

容器管理

容器管理是基于 Kubernetes 构建的面向云原生应用的容器化管理模块。它是 DCE 5.0 的核心， 基于原生多集群架构，解耦底层基础设施，实现多云与多集群统一化管理，大幅简化企业的应用上云流程，有效降低运维管理和人力成本。通过容器管理，您可以便捷创建 Kubernetes 集群，快速搭建企业级的容器云管理平台。

!!! tip

容器化是数据中心发展的不可或缺的趋势，它是应用封装、部署和托管的自然延伸。

- :material-server:{ .lg .middle } **集群管理**
-

DCE 5.0 容器管理目前支持四种集群：全局服务集群、管理集群、工作集群、接入集群。

- [创建集群或接入集群](#)
- [升级集群](#)
- [集群角色和集群状态](#)
- [选择运行时](#)

- :fontawesome-brands-node:{ .lg .middle } **节点管理**
-

节点上运行了 kubelet、容器运行时以及 kube-proxy 等组件。

- [节点调度](#)
- [标签与注解，污点管理](#)
- [节点扩容/缩容](#)
- [节点认证、节点可用性检查](#)

- :simple-myspace:{ .lg .middle } **命名空间**

命名空间是构建虚拟空间隔离物理资源的一种方式，仅作用于带有命名空间的对象。

- [创建/删除命名空间](#)
 - [命名空间独享节点](#)
 - [为命名空间配置 Pod 安全策略](#)
- : octicons-tasklist-16:{ .lg .middle } **工作负载**
-

工作负载是在 DCE 5.0 上所运行的各类应用程序。

- 创建 [Deployment](#) 和 [StatefulSet](#)
 - 创建 [DaemonSet](#)
 - 创建 [Job](#) 和 [CronJob](#)
- : material-expand-all:{ .lg .middle } **弹性伸缩**
-

通过配置 HPA、VPA 策略实现工作负载的弹性伸缩。

- 安装 [metrics-server](#)、[kubernetes-cronhpa-controller](#) 和 [VPA](#) 插件
 - [创建 HPA 策略](#)
 - [创建 VPA 策略](#)
- : simple-helm:{ .lg .middle } **Helm 应用**
-

Helm 是 DCE 5.0 的包管理工具，提供了数百个 Helm 模板，方便用户快速部署应用。

- [Helm 模板](#)
- [Helm 应用](#)
- [Helm 仓库](#)

- : material-dot-net:{ .lg .middle } 容器网络

DCE 5.0 自带的容器网络便于对外提供服务，通过 Ingress 定义路由规则，根据网络策略控制流量。

- [服务 Service](#) : ClusterIP、NodePort、LoadBalancer
- [路由 Ingress](#)
- [网络策略 NetworkPolicy](#)

- : material-hddisk:{ .lg .middle } 容器存储

DCE 5.0 容器管理奉行 Kubernetes 的容器化存储理念，支持原生 CSI，能够制备动态卷、卷快照、克隆等。

- [数据卷声明 PVC](#)
- [数据卷 Volume](#)
- [存储池 StorageClass](#)

- : material-security:{ .lg .middle } 安全管理

DCE 5.0 容器管理支持对节点、集群执行三种扫描：

- [合规性扫描](#)
- [权限扫描](#)
- [漏洞扫描](#)

- : material-key:{ .lg .middle } 配置与密钥

DCE 5.0 容器管理支持以键值对的形式管理 ConfigMap 和 Secret：

- [配置项 ConfigMap](#)

- [密钥 Secret](#)

- :material-card-search:{ .lg .middle } **集群巡检**
-

集群巡检可以自动/手动定期或随时检查集群的整体健康状态，让管理员获得保障集群安全的主动权。

- [创建巡检配置](#)
- [执行巡检](#)
- [查看巡检模块](#)

- :material-auto-fix:{ .lg .middle } **集群运维**
-

集群运维指的是查看集群的操作、升级集群以及集群配置等。

- [查看最近操作记录](#)
- [集群设置](#)
- [集群升级](#)

!!! success

通过容器化，您可以更快速、简单地开发和部署应用程序，比构建虚拟设备更加高效。

容器化架构带来了令人瞩目的运维和经济效益，包括更低的许可成本、更高的物理资源利用率、更好的可扩展性以及更高的服务可靠性。

展望未来，容器虚拟化将帮助企业更好地利用混合云和多云环境，实现更优化的资源管理和应用部署。

DCE 5.0 容器管理的逻辑架构

逻辑架构图

逻辑架构图

[下载 DCE 5.0](#) [安装 DCE 5.0](#) [申请社区免费体验](#)

容器管理功能特性

容器管理的主要功能特性如下：

类别	功能描述
集群全生命周期管理	<p>集群的统一纳管： - 支持所有特定版本范围内的任意 Kubernetes 集群纳入容器管理范围 - 实现云上、云下、多云、混合云容器云平台的统一管理</p> <p>集群的快速创建： - 基于 DaoCloud 自主开源项目 Kubebean - 支持接入/创建集群 - 支持创建集群时指定运行时类型</p> <p>一键式集群升级：一键升级自建容器云平台的 Kubernetes 版本，统一管理系统组件升级。</p> <p>集群高可用：内置集群容灾、备份能力，保障业务系统在主机故障、机房中断、自然灾害等情况下可恢复，提高生产环境的稳定性，降低业务中断风险。</p> <p>节点管理：支持自建集群增删节点，保障集群能够满足业务需求。</p>
应用负载	<p>应用全生命周期管理：支持基于 Kubernetes 原生的工作负载类型部署和管理能力，包括创建、配置、监控、扩容、升级、回滚、删除等全生命周期管理。</p> <p>一站式应用负载创建：解耦底层 Kubernetes 平台，一站式创建和运维业务工作负载。</p>

类别	功能描述
	<p>跨集群应用负载管理：跨集群负载统一管理、高效检索能力。</p>
	<p>应用负载的扩缩容：通过界面可实现应用负载的手动/自动扩缩容能力，自定义扩缩容策略，应对流量高峰。</p>
	<p>容器的生命周期设置：支持创建工作负载时设置回调函数、启动后参数、停止前参数，满足特定场景需求。</p>
	<p>容器就绪检查、存活检查设置： - 工作负载就绪检查：用于检测用户业务是否就绪，如未就绪，则不转发流量至当前实例。 - 工作负载存活检查：用户检测容器是否正常，如异常，集群会执行容器重启操作。</p>
	<p>容器的环境变量设置：可为业务容器运行环境设定指定环境变量。</p>
	<p>容器的自动调度：支持应用服务调度管理，自动按照主机资源用量进行容器调度，可以指定具体的部署主机，以及通过标签策略进行容器调度。</p>
	<p>亲和性和反亲和性：支持定义容器组间调度的亲和性和反亲和性；以及容器组与节点间的亲和性和反亲和性，满足业务自定义调度需求。</p>
	<p>容器安全用户设置：支持设定容器运行用户，如以</p>

类别	功能描述
	<p>Root 权限运行则填写 Root 用户 ID 0。</p>
	<p>自定义资源 (CRD) 支持：支持自定义资源的创建、配置、删除等全生命周期管理。</p>
服务与路由	<p>服务 (Service) 是 Kuberetes 原生资源，提供云原生的负载均衡能力，通过固定的 IP 地址和端口进行访问。目前支持的 Service 类型包括：</p> <p>集群内访问 (ClusterIP)：仅在集群内访问服务。</p> <p>节点访问 (NodePort)：使用节点 IP + 服务端口的方式进行访问。</p>
命名空间管理	<p>命名空间管理：支持命名空间创建、配额设置、资源限制设置等。</p> <p>跨集群命名空间管理：支持跨集群命名空间统一管理及高效检索能力，实现多云场景、灾备场景下命名空间的管理能力。</p>
容器存储	<p>数据卷管理：支持本地存储、文件存储、块存储，通过 CSI 能力接入，并提供给应用工作负载使用。</p> <p>数据卷动态创建：可支持存储池实现动态创建数据卷。</p>
策略管理	<p>策略统一管理和下发：支持以命名空间或集群粒度制定网络策略、配额策略、资源限制策略、灾备策略、安全策略等策略，并支持以集群/命名空间为粒度进行</p>

类别	功能描述
	<p>策略下发。</p>
	<p>网络策略：支持以命名空间或集群粒度制定网络策略，限定容器组与网络平上网络“实体”通信规则。</p>
	<p>配额策略：支持以命名空间或集群粒度设定配额策略，限制集群内的命名空间的资源使用。</p>
	<p>资源限制策略：支持以命名空间或集群粒度设定资源限制策略，约束对应命名空间内应用使用资源限制。</p>
	<p>灾备策略：支持以命名空间或集群粒度设定灾备策略，实现以命名空间为维度进行容灾备份，保障集群的安全性。</p>
	<p>安全策略：支持以命名空间或集群粒度设定安全策略，为 Pod 定义不同的隔离级别。</p>
扩展插件	<p>提供了丰富的系统插件，扩充云容器集群功能。扩展插件包括：DNS、HPA 等。</p>
权限管理	<p>支持命名空间授权，通过权限设置可以让不同的用户或用户组拥有操作指定命名空间下不同的 Kubernetes 资源的权限。</p>
集群运维	<p>全方位集群监控：全方位覆盖集群、节点的指标监控及告警，实时了解和查看集群和节点状态，及时实施运维措施，保障业务连续性。</p>
	<p>开放式 API：提供原生的 Kubernetes OpenAPI 能力。</p>

类别	功能描述
	<p>CloudShell 访问集群：支持通过 CloudShell 连接集群 并通过 Kubectl 访问集群。</p>
异构加速	<p>支持 Nvidia、天数智芯、昇腾等异构 GPU 硬件加速。 支持物理 GPU 单卡资源算力、显存、切分功能。多个服务容器可共享单张 GPU 卡，并支持限制和隔离每个服务容器所占用的 GPU 算力、显存额度。 支持按照项目对 GPU 资源进行配额管理。</p>

适用场景和优势

适用场景

应用的多云部署及跨云容灾

多云组合部署

根据不同性价比的云平台组合，结合不同应用场景，进行多云组合，降低总体成本。对于金融等安全性要求高的行业用户，基于业务数据的安全性和敏感性要求，将部分业务应用部署在私有云环境中，而将非敏感性应用部署在云上集群中，并进行统一管理。

场景一

场景一

跨云容灾备份

为保证业务高可用，同时将业务部署在不同地域的多个云容器平台上，帮助应用实现多地域的流量分发，并且实现跨云应用在同一平台管理，减少运维成本。另外当某个云容器平

台发生故障时，通过统一流量分发的机制，自动的将业务流量切换到其他云容器平台上。

场景二

场景二

跨云集群统一管理

跨云集群统一纳管

无论基于不同基础设施环境（公有云，私有云和混合云），或者不同容器云厂商搭建的 Kubernetes 平台，容器管理模块可统一纳管。

降低因不同基础设施、不同云提供商带来的额外管理成本，统一管理流程，降低成本。

跨云集群统一运维

不同的云容器提供商基于不同的 Kubernetes 发行版为客户提供不同云平台监控，实现统一平台监控运维难度大，无法便捷地了解整体运行状况，掌控故障处理进程和结果，基于容器管理模块，可基于集群的统一纳管实现一体化监控服务，实现多维度、跨云的统一监控运维。

场景三

场景三

弹性扩缩应对流量高峰

集群的弹性扩缩容

应对企业客户应对流量高峰，当业务应用部署所在 Kubernetes 容器平台资源已经达到阈值，需要结合底层 IaaS 弹性扩缩容能力，自动弹性扩缩容集群节点，从而应对超大流量的业务高峰，并在业务低谷期进行缩容，节约成本。

应用的弹性扩缩容

电商客户遇到双 11 等促销、限时秒杀等活动中，业务访问量迅速上升，需及时扩容应用计算资源，并根据用户设定的扩缩容策略自动调整，业务容器组数量随业务访问量上升而增加，随业务访问量下降而减少。

优势

容器管理模块具有如下优势：

集群的统一管理

- 支持不同集群统一纳管，支持所有特定版本范围内的任意 Kubernetes 集群纳入容器管理范围，实现云上、云下、多云、混合云容器云平台的统一管理，规避厂商锁定。
- 可通过 Web 界面一键式完成 Kubernetes 集群的平滑升级。
- 统一 Web 界面实现集群创建、集群节点的扩缩容等管理能力。
- 支持超大规模集群的统一纳管。

应用生产就绪

- 一站式应用分发，可通过镜像、YAML、Helm 进行应用分发，并实现跨云/跨集群的一体化管理。
- 应用高可用，支持应用的分布式部署，支持单点故障流量的自动切换。
- 丰富的监控指标，实现应用的全方位监控，提前预警应用流量高峰，应用故障。

策略的统一下发

- 支持以命名空间或集群粒度制定网络策略、配额策略、资源限制策略、灾备策略、安全策略等策略，并支持以集群/命名空间为粒度进行策略下发。

安全可靠

- 自建集群默认为高可用部署模式，保障您的业务高可用。当节点出现故障、自然灾害时，应用可持续运行，保障生产环境的高可用，实现业务应用系统不中断。
- 跨地域应用高可用，支持跨地域部署不同的容器集群，能够同时将业务部署在不同地区的多个云容器集群上，帮助应用实现多地域的流量分发。当某个云容器集群发生故障、机房宕机或者自然灾害时，通过统一流量分发的机制，自动将业务流量切换到其他云容器平台上，保障应用的高可用。
- 完善的用户权限体系，整合 [Kubernetes RBAC 权限体系](#)，支持为不同用户设置不同粒度权限。

异构兼容

- 提供高度自动化和具备弹性的异构多云支撑能力，适配 x86 以及信创云体系架构。
- 支持 x86、ARM 架构混合集群统一部署，统一管理和支撑应用运行，保障应用间网络互通。

开放兼容

- 基于原生的 Kubernetes 和 Docker 技术，完全兼容 Kubernetes API 和 Kubectl 命令。
- 提供了丰富的插件体系，扩充云容器集群功能，例如网络插件 [Multus](#)、[Cilium](#)、[Contour](#) 等组件。

基本概念

容器管理相关的基本概念如下。

概念	描述
----	----

概念	描述
集群 (Cluster)	集群指容器运行所需要的云资源组合，关联了若干云服务器节点。可以创建若干集群或接入若干 Kubernetes 标准集群。
节点 (Node)	每一个节点对应一台虚拟机/物理服务器，所有容器应用运行在集群的节点上。节点类型分为控制器节点 (Controller) 及工作节点 (Worker)。
容器组 (Pod)	Pod 是 Kubernetes 部署应用或服务的最小基本单位，可以封装一个或多个应用容器、存储资源、一个独立的网络 IP。
容器 (Container)	容器是通过容器镜像部署的实例，将应用程序从底层的主机设施中解耦，便于在不同的云或 OS 环境中部署。
工作负载 (Workload)	在 Kubernetes 上运行的应用程序，包括无状态服务、状态服务、守护进程服务、普通任务和定时任务等。
应用模板	标准模板的统一资源管理和调度，支持管理和部署社区标准应用模板及自定义业务应用模板。
镜像 (Image)	容器镜像是一个容器应用打包的标准格式模板，用于创建容器，包含程序、库、资源、配置等文件。
命名空间 (Namespace)	对一组资源和对象的抽象整合，不同命名空间中

概念	描述
	的数据彼此隔离。
服务 (Service)	将运行在一组 Pod 上的应用程序公开为网络服务的抽象方法，支持 ClusterIP、NodePort 和 LoadBalancer 等类型。
七层负载均衡 (Ingress)	为进入集群的请求提供路由规则的集合，支持 URL、负载均衡、SSL 终止等功能。
网络策略 (NetworkPolicy)	提供基于策略的网络控制，用于隔离应用并减少攻击面。
配置项 (ConfigMap)	保存配置非机密性的数据到键值对中，容器组可以将其用作环境变量、命令行参数或配置文件。
密钥 (Secret)	用于保存机密数据的配置信息，如密码、Token、密钥等。
标签 (Label)	一对 key/value 关联到对象上，用于标示对象的特点。
选择器 (LabelSelector)	Kubernetes 核心的分组机制，通过 Label Selector 识别一组有共同特征的资源对象。
注解 (Annotation)	将 Kubernetes 资源对象关联到任意的非标识性元数据，可以通过注解检索这些元数据。
存储卷 (PersistentVolume)	提供方便的持久化卷，PV 提供网络存储资源，PVC 申领存储资源。
存储声明 (PersistentVolumeClaim)	对 PV 的申领请求，类似于 Pod 消费 Node 资

概念	描述
	源。
弹性扩缩 (HPA)	Kubernetes 中实现 Pod 水平自动扩缩的功能。
亲和性与反亲和性	通过亲和性和反亲和性定义约束类型，实现就近部署和高可靠性。
节点亲和性 (NodeAffinity)	限制容器组被调度到特定的节点上。
节点反亲和性 (NodeAntiAffinity)	限制容器组不被调度到特定的节点上。
容器组负载亲和性 (PodAffinity)	指定工作负载部署在相同节点，减少网络消耗。
容器组反亲和性 (PodAntiAffinity)	指定工作负载部署在不同节点，减少宕机影响。
资源配额 (Resource Quota)	限制用户资源用量的机制。
资源限制 (Limit Range)	给命名空间增加资源限制，包括最小、最大和默认资源。
环境变量	容器运行环境中设定的变量，提供灵活性。

容器管理 Release Notes

本页列出容器管理的 Release Notes，便于您了解各版本的演进路径和特性变化。

*[kpanda]: DaoCloud 容器管理的内部开发代号

2025-01-31

v0.36

- **新增** 封装 addon 沐曦 metax-exporter：用于在集群环境中收集沐曦 GPU 设备指标

数据，安装后在可观测模块可进行图形化展示

- **新增** 封装 addon 沐曦 metax-extensions：提供 gpu-device 和 gpu-label 两个组件，为容器使用沐曦 GPU 提供必需的资源注册及分配能力
- **新增** 封装 addon 沐曦 metax-operator：提供全部组件，除了 metax-extensions 能力外，进一步云原生化，减轻集群中软件运行负担，降低运维难度
- **修复** 修复 PVC 列表制作快照 button 显示问题

2024-12-31

v0.35

- **优化** 除 cluster admin 之外，其他角色禁止访问 网络设置 模块
- **优化** 优化在没有空闲 Worker 的情况下，cloudshell 创建就绪耗时较长
- **修复** 修复 kpanda 相关 Pod 在生产环境部署没有设置 resource limits 问题
- **修复** 修复 cloudtty 控制台 history 信息泄露问题
- **修复** 修复使用密钥方式创建集群后，对应的密钥会丢失导致再次接入节点获取不到私钥信息问题
- **修复** 修复 Job 超时执行失败，但是 helmrelease 状态未更新问题

2024-11-30

v0.34.0

- **优化** etcd 备份 ListClusterSummary 集群列表页
- **修复** 控制台关闭后，cloudshell 在 VirtualService 中添加的路由配置没有清理的问题
- **修复** 集群创建失败，控制台一直重连，无法正常查看日志的问题

- **修复** cloudshell 资源到期没有及时清理的问题
- **修复** 备份恢复模块资源标签过滤为 `and` 并不使用选择器时备份失败的问题

2024-10-30

v0.33.0

新增

- **新增** 支持 寒武纪 GPU 卡的纳管和调度
- **新增** 实现 kubeconfig secret 资源删除的保护机制

优化

- **优化** 升级 hami、gpu-operator 的 addon 版本到 v2.4.1、v24.6.0
- **优化** kubeconfig secret 资源删除的保护机制
- **优化** helm push 插件集成到安装器离线包
- **优化** egress 端口分配，Cluster 的 egress 端口保持固定

修复

- **修复** 节点详情页的容器组页面根据 GPU 类型筛选不生效的问题
- **修复** metrics-server 插件卸载残留的问题
- **修复** 创建集群时自定义 yum_repos 重试安装时自定义的内容没有了的问题
- **修复** 当 worker 失联时删除 node 节点会导致 kpanda-controller-manager 出现 panic 的问题

2024-09-30**v0.32.0**

新增

- **新增** 支持 [Volcano Binpack](#) 和 [优先级抢占策略](#)
- **新增** 支持沐曦 GPU 卡的使用
- **新增** GPU 监控面板中增加 GPU 利用率指标
- **新增** 命名空间添加配额和使用量显示
- **新增** 支持平台下发的 kubeconfig 可以永久有效
- **新增** 自定义角色支持工作空间与命名空间之间的权限映射
- **新增** 通过快照创建 PVC，支持用户自己选择 StorageClass

优化

- **优化** GPU 整卡模式下可用 GPU 算力提示文字
- **优化** 命名空间绑定工作空间绑定后页面未显示绑定的工作空间，需要手动刷新页面问题
- **优化** 兼容 volumeMode 为 block 的 PV
- **优化** Addon 合并镜像架构后未清理中间镜像

修复

- **修复** vGPU 设置算力 100，但仪表盘 Pod 的算力使用率显示为 0% 的问题
- **修复** GPU Operator 未开启 Driver 选项导致不显示 GPU 模式切换功能问题

- **修复** 仪表盘 GPU 显示个数超过实际值问题
- **修复** MIG Mixed 模式下，节点详情页的 GPU 使用率数据显示有误
- **修复** 在 1000+ 集群中，修复大规模场景 Binding Syncer 内存占用过大的问题
- **修复** 在 1000+ 集群中，修复给每一个集群绑定工作空间时 Redis 的连接数突增且长时间不会关闭的问题
- **修复** 在 1000+ 集群中，修复大规模场景中出现的工作空间共享资源权限错乱的问题
- **修复** 集群绑定工作空间，没有出现审计日志的问题
- **修复** 安全管理中的 合规性扫描 的集群出现虚拟机集群的问题
- **修复** 卸载 Helm 应用，偶发出现 2 个卸载 Job 的问题
- **修复** 安装器 v0.19.0 升级至 v0.20.0 安装 metrics-server 失败的问题
- **修复** 创建集群时点击检测再取消，检测任务未取消，后续创建时点击检测都提示在检测中的问题
- **修复** 容器管理 -> 节点管理中，全局服务集群无法接入和移除节点的问题
- **修复** 卸载集群，一直在删除中，实际已卸载失败的问题
- **修复** 解除集群时 kpanda-system 下 finalizer 资源未被删除，导致命名空间 kpanda-system 无法删除的问题
- **修复** DCE 5.0 环境中已经做了 kubelet 向下兼容包，低版本集群应该支持卸载，但是并没有删除成功的问题
- **修复** 备份恢复触发时间显示有误的问题
- **修复** 集群巡检配置，定时任务小时数，触发时间与配置时间不符的问题

2024-08-30**v0.31.0**

新增

- **新增** 异构 GPU 卡，支持沐曦 GPU 卡
- **新增** 异构 GPU 卡，支持通过 gpu-operator 安装 redhat9.2 的驱动镜像
- **新增** 支持集群接入证书的有效期检查和提醒
- **新增** Helm 模板支持在界面上查看 Helm 内容及生成的编排文件

优化

- **优化** 增加切换节点 GPU 卡模式的审计日志
- **优化** NPU 面板布局
- **优化** GPU 资源界面文案
- **优化** gpu-operator 卸载之后，节点上的 GPU label 过了三分钟左右才不展示问题优化
- **优化** 提供 GPU 指标告警操作文档
- **优化** 单机多卡场景中一张 GPU 掉卡，在工作负载调度的时候 Pod 全处于“UnexpectedAdminisssonError”状态问题
- **优化** 对于 kubelet 创建的工作集群，采用 ServiceAccount Token 的认证方式访问子集群
- **优化** 移除 cluster_controller 中的 LeaseController
- **优化** kpanda-controller-manager 增加一些与业务相关的监控指标
- **优化** repo_controller 增加对 Repo 的同步/下载的业务监控指标

- 优化 cluster_setting_controller 增加对 插件 的同步监控指标
- 优化 移除 GPUSchedulerController 中的 informerFactory
- 优化 multi-controller 中的 Controller 支持指定开启或者关闭
- 优化 cluster_status_controller 逻辑引入 successThreshold 和 failureThreshold
- 优化 为 Controller 引入能够控制并发的参数
- 优化 统一日志输出级别
- 优化 控制器 重新入队时间
- 优化 binding-syner 中的 Controller 能够指定开启或者关闭
- 优化 自定义资源 CRD 的默认版本
- 优化 支持在服务列表通过服务名称、访问方式、访问端口搜索
- 优化 Kpanda 安装集群时对节点时间一致性的体验
- 优化 Helm 生命周期操作禁止并发
- 优化 Helm 安装增加 K8s 编排确认
- 优化 工作负载回滚支持展示版本详细信息

修复

- 修复 GPU driver Pod 对应进程的 GPU 显存使用和应用里查看的信息显示不一致的问题
- 修复 vGPU 模式的情况下，关机重启 vm 后，变成整卡模式的问题
- 修复 npu 开启虚拟化之后节点上的 label 丢失的问题
- 修复 GPU Pod 仪表盘 Pod GPU 算力使用率显示有误的问题
- 修复 GPU Pod 仪表盘中的 GPU Pod 显存使用量/使用率显示有误的问题

- **修复** 使用 Ascend 卡创建工作负载，监控指标没有数据展示的问题
- **修复** vGPU 修改 deviceSplitCount 未生效的问题
- **修复** vGPU 模式下，显存如果填写 5000，更新时页面不能回显的问题
- **修复** 集群设置菜单下的 Addon 插件 GPU 类型下拉框中未展示 mig 相关信息的问题
- **修复** 有 GPU 的节点重启或者驱动 driverpod 升级 vgpu-device-plugin 会持续 crash 的问题
- **修复** GPU 配额中 GPU 显存单位和内存配额单位不一致的问题
- **修复** 更新 Go dependency 失败，OTel 依赖冲突的问题
- **修复** cloudshell CRD 超时页面终端仍然提示可以重连的问题
- **修复** clusteradmin 权限的用户（无全局服务集群的权限），查看 **最近操作 -> 集群操作** 页面的立即前往，可以直接进入到全局服务集群的详情页的问题
- **修复** 使用 Namespace Admin 权限的用户查看工作负载详情页容器配置的基本信息，存在接口报 403 的问题
- **修复** NS 的 Quota 展示出来了，但是编辑界面的数据为空的问题
- **修复** 容器组详情页的镜像地址和工作负载详情页显示的镜像地址不一致的问题
- **修复** 更新 insight-ui 无状态负载，进入到 **容器配置 -> 健康检查** 页面，查看端口信息未正常展示的问题
- **修复** 节点 kubeProxy 版本显示异常的问题
- **修复** DCE5-0.19 接入节点或者创建集群时，对 Ubuntu 22.04 识别系统代号出错的问题
- **修复** Helm 创建应用状态处于未知状态时点击应用详情报错的问题
- **修复** 工作集群创建 Helm 应用时前端显示应用的初始状态为失败状态的问题

- **修复** 更新 Helm 应用时就绪等待配置未记录创建时的状态的问题
- **修复** 开启就绪等待接入子集群安装 Helm 应用失败时 Helm 应用状态始终处于安装中的问题

已知问题

- **修复** 容器管理模块的前端版本需保持一致，否则 vGPU 模式的情况下，关机重启 VM 后会变成整卡模式的问题

2024-07-31

v0.30.0

新增

- **新增** 支持通过 Addon 安装 Koordinator 插件，并完成在线和离线混部
- **新增** UI 生成 kubeconfig 时支持 7 天或者自定义日期选项
- **新增** Helm 模板支持在界面上查看 Helm 内容及生成的编排文件
- **新增** gpu-operator 离线化默认操作系统支持 CentOS 7、Ubuntu 22.04、Ubuntu 20.04
- **新增** 以文档方式支持华为昇腾 NPU 虚拟化
- **新增** NPU 监控面板支持中英文切换
- **新增** gpu-operator 支持开启 RDMA

优化

- **优化** Ingress 列表增加域名字段
- **优化** 支持全局服务集群的“集群厂商”名称 (DaoCloud Kubelet)

- 优化 Kpanda 性能
- 优化 kpanda GetClusterAdminKubeConfig 接口生成的证书只有一年有效期时过期无法正常使用的问题
- 优化 支持集群接入证书的有效期检查和提醒
- 优化 Helm charts 安装时获取安装配置的接口调用
- 优化 addon-pack charts 中镜像支持通过 –platform 参数指定拉取镜像架构
- 优化 Helm controller 解耦 cluster service 和 rbac service 等
- 优化 切换 GPU 模式体验，添加了切换状态
- 优化 nvidia-vgpu 和 gpu-operator 安装时，引导用户如何切换 GPU 模式
- 优化 当 MIG single 模式被识别成整卡且用户使用时存在误解的问题

修复

- 修复 全局服务集群默认未安装 metrics-server，导致各个模块创建的 HPA 无效的问题
- 修复 Kpanda 数据库连接的问题
- 修复 NS Admin 角色 PV/PVC 权限的问题
- 修复 修改集群的基础配置后，egress 端口被刷新，导致数据流中断，controller-manager 工作异常的问题
- 修复 在工作负载详情的访问方式页面执行重启工作负载操作，service 接口报 404 的问题
- 修复 addon 没有 cro-operator 对应的离线包的问题
- 修复 Ubuntu 内核自动更新升级时可能导致的系统在不经意间被重启的问题
- 修复 安装使用 MIG 模式，偶尔在节点上显示整卡模式的问题

- **修复** GPU 虚拟化后，内存超配功能不可用的问题
- **修复** 调整 GPU 调度策略偶发调度策略切换失败的问题

2024-06-30

v0.29.0

新增

- **新增** 接入集群的 K8s 发行版支持 k3s
- **新增** 支持 GPU 卡的状态监控，可通过 XDI 指标在可观测平台查看 GPU 卡状态
- **新增** 资源调度：支持 GPU 卡调度 Spread (集群维度)
- **新增** 资源调度：支持 GPU 卡调度 Binpack (集群维度)
- **新增** 资源调度：支持 GPU 节点调度 Spread (集群维度)
- **新增** 资源调度：支持 GPU 节点调度 Spread (工作负载维度)
- **新增** 资源调度：支持 GPU 节点调度 Binpack (工作负载维度)
- **新增** 资源调度：支持 GPU 卡调度 Spread (工作负载维度)
- **新增** 资源调度：支持 GPU 卡调度 Binpack (工作负载维度)

优化

- **优化** 下载站 addon 包支持多离线包自动化
- **优化** Helm 安装更加云原生化，使 controller 负责管理执行 Helm 操作的 job
- **优化** 节点详情页查看 GPU 资源分配情况的快捷链接，跳转到的监控页面未集成

GPU 配额问题

- 优化 gpu-operator 支持在同样的操作系统，不同的内核版本上安装驱动
- 优化 支持在集群维度 GPU 卡、节点级别的 binpack/spread 产品化支持级别的 binpack/spread
- 优化 configmap / secret 的编辑器支持左右移动
- 优化 更新模块时，选择版本支持筛选
- 优化 workspace admin 权限映射到容器管理中的 cluster admin 权限时页面无显示问题
- 优化 容器日志界面支持显示 1000 行

修复

- **修复** 被删除的用户依然显示在集群权限用户列表中的问题
- **修复** cloudtty Pod 无法使用带证书的 kubeconfig 访问子集群的问题
- **修复** 多架构融合，存在 addon 包里没有对应的镜像，导致服务不能正常运行的问题
- **修复** Deployment 实例列表显示了非当前 Deployment 的 Pod 的问题
- **修复** GPU node 仪表盘中的显存分配率和节点上的不一致的问题
- **修复** GPU 节点的标签显示有误的问题
- **修复** LoadBalancer 类型 service，做更新操作，修改 lb IP 地址后，查看 service 详情，会概率出现 nodeport 访问方式展示的问题
- **修复** 创建工作负载 mig 模式配置多类型 GPU 规格的问题
- **修复** MIG Mixed 模式下，Deployment 配置多个不同类型 GPU 规格配置，配置与实际不符的问题

参阅 [v0.29.0 及以上版本升级注意事项](#)。

2024-05-31**v0.28.0**

新增

- **新增** 支持 ws admin+cluster admin 在 kpanda 绑定集群/命名空间到工作空间
- **新增** 制作存储卷快照支持指定卷快照类
- **新增** 创建 workload 存储-临时路径增加存储容量修改
- **新增** 负载监控增加 NPU 相关指标
- **新增** 集群解除接入时，增加是否有拓展边缘单元实例的校验
- **新增** kpanda 的接入集群增加可选是否使用 egress
- **新增** addon 支持多离线包 (standard 离线包 / GPU 离线包)
- **新增** GPU 监控面板支持算力使用率指标
- **新增** workload 支持 NPU 相关指标
- **新增** Kpanda metrics 支持 Nvlink 指标

优化

- **优化** 解除接入管理集群及子集群时增加相关展示
- **优化** service 更新时，loadBalancerIP 不可用情况下的提示
- **优化** binding-syncer 支持 lease 选举
- **优化** kpanda 整卡模式的算力单位描述统一
- **优化** GPU 监控指标支持中英文切换
- **优化** vGPU 安装建议默认关闭 servicemonitor 提示

修复

- **修复** 创建了多云实例-命名空间，但在容器管理查询时报错问题
- **修复** job 列表点击重启按钮，提示报错且会删除 job 任务的问题
- **修复** 接入 kind 集群，kube-system 和 default 命名空间始终同步不到 ghippo 问题
- **修复** 工作空间绑定集群资源未过滤多云集群的问题
- **修复** 用户的 workspace admin 权限同步到 kpanda 中 cluster admin 权限时页面无显示的问题
- **修复** kpanda GetClusterAdminKubeConfig 接口生成的证书只有一年有效期，过期无法正常使用问题
- **修复** admin 用户限制了命名空间资源配置后，用 NS admin 账号登录，显示资源不限制问题
- **修复** ubuntu2004 在线环境，接入工作节点失败问题
- **修复** 通过安装器从 14.1 升级到 16.1 版本，创建集群报错并且无法选择 kube version 问题
- **修复** 创建集群失败后 回显问题
- **修复** kocral 不支持 集群资源 namespaces 的备份恢复问题
- **修复** 定时巡检关闭操作不生效问题
- **修复** insight 组件安装完毕仍有 Pod 存在问题时，前端返回信息有误问题
- **修复** vGPU 模式下，创建 Deployment 指定 GPU 型号时，GPU 列表缺失问题
- **修复** GPU Pod 监控仪表盘，有几处的展示信息不统一 问题

2024-04-30**v0.27.0**

新增

- **新增** 支持通过配置 CRD 的方式自定义 NS admin/editor/viewer 权限
- **新增** 支持通过 ssh 的方式连接 cloudtty
- **新增** 支持控制台进入运行中的容器
- **新增** 支持通过 Helm 应用 scend-mindxdl 安装昇腾组件 Device Plugin 和 NpuExporter，并通过 insight 查看昇腾 GPU 卡的相关指标

优化

- 优化 增加在 kpanda 界面 ns 绑定/解绑 workspace 的审计日志
- 优化 支持在本地容器内运行 make 命令
- 优化 ns quota 配置提示
- 优化 Helm 应用并发更新
- 优化 GPU 监控面板增加更多监控指标

修复

- **修复** 无法从 Deployment 的页面找到所挂载的 PVC 卷问题
- **修复** etcd 备份策略，无法选择 s3 region 问题
- **修复** kpanda openapi proxies 访问路线支持 token 权限认证
- **修复** liststorageclasses 接口返回结果不分页

- **修复** 版本发布流水线 `cd_to_prod_site` 任务执行失败，需要更新 CI/CD 脚本问题
- **修复** 多次上传文件到目标容器会出现上传失败的情况
- **修复** kairship 的 e2e 在线环境，`kpanda-binding-syncer` 长时间运行后出现负载升高集群不稳定
- **修复** 接入节点，运行时为 `docker` 时，`docker_rh_repo` 表单为空问题
- **修复** 创建集群失败之后，重试时的回显问题
- **修复** 管理集群解除接入后，子集群仍然显示被该集群纳管问题
- **修复** RedHat8-OS 创建集群，使用密钥认证节点检查失败
- **修复** 创建集群时勾选 `insight-agent` 安装，`fluent-bit` 无法正常启动问题
- **修复** 创建集群失败之后重试，由于 `kubelet` 有软链接，`kubespray` 执行 `reset.yml` 时失败
- **修复** `insight server` 升级时找不到 `Helm repo` 的问题
- **修复** 创建 `metallb` 时开启就绪检查，但是编辑进来开关显示为关闭问题
- **修复** `charts-syncer` 同步 `chart` 时，`relocateContainerImages` 设置为 `false` 会导致 同步出来的 `chart` 包 中 镜像地址有问题
- **修复** `sts+pvc` 恢复备份成功后，修改 `PVC` 的信息再次恢复备份，无法恢复修改 `PVC` 的数据问题
- **修复** 创建备份策略未开启备份数据卷，但是备份资源中包含 `PV`，导致备份失败问题
- **修复** `vGPU` 模式下 `GPU` 可用资源一直返回 0 问题
- **修复** `mig mixed` 模式下节点详情页的 `GPU` 类型展示了 MIG Mixed MIG Single 问题
- **修复** `ascend` 监控仪表盘 `ai` 处理器数目一直展示的总值
- **修复** 节点上的标签和节点详情页的 `GPU` 类型展示不一致

- **修复** GPU Pod Dashboard PCIE 数据错误问题

2024-03-28

v0.26.1

新增

- **新增** cloudtty 提供 ssh 代理的功能
- **新增** 支持接入 master 节点
- **新增** 应用备份计划支持通过 YAML 创建
- **新增** 应用备份可以通过资源类型来选择备份对象
- **新增** 集群巡检模板支持删除功能
- **新增** 支持通过 npu-exportor 部署 npu 监控面板
- **新增** 创建工作负载时，支持显示剩余可用的 GPU 资源
- **新增** 创建工作负载时，支持设置任务优先级
- **新增** vGPU 支持算力超配
- **新增** 提供 vGPU 场景化视频
- **新增** 创建集群时，支持让用户设置时区

优化

- **优化** 中标麒麟 v7u6 版本 产品化适配
- **优化** 资源使用率百分比显示
- **优化** 离线环境外置模式，优化创建集群时 yum repo 的信息需要手动选择的问题

- **优化** 对 Helm charts 上传的产品化引导并输出文档
- **优化** 当 GPU 开关打开的位置，引导用户部署 gpu-operator、nvidia vGPU
- **优化** GPU 切换逻辑
- **优化** 节点切换卡校验是否被分配
- **优化** vGPU 模式超配支持在节点详情看到超配后的资源

修复

- **修复** 创建集群的 kube-vip 异常问题
- **修复** 创建集群时，如果选择“为新建集群启用内核调优”，创建失败问题
- **修复** Helm 安装失败后重新安装失败
- **修复** 重新接入机器的 Helm 应用更新无法获取 value
- **修复** 安装 submariner 离线环境安装时，镜像地址默认渲染错误
- **修复** 通过 Kpanda 升级出现镜像地址重复拼接问题
- **修复** 备份恢复后，恢复始终处于处理中
- **修复** 集群巡检开启定时巡检后，没有在到达巡检频率后启动巡检
- **修复** 集群移除节点后，在仪表盘筛选中还是显示了
- **修复** 停掉带 GPU 的应用后，查看节点详情页的 GPU 显存分配率数据未更新，GPU 卡分配都已更新（开源问题）
- **修复** vGPU 模式下 Pod 的算力和显存监控指标有歧义
- **修复** MIG single 模式，分配数量不准确
- **修复** GPU 数量 MIG 模式下，偶尔出现不准确的情况
- **修复** 集群中多个节点存在 GPU，查看仪表盘不筛选节点的情况下，无法分辨到底是

哪个节点上的信息

- **修复** 创建工作负载使用天数 vGPU , 提示不清晰无法正确使用
- **修复** GPU 模式切换状态显示问题

2024-01-31

v0.25.0

新增

- **新增** 支持批量删除/停止多个工作负载
- **新增** 安装 集群的时候 , 支持让用户设置时区
- **新增** 安装 Velero 时支持一键开启 Velero 插件
- **新增** 创建集群时支持选择是否开启 kube-vip 的控制面负载均衡能力
- **新增** 支持导入异构 Addon 包
- **新增** 支持在指定型号的 GPU 卡上创建工作负载

优化

- **优化** GPU 节点切换功能可用性增强 , 切换时间降低到 2s 内
- **优化** GPU 模式切换逻辑更加的鲁邦
- **优化** gpu-operator Ubuntu 环境安装失败 , 文档增强
- **优化** GPU Dashboard 深度 review 优化 (涵盖 vGPU、MIG、GPU 整卡)
- **优化** 节点纬度 GPU 统计相关功能使用自定义指标进行优化
- **优化** 大规模集群详情页 -> 创建 PVC -> 数据卷 PV 下拉访问时延超过 100ms -> 400m

- **优化** 大规模集群详情页 -> 网络策略 -> 引用工作负载下拉访问延迟超过 100ms -> 300
- **优化** 大规模集群详情页 -> 路由创建 -> 目标服务下拉访问延迟超过 100ms -> 300ms
- **优化** 大规模添加转发规则后，切换命名空间（其中命名空间中存在 1000+ 服务）会导致浏览器卡死 2s 以上
- **优化** 镜像选择器，解决 1000+ 镜像空间时页面卡死问题
- **优化** 应用备份逻辑

修复

- **修复** crontphpa 配置使用定时 cron 表达式后会导致无法修改定时任务配置
- **修复** redis sentinel 配置导致安装器无限循环
- **修复** 控制台（cloudshell）重连机制，一直被刷新，影响命令运行
- **修复** 对接 DCE4 后，container cidr 显示不正确
- **修复** 安装器在线升级时，kcoral 镜像地址未改成在线地址
- **修复** 备份恢复时，Job 未恢复
- **修复** 同时开启 hpa 和 crontphpa，crontphpa 会被覆盖
- **修复** kpanda 创建集群时选择安装 insight 插件无效
- **修复** 当前 全局服务集群无法升级，页面显示可升级
- **修复** 创建集群的时候，高级设置不支持 calico_node_extra_envs 设置多行
- **修复** 集群巡检报告容器组内存使用率等相关指标结果显示异常
- **修复** NVIDIA GPU Pod 仪表盘中 Pod 的筛选未过滤到已经删除的 Pod 信息
- **修复** 创建集群的时候，关闭统一密码后，用户名和密码框依然显示
- **修复** 创建集群时，如果选择“为新建集群启用内核调优”，创建失败

2023-12-31**v0.24.0**

新增

- **新增** kpanda 审计日志中支持记录服务、路由、数据卷声明、数据卷、存储池资源的创建与删除操作
- **新增** kpanda 适配 kubeant 实现 k8s 版本的向下兼容
- **新增** Cloudtty 支持 Pod 热启动
- **新增** 实现 clusterpedia 对接 OTEL Tracing
- **新增** 安全、巡检、备份、虚拟机等组件支持最小化安装
- **新增【文档】**支持用户将自定义 Helm 导入到 系统内置的 addon repo 中去
- **新增【文档】**DCE 4.0 到 DCE 5.0 有限场景的迁移方案

优化

- **优化** kpanda 加入大量集群后 Pod 列表刷新时间太长了
- **优化** 升级 gpu-operator 到 v23.9.0 , 缩小与社区版本差距
- **优化** 备份整个 ns (ns 下存在 cr 以及 PVC 相关的内容) , 备份成功后进行恢复 , 显示部分成功但是看不出哪些是成功的哪些是失败的

修复

- **修复** addon 生命周期管理过程导致权限泄漏
- **修复** 定时伸缩的任务名称相同时 , 功能失效

- **修复** 离线环境安装了 kubernetes-cronhpa-controller 之后，页面无法检测到已经安装
- **修复** ListPodsByNodeOrigin 接口默认未按创建时间排序
- **修复** ListContainersByPod 接口小概率返回的 container 列表为空
- **修复** 流水线中的 scheduled_e2e 任务执行过程中报错提示 cluster member1 not exist ,
后续 tests 测试用例没有得到执行
- **修复** Data Collection 按照中文搜索无响应
- **修复** 命名空间-资源限额不生效和更新异常问题
- **修复** 工作负载-负载监控的读写数据永远为空
- **修复** gpu-operator 镜像没有全部离线问题
- **修复** Kpanda 的 Helm 安装会把集群管理权限泄露给普通用户
- **修复** 创建计划把备份数据卷打开后，备份详情里数据卷备份显示关闭
- **修复** 没有权限的用户，通过接口也能获取到其他集群的应用备份计划问题
- **修复** Velero 版本与 dce4 的 k8s 版本不兼容
- **修复** 大规模场景下用户和用户组列表接口加载缓慢
- **修复** 大规模场景下 clusterpedia 接口超时报错，云边协同状态无法正常获取，导致无法使用
- **修复** 大规模场景下命名空间没有展示绑定的所有命名空间
- **修复** 大规模场景下全局服务集群中的容器管理下的工作负载等页面 ns 接口加载缓慢，
导致页面使用卡顿

2023-11-30**v0.23.0**

新功能

- **新增** 支持重点功能的审计日志，如集群创建、删除、接入、解除接入、升级；节点接入、解除接入；（无状态、有状态、守护、任务、定时任务）的创建/删除、Helm 应用的部署/删除
- **新增** 对接 ghippo ldap 用户名超过 K8s 合法范围的用户体系
- **新增** 支持 insight-agent 等超大 chart 生命周期管理
- **新增** ConfigMap/Secret 支持热加载
- **新增** 数据存储支持 subPathExpr

优化

- **优化** 支持展示事件所属的 Namespace
- **优化** ETCD 备份策略状态
- **优化** Mysql 故障时报错信息
- **优化** 工作负载节点亲和性/工作负载亲和性/工作负载反亲和性
- **优化** 支持移除异常节点

修复

- **修复** 工作空间的可分配资源额度超过总配额
- **修复** SQL 注入的安全漏洞

- **修复** 创建 UOS 系统集群的失败问题

2023-11-06

v0.22.0

新功能

- **新增** 支持界面升级系统组件版本、修改系统组件参数
- **新增** 适配 [RedHat 9.2 创建集群](#)
- **新增** 支持 Nvidia 整卡、vGPU、MIG GPU 模式
- **新增** 支持天数智芯 GPU 卡
- **新增** 支持命名空间级 GPU 资源配额管理
- **新增** 支持应用级 GPU 资源配额
- **新增** 适配 [CentOS 7.9](#)、[Redhat8.4 gpu-operator](#) 的离线化部署和使用
- **新增** 支持集群、节点、应用级 GPU 资源监控
- **新增** 支持容器管理、应用备份恢复、集群巡检、安全扫描产品模块的离线升级
- **新增** 支持 Helm Chart 的多架构混部
- **新增** 支持集群同版本升级
- **新增** 支持 [Configmap/Secret 热加载](#)
- **新增** 创建集群-节点检查支持自定义参数配置，满足企业节点加密认证等场景

优化

- **优化** 支持在 Configmap/Secret 详情页查看关联信息

- **优化** 不同权限用户进入容器管理可见资源
- **优化** 新增 Helm Repo 支持自动刷新和间隔时间内自动刷新开关

修复

- **修复** 集群状态未知时，无法卸载集群的问题
- **修复** 容器组列表 CPU 使用率无数据问题
- **修复** ARM 架构无法安装 Insight-agent、Metrics-server 插件问题
- **修复** 使用密钥创建集群无法通过节点检查问题
- **修复** 创建负载无法添加环境变量问题
- **修复** 被删除用户数据残留问题
- **修复** CIS 合规性扫描、权限扫描以及漏洞扫描报告列表页面分页问题
- **修复** 创建静态 PV 指向错误 StorageClass 问题

2023-9-06

v0.21.0

新功能

- **新增** Helm Repo 密码连通性校验，支持跳过 TLS 证书认证
- **新增** 全局服务机器的工作节点扩容

优化

- **优化** 解除集群接入时支持卸载相关组件
- **优化** Pod 状态处理逻辑，新增 Pod 子状态

- **优化** 支持配置集群操作记录保留任务条数
- **优化** 创建工作集群支持配置控制节点数
- **优化** Insight-agent 未安装提示

修复

- **修复** 更新 Helm 应用实例时，配置参数丢失的问题
- **修复** Networkpolicy 关联实例展示报错问题
- **修复** 创建集群配置最大 Pod 数导致集群创建失败的问题
- **修复** 创建 Redhat 类型的工作集群失败的问题
- **修复** 命名空间级用户查看定时任务详情报“无权限”的问题
- **修复** 用户无法绑定工作空间的问题

2023-8-01

v0.20.0

新功能

- **新增** Helm 应用界面支持查看 Helm 操作日志
- **新增** 工作集群支持接入异构节点
- **新增** 创建集群支持批量导入节点
- **新增** 容器存储支持创建 NFS 类型的数据卷
- **新增** vGPU 支持，支持自动识别节点 CPU、支持新增为负载配置 CPU 配额

优化

- **优化** 集群接入逻辑，当接入集群二次接入新管理平台时，需要预先清理旧管理平台的数据冗余才能被接入，关于集群接入的更多细节，请参考[卸载/解除接入集群](#)
- **优化** 升级 clusterpedia 到 v0.7.0
- **优化** 基于权限的页面交互，无权限用户将无法进入无资源权限的页面
- **优化** 接入节点支持配置内核调优等高级参数配置
- **优化** Insight 组件安装检测机制

修复

- **修复** Helm 任务一直处在 **安装中**、**卸载中** 的问题
- **修复** 创建集群节点检查内核版本检测错误问题
- **修复** 创建集群插件无法自定义命名空间的问题
- **修复** 更新密钥默认增加 ca.crt 数据的问题

2023-7-06

v0.19.0

新功能

- **新增** 兼容 openAnolis / oracle linux 操作系统部署工作集群
- **新增** 离线环境创建集群支持自动添加 jfrog 的认证信息
- **新增** 创建工作负载新增环境变量规则校验
- **新增** 边缘负载和服务

- **新增** 双栈、系统内核作为节点前置检查项
- **新增** 创建工作负载新增 secretKey/configmapKey 作为配置项挂载在容器内

优化

- 优化 Helm 仓库刷新机制
- 优化 部分 I18N 英文翻译界面

修复

- **修复** 在创建集群时，填写自定义参数，如果 value 为 0 或者 1，会被错误转换为 true 或者 false 的问题
- **修复** 在离线环境创建集群时，无法写入 containerd 账号密码配置的问题
- **修复** 对 1.26 及以上版本的集群进行升级时，由于 kubernetes 镜像仓库更改的原因，导致集群升级失败的问题
- **修复** 命名空间级用户无法使用 StorageClasses 创建 PV 相关问题
- **修复** 创建路由时指定命名空间不生效的问题
- **修复** 集群升级后，日期返回错误问题

2023-6-03

v0.18.1

- 优化 安装集群设置自定义参数时不限最大长度

2023-5-28**v0.18.0**

新功能

- **新增** 巡检报告下载
- **新增** 对接高优先级操作全局审计日志
- **新增** 连接 Minio 的超时处理

优化

- **优化** CloudShell 由用 ConfigMap 进行 KubeConfig 挂载改为用 Secret 进行 KubeConfig 挂载
- **优化** 创建备份策略集群下拉列表新增过滤创建了备份策略的集群的开关

修复

- **修复** etcdbrctl 镜像离线化
- **修复** 镜像选择器无法选择镜像
- **修复** 创建集群时的 Repo 地址渲染

2023-04-28**v0.17.0**

新功能

- **新增** 巡检报告下载
- **新增** 查看 ETCD 备份日志
- **新增** 创建集群支持启用 Flannel、Kube-ovn 网络插件
- **新增** 创建集群启用 Cilium 双栈网络
- **新增** 创建集群支持自动识别节点 OS 类型
- **新增** Headless、External 类型的服务
- **新增** 离线环境下升级工作集群的 kubernetes 版本
- **新增** 集群级资源备份
- **新增** 使用私有密钥创建工作负载
- **新增** 配置 Helm job 的默认资源限制
- **新增** 使用 hwameistor 创建 PVC

优化

- **优化** 应用备份集群状态
- **优化** 负载详情内负载状态和负载下容器组状态不匹配的问题
- **优化** 离线模式下节点检查接口
- **优化** 多云应用的展示方式

修复

- **修复** 更新 Helm 应用配置丢失的问题
- **修复** 使用 YAML 创建多种类型资源由于 ns 不一致导致创建失败的问题
- **修复** 使用麒麟操作系统无法选择 Docker 19.03 运行时，导致创建集群失败的问题
- **修复** 英文界面的错误翻译

2023-04-04

v0.16.0

新功能

- **新增** 使用界面查询 PVC 事件
- **新增** 创建任务支持配置 backofflimit、completions、parallelism、activeDeadlineSeconds 等参数
- **新增** 集成自研开源存储组件 Hwameistor，支持在 容器存储 模块查看本地存储资源概览等信息
- **新增** 集群巡检功能，支持对集群进行秒级巡检（Alpha）
- **新增** 应用备份功能，支持界面化快速对应用进行备份和恢复（Alpha）
- **新增** 平台备份功能，支持对 ETCD 数据进行备份和恢复（Alpha）
- **新增** 支持 Ghippo 的自定义角色管理集群

优化

- **优化** Kpanda 卸载自建集群的流程，以此避免因用户误操作导致集群被删除

- **优化** 界面创建集群失败后重新创建集群的用户体验，支持用户基于失败前的配置快速重新安装集群
- **优化** 了当一个命名空间下存在多个 Quota 资源时的处理逻辑对多个 Quota 进行了聚合处理
- **优化** 工作负载详情内服务访问方式的信息展示，支持快速对负载服务进行访问
- **优化** Helm 仓库刷新机制，默认不开启自动刷新

修复

- **修复** Loadbalance 地址无法访问的问题
- **修复** 执行卸载集群操作失败的问题
- **修复** 接入集群超过 64 个字符导致集群无法获取的问题
- **修复** 离线环境集群无法展示集群插件的问题
- **修复** 全局服务集群无法更新配置的问题
- **修复** 创建集群时，第一次节点检查失败，无法再次执行节点检查的问题
- **修复** 创建/更新工作负载的环境变量不生效的问题

2023-02-27

v0.15.0

新功能

- **新增** 对 PV(Persistent Volumes) 的产品化支持，支持在创建 PVC 时选择已有数据卷
- **新增** 使用 kubernetes 无网络 CNI 创建集群的能力

- **新增** 支持负载、配置、服务等资源中文名称
- **新增** 通过 YAML 创建工作负载支持同时创建多种类型资源
- **新增** 工作负载的暂停、启动功能

优化

- **优化** 集群详情页，集群切换的使用体验
- **优化** 工作负载状态显示，增加 **已停止 (Stopped)** 状态
- **优化** 工作负载增加手动扩容窗口，简化用户手动扩容负载流程
- **优化** 接入集群无法接入 DCE4.X 集群问题

修复

- **修复** 了创建集群时，DNS 配置强制要求用户填写 upstream DNS 的问题
- **修复** 了工作负载版本记录排序混乱问题
- **修复** 通过 Helm 升级 Kubelet 无效的问题
- **修复** 创建集群执行节点检查失败后再次检查，上次的异常提示未消失的问题
- **修复** 创建工作负载，镜像拉取失败问题
- **修复** 定时的备份策略，无法执行 **立即执行** 操作的问题
- **修复** 修改无资源限制的工作负载时，UI 会自动添加资源限制问题
- **修复** 当 **工作空间** 没有与任何用户进行绑定时，往这个 **工作空间** 添加命名空间失败的问题
- **修复** 绑定、解绑命名空间会导致命名空间注解消失的问题
- **修复** 创建集群使用 **kube-vip** 策略不生效的问题

- **修复** 创建集群设置 `ntp servers` 为空时，将清空主机已有 `ntp` 地址问题

2022-12-29

v0.14.0

新功能

- **新增** Helm 模板支持展示中文名称和模板供应商
- **新增** CronHPA，支持定时伸缩工作负载
- **新增** VPA（垂直伸缩），支持手动/自动两种方式修改资源请求值，实现工作负载垂直伸缩
- **新增** Namespace 独享主机功能
- **新增** 存储池（StorageClass）支持授权给特定命名空间独享或共享
- **新增** 创建工作负载支持展示当前命名空间剩余资源配置
- **新增** 节点连通性检查功能
- **新增** 镜像选择器，支持创建工作负载时选择镜像仓库内的镜像
- **新增** 应用备份与恢复功能

优化

- **优化** 集群卸载流程，增加集群删除保护开关
- **优化** 通过 YAML 创建资源时支持同时创建多个资源
- **优化** 工作负载增加手动扩容窗口，简化用户手动扩容负载流程
- **优化** 服务（Service）访问方式体验，支持服务快速访问和展示节点、负载均衡地址

- 优化 文件上传下载支持选择特定容器
- 优化 支持不同 OS 系统离线安装
- 优化 离线环境下创建集群——节点配置支持选择节点操作系统和修改离线 Yum 源
- 优化 YAML 编辑器未填写 Namespace 字段，支持自动补齐为 Default
- 优化 集群升级界面交互体验
- 优化 使用 Helm 创建应用时，提供 Namespace 快速创建入口

修复

- 修复 无法使用密码新增节点的问题
- 修复 获取 Token 方式接入的集群 kubeconfig 错误问题
- 修复 授予权限时无法获取完整的用户和用户组
- 修复 Bindingsync 组件不正常时解绑工作空间原始权限存在问题
- 修复 Workspace Resync 无法正确将多余权限删除的问题
- 修复 删除中的 Namespace 还可以被选择的问题
- 修复 创建密钥，密钥数据单行显示的问题

2022-11-29

v0.13.0

新功能

- 新增 ReplicatSets 产品化：
 - 支持使用 Web 终端（CloudTTY）管理 ReplicatSets

- 支持查看 ReplicatSets 监控、日志、Yaml、事件、容器
 - 支持查看 ReplicatSets 详情
 - 联动应用工作台，由灰度发布管理 ReplicatSets 全生命周期
- **新增** Pod 详情页面
 - **新增** 命名空间详情页
 - **新增** 使用 Web 终端上传文件至容器内及从 Pod 内下载文件至本地
 - **新增** 工作负载基于自定义指标弹性伸缩，更加贴近用户实际业务弹性扩缩容需求

优化

- **优化** 部署集群支持：
 - 使用 Cilium CNI 部署集群
 - 使用不同用户名、密码、SSH 端口的节点创建集群
- **优化** Pod 列表支持查看容器组总数和运行中数量，以及支持查看容器类型
- **优化** 工作负载增加手动扩缩容窗口，简化用户手动扩缩负载流程
- **优化** 容器日志支持查看 init container 和 ephemeral container，提供更友好的运维体验
- **优化** 节点详情，注解 value 值未正确展示问题
- **优化** 操作提示反馈，给予用户操作以正确的反馈

修复

- **修复** 因创建命名空间和绑定工作空间强耦合导致创建命名空间失败的问题
- **修复** 更新路由规则无法修改转发策略的路径前缀问题

- **修复** 创建工作负载界面同时创建 Services 不生效的问题
- **修复** 更新服务异常报错问题
- **修复** 无法接入 AWS 集群问题
- **修复** 使用 WS Admin 用户绑定资源组后用户列表不同步问题
- **修复** 配置详情页，PageSize=50 时，ListClusterConfigMaps 接口异常报错问题

2022-10-28

v0.10.0

新功能

- **新增** NetworkPolicy 策略管理功能，包括创建、更新、删除 NetworkPolicy 策略，以及 NetworkPolicy 策略详情展示，帮助用户为 Pod 配置进网络出入流量策略
- **新增** 工作负载支持多网卡配置和支持 IP Pool 展示，满足用户为工作负载配置单独配置多网卡需求
- **新增** 集群创建失败后支持查看创建过程的操作日志，帮助用户快速定位故障
- **新增** 有状态工作负载支持使用动态数据卷模板
- **新增** 创建集群、创建 Secret、创建 Ingress、编辑命名空间配额的信息校验，帮助引导用户输入正确的配置参数，降低用户创建任务失败体验

优化

- **优化** 集群下拉列表支持展示集群状态，优化用户在创建集群选择被纳管集群、创建命名空间选择目标集群、集群授权选择目标集群的使用体验

- 优化 在 Helm 应用中安装 insight-agent 插件，支持自动获取并填充全局服务集群的 Insight-server 相关地址
- 优化 Helm 模板图标为空时的默认图标
- 优化 创建集群时选择网络模式为 None，以允许用户在集群创建完成后再安装网络插件
- 优化 集群操作信息架构：
 - 将集群列表和集群概览页面的集群升级操作，调整至集群详情内的集群运维功能下
 - 当某个管理集群在集群列表内移除后，基于这个管理集群创建的集群将在界面隐藏集群升级、纳管节点、删除节点操作

修复

- 修复 资源切换时，所选命名空间自动转换为全部命名空间的问题

离线升级容器管理模块

本页说明[下载容器管理模块](#)后，应该如何安装或升级。

!!! info

下述命令或脚本内出现的 `kpanda` 字样是容器管理模块的内部开发代号。

从安装包中加载镜像

从下载的安装包中加载镜像

您可以根据下面两种方式之一加载镜像，当环境中存在镜像仓库时，建议选择 chart-syncer

同步镜像到镜像仓库，该方法更加高效便捷。

方式一：使用 chart-syncer 同步镜像

使用 chart-syncer 可以将您下载的安装包中的 Chart 及其依赖的镜像包上传至安装器部署 DCE 时使用的镜像仓库和 Helm 仓库。

首先找到一台能够连接镜像仓库和 Helm 仓库的节点（如火种节点），在节点上创建 load-image.yaml 配置文件，填入镜像仓库和 Helm 仓库等配置信息。

1. 创建 load-image.yaml

!!! note

该 YAML 文件中的各项参数均为必填项。

==== “已添加 Helm repo”

若当前环境已安装 Chart repo，chart-syncer 也支持将 Chart 导出为 tgz 文件。

```
```yaml title="load-image.yaml"
source:
 intermediateBundlesPath: kpanda # (1)!
target:
 containerRegistry: 10.16.10.111 # (2)!
 containerRepository: release.daocloud.io/kpanda # (3)!
repo:
 kind: HARBOR # (4)!
 url: http://10.16.10.111/chartrepo/release.daocloud.io # (5)!
 auth:
 username: "admin" # (6)!
 password: "Harbor12345" # (7)!
containers:
 auth:
 username: "admin" # (8)!
 password: "Harbor12345" # (9)!
````
```

1. 使用 chart-syncer 之后 .tar.gz 包所在的路径
2. 镜像仓库地址
3. 镜像仓库路径
4. Helm Chart 仓库类别

5. Helm 仓库地址
6. 镜像仓库用户名
7. 镜像仓库密码
8. Helm 仓库用户名
9. Helm 仓库密码

==== “未添加 Helm repo”

若当前节点上未添加 helm repo, chart-syncer 也支持将 Chart 导出为 tgz 文件，并存放 在指定路径。

```
```yaml title="load-image.yaml"
source:
 intermediateBundlesPath: kpanda # (1)!
target:
 containerRegistry: 10.16.10.111 # (2)!
 containerRepository: release.daocloud.io/kpanda # (3)!
repo:
 kind: LOCAL
 path: ./local-repo # (4)!
containers:
 auth:
 username: "admin" # (5)!
 password: "Harbor12345" # (6)!

```

```

1. 使用 chart-syncer 之后 .tar.gz 包所在的路径
2. 镜像仓库 url
3. 镜像仓库路径
4. Chart 本地路径
5. 镜像仓库用户名
6. 镜像仓库密码

2. 执行同步镜像命令。

```
charts-syncer sync --config load-image.yaml
```

方式二：使用 Docker 或 containerd 加载镜像

解压并加载镜像文件。

1. 解压 tar 压缩包。

```
tar xvf kpanda.bundle.tar
```

解压成功后会得到 3 个文件：

- hints.yaml
- images.tar
- original-chart

2. 从本地加载镜像到 Docker 或 containerd。

```
==== "Docker"
```shell
docker load -i images.tar
```
==== "containerd"
```shell
ctr -n k8s.io image import images.tar
```

```

!!! note

每个 node 都需要做 Docker 或 containerd 加载镜像操作，
加载完成后需要 tag 镜像，保持 Registry、Repository 与安装时一致。

升级

有两种升级方式。您可以根据前置操作，选择对应的升级方案：

!!! note

从 kpanda 的 v0.21.0 版本开始，redis 支持设置 sentinel 密码，如果使用哨兵模式的 redis，
升级时需要变更 --set global.db.redis.url。例如：

- 原来是：`redis+sentinel://:3wPxzWffdn@rfs-mcamel-common-redis-cluster.mcamel-system.svc.cluster.local:26379/mymaster`
- 现在就要改成：`redis+sentinel://:3wPxzWffdn@rfs-mcamel-common-redis-cluster.mcamel-system.svc.cluster.local:26379/mymaster?master_password=3wPxzWffdn`

==== “通过 helm repo 升级”

1. 检查容器管理 Helm 仓库是否存在。

```
```shell
helm repo list | grep kpanda
```

```

若返回结果为空或如下提示，则进行下一步；反之则跳过下一步。

```
```none
Error: no repositories to show
```

```

1. 添加容器管理的 Helm 仓库。

```
```shell
helm repo add kpanda http://{harbor url}/chartrepo/{project}
```

```

1. 更新容器管理的 Helm 仓库。

```
```shell
helm repo update kpanda
```

```

1. 选择您想安装的容器管理版本（建议安装最新版本）。

```
```shell
helm search repo kpanda/kpanda --versions
```

```

输出类似于：

```
```none
NAME CHART VERSION APP VERSION DESCRIPTION
kpanda/kpanda 0.20.0 v0.20.0 A Helm Chart for kpanda
...
```

```

1. 备份 `--set` 参数。

在升级容器管理版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

```
```shell
helm get values kpanda -n kpanda-system -o yaml > bak.yaml
```

```

1. 更新 kpanda crds

```
```shell
helm pull kpanda/kpanda --version 0.21.0 && tar -zxf kpanda-0.21.0.tgz
kubectl apply -f kpanda/crds
```

```

1. 执行 `helm upgrade`。

升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 字段为当前使用的镜像仓库地址

。

```
```shell
export imageRegistry={你的镜像仓库}
```

```shell
helm upgrade kpanda kpanda/kpanda \
-n kpanda-system \
-f ./bak.yaml \
--set global.imageRegistry=$imageRegistry \
--version 0.21.0
```

```
```

### ==== “通过 Chart 包升级”

1. 备份 `--set` 参数。

在升级容器管理版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

```
```shell
helm get values kpanda -n kpanda-system -o yaml > bak.yaml
```
```
```

1. 更新 kpanda crds

```
```shell
kubectl apply -f ./crds
```
```
```

1. 执行 `helm upgrade` 。

升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 为当前使用的镜像仓库地址。

```
```shell
export imageRegistry={你的镜像仓库}
```

```shell
helm upgrade kpanda . \
-n kpanda-system \
-f ./bak.yaml \
--set global.imageRegistry=$imageRegistry
```
```
```

通过页面方式升级

前提条件

在安装 DCE 5.0 或在产品模块升级前已执行以下命令：

```
~/dce5-installer cluster-create -c /home/dce5/sample/clusterConfig.yaml -m /home/dce5/sample/manifest.yaml -d -j 14,15
```

操作步骤

1. 在 **集群列表** 页面中，搜索找到 `kpanda-global-cluster` 集群，进入集群详情

集群列表

集群列表

2. 在左侧导航栏中找到 Helm 应用，搜索 `kpanda` 找到容器管理模块，展开右侧操作栏，点击 **更新** 按钮，进行升级。

应用名称	状态	命名空间	Chart	仓库	更新时间	安装时间
kpanda	已部署	kpanda-system	kpanda:0.22.0+rc1	addon	2023-10-27 03:...	2023-10-27 03:...

集群列表

升级已知问题

升级到 v0.25.1 但 < v0.29.0

问题描述：通过页面方式将 kpanda 低版本升级到 v0.25.1 或更高版本时，可能存在镜像

地址拼接问题，导致升级失败，报错提示如下：

```
events:
  Type    Reason     Age   From          Message
  ----  -----  ----  ----
  Normal  Scheduled  23s   default-scheduler  Successfully assigned kpanda-system/kpanda-pre-hook-install-crds-wllx5 to g-master3
  Normal  BackOff   19s  (x2 over 20s)  kubelet    Back-off pulling image "10.6.135.222/release.daocloud.io/release.daocloud.io/kpanda/kpanda-shell:v0.0.9"
  Warning Failed    19s  (x2 over 20s)  kubelet    Error: ImagePullBackOff
  Normal  Pulling    6s   (x2 over 20s)  kubelet    Pulling image "10.6.135.222/release.daocloud.io/release.daocloud.io/kpanda/kpanda-shell:v0.0.9"
  Warning Failed    6s   (x2 over 20s)  kubelet    Failed to pull image "10.6.135.222/release.daocloud.io/release.daocloud.io/kpanda/kpanda-shell:v0.0.9": rpc error: code = NotFound desc = failed to pull and unpack image "10.6.135.222/release.daocloud.io/release.daocloud.io/kpanda/kpanda-shell:v0.0.9": failed to resolve reference "10.6.135.222/release.daocloud.io/release.daocloud.io/kpanda/kpanda-shell:v0.0.9": 10.6.135.222/release.daocloud.io/release.daocloud.io/kpanda/kpanda-shell:v0.0.9: not found
  Warning Failed    6s   (x2 over 20s)  kubelet    Error: ErrImagePull
[root@g-master1 ~]#
```

镜像地址报错

解决办法：

在 Helm 应用中更新 kpanda 时，修改 yaml 文件，将 repository 地址改成 repository:

xxx/xxx 形式。

??? note “点击查看详细的 YAML 示例”

```
```yaml
global:
 imageRegistry: 10.6.135.222/release.daocloud.io
 imagePullSecrets: []
 storageClass: ""
kpanda:
 imageTag: v0.25.1
 enableGhippoRoutes: true
 enableSidecar: true
db:
 builtIn: false
 redis:
 url: >-
 redis+sentinel://rfs-mcamel-common-redis-cluster.mcamel-system.svc.cluster.local:26379/m
 ymaster?master_password=XFDYqKEyJU
 image:
 registry: release.daocloud.io
 repository: kpanda/redis
 tag: 7.0.5-alpine
```

```

```
    pullPolicy: IfNotPresent
  telemetry:
    tracing:
      enabled: true
      addr: >-
        insight-agent-opentelemetry-collector.insight-system.svc.cluster.local:4317
  metrics:
    enabled: true
    path: /metrics
    port: 81
  busybox:
    image:
      registry: release.daocloud.io
      repository: library/busybox
      tag: 1.34.1
    pullPolicy: IfNotPresent
  shell:
    image:
      registry: release.daocloud.io
      repository: kpanda/kpanda-shell
      tag: v0.0.9
    pullPolicy: IfNotPresent
  controllerManager:
    labels:
      app: kpanda-controller-manager
    replicaCount: 2
    podAnnotations: {}
    podLabels:
      app: kpanda-controller-manager
    image:
      registry: release.daocloud.io
      repository: kpanda/kpanda-controller-manager
      tag: ""
    pullPolicy: IfNotPresent
    pullSecrets: []
  livenessProbe:
    enabled: true
    initialDelaySeconds: 30
    timeoutSeconds: 5
    periodSeconds: 30
    successThreshold: 1
    failureThreshold: 3
    scheme: HTTP
  readinessProbe:
```

```
enabled: true
initialDelaySeconds: 30
timeoutSeconds: 5
periodSeconds: 30
successThreshold: 1
failureThreshold: 3
scheme: HTTP

resources:
  requests:
    cpu: 200m
    memory: 200Mi
nodeSelector: {}
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 60
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values:
                  - kpanda-controller-manager
          topologyKey: kubernetes.io/hostname
tolerations: []
apiServer:
  createDefaultOrderIndex: true
  insightAgentRegistryOverride: true
  labels:
    app: kpanda-apiserver
  replicaCount: 2
  podAnnotations: {}
  podLabels: {}
  image:
    registry: release.daocloud.io
    repository: kpanda/kpanda-apiserver
    tag: ""
    pullPolicy: IfNotPresent
    pullSecrets: []
livenessProbe:
  enabled: true
  initialDelaySeconds: 30
  timeoutSeconds: 5
  periodSeconds: 30
```

```
  successThreshold: 1
  failureThreshold: 3
  scheme: HTTP
  readinessProbe:
    enabled: true
    initialDelaySeconds: 30
    timeoutSeconds: 5
    periodSeconds: 30
    successThreshold: 1
    failureThreshold: 3
    scheme: HTTP
  resources:
    requests:
      cpu: 200m
      memory: 200Mi
  hostNetwork: false
  nodeSelector: {}
  affinity: {}
  tolerations: []
  serviceType: ClusterIP
  nodePort: null
  configMap:
    addon:
      repo:
        - URL: http://10.6.135.222:8081
          name: addon
          password: rootpass123
          username: rootuser
  kpanda-proxy:
    enabled: true
    proxyIngress:
      replicaCount: 2
      podAnnotations: {}
      podLabels: {}
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
      nodeSelector: {}
      affinity: {}
      tolerations: []
    proxyEgress:
      replicaCount: 2
      podAnnotations: {}
```

```
podLabels: {}  
resources:  
  requests:  
    cpu: 100m  
    memory: 128Mi  
nodeSelector: {}  
affinity: {}  
tolerations: []  
clusterpedia:  
  enabled: true  
  podLabels:  
    sidecar.istio.io/inject: 'true'  
mysql:  
  enabled: false  
  image:  
    registry: release.daocloud.io  
    repository: kpanda/mysql  
    tag: 8.0.29  
  primary:  
    persistence:  
      enabled: false  
    resources:  
      limits:  
        cpu: 1  
        memory: 1Gi  
    requests:  
      cpu: 100m  
      memory: 128Mi  
postgresql:  
  enabled: false  
  image:  
    registry: release.daocloud.io  
    repository: kpanda/postgresql  
    tag: 15.3.0-debian-11-r7  
  primary:  
    persistence:  
      enabled: false  
    resources:  
      limits:  
        cpu: 1  
        memory: 1Gi  
    requests:  
      cpu: 100m  
      memory: 128Mi
```

```
storageInstallMode: external
externalStorage:
  type: mysql
  dsn: >-
    kpanda:@tcp(mcamel-common-kpanda-mysql-cluster-mysql-master.mcamel-system.svc.cluster
.local:3306)/kpanda?charset=utf8mb4&multiStatements=true&parseTime=true
    host: "
    port: null
    user: "
    password: ihKhByQ2Af
    database: "
    accessType: readwrite
    connMaxIdleSeconds: 1800
    connMaxLifetimeSeconds: 3600
    maxIdleConns: 10
    maxOpenConns: 100
installCRDs: true
persistenceMatchNode: None
apiserver:
  replicaCount: 2
  podAnnotations: {}
  podLabels:
    sidecar.istio.io/inject: 'true'
image:
  registry: release.daocloud.io
  repository: clusterpedia/apiserver
  tag: v0.7.1-rc.0
  pullPolicy: IfNotPresent
  pullSecrets: []
featureGates:
  RemainingItemCount: false
  AllowRawSQLQuery: true
resources: {}
tolerations: []
clustersynchroManager:
  replicaCount: 2
  podAnnotations: {}
  podLabels:
    sidecar.istio.io/inject: 'true'
    app: kpanda-clusterpedia-clustersynchro-manager
image:
  registry: release.daocloud.io
  repository: clusterpedia/clustersynchro-manager
  tag: v0.7.1-rc.0
```

```
pullPolicy: IfNotPresent
pullSecrets: []
featureGates:
  PruneManagedFields: true
  PruneLastAppliedConfiguration: true
  AllowSyncAllCustomResources: true
  AllowSyncAllResources: true
  HealthCheckerWithStandaloneTCP: true
resources: {}
nodeSelector: {}
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 60
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values:
                  - kpanda-clusterpedia-clustersynchro-manager
    topologyKey: kubernetes.io/hostname
tolerations: []
leaderElect:
  leaseDuration: 60s
  renewDeadline: 50s
  retryPeriod: 5s
  resourceLock: leases
controllerManager:
  labels: {}
  replicaCount: 1
  podAnnotations: {}
  podLabels:
    sidecar.istio.io/inject: 'true'
image:
  registry: release.daocloud.io
  repository: clusterpedia/controller-manager
  tag: v0.7.1-rc.0
  pullPolicy: IfNotPresent
  pullSecrets: []
hookJob:
  image:
    registry: release.daocloud.io
    repository: kpanda/kpanda-shell
```

```
    tag: v0.0.9
    pullPolicy: IfNotPresent
  ui:
    enabled: true
    replicaCount: 2
    podAnnotations: {}
    podLabels: {}
    image:
      registry: release.daocloud.io
      repository: kpanda/kpanda-ui
      tag: v0.24.1
      pullPolicy: IfNotPresent
      pullSecrets: []
    resources: {}
    nodeSelector: {}
    affinity: {}
    tolerations: []
  cloudtty:
    enabled: true
    labels: {}
    replicaCount: 1
    podAnnotations: {}
    podLabels:
      sidecar.istio.io/inject: 'true'
    image:
      registry: release.daocloud.io
      repository: cloudtty/cloudshell-operator
      tag: v0.6.3
      pullPolicy: IfNotPresent
      pullSecrets: []
    resources:
      requests:
        cpu: 100m
        memory: 128Mi
    nodeSelector: {}
    affinity: {}
    tolerations: []
  cloudshellImage:
    registry: release.daocloud.io
    # 将 repository 地址修改成 repository: xxx/xxx 形式, 如 repository: cloudtty/cloudshell
    repository: cloudtty/cloudshell
    tag: v0.6.3
  hookJob:
    image:
```

```

registry: release.daocloud.io
repository: kpanda/kpanda-shell
tag: v0.0.9
pullPolicy: IfNotPresent
helmJobImageOverride:
  enabled: true
  registry: release.daocloud.io
  repository: kpanda/kpanda-shell
  tag: v0.0.9
etcdbBackupRestore:
  image:
    registry: release.daocloud.io
    repository: kpanda/etcdbrctl
    tag: v0.22.0
```

```

## 升级到 v0.29.0 或更高版本

**问题描述：**将 Kpanda 低版本升级到 v0.29.0 或更高版本时，节点如果是 GPU MIG 模式，系统会将原来的 GPU MIG 模式强制切换成 GPU 整卡模式，这会影响业务运行。您可以执行如下操作可避免此问题。

**业务中断式升级（使用场景：已开启 GPU MIG 模式，但实际未使用）：**

1. 停止所有 GPU 应用
2. 并且卸载 gpu-operator 以及 nvidia-vgpu
3. 升级完成后重新安装 gpu-operator，并且保证 gpu-operator 版本大于 v23.9.0+1

**业务连续式升级（使用场景：已开启 GPU MIG 模式，并且有实际业务正在使用 MIG 模式）：**

1. 手动修改 node 标签 gpu.node.kpanda.io/nvidia-gpu-mode: mig
2. 升级 Kpanda 版本
3. 升级 gpu-operator 版本 >= v23.9.0+1

# 容器管理常见问题

本页面列出了一些在容器管理（Kpanda）中可能遇到的问题，为您提供便利的故障排除解决办法。

- [容器管理和全局管理模块的权限问题](#)
- Helm 安装：
  - [Helm 应用安装失败，提示“OOMKilled”](#)
  - [Helm 安装应用时，无法拉取 kpanda-shell 镜像](#)
  - [Helm Chart 界面未显示最新上传到 Helm Repo 的 Chart](#)
  - [Helm 安装应用失败时卡在安装中无法删除应用重新安装](#)
- [工作负载 -> 删除节点亲和性等调度策略后，调度异常](#)
- 应用备份：
  - [Kcoral 应用备份检测工作集群 Velero 状态的逻辑是什么](#)
  - [在跨集群备份还原时，Kcoral 如何获取可用集群](#)
  - [Kcoral 备份了相同标签的 Pod 和 Deployment，但还原备份后出现 2 个 Pod](#)
- [卸载 VPA、HPA、CronHPA 之后，为什么对应弹性伸缩记录依然存在](#)
- [为什么低版本集群的控制台打开异常](#)
- 创建和接入集群：
  - [如何重置创建的集群](#)
  - [接入集群安装插件失败](#)
  - [创建集群时在高级设置中启用 为新建集群内核调优，集群创建为什么会失败](#)

- [集群解除接入后，kpanda-system 命名空间一直处于 Terminating 状态](#)

## 权限问题

有关容器管理和全局管理模块的权限问题，经常有用户会问，为什么我这个用户可以看到这个集群，或者为什么我看不到这个集群，我们应该如何排查相关的权限问题？分为以下三种情况：

- 容器管理模块的权限分为集群权限、命名空间权限。如果绑定了用户，那该用户就可以查看到相对应的集群及资源。具体权限说明，可以参考[集群权限说明](#)。

### 容器管理权限

#### 容器管理权限

- 全局管理模块中用户的授权：使用 admin 账号，进入 **全局管理 -> 用户与访问控制** -> **用户** 菜单，找到对应用户。在 **授权所属用户组** 标签页，如果有类似 Admin、Kpanda Owner 等拥有容器管理权限的角色，那即使在容器管理没有绑定集群权限或命名空间权限，也可以看到全部的集群，可以参考[用户授权文档说明](#)

### 全局管理 用户授权

#### 全局管理 用户授权

- 全局管理模块中工作空间的绑定：使用账号进入 **全局管理 -> 工作空间与层级**，可以看到自己的被授权的工作空间，点击工作空间名称
  1. 如果该工作空间单独授权给自己，就可以在授权标签页内看到自己的账号，然后查看资源组或共享资源标签页，如果资源组绑定了命名空间或共享资源绑定了集群，那该账号就可以看到对应的集群
  2. 如果是被授予了全局管理相关角色，那就无法授权标签页内看到自己的账号，

也无法在容器管理模块中看到工作空间所绑定的集群资源

### 全局管理工作空间的绑定

#### 全局管理工作空间的绑定

[返回顶部 :arrow\\_up:](#)

## Helm 安装的问题

### 1. Helm 应用安装失败，提示“OOMKilled”

#### 失败情况

#### 失败情况

如图所示，容器管理会自动创建启动一个 Job 负责具体应用的安装工作，在 v0.6.0 版本中由于 job resources 设置不合理，导致 OOM，影响应用安装。该 bug 在 0.6.1 版本中已经被修复。如果是升级到 v0.6.1 的环境，仅仅会在新创建、接入的集群中生效，已经存在的集群需要进行手动调整，方能生效。

??? note “点击查看如何调整脚本”

- 以下脚本均在全局服务集群中执行
  - 找到对应集群，本文以 skoala-dev 为例，获取对应的 skoala-dev-setting configmap
  - 更新 configmap 之后即可生效

```
```shell
kubectl get cm -n kpanda-system skoala-dev-setting -o yaml
apiVersion: v1
data:
  clusterSetting: '{"plugins": [{"name": 1, "intelligent_detection": true}, {"name": 2, "enabled": true, "intelligent_detection": true}, {"name": 3, "enabled": true, "intelligent_detection": true}, {"name": 6, "intelligent_detection": true}, {"name": 7, "intelligent_detection": true}, {"name": 8, "intelligent_detection": true}, {"name": 9, "intelligent_detection": true}], "network": [{"name": 4, "enabled": true, "intelligent_detection": true}, {"name": 5, "intelligent_detection": true}, {"name": 10, "enabled": true, "intelligent_detection": true}, {"name": 11}], "addon_setting": {"helm_operation_history_limit": 100, "helm_repo_refresh_interval": 600, "helm_operation_base_image": "release-ci.daocloud.io/kpanda/kpanda-shell:v0.0.6", "helm_operation_job_template_resources": {"limits": {"cpu": "50m", "memory": "120Mi"}, "requests": {"cpu": "50m", "memory": "120Mi"}}, "clusterlcm_setting": {"enable_deletion_protection": true}, "etcd_backup_restore_setting": {"base
```

```

_image":"release.daocloud.io/kpanda/etcdbrctl:v0.22.0" }}'
kind: ConfigMap
metadata:
labels:
    kpanda.io/cluster-plugins: ""
name: skoala-dev-setting
namespace: kpanda-system
ownerReferences:
- apiVersion: cluster.kpanda.io/v1alpha1
    blockOwnerDeletion: true
    controller: true
    kind: Cluster
    name: skoala-dev
    uid: f916e461-8b6d-47e4-906e-5e807bfe63d4
uid: 8a25dfa9-ef32-46b4-bc36-b37b775a9632
```

```

修改 clusterSetting -> helm\_operation\_job\_template\_resources 到合适的值即可，  
v0.6.1 版本对应的值为 cpu: 100m,memory: 400Mi

[返回顶部 :arrow\\_up:](#)

## 2. Helm 安装应用时，无法拉取 kpanda-shell 镜像

使用离线安装后，接入的集群安装 helm 应用经常会遇到拉取 kpanda-shell 镜像失败，

如图：

拉取镜像失败

拉取镜像失败

此时，只需要去集群运维-集群设置页面，高级配置标签页，修改 Helm 操作基础镜像

为一个可以被该集群正常拉取到的 kpanda-shell 的镜像即可。

修改镜像

修改镜像

[返回顶部 :arrow\\_up:](#)

## 3. Helm Chart 界面未显示最新上传到对应 Helm Repo 的 Chart

## 模板

模板

此时，只需要去 Helm 仓库刷新对应的 Helm 仓库即可。

刷新仓库

刷新仓库

[返回顶部 :arrow\\_up:](#)

4. Helm 安装应用失败时卡在安装中无法删除应用重新安装

删除失败

删除失败

此时，只需要去自定义资源页面，找到 helmreleases.helm.kpanda.io CRD，然后找到对应的 helmreleases CR 删除即可。

找到 CR

找到 CR

删除 CR

删除 CR

[返回顶部 :arrow\\_up:](#)

## 调度的问题

在通过 **工作负载**，删除节点亲和性等调度策略后，调度异常

调度异常

调度异常

此时，可能是因为策略没有删除干净，**点击编辑**，删除所有策略。

编辑

编辑

删除

删除

正常调度

正常调度

[返回顶部 :arrow\\_up:](#)

## 应用备份的问题

1. Koral 检测工作集群 Velero 状态的逻辑是什么？

检测

检测

- 工作集群在 velero 命名空间下安装了标准的 velero 组件
- velero 控制面 velero deployment 处于运行状态，并达到期望的副本数
- velero 数据面 node agent 处于运行状态，并达到期望副本数
- velero 成功连接到目标 MinIO ( BSL 状态为 Available )

2. 在跨集群备份还原时，Koral 如何获取可用集群？

在通过 Koral 跨集群备份还原应用的时候，在恢复页面中，Koral 会帮助用户筛选可

以执行跨集群还原的集群列表，逻辑如下：

筛选

筛选

- 过滤未安装 Velero 的集群列表

- 过滤 Velero 状态异常的集群列表
- 获取与目标集群对接了相同 MinIO 和 Bucket 的集群列表并返回

所以只要对接了相同的 MinIO 和 Bucket , Velero 处于运行状态 , 就可以跨集群备份  
( 需要有写入权限 ) 和还原。

3.Koral 进行应用备份操作 , 同时备份相同标签的 Pod 和 Deployment 后 , 还原备份后  
出现 2 个 Pod.

出现这种现象的原因是 : 还原时 , 由于修改了 Pod 标签 , 导致其标签与其备份时的父  
资源 ReplicaSet / Deployment 标签不匹配 , 故还原时出现 2 倍数量 Pod.

为了避免出现以上这种情况 , 尽量避免修改关联资源中的某一资源的标签。

[返回顶部 :arrow\\_up:](#)

## 日志的问题

卸载 VPA、HPA、CronHPA 之后 , 为什么对应弹性伸缩记录依然存在 ?

虽然通过 Helm Addon 市场中把对应组件卸载 , 但是应用弹性伸缩界面相关记依然在 , 如  
下图所示:

编辑

编辑

这是 helm uninstall 的一个问题 , 它并不会卸载对应的 CRD , 因此导致数据残留 , 此时我  
们需要手动卸载对应的 CRD , 完成最终清理工作。

## 控制台的问题

为什么低版本集群的控制台打开异常 ?

在 kubernetes 低版本 ( v1.18 以下 ) 的集群中 , 打开控制台出现 CSR 资源请求失败。打开控制台的时候 , 会根据当前登录用户在目标集群中通过 CSR 资源申请证书 , 如果集群版本太低或者没有开启此功能 Controller , 会导致证书申请失败 , 从而无法连接到目标集群。

申请证书流程请参考 [Kubernetes 官网文档](#)。

### 解决办法 :

- 如果集群版本大于 v1.18 , 请检查 kube-controller-manager 是否开启 csr 功能 , 确保以下的 controller 是否正常开启  
ttl-after-finished,bootstrapsigner,csraproving,csrcleaner,csrsigning
- 低版本集群目前解决方案只有升级版本

[返回顶部 :arrow\\_up:](#)

## 创建和接入集群的问题

### 1. 如何重置创建的集群 ?

创建的集群分为两种情况 :

- 创建失败的集群 : 在创建集群的过程中 , 因为参数设置错误导致集群创建失败 , 这种情况可以在安装失败的集群选择重试 , 然后重新设置参数重新创建。
- 已经成功创建的集群 : 这种集群可以先卸载集群 , 然后重新创建集群。卸载集群需要关闭集群保护的功能才能卸载集群。

[关闭集群保护](#)

[关闭集群保护](#)

[卸载集群](#)

## 卸载集群

[返回顶部 :arrow\\_up:](#)

### 2. 接入集群安装插件失败

离线环境接入的集群，在安装插件之前，需要先配置 CRI 代理仓库，以忽略 TLS 验

证（所有节点都需要执行）。

==== “Docker”

1. 修改文件 `/etc/docker/daemon.json`  
2. 加入 "insecure-registries": ["172.30.120.243", "temp-registry.daocloud.io"],

修改之后内容如下：

![修改配置](docs/zh/docs/kpanda/images/faq01.png)

3. 重启 docker

```
```shell
systemctl restart docker
systemctl daemon-reload
```

```

==== “containerd”

1. 修改 `/etc/containerd/config.toml`  
2. 修改之后内容如下：

```
```shell
[plugins."io.containerd.grpc.v1.cri".registry.mirrors."docker.io"]
endpoint = ["https://registry-1.docker.io"]
[plugins."io.containerd.grpc.v1.cri".registry.mirrors."temp-registry.daocloud.io"]
endpoint = ["http://temp-registry.daocloud.io"]
[plugins."io.containerd.grpc.v1.cri".registry.configs."http://temp-registry.daocloud.io".t
ls]
insecure_skip_verify = true
```

```

![修改配置](docs/zh/docs/kpanda/images/faq02.png)

3. 注意空格和换行符，确保配置正确，修改完成之后执行

```
```shell
systemctl restart containerd
```

```

3. 创建集群时，在高级设置中启用 [为新建集群内核调优](#)，集群创建为什么会失败

1. 检查内核模块 `conntrack` 是否加载，执行如下命令：

```
lsmod |grep conntrack
```

2. 如果返回为空，表示没有加载。重新加载，执行如下命令：

```
modprobe ip_conntrack
```

**!!! note**

如果内核模块进行了升级操作，也会导致集群创建失败。

4. 集群解除接入后，`kpanda-system` 命名空间一直处于 `Terminating` 状态。

请检查 `APIServices` 服务状态是否正常，查看命令如下。如果当前状态为 `false`，请尝试修复 `APIServices` 或删除该服务。

```
kubectl get apiservices
```

[返回顶部 :arrow\\_up:](#)

## 创建工作集群

在 DCE 5.0 容器管理模块中，[集群角色](#)分四类：全局服务集群、管理集群、工作集群、接入集群。其中，接入集群只能从第三方厂商接入，参见[接入集群](#)。

本页介绍如何创建工作集群，默认情况下，新建工作集群的工作节点 OS 类型和 CPU 架构需要与全局服务集群保持一致。如需使用区别于全局服务集群 OS 或架构的节点创建集群，参阅[在 CentOS 管理平台上创建 Ubuntu 工作集群](#)进行创建。

推荐使用 [DCE 5.0 支持的操作系统](#)来创建集群。如您本地节点不在上述支持范围，可参考[在非主流操作系统上创建集群](#)进行创建。

## 前提条件

创建集群之前需要满足一定的前提条件：

- 根据业务需求准备一定数量的节点，且节点 OS 类型和 CPU 架构一致。
- 推荐 Kubernetes 版本 1.29.5，具体版本范围，参阅 [DCE 5.0 集群版本支持体系](#)，目前最新版 DCE 5.0 支持自建工作集群版本范围在 v1.28.0-v1.30.2。如需创建低版本的集群，请参考[集群版本支持范围](#)、[部署与升级 Kubelet 向下兼容版本](#)。
- 目标主机需要允许 IPv4 转发。如果 Pod 和 Service 使用的是 IPv6，则目标服务器需要允许 IPv6 转发。
- DCE 暂不提供对防火墙的管理功能，您需要预先自行定义目标主机防火墙规则。为了避免创建集群的过程中出现问题，建议禁用目标主机的防火墙。
- 参阅[节点可用性检查](#)。

## 操作步骤

1. 在 [集群列表](#) 页面中，点击 [创建集群](#) 按钮。

创建集群按钮

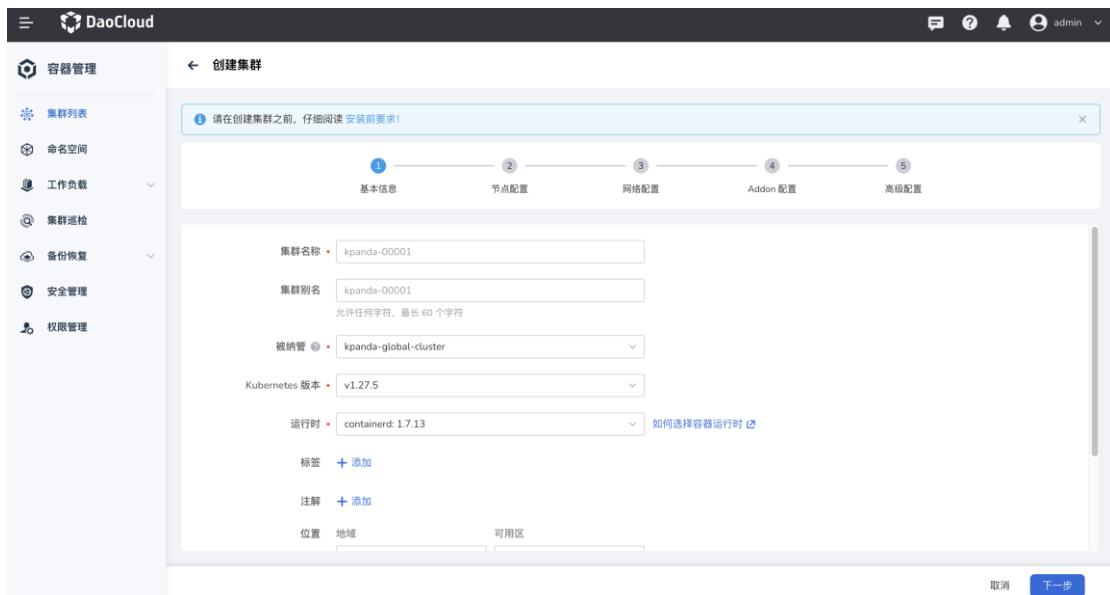
创建集群按钮

2. 参考下列要求填写集群基本信息，并点击 [下一步](#)。

- 集群名称：名称只包含小写字母、数字和连字符（“-”），必须以小写字母或者数字开头和结尾，最长 63 个字符。
- 被纳管：选择由哪个集群来管理此集群，例如在[集群生命周期](#)中创建、升级、节点扩缩容、删除集群等。
- 运行时：选择集群的运行时环境，目前支持 containerd 和 Docker，[如何选](#)

[选择容器运行时。](#)

- Kubernetes 版本：支持 3 个版本跨度，具体取决于被纳管集群所支持的版本。



### 填写基本信息

3. 填写节点配置信息，并点击 **下一步**。

- 高可用：开启后需要提供至少 3 个控制器节点。关闭后，只提供 1 个控制器节点即可。

生产环境中建议使用高可用模式。

- 认证方式：选择通过用户名/密码还是公私钥访问节点。

如果使用公私钥方式访问节点，需要预先配置节点的 SSH 密钥。参阅[使用 SSH 密钥认证节点](#)。

- 使用统一的密码：开启后集群中所有节点的访问密码都相同，需要在下方输入访问所有节点的统一密码。如果关闭，则可以为每个节点设置单独的用户名和密码。

- 节点信息：填写节点名称和 IP 地址。

- 自定义参数：设置变量控制 Ansible 与远程主机交互。可设置变量参考[连接](#)

#### 到主机：行为清单参数

- NTP 时间同步：开启后会自动同步各个节点上的时间，需要提供 NTP 服务  
器地址。

#### 节点配置

##### 节点配置

4. 在页面底部点击节点检查。如果检查通过则继续下一步操作。如果检查未通过，则更

新 节点信息 并再次执行检查。

5. 填写网络配置信息，并点击 **下一步**。

- 网络插件：负责为集群内的 Pod 提供网络服务，创建集群后不可更改网络  
插件。支持 [cilium](#) 和 [calico](#)。选择 **none** 表示暂不安装网络插件。

有关网络插件的参数配置，可参考 [cilium 安装参数配置](#) 或 [calico 安装参数  
配置](#)。

- 容器网段：集群下容器使用的网段，决定集群下容器的数量上限。创建后不  
可修改。

- 服务网段：同一集群下容器互相访问时使用的 Service 资源的网段，决定  
Service 资源的上限。创建后不可修改。

#### 网络配置 1

##### 网络配置 1

#### 网络配置 2

##### 网络配置 2

6. 填写插件配置信息，并点击 **下一步**。

## 插件配置

### 插件配置

7. 填写高级配置信息，并点击 **确定**。

- **kubelet\_max\_pods**：设置每个节点的最大 Pod 数量，默认为 110 个。
- **hostname\_overide**：重置主机名，建议使用默认值，采用系统默认生成的名称作为主机名称。
- **kubernetes\_audit**：Kubernetes 的审计日志，默认开启。
- **auto\_renew\_certificate**：在每月第一个星期一自动更新 Kubernetes 控制平面证书，默认开启。
- **disable\_firewalld&ufw**：禁用防火墙，避免节点在安装过程中无法被访问。
- **Insecure\_registries**：私有镜像仓库配置。使用私有镜像仓库创建集群时，为了避免证书问题导致容器引擎拒绝访问，需要在这里填写私有镜像仓库地址，以绕过容器引擎的证书认证而获取镜像。
- **yum\_repos**：填写 Yum 源仓库地址。离线环境下，默认给出的地址选项仅供参考，请根据实际情况填写。

## 高级配置

### 高级配置

#### !!! success

- 填写正确信息并完成上述步骤后，页面会提示集群正在创建中。
- 创建集群耗时较长，需要耐心等待。其间，可以点击 返回集群列表 按钮让安装后台运行。
- 如需查看当前状态，可点击 实时日志。

![查看实时日志](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/create009.png)

#### !!! note

- 当集群出现未知状态时，表示当前集群已失联。
- 系统展示数据为失联前缓存数据，不代表真实数据。

- 同时失联状态下执行的任何操作都将不生效，请检查集群网络连通性或主机状态。

![未知状态](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/createnew07.png)

## 参考文档

- [在 CentOS 管理平台上创建 Ubuntu 工作集群](#)
- [在 CentOS 管理平台上创建 RedHat 9.2 工作集群](#)
- [在非主流操作系统上创建集群](#)

## 接入集群

通过接入集群操作，能够对众多云服务平台集群和本地私有物理集群进行统一纳管，形成统一治理平台，有效避免了被厂商锁定风险，助力企业业务安全上云。

容器管理模块支持接入多种主流的容器集群，例如 DaoCloud KubeSpray, DaoCloud ClusterAPI, DaoCloud Enterprise 4.0、Redhat Openshift, SUSE Rancher, VMware Tanzu, Amazon EKS, Aliyun ACK, Huawei CCE, Tencent TKE, 标准 Kubernetes 集群。

## 前提条件

- 准备一个待接入的集群，确保容器管理集群和待接入集群之间网络通畅，并且集群的 Kubernetes 版本 1.22+。
- 当前操作用户应具有 [Kpanda Owner](#) 或更高权限。

## 操作步骤

1. 进入 **集群列表** 页面，点击右上角的 **接入集群** 按钮。

## 接入集群

### 接入集群

#### 2. 填写基本信息。

- 集群名称：名称应具有唯一性，设置后不可更改。最长 63 个字符，只能包含小写字母、数字及分隔符("-")，且必须以小写字母或数字开头及结尾。
- 集群别名：可输入任意字符，不超过 60 个字符。
- 发行版：集群的发行厂商，包括市场主流云厂商和本地私有物理集群。

#### 3. 填写目标集群的 KubeConfig，点击 验证 Config，验证通过后才能成功接入集群。

如果不知道如何获取集群的 KubeConfig 文件，可以在输入框右上角点击 **如何获取 kubeConfig** 查看对应步骤。

## 接入集群

### 接入集群

#### 4. 确认所有参数填写正确，在页面右下角点击 **确定**。

## 接入集群

### 接入集群

#### !!! note

- 新接入的集群状态为 接入中，接入成功后变为 运行中。
- 如果集群状态一直处于 接入中，请确认接入脚本是否在对应集群上执行成功。有关集群状态的更多详情，请参考[集群状态](cluster-status.md)。

## 参考文档

- [为工作集群添加异构节点](#)
- [DCE 4.0 -> DCE 5.0 有限场景迁移](#)

# 访问集群

使用 DCE 5.0 [容器管理](#) 平台接入或创建的集群，不仅可以通过 UI 界面直接访问，也可以通过其他两种方式进行访问控制：

- 通过 CloudShell 在线访问
- 下载集群证书后通过 kubectl 进行访问

!!! note

访问集群时，用户应具有 [Cluster Admin](./permissions/permission-brief.md) 权限或更高权限。

## 通过 CloudShell 访问

1. 在 [集群列表](#) 页选择需要通过 CloudShell 访问的集群，点击右侧的  操作图标并在下拉列表中点击 **控制台**。

调用 CloudShell 控制台

调用 CloudShell 控制台

2. 在 CloudShell 控制台执行 **kubectl get node** 命令，验证 CloudShell 与集群的连通性。

如图，控制台将返回集群下的节点信息。

验证连通性

验证连通性

现在，您可以通过 CloudShell 来访问并管理该集群了。

!!! note

如果您想通过 CloudShell 访问 AWS 集群，须升级 kpanda 版本到 v0.29.x 以上，并手动调整 kpanda-apiserver 服务配置项，具体操作流程如下：

1. 进入 **全局服务集群**，在 **工作负载** -> **无状态负载** 列表中，找到 kpanda-apiserver 服务；

![kpanda-apiserver 服务](docs/zh/docs/kpanda/images/access-cluster-100.png)

1. 进入负载详情页，点击页面右上角 `⋮` 并在下拉列表中点击 `更新`；

![更新 kpanda-apiserver 服务](docs/zh/docs/kpanda/images/access-cluster-101.png)

1. 进入向导的第 2 步 `容器配置`，找到 `生命周期 -> 启动命令` 中的参数 `--feature-gates=`，将其修改为 `--feature-gates=UseTokenInCloudShell=true`

![修改启动命令参数](docs/zh/docs/kpanda/images/access-cluster-102.png)

1. 修改完成后，进入 `kpanda-apiserver` 负载详情页，点击页面右上角 `⋮` 并在下拉列表中点击 `重启`；

![重启 kpanda-apiserver 服务](docs/zh/docs/kpanda/images/access-cluster-103.png)

以上操作步骤完成后，即可通过 CloudShell 访问 AWS 集群。

## 通过 `kubectl` 访问

通过本地节点访问并管理云端集群时，需要满足以下条件：

- 本地节点和云端集群的网络互联互通。
- 已经将集群证书下载到了本地节点。
- 本地节点已经安装了 `kubectl` 工具。关于详细的安装方式，请参阅安装 [kubectl](#)。

满足上述条件后，按照下方步骤从本地访问云端集群：

1. 在 **集群列表** 页选择需要下载证书的集群，点击右侧的 `⋮`，并在弹出菜单中点击 `证书获取`。

进入下载证书页面

进入下载证书页面

2. 选择证书有效期并点击 `下载证书`。

`下载证书`

`下载证书`

3. 打开下载好的集群证书，将证书内容复制至本地节点的 `config` 文件。

kubectl 工具默认会从本地节点的 `$HOME/.kube` 目录下查找名为 `config` 的文件。该文件存储了相关集群的访问凭证，kubectl 可以凭该配置文件连接至集群。

4. 在本地节点上执行如下命令验证集群的连通性：

```
kubectl get pod -n default
```

预期的输出类似于：

| NAME                          | READY | STATUS  | RESTARTS | AGE |
|-------------------------------|-------|---------|----------|-----|
| dao-2048-2048-58c7f7fc5-mq7h4 | 1/1   | Running | 0        | 30h |

现在您可以在本地通过 kubectl 访问并管理该集群了。

## 集群升级

Kubernetes 社区每个季度都会发布一次小版本，每个版本的维护周期大概只有 9 个月。

版本停止维护后就不会再更新一些重大漏洞或安全漏洞。手动升级集群操作较为繁琐，给管理人员带来了极大的工作负担。

本节将介绍如何在通过 Web UI 界面一键式在线升级工作集群 Kubernetes 版本，如需离线升级工作集群的 kubernetes 版本，请参阅[工作集群离线升级指南](#)进行升级。

!!! danger

版本升级后将无法回退到之前的版本，请谨慎操作。

!!! note

- Kubernetes 版本以 `_x.y.z_` 表示，其中 `_x_` 是主要版本，`_y_` 是次要版本，`_z_` 是补丁版本。
- 不允许跨次要版本对集群进行升级，例如不能从 1.23 直接升级到 1.25。
- `_接入集群_` 不支持版本升级。如果左侧导航栏没有 `_集群升级_`，请检查该集群是否为 `_接入集群_`。
- 全局服务集群只能通过终端进行升级。
- 升级工作集群时，该工作集群的[管理集群](cluster-role.md#\_3)应该已经接入容器管理模块，并且处于正常运行中。
- 如果需要修改集群参数，可以通过升级相同版本的方式实现，具体操作参考下文。

1. 在集群列表中点击目标集群的名称。

升级集群

升级集群

2. 然后在左侧导航栏点击 **集群运维 -> 集群升级**，在页面右上角点击 **版本升级**。

升级集群

升级集群

3. 选择可升级的版本，输入集群名称进行确认。

可升级版本

可升级版本

!!! note

如果您是想通过升级方式来修改集群参数，请参考以下步骤：

1. 找到集群对应的 ConfigMap，您可以登录控制节点执行如下命令，找到 varsConfRef 中的 ConfigMap 名称。

```
```shell
kubectl get cluster.kubeain.io <clusternname> -o yaml
```

```

2. 根据需要，修改 ConfigMap 中的参数信息。

3. 在此处选择相同版本进行升级操作，升级完成即可成功更新对应的集群参数。

4. 点击 **确定** 后，可以看到集群的升级进度。

升级进度

升级进度

5. 集群升级预计需要 30 分钟，可以点击 **实时日志** 按钮查看集群升级的详细日志。

实时日志

实时日志

## 卸载/解除接入集群

通过 DCE 5.0 [容器管理](#) 平台 创建的集群 支持 **卸载集群** 或 **解除接入** 操作，从其他环境  
上海道客网络科技有限公司

**直接 接入的集群 仅支持 解除接入 操作。**

#### !!! Info

如果想彻底删除一个接入的集群，需要前往创建该集群的原始平台操作。DCE 5.0 不支持删除接入的集群。

在 DCE 5.0 平台上，**卸载集群** 和 **解除接入** 的区别在于：

- **卸载集群** 操作会销毁该集群，并重置集群下所有节点的数据。所有数据都将被销毁，建议做好备份。后期需要时必须重新创建一个集群。

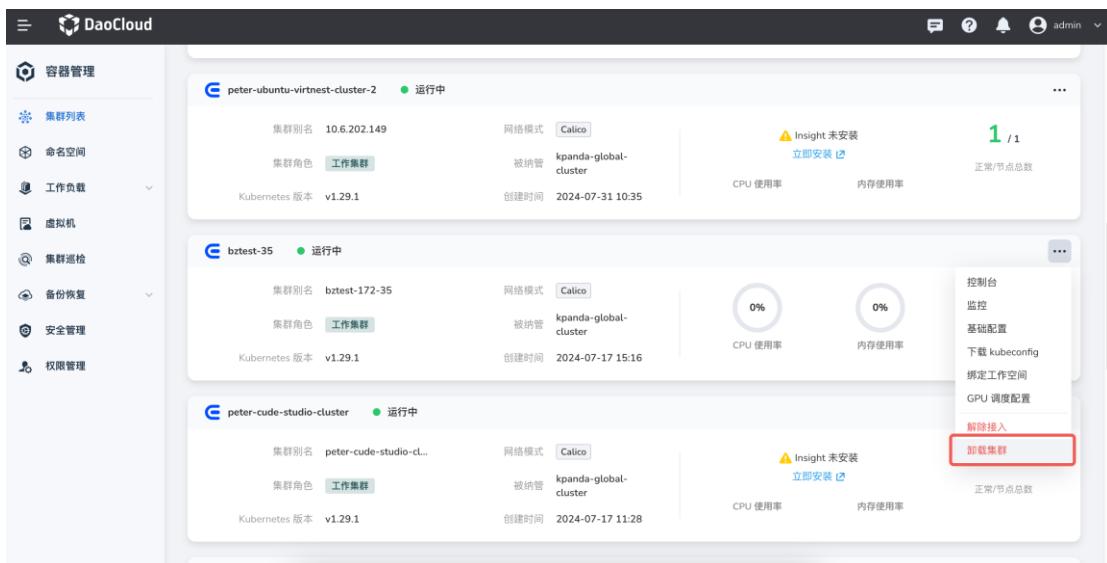
- **解除接入** 操作会将当前集群从平台中移除，不会摧毁集群，也不会销毁数据。

## 卸载集群

#### !!! note

- 当前操作用户应具备 [Admin](../../../../ghippo/user-guide/access-control/role.md) 或 [Kpanda Owner](../../../../ghippo/user-guide/access-control/global.md) 权限才能执行卸载集群的操作。
- 卸载集群之前，应该先在集群列表中点击某个集群名称，在 **集群运维** -> **集群设置** -> **高级配置** 中关闭 **集群删除保护**，否则不显示 **卸载集群** 的选项。
- **全局服务集群** 不支持卸载或移除操作。

1. 在 **集群列表** 页找到需要卸载集群，点击右侧的 并在下拉列表中点击 **卸载集群**。



点击删除按钮

2. 输入集群名称进行确认，然后点击 **删除**。



**确认删除**

如果提示集群中还有一些残留的资源，则需要按提示删除相关资源后才能执行卸载操作。

3. 返回 **集群列表** 页可以看到该集群的状态已经变成 **删除中**。卸载集群可能需要一段时间，请您耐心等候。

**删除中状态**

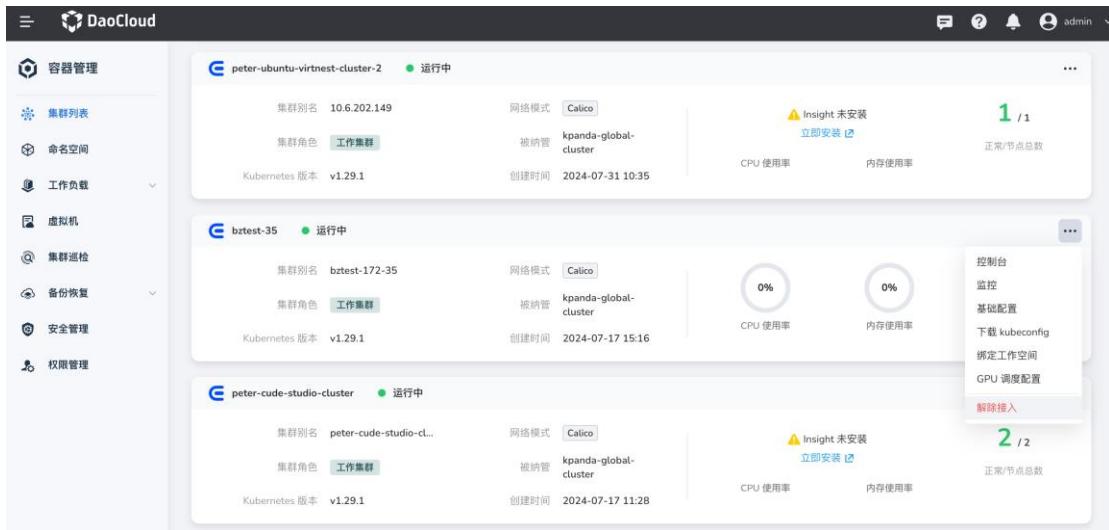
**删除中状态**

## 解除接入集群

### !!! note

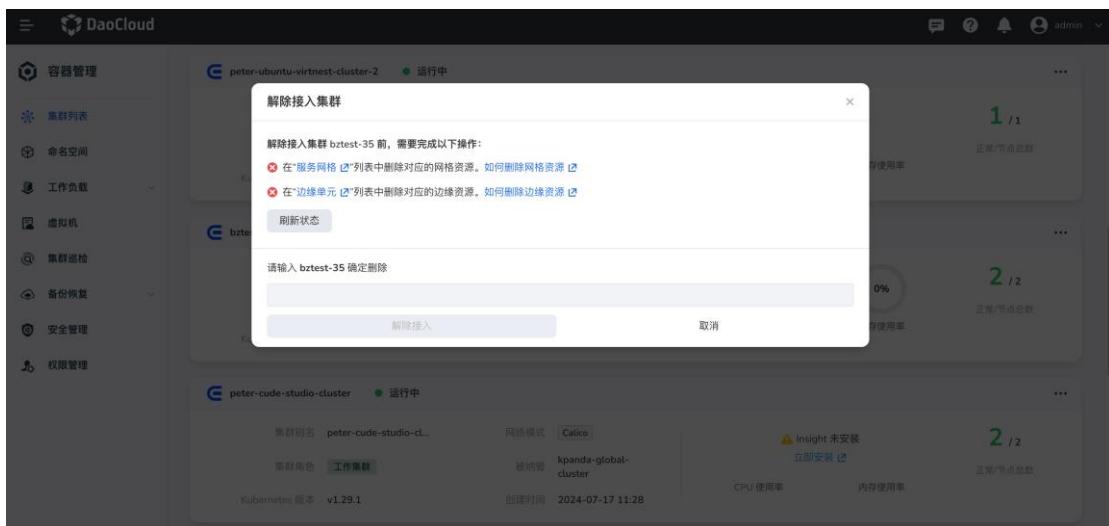
- 当前操作用户应具备 [Admin](../../../../ghippo/user-guide/access-control/role.md) 或 [Kpanda Owner](../../../../ghippo/user-guide/access-control/global.md) 权限才能执行解除接入的操作。
- 全局服务集群 不支持解除接入。

1. 在 **集群列表** 页找到需要卸载集群，点击右侧的 并在下拉列表中点击 **解除接入**。



点击解除接入按钮

2. 输入集群名称进行确认，然后点击 **解除接入**。



确认删除

如果提示集群中还有一些残留的资源，则需要按提示删除相关资源后才能解除接入。

## 清理解除接入集群配置数据

集群被移除后，集群中原有的管理平台数据不会被自动清除，如需将集群接入至新管理平

台则需要手动执行如下操作：

1. 删除 kpanda-system、insight-system 命名空间

