Menu

开发者

快速开始

Creating a simple application via image

Introduction

Important Notes

Prerequisites

Workflow Overview

Procedure

构建应用

Build application architecture

Introduction to build application

Core components

核心概念

命名空间

创建应用

应用的操作与维护

计算组件

使用 **Helm charts**

1. 了解 Helm

2 通过 CLI 将 Helm Charts 部署为 Applications

3 通过 UI 将 Helm Charts 部署为 Applications

配置

应用可观测

实用指南

镜像

镜像概述

理解容器和镜像

镜像

镜像仓库

镜像库

镜像标签

镜像 ID

容器

实用指南

镜像仓库

介绍

原则与命名空间隔离

认证与授权

优势

应用场景

安装

使用指南

S2I

概览

安装

升级

功能指南

How To

节点隔离策略

引言

优势

应用场景

架构

概念

功能指南

权限说明

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？

Menu

快速开始

Creating a simple application via image

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

目录

Introduction

Use Cases

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

Time Commitment

预计完成时间：10-15 分钟

Important Notes

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

Prerequisites

- 具备 Kubernetes 架构及 灵雀云容器平台 平台基础概念
- 按平台搭建流程预先创建项目

Workflow Overview

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

Procedure

Create namespace

> 命名空间提供资源分组和配额管理的逻辑隔离。

Prerequisites

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

Creation Process

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

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

4. 默认资源限制下完成创建

Configure Image Repository

灵雀云容器平台 支持多种镜像来源策略：

方法一：通过工具链集成注册表

1. 进入 管理员 > 工具链 > 集成
2. 新建集成：

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

3. 将注册表分配至目标平台项目

方法二：外部注册表服务

- 使用公开可访问的注册表地址（如 Docker Hub）
- 示例：`index.docker.io/library/nginx:latest`

验证要求

- 集群网络需具备访问注册表端点的出站权限

Create application via Deployment

> Deployment 提供 Pod 副本集的声明式更新。

Creation Process

1. 在 容器平台视图中：
   - 使用命名空间选择器选择目标隔离边界
2. 进入 工作负载 > **Deployments**

3. 点击 创建 **Deployment**
4. 指定镜像来源：
   - 选择集成注册表 *或*
   - 输入外部镜像地址（如 `index.docker.io/library/nginx:latest`）
5. 配置工作负载身份并启动

Management Operations

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

Expose Service via NodePort

服务使 Pod 组具备网络访问能力。

Creation Process

1. 进入 网络 > 服务
2. 点击 创建服务，配置参数：

参数	值
Type	NodePort
Selector	目标 Deployment 名称
Port Mapping	服务端口:容器端口 （如 8080:80 ）

3. 确认创建。

关键点

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

内部路由通过提供统一 IP 地址或主机端口，实现工作负载的服务发现和访问。

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

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

4. 点击 创建，服务创建成功。

Validate Application Accessibility

Verification Method

1. 获取暴露的端点信息：
   - **Node IP**：工作节点公网地址
   - **NodePort**：分配的外部端口
2. 构造访问 URL：`http://<Node_IP>:<NodePort>`
3. 预期结果：显示 Nginx 欢迎页面

Welcome to nginx!

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

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

Thank you for using nginx.

Menu

构建应用

Build application architecture

Build application architecture

Introduction to build application

Core components

核心概念

应用类型

Custom Applications

理解自定义应用

自定义应用 CRD 架构设计

Workload Types

理解参数

Overview

Core Concepts

使用场景与示例

CLI 示例与实用用法

最佳实践

常见问题排查

高级使用模式

理解环境变量

Overview

Core Concepts

使用场景与示例

CLI 示例与实际使用

最佳实践

理解启动命令

Overview

Core Concepts

用例与场景

CLI 示例与实际使用

最佳实践

高级使用模式

资源单位说明

命名空间

创建命名空间

理解命名空间

通过 Web 控制台创建命名空间

通过 CLI 创建命名空间

导入 Namespace

Overview

Use Cases

Prerequisites

Procedure

资源配额

理解资源请求与限制

配额

硬件加速器资源配额

Limit Range

理解 Limit Range

使用 CLI 创建 Limit Range

Pod Security Admission

Security Modes

Security Standards

Configuration

UID/GID 分配

启用 UID/GID 分配

验证 UID/GID 分配

Overcommit Ratio

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

管理命名空间成员

导入成员

添加成员

移除成员

更新命名空间

更新配额

更新容器 LimitRanges

更新 Pod Security Admission

删除/移除命名空间

删除命名空间

移除命名空间

创建应用

Creating applications from Image

Prerequisites

Procedure 1 - Workloads

Procedure 2 - Services

Procedure 3 - Ingress

应用管理操作

参考信息

Creating applications from Chart

注意事项

前提条件

操作步骤

状态分析参考

通过 YAML 创建应用

注意事项

前提条件

操作步骤

通过代码创建应用

前提条件

操作步骤

Creating applications from Operator Backed

了解 Operator Backed 应用

通过 Web 控制台创建 Operator Backed 应用

故障排查

通过 CLI 工具创建应用

前提条件

操作步骤

示例

参考

应用的操作与维护

Application Rollout

状态说明

Applications

KEDA(Kubernetes Event-driven Autoscaling)

配置 HPA

了解水平 Pod 自动扩缩器

前提条件

创建水平 Pod 自动扩缩器

计算规则

启动和停止原生应用

启动原生应用

停止原生应用

配置 VerticalPodAutoscaler (VPA)

了解 VerticalPodAutoscalers

前提条件

创建 VerticalPodAutoscaler

后续操作

配置 CronHPA

了解 Cron Horizontal Pod Autoscalers

前提条件

创建 Cron Horizontal Pod Autoscaler

调度规则说明

更新原生应用

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）

更新和删除 **Chart** 应用

重要说明

前提条件

状态分析说明

应用版本管理

创建版本快照

回滚到历史版本

删除原生应用

处理资源耗尽错误

Overview

配置驱逐策略

在节点配置中创建驱逐策略

驱逐信号

驱逐阈值

配置可调度资源

防止节点状态振荡

回收节点级资源

Pod 驱逐

服务质量与内存杀手（OOM Killer）

调度器与资源耗尽状态

示例场景

推荐实践

健康检查

理解健康检查

YAML 文件示例

通过 Web 控制台配置健康检查参数

探针失败故障排查

计算组件

Deployments

理解 Deployments

创建 Deployments

管理 Deployments

使用 CLI 进行故障排查

DaemonSets

理解守护进程集

创建守护进程集

管理守护进程集

StatefulSets

理解 StatefulSets

创建 StatefulSets

管理 StatefulSets

CronJobs

理解 CronJobs

创建 CronJobs

立即执行

删除 CronJobs

任务

了解任务

YAML 文件示例

执行概览

Pods

理解 Pods

YAML 文件示例

使用 CLI 管理 Pod

使用 Web 控制台管理 Pod

Containers

理解 Containers

理解 Ephemeral Containers

与 Containers 交互

使用 **Helm charts**

使用 Helm charts

1. 了解 Helm

2 通过 CLI 将 Helm Charts 部署为 Applications

3 通过 UI 将 Helm Charts 部署为 Applications

配置

Configuring ConfigMap

Understanding Config Maps

Config Map 限制

ConfigMap 示例

通过 Web 控制台创建 ConfigMap

通过 CLI 创建 ConfigMap

操作

通过 CLI 查看、编辑和删除

Pod 中使用 ConfigMap 的方式

ConfigMap 与 Secret 的对比

Configuring Secrets

理解 Secrets

创建 Opaque 类型的 Secret

创建 Docker registry 类型的 Secret

创建 Basic Auth 类型的 Secret

创建 SSH-Auth 类型的 Secret

创建 TLS 类型的 Secret

通过 Web 控制台创建 Secret

如何在 Pod 中使用 Secret

后续操作

操作

应用可观测

监控面板

前置条件

命名空间级别监控面板

工作负载级别监控

Logs

操作步骤

实时事件

操作步骤

事件记录解读

实用指南

设置定时任务触发规则

转换时间

编写 Crontab 表达式

Build application architecture

目录

Introduction to build application

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

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

Core components

Archon

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

Metis

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

- **Webhook** 管理：实现资源校验的准入 webhook，包括 `resources ratio` 强制执行和 `resource labeling` 策略等。
- 状态同步：通过以下方式维护分布式组件的一致性：
  - `Helm chart application` 状态调和
  - `Project quota` 同步
  - `Application` 状态更新（写入 application.status 字段）

Captain controller manager

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

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

Icarus

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

- 提供全面的监控面板视图，用于集群健康监控
- 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

    - 提供租户特定的视图隔离
- 仅运行在 `global cluster` 上，作为多集群操作的统一控制点

Menu

核心概念

应用类型

Custom Applications

理解自定义应用

自定义应用 CRD 架构设计

Workload Types

理解参数

Overview

Core Concepts

使用场景与示例

CLI 示例与实用用法

最佳实践

常见问题排查

高级使用模式

理解环境变量

Overview

Core Concepts

使用场景与示例

CLI 示例与实际使用

最佳实践

理解启动命令

Overview

Core Concepts

用例与场景

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）以及其他原生 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 编排和模板化部署，适用于开发、测试和生产环境。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 是预先配置好的 Kubernetes 资源包，可以作为一个整体进行部署，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、升级和维护服务，因为 Operator 作为针对特定应用的智能控制器发挥作用。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

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。用户可以通过标准的 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管理的场景。

核心能力

1. 声明式 **API** 驱动管理

- 通过 Application CRD 将分布式资源（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和协调。

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	将分散的资源（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 YAML 碎片化带来的管理混乱。
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 representation
            of an object. Servers should convert recognized schemas to the latest
            internal value, and may reject unrecognized values. More info:
https://github.com/kubernetes/community/blob/master/contributors/devel/sig-
architecture/api-conventions.md#resources'
          type: string
        kind:
          description: 'Kind is a string value representing the REST resource this
            object represents. Servers may infer this from the endpoint the client
            submits requests to. Cannot be updated. In CamelCase. More info:
https://github.com/kubernetes/community/blob/master/contributors/devel/sig-
architecture/api-conventions.md#types-kinds'
          type: string
        metadata:
          description: 'Metadata is a object value representing the metadata of the
kubernetes resource.
            More info:
https://github.com/kubernetes/community/blob/master/contributors/devel/sig-
architecture/api-conventions.md#metadata'
          type: object
        spec:
          properties:
            assemblyPhase:
              description: |
                The installer can set this field to indicate that the application's
```

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

```
                  description: 'A list of keywords that identify the application.'
                  items:
                    type: string
                  type: array
                links:
                  description: 'Links are a list of descriptive URLs intended 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 Application. Each
maintainer has a
                    name, email, and URL. This field is meant for the distributors 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 intended as a quick
start
                    for the users of the Application. They may be plain text 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. WordPress, 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 resource.'
                  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 reference.'
                        type: string
                      path:
                        description: 'The path of the ingress reference.'
                        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 reference.'
                        type: string
                      port:
                        description: 'The port of the service reference.'
                        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-app"}]} should be
used as the selector
                for an Application named "my-cool-app", and each component should contain
a label that matches.'
              type: object
          type: object
        status:
          description: 'The status summarizes the current state of the object.'
          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。

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: applicationhistories.app.k8s.io
spec:
  group: app.k8s.io
  names:
    kind: ApplicationHistory
    listKind: ApplicationHistoryList
    plural: applicationhistories
    singular: applicationhistory
  scope: Namespaced
  validation:
    openAPIV3Schema:
      properties:
        apiVersion:
          description: 'APIVersion defines the versioned schema of this representation
            of an object. Servers should convert recognized schemas to the latest
            internal value, and may reject unrecognized values. More info:
https://github.com/kubernetes/community/blob/master/contributors/devel/sig-
architecture/api-conventions.md#resources'
          type: string
        kind:
          description: 'Kind is a string value representing the REST resource this
            object represents. Servers may infer this from the endpoint the client
            submits requests to. Cannot be updated. In CamelCase. More info:
https://github.com/kubernetes/community/blob/master/contributors/devel/sig-
architecture/api-conventions.md#types-kinds'
          type: string
        metadata:
          description: 'Metadata is a object value representing the metadata of the
kubernetes resource.
            More info:
https://github.com/kubernetes/community/blob/master/contributors/devel/sig-
architecture/api-conventions.md#metadata'
          type: object
        spec:
          properties:
            changeCause:
              description: 'The change cause of the application to generate the
ApplicationHistory.'
              type: string
            creationTimestamp:
```

```
              description: 'The creation timestamp of the application history.'
              format: date-time
              type: string
            resourceDiffs:
              description: 'The resource differences between the current and last version
of application. It contains 3 types of diff: `create`,
                `delete` and `update`. The item of the diff compose of the kind and name
of the diff resource object.'
              properties:
                create:
                  items:
                    properties:
                      kind:
                        description: 'The kind of the created resource.'
                        type: string
                      name:
                        description: 'The name of the created resource.'
                        type: string
                    type: object
                  type: array
                delete:
                  items:
                    properties:
                      kind:
                        description: 'The kind of the deleted resource.'
                        type: string
                      name:
                        description: 'The name of the deleted resource.'
                        type: string
                    type: object
                  type: array
                update:
                  items:
                    properties:
                      kind:
                        description: 'The kind of the updated resource.'
                        type: string
                      name:
                        description: 'The name of the updated resource.'
                        type: string
                    type: object
                  type: array
              type: object
            revision:
```

```
              description: |
                The revision number of the application history. It's an integer that will
be incremented on
                every change of the application.'
              type: integer
            user:
              description: 'The user name who triggered the change of the application.'
              type: string
            yaml:
              description: |
                The YAML string of the snapshot of the application and it's components.
              type: string
          type: object
        status:
          description: 'The status summarizes the current state of the object.'
          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
      type: object
  version: v1beta1
  versions:
  - name: v1beta1
    served: true
    storage: true
```

Workload Types

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。当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，适合周期性任务，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

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，它们：

- 覆盖 Docker 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 Docker 的关系

在 Docker 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"]
```

使用场景与示例

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. 针对不同环境的部署

为不同环境设置不同参数：

```
# 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 示例与实用用法

使用 kubectl run

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

# 多参数传递
kubectl run myapp --image=myapp:latest --restart=Never -- --port=8080 --log-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 yaml >
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
```

多参数的应用

```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`，避免使用 `-p 8080`
- 提供合理的默认值：确保应用在无参数时也能正常运行
- 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

```
# ✅ 结合 ConfigMaps 使用参数
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：检查参数语法和应用文档
```

2. 参数覆盖无效

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

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

3. 调试参数问题

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

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

高级使用模式

1. 使用 Init 容器实现条件参数

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

2. 使用 **Helm** 进行参数模板化

```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，适应不同环境和需求。

理解环境变量

目录

Overview

Kubernetes 中的环境变量是以键值对形式存在的，用于在容器运行时提供配置信息。它们为向应用注入配置信息、密钥和运行时参数提供了一种灵活且安全的方式，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：

- 可供容器内运行进程使用的键值对
- 一种无需重建镜像的运行时配置机制
- 向应用传递配置信息的标准方式
- 可通过任何编程语言的标准操作系统 API 访问

Kubernetes 中环境变量的来源

Kubernetes 支持多种环境变量来源：

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

环境变量优先级

环境变量的覆盖顺序如下：

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）

使用场景与示例

1. 应用配置

基础应用设置：

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

2. 数据库配置

使用 ConfigMaps 和 Secrets 配置数据库连接：

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

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

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

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

3. 动态运行时信息

访问 Pod 和 Node 元数据：

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

4. 不同环境的配置

针对不同环境的配置示例：

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

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

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

CLI 示例与实际使用

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. 环境变量命名规范

```
# ✅ 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                # 过短
# - name: databaseHost      # 命名风格不一致
# - name: log-level         # 分隔符不一致
```

4. 默认值与校验

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    env:
    - name: PORT
      value: "8080"          # 提供合理的默认值
    - 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
```

理解启动命令

目录

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

Overview

Kubernetes 中的启动命令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。启动命令提供了对容器内运行进程的完全控制。

Core Concepts

什么是启动命令？

启动命令是：

- 容器启动时运行的主要可执行文件
- 覆盖 Docker 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 Docker 及参数的关系

理解 Docker 指令与 Kubernetes 字段之间的关系：

Docker	Kubernetes	作用
ENTRYPOINT	`command`	定义可执行文件
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	**Docker** 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

用例与场景

1. 自定义应用启动

使用相同基础镜像运行不同应用：

```yaml
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. 调试与故障排查

覆盖默认命令以启动 shell 进行调试：

```yaml
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

使用同一镜像满足不同用途：

```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 3600"

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

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

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

使用 kubectl create job

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

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

复杂启动命令示例

多步骤初始化

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

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

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

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

条件启动逻辑

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

最佳实践

1. 信号处理与优雅关闭

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

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

      # 等待进程结束
      wait $PID
```

2. 错误处理与日志记录

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

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

      log "Starting application initialization..."

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

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

3. 安全性考虑

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

4. 资源管理

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

高级使用模式

1. 带自定义命令的 Init 容器

```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
```

启动命令为 Kubernetes 中的容器执行提供了完全的控制。通过理解如何正确配置和使用启动命令，您可以创建灵活、易维护且健壮的容器化应用，以满足您的特定需求。

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）：该参数仅在集群下存在 GPU 资源时生效。虚拟 GPU 核心数；100 个虚拟核心等于 1 个物理 GPU 核心。支持正整数。
- Video Memory（可选）：该参数仅在集群下存在 GPU 资源时生效。虚拟 GPU 显存；1 个显存单位等于 256 Mi。支持正整数。

Menu

命名空间

创建命名空间

理解命名空间

通过 Web 控制台创建命名空间

通过 CLI 创建命名空间

导入 **Namespace**

Overview

Use Cases

Prerequisites

Procedure

资源配额

理解资源请求与限制

配额

硬件加速器资源配额

Limit Range

理解 Limit Range

使用 CLI 创建 Limit Range

Pod Security Admission

Security Modes

Security Standards

Configuration

UID/GID 分配

启用 UID/GID 分配

验证 UID/GID 分配

Overcommit Ratio

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

管理命名空间成员

导入成员

添加成员

移除成员

更新命名空间

更新配额

更新容器 LimitRanges

更新 Pod Security Admission

删除/移除命名空间

删除命名空间

移除命名空间

创建命名空间

目录

理解命名空间

参考官方 Kubernetes 文档：Namespaces ↗

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

通过 **Web** 控制台创建命名空间

> 在与项目关联的集群内，创建一个新的命名空间，该命名空间需符合项目可用资源配额。 新命名空间在项目分配的资源配额范围内运行（例如 CPU、内存），且命名空间内的所有资源

> 必须位于关联的集群中。

1. 在 项目管理 视图中，点击要创建命名空间的 *项目名称*。

2. 在左侧导航栏中，点击 **Namespaces** > **Namespaces**。

3. 点击 创建命名空间。

4. 配置 基本信息。

参数	说明
集群	选择与项目关联的集群，用于承载该命名空间。
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。

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

   注意：

   - 命名空间的资源配额继承自项目在集群中分配的配额。某一资源类型的最大允许配额不得超过项目剩余可用配额。如果某资源的可用配额为 0，则阻止创建命名空间。请联系平台管理员调整配额。
   - **GPU** 配额配置要求：
     - 仅当集群中配置了 GPU 资源时，才能配置 GPU 配额（虚拟 GPU 或物理 GPU）。
     - 使用虚拟 GPU 时，也可以设置内存配额。

     GPU 单位定义：

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

    - 1 GiB = 4 个内存单位（1024 MiB = 4 × 256 MiB）。

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

配额参数说明

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

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

6. （可选）配置 容器限制范围；详情请参见限制范围。

7. （可选）配置 **Pod** 安全准入；具体详情请参见Pod 安全准入。

8. （可选）在 更多配置 区域，为当前命名空间添加标签和注解。

   提示：可以通过标签定义命名空间的属性，或通过注解补充命名空间的额外信息；两者均可用于筛选和排序命名空间。

9. 点击 创建。

通过 **CLI** 创建命名空间

YAML 文件示例

example-namespace.yaml

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

example-resourcequota.yaml

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

example-limitrange.yaml

```
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
```

通过命令行直接创建

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

导入 Namespace

目录

Overview

Namespace 生命周期管理能力：

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

Namespace 解除关联：

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

重要说明：

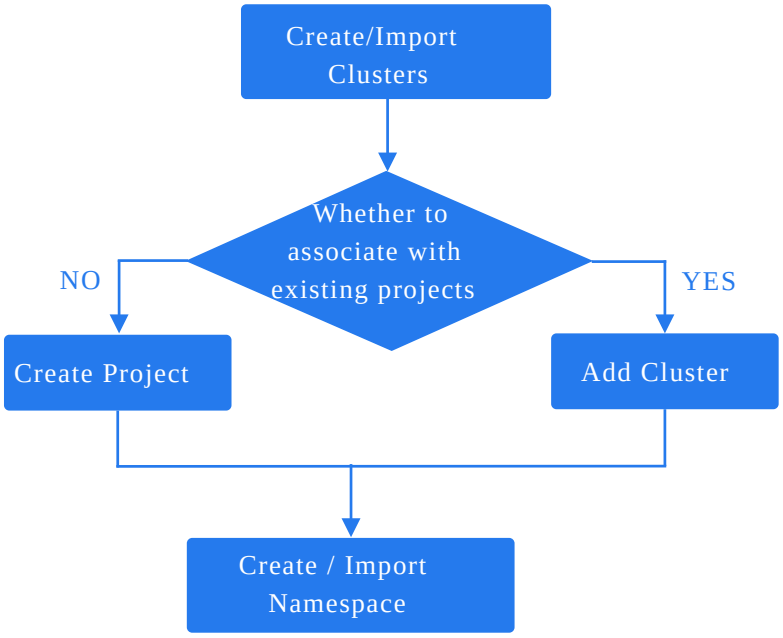
- 一个 Namespace 在任意时刻只能关联到一个 Project。

- 如果 Namespace 已经关联到某个 Project，则必须先解除与该 Project 的关联，才能导入或重新分配到另一个 Project。

Use Cases

Namespace 管理的常见使用场景包括：

- 当新 **Kubernetes** 集群 连接到平台时，可以使用 导入 **Namespace** 功能，将其已有的 **Kubernetes Namespace** 关联到某个 Project。只需选择目标 Project 和集群，即可发起导入操作。此操作赋予该 **project** 对这些 **namespace** 的治理能力，包括 资源配额、监控和策略执行。



- 已从某个 **project** 解除关联的 **namespace**，可以通过 导入 **Namespace** 功能无缝地重新关联到另一个 project，实现持续的集中治理。

- 当前未被任何 **project** 管理的 Namespace（例如通过集群级脚本创建的），必须使用 导入 **Namespace** 功能关联到目标 **project**，以启用平台级的治理，包括可视化和集中管理。

Prerequisites

- 该 Namespace 当前未被平台内任何已有 Project 管理。

- Namespace 只能导入到已关联其目标 Kubernetes 集群的 Project 中。如果不存在此类 Project，需先创建一个与该集群关联的 Project。

Procedure

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

资源配额

参考官方 Kubernetes 文档：Resource Quotas ↗

目录

理解资源请求与限制

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

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

资源限制：定义容器可使用的最大资源，防止资源耗尽，确保集群稳定。

配额

资源配额

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

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

支持的配额类型

字段	描述
资源请求	命名空间内所有 Pod 的资源请求总量： • CPU • 内存
资源限制	命名空间内所有 Pod 的资源限制总量： • CPU • 内存
Pod 数量	命名空间内允许的最大 Pod 数量。

注意：

- 命名空间配额来源于项目分配的集群资源。如果任何资源的可用配额为 0，则命名空间创建失败。请联系管理员。
- `Unlimited` 表示命名空间可使用该项目剩余的集群资源。

YAML 文件示例

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

使用 CLI 创建资源配额

通过 YAML 文件创建

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

通过命令行直接创建

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

存储配额

配额类型：

- 全部：命名空间内 PVC 的存储容量总量。
- 存储类：特定存储类的 PVC 存储容量总量。

注意：确保存储类已预先分配给包含该命名空间的项目。

硬件加速器资源配额

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

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

其他配额

自定义配额名称的格式必须符合以下规范：

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

Limit Range

目录

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 LimitRange 创建或更新后创建的容器或 Pod 设置默认请求值、限制值和最大值，同时持续监控容器的使用情况，确保命名空间内的资源不会超过定义的最大值。

> 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 中容器的资源请求总和）。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 是一个准入控制器：

- 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
```

通过命令行直接创建

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

Pod Security Admission

参考官方 Kubernetes 文档：Pod Security Admission ↗

Pod Security Admission（PSA）是一个 Kubernetes 的 admission controller，通过验证 Pod 规范是否符合预定义标准，在命名空间级别强制执行安全策略。

