开发者

总览

介绍

- 优势
- 应用场景
- 跨领域的云原生原则

核心概念

功能总览

- 应用构建
- 命名空间管理
- 应用可观测性
- 源到镜像
- 注册中心
- 节点隔离策略
- OAM 应用

快速开始

快速创建一个应用

- 介绍
- 注意事项
- 前提条件
- 工作流程概述
- 操作步骤

构建应用

概述

命名空间管理

应用生命周期管理

Kubernetes 工作负载管理

核心概念

使用指南

功能指南

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

介绍

Source to Image 概念

核心功能

核心优势

应用场景

使用限制

安装

功能指南

架构

How To

节点隔离策略

常见问题

总览

介绍

介绍

优势

应用场景

跨领域的云原生原则

核心概念

资源单元描述

应用类型

工作负载类型

功能总览

功能总览

应用构建

命名空间管理

应用可观测性

源到镜像

注册中心

节点隔离策略

OAM 应用

介绍

开发者视图模块为开发人员提供了云原生应用编排和操作能力。它提供了一个统一的界面，用于从多个来源组合应用，同时集成了内置的可观察性工具以支持生产操作。

目录

优势

> 开发者视图模块提供以下关键优势：

1. 统一的应用编排

   - 镜像：从公共/私有注册表部署镜像
   - YAML：直接使用带有模式验证的 Kubernetes 资源声明
   - 源代码到镜像 (S2I)：直接从源代码构建容器化应用
   - Helm Charts：从策划的应用目录中部署打包应用
   - 使用多种方法实现与 GitOps 对齐的应用组合

2. 全面的生命周期管理

> 实现对工作负载和命名空间的声明式管理：

- 渐进交付：通过 ServiceMesh 实现金丝雀/蓝绿部署
- 资源治理：
  - 通过 RBAC 策略进行命名空间配置
  - 通过 HPA/VPA 进行资源分配策略
  - 与集群自动扩展器集成的动态扩展
- 工作流自动化：与 Tekton 的 CI/CD 管道集成

3. 企业级命名空间控制

> 实现多租户命名空间管理：

- 完整的生命周期管理
- 资源保证：
  - 资源配额和限制范围配置
  - 可配置的 CPU/内存 超售比

4. 全栈可观察性

> 集成的监控栈包括：

- 事件关联：Kubernetes 事件和审计日志集成
- 日志分析：日志聚合
- 指标监控面板：监控和自定义告警规则

应用场景

> 开发者模块的主要应用场景包括：

- 多云部署

组织将工作负载分布在多个云服务提供商（AWS、Azure、GCP）之间，以避免供应商锁定、优化成本并确保弹性。云原生应用交付实现一致的部署流水线，抽象出特定于提供商的实现。

- 混合云环境

企业在公共云资源的同时维护本地基础设施。云原生交付提供统一的应用部署方法，跨混合环境管理异构基础设施的复杂性。

- 边缘计算集成

随着边缘计算的重要性日益增加，应用必须在集中式云、边缘设备和区域边缘节点上运行。云原生交付将部署能力扩展到这些分布式边缘环境。

- 开发到生产管道

云原生方法支持应用从开发到测试/预发布再到生产的无缝推广，保持配置一致性，同时满足环境特定的要求。

- 全球多地区部署

对于全球分布的应用，云原生交付确保跨地理区域的一致部署，解决延迟优化和数据本地化合规性问题。

- 灾难恢复和工作负载连续性

云原生交付促进灾难恢复环境的配置，镜像生产系统，实现快速故障转移并确保不间断操作。

跨领域的云原生原则

> 这些场景利用了核心的云原生原则：

- 容器化
- 基础设施即代码 (IaC)
- 声明式配置
- 不变基础设施
- GitOps 工作流

这些原则确保了跨异构计算环境的一致性、可靠性和自动化。

核心概念

资源单元描述

应用类型

工作负载类型

资源单元描述

- CPU：可选单位有：核、m（毫核）。其中 1 核 = 1000 m。
- 内存：可选单位有：Mi（1 MiB = 2^20 字节）、Gi（1 GiB = 2^30 字节）。其中 1 Gi = 1024 Mi。
- 虚拟 GPU（可选）：此参数仅在集群中有 GPU 资源时生效。虚拟 GPU 核心数量；100 个虚拟核心等于 1 个物理 GPU 核心。它支持正整数。
- 视频内存（可选）：此参数仅在集群中有 GPU 资源时生效。虚拟 GPU 视频内存；1 单位的视频内存等于 256 Mi。它支持正整数。

应用类型

在平台的 容器平台 **>** 应用管理 中，可以创建以下类型的应用：

- 应用：由一个或多个关联的计算组件（工作负载）、内部路由（服务）及其他原生 Kubernetes 资源组成的完整业务应用。它支持通过 UI 编辑、YAML 编排和模板进行创建，并可以在开发、测试或生产环境中运行。可以通过以下方式创建不同类型的原生应用：
  - 从镜像创建：使用现有的容器镜像快速创建应用。
  - 从目录创建：使用 Helm Chart 包创建应用。
  - 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。
- **Operator** 支持的应用：基于应用组件（Operator 支持），可以快速部署组件应用，并利用 Operator 的能力自动化整个应用的生命周期管理。
- **OAM** 应用：用于定义云原生应用的模型。与容器或 Kubernetes 编排逻辑相比，OAM 更加关注“应用”本身。基于 OAM，应用的通用能力被封装成高层接口供使用，贯穿于应用部署、开发和运营的整个过程中。

工作负载类型

除了创建原生应用和组件应用之外，工作负载还可以在 容器平台 > 计算组件 中直接创建：

- 部署（**Deployment**）：用于部署无状态应用最常用的工作负载控制器。它可以确保指定数量的 Pod 副本在集群中运行，支持滚动更新和回滚，适合用于无状态应用场景，如网页服务和 API 服务。
- 守护进程集（**DaemonSet**）：确保集群中的每个节点（或特定节点）运行一个 Pod 副本。当节点加入集群时，Pod 将自动创建；当节点从集群中移除时，这些 Pods 也会被回收。适合需要在每个节点上运行的场景，如日志记录、监控等。
- 有状态集（**StatefulSet**）：用于管理有状态应用的工作负载控制器。它为每个 Pod 维护一个固定的身份，提供稳定的存储和网络身份，即使 Pod 被重新调度，这些身份也不会改变。适合用于有状态应用，如数据库和分布式缓存。
- 任务（**Job**）：用于运行一次性任务的工作负载。一个 Job 创建一个或多个 Pods，并确保指定数量的 Pods 成功完成任务后才终止。适合用于批处理、数据迁移和其他一次性任务场景。
- 定时任务（**CronJob**）：用于管理基于时间调度运行的 Job。您可以设置任务执行的时间表达式，系统将在设定的时间自动创建并运行 Job。适合用于周期性任务，如数据备份、报告生成和定期清理。

除了通过平台的表单页面创建上述计算组件外，平台还支持通过 CLI 工具创建 Pods 和容器：

- **Pod**：Kubernetes 中最小的可部署单元，Pod 可以包含一个或多个共享存储、网络和配置声明的容器。Pods 通常由控制器（如部署）进行管理。
- 容器（**Container**）：一种标准软件单元，打包了代码和所有依赖项，使得应用程序能够在不同的计算环境中快速且可靠地运行。容器在 Pods 中运行，并共享 Pod 的资源。

TIP

将父模块下的所有功能模块进行罗列和简单介绍，方便用户快速了解该模块下的所有功能。

可以访问 功能总览示例 查看对应文档的示例。

功能总览

目录

应用构建

- 创建应用

  支持多种方式创建应用，包括镜像、yaml、代码和目录。

- 应用操作

使用应用来编排和操作工作负载及其相关资源。

- 工作负载管理

  管理工作负载的生命周期。

命名空间管理

- 命名空间生命周期管理

  管理命名空间的生命周期。

- 资源配额和限制管理

  管理命名空间的资源配额和限制。

- 命名空间资源超分配

  允许对命名空间的资源进行超分配。

应用可观测性

- 日志

  查询应用的历史日志或实时日志。

- 事件

  查询从应用中收集的事件。

- 监控

  监控应用状态，并在出现异常时触发警报。

源到镜像

- 从源构建镜像

从 Git 仓库的源代码构建镜像并将其推送到镜像仓库。

注册中心

- 开箱即用的注册中心服务器

  轻松部署可用于该平台的注册中心服务器。

节点隔离策略

- 节点隔离

  支持项目级别的节点隔离，以避免项目间的资源争用。

OAM 应用

- 高效的运维

  通过 OAM 应用，应用运维人员可以专注于业务逻辑，从应用的角度而不是平台的角度来管理应用，减少应用运维的门槛。平台运维人员可以统一处理平台插件、运维插件和其他配置，从而提高运维效率。

- 可移植性

  OAM 应用模型包括与应用运维、服务治理等相关的配置。与通过 Operators、Charts 和其他方法部署的应用相比，OAM 应用可以通过 YAML 进行重复部署，使跨环境迁移变得更加轻松。即使没有 Kubernetes 和特定厂商，OAM 应用仍然可以在各种平台上正常运行。

- 可扩展性

  平台上预装的几种类型的组件可以满足大多数应用开发需求：网络服务、有状态应用和原生 Kubernetes 资源。此外，平台还提供扩展组件和特征的能力，使开发人员能够轻松使用自定义设计和封装的组件和特征。

快速开始

快速创建一个应用

介绍

注意事项

前提条件

工作流程概述

操作步骤

快速创建一个应用

> 本技术指南演示了如何使用 Kubernetes 原生方法高效地创建、管理和访问容器化应用程序，适用于 灵雀云容器平台。

目录

介绍

适用场景

- 新用户希望了解 Kubernetes 平台上基本的应用创建工作流程
- 实践练习，演示核心平台功能，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

预计时间

预计完成时间：10-15 分钟

注意事项

- 本技术指南专注于基本参数，详细配置请参考综合文档
- 所需权限：
  - 创建项目/命名空间
  - 镜像仓库集成
  - 工作负载部署

前提条件

- 对 Kubernetes 架构和 灵雀云容器平台 平台概念有基本了解
- 按照平台建立程序预先配置的项目

工作流程概述

序号	操作步骤	描述
1	创建命名空间	建立资源隔离边界
2	配置镜像仓库	设置容器镜像来源
3	通过部署创建应用程序	创建部署工作负载
4	通过 NodePort 暴露服务	配置 NodePort 服务
5	验证应用可访问性	测试端点连接

操作步骤

创建命名空间

> 命名空间为资源分组和配额管理提供逻辑隔离。

前提条件

- 拥有创建、更新和删除命名空间的权限（例如，管理员或项目管理员角色）
- kubectl 已配置与集群的访问

创建过程

1. 登录并导航到 项目管理 > 命名空间
2. 选择 创建命名空间
3. 配置基本参数：

参数	描述
集群	从项目关联的集群中选择目标集群
命名空间	唯一标识符（自动以项目名称为前缀）

4. 使用默认资源限制完成创建

配置镜像仓库

灵雀云容器平台 支持多种镜像获取策略：

方法 1：通过工具链集成注册表

1. 访问 平台管理 > 工具链 > 集成

2. 启动新的集成：

参数	要求
名称	唯一的集成标识符
API 端点	注册表服务 URL（HTTP/HTTPS）
密钥	预先存在或新创建的凭证

3. 将注册表分配给目标平台项目

方法 2：外部注册表服务

- 使用公共可访问的注册表 URL（例如，Docker Hub）
- 示例：`index.docker.io/library/nginx:latest`

验证要求

- 集群网络必须能够访问注册表端点

通过部署创建应用程序

> 部署提供 Pod 副本集的声明式更新。

创建过程

1. 从 容器平台 视图：

- 使用命名空间选择器选择目标隔离边界

2. 导航到 工作负载 > 部署
3. 点击 创建部署
4. 指定镜像来源：
   - 选择集成注册表 *或*
   - 输入外部镜像 URL（例如，`index.docker.io/library/nginx:latest`）
5. 配置工作负载身份并启动

管理操作

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

通过 **NodePort** 暴露服务

服务使 Pod 组的网络可访问性得以实现。

创建过程

1. 导航到 网络 > 服务
2. 点击 创建服务，并配置参数：

参数	值
类型	NodePort
选择器	目标部署名称
端口映射	服务端口：容器端口（例如，8080:80 ）

3. 确认创建。

关键

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

内部路由通过提供统一的 IP 地址或主机端口来启用工作负载的服务发现。

1. 点击 网络 > 服务。
2. 点击 创建服务。
3. 根据下列参数配置 详细信息，保持其他参数为默认。

参数	描述
名称	输入服务名称。
类型	`NodePort`
工作负载名称	选择之前创建的 `Deployment`。
端口	服务端口：集群中由服务暴露的端口号，即端口，如 `8080`。 容器端口：服务端口映射的目标端口号（或名称），即 targetPort，如 `80`。

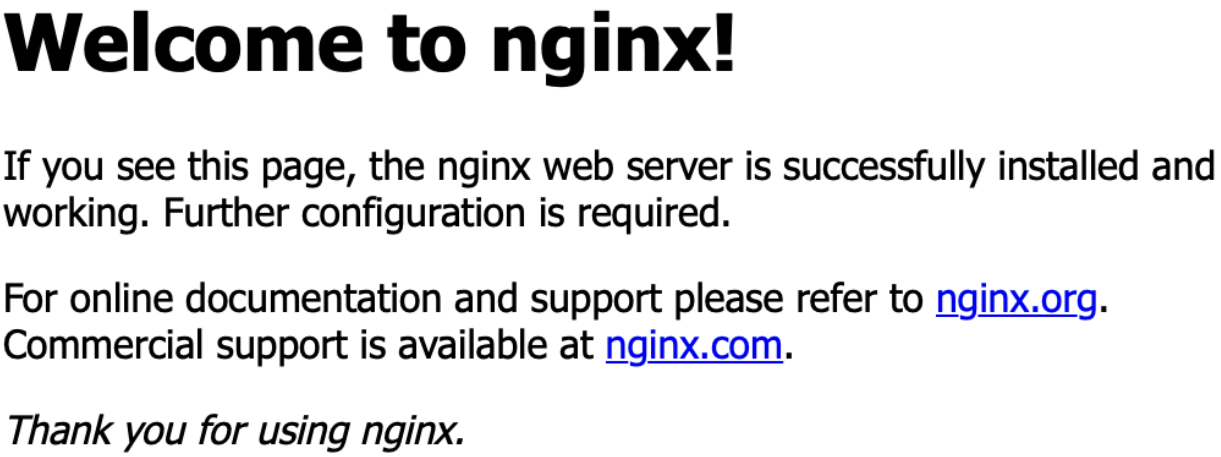
4. 点击 创建。此时，服务成功创建。

验证应用可访问性

验证方法

1. 获取暴露的端点组件：
   - 节点 **IP**：工作节点的公共地址
   - **NodePort**：分配的外部端口
2. 构建访问 URL：`http://<Node_IP>:<NodePort>`

3. 预期结果：Nginx 欢迎页面



构建应用

概述

概述

命名空间管理

应用生命周期管理

Kubernetes 工作负载管理

核心概念

理解参数

Overview

Core Concepts

使用场景和示例

CLI 示例及实际使用

最佳实践

常见问题排查

高级使用模式

理解启动命令

Overview

Core Concepts

使用场景和示例

CLI 示例与实际使用

最佳实践

高级使用模式

理解环境变量

Overview

Core Concepts

使用场景和示例

CLI 示例及实际

最佳实践

功能指南

Namespaces

创建应用前准备工作

创建应用

创建应用后的配置

运维

应用可观测

计算组件

使用 **Helm charts**

1. 了解 Helm
2 通过 CLI 将 Helm Charts 作为应用部署
3 通过 UI 将 Helm Charts 作为应用部署

Pods

使用指南

设置定时任务触发规则

转换时间

编写 Crontab 表达式

概述

灵雀云容器平台 提供统一的接口，通过 Web 控制台和 CLI（命令行界面）创建、编辑、删除和管理云原生应用。应用可以在多个命名空间中部署，并具有 RBAC 策略。

目录

命名空间管理

命名空间为 Kubernetes 资源提供逻辑隔离。主要操作包括：

- 创建命名空间：定义资源配额和 Pod 安全准入策略。
- 导入命名空间：将现有的 Kubernetes 命名空间导入到 灵雀云容器平台 中，提供与原生创建的命名空间完全相同的平台能力。

应用生命周期管理

灵雀云容器平台 支持端到端的生命周期管理，包括：

应用创建模式

在 灵雀云容器平台 中，应用可以通过多种方式创建。以下是一些常见方法：

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

应用操作

- 更新应用：更新应用的镜像版本、环境变量和其他配置，或导入现有的 Kubernetes 资源进行集中管理。
- 导出应用：以 YAML、Kustomize 或 Helm Chart 格式导出应用，然后导入以在其他命名空间或集群中创建新的应用实例。
- 版本管理：支持自动或手动创建应用版本，并在出现问题时可一键回滚到特定版本以快速恢复。
- 删除应用：删除应用时，同时删除应用本身及其直接包含的所有 Kubernetes 资源。此外，此操作还会切断应用与其他未直接包含在其定义中的 Kubernetes 资源之间的任何关联。

应用可观察性

为了持续的操作管理，平台提供日志、事件、监控等功能。

- 日志：支持查看当前运行 Pod 的实时日志，并提供之前容器重启的日志。
- 事件：支持查看命名空间内所有资源的事件信息。
- 监控仪表板：提供命名空间级监控仪表板，包括应用、工作负载和 Pod 的专用视图，并支持自定义监控仪表板以满足特定的操作需求。

Kubernetes 工作负载管理

支持核心工作负载类型：

- Deployments：管理无状态应用的滚动更新。
- StatefulSets：运行具有稳定网络 ID 的有状态应用。
- DaemonSets：部署节点级服务（例如，日志收集器）。
- CronJobs：调度具有重试策略的批处理作业。

核心概念

理解参数

- Overview
- Core Concepts
- 使用场景和示例
- CLI 示例及实际使用
- 最佳实践
- 常见问题排查
- 高级使用模式

理解启动命令

- Overview
- Core Concepts
- 使用场景和示例
- CLI 示例与实际使用
- 最佳实践
- 高级使用模式

理解环境变量

- Overview
- Core Concepts
- 使用场景和示例
- CLI 示例及实际
- 最佳实践

理解参数

目录

高级使用模式

1. 使用 Init Containers 条件传参
2. 使用 Helm 进行参数模板化

Overview

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

Core Concepts

什么是参数？

参数是运行时传递的参数，具有以下特点：

- 覆盖 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"]
```

使用场景和示例

1. 应用配置

向应用传递配置选项：

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

2. 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 示例及实际使用

使用 kubectl run

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

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

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

使用 **kubectl create**

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

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

kubectl apply -f deployment.yaml
```

复杂参数示例

带自定义配置的 **Web** 服务器

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

多参数的应用

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

2. 安全考虑

```
# ❌ 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

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

常见问题排查

1. 参数未被识别

```
# 查看容器日志
kubectl logs pod-name

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

2. 参数覆盖无效

```
# ❌ 错误示例：command 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/s
h

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

高级使用模式

1. 使用 **Init Containers** 条件传参

```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，以适应不同环境和需求。

理解启动命令

目录

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

Overview

Kubernetes 中的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

使用场景和示例

1. 自定义应用启动

使用相同基础镜像运行不同应用：

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

2. 调试和故障排查

覆盖默认命令启动 shell 进行调试：

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

3. 初始化脚本

在启动主应用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

同一镜像用于不同用途：

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

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

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

```
      restartPolicy: Never
```

CLI 示例与实际使用

使用 kubectl run

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

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

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

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

使用 kubectl create job

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

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

复杂启动命令示例

多步骤初始化

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

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

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

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

条件启动逻辑

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

最佳实践

1. 信号处理与优雅关闭

```
# ✅ 正确的信号处理
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: app
    image: myapp:latest
    command: ["/bin/bash"]
    args:
    - "-c"
    - |
      # 捕获 SIGTERM 实现优雅关闭
      trap 'echo "Received SIGTERM, shutting down gracefully..."; kill -T
ERM $PID; wait $PID' TERM

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

      # 等待进程结束
      wait $PID
```

2. 错误处理与日志记录

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

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

      log "Starting application initialization..."