```
kubectl delete ns kpanda-system insight-system
```

2. 如果为当前集群启用了服务网格能力 , 请参考[删除网格](#)文档删除当前集群中的网格实例。

## 集群角色

DaoCloud Enterprise 5.0 基于集群的不同功能定位对集群进行了角色分类 , 帮助用户更好地管理 IT 基础设施。

### 全局服务集群

此集群用于运行 DCE 5.0 组件 , 例如[容器管理](#)、[全局管理](#)、[可观测性](#)、[镜像仓库](#)等。一般不承载业务负载。

| 支持的功能     | 描述                                                                                                                                                   |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| K8s 版本    | 1.22+                                                                                                                                                |
| 操作系统      | RedHat 7.6 x86/ARM, RedHat 7.9 x86, RedHat 8.4 x86/ARM, RedHat 8.6 x86 ; Ubuntu 18.04 x86, Ubuntu 20.04 x86 ; CentOS 7.6 x86/AMD, CentOS 7.9 x86/AMD |
| 集群全生命周期管理 | 支持                                                                                                                                                   |
| K8s 资源管理  | 支持                                                                                                                                                   |
| 云原生存储     | 支持                                                                                                                                                   |
| 云原生网络     | Calico、Cilium、Multus 和其它 CNI                                                                                                                         |

| 支持的功能 | 描述                         |
|-------|----------------------------|
| 策略管理  | 支持网络策略、配额策略、资源限制、灾备策略、安全策略 |

## 管理集群

此集群用于管理工作集群，一般不承载业务负载。

- 经典模式将全局服务集群和管理集群部署在不同的集群，适用于企业多数据中心、多架构的场景。
- 简约模式将管理集群和全局服务集群部署在同一个集群内。

| 支持的功能     | 描述                                                                                                                                                   |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| K8s 版本    | 1.22+                                                                                                                                                |
| 操作系统      | RedHat 7.6 x86/ARM, RedHat 7.9 x86, RedHat 8.4 x86/ARM, RedHat 8.6 x86 ; Ubuntu 18.04 x86, Ubuntu 20.04 x86 ; CentOS 7.6 x86/AMD, CentOS 7.9 x86/AMD |
| 集群全生命周期管理 | 支持                                                                                                                                                   |
| K8s 资源管理  | 支持                                                                                                                                                   |
| 云原生存储     | 支持                                                                                                                                                   |
| 云原生网络     | Calico、Cilium、Multus 和其它 CNI                                                                                                                         |
| 策略管理      | 支持网络策略、配额策略、                                                                                                                                         |

| 支持的功能 | 描述           |
|-------|--------------|
|       | 资源限制、灾备策略、安全 |
|       | 策略           |

## 工作集群

这是使用[容器管理](#)创建的集群，主要用于承载业务负载。该集群由管理集群进行管理。

| 支持的功能     | 描述                                                                                                                                                         |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| K8s 版本    | 支持 K8s 1.22 及以上版本                                                                                                                                          |
| 操作系统      | RedHat 7.6 x86/ARM, RedHat 7.9 x86, RedHat 8.4 x86/ARM,<br>RedHat 8.6 x86 ; Ubuntu 18.04 x86, Ubuntu 20.04<br>x86 ; CentOS 7.6 x86/AMD, CentOS 7.9 x86/AMD |
| 集群全生命周期管理 | 支持                                                                                                                                                         |
| K8s 资源管理  | 支持                                                                                                                                                         |
| 云原生存储     | 支持                                                                                                                                                         |
| 云原生网络     | Calico、Cilium、Multus 和其它 CNI                                                                                                                               |
| 策略管理      | 支持网络策略、配额策略、资源限制、灾备策略、安全策略                                                                                                                                 |

## 接入集群

此集群用于接入已有的标准 K8s 集群，包括但不限于本地数据中心自建集群、公有云厂商提供的集群、私有云厂商提供的集群、边缘集群、信创集群、异构集群、Daocloud 不同发行版集群。主要用于承担业务负载。

| 支持的功能     | 描述                                                                                                |
|-----------|---------------------------------------------------------------------------------------------------|
| K8s 版本    | 1.18+                                                                                             |
| 支持友商      | Vmware Tanzu、Amazon EKS、Redhat Openshift、SUSE Rancher、阿里 ACK、华为 CCE、腾讯 TKE、标准 K8s 集群、Daocloud DCE |
| 集群全生命周期管理 | 不支持                                                                                               |
| K8s 资源管理  | 支持                                                                                                |
| 云原生存储     | 支持                                                                                                |
| 云原生网络     | 依赖于接入集群发行版网络模式                                                                                    |
| 策略管理      | 支持网络策略、配额策略、资源限制、灾备策略、安全策略                                                                        |

### !!! note

一个集群可以有多个集群角色，例如一个集群既可以是全局服务集群，也可以是管理集群或工作集群。

# 集群状态

容器管理模块支持纳管两种类型的集群：接入集群和自建集群。 关于集群纳管类型更多的信息，请参见[集群角色](#)。

这两种集群的状态如下所述。

## 接入集群

| 状态                    | 描述                                                                         |
|-----------------------|----------------------------------------------------------------------------|
| 接入中（Joining）          | 集群正在接入                                                                     |
| 解除接入中<br>( Removing ) | 集群正在解除接入                                                                   |
| 运行中（Running）          | 集群正常运行                                                                     |
| 未知（Unknown）           | 集群已失联，系统展示数据为失联前缓存数据， <b>不代表真实数据</b> ，同时失联状态下执行的任何操作都将不生效，请检查集群网络连通性或主机状态。 |

## 自建集群

| 状态            | 描述                 |
|---------------|--------------------|
| 创建中（Creating） | 集群正在创建             |
| 更新中（Updating） | 更新集群 Kubernetes 版本 |
| 删除中（Deleting） | 集群正在删除             |
| 运行中（Running）  | 集群正常运行             |

| 状态              | 描述                                                                |
|-----------------|-------------------------------------------------------------------|
| 未知 ( Unknown )  | 集群已失联，系统展示数据为失联前缓存数据，不代表真实数据，同时失联状态下执行的任何操作都将不生效，请检查集群网络连通性或主机状态。 |
| 创建失败 ( Failed ) | 集群创建失败，请查看日志以获取详细失败原因                                             |

## 如何选择容器运行时

容器运行时是 kubernetes 中对容器和容器镜像生命周期进行管理的重要组件。

kubernetes 在 1.19 版本中将 containerd 设为默认的容器运行时，并在 1.24 版本中移除了 Dockershim 组件的支持。

因此相较于 Docker 运行时，我们更加 **推荐您使用轻量的 containerd 作为您的容器运行时**，因为这已经成为当前主流的运行时选择。

除此之外，一些操作系统发行厂商对 Docker 运行时的兼容也不够友好，不同操作系统对运行时的支持如下表：

### 不同操作系统和推荐的运行时版本对应关系

| 操作系统     | 推荐的 containerd 版本 | 推荐的 Docker 版本                                 |
|----------|-------------------|-----------------------------------------------|
| CentOS   | 1.7.5             | 20.10                                         |
| RedHatOS | 1.7.5             | 20.10                                         |
| KylinOS  | 1.7.5             | 19.03 (仅 ARM 架构支持，在 x86 架构下不支持使用 Docker 作为运行) |

|             |                          |                      |
|-------------|--------------------------|----------------------|
| <b>操作系统</b> | <b>推荐的 containerd 版本</b> | <b>推荐的 Docker 版本</b> |
|             |                          | 时 )                  |

更多支持的运行时版本信息，请参考 [RedHatOS 支持的运行时版本](#) 和 [KylinOS 支持的运](#)

### 行时版本

!!! note

在离线安装模式下，需要提前准备相关操作系统的运行时离线包。

## 集群版本支持范围

在 DCE 5.0 平台中，[接入型集群](#)和[自建集群](#)采取不同的版本支持机制。

本文主要介绍自建集群的版本支持机制。

Kubernetes 社区支持 3 个版本范围，如 1.26、1.27、1.28。当社区新版本发布之后，支持的版本范围将会进行递增。如社区最新的 1.29 版本已经发布，此时社区支持的版本范围是 1.27、1.28、1.29。

为了保障集群的安全和稳定性，在 DCE 5.0 中使用界面创建集群时支持的版本范围将始终比社区版本 **低一个版本**。

例如，社区支持的版本范围是 1.25、1.26、1.27，则在 DCE 5.0 中使用界面创建工作集群的版本范围是 1.24、1.25、1.26，并且会为用户推荐一个稳定的版本，如 1.24.7。

除此之外，DCE 5.0 中使用界面创建工作集群的版本范围与社区保持高度同步，当社区版本进行递增后，DCE 5.0 中使用界面创建工作集群的版本范围也会同步递增一个版本。

## Kubernetes 版本支持范围

Kubernetes 社区版本范围

自建工作集群版本范围

## 自建工作集群推荐版本

### DCE 5.0 安装器

发布时间

1.26  
1.27  
1.28  
1.25  
1.26  
1.27  
1.27.5  
v0.13.0  
2023.11.30

版本支持机制

版本支持机制

# 接入 rancher 集群

本文介绍如何接入 rancher 集群。

## 前提条件

- 准备一个具有管理员权限的待接入 rancher 集群，确保容器管理集群和待接入集群之间网络通畅。
- 当前操作用户应具有 [Kpanda Owner](#) 或更高权限。

## 操作步骤

### 步骤一：在 rancher 集群创建具有管理员权限的 ServiceAccount 用户

1. 使用具有管理员权限的角色进入 rancher 集群，并使用终端新建一个名为 `sa.yaml` 的文件。

```
vi sa.yaml
```

然后按下 `i` 键进入插入模式，输入以下内容：

```
```yaml title="sa.yaml" apiVersion: rbac.authorization.k8s.io/v1 kind: ClusterRole metadata:
  name: rancher-rke rules:
    - apiGroups:
        - '*'
    - resources:
        - '*'
    - verbs:
        - '*'
    - nonResourceURLs:
        - '*'
    - verbs:
        - '*'
    - apiVersion: rbac.authorization.k8s.io/v1 kind: ClusterRoleBinding metadata:
      name: rancher-rke roleRef: apiGroup: rbac.authorization.k8s.io kind: ClusterRole
      name: rancher-rke subjects:
        - kind: ServiceAccount name: rancher-rke namespace: kube-system — apiVersion:
          v1 kind: ServiceAccount metadata: name: rancher-rke namespace: kube-system
        ...
```

```

按下 `esc` 键退出插入模式，然后输入 `__:wq__` 保存并退出。

2. 在当前路径下执行如下命令，新建名为 `rancher-rke` 的 ServiceAccount（以下简称为 **SA**）：

```
kubectl apply -f sa.yaml
```

预期输出如下：

```
clusterrole.rbac.authorization.k8s.io/rancher-rke created
clusterrolebinding.rbac.authorization.k8s.io/rancher-rke created
serviceaccount/rancher-rke created
```

3. 创建名为 `rancher-rke-secret` 的密钥，并将密钥和 `rancher-rke` SA 绑定。

```
kubectl apply -f - <<EOF
apiVersion: v1
```

```

kind: Secret
metadata:
 name: rancher-rke-secret
 namespace: kube-system
 annotations:
 kubernetes.io/service-account.name: rancher-rke
 type: kubernetes.io/service-account-token
EOF

```

预期输出如下：

```
secret/rancher-rke-secret created
```

**!!! note**

如果您的集群版本低于 1.24，请忽略此步骤，直接前往下一步。

#### 4. 查找 **rancher-rke** SA 的密钥：

```
kubectl -n kube-system get secret | grep rancher-rke | awk '{print $1}'
```

预期输出：

```
rancher-rke-secret
```

查看密钥 **rancher-rke-secret** 的详情：

```
kubectl -n kube-system describe secret rancher-rke-secret
```

预期输出：

```
Name: rancher-rke-secret
Namespace: kube-system
Labels: <none>
Annotations: kubernetes.io/service-account.name: rancher-rke
 kubernetes.io/service-account.uid: d83df5d9-bd7d-488d-a046-b740618a0174

Type: kubernetes.io/service-account-token
```

Data

=====

```
ca.crt: 570 bytes
namespace: 11 bytes
token: eyJhbGciOiJSUzI1NiIsImtpZCI6IjUtNE9nUWZLRzVpbEJORkZaNmtCQXhq
VzRsZHU4MHhHcDBfb0VCaUo0V1kifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2
NvdW50Iiwia3ViZXJuZXRLcy5pb9y9zZXJ2aWN1YWNg3VudC9uYW1lc3BhY2UiOiJrdWJl
LXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbW
UiOiJyYW5jaGVyLXRlzs1zZWNgZXQiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2Nvd
W50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoicmFuY2hlci1ya2UiLCJrdWJlcm5ldGVzLmlv
L3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiOjKODNkZjVkOS1iZDd
kLTQ4OGQtYTA0Ni1iNzQwNjE4YTAxNzQiLCJzdWIiOiJzeXN0ZW06c2VydmljZWFjY2
```

91bnQ6a3ViZS1zeXN0ZW06cmFuY2hlci1ya2UifQ.VNsMtPEFOdDDeGt\_8VHblcMRvjOw  
 PXMM-79o9UooHx6q-VkHOcIOP3FOT2hnEdNnIsyODZVKCpEdCgyozX-3y5x2cZSZpocn  
 kMcBbQm-qfTyUcUhAY7N5gcYUtHUhvRAsNWJcsDCn6d96gT\_qo-ddo\_cT8Ri39Lc123F  
 DYOnYG-YGFKSgRQV7Vv34HiajZCCjZzy7i--eE\_7o4DXeTjNqAFMFstUxxHBOXI3R  
 dn1zKQKqh5Jhg4ES7X-edSviSUfJUX-QV\_LlAw5DuAyGPH7bDH4QaQ5k-p6cIctmpWZE  
 -9wRDIKA4LYRblKE7MJcI6OmM4ldlMM0Jc8N-gCtl4w

## 步骤二：在本地使用 rancher-rke SA 的认证信息更新 kubeconfig 文件

在任意一台安装了 `kubelet` 的本地节点执行如下操作：

### 1. 配置 kubelet token：

`kubectl config set-credentials rancher-rke --token=` `rancher-rke-secret` 里面的 token 信息

例如：

```
kubectl config set-credentials eks-admin --token=eyJhbGciOiJSUzI1NiIsImtpZCI6IjUtNE9nU

WZLRzVpbEJORkZaNmtCQXhqVzRsZHU4MHhHcDBfb0VCaUo0V1kifQ.eyJpc3MiOiJrd

WJlcm5ldGVzL3NlcnPY2VhY2NvdW50Iiwia3ViZXJuZXRLcy5pb9zZXJ2aWNlYWNjb3

VudC9uYW1lc3BhY2UiOjJrdWJlXN5c3RlbSIsImt1YmVybmv0ZXMuaw8vc2VydmljZ

WFjY291bnQvc2VjcmV0Lm5hbWUiOjJyYW5jaGVyLXJrZS1zZWNyZXQiLCJrdWJlcm5l

dGVzLmlvL3NlcnPY2VhY2NvdW50L3NlcnPY2UtYWNjb3VudC5uYW1IIjoicmFuY2hl

ci1ya2UiLCJrdWJlcm5ldGVzLmlvL3NlcnPY2VhY2NvdW50L3NlcnPY2UtYWNjb3Vud

C51aWQiOjJkODNkZjVkJOS1iZDdkLTQ4OGQtYTA0Ni1iNzQwNjE4YTAXNzQiLCJzdWI

iOjzeXN0ZW06c2VydmljZWFjY291bnQ6a3ViZS1zeXN0ZW06cmFuY2hlci1ya2UifQ.VN

sMtPEFOdDDeGt_8VHblcMRvjOwPXMM-79o9UooHx6q-VkHOcIOP3FOT2hnEdNnIsyO

DZVKCpEdCgyozX-3y5x2cZSZpocnkMcBbQm-qfTyUcUhAY7N5gcYUtHUhvRAsNWJcs

DCn6d96gT_qo-ddo_cT8Ri39Lc123FDYOnYG-YGFKSgRQV7Vv34HiajZCCjZzy7i--eE

_7o4DXeTjNqAFMFstUxxHBOXI3Rdn1zKQKqh5Jhg4ES7X-edSviSUfJUX-QV_LlAw5Du

AyGPH7bDH4QaQ5k-p6cIctmpWZE-9wRDIKA4LYRblKE7MJcI6OmM4ldlMM0Jc8N-gCt

l4w
```

### 2. 配置 kubelet APIServer 信息：

`kubectl config set-cluster {集群名} --insecure-skip-tls-verify=true --server={APIServer}`

- **{集群名}**：指 rancher 集群的名称。
- **{APIServer}**：指集群的访问地址，一般为集群控制节点 IP + 6443 端口，如  
`https://10.X.X.X:6443`

例如：

```
kubectl config set-cluster rancher-rke --insecure-skip-tls-verify=true --server=https://10.X.X.X:6443
```

### 3. 配置 kubelet 上下文信息：

```
kubectl config set-context {上下文名称} --cluster={集群名} --user={SA 用户名}
```

例如：

```
kubectl config set-context rancher-rke-context --cluster=rancher-rke --user=rancher-rke
```

### 4. 在 kubelet 中指定我们刚刚新建的上下文 **rancher-rke-context**：

```
kubectl config use-context rancher-rke-context
```

### 5. 获取上下文 **rancher-rke-context** 中的 kubeconfig 信息。

```
kubectl config view --minify --flatten --raw
```

预期输出：

```
apiVersion: v1
clusters:
- cluster:
 insecure-skip-tls-verify: true
 server: https://77C321BCF072682C70C8665ED4BFA10D.gr7.ap-southeast-1.eks.amazonaws.com
 name: joincluster
contexts:
- context:
 cluster: joincluster
 user: eks-admin
 name: ekscontext
 current-context: ekscontext
 kind: Config
 preferences: {}
users:
- name: eks-admin
 user:
 token: eyJhbGciOiJSUzI1NiIsImtpZCI6ImcxTjJwNkktWm5IbmRJu1RFRExvdWY1TGFWVUtGQ3VJejFtNlFQcUNFaIeifQ.eyJpc3MiOiJrdWJlcmt5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRLcy5pbby9zZXJ2aWNlYWNgjb3VudC9uYW1lc3BhY2UiOjrdWJILXN5c3RlbSIsImt1YmVybmV0ZXMuaw8vc2VydmljZWFjY291bnQvc2V
```

## 步骤三：在 DCE 界面接入集群

使用刚刚获取的 kubeconfig 文件，参考[接入集群](#)文档，将 rancher 集群接入全局服务集

群。

# 如何在集群中部署第二调度器 scheduler-plugins

本文介绍如何在集群中部署第二个调度器 scheduler-plugins。

## 为什么需要 scheduler-plugins ?

通过平台创建的集群中会安装 K8s 原生的调度器，但是原生的调度器存在很多的局限性：

- 原生的调度器无法满足调度需求，你可以选择使用 [CoScheduling](#)、[CapacityScheduling](#) 等 scheduler-plugins 插件。
- 在特殊的场景，需要新的调度器来完成调度任务而不影响原生调度器的流程。
- 区分不同功能的调度器，通过切换调度器名称来实现不同的调度场景。

本文以使用 vgpu 调度器的同时，想结合 scheduler-plugins 的 coscheduling 插件能力的场景为示例，介绍如何安装并使用 scheduler-plugins。

## 安装 scheduler-plugins

### 前置条件

- kubelet 是在 v0.13.0 版本推出的新功能，选择管理集群时请确保版本不低于此版本。
- 安装 scheduler-plugins 版本为 v0.27.8，请确保集群版本是否与它兼容。参考文档 [Compatibility Matrix](#)。

# 安装流程

## 1. 在 创建集群 -> 高级配置 -> 自定义参数 中添加 scheduler-plugins 参数

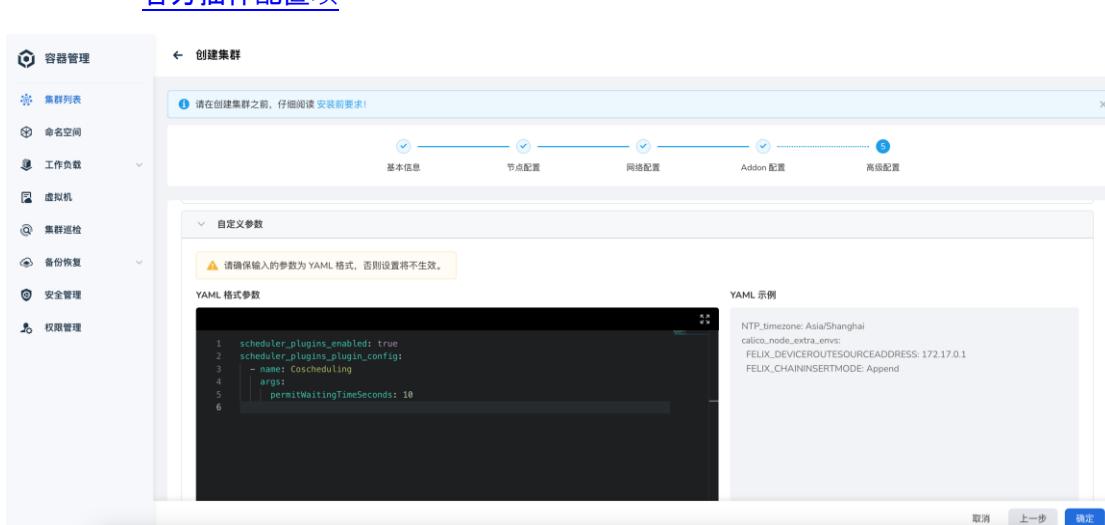
```

scheduler_plugins_enabled:true
scheduler_plugins_plugin_config:
 - name: Coscheduling
 args:
 permitWaitingTimeSeconds: 10 # default is 60

```

参数说明：

- scheduler\_plugins\_enabled 设置为 true 时，开启 scheduler-plugins 插件能力。
- 您可以 通过 设置 scheduler\_plugins\_enabled\_plugins 或 scheduler\_plugins\_disabled\_plugins 选项来启用或禁用某些插件。参阅 [K8s 官方插件名称](#)。
- 如果需要设置自定义插件的参数请配置 scheduler\_plugins\_plugin\_config，例如：设置 coscheduling 的 permitWaitingTimeSeconds 参数。参阅 [K8s 官方插件配置项](#)



添加 scheduler-plugins 参数

## 2. 集群创建成功后系统会自动安装 scheduler-plugins 和 controller 组件负载，可以在对

在应用集群的无状态负载中查看负载状态。

| 工作负载名称                       | 工作负载别名 | 状态  | 命名空间              | 容器组 (正常/总量) | 镜像                       | 创建时间             |
|------------------------------|--------|-----|-------------------|-------------|--------------------------|------------------|
| scheduler-plugins-controller | -      | 运行中 | scheduler-plugins | 1/1 个       | k8s.m.daocloud.io/sch... | 2024-02-23 15:30 |
| scheduler-plugins-scheduler  | -      | 运行中 | scheduler-plugins | 1/1 个       | k8s.m.daocloud.io/sch... | 2024-02-23 15:30 |

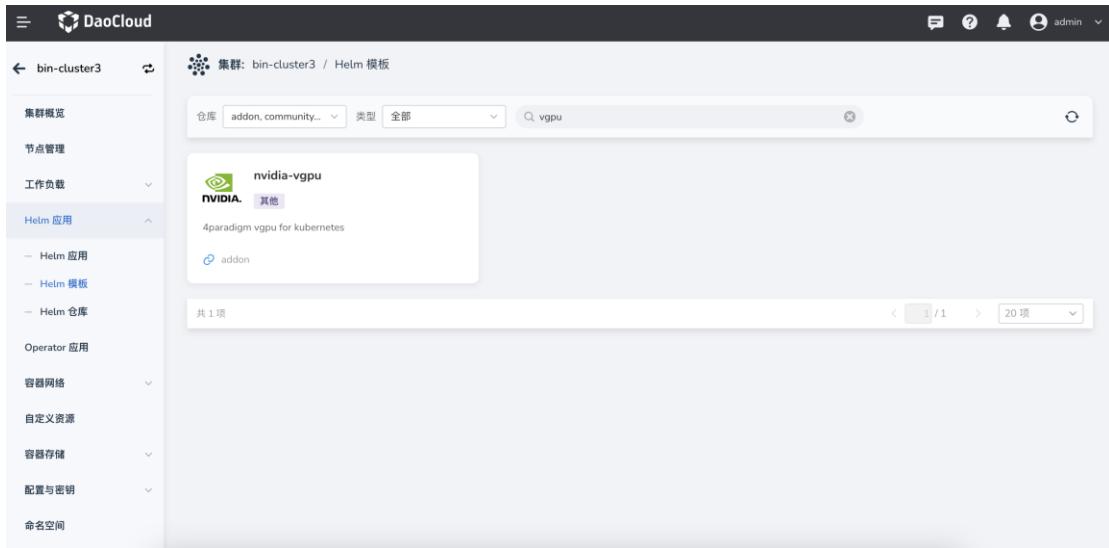
查看插件负载状态

## 使用 scheduler-plugins

以下以使用 vgpu 调度器的同时，想结合 scheduler-plugins 的 coscheduling 插件能力场景为示例，介绍如何使用 scheduler-plugins。

1. 在 Helm 模板中安装 vgpu，设置 values.yaml 参数。

- schedulerName: scheduler-plugins-scheduler，这是 kubelet 默认安装的 scheduler-plugins 的 scheduler 名称，目前不能修改。
- scheduler.kubeScheduler.enabled: false，不安装 kube-scheduler，将 vgpu-scheduler 作为单独的 extender。



## 安装 vgpu 插件

2. 在 scheduler-plugins 上扩展 vgpu-scheduler。

```
[root@master01 charts]# kubectl get cm -n scheduler-plugins scheduler-config -ojsonpath=".data.scheduler-config\yaml"
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
leaderElection:
 leaderElect: false
profiles:
 # Compose all plugins in one profile
 - schedulerName: scheduler-plugins-scheduler
 plugins:
 multiPoint:
 enabled:
 - name: Coscheduling
 - name: CapacityScheduling
 - name: NodeResourceTopologyMatch
 - name: NodeResourcesAllocatable
 disabled:
 - name: PrioritySort
 pluginConfig:
 - args:
 permitWaitingTimeSeconds: 10
 name: Coscheduling
```

修改 scheduler-plugins 的 scheduler-config 的 configmap 参数，如下：

```
[root@master01 charts]# kubectl get cm -n scheduler-plugins scheduler-config -ojsonpath=".data.scheduler-config\yaml"
```

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
leaderElection:
 leaderElect: false
profiles:
 # Compose all plugins in one profile
 - schedulerName: scheduler-plugins-scheduler
 plugins:
 multiPoint:
 enabled:
 - name: Coscheduling
 - name: CapacityScheduling
 - name: NodeResourceTopologyMatch
 - name: NodeResourcesAllocatable
 disabled:
 - name: PrioritySort
 pluginConfig:
 - args:
 permitWaitingTimeSeconds: 10
 name: Coscheduling
 extenders:
 - urlPrefix: "${urlPrefix}"
 filterVerb: filter
 bindVerb: bind
 nodeCacheCapable: true
 ignorable: true
 httpTimeout: 30s
 weight: 1
 enableHTTPS: true
 tlsConfig:
 insecure: true
 managedResources:
 - name: nvidia.com/vgpu
 ignoredByScheduler: true
 - name: nvidia.com/gpumem
 ignoredByScheduler: true
 - name: nvidia.com/gpucores
 ignoredByScheduler: true
 - name: nvidia.com/gpumem-percentage
 ignoredByScheduler: true
 - name: nvidia.com/priority
 ignoredByScheduler: true
 - name: cambricon.com/mlunum
 ignoredByScheduler: true
```

3. 安装完 `vgpu-scheduler` 后，系统会自动创建 `svc`，`urlPrefix` 指定 `svc` 的 URL。

**!!! note**

- `svc` 指 `pod` 服务负载，您可以到安装了 `nvidia-vgpu` 插件的命名空间下通过以下命令拿到 443 端口对应的外部访问信息。

```
```shell
kubectl get svc -n ${namespace}
```

```

- `urlprefix` 格式为 `https://\${ip 地址}:\${端口}`

4. 将 `scheduler-plugins` 的 `scheduler Pod` 重启，加载新的配置文件。

**!!! note**

在创建 `vgpu` 应用时不需要指定调度器名称，`vgpu-scheduler` 的 `Webhook` 会自动将 `Scheduler` 的名称修改为 `scheduler-plugins-scheduler`，不用手动指定。

## Kubernetes 集群证书更新

为保证 Kubernetes 各组件之间的通信安全，组件之间的调用会进行 TLS 身份验证，执行验证操作需要配置集群 PKI 证书。

集群证书有效期为 1 年，为避免证书过期导致业务无法使用，请及时更新证书。

本文介绍如何手动进行证书更新。

### 检查证书是否过期

您可以执行以下命令查看证书是否过期：

```
kubeadm certs check-expiration
```

输出类似于以下内容：

| CERTIFICATE                      | EXPIRES                 | RESIDUAL TIME | CERTIFIC |
|----------------------------------|-------------------------|---------------|----------|
| ATE AUTHORITY EXTERNALLY MANAGED |                         |               |          |
| admin.conf                       | Dec 14, 2024 07: 26 UTC | 204d          |          |
| no                               |                         |               |          |
| apiserver                        | Dec 14, 2024 07: 26 UTC | 204d          | ca       |
| no                               |                         |               |          |
| apiserver-etcd-client            | Dec 14, 2024 07: 26 UTC | 204d          | etcd-ca  |

```

no
apiserver-kubelet-client Dec 14, 2024 07: 26 UTC 204d ca
no
controller-manager.conf Dec 14, 2024 07: 26 UTC 204d
no
etcd-healthcheck-client Dec 14, 2024 07: 26 UTC 204d etcd-ca
no
etcd-peer Dec 14, 2024 07: 26 UTC 204d etcd-ca
no
etcd-server Dec 14, 2024 07: 26 UTC 204d etcd-ca
no
front-proxy-client Dec 14, 2024 07: 26 UTC 204d front-proxy-ca
no
scheduler.conf Dec 14, 2024 07: 26 UTC 204d
no

```

| CERTIFICATE AUTHORITY<br>LLY MANAGED | EXPIRES                 | RESIDUAL TIME | EXTERNAL |
|--------------------------------------|-------------------------|---------------|----------|
| ca                                   | Dec 12, 2033 07: 26 UTC | 9y            | no       |
| etcd-ca                              | Dec 12, 2033 07: 26 UTC | 9y            | no       |
| front-proxy-ca                       | Dec 12, 2033 07: 26 UTC | 9y            | no       |

## 手动更新证书

您可以通过以下命令手动更新证书，只需带上合适的命令行选项。更新证书前请先备份当前证书。

**更新指定证书：**

```
kubeadm certs renew
```

**更新全部证书：**

```
kubeadm certs renew all
```

更新后的证书可以在 /etc/kubernetes/pki 目录下查看，有效期延续 1 年。以下对应的几个配置文件也会同步更新：

- /etc/kubernetes/admin.conf
- /etc/kubernetes/controller-manager.conf
- /etc/kubernetes/scheduler.conf

**!!! note**

- 如果您部署的是一个高可用集群，这个命令需要在所有控制节点上执行。
- 此命令用 CA（或者 front-proxy-CA）证书和存储在 `/etc/kubernetes/pki` 中的密钥执行更新。

## 重启服务

执行更新操作之后，你需要重启控制面 Pod。因为动态证书重载目前还不被所有组件和证书支持，所有这项操作是必须的。

静态 Pod 是被本地 kubelet 而不是 API 服务器管理，所以 kubectl 不能用来删除或重启他们。

要重启静态 Pod，你可以临时将清单文件从 /etc/kubernetes/manifests/ 移除并等待 20 秒。

参考 [KubeletConfiguration 结构](#)中的 fileCheckFrequency 值。

如果 Pod 不在清单目录里，kubelet 将会终止它。在另一个 fileCheckFrequency 周期之后你可以将文件移回去，kubelet 可以完成 Pod 的重建，而组件的证书更新操作也得以完成。

```
mv ./manifests/* ./temp/
mv ./temp/* ./manifests/
```

!!! note

如果容器服务使用的是 Docker，为了让证书生效，可以使用以下命令对涉及到证书使用的几个服务进行重启：

```
```shell  
docker ps | grep -E 'k8s_kube-apiserver|k8s_kube-controller-manager|k8s_kube-scheduler|k8s_etcd_  
etcd' | awk '{print $1}' | xargs docker restart  
```
```

## 更新 KubeConfig

构建集群时通常会将 `admin.conf` 证书复制到 `$HOME/.kube/config` 中，为了在更新 `admin.conf` 后更新 `$HOME/.kube/config` 的内容，必须运行以下命令：

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

## 为 kubelet 配置证书轮换

完成以上操作后，基本完成了集群所有证书的更新，但不包括 kubelet。

因为 kubernetes 包含特性 kubelet 证书轮换，在当前证书即将过期时，将自动生成新的秘钥，并从 Kubernetes API 申请新的证书。一旦新的证书可用，它将被用于与 Kubernetes API 间的连接认证。

!!! note

此特性适用于 Kubernetes 1.8.0 或更高的版本。

启用客户端证书轮换，配置参数如下：

- kubelet 进程接收 `--rotate-certificates` 参数，该参数决定 kubelet 在当前使用的证书即将到期时，是否会自动申请新的证书。
- kube-controller-manager 进程接收 `--cluster-signing-duration` 参数（在 1.19 版本之前为 `--experimental-cluster-signing-duration`），用来控制签发证书的有效期限。

更多详情参考[为 kubelet 配置证书轮换](#)。

## 自动更新证书

为了更高效便捷处理已过期或者即将过期的 kubernetes 集群证书，可参考[k8s 版本集群证书更新](#)。

## 命名空间

命名空间是 Kubernetes 中用来进行资源隔离的一种抽象。一个集群下可以包含多个不重名的命名空间，每个命名空间中的资源相互隔离。有关命名空间的详细介绍，可参考[命名空间](#)。

本文将介绍命名空间的相关操作。

## 创建命名空间

支持通过表单轻松创建命名空间，也支持通过编写或导入 YAML 文件快速创建命名空间。

### !!! note

- 在创建命名空间之前，需要在容器管理模块[接入 Kubernetes 集群](./clusters/integrate-cluster.md)或者[创建 Kubernetes 集群](./clusters/create-cluster.md)。
- 集群初始化后通常会自动生成默认的命名空间 `_default`。但对于生产集群而言，为便于管理，建议创建其他的命名空间，而非直接使用 `_default` 命名空间。

## 表单创建

1. 在 **集群列表** 页面点击目标集群的名称。

集群详情

2. 在左侧导航栏点击 **命名空间**，然后点击页面右侧的 **创建** 按钮。

The screenshot shows the 'Namespaces' section of the DaoCloud interface. On the left, there's a sidebar with various cluster management sections like Cluster Overview, Node Management, Workload Management, Helm Applications, Operator Applications, Container Network, Custom Resources, Container Storage, Configuration and Secrets, and Namespace Management. The 'Namespace Management' section is currently selected. The main area displays a table of existing namespaces, each with columns for Name, Status, Workspace, Labels, and Creation Time. A search bar at the top allows filtering by namespace name. At the top right of the table, there are several buttons: 'Edit', 'YAML Create', and a prominent blue 'Create' button which is highlighted with a red box.

点击创建

3. 填写命名空间的名称，配置工作空间和标签（可选设置），然后点击 确定。

#### !!! info

- 命名空间绑定工作空间之后，该命名空间的资源就会共享给所绑定的工作空间。有关工作空间的详细说明，可参考[工作空间与层级](../../../../ghippo/user-guide/workspace/workspace.md)。
- 命名空间创建完成后，仍然可以绑定/解绑工作空间。

This screenshot shows the 'Create Namespace' dialog box. It has a form with fields for 'Name' (set to 'ns-01'), 'Labels' (containing 'ns-01' and 'demo', with a '+ Add' button), and a 'Create' button. In the background, the main namespace list table is visible, showing other existing namespaces like 'cert-manager', 'cro-test', etc. The 'Create' button in the dialog is highlighted with a blue box.

填写表单

4. 点击 确定，完成命名空间的创建。在命名空间列表右侧，点击 ，可以从弹出菜单中选择查看 YAML、修改标签、绑定/解绑工作空间、配额管理、删除等更多操作。

作。

The screenshot shows the 'Namespace' section of the DaoCloud interface. On the right, there is a table listing various namespaces with columns for Name, Status, WorkSpace, Labels, and Creation Time. A specific row for 'cert-manager' is selected. A context menu is open over this row, with a red box highlighting the 'YAML 创建' (Create YAML) option.

[更多操作](#)

## YAML 创建

1. 在 **集群列表** 页面点击目标集群的名称。

The screenshot shows the 'Cluster List' page. It displays two clusters: 'kpanda-e2e-cluster' and 'kpanda-global-cluster'. The 'kpanda-global-cluster' entry is highlighted with a red box. The page includes a search bar, filter buttons for 'Cluster' and 'Cluster Type', and a 'Create Cluster' button.

[集群详情](#)

2. 在左侧导航栏点击 **命名空间**，然后点击页面右侧的 **YAML 创建** 按钮。

The screenshot shows the 'Namespace' section of the DaoCloud interface. On the left, there's a sidebar with various cluster management sections like 'Cluster Overview', 'Node Management', 'Workload Management', etc. The 'Namespace' section is highlighted with a red box. On the right, there's a table listing existing namespaces such as 'cert-manager', 'cro-test', 'cro-operator', etc. At the top right of the table area, there's a 'YAML 创建' (Create via YAML) button, which is also highlighted with a red box.

点击创建

3. 输入或粘贴事先准备好的 YAML 内容，或者从本地直接导入已有的 YAML 文件。

输入 YAML 内容后，点击 下载 可以将该 YAML 文件保存到本地。

This screenshot shows the 'YAML 创建' (Create via YAML) dialog box. It contains a code editor with the following YAML configuration:

```

1 kind: Namespace
2 apiVersion: v1
3 metadata:
4 name: test-demo
5 spec:
6 finalizers:
7 - kubernetes
8 status:
9 phase: Active

```

At the bottom right of the dialog box, there are two buttons: '取消' (Cancel) and '确定' (Confirm), with '确定' being highlighted with a red box.

点击创建

4. 最后在弹框右下角点击 确定 即可。

## 启用命名空间独享节点

命名空间独享节点指在 Kubernetes 集群中，通过污点和污点容忍的方式实现特定命名空

间对一个或多个节点 CPU、内存等资源的独享。为特定命名空间配置独享节点后，其它非此命名空间的应用和服务均不能运行在被独享的节点上。使用独享节点可以让重要应用独享一部分计算资源，从而和其他应用实现物理隔离。

#### !!! note

在节点被设置为独享节点前已经运行在此节点上的应用和服务将不会受影响，依然会正常运行在该节点上，仅当这些 Pod 被删除或重建时，才会调度到其它非独享节点上。

## 准备工作

检查当前集群的 kube-apiserver 是否启用了 **PodNodeSelector** 和 **PodTolerationRestriction** 准入控制器。

使用命名空间独享节点功能需要用户启用 kube-apiserver 上的 **PodNodeSelector** 和 **PodTolerationRestriction** 两个特性准入控制器（Admission Controllers），关于准入控制器更多说明请参阅 [kubernetes Admission Controllers Reference](#)。

您可以前往当前集群下任意一个 Master 节点上检查 **kube-apiserver.yaml** 文件内是否启用了这两个特性，也可以在 Master 节点上执行如下命令进行快速检查：

```
```bash
[root@g-master1 ~]# cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep --enable-admission-plugins
```

```
# 预期输出如下：
- --enable-admission-plugins=NodeRestriction,PodNodeSelector,PodTolerationRestriction
```

```

## 在全局服务集群上启用命名空间独享节点

由于全局服务集群上运行着 kpanda、ghippo、insight 等平台基础组件，在 Global 启用命名空间独享节点将可能导致当系统组件重启后，系统组件无法调度到被独享的节点上，影响系统的整体高可用能力。因此，通常情况下，我们不推荐用户在全局服务集群上启用命

## 名空间独享节点特性。

如果您确实需要在全局服务集群上启用命名空间独享节点，请参考以下步骤进行开启：

1. 为全局服务集群的 kube-apiserver 启用了 **PodNodeSelector** 和

### **PodTolerationRestriction** 准入控制器

!!! note

如果集群已启用了上述的两个准入控制器，请跳过此步，直接前往配置系统组件容忍。

前往当前集群下任意一个 Master 节点上修改 **kube-apiserver.yaml** 配置文件，也可以

在 Master 节点上执行执行如下命令进行配置：

```
[root@g-master1 ~]# vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

# 预期输出如下:

```
apiVersion: v1
kind: Pod
metadata:

spec:
 containers:
 - command:
 - kube-apiserver

 - --default-not-ready-toleration-seconds=300
 - --default-unreachable-toleration-seconds=300
 - --enable-admission-plugins=NodeRestriction #启用的准入控制器列表
 - --enable-aggregator-routing=False
 - --enable-bootstrap-token-auth=true
 - --endpoint-reconciler-type=lease
 - --etcd-cafile=/etc/kubernetes/ssl/etcd/ca.crt

```

找到 **--enable-admission-plugins** 参数，加入（以英文逗号分隔的）**PodNodeSelector**

和 **PodTolerationRestriction** 准入控制器。参考如下：

```
加入 __ ,PodNodeSelector,PodTolerationRestriction__
- --enable-admission-plugins=NodeRestriction,PodNodeSelector,PodTolerationRestriction
```

2. 为平台组件所在的命名空间添加容忍注解

完成准入控制器的开启后，您需要为平台组件所在的命名空间添加容忍注解，以保证

平台组件的高可用。

目前 DCE 5.0 的系统组件命名空间如下表：

| 命名空间                | 所包含的系统组件                                                                                                            |
|---------------------|---------------------------------------------------------------------------------------------------------------------|
| kpanda-system       | kpanda                                                                                                              |
| hwameiStor-system   | hwameiStor                                                                                                          |
| istio-system        | istio                                                                                                               |
| metallb-system      | metallb                                                                                                             |
| cert-manager-system | cert-manager                                                                                                        |
| contour-system      | contour                                                                                                             |
| kubean-system       | kubean                                                                                                              |
| ghippo-system       | gippo                                                                                                               |
| kcoral-system       | kcoral                                                                                                              |
| kcollie-system      | kcollie                                                                                                             |
| insight-system      | insight、 insight-agent:                                                                                             |
| ipavo-system        | ipavo                                                                                                               |
| kairship-system     | kairship                                                                                                            |
| karmada-system      | karmada                                                                                                             |
| amamba-system       | amamba、 jenkins                                                                                                     |
| skoala-system       | skoala                                                                                                              |
| mspider-system      | mspider                                                                                                             |
| mcamel-system       | mcamel-rabbitmq、 mcamel-elasticsearch、 mcamel-mysql、<br>mcamel-redis、 mcamel-kafka、 mcamel-minio、 mcamel-postgresql |
| spidernet-system    | spidernet                                                                                                           |
| kangaroo-system     | kangaroo                                                                                                            |
| gmagpie-system      | gmagpie                                                                                                             |
| dowl-system         | dowl                                                                                                                |

检查当前集群中所有命名空间是否存在上述的命名空间，执行如下命令，分别为每个

命 名 空 间 添 加 注 解 :    scheduler.alpha.kubernetes.io/defaultTolerations:

```
'[{"operator": "Exists", "effect": "NoSchedule", "key": "ExclusiveNamespace"}]'。
```

```
kubectl annotate ns <namespace-name> scheduler.alpha.kubernetes.io/defaultTolerations: '[{"operator": "Exists", "effect": "NoSchedule", "key": "ExclusiveNamespace"}]'
```

请确保将 <namespace-name> 替换为要添加注解的平台命名空间名称。

### 3. 使用界面为命名空间设置独享节点

当您确认集群 API 服务器上的 **PodNodeSelector** 和 **PodTolerationRestriction** 两个特性准入控制器已经开启后，请参考如下步骤使用 DCE 5.0 的 UI 管理界面为命名空间设置独享节点了。

1. 在集群列表页面点击集群名称，然后在左侧导航栏点击 **命名空间**。

命名空间

命名空间

2. 点击命名空间名称，然后点击 **独享节点** 页签，在下方右侧点击 **添加节点**。

添加节点

添加节点

3. 在页面左侧选择让该命名空间独享哪些节点，在右侧可以清空或删除某个已选节点，最后在底部点击 **确定**。

确定

确定

4. 可以在列表中查看此命名空间的已有的独享节点，在节点右侧可以选择 **取消独享**。

取消独享之后，其他命名空间下的 Pod 也可以被调度到该节点上。

取消独享

取消独享

## 在 非全局服务集群上启用命名空间独享节点

在 非全局服务集群上启用命名空间独享节点 , 请参考以下步骤进行开启 :

1. 为当前集群的 `kube-apiserver` 启用了 `PodNodeSelector` 和 `PodTolerationRestriction` 准入控制器

**!!! note**

如果集群已启用了上述的两个准入控制器, 请跳过此步, 直接前往界面为命名空间设置独享节点

前往当前集群下任意一个 Master 节点上修改 `kube-apiserver.yaml` 配置文件 , 也可以

在 Master 节点上执行执行如下命令进行配置 :

```
[root@g-master1 ~]# vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

# 预期输出如下:

```
apiVersion: v1
kind: Pod
metadata:

spec:
 containers:
 - command:
 - kube-apiserver

 - --default-not-ready-toleration-seconds=300
 - --default-unreachable-toleration-seconds=300
 - --enable-admission-plugins=NodeRestriction #启用的准入控制器列表
 - --enable-aggregator-routing=False
 - --enable-bootstrap-token-auth=true
 - --endpoint-reconciler-type=lease
 - --etcd-cafile=/etc/kubernetes/ssl/etcd/ca.crt

```

找到 `--enable-admission-plugins` 参数 , 加入 ( 以英文逗号分隔的 ) `PodNodeSelector`

和 `PodTolerationRestriction` 准入控制器。参考如下 :

```
加入 __ ,PodNodeSelector,PodTolerationRestriction__
- --enable-admission-plugins=NodeRestriction,PodNodeSelector,PodTolerationRestriction
```

2. 使用界面为命名空间设置独享节点

当您确认集群 API 服务器上的 **PodNodeSelector** 和 **PodTolerationRestriction** 两个特性准入控制器已经开启后，请参考如下步骤使用 DCE 5.0 的 UI 管理界面为命名空间设置独享节点了。

1. 在集群列表页面点击集群名称，然后在左侧导航栏点击 **命名空间**。

命名空间

命名空间

2. 点击命名空间名称，然后点击 **独享节点** 页签，在下方右侧点击 **添加节点**。

添加节点

添加节点

3. 在页面左侧选择让该命名空间独享哪些节点，在右侧可以清空或删除某个已选节点，最后在底部点击 **确定**。

确定

确定

4. 可以在列表中查看此命名空间的已有的独享节点，在节点右侧可以选择 **取消独享**。

取消独享之后，其他命名空间下的 Pod 也可以被调度到该节点上。

取消独享

取消独享

3. 为需要高可用的组件所在的命名空间添加容忍注解（可选）

执行如下命令，需要高可用的组件所在的命名空间添加注解：

```
scheduler.alpha.kubernetes.io/defaultTolerations: '[{"operator": "Exists", "effect": "NoSchedule", "key": "ExclusiveNamespace"}]'。
```

```
kubectl annotate ns <namespace-name> scheduler.alpha.kubernetes.io/defaultTolerations: '[{"operator": "Exists", "effect": "NoSchedule", "key": "ExclusiveNamespace"}]'
```

请确保将 `<namespace-name>` 替换为要添加注解的平台命名空间名称。

## 容器组安全策略

容器组安全策略指在 Kubernetes 集群中，通过为指定命名空间配置不同的等级和模式，

实现在安全的各个方面控制 Pod 的行为，只有满足一定的条件的 Pod 才会被系统接受。

它设置三个等级和三种模式，用户可以根据自己的需求选择更加合适的方案来设置限制策略。

### !!! note

一条安全模式仅能配置一条安全策略。同时请谨慎为命名空间配置 enforce 的安全模式，违反后将会导致 Pod 无法创建。

本节将介绍如何通过容器管理界面为命名空间配置容器组安全策略。

## 前提条件

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，集群的版本需要在 v1.22 以上，且能够访问集群的 UI 界面。
- 已完成一个[命名空间的创建](#)、[用户的创建](#)，并为用户授予 [NS Admin](#) 或更高权限，详情可参考[命名空间授权](#)。

## 为命名空间配置容器组安全策略

- 选择需要配置容器组安全策略的命名空间，进入详情页。在 容器组安全策略 页面点击 **配置策略**，进入配置页。

### 配置策略列表

## 配置策略列表

2. 在配置页点击 **添加策略**，则会出现一条策略，包括安全级别和安全模式，以下是对安全级别和安全策略的详细介绍。

### 安全级别    描述

|            |                                         |
|------------|-----------------------------------------|
| Privileged | 不受限制的策略，提供最大可能范围的权限许可。此策略允许已知的特权提升。     |
| Baseline   | 限制性最弱的策略，禁止已知的策略提升。允许使用默认的（规定最少）Pod 配置。 |
| Restricted | 限制性非常强的策略，遵循当前的保护 Pod 的最佳实践。            |

### 安全模式    描述

#### 式

|         |                                  |
|---------|----------------------------------|
| Audit   | 违反指定策略会在审计日志中添加新的审计事件，Pod 可以被创建。 |
| Warn    | 违反指定策略会返回用户可见的告警信息，Pod 可以被创建。    |
| Enforce | 违反指定策略会导致 Pod 无法创建。              |

## 添加策略

### 添加策略

3. 不同的安全级别对应不同的检查项，若您不知道该如何为您的命名空间配置，可以点击页面右上角的 **策略配置项说明** 查看详细信息。

### 配置项说明 01

#### 配置项说明 01

#### 配置项说明 01

#### 配置项说明 01

4. 点击确定，若创建成功，则页面上将出现您配置的安全策略。

创建成功

创建成功

5. 点击  还可以编辑或者删除您配置的安全策略。

操作

操作

## 创建集群节点可用性检查

在创建集群或为已有集群添加节点时，请参阅下表，检查节点配置，以避免因节点配置错误导致集群创建或扩容失败。

检查项      描述

操作系统      参考[支持的架构及操作系统](#)

SELinux      关闭

防火墙      关闭

架构一致性      节点间 CPU 架构一致（如均为 ARM 或 x86）

主机时间      所有主机间同步误差小于 10 秒。

网络联通性      节点及其 SSH 端口能够正常被平台访问。

CPU      可用 CPU 资源大于 4 Core

内存      可用内存资源大于 8 GB

## 支持的架构及操作系统

| 架<br>构  | 操作系<br>统                                            | 备<br>注 |
|---------|-----------------------------------------------------|--------|
| AR<br>M | Kylin Linux Advanced Server release V10 (Sword) SP2 | 推<br>荐 |
| AR<br>M | UOS Linux                                           |        |
| AR<br>M | openEuler                                           |        |
| x8<br>6 | CentOS 7.x                                          | 推<br>荐 |
| x8<br>6 | Redhat 7.x                                          | 推<br>荐 |
| x8<br>6 | Redhat 8.x                                          | 推<br>荐 |
| x8<br>6 | Flatcar Container Linux by Kinvolk                  |        |
| x8<br>6 | Debian Bullseye, Buster, Jessie, Stretch            |        |
| x8<br>6 | Ubuntu 16.04, 18.04, 20.04, 22.04                   |        |
| x8<br>6 | Fedora 35, 36                                       |        |
| x8<br>6 | Fedora CoreOS                                       |        |
| x8<br>6 | openSUSE Leap 15.x/Tumbleweed                       |        |
| x8<br>6 | Oracle Linux 7, 8, 9                                |        |
| x8<br>6 | Alma Linux 8, 9                                     |        |
| x8<br>6 | Rocky Linux 8, 9                                    |        |

| 架构 | 操作系统                                                     | 备注 |
|----|----------------------------------------------------------|----|
| 6  |                                                          |    |
| x8 | Amazon Linux 2                                           |    |
| 6  |                                                          |    |
| x8 | Kylin Linux Advanced Server release V10 (Sword) - SP2 海光 |    |
| 6  |                                                          |    |
| x8 | UOS Linux                                                |    |
| 6  |                                                          |    |
| x8 | openEuler                                                |    |
| 6  |                                                          |    |

## 节点认证

### 使用 SSH 密钥认证节点

如果您选择使用 SSH 密钥作为待创建集群的节点认证方式，您需要按照如下说明配置公私钥。

1. 执行如下命令，在\*\* 待建集群的管理集群中的任意节点 \*\*上生成公私钥。

```
cd /root/.ssh
ssh-keygen -t rsa
```

2. 执行 **ls** 命令查看管理集群上的密钥是否创建成功，正确反馈如下：

```
ls
id_rsa id_rsa.pub known_hosts
```

其中名为 **id\_rsa** 的文件是私钥，名为 **id\_rsa.pub** 的文件是公钥。

3. 执行如下命令，分别将公钥文件 **id\_rsa.pub** 加载到待创建集群的所有节点上。

```
ssh-copy-id -i /root/.ssh/id_rsa.pub root@10.0.0.0
```

将上面命令中的 **root@10.0.0.0** 用户账号和节点 IP 替换为待创建集群的节点用户名

和 IP。\*\* 需要在待创建集群的每台节点都执行相同的操作 \*\*。

4. 执行如下命令，查看步骤 1 所创建的私钥文件 **id\_rsa**。

```
cat /root/.ssh/id_rsa
```

输出如下内容：

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEA3UvyKINzY5BFuemQ+uJ6q+GqgfvnWwNC8HzZhpcMSjJy26MM
UtBEBJxy8fMi57XcjYxPibXW/wnd+32ICCCycqCwByUmuXeCC1cjICQDqjcAvXae7
Y54IXGF7wm2IsMNwf0kjFEXjuS48FLDA0mGRaN3BG+Up5geXcHckg3K5LD8kXFFx
dEmSIjdyw55NaUitmEdHzN7cIdfi6Z56jcV8dcFBgWKUx+ebiyPmZBkXToz6GnMF
rswzzZCl+G6Jb2xTGy7g7ozb4BoZd1IpSD5EhDanRrESVE0C5Yu5zUAC0CvVd11
v67AK8Ko6MXToHp01/bcsvIM6cqgwUFXZKVeOwIDAQABoIBAQCO36GQlo3BEjxy
M2HvGJmqrx+unDxaflIRe4nVY2AD515Qf4xNSzke4QM1QoyerM0wf446krQkJPK0
k+9nl6Xszby5gGCbK4BNFk8I6RaGPjZWeRx6zGUJf8avWJiPxx6yjz2esSC9RiR0
F0nmieefVMyAfgv2/5++dK2WUFNNRKLgSRRpP5bRaD5wMzzxtSSxUon6217HO8p
3RoWsI51MbVzhvGpHUNABC0a0rpr9svT6XLKZxY8mxpKFYjM0Wv2JIDABg3kBvh
QbJ7kStCO3naZjKMU9UuSqVJs06cfIGYw7Or8/tABR3LErNQKPjkhAQqt0DXw7Iw
3tKdT AJBAoGBAP687U7JAQoQKcphek2E/A/sbO/d37ix7Z3vNOy065STrA+ZWMZn
pZ6Ui1B/oJpoZssnfvIoz9sn559X0j67TljFALFd2ZGS0Fqh9KVCqDvfk+Vst1dq
+3r/yZdT OyswoccxkjiC/GDwZGK0amJWqvob39JCZhDAKIGLbGMmjAHAAoGBAN5k
m1WGnni1nZ+3dryIwgB6z1hWcnLTamzSET6KhSuo946ET0IRG9xtlheCx6dqICbr
VkJY4NrRZjK/p/YGx59rDWf7E3I8ZMgR7mjieOcUZ4lU1A4l7ZIIW/2WZHW+nUXO
Ti20fqJ8qSp4BUvOvuth1pz2GLUHe2/Fxjf7HIstAoGBAPHpPr9r+TfIIPsJeRj2
6lzA3G8qWFRQfGRYjv0fjv0pA+RIb1rzgP/I90g5+63G6Z+R4WdcxI/OJJNY1iuG
uw9n/pFxm7U4JC990BPE6nj5Lz+clpNGYckNDBF9VG9vFSrSDLdaYkxoVNvG/xJ
a9Na90H4lm7f3VewrPy310KvAoGAZr+mwNoEh5Kpc6xo8Gxi7aPP/mlaUV6X7Ki
gvmu02AqmC7rC4QqEiqTaONkaSXwGusqIWxJ3yp5hELmUBYLzsxAEeV/s4zRp1oZ
g133LBRSTbHFAdBmNdqK6Nu+KGRb92980UMOKvZbliKDI+W6cbfvVu+gtKrzTc3b
aevb4TUCgYEAnJAxVYDP1nJf7bjBSHXQu1E/DMwbtrqw7dylRJ8cAzI7IxfSCez
7BYWq41PqVd9/zrb3Pbh2phiVzKe783igAIMqummcjo/kZyCwFsYBzK77max1jF5
aPQsLbRS2aDz8kIH6jHPZ/R+15ER0mdtLmA7vIJZGerWWQR0dUU+XXA=
```

将私钥内容复制后填至界面密钥输入框。

SSH 认证

SSH 认证

## 集群节点扩容

随着业务应用不断增长，集群资源日趋紧张，这时可以基于 kubelet 对集群节点进行扩容。

扩容后，应用可以运行在新增的节点上，缓解资源压力。

只有通过容器管理模块[创建的集群](#)才支持节点扩缩容，从外部接入的集群不支持此操作。

本文主要介绍同种架构下工作集群的 **工作节点** 扩容。

1. 在 **集群列表** 页面点击目标集群的名称。

若 **集群角色** 中带有 **接入集群** 的标签，则说明该集群不支持节点扩缩容。

进入集群列表页面

进入集群列表页面

2. 在左侧导航栏点击 **节点管理**，然后在页面右上角点击 **接入节点**。

节点管理

节点管理

3. 输入主机名称和节点 IP 并点击 **确定**。

点击 **添加工作节点** 可以继续接入更多节点。

节点管理

节点管理

!!! note

接入节点大约需要 20 分钟，请您耐心等待。

## 参考文档

- [对工作集群的控制节点扩容](#)
- [为工作集群添加异构节点](#)
- [为全局服务集群的工作节点扩容](#)
- [替换工作集群的首个控制节点](#)

# 集群节点缩容

当业务高峰期结束之后，为了节省资源成本，可以缩小集群规模，卸载冗余的节点，即节点缩容。节点卸载后，应用无法继续运行在该节点上。

## 前提条件

- 当前操作用户具有 [Cluster Admin](#) 角色授权。
- 只有通过容器管理模块[创建的集群](#)才支持节点扩缩容，从外部接入的集群不支持此操作。
- 卸载节点之前，需要[暂停调度该节点](#)，并且将该节点上的应用都驱逐至其他节点。
- 驱逐方式：登录控制器节点，通过 kubectl drain 命令驱逐节点上所有 Pod。安全驱逐的方式可以允许容器组里面的容器优雅地中止。

## 注意事项

1. 集群节点缩容时，只能逐个进行卸载，无法批量卸载。
2. 如需卸载集群控制器节点，需要确保最终控制器节点数为 **奇数**。
3. 集群节点缩容时不可下线 **第一个控制器** 节点。如果必须执行此操作，请联系售后工程师。

## 操作步骤

1. 在 **集群列表** 页面点击目标集群的名称。  
若 **集群角色** 中带有 **接入集群** 的标签，则说明该集群不支持节点扩缩容。

进入集群列表页面

进入集群列表页面

2. 在左侧导航栏点击 **节点管理**，找到需要卸载的节点，点击  选择 **移除节点**。

移除节点

移除节点

3. 输入节点名称，并点击 **删除** 进行确认。

移除节点

移除节点

## 节点污点管理

污点 (Taint) 能够使节点排斥某一类 Pod，避免 Pod 被调度到该节点上。每个节点上可以应用一个或多个污点，不能容忍这些污点的 Pod 则不会被调度该节点上。

### 注意事项

1. 当前操作用户应具备 [NS Editor](#) 角色授权或其他更高权限。

2. 为节点添加污点之后，只有能容忍该污点的 Pod 才能被调度到该节点。

### 操作步骤

1. 在 **集群列表** 页找到目标集群，点击集群名称，进入 **集群概览** 页面。

点击集群名称

点击集群名称

2. 在左侧导航栏，点击 **节点管理**，找到需要修改污点的节点，点击右侧的  操作图标

并点击 **修改污点** 按钮。

修改污点

修改污点

3. 在弹框内输入污点的键值信息，选择污点效果，点击 **确定**。

点击 **添加** 可以为节点添加多个污点，点击污点效果右侧的 **X** 可以删除污点。

目前支持三种污点效果：

- **NoSchedule**：新的 Pod 不会被调度到带有此污点的节点上，除非新的 Pod 具有相匹配的容忍度。当前正在节点上运行的 Pod **不会** 被驱逐。
- **NoExecute**：这会影响已在节点上运行的 Pod：
  - 如果 Pod 不能容忍此污点，会马上被驱逐。
  - 如果 Pod 能够容忍此污点，但是在容忍度定义中没有指定 tolerationSeconds，则 Pod 还会一直在该节点上运行。
  - 如果 Pod 能够容忍此污点而且指定了 tolerationSeconds，则 Pod 还能在该节点上继续运行指定的时长。这段时间过去后，再从节点上驱除这些 Pod。
- **PreferNoSchedule**：这是“软性”的 NoSchedule。控制平面将尝试避免将不容忍此污点的 Pod 调度到节点上，但不能保证完全避免。所以要尽量避免使用此污点。

修改污点

修改污点

有关污点的更多详情，请参阅 Kubernetes 官方文档：[污点和容忍度](#)。

# 节点调度

支持将节点暂停调度或恢复调度。暂停调度指，停止将 Pod 调度到该节点。恢复调度指，可以将 Pod 调度到该节点。

1. 在 **集群列表** 页面点击目标集群的名称。

进入集群列表页面

进入集群列表页面

2. 在左侧导航栏点击 **节点管理**，在节点右侧点击  操作图标，点击 **暂停调度** 按钮即可暂停调度该节点。

暂停调度

暂停调度

3. 在节点右侧点击  操作图标，点击 **恢复调度** 按钮即可恢复调度该节点。

节点管理

节点管理

节点调度状态可能因网络情况有所延迟，点击搜索框右侧的刷新图标可以刷新节点调度状态。

节点管理

节点管理

# 节点详情

接入或创建集群之后，可以查看集群中各个节点的信息，包括节点状态、标签、资源用量、

Pod、监控信息等。

1. 在 **集群列表** 页面点击目标集群的名称。

进入集群列表页面

进入集群列表页面

2. 在左侧导航栏点击 **节点管理**，可以查看节点状态、角色、标签、CPU/内存使用情况、

IP 地址、创建时间。

暂停调度

暂停调度

3. 点击节点名称，可以进入节点详情页面查看更多信息，包括概览信息、容器组信息、

标签注解信息、事件列表、状态等。

节点管理

节点管理

此外，还可以查看节点的 YAML 文件、监控信息、标签和注解等。

节点管理

节点管理

## 标签与注解

标签（Labels）是为 Pod、节点、集群等 Kubernetes 对象添加的标识性键值对，可结合标签选择器查找并筛选满足某些条件的 Kubernetes 对象。每个键对于给定对象必须是唯一的。

注解（Annotations）和标签一样，也是键/值对，但不具备标识或筛选功能。使用注解可以为节点添加任意的元数据。注解的键通常使用的格式为 **前缀（可选）/名称（必填）**，

例如 `nfd.node.kubernetes.io/extended-resources`。如果省略前缀，表示该注解键是用户私有的。

有关标签和注解的更多信息，可参考 Kubernetes 的官方文档[标签和选择算符](#)或[注解](#)。

添加/删除标签与注解的步骤如下：

1. 在 **集群列表** 页面点击目标集群的名称。

进入集群列表页面

进入集群列表页面

2. 在左侧导航栏点击 **节点管理**，在节点右侧点击  操作图标，点击 **修改标签** 或 **修改注解**。

暂停调度

暂停调度

3. 点击  **添加** 可以添加标签或注解，点击  **X** 可以删除标签或注解，最后点击 **确定**。

节点管理

节点管理

## 创建无状态负载（Deployment）

本文介绍如何通过镜像和 YAML 文件两种方式创建无状态负载。

[无状态负载（Deployment）](#)是 Kubernetes 中的一种常见资源，主要为 [Pod](#) 和 [ReplicaSet](#) 提供声明式更新，支持弹性伸缩、滚动升级、版本回退等功能。在 Deployment 中声明期望的 Pod 状态，Deployment Controller 会通过 ReplicaSet 修改当前状态，使其达到预先声明的期望状态。Deployment 是无状态的，不支持数据持久化，适用于部署无状态的、不需要

要保存数据、随时可以重启回滚的应用。

通过 [DCE 5.0](#) 的容器管理模块，可以基于相应的角色权限轻松管理多云多集群上的工作负载，包括对无状态负载的创建、更新、删除、弹性扩缩、重启、版本回退等全生命周期管理。

## 前提条件

在使用镜像创建无状态负载之前，需要满足以下前提条件：

- 在[容器管理](#)模块中[接入 Kubernetes 集群](#)或者[创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建一个[命名空间](#)和[用户](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。
- 单个实例中有多个容器时，请确保容器使用的端口不冲突，否则部署会失效。

## 镜像创建

参考以下步骤，使用镜像创建一个无状态负载。

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情** 页面。

集群详情

集群详情

2. 在集群详情页面，点击左侧导航栏的 **工作负载 -> 无状态负载**，然后点击页面右上

角的 **镜像创建** 按钮。

工作负载

工作负载

3. 依次填写基本信息、容器配置、服务配置、高级配置后，在页面右下角点击**确定**完成创建。

系统将自动返回 **无状态负载** 列表。点击列表右侧的 ，可以对负载执行执行更新、删除、弹性扩缩、重启、版本回退等操作。如果负载状态出现异常，请查看具体异常信息，可参考[工作负载状态](#)。

### 操作菜单

#### 操作菜单

## 基本信息

- 负载名称：最多包含 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾，例如 deployment-01。同一命名空间内同一类型工作负载的名称不得重复，而且负载名称在工作负载创建好之后不可更改。
- 命名空间：选择将新建的负载部署在哪个命名空间，默认使用 default 命名空间。找不到所需的命名空间时可以根据页面提示去[创建新的命名空间](#)。
- 实例数：输入负载的 Pod 实例数量，默认创建 1 个 Pod 实例。
- 描述：输入负载的描述信息，内容自定义。字符数不超过 512。

### 基本信息

#### 基本信息

## 容器配置

容器配置分为基本信息、生命周期、健康检查、环境变量、数据存储、安全设置六部分，点击下方的相应页签可查看各部分的配置要求。

容器配置仅针对单个容器进行配置，如需在一个容器组中添加多个容器，可点击右侧的 **+**

添加多个容器。

### ==== “基本信息（必填）”

在配置容器相关参数时，必须正确填写容器的名称、镜像参数，否则将无法进入下一步。参考以下要求填写配置后，点击 **确认**。

![基本信息](docs/zh/docs/kpanda/user-guide/images/deploy05.png)

- 容器类型：默认为`工作容器`。有关初始化容器，参见 [k8s 官方文档](https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/init-containers/)。
- 容器名称：最多包含 63 个字符，支持小写字母、数字及分隔符（“-”）。必须以小写字母或数字开头及结尾，例如 `nginx-01`。
- 镜像：

    - 容器镜像：从列表中选择一个合适的镜像。输入镜像名称时，默认从官方的 [DockerHub](https://hub.docker.com/) 拉取镜像。

    安装 DCE 5.0 的[镜像仓库](../../../../kangaroo/intro/index.md)模块后，可以点击右侧的 **选择镜像** 按钮来选择镜像。

- 镜像版本：从下拉列表选择一个合适的版本。
- 镜像拉取策略：勾选 **总是拉取镜像** 后，负载每次重启/升级时都会从仓库重新拉取镜像。

如果不勾选，则只拉取本地镜像，只有当镜像在本地不存在时才从镜像仓库重新拉取。

    更多详情可参考[镜像拉取策略](https://kubernetes.io/zh-cn/docs/concepts/containers/images/#image-pull-policy)。

- 镜像仓库密钥：可选。如果目标仓库需要 `Secret` 才能访问，需要先去[创建一个密钥](../../../../configmaps-secrets/create-secret.md)。
- 特权容器：容器默认不可以访问宿主机上的任何设备，开启特权容器后，容器即可访问宿主机上的所有设备，享有宿主机上的运行进程的所有权限。
- CPU/内存配额：CPU/内存资源的请求值（需要使用的最小资源）和限制值（允许使用的最大资源）。请根据需要为容器配置资源，避免资源浪费和因容器资源超额导致系统故障。默认值如图所示。
- GPU 配置：为容器配置 GPU 用量，仅支持输入正整数。

    - 整卡模式：

        - 物理卡数量：容器能够使用的物理 GPU 卡数量。配置后，容器将占用整张物理 GPU 卡。同时物理卡数量需要  $\leq$  单节点插入的最大 GPU 卡数。

    - 虚拟化模式：

        - 物理卡数量：容器能够使用的物理 GPU 卡数量，物理卡数量需要  $\leq$  单节点插入的最大 GPU 卡数。

        - GPU 算力：每张物理 GPU 卡上需要使用的算力百分比，最多为 100%。

        - 显存：每张物理卡上需要使用的显存数量。

    - 调度策略（Binpack/Spread）：支持基于 GPU 卡和基于节点的两种维度的调度策略。Binpack 是集中式调度策略，优先将容器调度到同一个节点的同一张 GPU 卡上；Spread 是

分散式调度策略，优先将容器调度到不同节点的不同 GPU 卡上，根据实际场景可组合使用。

（当工作负载级别的 Binpack/Spread 调度策略与集群级别的 Binpack/Spread 调度策略冲突时，系统优先使用工作负载级别的调度策略）。

- 任务优先级：GPU 算力会优先供给高优先级任务使用，普通任务会减少甚至暂停使用 GPU 算力，直到高优先级任务结束，普通任务会重新继续使用 GPU 算力，常用于在离线混部场景。

- 指定型号：将工作负载调度到指定型号的 GPU 卡上，适用于对 GPU 型号有特殊要求的场景。

- Mig 模式

- 规格：切分后的物理 GPU 卡规格。

- 数量：使用该规格的数量。

> 设置 GPU 之前，需要管理员预先在集群上安装 [GPU Operator](./gpu/nvidia/install\_nvidia\_driver\_of\_operator.md) 和 [nvidia-vgpu](./gpu/nvidia/vgpu/vgpu\_addon.md)（仅 vGPU 模式需要安装），并在[集群设置](./clusterops/cluster-settings.md)中开启 GPU 特性。

#### ==== “生命周期（选填）”

设置容器启动时、启动后、停止前需要执行的命令。详情可参考[容器生命周期配置](pod-config/lifecycle.md)。

![生命周期](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy06.png)

#### ==== “健康检查（选填）”

用于判断容器和应用的健康状态，有助于提高应用的可用性。详情可参考[容器健康检查配置](pod-config/health-check.md)。

![健康检查](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy07.png)

#### ==== “环境变量（选填）”

配置 Pod 内的容器参数，为 Pod 添加环境变量或传递配置等。详情可参考[容器环境变量配置](pod-config/env-variables.md)。

![环境变量](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy08.png)

#### ==== “数据存储（选填）”

配置容器挂载数据卷和数据持久化的设置。详情可参考[容器数据存储配置](pod-config/env-variables.md)。

![数据存储](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy09.png)

#### ==== “安全设置（选填）”

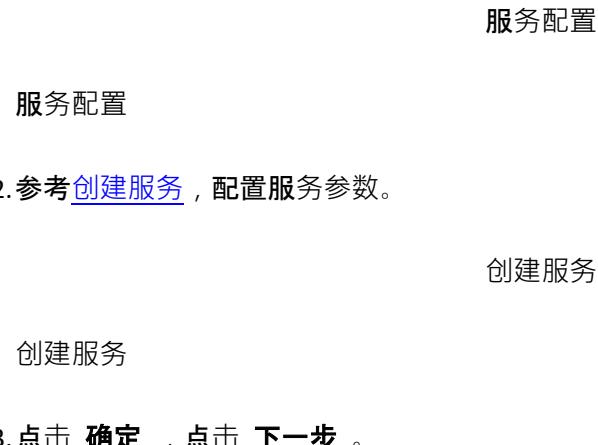
通过 Linux 内置的账号权限隔离机制来对容器进行安全隔离。您可以通过使用不同权限的账号 UID（数字身份标记）来限制容器的权限。例如，输入 \_\_0\_\_ 表示使用 root 账号的权限。

![安全设置](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy10.png)

## 服务配置

为无状态负载配置[服务 \( Service \)](#)，使无状态负载能够被外部访问。

1. 点击 **创建服务** 按钮。



2. 参考[创建服务](#)，配置服务参数。

## 高级配置

高级配置包括负载的网络配置、升级策略、调度策略、标签与注解四部分，可点击下方的页签查看各部分的配置要求。

--- “网络配置”

- 如在集群中部署了 [SpiderPool](../../network/modules/spiderpool/index.md) 和 [Multus](../../network/modules/multus-underlay/index.md) 组件，则可以在网络配置中配置容器网卡。详情参考[工作负载使用 IP 池](../../network/config/use-ippool/usage.md)。

- DNS 配置：应用在某些场景下会出现冗余的 DNS 查询。Kubernetes 为应用提供了与 DNS 相关的配置选项，能够在某些场景下有效地减少冗余的 DNS 查询，提升业务并发量。

- DNS 策略

- Default：使容器使用 kubelet 的 `--resolv-conf` 参数指向的域名解析文件。该配置只能解析注册到互联网上的外部域名，无法解析集群内部域名，且不存在无效的 DNS 查询。

- ClusterFirstWithHostNet：应用对接主机的域名文件。

- ClusterFirst：应用对接 Kube-DNS/CoreDNS。

- **None**: Kubernetes v1.9 (Beta in v1.10) 中引入的新选项值。设置为 None 之后，必须设置 dnsConfig，此时容器的域名解析文件将完全通过 dnsConfig 的配置来生成。

- **域名服务器**: 填写域名服务器的地址，例如 `_10.6.175.20_`。
- **搜索域**: 域名查询时的 DNS 搜索域列表。指定后，提供的搜索域列表将合并到基于 dnsPolicy 生成的域名解析文件的 search 字段中，并删除重复的域名。Kubernetes 最多允许 6 个搜索域。
- **Options**: DNS 的配置选项，其中每个对象可以具有 name 属性（必需）和 value 属性（可选）。该字段中的内容将合并到基于 dnsPolicy 生成的域名解析文件的 options 字段中，dnsConfig 的 options 的某些选项如果与基于 dnsPolicy 生成的域名解析文件的选项冲突，则会被 dnsConfig 所覆盖。
- **主机别名**: 为主机设置的别名。

![DNS 配置](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy17.png)

#### ==== “升级策略”

- **升级方式**: `_滚动升级_` 指逐步用新版本的实例替换旧版本的实例，升级的过程中，业务流量会同时负载均衡分布到新老的实例上，因此业务不会中断。`_重建升级_` 指先删除老版本的负载实例，再安装指定的新版本，升级过程中业务会中断。
- **最大不可用**: 指定负载更新过程中不可用 Pod 的最大值或比率，默认 25%。如果等于实例数有服务中断的风险。
- **最大峰值**: 更新 Pod 的过程中 Pod 总数超过 Pod 期望副本数部分的最大值或比率。默认 25%。
- **最大保留版本数**: 设置版本回滚时保留的旧版本数量。默认 10。
- **Pod 可用最短时间**: Pod 就绪的最短时间，只有超出这个时间 Pod 才被认为可用，默认 0 秒。
- **升级最大持续时间**: 如果超过所设置的时间仍未部署成功，则将该负载标记为失败。默认 600 秒。
- **缩容时间窗**: 负载停止前命令的执行时间窗（0-9,999 秒），默认 30 秒。

![升级策略](docs/zh/docs/kpanda/user-guide/images/deploy14.png)

#### ==== “调度策略”

- **容忍时间**: 负载实例所在的节点不可用时，将负载实例重新调度到其它可用节点的时间，默认为 300 秒。
- **节点亲和性**: 根据节点上的标签来约束 Pod 可以调度到哪些节点上。
- **工作负载亲和性**: 基于已经在节点上运行的 Pod 的标签来约束 Pod 可以调度到哪些节点。
- **工作负载反亲和性**: 基于已经在节点上运行的 Pod 的标签来约束 Pod 不可以调度到哪些节点。

> 具体详情请参考[调度策略](pod-config/scheduling-policy.md)。

![调度策略](docs/zh/docs/kpanda/user-guide/images/deploy15.png)

#### ==== “标签与注解”

可以点击 添加 按钮为工作负载和容器组添加标签和注解。

![标签与注解](docs/zh/docs/kpanda/user-guide/images/deploy16.png)

## YAML 创建

除了通过镜像方式外，还可以通过 YAML 文件更快速地创建无状态负载。

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情** 页面。

集群详情

集群详情

2. 在集群详情页面，点击左侧导航栏的 **工作负载 -> 无状态负载**，然后点击页面右上角的 **YAML 创建** 按钮。

工作负载

工作负载

3. 输入或粘贴事先准备好的 YAML 文件，点击 **确定** 即可完成创建。

工作负载

工作负载

??? note “点击查看创建无状态负载的 YAML 示例”

```
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # 告知 Deployment 运行 2 个与该模板匹配的 Pod
  template:
    metadata:
      labels:
        app: nginx
```
```

```
spec:
 containers:
 - name: nginx
 image: nginx:1.14.2
 ports:
 - containerPort: 80
 ...
```

## 创建有状态负载（StatefulSet）

本文介绍如何通过镜像和 YAML 文件两种方式创建有状态负载（StatefulSet）。

[有状态负载（StatefulSet）](#)是 Kubernetes 中的一种常见资源，和[无状态负载（Deployment）](#)

类似，主要用于管理 Pod 集合的部署和伸缩。二者的主要区别在于，Deployment 是无状态的，不保存数据，而 StatefulSet 是有状态的，主要用于管理有状态应用。此外，StatefulSet 中的 Pod 具有永久不变的 ID，便于在匹配存储卷时识别对应的 Pod。

通过 [DCE 5.0](#) 的容器管理模块，可以基于相应的角色权限轻松管理多云多集群上的工作负载，包括对有状态工作负载的创建、更新、删除、弹性扩缩、重启等全生命周期管理。

## 前提条件

在使用镜像创建有状态负载之前，需要满足以下前提条件：

- 在[容器管理](#)模块中[接入 Kubernetes 集群](#)或者[创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建一个[命名空间](#)和[用户](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。
- 单个实例中有多个容器时，请确保容器使用的端口不冲突，否则部署会失效。

## 镜像创建

参考以下步骤，使用镜像创建一个有状态负载。

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情**。

集群详情

集群详情

2. 点击左侧导航栏的 **工作负载 -> 有状态负载**，然后点击右上角 **镜像创建** 按钮。

工作负载

工作负载

3. 依次填写 基本信息、容器配置、服务配置、高级配置后，在页面右下角点击 **确定** 完成创建。

系统将自动返回 **有状态工作负载** 列表，等待工作负载状态变为 **运行中**。如果工作负载状态出现异常，请查看具体异常信息，可参考[工作负载状态](#)。

点击新建工作负载列右侧的 ，可以对工作负载执行执行更新、删除、弹性扩缩、重启等操作。

操作菜单

操作菜单

## 基本信息

- 负载名称：最多包含 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾，例如 deployment-01。同一命名空间内同一类型工作负载的名称不得重复，而且负载名称在工作负载创建好之后不可更改。
- 命名空间：选择将新建的负载部署在哪个命名空间，默认使用 default 命名空间。找

不到所需的命名空间时可以根据页面提示去[创建新的命名空间](#)。

- 实例数：输入负载的 Pod 实例数量，默认创建 1 个 Pod 实例。
- 描述：输入负载的描述信息，内容自定义。字符数不超过 512。

### 基本信息

#### 基本信息

## 容器配置

容器配置分为基本信息、生命周期、健康检查、环境变量、数据存储、安全设置六部分，

点击下方的相应页签可查看各部分的配置要求。

容器配置仅针对单个容器进行配置，如需在一个容器组中添加多个容器，可点击右侧的 +

添加多个容器。

#### == “基本信息（必填）”

在配置容器相关参数时，必须正确填写容器的名称、镜像参数，否则将无法进入下一步。参考以下要求填写配置后，点击 确认。

![基本信息](<https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/state11.png>)

- 容器类型：默认为“工作容器”。有关初始化容器，参见 [\[k8s 官方文档\]](https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/init-containers/)(<https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/init-containers/>)。

- 容器名称：最多包含 63 个字符，支持小写字母、数字及分隔符（“-”）。必须以小写字母或数字开头及结尾，例如 nginx-01。

- 镜像：

- 容器镜像：从列表中选择一个合适的镜像。输入镜像名称时，默认从官方的 [\[DockerHub\]](https://hub.docker.com/)(<https://hub.docker.com/>) 拉取镜像。

接入 DCE 5.0 的[镜像仓库]([../../kangaroo/intro/index.md](#))模块后，可以点击右侧的 选择镜像 按钮来选择镜像。

- 镜像版本：从下拉列表选择一个合适的版本。

- 镜像拉取策略：勾选 总是拉取镜像 后，负载每次重启/升级时都会从仓库重新拉取镜像。

如果不勾选，则只拉取本地镜像，只有当镜像在本地不存在时才从镜像仓库重新拉取。

更多详情可参考[镜像拉取策略](<https://kubernetes.io/zh-cn/docs/concepts/containers/images/#image-pull-policy>)。

- 镜像仓库密钥：可选。如果目标仓库需要 Secret 才能访问，需要先去[创建一个密钥]([..](#)/

configmaps-secrets/create-secret.md)。

- 特权容器：容器默认不可以访问宿主机上的任何设备，开启特权容器后，容器即可访问宿主机上的所有设备，享有宿主机上的运行进程的所有权限。
- CPU/内存配额：CPU/内存资源的请求值（需要使用的最小资源）和限制值（允许使用的最大资源）。请根据需要为容器配置资源，避免资源浪费和因容器资源超额导致系统故障。默认值如图所示。
- GPU 配置：为容器配置 GPU 用量，仅支持输入正整数。
  - 整卡模式：
    - 物理卡数量：容器能够使用的物理 GPU 卡数量。配置后，容器将占用整张物理 GPU 卡。同时物理卡数量需要 ≤ 单节点插入的最大 GPU 卡数。
    - 虚拟化模式：
      - 物理卡数量：容器能够使用的物理 GPU 卡数量，物理卡数量需要 ≤ 单节点插入的最大 GPU 卡数。
      - GPU 算力：每张物理 GPU 卡上需要使用的算力百分比，最多为 100%。
      - 显存：每张物理卡上需要使用的显存数量。
    - 调度策略（Binpack/Spread）：支持基于 GPU 卡和基于节点的两种维度的调度策略。Binpack 是集中式调度策略，优先将容器调度到同一个节点的同一张 GPU 卡上；Spread 是分散式调度策略，优先将容器调度到不同节点的不同 GPU 卡上，根据实际场景可组合使用。（当工作负载级别的 Binpack/Spread 调度策略与集群级别的 Binpack/Spread 调度策略冲突时，系统优先使用工作负载级别的调度策略）。
    - 任务优先级：GPU 算力会优先供给高优先级任务使用，普通任务会减少甚至暂停使用 GPU 算力，直到高优先级任务结束，普通任务会重新继续使用 GPU 算力，常用于在离线混部场景。
    - 指定型号：将工作负载调度到指定型号的 GPU 卡上，适用于对 GPU 型号有特殊要求的场景。
  - Mig 模式
    - 规格：切分后的物理 GPU 卡规格。
    - 数量：使用该规格的数量。

> 设置 GPU 之前，需要管理员预先在集群上安装 [GPU Operator](./gpu/nvidia/install\_nvidia\_driver\_of\_operator.md) 和 [nvidia-vgpu](./gpu/nvidia/vgpu/vgpu\_addon.md)（仅 vGPU 模式需要安装），并在[集群设置](./clusterops/cluster-settings.md)中开启 GPU 特性。

### ==== “生命周期（选填）”

设置容器启动时、启动后、停止前需要执行的命令。详情可参考[容器生命周期配置](pod-config/lifecycle.md)。

![生命周期](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy06.png)

### ==== “健康检查（选填）”

用于判断容器和应用的健康状态。有助于提高应用的可用性。详情可参考[容器健康检查配置](pod-config/health-check.md)。

![健康检查](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy07.png)

==== “环境变量（选填）”

配置 Pod 内的容器参数，为 Pod 添加环境变量或传递配置等。详情可参考[容器环境变量配置](pod-config/env-variables.md)。

![环境变量](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy08.png)

==== “数据存储（选填）”

配置容器挂载数据卷和数据持久化的设置。详情可参考[容器数据存储配置](pod-config/env-variables.md)。

![数据存储](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy09.png)

==== “安全设置（选填）”

通过 Linux 内置的账号权限隔离机制来对容器进行安全隔离。您可以通过使用不同权限的账号 UID（数字身份标记）来限制容器的权限。例如，输入 \_0\_ 表示使用 root 账号的权限。

![安全设置](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy10.png)

## 服务配置

为有状态负载配置[服务（Service）](#)，使有状态负载能够被外部访问。

1. 点击 **创建服务** 按钮。

服务配置

服务配置

2. 参考[创建服务](#)，配置服务参数。

创建服务

创建服务

3. 点击 **确定**，点击 **下一步**。

## 高级配置

高级配置包括负载的网络配置、升级策略、调度策略、标签与注解四部分，可点击下方的

页签查看各部分的配置要求。

#### ==== “网络配置”

![DNS 配置](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy17.png)

- 如在集群中部署了 [SpiderPool](../../network/modules/spiderpool/index.md) 和 [Multus](../../network/modules/multus-underlay/index.md) 组件，则可以在网络配置中配置容器网卡。详情参考[工作负载使用 IP 池](../../network/config/use-ipool/usage.md)。

- DNS 配置：应用在某些场景下会出现冗余的 DNS 查询。Kubernetes 为应用提供了与 DNS 相关的配置选项，能够在某些场景下有效地减少冗余的 DNS 查询，提升业务并发量。

- DNS 策略

- Default：使容器使用 kubelet 的 `--resolv-conf` 参数指向的域名解析文件。该配置只能解析注册到互联网上的外部域名，无法解析集群内部域名，且不存在无效的 DNS 查询。

- ClusterFirstWithHostNet：应用对接主机的域名文件。

- ClusterFirst：应用对接 Kube-DNS/CoreDNS。

- None：Kubernetes v1.9 (Beta in v1.10) 中引入的新选项值。设置为 None 之后，必须设置 `dnsConfig`，此时容器的域名解析文件将完全通过 `dnsConfig` 的配置来生成。

- 域名服务器：填写域名服务器的地址，例如 `10.6.175.20`。

- 搜索域：域名查询时的 DNS 搜索域列表。指定后，提供的搜索域列表将合并到基于 `dnsPolicy` 生成的域名解析文件的 `search` 字段中，并删除重复的域名。Kubernetes 最多允许 6 个搜索域。

- Options：DNS 的配置选项，其中每个对象可以具有 `name` 属性（必需）和 `value` 属性（可选）。该字段中的内容将合并到基于 `dnsPolicy` 生成的域名解析文件的 `options` 字段中，`dnsConfig` 的 `options` 的某些选项如果与基于 `dnsPolicy` 生成的域名解析文件的选项冲突，则会被 `dnsConfig` 所覆盖。

- 主机别名：为主机设置的别名。

#### ==== “升级策略”

![升级策略](docs/zh/docs/kpanda/images/state-deploy14.png)

- 升级方式：

- `滚动升级(RollingUpdate)` 指逐步用新版本的实例替换旧版本的实例，升级的过程中，业务流量会同时负载均衡分布到新老的实例上，因此业务不会中断。

- `重建升级(OnDelete)` 指先删除老版本的负载实例，再安装指定的新版本，升级过程中业务会中断。

- 最大保留版本数：设置版本回滚时保留的旧版本数量。默认 10。

- 缩容时间窗：负载停止前命令的执行时间窗（0-9,999 秒），默认 30 秒。

#### ==== “容器管理策略”

![容器管理策略](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/state05.png)

Kubernetes v1.7 及其之后的版本可以通过 `spec.podManagementPolicy` 设置 Pod 的管理策略，支持以下两种方式：

- `OrderedReady`：默认的 Pod 管理策略，表示按顺序部署 Pod，只有前一个 Pod 部署成功完成后，有状态负载才会开始部署下一个 Pod。删除 Pod 时则采用逆序，最后创建的最先被删除。
- `Parallel`：并行创建或删除容器，和 Deployment 类型的 Pod 一样。StatefulSet 控制器并行地启动或终止所有的容器。启动或者终止其他 Pod 前，无需等待 Pod 进入 Running 和 ready 或者完全停止状态。这个选项只会影响扩缩操作的行为，不影响更新时的顺序。

#### ==== “调度策略”

![调度策略](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy15.png)

- 容忍时间：负载实例所在的节点不可用时，将负载实例重新调度到其它可用节点的时间，默认为 300 秒。
- 节点亲和性：根据节点上的标签来约束 Pod 可以调度到哪些节点上。
- 工作负载亲和性：基于已经在节点上运行的 Pod 的标签来约束 Pod 可以调度到哪些节点。
- 工作负载反亲和性：基于已经在节点上运行的 Pod 的标签来约束 Pod 不可以调度到哪些节点。
- 拓扑域：即 `topologyKey`，用于指定可以调度的一组节点。例如，`kubernetes.io/os` 表示只要某个操作系统的节点满足 `labelSelector` 的条件就可以调度到该节点。

> 具体详情请参考[调度策略](pod-config/scheduling-policy.md)。

#### ==== “标签与注解”

可以点击 `添加` 按钮为工作负载和容器组添加标签和注解。

![标签与注解](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy16.png)

## YAML 创建

除了通过镜像方式外，还可以通过 YAML 文件更快速地创建创建有状态负载。

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情** 页面。

### 集群详情

#### 集群详情

2. 在集群详情页面，点击左侧导航栏的 **工作负载 -> 有状态负载**，然后点击页面右上角的 **YAML 创建** 按钮。

工作负载

工作负载

3. 输入或粘贴事先准备好的 YAML 文件，点击 **确定** 即可完成创建。

工作负载

工作负载

??? note “[点击查看创建有状态负载的 YAML 示例](#)”

```
```yaml
kind: StatefulSet
apiVersion: apps/v1
metadata:
  name: test-mysql-123-mysql
  namespace: default
  uid: d3f45527-a0ab-4b22-9013-5842a06f4e0e
  resourceVersion: '20504385'
  generation: 1
  creationTimestamp: '2022-09-22T09:34:10Z'
  ownerReferences:
    - apiVersion: mysql.presslabs.org/v1alpha1
      kind: MysqlCluster
      name: test-mysql-123
      uid: 5e877cc3-5167-49da-904e-820940cf1a6d
      controller: true
      blockOwnerDeletion: true
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/managed-by: mysql.presslabs.org
      app.kubernetes.io/name: mysql
      mysql.presslabs.org/cluster: test-mysql-123
  template:
    metadata:
      creationTimestamp: null
      labels:
```

```
app.kubernetes.io/component: database
app.kubernetes.io/instance: test-mysql-123
app.kubernetes.io/managed-by: mysql.presslabs.org
app.kubernetes.io/name: mysql
app.kubernetes.io/version: 5.7.31
mysql.presslabs.org/cluster: test-mysql-123

annotations:
  config_rev: '13941099'
  prometheus.io/port: '9125'
  prometheus.io/scrape: 'true'
  secret_rev: '13941101'

spec:
  volumes:
    - name: conf
      emptyDir: {}
    - name: init-scripts
      emptyDir: {}
    - name: config-map
      configMap:
        name: test-mysql-123-mysql
        defaultMode: 420
    - name: data
  persistentVolumeClaim:
    claimName: data
  initContainers:
    - name: init
      image: docker.m.daocloud.io/bitpoke/mysql-operator-sidecar-5.7:v0.6.1
      args:
        - clone-and-init
      envFrom:
        - secretRef:
            name: test-mysql-123-mysql-operated
      env:
        - name: MY_NAMESPACE
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.name
        - name: MY_POD_IP
```

```
valueFrom:  
  fieldRef:  
    apiVersion: v1  
    fieldPath: status.podIP  
  - name: MY_SERVICE_NAME  
    value: mysql  
  - name: MY_CLUSTER_NAME  
    value: test-mysql-123  
  - name: MY_FQDN  
    value: ${MY_POD_NAME}.${MY_SERVICE_NAME}.${MY_NAMESPACE}  
  - name: MY_MYSQL_VERSION  
    value: 5.7.31  
  - name: BACKUP_USER  
    valueFrom:  
      secretKeyRef:  
        name: test-mysql-123-mysql-operated  
        key: BACKUP_USER  
        optional: true  
  - name: BACKUP_PASSWORD  
    valueFrom:  
      secretKeyRef:  
        name: test-mysql-123-mysql-operated  
        key: BACKUP_PASSWORD  
        optional: true  
resources: {}  
volumeMounts:  
  - name: conf  
    mountPath: /etc/mysql  
  - name: config-map  
    mountPath: /mnt/conf  
  - name: data  
    mountPath: /var/lib/mysql  
terminationMessagePath: /dev/termination-log  
terminationMessagePolicy: File  
imagePullPolicy: IfNotPresent  
containers:  
  - name: mysql  
    image: docker.m.daocloud.io/mysql:5.7.31  
    ports:  
      - name: mysql  
        containerPort: 3306  
        protocol: TCP  
    env:  
      - name: MY_NAMESPACE
```

```
valueFrom:  
  fieldRef:  
    apiVersion: v1  
    fieldPath: metadata.namespace  
- name: MY_POD_NAME  
  valueFrom:  
    fieldRef:  
      apiVersion: v1  
      fieldPath: metadata.name  
- name: MY_POD_IP  
  valueFrom:  
    fieldRef:  
      apiVersion: v1  
      fieldPath: status.podIP  
- name: MY_SERVICE_NAME  
  value: mysql  
- name: MY_CLUSTER_NAME  
  value: test-mysql-123  
- name: MY_FQDN  
  value: ${MY_POD_NAME}.${MY_SERVICE_NAME}.${MY_NAMESPACE}  
- name: MY_MYSQL_VERSION  
  value: 5.7.31  
- name: ORCH_CLUSTER_ALIAS  
  value: test-mysql-123.default  
- name: ORCH_HTTP_API  
  value: http://mysql-operator.mcamel-system/api  
- name: MYSQL_ROOT_PASSWORD  
  valueFrom:  
    secretKeyRef:  
      name: test-mysql-123-secret  
      key: ROOT_PASSWORD  
      optional: false  
- name: MYSQL_USER  
  valueFrom:  
    secretKeyRef:  
      name: test-mysql-123-secret  
      key: USER  
      optional: true  
- name: MYSQL_PASSWORD  
  valueFrom:  
    secretKeyRef:  
      name: test-mysql-123-secret  
      key: PASSWORD  
      optional: true
```

```
- name: MYSQL_DATABASE
  valueFrom:
    secretKeyRef:
      name: test-mysql-123-secret
      key: DATABASE
      optional: true
  resources:
    limits:
      cpu: '1'
      memory: 1Gi
    requests:
      cpu: 100m
      memory: 512Mi
  volumeMounts:
    - name: conf
      mountPath: /etc/mysql
    - name: data
      mountPath: /var/lib/mysql
  livenessProbe:
    exec:
      command:
        - mysqladmin
        - '--defaults-file=/etc/mysql/client.conf'
        - ping
    initialDelaySeconds: 60
    timeoutSeconds: 5
    periodSeconds: 5
    successThreshold: 1
    failureThreshold: 3
  readinessProbe:
    exec:
      command:
        - /bin/sh
        - '-c'
        - '>-
          test $(mysql --defaults-file=/etc/mysql/client.conf -NB -e
          'SELECT COUNT(*) FROM sys_operator.status WHERE
          name="configured" AND value="1") -eq 1
  initialDelaySeconds: 5
  timeoutSeconds: 5
  periodSeconds: 2
  successThreshold: 1
  failureThreshold: 3
lifecycle:
```

```
preStop:  
  exec:  
    command:  
      - bash  
      - /etc/mysql/pre-shutdown-ha.sh  
terminationMessagePath: /dev/termination-log  
terminationMessagePolicy: File  
imagePullPolicy: IfNotPresent  
- name: sidecar  
  image: docker.m.daocloud.io/bitpoke/mysql-operator-sidecar-5.7:v0.6.1  
  args:  
    - config-and-serve  
  ports:  
    - name: sidecar-http  
      containerPort: 8080  
      protocol: TCP  
  envFrom:  
    - secretRef:  
        name: test-mysql-123-mysql-operated  
  env:  
    - name: MY_NAMESPACE  
      valueFrom:  
        fieldRef:  
          apiVersion: v1  
          fieldPath: metadata.namespace  
    - name: MY_POD_NAME  
      valueFrom:  
        fieldRef:  
          apiVersion: v1  
          fieldPath: metadata.name  
    - name: MY_POD_IP  
      valueFrom:  
        fieldRef:  
          apiVersion: v1  
          fieldPath: status.podIP  
    - name: MY_SERVICE_NAME  
      value: mysql  
    - name: MY_CLUSTER_NAME  
      value: test-mysql-123  
    - name: MY_FQDN  
      value: ${MY_POD_NAME}.${MY_SERVICE_NAME}.${MY_NAMESPACE}  
    - name: MY_MYSQL_VERSION  
      value: 5.7.31  
    - name: XTRABACKUP_TARGET_DIR
```

```
      value: /tmp/xtrabackup_backupfiles/
resources:
  limits:
    cpu: '1'
    memory: 1Gi
  requests:
    cpu: 10m
    memory: 64Mi
volumeMounts:
  - name: conf
    mountPath: /etc/mysql
  - name: data
    mountPath: /var/lib/mysql
readinessProbe:
  httpGet:
    path: /health
    port: 8080
    scheme: HTTP
  initialDelaySeconds: 30
  timeoutSeconds: 5
  periodSeconds: 5
  successThreshold: 1
  failureThreshold: 3
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
imagePullPolicy: IfNotPresent
- name: metrics-exporter
  image: prom/mysqld-exporter:v0.13.0
  args:
    - '--web.listen-address=0.0.0.0:9125'
    - '--web.telemetry-path=/metrics'
    - '--collect.heartbeat'
    - '--collect.heartbeat.database=sys_operator'
ports:
  - name: prometheus
    containerPort: 9125
    protocol: TCP
env:
  - name: MY_NAMESPACE
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: metadata.namespace
  - name: MY_POD_NAME
```

```
valueFrom:  
  fieldRef:  
    apiVersion: v1  
    fieldPath: metadata.name  
- name: MY_POD_IP  
  valueFrom:  
    fieldRef:  
      apiVersion: v1  
      fieldPath: status.podIP  
- name: MY_SERVICE_NAME  
  value: mysql  
- name: MY_CLUSTER_NAME  
  value: test-mysql-123  
- name: MY_FQDN  
  value: $(MY_POD_NAME).$(MY_SERVICE_NAME).$(MY_NAMESPACE)  
- name: MY_MYSQL_VERSION  
  value: 5.7.31  
- name: USER  
  valueFrom:  
    secretKeyRef:  
      name: test-mysql-123-mysql-operated  
      key: METRICS_EXPORTER_USER  
      optional: false  
- name: PASSWORD  
  valueFrom:  
    secretKeyRef:  
      name: test-mysql-123-mysql-operated  
      key: METRICS_EXPORTER_PASSWORD  
      optional: false  
- name: DATA_SOURCE_NAME  
  value: $(USER):$(PASSWORD)@(127.0.0.1:3306)/  
resources:  
  limits:  
    cpu: 100m  
    memory: 128Mi  
  requests:  
    cpu: 10m  
    memory: 32Mi  
livenessProbe:  
  httpGet:  
    path: /metrics  
    port: 9125  
    scheme: HTTP  
  initialDelaySeconds: 30
```

```
    timeoutSeconds: 30
    periodSeconds: 30
    successThreshold: 1
    failureThreshold: 3
  terminationMessagePath: /dev/termination-log
  terminationMessagePolicy: File
  imagePullPolicy: IfNotPresent
- name: pt-heartbeat
  image: docker.m.daocloud.io/bitpoke/mysql-operator-sidecar-5.7:v0.6.1
  args:
    - pt-heartbeat
    - '--update'
    - '--replace'
    - '--check-read-only'
    - '--create-table'
    - '--database'
    - sys_operator
    - '--table'
    - heartbeat
    - '--utc'
    - '--defaults-file'
    - /etc/mysql/heartbeat.conf
    - '--fail-successive-errors=20'
  env:
    - name: MY_NAMESPACE
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.namespace
    - name: MY_POD_NAME
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.name
    - name: MY_POD_IP
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: status.podIP
    - name: MY_SERVICE_NAME
      value: mysql
    - name: MY_CLUSTER_NAME
      value: test-mysql-123
    - name: MY_FQDN
```

```
        value: ${MY_POD_NAME}.${MY_SERVICE_NAME}.${MY_NAMESPACE}
      - name: MY_MYSQL_VERSION
        value: 5.7.31
    resources:
      limits:
        cpu: 100m
        memory: 64Mi
      requests:
        cpu: 10m
        memory: 32Mi
    volumeMounts:
      - name: conf
        mountPath: /etc/mysql
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
  terminationGracePeriodSeconds: 30
  dnsPolicy: ClusterFirst
  securityContext:
    runAsUser: 999
    fsGroup: 999
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchLabels:
                app.kubernetes.io/component: database
                app.kubernetes.io/instance: test-mysql-123
                app.kubernetes.io/managed-by: mysql.presslabs.org
                app.kubernetes.io/name: mysql
                app.kubernetes.io/version: 5.7.31
                mysql.presslabs.org/cluster: test-mysql-123
            topologyKey: kubernetes.io/hostname
    schedulerName: default-scheduler
  volumeClaimTemplates:
    - kind: PersistentVolumeClaim
      apiVersion: v1
      metadata:
        name: data
      creationTimestamp: null
      ownerReferences:
```

```
- apiVersion: mysql.presslabs.org/v1alpha1
  kind: MysqlCluster
  name: test-mysql-123
  uid: 5e877cc3-5167-49da-904e-820940cf1a6d
  controller: true
  spec:
    accessModes:
      - ReadWriteOnce
    resources:
      limits:
        storage: 1Gi
      requests:
        storage: 1Gi
    storageClassName: local-path
    volumeMode: Filesystem
  status:
    phase: Pending
  serviceName: mysql
  podManagementPolicy: OrderedReady
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 0
  revisionHistoryLimit: 10
status:
  observedGeneration: 1
  replicas: 1
  readyReplicas: 1
  currentReplicas: 1
  updatedReplicas: 1
  currentRevision: test-mysql-123-mysql-6b8f5577c7
  updateRevision: test-mysql-123-mysql-6b8f5577c7
  collisionCount: 0
  availableReplicas: 1
```

```

## 创建守护进程(DaemonSet)

本文介绍如何通过镜像和 YAML 文件两种方式创建守护进程 ( DaemonSet )。

守护进程 ( DaemonSet ) 通过[节点亲和性](#)与[污点](#)功能确保在全部或部分节点上运行一个

Pod 的副本。对于新加入集群的节点，DaemonSet 自动在新节点上部署相应的 Pod，并跟踪 Pod 的运行状态。当节点被移除时，DaemonSet 则删除其创建的所有 Pod。

守护进程的常见用例包括：

- 在每个节点上运行集群守护进程
- 在每个节点上运行日志收集守护进程
- 在每个节点上运行监控守护进程。

简单起见，可以在每个节点上为每种类型的守护进程都启动一个 DaemonSet。如需更精细、更高级地管理守护进程，也可以为同一种守护进程部署多个 DaemonSet。每个 DaemonSet 具有不同的标志，并且对不同硬件类型具有不同的内存、CPU 要求。

## 前提条件

创建 DaemonSet 之前，需要满足以下前提条件：

- 在[容器管理](#)模块中[接入 Kubernetes 集群](#)或者[创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建一个[命名空间](#)和[用户](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。
- 单个实例中有多个容器时，请确保容器使用的端口不冲突，否则部署会失效。

## 镜像创建

参考以下步骤，使用镜像创建一个守护进程。

1. 点击左侧导航栏上的 [集群列表](#)，然后点击目标集群的名称，进入 [集群详情](#) 页面。

[集群详情](#)

## 集群详情

2. 在集群详情页面，点击左侧导航栏的 **工作负载** -> **守护进程**，然后点击页面右上角的 **镜像创建** 按钮。

### 工作负载

#### 工作负载

3. 依次填写 基本信息、容器配置、服务配置、高级配置后，在页面右下角点击 **确定** 完成创建。

系统将自动返回 **守护进程** 列表。点击列表右侧的 ，可以对守护进程执行执行更新、删除、重启等操作。

### 操作菜单

#### 操作菜单

## 基本信息

在 **创建守护进程** 页面中，根据下表输入信息后，点击 **下一步**。

### 基本信息

#### 基本信息

- 负载名称：最多包含 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾。同一命名空间内同一类型工作负载的名称不得重复，而且负载名称在工作负载创建好之后不可更改。
- 命名空间：选择将新建的守护进程部署在哪个命名空间，默认使用 default 命名空间。  
找不到所需的命名空间时可以根据页面提示去[创建新的命名空间](#)。
- 描述：输入工作负载的描述信息，内容自定义。字符数量应不超过 512 个。

## 容器配置

容器配置分为基本信息、生命周期、健康检查、环境变量、数据存储、安全设置六部分，

点击下方的相应页签可查看各部分的配置要求。

容器配置仅针对单个容器进行配置，如需在一个容器组中添加多个容器，可点击右侧的 **+** 添加多个容器。

### ==== “基本信息（必填）”

在配置容器相关参数时，必须正确填写容器的名称、镜像参数，否则将无法进入下一步。参考以下要求填写配置后，点击 **确认**。

![基本信息](<https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/daemon06.png>)

- 容器类型：默认为`工作容器`。有关初始化容器，参见 [\[k8s 官方文档\]\(https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/init-containers/\)](https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/init-containers/)。
- 容器名称：最多包含 63 个字符，支持小写字母、数字及分隔符（“-”）。必须以小写字母或数字开头及结尾，例如 `nginx-01`。
- 镜像：
  - 容器镜像：从列表中选择一个合适的镜像。输入镜像名称时，默认从官方的 [\[DockerHub\]\(https://hub.docker.com/\)](https://hub.docker.com/) 拉取镜像。  
接入 DCE 5.0 的[镜像仓库](../../../../kangaroo/intro/index.md)模块后，可以点击右侧的 **选择镜像** 按钮来选择镜像。
  - 镜像版本：从下拉列表选择一个合适的版本。
  - 镜像拉取策略：勾选 **总是拉取镜像** 后，负载每次重启/升级时都会从仓库重新拉取镜像。  
如果不勾选，则只拉取本地镜像，只有当镜像在本地不存在时才从镜像仓库重新拉取。  
更多详情可参考[\[镜像拉取策略\]\(https://kubernetes.io/zh-cn/docs/concepts/containers/images/#image-pull-policy\)](https://kubernetes.io/zh-cn/docs/concepts/containers/images/#image-pull-policy)。
- 镜像仓库密钥：可选。如果目标仓库需要 `Secret` 才能访问，需要先去[\[创建一个密钥\]\(./configmaps-secrets/create-secret.md\)](#)。
- 特权容器：容器默认不可以访问宿主机上的任何设备，开启特权容器后，容器即可访问宿主机上的所有设备，享有宿主机上的运行进程的所有权限。
- CPU/内存配额：CPU/内存资源的请求值（需要使用的最小资源）和限制值（允许使用的最大资源）。请根据需要为容器配置资源，避免资源浪费和因容器资源超额导致系统故障。默认值如图所示。
- GPU 配置：为容器配置 GPU 用量，仅支持输入正整数。
  - 整卡模式：
    - 物理卡数量：容器能够使用的物理 GPU 卡数量。配置后，容器将占用整张物理 GPU 卡。同时物理卡数量需要 ≤ 单节点插入的最大 GPU 卡数。

- 虚拟化模式：
  - 物理卡数量：容器能够使用的物理 GPU 卡数量，物理卡数量需要  $\leq$  单节点插入的最大 GPU 卡数。
  - GPU 算力：每张物理 GPU 卡上需要使用的算力百分比，最多为 100%。
  - 显存：每张物理卡上需要使用的显存数量。
  - 调度策略（Binpack/Spread）：支持基于 GPU 卡和基于节点的两种维度的调度策略。Binpack 是集中式调度策略，优先将容器调度到同一个节点的同一张 GPU 卡上；Spread 是分散式调度策略，优先将容器调度到不同节点的不同 GPU 卡上，根据实际场景可组合使用。  
(当工作负载级别的 Binpack/Spread 调度策略与集群级别的 Binpack/Spread 调度策略冲突时，系统优先使用工作负载级别的调度策略)。
  - 任务优先级：GPU 算力会优先供给高优先级任务使用，普通任务会减少甚至暂停使用 GPU 算力，直到高优先级任务结束，普通任务会重新继续使用 GPU 算力，常用于在离线混部场景。
  - 指定型号：将工作负载调度到指定型号的 GPU 卡上，适用于对 GPU 型号有特殊要求的场景。
    - Mig 模式
      - 规格：切分后的物理 GPU 卡规格。
      - 数量：使用该规格的数量。

> 设置 GPU 之前，需要管理员预先在集群上安装 [GPU Operator](../gpu/nvidia/install\_nvidia\_driver\_of\_operator.md) 和 [nvidia-vgpu](../gpu/nvidia/vgpu/vgpu\_addon.md) (仅 vGPU 模式需要安装)，并在[集群设置](../clusterops/cluster-settings.md)中开启 GPU 特性。

#### ==== “生命周期 (选填)”

设置容器启动时、启动后、停止前需要执行的命令。详情可参考[容器生命周期配置](pod-config/lifecycle.md)。

![生命周期](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy06.png)

#### ==== “健康检查 (选填)”

用于判断容器和应用的健康状态，有助于提高应用的可用性。详情可参考[容器健康检查配置](pod-config/health-check.md)。

![健康检查](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy07.png)

#### ==== “环境变量 (选填)”

配置 Pod 内的容器参数，为 Pod 添加环境变量或传递配置等。详情可参考[容器环境变量配置](pod-config/env-variables.md)。

![环境变量](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy08.png)

#### ==== “数据存储 (选填)”

配置容器挂载数据卷和数据持久化的设置。详情可参考[容器数据存储配置](pod-config/env-variables.md)。

![数据存储](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy09.png)

==== “安全设置（选填）”

通过 Linux 内置的账号权限隔离机制来对容器进行安全隔离。您可以通过使用不同权限的账号 UID（数字身份标记）来限制容器的权限。例如，输入 \_0\_ 表示使用 root 账号的权限。

![安全设置](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy10.png)

## 服务配置

为守护进程创建[服务（Service）](#)，使守护进程能够被外部访问。

1. 点击 **创建服务** 按钮。

服务配置

服务配置

2. 配置服务参数，详情请参考[创建服务](#)。

创建服务

创建服务

3. 点击 **确定**，点击 **下一步**。

## 高级配置

高级配置包括负载的网络配置、升级策略、调度策略、标签与注解四部分，可点击下方的页签查看各部分的配置要求。

==== “网络配置”

