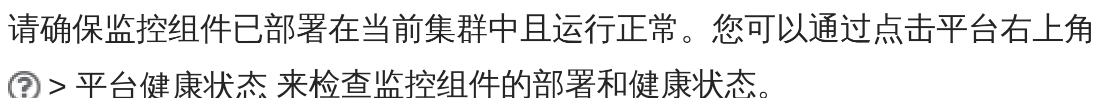
CronHPA 通过定义一个或多个调度规则来工作，每条规则指定一个时间（使用 crontab 格式）和目标 Replica 数量。当达到调度时间时，CronHPA 会将 Pod 数量调整为指定的目标，无论当前资源利用情况如何。

CronHPA 如何工作？

cron horizontal pod autoscaler（CronHPA）扩展了基于 Pod 自动扩缩容的概念，增加了基于时间的控制。CronHPA 允许您定义特定时间点调整 Pod 数量，以便为可预测的流量模式做准备，或在非高峰时段减少资源使用。

CronHPA 通过持续检查当前时间与定义的调度进行比较来工作。当达到调度时间时，控制器会将 Pod 数量调整为该调度指定的目标 Replica 数量。如果多个调度同时触发，平台将使用优先级更高的规则（即配置中定义较早的规则）。

前提条件



请确保监控组件已部署在当前集群中且运行正常。您可以通过点击平台右上角 ? > 平台健康状态 来检查监控组件的部署和健康状态。

创建 Cron Horizontal Pod Autoscaler

使用 CLI

您可以通过定义 YAML 文件并使用 `kubectl create` 命令来创建 cron horizontal pod autoscaler。以下示例展示了针对 Deployment 对象的定时扩缩容：

1. 创建名为 `cronhpa.yaml` 的 YAML 文件，内容如下：

```
apiVersion: tkestack.io/v1 ①
kind: CronHPA ②
metadata:
  name: my-deployment-cronhpa ③
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1 ④
    kind: Deployment ⑤
    name: my-deployment ⑥
  crons:
    - schedule: '0 0 * * *' ⑦
      targetReplicas: 0 ⑧
    - schedule: '0 8 * * 1-5' ⑨
      targetReplicas: 3 ⑩
    - schedule: '0 18 * * 1-5' ⑪
      targetReplicas: 1 ⑫
```

① 使用 tkestack.io/v1 API。

② CronHPA 资源的名称。

③ 需要扩缩容的 Deployment 名称。

④ 指定扩缩容对象的 API 版本。

⑤ 指定对象类型，对象必须是 Deployment、ReplicaSet 或 StatefulSet。

⑥ CronHPA 作用的目标资源。

⑦ 使用标准 crontab 格式的定时调度（分钟 小时 日 月 星期）。

⑧ 调度触发时扩缩容的目标 Replica 数量。

该示例配置了 Deployment：

- 每天午夜（00:00 ）缩容至 0 个 Replica
- 工作日（周一至周五）上午 8:00 扩容至 3 个 Replica
- 工作日（周一至周五）下午 6:00 缩容至 1 个 Replica

2. 应用 YAML 文件创建 CronHPA：

```
kubectl create -f cronhpa.yaml
```

使用 Web 控制台

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Workloads** > **Deployments**。
3. 点击 ***Deployment 名称***。
4. 向下滚动至 弹性伸缩 区域，点击右侧的 更新。
5. 选择 定时伸缩，配置伸缩规则。当类型为 自定义 时，必须提供 Crontab 表达式作为触发条件，格式为 `分钟 小时 日 月 星期`。详细介绍请参考 编写 Crontab 表达式。
6. 点击 更新。

调度规则说明

* Scaling Rules:

	Type	* Trigger Condition		* Target Replicas
1	Time	Sunday ×	01:00	1
2	Customize	0 2 * * 2		2
3	Customize	0 2 * * 2		3

Add

1. 表示从每周一凌晨 01:00 开始，仅保留 1 个 Pod。
2. 表示从每周二凌晨 02:00 开始，仅保留 2 个 Pod。
3. 表示从每周二凌晨 02:00 开始，仅保留 3 个 Pod。

重要说明：

- 当多个规则触发时间相同时（示例 2 和 3），平台仅根据优先级更高的规则执行自动扩缩容（示例 2）。
- CronHPA 独立于 HPA 运行。如果同一工作负载同时配置了两者，可能会产生冲突，请谨慎规划扩缩容策略。
- 调度使用 crontab 格式（分钟 小时 日 月 星期），规则与 Kubernetes CronJobs 相同。
- 时间基于集群时区设置。
- 对于对可用性要求较高的工作负载，请确保定时伸缩不会在高峰期意外降低容量。

更新原生应用

自定义原生应用极大地方便了对工作负载、网络、存储和配置的统一管理，但并非所有资源都属于该原生应用。

- 在创建原生应用过程中添加的资源，或通过更新原生应用添加的资源，默认均关联到该原生应用，无需额外导入。
- 在原生应用外创建的资源不属于该原生应用，且无法在原生应用详情中找到。但只要资源定义满足业务需求，业务可以正常运行。此时建议将资源导入到原生应用中进行统一管理。
- 镜像管理
  - 通过标签/补丁版本控制发布新的容器镜像
  - 配置 imagePullPolicy（Always/IfNotPresent/Never）
- 运行时配置
  - 通过 ConfigMaps/Secrets 修改环境变量
  - 更新资源请求/限制（CPU/内存）
- 资源编排
  - 导入已有的 Kubernetes 资源（Deployments/Services/Ingresses）
  - 使用 `kubectl apply -f` 跨命名空间同步配置

导入到原生应用中的资源可以享受以下功能：

功能	描述
版本快照	在为原生应用创建版本快照时，也会为原生应用内的资源生成快照。 • 如果回滚原生应用，资源也会回滚到快照中的状态。 • 如果分发原生应用的某个特定版本，平台会在重新部署原生应用时自动创建快照中记录的资源。
随原生应用删除	如果不再需要某个原生应用，删除该原生应用时会自动删除所有关联的资源，包括计算组件、内部路由和入站规则。
更易查找	在原生应用详情信息中，可以快速查看与原生应用关联的资源。 例如：外部流量可以通过属于*原生应用 A* 的 *Service S* 访问 *Deployment D*，但只有当 *Service S* 也属于*原生应用 A* 时，才能在原生应用详情中快速找到对应的访问地址。

目录

导入资源

移除/批量移除资源

导入资源

> 批量导入原生应用所在命名空间下的相关资源；一个资源只能属于一个原生应用。

1. 进入 容器平台。
2. 在左侧导航栏点击 原生应用管理 > 原生应用。
3. 点击 *原生应用名称*。
4. 点击 操作 > 管理资源。
5. 在底部的 资源类型 中，选择要导入的资源类型。

注意：常见资源类型包括 Deployment、DaemonSet、StatefulSet、Job、CronJob、Service、Ingress、PVC、ConfigMap、Secret 和 HorizontalPodAutoscaler，显示在顶部；其他资源按字母顺序排列，可通过搜索关键词快速查询特定资源类型。

6. 在 资源 区域，选择要导入的资源。
   注意：对于 **Job** 类型资源，仅支持通过 YAML 创建的任务导入。

7. 点击 导入资源。

移除/批量移除资源

> 从原生应用中移除/批量移除资源仅解除资源与原生应用的关联，不会删除资源。

如果原生应用下的资源之间存在关联，移除其中任一资源不会改变资源之间的关联关系。例如，即使将 *Service S* 从 *原生应用 A* 中移除，外部流量仍可通过 *Service S* 访问 *Deployment D*。

1. 进入 容器平台。
2. 在左侧导航栏点击 原生应用管理 > 原生应用。
3. 点击 *原生应用名称*。
4. 点击 操作 > 管理资源。
5. 点击某资源右侧的 移除 按钮进行移除；或批量选择多个资源，点击表格顶部的 移除 按钮批量移除资源。

导出应用

为了规范开发、测试和生产环境之间应用的导出流程，便于业务快速迁移到新环境，您可以将原生应用导出为应用模板（Charts），或导出可直接用于部署的简化 YAML 文件。这样可以使原生应用在不同环境或命名空间中运行。您还可以将 YAML 文件导出到代码仓库，利用 GitOps 功能快速实现跨集群应用部署。

目录

导出 Helm Chart

操作步骤

后续操作

导出 YAML 到本地

操作步骤

方式一

方式二

后续操作

导出 YAML 到代码仓库（Alpha）

注意事项

操作步骤

后续操作

导出 Helm Chart

操作步骤

1. 进入 容器平台。
2. 在左侧导航栏点击 应用管理 > 原生应用。
3. 点击类型为 `Custom Application` 的 *应用名称*。
4. 点击 操作 > 导出；也可以在应用详情页导出指定版本。
5. 根据需要选择一种导出方式，并参考以下说明配置相关信息。

- 导出 Helm Chart 到具有管理权限的模板仓库
  注意：模板仓库由平台管理员添加。请联系平台管理员获取具有 管理 权限的 **Chart** 或 **OCI Chart** 类型的有效模板仓库。

参数	说明
目标位置	选择 模板仓库，将模板直接同步到具有 管理 权限的 **Chart** 或 **OCI Chart** 类型模板仓库。分配给该 模板仓库 的项目负责人可直接使用该模板。
模板目录	当选择的模板仓库类型为 OCI Chart 时，需要选择或手动输入存放 Helm Chart 的目录。 注意：手动输入新模板目录时，平台会在模板仓库中创建该目录，但存在创建失败的风险。
版本	应用模板的版本号。 格式应为 `v<Major>.<Minor>.<Patch>`。默认值为当前应用版本或当前快照版本。
图标	支持 JPG、PNG 和 GIF 格式，文件大小不超过 500KB。建议尺寸为 80*60 像素。
描述	描述内容将在应用目录的应用模板列表中展示。
README	描述文件，支持 Markdown 格式编辑，将显示在应用模板详情页。
NOTES	模板帮助文件，支持标准纯文本编辑；部署模板完成后，将显示在模板应用详情页。

- 导出 Helm Chart 到本地，手动上传至模板仓库：选择目标位置为 本地，文件格式选择 **Helm Chart**，生成 Helm Chart 包并下载到本地，便于离线传输。

6. 点击 导出。

后续操作

- 若导出 Helm Chart 到本地，需参考添加模板到具有管理权限的模板仓库。
- 无论选择何种导出方式，均可参考创建原生应用 - 模板方式在非当前命名空间创建 `Template Application` 类型的原生应用。

导出 YAML 到本地

操作步骤

方式一

1. 进入 容器平台。
2. 在左侧导航栏点击 应用管理 > 原生应用。
3. 点击 *应用名称*。
4. 点击 操作 > 导出；也可以在应用详情页导出指定版本。
5. 选择目标位置为 本地，文件格式选择 **YAML**，即可导出可直接在其他环境部署的简化 YAML 文件。
6. 点击 导出。

方式二

1. 进入 容器平台。
2. 在左侧导航栏点击 应用管理 > 原生应用。
3. 点击 *应用名称*。
4. 点击 **YAML** 标签页，根据需要配置设置并预览 YAML 文件。

类型	说明
完整 **YAML**	默认未选中 预览简化 **YAML**，显示隐藏了 **managedFields** 字段 的 YAML 文件。您可以预览并直接导出；也可取消勾选 隐藏 **managedFields** 字段 导出完整 YAML 文件。 注意：完整 YAML 主要用于运维和排错，不能用于平台快速创建原生应用。
简化 **YAML**	勾选 预览简化 **YAML**，即可预览并导出可直接在其他环境部署的简化 YAML 文件。

5. 点击 导出。

后续操作

导出简化 YAML 后，可参考创建原生应用 - YAML 方式在非当前命名空间创建 `Custom Application` 类型的原生应用。

导出 YAML 到代码仓库（Alpha）

注意事项

- 仅平台管理员和项目管理员可直接将原生应用 YAML 文件导出到代码仓库。
- `Template Application` 不支持导出 Kustomize 格式的应用配置文件或直接导出 YAML 文件到代码仓库；您可先脱离模板，转换为 `Custom Application`。

操作步骤

1. 进入 容器平台。
2. 在左侧导航栏点击 应用管理 > 原生应用。
3. 点击类型为 `Custom` 的 *应用名称*。
4. 点击 操作 > 导出；也可以在应用详情页导出指定版本。
5. 根据需要选择一种导出方式，并参考以下说明配置相关信息。

- 导出 YAML 到代码仓库：

参数	说明
目标位置	选择 代码仓库，将 YAML 文件直接同步到指定的 Git 代码仓库。分配给该 代码仓库 的项目负责人可直接使用该 YAML 文件。
集成项目名称	由平台管理员分配或关联给您项目的集成工具项目名称。
仓库地址	集成工具项目下分配给您使用的仓库地址。
导出方式	• 现有分支：将应用 YAML 导出到所选分支。 • 新建分支：基于所选的 分支**/**标签**/**提交 **ID** 创建新分支，并将应用 YAML 导出到新分支。 　◦ 勾选 提交 **PR**（**Pull Request**） 时，平台会创建新分支并提交 Pull Request。 　◦ 勾选 合并 **PR** 后自动删除源分支 时，您在 Git 代码仓库合并 PR 后，源分支会被自动删除。
文件路径	文件在代码仓库中保存的具体位置；也可输入文件路径，平台会基于输入在代码仓库中新建路径。
提交信息	填写提交信息，用于标识本次提交内容。
预览	预览待提交的 YAML 文件，并与代码仓库中已有 YAML 进行差异对比，采用颜色区分显示。

- 导出 Kustomize 类型文件到本地，手动上传至代码仓库：选择目标位置为 本地，文件格式选择 **Kustomize**，导出 Kustomize 类型的应用配置文件。该文件支持差异化配置，适用于跨集群应用部署。

6. 点击 导出。

后续操作

导出 YAML 到 Git 代码仓库后，可参考 创建 GitOps 应用↗，跨集群创建 `Custom Application` 类型的 GitOps 应用。

更新和删除 Chart 应用

由于当前模板应用与原生应用功能存在重叠，且原生应用具备更强的运维能力，未来版本将不再提供模板应用的独立管理功能。请尽快将您当前已成功部署的模板应用升级为原生应用。

目录

重要说明
前提条件
状态分析说明

重要说明

此功能即将被废弃。请尽快将您当前已成功部署的模板应用升级为原生应用。

前提条件

请联系平台管理员以启用模板应用相关功能。

状态分析说明

点击*模板应用名称*，可在详细信息中展示该 Chart 的部署状态分析详情。

<table>
<tr><th>类型</th><th>原因</th></tr>
<tr><td>Initialized</td><td>表示 Chart 模板下载状态。
当状态为 True 时，表示 Chart 模板下载成功。
当状态为 False 时，表示 Chart 模板下载失败，失败原因可在消息列查看。
ChartLoadFailed：Chart 模板下载失败。
InitializeFailed：下载 Chart 前初始化时发生异常。</td></tr>
<tr><td>Validated</td><td>表示 Chart 模板的用户权限及依赖校验状态。
当状态为 True 时，表示所有校验均通过。
当状态为 False 时，表示存在校验未通过，失败原因可在消息列查看。
DependenciesCheckFailed：Chart 依赖校验失败。
PermissionCheckFailed：当前用户缺少某些资源操作权限。
ConsistentNamespaceCheckFailed：将模板应用作为原生应用部署时，Chart 包含需要跨命名空间部署的资源。</td></tr>
<tr><td>Synced</td><td>表示 Chart 模板部署状态。
当状态为 True 时，表示 Chart 模板部署成功。
当状态为 False 时，表示 Chart 模板部署失败，失败原因显示为 ChartSyncFailed，具体失败原因可在消息列查看。</td></tr>
</table>

应用版本管理

通过平台界面更新应用后，会自动生成历史版本记录。对于通过非界面操作触发的应用更新，例如通过 API 调用更新应用，可以手动创建版本快照以记录变更。

注意：当版本快照条目数量超过 6 条时，平台仅保留最新的 6 条，自动删除其他条目，优先删除最旧的版本快照条目。

目录

创建版本快照
- 操作步骤

回滚到历史版本
- 操作步骤

创建版本快照

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏点击 应用管理 > 原生应用。
3. 点击 *应用名称*。
4. 在 版本快照 标签页，点击 创建版本快照。
5. 配置相关信息后点击 确认。

注意：你也可以分发应用，将应用的版本快照以 Chart 形式分发，方便在平台上快速部署同一应用到多个集群和命名空间。

回滚到历史版本

将当前应用配置回滚到历史版本。

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏点击 应用管理 > 原生应用。
3. 点击 *应用名称*。
4. 在 历史版本 标签页，点击 *版本号*。
5. 点击 ⋮ > 回滚到此版本。
6. 点击 回滚。

删除原生应用

删除一个原生应用时，会同时删除该原生应用本身及其所有直接包含的 Kubernetes 资源。此外，此操作还会切断该原生应用与其他非其定义中直接包含的 Kubernetes 资源之间的任何关联。

处理资源耗尽错误

目录

Overview
配置驱逐策略
在节点配置中创建驱逐策略
驱逐信号
驱逐阈值
硬驱逐阈值
默认硬驱逐阈值
软驱逐阈值
配置可调度资源
防止节点状态振荡
回收节点级资源
Pod 驱逐
服务质量与内存杀手（OOM Killer）
调度器与资源耗尽状态
示例场景
推荐实践
DaemonSet 与资源耗尽处理

Overview

本指南介绍如何防止 Alauda Container Platform 节点出现内存（OOM）或磁盘空间耗尽的情况。节点的稳定运行至关重要，尤其是对于内存和磁盘等不可压缩资源。资源耗尽可能导致节点不稳定。

管理员可以配置驱逐策略，监控节点并在稳定性受损前回收资源。

本文档涵盖 Alauda Container Platform 如何处理资源耗尽场景，包括资源回收、Pod 驱逐、Pod 调度以及内存杀手（OOM Killer）。还提供了示例配置和最佳实践。

> **NOTE**
>
> 如果节点启用了交换内存（swap），则无法检测内存压力。请禁用 swap 以启用基于内存的驱逐。

配置驱逐策略

驱逐策略允许节点在资源不足时终止 Pod，以回收所需资源。策略结合驱逐信号和阈值，在节点配置或命令行中设置。驱逐分为：

- **Hard**（硬驱逐）：阈值超出时立即执行。
- **Soft**（软驱逐）：在宽限期后执行。

合理配置驱逐策略有助于节点主动防止资源耗尽。

> **NOTE**
>
> 当 Pod 被驱逐时，Pod 中的所有容器都会被终止，PodPhase 状态变为 Failed。

对于磁盘压力，节点监控 `nodefs`（根文件系统）和 `imagefs`（容器镜像存储）。

- **nodefs/rootfs**：用于本地磁盘卷、日志及其他存储（如 `/var/lib/kubelet`）。
- **imagefs**：容器运行时用于镜像和可写层。

> **NOTE**
>
> 如果没有本地存储隔离（临时存储）或 XFS 配额（volumeConfig），无法限制 Pod 的磁盘使用。

在节点配置中创建驱逐策略

要设置驱逐阈值，编辑节点配置映射中的 `eviction-hard` 或 `eviction-soft`。

硬驱逐示例：

```yaml
kubeletArguments:
  eviction-hard: ①
    - memory.available<100Mi ②
    - nodefs.available<10%
    - nodefs.inodesFree<5%
    - imagefs.available<15%
    - imagefs.inodesFree<10%
```

① 驱逐类型：使用 `eviction-hard` 表示硬驱逐阈值。

② 每个驱逐阈值格式为 `<eviction_signal><operator><quantity>`，例如 `memory.available<500Mi` 或 `nodefs.available<10%`。

> **NOTE**
>
> `inodesFree` 必须使用百分比值，其他参数可使用百分比或数值。

软驱逐示例：

```yaml
kubeletArguments:
  eviction-soft: ①
    - memory.available<100Mi ②
    - nodefs.available<10%
    - nodefs.inodesFree<5%
    - imagefs.available<15%
    - imagefs.inodesFree<10%
  eviction-soft-grace-period: ③
    - memory.available=1m30s
    - nodefs.available=1m30s
    - nodefs.inodesFree=1m30s
    - imagefs.available=1m30s
    - imagefs.inodesFree=1m30s
```

1. 驱逐类型：使用 `eviction-soft` 表示软驱逐阈值。
2. 每个驱逐阈值格式为 `<eviction_signal><operator><quantity>`，例如 `memory.available<500Mi` 或 `nodefs.available<10%`。
3. 软驱逐的宽限期。建议保留默认值以获得最佳性能。

修改后重启 kubelet 服务使配置生效：

```
$ systemctl restart kubelet
```

驱逐信号

节点可基于以下信号触发驱逐：

节点状态	驱逐信号	描述
MemoryPressure	memory.available	可用内存低于阈值
DiskPressure	nodefs.available	节点根文件系统空间低于阈值
	nodefs.inodesFree	空闲 inode 低于阈值
	imagefs.available	镜像文件系统空间低于阈值
	imagefs.inodesFree	imagefs 中空闲 inode 低于阈值

- `inodesFree` 必须以百分比指定。
- 内存计算不包含可回收的非活跃文件内存。
- 不要在容器内使用 `free -m` 命令。

节点每 10 秒监控一次这些文件系统。专用的卷或日志文件系统不被监控。

> **NOTE**
>
> 在因磁盘压力驱逐 Pod 之前，节点会执行容器和镜像垃圾回收。

驱逐阈值

驱逐阈值触发资源回收。当阈值达到时，节点报告压力状态，阻止新 Pod 调度，直到资源被回收。

- 硬阈值：立即执行操作。
- 软阈值：宽限期后执行操作。

阈值格式为：

```
<eviction_signal><operator><quantity>
```

示例：

- `memory.available<1Gi`
- `memory.available<10%`

节点每 10 秒评估一次阈值。

硬驱逐阈值

无宽限期，立即执行驱逐。

示例：

```
kubeletArguments:
  eviction-hard:
    - memory.available<500Mi
    - nodefs.available<500Mi
    - nodefs.inodesFree<5%
    - imagefs.available<100Mi
    - imagefs.inodesFree<10%
```

默认硬驱逐阈值

```
kubeletArguments:
  eviction-hard:
    - memory.available<100Mi
    - nodefs.available<10%
    - nodefs.inodesFree<5%
    - imagefs.available<15%
```

软驱逐阈值

软阈值需要宽限期。可选设置最大 Pod 终止宽限期（`eviction-max-pod-grace-period`）。

示例：

```
kubeletArguments:
  eviction-soft:
    - memory.available<500Mi
    - nodefs.available<500Mi
    - nodefs.inodesFree<5%
    - imagefs.available<100Mi
    - imagefs.inodesFree<10%
  eviction-soft-grace-period:
    - memory.available=1m30s
    - nodefs.available=1m30s
    - nodefs.inodesFree=1m30s
    - imagefs.available=1m30s
    - imagefs.inodesFree=1m30s
```

配置可调度资源

通过设置 `system-reserved` 为系统守护进程保留资源，控制节点可用于调度的资源量。只有当 Pod 超出请求资源时才会触发驱逐。

- **Capacity**（容量）：节点总资源。
- **Allocatable**（可调度）：可用于调度的资源。

示例：

```yaml
kubeletArguments:
  eviction-hard:
    - "memory.available<500Mi"
  system-reserved:
    - "memory=1.5Gi"
```

可通过节点摘要 API 确定合适的值。

修改后重启 kubelet：

```
$ systemctl restart kubelet
```

防止节点状态振荡

为避免软驱逐阈值上下振荡，设置 `eviction-pressure-transition-period`：

示例：

```yaml
kubeletArguments:
  eviction-pressure-transition-period:
    - 5m
```

默认值为 5 分钟。修改后重启服务。

回收节点级资源

当满足驱逐条件时，节点会先回收资源，再驱逐用户 Pod。

- 启用 **imagefs**：
  - 达到 `nodefs` 阈值时：删除死亡的 Pod/容器。
  - 达到 `imagefs` 阈值时：删除未使用的镜像。
- 未启用 **imagefs**：

- 达到 `nodefs` 阈值时：先删除死亡的 Pod/容器，再删除未使用的镜像。

Pod 驱逐

当阈值和宽限期满足时，驱逐 Pod，直到信号低于阈值。

Pod 按服务质量（QoS）和资源消耗排序驱逐。

QoS 等级	描述
Guaranteed	优先驱逐资源消耗最高的 Pod。
Burstable	优先驱逐相对于请求资源消耗最高的 Pod。
BestEffort	优先驱逐资源消耗最高的 Pod。

只有当系统守护进程超过保留资源或只剩 Guaranteed Pod 时，才会驱逐 Guaranteed Pod。

磁盘为 BestEffort 资源，Pod 按 QoS 和磁盘使用量逐个驱逐以回收磁盘空间。

服务质量与内存杀手（OOM Killer）

如果在内存回收前发生系统 OOM 事件，OOM 杀手会介入。

OOM 分数根据 QoS 设置：

QoS 等级	oom_score_adj 值
Guaranteed	-998
Burstable	min(max(2, 1000 - (1000 * memoryRequestBytes) / machineMemoryCapacityBytes), 999)
BestEffort	1000

OOM 杀手终止分数最高的容器。优先终止 QoS 最低且内存使用最高的容器。容器可能根据节点策略被重启。

调度器与资源耗尽状态

调度器在调度 Pod 时考虑节点状态。

节点状态	调度器行为
MemoryPressure	不调度 BestEffort Pod。
DiskPressure	不调度任何新增 Pod。

示例场景

运维希望：

- 节点内存为 10Gi。
- 为系统守护进程保留 10%。
- 在利用率达到 95% 时驱逐 Pod。

计算：

- `capacity = 10Gi`
- `system-reserved = 1Gi`
- `allocatable = 9Gi`

若要在可用内存低于 10% 持续 30 秒时触发软驱逐，或低于 5% 时立即驱逐：

- `system-reserved = 2Gi`
- `allocatable = 8Gi`

配置示例：

```yaml
kubeletArguments:
  system-reserved:
    - "memory=2Gi"
  eviction-hard:
    - "memory.available<.5Gi"
  eviction-soft:
    - "memory.available<1Gi"
  eviction-soft-grace-period:
    - "memory.available=30s"
```

此配置防止调度后立即出现内存压力和驱逐。

推荐实践

DaemonSet 与资源耗尽处理

DaemonSet 创建的 Pod 被驱逐后会立即重建。DaemonSet 应避免使用 BestEffort Pod，采用 Guaranteed QoS 以降低被驱逐风险。

健康检查

目录

理解健康检查
探针类型
HTTP `GET` 操作
`exec` 操作
TCP `Socket` 操作
最佳实践
YAML 文件示例
通过 Web 控制台配置健康检查参数
常用参数
协议特定参数
探针失败排查
查看 **Pod** 事件
查看容器日志
手动测试探针端点
检查探针配置
检查应用代码
资源限制
网络问题

理解健康检查

请参考官方 Kubernetes 文档：

- Liveness, Readiness, and Startup Probes ↗
- Configure Liveness, Readiness and Startup Probes ↗