目录

Security Modes

PSA 定义了三种模式来控制如何处理策略违规：

Mode	行为	使用场景
Enforce	拒绝创建/修改不合规的 Pod。	需要严格安全执行的生产环境。
Audit	允许创建 Pod，但在审计日志中记录违规行为。	监控和分析安全事件而不阻止工作负载。

Mode	行为	使用场景
Warn	允许创建 Pod，但向客户端返回违规警告。	测试环境或策略调整的过渡阶段。

关键说明：

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

Security Standards

PSA 定义了三种安全标准来限制 Pod 权限：

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

Configuration

Namespace Labels

通过给命名空间应用标签来定义 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
# 第一步：更新 Pod Admission 标签
kubectl label namespace <namespace-name> \
  pod-security.kubernetes.io/enforce=baseline \
  pod-security.kubernetes.io/audit=restricted \
  --overwrite

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

Exemptions

对特定用户、命名空间或 runtime classes 免除 PSA 检查。

示例配置：

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

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

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 强制执行 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，请移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，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中 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 UID 范围中的第一个值。

1. 在该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. 在该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 创建将失败，并显示错误信息，提示指定的 UID/GID 超出分配范围。

Overcommit Ratio

目录

理解命名空间资源超售比

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

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

关键概念

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

核心能力

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

示例

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

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

CRD 定义

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

使用 CLI 创建超售比

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

使用 Web 控制台创建/更新超售比

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

注意事项

如果集群使用节点虚拟化（如虚拟节点），请在为虚拟机配置之前先在集群/命名空间层面禁用超售。

操作步骤

1. 进入项目管理，导航至命名空间 > 命名空间列表。

2. 点击目标*命名空间名称*。

3. 点击操作 > 更新超售比。

4. 选择合适的超售比配置方式，为命名空间设置 CPU 或内存超售比。

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

5. 点击更新。

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

管理命名空间成员

目录

导入成员

平台支持将成员批量导入命名空间，并分配命名空间管理员或开发人员等角色以授予相应权限。

约束与限制

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

前提条件

要将用户导入为命名空间成员，必须先将其添加到该命名空间所属的项目中。

操作步骤

1. 进入项目管理，点击包含要导入成员的*项目名称*。
2. 导航至命名空间 > 命名空间。
3. 点击要导入成员的*命名空间名称*。
4. 在命名空间成员标签页，点击导入成员。
5. 按照以下操作步骤，将列表中的全部或部分用户导入命名空间。

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

- 将列表中所有用户作为命名空间成员导入，并批量分配角色。
  1. 点击对话框底部设置角色项右侧的下拉框，选择要分配的角色名称。
  2. 点击全部导入。
- 从列表中导入一个或多个用户作为命名空间成员。
  1. 点击用户名/显示名前的复选框，选择一个或多个用户。
  2. 点击对话框底部设置角色项右侧的下拉框，选择要分配给所选用户的角色名称。
  3. 点击导入。

添加成员

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

你可以添加符合输入要求的有效 OIDC 账号作为命名空间角色，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。请确保所添加的账号有效，否则这些账号将无法成功登录平台。

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

前提条件

平台已添加**OICD**类型的 IDP。

操作步骤

1. 进入项目管理，点击包含要添加成员的*项目名称*。
2. 导航至命名空间 > 命名空间。
3. 点击要添加成员的*命名空间名称*。
4. 在命名空间成员标签页，点击添加成员。
5. 在用户名输入框中，输入平台支持的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本平台。
6. 在角色下拉框中，选择要为该用户配置的角色名称。
7. 点击添加。 添加成功后，可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）中也可查看该用户。用户在成功登录或同步到本平台之前，来源显示为 `-` ，且可删除；当账号成功登录或同步到平台后，平台会记录账号的来源信息并在用户列表中显示。

移除成员

移除指定的命名空间成员，并删除其关联的角色以撤销其命名空间权限。

操作步骤

1. 进入项目管理，点击包含要移除成员的*项目名称*。

2. 导航至命名空间 > 命名空间。

3. 点击要移除成员的*命名空间名称*。

4. 在命名空间成员标签页，点击要移除成员记录右侧的⋮> 移除。

5. 点击移除。

更新命名空间

目录

更新配额

> Resource Quota

通过 **Web** 控制台更新资源配额

1. 进入 **Project Management**，在左侧边栏导航到 **Namespaces** > **Namespace** 列表。
2. 点击目标 ***namespace name***。
3. 点击 **Actions** > **Update Quota**。

4. 调整资源配额（CPU、Memory、Pods 等），然后点击 **Update**。

通过 CLI 更新资源配额

> Resource Quota YAML file example

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

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

更新容器 LimitRanges

> Limit Range

通过 Web 控制台更新 LimitRange

1. 进入 **Project Management** 视图，在左侧边栏导航到 **Namespaces** > **Namespace** 列表。

2. 点击目标 ***namespace name***。

3. 点击 **Actions** > **Update Container LimitRange**。

4. 调整容器限制范围（`defaultRequest`、`default`、`max`），然后点击 **Update**。

通过 CLI 更新 LimitRange

> Limit Range YAML file example

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

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

更新 Pod Security Admission

> Pod Security Admission

通过 Web 控制台更新 Pod Security Admission

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

通过 CLI 更新 Pod Security Admission

> Update Pod Security Admission CLI command

删除/移除命名空间

您可以选择永久删除命名空间，或将其从当前项目中移除。

目录

删除命名空间

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

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

移除命名空间

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

NOTE

- 此功能仅限于 灵雀云容器平台。

- Kubernetes 本身不支持将命名空间“移除”出项目。

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

Menu

创建应用

Creating applications from Image

Prerequisites

Procedure 1 - Workloads

Procedure 2 - Services

Procedure 3 - Ingress

应用管理操作

参考信息

Creating applications from Chart

注意事项

前提条件

操作步骤

状态分析参考

通过 YAML 创建应用

注意事项

前提条件

操作步骤

通过代码创建应用

前提条件

操作步骤

Creating applications from Operator Backed

了解 Operator Backed 应用

通过 Web 控制台创建 Operator Backed 应用

故障排查

通过 CLI 工具创建应用

前提条件

操作步骤

示例

参考

Creating applications from Image

目录

Prerequisites

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

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

- 如果是第三方平台的镜像仓库，请确保当前集群可以直接拉取该镜像。

Procedure 1 - Workloads

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

2. 点击 **Create**。

3. 选择 **Create from Image** 作为创建方式。

4. 选择或输入镜像，点击 **Confirm**。

> **INFO**
>
> 注意：使用集成到 Web 控制台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 images (docker-registry-projectname)，其中包含该 Web 控制台中的项目名 projectname 以及镜像仓库中的项目名 containers。

6. 按照以下说明配置相关参数。

Workload 1 - 配置基本信息

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

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

参数	说明
More > **Update Strategy**	配置零停机部署的 `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。 • 支持绝对值（如 `2` ）或百分比（如 `20%` ）。 • 百分比计算方式：`ceil(current_replicas × percentage)` 。 • 示例：从 10 个副本计算，4.1 → `5` 。 **Max unavailable** (`maxUnavailable`)： • 更新期间允许不可用的最大 Pod 数量。 • 百分比值不可超过 `100%` 。 • 百分比计算方式：`floor(current_replicas × percentage)` 。 • 示例：从 10 个副本计算，4.9 → `4` 。 注意： 1. 默认值：未显式设置时， `maxSurge=1` ， `maxUnavailable=1` 。 2. 非运行状态的 **Pod**（如 `Pending` / `CrashLoopBackOff` ）视为不可用。 3. 同时约束： • `maxSurge` 和 `maxUnavailable` 不能同时为 `0` 或 `0%` 。 • 若两者百分比均计算为 `0` ，Kubernetes 会强制设置 `maxUnavailable=1` 以保证更新进度。 示例： 对于 10 个副本的 Deployment： • `maxSurge=2` → 更新期间总 Pod 数量为 `10 + 2 = 12` 。 • `maxUnavailable=3` → 最小可用 Pod 数量为 `10 - 3 = 7` 。 • 保证了可用性的同时允许受控滚动更新。

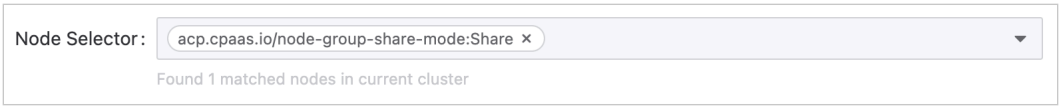
Workload 2 - 配置 Pod

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

1. 在 **Pod** 部分，配置容器运行时参数及生命周期管理：

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

2. 节点亲和规则

参数	说明
More > Node Selector	限制 Pod 调度到具有特定标签的节点（例如 `kubernetes.io/os: linux`）。 
More > Affinity	基于已有 Pod 定义细粒度调度规则。 **Pod** 亲和类型： • **Pod** 亲和：将新 Pod 调度到运行特定 Pod 的节点（同拓扑域）。 • **Pod** 反亲和：避免新 Pod 与特定 Pod 共置。 执行模式： • **RequiredDuringSchedulingIgnoredDuringExecution**：仅当规则满足时调度 Pod。

参数	说明
	• **PreferredDuringSchedulingIgnoredDuringExecution**：优先满足规则的节点，但允许例外。 配置字段： • `topologyKey`：定义拓扑域的节点标签（默认：`kubernetes.io/hostname`）。 • `labelSelector`：通过标签查询过滤目标 Pod。

3. 网络配置

- Kube-OVN

参数	说明
Bandwidth Limits	对 Pod 网络流量实施 QoS： • 出站速率限制：最大出站流量速率（如 `10Mbps`）。 • 入站速率限制：最大入站流量速率。
Subnet	从预定义子网池分配 IP。若未指定，使用命名空间默认子网。
Static IP Address	绑定持久 IP 地址到 Pod： • 多个 Deployment 中的 Pod 可以声明相同 IP，但同一时刻仅允许一个 Pod 使用。 • 关键：静态 IP 数量必须≥Pod 副本数。

- Calico

参数	说明
Static IP Address	分配固定 IP，严格唯一： • 每个 IP 只能绑定到集群中的一个 **Pod**。

参数	说明
	• 关键：静态 IP 数量必须≥Pod 副本数。

Workload 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**：容器运行时允许的最大 CPU/内存。单位定义见资源单位。 命名空间超售比： • 无超售比： 若存在命名空间资源配额，容器请求/限制继承命名空间默认值（可修改）。 无命名空间配额时，无默认值，自定义 Requests。 • 有超售比： 请求自动计算为 `Limits / 超售比`（不可修改）。 约束： • Request ≤ Limit ≤ 命名空间配额最大值。 • 超售比变更需重建 Pod 生效。 • 超售比启用时禁用手动请求配置。 • 无命名空间配额时无容器资源限制。
Extended Resources	配置集群可用的扩展资源（如 vGPU、pGPU）。
Volume Mount	持久存储配置。详见存储卷挂载说明。 操作： • 已有 Pod 卷时点击 **Add**

参数	说明
	• 无 Pod 卷时点击 **Add & Mount** 参数： • `mountPath`：容器文件系统路径（如 `/data`） • `subPath`：卷内相对文件/目录路径。 对于 `ConfigMap` / `Secret`：选择具体 key • `readOnly`：以只读方式挂载（默认读写） 详见Kubernetes 卷↗。
Port	暴露容器端口。 示例：暴露 TCP 端口 `6379`，名称为 `redis`。 字段： • `protocol`：TCP/UDP • `Port`：暴露端口（如 `6379`） • `name`：符合 DNS 规范的标识符（如 `redis`）
Startup Commands & Arguments	覆盖默认 ENTRYPOINT/CMD： 示例 **1**：执行 `top -b` - **Command**: `["top", "-b"]` - 或 Command: `["top"]`，Args: `["-b"]` 示例 **2**：输出 `$MESSAGE`： `/bin/sh -c "while true; do echo $(MESSAGE); sleep 10; done"` 详见定义命令↗。
More > Environment Variables	• 静态值：直接键值对 • 动态值：引用 ConfigMap/Secret 键，Pod 字段（`fieldRef`），资源指标（`resourceFieldRef`） 注意：环境变量会覆盖镜像或配置文件中的设置。

参数	说明
More > **Referenced ConfigMap**	将整个 ConfigMap/Secret 注入为环境变量。支持的 Secret 类型：`Opaque`、`kubernetes.io/basic-auth`。
More > **Health Checks**	• **Liveness Probe**：检测容器健康状况（失败时重启） • **Readiness Probe**：检测服务可用性（失败时从 Endpoints 移除） 详见健康检查参数。
More > **Log File**	配置日志路径： - 默认收集 `stdout` - 文件模式示例：`/var/log/*.log` 要求： • 存储驱动 `overlay2`：默认支持 • `devicemapper`：需手动挂载 EmptyDir 到日志目录 • Windows 节点：确保挂载父目录（如 `c:/a` 用于 `c:/a/b/c/*.log`）
More > **Exclude Log File**	排除特定日志收集（如 `/var/log/aaa.log`）。
More > **Execute before Stopping**	容器终止前执行命令。 示例：`echo "stop"` 注意：命令执行时间必须小于 Pod 的 `terminationGracePeriodSeconds`。

2. 点击右上角 **Add Container** 或 **Add Init Container**。

   参见Init Containers↗。 Init Container：

   1. 在应用容器启动前运行（顺序执行）。
   2. 完成后释放资源。
   3. 允许删除条件：

- Pod 有多个应用容器且至少一个 Init Container。
- 单应用容器 Pod 不允许删除 Init Container。

3. 点击 **Create**。

Procedure 2 - Services

参数	说明
Service	Kubernetes **Service**，将集群内运行的应用暴露为单一对外访问端点，即使工作负载分布在多个后端。具体参数说明请参见创建 Service。 注意：应用下创建的内部路由默认名称前缀为计算组件名称。如果计算组件类型（部署模式）为 StatefulSet，建议不要更改内部路由（工作负载名称）默认名称，否则可能导致工作负载访问异常。

Procedure 3 - Ingress

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

7. 点击 **Create**。

应用管理操作

修改应用配置时，可使用以下任一方式：

1. 点击应用列表右侧的竖点（⋮）。
2. 在应用详情页面右上角选择 **Actions**。

操作	说明

Update	• 更新：仅修改目标工作负载，使用其定义的更新策略（以 Deployment 策略为例）。保留现有副本数和滚动配置。
	• 强制更新：触发全应用滚动，使用各组件的更新策略。
	1. 适用场景：
	  • 批量配置变更需立即全局生效（如作为环境变量引用的 ConfigMap/Secret 更新）。
	  • 关键安全更新需协调组件重启。
	2. 警告注意：
	  • 批量重启可能导致短暂服务降级。
	  • 生产环境使用前需验证业务连续性。
	• 网络影响：
	  • Ingress 规则删除：若
	  1) LoadBalancer Service 使用默认端口，
	  2) 存在引用应用组件的存活路由规则，
	  外部访问仍可通过 `LB_IP:NodePort` 访问。
	  完全终止外部访问需删除 Service。
	  • Service 删除：应用组件网络连接不可恢复，关联 Ingress 规则失效，尽管 API 对象仍存在。

Delete	• 级联删除： 1. 删除所有子资源，包括 Deployment、Service 和 Ingress 规则。 2. Persistent Volume Claim（PVC）遵循 StorageClass 中定义的保留策略。 • 删除前检查： 1. 确认无活跃流量通过关联 Service。

操作	说明
	2. 确认有状态组件数据已备份。 3. 使用 `kubectl describe ownerReferences` 检查依赖资源关系。

参考信息

存储卷挂载说明

类型	用途
Persistent Volume Claim	绑定已有的 PVC 以请求持久存储。 注意：仅可选择已绑定（关联 PV）的 PVC。未绑定 PVC 会导致 Pod 创建失败。
ConfigMap	挂载完整或部分 ConfigMap 数据为文件： • 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（如 `my.cnf`）
Secret	挂载完整或部分 Secret 数据为文件： • 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（如 `tls.crt`）
Ephemeral Volumes	集群动态提供的临时卷，具备以下特性： • 动态配置 • 生命周期与 Pod 绑定 • 支持声明式配置 使用场景：临时数据存储。详见临时卷

类型	用途
Empty Directory	Pod 内容器间共享的临时存储： - Pod 启动时在节点创建 - Pod 删除时删除 使用场景：容器间文件共享、临时数据存储。详见EmptyDir
Host Path	挂载宿主机目录（必须以 `/` 开头，如 `/volumepath`）。

健康检查参数

通用参数

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

协议特定参数

参数	适用协议	说明
Protocol	HTTP/HTTPS	健康检查协议
Port	HTTP/HTTPS/TCP	目标容器端口。
Path	HTTP/HTTPS	端点路径（如 `/healthz`）。

参数	适用协议	说明
HTTP Headers	HTTP/HTTPS	自定义请求头（添加键值对）。
Command	EXEC	容器内可执行的检查命令（如 `sh -c "curl -I localhost:8080 | grep OK"`）。 注意：需转义特殊字符并测试命令有效性。

Creating applications from Chart

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

目录

注意事项

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

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

前提条件

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

操作步骤

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

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

状态分析参考

点击 *应用名称*，可显示 Chart 的详细状态分析信息。

类型	原因

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

Validated	表示 Chart 模板的用户权限、依赖关系及其他校验状态。 • **True**：表示所有校验均通过。

类型	原因

	• **False**：表示存在未通过的校验；可在消息栏查看具体失败原因。
	  • `DependenciesCheckFailed`：Chart 依赖检查失败。
	  • `PermissionCheckFailed`：当前用户缺少对某些资源的操作权限。
	  • `ConsistentNamespaceCheckFailed`：通过原生应用模板部署应用时，Chart 包含需要跨命名空间部署的资源。

Synced	表示 Chart 模板的部署状态。 • **True**：表示 Chart 模板已成功部署。 • **False**：表示 Chart 模板部署失败；原因栏显示 `ChartSyncFailed`，可在消息栏查看具体失败原因。

WARNING

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

通过 YAML 创建应用

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

目录

注意事项

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

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

前提条件

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

操作步骤

1. 进入 **Container Platform**，导航至 **Application** > **Applications**。

2. 点击 **Create**。

3. 选择 **Create from YAML**。

4. 完成配置后点击 **Create**。

5. 可在详情页查看对应的 **Deployment**。

```yaml
# 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
```

通过代码创建应用

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。该方法标准化并自动化了应用构建流程，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

前提条件

- 完成 Alauda Container Platform Builds 的安装

操作步骤

1. 进入 **Container Platform**，导航至 **Application** > **Applications**。
2. 点击 **Create**。
3. 选择 **Create from Code**。
4. 详细参数说明，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

了解 Operator Backed 应用

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 应用指通过预集成或用户自定义的 Operator 创建的应用实例，其运行流程由 Operator Lifecycle Manager (OLM) 管理，包括安装、升级、依赖关系解析和访问控制等关键环节。

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更新等复杂问题。例如：Operator 编码逻辑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），持续对比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：

   - 滚动更新与回滚：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 规范 Operator 打包（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 应用 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，详细描述创建 Operator 及其依赖资源的具体步骤。

4. OperatorGroup↗：定义 Operator 提供服务和资源调和的目标命名空间集合，同时限制 Operator 的 RBAC 权限范围。

5. Subscription↗：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 利用 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，导航至左侧栏的 **Applications** > **Applications**。

2. 点击 **Create**。

3. 选择 **Create from Catalog** 作为创建方式。

4. 选择一个 Operator Backed 实例并配置 **Custom Resource Parameters**。选择一个 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）。其他 CR 特定参数，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，完成时间取决于集群状态，可能需要几分钟。

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 准备就绪后重试创建：

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

通过 **CLI** 工具创建应用

`kubectl` 是与 Kubernetes 集群交互的主要命令行界面（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 操作均通过 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）。

该 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有所不同：

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

目录

前提条件

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

操作步骤

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

示例

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 命令

```
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}
```

参考

- 概念指南: kubectl Overview ↗
- 语法参考: kubectl Cheat Sheet ↗
- 命令手册: kubectl Commands ↗

Menu

应用的操作与维护

Application Rollout

安装 Alauda Container Platform Argo Rollouts

前提条件

安装 Alauda Container Platform Argo Rollouts

Application Blue Green Deployment

前提条件

操作步骤

Application Canary Deployment

前提条件

操作步骤

状态说明

状态说明

Applications

KEDA(Kubernetes Event-driven Autoscaling)

KEDA 概览

介绍

优势

KEDA 的工作原理

Installing KEDA

前提条件

通过命令行安装

通过 Web 控制台安装

验证

其他场景

卸载 KEDA Operator

实用指南

配置 HPA

配置 HPA

了解水平 Pod 自动扩缩器

前提条件

创建水平 Pod 自动扩缩器

计算规则

启动和停止原生应用

配置 VerticalPodAutoscaler (VPA)

配置 CronHPA

更新原生应用

更新原生应用

导入资源

移除/批量移除资源

导出应用

导出应用

导出 Helm Chart

导出 YAML 到本地

导出 YAML 到代码仓库（Alpha）

更新和删除 **Chart** 应用

更新和删除 **Chart** 应用

重要说明

前提条件

状态分析说明

应用版本管理

应用版本管理

创建版本快照

回滚到历史版本

删除原生应用

删除原生应用

处理资源耗尽错误

处理资源耗尽错误

Overview

配置驱逐策略

在节点配置中创建驱逐策略

驱逐信号

驱逐阈值

配置可调度资源

防止节点状态振荡

回收节点级资源

Pod 驱逐

服务质量与内存杀手（OOM Killer）

调度器与资源耗尽状态

示例场景

推荐实践

健康检查

健康检查

理解健康检查

YAML 文件示例

通过 Web 控制台配置健康检查参数

探针失败故障排查

Menu

Application Rollout

安装 Alauda Container Platform Argo Rollouts

前提条件

安装 Alauda Container Platform Argo Rollouts

Application Blue Green Deployment

前提条件

操作步骤

Application Canary Deployment

前提条件

操作步骤

安装 Alauda Container Platform Argo Rollouts

目录

前提条件

1. 下载与您的平台架构对应的 **Alauda Container Platform Argo Rollouts** 集群插件安装包。
2. 使用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中。

> **INFO**
>
> 上架软件包： 进入 管理员 > **Marketplace** > **Upload Packages** 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。更多详情请参考 CLI。

安装 Alauda Container Platform Argo Rollouts

操作步骤

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

Application Blue Green Deployment

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，将更新推送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。蓝绿发布（Blue-Green Deployment）是一种部署策略，通过确保应用新版本能够实现零停机部署，从而降低了这种风险。

蓝绿发布是一种部署策略，其中设置两个相同的环境，即“蓝色”环境和“绿色”环境。蓝色环境是生产环境，当前运行着应用的在线版本；绿色环境是非生产环境，用于部署应用的新版本。

当应用的新版本准备好部署时，会先部署到绿色环境。一旦新版本部署并测试完成，流量切换到绿色环境，使其成为新的生产环境。蓝色环境则变为非生产环境，用于部署未来的应用版本。

蓝绿发布的优势

- 零停机：蓝绿发布允许应用新版本实现零停机部署，因为流量可以无缝地从蓝色环境切换到绿色环境。
- 简单回滚：如果新版本出现问题，回滚到之前的版本非常简单，因为蓝色环境仍然可用。
- 降低风险：通过使用蓝绿发布，部署新版本的风险显著降低。因为新版本可以先在绿色环境中部署和测试，然后再将流量从蓝色环境切换过去。这允许进行充分测试，减少生产环境出现问题的可能性。
- 提高可靠性：使用蓝绿发布可以提升应用的可靠性。蓝色环境始终可用，绿色环境出现问题时可以快速识别并解决，而不会影响用户。
- 灵活性：蓝绿发布为部署过程提供灵活性。可以并行部署多个版本的应用，方便测试和实验。

使用 **Argo Rollouts** 进行蓝绿发布

Argo Rollouts 是一个 Kubernetes 控制器及一组 CRD，提供了蓝绿、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（可选）集成了 ingress 控制器和服务网格，利用它们的流量调度能力在更新过程中逐步切换流量到新版本。此外，Rollouts 可以查询和解析来自多种提供者的指标，以验证关键 KPI，并在更新过程中驱动自动升级或回滚。

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

1. 定义 Rollout 资源以管理不同的应用版本。
2. 配置 Kubernetes 服务以在蓝色（当前）和绿色（新）环境之间路由流量。
3. 将新版本部署到绿色环境。
4. 验证并测试新版本。
5. 通过切换流量将绿色环境提升为生产环境。

该方法最大限度地减少停机时间，实现受控且安全的部署。

> 关键概念：
>
> - **Rollout**：Kubernetes 中的一种自定义资源定义（CRD），替代标准的 Deployment 资源，实现蓝绿、金丝雀等高级部署控制。

目录

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. 安装 Argo Rollouts 的 kubectl 插件。
5. 一个用于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 创建部署

