开发人员

本节用于面向开发人员的 ACP 指南，包括原生应用构建、运维，以及用于原生应用管理工作流的汇总宿主页面。

目录

从汇总路径开始

概览

快速开始

构建应用

镜像

Alauda Container Platform Registry 镜像使用

Source to Image

节点隔离策略

常见问题

从汇总路径开始

对于大多数原生应用交付和后续运维，先从构建原生应用开始。此路径是用于原生应用创建、工作负载、命名空间、配置、运维和可观测性指导的长期宿主页。

在需要高级生命周期图时，请使用概览，并继续查看 构建原生应用 下的特定任务页面。

概览

概览

命名空间管理

原生应用管理工作流

原生应用生命周期管理

Kubernetes 工作负载管理

快速开始

通过镜像创建简单应用

简介

重要说明

前提条件

工作流概览

操作步骤

构建应用

构建应用架构

构建应用简介

核心组件

核心概念

命名空间

选择正确的入口

创建应用

应用的操作与维护

工作负载可用性

使用 Helm charts

1. 了解 Helm

2 通过 CLI 以 Application 方式部署 Helm Charts

配置

选择正确的入口视图

非敏感配置

敏感配置

运维提示

应用可观测

实用指南

镜像

镜像概述

理解容器和镜像

镜像

镜像 registry

镜像 repository

镜像 tag

镜像 ID

容器

实用指南

Alauda Container Platform Registry 镜像使用

访问 Registry

前提条件

所需的命名空间权限

从集群内部访问 Registry

使用 ac 管理镜像

前提条件

Registry 模式

命令组

验证外部 OCI 客户端

推送和拉取镜像

查询镜像 API 资源

发现并登录 Registry

管理 ImageStreams

查询镜像和 ImageStream

删除镜像引用

直接处理 OCI 镜像

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 提供统一界面，可通过 Web 控制台和 CLI（Command-Line Interface）创建、编辑、删除和管理云原生应用。应用可使用 RBAC 策略部署到多个命名空间中。

目录

命名空间管理
原生应用管理工作流
原生应用生命周期管理
原生应用创建模式
原生应用操作
原生应用可观测性
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 灵雀云容器平台 后，可获得与原生创建的命名空间完全一致的平台能力。

原生应用管理工作流

请使用 构建应用 下的聚合页面来完成持续的应用生命周期工作。

大多数迁移后的 Web 控制台工作流仍遵循相同的作用域模型：

- 原生应用监控面板和摘要视图可以汇总当前项目关联的集群中的数据。
- 具体的创建、更新、查看或故障排查操作仍在选定的集群和命名空间上下文中执行。
- 同一种资源类型可以同时出现在 原生应用 视图和 集群 视图中，但入口路径会改变任务的表述方式和操作作用域。

根据当前项目、已启用的集成以及您的权限，控制台可能会针对同一类任务显示不同的选项卡、操作、警告横幅或监控信号。

原生应用生命周期管理

灵雀云容器平台 支持端到端生命周期管理，包括：

原生应用创建模式

在 灵雀云容器平台 中，可以通过多种方式创建应用。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` 以及其他 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 Chart 或 Operator Backed）进行创建。
- 从 YAML 创建：通过导入 YAML 文件，一次性创建包含全部资源的自定义应用。
- 从代码创建：通过 Source to Image（S2I）构建镜像。

这些页面也是从新版控制台体验迁移而来的应用创建工作流的主要入口。

原生应用操作

- 更新应用：更新应用的镜像版本、环境变量和其他配置，或导入现有 Kubernetes 资源以进行集中管理。

- 导出应用：将应用导出为 YAML、Kustomize 或 Helm Chart 格式，然后导入到其他命名空间或集群中，以创建新的应用实例。
- 版本管理：支持自动或手动创建应用版本，并且在出现问题时可一键回滚到指定版本，以便快速恢复。
- 删除应用：删除应用时，会同时删除应用本身及其直接包含的所有 Kubernetes 资源。此外，此操作还会切断该应用可能与其他并非其定义直接组成部分的 Kubernetes 资源之间的关联。

原生应用可观测性

在持续运行管理方面，平台提供日志、事件、监控等能力。

- 日志：支持查看当前正在运行的 Pod 的实时日志，同时还提供之前容器重启时的日志。
- 事件：支持查看命名空间内所有资源的事件信息。
- 监控面板：为原生应用、工作负载和 Pod 提供监控面板。原生应用概览 监控面板以项目为作用域，而工作负载和 Pod 监控面板仍以命名空间为作用域。您还可以自定义监控面板以满足特定的操作需求。

Kubernetes 工作负载管理

支持的核心工作负载类型：

- Deployments：通过滚动更新管理无状态应用。
- StatefulSets：运行具有稳定网络 ID 的有状态应用。
- DaemonSets：部署节点级服务（例如日志收集器）。
- CronJobs：使用重试策略调度批处理任务。

`building_application` 下的工作负载页面同样是从新版控制台体验迁移而来的工作负载工作流的主要入口。

快速开始

通过镜像创建简单应用

简介

重要说明

前提条件

工作流概览

操作步骤

通过镜像创建简单应用

> 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

- 简介
  - 使用场景
  - 所需时间
- 重要说明
- 前提条件
- 工作流概览
- 操作步骤
  - 创建 Namespace
  - 配置镜像仓库
    - 方法 1：通过 Toolchain 集成 Registry
    - 方法 2：外部 Registry 服务
  - 通过 Deployment 创建应用
  - 通过 NodePort 暴露 Service
  - 验证应用可访问性

简介

使用场景

- 希望了解 Kubernetes 平台上基本应用创建工作流的新用户
- 展示平台核心功能的实践练习，包括：
  - Project/Namespace 编排
  - 创建 Deployment
  - Service 暴露模式
  - 验证应用可访问性

所需时间

预计完成时间：10-15 分钟

重要说明

- 本技术指南重点介绍必要参数；如需高级配置，请参阅完整文档
- 所需权限：
  - 创建 Project/Namespace
  - 集成镜像仓库
  - 部署工作负载

前提条件

- 基本了解 Kubernetes 架构和 灵雀云容器平台 平台概念
- 已按照平台建立操作步骤预配置项目

工作流概览

编号	操作	描述
1	创建 Namespace	建立资源隔离边界
2	配置镜像仓库	设置容器镜像来源
3	通过 Deployment 创建应用	创建 Deployment 工作负载
4	通过 NodePort 暴露 Service	配置 NodePort Service
5	验证应用可访问性	测试端点连通性

操作步骤

创建 **Namespace**

> Namespace 为资源分组和配额管理提供逻辑隔离。

前提条件

- 具有创建、更新和删除 Namespace 的权限（例如 Administrator 或 Project Administrator 角色）
- 已配置 kubectl 并可访问集群

创建过程

1. 登录，然后转到 **Project Management** > **Namespaces**
2. 选择 **Create Namespace**
3. 配置必要参数：

参数	描述
Cluster	从项目关联的集群中选择目标集群

参数	描述
Namespace	唯一标识符（自动添加项目名称前缀）

4. 使用默认资源约束完成创建

配置镜像仓库

灵雀云容器平台 支持多种镜像来源策略：

方法 1：通过 **Toolchain** 集成 **Registry**

1. 访问 **Administrator** > **Toolchain** > **Integration**
2. 发起新集成：

参数	要求
Name	唯一的集成标识符
API Endpoint	Registry 服务 URL（HTTP/HTTPS）
Secret	已存在或新创建的凭证

3. 将 Registry 分配给目标平台项目

方法 2：外部 **Registry** 服务

- 使用可公开访问的 Registry URL
- 示例：`nginx:latest`

Registry 访问和身份验证

要使集群成功拉取容器镜像，必须满足两个关键要求：

- 网络可访问性：集群节点必须能够通过出站（egress）网络访问容器 Registry 的端点。如果连接失败，可能会遇到 `ImagePullBackOff` 或其他网络相关错误。

- 身份验证：如果 Registry 要求身份验证，请创建 `ImagePullSecret`，并在 Pod 或 Deployment 中引用它，以便集群拉取镜像。有关分步操作说明，请参阅镜像拉取 Secret 文档。

通过 Deployment 创建应用

> Deployment 为 Pod ReplicaSet 提供声明式更新。

创建过程

1. 在 **Container Platform** 视图中：
   - 使用 Namespace 选择器选择目标隔离边界
2. 转到 **Workloads** > **Deployments**
3. 单击 **Create Deployment**
4. 指定镜像来源：
   - 选择已集成的 Registry，或
   - 输入外部镜像 URL（例如 `nginx:latest`）
5. 配置工作负载身份并启动

管理操作

- 监控 Replica 状态
- 查看事件和日志
- 检查 YAML manifest
- 分析资源指标和告警

通过 NodePort 暴露 Service

Service 为 Pod 组提供网络可访问性。

创建过程

1. 转到 **Networking** > **Services**

2. 单击 **Create Service** 并配置参数：

参数	值
Type	NodePort
Selector	目标 Deployment 名称
Port Mapping	Service 端口：Container 端口（例如 8080:80 ）

3. 确认创建。

关键事项

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

内部路由通过提供统一的 IP 地址或主机端口来访问 Service，从而支持工作负载的服务发现。

1. 单击 **Network** > **Service**。
2. 单击 **Create Service**。
3. 根据以下参数配置 **Details**，其他参数保留默认值。

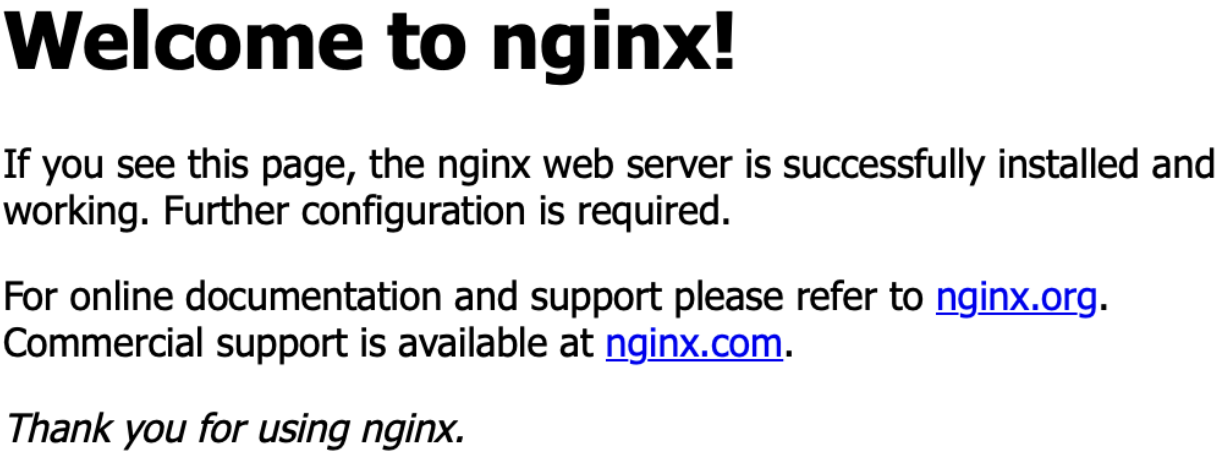
参数	描述
Name	输入 Service 的名称。
Type	`NodePort`
Workload Name	选择之前创建的 `Deployment`。
Port	**Service Port**：Service 在集群内暴露的端口号，即 Port，例如 `8080`。 **Container Port**：Service 端口映射的目标端口号（或名称），即 targetPort，例如 `80`。

4. 单击 **Create**。此时，Service 已成功创建。

验证应用可访问性

验证方法

1. 获取已暴露端点的组成部分：
   - **Node IP**：工作节点的公共地址
   - **NodePort**：分配的外部端口
2. 构造访问 URL：`http://<Node_IP>:<NodePort>`
3. 预期结果：Nginx 欢迎页面



构建应用

本节是 ACP 中面向应用的工作流的综合入口。

当你需要创建应用、管理工作负载、准备 namespace、处理 ConfigMap 或 Secret，或执行常规的应用操作和可观测性任务时，请优先使用此路径。

目录

与迁移后的 web console 文档的关系
构建应用架构
核心概念
命名空间
创建应用
应用的操作与维护
工作负载
使用 Helm charts
配置
应用可观测
实用指南

与迁移后的 **web console** 文档的关系

新版 console 体验中的许多工作流现在都可以直接映射到本节中的页面。

在大多数情况下：

- 对于应用创建和更新路径，使用 `create_applications`。
- 对于 Deployment、StatefulSet、DaemonSet、Pod、Job 和 CronJob，使用 `application_workloads`。
- 对于 namespace 准备和治理，使用 `namespace`。
- 对于 ConfigMap 和 Secret，使用 `configuration`。
- 对于状态、版本管理、日志、事件、监控以及相关的运维任务，使用 `operation_maintaining` 和 `application_observability`。

在这些任务类别中，请注意以下范围规则：

- 汇总型应用仪表板可以汇总当前项目所关联集群中的信息。
- 具体的资源操作仍然会在选定的集群和 namespace 上下文中执行。
- 当某个工作流既可以从 **Applications** 视图启动，也可以从 **Clusters** 视图启动时，请在执行操作的页面中记录其特定于范围的行为，而不是单独保留一套并行的操作步骤。

构建应用架构

构建应用架构

构建应用简介

核心组件

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 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

理解环境变量

Overview

资源单位说明

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

通过新版 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

导入命名空间

概述

使用场景

前提条件

操作步骤

使用新 Web 控制台导入命名空间

相关指南

资源配额

理解 Resource 配额

硬件加速器资源

Limit Range

理解 Limit Range

UID/GID 分配

启用 UID/GID 分配

验证 UID/GID 分配

超售比

理解命名空间资

更新命名空间

更新配额

更新容器 LimitRange

更新 Pod Security Admission

在新 web 控制台中从 Clusters 视图更新命名空间

删除/移除命

删除命名空间

移除命名空间

如何在移除和删

使用新的 Web

创建应用

从 Image 创建应用

前提条件

操作步骤 1 - 工作负载

操作步骤 2 - Services

操作步骤 3 - Ingress

应用管理操作

参考信息

通过 Chart 创建原生应用

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

Applications 视

Applications 视

Creating applications from Code

Creating applications from Operator Backed

Understanding Operator Backed Application

通过 Web 控制台创建 Operator Backed 应用

故障排查

Creating ap

Prerequisites

Procedure

Example

Reference

应用的操作与维护

Application Rollout

启动和停止原生应用

启动原生应用

停止原生应用

更新应用

导入资源

移除/批量移除资源

导出应用

导出 Helm Chart

导出 YAML 到本地

导出 YAML 到代码仓库（Alpha）

应用版本管理

创建版本快照

回滚到历史版本

为工作负载动态注入 PSA 安全上下文

前提条件

状态说明

原生应用

配置 VerticalPodAutoscaler (VPA)

概述

更新和删除 Chart 应用

重要说明

前提条件

状态分析说明

删除原生应用

使用 PodDisruptionBudgets

配置 HPA

理解 Horizontal

ntal

nH

Horiz

Horiz

调度规则说明

工作负载再平

Pod 驱逐规则和

理解 Deschedu

安装

处理资源耗尽

工作原理

PSA 强制执行配置

验证

Overview

配置驱逐策略

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

服务质量与内存

调度器与资源耗

示例场景

推荐实践

工作负载

部署

理解 Deployments

创建 Deployments

何时在 Applications 视图中使用 Deployme

提交前的 Deployment 检查点

管理 Deployments

使用 CLI 排障

守护进程集

理解 DaemonSets

创建 DaemonSets

管理 DaemonSets

有状态副本集

了解有状态副本

创建有状态副本

何时在 Applicat

有状态工作负载

定时任务

了解定时任务

创建定时任务

Jobs

理解 Jobs

YAML 文件示例

Pods

理解 Pod

执行概览

Containers

在不影响 Pod 的情况下调整 Pod 资源级别

前提条件

关于原地 Pod 调整大小

配置原地 Pod 调整大小

调整正在运行的 Pod

验证重启行为

检查调整大小状态

限制

其他资源

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

配置 ConfigMap

了解 ConfigMap

ConfigMap 限制

配置 Secrets

了解 Secrets

创建 Opaque 类型 Secret

ConfigMap 示例
使用 web console 创建 ConfigMap
使用新的 web console 创建 ConfigMap
使用 CLI 创建 ConfigMap
操作
使用 CLI 查看、编辑和删除
在 Pod 中使用 ConfigMap 的方式
ConfigMap 与 Secret 的对比

创建容器镜像仓库类型 Secret
创建 Basic Auth 类型 Secret
创建 SSH-Auth 类型 Secret
创建 TLS 类型 Secret
通过 web 控制台创建 Secret
如何在 Pod 中使用 Secret
后续操作
操作
使用新 web 控制台创建 Secret

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

将 ImagePullSecrets 添加到 Ser

创建 ImagePullSecret
将 ImagePullSecret 添加到 ServiceAccour
验证新 Pod 已设置 imagePullSecrets
在 OCI Connector 中使用 ImagePullSecre

Service 服务

概述
功能特性
配置和部署（管
用户指南（面向

构建应用架构

目录

构建应用简介
核心组件
Archon
Metis
Captain 控制器管理器
Icarus

构建应用简介

Alauda Container Platform 是一个用于开发和运行容器化应用的平台。它旨在支持应用及其依赖的数据中心从仅包含少量机器和应用，扩展到包含数千台机器并为数百万客户端提供服务的规模。

Alauda Container Platform 构建于 Kubernetes 之上，利用了支持大规模电信、流媒体视频、游戏、银行及其他关键应用的同类可靠技术。这一基础使您能够跨混合环境扩展容器化应用，从本地基础设施扩展到多云部署。

核心组件

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` 仅在 `global` 集群上运行，作为集群范围操作的中央管理接口。其 API 层支持在整个 Alauda Container Platform 中以声明方式配置应用、命名空间和基础设施资源。

Metis

在 `workload clusters` 中充当多用途控制器，提供关键的集群级操作：

- **Webhook** 管理：用于资源验证的准入 Webhook，包括资源比例和资源标签策略的强制执行。
- 状态同步：通过以下机制维护分布式组件之间的一致性：
  - `Helm chart application` 状态协调
  - `Project quota` 同步
  - `Application` 状态更新（写入 `Application.status`）

Captain 控制器管理器

充当 `Helm chart` 应用生命周期管理控制器，仅在 `global cluster` 上运行。其职责包括：

- **Chart** 安装：协调跨集群部署 `Helm chart`
- 版本管理：处理 `Helm chart` release 的无缝升级和回滚
- 卸载：完全移除 `Helm chart` 应用及其相关资源
- **Release** 跟踪：维护所有已部署 `Helm chart` release 的状态和历史记录

Icarus

为 `Container Platform` 提供集中式 Web 管理界面。作为表示层组件，`Icarus`：

- 提供全面的监控面板可视化功能，用于监控集群运行状况
- 支持基于 GUI 的应用部署和管理工作流
- 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
  - 提供特定于租户的视图隔离
- 仅在 `global cluster` 上运行，作为多集群操作的统一控制点

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 app
lication's components
                are still being deployed ("Pending") or all are deployed
already ("Succeeded"). When the
                application cannot be successfully assembled, the install
er 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": "Servic
e"},{"group":"apps","kind":"Deployment"}]
              items:
                description: 'The item of the GroupKinds, with a structur
e like \"{"group":"core","kind": "Service"}\"'
                type: object
              type: array
            descriptor:
              properties:
                description:
                  description: 'A short, human readable textual descripti
on 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:
    kind: ApplicationHistory
```

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

```
# ❌ 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

```
# ✅ 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/s
h

# 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

来源类型	描述	使用场景
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：

```
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

```
# ✅ 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

```
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

Dockerfile	Kubernetes	作用
CMD	`args`	提供默认参数

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

```
# 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

```
# ✅ 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 -T
ERM $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。

命名空间

命名空间部分涵盖应用工作负载的命名空间生命周期和治理操作。

当你需要创建命名空间、将现有命名空间导入到项目中、更新命名空间治理设置，或移除/删除命名空间时，请使用这些页面。

目录

选择正确的入口视图

主要任务

选择正确的入口视图

在新的 Web 控制台中，命名空间管理可以从不同的视图开始，而入口路径决定了操作范围。

- 当目标是在已与当前项目关联的集群中进行项目范围内的命名空间管理时，请使用 原生应用 视图。
- 当目标是针对具体集群资源直接进行集群侧命名空间治理时，请使用 集群 视图。

集群 视图不涵盖项目关联工作流，例如将命名空间导入项目或从项目中移除命名空间。

主要任务

- 创建命名空间
- 导入命名空间
- 更新命名空间
- 删除/移除命名空间

当你选择目标页面时，请将工作流指导放在用户执行操作的位置：

- 创建指导应放在创建页面中，并与配额、限制范围和策略审查一起说明
- 导入指导应放在导入页面中，因为它会改变项目治理预期
- 更新指导应放在更新页面中，并与配额、限制范围和安全操作放在一起
- 移除或删除指导应放在移除页面中，因为项目解除关联和资源删除是不同的操作

创建命名空间

了解命名空间

通过 Web 控制台创建命名空间

通过新版 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

导入命名空间

概述

使用场景

前提条件

操作步骤

使用新 Web 控制台导入命名空间

相关指南

资源配额

理解 Resource

配额

硬件加速器资源

Limit Range

UID/GID 分配

启用 UID/GID 分配

验证 UID/GID 分配

超售比

更新命名空间

更新配额

更新容器 LimitRange

更新 Pod Security Admission

在新 web 控制台中从 Clusters 视图更新命名空间

删除/移除命
删除命名空间
移除命名空间
如何在移除和删
使用新的 Web 控

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
通过新版 Web 控制台创建命名空间
从 Applications 视图创建
从 Clusters 视图创建
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

请参阅官方 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。资源名称在同一个命名空间内必须唯一，但不要求在不同命名空间之间唯一。基于命名空间的作用域仅适用于带命名空间的对象（例如 Deployments、Services 等），不适用于集群范围对象（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中，创建一个与项目可用资源配额相匹配的新命名空间。 新命名空间将在项目分配的资源配额范围内运行（例如 CPU、内存），并且命名空间中的所有资源都必须位于关联的集群中。

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

参数	描述
集群	选择与项目关联的集群来承载该命名空间。
命名空间	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。
   每当为命名空间中的容器指定计算资源或存储资源的资源限制（limits），或者每次添加新的 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该项目在集群中分配的配额。某一资源类型可设置的最大配额不得超过该项目剩余的可用配额。如果任何资源的可用配额达到 0，将无法创建命名空间。请联系平台管理员调整配额。
   - **GPU** 配额配置要求：
     - 仅当集群中已提供 GPU 资源时，才能配置 GPU 配额（vGPU 或 pGPU）。
     - 使用 vGPU 时，也可以设置内存配额。

     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 单位仅按整数计算（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：
  - 如果未为某一资源类型指定配额，则默认为无限制。
  - 这意味着该命名空间可以在不显式设置限制的情况下消耗该项目分配给此类资源的全部可用资源。

配额参数说明

类别	配额类型	数值和单位	描述
存储资源配额	全部	Gi	此命名空间中所有 Persistent Volume Claims（PVCs）的总请求存储容量不能超过此值。
	Storage Class		此命名空间中与所选 StorageClass 关联的所有 Persistent Volume Claims（PVCs）的总请求存储容量不能超过此值。 注意：请提前将 StorageClass 分配给该命名空间所属的项目。
扩展资源	从配置字典（ConfigMap）中获取；详情请参阅 扩展资源配额说明。	-	如果不存在对应的配置字典，则不会显示此类别。
其他配额	输入自定义配额；有关具体输入规则，请参阅 其他配额说明。	-	为避免资源重复，以下值不允许作为配额类型： • limits.cpu • limits.memory • requests.cpu

类别	配额类型	数值和单位	描述
			• requests.memory • pods • cpu • memory

6. （可选）配置 容器限制范围；有关更多详细信息，请参阅 Limit Range。
7. （可选）配置 **Pod Security Admission**；有关具体细节，请参阅 Pod Security Admission。
8. （可选）在 更多配置 区域中，为当前命名空间添加标签和注解。
   提示：您可以通过标签定义命名空间的属性，也可以通过注解为命名空间补充附加信息；二者都可用于筛选和排序命名空间。
9. 单击 创建。

通过新版 **Web** 控制台创建命名空间

此页面主要描述旧版控制台工作流，但在新版 Web 控制台中，命名空间创建可以从不同视图开始。

从 **Applications** 视图创建

当您为当前项目创建命名空间时，请使用 **Applications** 视图。

入口路径和操作顺序：

- 打开当前项目，然后转到 **Applications** > **Namespaces**。
- 单击 创建命名空间。
- 首先选择目标 集群，然后继续设置命名空间名称、配额、限制范围、Pod Security Admission、标签和注解。

在提交表单之前，请确认：

- 目标集群已与当前项目关联

- 由于命名空间是当前项目集群范围治理流程的一部分，因此必须先选择集群，再提交命名空间
- 命名空间命名规则符合当前项目的治理要求
- 配额、限制范围、安全策略、标签和注解符合预期的工作负载基线
- 命名空间随后会显示在当前项目中，并关联到预期的集群

从 **Clusters** 视图创建

当您要在某个具体集群中将命名空间作为集群资源创建时，请使用 **Clusters** 视图。

入口路径和操作顺序：

- 先打开目标集群。
- 转到该集群上下文中的 **Namespaces**。
- 单击 创建命名空间，然后继续设置命名空间名称、标签和注解。

通过 **ac CLI** 在项目中创建命名空间

要使用 `ac` 命令行工具在项目中创建命名空间，请运行以下命令：

```
# Create a namespace in project with specific clusters
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 securi
ty, 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

```
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 --defaul
t='cpu=100m,memory=100Mi' --default-request='cpu=50m,memory=50Mi' --max
='cpu=1000m,memory=1000Mi'
```

导入命名空间

目录

概述
使用场景
前提条件
操作步骤
使用新 Web 控制台导入命名空间
相关指南

概述

命名空间生命周期管理能力：

- 跨集群命名空间导入：将命名空间导入到某个项目后，可在平台所提供的所有 Kubernetes 集群之间集中管理这些命名空间。这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。

命名空间解除关联：

- 解除关联命名空间功能可让你将命名空间与当前项目取消绑定，并重置其关联关系，以便后续重新分配或清理。

- 将命名空间导入到某个项目后，它将获得与平台上原生创建的命名空间相同的能力。这包括继承的项目级策略（例如 Resource Quotas）、统一监控以及集中化治理控制。

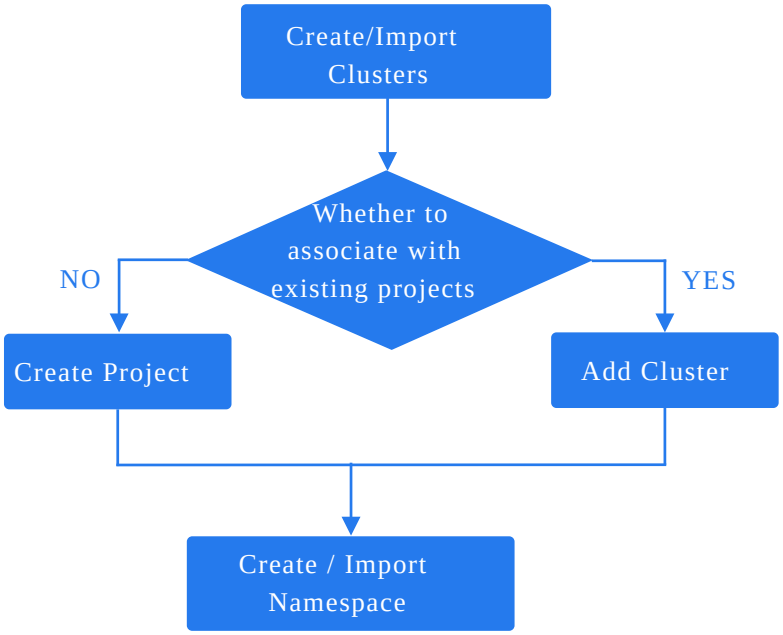
重要说明：

- 在任何给定时间，一个命名空间只能关联到一个项目。
- 如果某个命名空间已经关联到一个项目，则在先将其与原始项目解除关联之前，不能将其导入到另一个项目或重新分配给另一个项目。

使用场景

命名空间管理的常见使用场景包括：

- 当你将新的 **Kubernetes** 集群连接到平台后，可以使用 导入命名空间 功能将其现有的 **Kubernetes** 命名空间 关联到某个项目。只需选择目标项目和集群即可开始导入。此操作会赋予该 项目 对这些 命名空间 的治理能力，包括 **Resource Quotas**、监控和策略执行。



- 已从某个 项目 解除关联的 命名空间，可以通过 导入命名空间 功能无缝重新关联到另一个项目，以继续进行集中化治理。
- 当前未由任何 项目 管理的命名空间（例如通过集群级脚本创建的命名空间）必须使用 导入命名空间 功能关联到目标 项目，以启用平台级治理，包括可见性和集中管理。

前提条件

- 该命名空间当前未由平台中的任何现有项目管理。
- 命名空间只能导入到已经关联其目标 Kubernetes 集群的项目中。如果不存在这样的项目，则必须先创建一个与该集群关联的项目。

操作步骤

1. 在 **Project Management** 中，单击要导入命名空间的 ***Project name***。
2. 导航到 **Namespaces** > **Namespaces**。
3. 单击 **Create Namespace** 旁边的 **Dropdown** 按钮，然后选择 **Import Namespace**。
4. 有关参数配置详情，请参阅 Creating Namespaces 文档。
5. 单击 **Import**。

使用新 **Web** 控制台导入命名空间

新的 Web 控制台将命名空间导入保留在 **Applications** 视图中，因为导入是一个以项目为范围的治理操作。

当你使用该工作流时：

- 从目标项目开始，并打开 **Namespaces**。
- 仅对已经与当前项目关联的集群中的命名空间使用 **Import Namespace**。
- 在导入前检查命名空间状态，以确保关联后监控、配额治理和策略预期仍然合理。
- 导入后验证该命名空间是否出现在当前项目中，并保持与预期集群的关联。

相关指南

有关以项目为范围的命名空间治理，请继续阅读本节中的命名空间页面：

- Creating Namespaces

- Updating Namespaces
- Deleting/Removing Namespaces

如果你需要有关产品 UI 入口和范围的背景信息，请参阅 Web Console。

资源配额

请参考官方 Kubernetes 文档：Resource Quotas ↗

目录

理解 Resource Requests 与 Limits
配额
Resource Quotas
YAML file example
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

理解 Resource Requests 与 Limits

用于限制特定 namespace 可用的资源。namespace 中所有 Pods 的总资源使用量（不包括处于 `Terminating` 状态的对象）不得超过配额。

Resource Requests：定义容器所需的最小资源（例如 CPU、内存），用于指导 Kubernetes Scheduler 将 Pod 调度到具有足够容量的节点上。

Resource Limits：定义容器可消耗的最大资源，防止资源耗尽并确保集群稳定性。

配额

Resource Quotas

> 如果某个资源标记为 `Unlimited`，则不会强制执行显式配额，但其使用量不能超过集群的可用容量。

Resource Quotas 会跟踪 namespace 内累积的资源消耗（例如容器 limits、新建 Pods 或 PVCs）。

支持的配额类型

字段	描述
Resource Requests	namespace 中所有 Pods 的请求资源总量： • CPU • 内存
Resource Limits	namespace 中所有 Pods 的限制资源总量： • CPU • 内存
Number of Pods	namespace 中允许的最大 Pod 数量。

注意：

- namespace 配额来源于项目已分配的集群资源。如果任何资源的可用配额为 0，创建 namespace 将失败。请联系管理员。
- `Unlimited` 表示该 namespace 可以消耗项目中该资源类型剩余的集群资源。

YAML file example

```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：

- **All**：namespace 中 PVC 存储容量总和。
- **Storage Class**：特定 storage class 的 PVC 存储容量总和。

注意：请确保该 storage class 已预先分配给包含该 namespace 的项目。

硬件加速器资源配额

当目标集群中安装了 HAMi、NVIDIA GPU Device Plugin 或 NPU Operator 等加速器实现时，ACP 可通过 Kubernetes 扩展资源公开相应的加速器配额字段。

本页面负责 namespace 配额操作。关于加速器资源名称、实现映射以及产品文档入口，请参考 Accelerator Resource Quota。

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（/）：它必须以字母或数字开头并结尾，且可以包含字母、数字、连字符（-）、下划线（_）或句点（.），形成一个长度最多为 63 个字符的 qualified name。
- 如果自定义配额名称包含斜杠（/）：名称会分为两部分，即 prefix 和 name，格式为：prefix/name。prefix 必须是有效的 DNS subdomain，而 name 必须符合 qualified name 的规则。
- DNS Subdomain：
  - Label：必须以小写字母或数字开头并结尾，可以包含连字符（-），但不能仅由连字符组成，最大长度为 63 个字符。
  - Subdomain：在 label 规则基础上扩展，允许通过句点（.）连接多个 label 形成 subdomain，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 --defaul
t='cpu=100m,memory=100Mi' --default-request='cpu=50m,memory=50Mi' --max
='cpu=1000m,memory=1000Mi'
```

Pod 安全策略

ACP 支持 Kubernetes Pod Security Admission (PSA) 和 Kyverno Policy，以帮助强制实施集群中运行的 Pod 的安全标准。

目录

- Pod Security Admission
  - 安全模式
  - 安全标准
  - 配置
    - 命名空间标签
  - 豁免
- Kyverno Policy
  - 前提条件
  - 应用 Kyverno Policies
  - Web Console
  - CLI

Pod Security Admission

请参阅 Kubernetes 官方文档：Pod Security Admission ↗

Pod Security Admission (PSA) 是 Kubernetes 的准入控制器，可在命名空间级别强制执行安全策略，方法是根据预定义标准验证 Pod 规范。

安全模式

PSA 定义了三种模式，用于控制如何处理违反策略的情况：

模式	行为	使用场景
Enforce	拒绝创建或修改不符合要求的 Pod。	需要严格强制执行安全策略的生产环境。
Audit	允许创建 Pod，但会在审计日志中记录违规情况。	监控和分析安全事件，同时不阻止工作负载。
Warn	允许创建 Pod，但会针对违规情况向客户端返回警告。	用于调整策略的测试环境或过渡阶段。

关键说明：

- **Enforce** 仅对 Pod 生效（例如，拒绝 Pod，但允许 Deployment 等非 Pod 资源）。
- **Audit** 和 **Warn** 同时适用于 Pod 及其控制器（例如 Deployment）。

安全标准

PSA 定义了三种用于限制 Pod 权限的安全标准：

标准	描述	主要限制
Privileged	不受限制的访问权限。适用于可信工作负载（例如系统组件）。	不会验证 `securityContext` 字段。
Baseline	施加最低限度的限制，以防止已知的权限提升。	阻止 `hostNetwork`、`hostPID`、特权容器以及不受限制的 `hostPath` 卷。
Restricted	最严格的策略，强制实施安全最佳实践。	要求： - `runAsNonRoot: true` - `seccompProfile.type:`

标准	描述	主要限制
		`RuntimeDefault` - 删除 Linux capabilities。

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

```bash
# Step 1: Update Pod Admission labels
kubectl label namespace <namespace-name> \
  pod-security.kubernetes.io/enforce=baseline \
  pod-security.kubernetes.io/audit=restricted \
  --overwrite