> 在 Kubernetes 中，健康检查，也称为探针，是确保应用高可用性和弹性的关键机制。Kubernetes 使用这些探针来判断 Pod 的健康状态和就绪状态，从而允许系统采取适当的操作，例如重启容器或路由流量。没有适当的健康检查，Kubernetes 无法可靠地管理应用的生命周期，可能导致服务性能下降或中断。

Kubernetes 提供三种类型的探针：

- `livenessProbe`：检测容器是否仍在运行。如果存活探针失败，Kubernetes 会根据重启策略终止并重启 Pod。
- `readinessProbe`：检测容器是否准备好提供服务。如果就绪探针失败，Endpoint Controller 会将 Pod 从 Service 的 Endpoint 列表中移除，直到探针成功。
- `startupProbe`：专门检测应用是否成功启动。存活和就绪探针在启动探针成功之前不会执行。对于启动时间较长的应用非常有用。

正确配置这些探针对于构建健壮且自愈的 Kubernetes 应用至关重要。

探针类型

Kubernetes 支持三种实现探针的机制：

HTTP `GET` 操作

> 对 Pod 的 IP 地址指定端口和路径执行 HTTP `GET` 请求。如果响应码在 200 到 399 之间，则探针成功。

- 适用场景：Web 服务器、REST API 或任何暴露 HTTP 端点的应用。
- 示例：

```yaml
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 20
```

`exec` 操作

> 在容器内执行指定命令。如果命令以状态码 0 退出，则探针成功。

- 适用场景：无 HTTP 端点的应用，检查内部应用状态，或执行需要特定工具的复杂健康检查。
- 示例：

```yaml
readinessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

TCP `Socket` 操作

> 尝试在容器的 IP 地址和指定端口打开 TCP 套接字。如果能建立 TCP 连接，则探针成功。

- 适用场景：数据库、消息队列或任何通过 TCP 端口通信但可能没有 HTTP 端点的应用。
- 示例：

```yaml
startupProbe:
  tcpSocket:
    port: 3306
  initialDelaySeconds: 5
  periodSeconds: 10
  failureThreshold: 30
```

最佳实践

- 存活探针 vs. 就绪探针：
  - 存活探针：如果应用无响应，最好重启它。失败时，Kubernetes 会重启容器。
  - 就绪探针：如果应用暂时无法提供服务（例如连接数据库中），但可能无需重启即可恢复，使用就绪探针。这可防止流量路由到不健康的实例。
- 慢启动应用使用启动探针：对于启动时间较长的应用，使用启动探针。这样可以避免因存活探针失败导致的过早重启，或因就绪探针失败导致的流量路由问题。
- 轻量级探针：确保探针端点轻量且响应迅速。探针不应涉及重计算或依赖外部服务（如数据库调用），以免探针本身不可靠。
- 有意义的检查：探针检查应真实反映应用的健康和就绪状态，而不仅仅是进程是否运行。例如，对于 Web 服务器，应检查是否能提供基本页面，而不仅是端口是否开放。
- 调整 **initialDelaySeconds**：合理设置 initialDelaySeconds，给予应用足够启动时间后再开始探测。
- 调整 **periodSeconds** 和 **failureThreshold**：在快速发现故障和避免误判之间取得平衡。探针过于频繁或 failureThreshold 过低可能导致不必要的重启或不就绪状态。
- 调试日志：确保应用日志清晰记录健康检查端点调用及内部状态，方便排查探针失败原因。
- 组合使用探针：通常同时使用三种探针（存活、就绪、启动）以有效管理应用生命周期。

YAML 文件示例

```yaml
spec:
  template:
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2 # 容器镜像
          ports:
            - containerPort: 80 # 容器暴露端口
          startupProbe:
            httpGet:
              path: /startup-check
              port: 8080
            initialDelaySeconds: 0 # 启动探针通常为 0 或非常小
            periodSeconds: 5
            failureThreshold: 60 # 允许 60 * 5 = 300 秒（5 分钟）启动时间
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 5 # Pod 启动后延迟 5 秒开始检查
            periodSeconds: 10 # 每 10 秒检查一次
            timeoutSeconds: 5 # 超时 5 秒
            failureThreshold: 3 # 连续失败 3 次视为不健康
          readinessProbe:
            httpGet:
              path: /ready
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 10
            timeoutSeconds: 5
            failureThreshold: 3
```

通过 Web 控制台配置健康检查参数

常用参数

参数	描述
Initial Delay	`initialDelaySeconds`：开始探测前的宽限时间（秒）。默认值：`300`。
Period	`periodSeconds`：探测间隔（1-120秒）。默认值：`60`。
Timeout	`timeoutSeconds`：探测超时时长（1-300秒）。默认值：`30`。
Success Threshold	`successThreshold`：标记为健康所需的最小连续成功次数。默认值：`0`。
Failure Threshold	`failureThreshold`：触发动作的最大连续失败次数： - `0`：禁用基于失败的动作 - 默认：连续 `5` 次失败 → 容器重启。

协议特定参数

参数	适用协议	描述
Protocol	HTTP/HTTPS	健康检查协议
Port	HTTP/HTTPS/TCP	探测目标容器端口
Path	HTTP/HTTPS	端点路径（例如 `/healthz`）
HTTP Headers	HTTP/HTTPS	自定义请求头（添加键值对）
Command	EXEC	容器内执行的检查命令（例如 `sh -c "curl -I localhost:8080 | grep OK"`）。 注意：转义特殊字符并测试命令有效性。

探针失败排查

当 Pod 状态显示与探针相关的问题时，可按以下步骤排查：

查看 Pod 事件

```
kubectl describe pod <pod-name>
```

查找与 LivenessProbe failed、ReadinessProbe failed 或 StartupProbe failed 相关的事件。这些事件通常包含具体错误信息（如连接拒绝、HTTP 500 错误、命令退出码等）。

查看容器日志

```
kubectl logs <pod-name> -c <container-name>
```

检查应用日志，查看探针失败时是否有错误或警告。应用可能记录了健康检查端点未正确响应的原因。

手动测试探针端点

- **HTTP**：如果可能，使用 `kubectl exec -it <pod-name> -- curl <probe-path>:<probe-port>` 或容器内的 `wget` 查看实际响应。
- **Exec**：手动执行探针命令：`kubectl exec -it <pod-name> -- <command-from-probe>`，检查退出码和输出。
- **TCP**：从同一网络内的另一个 Pod 或允许的主机使用 `nc`（netcat）或 `telnet` 测试 TCP 连接：`kubectl exec -it <another-pod> -- nc -vz <pod-ip> <probe-port>`。

检查探针配置

- 仔细核对 Deployment/Pod YAML 中的探针参数（路径、端口、命令、延迟、阈值）。常见错误包括端口或路径配置错误。

检查应用代码

- 确保应用的健康检查端点实现正确，真实反映应用的就绪和存活状态。有时端点可能返回成功，但应用本身已损坏。

资源限制

- CPU 或内存资源不足可能导致应用无响应，引发探针失败。检查 Pod 资源使用情况（`kubectl top pod <pod-name>`），并考虑调整 `resources` 的限制和请求。

网络问题

- 极少数情况下，网络策略或 CNI 问题可能阻止探针访问容器。验证集群内的网络连通性。

计算组件

Deployments

Understanding Deployments

Creating Deployments

Managing Deployments

使用 CLI 进行故障排查

DaemonSets

理解守护进程集

创建守护进程集

管理守护进程集

StatefulSets

理解 StatefulSe

创建 StatefulSe

管理 StatefulSe

Pods

理解 Pods

YAML 文件示例

使用 CLI 管理 Pod

使用 Web 控制台管理 Pod

示例

Containers

理解 Containers

理解 Ephemera

与 Containers 交

Deployments

目录

Understanding Deployments
Creating Deployments
使用 CLI 创建 Deployment
前提条件
YAML 文件示例
通过 YAML 创建 Deployment
使用 Web 控制台创建 Deployment
前提条件
操作步骤 - 配置基本信息
操作步骤 - 配置 Pod
操作步骤 - 配置容器
参考信息
健康检查
Managing Deployments
使用 CLI 管理 Deployment
查看 Deployment
更新 Deployment
扩缩 Deployment
回滚 Deployment
删除 Deployment
使用 Web 控制台管理 Deployment
查看 Deployment

更新 Deployment

删除 Deployment

使用 CLI 进行故障排查

检查 Deployment 状态

检查 ReplicaSet 状态

检查 Pod 状态

查看日志

进入 Pod 进行调试

检查健康检查配置

检查资源限制

Understanding Deployments

参考官方 Kubernetes 文档：Deployments ↗

> **Deployment** 是 Kubernetes 中一种高级工作负载资源，用于声明式地管理和更新应用的 Pod 副本。它提供了一种强大且灵活的方式来定义应用的运行方式，包括维护多少副本以及如何安全地执行滚动更新。

Deployment 是 Kubernetes API 中管理 Pods 和 ReplicaSets 的对象。当你创建一个 Deployment 时，Kubernetes 会自动创建一个 ReplicaSet，负责维护指定数量的 Pod 副本。

使用 **Deployments** 可以实现：

- 声明式管理：定义应用的期望状态，Kubernetes 自动确保集群的实际状态与期望状态一致。
- 版本控制与回滚：跟踪 Deployment 的每个修订版本，出现问题时可以轻松回滚到之前的稳定版本。
- 零停机更新：使用滚动更新策略逐步更新应用，无服务中断。
- 自我修复：当 Pod 崩溃、终止或从节点移除时，Deployment 会自动替换 Pod 实例，确保指定数量的 Pod 始终可用。

工作原理：

1. 通过 Deployment 定义应用的期望状态（例如，使用哪个镜像，运行多少副本）。
2. Deployment 创建 ReplicaSet，确保指定数量的 Pod 正在运行。
3. ReplicaSet 创建并管理实际的 Pod 实例。
4. 当更新 Deployment（例如更改镜像版本）时，Deployment 会创建新的 ReplicaSet，并根据预定义的滚动更新策略逐步替换旧 Pod，直到所有新 Pod 运行，然后删除旧的 ReplicaSet。

Creating Deployments

使用 **CLI** 创建 **Deployment**

前提条件

- 确保已配置并连接到集群的 `kubectl`。

YAML 文件示例

```yaml
# example-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment # Deployment 名称
  labels:
    app: nginx # 用于识别和选择的标签
spec:
  replicas: 3 # 期望的 Pod 副本数
  selector:
    matchLabels:
      app: nginx # 选择器，匹配该 Deployment 管理的 Pods
  template:
    metadata:
      labels:
        app: nginx # Pod 的标签，必须匹配 selector.matchLabels
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2 # 容器镜像
          ports:
            - containerPort: 80 # 容器暴露端口
          resources: # 资源限制和请求
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 200m
              memory: 256Mi
```

通过 YAML 创建 Deployment

```bash
# 第一步：通过 yaml 创建 Deployment
kubectl apply -f example-deployment.yaml

# 第二步：检查 Deployment 状态
kubectl get deployment nginx-deployment # 查看 Deployment
kubectl get pod -l app=nginx # 查看该 Deployment 创建的 Pods
```

使用 **Web** 控制台创建 **Deployment**

前提条件

获取镜像地址。镜像来源可以是平台管理员通过工具链集成的镜像仓库，也可以是第三方平台的镜像仓库。

- 对于前者，管理员通常会将镜像仓库分配给你的项目，你可以使用其中的镜像。如果找不到所需的镜像仓库，请联系管理员分配。
- 如果是第三方平台的镜像仓库，确保当前集群可以直接拉取该镜像。
- 如果镜像仓库需要认证，需配置相应的镜像拉取密钥。详情见 为 ServiceAccount 添加 ImagePullSecrets。

操作步骤 - 配置基本信息

1. 在 **Container Platform**，左侧导航栏进入 **Workloads** > **Deployments**。
2. 点击 **Create Deployment**。
3. 选择或输入镜像，点击 **Confirm**。

> **INFO**
>
> 注意：使用 Web 控制台集成的镜像仓库时，可以通过 **Already Integrated** 过滤镜像。集成项目名称示例：images (registry-projectname)，其中包含 Web 控制台中的项目名 projectname 和镜像仓库中的项目名 containers。

4. 在 **Basic Info** 区域，配置 Deployment 工作负载的声明式参数：

参数	说明
Replicas	定义 Deployment 中期望的 Pod 副本数（默认：`1`）。根据工作负载需求调整。
More > **Update Strategy**	配置 `rollingUpdate` 策略，实现零停机部署： **Max surge** (`maxSurge`)： • 更新时允许超过期望副本数的最大 Pod 数量。

参数	说明
	• 支持绝对值（如 `2`）或百分比（如 `20%`）。 • 百分比计算方式：`ceil(当前副本数 × 百分比)`。 • 示例：10 副本时，4.1 → `5`。 **Max unavailable** (`maxUnavailable`)： • 更新时允许不可用的最大 Pod 数量。 • 百分比值不可超过 `100%`。 • 百分比计算方式：`floor(当前副本数 × 百分比)`。 • 示例：10 副本时，4.9 → `4`。 注意事项： 1. 默认值：若未显式设置，`maxSurge=1`，`maxUnavailable=1`。 2. 非运行状态的 **Pod**（如 `Pending`、`CrashLoopBackOff`）视为不可用。 3. 同时约束： • `maxSurge` 和 `maxUnavailable` 不能同时为 `0` 或 `0%`。 • 若两者百分比均计算为 `0`，Kubernetes 会强制设置 `maxUnavailable=1` 以保证更新进度。 示例： 对于 10 副本的 Deployment： • `maxSurge=2` → 更新期间总 Pod 数为 `10 + 2 = 12`。 • `maxUnavailable=3` → 最小可用 Pod 数为 `10 - 3 = 7`。 • 确保在控制滚动更新的同时保证可用性。

操作步骤 - 配置 Pod

注意：在混合架构集群中部署单架构镜像时，需确保正确配置 Node Affinity 规则 以实现 Pod 调度。

1. 在 **Pod** 区域，配置容器运行时参数及生命周期管理：

参数	说明
Volumes	挂载持久卷到容器。支持的卷类型包括 `PVC`、`ConfigMap`、`Secret`、`emptyDir`、`hostPath` 等。具体实现见 卷挂载指南。
Pull Secret	仅在从第三方镜像仓库（通过手动输入镜像 URL）拉取镜像时必需。 注意：用于认证拉取受保护镜像的 Secret。
Close Grace Period	Pod 接收到终止信号后允许的优雅关闭时间（默认：`30s`）。 - 在此期间，Pod 会完成正在处理的请求并释放资源。 - 设置为 `0` 会强制立即删除（SIGKILL），可能导致请求中断。

2. **Node Affinity** 规则

参数	说明
More > Node Selector	限制 Pod 只能调度到带有特定标签的节点（例如 `kubernetes.io/os: linux`）。 Node Selector: acp.cpaas.io/node-group-share-mode:Share × Found 1 matched nodes in current cluster

参数	说明
More > Affinity	定义基于现有条件的细粒度调度规则。 **Affinity** 类型： • **Pod Affinity**：将新 Pod 调度到运行特定 Pod 的节点（相同拓扑域）。 • **Pod Anti-affinity**：避免新 Pod 与特定 Pod 共存。 执行模式： • `requiredDuringSchedulingIgnoredDuringExecution`：仅当规则满足时才调度 Pod。 • `preferredDuringSchedulingIgnoredDuringExecution`：优先满足规则的节点，但允许例外。 配置字段： • `topologyKey`：定义拓扑域的节点标签（默认：`kubernetes.io/hostname`）。 • `labelSelector`：通过标签查询过滤目标 Pod。

3. 网络配置

- Kube-OVN

参数	说明
Bandwidth Limits	对 Pod 网络流量实施 QoS： • 出站速率限制：最大出站流量速率（如 `10Mbps`）。 • 入站速率限制：最大入站流量速率。
Subnet	从预定义子网池分配 IP。若未指定，使用命名空间默认子网。
Static IP Address	绑定持久 IP 地址给 Pod： • 多个 Deployment 的 Pod 可声明相同 IP，但同一时间仅允许一个 Pod 使用。

参数	说明
	• 关键：静态 IP 数量必须 ≥ Pod 副本数。

- Calico

参数	说明
Static IP Address	分配固定 IP，严格唯一： • 每个 IP 只能绑定给集群中的一个 **Pod**。 • 关键：静态 IP 数量必须 ≥ Pod 副本数。

操作步骤 - 配置容器

1. 在 **Container** 区域，参考以下说明配置相关信息。

参数	说明
Resource Requests & Limits	• **Requests**：容器运行所需的最小 CPU/内存。 • **Limits**：容器运行时允许的最大 CPU/内存。单位定义见 资源单位。 命名空间超售比： • 无超售比： 若存在命名空间资源配额，容器请求/限制继承命名空间默认值（可修改）。 无命名空间配额时，无默认值，自定义请求。 • 有超售比： 请求自动计算为 `Limits / 超售比`（不可修改）。 约束条件： • 请求 ≤ 限制 ≤ 命名空间配额最大值。 • 超售比变更需重建 Pod 生效。

<table>
<tr><th>参数</th><th>说明</th></tr>
<tr><td></td><td>超售比启用时禁止手动配置请求。
无命名空间配额时无容器资源限制。</td></tr>
<tr><td>Extended Resources</td><td>配置集群可用的扩展资源（如 vGPU、pGPU）。</td></tr>
<tr><td>Volume Mounts</td><td>持久存储配置。详见 存储卷挂载说明。
操作：
已有 Pod 卷：点击 Add
无 Pod 卷：点击 Add & Mount
参数：
<code>mountPath</code>：容器文件系统路径（如 <code>/data</code>）
<code>subPath</code>：卷内相对文件/目录路径。
对于 <code>ConfigMap</code> / <code>Secret</code>：选择特定键
<code>readOnly</code>：只读挂载（默认读写）
参考 Kubernetes 卷 ↗。</td></tr>
<tr><td>Ports</td><td>暴露容器端口。
示例：暴露 TCP 端口 <code>6379</code>，名称为 <code>redis</code>。
字段：
<code>protocol</code>：TCP/UDP
<code>Port</code>：暴露端口（如 <code>6379</code>）
<code>name</code>：符合 DNS 规范的标识符（如 <code>redis</code>）</td></tr>
<tr><td>Startup Commands & Arguments</td><td>覆盖默认 ENTRYPOINT/CMD：
示例 1：执行 <code>top -b</code>
- Command：<code>["top", "-b"]</code>
- 或 Command：<code>["top"]</code>，Args：<code>["-b"]</code>
示例 2：输出 <code>$MESSAGE</code>：
<code>/bin/sh -c "while true; do echo $(MESSAGE); sleep 10;</code></td></tr>
</table>

参数	说明
	`done"` 详见 定义命令↗。
More > **Environment Variables**	• 静态值：直接键值对 • 动态值：引用 ConfigMap/Secret 键，Pod 字段（`fieldRef`），资源指标（`resourceFieldRef`） 注意：环境变量会覆盖镜像或配置文件中的设置。
More > **Referenced ConfigMaps**	将整个 ConfigMap/Secret 注入为环境变量。支持的 Secret 类型：`Opaque`、`kubernetes.io/basic-auth`。
More > **Health Checks**	• **Liveness Probe**：检测容器健康（失败时重启） • **Readiness Probe**：检测服务可用性（失败时从 Endpoints 移除） 详见 健康检查参数。
More > **Log Files**	配置日志路径： - 默认收集 `stdout` - 文件模式，如 `/var/log/*.log` 要求： • 存储驱动 `overlay2`：默认支持 • `devicemapper`：需手动挂载 EmptyDir 到日志目录 • Windows 节点：确保父目录已挂载（如 `c:/a` 对应 `c:/a/b/c/*.log`）
More > **Exclude Log Files**	排除特定日志收集（如 `/var/log/aaa.log`）。
More > **Execute before Stopping**	容器终止前执行命令。 示例：`echo "stop"`

参数	说明
	注意：命令执行时间须短于 Pod 的 `terminationGracePeriodSeconds`。

2. 点击右上角的 **Add Container** 或 **Add Init Container**。
   参考 Init Containers ↗。 Init Container：

   1. 在应用容器之前启动（顺序执行）。
   2. 完成后释放资源。
   3. 允许删除条件：
      - Pod 有多个应用容器且至少有一个 Init Container。
      - 单应用容器的 Pod 不允许删除 Init Container。

3. 点击 **Create**。

参考信息

存储卷挂载说明

类型	用途
Persistent Volume Claim	绑定已有的 PVC 以请求持久存储。 注意：仅可选择已绑定（关联 PV）的 PVC。未绑定的 PVC 会导致 Pod 创建失败。
ConfigMap	将完整或部分 ConfigMap 数据挂载为文件： • 完整 ConfigMap：在挂载路径下创建以键名命名的文件 • 子路径选择：挂载特定键（如 `my.cnf`）
Secret	将完整或部分 Secret 数据挂载为文件： • 完整 Secret：在挂载路径下创建以键名命名的文件 • 子路径选择：挂载特定键（如 `tls.crt`）

类型	用途
Ephemeral Volumes	集群动态提供的临时卷，具备： • 动态配置 • 生命周期与 Pod 绑定 • 支持声明式配置 使用场景：临时数据存储。详见 Ephemeral Volumes
Empty Directory	Pod 内容器间共享的临时存储： • Pod 启动时在节点创建 • Pod 删除时删除 使用场景：容器间文件共享、临时数据存储。详见 EmptyDir
Host Path	挂载宿主机目录（必须以 `/` 开头，如 `/volumepath`）。

健康检查

- 健康检查 YAML 文件示例
- Web 控制台健康检查配置参数

Managing Deployments

使用 CLI 管理 Deployment

查看 Deployment

- 检查 Deployment 是否已创建。

```
kubectl get deployments
```

- 获取 Deployment 详情。

```
kubectl describe deployments
```

更新 Deployment

按照以下步骤更新 Deployment：

1. 将 nginx Pods 更新为使用 nginx:1 .16.1 镜像。

   ```
   kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
   ```

   或使用以下命令：

   ```
   kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
   ```

   也可以编辑 Deployment，将 `.spec.template.spec.containers[0].image` 从 `nginx:1.14.2` 改为 `nginx:1.16.1`：

   ```
   kubectl edit deployment/nginx-deployment
   ```

2. 查看滚动更新状态：

   ```
   kubectl rollout status deployment/nginx-deployment
   ```

   运行 `kubectl get rs` 查看 Deployment 通过创建新 ReplicaSet 并扩容到 3 副本，同时缩容旧 ReplicaSet 到 0 副本来更新 Pods。

   ```
   kubectl get rs
   ```

   运行 `kubectl get pods` 应只显示新 Pods：

   ```
   kubectl get pods
   ```

扩缩 Deployment

使用以下命令扩缩 Deployment：

```
kubectl scale deployment/nginx-deployment --replicas=10
```

回滚 Deployment

- 假设更新 Deployment 时输入镜像名错误，将 `nginx:1.16.1` 写成了 `nginx:1.161`：

  ```
  kubectl set image deployment/nginx-deployment nginx=nginx:1.161
  ```

- 滚动更新会卡住。可通过检查滚动状态验证：

  ```
  kubectl rollout status deployment/nginx-deployment
  ```

删除 Deployment

删除 Deployment 会同时删除其管理的 ReplicaSet 及所有关联的 Pods。

```
kubectl delete deployment <deployment-name>
```

使用 Web 控制台管理 Deployment

查看 Deployment

可查看 Deployment 以获取应用信息。

1. 在 **Container Platform**，导航至 **Workloads** > **Deployments**。
2. 找到要查看的 Deployment。
3. 点击 Deployment 名称查看 详情、拓扑、日志、事件、监控等。

更新 Deployment

1. 在 **Container Platform**，导航至 **Workloads** > **Deployments**。
2. 找到要更新的 Deployment。

3. 在 **Actions** 下拉菜单中选择 **Update**，进入编辑 Deployment 页面。

删除 Deployment

1. 在 **Container Platform**，导航至 **Workloads** > **Deployments**。
2. 找到要删除的 Deployment。
3. 在 **Actions** 下拉菜单中点击操作列的 **Delete** 按钮并确认。

使用 CLI 进行故障排查

当 Deployment 遇到问题时，以下是常用的排查方法。

检查 Deployment 状态

```
kubectl get deployment nginx-deployment
kubectl describe deployment nginx-deployment # 查看详细事件和状态
```

检查 ReplicaSet 状态

```
kubectl get rs -l app=nginx
kubectl describe rs <replicaset-name>
```

检查 Pod 状态

```
kubectl get pods -l app=nginx
kubectl describe pod <pod-name>
```

查看日志

```
kubectl logs <pod-name> -c <container-name> # 查看指定容器日志
kubectl logs <pod-name> --previous         # 查看上一个终止容器的日志
```

进入 **Pod** 进行调试

```
kubectl exec -it <pod-name> -- /bin/bash # 进入容器 Shell
```

检查健康检查配置

确保 livenessProbe 和 readinessProbe 配置正确，且应用的健康检查接口响应正常。探针失败排查

检查资源限制

确保容器资源请求和限制合理，避免因资源不足导致容器被杀死。

DaemonSets

目录

理解守护进程集
创建守护进程集
使用 CLI 创建守护进程集
前提条件
YAML 文件示例
通过 YAML 创建守护进程集
使用 Web 控制台创建守护进程集
前提条件
操作步骤 - 配置基础信息
操作步骤 - 配置 Pod
操作步骤 - 配置容器
操作步骤 - 创建
管理守护进程集
使用 CLI 管理守护进程集
查看守护进程集
更新守护进程集
删除守护进程集
使用 Web 控制台管理守护进程集
查看守护进程集
更新守护进程集
删除守护进程集

理解守护进程集

参考官方 Kubernetes 文档：DaemonSets

DaemonSet 是 Kubernetes 的一种控制器，用于确保所有（或部分）集群节点上运行指定 Pod 的一个副本。与以应用为中心的 Deployment 不同，DaemonSet 以节点为中心，非常适合部署集群范围的基础设施服务，如日志收集器、监控代理或存储守护进程。

> **WARNING**
>
> **DaemonSet** 操作注意事项
>
> 1. 行为特征
>
>   - **Pod** 分布：DaemonSet 在每个符合条件的可调度节点上部署且仅部署一个 **Pod** 副本：
>   - 在每个符合以下条件的可调度节点上部署且仅部署一个 Pod 副本：
>   - 满足 `nodeSelector` 或 `nodeAffinity` 条件（如果指定）。
>   - 节点状态不是 `NotReady`。
>   - 节点没有 `NoSchedule` 或 `NoExecute` **Taints**，除非 Pod 模板中配置了对应的 **Tolerations**。
>   - **Pod** 数量公式：DaemonSet 管理的 **Pod** 数量 等于 符合条件的节点数量。
>   - 双重角色节点处理：同时担任 控制平面 和 工作节点 角色的节点，只会运行一个 DaemonSet 的 Pod 实例（无论其角色标签），前提是该节点可调度。
>
> 2. 关键限制（排除节点）
>
>   - 明确标记为 `Unschedulable: true` 的节点（例如通过 `kubectl cordon`）。
>   - 处于 `NotReady` 状态的节点。
>   - 具有不兼容的 **Taints** 且 DaemonSet 的 Pod 模板中未配置相应 Tolerations 的节点。

创建守护进程集

使用 **CLI** 创建守护进程集

前提条件

- 确保已配置好 `kubectl` 并连接到集群。

YAML 文件示例