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

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

3. 安全性考虑

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

4. 资源管理

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

高级使用模式

1. 带自定义命令的 Init Containers

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

2. 使用不同命令的 Sidecar 容器

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

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

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

3. 带自定义命令的 Job 模式

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

启动命令为 Kubernetes 中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、可维护且健壮的容器化应用，以满足您的特定需求。

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。它们为向应用程序注入配置信息、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：

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

Kubernetes 中环境变量的来源

Kubernetes 支持多种环境变量来源：

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

环境变量优先级

环境变量覆盖配置的顺序如下：

1. **Kubernetes env**（最高优先级）
2. 引用的 **ConfigMaps/Secrets**
3. **Dockerfile** 中的 **ENV** 指令
4. 应用程序默认值（最低优先级）

使用场景和示例

1. 应用配置

基本的应用设置：

```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 的元数据：

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

4. 不同环境的配置

针对不同环境的配置示例：

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

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

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

CLI 示例及实际使用

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

# 创建用于敏感数据的 Secret
kubectl create secret generic db-secret \
  --from-literal=username=myuser \
  --from-literal=password=mypassword
```

复杂环境变量示例

带服务发现的微服务

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

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

多容器 **Pod** 共享配置

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

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

最佳实践

1. 安全最佳实践

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

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

2. 配置组织

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

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

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

3. 环境变量命名

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

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

4. 默认值和校验

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

功能指南

Namespaces

创建 Namespaces

了解 namespaces

通过 Web 控制台创建 namespaces

通过 CLI 创建 namespace

导入 Namespace

Overview

Use Cases

Prerequisites

Procedure

资源配额

理解资源请求与

配额

扩展资源配额

Pod Security Admission

安全模式

安全标准

配置

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 Names

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

创建应用前准备工作

配置 ConfigMap

理解 ConfigMap

ConfigMap 限制

ConfigMap 与 Secret 对比

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

在 Pod 中使用 ConfigMap 的方式

配置 Secrets

理解 Secrets

创建 Opaque 类型 Secret

创建 Docker 注册表类型 Secret

创建 Basic Auth 类型 Secret

创建 SSH-Auth 类型 Secret

创建 TLS 类型 Secret

通过 Web 控制台创建 Secret

如何在 Pod 中使用 Secret

后续操作

相关操作

创建应用

Creating applications from Ima

Prerequisites

Procedure 1 - Workloads

Procedure 2 - Services

Procedure 3 - Ingress

应用管理操作

参考信息

通过 Chart 创建应用

weight: 40 i18n: title: en: Creating applicat

注意事项

前提条件

操作步骤

状态分析参考

weight: 40 i18n: title: en: Creating applicat

注意事项

前提条件

通过 YAML

注意事项

前提条件

操作步骤

Creating ap

Prerequisites

Procedure

通过 Operator Backed 创建应用

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a417528
70

操作步骤

故障排除

通过 CLI 工

前提条件

操作步骤

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a41752{ 示例
70
参考

操作步骤

故障排除

最终结果：

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a4175286 weight: 70

操作步骤

故障排除

创建应用后的配置

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

配置 VerticalPodAutoscaler (VP

了解 VerticalPodAutoscalers

前提条件

创建 VerticalPodAutoscaler

后续操作

配置 CronH

了解 Cron Horiz

前提条件

创建 Cron Horiz

调度规则说明

运维

状态说明

原生应用

原生应用的启动与停止

启动应用

停止应用

更新应用程序

导入资源

移除/批量移除资

导出应用

导出 Helm Chart

导出 YAML 到本地

导出 YAML 到代码仓库（Alpha）

模板应用的升级与删除

注意事项

前提条件

状态分析说明

原生应用的版

创建版本快照

回滚到历史版本

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

应用可观测

Monitoring Dashboards

Prerequisites

Namespace-Level Monitoring Dashboards

Workload-Level Monitoring

Logs

操作步骤

实时事件

操作步骤

事件记录说明

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 StatefulSe

创建 StatefulSe

管理 StatefulSe

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

使用 **Helm charts**

使用 **Helm charts**

1. 了解 Helm

2 通过 CLI 将 Helm Charts 作为应用部署

3 通过 UI 将 Helm Charts 作为应用部署

Pods

Introduction

Pod 参数

删除 Pods

使用场景

容器

Namespaces

创建 Namespaces

了解 namespaces

通过 Web 控制台创建 namespaces

通过 CLI 创建 namespace

导入 Namespace

Overview

Use Cases

Prerequisites

Procedure

资源配额

理解资源请求与

配额

扩展资源配额

Pod Security Admission

安全模式

安全标准

配置

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 Names

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 Namespaces

目录

了解 namespaces

参考官方 Kubernetes 文档：Namespaces ↗

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

通过 Web 控制台创建 namespaces

> 在与项目关联的集群内，创建一个新的 namespace，需符合项目可用资源配额。 新的 namespace 运行在项目分配的资源配额内（如 CPU、内存），且 namespace 中的所有资源

> 必须位于关联的集群中。

1. 在 项目管理 视图中，点击要创建 namespace 的 *项目名称*。
2. 在左侧导航栏中，点击 **Namespaces** > **Namespaces**。
3. 点击 创建 **Namespace**。
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 namespace。
Namespace	namespace 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 namespace 中的容器指定计算或存储资源的限制（limits），或每次新增 Pod 或 PVC 时，都会消耗此处设置的配额。

   注意：

   - namespace 的资源配额继承自项目在集群中分配的配额。某资源类型的最大允许配额不得超过项目剩余可用配额。如果任何资源的可用配额为 0，则会阻止创建 namespace。请联系平台管理员调整配额。
   - **GPU** 配额配置要求：
     - 仅当集群中已配置 GPU 资源时，才可配置 GPU 配额（vGPU 或 pGPU）。
     - 使用 vGPU 时，也可设置内存配额。

     GPU 单位定义：

     - **vGPU** 单位：100 个虚拟 GPU 单位（vGPU）= 1 个物理 GPU 核心（pGPU）。
       - 注意：pGPU 单位仅以整数计数（如 1 pGPU = 1 核心 = 100 vGPU）。
     - 内存单位：
       - 1 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，默认不设限。
  - 即 namespace 可使用项目分配的该类型所有可用资源，无需显式限制。

配额参数说明

类别	配额类型	数值及单位	说明
存储资源配额	全部	Gi	该 namespace 中所有 Persistent Volume Claims（PVC）请求的存储总容量不得超过此值。
	存储类		该 namespace 中所有关联所选 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 namespace 所属项目。
扩展资源	从配置字典（ConfigMap）获取；详情请参考扩展资源配额说明。	-	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；具体输入规则请参考其他配额说明。	-	为避免资源重复问题，以下配额类型不允许使用： • limits.cpu • limits.memory

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

6. （可选）配置 容器限制范围；详情请参考 Limit Range。

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

8. （可选）在 更多配置 区域，为当前 namespace 添加标签和注解。

   提示：可通过标签定义 namespace 属性，或通过注解补充 namespace 额外信息；两者均可用于过滤和排序 namespaces。

9. 点击 创建。

通过 **CLI** 创建 **namespace**

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,
    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 --defaul
t='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。这为管理员提供了跨分布式环境的统一资源治理和监控能力。

Namespace 解绑：

- 解绑 Namespace 功能允许您将 Namespace 从当前 Project 中解除关联，重置其关联状态，以便后续重新分配或清理。
- 将 Namespace 导入到 Project 中后，该 Namespace 将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（例如资源配额）、统一监控和集中治理控制。

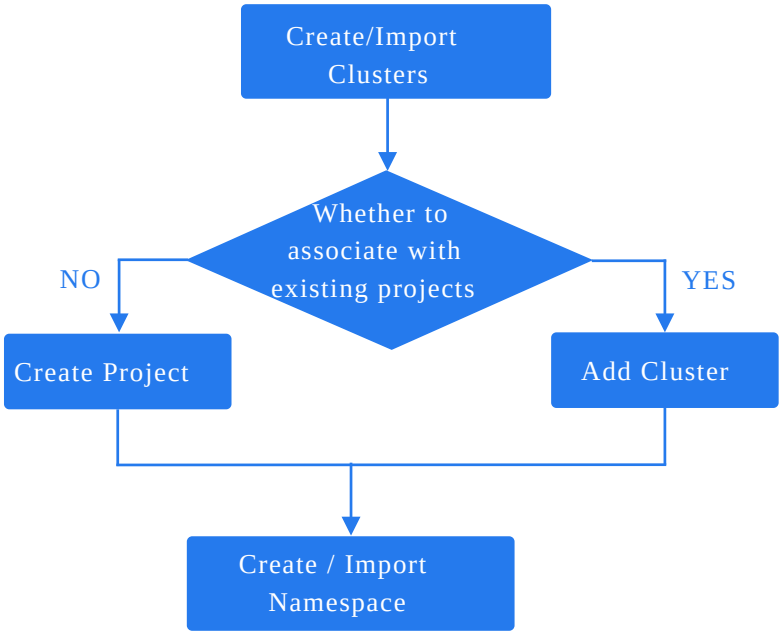
重要说明：

- 一个 Namespace 在任一时刻只能关联到一个 Project。
- 如果 Namespace 已经关联到某个 Project，则必须先解除关联，才能导入或重新分配到另一个 Project。

Use Cases

常见的 **Namespace** 管理用例包括：

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



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

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. 进入 **Project Management**，点击要导入 Namespace 的 ***Project** 名称*。
2. 导航至 **Namespaces** > **Namespaces**。
3. 点击 **Create Namespace** 旁的 下拉按钮，然后选择 **Import Namespace**。
4. 参考创建 Namespace文档了解参数配置详情。
5. 点击 **Import**。

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

资源限制（**Resource Limits**）：定义容器可使用的最大资源，防止资源耗尽，确保集群稳定。

配额

资源配额

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

资源配额用于跟踪命名空间内资源的累计消耗（如容器限制、新建 Pod 或 PVC）。

支持的配额类型

字段	描述
资源请求	命名空间内所有 Pod 的总请求资源： • CPU • 内存
资源限制	命名空间内所有 Pod 的总限制资源： • CPU • 内存
Pod 数量	命名空间内允许的最大 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，则命名空间创建将失败。请联系管理员。
- `Unlimited` 表示该命名空间可使用项目剩余的该资源类型的集群资源。

YAML 文件示例

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

使用 **CLI** 创建资源配额

通过 **YAML** 文件创建

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

直接通过命令行创建

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

存储配额

配额类型：

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

注意：确保存储类已预先分配给包含该命名空间的项目。

扩展资源配额

扩展资源配额通过 **ConfigMap** 定义。如果缺少 ConfigMap，则该资源类别不会出现。

ConfigMap 字段说明

字段	描述
data.dataType	数据类型（例如 `vGPU`）。
data.defaultValue	默认值（为空表示无默认值）。
data.descriptionEn	英文提示文本（鼠标悬停时显示）。
data.descriptionZh	中文提示文本（鼠标悬停时显示）。
data.excludeResources	互斥资源（逗号分隔）。
data.group	资源组（例如 `MPS`）。
data.groupI18n	UI 下拉菜单中显示的英文/中文组名。
data.key	指定键的值。一个配置字典只能描述一个键。
data.labelEn/data.labelZh	资源的英文/中文名称，可在对应配额类型的下拉选项中查看和选择。该字段功能与 data.groupI18n 相同，但仅适用于同一资源只有单一值的情况，确保兼容旧版本配置字典（ConfigMap）。
data.limits	是否配置资源限制。有效值包括：disabled 表示不能配置限制，required 表示必须输入，optional 表示可选输入。
data.requests	是否配置资源请求。有效值包括：disabled 表示不能配置请求，required 表示必须输入，optional 表示可选输入，fromLimits 表示使用与限制相同的配置。
data.relatedResources	关联资源。该字段预留，当前不可用。
data.resourceUnit	资源单位（例如 `cores`、`GiB`）。不支持中文输入。
data.runtimeClassName	运行时类别（默认 GPU 为 `nvidia`）。

字段	描述
metadata.labels	必填标签： • `features.cpaas.io/type: CustomResourceLimitation` • `features.cpaas.io/group: <groupName>` • `features.cpaas.io/enabled`：`true` 或 `false`，该标签必填，表示是否启用，默认值为 true。

metadata.name	格式为 `cf-crl-<*groupName*>-<*name*>`，其中
	• cf-crl 为固定字段，不可更改。
	• groupName 为对应资源组名称，如 gpu-manager、galaxy 等。
	• name 为资源名称：
	  • 资源名称可以是标准资源类型名称，如 cpu、memory、pods 等。标准资源名称必须符合 Kubernetes 的合格名称规则，且必须存在于 Kubernetes 定义的标准资源类型中。
	  • 资源名称也可以是以特定前缀开头的特殊资源类型，如：`hugepages-` 或 `requests.hugepages-`。

metadata.namespace	必须为 `kube-public`

其他配额

自定义配额名称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，中间可包含字母、数字、连字符（-）、下划线（_）或点（.），形成最长 63 字符的合格名称。
- 如果自定义配额名称包含斜杠（/）：名称分为前缀和名称两部分，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规则。
- DNS 子域名：

- 标签：必须以小写字母或数字开头和结尾，可包含连字符（-），但不能全部由连字符组成，最长 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 253 字符。

限制范围

目录

理解限制范围

请参考官方 Kubernetes 文档：限制范围 ↗

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

> 容器的资源请求是资源限制与集群超售之间的比率。资源请求值作为调度器调度容器时的参考和标准。调度器将检查每个节点的可用资源（总资源 - 在该节点上调度的 Pods 中容器的资源请求总和）。如果新 Pod 容器的总资源请求超过该节点剩余的可用资源，则该 Pod 将无法在该节点上调度。

LimitRange 是一个准入控制器：

- 它为所有未设置计算资源要求的容器应用默认请求和限制值。

- 它跟踪使用情况，以确保不超过命名空间内任何 LimitRange 中定义的资源最大值和比率。

包括以下配置

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

使用 **CLI** 创建限制范围

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 --defaul
t='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

安全模式

PSA 定义了三种模式来控制如何处理策略违规：

模式	行为	使用场景
Enforce	拒绝创建/修改不合规的 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。

模式	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** 适用于 Pod 及其控制器（例如 Deployment）。

安全标准

PSA 定义了三种安全标准来限制 Pod 权限：

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

配置

命名空间标签

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
```

例外

对特定用户、命名空间或 runtime class 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']
```

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，提高整个集群的资源效率。

关键概念

- Limits：容器可使用的最大资源。超过限制可能导致 CPU 限流或内存终止。
- 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。该设置定义了命名空间内此比率的可接受范围，平衡资源保障与防止过度消耗。