首先定义应用的“蓝色”版本，即用户当前访问的版本。创建一个 Kubernetes Deployment，指定适当的副本数、容器镜像版本（例如 `hello:1.23.1`）以及合适的标签，如 `app=web`。

使用以下 YAML：

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

YAML 字段说明：

- `apiVersion`：用于创建资源的 Kubernetes API 版本。
- `kind`：指定资源类型为 Deployment。
- `metadata.name`：部署名称。
- `spec.replicas`：期望的 Pod 副本数。
- `spec.selector.matchLabels`：定义 Deployment 管理哪些 Pod。
- `template.metadata.labels`：Pod 上的标签，供 Service 选择使用。
- `spec.containers`：Pod 中运行的容器列表。
- `containers.name`：容器名称。
- `containers.image`：运行的 Docker 镜像。
- `containers.ports.containerPort`：容器暴露的端口。

使用 `kubectl` 应用配置：

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

这将搭建生产环境。

另外，也可以使用 Helm Chart 创建部署和服务。

2 创建蓝色服务

创建一个 Kubernetes `Service`，用于暴露蓝色部署。该服务根据匹配标签将流量转发到蓝色 Pod。初始时，服务选择器指向带有 `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`：资源类型为 Service。
- `metadata.name`：Service 名称。
- `spec.selector`：根据标签选择 Pod 以路由流量。
- `ports.protocol`：使用的协议（TCP）。
- `ports.port`：Service 暴露的端口。
- `ports.targetPort`：容器接收流量的端口。

使用以下命令应用：

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

这允许外部访问蓝色部署。

3 验证蓝色部署

通过列出 Pod 确认蓝色部署运行正常：

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

检查所有预期副本（2个）均处于 `Running` 状态，确保应用已准备好提供服务。

4 验证流量路由到蓝色

确认 `web` 服务正确将流量转发到蓝色部署。执行：

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

输出应列出蓝色 Pod 的 IP 地址，这些即为接收流量的端点。

5 创建 **Rollout**

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

```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。现有 ReplicaSet 中被此选择器选中的 Pod 将受此 Rollout 影响。必须与 Pod 模板标签匹配。
- `workloadRef`：指定工作负载引用及缩容策略。
  - `scaleDown`：指定迁移到 Rollout 后是否缩减工作负载（Deployment）。可选值：
    - "never"：不缩减 Deployment。
    - "onsuccess"：Rollout 健康后缩减 Deployment。
    - "progressively"：Rollout 扩容时逐步缩减 Deployment，若 Rollout 失败则恢复 Deployment。
- `strategy`：部署策略，支持 `BlueGreen` 和 `Canary`。
  - `blueGreen`：蓝绿部署策略定义。
    - `activeService`：指定在升级时更新新模板哈希的服务。此字段为蓝绿策略必填。

- `autoPromotionEnabled`：禁用自动升级新版本，部署将在升级前暂停。默认行为是 ReplicaSet 完全就绪后自动升级。可用命令 `kubectl argo rollouts promote ROLLOUT` 恢复升级。

使用以下命令应用：

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

这将为部署设置蓝绿策略的 Rollout。

6 验证 Rollout

创建 `Rollout` 后，Argo Rollouts 会创建一个与 Deployment 模板相同的新 ReplicaSet。当新 ReplicaSet 的 Pod 健康时，Deployment 会缩容至 0。

使用以下命令确认 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     ✔ Healthy  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
```

服务 `web` 会将流量转发给 Rollouts 创建的 Pod。使用命令：

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

7 准备绿色部署

接下来，准备应用的新版本作为绿色部署。更新 Deployment `web`，使用新镜像版本（例如 `hello:1.23.2`）。

```
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：

- 与原部署相同，唯一不同的是：
  - `containers.image`：更新为新镜像版本。

使用以下命令应用：

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

这将搭建新版本应用以供测试。

Rollouts 会创建新的 ReplicaSet 管理绿色 Pod，流量仍然转发到蓝色 Pod。使用以下命令验证：

```
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 运行，包含蓝色和绿色版本。活动服务指向蓝色版本，Rollout 处于暂停状态。

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

8 升级 Rollout 到绿色

当绿色版本准备就绪，执行升级操作，将流量切换到绿色 Pod。使用命令：

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

验证升级是否完成：

```
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     ✔ Healthy      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  • ScaledDown   3h1m
      ├──□ rollout-bluegreen-595d4567cc-mc769  Pod         ◌ Terminating  3h
ready:1/1
      └──□ rollout-bluegreen-595d4567cc-zdc5x  Pod         ◌ Terminating  3h
ready:1/1
```

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

Application Canary Deployment

灰度发布是一种渐进式发布策略，通过逐步将新版本应用引入到少部分用户或流量中。此增量式的发布方式使团队能够监控系统行为、收集指标并确保稳定性，然后再进行全面部署。该方法显著降低了风险，尤其是在生产环境中。

Argo Rollouts 是一个 Kubernetes 原生的渐进式交付控制器，支持高级部署策略。它扩展了 Kubernetes 的能力，提供了灰度发布、蓝绿部署、分析运行、实验和自动回滚等功能。它可与可观测性栈集成，实现基于指标的健康检查，并通过 CLI 和监控面板对应用交付进行控制。

> 关键概念：
>
> - **Rollout**：Kubernetes 中的自定义资源定义（CRD），替代标准的 Deployment 资源，实现蓝绿、灰度等高级部署控制。
> - **Canary Steps**：一系列逐步调整流量的操作，例如先将 25% 流量导向新版本，再调整到 50%。
> - **Pause Steps**：引入等待间隔，用于手动或自动验证后再继续下一个灰度步骤。

灰度发布的优势

- 风险缓解：通过先将变更部署到少量服务器，可以发现并解决问题，减少对用户的影响。
- 渐进式发布：允许逐步暴露新功能，有助于有效监控性能和用户反馈。
- 实时反馈：灰度发布能即时反映新版本在真实环境中的性能和稳定性。
- 灵活性：可根据性能指标调整发布流程，实现动态发布，可暂停或回滚。
- 成本效益：与蓝绿部署不同，灰度发布无需独立环境，更节省资源。

使用 Argo Rollouts 进行灰度发布

Argo Rollouts 支持灰度发布策略来滚动更新 Deployment，并通过 Gateway API 插件控制流量。在 ACP 中，可以使用 ALB 作为 Gateway API Provider 来实现 Argo Rollouts 的流量控制。

目录

前提条件

1. 集群中已安装 Argo Rollouts 及 Gateway API 插件。
2. 安装 Argo Rollouts 的 kubectl 插件（安装地址见 here↗）。
3. 已有一个项目用于创建命名空间。
4. 集群中已部署 ALB 并分配给该项目。
5. 集群中已有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“稳定”版本，即当前用户访问的版本。创建一个 Kubernetes Deployment，设置合适的副本数、容器镜像版本（例如 `hello:1.23.1`）及标签，如 `app=web`。

使用以下 YAML：

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

YAML 字段说明：

- `apiVersion`：用于创建资源的 Kubernetes API 版本。
- `kind`：资源类型，这里是 Deployment。
- `metadata.name`：Deployment 的名称。
- `spec.replicas`：期望的 Pod 副本数。
- `spec.selector.matchLabels`：Deployment 用于选择管理的 Pod 标签。
- `template.metadata.labels`：Pod 的标签，Service 用于选择 Pod。
- `spec.containers`：Pod 中运行的容器列表。

- `containers.name`：容器名称。
- `containers.image`：容器镜像。
- `containers.ports.containerPort`：容器暴露的端口。

使用 `kubectl` 应用配置：

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

这将搭建生产环境。

也可以使用 Helm Chart 创建 Deployment 和 Service。

2 创建稳定服务

创建一个 Kubernetes `Service`，用于暴露稳定版本的 Deployment。该服务根据标签选择 Pod，初始选择器为 `app=web`。

```
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`：资源类型，这里是 Service。
- `metadata.name`：Service 名称。
- `spec.selector`：根据标签选择 Pod。
- `ports.protocol`：使用的协议（TCP）。

- `ports.port`：Service 暴露的端口。
- `ports.targetPort`：容器接收流量的端口。

使用命令应用：

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

该服务允许外部访问稳定版本。

3 创建灰度服务

创建一个 Kubernetes `Service`，用于暴露灰度版本的 Deployment。该服务根据标签选择 Pod，初始选择器为 `app=web`。

```
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`：资源类型，这里是 Service。
- `metadata.name`：Service 名称。
- `spec.selector`：根据标签选择 Pod。
- `ports.protocol`：使用的协议（TCP）。
- `ports.port`：Service 暴露的端口。
- `ports.targetPort`：容器接收流量的端口。

使用命令应用：

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

该服务允许外部访问灰度版本。

4 创建 Gateway

使用 `example.com` 作为访问域名，创建 Gateway 以该域名暴露服务：

```
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
```

使用命令：

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

Gateway 会分配一个外部 IP 地址，可通过 Gateway 资源的 `status.addresses` 中类型为 `IPAddress` 的字段获取该 IP。

```
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，将域名解析到 Gateway 的 IP 地址。使用命令验证 DNS 解析：

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

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

应返回 Gateway 的地址。

6 创建 HTTPRoute

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

使用命令：

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

7 访问稳定服务

集群外部通过域名访问服务：

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

或者在浏览器中访问 `http://example.com`。

8 创建 Rollout

接下来，使用 Argo Rollouts 创建采用 `Canary` 策略的 `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 标签选择器。现有 ReplicaSet 中被选择的 Pod 会受此 Rollout 影响，必须与 Pod 模板标签匹配。
- `workloadRef`：指定工作负载引用及缩容策略。
- `scaleDown`：指定迁移到 Rollout 后是否缩减工作负载（Deployment）。可选值：
  - "never"：不缩减 Deployment。
  - "onsuccess"：Rollout 健康后缩减 Deployment。
  - "progressively"：Rollout 扩容时逐步缩减 Deployment，失败时恢复 Deployment。
- `strategy`：部署策略，支持 `BlueGreen` 和 `Canary`。
- `canary`：灰度策略定义。
  - `canaryService`：控制器更新以选择灰度 Pod 的服务，流量路由必需。
  - `stableService`：控制器更新以选择稳定 Pod 的服务，流量路由必需。
  - `steps`：定义灰度更新的步骤序列，首次部署时跳过。
    - `setWeight`：设置灰度 ReplicaSet 的流量比例。
    - `pause`：暂停发布，支持单位 s、m、h，`{}` 表示无限期暂停。
    - `plugin`：执行配置的插件，此处配置了 `gatewayAPI` 插件。

使用命令应用：

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

这将为部署设置灰度发布策略。初始设置权重为 50，等待推广。50% 流量转发至灰度服务。推广后权重设为 100，100% 流量转发至灰度服务，最终灰度服务成为稳定服务。

9 验证 Rollout

创建 `Rollout` 后，Argo Rollouts 会创建一个与 Deployment 模板相同的新 ReplicaSet。当新 ReplicaSet 的 Pod 健康时，Deployment 会缩容至 0。

使用以下命令确认 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  32s   stable
       ├──□ rollout-canary-5c9d79697b-fh78d  Pod         ✔ Running  32s   ready:1/1
       └──□ rollout-canary-5c9d79697b-rrbtj  Pod         ✔ Running  32s   ready:1/1
```

10 准备灰度部署

接下来，准备新版本应用作为绿色部署。更新 Deployment `web` 的镜像版本（例如 `hello:1.23.2`）。使用命令：

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

这将为测试设置新版本应用。

Rollout 会创建新的 ReplicaSet 管理灰度 Pod，50% 流量转发至灰度 Pod。使用以下命令验证：

```
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   95s
├──# revision:2
│  └──⧉ rollout-canary-5898765588          ReplicaSet  ✔ Healthy  46s  canary
│     └──□ rollout-canary-5898765588-ls5jk  Pod         ✔ Running  45s
ready:1/1
└──# revision:1
   └──⧉ rollout-canary-5c9d79697b          ReplicaSet  ✔ Healthy  95s  stable
      ├──□ rollout-canary-5c9d79697b-fk269  Pod         ✔ Running  94s  ready:1/1
      └──□ rollout-canary-5c9d79697b-wkmcn  Pod         ✔ Running  94s  ready:1/1
```

当前运行 3 个 Pod，包含稳定和灰度版本。权重为 50，50% 流量转发至灰度服务。发布流程暂停，等待推广。

如果使用 Helm Chart 部署应用，可用 Helm 工具升级到灰度版本。

访问 `http://example.com`，50% 流量将转发至灰度服务，响应应有所不同。

11 推广 Rollout

当灰度版本测试通过后，可推广 Rollout，将所有流量切换至灰度 Pod。使用命令：

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

验证 Rollout 是否完成：

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

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

若稳定版本镜像更新为 `hello:1.23.2`，且 revision 1 的 ReplicaSet 缩容为 0，表示发布完成。

访问 `http://example.com`，100% 流量将转发至灰度服务。

12 中止 **Rollout**（可选）

若在发布过程中发现灰度版本存在问题，可中止发布，将所有流量切回稳定服务。使用命令：

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

验证结果：

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

NAME                                       KIND        STATUS        AGE  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`，100% 流量将转发至稳定服务。

状态说明

目录

Applications

原生应用的状态及其对应含义如下。状态后面的数字表示计算组件的数量。

状态	含义
Running	所有计算组件均处于正常运行状态。
Partially Running	部分计算组件正在运行，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待处理状态。
No Computing Components	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。

Deployment

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

Menu

KEDA(Kubernetes Event-driven Autoscaling)

KEDA 概览

Installing KEDA

实用指南

Integrating ACP Monitoring with Prometheus Plugin

在 KEDA 中暂停自动扩缩容

KEDA 概览

目录

介绍

KEDA 是一个基于 Kubernetes 的事件驱动自动扩缩器。主页↗。通过 KEDA，您可以根据需要处理的事件数量驱动 Kubernetes 中任何容器的扩缩。

KEDA 是一个单一用途且轻量级的组件，可以添加到任何 Kubernetes 集群中。KEDA 与标准的 Kubernetes 组件（如 Horizontal Pod Autoscaler↗）协同工作，能够扩展功能而不会覆盖或重复。使用 KEDA，您可以明确映射希望使用事件驱动扩缩的应用程序，而其他应用程序继续正常运行。这使得 KEDA 成为一个灵活且安全的选项，可以与任意数量的其他 Kubernetes 应用或框架并行运行。

更多详情请参阅官方文档：Keda Documentation↗

优势

KEDA 的核心优势：

- 自动扩缩简单化： 为 Kubernetes 集群中的每个工作负载带来丰富的扩缩能力。
- 事件驱动： 智能地扩缩您的事件驱动应用。
- 内置扩缩器： 拥有 70 多种内置扩缩器，支持各种云平台、数据库、消息系统、遥测系统、CI/CD 等。
- 多种工作负载类型： 支持多种工作负载类型，如 deployments、jobs 以及带有 **/scale** 子资源的自定义资源。
- 减少环境影响： 通过优化工作负载调度和零扩缩构建可持续平台。
- 可扩展： 支持自定义扩缩器或使用 Community 维护的扩缩器。
- 供应商无关： 支持多种云提供商和产品的触发器。
- 支持 **Azure Functions**： 在 Kubernetes 上运行并扩缩生产环境中的 Azure Functions。

KEDA 的工作原理

KEDA 监控外部事件源，并根据需求调整应用资源。其主要组件协同工作实现这一目标：

1. **KEDA Operator** 负责跟踪事件源，并根据需求上下调整应用实例数量。
2. **Metrics Server** 向 Kubernetes 的 HPA 提供外部指标，以便其做出扩缩决策。
3. **Scalers** 连接到消息队列或数据库等事件源，获取当前使用量或负载数据。
4. **Custom Resource Definitions (CRDs)** 定义应用如何基于队列长度或 API 请求率等触发器进行扩缩。

简单来说，KEDA 监听 Kubernetes 外部的事件，获取所需数据，并相应地扩缩应用。它高效且与 Kubernetes 集成良好，实现动态扩缩。

KEDA Custom Resource Definitions (CRDs)

KEDA 使用 **Custom Resource Definitions (CRDs)** 来管理扩缩行为：

- **ScaledObject**：将您的应用（如 Deployment 或 StatefulSet）与外部事件源关联，定义扩缩规则。
- **ScaledJob**：通过基于外部指标扩缩 Jobs 来处理批量任务。
- **TriggerAuthentication**：提供安全访问事件源的方式，支持环境变量或云特定凭据等方法。

这些 CRDs 让您在保持应用安全和响应需求的同时，灵活控制扩缩。

ScaledObject 示例：

以下示例针对整个 Pod 的 CPU 利用率进行扩缩。如果 Pod 中有多个容器，则为所有容器利用率之和。

```yaml
kind: ScaledObject
metadata:
  name: cpu-scaledobject
  namespace: <your-namespace>
spec:
  scaleTargetRef:
    name: <your-deployment>
  triggers:
  - type: cpu
    metricType: Utilization # Allowed types are 'Utilization' or 'AverageValue'
    metadata:
      value: "50"
```

Installing KEDA

目录

前提条件

KEDA 是一个帮助 Kubernetes 根据实际事件自动扩缩应用的工具。通过 KEDA，您可以根据工作负载的变化（例如队列中的消息数量或传入请求数）自动调整容器的规模。

1. 从 **Alauda Cloud** 下载 **KEDA** 安装包。

2. 使用上架软件包机制上传安装包。

> **INFO**
>
> 上架软件包： 进入 管理员 > **Marketplace** > 上传软件包 页面。 点击右侧的 帮助文档 获取如何将 operator 发布到集群的说明。更多详情请参考 CLI。

通过命令行安装

安装 KEDA Operator

如果不存在，则创建 KEDA operator 的命名空间：

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Namespace
metadata:
  name: "keda"
EOF
```

运行以下命令在目标集群中安装 KEDA Operator：

```
kubectl apply -f - <<EOF
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  annotations:
    cpaas.io/target-namespaces: ""
  labels:
    catalog: platform
  name: keda
  namespace: keda
spec:
  channel: stable
  installPlanApproval: Automatic
  name: keda
  source: custom
  sourceNamespace: cpaas-system
  startingCSV: keda.v2.17.2
EOF
```

配置参数：

参数	推荐配置
metadata.name	`keda`：Subscription 名称设置为 **keda**。
metadata.namespace	`keda`：Subscription 命名空间设置为 **keda**。
spec.channel	`stable`：默认 Channel 设置为 **stable**。
spec.installPlanApproval	`Automatic`：升级操作将自动执行。
spec.name	`keda`：operator 软件包名称，必须为 **keda**。
spec.source	`custom`：keda operator 的 catalog 来源，必须为 **custom**。
spec.sourceNamespace	`cpaas-system`：catalog 来源的命名空间，必须为 **cpaas-system**。
spec.startingCSV	`keda.v2.17.2`：keda operator 的起始 CSV 名称。

创建 KedaController 实例

在命名空间 keda 中创建名为 keda 的 KedaController 资源：

```
kubectl apply -f - <<EOF
apiVersion: keda.sh/v1alpha1
kind: KedaController
metadata:
  name: keda
  namespace: keda
spec:
  admissionWebhooks:
    logEncoder: console
    logLevel: info
  metricsServer:
    logLevel: "0"
  operator:
    logEncoder: console
    logLevel: info
  serviceAccount: null
  watchNamespace: ""
EOF
```

通过 Web 控制台安装

安装 KEDA Operator

1. 登录后，进入 管理员 页面。
2. 点击 **Marketplace** > **OperatorHub**。
3. 找到 **KEDA** operator，点击 安装，进入 安装 页面。

配置参数：

参数	推荐配置
Channel	`stable`：默认 Channel 设置为 **stable**。

参数	推荐配置
Version	请选择最新版本。
Installation Mode	`Cluster`：单个 Operator 在集群所有命名空间共享，用于实例创建和管理，降低资源消耗。
Installation Location	`Recommended`：如果不存在会自动创建。
Upgrade Strategy	请选择 `Auto`。 • 升级操作将自动执行。

4. 在 安装 页面，选择默认配置，点击 安装，完成 **KEDA** Operator 的安装。

创建 KedaController 实例

1. 点击 **Marketplace** > **OperatorHub**。

2. 找到已安装的 **KEDA** operator，进入 所有实例。

3. 点击 创建实例 按钮，在资源区域点击 **KedaController** 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 创建。

验证

实例创建成功后，等待约 20 分钟，然后通过以下命令检查 KEDA 组件是否已运行：

```
kubectl get pods -n keda
```

其他场景

集成 ACP 日志采集器

- 确保目标集群已安装 **ACP** 日志采集器插件。参考 ACP 日志采集器插件安装。
- 安装 **ACP** 日志采集器插件 时，开启 平台 日志开关。
- 使用以下命令为 **keda** 命名空间添加标签：

```
kubectl label namespace keda cpaas.io/product=Container-Platform --overwrite
```

卸载 KEDA Operator

删除 KedaController 实例

```
kubectl delete kedacontroller keda -n keda
```

通过 CLI 卸载 KEDA Operator

```
kubectl delete subscription keda -n keda
```

通过 Web 控制台卸载 KEDA Operator

进入 **Marketplace** > **OperatorHub**，选择已安装的 **KEDA** operator，点击 卸载。

Menu

实用指南

Integrating ACP Monitoring with Prometheus Plugin

Prerequisites

Procedure

Verification

在 KEDA 中暂停自动扩缩容

操作步骤

缩容至零

验证

Integrating ACP Monitoring with Prometheus Plugin

> 本指南介绍如何配置与 **ACP Monitoring with Prometheus Plugin** 的集成，以实现基于 Prometheus 指标的应用自动伸缩。

目录

Prerequisites

使用此功能前，请确保：

- 安装 ACP Monitoring with Prometheus Plugin
- 获取当前 Kubernetes 集群的 Prometheus 端点 URL 和 secretName：

```
PrometheusEndpoint=$(kubectl get feature monitoring -o
jsonpath='{.spec.accessInfo.database.address}')
```

- 获取当前 Kubernetes 集群的 Prometheus secret：

```
PrometheusSecret=$(kubectl get feature monitoring -o
jsonpath='{.spec.accessInfo.database.basicAuth.secretName}')
```

- 在 `<your-namespace>` 命名空间中创建名为 `<your-deployment>` 的 deployment。

Procedure

- 在 **keda** 命名空间中配置 Prometheus 认证 Secret。

将 **Secret** 从 **cpaas-system** 复制到 **keda** 命名空间的步骤

```
# 获取 Prometheus 认证信息
PrometheusUsername=$(kubectl get secret $PrometheusSecret -n cpaas-system -o
jsonpath='{.data.username}' | base64 -d)
PrometheusPassword=$(kubectl get secret $PrometheusSecret -n cpaas-system -o
jsonpath='{.data.password}' | base64 -d)

# 在 keda 命名空间创建 secret
kubectl create secret generic $PrometheusSecret \
  -n keda \
  --from-literal=username=$PrometheusUsername \
  --from-literal=password=$PrometheusPassword
```

- 使用 **ClusterTriggerAuthentication** 配置 KEDA 访问 Prometheus 的认证。

为配置 KEDA 访问 Prometheus 的认证凭据，定义一个 ClusterTriggerAuthentication 资源，引用包含用户名和密码的 Secret。示例如下：

```
kubectl apply -f - <<EOF
apiVersion: keda.sh/v1alpha1
kind: ClusterTriggerAuthentication
metadata:
  name: cluster-prometheus-auth
spec:
  secretTargetRef:
    - key: username
      name: $PrometheusSecret
      parameter: username
    - key: password
      name: $PrometheusSecret
      parameter: password
EOF
```

- 使用 **ScaledObject** 配置基于 Prometheus 指标的 Kubernetes Deployment 自动伸缩。

要基于 Prometheus 指标对 Kubernetes Deployment 进行伸缩，定义一个引用已配置 ClusterTriggerAuthentication 的 **ScaledObject** 资源。示例如下：

```
kubectl apply -f - <<EOF
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: prometheus-scaledobject
  namespace: <your-namespace>
spec:
  cooldownPeriod: 300          # 缩容前等待时间（秒）
  maxReplicaCount: 5           # 最大副本数
  minReplicaCount: 1           # 最小副本数（注意：HPA 可能强制最小为 1）
  pollingInterval: 30          # 轮询 Prometheus 指标的间隔（秒）
  scaleTargetRef:
    name: <your-deployment>    # 目标 Kubernetes Deployment 名称
  triggers:
    - authenticationRef:
        kind: ClusterTriggerAuthentication
        name: cluster-prometheus-auth  # 引用 ClusterTriggerAuthentication
      metadata:
        authModes: basic       # 认证方式（此处为 basic auth）
        query:
sum(container_memory_working_set_bytes{container!="POD",container!="",namespace="<your-
namespace>",pod=~"<your-deployment-name>.*"})
        queryParameters: timeout=10s  # 可选查询参数
        serverAddress: $PrometheusEndpoint
        threshold: "1024000"   # 伸缩阈值
        unsafeSsl: "true"      # 跳过 SSL 证书验证（生产环境不推荐）
      type: prometheus         # 触发器类型
EOF
```

Verification

要验证 ScaledObject 是否已对 deployment 进行伸缩，可以检查目标 deployment 的副本数：