```yaml
# example-daemonSet.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  labels:
    k8s-app: fluentd-logging
spec:
  selector: # 定义 DaemonSet 如何识别其管理的 Pods，必须匹配 `template.metadata.labels`
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template: # 定义 DaemonSet 的 Pod 模板，每个由该 DaemonSet 创建的 Pod 都符合此模板
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations: # 这些容忍用于允许守护进程集在控制平面节点上运行，如不希望控制平面节点运行 Pod 可移除
        - key: node-role.kubernetes.io/control-plane
          operator: Exists
          effect: NoSchedule
        - key: node-role.kubernetes.io/master
          operator: Exists
          effect: NoSchedule
      containers:
        - name: fluentd-elasticsearch
          image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:
            - name: varlog
              mountPath: /var/log
      # 可以设置较高的优先级类以确保 DaemonSet Pod
```

```yaml
      # 优先抢占正在运行的 Pod
      # priorityClassName: important
      terminationGracePeriodSeconds: 30
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

通过 **YAML** 创建守护进程集

```
# 第一步：执行以下命令创建 *example-daemonSet.yaml* 中定义的守护进程集
kubectl apply -f example-daemonSet.yaml

# 第二步：验证守护进程集及其管理的 Pod 状态：
kubectl get daemonset fluentd-elasticsearch # 查看守护进程集
kubectl get pods -l name=fluentd-elasticsearch -o wide # 查看该守护进程集管理的 Pod 及所在节点
```

使用 **Web** 控制台创建守护进程集

前提条件

获取镜像地址。镜像来源可以是平台管理员通过工具链集成的镜像仓库，也可以是第三方平台的镜像仓库。

- 对于前者，管理员通常会将镜像仓库分配给您的项目，您可以使用其中的镜像。如果找不到所需镜像仓库，请联系管理员进行分配。
- 对于第三方平台的镜像仓库，确保当前集群可以直接拉取镜像。
- 如果镜像仓库需要认证，需配置相应的镜像拉取密钥。详情请参见 为 ServiceAccount 添加 ImagePullSecrets。

操作步骤 **-** 配置基础信息

1. 在 **Container Platform** 中，左侧导航栏进入 **Workloads** > **DaemonSets**。
2. 点击 **Create DaemonSet**。
3. 选择或输入镜像，点击 **Confirm**。

INFO

注意：使用 Web 控制台集成的镜像仓库时，可通过 已集成 过滤镜像。集成项目名称 例如 images (registry-projectname)，其中包含该 Web 控制台中的项目名 projectname 以及镜像仓库中的项目名 containers。

在 基础信息 部分，配置守护进程集工作负载的声明式参数：

参数	说明
更多 > 更新策略	配置 DaemonSet Pod 零停机更新的 `rollingUpdate` 策略。 最大不可用数 (`maxUnavailable`)：更新期间允许暂时不可用的最大 Pod 数量。支持绝对值（如 1）或百分比（如 10%）。 示例：若有 10 个节点且 maxUnavailable 为 10%，则 floor(10 * 0.1) = 1 个 Pod 可不可用。 注意： • 默认值：若未显式设置，`maxSurge` 默认为 0，`maxUnavailable` 默认为 1（或百分比时为 10%）。 • 非运行状态 **Pod**：处于 `Pending` 或 `CrashLoopBackOff` 等状态的 Pod 被视为不可用。 • 同时限制：`maxSurge` 和 `maxUnavailable` 不能同时为 0 或 0%。若百分比计算结果均为 0，Kubernetes 会强制将 `maxUnavailable` 设为 1 以保证更新进度。

操作步骤 - 配置 Pod

Pod 部分，请参见 Deployment - 配置 Pod

操作步骤 - 配置容器

Containers 部分，请参见 Deployment - 配置容器

操作步骤 - 创建

点击 **Create**。

点击 **Create** 后，DaemonSet 将：

- ✅ 自动在所有符合条件的节点上部署 Pod 副本，条件包括：
  - 满足 `nodeSelector` 条件（如定义）。
  - 配置了 `tolerations`（允许调度到带有污点的节点）。
  - 节点处于 `Ready` 状态且 `Schedulable: true`。
- ❌ 排除节点：
  - 带有 `NoSchedule` 污点的节点（除非显式容忍）。
  - 手动标记为不可调度的节点（`kubectl cordon`）。
  - 处于 `NotReady` 或 `Unschedulable` 状态的节点。

管理守护进程集

使用 CLI 管理守护进程集

查看守护进程集

- 获取某命名空间下所有守护进程集的摘要：

```
kubectl get daemonsets -n <namespace>
```

- 获取指定守护进程集的详细信息，包括事件和 Pod 状态：

```
kubectl describe daemonset <daemonset-name>
```

更新守护进程集

当修改守护进程集的 **Pod** 模板（例如更改容器器镜像或添加卷挂载）时，Kubernetes 默认会执行滚动更新（前提是 `updateStrategy.type` 为 `RollingUpdate`，且这是默认值）。

- 首先编辑 YAML 文件（如 `example-daemonset.yaml`）并保存修改，然后应用：

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

- 可通过以下命令监控滚动更新进度：

  ```
  kubectl rollout status daemonset/<daemonset-name>
  ```

删除守护进程集

删除守护进程集及其管理的所有 Pod：

```
kubectl delete daemonset <daemonset-name>
```

使用 **Web** 控制台管理守护进程集

查看守护进程集

1. 在 **Container Platform** 中，进入 **Workloads** > **DaemonSets**。
2. 找到想查看的守护进程集。
3. 点击守护进程集名称，查看 详情、拓扑、日志、事件、监控 等信息。

更新守护进程集

1. 在 **Container Platform** 中，进入 **Workloads** > **DaemonSets**。
2. 找到想更新的守护进程集。
3. 在 操作 下拉菜单中选择 **Update**，进入编辑守护进程集页面，可更新 `Replicas`、`image`、`updateStrategy` 等参数。

删除守护进程集

1. 在 **Container Platform** 中，进入 **Workloads** > **DaemonSets**。
2. 找到想删除的守护进程集。
3. 在 操作 下拉菜单中点击 **Delete** 按钮并确认。

StatefulSets

目录

理解 StatefulSets
创建 StatefulSets
使用 CLI 创建 StatefulSet
前提条件
YAML 文件示例
通过 YAML 创建 StatefulSet
使用 Web 控制台创建 StatefulSet
前提条件
操作步骤 - 配置基本信息
操作步骤 - 配置 Pod
操作步骤 - 配置容器
操作步骤 - 创建
健康检查
管理 StatefulSets
使用 CLI 管理 StatefulSet
查看 StatefulSet
扩缩容 StatefulSet
更新 StatefulSet（滚动更新）
删除 StatefulSet
使用 Web 控制台管理 StatefulSet
查看 StatefulSet
更新 StatefulSet

删除 StatefulSet

理解 StatefulSets

请参考 Kubernetes 官方文档：StatefulSets

StatefulSet 是 Kubernetes 的一种工作负载 API 对象，旨在通过提供以下功能来管理有状态应用：

- 稳定的网络身份：DNS 主机名格式为 `<statefulset-name>-<ordinal>.<service-name>.ns.svc.cluster.local`。
- 稳定的持久存储：通过 `volumeClaimTemplates` 实现。
- 有序的部署/扩缩容：Pod 按顺序创建/删除：Pod-0 → Pod-1 → Pod-N。
- 有序的滚动更新：Pod 按逆序号更新：Pod-N → Pod-0。

在分布式系统中，可以部署多个 StatefulSets 作为独立组件来提供专门的有状态服务（例如 *Kafka brokers*、*MongoDB shards*）。

创建 StatefulSets

使用 CLI 创建 StatefulSet

前提条件

- 确保已配置好 `kubectl` 并连接到集群。

YAML 文件示例

```yaml
# example-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # 必须与 .spec.template.metadata.labels 匹配
  serviceName: 'nginx' # 该无头 Service 负责 Pod 的网络身份
  replicas: 3 # 定义期望的 Pod 副本数（默认：1）
  minReadySeconds: 10 # 默认值为 0
  template: # 定义 StatefulSet 的 Pod 模板
    metadata:
      labels:
        app: nginx # 必须与 .spec.selector.matchLabels 匹配
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: nginx
          image: registry.k8s.io/nginx-slim:0.24
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: www
              mountPath: /usr/share/nginx/html
  volumeClaimTemplates: # 定义 PersistentVolumeClaim (PVC) 模板。每个 Pod 会基于此动态创建唯一的 PersistentVolume (PV)。
    - metadata:
        name: www
      spec:
        accessModes: ['ReadWriteOnce']
        storageClassName: 'my-storage-class'
        resources:
          requests:
            storage: 1Gi
---
# example-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
```

```yaml
  labels:
    app: nginx
spec:
  ports:
    - port: 80
      name: web
  clusterIP: None
  selector:
    app: nginx
```

通过 YAML 创建 StatefulSet

```
# 第一步：执行以下命令创建 example-statefulset.yaml 中定义的 StatefulSet
kubectl apply -f example-statefulset.yaml

# 第二步：验证 StatefulSet 及其关联的 Pods 和 PVC 的创建状态：
kubectl get statefulset web # 查看 StatefulSet
kubectl get pods -l app=nginx # 查看该 StatefulSet 管理的 Pods
kubectl get pvc -l app=nginx # 查看 volumeClaimTemplates 创建的 PVC
```

使用 Web 控制台创建 StatefulSet

前提条件

获取镜像地址。镜像来源可以是平台管理员通过工具链集成的镜像仓库，也可以是第三方平台的镜像仓库。

- 对于前者，管理员通常会将镜像仓库分配给你的项目，你可以使用其中的镜像。如果找不到所需镜像仓库，请联系管理员分配。
- 对于第三方平台的镜像仓库，确保当前集群可以直接拉取镜像。
- 如果镜像仓库需要认证，则需配置相应的镜像拉取密钥。详情请参见 Add ImagePullSecrets to ServiceAccount。

操作步骤 - 配置基本信息

1. 在 **Container Platform** 左侧导航栏进入 **Workloads** > **StatefulSets**。
2. 点击 **Create StatefulSet**。
3. 选择或输入镜像，点击 **Confirm**。

> **INFO**
>
> 注意：使用 Web 控制台集成的镜像仓库时，可以通过 **Already Integrated** 过滤镜像。**Integration Project Name** 例如 images (registry-projectname)，其中包含该 Web 控制台中的项目名 projectname 以及镜像仓库中的项目名 containers。

在 **Basic Info** 部分，配置 StatefulSet 工作负载的声明式参数：

参数	说明
Replicas	定义 StatefulSet 中期望的 Pod 副本数（默认：1）。根据工作负载需求和预期请求量调整。
Update Strategy	控制 StatefulSet 滚动更新时的分阶段更新策略。默认且推荐使用 `RollingUpdate` 策略。 **Partition** 值：Pod 更新的序号阈值。 • 序号 ≥ `partition` 的 Pod 会立即更新。 • 序号 < `partition` 的 Pod 保持之前的规格。 示例： • `Replicas=5` （Pods：web-0 ~ web-4） • `Partition=3` （仅更新 web-3 和 web-4）

Volume Claim Templates	`volumeClaimTemplates` 是 StatefulSet 的关键特性，支持为每个 Pod 动态创建持久存储。StatefulSet 中的每个 Pod 副本都会基于预定义模板自动获得专属的 PersistentVolumeClaim (PVC)。
	• 1. 动态 **PVC** 创建：为每个 Pod 自动创建唯一 PVC，命名格式为 `<statefulset-name>-<claim-template-name>-<pod-ordinal>`。示例：`web-www-web-0`、`web-www-web-1`。
	• 2. 访问模式：支持所有 Kubernetes 访问模式。
	  • ReadWriteOnce (RWO - 单节点读写)
	  • ReadOnlyMany (ROX - 多节点只读)
	  • ReadWriteMany (RWX - 多节点读写)

参数	说明
	• 3. 存储类：通过 storageClassName 指定存储后端，未指定时使用集群默认 StorageClass。支持多种云/本地存储类型（如 SSD、HDD）。 • 4. 容量：通过 resources.requests.storage 配置存储容量。示例：1Gi。若 StorageClass 支持，支持动态扩容。

操作步骤 - 配置 Pod

Pod 部分，请参考 Deployment - Configure Pod

操作步骤 - 配置容器

Containers 部分，请参考 Deployment - Configure Containers

操作步骤 - 创建

点击 **Create**。

健康检查

- 健康检查 YAML 文件示例
- Web 控制台健康检查配置参数

管理 StatefulSets

使用 CLI 管理 StatefulSet

查看 StatefulSet

可以查看 StatefulSet 以获取应用信息。

- 查看 StatefulSet 是否已创建。

```
kubectl get statefulsets
```

- 获取 StatefulSet 详细信息。

```
kubectl describe statefulsets
```

扩缩容 StatefulSet

- 修改已有 StatefulSet 的副本数：

```
kubectl scale statefulset <statefulset-name> --replicas=<new-replica-count>
```

- 示例：

```
kubectl scale statefulset web --replicas=5
```

更新 StatefulSet（滚动更新）

当修改 StatefulSet 的 Pod 模板（例如更改容器镜像）时，Kubernetes 默认会执行滚动更新（前提是 updateStrategy 设置为 RollingUpdate，且这是默认值）。

- 首先编辑 YAML 文件（例如 example-statefulset.yaml）进行所需更改，然后应用：

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

- 然后，可以监控滚动更新进度：

```
kubectl rollout status statefulset/<statefulset-name>
```

删除 StatefulSet

删除 StatefulSet 及其关联的 Pods：

```
kubectl delete statefulset <statefulset-name>
```

默认情况下，删除 StatefulSet 不会删除其关联的 PersistentVolumeClaims (PVCs) 或 PersistentVolumes (PVs)，以防止数据丢失。若需同时删除 PVC，请显式执行：

```
kubectl delete pvc -l app=<label-selector-for-your-statefulset> # 示例：kubectl delete pvc -l app=nginx
```

另外，如果你的 `volumeClaimTemplates` 使用了 `reclaimPolicy` 为 `Delete` 的 StorageClass，则在删除 PVC 时，PV 及其底层存储也会自动删除。

使用 Web 控制台管理 StatefulSet

查看 StatefulSet

1. 在 **Container Platform**，进入 **Workloads** > **StatefulSets**。
2. 找到要查看的 StatefulSet。
3. 点击 StatefulSet 名称查看 **Details**、**Topology**、**Logs**、**Events**、**Monitoring** 等信息。

更新 StatefulSet

1. 在 **Container Platform**，进入 **Workloads** > **StatefulSets**。
2. 找到要更新的 StatefulSet。
3. 在 **Actions** 下拉菜单中选择 **Update**，进入编辑 StatefulSet 页面，可更新 `Replicas`、`image`、`updateStrategy` 等参数。

删除 StatefulSet

1. 在 **Container Platform**，进入 **Workloads** > **StatefulSets**。
2. 找到要删除的 StatefulSet。
3. 在 **Actions** 下拉菜单中点击操作列的 **Delete** 按钮并确认。

CronJobs

目录

- 理解 CronJobs
- 创建 CronJobs
  - 使用 CLI 创建 CronJob
    - 前提条件
    - YAML 文件示例
    - 通过 YAML 创建 CronJobs
  - 使用 Web 控制台创建 CronJobs
    - 前提条件
    - 操作步骤 - 配置基本信息
    - 操作步骤 - 配置 Pod
    - 操作步骤 - 配置容器
    - 创建
- 立即执行
  - 定位 CronJob 资源
  - 发起临时执行
  - 查看 Job 详情：
  - 监控执行状态
- 删除 CronJobs
  - 使用 Web 控制台删除 CronJobs
  - 使用 CLI 删除 CronJobs

理解 CronJobs

请参考官方 Kubernetes 文档：

- CronJobs ↗
- 使用 CronJob 运行自动化任务 ↗

CronJob 定义了运行至完成后停止的任务。它允许您根据计划多次运行相同的 Job。

CronJob 是 Kubernetes 中的一种工作负载控制器。您可以通过 Web 控制台或 CLI 创建 CronJob，定期或重复运行非持久化程序，例如定时备份、定时清理或定时邮件发送。

创建 CronJobs

使用 CLI 创建 CronJob

前提条件

- 确保已配置并连接到集群的 `kubectl`。

YAML 文件示例

```yaml
# example-cronjob.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

通过 YAML 创建 CronJobs

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

使用 Web 控制台创建 CronJobs

前提条件

获取镜像地址。镜像可以来自平台管理员通过工具链集成的镜像仓库，或来自第三方镜像仓库。

- 对于集成仓库中的镜像，管理员通常会将镜像仓库分配给您的项目，允许您使用其中的镜像。如果找不到所需的镜像仓库，请联系管理员进行分配。
- 如果使用第三方镜像仓库，请确保当前集群内可以直接拉取该镜像。
- 如果镜像仓库需要认证，您需要配置相应的镜像拉取密钥。详情请参见 向 ServiceAccount 添加 ImagePullSecrets。

操作步骤 - 配置基本信息

1. 在 **Container Platform** 中，左侧导航栏进入 **Workloads** > **CronJobs**。
2. 点击 创建 **CronJob**。
3. 选择或输入镜像，点击 确认。
   注意：只有使用平台集成的镜像仓库中的镜像时，才支持镜像过滤。例如，集成项目名为 containers（registry-projectname）表示平台项目名为 projectname，镜像仓库项目名为 containers。
4. 在 **Cron** 配置 部分，配置任务执行方式及相关参数。
   执行类型：

- 手动：手动执行需要每次任务运行时显式触发。
- 定时：定时执行需要配置以下调度参数：

参数	说明
Schedule	使用 Crontab 语法 定义定时计划。CronJob 控制器根据所选时区计算下一次执行时间。 注意： • Kubernetes 集群版本 < v1.25：不支持时区选择，计划必须使用 UTC。 • Kubernetes 集群版本 ≥ v1.25：支持时区感知调度（默认使用用户本地时区）。
并发策略	指定并发 Job 执行的处理方式（`Allow`、`Forbid` 或 `Replace`，详见 K8s 规范）。

Job 历史保留：

- 设置已完成 Job 的保留限制：
  - 历史限制：成功 Job 的历史保留数量（默认：20）
  - 失败 **Job**：失败 Job 的历史保留数量（默认：20）
- 超出保留限制时，最旧的 Job 会被优先回收。

5. 在 **Job** 配置 部分，选择 Job 类型。 CronJob 管理由 Pod 组成的 Job。根据您的工作负载类型配置 Job 模板：

参数	说明
Job 类型	选择 Job 完成模式（`非并行`、`固定完成次数的并行` 或 `索引 Job`，详见 K8s Job 模式）。
重试限制	设置 Job 标记为失败前的最大重试次数。

操作步骤 - 配置 Pod

- **Pod** 部分，请参考 Deployment - 配置 Pod

操作步骤 - 配置容器

- 容器 部分，请参考 Deployment - 配置容器

创建

- 点击 创建。

立即执行

定位 CronJob 资源

- **Web** 控制台：在 **Container Platform** 中，左侧导航栏进入 **Workloads** > **CronJobs**。
- **CLI**：

```
kubectl get cronjobs -n <namespace>
```

发起临时执行

- **Web** 控制台：点击 立即执行

  1. 在 CronJob 列表右侧点击竖排省略号 (⋮)。
  2. 点击 立即执行。（或者在 CronJob 详情页右上角点击操作，选择 立即执行）。

- **CLI**：

  ```
  kubectl create job --from=cronjob/<cronjob-name> <job-name> -n <namespace>
  ```

查看 **Job** 详情：

```
kubectl describe job/<job-name> -n <namespace>
kubectl logs job/<job-name> -n <namespace>
```

监控执行状态

状态	说明
Pending	Job 已创建但尚未调度执行。
Running	Job 的 Pod 正在执行中。
Succeeded	与 Job 关联的所有 Pod 均成功完成（退出码为 0）。
Failed	至少有一个与 Job 关联的 Pod 非正常终止（退出码非 0）。

删除 **CronJobs**

使用 **Web** 控制台删除 **CronJobs**

1. 在 **Container Platform** 中，左侧导航栏进入 **Workloads** > **CronJobs**。
2. 找到需要删除的 CronJobs。

3. 在 操作 下拉菜单中，点击 删除 按钮并确认。

使用 **CLI** 删除 **CronJobs**

```
kubectl delete cronjob <cronjob-name>
```

Jobs

目录

理解 Jobs

YAML 文件示例

执行概览

理解 Jobs

参考官方 Kubernetes 文档：Jobs

Job 提供了多种定义任务的方式，这些任务运行至完成后停止。您可以使用 Job 来定义只运行一次并完成的任务。

- 原子执行单元：每个 Job 管理一个或多个 Pod，直到成功完成。
- 重试机制：由 `spec.backoffLimit` 控制（默认值：6）。
- 完成跟踪：使用 `spec.completions` 定义所需的成功次数。

YAML 文件示例

```yaml
# example-job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: data-processing-job
spec:
  completions: 1 # 需要的成功完成次数
  parallelism: 1 # 最大并行 Pod 数量
  backoffLimit: 3 # 最大重试次数
  template:
    spec:
      restartPolicy: Never # Job 特定策略（Never/OnFailure）
      containers:
        - name: processor
          image: alpine:3.14
          command: ['/bin/sh', '-c']
          args:
            - echo "Processing data..."; sleep 30; echo "Job completed"
```

执行概览

Kubernetes 中每次 Job 执行都会创建一个专用的 Job 对象，使用户能够：

- 通过以下命令创建 **Job**

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

- 通过以下命令跟踪 **Job** 生命周期

  ```
  kubectl get jobs
  ```

- 通过以下命令查看执行详情

  ```
  kubectl describe job/<job-name>
  ```

- 通过以下命令查看 **Pod** 日志

```
kubectl logs <pod-name>
```

Pods

目录

- 理解 Pods
- YAML 文件示例
- 使用 CLI 管理 Pod
  - 查看 Pod
  - 查看 Pod 日志
  - 在 Pod 中执行命令
  - Pod 端口转发
  - 删除 Pod
- 使用 Web 控制台管理 Pod
  - 查看 Pod
    - 操作步骤
    - Pod 参数说明
  - 删除 Pod
  - 使用场景
  - 操作步骤

理解 Pods

请参考 Kubernetes 官方网站文档：Pod ↗

Pod 是 Kubernetes 中可以创建和管理的最小可部署计算单元。一个 **Pod**（类似于鲸鱼群或豆荚）是一组一个或多个容器（如容器），共享存储和网络资源，并包含如何运行这些容器的规范。**Pods** 是所有更高级控制器（如 **Deployments**、**StatefulSets**、**DaemonSets**）构建的基础模块。

YAML 文件示例

pod-example.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:latest # The container image to use.
      ports:
        - containerPort: 80 # Container ports exposed.
      resources: # Defines CPU and memory requests and limits for the container.
        requests:
          cpu: '100m'
          memory: '128Mi'
        limits:
          cpu: '200m'
          memory: '256Mi'
```

使用 CLI 管理 Pod

虽然 Pods 通常由更高级的控制器管理，但直接使用 kubectl 操作 Pods 对于故障排查、检查和临时任务非常有用。

查看 Pod

- 列出当前命名空间中的所有 Pods：

```
kubectl get pods
```

- 列出所有命名空间中的所有 Pods：

```
kubectl get pods --all-namespaces
# 或简写版本：
kubectl get pods -A
```

- 获取指定 Pod 的详细信息：

```
kubectl describe pod <pod-name> -n <namespace>

# 示例
kubectl describe pod my-nginx-pod -n default
```

查看 **Pod** 日志

- 流式查看 Pod 中容器的日志（有助于调试）：

```
kubectl logs <pod-name> -n <namespace>
```

- 如果 Pod 中有多个容器，必须指定容器名称：

```
kubectl logs <pod-name> -c <container-name> -n <namespace>
```

- 跟随日志输出（实时查看新日志）：

```
kubectl logs -f <pod-name> -n <namespace>
```

在 **Pod** 中执行命令

在 Pod 中的指定容器内执行命令（有助于调试，如访问 shell）：

```
kubectl exec -it <pod-name> -n <namespace> -- <command>

# 示例（进入 shell）：
kubectl exec -it my-nginx-pod -n default -- /bin/bash
```

Pod 端口转发

将本地端口转发到 Pod 的端口，允许从本地机器直接访问 Pod 内运行的服务（适用于测试或无需外部暴露服务的直接访问）：

```
kubectl port-forward <pod-name> <local-port>:<pod-port> -n <namespace>

# 示例
kubectl port-forward my-nginx-pod 8080:80 -n default
```

运行此命令后，可以通过浏览器访问 localhost:8080 来访问运行在 my-nginx-pod 中的 Nginx Web 服务器。

删除 Pod

- 删除指定 Pod：

  ```
  kubectl delete pod <pod-name> -n <namespace>

  # 示例
  kubectl delete pod my-nginx-pod -n default
  ```

- 按名称删除多个 Pods：

  ```
  kubectl delete pod <pod-name-1> <pod-name-2> -n <namespace>
  ```

- 根据标签选择器删除 Pods（例如删除所有标签为 app=nginx 的 Pods）：

  ```
  kubectl delete pods -l app=nginx -n <namespace>
  ```

使用 Web 控制台管理 Pod

查看 Pod

平台界面提供了多种关于 Pods 的信息，方便快速查看。

操作步骤

1. 进入 **Container Platform**，在左侧边栏导航中选择 **Workloads** > **Pods**。
2. 找到想要查看的 Pod。
3. 点击部署名称，查看 **Details**、**YAML**、**Configuration**、**Logs**、**Events**、**Monitoring** 等信息。

Pod 参数说明

以下是部分参数的解释：

<table>
<tr><th>参数</th><th>说明</th></tr>
<tr><td>资源请求与限制</td><td>资源请求 和 限制 定义了 Pod 中容器的 CPU 和内存消耗边界，进而汇总形成 Pod 的整体资源配置。这些值对于 Kubernetes 调度器高效地将 Pods 安排到节点上，以及 kubelet 执行资源管理至关重要。

• 请求（Requests）：容器被调度和运行所需的最小保证 CPU/内存资源。Kubernetes 调度器根据此值决定 Pod 可运行的 节点。
• 限制（Limits）：容器运行时允许消耗的最大 CPU/内存资源。超过 CPU 限制会导致节流，超过内存限制则会导致容器被终止（内存不足 - OOM Killed）。