应用在某些场景下会出现冗余的 DNS 查询。Kubernetes 为应用提供了与 DNS 相关的配置选项，能够在某些场景下有效地减少冗余的 DNS 查询，提升业务并发量。

![DNS 配置](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy17.png)

- DNS 策略

- **Default:** 使容器使用 kubelet 的 `--resolv-conf` 参数指向的域名解析文件。该配置只能解析注册到互联网上的外部域名，无法解析集群内部域名，且不存在无效的 DNS 查询。
- **ClusterFirstWithHostNet:** 应用对接主机的域名文件。
- **ClusterFirst:** 应用对接 Kube-DNS/CoreDNS。
- **None:** Kubernetes v1.9 (Beta in v1.10) 中引入的新选项值。设置为 None 之后，必须设置 dnsConfig，此时容器的域名解析文件将完全通过 dnsConfig 的配置来生成。
  
- **域名服务器:** 填写域名服务器的地址，例如 `10.6.175.20`。
- **搜索域:** 域名查询时的 DNS 搜索域列表。指定后，提供的搜索域列表将合并到基于 `dnsPolicy` 生成的域名解析文件的 `search` 字段中，并删除重复的域名。Kubernetes 最多允许 6 个搜索域。
- **Options:** DNS 的配置选项，其中每个对象可以具有 `name` 属性（必需）和 `value` 属性（可选）。该字段中的内容将合并到基于 `dnsPolicy` 生成的域名解析文件的 `options` 字段中，`dnsConfig` 的 `options` 的某些选项如果与基于 `dnsPolicy` 生成的域名解析文件的选项冲突，则会被 `dnsConfig` 所覆盖。
- **主机别名:** 为主机设置的别名。

#### ==== “升级策略”

![升级策略](docs/zh/docs/kpanda/images/deploy14.png)

- **升级方式:**
  - **滚动升级(RollingUpdate)** 指逐步用新版本的实例替换旧版本的实例，升级的过程中，业务流量会同时负载均衡分布到新老的实例上，因此业务不会中断。
  - **替换升级(OnDelete)** 指先删除老版本的负载实例，再安装指定的新版本，升级过程中业务会中断。
- **最大不可用:** 指定负载更新过程中不可用 Pod 的最大值或比率，默认 25%。如果等于实例数有服务中断的风险。
- **最大保留版本数:** 设置版本回滚时保留的旧版本数量。默认 10。
- **Pod 可用最短时间:** Pod 就绪的最短时间，只有超出这个时间 Pod 才被认为可用，默认 0 秒。
- **缩容时间窗:** 负载停止前命令的执行时间窗（0-9,999 秒），默认 30 秒。

#### ==== “调度策略”

![调度策略](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy15.png)

- **容忍时间:** 负载实例所在的节点不可用时，将负载实例重新调度到其它可用节点的时间，默认为 300 秒。
- **节点亲和性:** 根据节点上的标签来约束 Pod 可以调度到哪些节点上。
- **工作负载亲和性:** 基于已经在节点上运行的 Pod 的标签来约束 Pod 可以调度到哪些节点。
- **工作负载反亲和性:** 基于已经在节点上运行的 Pod 的标签来约束 Pod 不可以调度到哪些节点。
- **拓扑域:** 即 `topologyKey`，用于指定可以调度的一组节点。例如，`kubernetes.io/os` 表示只要某个操作系统的节点满足 `labelSelector` 的条件就可以调度到该节点。

> 具体详情请参考[调度策略](pod-config/scheduling-policy.md)。

==== “标签与注解”

可以点击 添加 按钮为工作负载和容器组添加标签和注解。

![标签与注解](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy16.png)

## YAML 创建

除了通过镜像方式外，还可以通过 YAML 文件更快速地创建守护进程。

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情** 页面。

集群详情

集群详情

2. 在集群详情页面，点击左侧导航栏的 **工作负载 -> 守护进程**，然后点击页面右上角

的 **YAML 创建** 按钮。

工作负载

工作负载

3. 输入或粘贴事先准备好的 YAML 文件，点击 **确定** 即可完成创建。

工作负载

工作负载

??? note “点击查看创建守护进程的 YAML 示例”

```
```yaml
kind: DaemonSet
apiVersion: apps/v1
metadata:
  name: hwameistor-local-disk-manager
  namespace: hwameistor
  uid: ccbdc098-7de3-4a8a-96dd-d1cee159c92b
  resourceVersion: '90999552'
  generation: 1
  creationTimestamp: '2022-12-15T09:03:44Z'
```

```
labels:  
  app.kubernetes.io/managed-by: Helm  
annotations:  
  deprecated.daemonset.template.generation: '1'  
  meta.helm.sh/release-name: hwameistor  
  meta.helm.sh/release-namespace: hwameistor  
spec:  
  selector:  
    matchLabels:  
      app: hwameistor-local-disk-manager  
  template:  
    metadata:  
      creationTimestamp: null  
      labels:  
        app: hwameistor-local-disk-manager  
    spec:  
      volumes:  
        - name: udev  
          hostPath:  
            path: /run/udev  
            type: Directory  
        - name: procmount  
          hostPath:  
            path: /proc  
            type: Directory  
        - name: devmount  
          hostPath:  
            path: /dev  
            type: Directory  
        - name: socket-dir  
          hostPath:  
            path: /var/lib/kubelet/plugins/disk.hwameistor.io  
            type: DirectoryOrCreate  
        - name: registration-dir  
          hostPath:  
            path: /var/lib/kubelet/plugins_registry/  
            type: Directory  
        - name: plugin-dir  
          hostPath:  
            path: /var/lib/kubelet/plugins  
            type: DirectoryOrCreate  
        - name: pods-mount-dir  
          hostPath:  
            path: /var/lib/kubelet/pods
```

```
type: DirectoryOrCreate
containers:
  - name: registrar
    image: k8s-gcr.m.daocloud.io/sig-storage/csi-node-driver-registrar:v2.5.0
    args:
      - '--v=5'
      - '--csi-address=/csi/csi.sock'
      - '>--kubelet-registration-path=/var/lib/kubelet/plugins/disk.hwameistor.io/csi.sock'
    env:
      - name: KUBE_NODE_NAME
        valueFrom:
          fieldRef:
            apiVersion: v1
            fieldPath: spec.nodeName
    resources: {}
  volumeMounts:
    - name: socket-dir
      mountPath: /csi
    - name: registration-dir
      mountPath: /registration
  lifecycle:
    preStop:
      exec:
        command:
          - /bin/sh
          - '-c'
          - '>rm -rf /registration/disk.hwameistor.io
           /registration/disk.hwameistor.io-reg.sock'
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    imagePullPolicy: IfNotPresent
  - name: manager
    image: ghcr.m.daocloud.io/hwameistor/local-disk-manager:v0.6.1
    command:
      - /local-disk-manager
    args:
      - '--endpoint=$(CSI_ENDPOINT)'
      - '--nodeid=$(NODENAME)'
      - '--csi-enable=true'
    env:
      - name: CSI_ENDPOINT
        value: unix://var/lib/kubelet/plugins/disk.hwameistor.io/csi.sock
```

```
- name: NAMESPACE
  valueFrom:
    fieldRef:
      apiVersion: v1
      fieldPath: metadata.namespace
- name: WATCH_NAMESPACE
  valueFrom:
    fieldRef:
      apiVersion: v1
      fieldPath: metadata.namespace
- name: POD_NAME
  valueFrom:
    fieldRef:
      apiVersion: v1
      fieldPath: metadata.name
- name: NODENAME
  valueFrom:
    fieldRef:
      apiVersion: v1
      fieldPath: spec.nodeName
- name: OPERATOR_NAME
  value: local-disk-manager
resources: {}
volumeMounts:
- name: udev
  mountPath: /run/udev
- name: procmount
  readOnly: true
  mountPath: /host/proc
- name: devmount
  mountPath: /dev
- name: registration-dir
  mountPath: /var/lib/kubelet/plugins_registry
- name: plugin-dir
  mountPath: /var/lib/kubelet/plugins
  mountPropagation: Bidirectional
- name: pods-mount-dir
  mountPath: /var/lib/kubelet/pods
  mountPropagation: Bidirectional
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
imagePullPolicy: IfNotPresent
securityContext:
  privileged: true
```

```
restartPolicy: Always
terminationGracePeriodSeconds: 30
dnsPolicy: ClusterFirst
serviceAccountName: hwameistor-admin
serviceAccount: hwameistor-admin
hostNetwork: true
hostPID: true
securityContext: {}
schedulerName: default-scheduler
tolerations:
  - key: CriticalAddonsOnly
    operator: Exists
  - key: node.kubernetes.io/not-ready
    operator: Exists
    effect: NoSchedule
  - key: node-role.kubernetes.io/master
    operator: Exists
    effect: NoSchedule
  - key: node-role.kubernetes.io/control-plane
    operator: Exists
    effect: NoSchedule
  - key: node.cloudprovider.kubernetes.io/uninitialized
    operator: Exists
    effect: NoSchedule
updateStrategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 0
  revisionHistoryLimit: 10
status:
  currentNumberScheduled: 4
  numberMisscheduled: 0
  desiredNumberScheduled: 4
  numberReady: 4
  observedGeneration: 1
  updatedNumberScheduled: 4
  numberAvailable: 4
````
```

# 创建定时任务 ( CronJob )

本文介绍如何通过镜像和 YAML 文件两种方式创建定时任务 ( CronJob )。

定时任务 ( CronJob ) 适用于执行周期性的操作，例如备份、报告生成等。这些任务可以配置为周期性重复的（例如：每天/每周/每月一次），可以定义任务开始执行的时间间隔。

## 前提条件

创建定时任务 ( CronJob ) 之前，需要满足以下前提条件：

- 在[容器管理](#)模块中[接入 Kubernetes 集群](#)或者[创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建一个[命名空间](#)和[用户](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。
- 单个实例中有多个容器时，请确保容器使用的端口不冲突，否则部署会失效。

## 镜像创建

参考以下步骤，使用镜像创建一个定时任务。

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情** 页面。

集群详情

集群详情

2. 在集群详情页面，点击左侧导航栏的 **工作负载 -> 定时任务**，然后点击页面右上角的 **镜像创建** 按钮。

## 工作负载

### 工作负载

3. 依次填写基本信息、容器配置、定时任务配置、高级配置后，在页面右下角点击**确定**完成创建。

系统将自动返回**定时任务**列表。点击列表右侧的⋮，可以对定时任务执行执行删除、

**重启等操作。**

| 定时任务名称 | 定时任务别名 | 状态  | 命名空间    | 任务 (正常/总...) | 镜像    | 触发条件       | 创建时间             |
|--------|--------|-----|---------|--------------|-------|------------|------------------|
| test   | -      | 已启动 | default | 0/1          | nginx | 1 15 * * 1 | 2024-05-28 15... |

### 操作菜单

## 基本信息

在**创建定时任务**页面中，根据下表输入信息后，点击**下一步**。

## 基本信息

- 负载名称：最多包含 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾。同一命名空间内同一类型工作负载的名称不得重复，而且负载名称在工作负载创建好之后不可更改。
- 命名空间：选择将新建的定时任务部署在哪个命名空间，默认使用 default 命名空间。  
找不到所需的命名空间时可以根据页面提示去[创建新的命名空间](#)。
- 描述：输入工作负载的描述信息，内容自定义。字符数量应不超过 512 个。

## 容器配置

容器配置分为基本信息、生命周期、健康检查、环境变量、数据存储、安全设置六部分，

点击下方的相应页签可查看各部分的配置要求。

容器配置仅针对单个容器进行配置，如需在一个容器组中添加多个容器，可点击右侧的 +

添加多个容器。

==== “基本信息（必填）”

在配置容器相关参数时，必须正确填写容器的名称、镜像参数，否则将无法进入下一步。参考以下要求填写配置后，点击 确认。

![基本信息](docs/zh/docs/kpanda/user-guide/images/cronjob03.png)

- 容器类型：默认为`工作容器`。有关初始化容器，参见 [k8s 官方文档](https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/init-containers/)。
- 容器名称：最多包含 63 个字符，支持小写字母、数字及分隔符（“-”）。必须以小写字母或数字开头及结尾，例如 nginx-01。
- 镜像：

- 容器镜像：从列表中选择一个合适的镜像。输入镜像名称时，默认从官方的 [DockerHub](https://hub.docker.com/) 拉取镜像。

接入 DCE 5.0 的[镜像仓库](../../../../kangaroo/intro/index.md)模块后，可以点击右侧的 选择镜像 按钮来选择镜像。

- 镜像版本：从下拉列表选择一个合适的版本。
- 镜像拉取策略：勾选 总是拉取镜像 后，负载每次重启/升级时都会从仓库重新拉取镜像。

如果不勾选，则只拉取本地镜像，只有当镜像在本地不存在时才从镜像仓库重新拉取。

更多详情可参考[镜像拉取策略](<https://kubernetes.io/zh-cn/docs/concepts/containers/images/#image-pull-policy>)。

- 镜像仓库密钥：可选。如果目标仓库需要 Secret 才能访问，需要先去[创建一个密钥]([..../configmaps-secrets/create-secret.md](#))。

- 特权容器：容器默认不可以访问宿主机上的任何设备，开启特权容器后，容器即可访问宿主机上的所有设备，享有宿主机上的运行进程的所有权限。

- CPU/内存配额：CPU/内存资源的请求值（需要使用的最小资源）和限制值（允许使用的最大资源）。请根据需要为容器配置资源，避免资源浪费和因容器资源超额导致系统故障。默认值如图所示。

- GPU 配置：为容器配置 GPU 用量，仅支持输入正整数。

- 整卡模式：

- 物理卡数量：容器能够使用的物理 GPU 卡数量。配置后，容器将占用整张物理 GPU 卡。同时物理卡数量需要 ≤ 单节点插入的最大 GPU 卡数。

- 虚拟化模式：

- 物理卡数量：容器能够使用的物理 GPU 卡数量，物理卡数量需要 ≤ 单节点插入的最大 GPU 卡数。

- GPU 算力：每张物理 GPU 卡上需要使用的算力百分比，最多为 100%。

- 显存：每张物理卡上需要使用的显存数量。

- 调度策略（Binpack/Spread）：支持基于 GPU 卡和基于节点的两种维度的调度策略。Binpack 是集中式调度策略，优先将容器调度到同一个节点的同一张 GPU 卡上；Spread 是分散式调度策略，优先将容器调度到不同节点的不同 GPU 卡上，根据实际场景可组合使用。

（当工作负载级别的 Binpack/Spread 调度策略与集群级别的 Binpack/Spread 调度策略冲突时，系统优先使用工作负载级别的调度策略）。

- 任务优先级：GPU 算力会优先供给高优先级任务使用，普通任务会减少甚至暂停使用 GPU 算力，直到高优先级任务结束，普通任务会重新继续使用 GPU 算力，常用于在离线混部场景。

- 指定型号：将工作负载调度到指定型号的 GPU 卡上，适用于对 GPU 型号有特殊要求的场景。

- Mig 模式

- 规格：切分后的物理 GPU 卡规格。

- 数量：使用该规格的数量。

> 设置 GPU 之前，需要管理员预先在集群上安装 [GPU Operator]([..../gpu/nvidia/install\\_nvidia\\_driver\\_of\\_operator.md](#)) 和 [nvidia-vgpu]([..../gpu/nvidia/vgpu/vgpu\\_addon.md](#))（仅 vGPU 模式需要安装），并在[集群设置]([..../clusterops/cluster-settings.md](#))中开启 GPU 特性。

==== “生命周期（选填）”

设置容器启动时、启动后、停止前需要执行的命令。详情可参考[容器生命周期配置]([pod-config/lifecycle.md](#))。

![生命周期](<https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy06.png>)

==== “健康检查（选填）”

用于判断容器和应用的健康状态，有助于提高应用的可用性。详情可参考[容器健康检查配置](pod-config/health-check.md)。

![健康检查](<https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy07.png>)

**==== “环境变量（选填）”**

配置 Pod 内的容器参数，为 Pod 添加环境变量或传递配置等。详情可参考[容器环境变量配置](pod-config/env-variables.md)。

![环境变量](<https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy08.png>)

**==== “数据存储（选填）”**

配置容器挂载数据卷和数据持久化的设置。详情可参考[容器数据存储配置](pod-config/env-variables.md)。

![数据存储](<https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy09.png>)

**==== “安全设置（选填）”**

通过 Linux 内置的账号权限隔离机制来对容器进行安全隔离。您可以通过使用不同权限的账号 UID（数字身份标记）来限制容器的权限。例如，输入 `_0_` 表示使用 root 账号的权限。

![安全设置](<https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy10.png>)

## 定时任务配置

[← 创建定时任务](#)

并发策略

Allow  Forbid  Replace

定时规则

小时  日  周  月  自定义

在每周 星期一 时间为 10:14 重复执行操作。

时区 Asia/Shanghai (UTC+08:00)

任务记录 \*

保留执行成功的个数 3

保留执行失败的个数 1

其他配置

超时时间 360 秒

重试次数 6 次

重启策略 不重启  失败时重启

取消 上一步 下一步

### 定时任务配置

- **并发策略**：是否允许多个 Job 任务并行执行。
  - **Allow**：可以在前一个任务未完成时就创建新的定时任务，而且多个任务可以并行。任务太多可能抢占集群资源。
  - **Forbid**：在前一个任务完成之前，不能创建新任务，如果新任务的执行时间到了而之前的任务仍未执行完，CronJob 会忽略新任务的执行。
  - **Replace**：如果新任务的执行时间到了，但前一个任务还未完成，新的任务会取代前一个任务。

上述规则仅适用于同一个 CronJob 创建的多个任务。多个 CronJob 创建的多个任务总是允许并发执行。

- 定时规则：

- 基于分钟、小时、天、周、月设置任务执行的时间周期。支持用数字和 \* 自定义 Cron 表达式，输入表达式后下方会提示当前表达式的含义。有关详细的表达式语法规则，可参考 [Cron 时间表语法](#)。

- 时区：集成了所有 UTC 时区，可以选择其一

- 任务记录：设定保留多少条任务执行成功或失败的记录。0 表示不保留。

- 超时时间：超出该时间时，任务就会被标识为执行失败，任务下的所有 Pod 都会被删除。为空时表示不设置超时时间。默认值为 360 s。

- 重试次数：任务可重试次数，默认值为 6。

- 重启策略：设置任务失败时是否重启 Pod。

## 服务配置

为有状态负载配置[服务 \( Service \)](#)，使有状态负载能够被外部访问。

1. 点击 创建服务 按钮。

| 服务名称 | 服务别名 | 访问方式 | 访问端口 -> 容器端口 / 协议 | 创建时间 |
|------|------|------|-------------------|------|
| 暂无数据 |      |      |                   |      |

### 服务配置

2. 参考[创建服务](#)，配置服务参数。

创建服务

创建服务

3. 点击**确定**，点击**下一步**。

## 高级配置

定时任务的高级配置主要涉及标签与注解。

可以点击**添加**按钮为工作负载实例Pod添加标签和注解。

定时任务配置

定时任务配置

## YAML 创建

除了通过镜像方式外，还可以通过YAML文件更快速地创建定时任务。

1. 点击左侧导航栏上的[集群列表](#)，然后点击目标集群的名称，进入[集群详情](#)页面。

集群详情

集群详情

2. 在集群详情页面，点击左侧导航栏的[工作负载](#)→[定时任务](#)，然后点击页面右上角的**YAML创建**按钮。

工作负载

工作负载

3. 输入或粘贴事先准备好的YAML文件，点击**确定**即可完成创建。

工作负载

## 工作负载

??? note “[点击查看创建定时任务的 YAML 示例](#)”

```
```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  creationTimestamp: '2022-12-26T09:45:47Z'
  generation: 1
  name: demo
  namespace: default
  resourceVersion: '92726617'
  uid: d030d8d7-a405-4dcd-b09a-176942ef36c9
spec:
  concurrencyPolicy: Allow
  failedJobsHistoryLimit: 1
  jobTemplate:
    metadata:
      creationTimestamp: null
    spec:
      activeDeadlineSeconds: 360
      backoffLimit: 6
      template:
        metadata:
          creationTimestamp: null
        spec:
          containers:
            - image: nginx
              imagePullPolicy: IfNotPresent
              lifecycle: {}
              name: container-3
              resources:
                limits:
                  cpu: 250m
                  memory: 512Mi
                requests:
                  cpu: 250m
                  memory: 512Mi
              securityContext:
                privileged: false
              terminationMessagePath: /dev/termination-log
              terminationMessagePolicy: File
            dnsPolicy: ClusterFirst
```

```
restartPolicy: Never
schedulerName: default-scheduler
securityContext: {}
terminationGracePeriodSeconds: 30
schedule: 0 0 13 * 5
successfulJobsHistoryLimit: 3
suspend: false
status: {}
````
```

## 创建任务 ( Job )

本文介绍如何通过镜像和 YAML 文件两种方式创建任务 ( Job )。

任务 ( Job ) 适用于执行一次性任务。Job 会创建一个或多个 Pod，Job 会一直重新尝试执行 Pod，直到成功终止的 Pod 达到一定数量。成功终止的 Pod 达到指定的数量后，Job 也随之结束。删除 Job 时会一同清除该 Job 创建的所有 Pod。暂停 Job 时删除该 Job 中的所有活跃 Pod，直到 Job 被继续执行。有关任务 ( Job ) 的更多介绍，可参考 [Job](#)。

## 前提条件

- 在 [容器管理](#) 模块中 [接入 Kubernetes 集群](#) 或者 [创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建一个 [命名空间](#) 和 [用户](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考 [命名空间授权](#)。
- 单个实例中有多个容器时，请确保容器使用的端口不冲突，否则部署会失效。

## 镜像创建

参考以下步骤，使用镜像创建一个任务。

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情** 页面。

## 集群详情

### 集群详情

2. 在集群详情页面，点击左侧导航栏的 **工作负载 -> 任务**，然后点击页面右上角的 **镜像创建** 按钮。

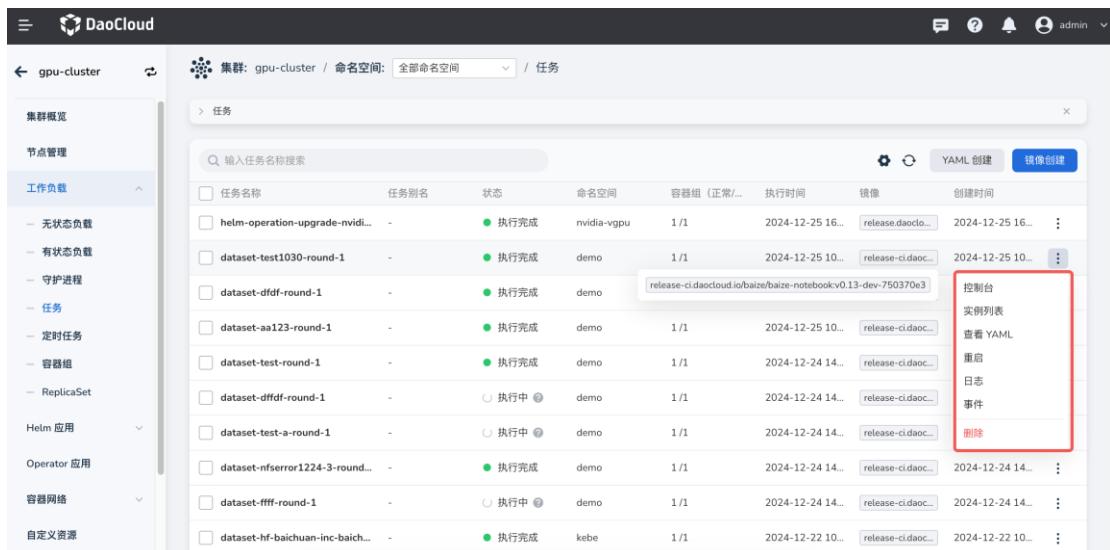
## 工作负载

### 工作负载

3. 依次填写 [基本信息](#)、[容器配置](#)、[服务配置](#)、[高级配置](#)后，在页面右下角点击 **确定完成** 创建。

系统将自动返回 **任务** 列表。点击列表右侧的 ，可以对任务执行执行删除、重启等操作。

### 操作



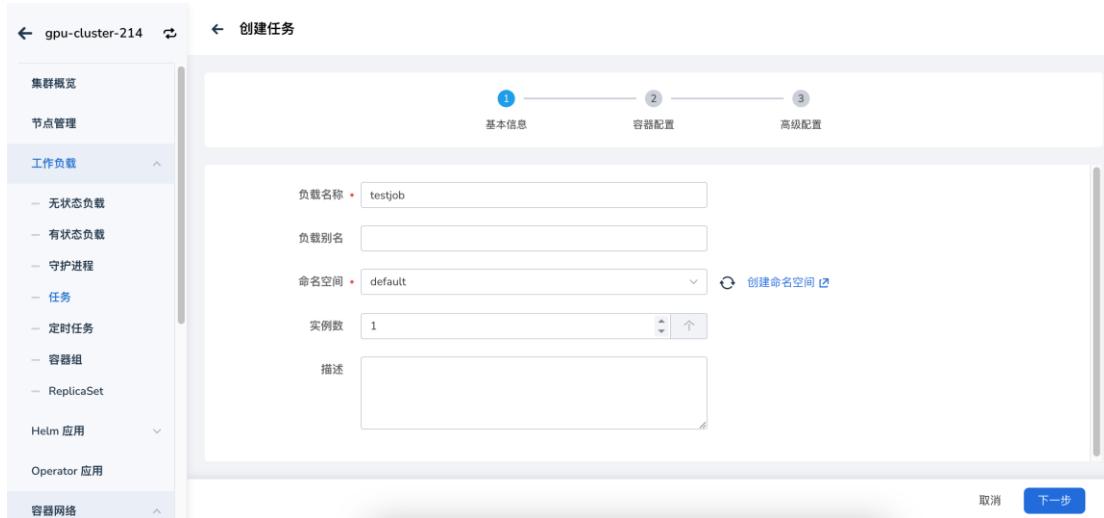
The screenshot shows the 'Tasks' section of the 'gpu-cluster' cluster details page. A context menu is open over a task named 'dataset-test1030-round-1'. The menu options are: 控制台 (Console), 实例列表 (Instance List), 查看 YAML (View YAML), 重启 (Restart), 日志 (Logs), and 事件 (Events). The '删除' (Delete) option is highlighted with a red box.

| 任务名称                             | 任务别名 | 状态        | 命名空间        | 容器组 (正常/...) | 执行时间             | 镜像                 | 创建时间             |
|----------------------------------|------|-----------|-------------|--------------|------------------|--------------------|------------------|
| helm-operation-upgrade-nvidi...  | -    | ● 执行完成    | nvidia-vgpu | 1 / 1        | 2024-12-25 16... | release.daoc...    | 2024-12-25 16... |
| dataset-test1030-round-1         | -    | ● 执行完成    | demo        | 1 / 1        | 2024-12-25 10... | release-ci.daoc... | 2024-12-25 10... |
| dataset-dffdf-round-1            | -    | ● 执行完成    | demo        | 1 / 1        | 2024-12-25 10... | release-ci.daoc... | 2024-12-25 10... |
| dataset-aa123-round-1            | -    | ● 执行完成    | demo        | 1 / 1        | 2024-12-25 10... | release-ci.daoc... | 2024-12-25 10... |
| dataset-test-round-1             | -    | ● 执行完成    | demo        | 1 / 1        | 2024-12-24 14... | release-ci.daoc... | 2024-12-24 14... |
| dataset-dffdf-round-1            | -    | ○ 执行中 (●) | demo        | 1 / 1        | 2024-12-24 14... | release-ci.daoc... | 2024-12-24 14... |
| dataset-test-a-round-1           | -    | ○ 执行中 (●) | demo        | 1 / 1        | 2024-12-24 14... | release-ci.daoc... | 2024-12-24 14... |
| dataset-nserror1224-3-round...   | -    | ● 执行完成    | demo        | 1 / 1        | 2024-12-24 14... | release-ci.daoc... | 2024-12-24 14... |
| dataset-ffff-round-1             | -    | ○ 执行中 (●) | demo        | 1 / 1        | 2024-12-24 14... | release-ci.daoc... | 2024-12-24 14... |
| dataset-hf-baichuan-inc-baich... | -    | ● 执行完成    | kebe        | 1 / 1        | 2024-12-22 10... | release-ci.daoc... | 2024-12-22 10... |

### 操作菜单

## 基本信息

在 **创建任务** 页面中，根据下表输入基本信息后，点击 **下一步**。



## 创建任务

- **负载名称**：最多包含 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾。同一命名空间内同一类型工作负载的名称不得重复，而且负载名称在工作负载创建好之后不可更改。
- **命名空间**：选择将新建的任务部署在哪个命名空间，默认使用 default 命名空间。找不到所需的命名空间时可以根据页面提示去[创建新的命名空间](#)。
- **实例数**：输入工作负载的 Pod 实例数量。默认创建 1 个 Pod 实例。
- **描述**：输入工作负载的描述信息，内容自定义。字符数量应不超过 512 个。

## 容器配置

容器配置分为基本信息、生命周期、健康检查、环境变量、数据存储、安全设置六部分，点击下方的相应页签可查看各部分的配置要求。

容器配置仅针对单个容器进行配置，如需在一个容器组中添加多个容器，可点击右侧的 + 添加多个容器。

### == “基本信息（必填）”

在配置容器相关参数时，必须正确填写容器的名称、镜像参数，否则将无法进入下一步。参考以下要求填写配置后，点击 确认。

![基本信息](docs/zh/docs/kpanda/images/job02-1.png)

- 容器类型：默认为`工作容器`。有关初始化容器，参见 [k8s 官方文档](<https://kubernetes.io/zh-cn/docs/concepts/workloads/pods/init-containers/>)。

- 容器名称：最多包含 63 个字符，支持小写字母、数字及分隔符（“-”）。必须以小写字母或数字开头及结尾，例如 nginx-01。

- 镜像：

- 容器镜像：从列表中选择一个合适的镜像。输入镜像名称时，默认从官方的 [DockerHub](<https://hub.docker.com/>) 拉取镜像。

接入 DCE 5.0 的[镜像仓库]([../../kangaroo/intro/index.md](#))模块后，可以点击右侧的 选择镜像 按钮来选择镜像。

- 镜像版本：从下拉列表选择一个合适的版本。

- 镜像拉取策略：勾选 总是拉取镜像 后，负载每次重启/升级时都会从仓库重新拉取镜像。

如果不勾选，则只拉取本地镜像，只有当镜像在本地不存在时才从镜像仓库重新拉取。

更多详情可参考[镜像拉取策略](<https://kubernetes.io/zh-cn/docs/concepts/containers/images/#image-pull-policy>)。

- 镜像仓库密钥：可选。如果目标仓库需要 Secret 才能访问，需要先去[创建一个密钥]([./configmaps-secrets/create-secret.md](#))。

- 特权容器：容器默认不可以访问宿主机上的任何设备，开启特权容器后，容器即可访问宿主机上的所有设备，享有宿主机上的运行进程的所有权限。

- CPU/内存配额：CPU/内存资源的请求值（需要使用的最小资源）和限制值（允许使用的最大资源）。请根据需要为容器配置资源，避免资源浪费和因容器资源超额导致系统故障。默认值如图所示。

- GPU 配置：为容器配置 GPU 用量，仅支持输入正整数。

- 整卡模式：

- 物理卡数量：容器能够使用的物理 GPU 卡数量。配置后，容器将占用整张物理 GPU 卡。同时物理卡数量需要  $\leq$  单节点插入的最大 GPU 卡数。

- 虚拟化模式：

- 物理卡数量：容器能够使用的物理 GPU 卡数量，物理卡数量需要  $\leq$  单节点插入的最大 GPU 卡数。

- GPU 算力：每张物理 GPU 卡上需要使用的算力百分比，最多为 100%。

- 显存：每张物理卡上需要使用的显存数量。

- 调度策略（Binpack/Spread）：支持基于 GPU 卡和基于节点的两种维度的调度策略。Binpack 是集中式调度策略，优先将容器调度到同一个节点的同一张 GPU 卡上；Spread 是分散式调度策略，优先将容器调度到不同节点的不同 GPU 卡上，根据实际场景可组合使用。

（当工作负载级别的 Binpack/Spread 调度策略与集群级别的 Binpack/Spread 调度策略冲突时，系统优先使用工作负载级别的调度策略）。

- 任务优先级：GPU 算力会优先供给高优先级任务使用，普通任务会减少甚至暂停使用 GPU 算力，直到高优先级任务结束，普通任务会重新继续使用 GPU 算力，常用于在离线混部场景。

- 指定型号：将工作负载调度到指定型号的 GPU 卡上，适用于对 GPU 型号有特殊要求的场景。

- Mig 模式
  - 规格：切分后的物理 GPU 卡规格。
  - 数量：使用该规格的数量。

!!! tip

设置 GPU 之前，需要管理员预先在集群上安装

[GPU Operator](../gpu/nvidia/install\_nvidia\_driver\_of\_operator.md)  
和 [nvidia-vgpu](../gpu/nvidia/vgpu/vgpu\_addon.md)（仅 vGPU 模式需要安装），  
并在[集群设置](../clusterops/cluster-settings.md)中开启 GPU 特性。

==== “生命周期（选填）”

设置容器启动时、启动后、停止前需要执行的命令。

详情可参考[容器生命周期配置](pod-config/lifecycle.md)。

![生命周期](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy06.png)

==== “健康检查（选填）”

用于判断容器和应用的健康状态，有助于提高应用的可用性。

详情可参考[容器健康检查配置](pod-config/health-check.md)。

![健康检查](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy07.png)

==== “环境变量（选填）”

配置 Pod 内的容器参数，为 Pod 添加环境变量或传递配置等。

详情可参考[容器环境变量配置](pod-config/env-variables.md)。

![环境变量](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy08.png)

==== “数据存储（选填）”

配置容器挂载数据卷和数据持久化的设置。

详情可参考[容器数据存储配置](pod-config/env-variables.md)。

![数据存储](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy09.png)

==== “安全设置（选填）”

通过 Linux 内置的账号权限隔离机制来对容器进行安全隔离。您可以通过使用不同权限的账号

UID

（数字身份标记）来限制容器的权限。例如，输入 `_0_` 表示使用 root 账号的权限。

![安全设置](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/deploy10.png)

## 高级配置

高级配置包括任务设置、标签与注解两部分。

### ==== “任务设置”

![任务设置](docs/zh/docs/kpanda/images/job03.png)

- 并行数：任务执行过程中允许同时创建的最大 Pod 数，平行数应不大于 Pod 总数。默认为 1。
- 超时时间：超出该时间时，任务会被标识为执行失败，任务下的所有 Pod 都会被删除。为空时表示不设置超时时间。
- 重启策略：设置失败时是否重启 Pod。

### ==== “标签与注解”

可以点击 添加 按钮为工作负载实例 Pod 添加标签和注解。

![标签与注解](docs/zh/docs/kpanda/images/job04.png)

## YAML 创建

除了通过镜像方式外，还可以通过 YAML 文件更快速地创建任务。

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情** 页面。

集群详情

集群详情

2. 在集群详情页面，点击左侧导航栏的 **工作负载 -> 任务**，然后点击页面右上角的

**YAML 创建** 按钮。

工作负载

工作负载

3. 输入或粘贴事先准备好的 YAML 文件，点击 **确定** 即可完成创建。

工作负载

工作负载

这里有一个 YAML 示例供参考：

```
kind: Job
apiVersion: batch/v1
metadata:
 name: helm-operation-install-gpu-operator-r9qgkbfzr
 generateName: helm-operation-install-gpu-operator-r9qgk
 namespace: gpu-operator
 uid: b4eeba61-9c6a-4198-a3c3-3a1bb59b637a
 resourceVersion: '419300'
 generation: 1
 creationTimestamp: '2024-12-09T09:10:57Z'
 labels:
 batch.kubernetes.io/controller-uid: b4eeba61-9c6a-4198-a3c3-3a1bb59b637a
 batch.kubernetes.io/job-name: helm-operation-install-gpu-operator-r9qgkbfzr
 controller-uid: b4eeba61-9c6a-4198-a3c3-3a1bb59b637a
 job-name: helm-operation-install-gpu-operator-r9qgkbfzr
 sidecar.istio.io/inject: 'false'
 annotations:
 batch.kubernetes.io/job-tracking: ""
 job.kpanda.io/last-reversion-handle-uid: b4eeba61-9c6a-4198-a3c3-3a1bb59b637a
 revisions: >-
 {"1":{"status":"running","desire":1,"uid":"b4eeba61-9c6a-4198-a3c3-3a1bb59b637a","start-tim
e":"2024-12-09T09:10:57Z","completion-time":"0001-01-01T00:00:00Z"}}
 ownerReferences:
 - apiVersion: helm.kpanda.io/v1alpha1
 kind: HelmApp
 name: gpu-operator
 uid: bc2d26b4-651d-444b-bfe2-76a51b364201
spec:
 parallelism: 1
 completions: 1
 activeDeadlineSeconds: 1800
 backoffLimit: 3
 selector:
 matchLabels:
 batch.kubernetes.io/controller-uid: b4eeba61-9c6a-4198-a3c3-3a1bb59b637a
 template:
 metadata:
 generateName: helm-operation-
 namespace: gpu-operator
 creationTimestamp: null
 labels:
 batch.kubernetes.io/controller-uid: b4eeba61-9c6a-4198-a3c3-3a1bb59b637a
```

```
batch.kubernetes.io/job-name: helm-operation-install-gpu-operator-r9qgkbfzr
controller-uid: b4eeba61-9c6a-4198-a3c3-3a1bb59b637a
job-name: helm-operation-install-gpu-operator-r9qgkbfzr
sidecar.istio.io/inject: 'false'

spec:
 volumes:
 - name: kubeconfig
 secret:
 secretName: helm-operation-kubeconfig-h8td2
 defaultMode: 420
 - name: data
 secret:
 secretName: helm-operation-8t4pc
 defaultMode: 420
 containers:
 - name: helm
 image: release.daocloud.io/kpanda/kpanda-shell:v0.0.11
 command:
 - helm-cmd
 workingDir: /home/shell/helm
 env:
 - name: KUBECONFIG
 value: /home/.kube/config
 resources:
 limits:
 cpu: 100m
 memory: 400Mi
 requests:
 cpu: 100m
 memory: 400Mi
 volumeMounts:
 - name: data
 readOnly: true
 mountPath: /home/shell/helm
 - name: kubeconfig
 mountPath: /home/.kube
 terminationMessagePath: /dev/termination-log
 terminationMessagePolicy: File
 imagePullPolicy: IfNotPresent
 restartPolicy: Never
 terminationGracePeriodSeconds: 0
 dnsPolicy: ClusterFirst
 nodeSelector:
 kubernetes.io/os: linux
```

```
securityContext: {}
schedulerName: default-scheduler
completionMode: NonIndexed
suspend: false
status:
 conditions:
 - type: Complete
 status: 'True'
 lastProbeTime: '2024-12-09T09:11:23Z'
 lastTransitionTime: '2024-12-09T09:11:23Z'
 startTime: '2024-12-09T09:10:57Z'
 completionTime: '2024-12-09T09:11:23Z'
 succeeded: 1
 uncountedTerminatedPods: {}
 ready: 0
```

## 工作负载状态

工作负载是运行在 Kubernetes 上的一个应用程序，在 Kubernetes 中，无论您的应用程序是由单个同一组件或是由多个不同的组件构成，都可以使用一组 Pod 来运行它。

Kubernetes 提供了五种内置的工作负载资源来管理 Pod：

- [无状态工作负载](#)
- [有状态工作负载](#)
- [守护进程](#)
- [任务](#)
- [定时任务](#)

您也可以通过设置[自定义资源 CRD](#) 来实现对工作负载资源的扩展。在第五代容器管理中，支持对工作负载进行创建、更新、扩容、监控、日志、删除、版本管理等全生命周期管理。

## Pod 状态

Pod 是 Kubernetes 中创建和管理的、**最小的计算单元**，即一组容器的集合。这些容器共享存储、网络以及管理控制容器运行方式的策略。Pod 通常不由用户直接创建，而是通过工作负载资源来创建。Pod 遵循一个预定义的生命周期，起始于 [Pending 阶段](#)，如果至少其中一个主要容器正常启动，则进入 [Running](#)，之后取决于 Pod 中是否有容器以失败状态结束而进入 [Succeeded](#) 或者 [Failed](#) 阶段。

## 工作负载状态

第五代容器管理模块依据 Pod 的状态、副本数等因素，设计了一种内置的工作负载生命周期的状态集，以让用户能够更加真实的感知工作负载运行情况。由于不同的工作负载类型（比如无状态工作负载和任务）对 Pod 的管理机制不一致，因此，不同的工作负载在运行过程中会呈现不同的生命周期状态，具体如下表：

### 无状态负载、有状态负载、守护进程状态

状态 | 描述 |

:-----|:-----|

等待中 | 1. 工作负载创建正在进行中，工作负载处于此状态。2. 触发升级或者回滚动作后，工作负载处于此状态。3. 触发暂停/扩缩容等操作，工作负载处在此状态。 |

运行中 | 负载下的所有实例都在运行中且副本数与用户预定义的数量一致时处于此状态。 |

|

删除中 | 执行删除操作时，负载处于此状态，直到删除成功。 |

异常 | 因为某些原因无法取得工作负载的状态。这种情况通常是因为与 Pod 所在主机通信失败。 |

未就绪 | 容器处于异常，pending 状态时，因未知错误导致负载无法启动时显示此状态 |

## 任务状态

状态 | 描述 |

:--- | :----- |

等待中 | 任务创建正在进行中，工作负载处于此状态。 |

执行中 | 任务正在执行中，工作负载处于此状态。 |

执行完成 | 任务执行完成，工作负载处于此状态。 |

删除中 | 触发删除操作，工作负载处在此状态。 |

异常 | 因为某些原因无法取得 Pod 的状态。这种情况通常是因为与 Pod 所在主机通信失败。 |

## 定时任务状态

状态 | 描述 |

:--- | :----- |

等待中 | 定时任务创建正在进行中，定时任务处于此状态。 |

已启动 | 创建定时任务成功后，正常运行或将已暂停的任务启动时定时任务处于此状态。

|

已停止 | 执行停止任务操作时，定时任务处于此状态。 |

删除中 | 触发删除操作，定时任务处在此状态。 |

当工作负载处于异常或未就绪状态时，您可以通过将鼠标移动到负载的状态值上，系统将通过提示框展示更加详细的错误信息。您也可以通过查看[日志](#)或事件来获取工作负载的相关运行信息。

# 任务参数说明

根据 `.spec.completions` 和 `.spec.Parallelism` 的设置，可以将任务（Job）划分为以下几种

类型：

| Job 类型          | 说明                                                       |
|-----------------|----------------------------------------------------------|
| 非并行 Job         | 创建一个 Pod 直至其 Job 成功结束                                    |
| 具有确定完成计数的并行 Job | 当成功的 Pod 个数达到 <code>.spec.completions</code> 时，Job 被视为完成 |
| 并行 Job          | 创建一个或多个 Pod 直至有一个成功结束                                    |

## 参数说明

|                                          |                                                                                                                                                                                               |
|------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>RestartPolicy</code>               | 创建一个 Pod 直至其成功结束                                                                                                                                                                              |
| <code>.spec.completions</code>           | 表示 Job 结束需要成功运行的 Pod 个数，默认为 1                                                                                                                                                                 |
| <code>.spec.parallelism</code>           | 表示并行运行的 Pod 的个数，默认为 1                                                                                                                                                                         |
| <code>spec.backoffLimit</code>           | 表示失败 Pod 的重试最大次数，超过这个次数不会继续重试。                                                                                                                                                                |
| <code>.spec.activeDeadlineSeconds</code> | 表示 Pod 运行时间，一旦达到这个时间，Job 即其所有的 Pod 都会停止。且 <code>activeDeadlineSeconds</code> 优先级高于 <code>backoffLimit</code> ，即到达 <code>activeDeadlineSeconds</code> 的 Job 会忽略 <code>backoffLimit</code> 的设置。 |

以下是一个 Job 配置示例，保存在 `myjob.yaml` 中，其计算  $\pi$  到 2000 位并打印输出。

```
apiVersion: batch/v1
kind: Job #当前资源的类型
metadata:
 name: myjob
```

```

spec:
 completions: 50 # Job 结束需要运行 50 个 Pod, 这个示例中就是打印 π 50 次
 parallelism: 5 # 并行 5 个 Pod
 backoffLimit: 5 # 最多重试 5 次
 template:
 spec:
 containers:
 - name: pi
 image: perl
 command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
 restartPolicy: Never # 重启策略

```

## 相关命令

```

kubectl apply -f myjob.yaml # 启动 job
kubectl get job # 查看这个 job
kubectl logs myjob-1122dswzs 查看 Job Pod 的日志

```

# 配置容器生命周期

Pod 遵循一个预定义的生命周期，起始于 **Pending** 阶段，如果 Pod 内至少有一个容器正常启动，则进入 **Running** 状态。如果 Pod 中有容器以失败状态结束，则状态变为 **Failed**。以下 **phase** 字段值表明了一个 Pod 处于生命周期的哪个阶段。

值 | 描述

--|:— **Pending** ( 悬决 ) | Pod 已被系统接受，但有一个或者多个容器尚未创建亦未运行。  
这个阶段包括等待 Pod 被调度的时间和通过网络下载镜像的时间。 **Running** ( 运行中 )  
| Pod 已经绑定到了某个节点，Pod 中的所有容器都已被创建。至少有一个容器仍在运行，或者正处于启动或重启状态。 **Succeeded** ( 成功 ) | Pod 中的所有容器都已成功终止，并且不会再重启。 **Failed** ( 失败 ) | Pod 中的所有容器都已终止，并且至少有一个容器是因为失败而终止。也就是说，容器以非 0 状态退出或者被系统终止。 **Unknown** ( 未知 )  
| 因为某些原因无法取得 Pod 的状态，这种情况通常是因为与 Pod 所在主机通信失败所致。

在 DCE 容器管理中创建一个工作负载时，通常使用镜像来指定容器中的运行环境。默认情况下，在构建镜像时，可以通过 **Entrypoint** 和 **CMD** 两个字段来定义容器运行时执行的命令和参数。如果需要更改容器镜像启动前、启动后、停止前的命令和参数，可以通过设置容器的生命周期事件命令和参数，来覆盖镜像中默认的命令和参数。

## 生命周期配置

根据业务需要对容器的启动命令、启动后命令、停止前命令进行配置。

参数 | 说明 | 举例值 |

: - | : - | : - - |

启动命令 | 【类型】选填【含义】容器将按照启动命令进行启动 ||

启动后命令 | 【类型】选填【含义】容器启动后发出的命令 ||

停止前命令 | 【类型】选填【含义】容器在收到停止命令后执行的命令。通知应用停止接收新请求，完成已有请求后再终止，避免数据丢失或请求中断 | - |

## 启动命令

根据下表对启动命令进行配置。

参数 | 说明 | 举例值 |

: - | : - | : - - |

运行命令 | 【类型】必填【含义】输入可执行的命令，多个命令之间用空格进行分割，如命令本身带空格，则需要加（“”）。【含义】多命令时，运行命令建议用 /bin/sh 或其他的 Shell，其他全部命令作为参数来传入。 | /run/server |

运行参数 | 【类型】选填【含义】输入控制容器运行命令参数 | port=8080 |

## 启动后命令

DCE 提供命令行脚本和 HTTP 请求两种处理类型对启动后命令进行配置。您可以根据下表选择适合您的配置方式。

### 命令行脚本配置

参数 | 说明 | 举例值 |

: - | : - | : - - |

运行命令 | 【类型】选填【含义】输入可执行的命令，多个命令之间用空格进行分割，如命令本身带空格，则需要加（“”）。【含义】多命令时，运行命令建议用 /bin/sh 或其他的 Shell，其他全部命令作为参数来传入 | /run/server |

运行参数 | 【类型】选填【含义】输入控制容器运行命令参数 | port=8080 |

## 停止前命令

DCE 提供命令行脚本和 HTTP 请求两种处理类型对停止前命令进行配置。您可以根据下表选择适合您的配置方式。

### HTTP 请求配置

参数 | 说明 | 举例值 |

: - | : - | : - - |

URL 路径 | 【类型】选填【含义】请求的 URL 路径。【含义】多命令时，运行命令建议用 /bin/sh 或其他的 Shell，其他全部命令作为参数来传入。 | /run/server |

端口 | 【类型】必填【含义】请求的端口 | port=8080 |

节点地址 | 【类型】选填【含义】请求的 IP 地址，默认是容器所在的节点 IP | - |

# 配置环境变量

环境变量是指容器运行环境中设定的一个变量，用于给 Pod 添加环境标志或传递配置等，

支持通过键值对的形式为 Pod 配置环境变量。

DCE 容器管理在原生 Kubernetes 的基础上增加了图形化界面为 Pod 配置环境变量，支持以下几种配置方式：

- **键值对 ( Key/Value Pair )**：将自定义的键值对作为容器的环境变量
- **资源引用 ( Resource )**：将 Container 定义的字段作为环境变量的值，例如容器的内存限制、副本数等
- **变量/变量引用 ( Pod Field )**：将 Pod 字段作为环境变量的值，例如 Pod 的名称
- **配置项键值导入 ( ConfigMap key )**：导入配置项中某个键的值作为某个环境变量的值
- **密钥键值导入 ( Secret Key )**：使用来自 Secret 中的数据定义环境变量的值
- **密钥导入 ( Secret )**：将 Secret 中的所有键值都导入为环境变量
- **配置项导入 ( ConfigMap )**：将配置项中所有键值都导入为环境变量

# 容器的健康检查

容器健康检查根据用户需求，检查容器的健康状况。配置后，容器内的应用程序如果异常，容器会自动进行重启恢复。Kubernetes 提供了存活 ( Liveness ) 检查、就绪 ( Readiness ) 检查和启动 ( Startup ) 检查。

- **存活检查 ( LivenessProbe )** 可探测到应用死锁（应用程序在运行，但是无法继续执行后面的步骤）情况。重启这种状态下的容器有助于提高应用的可用性，即使其中

存在缺陷。

- **就绪检查 ( ReadinessProbe )** 可探知容器何时准备好接受请求流量，当一个 Pod 内的所有容器都就绪时，才能认为该 Pod 就绪。这种信号的一个用途就是控制哪个 Pod 作为 Service 的后端。若 Pod 尚未就绪，会被从 Service 的负载均衡器中剔除。
- **启动检查 ( StartupProbe )** 可以了解应用容器何时启动，配置后，可控制容器在启动成功后再进行存活性和就绪态检查，确保这些存活、就绪探测器不会影响应用的启动。启动探测可以用于对慢启动容器进行存活性检测，避免它们在启动运行之前就被杀掉。

## 存活和就绪检查

存活检查 ( LivenessProbe ) 的配置和就绪检查 ( ReadinessProbe ) 的配置参数相似，唯一区别是要使用 `readinessProbe` 字段，而不是 `livenessProbe` 字段。

**HTTP GET 参数说明：**

| 参数                           | 参数说明                                                                         |
|------------------------------|------------------------------------------------------------------------------|
| 路径 ( Path )                  | 访问的请求路径。如：示例中的 /healthz 路径                                                   |
| 端口(Port)                     | 服务监听端口。如：示例中的 8080 端口                                                        |
| 协议                           | 访问协议，Http 或者Https                                                            |
| 延迟时间 ( initialDelaySeconds ) | 延迟检查时间，单位为秒，此设置与业务程序正常启动时间相关。例如，设置为 30，表明容器启动后 30 秒才开始健康检查，该时间是预留给业务程序启动的时间。 |
| 超时时间 ( timeoutSeconds )      | 超时时间，单位为秒。例如，设置为 10，表明执行健康检                                                  |

| 参数                          | 参数说明                                                                                                   |
|-----------------------------|--------------------------------------------------------------------------------------------------------|
| 超时时间 ( timeoutSeconds )     | 超时时间 , 单位为秒。例如 , 设置为 10 , 表明执行健康检查的超时等待时间为 10 秒 , 如果超过这个时间 , 本次健康检查就被视为失败。若设置为 0 或不设置 , 默认超时等待时间为 1 秒。 |
| 成功阈值 ( successThreshold )   | 探测失败后 , 被视为成功的最小连续成功数。默认值是 1 , 最小值是 1。存活和启动探测的这个值必须是 1。                                                |
| 最大失败次数 ( failureThreshold ) | 当探测失败时重试的次数。存活探测情况下的放弃就意味着重新启动容器。就绪探测情况下的放弃 Pod 会被打上未就绪的标签。默认值是 3。最小值是 1。                              |

## 使用 HTTP GET 请求检查

**YAML 示例 :**

```
apiVersion: v1
kind: Pod
metadata:
 labels:
 test: liveness
 name: liveness-http
spec:
 containers:
 - name: liveness
 image: k8s.gcr.io/liveness
 args:
 - /server
 livenessProbe:
```

```

httpGet:
 path: /healthz # 访问的请求路径
 port: 8080 # 服务监听端口
 httpHeaders:
 - name: Custom-Header
 value: Awesome
 initialDelaySeconds: 3 # kubelet 在执行第一次探测前应该等待 3 秒
 periodSeconds: 3 # kubelet 每隔 3 秒执行一次存活探测

```

按照设定的规则，Kubelet 向容器内运行的服务（服务在监听 8080 端口）发送一个 HTTP GET 请求来执行探测。如果服务器上 `/healthz` 路径下的处理程序返回成功代码，则 kubelet 认为容器是健康存活的。如果处理程序返回失败代码，则 kubelet 会杀死这个容器并将其重启。返回大于或等于 200 并且小于 400 的任何代码都标示成功，其它返回代码都标示失败。容器存活期间的最开始 10 秒中，`/healthz` 处理程序返回 200 的状态码。之后处理程序返回 500 的状态码。

## 使用 TCP 端口检查

**TCP 端口参数说明：**

| 参数                    | 参数说明                           |
|-----------------------|--------------------------------|
| 端口(Port)              | 服务监听端口。如：示例中的 8080 端口          |
| 延迟时间                  | 延迟检查时间，单位为秒，此设置与业务程序正常启动时      |
| (initialDelaySeconds) | 间相关。例如，设置为 30，表明容器启动后 30 秒才开始健 |
|                       | 康检查，该时间是预留给业务程序启动的时间。          |
| 超时时间 (timeoutSeconds) | 超时时间，单位为秒。例如，设置为 10，表明执行健康检    |
|                       | 查的超时等待时间为 10 秒，如果超过这个时间，本次健康   |
|                       | 检查就视为失败。若设置为 0 或不设置，默认超时等待     |
|                       | 时间为 1 秒。                       |

对于提供 TCP 通信服务的容器，基于此配置，按照设定规则集群对该容器建立 TCP 连接，如果连接成功，则证明探测成功，否则探测失败。选择 TCP 端口探测方式，必须指定容器监听的端口。

#### YAML 示例：

```
apiVersion: v1
kind: Pod
metadata:
 name: goproxy
 labels:
 app: goproxy
spec:
 containers:
 - name: goproxy
 image: k8s.gcr.io/goproxy:0.1
 ports:
 - containerPort: 8080
 readinessProbe:
 tcpSocket:
 port: 8080
 initialDelaySeconds: 5
 periodSeconds: 10
 livenessProbe:
 tcpSocket:
 port: 8080
 initialDelaySeconds: 15
 periodSeconds: 20
```

此示例同时使用就绪和存活探针。kubelet 在容器启动 5 秒后发送第一个就绪探测。尝试连接 **goproxy** 容器的 8080 端口，如果探测成功，这个 Pod 会被标记为就绪状态，kubelet 将继续每隔 10 秒运行一次检测。

除了就绪探测，这个配置包括了一个存活探测。 kubelet 会在容器启动 15 秒后进行第一次存活探测。 就绪探测会尝试连接 **goproxy** 容器的 8080 端口。 如果存活探测失败，容器会被重新启动。

## 执行命令检查

**YAML 示例:**

```
apiVersion: v1
kind: Pod
metadata:
 labels:
 test: liveness
 name: liveness-exec
spec:
 containers:
 - name: liveness
 image: k8s.gcr.io/busybox
 args:
 - /bin/sh
 - -c
 - touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600
 livenessProbe:
 exec:
 command:
 - cat
 - /tmp/healthy
 initialDelaySeconds: 5 # kubelet 在执行第一次探测前等待 5 秒
 periodSeconds: 5 #kubelet 每 5 秒执行一次存活探测
```

**periodSeconds** 字段指定了 kubelet 每 5 秒执行一次存活探测，**initialDelaySeconds** 字段指定 kubelet 在执行第一次探测前等待 5 秒。按照设定规则，集群周期性的通过 kubelet 在容器内执行命令 `cat /tmp/healthy` 来进行探测。如果命令执行成功并且返回值为 0，kubelet 就会认为这个容器是健康存活的。如果这个命令返回非 0 值，kubelet 会杀死这个容器并重新启动它。

## 使用启动前检查保护慢启动容器

有些应用在启动时需要较长的初始化时间，需要使用相同的命令来设置启动探测，针对 HTTP 或 TCP 检测，可以通过将 **failureThreshold \* periodSeconds** 参数设置为足够长的时

间来应对启动需要较长时间的场景。

#### YAML 示例：

```
ports:
- name: liveness-port
 containerPort: 8080
 hostPort: 8080
```

```
livenessProbe:
 httpGet:
 path: /healthz
 port: liveness-port
 failureThreshold: 1
 periodSeconds: 10
```

```
startupProbe:
 httpGet:
 path: /healthz
 port: liveness-port
 failureThreshold: 30
 periodSeconds: 10
```

如上设置，应用将有最多 5 分钟 ( $30 * 10 = 300s$ ) 的时间来完成启动过程，一旦启动探测成功，存活探测任务就会接管对容器的探测，对容器死锁作出快速响应。如果启动探测一直没有成功，容器会在 300 秒后被杀死，并且根据 `restartPolicy` 来执行进一步处置。

## 调度策略

在 Kubernetes 集群中，节点也有[标签](#)。您可以[手动添加标签](#)。Kubernetes 也会为集群中所有节点添加一些标准的标签。参见[常用的标签、注解和污点](#)以了解常见的节点标签。通过为节点添加标签，您可以让 Pod 调度到特定节点或节点组上。您可以使用这个功能来确保特定的 Pod 只能运行在具有一定隔离性，安全性或监管属性的节点上。

`nodeSelector` 是节点选择约束的最简单推荐形式。您可以将 `nodeSelector` 字段添加到 Pod 的规约中设置您希望目标节点所具有的[节点标签](#)。Kubernetes 只会将 Pod 调度到拥有指

定每个标签的节点上。**nodeSelector** 提供了一种最简单的方法来将 Pod 约束到具有特定标签的节点上。亲和性和反亲和性扩展了您可以定义的约束类型。使用亲和性与反亲和性的一些好处有：

- 亲和性、反亲和性语言的表达能力更强。**nodeSelector** 只能选择拥有所有指定标签的节点。亲和性、反亲和性为您提供对选择逻辑的更强控制能力。
- 您可以标明某规则是“软需求”或者“偏好”，这样调度器在无法找到匹配节点时，会忽略亲和性/反亲和性规则，确保 Pod 调度成功。
- 您可以使用节点上（或其他拓扑域中）运行的其他 Pod 的标签来实施调度约束，而不是只能使用节点本身的标签。这个能力让您能够定义规则允许哪些 Pod 可以被放置在一起。

您可以通过设置亲和（affinity）与反亲和（anti-affinity）来选择 Pod 要部署的节点。

## 容忍时间

当工作负载实例所在的节点不可用时，系统将实例重新调度到其它可用节点的时间窗。默认认为 300 秒。

## 节点亲和性（nodeAffinity）

节点亲和性概念上类似于 **nodeSelector**，它使您可以根据节点上的标签来约束 Pod 可以调度到哪些节点上。节点亲和性有两种：

- 必须满足：（**requiredDuringSchedulingIgnoredDuringExecution**）调度器只有在规则被满足的时候才能执行调度。此功能类似于 **nodeSelector**，但其语法表达能力更强。您可以定义多条硬约束规则，但只需满足其中一条。

- **尽量满足**：( `preferredDuringSchedulingIgnoredDuringExecution` ) 调度器会尝试寻找满足对应规则的节点。如果找不到匹配的节点，调度器仍然会调度该 Pod。您还可以为软约束规则设定权重，具体调度时，若存在多个符合条件的节点，权重最大的节点会被优先调度。同时您还可以定义多条硬约束规则，但只需满足其中一条。

## 标签名

对应节点的标签，可以使用默认的标签也可以用户自定义标签。

## 操作符

- In : 标签值需要在 `values` 的列表中
- NotIn : 标签的值不在某个列表中
- Exists : 判断某个标签是否存在，无需设置标签值
- DoesNotExist : 判断某个标签是不存在，无需设置标签值
- Gt : 标签的值大于某个值（字符串比较）
- Lt : 标签的值小于某个值（字符串比较）

## 权重

仅支持在“尽量满足”策略中添加，可以理解为调度的优先级，权重大大的会被优先调度。取值范围是 1 到 100。

## 工作负载亲和性

与节点亲和性类似，工作负载的亲和性也有两种类型：

- **必须满足**：( `requiredDuringSchedulingIgnoredDuringExecution` ) 调度器只有在规则被

满足的时候才能执行调度。此功能类似于 `nodeSelector`，但其语法表达能力更强。

您可以定义多条硬约束规则，但只需满足其中一条。

- **尽量满足**：( `preferredDuringSchedulingIgnoredDuringExecution` ) 调度器会尝试寻找

满足对应规则的节点。如果找不到匹配的节点，调度器仍然会调度该 Pod。您还可

为软约束规则设定权重，具体调度时，若存在多个符合条件的节点，权重最大的节

点会被优先调度。同时您还可以定义多条硬约束规则，但只需满足其中一条。

工作负载的亲和性主要用来决定工作负载的 Pod 可以和哪些 Pod 部署在同一拓扑域。例如，对于相互通信的服务，可通过应用亲和性调度，将其部署到同一拓扑域（如同一可用区）中，减少它们之间的网络延迟。

## 标签名

对应节点的标签，可以使用默认的标签也可以用户自定义标签。

## 命名空间

指定调度策略生效的命名空间。

## 操作符

- In：标签值需要在 values 的列表中
- NotIn：标签的值不在某个列表中
- Exists：判断某个标签是否存在，无需设置标签值
- DoesNotExist：判断某个标签是不存在，无需设置标签值

## 拓扑域

指定调度时的影响范围。例如，如果指定为 `kubernetes.io/Clustername` 表示以 Node 节点为区分范围。

## 工作负载反亲和性

与节点亲和性类似，工作负载的反亲和性也有两种类型：

- 必须满足：( `requiredDuringSchedulingIgnoredDuringExecution` ) 调度器只有在规则被满足的时候才能执行调度。此功能类似于 `nodeSelector`，但其语法表达能力更强。您可以定义多条硬约束规则，但只需满足其中一条。
- 尽量满足：( `preferredDuringSchedulingIgnoredDuringExecution` ) 调度器会尝试寻找满足对应规则的节点。如果找不到匹配的节点，调度器仍然会调度该 Pod。您还可为软约束规则设定权重，具体调度时，若存在多个符合条件的节点，权重最大的节点会被优先调度。同时您还可以定义多条硬约束规则，但只需满足其中一条。

工作负载的反亲和性主要用来决定工作负载的 Pod 不可以和哪些 Pod 部署在同一拓扑域。例如，将一个负载的相同 Pod 分散部署到不同的拓扑域（例如不同主机）中，提高负载本身的稳定性。

## 标签名

对应节点的标签，可以使用默认的标签也可以用户自定义标签。

## 命名空间

指定调度策略生效的命名空间。

## 操作符

- In：标签值需要在 values 的列表中
- NotIn：标签的值不在某个列表中
- Exists：判断某个标签是否存在，无需设置标签值
- DoesNotExist：判断某个标签是不存在，无需设置标签值

## 拓扑域

指定调度时的影响范围。例如，如果指定为 `kubernetes.io/Clustername` 表示以 Node 节点为区分范围。

# 安装 metrics-server 插件

`metrics-server` 是 Kubernetes 内置的资源使用指标采集组件。您可以通过配置弹性伸缩（HPA）策略来实现工作负载资源自动水平伸缩 Pod 副本。

本节介绍如何安装 `metrics-server`。

## 前提条件

安装 `metrics-server` 插件前，需要满足以下前提条件：

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 已完成一个[命名空间的创建](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。

## 操作步骤

请执行如下步骤为集群安装 **metrics-server** 插件。

- 在工作负载详情下的弹性伸缩页面，点击 **去安装**，进入 **metrics-server** 插件安装界面。

The screenshot shows the DaoCloud Enterprise 5.0 web interface. On the left, there's a sidebar with various navigation items like '集群概览', '节点管理', '工作负载' (selected), and '无状态负载'. The main content area is titled '集群: cluster202 / 命名空间: default / 无状态负载 / app2'. It displays basic information for the application 'app2' in namespace 'default': status '运行中' (Running), 1/1 instances, and creation time '2024-07-13 10:29'. Below this, there are tabs for '容器组', '容器配置', '访问方式', '调度策略', '标签与注解', '弹性伸缩' (selected), '版本记录', and '事件列表'. A note below the tabs says '指标伸缩 (HPA) 策略可以根据资源利用率指标 (CPU/内存) 或自定义指标自动扩展负载中 Pod 副本的数量。'. At the bottom, a warning message in a red-bordered box says 'metrics-server 插件未安装。请先安装后再继续!' followed by a blue '去安装' button.

### 工作负载

- 阅读 **metrics-server** 插件相关介绍，选择版本后点击 **安装** 按钮。本文将以 **3.8.2** 版本为例进行安装，推荐您安装 **3.8.2** 及更高版本。

**Helm 模板详情**

**metrics-server**

Metrics Server is a scalable, efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines.

**Kubernetes Metrics Server**

Metrics Server is a scalable, efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines.

**Installing the Chart**

Before you can install the chart you will need to add the `metrics-server` repo to Helm.

```
helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
```

After you've installed the repo you can install the chart.

```
helm upgrade --install metrics-server metrics-server/metrics-server
```

**Configuration**

The following table lists the configurable parameters of the Metrics Server chart and their default values.

| Parameter | Description | Default |
|-----------|-------------|---------|
|           |             |         |

## 工作负载

### 3. 在安装配置界面配置基本参数。

**安装 - metrics-server**

**metrics-server**

Metrics Server is a scalable, efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines.

**名称** metrics-server-01

**命名空间**  已有命名空间  新建命名空间  
default [创建命名空间](#)

**版本** 3.8.2

**参数配置**

**YAML**

```

1 image:
2 repository: k8s.gcr.io/metrics-server/metrics-server
3 tag: ''
4 pullPolicy: IfNotPresent
5 imagePullSecrets: []
6 nameOverride: ''
7 fullnameOverride: ''
8 serviceAccount:

```

## 工作负载

- **名称**：输入插件名称，请注意名称最长 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾，例如 **metrics-server-01**。
- **命名空间**：选择插件安装的命名空间，此处以 **default** 为例。

- 版本：插件的版本，此处以 **3.8.2** 版本为例。
- 就绪等待：启用后，将等待应用下所有关联资源处于就绪状态，**才会**标记应用安装成功。
- 失败删除：开启后，将默认同步开启就绪等待。如果安装失败，将删除安装相关资源。
- 详情日志：开启安装过程日志的详细输出。

**!!! note**

开启 **就绪等待** 和/或 **失败删除** 后，应用需要经过较长时间才会被标记为 **运行中** 状态。

#### 4. 高级参数配置

- 如果集群网络无法访问 `k8s.gcr.io` 仓库，请尝试修改 `repository` 参数为 `repository: k8s.m.daocloud.io/metrics-server/metrics-server`
- 安装 **metrics-server** 插件还需提供 SSL 证书。如需绕过证书校验，需要在 `defaultArgs:` 处添加 `--kubelet-insecure-tls` 参数。

**??? note “点击查看推荐的 YAML 参数”**

```
```yaml
image:
  repository: k8s.m.daocloud.io/metrics-server/metrics-server # 将仓库源地址修改为 k8s.m.daocloud.io
  tag: ""
  pullPolicy: IfNotPresent
imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""
serviceAccount:
  create: true
  annotations: {}
  name: ""
rbac:
  create: true
  pspEnabled: false
apiService:
  create: true
```

```

```
podLabels: {}
podAnnotations: {}
podSecurityContext: {}
securityContext:
 allowPrivilegeEscalation: false
 readOnlyRootFilesystem: true
 runAsNonRoot: true
 runAsUser: 1000
priorityClassName: system-cluster-critical
containerPort: 4443
hostNetwork:
 enabled: false
replicas: 1
updateStrategy: {}
podDisruptionBudget:
 enabled: false
 minAvailable: null
 maxUnavailable: null
defaultArgs:
 - '--cert-dir=/tmp'
 - '--kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname'
 - '--kubelet-use-node-status-port'
 - '--metric-resolution=15s'
 - --kubelet-insecure-tls # 绕过证书校验
args: []
livenessProbe:
 httpGet:
 path: /livez
 port: https
 scheme: HTTPS
initialDelaySeconds: 0
periodSeconds: 10
failureThreshold: 3
readinessProbe:
 httpGet:
 path: /readyz
 port: https
 scheme: HTTPS
initialDelaySeconds: 20
periodSeconds: 10
failureThreshold: 3
service:
 type: ClusterIP
 port: 443
```

```

 annotations: {}
 labels: {}
metrics:
 enabled: false
serviceMonitor:
 enabled: false
 additionalLabels: {}
 interval: 1m
 scrapeTimeout: 10s
resources: {}
extraVolumeMounts: []
extraVolumes: []
nodeSelector: {}
tolerations: []
affinity: {}
```

```

5. 点击 **确定** 按钮，完成 **metrics-server** 插件的安装，之后系统将自动跳转至 **Helm 应用列表** 页面，稍等几分钟后，为页面执行刷新操作，即可看到刚刚安装的应用。

!!! note

删除 **_metrics-server_** 插件时，在 **_Helm 应用_** 列表页面才能彻底删除该插件。如果仅在工作负载页面删除 **_metrics-server_**，这只是删除了该应用的工作负载副本，应用本身仍未删除，后续重新安装该插件时也会提示错误。

安装 **kubernetes-cronhpa-controller** 插件

定时伸缩 (CronHPA) 策略是指标伸缩 (HPA) 机制的扩展，允许根据时间模式指定负载中的 Pod 副本数量进行扩缩容。通过 CronHPA，用户可以在特定时间点自动扩展或缩减 Pod 数量。

CronHPA 能够为周期性高并发应用提供稳定的计算资源保障，**kubernetes-cronhpa-controller** 是实现 CronHPA 的关键组件。

本节介绍如何安装 **kubernetes-cronhpa-controller** 插件。

!!! note

为了使用 CronHPA，不仅需要安装 `kubernetes-cronhpa-controller` 插件，还要[安装 `metrics-server` 插件](install-metrics-server.md)。

前提条件

安装 `kubernetes-cronhpa-controller` 插件之前，需要满足以下前提条件：

- 在[容器管理](#)模块中[接入 Kubernetes 集群](#)或者[创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建一个[命名空间](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。

操作步骤

参考如下步骤为集群安装 `kubernetes-cronhpa-controller` 插件。

1. 在[集群列表](#) 页面找到需要安装此插件的目标集群，点击该集群的名称，然后在左侧点击[工作负载](#) -> [无状态工作负载](#)，点击目标工作负载的名称。

2. 在工作负载详情页面，点击[弹性伸缩](#) 页签，在[CronHPA](#) 右侧点击[安装](#)。

工作负载

工作负载

3. 阅读该插件的相关介绍，选择版本后点击[安装](#) 按钮。推荐安装 **1.3.0** 或更高版本。

工作负载

工作负载

4. 参考以下说明配置参数。

工作负载

工作负载

- 名称：输入插件名称，请注意名称最长 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾，例如 `kubernetes-cronhpa-controller`。
- 命名空间：选择将插件安装在哪个命名空间，此处以 `default` 为例
- 版本：插件的版本，此处以 `1.3.0` 版本为例
- 就绪等待：启用后，将等待应用下的所有关联资源都处于就绪状态，才会标记应用安装成功
- 失败删除：如果插件安装失败，则删除已经安装的关联资源。开启后，将默认同步开启 就绪等待
- 详情日志：开启后，将记录安装过程的详细日志

!!! note

开启 就绪等待 和/或 失败删除 后，应用需要较长时间才会被标记为“运行中”状态。

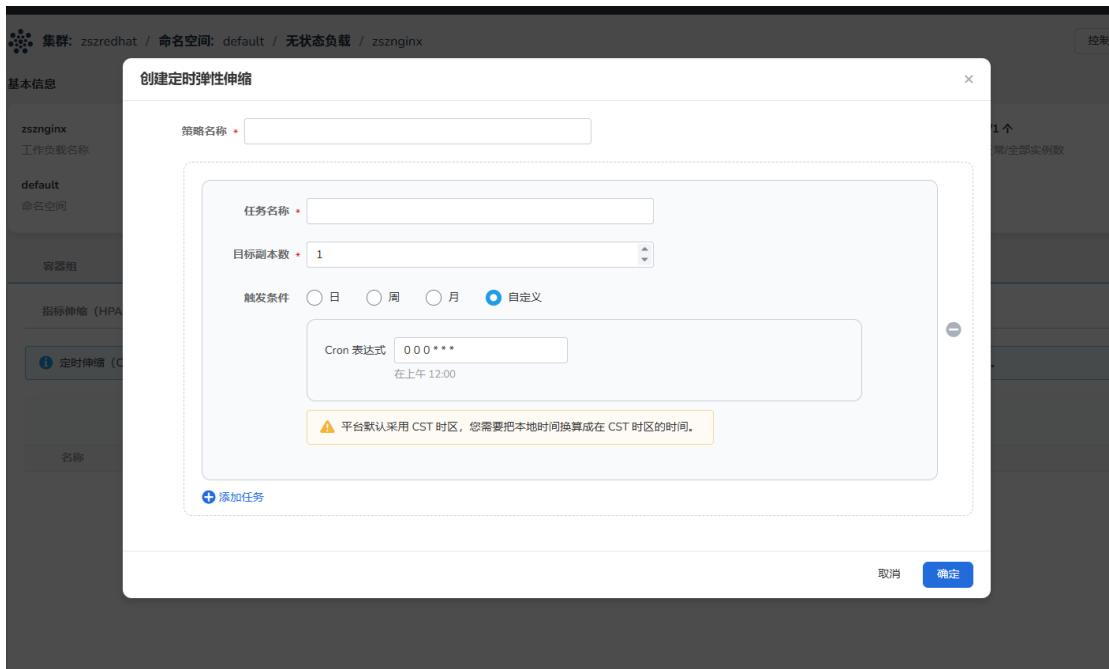
5. 在页面右下角点击 **确定**，系统将自动跳转至 **Helm 应用** 列表页面。稍等几分钟后刷新页面作，即可看到刚刚安装的应用。

!!! warning

如需删除 kubernetes-cronhpa-controller 插件，应在 Helm 应用 列表页面才能将其彻底删除。

如果在工作负载的 弹性伸缩 页签下删除插件，这只是删除了该插件的工作负载副本，插件本身仍未删除，后续重新安装该插件时也会提示错误。

6. 回到工作负载详情页面下的 **弹性伸缩** 页签，可以看到界面显示 **插件已安装**。现在可以开始创建 CronHPA 策略了。



工作负载

安装 vpa 插件

容器垂直扩缩容策略（Vertical Pod Autoscaler, VPA）能够让集群的资源配置更加合理，避免集群资源浪费。 **vpa** 则是实现容器垂直扩缩容的关键组件。

本节介绍如何安装 **vpa** 插件。

为了使用 VPA 策略，不仅需要安装 **_vpa_** 插件，还要[安装 **_metrics-server_** 插件](install-metrics-server.md)。

前提条件

安装 **vpa** 插件之前，需要满足以下前提条件：

- 在[容器管理](#)模块中[接入 Kubernetes 集群](#)或者[创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建一个[命名空间](#)。

- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。

操作步骤

参考如下步骤为集群安装 **vpa** 插件。

1. 在 **集群列表** 页面找到需要安装此插件的目标集群，点击该集群的名称，然后在左侧

点击 **工作负载 -> 无状态工作负载**，点击目标工作负载的名称。

2. 在工作负载详情页面，点击 **弹性伸缩** 页签，在 **VPA** 右侧点击 **安装**。

The screenshot shows the 'Vertical Pod Autoscaler (VPA)' configuration page for a Stateless workload named 'app1'. The 'Vertical Pod Autoscaler (VPA)' tab is selected. A red box highlights the 'Install' button at the bottom right of the main content area.

工作负载

3. 阅读该插件的相关介绍，选择版本后点击 **安装** 按钮。推荐安装 **1.5.0** 或更高版本。

VPA

A Helm chart for Kubernetes Vertical Pod Autoscaler

This chart is mostly based on the manifests and various scripts in the `deploy` and `hack` directories of the VPA repository.

Tests and Debugging

There are a few tests included with this chart that can help debug why your installation of VPA isn't working as expected. You can run `helm test -n <Release Namespace> <Release Name>` to run them.

- `crds-available` - Checks for both the `verticalpodautoscalers` and `verticalpodautoscalercheckpoints` CRDs
- `metrics-api-available` - Checks to make sure that the metrics API endpoint is available. If it's not, install `metrics-server` in your cluster.
- `create-vpa` - A simple check to make sure that VPA objects can be created in your cluster. Does not check for functionality of that VPA.
- `webhook-configuration` - Checks that both the service and the CA bundle in the `MutatingWebhookConfiguration` are configured correctly.

Components

There are three primary components to the Vertical Pod Autoscaler that can be enabled individually here.

- recommender
- updater
- admissionController

The admissionController is the only one that poses a stability consideration because it will create a `MutatingWebhookConfiguration` in your cluster. This could cause the cluster to stop accepting pod creation requests, if it is not configured correctly. Because of this, the `MutatingWebhookConfiguration` has its `failurePolicy` set to `Ignore` by default.

工作负载

4. 查看以下说明配置参数。

名称: vpa
命名空间: kube-system
版本: 4.5.0

参数配置

Vpa

AdmissionController

CertGen

Image

pullPolicy: Always
repository: ingress-nginx/kube-webhook-certgen

工作负载

- 名称：输入插件名称，请注意名称最长 63 个字符，只能包含小写字母、数

字及分隔符（“-”），且必须以小写字母或数字开头及结尾，例如

`kubernetes-cronhpa-controller`。

- 命名空间：选择将插件安装在哪个命名空间，此处以 `default` 为例。

- 版本：插件的版本，此处以 **4.5.0** 版本为例。
- 就绪等待：启用后，将等待应用下的所有关联资源都处于就绪状态，**才会标记**应用安装成功。
- 失败删除：如果插件安装失败，则删除已经安装的关联资源。**开启后，将默认**同步开启 **就绪等待**。
- 详情日志：**开启后，将记录安装过程的详细日志。**

!!! note

开启 **就绪等待** 和/或 **失败删除** 后，应用需要经过较长时间才会被标记为“运行中”状态。

5. 在页面右下角点击 **确定**，系统将自动跳转至 **Helm 应用** **列表**页面。稍等几分钟后刷新页面作，即可看到刚刚安装的应用。

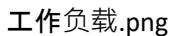
!!! warning

如需删除 **vpa** 插件，应在 **Helm 应用** **列表**页面才能将其彻底删除。

如果在工作负载的 **弹性伸缩** 页签下删除插件，这只是删除了该插件的工作负载副本，插件本身仍未删除，后续重新安装该插件时也会提示错误。

6. 回到工作负载详情页面下的 **弹性伸缩** 页签，可以看到界面显示 **插件已安装**。现在可以开始创建 **VPA** 策略了。

The screenshot shows the 'Basic Information' section of the 'Workload Details' page. On the left, there's a sidebar with 'cluster202' selected. Under 'Workload', 'Vertical Pod Autoscaler' is highlighted. The main area shows two workloads: 'app1' and 'default'. For 'app1', the status is 'Running' with 1/1 pods. For 'default', it's 'Rolling Upgrade' with a creation time of '2024-07-05 14:16'. Below this, there are tabs for 'Horizontal Pod Autoscaler (HPA)', 'CronHPA', and 'Vertical Pod Autoscaler (VPA)'. The 'Vertical Pod Autoscaler (VPA)' tab is active. A tooltip explains that VPA strategies can automatically adjust resources based on usage. At the bottom right, a red box surrounds the 'Create VPA' button.



基于内置指标创建 HPA

DaoCloud Enterprise 5.0 支持 Pod 资源基于指标进行弹性伸缩 (Horizontal Pod Autoscaling, HPA)。用户可以通过设置 CPU 利用率、内存用量及自定义指标指标来动态调整 Pod 资源的副本数量。例如，为工作负载设置基于 CPU 利用率指标弹性伸缩策略后，当 Pod 的 CPU 利用率超过/低于您设置的指标阈值，工作负载控制器将会自动增加/较少 Pod 副本数。

本文将介绍如何为工作负载配置基于内置指标的弹性伸缩。

!!! note

1. HPA 仅适用于 Deployment 和 StatefulSet，每个工作负载只能创建一个 HPA。
2. 如果基于 CPU 利用率创建 HPA 策略，必须预先为工作负载设置配置限制（Limit），否则无法计算 CPU 利用率。
3. 如果同时使用内置指标和多种自定义指，HPA 会根据多项指标分别计算所需伸缩副本数，取较大值（但不会超过设置 HPA 策略时配置的最大副本数）进行弹性伸缩。

内置指标弹性伸缩策略

系统内置了 CPU 和内存两种弹性伸缩指标以满足用户的基础业务使用场景。

前提条件

在为工作负载配置内置指标弹性伸缩策略之前，需要满足以下前提条件：

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 已完成一个[命名空间的创建](#)、[无状态工作负载的创建](#)或[有状态工作负载的创建](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。

- 已完成 [metrics-server 插件安装](#)。

操作步骤

参考以下步骤，为工作负载配置内置指标弹性伸缩策略。

1. 点击左侧导航栏上的 **集群列表** 进入集群列表页面。点击一个集群名称，进入 **集群详情** 页面。

[集群详情](#)

[集群详情](#)

2. 在集群详情页面，点击左侧导航栏的 **工作负载** 进入工作负载列表后，点击一个负载名称，进入 **工作负载详情** 页面。

[工作负载](#)

[工作负载](#)

3. 点击 **弹性伸缩** 页签，查看当前集群的弹性伸缩配置情况。

[弹性伸缩](#)

[弹性伸缩](#)

4. 确认集群已[安装了 metrics-server 插件](#)，且插件运行状态为正常后，即可点击 **新建伸缩** 按钮。

[新建伸缩](#)

[新建伸缩](#)

5. 创建自定义指标弹性伸缩策略参数。

[工作负载](#)

[工作负载](#)

- 策略名称：输入弹性伸缩策略的名称，请注意名称最长 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾，例如 hpa-my-dep。
- 命名空间：负载所在的命名空间。
- 工作负载：执行弹性伸缩的工作负载对象。
- 副本范围：设置允许的最小容器组副本数量，默认值为 1。设置允许的最大容器组副本数量，默认值为 10。
- 稳定时间窗口：扩缩容的稳定窗口时间需要大于等于 0，小于等于 3600，取值范围为 [0,3600] 秒
- 系统指标：
 - CPU 利用率：工作负载资源下 Pod 的 CPU 使用率。计算方式为：工作负载下所有的 Pod 资源 / 工作负载的请求 (request) 值。当实际 CPU 用量大于/小于目标值时，系统自动减少/增加 Pod 副本数量。
 - 内存用量：工作负载资源下的 Pod 的内存用量。当实际内存用量大于/小于目标值时，系统自动减少/增加 Pod 副本数量。
- 自定义指标：参见[基于自定义指标创建 HPA](#)

6. 完成参数配置后，点击 **确定** 按钮，自动返回弹性伸缩详情页面。点击列表右侧的 ，可以执行编辑、删除操作，还可以查看相关事件。

工作负载

工作负载

基于自定义指标创建 HPA

当系统内置的 CPU 和内存两种指标不能满足您业务的实际需求时，您可以通过配置 ServiceMonitoring 来添加自定义指标，并基于自定义指标实现弹性伸缩。本文将介绍如何为工作负载配置基于自定义指标进行弹性伸缩。

!!! note

1. HPA 仅适用于 Deployment 和 StatefulSet，每个工作负载只能创建一个 HPA。
2. 如果同时使用内置指标和多种自定义指，HPA 会根据多项指标分别计算所需伸缩副本数，取较大值（但不会超过设置 HPA 策略时配置的最大副本数）进行弹性伸缩。

前提条件

在为工作负载配置自定义指标弹性伸缩策略之前，需要满足以下前提条件：

- 已接入 Kubernetes 集群或者已创建 Kubernetes 集群，且能够访问集群的 UI 界面
- 已完成一个命名空间的创建、无状态工作负载的创建或有状态工作负载的创建
- 当前操作用户应具有 NS Editor 或更高权限，详情可参考[命名空间授权](#)
- 已安装 metrics-server 插件
- 已安装 insight-agent 插件
- 已安装 Prometheus-adapter 插件

操作步骤

参考以下步骤，为工作负载配置指标弹性伸缩策略。

1. 点击左侧导航栏上的 **集群列表** 进入集群列表页面。点击一个集群名称，进入 **集群详情** 页面。

集群详情

集群详情

- 在集群详情页面，点击左侧导航栏的 **工作负载** 进入工作负载列表后，点击一个负载名称，进入 **工作负载详情** 页面。

工作负载

工作负载

- 点击 **弹性伸缩** 页签，查看当前集群的弹性伸缩配置情况。

弹性伸缩配置

弹性伸缩配置

- 确认集群已安装了 metrics-server、Insight、Prometheus-adapter 插件且插件运行状态为正常后，即可点击 **新建伸缩** 按钮。

!!! note

如果相关插件未安装或插件处于异常状态，您在页面上将无法看见创建自定义指标弹性伸缩入口。

新建伸缩

新建伸缩

- 创建自定义指标弹性伸缩策略参数。

伸缩策略参数

伸缩策略参数

- 策略名称**：输入弹性伸缩策略的名称，请注意名称最长 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾，例如 hpa-my-dep。
- 命名空间**：负载所在的命名空间。
- 工作负载**：执行弹性伸缩的工作负载对象。
- 资源类型**：进行监控的自定义指标类型，包含 Pod 和 Service 两种类型。

- 指标：使用 ServiceMonitoring 创建的自定义指标名称或系统内置的自定义指标名称。
- 数据类型：用于计算指标值的方法，包含目标值和目标平均值两种类型，当资源类型为 Pod 时，只支持使用目标平均值。

操作示例

本案例以 Golang 业务程序为例，该示例程序暴露了 httpserver_requests_total 指标，并记录 HTTP 的请求，通过该指标可以计算出业务程序的 QPS 值。

部署业务程序

使用 Deployment 部署业务程序：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpserver
  namespace: httpserver
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpserver
  template:
    metadata:
      labels:
        app: httpserver
    spec:
      containers:
        - name: httpserver
          image: registry.imroc.cc/test/httpserver:custom-metrics
          imagePullPolicy: Always

```

```
--
```

```
apiVersion: v1
kind: Service
```

```

metadata:
  name: httpserver
  namespace: httpserver
  labels:
    app: httpserver
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: "/metrics"
    prometheus.io/port: "http"
spec:
  type: ClusterIP
  ports:
    - port: 80
      protocol: TCP
      name: http
  selector:
    app: httpserver

```

Prometheus 采集业务监控

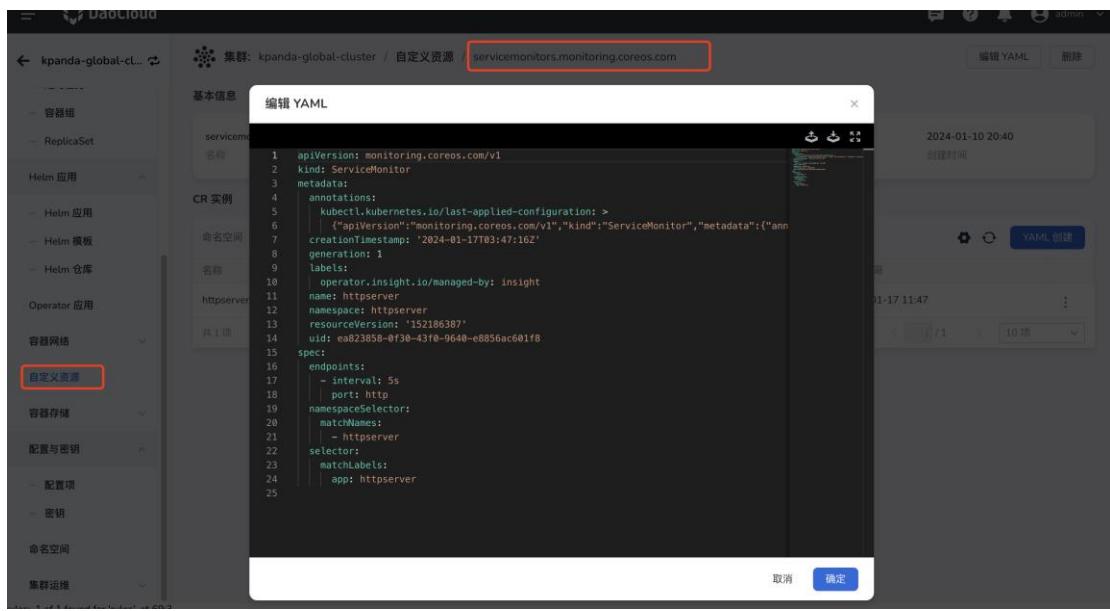
若已安装 insight-agent，可以通过创建 ServiceMonitor 的 CRD 对象配置 Prometheus。

操作步骤：在 集群详情 -> 自定义资源 搜索“servicemonitors.monitoring.coreos.com”，点击名称进入详情。通过创建 YAML，在命名空间 **httpserver** 下创建如下示例的 CRD：

```

apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: httpserver
  namespace: httpserver
  labels:
    operator.insight.io/managed-by: insight
spec:
  endpoints:
    - port: http
      interval: 5s
  namespaceSelector:
    matchNames:
      - httpserver
  selector:
    matchLabels:
      app: httpserver

```



servicemonitor

!!! note

若通过 insight 安装 Prometheus，则 serviceMonitor 上必须打上 `operator.insight.io/managed-by : insight`

这个 label，通过其他方式安装则无需此 label。

在 prometheus-adapter 中配置指标规则

操作步骤：在 集群详情 -> Helm 应用 搜索 “prometheus-adapter”，通过操作栏进入更新

页面，在 YAML 中配置自定义指标，示例如下：

rules:

 custom:

- metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}[1m])) by (<<.GroupBy>>)

 name:

 as: httpserver_requests_qps

 matches: httpserver_requests_total

 resources:

 template: <<.Resource>>

 seriesQuery: httpserver_requests_total

```

68     memory: 320Mi
69   rules:
70     custom:
71       - metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}{1m})) by (<<.GroupBy>>)
72       name:
73         as: httpserver_requests_qps
74         matches: httpserver_requests_total
75       resources:
76         template: <<.Resource>>
77           default: true
78           existing: null
79           external: []
80       securityContext:
81         allowPrivilegeEscalation: false
82       capabilities:
83

```

取消 确定

rules

创建自定义指标弹性伸缩策略参数

按照上述步骤在 Deployment 中找到应用程序 httpserver 并通过自定义指标创建弹性伸缩。

集群概览 节点管理 工作负载 无状态负载 有状态负载 守护进程 任务 定时任务 容器组 ReplicaSet Helm 应用 Helm 应用 Helm 模板 Helm 仓库 Operator 应用 容器网络

集群: kpanda-global-cluster / 命名空间: httpserver / 无状态负载 httpserver

创建弹性伸缩

如果弹性伸缩策略指定了多个指标,那么会按照每个指标分别计算扩缩副本数,取最大值进行扩缩。

策略名称: hpa-httpserver
命名空间: httpserver
工作负载: httpserver
副本范围: 1 - 10
稳定窗口时间: 缩容 300 秒 扩容 300 秒
系统指标: + 添加系统指标 (httpserver.request.qps)
自定义指标: + 添加自定义指标 (httpserver.request.qps)

取消 确定

custommetrics

创建 VPA

垂直伸缩（VPA）策略可以根据容器实际资源使用情况手动或自动调整 Pod 中容器的 CPU 和内存资源请求量。这有助于确保每个容器都能获得所需的资源，而不会过度提供资源浪费。

容器垂直扩缩容策略（Vertical Pod Autoscaler, VPA）通过监控 Pod 在一段时间内的资源申请和用量，计算出对该 Pod 而言最适合的 CPU 和内存请求值。使用 VPA 可以更加合理地为集群下每个 Pod 分配资源，提高集群的整体资源利用率，避免集群资源浪费。

DCE 5.0 支持通过容器垂直扩缩容策略（Vertical Pod Autoscaler, VPA），基于此功能可以根据容器资源的使用情况动态调整 Pod 请求值。DCE 5.0 支持通过手动和自动两种方式来修改资源请求值，您可以根据实际需要进行配置。

本文将介绍如何为工作负载配置 Pod 垂直伸缩。

!!! warning

使用 VPA 修改 Pod 资源请求会触发 Pod 重启。由于 Kubernetes 本身的限制，Pod 重启后可能会被调度到其它节点上。

前提条件

为工作负载配置垂直伸缩策略之前，需要满足以下前提条件：

- 在[容器管理](#)模块中[接入 Kubernetes 集群](#)或者[创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建一个[命名空间](#)、[用户](#)、[无状态工作负载](#)或[有状态工作负载](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。
- 当前集群已经安装 [metrics-server](#) 和 [VPA](#) 插件。

操作步骤

参考以下步骤，为工作负载配置内置指标弹性伸缩策略。

- 在 **集群列表** 中找到目前集群，点击目标集群的名称。

集群详情

- 在左侧导航栏点击 **工作负载**，找到需要创建 VPA 的负载，点击该负载的名称。

工作负载

- 点击 **弹性伸缩** 页签，查看当前集群的弹性伸缩配置，确认已经安装了相关插件并且插件是否运行正常。

The screenshot shows the 'Vertical Scaling (VPA)' tab selected in the 'Horizontal Pod Auto-Scale (HPA) and CronHPA Scheduling Rules' section. It displays a table with two rows: 'app1' and 'default'. The 'app1' row shows a status of 'Running' with 1/1 instances. The 'default' row shows a status of 'Rolling Upgrade' with 1 instance created on 2024-07-05 14:16. Below the table are tabs for 'Horizontal Scaling (HPA)', 'CronHPA', and 'Vertical Scaling (VPA)', with 'Vertical Scaling (VPA)' being the active tab. A note at the bottom states: 'Vertical scaling (VPA) policies can adjust container resource usage based on actual usage, either manually or automatically, to ensure each container gets the resources it needs without wasting resources.' A 'Create New Scaling Rule' button is visible.

垂直伸缩

4. 点击 **新建伸缩** 按钮，并配置 VPA 垂直伸缩策略参数。

The screenshot shows the 'Create Vertical Scaling (VPA)' dialog box. It has fields for '策略名称' (Policy Name), '伸缩模式' (Scaling Mode) set to '手动伸缩' (Manual Scaling), and '目标容器' (Target Container). There are '取消' (Cancel) and '确定' (Confirm) buttons at the bottom. A note at the bottom of the dialog box states: 'Vertical scaling (VPA) policies can adjust container resource usage based on actual usage, either manually or automatically, to ensure each container gets the resources it needs without wasting resources.'

新建伸缩

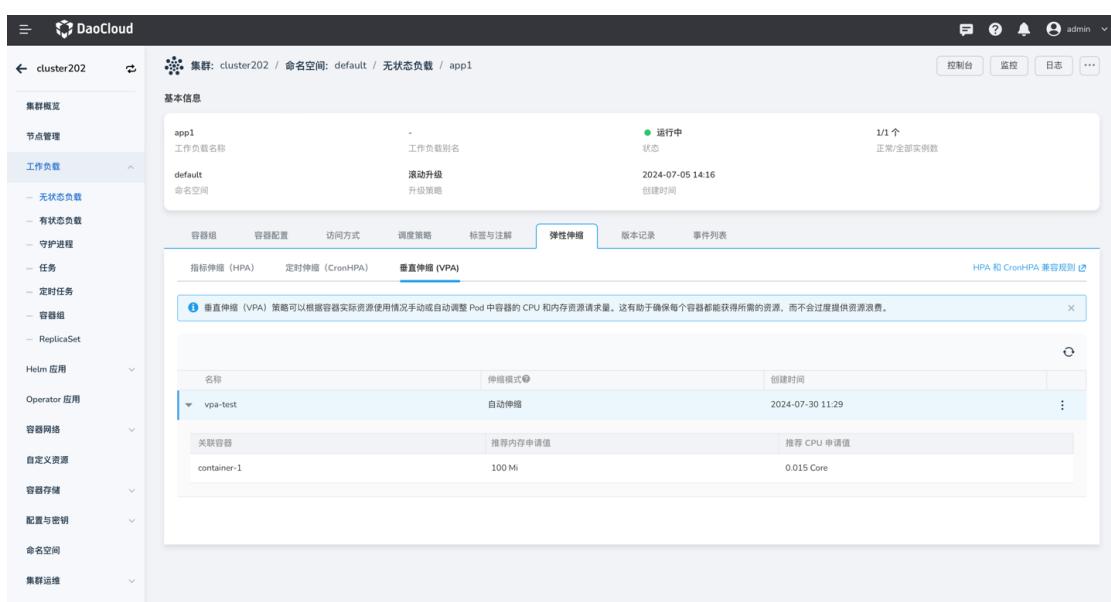
- **策略名称**：输入垂直伸缩策略的名称，请注意名称最长 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾，例如 vpa-my-dep。

- **伸缩模式**：执行修改 CPU 和内存请求值的方式，目前垂直伸缩支持手动和

自动两种伸缩模式。

- 手动伸缩：垂直伸缩策略计算出推荐的资源配置值后，需用户手动修改应用的资源配置。
 - 自动伸缩：垂直伸缩策略自动计算和修改应用的资源配置。
- 目标容器：选择需要进行垂直伸缩的容器。

5. 完成参数配置后，点击 **确定** 按钮，自动返回弹性伸缩详情页面。点击列表右侧的 ，可以执行编辑、删除操作。



名称	伸缩模式	创建时间
vpa-test	自动伸缩	2024-07-30 11:29

垂直伸缩 (VPA) 策略可以根据容器实际资源使用情况手动或自动调整 Pod 中容器的 CPU 和内存资源请求量。这有助于确保每个容器都能获得所需的资源，而不会过度提供资源浪费。

工作负载

!!! note

默认情况下，`--min-replicas` 的值为 2。表示当副本数大于 1 时，VPA 才会生效，可以通过修改 `updater` 的 `--min-replicas` 参数值来改变这一默认行为。

```
```yaml
spec:
 containers:
 - name: updater
 args:
 - "--min-replicas=2"
```

```

HPA 和 CronHPA 兼容规则

HPA 全称为 HorizontalPodAutoscaler，即 Pod 水平自动伸缩。HPA 策略可以根据资源利用率指标（CPU/内存）或自定义指标自动扩展负载中 Pod 副本的数量。

CronHPA 全称为 Cron HorizontalPodAutoscaler，即 Pod 定时的水平自动伸缩。CronHPA 策略是指标伸缩（HPA）机制的扩展，允许根据时间模式指定负载中的 Pod 副本数量进行扩缩容。通过 CronHPA，用户可以在特定时间点自动扩展或缩减 Pod 数量。

CronHPA 和 HPA 兼容冲突

定时伸缩 CronHPA 通过设置定时的方式触发容器的水平副本伸缩。为了防止突发的流量冲击等状况，您可能已经配置 HPA 保障应用的正常运行。如果同时检测到了 HPA 和 CronHPA 的存在，由于 CronHPA 和 HPA 相互独立无法感知，就会出现两个控制器各自工作，后执行的操作会覆盖先执行的操作。

对比 CronHPA 和 HPA 的定义模板，可以观察到以下几点：

- CronHPA 和 HPA 都是通过 scaleTargetRef 字段来获取伸缩对象。
- CronHPA 通过 Job 的 crontab 规则定时伸缩副本数。
- HPA 通过资源利用率判断伸缩情况。

!!! note

如果同时设置 CronHPA 和 HPA，会出现 CronHPA 和 HPA 同时操作一个 scaleTargetRef 的场景。

CronHPA 和 HPA 兼容方案

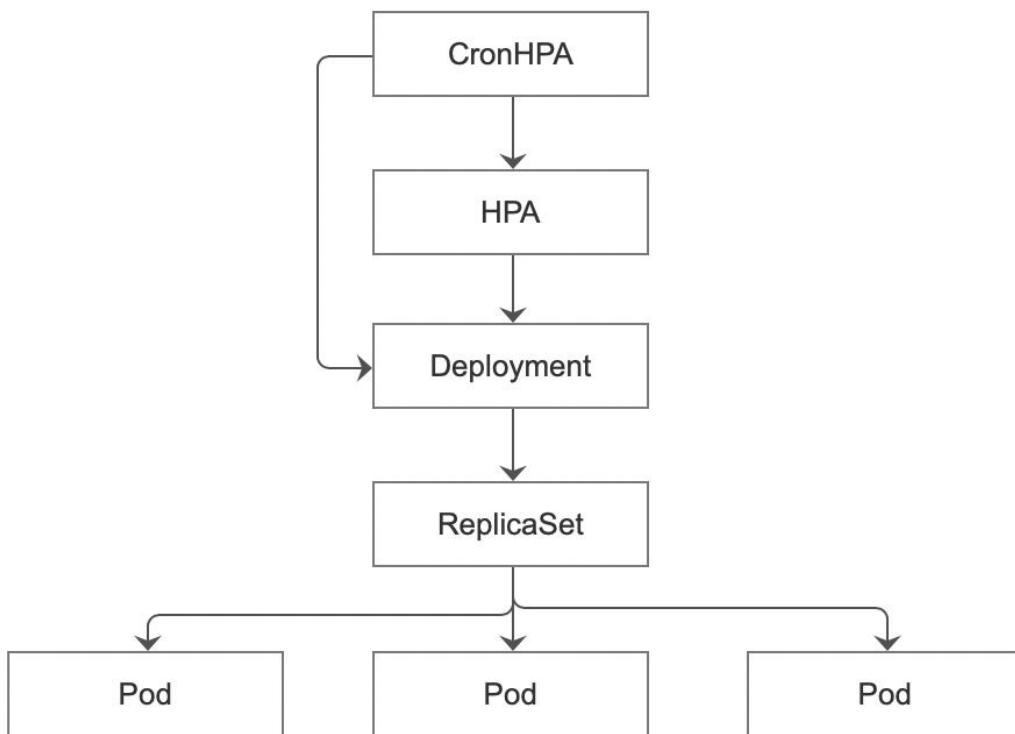
从上文可知，CronHPA 和 HPA 同时使用会导致后执行的操作覆盖先执行操作的本质原因是两个控制器无法相互感知，那么只需要让 CronHPA 感知 HPA 的当前状态就能解决冲

突问题。

系统会将 HPA 作为定时伸缩 CronHPA 的扩缩容对象，从而实现对该 HPA 定义的 Deployment 对象的定时扩缩容。

HPA 的定义将 Deployment 配置在 scaleTargetRef 字段下，然后 Deployment 通过自身定义查找 ReplicaSet，最后通过 ReplicaSet 调整真实的副本数目。

DCE 5.0 将 CronHPA 中的 scaleTargetRef 设置为 HPA 对象，然后通过 HPA 对象来寻找真实的 scaleTargetRef，从而让 CronHPA 感知 HPA 的当前状态。



CronHPA 和 HPA 兼容方案

CronHPA 会通过调整 HPA 的方式感知 HPA。CronHPA 通过识别要达到的副本数与当前副本数两者间的较大值，判断是否需要扩缩容及修改 HPA 的上限；CronHPA 通过识别 CronHPA 要达到的副本数与 HPA 的配置间的较小值，判断是否需要修改 HPA 的下限。

Knative 介绍

Knative 提供了一种更高层次的抽象，简化并加速了在 Kubernetes 上构建、部署和管理应用的过程。它使得开发人员能够更专注于业务逻辑的实现，而将大部分基础设施和运维工作交给 Knative 去处理，从而显著提高生产力。

组件

knative-operator 运行组件如下。

| | | | | | |
|------------------|-----------------------------------|-----|---------|---|-------|
| knative-operator | knative-operator-58f7d7db5c-7f6r5 | 1/1 | Running | 0 | 6m55s |
| knative-operator | operator-webhook-667dc67bc-qrvv4 | 1/1 | Running | 0 | 6m55s |

knative-serving 组件如下。

| | | | | | |
|-----------------|--|-----|-----------|---|--|
| knative-serving | 3scale-kourier-gateway-d69fbfbdbd8d8 | 1/1 | Running | 0 | |
| | 7m13s | | | | |
| knative-serving | activator-7c6fddd698-wdlng | 1/1 | Running | 0 | |
| | 7m3s | | | | |
| knative-serving | autoscaler-8f4b876bb-kd25p | 1/1 | Running | 0 | |
| | 7m17s | | | | |
| knative-serving | autoscaler-hpa-5f7f74679c-vkc7p | 1/1 | Running | 0 | |
| | 7m15s | | | | |
| knative-serving | controller-789c896c46-tfvsv | 1/1 | Running | 0 | |
| | 7m17s | | | | |
| knative-serving | net-kourier-controller-7db578c889-7gd5l | 1/1 | Running | 0 | |
| | 7m14s | | | | |
| knative-serving | webhook-5c88b94c5-78x7m | 1/1 | Running | 0 | |
| | 7m1s | | | | |
| knative-serving | storage-version-migration-serving-1.12.2-t7zvd | 0/1 | Completed | | |
| 0 | 7m15s | | | | |

| 组件 | 作用 |
|-----------|---|
| Activator | 对请求排队（如果一个 Knative Service 已经缩减到零）。调用 autoscaler，将缩减到 0 的服务恢复并转发排队的请求。Activator 还可以充当请求 |

| 组件 | 作用 |
|-------------|---|
| Autoscaler | 缓冲器，处理突发流量。
Autoscaler 负责根据配置、指标和进入的请求来缩放 Knative 服务。 |
| Controller | 管理 Knative CR 的状态。它会监视多个对象，管理依赖资源的生命周期，并更新资源状态。 |
| Queue-Proxy | Sidecar 容器，每个 Knative Service 都会注入一个。负责收集流量数据并报告给 Autoscaler，Autoscaler 根据这些数据和预设的规则来发起扩容或缩容请求。 |
| Webhooks | Knative Serving 有几个 Webhooks 负责验证和变更 Knative 资源。 |

Ingress 流量入口方案

| 方案 | 适用场景 |
|---------|---|
| Istio | 如果已经用了 Istio，可以选择 Istio 作为流量入口方案。 |
| Contour | 如果集群中已经启用了 Contour，可以选择 Contour 作为流量入口方案。 |
| Kourier | 如果在没有上述 2 种 Ingress 组件时，可以使用 Knative 基于 Envoy 实现的 Kourier Ingress 作为流量入口。 |

Autoscaler 方案对比

| Autoscaler
类型 | 是否为 Knative Serving 核心部分 | 默认启用 | Scale to Zero 支持 | 基于 CPU 的
Autoscaling
支持 |
|--|--------------------------|-----------------------------|------------------|-------------------------------|
| Knative Pod
Autoscaler
(KPA) | 是 | 是 | 是 | 否 |
| Horizontal
Pod
Autoscaler
(HPA) | 否 | 需安装
Knative
Serving 后 | 否 | 是 |
| | | 启用 | | |

CRD

| 资源类型 | API 名称 | 描述 |
|----------|-----------------------------|--------------------------------------|
| Services | service.serving.knative.dev | 自动管理 Workload 的整个生命周期，控制其他对象的创建，确保应用 |

| 资源类型 | API 名称 | 描述 |
|----------------|-----------------------------------|----------------|
| | | 具有 |
| | | Routes |
| | | 、 |
| | | Configurations |
| | | 以及每 |
| | | 次更新 |
| | | 时的新 |
| | | revision |
| | | 。 |
| Routes | route.serving.knative.dev | 将网络 |
| | | 端点映 |
| | | 射到一 |
| | | 个或多 |
| | | 个修订 |
| | | 版本， |
| | | 支持流 |
| | | 量分配 |
| | | 和版本 |
| | | 路由。 |
| Configurations | configuration.serving.knative.dev | 维护部 |
| | | 署的期 |
| | | 望状 |

| 资源类型 | API 名称 | 描述 |
|-----------|------------------------------|--|
| | | 态， 提供代码和配置之间的分离，遵循 Twelve-Factor 应用程序方法论，修改配置会创建新的 revision。 |
| Revisions | revision.serving.knative.dev | 每次对工作负载修改的时间点快照，是 |

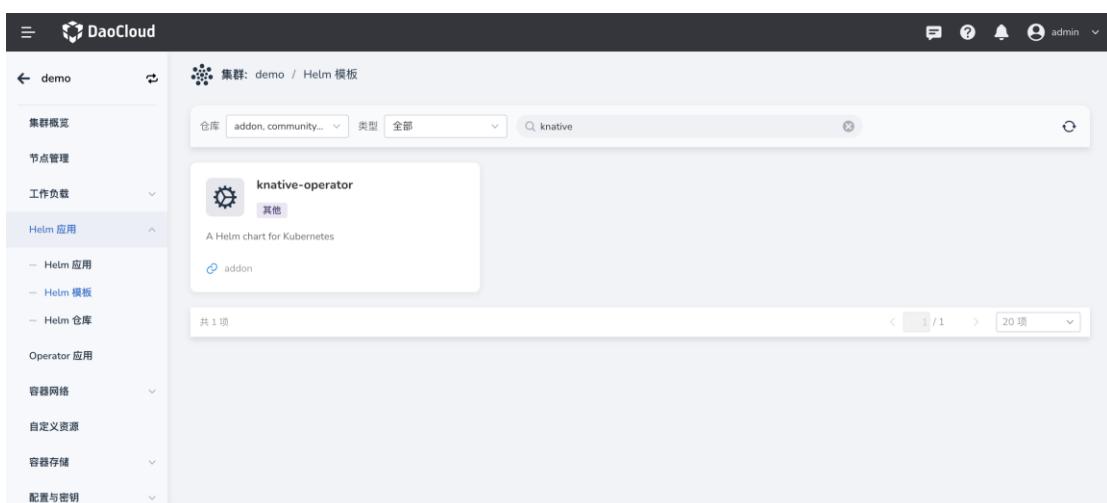
| 资源类型 | API 名称 | 描述 |
|------|--------|-----|
| | | 不可变 |
| | | 对象， |
| | | 可根据 |
| | | 流量自 |
| | | 动扩容 |
| | | 和缩 |
| | | 容。 |

安装

Knative 是一个面向无服务器部署的跨平台解决方案。

步骤

1. 登录集群，点击侧边栏 **Helm 应用** -> **Helm 模板**，在右侧上方搜索框输入 **knative**，然后按回车键搜索。



Install-1

2. 点击搜索出的 **knative-operator**，进入安装配置界面。你可以在该界面查看可用版本以及 Helm values 的 Parameters 可选项。

The screenshot shows the Helm chart details for the **knative-operator**. The sidebar on the left is titled "Helm 应用" and includes sections for Helm 应用, Helm 模板, Helm 仓库, Operator 应用, 容器网络, 自定义资源, 容器存储, and 配置与密钥. The main content area has tabs for Overview, Install, and Parameters. The Install tab contains a command block:

```
helm repo add daocloud-addon https://release.daocloud.io/chartrepo/addon
helm install knative-operator -f values.yaml --wait --debug daocloud-addon/knative-operator
helm ls
```

The Parameters tab lists "operator parameters". On the right side, there are filters for 模板 (Templates) and 关键词 (Keywords), with "knative" and "knative-operator" selected.

Install-2

3. 点击安装按钮后，进入安装配置界面。

The screenshot shows the Helm installation configuration for the **knative-operator**. The sidebar on the left is identical to the previous screen. The main configuration area includes fields for Name (knative-operator), Namespace (knative-operator), Version (0.1.0), and several toggle switches: 失败删除 (Delete on Fail) set to 关闭 (Off), 就绪等待 (Ready Wait) set to 启用 (On), and 详细日志 (Detailed Log) set to 启用 (On). Below this, there's a "参数配置" (Parameter Configuration) section with tabs for 表单 (Form), YAML, and 变化 (Changes). The "Serving" section is expanded, showing "Enable" set to "On" and "Namespace" set to "knative-serving". The "High Availability" section shows "Replicas" set to "2". The "Magic DNS" section shows "Enable" set to "On". At the bottom right are "取消" (Cancel) and "确定" (Confirm) buttons.