```
kubectl get deployment <your-deployment> -n <your-namespace>
```

或者使用以下命令查看 pod 数量：

```
kubectl get pods -n <your-namespace> -l <your-deployment-label-key>=<your-deployment-label-value>
```

副本数应根据 ScaledObject 中指定的指标进行增减。 如果部署伸缩正常，您应看到 pod 数量已变更为 `maxReplicaCount` 的值。

Other KEDA scalers

KEDA **scalers** 既可以检测部署是否应激活或停用，也可以为特定事件源提供自定义指标。

KEDA 支持多种额外的 **scalers**。详情请参阅官方文档：KEDA Scalers ↗ 。

在 KEDA 中暂停自动扩缩容

KEDA 允许您临时暂停工作负载的自动扩缩容，这在以下情况下非常有用：

- 集群维护。
- 通过缩减非关键工作负载避免资源匮乏。

目录

操作步骤

立即暂停并保持当前从节点数

在您的 **ScaledObject** 定义中添加以下注解，以暂停扩缩容但不改变当前的从节点数：

```yaml
metadata:
  annotations:
    autoscaling.keda.sh/paused: "true"
```

缩放到指定从节点数后暂停

使用此注解将工作负载缩放到指定的从节点数后暂停：

```yaml
metadata:
  annotations:
    autoscaling.keda.sh/paused-replicas: "<number>"
```

两个注解同时设置时的行为

如果同时指定了 **paused** 和 **paused-replicas**：

- KEDA 会将工作负载缩放到 **paused-replicas** 中定义的值。
- 随后暂停自动扩缩容。

恢复自动扩缩容

要恢复自动扩缩容：

- 从 ScaledObject 中移除 paused 和 paused-replicas 两个注解。
- 如果只使用了 paused: "true"，则将其设置为 false：

```yaml
metadata:
  annotations:
    autoscaling.keda.sh/paused: "false"
```

缩容至零

示例 ScaledObject 配置：

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: example-scaledobject
  namespace: <your-namespace>
  annotations:
    autoscaling.keda.sh/paused-replicas: "0"  # 缩容至 0 个从节点并暂停
```

验证

要验证 ScaledObject 是否已缩容至零，可以检查目标 Deployment 的从节点数：

```
kubectl get deployment <your-deployment> -n <your-namespace>
```

或者检查目标 Deployment 中的 Pod 数量：

```
kubectl get pods -n <your-namespace> -l <your-deployment-label-key>=<your-deployment-label-value>
```

Pod 数量应为零，表示该 Deployment 已缩容至零。

配置 **HPA**

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

目录

了解水平 **Pod** 自动扩缩器

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

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

对于 replication controllers，此扩缩容直接对应 replication controller 的 Replica 数量。对于 deployment configurations，扩缩容直接对应 deployment configuration 的 Replica 数量。注意，自动扩缩容仅适用于处于 Complete 阶段的最新部署。

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

HPA 是如何工作的？

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

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

如果设置了利用率目标，controller 会将利用率值计算为各 Pod 容器请求资源的百分比。然后 controller 计算所有目标 Pod 的平均利用率，并生成一个比例，用于调整期望的 Replica 数量。

支持的指标

水平 Pod 自动扩缩器支持以下指标：

指标	描述
CPU 利用率	使用的 CPU 核数。可用于计算 Pod 请求 CPU 的百分比。
内存利用率	使用的内存量。可用于计算 Pod 请求内存的百分比。
网络入流量	进入 Pod 的网络流量，单位为 KiB/s。

指标	描述
网络出流量	从 Pod 发出的网络流量，单位为 KiB/s。
存储读流量	从存储读取的数据量，单位为 KiB/s。
存储写流量	写入存储的数据量，单位为 KiB/s。

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

前提条件

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

创建水平 **Pod** 自动扩缩器

使用 **CLI**

您可以通过命令行界面创建水平 Pod 自动扩缩器，方法是定义一个 YAML 文件并使用 `kubectl create` 命令。以下示例展示了针对 Deployment 对象的自动扩缩容。初始部署需要 3 个 Pod，HPA 对象将最小值提升到 5。如果 Pod 的 CPU 使用率达到 75%，Pod 数量将增加到 7：

1. 创建名为 `hpa.yaml` 的 YAML 文件，内容如下：

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

1. 使用 autoscaling/v2 API。
2. HPA 资源的名称。
3. 需要扩缩容的 Deployment 名称。
4. 最大扩缩容副本数。
5. 最小维持副本数。
6. 指定扩缩容对象的 API 版本。
7. 指定对象类型。对象必须是 Deployment、ReplicaSet 或 StatefulSet。
8. HPA 作用的目标资源。
9. 触发扩缩容的目标 CPU 利用率百分比。

2. 应用 YAML 文件创建 HPA：

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

示例输出：

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

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

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

示例输出：

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

4. 也可以检查 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** 控制台

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

参数	描述
Pod 数量	部署成功后，需要评估对应已知和常规业务量变化的最小 **Pod** 数量，以及在高业务压力下命名空间配额可支持的最大 **Pod** 数量。最大或最小 Pod 数量可在设置后更改，建议先通过性能测试推导更准确的值，并在使用过程中持续调整以满足业务需求。

参数	描述
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，设置 *CPU 利用率 = 60%*，一旦 CPU 利用率偏离 60%，平台将根据当前部署资源不足或过剩情况自动调整 Pod 数量。 注意：指标类型包括内置指标和自定义指标。自定义指标仅适用于原生应用中的部署，且需先添加自定义指标。
扩缩容步长（**Alpha**）	对于有特定扩缩容速率要求的业务，可以通过指定扩容步长或缩容步长逐步适应业务量变化。 缩容步长可自定义稳定窗口，默认 300 秒，表示执行缩容操作前需等待 300 秒。

6. 点击 更新。

使用自定义指标进行 **HPA**

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

要求

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

传统（核心指标）**HPA**

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

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

该 YAML 中，`scaleTargetRef` 指定了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**

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

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


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

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

示例中，`scaleTargetRef` 指定了工作负载。

触发条件定义

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

指标的主要结构如下：

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

该指标数据由 Prometheus 收集并更新。

自定义指标 **HPA** 兼容性

自定义指标 HPA YAML 实际兼容原核心指标（CPU），写法如下：

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

- `targetAverageValue` 是每个 Pod 获取的平均值
- `targetAverageUtilization` 是根据直接值计算的利用率

算法参考为：

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

autoscaling/v2beta2 的更新

autoscaling/v2beta2 支持内存利用率：

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

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

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

注意：

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

计算规则

当业务指标变化时，平台会根据以下规则自动计算匹配业务量的目标 Pod 数量并进行调整。如果业务指标持续波动，数值会被调整到设置的最小 **Pod** 数量或最大 **Pod** 数量。

- 单策略目标 Pod 数量：ceil[(sum(实际指标值)/指标阈值)]。即所有 Pod 的实际指标值之和除以指标阈值后向上取整。例如：当前有 3 个 Pod，CPU 利用率分别为 80%、80%、90%，设置的 CPU 利用率阈值为 60%。根据公式，Pod 数量自动调整为：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。如果调整最大 Pod 数量后指标仍持续偏高，可能需要采用其他扩缩容方式，如增加命名空间 Pod 配额或增加硬件资源。
  - 如果计算出的目标 Pod 数量（如上例的 *5*）小于根据扩容步长调整后的 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

1. 访问 **Container Platform**。

2. 在左侧导航栏中，点击 **Application** > **Applications**。

3. 点击原生应用名称。

4. 点击 **Start**。

停止原生应用

1. 访问 **Container Platform**。

2. 在左侧导航栏中，点击 **Application** > **Applications**。

3. 点击原生应用名称。

4. 点击 **Stop**。

5. 阅读提示信息，确认无误后，点击 **Stop**。

配置 VerticalPodAutoscaler (VPA)

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

目录

了解 VerticalPodAutoscalers

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

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

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

VPA 是如何工作的？

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

VPA 通过持续监控 Pod 的资源使用情况，并随着新数据的产生不断更新推荐。VPA 可运行于以下模式：

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

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

支持的功能

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

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

前提条件

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

安装 Vertical Pod Autoscaler 插件

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

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

创建 VerticalPodAutoscaler

使用 CLI

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

1. 创建名为 `vpa.yaml` 的 YAML 文件，内容如下：

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

1. 使用 autoscaling.k8s.io/v1 API。

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

2. 应用 YAML 文件创建 VPA：

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

示例输出：

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

3. 创建 VPA 后，可运行以下命令查看推荐：

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

示例输出（部分）：

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

使用 **Web** 控制台

1. 进入 **Container Platform**。

2. 在左侧导航栏点击 **Workloads** > **Deployments**。

3. 点击 ***Deployment*** *名称*。

4. 向下滚动至 弹性伸缩 区域，点击右侧的 更新。

5. 选择 垂直伸缩 并配置伸缩规则。

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

6. 点击 更新。

高级 **VPA** 配置

更新策略选项

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

容器策略选项

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

注意：

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

后续操作

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

配置 CronHPA

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

目录

了解 Cron Horizontal Pod Autoscalers

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，平台会开始监控该计划，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，每个规则指定一个时间（使用 crontab 格式）和目标的 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，无论当前资源利用率如何。

CronHPA 是如何工作的？

cron horizontal pod autoscaler（CronHPA）扩展了基于 Pod 自动扩缩容的概念，加入了基于时间的控制。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。您可以点击平台右上角 ? > 平台健康状态，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. 指定要扩缩容对象的 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（分钟 小时 日 月 星期）的定时计划。
8. 触发该计划时的目标 Replica 数量。

该示例配置该 Deployment：

- 每天午夜（0 点）缩容至 0 个 Replica
- 工作日（周一至周五）上午 8 点扩容至 3 个 Replica
- 工作日晚上 6 点缩容至 1 个 Replica

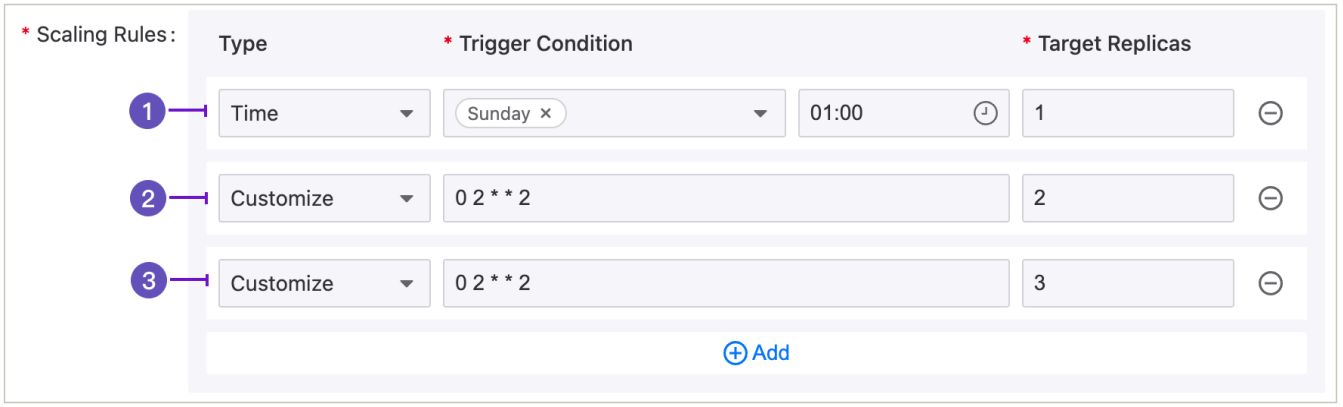
2. 应用 YAML 文件以创建 CronHPA：

```
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 部分，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，请谨慎考虑扩缩容策略。
- 调度使用 crontab 格式（分钟 小时 日 月 星期），规则与 Kubernetes CronJobs 相同。

- 时间基于集群的时区设置。
- 对于对可用性要求较高的工作负载，请确保定时伸缩不会在高峰期意外降低容量。

更新原生应用

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

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

导入到原生应用中的资源可享受以下功能：

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

目录

导入资源

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

1. 进入 **Container Platform**。

2. 在左侧导航栏点击 应用管理 > 原生应用。

3. 点击 *原生应用名称*。

4. 点击 操作 > 管理资源。

5. 在底部的 资源类型 中选择要导入的资源类型。

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

6. 在 资源 区域选择要导入的资源。

   注意：对于 **Job** 类型资源，仅支持通过 YAML 创建的任务导入。

7. 点击 导入资源。

移除/批量移除资源

> 从原生应用中移除/批量移除资源仅解除原生应用与资源的关联，不会删除资源。

如果原生应用下的资源之间存在关联，移除某个资源不会改变资源之间的关联关系。例如，即使 *Service S* 从 *原生应用 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. 在左侧导航栏点击 应用管理 > 原生应用。

3. 点击 *原生应用名称*。

4. 点击 操作 > 管理资源。

5. 点击资源右侧的 移除 按钮进行移除；或一次选择多个资源，点击表格顶部的 移除 按钮批量移除资源。

导出应用

为了规范开发、测试和生产环境之间的应用导出流程，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 功能实现跨集群快速部署应用。

目录

导出 **Helm Chart**

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。

   - 将 Helm Chart 导出至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。该模板仓库分配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显示。
README	描述文件，支持 Markdown 格式编辑，将显示在应用模板详情页。
NOTES	模板帮助文件，支持标准纯文本编辑；部署模板完成后，将显示在模板应用详情页。

- 将 Helm Chart 导出到本地，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

- 如果将 Helm Chart 导出到本地，您需要将模板添加到具有管理权限的模板仓库。
- 无论选择哪种导出方式，您都可以参考创建原生应用 - 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

方法一

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. 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，文件格式为 **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。

方法二

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. 点击 **YAML** 标签，按需配置设置并预览 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；也可以取消勾选 隐藏 **managedFields** 字段 导出完整 YAML 文件。 注意：完整 YAML 主要用于运维和排障，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，您可以参考创建原生应用 - 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；您可以先脱离模板，转换为 `Custom Application`。

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。

- 将 YAML 导出到代码仓库：

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。该代码仓库分配的项目负责人可直接使用该 YAML 文件。
集成项目名称	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	分配给您使用的集成工具项目下的仓库地址。

导出方式	• 已有分支：将应用 YAML 导出到选定的分支。
	• 新建分支：基于选定的 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，源分支会自动删除。

文件路径	文件在代码仓库中保存的具体位置；您也可以输入文件路径，平台会根据输入在代码仓库中创建新路径。
提交信息	填写提交信息，用于标识此次提交的内容。
预览	预览待提交的 YAML 文件，并与代码仓库中已有的 YAML 文件进行差异对比，差异以颜色区分显示。

- 将 Kustomize 类型文件导出到本地，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

将 YAML 导出到 Git 代码仓库后，您可以参考创建 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。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

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*，可展示 Chart 的详细部署状态分析信息。

类型	说明

Initialized	表示 Chart 模板下载状态。
	• 状态为 True 表示 Chart 模板下载成功。
	• 状态为 False 表示 Chart 模板下载失败，失败原因可在消息栏查看。
	  • 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。
	• 状态为 True 表示所有校验均通过。
	• 状态为 False 表示存在校验未通过，失败原因可在消息栏查看。
	  • DependenciesCheckFailed：Chart 依赖检查失败。
	  • PermissionCheckFailed：当前用户缺少部分资源操作权限。
	  • ConsistentNamespaceCheckFailed：将模板应用部署为原生应用时，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。 • 状态为 True 表示 Chart 模板部署成功。 • 状态为 False 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栏查看。

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。对于非界面操作触发的应用更新，例如通过 API 调用更新应用，您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，优先删除最早的版本快照条目。

目录

创建版本快照

操作步骤

1. 进入 **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：您也可以分发应用，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. 进入 **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。

Menu

删除原生应用

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

处理资源耗尽错误

目录

Overview

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之前回收资源。

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 调度和内存杀手（OOM Killer）。同时提供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。策略由驱逐信号和阈值组成，可在节点配置中或通过命令行设置。驱逐分为：

- 硬驱逐：当阈值被超过时立即执行。
- 软驱逐：在采取行动前有宽限期。

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**：容器运行时使用的镜像和可写层存储（例如 Docker overlay2 驱动的 `/var/lib/docker/overlay2`，CRI-O 的 `/var/lib/containers/storage`）。

> **NOTE**

如果没有本地存储隔离（临时存储）或 XFS 配额（volumeConfig），则无法限制 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，请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

节点可根据以下信号触发驱逐：

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` 必须以百分比形式指定。
- 内存计算不包括可回收的非活动文件内存。
- 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，节点会先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：立即采取行动。
- 软阈值：宽限期后采取行动。

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 超出其请求资源时才会触发驱逐。

- **Capacity**：节点的总资源。
- **Allocatable**：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。

- 有 **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。
- 无 **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，节点驱逐 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。

QoS 等级	描述
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。

磁盘资源为 BestEffort 类型，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，OOM Killer 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 Killer 会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重启。

调度器与资源耗尽状态

调度器在调度 Pod 时会考虑节点状态。

节点状态	调度器行为
MemoryPressure	不调度 BestEffort Pod。

节点状态	调度器行为
DiskPressure	不调度任何新 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

守护进程集与资源耗尽处理

由守护进程集创建的 Pod 被驱逐后会立即重建。守护进程集应避免使用 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

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，从而使系统能够采取适当的措施，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 会将该 Pod 从 Service 的 Endpoint 列表中移除，直到探针成功。
- `startupProbe`：专门检查应用是否已成功启动。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 地址的指定端口和路径执行 HTTP `GET` 请求。如果响应码在 200 到 399 之间，则探针视为成功。

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

```
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。

- 使用场景：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：

```
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。

- 使用场景：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：

```
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），但可能无需重启即可恢复，使用就绪探针。这可防止流量被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外部依赖（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，给予应用足够时间启动后再进行首次探测。
- 调整 **periodSeconds** 和 **failureThreshold**：在快速检测故障和避免误报之间取得平衡。探测过于频繁或 failureThreshold 过低可能导致不必要的重启或不就绪状态。
- 日志调试：确保应用日志清晰记录健康检查端点调用和内部状态，有助于调试探针失败。
- 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 或很小
            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 秒开始检测
            periodSeconds: 10 # 每 10 秒检测一次
            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

通用参数

参数	描述
Initial Delay	`initialDelaySeconds`：探针开始前的宽限时间（秒）。默认值：`300`。
Period	`periodSeconds`：探针间隔（1-120秒）。默认值：`60`。
Timeout	`timeoutSeconds`：探针超时时长（1-300秒）。默认值：`30`。
Success Threshold	`successThreshold`：标记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：`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	自定义头部（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。

探针失败故障排查

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）。常见错误是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正确实现，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，进而导致探针失败。检查 Pod 资源使用情况（`kubectl top pod <pod-name>`），并考虑调整 `resources` 限制/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。

Menu

计算组件

Deployments

理解 Deployments

创建 Deployments

管理 Deployments

使用 CLI 进行故障排查

DaemonSets

理解守护进程集

创建守护进程集

管理守护进程集

StatefulSets

理解 StatefulSets

创建 StatefulSets

管理 StatefulSets

CronJobs

理解 CronJobs

创建 CronJobs

立即执行

删除 CronJobs

任务

了解任务

YAML 文件示例

执行概览

Pods

理解 Pods

YAML 文件示例

使用 CLI 管理 Pod

使用 Web 控制台管理 Pod

Containers

理解 Containers

理解 Ephemeral Containers

与 Containers 交互

Deployments

目录

理解 **Deployments**

请参考 Kubernetes 官方文档：Deployments ↗

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

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

使用 **Deployments**，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。
- 版本控制与回滚：跟踪 Deployment 的每个版本，出现问题时可以轻松回滚到之前的稳定版本。
- 零停机更新：使用滚动更新策略逐步更新应用，无服务中断。
- 自我修复：Deployment 会自动替换崩溃、终止或从节点移除的 Pod，确保指定数量的 Pod 始终可用。

工作原理：

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

创建 **Deployments**

使用 **CLI** 创建 **Deployment**

前提条件

- 确保已配置并连接到集群的 `kubectl`。

YAML 文件示例

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

通过 YAML 创建 Deployment

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

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

使用 Web 控制台创建 Deployment

前提条件

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

- 对于前者，管理员通常会将镜像仓库分配给你的项目，你可以使用其中的镜像。如果找不到所需镜像仓库，请联系管理员分配。
- 对于第三方平台的镜像仓库，确保当前集群能够直接拉取镜像。

操作步骤 - 配置基础信息

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

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

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

参数	说明
Replicas	定义 Deployment 中期望的 Pod 副本数（默认：`1`）。根据工作负载需求调整。
More > **Update Strategy**	配置 `rollingUpdate` 策略，实现零停机部署： **Max surge** (`maxSurge`)： • 更新期间允许超过期望副本数的最大 Pod 数量。 • 支持绝对值（如 `2`）或百分比（如 `20%`）。 • 百分比计算方式：`ceil(当前副本数 × 百分比)`。 • 示例：10 副本时，4.1 计算结果为 `5`。 **Max unavailable** (`maxUnavailable`)：

参数	说明
	• 更新期间允许不可用的最大 Pod 数量。 • 百分比值不可超过 `100%`。 • 百分比计算方式：`floor(当前副本数 × 百分比)`。 • 示例：10 副本时，4.9 计算结果为 `4`。 注意事项： 1. 默认值：未显式设置时，`maxSurge=1`，`maxUnavailable=1`。 2. 非运行状态的 **Pod**（如 `Pending`、`CrashLoopBackOff`）视为不可用。 3. 同时约束： • `maxSurge` 和 `maxUnavailable` 不能同时为 `0` 或 `0%`。 • 若两者百分比均计算为 `0`，Kubernetes 会强制设置 `maxUnavailable=1` 以保证更新进度。 示例： 对于 10 副本的 Deployment： • `maxSurge=2` → 更新期间总 Pod 数量为 `10 + 2 = 12`。 • `maxUnavailable=3` → 最小可用 Pod 数量为 `10 - 3 = 7`。 • 确保可用性的同时允许受控滚动更新。

操作步骤 - 配置 **Pod**

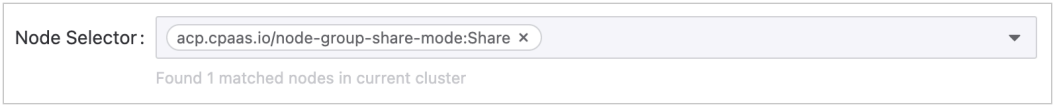
注意：在混合架构集群中部署单架构镜像时，请确保为 Pod 调度配置了正确的 Node Affinity 规则。

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	挂载持久卷到容器。支持的卷类型包括 `PVC`、`ConfigMap`、`Secret`、`emptyDir`、`hostPath` 等。具体实现请参见 卷挂载指南。

参数	说明
Pull Secret	仅当从第三方镜像仓库（通过手动输入镜像 URL）拉取镜像时需要。 注意：用于从私有仓库拉取镜像的认证 Secret。
Close Grace Period	Pod 接收到终止信号后允许的优雅关闭时间（默认：`30s`）。 - 期间 Pod 会完成正在处理的请求并释放资源。 - 设置为 `0` 会强制立即删除（SIGKILL），可能导致请求中断。

2. **Node Affinity** 规则

参数	说明
More > Node Selector	限制 Pod 调度到具有特定标签的节点（例如 `kubernetes.io/os: linux`）。 
More > Affinity	基于现有规则定义更细粒度的调度规则。 **Affinity** 类型： • **Pod Affinity**：将新 Pod 调度到已运行特定 Pod 的节点（相同拓扑域）。 • **Pod Anti-affinity**：避免新 Pod 与特定 Pod 共址。 执行模式： • `requiredDuringSchedulingIgnoredDuringExecution`：仅当规则满足时调度 Pod。 • `preferredDuringSchedulingIgnoredDuringExecution`：优先满足规则的节点，但允许例外。 配置字段： • `topologyKey`：定义拓扑域的节点标签（默认：`kubernetes.io/hostname`）。 • `labelSelector`：通过标签查询过滤目标 Pod。

3. 网络配置

- Kube-OVN

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

- Calico

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

操作步骤 - 配置容器

1. 在 **Container** 部分，参考以下说明配置相关信息。

参数	说明
资源请求与限制	• **Requests**：容器运行所需的最小 CPU/内存。 • **Limits**：容器运行时允许的最大 CPU/内存。单位定义见 资源单位。 命名空间超售比：