详细单位定义（如 <code>m</code> 表示 milliCPU，<code>Mi</code> 表示 mebibytes）请参考 Resource Units。

Pod 级资源计算逻辑
Pod 的有效 CPU 和内存请求与限制值由其各个容器规格的求和与最大值计算得出。Pod 级请求与限制的计算方法类似，本文以限制值为例说明。 当 Pod 仅包含标准容器（业务容器）时：Pod 的有效 CPU/内存限制值为 Pod 内所有容器 CPU/内存限制值的总和。

示例：若 Pod 包含两个容器，CPU/内存限制分别为 100m/100Mi 和 50m/200Mi，则 Pod 的聚合 CPU/内存限制为 150m/300Mi。 当 Pod 同时包含 initContainers 和标准容器时，Pod 的 CPU/内存限制值计算步骤如下：</td></tr>
</table>

<table>
<tr><th>参数</th><th>说明</th></tr>
<tr><td></td><td>1. 取所有 initContainers 中 CPU/内存限制的最大值。
2. 计算所有标准容器 CPU/内存限制的总和。
3. 比较步骤 1 和步骤 2 的结果，Pod 的综合 CPU/内存限制为 CPU 值的最大值（initContainers 最大值与容器总和）和内存值的最大值（initContainers 最大值与容器总和）。
计算示例：若 Pod 包含两个 initContainers，CPU/内存限制分别为 100m/200Mi 和 200m/100Mi，则 initContainers 的最大有效 CPU/内存限制为 200m/200Mi。同时，若 Pod 还包含两个标准容器，CPU/内存限制分别为 100m/100Mi 和 50m/200Mi，则标准容器的总限制为 150m/300Mi。因此，Pod 的综合 CPU/内存限制为 CPU max(200m, 150m) 和内存 max(200Mi, 300Mi)，即 200m/300Mi。</td></tr>
<tr><td>来源</td><td>管理该 Pod 生命周期的 Kubernetes 工作负载控制器，包括 Deployments、StatefulSets、DaemonSets、Jobs。</td></tr>
<tr><td>重启次数</td><td>Pod 启动以来，容器重启的次数。重启次数过多通常表明应用或其环境存在问题。</td></tr>
<tr><td>节点</td><td>Pod 当前调度并运行的 Kubernetes 节点名称。</td></tr>
<tr><td>服务账户</td><td>服务账户是 Kubernetes 对象，为 Pod 内运行的进程和服务提供身份认证，使其能够访问 Kubernetes APIServer。该字段通常仅在当前登录用户具有平台管理员角色或平台审计员角色时可见，以便查看服务账户的 YAML 定义。</td></tr>
</table>

删除 Pod

删除 Pod 可能影响计算组件的运行，请谨慎操作。

使用场景

- 及时恢复 Pod 到期望状态：如果 Pod 处于影响业务的状态，如 `Pending` 或 `CrashLoopBackOff`，在排查并解决错误信息后，手动删除 Pod 可以帮助其快速恢复到期望状态（如 `Running`）。此时，删除的 Pod 会在当前节点重建或重新调度。
- 资源清理与运维管理：部分 Pod 达到指定阶段后不再变化，这些 Pod 往往大量积累，影响其他 Pod 的管理。待清理的 Pod 可能包括因节点资源不足而处于 `Evicted` 状态的 Pod，

或因周期性定时任务触发而处于 `Completed` 状态的 Pod。此时，删除的 Pod 将不再存在。
注意：对于定时任务，如果需要查看每次任务执行的日志，不建议删除对应的 `Completed` 状态 Pod。

操作步骤

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Workloads** > **Pods**。
3. （单个删除）点击待删除 Pod 右侧的 ⋮ 按钮 > **Delete**，确认操作。
4. （批量删除）勾选待删除的 Pods，点击列表上方的 **Delete**，确认操作。

Containers

目录

理解 Containers

理解 Ephemeral Containers

实现原理：利用 Ephemeral Containers

使用 CLI 调试 Ephemeral Containers

使用 Web 控制台调试 Ephemeral Containers

与 Containers 交互

使用 CLI 与 Containers 交互

Exec

文件传输

使用 Web 控制台与 Containers 交互

通过 Applications 进入容器

通过 Pod 进入容器

理解 Containers

请参考 Kubernetes 官方网站文档：Containers↗。

Container 是一个轻量级、可执行的软件包，包含运行应用程序所需的一切：代码、运行时、系统工具、系统库和设置。虽然 Pod 是最小的可部署单元，但 containers 是 Pod 内的核心组件。

理解 Ephemeral Containers

调试 Containers
请参考 Kubernetes 官方网站文档：Ephemeral Containers ↗

Kubernetes 的 Ephemeral Containers 功能通过向现有 Pod 注入专用的调试工具（系统、网络和磁盘工具），为调试运行中的容器提供了强大的方式。

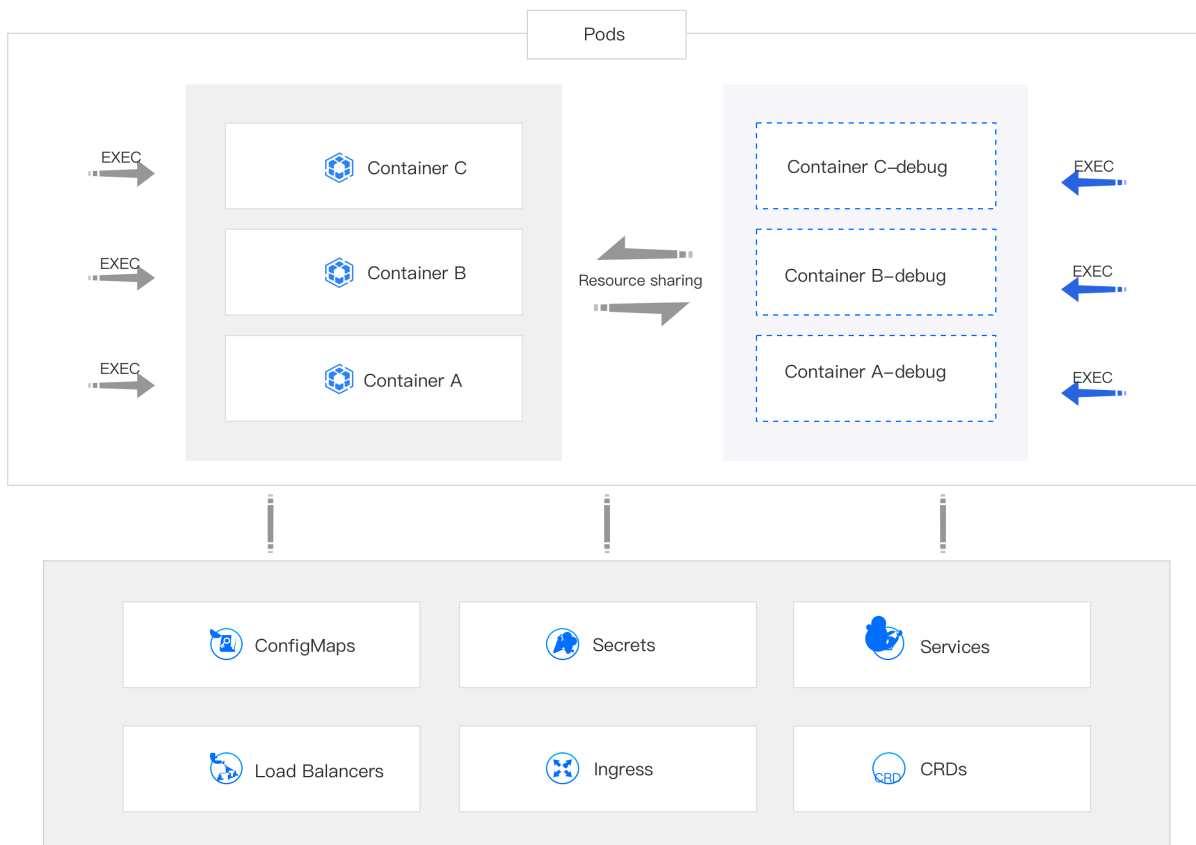
虽然通常可以使用 kubectl exec 直接在运行中的容器内执行命令，但许多生产环境的容器镜像故意保持精简，可能缺少关键的调试工具（如 bash、net-tools、tcpdump），以减小镜像大小和攻击面。Ephemeral Containers 通过提供预配置的丰富调试工具环境，解决了这一限制，非常适合以下场景：

- 故障诊断：当主应用容器出现问题（如意外崩溃、性能下降、网络连接异常）时，除了查看标准的 Pod 事件和日志外，通常需要在 Pod 的运行时环境中进行更深入的交互式排查。
- 配置调优与试验：如果当前应用配置表现不佳，可能希望临时调整组件设置或测试新配置，直接在运行中的容器内观察即时效果并制定改进方案。

实现原理：利用 Ephemeral Containers

调试功能是通过 **Ephemeral Containers** 实现的。Ephemeral Container 是一种专门用于内省和调试的特殊容器。它与现有的主容器共享 Pod 的网络命名空间和进程命名空间（如果启用），可以直接与应用进程交互和观察。

你可以动态地向运行中的 Pod 添加一个 Ephemeral Container（例如 `my-app-debug`），并使用其预装的调试工具。该 Ephemeral Container 的诊断结果与同一 Pod 内主应用 containers 的行为和状态直接相关。



:::Notes * 不能通过直接修改 Pod 的静态清单（PodSpec）来添加 Ephemeral Container。Ephemeral Containers 功能设计用于动态注入运行中的 Pod，通常通过 API 调用（如 kubectl debug）实现。 * 通过 debug 功能创建的 **Ephemeral Containers** 不具备资源（CPU/内存）或调度保障（即不会阻塞 Pod 启动，也没有独立的 QoS 类别），且退出后不会自动重启。因此，避免在其中运行持久业务应用，仅限调试用途。 * 如果 Pod 所在的 Node 资源利用率高或接近资源耗尽，使用 debug 功能时需谨慎。即使 Ephemeral Container 资源占用极小，也可能在资源紧张时加剧 Pod 被驱逐的风险。 :::

使用 **CLI** 调试 **Ephemeral Containers**

Kubernetes 1.25+ 提供了 `kubectl debug` 命令来创建 ephemeral containers。此方法为调试提供了强大的命令行替代方案。

命令

```
kubectl debug -it <pod-name> --image=<debug-image> --target=<target-conta
iner-name> -n <namespace>
# --image: 指定包含必要工具的调试镜像（如 busybox、ubuntu、nicolaka/netshoot）。
# --target: （可选）指定 Pod 中目标容器名称。若省略且只有一个容器，则默认该容器；若多
个，则默认第一个。
# -n: 指定命名空间。
```

Pod YAML 文件示例

示例：调试 `my-nginx-pod` 中的 `nginx`

- 首先，确保有一个 Pod 正在运行：

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

- 现在，在 `my-nginx-pod` 中创建一个名为 `debugger` 的 ephemeral debug 容器，目标容器为 `nginx`，使用 `busybox` 镜像：

  ```
  kubectl debug -it my-nginx-pod --image=busybox --target=nginx -- /bin/sh
  ```

  该命令会将你连接到 debugger ephemeral container 的 shell，你现在可以使用 busybox 工具调试 `my-nginx-container`。

- 查看附加到 Pod 的 ephemeral containers：

  ```
  kubectl describe pod my-nginx-pod
  ```

  在输出中查找 `Ephemeral Containers` 部分。

使用 **Web** 控制台调试 **Ephemeral Containers**

1. 进入 **Container Platform**，在左侧边栏导航至 **Workloads** > **Pods**。
2. 找到要查看的 Pod，点击 ⋮ > **Debug**。
3. 选择 Pod 中要调试的具体容器。
4. （可选）如果界面提示需要 初始化（例如设置必要的调试环境），点击 **Initialize**。

> **INFO**
>
> 初始化 Debug 功能后，只要 Pod 未被重新创建，即可直接进入 Ephemeral Container（例如 *Container A-debug*）进行调试。

5. 等待调试终端窗口准备就绪，然后开始调试操作。
   提示：点击终端右上角的“命令查询”选项，可查看常用调试工具及其使用示例。

   > **INFO**
   >
   > 点击右上角的命令查询，查看常用工具及用法。

6. 调试完成后，关闭终端窗口。

与 Containers 交互

你可以使用 `kubectl exec` 命令直接与运行中容器的内部实例交互，执行任意命令行操作。此外，Kubernetes 还提供了方便的文件上传和下载功能。

使用 CLI 与 Containers 交互

Exec

在 Pod 中的指定容器内执行命令（用于获取 shell、运行诊断命令等）：

```
kubectl exec -it <pod-name> -c <container-name> -n <namespace> -- <command>
# -it: 确保交互模式和伪终端（TTY），适合 shell 会话。
# -c: 指定 Pod 中的目标容器名称。若 Pod 仅有一个容器，可省略。
# --: 分隔 kubectl 参数和容器内执行的命令。
```

- 示例：进入 `my-nginx-pod` 中 `nginx` 容器的 Bash shell

  ```
  kubectl exec -it my-nginx-pod -c nginx -n default -- /bin/bash
  ```

- 示例：列出容器 `/tmp` 目录下的文件

  ```
  kubectl exec my-nginx-pod -c nginx -n default -- ls /tmp
  ```

文件传输

- 从本地复制文件到 Pod 中的容器：

```
kubectl cp <local-file-path> <namespace>/<pod-name>:<container-file-path> -c <container-name>
# -c: （可选）指定目标容器名称，适用于多容器 Pod。

# 示例：上传 `my-config.txt` 到 Nginx 的 HTML 目录
kubectl cp my-config.txt default/my-nginx-pod:/usr/share/nginx/html/my-config.txt -c nginx
```

- 从 Pod 中的容器复制文件到本地：

```
kubectl cp <namespace>/<pod-name>:<container-file-path> <local-file-path> -c <container-name>

# 示例：下载 Nginx 访问日志
kubectl cp default/my-nginx-pod:/var/log/nginx/access.log ./nginx_access.log -c nginx
```

使用 Web 控制台与 Containers 交互

通过 Applications 进入容器

你可以使用 `kubectl exec` 命令进入容器内部实例，在 Web 控制台窗口执行命令行操作。同时，可以通过文件传输功能方便地上传和下载容器内的文件。

1. 进入 **Container Platform**，导航至左侧边栏的 **Application** > **Applications**。
2. 点击 ***Application Name***。
3. 找到关联的工作负载（如 Deployment、StatefulSet），点击 **EXEC**，然后选择要进入的具体 *Pod Name*。接着选择 **EXEC** > ***Container Name***。
4. 输入要执行的命令。
5. 点击 **OK**，进入 Web 控制台窗口，执行命令行操作。
6. 点击 **File Transfer**。
   - 输入 **Upload Path**，将本地文件上传到容器（例如测试用配置文件）。

- 输入 **Download Path**，将日志、诊断数据或其他文件从容器下载到本地进行分析。

通过 **Pod** 进入容器

1. 进入 **Container Platform**，导航至 **Workloads** > **Pods**。
2. 找到目标 Pod，点击其旁边的垂直省略号（⋮），选择 EXEC，然后选择该 Pod 中要进入的具体 Container Name。
3. 输入要执行的命令。
4. 点击 **OK**，进入 Web 控制台窗口，执行命令行操作。
5. 点击 **File Transfer**。

   - 输入 **Upload Path**，将本地文件上传到容器（例如测试用配置文件）。
   - 输入 **Download Path**，将日志、诊断数据或其他文件从容器下载到本地进行分析。

使用 Helm charts

目录

1. 了解 Helm

1.1. 主要功能

1.2. 目录

术语定义

1.3 了解 HelmRequest

HelmRequest 与 Helm 的区别

HelmRequest 与 Application 集成

部署工作流程

组件定义

2 通过 CLI 以 Application 方式部署 Helm Charts

2.1 工作流程概览

2.2 准备 Chart

2.3 打包 Chart

2.4 获取 API 令牌

2.5 创建 Chart 仓库

2.6 上传 Chart

2.7 上传相关镜像

2.8 部署应用

2.9 更新应用

2.10 卸载应用

2.11 删除 Chart 仓库

3 通过 UI 以 Application 方式部署 Helm Charts

3.1 工作流程概览

3.2 前提条件

3.3 将模板添加到可管理仓库

3.4 删除模板的特定版本

操作步骤

---

1. 了解 Helm

Helm 是一个包管理器，用于简化在 Alauda Container Platform 集群上部署应用和服务的过程。
Helm 使用一种称为 *charts* 的打包格式。Helm chart 是一组描述 Kubernetes 资源的文件集合。
在集群中创建一个 chart 会生成一个运行中的 chart 实例，称为 *release*。
每次创建 chart，或升级、回滚 release 时，都会创建一个增量修订版本。

1.1. 主要功能

Helm 提供以下能力：

- 在 chart 仓库中搜索大量 charts
- 修改已有的 charts
- 使用 Kubernetes 资源创建自定义 charts
- 打包应用并以 charts 形式共享

1.2. 目录

目录基于 Helm 构建，提供了一个全面的 Chart 分发管理平台，突破了 Helm CLI 工具的限制。
该平台通过用户友好的界面，使开发者更方便地管理、部署和使用 charts。

术语定义

术语	定义	备注
Application Catalog	Helm Charts 的一站式管理平台	
Helm Charts	一种应用打包格式	
HelmRequest	CRD。定义部署 Helm Chart 所需的配置	模板应用
ChartRepo	CRD。对应 Helm charts 仓库	模板仓库
Chart	CRD。对应 Helm Charts	模板

1.3 了解 HelmRequest

在 Alauda Container Platform 中，Helm 部署主要通过一个名为 **HelmRequest** 的自定义资源进行管理。该方式扩展了标准 Helm 的功能，并将其无缝集成到 Kubernetes 原生资源模型中。

HelmRequest 与 Helm 的区别

标准 Helm 使用 CLI 命令管理 release，而 Alauda Container Platform 使用 HelmRequest 资源来定义、部署和管理 Helm charts。主要区别包括：

1. 声明式 **vs** 命令式：HelmRequest 提供声明式的 Helm 部署方式，而传统 Helm CLI 是命令式的。
2. **Kubernetes** 原生：HelmRequest 是直接集成 Kubernetes API 的自定义资源。
3. 持续调和：Captain 持续监控并调和 HelmRequest 资源与其期望状态。
4. 多集群支持：HelmRequest 支持通过平台跨多个集群部署。
5. 平台功能集成：HelmRequest 可与其他平台功能（如 Application 资源）集成。

HelmRequest 与 Application 集成

HelmRequest 与 Application 资源在概念上相似，用户可能希望统一查看它们。平台提供机制将 HelmRequest 同步为 Application 资源。

用户可通过添加以下注解将 HelmRequest 标记为以 Application 方式部署：

```
alauda.io/create-app: "true"
```

启用此功能后，平台 UI 会显示额外字段，并提供跳转到对应 Application 页面的链接。

部署工作流程

通过 HelmRequest 部署 charts 的流程包括：

1. 用户 创建或更新 HelmRequest 资源
2. **HelmRequest** 包含 chart 引用及应用的 values
3. **Captain** 处理 HelmRequest 并创建 Helm Release
4. **Release** 包含已部署的资源
5. **Metis** 监控带有应用注解的 HelmRequest 并同步到 Application
6. **Application** 提供已部署资源的统一视图

组件定义

- **HelmRequest**：描述期望 Helm chart 部署的自定义资源定义
- **Captain**：处理 HelmRequest 资源并管理 Helm release 的控制器（源码地址：https://github.com/alauda/captain↗）
- **Release**：Helm chart 的已部署实例
- **Charon**：监控 HelmRequest 并创建对应 Application 资源的组件
- **Application**：已部署资源的统一表示，提供额外管理能力
- **Archon-api**：平台内负责特定高级 API 功能的组件

2 通过 CLI 以 Application 方式部署 Helm Charts

2.1 工作流程概览

准备 chart → 打包 chart → 获取 API 令牌 → 创建 chart 仓库 → 上传 chart → 上传相关镜像 → 部署应用 → 更新应用 → 卸载应用 → 删除 chart 仓库

2.2 准备 Chart

Helm 使用一种称为 charts 的打包格式。chart 是一组描述 Kubernetes 资源的文件集合。单个 chart 可用于部署从简单 Pod 到复杂应用栈的任何内容。

参考官方文档：Helm Charts Documentation ↗

示例 chart 目录结构：

```
nginx/
├── Chart.lock
├── Chart.yaml
├── README.md
├── charts/
│   └── common/
│       ├── Chart.yaml
│       ├── README.md
│       ├── templates/
│       │   ├── _affinities.tpl
│       │   ├── _capabilities.tpl
│       │   ├── _errors.tpl
│       │   ├── _images.tpl
│       │   ├── _ingress.tpl
│       │   ├── _labels.tpl
│       │   ├── _names.tpl
│       │   ├── _secrets.tpl
│       │   ├── _storage.tpl
│       │   ├── _tplvalues.tpl
│       │   ├── _utils.tpl
│       │   ├── _warnings.tpl
│       │   └── validations/
│       │       ├── _cassandra.tpl
│       │       ├── _mariadb.tpl
│       │       ├── _mongodb.tpl
│       │       ├── _postgresql.tpl
│       │       ├── _redis.tpl
│       │       └── _validations.tpl
│       └── values.yaml
├── ci/
│   ├── ct-values.yaml
│   └── values-with-ingress-metrics-and-serverblock.yaml
├── templates/
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── extra-list.yaml
│   ├── health-ingress.yaml
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── ldap-daemon-secrets.yaml
│   ├── pdb.yaml
│   ├── server-block-configmap.yaml
```

```
│   ├── serviceaccount.yaml
│   ├── servicemonitor.yaml
│   ├── svc.yaml
│   └── tls-secrets.yaml
├── values.descriptor.yaml
├── values.schema.json
└── values.yaml
```

关键文件说明：

- `values.descriptor.yaml`（可选）：配合 ACP UI 显示用户友好表单
- `values.schema.json`（可选）：校验 values.yaml 内容并渲染简单 UI
- `values.yaml`（必需）：定义 chart 部署参数

2.3 打包 Chart

使用 `helm package` 命令打包 chart：

```
helm package nginx
# 输出: Successfully packaged chart and saved it to: /charts/nginx-8.8.0.tgz
```

2.4 获取 API 令牌

1. 在 **Alauda Container Platform** 中，点击右上角头像 => **Profile**
2. 点击 **Add Api Token**
3. 输入合适的描述和剩余有效期
4. 保存显示的令牌信息（仅显示一次）

2.5 创建 Chart 仓库

通过 API 创建本地 chart 仓库：

```
curl -k --request POST \
--url https://$ACP_DOMAIN/catalog/v1/chartrepos \
--header 'Authorization:Bearer $API_TOKEN' \
--header 'Content-Type: application/json' \
--data '{
  "apiVersion": "v1",
  "kind": "ChartRepoCreate",
  "metadata": {
    "name": "test",
    "namespace": "cpaas-system"
  },
  "spec": {
    "chartRepo": {
      "apiVersion": "app.alauda.io/v1beta1",
      "kind": "ChartRepo",
      "metadata": {
        "name": "test",
        "namespace": "cpaas-system",
        "labels": {
          "project.cpaas.io/catalog": "true"
        }
      },
      "spec": {
        "type": "Local",
        "url": null,
        "source": null
      }
    }
  }
}'
```

2.6 上传 Chart

将打包好的 chart 上传到仓库：

```
curl -k --request POST \
--url https://$ACP_DOMAIN/catalog/v1/chartrepos/cpaas-system/test/charts
\
--header 'Authorization:Bearer $API_TOKEN' \
--data-binary @"/root/charts/nginx-8.8.0.tgz"
```

2.7 上传相关镜像

1. 拉取镜像：`podman pull nginx`
2. 保存为 tar 包：`podman save nginx > nginx.latest.tar`
3. 加载并推送到私有仓库：

```
podman load -i nginx.latest.tar
podman tag nginx:latest 192.168.80.8:30050/nginx:latest
podman push 192.168.80.8:30050/nginx:latest
```

2.8 部署应用

通过 API 创建 Application 资源：

```
curl -k --request POST \
--url https://$ACP_DOMAIN/acp/v1/kubernetes/$CLUSTER_NAME/namespaces/$NAM
ESPACE/applications \
--header 'Authorization:Bearer $API_TOKEN' \
--header 'Content-Type: application/json' \
--data '{
  "apiVersion": "app.k8s.io/v1beta1",
  "kind": "Application",
  "metadata": {
    "name": "test",
    "namespace": "catalog-ns",
    "annotations": {
      "app.cpaas.io/chart.source": "test/nginx",
      "app.cpaas.io/chart.version": "8.8.0",
      "app.cpaas.io/chart.values": "{\"image\":{\"pullPolicy\":\"IfNotPre
sent\"}}"
    },
    "labels": {
      "sync-from-helmrequest": "true"
    }
  }
}'
```

2.9 更新应用

使用 PATCH 请求更新应用：

```
curl -k --request PATCH \
--url https://$ACP_DOMAIN/acp/v1/kubernetes/$CLUSTER_NAME/namespaces/$NAM
ESPACE/applications/test \
--header 'Authorization:Bearer $API_TOKEN' \
--header 'Content-Type: application/merge-patch+json' \
--data '{
  "apiVersion": "app.k8s.io/v1beta1",
  "kind": "Application",
  "metadata": {
    "annotations": {
      "app.cpaas.io/chart.values": "{\"image\":{\"pullPolicy\":\"Always
\"}}"
    }
  }
}'
```

2.10 卸载应用

删除 Application 资源：

```
curl -k --request DELETE \
--url https://$ACP_DOMAIN/acp/v1/kubernetes/$CLUSTER_NAME/namespaces/$NAM
ESPACE/applications/test \
--header 'Authorization:Bearer $API_TOKEN'
```

2.11 删除 Chart 仓库

```
curl -k --request DELETE \
--url https://$ACP_DOMAIN/apis/app.alauda.io/v1beta1/namespaces/cpaas-sys
tem/chartrepos/test \
--header 'Authorization:Bearer $API_TOKEN'
```

3 通过 UI 以 Application 方式部署 Helm Charts

3.1 工作流程概览

将模板添加到可管理仓库 → 上传模板 → 管理模板版本

3.2 前提条件

模板仓库由平台管理员添加。请联系平台管理员获取具有 管理 权限的 Chart 或 OCI Chart 类型模板仓库名称。

3.3 将模板添加到可管理仓库

1. 进入 **Catalog**。
2. 在左侧导航栏点击 **Helm Charts**。
3. 点击页面右上角的 **Add Template**，根据以下参数选择模板仓库。

参数	说明
模板仓库	直接同步模板到具有 管理 权限的 Chart 或 OCI Chart 类型模板仓库。分配给该 模板仓库 的项目负责人可直接使用该模板。
模板目录	当选择的模板仓库类型为 OCI Chart 时，必须选择或手动输入存放 Helm Chart 的目录。 注意：手动输入新模板目录时，平台会在模板仓库中创建该目录，但存在创建失败风险。