Install-3

4. 输入名称，安装租户，建议勾选 **就绪等待** 和 **详细日志**。

5. 在下方设置，可以勾选 **Serving**，并输入 Knative Serving 组件的安装租户，会在安装后部署 Knative Serving 组件，该组件由 Knative Operator 管理。

使用场景

适合的场景

- 短连接高并发业务
- 需要弹性伸缩的业务
- 大量应用需要缩容到 0 提高资源利用率
- AI Serving 服务，基于特定指标进行扩容

!!! tip

短连接高并发业务以及需要弹性伸缩的业务，推荐使用 HPA 和 VPA 能力。

不适合的场景

- 长连接业务
- 延时敏感业务
- 基于 cookie 的流量分流
- 基于 header 的流量分流

Knative 使用实践

在本节中，我们将通过几个实践来深入了解学习 Knative。

case 1 - Hello World

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello
spec:
```

```

template:
spec:
  containers:
    - image: m.daocloud.io/ghcr.io/knative/helloworld-go:latest
      ports:
        - containerPort: 8080
      env:
        - name: TARGET
          value: "World"

```

可以使用 kubectl 已部署的应用的状态，这个应用由 knative 自动配置了 ingress 和伸缩器。

```

~ kubectl get service.serving.knative.dev/hello
NAME      URL                                     LATESTCREATED   LA
TESTREADY READY   REASON
hello     http://hello.knative-serving.knative.loulan.me   hello-00001   hello-00001   True

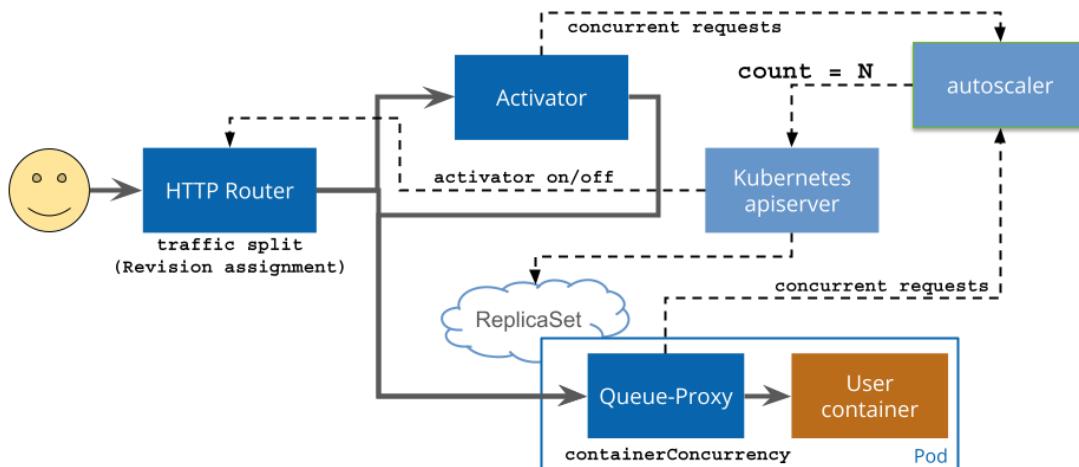
```

部署出的 Pod YAML 如下，由 2 个 Pod 组成：user-container 和 queue-proxy。

```

apiVersion: v1
kind: Pod
metadata:
  name: hello-00003-deployment-5fcb8ccbf-7qjfk
spec:
  containers:
    - name: user-container
    - name: queue-proxy

```



knative-request-flow

请求流：

1. case1 在低流量或零流量时，流量将路由到 activator

2.case2 流量大时，流量大于 target-burst-capacity 时才直接路由到 Pod

1. 配置为 0，只有从 0 扩容存在
2. 配置为 -1，activator 会一直存在请求路径
3. 配置为 >0，触发扩缩容之前，系统能够额外处理的并发请求数量。

3.case3 流量再变小时，流量低于 current_demand + target-burst-capacity > (pods * concurrency-target) 时将再次路由到 activator

待处理的请求总数 + 能接受的超过目标并发数的请求数量 > 每个 Pod 的目标并发数

* Pod 数量

case 2 - 基于并发弹性伸缩

我们首先在集群应用下面 YAML 定义。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: "1"
        autoscaling.knative.dev/class: "kpa.autoscaling.knative.dev"
    spec:
      containers:
        - image: m.daocloud.io/ghcr.io/knative/helloworld-go:latest
          ports:
            - containerPort: 8080
          env:
            - name: TARGET
              value: "World"
```

执行下面命令测试，并可以通过 kubectl get pods -A -w 来观察扩容的 Pod。

wrk -t2 -c4 -d6s http://hello.knative-serving.knative.daocloud.io/

case 3 - 基于并发弹性伸缩，达到特定比例提前扩容

我们 可以 很 轻 松 的 实 现 ， 例 如 限 制 每 个 容 器 并 发 为 10 ， 可 以 通 过 autoscaling.knative.dev/target-utilization-percentage: 70 来 实 现 ， 达 到 70% 就 开 始 扩 容 Pod。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: "10"
        autoscaling.knative.dev/class: "kpa.autoscaling.knative.dev"
        autoscaling.knative.dev/target-utilization-percentage: "70"
        autoscaling.knative.dev/metric: "concurrency"
    spec:
      containers:
        - image: m.daocloud.io/ghcr.io/knative/helloworld-go:latest
          ports:
            - containerPort: 8080
          env:
            - name: TARGET
              value: "World"
```

case 4 - 灰度发布/流量百分比

我 们 可 以 通 过 spec.traffic 实 现 到 每 个 版 本 流 量 的 控 制。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: "1"
        autoscaling.knative.dev/class: "kpa.autoscaling.knative.dev"
```

```

spec:
  containers:
    - image: m.daocloud.io/ghcr.io/knative/helloworld-go:latest
      ports:
        - containerPort: 8080
      env:
        - name: TARGET
          value: "World"
  traffic:
    - latestRevision: true
      percent: 50
    - latestRevision: false
      percent: 50
  revisionName: hello-00001

```

Helm 模板

Helm 是 Kubernetes 的包管理工具，方便用户快速发现、共享和使用 Kubernetes 构建的应用。第五代[容器管理模块](#)提供了上百个 Helm 模板，涵盖存储、网络、监控、数据库等主要场景。借助这些模板，您可以通过 UI 界面快速部署、便捷管理 Helm 应用。此外，支持通过[添加 Helm 仓库](#) 添加更多的个性化模板，满足多样需求。

模板

模板

关键概念：

使用 Helm 时需要了解以下几个关键概念：

- Chart：一个 Helm 安装包，其中包含了运行一个应用所需要的镜像、依赖和资源定义等，还可能包含 Kubernetes 集群中的服务定义，类似 Homebrew 中的 formula、APT 的 dpkg 或者 Yum 的 rpm 文件。Chart 在 DCE 5.0 中称为 **Helm 模板**。
- Release：在 Kubernetes 集群上运行的一个 Chart 实例。一个 Chart 可以在同一个集群内多次安装，每次安装都会创建一个新的 Release。Release 在 DCE 5.0 中称为

Helm 应用。

- Repository：用于发布和存储 Chart 的存储库。Repository 在 DCE 5.0 中称为 **Helm 仓库**。

更多详细信息，请前往 [Helm 官网](#)查看。

相关操作：

- [上传 Helm 模板](#)，介绍上传 Helm 模板操作。
- [管理 Helm 应用](#)，包括安装、更新、卸载 Helm 应用，查看 Helm 操作记录等。
- [管理 Helm 仓库](#)，包括安装、更新、删除 Helm 仓库等。

上传 Helm 模板

本文介绍如何上传 Helm 模板，操作步骤见下文。

1. 引入 Helm 仓库，操作步骤参考[引入第三方 Helm 仓库](#)。

2. 上传 Helm Chart 到 Helm 仓库。

==== “客户端上传”

!!! note

此方式适用于 Harbor、ChartMuseum、JFrog 类型仓库。

1. 登录一个可以访问到 Helm 仓库的节点，将 Helm 二进制文件上传到节点，并安装 cm-push 插件（需要连通外网并提前安装 [Git](<https://git-scm.com/downloads>)）。

安装插件流程参考[安装 cm-push 插件](<https://github.com/chartmuseum/helm-push>)。

2. 推送 Helm Chart 到 Helm 仓库，执行如下命令：

```
```shell
helm cm-push ${charts-dir} ${HELM_REPO_URL} --username ${username} --password ${password}
```

---

字段说明：

- `charts-dir`：Helm Chart 的目录，或者是打包好的 Chart（即 .tgz 文件）。
- `HELM\_REPO\_URL`：Helm 仓库的 URL。
- `username`/`password`：有推送权限的 Helm 仓库用户名和密码。
- 如果采用 https 访问且需要跳过证书验证，可添加参数 `--insecure`

==== “页面上传”

!!! note

此方式仅适用于 Harbor 类型仓库。

1. 登录网页 Harbor 仓库，请确保登录用户有推送权限；
2. 进入到对应项目，选择 Helm Charts 页签，点击页面 上传 按钮，完成 Helm Chart 上传。

![上传 Helm Chart](docs/zh/docs/kpanda/images/upload-helm-01.png)

3. 同步远端仓库数据

==== “手动同步”

默认集群未开启 Helm 仓库自动刷新，需要执行手动同步操作，大致步骤为：

进入 Helm 应用 -> Helm 仓库，点击仓库列表右侧的 | 按钮，选择 同步仓库，完成仓库数据同步。

![上传 Helm Chart](docs/zh/docs/kpanda/images/upload-helm-02.png)

==== “自动同步”

如需开启 Helm 仓库自动同步功能，可进入 集群运维 -> 集群设置 -> 高级配置，开启 Helm 仓库自动刷新开关。

![自动同步](docs/zh/docs/kpanda/user-guide/images/auto-helm.png)

## 管理 Helm 应用

容器管理模块支持对 Helm 进行界面化管理，包括使用 Helm 模板创建 Helm 实例、自定义 Helm 实例参数、对 Helm 实例进行全生命周期管理等功能。

本节将以 [cert-manager](#) 为例，介绍如何通过容器管理界面创建并管理 Helm 应用。

## 前提条件

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 已完成一个[命名空间的创建](#)、[用户的创建](#)，并为用户授予 [NS Admin](#) 或更高权限，详情可参考[命名空间授权](#)。

## 安装 Helm 应用

参照以下步骤安装 Helm 应用。

1. 点击一个集群名称，进入 [集群详情](#)。

集群详情

集群详情

2. 在左侧导航栏，依次点击 **Helm 应用 -> Helm 模板**，进入 Helm 模板页面。

在 Helm 模板页面选择名为 **addon** 的 [Helm 仓库](#)，此时界面上将呈现 **addon** 仓库下所有的 Helm chart 模板。点击名称为 **cert-manager** 的 Chart。

找到 chart

找到 chart

3. 在安装页面，能够看到 Chart 的相关详细信息，在界面右上角选择需要安装的版本，

点击 **安装** 按钮。此处选择 v1.9.1 版本进行安装。

点击安装

点击安装

4. 配置 **名称**、**命名空间** 及 **版本信息**，也可以在下方的 **参数配置** 区域通过修改

YAML 来自定义参数。点击 **确定**。

填写参数

填写参数

5. 系统将自动返回 Helm 应用列表，新创建的 Helm 应用状态为 **安装中**，等待一段时

间后状态变为 **运行中**。

查看状态

查看状态

## 更新 Helm 应用

当我们通过界面完成一个 Helm 应用的安装后，我们可以对 Helm 应用执行更新操作。注

意：只有通过界面安装的 Helm 应用才支持使用界面进行更新操作。

参照以下步骤更新 Helm 应用。

1. 点击一个集群名称，进入 **集群详情**。

集群详情

集群详情

2. 在左侧导航栏，点击 **Helm 应用**，进入 Helm 应用列表页面。

在 Helm 应用列表页选择需要更新的 Helm 应用，点击列表右侧的 **操作** 按钮，在下拉选择中选择 **更新** 操作。

点击更新

点击更新

3. 点击 **更新** 按钮后，系统将跳转至更新界面，您可以根据需要对 Helm 应用进行更新，

此处我们以更新 **dao-2048** 这个应用的 http 端口为例。

更新页面

更新页面

4. 修改完相应参数后。您可以在参数配置下点击 **变化** 按钮，对比修改前后的文件，确定无误后，点击底部 **确定** 按钮，完成 Helm 应用的更新。

对比变化

对比变化

5. 系统将自动返回 Helm 应用列表，右上角弹窗提示 **更新成功**。

更新成功

更新成功

## 查看 Helm 操作记录

Helm 应用的每次安装、更新、删除都有详细的操作记录和日志可供查看。

1. 在左侧导航栏，依次点击 **集群运维 -> 最近操作**，然后在页面上方选择 **Helm 操作** 标签页。每一条记录对应一次安装/更新/删除操作。

helm 操作

helm 操作

2. 如需查看每一次操作的详细日志：在列表右侧点击 ，在弹出菜单中选择 **日志**。

选择日志

选择日志

3. 此时页面下方将以控制台的形式展示详细的运行日志。

查看运行日志

查看运行日志

## 删除 Helm 应用

参照以下步骤删除 Helm 应用。

1. 找到待删除的 Helm 应用所在的集群，点击集群名称，进入 **集群详情**。

集群详情

集群详情

2. 在左侧导航栏，点击 **Helm 应用**，进入 Helm 应用列表页面。

在 Helm 应用列表页选择您需要删除的 Helm 应用，点击列表右侧的 **操作** 按钮，

在下拉选择中选择 **删除**。

点击删除

点击删除

3. 在弹窗内输入 Helm 应用的名称进行确认，然后点击 **删除** 按钮。

确认删除

确认删除

## 管理 Helm 仓库

Helm 仓库是用来存储和发布 Chart 的存储库。Helm 应用模块支持通过 HTTP(s) 协议来访问存储库中的 Chart 包。系统默认内置了下表所示的 4 个 Helm 仓库以满足企业生产过程中的常见需求。

仓库	描述	示例
----	----	----

仓库	描述	示例
partner	由生态合作伙伴所提供的各类优质	tidb
	特色 Chart	
system	系统核心功能组件及部分高级功能	Insight
	所必需依赖的 Chart，如必需安装	
	insight-agent 才能够获取集群的监	
	控信息	
addon	业务场景中常见的 Chart	cert-manager
community	Kubernetes 社区较为热门的开源组	Istio
	件 Chart	

除上述预置仓库外，您也可以自行添加第三方 Helm 仓库。本文将介绍如何添加、更新第三方 Helm 仓库。

## 前提条件

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面
- 已完成一个[命名空间的创建](#)、[用户的创建](#)，并为用户授予 [NS Admin](#) 或更高权限，详情可参考[命名空间授权](#)。
- 如果使用私有仓库，当前操作用户应拥有对该私有仓库的读写权限。

## 引入第三方 Helm 仓库

下面以 Kubevela 公开的镜像仓库为例，引入 Helm 仓库并管理。

1. 找到需要引入第三方 Helm 仓库的集群，点击集群名称，进入 **集群详情**

集群别名	网络模式	Kubernetes 版本	节点数
kpanda-e2e-cluster	Calico	v1.23.0	3 / 3 正常/节点总数
kpanda-global-cluster	Calico	v1.27.5	1 / 1 正常/节点总数

**集群详情**

2. 在左侧导航栏，依次点击 **Helm 应用 -> Helm 仓库**，进入 Helm 仓库页面。

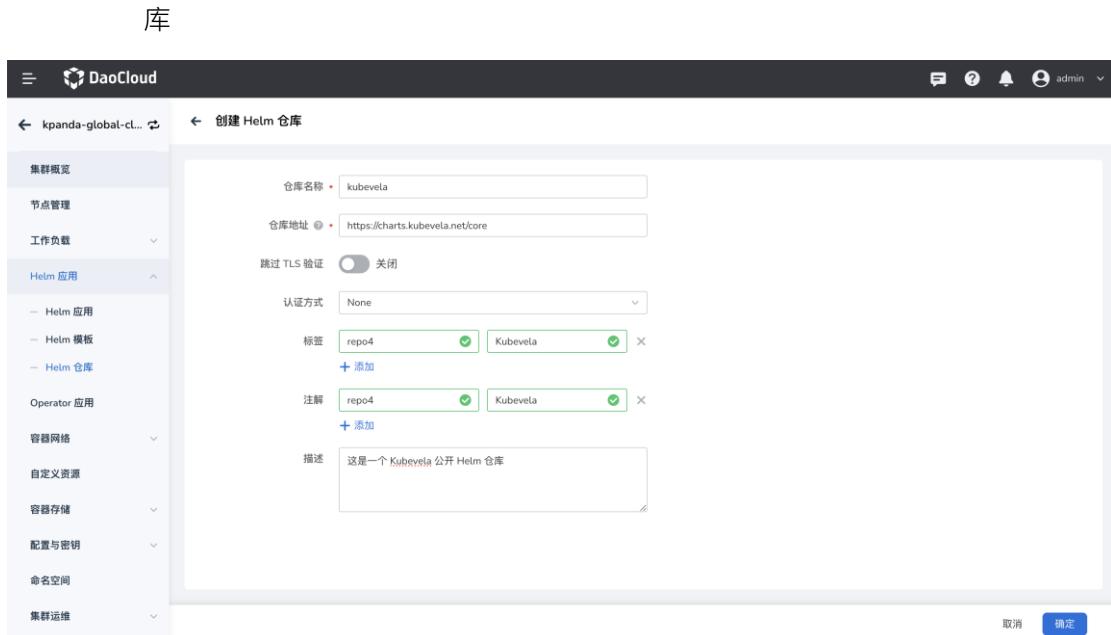
仓库名称	状态	URL	创建时间	最近一次同步时间	最近一次同步状态
addon	运行中	http://10.6.201.201:8081	2024-07-19 11:36	2024-07-19 11:36	同步成功
kpanda	运行中	https://release.daocloud.io/chartrepo/kpanda	2024-07-19 11:18	2024-07-19 11:18	同步成功
kubeain	运行中	https://kubeain-io.github.io/kubeain-helm-chart	2024-06-25 14:08	2024-06-25 14:08	同步成功
charts-syncer	运行中	https://release-ci.daocloud.io/chartrepo/charts-syncer	2024-05-07 14:46	2024-05-14 10:49	同步成功

**helm 仓库**

3. 在 Helm 仓库页面点击 **创建仓库** 按钮，进入创建仓库页面，按照下表配置相关参数。

- 仓库名称：设置仓库名称。最长 63 个字符，只能包含小写字母、数字及分隔符 -，且必须以小写字母或数字开头并结尾，例如 kubevela

- 仓库地址：用来指向目标 Helm 仓库的 http ( s ) 地址。例如 <https://kubvela.github.io/charts>
- 跳过 TLS 验证：如果添加的 Helm 仓库为 https 地址且需跳过 TLS 验证，可以勾选此选项，默认为不勾选
- 认证方式：连接仓库地址后用来进行身份校验的方式。对于公开仓库，可以选择 None，私有的仓库需要输入用户名/密码以进行身份校验
- 标签：为该 Helm 仓库添加标签。例如 key: repo4 ; value: Kubevela
- 注解：为该 Helm 仓库添加注解。例如 key: repo4 ; value: Kubevela
- 描述：为该 Helm 仓库添加描述。例如：这是一个 Kubevela 公开 Helm 仓库



### 填写参数

4. 点击 **确定**，完成 Helm 仓库的创建。页面会自动跳转至 Helm 仓库列表。

**确定**

## 更新 Helm 仓库

当 Helm 仓库的地址信息发生变化时，可以更新 Helm 仓库的地址、认证方式、标签、注解及描述信息。

### 1. 找到待更新仓库所在的集群，点击集群名称，进入 集群详情

**集群详情**

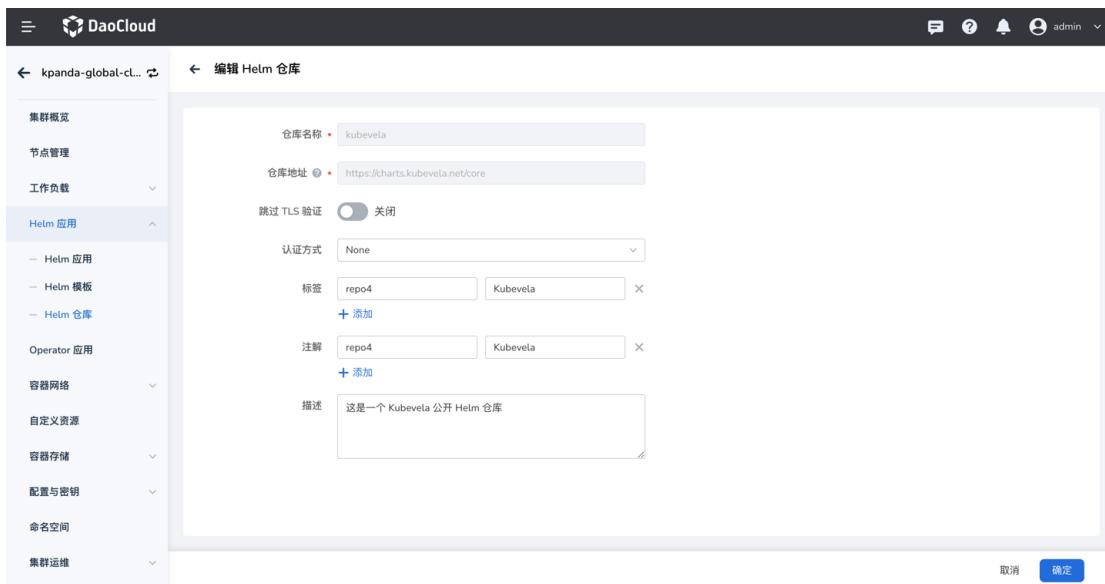
### 2. 在左侧导航栏，依次点击 Helm 应用 -> Helm 仓库，进入 Helm 仓库列表页面。

### helm 仓库

3. 在仓库列表页面找到需要更新的 Helm 仓库，在列表右侧点击  按钮，在弹出菜单中点击 **更新**

### 点击更新

4. 在 编辑 Helm 仓库 页面进行更新，完成后点击 确定



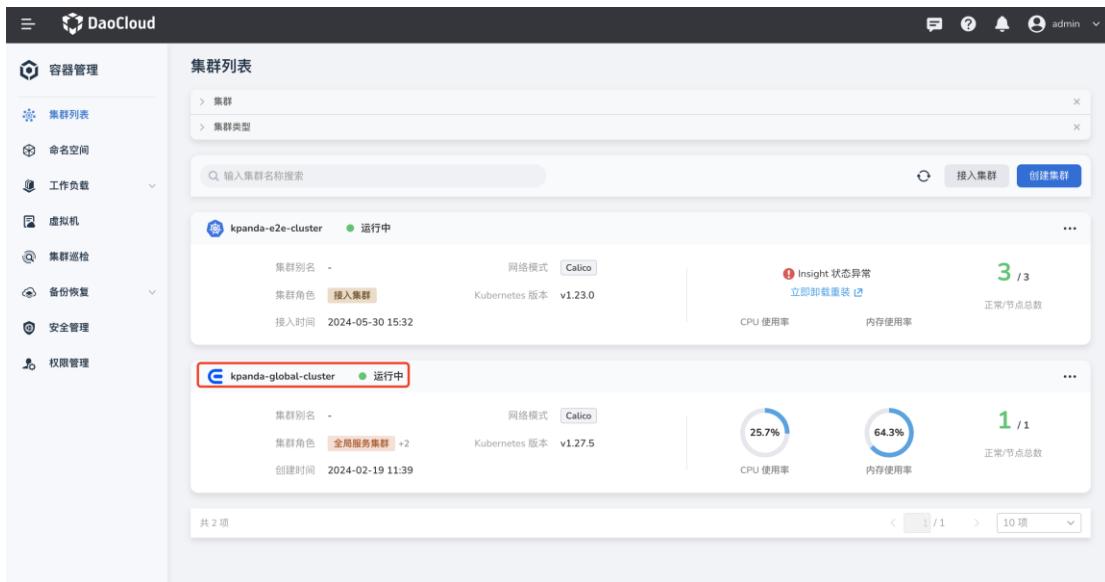
**确定**

5. 返回 Helm 仓库列表，屏幕提示更新成功。

## 删除 Helm 仓库

除了引入、更新仓库外，您也可以将不需要的仓库删除，包括系统预置仓库和第三方仓库。

1. 找到待删除仓库所在的集群，点击集群名称，进入 **集群详情**



**集群详情**

2. 在左侧导航栏，依次点击 Helm 应用 -> Helm 仓库，进入 Helm 仓库列表页面。

仓库名称	状态	URL	创建时间	最近一次同步时间	最近一次同步状态
addon	运行中	http://10.6.201.201:8081	2024-07-19 11:36	2024-07-19 11:36	同步成功
kpanda	运行中	https://release.daocloud.io/chartrepo/kpanda	2024-07-19 11:18	2024-07-19 11:18	同步成功
kubebean	运行中	https://kubean-io.github.io/kubebean-helm-chart	2024-06-25 14:08	2024-06-25 14:08	同步成功
charts-syncer	运行中	https://release-ci.daocloud.io/chartrepo/charts-syncer	2024-05-07 14:46	2024-05-14 10:49	同步成功

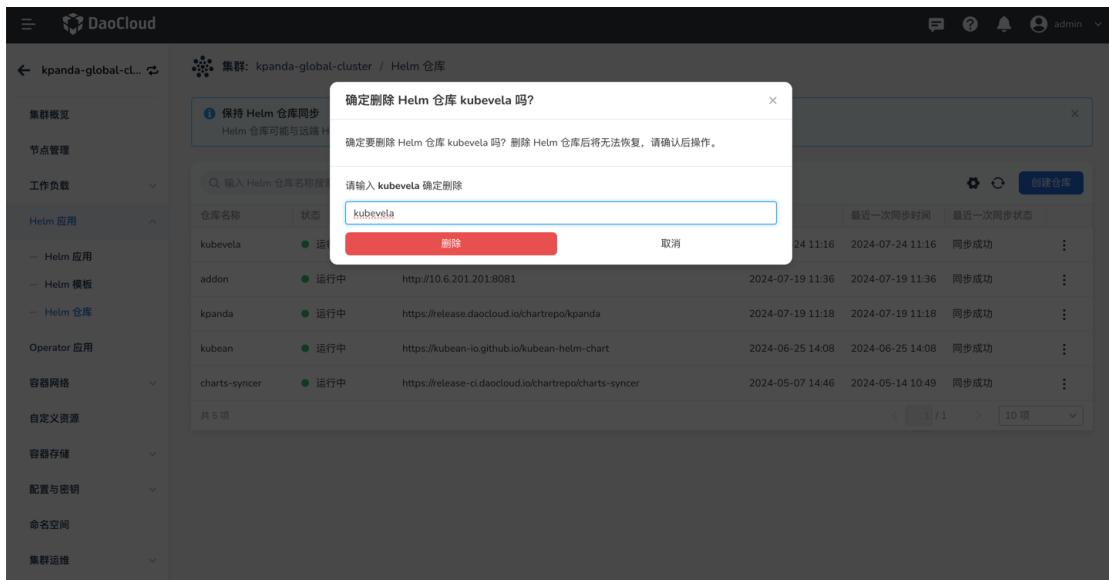
helm 仓库

3. 在仓库列表页面找到需要更新的 Helm 仓库，在列表右侧点击 按钮，在弹出菜单中点击 **删除**

仓库名称	状态	URL	创建时间	最近一次同步时间	最近一次同步状态
kubevela	运行中	https://charts.kubevela.net/core	2024-07-24 11:16	2024-07-24 11:16	同步成功
addon	运行中	http://10.6.201.201:8081	2024-07-19 11:36	2024-07-19 11:36	更新
kpanda	运行中	https://release.daocloud.io/chartrepo/kpanda	2024-07-19 11:18	2024-07-19 11:18	克隆
kubebean	运行中	https://kubean-io.github.io/kubebean-helm-chart	2024-06-25 14:08	2024-06-25 14:08	同步仓库
charts-syncer	运行中	https://release-ci.daocloud.io/chartrepo/charts-syncer	2024-05-07 14:46	2024-05-14 10:49	修改标签
					修改注解

点击删除

4. 输入仓库名称进行确认，点击 **删除**



确认删除

5. 返回 Helm 仓库列表，屏幕提示删除成功。

## Helm 应用多架构和升级导入步骤

通常在多架构集群中，也会使用多架构的 Helm 包来部署应用，以解决架构差异带来的部署问题。本文将介绍如何将单架构 Helm 应用融合为多架构，以及多架构与多架构 Helm 应用的相互融合。

### 导入

#### 单架构导入

准备好待导入的离线包 `addon-offline-full-package-${version}-${arch}.tar.gz`，可从 [下载中心](#)

下载。把路径填写至 `clusterConfig.yml` 配置文件，例如：

`addonPackage:`

`path: "/home/addon-offline-full-package-v0.9.0-amd64.tar.gz"`

然后执行导入命令：

```
~/dce5-installer cluster-create -c /home/dce5/sample/clusterConfig.yaml -m /home/dce5/sample/manifest.yaml -d -j13
```

## 多架构融合

准备好待融合的离线包 `addon-offline-full-package-${version}-${arch}.tar.gz`，可从[下载中心](#)下载。

以 `addon-offline-full-package-v0.9.0-arm64.tar.gz` 为例，执行导入命令：

```
~/dce5-installer import-addon -c /home/dce5/sample/clusterConfig.yaml --addon-path=/home/addon-offline-full-package-v0.9.0-arm64.tar.gz
```

## 升级

### 单架构升级

准备好待导入的离线包 `addon-offline-full-package-${version}-${arch}.tar.gz`，可从[下载中心](#)下载。

把路径填写至 `clusterConfig.yaml` 配置文件，例如：

`addonPackage:`

```
path: "/home/addon-offline-full-package-v0.11.0-amd64.tar.gz"
```

然后执行导入命令：

```
~/dce5-installer cluster-create -c /home/dce5/sample/clusterConfig.yaml -m /home/dce5/sample/manifest.yaml -d -j13
```

## 多架构融合

准备好待融合的离线包 `addon-offline-full-package-${version}-${arch}.tar.gz`，可从[下载中心](#)下载。

以 `addon-offline-full-package-v0.11.0-arm64.tar.gz` 为例，执行导入命令：

```
~/dce5-installer import-addon -c /home/dce5/sample/clusterConfig.yaml --addon-path=/home/addon-offline-full-package-v0.11.0-arm64.tar.gz
```

## 注意事项

### 磁盘空间

离线包比较大，且过程中需要解压和 load 镜像，需要预留充足的空间，否则可能在过程中报“no space left”而中断。

### 失败后重试

如果在多架构融合步骤执行失败，重试前需要清理一下残留：

```
rm -rf addon-offline-target-package
```

### 镜像空间

如果融合的离线包中包含了与导入的离线包不一致的镜像空间，可能会在融合过程中因为镜像空间不存在而报错：

```
helm
```

```
helm
```

解决办法：只需要在融合之前创建好该镜像空间即可，例如上图报错可通过创建镜像空间 localhost 提前避免。

### 架构冲突

升级至低于 0.12.0 版本的 addon 时，由于目标离线包里的 charts-syncer 没有检查镜像存在则不推送功能，因此会在升级的过程中会重新把多架构冲成单架构。例如：在 v0.10 版本将 addon 实现为多架构，此时若升级为 v0.11 版本，则多架构 addon 会被覆盖为单架构；若升级为 0.12.0 及以上版本则仍能够保持多架构。

# 将自定义 Helm 应用导入系统内置 Addon

本文从离线和在线两种环境说明如何将 Helm 应用导入到系统内置的 Addon 中。

## 离线环境

离线环境指的是无法连通互联网或封闭的私有网络环境。

## 前提条件

- 存在可以运行的 [charts-syncer](#)。若没有，可[点击下载](#)。
- Helm Chart 已经完成适配 [charts-syncer](#)。即在 Helm Chart 内添加了 `.relok8s-images.yaml` 文件。该文件需要包含 Chart 中所有使用到镜像，也可以包含 Chart 中未直接使用的镜像，类似 Operator 中使用的镜像。

### !!! note

- 如何编写 Chart 可参考 [image-hints-file](<https://github.com/vmware-tanzu/asset-relocation-tool-for-kubernetes#image-hints-file>)。  
要求镜像的 registry 和 repository 必须分开，因为 load 镜像时需替换或修改 registry/repository。
- 安装器所在的火种集群已安装 [charts-syncer](<https://github.com/DaoCloud/charts-syncer>)。  
若将自定义 Helm 应用导入安装器所在火种集群，可跳过下载直接适配；  
若未安装 [charts-syncer](<https://github.com/DaoCloud/charts-syncer>) 二进制文件，  
可[\[立即下载\]](#)(<https://github.com/DaoCloud/charts-syncer/releases>)。

## 同步 Helm Chart

1. 进入容器管理 -> Helm 应用 -> Helm 仓库，搜索 addon，获取内置仓库地址和用户名 /密码（系统内置仓库默认用户名/密码为 rootuser/rootpass123）。

```
helmlist
helmlist
helmdetail
helmdetail
```

## 1. 同步 Helm Chart 到容器管理内置仓库 Addon

- 编写如下配置文件，可以根据具体配置修改，并保存为

`sync-dao-2048.yaml`。

```
source: # helm charts 源信息
repo:
 kind: HARBOR # 也可以是任何其他支持的 Helm Chart 仓库类别，比如 CHARTMUSEUM
 url: https://release-ci.daocloud.io/chartrepo/community # 需更改为 chart repo url
 #auth: # 用户名/密码，若没有设置密码可以不填写
 #username: "admin"
 #password: "Harbor12345"
charts: # 需要同步
 - name: dao-2048 # helm charts 信息，若不填写则同步源 helm repo 内所有 charts
 versions:
 - 1.4.1
target: # helm charts 目标信息
 containerRegistry: 10.5.14.40 # 镜像仓库 url
repo:
 kind: CHARTMUSEUM # 也可以是任何其他支持的 Helm Chart 仓库类别，比如 HARBOR
 url: http://10.5.14.40:8081 # 需更改为正确 chart repo url，可以通过 helm repo add $HELM-REPO 验证地址是否正确
 auth: # 用户名/密码，若没有设置密码可以不填写
 #username: "rootuser"
 #password: "rootpass123"
containers:
 # kind: HARBOR # 若镜像仓库为 HARBOR 且希望 charts-syncer 自动创建镜像 Repository 则填写该字段
 # auth: # 用户名/密码，若没有设置密码可以不填写
 #username: "admin"
 #password: "Harbor12345"

leverage .relok8s-images.yaml file inside the Charts to move the container images too
relocateContainerImages: true
```

- 执行 charts-syncer 命令同步 Chart 及其包含的镜像

`charts-syncer sync --config sync-dao-2048.yaml --insecure --auto-create-repository`

预期输出为：

```
I1222 15:01:47.119777 8743 sync.go:45] Using config file: "examples/sync-da
o-2048.yaml"
W1222 15:01:47.234238 8743 syncer.go:263] Ignoring skipDependencies opt
ion as dependency sync is not supported if container image relocation is true
or syncing from/to intermediate directory
I1222 15:01:47.234685 8743 sync.go:58] There is 1 chart out of sync!
I1222 15:01:47.234706 8743 sync.go:66] Syncing "dao-2048_1.4.1" chart...
.relok8s-images.yaml hints file found
Computing relocation...

Relocating dao-2048@1.4.1...
Pushing 10.5.14.40/daocloud/dao-2048:v1.4.1...
Done
Done moving /var/folders/vm/08vw0t3j68z9z_4lcqyhg8nm0000gn/T/charts-syncer8
69598676/dao-2048-1.4.1.tgz
```

2.待上一步执行完成后，进入容器管理 -> Helm 应用 -> Helm 仓库，找到对应 Addon，

在操作栏点击同步仓库，回到 Helm 模板就可以看到上传的 Helm 应用

helm 同步

helm 同步

详情 2048

详情 2048

3.后续可正常进行安装、升级、卸载

安装升级

安装升级

## 在线环境

在线环境的 Helm Repo 地址为 release.daocloud.io。如果用户无权限添加 Helm Repo，则无法将自定义 Helm 应用导入系统内置 Addon。您可以添加自己搭建的 Helm 仓库，然后按照离线环境中同步 Helm Chart 的步骤将您的 Helm 仓库集成到平台使用。

# 导入离线 Minio Operator

本文将介绍在离线环境下如何导入 Minio Operator。

## 前提条件

- 当前集群已接入容器管理且全局服务集群已经安装 **kolm** 组件（ helm 模板搜索 kolm ）
- 当前集群已经安装 **olm** 组件且版本  $\geq 0.2.4$  (helm 模板搜索 olm)
- 支持执行 Docker 命令
- 准备一个镜像仓库

## 操作步骤

1. 在执行环境中设置环境变量并在后续步骤使用，执行命令：

```
export OPM_IMG=10.5.14.200/quay.m.daocloud.io/operator-framework/opm:v1.29.0
export BUNDLE_IMG=10.5.14.200/quay.m.daocloud.io/operatorhubio/minio-operator:v5.0.3
```

如何获取上述镜像地址：

前往 **容器管理** -> 选择当前集群 -> **helm** 应用 -> 查看 **olm** 组件 -> **插件设置**，找到后续步骤所需 opm , minio , minio bundle , minio operator 的镜像。

olm  
olm

以上诉截图为例，则四个镜像地址如下

```
opm 镜像
10.5.14.200/quay.m.daocloud.io/operator-framework/opm:v1.29.0

minio 镜像
10.5.14.200/quay.m.daocloud.io/minio/minio:RELEASE.2023-03-24T21-41-23Z
```

```
minio bundle 镜像
10.5.14.200/quay.m.daocloud.io/operatorhubio/minio-operator:v5.0.3
```

```
minio operator 镜像
10.5.14.200/quay.m.daocloud.io/minio/operator:v5.0.3
```

2. 执行 opm 命令获取离线 bundle 镜像包含的 operator。

```
创建 operator 存放目录
$ mkdir minio-operator && cd minio-operator

获取 operator yaml
$ docker run --user root -v $PWD/minio-operator:/minio-operator ${OPM_IMG} alpha
bundle unpack --skip-tls-verify -v -d ${BUNDLE_IMG} -o ./minio-operator
```

# 预期结果

```
.
└── minio-operator
 ├── manifests
 │ ├── console-env_v1_configmap.yaml
 │ ├── console-sa-secret_v1_secret.yaml
 │ ├── console_v1_service.yaml
 │ ├── minio-operator.clusterserviceversion.yaml
 │ ├── minio.min.io_tenants.yaml
 │ ├── operator_v1_service.yaml
 │ ├── sts.min.io_policybindings.yaml
 │ └── sts_v1_service.yaml
 └── metadata
 └── annotations.yaml
```

3 directories, 9 files

3. 替换 minio-operator/manifests/minio-operator.clusterserviceversion.yaml 文件中的所有镜像地址为离线镜像仓库地址镜像。

替换前：

```
image1
image1
```

替换后：

```
image2
image2
```

4. 生成构建 bundle 镜像的 Dockerfile

```
$ docker run --user root -v $PWD:/minio-operator -w /minio-operator ${OPM_IMG} alpha
 a bundle generate --channels stable,beta -d /minio-operator/minio-operator/manifests -e stable -p minio-operator
```

# 预期结果

```
.
├── bundle.Dockerfile
└── minio-operator
 ├── manifests
 │ ├── console-env_v1_configmap.yaml
 │ ├── console-sa-secret_v1_secret.yaml
 │ ├── console_v1_service.yaml
 │ ├── minio-operator.clusterserviceversion.yaml
 │ ├── minio.min.io_tenants.yaml
 │ ├── operator_v1_service.yaml
 │ ├── sts.min.io_policybindings.yaml
 │ └── sts_v1_service.yaml
 └── metadata
 └── annotations.yaml
```

3 directories, 10 files

#### 5. 执行构建命令，构建 bundle 镜像且推送到离线 registry。

```
设置新的 bundle image
export OFFLINE_BUNDLE_IMG=10.5.14.200/quay.m.daocloud.io/operatorhubio/minio-operator:v5.0.3-offline

$ docker build . -f bundle.Dockerfile -t ${OFFLINE_BUNDLE_IMG}

$ docker push ${OFFLINE_BUNDLE_IMG}
```

#### 6. 生成构建 catalog 镜像的 Dockerfile。

```
$ docker run --user root -v $PWD:/minio-operator -w /minio-operator ${OPM_IMG} index
 ex add --bundles ${OFFLINE_BUNDLE_IMG} --generate --binary-image ${OPM_IMG}
} --skip-tls-verify
```

# 预期结果

```
.
├── bundle.Dockerfile
├── database
│ └── index.db
├── index.Dockerfile
└── minio-operator
 └── manifests
```

```

| └── console-env_v1_configmap.yaml
| └── console-sa-secret_v1_secret.yaml
| └── console_v1_service.yaml
| └── minio.min.io_tenants.yaml
| └── minio-operator.clusterserviceversion.yaml
| └── operator_v1_service.yaml
| └── sts.min.io_policybindings.yaml
| └── sts_v1_service.yaml
└── metadata
 └── annotations.yaml

```

4 directories, 12 files

## 7. 构建 catalog 镜像

```

设置新的 catalog image
export OFFLINE_CATALOG_IMG=10.5.14.200/release.daocloud.io/operator-framework/sy
stem-operator-index:v0.1.0-offline

$ docker build . -f index.Dockerfile -t ${OFFLINE_CATALOG_IMG}

$ docker push ${OFFLINE_CATALOG_IMG}

```

## 8. 前往容器管理，更新 helm 应用 olm 的内置 catsrc 镜像（填写构建 catalog 镜像指 定的 \${catalog-image} 即可）

```

olm1
olm1
olm2
olm2
olm3
olm3

```

## 9. 更新成功后，Operator Hub 中会出现 **minio-operator** 组件

```

olm3
olm3

```

# 创建服务 ( Service )

在 Kubernetes 集群中，每个 Pod 都有一个内部独立的 IP 地址，但是工作负载中的 Pod 可能会被随时创建和删除，直接使用 Pod IP 地址并不能对外提供服务。

这就需要创建服务，通过服务您会获得一个固定的 IP 地址，从而实现工作负载前端和后

端的解耦，让外部用户能够访问服务。同时，服务还提供了负载均衡（LoadBalancer）功能，使用户能从公网访问到工作负载。

## 前提条件

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 已完成一个[命名空间的创建](#)、[用户的创建](#)，并将用户授权为 [NS Editor](#) 角色，详情可参考[命名空间授权](#)。
- 单个实例中有多个容器时，请确保容器使用的端口不冲突，否则部署会失效。

## 创建服务

- 以 [NS Editor](#) 用户成功登录后，点击左上角的 [集群列表](#) 进入 [集群列表](#) 页面。在集群列表中，点击一个集群名称。

集群列表

集群列表

- 在左侧导航栏中，点击 [容器网络 -> 服务](#) 进入服务列表，点击右上角 [创建服务](#) 按钮。

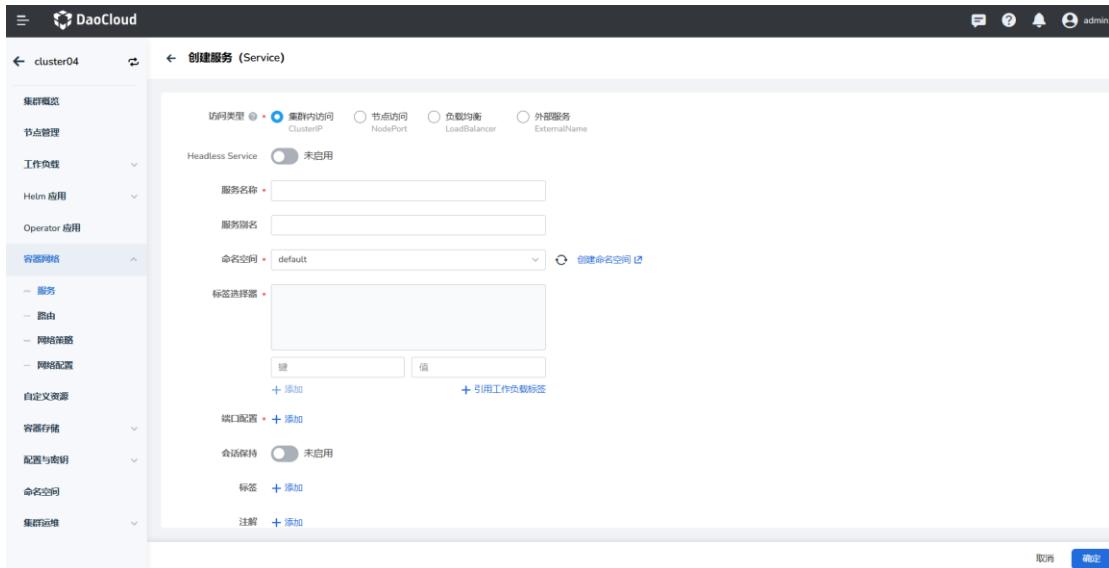
服务与路由

服务与路由

!!! tip

也可以[通过 YAML 创建一个服务](#yaml)。

- 打开 [创建服务](#) 页面，选择一种访问类型，参考以下几个参数表进行配置。



## 创建服务

### == “创建 ClusterIP 服务”

点选 \_\_集群内访问 (ClusterIP) \_\_，这是指通过集群的内部 IP 暴露服务，  
选择此项的服务只能在集群内部访问。这是默认的服务类型。

参数   说明   举例值
---   ---   ---
访问类型   【类型】必填 【含义】指定 Pod 服务发现的方式，这里选择集群内访问 (ClusterIP)。  ClusterIP
服务名称   【类型】必填 【含义】输入新建服务的名称。 【注意】请输入 4 到 63 个字符的字符串，可以包含小写英文字母、数字和中划线 (-)，并以小写英文字母开头，小写英文字母或数字结尾。  Svc-01
命名空间   【类型】必填 【含义】选择新建服务所在的命名空间。关于命名空间更多信息请参考[命名空间概述](./namespaces/createnamespace.md)。 【注意】请输入 4 到 63 个字符的字符串，可以包含小写英文字母、数字和中划线 (-)，并以小写英文字母开头，小写英文字母或数字结尾。  default
标签选择器   【类型】必填 【含义】添加标签，Service 根据标签选择 Pod，填写后点击 **添加**。也可以引用已有工作负载的标签，点击 __引用负载标签__，在弹出的窗口中选择负载，系统会默认将所选的负载标签作为选择器。  app:job01
端口配置   【类型】必填 【含义】为服务添加协议端口，需要先选择端口协议类型，目前支持 TCP、UDP 两种传输协议。 **端口名称**：输入自定义的端口的名称。 **服务端口 (port)**：Pod 对外提供服务的访问端口。 **容器端口 (targetport)**：工作负载实际监听的容器端口，用来对集群内暴露服务。
会话保持   【类型】选填 【含义】开启后，相同客户端的请求将转发至同一 Pod   开启
会话保持最大时长   【类型】选填 【含义】开启会话保持后，保持的最大时长   30 秒
注解   【类型】选填 【含义】为服务添加注解

### ==== “创建 NodePort 服务”

点选 \_\_节点访问 (NodePort) \_\_，这是指通过每个节点上的 IP 和静态端口 ( \_\_NodePort\_\_ ) 暴露服务。

\_\_NodePort\_\_ 服务会路由到自动创建的 \_\_ClusterIP\_\_ 服务。通过请求 \_\_<节点 IP>: <节点端口>\_\_，

您可以从集群的外部访问一个 \_\_NodePort\_\_ 服务。

| 参数 | 说明 | 举例值 |

| --- | :-- | :---- |

| 访问类型 |【类型】必填<br />【含义】指定 Pod 服务发现的方式 | NodePort |

| 服务名称 |【类型】必填<br />【含义】输入新建服务的名称。<br />【注意】请输入 4 到 63 个字符的字符串，可以包含小写英文字母、数字和中划线 (-)，并以小写英文字母开头，小写英文字母或数字结尾。| Svc-01 |

| 命名空间 |【类型】必填<br />【含义】选择新建服务所在的命名空间。关于命名空间更多信息请参考[命名空间概述](./namespaces/createns.md)。<br />【注意】请输入 4 到 63 个字符的字符串，可以包含小写英文字母、数字和中划线 (-)，并以小写英文字母开头，小写英文字母或数字结尾。| default |

| 标签选择器 |【类型】必填<br />【含义】添加标签，Service 根据标签选择 Pod，填写后点击 \*\*添加\*\*。也可以引用已有工作负载的标签，点击 \_\_引用负载标签\_\_，在弹出的窗口中选择负载，系统会默认将所选的负载标签作为选择器。||

| 端口配置 |【类型】必填<br />【含义】为服务添加协议端口，需要先选择端口协议类型，目前支持 TCP、UDP 两种传输协议。<br />\*\*端口名称\*\*：输入自定义的端口的名称。<br />\*\*服务端口 (port)\*\*：Pod 对外提供服务的访问端口。默认为了方便起见，服务端口被设置为与容器端口字段相同的值。<br />\*\*容器端口 (targetport)\*\*：工作负载实际监听的容器端口。<br />\*\*节点端口 (nodeport)\*\*：节点的端口，接收来自 ClusterIP 传输的流量。用来做外部流量访问的入口。||

| 会话保持 |【类型】选填<br />【含义】开启后，相同客户端的请求将转发至同一 Pod<br />开启后 Service 的 `spec.sessionAffinity` 为 \_\_ClientIP\_\_，详情请参考 [Service 的会话亲和性](https://kubernetes.io/zh-cn/docs/reference/networking/virtual-ips/#session-affinity) | 开启 |

| 会话保持最大时长 |【类型】选填<br />【含义】开启会话保持后，保持的最大时长<br />.spec.sessionAffinityConfig.clientIP.timeoutSeconds 默认设置为 30 秒 | 30 秒 |

| 注解 |【类型】选填<br />【含义】为服务添加注解 ||

### ==== “创建 LoadBalancer 服务”

点选 \_\_负载均衡 (LoadBalancer) \_\_，这是指使用云提供商的负载均衡器向外部暴露服务。

外部负载均衡器可以将流量路由到自动创建的 \_\_NodePort\_\_ 服务和 \_\_ClusterIP\_\_ 服务上。

| 参数 | 说明 | 举例值 |

| --- | :-- | :--- |

| 访问类型 |【类型】必填<br />【含义】指定 Pod 服务发现的方式 | LoadBalancer |

| 服务名称 |【类型】必填<br />【含义】输入新建服务的名称。<br />【注意】请输入 4 到 63 个字符的字符串，可以包含小写英文字母、数字和中划线 (-)，并以小写英文字母开头，小写英文字母或数字结尾。| Svc-01 |

| 命名空间 |【类型】必填<br />【含义】选择新建服务所在的命名空间。关于命名空间更多信息请参考[命名空间概述](./namespaces/createns.md)。<br />【注意】请输入 4 到 63 个字符的字符串，可以包含小写英文字母、数字和中划线 (-)，并以小写英文字母开头，小写英文字母或数字结尾。| default |

| 外部流量策略 |【类型】必填<br />【含义】设置外部流量策略。<br />\*\*Cluster\*\*：流量可以转发到集群中所有节点上的 Pod。<br />\*\*Local\*\*：流量只发给本节点上的 Pod。<br />【注意】请输入 4 到 63 个字符的字符串，可以包含小写英文字母、数字和中划线 (-)，并以小写英文字母开头，小写英文字母或数字结尾。| Cluster |

| 标签选择器 |【类型】必填<br />【含义】添加标签，Service 根据标签选择 Pod，填写后点击 \*\*添加\*\*。也可以引用已有工作负载的标签，点击 \_\_引用负载标签\_\_，在弹出的窗口中选择负载，系统会默认将所选的负载标签作为选择器。||

| 负载均衡类型 |【类型】必填<br />【含义】使用的负载均衡类型，当前支持 MetalLB 和其他。| MetalLB |

| MetalLB IP 池 |【类型】必填<br />【含义】选择的负载均衡类型为 MetalLB 时，LoadBalancer Service 默认会从这个池中分配 IP 地址，并且通过 APR 宣告这个池中的所有 IP 地址，详情请参考[安装 MetalLB](../../network/modules/metallb/install.md) ||

| 负载均衡地址 |【类型】必填<br />【含义】<br />1. 如使用的是公有云 CloudProvider，此处填写的为云厂商提供的负载均衡地址；<br />2. 如果上述负载均衡类型选择为 MetalLB，默认从上述 IP 池中获取 IP，如果不填则自动获取。| 自动获取 |

| 端口配置 |【类型】必填<br />【含义】为服务添加协议端口，需要先选择端口协议类型，目前支持 TCP、UDP 两种传输协议。<br />\*\*端口名称\*\*：输入自定义的端口的名称。<br />\*\*服务端口 (port)\*\*：Pod 对外提供服务的访问端口。默认为了方便起见，服务端口被设置为与容器端口字段相同的值。<br />\*\*容器端口 (targetport)\*\*：工作负载实际监听的容器端口。<br />\*\*节点端口 (nodeport)\*\*：节点的端口，接收来自 ClusterIP 传输的流量。用来做外部流量访问的入口。||

| 注解 |【类型】选填<br />【含义】为服务添加注解 ||

#### ==== “创建 ExternalName 服务”

点选 \_\_外部服务 (ExternalName) \_\_，这是指通过将服务映射到外部域名来暴露服务。

选择此项的服务不会创建典型的 ClusterIP 或 NodePort，而是通过 DNS 名称解析将请求重定向到外部的服务地址。

| 参数 | 说明 | 举例值 |

| --- | --- | --- |

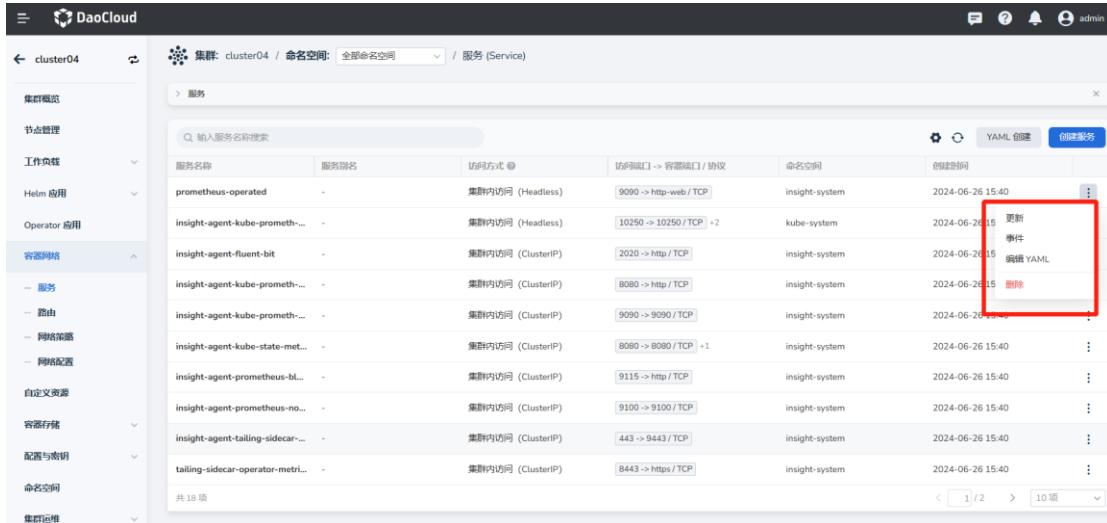
| 访问类型 |【类型】必填<br />【含义】指定 Pod 服务发现的方式，这里选择外部服务 (ExternalName)。| ExternalName |

| 服务名称 |【类型】必填<br />【含义】输入新建服务的名称。<br />【注意】请输入 4 到 63 个字符的字符串，可以包含小写英文字母、数字和中划线 (-)，并以小写英文字母开头，小写英文字母或数字结尾。| Svc-01 |

| 命名空间 |【类型】必填<br />【含义】选择新建服务所在的命名空间。关于命名空间更多信息请参考[命名空间概述](./namespaces/createns.md)。<br />【注意】请输入 4

到 63 个字符的字符串，可以包含小写英文字母、数字和中划线 (-)，并以小写英文字母开头，小写英文字母或数字结尾。| default |  
| 域名 |【类型】必填 | |

4. 配置完所有参数后，点击 **确定** 按钮，自动返回服务列表。在列表右侧，点击 ，可以修改或删除所选服务。



服务名称	服务别名	访问方式	访问端口 -> 容器端口 / 协议	命名空间	创建时间	操作
prometheus-operated	-	集群内访问 (Headless)	9090 -> http-web / TCP	insight-system	2024-06-26 15:40	 
insight-agent-kube-prometh...	-	集群内访问 (Headless)	10250 -> 10250 / TCP +2	kube-system	2024-06-26 15:15	 
insight-agent-fluent-bit	-	集群内访问 (ClusterIP)	2020 -> http / TCP	insight-system	2024-06-26 15:15	 
insight-agent-kube-prometh...	-	集群内访问 (ClusterIP)	8080 -> http / TCP	insight-system	2024-06-26 15:15	 
insight-agent-kube-prometh...	-	集群内访问 (ClusterIP)	9090 -> 9090 / TCP	insight-system	2024-06-26 15:40	 
insight-agent-kube-state-met...	-	集群内访问 (ClusterIP)	8080 -> 8080 / TCP +1	insight-system	2024-06-26 15:40	 
insight-agent-prometheus-bl...	-	集群内访问 (ClusterIP)	9115 -> http / TCP	insight-system	2024-06-26 15:40	 
insight-agent-prometheus-no...	-	集群内访问 (ClusterIP)	9100 -> 9100 / TCP	insight-system	2024-06-26 15:40	 
insight-agent-tailing-sidecar...	-	集群内访问 (ClusterIP)	443 -> 9443 / TCP	insight-system	2024-06-26 15:40	 
tailing-sidecar-operator-metri...	-	集群内访问 (ClusterIP)	8443 -> https / TCP	insight-system	2024-06-26 15:40	 

## 服务列表

## YAML 示例

```

kind: Service
apiVersion: v1
metadata:
 name: nvidia-dcgm-exporter
 namespace: gpu-operator
 uid: 7e412db9-2d23-4599-b48c-91e434aebef1
 resourceVersion: '408861'
 creationTimestamp: '2024-12-09T09:11:41Z'
 labels:
 app: nvidia-dcgm-exporter
 annotations:
 prometheus.io/scrape: 'true'
 ownerReferences:
 - apiVersion: nvidia.com/v1
 kind: ClusterPolicy
 name: cluster-policy
 uid: 59e6c966-abb9-45be-b13e-e51e31e7e55b
 controller: true
 blockOwnerDeletion: true

```

```

spec:
 ports:
 - name: gpu-metrics
 protocol: TCP
 port: 9400
 targetPort: 9400
 selector:
 app: nvidia-dcgm-exporter
 clusterIP: 10.233.29.230
 clusterIPs:
 - 10.233.29.230
 type: ClusterIP
 sessionAffinity: None
 ipFamilies:
 - IPv4
 ipFamilyPolicy: SingleStack
 internalTrafficPolicy: Cluster
 status:
 loadBalancer: {}

```

## 创建路由 ( Ingress )

在 Kubernetes 集群中，[Ingress](#) 公开从集群外部到集群内服务的 HTTP 和 HTTPS 路由。

流量路由由 Ingress 资源上定义的规则控制。下面是一个将所有流量都发送到同一 Service 的简单 Ingress 示例：

```

ingress-diagram
ingress-diagram

Ingress 是对集群中服务的外部访问进行管理的 API 对象，典型的访问方式是 HTTP。
Ingress 可以提供负载均衡、SSL 终结和基于名称的虚拟托管。

```

## 前提条件

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes](#)，且能够访问集群的 UI 界面。

- 已完成一个[命名空间的创建](#)、[用户的创建](#)，并将用户授权为[NS\\_Editor 角色](#)，详情可参考[命名空间授权](#)。
- 已经完成[Ingress 实例的创建](#)，已[部署应用工作负载](#)，并且已[创建对应 Service](#)
- 单个实例中有多个容器时，请确保容器使用的端口不冲突，否则部署会失效。

## 创建路由

1. 以 **NS Editor** 用户成功登录后，点击左上角的 **集群列表** 进入 **集群列表** 页面。在集群列表中，点击一个集群名称。

集群列表

集群列表

2. 在左侧导航栏中，点击 **容器网络 -> 路由** 进入服务列表，点击右上角 **创建路由** 按钮。

服务与路由

服务与路由

!!! note

也可以通过 [\\_YAML 创建](#) 一个路由。

3. 打开 **创建路由** 页面，进行配置。可选择两种协议类型，参考以下两个参数表进行配置。

## 创建 HTTP 协议路由

输入如下参数：

创建路由

创建路由

- **路由名称**：必填，输入新建路由的名称。
- **命名空间**：必填，选择新建服务所在的命名空间。关于命名空间更多信息请参考[命名空间概述](#)。
- **设置路由规则**：
  - **域名**：必填，使用域名对外提供访问服务。默认为集群的域名。
  - **协议**：必填，指授权入站到达集群服务的协议，支持 HTTP（不需要身份认证）或 HTTPS（需需要配置身份认证）协议。这里选择 HTTP 协议的路由。
  - **转发策略**：选填，指定 Ingress 的访问策略
  - **路径**：指定服务访问的 URL 路径，默认为根路径
  - **目标服务**：进行路由的服务名称
  - **目标服务端口**：服务对外暴露的端口
- **负载均衡器类型**：必填，[Ingress 实例的使用范围](#)
  - **平台级负载均衡器**：同一个集群内，共享同一个 Ingress 实例，其中 Pod 都可以接收到由该负载均衡分发的请求
  - **租户级负载均衡器**：租户负载均衡器，Ingress 实例独属于当前命名空，或者独属于某一工作空间，并且设置的工作空间中包含当前命名空间，其中 Pod 都可以接收到由该负载均衡分发的请求
- **Ingress Class**：选填，选择对应的 Ingress 实例，选择后将流量导入到指定的 Ingress 实例。
  - 为 None 时使用默认的 DefaultClass，请在创建 Ingress 实例时设置 DefaultClass，更多信息请参考 [Ingress Class](#)

- 若选择其他实例（如 `nginx`），则会出现高级配置，可设置 **会话保持**、**路径重写**、**重定向** 和 **流量分发**。
- **会话保持**：选填，会话保持分为 **三种类型**：**L4 源地址哈希**、**Cookie Key**、**L7 Header Name**，开启后根据对应规则进行会话保持。
  - **L4 源地址哈希**：开启后默认在 Annotation 中加入如下标签：  
`nginx.ingress.kubernetes.io/upstream-hash-by: "$binary_remote_addr"`
  - **Cookie Key**：开启后来自特定客户端的连接将传递至相同 Pod，开启后默认在 Annotation 中增加如下参数：  
`nginx.ingress.kubernetes.io/affinity: "cookie"`。  
`nginx.ingress.kubernetes.io/affinity-mode: persistent`
  - **L7 Header Name**：开启后默认在 Annotation 中加入如下标签：  
`nginx.ingress.kubernetes.io/upstream-hash-by: "$http_x_forwarded_for"`
- **路径重写**：选填，**rewrite-target**，某些场景中后端服务暴露的 URL 与 Ingress 规则中指定的路径不同，如果不进行 URL 重写配置，访问会出现错误。
- **重定向**：选填，**permanent-redirect**，永久重定向，输入重写路径后，访问路径重定向至设置的地址。
- **流量分发**：选填，**开启后并设置后**，根据设定条件进行流量分发。
  - **基于权重**：设定权重后，在创建的 Ingress 添加如下 Annotation：  
`nginx.ingress.kubernetes.io/canary-weight: "10"`
  - **基于 Cookie**：设定 Cookie 规则后，流量根据设定的 Cookie 条件进行流量分发
  - **基于 Header**：设定 Header 规则后，流量根据设定的 Header 条件进行流量分发
- **标签**：选填，为路由添加标签
- **注解**：选填，为路由添加注解

## 创建 HTTPS 协议路由

输入如下参数： 创建路由

!!! note

注意：与 HTTP 协议 设置路由规则 不同，增加密钥选择证书，其他基本一致。

- **协议**：必填指授权入站到达集群服务的协议，支持 HTTP（不需要身份认证）或 HTTPS（需要配置身份认证）协议。这里选择 HTTPS 协议的路由。
- **密钥**：必填，Https TLS 证书，[创建秘钥](#)。

## 完成路由创建

配置完所有参数后，点击 **确定** 按钮，自动返回路由列表。在列表右侧，点击 ，可以修改或删除所选路由。

### 路由列表

#### 路由列表

## 网络策略

网络策略（NetworkPolicy）可以在 IP 地址或端口层面（OSI 第 3 层或第 4 层）控制网络流量。容器管理模块目前支持创建基于 Pod 或命名空间的网络策略，支持通过标签选择器来设定哪些流量可以进入或离开带有特定标签的 Pod。

有关网络策略的更多详情，可参考 Kubernetes 官方文档[网络策略](#)。

## 创建网络策略

目前支持通过 YAML 和表单两种方式创建网络策略，这两种方式各有优劣，可以满足不同

用户的使用需求。

通过 YAML 创建步骤更少、更高效，但门槛要求较高，需要熟悉网络策略的 YAML 文件配置。

通过表单创建更直观更简单，根据提示填写对应的值即可，但步骤更加繁琐。

## YAML 创建

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 容器网络 -> 网络策略 -> YAML 创建。

路径

路径

2. 在弹框中输入或粘贴事先准备好的 YAML 文件，然后在弹框底部点击 确定。

yaml  
yaml

## 表单创建

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 容器网络 -> 网络策略 -> 创建策略。

路径

路径

2. 填写基本信息。

名称和命名空间在创建之后不可更改。

基本信息

基本信息

### 3. 填写策略配置。

策略配置分为入流量策略和出流量策略。如果源 Pod 想要成功连接到目标 Pod，源 Pod 的出流量策略和目标 Pod 的入流量策略都需要允许连接。如果任何一方不允许连接，都会导致连接失败。

- 入流量策略：点击  开始配置策略，支持配置多条策略。多条网络策略的效果相互叠加，只有同时满足所有网络策略，才能成功建立连接。

ingress  
ingress

- 出流量策略

egress  
egress

## 查看网络策略

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 容器网络 -> 网络策略，  
点击网络策略的名称。

路径

路径

2. 查看该策略的基本配置、关联实例信息、入流量策略、出流量策略。

详情

详情

**!!! info**

在关联实例页签下，支持查看实例监控、日志、容器列表、YAML 文件、事件等。

![查看实例信息](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/networkpolicy09.png)

## 更新网络策略

有两种途径可以更新网络策略。支持通过表单或 YAML 文件更新网络策略。

- 在网络策略列表页面，找到需要更新的策略，在右侧的操作栏下选择 **更新** 即可通过

表单更新，选择 **编辑 YAML** 即可通过 YAML 更新。

更新

更新

- 点击网络策略的名称，进入网络策略的详情页面后，在页面右上角选择 **更新** 即可通

过表单更新，选择 **编辑 YAML** 即可通过 YAML 更新。

更新

更新

## 删除网络策略

有两种途径可以删除网络策略。支持通过表单或 YAML 文件更新网络策略。

- 在网络策略列表页面，找到需要更新的策略，在右侧的操作栏下选择 **更新** 即可通过

表单更新，选择 **编辑 YAML** 即可通过 YAML 删除。

删除

删除

- 点击网络策略的名称，进入网络策略的详情页面后，在页面右上角选择 **更新** 即可通

过表单更新，选择 **编辑 YAML** 即可通过 YAML 删除。

删除

删除

# 创建自定义资源 (CRD)

在 Kubernetes 中一切对象都被抽象为资源，如 Pod、Deployment、Service、Volume 等是 Kubernetes 提供的默认资源，这为我们的日常运维和管理工作提供了重要支撑，但是在一些特殊的场景中，现有的预置资源并不能满足业务的需要，因此我们希望去扩展 Kubernetes API 的能力，自定义资源（CustomResourceDefinition, CRD）正是基于这样的需求应运而生。

容器管理模块支持对自定义资源的界面化管理，主要功能如下：

- 获取集群下自定义资源列表和详细信息
- 基于 YAML 创建自定义资源
- 基于 YAML 创建自定义资源示例 CR ( Custom Resource )
- 删除自定义资源

## 前提条件

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes](#)，且能够访问集群的 UI 界面
- 已完成一个[命名空间的创建、用户的创建](#)，并将用户授权为 [Cluster Admin](#) 角色，详情可参考[集群和命名空间授权](#)

## 通过 YAML 创建自定义资源

1. 点击一个集群名称，进入 [集群详情](#)。

集群列表

集群别名	集群角色	被纳管	创建时间	CPU 使用率	内存使用率
cluster202	工作集群	kpanda-global-cluster	2024-04-12 16:50	正常	1 / 1 正常/节点总数
cluster204	工作集群	binge-cluster 已删除	2024-03-28 21:25	数据同步中...	正常/节点总数

### 集群详情

2. 在左侧导航栏，点击 **自定义资源**，点击右上角 **YAML 创建** 按钮。

集群: cluster202 / 自定义资源

名称	CRD 范围	API 组	API 版本	创建时间
helmaps.helm.kpanda.io	命名空间	helm.kpanda.io	v1alpha1	2024-06-25 22:29
nodeovercommitconfigs.overcommit.katalyst.kubewharf.io	集群	overcommit.katalyst.kubewharf.io	v1alpha1	2024-05-27 22:30
customnoderesources.node.katalyst.kubewharf.io	集群	node.katalyst.kubewharf.io	v1alpha1	2024-05-27 22:30
cloudproxies.cloudshell.cloudtty.io	集群	cloudshell.cloudtty.io	v1alpha1	2024-05-06 16:41
cloudshells.cloudshell.cloudtty.io	命名空间	cloudshell.cloudtty.io	v1alpha1	2024-05-06 14:54
instrumentations.opentelemetry.io	命名空间	opentelemetry.io	v1alpha1	2024-04-15 16:45
opentelemetrycollectors.opentelemetry.io	命名空间	opentelemetry.io	v1alpha1	2024-04-15 16:45
thanosrulers.monitoring.coreos.com	命名空间	monitoring.coreos.com	v1	2024-04-15 16:45
prometheuses.monitoring.coreos.com	命名空间	monitoring.coreos.com	v1	2024-04-15 16:45
servicemonitors.monitoring.coreos.com	命名空间	monitoring.coreos.com	v1	2024-04-15 16:45

点击创建按钮

3. 在 **YAML 创建** 页面中，填写 **YAML** 语句后，点击 **确定**。

填写 yaml

填写 yaml

4. 返回自定义资源列表页，即可查看刚刚创建的名为 crontabs.stable.example.com 的自

## 定义资源。

The screenshot shows the 'Custom Resources' section of the DaoCloud Enterprise 5.0 interface. The left sidebar includes sections like '集群概览', '节点管理', '工作负载', 'Helm 应用', 'Operator 应用', '容器网络', '自定义资源' (which is currently selected), '容器存储', '配置与密钥', '命名空间', and '集群运维'. The main area displays a table of custom resources. One row, 'crontabs.stable.example.com', is highlighted with a red box. The table columns include '名称' (Name), 'CRD 范围' (CRD Scope), 'API 组' (API Group), 'API 版本' (API Version), and '创建时间' (Created Time). A success message at the top right says '创建自定义资源 crontabs.stable.example.com 任务下发成功, 请稍后刷新自定义资源列表查看。' (Custom resource creation task completed successfully, please refresh the custom resource list later).

查看

### 自定义资源示例：

```
yaml title="CRD example" apiVersion: apiextensions.k8s.io/v1 kind: CustomResourceDefinition
metadata: name: crontabs.stable.example.com spec: group: stable.example.com
versions: - name: v1 served: true storage: true schema:
openAPIV3Schema: type: object properties: spec:
type: object properties: cronSpec:
type: string image: type: string
replicas: type: integer scope: Namespaced names: plural:
crontabs singular: crontab kind: CronTab shortNames: - ct
```

### 通过 YAML 创建自定义资源示例

1. 点击一个集群名称，进入 **集群详情**。

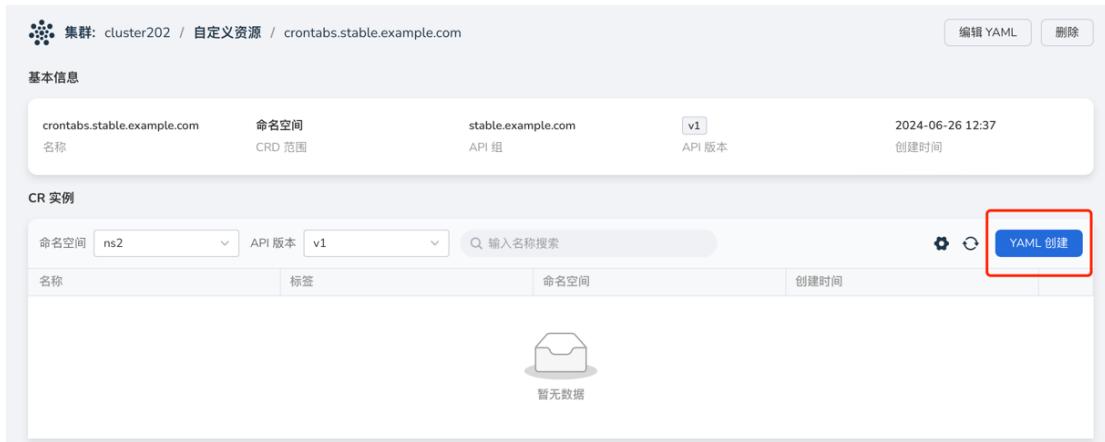
### 集群详情

2. 在左侧导航栏，点击 **自定义资源**，进入自定义资源列表页面。

点击创建按钮

3. 点击名为 crontabs.stable.example.com 的自定义资源，进入详情，点击右上角 YAML

**创建** 按钮。



点击创建按钮

4. 在 **YAML 创建** 页面中，填写 YAML 语句后，点击 **确定**。

填写 yaml

填写 yaml

5. 返回 crontabs.stable.example.com 的详情页面，即可查看刚刚创建的名为

**my-new-cron-object** 的自定义资源。

**CR 示例：**

```
yaml title="CR example" apiVersion: "stable.example.com/v1" kind: CronTab metadata: name: my-new-cron-object spec: cronSpec: "* * * * */5" image: my-awesome-cron-image
```

## 数据卷声明(PVC)

持久卷声明 ( PersistentVolumeClaim , PVC ) 表达的是用户对存储的请求。PVC 消耗 PV 资源，申领使用特定大小、特定访问模式的数据卷，例如要求 PV 卷以 `ReadWriteOnce`、`ReadOnlyMany` 或 `ReadWriteMany` 等模式来挂载。

## 创建数据卷声明

目前支持通过 YAML 和表单两种方式创建数据卷声明，这两种方式各有优劣，可以满足不

同用户的使用需求。

- 通过 YAML 创建步骤更少、更高效，但门槛要求较高，需要熟悉数据卷声明的 YAML 文件配置。
- 通过表单创建更直观更简单，根据提示填写对应的值即可，但步骤更加繁琐。

## YAML 创建

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 **容器存储 -> 数据卷声明 (PVC) -> YAML 创建**。

路径

路径

2. 在弹框中输入或粘贴事先准备好的 YAML 文件，然后在弹框底部点击 **确定**。

支持从本地导入 YAML 文件或将填写好的文件下载保存到本地。

yaml  
yaml

## 表单创建

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 **容器存储 -> 数据卷声明 (PVC) -> 创建数据卷声明 (PVC)**。

路径

路径

2. 填写基本信息。

- 数据卷声明的名称、命名空间、创建方式、数据卷、容量、访问模式在创建之后不可更改。

- 创建方式：在已有的存储池或者数据卷中动态创建新的数据卷声明，或者基于数据卷声明的快照创建新的数据卷声明。  
基于快照创建时无法修改数据卷声明的容量，可以在创建完成后再进行修改。
  - 选择创建方式之后，在下拉列表中选择想要使用的存储池/数据卷/快照。
  - 访问模式：
    - `ReadWriteOnce`，数据卷声明可以被一个节点以读写方式挂载。
    - `ReadWriteMany`，数据卷声明可以被多个节点以读写方式挂载。
    - `ReadOnlyMany`，数据卷声明可以被多个节点以只读方式挂载。
    - `ReadWriteOncePod`，数据卷声明可以被单个 Pod 以读写方式挂载。

#### 基本信息

##### 基本信息

## 查看数据卷声明

在集群列表中点击目标集群的名称，然后在左侧导航栏点击 **容器存储 -> 数据卷声明 (PVC)**。

- 该页面可以查看当前集群中的所有数据卷声明，以及各个数据卷声明的状态、容量、命名空间等信息。
- 支持按照数据卷声明的名称、状态、命名空间、创建时间进行顺序或逆序排序。

#### 详情

##### 详情

- 点击数据卷声明的名称，可以查看该数据卷声明的基本配置、存储池信息、标签、注

解等信息。

详情

详情

## 扩容数据卷声明

1. 在左侧导航栏点击 **容器存储** -> **数据卷声明(PVC)**，找到想要调整容量的数据卷声明。

扩容

扩容

2. 点击数据卷声明的名称，然后在页面右上角点击操作按钮选择 **扩容**。

扩容

扩容

3. 输入目标容量，然后点击 **确定**。

克隆

克隆

## 克隆数据卷声明

通过克隆数据卷声明，可以基于被克隆数据卷声明的配置，重新创建一个新的数据卷声明。

1. 进入克隆页面

- 在数据卷声明列表页面，找到需要克隆的数据卷声明，在右侧的操作栏下选择 **克隆**。

也可以点击数据卷声明的名称，在详情页面的右上角点击操作按钮选择 **克**

**隆**。

克隆

克隆

2. 直接使用原配置，或者按需进行修改，然后在页面底部点击 **确定**。

克隆

克隆

## 更新数据卷声明

有两种途径可以更新数据卷声明。支持通过表单或 YAML 文件更新数据卷声明。

!!! note

仅支持更新数据卷声明的别名、标签和注解。

- 在数据卷列表页面，找到需要更新的数据卷声明，在右侧的操作栏下选择 **更新** 即可

通过表单更新，选择 **编辑 YAML** 即可通过 YAML 更新。

更新

更新

- 点击数据卷声明的名称，进入数据卷声明的详情页面后，在页面右上角选择 **更新** 即

可通过表单更新，选择 **编辑 YAML** 即可通过 YAML 更新。

更新

更新

## 删除数据卷声明

在数据卷声明列表页面，找到需要删除的数据，在右侧的操作栏下选择 **删除**。

也可以点击数据卷声明的名称，在详情页面的右上角点击操作按钮选择 **删除**。

删除

删除

## 常见问题

1. 如果列表中没有可选的存储池或数据卷，可以[创建存储池](#)或[创建数据卷](#)。

2. 如果列表中没有可选的快照，可以进入数据卷声明的详情页，在右上角制作快照。

制作快照

制作快照

3. 如果数据卷声明所使用的存储池 (SC) 没有启用快照，则无法制作快照，页面不会显示“制作快照”选项。

4. 如果数据卷声明所使用的存储池 (SC) 没有开启扩容功能，则该数据卷不支持扩容，页面不会显示扩容选项。

开启快照

开启快照

## 数据卷(PV)

数据卷 ( PersistentVolume , PV ) 是集群中的一块存储，可由管理员事先制备，或使用存储类 ( Storage Class ) 来动态制备。PV 是集群资源，但拥有独立的生命周期，不会随着 Pod 进程结束而被删除。将 PV 挂载到工作负载可以实现工作负载的数据持久化。PV 中保存了可被 Pod 中容器访问的数据目录。

## 创建数据卷

目前支持通过 YAML 和表单两种方式创建数据卷，这两种方式各有优劣，可以满足不同用户的使用需求。

- 通过 YAML 创建步骤更少、更高效，但门槛要求较高，需要熟悉数据卷的 YAML 文件配置。
- 通过表单创建更直观更简单，根据提示填写对应的值即可，但步骤更加繁琐。

### YAML 创建

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 **容器存储 -> 数据卷(PV)** -> **YAML 创建**。

路径

路径

2. 在弹框中输入或粘贴事先准备好的 YAML 文件，然后在弹框底部点击 **确定**。

支持从本地导入 YAML 文件或将填写好的文件下载保存到本地。

yaml  
yaml

### 表单创建

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 **容器存储 -> 数据卷(PV)** -> **创建数据卷(PV)**。

路径

路径

2. 填写基本信息。

- 数据卷名称、数据卷类型、挂载路径、卷模式、节点亲和性在创建之后不可更改。
  - 数据卷类型：有关卷类型的详细介绍，可参考 [Kubernetes 官方文档卷](#)。
    - Local：将 Node 节点的本地存储包装成 PVC 接口，容器直接使用 PVC 而无需关注底层的存储类型。Local 卷不支持动态配置数据卷，但支持配置节点亲和性，可以限制能从哪些节点上访问该数据卷。
    - HostPath：使用 Node 节点的文件系统上的文件或目录作为数据卷，不支持基于节点亲和性的 Pod 调度。
  - 挂载路径：将数据卷挂载到容器中的某个具体目录下。
  - 访问模式：
    - ReadWriteOnce：数据卷可以被一个节点以读写方式挂载。
    - ReadWriteMany：数据卷可以被多个节点以读写方式挂载。
    - ReadOnlyMany：数据卷可以被多个节点以只读方式挂载。
    - ReadWriteOncePod：数据卷可以被单个 Pod 以读写方式挂载。
  - 回收策略：
    - Retain：不删除 PV，仅将其状态变为 **released**，需要用户手动回收。有关如何手动回收，可参考[持久卷](#)。
    - Recycle：保留 PV 但清空其中的数据，执行基本的擦除操作（`rm -rf /thevolume/*`）。
    - Delete：删除 PV 时及其中的数据。
  - 卷模式：
    - 文件系统：数据卷将被 Pod 挂载到某个目录。如果数据卷的存储来

自某块设备而该设备目前为空，第一次挂载卷之前会在设备上创建

文件系统。

- 块：将数据卷作为原始块设备来使用。这类卷以块设备的方式交给 Pod 使用，其上没有任何文件系统，可以让 Pod 更快地访问数据卷。

- 节点亲和性：

基本信息

基本信息

## 查看数据卷

在集群列表中点击目标集群的名称，然后在左侧导航栏点击 容器存储 -> 数据卷(PV)。

- 该页面可以查看当前集群中的所有数据卷，以及各个数据卷的状态、容量、命名空间等信息。
- 支持按照数据卷的名称、状态、命名空间、创建时间进行顺序或逆序排序。

详情

详情

- 点击数据卷的名称，可以查看该数据卷的基本配置、存储池信息、标签、注解等信息。

详情

详情

## 克隆数据卷

通过克隆数据卷，可以基于被克隆数据卷的配置，重新创建一个新的数据卷。

### 1. 进入克隆页面

- 在数据卷列表页面，找到需要克隆的数据卷，在右侧的操作栏下选择 **克隆**。

也可以点击数据卷的名称，在详情页面的右上角点击操作按钮选择 **克隆**。

克隆

克隆

### 2. 直接使用原配置，或者按需进行修改，然后在页面底部点击 **确定**。

## 更新数据卷

有两种途径可以更新数据卷。支持通过表单或 YAML 文件更新数据卷。

#### !!! note

仅支持更新数据卷的别名、容量、访问模式、回收策略、标签和注解。

- 在数据卷列表页面，找到需要更新的数据卷，在右侧的操作栏下选择 **更新** 即可通过表单更新，选择 **编辑 YAML** 即可通过 YAML 更新。

更新

更新

- 点击数据卷的名称，进入数据卷的详情页面后，在页面右上角选择 **更新** 即可通过表单更新，选择 **编辑 YAML** 即可通过 YAML 更新。

更新

更新

## 删除数据卷

在数据卷列表页面，找到需要删除的数据，在右侧的操作栏下选择 **删除**。

也可以点击数据卷的名称，在详情页面的右上角点击操作按钮选择 **删除**。

删除

删除

## 存储池(SC)

存储池指将许多物理磁盘组成一个大型存储资源池，本平台支持接入各类存储厂商后创建块存储池、本地存储池、自定义存储池，然后为工作负载动态配置数据卷。

## 创建存储池(SC)

目前支持通过 YAML 和表单两种方式创建存储池，这两种方式各有优劣，可以满足不同用户的使用需求。

- 通过 YAML 创建步骤更少、更高效，但门槛要求较高，需要熟悉存储池的 YAML 文件配置。
- 通过表单创建更直观更简单，根据提示填写对应的值即可，但步骤更加繁琐。

### YAML 创建

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 **容器存储 -> 存储池(SC) -> YAML 创建**。

存储池 (SC) 名称	存储池 (SC) 别名	CSI 存储驱动	回收策略	创建时间
rook-cephfs	-	rook-ceph.cephfs.csi.ceph.com	删除数据 (Delete)	2024-07-04 09:03
rook-ceph-block	-	rook-ceph.rbd.csi.ceph.com	删除数据 (Delete)	2024-07-04 08:53

## 路径

2. 在弹框中输入或粘贴事先准备好的 YAML 文件，然后在弹框底部点击 确定。

支持从本地导入 YAML 文件或将填写好的文件下载保存到本地。

yaml  
yaml

## 表单创建

1. 在集群列表中点击目标集群的名称，然后在左侧导航栏点击 容器存储 -> 存储池(SC) ->

创建存储池(SC)。

存储池 (SC) 名称	存储池 (SC) 别名	CSI 存储驱动	回收策略	创建时间
rook-cephfs	-	rook-ceph.cephfs.csi.ceph.com	删除数据 (Delete)	2024-07-04 09:03
rook-ceph-block	-	rook-ceph.rbd.csi.ceph.com	删除数据 (Delete)	2024-07-04 08:53

## 路径

2. 填写基本信息，然后在底部点击 **确定**。

### 自定义存储系统

- 存储池名称、驱动、回收策略在创建后不可修改。
- CSI 存储驱动：基于标准 Kubernetes 的容器存储接口插件，需遵守存储厂商规定的格式，例如 `rancher.io/local-path`。
  - 有关如何填写不同厂商提供的 CSI 驱动，可参考 Kubernetes 官方文档[存储类](#)。
- 回收策略：删除数据卷时，保留数据卷中的数据或者删除其中的数据。
- 快照/扩容：开启后，基于该存储池的数据卷/数据卷声明才能支持扩容和快照功能，但 **前提是底层使用的存储驱动支持快照和扩容功能**。

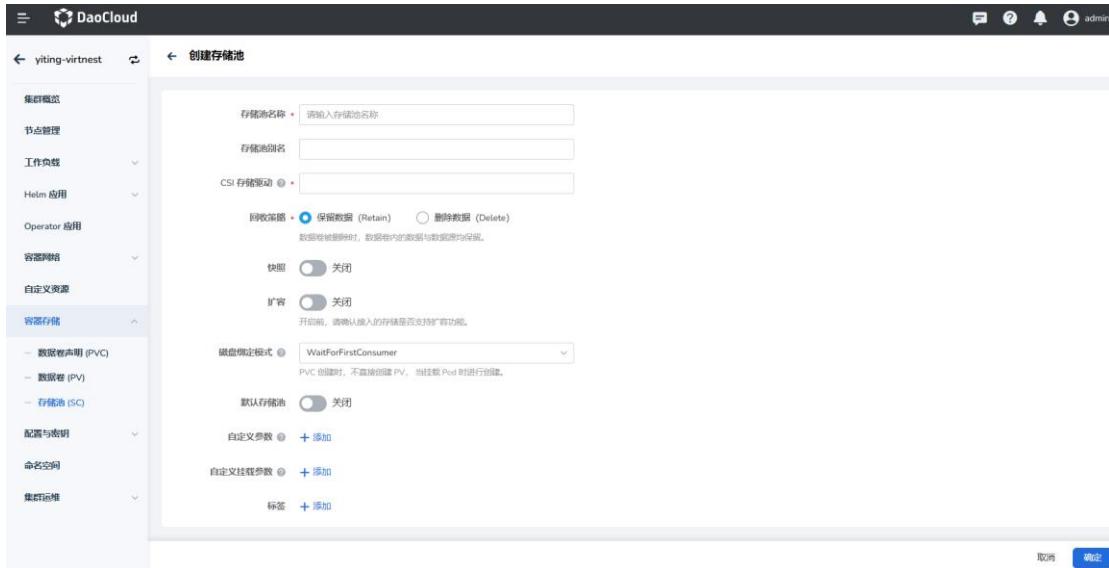
### HwameiStor 存储系统

- 存储池名称、驱动、回收策略在创建后不可修改。
- 存储系统：HwameiStor 存储系统。
- 存储类型：支持 LVM，裸磁盘类型
  - **LVM 类型**：HwameiStor 推荐使用此方式，可使用高可用数据卷，对应的 CSI 存储驱动为 `lvm.hwameistor.io`。
  - **裸磁盘数据卷**：适用于非高可用场景，无高可用能力，对应的 CSI 驱动为 `hdd.hwameistor.io`
- 高可用模式：使用高可用能力之前请确认 **DRBD** 组件 已安装。开启高可用模式后，可将数据卷副本数设置为 1 和 2。如需要可将数据卷副本从 1 Convert 成 1
- 回收策略：删除数据卷时，保留数据卷中的数据或者删除其中的数据。

- 快照/扩容：开启后，基于该存储池的数据卷/数据卷声明才能支持扩容和快照功能，但前提是底层使用的存储驱动支持快照和扩容功能。

**!!! note**

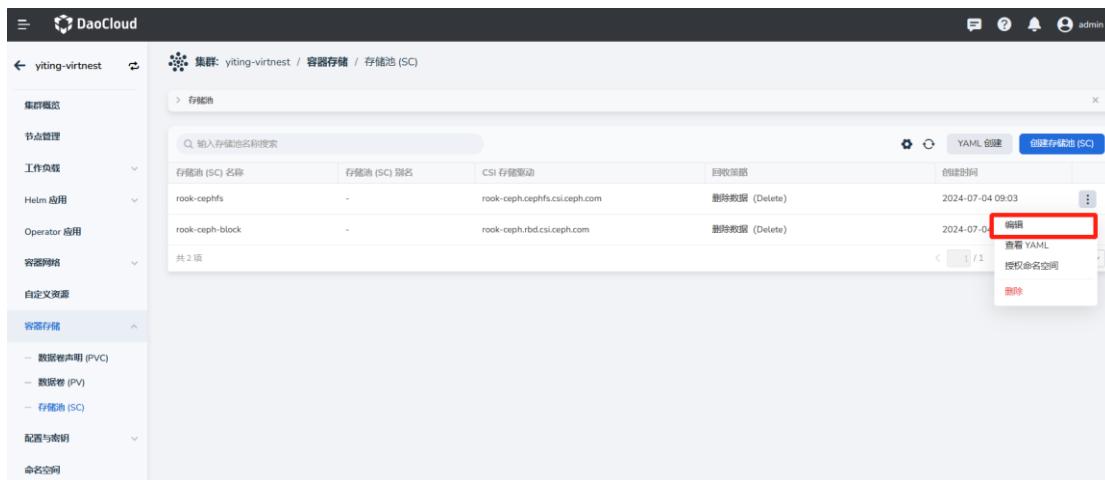
目前 HwameiStor xfs、ext4 两种文件系统，其中默认使用的是 xfs 文件系统，如果想要替换为 ext4，可以在自定义参数添加 `_csi.storage.k8s.io/fstype: ext4`



### 基本信息

## 更新存储池(SC)

在存储池列表页面，找到需要更新的存储池，在右侧的操作栏下选择 **编辑** 即可通过更新存储池。



### 更新

**!!! info**

选择 查看 YAML 可以查看该存储池的 YAML 文件，但不支持编辑。

## 删除存储池(SC)

在存储池列表页面，找到需要删除的存储池，在右侧的操作栏下选择 **删除**。

The screenshot shows the 'Storage Pools (SC)' section of the 'Container Storage' page. It lists two storage pools: 'rook-cephfs' and 'rook-ceph-block'. The 'rook-cephfs' row has a red box around its 'Delete' button in the operations column. The interface includes a sidebar with navigation links like 'Cluster Overview', 'Node Management', 'Workload', 'Helm Application', 'Operator Application', 'Container Network', 'Custom Resources', 'Container Registry', and 'Namespaces'. The 'Container Registry' section is expanded, showing 'Persistent Volumes (PV)', 'Persistent Volume Claims (PVC)', and 'Storage Pools (SC)'. The 'Storage Pools (SC)' link is also highlighted with a red box.

**删除**

## 共享存储池

DCE 5.0 容器管理模块支持将一个存储池共享给多个命名空间使用，以便提高资源利用效率。

1. 在存储池列表中找到需要共享的存储池，在右侧操作栏下点击 **授权命名空间**。

**授权**

**授权**

2. 点击 **自定义命名空间** 可以逐一选择需要将此存储池共享到哪些命名空间。

- 点击 **授权所有命名空间** 可以一次性将此存储池共享到当前集群下的所有命名空间。

- 在列表右侧的操作栏下方点击 **移除授权**，可以解除授权，停止将此存储池

共享到该命名空间。

授权

授权

## 创建配置项

配置项（ConfigMap）以键值对的形式存储非机密性数据，实现配置数据和应用代码相互解耦的效果。配置项可用作容器的环境变量、命令行参数或者存储卷中的配置文件。

!!! note

- 在配置项中保存的数据不可超过 1 MiB。如果需要存储体积更大的数据，建议挂载存储卷或者使用独立的数据库或者文件服务。
- 配置项不提供保密或者加密功能。如果要存储加密数据，建议使用[密钥](use-secret.md)，或者其他第三方工具来保证数据的私密性。

支持两种创建方式：

- 图形化表单创建
- YAML 创建

## 前提条件

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面
- 已完成一个[命名空间的创建](#)、[用户的创建](#)，并将用户授权为 [NS Editor 角色](#)，详情可参考[命名空间授权](#)。

## 图形化表单创建

1. 在 [集群列表](#) 页面点击某个集群的名称，进入 [集群详情](#)。

## 集群详情

集群详情

2. 在左侧导航栏，点击 **配置与密钥 -> 配置项**，点击右上角 **创建配置项** 按钮。

创建配置项

创建配置项

3. 在 **创建配置项** 页面中填写配置信息，点击 **确定**。

!!! note

点击 上传文件 可以从本地导入已有的文件，快速创建配置项。

创建配置项

创建配置项

4. 创建完成后在配置项右侧点击更多可以，**可以编辑 YAML、更新、导出、删除等操作**。

创建配置项

创建配置项

## YAML 创建

1. 在 **集群列表** 页面点击某个集群的名称，进入 **集群详情**。

集群详情

集群详情

2. 在左侧导航栏，点击 **配置与密钥 -> 配置项**，点击右上角 **YAML 创建** 按钮。

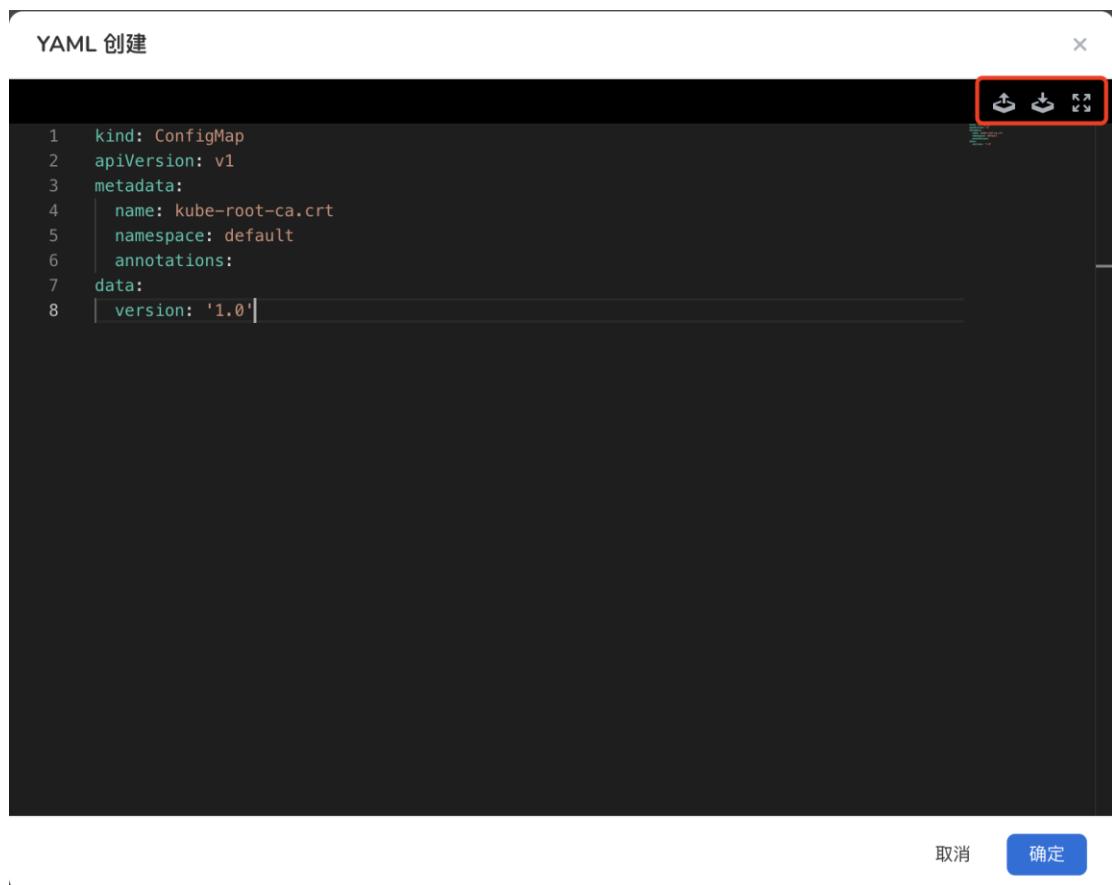
创建配置项

创建配置项

3. 填写或粘贴事先准备好的配置文件，然后在弹框右下角点击 **确定**。

!!! note

- 点击 导入 可以从本地导入已有的文件，快速创建配置项。
- 填写数据之后点击 下载 可以将配置文件保存在本地。



创建配置项

4. 创建完成后在配置项右侧点击更多可以，可以编辑 YAML、更新、导出、删除等操作。

创建配置项

创建配置项

## 配置项 YAML 示例

```
```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: kube-root-ca.crt
  namespace: default
  annotations:
```

```
data:  
  version: '1.0'  
...  
...
```

[下一步：使用配置项](#)

使用配置项

配置项（ConfigMap）是 Kubernetes 的一种 API 对象，用来将非机密性的数据保存到键值对中，可以存储其他对象所需要使用的配置。使用时，容器可以将其用作环境变量、命令行参数或者存储卷中的配置文件。通过使用配置项，能够将配置数据和应用程序代码分离开，为应用配置的修改提供更加灵活的途径。

!!! note

配置项并不提供保密或者加密功能。如果要存储的数据是机密的，请使用[密钥](use-secret.md)，或者使用其他第三方工具来保证数据的私密性，而不是用配置项。

此外在容器里使用配置项时，容器和配置项必须处于同一集群的命名空间中。

使用场景

您可以在 Pod 中使用配置项，有多种使用场景，主要包括：

- 使用配置项设置容器的环境变量
- 使用配置项设置容器的命令行参数
- 使用配置项作为容器的数据卷

设置容器的环境变量

您可以通过图形化界面或者终端命令行来使用配置项作为容器的环境变量。

!!! note

配置项导入是将配置项作为环境变量的值；配置项键值导入是将配置项中某一参数作为环境变量的值。

图形化界面操作

通过镜像创建工作负载时，可以在 环境变量 界面通过选择 配置项导入 或 配置项键值导

入 为容器设置环境变量。

1. 进入[镜像创建工作负载](#)页面中，在 容器配置 这一步中，选择 环境变量 配置，点击 添加环境变量 按钮。

添加环境变量

添加环境变量

2. 在环境变量类型处选择 配置项导入 或 配置项键值导入 。

- 当环境变量类型选择为 配置项导入 时，依次输入 变量名 、 前缀 名称、
配置项 的名称。
- 当环境变量类型选择为 配置项键值导入 时，依次输入 变量名 、 配置项
名称、 键 的名称。

命令行操作

您可以在创建工作负载时将配置项设置为环境变量，使用 valueFrom 参数引用 ConfigMap 中的 Key/Value。

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod-1
spec:
  containers:
    - name: test-container
      image: busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom: # (1)!
```

```

configMapKeyRef:
  name: kpanda-configmap # (2)!
  key: SPECIAL_LEVEL      # (3)!

restartPolicy: Never

```

1. 使用 **valueFrom** 来指定 env 引用配置项的 value 值
2. 引用的配置文件名称
3. 引用的配置项 key

设置容器的命令行参数

您可以使用配置项设置容器中的命令或者参数值，使用环境变量替换语法 **`$(VAR_NAME)`**

来进行。如下所示。

```

apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod-3
spec:
  containers:
    - name: test-container
      image: busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
  env:
    - name: SPECIAL_LEVEL_KEY
      valueFrom:
        configMapKeyRef:
          name: kpanda-configmap
          key: SPECIAL_LEVEL
    - name: SPECIAL_TYPE_KEY
      valueFrom:
        configMapKeyRef:
          name: kpanda-configmap
          key: SPECIAL_TYPE
  restartPolicy: Never

```

这个 Pod 运行后，输出如下内容。

Hello Kpanda

用作容器数据卷

您可以通过图形化界面或者终端命令行来使用配置项作为容器的环境变量。

图形化操作

在通过镜像创建工作负载时，您可以通过在 **数据存储** 界面选择存储类型为 **配置项**，将配置项作为容器的数据卷。

1. 进入[镜像创建工作负载](#)页面中，在**容器配置**这一步中，选择**数据存储**配置，在**节点路径映射**列表点击**添加**按钮。



添加环境变量

2. 在存储类型处选择**配置项**，并依次输入**容器路径**、**子路径**等信息。

命令行操作

要在单个 Pod 的存储卷中使用 ConfigMap。

下面是一个将 ConfigMap 以卷的形式进行挂载的 Pod 示例：

```
apiVersion: v1
```

```
kind: Pod
```

```

metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: redis
      volumeMounts:
        - name: foo
          mountPath: "/etc/foo"
          readOnly: true
  volumes:
    - name: foo
      configMap:
        name: myconfigmap

```

如果 Pod 中有多个容器，则每个容器都需要自己的 **volumeMounts** 块，但针对每个 ConfigMap，您只需要设置一个 **spec.volumes** 块。

!!! note

将配置项作为容器挂载的数据卷时，配置项只能作为只读文件进行读取。

创建密钥

密钥是一种用于存储和管理密码、OAuth 令牌、SSH、TLS 凭据等敏感信息的资源对象。使用密钥意味着您不需要在应用程序代码中包含敏感的机密数据。

密钥使用场景：

- 作为容器的环境变量使用，提供容器运行过程中所需的一些必要信息。
- 使用密钥作为 Pod 的数据卷。
- 在 kubelet 拉取容器镜像时作为镜像仓库的身份认证凭证。

支持两种创建方式：

- 图形化表单创建
- YAML 创建

前提条件

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面
- 已完成一个[命名空间的创建](#)、[用户的创建](#)，并将用户授权为[NS Editor](#)，详情可参考[集群和命名空间授权](#)。

图形化表单创建

1. 在 **集群列表** 页面点击某个集群的名称，进入 **集群详情**。

集群详情

集群详情

2. 在左侧导航栏，点击 **配置与密钥 -> 密钥**，点击右上角 **创建密钥** 按钮。

创建密钥

创建密钥

3. 在 **创建密钥** 页面中填写配置信息，点击 **确定**。

创建密钥

创建密钥

填写配置时需要注意：

- 密钥的名称在同一个命名空间中必须具有唯一性
- 密钥类型：
 - 默认 (**Opaque**)：Kubernetes 默认的密钥类型，支持用户定义的任意数据。
 - TLS (`kubernetes.io/tls`)：用于 TLS 客户端或者服务器端数据访问的凭

证。

- 镜像仓库信息 (kubernetes.io/dockerconfigjson) : 用于镜像仓库访问的凭证。
- 用户名和密码 (kubernetes.io/basic-auth) : 用于基本身份认证的凭证。
- 自定义 : 用户根据业务需要自定义的类型。
 - 密钥数据 : 密钥所存储的数据 , 不同数据需要填写的参数有所不同
 - 当密钥类型为默认 (Opaque) / 自定义 : 可以填入多个键值对数据。
 - 当密钥类型为 TLS (kubernetes.io/tls) : 需要填入证书凭证和私钥数据。证书是自签名或 CA 签名过的凭据 , 用来进行身份认证。证书请求是对签名的请求 , 需要使用私钥进行签名。
 - 当密钥类型为镜像仓库信息 (kubernetes.io/dockerconfigjson) : 需要填入私有镜像仓库的账号和密码。
 - 当密钥类型为用户名和密码 (kubernetes.io/basic-auth) : 需要指定用户名和密码。

YAML 创建

1. 在 **集群列表** 页面点击某个集群的名称 , 进入 **集群详情** 。

集群详情

集群详情

2. 在左侧导航栏 , 点击 **配置与密钥 -> 密钥** , 点击右上角 **YAML 创建** 按钮。

YAML 创建

YAML 创建

3. 在 **YAML 创建** 页面中填写 YAML 配置，点击 **确定**。

支持从本地导入 YAML 文件或将填写好的文件下载保存到本地。

```

apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
  type: kubernetes.io/basic-auth
stringData:
  username: admin
  password: t0p-Secret
  
```

The screenshot shows the 'Secret' section of the 'YAML 创建' dialog. The right side of the screen displays a list of existing secrets with their creation times. At the bottom right of the dialog, there are '取消' (Cancel) and '确定' (Confirm) buttons.

YAML 创建

密钥 YAML 示例

```

```yaml
apiVersion: v1
kind: Secret
metadata:
 name: secretdemo
type: Opaque
data:
 username: *****
 password: *****
```
  
```

[下一步：使用密钥](#)

使用密钥

密钥是一种用于存储和管理密码、OAuth 令牌、SSH、TLS 凭据等敏感信息的资源对象。使用密钥意味着您不需要在应用程序代码中包含敏感的机密数据。

使用场景

您可以在 Pod 中使用密钥，有多种使用场景，主要包括：

- 作为容器的环境变量使用，提供容器运行过程中所需的一些必要信息。
- 使用密钥作为 Pod 的数据卷。
- 在 kubelet 拉取容器镜像时用作镜像仓库的身份认证凭证使用。

使用密钥设置容器的环境变量

您可以通过图形化界面或者终端命令行来使用密钥作为容器的环境变量。

!!! note

密钥导入是将密钥作为环境变量的值；密钥键值导入是将密钥中某一参数作为环境变量的值。

图形界面操作

在通过镜像创建工作负载时，您可以在 环境变量 界面通过选择 **密钥导入** 或 **密钥键值导入** 为容器设置环境变量。

1. 进入[镜像创建工作负载](#)页面。

创建 deployment

创建 deployment

2. 在 容器配置 选择 环境变量 配置，点击 **添加环境变量** 按钮。

添加环境变量

添加环境变量

3. 在环境变量类型处选择 **密钥导入** 或 **密钥键值导入**。

密钥导入

密钥导入

- 当环境变量类型选择为 **密钥导入** 时，依次输入 **变量名**、**前缀**、**密钥**的名称。
- 当环境变量类型选择为 **密钥键值导入** 时，依次输入 **变量名**、**密钥**、**键**的名称。

命令行操作

如下例所示，您可以在创建工作负载时将密钥设置为环境变量，使用 **valueFrom** 参数引用

Secret 中的 Key/Value。

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
    - name: mycontainer
      image: redis
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
              optional: false # (1)!
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
```

```
key: password
optional: false # (2)!
```

1. 此值为默认值；意味着“mysecret”，必须存在且包含名为“username”的主键
2. 此值为默认值；意味着“mysecret”，必须存在且包含名为“password”的主键

使用密钥作为 Pod 的数据卷

图形界面操作

在通过镜像创建工作负载时，您可以通过在 **数据存储** 界面选择存储类型为 **密钥**，将密钥作为容器的数据卷。

1. 进入[镜像创建工作负载](#)页面。

创建 deployment

创建 deployment

2. 在 **容器配置** 选择 **数据存储** 配置，在 **节点路径映射** 列表点击 **添加** 按钮。



创建 deployment

3. 在存储类型处选择 **密钥**，并依次输入 **容器路径**、**子路径** 等信息。

命令行操作

下面是一个通过数据卷来挂载名为 `mysecret` 的 Secret 的 Pod 示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: redis
      volumeMounts:
        - name: foo
          mountPath: "/etc/foo"
          readOnly: true
  volumes:
    - name: foo
      secret:
        secretName: mysecret
        optional: false # (1)!
```

1. 默认设置，意味着“`mysecret`”必须已经存在

如果 Pod 中包含多个容器，则每个容器需要自己的 `volumeMounts` 块，不过针对每个 Secret 而言，只需要一份 `.spec.volumes` 设置。

在 kubelet 拉取容器镜像时用作镜像仓库的身份认证凭证

您可以通过图形化界面或者终端命令行来使用密钥作为镜像仓库身份认证凭证。

图形化操作

在通过镜像创建工作负载时，您可以通过在 **数据存储** 界面选择存储类型为 **密钥**，将密钥作为容器的数据卷。

1. 进入[镜像创建工作负载](#)页面。

创建 deployment

创建 deployment

2. 在第二步 容器配置 时选择 基本信息 配置，点击 选择镜像 按钮。

选择镜像

选择镜像

3. 在弹框的 镜像仓库 下拉选择私有镜像仓库名称。关于私有镜像密钥创建请查看[创建密钥](#)了解详情。

选择镜像

选择镜像

4. 输入私有仓库内的镜像名称，点击 确定，完成镜像选择。

!!! note

创建密钥时，需要确保输入正确的镜像仓库地址、用户名、密码并选择正确的镜像名称，否则将无法获取镜像仓库中的镜像。

configmap/secret 热加载

configmap/secret 热加载是指将 configmap/secret 作为数据卷挂载在容器中挂载时，当配置发生改变时，容器将自动读取 configmap/secret 更新后的配置，而无需重启 Pod。

操作步骤

1. 参考创建工作负载 - [容器配置](#)，配置容器数据存储，选择 Configmap 、 Configmap

Key 、 Secret 、 Secret Key 作为数据卷挂载至容器。

使用 config 作为数据卷

使用 config 作为数据卷

!!! note

使用子路径（SubPath）方式挂载的配置文件不支持热加载。

- 进入【配置与密钥】页面，进入配置项详情页面，在【关联资源】中找到对应的 container 资源，点击 **立即加载** 按钮，进入配置热加载页面。

使用 config 作为数据卷

使用 config 作为数据卷

!!! note

如果您的应用支持自动读取 configmap/secret 更新后的配置，则无需手动执行热加载操作。

- 在热加载配置弹窗中，输入进入容器内的 执行命令 并点击 确定 按钮，以重载配置。

例如，在 nginx 容器中，以 root 用户权限，执行 `nginx -s reload` 命令来重载配置。

使用 config 作为数据卷

使用 config 作为数据卷

- 在界面弹出的 web 终端中查看应用重载情况。

使用 config 作为数据卷

使用 config 作为数据卷

最近操作

在该页面可以查看最近的集群操作记录和 Helm 操作记录，以及各项操作的 YAML 文件和日志，也可以删除某一条记录。

| 操作名称 | 状态 | 操作 | 操作对象 | 创建时间 |
|------------------------------------|----|-----------|----------------|------------------|
| bin-cluster122-ops-pre-check-zjfk5 | 失败 | pre-check | bin-cluster122 | 2025-02-18 17:49 |
| bin-cluster122-ops-pre-check-ncz8g | 失败 | pre-check | bin-cluster122 | 2025-02-18 17:46 |
| bin-cluster122-ops-pre-check-6vxl | 失败 | pre-check | bin-cluster122 | 2025-02-18 17:43 |

操作记录

在集群操作列表中，通过右侧 ，可以查看 YAML 和日志，也可以执行删除。

设置 Helm 操作的保留条数

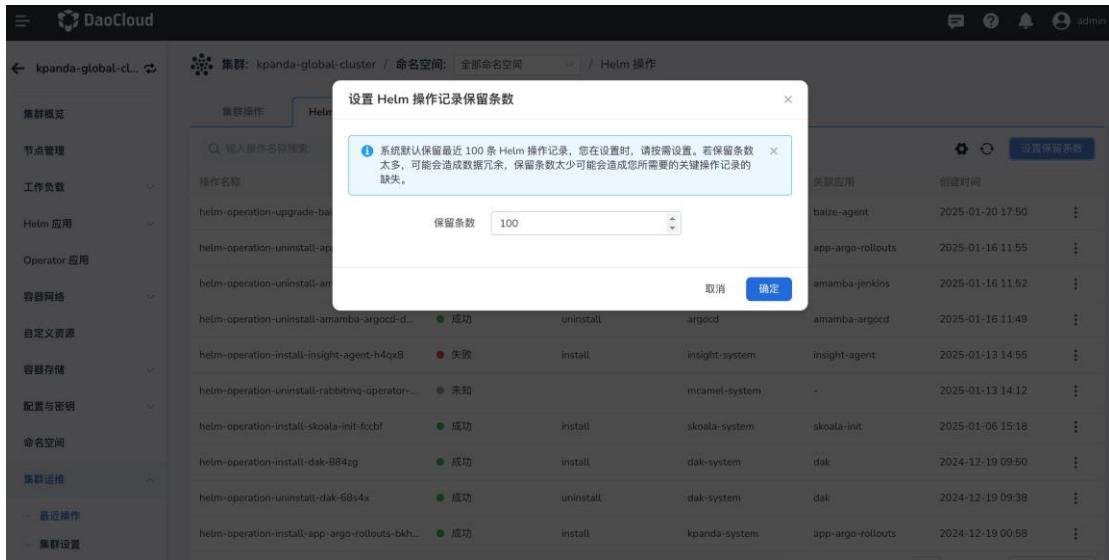
系统默认保留最近 100 条 Helm 操作记录。若保留条数太多，可能会造成数据冗余，保留条数太少可能会造成您所需要的关键操作记录的缺失。需要根据实际情况设置合理的保留数量。具体步骤如下：

1. 点击目标集群的名称，在左侧导航栏点击 **最近操作** -> **Helm 操作** -> **设置保留条数**

保留条数

保留条数

2. 设置需要保留多少条 Helm 操作记录，并点击 **确定**



保留条数

集群升级

Kubernetes 社区每个季度都会发布一次小版本，每个版本的维护周期大概只有 9 个月。

版本停止维护后就不会再更新一些重大漏洞或安全漏洞。手动升级集群操作较为繁琐，给管理人员带来了极大的工作负担。

本节将介绍如何在通过 Web UI 界面一键式在线升级工作集群 Kubernetes 版本，如需离线升级工作集群的 Kubernetes 版本，请参阅[工作集群离线升级指南](#)进行升级。

!!! danger

版本升级后将无法回退到之前的版本，请谨慎操作。

!!! note

- Kubernetes 版本以 x.y.z 表示，其中 x 是主要版本，y 是次要版本，z 是补丁版本。
- 不允许跨次要版本对集群进行升级，例如不能从 1.23 直接升级到 1.25。
- 接入集群 不支持版本升级。如果左侧导航栏没有 集群升级，请检查该集群是否为 接入集群。
- 全局服务集群只能通过终端进行升级。
- 升级工作集群时，该工作集群的[管理集群](cluster-role.md#_3)应该已经接入容器管理模块，并且处于正常运行中。
- 如果需要修改集群参数，可以通过升级相同版本的方式实现，具体操作参考下文。

1. 在集群列表中点击目标集群的名称。

升级集群

升级集群

2. 然后在左侧导航栏点击 **集群运维 -> 集群升级**，在页面右上角点击 **版本升级**。

升级集群

升级集群

3. 选择可升级的版本，输入集群名称进行确认。

可升级版本

可升级版本

!!! note

如果您是想通过升级方式来修改集群参数，请参考以下步骤：

1. 找到集群对应的 ConfigMap，您可以登录控制节点执行如下命令，找到 varsConfRef 中的 ConfigMap 名称。