核心能力

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

示例

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

CPU Request = CPU Limit / Overcommit ratio。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 # 若无此字段则继承集群超售比；0 表示不限制。
  memory: 4 # 若无此字段则继承集群超售比；0 表示不限制。
status:
  clusterCPU: 2 # 集群超售比
  clusterMemory: 3
```

使用 CLI 创建超售比

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

使用 Web 控制台创建/更新超售比

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

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 **Namespaces** > **Namespace** 列表。
2. 点击目标 ***Namespace** 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，设置该命名空间的 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，命名空间默认为 4。 • 容器请求 = 限制 / 集群超售比。 • 若未设置限制，则使用命名空间默认容器配额。
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，直到重建。

管理 **Namespace** 成员

目录

导入成员

平台支持将成员批量导入到 Namespace 中，并分配 Namespace Administrator 或 Developer 等角色，以授予相应权限。

约束与限制

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

前提条件

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

操作步骤

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

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

- 将列表中所有用户作为 Namespace 成员导入，并批量分配角色。
  1. 点击对话框底部 **Set Role** 项右侧的下拉框，选择要分配的角色名称。
  2. 点击 **Import All**。
- 从列表中导入一个或多个用户作为 Namespace 成员。
  1. 勾选用户名/显示名前的复选框，选择一个或多个用户。
  2. 点击对话框底部 **Set Role** 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 **Import**。

添加成员

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

您可以将符合输入要求的有效 OIDC 账号作为 Namespace 角色添加，并为该用户分配相应的 Namespace 角色。

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

有效的 **OIDC** 账号包括：通过平台配置的 IDP 的 OIDC 身份认证服务中的有效账号，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 **Project Management**，点击包含待添加成员的 *项目名称*。
2. 导航至 **Namespaces** > **Namespaces**。
3. 点击待添加成员所在的 ***Namespace** 名称*。
4. 在 **Namespace Members** 标签页，点击 **Add Member**。
5. 在 **Username** 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 **Role** 下拉框中，选择为该用户配置的角色名称。
7. 点击 **Add**。
   添加成功后，您可以在 Namespace 成员列表中查看该成员。
   同时，在用户列表（**Platform Management > User Management**）中也能查看该用户。
   在用户成功登录或同步到本平台之前，来源显示为 `-`，此时可删除该用户；当账号成功登录或同步到平台后，平台会记录账号的来源信息并在用户列表中显示。

移除成员

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

操作步骤

1. 进入 **Project Management**，点击包含待移除成员的 *项目名称*。

2. 导航至 **Namespaces** > **Namespaces**。

3. 点击待移除成员所在的 ***Namespace*** *名称*。

4. 在 **Namespace Members** 标签页，点击待移除成员记录右侧的⋮> **Remove**。

5. 点击 **Remove**。

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**，在左侧边栏导航至 **Namespaces** > **Namespace** 列表。

2. 点击目标 ***namespace name***。

3. 点击 **Actions** > **Update Quota**。

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

通过 **CLI** 更新资源配额

> Resource Quota YAML file example

```
# 第 1 步：编辑命名空间配额
kubectl edit resourcequota <quota-name> -n <namespace-name>

# 第 2 步：验证更改
kubectl get resourcequota <quota-name> -n <namespace-name> -o yaml
```

更新容器 **LimitRanges**

> Limit Range

通过 **Web** 控制台更新 **LimitRange**

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

2. 点击目标 ***namespace name***。

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

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

通过 **CLI** 更新 **LimitRange**

> Limit Range YAML file example

```
# 第 1 步：编辑 LimitRange
kubectl edit limitrange <limitrange-name> -n <namespace-name>

# 第 2 步：验证更改
kubectl get limitrange <limitrange-name> -n <namespace-name> -o yaml
```

更新 Pod Security Admission

> Pod Security Admission

通过 Web 控制台更新 Pod Security Admission

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

通过 CLI 更新 Pod Security Admission

> Update Pod Security Admission CLI command

删除/移除命名空间

您可以选择永久删除命名空间，或将其从当前项目中移除。

目录

删除命名空间

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

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

移除命名空间

移除命名空间：将命名空间从当前项目中移除，但不删除其资源。该命名空间仍然存在于集群中，可以通过 Import Namespace 导入到其他项目中。

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
```

创建应用前准备工作

配置 ConfigMap

理解 ConfigMap

ConfigMap 限制

ConfigMap 与 Secret 对比

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

在 Pod 中使用 ConfigMap 的方式

配置 Secrets

理解 Secrets

创建 Opaque 类型 Secret

创建 Docker 注册表类型 Secret

创建 Basic Auth 类型 Secret

创建 SSH-Auth 类型 Secret

创建 TLS 类型 Secret

通过 Web 控制台创建 Secret

如何在 Pod 中使用 Secret

后续操作

相关操作

配置 ConfigMap

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

目录

理解 ConfigMap

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

`ConfigMap` 对象提供了向容器注入配置信息的机制，同时使容器对 OpenShift Container Platform 保持无感知。ConfigMap 可以用于存储细粒度的信息，如单个属性，也可以存储粗粒度的信息，如整个配置文件或 JSON 块。

`ConfigMap` 对象保存了配置数据的键值对，这些数据可以被 Pod 消费，或者用于存储系统组件（如控制器）的配置信息。例如：

```yaml
# 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（如图片）创建 ConfigMap 时，可以使用 `binaryData` 字段。

配置数据可以通过多种方式在 Pod 中被消费。ConfigMap 可以用于：

- 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 ConfigMap 中。ConfigMap 类似于 Secret，但设计上更方便处理不包含敏感信息的字符串。

ConfigMap 限制

- 必须先创建 ConfigMap，才能在 Pod 中消费其内容。
- 控制器可以编写为容忍缺失的配置信息。请根据具体使用 ConfigMap 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 ConfigMap，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创建 Pod 方式。

ConfigMap 与 Secret 对比

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

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。

参数	说明
	• 添加：可以逐条添加配置项，也可以在 Key 输入框粘贴一行或多行 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，导入的文件不可修改。

批量添加格式示例：

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

5. 点击 **Create**。

通过 CLI 创建 ConfigMap

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

或者从文件创建：

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

操作

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

ConfigMap 的变更会影响引用该配置的工作负载，请提前阅读操作说明。

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

在 Pod 中使用 ConfigMap 的方式

作为环境变量

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

每个键都会成为容器中的一个环境变量。

作为卷中的文件

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

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

每个键对应 `/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
```

配置 Secrets

目录

理解 Secrets

在 Kubernetes (k8s) 中，Secret 是一个基本对象，旨在存储和管理敏感信息，例如密码、OAuth 令牌、SSH 密钥、TLS 证书和 API 密钥。其主要目的是防止敏感数据直接嵌入 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。它们通常经过 base64 编码以便存储，并可以通过多种方式被 Pods 消费，包括作为卷挂载或作为环境变量暴露。

使用特性

- 增强安全性：与明文配置映射（Kubernetes ConfigMap）相比，Secrets 通过使用 Base64 编码存储敏感信息，提供了更好的安全性。此机制结合 Kubernetes 的访问控制能力，显著降低了数据暴露的风险。
- 灵活性和管理：使用 Secrets 提供了一种比将敏感信息硬编码到 Pod 定义文件或容器镜像中更安全、更灵活的方法。这种分离简化了敏感数据的管理和修改，无需更改应用程序代码或容器镜像。

支持的类型

Kubernetes 支持多种类型的 Secrets，每种类型都针对特定用例。平台通常支持以下类型：

- **Opaque**：一种通用的 Secret 类型，用于存储任意键值对的敏感数据，例如密码或 API 密钥。
- **TLS**：专门用于存储 TLS（传输层安全）协议证书和私钥信息，通常用于 HTTPS 通信和安全的入口。
- **SSH Key**：用于存储 SSH 私钥，通常用于安全访问 Git 仓库或其他支持 SSH 的服务。
- **SSH Authentication (kubernetes.io/ssh-auth)**：存储通过 SSH 协议传输的数据的认证信息。
- **Username/Password (kubernetes.io/basic-auth)**：用于存储基本认证凭证（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。

使用方法

Secrets 可以通过不同的方法被应用程序在 Pods 中消费：

- 作为环境变量：可以将 Secret 中的敏感数据直接注入到容器的环境变量中。
- 作为挂载文件（卷）：Secrets 可以作为文件挂载到 Pod 的卷中，允许应用程序从指定的文件路径读取敏感数据。

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

创建 Opaque 类型 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"
```

您可以通过以下方式解码：

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

创建 Docker 注册表类型 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: eyJhdXRocyI6eyJodHRwczovL2luZGV4LmRvY2tlci5pby92MS8i
OnsidXNlcm5hbWUiOiJteXVzZXIiLCJwYXNzd29yZCI6Im15cGFzcyIsImVtYWlsIjoibXlAZ
XhhbXBsZS5jb20iLCJhdXRoIjoiYlhsMWMyVnlPbTE1Y0dGemN3PT0ifX19
```

K8s 会自动将您的用户名、密码、电子邮件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(username:password)
    }
  }
}
```

此 JSON 随后被 base64 编码并用作 Secret 的数据字段值。

在 Pod 中使用它：

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

创建 Basic Auth 类型 Secret

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

创建 SSH-Auth 类型 Secret

用例：存储 SSH 私钥（例如，用于 Git 访问）。

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: ssh-key-secret
type: kubernetes.io/ssh-auth
stringData:
  ssh-privatekey: |
    -----BEGIN OPENSSH PRIVATE KEY-----
    ...
    -----END OPENSSH PRIVATE KEY-----
```

创建 TLS 类型 Secret

用例：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. 在左侧导航栏中，单击 配置 > **Secrets**。
3. 单击 创建 **Secret**。
4. 配置参数。

   提示：在表单视图中，对于输入的用户名、密码等敏感数据，将自动经过 Base64 编码格式转换后储存到 Secret 中，转换后的数据可在 YAML 视图中预览。
5. 单击 创建。

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，并将其分配给环境变量 `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，为原生应用创建工作负载时，可以引用已经创建的 Secrets。

相关操作

您可以在列表页面单击右侧的 (⋮) 或在详情页面单击右上角的 操作，按需更新或删除 Secret。

操作	说明
更新	添加或更新一个 Secret 后，已经通过环境变量引用该 Secret（或其配置项）的工作负载需要重建 Pods，新的配置才能生效。

操作	说明
删除	• 删除 Secret 后，已经通过环境变量引用该 Secret（或其配置项）的工作负载，重建 Pods 时可能会因找不到引用源而受到影响。 • 请不要删除平台自动生成的 Secrets，否则可能导致平台功能无法正常使用。例如：类型为 service-account-token 且包含命名空间资源的认证信息的 Secrets，以及系统命名空间（如 kube-system）中的 Secrets。

创建应用

Creating applications from Ima

Prerequisites

Procedure 1 - Workloads

Procedure 2 - Services

Procedure 3 - Ingress

应用管理操作

参考信息

通过 Chart 创建应用

weight: 40 i18n: title: en: Creating applicat

注意事项

前提条件

操作步骤

状态分析参考

weight: 40 i18n: title: en: Creating applicat

注意事项

前提条件

通过 YAML

注意事项

前提条件

操作步骤

Creating ap

Prerequisites

Procedure

通过 Operator Backed 创建应用

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a41752
70

操作步骤

故障排除

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a41752
70

操作步骤

故障排除

最终结果：

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a4175286 weight: 70

操作步骤

故障排除

通过 CLI 工

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您的项目，您可以使用其中的镜像。如果找不到所需的镜像仓库，请联系管理员进行分配。

- 如果是第三方平台的镜像仓库，请确保当前集群可以直接从该仓库拉取镜像。

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** 进行过滤。**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`）。根据工作负载需求进行调整。

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

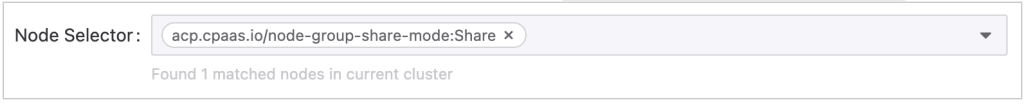
Workload 2 - 配置 Pod

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

1. 在 **Pod** 部分，配置容器运行时参数及生命周期管理：

参数	说明
Volumes	挂载持久卷到容器。支持的卷类型包括 `PVC`、`ConfigMap`、`Secret`、`emptyDir`、`hostPath` 等。具体实现详情见存储卷挂载说明。
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相关 Secret。
More > **Close Grace Period**	Pod 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 共存于同一节点。 执行模式： • **RequiredDuringSchedulingIgnoredDuringExecution**：仅当规则满足时才调度 Pod。 • **PreferredDuringSchedulingIgnoredDuringExecution**：优先选择满足规则的节点，但允许例外。

参数	说明
	配置字段： • `topologyKey`：定义拓扑域的节点标签（默认：`kubernetes.io/hostname`）。 • `labelSelector`：使用标签查询过滤目标 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，则使用命名空间的默认子网。
Static IP Address	绑定持久 IP 地址给 Pod： • 多个 Deployment 中的 Pod 可以申领相同 IP，但同一时间仅允许一个 Pod 使用该 IP。 • 关键：静态 IP 数量必须大于等于 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 数量必须大于等于 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
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： • 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：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： • 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。 • 无命名空间配额则无容器资源限制。
扩展资源	配置集群可用的扩展资源（如 vGPU、pGPU）。
卷挂载	持久存储配置。详见存储卷挂载说明。 操作： • 已有 Pod 卷：点击 **Add** • 无 Pod 卷：点击 **Add & Mount** 参数： • `mountPath`：容器文件系统路径（如 `/data`）

参数	说明
	• `subPath`：卷内相对文件/目录路径。 对于 `ConfigMap` / `Secret`：选择具体键 • `readOnly`：只读挂载（默认读写） 参考 Kubernetes 卷↗。
端口	暴露容器端口。 示例：暴露 TCP 端口 `6379`，名称为 `redis`。 字段： • `protocol`：TCP/UDP • `Port`：暴露端口（如 `6379`） • `name`：符合 DNS 规范的标识（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 **ConfigMap**	以环境变量形式注入整个 ConfigMap/Secret。支持的 Secret 类型：`Opaque`、`kubernetes.io/basic-auth`。
More > 健康检查	• 存活探针：检测容器健康（失败则重启）

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

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

   参见 Init Containers↗。 Init Container：

   1. 在应用容器之前启动（顺序执行）。
   2. 完成后释放资源。
   3. 允许删除条件：

      - Pod 有多个应用容器且至少一个 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**，为集群内运行的应用暴露单一外部访问端点，即使工作负载分布在多个后端。具体参数说明请参见创建 Service。 注意：应用下创建的内部路由默认名称前缀为计算组件名称。若计算组件类型（部署模式）为 StatefulSet，建议不要更改内部路由（工作负载名称）的默认名称，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向省略号（⋮）。
2. 在应用详情页右上角选择 **Actions**。

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

操作	说明
	• 强制更新：触发应用全量滚动，按各组件更新策略执行。 1. 适用场景： • 批量配置变更需立即全集群生效（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： • 大规模重启可能导致短暂服务降级。 • 生产环境使用前需验证业务连续性。 • 网络影响： • 删除 Ingress 规则：若满足条件，外部访问仍可通过 `LB_IP:NodePort` 访问： 1) LoadBalancer Service 使用默认端口。 2) 存在引用应用组件的存活路由规则。 完全终止外部访问需删除 Service。 • 删除 Service：应用组件网络连接不可逆丢失，相关 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。 • 删除前检查清单： 1. 确认关联 Service 无活跃流量。 2. 确认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（含关联 PV）的 PVC。未绑定 PVC 会导致 Pod 创建失败。
ConfigMap	将完整或部分 ConfigMap 数据以文件形式挂载： • 完整 ConfigMap：在挂载路径下创建以键名命名的文件 • 子路径选择：挂载特定键（如 `my.cnf`）
Secret	将完整或部分 Secret 数据以文件形式挂载： • 完整 Secret：在挂载路径下创建以键名命名的文件 • 子路径选择：挂载特定键（如 `tls.crt`）
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。详见临时卷
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`：禁用基于失败的动作 - 默认：连续失败 5 次 → 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`）
HTTP Headers	HTTP/HTTPS	自定义请求头（添加键值对）
Command	EXEC	容器可执行的检查命令（如 `sh -c "curl -I localhost:8080 | grep OK"`）。 注意：需转义特殊字符并测试命令有效性。

1. 直译结果：

目录

weight: 40 i18n: title: en: Creating applications from Chart zh: 通过 Chart 创建应用

通过 **Chart** 创建应用

基于 Helm Chart 表示原生应用的部署模式。Helm Chart 是一组定义 Kubernetes 资源的文件，旨在打包应用程序并促进应用程序的分发，同时具备版本控制功能。这使得环境之间的无缝过渡成为可能，例如从开发环境迁移到生产环境。

注意事项

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

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

前提条件

如果模板源于一个应用，并引用了相关资源（例如，保密字典），请确保待引用的资源在应用部署前已存在于当前命名空间中。

操作步骤

1. 容器平台，在左侧边栏中导航至 应用 > 应用。
2. 单击 创建。
3. 选择 从目录创建 作为创建方式。
4. 选择一个 Chart 并配置参数，选择一个 Chart 并配置必要的参数，如 `resources.requests`、`resources.limits` 和其他与 Chart 紧密相关的参数。
5. 单击 创建。

网页控制台将重定向到 应用 > [原生应用] 详情页面。此过程将需要一些时间，请耐心等待。在操作失败的情况下，请根据界面提示完成操作。

状态分析参考

单击 *应用名称* 以显示 Chart 的详细状态分析。

类型	原因

Initialized	表示 Chart 模板下载的状态。
	• **True**: 表示 Chart 模板已成功下载。
	• **False**: 表示 Chart 模板下载失败；您可以在消息列中检查具体的失败原因。
	  • `ChartLoadFailed`: Chart 模板下载失败。
	  • `InitializeFailed`: 在下载 Chart 之前的初始化过程中出现异常。

Validated	表示用户权限、依赖关系和其他验证的状态。
	• **True**: 表示所有验证检查均已通过。
	• **False**: 表示存在未通过的验证检查；您可以在消息列中检查具体的失败原因。
	  • `DependenciesCheckFailed`: Chart 依赖检查失败。
	  • `PermissionCheckFailed`: 当前用户缺少对某些资源进行操作的权限。
	  • `ConsistentNamespaceCheckFailed`: 在通过模板在原生应用中部署应用时，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；原因列显示为 `ChartSyncFailed`，您可以在消息列中检查具体的失败原因。

WARNING

- 如果模板引用了跨命名空间资源，请联系管理员以获得创建协助。之后，您可以正常在网页控制台上 更新或删除应用。
- 如果模板引用了集群级别资源（例如，存储类），建议联系管理员以获得创建协助。