# Step 2: Verify labels
kubectl get namespace <namespace-name> --show-labels
```

豁免

将特定用户、命名空间或运行时类豁免于 PSA 检查。

配置示例：

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

Kyverno Policy

ACP 提供了多个用于创建 Pod 安全 Kyverno policy 的示例。这些示例包括：

- **Restricted**：Restricted 拒绝访问所有主机功能，并要求 Pod 使用分配给命名空间的 UID 和 SELinux 上下文运行。
- **Restricted-v2**：Restricted-v2 拒绝访问所有主机功能，并要求 Pod 使用分配给命名空间的 UID 和 SELinux 上下文运行。这是限制最严格的 policy，默认用于已认证用户。除了旧版的“restricted”之外，它还要求删除所有 capabilities，并且不允许使用权限提升二进制文件。如果未设置 seccomp profile，它还会默认使用 runtime/default；否则必须指定此 seccomp profile。
- **Anyuid**：Anyuid 提供 Restricted policy 的所有功能，但允许用户使用任意 UID 和任意 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 policy 允许访问主机命名空间、文件系统和 PIDS。仅应将其用于受信任的 Pod。请谨慎授予。
- **Hostmount-anyuid**：Hostmount-anyuid 提供 Restricted policy 的所有功能，但允许 Pod 使用主机挂载和任意 UID。此 policy 主要由 persistent volume recycler 使用。警告：此 policy 允许以任意 UID（包括 UID 0）访问主机文件系统。请谨慎授予。
- **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。除了旧版的“hostnetwork” policy 之外，它还要求删除所有 capabilities，并且不允许使用权限提升二进制文件。如果未设置 seccomp profile，它还会默认使用 runtime/default；否则必须指定此 seccomp profile。
- **Node-exporter**：Node-exporter policy 用于 Prometheus node exporter。

- **Nonroot**：Nonroot 提供 Restricted policy 的所有功能，但允许用户使用任意非 root UID 运行。用户必须指定 UID，或者必须在容器运行时的 manifest 中指定 UID。
- **Nonroot-v2**：Nonroot-v2 提供 Restricted policy 的所有功能，但允许用户使用任意非 root UID 运行。用户必须指定 UID，或者必须在容器运行时的 manifest 中指定 UID。除了旧版的“nonroot” policy 之外，它还要求删除所有 capabilities，并且不允许使用权限提升二进制文件。如果未设置 seccomp profile，它还会默认使用 runtime/default；否则必须指定此 seccomp profile。
- **Privileged**：Privileged 允许访问所有特权功能和主机功能，并能够以任意用户、任意组、任意 fsGroup 以及任意 SELinux 上下文运行。警告：这是限制最宽松的 policy，仅应将其用于集群管理。请谨慎授予。

> **NOTICE**
>
> **Restricted** policy 不等同于 Kubernetes Pod Security Admission 的“restricted”标准。 如果要使用 kyverno **Restricted** policy，而不是 Kubernetes Pod Security Admission 的“restricted”标准，可能需要更改 Pod 安全配置。

前提条件

- 安装 Alauda Container Platform Policy Management，参阅文档。
- 在 ACP featuregate 设置中启用 feature gate `namespace-resource-manage`。

应用 Kyverno Policies

Web Console

1. 在 ACP 控制台中，导航到 **Container Platform**，选择要应用安全 policy 的命名空间。
2. 转到 **Advanced** > **Resources**。
3. 搜索资源类型名称为 **Policy** 且组为 **kyverno.io** 的 **Policy** 资源。
4. 选择版本“v2beta1”，点击 **Create** 以创建新的 Kyverno Policy。
5. 在 **Create Resource** 对话框中，选择 **Samples** 选项卡。
6. 选择所需的 Pod 安全 policy 示例（例如 `Restricted`），然后点击 **Try**。

7. 根据需要查看并修改 policy YAML，然后点击 **Update** 以应用 policy。

CLI

1. 登录要应用安全 policy 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

使用新版 Web 控制台更新超售比

从原生应用视图更新

理解命名空间资源超售比

灵雀云容器平台 允许你按命名空间设置资源超售比（CPU 和内存）。这用于管理该命名空间内容器限制值（最大使用量）与请求值（保证的最小值）之间的关系，从而优化资源利用率。

通过配置此比例，你可以确保用户定义的容器限制值和请求值保持在合理范围内，从而提升整个集群的资源效率。

关键概念

- 限制值：容器可使用的最大资源。超过限制值可能导致节流（CPU）或终止（内存）。

- 请求值：容器所需的保证最小资源。Kubernetes 会根据这些请求来调度容器。
- 超售比：限制值 / 请求值。此设置定义了命名空间内该比值的可接受范围，在资源保障和防止过度消耗之间取得平衡。

核心能力

- 通过设置合适的超售比来管理资源限制值与请求值之间的平衡，从而提升命名空间内的资源密度和应用稳定性。

示例

> 假设命名空间超售比设置为 2，当创建一个原生应用并指定 CPU 限制值为 4c 时，对应的 CPU 请求值计算如下：

CPU 请求 = CPU 限制值 / 超售比。因此，CPU 请求变为 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 # Absence of this field indicates inheritance of the cluster ove
rcommit ratio; 0 means no limitation.
  memory: 4 # Absence of this field indicates inheritance of the cluster
overcommit ratio; 0 means no limitation.
status:
  clusterCPU: 2 # Cluster Overcommit Ratio
  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，以管理资源限制值与请求值之间的比例。这可确保容器的资源分配保持在定义的边界内，从而提高集群资源利用率。

注意事项

如果集群使用节点虚拟化（例如虚拟节点），请先在集群/命名空间级别禁用超售，然后再为虚拟机进行配置。

操作步骤

1. 进入 项目管理，然后导航到 **Namespaces** > **Namespace** 列表。
2. 单击目标 *命名空间名称*。
3. 单击 操作 > 更新超售比。
4. 选择合适的超售比配置方式，为该命名空间设置 CPU 或内存超售比。

参数	描述
从集群继承	• 命名空间继承集群的超售比。 • 示例：如果集群 CPU/内存比为 4，则命名空间默认值为 4。 • 容器请求值 = 限制值 / 集群比值。 • 如果未设置限制值，则使用命名空间的默认容器配额。
自定义	• 设置命名空间专用的比值（整数 > 1）。 • 示例：集群比值 = 4，命名空间比值 = 2 → 请求值 = 限制值 / 2。 • 留空可为该命名空间禁用超售。

5. 单击 更新。

注意：更改仅适用于新创建的 Pod。现有 Pod 会保留其原始请求值，直到被重建。

使用新版 **Web** 控制台更新超售比

此设置属于命名空间治理，因此入口取决于你是在项目上下文中操作，还是在具体的集群上下文中操作。

从原生应用视图更新

当命名空间已与当前项目关联，并且你希望在该项目范围内调整请求与限制的治理策略时，请使用 原生应用视图。

在更新比值之前，请确认以下内容：

- 该命名空间属于当前项目
- 新的 CPU 和内存比值仍然适用于后续将在该命名空间中创建或重建的工作负载
- 该更改会与当前项目中的配额、默认限制值以及调度预期一起进行评估

管理命名空间成员

目录

导入成员
限制和约束
前提条件
操作步骤
添加成员
操作步骤
移除成员
操作步骤
使用新的 Web 控制台管理命名空间成员

导入成员

平台支持将成员批量导入到命名空间中，并分配诸如 Namespace Administrator 或 Developer 等角色，以授予相应权限。

限制和约束

- 成员只能从该命名空间所属项目的 项目成员 中导入到命名空间。
- 平台不支持导入系统默认创建的管理员用户或当前活动用户。

前提条件

要将用户导入为命名空间成员，必须先将其添加到该命名空间所属的项目中。

操作步骤

1. 在 项目管理 中，单击待导入成员所在的 *项目名称*。
2. 导航到 **Namespaces** > **Namespaces**。
3. 单击待导入成员所在的 *命名空间名称*。
4. 在 命名空间成员 选项卡中，单击 导入成员。
5. 按照以下步骤，将列表中的全部或部分用户导入到命名空间中。

> **TIP**
>
> 你可以使用对话框右上角的下拉框选择一个用户组，并在用户名搜索框中输入用户名进行模糊搜索。

- 将列表中的所有用户作为命名空间成员导入，并批量为用户分配角色。
  1. 单击对话框底部 设置角色 项右侧的下拉菜单，并选择要分配的角色名称。
  2. 单击 全部导入。
- 将列表中的一个或多个用户作为命名空间成员导入。
  1. 单击用户名/显示名称前面的复选框，选择一个或多个用户。
  2. 单击对话框底部 设置角色 项右侧的下拉菜单，并选择要分配给所选用户的角色名称。
  3. 单击 导入。

添加成员

当平台添加了 OICD 类型的 IDP 后，可以将 OIDC 用户添加为命名空间成员。

你可以将满足输入要求的有效 OIDC 账号添加为命名空间角色，并为该用户分配相应的命名空间角色。

注意：添加成员时，系统不会验证账号的有效性。请确保你添加的账号是有效的；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 IDP 为平台配置的 OIDC 身份认证服务中的有效账号，包括已成功登录平台的账号以及尚未登录平台的账号。

前提条件

平台已添加 **OICD** 类型的 IDP。

操作步骤

1. 在 项目管理 中，单击成员所在的 *项目名称*。
2. 导航到 **Namespaces** > **Namespaces**。
3. 单击待添加成员所在的 *命名空间名称*。
4. 在 命名空间成员 选项卡中，单击 添加成员。
5. 在 用户名 输入框中，输入平台支持的现有第三方平台账号的用户名。
   注意：请确认输入的用户名对应第三方平台上的现有账号；否则，该账号将无法成功登录此平台。
6. 在 角色 下拉菜单中，选择要为该用户配置的角色名称。
7. 单击 添加。 添加成功后，你可以在命名空间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，你也可以查看到该用户。在该用户成功登录或同步到此平台之前，其来源显示为 `-`，并且可以删除；当该账号成功登录或同步到平台后，平台会记录该账号的来源信息，并在用户列表中显示。

移除成员

移除指定的命名空间成员并删除其关联角色，以撤销其命名空间权限。

操作步骤

1. 在 项目管理 中，单击待移除成员所在的 *项目名称*。
2. 导航到 **Namespaces** > **Namespaces**。
3. 单击待移除成员所在的 *命名空间名称*。

4. 在 命名空间成员 选项卡中，单击待移除成员记录右侧的⋮> 移除。
5. 单击 移除。

使用新的 **Web** 控制台管理命名空间成员

本页主要描述的是旧版控制台的操作流程，但在新的 Web 控制台中，命名空间成员管理仍然是一个项目作用域内的操作。

当你需要管理已属于当前项目的命名空间成员访问权限时，请使用 **Applications** 视图。

当你从该视图管理命名空间成员时，请确认以下内容：

- 该命名空间已经关联到当前项目
- 成员来源仍然符合项目级所有权模型，因为导入来源于项目成员
- 角色变更仅影响命名空间访问权限，不会替代项目级成员关系治理
- 最终生成的命名空间成员列表仍与该命名空间中工作负载的预期运维归属一致

更新命名空间

目录

更新配额

使用 web 控制台更新资源配额

使用新 web 控制台更新资源配额

使用 CLI 更新资源配额

更新容器 LimitRange

使用 web 控制台更新 LimitRange

使用新 web 控制台更新 LimitRange

使用 CLI 更新 LimitRange

更新 Pod Security Admission

使用 web 控制台更新 Pod Security Admission

使用新 web 控制台更新 Pod Security Admission

使用 CLI 更新 Pod Security Admission

在新 web 控制台中从 Clusters 视图更新命名空间

从 Clusters 视图更新

更新配额

资源配额

使用 web 控制台更新资源配额

1. 在 **Project Management** 视图中，导航到左侧边栏的 **Namespaces** > **Namespace** 列表。
2. 单击目标 ***namespace name***。
3. 单击 **Actions** > **Update Quota**。
4. 调整资源配额（CPU、Memory、Pods 等），然后单击 **Update**。

使用新 web 控制台更新资源配额

当你在当前项目范围内调整命名空间治理时，请使用 **Applications** 视图。

入口路径和操作顺序：

- 打开当前项目，然后转到 **Applications** > **Namespaces**。
- 单击目标命名空间。
- 使用 **Actions** 选择 **Update Quota**。

当你从此视图更新命名空间配额时，请确认：

- 该变更属于已与当前项目关联的命名空间
- 操作顺序应为先选择命名空间，再执行具体的治理操作，因为在更新期间集群和命名空间标识保持固定
- 在更新配额值时，集群和命名空间标识保持不变
- 该变更仍符合项目预期的资源治理模型

使用 CLI 更新资源配额

> 资源配额 YAML 文件示例

```
# Step 1: Edit the namespace quota
kubectl edit resourcequota <quota-name> -n <namespace-name>

# Step 2: Verify changes
kubectl get resourcequota <quota-name> -n <namespace-name> -o yaml
```

当你更新命名空间配额时，请检查剩余的项目或集群容量是否仍支持后续的工作负载创建、调度和扩缩容。更严格的配额通常会先影响下一次部署或扩容尝试，而不是已经运行中的 Pod。

更新容器 **LimitRange**

> Limit Range

使用 **web** 控制台更新 **LimitRange**

1. 在 **Project Management** 视图中，导航到左侧边栏的 **Namespaces** > **Namespace** 列表。
2. 单击目标 ***namespace name***。
3. 单击 **Actions** > **Update Container LimitRange**。
4. 调整容器限制范围（`defaultRequest`、`default`、`max`），然后单击 **Update**。

使用新 **web** 控制台更新 **LimitRange**

当你在当前项目范围内调整命名空间治理时，请使用 **Applications** 视图。

入口路径和操作顺序：

- 打开当前项目，然后转到 **Applications** > **Namespaces**。
- 单击目标命名空间。
- 使用 **Actions** 选择 **Update Container LimitRange**。

当你从此视图更新命名空间限制范围时，请确认：

- 该变更属于已与当前项目关联的命名空间
- 操作顺序应为先选择命名空间，再执行具体的治理操作，因为在更新期间集群和命名空间标识保持固定
- 在更新限制范围值时，集群和命名空间标识保持不变
- 该变更仍符合项目预期的资源治理模型

使用 **CLI** 更新 **LimitRange**

> Limit Range YAML 文件示例

```
# Step 1: Edit the LimitRange
kubectl edit limitrange <limitrange-name> -n <namespace-name>

# Step 2: Verify changes
kubectl get limitrange <limitrange-name> -n <namespace-name> -o yaml
```

容器限制范围的变更通常对更新后创建或重新创建的工作负载影响最大。请检查新的默认请求、默认限制和最大值将如何影响后续的工作负载表单和准入行为。

更新 Pod Security Admission

> Pod Security Admission

使用 web 控制台更新 Pod Security Admission

1. 在 **Project Management** 视图中，导航到左侧边栏的 **Namespaces** > **Namespace** 列表。
2. 单击目标 ***namespace name***。
3. 单击 **Actions** > **Update Pod Security Admission**。
4. 调整安全标准（`enforce`、`audit`、`warn`），然后单击 **Update**。

使用新 web 控制台更新 Pod Security Admission

当你在当前项目范围内调整命名空间治理时，请使用 **Applications** 视图。

入口路径和操作顺序：

- 打开当前项目，然后转到 **Applications** > **Namespaces**。
- 单击目标命名空间。
- 使用 **Actions** 选择 **Update Pod Security Admission**。

当你从此视图更新命名空间策略时，请确认：

- 该变更属于已与当前项目关联的命名空间

- 操作顺序应为先选择命名空间，再执行具体的治理操作，因为在更新期间集群和命名空间标识保持固定
- 在更新策略值时，集群和命名空间标识保持不变
- 该变更仍符合项目预期的资源和安全治理模型

使用 CLI 更新 Pod Security Admission

> 更新 Pod Security Admission CLI 命令

更严格的命名空间级安全设置可能会阻止依赖提升权限、主机访问或不兼容安全上下文的 Pod。请在收紧策略前检查预期的工作负载需求。

在新 web 控制台中从 Clusters 视图更新命名空间

此页面主要描述的是旧版控制台工作流，但在新 web 控制台中，命名空间更新可以从不同视图开始。

从 Clusters 视图更新

当你需要从具体集群调整命名空间治理时，请使用 **Clusters** 视图。

入口路径和操作顺序：

- 先打开目标集群。
- 转到该集群上下文中的 **Namespaces**，并打开目标命名空间。
- 使用 **Actions** 选择 **Update**, **Delete**

你可以在详情页的 **Resource Quota** 选项卡和 **LimitRange** 选项卡中管理命名空间下的相应资源。

删除/移除命名空间

你可以永久删除一个命名空间，也可以将其从当前项目中移除。

目录

删除命名空间
移除命名空间
如何在移除和删除之间进行选择
使用新的 Web 控制台移除或删除命名空间
从 Applications 视图中移除
从 Clusters 视图中删除

删除命名空间

删除命名空间：永久删除一个命名空间及其内的所有资源（例如 Pods、Services、ConfigMaps）。此操作无法撤销，并会释放已分配的资源配额。

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

移除命名空间

移除命名空间：将一个命名空间从当前项目中移除，而不删除其资源。该命名空间仍保留在集群中，并且可以通过 导入命名空间 导入到其他项目中。

> **NOTE**
>
> - 此功能仅适用于 灵雀云容器平台。
> - Kubernetes 原生不支持将命名空间从项目中“移除”。

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

如何在移除和删除之间进行选择

当命名空间资源应保留在集群中，但不应再属于当前项目时，请使用 移除命名空间。

当命名空间资源本身已不再需要，并且其中的工作负载和命名空间级资源可以被移除时，请使用 删除命名空间。

在删除命名空间之前，请确认该命名空间中的工作负载、ConfigMaps、Secrets、存储使用情况以及任何剩余的排障需求都已不再需要。

使用新的 **Web** 控制台移除或删除命名空间

本页主要描述旧版控制台的工作流，但在新的 Web 控制台中，命名空间的移除或删除可以从不同视图开始。

从 **Applications** 视图中移除

当你需要将命名空间从当前项目中移除，但不一定要删除命名空间资源本身时，请使用 **Applications** 视图。

入口路径和操作顺序：

- 打开当前项目，然后转到 **Applications** > **Namespaces**。

- 打开目标命名空间，或使用其 **Actions** 菜单。
- 当目标是结束项目关联，同时保留集群中的命名空间资源时，选择 **Remove Namespace**。
- 仅当必须删除命名空间资源本身，并且环境在当前视图中提供该操作时，才选择 **Delete Namespace**。

从 **Clusters** 视图中删除

当你需要从具体集群中删除命名空间资源时，请使用 **Clusters** 视图。

入口路径和操作顺序：

- 先打开目标集群。
- 在该集群上下文中转到 **Namespaces**，并选择目标命名空间。
- 从可用的操作入口中选择 **Delete Namespace**，然后确认删除。

创建应用

从 Image 创建应用

前提条件
操作步骤 1 - 工作负载
操作步骤 2 - Services
操作步骤 3 - Ingress
应用管理操作
参考信息

通过 Chart 创建原生应用

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
Applications 视
Applications 视

Creating applications from Code

Creating applications from Operator Backed

Understanding Operator Backed Application
通过 Web 控制台创建 Operator Backed 应用
故障排查

Creating ap

Prerequisites
Procedure
Example
Reference

从 Image 创建应用

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 - Services
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
协议相关参数

前提条件

获取镜像地址。镜像来源可以是平台管理员通过 toolchain 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到你的项目，你可以使用其中的镜像。如果未找到所需的镜像仓库，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从该仓库拉取镜像。
- 通过 OCI Connector 选择镜像时，还需要为 Connector 代理选择镜像拉取 Secret，或者确认工作负载的 ServiceAccount 已经引用了该 Secret。请参阅在 OCI Connector 中使用 ImagePullSecret。

操作步骤 1 - 工作负载

1. 在左侧导航栏中，进入 **Container Platform** > **Applications** > **Applications**。
2. 单击 **Create**。
3. 选择 **Create from Image** 作为创建方式。
4. 选择或输入一个镜像，然后单击 **Confirm**。

> **INFO**
>
> 注意：使用 web console 中集成的镜像仓库镜像时，可以通过 **Already Integrated** 进行筛选。**Integration Project Name**，例如，镜像（registry-projectname），其中包含了该 web console 中的项目名称 projectname 以及镜像仓库中的项目名称 containers。
>
> 使用私有 registry 中的镜像时，需要配置对应的 image pull secret。更多信息，请参阅向 ServiceAccount 添加 ImagePullSecrets。
>
> 使用 OCI Connector 时，可直接从 Pod 模板中，或通过工作负载的 ServiceAccount 使用 Connector 代理镜像拉取 Secret。该 Secret 必须用于代理身份验证，而不是原始 registry。

6. 请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** 部分中，为工作负载配置声明式参数。

参数	说明
Model	根据需要选择工作负载：

参数	说明
	• **Deployment**：详细的参数说明，请参阅创建 Deployment。 • **DaemonSet**：详细的参数说明，请参阅创建 DaemonSet。 • **StatefulSet**：详细的参数说明，请参阅创建 StatefulSet。
Replicas	定义 Deployment 中期望的 Pod 副本数量（默认值：`1`）。可根据工作负载需求进行调整。
More > **Update Strategy**	配置用于零停机部署的 `rollingUpdate` 策略： **Max surge**（`maxSurge`）： • 更新期间可超出期望副本数的 Pod 最大数量。 • 支持绝对值（例如 `2`）或百分比（例如 `20%`）。 • 百分比计算公式：`ceil(current_replicas × percentage)`。 • 示例：基于 10 个副本计算时，4.1 → `5`。 **Max unavailable**（`maxUnavailable`）： • 更新期间可临时不可用的 Pod 最大数量。 • 百分比值不能超过 `100%`。 • 百分比计算公式：`floor(current_replicas × percentage)`。 • 示例：基于 10 个副本计算时，4.9 → `4`。 说明： 1. 默认值：如果未显式设置，则 `maxSurge=1`、`maxUnavailable=1`。 2. 非运行中的 **Pod**（例如处于 `Pending` / `CrashLoopBackOff` 状态）会被视为不可用。 3. 同时约束： • `maxSurge` 和 `maxUnavailable` 不能同时为 `0` 或 `0%`。 • 如果两个参数按百分比计算后都为 `0`，Kubernetes 会强制将 `maxUnavailable=1` 以确保更新能够继续进行。 示例： 对于一个有 10 个副本的 Deployment： • `maxSurge=2` → 更新期间的 Pod 总数：`10 + 2 = 12`。

参数	说明
	• `maxUnavailable=3` → 最少可用 Pod 数：`10 - 3 = 7`。 • 这可在保证可用性的同时，允许受控发布。

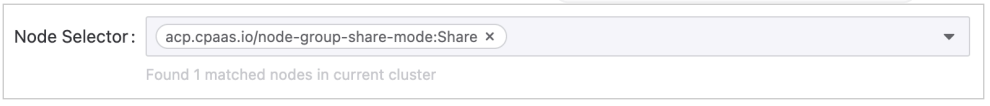
工作负载 2 - 配置 Pod

注意：在混合架构集群中部署单架构镜像时，请确保已正确配置 Node Affinity 规则，以便进行 Pod 调度。

1. 在 **Pod** 部分，配置容器运行时参数和生命周期管理：

参数	说明
Volumes	将持久卷挂载到容器。支持的卷类型包括 `PVC`、`ConfigMap`、`Secret`、`emptyDir`、`hostPath` 等。实现细节请参阅存储卷挂载说明。
Image Credential	对于私有 registry 中的镜像，请使用包含该 registry 凭证的 Secret。对于 OCI Connector 镜像，请选择 Connector 代理镜像拉取 Secret，或者使用已引用该 Secret 的 ServiceAccount。来自平台集成 registry 的镜像会继续继承关联的凭证。请参阅在 OCI Connector 中使用 ImagePullSecret。
More > **Close Grace Period**	Pod 在收到终止信号后，允许完成优雅关闭的持续时间（默认：`30s`）。 - 在此期间，Pod 会完成进行中的请求并释放资源。 - 设置为 `0` 会强制立即删除（SIGKILL），这可能导致请求中断。

2. **Node Affinity** 规则

参数	说明
More > **Node Selector**	将 Pod 约束到具有特定标签的节点上（例如 `kubernetes.io/os: linux`）。 
More > **Affinity**	基于现有 Pod 定义更细粒度的调度规则。

参数	说明
	Pod Affinity 类型： • **Inter-Pod Affinity**：将新的 Pod 调度到承载特定 Pod 的节点上（同一拓扑域）。 • **Inter-Pod Anti-affinity**：防止新的 Pod 与特定 Pod 共同调度到同一位置。 强制模式： • **RequiredDuringSchedulingIgnoredDuringExecution**：仅当规则满足时才调度 Pod。 • **PreferredDuringSchedulingIgnoredDuringExecution**：优先选择满足规则的节点，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`：使用标签查询筛选目标 Pod。

3. 网络配置

- Kube-OVN

参数	说明
Bandwidth Limits	为 Pod 网络流量实施 QoS： • **Egress rate limit**：最大出站流量速率（例如 `10Mbps`）。 • **Ingress rate limit**：最大入站流量速率。
Subnet	从预定义的子网池中分配 IP。如果未指定，则使用命名空间的默认子网。

参数	说明
Static IP Address	为 Pod 绑定持久 IP 地址： • 多个跨 Deployment 的 Pod 可以申请同一个 IP，但同一时刻只能有一个 Pod 使用它。 • 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，并严格保证唯一性： • 每个 IP 在集群中只能绑定一个 **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
Resource Requests & Limits	• **Requests**：容器运行所需的最小 CPU/内存。 • **Limits**：容器运行期间允许使用的最大 CPU/内存。单位定义请参阅资源单位. **Namespace overcommit ratio**： • **Without overcommit ratio**： 如果命名空间存在资源配额：容器 requests/limits 继承命名空间默认值（可修改）。 没有命名空间配额：无默认值；使用自定义 Request。 • **With overcommit ratio**： Requests 自动计算为 `Limits / Overcommit ratio`（不可修

参数	说明
	改）。 约束： • Request ≤ Limit ≤ 命名空间配额上限。 • 更改 overcommit ratio 需要重建 pod 才会生效。 • overcommit ratio 会禁用手动配置 request。 • 没有命名空间配额 → 没有容器资源约束。
Extended Resources	配置集群中可用的扩展资源（例如 vGPU、pGPU）。
Volume Mount	持久存储配置。请参阅存储卷挂载说明。 操作： • 已有 pod 卷：单击 **Add** • 没有 pod 卷：单击 **Add & Mount** 参数： • `mountPath`：容器文件系统路径（例如 `/data`） • `subPath`：卷内的相对文件/目录路径。 对于 `ConfigMap` / `Secret`：选择特定 key • `readOnly`：以只读方式挂载（默认：读写） 请参阅 Kubernetes Volumes ↗。
Port	暴露容器端口。 示例：暴露名为 `redis` 的 TCP 端口 `6379`。 字段： • `protocol`：TCP/UDP • `Port`：暴露端口（例如 `6379`） • `name`：符合 DNS 规范的标识符（例如 `redis`）

参数	说明
Startup Commands & Arguments	覆盖默认的 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"` 请参阅定义命令↗。
More > **Environment Variables**	• 静态值：直接使用键值对 • 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/配置文件中的设置。
More > **Referenced ConfigMap**	将整个 ConfigMap/Secret 作为环境变量注入。支持的 Secret 类型：`Opaque`、`kubernetes.io/basic-auth`。
More > **Health Checks**	• **Liveness Probe**：检测容器健康状态（失败时重启） • **Readiness Probe**：检测服务可用性（失败时从 endpoints 中移除） 请参阅健康检查参数。

参数	说明
More > **Log File**	配置日志路径： - 默认：采集 `stdout` - 文件模式：例如 `/var/log/*.log` 要求： • 存储驱动 `overlay2`：默认支持 • `devicemapper`：需要手动将 EmptyDir 挂载到日志目录 • Windows 节点：确保父目录已挂载（例如 `c:/a/b/c/*.log` 需要挂载 `c:/a`）
More > **Exclude Log File**	排除特定日志不进行采集（例如 `/var/log/aaa.log`）。
More > **Execute before Stopping**	在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. 单击右上角的 **Add Container** 或 **Add Init Container**。
   请参阅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. 允许删除的条件：
      - Pod 中有 >1 个应用容器且有 ≥1 个 init container。
      - 单应用容器的 pod 不允许删除。
3. 单击 **Create**。

操作步骤 2 - Services

参数	说明
Service	Kubernetes **Service**，即使工作负载分布在多个后端中，也能通过单个对外端点暴露集群中运行的应用。关于具体参数说明，请参阅创建 Services。 注意：在应用下创建的内部路由默认名称前缀为计算组件的名称。如果计算组件类型（部署模式）是 StatefulSet，建议不要修改内部路由的默认名称（即工作负载名称）；否则可能会导致该工作负载的可访问性问题。

操作步骤 **3 - Ingress**

参数	说明
Ingress	Kubernetes **Ingress**，通过支持协议感知的配置机制使你的 HTTP（或 HTTPS）网络服务可用，该机制能够理解 URI、hostname、path 等 web 概念。Ingress 概念允许你根据通过 Kubernetes API 定义的规则，将流量映射到不同的后端。关于详细参数说明，请参阅创建 Ingresses。 注意：在应用下创建 **Ingress** 时使用的 **Service** 必须是当前应用下创建的资源。不过，请确保该 **Service** 已关联到应用下的工作负载；否则，工作负载的服务发现和访问将失败。

7. 单击 **Create**。

应用管理操作

如需修改应用配置，请使用以下任一方式：

1. 单击应用列表右侧的垂直省略号（⋮）。
2. 在应用详情页右上角选择 **Actions**。

操作	说明
Update	• **Update**：仅使用已定义的 update strategy 修改目标工作负载（以 Deployment 策略为例）。保留现有副本数和发布配置。

操作	说明

	• **Force Update**：使用各组件的更新策略触发整个应用的完整发布。
	1. 适用场景：
	  • 需要立即在集群范围传播的批量配置变更（例如作为环境变量引用的 ConfigMap/Secret 更新）。
	  • 为安全需要而进行的协同组件重启。
	2. 警告：
	  • 在批量重启期间可能导致服务短暂降级。
	  • 如果未验证业务连续性，不建议在生产环境中使用。
	• 网络影响：
	  • 删除 Ingress 规则：如果满足以下条件，外部访问仍可通过 `LB_IP:NodePort` 使用：
	（1）LoadBalancer Service 使用默认端口。
	（2）保留的路由规则引用了应用组件。
	如需完全终止外部访问，必须删除 Service。
	  • 删除 Service：应用组件的网络连接将不可恢复地丢失。即使 API 对象仍然存在，关联的 Ingress 规则也会失效。

Delete	• 级联删除： 1. 删除所有子资源，包括 Deployments、Services 和 Ingress 规则。 2. Persistent Volume Claims（PVCs）将遵循 StorageClass 中定义的保留策略。 • 删除前检查清单： 1. 确认关联 Services 上没有活跃流量。 2. 确认有状态组件的数据备份已完成。 3. 使用 `kubectl describe ownerReferences` 检查依赖资源关系。

更新组件镜像或 OCI Connector 时，还应检查该组件 Pod 模板中的 image pull Secret 或 ServiceAccount 以及 token 的有效性。更新后，请验证 Connector 注解、实际重写后的代理镜像、Pod 发布状态以及 Events。

参考信息

存储卷挂载说明

类型	用途
Persistent Volume Claim	绑定现有 PVC 以请求持久存储。 注意：仅可选择已绑定的 PVC（即已关联 PV）。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 key 命名的文件 • 子路径选择：挂载特定 key（例如 `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 key 命名的文件 • 子路径选择：挂载特定 key（例如 `tls.crt`）
Ephemeral Volumes	由集群提供的临时卷，具有以下特性： • 动态供给 • 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。请参阅Ephemeral Volumes
Empty Directory	用于同一 pod 内容器之间共享的临时存储： - pod 启动时在节点上创建 - 随 pod 删除而删除 使用场景：容器间文件共享、临时数据存储。请参阅EmptyDir
Host Path	挂载主机机器目录（必须以 `/` 开头，例如 `/volumepath`）。

健康检查参数

通用参数

参数	说明
Initial Delay	开始探测前的宽限时间（秒）。默认：`300`。
Period	探测间隔（1-120s）。默认：`60`。
Timeout	探测超时时间（1-300s）。默认：`30`。
Success Threshold	判定健康所需的最小连续成功次数。默认：`0`。
Failure Threshold	触发操作所需的最大连续失败次数： - `0`：禁用基于失败的操作 - 默认：`5` 次失败 → 重启容器。

协议相关参数

参数	适用协议	说明
Protocol	HTTP/HTTPS	健康检查协议
Port	HTTP/HTTPS/TCP	用于探测的目标容器端口。
Path	HTTP/HTTPS	端点路径（例如 `/healthz`）。
HTTP Headers	HTTP/HTTPS	自定义 header（添加键值对）。
Command	EXEC	容器可执行的检查命令（例如 `sh -c "curl -I localhost:8080 | grep OK"`）。 注意：请转义特殊字符并测试命令是否可用。

通过 Chart 创建原生应用

基于 Helm Chart 代表一种原生应用部署模式。Helm Chart 是一组用于定义 Kubernetes 资源的文件，旨在打包应用并支持版本控制能力的应用分发。这使得环境切换更加顺畅，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

当集群同时包含 Linux 和 Windows 节点时，必须配置显式的节点选择，以避免调度冲突。示例：

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

前提条件

如果模板来自原生应用并引用了相关资源（例如 secret 字典），请确保在应用部署之前，当前 namespace 中已存在待引用的资源。

操作步骤

1. 在左侧边栏中，进入 **Container Platform** > **Applications** > **Applications**。
2. 单击 **Create**。
3. 选择 **Create from Catalog** 作为创建方式。
4. 选择一个 Chart 并配置参数，例如 `resources.requests`、`resources.limits` 以及其他与 chart 密切相关的参数。
5. 单击 **Create**。

Web 控制台将跳转到 **Application** > [**Native Applications**] 详情页。该过程需要一些时间，请耐心等待。如果操作失败，请按照界面提示完成操作。

状态分析参考

单击 ***Application Name***，即可在详情信息中查看 Chart 的详细状态分析。

类型	原因

Initialized	表示 Chart 模板的下载状态。
	• **True**：表示已成功下载 Chart 模板。
	• **False**：表示 Chart 模板下载失败；可在 message 列中查看具体失败原因。
	  • `ChartLoadFailed`：Chart 模板下载失败。
	  • `InitializeFailed`：在 Chart 下载之前，初始化过程发生异常。

Validated	表示 Chart 模板的用户权限、依赖项及其他校验状态。 • **True**：表示所有校验均已通过。

类型	原因

	• **False**：表示存在未通过的校验；可在 message 列中查看具体失败原因。
	  • `DependenciesCheckFailed`：Chart 依赖检查失败。
	  • `PermissionCheckFailed`：当前用户没有权限对某些资源执行操作。
	  • `ConsistentNamespaceCheckFailed`：通过原生应用中的模板部署应用时，Chart 包含需要跨 namespace 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；原因列显示 `ChartSyncFailed`，可在 message 列中查看具体失败原因。

WARNING

- 如果模板引用了跨 namespace 资源，请联系管理员协助创建。之后，您可以在 web 控制台上正常 更新和删除 Chart Applications。
- 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，则可以通过 YAML 创建应用。此方式可为应用配置及其引用资源提供更灵活的控制。

目录

注意事项

前提条件

操作步骤

Applications 视图与 Clusters 视图之间的创建入口差异

Applications 视图的其他指导

理解 YAML 验证和一致性

提交前的检查点

注意事项

当集群中同时存在 Linux 和 Windows 节点时，请考虑配置节点选择，使工作负载仅调度到兼容的节点上。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. **Container Platform**，然后导航到 **Application** > **Applications**。
2. 单击 **Create**。
3. 选择 **Create from YAML**。
4. 完成配置后，单击 **Create**。
5. 在详情页面查看对应的 **Deployment**。

Applications 视图与 **Clusters** 视图之间的创建入口差异

新的 Web 控制台可以从不同视图中提供基于 YAML 的创建入口。请根据您是在项目范围内操作，还是在具体集群中操作，选择相应入口。

- **Applications** 视图：从当前项目开始。打开 **Applications** > **Applications**，单击 **Create from YAML**，先选择目标 **Cluster**，然后检查或确认 YAML 内容中的目标 **Namespace**，最后提交资源。
- **Clusters** 视图：从具体集群开始。先打开目标集群，进入目标命名空间，然后导航到 **Applications** 或相关资源列表，以在该固定集群上下文中打开 YAML 创建工作流。

在继续之前，请确认以下差异：

- 在 **Applications** 视图中，集群选择是项目范围内应用工作流的一部分。
- 在 **Clusters** 视图中，由于您是从具体集群内部开始，因此集群已经固定。
- 在 **Applications** 视图中，主要顺序是先选择集群，再确认 YAML 中声明的命名空间。
- 在 **Clusters** 视图中，主要顺序是先进入具体集群，再在目标命名空间上下文中操作。

Applications 视图的其他指导

如果您的环境在 **Applications** 视图中提供 YAML 创建功能，请使用相同的 YAML 创建路径，并查看以下仅适用于 Applications 视图的行为。

1. 在 **Applications** 视图中打开应用页面。
2. 单击 **Create from YAML**。
3. 选择目标 **Cluster**。
4. 检查 YAML 元数据中声明的命名空间。
5. 粘贴或编辑 YAML 内容。
6. 查看可见的元数据和验证结果。
7. 单击 **Create**。

理解 **YAML** 验证和一致性

当您在 **Applications** 视图中通过 YAML 创建应用时：

- 所选集群决定工作流所使用的项目上下文。
- 工作流不会强制覆盖 YAML 中声明的命名空间。
- 命名空间验证会依据 YAML 中声明的命名空间以及命名空间资源的标准 Kubernetes 规则进行。
- YAML 验证会在提交前确认内容可以被解析并被工作流接受。
- 如果声明的命名空间不存在，或者您没有使用该命名空间的权限，提交将返回明确的错误。
- 验证成功并不保证运行时依赖项已经满足。

如果您的 YAML 包含多个资源，请确保在资源属于命名空间范围时，每个资源都显式声明了预期的命名空间。

提交前的检查点

在单击 **Create** 之前，请确认：

- YAML 中声明的命名空间就是您打算创建资源的命名空间。
- 声明的命名空间已存在，并且当前身份有权限在该命名空间中创建资源。
- 镜像源可从所选集群访问。
- 引用的 ConfigMaps、Secrets、Service、PVC 以及其他依赖项要么已经存在，要么在提交的 YAML 中保持一致地包含。
- 所有提交对象中的 labels、selectors、ports 和环境设置彼此保持内部一致。
- YAML 不会无意间与声明命名空间中具有相同标识的现有对象发生冲突。
- 任何调度约束仍然与目标集群匹配，尤其是在混合架构环境中。

当您一次性提交多个相关资源时，不仅要逐个检查每个对象，还要检查工作负载、Service、配置对象和存储引用之间的关系。

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"
```

```
            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

本节介绍通常在应用创建并部署后开始的持续管理工作。

典型任务包括：

- 在决定是进行故障排查、更新还是回滚之前，先查看应用状态
- 更新应用配置并管理与应用关联的资源
- 创建版本快照、比较历史修订，并在需要时回滚
- 导出当前版本或历史版本，以便在其他环境中复用

如果你主要需要日志、事件、指标或告警规则，请继续阅读 Application Observability。

目录

工作负载可用性

Application Rollout

状态说明

配置 HPA

启动和停止原生应用

配置 VerticalPodAutoscaler (VPA)

配置 CronHPA

更新应用

导出应用

更新和删除 Chart 应用

工作负载再平衡（Descheduler）

应用版本管理

删除原生应用

使用 PodDisruptionBudgets

为工作负载动态注入 PSA 安全上下文

处理资源耗尽错误

健康检查

工作负载可用性

使用这些页面来管理工作负载在维护期间的高可用性和约束条件：

- PodDisruptionBudgets

Application Rollout

安装 Alauda Container Platform

前提条件

安装 Alauda Container Platform Progressi

应用蓝绿发布

蓝绿发布的优势

使用 Alauda Container Platform Progressi

前提条件

操作步骤

应用灰度发布

灰度发布的优势

使用 Alauda Co

前提条件

操作步骤

状态说明

状态说明

原生应用
部署

配置 HPA

配置 HPA

理解 Horizontal Pod Autoscaler
前提条件
创建 Horizontal Pod Autoscaler
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)

概述
了解 VerticalPodAutoscaler
先决条件
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

更新应用

更新应用

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

工作负载再平衡（**Descheduler**）

工作负载再平衡（**Descheduler**）

Pod 驱逐规则和资格

理解 Descheduler 策略

在 ACP 中安装 Descheduler

配置调度策略

验证安装和驱逐

运维指导

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

使用 PodDisruptionBudgets

使用 PodDisruptionBudgets

了解 PodDisruptionBudgets

创建 PodDisruptionBudgets

何时使用 PodDisruptionBudget

为工作负载动态注入 PSA 安全上下文

为工作负载动态注入 PSA 安全上下文

前提条件

工作原理

PSA 强制执行配置

验证

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 Progressi

应用蓝绿发布

蓝绿发布的优势

使用 Alauda Container Platform Progressi

前提条件

操作步骤

应用灰度发布

灰度发布的优势

使用 Alauda Co

前提条件

操作步骤

安装 Alauda Container Platform Progressive Delivery

目录

前提条件

安装 Alauda Container Platform Progressive Delivery

操作步骤

前提条件

1. 下载与平台架构对应的 **Alauda Container Platform Progressive Delivery** 集群插件安装包。
2. 使用“上架软件包”机制上传安装包。
3. 使用集群插件机制将安装包安装到集群。

> **INFO**
>
> 上架软件包： 管理员 > **Marketplace** > 上架软件包页面。 点击右侧的帮助文档，获取如何将集群插件发布到集群的操作说明。有关详细信息，请参阅 CLI。

安装 **Alauda Container Platform Progressive Delivery**

操作步骤

2. 点击 **Marketplace** > 集群插件，进入集群插件列表页面。
3. 找到 **Alauda Container Platform Progressive Delivery** 集群插件，点击安装，进入安装 **Alauda Container Platform Progressive Delivery** 插件页面。
4. 直接点击安装，即可完成 **Alauda Container Platform Progressive Delivery** 集群插件的安装。

由 Alauda 修改，2026-10-21。

Argo® 是 LF Projects, LLC 的注册商标。

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

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环节。然而，将更新发布到生产环境可能存在风险，因为即使是小问题也可能导致严重的停机和收入损失。蓝绿部署是一种能够降低此类风险的部署策略，可确保应用的新版本实现零停机部署。

蓝绿部署是一种部署策略，在这种策略中会设置两个完全相同的环境，即“蓝”环境和“绿”环境。蓝环境是生产环境，当前正在运行应用的线上版本；绿环境是非生产环境，用于部署应用的新版本。

当应用的新版本准备就绪后，会将其部署到绿环境。新版本完成部署和测试后，流量会切换到绿环境，使其成为新的生产环境。随后，蓝环境将成为非生产环境，用于部署应用的后续版本。

目录

- 蓝绿发布的优势
- 使用 Alauda Container Platform Progressive Delivery 进行蓝绿发布
- 前提条件
- 操作步骤
  - 创建 Deployment
  - 创建蓝色 Service
  - 验证蓝色 Deployment
  - 验证到蓝色环境的流量路由
  - 创建 Rollout
  - 验证 Rollout

准备绿色 Deployment

将 Rollout 提升为绿色环境

蓝绿发布的优势

- 零停机：蓝绿部署可在零停机的情况下部署应用的新版本，因为流量可以无缝地从蓝环境切换到绿环境。
- 轻松回滚：如果应用的新版本存在问题，可以轻松回滚到上一版本，因为蓝环境仍然可用。
- 降低风险：采用蓝绿部署可以显著降低部署应用新版本的风险。这是因为在流量从蓝环境切换之前，可以先在绿环境中部署并测试新版本。这样可以进行全面测试，减少生产环境出现问题的可能性。
- 提高可靠性：采用蓝绿部署可以提高应用的可靠性。这是因为蓝环境始终可用，即使绿环境出现问题，也可以快速发现并解决，而不会影响用户。
- 灵活性：蓝绿部署为部署过程提供了灵活性。可以并行部署应用的多个版本，从而便于测试和试验。

使用 **Alauda Container Platform Progressive Delivery** 进行蓝绿发布

Argo Rollouts 是 Alauda Container Platform Progressive Delivery 使用的上游 Kubernetes 控制器。它为 Kubernetes 提供蓝绿、金丝雀、金丝雀分析、实验和渐进式交付等高级部署功能。

Argo Rollouts 可以选择与 ingress 控制器和服务网格集成，利用其流量整形能力，在更新期间逐步将流量切换到新版本。此外，Rollouts 可以查询和解析来自各种提供商的指标，以验证关键 KPI，并在更新期间驱动自动提升或回滚。

借助 Alauda Container Platform Progressive Delivery，您可以在 Alauda Container Platform (ACP) 集群上自动执行蓝绿发布。典型流程包括：

1. 定义 Rollout 资源，以管理不同的应用版本。
2. 配置 Kubernetes 服务，以在蓝色（当前）环境和绿色（新）环境之间路由流量。

3. 将新版本部署到绿色环境。
4. 验证并测试新版本。
5. 通过切换流量，将绿色环境提升为生产环境。

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部署。

> 关键概念：
>
> - **Rollout**：Kubernetes 中用于替代标准 Deployment 资源的自定义资源定义 (CRD)，支持蓝绿、金丝雀部署等高级部署控制。

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. 用于在其中创建命名空间的项目。
6. 集群中用于部署应用的命名空间。

操作步骤

1 创建 Deployment

首先定义应用的“蓝色”版本。这是用户将访问的当前版本。创建 Kubernetes deployment，并设置适当的副本数、容器镜像版本（例如 `hello:1.23.1`）以及 `app=web` 等正确标签。

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`：所需 Pod 副本数。
- `spec.selector.matchLabels`：定义 Deployment 如何查找要管理的 Pod。
- `template.metadata.labels`：应用于 Pod 的标签，供 Service 选择 Pod。
- `spec.containers`：每个 Pod 中要运行的容器。
- `containers.name`：容器名称。
- `containers.image`：要运行的容器镜像。
- `containers.ports.containerPort`：容器公开的端口。

使用 `kubectl` 应用配置：

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

这将建立生产环境。

或者，也可以使用 Helm Chart 创建 deployments 和 services。

2 创建蓝色 Service

创建一个 Kubernetes `Service`，以公开蓝色 deployment。此 Service 将根据匹配的标签将流量转发到蓝色 Pod。最初，Service selector 将目标指向带有 `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 Service 的 Kubernetes API 版本。
- `kind`：指定此资源为 Service。
- `metadata.name`：Service 的名称。
- `spec.selector`：根据标签标识要将流量路由到的 Pod。
- `ports.protocol`：使用的协议 (TCP)。
- `ports.port`：Service 公开的端口。
- `ports.targetPort`：流量要转发到的容器端口。

使用以下命令应用：

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

这样即可访问蓝色 deployment。

3 验证蓝色 Deployment

通过列出 Pod，确认 `blue` deployment 正常运行：

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

检查所有预期副本（2 个）是否处于 `Running` 状态。这可确保应用已准备好接收流量。

4 验证到蓝色环境的流量路由

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。这些是接收流量的端点。

5 创建 Rollout

接下来，使用 `BlueGreen` 策略创建 Argo Rollouts 的 `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。Pod 被此选择器选中的现有 ReplicaSet 将受到此 Rollout 的影响。它必须与 Pod 模板的标签匹配。
- `workloadRef`：指定要应用于 Rollout 的工作负载引用和缩容策略。
  - `scaleDown`：指定工作负载（Deployment）在迁移到 Rollout 后是否缩容。可选值包括：
    - "never"：Deployment 不缩容。
    - "onsuccess"：Rollout 变为 healthy 后，Deployment 缩容。
    - "progressively"：随着 Rollout 扩容，Deployment 逐步缩容。如果 Rollout 失败，Deployment 将重新扩容。
- `strategy`：Rollout 策略，支持 `BlueGreen` 和 `Canary` 策略。
  - `blueGreen`：`BlueGreen` Rollout 策略定义。
    - `activeService`：指定在提升时使用新模板哈希值更新的 Service。对于 blueGreen 更新策略，此字段为必需项。

- `autoPromotionEnabled`：autoPromotionEnabled 禁用新堆栈的自动提升，方法是在提升前立即暂停 Rollout。如果省略，则默认行为是在 ReplicaSet 完全就绪/可用后立即提升新堆栈。可以使用以下命令恢复 Rollout：`kubectl argo rollouts promote ROLLOUT`

使用以下命令应用：

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

这将为使用 `BlueGreen` 策略的 deployment 设置 Rollout。

6 验证 Rollout

创建 `Rollout` 后，Argo Rollouts 将创建一个使用与 deployment 相同模板的新 ReplicaSet。当新 ReplicaSet 的 Pod 处于 healthy 状态时，deployment 将缩容为 0。

使用以下命令确保 Pod 正常运行：