```
```shell
kubectl get cluster.kubeain.io <clustername> -o yaml
````
```

2. 根据需要，修改 ConfigMap 中的参数信息。

3. 在此处选择相同版本进行升级操作，升级完成即可成功更新对应的集群参数。

4. 点击 **确定** 后，可以看到集群的升级进度。

升级进度

升级进度

5. 集群升级预计需要 30 分钟，可以点击 **实时日志** 按钮查看集群升级的详细日志。

实时日志

实时日志

集群设置

集群设置用于为您的集群自定义高级特性设置，包括是否启用 GPU、Helm 仓库刷新周期、Helm 操作记录保留等。

- 启用 GPU：需要预先在集群上安装 GPU 卡及对应驱动插件。

点击目标集群的名称，在左侧导航栏点击 **最近操作 -> 集群设置 -> Addon 插件**。

配置 gpu

配置 gpu

- Helm 操作基础镜像、仓库刷新周期、操作记录保留条数、**是否开启集群删除保护**
(开启后集群将不能直接卸载)

高级配置

高级配置

集群动态资源超卖

目前，许多业务存在峰值和低谷的现象。为了确保服务的性能和稳定性，在部署服务时，通常会根据峰值需求来申请资源。然而，峰值期可能非常短暂，导致在非峰值期时资源被浪费。**集群资源超卖** 就是将这些申请了而未使用的资源（即申请量与使用量的差值）利用起来，从而提升集群资源利用率，减少资源浪费。

本文主要介绍如何使用集群动态资源超卖功能。

前提条件

- 容器管理模块已[接入 Kubernetes 集群](#)或者已[创建 Kubernetes 集群](#)，且能够访问集群

的 UI 界面。

- 已完成一个[命名空间的创建](#)，并为用户授予 [Cluster Admin](#)，详情可参考[集群授权](#)。
- 若为离线环境，则需完成 [Addon 离线包](#)导入，且在 Helm 模板界面可找到 cro-operator 模板。

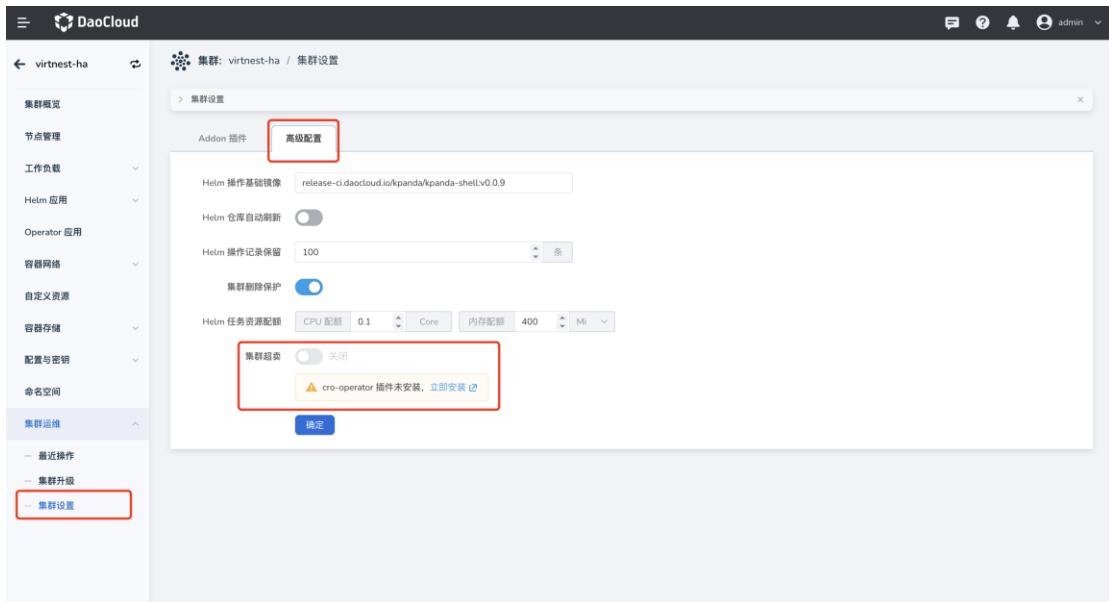
开启集群超卖

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情** 页面

集群列表

2. 在集群详情页面，点击左侧导航栏的 **集群运维** -> **集群设置**，然后选择 **高级配置**

页签



高级设置

3. 打开集群超卖，设置超卖比

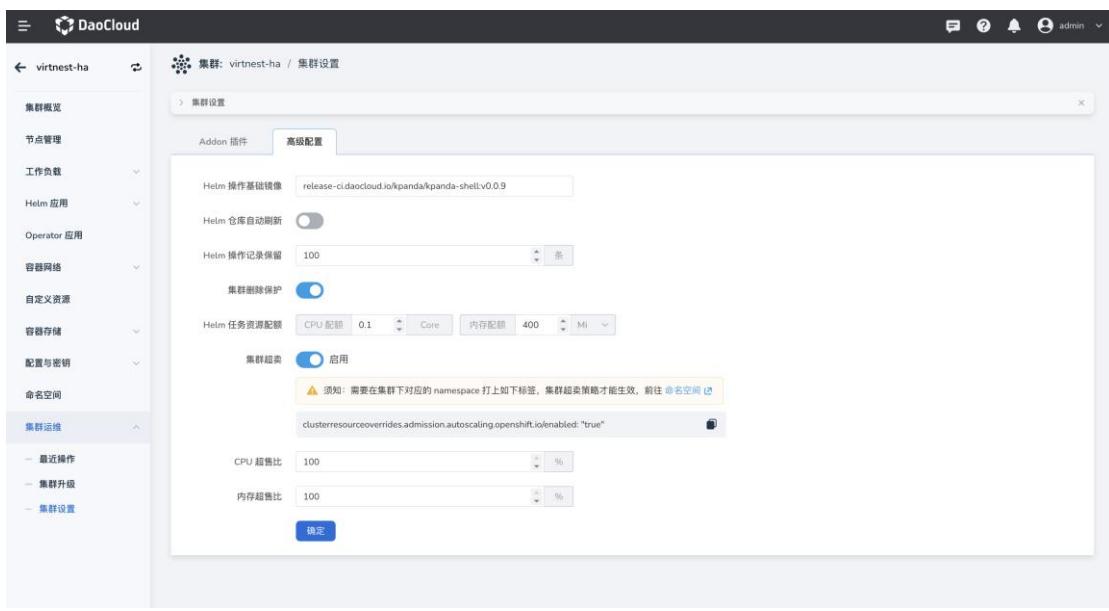
- 若未安装 cro-operator 插件，点击 立即安装 按钮，安装流程参考[管理 Helm 应用](#)

[Helm 应用](#)

- 若已安装 cro-operator 插件，打开集群超卖开关，则可以开始使用集群超卖功能。

!!! note

需要在集群下对应的 namespace 打上如下标签，集群超卖策略才能生效。
`clusterresourceoverrides.admission.autoscaling.openshift.io/enabled: "true"`



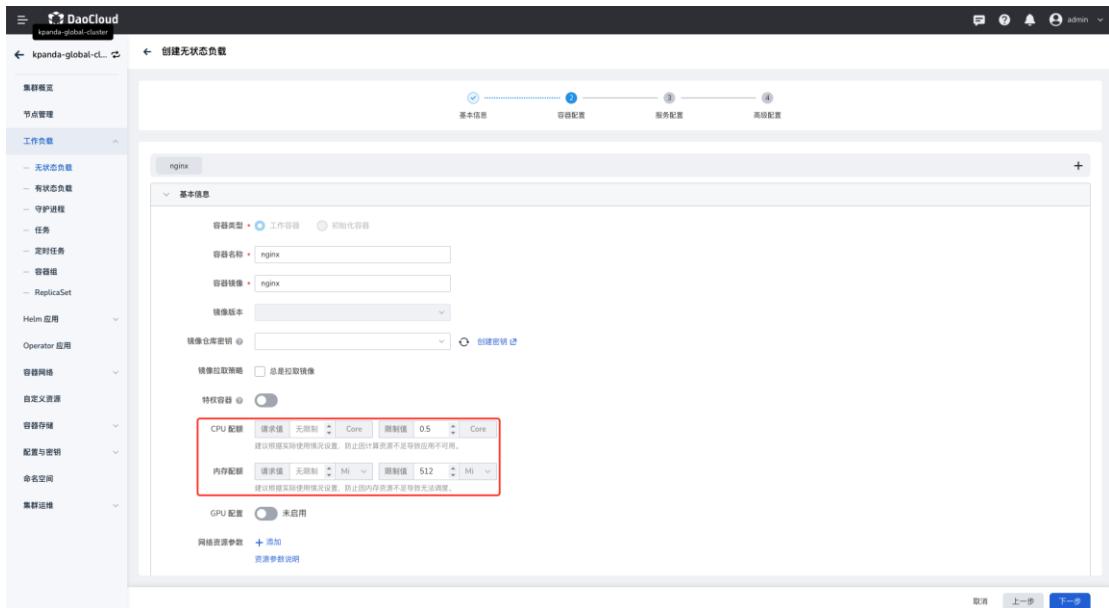
集群超卖

使用集群超卖

设置好集群动态资源超卖比后，会在工作负载运行时生效。下文以 nginx 为例，验证使用资源超卖能力。

1. 创建工作负载 nginx 并设置对应的资源限制值，创建流程参考[创建无状态负载](#)

(Deployment)



创建工作负载

2. 查看工作负载的 Pod 资源申请值与限制值的比值是否符合超售比

| 容器组名称 | 状态 | 容器 (正常/总量) | 容器组 IP | 节点 | 重启次数 | CPU 请求值/限制值 | 内存请求值/限制值 | GPU 配置 | 创建时间 |
|-----------------------|-----|------------|-----------------|-----------|------|----------------------|-----------------|--------|------------------|
| nginx-5b7446898-fnbfp | 运行中 | 1/1 | 10.23.1.120.177 | master-01 | 0 | 0.25 Core / 0.5 Core | 256 Mi / 0.5 Gi | | 2024-05-16 11:13 |

查看 Pod 资源

集群巡检

集群巡检可以通过自动或手动方式，定期或随时检查集群的整体健康状态，让管理员获得保障集群安全的主动权。基于合理的巡检计划，这种主动自发的集群检查可以让管理员随时掌握集群状态，摆脱之前出现故障时只能被动排查问题的困境，做到事先监控、提前防范。

DCE 5.0 容器管理模块提供的集群巡检功能，支持从集群、节点、容器组（Pod）三个维度进行自定义巡检项，巡检结束后会自动生成可视化的巡检报告。

- **集群维度**：检查集群中系统组件的运行情况，包括集群状态、资源使用情况以及控制节点特有的巡检项等，例如 **kube-apiserver** 和 **etcd** 的状态。
- **节点维度**：包括控制节点和工作节点通用的检查项，例如节点资源使用情况、句柄数、PID 状态、网络状态。

- 容器组维度：检查 Pod 的 CPU 和内存使用情况、运行状态、PV 和 PVC 的状态等。

如需了解或执行安全方面的巡检，可参考 DCE 5.0 支持的[安全扫描类型](#)。

创建巡检配置

DCE 5.0 容器管理模块提供集群巡检功能，支持从集群维度、节点维度、容器组维度进行巡检。

- 集群维度：检查集群中系统组件的运行情况，包括集群状态、资源使用情况，以及控制节点特有的巡检项等，例如 `kube-apiserver` 和 `etcd` 的状态。
- 节点维度：包括控制节点和工作节点通用的检查项，例如节点资源使用情况、句柄数、PID 状态、网络状态。
- 容器组维度：检查 Pod 的 CPU 和内存使用情况、运行状态、PV 和 PVC 的状态等。

下面介绍如何创建巡检配置。

前提条件

- 在容器管理模块中[接入](#)或[创建集群](#)
- 所选集群处于[运行中](#)状态且已经在集群中[安装了 insight 组件](#)

操作步骤

- 在左侧导航栏点击 **集群巡检**。

nav

nav

- 在页面右侧点击 **巡检配置**。

The screenshot shows the 'Cluster Inspection' configuration page. At the top right, there are several icons: a message bubble, a question mark, a bell, and a user profile for 'admin'. Below the header, there's a breadcrumb navigation: '← 集群巡检' and '集群巡检'. On the right side of the page, there are two buttons: '创建巡检配置' (highlighted with a red box) and '巡检'.

| 集群 | 最近巡检状态 | 最近巡检 | 触发方式 | 巡检项 | 巡检记录 | 开始时间 | 创建时间 | 更多操作 |
|-----------------------|--------|-----------------------------|------|-----|------|------------------|------------------|------|
| cluster202 | 中危 | Inspect-cluster202-20240... | 定时巡检 | 24 | 3 | 2024-06-24 17:21 | 2024-05-07 11:44 | ⋮ |
| kpanda-global-cluster | 中危 | Inspect-kpanda-global-cl... | 定时巡检 | 28 | 5 | 2024-06-24 14:47 | 2024-05-28 09:27 | ⋮ |

create

3. 参考以下说明填写巡检配置，然后在页面底部点击 **确定** 即可。

- 集群：下拉选择要对哪些集群进行巡检。如果选择多个集群，则自动生成多个巡检配置（仅巡检的集群不一致，其他配置都完全一致）

- 定时巡检：启用后可根据事先设置的巡检频率定期自动执行集群巡检

- 巡检频率：设置自动巡检的周期，例如每周二上午十点。支持自定义

CronExpression，可参考 [Cron 时间表语法](#)

- 巡检记录保留条数：累计最多保留多少条巡检记录，包括所有集群的巡检记录

- 参数配置：参数配置分为集群维度、节点维度、容器组维度三部分，可以根据场景需求启用或禁用某些巡检项。

basic

basic

巡检配置创建完成后，会自动显示在巡检配置列表中。在配置右侧点击更多操作按钮可以立即执行巡检、修改巡检配置、删除巡检配置和巡检记录。

- 点击 **巡检** 可以根据该配置立即执行一次巡检。
- 点击 **巡检配置** 可以修改巡检配置。
- 点击 **删除** 可以删除该巡检配置和历史的巡检记录

| 集群 | 最近巡检状态 | 最近巡检 | 触发方式 | 巡检项 | 巡检记录 | 开始时间 | 创建时间 |
|-----------------------|--------|-----------------------------|------|-----|------|------------------|------------------|
| cluster202 | ● 中危 | Inspect-cluster202-20240... | 定时巡检 | 24 | 3 | 2024-06-24 17:21 | 2024-05-07 11:44 |
| kpanda-global-cluster | ● 中危 | Inspect-kpanda-global-cl... | 定时巡检 | 28 | 5 | 2024-06-24 14:47 | 2024-05-07 11:44 |

basic

!!! note

- 巡检配置创建完成后，如果启用了 定时巡检 配置，则会在指定时间自动执行巡检。
- 如未启用 定时巡检 配置，则需要手动[触发巡检](inspect.md)。

执行集群巡检

巡检配置创建完成后，如果启用了 **定时巡检** 配置，则会在指定时间自动执行巡检。如未启用 **定时巡检** 配置，则需要手动触发巡检。

此页介绍如何手动执行集群巡检。

前提条件

- 在容器管理模块中接入或创建集群
- 已创建巡检配置
- 所选集群处于**运行中**状态且已经在集群中安装了 insight 组件

操作步骤

执行巡检时，支持勾选多个集群进行批量巡检，或者仅对某一个集群进行单独巡检。

==== “批量巡检”

1. 在容器管理模块的一级导航栏点击 集群巡检，然后在页面右侧点击 巡检。

![start](docs/zh/docs/kpanda/user-guide/images/inspection-start.png)

2. 勾选需要巡检的集群，然后在页面底部点击 确定 即可。

- 若选择多个集群进行同时巡检，系统将根据不同集群的巡检配置进行巡检。
- 如未设置集群巡检配置，将使用系统默认配置。

![start](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/kpanda/images/inspect05.png)

单独巡检

1. 进入集群巡检页面。
2. 在对应巡检配置的右侧点击 更多操作 按钮，然后在弹出的菜单中选择 巡检 即可。

![basic](docs/zh/docs/kpanda/user-guide/images/inspection-start-alone.png)

查看巡检报告

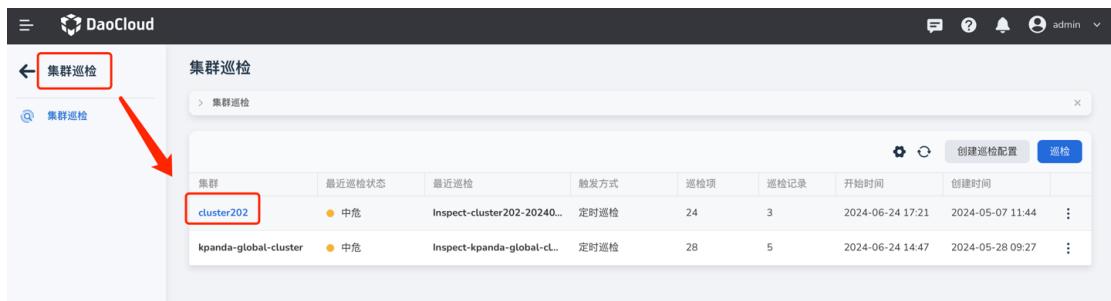
巡检执行完成后，可以查看巡检记录和详细的巡检报告。

前提条件

- 已经创建了巡检配置
- 已经执行过至少一次巡检

操作步骤

1. 进入集群巡检页面，点击目标巡检集群的名称。



start

2. 点击想要查看的巡检记录名称。

- 每执行一次巡检，就会生成一条巡检记录。
- 当巡检记录超过巡检配置中设置的最大保留条数时，从执行时间最早的记录

开始删除。

| Audit Name | Audit Status | Trigger Type | Audit Items | Start Time | End Time |
|----------------------------------|--------------|--------------|-------------|------------------|------------------|
| Inspect-cluster202-20240624-1721 | ● 检查完成 | 定时巡检 | 24 | 2024-06-24 17:21 | 2024-06-24 17:21 |
| Inspect-cluster202-20240613-1425 | ● 检查完成 | 手动触发 | 24 | 2024-06-13 14:25 | 2024-06-13 14:25 |
| Inspect-cluster202-20240613-1425 | ● 检查完成 | 手动触发 | 24 | 2024-06-13 14:25 | 2024-06-13 14:25 |

start

3. 查看巡检的详细信息，根据巡检配置可能包括集群资源概览、系统组件的运行情况

等。

在页面右上角可以下载巡检报告或删除该项巡检报告。

| 集群深度 | 检查项 | 检查结果 | 规则 | 状态 |
|------|-----------------------------|------|------------|------|
| | kube-apiserver 的状态 | 1 | 检测组件是否正在运行 | ● 正常 |
| | kube-controller-manager 的状态 | 1 | 检测组件是否正在运行 | ● 正常 |

start

离线升级集群巡检模块

本页说明[下载集群巡检模块](#)后，应该如何安装或升级。

!!! info

下述命令或脚本内出现的 `_kcollie_` 字样是集群巡检模块的内部开发代号。

从下载的安装包中加载镜像

您可以根据下面两种方式之一加载镜像，当环境中存在镜像仓库时，建议选择 `chart-syncer` 同步镜像到镜像仓库，该方法更加高效便捷。

方式一：使用 `chart-syncer` 同步镜像

使用 `chart-syncer` 可以将您下载的安装包中的 `chart` 及其依赖的镜像包上传至安装器部署 DCE 时使用的镜像仓库和 `helm` 仓库。

首先找到一台能够连接镜像仓库和 `helm` 仓库的节点（如火种节点），在节点上创建 `load-image.yaml` 配置文件，填入镜像仓库和 `helm` 仓库等配置信息。

1. 创建 `load-image.yaml`

!!! note

该 YAML 文件中的各项参数均为必填项。

==== “已添加 Helm repo”

若当前环境已安装 `chart` `repo`, `chart-syncer` 也支持将 `chart` 导出为 `tgz` 文件。

```
```yaml title="load-image.yaml"
source:
 intermediateBundlesPath: kcollie # 使用 chart-syncer 之后 .tar.gz 包所在的路径
target:
 containerRegistry: 10.16.10.111 # 镜像仓库地址
 containerRepository: release.daocloud.io/kcollie # 镜像仓库路径
repo:
 kind: HARBOR # Helm Chart 仓库类别
 url: http://10.16.10.111/chartrepo/release.daocloud.io # Helm 仓库地址
```

```

auth:
 username: "admin" # 镜像仓库用户名
 password: "Harbor12345" # 镜像仓库密码
containers:
 auth:
 username: "admin" # Helm 仓库用户名
 password: "Harbor12345" # Helm 仓库密码
```

```

==== “未添加 Helm repo”

若当前节点上未添加 helm repo, chart-syncer 也支持将 chart 导出为 tgz 文件，并存放 在指定路径。

```

```yaml title="load-image.yaml"
source:
 intermediateBundlesPath: kcollie # 使用 chart-syncer 之后 .tar.gz 包所在的路径
target:
 containerRegistry: 10.16.10.111 # 镜像仓库 url
 containerRepository: release.daocloud.io/kcollie # 镜像仓库路径
repo:
 kind: LOCAL
 path: ./local-repo # chart 本地路径
containers:
 auth:
 username: "admin" # 镜像仓库用户名
 password: "Harbor12345" # 镜像仓库密码
```

```

2. 执行同步镜像命令。

```
charts-syncer sync --config load-image.yaml
```

方式二：使用 Docker 或 containerd 加载镜像

解压并加载镜像文件。

1. 解压第一层压缩包。

```
tar xvf kcollie.amd64.tar
```

解压成功后会得到 1 个新的压缩包：

- kcollie.bundle.tar

2. 解压新的压缩包。

```
tar xvf kcollie.bundle.tar
```

解压成功后会得到 3 个文件：

- hints.yaml
- images.tar
- original-chart

3. 从本地加载镜像到 Docker 或 containerd。

```
==== "Docker"
```shell
docker load -i images.tar
```
==== "containerd"
```shell
ctr -n k8s.io image import images.tar
```

```

!!! note

每个 node 都需要做 Docker 或 containerd 加载镜像操作，
加载完成后需要 tag 镜像，保持 Registry、Repository 与安装时一致。

升级

有两种升级方式。您可以根据前置操作，选择对应的升级方案：

==== “通过 helm repo 升级”

1. 检查集群巡检 helm 仓库是否存在。

```
```shell
helm repo list | grep kcollie
```

```

若返回结果为空或如下提示，则进行下一步；反之则跳过下一步。

```
```none
Error: no repositories to show
```

```

1. 添加集群巡检的 helm 仓库。

```
```shell
helm repo add kcollie http://{harbor url}/chartrepo/{project}
```

```

1. 更新集群巡检的 helm 仓库。

```
```shell
helm repo update kcollie
```

```

1. 选择您想安装的集群巡检版本（建议安装最新版本）。

```
```shell
helm search repo kcollie/kcollie --versions
```

```none
[root@master ~]# helm search repo kcollie/kcollie --versions
NAME CHART VERSION APP VERSION DESCRIPTION
kcollie/kcollie 0.20.0 v0.20.0 A Helm chart for kcollie
...
```

```

1. 备份 `--set` 参数。

在升级集群巡检版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

```
```shell
helm get values kcollie -n kcollie-system -o yaml > bak.yaml
```

```

1. 更新 kcollie crds (需要先解压并进入 original-chart 文件)

```
```shell
helm pull kcollie/kcollie --version 0.6.0 && tar -zxf kcollie-0.6.0.tgz
kubectl apply -f kcollie/crds
```

```

1. 执行 `helm upgrade` 。

升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 字段为当前使用的镜像仓库地址

。

```
```shell
export imageRegistry={你的镜像仓库}
```

```

```
```shell
helm upgrade kcollie kcollie/kcollie \
```

```

```
-n kcollie-system \
-f ./bak.yaml \
--set global.imageRegistry=$imageRegistry \
--version 0.6.0
```

```

### ==== “通过 chart 包升级”

1. 备份 `--set` 参数。

在升级集群巡检版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

```
```shell
helm get values kcollie -n kcollie-system -o yaml > bak.yaml
```

```

2. 更新 kcollie crds (需要先解压并进入 original-chart 文件)

```
```shell
kubectl apply -f ./crds
```

```

3. 执行 `helm upgrade`。

升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 为当前使用的镜像仓库地址。

```
```shell
export imageRegistry={你的镜像仓库}
```

```shell
helm upgrade kcollie . \
-n kcollie-system \
-f ./bak.yaml \
--set global.imageRegistry=$imageRegistry
```

```

## 备份恢复

备份恢复分为备份和恢复两方面，实际应用时需要先备份系统在某一时点的数据，然后安全存储地备份数据。后续如果出现数据损坏、丢失、误删等事故，就可以基于之前的数据

备份快速还原系统，缩短故障时间，减少损失。

- 在真实的生产环境中，服务可能分布式地部署在不同的云、不同区域或可用区，如果某一个基础设施自身出现故障，企业需要在其他可用环境中快速恢复应用。在这种情况下，跨云/跨集群的备份恢复显得非常重要。
- 在大规模系统中往往有很多角色和用户，权限管理体系复杂，操作者众多，难免有人误操作导致系统故障。在这种情况下，也需要能够通过之前备份的数据快速回滚系统，否则如果依赖人为排查故障、修复故障、恢复系统就会耗费大量时间，系统不可用时间越长，企业的损失越大。
- 此外，还有网络攻击、自然灾害、设备故障等各种因素也可能导致数据事故因此，备份恢复非常重要，可以视之为维护系统稳定和数据安全的最后一道保险。

备份通常分为全量备份、增量备份、差异备份三种。DCE 5.0 目前支持全量备份和增量备份。

DCE 5.0 提供的备份恢复可以分为 应用备份 和 ETCD 备份 两种，支持手动备份，或基于 CronJob 定时自动备份。

- 应用备份

应用备份指，备份集群中的某个工作负载的数据，然后将该工作负载的数据恢复到本集群或者其他集群。支持备份整个命名空间下的所有资源，也支持通过标签选择器过滤，仅备份带有特定标签的资源。

应用备份支持跨集群备份有状态应用，具体步骤可参考 [MySQL 应用及数据的跨集群备份恢复](#)。

- ETCD 备份

etcd 是 Kubernetes 的数据存储组件，Kubernetes 将自身的组件数据和其中的应用数

据都存储在 etcd 中。因此，备份 etcd 就相当于备份整个集群的数据，可以在故障时快速将集群恢复到之前某一时点的状态。

需要注意的是，目前仅支持将 etcd 备份数据恢复到同一集群（原集群）。如需了解与此相关的最佳实践，可参考 [ETCD 备份还原](#)。

## 安装 velero 插件

velero 是一个备份和恢复 Kubernetes 集群资源的开源工具。它可以将 Kubernetes 集群中的资源备份到云存储服务、本地存储或其他位置，并且可以在需要时将这些资源恢复到同一或不同的集群中。

本节介绍如何在 DCE 5.0 中使用 Helm 应用 部署 velero 插件。

### 前提条件

安装 velero 插件前，需要满足以下前提条件：

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建 [velero 命名空间](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。

### 操作步骤

请执行如下步骤为集群安装 velero 插件。

- 在集群列表页面找到需要安装 velero 插件的目标集群，点击集群名称，在左侧导航栏依次点击 **Helm 应用 -> Helm 模板**，在搜索栏输入 **velero** 进行搜索。

## 备份恢复

### 备份恢复

2. 阅读 **velero** 插件相关介绍，选择版本后点击 **安装** 按钮。本文将以 **4.0.2** 版本为例进行安装，推荐安装 **4.0.2** 或更高版本。

## 备份恢复

### 备份恢复

3. 填写和配置参数后点击 **下一步**

#### ==== “基本参数”

![备份恢复](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/kpanda/images/backups3.png)

- **名称**: 必填参数，输入插件名称，请注意名称最长 63 个字符，只能包含小写字母、数字及分隔符（“-”），且必须以小写字母或数字开头及结尾，例如 metrics-server-01。
- **命名空间**: 插件安装的命名空间，默认为 **\_velero\_** 命名空间。
- **版本**: 插件的版本，此处以 **\_4.0.2\_** 版本为例。
- **就绪等待**: 可选参数，启用后，将等待应用下所有关联资源处于就绪状态，才会标记应用安装成功。
- **失败删除**: 可选参数，开启后，将默认同步开启就绪等待。如果安装失败，将删除安装相关资源。
- **详情日志**: 可选参数，开启后将输出安装过程的详细日志。

#### !!! note

开启 **\_就绪等待\_** 和/或 **\_失败删除\_** 后，应用需要经过较长时间才会被标记为 **\_运行中\_** 状态。

#### ==== “参数配置”

##### - S3 Credentials:

- **\_Use secret\_** : 保持默认配置 **\_true\_**。
- **\_Secret name\_** : 保持默认配置 **\_velero-s3-credential\_**。
- **\_SecretContents.aws\_access\_key\_id = <modify>\_** : 配置访问对象存储的用户名，替换 **\_<modify>\_** 为真实参数。
- **\_SecretContents.aws\_secret\_access\_key = <modify>\_** : 配置访问对象存储的密码，替换 **\_<modify>\_** 为真实参数。

```
```config "SecretContents 样例"
[default]
aws_access_key_id = minio
aws_secret_access_key = minio123
```

```

- Velero Configuration:

- Backupstorage location : velero 备份数据存储的位置
- S3 bucket : 用于保存备份数据的存储桶名称(需为 minio 已经存在的真实存储桶)
- Is default BackupStorage : 保持默认配置 true
- S3 access mode : velero 对数据的访问模式, 可以选择
  - ReadWrite : 允许 velero 读写备份数据
  - ReadOnly : 允许 velero 读取备份数据, 不能修改备份数据
  - WriteOnly : 只允许 velero 写入备份数据, 不能读取备份数据
- S3 Configs : S3 存储 (minio) 的详细配置
- S3 region : 云存储的地理区域。默认使用 us-east-1 参数, 由系统管理员提供
  - S3 force path style : 保持默认配置 true
  - S3 server URL : 对象存储 (minio) 的控制台访问地址, minio 一般提供了 UI 访问和控制台访问两个服务, 此处请使用控制台访问的地址

**!!! note**

请确保 s3 存储服务时间跟备份还原集群时间差在 10 分钟以内, 最好是时间保持同步, 否则将无法执行备份操作。

- migration plugin configuration: 启用之后, 将在下一步的 YAML 代码段中新增:

```
```yaml
...
initContainers:
- image: 'release.daocloud.io/kcoral/velero-plugin-for-migration:v0.3.0'
  imagePullPolicy: IfNotPresent
  name: velero-plugin-for-migration
  volumeMounts:
    - mountPath: /target
      name: plugins
- image: 'docker.m.daocloud.io/velero/velero-plugin-for-csi:v0.7.0'
  imagePullPolicy: IfNotPresent
  name: velero-plugin-for-csi
  volumeMounts:
    - mountPath: /target

```

```

    name: plugins
  - image: 'docker.m.daocloud.io/velero/velero-plugin-for-aws:v1.9.0'
    imagePullPolicy: IfNotPresent
    name: velero-plugin-for-aws
    volumeMounts:
      - mountPath: /target
        name: plugins
...
...

```

4. 确认 YAML 无误后点击 **确定**，完成 **velero** 插件的安装。之后系统将自动跳转至 **Helm 应用** 列表页面，稍等几分钟后，为页面执行刷新操作，即可看到刚刚安装的应用。

应用备份

本文介绍如何在 DCE 5.0 中为应用做备份，本教程中使用的演示应用名为 **dao-2048**，属于无状态工作负载。

前提条件

在对无状态工作负载进行备份前，需要满足以下前提条件：

- 在[容器管理](#)模块中[接入 Kubernetes 集群](#)或者[创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建一个[命名空间](#)和[用户](#)。
- 当前操作用户应具有 [NS Editor](#) 或更高权限，详情可参考[命名空间授权](#)。
- [安装 velero 组件](#)，且 velero 组件运行正常。
- [创建一个无状态工作负载](#)（本教程中的负载名为 **dao-2048**），并为无状态工作负载打上 **app: dao-2048** 的标签。

备份工作负载

参考以下步骤，备份无状态工作负载 **dao-2048**。

- 在左侧导航栏，点击 **容器管理 -> 备份恢复**。

The screenshot shows the DaoCloud Enterprise 5.0 dashboard. On the left, there is a navigation sidebar with various options like '仪表盘', '应用工作台', '容器', '可观测性', etc. Under '容器' (Containers), '容器管理' (Container Management) is selected, and '备份恢复' (Backup & Recovery) is highlighted with a red box. The main area displays real-time monitoring data for CPU, memory, and disk usage across the cluster. A prominent red '不健康' (Unhealthy) status indicator is visible. On the right, there's a '告警' (Alerts) section showing counts for critical (145), warning (3395), and info (300) alerts, along with a log viewer.

集群列表

- 进入 **应用备份** 列表页面，从集群下拉列表中选择已安装了 velero 和 **dao-2048** 的集群。点击右侧的 **创建备份计划** 按钮。

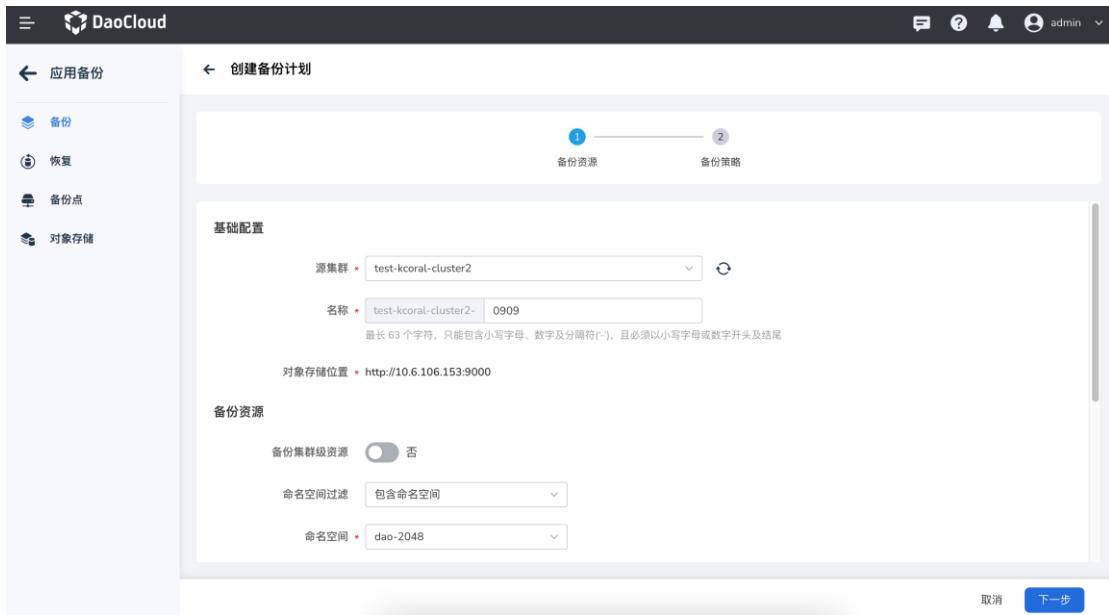
The screenshot shows the 'Application Backup' page. On the left, there's a sidebar with '应用备份' (Application Backup) selected, followed by '备份' (Backup), '恢复' (Recovery), '备份点' (Backup Points), and '对象存储' (Object Storage). The main area is titled '备份' (Backup) and shows a table of existing backup schedules:

名称	源集群	资源选择类型	状态	上次执行状态	下次执行时间
aaa-test-schedule	test-kcoral-cluster2	命名空间	运行中	成功	2024-09-09 19:00
zzz-test-schedule	test-kcoral-cluster2	命名空间	运行中	成功	2024-09-09 19:00

At the top right of the table, there are buttons for 'YAML 创建' (Create via YAML) and '创建备份计划' (Create Backup Plan). The URL of the page is https://demo-dev.daocloud.io/kpanda/backup-restore.

应用备份

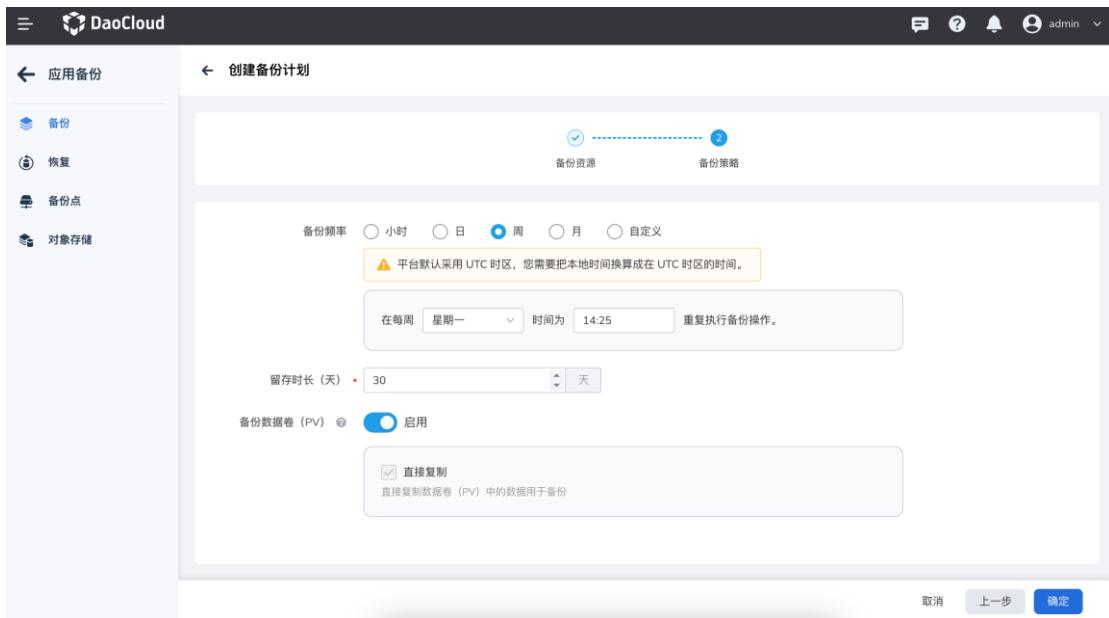
- 参考下方说明填写备份配置。



操作菜单

4. 参考下方说明设置备份执行频率，然后点击 **下一步**。

- 备份频率：基于分钟、小时、天、周、月设置任务执行的时间周期。支持用数字和 * 自定义 Cron 表达式，输入表达式后下方会提示当前表达式的含义。有关详细的表达式语法规则，可参考 [Cron 时间表语法](#)。
- 留存时长（天）：设置备份资源保存的时间，默认为 30 天，过期后将被删除。
- 备份数据卷（PV）：是否备份数据卷（PV）中的数据，支持直接复制和使用 CSI 快照两种方式。
 - 直接复制：直接复制数据卷（PV）中的数据用于备份；
 - 使用 CSI 快照：使用 CSI 快照来备份数据卷（PV）。需要集群中有可用于备份的 CSI 快照类型。



操作菜单

5. 点击 **确定**，页面会自动返回应用备份计划列表。您可以找到新建的 **dao-2048** 备份计划，在右侧点击 ，选择 **立即执行** 开始备份。

操作菜单

操作菜单

6. 此时集群的 **上一次执行状态** 将转变为 **备份中**。等待备份完成后可以点击备份计划的名称，查看备份计划详情。

操作菜单

操作菜单

!!! note

如果 Job 类型的工作负载状态为 ****执行完成****，则不支持备份。

etcd 备份

etcd 备份是以集群数据为核心的备份。在硬件设备损坏，开发测试配置错误等场景中，可以通过 etcd 备份恢复集群数据。

本文介绍如何为集群制作 etcd 备份。另请参阅 [etcd 备份还原最佳实践](#)。

前提条件

- 接入或者[创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 创建[命名空间](#)和[用户](#)，并为用户授予 [NS Admin](#) 或更高权限。详情可参考[命名空间授权](#)。
- 准备一个 MinIO 实例。建议通过 DCE 5.0 的 MinIO 中间件进行创建，具体步骤可参考 [MinIO 对象存储](#)。

创建 etcd 备份

参照以下步骤创建 etcd 备份。

1. 进入 [容器管理](#) -> [备份恢复](#) -> [etcd 备份](#)，点击 [备份策略](#) 页签，然后在右侧点击 [创建备份策略](#)。

备份策略列表

备份策略列表

2. 参考以下说明填写 [基本信息](#)。填写完毕后点击 [下一步](#)，系统将自动校验 etcd 的联通性，校验通过之后可以进行下一步。

- 备份集群：选择需要备份哪个集群的 etcd 数据，并在终端登录
- etcd 地址：格式为 `https://${节点 IP}:${端口号}`
 - 在标准 Kubernetes 集群中，etcd 的默认端口号为 **2379**
 - 在 DCE 4.0 集群中，etcd 的默认端口号为 **12379**
 - 在公有云托管集群中，需要联系相关开发人员获取 etcd 的端口号。

这是因为公有云集群的控制面组件由云服务提供商维护和管理，用户无法直接访问或查看这些组件，也无法通过常规命令（如 `kubectl`）无法获取到控制面的端口号等信息。

??? note “获取端口号的方式”

1. 在 `_kube-system_` 命名空间下查找 etcd Pod

```
```shell
kubectl get po -n kube-system | grep etcd
```
```

```

2. 获取 etcd Pod 的 `_listen-client-urls_` 中的端口号

```
```shell
kubectl get po -n kube-system ${etcd_pod_name} -oyaml | grep listen-client-urls # (1)
```
```

```

1. 将 `_etcd_pod_name_` 替换为实际的 Pod 名称

预期输出结果如下，节点 IP 后的数字即为端口号：

```
```shell
- --listen-client-urls=https://127.0.0.1:2379,https://10.6.229.191:2379
```
```

```

- CA 证书：可通过如下命令查看证书，然后将证书内容复制粘贴到对应位置：

#### ==== “标准的 Kubernetes 集群”

```
```shell
cat /etc/kubernetes/ssl/etcd/ca.crt
```
```

```

==== “DCE 4.0 集群”

```
```shell
cat /etc/daocloud/dce/certs/ca.crt
```
```

```

- Cert 证书：可通过如下命令查看证书，然后将证书内容复制粘贴到对应位

置：

==== “标准的 Kubernetes 集群”

```
```shell
cat /etc/kubernetes/ssl/apiserver-etcd-client.crt
```

```

==== “DCE 4.0 集群”

```
```shell
cat /etc/daocloud/dce/certs/etcd/server.crt
```

```

- **Key**：可通过如下命令查看证书，然后将证书内容复制粘贴到对应位置：

==== “标准的 Kubernetes 集群”

```
```shell
cat /etc/kubernetes/ssl/apiserver-etcd-client.key
```

```

==== “DCE 4.0 集群”

```
```shell
cat /etc/daocloud/dce/certs/etcd/server.key
```

```

创建基本信息

创建基本信息

#### !!! note

点击输入框下方的 如何获取 可以在 UI 页面查看获取对应信息的方式。

### 3.参考以下信息填写 备份策略。

- 备份方式：选择手动备份或定时备份
  - 手动备份：基于备份配置立即执行一次 etcd 全量数据的备份。
  - 定时备份：按照设置的备份频率对 etcd 数据进行周期性全量备份。
- 备份链长度：最多保留多少条备份数据。默认为 30 条。
- 备份频率：支持小时、日、周、月级别和自定义方式。

## 定时备份

### 定时备份

4. 参考以下信息填写 **存储位置**。

- 存储供应商：默认选择 S3 存储
- 对象存储访问地址：MinIO 的访问地址
- 存储桶：在 MinIO 中创建一个 Bucket，填写 Bucket 的名称
- 用户名：MinIO 的登录用户名
- 密码：MinIO 的登录密码

## 存储位置

### 存储位置

5. 点击 **确定** 后页面自动跳转到备份策略列表，可以查看目前创建好的所有策略。

- 在策略右侧点击  操作按钮可以查看日志、查看 YAML、更新策略、停止策略、立即执行策略等。
- 当备份方式为手动时，可以点击 **立即执行** 进行备份。
- 当备份方式为定时备份时，则会根据配置的时间进行备份。

## 成功创建

### 成功创建

## 查看备份策略日志

点击 **日志** 可以查看日志内容，默认展示 100 行。若想查看更多日志信息或者下载日志，可在日志上方根据提示前往可观测性模块。

## 查看日志

查看日志

## 查看备份策略详情

进入 **容器管理** -> **备份恢复** -> **etcd 备份** , 点击 **备份策略** 页签 , 接着点击策略名称可以查看策略详情。

备份策略详情

备份策略详情

## 查看备份点

1. 进入 **容器管理** -> **备份恢复** -> **etcd 备份** , 点击 **备份点** 页签。

2. 选择目标集群后 , 可以查看该集群下所有备份信息。

每执行一次备份 , 对应生成一个备份点 , 可通过成功状态的备份点快速恢复应用。

备份点

备份点

## 离线升级备份恢复模块

本页说明[下载备份恢复模块](#)后 , 应该如何安装或升级。

!!! info

下述命令或脚本内出现的 `_kcoral_` 字样是备份恢复模块的内部开发代号。

### 从下载的安装包中加载镜像

您可以根据下面两种方式之一加载镜像 , 当环境中存在镜像仓库时 , 建议选择 `chart-syncher` 同步镜像到镜像仓库 , 该方法更加高效便捷。

## 方式一：使用 chart-syncer 同步镜像

使用 chart-syncer 可以将您下载的安装包中的 chart 及其依赖的镜像包上传至安装器部署

DCE 时使用的镜像仓库和 helm 仓库。

首先找到一台能够连接镜像仓库和 helm 仓库的节点（如火种节点），在节点上创建 load-image.yaml 配置文件，填入镜像仓库和 helm 仓库等配置信息。

### 1. 创建 load-image.yaml

!!! note

该 YAML 文件中的各项参数均为必填项。

==== “已添加 Helm repo”

若当前环境已安装 chart repo，chart-syncer 也支持将 chart 导出为 tgz 文件。

```
```yaml title="load-image.yaml"
source:
  intermediateBundlesPath: kcoral # (1)!
target:
  containerRegistry: 10.16.10.111 # (2)!
  containerRepository: release.daocloud.io/kcoral # (3)!
repo:
  kind: HARBOR # (4)!
  url: http://10.16.10.111/chartrepo/release.daocloud.io # (5)!
  auth:
    username: "admin" # (6)!
    password: "Harbor12345" # (7)!
containers:
  auth:
    username: "admin" # (8)!
    password: "Harbor12345" # (9)!
````
```

1. 使用 chart-syncer 之后 .tar.gz 包所在的路径
2. 镜像仓库地址
3. 镜像仓库路径
4. Helm Chart 仓库类别
5. Helm 仓库地址
6. 镜像仓库用户名
7. 镜像仓库密码

8. Helm 仓库用户名
9. Helm 仓库密码

==== “未添加 Helm repo”

若当前节点上未添加 helm repo, chart-syncer 也支持将 chart 导出为 tgz 文件，并存放  
在指定路径。

```
```yaml title="load-image.yaml"
source:
  intermediateBundlesPath: kcoral # (1)!
target:
  containerRegistry: 10.16.10.111 # (2)!
  containerRepository: release.daocloud.io/kcoral # (3)!
repo:
  kind: LOCAL
  path: ./local-repo # (4)!
containers:
  auth:
    username: "admin" # (5)!
    password: "Harbor12345" # (6)!

```

```

1. 使用 chart-syncer 之后 .tar.gz 包所在的路径
2. 镜像仓库 url
3. 镜像仓库路径
4. chart 本地路径
5. 镜像仓库用户名
6. 镜像仓库密码

2. 执行同步镜像命令。

```
charts-syncer sync --config load-image.yaml
```

## 方式二：使用 Docker 或 containerd 加载镜像

解压并加载镜像文件。

1. 解压第一层压缩包。

```
tar xvf kcoral.amd64.tar
```

解压成功后会得到 1 个新的压缩包：

- kcoral.bundle.tar

2. 解压新的压缩包。

```
tar xvf kcoral.bundle.tar
```

解压成功后会得到 3 个文件：

- hints.yaml
- images.tar
- original-chart

### 3. 从本地加载镜像到 Docker 或 containerd。

```
==== "Docker"
```shell
docker load -i images.tar
```
==== "containerd"
```shell
ctr -n k8s.io image import images.tar
```

```

#### !!! note

每个 node 都需要做 Docker 或 containerd 加载镜像操作，  
加载完成后需要 tag 镜像，保持 Registry、Repository 与安装时一致。

## 升级

有两种升级方式。您可以根据前置操作，选择对应的升级方案：

#### ==== “通过 helm repo 升级”

1. 检查备份恢复 helm 仓库是否存在。

```
```shell
helm repo list | grep kcoral
```

```

若返回结果为空或如下提示，则进行下一步；反之则跳过下一步。

```
```none
Error: no repositories to show
```

```

1. 添加备份恢复的 helm 仓库。

```
```shell
helm repo add kcoral http://{harbor url}/chartrepo/{project}
```

```

- 更新备份恢复的 helm 仓库。

```
```shell
helm repo update koral
```

```

- 选择您想安装的备份恢复版本（建议安装最新版本）。

```
```shell
helm search repo koral/koral --versions
```

```

```
```none
[root@master ~]# helm search repo koral/koral --versions
NAME          CHART VERSION APP VERSION DESCRIPTION
koral/koral   0.20.0      v0.20.0     A Helm chart for koral
...
```

```

- 备份 `--set` 参数。

在升级备份恢复版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

```
```shell
helm get values koral -n koral-system -o yaml > bak.yaml
```

```

- 执行 `helm upgrade`。

升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 字段为当前使用的镜像仓库地址

。

```
```shell
export imageRegistry={你的镜像仓库}
```

```

```
```shell
helm upgrade koral koral/koral \
-n koral-system \
-f ./bak.yaml \
--set global.imageRegistry=$imageRegistry \
--version 0.5.0
```

```

==== “通过 chart 包升级”

## 1. 备份 `--set` 参数。

在升级备份恢复版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

```
```shell
helm get values koral -n koral-system -o yaml > bak.yaml
```

```

## 2. 执行 `helm upgrade`。

升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 为当前使用的镜像仓库地址。

```
```shell
export imageRegistry={你的镜像仓库}
```

```shell
helm upgrade koral . \
-n koral-system \
-f ./bak.yaml \
--set global.imageRegistry=$imageRegistry
```

```

# 安全扫描类型

在 Kubernetes (简称 K8s) 环境中，安全扫描是确保集群安全性的关键措施之一。其中，合规性扫描（基于 CIS Benchmark）、权限扫描（基于 kube-audit 审计功能）、漏洞扫描（基于 kube-hunter）是三种常见且重要的安全扫描手段：

- 合规性扫描：基于 [CIS Benchmark](#) 对集群节点进行安全扫描。CIS Benchmark 是一套全球公认的最佳实践标准，为 Kubernetes 集群提供了详细的安全配置指南和自动化检查工具（如 Kube-Bench），帮助组织确保其 K8s 集群符合安全基线要求，保护系统和数据免受威胁。
- 权限扫描：基于 kube-audit 审计功能。权限扫描主要解决集群访问控制和操作透明度的问题。通过审计日志，集群管理员能够追溯集群资源的访问历史，识别异常行为，

如未经授权的访问、敏感数据的泄露、有安全漏洞的操作记录等。这对于故障排查、安全事件响应以及满足合规性要求至关重要。此外，权限扫描还可以帮助组织发现潜在的权限滥用问题，及时采取措施防止安全事件的发生。

- 漏洞扫描：基于 kube-hunter，主要解决 Kubernetes 集群中存在的已知漏洞和配置错误问题。kube-hunter 通过模拟攻击行为，能够识别集群中可被恶意利用的漏洞，如未授权访问、暴露的服务和 API 端点、配置错误的角色和绑定策略等。特别地，kube-hunter 能够识别并报告 CVE 漏洞，这些漏洞如果被恶意利用，可能导致数据泄露、服务中断等严重后果。[CVE 漏洞](#)是由国际知名的安全组织如 MITRE 所定义和维护的，CVE 数据库为软件和固件中的已知漏洞提供了唯一标识符，成为全球安全社区共同遵循的标准。kube-hunter 通过利用 CVE 数据库中的信息，能够帮助用户快速识别并响应 Kubernetes 集群中的安全威胁。

## 合规性扫描

合规性扫描的对象是集群节点。扫描结果中会列出扫描项以及扫描结果，并针对未通过的扫描项给出修复建议。有关扫描时用到的具体安全规则，可参考 [CIS Kubernetes Benchmark](#) 检查不同类型的节点时，扫描的侧重点有所不同。

- 扫描控制平面节点（Controller）
  - 关注 API Server 、 controller-manager 、 scheduler 、 kubelet 等系统组件的安全性
  - 检查 Etcd 数据库的安全配置
  - 检查集群身份验证机制、授权策略和网络安全配置是否符合安全标准
- 扫描工作节点（Worker）

- 检查 kubelet、Docker 等容器运行时的配置否符合安全标准
- 检查容器镜像是否经过信任验证
- 检查节点的网络安全配置否符合安全标准

!!! tip

使用合规性扫描时，需要先创建[扫描配置](cis/config.md)，然后基于该配置创建[扫描策略](cis/policy.md)。执行扫描策略之后，可以[查看扫描报告](cis/report.md)。

## 权限扫描

权限扫描侧重于权限问题引发的安全漏洞。权限扫描可以帮助用户识别 Kubernetes 集群中的安全威胁，标识哪些资源需要进行进一步的审查和保护措施。通过执行这些检查项，用户可以更清楚、更全面地了解自己的 Kubernetes 环境，确保集群环境符合 Kubernetes 的最佳实践和安全标准。

具体而言，权限扫描支持以下操作：

- 扫描集群中的所有节点的健康状态。
- 扫描集群组件的运行状况，如 **kube-apiserver**、**kube-controller-manager**、**kube-scheduler** 等。
- 扫描安全配置：检查 Kubernetes 的安全配置
  - API 安全：启用了不安全的 API 版本，是否设置了适当的 RBAC 角色和权限限制等
  - 容器安全：是否使用了不安全的 Image、是否开放了特权模式，是否设置了合适的安全上下文等
  - 网络安全：是否启用了合适的网络策略来限制流量，是否使用了 TLS 加密等
  - 存储安全：是否启用了适当的加密、访问控制等。
  - 应用程序安全：是否设置了必要的安全措施，例如密码管理、跨站脚本攻击

防御等。

- 提供警告和建议：建议集群管理员执行的安全最佳实践，例如定期轮换证书、使用强密码、限制网络访问等。

!!! tip

使用合规性扫描时，需要先创建扫描策略。执行扫描策略之后，可以查看扫描报告。详情可参考[\[安全扫描\]\(audit.md\)](#)。

## 漏洞扫描

漏洞扫描侧重于扫描潜在的恶意攻击和安全漏洞，例如远程代码执行、SQL 注入、XSS 攻击等，以及一些针对 Kubernetes 特定的攻击。最终的扫描报告会列出集群中存在的安全漏洞，并提出修复建议。

!!! tip

使用合规性扫描时，需要先创建扫描策略。执行扫描策略之后，可以查看扫描报告。详情可参考[\[漏洞扫描\]\(hunter.md\)](#)。

## 扫描配置

使用[合规性扫描](#)的第一步，就是先创建扫描配置。基于扫描配置再创建扫描策略、执行扫描策略，最后查看扫描结果。

### 创建扫描配置

创建扫描配置的步骤如下：

1. 在容器管理模块的首页左侧导航栏点击 **安全管理**。

安全管理

安全管理

2. 默认进入 **合规性扫描** 页面，点击 **扫描配置** 页签，然后在右上角点击 **创建扫描配**

置。

## 安全管理

### 安全管理

3. 填写配置名称、选择配置模板、按需勾选扫描项，最后点击 **确定**。

扫描模板：目前提供了两个模板。**kubeadm** 模板适用于一般情况下的 Kubernetes 集群。**daocloud** 模板在 **kubeadm** 模板基础上，结合 DCE 5.0 的平台设计忽略了不适用于 DCE 5.0 的扫描项。

## 安全管理

### 安全管理

## 查看扫描配置

在扫描配置页签下，点击扫描配置的名称，可以查看该配置的类型、扫描项数量、创建时间、配置模板，以及该配置启用的具体扫描项。

## 安全管理

### 安全管理

## 更新/删除扫描配置

扫描配置创建成功之后，可以根据需求更新配置或删除该配置。

在扫描配置页签下，点击配置右侧的  操作按钮：

- 选择 **编辑** 可以更新配置，支持更新描述、模板和扫描项。不可更改配置名称。
- 选择 **删除** 可以删除该配置。

## 安全管理

安全管理

# 扫描策略

## 创建扫描策略

[创建扫描配置](#)之后，可以基于配置创建扫描策略。

1. 在 **安全管理** -> **合规性扫描** 页面的 **扫描策略** 页签下，在右侧点击创建扫描策略。

创建扫描配置

创建扫描配置

2. 参考下列说明填写配置后，点击 **确定**。

- 集群：选择需要扫描哪个集群。可选的集群列表来自[容器管理](#)模块中接入或

创建的集群。如果没有想选的集群，可以去容器管理模块中[接入](#)或[创建](#)集群。

- 扫描配置：选择事先创建好的扫描配置。扫描配置规定了需要执行哪些具体的扫描项。

- 扫描类型：

- 立即扫描：在扫描策略创建好之后立即执行一次扫描，后续不可以自动/手动再次执行扫描。

- 定时扫描：通过设置扫描周期，自动按时重复执行扫描。

- 扫描报告保留数量：设置最多保留多少扫描报告。超过指定的保留数量时，从最早的报告开始删除。

创建扫描配置

创建扫描配置

## 更新/删除扫描策略

创建扫描策略之后，可以根据需要更新或删除扫描策略。

在 **扫描策略** 页签下，点击配置右侧的  操作按钮：

- 对于周期性的扫描策略：
  - 选择 **立即执行** 意味着，在周期计划之外立即再扫描一次集群
  - 选择 **禁用** 会中断扫描计划，直到点击 **启用** 才可以继续根据周期计划执行该扫描策略。
  - 选择 **编辑** 可以更新配置，支持更新扫描配置、类型、扫描周期、报告保留数量，不可更改配置名称和需要扫描的目标集群。
  - 选择 **删除** 可以删除该配置
- 对于一次性的扫描策略：仅支持 **删除** 操作。

创建扫描配置

创建扫描配置

## 扫描报告

执行扫描策略之后会自动生成扫描报告。您可以在线查看扫描报告或将其下载到本地查看。

- 下载查看扫描报告

**安全管理 -> 合规性扫描** 页面的 **扫描报告** 页签点击报告右侧的  操作按钮选择 **下载**。

### 报告列表截图

#### 报告列表截图

- 在线查看扫描报告

点击某个报告的名称，您可以在线查看 CIS 合规性扫描的报告内容。具体包括：

- 扫描的目标集群
- 使用的扫描策略和扫描配置
- 扫描开始时间
- 扫描项总数、通过数与未通过数
- 对于未通过的扫描项给出对应的修复建议
- 对于通过的扫描项给出更安全的操作建议

### 报告列表截图

#### 报告列表截图

## 权限扫描

为了使用[权限扫描](#)功能，需要先创建扫描策略，执行该策略之后会自动生成扫描报告以供查看。

## 创建扫描策略

1. 在容器管理模块的首页左侧导航栏点击 **安全管理**。

### 安全管理

#### 安全管理

2. 在左侧导航栏点击 **权限扫描**，点击 **扫描策略** 页签，在右侧点击 **创建扫描策略**。

## 安全管理

### 安全管理

3. 参考下列说明填写配置，最后点击 **确定** 即可。

- 集群：选择需要扫描哪个集群。可选的集群列表来自[容器管理](#)模块中接入或创建的集群。如果没有想选的集群，可以去容器管理模块中[接入](#)或[创建](#)集群。
- 扫描类型：
  - 立即扫描：在扫描策略创建好之后立即执行一次扫描，后续不可以自动/手动再次执行扫描。
  - 定时扫描：通过设置扫描周期，自动按时重复执行扫描。
- 扫描报告保留数量：设置最多保留多少扫描报告。超过指定的保留数量时，从最早的报告开始删除。

## 安全管理

### 安全管理

## 更新/删除扫描策略

创建扫描策略之后，可以根据需要更新或删除扫描策略。

在 **扫描策略** 页签下，点击配置右侧的  操作按钮：

- 对于周期性的扫描策略：
  - 选择 **立即执行** 意味着，在周期计划之外立即再扫描一次集群
  - 选择 **禁用** 会中断扫描计划，直到点击 **启用** 才可以继续根据周期计划执行该扫描策略。

- 选择 **编辑** 可以更新配置，支持更新扫描配置、类型、扫描周期、报告保留数量，不可更改配置名称和需要扫描的目标集群。
  - 选择 **删除** 可以删除该配置
- 对于一次性的扫描策略：仅支持 **删除** 操作。

创建扫描配置

创建扫描配置

## 查看扫描报告

1. 在 **安全管理** -> **权限扫描** -> **扫描报告** 页签下，点击报告名称

在报告右侧点击 **删除** 可以手动删除报告。

创建扫描配置

创建扫描配置

2. 查看扫描报告内容，包括：

- 扫描的目标集群
- 使用的扫描策略
- 扫描项总数、警告数、错误数
- 在周期性扫描策略生成的扫描报告中，还可以查看扫描频率
- 扫描开始的时间
- 检查详情，例如被检查的资源、资源类型、扫描结果、错误类型、错误详情

创建扫描配置

创建扫描配置

# 漏洞扫描

为了使用[漏洞扫描](#)功能，需要先创建扫描策略，执行该策略之后会自动生成扫描报告以供查看。

## 创建扫描策略

1. 在容器管理模块的首页左侧导航栏点击 **安全管理**。

安全管理

安全管理

2. 在左侧导航栏点击 **漏洞扫描**，点击 **扫描策略** 页签，在右侧点击 **创建扫描策略**。

安全管理

安全管理

3. 参考下列说明填写配置，最后点击 **确定** 即可。

- 集群：选择需要扫描哪个集群。可选的集群列表来自[容器管理](#)模块中接入或创建的集群。如果没有想选的集群，可以去容器管理模块中[接入](#)或[创建](#)集群。

- 扫描类型：

- 立即扫描：在扫描策略创建好之后立即执行一次扫描，后续不可以自动/手动再次执行扫描。

- 定时扫描：通过设置扫描周期，自动按时重复执行扫描。

- 扫描报告保留数量：设置最多保留多少扫描报告。超过指定的保留数量时，从最早的报告开始删除。

## 安全管理

### 安全管理

## 更新/删除扫描策略

创建扫描策略之后，可以根据需要更新或删除扫描策略。

在 **扫描策略** 页签下，点击配置右侧的  操作按钮：

- 对于周期性的扫描策略：
  - 选择 **立即执行** 意味着，在周期计划之外立即再扫描一次集群
  - 选择 **禁用** 会中断扫描计划，直到点击 **启用** 才可以继续根据周期计划执行该扫描策略。
  - 选择 **编辑** 可以更新配置，支持更新扫描配置、类型、扫描周期、报告保留数量，不可更改配置名称和需要扫描的目标集群。
  - 选择 **删除** 可以删除该配置
- 对于一次性的扫描策略：仅支持 **删除** 操作。

### 创建扫描配置

#### 创建扫描配置

## 查看扫描报告

1. 在 **安全管理** -> **权限扫描** -> **扫描报告** 页签下，点击报告名称

在报告右侧点击 **删除** 可以手动删除报告。

### 创建扫描配置

#### 创建扫描配置

2. 查看扫描报告内容，包括：

- 扫描的目标集群
- 使用的扫描策略
- 扫描频率
- 风险总数、高风险数、中风险数、低风险数
- 扫描时间
- 检查详情，例如漏洞 ID、漏洞类型、漏洞名称、漏洞描述等

创建扫描配置

创建扫描配置

## 离线升级安全管理模块

本页说明[下载安全管理模块](#)后，应该如何安装或升级。

!!! info

下述命令或脚本内出现的 `_dowl_` 字样是安全管理模块的内部开发代号。

### 从下载的安装包中加载镜像

您可以根据下面两种方式之一加载镜像，当环境中存在镜像仓库时，建议选择 `chart-syncer`

同步镜像到镜像仓库，该方法更加高效便捷。

#### 方式一：使用 `chart-syncer` 同步镜像

使用 `chart-syncer` 可以将您下载的安装包中的 `chart` 及其依赖的镜像包上传至安装器部署 DCE 时使用的镜像仓库和 `helm` 仓库。

首先找到一台能够连接镜像仓库和 `helm` 仓库的节点（如火种节点），在节点上创建

load-image.yaml 配置文件，填入镜像仓库和 helm 仓库等配置信息。

### 1. 创建 load-image.yaml

#### !!! note

该 YAML 文件中的各项参数均为必填项。

#### ==== “已添加 Helm repo”

若当前环境已安装 chart repo, chart-syncer 也支持将 chart 导出为 tgz 文件。

```
```yaml title="load-image.yaml"
source:
  intermediateBundlesPath: dowl # 使用 chart-syncer 之后 .tar.gz 包所在的路径
target:
  containerRegistry: 10.16.10.111 # 镜像仓库地址
  containerRepository: release.daocloud.io/dowl # 镜像仓库路径
repo:
  kind: HARBOR # Helm Chart 仓库类别
  url: http://10.16.10.111/chartrepo/release.daocloud.io # Helm 仓库地址
  auth:
    username: "admin" # 镜像仓库用户名
    password: "Harbor12345" # 镜像仓库密码
containers:
  auth:
    username: "admin" # Helm 仓库用户名
    password: "Harbor12345" # Helm 仓库密码
````
```

#### ==== “未添加 Helm repo”

若当前节点上未添加 helm repo, chart-syncer 也支持将 chart 导出为 tgz 文件，并存放在指定路径。

```
```yaml title="load-image.yaml"
source:
  intermediateBundlesPath: dowl # 使用 chart-syncer 之后 .tar.gz 包所在的路径
target:
  containerRegistry: 10.16.10.111 # 镜像仓库 url
  containerRepository: release.daocloud.io/dowl # 镜像仓库路径
repo:
  kind: LOCAL
  path: ./local-repo # chart 本地路径
containers:
  auth:
    username: "admin" # 镜像仓库用户名
```

```

password: "Harbor12345" # 镜像仓库密码
```

```

## 2. 执行同步镜像命令。

```
charts-syncer sync --config load-image.yaml
```

## 方式二：使用 Docker 或 containerd 加载镜像

解压并加载镜像文件。

### 1. 解压第一层压缩包。

```
tar xvf dowl.amd64.tar
```

解压成功后会得到 1 个新的压缩包：

- dowl.bundle.tar

### 2. 解压新的压缩包。

```
tar xvf dowl.bundle.tar
```

解压成功后会得到 3 个文件：

- hints.yaml
- images.tar
- original-chart

### 3. 从本地加载镜像到 Docker 或 containerd。

```

==== "Docker"
```
shell
docker load -i images.tar
```
==== "containerd"
```
shell
ctr -n k8s.io image import images.tar
```

```

### !!! note

每个 node 都需要做 Docker 或 containerd 加载镜像操作，  
加载完成后需要 tag 镜像，保持 Registry、Repository 与安装时一致。

## 升级

有两种升级方式。您可以根据前置操作，选择对应的升级方案：

### ==== “通过 helm repo 升级”

1. 检查安全管理 helm 仓库是否存在。

```
```shell
helm repo list | grep dowl
````
```

若返回结果为空或如下提示，则进行下一步；反之则跳过下一步。

```
```none
Error: no repositories to show
````
```

1. 添加安全管理的 helm 仓库。

```
```shell
helm repo add dowl http://{harbor url}/chartrepo/{project}
````
```

1. 更新安全管理的 helm 仓库。

```
```shell
helm repo update dowl
````
```

> helm 版本过低会导致失败，若失败，请尝试执行 helm update repo

1. 选择您想安装的安全管理版本（建议安装最新版本）。

```
```shell
helm search repo dowl/dowl --versions
````

```none
[root@master ~]# helm search repo dowl/dowl --versions
NAME          CHART VERSION APP VERSION DESCRIPTION
dowl/dowl    0.4.0      v0.4.0       A Helm chart for dowl
...
````
```

1. 备份 `--set` 参数。

在升级安全管理版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

```
```shell
helm get values dowl -n dowl-system -o yaml > bak.yaml
````
```

1. 更新 dowl crds (需要先解压并进入 original-chart 文件)

```
```shell
helm pull dowl/dowl --version 0.4.0 && tar -zxf dowl-0.4.0.tgz
kubectl apply -f dowl/crds
````
```

1. 执行 `helm upgrade`。

升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 字段为当前使用的镜像仓库地址  
。

```
```shell
export imageRegistry={你的镜像仓库}
````
```

```
```shell
helm upgrade dowl dowl/dowl \
-n dowl-system \
-f ./bak.yaml \
--set global.imageRegistry=$imageRegistry \
--version 0.4.0
````
```

### ==== “通过 chart 包升级”

1. 备份 `--set` 参数。

在升级安全管理版本之前，建议您执行如下命令，备份老版本的 `--set` 参数。

```
```shell
helm get values dowl -n dowl-system -o yaml > bak.yaml
````
```

1. 更新 dowl crds (需要先解压并进入 original-chart 文件)

```
```shell
kubectl apply -f ./crds
````
```

1. 执行 `helm upgrade`。

升级前建议您覆盖 bak.yaml 中的 `global.imageRegistry` 为当前使用的镜像仓库地址。

```
```shell
export imageRegistry={你的镜像仓库}
```

```shell
helm upgrade dowl . \
-n dowl-system \
-f ./bak.yaml \
--set global.imageRegistry=$imageRegistry
```
```

```

什么是 Falco

[Falco](#) 是一个云原生运行时安全工具，旨在检测应用程序中的异常活动，可用于监控 Kubernetes 应用程序和内部组件的运行时安全性。仅需为 Falco 撰写一套规则，即可持续监测并监控容器、应用、主机及网络的异常活动。

Falco 能检测到什么？

Falco 可对任何涉及 Linux 系统调用的行为进行检测和报警。Falco 的警报可以通过使用特定的系统调用、参数以及调用进程的属性来触发。例如，Falco 可以轻松检测到包括但不限于以下事件：

- Kubernetes 中的容器或 pod 内正在运行一个 shell。
- 容器以特权模式运行，或从主机挂载敏感路径，如 /proc。
- 一个服务器进程正在生成一个意外类型的子进程。
- 意外读取一个敏感文件，如 /etc/shadow。
- 一个非设备文件被写到 /dev。

- 一个标准的系统二进制文件，如 ls，正在进行一个外向的网络连接。
- 在 Kubernetes 集群中启动一个有特权的 Pod。

关于 Falco 附带的更多默认规则，请参考 [Rules 文档](#)。

什么是 Falco 规则？

Falco 规则定义 Falco 应监视的行为及事件；可以在 Falco 规则文件或通用配置文件撰写规则。有关编写、管理和部署规则的更多信息，请参阅 Falco [Rules](#)。

什么是 Falco 警报？

警报是可配置的下游操作，可以像记录日志一样简单，也可以像 STDOUT 向客户端传递 gRPC 调用一样复杂。有关配置、理解和开发警报的更多信息，请参阅 Falco [警报](#)。Falco 可以将警报发送至：

- 标准输出
- 一份文件
- 系统日志
- 生成的程序
- 一个 HTTP[s] 端点
- 通过 gRPC API 的客户端

Falco 由哪些部分组成？

Falco 由以下几个主要组件组成：

- 用户空间程序：CLI 工具，可用于与 Falco 交互。用户空间程序处理信号，解析来自 Falco 驱动的信息，并发送警报。

- 配置：定义 Falco 的运行方式、要断言的规则以及如何执行警报。有关详细信息，请参阅[配置](#)。
- Driver：一款遵循 Falco 驱动规范并发送系统调用信息流的软件。如果不安装驱动程序，将无法运行 Falco。目前，Falco 支持以下驱动程序：
 - 基于 C++ 库构建 libscap 的内核模块 libsinsp（默认）
 - 由相同模块构建的 BPF 探针
 - 用户空间检测有关详细信息，请参阅[Falco 驱动程序](#)。
- 插件：可用于扩展 falco libraries/falco 可执行文件的功能，扩展方式是通过添加新的事件源和从事件中提取信息的新字段。有关详细信息，请参阅[插件](#)。

安装 Falco

请确认您的集群已成功接入容器管理平台，然后执行以下步骤安装 Falco。

1. 从左侧导航栏点击容器管理—>集群列表，然后找到准备安装 Falco 的集群名称。

falco_cluster
falco_cluster

2. 在左侧导航栏中选择 Helm 应用 -> Helm 模板，找到并点击 Falco。

falco_helm-1
falco_helm-1

3. 在版本选择中选择希望安装的版本，点击安装。

falco-helm-2
falco-helm-2

4. 在安装界面，填写所需的安装参数。

falco_helm-3
falco_helm-3

在如上界面中，填写应用名称、命名空间、版本等。

```
falco_helm-4
falco_helm-4
```

在如上界面中，填写以下参数：

- Falco -> Image Settings -> Registry：设置 Falco 镜像的仓库地址，已经默认填写可用的在线仓库。如果是私有化环境，可修改为私有仓库地址。
- Falco -> Image Settings -> Repository：设置 Falco 镜像名。
- Falco -> Falco Driver -> Image Settings -> Registry：设置 Falco Driver 镜像的仓库地址，已经默认填写可用的在线仓库。如果是私有化环境，可修改为私有仓库地址。
- Falco -> Falco Driver -> Image Settings -> Repository：设置 Falco Driver 镜像名。
- Falco -> Falco Driver -> Image Settings -> Driver Kind：设置 Driver Kind，提供以下两种选择：
 1. ebpf：使用 ebpf 来检测事件，这需要 Linux 内核支持 ebpf，并启用 CONFIG_BPF_JIT 和 sysctl net.core.bpf_jit_enable=1。
 2. module：使用内核模块检测，支持有限的操作系统版本，参考 module 支持[系统版本](#)。
- Falco -> Falco Driver -> Image Settings -> Log Level：要包含在日志中的最小日志级别。
可选择值为：emergency, alert, critical, error, warning, notice, info, debug。
- Falco -> Falco Driver -> Image Settings -> Registry：设置 Falco Driver 镜像的仓库地址，已经默认填写可用的在线仓库。如果是私有化环境，可修改为私有仓库地址。

- Falco -> Falco Driver -> Image Settings -> Repository : 设置 Falco Driver 镜像名。
 - Falco -> Falco Driver -> Image Settings -> Driver Kind : 设置 Driver Kind , 提供以下两种选择 :
 1. ebpf : 使用 ebpf 来检测事件 , 这需要 Linux 内核支持 ebpf , 并启用 CONFIG_BPF_JIT 和 sysctl net.core.bpf_jit_enable=1。
 2. module : 使用内核模块检测 , 支持有限的操作系统版本 , 参考 module 支持[系统版本](#)。
 - Falco -> Falco Driver -> Image Settings -> Log Level : 要包含在日志中的最小日志级别。

可选择值为 : emergency、alert、critical、error、warning、notice、info、debug。
5. 点击右下角确定按钮即可完成安装。

Falco-exporter

[Falco-exporter](#) 是一个 Falco 输出事件的 Prometheus Metrics 导出器。

Falco-exporter 会部署为 Kubernetes 集群上的守护进程集。如果集群中已安装并运行 Prometheus , Prometheus 将自动发现 Falco-exporter 提供的指标。

安装 Falco-exporter

本页介绍如何安装 Falco-exporter 组件。

!!! note

在安装使用 Falco-exporter 之前，需要[安装](./falco-install.md)并运行 Falco，并启用 gRPC 输出（默认通过 Unix 套接字启用）。

关于启用 gRPC 输出的更多信息，可参阅[在 Falco Helm Chart 中启用 gRPC 输出](<https://github.com/falcosecurity/charts/tree/master/falco#enabling-grpc>)。

请确认您的集群已成功接入容器管理平台，然后执行以下步骤安装 Falco-exporter。

1. 从左侧导航栏点击容器管理—>集群列表，然后找到准备安装 Falco-exporter 的集群名

称。

falco_cluster

falco_cluster

2. 在左侧导航栏中选择 Helm 应用 -> Helm 模板，找到并点击 falco-exporter。

falco-exporter_helm-1

falco-exporter_helm-1

3. 在版本选择中选择希望安装的版本，点击安装。

falco-exporter_helm-2

falco-exporter_helm-2

4. 在安装界面，填写所需的安装参数。

falco-exporter_helm-3

falco-exporter_helm-3

在如上界面中，填写应用名称、命名空间、版本等。

falco-exporter_helm-4

falco-exporter_helm-4

在如上界面中，填写以下参数：

– Falco Prometheus Exporter -> Image Settings -> Registry：设置 falco-exporter 镜

像的仓库地址，已经默认填写可用的在线仓库。如果是私有化环境，可修

改为私有仓库地址。

– Falco Prometheus Exporter -> Prometheus ServiceMonitor Settings -> Repository：

设置 falco-exporter 镜像名。

– Falco Prometheus Exporter -> Prometheus ServiceMonitor Settings -> Install

ServiceMonitor：安装 Prometheus Operator 服务监视器，默认开启。

– Falco Prometheus Exporter -> Prometheus ServiceMonitor Settings -> Scrape

Interval : 用户自定义的间隔；如果未指定，则使用 Prometheus 默认间

隔。

- Falco Prometheus Exporter -> Prometheus ServiceMonitor Settings -> Scrape

Timeout : 用户自定义的抓取超时时间；如果未指定，则使用 Prometheus

默认的抓取超时时间。

falco-exporter_helm-4

falco-exporter_helm-4

falco-exporter_helm-4

falco-exporter_helm-4

在如上界面中，填写以下参数：

- Falco Prometheus Exporter -> Prometheus prometheusRules -> Install

prometheusRules : 创建 PrometheusRules，对优先事件发出警报，**默认开启**。

- Falco Prometheus Exporter -> Prometheus prometheusRules -> Alerts settings : 警

报设置，为不同级别的日志事件设置警报是否启用、警报的间隔时间、警报的阈值。

5. 点击右下角确定按钮即可完成安装。

容器管理权限说明

容器管理权限基于全局权限管理以及 Kubernetes RBAC 权限管理打造的多维度权限管理体系。

支持集群级、命名空间级的权限控制，帮助用户便捷灵活地对租户下的 IAM 用户、

用户组（用户的集合）设定不同的操作权限。

集群权限

集群权限基于 Kubernetes RBAC 的 ClusterRolebinding 授权，集群权限设置可让用户/用户

组具备集群相关权限。 目前的默认集群角色为 **Cluster Admin** (不具备集群的创建、删除权限)。

Cluster Admin

Cluster Admin 具有以下权限：

- 可管理、编辑、查看对应集群
- 管理、编辑、查看 命名空间下的所有工作负载及集群内所有资源
- 可授权用户为集群内角色 (Cluster Admin、NS Admin、NS Editor、NS Viewer)

该集群角色的 YAML 示例如下：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    kpanda.io/creator: system
  creationTimestamp: "2022-06-16T09:42:49Z"
  labels:
    iam.kpanda.io/role-template: "true"
    name: role-template-cluster-admin
  resourceVersion: "15168"
  uid: f8f86d42-d5ef-47aa-b284-097615795076
rules:
- apiGroups:
  - '*'
  resources:
  - '*'
  verbs:
  - '*'
- nonResourceURLs:
  - '*'
  verbs:
  - '*'
```

命名空间权限

命名空间权限是基于 Kubernetes RBAC 能力的授权，可以实现不同的用户/用户组对命名空间下的资源具有不同的操作权限(包括 Kubernetes API 权限)，详情可参考：[Kubernetes RBAC](#)。目前容器管理的默认角色为：NS Admin、NS Editor、NS Viewer。

NS Admin

NS Admin 具有以下权限：

- 可查看对应命名空间
- 管理、编辑、查看 命名空间下的所有工作负载，及自定义资源
- 可授权用户为对应命名空间角色 (NS Editor、NS Viewer)

该集群角色的 YAML 示例如下：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    kpanda.io/creator: system
  creationTimestamp: "2022-06-16T09:42:49Z"
  labels:
    iam.kpanda.io/role-template: "true"
    name: role-template-ns-admin
    resourceVersion: "15173"
    uid: 69f64c7e-70e7-4c7c-a3e0-053f507f2bc3
rules:
- apiGroups:
  - '*'
  resources:
  - '*'
  verbs:
  - '*'
- nonResourceURLs:
  - '*'
  verbs:
  - '*'
```

NS Editor

NS Editor 具有以下权限：

- 可查看对应有权限的命名空间
- 管理、编辑、查看 命名空间下的所有工作负载

??? note “[点击查看集群角色的 YAML 示例](#)”

```
```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
 annotations:
 kpanda.io/creator: system
 creationTimestamp: "2022-06-16T09:42:50Z"
 labels:
 iam.kpanda.io/role-template: "true"
 name: role-template-ns-edit
 resourceVersion: "15175"
 uid: ca9e690e-96c0-4978-8915-6e4c00c748fe
rules:
- apiGroups:
 - ""
 resources:
 - configmaps
 - endpoints
 - persistentvolumeclaims
 - persistentvolumeclaims/status
 - pods
 - replicationcontrollers
 - replicationcontrollers/scale
 - serviceaccounts
 - services
 - services/status
 verbs:
 - '*'
- apiGroups:
 - ""
 resources:
 - bindings
 - events
 - limitranges
```

- namespaces/status
- pods/log
- pods/status
- replicationcontrollers/status
- resourcequotas
- resourcequotas/status
- verbs:
  - '\*'
- apiGroups:
  - ""
- resources:
  - namespaces
- verbs:
  - '\*'
- apiGroups:
  - apps
- resources:
  - controllerrevisions
  - daemonsets
  - daemonsets/status
  - deployments
  - deployments/scale
  - deployments/status
  - replicasetss
  - replicasetss/scale
  - replicasetss/status
  - statefulsets
  - statefulsets/scale
  - statefulsets/status
- verbs:
  - '\*'
- apiGroups:
  - autoscaling
- resources:
  - horizontalpodautoscalers
  - horizontalpodautoscalers/status
- verbs:
  - '\*'
- apiGroups:
  - batch
- resources:
  - cronjobs
  - cronjobs/status
  - jobs

- jobs/status
- verbs:
  - '\*'
- apiGroups:
  - extensions
- resources:
  - daemonsets
  - daemonsets/status
  - deployments
  - deployments/scale
  - deployments/status
  - ingresses
  - ingresses/status
  - networkpolicies
  - replicaset
  - replicaset/scale
  - replicaset/status
  - replicationcontrollers/scale
- verbs:
  - '\*'
- apiGroups:
  - policy
- resources:
  - poddisruptionbudgets
  - poddisruptionbudgets/status
- verbs:
  - '\*'
- apiGroups:
  - networking.k8s.io
- resources:
  - ingresses
  - ingresses/status
  - networkpolicies
- verbs:
  - '\*'

```

NS Viewer

NS Viewer 具有以下权限：

- 可查看对应命名空间

- 可查看对应命名空间下的所有工作负载，及自定义资源

??? note “点击查看集群角色的 YAML 示例”

```
```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
 annotations:
 kpanda.io/creator: system
 creationTimestamp: "2022-06-16T09:42:50Z"
 labels:
 iam.kpanda.io/role-template: "true"
 name: role-template-ns-view
 resourceVersion: "15183"
 uid: 853888fd-6ee8-42ac-b91e-63923918baf8
rules:
- apiGroups:
 - ""
 resources:
 - configmaps
 - endpoints
 - persistentvolumeclaims
 - persistentvolumeclaims/status
 - pods
 - replicationcontrollers
 - replicationcontrollers/scale
 - serviceaccounts
 - services
 - services/status
 verbs:
 - get
 - list
 - watch
- apiGroups:
 - ""
 resources:
 - bindings
 - events
 - limitranges
 - namespaces/status
 - pods/log
 - pods/status
 - replicationcontrollers/status
```

```
- resourcequotas
- resourcequotas/status
verbs:
- get
- list
- watch
- apiGroups:
 - ""
resources:
- namespaces
verbs:
- get
- list
- watch
- apiGroups:
 - apps
resources:
- controllerrevisions
- daemonsets
- daemonsets/status
- deployments
- deployments/scale
- deployments/status
- replicasesets
- replicasesets/scale
- replicasesets/status
- statefulsets
- statefulsets/scale
- statefulsets/status
verbs:
- get
- list
- watch
- apiGroups:
 - autoscaling
resources:
- horizontalpodautoscalers
- horizontalpodautoscalers/status
verbs:
- get
- list
- watch
- apiGroups:
 - batch
```

```
resources:
- cronjobs
- cronjobs/status
- jobs
- jobs/status
verbs:
- get
- list
- watch
- apiGroups:
- extensions
resources:
- daemonsets
- daemonsets/status
- deployments
- deployments/scale
- deployments/status
- ingresses
- ingresses/status
- networkpolicies
- replicaset
- replicaset/scale
- replicaset/status
- replicationcontrollers/scale
verbs:
- get
- list
- watch
- apiGroups:
- policy
resources:
- poddisruptionbudgets
- poddisruptionbudgets/status
verbs:
- get
- list
- watch
- apiGroups:
- networking.k8s.io
resources:
- ingresses
- ingresses/status
- networkpolicies
verbs:
```

```
- get
- list
- watch
```
```

权限 FAQ

1. 全局权限和容器管理权限管理的关系？

答：全局权限仅授权为粗粒度权限，可管理所有集群的创建、编辑、删除；而对于细粒度的权限，如单个集群的管理权限，单个命名空间的管理、编辑、删除权限，需要基于 Kubernetes RBAC 的容器管理权限进行实现。一般权限的用户仅需要在容器管理中进行授权即可。

2. 目前仅支持四个默认角色，后台自定义角色的 **RoleBinding** 以及 **ClusterRoleBinding** (Kubernetes 细粒度的 RBAC) 是否也能生效？

答：目前自定义权限暂时无法通过图形界面进行管理，但是通过 kubectl 创建的权限规则同样能生效。

集群和命名空间授权

容器管理基于全局权限管理及全局用户/用户组管理实现授权，如需为用户授予容器管理的最高权限（可以创建、管理、删除所有集群），请参见[什么是用户与访问控制](#)。

前提条件

给用户/用户组授权之前，请完成如下准备：

- 已在全局管理中创建了待授权的用户/用户组，请参考[用户](#)。
- 仅 [Kpanda Owner](#) 及当前集群的 [Cluster Admin](#) 具备集群授权能力。详情可参考[权限](#)

[说明](#)。

- 仅 [Kpanda Owner](#)、当前集群的 [Cluster Admin](#)，当前命名空间的 [NS Admin](#) 具备命名空间授权能力。

集群授权

1. 用户登录平台后，点击左侧菜单栏 **容器管理** 下的 **权限管理**，默认位于 **集群权限** 页签。

[集群权限](#)

[集群权限](#)

2. 点击 **添加授权** 按钮。

[添加授权](#)

[添加授权](#)

3. 在 **添加集群权限** 页面中，选择目标集群、待授权的用户/用户组后，点击 **确定**。

目前仅支持的集群角色为 **Cluster Admin**，详情权限可参考[权限说明](#)。如需要给多个用户/用户组同时进行授权，可点击 **添加用户权限** 进行多次添加。

[添加集群权限](#)

[添加集群权限](#)

4. 返回集群权限管理页面，屏幕出现消息：**添加集群权限成功**。

[添加成功](#)

[添加成功](#)

命名空间授权

1. 用户登录平台后，点击左侧菜单栏 **容器管理** 下的 **权限管理**，点击 **命名空间权限** 页签。

命名空间权限

命名空间权限

2. 点击 **添加授权** 按钮。在 **添加命名空间权限** 页面中，选择目标集群、目标命名空间，以及待授权的用户/用户组后，点击 **确定**。

目前支持的命名空间角色为 NS Admin、NS Editor、NS Viewer，详情权限可参考[权限说明](#)

明。如需给多个用户/用户组同时进行授权，可点击 **添加用户权限** 进行多次添加。

点击 **确定** 完成权限授权。

添加命名空间权限

添加命名空间权限

3. 返回命名空间权限管理页面，屏幕出现消息：**添加集群权限成功**。

添加成功

添加成功

!!! tip

后续如需删除或编辑权限，可点击列表右侧的 或 ，选择 **编辑** 或 **删除**。

![编辑或删除](https://docs.daocloud.io/daocloud-docs-images/docs/kpanda/images/perm08.png)

增加 Kpanda 内置角色权限点

*[Kpanda]: 容器管理的开发代号

过去 Kpanda 内置角色的权限点 (rbac rules) 都是提前预定义好的且用户无法修改，因为

以前修改内置角色的权限点之后也会被 Kpanda 控制器还原成预定义的权限点。为了支持更加灵活的权限配置，满足对系统角色的自定义需求，目前 Kpanda 支持为内置系统角色（cluster admin、ns admin、ns editor、ns viewer）修改权限点。以下示例演示如何新增 ns-viewer 权限点，尝试增加可以删除 Deployment 的权限。其他权限点操作类似。

前提条件

- 适用于容器管理 v0.27.0 及以上版本。
- 已接入 Kubernetes 集群或者已创建 Kubernetes 集群，且能够访问集群的 UI 界面。
- 已完成一个[命名空间的创建](#)、[用户的创建](#)，并为用户授予 [NS Viewer](#)，详情可参考[命名空间授权](#)。

!!! note

- 只需在 Global Cluster 增加权限点，Kpanda 控制器会把 Global Cluster 增加的权限点同步到所有接入子集群中，同步需一段时间才能完成
- 只能在 Global Cluster 增加权限点，在子集群新增的权限点会被 Global Cluster 内置角色权限点覆盖
- 只支持使用固定 Label 的 ClusterRole 追加权限，不支持替换或者删除权限，也不能使用 role 追加权限，内置角色跟用户创建的 ClusterRole Label 对应关系如下

```
```output
cluster-admin: rbac.kpanda.io/role-template-cluster-admin: "true"
cluster-edit: rbac.kpanda.io/role-template-cluster-edit: "true"
cluster-view: rbac.kpanda.io/role-template-cluster-view: "true"
ns-admin: rbac.kpanda.io/role-template-ns-admin: "true"
ns-edit: rbac.kpanda.io/role-template-ns-edit: "true"
ns-view: rbac.kpanda.io/role-template-ns-view: "true"
```

```

操作步骤

1. 使用 admin 或者 cluster admin 权限的用户[创建无状态负载](#)

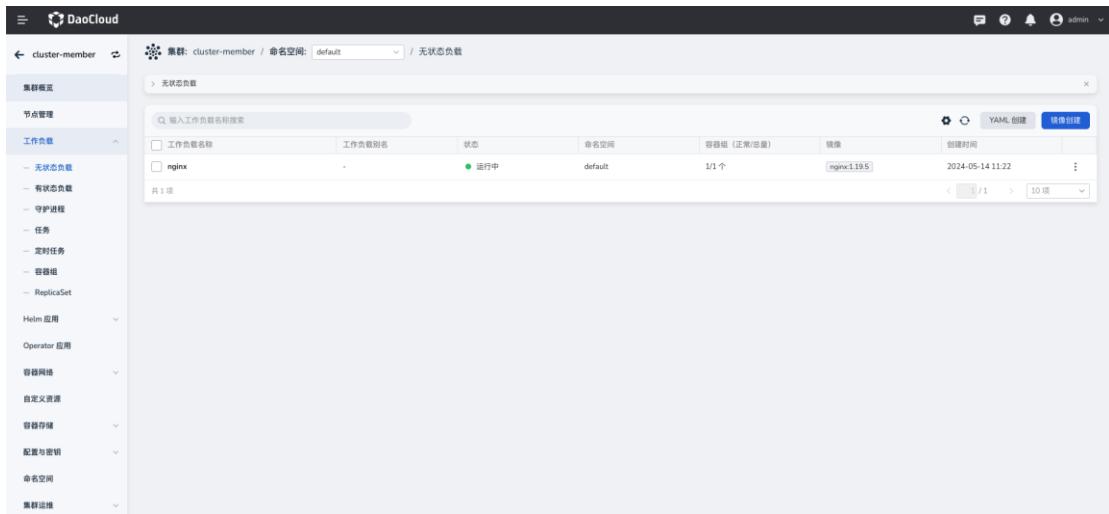


image-20240514112742395

2. 授权 ns-viewer，用户有该 namespace ns-view 权限

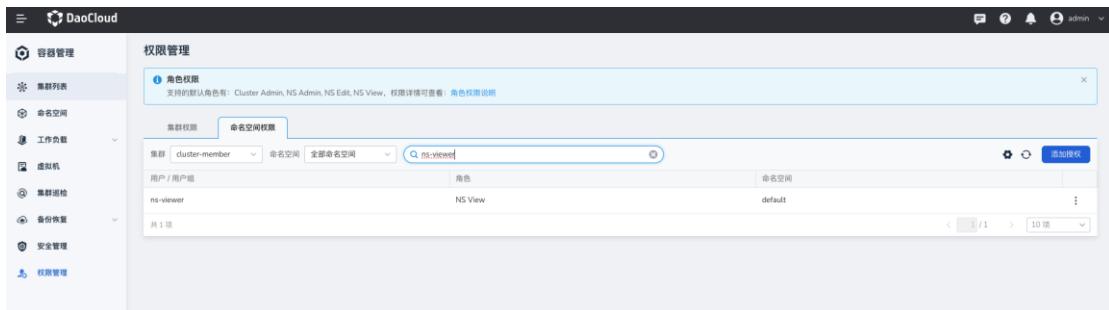


image-20240514113009311

3. 切换登录用户为 ns-viewer，打开控制台获取 ns-viewer 用户对应的 token，使用 curl

请求删除上述的 deployment nginx，发现无删除权限

```
[root@master-01 ~]# curl -k -X DELETE 'https://${URL}/apis/kpanda.io/v1alpha1/clusters/cluster-member/namespaces/default/deployments/nginx' -H 'authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldeIiwi2lkIiA6ICJOU044MG9BclBRMzUwZ2VVU2ZyNy1xMEREVWY4MmEtZmJqR05uRE1sd1lFIn0.eyJleHAiOiE3MTU3NjY1NzksImlhdCI6MTcxNTY4MDE3OSwiYXY0aF90aW1lIjoxNzE1NjgwMTc3LCJqdGkiOiIxZjI3MzJlNC1jYjFhLTQ4OTktYjBiZC1iN2IxZWY1MzAxNDEiLCJpc3MiOiJodHRwczovLzEwLjYuMjAxLjIwMTozMDE0Ny9hdXRoL3JlYWxty9naGlwcG8iLCJhdWQiOjF2ludGVybmfFsLWdoaXByIsInN1YiI6ImMxZmMzM2ViLTAwZGUtNDFiYS05ZTllWE5OGU2OGM0MmVmMCIsInR5cCI6IkIiEliwiYXpwIjoiX19pbnRlcmt5hbC1naGlwcG8iLCJzZXNzaW9uX3N0YXRlIjoiMGJjZWRjZTctMTliYS00NmU1LTkwYmUtOTliMWY2MWEyNzI0IiwiYXRfaGFzaCI6IJhTHoyQjlKQ2FNc1RrbGVMR3V6blEiLCJhY3liOiIwIiwi2lkIjoiMGJjZWRjZTctMTliYS00NmU1LTkwYmUtOTliMWY2MWEyNzI0IiwiZW1haWxfdmVyaWZpZWQiOmZhbHNlLCJncm91cHMiOltdLCJwcmVmZXJyZWRfdXNlcm5hbWUiOjJucy12aWV3ZXiiLCJsb2NhGUiOifQ.As2ipMjfVzvgONAGlc9RnqOd3zMwAj82VXlcqcR74ZK9tAq3Q4ruQ1a6WuIfqiq8Kq4F77ljwwzYUuunfBli2zhU2II8zyxVhLoCEBu4pBVBD_oJyUycXuNa6HfQGnl36E1
```

```
M7-_QG8b-_T51wFxxVb5b7SEDE1AvIf54NA1Ar-rhDmGRdOK1c9CohQcS00ab52MD3IPi
FFZ8_Iljnii-RpXKZoTjdcULJVn_uZNk_SzSUK-7MVWmPBK15m6sNktOMSf0pCObKWR
qHd15JSe-2aA2PKBo1jBH3tHbOgZyMPdsLI0QdmEnKB5FiiOeMpwn_oHnT6IjT-BZlB18
VkB8rA'
{"code":7,"message":"[RBAC] delete resources(deployments: nginx) is forbidden for user
(ns-viewer) in cluster(cluster-member)","details":[]}[root@master-01 ~]#
[root@master-01 ~]#
```

4. 在全局服务集群上创建如下 ClusterRole :

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: append-ns-view # (1)!
  labels:
    rbac.kpanda.io/role-template-ns-view: "true" # (2)!
rules:
  - apiGroups: [ "apps" ]
    resources: [ "deployments" ]
    verbs: [ "delete" ]
```

1. 此字段值可任意指定，只需不重复且符合 Kubernetes 资源名称规则要求

2. 注意给不同的角色添加权限时应打上不同的 label

5. 等待 Kpanda 控制器添加用户创建权限到内置角色 ns-viewer 中，可查看对应内置角

色如是否有上一步新增的权限点

```
[root@master-01 ~]# kubectl get clusterrole role-template-ns-view -oyaml|grep deployments
-C 10|tail -n 6
- apiGroups:
  - apps
  resources:
  - deployments
  verbs:
  - delete
```

6. 再次使用 curl 请求删除上述的 deployment nginx，这次成功删除了。也就是说，

ns-viewer 成功新增了删除 Deployment 的权限。

```
[root@master-01 ~]# curl -k -X DELETE 'https://${URL}/apis/kpanda.io/v1alpha1/clusters/
cluster-member/namespaces/default/deployments/nginx' -H 'authorization: Bearer eyJhbGci
OiJSUzI1NiIsInR5cCIgOiAiSldeIiwia2lkIiA6ICJOU044MG9BclBRMzUwZ2VVU2ZyNy1x
MEREVWY4MmEtZmJqR05uRE1sd1lFIn0.eyJleHAiOjE3MTU3NjY1NzksImhdCI6MTcx
NTY4MDE3OSwiYXV0aF90aW1lIjoxNzE1NjgwMTc3LCJqdGkiOiIxZjI3MzJlNC1jYjFhL
```

TQ4OTktYjBiZC1iN2IxZWY1MzAxNDEiLCJpc3MiOiJodHRwczovLzEwLjYuMjAxLjIwMT0zMDE0Ny9hdXRoL3JIYWxtcy9naGlwcG8iLCJhdWQiOiJfX2ludGVybmFsLWdoaXBwbyIsInN1YiI6ImMxZmMxM2ViLTAwZGUtNDFiYS05ZTllWE5OGU2OGM0MmVmMCIsInR5cCI6IkIiEliwiYXpwIjoiX19pbnRlc5hbC1naGlwcG8iLCJzZXNzaW9uX3N0YXRIIjoiMGJjZWRjZTctMTliYS00NmU1LTkwYmUtOTliMWY2MWEyNzI0IiwiYXRfaGFzaCI6IJhTHoyQjlKQ2FNc1RrbGVMR3V6blEiLCJhY3IiOiIwIiwi2lkIjoiMGJjZWRjZTctMTliYS00NmU1LTkwYmUtOTliMWY2MWEyNzI0IiwiZW1haWxfdmVyaWZpZWQiOmZhbHN1LCJncm91cHMiOltdLCJwcmVmZXJyZWRfdXNlcm5hbWUiOjucy12aWV3ZXIiLCJs2NhGUiOiIifQ.As2ipMjfVzvgONAGlc9RnqOd3zMwAj82VXlcqcR74ZK9tAq3Q4ruQ1a6WuIfqiq8Kq4F77ljwwzYUuunfBli2zhU2II8zyxVhLoCEBu4pBVBD_oJyUycXuNa6HfQGnl36E1M7-_QG8b-_T51wFxxVb5b7SEDE1AvIf54NAIAr-rhDmGRdOK1c9CohQcS00ab52MD3IPiFFZ8_Iljni-RpXKZoTjdcULJVn_uZNk_SzSUK-7MVWmPBK15m6sNktOMSf0pCObKWRqHd15JSe-2aA2PKBo1jBH3tHbOgZyMPdsLI0QdmEnKB5FiiOeMpwn_oHnT6IjT-BZlB18VkW8rA'

GPU 管理概述

本文介绍 DaoCloud 容器管理平台对 GPU 为代表的异构资源统一运维管理能力。

背景

随着 AI 应用、大模型、人工智能、自动驾驶等新兴技术的快速发展，企业面临着越来越多的计算密集型任务和数据处理需求。以 CPU 为代表的传统计算架构已无法满足企业日益增长的计算需求。此时，以 GPU 为代表的异构计算因在处理大规模数据、进行复杂计算和实时图形渲染方面具有独特的优势被广泛应用。

与此同时，由于缺乏异构资源调度管理等方面的经验和专业的解决方案，导致了 GPU 设备的资源利用率极低，给企业带来了高昂的 AI 生产成本。如何降本增效，提高 GPU 等异构资源的利用效率，成为了当前众多企业亟需跨越的一道难题。

GPU 能力介绍

DaoCloud 容器管理平台支持对 GPU、NPU 等异构资源进行统一调度和运维管理，充分释

放 GPU 资源算力，加速企业 AI 等新兴应用发展。 DaoCloud GPU 管理能力如下：

- 支持统一纳管 NVIDIA、华为昇腾、天数等国内外厂商的异构计算资源。
- 支持同一集群多卡异构调度，并支持集群 GPU 卡自动识别。
- 支持 NVIDIA GPU、vGPU、MIG 等 GPU 原生管理方案，并提供云原生能力。
- 支持单块物理卡切分给不同的租户使用，并支持对租户和容器使用 GPU 资源按照算力、显存进行 GPU 资源配额。
- 支持集群、节点、应用等多维度 GPU 资源监控，帮助运维人员管理 GPU 资源。
- 兼容 TensorFlow、PyTorch 等多种训练框架。

GPU Operator 介绍

同普通计算机硬件一样，NVIDIA GPU 卡作为物理硬件，必须安装 NVIDIA GPU 驱动后才能使用。为了降低用户在 Kubernetes 上使用 GPU 的成本，NVIDIA 官方提供了 NVIDIA GPU Operator 组件来管理使用 NVIDIA GPU 所依赖的各种组件。这些组件包括 NVIDIA 驱动程序（用于启用 CUDA）、NVIDIA 容器运行时、GPU 节点标记、基于 DCGM 的监控等。理论上来说用户只需要将 GPU 卡插在已经被 Kubernetes 所纳管的计算设备上，然后通过 GPU Operator 就能使用 NVIDIA GPU 的所有能力了。了解更多 NVIDIA GPU Operator 相关信息，请参考 [NVIDIA 官方文档](#)。如何部署请参考 [GPU Operator 离线安装](#)

NVIDIA GPU Operator 架构图：

NVIDIA GPU Operator 架构图

NVIDIA GPU Operator 架构图

GPU 支持矩阵

本页说明 DCE 5.0 支持的 GPU 及操作系统所对应的矩阵。

NVIDIA GPU

| GPU 厂商及
类型 | 支持 GPU
型号 | 适配的操作
系统 (在
线) | 推荐内核 | 推荐的操作
系统及内核 | 安装文档 |
|-------------------------------|-----------------------------|------------------------|---------------------------------------|---|--|
| NVIDIA GPU
(整卡
/vGPU) | NVIDIA Fermi
(2.1) 架构 | CentOS 7 | Kernel
3.10.0-123 ~
3.10.0-1160 | 操作系统 :
CentOS 7.9 ;
内核参考文
档建议使用
内核版本 :
3.10.0-1160 | GPU Operator
离线安装 |
| | | | | 操作系统对
应 Kernel 版
本 | |
| | NVIDIA
GeForce 400
系列 | CentOS 8 | Kernel
4.18.0-80 ~
4.18.0-348 | | |
| | NVIDIA
Quadro 4000
系列 | Ubuntu 20.04 | Kernel 5.4 | | |
| | NVIDIA Tesla
20 系列 | Ubuntu 22.04 | Kernel 5.19 | | |
| | NVIDIA
Ampere 架构
系列 | RHEL 7 | Kernel
3.10.0-123 ~
3.10.0-1160 | | |

| GPU 厂商及
类型 | 支持 GPU
型号 | 适配的操作
系统 (在
线) | 推荐内核 | 推荐的操作 | 安装文档 |
|---------------|---------------------------|------------------------|--|--|---|
| NVIDIA MIG | NVIDIA
Ampere 架构
系列 | (A100;A800;H
100) | RHEL 8
CentOS 7
Ubuntu 20.04
Ubuntu 22.04
RHEL 7
RHEL 8 | Kernel
4.18.0-80 ~
4.18.0-348
Kernel
3.10.0-123 ~
3.10.0-1160
Kernel
4.18.0-80 ~
4.18.0-348
Kernel 5.4
Kernel 5.19
Kernel
3.10.0-123 ~
3.10.0-1160
Kernel
4.18.0-80 ~
4.18.0-348 | 操作系统：
GPU Operator
离线安装
CentOS 7.9 ;
内核版本：
3.10.0-1160 |
| 昇腾 | Ascend 310 | Ubuntu 20.04 | Ubuntu 20.04 | 详情参考： | 操作系统：补充中 (300 |

昇腾 (Ascend) NPU

| GPU 厂商及
类型 | 支持 NPU
型号 | 适配的操作
系统 (在
线) | 推荐内核 | 推荐的操作 | 安装文档 |
|---------------|--------------|------------------------|--------------|-------|----------------|
| 昇腾 | Ascend 310 | Ubuntu 20.04 | Ubuntu 20.04 | 详情参考： | 操作系统：补充中 (300 |

| GPU 厂商及
类型 | 支持 NPU
型号 | 适配的操作
系统 (在
线) | 推荐内核
内核版本要
求 | 推荐的操作
系统及内核
内核版本 :
3.10.0-1160 | 安装文档 |
|-------------------|------------------|---|-----------------------------------|---|-------------------------------|
| (Ascend
310) | Ascend
310P ; | CentOS 7.6
CentOS 8.2
KylinV10SP1 | CentOS 7.9 ;
和 310P 驱
动文档) | | |
| | | 操作系统 | | | |
| | | openEuler 操
作系统 | | | |
| 昇腾 | Ascend 910B | Ubuntu 20.04
CentOS 7.6
CentOS 8.2
KylinV10SP1 | 详情参考内
核版本要求 | 操作系统 :
CentOS 7.9 ;
内核版本 :
3.10.0-1160 | 910 驱动文
档 |
| (Ascend
910) | | 操作系统 | | | |
| | | openEuler 操
作系统 | | | |

天数智芯 (Iluvatar) GPU

| GPU 厂商及
类型 | 支持的 GPU
型号 | 适配的操作
系统 (在
线) | 推荐内核 | 推荐的操作 | 安装文档 |
|----------------------------|---------------|------------------------|---|---|----------------------|
| 天数智芯
(Iluvatar
vGPU) | BI100 | CentOS 7 | Kernel
3.10.0-957.el
7.x86_64 ~
3.10.0-1160.4
2.2.el7.x86_6
4 | 操作系统 :
CentOS 7.9 ;
内核版本 :
3.10.0-1160 | 使用说明 |
| | MR100 ; | CentOS 8 | Kernel
4.18.0-80.el8.
x86_64 ~
4.18.0-305.19
.1.el8_4.x86_
64 | | |
| | | Ubuntu 20.04 | Kernel
4.15.0-20-gen
eric ~
4.15.0-160-ge
neric Kernel
5.4.0-26-gene
ric ~
5.4.0-89-gene
ric Kernel
5.8.0-23-gene
ric ~
5.8.0-63-gene
ric | | |
| | | Ubuntu 21.04 | Kernel
4.15.0-20-gen
eric ~
4.15.0-160-ge
neric Kernel
5.4.0-26-gene
ric ~
5.4.0-89-gene
ric Kernel | | |

| GPU 厂商及
类型 | 支持的 GPU
型号 | 适配的操作
系统 (在
线) | 推荐内核
5.8.0-23-gene
ric ~
5.8.0-63-gene
ric | 推荐的操作
Kernel 版本
大于等于
5.1 且小于
等于 5.10 | 安装文档 |
|------------------------|---------------|------------------------|--|--|------|
| openEuler
22.03 LTS | | | | | |

沐曦 (Metax) GPU

| GPU 厂商及
类型 | 支持的 GPU
型号 | 适配的操作
系统 (在
线) | 推荐内核 | 推荐的操作 | 安装文档 |
|-----------------------------|---------------|------------------------|------|--|------|
| 沐曦 Metax
(整卡
/vGPU) | 曦云 C500 | | | 沐曦 GPU
安装使用 | |

NVIDIA GPU 卡使用模式

NVIDIA 作为业内知名的图形计算供应商，为算力的提升提供了诸多软硬件解决方案，其中 NVIDIA 在 GPU 的使用方式上提供了如下三种解决方案：

整卡 (Full GPU)

整卡是指将整个 NVIDIA GPU 分配给单个用户或应用程序。在这种配置下，应用可以完全占用 GPU 的所有资源，并获得最大的计算性能。整卡适用于需要大量计算资源和内存的工作负载，如深度学习训练、科学计算等。

vGPU (Virtual GPU)

vGPU 是一种虚拟化技术，允许将一个物理 GPU 划分为多个虚拟 GPU，每个虚拟 GPU 分配给不同的虚拟机或用户。vGPU 使多个用户可以共享同一台物理 GPU，并在各自的虚拟环境中独立使用 GPU 资源。每个虚拟 GPU 可以获得一定的计算能力和显存容量。vGPU 适用于虚拟化环境和云计算场景，可以提供更高的资源利用率和灵活性。

MIG (Multi-Instance GPU)

MIG 是 NVIDIA Ampere 架构引入的一项功能，它允许将一个物理 GPU 划分为多个物理 GPU 实例，每个实例可以独立分配给不同的用户或工作负载。每个 MIG 实例具有自己的计算资源、显存和 PCIe 带宽，就像一个独立的虚拟 GPU。MIG 提供了更细粒度的 GPU 资源分配和管理，可以根据需求动态调整实例的数量和大小。MIG 适用于多租户环境、容器化应用程序和批处理作业等场景。

无论是在虚拟化环境中使用 vGPU，还是在物理 GPU 上使用 MIG，NVIDIA 为用户提供了更多的选择和优化 GPU 资源的方式。Daocloud 容器管理平台全面兼容了上述 NVIDIA 的能力特性，用户只需通过简单的界面操作，就能够获得全部 NVIDIA GPU 的计算能力，从而提高资源利用率并降低成本。

- **Single 模式**，节点仅在其所有 GPU 上公开单一类型的 MIG 设备，节点上的所有

GPU 必须：

- 属于同一个型号（例如 A100-SXM-40GB），只有同一型号 GPU 的 MIG Profile 才是一样的
 - 启用 MIG 配置，需要重启机器才能生效
 - 为在所有产品中公开“完全相同”的 MIG 设备类型，创建相同的 GI 和 CI
- **Mixed 模式**，节点在其所有 GPU 上公开混合 MIG 设备类型。请求特定的 MIG 设备类型需要设备类型提供的计算切片数量和内存总量。
- 节点上的所有 GPU 必须：属于同一产品线（例如 A100-SXM-40GB）
 - 每个 GPU 可启用或不启用 MIG，并且可以自由配置任何可用 MIG 设备类型的混合搭配。
 - 在节点上运行的 k8s-device-plugin 将：
 - 使用传统的 **nvidia.com/gpu** 资源类型公开任何不处于 MIG 模式的 GPU
 - 使用遵循架构 **nvidia.com/mig-g.gb** 的资源类型公开各个 MIG 设备

开启配置详情参考 [GPU Operator 离线安装](#)。

如何使用

您可以参考以下链接，快速使用 DaoCloud 关于 NVIDIA GPU 卡的管理能力。

- [NVIDIA GPU 整卡使用](#)
- [NVIDIA vGPU 使用](#)
- [NVIDIA MIG 使用](#)

GPU Operator 离线安装

DCE 5.0 预置了 Ubuntu22.04、Ubuntu20.04、CentOS 7.9 这三个操作系统的 Driver 镜像，驱动版本是 535.104.12；并且内置了各操作系统所需的 Toolkit 镜像，用户不再需要手动离线 Toolkit 镜像。

本文使用 AMD 架构的 CentOS 7.9 (3.10.0-1160) 进行演示。如需使用 Red Hat 8.4 部署，请参考[向火种节点仓库上传 Red Hat GPU Opreator 离线镜像和构建 Red Hat 8.4 离线 yum 源](#)。

前提条件

- 待部署 gpu-operator 的集群节点内核版本必须完全一致。节点所在的发行版和 GPU 卡型号在[GPU 支持矩阵](#)的范围内。
- 用户已经在平台上安装了 v0.12.0 及以上版本的[addon 离线包](#)（Addon v0.20 及以上版本内置 Ubuntu22.04、Ubuntu20.04、CentOS 7.9 三个操作系统）。
- 安装 gpu-operator 时选择 v23.9.0+2 及以上版本

操作步骤

参考如下步骤为集群安装 gpu-operator 插件。

- 登录平台，进入 **容器管理** -> **待安装 gpu-operator 的集群** -> 进入集群详情
- 在 **Helm 模板** 页面，选择 **全部仓库**，搜索 **gpu-operator**
- 选择 **gpu-operator**，点击 **安装**
- 参考下文参数配置，配置 **gpu-operator** 安装参数，完成 **gpu-operator** 的安装

参数配置

- **systemOS** : 选择机器的操作系统，当前内置了 Ubuntu 22.04、Ubuntu20.04、Centos7.9、other 四个选项，请正确的选择操作系统。

基本参数配置

- **名称** : 输入插件名称。
- **命名空间** : 选择将插件安装的命名空间。
- **版本** : 插件的版本，此处以 **v23.9.0+2** 版本为例。
- **失败删除** : 安装失败，则删除已经安装的关联资源。开启后，将默认同步开启 **就绪等待**。
 - **就绪等待** : 启用后，所有关联资源都处于就绪状态，才会标记应用安装成功。
 - **详情日志** : 开启后，将记录安装过程的详细日志。

高级参数配置

Operator 参数配置

- **InitContainer.image** : 配置 CUDA 镜像，推荐默认镜像：**nvidia/cuda**
- **InitContainer.repository** : CUDA 镜像所在的镜像仓库，默认为 **nvcr.m.daocloud.io** 仓库
- **InitContainer.version** : CUDA 镜像的版本，请使用默认参数

Driver 参数配置

- **Driver.enable** : 配置是否在节点上部署 NVIDIA 驱动，默认开启，如果您在使用 GPU

Operator 部署前，已经在节点上部署了 NVIDIA 驱动程序，请关闭。（若手动部署驱动程序需要关注 CUDA Toolkit 与 Toolkit Driver Version 的适配关系，通过 GPU operator 安装则无需关注）。

- **Driver.usePrecompiled**：启用预编译的 GPU 驱动
- **Driver.image**：配置 GPU 驱动镜像，推荐默认镜像：**nvidia/driver**。
- **Driver.repository**：GPU 驱动镜像所在的镜像仓库，默认为 nvidia 的 **nvcr.io** 仓库。
- **Driver.usePrecompiled**：开启预编译模式安装驱动。
- **Driver.version**：GPU 驱动镜像的版本，离线部署请使用默认参数，仅在线安装时需配置。不同类型操作系统的 Driver 镜像的版本存在如下差异，详情可参考：[Nvidia GPU Driver 版本](#)。如下不同操作系统的 Driver Version 示例：

!!! note

使用内置的操作系统版本无需修改镜像版本，其他操作系统版本请参考[向火种节点仓库上传镜像](./push_image_to_repo.md)。

注意版本号后无需填写 Ubuntu、CentOS、Red Hat 等操作系统名称，若官方镜像含有操作系统后缀，请手动移除。

- Red Hat 系统，例如 `525.105.17`
- Ubuntu 系统，例如 `535-5.15.0-1043-nvidia`
- CentOS 系统，例如 `525.147.05`

- **Driver.RepoConfig.ConfigMapName**：用来记录 GPU Operator 的离线 yum 源配置文件名称，当使用预置的离线包时，各类型的操作系统请参考如下的文档。
 - [构建 CentOS 7.9 离线 yum 源](#)
 - [构建 Red Hat 8.4 离线 yum 源](#)

Toolkit 配置参数

Toolkit.enable：默认开启，该组件让 conatainerd/docker 支持运行需要 GPU 的容器。

MIG 配置参数

详细配置方式请参考[开启 MIG 功能](#)

MigManager.Config.name : MIG 的切分配置文件名，用于定义 MIG 的 (GI, CI) 切分策略。

默认为 **default-mig-parted-config**。自定义参数参考[开启 MIG 功能](#)。

下一步操作

完成上述相关参数配置和创建后：

- 如果使用 **整卡模式**，[应用创建时可使用 GPU 资源](#)
- 如果使用 **vGPU 模式**，完成上述相关参数配置和创建后，下一步请完成 [vGPU Addon](#)

[安装](#)

- 如果使用 **MIG 模式**，并且需要给个别 GPU 节点按照某种切分规格进行使用，否则按照 MigManager.Config 中的 **default** 值进行切分。

- **single** 模式请给对应节点打上如下 Label：

```
kubectl label nodes {node} nvidia.com/mig.config="all-1g.10gb" --overwrite
```

- **mixed** 模式请给对应节点打上如下 Label：

```
kubectl label nodes {node} nvidia.com/mig.config="custom-config" --overwrite
```

切分后，应用可[使用 MIG GPU 资源](#)。

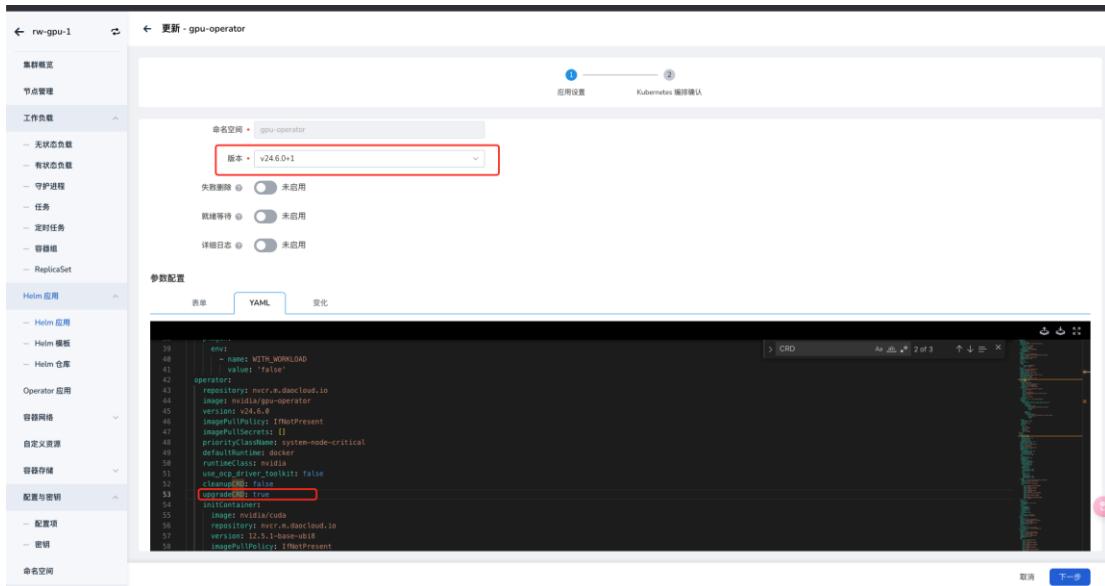
升级注意事项

1. 已知问题：gpu-operator 从 v23.9.0+3 版本升级到 v24.6.0+1 后，

gpu-operator-node-feature-discovery-master 一直处于 crash 状态。

2. 解决办法：在 Helm 应用页面，搜索 **gpu-operator**，点击操作栏中的 **更新** 按钮，

进入更新页面，选择版本为 24.6.0+1 后，将 upgradeCRD 设置为 true。



升级 GPU operator

向火种节点仓库上传 Red Hat GPU Operator 离线镜像

本文以 Red Hat 8.4 的 `nvcr.io/nvidia/driver: 525.105.17-rhel8.4` 离线驱动镜像为例，介绍如何向火种节点仓库上传离线镜像。

前提条件

1. 火种节点及其组件状态运行正常。
2. 准备一个能够访问互联网和火种节点的节点，且节点上已经完成 [Docker 的安装](#)。

操作步骤

在联网节点获取离线镜像

以下操作在联网节点上进行。

1. 在联网机器上拉取 nvcr.io/nvidia driver: 525.105.17-rhel8.4 离线驱动镜像：

```
docker pull nvcr.io/nvidia	driver:525.105.17-rhel8.4
```

2. 镜像拉取完成后，打包镜像为 nvidia-driver.tar 压缩包：

```
docker save nvcr.io/nvidia	driver:525.105.17-rhel8.4 > nvidia-driver.tar
```

3. 拷贝 nvidia-driver.tar 镜像压缩包到火种节点：

```
scp nvidia-driver.tar user@ip:/root
```

例如：

```
scp nvidia-driver.tar root@10.6.175.10:/root
```

推送镜像到火种节点仓库

以下操作在火种节点上进行。

1. 登录火种节点，将联网节点拷贝的镜像压缩包 nvidia-driver.tar 导入本地：

```
docker load -i nvidia-driver.tar
```

2. 查看刚刚导入的镜像：

```
docker images -a |grep nvidia
```

预期输出：

| | | | |
|-----------------------|--------------|------------|--------|
| nvcr.io/nvidia	driver | e3ed7dee73e9 | 1 days ago | 1.02GB |
|-----------------------|--------------|------------|--------|

3. 重新标记镜像，使其与远程 Registry 仓库中的目标仓库对应：

```
docker tag <image-name> <registry-url>/<repository-name>:<tag>
```

- <image-name> 是上一步 nvidia 镜像的名称，
- <registry-url> 是火种节点上 Registry 服务的地址，
- <repository-name> 是您要推送到的仓库名称，

- <tag> 是您为镜像指定的标签。

例如：

```
registry: docker tag nvcr.io/nvidia/driver 10.6.10.5/nvcr.io/nvidia/driver:525.105.17-rhel8.4
```

4. 将镜像推送到火种节点镜像仓库：

```
docker push {ip}/nvcr.io/nvidia/driver:525.105.17-rhel8.4
```

接下来

参考[构建 Red Hat 8.4 离线 yum 源](#)和[GPU Operator 离线安装](#)来为集群部署 GPU Operator。

构建 Red Hat 8.4 离线 yum 源

使用场景介绍

DCE 5 预置了 CentOS 7.9，内核为 3.10.0-1160 的 GPU operator 离线包。其它 OS 类型的节点或内核需要用户手动构建离线 yum 源。

本文介绍如何基于全局服务集群任意节点构建 Red Hat 8.4 离线 yum 源包，并在安装 Gpu Operator 时，通过 `RepoConfig.ConfigMapName` 参数来使用。

前提条件

1. 用户已经在平台上安装了 v0.12.0 及以上版本的 addon 离线包。
2. 待部署 GPU Operator 的集群节点 OS 必须为 Red Hat 8.4，且内核版本完全一致。
3. 准备一个能够和待部署 GPU Operator 的集群网络能够联通的文件服务器，如 nginx 或 minio。

4. 准备一个能够访问互联网、待部署 GPU Operator 的集群和文件服务器的节点，且节点上已经完成 [Docker 的安装](#)。 5. 全局服务集群的节点必须为 Red Hat 8.4 4.18.0-305.el8.x86_64。

操作步骤

本文以 Red Hat 8.4 4.18.0-305.el8.x86_64 节点为例，介绍如何基于全局服务集群任意节点构建 Red Hat 8.4 离线 yum 源包，并在安装 Gpu Operator 时，通过 RepoConfig.ConfigMapName 参数来使用。

下载火种节点中的 yum 源

以下操作在全局服务集群的 master 节点上执行。

1. 使用 ssh 或其它方式进入全局服务集群内任一节点执行如下命令：

```
cat /etc/yum.repos.d/extension.repo #查看 extension.repo 中的内容
```

预期输出如下：

[extension-0]

```
baseurl = http://10.5.14.200:9000/kubeany/redhat/$releasever/os/$basearch
gpgcheck = 0
name = kubeany extension 0
```

[extension-1]

```
baseurl = http://10.5.14.200:9000/kubeany/redhat-iso/$releasever/os/$basearch/AppStream
gpgcheck = 0
name = kubeany extension 1
```

[extension-2]

```
baseurl = http://10.5.14.200:9000/kubeany/redhat-iso/$releasever/os/$basearch/BaseOS
gpgcheck = 0
name = kubeany extension 2
```

2. 在 root 路径下新建一个名为 redhat-base-repo 的文件夹

```
mkdir redhat-base-repo
```

3. 下载 yum 源中的 rpm 包到本地：

```
yum install yum-utils
```

4. 下载 extension-1 中的 rpm 包：

```
reposync -p redhat-base-repo -n --repoid=extension-1
```

5. 下载 extension-2 中的 rpm 包：

```
reposync -p redhat-base-repo -n --repoid=extension-2
```

下载 **elfutils-libelf-devel-0.187-4.el8.x86_64.rpm** 包

以下操作在联网节点执行操作，在操作前，您需要保证联网节点和全局服务集群 master 节点间的网络联通性。

1. 在联网节点执行如下命令，下载 **elfutils-libelf-devel-0.187-4.el8.x86_64.rpm** 包：

```
wget https://rpmfind.net/linux/centos/8-stream/BaseOS/x86_64/os/Packages/elfutils-libelf-devel-0.187-4.el8.x86_64.rpm
```

2. 在当前目录下将 **elfutils-libelf-devel-0.187-4.el8.x86_64.rpm** 包传输至步骤一中的节点上：

```
scp elfutils-libelf-devel-0.187-4.el8.x86_64.rpm user@ip:~/redhat-base-repo/extension-2/Packages/
```

例如：

```
scp elfutils-libelf-devel-0.187-4.el8.x86_64.rpm root@10.6.175.10:~/redhat-base-repo/extension-2/Packages/
```

生成本地 yum repo

以下操作在步骤一中全局服务集群的 master 节点上执行。

1. 进入 yum repo 目录：

```
cd ~/redhat-base-repo/extension-1/Packages  
cd ~/redhat-base-repo/extension-2/Packages
```

2. 生成目录 repo 索引：

```
yum install createrepo -y # 若已安装 createrepo 可省略此步骤  
createrepo_c ./
```

至此，您已经生成了内核为 4.18.0-305.el8.x86_64 的离线的 yum 源：
redhat-base-repo。

将本地生成的 yum repo 上传至文件服务器

本操作示例采用的是 DCE5 火种节点内置的 Minio 作为文件服务器，用户可基于自身情况选择文件服务器。Minio 相关信息如下：

- 访问地址：http://10.5.14.200:9000 (一般为{火种节点 IP} + {9000 端口})
- 登录用户名：rootuser
- 登录密码：rootpass123

1. 在节点当前路径下，执行如下命令将节点本地 mc 命令行工具和 minio 服务器建立链接。

mc config host add minio 文件服务器访问地址 用户名 密码

例如：

```
mc config host add minio http://10.5.14.200:9000 rootuser rootpass123
```

预期输出如下：

Added `minio` successfully.

mc 命令行工具是 Minio 文件服务器提供的客户端命令行工具，详情请参考：[MinIO Client](#)。

2. 在节点当前路径下，新建一个名为 **redhat-base** 的存储桶(bucket)。

```
mc mb -p minio/redhat-base
```

预期输出如下：

Bucket created successfully `minio/redhat-base`.