2. 问题描述：

- “基于 Helm Chart 表示原生应用的部署模式。”这句话的表达不够清晰，建议明确“Helm Chart”是如何与“原生应用的部署模式”相关联的。
- “请确保待引用的资源在应用部署前已存在于当前命名空间中。”这句话可以更简洁地表达。
- “选择一个 Chart 并配置参数，选择一个 Chart 并配置必要的参数”重复了“选择一个 Chart”的表述。
- “网页控制台将重定向到 应用 > [原生应用] 详情页面。”中的“重定向”用词不够准确，建议使用“跳转”。
- “此过程将需要一些时间，请耐心等待。”可以更自然地表达为“该过程可能需要一些时间，请耐心等待。”

3. 意译结果：

---

weight: 40 i18n: title: en: Creating applications from Chart zh: 通过 Chart 创建应用

通过 Chart 创建应用

Helm Chart 是一种原生应用的部署模式。它是一组定义 Kubernetes 资源的文件，旨在打包应用程序并促进应用程序的分发，同时具备版本控制功能。这使得在不同环境之间（如从开发环境迁移到生产环境）进行无缝过渡成为可能。

注意事项

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

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

前提条件

如果模板源于一个应用并引用了相关资源（例如，保密字典），请确保待引用的资源在应用部署前已存在于当前命名空间中。

操作步骤

1. 容器平台，在左侧边栏中导航至 应用 > 应用。
2. 单击 创建。
3. 选择 从目录创建 作为创建方式。
4. 选择一个 Chart 并配置必要的参数，如 `resources.requests`、`resources.limits` 以及其他与 Chart 紧密相关的参数。
5. 单击 创建。

网页控制台将跳转到 应用 > [原生应用] 详情页面。该过程可能需要一些时间，请耐心等待。如果操作失败，请根据界面提示完成操作。

状态分析参考

单击 *应用名称* 以显示 Chart 的详细状态分析。

类型	原因

Initialized	表示 Chart 模板下载的状态。
	• **True**: 表示 Chart 模板已成功下载。
	• **False**: 表示 Chart 模板下载失败；您可以在消息列中检查具体的失败原因。
	  • `ChartLoadFailed`: Chart 模板下载失败。
	  • `InitializeFailed`: 在下载 Chart 之前的初始化过程中出现异常。

Validated	表示用户权限、依赖关系和其他验证的状态。
	• **True**: 表示所有验证检查均已通过。
	• **False**: 表示存在未通过的验证检查；您可以在消息列中检查具体的失败原因。
	  • `DependenciesCheckFailed`: Chart 依赖检查失败。
	  • `PermissionCheckFailed`: 当前用户缺少对某些资源进行操作的权限。
	  • `ConsistentNamespaceCheckFailed`: 在通过模板在原生应用中部署应用时，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；原因列显示为 `ChartSyncFailed`，您可以在消息列中检查具体的失败原因。

WARNING

- 如果模板引用了跨命名空间资源，请联系管理员以获得创建协助。之后，您可以正常在网页控制台上 更新或删除应用。
- 如果模板引用了集群级别资源（例如，存储类），建议联系管理员以获得创建协助。

4. 与之前翻译的文档比较：

- 第一段的内容与之前翻译的内容相似，但表达方式略有不同，保持之前翻译的内容即可。
- 注意事项、前提条件、操作步骤、状态分析参考等部分的内容与之前翻译的内容一致，保持之前翻译的内容即可。
- 警告部分的内容与之前翻译的内容一致，保持之前翻译的内容即可。

最终结果如下：

---

weight: 40 i18n: title: en: Creating applications from Chart zh: 通过 Chart 创建应用

通过 Chart 创建应用

Helm Chart 是一种原生应用的部署模式。它是一组定义 Kubernetes 资源的文件，旨在打包应用程序并促进应用程序的分发，同时具备版本控制功能。这使得在不同环境之间（如从开发环境迁移到生产环境）进行无缝过渡成为可能。

注意事项

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

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

前提条件

如果模板源于一个应用并引用了相关资源（例如，保密字典），请确保待引用的资源在应用部署前已存在于当前命名空间中。

操作步骤

1. 容器平台，在左侧边栏中导航至 应用 > 应用。
2. 单击 创建。
3. 选择 从目录创建 作为创建方式。
4. 选择一个 Chart 并配置必要的参数，如 `resources.requests`、`resources.limits` 以及其他与 Chart 紧密相关的参数。
5. 单击 创建。

网页控制台将跳转到 应用 > [原生应用] 详情页面。该过程可能需要一些时间，请耐心等待。如果操作失败，请根据界面提示完成操作。

状态分析参考

单击 *应用名称* 以显示 Chart 的详细状态分析。

类型	原因

Initialized	表示 Chart 模板下载的状态。
	• **True**: 表示 Chart 模板已成功下载。
	• **False**: 表示 Chart 模板下载失败；您可以在消息列中检查具体的失败原因。
	  • `ChartLoadFailed`: Chart 模板下载失败。
	  • `InitializeFailed`: 在下载 Chart 之前的初始化过程中出现异常。

Validated	表示用户权限、依赖关系和其他验证的状态。 • **True**: 表示所有验证检查均已通过。

类型	原因

	• **False**: 表示存在未通过的验证检查；您可以在消息列中检查具体的失败原因。
	  • `DependenciesCheckFailed`: Chart 依赖检查失败。
	  • `PermissionCheckFailed`: 当前用户缺少对某些资源进行操作的权限。
	  • `ConsistentNamespaceCheckFailed`: 在通过模板在原生应用中部署应用时，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；原因列显示为 `ChartSyncFailed`，您可以在消息列中检查具体的失败原因。

WARNING

- 如果模板引用了跨命名空间资源，请联系管理员以获得创建协助。之后，您可以正常在网页控制台上 更新或删除应用。
- 如果模板引用了集群级别资源（例如，存储类），建议联系管理员以获得创建协助。

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。该方式可以更灵活地配置云原生应用的基本信息和资源。

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**。

Creating applications from Code

通过代码创建应用是使用 Source to Image(S2I) 技术实现的。S2I 是一个自动化框架，用于直接从源代码构建容器镜像。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

Prerequisites

- 完成 Alauda Container Platform Builds 的安装

Procedure

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

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

1. 直译结果：

目录

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a4175286 weight: 70
操作步骤
故障排除
最终结果：
sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a4175286 weight: 70
操作步骤
故障排除

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a4175286 weight: 70

通过 Operator Backed 创建应用

Operator backed 应用是由 Operator 提供的一组资源集合。基于这些 Operator backed 应用，您可以快速部署一个组件应用，并利用 Operator 的能力自动化管理应用的整个生命周期。

操作步骤

1. **Container Platform**，在左侧导航栏中，导航至 **Applications** > **Applications**。
2. 单击 **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 自定义资源参数。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）规范，包括：
   - `spec.resources.limits` （容器级别资源限制）。
   - `spec.resourceQuota` （Operator 定义的配额政策）。其他特定于 CR 的参数如 `spec.replicas`、`spec.storage.className` 等。
5. 单击 **Create**。

网页控制台将导航至 **Applications** > **Operator Backed Apps** 页面。

> **INFO**
>
> 注意： Kubernetes 资源创建过程需要异步协调。根据集群状况，完成可能需要几分钟。

故障排除

如果资源创建失败：

1. 检查控制器协调错误：

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

2. 验证 API 资源可用性：

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

3. 验证 CRD/Operator 准备好后重试创建：

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

4. 存在的问题：

- “Operator backed 应用”中的“应用”可以更明确为“应用程序”以符合技术文档的表达习惯。
- “基于这些 Operator backed 应用”中的“应用”重复，建议改为“基于这些 Operator backed 应用程序”。
- “网页控制台将导航至”可以更清晰地表达为“网页控制台将跳转到”。
- “Kubernetes 资源创建过程需要异步协调”中的“协调”可以更准确地表述为“调和”。
- “根据集群状况，完成可能需要几分钟”中的“完成”不够明确，建议改为“创建完成”。

3. 意译结果：

---

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a4175286 weight: 70

通过 Operator Backed 创建应用程序

Operator backed 应用程序是由 Operator 提供的一组资源集合。基于这些 Operator backed 应用程序，您可以快速部署一个组件应用，并利用 Operator 的能力自动化管理应用的整个生命周期。

操作步骤

1. **Container Platform**，在左侧导航栏中，导航至 **Applications** > **Applications**。
2. 单击 **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 自定义资源参数。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）规范，包括：
   - `spec.resources.limits` （容器级别资源限制）。
   - `spec.resourceQuota` （Operator 定义的配额政策）。其他特定于 CR 的参数如 `spec.replicas`、`spec.storage.className` 等。
5. 单击 **Create**。

网页控制台将跳转到 **Applications** > **Operator Backed Apps** 页面。

> **INFO**
>
> 注意： Kubernetes 资源创建过程需要异步调和。根据集群状况，创建可能需要几分钟。

故障排除

如果资源创建失败：

1. 检查控制器调和错误：

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

2. 验证 API 资源可用性：

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

3. 验证 CRD/Operator 准备好后重试创建：

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

4. 比较结果：

- 第一段内容“通过 Operator Backed 创建应用”与之前翻译的“通过 Operator Backed 创建应用”一致，保持不变。
- 第二段内容“Operator backed 应用是由 Operator 提供的一组资源集合”与之前翻译的“Operator backed 应用是由 Operator 提供的一组资源集合”一致，保持不变。
- 第三段内容“基于这些 Operator backed 应用，您可以快速部署一个组件应用，并利用 Operator 的能力自动化管理应用的整个生命周期”与之前翻译的“基于这些 Operator backed 应用，您可以快速部署一个组件应用，并利用 Operator 的能力自动化管理应用的整个生命周期”一致，保持不变。
- 其余段落内容与之前翻译的内容相似，保持不变。

最终结果：

sourceSHA: d8decb364fe429c3f25f1e3195cc6816d6ef435a9b693262387ab419a4175286 weight: 70

通过 **Operator Backed** 创建应用程序

Operator backed 应用程序是由 Operator 提供的一组资源集合。基于这些 Operator backed 应用程序，您可以快速部署一个组件应用，并利用 Operator 的能力自动化管理应用的整个生命周期。

操作步骤

1. **Container Platform**，在左侧导航栏中，导航至 **Applications** > **Applications**。
2. 单击 **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 自定义资源参数。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）规范，包括：
   - `spec.resources.limits` （容器级别资源限制）。
   - `spec.resourceQuota` （Operator 定义的配额政策）。其他特定于 CR 的参数如 `spec.replicas`、`spec.storage.className` 等。
5. 单击 **Create**。

网页控制台将跳转到 **Applications** > **Operator Backed Apps** 页面。

> **INFO**
>
> 注意： Kubernetes 资源创建过程需要异步调和。根据集群状况，创建可能需要几分钟。

故障排除

如果资源创建失败：

1. 检查控制器调和错误：

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

2. 验证 API 资源可用性：

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

3. 验证 CRD/Operator 准备好后重试创建：

```
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 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↗

创建应用后的配置

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，指定希望运行的最小和最大 Pod 数量，以及 Pod 应达到的 CPU 利用率或内存利用率目标。

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 自动扩缩器会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。

对于复制控制器（replication controllers），此扩缩容直接对应复制控制器的副本数。对于部署配置（deployment configurations），扩缩容直接对应部署配置的副本数。请注意，自动扩缩仅适用于处于 Complete 阶段的最新部署。

平台会自动考虑资源情况，防止在资源峰值（如启动期间）时发生不必要的自动扩缩。处于未就绪状态的 Pod 在扩容时视为 0 CPU 使用率，缩容时会被忽略。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，必须配置就绪检查以判断新 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，控制器管理器会根据 HPA 配置查询 CPU、内存利用率或两者。控制器管理器通过资源指标 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，控制器会将利用率计算为每个 Pod 中容器对应资源请求的百分比。然后控制器计算所有目标 Pod 的平均利用率，并生成一个比例，用于调整期望副本数。

支持的指标

水平 Pod 自动扩缩器支持以下指标：

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

指标	描述
网络入站流量	进入 Pod 的网络流量，单位为 KiB/s。
网络出站流量	从 Pod 发出的网络流量，单位为 KiB/s。
存储读取流量	从存储读取的数据量，单位为 KiB/s。
存储写入流量	写入存储的数据量，单位为 KiB/s。

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

前提条件

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

创建水平 **Pod** 自动扩缩器

使用 **CLI**

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

1. 创建一个名为 `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. 要扩缩的部署名称。
4. 最大副本数。
5. 最小副本数。
6. 指定要扩缩对象的 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，可以通过以下命令查看部署的新状态：

```
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 *部署名称*。
4. 向下滚动至 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支持的最大 **Pod** 数量。最大或最小 Pod 数量可在设置后更改，建议先通过性能测试推导更准确的值，并在使用过程中持续调整以满足业务需求。
触发策略	列出对业务变化敏感的指标及其目标阈值，用于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**）	针对有特定扩缩速率需求的业务，可以通过指定扩容步长或缩容步长，逐步适应业务量变化。 缩容步长支持自定义稳定窗口，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。

需求

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

传统（核心指标）**HPA**

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

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

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

自定义指标 **HPA**

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

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


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

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

示例中，`scaleTargetRef` 指定工作负载。

触发条件定义

- `metrics` 为数组类型，支持多个指标
- `metric type` 可为：Object（描述 k8s 资源）、Pods（描述每个 Pod 的指标）、Resources（内置 k8s 指标：CPU、内存）、External（通常为集群外部指标）
- 若自定义指标非由 Prometheus 提供，需通过创建 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 实际兼容原有核心指标（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 的平均值
- `targetAverageUtilization` 是基于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`，并结合 `xxxValue` 和 `type`：

- `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% 范围外时才触发自动扩缩。
- 缩容可能影响正在运行的业务，请谨慎操作。

计算规则

当业务指标变化时，平台会根据以下规则自动计算匹配业务量的目标 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[(所有 Pod 实际指标值之和 / 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：

  - 若计算出的目标 Pod 数量超过设置的最大 **Pod** 数量（例如 *4*），平台仅扩容到 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方法，如增加命名空间 Pod 配额或添加硬件资源。
  - 若计算出的目标 Pod 数量（如示例中的 *5*）小于根据扩容步长调整后的 Pod 数量（如 *10*），平台仅扩容到 5 个 Pod。

- 多策略目标 Pod 数量：取各策略计算结果中的最大值。

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。当数据足够时，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，并随着新数据的获取不断更新其推荐。VPA 可运行于以下模式：

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

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

支持的功能

VerticalPodAutoscaler 根据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 管理员 页面。
2. 点击 **Marketplace** > 集群插件，进入 集群插件 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 自动伸缩配置：

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 通过工作负载的选择器查找需要调整资源的 Pod。支持的工作负载类型包括 DaemonSet、Deployment、ReplicaSet、StatefulSet、ReplicationController、Job 和 CronJob。
4. 指定要伸缩对象的 API 版本。
5. 指定对象类型。
6. VPA 应用的目标资源。
7. 定义 VPA 如何应用推荐的更新策略。updateMode 可选：
   - Auto：创建 Pod 时自动设置资源请求，并更新当前 Pod 至推荐资源请求。目前等同于“Recreate”。此模式可能导致应用停机。未来支持就地更新时，Auto 模式将采用该机制。
   - 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，可为不同容器设置特定策略。例如，将容器模式设为“Auto”表示为该容器计算推荐，“Off”表示不计算推荐。

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

2. 应用 YAML 文件创建 VPA：

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

示例输出：

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

3. 创建 VPA 后，可通过以下命令查看推荐：

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

示例输出（部分）：

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

使用 **Web** 控制台

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 工作负载 > **Deployments**。

3. 点击 ***Deployment** 名称*。

4. 向下滚动至 弹性伸缩 区域，点击右侧的 更新。

5. 选择 垂直伸缩 并配置伸缩规则。

参数	说明
伸缩模式	目前支持 手动伸缩 模式，通过分析历史资源使用情况提供推荐的资源配置，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，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，之后不修改。
- `minReplicas: <number>` - 最小副本数。确保在 Updater 驱逐 Pod 时，至少保持此数量的 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 模式下应用 VPA 推荐会触发 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）支持基于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，根据时间表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，平台会开始监控时间表，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）和目标从节点数。当达到调度时间时，CronHPA 会将 Pod 数量调整为指定的目标，从而不受当前资源利用率影响。

CronHPA 如何工作？

cron horizontal pod autoscaler（CronHPA）扩展了基于时间控制的 Pod 自动扩缩容概念。CronHPA 允许您定义特定时间点调整 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。

CronHPA 通过持续检查当前时间与定义的调度规则进行对比。当达到调度时间时，控制器会将 Pod 数量调整为该调度规则指定的目标从节点数。如果多个调度规则同时触发，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 > **Platform Health Status**，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）的定时调度。
8. 调度触发时的目标从节点数。