```
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
```

`web` Service 将把流量转发到 Rollout 创建的 Pod。使用以下命令：

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

7 准备绿色 Deployment

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

```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，但以下字段除外：
  - `containers.image`：更新为新的镜像版本。

使用以下命令应用：

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

这将设置用于测试的新应用版本。

Rollout 将创建一个新的 ReplicaSet 来管理绿色 Pod，而流量仍然转发到蓝色 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
```

当前共有 4 个 Pod 正在运行，分别对应蓝色版本和绿色版本。当前活动 Service 是蓝色版本，Rollout 过程已暂停。

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

8 将 **Rollout** 提升为绿色环境

绿色版本准备就绪后，提升 Rollout，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
```

如果活动的 `Images` 已更新为 `hello:1.23.2`，并且蓝色 ReplicaSet 已缩容为 0，则表示 Rollout 已完成。

由 Alauda 修改，2026-10-21。

Argo® 是 LF Projects, LLC 的注册商标。

Alauda 是独立供应商。本产品不隶属于 LF Projects, LLC，也未获得其认可或赞助。所有商标均归其各自所有者所有，仅用于标识目的。

应用灰度发布

灰度发布是一种渐进式发布策略，可将新版本应用逐步引入一小部分用户或流量中。这种增量式发布方式使团队能够监控系统行为、收集指标，并在全面部署前确保系统稳定性。尤其是在生产环境中，这种方式可以显著降低风险。 Alauda Container Platform Progressive Delivery 使用 **Argo Rollouts**，这是上游 Kubernetes 原生的渐进式发布控制器。它支持高级部署策略，并通过灰度发布、蓝绿部署、分析运行、实验和自动回滚等功能扩展 Kubernetes 的能力。它可与可观测性技术栈集成，以执行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 中的一种自定义资源定义（CRD），用于替代标准 Deployment 资源，从而实现蓝绿部署、灰度发布等高级部署控制。
> - 灰度步骤：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。
> - 暂停步骤：在进入下一灰度步骤前引入等待间隔，以便执行手动或自动验证。

目录

灰度发布的优势

使用 Alauda Container Platform Progressive Delivery 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

准备灰度发布

晋级 Rollout

中止 Rollout（可选）

灰度发布的优势

- 降低风险：通过先将变更部署到一小部分服务器，您可以在全面发布前发现并解决问题，从而最大限度地降低对用户的影响。
- 渐进式发布：这种方式可以逐步开放新功能，有助于您有效监控性能并收集用户反馈。
- 实时反馈：灰度发布可以在真实环境中即时了解新版本的性能和稳定性。
- 灵活性：您可以根据性能指标调整部署流程，从而实现动态发布，并根据需要暂停或回滚发布。
- 经济高效：与蓝绿部署不同，灰度发布不需要单独的环境，因此可以更高效地利用资源。

使用 **Alauda Container Platform Progressive Delivery** 进行灰度发布

Alauda Container Platform Progressive Delivery 使用 Argo Rollouts 支持灰度发布策略，并通过 Gateway API Plugin 控制流量。在 ACP 中，您可以使用 ALB 作为 Gateway API Provider 来实现流量控制。

前提条件

1. 集群中已安装 Argo Rollouts 和 Gateway API plugin。
2. Argo Rollouts kubectl plugin（从此处↗安装）。
3. 一个用于创建命名空间的项目。
4. 集群中已部署 ALB，并已将其分配给项目。
5. 集群中有一个用于部署应用的命名空间。

操作步骤

1 创建 Deployment

首先，定义应用的“stable”版本，即用户当前访问的版本。创建 Kubernetes Deployment，并设置适当的副本数、容器镜像版本（例如 `hello:1.23.1`）以及 `app=web` 等适当的标签。

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`：所需的 Pod 副本数。
- `spec.selector.matchLabels`：定义 Deployment 如何查找要管理的 Pod。
- `template.metadata.labels`：应用于 Pod 的标签，供 Service 用于选择 Pod。
- `spec.containers`：要在每个 Pod 中运行的容器。
- `containers.name`：容器的名称。
- `containers.image`：要运行的容器镜像。
- `containers.ports.containerPort`：容器公开的端口。

使用 `kubectl` 应用配置：

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

这样即可设置生产环境。

或者，您也可以使用 Helm Chart 创建 Deployment 和 Service。

2 创建 Stable Service

创建一个 Kubernetes `Service`，用于公开 stable Deployment。此 Service 将根据匹配的标签把流量转发到 stable 版本的 Pod。初始时，Service selector 会选择带有 `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 Service 的 Kubernetes API 版本。
- `kind`：指定此资源为 Service。
- `metadata.name`：Service 的名称。
- `spec.selector`：根据标签标识要将流量路由到的 Pod。
- `ports.protocol`：使用的协议（TCP）。
- `ports.port`：Service 公开的端口。
- `ports.targetPort`：流量转发到的容器端口。

使用以下命令应用：

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

这样即可从外部访问 stable Deployment。

3 创建 Canary Service

创建一个 Kubernetes `Service`，用于公开 canary Deployment。此 Service 将根据匹配的标签把流量转发到 canary 版本的 Pod。初始时，Service selector 会选择带有 `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 Service 的 Kubernetes API 版本。
- `kind`：指定此资源为 Service。
- `metadata.name`：Service 的名称。
- `spec.selector`：根据标签标识要将流量路由到的 Pod。
- `ports.protocol`：使用的协议（TCP）。
- `ports.port`：Service 公开的端口。
- `ports.targetPort`：流量转发到的容器端口。

使用以下命令应用：

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

这样即可从外部访问 canary Deployment。

4 创建 Gateway

使用 `example.com` 作为访问服务的域名，创建 Gateway，以通过该域名公开服务：

```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 将分配到一个外部 IP 地址。请从 Gateway 资源中类型为 `IPAddress` 的 `status.addresses` 获取该 IP 地址。

```
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 DNS 服务器中配置域名，使其解析到 Gateway 的 IP 地址。使用以下命令验证 DNS 解析结果：

```
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
```

返回结果应为 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**

在集群外部，使用以下命令通过域名访问服务：

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

或者，您也可以在浏览器中访问 `http://example.com`。

8 创建 Rollout

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

```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。此选择器选中的现有 ReplicaSet 将受到此 Rollout 的影响。它必须与 Pod 模板的标签匹配。
- `workloadRef`：指定工作负载引用以及应用 Rollout 时使用的缩容策略。
- `scaleDown`：指定迁移到 Rollout 后是否缩减工作负载（Deployment）。可选值包括：
  - "never"：不缩减 Deployment。
  - "onsuccess"：Rollout 状态变为健康后缩减 Deployment。
  - "progressively"：随着 Rollout 扩容，逐步缩减 Deployment。如果 Rollout 失败，则会重新扩容 Deployment。
- `strategy`：Rollout 策略，支持 `BlueGreen` 和 `Canary` 策略。
- `canary`：`Canary` Rollout 策略定义。
  - `canaryService`：对 Service 的引用，控制器会更新该 Service，使其选择 canary Pod。流量路由必需。
  - `stableService`：对 Service 的引用，控制器会更新该 Service，使其选择 stable Pod。流量路由必需。
  - `steps`：定义更新 canary 期间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 的比例。
    - `pause`：无限期暂停 Rollout，或暂停指定时长。支持的单位：s、m、h。`{}` 表示无限期暂停。
    - `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
```

这样即可为 Deployment 设置 `Canary` 策略的 Rollout。初始权重将设为 50，并等待晋级。50% 的流量将转发到 canary Service。Rollout 晋级后，权重将设为 100，所有流量都将转发到 canary Service。最后，canary Service 将成为 stable Service。

9 验证 Rollout

创建 `Rollout` 后，Argo Rollouts 将使用与 Deployment 相同的模板创建新的 ReplicaSet。新 ReplicaSet 的 Pod 健康后，Deployment 将缩容至 0。

使用以下命令确保 Pod 正常运行：

```
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 准备灰度发布

接下来，将应用的新版本作为 green Deployment 进行准备。使用新镜像版本（例如 `hello:1.23.2`）更新 Deployment `web`。使用以下命令：

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

这样即可准备好新版本应用以供测试。

Rollout 将创建新的 ReplicaSet 来管理 canary Pod，50% 的流量将转发到 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
```

当前共有 3 个 Pod 正在运行，分别使用 stable 和 canary 版本。权重为 50，因此 50% 的流量将转发到 canary Service。Rollout 流程已暂停，等待晋级。

如果您使用 Helm Chart 部署应用，请使用 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。您应该会从 URL 收到不同的响应。

11 晋级 Rollout

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

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

使用以下命令验证 Rollout 是否已完成：

```
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 `Images` 已更新为 `hello:1.23.2`，且 revision 1 的 ReplicaSet 已缩容至 0，则表示 Rollout 已完成。

访问 `http://example.com` 时，所有流量都将转发到 canary Service。

12 中止 **Rollout**（可选）

如果在 Rollout 过程中发现 canary 版本存在问题，可以中止流程，将所有流量切换到 stable Service。使用以下命令：

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

使用以下命令验证结果：

```
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，所有流量都将转发到 stable Service。

由 Alauda 于 2026-10-21 修改。

Argo® 是 LF Projects, LLC 的注册商标。

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

状态说明

目录

原生应用

部署

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。

Status	Meaning
Running	所有计算组件均正常运行。
Partially Running	部分计算组件正在运行，而其他组件已停止。
Stopped	所有计算组件均已停止。
Processing	至少有一个计算组件处于挂起状态。
No Computing Components	该应用下没有计算组件。
Failed	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。

请将这些状态与应用概览和应用详情页面结合使用：

- **Running**：所有受管工作负载均处于健康状态，通常可将该应用视为正在正常运行。
- **Partially Running**：部分工作负载处于健康状态，而其他工作负载已停止、失败或尚未就绪。这通常表示部分发布结果，或仅限于应用边界一部分的依赖问题。
- **Processing**：至少有一个工作负载仍在创建、调度、更新或协调中。
- **Stopped**：所有受管计算组件均已停止。
- **No Computing Components**：该应用当前没有关联任何工作负载资源。
- **Failed**：该应用包含一个或多个失败的工作负载，或部署结果不成功，需要进一步排查。

当应用未处于预期状态时，请继续在应用详情页面检查关联的 Pods、事件、日志和监控数据。

部署

- Running：所有 Pods 均正常运行。
- Processing：存在未处于运行状态的 Pods。
- Stopped：所有 Pods 均已停止。
- Failed：部署失败。

配置 HPA

HPA（Horizontal Pod Autoscaler）会根据预设策略和指标自动增加或减少 pod 数量，使应用能够应对业务负载的突然激增，同时在低流量时段优化资源利用率。

目录

- 理解 Horizontal Pod Autoscaler
  - HPA 如何工作？
  - 支持的指标
- 前提条件
- 创建 Horizontal Pod Autoscaler
  - 使用 CLI
  - 使用 Web Console
  - 将自定义指标用于 HPA
    - 要求
    - 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/v2 中的更新
- 计算规则

理解 Horizontal Pod Autoscaler

您可以创建一个 horizontal pod autoscaler，以指定希望运行的最小和最大 pod 数量，以及 pod 应达到的 CPU 利用率或内存利用率。

在创建 horizontal pod autoscaler 后，平台会开始查询 pod 上的 CPU 和/或内存资源指标。当这些指标可用时，horizontal pod autoscaler 会计算当前指标利用率与目标指标利用率之间的比率，并据此进行扩容或缩容。查询和伸缩会按固定间隔执行，但指标可能需要一到两分钟才会变得可用。

对于 replication controller，这种伸缩会直接对应 replication controller 的 replicas。对于 deployment configuration，这种伸缩会直接对应 deployment configuration 的 replica count。请注意，自动伸缩仅适用于 Complete 阶段中的最新 deployment。

平台会自动考虑资源情况，并防止在资源突增期间进行不必要的自动伸缩，例如启动期间。处于未就绪状态的 Pod 在扩容时 CPU 使用率按 0 处理，而在缩容时 autoscaler 会忽略这些 Pod。没有已知指标的 Pod 在扩容时 CPU 使用率按 0% 处理，在缩容时按 100% CPU 处理。这有助于提高 HPA 决策的稳定性。要使用此功能，您必须配置 readiness 检查，以确定新的 Pod 是否已准备好可供使用。

HPA 如何工作？

horizontal pod autoscaler（HPA）扩展了 pod 自动伸缩的概念。HPA 允许您创建和管理一组负载均衡节点。当给定的 CPU 或内存阈值被跨越时，HPA 会自动增加或减少 pod 数量。

HPA 以控制循环方式工作，默认同步周期为 15 秒。在此期间，controller manager 会根据 HPA 配置中定义的内容，查询 CPU、内存利用率或二者。controller manager 会从 resource metrics API 获取每个由 HPA 选中的 Pod 的逐 Pod 资源指标（如 CPU 或内存）的利用率指标。

如果设置了利用率值目标，controller 会将利用率值计算为每个 Pod 中容器对应资源请求的百分比。然后，controller 会对所有目标 Pod 的利用率取平均值，并生成一个比率，用于调整所需的 replicas 数量。

支持的指标

horizontal pod autoscaler 支持以下指标：

指标	描述
CPU Utilization	使用的 CPU 核心数。可用于计算 Pod 请求的 CPU 百分比。
Memory Utilization	使用的内存量。可用于计算 Pod 请求的内存百分比。
Network Inbound Traffic	进入 Pod 的网络流量，单位为 KiB/s。
Network Outbound Traffic	从 Pod 发出的网络流量，单位为 KiB/s。
Storage Read Traffic	从存储中读取的数据量，单位为 KiB/s。
Storage Write Traffic	写入存储的数据量，单位为 KiB/s。

> 重要：对于基于内存的自动伸缩，内存使用量必须与 replicas 数量成比例增减。平均而言：
>
> - replicas 数量增加必须导致每个 Pod 的内存（working set）使用量整体下降。
> - replicas 数量减少必须导致每个 Pod 的内存使用量整体上升。
> - 请使用平台检查应用的内存行为，并确保应用满足这些要求后，再使用基于内存的自动伸缩。

前提条件

请确保监控组件已部署到当前集群中，并且运行正常。您可以通过单击平台右上角的 > 平台健康状态。 来查看监控组件的部署和健康状态。

创建 **Horizontal Pod Autoscaler**

使用 **CLI**

您可以通过命令行界面创建 horizontal pod autoscaler：先定义一个 YAML 文件，然后使用 `kubectl create` 命令。以下示例展示了对 Deployment 对象进行自动伸缩。初始 deployment 需要 3 个 pod。HPA 对象将最小值提高到 5。如果 pod 上的 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
  metrics: 9
    - type: Resource 10
      resource: 11
        name: cpu 12
        target: 13
          type: Utilization 14
          averageUtilization: 75 15
```

1. 使用 autoscaling/v2 API。
2. HPA 资源的名称。
3. 需要扩缩容的 deployment 名称。
4. 允许扩容到的最大 replicas 数量。
5. 需要维持的最小 replicas 数量。
6. 指定要扩缩容对象的 API 版本。
7. 指定对象类型。该对象必须是 Deployment、ReplicaSet 或 StatefulSet。
8. HPA 适用的目标资源。
9. 触发扩缩容的目标 CPU 利用率百分比。

2. 应用 YAML 文件以创建 HPA：

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

示例输出：

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

3. 创建 HPA 后，您可以通过运行以下命令查看 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. 您还可以检查 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 Console**

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 工作负载 > **Deployments**。
3. 单击 ***Deployment Name***。
4. 向下滚动到 弹性伸缩 区域，然后单击右侧的 更新。
5. 选择 水平伸缩 并完成策略配置。

参数	描述
Pod 数量	deployment 成功创建后，您需要根据已知和常规业务量变化评估对应的 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

参数	描述
	量。设置后仍可修改最大或最小 Pod 数量，建议先通过性能测试得出更准确的值，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，以触发扩容或缩容操作。 例如，如果您设置 *CPU Utilization = 60%*，一旦 CPU 利用率偏离 60%，平台将开始根据当前 deployment 的资源分配不足或过剩情况自动调整 pod 数量。 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 deployment，且必须先添加自定义指标。
扩容/缩容步长（**Alpha**）	对于有特定伸缩速率要求的业务，您可以通过指定 扩容步长 或 缩容步长，逐步适应业务量变化。 对于缩容步长，您可以自定义 稳定窗口，默认值为 300 秒，这意味着在执行缩容操作之前必须等待 300 秒。

6. 单击 更新。

将自定义指标用于 **HPA**

自定义指标 HPA 在原有 HorizontalPodAutoscaler 的基础上进行了扩展，支持除 CPU 和内存利用率之外的其他指标。

要求

- kube-controller-manager: horizontal-pod-autoscaler-use-rest-clients=true
- 预安装 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，可动态调整 Pod 实例数量，如下例所示：

```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
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

在此 YAML 中，`scaleTargetRef` 指定用于扩缩容的工作负载对象，`targetCPUUtilizationPercentage` 指定 CPU 利用率触发指标。

自定义指标 **HPA**

要使用自定义指标，您需要安装 prometheus-operator 和 custom-metrics-api。安装完成后，custom-metrics-api 会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"]
    }
  ]
}
```


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

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: demo
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: demo
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Pods
      pods:
        metric:
          name: metric-demo
        target:
          type: AverageValue
          averageValue: 10
```

在此示例中，`scaleTargetRef` 指定工作负载。

触发条件定义

- `metrics` 是数组类型，支持多个指标
- `metric type` 可以是：Object（描述 k8s 资源）、Pods（描述每个 Pod 的指标）、Resources（内置 k8s 指标：CPU、内存）或 External（通常表示集群外部的指标）
- 如果自定义指标不是由 Prometheus 提供，则需要通过在 Prometheus 中创建规则等一系列操作来创建新指标

指标的主要结构如下：

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

这些指标数据由 Prometheus 收集并更新。

自定义指标 **HPA** 兼容性

自定义指标 HPA YAML 实际上与原始核心指标（CPU）兼容。写法如下：

```
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` 是为每个 Pod 获取的平均值

- `targetAverageUtilization` 是根据直接值计算出的利用率

算法参考如下：

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

autoscaling/v2 中的更新

autoscaling/v2 支持内存利用率：

```yaml
apiVersion: autoscaling/v2
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: 80
          type: Utilization
      type: Resource
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
```

变更：`targetAverageUtilization` 和 `targetAverageValue` 已更改为 `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 Utilization** 和 **Memory Utilization** 指标，只有当实际值波动超出目标阈值 ±10% 的范围时，才会触发自动伸缩。
- 缩容可能会影响正在进行的业务操作；请谨慎执行。

计算规则

当业务指标发生变化时，平台会根据以下规则自动计算与业务量匹配的目标 pod 数量，并进行相应调整。如果业务指标持续波动，该值将调整为设置的 最小 **Pod** 数量 或 最大 **Pod** 数量。

- 单策略目标 Pod 数量：ceil[(sum(actual metric values)/metric threshold)]。这意味着将所有 pod 的实际指标值之和除以指标阈值，再向上取整到大于或等于结果的最小整数。例如：当前有 3 个 pod，其 CPU 利用率分别为 80%、80% 和 90%，设置的 CPU 利用率阈值为 60%。根据公式，pod 数量将自动调整为：ceil[(80%+80%+90%)/60%] = ceil 4.1 = 5 个 pod。
  注意：
  - 如果计算出的目标 pod 数量超过设置的 最大 **Pod** 数量（例如 *4*），平台只会扩容到 4 个 pod。如果在调整最大 pod 数量后指标仍持续偏高，您可能需要使用其他伸缩方式，例如提高命名空间的 pod 配额或增加硬件资源。
  - 如果计算出的目标 pod 数量（在前面的示例中为 *5*）小于按照 扩容步长 调整后的 pod 数量（例如 *10*），平台只会扩容到 5 个 pod。
- 多策略目标 Pod 数量：取各项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)

目录

概述
Release Notes
兼容性矩阵
v4.4 Release Notes
v4.3 Release Notes
生命周期策略
了解 VerticalPodAutoscaler
VPA 如何工作？
支持的功能
先决条件
安装 Vertical Pod Autoscaler 插件
升级 Vertical Pod Autoscaler 插件
创建 VerticalPodAutoscaler
使用 CLI
使用 Web Console
高级 VPA 配置
更新策略选项
容器策略选项
后续操作

概述

Alauda Container Platform Vertical Pod Autoscaler 基于开源 Vertical Pod Autoscaler 组件。它会分析 Pod 的历史资源使用数据，并提供配额建议，以提高资源利用率。

对于无状态和有状态应用，VerticalPodAutoscaler (VPA) 会根据应用的需求自动建议（并可选择应用）更合适的 CPU 和内存资源限制。这有助于确保 Pod 拥有充足的资源，同时提高整体集群资源利用率。

Release Notes

本节详细介绍 **Alauda Container Platform Vertical Pod Autoscaler** 插件的新功能、增强功能、弃用功能和已知问题。

兼容性矩阵

VPA 版本	上游 VPA 版本	ACP v4.1	ACP v4.2	ACP v4.3
v4.4.0	**1.4.5**	—	✓	✓
v4.3.0	**0.11.0**	✓	✓	✓

v4.4 Release Notes

v4.4.0

1. 建议持久化 — 修复 recommender 在 VerticalPodAutoscaler 资源上缺少 `patch` 权限的问题，该问题可能导致资源建议无法保存。附加权限仅限于 recommender ServiceAccount。
2. 安全漏洞修复 — 相较于 v4.3.0，修复了 38 个 CVE，包括更新 Go 工具链和依赖项。安全公告 **ASA-2026:00343** 的严重性为 **HIGH**；完整的 CVE 列表请参阅 灵雀云 Customer Portal 中的公告。
3. 更新运行时镜像 — 使用 Go `1.26.9` 重新构建 admission-controller、recommender 和 updater 镜像，并使用镜像标签 `1.4.8` 发布。上游 VPA 版本仍为 `1.4.5`。软件包适用于 `amd64`、`arm64` 以及两种架构（`ALL`）。

VPA 独立于平台发布。要应用此更新，请按照升级 Vertical Pod Autoscaler 插件中的说明升级 VPA Cluster Plugin。如果已安装较早的 v4.4.0 软件包，请确认全部三个运行时组件均使用更新

后的 `1.4.8` 镜像。

v4.3 Release Notes

v4.3.0

1. 安全漏洞修复。

生命周期策略

下表列出了已发布 **Alauda Container Platform Vertical Pod Autoscaler** 插件版本的生命周期计划：

版本	发布日期	生命周期结束日期
v4.4.0	2026-10-14	2028-10-14
v4.3.0	2026-5-8	2028-5-8

了解 VerticalPodAutoscaler

VerticalPodAutoscaler（VPA）用于根据 Pod 的历史使用模式，建议或自动更新 CPU 和内存资源请求与限制。

创建 VPA 后，平台开始监控目标 Pod 的资源使用情况。收集到足够的数据后，VPA 会计算建议的资源值。根据配置的更新模式，VPA 可以自动应用这些建议，也可以仅提供建议供手动审核和应用。

通过分析一段时间内的资源使用情况，VPA 有助于确保为 Pod 分配其所需的资源，同时避免过度预分配，从而提高整个集群的资源利用效率。

VPA 如何工作？

VerticalPodAutoscaler（VPA）扩展了 Pod 资源优化的概念。VPA 会监控 Pod 的资源使用情况，并根据观察到的使用模式提供 CPU 和内存请求建议。

VPA 会持续监控 Pod 的资源使用情况，并在新数据可用时更新其建议。VPA 可以在以下模式下运行：

- 关闭：VPA 仅提供建议，不会自动应用这些建议。
- 手动调整：您可以根据 VPA 建议手动调整资源配置。

> 重要：当集群资源充足时，弹性伸缩（水平或垂直）效果最佳。当资源紧张时，伸缩操作可能导致 Pod 卡在 `Pending` 状态。请确保集群拥有充足的资源，设置合理的配额，并配置告警以监控伸缩事件。

支持的功能

VPA 根据历史使用模式提供资源建议，帮助您优化 Pod 的 CPU 和内存配置。

> 重要：手动应用 VPA 建议时会重新创建 Pod，这可能会导致应用暂时中断。对于生产工作负载，请考虑在维护窗口期间应用建议。

Info

VPA 建议和 Pod 原地调整大小解决的是资源管理的不同方面。VPA 根据观察到的使用情况建议或应用资源值。Pod 原地调整大小是 Kubernetes 用于通过 Pod `resize` 子资源更新运行中 Pod 的 CPU 或内存资源的机制。某些 VPA 版本支持 `InPlaceOrRecreate`，此时 VPA 可以尝试通过 Pod 原地调整大小来应用建议，并在无法进行原地更新时回退到重新创建 Pod。此模式要求 Kubernetes 支持 Pod 原地调整大小、VPA 组件启用 `InPlaceOrRecreate` 功能，以及 VPA CRD 接受此 `updateMode` 值。有关手动运行时调整和 `resizePolicy` 行为，请参阅在不干扰 Pod 的情况下调整 Pod 资源级别。

先决条件

使用 VPA 前，请确保满足以下条件：

- 集群中已安装 **Alauda Container Platform Vertical Pod Autoscaler** 集群插件。
  - 下载与您的平台版本兼容的最新插件包。

  - 使用 `violet` CLI 工具将 **Alauda Container Platform Vertical Pod Autoscalers** 和 **Alauda DevOps Pipelines** 软件包上传到目标集群。有关使用 `violet` 的详细说明，请参阅 CLI。

- 请确保当前集群中已部署监控组件，并且运行正常。您可以点击平台右上角的 > **Platform Health Status**。，查看监控组件的部署和运行状况。

安装 Vertical Pod Autoscaler 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 **Install**，然后进入安装页面。

升级 Vertical Pod Autoscaler 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 **Upgrade**，然后进入安装页面。

创建 VerticalPodAutoscaler

使用 CLI

您可以通过定义 YAML 文件并使用 `kubectl create` 命令行创建 VerticalPodAutoscaler。以下示例展示了如何为 Deployment 对象配置 Pod 垂直伸缩：

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 selector 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 指定要进行伸缩的对象的 API 版本。

5 指定对象类型。

6 VPA 应用的目标资源。

7 定义 VPA 如何应用建议的更新策略。updateMode 可以是：

- Off：不会自动修改 Pod 资源请求，仅在 VPA 对象中提供建议。
- Initial：仅在创建 Pod 时设置资源请求，之后不再修改。
- Recreate：创建 Pod 时自动设置资源请求，并驱逐当前 Pod 以更新为建议的资源请求。不使用原地更新。
- Auto：创建 Pod 时自动设置资源请求，并将当前 Pod 更新为建议的资源请求。目前等同于“Recreate”，可能导致应用中断。
- InPlaceOrRecreate：仅当集群支持 Pod 原地调整大小、VPA 组件启用 `InPlaceOrRecreate` 功能，并且 VPA CRD 接受此值时可用。VPA 首先尝试通过

Pod 原地调整大小来应用建议；如果无法应用原地更新，则回退到重新创建 Pod。容器的 `resizePolicy` 设置可以决定容器在调整大小期间是否重启。

8 可为不同容器设置特定策略的资源策略。例如，将容器的模式设置为“Auto”表示会为该容器计算建议，而“Off”表示不会计算建议。

9 将策略应用于 Pod 中的所有容器。

10 将模式设置为 Auto 或 Off。Auto 表示将为此容器生成建议，Off 表示不生成建议。

2. 应用 YAML 文件以创建 VPA：

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

示例输出：

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

3. 创建 VPA 后，可以运行以下命令查看建议：