3. 将存储桶 **redhat-base** 的访问策略设置为允许公开下载。以便在后期安装 GPU-operator 时能够被访问。

```
mc anonymous set download minio/redhat-base
```

预期输出如下：

```
Access permission for `minio/redhat-base` is set to `download`
```

4. 在节点当前路径下，将步骤二生成的离线 yum 源文件 **redhat-base-repo** 复制到

minio 服务器的 **minio/redhat-base** 存储桶中。

```
mc cp redhat-base-repo minio/redhat-base --recursive
```

在集群创建配置项用来保存 yum 源信息

本步骤在待部署 GPU Operator 集群的控制节点上进行操作。

1. 执行如下命令创建名为 **redhat.repo** 的文件，用来指定 yum 源存储的配置信息。

```
# 文件名称必须为 redhat.repo，否则安装 gpu-operator 时无法被识别
cat > redhat.repo << EOF
[extension-0]
baseurl = http://10.5.14.200:9000/redhat-base/redhat-base-repo/Packages #步骤一中，放置
yum 源的文件服务器地址
gpgcheck = 0
name = kubeant extension 0

[extension-1]
baseurl = http://10.5.14.200:9000/redhat-base/redhat-base-repo/Packages #步骤一中，放置
yum 源的文件服务器地址
gpgcheck = 0
name = kubeant extension 1
EOF
```

2. 基于创建的 **redhat.repo** 文件，在 **gpu-operator** 命名空间下，创建名为

local-repo-config 的配置文件：

```
kubectl create configmap local-repo-config -n gpu-operator --from-file=./redhat.repo
```

预期输出如下：

```
configmap/local-repo-config created
```

local-repo-config 配置文件用于在安装 **gpu-operator** 时，提供

RepoConfig.ConfigMapName 参数的值，配置文件名称用户可自定义。

3. 查看 **local-repo-config** 的配置文件的内容：

```
kubectl get configmap local-repo-config -n gpu-operator -oyaml
```

至此，您已成功为待部署 GPU Operator 的集群创建了离线 yum 源配置文件。通过在[离线安装 GPU Operator](#)时通过 RepoConfig.ConfigMapName 参数来使用。

构建 Red Hat 7.9 离线 yum 源

使用场景介绍

DCE 5.0 预置了 CentOS 7.9，内核为 3.10.0-1160 的 GPU Operator 离线包。其它 OS 类型的节点或内核需要用户手动构建离线 yum 源。

本文介绍如何基于全局服务集群任意节点构建 Red Hat 7.9 离线 yum 源包，并在安装 GPU Operator 时使用 RepoConfig.ConfigMapName 参数。

前提条件

1. 用户已经在平台上安装了 v0.12.0 及以上版本的[addon 离线包](#)
2. 待部署 GPU Operator 的集群节点 OS 必须为 Red Hat 7.9，且内核版本完全一致
3. 准备一个能够与待部署 GPU Operator 的集群网络联通的文件服务器，如 nginx 或 minio
4. 准备一个能够访问互联网、待部署 GPU Operator 的集群和文件服务器的节点，且节点上已经完成[Docker 的安装](#)
5. 全局服务集群的节点必须为 Red Hat 7.9

操作步骤

1. 构建相关内核版本的离线 Yum 源

1. [下载 rhel7.9 ISO](#)

Red Hat Enterprise Linux 8.4.0

| | | | |
|---------|----------|------------------------------|--------------------------------------|
| x86_64 | DVD iso | Release date
May 18, 2021 | Download (9.43 GB) |
| x86_64 | Boot iso | Release date
May 18, 2021 | Download (721 MB) |
| aarch64 | DVD iso | Release date
May 18, 2021 | Download (6.86 GB) |
| aarch64 | Boot iso | Release date
May 18, 2021 | Download (646.29 MB) |

Red Hat Enterprise Linux 7.9.0

| | | | |
|----------|-------------|---------------------------------|------------------------------------|
| Boot ISO | RHEL x86_64 | Release date
August 11, 2020 | Download (608 MB) |
| DVD ISO | RHEL x86_64 | Release date
August 11, 2020 | Download (4.22 GB) |

下载 rhel7.9 ISO

2. 下载与 Kubean 版本对应的的 [rhel7.9 ospackage](#)

在 容器管理 的全局服务集群中找到 Helm 应用，搜索 kubean，可查看 kubean 的

版本号。

The screenshot shows the Helm application interface within the DaoCloud platform. The sidebar is collapsed. The main area displays a search bar with 'kubean' and a table of applications. The 'kubean' entry is selected, showing its status as '已部署' (Deployed), namespace as 'kubean-system', and chart version as 'kubean.v0.12.2'. This row is also highlighted with a red box.

kubean

在 [kubean 的代码仓库](#) 中下载该版本的 rhel7.9 ospackage。

The screenshot shows the GitHub repository page for 'kubean-io/kubean'. The 'Tags' tab is selected. A list of tags is shown, with 'v0.12.2' highlighted by a red box. Other visible tags include 'v0.12.1', 'v0.12.0', and 'v0.11.2'. Each tag entry includes download links for zip and tar.gz formats.

kubean 的代码仓库

3. 通过安装器导入离线资源

参考[导入离线资源文档](#)。

2. 下载 Red Hat 7.9 OS 的离线驱动镜像

[点击查看下载地址](#)。

The screenshot shows a list of Docker containers under the 'NVIDIA GPU Driver' repository. The containers are listed in descending order of creation date. The container '450.80.02-rhel7.9' is highlighted with a red border. The details for this container are as follows:

| Container Name | Image URL |
|--------------------------|--|
| 460.32.03-ubuntu18.04 | nvcr.io/nvidia/driver:460.32.03-ubuntu18.04 |
| 450.80.02-rhcos4.7 | nvcr.io/nvidia/driver:450.80.02-rhcos4.7 |
| 450.80.02-centos7 | nvcr.io/nvidia/driver:450.80.02-centos7 |
| 450.80.02-centos8 | nvcr.io/nvidia/driver:450.80.02-centos8 |
| 450.80.02-rhel7.9 | nvcr.io/nvidia/driver:450.80.02-rhel7.9 |
| 450.80.02-1.0-rhel7 | nvcr.io/nvidia/driver:450.80.02-1.0-rhel7 |
| 450.80.02-rhcos4.6 | nvcr.io/nvidia/driver:450.80.02-rhcos4.6 |

driveimage

3. 向火种节点仓库上传 Red Hat GPU Operator 离线镜像

参考[向火种节点仓库上传 Red Hat GPU Operator 离线镜像](#)。

!!! note

此参考以 rhel8.4 为例，请注意修改成 rhel7.9。

4. 在集群创建配置项用来保存 Yum 源信息

在待部署 GPU Operator 集群的控制节点上运行以下命令。

- 执行如下命令创建名为 **CentOS-Base.repo** 的文件，用来指定 yum 源存储的配置信息。

```
# 文件名称必须为 CentOS-Base.repo, 否则安装 gpu-operator 时无法被识别
cat > CentOS-Base.repo << EOF
```

```
[extension-0]
baseurl = http://10.5.14.200:9000/centos-base/centos-base # 火种节点的的文件服务器地址，一般为{火种节点 IP} + {9000 端口}
gpgcheck = 0
name = kubeant extension 0

[extension-1]
baseurl = http://10.5.14.200:9000/centos-base/centos-base # 火种节点的的文件服务器地址，一般为{火种节点 IP} + {9000 端口}
gpgcheck = 0
name = kubeant extension 1
EOF
```

2. 基于创建的 **CentOS-Base.repo** 文件，在 `gpu-operator` 命名空间下，创建名为

local-repo-config 的配置文件：

```
kubectl create configmap local-repo-config -n gpu-operator --from-file=CentOS-Base.repo=/etc/yum.repos.d/extension.repo
```

预期输出如下：

```
configmap/local-repo-config created
```

local-repo-config 配置文件用于在安装 `gpu-operator` 时，提供 `RepoConfig.ConfigMapName` 参数的值，配置文件名称用户可自定义。

3. 查看 **local-repo-config** 的配置文件的内容：

```
kubectl get configmap local-repo-config -n gpu-operator -oyaml
```

预期输出如下：

```
yaml title="local-repo-config.yaml" apiVersion: v1 data: CentOS-Base.repo:
"[extension-0]\nbaseurl = http://10.6.232.5:32618/centos-base # 步骤 2 中，放置 yum 源的文件服务器路径 \ngpgcheck = 0\nname = kubeant extension 0\n

[extension-1]\nbaseurl = http://10.6.232.5:32618/centos-base # 步骤 2 中，放置 yum 源的文件服务器路径 \ngpgcheck = 0\nname = kubeant extension 1\n"

kind: ConfigMap metadata: creationTimestamp: "2023-10-18T01:59:02Z"
name: local-repo-config namespace: gpu-operator resourceVersion: "59445080"
uid: c5f0ebab-046f-442c-b932-f9003e014387
```

至此，您已成功为待部署 GPU Operator 的集群创建了离线 yum 源配置文件。其中在 [离](#)

[线安装 GPU Operator](#) 时使用了 `RepoConfig.ConfigMapName` 参数。

RHEL 9.2 离线安装 gpu-operator 驱动

前提条件：已安装 `gpu-operator v23.9.0+2 及更高版本`

RHEL 9.2 驱动镜像不能直接安装，官方的驱动脚本存在一点问题，在官方修复之前，提供如下的步骤来实现离线安装驱动。

禁用 nouveau 驱动

在 RHEL 9.2 中存在 `nouveau` 非官方的 Nvidia 驱动，因此需要先禁用。

```
# 创建一个新的文件  
sudo vi /etc/modprobe.d/blacklist-nouveau.conf  
# 添加以下两行内容:  
blacklist nouveau  
options nouveau modeset=0  
# 禁用 Nouveau  
sudo dracut --force  
# 重启 vm  
sudo reboot  
# 检查是否已经成功禁用  
lsmod | grep nouveau
```

自定义驱动镜像

先在本地创建 `nvidia-driver` 文件：

点击查看完整的 `nvidia-driver` 文件内容

```
#!/bin/bash -x  
# Copyright (c) 2018-2020, NVIDIA CORPORATION. All rights reserved.  
  
set -eu  
  
RUN_DIR=/run/nvidia  
PID_FILE=${RUN_DIR}/${0##*/}.pid
```

```

DRIVER_VERSION=${DRIVER_VERSION:?"Missing DRIVER_VERSION env"}
KERNEL_UPDATE_HOOK=/run/kernel/postinst.d/update-nvidia-driver
NUM_VGPU_DEVICES=0
NVIDIA_MODULE_PARAMS=()
NVIDIA_UVM_MODULE_PARAMS=()
NVIDIA_MODESET_MODULE_PARAMS=()
NVIDIA_PEERMEM_MODULE_PARAMS=()
TARGETARCH=${TARGETARCH:?"Missing TARGETARCH env"}
USE_HOST_MOFED="${USE_HOST_MOFED: -false}"
DNF_RELEASEVER=${DNF_RELEASEVER: -""}
RHEL_VERSION=${RHEL_VERSION: -""}
RHEL_MAJOR_VERSION=9

OPEN_KERNEL_MODULES_ENABLED=${OPEN_KERNEL_MODULES_ENABLED: -false}
[[ "$OPEN_KERNEL_MODULES_ENABLED" == "true" ]] && KERNEL_TYPE=kernel-open
|| KERNEL_TYPE=kernel

DRIVER_ARCH=${TARGETARCH/amd64/x86_64} && DRIVER_ARCH=${DRIVER_ARCH/arm64/aarch64}
echo "DRIVER_ARCH is $DRIVER_ARCH"

SCRIPT_DIR=$( cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> /dev/null && pwd )
source $SCRIPT_DIR/common.sh

_update_package_cache() {
    if [ "${PACKAGE_TAG:-}" != "builtin" ]; then
        echo "Updating the package cache..."
        if ! yum -q makecache; then
            echo "FATAL: failed to reach RHEL package repositories. \
                  Ensure that the cluster can access the proper networks."
            exit 1
        fi
    fi
}

_cleanup_package_cache() {
    if [ "${PACKAGE_TAG:-}" != "builtin" ]; then
        echo "Cleaning up the package cache..."
        rm -rf /var/cache/yum/*
    fi
}

_get_rhel_version_from_kernel() {
    local rhel_version_underscore rhel_version_arr

```

```

rhel_version_underscore=$(echo "${KERNEL_VERSION}" | sed 's/.*/el([0-9]+_[0-9]\+).*/\1/g')
# For e.g. :- from the kernel version 4.18.0-513.9.1.el8_9, we expect to extract the string
"8_9"
if [[ ! ${rhel_version_underscore} =~ ^[0-9]+_[0-9]+\$ ]]; then
    echo "Unable to resolve RHEL version from kernel version" >&2
    return 1
fi
IFS='_' read -r -a rhel_version_arr <<< "$rhel_version_underscore"
if [[ ${#rhel_version_arr[@]} -ne 2 ]]; then
    echo "Unable to resolve RHEL version from kernel version" >&2
    return 1
fi
RHEL_VERSION="${rhel_version_arr[0]}.${rhel_version_arr[1]}"
echo "RHEL VERSION successfully resolved from kernel: ${RHEL_VERSION}"
return 0
}

_resolve_rhel_version() {
_get_rhel_version_from_kernel || RHEL_VERSION="${RHEL_MAJOR_VERSION}"
# set dnf release version as rhel version by default
if [[ -z "${DNF_RELEASEVER}" ]]; then
    DNF_RELEASEVER="${RHEL_VERSION}"
fi
return 0
}

# Resolve the kernel version to the form major.minor.patch-revision.
_resolve_kernel_version() {
echo "Resolving Linux kernel version..."
local version=$(yum -q list available --showduplicates kernel-headers |
    awk -v arch=$(uname -m) 'NR>1 {print $2"."arch}' | tac | grep -E -m1 "^${KERNEL_VERSION}/latest.*$")
if [ -z "${version}" ]; then
    echo "Could not resolve Linux kernel version" >&2
    return 1
fi
KERNEL_VERSION="${version}"
echo "Proceeding with Linux kernel version ${KERNEL_VERSION}"
return 0
}

# Install the kernel modules header/builtin/order files and generate the kernel version string.

```

```

_install_prerequisites() (
    local tmp_dir=$(mktemp -d)

    trap "rm -rf ${tmp_dir}" EXIT
    cd ${tmp_dir}

    echo "Installing elfutils..."
    if ! dnf install -q -y elfutils-libelf.$DRIVER_ARCH; then
        echo "FATAL: failed to install elfutils packages. RHEL entitlement may be improperly deployed."
        exit 1
    fi
    if ! dnf install -q -y elfutils-libelf-devel.$DRIVER_ARCH; then
        echo "FATAL: failed to install elfutils packages. RHEL entitlement may be improperly deployed."
        exit 1
    fi

    rm -rf /lib/modules/${KERNEL_VERSION}
    mkdir -p /lib/modules/${KERNEL_VERSION}/proc

    echo "Enabling RHOC and EUS RPM repos..."
    if [ -n "${OPENSHIFT_VERSION:-}" ]; then
        dnf config-manager --set-enabled rhocp-${OPENSHIFT_VERSION}-for-rhel-9-$DRIVER_ARCH-rpms || true
        if ! dnf makecache --releasever=${DNF_RELEASEVER}; then
            dnf config-manager --set-disabled rhocp-${OPENSHIFT_VERSION}-for-rhel-9-$DRIVER_ARCH-rpms || true
        fi
    fi

    dnf config-manager --set-enabled rhel-9-for-$DRIVER_ARCH-baseos-eus-rpms || true
    if ! dnf makecache --releasever=${DNF_RELEASEVER}; then
        dnf config-manager --set-disabled rhel-9-for-$DRIVER_ARCH-baseos-eus-rpms || true
    fi

    # try with EUS disabled, if it does not work, then try just major version
    if ! dnf makecache --releasever=${DNF_RELEASEVER}; then
        # If pointing to DNF_RELEASEVER does not work, we point to the RHEL_MAJOR_VERSION as a last resort
        if ! dnf makecache --releasever=${RHEL_MAJOR_VERSION}; then
            echo "FATAL: failed to update the dnf metadata cache after multiple attempts with releasevers ${DNF_RELEASEVER}, ${RHEL_MAJOR_VERSION}"
        fi
    fi
)

```

```

    exit 1
else
    DNF_RELEASEVER=${RHEL_MAJOR_VERSION}
fi
fi

echo "Installing Linux kernel headers..."
dnf -q -y --releasever=${DNF_RELEASEVER} install kernel-headers-${KERNEL_VERSION}
kernel-devel-${KERNEL_VERSION} --allowerasing > /dev/null
ln -s /usr/src/kernels/${KERNEL_VERSION} /lib/modules/${KERNEL_VERSION}/build

echo "Installing Linux kernel module files..."
dnf -q -y --releasever=${DNF_RELEASEVER} install kernel-core-${KERNEL_VERSION}
> /dev/null

# Prevent depmod from giving a WARNING about missing files
touch /lib/modules/${KERNEL_VERSION}/modules.order
touch /lib/modules/${KERNEL_VERSION}/modules.builtin

depmod ${KERNEL_VERSION}

echo "Generating Linux kernel version string..."
if [ "$TARGETARCH" = "arm64" ]; then
    gunzip -c /lib/modules/${KERNEL_VERSION}/vmlinuz | strings | grep -E '^Linux version' | sed 's/^(\.*\s\+.*$)/\1/' > version
else
    extract-vmlinux /lib/modules/${KERNEL_VERSION}/vmlinuz | strings | grep -E '^Linux version' | sed 's/^(\.*\s\+.*$)/\1/' > version
fi
if [ -z "<version>" ]; then
    echo "Could not locate Linux kernel version string" >&2
    return 1
fi
mv version /lib/modules/${KERNEL_VERSION}/proc

# Parse gcc version
# gcc_version is expected to match x.y.z
# current_gcc is expected to match 'gcc-x.y.z-rel.el8.x86_64'
local gcc_version=$(cat /lib/modules/${KERNEL_VERSION}/proc/version | grep -Eo "gcc \|GCC\| ([0-9\.\.]+)" | grep -Eo "([0-9\.\.]+)")
local current_gcc=$(rpm -qa gcc)
echo "kernel requires gcc version: '$gcc_version', current gcc version is '$current_gcc'"

```

```

if ! [[ "${current_gcc}" =~ "gcc-${gcc_version}.*" ]]; then
    dnf install -q -y --releasever=${DNF_RELEASEVER} "gcc-${gcc_version}"
fi
)

# Cleanup the prerequisites installed above.
_remove_prerequisites() {
    true
    if [ "${PACKAGE_TAG:-}" != "builtin" ]; then
        dnf -q -y remove kernel-headers-${KERNEL_VERSION} kernel-devel-${KERNEL_VERSION} > /dev/null
    # TODO remove module files not matching an existing driver package.
    fi
}

# Check if the kernel version requires a new precompiled driver packages.
_kernel_requires_package() {
    local proc_mount_arg=""

    echo "Checking NVIDIA driver packages..."

    [[ ! -d /usr/src/nvidia-${DRIVER_VERSION}/${KERNEL_TYPE} ]] && return 0
    cd /usr/src/nvidia-${DRIVER_VERSION}/${KERNEL_TYPE}

    proc_mount_arg="--proc-mount-point /lib/modules/${KERNEL_VERSION}/proc"
    for pkg_name in $(ls -d -1 precompiled/** 2> /dev/null); do
        is_match=$(./mkprecompiled --match ${pkg_name} ${proc_mount_arg})
        if [ "$is_match" == "kernel interface matches." ]; then
            echo "Found NVIDIA driver package ${pkg_name##*/}"
            return 1
        fi
    done
    return 0
}

# Compile the kernel modules, optionally sign them, and generate a precompiled package for use by the nvidia-installer.
_create_driver_package() (
    local pkg_name="nvidia-modules-${KERNEL_VERSION%.*}${PACKAGE_TAG:+${PACKAGE_TAG}}"
    local nvidia_sign_args=""
    local nvidia_modeset_sign_args=""
    local nvidia_uvm_sign_args=""

```

```

trap "make -s -j ${MAX_THREADS} SYSSRC=/lib/modules/${KERNEL_VERSION}/build
clean > /dev/null" EXIT

echo "Compiling NVIDIA driver kernel modules..."
cd /usr/src/nvidia-${DRIVER_VERSION}/${KERNEL_TYPE}

if _gpu_direct_rdma_enabled; then
    ln -s /run/mellanox/drivers/usr/src/ofa_kernel /usr/src/
    # if arch directory exists(MOFED >=5.5) then create a symlink as expected by GPU
    driver installer
        # This is required as currently GPU driver installer doesn't expect headers in x86_64
    folder, but only in either default or kernel-version folder.
        # ls -ltr /usr/src/ofa_kernel/
        # lrwxrwxrwx 1 root root 36 Dec 8 20:10 default -> /etc/alternatives/ofa_kernel_h
    eaders
        # drwxr-xr-x 4 root root 4096 Dec 8 20:14 x86_64
        # lrwxrwxrwx 1 root root 44 Dec 9 19:05 5.4.0-90-generic -> /usr/src/ofa_kernel/x
    86_64/5.4.0-90-generic/
        if [[ -d "/run/mellanox/drivers/usr/src/ofa_kernel/$(uname -m)/$(uname -r)" ]]; then
            if [[ ! -e "/usr/src/ofa_kernel/$(uname -r)" ]]; then
                ln -s "/run/mellanox/drivers/usr/src/ofa_kernel/$(uname -m)/$(uname -r)" /usr/sr
    c/ofa_kernel/
            fi
        fi
    fi

make -s -j ${MAX_THREADS} SYSSRC=/lib/modules/${KERNEL_VERSION}/build nv-lin
ux.o nv-modeset-linux.o > /dev/null

echo "Relinking NVIDIA driver kernel modules..."
rm -f nvidia.ko nvidia-modeset.ko
ld -d -r -o nvidia.ko ./nv-linux.o ./nvidia/nv-kernel.o_binary
ld -d -r -o nvidia-modeset.ko ./nv-modeset-linux.o ./nvidia-modeset/nv-modeset-kernel.o_bina
ry

if [ -n "${PRIVATE_KEY}" ]; then
    echo "Signing NVIDIA driver kernel modules..."
    donkey get ${PRIVATE_KEY} sh -c "PATH=${PATH}:/usr/src/linux-headers-${KERN
EL_VERSION}/scripts && \
        sign-file sha512 ${DONKEY_FILE} pubkey.x509 nvidia.ko nvidia.ko.sign &&
        \
        sign-file sha512 ${DONKEY_FILE} pubkey.x509 nvidia-modeset.ko nvidia-modeset.ko
    .sign && \
        sign-file sha512 ${DONKEY_FILE} pubkey.x509 nvidia-uvm.ko"

```



```
entries=$(ls -1 /proc/driver/nvidia-nvswitch/devices/*)
if [ -z "${entries}" ]; then
    return 1
fi
return 0
}

# For each kernel module configuration file mounted into the container,
# parse the file contents and extract the custom module parameters that
# are to be passed as input to 'modprobe'.
#
# Assumptions:
# - Configuration files are named <module-name>.conf (i.e. nvidia.conf, nvidia-uvm.conf).
# - Configuration files are mounted inside the container at /drivers.
# - Each line in the file contains at least one parameter, where parameters on the same line
#   are space delimited. It is up to the user to properly format the file to ensure
#   the correct set of parameters are passed to 'modprobe'.

_get_module_params() {
    local base_path="/drivers"
    # nvidia
    if [ -f "${base_path}/nvidia.conf" ]; then
        while IFS="" read -r param || [ -n "$param" ]; do
            NVIDIA_MODULE_PARAMS+=("$(param)")
        done <"${base_path}/nvidia.conf"
        echo "Module parameters provided for nvidia: ${NVIDIA_MODULE_PARAMS[@]}"
    fi
    # nvidia-uvm
    if [ -f "${base_path}/nvidia-uvm.conf" ]; then
        while IFS="" read -r param || [ -n "$param" ]; do
            NVIDIA_UVM_MODULE_PARAMS+=("$(param)")
        done <"${base_path}/nvidia-uvm.conf"
        echo "Module parameters provided for nvidia-uvm: ${NVIDIA_UVM_MODULE_PARAMS[@]}"
    fi
    # nvidia-modeset
    if [ -f "${base_path}/nvidia-modeset.conf" ]; then
        while IFS="" read -r param || [ -n "$param" ]; do
            NVIDIA_MODESET_MODULE_PARAMS+=("$(param)")
        done <"${base_path}/nvidia-modeset.conf"
        echo "Module parameters provided for nvidia-modeset: ${NVIDIA_MODESET_MODULE_PARAMS[@]}"
    fi
    # nvidia-peermem
    if [ -f "${base_path}/nvidia-peermem.conf" ]; then
```

```

        while IFS="" read -r param || [ -n "$param" ]; do
            NVIDIA_PEERMEM_MODULE_PARAMS+=("$param")
        done <"${base_path}/nvidia-peermem.conf"
        echo "Module parameters provided for nvidia-peermem: ${NVIDIA_PEERMEM_MODULE_PARAMS[@]}"
    fi
}

# Load the kernel modules and start persisted.
_load_driver() {
    echo "Parsing kernel module parameters..."
    _get_module_params

    local nv_fw_search_path="$RUN_DIR/driver/lib/firmware"
    local set_fw_path="true"
    local fw_path_config_file="/sys/module/firmware_class/parameters/path"
    for param in "${NVIDIA_MODULE_PARAMS[@]}"; do
        if [[ "$param" == "NVreg_EnableGpuFirmware=0" ]]; then
            set_fw_path="false"
        fi
    done

    if [[ "$set_fw_path" == "true" ]]; then
        echo "Configuring the following firmware search path in '$fw_path_config_file': $nv_fw_search_path"
        if [[ ! -z $(grep '^[:space:]' $fw_path_config_file) ]]; then
            echo "WARNING: A search path is already configured in $fw_path_config_file"
            echo "          Retaining the current configuration"
        else
            echo -n "$nv_fw_search_path" > $fw_path_config_file || echo "WARNING: Failed
to configure the firmware search path"
        fi
    fi

    echo "Loading ipmi and i2c_core kernel modules..."
    modprobe -a i2c_core ipmi_msghandler ipmi_devintf

    echo "Loading NVIDIA driver kernel modules..."
    set -o xtrace +o nounset
    modprobe nvidia "${NVIDIA_MODULE_PARAMS[@]}"
    modprobe nvidia-uvm "${NVIDIA_UVM_MODULE_PARAMS[@]}"
    modprobe nvidia-modeset "${NVIDIA_MODESET_MODULE_PARAMS[@]}"
    set +o xtrace -o nounset
}

```

```
if __gpu_direct_rdma_enabled; then
    echo "Loading NVIDIA Peer Memory kernel module..."
    set -o xtrace +o nounset
    modprobe -a nvidia-peermem "${NVIDIA_PEERMEM_MODULE_PARAMS[@]}"
    set +o xtrace -o nounset
fi

echo "Starting NVIDIA persistence daemon..."
nvidia-persistenced --persistence-mode

if [ "${DRIVER_TYPE}" = "vgpu" ]; then
    echo "Copying gridd.conf..."
    cp /drivers/gridd.conf /etc/nvidia/gridd.conf
    if [ "${VGPU_LICENSE_SERVER_TYPE}" = "NLS" ]; then
        echo "Copying ClientConfigToken..."
        mkdir -p /etc/nvidia/ClientConfigToken/
        cp /drivers/ClientConfigToken/* /etc/nvidia/ClientConfigToken/
    fi

    echo "Starting nvidia-gridd.."
    LD_LIBRARY_PATH=/usr/lib64/nvidia/gridd nvidia-gridd

    # Start virtual topology daemon
    _start_vgpu_topology_daemon
fi

if __assert_nvswitch_system; then
    echo "Starting NVIDIA fabric manager daemon..."
    nv-fabricmanager -c /usr/share/nvidia/nvswitch/fabricmanager.cfg
fi

}

# Stop persisted and unload the kernel modules if they are currently loaded.
_unload_driver() {
    local rmmod_args=()
    local nvidia_deps=0
    local nvidia_refs=0
    local nvidia_uvm_refs=0
    local nvidia_modeset_refs=0
    local nvidia_peermem_refs=0

    echo "Stopping NVIDIA persistence daemon..."
    if [ -f /var/run/nvidia-persistenced/nvidia-persistenced.pid ]; then
        local pid=</var/run/nvidia-persistenced/nvidia-persistenced.pid>
    fi
}
```

```
kill -SIGTERM "${pid}"
for i in $(seq 1 50); do
    kill -0 "${pid}" 2> /dev/null || break
    sleep 0.1
done
if [ $i -eq 50 ]; then
    echo "Could not stop NVIDIA persistence daemon" >&2
    return 1
fi

if [ -f /var/run/nvidia-gridd/nvidia-gridd.pid ]; then
    echo "Stopping NVIDIA grid daemon..."
    local pid=$(< /var/run/nvidia-gridd/nvidia-gridd.pid)

    kill -SIGTERM "${pid}"
    for i in $(seq 1 10); do
        kill -0 "${pid}" 2> /dev/null || break
        sleep 0.1
    done
    if [ $i -eq 10 ]; then
        echo "Could not stop NVIDIA Grid daemon" >&2
        return 1
    fi
fi

if [ -f /var/run/nvidia-fabricmanager/nv-fabricmanager.pid ]; then
    echo "Stopping NVIDIA fabric manager daemon..."
    local pid=$(< /var/run/nvidia-fabricmanager/nv-fabricmanager.pid)

    kill -SIGTERM "${pid}"
    for i in $(seq 1 50); do
        kill -0 "${pid}" 2> /dev/null || break
        sleep 0.1
    done
    if [ $i -eq 50 ]; then
        echo "Could not stop NVIDIA fabric manager daemon" >&2
        return 1
    fi
fi

echo "Unloading NVIDIA driver kernel modules..."
if [ -f /sys/module/nvidia_modeset/refcnt ]; then
```

```

nvidia_modeset_refs=$(< /sys/module/nvidia_modeset/refcnt)
rmmmod_args+=("nvidia-modeset")
((++nvidia_deps))

fi
if [ -f /sys/module/nvidia_uvm/refcnt ]; then
    nvidia_uvm_refs=$(< /sys/module/nvidia_uvm/refcnt)
    rmmmod_args+=("nvidia-uvm")
    ((++nvidia_deps))
fi
if [ -f /sys/module/nvidia/refcnt ]; then
    nvidia_refs=$(< /sys/module/nvidia/refcnt)
    rmmmod_args+=("nvidia")
fi
if [ -f /sys/module/nvidia_peermem/refcnt ]; then
    nvidia_peermem_refs=$(< /sys/module/nvidia_peermem/refcnt)
    rmmmod_args+=("nvidia-peermem")
    ((++nvidia_deps))
fi
if [ ${nvidia_refs} -gt ${nvidia_deps} ] || [ ${nvidia_uvm_refs} -gt 0 ] || [ ${nvidia_modeset_refs} -gt 0 ] || [ ${nvidia_peermem_refs} -gt 0 ]; then
    echo "Could not unload NVIDIA driver kernel modules, driver is in use" >&2
    return 1
fi

if [ ${#rmmmod_args[@]} -gt 0 ]; then
    rmmmod ${rmmmod_args[@]}
fi
return 0
}

# Link and install the kernel modules from a precompiled package using the nvidia-installer.
_install_driver() {
    local install_args=()

    echo "Installing NVIDIA driver kernel modules..."
    cd /usr/src/nvidia-${DRIVER_VERSION}
    rm -rf /lib/modules/${KERNEL_VERSION}/video

    if [ "${ACCEPT_LICENSE}" = "yes" ]; then
        install_args+=("--accept-license")
    fi
    IGNORE_CC_MISMATCH=1 nvidia-installer --kernel-module-only --no-drm --ui=none --no-nouveau-check -m=${KERNEL_TYPE} ${install_args[@]}+"${install_args[@]}"

    # May need to add no-cc-check for Rhel, otherwise it complains about cc missing in path
}

```

```

# /proc/version and lib/modules/KERNEL_VERSION/proc are different, by default installer
looks at /proc/ so, added the proc-mount-point

# TODO: remove the -a flag. its not needed. in the new driver version, license-acceptance
is implicit

    #nvidia-installer --kernel-module-only --no-drm --ui=none --no-nouveau-check --no-cc-version
    -check --proc-mount-point /lib/modules/${KERNEL_VERSION}/proc ${install_args[@]}+"${install_
    args[@]}"

}

# Mount the driver rootfs into the run directory with the exception of sysfs.

_mount_rootfs() {

    echo "Mounting NVIDIA driver rootfs..."
    mount --make-runbindable /sys
    mount --make-private /sys
    mkdir -p ${RUN_DIR}/driver
    mount --rbind / ${RUN_DIR}/driver

    echo "Check SELinux status"
    if [ -e /sys/fs/selinux ]; then
        echo "SELinux is enabled"
        echo "Change device files security context for selinux compatibility"
        chcon -R -t container_file_t ${RUN_DIR}/driver/dev
    else
        echo "SELinux is disabled, skipping..."
    fi
}

# Unmount the driver rootfs from the run directory.

_unmount_rootfs() {

    echo "Unmounting NVIDIA driver rootfs..."
    if findmnt -r -o TARGET | grep "${RUN_DIR}/driver" > /dev/null; then
        umount -l -R ${RUN_DIR}/driver
    fi
}

# Write a kernel postinst.d script to automatically precompile packages on kernel update (similar
# to DKMS).

_write_kernel_update_hook() {

    if [ ! -d ${KERNEL_UPDATE_HOOK%/*} ]; then
        return
    fi

    echo "Writing kernel update hook..."
    cat > ${KERNEL_UPDATE_HOOK} <<'EOF'

```

```

#!/bin/bash

set -eu
trap 'echo "ERROR: Failed to update the NVIDIA driver" >&2; exit 0' ERR

NVIDIA_DRIVER_PID=$(< /run/nvidia/nvidia-driver.pid)

export "$(grep -z DRIVER_VERSION /proc/${NVIDIA_DRIVER_PID}/environ)"
nsenter -t "${NVIDIA_DRIVER_PID}" -m -- nvidia-driver update --kernel "$1"
EOF
    chmod +x ${KERNEL_UPDATE_HOOK}
}

_shutdown() {
    if _unload_driver; then
        _unmount_rootfs
        rm -f ${PID_FILE} ${KERNEL_UPDATE_HOOK}
        return 0
    fi
    return 1
}

_find_vgpu_driver_version() {
    local count=""
    local version=""
    local drivers_path="/drivers"

    if [ "${DISABLE_VGPU_VERSION_CHECK}" = "true" ]; then
        echo "vgpu version compatibility check is disabled"
        return 0
    fi
    # check if vgpu devices are present
    count=$(vgpu-util count)
    if [ $? -ne 0 ]; then
        echo "cannot find vgpu devices on host, please check /var/log/vgpu-util.log for more
details..."
        return 0
    fi
    NUM_VGPU_DEVICES=$(echo "$count" | awk -F= '{print $2}')
    if [ $NUM_VGPU_DEVICES -eq 0 ]; then
        # no vgpu devices found, treat as passthrough
        return 0
    fi
    echo "found $NUM_VGPU_DEVICES vgpu devices on host"
}

```

```

# find compatible guest driver using driver catalog
if [ -d "/mnt/shared-nvidia-driver-toolkit/drivers" ]; then
    drivers_path="/mnt/shared-nvidia-driver-toolkit/drivers"
fi
version=$(vgpu-util match -i "${drivers_path}" -c "${drivers_path}/vgpuDriverCatalog.yaml")
if [ $? -ne 0 ]; then
    echo "cannot find match for compatible vgpu driver from available list, please check
/var/log/vgpu-util.log for more details..."
    return 1
fi
DRIVER_VERSION=$(echo "$version" | awk -F= '{print $2}')
echo "vgpu driver version selected: ${DRIVER_VERSION}"
return 0
}

_start_vgpu_topology_daemon() {
    type nvidia-topologyd > /dev/null 2>&1 || return 0
    echo "Starting nvidia-topologyd.."
    nvidia-topologyd
}

_prepare() {
    if [ "${DRIVER_TYPE}" = "vgpu" ]; then
        _find_vgpu_driver_version || exit 1
    fi

    # Install the userspace components and copy the kernel module sources.
    sh NVIDIA-Linux-$DRIVER_ARCH-$DRIVER_VERSION.run -x && \
        cd NVIDIA-Linux-$DRIVER_ARCH-$DRIVER_VERSION && \
        sh /tmp/install.sh nvinstall && \
        mkdir -p /usr/src/nvidia-$DRIVER_VERSION && \
        mv LICENSE mkprecompiled ${KERNEL_TYPE} /usr/src/nvidia-$DRIVER_VERSION
    && \
        sed '9,${/^(kernel|LICENSE)/!d}' .manifest > /usr/src/nvidia-$DRIVER_VERSION/.ma
nifest

    echo -e "\n===== NVIDIA Software Installer =====\n"
    echo -e "Starting installation of NVIDIA driver version ${DRIVER_VERSION} for Linux
kernel version ${KERNEL_VERSION}\n"
}

_prepare_exclusive() {
    _prepare
}

```

```
exec 3> ${PID_FILE}
if ! flock -n 3; then
    echo "An instance of the NVIDIA driver is already running, aborting"
    exit 1
fi
echo $$ >&3

trap "echo 'Caught signal'; exit 1" HUP INT QUIT PIPE TERM
trap "_shutdown" EXIT

_unload_driver || exit 1
_unmount_rootfs
}

_build() {
    # Install dependencies
    if _kernel_requires_package; then
        _update_package_cache
        _install_prerequisites
        _create_driver_package
        #_remove_prerequisites
        _cleanup_package_cache
    fi

    # Build the driver
    _install_driver
}

_load() {
    _load_driver
    _mount_rootfs
    _write_kernel_update_hook

    echo "Done, now waiting for signal"
    sleep infinity &
    trap "echo 'Caught signal'; _shutdown && { kill $!; exit 0; }" HUP INT QUIT PIPE TE
RM
    trap - EXIT
    while true; do wait $! || continue; done
    exit 0
}

init() {
```

```

_prepare_exclusive

_build

_load
}

build() {
    _prepare

    _build
}

load() {
    _prepare_exclusive

    _load
}

update() {
    exec 3>&2
    if exec 2> /dev/null 4< ${PID_FILE}; then
        if ! flock -n 4 && read pid <&4 && kill -0 "${pid}"; then
            exec > >(tee -a "/proc/${pid}/fd/1")
            exec 2> >(tee -a "/proc/${pid}/fd/2" >&3)
        else
            exec 2>&3
        fi
        exec 4>&-
    fi
    exec 3>&-

# vgpu driver version is chosen dynamically during runtime, so pre-compile modules for
# only non-vgpu driver types
if [ "${DRIVER_TYPE}" != "vgpu" ]; then
    # Install the userspace components and copy the kernel module sources.
    if [ ! -e /usr/src/nvidia-${DRIVER_VERSION}/mkprecompiled ]; then
        sh NVIDIA-Linux-$DRIVER_ARCH-$DRIVER_VERSION.run -x && \
        cd NVIDIA-Linux-$DRIVER_ARCH-$DRIVER_VERSION && \
        sh /tmp/install.sh nvinstall && \
        mkdir -p /usr/src/nvidia-$DRIVER_VERSION && \
        mv LICENSE mkprecompiled ${KERNEL_TYPE} /usr/src/nvidia-$DRIVER_-
VERSION && \
        sed '9,${/^(kernel|LICENSE)/!d}' .manifest > /usr/src/nvidia-$DRIVER_VER

```

```

SION/.manifest
    fi
    fi

    echo -e "\n===== NVIDIA Software Updater =====\n"
    echo -e "Starting update of NVIDIA driver version ${DRIVER_VERSION} for Linux kernel
el version ${KERNEL_VERSION}\n"

    trap "echo 'Caught signal'; exit 1" HUP INT QUIT PIPE TERM

    _update_package_cache
    _resolve_kernel_version || exit 1
    _install_prerequisites
    if _kernel_requires_package; then
        _create_driver_package
    fi
    _remove_prerequisites
    _cleanup_package_cache

    echo "Done"
    exit 0
}

# Wait for MOFED drivers to be loaded and load nvidia-peermem whenever it gets unloaded
# during MOFED driver updates
reload_nvidia_peermem() {
    if [ "$USE_HOST_MOFED" = "true" ]; then
        until lsmod | grep mlx5_core > /dev/null 2>&1 && [ -f /run/nvidia/validations/.driver
-ctr-ready ];
        do
            echo "waiting for mellanox ofed and nvidia drivers to be installed"
            sleep 10
        done
    else
        # use driver readiness flag created by MOFED container
        until [ -f /run/mellanox/drivers/.driver-ready ] && [ -f /run/nvidia/validations/.driver-ct
r-ready ];
        do
            echo "waiting for mellanox ofed and nvidia drivers to be installed"
            sleep 10
        done
    fi
    # get any parameters provided for nvidia-peermem
    _get_module_params && set +o nounset
}

```

```

if chroot /run/nvidia/driver modprobe nvidia-peermem "${NVIDIA_PEERMEM_MODULE_PARAMETERS[@]}"; then
    if [ -f /sys/module/nvidia_peermem/refcnt ]; then
        echo "successfully loaded nvidia-peermem module, now waiting for signal"
        sleep inf
        trap "echo 'Caught signal'; exit 1" HUP INT QUIT PIPE TERM
    fi
fi
echo "failed to load nvidia-peermem module"
exit 1
}

# probe by gpu-operator for liveness/startup checks for nvidia-peermem module to be loaded when MOFED drivers are ready
probe_nvidia_peermem() {
    if lsmod | grep mlx5_core > /dev/null 2>&1; then
        if [ ! -f /sys/module/nvidia_peermem/refcnt ]; then
            echo "nvidia-peermem module is not loaded"
            return 1
        fi
    else
        echo "MOFED drivers are not ready, skipping probe to avoid container restarts..."
    fi
    return 0
}

usage() {
cat >&2 <<EOF
Usage: $0 COMMAND [ARG...]

```

Commands:

```

init  [-a | --accept-license] [-m | --max-threads MAX_THREADS]
build [-a | --accept-license] [-m | --max-threads MAX_THREADS]
load
update [-k | --kernel VERSION] [-s | --sign KEYID] [-t | --tag TAG] [-m | --max-threads MAX_THREADS]
EOF
exit 1
}

if [ $# -eq 0 ]; then
    usage
fi
command=$1; shift

```

```

case "${command}" in
    init) options=$(getopt -l accept-license,max-threads: -o am: -- "$@") ;;
    build) options=$(getopt -l accept-license,tag:,max-threads: -o at:m: -- "$@") ;;
    load) options="" ;;
    update) options=$(getopt -l kernel:,sign:,tag:,max-threads: -o ks:t:m: -- "$@") ;;
    reload_nvidia_peermem) options="" ;;
    probe_nvidia_peermem) options="" ;;
    *) usage ;;
esac
if [ $? -ne 0 ]; then
    usage
fi
eval set -- "${options}"

ACCEPT_LICENSE=""
MAX_THREADS=""
KERNEL_VERSION=$(uname -r)
PRIVATE_KEY=""
PACKAGE_TAG=""

for opt in ${options}; do
    case "$opt" in
        -a | --accept-license) ACCEPT_LICENSE="yes"; shift 1 ;;
        -k | --kernel) KERNEL_VERSION=$2; shift 2 ;;
        -m | --max-threads) MAX_THREADS=$2; shift 2 ;;
        -s | --sign) PRIVATE_KEY=$2; shift 2 ;;
        -t | --tag) PACKAGE_TAG=$2; shift 2 ;;
        --) shift; break ;;
    esac
done
if [ $# -ne 0 ]; then
    usage
fi

_resolve_rhel_version || exit 1

$command

```

使用官方的镜像来二次构建自定义镜像，如下是一个 Dockerfile 文件的内容：

```

FROM nvcr.io/nvidia/driver: 535.183.06-rhel9.2
COPY nvidia-driver /usr/local/bin
RUN chmod +x /usr/local/bin/nvidia-driver
CMD ["/bin/bash", "-c"]

```

构建命令并推送到火种集群：

```
docker build -t {火种 registry}/nvcr.m.daocloud.io/nvidia/driver: 535.183.06-01-rhel9.2 -f Dockerfile .
docker push {火种 registry}/nvcr.m.daocloud.io/nvidia/driver: 535.183.06-01-rhel9.2
```

安装驱动

1. 安装 gpu-operator addon
2. 设置 driver.version=535.183.06-01

Ubuntu22.04 离线安装 gpu-operator 驱动

前提条件：已安装 gpu-operator v23.9.0+2 及更高版本

准备离线镜像

1. 查看内核版本


```
$ uname -r
5.15.0-78-generic
```
2. 查看内核对应的 GPU Driver 镜像版本，<https://catalog.ngc.nvidia.com/orgs/nvidia/containers/driver/tags>。 使用内核查询镜像版本，通过 ctr export 保存镜像。


```
ctr i pull nvcr.io/nvidia/driver:535-5.15.0-78-generic-ubuntu22.04
ctr i export --all-platforms driver.tar.gz nvcr.io/nvidia/driver:535-5.15.0-78-generic-ubuntu
22.04
```

3. 把镜像导入到火种集群的镜像仓库中

```
ctr i import driver.tar.gz
ctr i tag nvcr.io/nvidia/driver:535-5.15.0-78-generic-ubuntu22.04 {火种 registry}/nvcr.m.da
ocloud.io/nvidia/driver:535-5.15.0-78-generic-ubuntu22.04
ctr i push {火种 registry}/nvcr.m.daocloud.io/nvidia/driver:535-5.15.0-78-generic-ubuntu22.
04 --skip-verify=true
```

安装驱动

1. 安装 gpu-operator addon

2. 若使用预编译模式，则设置 driver.usePrecompiled=true，并设置 driver.version=535。

这里要注意，需要将默认的 535.104.12 改成 535。（非预编译模式跳过此步，直接安装即可）

安装 - gpu-operator

应用设置 Kubernetes 编排确认

gpu-operator 基础设施

NVIDIA GPU Operator creates/configures/manages GPUs atop Kubernetes

名称 *

命名空间 * 已有命名空间 新建命名空间

版本 *

失败删除 未启用

就绪等待 未启用

详细日志 未启用

参数配置

表单 YAML 变化

Gpu-Operator
After installation, the default mode is full GPU. If you need to switch to MIG or vGPU, please go to the appropriate node to make the switch.

systemOS

Operator

InitContainer

image
repository
version

Driver

enabled On
usePrecompiled On

image
repository
version

RepoConfig

configMapName

Rdma

enabled Off
useHostMoFED Off

Toolkit

enabled On

MigManager

Config

name

取消

安装驱动

应用使用 GPU 整卡

本节介绍如何在 DCE 5.0 平台将整个 NVIDIA GPU 卡分配给单个应用。

前提条件

- 已经[部署 DCE 5.0](#) 容器管理平台，且平台运行正常。
- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 当前集群已离线安装 GPU Operator 并已启用 NVIDIA DevicePlugin，可参考[GPU Operator 离线安装](#)。
- 当前集群内 GPU 卡未进行任何虚拟化操作或被其它应用占用。

操作步骤

使用 UI 界面配置

1. 确认集群是否已检测 GPU 卡。点击对应 **集群 -> 集群设置 -> Addon 插件**，查看是否已自动启用并自动检测对应 GPU 类型。目前集群会自动启用 **GPU**，并且设置 **GPU** 类型为 **Nvidia GPU**。

集群设置

集群设置

2. 部署工作负载，点击对应 **集群 -> 工作负载**，通过镜像方式部署工作负载，选择类

型 (Nvidia GPU) 之后 , 需要配置应用使用的物理卡数量 :

物理卡数量 (nvidia.com/gpu) : 表示当前 Pod 需要挂载几张物理卡 , 输入值必须为整数且 **小于等于** 宿主机上的卡数量。

集群设置

集群设置

如果上述值配置的有问题则会出现调度失败 , 资源分配不了的情况。

使用 YAML 配置

创建工作负载申请 GPU 资源 , 在资源申请和限制配置中增加 **nvidia.com/gpu: 1** 参数配置

应用使用物理卡的数量。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: full-gpu-demo
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: full-gpu-demo
  template:
    metadata:
      labels:
        app: full-gpu-demo
    spec:
      containers:
        - image: chrstnhtschl/gpu_burn
          name: container-0
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
              nvidia.com/gpu: 1    # 申请 GPU 的数量
            limits:
              cpu: 250m
```

```

memory: 512Mi
nvidia.com/gpu: 1    # GPU 数量的使用上限
imagePullSecrets:
- name: default-secret
!!! note
使用 __nvidia.com/gpu__ 参数指定 GPU 数量时，requests 和 limits 值需要保持一致。

```

安装 NVIDIA vGPU Addon

如需将一张 NVIDIA 虚拟化成多个虚拟 GPU，并将其分配给不同的虚拟机或用户，您可以使用 NVIDIA 的 vGPU 能力。本节介绍如何在 DCE 5.0 平台中安装 vGPU 插件，这是使用 NVIDIA vGPU 能力的前提。## 前提条件

- 参考 [GPU 支持矩阵](#) 确认集群节点上具有对应型号的 GPU 卡。
- 当前集群已通过 Operator 部署 NVIDIA 驱动，具体参考 [GPU Operator 离线安装](#)。

操作步骤

1. 功能模块路径： 容器管理 -> 集群管理，点击目标集群的名称，从左侧导航栏点击

Helm 应用 -> Helm 模板 -> 搜索 nvidia-vgpu。

找到 nvidia-vgpu

找到 nvidia-vgpu

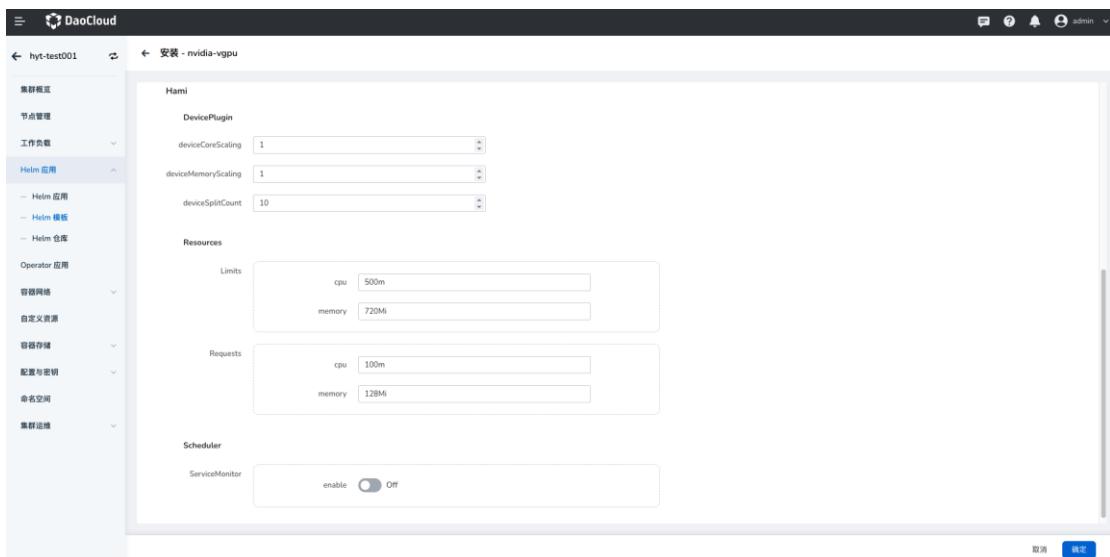
2. 在安装 vGPU 的过程中提供了几个基本修改的参数，如果需要修改高级参数点击

YAML 列进行修改：

- **deviceCoreScaling** : NVIDIA 装置算力使用比例，预设值是 1。可以大于 1
(启用虚拟算力，实验功能)。如果我们配置 **devicePlugin.deviceCoreScaling** 参数为 s，在部署了我们装置插件的 Kubernetes 集群中，这张 GPU 分出的 vGPU 将总共包含 **s * 100%** 算

力。

- **deviceMemoryScaling** : NVIDIA 装置显存使用比例，预设值是 1。可以大于 1 (启用虚拟显存，实验功能)。对于有 M 显存大小的 NVIDIA GPU，如果我们将配置 **devicePlugin.deviceMemoryScaling** 参数为 S，在部署了我们装置插件的 Kubernetes 集群中，这张 GPU 分出的 vGPU 将总共包含 $S * M$ 显存。
- **deviceSplitCount** : 整数类型，预设值是 10。GPU 的分割数，每一张 GPU 都不能分配超过其配置数目的任务。若其配置为 N 的话，每个 GPU 上最多可以同时存在 N 个任务。
- **Resources** : 就是对应 vgpu-device-plugin 和 vgpu-schedule pod 的资源使用量。
- **ServiceMonitor** : 默认不开启，开启后可前往可观测性模块查看 vGPU 相关监控。如需开启，请确保 insight-agent 已安装并处于运行状态，否则将导致 NVIDIA vGPU Addon 安装失败。



修改参数

3. 安装成功之后会在指定 **Namespace** 下出现如下两个类型的 Pod，即表示 NVIDIA

vGPU 插件已安装成功：

出现两个 Pod

出现两个 Pod

安装成功后，[部署应用可使用 vGPU 资源。](#)

!!! note

NVIDIA vGPU Addon 不支持从老版本 v2.0.0 直接升级为最新版 v2.0.0+1；如需升级，请卸载老版本后重新安装。

应用使用 Nvidia vGPU

本节介绍如何在 DCE 5.0 平台使用 vGPU 能力。

前提条件

- 集群节点上具有[对应型号的 GPU 卡](#)
- 已成功安装 vGPU Addon，详情参考 [GPU Addon 安装](#)
- 已安装 GPU Operator，并已 [关闭 Nvidia.DevicePlugin 能力](#)，可参考 [GPU Operator 离线安装](#)

操作步骤

界面使用 vGPU

- 确认集群是否已检测 GPU 卡。点击对应 **集群 -> 集群设置 -> Addon 插件**，查看是否已自动启用并自动检测对应 GPU 类型。目前集群会自动启用 **GPU**，并且设置 GPU 类型为 **Nvidia vGPU**。

安装 vgpu

安装 vgpu

2. 部署工作负载，点击对应 **集群** -> **工作负载**，通过镜像方式部署工作负载，选择类

型（Nvidia vGPU）之后，会自动出现如下几个参数需要填写：

- **物理卡数量**（nvidia.com/vgpu）：表示当前 Pod 需要挂载几张物理卡，输入值必须为整数且 **小于等于** 宿主机上的卡数量。
- **GPU 算力**（nvidia.com/gpucores）：表示每张卡占用的 GPU 算力，值范围为 0-100；如果配置为 0，则认为不强制隔离；配置为 100，则认为独占整张卡。
- **GPU 显存**（nvidia.com/gpumem）：表示每张卡占用的 GPU 显存，值单位为 MB，最小值为 1，最大值为整卡的显存值。

如果上述值配置的有问题则会出现调度失败，资源分配不了的情况。

部署工作负载

部署工作负载

YAML 配置使用 vGPU

参考如下工作负载配置，在资源申请和限制配置中增加 nvidia.com/vgpu: '1' 参数来配置应用使用物理卡的数量。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: full-vgpu-demo
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
```

```

app: full-vgpu-demo
template:
  metadata:
    creationTimestamp: null
  labels:
    app: full-vgpu-demo
spec:
  containers:
    - name: full-vgpu-demo1
      image: chrstnhtschl/gpu_burn
      resources:
        limits:
          nvidia.com/gpucores: '20'    # 申请每张卡占用 20% 的 GPU 算力
          nvidia.com/gpumem: '200'    # 申请每张卡占用 200MB 的显存
          nvidia.com/vgpu: '1'       # 申请 GPU 的数量
      imagePullPolicy: Always
      restartPolicy: Always

```

构建 vGPU 显存超配镜像

[Hami 项目](#) 中 vGPU 显存超配的功能已经不存在，目前使用有显存超配的 libvgpu.so 文件

重新构建。

```
bash title="Dockerfile" FROM docker.m.daocloud.io/projecthami/hami: v2.3.11 COPY libvgpu.so /k8s-vgpu/lib/nvidia/
```

执行以下命令构建镜像：

```
docker build -t release.daocloud.io/projecthami/hami: v2.3.11 -f Dockerfile .
```

然后把镜像 push 到 release.daocloud.io 中。

如何让集群纳管指定的 GPU 卡

本文介绍如何让集群纳管指定的 GPU 卡。

使用场景

- 客户服务器资源有限，且部分应用不支持容器部署模式的情况下，需要兼顾传统应用部署和容器化应用部署需求，允许客户纳管一部分的 GPU 卡，并预留一部分 GPU 卡供其传统应用使用。
- 在故障隔离机制中，device-plugin 能够自动识别并隔离出现故障的 GPU 卡，以确保这些故障卡不会再被调度用于执行任何任务。对于某些可能未被 device-plugin 自动检测到的故障 GPU 卡，可以通过选择性纳管的方式手动隔离。

前提条件

- 已经[部署 DCE 5.0](#) 容器管理平台，且平台运行正常。
- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 当前集群已安装 [GPU operator](#)
- 当前集群已安装 [NVIDIA-vGPU](#)，且 NVIDIA-vGPU 在 2.4.0+1 及以上版本

操作步骤

!!! note

仅 vGPU 模式支持纳管指定的 GPU 卡。

- 修改 nvidia-vgpu-hami-device-plugin。点击对 **集群 -> 配置与密钥-> 配置项**，搜索 **nvidia-vgpu-hami-device-plugin**，点击名称进入详情。

The screenshot shows the DaoCloud Enterprise 5.0 web interface. The left sidebar has sections for '节点管理', '工作负载', 'Helm 应用', 'Operator 应用', '容器网络', '自定义资源', '容器存储', '配置与密钥', and '命名空间'. The '配置与密钥' section is expanded, and '配置项' is selected. The main area shows a table of configuration items with columns for '配置项名称', '配置项别名', '标签', '命名空间', and '创建时间'. One item, 'nvidia-vgpu-hami-device-plugin', is highlighted with a red box.

config1

2. 编辑 YAML，填写被过滤的 GPU 卡的 uuid 或者 index 码。过滤后，pod 不会被调度到这些 GPU 卡上。

filterDevices：设备实例的过滤设备。过滤设备可以是以下值之一：

- **uuid**：设备的 UUID
- **index**：设备索引

The screenshot shows the 'Edit YAML' dialog for the 'nvidia-vgpu-hami-device-plugin' configuration item. The 'filterDevices' field is highlighted with a red box and contains the value '["uuid": "GPU-2ab63f5c-c3f6-6e25-bee2-7fa65fb6b6d3"]'. The dialog also shows other fields like 'nodeConfig' and 'data: config.json'.

config2

3. 重启 nvidia-vgpu-hami-device-plugin

The screenshot shows the 'GPU Operator' cluster details. A table lists GPU resources, including one named 'nvidia-vgpu-hami-device-plugin' which is currently in a 'pending' state. A tooltip at the top right indicates a successful task completion under the 'nvidia-vgpu-hami-device-plugin' task.

重启 device

4. 部署工作负载并查看调度情况。

The screenshot shows the 'Node Management' section. It displays resource usage statistics for nodes, including CPU, memory, and GPU utilization. A specific GPU resource is highlighted in red.

节点详情

```

root@controller-node-1:~# kubectl get po -n default
NAME          READY   STATUS    RESTARTS   AGE
test-vgpu-00-8449779448-wqw7q   1/1     Running   1 (25m ago)  77m
uuu-99-76ffb585d5-cbjj6       1/1     Running   0          10s
root@controller-node-1:~#
root@controller-node-1:~# kubectl exec -it uuu-99-76ffb585d5-cbjj6 -n default bash
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl exec [POD] -- [COMMAND] instead.
root@uuu-99-76ffb585d5-cbjj6:#
root@uuu-99-76ffb585d5-cbjj6:#
root@uuu-99-76ffb585d5-cbjj6:~# nvidia-smi -L
GPU 0: Tesla P4 (UUID: GPU-599adde5-51cd-1e86-d2db-34bdbe0248b9)
root@uuu-99-76ffb585d5-cbjj6:~# nvidia-smi
[HAMI-core Msg(32:140400656557888]: Initializing.....
Mon Oct 28 03:08:55 2024
+-----+
| NVIDIA-SMI 550.90.07      Driver Version: 550.90.07      CUDA Version: 12.4 |
+-----+
| GPU  Name Persistence-M  Bus-Id Disp.A  Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr/Usage/Cap | Memory-Usage | GPU-Util Compute M. |
|                               |             |             |          MIG M. |
|-----+
| 0  Tesla P4      On           00000000:0B:00.0 Off        8 |
| N/A   75C   P0    65W / 75W | 5461MiB / 6000MiB | 100%   Default   N/A |
+-----+
+-----+
| Processes:                   GPU Memory |
| GPU  GI  CI PID Type Process name        Usage  |
| ID  ID   |
|-----+
[HAMI-core Msg(32:140400656557888]: Calling exit handler 32
root@uuu-99-76ffb585d5-cbjj6:#

```

deploy

NVIDIA 多实例 GPU(MIG) 概述

MIG 场景

- **多租户云环境**

MIG 允许云服务提供商将一块物理 GPU 划分为多个独立的 GPU 实例，每个实例可以独立分配给不同的租户。这样可以实现资源的隔离和独立性，满足多个租户对 GPU 计算能力的需求。

- **容器化应用程序**

MIG 可以在容器化环境中实现更细粒度的 GPU 资源管理。通过将物理 GPU 划分为多个 MIG 实例，可以为每个容器分配独立的 GPU 计算资源，提供更好的性能隔离和资源利用。

- **批处理作业**

对于需要大规模并行计算的批处理作业，MIG 可以提供更高的计算性能和更大的显存容量。每个 MIG 实例可以利用物理 GPU 的一部分计算资源，从而加速大规模计算任务的处理。

- **AI/机器学习训练**

MIG 可以在训练大规模深度学习模型时提供更大的计算能力和显存容量。将物理 GPU 划分为多个 MIG 实例，每个实例可以独立进行模型训练，提高训练效率和吞吐量。

总体而言，NVIDIA MIG 适用于需要更细粒度的 GPU 资源分配和管理的场景，可以实现资源的隔离、提高性能利用率，并且满足多个用户或应用程序对 GPU 计算能力的需求。

MIG 概述

NVIDIA 多实例 GPU (Multi-Instance GPU , 简称 MIG) 是 NVIDIA 在 H100 , A100 , A30 系列 GPU 卡上推出的一项新特性 , 旨在将一块物理 GPU 分割为多个 GPU 实例 , 以提供更细粒度的资源共享和隔离。MIG 最多可将一块 GPU 划分成七个 GPU 实例 , 使得一个物理 GPU 卡可为多个用户提供单独的 GPU 资源 , 以实现最佳 GPU 利用率。这个功能使得多个应用程序或用户可以同时共享 GPU 资源 , 提高了计算资源的利用率 , 并增加了系统的可扩展性。

通过 MIG , 每个 GPU 实例的处理器在整个内存系统中具有独立且隔离的路径——芯片上的交叉开关端口、L2 高速缓存组、内存控制器和 DRAM 地址总线都唯一分配给单个实例。

这确保了单个用户的工作负载能够以可预测的吞吐量和延迟运行 , 并具有相同的二级缓存分配和 DRAM 带宽。 MIG 可以划分可用的 GPU 计算资源 (包括流多处理器或 SM 和 GPU 引擎 , 如复制引擎或解码器) 进行分区 , 以便为不同的客户端 (如虚拟机、容器或进程) 提供定义的服务质量 (QoS) 和故障隔离) 。 MIG 使多个 GPU 实例能够在单个物理 GPU 上并行运行。

MIG 允许多个 vGPU (以及虚拟机) 在单个 GPU 实例上并行运行 , 同时保留 vGPU 提供的隔离保证。 有关使用 vGPU 和 MIG 进行 GPU 分区的详细信息 , 请参阅 [NVIDIA Multi-Instance GPU and NVIDIA Virtual Compute Server](#) 。

MIG 架构

如下是一个 MIG 的概述图 , 可以看出 MIG 将一张物理 GPU 卡虚拟化成了 7 个 GPU 实例 , 这些 GPU 实例能够被多个 User 使用。



img

重要概念

- **SM** : 流式多处理器 (Streaming Multiprocessor) , GPU 的核心计算单元 , 负责执行图形渲染和通用计算任务。每个 SM 包含一组 CUDA 核心 , 以及共享内存、寄存器文件和其他资源 , 可以同时执行多个线程。每个 MIG 实例都拥有一定数量的 SM 和其他相关资源 , 以及被划分出来的显存。
- **GPU Memory Slice** : GPU 内存切片 , GPU 内存切片是 GPU 内存的最小部分 , 包括相应的内存控制器和缓存。GPU 内存切片大约是 GPU 内存资源总量的八分之一 , 包括容量和带宽。
- **GPU SM Slice** : GPU SM 切片是 GPU 上 SM 的最小计算单位。在 MIG 模式下配置时 , GPU SM 切片大约是 GPU 中可用 SMS 总数的七分之一。
- **GPU Slice** : GPU 切片是 GPU 中由单个 GPU 内存切片和单个 GPU SM 切片组合在一起的最小部分。
- **GPU Instance** : GPU 实例 (GI) 是 GPU 切片和 GPU 引擎 (DMA、NVDEC 等) 的组合。GPU 实例中的任何内容始终共享所有 GPU 内存切片和其他 GPU 引擎 , 但它的 SM 切片可以进一步细分为计算实例 (CI) 。GPU 实例提供内存 QoS。每个 GPU 切片都包含专用的 GPU 内存资源 , 这些资源会限制可用容量和带宽 , 并提供内存 QoS。每个 GPU 内存切片获得总 GPU 内存资源的八分之一 , 每个 GPU SM 切片获得 SM 总数的七分之一。
- **Compute Instance** : GPU 实例的计算切片可以进一步细分为多个计算实例 (CI) , 其中 CI 共享父 GI 的引擎和内存 , 但每个 CI 都有专用的 SM 资源。

GPU 实例 (GI)

本节介绍如何在 GPU 上创建各种分区。将使用 A100-40GB 作为示例演示如何对单个 GPU 物理卡上进行分区。

GPU 的分区是使用内存切片进行的，因此可以认为 A100-40GB GPU 具有 8x5GB 内存切片和 7 个 GPU SM 切片，如下图所示，展示了 A100 上可用的内存切片。

如上所述，创建 GPU 实例 (GI) 需要将一定数量的内存切片与一定数量的计算切片相结合。在下图中，一个 5GB 内存切片与 1 个计算切片相结合，以创建 **1g.5gb** GI 配置文件：

同样，4x5GB 内存切片可以与 4x1 计算切片结合使用以创建 **4g.20gb** 的 GI 配置文件：

计算实例 (CI)

GPU 实例的计算切片(GI)可以进一步细分为多个计算实例 (CI)，其中 CI 共享父 GI 的引擎和内存，但每个 CI 都有专用的 SM 资源。使用上面的相同 **4g.20gb** 示例，可以创建一个 CI 以仅使用第一个计算切片的 **1c.4g.20gb** 计算配置，如下图蓝色部分所示：

在这种情况下，可以通过选择任何计算切片来创建 4 个不同的 CI。还可以将两个计算切片组合在一起以创建 **2c.4g.20gb** 的计算配置)：

除此之外，还可以组合 3 个计算切片以创建计算配置文件，或者可以组合所有 4 个计算

切片以创建 **3c.4g.20gb** 、 **4c.4g.20gb** 计算配置文件。 合并所有 4 个计算切片时，配置文件简称为 **4g.20gb** 。

开启 MIG 功能

本章节介绍如何开启 NVIDIA MIG 功能方式，NVIDIA 当前提供两种在 Kubernetes 节点上公开 MIG 设备的策略：

- **Single 模式**，节点仅在其所有 GPU 上公开单一类型的 MIG 设备。
- **Mixed 模式**，节点在其所有 GPU 上公开混合 MIG 设备类型。

详情参考：[NVIDIA GPU 卡使用模式](#)

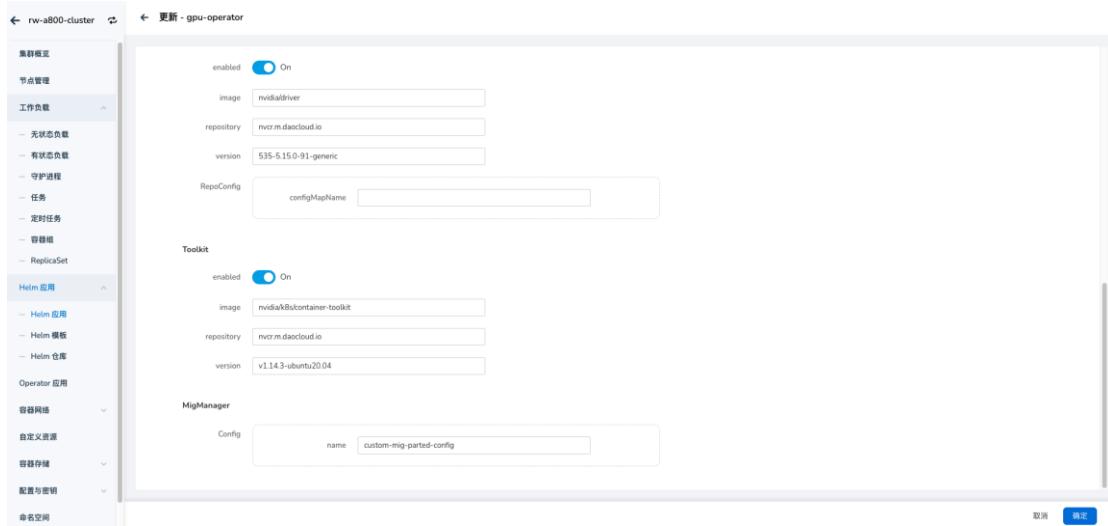
前提条件

- 待安装 GPU 驱动节点系统要求请参考：[GPU 支持矩阵](#)
- 确认集群节点上具有对应型号的 GPU 卡（[NVIDIA H100](#)、[A100](#) 和 [A30](#) Tensor Core GPU），详情参考 [GPU 支持矩阵](#)。
- 节点上的所有 GPU 必须：属于同一产品线（例如 A100-SXM-40GB）

安装 gpu-operator Addon

参数配置

[安装 Operator](#) 时需要对应设置 MigManager Config 参数，默认为 **default-mig-parted-config**，同时也可以自定义切分策略配置文件：



single

自定义切分策略

`## 自定义切分 GI 实例配置`

`all-disabled:`

- `devices: all`
- `mig-enabled: false`

`all-enabled:`

- `devices: all`
- `mig-enabled: true`
- `mig-devices: {}`

`all-1g.10gb:`

- `devices: all`
- `mig-enabled: true`
- `mig-devices:`

`1g.5gb: 7`

`all-1g.10gb.me:`

- `devices: all`
- `mig-enabled: true`
- `mig-devices:`

`1g.10gb+me: 1`

`all-1g.20gb:`

- `devices: all`
- `mig-enabled: true`
- `mig-devices:`

`1g.20gb: 4`

`all-2g.20gb:`

- `devices: all`
- `mig-enabled: true`
- `mig-devices:`

```

    2g.20gb: 3
all-3g.40gb:
  - devices: all
    mig-enabled: true
    mig-devices:
      3g.40gb: 2
all-4g.40gb:
  - devices: all
    mig-enabled: true
    mig-devices:
      4g.40gb: 1
all-7g.80gb:
  - devices: all
    mig-enabled: true
    mig-devices:
      7g.80gb: 1
all-balanced:
  - device-filter: ["0x233110DE", "0x232210DE", "0x20B210DE", "0x20B510DE", "0x20F310
DE", "0x20F510DE"]
    devices: all
    mig-enabled: true
    mig-devices:
      1g.10gb: 2
      2g.20gb: 1
      3g.40gb: 1
# 设置后会按照设置规格切分 CI 实例
custom-config:
  - devices: all
    mig-enabled: true
    mig-devices:
      3g.40gb: 2

```

在上述的 **YAML** 中设置 **custom-config**，设置后会按照规格切分 **CI** 实例。

```

custom-config:
  - devices: all
    mig-enabled: true
    mig-devices:
      1c.3g.40gb: 6

```

设置完成后，在确认部署应用时即可[使用 GPU MIG 资源](#)。

!!! note

若后续更新了切分策略配置文件，需要重启 nvidia-mig-manager，否则切分策略不生效。

切换节点 GPU 模式

当我们成功安装 gpu-operator 之后，节点默认是整卡模式，在节点管理页面会有标识，如

下图所示：

| 名称 | 状态 | 角色 | 标签 | CPU (分配/使用率) | 内存 (分配/使用率) | IP 地址 | 创建时间 |
|--------------------------------|----|------|---------------------------------|---------------|---------------|---------------|------------------|
| worker-01 [Nvidia-vGPU] | 健康 | 工作节点 | beta.kubernetes.io/os:linux +81 | 3.81% / 1.99% | 1.76% / 6.28% | 172.30.40.184 | 2024-06-26 10:03 |
| controller-node-1 [Nvidia-GPU] | 健康 | 工作节点 | beta.kubernetes.io/os:linux +96 | 7.42% / 1.91% | 1.23% / 2.09% | 10.20.100.31 | 2024-06-25 09:43 |

mixed

点击节点列表右侧的 ，选择 **GPU 模式切换**，然后选择对应的 **MIG 模式**以及切分的策略，这里以 **MIXED** 模式为例：

选择模式

- 整卡模式
- 虚拟化模式
- MIG 模式

Mig 策略: mixed

切分策略: all-1g-10gb

请根据 MIG 模式和 GPU 卡型号，在 config 中选择正确的切分策略并填入。
示例 all-1g-5gb。

mig

这里一共有两个配置：

1. Mig 策略：Mixed 以及 Single 。

2. 切分策略：这里的策略需要与 `default-mig-parted-config`（或者用户自定义的切分策略）

配置文件中的 key 保持一致。

点击确认按钮后，等待约一分钟左右刷新页面，MIG 模式切换成：

| 名称 | 状态 | 角色 | 标签 | CPU (分配/使用率) | 内存 (分配/使用率) | IP 地址 | 创建时间 |
|-----------------------------|----|------|----------------------------------|---------------|---------------|---------------|------------------|
| worker-01 Nvidia vGPU | 健康 | 工作节点 | beta.kubernetes.io/os:linux +0:1 | 3.81% / 2.25% | 1.76% / 6.3% | 172.30.40.184 | 2024-06-26 10:03 |
| controller-node-1 Nvidia MG | 健康 | 工作节点 | beta.kubernetes.io/os:linux +1:0 | 7.11% / 4.71% | 1.18% / 2.06% | 10.20.100.33 | 2024-06-25 09:43 |

切换 mig

使用 MIG GPU 资源

本节介绍应用如何使用 MIG GPU 资源。

前提条件

- 已经[部署 DCE 5.0](#) 容器管理平台，且平台运行正常。
- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 已安装[GPU Operator](#)。
- 集群节点上具有[对应型号的 GPU 卡](#)

UI 界面使用 MIG GPU

1. 确认集群是否已识别 GPU 卡类型

进入 **集群详情 -> 节点管理**，查看是否已正识别为 MIG 模式。

| 名称 | 状态 | 角色 | 标签 | CPU (分配/使用率) | 内存 (分配/使用率) | IP 地址 | 创建时间 |
|------------------------------|---------------------------------------|------|----------------------------------|---------------|---------------|---------------|------------------|
| worker-01 Nvidia vGPU | 健康 | 工作节点 | beta.kubernetes.io/os:linux +81 | 3.81% / 2.25% | 1.76% / 6.3% | 172.30.40.184 | 2024-06-26 10:03 |
| controller-node-1 Nvidia MIG | 健康 | 工作节点 | beta.kubernetes.io/os:linux +109 | 7.11% / 4.71% | 1.18% / 2.06% | 10.20.100.31 | 2024-06-25 09:43 |

gpu

2. 通过镜像部署应用，可选择并使用 NVIDIA MIG 资源。

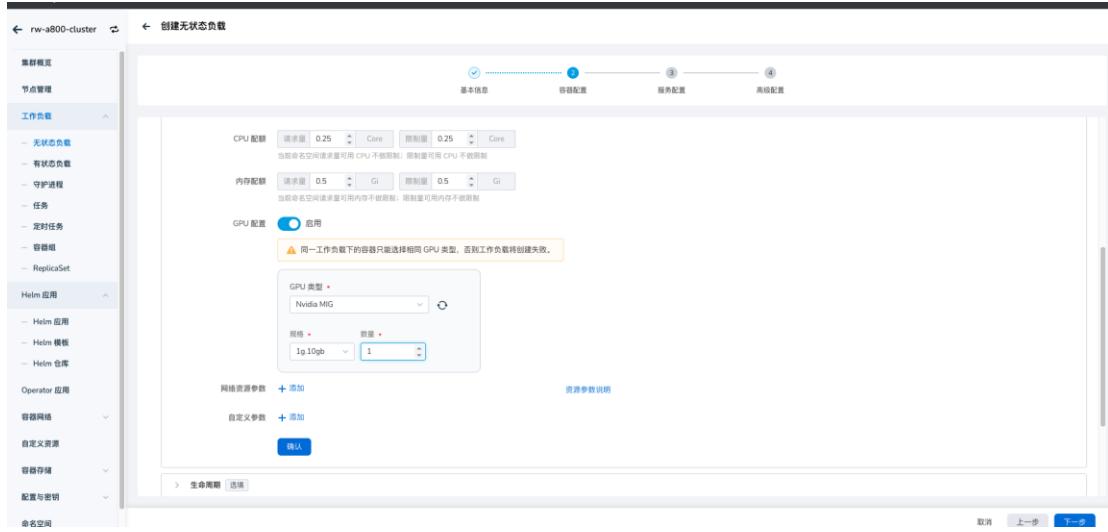
- MIG Single 模式示例（与整卡使用方式相同）：

!!! note

MIG single 策略允许用户以与 GPU 整卡相同的方式(`nvidia.com/gpu`)请求和使用 GPU 资源，不同的是这些资源可以是 GPU 的一部分(MIG 设备)，而不是整个 GPU。了解更多![GPU MIG 模式设计](<https://docs.google.com/document/d/1bshSICwNYRZGfywgwRHa07C0qRyOYKxWYxC1beJM-WM/edit#heading=h.jklusl667vn2>)

usemig

- MIG Mixed 模式示例：



mig02

YAML 配置使用 MIG

MIG Single__ 模式：

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: mig-demo
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mig-demo
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: mig-demo
    spec:
      containers:
        - name: mig-demo1
          image: chrstnhntschl/gpu_burn
          resources:
            limits:
              nvidia.com/gpu: 2 # (1)!
```

```
imagePullPolicy: Always
restartPolicy: Always
```

1. 申请 MIG GPU 的数量

MIG Mixed 模式 :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mig-demo
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mig-demo
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: mig-demo
    spec:
      containers:
        - name: mig-demo1
          image: chrstnhntschl/gpu_burn
          resources:
            limits:
              nvidia.com/mig-4g.20gb: 1 # (1)!
          imagePullPolicy: Always
          restartPolicy: Always
```

1. 通过 nvidia.com/mig-g.gb 的资源类型公开各个 MIG 设备

进入容器后可以查看只使用了一个 MIG 设备。

```
mig03
mig03
```

MIG 相关命令

GI 相关命名 :

| 子命令 | 说明 |
|---------------------------------------|-------------------------|
| nvidia-smi mig -lgi | 查看创建 GI 实例列表 |
| nvidia-smi mig -dgi -gi {Instance ID} | 删除指定的 GI 实例 |
| nvidia-smi mig -lgip | 查看 GI 的 profile |
| nvidia-smi mig -cgi {profile id} | 通过指定 profile 的 ID 创建 GI |

CI 相关命令：

| 子命令 | 说明 |
|---|---|
| nvidia-smi mig -lcip { -gi {gi Instance ID}} | 查看 CI 的 profile，指定 -gi 可以查看特定 GI 实例可以创建的 CI |
| nvidia-smi mig -lci | 查看创建的 CI 实例列表 |
| nvidia-smi mig -cci {profile id} -gi {gi instance id} | 指定的 GI 创建 CI 实例 |
| nvidia-smi mig -dci -ci {ci instance id} | 删除指定 CI 实例 |

GI+CI 相关命令：

| 子命令 | 说明 |
|---|-----------------|
| nvidia-smi mig -i 0 -cgi {gi profile id} -C {ci profile id} | 直接创建 GI + CI 实例 |

GPU 配额管理

本节介绍如何在 DCE 5.0 平台使用 vGPU 能力。

前提条件

当前集群已通过 Operator 或手动方式部署对应类型 GPU 驱动（NVIDIA GPU、NVIDIA MIG、

天数、昇腾)

操作步骤

1. 进入 Namespaces 中，点击 配额管理 可以配置当前 Namespace 可以使用的 GPU 资源。

配额管理

配额管理

2. 当前命名空间配额管理覆盖的卡类型为：NVIDIA vGPU、NVIDIA MIG、天数、昇腾。

NVIDIA vGPU 配额管理：配置具体可以使用的配额，会创建 ResourcesQuota CR：

- 物理卡数量 (nvidia.com/vgpu)：表示当前 POD 需要挂载几张物理卡，并且要 小于等于 宿主机上的卡数量。
- GPU 算力 (nvidia.com/gpucores)：表示每张卡占用的 GPU 算力，值范围为 0-100；如果配置为 0，则认为不强制隔离；配置为 100，则认为独占整张卡。
- GPU 显存 (nvidia.com/gpumem)：表示每张卡占用的 GPU 显存，值单位为 MB，最小值为 1，最大值为整卡的显存值。

卡类型

卡类型

GPU 资源动态调节

提供 GPU 资源动态调整功能，允许您在无需重新加载、重置或重启整个运行环境的情况下，对已经分配的 vGPU 资源进行实时、动态的调整。这一功能旨在最大程度地减少对

业务运行的影响，确保您的业务能够持续稳定地运行，同时根据实际需求灵活调整 GPU 资源。

使用场景

- **弹性资源分配**：当业务需求或工作负载发生变化时，可以快速调整 GPU 资源以满足新的性能要求。
- **即时响应**：在面对突发的高负载或业务需求时，可以迅速增加 GPU 资源而无需中断业务运行，以确保服务的稳定性和性能。

操作步骤

以下是一个具体的操作示例，展示如何在不重启 vGPU Pod 的情况下动态调整 vGPU 的算力和显存资源：

创建一个 vGPU Pod

首先，我们使用以下 YAML 创建一个 vGPU Pod，其算力初始不限制，显存限制为 200Mb。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: gpu-burn-test
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gpu-burn-test
  template:
    metadata:
      creationTimestamp: null
```

```

labels:
  app: gpu-burn-test
spec:
  containers:
    - name: container-1
      image: docker.io/chrstnhtschl/gpu_burn:latest
      command:
        - sleep
        - '100000'
      resources:
        limits:
          cpu: 1m
          memory: 1Gi
          nvidia.com/gpucores: '0'
          nvidia.com/gpumem: '200'
          nvidia.com/vgpu: '1'

```

调整前查看 Pod 中的资源 GPU 分配资源：

```

root@gpu-burn-test-5f45546dcc-v95hd:~# nvidia-smi
[HAMi-core Msg(23:140035311699776:libvgpu.c:873)]: Initializing.....
Thu Sep 12 02:24:53 2024
+-----+-----+-----+
| NVIDIA-SMI 535.54.03 | Driver Version: 535.54.03 | CUDA Version: 12.2 |
+-----+-----+-----+
GPU  Name     Persistence-M	Bus-Id     Disp.A	Volatile Uncorr. ECC					
Fan  Temp     Perf            Pwr:Usage/Cap	Memory-Usage	GPU-Util Compute M.					
+-----+-----+-----+-----+-----+-----+-----+							
0  Tesla P4      Off  00000000:03:00.0 Off	0MiB / 200MiB	0%   Default					
N/A   36C   P8          6W / 75W	0MiB / 200MiB	0%   Default					
+-----+-----+-----+-----+-----+-----+-----+							
Processes:                               GPU Memory							
GPU  GI  CI   PID  Type  Process name        Usage							
ID  ID							
+-----+-----+-----+							
No running processes found							
+-----+
[HAMi-core Msg(23:140035311699776:multiprocess_memory_limit.c:516)]: Calling exit handler 23
root@gpu-burn-test-5f45546dcc-v95hd:~# █

```

gpu-dynamic-regulation-before.png

动态调整算力

如果需要修改算力为 10%，可以按照以下步骤操作：

1. 进入容器：

```
kubectl exec -it <pod-name> -- /bin/bash
```

2. 执行：

```
export CUDA_DEVICE_SM_LIMIT=10
```

3. 在当前终端直接运行：

```
./gpu_burn 60
```

程序即可生效。注意，不能退出当前 Bash 终端。

动态调整显存

如果需要修改显存为 300 MB，可以按照以下步骤操作：

1. 进入容器：

```
kubectl exec -it <pod-name> -- /bin/bash
```

2. 执行以下命令来设置显存限制：

```
export CUDA_DEVICE_MEMORY_LIMIT_0=300m
export CUDA_DEVICE_MEMORY_SHARED_CACHE=/usr/local/vgpu/d.cache
```

!!! note

每次修改显存大小时，`d.cache` 这个文件名字都需要修改，比如改为 `a.cache`、`1.cache` 等，以避免缓存冲突。

3. 在当前终端直接运行：