参数	说明
	• 无超售比： 若存在命名空间资源配额，容器请求/限制继承命名空间默认值（可修改）。 无命名空间配额时，无默认值，自定义请求。 • 有超售比： 请求自动计算为 `Limits / 超售比`（不可修改）。 约束条件： • 请求 ≤ 限制 ≤ 命名空间配额最大值。 • 超售比变更需重建 Pod 生效。 • 超售比启用时禁用手动请求配置。 • 无命名空间配额时无容器资源限制。
扩展资源	配置集群可用的扩展资源（如 vGPU、pGPU）。
卷挂载	持久存储配置。详见 存储卷挂载说明。 操作： • 已有 Pod 卷：点击 **Add** • 无 Pod 卷：点击 **Add & Mount** 参数： • `mountPath`：容器文件系统路径（如 `/data`） • `subPath`：卷内相对文件/目录路径。 对于 `ConfigMap` / `Secret`：选择具体键 • `readOnly`：只读挂载（默认读写） 详见 Kubernetes 卷↗。
端口	暴露容器端口。 示例：暴露 TCP 端口 `6379`，名称为 `redis`。 字段： • `protocol`：TCP/UDP

参数	说明
	• `Port`：暴露端口（如 `6379`） • `name`：符合 DNS 规范的标识符（如 `redis`）
启动命令与参数	覆盖默认 ENTRYPOINT/CMD： 示例 **1**：执行 `top -b` - **Command**：`["top", "-b"]` - 或 Command：`["top"]`，Args：`["-b"]` 示例 **2**：输出 `$MESSAGE`： `/bin/sh -c "while true; do echo $(MESSAGE); sleep 10; done"` 详见 定义命令↗。
More > 环境变量	• 静态值：直接键值对 • 动态值：引用 ConfigMap/Secret 键，Pod 字段（`fieldRef`），资源指标（`resourceFieldRef`） 注意：环境变量会覆盖镜像/配置文件中的设置。
More > 引用的 **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 > 健康检查	• **Liveness Probe**：检测容器健康（失败则重启） • **Readiness Probe**：检测服务可用性（失败则从服务端点移除） 详见 健康检查参数。

参数	说明
More > 日志文件	配置日志路径： - 默认收集 `stdout` - 文件模式：如 `/var/log/*.log` 要求： • 存储驱动 `overlay2` ：默认支持 • `devicemapper` ：需手动挂载 EmptyDir 到日志目录 • Windows 节点：确保父目录已挂载（如 `c:/a` 对应 `c:/a/b/c/*.log` ）
More > 排除日志文件	排除特定日志收集（如 `/var/log/aaa.log` ）。
More > 停止前执行	容器终止前执行命令。 示例： `echo "stop"` 注意：命令执行时间必须短于 Pod 的 `terminationGracePeriodSeconds` 。

2. 点击右上角的 **Add Container** 或 **Add Init Container**。

   参见 Init Containers↗。 Init Container：

   1. 在应用容器启动前运行（顺序执行）。
   2. 完成后释放资源。
   3. 允许删除条件：

      - Pod 有多个应用容器且至少有一个 Init Container。
      - 单应用容器的 Pod 不允许删除 Init Container。

3. 点击 **Create**。

参考信息

存储卷挂载说明

类型	用途
Persistent Volume Claim	绑定已有的 PVC 以请求持久存储。 注意：仅可选择已绑定 PVC（关联 PV）。未绑定 PVC 会导致 Pod 创建失败。
ConfigMap	挂载完整或部分 ConfigMap 数据为文件： • 完整 ConfigMap：在挂载路径下创建以键名命名的文件 • 子路径选择：挂载指定键（如 `my.cnf`）
Secret	挂载完整或部分 Secret 数据为文件： • 完整 Secret：在挂载路径下创建以键名命名的文件 • 子路径选择：挂载指定键（如 `tls.crt`）
Ephemeral Volumes	集群动态提供的临时卷，具备： • 动态配置 • 生命周期与 Pod 绑定 • 支持声明式配置 使用场景：临时数据存储。详见 Ephemeral Volumes
Empty Directory	Pod 内容器间共享的临时存储： • Pod 启动时在节点创建 • Pod 删除时销毁 使用场景：容器间文件共享、临时数据存储。详见 EmptyDir
Host Path	挂载宿主机目录（必须以 `/` 开头，如 `/volumepath`）。

健康检查

- 健康检查 YAML 文件示例
- Web 控制台健康检查配置参数

管理 Deployments

使用 CLI 管理 Deployment

查看 Deployment

- 查看 Deployment 是否已创建。

```
kubectl get deployments
```

- 获取 Deployment 详细信息。

```
kubectl describe deployments
```

更新 Deployment

按照以下步骤更新 Deployment：

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

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

或使用以下命令：

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

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

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

2. 查看滚动更新状态：

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

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

```
kubectl get rs
```

运行 `kubectl get pods` 应只显示新 Pods：

```
kubectl get pods
```

扩缩 **Deployment**

使用以下命令扩缩 Deployment：

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

回滚 **Deployment**

- 假设更新 Deployment 时输入了错误的镜像名 `nginx:1.161`（应为 `nginx:1.16.1`）：

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

- 滚动更新卡住。可通过查看滚动状态验证：

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

删除 **Deployment**

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

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

使用 **Web** 控制台管理 **Deployment**

查看 **Deployment**

可以查看 Deployment 以获取应用信息。

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

更新 **Deployment**

1. 在 **Container Platform**，导航至 **Workloads** > **Deployments**。
2. 找到要更新的 Deployment。
3. 在 **Actions** 下拉菜单中选择 **Update**，进入编辑 Deployment 页面。

删除 **Deployment**

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

使用 **CLI** 进行故障排查

当 Deployment 遇到问题时，以下是一些常用的排查方法。

查看 **Deployment** 状态

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

查看 **ReplicaSet** 状态

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

查看 Pod 状态

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

查看日志

```
kubectl logs <pod-name> -c <container-name> # 查看指定容器日志
kubectl logs <pod-name> --previous          # 查看之前终止容器的日志
```

进入 Pod 进行调试

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

检查健康检查配置

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

检查资源限制

确保容器资源请求和限制合理，避免因资源不足导致容器被杀死。

DaemonSets

目录

理解守护进程集

请参考官方 Kubernetes 文档：DaemonSets↗

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

WARNING

DaemonSet 操作注意事项

1. 行为特征

- **Pod** 分布：DaemonSet 会在每个符合条件的可调度节点上部署且仅部署一个 **Pod** 副本：
  - 在每个符合以下条件的可调度节点上部署且仅部署一个 Pod 副本：
    - 匹配 `nodeSelector` 或 `nodeAffinity` 条件（如果有指定）。
    - 节点状态不是 `NotReady`。
    - 节点没有 `NoSchedule` 或 `NoExecute` **Taints**，除非 Pod 模板中配置了相应的 **Tolerations**。
- **Pod** 数量计算公式：DaemonSet 管理的 Pod 数量等于符合条件的节点数量。
- 双重角色节点处理：同时担任 控制平面 和 工作节点 角色的节点，只会运行一个 DaemonSet 的 Pod 实例（无论其角色标签如何），前提是该节点可调度。

2. 关键限制（排除节点）

- 明确标记为 `Unschedulable: true` 的节点（例如通过 `kubectl cordon` 设置）。
- 处于 `NotReady` 状态的节点。
- 具有不兼容的 **Taints** 且 DaemonSet 的 Pod 模板中未配置相应 Tolerations 的节点。

创建守护进程集

使用 **CLI** 创建守护进程集

前提条件

- 确保已配置并连接到集群的 `kubectl`。

YAML 文件示例

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

```
      # priorityClassName: important
      terminationGracePeriodSeconds: 30
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

通过 YAML 创建守护进程集

```
# 步骤 1：执行以下命令创建 example-daemonSet.yaml 中定义的守护进程集
kubectl apply -f example-daemonSet.yaml

# 步骤 2：验证守护进程集及其管理的 Pod 状态：
kubectl get daemonset fluentd-elasticsearch # 查看守护进程集
kubectl get pods -l name=fluentd-elasticsearch -o wide # 查看该守护进程集管理的 Pod 及其所在节点
```

使用 Web 控制台创建守护进程集

前提条件

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

- 对于前者，管理员通常会将镜像仓库分配给您的项目，您可以使用该仓库中的镜像。如果找不到所需的镜像仓库，请联系管理员进行分配。
- 如果是第三方平台的镜像仓库，请确保当前集群可以直接拉取该镜像。

操作步骤 - 配置基本信息

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

> **INFO**

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

在 **Basic Info** 部分，配置守护进程集工作负载的声明式参数：

参数	说明
More > Update Strategy	配置守护进程集 Pod 的 `rollingUpdate` 策略，实现零停机更新。 最大不可用数 (`maxUnavailable`)：更新过程中允许暂时不可用的最大 Pod 数量。支持绝对值（如 1）或百分比（如 10%）。 示例：若有 10 个节点，`maxUnavailable` 设置为 10%，则允许最多 floor(10 * 0.1) = 1 个 Pod 不可用。 注意事项： • 默认值：若未显式设置，`maxSurge` 默认为 0，`maxUnavailable` 默认为 1（若以百分比指定，则为 10%）。 • 非运行中 **Pod**：处于 `Pending` 或 `CrashLoopBackOff` 等状态的 Pod 被视为不可用。 • 同时限制：`maxSurge` 和 `maxUnavailable` 不能同时为 0 或 0%。若两者百分比均计算为 0，Kubernetes 会强制将 `maxUnavailable` 设为 1 以保证更新进度。

操作步骤 - 配置 Pod

Pod 部分，请参考 Deployment - Configure Pod

操作步骤 - 配置容器

Containers 部分，请参考 Deployment - Configure Containers

操作步骤 - 创建

点击 **Create**。

点击 **Create** 后，守护进程集将：

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

管理守护进程集

使用 **CLI** 管理守护进程集

查看守护进程集

- 获取某命名空间下所有守护进程集的摘要信息：

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

- 获取指定守护进程集的详细信息，包括事件和 Pod 状态：

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

更新守护进程集

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

- 首先编辑 YAML 文件（如 `example-daemonset.yaml` ）进行所需修改，然后应用：

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

- 可以监控滚动更新的进度：

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

删除守护进程集

删除守护进程集及其管理的所有 Pod：

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

使用 **Web** 控制台管理守护进程集

查看守护进程集

1. 在 **Container Platform** 中，进入 **Workloads** > **DaemonSets**。
2. 找到要查看的守护进程集。
3. 点击守护进程集名称，查看 **Details**、**Topology**、**Logs**、**Events**、**Monitoring** 等信息。

更新守护进程集

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

删除守护进程集

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

StatefulSets

目录

删除 StatefulSet

理解 StatefulSets

请参考 Kubernetes 官方文档：StatefulSets ↗

StatefulSet 是 Kubernetes 的一种工作负载 API 对象，专为管理有状态应用设计，提供以下功能：

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

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

创建 StatefulSets

使用 CLI 创建 StatefulSet

前提条件

- 确保已配置并连接到集群的 `kubectl`。

YAML 文件示例

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

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

通过 YAML 创建 StatefulSet

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

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

使用 Web 控制台创建 StatefulSet

前提条件

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

- 对于前者，管理员通常会将镜像仓库分配给您的项目，您可以使用其中的镜像。如果找不到所需的镜像仓库，请联系管理员分配。
- 对于第三方平台的镜像仓库，请确保当前集群可以直接拉取镜像。

操作步骤 - 配置基础信息

1. 在 容器平台，左侧导航栏进入 工作负载 > **StatefulSets**。
2. 点击 创建 **StatefulSet**。
3. 选择或输入镜像，点击 确认。

INFO

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

在 基础信息 区域，配置 StatefulSet 工作负载的声明式参数：

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

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

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

操作步骤 - 配置 Pod

Pod 部分，请参考 Deployment - 配置 Pod

操作步骤 - 配置容器

Containers 部分，请参考 Deployment - 配置容器

操作步骤 - 创建

点击 创建。

健康检查

- 健康检查 YAML 文件示例
- Web 控制台健康检查配置参数

管理 StatefulSets

使用 CLI 管理 StatefulSet

查看 StatefulSet

您可以查看 StatefulSet 以获取应用信息。

- 查看 StatefulSet 是否已创建。

```
kubectl get statefulsets
```

- 获取 StatefulSet 详细信息。

```
kubectl describe statefulsets
```

扩缩容 **StatefulSet**

- 修改已有 StatefulSet 的副本数：

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

- 示例：

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

更新 **StatefulSet**（滚动更新）

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

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

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

- 然后可以监控滚动更新进度：

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

删除 **StatefulSet**

删除 StatefulSet 及其关联的 Pods：

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

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

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

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

使用 **Web** 控制台管理 **StatefulSet**

查看 **StatefulSet**

1. 在 容器平台，导航至 工作负载 > **StatefulSets**。
2. 找到您要查看的 StatefulSet。
3. 点击 StatefulSet 名称查看 详情、拓扑、日志、事件、监控 等信息。

更新 **StatefulSet**

1. 在 容器平台，导航至 工作负载 > **StatefulSets**。
2. 找到您要更新的 StatefulSet。
3. 在 操作 下拉菜单中选择 更新，进入编辑 StatefulSet 页面，可更新 `Replicas`、`image`、`updateStrategy` 等参数。

删除 **StatefulSet**

1. 在 容器平台，导航至 工作负载 > **StatefulSets**。
2. 找到您要删除的 StatefulSet。
3. 在 操作 下拉菜单中点击 删除 按钮并确认。

CronJobs

目录

理解 CronJobs

请参考官方 Kubernetes 文档：

- CronJobs ↗
- 使用 CronJob 运行自动化任务 ↗

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

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

创建 CronJobs

使用 CLI 创建 CronJob

前提条件

- 确保已配置并连接到集群的 `kubectl`。

YAML 文件示例

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

通过 YAML 创建 CronJobs

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

使用 Web 控制台创建 CronJobs

前提条件

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

- 对于集成仓库中的镜像，管理员通常会将镜像仓库分配给您的项目，允许您使用其中的镜像。如果找不到所需的镜像仓库，请联系管理员进行分配。
- 如果使用第三方镜像仓库，请确保当前集群内可以直接拉取该镜像。

操作步骤 - 配置基本信息

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

2. 点击 **Create CronJob**。

3. 选择或输入镜像，点击 **Confirm**。

   注意：仅在使用平台集成的镜像仓库中的镜像时支持镜像过滤。例如，集成项目名为 containers（docker-registry-projectname）表示平台项目名为 projectname，镜像仓库项目名为 containers。

4. 在 **Cron** 配置 部分，配置任务执行方式及相关参数。

   执行类型：

   - **Manual**：手动执行，需显式手动触发每次任务运行。
   - **Scheduled**：定时执行，需要配置以下调度参数：

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

   Job 历史保留：

   - 设置已完成 Job 的保留限制：
     - 历史限制：成功 Job 的历史保留数量（默认：20）
     - 失败 **Job**：失败 Job 的历史保留数量（默认：20）

- 超出保留限制时，最旧的 Job 会被优先垃圾回收。

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

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

操作步骤 - 配置 **Pod**

- **Pod** 部分，请参考 Deployment - Configure Pod

操作步骤 - 配置容器

- **Container** 部分，请参考 Deployment - Configure Containers

创建

- 点击 **Create**。

立即执行

定位 **CronJob** 资源

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

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

发起临时执行

- **Web** 控制台：立即执行

  1. 在 CronJob 列表右侧点击竖直省略号 (⋮)。
  2. 点击 **Execute Immediately**。（或者在 CronJob 详情页右上角点击操作菜单，选择 **Execute Immediately**）。

- **CLI**：

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

查看 **Job** 详情：

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

监控执行状态

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

删除 **CronJobs**

使用 **Web** 控制台删除 **CronJobs**

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

使用 **CLI** 删除 **CronJobs**

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

任务

目录

了解任务

请参考官方 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 来定义一个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）。
- 完成追踪：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
  backoffLimit: 3 # 最大重试次数
  template:
    spec:
      restartPolicy: Never # 任务专用策略（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 中的每次 Job 执行都会创建一个专门的 Job 对象，用户可以：

- 通过以下命令创建任务

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

- 通过以下命令跟踪任务生命周期

```
kubectl get jobs
```

- 通过以下命令查看执行详情

```
kubectl describe job/<job-name>
```

- 通过以下命令查看 **Pod** 日志

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

Pods

目录

理解 Pods

请参考 Kubernetes 官方网站文档：Pod ↗

Pod 是 Kubernetes 中可以创建和管理的最小可部署计算单元。一个 **Pod**（如鲸鱼群或豆荚）是一个或多个容器（例如 Docker 容器）的集合，共享存储和网络资源，并包含运行这些容器的规范。**Pods** 是所有更高级别控制器（如 **Deployments**、**StatefulSets**、**DaemonSets**）构建的基础单元。

YAML 文件示例

pod-example.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:latest # The container image to use.
      ports:
        - containerPort: 80 # Container ports exposed.
      resources: # Defines CPU and memory requests and limits for the container.
        requests:
          cpu: '100m'
          memory: '128Mi'
        limits:
          cpu: '200m'
          memory: '256Mi'
```

使用 **CLI** 管理 **Pod**

虽然 Pods 通常由更高级别的控制器管理，但直接使用 kubectl 操作 Pods 对于故障排查、检查和临时任务非常有用。

查看 **Pod**

- 列出当前命名空间下的所有 Pods：

```
kubectl get pods
```

- 列出所有命名空间下的所有 Pods：

```
kubectl get pods --all-namespaces
# 或简写为：
kubectl get pods -A
```

- 获取指定 Pod 的详细信息：

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

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

查看 **Pod** 日志

- 流式查看 Pod 中容器的日志（有助于调试）：

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

- 如果 Pod 中有多个容器，必须指定容器名称：

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

- 跟随日志输出（实时显示新日志）：

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

在 **Pod** 中执行命令

在 Pod 中的指定容器内执行命令（有助于调试，如访问 shell）：

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

# 示例（进入 shell）：
kubectl exec -it my-nginx-pod -n default -- /bin/bash
```

Pod 端口转发

将本地端口转发到 Pod 的端口，允许从本地机器直接访问 Pod 内运行的服务（适用于测试或无需外部暴露服务的直接访问）：

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

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

运行该命令后，可以通过浏览器访问 localhost:8080 来访问运行在 my-nginx-pod 中的 Nginx Web 服务器。

删除 Pod

- 删除指定的 Pod：

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

  # 示例
  kubectl delete pod my-nginx-pod -n default
  ```

- 按名称删除多个 Pods：

  ```
  kubectl delete pod <pod-name-1> <pod-name-2> -n <namespace>
  ```

- 根据标签选择器删除 Pods（例如删除所有标签为 app=nginx 的 Pods）：

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

使用 **Web** 控制台管理 **Pod**

查看 **Pod**

平台界面提供了 Pods 的多种信息，便于快速查看。

操作步骤

1. 进入 **Container Platform**，在左侧导航栏选择 **Workloads** > **Pods**。
2. 找到需要查看的 Pod。
3. 点击部署名称，查看 **Details**、**YAML**、**Configuration**、**Logs**、**Events**、**Monitoring** 等信息。

Pod 参数说明

以下是部分参数说明：

参数	说明
资源请求与限制	资源请求 和 限制 定义了 Pod 中容器的 CPU 和内存使用边界，这些值汇总后形成 Pod 的整体资源配置。这些值对于 Kubernetes 调度器高效地将 Pods 安排到节点上以及 kubelet 执行资源管理至关重要。 • 请求（**Requests**）：容器调度和运行所需的最低保证 CPU/内存。Kubernetes 调度器根据此值决定 Pod 可运行的 **Node**。 • 限制（**Limits**）：容器运行时允许使用的最大 CPU/内存。超过 CPU 限制会被限流，超过内存限制则容器会被终止（OOM Killed）。 详细单位定义（如 `m` 表示毫 CPU，`Mi` 表示 Mebibytes）请参考 Resource Units。 **Pod** 级资源计算逻辑 Pod 的有效 CPU 和内存请求及限制值由其各个容器规格的求和和最大值计算得出。Pod 级请求和限制的计算方法类似，本文以限制值为例说明。当 Pod 仅包含标准容器（业务容器）时：Pod 的有效 CPU/内存限制值为所有容器 CPU/内存限

参数	说明
	制值之和。 示例：若 Pod 包含两个容器，CPU/内存限制分别为 100m/100Mi 和 50m/200Mi，则 Pod 汇总的 CPU/内存限制为 150m/300Mi。当 Pod 同时包含 initContainers 和标准容器时，Pod 的 CPU/内存限制计算步骤如下： • 1. 计算所有 initContainers 中 CPU/内存限制的最大值。 • 2. 计算所有标准容器 CPU/内存限制的总和。 • 3. 比较步骤 1 和步骤 2 的结果，Pod 的综合 CPU 限制为两者中较大值，内存限制同理。 计算示例：若 Pod 包含两个 initContainers，CPU/内存限制分别为 100m/200Mi 和 200m/100Mi，则 initContainers 的最大有效 CPU/内存限制为 200m/200Mi。同时，若 Pod 还包含两个标准容器，CPU/内存限制分别为 100m/100Mi 和 50m/200Mi，则标准容器的总限制为 150m/300Mi。因此，Pod 的综合 CPU/内存限制为 CPU max(200m, 150m) 和内存 max(200Mi, 300Mi)，即 200m/300Mi。
来源	管理该 Pod 生命周期的 Kubernetes 工作负载控制器，包括 **Deployments**、**StatefulSets**、**DaemonSets**、**Jobs**。
重启次数	Pod 启动以来其内容器重启的次数。重启次数过多通常表明应用或其运行环境存在问题。
节点	Pod 当前调度并运行所在的 Kubernetes 节点名称。
服务账户	服务账户是 Kubernetes 对象，为 Pod 内运行的进程和服务提供身份认证，使其能够访问 Kubernetes APIServer。该字段通常仅在当前登录用户拥有平台管理员角色或平台审计员角色时可见，允许查看服务账户的 YAML 定义。

删除 **Pod**

删除 Pod 可能影响计算组件的运行，请谨慎操作。

使用场景

- 快速恢复 Pod 到期望状态：当 Pod 处于影响业务的状态（如 `Pending` 或 `CrashLoopBackOff`）时，排查错误信息后手动删除 Pod，有助于其快速恢复到期望状态（如 `Running`）。此时，删除的 Pod 会在当前节点重建或重新调度。

- 资源清理与运维管理：部分 Pod 达到指定阶段后不再变化，这类 Pod 通常数量较多，影响其他 Pod 的管理。待清理的 Pod 可能包括因节点资源不足而处于 `Evicted` 状态的 Pod，或因周期性定时任务触发而处于 `Completed` 状态的 Pod。此类 Pod 删除后将不再存在。

  注意：对于定时任务，如果需要查看每次任务执行的日志，不建议删除对应的 `Completed` 状态 Pod。

操作步骤

1. 进入 **Container Platform**。

2. 在左侧导航栏点击 **Workloads** > **Pods**。

3. （单个删除）点击待删除 Pod 右侧的⋮按钮 > **Delete**，并确认。

4. （批量删除）勾选待删除的 Pods，点击列表上方的 **Delete**，并确认。

Containers

目录

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 功能提供了一种强大的方式，通过向现有 Pod 注入专用的调试工具（系统、网络和磁盘工具）来调试正在运行的 containers。