```
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 Console**

1. 登录并进入 **Container Platform**。
2. 在左侧导航栏中，点击 **Workloads** > **Deployments**。
3. 点击 ***Deployment Name***。
4. 向下滚动到 **Elastic Scaling** 区域，然后点击右侧的 **Update**。
5. 选择 **Vertical Scaling** 并配置伸缩规则。

参数	描述
Scaling Mode	当前支持 **Manual Scaling** 模式，该模式通过分析历史资源使用情况提供推荐的资源配置。您可以根据建议值手动调整。调整会导致 Pod 重新创建和重启，因此请选择合适的时间，以免影响正在运行的应用。 通常，Pod 运行超过 8 天后，推荐值会变得准确。 请注意，当集群资源不足时，伸缩可能导致 Pod 处于 Pending 状态。请确保集群拥有充足的资源或合理的配额，或者配置告警以监控伸缩情况。
Target Container	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建议。

6. 点击 **Update**。

高级 VPA 配置

更新策略选项

- `updateMode: "Off"` - VPA 仅提供建议，不会自动应用这些建议。您可以根据需要手动应用这些建议。
- `updateMode: "Initial"` - 仅在创建 Pod 时设置资源请求，之后不再修改。
- `updateMode: "Recreate"` - 创建 Pod 时自动设置资源请求，并驱逐当前 Pod 以更新为建议值。
- `updateMode: "Auto"` - 创建 Pod 时自动设置资源请求，并将当前 Pod 更新为建议值。目前等同于“Recreate”。
- `updateMode: "InPlaceOrRecreate"` - 仅当集群支持 Pod 原地调整大小、VPA 组件启用 `InPlaceOrRecreate` 功能，并且 VPA CRD 接受此值时可用。VPA 首先尝试通过 Pod 原地

调整大小来应用建议；如果无法应用原地更新，则回退到重新创建 Pod。容器的 `resizePolicy` 设置可以决定容器在调整大小期间是否重启。

- `minReplicas: <number>` - 副本数下限。确保 Updater 驱逐 Pod 时仍有不少于此数量的 Pod 可用。必须大于 0。

容器策略选项

- `containerName: "*"` - 将策略应用于 Pod 中的所有容器。
- `mode: "Auto"` - 自动为该容器生成建议。
- `mode: "Off"` - 不为该容器生成建议。

注意：

- VPA 建议基于历史使用数据，因此 Pod 运行数天后，建议值才可能变得准确。
- 在 Auto 或 Recreate 模式下应用 VPA 建议时，会重新创建 Pod，这可能导致应用暂时中断。如果您的 VPA 插件支持 InPlaceOrRecreate 模式，当请求的调整大小无法原地应用时，仍可能回退到重新创建 Pod。

后续操作

配置 VPA 后，可以在弹性伸缩区域查看目标容器的 CPU 和内存资源限制建议值。在容器区域中，选择目标容器选项卡，然后点击资源限制右侧的图标，根据建议值更新资源限制。

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：

```yaml
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

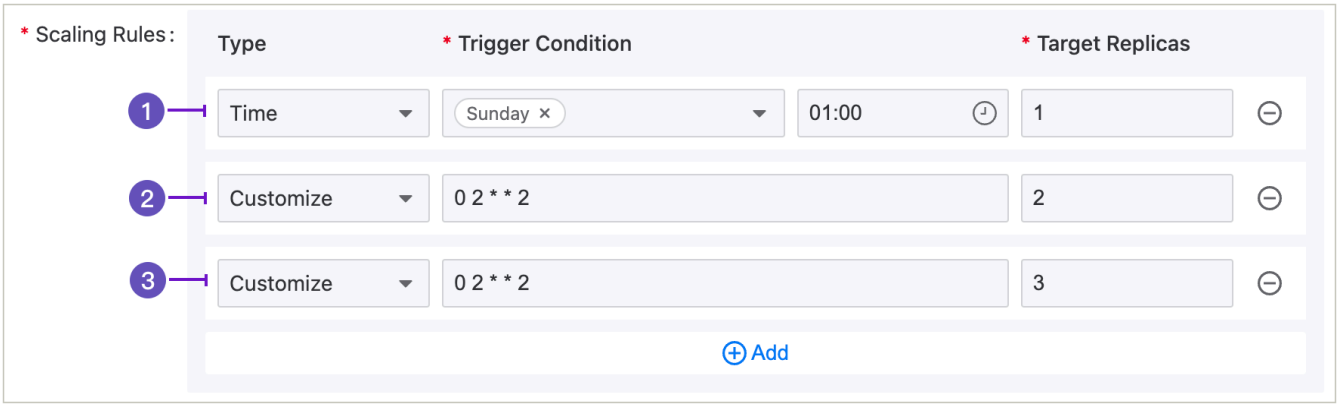
2. 应用 YAML 文件创建 CronHPA：

```
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



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。

更新应用

自定义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应用。

- 在应用创建过程中添加的资源，或通过应用更新添加的资源，默认会关联到该应用，无需额外导入。
- 在应用外部创建的资源不属于该应用，并且无法在应用详情中找到。不过，只要资源定义满足业务需求，业务仍可正常运行。在这种情况下，建议将资源导入到应用中进行统一管理。
- 镜像管理
  - 通过 tag/patch 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/Memory）
- 资源编排
  - 导入现有 Kubernetes 资源（Deployments/Services/Ingresses）
  - 使用 `kubectl apply -f` 在不同 namespace 之间同步配置

导入到应用中的资源可以享受以下特性：

功能	说明
版本快照	在为应用创建版本快照时，也会为应用内的资源生成快照。 • 如果应用回滚，资源也会回滚到快照中的状态。

功能	说明
	• 如果分发应用的某个特定版本，平台会在重新部署应用时自动创建快照中记录的资源。
随应用删除	如果不再需要某个应用，删除该应用时会自动移除所有与该应用关联的资源，包括计算组件、内部路由和入站规则。
更容易查找	在应用详情中，可以快速查看与该应用关联的资源。 例如：外部流量可以通过属于 *Application A* 的 *Service S* 访问 *Deployment D*，但只有当 *Service S* 也属于 *Application A* 时，才能在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

> 批量导入应用所在 namespace 下的相关资源；一个资源只能属于一个应用。

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 **Application Management** > **Native Applications**。
3. 单击 *应用名称*。
4. 单击 **Actions** > **Manage Resources**。
5. 在底部的 **Resource Type** 中，选择要导入的资源类型。
   Note：常见的资源类型包括 Deployment、DaemonSet、StatefulSet、Job、CronJob、Service、Ingress、PVC、ConfigMap、Secret 和 HorizontalPodAutoscaler，这些类型会显示在顶部；其他资源按字母顺序排列，可通过搜索关键字快速查询特定资源类型。

6. 在 **Resources** 区域，选择要导入的资源。
   Attention：对于 **Job** 类型资源，仅支持导入通过 YAML 创建的任务。
7. 单击 **Import Resources**。

移除/批量移除资源

> 从应用中移除/批量移除资源，只会解除应用与资源的关联，不会删除资源本身。

如果应用下的资源之间存在互相关联，从应用中移除任意资源都不会改变资源之间的关联关系。例如，即使将 *Service S* 从 *Application A* 中移除，外部流量仍然可以通过 *Service S* 访问 *Deployment D*。

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 **Application Management** > **Native Applications**。
3. 单击 *应用名称*。
4. 单击 **Actions** > **Manage Resources**。
5. 单击资源右侧的 **Remove** 进行移除；或一次选择多个资源，然后单击表格顶部的 **Remove** 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。

参数	说明
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。

类型	原因

Initialized	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。

Validated	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。

Synced	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。

工作负载再平衡（Descheduler）

Kubernetes 调度是一项基于某个时间点做出的决策。创建 Pod 时，kube-scheduler 会根据当时的集群状态选择最合适的节点。随着集群发生变化，最初的决策可能会变得过时：节点可能变得过度利用或利用不足，污点或标签可能发生变化，亲和性规则可能被更新，故障节点可能恢复，并且还可能添加新节点。

Alauda Container Platform Descheduler 插件通过识别违反调度策略或被放置在不太合适节点上的运行中 Pod，驱逐这些 Pod，并允许默认调度器将替代 Pod 放置到更合适的节点上，从而帮助您重新平衡集群。

Descheduler 不会自行调度替代 Pod。它只调用 Kubernetes 驱逐 API。默认调度器会重新调度由 Deployment、ReplicaSet、StatefulSet 或 Job 等控制器管理的 Pod。

使用 Descheduler 的典型场景包括：

- 节点利用不足或利用过度。
- Pod 完成调度后，节点标签、节点污点、Pod 亲和性或 Pod 反亲和性要求发生了变化。
- 节点故障导致 Pod 被迁移到更少的节点上，恢复的节点应再次接收工作负载。
- 添加了新节点，现有工作负载应重新分布。
- Pod 重启次数过多，或运行时间超过预期生命周期。

目录

Pod 驱逐规则和资格

理解 Descheduler 策略

在 ACP 中安装 Descheduler

配置调度策略

运行模式

编辑策略 YAML

示例：重新平衡利用率过高的节点

常见策略自定义

示例：生命周期和健康状况清理

示例：亲和性、污点和拓扑漂移

验证安装和驱逐

1. 检查插件安装状态
2. 验证策略 ConfigMap
3. 查看 Descheduler 运行情况和日志
4. 检查 Pod 事件

运维指导

Pod 驱逐规则和资格

为维护集群稳定性，Descheduler 必须谨慎决定驱逐哪些 Pod。除非您已确认工作负载能够安全地重新创建，并且驱逐在运维上是可接受的，否则请保留这些保护措施。

- 受保护的 **Pod**：
  - 系统或平台命名空间中的 Pod，例如 `kube-system` 和 ACP 平台命名空间。如果添加命名空间包含或排除规则，请继续排除平台命名空间。
  - 设置了 `priorityClassName` 且值为 `system-cluster-critical` 或 `system-node-critical` 的关键 Pod。
  - 静态 Pod、镜像 Pod 或不受控制器管理的独立 Pod，因为这些 Pod 不会自动重新创建。
  - 与 DaemonSet 关联的 Pod。
  - 使用本地存储的 Pod，除非策略明确禁用 `PodsWithLocalStorage` 保护。
- **Pod Disruption Budget**（**PDB**）：Descheduler 使用驱逐子资源。如果驱逐某个 Pod 会违反其 PDB，则不会驱逐该 Pod。

- 驱逐顺序：当多个 Pod 符合条件时，会先选择优先级较低的 Pod，再选择优先级较高的 Pod。对于优先级相同的 Pod，会先驱逐 BestEffort Pod，再驱逐 Burstable 和 Guaranteed Pod。
- 显式驱逐覆盖：带有 `descheduler.alpha.kubernetes.io/evict` 注解的 Pod 即使某些 Descheduler 内部检查通常会跳过它，也符合驱逐条件。此注解不会绕过 PDB 保护。仅当您清楚 Pod 将如何重新创建时才使用此注解。
- 驱逐偏好：带有 `descheduler.alpha.kubernetes.io/prefer-no-eviction` 注解的 Pod 请求 Descheduler 避免驱逐它。此设置是建议性的还是强制性的，取决于 `DefaultEvictor` `noEvictionPolicy` 设置。

理解 Descheduler 策略

Descheduler 策略用于启用策略插件。请使用与运维目标匹配的策略；避免启用会将集群引向相反方向的策略，例如在同一个策略配置文件中使用 `LowNodeUtilization` 分散工作负载，同时使用 `HighNodeUtilization` 压缩工作负载。

目标	策略	描述
移除重复的副本	`RemoveDuplicates`	当同一工作负载的多个副本 的 Pod。这有助于在节点故 Pod。
将工作负载从过载节点分散出去	`LowNodeUtilization`	查找利用不足的节点，并从 调度器更均衡地分散工作负 Pod 资源请求与节点可分配 `kubectl top` 的实际使用
压缩工作负载	`HighNodeUtilization`	从利用不足的节点驱逐 Po 少的节点上。仅当调度器启 率更高的节点（例如 `Mos` 否则 Pod 可能会再次分散
回收旧 Pod	`PodLifeTime`	驱逐运行时间超过配置生命 的 Pod。此策略适用于类似 略。

目标	策略	描述
移动反复失败的 Pod	`RemovePodsHavingTooManyRestarts`	驱逐其容器重启次数超过配置阈值的 Pod，以便这些 Pod 可以在其他节点上重新调度，从而减少容器重启。
强制执行更新后的污点	`RemovePodsViolatingNodeTaints`	驱逐不再容忍其当前节点上污点的 Pod。
强制执行所需的节点亲和性	`RemovePodsViolatingNodeAffinity`	驱逐当前节点不再满足所需节点亲和性规则的 Pod。例如，`requiredDuringSchedul`
强制执行 Pod 间反亲和性	`RemovePodsViolatingInterPodAntiAffinity`	在集群状态或工作负载定义发生变化后，驱逐违反 Pod 间反亲和性规则的 Pod。
重新平衡拓扑域	`RemovePodsViolatingTopologySpreadConstraint`	当拓扑分布约束被违反时驱逐 Pod。默认使用硬约束，例如 `DoN` 考虑软拓扑约束时，才包含

常见策略组：

组	适用场景	策略
亲和性和污点	您更改了污点、标签、节点亲和性或 Pod 间反亲和性，并希望运行中的 Pod 遵循新规则。	`RemovePodsViolatingNodeTaints`、`RemovePodsViolatingNodeAffinity`、`RemovePodsViolatingInterPodAntiAffinity`
拓扑和重复 Pod	您需要将相似的 Pod 分散到各个节点或拓扑域。	`RemoveDuplicates`、`RemovePodsViolatingTopologySpreadConstraint`
生命周期和利用率	您需要移除过时或不健康的 Pod，并将工作负载从过载节点分散出去。	`RemovePodsHavingTooManyRestarts`、`PodLifeTime`、`LowNodeUtilization`
压缩和扩缩容	您希望将工作负载集中到更少的节点上，以帮助缩减未使用的容量。	`HighNodeUtilization`

在 ACP 中安装 Descheduler

Alauda Container Platform Descheduler 在 ACP 中以 **Cluster Plugin** 的形式打包和管理。

1. 上架插件包：

   - 从 Alauda Customer Portal 获取 `Alauda Container Platform Descheduler` 插件包。
   - 使用 `violet` 工具将其发布到平台。有关 CLI 的详细说明，请参阅 CLI 工具。
   - 前往 管理员 > **Marketplace** > 上架软件包，并在 集群插件 标签页下确认软件包已存在。

2. 安装插件：

   - 前往 管理员 > **Marketplace** > 集群插件。
   - 选择目标集群，找到 **Alauda Container Platform Descheduler** 插件，然后点击 安装。
   - 如有需要，请在动态表单中调整安装时的配置选项，然后确认安装。

配置调度策略

安装插件后，请勿直接编辑 Helm 值。已安装的插件会渲染 Kubernetes 资源，运行时 descheduler 策略以 YAML 格式存储在 descheduler ConfigMap 中。

运行模式

- **CronJob**（推荐）：descheduler 定期以 Job 的形式运行。当集群状态稳定时，此模式可避免运行持久化智能体。更新后的策略 YAML 会由下一个计划运行的 Job 加载。
- **Deployment**：descheduler 以持续运行的 Pod 形式运行，并按插件安装期间配置的间隔协调集群。更新策略 YAML 后，重启 descheduler Pod，使正在运行的进程重新加载配置。

编辑策略 YAML

找到 descheduler 策略 ConfigMap：

```
kubectl get configmap -n kube-system -l app.kubernetes.io/name=deschedule
r
```

编辑前备份当前策略：

```
kubectl get configmap -n kube-system <descheduler-configmap-name> -o yaml
> descheduler-configmap.backup.yaml
```

编辑上一个命令返回的 ConfigMap：

```
kubectl edit configmap -n kube-system <descheduler-configmap-name>
```

在 `data.policy.yaml` 中，保留现有策略，仅合并需要更改的字段。不要用示例替换整个策略，否则可能会移除现有的保护措施、已启用的策略、命名空间筛选器或驱逐限制。

示例：重新平衡利用率过高的节点

此示例启用 `LowNodeUtilization`，并设置常用于分散式 descheduling 的相同阈值模式：CPU、内存和 Pod 容量低于 `20%` 的节点属于利用不足；上述任一资源利用率高于 `50%` 的节点属于利用过度。

相关策略片段：

```yaml
apiVersion: "descheduler/v1alpha2"
kind: "DeschedulerPolicy"
maxNoOfPodsToEvictTotal: 20
profiles:
  - name: default
    pluginConfig:
      - name: DefaultEvictor
        args:
          nodeFit: true
      - name: LowNodeUtilization
        args:
          thresholds:
            cpu: 20
            memory: 20
            pods: 20
          targetThresholds:
            cpu: 50
            memory: 50
            pods: 50
          evictionLimits:
            node: 5
    plugins:
      balance:
        enabled:
          - LowNodeUtilization
```

注意：

- `thresholds` 和 `targetThresholds` 必须定义相同的资源键。
- 有效百分比范围为 `0` 到 `100`。
- 对于同一资源，`thresholds` 不得大于 `targetThresholds`。
- 只有在至少存在一个利用不足的节点和一个利用过度的节点时，该策略才会运行。
- 默认情况下，节点利用率根据 Pod 资源请求和节点可分配资源计算。如果需要基于实际使用量进行 descheduling，请在策略中配置受支持的指标提供程序和 `metricsUtilization`。

常见策略自定义

需求	策略位置	说明
限制每次运行造成的总中断	顶层 `maxNoOfPodsToEvictTotal`、`maxNoOfPodsToEvictPerNode` 或 `maxNoOfPodsToEvictPerNamespace`	从较小 行为后
驱逐前检查替换 Pod 是否能够容纳	`DefaultEvictor.args.nodeFit: true`	根据当 和性、 防止驱 行的 P
排除或包含命名空间	策略 `args.namespaces` 或 `LowNodeUtilization` / `HighNodeUtilization` `args.evictableNamespaces`	使用 `excl` 将 AC 命名空
按 Pod 优先级限制	`DefaultEvictor.args.priorityThreshold.name` 或 `DefaultEvictor.args.priorityThreshold.value`	不要同 `valu` 在。
驱逐旧 Pod	`PodLifeTime.args.maxPodLifeTimeSeconds`	值以秒 `8640`
驱逐重启次数过多的 Pod	`RemovePodsHavingTooManyRestarts.args.podRestartThreshold`	如果应 计入， `incl` `true`
考虑软拓扑分布约束	`RemovePodsViolatingTopologySpreadConstraint.args.constraints`	仅当软 添加
允许驱逐使用本地存储的 Pod	使用 `PodsWithLocalStorage` 配置 `DefaultEvictor.args.podProtections.defaultDisabled`	这会禁 够安全
保护由 PVC 支持的 Pod	使用 `PodsWithPVC` 配置 `DefaultEvictor.args.podProtections.extraEnabled`	这会为 额外保

示例：生命周期和健康状况清理

此片段会驱逐运行时间超过 24 小时的 Pod，以及容器重启次数超过 100 次的 Pod：

```yaml
profiles:
  - name: default
    pluginConfig:
      - name: PodLifeTime
        args:
          maxPodLifeTimeSeconds: 86400
      - name: RemovePodsHavingTooManyRestarts
        args:
          podRestartThreshold: 100
          includingInitContainers: true
    plugins:
      deschedule:
        enabled:
          - PodLifeTime
          - RemovePodsHavingTooManyRestarts
```

示例：亲和性、污点和拓扑漂移

此片段会驱逐不再符合更新后的节点污点、所需节点亲和性、Pod 间反亲和性或硬拓扑分布约束的 Pod：

```
profiles:
  - name: default
    pluginConfig:
      - name: RemovePodsViolatingNodeTaints
      - name: RemovePodsViolatingNodeAffinity
        args:
          nodeAffinityType:
            - requiredDuringSchedulingIgnoredDuringExecution
      - name: RemovePodsViolatingInterPodAntiAffinity
      - name: RemovePodsViolatingTopologySpreadConstraint
        args:
          constraints:
            - DoNotSchedule
    plugins:
      deschedule:
        enabled:
          - RemovePodsViolatingNodeTaints
          - RemovePodsViolatingNodeAffinity
          - RemovePodsViolatingInterPodAntiAffinity
      balance:
        enabled:
          - RemovePodsViolatingTopologySpreadConstraint
```

验证安装和驱逐

1. 检查插件安装状态

验证 `ModuleInfo` 是否已转换为 `Running` 状态：

```
kubectl get moduleinfo -l cpaas.io/module-name=descheduler
```

2. 验证策略 ConfigMap

检查策略 YAML 是否包含预期的策略和保护措施：

```
kubectl get configmap -n kube-system <descheduler-configmap-name> -o yaml
```

3. 查看 Descheduler 运行情况和日志

如果以 `CronJob` 运行，请列出已完成或正在运行的 Job：

```
kubectl get jobs -n kube-system -l app.kubernetes.io/name=descheduler
```

如果以 `Deployment` 运行，请确认正在运行的 Pod，并在策略更改后重新启动该 Pod：

```
kubectl get pods -n kube-system -l app.kubernetes.io/name=descheduler
kubectl rollout restart deployment -n kube-system -l app.kubernetes.io/na
me=descheduler
```

获取 Descheduler 日志，以检查节点评估、跳过的 Pod 和驱逐操作：

```
kubectl logs -n kube-system -l app.kubernetes.io/name=descheduler --tail=
100
```

驱逐日志示例：

```
I0601 15:30:15.123456       1 evictions.go:160] "Evicting pod" pod="defau
lt/my-app-67d7f8d68c-xxxxx" reason="RemoveDuplicates"
```

4. 检查 Pod 事件

当 Pod 被 Descheduler 驱逐时，请检查 Pod 事件和 Pod 描述。事件原因可能因策略而异，因此不要依赖单个 `reason=Descheduled` 过滤条件。

```
kubectl get events -n <namespace> --field-selector involvedObject.kind=Po
d --sort-by=.lastTimestamp
kubectl describe pod -n <namespace> <pod-name>
```

运维指导

- 从较小的范围开始：限制命名空间、设置保守的驱逐限制，并在扩大策略范围前验证日志。
- 在启用可能驱逐大量 Pod 的策略之前，确保工作负载具有控制器和足够的副本数。
- 确保关键应用的 PDB 保持最新。Descheduler 会遵循 PDB，但缺少 PDB 时不会提供任何中断预算。
- 仅当调度器或 autoscaler 配置为压缩 Pod 时，才使用 `HighNodeUtilization`。否则，被驱逐的 Pod 可能会再次分散。
- 除非工作负载具有明确且经过测试的恢复路径，否则不要禁用本地存储、DaemonSet、系统关键或独立 Pod 保护。
- 对于基于实际使用量的决策，请确认已配置 metrics provider，并且所选策略会使用这些指标；否则，利用率策略将基于 requests。

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。

使用 PodDisruptionBudgets

目录

了解 PodDisruptionBudgets
创建 PodDisruptionBudgets
使用 CLI 创建 PDB
前提条件
YAML 文件示例
通过 YAML 创建 PDB
使用 CLI 管理 PDB
查看 PDB
更新 PDB
删除 PDB
何时使用 PodDisruptionBudget

了解 PodDisruptionBudgets

请参阅 Kubernetes 官方文档：为您的原生应用指定中断预算↗

PodDisruptionBudget（PDB）允许您在 Pod 因升级或 Kubernetes 节点上的例行维护等原因需要重新调度时，限制对原生应用造成的中断。

作为集群管理员或原生应用所有者，您可以为每个原生应用创建一个 PodDisruptionBudget。PDB 会限制复制型原生应用中因自愿中断而同时宕机的 Pod 数量。例如，基于仲裁的应用希望确保运行中的副本数量绝不会低于维持仲裁所需的数量。Web 前端可能希望确保提供负载服务的副本数量绝不会低于总数的某个百分比。

通过使用 **PodDisruptionBudgets**，您可以：

- 确保高可用性：在节点维护、集群升级或其他自愿中断期间，保持最少数量的可用副本。
- 控制中断影响：通过指定 `minAvailable` 或 `maxUnavailable`，精确定义原生应用可以容忍多少中断。
- 安全地清空节点：在执行 `kubectl drain` 等操作期间，系统会遵守 PDB，并等待直到可以在不违反中断预算的情况下安全驱逐 Pod。

工作原理：

1. 您使用标签选择器定义一个 PodDisruptionBudget 资源，以针对一组特定的 Pod。
2. 您指定 `minAvailable`（必须保持可用的 Pod 的最小数量或百分比）或 `maxUnavailable`（可以同时不可用的 Pod 的最大数量或百分比）。
3. 当发生自愿中断（例如为进行 OS 升级而清空节点）时，eviction API 会检查 PDB。如果驱逐 Pod 会违反预算，则会阻止或延迟驱逐，直到有足够数量的 Pod 重新变为可用。

创建 PodDisruptionBudgets

使用 CLI 创建 PDB

前提条件

- 确保您已配置 `kubectl` 并已连接到集群。
- 您必须有一个运行多个副本的原生应用（例如 Deployment 或 StatefulSet）。

YAML 文件示例

```yaml
# example-pdb.yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: nginx-pdb
spec:
  minAvailable: 2 # Alternatively, you can use maxUnavailable: 1
  selector:
    matchLabels:
      app: nginx
```

通过 YAML 创建 PDB

```bash
# Step 1: Create the PodDisruptionBudget
kubectl apply -f example-pdb.yaml

# Step 2: Verify the PDB status
kubectl get pdb nginx-pdb
```

输出将显示允许的中断数、最小可用数以及当前健康的 Pod 数。

使用 CLI 管理 PDB

查看 PDB

- 列出命名空间中的所有 PDB：

  ```bash
  kubectl get poddisruptionbudgets
  ```

- 获取某个特定 PDB 的详细信息：

  ```bash
  kubectl describe poddisruptionbudget <pdb-name>
  ```

更新 PDB

您可以通过编辑配置来更新 PDB。例如，要更改 `minAvailable`：

```
kubectl edit poddisruptionbudget <pdb-name>
```

修改 `minAvailable` 或 `maxUnavailable` 字段，然后保存并退出。

删除 **PDB**

如果您不再需要使用 PDB 保护工作负载，可以安全地将其删除：

```
kubectl delete poddisruptionbudget <pdb-name>
```

删除 PDB 不会影响实际运行中的原生应用，它只会移除用于节点维护和驱逐过程的中断预算约束。

何时使用 **PodDisruptionBudget**

在以下情况下，您应考虑使用 PDB：

- 您的原生应用需要严格的最小副本数来处理流量或维持仲裁（例如 ZooKeeper、Elasticsearch）。
- 您在一个集群上运行工作负载，而集群管理员经常执行节点升级或扩缩操作。
- 您希望在诸如节点清空之类的自愿中断期间确保零停机。

PDB 不适用于非自愿中断（例如节点硬件故障、kernel panic），因为这些中断无法由 Kubernetes 延迟或阻止。

为工作负载动态注入 PSA 安全上下文

在命名空间上启用 Pod Security Admission（PSA）后，在其中创建的任何 Workload（例如 Deployment 或 StatefulSet）都必须符合指定的安全配置文件。如果 Workload 的 `securityContext` 不符合要求，则由该 Workload 生成的 Pod 将在 Pod 准入期间被 Kubernetes API server 拒绝，即使 Workload 资源本身已被接受。

为了简化用户体验并确保持续合规，您可以将 **Kyverno** 与 **Alauda Container Platform Policy Management** 插件结合使用，根据命名空间的 PSA 策略动态注入所需的 `securityContext` 字段。

目录

前提条件

工作原理

PSA 强制执行配置

1. Restricted 策略变更

2. Baseline 策略变更

验证

前提条件

继续操作前，请确保：

- 集群中已安装并启用 **Alauda Container Platform Policy Management** 插件。有关安装详情，请参阅安装 Compliance 插件。
- 目标命名空间已应用 PSA 标签，例如 `pod-security.kubernetes.io/enforce: restricted` 或 `baseline`。

工作原理

1. 命名空间标记：为命名空间分配特定的 PSA 级别（例如 `restricted`）。
2. **Kyverno** 拦截：用户提交 Workload 创建请求时，Kyverno 会在准入阶段拦截该请求。
3. **Compliance** 插件变更：Kyverno 与 Compliance 插件协同工作，读取目标命名空间的 PSA 配置。
4. 动态修补：Kyverno 根据 PSA 级别，使用适当的 `securityContext` 动态修补 Workload 的 Pod 模板（例如添加 `runAsNonRoot: true`、删除特定 capabilities，或设置 `seccompProfile`）。

PSA 强制执行配置

安装 Compliance 插件后，您可以应用以下 `ClusterPolicy` 配置。这些策略使用上下文变量查询命名空间的 `pod-security.kubernetes.io/enforce` 标签，并针对相应的 PSA 配置文件，在所有容器类型（包括 `initContainers` 和 `ephemeralContainers`）中应用所需的精确 `securityContext` 变更。

1. Restricted 策略变更

当命名空间强制执行 `restricted` 配置文件时，Workload 必须删除所有权限、以非 root 用户身份运行，并使用默认 seccomp 配置文件。

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: mutate-psa-restricted
  annotations:
    policies.kyverno.io/title: Mutate Workload for PSA Restricted
    policies.kyverno.io/description: >-
      Dynamically injects the Restricted PSA securityContext into Workloa
ds based on the target Namespace label.
spec:
  rules:
    - name: inject-restricted-security-context-controllers
      match:
        any:
          - resources:
              kinds:
                - Deployment
                - StatefulSet
                - DaemonSet
                - Job
                - CronJob
      context:
        - name: namespacePSA
          apiCall:
            urlPath: "/api/v1/namespaces/{{request.namespace}}"
            jmesPath: "metadata.labels.\"pod-security.kubernetes.io/enfor
ce\" || 'none'"
      preconditions:
        all:
        - key: "{{namespacePSA}}"
          operator: Equals
          value: "restricted"
      mutate:
        patchStrategicMerge:
          spec:
            template:
              spec:
                securityContext:
                  runAsNonRoot: true
                  seccompProfile:
                    type: RuntimeDefault
                containers:
                  - (name): "?*"
```

```
                    securityContext:
                      allowPrivilegeEscalation: false
                      capabilities:
                        drop:
                          - ALL
                  initContainers:
                    - (name): "?*"
                      securityContext:
                        allowPrivilegeEscalation: false
                        capabilities:
                          drop:
                            - ALL
                  ephemeralContainers:
                    - (name): "?*"
                      securityContext:
                        allowPrivilegeEscalation: false
                        capabilities:
                          drop:
                            - ALL
    - name: inject-restricted-security-context-pods
      match:
        any:
          - resources:
              kinds:
                - Pod
      context:
        - name: namespacePSA
          apiCall:
            urlPath: "/api/v1/namespaces/{{request.namespace}}"
            jmesPath: "metadata.labels.\"pod-security.kubernetes.io/enfor
ce\" || 'none'"
      preconditions:
        all:
        - key: "{{namespacePSA}}"
          operator: Equals
          value: "restricted"
      mutate:
        patchStrategicMerge:
          spec:
            securityContext:
              runAsNonRoot: true
              seccompProfile:
                type: RuntimeDefault
            containers:
```

```
    - (name): "?*"
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
  initContainers:
    - (name): "?*"
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
  ephemeralContainers:
    - (name): "?*"
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
```

2. Baseline 策略变更

当命名空间强制执行 `baseline` 配置文件时，Workload 禁止使用主机命名空间和特权容器。

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: mutate-psa-baseline
  annotations:
    policies.kyverno.io/title: Mutate Workload for PSA Baseline
    policies.kyverno.io/description: >-
      Dynamically injects the Baseline PSA securityContext into Workloads
based on the target Namespace label.
spec:
  rules:
    - name: inject-baseline-security-context-controllers
      match:
        any:
          - resources:
              kinds:
                - Deployment
                - StatefulSet
                - DaemonSet
                - Job
                - CronJob
      context:
        - name: namespacePSA
          apiCall:
            urlPath: "/api/v1/namespaces/{{request.namespace}}"
            jmesPath: "metadata.labels.\"pod-security.kubernetes.io/enfor
ce\" || 'none'"
      preconditions:
        all:
        - key: "{{namespacePSA}}"
          operator: Equals
          value: "baseline"
      mutate:
        patchStrategicMerge:
          spec:
            template:
              spec:
                hostNetwork: false
                hostIPC: false
                hostPID: false
                containers:
                  - (name): "?*"
                    securityContext:
```

```
                    privileged: false
              initContainers:
                - (name): "?*"
                  securityContext:
                    privileged: false
              ephemeralContainers:
                - (name): "?*"
                  securityContext:
                    privileged: false
  - name: inject-baseline-security-context-pods
    match:
      any:
        - resources:
            kinds:
              - Pod
    context:
      - name: namespacePSA
        apiCall:
          urlPath: "/api/v1/namespaces/{{request.namespace}}"
          jmesPath: "metadata.labels.\"pod-security.kubernetes.io/enfor
ce\" || 'none'"
    preconditions:
      all:
      - key: "{{namespacePSA}}"
        operator: Equals
        value: "baseline"
    mutate:
      patchStrategicMerge:
        spec:
          hostNetwork: false
          hostIPC: false
          hostPID: false
          containers:
            - (name): "?*"
              securityContext:
                privileged: false
          initContainers:
            - (name): "?*"
              securityContext:
                privileged: false
          ephemeralContainers:
            - (name): "?*"
              securityContext:
                privileged: false
```

> 注意：对于 `privileged` PSA 配置文件，无需进行变更，因为它允许不受限制的访问。

验证

要验证动态配置是否正常工作：

1. 使用 `restricted` PSA 标签创建命名空间：

```
kubectl create namespace test-psa
kubectl label namespace test-psa pod-security.kubernetes.io/enforce=res
tricted
```

2. 部署一个未指定 `securityContext` 的简单原生应用：

```
kubectl create deployment nginx --image=nginxinc/nginx-unprivileged -n
test-psa
```

3. 检查所创建 Pod 的 YAML 定义：

```
kubectl get pod -n test-psa -l app=nginx -o yaml
```

   您应看到自动注入的 `securityContext` 字段，以满足 `restricted` 配置文件的要求。

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：

```
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：

```
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：

```
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：

```
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：

```
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。

工作负载

本节中的工作负载页面涵盖项目范围内用于运行应用代码、有限任务和计划任务的运行时资源。

当您需要在所选项目、cluster 和 namespace 上下文中创建、查看、更新、排查或删除工作负载时，请使用这些页面。

某些工作负载类型也可以从其他控制台视图访问，但本节围绕工作负载意图进行组织，这样您可以更快地选择正确的操作路径。

目录

长时间运行的应用工作负载

计划或有限执行工作负载

选择正确的工作负载页面

长时间运行的应用工作负载

将这些页面用于持续运行并代表应用主要运行时单元的工作负载：

当您从 Applications 视图管理这些工作负载时，请将所选 cluster 和 namespace 视为固定的运行时范围。更新操作会更改当前范围内的工作负载定义，但不会将工作负载移动到其他 cluster 或 namespace。

- Deployments
- StatefulSets
- DaemonSets
- Pods

计划或有限执行工作负载

将这些页面用于在完成后停止或按计划运行的工作负载：

- CronJobs
- Jobs

选择正确的工作负载页面

在大多数情况下：

- 常见的无状态应用工作负载请从 **Deployments** 开始。
- 当稳定标识、按顺序滚动发布或每个从节点的存储很重要时，请使用 **StatefulSets**。
- 当工作负载应在匹配的节点上广泛运行时，请使用 **DaemonSets**。
- 当主要问题是计划执行或有限执行时，请使用 **CronJobs** 或 **Jobs**。
- 当您需要检查更高级别工作负载背后的实际运行时实例时，请使用 **Pods**。

部署

理解 Deployments

创建 Deployments

何时在 Applications 视图中使用 Deployme

提交前的 Deployment 检查点

管理 Deployments

使用 CLI 排障

守护进程集

理解 DaemonSets

创建 DaemonSets

管理 DaemonSets

定时任务

有状态副本集

了解有状态副本

创建有状态副本

何时在 Applicat

有状态工作负载

了解定时任务

创建定时任务

Jobs

理解 Jobs

YAML 文件示例

执行概览

Pods

理解 Pod

YAML 文件示例

Containers

理解 Containers

在不影响 **Pod** 的情况下调整 **Pod** 资源级别

前提条件

关于原地 Pod 调整大小

配置原地 Pod 调整大小

调整正在运行的 Pod

验证重启行为

检查调整大小状态

限制

其他资源

部署

目录

- 理解 Deployments
- 创建 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 console 创建 Deployment
    - 前提条件
  - 使用新的 web console 创建 Deployment
    - 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
- 何时在 Applications 视图中使用 Deployment
- 提交前的 Deployment 检查点
- 管理 Deployments
  - 使用 CLI 管理 Deployment
    - 查看 Deployment
    - 更新 Deployment

扩缩容 Deployment
回滚 Deployment
删除 Deployment
使用 web console 管理 Deployment
查看 Deployment
更新 Deployment
删除 Deployment
使用 CLI 排障
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
检查健康配置
检查资源限制

理解 **Deployments**

请参考 Kubernetes 官方文档：Deployments ↗

> **Deployment** 是 Kubernetes 中用于以声明式方式管理和更新应用 Pod 副本的高级工作负载资源。它提供了一种强大而灵活的方式来定义应用的运行方式，包括要保持多少副本，以及如何安全地执行滚动更新。

Deployment 是 Kubernetes API 中用于管理 Pod 和 ReplicaSets 的对象。创建 Deployment 时，Kubernetes 会自动创建一个 ReplicaSet，然后由该 ReplicaSet 负责维持指定数量的 Pod 副本。

使用 **Deployments**，你可以：

- 声明式管理：定义应用的期望状态，Kubernetes 会自动确保集群的实际状态与期望状态一致。

- 版本控制和回滚：跟踪 Deployment 的每个修订版本，并在出现问题时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，而不会中断服务。
- 自愈：如果 Pod 崩溃、终止或从节点上移除，Deployment 会自动替换 Pod 实例，确保始终可用的 Pod 数量符合预期。

其工作原理如下：

1. 你通过 Deployment 定义应用的期望状态（例如使用哪个 image、运行多少个副本）。
2. Deployment 创建一个 ReplicaSet，以确保运行指定数量的 Pod。
3. ReplicaSet 创建并管理实际的 Pod 实例。
4. 当你更新 Deployment 时（例如更改 image 版本），Deployment 会创建一个新的 ReplicaSet，并根据预定义的滚动更新策略逐步用新的 Pod 替换旧的 Pod，直到所有新 Pod 都运行起来，然后再删除旧的 ReplicaSet。

创建 **Deployments**

使用 **CLI** 创建 **Deployment**

前提条件

- 确保已配置 `kubectl`，并且已连接到集群。

YAML 文件示例