4. 点击 **Upload Template**，上传本地模板到仓库。
5. 点击 **Confirm**。模板上传过程可能需要几分钟，请耐心等待。
   注意：当模板状态由 `Uploading` 变为 `Upload Successful` 时，表示模板上传成功。
6. 若上传失败，请根据提示排查。
   注意：非法文件格式表示上传的压缩包中存在文件问题，如内容缺失或格式错误。

3.4 删除模板的特定版本

如果某个模板版本不再适用，可以删除该版本。

操作步骤

1. 进入 **Catalog**。
2. 在左侧导航栏点击 **Helm Charts**。
3. 点击 Chart 卡片查看详情。
4. 点击 **Manage Versions**。
5. 找到不再适用的模板版本，点击 **Delete** 并确认。

   删除版本后，相关应用将无法更新。

Alauda Container Platform

配置

Configuring ConfigMap

- Understanding Config Maps
- Config Map 限制
- 示例 ConfigMap
- 通过 Web 控制台创建 ConfigMap
- 通过 CLI 创建 ConfigMap
- 操作
- 通过 CLI 查看、编辑和删除
- Pod 中使用 ConfigMap 的方式
- ConfigMap 与 Secret 对比

Configuring Secrets

- 理解 Secrets
- 创建 Opaque 类型的 Secret
- 创建容器镜像仓库类型的 Secret
- 创建 Basic Auth 类型的 Secret
- 创建 SSH-Auth 类型的 Secret
- 创建 TLS 类型的 Secret
- 通过 Web 控制台创建 Secret
- 在 Pod 中如何使用 Secret
- 后续操作
- 操作

Configuring ConfigMap

Config maps 允许您将配置工件与镜像内容解耦，以保持容器化应用的可移植性。
以下章节定义了 config maps 以及如何创建和使用它们。

目录

Understanding Config Maps
Config Map 限制
示例 ConfigMap
通过 Web 控制台创建 ConfigMap
通过 CLI 创建 ConfigMap
操作
通过 CLI 查看、编辑和删除
Pod 中使用 ConfigMap 的方式
作为环境变量
作为卷中的文件
作为单个环境变量
ConfigMap 与 Secret 对比

Understanding Config Maps

许多应用程序需要通过配置文件、命令行参数和环境变量的某种组合进行配置。在 OpenShift Container Platform 中，这些配置工件与镜像内容解耦，以保持容器化应用的可移植性。

`ConfigMap` 对象提供了将配置数据注入容器的机制，同时使容器对 OpenShift Container Platform 保持无感知。config map 可用于存储细粒度的信息，如单个属性，也可存储粗粒度的信息，如整个配置文件或 JSON 数据块。

`ConfigMap` 对象保存键值对形式的配置数据，这些数据可以被 pod 消费，或用于存储系统组件（如控制器）的配置数据。例如：

```yaml
# my-app-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-app-config
data:
  app_mode: "development"
  feature_flags: "true"
  database.properties: |-
    jdbc.url=jdbc:mysql://localhost:3306/mydb
    jdbc.username=user
    jdbc.password=password
  log_settings.json: |-
    {
      "level": "INFO",
      "format": "json"
    }
```

注意：当您从二进制文件（如图片）创建 config map 时，可以使用 `binaryData` 字段。

配置数据可以通过多种方式在 pod 中被消费。config map 可用于：

- 填充容器中的环境变量值
- 设置容器的命令行参数
- 在卷中填充配置文件

用户和系统组件都可以将配置数据存储在 config map 中。
config map 类似于 secret，但设计上更方便处理不包含敏感信息的字符串。

Config Map 限制

- 必须先创建 config map，才能在 pod 中消费其内容。
- 控制器可以编写为容忍缺失的配置数据。请根据具体使用 config map 配置的组件逐一确认。
- `ConfigMap` 对象存在于项目中。
- 只能被同一项目中的 pod 引用。
- Kubectl 仅支持对从 API 服务器获取的 pod 使用 config map，包括通过 CLI 创建的 pod 或间接由复制控制器创建的 pod。不包括通过 OpenShift Container Platform 节点的 `--manifest-url` 标志、`--config` 标志或其 REST API 创建的 pod，因为这些不是常见的 pod 创建方式。

> **NOTE**
>
> Pod 只能使用同一命名空间内的 ConfigMaps。

示例 ConfigMap

您现在可以在 `Pod` 中使用 app-config。

```yaml
# app-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_ENV: "production"
  LOG_LEVEL: "debug"
```

通过 Web 控制台创建 ConfigMap

1. 进入 **Container Platform**。
2. 在左侧边栏点击 **Configuration** > **ConfigMap**。
3. 点击 **Create ConfigMap**。

4. 参考以下说明配置相关参数。

参数	说明
Entries	指 `key:value` 键值对，支持 **Add** 和 **Import** 两种方式。 • **Add**：可以逐条添加配置项，也可以在 Key 输入框中粘贴一行或多行 key=value 形式的内容批量添加配置项。 • **Import**：导入不超过 1M 的文本文件，文件名作为 key，文件内容作为 value，填充为一个配置项。
Binary Entries	指不超过 1M 的二进制文件，文件名作为 key，文件内容作为 value，填充为一个配置项。 注意：创建 ConfigMap 后，导入的文件不可修改。

批量添加格式示例：

```
# 每行一个 key=value，多个键值对必须分行，否则粘贴后无法正确识别。
key1=value1
key2=value2
key3=value3
```

5. 点击 **Create**。

通过 CLI 创建 ConfigMap

```
kubectl create configmap app-config \
  --from-literal=APP_ENV=production \
  --from-literal=LOG_LEVEL=debug
```

或者从文件创建：

```
kubectl apply -f app-config.yaml
```

操作

您可以点击列表页右侧的 (⋮) 或详情页右上角的 **Actions**，根据需要更新或删除 ConfigMap。

ConfigMap 的变更会影响引用该配置的工作负载，请提前阅读操作说明。

操作	说明
更新	• 添加或更新 ConfigMap 后，任何通过环境变量引用该 ConfigMap（或其配置项）的工作负载需要重建 Pod，才能使新配置生效。 • 对于导入的二进制配置项，仅支持键的更新，不支持值的更新。
删除	删除 ConfigMap 后，任何通过环境变量引用该 ConfigMap（或其配置项）的工作负载在重建 Pod 时可能会因找不到引用源而受到影响。

通过 **CLI** 查看、编辑和删除

```
kubectl get configmap app-config -o yaml
kubectl edit configmap app-config
kubectl delete configmap app-config
```

Pod 中使用 **ConfigMap** 的方式

作为环境变量

```
envFrom:
  - configMapRef:
      name: app-config
```

每个键都会成为容器中的一个环境变量。

作为卷中的文件

```
volumes:
  - name: config-volume
    configMap:
      name: app-config

volumeMounts:
  - name: config-volume
    mountPath: /etc/config
```

每个键对应 `/etc/config` 下的一个文件，文件内容为对应的值。

作为单个环境变量

```
env:
  - name: APP_ENV
    valueFrom:
      configMapKeyRef:
        name: app-config
        key: APP_ENV
```

ConfigMap 与 Secret 对比

特性	ConfigMap	Secret
数据类型	非敏感	敏感（如密码）
编码方式	明文	Base64 编码
使用场景	配置、标志	密码、令牌

Configuring Secrets

目录

理解 Secrets

使用特点

支持的类型

使用方式

创建 Opaque 类型的 Secret

创建容器镜像仓库类型的 Secret

创建 Basic Auth 类型的 Secret

创建 SSH-Auth 类型的 Secret

创建 TLS 类型的 Secret

通过 Web 控制台创建 Secret

在 Pod 中如何使用 Secret

作为环境变量

作为挂载文件（卷）

后续操作

操作

理解 Secrets

在 Kubernetes (k8s) 中，Secret 是一种用于存储和管理敏感信息的基础对象，例如密码、OAuth 令牌、SSH 密钥、TLS 证书和 API 密钥。其主要目的是防止敏感数据直接嵌入 Pod 定

义或容器镜像中，从而增强安全性和可移植性。

Secrets 类似于 ConfigMaps，但专门用于机密数据。它们通常以 base64 编码的形式存储，并可以通过多种方式被 Pod 使用，包括挂载为卷或作为环境变量暴露。

使用特点

- 增强安全性：与明文配置映射（Kubernetes ConfigMap）相比，Secrets 通过 Base64 编码存储敏感信息，结合 Kubernetes 的访问控制能力，显著降低数据泄露风险。
- 灵活性与管理：使用 Secrets 提供了比将敏感信息硬编码在 Pod 定义文件或容器镜像中更安全且灵活的方式。这种分离简化了敏感数据的管理和修改，无需更改应用代码或容器镜像。

支持的类型

Kubernetes 支持多种类型的 Secrets，每种类型针对特定的使用场景。平台通常支持以下类型：

- **Opaque**：通用 Secret 类型，用于存储任意键值对的敏感数据，如密码或 API 密钥。
- **TLS**：专门用于存储 TLS（传输层安全）协议的证书和私钥信息，常用于 HTTPS 通信和安全入口。
- **SSH Key**：用于存储 SSH 私钥，通常用于安全访问 Git 仓库或其他支持 SSH 的服务。
- **SSH Authentication (kubernetes.io/ssh-auth)**：存储通过 SSH 协议传输的数据的认证信息。
- **Username/Password (kubernetes.io/basic-auth)**：用于存储基本认证凭据（用户名和密码）。
- **Image Pull Secret**：存储从私有镜像仓库拉取容器镜像所需的 JSON 认证字符串。

使用方式

Secrets 可以通过不同方式被 Pod 内的应用使用：

- 作为环境变量：可以将 Secret 中的敏感数据直接注入容器的环境变量。
- 作为挂载文件（卷）：Secrets 可以挂载为 Pod 卷中的文件，允许应用从指定文件路径读取敏感数据。

注意：工作负载中的 Pod 实例只能引用同一命名空间内的 Secrets。有关高级用法和 YAML 配置，请参阅 Kubernetes 官方文档↗。

创建 Opaque 类型的 Secret

```
kubectl create secret generic my-secret \
  --from-literal=username=admin \
  --from-literal=password=Pa$$w0rd
```

YAML

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4= # base64 编码的 "admin"
  password: UGEkJHcwcmQ= # base64 编码的 "Pa$$w0rd"
```

你可以这样解码：

```
echo YWRtaW4= | base64 --decode  # 输出: admin
```

创建容器镜像仓库类型的 Secret

```
kubectl create secret docker-registry my-container-registry-creds \
  --docker-username=myuser \
  --docker-password=mypass \
  --docker-server=https://registry.example.com \
  --docker-email=my@example.com
```

YAML

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-container-registry-creds
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: eyJhdXRocyI6eyJodHRwczovL2luZGV4LmRvY2tlci5pby92MS8i
OnsidXNlcm5hbWUiOiJteXVzZXIiLCJwYXNzd29yZCI6Im15cGFzcyIsImVtYWlsIjoibXlAZ
XhhbXBsZS5jb20iLCJhdXRoIjoiYlhsMWMyVnlPbTE1Y0dGemN3PT0ifX19
```

K8s 会自动将你的用户名、密码、邮箱和服务器信息转换为标准登录格式：

```json
{
  "auths": {
    "https://registry.example.com": {
      "username": "myuser",
      "password": "mypass",
      "email": "my@example.com",
      "auth": "bXl1c2VyOm15cGFzcw=="  # base64 编码的 username:password
    }
  }
}
```

该 JSON 会被 base64 编码后用作 Secret 的 data 字段值。

在 Pod 中使用：

```yaml
imagePullSecrets:
  - name: my-container-registry-creds
```

创建 Basic Auth 类型的 Secret

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: basic-auth-secret
type: kubernetes.io/basic-auth
stringData:
  username: myuser
  password: mypass
```

创建 SSH-Auth 类型的 Secret

使用场景：存储 SSH 私钥（例如用于 Git 访问）。

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: ssh-key-secret
type: kubernetes.io/ssh-auth
stringData:
  ssh-privatekey: |
    -----BEGIN OPENSSH PRIVATE KEY-----
    ...
    -----END OPENSSH PRIVATE KEY-----
```

创建 TLS 类型的 Secret

使用场景：TLS 证书（用于 Ingress、webhooks 等）

```
kubectl create secret tls tls-secret \
--cert=path/to/tls.crt \
--key=path/to/tls.key
```

YAML

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: tls-secret
type: kubernetes.io/tls
data:
  tls.crt: <base64>
  tls.key: <base64>
```

通过 **Web** 控制台创建 **Secret**

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Configuration** > **Secrets**。
3. 点击 **Create Secret**。
4. 配置参数。
   注意：在表单视图中，输入的用户名和密码等敏感数据会自动以 Base64 格式编码后存储到 Secret 中。转换后的数据可以在 YAML 视图中预览。
5. 点击 **Create**。

在 **Pod** 中如何使用 **Secret**

作为环境变量

```yaml
env:
  - name: DB_USERNAME
    valueFrom:
      secretKeyRef:
        name: my-secret
        key: username
```

从名为 `my-secret` 的 Secret 中获取键为 `username` 的值，并赋给环境变量 `DB_USERNAME`。

作为挂载文件（卷）

```yaml
volumes:
  - name: secret-volume
    secret:
      secretName: my-secret

volumeMounts:
  - name: secret-volume
    mountPath: "/etc/secret"
```

后续操作

在同一命名空间中创建原生应用的工作负载时，可以引用已创建的 Secrets。

操作

你可以点击列表页右侧的 (⋮) 或详情页右上角的 **Actions**，根据需要更新或删除 Secret。

操作	描述
更新	添加或更新 Secret 后，引用该 Secret（或其配置项）的工作负载通过环境变量使用时，需要重建 Pod 以使新配置生效。
删除	• 删除 Secret 后，引用该 Secret（或其配置项）的工作负载在重建 Pod 时可能因找不到引用源而受到影响。 • 请勿删除平台自动生成的 Secrets，否则可能导致平台无法正常运行。例如：包含命名空间资源认证信息的 service-account-token 类型的 Secrets 以及系统命名空间（如 kube-system）中的 Secrets。

应用可观测

监控面板

前提条件

命名空间级别监控面板

工作负载级别监控

Logs

Procedure

Events

操作步骤

事件记录解读

监控面板

- 支持查看平台上工作负载组件过去 7 天的资源监控数据（监控数据保留周期可配置）。包括应用、工作负载、Pod 的统计信息，以及 CPU/内存使用的趋势和排名。
- 支持命名空间级别监控。
- 支持的工作负载级别监控：**Applications**、**Deployments**、**DaemonSets**、**StatefulSets** 和 **Pods**

目录

前提条件
命名空间级别监控面板
操作步骤
创建命名空间级别监控面板
工作负载级别监控
默认监控面板
操作步骤
指标说明
自定义监控面板

前提条件

- 安装监控插件

命名空间级别监控面板

操作步骤

1. 在 **Container Platform**，点击 **Observe** > **Dashboards**。
2. 查看命名空间下的监控数据。提供三个面板：**Applications Overview**、**Workloads Overview** 和 **Pods Overview**。
3. 切换面板以监控目标 **Overview**。

创建命名空间级别监控面板

1. 作为 管理员，参考创建监控面板创建专用监控面板。
2. 配置以下标签以在 **Container Platform** 显示命名空间级别监控面板：
   - `cpaas.io/dashboard.folder: container-platform`
   - `cpaas.io/dashboard.tag.overview: "true"`

工作负载级别监控

> 本操作步骤演示如何通过 Deployment 界面查看 Pod 监控。

默认监控面板

操作步骤

1. 在 **Container Platform**，点击 **Workloads** > **Deployments**。
2. 从列表中点击一个 Deployment 名称。
3. 进入 **Monitoring** 标签页查看默认监控指标。

指标说明

监控资源	指标粒度	技术定义
CPU	利用率/使用量	利用率 = 使用量/限制（limits） 评估容器限制配置，利用率高表示限制不足。 使用量 表示实际资源消耗。
内存	利用率/使用量	利用率 = 使用量/限制（limits） 评估方法同 CPU。利用率高可能导致组件不稳定。
网络流量	入流速率/出流速率	Pod 的网络流量（字节/秒），表示流入/流出流量。
网络包	接收速率/发送速率	Pod 接收/发送的网络包数（个数/秒）。
磁盘速率	读/写	每个工作负载挂载卷的读/写吞吐量（字节/秒）。
磁盘 **IOPS**	读/写	每个工作负载挂载卷的每秒输入/输出操作次数（IOPS）。

自定义监控面板

4. 点击 切换图标 切换到自定义面板。参考自定义面板添加图表创建专用的 工作负载级别 监控面板。

INFO

鼠标悬停在图表曲线上可查看特定时间点的单 Pod 指标和 PromQL 表达式。若 Pod 数超过 15 个，仅显示按降序排序的前 15 条记录。

Logs

聚合容器运行时日志并提供可视化查询功能。当应用、工作负载或其他资源出现异常行为时，日志分析有助于诊断根本原因。

目录

Procedure

Procedure

> 本操作步骤演示如何通过 Deployment 界面查看容器运行时日志。

1. 在 **Container Platform** 中，点击 **Workloads** > **Deployments**。
2. 从列表中点击一个 Deployment 名称。
3. 切换到 **Logs** 选项卡查看详细记录。

操作	描述
Pod/Container	使用下拉选择器在 Pods 和 Containers 之间切换，查看对应的日志。
Previous Logs	查看已终止容器的日志（当容器 restartCount > 0 时可用）。
Lines	配置显示日志缓冲区大小：1k/10k/100k 行。

操作	描述
Wrap Line	切换长日志条目的换行显示（默认启用）。
Find	支持全文搜索，匹配高亮并可通过回车键导航。
Raw	直接捕获自容器运行时接口（CRI）的未处理日志流，无格式化、过滤或截断。
Export	下载原始日志。
Full Screen	点击被截断的行，在模态对话框中查看完整内容。

WARNING

- 截断处理：日志条目超过 2000 字符时会以省略号 `...` 截断
  - 被截断部分无法被页面的查找功能匹配。
  - 点击被截断行中的省略号 `...` 标记，可在模态对话框中查看完整内容。
- 复制可靠性：当看到截断标记（...）或 ANSI 颜色码时，避免直接从渲染的日志查看器复制。请始终使用 **Export**、**Raw** 功能获取完整日志。
- 保留策略：实时日志遵循 Kubernetes 日志轮转配置。历史分析请使用 Logs ↗ 。

Events

Kubernetes 资源状态变化和操作状态更新产生的事件信息，集成可视化查询界面。当应用、工作负载或其他资源遇到异常时，实时事件分析有助于排查根因。

目录

操作步骤
事件记录解读

操作步骤

> 本操作步骤演示如何通过 Deployment 界面查看容器运行时事件。

1. 在 **Container Platform** 中，点击 **Workloads** > **Deployments**。
2. 从列表中点击一个 Deployment 名称。
3. 切换到 **Events** 标签页查看详细记录。

事件记录解读

资源事件记录：在事件摘要图表下方，列出指定时间范围内所有匹配事件。点击事件卡片查看完整事件详情。每个卡片显示：

- 资源类型：Kubernetes 资源类型，用图标缩写表示：

  - `P` = Pod
  - `RS` = ReplicaSet
  - `D` = Deployment
  - `SVC` = Service
- 资源名称：目标资源名称。
- 事件原因：Kubernetes 报告的原因（例如 FailedScheduling）。
- 事件级别：事件严重性。
  - `Normal` ：信息类
  - `Warning` ：需立即关注
- 时间：最后发生时间，发生次数。

INFO

Kubernetes 允许管理员通过 Event TTL 控制器配置事件保留周期，默认保留周期为 1 小时。过期事件由系统自动清理。欲查看完整历史记录，请访问 All Events。

实用指南

设置定时任务触发规则

时间转换

编写 Crontab 表达式

添加 ImagePullSecrets 到 Servic

创建 ImagePullSecret

将 ImagePullSecret 添加到 ServiceAccour

验证新建 Pod 是否设置了 imagePullSecre

Service Ser

Overview

Features

Configuration a

User Guide (Fo

设置定时任务触发规则

定时任务触发规则支持输入 Crontab 表达式。

目录

时间转换
编写 Crontab 表达式

时间转换

时间转换规则：本地时间 - 时区偏移量 = UTC

以北京时间转 **UTC** 时间为例：

北京位于东八区，北京时间与 UTC 相差 8 小时，时间转换规则为：

`北京时间 - 8 = UTC`

示例 **1**：北京时间 9:42 转换为 UTC 时间：42 09 - 00 08 = 42 01，表示 UTC 时间为凌晨 1:42 。

示例 **2**：北京时间凌晨 4:32 转换为 UTC 时间：32 04 - 00 08 = -68 03。若结果为负数，表示前一天，需要再转换一次：-68 03 + 00 24 = 32 20，表示 UTC 时间为前一天晚上 8:32 。

编写 **Crontab** 表达式

Crontab 的基本格式和值范围：`minute hour day month weekday`，对应的值范围如下表所示：

分钟	小时	日期	月份	星期
[0-59]	[0-23]	[1-31]	[1-12] 或 [JAN-DEC]	[0-6] 或 [SUN-SAT]

`minute hour day month weekday` 字段中允许使用的特殊字符包括：

- `,`：值列表分隔符，用于指定多个值。例如：`1,2,5,7,8,9`。
- `-`：用户自定义的值范围。例如：`2-4`，表示 2、3、4。
- `*`：表示整个时间段。例如用于分钟时，表示每分钟。
- `/`：用于指定值的增量。例如：`n/m` 表示从 n 开始，每次增加 m。

转换工具参考↗

常见示例：

- 输入 `30 18 25 12 *` 表示任务在 `12 月 25 日 18:30:00` 触发。
- 输入 `30 18 25 * 6` 表示任务在每周六的 `18:30:00` 触发。
- 输入 `30 18 * * 6` 表示任务在每周六的 `18:30:00` 触发。
- 输入 `* 18 * * *` 表示任务从 `18:00:00` 开始每分钟触发（包含 `18:00:00`）。
- 输入 `0 18 1,10,22 * *` 表示任务在每月的 1 日、10 日和 22 日的 `18:00:00` 触发。
- 输入 `0,30 18-23 * * *` 表示任务在每天 18:00 至 23:00 每小时的 0 分和 30 分触发。
- 输入 `* */1 * * *` 表示任务每分钟触发。
- 输入 `* 2-7/1 * * *` 表示任务每天凌晨 2 点至 7 点之间每分钟触发。
- 输入 `0 11 4 * mon-wed` 表示任务在每月 4 日及每周一至周三的 `11:00` 触发。

添加 ImagePullSecrets 到 ServiceAccount

如果镜像仓库需要认证，您需要将对应的 `ImagePullSecrets` 添加到应用使用的 `ServiceAccount` 中。这样可以确保应用能够成功从私有仓库拉取镜像。

目录

创建 ImagePullSecret
将 ImagePullSecret 添加到 ServiceAccount
验证新建 Pod 是否设置了 imagePullSecrets

创建 ImagePullSecret

要创建 ImagePullSecret，请参考创建 Secret中关于创建 ImagePullSecret 的详细步骤。

将 ImagePullSecret 添加到 ServiceAccount

如果您的应用 Pod 使用的是 `ServiceAccount` `example`，您可以将 `ImagePullSecret` 添加到应用所在命名空间中的 `example` `ServiceAccount`。

使用 patch 命令编辑 `ServiceAccount` `example`：

```
kubectl patch serviceaccount example -p '{"imagePullSecrets": [{"name":
"my-registry-creds"}]}' -n <namespace>
```

将 `<namespace>` 替换为应用所在的命名空间，将 `my-registry-creds` 替换为您创建的 `ImagePullSecret` 的名称。

您可以通过描述 `ServiceAccount` 来验证 `ImagePullSecret` 是否添加成功：

```
kubectl describe serviceaccount example -n <namespace>
Name:                example
Namespace:           <namespace>
Labels:              <none>
Annotations:         <none>
Image pull secrets:  my-registry-creds
Mountable secrets:   <none>
Tokens:              <none>
Events:              <none>
```

您应该能在 `Image pull secrets` 一栏看到已添加的 secret。

> **NOTE**
>
> 注意：如果您的 Pod 没有指定 `ServiceAccount`，则默认会使用该命名空间下的 `default` `ServiceAccount`。您可以用同样的方式将 `ImagePullSecret` 添加到 `default` `ServiceAccount`。

验证新建 Pod 是否设置了 imagePullSecrets

当您创建一个使用 `ServiceAccount` `example` 的新 Pod 时，该 Pod 会自动使用 `ServiceAccount` 中指定的 `ImagePullSecrets`。

您可以通过运行以下命令进行验证：

```
kubectl get pod <pod-name> -n <namespace> -o=jsonpath='{.spec.imagePullSecrets}'
```

Service Serving Certificate

目录

Overview

Features

Configuration and Deployment (For Administrators)

Prerequisites

1. Configure RBAC Permissions

2. Create Certificate

3. Configure Kyverno Policies

User Guide (For Developers)

Overview

Service 服务端证书为容器云平台中的 Service 提供自动化的 TLS 证书生成和管理功能。该能力实现了内部组件之间的安全 HTTPS 通信和流量加密，保障数据隐私和系统安全。

Features

该自动化证书配置机制具备以下关键特性：

- 零接触配置：开发者无需手动生成证书签名请求（CSR）或与证书颁发机构交互。证书颁发通过标准 Kubernetes 注解实现全自动化。

- 跨命名空间 **CA** 同步：免去手动将平台根 CA 复制到每个业务命名空间的繁琐操作。根 CA 证书自动同步，使服务间轻松验证身份。
- 自动生命周期管理：借助 `cert-manager` ，证书不仅自动颁发，还能在到期前无缝续期，降低运维负担，避免因证书过期导致的服务中断。
- 安全标准统一：确保 TLS 证书由统一的、集中管理的 `ClusterIssuer` 生成，维护平台上所有应用的一致安全标准。

Configuration and Deployment (For Administrators)

为实现上述功能，平台系统管理员需提前配置 `Kyverno` 相关的准入策略（`ClusterPolicy`）及必要的 RBAC 权限。

Prerequisites

操作前请确保集群已安装并启用 **Alauda Container Platform Compliance with Kyverno** 插件。安装详情请参见 Install Compliance Plugin。

1. Configure RBAC Permissions

为赋予 Kyverno 创建相关资源及调用接口的权限，将以下权限聚合至平台后台控制器：

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    rbac.kyverno.io/aggregate-to-background-controller: "true"
    rbac.kyverno.io/aggregate-to-admission-controller: "true"
    rbac.kyverno.io/aggregate-to-reports-controller: "true"
  name: kyverno:serving-certs
rules:
  - apiGroups:
      - cert-manager.io
    resources:
      - certificates
    verbs:
      - create
      - update
      - delete
      - get
      - list
      - watch
  - apiGroups:
      - ""
    resources:
      - secrets
    verbs:
      - create
      - update
      - delete
      - get
      - list
      - watch
```