该示例配置了 Deployment：

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

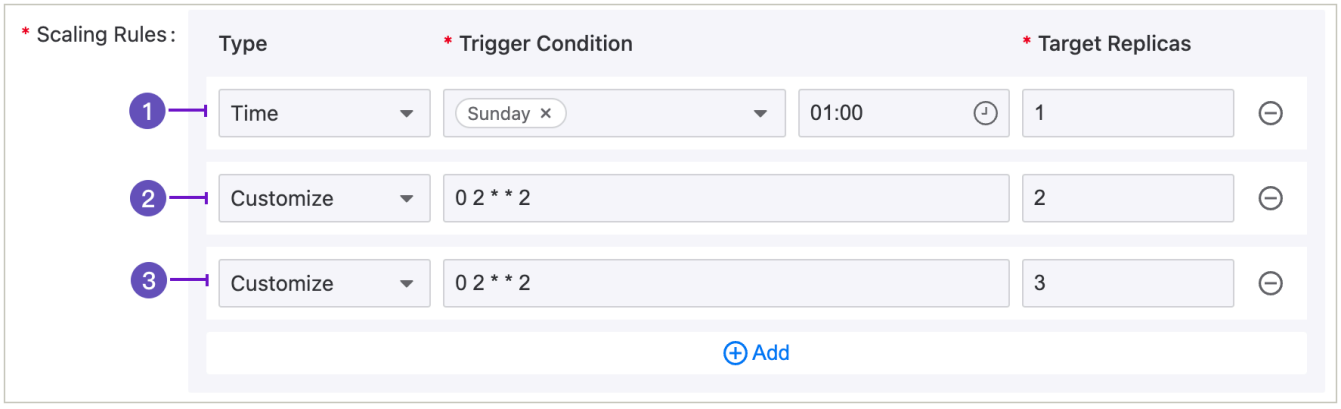
2. 应用 YAML 文件创建 CronHPA：

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

使用 **Web** 控制台

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Workloads** > **Deployments**。
3. 点击 ***Deployment Name***。
4. 向下滚动到 **Elastic Scaling** 部分，点击右侧的 **Update**。
5. 选择 **Scheduled Scaling**，配置扩缩容规则。当类型为 **Custom** 时，必须提供格式为 `minute hour day month week` 的 Crontab 表达式作为触发条件。详细介绍请参考 Writing Crontab Expressions。
6. 点击 **Update**。

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，请谨慎设计扩缩容策略。

- 调度使用 crontab 格式（`minute hour day month week`），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，请确保定时扩缩容不会在高峰期意外降低容量。

运维

状态说明

原生应用

原生应用的启动与停止

启动应用

停止应用

更新应用程

导入资源

移除/批量移除资

导出应用

导出 Helm Chart

导出 YAML 到本地

导出 YAML 到代码仓库（Alpha）

模板应用的升级与删除

注意事项

前提条件

状态分析说明

原生应用的版

创建版本快照

回滚到历史版本

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

状态说明

目录

原生应用

原生应用的状态及对应含义如下。状态后的数字表示计算组件个数。

状态	含义
运行中	所有计算组件均处于正常运行状态。
部分运行	部分计算组件运行中，部分计算组件已停止。
已停止	所有计算组件均已停止。
处理中	至少有一个计算组件处于待处理状态。
无计算组件	应用下没有计算组件。
失败	部署失败。

说明：相似地，计算组件状态中的数字表示容器组个数。

Deployment

- 运行中：所有 Pods 均处于正常运行状态。
- 处理中：有 Pods 未处于运行中状态。
- 已停止：所有 Pods 均已停止。
- 失败：部署失败。

原生应用的启动与停止

目录

启动应用

1. 进入 **Container Platform**。

2. 在左侧导航栏中，单击 应用 > 应用。

3. 单击应用的名称。

4. 单击 启动。

停止应用

1. 进入 **Container Platform**。

2. 在左侧导航栏中，单击 应用 > 应用。

3. 单击应用的名称。

4. 单击 停止。

5. 阅读提示信息，确认无误后，单击 停止。

更新应用程序

自定义应用程序极大地方便了工作负载、网络、存储和配置的统一管理，但并非所有资源都属于该应用程序。

- 在应用程序创建过程中添加的资源，或通过应用程序更新添加的资源，默认与应用程序关联，无需额外导入。
- 在应用程序外部创建的资源不属于该应用程序，无法在应用程序的详细信息中找到。然而，只要资源定义满足业务需求，业务仍然可以正常运行。在这种情况下，建议将资源导入应用程序以实现统一管理。
- 镜像管理
  - 使用标签/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
  - 导入现有的 Kubernetes 资源（Deployments/Services/Ingresses）
  - 使用 `kubectl apply -f` 在命名空间之间同步配置

导入到应用程序中的资源可以享受以下功能：

功能	描述
版本快照	当为应用程序创建版本快照时，应用程序内的资源也会生成快照。 • 如果应用程序回滚，资源也将回滚到快照中的状态。

功能	描述
	• 如果分发应用程序的特定版本，平台将在重新部署应用程序时自动创建快照中记录的资源。
与应用程序一起删除	如果不再需要某个应用程序，删除该应用程序将自动移除与该应用程序关联的所有资源，包括计算组件、内部路由和入站规则。
更易查找	在应用程序详细信息中，您可以快速查看与该应用程序关联的资源。 例如：外部流量可以通过属于 *Application A* 的 *Service S* 访问 *Deployment D*，但只有当 *Service S* 也属于 *Application A* 时，才能在应用程序详细信息中快速找到相应的访问地址。

目录

导入资源

> 在应用程序所在的命名空间下批量导入相关资源；一个资源只能属于一个应用程序。

1. 进入 容器平台。

2. 在左侧导航栏中，点击 应用程序管理 > 原生应用程序。

3. 点击 *应用程序名称*。

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：常见的资源类型包括 Deployment、DaemonSet、StatefulSet、Job、CronJob、Service、Ingress、PVC、ConfigMap、Secret 和 HorizontalPodAutoscaler，这些资源类型显示在顶部；其他资源按字母顺序排列，您可以通过搜索关键字快速查询特定资源类型。

6. 在 资源 部分，选择要导入的资源。

   注意：对于 **Job** 类型的资源，仅支持通过 YAML 创建的任务进行导入。

7. 点击 导入资源。

移除/批量移除资源

> 从应用程序中移除/批量移除资源仅会解除应用程序与资源的关联，并不会删除资源。

如果应用程序下的资源之间存在相互连接，从应用程序中移除任何资源不会改变资源之间的关联。例如，即使 *Service S* 从 *Application A* 中移除，外部流量仍然可以通过 *Service S* 访问 *Deployment D*。

1. 进入 容器平台。
2. 在左侧导航栏中，点击 应用程序管理 > 原生应用程序。
3. 点击 *应用程序名称*。
4. 点击 操作 > 管理资源。
5. 点击资源右侧的 移除 以移除该资源；或一次选择多个资源，然后点击表格顶部的 移除 以批量移除资源。

导出应用

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

目录

导出 **Helm Chart**

操作步骤

1. 进入 容器平台。

2. 在左侧导航栏点击 应用管理 > 原生应用。

3. 点击类型为 `Custom Application` 的 *应用名称*。

4. 点击 操作 > 导出；也可以在应用详情页导出指定版本。

5. 根据需要选择一种导出方式，并参考以下说明配置相关信息。

   - 导出 Helm Chart 到具有管理权限的模板仓库

     注意：模板仓库由平台管理员添加，请联系平台管理员获取具有 管理 权限的类型为 **Chart** 或 **OCI Chart** 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 权限的类型为 **Chart** 或 **OCI Chart** 的模板仓库。分配给该 模板仓库 的项目负责人可以直接使用该模板。
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	描述内容会显示在应用目录的应用模板列表中。
README	描述文件，支持 Markdown 格式编辑，会显示在应用模板详情页。

参数	说明
NOTES	模板帮助文件，支持标准纯文本编辑；部署模板完成后，会显示在模板应用详情页。

- 导出 Helm Chart 到本地以手动上传到模板仓库：选择目标位置为 本地，文件格式选择 **Helm Chart**，生成 Helm Chart 包并下载到本地进行离线传输。

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

方式一

1. 进入 容器平台。
2. 在左侧导航栏点击 应用管理 > 原生应用。
3. 点击 *应用名称*。
4. 点击 操作 > 导出；也可以在应用详情页导出指定版本。
5. 选择目标位置为 本地，文件格式为 **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。

方式二

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

类型	说明
完整 **YAML**	默认未选中 预览简化 **YAML**，显示隐藏了 **managedFields** 字段 的 YAML 文件。 您可以预览并直接导出；也可以取消勾选 隐藏 **managedFields** 字段 导出完整 YAML 文件。 注意：完整 YAML 主要用于运维和排查，不能用于平台上快速创建原生应用。
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以直接将原生应用 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，也不支持直接导出 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. 进入 容器平台。

2. 在左侧导航栏点击 应用管理 > 原生应用。

3. 点击类型为 `Custom` 的 *应用名称*。

4. 点击 操作 > 导出；也可以在应用详情页导出指定版本。

5. 根据需要选择一种导出方式，并参考以下说明配置相关信息。

   - 导出 YAML 到代码仓库：

参数	说明
目标位置	选择 代码仓库，将 YAML 文件直接同步到指定的 Git 代码仓库。分配给该 代码仓库 的项目负责人可以直接使用该 YAML 文件。
集成项目名称	由平台管理员分配或关联给您的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	• 现有分支：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，源分支会被自动删除。

文件路径	文件在代码仓库中应保存的具体位置；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。

参数	说明
提交信息	填写提交信息，用于标识本次提交的内容。
预览	预览将提交的 YAML 文件，并与代码仓库中已有的 YAML 文件进行差异对比，差异部分以颜色区分显示。

- 导出 Kustomize 类型文件到本地以手动上传到代码仓库：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到本地。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。

模板应用的升级与删除

由于当前模板应用与原生应用的功能存在重叠，且原生应用下的模板应用拥有更丰富的运维能力，因此未来版本将不再提供独立的模板应用管理能力，请尽快将当前成功部署的模板应用升级为原生应用。

目录

注意事项

此功能 即将下线。请尽快将当前成功部署的模板应用升级为原生应用。

前提条件

请联系平台管理员开启模板应用相关功能。

状态分析说明

单击 *模板应用名称*，在详情信息中可展示 Chart 的详细部署状态分析。

类型	原因

Initialized	表示 Chart 模板下载的状态。
	• 状态为 True 时表示 Chart 模板下载成功。
	• 状态为 False 时表示 Chart 模板下载失败，消息列可查看具体的失败原因。
	  • ChartLoadFailed：Chart 模板下载失败。
	  • InitializeFailed：在下载 Chart 之前的初始化过程中出现异常。

Validated	表示 Chart 模板用户权限、依赖等验证的状态。
	• 状态为 True 时表示所有验证检查均已通过。
	• 状态为 False 时表示存在验证检查未通过，消息列可查看具体的失败原因。
	  • 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：通过原生应用部署模板应用时，Chart 中包含的资源需要跨命名空间部署。

Synced	表示 Chart 模板部署的状态。 • 状态为 True 时表示 Chart 模板部署成功。 • 状态为 False 时表示 Chart 模板部署失败，原因列显示为 ChartSyncFailed，消息列可查看具体的失败原因。

原生应用的版本管理

通过平台界面更新应用后，会自动生成历史版本记录。对于非界面操作引发的应用更新，例如通过调用 API 更新应用后，可以手动创建版本快照来记录变更。

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，并自动删除其他条目，优先删除最早创建的版本快照条目。

目录

创建版本快照

操作步骤

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 应用管理 > 原生应用。
3. 单击 *应用名称*。
4. 在 版本快照 页签中，单击 创建版本快照。

5. 配置信息，单击 确定。

   说明：您也可以 分发应用，即将应用的版本快照分发为 Chart，方便在平台上多个集群的多个命名空间中快速部署同一个应用。

回滚到历史版本

将当前应用的配置回滚至历史版本。

操作步骤

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 应用管理 > 原生应用。
3. 单击 *应用名称*。
4. 在 历史版本 页签中，单击 *版本号*。
5. 单击⋮> 回滚至该版本。
6. 单击 回滚。

删除应用

删除一个应用时，它会同时删除该应用本身及其直接包含的所有 Kubernetes 资源。此外，此操作还会切断该应用与其他未直接包含在其定义中的 Kubernetes 资源之间的任何关联。

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操作，例如重启容器或路由流量。没有适当的健康检查，Kubernetes 无法可靠地管理应用的生命周期，可能导致服务性能下降或中断。

Kubernetes 提供三种类型的探针：

- `livenessProbe`：检测容器是否仍在运行。如果活跃性探针失败，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。活跃性和就绪探针在启动探针成功之前不会执行。对于启动时间较长的应用非常有用。

正确配置这些探针对于构建健壮且自愈的 Kubernetes 应用至关重要。

探针类型

Kubernetes 支持三种实现探针的机制：

HTTP `GET` 操作

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

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

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

`exec` 操作

> 在容器内执行指定命令。如果命令以状态码 0 退出，则探针成功。

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

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

TCP `Socket` 操作

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

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

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

最佳实践

- 活跃性 vs. 就绪性：
  - 活跃性：如果应用无响应，最好重启它。失败时，Kubernetes 会重启容器。
  - 就绪性：如果应用暂时无法提供服务（例如连接数据库中），但可能无需重启即可恢复，使用就绪探针。这可以防止流量路由到不健康的实例。
- 慢启动应用使用启动探针：对于启动时间较长的应用，使用启动探针。这样可以避免因活跃性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快速执行。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开始探测。
- 调节 **periodSeconds** 和 **failureThreshold**：平衡快速检测失败和避免误报。探针过于频繁或 failureThreshold 过低可能导致不必要的重启或未就绪状态。
- 调试日志：确保应用日志清晰记录健康检查端点调用和内部状态，便于调试探针失败。
- 组合使用探针：通常，活跃性、就绪性和启动探针会一起使用，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 # Container image
          ports:
            - containerPort: 80 # Container exposed port
          startupProbe:
            httpGet:
              path: /startup-check
              port: 8080
            initialDelaySeconds: 0 # Usually 0 for startup probes, or very small
            periodSeconds: 5
            failureThreshold: 60 # Allows 60 * 5 = 300 seconds (5 minutes) for startup
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 5 # Delay 5 seconds after Pod starts before checking
            periodSeconds: 10 # Check every 10 seconds
            timeoutSeconds: 5 # Timeout after 5 seconds
            failureThreshold: 3 # Consider unhealthy after 3 consecutive failures
          readinessProbe:
            httpGet:
              path: /ready
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 10
            timeoutSeconds: 5
            failureThreshold: 3
```

通过 **Web** 控制台配置健康检查参数

常用参数

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

协议特定参数

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

探针失败故障排查

当 Pod 状态显示与探针相关的问题时，可按以下步骤排查：

检查 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（如连接被拒绝、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**：使用 `nc`（netcat）或 `telnet` 从同一网络内的另一个 Pod 或主机测试 TCP 连接：`kubectl exec -it <another-pod> -- nc -vz <pod-ip> <probe-port>`。

检查探针配置

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

检查应用代码

- 确保应用的健康检查端点实现正确，真实反映应用的就绪和活跃状态。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` 限制和请求。

网络问题

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

应用可观测

Monitoring Dashboards

- 支持查看平台上工作负载组件过去7天的资源监控数据（监控数据保留周期可配置）。包括应用、工作负载、Pod 的统计信息，以及 CPU/内存使用的趋势和排名。
- 支持命名空间级别监控。
- 支持的工作负载级别监控包括：**Applications**、**Deployments**、**DaemonSets**、**StatefulSets** 和 **Pods**

目录

Prerequisites

- Installation of Monitoring Plugins

Namespace-Level Monitoring Dashboards

Procedure

1. 在 **Container Platform**，点击 **Observe** > **Dashboards**。
2. 查看命名空间下的监控数据。提供三个监控面板：**Applications Overview**、**Workloads Overview** 和 **Pods Overview**。
3. 切换不同监控面板以监控目标 **Overview**。

Creating Namespace-Level Monitoring Dashboard

1. 在 **Platform Management** 中，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"`

Workload-Level Monitoring

> 本操作步骤演示如何通过 Deployment 界面查看 Pod 监控。

Default Monitoring Dashboard

Procedure

1. 在 **Container Platform**，点击 **Workloads** > **Deployments**。
2. 从列表中点击某个 Deployment 名称。
3. 进入 **Monitoring** 标签页查看默认监控指标。

Metric interpretation

Monitoring Resource	Metric Granularity	Technical Definition
CPU	**Utilization/Usage**	**Utilization** = Usage/Limit (limits) 评估容器限制配置。高利用率表示限制不足。 **Usage** 表示实际资源消耗。
Memory	**Utilization/Usage**	**Utilization** = Usage/Limit (limits) 评估方法同 CPU。高比例可能导致组件不稳定。
Network Traffic	**Inflow Rate/Outflow Rate**	Pod 的网络流量（字节/秒），包括流入和流出。
Network Packet	**Receiving Rate/Transmit Rate**	Pod 接收和发送的网络包数（个数/秒）。
Disk Rate	**Read/Write**	每个工作负载挂载卷的读写吞吐量（字节/秒）。
Disk IOPS	**Read/Write**	每个工作负载挂载卷的每秒输入/输出操作次数（IOPS）。

Custom Monitoring Dashboard

4. 点击 **Toggle Icon** 切换到自定义监控面板。参考自定义监控面板中添加图表创建专用的 **Workload-Level** 监控面板。

> **INFO**
>
> 将鼠标悬停在图表曲线上，可查看特定时间点的每个 Pod 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 条数据。

Logs

聚合容器运行时日志并提供可视化查询功能。当应用、工作负载或其他资源出现异常行为时，日志分析有助于诊断根本原因。

目录

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** 标签页查看详细记录。

操作	说明
Pod/Container	通过下拉选择器切换 Pod 和 Container，以查看对应的日志。
Previous Logs	查看已终止容器的日志（当容器 restartCount > 0 时可用）。

操作	说明
Lines	配置显示日志缓冲区大小：1k/10k/100k 行。
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：当日志中出现截断标记（...）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** 或 **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。历史日志分析请使用 Observe 下的 Logs。

实时事件

Kubernetes 资源状态变化和操作状态更新所产生的事件信息，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. 从列表中点击某个 Deployment 名称。

3. 切换到 **Events** 标签页查看详细记录。

事件记录说明

资源事件记录：在事件摘要面板下方，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。每张卡片显示：

- 资源类型：Kubernetes 资源类型，用图标缩写表示：

  - `P` = Pod
  - `RS` = ReplicaSet
  - `D` = Deployment
  - `SVC` = Service
- 资源名称：目标资源名称。
- 事件原因：Kubernetes 报告的原因（例如 FailedScheduling）。
- 事件级别：事件严重性。
  - `Normal` ：信息类
  - `Warning` ：需立即关注
- 时间：最后发生时间，发生次数。

> **INFO**
>
> Kubernetes 允许管理员通过 Event TTL 控制器配置事件保留周期，默认保留周期为 1 小时。过期事件会被系统自动清理。欲查看完整历史记录，请访问 All Events。

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 StatefulSe
- 创建 StatefulSe
- 管理 StatefulSe

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

Deployments

目录

理解 Deployments

参考官方 Kubernetes 文档：Deployments ↗

> **Deployment** 是 Kubernetes 中的高级工作负载资源，用于声明式地管理和更新应用程序的 Pod 副本。它提供了一种强大且灵活的方式来定义应用程序的运行方式，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**，您可以：

- 声明式管理：定义应用程序的期望状态，Kubernetes 会自动确保集群的实际状态与期望状态匹配。
- 版本控制与回滚：跟踪 Deployment 的每个修订版本，出现问题时轻松回滚到之前的稳定版本。
- 零停机更新：使用滚动更新策略逐步更新应用程序，无需中断服务。
- 自我修复：Deployment 会自动替换崩溃、终止或从节点移除的 Pod 实例，确保始终有指定数量的 Pod 可用。

工作原理：

1. 通过 Deployment 定义应用程序的期望状态（例如使用哪个镜像，运行多少副本）。
2. Deployment 创建一个 ReplicaSet，确保指定数量的 Pod 正在运行。
3. ReplicaSet 创建并管理实际的 Pod 实例。
4. 当更新 Deployment（例如更改镜像版本）时，Deployment 会创建一个新的 ReplicaSet，并根据预定义的滚动更新策略逐步替换旧的 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，必须匹配 selector.matchLabels
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2 # 容器镜像
          ports:
            - containerPort: 80 # 容器暴露端口
          resources: # 资源限制和请求
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 200m
              memory: 256Mi
```