```
./gpu_burn 60
```

程序即可生效。同样地，不能退出当前 Bash 终端。

调整后查看 Pod 中的资源 GPU 分配资源：

```
root@gpu-burn-test-5f45546dcc-v95hd:~# export CUDA_DEVICE_MEMORY_LIMIT_0=300m
root@gpu-burn-test-5f45546dcc-v95hd:~# export CUDA_DEVICE_MEMORY_SHARED_CACHE=/usr/local/vgpu/d.cache
root@gpu-burn-test-5f45546dcc-v95hd:~# nvidia-smi
[HAMi-core Msg(24:139824279263040:libvgpu.c:873)]: Initializing.....
Thu Sep 12 02:26:20 2024
+-----+-----+-----+
| NVIDIA-SMI 535.54.03 | Driver Version: 535.54.03 | CUDA Version: 12.2 |
+-----+-----+-----+
GPU  Name   Persistence-M	Bus-Id      Disp.A	Volatile Uncorr. ECC		
Fan  Temp   Perf          Pwr:Usage/Cap	Memory-Usage	GPU-Util  Compute M.		
				MIG M.
+-----+-----+-----+				
0  Tesla P4           Off	00000000:03:00.0	Off        0%     Default		
N/A   36C   P8          6W / 75W	0MiB /  300MiB	0%    N/A		
+-----+-----+-----+				
Processes:          GPU Memory Usage				
PID  Type  Process name          ID  ID				
No running processes found				
+-----+
[HAMi-core Msg(24:139824279263040:multiprocess_memory_limit.c:516)]: Calling exit handler 24
root@gpu-burn-test-5f45546dcc-v95hd:~#
```

gpu-dynamic-regulation-after.png

通过上述步骤，您可以在不重启 vGPU Pod 的情况下动态地调整其算力和显存资源，从而

更灵活地满足业务需求并优化资源利用。

GPU 监控指标

本页列出一些常用的 GPU 监控指标。

集群维度

| 指标名称 | 描述 |
|-------------|--|
| GPU 卡数 | 集群下所有的 GPU 卡数量 |
| GPU 平均使用率 | 集群下所有 GPU 卡的平均算力使用率 |
| GPU 平均显存使用率 | 集群下所有 GPU 卡的平均显存使用率 |
| GPU 卡功率 | 集群下所有 GPU 卡的功率 |
| GPU 卡温度 | 集群下所有 GPU 卡的温度 |
| GPU 算力使用率细节 | 24 小时内，集群下所有 GPU 卡的使用率细节（包含 max、avg、current） |
| GPU 显存使用量细节 | 24 小时内，集群下所有 GPU 卡的显存使用量细节（包含 min、max、avg、current） |
| GPU 显存带宽使用率 | 表示内存带宽利用率。以 Nvidia GPU V100 为例，其最大内存带宽为 900 GB/sec，如果当前的内存带宽为 450 GB/sec，则内存带宽利用率为 50% |

节点维度

| 指标名称 | 描述 |
|-------------|--------------------------------------|
| GPU 模式 | 节点上 GPU 卡的使用模式，包含整卡模式、MIG 模式、vGPU 模式 |
| GPU 物理卡数 | 节点上所有的 GPU 卡数量 |
| GPU 虚拟卡数 | 节点上已经被创建出来的 vGPU 设备数量 |
| GPU MIG 实例数 | 节点上已经被创建出来的 MIG 实例数 |
| GPU 显存分配率 | 节点上所有 GPU 卡的显存分配率 |
| GPU 算力平均使用率 | 节点上所有 GPU 卡的算力平均使用率 |
| GPU 显存平均使用率 | 节点上所有 GPU 卡的平均显存使用率 |
| GPU 驱动版本 | 节点上 GPU 卡驱动的版本信息 |
| GPU 算力使用率细节 | 24 小时内，节点上每张 GPU 卡的算力使用率细 |

| 指标名称 | 描述 |
|-----------|---|
| GPU 显存使用量 | 节 (包含 max、avg、current)
24 小时内 , 节点上每张 GPU 卡的显存使用量细 |
| | 节 (包含 min、max、avg、current) |

根据 XID 状态排查 GPU 相关问题

XID 消息是 NVIDIA 驱动程序向操作系统的内核日志或事件日志打印的错误报告。XID 消息用于标识 GPU 错误事件，提供 GPU 硬件、NVIDIA 软件或应用中的错误类型、错误位置、错误代码等信息。如检查某 GPU 节点上的 XID 异常为空，表明无 XID 消息；如有，您可按照下表自助排查并解决问题，或查看[更多 XID 消息](#)。

| XID | 消息 | 说明 |
|-----|--|--|
| 13 | Graphics Engine Exception. | 通常是数组越界、指令错误，小概率是硬件问题。 |
| 31 | GPU memory page fault. | 通常是应用程序的非法地址访问，极小概率是驱动或者硬件问题。 |
| 32 | Invalid or corrupted push buffer stream. | 事件由 PCIE 总线上管理 NVIDIA 驱动和 GPU 之间通信的 DMA 控制器上报，通常是 PCI 质量问题导致，而 |

| XID | 消息 | 说明 |
|-----|--|--|
| 38 | Driver firmware error. | 通常是驱动固件错误而非硬件问题。 |
| 43 | GPU stopped processing. | 通常是您应用自身错误，而非硬件问题。 |
| 45 | Preemptive cleanup, due to previous errors – Most likely to see when running multiple cuda applications and hitting a DBE. | 通常是您手动退出或者其他故障（硬件、资源限制等）导致的 GPU 应用退出，XID 45 只提供一个结果，具体原因通常需要进一步分析日志。 |
| 48 | Double Bit ECC Error (DBE). | 当 GPU 发生不可纠正的错误时，会上报此事件，该错误也会同时反馈给您的应用程序。通常需要重置 GPU 或重启节点来清除这个错误。 |
| 61 | Internal micro-controller breakpoint/warning. | GPU 内部引擎停止工作，您的业务已经受到影响。 |
| 62 | Internal micro-controller halt. | 与 XID 61 的触发场景类似。 |

| XID | 消息 | 说明 |
|-----|--|--|
| 63 | ECC page retirement or row remapping recording event. | 当应用程序遭遇到 GPU 显存硬件错误时，NVIDIA 自纠错机制会将错误的内存区域 retire 或者 remap，retirement 和 remapped 信息需记录到 infoROM 中才能永久生效。Volt 架构：成功记录 ECC page retirement 事件到 infoROM。Ampere 架构：成功记录 row remapping 事件到 infoROM。 |
| 64 | ECC page retirement or row remapper recording failure. | 与 XID 63 的触发场景类似。但 XID 63 代表 retirement 和 remapped 信息成功记录到了 infoROM，XID 64 代表该记录操作失败。 |
| 68 | NVDECO Exception. | 通常是硬件或驱动问题。 |
| 74 | NVLINK Error. | NVLINK 硬件错误产生的 XID，表明 GPU 已经出现严 |

| XID | 消息 | 说明 |
|-----|---------------------------------|--|
| 79 | GPU has fallen off the bus. | 重硬件故障，需要下线维修。
GPU 硬件检测到掉卡，总线上无法检测该 GPU，表明该 GPU 已经出现严重硬件故障，需要下线维修。 |
| 92 | High single-bit ECC error rate. | 硬件或驱动故障。 |
| 94 | Contained ECC error. | 当应用程序遭遇到 GPU 不可纠正的显存 ECC 错误时，NVIDIA 错误抑制机制会尝试将错误抑制在发生硬件故障的应用程序，避免该错误影响 GPU 节点上运行的其他应用程序。当抑制机制成功抑制错误时，会产生该事件，仅出现不可纠正 ECC 错误的应用程序受到影响。 |
| 95 | Uncontained ECC error. | 与 XID 94 的触发场景类似。但 XID 94 代表抑制成功，而 XID 95 代表抑制失败，表明运行在该 GPU 上 |

| XID | 消息 | 说明 |
|-----|----|----------------|
| | | 的所有应用程序都已受到影响。 |

Pod 维度

| 分类 | 指标名称 | 描述 |
|----------------------|-----------------|--|
| 应用概览 GPU 卡 - 算力 & 显存 | Pod GPU 算力使用率 | 当前 Pod 所使用到的 GPU 卡的算力使用率 |
| | Pod GPU 显存使用率 | 当前 Pod 所使用到的 GPU 卡的显存使用率 |
| | Pod 显存使用量 | 当前 Pod 所使用到的 GPU 卡的显存使用量 |
| | 显存分配量 | 当前 Pod 所使用到的 GPU 卡的显存分配量 |
| | Pod GPU 显存复制使用率 | 当前 Pod 所使用到的 GPU 卡的显存显存复制比率 |
| GPU 卡 - 引擎概览 | GPU 图形引擎活动百分比 | 表示在一个监控周期内，Graphics 或 Compute 引擎处于 Active 的时间占总的时间 |

| 分类 | 指标名称 | 描述 |
|----|----------------|---|
| | GPU 内存带宽利用率 | <p>间的比例</p> <p>表示内存带宽利用率 (Memory BW Utilization) 将数据发送到设备内存或从设备内存接收数据的周期分數。该值表示时间间隔内的平均值，而不是瞬时值。较高的值表示设备内存的利用率较高。</p> <p>该值为 1 (100%) 表示在整个时间间隔内的每个周期执行一条 DRAM 指令 (实际上，峰值约为 0.8 (80%) 是可实现的最大值)。假设该值为 0.2 (20%)，表示 20% 的周期在时间间隔内读取或写入设备内存。</p> <p>表示在一个监控周期</p> |
| | Tensor 核心引擎使用率 | |

| 分类 | 指标名称 | 描述 |
|-----------------|------------|---|
| | FP16 引擎使用率 | 内 , Tensor Core 管道
(Pipe) 处于 Active 时
间占总时间的比例
表示在一个监控周期 |
| | FP32 引擎使用率 | 内 , FP16 管道处于
Active 的时间占总的时间
的比例
表示在一个监控周期 |
| | FP64 引擎使用率 | 内 , FP32 管道处于
Active 的时间占总的时间
的比例
表示在一个监控周期 |
| GPU 卡 - 温度 & 功耗 | GPU 卡温度 | GPU 卡解码引擎比率
GPU 卡编码引擎比率
集群下所有 GPU 卡的
温度 |
| | GPU 卡功率 | 集群下所有 GPU 卡的
功率 |

| 分类 | 指标名称 | 描述 |
|---------------|-----------------|---|
| | GPU 卡 - 总耗能 | GPU 卡总共消耗的能量 |
| GPU 卡 - Clock | GPU 卡内存频率 | 内存频率 |
| | GPU 卡应用 SM 时钟频率 | 应用的 SM 时钟频率 |
| | GPU 卡应用内存频率 | 应用内存频率 |
| | GPU 卡视频引擎频率 | 视频引擎频率 |
| | GPU 卡降频原因 | 降频原因 |
| GPU 卡 - 其他细节 | 图形引擎活动 | 图形或计算引擎的任何部分处于活动状态的时间比例。如果图形/计算上下文已绑定且图形/计算管道繁忙，则图形引擎处于活动状态。该值表示时间间隔内的平均值，而不是瞬时值。 |
| | SM 活动 | 多处理器上至少一个 Warp 处于活动状态的时间比例，所有多处理器的平均值。请注意，“活动”并不一定意味着 Warp 正在积极计算。例如，等待内存请求的 |

| 分类 | 指标名称 | 描述 |
|----|------|--|
| | | <p>Warp 被视为活动状态。该值表示时间间隔内的平均值，而不是瞬时值。0.8 或更大的值是有效使用 GPU 的必要条件，但还不够。小于 0.5 的值可能表示 GPU 使用效率低下。给出一个简化的 GPU 架构视图，如果 GPU 有 N 个 SM，则使用 N 个块并在整个时间间隔内运行的内核将对应于活动 1 (100%)。使用 N/5 个块并在整个时间间隔内运行的内核将对应于活动 0.2 (20%)。使用 N 个块并运行五分之一时间间隔的内核，如果 SM 处于空闲状态，则活动也将为</p> |

| 分类 | 指标名称 | 描述 |
|--------|------|---|
| | | 0.2 (20%)。该值与每
个块的线程数无关 (参
见 DCGM_FI_PROF_SM_OCC
UPANCY)。 |
| SM 入住率 | | 多处理器上驻留 Warp
的比例 , 相对于多处理
器上支持的最大并发
Warp 数。该值表示时
间间隔内的平均值 , 而
不是瞬时值。占用率越
高并不一定表示 GPU
使用率越高。对于 GPU
内存带宽受限的工作负
载 (参见 DCGM_FI_PROF_DRAM_
ACTIVE) , 占用率越高
表明 GPU 使用率越
高。但是 , 如果工作负
载是计算受限的 (即不
受 GPU 内存带宽或延
迟限制) , 则占用率越 |

| 分类 | 指标名称 | 描述 |
|------|------------------|--|
| | | <p>高并不一定与 GPU 使用率越高相关。计算占用率并不简单，它取决于 GPU 属性、每个块的线程数、每个线程的寄存器以及每个块的共享内存等因素。使用 CUDA 占用率计算器 探索各种占用率场景。</p> |
| 张量活动 | 张量 (HMMA / IMMA) | <p>管道处于活动状态的周期分数。该值表示时间间隔内的平均值，而不是瞬时值。值越高，张量核心的利用率越高。</p> <p>活动 1 (100%) 相当于在整个时间间隔内每隔一个周期发出一个张量指令。活动 0.2 (20%) 可能表示 20% 的 SM 在整个时间段内的利用</p> |

| 分类 | 指标名称 | 描述 |
|-----------|---|--|
| FP64 引擎活动 | FP64 (双精度) 管道处于活动状态的周期数。该值表示时间间隔内的平均值，而不是瞬时值。值越高，FP64 核心的利用率越高。活动量 1 (100%) 相当于整个时间间隔内 Volta 上每四个周期的每个 SM 上执行一条 FP64 指令。活动量 0.2 | 率为 100% , 100% 的 SM 在整个时间段内的利用率为 20% , 100% 的 SM 在 20% 的时间段内的利用率为 100% , 或者介于两者之间的任何组合 (请参阅 DCGM_FI_PROF_SM_ACT IVE 以帮助消除这些可能性的歧义) 。 |
| FP64 引擎活动 | FP64 (双精度) 管道处于活动状态的周期数。该值表示时间间隔内的平均值，而不是瞬时值。值越高，FP64 核心的利用率越高。活动量 1 (100%) 相当于整个时间间隔内 Volta 上每四个周期的每个 SM 上执行一条 FP64 指令。活动量 0.2 | 率为 100% , 100% 的 SM 在整个时间段内的利用率为 20% , 100% 的 SM 在 20% 的时间段内的利用率为 100% , 或者介于两者之间的任何组合 (请参阅 DCGM_FI_PROF_SM_ACT IVE 以帮助消除这些可能性的歧义) 。 |
| FP64 引擎活动 | FP64 (双精度) 管道处于活动状态的周期数。该值表示时间间隔内的平均值，而不是瞬时值。值越高，FP64 核心的利用率越高。活动量 1 (100%) 相当于整个时间间隔内 Volta 上每四个周期的每个 SM 上执行一条 FP64 指令。活动量 0.2 | 率为 100% , 100% 的 SM 在整个时间段内的利用率为 20% , 100% 的 SM 在 20% 的时间段内的利用率为 100% , 或者介于两者之间的任何组合 (请参阅 DCGM_FI_PROF_SM_ACT IVE 以帮助消除这些可能性的歧义) 。 |

| 分类 | 指标名称 | 描述 |
|-----------|---|---|
| | | (20%) 可能表示 20% 的 SM 在整个时间段内 |
| | | 利用率率为 100% , 100% 的 SM 在整个时间段内 |
| | | 利用率率为 20% , 100% 的 SM 在 20% 的时间 |
| | | 段内利用率为 100% , 或者介于两者之间的任 |
| | | 何组合 (请参阅 DCGM_FI_PROF_SM_ACT) 以帮助消除这些可能性的歧义) 。 |
| FP32 引擎活动 | FMA (FP32 (单精度) 和整数) 管道处于活动状态的周期分数。该值表示时间间隔内的平均值 , 而不是瞬时值。值越高 , FP32 核心的利用率越高。活动量 1 (100%) 相当于整个时间间隔内每隔一个周期 | |

| 分类 | 指标名称 | 描述 |
|-----------|---|---|
| | FP32 指令活动量 | 执行一次 FP32 指令。
活动量 0.2 (20%) 可能表示 20% 的 SM 在整个时间段内利用率为 100% , 100% 的 SM 在整个时间段内利用率为 20% , 100% 的 SM 在 20% 的时间段内利用率为 20% , 或者两者之间的任何组合 (请参阅 DCGM_FI_PROF_SM_ACT) 以帮助消除这些可能性的歧义) 。 |
| FP16 引擎活动 | FP16 (半精度) 管道处于活动状态的周期分数。该值表示时间间隔内的平均值 , 而不是瞬时值。值越高 , FP16 核心的利用率越高。活动量 1 (100%) 相当于整个时间间隔内每隔一个 | FP16 (半精度) 管道处于活动状态的周期分数。该值表示时间间隔内的平均值 , 而不是瞬时值。值越高 , FP16 核心的利用率越高。活动量 1 (100%) 相当于整个时间间隔内每隔一个 |
| | FP16 引擎活动 | FP16 引擎活动 |

| 分类 | 指标名称 | 描述 |
|----|---------|--|
| | | 周期执行一次 FP16 指令。活动量 0.2 (20%) |
| | | 可能表示 20% 的 SM 在整个时间段内利用率 |
| | | 为 100% , 100% 的 SM |
| | | 在整个时间段内利用率 |
| | | 为 20% , 100% 的 SM |
| | | 在 20% 的时间段内利用率为 100% , 或者介于两者之间的任何组合 |
| | | (请参阅 DCGM_FI_PROF_SM_ACT) |
| | | IVE 以帮助消除这些可能性的歧义) 。 |
| | 内存带宽利用率 | 向设备内存发送数据或从设备内存接收数据的周期比例。该值表示时间间隔内的平均值 , 而不是瞬时值。值越高 , 设备内存的利用率为 1 (100%) |

| 分类 | 指标名称 | 描述 |
|-----------|------|------------------|
| | | 相当于整个时间间隔内 |
| | | 每个周期执行一条 |
| | | DRAM 指令 (实际上 , |
| | | 峰值约为 0.8 (80%) 是 |
| | | 可实现的最大值)。活 |
| | | 动率为 0.2 (20%) 表示 |
| | | 在时间间隔内有 20% |
| | | 的周期正在读取或写入 |
| | | 设备内存。 |
| NVLink 带宽 | | 通过 NVLink 传输/接收 |
| | | 的数据速率 (不包括协 |
| | | 议标头) , 以每秒字节 |
| | | 数为单位。该值表示一 |
| | | 段时间内的平均值 , 而 |
| | | 不是瞬时值。速率是一 |
| | | 段时间内的平均值。例 |
| | | 如 , 如果 1 秒内传输了 |
| | | 1 GB 的数据 , 则无论数 |
| | | 据是以恒定速率还是突 |
| | | 发速率传输 , 速率都是 |
| | | 1 GB/s。理论上 , 每个 |

| 分类 | 指标名称 | 描述 |
|-----------|--|-----------------------|
| | NVLink 带宽 | 链路每个方向的最大带宽为 25 GB/s。 |
| PCIe 带宽 | 通过 PCIe 总线传输/接收的数据速率，包括协议头和数据有效负载，以字节/秒为单位。 | |
| | 该值表示一段时间内的平均值，而不是瞬时值。该速率是一段时间内的平均值。例如，如果 1 秒内传输了 1 GB 的数据，则无论数据是以恒定速率还是突发速率传输，速率都是 1 GB/s。理论上最大 PCIe Gen3 带宽为每通道 985 MB/s。 | |
| PCIe 传输速率 | 节点 GPU 卡通过 PCIe 总线传输的数据速率 | |
| PCIe 接收速率 | 节点 GPU 卡通过 PCIe | |

| 分类 | 指标名称 | 描述 |
|----|------|-----------|
| | | 总线接收的数据速率 |

GPU 告警规则

本文介绍如何在 DCE 5.0 平台设置 GPU 相关的告警规则。

前置条件

- 集群节点上已正确安装 GPU 设备
- 集群中已正确安装 [gpu-operator 组件](#)
- 如果用到了 vGPU 还需要在集群中安装 [Nvidia-vgpu 组件](#)，并且开启 servicemonitor
- 集群正确安装了 insight-agent 组件

告警常用 GPU 指标

本节介绍 GPU 告警常用的指标，分为两个部分：

- GPU 卡纬度的指标，主要反应单个 GPU 设备的运行状态。
- 应用纬度的指标，主要反应 Pod 在 GPU 上的运行状态。

GPU 卡指标

| 指标名称 | 指标单位 | 说明 |
|---------------------------|------|---------|
| DCGM_FI_DEV_GPU_UTIL | % | GPU 利用率 |
| DCGM_FI_DEV_MEM_COPY_UTIL | % | 显存利用率 |
| DCGM_FI_DEV_ENC_UTIL | % | 编码器利用率 |

| 指标名称 | 指标单位 | 说明 |
|-------------------------|------|---|
| DCGM_FI_DEV_DEC_UTIL | % | 解码器利用率 |
| DCGM_FI_DEV_FB_FREE | MB | 表示显存剩余量 |
| DCGM_FI_DEV_FB_USED | MB | 表示显存使用量 |
| DCGM_FI_DEV_GPU_TEMP | 摄氏度 | 表示当前 GPU 的温度度数 |
| DCGM_FI_DEV_POWER_USAGE | W | 设备电源使用情况 |
| DCGM_FI_DEV_XID_ERRORS | - | 表示一段时间内，最后发生
的 XID 错误号。XID 提供
GPU 硬件、NVIDIA 软件或
应用中的错误类型、错误位
置、错误代码等信息，更多
XID 信息 |

应用维度的指标

| 指标名称 | 指标单位 | 说明 |
|--------------------------------|------|---------------------|
| kpanda_gpu_pod_utilization | % | 表示 Pod 对 GPU 的使用率 |
| kpanda_gpu_mem_pod_usage | MB | 表示 Pod 对 GPU 显存的使用量 |
| kpanda_gpu_mem_pod_utilization | % | 表示 Pod 对 GPU 显存的使用率 |

设置告警规则

这里会介绍如何设置 GPU 告警规则，使用 GPU 卡利用率指标作为案例，请用户根据实际的业务场景选择指标以及编写 promql。

目标：当 GPU 卡利用率在五秒钟内一直保持 80% 的利用率时发出告警

1. 在可观测页面，点击 告警 -> 告警策略 -> 创建告警策略

The screenshot shows the 'Alert Strategies' section of the DaoCloud interface. On the left, there's a sidebar with various monitoring and management tabs like Overview, Dashboards, Infrastructure, Metrics, Logs, Tracing, and Alerts. Under the Alerts tab, 'Alert Strategies' is selected. In the main content area, there's a search bar and a table with columns for Alert Strategy Name, Cluster, Namespace, Resource Type, and Alert Object. A prominent blue button at the top right of the table area is labeled 'Create Alert Strategy'. A red arrow points to this button.

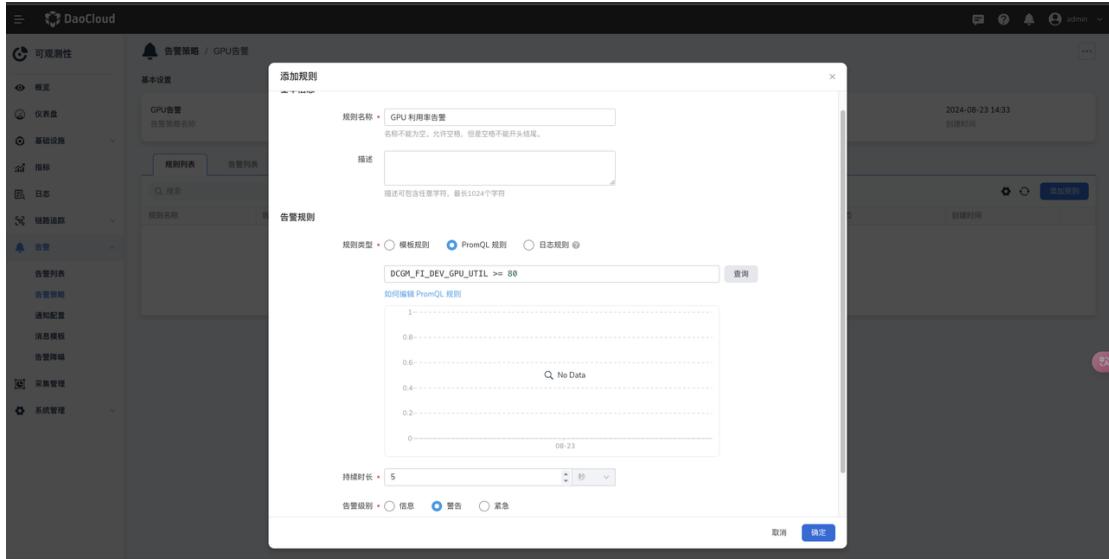
创建告警规则

2. 填写基本信息

This screenshot shows the first step of creating an alert strategy, titled 'Basic Information'. It includes fields for 'Strategy Name' (set to 'GPU报警'), 'Description' (empty), 'Cluster' (set to 'All Clusters'), 'Resource Type' (radio buttons for 'Cluster', 'Nodes', and 'Workload' with 'Nodes' selected), and 'Node' (set to 'All Nodes'). There are three numbered steps at the top: 1. Basic Information, 2. Add Rules, and 3. Notification Configuration. A red arrow points to the 'Next Step' button at the bottom right.

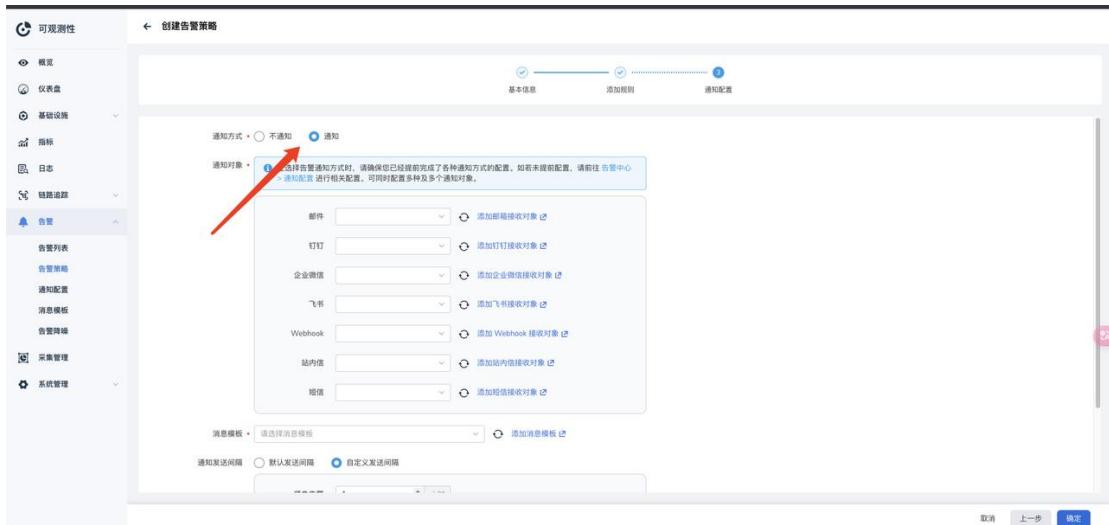
填写告警规则

3. 添加规则



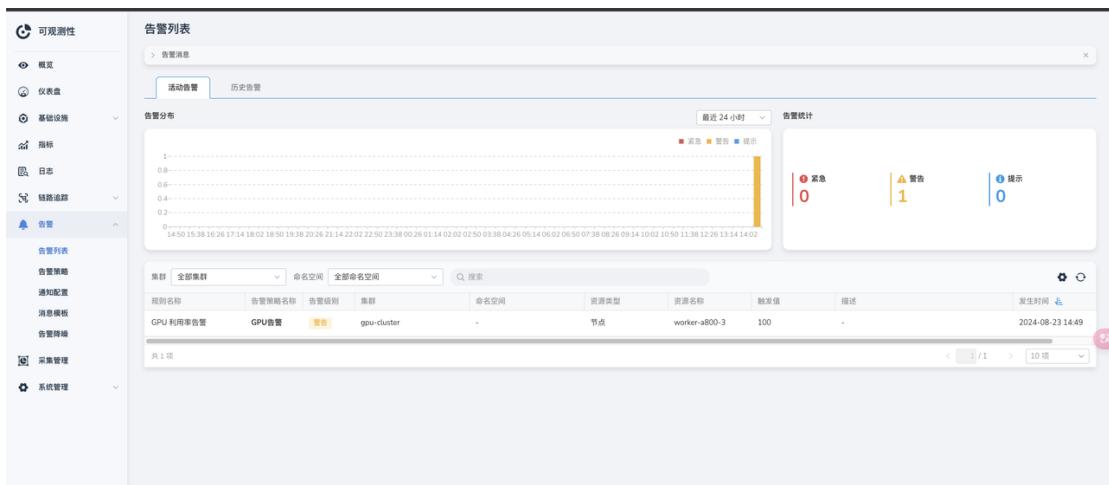
填写告警规则 2

4. 选择通知方式



通知方式

5. 设置完成后，当一个 GPU 在 5s 内一直保持 80% 的利用率，会收到如下的告警信息。



告警信息

安装 Volcano

随着 Kubernetes (K8s) 成为云原生应用编排与管理的首选平台，众多应用正积极向 K8s 迁移。在人工智能与机器学习领域，由于这些任务通常涉及大量计算，开发者倾向于在 Kubernetes 上构建 AI 平台，以充分利用其在资源管理、应用编排及运维监控方面的优势。

然而，Kubernetes 的默认调度器主要针对长期运行的服务设计，对于 AI、大数据等需要批量和弹性调度的任务存在诸多不足。例如，在资源竞争激烈的情况下，默认调度器可能导致资源分配不均，进而影响任务的正常执行。

以 TensorFlow 作业为例，其包含 PS（参数服务器）和 Worker 两种角色，两者需协同工作才能完成任务。若仅部署单一角色，作业将无法运行。而默认调度器对 Pod 的调度是逐个进行的，无法感知 TJob 中 PS 和 Worker 的依赖关系。在高负载情况下，这可能导致多个作业各自分配到部分资源，但均无法完成，从而造成资源浪费。

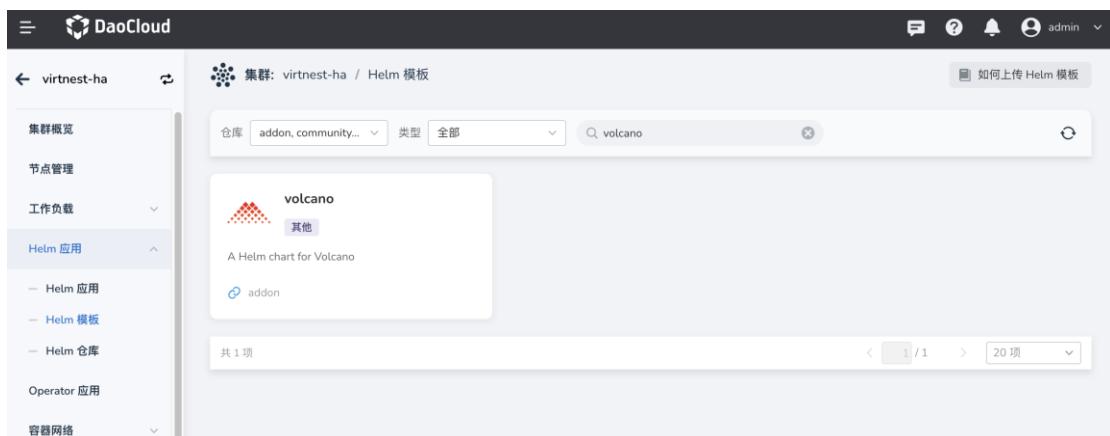
Volcano 的调度策略优势

Volcano 提供了多种调度策略，以应对上述挑战。其中，Gang-scheduling 策略能确保分布式机器学习训练过程中多个任务（Pod）同时启动，避免死锁；Preemption scheduling 策略则允许高优先级作业在资源不足时抢占低优先级作业的资源，确保关键任务优先完成。此外，Volcano 与 Spark、TensorFlow、PyTorch 等主流计算框架无缝对接，并支持 CPU 和 GPU 等异构设备的混合调度，为 AI 计算任务提供了全面的优化支持。

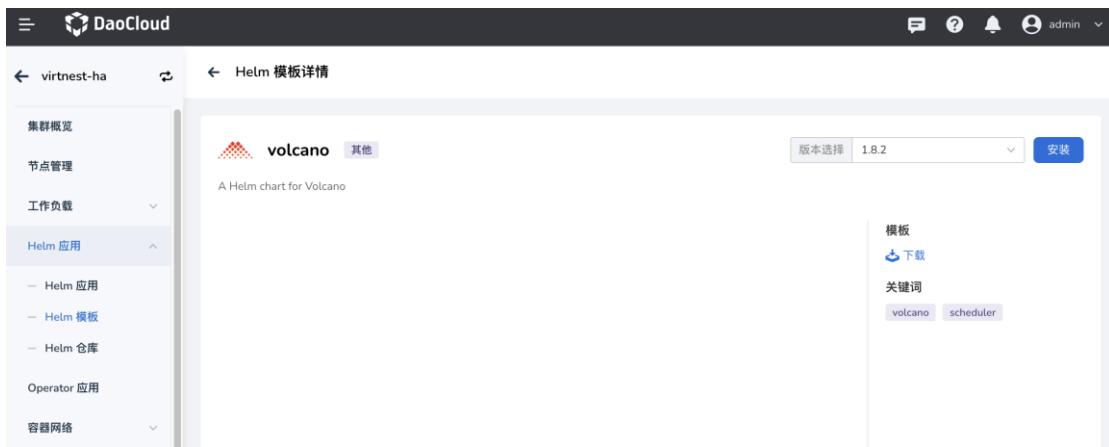
接下来，我们将介绍如何安装和使用 Volcano，以便您能够充分利用其调度策略优势，优化 AI 计算任务。

安装 Volcano

1. 在 **集群详情 -> Helm 应用 -> Helm 模板** 中找到 Volcano 并安装。



Volcano helm 模板



安装 Volcano

2. 检查并确认 Volcano 是否安装完成，即 volcano-admission、volcano-controllers、

volcano-scheduler 组件是否正常运行。

| 资源名称 | 类型 | 状态 | 命名空间 |
|------------------------------|--------------------------|-----|---------------|
| volcano-admission-role | ClusterRoleBinding | 运行中 | - |
| queues.scheduling.volcano.sh | CustomResourceDefinition | 运行中 | - |
| volcano-admission | Deployment | 运行中 | kpanda-system |
| volcano-controllers | Deployment | 运行中 | kpanda-system |
| volcano-scheduler | Deployment | 运行中 | kpanda-system |
| volcano-admission-init | Job | 运行中 | kpanda-system |
| volcano-admission | ClusterRole | 运行中 | - |
| volcano-scheduler | ClusterRole | 运行中 | - |

Volcano 组件

通常 Volcano 会和 [AI Lab 平台](#) 配合使用，以实现数据集、Notebook、任务训练的整个开发、训练流程的有效闭环。

使用 Volcano 的 Gang Scheduler

Gang 调度策略是 volcano-scheduler 的核心调度算法之一，它满足了调度过程中的 “All or

nothing”的调度需求，避免 Pod 的任意调度导致集群资源的浪费。具体算法是，观察 Job 下的 Pod 已调度数量是否满足了最小运行数量，当 Job 的最小运行数量得到满足时，为 Job 下的所有 Pod 执行调度动作，否则，不执行。

使用场景

基于容器组概念的 Gang 调度算法十分适合需要多进程协作的场景。AI 场景往往包含复杂的流程，Data Ingestion、Data Analysts、Data Splitting、Trainer、Serving、Logging 等，需要一组容器进行协同工作，就很适合基于容器组的 Gang 调度策略。MPI 计算框架下的多线程并行计算通信场景，由于需要主从进程协同工作，也非常适合使用 Gang 调度策略。容器组下的容器高度相关也可能存在资源争抢，整体调度分配，能够有效解决死锁。

在集群资源不足的场景下，Gang 的调度策略对于集群资源的利用率的提升是非常明显的。比如集群现在只能容纳 2 个 Pod，现在要求最小调度的 Pod 数为 3。那现在这个 Job 的所有的 Pod 都会 pending，直到集群能够容纳 3 个 Pod，Pod 才会被调度。有效防止调度部分 Pod，不满足要求又占用了资源，使其他 Job 无法运行的情况。

概念说明

Gang Scheduler 是 Volcano 的核心的调度插件，安装 Volcano 后默认就开启了。在创建工作负载时只需要指定调度器的名称为 Volcano 即可。

Volcano 是以 PodGroup 为单位进行调度的，在创建工作负载时，并不需要手动创建 PodGroup 资源，Volcano 会根据工作负载的信息自动创建。下面是一个 PodGroup 的示例：

```
apiVersion: scheduling.volcano.sh/v1beta1
kind: PodGroup
metadata:
```

```

name: test
namespace: default
spec:
  minMember: 1 # (1)!
  minResources: # (2)!
    cpu: "3"
    memory: "2048Mi"
  priorityClassName: high-prority # (3)!
  queue: default # (4)!

```

1. 表示该 PodGroup 下 **最少** 需要运行的 Pod 或任务数量。如果集群资源不满足

minMember 数量任务的运行需求，调度器将不会调度任何一个该 PodGroup 内的任务。

2. 表示运行该 PodGroup 所需要的最少资源。当集群可分配资源不满足 minResources

时，调度器将不会调度任何一个该 PodGroup 内的任务。

3. 表示该 PodGroup 的优先级，用于调度器为该 queue 中所有 PodGroup 进行调度时

进行排序。**system-node-critical** 和 **system-cluster-critical** 是 2 个预留的值，表示最高优先级。不特别指定时，默认使用 default 优先级或 zero 优先级。

4. 表示该 PodGroup 所属的 queue。queue 必须提前已创建且状态为 open。

使用案例

在 MPI 计算框架下的多线程并行计算通信场景中，我们要确保所有的 Pod 都能调度成功才能保证任务正常完成。设置 minAvailable 为 4，表示要求 1 个 mpimaster 和 3 个 mpiworker 能运行。

```

apiVersion: batch.volcano.sh/v1alpha1
kind: Job
metadata:
  name: lm-mpi-job
  labels:
    "volcano.sh/job-type": "MPI"
spec:
  minAvailable: 4

```

```

schedulerName: volcano
plugins:
  ssh: []
  svc: []
policies:
  - event: PodEvicted
    action: RestartJob
tasks:
  - replicas: 1
    name: mpimaster
    policies:
      - event: TaskCompleted
        action: CompleteJob
    template:
      spec:
        containers:
          - command:
              - /bin/sh
              - -c
              - |
                MPI_HOST=`cat /etc/volcano/mpiserver.host | tr "\n" ","`;
                mkdir -p /var/run/sshd; /usr/sbin/sshd;
                mpiexec --allow-run-as-root --host ${MPI_HOST} -np 3 mpi_hello_world;
            image: docker.m.daocloud.io/volcanosh/example-mpi:0.0.1
            name: mpimaster
            ports:
              - containerPort: 22
                name: mpijob-port
            workingDir: /home
        resources:
          requests:
            cpu: "500m"
          limits:
            cpu: "500m"
        restartPolicy: OnFailure
        imagePullSecrets:
          - name: default-secret
  - replicas: 3
    name: mpiworker
    template:
      spec:
        containers:
          - command:
              - /bin/sh

```

```

- -c
-
mkdir -p /var/run/sshd; /usr/sbin/sshd -D;
image: docker.m.daocloud.io/volcanosh/example-mpi:0.0.1
name: mpiworker
ports:
- containerPort: 22
  name: mpijob-port
workingDir: /home
resources:
requests:
  cpu: "1000m"
limits:
  cpu: "1000m"
restartPolicy: OnFailure
imagePullSecrets:
- name: default-secret

```

生成 PodGroup 的资源：

```

apiVersion: scheduling.volcano.sh/v1beta1
kind: PodGroup
metadata:
annotations:
creationTimestamp: "2024-05-28T09:18:50Z"
generation: 5
labels:
  volcano.sh/job-type: MPI
name: lm-mpi-job-9c571015-37c7-4a1a-9604-eaa2248613f2
namespace: default
ownerReferences:
- apiVersion: batch.volcano.sh/v1alpha1
  blockOwnerDeletion: true
  controller: true
  kind: Job
  name: lm-mpi-job
  uid: 9c571015-37c7-4a1a-9604-eaa2248613f2
resourceVersion: "25173454"
uid: 7b04632e-7cff-4884-8e9a-035b7649d33b
spec:
minMember: 4
minResources:
  count/pods: "4"
  cpu: 3500m
  limits.cpu: 3500m

```

```

pods: "4"
requests.cpu: 3500m
minTaskMember:
  mpimaster: 1
  mpiworker: 3
queue: default
status:
  conditions:
    - lastTransitionTime: "2024-05-28T09:19:01Z"
      message: '3/4 tasks in gang unschedulable: pod group is not ready, 1 Succeeded,
        3 Releasing, 4 minAvailable'
      reason: NotEnoughResources
      status: "True"
      transitionID: f875efa5-0358-4363-9300-06cebc0e7466
      type: Unschedulable
    - lastTransitionTime: "2024-05-28T09:18:53Z"
      message: tasks in gang are ready to be scheduled
      reason: tasks in gang are ready to be scheduled
      status: "True"
      transitionID: 5a7708c8-7d42-4c33-9d97-0581f7c06dab
      type: Scheduled
  phase: Pending
  succeeded: 1

```

从 PodGroup 可以看出，通过 ownerReferences 关联到工作负载，并设置最小运行的 Pod 数为 4。

优先级抢占（Preemption scheduling）策略

Volcano 通过 Priority 插件实现了优先级抢占策略，即 Preemption scheduling 策略。在集群资源有限且多个 Job 等待调度时，如果使用 Kubernetes 默认调度器，可能会导致具有更多 Pod 数量的 Job 分得更多资源。而 Volcano-scheduler 提供了算法，支持不同的 Job 以 fair-share 的形式共享集群资源。

Priority 插件允许用户自定义 Job 和 Task 的优先级，并根据需求在不同层次上定制调度策略。例如，对于金融场景、物联网监控场景等需要较高实时性的应用，Priority 插件能够确保其优先得到调度。

使用方式

优先级的决定基于配置的 PriorityClass 中的 Value 值，值越大优先级越高。默认已启用，

无需修改。可通过以下命令确认或修改。

```
kubectl -n volcano-system edit configmaps volcano-scheduler-configmap
```

使用案例

假设集群中存在两个空闲节点，并有三个优先级不同的工作负载：high-priority、med-priority 和 low-priority。当 high-priority 工作负载运行并占满集群资源后，再提交 med-priority 和 low-priority 工作负载。由于集群资源全部被更高优先级的工作负载占用，med-priority 和 low-priority 工作负载将处于 pending 状态。当 high-priority 工作负载结束后，根据优先级调度原则，med-priority 工作负载将优先被调度。

1. 通过 priority.yaml 创建 3 个优先级定义，分别为：high-priority，med-priority，low-priority。

```
```yaml title="查看 priority.yaml" cat <<EOF | kubectl apply -f -
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
items:
 - metadata:
 name: high-priority
 value: 100
 globalDefault: false
 description: "This priority class should be used for volcano job only."
 -
 metadata:
 name: med-priority
 value: 50
 globalDefault: false
 description: "This priority class should be used for volcano job only."
 -
 metadata:
 name: low-priority
 value: 10
 globalDefault: false
 description: "This priority class should be used for volcano job only."
EOF````
```

2. 查看优先级定义信息。

NAME	VALUE	GLOBAL-DEFAULT	AGE
high-priority	100	false	97s
low-priority	10	false	97s
med-priority	50	false	97s
system-cluster-critical	2000000000	false	6d6h
system-node-critical	2000001000	false	6d6h

3. 创建高优先级工作负载 high-priority-job，占用集群的全部资源。

```
bash title="查看 high-priority-job" cat <<EOF | kubectl apply -f -
apiVersion:
batch.volcano.sh/v1alpha1 kind: Job metadata: name: priority-high
spec: schedulerName: volcano minAvailable: 4 priorityClassName:
high-priority tasks: - replicas: 4 name: "test"
template: spec: containers: - image:
alpine command: ["/bin/sh", "-c", "sleep 1000"]
imagePullPolicy: IfNotPresent name: running
resources: requests: cpu: "4"
restartPolicy: OnFailure EOF
```

通过 kubectl get pod 查看 Pod 运行 信息：

NAME	READY	STATUS	RESTARTS	AGE
priority-high-test-0	1/1	Running	0	3s
priority-high-test-1	1/1	Running	0	3s
priority-high-test-2	1/1	Running	0	3s
priority-high-test-3	1/1	Running	0	3s

此时，集群节点资源已全部被占用。

4. 创建中优先级工作负载 med-priority-job 和低优先级工作负载 low-priority-job。

```
bash title="med-priority-job" cat <<EOF | kubectl apply -f -
apiVersion:
batch.volcano.sh/v1alpha1 kind: Job metadata: name: priority-medium
spec: schedulerName: volcano minAvailable: 4 priorityClassName:
med-priority tasks: - replicas: 4 name: "test"
template: spec: containers: - image:
alpine command: ["/bin/sh", "-c", "sleep 1000"]
imagePullPolicy: IfNotPresent name: running
resources: requests: cpu: "4"
restartPolicy: OnFailure EOF
bash title="low-priority-job" cat <<EOF | kubectl apply -f -
apiVersion:
batch.volcano.sh/v1alpha1 kind: Job metadata: name: priority-low
spec: schedulerName: volcano minAvailable: 4 priorityClassName:
low-priority tasks: - replicas: 4 name: "test"
template: spec: containers: - image:
alpine command: ["/bin/sh", "-c", "sleep 1000"]
imagePullPolicy: IfNotPresent name: running
resources: requests: cpu: "4"
restartPolicy: OnFailure EOF
```

通过 kubectl get pod 查看 Pod 运行信息，集群资源不足，Pod 处于 Pending 状态：

kubectl get pods					
NAME	READY	STATUS	RESTARTS	AGE	
priority-high-test-0	1/1	Running	0	3m29s	
priority-high-test-1	1/1	Running	0	3m29s	
priority-high-test-2	1/1	Running	0	3m29s	
priority-high-test-3	1/1	Running	0	3m29s	
priority-low-test-0	0/1	Pending	0	2m26s	
priority-low-test-1	0/1	Pending	0	2m26s	
priority-low-test-2	0/1	Pending	0	2m26s	
priority-low-test-3	0/1	Pending	0	2m26s	
priority-medium-test-0	0/1	Pending	0	2m36s	
priority-medium-test-1	0/1	Pending	0	2m36s	
priority-medium-test-2	0/1	Pending	0	2m36s	
priority-medium-test-3	0/1	Pending	0	2m36s	

5. 删除 `high_priority_job` 工作负载，释放集群资源，`med_priority_job` 会被优先调度。

执行 `kubectl delete -f high_priority_job.yaml` 释放集群资源，查看 Pod 的调度信息：

kubectl get pods					
NAME	READY	STATUS	RESTARTS	AGE	
priority-low-test-0	0/1	Pending	0	5m18s	
priority-low-test-1	0/1	Pending	0	5m18s	
priority-low-test-2	0/1	Pending	0	5m18s	
priority-low-test-3	0/1	Pending	0	5m18s	
priority-medium-test-0	1/1	Running	0	5m28s	
priority-medium-test-1	1/1	Running	0	5m28s	
priority-medium-test-2	1/1	Running	0	5m28s	
priority-medium-test-3	1/1	Running	0	5m28s	

## 使用 Volcano Binpack 调度策略

Binpack 调度算法的目标是尽量把已被占用的节点填满（尽量不往空白节点分配）。具体实现上，Binpack 调度算法会给投递的节点打分，分数越高表示节点的资源利用率越高。通过尽可能填满节点，将应用负载靠拢在部分节点，这种调度算法能够尽可能减小节点内的碎片，在空闲的机器上为申请了更大资源请求的 Pod 预留足够的资源空间，使集群下空闲资源得到最大化的利用。

## 前置条件

预先在 DCE 5.0 上[安装 Volcano 组件](#)。

## Binpack 算法原理

Binpack 在对一个节点打分时，会根据 Binpack 插件自身权重和各资源设置的权重值综合打分。首先，对 Pod 请求资源中的每类资源依次打分，以 CPU 为例，CPU 资源在待调度节点的得分信息如下：

$\text{CPU.weight} * (\text{request} + \text{used}) / \text{allocatable}$

即 CPU 权重值越高，得分越高，节点资源使用量越满，得分越高。Memory、GPU 等资源原理类似。其中：

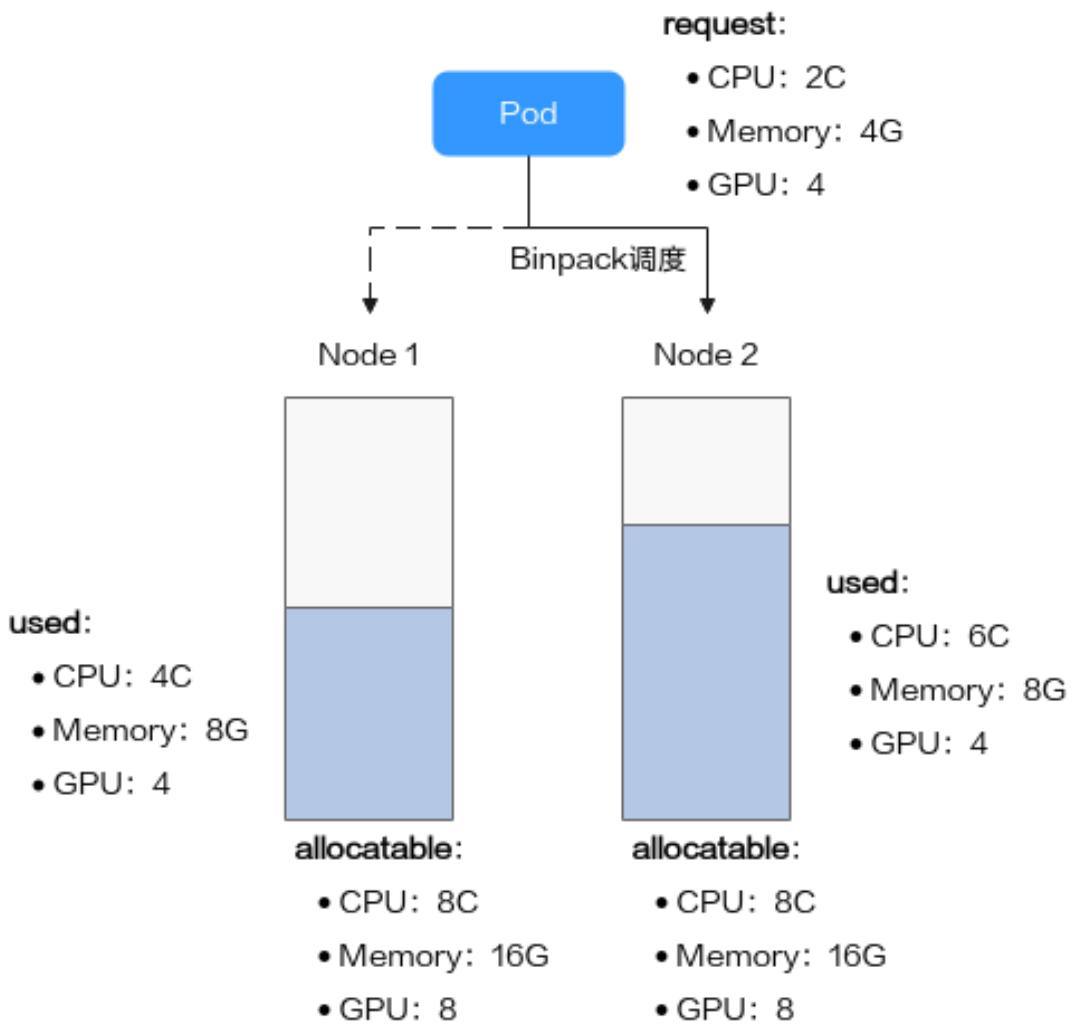
- CPU.weight 为用户设置的 CPU 权重
- request 为当前 Pod 请求的 CPU 资源量
- used 为当前节点已经分配使用的 CPU 量
- allocatable 为当前节点 CPU 可用总量

通过 Binpack 策略的节点总得分如下：

$\text{binpack.weight} - (\text{CPU.score} + \text{Memory.score} + \text{GPU.score}) / (\text{CPU.weight} + \text{Memory.weight} + \text{GPU.weight}) - 100$

即 Binpack 插件的权重值越大，得分越高，某类资源的权重越大，该资源在打分时的占比越大。其中：

- binpack.weight 为用户设置的装箱调度策略权重
- CPU.score 为 CPU 资源得分，CPU.weight 为 CPU 权重
- Memory.score 为 Memory 资源得分，Memory.weight 为 Memory 权重
- GPU.score 为 GPU 资源得分，GPU.weight 为 GPU 权重



## 原理

如图所示，集群中存在两个节点，分别为 Node1 和 Node 2，在调度 Pod 时，Binpack 策略对两个节点分别打分。假设集群中 CPU.weight 配置为 1，Memory.weight 配置为 1，GPU.weight 配置为 2，binpack.weight 配置为 5。

1. Binpack 对 Node 1 的资源打分，各资源的计算公式为：

– CPU Score :

$$\text{CPU.weight} - (\text{request} + \text{used}) / \text{allocatable} = 1 - (2 + 4) / 8 = 0.75$$

– Memory Score :

$$\text{Memory.weight} - (\text{request} + \text{used}) / \text{allocatable} = 1 - (4 + 8) / 16 = 0.75$$

– GPU Score :

$$\text{GPU.weight} - (\text{request} + \text{used}) / \text{allocatable} = 2 - (4 + 4) / 8 = 2$$

2. 节点总得分的计算公式为：

$$\text{binpack.weight} - (\text{CPU.score} + \text{Memory.score} + \text{GPU.score}) / (\text{CPU.weight} + \text{Memory.weight} + \text{GPU.weight}) - 100$$

假设 `binpack.weight` 配置为 5，Node 1 在 Binpack 策略下的得分为：

$$5 - (0.75 + 0.75 + 2) / (1 + 1 + 2) - 100 = 437.5$$

3. Binpack 对 Node 2 的资源打分：

- CPU Score :

$$\text{CPU.weight} - (\text{request} + \text{used}) / \text{allocatable} = 1 - (2 + 6) / 8 = 1$$

- Memory Score :

$$\text{Memory.weight} - (\text{request} + \text{used}) / \text{allocatable} = 1 - (4 + 8) / 16 = 0.75$$

- GPU Score :

$$\text{GPU.weight} - (\text{request} + \text{used}) / \text{allocatable} = 2 - (4 + 4) / 8 = 2$$

4. Node 2 在 Binpack 策略下的得分为：

$$5 - (1 + 0.75 + 2) / (1 + 1 + 2) - 100 = 468.75$$

综上，Node 2 得分大于 Node 1，按照 Binpack 策略，Pod 将会优先调度至 Node 2。

## 使用案例

Binpack 调度插件在安装 Volcano 的时候默认就会开启；如果用户没有配置权重，则使用

如下默认的配置权重。

```

- plugins:
 - name: binpack
 arguments:
 binpack.weight: 1
 binpack.cpu: 1
 binpack.memory: 1

```

默认权重不能体现堆叠特性，因此需要修改为 `binpack.weight: 10`。

```
kubectl -n volcano-system edit configmaps volcano-scheduler-configmap
```

```

- plugins:
 - name: binpack
 arguments:
 binpack.weight: 10

```

```

binpack.cpu: 1
binpack.memory: 1
binpack.resources: nvidia.com/gpu, example.com/foo
binpack.resources.nvidia.com/gpu: 2
binpack.resources.example.com/foo: 3

```

修改好之后重启 volcano-scheduler Pod 使其生效。

创建如下的 Deployment。

```

apiVersion: apps/v1
kind: Deployment
metadata:
 name: binpack-test
 labels:
 app: binpack-test
spec:
 replicas: 2
 selector:
 matchLabels:
 app: test
 template:
 metadata:
 labels:
 app: test
 spec:
 schedulerName: volcano
 containers:
 - name: test
 image: busybox
 imagePullPolicy: IfNotPresent
 command: ["sh", "-c", 'echo "Hello, Kubernetes!" && sleep 3600']
 resources:
 requests:
 cpu: 500m
 limits:
 cpu: 500m

```

在两个 Node 的集群上可以看到 Pod 被调度到一个 Node 上。

容器组	容器配置	访问方式	调度策略	标签与注解	弹性伸缩	版本记录	事件列表
<input type="text"/> 输入容器组名称搜索							
容器组名称	状态	容器 (正常/总...)	容器组 IP	节点	重启次数	CPU 请求...	内存请求...
binpack-test-7d...	运行中	1/1	10.233.74.1...	controller-node-1	0	0.5 Core / 0.5 ...	不限制 / 不限制
binpack-test-7d...	运行中	1/1	10.233.74.95	controller-node-1	0	0.5 Core / 0.5 ...	不限制 / 不限制

结果

# DRF ( Dominant Resource Fairness ) 调度策略

DRF 调度策略认为占用资源较少的任务具有更高的优先级。这样能够满足更多的作业，不会因为一个胖业务，饿死大批小业务。DRF 调度算法能够确保在多种类型资源共存的环境下，尽可能满足分配的公平原则。

## 使用方式

DRF 调度策略默认已启用，无需任何配置。

```
kubectl -n volcano-system view configmaps volcano-scheduler-configmap
```

## 使用案例

在 AI 训练，或大数据计算中，通过有限运行使用资源少的任务，这样可以让集群资源使用率更高，而且还能避免小任务被饿死。如下创建两个 Job，一个是小资源需求，一个是大资源需求，可以看出来小资源需求的 Job 优先运行起来。

```
cat <<EOF | kubectl apply -f -
apiVersion: batch.volcano.sh/v1alpha1
kind: Job
metadata:
 name: small-resource
spec:
 schedulerName: volcano
 minAvailable: 4
 priorityClassName: small-resource
 tasks:
 - replicas: 4
 name: "test"
```

```
template:
 spec:
 containers:
 - image: alpine
 command: ["/bin/sh", "-c", "sleep 1000"]
 imagePullPolicy: IfNotPresent
 name: running
 resources:
 requests:
 cpu: "1"
 restartPolicy: OnFailure

apiVersion: batch.volcano.sh/v1alpha1
kind: Job
metadata:
 name: large-resource
spec:
 schedulerName: volcano
 minAvailable: 4
 priorityClassName: large-resource
 tasks:
 - replicas: 4
 name: "test"
 template:
 spec:
 containers:
 - image: alpine
 command: ["/bin/sh", "-c", "sleep 1000"]
 imagePullPolicy: IfNotPresent
 name: running
 resources:
 requests:
 cpu: "2"
 restartPolicy: OnFailure
EOF
```

## NUMA 亲和性调度

NUMA 节点是 Non-Uniform Memory Access ( 非统一内存访问 ) 架构中的一个基本组成单元 , 一个 Node 节点是多个 NUMA 节点的集合 , 在多个 NUMA 节点之间进行内存访问

时会产生延迟，开发者可以通过优化任务调度和内存分配策略，来提高内存访问效率和整体性能。

## 使用场景

Numa 亲和性调度的常见场景是那些对 CPU 参数敏感/调度延迟敏感的计算密集型作业。如科学计算、视频解码、动漫动画渲染、大数据离线处理等具体场景。

## 调度策略

Pod 调度时可以采用的 NUMA 放置策略，具体策略对应的调度行为请参见 Pod 调度行为说明。

- `single-numa-node`：Pod 调度时会选择拓扑管理策略已经设置为 `single-numa-node` 的节点池中的节点，且 CPU 需要放置在相同 NUMA 下，如果节点池中没有满足条件的节点，Pod 将无法被调度。
- `restricted`：Pod 调度时会选择拓扑管理策略已经设置为 `restricted` 节点池的节点，且 CPU 需要放置在相同的 NUMA 集合下，如果节点池中没有满足条件的节点，Pod 将无法被调度。
- `best-effort`：Pod 调度时会选择拓扑管理策略已经设置为 `best-effort` 节点池的节点，且尽量将 CPU 放置在相同 NUMA 下，如果没有节点满足这一条件，则选择最优节点进行放置。

## 调度原理

当 Pod 设置了拓扑策略时，Volcano 会根据 Pod 设置的拓扑策略预测匹配的节点列表。调度过程如下：

1. 根据 Pod 设置的 Volcano 拓扑策略，筛选具有相同策略的节点。
2. 在设置了相同策略的节点中，筛选 CPU 拓扑满足该策略要求的节点进行调度。

Pod 可配置的 1. 根据 Pod 设置的拓扑策略，筛 2. 进一步筛选 CPU 拓扑满足策略

拓扑策略 选可调度的节点 的节点进行调度

none 针对配置了以下几种拓扑策略的节

点，调度时均无筛选行为。none：

可调度；best-effort：可调度；

restricted：可调度；

single-numa-node：可调度

best-effort 筛选拓扑策略同样为“best-effort”的 尽可能满足策略要求进行调度：优

节点：none：不可调度； 先调度至单 NUMA 节点，如果单

best-effort：可调度；restricted：不 NUMA 节点无法满足 CPU 申请

可调度；single-numa-node：不可调 值，允许调度至多个 NUMA 节点。

度

restricted 筛选拓扑策略同样为“restricted”的 严格限制的调度策略：单 NUMA 节

节点：none：不可调度； 点的 CPU 容量上限大于等于 CPU 的

best-effort：不可调度；restricted： 申请值时，仅允许调度至单 NUMA

可调度；single-numa-node：不可调 节点。此时如果单 NUMA 节点剩余

度 的 CPU 可使用量不足，则 Pod 无

法调度。单 NUMA 节点的 CPU 容

量上限小于 CPU 的申请值时，可允

许调度至多个 NUMA 节点。

Pod 可配置的 1. 根据 Pod 设置的拓扑策略，筛选进一步筛选 CPU 拓扑满足策略

拓扑策略 选可调度的节点 的节点进行调度

single-numa-node 筛选拓扑策略同样为 仅允许调度至单 NUMA 节点。

“single-numa-node”的节点：none：

不可调度；best-effort：不可调度；

restricted：不可调度；

single-numa-node：可调度

## 配置 NUMA 亲和调度策略

1. 在 Job 中配置 policies

task:

```
- replicas: 1
 name: "test-1"
 topologyPolicy: single-numa-node
- replicas: 1
 name: "test-2"
 topologyPolicy: best-effort
```

2. 修改 kubelet 的调度策略，设置 --topology-manager-policy 参数，支持的策略有四

种：

- none (默认)
- best-effort
- restricted
- single-numa-node

## 使用案例

1. 示例一：在无状态工作负载中配置 NUMA 亲和性。

```
kind: Deployment
apiVersion: apps/v1
metadata:
```

```

name: numa-tset
spec:
 replicas: 1
 selector:
 matchLabels:
 app: numa-tset
 template:
 metadata:
 labels:
 app: numa-tset
 annotations:
 volcano.sh/numa-topology-policy: single-numa-node # set the topology policy
y
spec:
 containers:
 - name: container-1
 image: nginx:alpine
 resources:
 requests:
 cpu: 2 # 必须为整数, 且需要与limits中一致
 memory: 2048Mi
 limits:
 cpu: 2 # 必须为整数, 且需要与requests中一致
 memory: 2048Mi
 imagePullSecrets:
 - name: default-secret

```

## 2.示例二：创建一个 Volcano Job，并使用 NUMA 亲和性。

```

apiVersion: batch.volcano.sh/v1alpha1
kind: Job
metadata:
 name: vj-test
spec:
 schedulerName: volcano
 minAvailable: 1
 tasks:
 - replicas: 1
 name: "test"
 topologyPolicy: best-effort # set the topology policy for task
 template:
 spec:
 containers:
 - image: alpine
 command: [/bin/sh, "-c", "sleep 1000"]

```

```

imagePullPolicy: IfNotPresent
name: running
resources:
 limits:
 cpu: 20
 memory: "100Mi"
restartPolicy: OnFailure

```

## NUMA 调度分析

假设 NUMA 节点情况如下：

工作节点	节点策略拓扑管理器	NUMA 节点 0 上的可分	NUMA 节点 1 上的可分
点	策略	配 CPU	配 CPU

node-1	single-numa-node	16U	16U
node-2	best-effort	16U	16U
node-3	best-effort	20U	20U

- [示例一](#) 中，Pod 的 CPU 申请值为 2U，设置拓扑策略为“single-numa-node”，因此会被调度到相同策略的 node-1。
- [示例二](#) 中，Pod 的 CPU 申请值为 20U，设置拓扑策略为“best-effort”，它将被调度到 node-3，因为 node-3 可以在单个 NUMA 节点上分配 Pod 的 CPU 请求，而 node-2 需要在两个 NUMA 节点上执行此操作。

## 查看当前节点的 CPU 概况

您可以通过 lscpu 命令查看当前节点的 CPU 概况：

```

lscpu
...
CPU(s): 32
NUMA node(s): 2
NUMA node0 CPU(s): 0-15
NUMA node1 CPU(s): 16-31

```

## 查看当前节点的 CPU 分配

然后查看 NUMA 节点使用情况：

```
查看当前节点的 CPU 分配
cat /var/lib/kubelet/cpu_manager_state
{"policyName": "static","defaultCpuSet":"0,10-15,25-31","entries":{ "777870b5-c64f-42f5-9296-688b9dc212ba":{"container-1":"16-24"}, "fb15e10a-b6a5-4aaa-8fcf-76c1aa64e6fd":{"container-1":"1-9"}}, "checksum":318470969}
```

以上示例中表示，节点上运行了两个容器，一个占用了 NUMA node0 的 1-9 核，另一个占用了 NUMA node1 的 16-24 核。

## GPU 调度配置（Binpack 和 Spread ）

本文介绍使用 NVIDIA vGPU 时，如何通过 Binpack 和 Spread 的 GPU 调度配置减少 GPU 资源碎片、防止单点故障等，实现 vGPU 的高级调度。DCE 5.0 平台提供了集群和工作负载两种维度的 Binpack 和 Spread 调度策略，分别满足不同场景下的使用需求。

### 前置条件

- 集群节点上已正确安装 GPU 设备。
- 集群中已正确安装 [gpu-operator 组件](#) 和 [Nvidia-vgpu 组件](#)。
- 集群节点列表中，GPU 模式下存在 NVIDIA-vGPU 类型。

### 使用场景

- 基于 GPU 卡维度调度策略
  - Binpack：优先选择节点的同一张 GPU 卡，适用于提高 GPU 利用率，减少资源碎片。

- Spread：多个 Pod 会分散在节点的不同 GPU 卡上，适用于高可用场景，避免单卡故障。
- 基于节点维度的调度策略
  - Binpack：多个 Pod 会优先选择同一个节点，适用于提高 GPU 利用率，减少资源碎片。
  - Spread：多个 Pod 会分散在不同节点上，适用于高可用场景，避免单节点故障。

## 集群维度使用 Binpack 和 Spread 调度配置

### !!! note

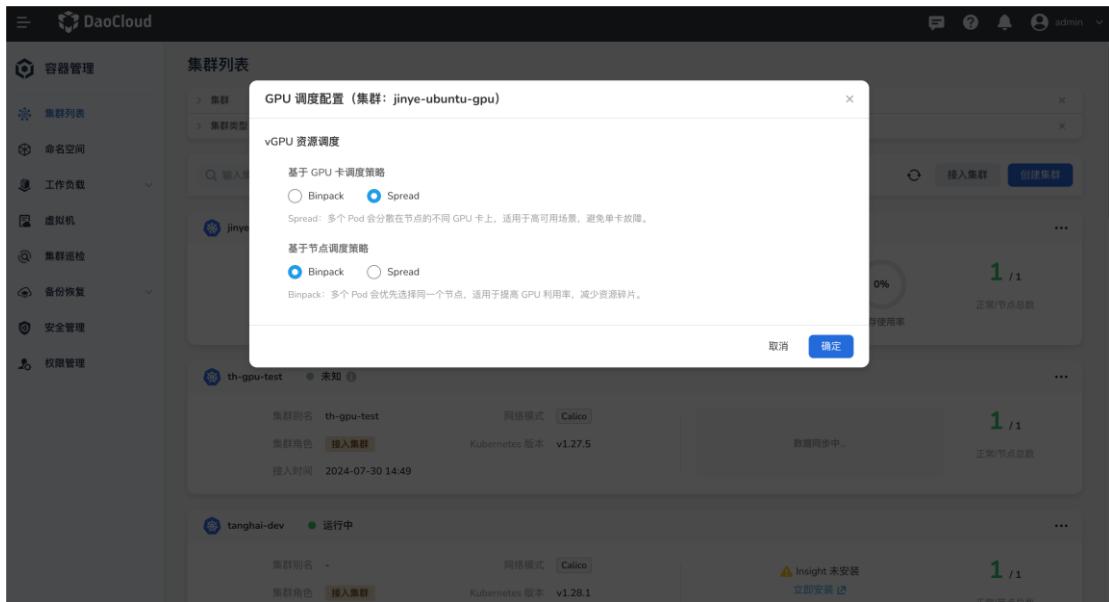
默认情况下，工作负载会遵循集群级别的 Binpack 和 Spread 调度配置。

若工作负载单独设置了与集群不一致的 Binpack 和 Spread 调度策略，则该工作负载优先遵循其本身的调度策略。

1. 在 **集群列表** 页选择需要调整 Binpack 和 Spread 调度策略的集群，点击右侧的操作图标并在下拉列表中点击 **GPU 调度配置**。

### 集群列表

2. 根据业务场景调整 GPU 调度配置，并点击 **确定** 后保存。



### binpack 配置

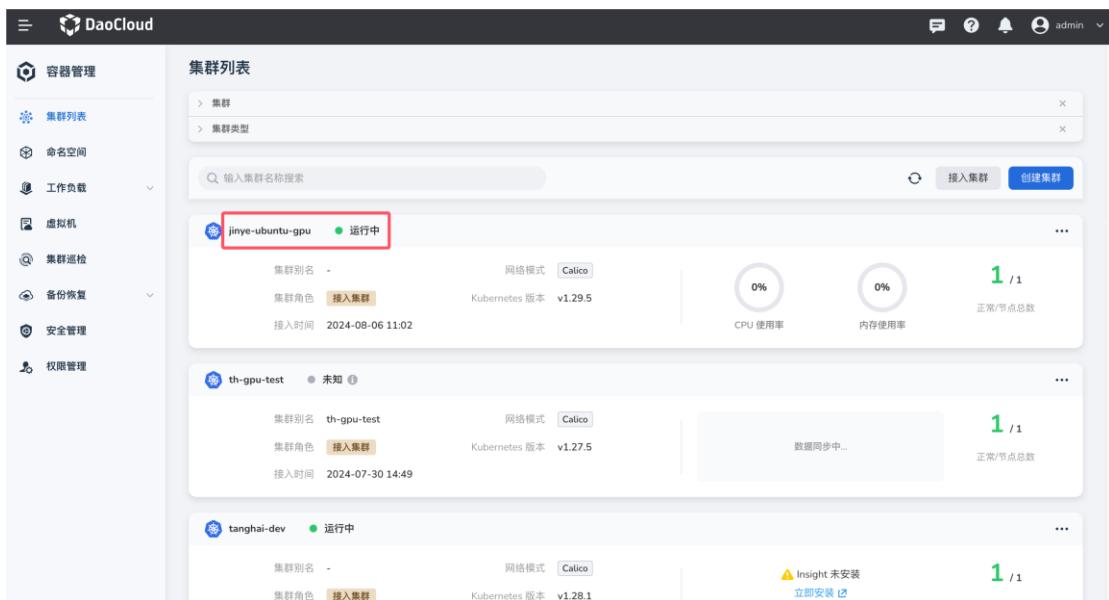
## 工作负载维度使用 Binpack 和 Spread 调度配置

#### !!! note

当工作负载维度的 Binpack 和 Spread 调度策略与集群级别的配置冲突时，优先遵循工作负载维度的配置。

参考以下步骤，使用镜像创建一个无状态负载，并在工作负载中配置 Binpack 和 Spread 调度策略。

1. 点击左侧导航栏上的 **集群列表**，然后点击目标集群的名称，进入 **集群详情** 页面。



## 集群 list

2. 在集群详情页面，点击左侧导航栏的 **工作负载 -> 无状态负载**，然后点击页面右上角的 **镜像创建** 按钮。

## 创建工作负载

3. 依次填写 基本信息、容器配置，并在 容器配置 中启用 GPU 配置，选择 GPU 类型为 NVIDIA vGPU，点击 **高级设置**，启用 Binpack / Spread 调度策略，根据业务场景调整 GPU 调度配置。配置完成后点击 **下一步**，进入 服务配置、高级配置后，在页面右下角点击 **确定** 完成创建。

配置 binpack

# 昇腾 NPU 组件安装

本章节提供昇腾 NPU 驱动、Device Plugin、NPU-Exporter 等组件的安装指导。

## 前提条件

1. 安装前请确认支持的 NPU 型号，详情请参考[昇腾 NPU 矩阵](#)
2. 请确认 对应 NPU 型号所要求的内核版本是否匹配，详情请参考[昇腾 NPU 矩阵](#)
3. 准备 Kubernetes 基础环境

## 安装步骤

使用 NPU 资源之前，需要完成固件安装、NPU 驱动安装、Docker Runtime 安装、用户创建、日志目录创建以及 NPU Device Plugin 安装，详情参考如下步骤。

## 安装固件

1. 安装前请确认内核版本在“二进制安装”安装方式对应的版本范围内，则可以直接安装 NPU 驱动固件。
2. 固件与驱动下载请参考[固件下载地址](#)
3. 固件安装请参考[安装 NPU 驱动固件](#)

## 安装 NPU 驱动

1. 如驱动未安装，请参考昇腾官方文档进行安装。例如 Ascend910，参考 [910 驱动安装](#)

[文档](#)。

2. 运行 `npu-smi info` 命令，并且能够正常返回 NPU 信息，表示 NPU 驱动与固件已就绪。

昇腾信息

昇腾信息

## 安装 Docker Runtime

1. 下载 Ascend Docker Runtime

社区版下载地址：<https://www.hiascend.com/zh/software/mindx-dl/community>

```
wget -c https://mindx.obs.cn-south-1.myhuaweicloud.com/OpenSource/MindX/MindX%205.0.RC2/MindX%20DL%205.0.RC2/Ascend-docker-runtime_5.0.RC2_linux-x86_64.run
```

安装到指定路径下，依次执行以下两条命令，参数为指定的安装路径：

```
chmod u+x Ascend-docker-runtime_5.0.RC2_linux-x86_64.run
./Ascend-docker-runtime_{version}_linux-{arch}.run --install --install-path=<path>
```

2. 修改 containerd 配置文件

containerd 无默认配置文件时，依次执行以下 3 条命令，创建配置文件：

```
mkdir /etc/containerd
containerd config default > /etc/containerd/config.toml
vim /etc/containerd/config.toml
```

containerd 有配置文件时：

```
vim /etc/containerd/config.toml
```

根据实际情况修改 runtime 的安装路径，主要修改 runtime 字段：

```
...
[plugins."io.containerd.monitor.v1.cgroups"]
no_prometheus = false
[plugins."io.containerd.runtime.v1.linux"]
shim = "containerd-shim"
runtime = "/usr/local/Ascend/Ascend-Docker-Runtime/ascend-docker-runtime"
runtime_root = ""
no_shim = false
shim_debug = false
```

```
[plugins."io.containerd.runtime.v2.task"]
platforms = ["linux/amd64"]
...
```

执行以下命令，重启 containerd：

```
systemctl restart containerd
```

## 用户创建

在对应组件安装的节点上执行以下命令创建用户。

```
Ubuntu 操作系统
useradd -d /home/hwMindX -u 9000 -m -s /usr/sbin/nologin hwMindX
usermod -a -G HwHiAiUser hwMindX
Centos 操作系统
useradd -d /home/hwMindX -u 9000 -m -s /sbin/nologin hwMindX
usermod -a -G HwHiAiUser hwMindX
```

## 日志目录创建

在对应节点创建组件日志父目录和各组件的日志目录，并设置目录对应属主和权限。执行下述命令，创建组件日志父目录。

```
mkdir -m 755 /var/log/mindx-dl
chown root: root /var/log/mindx-dl
```

执行下述命令，创建 Device Plugin 组件日志目录。

```
mkdir -m 750 /var/log/mindx-dl/devicePlugin
chown root: root /var/log/mindx-dl/devicePlugin
!!! note
```

请分别为所需组件创建对应的日志目录，当前案例中只需要 Device Plugin 组件。

如果有其他组件需求请参考[官方文档]([https://www.hiascend.com/document/detail/zh/mindx-dl/50rc3/clusterscheduling/clusterschedulingig/dlug\\_installation\\_016.html](https://www.hiascend.com/document/detail/zh/mindx-dl/50rc3/clusterscheduling/clusterschedulingig/dlug_installation_016.html))

## 创建节点 Label

参考下述命令在对应节点上创建 Label：

```
在安装了驱动的计算节点创建此标签
kubectl label node {nodename} huawei.com.ascend/Driver=installed
kubectl label node {nodename} node-role.kubernetes.io/worker=worker
```

```
kubectl label node {nodename} workerselector=dls-worker-node
kubectl label node {nodename} host-arch=huawei-arm //或者 host-arch=huawei-x86 , 根据实际情况选择
kubectl label node {nodename} accelerator=huawei-Ascend910 //根据实际情况进行选择
在控制节点创建此标签
kubectl label node {nodename} masterselector=dls-master-node
```

## 安装 Device Plugin 和 NpuExporter

功能模块路径： 容器管理 -> 集群管理 ，点击目标集群的名称，从左侧导航栏点击 Helm

应用 -> Helm 模板 -> 搜索 ascend-mindxdl 。

找到 ascend-mindxdl

ascend-mindxdl 详情

- **DevicePlugin**：通过提供通用设备插件机制和标准的设备 API 接口，供 Kubernetes 使用设备。建议使用默认的镜像及版本。
- **NpuExporter**：基于 Prometheus/Telegraf 生态，该组件提供接口，帮助用户能够关注到昇腾系列 AI 处理器以及容器级分配状态。建议使用默认的镜像及版本。
- **ServiceMonitor**：默认不开启，开启后可前往可观测性模块查看 NPU 相关监控。如需开启，请确保 insight-agent 已安装并处于运行状态，否则将导致 ascend-mindxdl 安装失败。
- **isVirtualMachine**：默认不开启，如果 NPU 节点为虚拟机场景，请开启 isVirtualMachine 参数。

安装成功后，对应命名空间下会出现两个组件，如下图：

容器组名称	状态	容器 (正常/总量)	命名空间	容器组 IP	节点	重启次数	CPU 请求值/限制值	内存请求值/限制值	GPU 数量	创建时间
ascend-device-plugin-daemonset-v7qzw	运行中	1/1	ascend-mindxdl	172.20.48.70	192.168.0.207	0	0.5 Core / 0.5 Core	500 Mi / 500 Mi	-	2024-04-18 10:45
npu-exporter-6m4rw	运行中	1/1	ascend-mindxdl	172.20.48.77	192.168.0.207	0	不限制 / 不限制	不限制 / 不限制	-	2024-04-18 10:45
helm-operation-uninstall-ascend-mindx...	已完成	0/1	ascend-mindxdl	172.20.48.85	192.168.0.207	0	0.1 Core / 0.1 Core	400 Mi / 400 Mi	-	2024-04-18 10:42
helm-operation-upgrade-ascend-mindx...	已完成	0/1	ascend-mindxdl	-	192.168.0.207	0	0.1 Core / 0.1 Core	400 Mi / 400 Mi	-	2024-04-17 18:41

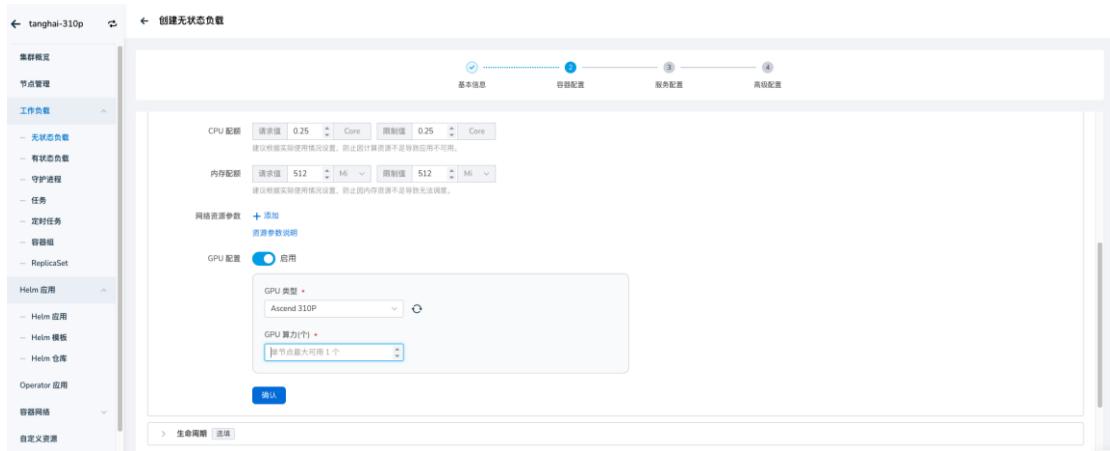
ascend-mindxdl 列表

同时节点信息上也会出现对应 NPU 的信息：

名称	状态	角色	标签	CPU (分配/使用率)	内存 (分配/使用率)	IP 地址	创建时间
192.168.0.207 [Ascend-310P]	健康 (正常)	工作节点	beta.kubernetes.io/os:linux +13	0% / 0%	0% / 0%	192.168.0.207	2024-04-16 16:23

## 节点标签

一切就绪后，我们通过页面创建工作负载时，就能够选择到对应的 NPU 设备，如下图：



## 使用 NPU

### !!! note

有关详细使用步骤，请参照[应用使用昇腾（Ascend）NPU](./ascend\_usage.md)。

# 应用使用昇腾（Ascend）NPU

本节介绍如何在 DCE 5.0 平台使用昇腾 GPU。

## 前提条件

- 当前 NPU 节点已安装昇腾（Ascend）驱动。
- 当前 NPU 节点已安装 Ascend-Docker-Runtime 组件。
- 当前集群已安装 NPU MindX DL 套件。
- 当前集群内 NPU 卡未进行任何虚拟化操作或被其它应用占用。

请参考[昇腾 NPU 组件安装文档](#)安装基础环境。

# 快速使用

本文使用昇腾示例库中的 [AscentCL 图片分类应用](#)示例。

## 1. 下载昇腾代码库

运行以下命令下载昇腾 Demo 示例代码库，并且请记住代码存放的位置，后续需要使用。

```
git clone https://gitee.com/ascend/samples.git
```

## 2. 准备基础镜像

此例使用 Ascent-pytorch 基础镜像，可访问[昇腾镜像仓库](#)获取。

## 3. 准备 YAML

```
yaml title="ascend-demo.yaml" apiVersion: batch/v1 kind: Job metadata: name: resnetinfer1-1-1usoc spec: template: spec: containers: - image: ascendhub.huawei.com/public-ascendhub/ascend-pytorch:23.0.RC2-ubuntu18.04 # Inference image name imagePullPolicy: IfNotPresent name: resnet50infer securityContext: runAsUser: 0 command: - "/bin/bash" - "-c" - | source /usr/local/Ascend/ascend-toolkit/set_env.sh && TEMP_DIR=/root/samples_copy_$(date '+%Y%m%d_%H%M%S_%N') && cp -r /root/samples "$TEMP_DIR" && cd "$TEMP_DIR"/inference/modelInference/sampleResnetQuickStart/python/model && wget https://obs-9be7.obs.cn-east-2.myhuaweicloud.com/003_Atc_Models/resnet50/resnet50.onnx && atc --model=resnet50.onnx --framework=5 --output=resnet50 --input_shape="actual_input_1:1,3,224,224" --soc_version=Ascend910 && cd/data && wget https://obs-9be7.obs.cn-east-2.myhuaweicloud.com/models/aclsample/dog1_1024_683.jpg && cd/scripts && bash sample_run.sh resources: requests: huawei.com/Ascend910: 1 # Number of the Ascend 910 Processors limits: huawei.com/Ascend910: 1 # The value should be the same as that of requests volumeMounts: - name: hiai-driver mountPath: /usr/local/Ascend/driver readOnly: true - name: slog mountPath: /var/log/npu/conf/slog/slog.conf - name: localtime # The container time must be the same as the host time mountPath: /etc/localtime - name: dmp mountPath:
```

```

/var/dmp_daemon - name: slogd mountPath:
/var/slogd - name: hbasic mountPath:
/etc/hdcBasic.cfg - name: sys-version mountPath:
/etc/sys_version.conf - name: aicpu mountPath:
/usr/lib64/aicpu_kernels - name: tfso mountPath:
/usr/lib64/libtensorflow.so - name: sample-path
mountPath: /root/samples volumes: - name: hiai-driver
hostPath: path: /usr/local/Ascend/driver - name: slog
hostPath: path: /var/log/npu/conf/slog/slog.conf - name:
localtime hostPath: path: /etc/localtime -
name: dmp hostPath: path: /var/dmp_daemon
- name: slogd hostPath: path: /var/slogd -
name: hbasic hostPath: path: /etc/hdcBasic.cfg
- name: sys-version hostPath: path: /etc/sys_version.conf
- name: aicpu hostPath: path: /usr/lib64/aicpu_kernels
- name: tfso hostPath: path: /usr/lib64/libtensorflow.so
- name: sample-path hostPath: path: /root/samples
restartPolicy: OnFailure

```

以上 YAML 中有一些字段需要根据实际情况进行修改：

1. **atc ... --soc\_version=Ascend910** 使用的是 **Ascend910**，请以实际情况为主 您  
可以使用 **npu-smi info** 命令查看显卡型号然后加上 Ascend 前缀即可
2. **samples-path** 以实际情况为准
3. **resources** 以实际情况为准
4. 部署 Job 并查看结果

使用如下命令创建 Job：

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

查看 Pod 运行状态：

昇腾 Pod 状态

昇腾 Pod 状态

Pod 成功运行后，查看日志结果。在屏幕上的关键提示信息示例如下图，提示信息中

的 Label 表示类别标识，Conf 表示该分类的最大置信度，Class 表示所属类别。

这些值可能会根据版本、环境有所不同，请以实际情况为准：

## 昇腾 demo 运行结果

昇腾 demo 运行结果

结果图片展示：

昇腾 demo 运行结果图片

昇腾 demo 运行结果图片

## 界面使用

1. 确认集群是否已检测 GPU 卡。点击对应 **集群 -> 集群设置 -> Addon 插件**，查看是否已自动启用并自动检测对应 GPU 类型。目前集群会自动启用 **GPU**，并且设置 **GPU** 类型为 **Ascend**。

集群设置

集群设置

2. 部署工作负载，点击对应 **集群 -> 工作负载**，通过镜像方式部署工作负载，选择类型（Ascend）之后，需要配置应用使用的物理卡数量：

**物理卡数量 ( huawei.com/Ascend910 )**：表示当前 Pod 需要挂载几张物理卡，输入值必须为整数且**小于等于宿主机上的卡数量**。

负载使用

负载使用

如果上述值配置的有问题则会出现调度失败，资源分配不了的情况。

## 启用昇腾虚拟化

昇腾虚拟化分为动态虚拟化和静态虚拟化，本文介绍如何开启并使用昇腾静态虚拟化能

力。

## 前提条件

- Kubernetes 集群环境搭建。
- 当前 NPU 节点已安装昇腾（Ascend）驱动。
- 当前 NPU 节点已安装 Ascend-Docker-Runtime 组件。
- 当前集群已安装 NPU MindX DL 套件。
- 支持的 NPU 卡型号：
  - Ascend 310P，已验证
  - Ascend 910b（20 核），已验证
  - Ascend 910（32 核），官方介绍支持，未实际验证
  - Ascend 910（30 核），官方介绍支持，未实际验证

更多细节参阅[官方虚拟化硬件说明](#)。

请参考[昇腾 NPU 组件安装文档](#)安装基础环境。

## 开启虚拟化能力

开启虚拟化能力需要手动修改 ascend-device-plugin-daemonset 组件的启动参数，参考下述

命令：

```
- device-plugin -useAscendDocker=true -volcanoType=false -presetVirtualDevice=true
- logFile=/var/log/mindx-dl/devicePlugin/devicePlugin.log -logLevel=0
```

## 切分 VNPU 实例

静态虚拟化需要手动对 VNPU 实例的切分，请参考下述命令：

```
npu-smi set -t create-vnpu -i 13 -c 0 -f vir02
```

- i 指的是 card id
- c 指的是 chip id
- vir02 指的是切分规格模板

关于 card id 和 chip id , 可以通过 npu-smi info 查询 , 切分规格可通过 [ascend 官方模板](#)

进行查询。

切分实例过后可通过下述命令查询切分结果 :

```
npu-smi info -t info-vnpu -i 13 -c 0
```

查询结果如下 :

```
[root@ecs-ec2e ~]# npu-smi set -t create-vnpu -i 13 -c 0 -f vir02
 Status : OK
 Message : Create vnpu success
[root@ecs-ec2e ~]# npu-smi set -t create-vnpu -i 13 -c 0 -f vir02
 Status : OK
 Message : Create vnpu success
[root@ecs-ec2e ~]# npu-smi set -t create-vnpu -i 13 -c 0 -f vir02
 Status : OK
 Message : Create vnpu success
[root@ecs-ec2e ~]# npu-smi info -t info-vnpu -i 13 -c 0
+-----+
| NPU resource static info as follow: |
| Format:Free/Total NA: Currently, query is not supported. |
| AICORE Memory AICPU VPC VENC VDEC JPEGD JPEGE PNGD |
| GB | |
+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
| 2/8 5/21 1/7 3/12 0/3 3/12 4/16 2/8 NA/NA |
+-----+
| Total number of vnpu: 3 |
+-----+
| Vnpu ID | Vgroup ID | Container ID | Status | Template Name |
+-----+
| 100 | 0 | 000000000000 | 0 | vir02 |
+-----+
| 101 | 1 | 000000000000 | 0 | vir02 |
+-----+
| 102 | 2 | 000000000000 | 0 | vir02 |
+-----+
```

vnpu1

## 重启 ascend-device-plugin-daemonset

切分实例后手动重启 device-plugin pod , 然后使用 kubectl describe 命令查看已注册 node 的资源 :

```
kubectl describe node {{nodename}}
```

```
Hostname: 192.168.0.129
Capacity:
cpu: 16
ephemeral-storage: 206291924Ki
huawei.com/Ascend310P: 1
huawei.com/Ascend310P-2c: 3
huawei.com/npu-core: 8
hugepages-1Gi: 0
hugepages-2Mi: 0
memory: 65806088Ki
pods: 110
Allocatable:
cpu: 16
ephemeral-storage: 190118636844
huawei.com/Ascend310P: 0
huawei.com/Ascend310P-2c: 3
huawei.com/npu-core: 0
hugepages-1Gi: 0
hugepages-2Mi: 0
memory: 65498888Ki
pods: 110
System Info:
vnpu2
```

## 如何使用设备

在创建应用时，指定资源 key，参考下述 YAML：

```
.....
resources:
 requests:
 huawei.com/Ascend310P-2c: 1
 limits:
 huawei.com/Ascend310P-2c: 1
.....
```

## App 使用天数智芯（Iluvatar）GPU

本节介绍如何在 DCE 5.0 平台使用天数智芯虚拟 GPU。

### 前提条件

- 已经[部署 DCE 5.0](#) 容器管理平台，且平台运行正常。

- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 当前集群已安装天数智芯 GPU 驱动，驱动安装请参考[天数智芯官方文档](#)，或联系道客生态团队获取企业级支持：[peg-pem@daocloud.io](mailto:peg-pem@daocloud.io)。
- 当前集群内 GPU 卡未进行任何虚拟化操作且未被其它 App 占用。

## 操作步骤

### 使用界面配置

- 确认集群是否已检测 GPU 卡。点击对应 **集群** -> **集群设置** -> **Addon 插件**，查看是否已自动启用并自动检测对应 GPU 类型。目前集群会自动启用 **GPU**，并且设置 **GPU** 类型为 **Iluvatar**。

集群设置

集群设置

- 部署工作负载。点击对应 **集群** -> **工作负载**，通过镜像方式部署工作负载，选择类型（**Iluvatar**）之后，需要配置 App 使用的 GPU 资源：

- 物理卡数量（**iluvatar.ai/vcuda-core**）：表示当前 Pod 需要挂载几张物理卡，输入值必须为整数且 **小于等于** 宿主机上的卡数量。
- 显存使用数量（**iluvatar.ai/vcuda-memory**）：表示每张卡占用的 GPU 显存，值单位为 MB，最小值为 1，最大值为整卡的显存值。

负载使用

负载使用

如果上述值配置有问题则会出现调度失败，资源分配不了的情况。

## 使用 YAML 配置

创建工作负载申请 GPU 资源，在资源申请和限制配置中增加 iluvatar.ai/vcuda-core: 1、

iluvatar.ai/vcuda-memory: 200 参数，配置 App 使用物理卡的资源。

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: full-iluvatar-gpu-demo
 namespace: default
spec:
 replicas: 1
 selector:
 matchLabels:
 app: full-iluvatar-gpu-demo
 template:
 metadata:
 labels:
 app: full-iluvatar-gpu-demo
 spec:
 containers:
 - image: nginx:perl
 name: container-0
 resources:
 limits:
 cpu: 250m
 iluvatar.ai/vcuda-core: '1'
 iluvatar.ai/vcuda-memory: '200'
 memory: 512Mi
 requests:
 cpu: 250m
 memory: 512Mi
 imagePullSecrets:
 - name: default-secret
```

## 沐曦 GPU 组件安装与使用

本章节提供沐曦 gpu-extensions、gpu-operator 等组件的安装指导和沐曦 GPU 整卡和 vGPU 两种模式的使用方法。

## 前提条件

1. 已在 [沐曦软件中心](#) 下载并安装所需的 tar 包，本文以 metax-gpu-k8s-package.0.7.10.tar.gz 为例。
2. 准备 Kubernetes 基础环境

## 组件介绍

Metax 提供了两个 helm-chart 包，一个是 metax-extensions，一个是 gpu-operator，根据使用场景可选择安装不同的组件。

1. Metax-extensions：包含 gpu-device 和 gpu-label 两个组件。在使用 Metax-extensions 方案时，用户的应用容器镜像需要基于 MXMACA® 基础镜像构建。且 Metax-extensions 仅适用于 GPU 整卡使用场景。
2. gpu-operator：包含 gpu-device、gpu-label、driver-manager、container-runtime、operator-controller 这些组件。使用 gpu-operator 方案时，用户可选择制作不包含 MXMACA® SDK 的应用容器镜像。gpu-operator 适用于 GPU 整卡和 vGPU 场景。

## 操作步骤

1. 从 /home/metax/metax-docs/k8s/metax-gpu-k8s-package.0.7.10.tar.gz 文件中解压出
  - deploy-gpu-extensions.yaml # 部署 yaml
  - metax-gpu-extensions-0.7.10.tgz、metax-operator-0.7.10.tgz # helm chart 文件
  - metax-k8s-images.0.7.10.run # 离线镜像
2. 查看系统是否安装驱动

```
$ lsmod | grep metax
metax 1605632 0
ttm 86016 3 drm_vram_helper,metax,drm_ttm_helper
drm 618496 7 drm_kms_helper,drm_vram_helper,ast,metax,drm_ttm_helper,ttm
```

- 如没有内容显示，就表示没有安装过软件包。如有内容显示，则表示安装过软件包。
- 使用 metax-operator 时，不推荐在工作节点预安装 MXMACA 内核态驱动，若已安装也无需卸载。

### 3. 安装驱动

## gpu-extensions

### 1. 推送镜像

```
tar -xf metax-gpu-k8s-package.0.7.10.tar.gz
./metax-k8s-images.0.7.10.run push {registry}/metax
```

### 2. 推送 Helm Chart

```
helm plugin install https://github.com/chartmuseum/helm-push
helm repo add --username rootuser --password rootpass123 metax http://172.16.16.5:8081
helm cm-push metax-operator-0.7.10.tgz metax
helm cm-push metax-gpu-extensions-0.7.10.tgz metax
```

### 3. 在 DCE 5.0 平台上安装 metax-gpu-extensions

部署成功之后，可以在节点上查看到资源。

```

Addresses:
 InternalIP: 172.16.16.1
 Hostname: metax

Capacity:
 cpu: 96
 ephemeral-storage: 195812980Ki
 hugepages-1Gi: 0
 hugepages-2Mi: 0
 memory: 1056550084Ki
 metax-tech.com/gpu: 4
 pods: 250

Allocatable:
 cpu: 96
 ephemeral-storage: 180461242070
 hugepages-1Gi: 0
 hugepages-2Mi: 0
 memory: 1056447684Ki
 metax-tech.com/gpu: 4
 pods: 250

System Info:
 Machine ID: a6d63dd6ca54af81a93498bf8f7e44

```

[查看资源](#)

#### 4. 修改成功之后就可以在节点上看到带有 Metax GPU 的标签

名称	状态	角色	标签	CPU (分配/使用率)	内存 (分配/使用率)	IP 地址	创建时间
metax	Metax-GPU ● 健康 ① 可调度	控制器	+1	0% / 0%	0% / 0%	172.16.16.1	2024-07-23 11:10

metax 节点标签

## gpu-operator

安装 gpu-operator 时的已知问题：

1. metax-operator、gpu-label、gpu-device、container-runtime 这几个组件镜像要带有 amd64 后缀。
2. metax-maca 组件的镜像不在 metax-k8s-images.0.7.13.run 包里面，需要单独下载 maca-mxc500-2.23.0.23-ubuntu20.04-x86\_64.tar.xz 这类镜像，load 之后重新修改

metax-maca 组件的镜像。

3. metax-driver 组件的镜像需要从 <https://pub-docstore.metax-tech.com:7001> 这个网站

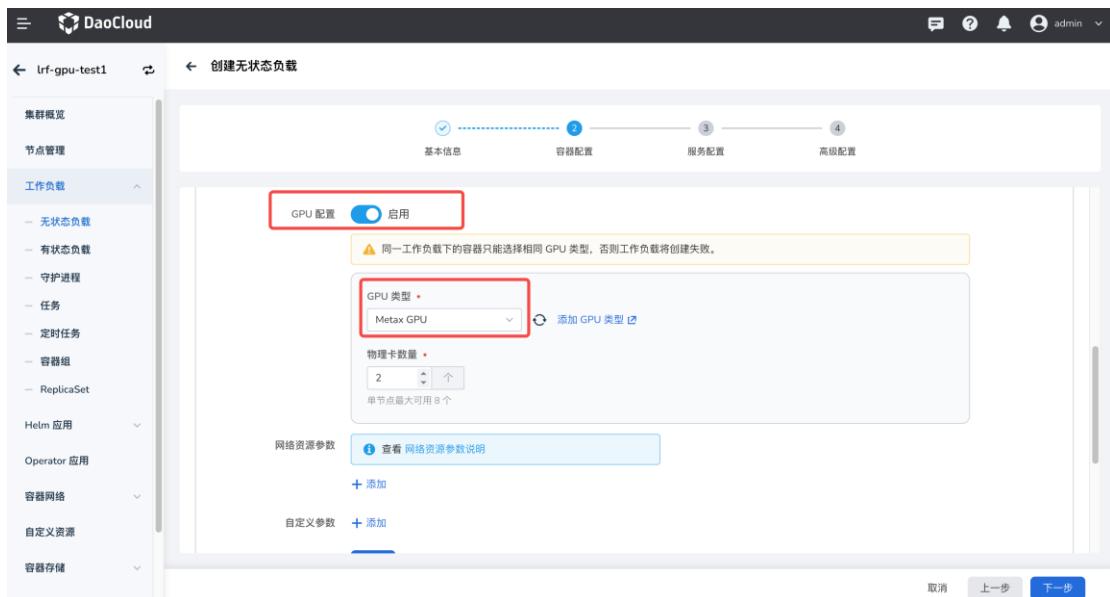
下载 k8s-driver-image.2.23.0.25.run 文件，然后执行 k8s-driver-image.2.23.0.25.run

push {registry}/metax 命令把镜像推送到镜像仓库。推送之后修改 metax-driver 组

件的镜像地址。

## 使用 GPU

安装后可在工作负载中[使用沐曦 GPU](#)。注意启用 GPU 后，需选择 GPU 类型为 Metax GPU



### 使用 GPU

进入容器，执行 mx-smi 可查看 GPU 的使用情况。

```

root@metax:/home/daocloud# kubectl exec -it ubuntu-deployment-7c8fc58b89-fhvkv2 /bin/bash
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl exec [POD] -- [COMMAND] instead.
root@ubuntu-deployment-7c8fc58b89-fhvkv2:/# mx-smi
mx-smi version: 2.1.3

===== MetaX System Management Interface Log =====
Timestamp : Thu Aug 22 11:18:58 2024

Attached GPUs : 2
+-----+-----+
| MX-SMI 2.1.3 | Kernel Mode Driver Version: 2.3.0 |
| MACA Version: 2.20 | BIOS Version: 1.7.4.0 |
+-----+
| GPU NAME Bus-id GPU-Util |
| Temp Power | Memory-Usage |
+-----+
| 0 MXC280 VF | 0000:2a:00.2 | 0% |
| N/A N/A | 522/32768 MiB | |
+-----+
| 1 MXC280 VF | 0000:99:00.1 | 0% |
| N/A N/A | 522/32512 MiB | |
+-----+
+-----+
| Process: |
| GPU PID Process Name GPU Memory |
| | Usage(MiB) |
+-----+
| no process found |
+-----+

```

## 使用 GPU

# 使用寒武纪 GPU

本文介绍如何在 DCE 5.0 中使用寒武纪 GPU。

## 前置条件

- 已经[部署 DCE 5.0](#) 容器管理平台，且平台运行正常。
- 容器管理模块[已接入 Kubernetes 集群](#)或者[已创建 Kubernetes 集群](#)，且能够访问集群的 UI 界面。
- 当前集群已安装寒武纪固件、驱动以及 DevicePlugin 组件，安装详情请参考官方文档：
  - [驱动固件安装](#)
  - [DevicePlugin 安装](#)

在安装 DevicePlugin 时请关闭 `--enable-device-type` 参数，否则 DCE5.0 将无法正确识别寒武纪 GPU。

## 寒武纪 GPU 模式介绍

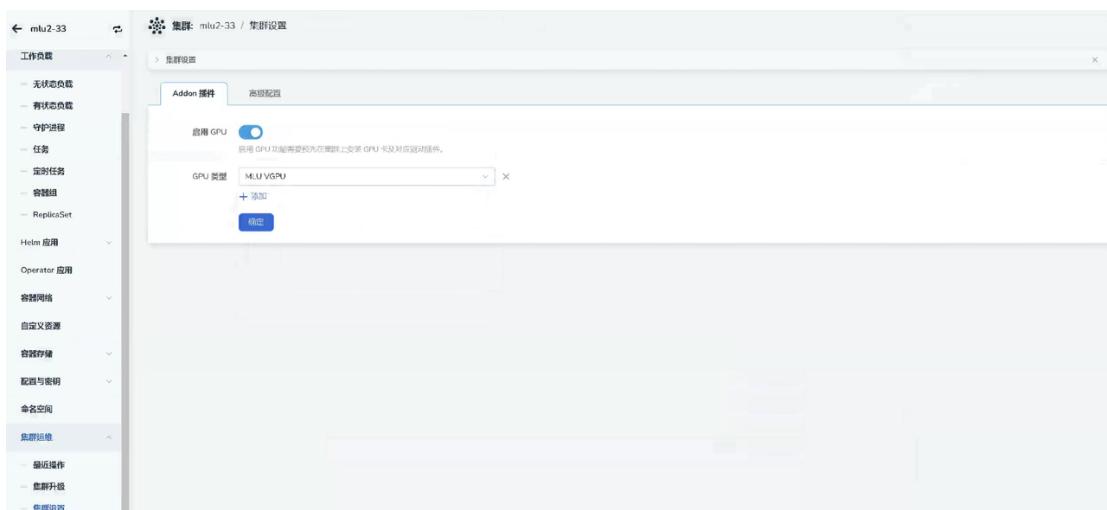
寒武纪 GPU 有以下几种模式：

- 整卡模式：将寒武纪 GPU 以整卡的方式注册到集群当中进行使用。
- Share 模式：可以将一张寒武纪 GPU 共享给多个 Pod 进行使用，可以通过 `virtualization-num` 参数进行设置可共享容器的数量。
- Dynamic smlu 模式：进一步对资源进行了细化，可以控制分配给容器的显存、算力的大小。
- Mim 模式：可以将寒武纪 GPU 按照固定的规格切分成多张 GPU 进行使用。

## DCE 5.0 使用寒武纪

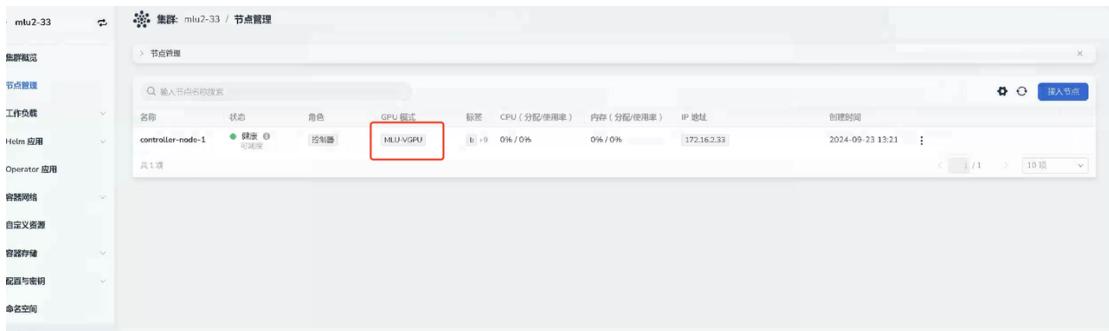
这里以 Dynamic smlu 模式为例：

1. 在正确安装 DevicePlugin 等组件后，点击对应 集群 -> 集群运维-> 集群设置 -> Addon 插件，查看是否已自动启用并自动检测对应 GPU 类型。



mlu 类型

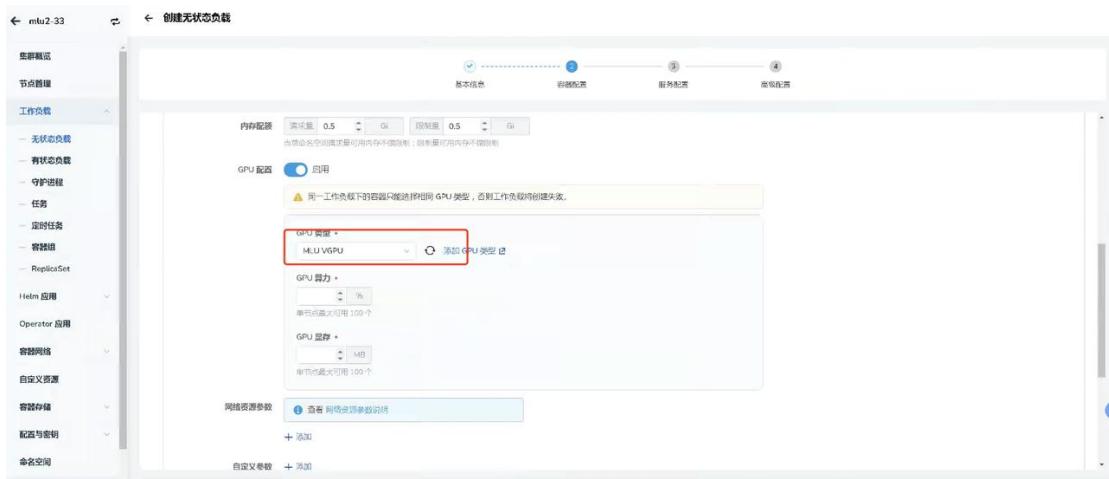
2. 点击节点管理页面，查看节点是否已经正确识别到对应的 GPU 类型。



## 节点列表

3. 部署工作负载。点击对应 **集群 -> 工作负载**，通过镜像方式部署工作负载，选择类型 ( MLU VGPU ) 之后，需要配置 App 使用的 GPU 资源：

- GPU 算力 ( cambricon.com/mlu.smlu.vcore ) : 表示当前 Pod 需要使用核心的百分比数量。
- GPU 显存 ( cambricon.com/mlu.smlu.vmemory ) : 表示当前 Pod 需要使用显存的大小，单位是 MB。



## 使用 mlu

# 使用 YAML 配置

参考 YAML 文件如下：

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```

name: pod1
spec:
 restartPolicy: OnFailure
 containers:
 - image: ubuntu:16.04
 name: pod1-ctr
 command: ["sleep"]
 args: ["100000"]
 resources:
 limits:
 cambricon.com/mlu: "1" # use this when device type is not enabled, else delete this
line.
 #cambricon.com/mlu: "1" #uncomment to use when device type is enabled
 #cambricon.com/mlu.share: "1" #uncomment to use device with env-share mode
 #cambricon.com/mlu.mim-2m.8gb: "1" #uncomment to use device with mim mode
 #cambricon.com/mlu.smlu.vcore: "100" #uncomment to use device with mim mode
 #cambricon.com/mlu.smlu.vmemory: "1024" #uncomment to use device with mim mo
de

```

## GPU 相关 FAQ

### Pod 内 nvidia-smi 看不到 GPU 进程

Q: 在使用 GPU 的 Pod 内执行 nvidia-smi 命令看不到使用 GPU 的进程信息，包括整卡模式、vGPU 模式等。

A: 因为有 PID namespace 隔离，导致在 Pod 内查看不到 GPU 进程，如果要查看 GPU 进程有如下几种方法：

- 在使用 GPU 的工作负载配置 hostPID: true，使其可以查看到宿主机上的 PID
- 在 gpu-operator 的 driver Pod 中执行 nvidia-smi 命令查看进程
- 在宿主机上执行 chroot /run/nvidia/driver nvidia-smi 命令查看进程

# ETCD 备份还原

使用 ETCD 备份功能创建备份策略，可以将指定集群的 etcd 数据定时备份到 S3 存储中。

本文主要介绍如何将已经备份的数据还原到当前集群中。

!!! note

- DCE 5.0 ETCD 备份还原仅限于针对同一集群（节点数和 IP 地址没有变化）进行备份与还原。

例如，备份了 A 集群的 etcd 数据后，只能将备份数据还原到 A 集群中，不能还原到 B 集群。

- 对于跨集群的备份与还原，建议使用[应用备份还原](./user-guide/backup/deployment.md)功能
  -
- 首先创建备份策略，备份当前状态，建议参考[ETCD 备份](./user-guide/backup/etcd-backup.md)功能。

下面通过具体的案例来说明备份还原的整个过程。

## 环境信息

首先介绍还原的目标集群和 S3 存储的基本信息。这里以 Minio 作为 S3 存储，整个集群有 3 个控制面（3 个 etcd 副本）。

IP	主机	角色	备注
10.6.212.10	host01	k8s-master01	k8s 节点 1
10.6.212.11	host02	k8s-master02	k8s 节点 2
10.6.212.12	host03	k8s-master03	k8s 节点 3
10.6.212.13	host04	minio	minio 服务

## 前提条件

### 安装 etcdbrctl 工具

为了实现 ETCD 数据备份还原，需要在上述任意一个 Kubernetes 节点上安装 etcdbrctl 开源工具。此工具暂时没有二进制文件，需要自行编译。编译方式请参考 [Gardener / etcd-backup-restore 本地开发文档](#)。

安装完成后用如下命令检查工具是否可用：

```
etcdbrctl -v
```

预期输出如下：

```
INFO[0000] etcd-backup-restore Version: v0.23.0-dev
INFO[0000] Git SHA: b980beec
INFO[0000] Go Version: go1.19.3
INFO[0000] Go OS/Arch: linux/amd64
```

### 检查备份数据

还原之前需要检查下列事项：

- 是否已经在 DCE 5.0 中成功备份了数据
- 检查 S3 存储中备份数据是否存在

!!! note

DCE 5.0 的备份是全量数据备份，还原时将还原最后一次备份的全量数据。

minio

minio

### 关闭集群

在备份之前，必须要先关闭集群。默认集群 **etcd** 和 **kube-apiserver** 都是以静态 Pod 的形式启动的。这里的关闭集群是指将静态 Pod manifest 文件移动到 **/etc/kubernetes/manifest** 目录外，集群就会移除对应 Pod，达到关闭服务的作用。

1. 首先删除之前的备份数据，移除数据并非将现有 etcd 数据删除，而是指修改 etcd 数据目录的名称。等备份还原成功之后再删除此目录。这样做的目的是，如果 etcd 备份还原失败，还可以尝试还原当前集群。此步骤每个节点均需执行。

```
rm -rf /var/lib/etcd_bak
```

2. 然后需要关闭 **kube-apiserver** 的服务，确保 etcd 的数据没有新变化。此步骤每个节点均需执行。

```
mv /etc/kubernetes/manifests/kube-apiserver.yaml /tmp/kube-apiserver.yaml
```

3. 同时还需要关闭 etcd 的服务。此步骤每个节点均需执行。

```
mv /etc/kubernetes/manifests/etcd.yaml /tmp/etcd.yaml
```

4. 确保所有控制平面的 **kube-apiserver** 和 **etcd** 服务都已经关闭。

5. 关闭所有的节点后，使用如下命令检查 **etcd** 集群状态。此命令在任意一个节点执行即可。

**endpoints** 的值需要替换为实际节点名称

```
etcdctl endpoint status --endpoints=controller-node-1:2379,controller-node-2:2379,controller-node-3:2379 -w table \
--cacert="/etc/kubernetes/ssl/etcd/ca.crt" \
--cert="/etc/kubernetes/ssl/apiserver-etcd-client.crt" \
--key="/etc/kubernetes/ssl/apiserver-etcd-client.key"
```

预期输出如下，表示所有的 **etcd** 节点都被销毁：

```
{"level":"warn","ts":"2023-03-29T17:51:50.817+0800","logger":"etcd-client","caller":"v3@v3.5.6/retry_interceptor.go:62","msg":"retrying of unary invoker failed","target":"etcd-endpoints://0xc0001ba000/controller-node-1:2379","attempt":0,"error":"rpc error: code = DeadlineExceeded desc = latest balancer error: last connection error: connection error: desc = \"transport: Error while dialing dial tcp 10.5.14.31:2379: connect: connection refused\""} Failed to get the status of endpoint controller-node-1:2379 (context deadline exceeded) {"level":"warn","ts":"2023-03-29T17:51:55.818+0800","logger":"etcd-client","caller":"v3@v3.5.6/retry_interceptor.go:62","msg":"retrying of unary invoker failed","target":"etcd-endpoints://0xc0001ba000/controller-node-2:2379","attempt":0,"error":"rpc error: code = DeadlineExceeded desc = latest balancer error: last connection error: connection error: desc = \"transport: Error while dialing dial tcp 10.5.14.32:2379: connect: connection refused\""} Failed to get the status of endpoint controller-node-2:2379 (context deadline exceeded) {"level":"warn","ts":"2023-03-29T17:52:00.820+0800","logger":"etcd-client","caller":"v3@v3.5.6/retry_interceptor.go:62","msg":"retrying of unary invoker failed","target":"etcd-endpoints://0xc0001ba000/controller-node-3:2379","attempt":0,"error":"rpc error: code = DeadlineExceeded desc = latest balancer error: last connection error: connection error: desc = \"transport: Error while dialing dial tcp 10.5.14.33:2379: connect: connection refused\""} Failed to get the status of endpoint controller-node-3:2379 (context deadline exceeded)
```

```

3.5.6/retry_interceptor.go:62","msg":"retrying of unary invoker failed","target":"etcd-endpo
ints://0xc0001ba000/controller-node-1:2379","attempt":0,"error":"rpc error: code = Deadlin
eExceeded desc = latest balancer error: last connection error: connection error: desc =
\"transport: Error while dialing dial tcp 10.5.14.33:2379: connect: connection refused\""
}

Failed to get the status of endpoint controller-node-3:2379 (context deadline exceeded)
+-----+-----+-----+-----+-----+-----+
+-----+
| ENDPOINT | ID | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TE
RM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+-----+-----+-----+-----+-----+-----+
+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+

```

## 还原备份

只需要还原一个节点的数据，其他节点的 etcd 数据就会自动进行同步。

### 1. 设置环境变量

使用 etcdbrctl 还原数据之前，执行如下命令将连接 S3 的认证信息设置为环境变

量：

```

export ECS_ENDPOINT=http://10.6.212.13:9000 # (1)!
export ECS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE # (2)!
export ECS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLE
KEY # (3)!
```

1. S3 存储的访问点

2. S3 存储的用户名

3. S3 存储的密码

### 2. 执行还原操作

执行 etcdbrctl 命令行工具执行还原，这是最关键的一步。

```

etcdbrctl restore --data-dir /var/lib/etcd/ --store-container="etcd-backup" \
--storage-provider=ECS \
--initial-cluster=controller-node1=https://10.6.212.10:2380 \
--initial-advertise-peer-urls=https://10.6.212.10:2380
```

参数说明如下：

- –data-dir: etcd 数据目录，此目录必须跟 etcd 数据目录一致，etcd 才能正常加载数据。
- –store-container : S3 存储的位置，MinIO 中对应的 bucket，必须跟数据备份的 bucket 相对应。
- –initial-cluster : etcd 初始化配置, etcd 集群的名称必须跟原来一致。
- –initial-advertise-peer-urls : etcd member 集群之间访问地址。必须跟 etcd 的配置保持一致。

预期输出如下：

```
INFO[0000] Finding latest set of snapshot to recover from...
INFO[0000] Restoring from base snapshot: Full-00000000-00111147-1679991074 actor=restorer
INFO[0001] successfully fetched data of base snapshot in 1.241380207 seconds actor=restorer
{"level":"info","ts":1680011221.2511616,"caller":"mvcc/kvstore.go:380","msg":"restored last compact revision","meta-bucket-name":"meta","meta-bucket-name-key":"finishedCompactRev","restored-compact-revision":110327}
{"level":"info","ts":1680011221.3045986,"caller":"membership/cluster.go:392","msg":"added member","cluster-id":"66638454b9dd7b8a","local-member-id":"0","added-peer-id":"123c2503a378fc46","added-peer-peer-urls":["https://10.6.212.10:2380"]}
INFO[0001] Starting embedded etcd server... actor=restorer
....
{"level":"info","ts":2023-03-28T13:47:02.922Z,"caller":"embed/etcd.go:565","msg":"stopped serving peer traffic","address":"127.0.0.1:37161"}
{"level":"info","ts":2023-03-28T13:47:02.922Z,"caller":"embed/etcd.go:367","msg":"closed etcd server","name":"default","data-dir":"/var/lib/etcd","advertise-peer-urls":["http://localhost:0"],"advertise-client-urls":["http://localhost:0"]}
INFO[0003] Successfully restored the etcd data directory.
```

!!! note “可以查看 etcd 的 YAML 文件进行对照，以免配置错误”

```
```shell
cat /tmp/etcd.yaml | grep initial-
- --experimental-initial-corrupt-check=true
- --initial-advertise-peer-urls=https://10.6.212.10:2380
```

```
- --initial-cluster=controller-node-1=https://10.6.212.10:2380
```

```

3. 以下命令在节点 01 上执行，为了恢复节点 01 的 etcd 服务。

首先将 etcd 静态 Pod 的 manifest 文件移动到 `/etc/kubernetes/manifests` 目录下，

kubelet 将会重启 etcd：

```
mv /tmp/etcd.yaml /etc/kubernetes/manifests/etcd.yaml
```

然后等待 etcd 服务启动完成以后，检查 etcd 的状态，etcd 相关证书默认目录：

`/etc/kubernetes/ssl`。如果集群证书存放在其他位置，请指定对应路径。

- 检查 etcd 集群列表：

```
etcdctl member list -w table \
--cacert="/etc/kubernetes/ssl/etcd/ca.crt" \
--cert="/etc/kubernetes/ssl/apiserver-etcd-client.crt" \
--key="/etc/kubernetes/ssl/apiserver-etcd-client.key"
```

预期输出如下：

```
+-----+-----+-----+-----+
|-----+
| ID | STATUS | NAME | PEER AD
| DRS | CLIENT ADDRS | IS LEARNER |
+-----+-----+-----+-----+
|-----+
| 123c2503a378fc46 | started | controller-node-1 | https://10.6.212.10:2380 | http
| s://10.6.212.10:2379 | false |
+-----+-----+-----+-----+
|-----+
```

- 查看 controller-node-1 状态：

```
etcdctl endpoint status --endpoints=controller-node-1:2379 -w table \
--cacert="/etc/kubernetes/ssl/etcd/ca.crt" \
--cert="/etc/kubernetes/ssl/apiserver-etcd-client.crt" \
--key="/etc/kubernetes/ssl/apiserver-etcd-client.key"
```

预期输出如下：

```
+-----+-----+-----+-----+
|-----+
| ENDPOINT | ID | VERSION | DB SIZE | I
| S LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIE
| D INDEX | ERRORS |
+-----+-----+-----+-----+
```

|  | controller-node-1:2379 |  | 123c2503a378fc46 |  | 3.5.6 |  | 15 MB |  | true |  |  |
|--|------------------------|--|------------------|--|-------|--|-------|--|------|--|--|
|  | false                  |  | 3                |  | 1200  |  | 1199  |  |      |  |  |
|  |                        |  |                  |  |       |  |       |  |      |  |  |
|  |                        |  |                  |  |       |  |       |  |      |  |  |

#### 4. 恢复其他节点数据

上述步骤已经还原了节点 01 的数据，若想要还原其他节点数据，只需要将 etcd 的

Pod 启动起来，让 etcd 自己完成数据同步。

- 在节点 02 和节点 03 都执行相同的操作：

```
mv /tmp/etcd.yaml /etc/kubernetes/manifests/etcd.yaml
```

- etcd member 集群之间的数据同步需要一定的时间，可以查看 etcd 集群状态，确保所有 etcd 集群正常：

检查 etcd 集群状态是否正常：

```
etcdctl member list -w table \
--cacert="/etc/kubernetes/ssl/etcd/ca.crt" \
--cert="/etc/kubernetes/ssl/apiserver-etcd-client.crt" \
--key="/etc/kubernetes/ssl/apiserver-etcd-client.key"
```

预期输出如下：

|    | ID               | STATUS       | NAME              | PEER ADDRS              |                                 |
|----|------------------|--------------|-------------------|-------------------------|---------------------------------|
| RS |                  | CLIENT ADDRS |                   | IS LEARNER              |                                 |
|    |                  |              |                   |                         |                                 |
|    |                  |              |                   |                         |                                 |
|    | 6ea47110c5a87c03 | started      | controller-node-1 | https://10.5.14.31:2380 | https://10.5.14.31:2379   false |
|    | e222e199f1e318c4 | started      | controller-node-2 | https://10.5.14.32:2380 | https://10.5.14.32:2379   false |
|    | f64eeda321aab2d  | started      | controller-node-3 | https://10.5.14.33:2380 | https://10.5.14.33:2379   false |
|    |                  |              |                   |                         |                                 |
|    |                  |              |                   |                         |                                 |

检查 3 个 member 节点是否正常：

```
etcdctl endpoint status --endpoints=controller-node-1:2379,controller-node-2:2379,co
ntroller-node-3:2379 -w table \
--cacert="/etc/kubernetes/ssl/etcd/ca.crt" \
--cert="/etc/kubernetes/ssl/apiserver-etcd-client.crt" \
--key="/etc/kubernetes/ssl/apiserver-etcd-client.key"
```

预期输出如下：

| ENDPOINT               | ID               | VERSION | DB SIZE | I<br>S LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIE<br>D INDEX | ERRORS |
|------------------------|------------------|---------|---------|----------------|-----------|------------|------------------------|--------|
| controller-node-1:2379 | 6ea47110c5a87c03 | 3.5.6   | 88 MB   | true           | false     | 199008     | 199008                 |        |
| controller-node-2:2379 | e222e199f1e318c4 | 3.5.6   | 88 MB   | false          | false     | 199114     | 199114                 |        |
| controller-node-3:2379 | f64eeda321aabe2d | 3.5.6   | 88 MB   | false          | false     | 199316     | 199316                 |        |

## 恢复集群

等所有节点的 etcd 数据同步完成后，则可以将 kube-apiserver 进行重新启动，将整个集群恢复到可访问状态：

### 1. 重新启动 node1 的 kube-apiserver 服务

```
mv /tmp/kube-apiserver.yaml /etc/kubernetes/manifests/kube-apiserver.yaml
```

### 2. 重新启动 node2 的 kube-apiserver 服务

```
mv /tmp/kube-apiserver.yaml /etc/kubernetes/manifests/kube-apiserver.yaml
```

### 3. 重新启动 node3 的 kube-apiserver 服务

```
mv /tmp/kube-apiserver.yaml /etc/kubernetes/manifests/kube-apiserver.yaml
```

### 4. 等待 kubelet 将 kube-apiserver 启动后，检查还原的 k8s 数据是否正常：

```
kubectl get nodes
```

预期输出如下：

| NAME              | STATUS | ROLES         | AGE   | VERSION |
|-------------------|--------|---------------|-------|---------|
| controller-node-1 | Ready  | <none>        | 3h30m | v1.25.4 |
| controller-node-3 | Ready  | control-plane | 3h29m | v1.25.4 |
| controller-node-3 | Ready  | control-plane | 3h28m | v1.25.4 |

# MySQL 应用及数据的跨集群备份恢复

本次演示将基于 DCE 5.0 的应用备份功能，实现一个有状态应用的跨集群备份迁移。

!!! note

当前操作者应具有 DCE 5.0 平台管理员的权限。

## 准备演示环境

### 准备两个集群

**main-cluster** 作为备份数据的源集群， **recovery-cluster** 集群作为需要恢复数据的目标集群。

| 集群               | IP           | 节点   |
|------------------|--------------|------|
| main-cluster     | 10.6.175.100 | 1 节点 |
| recovery-cluster | 10.6.175.110 | 1 节点 |