2. Create Certificate

创建以下 Certificate 资源，作为服务根 CA：

```yaml
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: service-root-ca
  namespace: cert-manager
spec:
  commonName: ServiceRootCA
  isCA: true
  issuerRef:
    kind: ClusterIssuer
    name: cpaas-ca
  secretName: service-root-ca
```

3. Configure Kyverno Policies

创建并应用以下三个核心 `ClusterPolicy` 资源：

- **clone-ca-secret**：监听 Namespace 创建事件，为新命名空间生成 `service-root-ca` Secret 资源。
- **sync-ca-rotation**：监听 `service-root-ca` 变更，通过 API 获取真实 CA 证书内容，并同步 `ca.crt` 到目标命名空间的 Secret。
- **generate-service-cert**：监听带有 `service.alauda.io/serving-cert-secret-name` 注解的 Service 资源，基于其名称和命名空间生成支持泛域名的 `Certificate` 颁发请求。

将以下 YAML 内容保存为文件并执行应用（`kubectl apply -f`）：

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: clone-ca-secret
spec:
  background: true
  rules:
  - name: generate-ca-secret-shell
    match:
      any:
      - resources:
          kinds:
          - Namespace
    exclude:
      any:
      - resources:
          namespaces:
          - kube-system
          - cpaas-system
          - cert-manager
    generate:
      apiVersion: v1
      kind: Secret
      name: service-root-ca
      namespace: "{{request.object.metadata.name}}"
      synchronize: true
      generateExisting: true
      data:
        kind: Secret
        type: Opaque
        metadata:
          labels:
            app.kubernetes.io/managed-by: kyverno
            generate.kyverno.io/policy-name: clone-ca-secret

  - name: initial-sync-on-creation
    match:
      any:
      - resources:
          kinds:
          - Secret
          names:
          - service-root-ca
```

```
    exclude:
      any:
      - resources:
          namespaces:
          - cert-manager
          - kube-system
          - cpaas-system
    context:
    - name: cacrt
      apiCall:
        method: GET
        urlPath: "/api/v1/namespaces/cert-manager/secrets/service-root-c
a"
        jmesPath: 'data."ca.crt"'
    mutate:
      patchStrategicMerge:
        data:
          ca.crt: "{{ cacrt }}"
---
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: sync-ca-rotation
spec:
  mutateExistingOnPolicyUpdate: true
  background: true
  rules:
  - name: rotation-sync
    match:
      any:
      - resources:
          kinds:
          - Secret
          namespaces:
          - cert-manager
          names:
          - service-root-ca
    mutate:
      targets:
      - apiVersion: v1
        kind: Secret
        name: service-root-ca
        selector:
          matchLabels:
```

```yaml
            app.kubernetes.io/managed-by: kyverno
            generate.kyverno.io/policy-name: clone-ca-secret
      patchStrategicMerge:
        data:
          ca.crt: '{{ request.object.data."ca.crt" }}'
---
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: generate-service-cert
spec:
  background: true
  rules:
    - name: generate-cert
      match:
        any:
          - resources:
              kinds:
                - Service
              annotations:
                service.alauda.io/serving-cert-secret-name: "?*"
      generate:
        apiVersion: cert-manager.io/v1
        kind: Certificate
        name: "{{request.object.metadata.name}}-cert"
        namespace: "{{request.object.metadata.namespace}}"
        synchronize: true
        data:
          spec:
            secretName: '{{request.object.metadata.annotations."service.a
lauda.io/serving-cert-secret-name"}}'
            issuerRef:
              name: cpaas-ca
              kind: ClusterIssuer
            dnsNames:
              - "{{request.object.metadata.name}}"
              - "{{request.object.metadata.name}}.{{request.object.metada
ta.namespace}}"
              - "{{request.object.metadata.name}}.{{request.object.metada
ta.namespace}}.svc"
              - "{{request.object.metadata.name}}.{{request.object.metada
ta.namespace}}.svc.cluster.local"
```

User Guide (For Developers)

对于应用开发者，底层集群策略配置完成后，无需关心底层证书颁发逻辑。只需在定义业务 Service 资源时添加专属注解，即可实现自动证书配置：

1. 为 **Service** 添加注解

在业务命名空间（如 `my-namespace`）下创建或更新 Service，添加 `service.alauda.io/serving-cert-secret-name` 注解，指定生成的证书 Secret 名称。例如：

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-secure-service
  namespace: my-namespace
  annotations:
    service.alauda.io/serving-cert-secret-name: "my-secure-service-tls" # 将为您生成名为 my-secure-service-tls 的证书 Secret
spec:
  ports:
    - port: 443
      targetPort: 8443
  selector:
    app: my-app
```

2. 验证生成的证书 **Secret**

应用上述 Service 后：

- 检查自动生成的 TLS 证书 Secret 是否存在于命名空间：

```
kubectl get secret my-secure-service-tls -n my-namespace
```

- 检查用于验证的根 CA 证书是否也已自动复制到当前命名空间：

```
kubectl get secret service-root-ca -n my-namespace
```

3. 在 **Pod** 中挂载并使用证书

由于 CA 证书 Secret（`service-root-ca`）和当前应用的 TLS 服务端证书 Secret（`my-secure-service-tls`）均已就绪，您可以在目标应用的 `Deployment` 或 `StatefulSet` 等工作负载中，将它们作为数据卷挂载。

以下是一个具体的 `Deployment` YAML 示例，演示如何挂载这些证书：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: my-namespace
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app-container
          image: my-app-image:latest
          volumeMounts:
            # 挂载服务端证书
            - name: tls-cert
              mountPath: "/etc/tls/certs"
              readOnly: true
            # 挂载根 CA 证书
            - name: root-ca
              mountPath: "/etc/tls/ca"
              readOnly: true
      volumes:
        # 引用自动生成的 Service 证书 Secret
        - name: tls-cert
          secret:
            secretName: my-secure-service-tls
        # 引用自动同步的 CA 证书 Secret
        - name: root-ca
          secret:
            secretName: service-root-ca
```

更多关于通过 Web 控制台配置工作负载的详情，请参见 Configure Containers 部分。

镜像

Overview of images

Overview of images

Understanding containers and images

Images

Image registry

Image repository

Image tags

Image IDs

Containers

实用指南

Creating images

学习容器最佳实践

在镜像中包含元数据

管理镜像

镜像拉取策略

使用镜像拉取 Secret

Overview of images

目录

Understanding containers and images
Images
Image registry
Image repository
Image tags
Image IDs
Containers

Understanding containers and images

容器和镜像是在创建和管理容器化软件时需要理解的重要概念。镜像包含一组已准备好运行的软件，而容器是镜像的一个运行实例。不同的版本通过同一镜像名称上的不同标签来表示。

Images

Alauda Container Platform 中的容器基于 OCI 格式的容器镜像。镜像是一个二进制文件，包含运行单个容器所需的所有内容，以及描述其需求和能力的元数据。

你可以将其视为一种打包技术。容器只能访问镜像中定义的资源，除非在创建时授予了额外访问权限。通过在多个主机上部署相同的镜像到多个容器，并在它们之间进行负载均衡，Alauda Container Platform 可以为打包在镜像中的服务提供冗余和水平扩展。

你可以直接使用 `nerdctl` 或 `container` CLI 来构建镜像，但 Alauda Container Platform 也提供了构建器镜像，帮助通过将你的代码或配置添加到现有镜像中来创建新镜像。

由于应用程序会随着时间发展，单个镜像名称实际上可以指代同一镜像的多个不同版本。每个不同的镜像通过其哈希值唯一标识，哈希值是一个长的十六进制数字，如 fd44297e2ddb050ec4f...，通常缩短为 12 个字符，如 fd44297e2ddb。

Image registry

镜像仓库是一个内容服务器，可以存储和提供容器镜像。例如：

- Quay.io Container Registry ↗
- Alauda Container Platform Registry

仓库包含一个或多个镜像仓库集合，每个仓库包含一个或多个带标签的镜像。Alauda Container Platform 可以提供自己的镜像仓库来管理自定义容器镜像。

Image repository

镜像仓库是相关容器镜像及其标签的集合。例如，Alauda Container Platform 的 Jenkins 镜像位于仓库：

```
jenkins-2-centos7
```

Image tags

镜像标签是应用于仓库中容器镜像的标签，用于区分镜像流中的特定镜像。通常，标签表示某种版本号。例如，这里的 :v3 .11.59-2 是标签：

```
jenkins-2-centos7:v3.11.59-2
```

你可以为镜像添加额外的标签。例如，一个镜像可能被赋予标签 :v3 .11.59-2 和 :latest 。

Image IDs

镜像 ID 是一个 SHA（安全哈希算法）代码，可用于拉取镜像。SHA 镜像 ID 不会改变。特定的 SHA 标识符总是引用完全相同的容器镜像内容。例如：

```
jenkins-2-centos7@sha256:ab312bda324
```

Containers

Alauda Container Platform 应用的基本单元称为容器。Linux 容器技术是一种轻量级机制，用于隔离运行的进程，使其仅限于与指定资源交互。容器一词定义为容器镜像的一个特定运行或暂停的实例。

许多应用实例可以在单个主机上的容器中运行，彼此之间无法访问对方的进程、文件、网络等。通常，每个容器提供单一服务，通常称为微服务，如 Web 服务器或数据库，但容器也可用于任意工作负载。

Linux 内核多年来一直在集成容器技术的能力。早期的容器运行时项目引入了便捷的管理接口，用于在主机上运行隔离的应用。最近，Open Container Initiative ↗ 制定了容器格式和容器运行时的开放标准。Alauda Container Platform 和 Kubernetes 增加了跨多主机安装编排 OCI 格式容器的能力。

虽然在使用 Alauda Container Platform 时你不会直接与容器运行时交互，但理解它们的能力和术语对于理解它们在 Alauda Container Platform 中的角色以及你的应用如何在容器内运行非常重要。

实用指南

Creating images

学习容器最佳实践

在镜像中包含元数据

管理镜像

镜像拉取策略

使用镜像拉取 Secret

Creating images

了解如何基于预构建镜像创建您自己的容器镜像，这些预构建镜像已准备好为您提供帮助。该过程包括学习编写镜像的最佳实践、定义镜像的元数据、测试镜像，以及使用自定义构建工作流创建可用于 Alauda Container Platform Registry 的镜像。创建镜像后，您可以将其推送到 **Alauda Container Platform Registry**。

目录

学习容器最佳实践

通用容器镜像指南

在镜像中包含元数据

定义镜像元数据

学习容器最佳实践

在为 Alauda Container Platform 创建容器镜像时，作为镜像作者需要考虑一些最佳实践，以确保镜像使用者获得良好的体验。由于镜像旨在保持不可变并按原样使用，以下指南有助于确保您的镜像在 Alauda Container Platform 上高度可用且易于使用。

通用容器镜像指南

以下指南适用于一般容器镜像的创建，与镜像是否用于 Alauda Container Platform 无关。

复用镜像

尽可能基于合适的上游镜像使用 FROM 语句构建您的镜像。这确保您的镜像在上游镜像更新时能够轻松获取安全修复，而无需您直接更新依赖项。

此外，在 FROM 指令中使用标签，例如 `alpine:3.20`，可以让用户明确知道您的镜像基于哪个版本的镜像。使用非 latest 的标签可以避免您的镜像受到上游镜像最新版本中可能引入的破坏性更改的影响。

保持标签内兼容性

为自己的镜像打标签时，尽量保持标签内的向后兼容性。例如，如果您提供了一个名为 image 的镜像，当前包含版本 `1.0`，您可以提供一个标签 `image:v1`。当您更新镜像，只要它仍与原始镜像兼容，就可以继续使用 `image:v1` 标签，下游使用该标签的用户能够获得更新而不会出现兼容性问题。

如果之后发布了不兼容的更新，则切换到新标签，例如 `image:v2`。这允许下游用户根据需要升级到新版本，而不会因不兼容的镜像而意外中断。任何使用 `image:latest` 的下游用户都承担了引入不兼容更改的风险。

避免多进程

不要在一个容器内启动多个服务，例如数据库和 `SSHD`。这没有必要，因为容器轻量且可以轻松链接以协调多个进程。Alauda Container Platform 允许您通过将相关镜像分组到单个 pod 中，轻松实现共置和共管。

这种共置确保容器共享网络命名空间和存储以进行通信。更新也更不具破坏性，因为每个镜像可以更少频率且独立地更新。信号处理流程也更清晰，只有单个进程，无需管理信号路由到派生进程。

在包装脚本中使用 exec

许多镜像使用包装脚本在启动运行软件的进程之前进行一些设置。如果您的镜像使用此类脚本，该脚本应使用 `exec`，以便脚本进程被您的软件进程替换。如果不使用 `exec`，则容器运行时发送的信号会发送到包装脚本而非软件进程，这不是您想要的。

例如，您有一个包装脚本启动某个服务器进程。您使用 `podman run -i` 启动容器，运行包装脚本，进而启动您的进程。如果您想用 `CTRL+C` 关闭容器，且包装脚本使用了 `exec` 启动服务器进程，`podman` 会将 SIGINT 发送给服务器进程，一切按预期工作。如果未使用 `exec`，`podman` 会将 SIGINT 发送给包装脚本进程，您的服务器进程则继续运行，仿佛什么都没发生。

还要注意，您的进程在容器中以 `PID 1` 运行。这意味着如果主进程终止，整个容器会停止，取消由 `PID 1` 进程启动的任何子进程。

清理临时文件

删除构建过程中创建的所有临时文件，包括使用 `ADD` 命令添加的文件。例如，在执行 `yum install` 操作后运行 `yum clean` 命令。

您可以通过如下方式防止 `yum` 缓存进入镜像层：

```
RUN yum -y install mypackage && yum -y install myotherpackage && yum clean all -y
```

注意，如果写成：

```
RUN yum -y install mypackage
RUN yum -y install myotherpackage && yum clean all -y
```

则第一次 `yum` 调用会在该层留下额外文件，后续运行的 `yum clean` 无法删除这些文件。虽然最终镜像中看不到这些文件，但它们存在于底层层中。

当前容器构建过程不允许后续层运行的命令缩减前面层中删除文件所占用的空间，但未来可能会改变。这意味着如果您在后续层执行 `rm` 命令，虽然文件被隐藏，但不会减少下载镜像的总体大小。因此，像 `yum clean` 示例一样，最好在创建文件的同一命令中删除它们，避免写入层。

此外，在单个 `RUN` 语句中执行多个命令可以减少镜像层数，从而提升下载和解压速度。

按正确顺序放置指令

容器构建器从上到下读取 `Dockerfile` 并执行指令。每个成功执行的指令都会创建一个可在下次构建相同或其他镜像时复用的层。将不常更改的指令放在 `Dockerfile` 顶部非常重要，这样可以确保后续构建速度很快，因为缓存不会因上层更改而失效。

例如，如果您正在编写包含 `ADD` 命令安装正在迭代的文件和 `RUN` 命令执行 `yum install` 的 `Dockerfile`，最好将 `ADD` 命令放在最后：

```
FROM foo
RUN yum -y install mypackage && yum clean all -y
ADD myfile /test/myfile
```

这样每次编辑 `myfile` 并重新运行 `podman build` 时，系统会重用 `yum` 命令的缓存层，只为 `ADD` 操作生成新层。

如果写成：

```
FROM foo
ADD myfile /test/myfile
RUN yum -y install mypackage && yum clean all -y
```

每次更改 `myfile` 并重新构建时，`ADD` 操作会使 `RUN` 层缓存失效，因此 `yum` 操作也必须重新执行。

标记重要端口

`EXPOSE` 指令使容器中的端口对主机系统和其他容器可用。虽然可以在 `podman run` 调用时指定端口暴露，但在 `Dockerfile` 中使用 `EXPOSE` 指令通过显式声明软件运行所需端口，使人和软件更容易使用您的镜像：

- 暴露的端口会在基于您的镜像创建的容器的 `podman ps` 中显示。
- 暴露的端口存在于 `podman inspect` 返回的镜像元数据中。
- 暴露的端口在链接容器时会被关联。

设置环境变量

使用 `ENV` 指令设置环境变量是良好实践。例如设置项目版本号，方便用户无需查看 `Dockerfile` 即可了解版本；或者声明系统路径供其他进程使用，如 `JAVA_HOME`。

避免默认密码

避免设置默认密码。许多人扩展镜像时忘记删除或更改默认密码，这可能导致生产环境用户使用众所周知的密码，带来安全隐患。密码应通过环境变量进行配置。

如果确实设置了默认密码，确保容器启动时显示适当的警告信息，告知用户默认密码的值及如何更改，例如设置哪个环境变量。

避免 sshd

最好避免在镜像中运行 `sshd`。您可以使用 `podman exec` 访问本地主机上的容器。在集群中，使用 `kubectl exec` 访问由 Alauda Container Platform 管理的容器。在镜像中安装和运行 `sshd` 会增加攻击面并增加补丁负担。

使用卷存储持久数据

镜像应使用卷存储持久数据。这样，Alauda Container Platform 会将网络存储挂载到运行容器的节点上，如果容器迁移到新节点，存储会重新挂载到该节点。通过使用卷存储所有持久存储需求，即使容器重启或迁移，内容也能保留。如果镜像将数据写入容器内任意位置，则无法保证内容持久。

所有需要在容器销毁后仍保留的数据必须写入卷。容器引擎支持容器的 `readonly` 标志，可严格执行不向容器临时存储写入数据的良好实践。现在围绕该能力设计镜像，将来更易利用。

在 `Dockerfile` 中显式定义卷，方便镜像使用者了解运行镜像时必须定义哪些卷。

有关卷在 Alauda Container Platform 中使用的更多信息，请参见 Kubernetes 文档↗。

> 注意：
> 即使使用持久卷，您的镜像的每个实例都有自己的卷，实例间文件系统不共享。这意味着卷不能用于集群中共享状态。

在镜像中包含元数据

定义镜像元数据有助于 Alauda Container Platform 更好地使用您的容器镜像，为使用您镜像的开发者创造更佳体验。例如，您可以添加元数据提供镜像的有用描述，或建议可能还需要的其他镜像。

本主题仅定义当前用例所需的元数据，未来可能会添加更多元数据或用例。

定义镜像元数据

您可以在 `Dockerfile` 中使用 `LABEL` 指令定义镜像元数据。标签类似于环境变量，是附加到镜像或容器的键值对。标签不同于环境变量的是，它们对运行中的应用不可见，也可用于快速查找镜像和容器。

有关 `LABEL` 指令的更多信息，请参见 Dockerfile 参考↗。

标签名称通常带有命名空间。命名空间根据将使用标签的项目进行设置。对于 Kubernetes，命名空间为 io.k8s。

管理镜像

使用 Alauda Container Platform，你可以与镜像进行交互；具体取决于镜像所在的 registry、这些 registry 的任何身份验证要求，以及你希望构建和部署的行为方式。

目录

镜像拉取策略

镜像拉取策略概览

使用镜像拉取 Secret

允许 pod 引用来自其他受保护 registry 的镜像

创建拉取 Secret

在工作负载中使用拉取 Secret

镜像拉取策略

pod 中的每个容器都有一个容器镜像。创建镜像并将其推送到 registry 后，你就可以在 pod 中引用它。

镜像拉取策略概览

当 Alauda Container Platform 创建容器时，会使用容器的 `imagePullPolicy` 来判断是否应在启动容器前拉取镜像。`imagePullPolicy` 有三种可能的值：

`imagePullPolicy` 值表：

值	描述
Always	始终拉取镜像。
IfNotPresent	仅当节点上尚不存在该镜像时才拉取。
Never	从不拉取镜像。

如果未指定容器的 imagePullPolicy 参数，Alauda Container Platform 会根据镜像标签进行设置：

1. 如果标签为 latest，Alauda Container Platform 会将 imagePullPolicy 默认设置为 Always。
2. 否则，Alauda Container Platform 会将 imagePullPolicy 默认设置为 IfNotPresent。

使用镜像拉取 Secret

如果你使用的是 Alauda Container Platform 镜像 registry，那么你的 pod ServiceAccount 应该已经具有正确的权限，无需执行其他操作。

但是，在其他场景下，例如跨 Alauda Container Platform 项目引用镜像，或从受保护的 registry 中引用镜像时，则需要额外的配置步骤。

允许 pod 引用来自其他受保护 registry 的镜像

要从其他私有或受保护的 registry 拉取受保护的容器镜像，你必须根据容器客户端凭据（例如 `Podman`）创建一个拉取 Secret，并将其添加到你的 ServiceAccount 中。

容器客户端使用配置文件存储身份验证详细信息，以登录到受保护或不受保护的 registry：

如果你之前已登录到受保护或不受保护的 registry，这些文件会存储你的身份验证信息。

创建拉取 Secret

你可以获取用于从私有容器镜像 registry 或 repository 拉取镜像的 image pull secret。你可以参考 Pull an Image from a Private Registry↗。

在工作负载中使用拉取 Secret

你可以使用拉取 Secret，通过以下方法之一允许工作负载从私有 registry 拉取镜像：

- 将 Secret 关联到 `ServiceAccount`，这样会自动将该 Secret 应用于使用该 ServiceAccount 的所有 pod。
- 在 pod 规范中定义 `imagePullSecrets`，这对于 GitOps 或 ArgoCD 等环境很有用。

你可以通过将 Secret 添加到 ServiceAccount 中，为 pod 拉取镜像。请注意，ServiceAccount 的名称应与 pod 使用的 ServiceAccount 名称匹配。

示例输出：

```
apiVersion: v1
imagePullSecrets:
- name: default-cfg-123456
- name: <pull_secret_name>
kind: ServiceAccount
metadata:
  name: default
  namespace: default
secrets:
- name: <pull_secret_name>
```

除了将 Secret 关联到 ServiceAccount 之外，你还可以在 pod 或工作负载定义中直接引用它。这对于 ArgoCD 等 GitOps 工作流非常有用。例如：

示例 pod 规范：

```
apiVersion: v1
kind: Pod
metadata:
  name: <secure_pod_name>
spec:
  containers:
  - name: <container_name>
    image: your.registry.io/my-private-image
  imagePullSecrets:
  - name: <pull_secret_name>
```

示例 ArgoCD 工作流：

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: <example_workflow>
spec:
  entrypoint: <main_task>
  imagePullSecrets:
  - name: <pull_secret_name>
```

Alauda Container Platform

镜像仓库

介绍

介绍

原则与命名空间隔离

认证与授权

优势

应用场景

安装

通过 YAML 安装

何时使用此方法？

前提条件

通过 YAML 安装 Alauda Container Platforr

更新/卸载 Alauda Container Platform Regi

通过 Web UI 安装

何时使用此方法？

前提条件

使用 Web 控制台安装 Alauda Container Platform Registry 集

实用指南

Common CLI Command Operat

登录 Registry

为用户添加命名空间权限

为服务账户添加命名空间权限

拉取镜像

推送镜像

Using Alauda Container Platfor

Registry Access Guidelines

Deploy Sample Application

Cross-Namespace Access

Best Practices

Verification Checklist

Troubleshooting

Alauda Con

Overview

前提条件

数据备份

数据恢复

验证

方案通用性

升级

Alauda Container Platform Registry 升级指南

前提条件

概述

升级步骤

介绍

构建、存储和管理容器镜像是云原生应用开发流程的核心部分。Alauda Container Platform(ACP) 提供了一个高性能、高可用的内置容器镜像仓库服务，旨在为用户提供安全便捷的镜像存储和管理体验，大大简化平台内的应用开发、持续集成/持续交付（CI/CD）及应用部署流程。

Alauda Container Platform Registry 深度集成于平台架构中，相较于外部独立部署的镜像仓库，提供了更紧密的平台协作、更简化的配置以及更高效的内部访问能力。

目录

原则与命名空间隔离
认证与授权
认证
授权
优势
应用场景

原则与命名空间隔离

作为平台的核心组件之一，Alauda Container Platform 内置的镜像仓库以高可用方式运行在集群内部，并利用平台提供的持久化存储能力，确保镜像数据的安全可靠。

其核心设计理念之一是基于 Namespace 的逻辑隔离与管理。在 Registry 中，镜像仓库按命名空间组织。这意味着每个命名空间可视为该命名空间所属镜像的独立“区域”，不同命名空间之间

的镜像默认隔离，除非获得明确授权。

认证与授权

Alauda Container Platform Registry 的认证与授权机制深度集成 ACP 的平台级认证与授权系统，实现了细粒度到命名空间的访问控制：

认证

用户或自动化流程（例如平台上的 CI/CD 流水线、自动构建任务等）无需为 Registry 维护单独的账户密码。它们通过平台的标准认证机制进行认证（例如使用平台提供的 API Token、集成的企业身份系统等）。通过 CLI 或其他工具访问 Alauda Container Platform Registry 时，通常利用现有的平台登录会话或 ServiceAccount Token 实现透明认证。

授权

授权控制在命名空间级别实现。Alauda Container Platform Registry 中镜像仓库的 Pull 或 Push 权限取决于用户或 ServiceAccount 在对应命名空间内所拥有的平台角色和权限。

- 通常，命名空间的所有者或开发人员角色会自动获得该命名空间下镜像仓库的 Push 和 Pull 权限。
- 其他命名空间的用户或希望跨命名空间拉取镜像的用户，需由目标命名空间的管理员明确授予相应权限（例如通过 RBAC 绑定允许拉取镜像的角色），方可访问该命名空间内的镜像。
- 基于命名空间的授权机制确保了命名空间间镜像的隔离，提高安全性，避免未授权访问和修改。

优势

Alauda Container Platform Registry 的核心优势：

- 开箱即用： 快速部署私有镜像仓库，无需复杂配置。
- 访问灵活： 支持集群内及外部访问模式。
- 安全保障： 提供 RBAC 授权及镜像扫描能力。

- 高可用性： 通过复制机制保障服务连续性。
- 生产级： 在企业环境中验证，具备 SLA 保证。

应用场景

- 轻量级部署： 在低流量环境中实现精简的 Registry 方案，加速应用交付。
- 边缘计算： 为边缘集群提供自治管理的专用 Registry。
- 资源优化： 在基础设施利用率不足时，通过集成的 Source to Image (S2I) 方案展示完整工作流能力。

安装

通过 YAML 安装

何时使用此方法？

前提条件

通过 YAML 安装 Alauda Container Platforr

更新/卸载 Alauda Container Platform Regi

通过 Web UI 安装

何时使用此方法？

前提条件

使用 Web 控制台安装 Alauda Container Platform Registry 集र

更新/卸载 Alauda Container Platform Registry

通过 YAML 安装

目录

- 何时使用此方法？
- 前提条件
- 通过 YAML 安装 Alauda Container Platform Registry
  - 操作步骤
  - 配置参考
    - 必填字段
  - 验证
- 更新/卸载 Alauda Container Platform Registry
  - 更新
  - 卸载