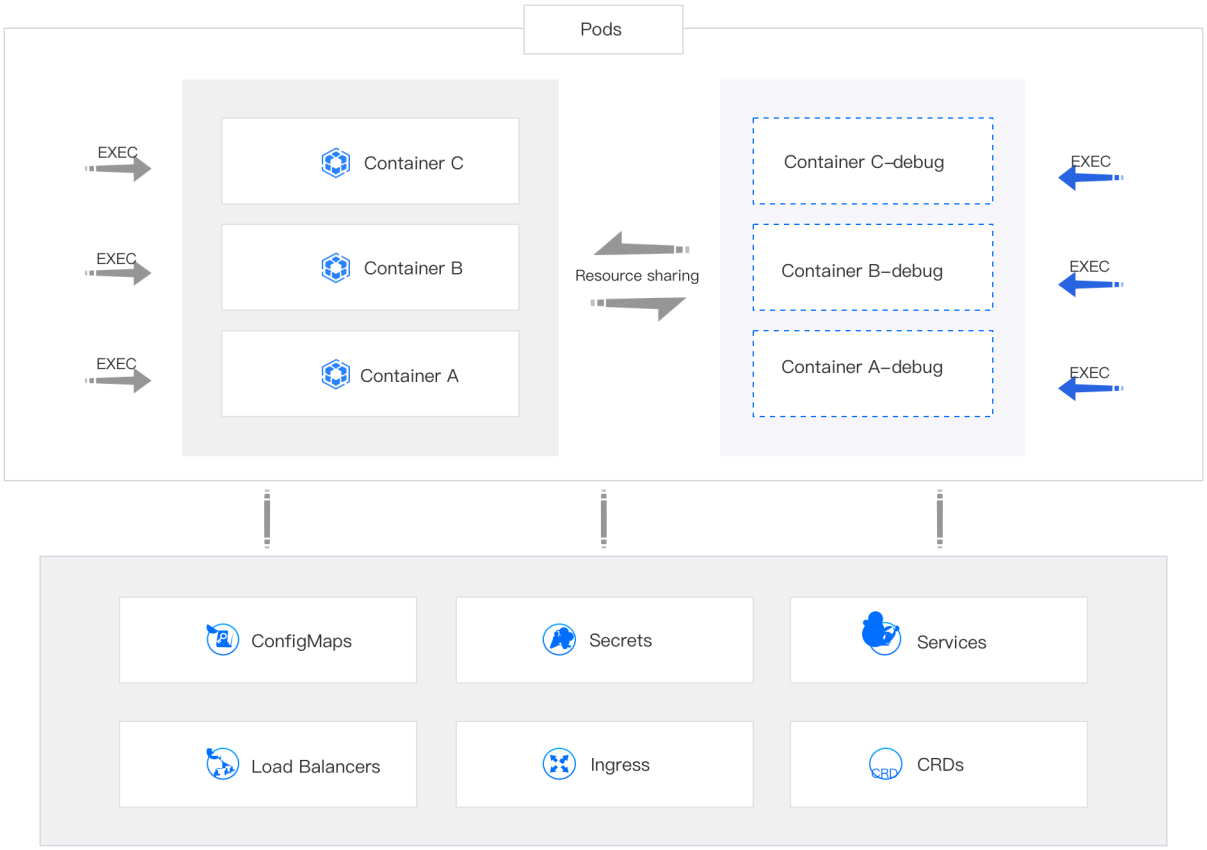
虽然通常可以使用 kubectl exec 直接在运行中的 container 内执行命令，但许多生产环境的 container 镜像故意保持精简，可能缺少关键的调试工具（例如 bash、net-tools、tcpdump），以减小镜像体积和攻击面。Ephemeral Containers 解决了这一限制，提供了一个预配置的环境，内置丰富的调试工具，非常适合以下场景：

- 故障诊断：当主应用 container 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 的运行环境中进行更深入的交互式排查。
- 配置调优与实验：如果当前应用配置表现不佳，可能需要临时调整组件设置或测试新配置，直接在运行中的 container 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 container。它与现有的主 `containers` 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 的诊断结果直接关联于同一 Pod 内主应用 `containers` 的行为和状态。



:::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 Pods，通常通过 API 调用（如 kubectl debug）实现。 * 通过调试功能创建的 **Ephemeral Containers** 不具备资源（CPU/内存）或调度保证（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，它们仅限于调试用途。 * 如果 Pod 所在的 Node 正处于高资源利用率或接近资源枯竭状态，使用调试功能时需谨慎。即使 Ephemeral Container 资源占用极小，也可能在严重资源压力下加剧 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-container-name> -n
<namespace>
# --image：指定包含必要工具的调试镜像（例如 busybox、ubuntu、nicolaka/netshoot）。
# --target：（可选）指定 Pod 中目标 container 的名称。若省略且只有一个 container，则默认
该 container；若有多个，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，确保有一个正在运行的 Pod：

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

- 现在，在 `my-nginx-pod` 内创建一个名为 `debugger` 的 ephemeral 调试 container，目标为 my-nginx-container，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 内的 shell，可以使用 busybox 工具调试 my-nginx-container。

- 查看 Pod 附加的 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**，在左侧导航栏选择 **Workloads** > **Pods**。
2. 找到想要查看的 Pod，点击⋮> **Debug**。
3. 选择 Pod 中想要调试的具体 container。
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，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` 命令直接与运行中的 container 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 中的特定 container 内执行命令（例如获取 shell、运行诊断命令等）：

```
kubectl exec -it <pod-name> -c <container-name> -n <namespace> -- <command>
# -it：确保交互模式和伪终端（TTY），适合 shell 会话。
# -c：指定 Pod 中目标 container 名称。Pod 仅有一个 container 时可省略。
# --：分隔 kubectl 参数和容器内执行的命令。
```

- 示例：进入 `my-nginx-pod` 中 `nginx` 的 Bash shell

```
kubectl exec -it my-nginx-pod -c nginx -n default -- /bin/bash
```

- 示例：列出 container `/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 container：

```
kubectl cp <local-file-path> <namespace>/<pod-name>:<container-file-path> -c
<container-name>
# -c：（可选）指定 Pod 中目标 container 名称（多 container Pod 时需要）。

# 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 container 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 **Container**

你可以使用 `kubectl exec` 命令进入 container 内部实例，在 Web 控制台窗口执行命令行操作。同时，支持文件上传和下载功能，方便在 container 内传输文件。

1. 进入 **Container Platform**，在左侧导航栏选择 **Application** > **Applications**。
2. 点击 ***Application Name***。
3. 找到关联的工作负载（如 Deployment、StatefulSet），点击 **EXEC**，然后选择想要进入的具体 *Pod Name*。再选择 **EXEC** > ***Container Name***。
4. 输入想要执行的命令。
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 container（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 container 下载到本地进行分析。

通过 **Pod** 进入 **Container**

1. 进入 **Container Platform**，在左侧导航栏选择 **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 中想要进入的具体 Container Name。
3. 输入想要执行的命令。
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 container（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 container 下载到本地进行分析。

使用 **Helm charts**

目录

1. 了解 Helm

Helm 是一个包管理器，简化了在 Alauda Container Platform 集群上部署应用和服务的流程。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 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，都会创建一个递增的修订版本。

1.1. 主要特性

Helm 提供以下功能：

- 在 chart 仓库中搜索大量 charts
- 修改现有的 charts
- 使用 Kubernetes 资源创建自己的 charts
- 打包应用并以 charts 形式分享

1.2. 目录

Catalog 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，使开发者更便捷地管理、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	

术语	定义	备注
Helm Charts	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 部署主要通过自定义资源 **HelmRequest** 管理。此方法扩展了标准 Helm 功能，并无缝集成到 Kubernetes 原生资源模型中。

HelmRequest 与 Helm 的区别

标准 Helm 使用 CLI 命令管理 releases，而 Alauda Container Platform 使用 HelmRequest 资源定义、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，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 和 Application 资源在概念上相似，用户可能希望统一查看。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 会显示额外字段及链接至对应的 Application 页面。

部署工作流

通过 HelmRequest 部署 charts 的工作流包括：

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 并同步至 Applications
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 releases 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 将 Helm Charts 部署为 Applications

2.1 工作流概览

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 上传至仓库：

```
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：`docker pull nginx`

2. 保存为 tar 包：`docker save nginx > nginx.latest.tar`

3. 加载并推送至私有仓库：

```
docker load -i nginx.latest.tar
docker tag nginx:latest 192.168.80.8:30050/nginx:latest
docker 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/$NAMESPACE/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\":\"IfNotPresent\"}}"
    },
    "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/$NAMESPACE/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/$NAMESPACE/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-
system/chartrepos/test \
--header 'Authorization:Bearer $API_TOKEN'
```

3 通过 UI 将 Helm Charts 部署为 Applications

3.1 工作流概览

将模板添加至可管理仓库 → 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 将模板添加至可管理仓库

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 的项目所有者可直接使用该模板。
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**，上传本地模板至仓库。
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，请根据提示进行排查。

   注意：非法文件格式表示上传的压缩包内文件存在问题，如内容缺失或格式错误。

3.4 删除模板的特定版本

如果某版本模板不再适用，可删除该版本。

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。

Menu

配置

Configuring ConfigMap

Understanding Config Maps

Config Map 限制

ConfigMap 示例

通过 Web 控制台创建 ConfigMap

通过 CLI 创建 ConfigMap

操作

通过 CLI 查看、编辑和删除

Pod 中使用 ConfigMap 的方式

ConfigMap 与 Secret 的对比

Configuring Secrets

理解 Secrets

创建 Opaque 类型的 Secret

创建 Docker registry 类型的 Secret

创建 Basic Auth 类型的 Secret

创建 SSH-Auth 类型的 Secret

创建 TLS 类型的 Secret

通过 Web 控制台创建 Secret

如何在 Pod 中使用 Secret

后续操作

操作

Configuring ConfigMap

Config maps 允许您将配置工件与镜像内容解耦，以保持容器化应用的可移植性。 以下章节定义了 config maps 以及如何创建和使用它们。

目录

Understanding Config Maps

许多应用程序需要通过配置文件、命令行参数和环境变量的某种组合进行配置。在 OpenShift Container Platform 中，这些配置工件与镜像内容解耦，以保持容器化应用的可移植性。

`ConfigMap` 对象提供了向容器注入配置信息的机制，同时保持容器对 OpenShift Container Platform 的无感知。config map 可用于存储细粒度的信息，如单个属性，也可存储粗粒度的信息，如整个配置文件或 JSON 数据块。

`ConfigMap` 对象保存键值对形式的配置信息，这些信息可以被 Pod 消费，或用于存储系统组件（如控制器）的配置信息。例如：

```
# my-app-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-app-config
  namespace: default
data:
  app_mode: "development"
  feature_flags: "true"
  database.properties: |-
    jdbc.url=jdbc:mysql://localhost:3306/mydb
    jdbc.username=user
    jdbc.password=password
  log_settings.json: |-
    {
      "level": "INFO",
      "format": "json"
    }
```

注意：当您从二进制文件（如镜像）创建 config map 时，可以使用 `binaryData` 字段。

配置数据可以通过多种方式在 Pod 中被消费。config map 可用于：

- 填充容器中的环境变量值
- 设置容器的命令行参数
- 在卷中填充配置文件

用户和系统组件都可以将配置信息存储在 config map 中。 config map 类似于 secret，但设计上更方便处理不包含敏感信息的字符串。

Config Map 限制

- 必须先创建 config map，才能在 Pod 中消费其内容。
- 控制器可以编写为容忍缺失的配置信息。请根据具体使用 config map 配置的组件逐一确认。
- `ConfigMap` 对象存在于项目中。
- 只能被同一项目中的 Pod 引用。
- Kubectl 仅支持对从 API 服务器获取的 Pod 使用 config map。这包括通过 CLI 创建的 Pod，或通过复制控制器间接创建的 Pod。不包括通过 OpenShift Container Platform 节点的 `--manifest-url` 标志、`--config` 标志或其 REST API 创建的 Pod，因为这些不是常见的 Pod 创建方式。

> **NOTE**
>
> Pod 只能使用同一命名空间内的 ConfigMaps。

ConfigMap 示例

您现在可以在 `Pod` 中使用 app-config。

```yaml
# app-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: k-1
data:
  APP_ENV: "production"
  LOG_LEVEL: "debug"
```

通过 Web 控制台创建 ConfigMap

1. 进入 **Container Platform**。
2. 在左侧边栏点击 **Configuration** > **ConfigMap**。

3. 点击 **Create ConfigMap**。

4. 参考以下说明配置相关参数。

参数	说明
Entries	指 `key:value` 键值对，支持 添加 和 导入 两种方式。 • 添加：可以逐条添加配置项，也可以在 Key 输入框中粘贴一行或多行 key=value 格式的内容批量添加配置项。 • 导入：导入不超过 1M 的文本文件，文件名作为 key，文件内容作为 value，填充为一个配置项。
Binary Entries	指不超过 1M 的二进制文件，文件名作为 key，文件内容作为 value，填充为一个配置项。 注意：创建 ConfigMap 后，导入的文件无法修改。

批量添加格式示例：

```
# 每行一个 key=value，多个键值对必须分行，否则粘贴后无法正确识别。
key1=value1
key2=value2
key3=value3
```

5. 点击 **Create**。

通过 CLI 创建 ConfigMap

```
kubectl create configmap app-config \
  --from-literal=APP_ENV=production \
  --from-literal=LOG_LEVEL=debug
```

或者从文件创建：

```
kubectl apply -f app-config.yaml -n k-1
```

操作

您可以点击列表页右侧的 (⋮) 按钮，或在详情页右上角点击 **Actions**，根据需要更新或删除 ConfigMap。

ConfigMap 的变更会影响引用该配置的工作负载，请提前阅读操作说明。

操作	说明
更新	• 添加或更新 ConfigMap 后，任何通过环境变量引用该 ConfigMap（或其配置项）的工作负载需要重建 Pod，才能使新配置生效。 • 对于导入的二进制配置项，仅支持键的更新，不支持值的更新。
删除	删除 ConfigMap 后，任何通过环境变量引用该 ConfigMap（或其配置项）的工作负载在重建 Pod 时，若找不到引用源，可能会受到不利影响。

通过 **CLI** 查看、编辑和删除

```
kubectl get configmap app-config -n k-1 -o yaml
kubectl edit configmap app-config -n k-1
kubectl delete configmap app-config -n k-1
```

Pod 中使用 **ConfigMap** 的方式

作为环境变量

```
envFrom:
  - configMapRef:
      name: app-config
```

每个键都会成为容器中的一个环境变量。

作为卷中的文件

```
volumes:
  - name: config-volume
    configMap:
      name: app-config

volumeMounts:
  - name: config-volume
    mountPath: /etc/config
```

每个键对应 `/etc/config` 下的一个文件，文件内容为对应的值。

作为单个环境变量

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

ConfigMap 与 Secret 的对比

特性	ConfigMap	Secret
数据类型	非敏感	敏感（如密码）
编码方式	明文	Base64 编码
使用场景	配置、标志	密码、令牌

Configuring Secrets

目录

理解 Secrets

在 Kubernetes (k8s) 中，Secret 是一个用于存储和管理敏感信息的基础对象，例如密码、OAuth 令牌、SSH 密钥、TLS 证书和 API 密钥。其主要目的是防止敏感数据直接嵌入 Pod 定

义或容器镜像中，从而增强安全性和可移植性。

Secrets 类似于 ConfigMaps，但专门用于机密数据。它们通常以 base64 编码的形式存储，并可以通过多种方式被 Pod 使用，包括挂载为卷或作为环境变量暴露。

使用特点

- 增强安全性：相比明文配置映射（Kubernetes ConfigMap），Secrets 通过 Base64 编码存储敏感信息，结合 Kubernetes 的访问控制能力，显著降低数据泄露风险。
- 灵活管理：使用 Secrets 提供了比将敏感信息硬编码到 Pod 定义文件或容器镜像中更安全且灵活的方法。这种分离简化了敏感数据的管理和修改，无需更改应用代码或容器镜像。

支持的类型

Kubernetes 支持多种类型的 Secrets，针对不同使用场景设计。平台通常支持以下类型：

- **Opaque**：通用 Secret 类型，用于存储任意键值对的敏感数据，如密码或 API 密钥。
- **TLS**：专门用于存储 TLS（传输层安全）协议的证书和私钥信息，常用于 HTTPS 通信和安全的 ingress。
- **SSH Key**：用于存储 SSH 私钥，通常用于安全访问 Git 仓库或其他支持 SSH 的服务。
- **SSH Authentication (kubernetes.io/ssh-auth)**：存储通过 SSH 协议传输的数据的认证信息。
- **Username/Password (kubernetes.io/basic-auth)**：用于存储基本认证凭据（用户名和密码）。
- **Image Pull Secret (kubernetes.io/dockerconfigjson)**：存储从私有镜像仓库（Docker Registry）拉取容器镜像所需的 JSON 认证字符串。

使用方式

Secrets 可以通过不同方式被 Pod 内的应用使用：

- 作为环境变量：Secret 中的敏感数据可以直接注入到容器的环境变量中。

- 作为挂载文件（卷）：Secrets 可以挂载为 Pod 中的文件，应用可以从指定路径读取敏感数据。

注意：工作负载中的 Pod 实例只能引用同一命名空间内的 Secrets。有关高级用法和 YAML 配置，请参阅 Kubernetes 官方文档↗。

创建 Opaque 类型的 Secret

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

YAML

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

你可以这样解码：

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

创建 Docker registry 类型的 Secret

```
kubectl create secret docker-registry my-docker-creds \
  --docker-username=myuser \
  --docker-password=mypass \
  --docker-server=https://index.docker.io/v1/ \
  --docker-email=my@example.com
```

YAML

```
apiVersion: v1
kind: Secret
metadata:
  name: my-docker-creds
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson:
eyJhdXRocyI6eyJodHRwczovL2luZGV4LmRvY2tlci5pby92MS8iOnsidXNlcm5hbWUiOiJteXVzZXIiLCJwYXNzd29yZ
```

K8s 会自动将你的用户名、密码、邮箱和服务器信息转换为 Docker 标准登录格式：

```
{
  "auths": {
    "https://index.docker.io/v1/": {
      "username": "myuser",
      "password": "mypass",
      "email": "my@example.com",
      "auth": "bXl1c2VyOm15cGFzcw=="  # base64 编码的 username:password
    }
  }
}
```

该 JSON 会被 base64 编码后用作 Secret 的 data 字段值。

在 Pod 中使用：

```
imagePullSecrets:
  - name: my-docker-creds
```

创建 **Basic Auth** 类型的 **Secret**

```
apiVersion: v1
kind: Secret
metadata:
  name: basic-auth-secret
type: kubernetes.io/basic-auth
stringData:
  username: myuser
  password: mypass
```

创建 **SSH-Auth** 类型的 **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** 类型的 **Secret**

使用场景：TLS 证书（用于 Ingress、webhooks 等）

```
kubectl create secret tls tls-secret \
--cert=path/to/tls.crt \
--key=path/to/tls.key
```

YAML

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: tls-secret
type: kubernetes.io/tls
data:
  tls.crt: <base64>
  tls.key: <base64>
```

通过 **Web** 控制台创建 **Secret**

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Configuration** > **Secrets**。
3. 点击 **Create Secret**。
4. 配置参数。

   注意：在表单视图中，输入的用户名和密码等敏感数据会自动以 Base64 格式编码后存储到 Secret 中。转换后的数据可在 YAML 视图中预览。
5. 点击 **Create**。

如何在 **Pod** 中使用 **Secret**

作为环境变量

```
env:
  - name: DB_USERNAME
    valueFrom:
      secretKeyRef:
        name: my-secret
        key: username
```

从名为 `my-secret` 的 Secret 中获取键为 `username` 的值，并赋值给环境变量 `DB_USERNAME`。

作为挂载文件（卷）

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

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

后续操作

在同一命名空间中创建原生应用的工作负载时，可以引用已创建的 Secrets。

操作

你可以点击列表页右侧的 (⋮) 或详情页右上角的 **Actions**，根据需要更新或删除 Secret。

操作	描述
更新	添加或更新 Secret 后，已通过环境变量引用该 Secret（或其配置项）的工作负载需要重建 Pod，才能使新配置生效。

操作	描述
删除	• 删除 Secret 后，已通过环境变量引用该 Secret（或其配置项）的工作负载在重建 Pod 时可能因找不到引用源而受到影响。 • 请勿删除平台自动生成的 Secrets，否则可能导致平台无法正常运行。例如：包含命名空间资源认证信息的 service-account-token 类型 Secrets 以及系统命名空间（如 kube-system）中的 Secrets。

Menu

应用可观测

监控面板

前置条件

命名空间级别监控面板

工作负载级别监控

Logs

操作步骤

实时事件

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，以及 CPU/内存使用趋势和排名。
- 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

前置条件

- 监控插件安装

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. 由 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** 上展示命名空间级监控面板：
   - `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. 点击列表中的某个 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。高利用率表示限制不足。 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。高比例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）进出情况。
网络包	接收速率/发送速率	Pod 接收/发送的网络包数量（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各 Pod 在特定时间点的指标和 PromQL 表达式。若 Pod 数量超过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，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

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	说明
Pod/Container	通过下拉选择器在 Pods 和 Containers 之间切换，查看对应日志。
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	说明
Wrap Line	切换长日志条目的换行显示（默认开启）。
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 Observe 下的 Logs。

实时事件

由 Kubernetes 资源状态变化和操作状态更新产生的事件信息，集成了可视化查询界面。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本原因。

目录

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的事件。点击事件卡片查看完整事件详情。每个卡片显示：

- 资源类型：以图标缩写表示的 Kubernetes 资源类型：
  - `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` ：需要立即关注
- 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。

Menu

实用指南

设置定时任务触发规则

转换时间

编写 Crontab 表达式

设置定时任务触发规则

定时任务的定时触发规则支持输入 Crontab 表达式。

目录

转换时间

时间转换规则：本地时间 - 时差 = UTC

以 北京时间转 **UTC** 时间 为例进行说明：

北京位于东八区，北京时间和 UTC 时间的时差是 8 小时，时间转换规则：

```
北京时间 - 8 = UTC
```

示例 **1**：北京时间 9 点 42 分，转换成 UTC 时间：42 09 - 00 08 = 42 01，即 UTC 时间为凌晨 1 点 42 分。

示例 **2**：北京时间凌晨 4 点 32 分，转换成 UTC 时间：32 04 - 00 08 = -68 03，如果结果为负数，表明是前一天，需要再进行一次转换：-68 03 + 00 24 = 32 20，即 UTC 时间为前一天晚上 8 点 32 分。

编写 Crontab 表达式

Crontab 基本格式及取值范围：`分钟 小时 日 月 星期`，对应的取值范围请参见下表：

分钟	小时	日	月	星期
[0-59]	[0-23]	[1-31]	[1-12] 或 [JAN-DEC]	[0-6] 或 [SUN-SAT]

`分钟 小时 日 月 星期` 位允许输入的特殊字符包括：

- `,`：值列表分隔符，用于指定多个值。例如：`1,2,5,7,8,9`。
- `-`：用户指定值的范围。例如：`2-4`，表示 2、3、4。
- `*`：代表整个时间段。例如：用作分钟时，表示每分钟。
- `/`：用于指定值的增加幅度。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 `12 月 25 日 18:30:00` 触发任务。
- 输入 `30 18 25 * 6` 表示 `每周六的 18:30:00` 触发任务。
- 输入 `30 18 * * 6` 表示 `每周六的 18:30:00` 触发任务。
- 输入 `* 18 * * *` 表示 `从 18:00:00 开始，每过一分钟（包括 18:00:00）` 触发任务。
- 输入 `0 18 1,10,22 * *` 表示 `每月 1、10、22 日的 18:00:00` 触发任务。
- 输入 `0,30 18-23 * * *` 表示 `每天 18:00 至 23:00 之间，每个小时的 00 分和 30 分` 触发任务。
- 输入 `* */1 * * *` 表示 `每分钟` 触发任务。
- 输入 `* 2-7/1 * * *` 表示 `每天 2 点到 7 点之间，每分钟` 触发任务。
- 输入 `0 11 4 * mon-wed` 表示 `每月 4 日与每周一到周三的 11 点` 触发任务。

Menu

镜像

镜像概述

实用指南

Managing images

镜像概述

目录

理解容器和镜像

容器和镜像是创建和管理容器化软件时需要理解的重要概念。镜像包含一组准备运行的软件，而容器是镜像的一个运行实例。不同版本通过同一镜像名称上的不同标签来表示。

镜像

Alauda Container Platform 中的容器基于 OCI 或 Docker 格式的容器镜像。镜像是一个二进制文件，包含运行单个容器所需的所有内容，以及描述其需求和能力的元数据。

你可以将其视为一种打包技术。容器只能访问镜像中定义的资源，除非在创建时授予了额外访问权限。通过在多个主机上的多个容器中部署相同的镜像，并在它们之间进行负载均衡，

Alauda Container Platform 可以为打包在镜像中的服务提供冗余和横向扩展能力。

你可以直接使用 `nerdctl` 或 `docker` CLI 来构建镜像，但 Alauda Container Platform 也提供了构建器镜像，帮助通过将你的代码或配置添加到现有镜像中来创建新镜像。

由于应用程序会随着时间发展，单个镜像名称实际上可以指代同一镜像的多个不同版本。每个不同的镜像通过其哈希值唯一标识，哈希值是一个长的十六进制数字，如 fd44297e2ddb050ec4f…，通常缩短为 12 个字符，如 fd44297e2ddb。

镜像仓库

镜像仓库是一个内容服务器，可以存储和提供容器镜像。例如：

- Docker Hub↗
- Quay.io Container Registry↗
- Alauda Container Platform Registry

仓库包含一个或多个镜像库，镜像库中包含一个或多个带标签的镜像。Alauda Container Platform 可以提供自己的镜像仓库，用于管理自定义容器镜像。

镜像库

镜像库是相关容器镜像及其标签的集合。例如，Alauda Container Platform 的 Jenkins 镜像位于以下镜像库中：

```
docker.io/alauda/jenkins-2-centos7
```

镜像标签

镜像标签是应用于镜像库中容器镜像的标签，用于区分镜像流中的特定镜像。通常，标签表示某种版本号。例如，这里的 :v3 .11.59-2 是标签：