通过 YAML 创建 Deployment

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

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

使用 Web 控制台创建 Deployment

前提条件

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

- 对于前者，管理员通常会将镜像仓库分配给您的项目，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。
- 如果是第三方平台的镜像仓库，确保当前集群可以直接拉取该镜像。

操作步骤 - 配置基本信息

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

> **INFO**
>
> 注意：使用 Web 控制台集成的镜像仓库中的镜像时，可以通过 **Already Integrated** 过滤镜像。集成项目名称，例如镜像（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。

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： 最大激增数（`maxSurge`）： • 更新期间允许超过期望副本数的最大 Pod 数。

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

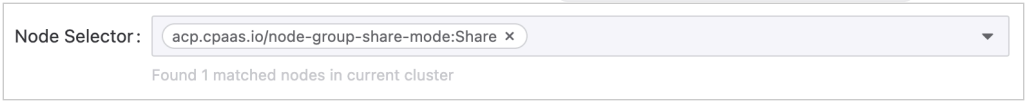
操作步骤 - 配置 **Pod**

注意：在混合架构集群中部署单架构镜像时，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。具体实现请参见 卷挂载指南。
Pull Secret	仅在从第三方镜像仓库拉取镜像（通过手动输入镜像 URL）时需要。 注意：用于从受保护的镜像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`）。 - 在此期间，Pod 会完成正在处理的请求并释放资源。 - 设置为 `0` 会强制立即删除（SIGKILL），可能导致请求中断。

2. **Node Affinity** 规则

参数	说明
More > **Node Selector**	限制 Pod 调度到具有特定标签的节点（例如 `kubernetes.io/os: linux`）。 
More > **Affinity**	基于现有规则定义细粒度调度规则。 **Affinity** 类型： • **Pod Affinity**：将新 Pod 调度到已运行特定 Pod 的节点（同拓扑域）。 • **Pod Anti-affinity**：防止新 Pod 与特定 Pod 共置。 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`：使用标签查询过滤目标 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。如未指定，使用命名空间的默认子网。
Static IP Address	绑定持久 IP 地址到 Pod： • 多个 Deployment 中的 Pod 可声明相同 IP，但同一时间仅允许一个 Pod 使用该 IP。 • 关键：静态 IP 数量必须 ≥ Pod 副本数。

- Calico

参数	说明
Static IP Address	分配严格唯一的固定 IP： • 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。

参数	说明
	• **Limits**：容器运行时允许的最大 CPU/内存。单位定义请参见 资源单位。 命名空间超售比： • 无超售比： 若存在命名空间资源配额，容器请求/限制继承命名空间默认值（可修改）。 无命名空间配额时，无默认值，自定义请求。 • 有超售比： 请求自动计算为 `Limits / 超售比`（不可修改）。 约束： • 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`：选择特定键 • `readOnly`：以只读方式挂载（默认读写） 详见 Kubernetes 卷↗。

参数	说明
端口	暴露容器端口。 示例：暴露 TCP 端口 `6379`，名称为 `redis`。 字段： • `protocol`：TCP/UDP • `Port`：暴露端口（如 `6379`） • `name`：符合 DNS 规范的标识符（如 `redis`）
启动命令与参数	覆盖默认 ENTRYPOINT/CMD： 示例 **1**：执行 `top -b` - **Command**: `["top", "-b"]` - 或 Command: `["top"]`，Args: `["-b"]` 示例 **2**：输出 `$MESSAGE`： `/bin/sh -c "while true; do echo $(MESSAGE); sleep 10; done"` 详见 定义命令↗。
More > 环境变量	• 静态值：直接的键值对 • 动态值：引用 ConfigMap/Secret 键、Pod 字段（`fieldRef`）、资源指标（`resourceFieldRef`） 注意：环境变量会覆盖镜像或配置文件中的设置。
More > 引用的 **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。

参数	说明
More > 日志文件	配置日志路径： - 默认：采集 `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	排除特定日志采集（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. 在应用容器之前启动（顺序执行）。
   2. 完成后释放资源。
   3. 允许删除条件：
      - Pod 有多个应用容器且至少一个 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：挂载特定键（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：挂载特定键（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 删除时删除 使用场景：容器间文件共享、临时数据存储。详见 EmptyDir
Host Path	挂载宿主机目录（必须以 `/` 开头，如 `/volumepath`）。

健康检查

- 健康检查 YAML 文件示例
- Web 控制台健康检查配置参数

管理 **Deployments**

使用 **CLI** 管理 **Deployment**

查看 **Deployment**

- 检查 Deployment 是否已创建。

```
kubectl get deployments
```

- 获取 Deployment 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。可通过检查滚动状态验证：

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

删除 **Deployment**

删除 Deployment 会同时删除其管理的 ReplicaSet 及所有相关 Pods。

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

使用 **Web** 控制台管理 **Deployment**

查看 **Deployment**

您可以查看 Deployment 以获取应用信息。

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

删除 **Deployment**

1. 在 **Container Platform**，进入 **Workloads** > **Deployments**。
2. 找到要删除的 Deployment。
3. 在 操作 下拉菜单中点击 **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。

检查 **Deployment** 状态

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

检查 **ReplicaSet** 状态

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

检查 Pod 状态

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

查看日志

```
kubectl logs <pod-name> -c <container-name> # 查看指定容器日志
kubectl logs <pod-name> --previous          # 查看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

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 **Node** 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 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）。
- 状态为 `NotReady` 的节点。
- 具有不兼容 **Taints** 且在 DaemonSet Pod 模板中未配置匹配 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，DaemonSet 创建的每个 Pod 都符合此模板
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations: # 这些容忍配置允许 daemonset 在控制平面节点上运行，如不需运行可移除
        - key: node-role.kubernetes.io/control-plane
          operator: Exists
          effect: NoSchedule
        - key: node-role.kubernetes.io/master
          operator: Exists
          effect: NoSchedule
      containers:
        - name: fluentd-elasticsearch
          image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:
            - name: varlog
              mountPath: /var/log
      # 可以设置较高的优先级类，确保 DaemonSet 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

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

# 第二步：验证守护进程集及其管理的 Pods 状态
kubectl get daemonset fluentd-elasticsearch # 查看 DaemonSet
kubectl get pods -l name=fluentd-elasticsearch -o wide # 查看该 DaemonSet
管理的 Pods 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，请联系管理员进行分配。
- 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。**Integration Project Name** 例如 images (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。

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 DaemonSet Pod 零停机更新的 `rollingUpdate` 策略。 最大不可用数（`maxUnavailable`）：更新过程中允许临时不可用的最大 Pod 数量。支持绝对值（如 1）或百分比（如 10%）。 示例：若有 10 个节点且 `maxUnavailable` 为 10%，则允许最多 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（或百分比时为 10%）。 • 非运行状态 **Pod**：处于 `Pending` 或 `CrashLoopBackOff` 等状态的 Pod 被视为不可用。 • 同时限制：`maxSurge` 和 `maxUnavailable` 不能同时为 0 或 0%。若百分比计算结果均为 0，Kubernetes 会强制将 `maxUnavailable` 设为 1 以保证更新进度。

操作步骤 - 配置 Pod

Pod 部分，请参考 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，DaemonSet 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 Taints 的节点）。
  - 节点状态为 `Ready` 且 `Schedulable: true` 。
- ❌ 排除节点：
  - 带有 `NoSchedule` Taint（除非显式容忍）。
  - 手动标记为不可调度的节点（ `kubectl cordon` ）。
  - 状态为 `NotReady` 或 `Unschedulable` 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

请参考官方 Kubernetes 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 按逆序编号更新：Pod-N → Pod-0。

在分布式系统中，可以将多个 StatefulSets 作为独立组件部署，以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 `kubectl` 并连接到集群。

YAML 文件示例

```yaml
# example-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # 必须与 .spec.template.metadata.labels 匹配
  serviceName: 'nginx' # 该无头 Service 负责 Pod 的网络身份
  replicas: 3 # 定义期望的 Pod 副本数（默认：1）
  minReadySeconds: 10 # 默认为 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 根据这些模板动态创建唯一的 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 的创建和状态：
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 # 查看由 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，请联系管理员进行分配。
- 对于第三方平台的镜像仓库，请确保当前集群能够直接拉取镜像。

操作步骤 - 配置基本信息

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

INFO

注意：使用 Web 控制台集成的镜像仓库时，可以通过 **Already Integrated** 过滤镜像。集成项目名称例如 images (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。

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

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

Volume Claim Templates	`volumeClaimTemplates` 是 StatefulSets 的关键特性，支持为每个 Pod 动态创建持久存储。StatefulSet 中的每个 Pod 副本都会基于预定义模板自动获得独立的 PersistentVolumeClaim (PVC)。
	• 1. 动态 **PVC** 创建：为每个 Pod 自动创建唯一的 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	描述
	• 3. 存储类：通过 storageClassName 指定存储后端。若未指定，则使用集群默认 StorageClass。支持多种云/本地存储类型（如 SSD、HDD）。 • 4. 容量：通过 resources.requests.storage 配置存储容量。示例：1Gi。若 StorageClass 支持，支持动态扩容。

操作步骤 - 配置 **Pod**

Pod 部分，请参考 Deployment - Configure Pod

操作步骤 - 配置容器

Containers 部分，请参考 Deployment - Configure Containers

操作步骤 - 创建

点击 **Create**。

健康检查

- 健康检查 YAML 文件示例
- Web 控制台健康检查配置参数

管理 **StatefulSets**

使用 **CLI** 管理 **StatefulSet**

查看 **StatefulSet**

您可以查看 StatefulSet 以获取应用信息。

- 查看已创建的 StatefulSet。

```
kubectl get statefulsets
```

- 获取 StatefulSet 详情。

```
kubectl describe statefulsets
```

扩缩容 **StatefulSet**

- 修改现有 StatefulSet 的副本数：

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

- 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）并应用更改：

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

- 然后，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 `volumeClaimTemplates` 使用的 StorageClass 的 `reclaimPolicy` 为 `Delete`，则在删除 PVC 时，PV 及其底层存储会自动被删除。

使用 **Web** 控制台管理 **StatefulSet**

查看 **StatefulSet**

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

删除 **StatefulSet**

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

CronJobs

目录

理解 CronJobs

请参考官方 Kubernetes 文档：

- CronJobs↗
- 使用 CronJob 运行自动化任务↗

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

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

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**，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 区域，配置任务执行方式及相关参数。

   执行类型：

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

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，调度必须使用 UTC。 • Kubernetes 集群版本 ≥ v1.25：支持时区感知调度（默认使用用户本地时区）。
并发策略	指定并发 Job 执行的处理方式（`Allow`、`Forbid` 或 `Replace`，详见 K8s 规范↗）。

   Job 历史保留：

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

- 超出保留限制时，最旧的 Job 会被优先垃圾回收。

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

参数	说明
Job 类型	选择 Job 完成模式（`非并行`、`固定完成次数的并行`或`索引 Job`，详见 K8s Job 模式↗）。
重试次数限制	设置标记 Job 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**，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. 点击 CronJob 列表右侧的竖向省略号（⋮）。
  2. 点击 立即执行。（或者在 CronJob 详情页，点击右上角的操作菜单，选择 立即执行）。

- **CLI**：

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

查看 **Job** 详情：

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

监控执行状态

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

删除 **CronJobs**

使用 **Web** 控制台删除 **CronJobs**

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

2. 找到要删除的 CronJobs。

3. 在 操作 下拉菜单中，点击 删除 按钮并确认。

使用 **CLI** 删除 **CronJobs**

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

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 # 任务特定策略（Never/OnFailure）
      containers:
        - name: processor
          image: alpine:3.14
          command: ['/bin/sh', '-c']
          args:
            - echo "Processing data..."; sleep 30; echo "Job completed"
```

执行概览

Kubernetes 中的每次 Job 执行都会创建一个专用的 Job 对象，允许用户：

- 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>
```

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过程。Helm 使用一种称为 *charts* 的打包格式。Helm chart 是一组描述 Kubernetes 资源的文件集合。在集群中创建 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版本。

1.1. 主要功能

Helm 提供以下能力：

- 在 chart 仓库中搜索大量 charts
- 修改已有的 charts
- 使用 Kubernetes 资源创建自己的 charts
- 打包应用并以 charts 形式共享

1.2. 目录

目录基于 Helm 构建，提供了一个全面的 Chart 分发管理平台，突破了 Helm CLI 工具的局限。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	一种应用打包格式	
HelmRequest	CRD。定义部署 Helm Chart 所需的配置	模板应用
ChartRepo	CRD。对应 Helm charts 仓库	模板仓库
Chart	CRD。对应 Helm Charts	模板

1.3 了解 HelmRequest

在 Alauda Container Platform 中，Helm 部署主要通过一个名为 **HelmRequest** 的自定义资源进行管理。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 release，而 Alauda Container Platform 使用 HelmRequest 资源来定义、部署和管理 Helm charts。主要区别包括：

1. 声明式 **vs** 命令式：HelmRequest 提供声明式的 Helm 部署方式，而传统 Helm CLI 是命令式的。
2. **Kubernetes** 原生：HelmRequest 是直接集成 Kubernetes API 的自定义资源。
3. 持续调和：Captain 持续监控并调和 HelmRequest 资源与其期望状态。
4. 多集群支持：HelmRequest 支持通过平台跨多个集群部署。
5. 平台功能集成：HelmRequest 可与其他平台功能（如 Application 资源）集成。

HelmRequest 与 Application 集成

HelmRequest 和 Application 资源在概念上相似，用户可能希望统一查看它们。平台提供了将 HelmRequest 同步为 Application 资源的机制。

用户可以通过添加以下注解，将 HelmRequest 标记为以 Application 形式部署：

```
alauda.io/create-app: "true"
```

启用此功能后，平台 UI 会显示额外字段，并提供跳转到对应 Application 页面的链接。

部署工作流

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 并同步为 Application
6. **Application** 提供已部署资源的统一视图

组件定义

- **HelmRequest**：描述期望 Helm chart 部署的自定义资源定义
- **Captain**：处理 HelmRequest 资源并管理 Helm release 的控制器（源码地址：https://github.com/alauda/captain↗）
- **Release**：Helm chart 的已部署实例
- **Charon**：监控 HelmRequest 并创建对应 Application 资源的组件
- **Application**：已部署资源的统一表示，提供额外管理能力
- **Archon-api**：平台内负责特定高级 API 功能的组件

2 通过 CLI 将 Helm Charts 作为应用部署

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 使用一种称为 charts 的打包格式。chart 是一组描述 Kubernetes 资源的文件集合。单个 chart 可用于部署从简单 Pod 到复杂应用栈的任何内容。

参考官方文档：Helm Charts Documentation↗

示例 chart 目录结构：

```
nginx/
├── Chart.lock
├── Chart.yaml
├── README.md
├── charts/
│   └── common/
│       ├── Chart.yaml
│       ├── README.md
│       ├── templates/
│       │   ├── _affinities.tpl
│       │   ├── _capabilities.tpl
│       │   ├── _errors.tpl
│       │   ├── _images.tpl
│       │   ├── _ingress.tpl
│       │   ├── _labels.tpl
│       │   ├── _names.tpl
│       │   ├── _secrets.tpl
│       │   ├── _storage.tpl
│       │   ├── _tplvalues.tpl
│       │   ├── _utils.tpl
│       │   ├── _warnings.tpl
│       │   └── validations/
│       │       ├── _cassandra.tpl
│       │       ├── _mariadb.tpl
│       │       ├── _mongodb.tpl
│       │       ├── _postgresql.tpl
│       │       ├── _redis.tpl
│       │       └── _validations.tpl
│       └── values.yaml
├── ci/
│   ├── ct-values.yaml
│   └── values-with-ingress-metrics-and-serverblock.yaml
├── templates/
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── extra-list.yaml
│   ├── health-ingress.yaml
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── ldap-daemon-secrets.yaml
│   ├── pdb.yaml
│   ├── server-block-configmap.yaml
```

```
|   ├── serviceaccount.yaml
|   ├── servicemonitor.yaml
|   ├── svc.yaml
|   └── tls-secrets.yaml
├── values.descriptor.yaml
├── values.schema.json
└── values.yaml
```

关键文件说明：

- `values.descriptor.yaml`（可选）：配合 ACP UI 显示用户友好表单
- `values.schema.json`（可选）：验证 values.yaml 内容并渲染简单 UI
- `values.yaml`（必需）：定义 chart 部署参数