何时使用此方法？

推荐用于：

- 具备 Kubernetes 专业知识，偏好手动操作的高级用户。
- 需要企业级存储（NAS、AWS S3、Ceph 等）的生产级部署。
- 需要对 TLS、ingress 进行细粒度控制的环境。
- 需要进行完整 **YAML** 自定义以实现高级配置的场景。

前提条件

- 安装 Alauda Container Platform Registry 集群插件到目标集群。
- 配置好 kubectl，访问目标 **Kubernetes** 集群。
- 具备创建集群范围资源的集群管理员权限。
- 获取已注册的域名（例如 registry.yourcompany.com）创建域名
- 提供有效的 **NAS** 存储（例如 NFS、GlusterFS 等）。
- （可选）提供有效的 **S3** 存储（例如 AWS S3、Ceph 等）。如果没有现成的 S3 存储，可在集群中部署 MinIO（内置 S3）实例 部署 MinIO。

通过 YAML 安装 **Alauda Container Platform Registry**

操作步骤

1. 创建一个名为 **registry-plugin.yaml** 的 **YAML** 配置文件，内容模板如下：

```yaml
apiVersion: cluster.alauda.io/v1alpha1
kind: ClusterPluginInstance
metadata:
  annotations:
    cpaas.io/display-name: image-registry
  labels:
    create-by: cluster-transformer
    manage-delete-by: cluster-transformer
    manage-update-by: cluster-transformer
  name: image-registry
spec:
  config:
    access:
      address: ''
      enabled: false
    fake:
      replicas: 2
    infra:
      enabled: false
    global:
      expose: false
      isIPv6: false
      replicas: 2
      oidc:
        ldapID: ''
      resources:
        limits:
          cpu: 500m
          memory: 512Mi
        requests:
          cpu: 250m
          memory: 256Mi
    ingress:
      enabled: true
      hosts:
        - name: <YOUR-DOMAIN> # [REQUIRED] 自定义域名
          tlsCert: <NAMESPACE>/<TLS-SECRET> # [REQUIRED] 命名空间/Secret名称
      ingressClassName: '<INGRESS-CLASS-NAME>' # [REQUIRED] IngressClassName
      insecure: false
    persistence:
      accessMode: ReadWriteMany
```

```
      nodes: ''
      path: <YOUR-HOSTPATH> # [REQUIRED] LocalVolume 本地路径
      size: <STORAGE-SIZE> # [REQUIRED] 存储大小（例如 10Gi）
      storageClass: <STORAGE-CLASS-NAME> # [REQUIRED] StorageClass 名称
      type: StorageClass
    s3storage:
      bucket: <S3-BUCKET-NAME> # [REQUIRED] S3 桶名称
      enabled: false # 本地存储设置为 false
      env:
        REGISTRY_STORAGE_S3_SKIPVERIFY: false # 自签名证书设置为 true
      region: <S3-REGION> # S3 区域
      regionEndpoint: <S3-ENDPOINT> # S3 端点
      secretName: <S3-CREDENTIALS-SECRET> # S3 凭证 Secret
    service:
      nodePort: ''
      type: ClusterIP
  pluginName: image-registry
```

2. 根据实际环境自定义以下字段：

```
spec:
  config:
    global:
      oidc:
        ldapID: '<LDAP-ID>' # LDAP ID
    infra:
      enabled: false  # 是否将组件部署到 infra 节点，默认 false 表示所有节点
    ingress:
      hosts:
        - name: '<YOUR-DOMAIN>' # 例如 registry.your-company.com
          tlsCert: '<NAMESPACE>/<TLS-SECRET>' # 例如 cpaas-system/tls-secret
      ingressClassName: '<INGRESS-CLASS-NAME>' # 例如 cluster-alb-1
    persistence:
      size: '<STORAGE-SIZE>' # 例如 10Gi
      storageClass: '<STORAGE-CLASS-NAME>' # 例如 cpaas-system-storage
    s3storage:
      bucket: '<S3-BUCKET-NAME>' # 例如 prod-registry
      region: '<S3-REGION>' # 例如 us-west-1
      regionEndpoint: '<S3-ENDPOINT>' # 例如 https://s3.amazonaws.com
      secretName: '<S3-CREDENTIALS-SECRET>' # 包含 AWS_ACCESS_KEY_ID/AWS_SECRET_ACCESS_KEY 的 Secret
      env:
        REGISTRY_STORAGE_S3_SKIPVERIFY: 'true' # 自签名证书设置为 "true"
```

3. 创建 **S3** 凭证 **Secret** 的示例命令：

```
kubectl create secret generic <S3-CREDENTIALS-SECRET> \
  --from-literal=access-key-id=<YOUR-S3-ACCESS-KEY-ID> \
  --from-literal=secret-access-key=<YOUR-S3-SECRET-ACCESS-KEY> \
  -n cpaas-system
```

将 `<S3-CREDENTIALS-SECRET>` 替换为你的 S3 凭证 Secret 名称。

4. 将配置应用到集群：

```
kubectl apply -f registry-plugin.yaml
```

配置参考

必填字段

参数	说明	示例值
`spec.config.global.oidc.ldapID`	OIDC 认证的 LDAP ID	`ldap-te`
`spec.config.ingress.hosts[0].name`	镜像仓库访问的自定义域名	`registr`
`spec.config.ingress.hosts[0].tlsCert`	TLS 证书 Secret 引用（命名空间/Secret 名称）	`cpaas-s` `tls`
`spec.config.ingress.ingressClassName`	镜像仓库的 Ingress 类名	`cluster`
`spec.config.persistence.size`	镜像仓库存储大小	`10Gi`
`spec.config.persistence.storageClass`	镜像仓库使用的 StorageClass 名称	`nfs-stc`
`spec.config.s3storage.bucket`	镜像存储使用的 S3 桶名称	`prod-im`
`spec.config.s3storage.region`	S3 存储的 AWS 区域	`us-west`
`spec.config.s3storage.regionEndpoint`	S3 服务端点 URL	`https:/`
`spec.config.s3storage.secretName`	包含 S3 凭证的 Secret	`s3-acce`
`spec.config.s3storage.env.REGISTRY_STORAGE_S3_SKIPVERIFY`	自签名证书时设置为 `true`	`true`

参数	说明	示例值
`spec.config.infra.enabled`	是否将组件部署到 infra 节点或所有节点	`false`

验证

1. 查看插件状态：

```
kubectl get clusterplugininstances image-registry -o yaml
```

2. 查看镜像仓库 Pod：

```
kubectl get pods -n cpaas-system -l app=image-registry
```

更新/卸载 **Alauda Container Platform Registry**

更新

在 global 集群执行以下命令，根据上述参数说明修改资源值完成更新：

```
# <CLUSTER-NAME> 是插件安装的集群名称
kubectl edit -n cpaas-system \
  $(kubectl get moduleinfo -n cpaas-system -l cpaas.io/cluster-name=<CLUSTER-NAME>,cpaas.io/module-name=image-registry -o name)
```

卸载

在 global 集群执行以下命令：

```
# <CLUSTER-NAME> 是插件安装的集群名称
kubectl get moduleinfo -n cpaas-system -l cpaas.io/cluster-name=<CLUSTER-NAME>,cpaas.io/module-name=image-registry -o name | xargs kubectl delete -n cpaas-system
```

通过 Web UI 安装

目录

何时使用此方法？
前提条件
使用 Web 控制台安装 Alauda Container Platform Registry 集群插件
操作步骤
验证
更新/卸载 Alauda Container Platform Registry

何时使用此方法？

推荐使用场景：

- 首次使用者，偏好有引导的可视化界面。
- 非生产环境中的快速概念验证部署。
- 具备有限 **Kubernetes** 经验的团队，寻求简化的部署流程。
- 需要最小化自定义的场景（例如，默认存储配置）。
- 基础网络配置，无特定 ingress 规则需求。
- 针对高可用性的 **StorageClass** 配置。

不推荐使用场景：

- 生产环境中需要高级存储（如 S3 存储）配置。

- 需要特定 ingress 规则的网络配置。

前提条件

- 使用 Cluster Plugin 机制，安装 **Alauda Container Platform Registry** 集群插件到目标集群。

使用 Web 控制台安装 Alauda Container Platform Registry 集群插件

操作步骤

1. 登录并进入 管理员 页面。
2. 点击 **Marketplace** > **Cluster Plugins**，进入 **Cluster Plugins** 列表页面。
3. 找到 **Alauda Container Platform Registry** 集群插件，点击 **Install**，进入安装页面。
4. 按照以下参数说明配置参数，点击 **Install** 完成部署。

参数说明如下：

参数	说明
Expose Service	启用后，管理员可通过访问地址对镜像仓库进行外部管理。此操作存在较大安全风险，需谨慎启用。
Enable IPv6	当集群使用 IPv6 单栈网络时启用此选项。
NodePort	在启用 Expose Service 时，配置 NodePort 以允许通过该端口对 Registry 进行外部访问。
Storage Type	选择存储类型。支持类型：LocalVolume 和 StorageClass。
Nodes	选择用于运行 Registry 服务以存储和分发镜像的节点。（仅当存储类型为 LocalVolume 时可用）

参数	说明
StorageClass	选择 StorageClass。当副本数超过 1 时，需选择具备 RWX（ReadWriteMany）能力的存储（如文件存储）以确保高可用性。（仅当存储类型为 StorageClass 时可用）
Storage Size	分配给 Registry 的存储容量（单位：Gi）。
Replicas	配置 Registry Pod 的副本数： • **LocalVolume**：默认 1（固定） • **StorageClass**：默认 3（可调整）
Resource Requirements	定义 Registry Pod 的 CPU 和内存资源请求及限制。

验证

1. 进入 **Marketplace** > **Cluster Plugins**，确认插件状态显示为 **Installed**。
2. 点击插件名称查看详情。
3. 复制 **Registry Address**，使用容器客户端（如 Podman）进行镜像的推送/拉取操作。

更新/卸载 Alauda Container Platform Registry

您可以在列表页面或详情页面对 **Alauda Container Platform Registry** 插件进行更新或卸载操作。

实用指南

Common CLI Command Operat

登录 Registry

为用户添加命名空间权限

为服务账户添加命名空间权限

拉取镜像

推送镜像

Using Alauda Container Platfor

Registry Access Guidelines

Deploy Sample Application

Cross-Namespace Access

Best Practices

Verification Checklist

Troubleshooting

Alauda Con

Overview

前提条件

数据备份

数据恢复

验证

方案通用性

Common CLI Command Operations

Alauda Container Platform 提供命令行工具，供用户与 Alauda Container Platform Registry 交互。以下是一些常用操作和命令示例：

假设集群的 Alauda Container Platform Registry 服务地址为 registry.cluster.local，您当前操作的命名空间为 my-ns。

> 请联系技术服务获取 kubectl-acp 插件，并确保其已正确安装在您的环境中。

目录

登录 Registry

为用户添加命名空间权限

为服务账户添加命名空间权限

拉取镜像

推送镜像

登录 Registry

通过登录 ACP 来登录集群的 Registry。

```
kubectl acp login <ACP-endpoint>
```

为用户添加命名空间权限

为用户添加命名空间拉取权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-pull
er --user=<username> -n <namespace>
```

为用户添加命名空间推送权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-push
er --user=<username> -n <namespace>
```

为服务账户添加命名空间权限

为服务账户添加命名空间拉取权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-pull
er --serviceaccount=<namespace>:<serviceaccount-name> -n <namespace>
```

为服务账户添加命名空间推送权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-push
er --serviceaccount=<namespace>:<serviceaccount-name> -n <namespace>
```

拉取镜像

从 Registry 拉取镜像到集群内部（例如用于 Pod 部署）。

```
# 从当前命名空间（my-ns）的 Registry 拉取名为 my-app，标签为 latest 的镜像
kubectl acp pull registry.cluster.local/my-ns/my-app:latest

# 从其他命名空间（例如 shared-ns）拉取镜像（需要拥有 shared-ns 命名空间的拉取权限）
kubectl acp pull registry.cluster.local/shared-ns/base-image:latest
```

该命令会验证您在目标命名空间的身份和拉取权限，然后从 Registry 拉取镜像。

推送镜像

将本地构建的镜像或从其他地方拉取的镜像推送到 Registry 的指定命名空间。

您需要先使用标准容器器命令行工具（如 podman）将本地镜像打标签（tag）为目标 Registry 的地址和命名空间格式。

```
# 打标签为目标地址：
podman tag my-app:latest registry.cluster.local/my-ns/my-app:v1

# 使用 kubectl 命令将其推送到当前命名空间（my-ns）的 Registry
kubectl acp push registry.cluster.local/my-ns/my-app:v1
```

将远程镜像仓库中的镜像推送到 Alauda Container Platform Registry 的指定命名空间。

```
# 如果您的远程镜像仓库中有镜像 remote.registry.io/demo/my-app:latest
# 使用 kubectl 命令将其推送到 Registry 的命名空间（my-ns）
kubectl acp push remote.registry.io/demo/my-app:latest registry.cluster.local/my-ns/my-app:latest
```

该命令会验证您在 my-ns 命名空间内的身份和推送权限，然后将本地打标签的镜像上传到 Registry。

Using Alauda Container Platform Registry in Kubernetes Clusters

Alauda Container Platform (ACP) Registry 为 Kubernetes 工作负载提供安全的容器镜像管理。

目录

- Registry Access Guidelines
- Deploy Sample Application
- Cross-Namespace Access
  - Example Role Binding
- Best Practices
- Verification Checklist
- Troubleshooting

Registry Access Guidelines

- 推荐使用内部地址：对于存储在集群注册表中的镜像，部署时优先使用集群内部服务地址 `image-registry.cpaas-system.svc`，以确保最佳的网络性能并避免不必要的外部路由。
- 外部地址使用场景：外部 ingress 域名（例如 `registry.cluster.local`）主要用于：
  - 集群外部的镜像推送/拉取（例如开发人员机器、CI/CD 系统）
  - 需要访问注册表的集群外部操作

Deploy Sample Application

1. 在 `my-ns` 命名空间中创建名为 `my-app` 的应用。
2. 将应用镜像存储在注册表地址 `image-registry.cpaas-system.svc/my-ns/my-app:v1`。
3. 每个命名空间的 默认 ServiceAccount 会自动配置 imagePullSecret，用于访问 `image-registry.cpaas-system.svc` 上的镜像。

示例 Deployment：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: my-ns
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: main-container
        image: image-registry.cpaas-system.svc/my-ns/my-app:v1
        ports:
        - containerPort: 8080
```

Cross-Namespace Access

为了允许 `my-ns` 的用户拉取 `shared-ns` 的镜像，`shared-ns` 的管理员可以创建角色绑定以授予必要权限。

Example Role Binding

```
# 访问共享命名空间的镜像（需要权限）
kubectl create rolebinding cross-ns-pull \
  --clusterrole=system:image-puller \
  --serviceaccount=my-ns:default \
  -n shared-ns
```

Best Practices

- 注册表使用：部署时始终使用 `image-registry.cpaas-system.svc`，确保安全性和性能。
- 命名空间隔离：利用命名空间隔离实现更好的安全性和镜像管理。
  - 使用基于命名空间的镜像路径：`image-registry.cpaas-system.svc/<namespace>/<image>:<tag>`。
- 访问控制：通过角色绑定管理跨命名空间的用户和 ServiceAccount 访问权限。

Verification Checklist

1. 验证 `my-ns` 默认 ServiceAccount 的镜像访问权限：

```
kubectl auth can-i get images.registry.alauda.io --namespace my-ns --as
=system:serviceaccount:my-ns:default
```

2. 验证 `my-ns` 中某用户的镜像访问权限：

```
kubectl auth can-i get images.registry.alauda.io --namespace my-ns --as
=<username>
```

Troubleshooting

- 镜像拉取错误：检查 Pod 规范中的 imagePullSecrets，确保配置正确。
- 权限拒绝：确认用户或 ServiceAccount 在目标命名空间拥有必要的角色绑定。

- 网络问题：验证网络策略和服务配置，确保能连接到内部注册表。
- **DNS** 解析失败：检查节点上的 `/etc/hosts` 文件内容，确保 `image-registry.cpaas-system.svc` 的 DNS 解析配置正确。
  - 验证节点的 /etc/hosts 配置，确保 `image-registry.cpaas-system.svc` 的 DNS 解析正确
  - 注册表服务映射示例（image-registry 服务的 ClusterIP）：

    ```
    # /etc/hosts
    127.0.0.1   localhost localhost.localdomain
    10.4.216.11 image-registry.cpaas-system image-registry.cpaas-system.svc image-registry.cpaas-system.svc.cluster.local # cpaas-generated-node-resolver
    ```

  - 如何获取 `image-registry` 当前 **ClusterIP**：

    ```
    kubectl get svc -n cpaas-system image-registry -o jsonpath='{.spec.clusterIP}'
    ```

Alauda Container Platform Registry 数据备份和恢复

目录

Overview
前提条件
数据备份
步骤 1：获取当前 S3 配置
步骤 2：执行 S3 Bucket 数据备份
数据恢复
步骤 1：准备备份数据
步骤 2：更新 ModuleInfo 配置
验证
检查模块状态（在 global 集群）
验证数据访问（API 测试）
功能测试
方案通用性

Overview

本方案提供了在 Alauda Container Platform (ACP) 中，针对使用 S3 兼容对象存储的 `Alauda Container Platform Registry` 数据备份与恢复的指导。

核心理念：将镜像数据本身（存储于 S3）与 `cluster plugin` 配置（定义于 Kubernetes 的 `ModuleInfo` 自定义资源）解耦管理。

- 备份：从 `ModuleInfo` 资源中获取 S3 配置，并备份指定存储 `bucket` 中的数据。
- 恢复：在新集群安装 `cluster plugin` 后，更新其 `ModuleInfo` 资源中的 S3 配置，指向已恢复数据的 bucket，从而完成数据接入。

优势：

- 操作解耦：数据备份/恢复独立于 ACP `cluster plugin` 的部署和升级流程。
- 配置驱动：所有连接信息通过声明式的 `ModuleInfo` 资源管理，确保变更安全可靠。
- 可扩展：该模式可扩展至其他存储后端（如本地文件系统、StorageClass、NAS 等）。

前提条件

- 拥有对目标 Kubernetes 集群的 `kubectl` 访问权限及相应操作权限。
- 拥有访问和操作用于存储镜像数据的 S3 兼容存储的凭据及客户端工具（如 awscli、rclone、minio-client）。
- 已安装并配置 `Alauda Container Platform Registry` 集群插件，其 `ModuleInfo` 资源存在且处于健康状态。
- 准备了独立且容量充足的备份存储空间（例如另一个 S3 bucket）。

数据备份

本阶段目标是获取当前生产环境的 S3 配置，并对存储桶中的镜像数据进行全量备份。

步骤 1：获取当前 S3 配置

从管理 `Alauda Container Platform Registry` 集群插件的 `ModuleInfo` 资源中提取 S3 存储配置，该信息是备份操作的基础。

请在 ACP 的 `global` 集群上执行以下命令：

```
# 1. 确认 image-registry 模块对应的 ModuleInfo 资源名称
MODULE_INFO_NAME=$(kubectl get moduleinfo -l cpaas.io/module-name=image-registry -o jsonpath='{.items[0].metadata.name}')
echo "目标 ModuleInfo 资源: $MODULE_INFO_NAME"

# 2. 提取关键 S3 配置信息
S3_BUCKET=$(kubectl get moduleinfo $MODULE_INFO_NAME -o jsonpath='{.spec.config.s3storage.bucket}')
S3_ENDPOINT=$(kubectl get moduleinfo $MODULE_INFO_NAME -o jsonpath='{.spec.config.s3storage.regionEndpoint}')
S3_REGION=$(kubectl get moduleinfo $MODULE_INFO_NAME -o jsonpath='{.spec.config.s3storage.region}')
S3_SECRET_NAME=$(kubectl get moduleinfo $MODULE_INFO_NAME -o jsonpath='{.spec.config.s3storage.secretName}')

# 3. 从 Secret 中获取访问密钥（通常为 access-key-id 和 secret-access-key）
# 注意：输出为 Base64 编码，需要相应解码。
kubectl get secret -n cpaas-system $S3_SECRET_NAME -o jsonpath='{.data}'
```

关键变量说明：

- `S3_BUCKET`：实际存储镜像数据的源 bucket 名称。
- `S3_ENDPOINT`：连接 S3 兼容服务的端点 URL。
- `S3_REGION`：S3 服务的地域标识。
- `S3_SECRET_NAME`：存储认证密钥的 Kubernetes Secret 名称。

步骤 2：执行 S3 Bucket 数据备份

使用您选择的 S3 客户端工具，利用上一步获取的配置对源 bucket 的数据进行全量备份。

操作逻辑：

- 使用端点（$S3_ENDPOINT）、地域（$S3_REGION）及从 Secret 解码得到的访问密钥配置客户端。
- 执行同步或复制命令，将源 bucket（$S3_BUCKET）中的所有数据备份至准备好的独立备份位置（例如另一个 S3 bucket 或路径）。
- 记录备份时间戳、所用 bucket 名称和端点信息，并与备份文件一同归档保存。

数据恢复

本阶段假设已通过平台在目标环境（新集群或修复集群）成功安装了 `Alauda Container Platform Registry` 集群插件，目标是修改其配置以访问恢复后的镜像数据。

步骤 **1**：准备备份数据

使用您选择的 S3 客户端工具，将备份的镜像数据恢复到确定可访问的目标 **S3** 存储 **`bucket`** 中。例如，恢复到名为 `registry-bucket-restored` 的新 bucket，确保您对该目标 bucket 拥有写权限。

步骤 **2**：更新 **ModuleInfo** 配置

恢复的关键在于更新新 `cluster plugin` 的 `ModuleInfo` 资源，将其 S3 配置指向包含备份数据的目标 bucket。

1. 确定新的 **S3** 连接信息：

- `NEW_BUCKET`：备份数据恢复后的目标 **bucket** 名称（如 registry-bucket-restored）。
- `NEW_ENDPOINT`：目标 S3 服务的端点，若 S3 服务地址与备份时相同，则保持不变。
- `NEW_REGION`：目标 S3 服务的地域。
- `NEW_SECRET_NAME`：具有目标 bucket 读写权限的 Kubernetes Secret 名称。若访问密钥未变，仍为 `$S3_SECRET_NAME`。

2. 更新 **ModuleInfo** 资源： 使用 kubectl patch 命令直接更新 `ModuleInfo` 中的 S3 配置部分，平台控制器会自动将此变更同步到相关的 Deployment、Pod 等资源。

```
# 执行配置更新
kubectl patch moduleinfo $MODULE_INFO_NAME --type=merge -p '{
  "spec": {
    "config": {
      "s3storage": {
        "bucket": "'"$NEW_BUCKET"'",
        "regionEndpoint": "'"$NEW_ENDPOINT"'",
        "region": "'"$NEW_REGION"'",
        "secretName": "'"$NEW_SECRET_NAME"'"
      }
    }
  }
}'
```

关键点：此操作会触发 `Alauda Container Platform Registry` 相关 Pod 的滚动更新，新启动的 Pod 将使用新配置连接指定的目标存储 bucket。

验证

更新完成后，按照以下步骤验证数据恢复成功及服务正常运行。

检查模块状态（在 **global** 集群）

```
# 检查 Pod 是否已成功重启并使用新配置运行
kubectl get pods -n cpaas-system -l app=image-registry
# 查看 Pod 日志确认无 S3 连接错误
kubectl logs -n cpaas-system -l app=image-registry -c registry --tail=50
```

验证数据访问（**API** 测试）

通过 Registry 的 API 接口直接验证是否能读取恢复的镜像数据。

```
# 获取 Registry 服务访问地址（假设为 ClusterIP 类型）
REGISTRY_SVC_IP=$(kubectl get svc -n cpaas-system image-registry -o jsonp
ath='{.spec.clusterIP}')

# 测试 1：查询仓库目录
curl -s http://$REGISTRY_SVC_IP/v2/_catalog | jq .
# 预期成功返回：{"repositories":["image1","image2",...]}

# 测试 2：查询指定镜像的标签列表（例如镜像名为 `myns/nginx`）
curl -s http://$REGISTRY_SVC_IP/v2/myns/nginx/tags/list | jq .
# 预期成功返回：{"name":"myns/nginx","tags":["v1.0","latest",...]}
```

功能测试

尝试从恢复后的 Registry 拉取已知镜像，或推送新镜像，全面验证读写功能。

方案通用性

虽然本方案以 S3 存储为例，但其设计模式适用于 **Registry** 支持的各类存储后端（如本地文件系统、StorageClass、NAS 等）。

通用原则：无论存储类型，核心备份与恢复流程一致。首先从对应的 `ModuleInfo` 资源中提取存储连接参数（如 `s3storage`、`persistence` 等配置块），然后使用相应存储工具进行数据备份。恢复时，将数据恢复至目标位置后，更新 `ModuleInfo` 中对应配置字段，平台自动引导新部署实例访问该位置。

核心价值：通过利用统一的配置抽象层（`ModuleInfo`），本方案实现了数据备份/恢复流程与具体存储实现及 Kubernetes 应用部署的解耦，达成标准化管理与良好扩展性。

升级

Alauda Container Platform Registry 升级指南

前提条件

概述

升级步骤

Alauda Container Platform

Alauda Container Platform Registry 升级指南

目录

前提条件

概述

升级步骤

环境检查与预处理

未安装旧插件

已安装旧插件

S3 存储迁移

NFS 存储迁移

本地存储迁移

前提条件

1. 拥有 Alauda Container Platform 的管理员权限。
2. 旧插件指 Alauda Container Platform v4.1（含）及更早版本。
3. 新插件指 Alauda Container Platform v4.2（含）及更高版本。

概述

本文档提供了将 `Old Plugin` 升级到 `New Plugin` 的操作指南。由于集群插件名称发生变化，需根据 存储类型 进行手动干预。

升级步骤

环境检查与预处理

未安装旧插件

如果从未安装过旧插件，只需从 [Marketplace/Cluster Plugins] 清理旧插件：

```
# 在 global 集群执行
kubectl delete moduleplugins internal-docker-registry
kubectl get moduleconfig -l cpaas.io/module-name=internal-docker-registry
-o name | xargs kubectl delete
```

已安装旧插件

如果在任一集群中安装了旧插件，请根据您的 存储类型 按照迁移操作步骤执行。

S3 存储迁移

特点：自动数据迁移，无需手动操作

操作步骤：

1. 备份旧插件配置：（在安装旧插件的业务集群执行）

```
kubectl get clusterplugininstances internal-docker-registry -o yaml > b
ackup-clusterplugininstances.yaml
```

2. 卸载旧插件：（在 global 集群执行）

```
# 将 <CLUSTER-NAME> 替换为实际安装插件的集群名称
kubectl get moduleinfo -l cpaas.io/cluster-name=<CLUSTER-NAME>,cpaas.io/module-name=internal-docker-registry -o name | xargs kubectl delete
```

3. 安装新插件：（在将安装插件的业务集群执行）

- 编辑备份文件 `backup-clusterplugininstances.yaml`，将所有 `internal-docker-registry` 替换为 `image-registry`
- 应用新配置：