```
docker.io/alauda/jenkins-2-centos7:v3.11.59-2
```

你可以为镜像添加额外的标签。例如，一个镜像可能被赋予 :v3 .11.59-2 和 :latest 两个标签。

镜像 ID

镜像 ID 是一个 SHA（安全哈希算法）代码，可用于拉取镜像。SHA 镜像 ID 不会改变。特定的 SHA 标识符始终引用完全相同的容器镜像内容。例如：

```
docker.io/alauda/jenkins-2-centos7@sha256:ab312bda324
```

容器

Alauda Container Platform 应用的基本单元称为容器。Linux 容器技术是一种轻量级机制，用于隔离运行的进程，使其仅限于与指定资源交互。容器一词定义为容器镜像的特定运行或暂停实例。

许多应用实例可以在单个主机上的容器中运行，彼此之间无法访问对方的进程、文件、网络等。通常，每个容器提供单一服务，通常称为微服务，例如 Web 服务器或数据库，但容器也可用于任意工作负载。

Linux 内核多年来一直在集成容器技术的能力。Docker 项目开发了一个方便的管理接口，用于管理主机上的 Linux 容器。最近，Open Container Initiative↗ 制定了容器格式和容器运行时的开放标准。Alauda Container Platform 和 Kubernetes 增加了跨多主机安装编排 OCI 和 Docker 格式容器的能力。

虽然使用 Alauda Container Platform 时你不会直接与容器运行时交互，但理解其能力和术语对于理解它们在 Alauda Container Platform 中的作用以及你的应用如何在容器内运行非常重要。

Menu

实用指南

Creating images

Learning container best practices

Including metadata in images

Managing images

Image pull policy overview

Allowing pods to reference images from other secured registries

Creating a pull secret

Using a pull secret in a workload

Creating images

学习如何基于预构建镜像创建您自己的容器镜像，这些预构建镜像已准备好为您提供帮助。该过程包括学习编写镜像的最佳实践、定义镜像元数据、测试镜像，以及使用自定义构建流程创建可用于 Alauda Container Platform Registry 的镜像。创建镜像后，您可以将其推送到 **Alauda Container Platform Registry**。

目录

Learning container best practices

在为 Alauda Container Platform 创建容器镜像时，作为镜像作者需要考虑多项最佳实践，以确保镜像使用者获得良好体验。由于镜像旨在保持不可变并按原样使用，以下指南有助于确保您的镜像高度可用且易于在 Alauda Container Platform 上使用。

General container image guidelines

以下指南适用于一般容器镜像的创建，与镜像是否在 Alauda Container Platform 上使用无关。

复用镜像

尽可能基于合适的上游镜像使用 FROM 语句构建您的镜像。这确保当上游镜像更新时，您的镜像可以轻松获得安全修复，而无需您直接更新依赖项。

此外，在 FROM 指令中使用标签，例如 `alpine:3.20` ，可以让用户明确知道您的镜像基于哪个版本。使用非 latest 的标签可避免您的镜像受到上游镜像最新版本中可能引入的破坏性更改的影响。

保持标签内兼容性

为自己的镜像打标签时，尽量保持标签内的向后兼容性。例如，如果您提供了名为 image 的镜像，当前包含版本 `1.0` ，您可以提供标签 `image:v1` 。当您更新镜像时，只要保持与原镜像兼容，就可以继续使用 `image:v1` 标签，下游用户可以在不被破坏的情况下获取更新。

如果后续发布了不兼容的更新，则切换到新标签，例如 `image:v2` 。这样允许下游用户根据需要升级到新版本，而不会被不兼容的镜像意外破坏。使用 `image:latest` 的下游用户则承担了引入不兼容更改的风险。

避免多进程

不要在一个容器内启动多个服务，例如数据库和 `SSHD` 。这没有必要，因为容器轻量且可以轻松链接以编排多个进程。Alauda Container Platform 允许您通过将相关镜像分组到单个 pod 中，轻松实现共置和共管。

这种共置确保容器共享网络命名空间和存储以进行通信。更新也更不具破坏性，因为每个镜像可以更少频率且独立地更新。单进程的信号处理流程也更清晰，无需管理信号路由到派生进程。

在包装脚本中使用 exec

许多镜像使用包装脚本在启动软件进程前做一些设置。如果您的镜像使用此类脚本，脚本应使用 `exec` ，以便脚本进程被您的软件进程替换。如果不使用 `exec` ，容器运行时发送的信号会发送给包装脚本，而非您的软件进程，这不是您想要的。

例如，您有一个包装脚本启动某个服务器进程。您使用 `docker run -i` 启动容器，运行包装脚本，进而启动进程。如果您想用 `CTRL+C` 关闭容器，且包装脚本使用了 `exec` 启动服务器进程， `docker` 会将 SIGINT 发送给服务器进程，一切按预期工作。如果未使用 `exec` ， `docker` 会将 SIGINT 发送给包装脚本进程，您的服务器进程则继续运行。

另外，您的进程在容器中作为 `PID 1` 运行。这意味着如果主进程终止，整个容器会停止，您从 `PID 1` 进程启动的任何子进程也会被取消。

清理临时文件

删除构建过程中创建的所有临时文件，包括使用 `ADD` 命令添加的文件。例如，在执行 `yum install` 操作后运行 `yum clean` 命令。

您可以通过如下方式避免 `yum` 缓存进入镜像层：

```
RUN yum -y install mypackage && yum -y install myotherpackage && yum clean all -y
```

注意，如果写成：

```
RUN yum -y install mypackage
RUN yum -y install myotherpackage && yum clean all -y
```

则第一次 `yum` 调用会在该层留下额外文件，这些文件在后续运行 `yum clean` 时无法被删除。虽然这些额外文件在最终镜像中不可见，但它们存在于底层镜像层中。

当前容器构建过程不允许后续层运行的命令缩小镜像因前层删除文件而占用的空间，但未来可能会改变。这意味着如果您在后续层执行 `rm` 命令，虽然文件被隐藏，但不会减少下载镜像的总体大小。因此，像 `yum clean` 示例一样，最好在创建文件的同一命令中删除它们，避免写入镜像层。

此外，在单个 `RUN` 语句中执行多个命令可以减少镜像层数，提升下载和解压速度。

按正确顺序放置指令

容器构建器从上到下读取 `Dockerfile` 并执行指令。每个成功执行的指令都会创建一个可在下一次构建相同或其他镜像时复用的层。将不常变更的指令放在 `Dockerfile` 顶部非常重要，这样下一次构建相同镜像时缓存不会因上层更改而失效，构建速度更快。

例如，如果您正在编写包含 `ADD` 命令安装您正在迭代的文件和 `RUN` 命令安装软件包的 `Dockerfile`，最好将 `ADD` 命令放在最后：

```
FROM foo
RUN yum -y install mypackage && yum clean all -y
ADD myfile /test/myfile
```

这样每次编辑 `myfile` 并重新运行 `docker build` 时，系统会复用 `yum` 命令的缓存层，仅为 `ADD` 操作生成新层。

如果写成：

```
FROM foo
ADD myfile /test/myfile
RUN yum -y install mypackage && yum clean all -y
```

则每次更改 `myfile` 并重新构建时，`ADD` 操作会使 `RUN` 层缓存失效，导致 `yum` 操作也必须重新执行。

标记重要端口

`EXPOSE` 指令使容器中的端口对主机系统和其他容器可用。虽然可以通过 `docker run` 指定端口暴露，但在 `Dockerfile` 中使用 `EXPOSE` 指令明确声明软件运行所需端口，便于人和软件使用您的镜像：

- 暴露端口会在 `docker ps` 中显示，关联到由您的镜像创建的容器。
- 暴露端口存在于 `docker inspect` 返回的镜像元数据中。
- 连接容器时，暴露端口会被链接。

设置环境变量

使用 `ENV` 指令设置环境变量是良好实践。例如设置项目版本，方便用户无需查看 `Dockerfile` 即可获知版本。另一个例子是声明系统路径供其他进程使用，如 `JAVA_HOME`。

避免默认密码

避免设置默认密码。许多人扩展镜像时忘记删除或更改默认密码，可能导致生产环境用户使用众所周知的密码，带来安全隐患。密码应通过环境变量配置。

如果确实设置了默认密码，确保容器启动时显示适当警告，告知用户默认密码的值及如何更改（例如设置哪个环境变量）。

避免 sshd

最好避免在镜像中运行 `sshd`。您可以使用 `docker exec` 命令访问本地主机上运行的容器，或访问运行在 Alauda Container Platform 集群上的容器。安装和运行 `sshd` 会增加攻击面和安全补丁需求。

使用卷存储持久数据

镜像应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，容器迁移到新节点时，存储会重新挂载。通过卷存储所有持久数据，即使容器重启或迁移，内容也能保留。如果镜像将数据写入容器内任意位置，内容无法保留。

所有需要在容器销毁后仍保留的数据必须写入卷。容器引擎支持容器的 `readonly` 标志，可严格执行不向容器临时存储写入数据的良好实践。现在围绕该能力设计镜像，未来更易利用。

在 `Dockerfile` 中显式定义卷，方便镜像使用者了解运行镜像时必须定义的卷。

有关 Alauda Container Platform 中卷的更多信息，请参阅 Kubernetes 文档↗。

> 注意：
> 即使使用持久卷，您的镜像的每个实例都有自己的卷，实例间文件系统不共享。这意味着卷不能用于在集群中共享状态。

Including metadata in images

定义镜像元数据有助于 Alauda Container Platform 更好地使用您的容器镜像，为使用您的镜像的开发者创造更佳体验。例如，您可以添加元数据提供镜像的有用描述，或建议可能还需要的其他镜像。

本主题仅定义当前用例所需的元数据，未来可能添加更多元数据或用例。

Defining image metadata

您可以在 `Dockerfile` 中使用 `LABEL` 指令定义镜像元数据。标签类似于环境变量，是附加到镜像或容器的键值对。标签不同于环境变量的是，它们对运行中的应用不可见，也可用于快速查找镜像和容器。

有关 `LABEL` 指令的更多信息，请参阅 Docker 文档↗。

标签名称通常带有命名空间。命名空间根据将使用标签的项目设置。对于 Kubernetes，命名空间为 io.k8s。

有关格式的详细信息，请参阅 Docker 自定义元数据文档↗。

Managing images

通过 Alauda Container Platform，您可以根据镜像仓库的位置、这些仓库的认证要求，以及您希望构建和部署的行为，来管理镜像。

Image pull policy

Pod 中的每个容器都有一个容器镜像。在您创建镜像并将其推送到仓库之后，就可以在 Pod 中引用该镜像。

目录

Image pull policy overview

当 Alauda Container Platform 创建容器时，会使用容器的 `imagePullPolicy` 来决定是否在启动容器之前拉取镜像。`imagePullPolicy` 有三种可能的取值：

表格 `imagePullPolicy` 取值：

Value	Description
Always	始终拉取镜像。
IfNotPresent	仅当节点上不存在该镜像时才拉取。
Never	从不拉取镜像。

如果未指定容器的 imagePullPolicy 参数，Alauda Container Platform 会根据镜像标签设置默认值：

1. 如果标签是 latest，Alauda Container Platform 默认将 imagePullPolicy 设置为 Always。
2. 否则，Alauda Container Platform 默认将 imagePullPolicy 设置为 IfNotPresent。

Using image pull secrets

如果您使用的是 Alauda Container Platform 的镜像仓库，那么您的 Pod 服务账户应该已经拥有正确的权限，无需额外操作。

但是，对于其他场景，例如跨 Alauda Container Platform 项目引用镜像或从受保护的仓库拉取镜像，则需要额外的配置步骤。

Allowing pods to reference images from other secured registries

要从其他私有或受保护的仓库拉取受保护的容器镜像，必须使用容器客户端凭据（例如 `Docker`）创建拉取密钥，并将其添加到您的服务账户中。

Docker 使用配置文件存储登录受保护或非受保护仓库的认证信息：

默认情况下，Docker 使用 $HOME/.docker/config.json。

如果您之前登录过受保护或非受保护的仓库，这些文件会存储您的认证信息。

Creating a pull secret

您可以获取拉取密钥，以便从私有容器镜像仓库或仓库中拉取镜像。您可以参考 Pull an Image from a Private Registry↗。

Using a pull secret in a workload

您可以通过以下方法之一使用拉取密钥，允许工作负载从私有仓库拉取镜像：

- 将密钥关联到 `ServiceAccount`，这样所有使用该服务账户的 Pod 会自动应用该密钥。
- 在 Pod 规范中定义 `imagePullSecrets`，这对于 GitOps 或 ArgoCD 等环境非常有用。

您可以通过将密钥添加到服务账户来为 Pod 拉取镜像使用该密钥。请注意，服务账户的名称应与 Pod 使用的服务账户名称匹配。

示例输出：

```
apiVersion: v1
imagePullSecrets:
- name: default-dockercfg-123456
- name: <pull_secret_name>
kind: ServiceAccount
metadata:
  name: default
  namespace: default
secrets:
- name: <pull_secret_name>
```

除了将密钥关联到服务账户之外，您还可以直接在 Pod 或工作负载定义中引用该密钥。这对于 ArgoCD 等 GitOps 工作流非常有用。例如：

示例 Pod 规范：

```
apiVersion: v1
kind: Pod
metadata:
  name: <secure_pod_name>
spec:
  containers:
  - name: <container_name>
    image: your.registry.io/my-private-image
  imagePullSecrets:
  - name: <pull_secret_name>
```

示例 ArgoCD 工作流：

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: <example_workflow>
spec:
  entrypoint: <main_task>
  imagePullSecrets:
  - name: <pull_secret_name>
```

Menu

镜像仓库

介绍

安装

使用指南

Common CLI Command Operations

登录 Registry

为用户添加命名空间权限

为服务账户添加命名空间权限

拉取镜像

推送镜像

Using Alauda Container Platform Registry in Kubernetes Clusters

Registry Access Guidelines

Deploy Sample Application

Cross-Namespace Access

Best Practices

Verification Checklist

Troubleshooting

介绍

构建、存储和管理容器镜像是云原生应用开发流程的核心部分。Alauda Container Platform(ACP) 提供了一个高性能、高可用的内置容器镜像仓库服务，旨在为用户提供安全便捷的镜像存储和管理体验，大大简化平台内的应用开发、持续集成/持续交付（CI/CD）及应用部署流程。

Alauda Container Platform Registry 深度集成于平台架构中，相较于外部独立部署的镜像仓库，提供了更紧密的平台协作、更简化的配置以及更高效的内部访问能力。

目录

原则与命名空间隔离

作为平台的核心组件之一，Alauda Container Platform 内置的镜像仓库以高可用方式运行在集群内部，并利用平台提供的持久化存储能力，确保镜像数据的安全可靠。

其核心设计理念之一是基于 Namespace 的逻辑隔离与管理。在 Registry 中，镜像仓库按命名空间组织。这意味着每个命名空间都可视为该命名空间镜像的独立“区域”，不同命名空间之间的

镜像默认隔离，除非获得明确授权。

认证与授权

Alauda Container Platform Registry 的认证与授权机制深度集成 ACP 平台级认证与授权系统，实现了细粒度到命名空间的访问控制：

认证

用户或自动化流程（例如平台上的 CI/CD 流水线、自动构建任务等）无需为 Registry 维护单独的账户密码。它们通过平台的标准认证机制进行认证（例如使用平台提供的 API Token、集成的企业身份系统等）。通过 CLI 或其他工具访问 Alauda Container Platform Registry 时，通常利用已有的平台登录会话或 ServiceAccount Token 实现透明认证。

授权

授权控制在命名空间级别实现。对 Alauda Container Platform Registry 中镜像仓库的 Pull 或 Push 权限，取决于用户或 ServiceAccount 在对应命名空间内所拥有的平台角色和权限。

- 通常情况下，命名空间的所有者或开发人员角色会自动获得该命名空间下镜像仓库的 Push 和 Pull 权限。
- 其他命名空间的用户或希望跨命名空间拉取镜像的用户，需由目标命名空间的管理员显式授予相应权限（例如通过 RBAC 绑定允许拉取镜像的角色）后，方可访问该命名空间内的镜像。
- 基于命名空间的授权机制确保了命名空间间镜像的隔离，提高安全性，避免未授权访问和修改。

优势

Alauda Container Platform Registry 的核心优势：

- 开箱即用： 快速部署私有镜像仓库，无需复杂配置。
- 访问灵活： 支持集群内及外部访问模式。

- 安全保障： 提供 RBAC 授权及镜像扫描能力。
- 高可用性： 通过复制机制保障服务连续性。
- 生产级别： 在企业环境中验证，具备 SLA 保证。

应用场景

- 轻量级部署： 在低流量环境中实现精简的仓库方案，加速应用交付。
- 边缘计算： 为边缘集群提供专属仓库，实现自治管理。
- 资源优化： 在基础设施利用率不足时，通过集成的 Source to Image (S2I) 方案展示完整工作流能力。

Menu

安装

通过 **YAML** 安装

何时使用此方法？

前提条件

通过 YAML 安装 Alauda Container Platform Registry

更新/卸载 Alauda Container Platform Registry

通过 **Web UI** 安装

何时使用此方法？

前提条件

使用 Web 控制台安装 Alauda Container Platform Registry 集群插件

更新/卸载 Alauda Container Platform Registry

通过 YAML 安装

目录

何时使用此方法？

推荐用于：

- 具备 Kubernetes 专业知识，偏好手动操作的高级用户。
- 需要企业级存储（NAS、AWS S3、Ceph 等）的生产级部署。
- 需要对 TLS、ingress 进行细粒度控制的环境。
- 需要进行完整 **YAML** 自定义以实现高级配置的场景。

前提条件

- 安装Alauda Container Platform Registry 集群插件到目标集群。
- 配置好 kubectl，能够访问目标 **Kubernetes** 集群。
- 拥有创建集群范围资源的集群管理员权限。
- 获取已注册的域名（例如 registry.yourcompany.com）创建域名
- 提供有效的**NAS** 存储（如 NFS、GlusterFS 等）。
- （可选）提供有效的**S3** 存储（如 AWS S3、Ceph 等）。如果没有现成的 S3 存储，可在集群中部署 MinIO（内置 S3）实例 部署 MinIO。

通过 YAML 安装 Alauda Container Platform Registry

操作步骤

1. 创建一个名为 **registry-plugin.yaml** 的 **YAML** 配置文件，内容模板如下：

```yaml
apiVersion: cluster.alauda.io/v1alpha1
kind: ClusterPluginInstance
metadata:
  annotations:
    cpaas.io/display-name: internal-docker-registry
  labels:
    create-by: cluster-transformer
    manage-delete-by: cluster-transformer
    manage-update-by: cluster-transformer
  name: internal-docker-registry
spec:
  config:
    access:
      address: ""
      enabled: false
    fake:
      replicas: 2
    global:
      expose: false
      isIPv6: false
      replicas: 2
      oidc:
        ldapID: ""
      resources:
        limits:
          cpu: 500m
          memory: 512Mi
        requests:
          cpu: 250m
          memory: 256Mi
    ingress:
      enabled: true
      hosts:
        - name: <YOUR-DOMAIN>   # [REQUIRED] 自定义域名
          tlsCert: <NAMESPACE>/<TLS-SECRET>  # [REQUIRED] 命名空间/Secret 名称
      ingressClassName: "<INGRESS-CLASS-NAME>"  # [REQUIRED] IngressClassName
      insecure: false
    persistence:
      accessMode: ReadWriteMany
      nodes: ""
      path: <YOUR-HOSTPATH>  # [REQUIRED] LocalVolume 的本地路径
      size: <STORAGE-SIZE>  # [REQUIRED] 存储大小（例如 10Gi）
      storageClass: <STORAGE-CLASS-NAME>  # [REQUIRED] StorageClass 名称
```

```
      type: StorageClass
    s3storage:
      bucket: <S3-BUCKET-NAME>            # [REQUIRED] S3 桶名称
      enabled: false                       # 本地存储时设置为 false
      env:
        REGISTRY_STORAGE_S3_SKIPVERIFY: false  # 自签名证书时设置为 true
      region: <S3-REGION>                      # S3 区域
      regionEndpoint: <S3-ENDPOINT>  # S3 端点
      secretName: <S3-CREDENTIALS-SECRET>            # S3 凭据 Secret
    service:
      nodePort: ""
      type: ClusterIP
  pluginName: internal-docker-registry
```

2. 根据您的环境自定义以下字段：

```
spec:
  config:
    global:
      oidc:
        ldapID: "<LDAP-ID>"                  # LDAP ID
    ingress:
      hosts:
        - name: "<YOUR-DOMAIN>"               # 例如 registry.your-company.com
          tlsCert: "<NAMESPACE>/<TLS-SECRET>"   # 例如 cpaas-system/tls-secret
      ingressClassName: "<INGRESS-CLASS-NAME>"  # 例如 cluster-alb-1
    persistence:
      size: "<STORAGE-SIZE>"                   # 例如 10Gi
      storageClass: "<STORAGE-CLASS-NAME>"      # 例如 cpaas-system-storage
    s3storage:
      bucket: "<S3-BUCKET-NAME>"               # 例如 prod-registry
      region: "<S3-REGION>"                    # 例如 us-west-1
      regionEndpoint: "<S3-ENDPOINT>"          # 例如 https://s3.amazonaws.com
      secretName: "<S3-CREDENTIALS-SECRET>"     # 包含
AWS_ACCESS_KEY_ID/AWS_SECRET_ACCESS_KEY 的 Secret
      env:
        REGISTRY_STORAGE_S3_SKIPVERIFY: "true"  # 自签名证书时设置为 "true"
```

3. 如何创建 **S3** 凭据的 **Secret**：

```
kubectl create secret generic <S3-CREDENTIALS-SECRET> \
  --from-literal=access-key-id=<YOUR-S3-ACCESS-KEY-ID> \
  --from-literal=secret-access-key=<YOUR-S3-SECRET-ACCESS-KEY> \
  -n cpaas-system
```

将 `<S3-CREDENTIALS-SECRET>` 替换为您的 S3 凭据 Secret 名称。

4. 将配置应用到集群：

```
kubectl apply -f registry-plugin.yaml
```

配置参考

必填字段

参数	说明	示例值
`spec.config.global.oidc.ldapID`	OIDC 认证的 LDAP ID	`ldap-test`
`spec.config.ingress.hosts[0].name`	Registry 访问的自定义域名	`registry.yourcompany.com`
`spec.config.ingress.hosts[0].tlsCert`	TLS 证书 Secret 引用（命名空间/Secret 名称）	`cpaas-system/registry-tls`
`spec.config.ingress.ingressClassName`	Registry 使用的 Ingress 类名	`cluster-alb-1`
`spec.config.persistence.size`	Registry 存储大小	`10Gi`
`spec.config.persistence.storageClass`	Registry 使用的 StorageClass 名称	`nfs-storage-sc`

参数	说明	示例值
`spec.config.s3storage.bucket`	镜像存储使用的 S3 桶名称	`prod-image-store`
`spec.config.s3storage.region`	S3 存储的 AWS 区域	`us-west-1`
`spec.config.s3storage.regionEndpoint`	S3 服务端点 URL	`https://s3.amazonaws.com`
`spec.config.s3storage.secretName`	包含 S3 凭据的 Secret	`s3-access-keys`

验证

1. 查看插件状态：

```
kubectl get clusterplugininstances internal-docker-registry -o yaml
```

2. 验证 registry Pod：

```
kubectl get pods -n cpaas-system -l app=internal-docker-registry
```

更新/卸载 Alauda Container Platform Registry

更新

在 global 集群上执行以下命令，根据上述参数说明更新资源中的值以完成更新：

```
# <CLUSTER-NAME> 是插件安装所在的集群名称
kubectl edit -n cpaas-system \
  $(kubectl get moduleinfo -n cpaas-system -l cpaas.io/cluster-name=<CLUSTER-
NAME>,cpaas.io/module-name=internal-docker-registry -o name)
```

卸载

在 global 集群上执行以下命令：

```
# <CLUSTER-NAME> 是插件安装所在的集群名称
kubectl get moduleinfo -n cpaas-system -l cpaas.io/cluster-name=<CLUSTER-
NAME>,cpaas.io/module-name=internal-docker-registry -o name | xargs kubectl delete -n
cpaas-system
```

通过 **Web UI** 安装

目录

何时使用此方法？

推荐使用场景：

- 首次使用者，偏好引导式、可视化界面操作。
- 在非生产环境中进行快速概念验证的部署。
- 具备有限 **Kubernetes** 经验的团队，寻求简化的部署流程。
- 需要进行最小化定制的场景（例如，默认存储配置）。
- 基础网络配置，无特定 ingress 规则需求。
- 针对高可用性的 **StorageClass** 配置。