2.3 打包 Chart

使用 `helm package` 命令打包 chart：

```
helm package nginx
# 输出：Successfully packaged chart and saved it to: /charts/nginx-8.8.0.tgz
```

2.4 获取 API 令牌

1. 在 **Alauda Container Platform** 中，点击右上角头像 => **Profile**
2. 点击 **Add Api Token**
3. 输入合适的描述和剩余有效期
4. 保存显示的令牌信息（仅显示一次）

2.5 创建 Chart 仓库

通过 API 创建本地 chart 仓库：

```
curl -k --request POST \
--url https://$ACP_DOMAIN/catalog/v1/chartrepos \
--header 'Authorization:Bearer $API_TOKEN' \
--header 'Content-Type: application/json' \
--data '{
  "apiVersion": "v1",
  "kind": "ChartRepoCreate",
  "metadata": {
    "name": "test",
    "namespace": "cpaas-system"
  },
  "spec": {
    "chartRepo": {
      "apiVersion": "app.alauda.io/v1beta1",
      "kind": "ChartRepo",
      "metadata": {
        "name": "test",
        "namespace": "cpaas-system",
        "labels": {
          "project.cpaas.io/catalog": "true"
        }
      },
      "spec": {
        "type": "Local",
        "url": null,
        "source": null
      }
    }
  }
}'
```

2.6 上传 Chart

将打包好的 chart 上传到仓库：

```
curl -k --request POST \
--url https://$ACP_DOMAIN/catalog/v1/chartrepos/cpaas-system/test/charts
\
--header 'Authorization:Bearer $API_TOKEN' \
--data-binary @"/root/charts/nginx-8.8.0.tgz"
```

2.7 上传相关镜像

1. 拉取镜像：`docker pull nginx`

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

3. 加载并推送到私有仓库：

```
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/$NAM
ESPACE/applications \
--header 'Authorization:Bearer $API_TOKEN' \
--header 'Content-Type: application/json' \
--data '{
  "apiVersion": "app.k8s.io/v1beta1",
  "kind": "Application",
  "metadata": {
    "name": "test",
    "namespace": "catalog-ns",
    "annotations": {
      "app.cpaas.io/chart.source": "test/nginx",
      "app.cpaas.io/chart.version": "8.8.0",
      "app.cpaas.io/chart.values": "{\"image\":{\"pullPolicy\":\"IfNotPre
sent\"}}"
    },
    "labels": {
      "sync-from-helmrequest": "true"
    }
  }
}'
```

2.9 更新应用

使用 PATCH 请求更新应用：

```
curl -k --request PATCH \
--url https://$ACP_DOMAIN/acp/v1/kubernetes/$CLUSTER_NAME/namespaces/$NAM
ESPACE/applications/test \
--header 'Authorization:Bearer $API_TOKEN' \
--header 'Content-Type: application/merge-patch+json' \
--data '{
  "apiVersion": "app.k8s.io/v1beta1",
  "kind": "Application",
  "metadata": {
    "annotations": {
      "app.cpaas.io/chart.values": "{\"image\":{\"pullPolicy\":\"Always
\"}}"
    }
  }
}'
```

2.10 卸载应用

删除 Application 资源：

```
curl -k --request DELETE \
--url https://$ACP_DOMAIN/acp/v1/kubernetes/$CLUSTER_NAME/namespaces/$NAM
ESPACE/applications/test \
--header 'Authorization:Bearer $API_TOKEN'
```

2.11 删除 Chart 仓库

```
curl -k --request DELETE \
--url https://$ACP_DOMAIN/apis/app.alauda.io/v1beta1/namespaces/cpaas-sys
tem/chartrepos/test \
--header 'Authorization:Bearer $API_TOKEN'
```

3 通过 UI 将 Helm Charts 作为应用部署

3.1 工作流概览

将模板添加到可管理仓库 → 上传模板 → 管理模板版本

3.2 前提条件

模板仓库由平台管理员添加。请联系平台管理员获取具有 管理 权限的 Chart 或 OCI Chart 类型模板仓库名称。

3.3 将模板添加到可管理仓库

1. 进入 **Catalog**。
2. 在左侧导航栏点击 **Helm Charts**。
3. 点击页面右上角的 **Add Template**，根据以下参数选择模板仓库。

参数	说明
模板仓库	直接同步模板到具有 管理 权限的 Chart 或 OCI Chart 类型模板仓库。分配给该 模板仓库 的项目负责人可以直接使用该模板。
模板目录	当选择的模板仓库类型为 OCI Chart 时，必须选择或手动输入存放 Helm Chart 的目录。 注意：手动输入新模板目录时，平台会在模板仓库中创建该目录，但存在创建失败风险。

4. 点击 **Upload Template**，上传本地模板到仓库。
5. 点击 **Confirm**。模板上传过程可能需要几分钟，请耐心等待。

   注意：当模板状态从 `Uploading` 变为 `Upload Successful` 时，表示模板上传成功。
6. 若上传失败，请根据提示排查。

注意：非法文件格式表示上传的压缩包内文件存在问题，如内容缺失或格式错误。

3.4 删除模板的特定版本

当某个模板版本不再适用时，可以删除该版本。

操作步骤

1. 进入 **Catalog**。
2. 在左侧导航栏点击 **Helm Charts**。
3. 点击对应 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该版本。

Pod

Introduction

Introduction

Pod 参数

Pod 参数

删除 Pods

删除 Pods

使用场景

操作步骤

容器

介绍

Debug 容器

- 实现原理
- 注意事项
- 使用场景
- 操作步骤

使用 **Exec** 进

- 通过应用程序进
- 通过 Pod 进入容

介绍

参考 Kubernetes 官方网站文档：Pod ↗

Pod 参数

平台界面提供了容器组的各类信息，以供快捷查找。以下是部分参数解释。

参数	说明
资源请求/限制	容器组内有效的资源（CPU、内存）请求与限制值。请求与限制值的计算方式相同，文档以限制值为例进行介绍，具体的规则及算法如下： • 当容器组仅包含 业务容器 时，**CPU/**内存 的限制值为容器组内所有容器的 **CPU/**内存 限制值的总和。例如：如果容器组包含两个业务容器，**CPU/**内存 的限制值分别为 100m/100Mi 和 50m/200Mi，则容器组的 **CPU/**内存 限制值将为 150m/300Mi。 • 当容器组既有 初始化容器 又有 业务容器 时，容器组的 **CPU/**内存 限制值计算步骤如下： 1. 取所有初始化容器 **CPU/**内存 限制值的最大值。 2. 取所有业务容器 **CPU/**内存 限制值的总和。 3. 将结果进行比较，取初始化容器和业务容器中 **CPU**、内存 的最大值作为容器组的 **CPU/**内存 限制值。 计算示例：如果容器组包含两个初始化容器，**CPU/**内存 的限制值分别为 100m/200Mi 和 200m/100Mi，则初始化容器的 **CPU/**内存 最大限制值为 200m/200Mi。同时，如果容器组还包含两个业务容器，**CPU/**内存 的限制值分别为 100m/100Mi 和 50m/200Mi，则业务容器的 **CPU/**内存 限制值总和为 150m/300Mi。因此，容器组的综合 **CPU/**内存 限制值为 200m/300Mi。
来源	容器组所属的计算组件。
重启次数	当容器组状态异常时，重启的次数。

参数	说明
节点	容器组所在节点的名称。
Service Account	Service Account 是 Pod 里的进程和服务访问 Kubernetes APIServer 的一个账号，为进程和服务提供了一种身份标识。仅当当前登录用户拥有平台管理员角色或平台审计人员角色时，**Service Account** 字段可见，且可以查看 Service Account 的 YAML 文件。

删除 **Pods**

删除 Pods 可能会影响计算组件的运行，请谨慎操作。

目录

使用场景

- 迅速恢复 Pods 至期望状态：如果 Pods 处于影响业务运行的状态，如 `Pending` 或 `CrashLoopBackOff`，在根据报错信息处理后，手动删除 Pods 可以帮助其迅速恢复至期望状态，例如 `Running`。此时，被删除的 Pods 将在当前节点重建或被重新调度。
- 运维管理中的资源清理：一些 Pods 到达指定阶段后不再变化，这些组通常会大量积累，给其他 Pods 的管理带来困扰。待清理的 Pods 包括因节点资源不足而被驱逐的 `Evicted` 状态 Pods，或是由周期性调度任务触发的 `Completed` 状态 Pods。在此情况下，被删除的 Pods 将不再存在。

  注意：对于调度任务，如果您需要查看每次任务的执行日志，不建议删除对应的 `Completed` 状态 Pods。

操作步骤

1. 进入 **Container Platform**。

2. 在左侧导航栏中，单击 **Workloads** > **Pods**。

3. （逐个删除）单击待删除 Pods 右侧的⋮> 删除，并确认。

4. （批量删除）选择待删除 Pods，单击列表上方的 删除，并确认。

容器

介绍

Debug 容器

实现原理

注意事项

使用场景

操作步骤

使用 **Exec** 进

通过应用程序进

通过 Pod 进入容

介绍

参考 Kubernetes 官方网站文档：Containers↗。

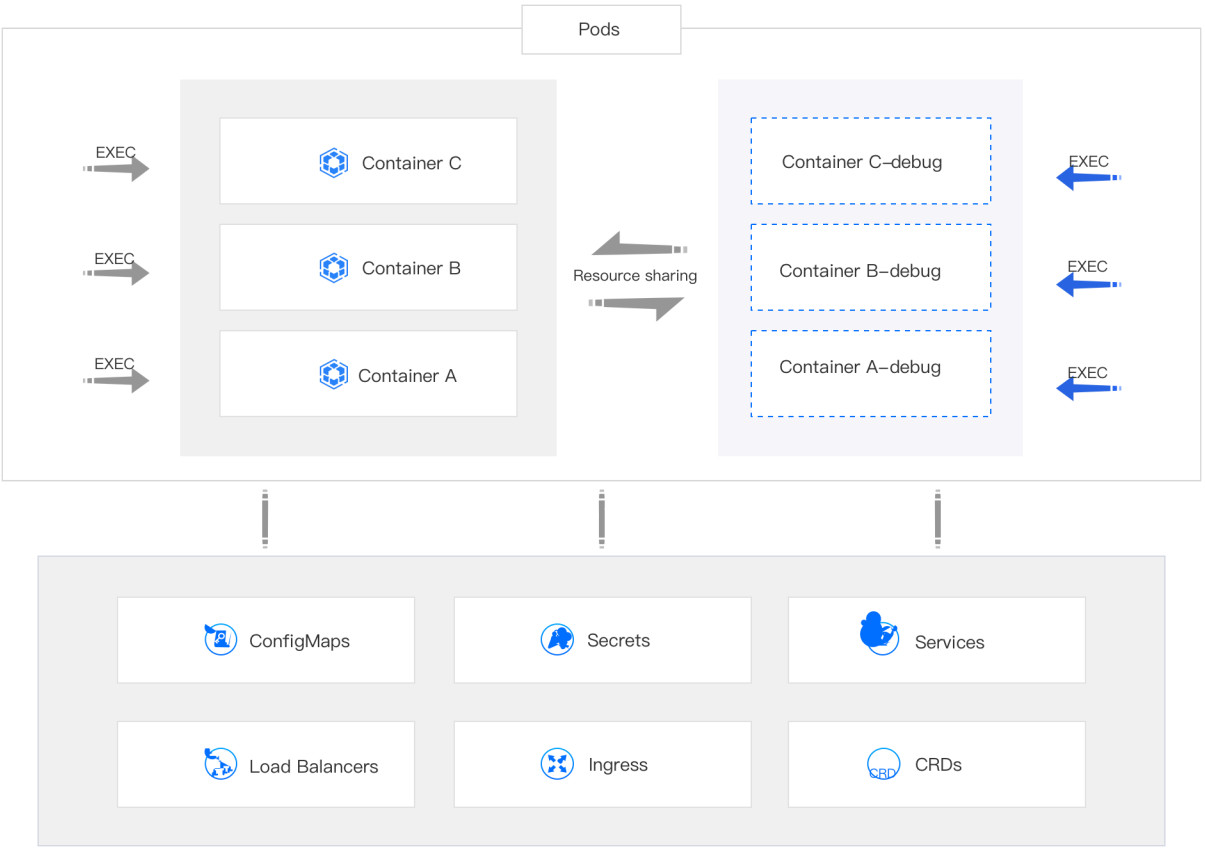
Debug 容器（Alpha）

Debug 功能提供了系统、网络及磁盘等相关工具，可用于调试运行中的容器。

目录

实现原理

Debug 功能通过临时容器（Ephemeral Container）实现。临时容器是一种特殊类型的容器，与业务容器共享资源。您可以将临时容器（例如 *容器 A-debug*）添加到容器组中，并在该容器内使用调试工具进行调试。调试结果将直接应用于业务容器（如 *容器 A*）。



注意事项

- 您无法通过直接更新容器组配置的方式来添加临时容器，请务必通过 Debug 功能启用临时容器。
- Debug 功能启用的临时容器没有资源或调度保证，并且不会自动重启。请避免在其中运行业务应用，除非是进行调试。
- 如果容器组所在节点的资源即将耗尽，请谨慎使用 Debug 功能，因为这可能导致容器组被驱逐。

使用场景

尽管使用 EXEC 功能也可以登录容器并进行调试，但由于镜像体积的原因，许多容器镜像中并未包含所需的调试工具（如 bash、net-tools 等）。相比之下，预装调试工具的 Debug 功能更适合以下场景。

- 故障排查：如果业务容器出现问题，除了查看事件和日志，您可能还需要在容器内进行更详细的故障排查和处理。

- 配置调优：如果当前的业务解决方案存在缺陷，您可能希望在容器内对业务组件进行配置调优，以制定出更加有效的配置方案，帮助业务更好地运行。

操作步骤

1. 进入 **Container Platform**。
2. 在左侧导航栏中，单击 工作负载 > 容器组。
3. 找到容器组，单击⋮> **Debug**。
4. 选择要调试的容器。
5. （可选）如果界面提示 需要初始化，请单击 初始化。

   说明：初始化 Debug 功能后，只要容器组未重建，您便可直接进入临时容器（例如 *容器 A-debug*）进行调试。
6. 等待调试窗口准备好后，开始调试。

   提示：单击右上角的 命令查询 可查看常用工具及其使用方法。
7. 完成操作后，关闭调试窗口。

使用 **Exec** 进入容器

目录

通过应用程序进入容器

您可以使用 `kubectl exec` 命令进入容器的内部实例，从而在 Web 控制台窗口中执行命令行操作。此外，您可以轻松地使用文件传输功能在容器内上传和下载文件。

前提条件

- 容器必须正常运行。
- 使用文件传输功能时，容器中必须存在 `tar` 工具，并且容器的操作系统不能是 Windows。

操作步骤

1. 进入 **Container Platform**。

2. 在左侧导航栏中，点击 应用 > 应用程序。

3. 点击 *应用名称*。

4. 找到工作负载并点击 **EXEC** > ***Pod** 名称*。

5. 输入您希望执行的命令。

6. 点击 确定，进入 Web 控制台窗口并执行命令行操作。

7. 点击 文件传输。输入 上传路径 将文件上传到容器进行测试；或者输入 下载路径 从容器下载日志和其他文件到本地以进行分析。

通过 Pod 进入容器

您可以使用 `kubectl exec` 命令进入容器的内部实例，从而在 Web 控制台窗口中执行命令行操作。此外，您可以轻松地使用文件传输功能在容器内上传和下载文件。

前提条件

- 容器必须正常运行。
- 使用文件传输功能时，容器中必须存在 `tar` 工具，并且容器的操作系统不能是 Windows。

操作步骤

1. 在左侧导航栏中，点击 工作负载 > **Pods**。

2. 点击⋮> **EXEC** > *容器名称*。

3. 输入您希望执行的命令。

4. 点击 确定，进入 Web 控制台窗口并执行命令行操作。

5. 点击 文件传输。输入 上传路径 将文件上传到容器进行测试；或者输入 下载路径 从容器下载日志和其他文件到本地以进行分析。

使用指南

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。

镜像仓库

介绍

介绍

安装

通过 YAML 安装

通过 Web UI 安装

使用指南

Common CLI Command Operat

- 登录 Registry
- 为用户添加命名空间权限
- 为服务账户添加命名空间权限
- 拉取镜像
- 推送镜像

Using Alauda Container Platform Registry in K

- 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，极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，镜像仓库按照命名空间进行组织。这意味着每个命名空间都可以被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相互隔离，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（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身份验证（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，通常会利用现有的平台登录会话或 ServiceAccount Token 实现透明认证。

授权

授权控制在命名空间级别实施。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），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，提升安全性，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。
- 灵活访问： 支持集群内及外部访问模式。
- 安全保障： 提供 RBAC 授权和镜像扫描能力。

- 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： 为边缘集群提供自主管理的专用镜像仓库。
- 资源优化： 在基础设施利用率不足时，通过集成的 Source to Image (S2I) 方案展示完整工作流能力。

安装

通过 YAML 安装

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。
- 需要完全自定义 **YAML** 以实现高级配置。

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。
- 具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（例如 NFS、GlusterFS 等）。
- （可选）提供有效的 **S3** 存储（例如 AWS S3、Ceph 等）。如果没有现成的 S3 存储，可在集群内部署 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
      resources:
        limits:
          cpu: 500m
          memory: 512Mi
        requests:
          cpu: 250m
          memory: 256Mi
    ingress:
      enabled: true
      hosts:
        - name: <YOUR-DOMAIN>   # [必填] 自定义域名
          tlsCert: <NAMESPACE>/<TLS-SECRET>  # [必填] 命名空间/Secret 名称
      ingressClassName: "<INGRESS-CLASS-NAME>"  # [必填] IngressClassNam
e
      insecure: false
    persistence:
      accessMode: ReadWriteMany
      nodes: ""
      path: <YOUR-HOSTPATH>  # [必填] LocalVolume 本地路径
      size: <STORAGE-SIZE>  # [必填] 存储大小（例如 10Gi）
      storageClass: <STORAGE-CLASS-NAME>  # [必填] StorageClass 名称
      type: StorageClass
```