```
kubectl apply -f backup-clusterplugininstances.yaml
```

4. 从 [Marketplace/Cluster Plugins] 清理旧插件：（在 global 集群执行）

```
kubectl delete moduleplugins internal-docker-registry
kubectl get moduleconfig -l cpaas.io/module-name=internal-docker-registry -o name | xargs kubectl delete
```

NFS 存储迁移

特点：需手动保留 PV 并修改回收策略。

操作步骤：

1. 记录 PV 信息：（在安装旧插件的业务集群执行）

```
OLD_PV=$(kubectl get pvc internal-docker-registry -n cpaas-system -o jsonpath='{.spec.volumeName}')
echo "旧 PV 名称: $OLD_PV"
```

2. 修改 PV 回收策略：（在安装旧插件的业务集群执行）

```
kubectl patch pv $OLD_PV -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

3. 备份旧插件配置（同 S3 步骤 1）

4. 卸载旧插件（同 S3 步骤 2）
5. 释放 PV 绑定：（在安装旧插件的业务集群执行）

```
kubectl patch pv $OLD_PV -p '{"spec":{"claimRef":null}}'
```

6. 安装新插件（重要：必须为新插件设置 `config.persistence.volumeName` 以保留旧数据）

- 编辑配置文件 `backup-clusterplugininstances.yaml`，设置 config.persistence.volumeName: `<OLD_PV>`
- 应用配置（同 S3 步骤 3）

7. 从 [Marketplace/Cluster Plugins] 清理旧插件（同 S3 步骤 4）

本地存储迁移

特点：需手动复制数据

操作步骤：

1. 备份旧插件配置（同 S3 步骤 1）
2. 卸载旧插件（同 S3 步骤 2）
3. 安装新插件（同 S3 步骤 3）
4. 数据迁移：

- ssh 登录旧插件 Pod 所在节点，复制旧目录数据到新目录，保持所有文件权限：

```
sudo cp -a /cpaas/internal-docker-registry/* /cpaas/image-registry/
```

5. 从 [Marketplace/Cluster Plugins] 清理旧插件（同 S3 步骤 4）

Source to Image

概览

介绍

Source to Image 概念

核心功能

核心优势

应用场景

使用限制

架构

生命周期策略

版本生命周期时间线

发版日志

Alauda Contain

安装

Installing Alauda Container Platform Builds

Prerequisites

Procedure

升级

升级 Alauda Container Platform Builds

前提条件

操作步骤

操作指南

Managing applications created from Code

主要功能

优势

前提条件

操作步骤

相关操作

实用指南

Creating an application from Code

Prerequisites

Procedure

概览

介绍

Source to Image 概念

核心功能

核心优势

应用场景

使用限制

架构

生命周期策略

版本生命周期时间线

发版日志

Alauda Contain

介绍

Alauda Container Platform Builds 是由 灵雀云容器平台 提供的一款云原生容器工具，集成了 Source to Image (S2I) 功能和自动化流水线。它通过支持多种编程语言（包括 Java、Go、Python 和 Node.js）的全自动 CI/CD 流水线，加速企业云原生转型。此外，Alauda Container Platform Builds 提供可视化发布管理，并与 Kubernetes 原生工具如 Helm 和 GitOps 无缝集成，确保从开发到生产的高效应用生命周期管理。

目录

Source to Image 概念
核心功能
核心优势
应用场景
使用限制

Source to Image 概念

Source to Image (S2I) 是一种从源代码构建可复现容器镜像的工具和工作流程。它将应用的源代码注入到预定义的构建镜像中，自动完成编译、打包等步骤，最终生成可运行的容器镜像。这样开发者可以更多地专注于业务代码开发，而无需关心容器化的细节。

核心功能

Alauda Container Platform Builds 促进了从代码到应用的全栈云原生工作流，支持多语言构建和可视化发布管理。它利用 Kubernetes 原生能力将源代码转换为可运行的容器镜像，确保无缝集成到完整的云原生平台中。

- 多语言构建：支持 Java、Go、Python 和 Node.js 等多种编程语言的应用构建，满足多样化的开发需求。
- 可视化界面：提供直观的界面，方便您轻松创建、配置和管理构建任务，无需深厚技术背景。
- 全生命周期管理：覆盖从代码提交到应用部署的整个生命周期，实现构建、部署和运维管理自动化。
- 深度集成：与您的 Container Platform product 无缝集成，提供流畅的开发体验。
- 高扩展性：支持自定义插件和扩展，满足您的特定需求。

核心优势

- 加速开发：简化构建流程，加快应用交付速度。
- 增强灵活性：支持多种编程语言的构建。
- 提升效率：自动化构建和部署流程，减少人工干预。
- 提高可靠性：提供详细的构建日志和可视化监控，便于故障排查。

应用场景

S2I 的主要应用场景如下：

- Web 应用
  S2I 支持多种编程语言，如 Java、Go、Python 和 Node.js。借助 灵雀云容器平台 的应用管理能力，只需输入代码仓库 URL，即可快速构建和部署 Web 应用。
- CI/CD
  S2I 与 DevOps 流水线无缝集成，利用 Kubernetes 原生工具如 Helm 和 GitOps 自动化镜像构建和部署流程，实现应用的持续集成和持续交付。

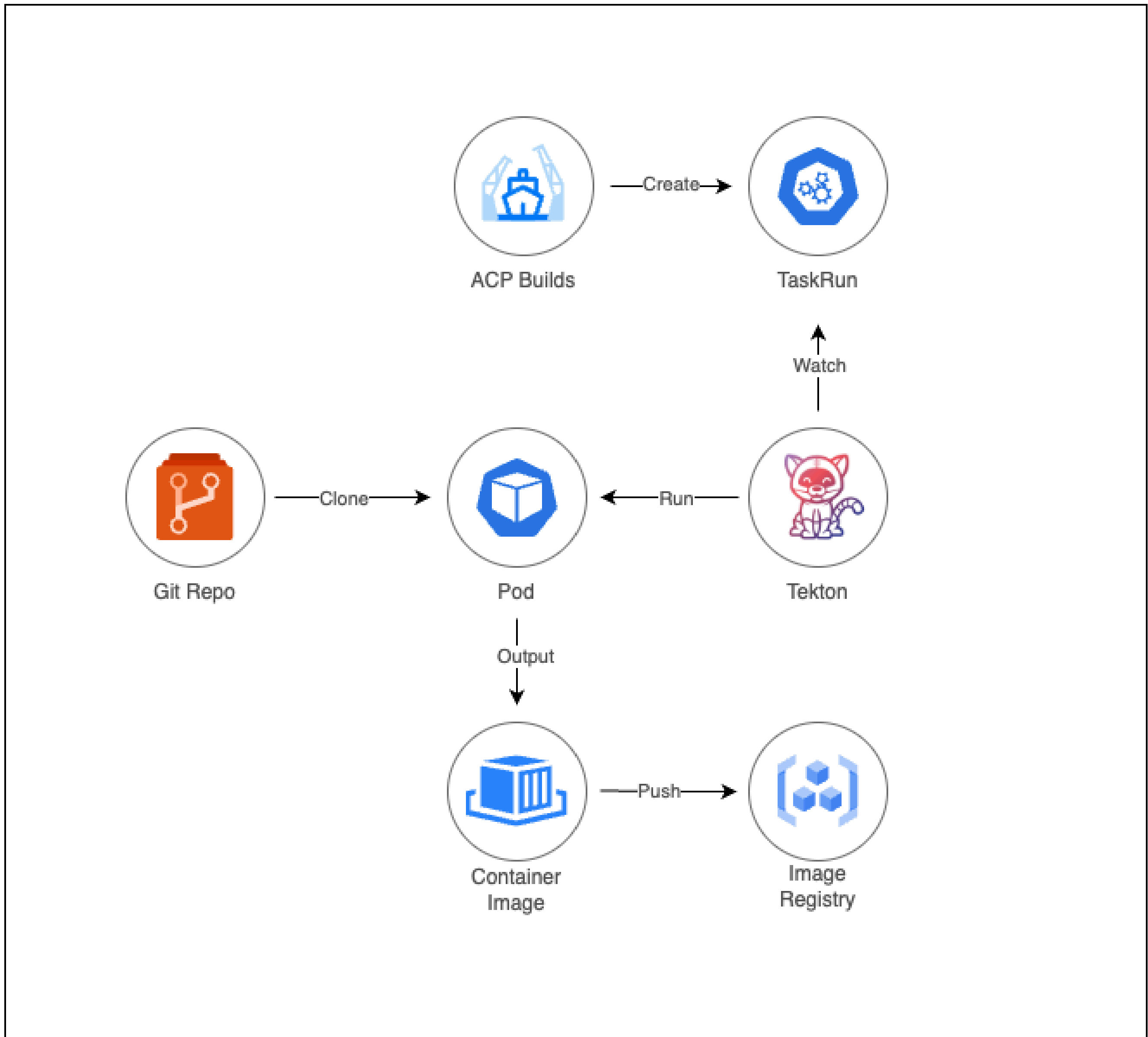
使用限制

> 当前版本仅支持 Java、Go、Python 和 Node.js 语言。

WARNING

前提条件：Alauda DevOps Pipelines operator 现已在集群 OperatorHub 中提供。

架构



Source to Image (S2I) 功能通过 **Alauda Container Platform Builds** operator 实现， 支持从 Git 仓库源代码自动构建容器镜像，并将镜像推送到指定的镜像仓库。 核心组件包括：

- **Alauda Container Platform Builds** operator：管理端到端的构建生命周期，并协调 Tekton pipelines。

- **Tekton** pipelines：通过 Kubernetes 原生的 `TaskRun` 资源执行 S2I 工作流。

发版日志

目录

Alauda Container Platform Builds Release Notes

支持的版本

v1.1 发版日志

v1.1.0

Alauda Container Platform Builds Release Notes

Alauda Container Platform Builds operator 的 release notes 描述了新功能和增强、已弃用功能以及已知问题。

> **INFO**
>
> **Alauda Container Platform Builds** operator 作为一个可安装组件提供，其发布周期与核心 灵雀云容器平台 不同。Alauda Container Platform Builds operator Lifecycle Policy 说明了发布兼容性。

支持的版本

版本	灵雀云容器平台 版本	Alauda DevOps Pipelines 版本
v1.1.0	**v4.1**	**v4.1**

v1.1 发版日志

v1.1.0

1. 安全漏洞修复。
2. 独立发布。

生命周期策略

目录

版本生命周期时间线

版本生命周期时间线

以下是 Alauda Container Platform Builds operator 已发布版本的生命周期计划：

版本	发布日期	生命周期结束
v1.1.0	2025-08-15	2027-08-15

安装

Installing Alauda Container Platform Builds

Prerequisites

Procedure

Installing Alauda Container Platform Builds

目录

Prerequisites
Procedure
Install the Alauda Container Platform Builds Operator
Install the Shipyard instance
Verification

Prerequisites

> Alauda Container Platform Builds 是 灵雀云容器平台 提供的一个容器工具，集成了构建（支持 Source to Image）和创建应用的功能。

1. 下载与您的平台匹配的最新版本 **Alauda Container Platform Builds** 安装包。如果 Kubernetes 集群中尚未安装 **Alauda DevOps Pipelines** operator，建议一并下载。
2. 使用 `violet` CLI 工具将 **Alauda Container Platform Builds** 和 **Alauda DevOps Pipelines** 安装包上传到目标集群。有关 `violet` 的详细使用说明，请参见 CLI。

Procedure

Install the Alauda Container Platform Builds Operator

1. 登录后，进入 **Administrator** 页面。
2. 点击 **Marketplace** > **OperatorHub**。
3. 找到 **Alauda Container Platform Builds** operator，点击 **Install**，进入 **Install** 页面。

配置参数：

Parameter	Recommended Configuration
Channel	`Alpha`：默认 Channel 设置为 **alpha**。
Version	请选择最新版本。
Installation Mode	`Cluster`：单个 Operator 共享整个集群所有命名空间，用于实例创建和管理，资源占用较低。
Namespace	`Recommended`：建议使用 **shipyard-operator** 命名空间；如果不存在会自动创建。
Upgrade Strategy	请选择 `Manual`。 • `Manual`：当 OperatorHub 有新版本时，**Upgrade** 操作不会自动执行。

4. 在 **Install** 页面选择默认配置，点击 **Install**，完成 **Alauda Container Platform Builds** Operator 的安装。

Install the Shipyard instance

1. 点击 **Marketplace** > **OperatorHub**。
2. 找到已安装的 **Alauda Container Platform Builds** operator，进入 **All Instances**。
3. 点击 **Create Instance** 按钮，在资源区域点击 **Shipyard** 卡片。
4. 在实例参数配置页面，除非有特殊需求，否则可使用默认配置。
5. 点击 **Create**。

Verification

- 实例创建成功后，等待约 20 分钟，进入 **Container Platform** > **Applications** > **Applications**，点击 **Create**。
- 应能看到 **Create from Code** 入口。此时，Alauda Container Platform Builds 已成功安装，您可以开始使用 Creating an application from Code 开启您的 S2I 之旅。

升级

升级 Alauda Container Platform Builds

前提条件

操作步骤

升级 Alauda Container Platform Builds

目录

前提条件

操作步骤

升级 Alauda Container Platform Builds Operator

前提条件

> Alauda Container Platform Builds 是 灵雀云容器平台 提供的一个容器工具，集成了构建（支持 Source to Image）和创建应用功能。

1. 下载与您的平台匹配的 **Alauda Container Platform Builds** 新版本包。
2. 使用 `violet` CLI 工具将 **Alauda Container Platform Builds** 和 **Alauda DevOps Pipelines** 包上传到目标集群。有关 `violet` 的详细使用说明，请参见 CLI。

操作步骤

升级 Alauda Container Platform Builds Operator

INFO

如果您从 v4.0 及更早版本升级，请先将 **Alauda DevOps Tekton v3** 迁移到 **Alauda DevOps Pipelines**。 详情请参见 migration guide。

1. 登录并进入 管理员 页面。
2. 点击 **Marketplace** > **OperatorHub**。
3. 在导航栏中选择安装了 operator 的集群。
4. 找到 **Alauda Container Platform Builds** operator 并打开其 详情 页面。
5. 点击 确认 开始升级，等待 operator 升级完成。

操作指南

Managing applications created from Code

主要功能

优势

前提条件

操作步骤

相关操作

Managing applications created from Code

目录

主要功能
优势
前提条件
操作步骤
相关操作
构建

主要功能

- 输入代码仓库 URL 触发 S2I 流程，将源代码转换为镜像并发布为应用。
- 当源代码更新时，通过可视化界面发起 **Rebuild** 操作，一键更新应用版本。

优势

- 简化从代码创建和升级应用的流程。
- 降低开发人员门槛，无需了解容器化细节。
- 提供可视化构建流程和运维管理，便于定位、分析和排查问题。

前提条件

- 已完成安装 Alauda Container Platform Builds。
- 需要访问镜像仓库；若无，请联系管理员进行安装 Alauda Container Platform Registry。

操作步骤

1. 在 **Container Platform** 中，导航至 **Application** > **Application**。
2. 点击 **Create**。
3. 选择 **Create from Code**。
4. 参考以下参数说明完成配置。

区域	参数	说明
代码仓库	类型	• 平台集成：选择已与平台集成且分配给当前项目的代码仓库；平台支持 GitLab、GitHub 和 Bitbucket。 • 输入：使用未与平台集成的代码仓库 URL。
	集成项目名称	管理员分配或关联给当前项目的集成工具项目名称。
	仓库地址	选择或输入存储源代码的代码仓库地址。
	版本标识	支持基于代码仓库中的分支、标签或提交创建应用。其中： • 当版本标识为分支时，仅支持使用所选分支下的最新提交创建应用。 • 当版本标识为标签或提交时，默认选择代码仓库中的最新标签或提交，也可根据需要选择其他版本。

	上下文目录	可选的源代码目录，作为构建的上下文目录。
	Secret	使用输入类型代码仓库时，可根据需要添加认证 Secret。
	构建器镜像	• 包含特定编程语言运行环境、依赖库和 S2I 脚本的镜像，主要用于将源代码转换为可运行的应用镜像。 • 支持的构建器镜像包括：Golang、Java、Node.js 和 Python。
构建	版本	选择与源代码兼容的运行环境版本，确保应用顺利运行。
	构建类型	目前仅支持通过 **Build** 方式构建应用镜像。该方式简化并自动化复杂的镜像构建流程，使开发人员专注于代码开发。整体流程如下： 1. 安装 Alauda Container Platform Builds 并创建 Shipyard 实例后，系统自动生成集群级资源，如 ClusterBuildStrategy，定义标准化构建流程，包括详细构建步骤和必要构建参数，从而支持 Source-to-Image (S2I) 构建。详情请参见：安装 Alauda Container Platform Builds 2. 根据上述策略及表单信息创建 Build 类型资源，指定构建策略、构建参数、源代码仓库、输出镜像仓库等相关信息。 3. 创建 BuildRun 类型资源以启动具体构建实例，协调整个构建流程。 4. BuildRun 创建完成后，系统自动生成对应的 TaskRun 资源实例。该 TaskRun 实例触发 Tekton pipeline 构建并创建 Pod 执行构建过程。Pod 负责实际构建工作，包括： 从代码仓库拉取源代码。 调用指定的构建器镜像。 执行构建流程。
	镜像 **URL**	构建完成后，指定应用的目标镜像仓库地址。

应用	-	根据需要填写应用配置，具体请参见从镜像创建应用文档中的参数说明。
网络	-	• 目标端口：容器内应用实际监听的端口。启用外部访问时，所有匹配流量将转发至该端口以提供外部服务。 • 其他参数：请参见创建 Ingress文档中的参数说明。
标签注解	-	根据需要填写相关标签和注解。

5. 填写完参数后，点击 **Create**。
6. 可在 **Details** 页面查看对应部署信息。

相关操作

构建

应用创建完成后，可在详情页查看对应信息。

参数	说明
Build	点击链接查看具体的构建（Build）及构建任务（BuildRun）资源信息和 YAML。
Start Build	构建失败或源代码变更时，可点击此按钮重新执行构建任务。

实用指南

Creating an application from Code

Prerequisites

Procedure

Creating an application from Code

> 利用 Alauda Container Platform Builds 安装的强大功能，实现从 Java 源代码到创建应用的整个过程，最终使应用能够在 Kubernetes 上以容器化方式高效运行。

目录

Prerequisites
Procedure

Prerequisites

在使用此功能之前，请确保：

- Installing Alauda Container Platform Builds
- 平台上有可访问的镜像仓库。如无，请联系管理员进行 Installing ACP Registry

Procedure

1. 在 **Container Platform** 中，点击 **Applications** > **Applications**。
2. 点击 **Create**。
3. 选择 **Create from Code**。
4. 根据以下参数完成配置：

参数	推荐配置
Code Repository	类型：`Input` 仓库 **URL**：`https://github.com/alauda/spring-boot-hello-world`
Build Method	`Build`
Image Repository	联系管理员。
Application	**Application**：`spring-boot-hello-world` 名称：`spring-boot-hello-world` 资源限制：使用默认值。
Network	目标端口：`8080`

5. 填写完参数后，点击 **Create**。
6. 可在 **Details** 页面查看对应应用状态。

节点隔离策略

节点隔离策略提供了一个项目级别的节点隔离方案，允许项目独占使用集群节点。

介绍

介绍

优势

适用场景

架构

架构

核心概念

Core Concepts

Node Isolation

操作指南

创建节点隔离策略

创建节点隔离策略

删除节点隔离策略

权限

权限

介绍

Node Isolation Strategy 提供了一个项目级别的节点隔离策略，允许项目独占使用集群节点。

目录

优势
适用场景

优势

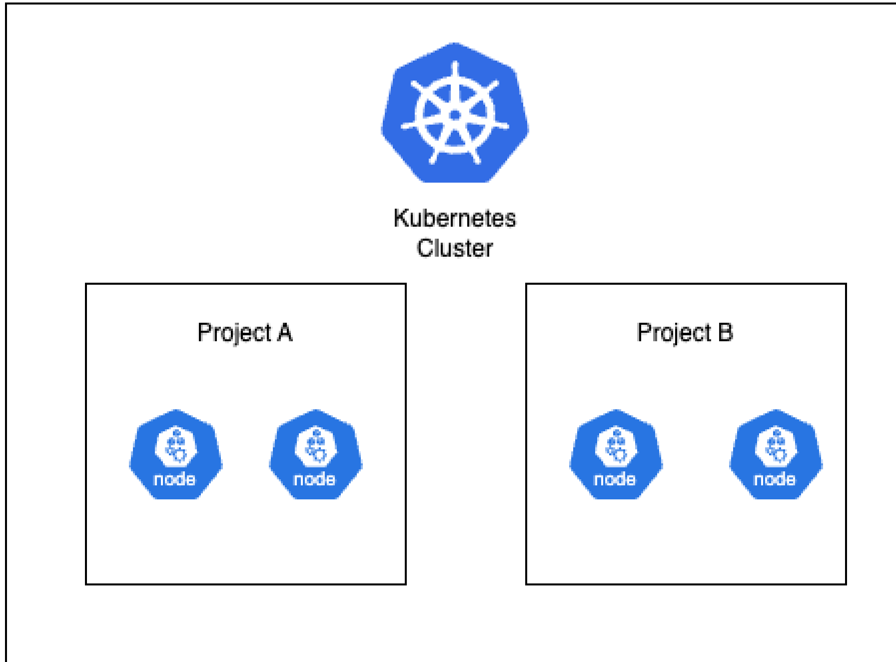
方便地以独占或共享的方式将节点分配给项目，防止项目之间的资源争用。

适用场景

Node Isolation Strategy 适用于需要加强项目间资源隔离，防止其他项目组件占用节点导致资源受限或无法满足性能要求的场景。

Alauda Container Platform

架构



节点隔离策略基于容器平台集群核心组件实现，通过在每个业务集群上分配节点，提供项目间节点隔离的能力。当在某个项目中创建容器时，容器会被强制调度到该项目分配的节点上。

核心概念

Core Concepts

Node Isolation

Core Concepts

目录

Node Isolation

Node Isolation

节点隔离是指在集群中隔离节点，防止不同项目的容器同时使用同一节点，从而避免资源争用和性能下降。

操作指南

创建节点隔离策略

创建节点隔离策略

删除节点隔离策略

创建节点隔离策略

为当前集群创建节点隔离策略，允许指定项目独占集群内分组资源的节点，从而限制该项目下 Pod 可运行的节点，实现项目间的物理资源隔离。

目录

创建节点隔离策略

删除节点隔离策略

创建节点隔离策略

1. 在左侧导航栏，点击 **Security** > **Node Isolation Strategy**。
2. 点击 **Create Node Isolation Strategy**。
3. 参考以下说明配置相关参数。

参数	说明
Project Exclusivity	是否开启策略中配置的项目隔离策略所包含节点的开关；点击切换开关，默认为开启。 开关开启时，策略中指定项目下的 Pod 才能运行在策略包含的节点上；关闭时，当前集群中除指定项目外的其他项目下的 Pod 也可以运行在策略包含的节点上。
Project	配置使用策略中节点的项目。 点击 **Project** 下拉选择框，勾选项目名称前的复选框以选择多个项目。

参数	说明
	注意： 一个项目只能设置一个节点隔离策略；若项目已被分配节点隔离策略，则不可选择； 支持在下拉选择框中输入关键词过滤并选择项目。
Node	策略中分配给项目使用的计算节点 IP 地址。 点击 **Node** 下拉选择框，勾选节点名称前的复选框以选择多个节点。 注意： 一个节点只能属于一个隔离策略；若节点已属于其他隔离策略，则不可选择； 支持在下拉选择框中输入关键词过滤并选择节点。

4. 点击 **Create**。
   注意：

- 策略创建后，项目中不符合当前策略的现有 Pod 在重建后会调度到当前策略包含的节点上；
- 当 **Project Exclusivity** 开启时，节点上当前存在的 Pod 不会被自动驱逐；如需驱逐，需手动调度。

删除节点隔离策略

注意：删除节点隔离策略后，项目将不再限制在特定节点上运行，节点也不再被项目独占。

1. 在左侧导航栏，点击 **Security** > **Node Isolation Strategy**。
2. 找到节点隔离策略，点击 ⋮ > **Delete**。

权限

功能	操作	平台管理员	平台审计人员	项目管理员	命名空间管理员	开发人员
节点隔离策略 `acp-nodegroups`	查看	✓	✓	✓	✓	✓
	创建	✓	✕	✕	✕	✕
	更新	✓	✕	✕	✕	✕
	删除	✓	✕	✕	✕	✕

常见问题

目录

为什么导入命名空间时不应存在多个 ResourceQuota？

为什么导入命名空间时不应存在多个 LimitRange？

为什么导入命名空间时不应存在多个 **ResourceQuota**？

导入命名空间时，如果该命名空间包含多个 ResourceQuota 资源，平台会从所有 ResourceQuota 中选择每个配额项的最小值并进行合并，最终创建一个名为 `default` 的单一 ResourceQuota。

示例：

待导入的命名空间 `to-import` 包含以下 `resourcequota` 资源：

```yaml
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: a
  namespace: to-import
spec:
  hard:
    requests.cpu: "1"
    requests.memory: "500Mi"
    limits.cpu: "3"
    limits.memory: "1Gi"
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: b
  namespace: to-import
spec:
  hard:
    requests.cpu: "2"
    requests.memory: "300Mi"
    limits.cpu: "2"
    limits.memory: "2Gi"
```

导入 `to-import` 命名空间后，该命名空间中将创建以下名为 `default` 的 ResourceQuota：

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: default
  namespace: to-import
spec:
  hard:
    requests.cpu: "1"
    requests.memory: "300Mi"
    limits.cpu: "2"
    limits.memory: "1Gi"
```

对于每个 ResourceQuota，资源配额取 `a` 和 `b` 之间的最小值。

当命名空间中存在多个 ResourceQuota 时，Kubernetes 会独立验证每个 ResourceQuota。因此，导入命名空间后，建议删除除 `default` 以外的所有 ResourceQuota，以避免多个 ResourceQuota 导致配额计算复杂化，从而容易产生错误。

为什么导入命名空间时不应存在多个 LimitRange？

导入命名空间时，如果该命名空间包含多个 LimitRange 资源，平台无法将它们合并为单个 LimitRange。由于 Kubernetes 在存在多个 LimitRange 时会独立验证每个 LimitRange，且 Kubernetes 选择哪个 LimitRange 的默认值行为不可预测。

如果命名空间仅包含单个 LimitRange，平台会创建一个名为 `default` 的 LimitRange，值来自该 LimitRange。

因此，导入命名空间前，命名空间中应仅存在单个 LimitRange。导入后，建议删除除名为 `default` 的 LimitRange 以外的其他 LimitRange，以避免多个 LimitRange 导致的不可预测行为。