```yaml
# example-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment # Name of the Deployment
  labels:
    app: nginx # Labels for identification and selection
spec:
  replicas: 3 # Desired number of Pod replicas
  selector:
    matchLabels:
      app: nginx # Selector to match Pods managed by this Deployment
  template:
    metadata:
      labels:
        app: nginx # Pod's labels, must match selector.matchLabels
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2 # Container image
          ports:
            - containerPort: 80 # Container exposed port
          resources: # Resource limits and requests
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 200m
              memory: 256Mi
```

通过 **YAML** 创建 **Deployment**

```shell
# Step 1: Create Deployment via yaml
kubectl apply -f example-deployment.yaml

# Step 2: Check the Deployment status
kubectl get deployment nginx-deployment # View Deployment
kubectl get pod -l app=nginx # View Pods created by this Deployment
```

使用 web console 创建 Deployment

前提条件

获取 image 地址。image 的来源可以是平台管理员通过 toolchain 集成的 image repository，也可以是第三方平台的 image repository。

- 对于前者，管理员通常会将 image repository 分配给你的项目，你可以使用其中的 image。如果未找到所需的 image repository，请联系管理员进行分配。
- 如果是第三方平台的 image repository，请确保当前集群能够直接从中拉取 image。
- 如果 image registry 需要认证，则需要配置相应的 image pull secret。更多信息，请参见 将 ImagePullSecrets 添加到 ServiceAccount。
- 通过 OCI Connector 选择 image 时，还需要为 Connector proxy 选择一个 image pull Secret，或者确认该工作负载的 ServiceAccount 已经引用了该 Secret。这两种引用方式是二选一的。请参见 在 OCI Connector 中使用 ImagePullSecret。

使用新的 web console 创建 Deployment

新的 web console 可以从不同视图暴露 Deployment 创建入口。请根据你是从项目范围还是从具体集群范围操作，选择合适的入口。

- **Applications** 视图：从当前项目开始。打开 **Applications** > **Applications**，选择目标 **Cluster** 和 **Namespace**，然后选择会创建 Deployment 的工作负载模型或创建流程。
- **Clusters** 视图：从具体集群开始。先打开目标集群，进入目标 namespace，然后导航到 **Workloads** > **Deployments**，点击 **Create Deployment**。

继续之前，请确认以下差异：

- 在 **Applications** 视图中，集群选择是项目范围应用工作流的一部分。
- 在 **Clusters** 视图中，由于你是从具体集群中进入，因此集群已经固定。
- 在 **Applications** 视图中，请确认 namespace 是否属于预期的项目范围。
- 在 **Clusters** 视图中，请确认该集群中的 namespace 侧资源是否已准备好供 Deployment 使用。

操作步骤 - 配置基本信息

1. 在 **Container Platform** 左侧边栏中，导航到 **Workloads** > **Deployments**。
2. 点击 **Create Deployment**。
3. **Select** 或 **Input** 一个 image，然后点击 **Confirm**。

> **INFO**
>
> 说明：当使用已集成到 web console 的 image repository 中的 image 时，可以通过 **Already Integrated** 进行筛选。**Integration Project Name**，例如 images (registry-projectname)，其中在该 web console 中包含项目名 projectname，而在 image repository 中包含项目名 containers。

4. 在 **Basic Info** 区域中，为 Deployment 工作负载配置声明式参数：

Parameters	Description
Replicas	定义 Deployment 中期望的 Pod 副本数量（默认值：`1`）。请根据工作负载需求进行调整。
More > **Update Strategy**	配置用于零停机部署的 `rollingUpdate` 策略： **Max surge**（`maxSurge`）： • 更新期间，允许超过期望副本数的最大 Pod 数量。 • 可接受绝对值（例如 `2`）或百分比（例如 `20%`）。 • 百分比计算方式：`ceil(current_replicas × percentage)`。 • 示例：当有 10 个副本时，计算结果 4.1 会取为 `5`。 **Max unavailable**（`maxUnavailable`）： • 更新期间，允许临时不可用的最大 Pod 数量。 • 百分比值不能超过 `100%`。 • 百分比计算方式：`floor(current_replicas × percentage)`。 • 示例：当有 10 个副本时，计算结果 4.9 会取为 `4`。 注意： 1. 默认值：如果未显式设置，则 `maxSurge=1`、`maxUnavailable=1`。

Parameters	Description
	2. 未运行的 **Pod**（例如处于 `Pending` / `CrashLoopBackOff` 状态）会被视为不可用。 3. 同时约束： • `maxSurge` 和 `maxUnavailable` 不能同时为 `0` 或 `0%`。 • 如果百分比计算后两个参数都为 `0`，Kubernetes 会强制将 `maxUnavailable=1`，以确保更新能够继续。 示例： 对于一个有 10 个副本的 Deployment： • `maxSurge=2` → 更新期间的 Pod 总数：`10 + 2 = 12`。 • `maxUnavailable=3` → 最少可用 Pod 数：`10 - 3 = 7`。 • 这样既能保证可用性，又能实现受控发布。

操作步骤 - 配置 **Pod**

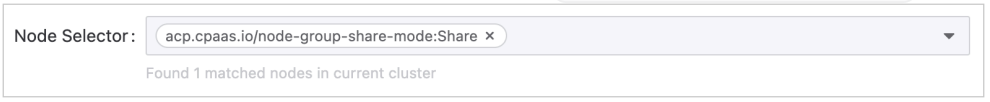
说明：在混合架构集群中部署单架构 image 时，请确保已正确配置 节点亲和性规则，以便进行 Pod 调度。

1. 在 **Pod** 区域中，配置容器运行时参数和生命周期管理：

Parameters	Description
Volumes	将持久卷挂载到容器。支持的卷类型包括 `PVC`、`ConfigMap`、`Secret`、`emptyDir`、`hostPath` 等。实现细节请参见 Volume Mounting Guide。
Pull Secret	对于私有 registry image，请使用包含该 registry 凭据的 Secret。对于 OCI Connector image，请选择 Connector proxy 的 image pull Secret，或者使用已经引用该 Secret 的 ServiceAccount。来自平台集成的 image 将继续使用其关联凭据。请参见 在 OCI Connector 中使用 ImagePullSecret。

Parameters	Description
Close Grace Period	Pod 在接收到终止信号后允许完成优雅关闭的时长（默认值：`30s`）。 - 在此期间，Pod 会完成正在处理中的请求并释放资源。 - 设置为 `0` 会强制立即删除（SIGKILL），这可能会导致请求中断。

2. 节点亲和性规则

Parameters	Description
More > **Node Selector**	将 Pod 限制到带有特定标签的节点上（例如 `kubernetes.io/os: linux`）。 
More > **Affinity**	基于已有条件定义细粒度的调度规则。 **Affinity** 类型： • **Pod Affinity**：将新 Pod 调度到承载特定 Pod 的节点上（相同拓扑域）。 • **Pod Anti-affinity**：防止新 Pod 与特定 Pod 同节点共存。 强制模式： • `requiredDuringSchedulingIgnoredDuringExecution`：仅当规则满足时才调度 Pod。 • `preferredDuringSchedulingIgnoredDuringExecution`：优先选择满足规则的节点，但允许例外。 配置字段： • `topologyKey`：定义拓扑域的节点标签（默认值：`kubernetes.io/hostname`）。 • `labelSelector`：使用标签查询过滤目标 Pod。

3. 网络配置

- Kube-OVN

Parameters	Description
Bandwidth Limits	为 Pod 网络流量强制执行 QoS： • **Egress rate limit**：最大出站流量速率（例如 `10Mbps`）。 • **Ingress rate limit**：最大入站流量速率。
Subnet	从预定义的子网池中分配 IP。如果未指定，则使用 namespace 的默认子网。
Static IP Address	为 Pod 绑定持久 IP 地址： • 多个 Deployment 中的 Pod 可以声明同一个 IP，但同一时刻只能有一个 Pod 使用它。 • 关键：静态 IP 数量必须 ≥ Pod 副本数。

- Calico

Parameters	Description
Static IP Address	分配固定 IP，并强制唯一性： • 每个 IP 在集群中只能绑定一个 **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** 区域中，按照以下说明配置相关信息。

Parameters	Description
Resource Requests & Limits	• **Requests**：容器运行所需的最小 CPU/内存。 • **Limits**：容器运行期间允许使用的最大 CPU/内存。有关单位定义，请参见 Resource Units。

Parameters	Description
	Namespace overcommit ratio： • **Without overcommit ratio**： 如果存在 namespace 资源配额：容器 requests/limits 继承 namespace 默认值（可修改）。 无 namespace 配额：没有默认值；使用自定义 Request。 • **With overcommit ratio**： Requests 会自动计算为 `Limits / Overcommit ratio`（不可修改）。 约束： • Request ≤ Limit ≤ Namespace 配额上限。 • 更改 overcommit ratio 后需要重建 Pod 才会生效。 • 启用 overcommit ratio 后将禁用手动配置 request。 • 没有 namespace 配额 → 没有容器资源约束。
Extended Resources	配置集群中可用的扩展资源（例如 vGPU、pGPU）。
Volume Mounts	持久化存储配置。请参见 Storage Volume Mounting Instructions。 操作： • 已有 pod 卷：点击 **Add** • 没有 pod 卷：点击 **Add & Mount** 参数： • `mountPath`：容器文件系统路径（例如 `/data`） • `subPath`：卷内的相对文件/目录路径。 对于 `ConfigMap` / `Secret`：选择特定键 • `readOnly`：以只读方式挂载（默认值：读写） 请参见 Kubernetes Volumes↗。

Parameters	Description
Ports	暴露容器端口。 示例：暴露名为 `redis` 的 TCP 端口 `6379`。 字段： • `protocol`：TCP/UDP • `Port`：暴露的端口（例如 `6379`） • `name`：符合 DNS 规范的标识符（例如 `redis`）
Startup Commands & Arguments	覆盖默认 ENTRYPOINT/CMD： 示例 **1**：执行 `top -b` - **Command**：`["top", "-b"]` - 或者 Command：`["top"]`，Args：`["-b"]` 示例 **2**：输出 `$MESSAGE`： `/bin/sh -c "while true; do echo $(MESSAGE); sleep 10; done"` 请参见 Defining Commands↗。
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`） 说明：环境变量会覆盖 image/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**：检测容器健康状态（失败时重启） • **Readiness Probe**：检测服务可用性（失败时从 endpoints 中移除） 请参见 Health Check Parameters。

Parameters	Description
More > **Log Files**	配置日志路径： - 默认：收集 `stdout` - 文件模式：例如 `/var/log/*.log` 要求： • 存储驱动 `overlay2`：默认支持 • `devicemapper`：需要手动将 EmptyDir 挂载到日志目录 • Windows 节点：确保已挂载父目录（例如 `c:/a/b/c/*.log` 对应 `c:/a`）
More > **Exclude Log Files**	排除特定日志不进行采集（例如 `/var/log/aaa.log`）。
More > **Execute before Stopping**	在容器终止前执行命令。 示例：`echo "stop"` 说明：命令执行时间必须短于 pod 的 `terminationGracePeriodSeconds`。

2. 点击右上角的 **Add Container**，或者点击 **Add Init Container**。
   请参见 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. 允许删除的条件：
      - Pod 中有 >1 个应用容器且至少有 1 个 init container。
      - 单应用容器 Pod 不允许删除。
3. 点击 **Create**。

参考信息

存储卷挂载说明

Type	Purpose
Persistent Volume Claim	将现有的 PVC 绑定进来，以请求持久化存储。 注意：只有已绑定的 PVC（已关联 PV）才可选择。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（例如 `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（例如 `tls.crt`）
Ephemeral Volumes	集群提供的临时卷，具有以下特性： • 动态供给 • 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。请参见 Ephemeral Volumes
Empty Directory	同一 pod 内容器之间共享的临时存储： • pod 启动时在节点上创建 • pod 删除时一并删除 使用场景：容器间文件共享、临时数据存储。请参见 EmptyDir
Host Path	挂载主机目录（必须以 `/` 开头，例如 `/volumepath`）。

健康检查

- 健康检查 YAML 文件示例
- web console 中的健康检查配置参数

何时在 Applications 视图中使用 Deployment

当工作负载主要是无状态的，并且应通过副本 Pod 替换来更新，而不是依赖稳定标识或每个副本的持久状态时，请使用 Deployment。

通常在以下情况下，Deployment 更适合：

- 副本标识在替换后不需要保持稳定
- 工作负载应通过可互换的 Pod 进行横向扩展
- 应用交付更倾向于使用滚动更新行为

如果工作负载依赖稳定的序号标识或每个副本的持久存储，请改为查看 StatefulSets。

提交前的 Deployment 检查点

在 Applications 视图中创建或更新 Deployment 之前，请仔细检查以下内容：

- 所选项目、集群和 namespace 是否为预期的运行范围
- 副本数量是否与预期的可用性目标一致
- 所选 image 和拉取凭据在目标集群中是否有效
- 引用的 ConfigMap、Secret、PVC 或 ServiceAccount 是否已存在于同一 namespace 中
- 资源 requests、limits 和探针是否与应用真实的启动和健康行为一致
- 发布策略是否适合该工作负载和流量模式

管理 Deployments

使用 CLI 管理 Deployment

查看 Deployment

- 检查 Deployment 是否已创建。

```
kubectl get deployments
```

- 获取 Deployment 的详细信息。

```
kubectl describe deployments
```

更新 **Deployment**

按照以下步骤更新 Deployment：

1. 将 nginx Pod 更新为使用 nginx:1 .16.1 image。

```
kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.
1
```

或者使用以下命令：

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

另外，你也可以编辑 Deployment，并将 `.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 rollout 状态，运行：

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

运行 kubectl get rs 可以看到 Deployment 通过创建新的 ReplicaSet 并将其扩容到 3 个副本，同时将旧的 ReplicaSet 缩减到 0 个副本，从而完成了 Pod 更新。

```
kubectl get rs
```

运行 get pods 现在应该只会显示新的 Pod：

```
kubectl get pods
```

当更改 image 或 OCI Connector 时，也请检查 Pod 模板的 **Pull Secret**、`serviceAccountName` 和 token 有效期。更新后，请验证新的 ReplicaSet 和 Pod：检查 Connector 注解，确认实际 Pod image 已重写为 Connector proxy，等待 Pod 变为 `Running`，并查看 Events。

扩缩容 **Deployment**

你可以使用以下命令对 Deployment 进行扩缩容：

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

回滚 **Deployment**

- 假设你在更新 Deployment 时出现了拼写错误，把 image 名称写成了 `nginx:1.161`，而不是 `nginx:1.16.1`：

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

- rollout 会卡住。你可以通过检查 rollout 状态来验证这一点：

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

删除 **Deployment**

删除 Deployment 时，也会删除其托管的 ReplicaSet 和所有关联的 Pod。

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

使用 **web console** 管理 **Deployment**

查看 **Deployment**

你可以查看 Deployment，以获取应用的信息。

1. 在 **Container Platform** 中，导航到 **Workloads** > **Deployments**。
2. 找到要查看的 Deployment。
3. 点击 Deployment 名称，查看 **Details**、**Topology**、**Logs**、**Events**、**Monitoring** 等信息。

更新 **Deployment**

1. 在 **Container Platform** 中，导航到 **Workloads** > **Deployments**。
2. 找到要更新的 Deployment。
3. 在 **Actions** 下拉菜单中选择 **Update**，打开 Edit Deployment 页面。

更新操作会让 Deployment 保持在当前的 cluster 和 namespace 中。如果工作负载必须运行在不同的 cluster 或 namespace 中，请在目标范围内创建或应用一个单独的 Deployment，而不要把更新当作迁移操作。

在提交更新之前，请确认对 image、探针、环境变量、端口、卷挂载或资源设置的更改不会破坏正在运行的工作负载。

对 Pod 模板的更改可能会触发新的 rollout。成功的更新请求本身并不意味着 rollout 已经健康。只有当新的 Pod 修订版本正确调度、变为就绪，并且替换了之前的修订版本而没有导致工作负载不可用时，实际效果才会体现出来。

在 web console 中更改 image 或 OCI Connector 时，也请检查 **Pull Secret**、工作负载 ServiceAccount 和 token 有效期。确认最终生成的 Pod 使用了 Connector proxy image 并完成 rollout；仅更新成功并不能验证 image 拉取是否正常。

删除 **Deployment**

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

删除后，请验证 Deployment 已不再出现在列表中，并确认需要保留的独立 Service、ConfigMap、Secret 或 PVC 仍然存在。

使用 CLI 排障

当 Deployment 出现问题时，可以采用以下常见排障方法。

对于 OCI Connector image 拉取失败，请按以下顺序检查：

1. 检查 Pod 模板中的 `connectors.<baseDomain>/connectors` 注解。
2. 检查 `spec.imagePullSecrets` 以及工作负载 ServiceAccount 的 `imagePullSecrets`。
3. 检查实际 Pod image 是否已重写为 Connector proxy。
4. 检查 Pod 状态和 Events。

如果 Events 中显示原始 HTTP registry 出现 HTTPS 错误，则工作负载可能没有进入 Connector proxy 路径。请在更改 registry 运行时设置之前，检查 Connector proxy Secret、ServiceAccount 和 token。请参见 在 OCI Connector 中使用 ImagePullSecret。

检查 Deployment 状态

```
kubectl get deployment nginx-deployment
kubectl describe deployment nginx-deployment # View detailed events and s
tatus
```

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> # View logs for a specific co
ntainer
kubectl logs <pod-name> --previous          # View logs for the previously
terminated container
```

进入 **Pod** 进行调试

```
kubectl exec -it <pod-name> -- /bin/bash # Enter the container shell
```

检查健康配置

确保 livenessProbe 和 readinessProbe 已正确配置，并且应用的健康检查端点能够正常响应。
探针失败排障

检查资源限制

确保容器的 resource requests 和 limits 设置合理，并且容器没有因为资源不足而被杀死。

守护进程集

目录

- 理解 DaemonSets
- 创建 DaemonSets
  - 使用 CLI 创建 DaemonSet
    - 前提条件
    - YAML 文件示例
    - 通过 YAML 创建 DaemonSet
  - 使用 web console 创建 DaemonSet
    - 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
  - 使用新 web console 创建 DaemonSet
- 管理 DaemonSets
  - 使用 CLI 管理 DaemonSet
    - 查看 DaemonSet
    - 更新 DaemonSet
    - 删除 DaemonSet
  - 使用 web console 管理 DaemonSet
    - 查看 DaemonSet

更新 DaemonSet

删除 DaemonSet

理解 DaemonSets

请参阅 Kubernetes 官方文档：DaemonSets↗

DaemonSet 是一种 Kubernetes 控制器，可确保集群中的所有节点（或其子集）都恰好运行指定 Pod 的一个副本。与 Deployment 不同，DaemonSet 以节点为中心，而不是以应用为中心，因此非常适合部署集群级基础设施服务，例如日志收集器、监控代理或存储守护进程。

WARNING

DaemonSet 运行说明

1. 行为特征
   - **Pod** 分布：DaemonSet 会为每个满足条件且可调度的 **Node** 部署恰好一个 **Pod** 副本：
     - 为每个满足以下条件的可调度节点恰好部署一个 Pod 副本：
       - 匹配 `nodeSelector` 或 `nodeAffinity` 条件（如果已指定）。
       - 不处于 `NotReady` 状态。
       - 没有 `NoSchedule` 或 `NoExecute` **Taints**，除非在 **Pod Template** 中配置了相应的 **Tolerations**。
   - **Pod** 数量公式：DaemonSet 管理的 **Pod** 数量 等于 符合条件的 **Node** 数量。
   - 双角色节点处理：同时承担 控制平面 和 工作节点 角色的节点，只要可调度，无论其角色标签如何，都只会运行一个 DaemonSet 的 **Pod** 实例。
2. 关键约束（被排除的节点）
   - 明确标记为 `Unschedulable: true` 的节点（例如，通过 `kubectl cordon`）。
   - 处于 `NotReady` 状态的节点。
   - 存在不兼容 **Taints** 且未在 DaemonSet 的 **Pod Template** 中配置匹配 **Tolerations** 的节点。

创建 DaemonSets

使用 CLI 创建 DaemonSet

前提条件

- 确保已配置 `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: # defines how the DaemonSet identifies its managed Pods. Must match `template.metadata.label`s.
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template: # defines the Pod Template for the DaemonSet. Each Pod created by this DaemonSet will conform to this template
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations: # these tolerations are to have the daemonset runnable on control plane nodes, remove them if your control plane nodes should not run pods
        - 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
```

```
      # it may be desirable to set a high priority class to ensure that a
DaemonSet Pod
      # preempts running Pods
      # priorityClassName: important
      terminationGracePeriodSeconds: 30
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

通过 YAML 创建 DaemonSet

```
# Step 1: To create the DaemonSet defined in *example-daemonSet.yaml*, ex
ecute the following command
kubectl apply -f example-daemonSet.yaml

# Step 2: To verify the creation and status of your DaemonSet and its ass
ociated Pods:
kubectl get daemonset fluentd-elasticsearch # View DaemonSet
kubectl get pods -l name=fluentd-elasticsearch -o wide # Check Pods manag
ed by this DaemonSet on specific nodes
```

使用 web console 创建 DaemonSet

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。如果未找到所需的镜像仓库，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从该仓库拉取镜像。
- 如果镜像仓库需要认证，你需要配置相应的 image pull secret。更多信息，请参阅 为 ServiceAccount 添加 ImagePullSecrets。
- 通过 OCI Connector 选择镜像时，还需要为 Connector 代理选择一个 image pull Secret，或者确认工作负载的 ServiceAccount 已经引用了该 Secret。请参阅 在 OCI Connector 中使用 ImagePullSecret。

操作步骤 - 配置基本信息

1. 在 **Container Platform** 左侧导航栏中，进入 **Workloads** > **DaemonSets**。
2. 点击 **Create DaemonSet**。
3. 选择或输入一个镜像，然后点击 **Confirm**。

> **INFO**
>
> 注意：使用 web console 中集成的镜像仓库镜像时，可以通过 **Already Integrated** 进行筛选。**Integration Project Name**，例如镜像（registry-projectname），其中在此 web console 中包含项目名称 projectname，在镜像仓库中包含项目名称 containers。

在 **Basic Info** 部分，为 DaemonSet 工作负载配置声明式参数：

参数	说明
More > **Update Strategy**	配置 DaemonSet Pod 的 `rollingUpdate` 策略，以实现零停机更新。 **Max unavailable** (`maxUnavailable`)：更新期间可临时不可用的 Pod 最大数量。支持绝对值（例如 1）或百分比（例如 10%）。 示例：如果有 10 个节点，且 `maxUnavailable` 为 10%，则 floor(10 * 0.1) = 1 个 Pod 可以不可用。 注意： • 默认值：如果未显式设置，`maxSurge` 默认为 0，`maxUnavailable` 默认为 1（如果 `maxUnavailable` 指定为百分比，则默认为 10%）。 • 未运行的 **Pod**：处于 `Pending` 或 `CrashLoopBackOff` 等状态的 Pod 视为不可用。 • 同时限制：`maxSurge` 和 `maxUnavailable` 不能同时为 0 或 0%。如果百分比值换算后两个参数都为 0，Kubernetes 会强制设置 `maxUnavailable=1` 以确保更新能够继续进行。

操作步骤 - 配置 **Pod**

Pod 部分请参阅 Deployment - 配置 Pod

操作步骤 - 配置容器

Containers 部分请参阅 Deployment - 配置容器

操作步骤 - 创建

点击 **Create**。

点击 **Create** 后，DaemonSet 将：

- ✅ 自动将 Pod 副本部署到所有符合条件的节点，这些节点满足：
  - `nodeSelector` 条件（如果已定义）。
  - `tolerations` 配置（允许在带污点节点上调度）。
  - 节点处于 `Ready` 状态且 `Schedulable: true`。
- ❌ 被排除的节点：
  - 带有 `NoSchedule` 污点的节点（除非已显式容忍）。
  - 手动封锁的节点（`kubectl cordon`）。
  - 处于 `NotReady` 或 `Unschedulable` 状态的节点。

使用新 web console 创建 DaemonSet

新的 web console 可以从不同视图暴露 DaemonSet 创建入口。根据你是从项目范围还是从具体集群进行操作，选择相应入口。

- **Applications** 视图：从当前项目开始。打开 **Workloads** > **DaemonSets**，选择目标 **Cluster** 和 **Namespace**，然后选择会生成 DaemonSet 的工作负载模型或创建流程。
- **Clusters** 视图：从具体集群开始。先打开目标集群，进入目标命名空间，然后导航到 **Workloads** > **DaemonSets**，点击 **Create DaemonSet**。

管理 DaemonSets

使用 CLI 管理 DaemonSet

查看 DaemonSet

- 获取命名空间中所有 DaemonSet 的摘要。

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

- 获取特定 DaemonSet 的详细信息，包括事件和 Pod 状态。

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

更新 DaemonSet

当你修改 DaemonSet 的 **Pod Template** 时（例如更改容器镜像或添加卷挂载），Kubernetes 默认会自动执行滚动更新（如果 `updateStrategy.type` 为 `RollingUpdate`，这是默认值）。

在更改镜像或 OCI Connector 之前，请确认 Connector 代理的 image pull Secret 可以直接供 Pod 使用，或可通过其 ServiceAccount 使用。目标节点上的容器运行时必须能够访问 Connector 代理地址。在更新过程中，请检查每个节点上 Pod 的 Connector annotation、重写后的代理镜像、状态以及 Events。

- 首先，编辑 YAML 文件（例如 `example-daemonset.yaml`）并加入所需更改，然后应用：

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

- 你可以监控滚动更新的进度：

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

删除 DaemonSet

删除 DaemonSet 及其管理的所有 Pod：

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

使用 **web console** 管理 **DaemonSet**

查看 **DaemonSet**

1. 进入 **Container Platform**，然后导航到 **Workloads** > **DaemonSets**。
2. 找到要查看的 DaemonSet。
3. 点击 DaemonSet 名称查看 **Details**、**Topology**、**Logs**、**Events**、**Monitoring** 等信息。

更新 **DaemonSet**

1. 进入 **Container Platform**，然后导航到 **Workloads** > **DaemonSets**。
2. 找到要更新的 DaemonSet。
3. 在 **Actions** 下拉菜单中选择 **Update**，进入 Edit DaemonSet 页面，你可以更新 `Replicas`、`image`、`updateStrategy` 等。

更改镜像或 OCI Connector 时，还应检查 Connector 代理的 image pull Secret、工作负载的 ServiceAccount 以及令牌有效性。在滚动更新期间，请验证每个目标节点上的实际 Pod 镜像和 Events。

删除 **DaemonSet**

1. 进入 **Container Platform**，然后导航到 **Workloads** > **DaemonSets**。
2. 找到要删除的 DaemonSet。
3. 在 **Actions** 下拉菜单中，点击操作列中的 **Delete** 按钮并确认。

有状态副本集

目录

- 了解有状态副本集
- 创建有状态副本集
  - 使用 CLI 创建有状态副本集
    - 前提条件
    - YAML 文件示例
    - 通过 YAML 创建有状态副本集
  - 使用 web 控制台创建有状态副本集
    - 前提条件
  - 使用新的 web 控制台创建有状态副本集
    - 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
- 何时在 Applications 视图中使用有状态副本集
- 有状态工作负载审查要点
- 健康检查
- 管理有状态副本集
  - 使用 CLI 管理有状态副本集
    - 查看有状态副本集
    - 扩缩有状态副本集

更新有状态副本集（Rolling Update）
删除有状态副本集
使用 web 控制台管理有状态副本集
查看有状态副本集
更新有状态副本集
删除有状态副本集

了解有状态副本集

参考官方 Kubernetes 文档：StatefulSets↗

StatefulSet 是一个 Kubernetes 工作负载 API 对象，旨在通过以下能力管理有状态应用：

- 稳定的网络标识：DNS 主机名 `<statefulset-name>-<ordinal>.<service-name>.ns.svc.cluster.local`。
- 稳定的持久存储：通过 `volumeClaimTemplates` 提供。
- 有序部署/扩缩容：按顺序创建/删除 Pod：Pod-0 → Pod-1 → Pod-N。
- 有序滚动更新：按逆序序号更新 Pod：Pod-N → Pod-0。

在分布式系统中，可以将多个 StatefulSet 作为独立组件部署，以提供专用的有状态服务（例如 *Kafka brokers*、*MongoDB shards*）。

创建有状态副本集

使用 CLI 创建有状态副本集

前提条件

- 确保你已配置 `kubectl` 并已连接到集群。

YAML 文件示例

```yaml
# example-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: 'nginx' # this headless Service is responsible for the net
work identity of the Pods
  replicas: 3 # defines the desired number of Pod replicas (default: 1)
  minReadySeconds: 10 # by default is 0
  template: # defines the Pod template for the StatefulSet
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    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: # defines PersistentVolumeClaim (PVC) templates.
Each Pod gets a unique PersistentVolume (PV) dynamically provisioned base
d on these templates.
    - metadata:
        name: www
      spec:
        accessModes: ['ReadWriteOnce']
        storageClassName: 'my-storage-class'
        resources:
          requests:
            storage: 1Gi
---
# example-service.yaml
apiVersion: v1
kind: Service
```

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

通过 **YAML** 创建有状态副本集

```
# Step 1: To create the StatefulSet defined in *example-statefulset.yaml
*, execute the following command
kubectl apply -f example-statefulset.yaml

# Step 2: To verify the creation and status of your StatefulSet and its a
ssociated Pods and PVCs:
kubectl get statefulset web # View StatefulSet
kubectl get pods -l app=nginx # Check Pods managed by this StatefulSet
kubectl get pvc -l app=nginx # Check PVCs created by volumeClaimTemplates
```

使用 **web** 控制台创建有状态副本集

前提条件

获取镜像地址。镜像来源可以是平台管理员通过 toolchain 集成到平台中的镜像仓库，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到你的项目中，你可以直接使用其中的镜像。如果未找到所需的镜像仓库，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从该仓库拉取镜像。
- 如果镜像仓库需要认证，你需要配置相应的 image pull Secret。更多信息，请参见 将 ImagePullSecrets 添加到 ServiceAccount。
- 通过 OCI Connector 选择镜像时，还需要为 Connector proxy 选择一个 image pull Secret，或者确认工作负载的 ServiceAccount 已经引用了该 Secret。参见 在 OCI Connector 中使用

ImagePullSecret。

使用新的 web 控制台创建有状态副本集

新的 web 控制台可以从不同视图暴露 StatefulSet 的创建入口。请根据你是从项目范围还是从具体集群进行操作，选择相应入口。

- **Applications view**：从当前项目开始。打开 **Workloads** > **StatefulSets**，选择目标 **Cluster** 和 **Namespace**，然后选择会生成 StatefulSet 的工作负载模型或创建流程。
- **Clusters view**：从具体集群开始。先打开目标集群，进入目标命名空间，然后导航到 **Workloads** > **StatefulSets**，点击 **Create StatefulSet**。

操作步骤 - 配置基本信息

1. 在 **Container Platform** 左侧边栏中，导航到 **Workloads** > **StatefulSets**。
2. 点击 **Create StatefulSet**。
3. **Select** 或 **Input** 一个镜像，然后点击 **Confirm**。

> **INFO**
>
> 注意：使用 web 控制台中集成的镜像仓库中的镜像时，可以通过 **Already Integrated** 进行筛选。**Integration Project Name** 例如镜像 `images (registry-projectname)`，其中项目名 `projectname` 是 web 控制台中的项目名，`containers` 是镜像仓库中的项目名。

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

参数	描述
Replicas	定义 StatefulSet 中所需的 Pod 副本数量（默认值：1）。请根据工作负载需求和预期请求量进行调整。
Update Strategy	控制 StatefulSet 滚动更新期间的分阶段更新。`RollingUpdate` 策略为默认值，且推荐使用。 **Partition** 值：用于 Pod 更新的序号阈值。 • 索引 ≥ `partition` 的 Pod 会立即更新。 • 索引 < `partition` 的 Pod 将保留之前的规格。

参数	描述
	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，它支持为每个 Pod 动态提供持久存储。StatefulSet 中的每个 Pod 副本都会根据预定义模板自动获得自己的专用 PersistentVolumeClaim (PVC)。
	• 1. 动态创建 **PVC**：自动为每个 Pod 创建唯一的 PVC，命名模式为：`<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)。
	• 3. **Storage Class**：通过 storageClassName 指定存储后端。如果未指定，则使用集群默认 StorageClass。支持各种云端/本地部署存储类型（例如 SSD、HDD）。
	• 4. 容量：通过 resources.requests.storage 配置存储容量。示例：1Gi。如果 StorageClass 启用了动态卷扩容，则支持动态扩容。

操作步骤 - 配置 **Pod**

Pod 部分，请参阅 Deployment - Configure Pod

操作步骤 - 配置容器

Containers 部分，请参阅 Deployment - Configure Containers

操作步骤 - 创建

点击 **Create**。

何时在 **Applications** 视图中使用有状态副本集

当工作负载依赖于每个副本的标识、稳定命名、有序生命周期行为，或者需要与特定副本保持关联的持久存储时，应使用有状态副本集。

通常在以下场景中，有状态副本集更适合：

- 每个副本都应保留稳定的序号标识
- 工作负载依赖稳定的网络命名或相关的 headless Service
- 每个副本都需要自己的 persistent volume claim
- 应用对启动、关闭或更新顺序有要求

如果工作负载可以视为可互换的无状态副本，请改为查看 Deployments。

有状态工作负载审查要点

在创建或更新 StatefulSet 之前，请仔细检查以下内容：

- 所选项目、集群和命名空间是否为预期的运行范围
- 每个副本是否都需要稳定标识
- 每个副本是否都需要持久存储
- 工作负载是否依赖有序的启动、关闭或扩缩容行为
- 引用的存储设置在目标环境中是否可用
- 配置更改是否可能影响数据连续性或副本替换行为
- 工作负载是否还通过相关 Service 依赖稳定的网络标识

健康检查

- 健康检查 YAML 文件示例
- web 控制台中的健康检查配置参数

管理有状态副本集

使用 **CLI** 管理有状态副本集

查看有状态副本集

你可以查看 StatefulSet 以获取应用信息。

- 检查 StatefulSet 是否已创建。

```
kubectl get statefulsets
```

- 获取 StatefulSet 的详细信息。

```
kubectl describe statefulsets
```

扩缩有状态副本集

- 修改现有 StatefulSet 的副本数量：

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

- 示例：

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

更新有状态副本集（**Rolling Update**）

当你修改 StatefulSet 的 Pod 模板时（例如更改容器镜像），Kubernetes 默认会执行滚动更新（前提是 updateStrategy 设置为 RollingUpdate，而这也是默认值）。

在更改镜像或 OCI Connector 之前，请确认每个新的 Pod 版本都能直接或通过其 ServiceAccount 获取有效的 Connector proxy image pull Secret。在有序更新期间，请检查每个 Pod 的 Connector 注解、重写后的 proxy 镜像、状态和 Events。如果某个序号 Pod 进入