不推荐使用场景：

- 生产环境中需要高级存储（如 S3 存储）配置。

- 需要特定 ingress 规则的网络配置。

前提条件

- 通过 Cluster Plugin 机制，将 **Alauda Container Platform Registry** 集群插件安装到目标集群。

使用 **Web** 控制台安装 **Alauda Container Platform Registry** 集群插件

操作步骤

1. 登录并进入 管理员 页面。
2. 点击 **Marketplace** > **Cluster Plugins**，进入 **Cluster Plugins** 列表页面。
3. 找到 **Alauda Container Platform Registry** 集群插件，点击 **Install**，进入安装页面。
4. 按照以下参数说明配置参数，点击 **Install** 完成部署。

参数说明如下：

参数	说明
Expose Service	启用后，管理员可以通过访问地址对镜像仓库进行外部管理。此操作存在较大安全风险，需谨慎启用。
Enable IPv6	当集群使用 IPv6 单栈网络时，启用此选项。
NodePort	在启用 Expose Service 时，配置 NodePort 以允许通过该端口外部访问 Registry。
Storage Type	选择存储类型。支持类型包括 LocalVolume 和 StorageClass。
Nodes	选择运行 Registry 服务的节点，用于镜像存储和分发。（仅当存储类型为 LocalVolume 时可用）

参数	说明
StorageClass	选择 StorageClass。当副本数超过 1 时，需选择具备 RWX（ReadWriteMany）能力的存储（如文件存储），以保证高可用性。（仅当存储类型为 StorageClass 时可用）
Storage Size	分配给 Registry 的存储容量（单位：Gi）。
Replicas	配置 Registry Pod 的副本数： • **LocalVolume**：默认 1（固定） • **StorageClass**：默认 3（可调整）
Resource Requirements	定义 Registry Pod 的 CPU 和内存资源请求及限制。

验证

1. 进入 **Marketplace** > **Cluster Plugins**，确认插件状态显示为 **Installed**。
2. 点击插件名称查看详情。
3. 复制 **Registry Address**，使用 Docker 客户端进行镜像的推送/拉取操作。

更新/卸载 **Alauda Container Platform Registry**

您可以在列表页面或详情页面对 **Alauda Container Platform Registry** 插件进行更新或卸载操作。

Menu

使用指南

Common CLI Command Operations

登录 Registry

为用户添加命名空间权限

为服务账户添加命名空间权限

拉取镜像

推送镜像

Using Alauda Container Platform Registry in Kubernetes Clusters

Registry Access Guidelines

Deploy Sample Application

Cross-Namespace Access

Best Practices

Verification Checklist

Troubleshooting

Common CLI Command Operations

Alauda Container Platform 提供了命令行工具，供用户与 Alauda Container Platform Registry 进行交互。以下是一些常见操作和命令示例：

假设集群的 Alauda Container Platform Registry 服务地址为 registry.cluster.local，且您当前操作的命名空间为 my-ns。

> 请联系技术服务获取 kubectl-acp 插件，并确保其已正确安装在您的环境中。

目录

登录 Registry

通过登录 ACP 来登录集群的 Registry。

```
kubectl acp login <ACP-endpoint>
```

为用户添加命名空间权限

为用户添加命名空间拉取权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-puller --user=
<username> -n <namespace>
```

为用户添加命名空间推送权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-pusher --user=
<username> -n <namespace>
```

为服务账户添加命名空间权限

为服务账户添加命名空间拉取权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-puller --
serviceaccount=<namespace>:<serviceaccount-name> -n <namespace>
```

为服务账户添加命名空间推送权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-pusher --
serviceaccount=<namespace>:<serviceaccount-name> -n <namespace>
```

拉取镜像

从 Registry 拉取镜像到集群内（例如用于 Pod 部署）。

```
# 从当前命名空间（my-ns）的 Registry 拉取名为 my-app，标签为 latest 的镜像
kubectl acp pull registry.cluster.local/my-ns/my-app:latest

# 从其他命名空间（例如 shared-ns）拉取镜像（需要有 shared-ns 命名空间的拉取权限）
kubectl acp pull registry.cluster.local/shared-ns/base-image:latest
```

该命令会验证您在目标命名空间的身份和拉取权限，然后从 Registry 拉取镜像。

推送镜像

将本地构建的镜像或从其他地方拉取的镜像推送到 Registry 的指定命名空间。

您需要先使用标准容器命令行工具（如 docker）将本地镜像打标签（tag）为目标 Registry 的地址和命名空间格式。

```
# 给镜像打上目标地址标签：
docker tag my-app:latest registry.cluster.local/my-ns/my-app:v1

# 使用 kubectl 命令将其推送到当前命名空间（my-ns）的 Registry
kubectl acp push registry.cluster.local/my-ns/my-app:v1
```

将远程镜像仓库中的镜像推送到 Alauda Container Platform Registry 的指定命名空间。

```
# 如果您的远程镜像仓库中有镜像 remote.registry.io/demo/my-app:latest
# 使用 kubectl 命令将其推送到 Registry 的命名空间（my-ns）
kubectl acp push remote.registry.io/demo/my-app:latest registry.cluster.local/my-ns/my-
app:latest
```

该命令会验证您在 my-ns 命名空间内的身份和推送权限，然后将本地打标签的镜像上传到 Registry。

Using Alauda Container Platform Registry in Kubernetes Clusters

Alauda Container Platform (ACP) Registry 为 Kubernetes 工作负载提供安全的容器镜像管理。

目录

Registry Access Guidelines

- 推荐使用内部地址：对于存储在集群注册表中的镜像，部署时应优先使用集群内部服务地址 `internal-docker-registry.cpaas-system.svc`，以确保最佳的网络性能并避免不必要的外部路由。
- 外部地址使用场景：外部 ingress 域名（例如 `registry.cluster.local`）主要用于：
  - 集群外部的镜像推送/拉取（如开发人员机器、CI/CD 系统）
  - 需要访问注册表的集群外部操作

Deploy Sample Application

1. 在 `my-ns` 命名空间中创建名为 `my-app` 的应用。
2. 将应用镜像存储在注册表地址 `internal-docker-registry.cpaas-system.svc/my-ns/my-app:v1`。
3. 每个命名空间的 默认 ServiceAccount 会自动配置 imagePullSecret，用于访问 `internal-docker-registry.cpaas-system.svc` 上的镜像。

示例 Deployment：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: my-ns
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: main-container
        image: internal-docker-registry.cpaas-system.svc/my-ns/my-app:v1
        ports:
        - containerPort: 8080
```

Cross-Namespace Access

为了允许 `my-ns` 的用户从 `shared-ns` 拉取镜像，`shared-ns` 的管理员可以创建角色绑定以授予必要权限。

示例角色绑定

```
# 访问共享命名空间的镜像（需要权限）
kubectl create rolebinding cross-ns-pull \
  --clusterrole=system:image-puller \
  --serviceaccount=my-ns:default \
  -n shared-ns
```

Best Practices

- 注册表使用：部署时始终使用 `internal-docker-registry.cpaas-system.svc`，确保安全性和性能。
- 命名空间隔离：利用命名空间隔离实现更好的安全性和镜像管理。
  - 使用基于命名空间的镜像路径：`internal-docker-registry.cpaas-system.svc/<namespace>/<image>:<tag>`。
- 访问控制：通过角色绑定管理跨命名空间的用户和服务账户访问权限。

Verification Checklist

1. 验证 `my-ns` 中默认 ServiceAccount 的镜像访问权限：

   ```
   kubectl auth can-i get images.registry.alauda.io --namespace my-ns --
   as=system:serviceaccount:my-ns:default
   ```

2. 验证 `my-ns` 中某用户的镜像访问权限：

   ```
   kubectl auth can-i get images.registry.alauda.io --namespace my-ns --as=<username>
   ```

Troubleshooting

- 镜像拉取错误：检查 Pod 规范中的 imagePullSecrets，确保配置正确。

- 权限拒绝：确认用户或 ServiceAccount 在目标命名空间中拥有必要的角色绑定。
- 网络问题：核实网络策略和服务配置，确保能连接到内部注册表。
- **DNS** 解析失败：检查节点上的 `/etc/hosts` 文件内容，确保 `internal-docker-registry.cpaas-system.svc` 的 DNS 解析配置正确。
  - 验证节点的 /etc/hosts 配置，确保 `internal-docker-registry.cpaas-system.svc` 的 DNS 解析正确
  - 注册表服务映射示例（internal-docker-registry 服务的 ClusterIP）：

    ```
    # /etc/hosts
    127.0.0.1   localhost localhost.localdomain
    10.4.216.11 internal-docker-registry.cpaas-system internal-docker-registry.cpaas-system.svc internal-docker-registry.cpaas-system.svc.cluster.local # cpaas-generated-node-resolver
    ```

  - 如何获取 `internal-docker-registry` 当前 **ClusterIP**：

    ```
    kubectl get svc -n cpaas-system internal-docker-registry -o jsonpath='{.spec.clusterIP}'
    ```

Menu

源代码到镜像

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

发版日志

Alauda Container Platform Builds 发布说明

支持的版本

v1.1 版本说明

生命周期策略

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

功能指南

Managing applications created from Code

主要功能

优势

前提条件

操作步骤

相关操作

How To

通过代码创建应用

前提条件

操作步骤

Menu

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

发版日志

Alauda Container Platform Builds 发布说明

支持的版本

v1.1 版本说明

生命周期策略

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) 功能与自动化流水线。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

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 支持从代码到应用的全栈云原生工作流，实现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，确保与完整云原生平台的无缝集成。

- 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、部署和运维管理的自动化。
- 深度集成：与您的 Container Platform 产品无缝集成，提供流畅的开发体验。
- 高扩展性：支持自定义插件和扩展，以满足您的特定需求。

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

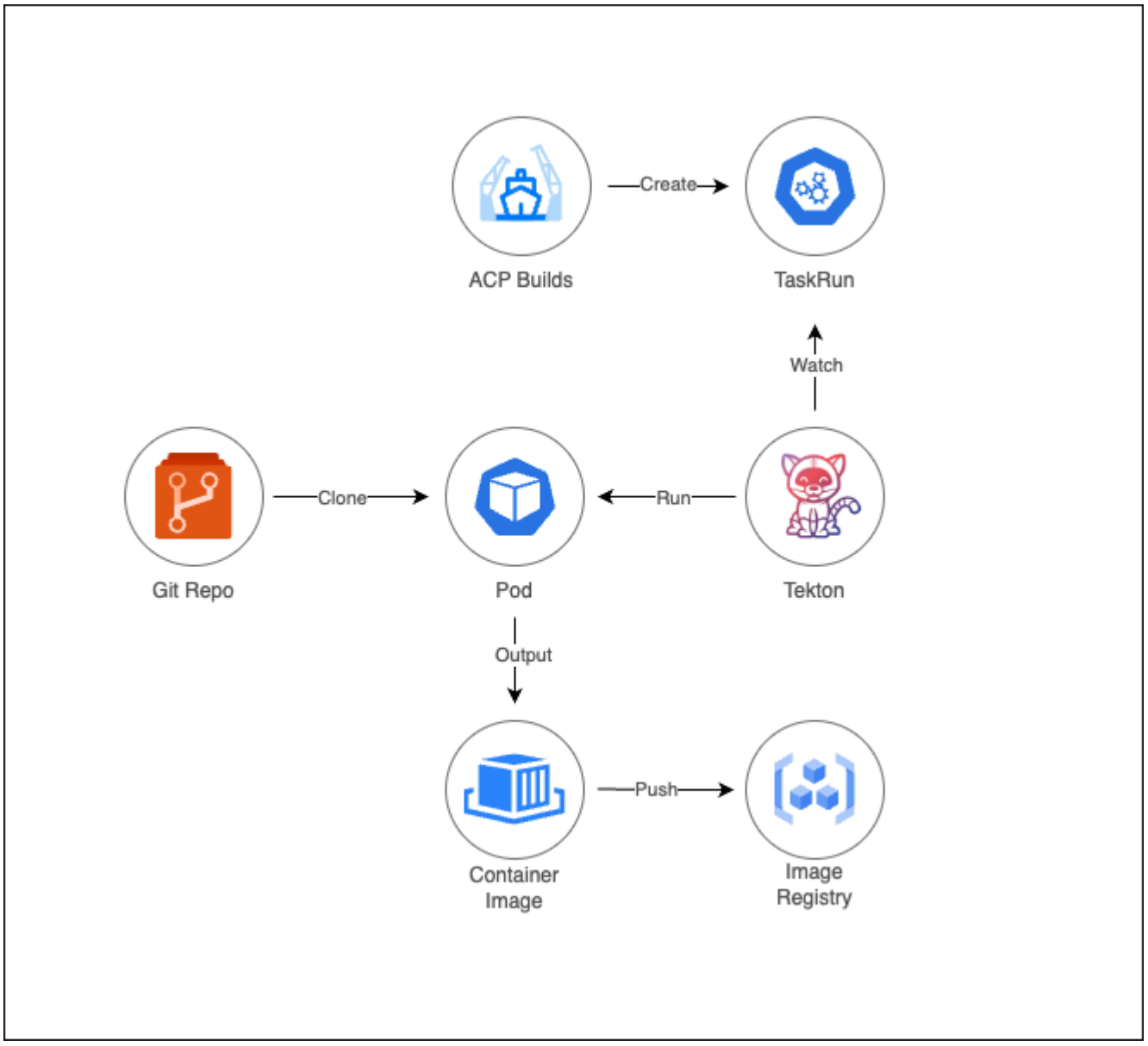
> 当前版本仅支持 Java、Go、Python 和 Node.js 语言。

WARNING

前提条件：Alauda DevOps Pipelines operator↗ 现已在集群 OperatorHub 中提供。

Menu

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，并将其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 发布说明

Alauda Container Platform Builds operator 的发布说明描述了新功能和增强功能、已废弃的功能以及已知问题。

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 生命周期政策 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 版本说明

v1.1.0

1. 安全漏洞修复。
2. 独立发布。

生命周期策略

版本生命周期时间表

以下是 Alauda Container Platform Builds Operator 已发布版本的生命周期时间表：

版本	发布日期	终止支持日期
v1.1.0	2025-08-15	2027-08-15

Menu

安装

Installing Alauda Container Platform Builds

Prerequisites

Procedure

Installing Alauda Container Platform Builds

目录

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 **Alauda Container Platform Builds** 最新版本包。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** 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 共享于集群中所有命名空间，用于实例的创建和管理，资源占用较低。
Namespace	`Recommended`：建议使用 **shipyard-operator** 命名空间；如果不存在将自动创建。
Upgrade Strategy	请选择 `Manual`。 • `Manual`：当 OperatorHub 有新版本时 • 不会自动执行 **Upgrade** 操作。

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，然后进入 **Container Platform** > **Applications** > **Applications**，点击 **Create**。
- 应该能看到 **Create from Code** 的入口。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。

Menu

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

> 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，请先迁移 **Alauda DevOps Tekton v3 到 Alauda DevOps Pipelines**↗。

1. 登录并进入 管理员 页面。
2. 点击 **Marketplace** > **OperatorHub**。
3. 在导航栏选择安装了该 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。

Menu

功能指南

Managing applications created from Code

主要功能

优势

前提条件

操作步骤

相关操作

Managing applications created from Code

目录

主要功能

- 输入代码仓库 URL 触发 S2I 流程，将源代码转换为镜像并发布为应用。
- 当源代码更新时，通过可视化界面发起 **Rebuild** 操作，一键更新应用版本。

优势

- 简化了从代码创建和升级应用的流程。
- 降低开发人员门槛，无需了解容器化细节。
- 提供可视化构建流程和运维管理，便于定位问题、分析和排查。

前提条件

- 已完成安装 Alauda Container Platform Builds。
- 需要访问镜像仓库；若无，请联系管理员进行Alauda Container Platform Registry 安装。

操作步骤

1. 在 **Container Platform** 中，进入 **Application** > **Application**。
2. 点击 **Create**。
3. 选择 **Create from Code**。
4. 参考以下参数说明完成配置。

区域	参数	说明
代码仓库	类型	• 平台集成：选择已与平台集成且分配给当前项目的代码仓库；平台支持 GitLab、GitHub 和 Bitbucket。 • 输入：使用未与平台集成的代码仓库 URL。
	集成项目名称	管理员分配或关联给当前项目的集成工具项目名称。
	仓库地址	选择或输入存储源代码的代码仓库地址。
	版本标识	支持基于代码仓库中的分支、标签或提交创建应用。其中： • 当版本标识为分支时，仅支持使用所选分支下的最新提交创建应用。

		• 当版本标识为标签或提交时，默认选择代码仓库中的最新标签或提交，也可根据需要选择其他版本。
	上下文目录	可选的源代码目录，作为构建的上下文目录。
	Secret	使用输入类型代码仓库时，可根据需要添加认证 Secret。
	Builder 镜像	• 包含特定编程语言运行环境、依赖库和 S2I 脚本的镜像，主要用于将源代码转换为可运行的应用镜像。 • 支持的 Builder 镜像包括：Golang、Java、Node.js 和 Python。
	版本	选择与源代码兼容的运行环境版本，确保应用顺利运行。
构建	构建类型	目前仅支持通过 **Build** 方式构建应用镜像。该方式简化并自动化复杂的镜像构建过程，使开发人员专注于代码开发。整体流程如下： 1. 安装 Alauda Container Platform Builds 并创建 Shipyard 实例后，系统自动生成集群级资源，如 ClusterBuildStrategy，定义标准化构建流程，包括详细构建步骤和所需构建参数，从而支持 Source-to-Image (S2I) 构建。详细信息请参见：安装 Alauda Container Platform Builds 2. 根据上述策略及表单填写信息创建 Build 类型资源，指定构建策略、构建参数、代码仓库、输出镜像仓库等相关信息。 3. 创建 BuildRun 类型资源以启动具体构建实例，协调整个构建流程。 4. BuildRun 创建完成后，系统自动生成对应的 TaskRun 资源实例。该 TaskRun 实例触发 Tekton pipeline 构建并创建 Pod 执

		行构建过程。Pod 负责实际构建工作，包括：拉取代码仓库中的源代码。 调用指定的 Builder 镜像。 执行构建流程。
	镜像 **URL**	构建完成后，指定应用的目标镜像仓库地址。
应用	-	根据需要填写应用配置，具体详情请参考基于镜像创建应用文档中的参数说明。
网络	-	• 目标端口：容器内应用实际监听的端口。启用外部访问时，所有匹配流量将转发至该端口以提供外部服务。 • 其他参数：请参考创建 Ingress文档中的参数说明。
标签注解	-	根据需要填写相关标签和注解。

5. 填写完参数后，点击 **Create**。

6. 可在 **Details** 页面查看对应部署信息。

相关操作

构建

应用创建完成后，可在详情页面查看对应信息。

参数	说明
Build	点击链接查看具体的构建（Build）和构建任务（BuildRun）资源信息及 YAML。
Start Build	构建失败或源代码变更时，可点击此按钮重新执行构建任务。

Menu

How To

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

> 利用 Alauda Container Platform Builds 安装的强大功能，实现从 Java 源代码到创建应用的整个流程，最终使应用能够高效地以容器化方式在 Kubernetes 上运行。

目录

前提条件

在使用此功能之前，请确保：

- 已完成安装 Alauda Container Platform Builds
- 平台上有可访问的镜像仓库。如果没有，请联系管理员进行安装 ACP Registry

操作步骤

1. 在 **Container Platform** 中，点击 **Applications** > **Applications**。
2. 点击 **Create**。
3. 选择 **Create from Code**。

4. 根据以下参数完成配置：

参数	推荐配置
代码仓库	类型：`Input` 仓库 **URL**：`https://github.com/alauda/spring-boot-hello-world`
构建方式	`Build`
镜像仓库	联系管理员。
应用	**Application**：`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. 填写完参数后，点击 **Create**。

6. 可在 **Details** 页面查看对应应用的状态。

Menu

节点隔离策略

节点隔离策略提供了一种项目级别的节点隔离策略，使项目能够独占使用集群节点。

引言

架构

概念

功能指南

创建节点隔离策略

创建节点隔离策略

删除节点隔离策略

权限说明

权限说明

引言

节点隔离策略提供了一种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

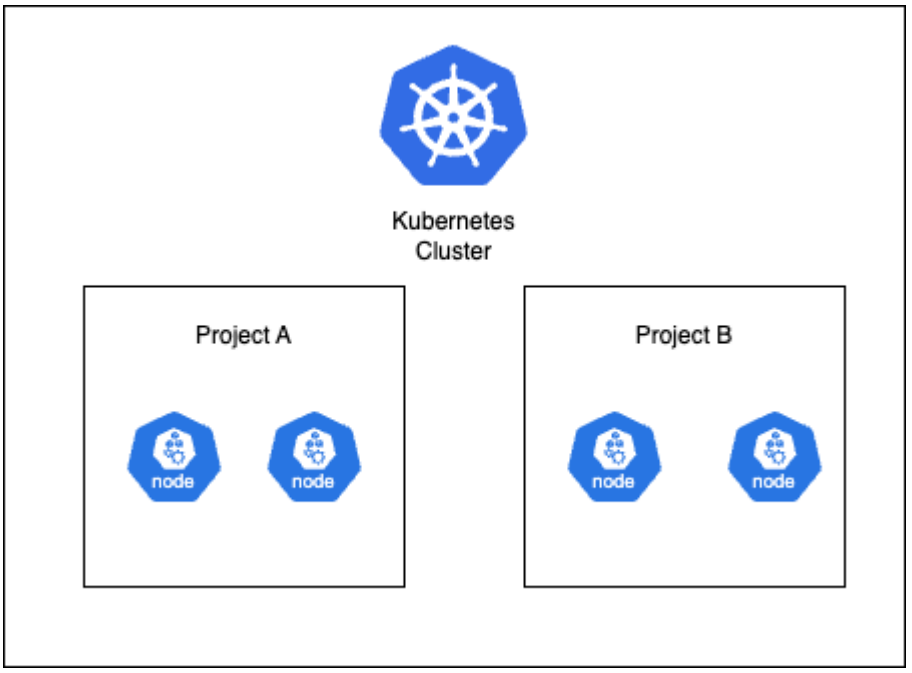
优势

可以便捷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竞争。

应用场景

节点隔离策略适用于需要增强项目之间资源隔离的场景，以及希望防止其他项目的组件占用节点，从而导致资源限制或无法满足性能要求的情况。

架构



节点隔离策略是基于容器平台集群核心组件实现的，通过在每个工作负载集群上分配节点，提供项目之间节点隔离的能力。当在项目中创建容器时，它们会被强制调度到分配给该特定项目的节点上。

Menu

概念

核心概念

节点隔离

核心概念

目录

节点隔离

节点隔离是指在集群中隔离节点，以防止来自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。

Menu

功能指南

创建节点隔离策略

为当前集群创建节点隔离策略，允许指定项目对集群内分组资源的节点进行独占访问，从而限制项目下 Pods 可运行的节点，实现项目间的物理资源隔离。

目录

创建节点隔离策略

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. 根据以下说明配置相关参数。

参数	描述
项目独占性	是否启用或禁用策略中配置的项目隔离政策包含的节点的开关；点击可切换开启或关闭，默认开启。 当开关开启时，只有政策中指定项目下的 Pods 可以在政策包含的节点上运行；当关闭时，当前集群中其他项目下的 Pods 也可以在政策包含的节点上运行，除了指定项目。

参数	描述
项目	配置为使用政策中节点的项目。 点击 项目 下拉选择框，勾选项目名称前的复选框以选择多个项目。 注意： 一个项目只能设置一个节点隔离政策；如果项目已经被分配了节点隔离政策，则无法再次选择； 支持在下拉选择框中输入关键字以过滤和选择项目。
节点	分配给项目使用的计算节点的 IP 地址。 点击 节点 下拉选择框，勾选节点名称前的复选框以选择多个节点。 注意： 一个节点只能属于一个隔离政策；如果节点已经属于另一个隔离政策，则无法再次选择； 支持在下拉选择框中输入关键字以过滤和选择节点。

4. 点击 创建。

   注意：

   - 策略创建后，项目中现有的 Pods 如果不符合当前政策，将在重建后被调度到当前政策包含的节点上；
   - 当 项目独占性 开启时，当前节点上现有的 Pods 不会被自动驱逐；如果需要驱逐，需手动调度。

删除节点隔离策略

注意：删除节点隔离政策后，项目将不再受到限制，无法仅在特定节点上运行，节点将不再被项目独占使用。

1. 在左侧导航栏中，点击 安全 > 节点隔离策略。

2. 找到节点隔离政策，点击⋮> 删除。

Menu

权限说明

功能	操作	平台管理员	平台审计人员	项目管理员	命名空间管理员	开发人员
节点隔离策略 `acp-nodegroups`	查看	✓	✓	✓	✓	✓
	创建	✓	×	×	×	×
	更新	✓	×	×	×	×
	删除	✓	×	×	×	×

常见问题

目录

为什么导入命名空间时不应存在多个 **ResourceQuota**？

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` 中的最小值。

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 导致配额计算复杂且容易出错。

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一个 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。