```
    s3storage:
      bucket: <S3-BUCKET-NAME>              # [必填] S3 桶名称
      enabled: false                         # 本地存储设置为 false
      env:
        REGISTRY_STORAGE_S3_SKIPVERIFY: false  # 自签名证书设置为 true
      region: <S3-REGION>                      # S3 区域
      regionEndpoint: <S3-ENDPOINT>  # S3 端点
      secretName: <S3-CREDENTIALS-SECRET>             # S3 凭据 Secret
    service:
      nodePort: ""
      type: ClusterIP
  pluginName: internal-docker-registry
```

2. 根据环境自定义以下字段：

```
spec:
  config:
    ingress:
      hosts:
        - name: "<YOUR-DOMAIN>"                # 例如 registry.your-comp
any.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-stor
age
    s3storage:
      bucket: "<S3-BUCKET-NAME>"                # 例如 prod-registry
      region: "<S3-REGION>"                    # 例如 us-west-1
      regionEndpoint: "<S3-ENDPOINT>"          # 例如 https://s3.amazona
ws.com
      secretName: "<S3-CREDENTIALS-SECRET>"     # 包含 AWS_ACCESS_KEY_I
D/AWS_SECRET_ACCESS_KEY 的 Secret
      env:
        REGISTRY_STORAGE_S3_SKIPVERIFY: "true"  # 自签名证书设置为 "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.ingress.hosts[0].name`	用于访问 registry 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 的 Ingress 类名	`cluster-alb-1`
`spec.config.persistence.size`	registry 的存储大小	`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 集群执行以下命令：

```
# <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=<CLUS
TER-NAME>,cpaas.io/module-name=internal-docker-registry -o name)
```

卸载

在 global 集群执行以下命令：

```
# <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。
- 非生产环境中的快速概念验证部署。
- 具备有限 **Kubernetes** 经验的团队，寻求简化的部署流程。
- 需要最小化定制的场景（例如，默认存储配置）。
- 基础网络配置，无特定 ingress 规则需求。
- 需要配置**StorageClass**以实现高可用性。

不推荐使用场景：

- 生产环境中需要高级存储（如 S3 存储）配置。

- 需要特定 ingress 规则的网络配置。

前提条件

- 使用Cluster Plugin机制，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	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：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。

使用指南

Common CLI Command Operat

- 登录 Registry
- 为用户添加命名空间权限
- 为服务账户添加命名空间权限
- 拉取镜像
- 推送镜像

Using Alauda Container Platform Registry in K

- Registry Access Guidelines
- Deploy Sample Application
- Cross-Namespace Access
- Best Practices
- Verification Checklist
- Troubleshooting

Common CLI Command Operations

Alauda Container Platform 提供命令行工具，供用户与 Alauda Container Platform Registry 交互。以下是一些常用操作和命令示例：

假设集群的 Alauda Container Platform Registry 服务地址为 registry.cluster.local，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-pull
er --user=<username> -n <namespace>
```

为用户添加命名空间推送权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-push
er --user=<username> -n <namespace>
```

为服务账户添加命名空间权限

为服务账户添加命名空间拉取权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-pull
er --serviceaccount=<namespace>:<serviceaccount-name> -n <namespace>
```

为服务账户添加命名空间推送权限。

```
kubectl create rolebinding <binding-name> --clusterrole=system:image-push
er --serviceaccount=<namespace>:<serviceaccount-name> -n <namespace>
```

拉取镜像

从 Registry 拉取镜像到集群内部（例如用于 Pod 部署）。

```
# 从当前命名空间（my-ns）的 Registry 拉取名为 my-app，标签为 latest 的镜像
kubectl acp pull registry.cluster.local/my-ns/my-app:latest

# 从其他命名空间（例如 shared-ns）拉取镜像（需要拥有 shared-ns 命名空间的拉取权限）
kubectl acp pull registry.cluster.local/shared-ns/base-image:latest
```

该命令会验证您在目标命名空间的身份和拉取权限，然后从 Registry 拉取镜像。

推送镜像

将本地构建的镜像或从其他地方拉取的镜像推送到 Registry 中的指定命名空间。

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 命令将镜像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。

```
# 假设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.l
ocal/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至 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：外部入口域名（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. 每个命名空间中的 默认 ServiceAccount 会自动配置 imagePullSecret，用于访问 `internal-docker-registry.cpaas-system.svc` 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` 的用户拉取 `shared-ns` 的镜像，`shared-ns` 的管理员可以创建角色绑定以授予必要权限。

Example Role Binding

```
# 访问共享命名空间的镜像（需要权限）
kubectl create rolebinding cross-ns-pull \
  --clusterrole=system:image-puller \
  --serviceaccount=my-ns:default \
  -n shared-ns
```

Best Practices

- 注册表使用：部署时始终使用 `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 ServiceAccount 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 是否正确配置。
- 权限拒绝：确保用户或 ServiceAccount 在目标命名空间拥有必要的角色绑定。
- 网络问题：验证网络策略和服务配置，确保能连接到内部注册表。
- **DNS** 解析失败：检查节点上的 `/etc/hosts` 文件内容，确保 `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-reg
    istry.cpaas-system.svc internal-docker-registry.cpaas-system.svc.clus
    ter.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}'
    ```

源代码到镜像

介绍

介绍

Source to Image 概念

核心功能

核心优势

应用场景

使用限制

安装

安装 Alauda Container Platform Builds

前提条件

操作步骤

架构

架构

功能指南

管理通过代码创建的应用

主要功能

优势

前提条件

操作步骤

相关操作

How To

通过代码创建应用

前提条件

操作步骤

介绍

Alauda Container Platform Builds 是由 灵雀云容器平台 提供的云原生容器工具，集成了 Source to Image (S2I) 能力与自动化流水线。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，加速企业云原生之旅。此外，Alauda Container Platform Builds 提供可视化的发布管理，并与 Kubernetes 原生工具如 Helm 和 GitOps 实现无缝集成，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。它将应用的源代码注入到预定义的构建镜像中，并自动完成编译、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，而无需担心容器化的细节。

核心功能

Alauda Container Platform Builds 促进了从代码到应用的全栈云原生工作流，支持多语言构建和可视化发布管理。它利用 Kubernetes 原生能力将源代码转换为可运行的容器镜像，确保无缝集成到完整的云原生平台中。

- 多语言构建：支持 Java、Go、Python 和 Node.js 等多种编程语言的应用构建，满足多样化的开发需求。
- 可视化界面：提供直观的界面，方便您轻松创建、配置和管理构建任务，无需深厚的技术背景。
- 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、部署和运维的自动化管理。
- 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 自动化镜像构建与部署流程，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：Tekton Operator 现已在集群 OperatorHub 中可用。

安装

安装 Alauda Container Platform Builds

前提条件

操作步骤

安装 Alauda Container Platform Builds

目录

前提条件

> Alauda Container Platform Builds 是由 灵雀云容器平台 提供的一款容器工具，集成了构建（支持 Source to Image）和应用创建。

1. 下载与您的平台匹配的 **Alauda Container Platform Builds** 最新版本安装包。如果 Kubernetes 集群中尚未安装 **Tekton** Operator，建议一起下载。
2. 利用 `violet` CLI 工具将 **Alauda Container Platform Builds** 包和 **Tekton** 包上传至目标集群。有关使用 `violet` 的详细说明，请参阅 CLI。

操作步骤

安装 Alauda Container Platform Builds Operator

1. 登录平台，导航至 平台管理 页面。
2. 点击 市场 > **OperatorHub**。
3. 找到 **Alauda Container Platform Builds** Operator，点击 安装，并进入 安装 页面。

配置参数：

参数	推荐配置
频道	`Alpha`：默认频道设置为 **alpha**。
版本	请选取最新版本。
安装模式	`集群`：一个 Operator 在整个集群的所有命名空间中共享，以便进行实例创建和管理，从而降低资源占用。
命名空间	`推荐`：建议使用 **shipyard-operator** 命名空间；如果不存在，将自动创建。
升级策略	请选取 `手动`。 • `手动`：当 OperatorHub 中有新版本时， • 升级 操作不会自动执行。

4. 在 安装 页面，选择默认配置，点击 安装，完成 **Alauda Container Platform Builds** Operator 的安装。

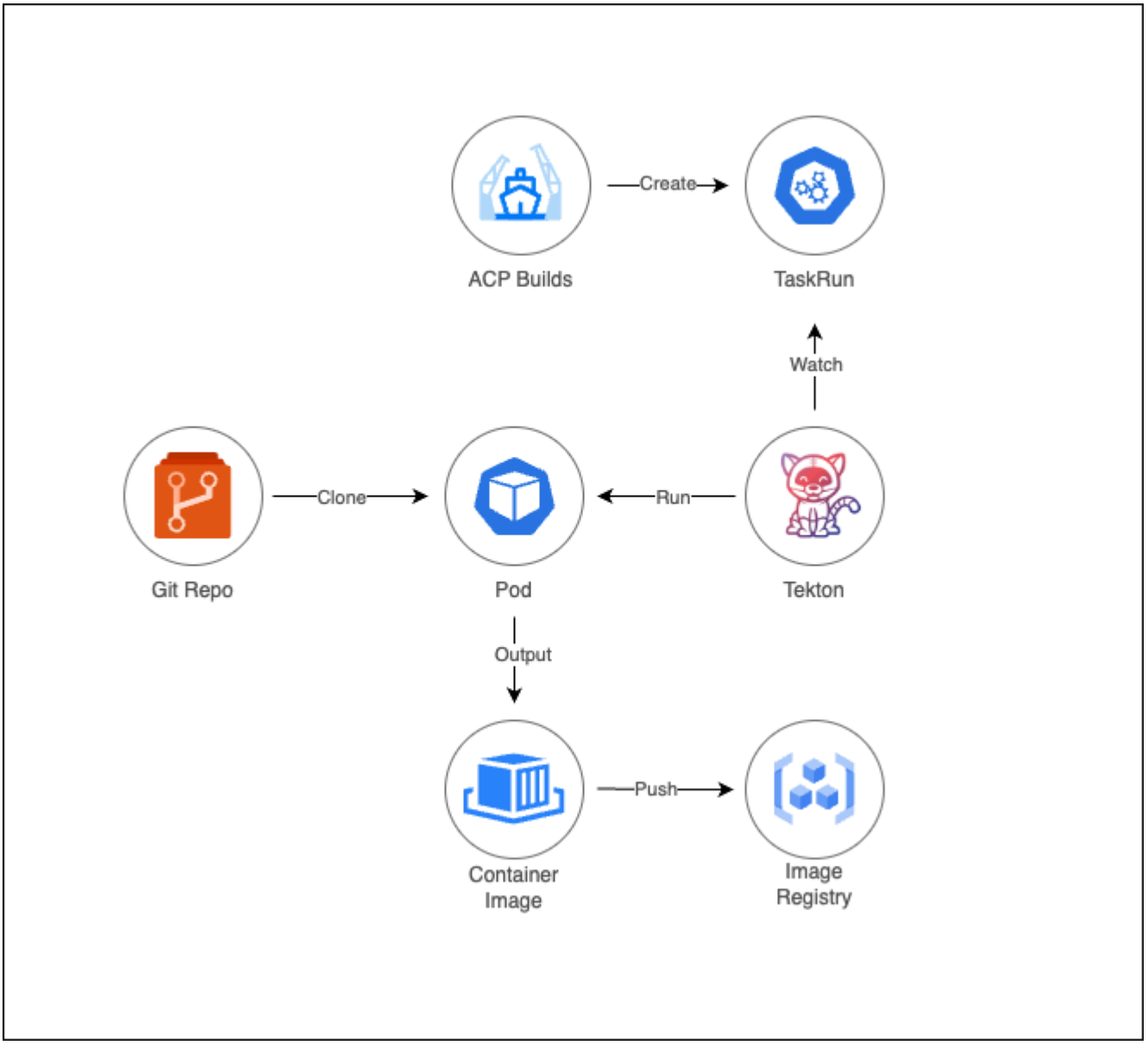
安装 **Shipyard** 实例

1. 点击 市场 > **OperatorHub**。
2. 找到已安装的 **Alauda Container Platform Builds** Operator，导航至 所有实例。
3. 点击 创建实例 按钮，在资源区域点击 **Shipyard** 卡片。
4. 在实例参数配置页面，除非有特定要求，您可以使用默认配置。
5. 点击 创建。

验证

- 实例成功创建后，预计等待 "20 分钟" 后切换至 **Container Platform** > 应用 > 应用 并点击创建。
- 您应该能看到 通过代码创建 的入口；此时，Alauda Container Platform Builds 的安装已成功，您可以通过 通过代码创建应用 开始您的 S2I 之旅。

架构



源到镜像 (S2I) 能力通过 **Alauda Container Platform Builds** 操作符实现，允许通过 Git 仓库源代码自动生成容器镜像并随后推送到指定的镜像注册表。核心组件包括：

- **Alauda Container Platform Builds** 操作符：管理端到端构建生命周期并协调 Tekton 管道。

- **Tekton** 管道：通过 Kubernetes 原生的 `TaskRun` 资源执行 S2I 工作流程。

功能指南

管理通过代码创建的应用

管理通过代码创建的应用

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、分析和排查问题。

前提条件

- 已完成安装 Alauda Container Platform Builds。
- 需要访问镜像仓库；如无权限，请联系管理员进行安装 Alauda Container Platform Registry。

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。

地域	参数	说明
代码仓库	类型	• 平台集成：选择已集成到平台且已分配给当前项目的代码仓库；平台支持 GitLab、GitHub 和 Bitbucket。 • 输入：使用未集成到平台的代码仓库 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流程，使开发人员专注于代码开发。整体流程如下： 1. 安装 Alauda Container Platform Builds 并创建 Shipyard 实例后，系统自动生成集群级资源，如 ClusterBuildStrategy，定义标准化构建流程，包括详细构建步骤和必要构建参数，从而支持 Source-to-Image (S2I) 构建。详情请参见：安装 Alauda Container Platform Builds 2. 根据上述策略和表单信息创建 Build 类型资源，指定构建策略、构建参数、源代码仓库、输出镜像仓库等相关信息。 3. 创建 BuildRun 类型资源以启动具体构建实例，协调整个构建过程。 4. 创建 BuildRun 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 执行构建流程。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。 执行构建过程。
	镜像 **URL**	构建完成后，指定应用的目标镜像仓库地址。
应用	-	根据需要填写应用配置，具体请参见从镜像创建应用文档中的参数说明。
网络	-	• 目标端口：容器内应用实际监听的端口。启用外部访问时，所有匹配流量将转发至该端口以提供外部服务。 • 其他参数：请参见配置 Ingress 文档中的参数说明。
标签注解	-	根据需要填写相关标签和注解。

5. 填写完参数后，点击 **Create**。

6. 可在 **Details** 页面查看对应部署信息。

相关操作

构建

应用创建完成后，可在详情页查看对应信息。

参数	说明
Build	点击链接查看具体的构建（Build）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。

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过程，最终使应用能够在 Kubernetes 上以容器化方式高效运行。

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 Alauda Container Platform 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. 填写参数后，点击 **Create**。

6. 可在 **Details** 页面查看对应应用状态。

节点隔离策略

节点隔离策略提供了一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

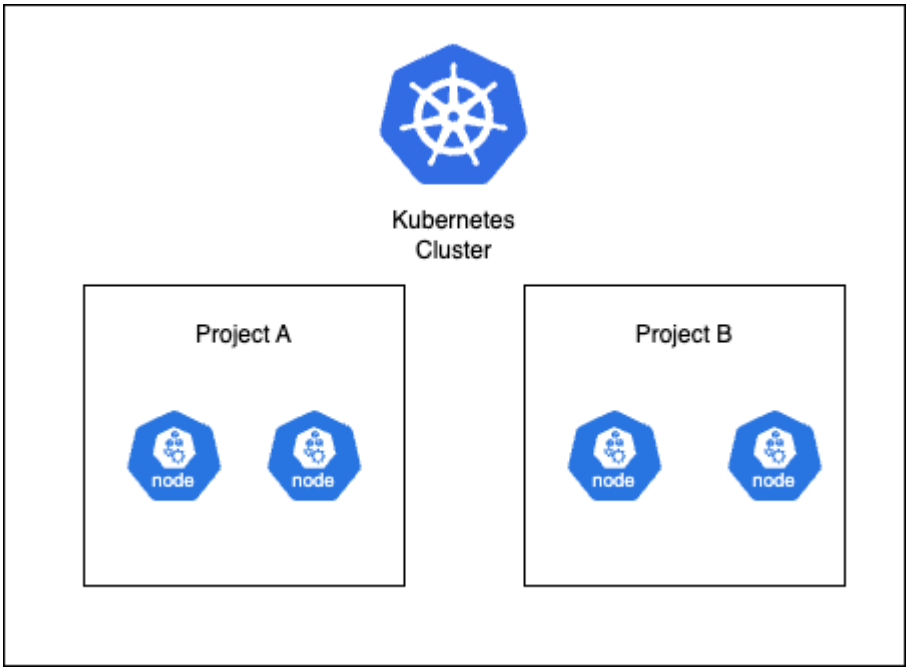
优势

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。

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。

功能指南

创建节点隔离策略

创建节点隔离策略

为当前集群创建节点隔离策略，允许指定项目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。

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 中针对每个配额项选择最小值进行合并，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` 之外的所有 ResourceQuota。这有助于避免因多个 ResourceQuota 导致配额计算复杂化，从而容易引发错误。

为什么导入命名空间时不应存在多个 **LimitRange** 或 **LimitRange** 名称不是 `default` ？

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。

平台在创建命名空间时会创建一个名为 `default` 的 LimitRange。因此，导入命名空间前，该命名空间中应仅存在一个名为 `default` 的 LimitRange。