`ImagePullBackOff`，请停止 rollout，并在继续之前检查 Secret、ServiceAccount 和 token 的有效性。

- 首先，编辑 YAML 文件（例如 example-statefulset.yaml）并应用所需更改，然后执行：

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

- 然后，你可以监控滚动更新的进度：

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

删除有状态副本集

要删除 StatefulSet 及其关联的 Pod：

```
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。若要同时删除 PVC，则必须显式执行：

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

另外，如果你的 `volumeClaimTemplates` 使用的 `StorageClass` 的 `reclaimPolicy` 为 `Delete`，则在 PVC 被删除时，PVs 及其底层存储也会自动删除。

使用 **web** 控制台管理有状态副本集

查看有状态副本集

1. 在 **Container Platform** 中，导航到 **Workloads** > **StatefulSets**。
2. 找到你要查看的 StatefulSet。
3. 点击 StatefulSet 名称以查看 **Details**、**Topology**、**Logs**、**Events**、**Monitoring** 等信息。

更新有状态副本集

1. 在 **Container Platform** 中，导航到 **Workloads** > **StatefulSets**。
2. 找到你要更新的 StatefulSet。
3. 在 **Actions** 下拉菜单中选择 **Update**，进入 Edit StatefulSet 页面，你可以更新 `Replicas`、`image`、`updateStrategy` 等参数。

更改镜像或 OCI Connector 时，还要检查 Connector proxy image pull Secret、工作负载的 ServiceAccount 以及 token 的有效性。观察有序的 Pod 替换过程，并验证每个新 Pod 的实际镜像重写情况和 Events。

删除有状态副本集

1. 在 **Container Platform** 中，导航到 **Workloads** > **StatefulSets**。
2. 找到你要删除的 StatefulSet。
3. 在 **Actions** 下拉菜单中，点击操作列中的 **Delete** 按钮并确认。

StatefulSet 的删除和缩容可能带来与无状态工作负载不同的存储影响。在删除或缩容之前，请确认被移除的副本不再持有必需数据或集群成员资格。

定时任务

目录

了解定时任务
创建定时任务
使用 CLI 创建 CronJob
前提条件
YAML 文件示例
通过 YAML 创建 CronJob
使用 Web 控制台创建 CronJob
前提条件
使用新版 Web 控制台创建 CronJob
Applications 和 Clusters 视图中的创建入口差异
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
验证 Job 详情：
监控执行状态
删除定时任务

使用 Web 控制台删除 CronJob

使用 CLI 删除 CronJob

了解定时任务

请参阅 Kubernetes 官方文档：

- 定时任务 ↗
- 使用 CronJob 运行自动化任务 ↗

CronJob 定义运行至完成后停止的任务。它们允许你按照计划多次运行同一个 Job。

CronJob 是 Kubernetes 中的一种工作负载控制器。你可以通过 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、定时清理或定时发送电子邮件。

创建定时任务

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 CronJob

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

使用 Web 控制台创建 CronJob

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，也可以来自第三方镜像仓库。

- 对于来自集成镜像仓库的镜像，管理员通常会将镜像仓库分配给你的项目，使你能够使用其中的镜像。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，请确保当前集群可以直接从该仓库拉取镜像。
- 如果镜像仓库需要身份验证，则需要配置相应的镜像拉取 Secret。有关更多信息，请参阅将 ImagePullSecrets 添加到 ServiceAccount。

- 通过 OCI Connector 选择镜像时，还要为 Connector proxy 选择镜像拉取 Secret，或确认工作负载的 ServiceAccount 已引用该 Secret。请参阅将 ImagePullSecret 与 OCI Connector 配合使用。

使用新版 **Web** 控制台创建 **CronJob**

Applications 和 **Clusters** 视图中的创建入口差异

新版 Web 控制台可以从不同视图进入 CronJob 创建页面。请根据你是在项目范围内操作，还是从具体集群开始操作，选择相应入口。

- **Applications** 视图：从当前项目开始。打开 **Workloads** > **CronJobs**，选择目标 **Cluster** 和 **Namespace**，然后选择会创建 CronJob 的创建流程。
- **Clusters** 视图：从具体集群开始。先打开目标集群，进入目标命名空间，然后导航到 **Workloads** > **CronJobs**，点击 **Create CronJob**。

操作步骤 **-** 配置基本信息

1. 在 **Container Platform** 中，导航到左侧边栏的 **Workloads** > **CronJobs**。
2. 点击 **Create CronJob**。
3. **Select** 或 **Input** 镜像，然后点击 **Confirm**。
   注意：只有使用平台集成镜像仓库中的镜像时，才可进行镜像筛选。例如，类似 containers (registry-projectname) 的集成项目名称表示平台项目名称为 projectname，镜像仓库项目名称为 containers。
   对于 OCI Connector 镜像，请确认 Pod 模板已选择 Connector proxy 镜像拉取 Secret，或该 Secret 已从其 ServiceAccount 继承。创建 CronJob 对象并不能证明后续 Job 能够拉取镜像；Job Pod 启动时令牌仍必须有效。
4. 在 **Cron Configuration** 部分，配置任务执行方式及相关参数。
   Execute Type：
   - **Manual**：手动执行要求每次运行任务时显式手动触发。
   - **Scheduled**：计划执行要求配置以下调度参数：

参数	描述
Schedule	使用 Crontab 语法↗ 定义 cron 计划。CronJob 控制器根据选定的时区计算下一次执行时间。 注意： • 对于 Kubernetes 集群 < v1.25：不支持选择时区；计划 MUST 使用 UTC。 • 对于 Kubernetes 集群 ≥ v1.25：支持基于时区的调度（默认：用户的本地时区）。
Concurrency Policy	指定如何处理并发 Job 执行（根据 K8s 规范↗ 使用 `Allow`、`Forbid` 或 `Replace`）。

Job History Retention：

- 设置已完成 Job 的保留限制：
  - **History Limits**：成功 Job 的历史记录限制（默认：20）
  - **Failed Jobs**：失败 Job 的历史记录限制（默认：20）
- 超出保留限制时，系统会首先进行垃圾回收，删除最早的 Job。

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

参数	描述
Job Type	选择 Job 完成模式（根据 K8s Job 模式↗ 使用 `Non-parallel`、`Parallel with fixed completion count` 或 `Indexed Job`）。
Backoff Limit	设置将 Job 标记为失败前的最大重试次数。

操作步骤 - 配置 **Pod**

- **Pod** 部分请参阅Deployment - 配置 Pod

操作步骤 - 配置容器

- **Container** 部分请参阅 Deployment - 配置容器

创建

- 点击 **Create**。

立即执行

定位 **CronJob** 资源

- **Web** 控制台：在 **Container Platform** 中，导航到左侧边栏的 **Workloads** > **CronJobs**。
- **CLI**：

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

发起临时执行

- **Web** 控制台：**Execute Immediately**

  1. 点击 CronJob 列表右侧的垂直省略号（⋮）。
  2. 点击 **Execute Immediately**。（或者在 CronJob 详情页中，点击右上角的 Actions，然后选择 **Execute Immediately**。）

- **CLI**：

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

验证 **Job** 详情：

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

对于使用 OCI Connector 的 Job，请检查生成的 Pod，而不要只依赖 CronJob 或 Job 状态。验证 Connector 注解、`imagePullSecrets` 或 ServiceAccount 继承关系、重写后的 Connector proxy 镜像、Pod 的 `Running` 或 `Completed` 状态以及 Events。将过期令牌视为身份验证材料问题，并在下一次执行前轮换或重新创建 Secret。

监控执行状态

状态	描述
Pending	Job 已创建，但尚未进行调度。
Running	Job Pod 正在执行。
Succeeded	与 Job 关联的所有 Pod 均已成功完成（退出代码为 0）。
Failed	至少一个与 Job 关联的 Pod 未成功终止（退出代码非零）。

更新 CronJob 镜像或 OCI Connector 时，请检查 Pod 模板的 Connector proxy 镜像拉取 Secret 或 ServiceAccount，以及令牌的有效性。还要在下一个生成的 Job Pod 上再次验证这些值，因为 API 接受更新并不能证明计划 Pod 能够拉取其镜像。

删除定时任务

使用 **Web** 控制台删除 **CronJob**

1. 在 **Container Platform** 中，导航到 **Workloads** > **CronJobs**。
2. 找到要删除的 CronJob。
3. 在 **Actions** 下拉菜单中，点击 **Delete** 按钮并确认。

使用 **CLI** 删除 **CronJob**

```
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 Pod
YAML 文件示例
使用 CLI 管理 Pod
调整 Pod 资源
查看 Pod
查看 Pod 日志
在 Pod 中执行命令
转发到 Pod 的端口
删除 Pod
使用 Web 控制台管理 Pod
查看 Pod
操作步骤
Pod 参数
删除 Pod
使用场景
操作步骤

理解 Pod

请参阅 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。**Pod**（如鲸鱼群中的 pod 或豌豆荚）是一组一个或多个容器（例如 containers），它们共享存储和网络资源，并包含用于描述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**）构建的基础组件。

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 con
tainer.
        requests:
          cpu: '100m'
          memory: '128Mi'
        limits:
          cpu: '200m'
          memory: '256Mi'
```

使用 **CLI** 管理 **Pod**

虽然 Pods 通常由更高级控制器管理，但直接对 Pods 执行 kubectl 操作对于故障排查、检查和临时任务非常有用。

调整 Pod 资源

要通过 Kubernetes `resize` 子资源更新正在运行的 Pod 的 CPU 或内存 requests 和 limits，请参阅 无需中断 Pod 即可调整 Pod 资源级别。

查看 Pod

- 列出当前 namespace 中的所有 Pods：

```
kubectl get pods
```

- 列出所有 namespace 中的所有 Pods：

```
kubectl get pods --all-namespaces
# Or a shorter version:
kubectl get pods -A
```

- 获取特定 Pod 的详细信息：

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

# Example
kubectl describe pod my-nginx-pod -n default
```

查看 Pod 日志

- 流式获取 Pod 中某个容器的日志（适用于调试）：

```
kubectl logs <pod-name> -n <namespace>
```

- 如果 Pod 有多个容器，必须指定容器名称：

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

- 跟踪日志（随着新日志出现而持续输出）：

```
kubectl logs -f <pod-name> -n <namespace>
```

在 Pod 中执行命令

在 Pod 中的特定容器内执行命令（适用于调试，例如访问 shell）：

```
kubectl exec -it <pod-name> -n <namespace> -- <command>

# Example (to get a shell):
kubectl exec -it my-nginx-pod -n default -- /bin/bash
```

转发到 Pod 的端口

将本地端口转发到 Pod 上的某个端口，使您可以从本地机器直接访问运行在 Pod 内的服务（适用于测试，或无需对外暴露服务即可直接访问）：

```
kubectl port-forward <pod-name> <local-port>:<pod-port> -n <namespace>

#Example
kubectl port-forward my-nginx-pod 8080:80 -n default
```

运行此命令后，您可以在 Web 浏览器中访问 localhost:8080 ，查看运行在 my-nginx-pod 中的 Nginx Web server。

删除 Pod

- 删除特定 Pod：

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

# Example
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 label selector 删除 Pods（例如，删除所有带有 label app=nginx 的 Pods）：

```
kubectl delete pods -l app=nginx -n <namespace>
```

使用 **Web** 控制台管理 **Pod**

查看 **Pod**

平台界面提供了有关 pods 的各种信息，便于快速参考。

操作步骤

1. 在 **Container Platform** 中，点击左侧边栏的 **Workloads** > **Pods**。
2. 找到您要查看的 Pod。
3. 点击 deployment 名称以查看 **Details**、**YAML**、**Configuration**、**Logs**、**Events**、**Monitoring** 等内容。

Pod 参数

下面是一些参数说明：

参数	说明
Resource Requests & Limits	**Resource Requests** 和 **Limits** 定义了 Pod 内 Containers 的 CPU 和内存消耗边界，随后汇总形成 Pod 的整体资源画像。这些值对于 Kubernetes 的 scheduler 将 Pods 高效调度到 Nodes 上，以及 kubelet 执行资源治理都至关重要。 • **Requests**：容器被 调度 并运行所需的最低 CPU/内存保障值。Kubernetes scheduler 使用该值来决定 Pod 可以运行在哪个 **Node** 上。 • **Limits**：容器在执行期间允许消耗的最大 CPU/内存值。超过 CPU limit 会导致节流，而超过 memory limit 则会导致容器被终止（Out Of Memory - OOM Killed）。

参数	说明
	有关详细单位定义（例如，`m` 表示 milliCPU，`Mi` 表示 mebibytes），请参阅 资源单位。 **Pod** 级资源计算逻辑 Pod 的有效 CPU 和内存 Requests 及 Limits 值由其各个容器规格的求和和最大值推导而来。Pod 级 Requests 和 Limits 的计算方式类似；本文档以 Limit 值为例说明其逻辑。当一个 Pod 仅包含标准容器（业务容器）时：Pod 的有效 CPU/Memory Limit 值等于 Pod 内所有容器 CPU/Memory Limit 值的总和。 示例：如果一个 Pod 包含两个容器，其 CPU/Memory Limits 分别为 100m/100Mi 和 50m/200Mi，则该 Pod 聚合后的 CPU/Memory Limit 为 150m/300Mi。当一个 Pod 同时包含 initContainers 和标准容器时：Pod 的 CPU/Memory Limit 值计算步骤如下： • 1. 确定所有 initContainers 中最大的 CPU/Memory Limit 值。 • 2. 计算所有标准容器 CPU/Memory Limit 值之和。 • 3. 比较步骤 1 和步骤 2 的结果。Pod 的综合 CPU/Memory Limit 将取 CPU 值中的较大者（initContainers 最大值与 containers 总和之间的较大者），以及 Memory 值中的较大者（initContainers 最大值与 containers 总和之间的较大者）。 计算示例：如果一个 Pod 包含两个 initContainers，其 CPU/Memory Limits 分别为 100m/200Mi 和 200m/100Mi，则 initContainers 的最大有效 CPU/Memory Limit 为 200m/200Mi。同时，如果该 Pod 还包含两个标准容器，其 CPU/Memory Limits 分别为 100m/100Mi 和 50m/200Mi，则标准容器的总聚合 Limit 为 150m/300Mi。因此，该 Pod 的综合 CPU/Memory Limit 为 CPU 取 max(200m, 150m)，Memory 取 max(200Mi, 300Mi)，结果为 200m/300Mi。
Source	管理此 Pod 生命周期的 Kubernetes workload controller。这包括 **Deployments**、**StatefulSets**、**DaemonSets**、**Jobs**。
Restart	自 **Pod** 启动以来，**Pod** 内的 Container 已重启的次数。较高的重启次数通常表示应用程序或其环境存在问题。
Node	当前调度并运行该 Pod 的 Kubernetes Node 名称。
Service Account	Service Account 是 Kubernetes 对象，它为运行在 Pod 内的进程和服务提供身份，使其能够向 Kubernetes APIServer 进行身份验证并访问它。此字段通常仅

参数	说明
	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，从而允许查看 Service Account 的 YAML 定义。

删除 **Pod**

删除 pods 可能会影响计算组件的运行；请谨慎操作。

使用场景

- 及时将 pods 恢复到期望状态：如果某个 pods 处于影响业务运行的状态，例如 `Pending` 或 `CrashLoopBackOff`，在处理完错误信息后手动删除该 pods，可以帮助其快速恢复到期望状态，例如 `Running`。此时，被删除的 pods 会在当前节点上重新创建或重新调度。
- 运维清理资源：某些 podss 会达到一个不再变化的指定阶段，这些组通常会大量积累，给其他 podss 的管理带来复杂性。需要清理的 podss 可能包括由于节点资源不足而处于 `Evicted` 状态的 podss，或者由周期性计划任务触发而处于 `Completed` 状态的 podss。在这种情况下，被删除的 podss 将不再存在。
  注意：对于计划任务，如果您需要查看每次任务执行的日志，不建议删除对应处于 `Completed` 状态的 podss。

操作步骤

1. 进入 **Container Platform**。
2. 在左侧导航栏中，点击 **Workloads** > **Pods**。
3. （单独删除）点击要删除的 pods 右侧的⋮> **Delete**，然后确认。
4. （批量删除）选择要删除的 podss，点击列表上方的 **Delete**，然后确认。

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。

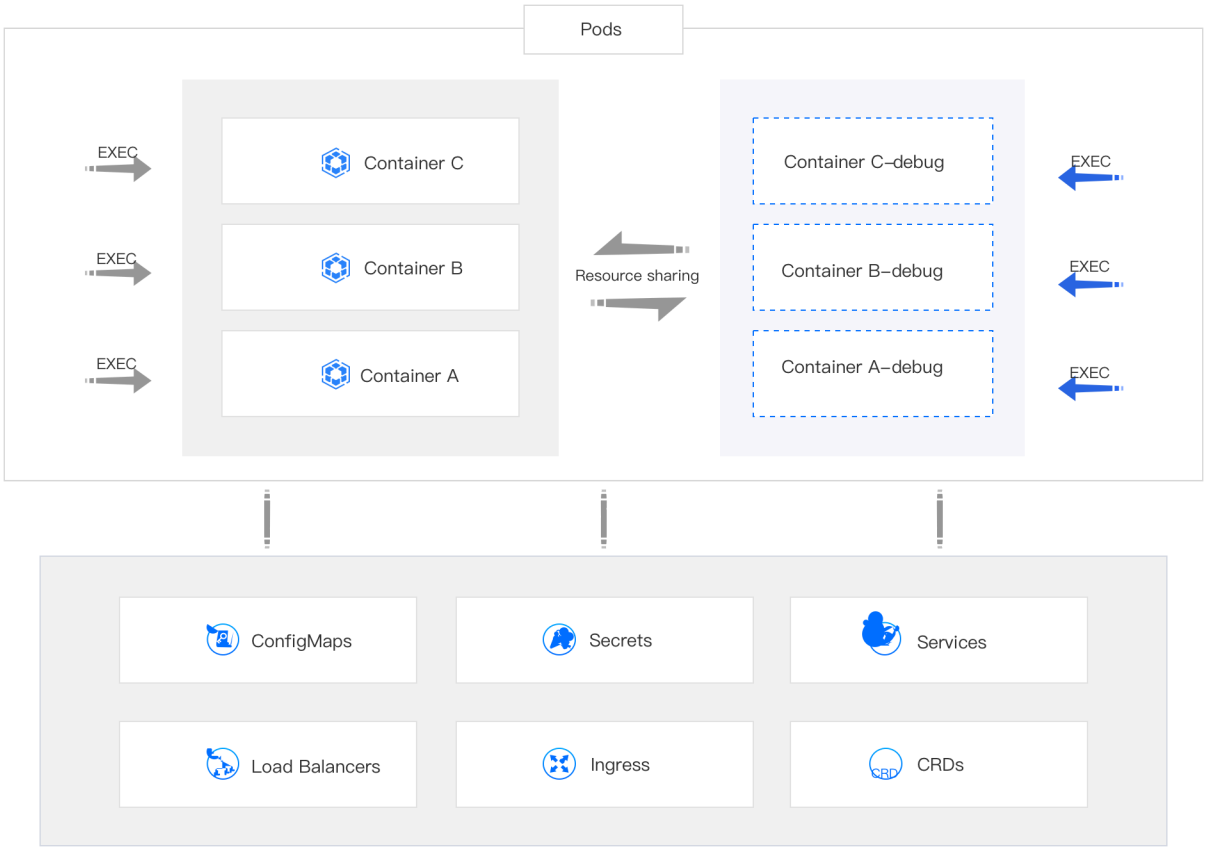
虽然通常可以使用 kubectl exec 直接在运行中的容器内执行命令，但许多生产环境的容器镜像故意保持精简，可能缺少关键的调试工具（如 bash、net-tools、tcpdump），以减小镜像大小和攻击面。Ephemeral Containers 通过提供预配置的丰富调试工具环境，解决了这一限制，非常适合以下场景：

- 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> -- <comman
d>
# -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。

在不影响 Pod 的情况下调整 Pod 资源级别

当你需要在不重新创建 Pod 的情况下，为正在运行的 Pod 调整 CPU 或内存的 requests 和 limits 时，可以使用原地 Pod 调整大小。此任务使用 Kubernetes Pod `resize` 子资源。

目录

前提条件

关于原地 Pod 调整大小

配置原地 Pod 调整大小

调整正在运行的 Pod

验证重启行为

检查调整大小状态

限制

其他资源

前提条件

确保目标 Kubernetes 集群支持 Pod `resize` 子资源。请使用 Kubernetes v1.33 或更高版本，并在控制平面和节点上启用 `InPlacePodVerticalScaling` 特性门控。在 Kubernetes v1.35 中，此特性已稳定并默认启用。

关于原地 **Pod** 调整大小

原地 Pod 调整大小可让你更改正在运行的 Pod 中容器的 CPU 和内存资源 requests 和 limits。标准的资源变更通常需要重新创建 Pod，这可能会中断应用程序或丢失运行时状态。原地 Pod 调整大小会通过 Kubernetes Pod `resize` 子资源应用受支持的资源变更。

你可以通过在 Pod 规范中配置 `resizePolicy` 来控制容器在调整大小期间是否重启。例如，你可以允许 CPU 变更而不重启容器，并要求在内存变更时重启容器。

配置原地 **Pod** 调整大小

在调整正在运行的 Pod 之前配置 `resizePolicy`。你不能在现有 Pod 上添加或修改 `resizePolicy`。如果该 Pod 由 Deployment 或 StatefulSet 等控制器对象管理，请在控制器对象的 Pod 模板中添加或更新 `resizePolicy`。

deployment-resize-policy.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: resize-demo
spec:
  selector:
    matchLabels:
      app: resize-demo
  template:
    metadata:
      labels:
        app: resize-demo
    spec:
      containers:
        - name: app
          image: nginx:latest
          resizePolicy:
            - resourceName: cpu
              restartPolicy: NotRequired
            - resourceName: memory
              restartPolicy: RestartContainer
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 500m
              memory: 512Mi
```

对于 CPU 和内存资源，请使用以下 `restartPolicy` 值之一：

值	说明
`NotRequired`	在不重启容器的情况下应用受支持的资源变更。
`RestartContainer`	应用受支持的资源变更，并重启容器。

当使用 `NotRequired` 或未配置内存 `resizePolicy` 降低内存 limits 时，kubelet 会尽力应用该变更，但不保证容器不会被 out-of-memory kill。如果当前内存使用量高于请求的限制，调整

可能会被跳过并保持进行中。对于需要更可预测行为的内存变更，请使用 `RestartContainer`。

调整正在运行的 Pod

编辑 Pod 资源时，请使用 `resize` 子资源。请从可以访问目标集群的工作站运行这些命令。

> **Tip**
>
> 请使用 `kubectl` v1.32.0 或更高版本，或其他支持 Pod `resize` 子资源的 API 客户端。旧版 `kubectl` 客户端可能会因为 `invalid subresource value: "resize"` 等错误而拒绝 `--subresource=resize`，或者即使 Kubernetes API server 支持 `pods/resize`，也可能不会为某些命令提供 `--subresource` 标志。

交互式编辑 Pod 资源：

```
kubectl edit pod <pod_name> -n <namespace> --subresource=resize
```

通过 `resize` 子资源使用 server-side apply 应用更新后的 Pod 清单：

```
kubectl apply --server-side -f <file_name>.yaml --subresource=resize
```

直接 patch 目标容器资源：

```
kubectl patch pod <pod_name> \
  -n <namespace> \
  --subresource=resize \
  --patch '{"spec":{"containers":[{"name":"<container_name>","resources":
{"requests":{"cpu":"800m"},"limits":{"cpu":"800m"}}}]}}'
```

将 `<pod_name>`、`<namespace>` 和 `<container_name>` 替换为目标 Pod、namespace 和容器名称。

由于此操作需要 Pod `resize` 子资源，因此请使用支持该子资源的 `kubectl` 或 API 客户端版本。

验证重启行为

检查 Pod 在调整大小后是否已重启：

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

如果对已变更的资源，`restartPolicy` 为 `NotRequired`，则 Pod 应保持运行且容器不会重启。

```
NAME         READY   STATUS    RESTARTS   AGE
resize-pod   1/1     Running   0          5s
```

如果对已变更的资源，`restartPolicy` 为 `RestartContainer`，则容器会重启，而 Pod 仍保持运行。

```
NAME         READY   STATUS    RESTARTS      AGE
resize-pod   1/1     Running   1 (5s ago)    5s
```

检查调整大小状态

检查当前应用到运行中容器的资源：

```
kubectl get pod <pod_name> \
  -n <namespace> \
  -o jsonpath='{.status.containerStatuses[?(@.name=="<container_name>")].
resources}'
```

检查调整大小条件和事件：

```
kubectl describe pod <pod_name> -n <namespace>
```

调整大小条件包括：

- `PodResizeInProgress`：kubelet 可以分配请求的资源，且正在应用更改。
- `PodResizePending`：kubelet 无法立即应用更改。
  - `Infeasible`：无法在当前节点上执行请求的调整。
  - `Deferred`：当前无法执行请求的调整，但以后可能可以。
  - `Error`：kubelet 在资源分配期间报告错误。

限制

- 只有 CPU 和内存 requests 和 limits 可以调整。
- 对于不可重启的 init containers 或 ephemeral containers，不支持原地 Pod 调整大小。
- 调整操作不得违反其他 Pod 可变性约束，例如更改 Pod QoS class。
- 由静态 `cpuManagerPolicy` 或 `memoryManagerPolicy` 管理的 Pod 不能原地调整大小。
- 使用 swap memory 的 Pod 在调整内存 requests 时必须使用 `RestartContainer`。
- ResourceQuota、LimitRange 和 admission policies 仍可能拒绝调整请求。

> **Note**
>
> 由于此操作需要 Pod `resize` 子资源，因此可能无法通过 web console 的资源编辑功能使用。请使用 `kubectl` 或支持 `resize` 子资源的 API 客户端版本。

其他资源

- Pods
- Resource Units
- Configuring VerticalPodAutoscaler (VPA)

- Pod API Reference

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。

配置

配置部分涵盖工作负载在运行时使用的命名空间作用域配置资源。

当您需要为应用、工作负载或集群侧命名空间资源创建、审阅、更新或删除配置数据时，请使用这些页面。

目录

选择正确的入口视图

非敏感配置

敏感配置

运维提示

选择正确的入口视图

ConfigMaps 和 Secrets 可以在新的 Web 控制台的不同视图中进行管理，但资源本身仍然属于某个具体的集群和命名空间。

- 当您管理当前项目范围内的工作负载配置时，请使用 **Applications** 视图。
- 当您从某个具体集群开始，并将配置作为集群侧命名空间操作的一部分进行管理时，请使用 **Clusters** 视图。

在这两种视图中，工作负载只能引用来自同一命名空间的 ConfigMap 或 Secret。

非敏感配置

将 **ConfigMaps** 用于不包含凭据或私密内容的运行时配置：

- Configuring ConfigMap

敏感配置

将 **Secrets** 用于凭据、令牌、密钥、证书或镜像拉取认证：

- Configuring Secrets

运维提示

当您更新或删除 ConfigMaps 或 Secrets 时，请评估其对下游工作负载的影响。根据工作负载使用这些数据的方式，变更在完全生效之前可能需要进行 Pod 重建、滚动更新、重启或 image-pull 重新验证。

请将运维指导放在操作发生的页面中：

- 创建和入口作用域检查应放在创建步骤旁边
- 工作负载消费检查应放在更新和删除行为旁边
- 集群视图差异应放在相同能力的旁边，而不是放在单独的附录中

配置 **ConfigMap**

了解 ConfigMap

ConfigMap 限制

ConfigMap 示例

使用 web console 创建 ConfigMap

使用新的 web console 创建 ConfigMap

使用 CLI 创建 ConfigMap

操作

配置 **Secrets**

了解 Secrets

创建 Opaque 类型 Secret

创建容器镜像仓库类型 Secret

创建 Basic Auth 类型 Secret

创建 SSH-Auth 类型 Secret

创建 TLS 类型 Secret

通过 web 控制台创建 Secret

使用 CLI 查看、编辑和删除
在 Pod 中使用 ConfigMap 的方式
ConfigMap 与 Secret 的对比
如何在 Pod 中使用 Secret
后续操作
操作
使用新 web 控制台创建 Secret

配置 ConfigMap

ConfigMap 允许您将配置工件与镜像内容解耦，以保持容器化应用程序的可移植性。以下各节将定义 ConfigMap 以及如何创建和使用它们。

目录

了解 ConfigMap
ConfigMap 限制
ConfigMap 示例
使用 web console 创建 ConfigMap
使用新的 web console 创建 ConfigMap
从 Applications 视图创建或更新
从 Clusters 视图创建或更新
使用 CLI 创建 ConfigMap
操作
使用 CLI 查看、编辑和删除
在 Pod 中使用 ConfigMap 的方式
作为环境变量
作为卷中的文件
作为单独的环境变量
ConfigMap 与 Secret 的对比

了解 ConfigMap

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，以保持容器化应用程序的可移植性。

`ConfigMap` 对象提供了将配置数据注入容器的机制，同时使容器不依赖于 OpenShift Container Platform。ConfigMap 可用于存储细粒度信息，例如单个属性；也可用于存储粗粒度信息，例如整个配置文件或 JSON blob。

`ConfigMap` 对象保存键值对形式的配置数据，这些数据可在 Pod 中使用，也可用于存储控制器等系统组件的配置数据。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（例如镜像）创建 ConfigMap 时，可以使用 `binaryData` 字段。

配置数据可以通过多种方式在 Pod 中使用。ConfigMap 可用于：

- 为容器中的环境变量赋值
- 在容器中设置命令行参数
- 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 ConfigMap 中。 ConfigMap 类似于 secret，但其设计更便于处理不包含敏感信息的字符串。

ConfigMap 限制

- 必须先创建 ConfigMap，才能在 Pod 中使用其内容。
- 可以编写控制器以容忍缺失的配置数据。请根据具体情况查阅使用 ConfigMap 配置的各个组件。
- `ConfigMap` 对象位于某个项目中。
- 它们只能被同一项目中的 Pod 引用。
- Kubectl 仅支持将 ConfigMap 用于它从 API server 获取的 Pod。这包括使用 CLI 创建的任何 Pod，或由 replication controller 间接创建的 Pod。它不包括使用 OpenShift Container Platform 节点的 `--manifest-url` 标志、`--config` 标志或其 REST API 创建的 Pod，因为这些并不是创建 Pod 的常见方式。

> **NOTE**
>
> Pod 只能使用同一 namespace 中的 ConfigMap。

ConfigMap 示例

现在您可以在 `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"
```

使用 **web console** 创建 **ConfigMap**

1. 转到 **Container Platform**。
2. 在左侧边栏中，单击 **Configuration** > **ConfigMap**。
3. 单击 **Create ConfigMap**。
4. 请参考以下说明配置相关参数。

参数	描述
Entries	指 `key:value` 键值对，支持 **Add** 和 **Import** 两种方式。 • **Add**：您可以逐个添加配置项，也可以在 Key 输入区域中粘贴一行或多行 key=value 对以批量添加配置项。 • **Import**：导入一个不大于 1M 的文本文件。文件名将作为键，文件内容将作为值，填充到一个配置项中。
Binary Entries	指不大于 1M 的二进制文件。文件名将作为键，文件内容将作为值，填充到一个配置项中。 注意：创建 ConfigMap 后，导入的文件无法修改。

批量添加格式示例：

```
# One key=value pair per line, multiple pairs must be on separate line
s, otherwise they will not be recognized correctly after pasting.
key1=value1
key2=value2
key3=value3
```

5. 单击 **Create**。

使用新的 **web console** 创建 **ConfigMap**

本页面主要描述传统控制台的工作流，但在新的 web console 中，也可以从不同视图访问相同功能。

从 **Applications** 视图创建或更新

当 ConfigMap 作为当前项目范围的一部分进行管理时，请使用 **Applications** 视图。

在从此视图创建或更新 ConfigMap 之前，请确认：

- 所选 cluster 已与当前项目关联
- 目标 namespace 与使用该配置的工作负载所使用的 namespace 相同
- 键、文件名和值格式与工作负载读取配置的方式匹配
- 之后 ConfigMap 会出现在所选项目和 cluster 上下文中

从 **Clusters** 视图创建或更新

当您从具体 cluster 开始，并将 ConfigMap 作为该 cluster 中的 namespace 资源进行管理时，请使用 **Clusters** 视图。

在从此视图创建或更新 ConfigMap 之前，请确认：

- 您已先打开目标 cluster
- 您已在该 cluster 中选择或确认目标 namespace
- 您关注的是 namespace 本地的工作负载影响，而不是项目成员关系
- 之后 ConfigMap 会出现在预期的 namespace 中

使用 **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

您可以单击列表页右侧的 (⋮)，或在详情页右上角单击 **Actions**，按需更新或删除 ConfigMap。

对 ConfigMap 的更改会影响引用该配置的工作负载，因此请提前阅读操作说明。

操作	描述
Update	• 添加或更新 ConfigMap 后，任何通过环境变量引用了此 ConfigMap（或其配置项）的工作负载都需要重新构建其 Pod，新的配置才会生效。 • 对于导入的二进制配置项，仅支持更新键，不支持更新值。
Delete	删除 ConfigMap 后，如果引用了此 ConfigMap（或其配置项）的工作负载在重新构建 Pod 时无法找到引用源，则在 Pod 创建期间可能会受到不利影响。

使用 **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
```

在 **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
```

每个键都是 `/etc/config` 下的一个文件，文件内容就是其值。

作为单独的环境变量

```
env:
  - name: APP_ENV
    valueFrom:
      configMapKeyRef:
        name: app-config
        key: APP_ENV
```

ConfigMap 与 Secret 的对比

功能	ConfigMap	Secret
数据类型	非敏感	敏感（例如密码）
编码	明文	Base64 编码
使用场景	配置、标志位	密码、令牌

配置 Secrets

目录

了解 Secrets
- 使用特性
- 支持的类型
- 使用方式

创建 Opaque 类型 Secret

创建容器镜像仓库类型 Secret

创建 Basic Auth 类型 Secret

创建 SSH-Auth 类型 Secret

创建 TLS 类型 Secret

通过 web 控制台创建 Secret

如何在 Pod 中使用 Secret
- 作为环境变量
- 作为挂载文件（Volume）

后续操作

操作

使用新 web 控制台创建 Secret
- 从 Applications 视图创建或更新
- 从 Clusters 视图创建或更新
- 验证工作负载影响

了解 Secrets

在 Kubernetes (k8s) 中，Secret 是一种基础对象，专用于存储和管理敏感信息，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到 Pod 定义或容器镜像中，从而增强安全性和可移植性。

Secret 类似于 ConfigMap，但它们专门用于机密数据。它们通常以 Base64 编码形式存储，并且可以通过多种方式被 pods 使用，包括挂载为卷或暴露为环境变量。

使用特性

- 增强安全性：与明文配置映射（Kubernetes ConfigMap）相比，Secret 通过使用 Base64 编码存储敏感信息，提供了更好的安全性。该机制结合 Kubernetes 的访问控制能力，显著降低了数据泄露风险。
- 灵活性和管理：与将敏感信息直接硬编码到 Pod 定义文件或容器镜像中相比，使用 Secret 提供了一种更安全、更灵活的方式。这种分离简化了敏感数据的管理和修改，而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 Secret，每种类型都针对特定使用场景进行了设计。平台通常支持以下类型：

- **Opaque**：一种通用的 Secret 类型，用于存储任意键值对形式的敏感数据，例如密码或 API 密钥。
- **TLS**：专门用于存储 TLS（Transport Layer Security）协议的证书和私钥信息，通常用于 HTTPS 通信和安全入口。
- **SSH Key**：用于存储 SSH 私钥，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)**：用于存储通过 SSH 协议传输的数据的身份验证信息。
- **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 身份验证字符串。

使用方式

应用程序可以通过以下不同方式在 pod 内使用 Secret：

- 作为环境变量：Secret 中的敏感数据可以直接注入容器的环境变量中。
- 作为挂载文件（**Volume**）：Secret 可以作为文件挂载到 pod 的卷中，使应用程序能够从指定的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 namespace 内的 Secret。有关高级用法和 YAML 配置，请参阅 Kubernetes 官方文档↗。

创建 **Opaque** 类型 **Secret**

```
kubectl create secret generic my-secret \
  --from-literal=username=admin \
  --from-literal=password=Pa$$w0rd
```

YAML

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4= # base64 of "admin"
  password: UGEkJHcwcmQ= # base64 of "Pa$$w0rd"
```

你可以像下面这样对其进行解码：

```
echo YWRtaW4= | base64 --decode  # output: admin
```

创建容器镜像仓库类型 **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

```
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、电子邮件和服务器信息转换为标准登录格式：

```
{
  "auths": {
    "https://registry.example.com": {
      "username": "myuser",
      "password": "mypass",
      "email": "my@example.com",
      "auth": "bXl1c2VyOm15cGFzcw==" # base64(username:password)
    }
  }
}
```

然后将该 JSON 进行 base64 编码，并将其用作 Secret 的 data 字段值。

在 Pod 中使用它：

```
imagePullSecrets:
  - name: my-container-registry-creds
```

创建 Basic Auth 类型 Secret

```
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 Secret

使用场景：存储 SSH 私钥（例如，用于 Git 访问）。

```
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 Secret

使用场景：TLS 证书（供 Ingress、webhooks 等使用）

```
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. 在左侧导航栏中，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在存储到 Secret 之前自动进行 Base64 编码。转换后的数据可在 YAML 视图中预览。
5. 点击 **Create**。

如何在 **Pod** 中使用 **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，取出键 `username` 对应的值，并将其赋给环境变量 `DB_USERNAME`。

作为挂载文件（Volume）

```
volumes:
  - name: secret-volume
    secret:
      secretName: my-secret

volumeMounts:
  - name: secret-volume
    mountPath: "/etc/secret"
```

后续操作

在同一 namespace 中创建原生应用的工作负载时，你可以引用已经创建好的 Secret。

操作

你可以点击列表页右侧的 (⋮)，或者在详情页右上角点击 **Actions**，按需更新或删除 Secret。

操作	说明
Update	在添加或更新 Secret 之后，通过环境变量引用了该 Secret（或其配置项）的工作负载，需要重建其 Pod，新的配置才会生效。
Delete	• 删除 Secret 后，通过环境变量引用了该 Secret（或其配置项）的工作负载，在重建 Pod 时可能会因无法找到引用源而受到影响。 • 请不要删除平台自动生成的 Secret，因为这可能会导致平台无法正常运行。例如：包含 namespace 资源认证信息的 service-account-token 类型 Secret，以及系统 namespace 中的 Secret（例如 kube-system）。

使用新 **web** 控制台创建 **Secret**

本页主要描述的是旧版控制台的工作流，但在新版 web 控制台的不同视图中，也可以实现相同的功能。

从 **Applications** 视图创建或更新

当 Secret 作为当前项目范围的一部分进行管理时，请使用 **Applications** 视图。

在通过该视图创建或更新 Secret 之前，请确认：

- 所选集群已与当前项目关联
- 目标 namespace 与使用该 Secret 的工作负载所使用的 namespace 相同
- 所选 Secret 类型和键名与预期的工作负载配置一致
- 随后该 Secret 会出现在所选项目和集群上下文中

从 **Clusters** 视图创建或更新

当你从某个具体集群开始，并在该集群中将 Secret 作为 namespace 资源进行管理时，请使用 **Clusters** 视图。

在通过该视图创建或更新 Secret 之前，请确认：

- 你已先打开目标集群
- 你已选择或确认该集群中的目标 namespace
- 你正在查看该 namespace 中的工作负载启动、挂载文件或镜像拉取影响
- 随后该 Secret 会出现在预期的 namespace 中

验证工作负载影响

无论你使用哪个视图，还需要确认：

- 引用该 Secret 的工作负载仍然位于与该 Secret 相同的 namespace 中
- 所需的 Pod 重建、滚动更新、重启或镜像拉取重新验证已经完成
- 删除该 Secret 不会因为缺少引用而破坏后续的工作负载启动、挂载或镜像拉取

在删除平台生成的或对工作负载至关重要的 Secret 之前，请务必格外谨慎。

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

将 ImagePullSecrets 添加到 Ser

- 创建 ImagePullSecret
- 将 ImagePullSecret 添加到 ServiceAccour
- 验证新 Pod 已设置 imagePullSecrets
- 在 OCI Connector 中使用 ImagePullSecre

Service 服务

- 概述
- 功能特性
- 配置和部署（管
- 用户指南（面向

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。

将 ImagePullSecrets 添加到 ServiceAccount

如果镜像仓库需要身份验证，你需要将相应的 `ImagePullSecrets` 添加到应用所使用的 `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 Pod 已设置 imagePullSecrets

在 OCI Connector 中使用 ImagePullSecret

前提条件

创建 Connector 代理 Secret

引用该 Secret

验证与排障

创建 ImagePullSecret

要创建 ImagePullSecret，请参阅 创建 Secret 了解创建 ImagePullSecret 的详细步骤。

将 ImagePullSecret 添加到 ServiceAccount

如果你的应用的 Pod 使用 `ServiceAccount` `example`，你可以将 `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>` 替换为你的应用所在的命名空间，并将 `my-registry-creds` 替换为你创建的 `ImagePullSecret` 名称。

你可以通过描述该 `ServiceAccount` 来验证 `ImagePullSecret` 是否已添加：

```
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>
```

你应该会看到 `Image pull secrets` 部分显示已添加的 secret。

> **NOTE**
>
> 注意：如果你的 Pod 未指定 `ServiceAccount`，则默认会使用该命名空间中的 `default` `ServiceAccount`。你也可以用同样的方式将 `ImagePullSecret` 添加到 `default` `ServiceAccount`。

验证新 Pod 已设置 imagePullSecrets

当你创建一个使用 `ServiceAccount` `example` 的新 Pod 时，Pod 会自动使用 `ServiceAccount` 中指定的 `ImagePullSecrets`。

你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}'
```

在 OCI Connector 中使用 ImagePullSecret

当通过 OCI Connector 选择工作负载镜像时，工作负载仍然需要一个镜像拉取 Secret 来对 Connector 代理进行身份验证。Connector 避免将原始 registry 的长期凭证分发到工作负载命名空间，但这并不意味着工作负载不需要 Kubernetes 镜像拉取 Secret。Connector 代理 Secret 使用的是一个被授权访问 Connector 的 ServiceAccount token，而不是原始 registry 密码。

你可以直接在 Pod 模板中引用该 Secret，或者将其附加到工作负载所使用的 ServiceAccount 上。任一方式都可以。

前提条件

在创建 Secret 之前，请确认以下事项：

- 已安装 Alauda DevOps Connectors，Connector 处于 Ready 状态，并且已启用 `enable-pod-image-pull-via-connector`。
- OCI 代理已通过主机端口或 Ingress 暴露，并且每个目标节点的 container runtime 都可以访问该代理地址。如果代理使用 HTTP，则 runtime 必须允许该代理地址。
- 用于创建 token 的 ServiceAccount 已获授权访问 Connector。
- Secret 已创建在与工作负载相同的 namespace 中。

创建 Connector 代理 Secret

获取 Connector 代理地址，并为工作负载的 ServiceAccount 创建 token：

```
kubectl get connector <connector-name> -n <connector-namespace> \
  -o jsonpath='{.status.proxy.httpAddress.url}'

kubectl create token <service-account> \
  -n <workload-namespace> \
  --duration=<approved-duration>
```

创建镜像拉取 Secret：

```
kubectl create secret docker-registry <connector-pull-secret> \
  --docker-server='<connector-proxy-host-and-port>' \
  --docker-username='u' \
  --docker-password='<service-account-token>' \
  --docker-email='<email>' \
  -n <workload-namespace>
```

`--docker-server` 的值必须指向 Connector 代理，而不是原始 registry。如果 `.status.proxy.httpAddress.url` 包含 Connector 路径，请使用代理的 host 和 port 作为 registry server。该命令会创建一个类型为 `kubernetes.io/dockerconfigjson` 的 Secret。

token 必须来自一个有权限访问 Connector 的 ServiceAccount。未指定经批准持续时间创建的 token，默认有效期约为一小时。请根据你所在组织的安全策略选择 `--duration`，并在 token 过期前轮换或重新创建 Secret。不要将手动复制的 token 视为永久凭证或轮换机制。

引用该 Secret

要直接在 Pod 模板中引用该 Secret，请配置：

```
spec:
  imagePullSecrets:
    - name: <connector-pull-secret>
```

或者，将该 Secret 附加到工作负载所使用的 ServiceAccount：

```
kubectl patch serviceaccount <service-account> \
  -n <workload-namespace> \
  -p '{"imagePullSecrets":[{"name":"<connector-pull-secret>"}]}'
```

Pod 模板引用和 ServiceAccount 继承是两种可选的有效来源。你无需同时配置两者。

验证与排障

创建或更新工作负载后，请按以下顺序验证 Connector 路径：

1. 检查工作负载 Pod 模板是否具有期望的 `connectors.<baseDomain>/connectors` 注解，并且包含每个使用 OCI Connector 的 container 和 init container 的映射。
2. 检查 Pod 选择的，或从其 ServiceAccount 继承的镜像拉取 Secret。
3. 检查实际 Pod 镜像是否已重写为 Connector 代理地址。
4. 检查 Pod 是否到达 `Running` 状态（对于已完成的 Job，则为 `Completed`），然后查看其 Events。

运行以下命令：

```
kubectl get serviceaccount <service-account> -n <workload-namespace> -o y
aml
kubectl get pod <pod-name> -n <workload-namespace> -o jsonpath='{.spec.im
agePullSecrets}'
kubectl get pod <pod-name> -n <workload-namespace> -o jsonpath='{.spec.co
ntainers[*].image}'
kubectl get pod <pod-name> -n <workload-namespace>
kubectl describe pod <pod-name> -n <workload-namespace>
```

如果实际 Pod 镜像仍然指向原始 registry，请先检查 Connector 注解、Secret 的 namespace 和类型、ServiceAccount 关联以及 token 有效性。原始 HTTP registry 的 HTTPS 错误可能表明工作负载没有进入 Connector 代理路径；不要通过永久允许原始 registry 作为不安全 registry 来规避此问题。

Service 服务端证书

目录

概述
功能特性
配置和部署（管理员）
前提条件
1. 配置 RBAC 权限
2. 创建证书
3. 配置 Kyverno 策略
用户指南（面向开发者）

概述

Service 服务端证书为容器云平台中的 Service 提供自动化 TLS 证书生成和管理功能。此功能支持安全的 HTTPS 通信以及内部组件之间的流量加密，确保数据隐私和系统安全。

功能特性

此自动化证书签发机制提供以下主要功能：

- 零接触签发：开发者无需手动生成证书签名请求（CSR），也无需与证书颁发机构交互。证书签发通过标准 Kubernetes 注解实现全自动化。
- 跨命名空间 **CA** 同步：无需再手动将平台根 CA 复制到每个业务命名空间，从而消除相关繁琐操作。根 CA 证书会自动同步，使服务能够轻松验证彼此的身份。
- 自动化生命周期管理：利用 `cert-manager` 确保证书不仅能够自动签发，还能在过期前无缝续期，最大限度降低运维开销，并防止因证书过期导致服务中断。
- 标准化安全性：确保 TLS 证书通过统一、集中管理的 `ClusterIssuer` 生成，在平台上的所有应用之间保持一致的安全标准。

配置和部署（管理员）

要实现上述功能，平台系统管理员需要提前为 `Kyverno` 配置相关准入策略（`ClusterPolicy`）以及必要的 RBAC 权限。

前提条件

开始操作前，请确保已在集群中安装并启用 **Alauda Container Platform Policy Management** 插件。有关安装详情，请参阅安装 Compliance Plugin。

1. 配置 **RBAC** 权限

要授予 Kyverno 创建相应资源和调用相关接口的权限，请将以下权限聚合到平台后台控制器：

```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. 创建证书

创建以下 Certificate 资源，将其用作服务根 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. 配置 Kyverno 策略

创建并应用以下三个核心 `ClusterPolicy` 资源：

- **clone-ca-secret**：监听 Namespace 创建事件，并为新命名空间生成 `service-root-ca` Secret 资源。
- **sync-ca-rotation**：监听 `service-root-ca` 的变更，通过 API 获取实际的 CA 证书内容，并将 `ca.crt` 值同步到目标命名空间中的 Secret。
- **generate-service-cert**：监听带有 `service.alauda.io/serving-cert-secret-name` 注解的 Service 资源，并根据其名称和命名空间生成支持泛域名的 `Certificate` 签发请求。

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:
```

```
            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"
```

用户指南（面向开发者）

对于应用开发者，只要底层集群策略已配置完成，就无需关注底层证书签发逻辑。定义业务 Service 资源时，只需添加专属注解，即可自动配置证书：

1. 向 **Service** 添加注解

在业务命名空间（例如 `my-namespace`）中创建或更新 Service，并添加 `service.alauda.io/serving-cert-secret-name` 注解，以指定生成的证书 Secret 的名称。例如：

```
apiVersion: v1
kind: Service
metadata:
  name: my-secure-service
  namespace: my-namespace
  annotations:
    service.alauda.io/serving-cert-secret-name: "my-secure-service-tls" #
Will generate a certificate Secret named my-secure-service-tls for you
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：

- 检查命名空间中是否存在自动生成的 TLS 证书 Secret：

  ```
  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 服务证书 Secret（`my-secure-service-tls`）均已就绪，因此可以将它们作为数据卷挂载到目标应用的 `Deployment` 或 `StatefulSet` 等工作负载中。

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:
            # Mount the serving certificate
            - name: tls-cert
              mountPath: "/etc/tls/certs"
              readOnly: true
            # Mount the root CA certificate
            - name: root-ca
              mountPath: "/etc/tls/ca"
              readOnly: true
      volumes:
        # Reference the automatically generated Service certificate Secre
t
        - name: tls-cert
          secret:
            secretName: my-secure-service-tls
        # Reference the automatically synchronized CA certificate Secret
        - name: root-ca
          secret:
            secretName: service-root-ca
```

有关通过 Web 控制台配置工作负载的更多详情，请参阅配置容器部分。

镜像

镜像概述

镜像概述

理解容器和镜像

镜像

镜像 registry

镜像 repository

镜像 tag

镜像 ID

容器

实用指南

创建镜像

了解容器最佳实践

在镜像中包含元数据

管理镜像

镜像拉取策略

使用 image pull secret

镜像概述

目录

理解容器和镜像

镜像

镜像 registry

镜像 repository

镜像 tag

镜像 ID

容器

理解容器和镜像

当你着手创建和管理容器化软件时，理解容器和镜像这两个重要概念非常关键。镜像包含一组可直接运行的软件，而容器则是容器镜像的一个运行中实例。同一个镜像名称下的不同版本通过不同的 tag 表示。

镜像

灵雀云容器平台 中的容器基于 OCI 格式的容器镜像。镜像是一种二进制文件，其中包含运行单个容器所需的全部内容，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。除非在创建时授予额外访问权限，否则容器只能访问镜像中定义的资源。通过在多个主机上的多个容器中部署同一个镜像，并在它们之间进行负载均衡，灵雀云容器平台 可以为封装在镜像中的服务提供冗余和水平扩展能力。

你可以直接使用 `nerdctl` 或 `container` CLI 来构建镜像，但 灵雀云容器平台 也提供了 builder images，可通过向现有镜像中添加你的代码或配置来协助创建新镜像。

由于应用会随着时间不断演进，同一个镜像名称实际上可能指向同一镜像的多个不同版本。每个不同的镜像都通过其 hash 唯一标识，即一个很长的十六进制数，例如 fd44297e2ddb050ec4f…，通常会缩短为 12 个字符，例如 fd44297e2ddb。

镜像 registry

镜像 registry 是一种内容服务器，可用于存储和提供容器镜像。例如：

- Quay.io Container Registry ↗
- 灵雀云容器平台 Registry

registry 包含一个或多个镜像 repository 的集合，每个 repository 又包含一个或多个带标签的镜像。灵雀云容器平台 可以提供自己的镜像 registry，用于管理自定义容器镜像。

镜像 repository

镜像 repository 是相关容器镜像及其标识 tag 的集合。例如，灵雀云容器平台 Jenkins 镜像位于以下 repository 中：

```
jenkins-2-centos7
```

镜像 tag

镜像 tag 是应用于 repository 中容器镜像的标签，用于将特定镜像与 image stream 中的其他镜像区分开来。通常，tag 表示某种版本号。例如，下面的 :v3 .11.59-2 就是 tag：

```
jenkins-2-centos7:v3.11.59-2
```

你可以为镜像添加额外的 tag。例如，一个镜像可能被分配 :v3 .11.59-2 和 :latest 这两个 tag。

镜像 ID

镜像 ID 是一个 SHA（Secure Hash Algorithm）代码，可用于拉取镜像。SHA 镜像 ID 不能更改。特定的 SHA 标识符始终引用完全相同的容器镜像内容。例如：

```
jenkins-2-centos7@sha256:ab312bda324
```

容器

灵雀云容器平台 应用的基本单元称为容器。Linux 容器技术是一种轻量级机制，用于隔离运行中的进程，使其仅能与指定资源交互。container 这个词被定义为容器镜像的一个特定运行中或暂停中的实例。

在单个主机上，可以同时运行许多应用实例，这些实例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，例如 Web server 或 database，不过容器也可用于任意工作负载。

多年来，Linux kernel 一直在为容器技术逐步加入相关能力。早期的容器 runtime 项目引入了便于在主机上运行隔离应用的管理界面。近年来，Open Container Initiative↗ 已为容器格式和容器 runtime 制定了开放标准。灵雀云容器平台 和 Kubernetes 增加了在多主机部署之间编排 OCI 格式容器的能力。

尽管在使用 灵雀云容器平台 时你不会直接与容器 runtime 交互，但了解它们的能力和术语，对于理解它们在 灵雀云容器平台 中的作用，以及你的应用如何在容器内部运行，仍然很重要。

实用指南

创建镜像

了解容器最佳实践

在镜像中包含元数据

管理镜像

镜像拉取策略

使用 image pull secret

创建镜像

了解如何基于可直接使用的预构建镜像创建你自己的容器镜像。这个过程包括学习编写镜像的最佳实践、定义镜像元数据、测试镜像，以及使用自定义构建器工作流创建可用于 Alauda Container Platform Registry 的镜像。创建镜像后，你可以将其推送到 **Alauda Container Platform Registry**。

目录

了解容器最佳实践

通用容器镜像指南

在镜像中包含元数据

定义镜像元数据

了解容器最佳实践

在 Alauda Container Platform 上创建用于运行的容器镜像时，作为镜像作者，有许多最佳实践需要考虑，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你的镜像具有很高的可用性，并且易于在 Alauda Container Platform 上使用。

通用容器镜像指南

以下指南适用于创建容器镜像的一般场景，与这些镜像是否在 Alauda Container Platform 上使用无关。

复用镜像

在可能的情况下，使用 FROM 语句将你的镜像基于合适的上游镜像。这可确保当上游镜像更新时，你的镜像能够轻松获取安全修复，而无需你直接更新依赖项。

另外，请在 FROM 指令中使用标签，例如 `alpine:3.20`，以便用户清楚地知道你的镜像基于哪个具体版本。使用除 latest 之外的标签可确保你的镜像不会受到上游镜像最新版本中可能引入的破坏性更改影响。

在标签内保持兼容性

为你自己的镜像打标签时，请尽量在同一标签内保持向后兼容。例如，如果你提供一个名为 image 的镜像，并且它当前包含版本 `1.0`，你可以提供一个 `image:v1` 标签。当你更新镜像时，只要它仍然与原始镜像兼容，就可以继续将新镜像标记为 `image:v1`，这样使用该标签的下游使用者就可以在不受破坏的情况下获得更新。

如果之后你发布了一个不兼容的更新，那么请切换到新标签，例如 `image:v2`。这样下游使用者可以按需升级到新版本，而不会被新的不兼容镜像意外破坏。任何使用 `image:latest` 的下游使用者都要承担引入不兼容更改的风险。

避免多个进程

不要在一个容器中启动多个服务，例如数据库和 `SSHD`。这样做并非必要，因为容器是轻量级的，可以轻松地关联在一起，以编排多个进程。Alauda Container Platform 允许你通过将相关镜像分组到单个 Pod 中，轻松地将它们共同部署并统一管理。

这种共置可确保容器共享用于通信的网络命名空间和存储。由于每个镜像可以更少地、独立地更新，因此更新造成的影响也更小。单进程的信号处理流程也更清晰，因为你不必管理向已派生进程的信号路由。

在包装脚本中使用 exec

许多镜像会使用包装脚本在启动要运行的软件进程之前执行一些初始化设置。如果你的镜像使用了这种脚本，请让该脚本使用 `exec`，这样脚本进程就会被你的软件进程替换。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，而不是你的软件进程。这不是你想要的结果。

如果你有一个包装脚本，用来为某个服务器启动进程。你启动容器时，例如使用 `podman run -i`，这会运行包装脚本，而包装脚本又会启动你的进程。如果你希望使用 `CTRL+C` 关闭容器。如果你的包装脚本使用 `exec` 启动服务器进程，`podman` 会将 SIGINT 发送到服务器进

程，一切都会按预期运行。如果你在包装脚本中没有使用 `exec` ，`podman` 会将 SIGINT 发送到包装脚本进程，而你的进程会继续运行，仿佛什么都没有发生。

另外还要注意，容器中的进程会作为 `PID 1` 运行。这意味着，如果主进程终止，整个容器也会停止，从而取消从 `PID 1` 进程启动的任何子进程。

清理临时文件

移除在构建过程中创建的所有临时文件。这也包括通过 `ADD` 命令添加的任何文件。例如，在执行 `yum install` 操作后运行 `yum clean` 命令。

你可以通过如下方式编写 `RUN` 语句，防止 `yum` 缓存进入镜像层：

```
RUN yum -y install mypackage && yum -y install myotherpackage && yum clea
n all -y
```

请注意，如果你改为编写：

```
RUN yum -y install mypackage
RUN yum -y install myotherpackage && yum clean all -y
```

那么第一次 `yum` 调用会在该层中留下额外文件，而当之后执行 `yum clean` 操作时，这些文件无法被移除。多余文件不会在最终镜像中可见，但它们仍存在于底层镜像层中。

当前的容器构建过程不允许在较晚层中运行的命令在早期层中已删除内容后，缩减镜像所占用的空间。不过，这种情况在未来可能会改变。这意味着，如果你在后续层中执行 `rm` 命令，尽管文件已被隐藏，但并不会减少要下载的镜像总体大小。因此，与 `yum clean` 的示例一样，最好在创建文件的同一个命令中将其移除，这样它们就不会被写入某一层。

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时间。

按正确顺序放置指令

容器构建器会读取 `Dockerfile` ，并从上到下执行指令。每条成功执行的指令都会创建一个层，该层可在下次构建此镜像或其他镜像时复用。将很少变更的指令放在 `Dockerfile` 顶部非常重要。这样可以确保后续对同一镜像的构建速度非常快，因为缓存不会因上层更改而失效。

例如，如果你正在处理一个 `Dockerfile`，其中包含一个用于安装你正在迭代的文件的 `ADD` 命令，以及一个用于 `yum install` 软件包的 `RUN` 命令，那么最好将 `ADD` 命令放在最后：

```
FROM foo
RUN yum -y install mypackage && yum clean all -y
ADD myfile /test/myfile
```

这样，每次你编辑 `myfile` 并重新运行 `podman build` 时，系统都会复用 `yum` 命令对应的缓存层，并且只会为 `ADD` 操作生成新层。

如果你反过来将 `Dockerfile` 编写为：

```
FROM foo
ADD myfile /test/myfile
RUN yum -y install mypackage && yum clean all -y
```

那么每次你更改 `myfile` 并重新运行 `podman build` 时，`ADD` 操作都会使 `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` 指令会使容器中的端口对主机系统和其他容器可用。虽然也可以通过 `podman run` 调用来指定某个端口应被暴露，但在 `Dockerfile` 中使用 `EXPOSE` 指令，通过显式声明软件运行所需的端口，可以让人和软件都更容易使用你的镜像：

- 暴露的端口会显示在由你的镜像创建的容器关联的 `podman ps` 输出中。
- 暴露的端口会出现在 `podman inspect` 返回的镜像元数据中。
- 当你将一个容器链接到另一个容器时，暴露的端口也会被关联。

设置环境变量

使用 `ENV` 指令设置环境变量是一种良好实践。其中一个例子是设置项目版本。这样人们无需查看 `Dockerfile` 就能轻松找到版本。另一个例子是在系统中声明一个可供其他进程使用的路径，例如 `JAVA_HOME`。

避免默认密码

避免设置默认密码。很多人会扩展镜像，却忘记移除或修改默认密码。如果生产环境中的用户被分配了一个众所周知的密码，这可能会导致安全问题。应改为通过环境变量配置密码。

如果你确实选择设置默认密码，请确保容器启动时显示适当的警告消息。该消息应告知用户默认密码的值，并说明如何修改它，例如需要设置哪个环境变量。

避免 sshd

最好不要在镜像中运行 `sshd`。你可以使用 `podman exec` 访问在本地主机上运行的容器。在集群中，使用 `kubectl exec` 访问由 Alauda Container Platform 管理的容器。在镜像中安装并运行 `sshd` 会扩大攻击面，并增加补丁维护负担。

将卷用于持久数据

镜像使用卷来存储持久数据。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。通过将卷用于所有持久化存储需求，即使容器重启或迁移，内容也会被保留。如果你的镜像将数据写入容器内的任意位置，这些内容将无法被保留。

所有即使在容器销毁后仍需保留的数据，都必须写入卷。容器引擎支持容器的 `readonly` 标志，可用于严格强制不向容器中的临时存储写入数据的良好实践。现在就围绕该能力设计镜像，将使你将来更容易利用它。

在 `Dockerfile` 中显式定义卷，可以让镜像使用者更容易理解在运行你的镜像时需要定义哪些卷。

有关卷在 Alauda Container Platform 中如何使用的更多信息，请参见 Kubernetes documentation↗。

> 注意：
> 即使使用持久卷，每个镜像实例也都有自己的卷，并且各实例之间的文件系统不共享。这意味着卷不能用于在集群中共享状态。

在镜像中包含元数据

定义镜像元数据有助于 Alauda Container Platform 更好地使用你的容器镜像，从而为使用你镜像的开发人员提供更好的体验。例如，你可以添加元数据来提供有用的镜像描述，或给出其他可能也需要的镜像建议。

当前的元数据集合覆盖了现有用例所需的字段。未来可能会增加更多元数据或用例。

定义镜像元数据

你可以在 `Dockerfile` 中使用 `LABEL` 指令定义镜像元数据。标签类似于环境变量，都是附加到镜像或容器上的键值对。标签与环境变量不同之处在于，它们对正在运行的应用不可见，并且还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 reference ↗。

标签名称通常会使用命名空间。命名空间会相应设置，以反映将要获取这些标签并使用它们的项目。对于 Kubernetes，命名空间是 io.k8s。

管理镜像

使用 Alauda Container Platform 时，你可以与镜像进行交互，具体取决于镜像的 registry 所在位置、这些 registry 所需的认证要求，以及你希望构建和部署的行为方式。

目录

镜像拉取策略

镜像拉取策略概述

使用 image pull secret

允许 pod 引用来自其他受保护 registry 的镜像

创建 pull secret

在工作负载中使用 pull secret

镜像拉取策略

pod 中的每个容器都有一个容器镜像。在你创建镜像并将其推送到 registry 后，就可以在 pod 中引用它。

镜像拉取策略概述

当 Alauda Container Platform 创建容器时，它会使用容器 `imagePullPolicy` 来确定在启动容器之前是否应拉取镜像。`imagePullPolicy` 有三种可能的值：

`imagePullPolicy` 的取值表：

Value	Description
Always	始终拉取镜像。
IfNotPresent	仅当镜像尚未存在于节点上时才拉取。
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 默认为 Always。
2. 否则，Alauda Container Platform 会将 imagePullPolicy 默认为 IfNotPresent。

使用 image pull 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，无需执行额外操作。

但是，在其他场景中，例如跨 Alauda Container Platform 项目引用镜像，或从受保护的 registry 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，你必须使用容器客户端凭据（例如 `Podman`）创建一个 pull secret，并将其添加到你的 ServiceAccount 中。

容器客户端使用配置文件来存储登录受保护或不受保护 registry 的认证详情：

如果你之前登录过受保护或不受保护的 registry，这些文件会保存你的认证信息。

创建 pull secret

你可以获取 image pull secret，以便从私有容器镜像 registry 或 repository 拉取镜像。你可以参考 从私有 Registry 拉取镜像↗。

在工作负载中使用 **pull secret**

你可以通过以下任一方式使用 pull secret，允许工作负载从私有 registry 拉取镜像：

- 将 secret 关联到 `ServiceAccount`，这会自动将该 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 这类环境很有用。

你可以通过将 secret 添加到你的 ServiceAccount 来为 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，你也可以在 pod 或工作负载定义中直接引用它。这对于 ArgoCD 等 GitOps 工作流很有用。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：

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: <example_workflow>
spec:
  entrypoint: <main_task>
  imagePullSecrets:
  - name: <pull_secret_name>
```

Alauda Container Platform Registry 镜像使用

本节介绍开发人员和工作负载如何访问 ACP 4.4 的 Alauda Container Platform Registry。

Registry 管理员任务（包括安装、配置和暴露）记录在 Alauda Container Platform Registry 管理中。

目录

文档结构

标准地址

文档结构

需求	继续阅读
从工作负载或外部客户端访问 Registry。	访问 Registry
使用 `ac` 列出、标记、导入、镜像、删除和检查镜像。	使用 ac 管理镜像
查看 Registry 组件和管理员设置。	Registry 管理

标准地址

对于在集群中运行的工作负载，请使用内部服务的完全限定地址：

```
image-registry.image-registry-system.svc:5000
```

仅对于开发人员计算机、CI 系统或集群外的其他客户端，使用外部 Ingress 主机名。请参阅暴露 Registry。

访问 **Registry**

前提条件

所需的命名空间权限

从集群内部访问 Registry

验证外部 OCI 客户端

推送和拉取镜像

查询镜像 API 资源

使用 **ac** 管理镜像

前提条件

Registry 模式

命令组

发现并登录 Registry

管理 ImageStreams

查询镜像和 ImageStream

删除镜像引用

直接处理 OCI 镜像

访问 Registry

使用此页面从集群内部、外部客户端以及通过 ACP Image API 访问 Alauda Container Platform Registry。

目录

前提条件

所需的命名空间权限

从集群内部访问 Registry

验证外部 OCI 客户端

推送和拉取镜像

查询镜像 API 资源

前提条件

- Alauda Container Platform Registry 已安装并在目标集群中可用。
- 镜像命名空间已存在。
- 当前用户或服务账户具有所需的命名空间权限。
- 使用 Registry 或 Image API 命令时，已安装并完成身份验证的 `ac` CLI。

对于 `ac` Image API 命令，请选择 ACP 4.4 Registry 后端：

```
ac config set-registry-mode modern
```

所需的命名空间权限

您的用户账户或工作负载服务账户必须在拥有镜像仓库的命名空间中具有权限。需要跨命名空间访问时，请让命名空间管理员授予所需的镜像角色。

Registry 使用 ImageStream 层授权：

操作	典型角色	**Image API** 权限
拉取	`system:image-puller`	`image.alauda.io` `imagestreams/layers` `get`
推送	`system:image-pusher`	`image.alauda.io` `imagestreams/layers` `update`
删除	`system:image-deleter`	目标镜像元数据的 Image API 删除权限

其他 Registry 角色（`system:image-signature-writer`、`system:image-registry-catalog-reader` 和 `system:image-registry-metrics-reader`）列于 管理访问权限和清理中的完整权限表中。

有关角色绑定示例，请参阅管理访问权限和清理。

从集群内部访问 **Registry**

对于集群内部的工作负载，请使用内部服务地址：

```
image-registry.image-registry-system.svc:5000
```

工作负载镜像引用示例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: team-a
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      serviceAccountName: default
      containers:
        - name: app
          image: image-registry.image-registry-system.svc:5000/team-a/my-
app:v1
```

将清单保存为 `my-app.yaml` 并应用：

```
kubectl apply -f my-app.yaml
kubectl -n team-a rollout status deployment/my-app --timeout=300s
kubectl -n team-a get pods -l app=my-app
```

预期结果：

- Deployment 在 `team-a` 中创建，其 Pod 使用内部 Registry 镜像并变为就绪状态。

镜像仓库路径限定于命名空间。使用 `<registry-host>/<namespace>/<repository>:<tag>`；不要在命名空间后包含其他路径段。

对于自定义服务账户，请确保其在镜像命名空间中具有拉取权限。当配置 Registry 管理服务账户拉取密钥时，Operator 会注入托管的拉取密钥。

在工作负载命名空间中创建服务账户：

```
kubectl create serviceaccount app-puller -n team-b
```

让命名空间管理员在镜像命名空间中授予服务账户拉取权限：

```
kubectl create rolebinding image-puller-app-puller \
  --clusterrole=system:image-puller \
  --serviceaccount=team-b:app-puller \
  -n team-a
```

验证是否已注入托管的拉取密钥：

```
PULL_SECRET="$(
  kubectl -n team-b get serviceaccount app-puller \
    -o jsonpath='{.imagePullSecrets[0].name}'
)"
test -n "$PULL_SECRET" || {
  echo "The managed pull Secret has not been injected yet." >&2
  exit 1
}
printf 'Managed pull Secret: %s\n' "$PULL_SECRET"
kubectl -n team-b get secret "$PULL_SECRET" \
  -o jsonpath='{.data}' | wc -c
```

预期结果：

- 命令会输出托管拉取 Secret 名称以及非零的数据大小。不要输出或解码 Secret 数据。

控制器会异步刷新托管的拉取凭据。如果未列出拉取 Secret，请等待协调完成，并让 Registry 管理员确认已启用托管服务账户拉取密钥，且工作负载命名空间未被忽略。

在工作负载中使用该服务账户：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: team-b
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      serviceAccountName: app-puller
      containers:
        - name: app
          image: image-registry.image-registry-system.svc:5000/team-a/my-app:v1
          imagePullPolicy: Always
```

将清单保存为 `my-app-app-puller.yaml` 并应用：

```bash
kubectl apply -f my-app-app-puller.yaml
```

验证发布状态：

```bash
kubectl -n team-b rollout status deployment/my-app --timeout=300s
kubectl -n team-b get pods -l app=my-app
```

预期结果：

- 发布成功完成。
- Pod 已就绪，且未报告镜像拉取错误。如果 Pod 处于 `ImagePullBackOff`，请验证 ServiceAccount 权限、注入的拉取 Secret、内部服务地址和镜像仓库路径。

验证外部 OCI 客户端

对于外部访问，请使用 `ac registry login` 为具有命名空间权限的 ACP 账户写入凭据：

```
ac registry login \
  --registry registry.example.com
```

要将凭据写入指定的 OCI registry auth 文件，请使用 `--to`：

```
mkdir -p /tmp/registry-auth

ac registry login \
  --registry registry.example.com \
  --to /tmp/registry-auth/config.json

export REGISTRY_AUTH_FILE=/tmp/registry-auth/config.json
```

该命令会将 `registry.example.com` 的凭据写入指定的 auth 文件。将导出的 `REGISTRY_AUTH_FILE` 与需要 Registry 身份验证的 `ac image` 命令结合使用。

如果 registry 使用私有 CA，请在推送和拉取操作之前配置客户端信任。仅当端点使用 HTTP，或使用客户端不信任的非生产环境测试证书时，才使用 `--insecure`。

推送和拉取镜像

标记并推送镜像：

```
nerdctl tag my-app:latest registry.example.com/team-a/my-app:v1
nerdctl push registry.example.com/team-a/my-app:v1
```

按标签拉取：

```
nerdctl pull registry.example.com/team-a/my-app:v1
```

按摘要拉取：

```
nerdctl pull registry.example.com/team-a/my-app@sha256:<digest>
```

查询镜像 API 资源

列出 ImageStreams：

```
kubectl get imagestreams.image.alauda.io -A
ac get imagestreams -A
```

显示一个 ImageStream：

```
ac get imagestreams my-app -n team-a -o yaml
```

显示某个标签的当前摘要和历史摘要：

```
ac get imagestreamtags my-app:v1 -n team-a -o wide
```

显示某个摘要对应的 Image 元数据：

```
ac get imagestreamimages my-app@sha256:<digest> -n team-a -o yaml
```

通过当前的 `modern` registry 模式列出集群范围的 Image 资源：

```
ac config set-registry-mode modern
ac get images
ac get images -o yaml
```

对于命名空间范围的仓库视图，请查询 ImageStream 资源，而不是传递 Registry URL 覆盖：

```
ac get imagestreams -n team-a
ac get imagestreamtags my-app:v1 -n team-a -o wide
```

使用 ac 管理镜像

使用 `ac` 镜像命令处理 Alauda Container Platform Registry 以及 ACP Image API。ACP 4.4 中的 Alauda Container Platform Registry 使用 `modern` Registry 模式，并为 ACP 集群提供 Image 和 ImageStream 工作流。

目录

前提条件

Registry 模式

命令组

发现并登录 Registry

管理 ImageStreams

查询镜像和 ImageStream

删除镜像引用

直接处理 OCI 镜像

前提条件

- 已安装 `ac` CLI，并已通过身份验证连接到目标集群。
- Registry 和 `image.alauda.io/v1` Image API 可用。
- 你拥有命令所使用的 ImageStream、Image 和工作负载资源所需的权限。

Registry 模式

`ac` 根据当前集群的 Registry 模式选择镜像后端：

模式	后端	用途
`modern`	`image.alauda.io/v1` Image 和 ImageStream API 以及 Registry 服务。	ACP 4.4 中的 Registry 管理。
`auto`	自动检测。	发现。在执行 ImageStream 或 Image API 操作前设置 `modern`，前提是命令需要显式模式。

检查并设置模式：

```
ac config get-registry-mode
ac config set-registry-mode modern
```

在管理 ImageStream、导入镜像、为镜像添加标签，或通过 Image API 图查看 Image API 使用情况之前，使用 `modern` 模式。

占位符	描述
`<digest>`	`ac get images` 返回的镜像摘要或 ImageStream 标签查询结果。包含 `sha256:` 前缀。
`REGISTRY_AUTH_FILE`	`ac registry login` 写入的本地身份验证文件路径；使用 `--registry-config` 将其传递给直接镜像命令。

命令组

工作流	命令	**Registry** 模式行为
发现并登录集成 Registry	`ac registry info`、`ac registry login`	使用当前 Registry 模式解析 Registry 主机。

工作流	命令	**Registry** 模式行为
列出并删除集成 Registry 镜像	`ac get images`、`ac delete images`	使用选定的 Registry 模式。
管理 ImageStream	`ac create imagestream`、`ac create imagestreamtag`、`ac import-image`、`ac tag`、`ac set image-lookup`	需要 `modern` 模式。
查询 Image API 资源	`ac get imagestreams`、`ac get imagestreamtags`、`ac get imagestreamimages`	需要 Image API。
直接使用 OCI 镜像	`ac image info`、`ac image mirror`、`ac image append`、`ac image extract`	直接与 Registry 通信。仅在限定范围的测试或私有 CA 场景中使用 `--insecure`。

管理使用情况报告、清理和信任验证，请参阅 管理访问权限和清理。

发现并登录 **Registry**

打印当前 Registry 模式选定的 Registry 主机：

```
ac registry info
```

打印内部主机：

```
ac registry info --internal
```

打印从配置的外部路由发现的公共主机：

```
ac registry info --public
```

检查 Registry `/v2/` 端点是否可访问：

```
ac registry info --check
```

将 Registry 凭证写入默认本地凭证文件：

```
ac registry login
```

将凭证写入临时或 CI auth 文件：

```
mkdir -p /tmp/registry-auth

ac registry login \
  --registry registry.example.com \
  --to /tmp/registry-auth/config.json

export REGISTRY_AUTH_FILE=/tmp/registry-auth/config.json
```

预期结果：

- Registry 凭证将写入 `/tmp/registry-auth/config.json`，并可通过 `REGISTRY_AUTH_FILE` 供后续直接镜像命令使用。

`--to` 和 `--registry-config`（或 `-a`）选择 `ac registry login` 的输出 auth 文件。

当 Registry 端点无法从当前计算机访问，但仍需要写入凭证文件时，使用 `--skip-check`。仅当端点使用 HTTP 或使用客户端不信任的证书时，才使用 `--insecure`。

管理 ImageStreams

创建空的 ImageStream：

```
ac create imagestream demo
```

创建启用本地镜像查找的 ImageStream：

```
ac create imagestream demo --lookup-local
```

从外部镜像创建 ImageStreamTag：

```
ac create imagestreamtag demo:latest \
  --from-image=source.example.com/source-team/source-app:1.0
```

将外部镜像导入 ImageStreamTag。如果不使用 `--confirm`，该命令会预览导入结果。添加 `--confirm` 以持久化该标签：

```
ac import-image demo:latest \
  --from=source.example.com/source-team/source-app:1.0 \
  --confirm
```

验证已持久化的 ImageStreamTag：

```
ac get imagestreamtags demo:latest -o yaml
```

预期结果：

- ImageStreamTag 包含已导入的镜像引用和摘要。

常用导入选项：

选项	用途
`--scheduled`	定期导入外部镜像或仓库。
`--insecure`	从未完全由受信任 HTTPS 保护的 Registry 导入。
`--reference-policy=source|local`	控制工作负载使用源引用还是本地 Registry 引用。
`--import-mode=PreserveOriginal`	当服务器支持此导入模式时，保留原始源引用。
`-o json|yaml`	以结构化格式打印导入结果。

将外部镜像标记到 ImageStreamTag：

```
ac tag source.example.com/source-team/source-app:1.0 demo:latest
```

将一个 ImageStreamTag 后的当前镜像复制到另一个 ImageStreamTag：

```
ac tag --source=imagestreamtag team-a/demo:stable team-b/demo:latest
```

使用标签选项控制跟踪和导入行为：

```
ac tag --source=imagestreamtag --alias team-a/demo:stable team-a/demo:lat
est
ac tag --source=imagestreamtag --reference team-a/demo:stable team-a/dem
o:latest
ac tag source.example.com/source-team/source-app:latest demo:latest \
  --scheduled \
  --insecure \
  --reference-policy=local \
  --import-mode=PreserveOriginal
```

`--alias` 要求源和目标位于同一个 ImageStream 中。`--scheduled` 和 `--insecure` 选项适用于外部镜像源。

删除标签：

```
ac tag -d demo:old
```

为一个 ImageStream 启用本地镜像查找：

```
ac set image-lookup demo
```

禁用本地镜像查找：

```
ac set image-lookup demo --enabled=false
```

列出本地查找状态：

```
ac set image-lookup --list
```

根据当前 ImageStreamTag 的拉取规范设置工作负载镜像：

```
ac set image deployment/my-app app=team-a/demo:latest --source=imagestrea
mtag -n team-a
```

根据 ImageStreamImage 摘要设置工作负载镜像：

```
ac set image deployment/my-app app=team-a/demo@sha256:<digest> \
  --source=imagestreamimage -n team-a
```

验证工作负载更新：

```
kubectl -n team-a rollout status deployment/my-app --timeout=300s
kubectl -n team-a get deployment my-app -o jsonpath='{.spec.template.spe
c.containers[?(@.name=="app")].image}{"\n"}'
```

预期结果：

- Deployment 使用从 ImageStream 资源解析出的拉取规范，并且滚动部署成功完成。

查询镜像和 **ImageStream**

列出 ImageStream：

```
ac get imagestreams -A
```

显示 ImageStream：

```
ac get imagestreams demo -n team-a -o yaml
```

显示某个标签的当前摘要和标签历史记录：

```
ac get imagestreamtags demo:latest -n team-a -o wide
```

按摘要显示 ImageStreamImage 元数据：

```
ac get imagestreamimages demo@sha256:<digest> -n team-a -o yaml
```

通过集成的 Registry 命令列出镜像：

```
ac get images
```

在 `modern` 模式下，`ac get images` 会列出集群范围的 Image 资源，并显示 `NAME` 和 `IMAGE REFERENCE` 列。 使用结构化输出：

```
ac get images -o yaml
ac get images -o json
```

删除镜像引用

首选的 Registry 工作流是删除 ImageStream 标签：

```
ac tag -d demo:old
```

`ac delete images` 命令遵循所选的 Registry 模式。除非设置了 `--confirm`，否则该命令以试运行方式执行：

```
ac delete images --repo team-a/demo:old
ac delete images --repo team-a/demo:old --confirm
```

使用多个 `--repo` 标志删除多个标签引用：

```
ac delete images \
  --repo team-a/demo:old \
  --repo team-a/api:canary \
  --confirm
```

删除镜像引用不一定会立即释放 blob 存储空间。需要回收存储空间时，请分别运行镜像清理和 Registry 垃圾回收。

直接处理 OCI 镜像

显示 Registry 中的镜像元数据：

```
ac image info registry.example.com/team-a/demo:latest \
  --registry-config="$REGISTRY_AUTH_FILE"
```

从多架构镜像中选择一个平台：

```
ac image info external-registry.example.com/base/base-app:latest \
  --filter-by-os=linux/arm64 \
  --registry-config="$REGISTRY_AUTH_FILE"
```

显示 manifest 列表条目：

```
ac image info registry.example.com/team-a/demo:latest \
  --show-multiarch \
  --registry-config="$REGISTRY_AUTH_FILE"
```

将一个镜像复制到另一个标签：

```
ac image mirror \
  registry.example.com/team-a/demo:latest \
  registry.example.com/team-a/demo:stable \
  --registry-config="$REGISTRY_AUTH_FILE"
```

根据映射文件复制多个镜像：

```
cat > mirror-map.txt <<EOF
source-registry.example.com/team-a/demo:latest=registry.example.com/team-a/demo:latest
source-registry.example.com/team-a/api:v1=registry.example.com/team-a/api:v1
EOF

ac image mirror \
  -f mirror-map.txt \
  --dry-run \
  --keep-manifest-list

ac image mirror \
  -f mirror-map.txt \
  --continue-on-error \
  --keep-manifest-list \
  --max-per-registry=4 \
  --skip-missing \
  --filter-by-os=linux/amd64 \
  --registry-config="$REGISTRY_AUTH_FILE"
```

如果找不到输入镜像，`--skip-missing` 会继续执行。`--filter-by-os` 选择请求的平台；当目标必须保持为包含所选平台条目的多架构 manifest 列表时，将其与 `--keep-manifest-list` 结合使用。

验证复制的镜像：

```
ac image info registry.example.com/team-a/demo:stable \
  --registry-config="$REGISTRY_AUTH_FILE"
```

向镜像添加层或元数据并推送结果：

```
ac image append \
  --from external-registry.example.com/base/base-app:latest \
  --to registry.example.com/team-a/base-app:patched \
  layer.tar.gz \
  --registry-config="$REGISTRY_AUTH_FILE"
```

预览追加操作：

```
ac image append \
  --from external-registry.example.com/base/base-app:latest \
  --to registry.example.com/team-a/base-app:patched \
  layer.tar.gz \
  --dry-run \
  --registry-config="$REGISTRY_AUTH_FILE"
```

`--image` 选项会将 JSON 对象合并到运行时镜像配置中。`--meta` 选项会将 JSON 对象合并到顶层镜像元数据中。也可以使用 `--drop-history` 或 `--created-at`，而不添加层：

```
ac image append \
  --from external-registry.example.com/base/base-app:latest \
  --to registry.example.com/team-a/base-app:annotated \
  --image='{"Entrypoint":["/app/start"]}' \
  --meta='{"author":"team-a"}' \
  --created-at=2024-01-01T00:00:00Z \
  --registry-config="$REGISTRY_AUTH_FILE"

ac image append \
  --from external-registry.example.com/base/base-app:latest \
  --to registry.example.com/team-a/base-app:no-history \
  --drop-history \
  --keep-manifest-list \
  --registry-config="$REGISTRY_AUTH_FILE"
```

预期结果：

- `ac image append` 会推送生成的镜像，或者使用 `--dry-run` 输出计划执行的操作。
- `--keep-manifest-list` 会保留多架构索引，同时将追加操作应用于匹配的平台镜像。

在没有本地容器运行时的情况下从镜像中提取文件：

```
ac image extract external-registry.example.com/base/base-app:latest \
  --path /:/tmp/base-app \
  --confirm \
  --registry-config="$REGISTRY_AUTH_FILE"
```

将选定的文件提取到当前目录：

```
ac image extract external-registry.example.com/base/base-app:latest \
  --file /etc/app/config.yaml \
  --registry-config="$REGISTRY_AUTH_FILE"
```

预览提取操作而不写入文件：

```
ac image extract external-registry.example.com/base/base-app:latest \
  --path /:/tmp/base-app \
  --dry-run \
  --registry-config="$REGISTRY_AUTH_FILE"
```

预期结果：

- `--file` 会将请求的文件写入当前目录。
- `--dry-run` 会列出将要提取的文件，但不会写入镜像内容。

仅在测试 Registry、HTTP 端点或客户端不信任的私有证书中，对 `ac image` 命令使用 `--insecure`。

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

管理通过代码创建的应用

主要功能

优势

前提条件

操作步骤

相关操作

实用指南

通过代码创建应用

前提条件

操作步骤

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。

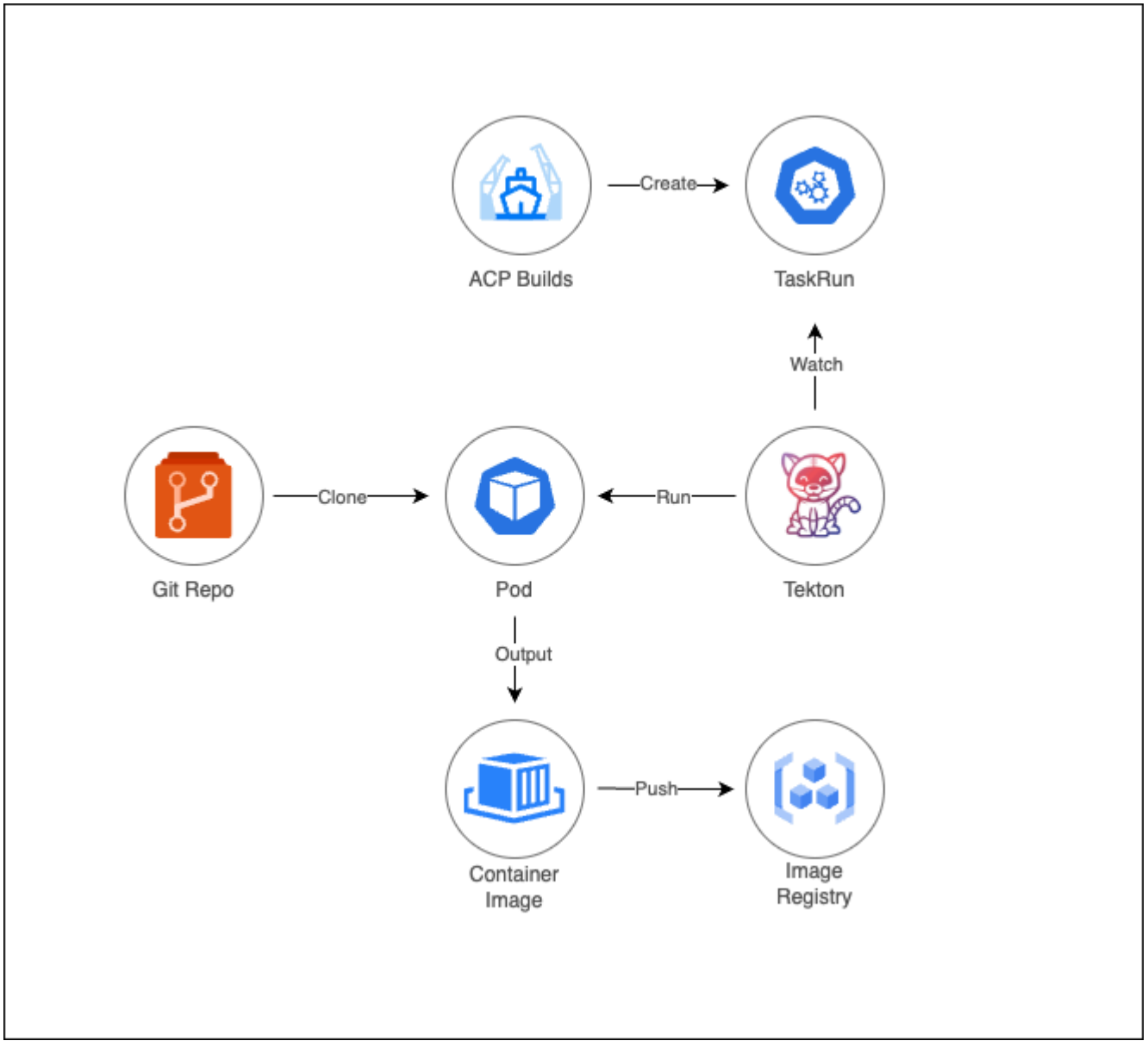
使用限制

> 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

管理通过代码创建的应用

主要功能

优势

前提条件

操作步骤

相关操作

管理通过代码创建的应用

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 过程，将源代码转换为镜像并将其发布为应用。
- 源代码更新后，通过可视化界面执行 **Rebuild** 操作，即可一键更新应用版本。

优势

- 简化通过代码创建和升级应用的过程。
- 降低开发人员的使用门槛，无需了解容器化的详细信息。
- 提供可视化的构建过程和运维管理，便于问题定位、分析和排障。

前提条件

- 已完成安装 Alauda Container Platform Builds。
- 需要访问镜像仓库。如果没有可用的仓库，请联系管理员准备 Registry。

操作步骤

1. 在 **Container Platform** 中，导航至 **Application** > **Application**。
2. 单击 **Create**。
3. 选择 **Create from Code**。
4. 参考以下参数说明完成配置。

地域	参数	描述
代码仓库	类型	• 平台集成：选择与平台集成且已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	管理员为当前项目分配或关联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。
	版本标识符	支持基于代码仓库中的分支、标签或提交创建应用。其中： • 当版本标识符为分支时，仅支持使用所选分支下的最新提交创建应用。 • 当版本标识符为标签或提交时，默认选择代码仓库中的最新标签或提交。但您也可以根据需要选择其他版本。
	上下文目录	源代码的可选目录，用作构建的上下文目录。

	Secret	使用输入的代码仓库时，可以根据需要添加身份验证 Secret。
	构建器镜像	• 包含特定编程语言运行时环境、依赖库和 S2I 脚本的镜像。其主要用途是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。
	版本	选择与源代码兼容的运行时环境版本，以确保应用顺利运行。
构建	构建类型	当前仅支持使用 **Build** 方法构建应用镜像。此方法简化并自动化了复杂的镜像构建过程，使开发人员能够专注于代码开发。常规流程如下： 1. 安装 Alauda Container Platform Builds 并创建 Shipyard 实例后，系统会自动生成 ClusterBuildStrategy 等集群级资源，并定义标准化的构建过程。此过程包括详细的构建步骤和必要的构建参数，从而支持 Source-to-Image (S2I) 构建。有关详细信息，请参阅：安装 Alauda Container Platform Builds 2. 根据上述策略和表单中提供的信息创建 Build 类型资源。这些资源指定构建策略、构建参数、源代码仓库、输出镜像仓库及其他相关信息。 3. 创建 BuildRun 类型资源以启动特定的构建实例，该实例负责协调整个构建过程。 4. 完成 BuildRun 创建后，系统会自动生成相应的 TaskRun 资源实例。此 TaskRun 实例会触发 Tekton 流水线构建，并创建 Pod 来执行构建过程。Pod 负责实际的构建工作，包括：从代码仓库拉取源代码。 调用指定的构建器镜像。

		执行构建过程。
	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。具体信息请参阅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。启用外部访问后，所有匹配的流量都将转发到此端口，以提供外部服务。 • 其他参数：请参阅创建 Ingress 文档中的参数说明。
标签和注释	-	根据需要填写相关标签和注释。

5. 填写参数后，单击 **Create**。
6. 您可以在 **Details** 页面查看相应的部署。

相关操作

构建

创建应用后，可以在详情页面查看相应信息。

参数	描述
构建	单击链接查看具体构建（Build）和构建任务（BuildRun）的资源信息及 YAML。
开始构建	构建失败或源代码发生更改时，可以单击此按钮重新执行构建任务。

实用指南

通过代码创建应用

前提条件

操作步骤

通过代码创建应用

> 使用 Alauda Container Platform Builds 安装的强大功能，实现从 Java 源代码创建应用的完整流程，最终使应用能够以容器化方式在 Kubernetes 上高效运行。

目录

前提条件

操作步骤

前提条件

使用此功能前，请确保：

- 安装 Alauda Container Platform Builds
- 平台上存在可访问的镜像仓库。如果没有可用的仓库，请联系管理员准备 Registry。

操作步骤

1. 在容器平台中，单击应用 > 应用。
2. 单击创建。
3. 选择从代码创建。

4. 根据以下参数完成配置：

参数	推荐配置
代码仓库	类型：`Input` 仓库 **URL**：`https://github.com/alauda/spring-boot-hello-world`
构建方法	`Build`
镜像仓库	联系管理员。
原生应用	原生应用：`spring-boot-hello-world` 名称：`spring-boot-hello-world` 资源限制：使用默认值。
网络	目标端口：`8080`

5. 填写参数后，单击创建。

6. 您可以在详情页面查看相应的应用状态。

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

使用 CLI 管理节点隔离策略

- 节点隔离策略的工作原理
- 开始之前
- 向 ACP 进行身份验证
- 查看节点隔离策略
- 检查可用的项目和节点
- 创建节点隔离策略
- 更新节点隔离策略
- 删除节点隔离策略
- 验证调度行为
- 故障排除
- 推荐操作流程

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

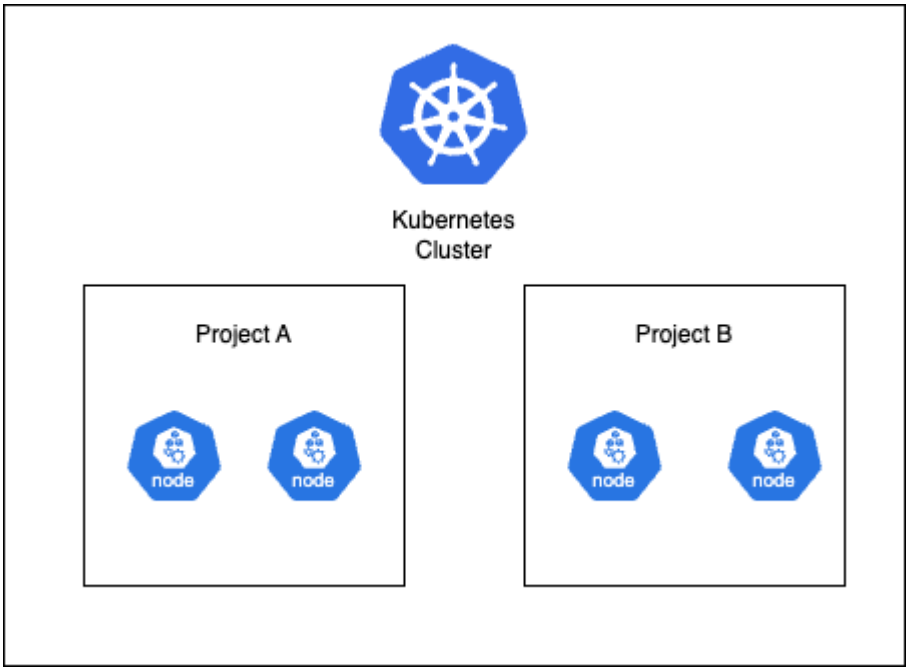
优势

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。

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

使用 CLI 管理节点隔离策略

节点隔离策略的工作原理

开始之前

向 ACP 进行身份验证

查看节点隔离策略

检查可用的项目和节点

创建节点隔离策略

更新节点隔离策略

删除节点隔离策略

验证调度行为

故障排除

推荐操作流程

创建节点隔离策略

为当前集群创建节点隔离策略，使指定项目能够独占访问集群中分组资源的节点，从而限制项目下 Pod 可运行的节点，实现项目之间的物理资源隔离。

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. 在左侧导航栏中，点击 安全 > 节点隔离策略。
2. 点击 创建节点隔离策略。
3. 参考以下说明配置相关参数。

参数	描述
项目独占性	是否启用策略中配置的项目隔离策略所包含节点的开关；点击即可开启或关闭，默认为开启。 开关开启时，策略中指定项目下的 Pod 才能在策略所包含的节点上运行；开关关闭时，除指定项目外，当前集群中其他项目下的 Pod 也可以在策略所包含的节点上运行。

参数	描述
项目	配置为使用策略中节点的项目。 点击 项目 下拉选择框，勾选项目名称前的复选框即可选择多个项目。 注意： 一个项目只能设置一个节点隔离策略；如果项目已分配节点隔离策略，则无法选择； 支持在下拉选择框中输入关键字，以筛选和选择项目。
节点	策略中分配给项目使用的工作节点 IP 地址。 点击 节点 下拉选择框，勾选节点名称前的复选框即可选择多个节点。 注意： 一个节点只能属于一个隔离策略；如果节点已属于其他隔离策略，则无法选择； 支持在下拉选择框中输入关键字，以筛选和选择节点。

4. 点击 创建。
   注意：

   - 创建策略后，项目中不符合当前策略的现有 Pod 在重建后将被调度到当前策略所包含的节点上；
   - 当 项目独占性 开启时，节点上当前已存在的 Pod 不会被自动驱逐；如需驱逐，则必须手动进行调度。

删除节点隔离策略

注意：删除节点隔离策略后，项目将不再被限制只能在指定节点上运行，节点也不再由该项目独占使用。

1. 在左侧导航栏中，点击 安全 > 节点隔离策略。
2. 找到节点隔离策略，点击⋮> 删除。

使用 CLI 管理节点隔离策略

节点隔离策略提供项目级别的节点隔离。平台管理员可以将一个或多个项目绑定到选定的集群节点，并选择这些节点是仅供绑定的项目使用，还是与其他项目共享。

如果所使用的版本中的 Web 控制台不再提供管理员 > 安全设置 > 节点隔离策略，您可以继续使用 `ac` 或 `kubectl` 管理节点隔离策略。

目录

- 节点隔离策略的工作原理
- 开始之前
- 向 ACP 进行身份验证
- 查看节点隔离策略
- 检查可用的项目和节点
- 创建节点隔离策略
- 更新节点隔离策略
  - 更新显示名称
  - 更改共享模式
  - 替换绑定的项目
  - 替换绑定的节点
- 删除节点隔离策略
- 验证调度行为
  - 验证标签
  - 验证新 Pod 的调度

故障排除

推荐操作流程

节点隔离策略的工作原理

节点隔离策略以名为 `NodeGroup` 的 Kubernetes 自定义资源形式存储。

项目	值
API 组	`security.alauda.io`
API 版本	`v1beta1`
Kind	`NodeGroup`
资源	`nodegroups.security.alauda.io`
作用域	集群

关键字段如下：

字段	描述
`metadata.name`	策略名称。必须以小写字母或数字开头和结尾，可以包含小写字母、数字和 `-`，长度最多为 63 个字符。
`metadata.annotations["alauda.io/display-name"]`	可选的显示名称。
`spec.shareMode`	节点共享模式。有效值为 `Exclusive` 和 `Share`。
`spec.projects[].name`	使用该策略的项目名称。
`spec.nodes[].name`	分配给该策略的节点名称。

字段	描述
`status.phase`	策略协调状态，例如 `Active`。

共享模式如下：

- `Exclusive`：只有绑定到该策略的项目才能在策略节点上调度新 Pod。其他项目无法在这些节点上调度新 Pod。
- `Share`：绑定到该策略的项目仅限使用策略节点，但其他项目仍可使用这些节点。

创建或更新策略后，平台会为相关的 `Node` 和 `Namespace` 资源添加标签，在 `ProjectQuota` 对象存在时为其添加标签，并在创建或更新 Pod 时注入调度约束。常见的标签和污点包括：

类型	名称	描述
标签	`acp.cpaas.io/node-group-name`	节点隔离策略名称。
标签	`acp.cpaas.io/node-group-share-mode`	节点隔离模式，通常为 `Exclusive` 或 `Share`。
污点	`nodeGroup=<policy-name>:NoSchedule`	在 `Exclusive` 模式下添加到策略节点，用于阻止不容忍该污点的 Pod。

开始之前

运行本指南中的命令前，请确认以下事项：

- 已安装 `ac` CLI。如果使用 `kubectl`，请先为目标集群准备 kubeconfig。
- 您的账户可以管理 `nodegroups.security.alauda.io`。
- 您知道目标业务集群的名称，例如 `business-1`。
- 您知道要绑定的项目名称和节点名称。
- 您已评估对现有工作负载的影响。

重要限制：

- `NodeGroup` 是集群范围的资源。管理 `NodeGroup` 资源时，不要指定 `-n`。
- 在一个业务集群中，每个项目只能属于一个节点隔离策略。
- 在一个业务集群中，每个节点只能属于一个节点隔离策略。
- 创建策略不会迁移现有 Pod。对于不符合策略的现有 Pod，只有在重新创建后才会受到约束。
- 在 `Exclusive` 模式下，策略节点上的现有 Pod 不会自动驱逐。如有需要，请手动迁移。
- Web 控制台创建页面会过滤掉控制平面节点，并禁用已被其他策略使用的项目和节点。使用 CLI 时，请在创建或更新策略前自行检查冲突。

向 ACP 进行身份验证

使用 `ac` 管理节点隔离策略。`ac` CLI 用于管理 ACP 多集群会话，并可切换到目标业务集群。

```
ac login <acp-url> --name <session-name>
ac config get-clusters
ac config use-cluster <cluster-name>
ac config current-context
```

示例：

```
ac login https://acp.example.com --name prod
ac config use-cluster business-1
ac config current-context
```

确认 `NodeGroup` CRD 存在：

```
ac api-resources --api-group=security.alauda.io
ac get crd nodegroups.security.alauda.io
```

如果您更倾向于使用 `kubectl`，请从 `ac` 导出 kubeconfig，然后选择目标 context。将 `kubectl config get-contexts` 中 `NAME` 列的值用作 `<target-context>`。

```
ac config view --raw > kubeconfig
export KUBECONFIG=./kubeconfig
kubectl config get-contexts
kubectl config use-context <target-context>
kubectl get nodegroups.security.alauda.io
```

以下示例使用 `ac`。如果您的 `kubectl` context 配置正确，可以将 `ac` 替换为 `kubectl`。

查看节点隔离策略

查看当前集群中的所有节点隔离策略：

```
ac get nodegroups.security.alauda.io
```

查看包含所选字段的策略列表：

```
ac get nodegroups.security.alauda.io \
  -o 'custom-columns=NAME:.metadata.name,MODE:.spec.shareMode,PROJECTS:.s
pec.projects[*].name,NODES:.spec.nodes[*].name,PHASE:.status.phase'
```

查看某个策略的详细信息：

```
ac get nodegroups.security.alauda.io <policy-name> -o yaml
ac describe nodegroups.security.alauda.io <policy-name>
```

检查策略是否已调谐：

```
ac get nodegroups.security.alauda.io <policy-name> \
  -o jsonpath='{.status.phase}{"\n"}'
```

当 `status.phase` 为 `Active` 时，控制器已调谐该策略。

检查可用的项目和节点

创建或更新策略之前，请检查这些项目和节点是否已被其他策略使用。

如果环境中存在 `ProjectQuota` 对象，请查看其节点隔离标签：

```
ac get projectquotas.auth.alauda.io \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

查看未绑定到策略的 `ProjectQuota` 对象：

```
ac get projectquotas.auth.alauda.io \
  --selector='!acp.cpaas.io/node-group-name'
```

查看节点及其节点隔离标签：

```
ac get nodes \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

查看未绑定到策略的工作节点：

```
ac get nodes \
  --selector='!node-role.kubernetes.io/master,!node-role.kubernetes.io/co
ntrol-plane,!acp.cpaas.io/node-group-name'
```

创建节点隔离策略

创建 `nodegroup-pro.yaml` ：

```yaml
apiVersion: security.alauda.io/v1beta1
kind: NodeGroup
metadata:
  name: pro-nodegroup
  annotations:
    alauda.io/display-name: "pro nodes"
spec:
  shareMode: Exclusive
  projects:
    - name: pro
  nodes:
    - name: node-1
    - name: node-2
```

当绑定的项目必须独占节点时，将 `spec.shareMode` 设置为 `Exclusive`。当绑定的项目必须在选定节点上运行、但其他项目仍可使用这些节点时，将其设置为 `Share`。

应用策略：

```
ac apply -f nodegroup-pro.yaml
```

确认策略状态：

```
ac get nodegroups.security.alauda.io pro-nodegroup
ac get nodegroups.security.alauda.io pro-nodegroup -o yaml
```

确认节点已添加标签：

```
ac get nodes \
  --selector='acp.cpaas.io/node-group-name=pro-nodegroup' \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

对于 `Exclusive` 策略，确认策略节点具有 `nodeGroup` 污点：

```
ac describe node node-1
```

在输出中，检查 `Taints` 是否包含：

```
nodeGroup=pro-nodegroup:NoSchedule
```

共享节点通常不会保留 `nodeGroup=<policy-name>:NoSchedule` 污点。

更新节点隔离策略

更新策略之前，请检查其当前配置：

```
ac get nodegroups.security.alauda.io <policy-name> -o yaml
```

更新显示名称

```
ac annotate nodegroups.security.alauda.io <policy-name> \
  alauda.io/display-name="<display-name>" \
  --overwrite
```

更改共享模式

将策略切换为共享模式：

```
ac patch nodegroups.security.alauda.io <policy-name> \
  --type=merge \
  -p '{"spec":{"shareMode":"Share"}}'
```

将策略切换为独占模式：

```
ac patch nodegroups.security.alauda.io <policy-name> \
  --type=merge \
  -p '{"spec":{"shareMode":"Exclusive"}}'
```

策略切换为 `Exclusive` 后，控制器会向策略节点添加 `nodeGroup=<policy-name>:NoSchedule`。现有 Pod 不会因该污点被驱逐，但新 Pod 会受到约束。

替换绑定的项目

以下命令会将整个项目列表替换为 `pro` 和 `dev`：

```
ac patch nodegroups.security.alauda.io <policy-name> \
  --type=merge \
  -p '{"spec":{"projects":[{"name":"pro"},{"name":"dev"}]}}'
```

替换绑定的节点

以下命令会将整个节点列表替换为 `node-1` 和 `node-2`：

```
ac patch nodegroups.security.alauda.io <policy-name> \
  --type=merge \
  -p '{"spec":{"nodes":[{"name":"node-1"},{"name":"node-2"}]}}'
```

> 重要：
> `spec.projects` 和 `spec.nodes` 是数组。合并补丁会替换整个数组。请在补丁中包含完整的最终列表，以避免意外移除项目或节点。

删除节点隔离策略

删除策略后，绑定的项目将不再受限于该策略的节点，这些节点也不再为该策略保留。

```
ac delete nodegroups.security.alauda.io <policy-name>
```

确认策略已删除：

```
ac get nodegroups.security.alauda.io
```

确认相关标签已清理：

```
ac get nodes \
  --selector='acp.cpaas.io/node-group-name=<policy-name>'

ac get projectquotas.auth.alauda.io \
  --selector='acp.cpaas.io/node-group-name=<policy-name>'

ac get namespaces \
  --selector='acp.cpaas.io/node-group-name=<policy-name>'
```

如果策略长时间处于删除状态，请勿直接移除其 finalizer。请先检查节点隔离控制器和 webhook。

验证调度行为

创建或更新策略后，请验证策略是否已生效。

验证标签

查看策略节点：

```
ac get nodes \
  --selector='acp.cpaas.io/node-group-name=<policy-name>' \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

如果环境中存在 `ProjectQuota` 对象，请查看策略项目配额：

```
ac get projectquotas.auth.alauda.io \
  --selector='acp.cpaas.io/node-group-name=<policy-name>' \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

查看属于策略项目的命名空间：

```
ac get namespaces \
  --selector='acp.cpaas.io/node-group-name=<policy-name>' \
  -L acp.cpaas.io/node-group-name,acp.cpaas.io/node-group-share-mode
```

验证新 **Pod** 的调度

在属于绑定项目的命名空间中创建测试 Pod：

```
ac -n <namespace> run nodegroup-check \
  --image=busybox:1.36 \
  --restart=Never \
  --command -- sleep 3600
```

查看 Pod 被调度到的节点：

```
ac -n <namespace> get pod nodegroup-check -o wide
```

查看注入的节点选择器和容忍度：

```
ac -n <namespace> get pod nodegroup-check -o yaml
```

在输出中检查以下字段：

```
spec:
  nodeSelector:
    acp.cpaas.io/node-group-name: <policy-name>
  tolerations:
    - key: nodeGroup
      operator: Equal
      value: <policy-name>
      effect: NoSchedule
```

验证后删除测试 Pod：

```
ac -n <namespace> delete pod nodegroup-check
```

> 注意：
> 现有 Pod 只有在重新创建后才会再次经过调度流程。

故障排除

问题	可能原因	操作
`nodegroups.security.alauda.io` 不存在	目标集群未安装或启用节点隔离组件。	确认选择了正确的集群，然后检查节点隔离功能开关和相关组件。
创建或更新被拒绝	项目或节点已属于其他策略，或者你的帐户没有相应权限。	检查 `acp.cpaas.io/node-group-name` 标签并确认你的权限。
策略未变为 `Active`	控制器尚未完成协调，或者引用的项目或节点不存在。	检查项目和节点名称，然后查看节点隔离控制器日志。
新 Pod 未调度到策略节点	命名空间未添加标签、Pod 未重新创建，或者 webhook 未注入调度约束。	检查 Namespace 标签，删除并重新创建 Pod，然后检查 webhook 和控制器。
其他项目的现有 Pod 仍留在独占节点上	这是预期行为。`NoSchedule` 不会驱逐现有 Pod。	如果需要迁移，请手动重新创建或驱逐相关工作负载，或调整相关工作负载。
更新节点列表后节点标签未发生变化	控制器仍在同步，或者策略运行状况不佳。	稍等片刻后重新查询。如果标签仍未更新，请检查控制器日志和 `NodeGroup` 事件。

推荐操作流程

对于生产环境操作，请使用以下流程：

1. 确认目标业务集群上下文。
2. 查询现有的 `NodeGroup` 列表。
3. 检查项目和节点是否已绑定到其他策略。
4. 编写并保存 `NodeGroup` YAML。
5. 使用 `ac apply -f` 创建或更新策略。
6. 等待 `status.phase` 变为 `Active`。
7. 相关资源存在时，检查 `Node`、`Namespace` 和 `ProjectQuota` 标签。
8. 在测试命名空间中创建新的测试 Pod，以验证调度。
9. 仅在验证完成后重新创建业务工作负载或迁移现有 Pod。

权限

功能	操作	平台管理员	平台审计人员	项目管理员	命员
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：

```
---
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：

```
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。