## 搭建 MinIO 配置

MinIO 服务器访问地址 存储桶 用户名 密码

`http://10.7.209.110:9000 mysql-demo root dangerous`

## 在两个集群部署 NFS 存储服务

!!! note

需要在 \*\*源集群和目标集群\*\* 上的所有节点上部署 NFS 存储服务。

1. 在两个集群中的所有节点安装 NFS 所需的依赖。

```
yum install nfs-utils iscsi-initiator-utils nfs-utils iscsi-initiator-utils nfs-utils iscsi-initiator-util
s -y
```

预期输出为：

```
[root@g-master1 ~]# kubectl apply -f nfs.yaml
clusterrole.rbac.authorization.k8s.io/nfs-provisioner-runner created
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-provisioner created
role.rbac.authorization.k8s.io/leader-locking-nfs-provisioner created
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-provisioner created
serviceaccount/nfs-provisioner created
service/nfs-provisioner created
deployment.apps/nfs-provisioner created
storageclass.storage.k8s.io/nfs created
```

## 2. 为 MySQL 应用准备 NFS 存储服务。

登录 **main-cluster** 集群和 **recovery-cluster** 集群的任一控制节点，使用 **vi nfs.yaml** 命令在节点上创建一个名为 **nfs.yaml** 的文件，将下面的 YAML 内容复制到 **nfs.yaml** 文件。

[点击查看完整的 nfs.yaml](#)

```
```yaml title="nfs.yaml" kind: ClusterRole apiVersion: rbac.authorization.k8s.io/v1 metadata:
  name: nfs-provisioner-runner namespace: nfs-system rules:
    - apiGroups: [""]
      resources: ["persistentvolumes"]
      verbs: ["get", "list", "watch", "create", "delete"]
    - apiGroups: [""]
      resources: ["persistentvolumeclaims"]
      verbs: ["get", "list", "watch", "update"]
    - apiGroups: ["storage.k8s.io"]
      resources: ["storageclasses"]
      verbs: ["get", "list", "watch"]
    - apiGroups: [""]
      resources: ["events"]
      verbs: ["create", "update", "patch"]
    - apiGroups: [""]
      resources: ["services", "endpoints"]
      verbs: ["get"]
    - apiGroups: ["extensions"]
      resources: ["podsecuritypolicies"]
      resourceNames: ["nfs-provisioner"]
      verbs: ["use"]
    — kind: ClusterRoleBinding apiVersion: rbac.authorization.k8s.io/v1 metadata: name: run-nfs-provisioner subjects:
      - kind: ServiceAccount name: nfs-provisioner # replace with namespace where provisioner is deployed
        namespace: default
        roleRef:
          kind: ClusterRole
          name: nfs-provisioner-runner
          apiGroup: rbac.authorization.k8s.io
    — kind: Role apiVersion: rbac.authorization.k8s.io/v1 metadata: name: leader-locking-nfs-provisioner rules:
      - apiGroups: [""]
        resources: ["endpoints"]
        verbs: ["get", "list", "watch", "create", "update", "patch"]
    — kind: RoleBinding apiVersion: rbac.authorization.k8s.io/v1 metadata: name: leader-locking-nfs-provisioner subjects:
```

```
- kind: ServiceAccount name: nfs-provisioner # replace with namespace where
provisioner is deployed namespace: default roleRef: kind: Role name:
leader-locking-nfs-provisioner apiGroup: rbac.authorization.k8s.io — apiVersion:
v1 kind: ServiceAccount metadata: name: nfs-provisioner — kind: Service
apiVersion: v1 metadata: name: nfs-provisioner labels: app: nfs-provisioner spec:
ports:
  • name: nfs port: 2049
  • name: nfs-udp port: 2049 protocol: UDP
  • name: nlockmgr port: 32803
  • name: nlockmgr-udp port: 32803 protocol: UDP
  • name: mountd port: 20048
  • name: mountd-udp port: 20048 protocol: UDP
  • name: rquotad port: 875
  • name: rquotad-udp port: 875 protocol: UDP
  • name: rpcbind port: 111
  • name: rpcbind-udp port: 111 protocol: UDP
  • name: statd port: 662
  • name: statd-udp port: 662 protocol: UDP selector: app: nfs-provisioner
    — kind: Deployment apiVersion: apps/v1 metadata: name:
nfs-provisioner spec: selector: matchLabels: app: nfs-provisioner replicas: 1 strategy: type: Recreate template: metadata: labels: app:
nfs-provisioner spec: serviceAccount: nfs-provisioner containers:
  — name: nfs-provisioner resources: limits: cpu: "1" memory:
"4294967296" image:
release.daocloud.io/velero/nfs-provisioner:v3.0.0 ports:
  • name: nfs containerPort: 2049
  • name: nfs-udp containerPort: 2049 protocol: UDP
  • name: nlockmgr containerPort: 32803
  • name: nlockmgr-udp containerPort: 32803 protocol: UDP
  • name: mountd containerPort: 20048
  • name: mountd-udp containerPort: 20048 protocol: UDP
  • name: rquotad containerPort: 875
  • name: rquotad-udp containerPort: 875 protocol: UDP
  • name: rpcbind containerPort: 111
  • name: rpcbind-udp containerPort: 111 protocol: UDP
  • name: statd containerPort: 662
  • name: statd-udp containerPort: 662 protocol: UDP
    securityContext: capabilities: add:
      - DAC_READ_SEARCH
      - SYS_RESOURCE args:
        • "-provisioner=example.com/nfs" env:
        • name: POD_IP valueFrom: fieldRef: fieldPath:
status.podIP
        • name: SERVICE_NAME value: nfs-provisioner
```

```

        • name: POD_NAMESPACE valueFrom: fieldRef: fieldPath:
          metadata.namespace imagePullPolicy: "IfNotPresent"
          volumeMounts:
        • name: export-volume mountPath: /export volumes:
      -name: export-volume hostPath: path: /data — kind: StorageClass
        apiVersion: storage.k8s.io/v1 metadata: name: nfs provisioner:
          example.com/nfs mountOptions:
      - vers=4.1 ``

```

3. 在两个集群的控制节点上执行 **nfs.yaml** 文件。

kubectl apply -f nfs.yaml

4. 查看 NFS Pod 状态，等待其状态变为 **running**（大约需要 2 分钟）。

kubectl get pod -n nfs-system -owide

预期输出为：

```
[root@g-master1 ~]# kubectl get pod -owide
  NAME           READY   STATUS    RESTARTS   AGE
  IP             NODE     NOMINATED_NODE   READINESS GATES
  nfs-provisioner-7dfb9bcc45-74ws2   1/1     Running   0          4m45s   10.6.175.
  100   g-master1   <none>            <none>
```

部署 MySQL 应用

1. 为 MySQL 应用准备基于 NFS 存储的 PVC，用来存储 MySQL 服务内的数据。

使用 **vi pvc.yaml** 命令在节点上创建名为 **pvc.yaml** 的文件，将下面的 YAML 内容复制到 **pvc.yaml** 文件内。

```
```yaml title="pvc.yaml" apiVersion: v1 kind: PersistentVolumeClaim metadata: name: mydata namespace: default spec: accessModes:
 - ReadWriteOnce resources: requests: storage: "1Gi" storageClassName: nfs volumeMode: Filesystem```

```

### 2. 在节点上使用 kubectl 工具执行 **pvc.yaml** 文件。

kubectl apply -f pvc.yaml

预期输出为：

```
[root@g-master1 ~]# kubectl apply -f pvc.yaml
persistentvolumeclaim/mydata created
```

### 3. 部署 MySQL 应用。

使用 `vi mysql.yaml` 命令在节点上创建名为 `mysql.yaml` 的文件，将下面的 YAML 内容复制到 `mysql.yaml` 文件。

[点击查看完整的 mysql.yaml](#)

```
yaml title="nfs.yaml" apiVersion: apps/v1 kind: Deployment metadata: labels:
 app: mysql-deploy name: mysql-deploy namespace: default spec:
 progressDeadlineSeconds: 600 replicas: 1 revisionHistoryLimit: 10 selector:
 matchLabels: app: mysql-deploy strategy: rollingUpdate:
 maxSurge: 25% maxUnavailable: 25% type: RollingUpdate template:
 metadata: creationTimestamp: null labels: app:
 mysql-deploy name: mysql-deploy spec: containers: -
 args: - --ignore-db-dir=lost+found env: - name:
 MYSQL_ROOT_PASSWORD value: dangerous image:
 release.daocloud.io/velero/mysql:5 imagePullPolicy: IfNotPresent
 name: mysql-deploy ports: - containerPort: 3306
 protocol: TCP resources: limits: cpu: "1"
 memory: "4294967296" terminationMessagePath: /dev/termination-log
 terminationMessagePolicy: File volumeMounts: - mountPath:
 /var/lib/mysql name: data dnsPolicy: ClusterFirst
 restartPolicy: Always schedulerName: default-scheduler
 securityContext: fsGroup: 999 terminationGracePeriodSeconds: 30
 volumes: - name: data persistentVolumeClaim:
 claimName: mydata
```

4. 在节点上使用 `kubectl` 工具执行 `mysql.yaml` 文件。

`kubectl apply -f mysql.yaml`

预期输出为：

```
[root@g-master1 ~]# kubectl apply -f mysql.yaml
deployment.apps/mysql-deploy created
```

5. 查看 MySQL Pod 状态。

执行 `kubectl get pod | grep mysql` 查看 MySQL Pod 状态，等待其状态变为 `running`（大约需要 2 分钟）。

预期输出为：

```
[root@g-master1 ~]# kubectl get pod |grep mysql
mysql-deploy-5d6f94cb5c-gkrks 1/1 Running 0 2m53s
!!! note
```

- 如果 MySQL Pod 状态长期处于非 running 状态，通常是因为没有在集群的所有节点上安装 NFS 依赖。
- 执行 `kubectl describe pod ${mysql pod 名称}` 查看 Pod 的详细信息。
- 如果报错中有 `MountVolume.SetUp failed for volume "pvc-4ad70cc6-df37-4253-b0c9-8cb86518ccf8" : mount failed: exit status 32` 之类的信息，请分别执行 `kubectl delete -f nfs.yaml/pvc.yaml/mysql.yaml` 删除之前的资源后，重新从部署 NFS 服务开始。

## 6. 向 MySQL 应用写入数据。

为了便于后期验证迁移数据是否成功，可以使用脚本向 MySQL 应用中写入测试数据。

1. 使用 `vi insert.sh` 命令在节点上创建名为 `insert.sh` 的脚本，将下面的 YAML 内容复制到该脚本。

```
```shell title="insert.sh" #!/bin/bash
function rand(){ min= 1max =(( 2 - min+1)) num=$((date +%s%N) echo
((num%max +min)) }
function insert(){ user=(date +age =(rand 1 100)
sql="INSERT INTO test.users(user_name, age)VALUES('${user}', ${age});"
echo -e ${sql}

kubectl exec deploy/mysql-deploy -- mysql -uroot -pdangerous -e "${sql}"
}
kubectl exec deploy/mysql-deploy - mysql -uroot -pdangerous -e "CREATE
DATABASE IF NOT EXISTS test;" kubectl exec deploy/mysql-deploy - mysql -uroot
-pdangerous -e "CREATE TABLE IF NOT EXISTS test.users(user_name
VARCHAR(10) NOT NULL,age INT UNSIGNED)ENGINE=InnoDB DEFAULT
CHARSET=utf8;"
while true;do insert sleep 1 done ```

2. 为 insert.sh 脚本添加权限并运行该脚本。
```

```
[root@g-master1 ~]# chmod +x insert.sh
[root@g-master1 ~]# ./insert.sh
```

预期输出为：

```
mysql: [Warning] Using a password on the command line interface can be insec
ure.
mysql: [Warning] Using a password on the command line interface can be ins
ecure.
INSERT INTO test.users(user_name, age)VALUES('dc09195ba', 10);
mysql: [Warning] Using a password on the command line interface can be ins
```

```
ecure.  
INSERT INTO test.users(user_name, age)VALUES('80ab6aa28', 70);  
mysql: [Warning] Using a password on the command line interface can be ins  
ecure.  
INSERT INTO test.users(user_name, age)VALUES('f488e3d46', 23);  
mysql: [Warning] Using a password on the command line interface can be ins  
ecure.  
INSERT INTO test.users(user_name, age)VALUES('e6098695c', 93);  
mysql: [Warning] Using a password on the command line interface can be ins  
ecure.  
INSERT INTO test.users(user_name, age)VALUES('eda563e7d', 63);  
mysql: [Warning] Using a password on the command line interface can be ins  
ecure.  
INSERT INTO test.users(user_name, age)VALUES('a4d1b8d68', 17);  
mysql: [Warning] Using a password on the command line interface can be ins  
ecure.
```

3. 在键盘上同时按下 **control** 和 **c** 暂停脚本的执行。

4. 前往 MySQL Pod 查看 MySQL 中写入的数据。

```
kubectl exec deploy/mysql-deploy -- mysql -uroot -pdangerous -e "SELECT * F  
ROM test.users;"
```

预期输出为：

```
mysql: [Warning] Using a password on the command line interface can be insec  
ure.  
+-----+-----+  
| user_name | age |  
+-----+-----+  
| dc09195ba | 10 |  
| 80ab6aa28 | 70 |  
| f488e3d46 | 23 |  
| e6098695c | 93 |  
| eda563e7d | 63 |  
| a4d1b8d68 | 17 |  
| ea47546d9 | 86 |  
| a34311f2e | 47 |  
| 740cefe17 | 33 |  
| ede85ea28 | 65 |  
| b6d0d6a0e | 46 |  
| f0eb38e50 | 44 |  
| c9d2f28f5 | 72 |  
| 8ddaaafc6f | 31 |  
| 3ae078d0e | 23 |  
| 6e041631e | 96 |
```

在两个集群安装 velero 插件

!!! note

需要在 **源集群和目标集群** 上均安装 velero 插件。

参考[安装 velero 插件](#)文档和下方的 MinIO 配置，在 main-cluster 集群和 recovery-cluster

集群上安装 velero 插件。

minio 服务器访问地址 存储桶 用户名 密码

http://10.7.209.110:9000 mysql-demo root dangerous

!!! note

安装插件时需要将 S3url 替换为此次演示准备的 MinIO 服务器访问地址，存储桶替换为 MinIO 中真实存在的存储桶。

备份 MySQL 应用及数据

1. 在备份前我们需要先保证数据库不能有新数据进来，所以要设置为只读模式：

```
mysql> set global read_only=1; #1 是只读, 0 是读写
mysql> show global variables like "%read_only%"; #查询状态
```

2. 为 MySQL 应用及 PVC 数据添加独有的标签： **backup=mysql**，便于备份时选择资源。

```
kubectl label deploy mysql-deploy backup=mysql # 为 __mysql-deploy__ 负载添加标签
kubectl label pod mysql-deploy-5d6f94cb5c-gkrks backup=mysql # 为 mysql pod 添加标签
kubectl label pvc mydata backup=mysql # 为 mysql 的 pvc 添加标签
```

3. 参考[应用备份](#)中介绍的步骤，以及下方的参数创建应用备份。

- 名称： **backup-mysql**（可以自定义）
- 源集群： **main-cluster**
- 命名空间： default
- 资源过滤-指定资源标签： **backup:mysql**

img
img

4. 创建好备份计划之后页面会自动返回备份计划列表，找到新建的备份计划

backup-mysq , 点击更多操作按钮 ... , 选择 **立即执行** 执行新建的备份计划。

img
img

5. 等待备份计划执行完成后，即可执行后续操作。

跨集群恢复 MySQL 应用及数据

1. 登录 DCE 5.0 平台，在左侧导航选择 **容器管理** -> **备份恢复** -> **应用备份**。

img
img

2. 在左侧功能栏选择 **恢复**，然后在右侧点击 **恢复备份**。

img
img

3. 查看以下说明填写参数：

- 名称： **restore-mysql** (可以自定义)
- 备份源集群： **main-cluster**
- 备份计划： **backup-mysql**
- 备份点： **default**
- 恢复目标集群： **recovery-cluster**

img
img

4. 刷新备份计划列表，等待备份计划执行完成。

验证数据是否成功恢复

1. 登录 **recovery-cluster** 集群的控制节点，查看 **mysql-deploy** 负载是否已经成功备份到

当前集群。

kubectl get pod

预期输出如下：

NAME	READY	STATUS	RESTARTS	AGE
mysql-deploy-5798f5d4b8-62k6c	1/1	Running	0	24h

2. 检查 MySQL 数据表中的数据是否恢复成功。

```
kubectl exec deploy/mysql-deploy -- mysql -uroot -pdangerous -e "SELECT * FROM test.users;"
```

预期输出如下：

```
mysql: [Warning] Using a password on the command line interface can be insecure.
```

user_name	age
dc09195ba	10
80ab6aa28	70
f488e3d46	23
e6098695c	93
eda563e7d	63
a4d1b8d68	17
ea47546d9	86
a34311f2e	47
740cefe17	33
ede85ea28	65
b6d0d6a0e	46
f0eb38e50	44
c9d2f28f5	72
8ddaafc6f	31
3ae078d0e	23
6e041631e	96

```
!!! success
```

可以看到，Pod 中的数据和 __main-cluster__ 集群中 Pod 里面的数据一致。这说明已经成功地将 __main-cluster__ 中的 MySQL 应用及其数据跨集群恢复到了 __recovery-cluster__ 集群。

如何加固自建工作集群

在 DCE 5.0 中，使用 CIS Benchmark (CIS) 扫描使用界面创建的工作集群，有一些扫描项并没有通过扫描。本文将基于不同的 CIS Benchmark 版本进行加固说明。

CIS Benchmark 1.27

扫描环境说明：

- kubernetes version: 1.25.4
- containerd: 1.7.0
- kubelet version: 0.4.9
- kubespray version: v2.22

未通过扫描项

- 1.[FAIL] 1.2.5 Ensure that the –kubelet-certificate-authority argument is set as appropriate (Automated)
- 2.[FAIL] 1.3.7 Ensure that the –bind-address argument is set to 127.0.0.1 (Automated)
- 3.[FAIL] 1.4.1 Ensure that the –profiling argument is set to false (Automated)
- 4.[FAIL] 1.4.2 Ensure that the –bind-address argument is set to 127.0.0.1 (Automated)

扫描失败原因分析

- 1.[FAIL] 1.2.5 Ensure that the –kubelet-certificate-authority argument is set as appropriate (Automated)

原因：CIS 要求 kube-apiserver 必须指定 kubelet 的 CA 证书路径：

img

img

- 2.[FAIL] 1.3.7 Ensure that the –bind-address argument is set to 127.0.0.1 (Automated)

原因：CIS 要求 kube-controller-manager 的 –bing-address=127.0.0.1

img

img

- 3.[FAIL] 1.4.1 Ensure that the –profiling argument is set to false (Automated)

原因：CIS 要求 kube-scheduler 设置 –profiling=false

img

img

- 4.[FAIL] 1.4.2 Ensure that the –bind-address argument is set to 127.0.0.1 (Automated)

原因：CIS 要求 设置 kube-scheduler 的 –bind-address=127.0.0.1

img

img

加固配置以通过 CIS 扫描

kubespray 官方为了解决这些安全扫描问题，在 v2.22 中添加默认值解决了一部分问题，

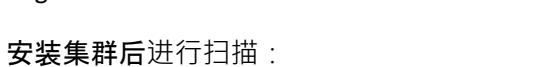
更多细节请参考 [kubespray 加固文档](#)。

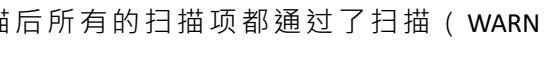
- 通过修改 kubelet var-config 配置文件来添加参数：

```
kubernetes_audit: true  
kube_controller_manager_bind_address: 127.0.0.1  
kube_scheduler_bind_address: 127.0.0.1  
kube_kubeadm_scheduler_extra_args:  
    profiling: false  
kubelet_rotate_server_certificates: true
```

- 在 DCE 5.0 中，也提供了通过 UI 来配置高级参数的功能，在创建集群最后一步添加自定义参数：

- 
- 设置自定义参数后，在 kubelet 的 var-config 的 configmap 中添加了如下参数：

- 
- 安装集群后进行扫描：



扫描后所有的扫描项都通过了扫描（WARN 和 INFO 计算为 PASS），由于 CIS Benchmark 会不断更新，此文档的内容只适用于 CIS Benchmark 1.27。

工作集群离线部署/升级指南

!!! note

本文仅针对离线模式下，使用 DCE 5.0 平台所创建的工作集群的 kubernetes 的版本进行部署或升级，不包括其它 kubenetes 组件的部署或升级。

本文适用以下离线场景：

- 用户可以通过以下操作指南，部署 DCE 5.0 平台所创建的非界面中推荐的 Kubernetes 版本。
- 用户可以通过制作增量离线包的方式对使用 DCE 5.0 平台所创建的工作集群的 kubernetes 的版本进行升级。

整体的思路为：

1. 在联网节点构建离线包
2. 将离线包导入火种节点
3. 更新[全局服务集群](#)的 Kubernetes 版本清单
4. 使用平台 UI 创建工作集群或升级工作集群的 kubernetes 版本

!!! note

目前支持构建的离线 kubernetes 版本，请参考 [kubeant 支持的 kubernetes 版本列表](../../community/kubeant.md#kubernetes)。

在联网节点构建离线包

由于离线环境无法联网，用户需要事先准备一台能够 联网的节点 来进行增量离线包的构建，并且在这个节点上启动 Docker 或者 podman 服务。参阅[如何安装 Docker？](#)

1. 检查联网节点的 Docker 服务运行状态

```
ps aux|grep docker
```

预期输出如下：

```
root      12341  0.5  0.2 654372 26736 ?          Ssl  23:45   0:00 /usr/bin/dockerd
root      12351  0.2  0.1 625080 13740 ?          Ssl  23:45   0:00 docker-containerd --config /var/run/docker/containerd/containerd.toml
root      13024  0.0  0.0 112824   980 pts/0    S+   23:45   0:00 grep --color=auto docker
```

2. 在联网节点的 `/root` 目录下创建一个名为 `manifest.yaml` 的文件，命令如下：

```
vi manifest.yaml
```

`manifest.yaml` 内容如下：

```
'''yaml title="manifest.yaml" image_arch:
```

- “amd64” kube_version: # 填写待升级的集群版本
- “v1.28.0”'''
 - **image_arch** 用于指定 CPU 的架构类型，可填入的参数为 **amd64** 和 **arm64**。

- **kube_version** 用于指定需要构建的 kubernetes 离线包版本，可参考上文的支持构建的离线 kubernetes 版本。

3. 在 **/root** 目录下新建一个名为 **/data** 的文件夹来存储增量离线包。

`mkdir data`

执行如下命令，使用 `kubean airgap-patch` 镜像生成离线包。`airgap-patch` 镜像 tag 与 `Kubean` 版本一致，需确保 `Kubean` 版本覆盖需要升级的 `Kubernetes` 版本。

```
# 假设 kubean 版本为 v0.13.9
docker run --rm -v $(pwd)/manifest.yml:/manifest.yml -v $(pwd)/data:/data ghcr.m.daocloud.io/kubean-io/airgap-patch:v0.13.9
```

等待 Docker 服务运行完成后，检查 **/data** 文件夹下的文件，文件目录如下：

```
data
├── amd64
│   ├── files
│   │   ├── import_files.sh
│   │   └── offline-files.tar.gz
│   ├── images
│   │   ├── import_images.sh
│   │   └── offline-images.tar.gz
│   └── os-pkgs
│       └── import_ospkgs.sh
└── localartifactset.cr.yaml
```

将离线包导入火种节点

1. 将联网节点的 **/data** 文件拷贝至火种节点的 **/root** 目录下，在 联网节点 执行如下命令：

`scp -r data root@x.x.x.x:/root`

x.x.x.x 为火种节点 IP 地址

2. 在火种节点上将 **/data** 文件内的镜像文件拷贝至火种节点内置的 `docker resgitry` 仓库。

登录火种节点后执行如下命令：

1. 进入镜像文件所在的目录

`cd data/amd64/images`

2. 执行 `import_images.sh` 脚本将镜像导入火种节点内置的 Docker Registry 仓库。

```
REGISTRY_ADDR="127.0.0.1" ./import_images.sh
```

!!! note

上述命令仅仅适用于火种节点内置的 Docker Registry 仓库，如果使用外部仓库请使用如下命令：

```
```shell
REGISTRY_SCHEME=https REGISTRY_ADDR=${registry_address} REGISTRY_USER=$
R=${username} REGISTRY_PASS=${password} ./import_images.sh
```

```

- REGISTRY_ADDR 是镜像仓库的地址，比如 1.2.3.4:5000
- 当镜像仓库存在用户名密码验证时，需要设置 REGISTRY_USER 和 REGISTRY_P
ASS

3. 在火种节点上将 `/data` 文件内的二进制文件拷贝至火种节点内置的 Minio 服务上。

1. 进入二进制文件所在的目录

```
cd data/amd64/files/
```

2. 执行 `import_files.sh` 脚本将二进制文件导入火种节点内置的 Minio 服务上。

```
MINIO_USER=rootuser MINIO_PASS=rootpass123 ./import_files.sh http://127.0.0.
1:9000
```

!!! note

上述命令仅仅适用于火种节点内置的 Minio 服务，如果使用外部 Minio 请将 `http://127.0.0.1:9000` 替换为外部 Minio 的访问地址。

“rootuser” 和 “rootpass123”是火种节点内置的 Minio 服务的默认账户和密码。

更新全局服务集群的 kubernetes 版本清单

火种节点上执行如下命令，将 localartifactset 资源部署到全局服务集群：

```
kubectl apply -f data/kubeonofflineversion.cr.patch.yaml
```

下一步

登录 DCE 5.0 的 UI 管理界面，您可以继续执行以下操作：

1. 参照[创建集群的文档](#)进行工作集群创建，此时可以选择 Kubernetes 增量版本。

2. 参照[升级集群的文档](#)对自建的工作集群进行升级。

为工作集群添加异构节点

本文介绍如何为 AMD 架构，操作系统为 CentOS 7.9 的工作集群添加 ARM 架构，操作系统为 Kylin v10 sp2 的工作节点

!!! note

本文仅针对离线模式下，使用 DCE 5.0 平台所创建的工作集群进行异构节点的添加，不包括接入的集群。

前提条件

- 已经部署好一个 DCE 5.0 全模式，并且火种节点还存活，部署参考文档[离线安装 DCE 5.0 商业版](#)
- 已经通过 DCE 5.0 平台创建好一个 AMD 架构，操作系统为 CentOS 7.9 的工作集群，创建参考文档[创建工作集群](#)

操作步骤

下载并导入离线包

以 ARM 架构、操作系统 Kylin v10 sp2 为例。

请确保已经登录到火种节点！并且之前部署 DCE 5.0 时使用的 clusterConfig.yaml 文件还在。

离线镜像包

!!! note

可以在[下载中心](<https://docs.daocloud.io/download/dce5/>)下载最新版本。请确保在容器管理 v0.31 及以上版本使用该能力，对应安装器 v0.21.0 及以上版本

CPU 架构 | 版本 | 下载地址 |

```
:---|:---|:---|:---|  
AMD64           |           v0.21.0           |           <https:  
/qiniu-download-public.daocloud.io/DaoCloud_Enterprise/dce5/offline-v0.21.0-amd64.tar> |  
ARM64           |           v0.21.0           |           <https:  
/qiniu-download-public.daocloud.io/DaoCloud_Enterprise/dce5/offline-v0.21.0-arm64.tar> |
```

下载完毕后解压离线包。此处我们下载 arm64 架构的离线包：

```
tar -xvf offline-v0.21.0-arm64.tar
```

ISO 离线包 (Kylin v10 sp2)

CPU 架构 | 操作系统版本 | 下载地址 |

```
:---|:---|:---|  
ARM64 | Kylin Linux Advanced Server release V10 (Sword) SP2 | 申请地址 : <https://www.kylinos.cn/support/trial.html> |  
!!! note  
麒麟操作系统需要提供个人信息才能下载使用，下载时请选择 V10 (Sword) SP2。
```

osPackage 离线包 (Kylin v10 sp2)

其中 [Kubean](#) 提供了不同操作系统的 osPackage 离线包，可以前往

<https://github.com/kubean-io/kubean/releases> 查看。

操作系统版本 | 下载地址 |

```
:---|:---|:---|  
Kylin   Linux   Advanced   Server   release   V10   (Sword)   SP2   |   <https://github.com/kubean-io/kubean/releases/download/v0.18.5/os-pkgs-kylin-v10sp2-v0.18.5.tar.gz>  
|
```

!!! note

osPackage 离线包的具体对应版本请查看离线镜像包中 `__offline/sample/clusterConfig.yaml__` 中对应的 kubean 版本

导入离线包至火种节点

执行 `import-artifact` 命令：

```
./offline/dce5-installer import-artifact -c clusterConfig.yaml \
--offline-path=/root/offline \
--iso-path=/root/Kylin-Server-10-SP2-aarch64-Release-Build09-20210524.iso \
--os-pkgs-path=/root/os-pkgs-kylin-v10sp2-v0.18.5.tar.gz
```

!!! note

参数说明：

- `--c clusterConfig.yaml` 指定之前部署 DCE5.0 时使用的 `clusterConfig.yaml` 文件
- `--offline-path` 指定下载的离线镜像包文件地址
- `--iso-path` 指定下载的 ISO 操作系统镜像文件地址
- `--os-pkgs-path` 指定下载的 `osPackage` 离线包文件地址

导入命令执行成功后，会将离线包上传到火种节点的 Minio 中。

添加异构工作节点

!!! note

如果您安装的 DCE 5.0 版本高于（包含）[DCE5.0-20230731](../../dce/dce-rn/20230731.md)，完成以上步骤后，您可以直接在界面中接入节点；反之，您需要继续执行以下步骤来接入异构节点。

请确保已经登录到 DCE 5.0 [管理集群](#)的管理节点上。

修改主机清单文件

主机清单文件示例：

```
==== “新增节点前”

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
 name: ${cluster-name}-hosts-conf
 namespace: kubelet-system
data:
 hosts.yml: |
 all:
```

```

children:
 etcd:
 hosts:
 centos-master:
 k8s_cluster:
 children:
 kube_control_plane:
 kube_node:
 kube_control_plane:
 hosts:
 centos-master:
 kube_node:
 hosts:
 centos-master:
 hosts:
 centos-master:
 ip: 10.5.10.183
 access_ip: 10.5.10.183
 ansible_host: 10.5.10.183
 ansible_connection: ssh
 ansible_user: root
 ansible_ssh_pass: *****
 ansible_password: *****
 ansible_become_password: *****
```

```

==== “新增节点后”

```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
 name: ${cluster-name}-hosts-conf
 namespace: kube-system
data:
 hosts.yml: |
 all:
 hosts:
 centos-master:
 ip: 10.5.10.183
 access_ip: 10.5.10.183
 ansible_host: 10.5.10.183
 ansible_connection: ssh
 ansible_user: root
 ansible_ssh_pass: *****

```

```

ansible_password: *****
ansible_become_password: *****
添加异构节点信息
kylin-worker:
 ip: 10.5.10.181
 access_ip: 10.5.10.181
 ansible_host: 10.5.10.181
 ansible_connection: ssh
 ansible_user: root
 ansible_ssh_pass: *****
 ansible_password: *****
 ansible_become_password: *****
children:
 kube_control_plane:
 hosts:
 - centos-master
 kube_node:
 hosts:
 - centos-master
 - kylin-worker # 添加新增的异构节点名称
 etcd:
 hosts:
 - centos-master
 k8s_cluster:
 children:
 - kube_control_plane
 - kube_node
```

```

按照上述的配置注释，添加新增的工作节点信息。

```
kubectl edit cm ${cluster-name}-hosts-conf -n kube-system
```

cluster-name 为工作集群的名称，通过容器管理创建集群时会默认生成。

通过 ClusterOperation.yml 新增扩容任务

示例：

```

yaml title="ClusterOperation.yml" apiVersion: kubeany.io/v1alpha1 kind: ClusterOperation
metadata: name: add-worker-node spec: cluster: ${cluster-name} # 指定 cluster name
image: 10.5.14.30/ghcr.m.daocloud.io/kubeany-io/spray-job:v0.18.5 actionType: playbook
action: scale.yml extraArgs: --limit=kylin-worker preHook: - actionType: playbook

```

```

action: ping.yml      - actionType: playbook           action: disable-firewalld.yml      -
actionType: playbook      action: enable-repo.yml       extraArgs: |                  -e
"repo_list:
["http://10.5.14.30:9000/kubeany/kylin-iso/\$releasever/sp2/os/\$basearch","http://10.5.14.30:9
00/kubeany/kylin/\$releasever/sp2/os/\$basearch"]}" --limit=kylin-worker   postHook:      -
actionType: playbook      action: cluster-info.yml
!!! note
- spec.image 镜像地址要与之前执行部署时的 job 其内镜像保持一致
- spec.action 设置为 scale.yml
- spec.extraArgs 设置为 --limit=g-worker
- spec.preHook 中的 enable-repo.yml 剧本参数，要填写相关 OS 的正确的 repo_list

```

按照上述的配置，创建并部署 join-node-ops.yaml：

```

vi join-node-ops.yaml
kubectl apply -f join-node-ops.yaml -n kubeany-system

```

检查任务执行状态

```
kubectl -n kubeany-system get pod | grep add-worker-node
```

了解缩容任务执行进度，可查看该 Pod 日志。

前往界面验证

1. 前往 容器管理 -> 集群 -> 节点管理

| 名称 | 状态 | 角色 | GPU 模式 | 标签 | CPU (分配/使用率) | 内存 (分配/使用率) | IP 地址 | 创建时间 |
|-------------------|----|------|--------|-----------------------------|--------------|-------------|-------------|------------------|
| kylin-worker | 健康 | 工作节点 | - | beta.kubernetes.io/os:linux | 0% / 0% | 0% / 0% | 10.5.10.181 | 2024-09-24 20:04 |
| controller-node-1 | 健康 | 控制台 | - | beta.kubernetes.io/os:linux | 0% / 0% | 0% / 0% | 10.5.10.183 | 2024-09-24 16:22 |

节点管理

2. 点击新增的节点，查看详情

The screenshot shows the 'worker-cluster' section of the DaoCloud Enterprise 5.0 dashboard. On the left is a sidebar with various management options. The main area displays detailed information about a specific node named 'kylin-worker'. It shows the node's status as healthy and its role as a worker node. Resource usage metrics are provided for CPU, memory, and disk. There are also charts showing the distribution of resources across different components.

节点详情

对工作集群的控制节点扩容

本文将以一个单控制节点的工作集群为例，介绍如何手动为工作集群的控制节点进行扩容，以实现自建工作集群的高可用。

!!! note

- 推荐在界面创建工作集群时即开启高可用模式，手动扩容工作集群的控制节点存在一定的操作风险，请谨慎操作。
- 当工作集群的首个控制节点故障或异常时，如果您想替换或重新接入首个控制节点，请参考[替换工作集群的首个控制节点](./best-practice/replace-first-master-node.md)

前提条件

- 已经通过 DCE 5.0 平台创建好一个工作集群，可参考文档[创建工作集群](#)。
- 工作集群的被纳管集群存在当前平台中，并且状态运行正常。

!!! note

被纳管集群：在界面创建集群时指定的用来管理当前集群，并为当前集群提供 kubernetes 版本升级、节点扩缩容、卸载、操作记录等能力的集群。

修改主机清单文件

1. 登录到容器管理平台，进入需要进行控制节点扩容的集群概览页面，在 **基本信息** 处，

找到当前集群的 **被纳管集群**，点击被纳管集群的名称，进入被纳管集群的概览

界面。

找到被纳管集群

找到被纳管集群

2. 在被纳管集群的概览界面，点击 **控制台**，打开云终端控制台，并执行如下命令，找到

待扩容工作集群的主机清单文件。

```
kubectl get cm -n kube-system ${ClusterName}-hosts-conf -oyaml
```

`${ClusterName}`：为待扩容工作集群的名称。

控制台

控制台

3. 参考下方示例修改主机清单文件，新增控制节点信息。

==== “修改前”

```
'''yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: tanghai-dev-hosts-conf
  namespace: kube-system
data:
  hosts.yml: |
    all:
      hosts:
        node1:
          ip: 10.6.175.10
          access_ip: 10.6.175.10
          ansible_host: 10.6.175.10
          ansible_connection: ssh
          ansible_user: root
          ansible_password: password01
    children:
      kube_control_plane:
        hosts:
          node1:
```

```

kube_node:
  hosts:
    node1:
      etcd:
        hosts:
          node1:
            k8s_cluster:
              children:
                kube_control_plane:
                  kube_node:
                    calico_rr:
                      hosts: {}
.....
```

```

==== “修改后”

```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: tanghai-dev-hosts-conf
  namespace: kube-system
data:
  hosts.yml: |
    all:
      hosts:
        node1: # 原集群中已存在的主节点
          ip: 10.6.175.10
          access_ip: 10.6.175.10
          ansible_host: 10.6.175.10
          ansible_connection: ssh
          ansible_user: root
          ansible_password: password01
        node2: # 集群扩容待新增的控制节点
          ip: 10.6.175.20
          access_ip: 10.6.175.20
          ansible_host: 10.6.175.20
          ansible_connection: ssh
          ansible_user: root
          ansible_password: password01
        node3: # 集群扩容待新增的控制节点
          ip: 10.6.175.30
          access_ip: 10.6.175.30
          ansible_host: 10.6.175.30

```

```

ansible_connection: ssh
ansible_user: root
ansible_password: password01
children:
  kube_control_plane:
    hosts: # 集群中的控制节点组
      node1:
      node2: # 新增控制节点 node2 内容
      node3: # 新增控制节点 node3 内容
  kube_node:
    hosts: # 集群中的工作节点组
      node1:
      node2: # 新增控制节点 node2 内容
      node3: # 新增控制节点 node3 内容
  etcd:
    hosts: # 集群中的 ETCD 节点组
      node1:
      node2: # 新增控制节点 node2 内容
      node3: # 新增控制节点 node3 内容
  k8s_cluster:
    children:
      kube_control_plane:
      kube_node:
      calico_rr:
        hosts: {}
```

```

## 新增 ClusterOperation.yml 扩容任务

使用基于下面的 `ClusterOperation.yml` 模板，新增一个集群控制节点扩容任务

`scale-master-node-ops.yaml`。

```

yaml title="ClusterOperation.yml" apiVersion: kubeanywhere/v1alpha1 kind: ClusterOperation
metadata: name: cluster1-online-install-ops spec: cluster: ${cluster-name} # (1)!
image: ghcr.m.daocloud.io/kubeanywhere-spray-job:v0.18.0 # (2)! actionType: playbook action:
cluster.yml # (3)! extraArgs: --limit=etcd,kube_control_plane -e ignore_assert_errors=yes
preHook: - actionType: playbook action: ping.yml - actionType: playbook
action: disable-firewalld.yml - actionType: playbook action: enable-repo.yml # (4)!

extraArgs: | # 如果是离线环境，需要添加 enable-repo.yml，并且 extraArgs 参数填写相关

```

OS 的 正 确 repo\_list -e "{repo\_list":

[`http://172.30.41.0:9000/kubeanywhere/centos/\\$releasever/os/\\$basearch`,'http://172.30.41.0:9000/`]

```
kubean/centos-iso/\$releasever/os/\$basearch']}" postHook: - actionType: playbook
action: upgrade-cluster.yml extraArgs: --limit=etcd,kube_control_plane -e
ignore_assert_errors=yes - actionType: playbook action: kubeconfig.yml -
actionType: playbook action: cluster-info.yml
```

1. 指定 cluster name

2. 指定 kubean 任务运行的镜像，镜像地址要与之前执行部署时的 job 其内镜像保持一

致

3. 如果一次性添加 Master ( etcd ) 节点超过 ( 包含 ) 三个，需在 cluster.yaml 追加额外参数 -e etcd\_retries=10 以增大 etcd node join 重试次数

4. 离线环境下需要添加此 yaml，并且设置正确的 repo-list ( 安装操作系统软件包 )，以下参数值仅供参考

然后创建并部署 scale-master-node-ops.yaml。

```
vi scale-master-node-ops.yaml
kubectl apply -f scale-master-node-ops.yaml -n kube-system
```

执行完上述步骤，执行如下命令进行验证：

```
kubectl get node
```

## 替换工作集群的首个控制节点

本文将以一个高可用三控制节点的工作集群为例。当工作集群的首个控制节点故障或异常时，如何替换或重新接入首个控制节点。

本文的高可用集群有 3 个 Master 节点：

- node1 (172.30.41.161)
- node2 (172.30.41.162)
- node3 (172.30.41.163)

假设 node1 崩机，接下来介绍如何将宕机后恢复的 node1 重新接入工作集群。

## 准备工作

在执行替换操作之前，先获取集群资源基本信息，修改相关配置时会用到。

### !!! note

以下获取集群资源信息的命令均在管理集群中执行。

#### 1. 获取集群名称

执行如下命令，找到集群对应的 clusters.kubebean.io 资源：

```
比如 clusters.kubebean.io 的资源名称为 cluster-mini-1
则获取集群的名称
CLUSTER_NAME=$(kubectl get clusters.kubebean.io cluster-mini-1 -o=jsonpath=".meta
ta.name{\n}")
```

#### 2. 获取集群的主机清单 configmap

```
kubectl get clusters.kubebean.io cluster-mini-1 -o=jsonpath=".spec.hostsConfRef{\n}"
{"name":"mini-1-hosts-conf","namespace":"kubebean-system"}
```

#### 3. 获取集群的配置参数 configmap

```
kubectl get clusters.kubebean.io cluster-mini-1 -o=jsonpath=".spec.varsConfRef{\n}"
 {"name":"mini-1-vars-conf","namespace":"kubebean-system"}
```

## 操作步骤

#### 1. 调整控制平面节点顺序

重置 node1 节点使其恢复到安装集群之前的状态（或使用新的节点），保持 node1 节点的网络连通性。

调整主机清单中 node1 节点在 kube\_control\_plane、kube\_node、etcd 中的顺序

( node1/node2/node3 -> node2/node3/node1 ) :

```
function change_control_plane_order() {
 cat << EOF | kubectl apply -f -

 apiVersion: v1
 kind: ConfigMap
 metadata:
 name: mini-1-hosts-conf
 namespace: kubebean-system
```

```
data:
 hosts.yml: |
 all:
 hosts:
 node1:
 ip: "172.30.41.161"
 access_ip: "172.30.41.161"
 ansible_host: "172.30.41.161"
 ansible_connection: ssh
 ansible_user: root
 ansible_password: dangerous
 node2:
 ip: "172.30.41.162"
 access_ip: "172.30.41.162"
 ansible_host: "172.30.41.162"
 ansible_connection: ssh
 ansible_user: root
 ansible_password: dangerous
 node3:
 ip: "172.30.41.163"
 access_ip: "172.30.41.163"
 ansible_host: "172.30.41.163"
 ansible_connection: ssh
 ansible_user: root
 ansible_password: dangerous
 children:
 kube_control_plane:
 hosts:
 node2:
 node3:
 node1:
 kube_node:
 hosts:
 node2:
 node3:
 node1:
 etcd:
 hosts:
 node2:
 node3:
 node1:
 k8s_cluster:
 children:
 kube_control_plane:
```

```

 kube_node:
 calico_rr:
 hosts: {}
EOF
}

```

change\_control\_plane\_order

## 2. 移除异常状态的首个 master 节点

调整主机清单的节点顺序后，移除 K8s 控制平面异常状态的 node1。

### !!! note

如果 node1 离线或故障，则 extraArgs 须添加以下配置项，node1 在线时不需要添加。

```

```toml
reset_nodes=false # 跳过重置节点操作
allow_ungraceful_removal=true # 允许非优雅的移除操作
```
镜像 spray-job 这里可以采用加速器地址

SPRAY_IMG_ADDR="ghcr.m.daocloud.io/kubeanywhere/spray-job"
SPRAY_RLS_2_22_TAG="2.22-336b323"
KUBE_VERSION="v1.24.14"
CLUSTER_NAME="cluster-mini-1"
REMOVE_NODE_NAME="node1"

cat << EOF | kubectl apply -f -

apiVersion: kubeanywhere/v1alpha1
kind: ClusterOperation
metadata:
 name: cluster-mini-1-remove-node-ops
spec:
 cluster: ${CLUSTER_NAME}
 image: ${SPRAY_IMG_ADDR}:${SPRAY_RLS_2_22_TAG}
 actionType: playbook
 action: remove-node.yml
 extraArgs: -e node=${REMOVE_NODE_NAME} -e reset_nodes=false -e allow_ungraceful_removal=true -e kube_version=${KUBE_VERSION}
 postHook:
 - actionType: playbook
 action: cluster-info.yml
EOF

```

## 3. 手动修改集群配置，编辑更新 cluster-info

```
编辑 cluster-info
kubectl -n kube-public edit cm cluster-info

1. 若 ca.crt 证书更新, 则需要更新 certificate-authority-data 字段的内容
查看 ca 证书的 base64 编码:
cat /etc/kubernetes/ssl/ca.crt | base64 | tr -d '\n'

2. 需改 server 字段的 IP 地址为新 first master IP, 本文档场景将使用 node2 的 IP 地址 172.30.41.162
```

#### 4. 手动修改集群配置, 编辑更新 kubeadm-config

```
编辑 kubeadm-config
kubectl -n kube-system edit cm kubeadm-config

修改 controlPlaneEndpoint 为新 first master IP, 本文档场景将使用 node2 的 IP 地址 172.30.41.162
```

#### 5. 重新扩容 master 节点并更新集群

##### !!! note

- 使用 `--limit` 限制更新操作仅作用于 etcd 和 kube\_control\_plane 节点组。
- 如果是离线环境, spec.preHook 需要添加 enable-repo.yml, 并且 extraArgs 参数填写相关 OS 的正确 repo\_list。
- 扩容完成后, node2 变更为首个 master

```
cat << EOF | kubectl apply -f -

apiVersion: kubelet.io/v1alpha1
kind: ClusterOperation
metadata:
 name: cluster-mini-1-update-cluster-ops
spec:
 cluster: ${CLUSTER_NAME}
 image: ${SPRAY_IMG_ADDR}:${SPRAY_RLS_2_22_TAG}
 actionType: playbook
 action: cluster.yml
 extraArgs: --limit=etcd,kube_control_plane -e kube_version=${KUBE_VERSION}
 preHook:
 - actionType: playbook
 action: enable-repo.yml # 离线环境下需要添加此 yaml, 并且设置正确的 repo-list(安装操作系统软件包), 以下参数值仅供参考
 extraArgs: |
 -e "{repo_list: ['http://172.30.41.0:9000/kubelet/centos/\$releasever/os/\$basearch','http://172.30.41.0:9000/kubelet/centos-iso/\$releasever/os/\$basearch']}"
 postHook:
 - actionType: playbook
```

```
action: cluster-info.yml
EOF
```

至此，完成了首个 Master 节点的替换。

## 为全局服务集群的工作节点扩容

本文将介绍离线模式下，如何手动为全局服务集群的工作节点进行扩容。默认情况下，不建议在部署 DCE 5.0 后对[全局服务集群](#)进行扩容，请在部署 DCE 5.0 前做好资源规划。

!!! note

全局服务集群的控制节点不支持扩容。

### 前提条件

- 已经通过[火种节点](#)完成 DCE 5.0 平台的部署，并且火种节点上的 kind 集群运行正常。
- 必须使用平台 Admin 权限的用户登录。

### 获取火种节点上 kind 集群的 kubeconfig

1. 执行如下命令登录火种节点：

```
ssh root@火种节点 IP 地址
```

2. 在火种节点上执行如下命令获取 kind 集群的 CONTAINER ID：

```
[root@localhost ~]# podman ps
```

# 预期输出如下:

| CONTAINER ID                        | IMAGE                                                                     | COMMAND     |
|-------------------------------------|---------------------------------------------------------------------------|-------------|
| CREATED                             | STATUS                                                                    | PORTS       |
|                                     |                                                                           | NAMES       |
| 220d662b1b6a                        | docker.m.daocloud.io/kindest/node:v1.26.2                                 | 2 weeks ago |
| Up 2 weeks                          | 0.0.0.0:443->30443/tcp, 0.0.0.0:8081->30081/tcp, 0.0.0.0:9000-9001->32000 |             |
| -32001/tcp, 0.0.0.0:36674->6443/tcp | my-cluster-installer-control-plane                                        |             |

3. 执行如下命令，进入 kind 集群容器内：

```
podman exec -it {CONTAINER ID} bash
```

{CONTAINER ID} 替换为您真实的容器 ID

4. 在 kind 集群容器内执行如下命令获取 kind 集群的 kubeconfig 配置信息：

```
kubectl config view --minify --flatten --raw
```

待控制台输出后，复制 kind 集群的 kubeconfig 配置信息，为下一步做准备。

## 在火种节点上 kind 集群内创建 cluster.kubebean.io 资源

1. 使用 podman exec -it {CONTAINER ID} bash 命令进入 kind 集群容器内。

2. 在 kind 集群容器内，执行如下命令，获取 kind 集群名称：

```
kubectl get clusters
```

3. 复制并执行如下命令，在 kind 集群内执行，以创建 cluster.kubebean.io 资源：

```
kubectl apply -f - <<EOF
apiVersion: kubebean.io/v1alpha1
kind: Cluster
metadata:
 labels:
 clusterName: kpanda-global-cluster
 name: kpanda-global-cluster
spec:
 hostsConfRef:
 name: my-cluster-hosts-conf
 namespace: kubebean-system
 kubeconfRef:
 name: my-cluster-kubeconf
 namespace: kubebean-system
 varsConfRef:
 name: my-cluster-vars-conf
 namespace: kubebean-system
EOF
```

!!! note

`spec.hostsConfRef.name`、`spec.kubeconfRef.name`、`spec.varsConfRef.name` 中集群名称默认为 `my-cluster`，需替换成上一步骤中获取的 \*\*kind 集群名称\*\*。

4. 在 kind 集群内执行如下命令，检验 cluster.kubebean.io` 资源是否正常创建：

```
kubectl get clusters
```

预期输出如下：

| NAME                  | AGE |
|-----------------------|-----|
| kpanda-global-cluster | 3s  |
| my-cluster            | 16d |

## 更新火种节点上的 kind 集群里的 containerd 配置

1. 执行如下命令，登录全局服务集群的其中一个控制节点：

```
ssh root@全局服务集群控制节点 IP 地址
```

2. 在全局服务集群控制节点上执行如下命令，将控制节点的 containerd 配置文件

**config.toml** 复制到火种节点上：

```
scp /etc/containerd/config.toml root@{火种节点 IP}:root
```

3. 在火种节点上，从控制节点拷贝过来的 containerd 配置文件 **config.toml** 中选取 非

**安全镜像 registry 的部分** 加入到 kind 集群内 config.toml

非安全镜像 registry 部分示例如下：

```
[plugins."io.containerd.grpc.v1.cri".registry]
[plugins."io.containerd.grpc.v1.cri".registry.mirrors]
[plugins."io.containerd.grpc.v1.cri".registry.mirrors."10.6.202.20"]
 endpoint = ["https://10.6.202.20"]
[plugins."io.containerd.grpc.v1.cri".registry.configs."10.6.202.20".tls]
 insecure_skip_verify = true
```

!!! note

由于 kind 集群内不能直接修改 config.toml 文件，故可以先复制一份文件出来修改，再拷贝到 kind 集群，步骤如下：

1. 在火种节点上执行以下命令，将文件拷贝出来

```
```bash
podman cp {CONTAINER ID}:/etc/containerd/config.toml ./config.toml.kind
```

```

1. 执行如下命令编辑 config.toml 文件

```
```bash
vim ./config.toml.kind
```

```

- 将修改好的文件再复制到 kind 集群，执行如下命令

```
```bash
podman cp ./config.toml.kind {CONTAINER ID}:/etc/containerd/config.toml
```

```

\*\*{CONTAINER ID}\*\* 替换为您真实的容器 ID

- 在 kind 集群内执行如下命令，重启 containerd 服务

```
systemctl restart containerd
```

## 将 kind 集群接入 DCE 5.0 集群列表

- 登录 DCE 5.0，进入容器管理，在集群列表页右侧点击 **接入集群** 按钮，进入接入集群页面。

- 在接入配置处，填入并编辑刚刚复制的 kind 集群的 kubeconfig 配置。

```
apiVersion: v1
clusters:
- cluster:
 insecure-skip-tls-verify: true # (1)
 certificate-authority-data: LS0TLSCFDWEFEWFGEWGFWEWGWEFGFEW
 GEWGSDGFSDS
 server: https://my-cluster-installer-control-plane:6443 # (2)!
 name: my-cluster-installer
contexts:
- context:
 cluster: my-cluster-installer
 user: kubernetes-admin
 name: kubernetes-admin@my-cluster-installer
 current-context: kubernetes-admin@my-cluster-installer
kind: Config
preferences: {}
users:
```

- 跳过 tls 验证，这一行需要手动添加

- 替换为火种节点的 IP，端口 6443 替换为在节点映射的端口（你可以执行

`podman ps|grep 6443` 命令查看映射的端口）

`kubeconfig`

kubeconfig

3. 点击 **确认** 按钮，完成 kind 集群的接入。

## 为全局服务集群添加标签

1. 登录 DCE 5.0，进入容器管理，找到 **kpanda-global-cluster** 集群，在右侧操作列表找

到 **基础配置** 菜单项并进入基础配置界面。

2. 在基础配置页面，为全局服务集群添加的标签 `kpanda.io/managed-by=my-cluster`：

!!! note

标签 `kpanda.io/managed-by=my-cluster` 中的 value 值为接入集群时指定的集群名称，默认为 `my-cluster`，具体依据您的实际情况。

![标签](https://docs.daocloud.io/daocloud-docs-images/docs/zh/docs/kpanda/images/add-global-node02.png)

## 为全局服务集群添加节点

1. 进入全局服务集群节点列表页，点击右侧的 **接入节点** 按钮。

2. 填入待接入节点的 IP 和认证信息后点击 **开始检查**，通过节点检查后点击 **下一步**。

3. 在 **自定义参数** 处添加如下自定义参数：

```
download_run_once: false
download_container: false
download_force_cache: false
download_localhost: false
```

img

img

4. 点击 **确定** 等待节点添加完成。

# 在 CentOS 管理平台上创建 Ubuntu 工作集群

本文介绍如何在已有的 CentOS 管理平台上创建 Ubuntu 工作集群。

!!! note

本文仅针对离线模式下，使用 DCE 5.0 平台创建工作集群，管理平台和待建工作集群的架构均为 AMD。

创建集群时不支持异构（AMD 和 ARM 混合）部署，您可以在集群创建完成后，通过接入异构节点的方式进行集群混合部署管理。

## 前提条件

- 已经部署好一个 DCE 5.0 全模式，并且火种节点还存活，部署参考文档[离线安装 DCE 5.0 商业版](#)

## 下载并导入 Ubuntu 相关离线包

请确保已经登录到火种节点！并且之前部署 DCE 5.0 时使用的 clusterConfig.yaml 文件还在。

### 下载 Ubuntu 相关离线包

下载所需的 Ubuntu OS package 包和 ISO 离线包：

| 资源名                                   | 说明                         | 下载地址                                                                                                                                                                                                              |
|---------------------------------------|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| os-pkgs-ubuntu2204-v0.18.2.t<br>ar.gz | Ubuntu1804<br>OS-package 包 | <a href="https://github.com/kubeain-io/kubeain/releases/download/v0.18.2/os-pkgs-ubuntu2204-v0.18.2.tar.gz">https://github.com/kubeain-io/kubeain/releases/download/v0.18.2/os-pkgs-ubuntu2204-v0.18.2.tar.gz</a> |
| ISO 离线包                               | ISO 包                      | <a href="http://mirrors.melbourne.co.uk/ubuntu-releases/">http://mirrors.melbourne.co.uk/ubuntu-releases/</a>                                                                                                     |

## 导入 OS Package 和 ISO 离线包至火种节点的 MinIO

参考文档[离线资源导入](#)，导入离线资源至火种节点的 MinIO。

## 前往 UI 界面创建集群

参考文档[创建工作集群](#)，创建 Ubuntu 集群。

# 在 CentOS 管理平台上创建 RedHat 9.2 工作集群

本文介绍如何在已有的 CentOS 管理平台上创建 RedHat 9.2 工作集群。

!!! note

本文仅针对离线模式下，使用 DCE 5.0 平台创建工作集群，管理平台和待建工作集群的架构均为 AMD。

创建集群时不支持异构（AMD 和 ARM 混合）部署，您可以在集群创建完成后，通过接入异构节点的方式进行集群混合部署管理。

## 前提条件

你已经部署好一个 DCE 5.0 全模式，并且火种节点还存活。具体部署，请参考文档[离线安装 DCE 5.0 商业版](#)。

## 下载并导入 RedHat 相关离线包

请确保已经登录到火种节点！并且之前部署 DCE 5.0 时使用的 clusterConfig.yaml 文件还在。

## 下载 RedHat 相关离线包

下载所需的 RedHat OS package 包和 ISO 离线包：

| 资源名                           | 说明                        | 下载地址                                                                                                                                                                                                |
|-------------------------------|---------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| os-pkgs-redhat9-v0.9.3.tar.gz | RedHat9.2<br>OS-package 包 | <a href="https://github.com/kubean-io/kubean/releases/download/v0.9.3/os-pkgs-redhat9-v0.9.3.tar.gz">https://github.com/kubean-io/kubean/releases/download/v0.9.3/os-pkgs-redhat9-v0.9.3.tar.gz</a> |
| ISO 离线包                       | ISO 包导入火种                 | 前往 <a href="#">RedHat 官方地址登录下载</a>                                                                                                                                                                  |
|                               | 节点脚本                      |                                                                                                                                                                                                     |
| import-iso                    | ISO 导入火种节点脚本              | <a href="https://github.com/kubean-io/kubean/releases/download/v0.9.3/import_iso.sh">https://github.com/kubean-io/kubean/releases/download/v0.9.3/import_iso.sh</a>                                 |

## 导入 OS package 离线包至火种节点

执行如下命令，导入 os package 包：

```
采用 build-in 内建模式部署火种集群时，我们可以不用指定 clusterConfig.yml 配置文件
dce5-installer import-artifact --os-pkgs-path=/home/os-pkgs/os-pkgs-redhat9-v0.9.3.tar.gz

采用 external 外接模式部署火种集群时，我们需要指定 clusterConfig.yml 配置文件
dce5-installer import-artifact -c clusterConfig.yml --os-pkgs-path=/home/os-pkgs/os-pkgs-redhat9-v0.9.3.tar.gz
```

!!! note

上述命令中，“/home/os-pkgs”为 os package 包下载目录，“os-pkgs-redhat9-v0.9.3.tar.gz“为所下载的 os package 离线包名称。

## 导入 ISO 离线包至火种节点

执行如下命令，导入 ISO 包：

```
采用 build-in 内建模式部署火种集群时，我们可以不用指定 clusterConfig.yml 配置文件
dce5-installer import-artifact --iso-path=/home/iso/rhel-9.2-x86_64-dvd.iso

采用 external 外接模式部署火种集群时，我们需要指定 clusterConfig.yml 配置文件
dce5-installer import-artifact -c clusterConfig.yml --iso-path=/home/iso/rhel-9.2-x86_64-dvd.iso
```

**!!! note**

上述命令中，“/home/iso”为 ISO 包下载目录，“rhel-9.2-x86\_64-dvd.iso“ 为所下载的 ISO 离线包名称。

## 前往 UI 界面创建集群

参考文档[创建工作集群](#)，创建 RedHat 9.2 集群。

## 在非主流操作系统上创建集群

本文介绍离线模式下如何在 **未声明支持的 OS** 上创建工作集群。DCE 5.0 声明支持的 OS

范围请参考 [DCE 5.0 支持的操作系统](#)

离线模式下在未声明支持的 OS 上创建工作集群，主要的流程如下图：

流程

流程

接下来，本文将以 openAnolis 操作系统为例，介绍如何在非主流操作系统上创建集群。

## 前提条件

- 已经部署好一个 DCE 5.0 全模式，部署参考文档[离线安装 DCE 5.0 商业版](#)
- 至少拥有一台可以联网的同架构同版本的节点。

## 在线节点构建离线包

找到一个和待建集群节点架构和 OS 均一致的在线环境，本文以 [AnolisOS 8.8 GA](#) 为例。

执行如下命令，生成离线 os-pkgs 包。

```
下载相关脚本并构建 os packages 包
curl -Lo ./pkgs.yml https://raw.githubusercontent.com/kubean-io/kubean/main/build/os-packages/others/pkgs.yml
```

```
curl -Lo ./other_os_pkgs.sh https://raw.githubusercontent.com/kubean-io/kubean/main/build/os-pkgs/others/other_os_pkgs.sh && chmod +x other_os_pkgs.sh
./other_os_pkgs.sh build # 构建离线包
```

执行完上述命令后，预期将在当前路径下生成一个名为 **os-pkgs-anolis-8.8.tar.gz** 的压缩包。

当前路径下文件目录大概如下：

```
.
├── other_os_pkgs.sh
├── pkgs.yml
└── os-pkgs-anolis-8.8.tar.gz
```

## 离线节点安装离线包

将在线节点中生成的 **other\_os\_pkgs.sh**、**pkgs.yml**、**os-pkgs-anolis-8.8.tar.gz** 三个文件

拷贝至离线环境中的待建集群的**所有**节点上。

登录离线环境中，任一待建集群的其中一个节点上，执行如下命令，为节点安装 **os-pkg** 包。

```
配置环境变量
export PKGS_YML_PATH=/root/workspace/os-pkgs/pkgs.yml # 当前离线节点 pkgs.yml 文件的
路径
export PKGS_TAR_PATH=/root/workspace/os-pkgs/os-pkgs-anolis-8.8.tar.gz # 当前离线节点 os-
pkgs-anolis-8.8.tar.gz 的路径
export SSH_USER=root # 当前离线节点的用户名
export SSH_PASS=dangerous # 当前离线节点的密码
export HOST_IPS='172.30.41.168' # 当前离线节点的 IP
./other_os_pkgs.sh install # 安装离线包
```

执行完成上述命令后，等待界面提示：**All packages for node (X.X.X.X) have been installed** 即表示安装完成。

## 下一步

参考文档[创建工作集群](#)，在 UI 界面上创建 openAnolis 集群。

# DCE 4.0 -> DCE 5.0 有限场景迁移

本文介绍部分场景下 DCE 4.0 -> DCE 5.0 迁移流程。

## 环境准备

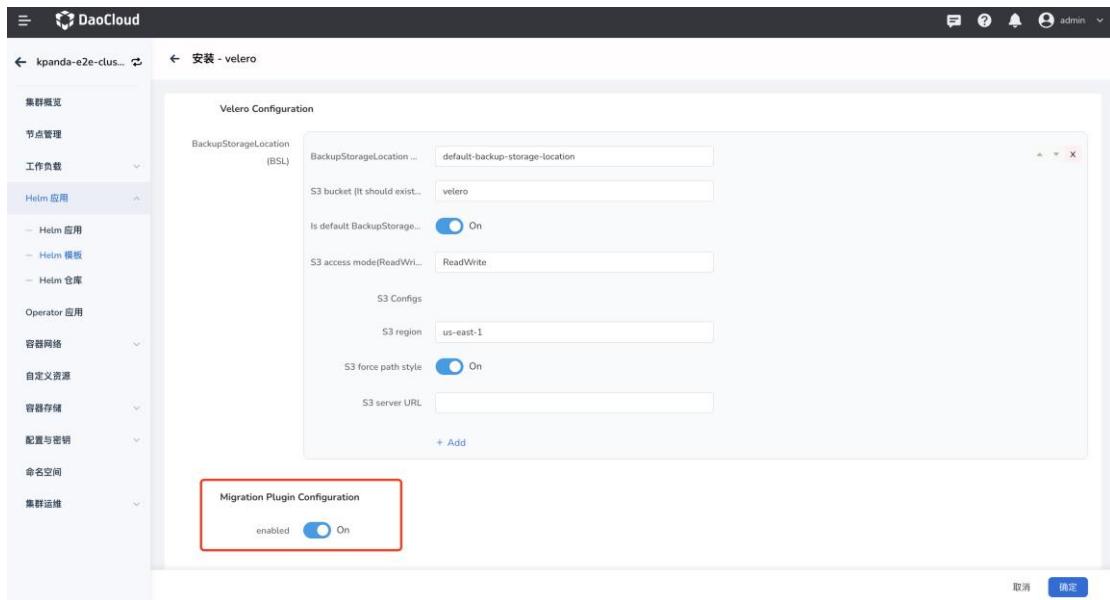
1. 可用的 DCE 4.0 环境
2. 可用的 DCE 5.0 环境
3. 可用于数据还原的 Kubernetes 集群，以下简称 还原集群

## 前置步骤

1. 在 DCE 4.0 上，安装 CoreDNS 插件。
2. 将 DCE 4.0 纳管到 DCE 5.0，纳管步骤参考[接入 DCE 4.0](#)，被纳管到 DCE 5.0 的 DCE 4.0 集群，以下简称 备份集群。  
!!! note  
接入集群时，发行版选择 \_\_DaoCloud DCE4\_\_。
3. 在纳管的 DCE 4.0 集群上安装 velero，安装步骤参考[安装 velero](#)。
4. 将还原集群纳管到 DCE 5.0，通过创建集群方式或接入方式都可。
5. 在还原集群中安装 velero，安装步骤参考[安装 velero](#)。

### !!! note

- 被纳管的 DCE 4.0 集群和还原集群中安装的 velero，对象存储配置必须保持一致。
- 如果您需要进行 Pod 迁移，请将表单参数中的 \_\_Migration Plugin Configuration\_\_ 开关打开（\*\*velero 5.2.0+\*\* 版本支持此配置）。



安装 plugin

## 可选配置

如果您需要进行 Pod 迁移 , 请在前置步骤完成后执行以下步骤 , 非 Pod 迁移场景可以忽略。

### !!! note

以下步骤都在被 DCE 5.0 纳管的还原集群中执行。

## 配置 Velero 插件

1. velero 插件安装完成 , 可以执行以下 yaml 文件完成 velero 插件的配置。

### !!! note

安装 velero 插件时 , 必须将表单参数中的 Migration Plugin Configuration 开关打开

◦

??? note “点击查看完整的 YAML 示例”

```
```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: velero-plugin-for-migration # (1)!
  namespace: velero # (2)!
  labels: # (3)!
```

```

velero.io/plugin-config: "velero-plugin-for-migration" # (4)!

velero.io/velero-plugin-for-migration: RestoreItemAction # (5)!

data:

  velero-plugin-for-migration: '{"resourcesSelector":{"includedNamespaces":["kube-system"],"excludedNamespaces":["default"],"includedResources":["pods","deployments","ingress"],"excludedResources":["secrets"],"skipRestoreKinds":["endpointslice"],"labelSelector":"app:dao-2048"},"resourcesConverter":[{"ingress":{"enabled":true,"apiVersion":"extensions/v1beta1"}],"resourcesOperation":[{"kinds":["pod"],"domain":"labels","operation":{"add":{"key1":"values","key2":""}, "remove":{"key3":"values","key4":""}, "replace":{"key5":["source","dest"]}}, {"kinds":["deployment","daemonset"],"domain":{"annotations"}, "scope":"resourceSpec","operation":{"add":{"key1":"values","key2":""}, "remove":{"key3":"values","key4":""}, "replace":{"key5":["source","dest"], "key6:[],"key7:[source,""]}}}]}'`~~~`
```

1. any name can be used; Velero uses the labels (below) to identify it rather than the name
2. must be in the velero namespace
3. the below labels should be used verbatim in your ConfigMap
4. this value-less label identifies the ConfigMap as config for a plugin (i.e. the built-in restore item action plugin)
5. this label identifies the name and kind of plugin that this ConfigMap is for.

!!! note

- 不能修改 plugin 配置 cm 的名字，且必须创建在 velero 命名空间下
- 填写 plugin 配置需注意区分填写资源 resources 还是 kind
- 修改 plugin 配置后需重启 velero pod
- 以下 yaml 是 plugin 配置展示样式，需转换为 json 添加到 configmap 中

如何配置 **velero-plugin-for-migration** 可参考以下 yaml 和注释

```

resourcesSelector: # (1)!

includedNamespaces: # (2)!
  - kube-system

excludedNamespaces: # (3)!

includedResources: # (4)!
  - pods
  - deployments
  - ingress

excludedResources: # (5)!
  - secrets

skipRestoreKinds:
  - endpointslice # (6)

labelSelector: 'app:dao-2048'

resourcesConverter: # (7)!
```

```

- ingress:
  enabled: true
  apiVersion: extensions/v1beat1
resourcesOperation: # (8)!

- kinds: ['pod'] # (9)!
  domain: labels # (10)!

operation:
  add:
    key1: values # (11)!

    key2: ""

remove:
  key3: values # (12)!

  key4: "" # (13)!

replace:
  key5: # (14)!

  - source
  - dest

  key6: # (15)!

  - ""
  - dest

  key7: # (16)!

  - source
  - ""

- kinds: ['deployment', 'daemonset'] # (17)!

  domain: annotations # (18)!

  scope: resourceSpec # (19)!

operation:
  add:
    key1: values # (20)!

    key2: ""

remove:
  key3: values # (21)!

  key4: "" # (22)!

replace:
  key5: # (23)!

  - source
  - dest

  key6: # (24)!

  - ""
  - dest

  key7: # (25)!

  - source
  - ""

```

1. plugin 需要处理或者忽略的资源

2. plugin 排除 backup 包含的 namespace
3. plugin 不处理 backup 包含的 namespace
4. plugin 处理 backup 包含的资源
5. plugin 不处理 backup 包含的资源
6. restore plugin 跳过 backup 内包含的资源，即不执行 restore 操作，该资源需要包含在 includedResources 中，才会被 plugin 捕捉到，该字段需填写资源 kind，不区分大小写
7. restore plugin 需要转换的资源，不支持配置具体资源字段转换
8. restore plugin 修改 resource/template 的 annotations/labels
9. 填写 backup 包含资源 kind，不区分大小写
10. 处理 resources labels
11. 添加 labels key1:values
12. 删除 labels key3:values，匹配 key,values
13. 删除 labels key4，只匹配 key，不匹配 values
14. 替换 labels key5:source -> key5:dest
15. 替换 labels key6: -> key6:dest，不匹配 key6 values
16. 替换 labels key7:source -> key7:""
17. 填写 backup 包含资源 kind，不区分大小写
18. 处理 resources template annotations
19. 处理 resources template spec 的 annotations 或者 labels，取决于 domain 配置
20. 添加 annotations key1:values

21. 删除 annotations key3:values , 匹配 key,values
 22. 删除 annotations key4 , 只匹配 key , 不匹配 values
 23. 替换 annotations key5:source -> key5:dest
 24. 替换 annotations key6: -> key6:dest , 不匹配 key6 values
 25. 替换 annotations key7:source -> key7:""
- 2.velero-plugin-for-dce plugin 获取以上配置后，根据配置对资源做链式操作，例如 ingress 经过 resourcesConverter 处理之后还会经过 resourcesOperation 处理。

镜像仓库替换

如果镜像地址发上了变化，可以通过在 Velero 命名空间中创建一个 ConfigMap 来配置映射，完成镜像地址的替换。

此配置适用于的迁移资源：

pod/deployment/statefulsets/daemonset/replicaset/replicationcontroller/job/cronjob。

!!! note

- ConfigMap 在还原集群中的 Velero 命名空间中创建。
- 标签为 velero.io/change-image-name: RestoreItemAction 的 ConfigMap 只能配置一个。
- 映射规则只会匹配符合的第一条规则，对应 ConfigMap 中的 case。

`apiVersion: v1`

`kind: ConfigMap`

`metadata:`

```

name: change-image-name-config # (1)!

namespace: velero # (2)!

labels: # (3)!

velero.io/plugin-config: "" # (4)!

velero.io/change-image-name: RestoreItemAction # (5)!
```

`data:`

```

"case1": "1.1.1.1:5000,2.2.2.2:3000" # (6)!

"case2": "5000,3000"

"case3": "abc:test,edf:test"

"case5": "test,latest"

"case4": "1.1.1.1:5000/abc:test,2.2.2.2:3000/edf:test"

"case5": "dev/,test/" # (7)!
```

- 1.any name can be used; Velero uses the labels (below) to identify it rather than the name
- 2.must be in the velero namespace
- 3.the below labels should be used verbatim in your ConfigMap
- 4.this value-less label identifies the ConfigMap as config for a plugin (i.e. the built-in restore item action plugin)
- 5.this label identifies the name and kind of plugin that this ConfigMap is for.
- 6.add 1+ key-value pairs here, where the key can be any words that ConfigMap accepts. the value should be <old_image_name_sub_part><delimiter><new_image_name_sub_part> for current implementation the <delimiter> can only be “,” eg: in case your old image name is 1.1.1.1:5000/abc:test
- 7.Please note that image name may contain more than one part that matching the replacing words. eg: in case your old image names are dev/image1:dev and dev/image2:dev, you want change to test/image1: dev and test/image2:dev the suggested replacing rule is: this will avoid unexpected replacement to the second “dev”.

迁移场景

资源和数据迁移

下文以业务应用迁移作为示例。

迁移情境：有状态应用 StatefulSet + PVC 备份和恢复

已知前提：备份集群中某个命名空间下已部署一个有状态应用（StatefulSet），例如 etcd，并挂载了 PVC。

备份

先将需要备份的资源在备份集群中进行备份，流程如下，在此之前，您可以先阅读 [应用备份](#)：

- 1.进入容器管理模块，点击左侧导航栏上的 **备份恢复** -> **应用备份**，进入 **应用备份列表**页面。



集群列表

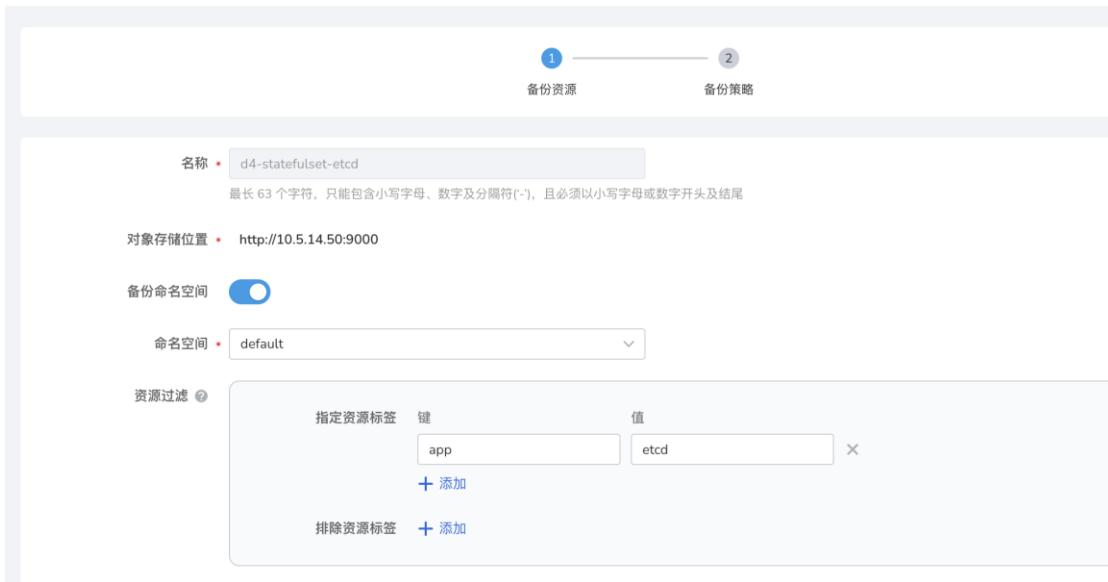
2. 在 **应用备份** 列表页面，选择备份集群。点击右上角的 **备份计划**，新建一个备份计划。

3. 参考下方说明填写备份配置。

- **名称**：新建备份计划的名称。
- **源集群**：计划执行应用备份的集群。
- **对象存储位置**：在**源集群安装 velero** 时配置的对象存储的访问路径。
- **命名空间**：需要进行备份的命名空间，支持多选。
- **高级配置**：根据资源标签对命名空间内的特定资源进行备份如某个应用，或者备份时根据资源标签对命名空间内的特定资源不进行备份。

!!! note

- 如果需要对多个或全部命名空间资源进行批量备份，请在命名空间选项选择多个或全部命名空间。
- 如果需要对命名空间内特定资源进行备份，请设置标签进行资源过滤。

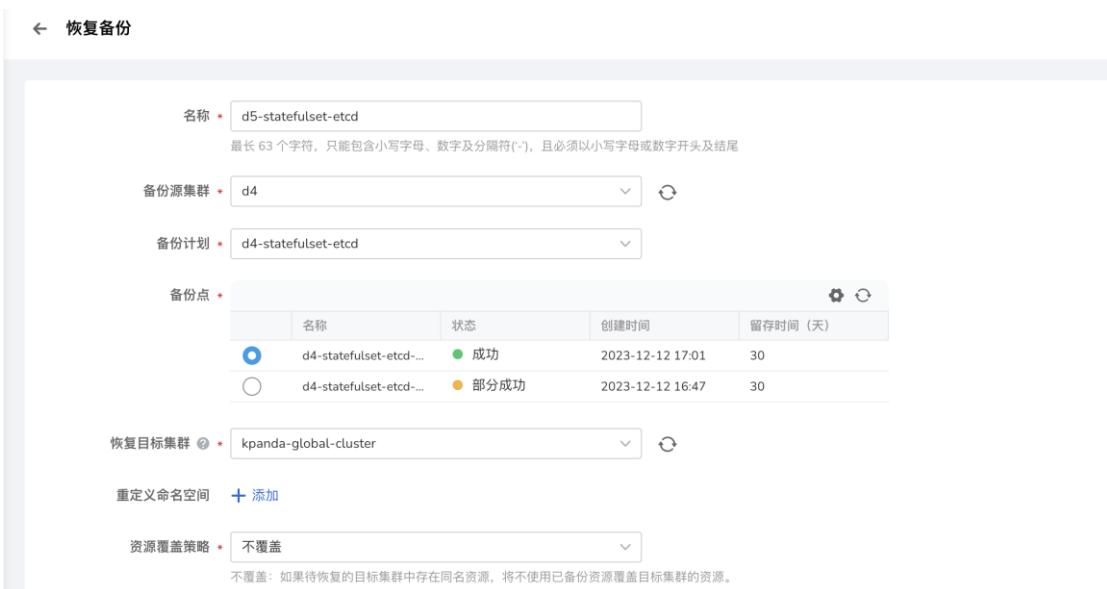


操作菜单

恢复

备份集群数据备份好后，将资源和数据在还原集群中恢复，操作步骤如下：

1. 进入容器管理模块，点击左侧导航栏上的 **备份恢复** -> **应用备份**，进入 **应用恢复** 列表页面。
2. 在 **应用恢复** 列表页面，选择恢复集群。点击右上角的 **恢复备份**，创建一个恢复备份任务。



恢复备份

3. 填写备份恢复配置，执行备份。

名称	目标集群	关联备份计划	状态	恢复时间
d5-statefulset-etcd	kpanda-global-cluster	d4-statefulset-etcd	成功	2023-12-12 17:11

恢复备份

!!! note

- 以上迁移流程同样适用于以下资源：
 - workload 的附属资源，如 secret、configmap
 - 多服务场景：Helm 应用 + Redis
- namespace 资源 和 cluster 资源如果配置了 RBAC，那对应类别的资源迁移成功后对应的 RBAC 也会一并迁移到 DCE 5.0

镜像仓库镜像迁移

下文介绍镜像仓库镜像迁移步骤。

1. 将 DCE 4.0 镜像仓库接入到 Kangaroo 仓库集成（管理员），操作步骤参考 [仓库集成](#)

成。

源仓库集成

!!! note

- 仓库地址使用 dce-registry 的 vip 地址 ip
- 账号密码使用 DCE 4.0 管理员的账号密码

2. 在管理员界面创建或集成一个 Harbor 仓库，用于迁移源镜像。

The screenshot shows the 'Repository Integration' section of the DaoCloud Admin interface. It displays a grid of Harbor instances:

实例名	状态	URL
inte-202-110	集成 健康	https://10.6.202.110/v2
inte-dce4-10-120-3-131	集成 健康	http://10.120.3.131
inte-hyt-test002	集成 健康	http://10.6.135.160:5005
inte-re-ci	集成 健康	https://release-ci.daocloud.io
inte-test-docker1	集成 健康	http://10.6.135.160:5006
inte-test-hyt-test-habor	集成 健康	http://10.6.135.168:30002
trust-harbor	托管 不健康	http://10.6.232.5:30727
trust-test-harbor	托管 不健康	http://10.6.88.52:31001

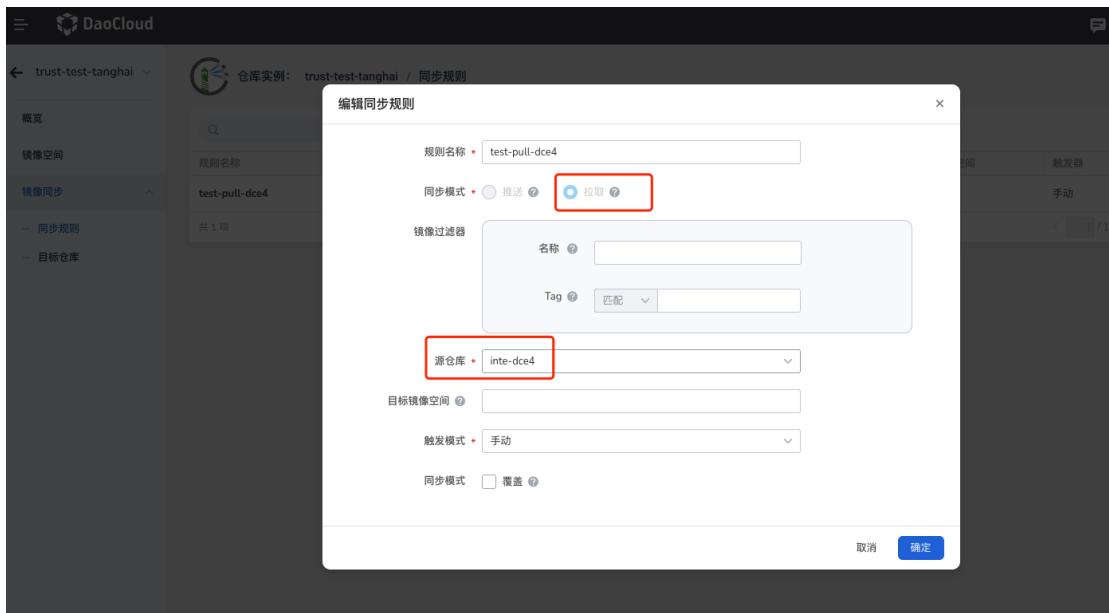
目标仓库创建

3. 进入 Harbor 仓库实例配置目标仓库与同步规则，规则触发后，Harbor 会自动从 dce-registry 拉取镜像。

The screenshot shows the configuration page for the 'trust-test-tanghai' Harbor instance. Under the 'Target Repository' tab, there is one listed repository:

仓库名称	状态	目标URL	提供者	验证远程证书	认证	创建时间
inte-dce4	健康	http://172.30.120.233	Docker Registry	否	basic	2023-05-16 13:34

目标仓库配置



同步规则配置

4. 点击 同步规则名称，进入同步任务详情页，可以查看镜像同步是否成功。

ID	状态	触发模式	创建时间	持续时间	成功百分比	总数	操作
395	成功	手动	2023-11-16 15:19	117s	100%	1047	⋮
335	成功	手动	2023-11-15 14:36	427s	100%	1047	⋮
333	失败	手动	2023-11-15 13:36	632s	17%	1047	⋮

目标仓库配置

网络策略迁移

Calico 网络策略迁移

参考资源和数据迁移流程，将 DCE 4.0 中的 Calico 服务迁移至 DCE 5.0。由于 IPPool 名

称不同，会导致服务异常，请迁移后手动删除服务 YAML 中的注解，以确保服务正常启动。

!!! note

- DCE 4.0 中，名称为 default-ipv4-ipool

- DCE 5.0 中，名称为 default-pool

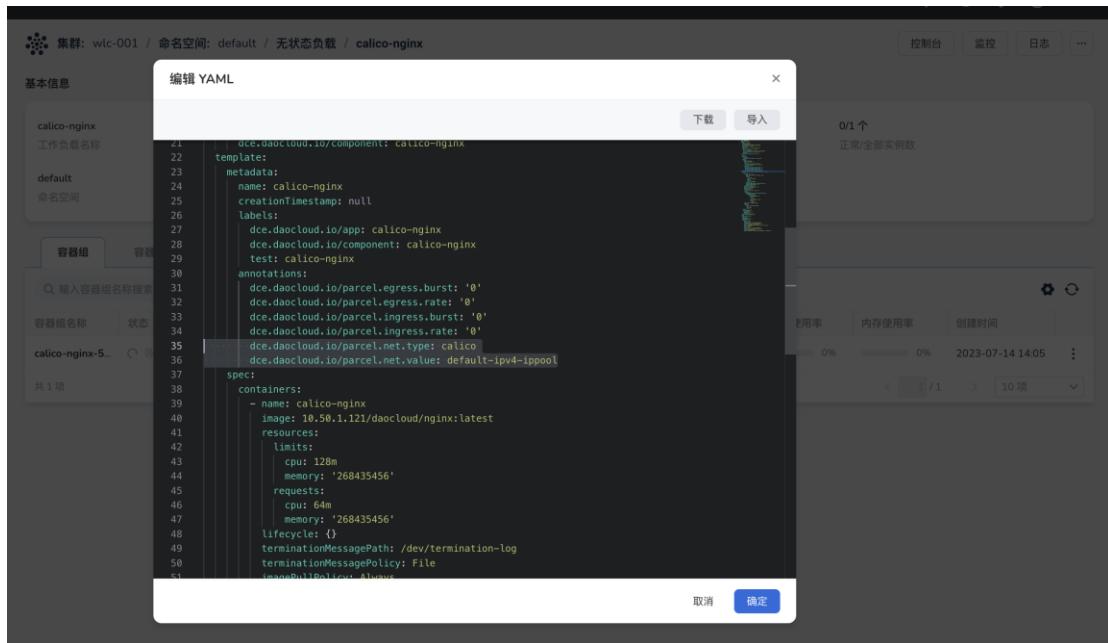
annotations:

dce.daocloud.io/parcel.net.type: calico

dce.daocloud.io/parcel.net.type: default-ipv4-ipool

DCE 4.0 服务

服务迁移异常



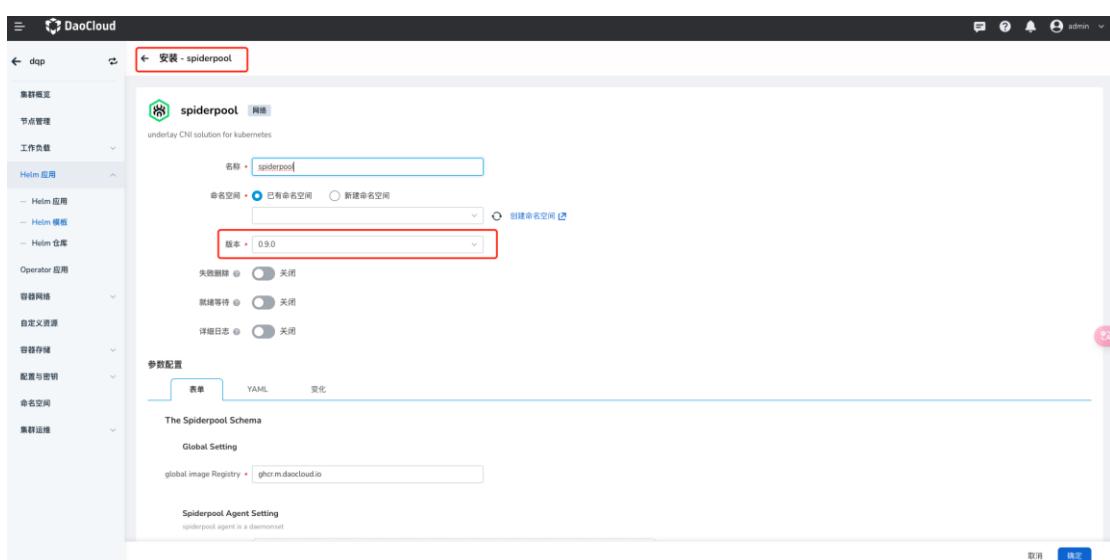
编辑服务 YAML

Parcel Underlay 网络策略迁移

下文介绍 Parcel Underlay 网络策略迁移步骤。

!!! note - 迁移时，DCE 5.0 中创建的 IP 地址，应与 DCE 4.0 中使用的 IP 地址保持一致，且创建的副本数量保持一致。

1. 在 还原集群 中安装 Helm 应用 spiderpool，安装流程参考[安装 spiderpool](#)。



安装 spiderpool

2. 进入 **还原集群** 详情页，选择左侧菜单 **容器网络 -> 网络配置**。

The screenshot shows the DaoCloud Enterprise 5.0 interface. On the left, there's a sidebar with various cluster management options like '节点管理', '工作负载', 'Helm 应用', 'Operator 应用', and '容器网络'. The '容器网络' option is selected and highlighted with a red box. The main content area shows '集群概览' (Cluster Overview) for the 'dqp' cluster. It includes sections for '基本信息' (Basic Information), '网络信息' (Network Information), '近一小时指标' (One-hour Metrics), '工作负载状态' (Workload Status), and '容器组与节点数' (Container Groups and Node Count). The '近一小时指标' section contains two line charts for CPU and memory usage.

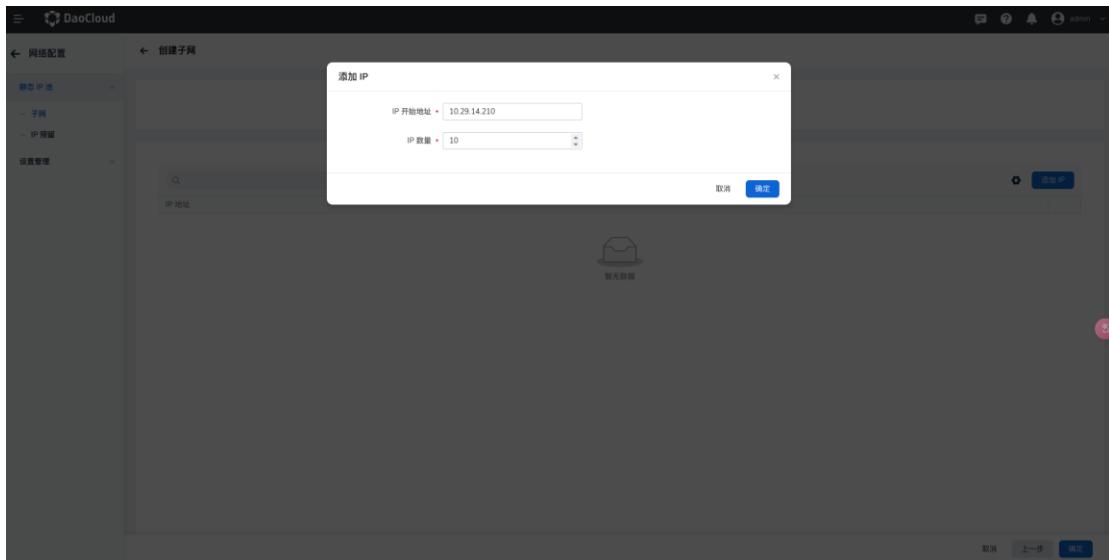
网络配置

3. 查看 DCE4 中使用的 IP 地址，在 DCE5.0 静态 IP 池 中创建与 DCE4 中相同的子网

IP 地址及 IP 池。子网及 IP 池的使用请参考[创建子网及 IP 池](#)。

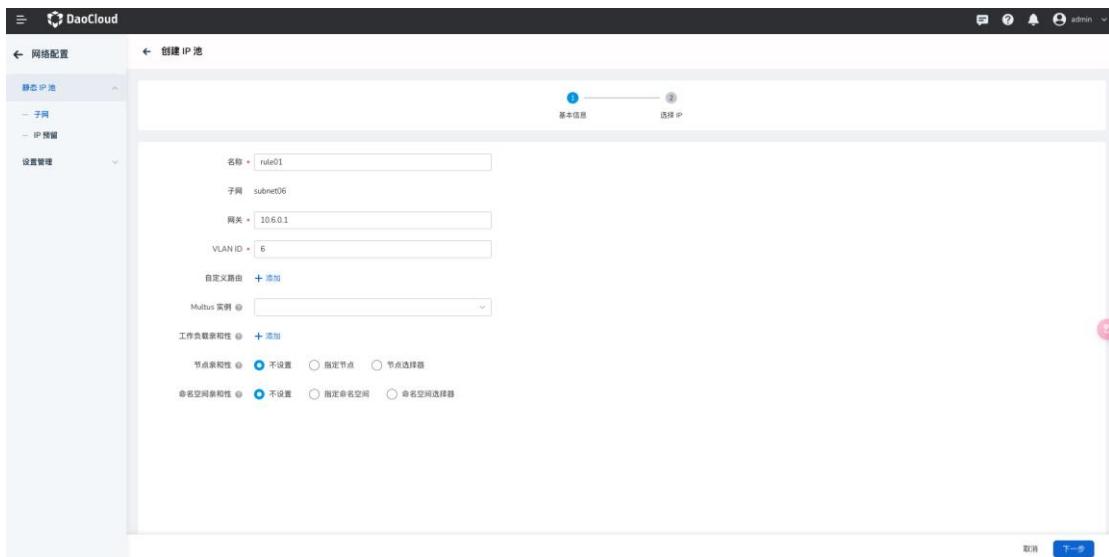
The screenshot shows the 'Create Subnet' wizard in the DaoCloud Enterprise 5.0 interface. The left sidebar has '静态 IP 池' expanded, with '子网' selected. The main panel is titled '创建子网' (Create Subnet) and is at step 1: '基本信息' (Basic Information). It requires filling in the '子网名称' (Subnet Name) field with 'subnet06', selecting 'IPv4' from the 'IPv4/IPv6 类型' (IPv4/IPv6 Type) dropdown, and providing the '子网' (Subnet) address as '10.29.14.210/24' and '网关' (Gateway) as '10.29.14.210'. There's also a 'VLAN ID' field set to '0'. At the bottom right, there are '取消' (Cancel) and '下一步' (Next Step) buttons.

创建子网



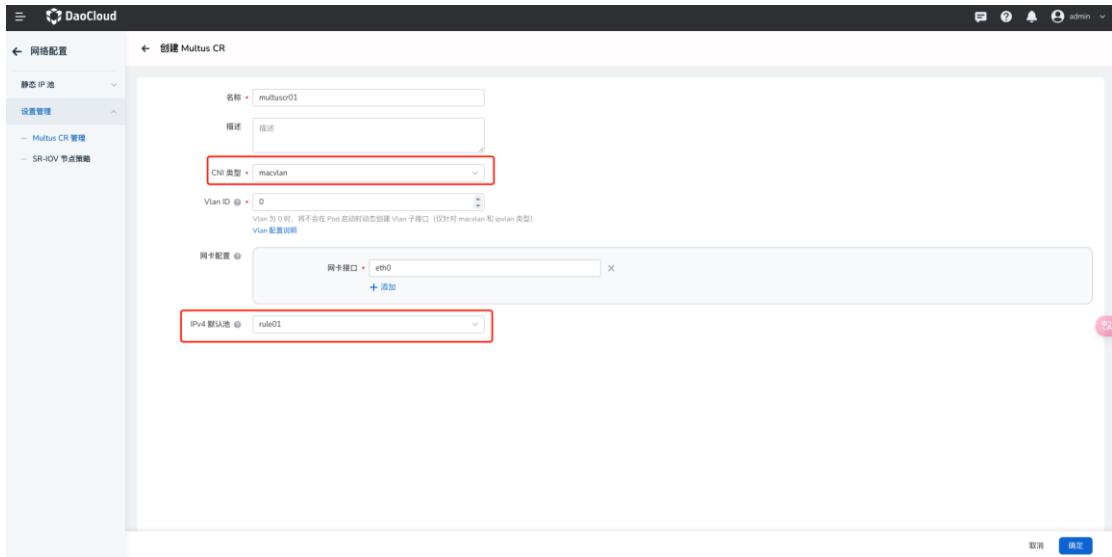
添加 IP

创建完子网后，在子网详情页创建 IP 池及添加 IP 开始地址与 IP 数量。



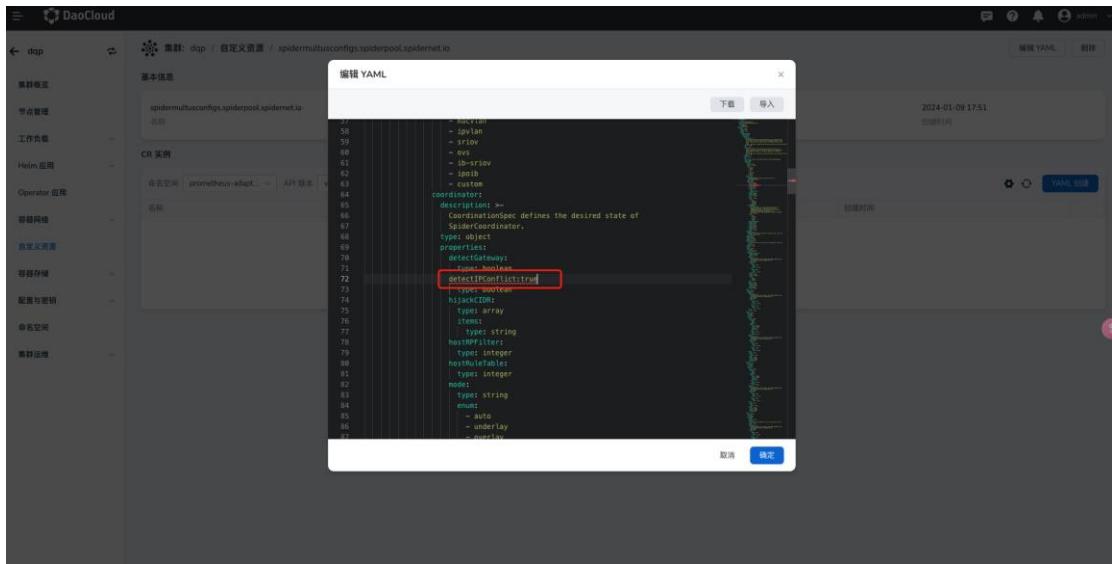
创建 IP 池

4. 创建 macvlan 类型的 Multus CR 实例，并选择刚才创建好的 IP 池。具体使用请参考[创建 Multus CR](#)



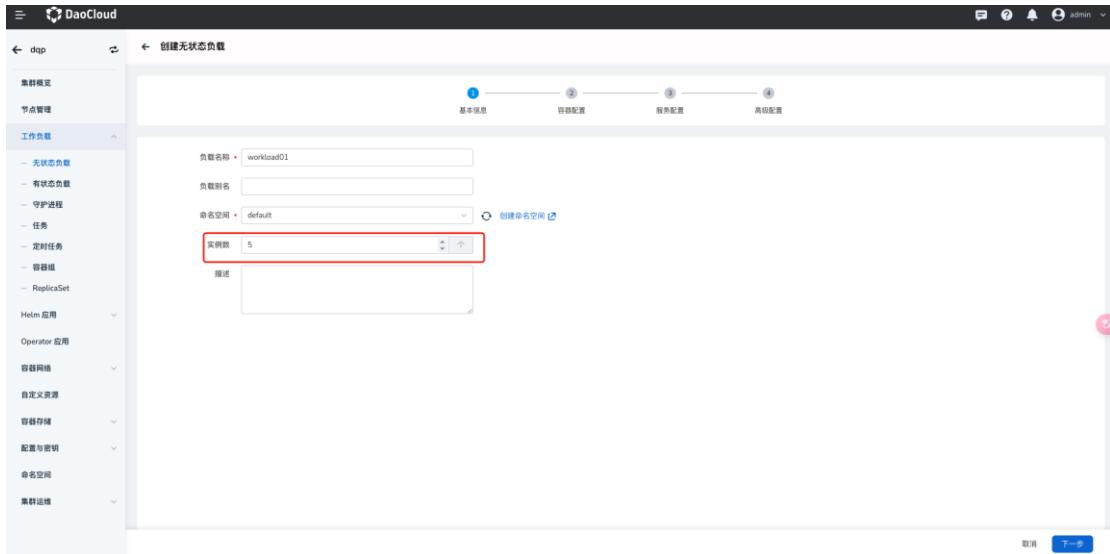
创建 Multus CR

5. 进入 **自定义资源** 界面，并手动修改 `spidermultusconfigs.spiderpool.spidernet.io` 的 `detectIPConflict` 字段为：`true`，此为开启 IP 冲突检测。

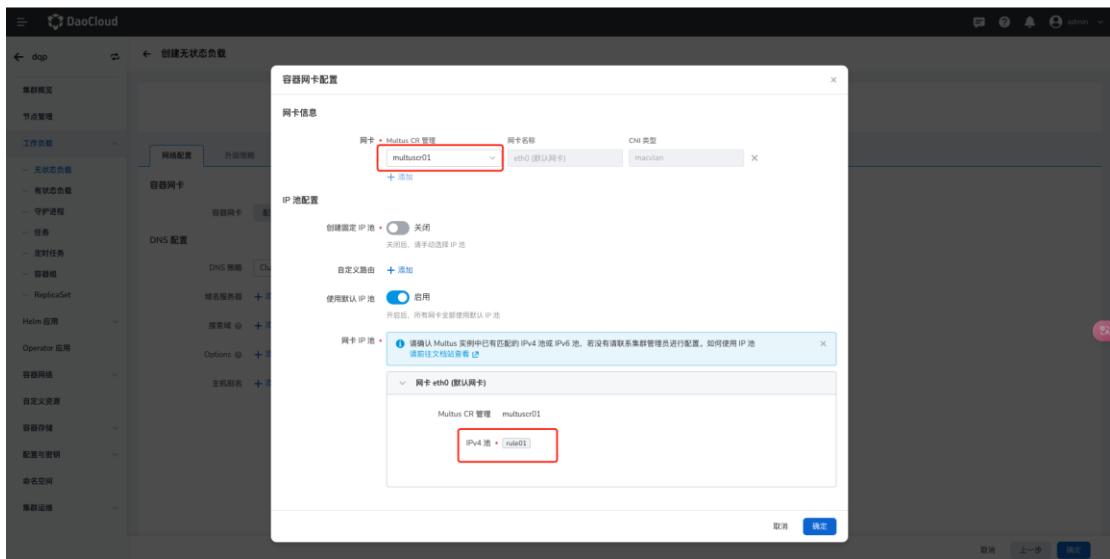


IP 检查

6. 进入 **工作负载 -> 容器网卡配置**，网卡选择刚才创建好的 `macvlan` 类型的 Multus CR，网卡 IP 池选择创建好的 IP 池，点击确定创建完成。此时容器组为运行中，则代表可以正常访问。



负载



选择网卡 IP 池

7. 创建 velero dce plugin configmap。

```

resourcesSelector:
  includedResources:
    - pods
    - deployments
resourcesConverter:
resourcesOperation:
  - kinds:
    - pod
  domain: annotations

```

```

operation:
replace:
  cni.projectcalico.org/ipv4pools:
    - ["default-ipv4-ippool"]
    - default-pool
  - kinds:
    - deployment
  domain: annotations
  scope: resourceSpec
operation:
remove:
  dce.daocloud.io/parcel.egress.burst:
  dce.daocloud.io/parcel.egress.rate:
  dce.daocloud.io/parcel.ingress.burst:
  dce.daocloud.io/parcel.ingress.rate:
  dce.daocloud.io/parcel.net.type:
  dce.daocloud.io/parcel.net.value:
  dce.daocloud.io/parcel.ovs.network.status:
add:
  ipam.spidernet.io/subnets: ' [ { "interface": "eth0", "ipv4": ["d5"] } ]'
  v1.multus-cni.io/default-network: kube-system/d5multus

```

8. 验证是否迁移成功。

1. 查看应用 YAML 中是否有 annotation。

```

annotations:
  ipam.spidernet.io/subnets: ' [ { "interface": "eth0", "ipv4": ["d5"] } ]'
  v1.multus-cni.io/default-network: kube-system/d5multus

```

2. 查看 Pod IP 是否在 配置的 IP 池内。

离线场景 Kubean 向下兼容版本的部署与 升级操作

为了满足客户对低版本的 K8s 集群的搭建，Kubean 提供了向下兼容并创建低版本的 K8s 集群能力，简称向下兼容版本的能力。

目前支持自建工作集群版本范围在 v1.26-v1.28，可以参阅 [DCE 5.0 集群版本支持体系](#)。

本文将演示如何部署低版本的 K8s 集群。

!!! note

本文演示的节点环境为：

- X86 架构
- CentOS 7 Linux 发行版

前提条件

- 准备一个 Kubean 所在的管理集群，并且当前环境已经部署支持 podman、skopeo、minio client 命令。如果不支持，可通过脚本进行安装依赖组件，[安装前置依赖](#)。
- 前往 [kubean](#) 查看发布的[制品](#)，并根据实际情况选择具体的制品版本。目前支持的制品版本及对应的集群版本范围如下：

制品包版本	支持集群范围	DCE 5.0 支持情况
release-2.21	v1.23.0 ~ v1.25.6	安装器 v0.14.0+ 已支持
release-2.22	v1.24.0 ~ v1.26.13	安装器 v0.15.0+ 已支持
release-2.23	v1.25.0 ~ v1.27.10	安装器 v0.16.0+ 已支持
release-2.24	v1.26.0 ~ v1.29.1	安装器 v0.17.0+ 已支持
release-2.25	v1.27.0 ~ v1.29.5	安装器 v0.20.0+ 已支持

!!! tip

在选择制品版本时，不仅需要参考集群版本范围，还需判断该制品 manifest 资源中相应组件(如 calico、containerd)版本范围是否覆盖当前集群该组件版本！

本文演示离线部署 K8s 集群到 1.23.0 版本及离线升级 K8s 集群从 1.23.0 版本到 1.24.0 版本，所以选择 release-2.21 的制品。

操作步骤

准备 Kubespray Release 低版本的相关制品

将 spray-job 镜像导入到离线环境的 Registry (镜像仓库) 中。

```
# 假设火种集群中的 registry 地址为 172.30.41.200
REGISTRY_ADDR="172.30.41.200"

# 镜像 spray-job 这里可以采用加速器地址, 镜像地址根据选择制品版本来决定
SPRAY_IMG_ADDR="ghcr.m.daocloud.io/kubean-io/spray-job: 2.21-d6f688f"

# skopeo 参数
SKOPEO_PARAMS="--insecure-policy -a --dest-tls-verify=false --retry-times=3"

# 在线环境: 导出 release-2.21 版本的 spray-job 镜像, 并将其转移到离线环境
skopeo copy docker:// ${SPRAY_IMG_ADDR} docker-archive:spray-job-2.21.tar

# 离线环境: 导入 release-2.21 版本的 spray-job 镜像到火种 registry
skopeo copy ${SKOPEO_PARAMS} docker-archive: spray-job-2.21.tar docker://${REGISTRY_A
DDR}/${SPRAY_IMG_ADDR}/m.daocloud/}
```

制作低版本 K8s 离线资源

1. 准备 manifest.yml 文件。

```
cat > "manifest.yml" <<EOF
image_arch:
  - "amd64" ## "arm64"
kube_version: ## 根据实际场景填写集群版本
  - "v1.23.0"
  - "v1.24.0"
EOF
```

2. 制作离线增量包。

```
# 创建 data 目录
mkdir data
# 制作离线包,
AIRGAP_IMG_ADDR="ghcr.m.daocloud.io/kubean-io/airgap-patch:2.21-d6f688f" # (1)
podman run --rm -v $(pwd)/manifest.yml:/manifest.yml -v $(pwd)/data:/data -e ZONE=
CN -e MODE=FULL ${AIRGAP_IMG_ADDR}
```

1. 镜像 spray-job 这里可以采用加速器地址，镜像地址根据选择制品版本来决定

3. 导入对应 k8s 版本的离线镜像与二进制包

```
# 将上一步 data 目录中的二进制导入二进制包至火种节点的 MinIO 中
cd ./data/amd64/files/
MINIO_ADDR="http://127.0.0.1:9000" # (1)!
MINIO_USER=rootuser MINIO_PASS=rootpass123 ./import_files.sh ${MINIO_ADDR}

# 将上一步 data 目录中的镜像导入二进制包至火种节点的镜像仓库中
cd ./data/amd64/images/
REGISTRY_ADDR="127.0.0.1" ./import_images.sh # (2)!
```

1. IP 替换为实际的仓库地址

2. IP 替换为实际的仓库地址

4. 将 manifest、localartifactset.cr.yaml 自定义资源部署到 Kubean 所在的管理集群或者全局服务集群 当中，本例使用的是全局服务集群。

```
# 部署 data 文件目录下的 localArtifactSet 资源
cd ./data
kubectl apply -f localartifactset.cr.yaml

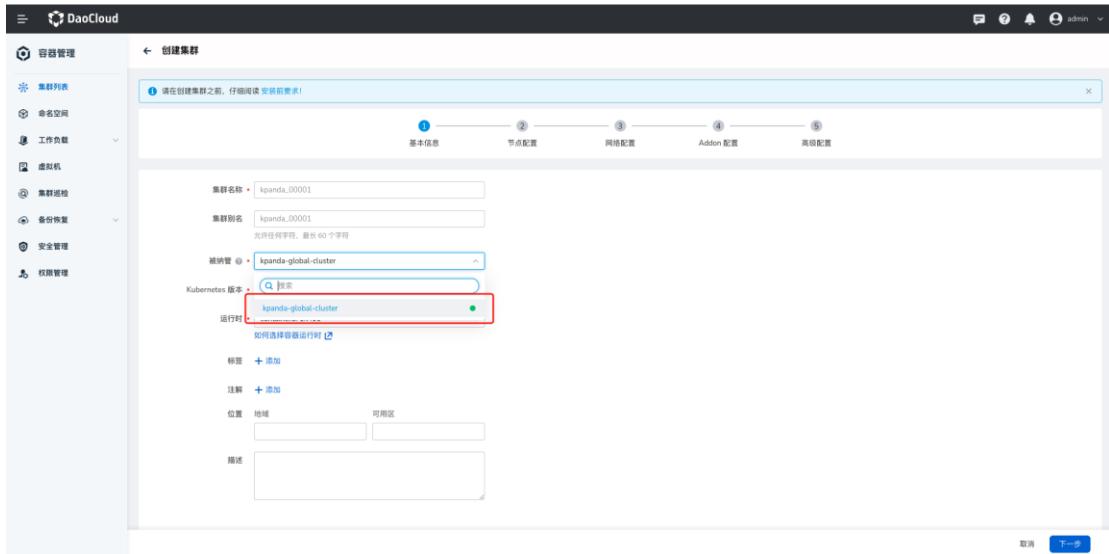
# 下载 release-2.21 版本的 manifest 资源
wget https://raw.githubusercontent.com/kubean-io/kubean-manifest/main/manifests/manifest-2.21-d6f688f.yaml

# 部署 release-2.21 对应的 manifest 资源
kubectl apply -f manifest-2.21-d6f688f.yaml
```

部署和升级 K8s 集群兼容版本

部署

1. 前往 容器管理，在 集群列表 页面中，点击 创建集群 按钮。
2. 被纳管参数选择 manifest、localartifactset.cr.yaml 自定义资源部署的集群，本例使用的是全局服务集群。



cluster01

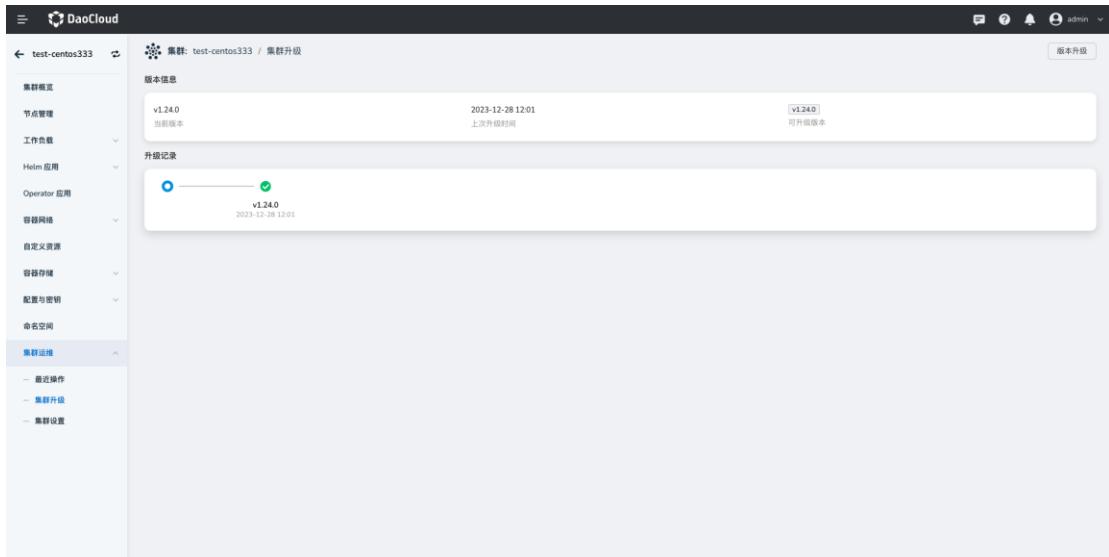
3. 其余参数参考[创建集群](#)。

cluster02

升级

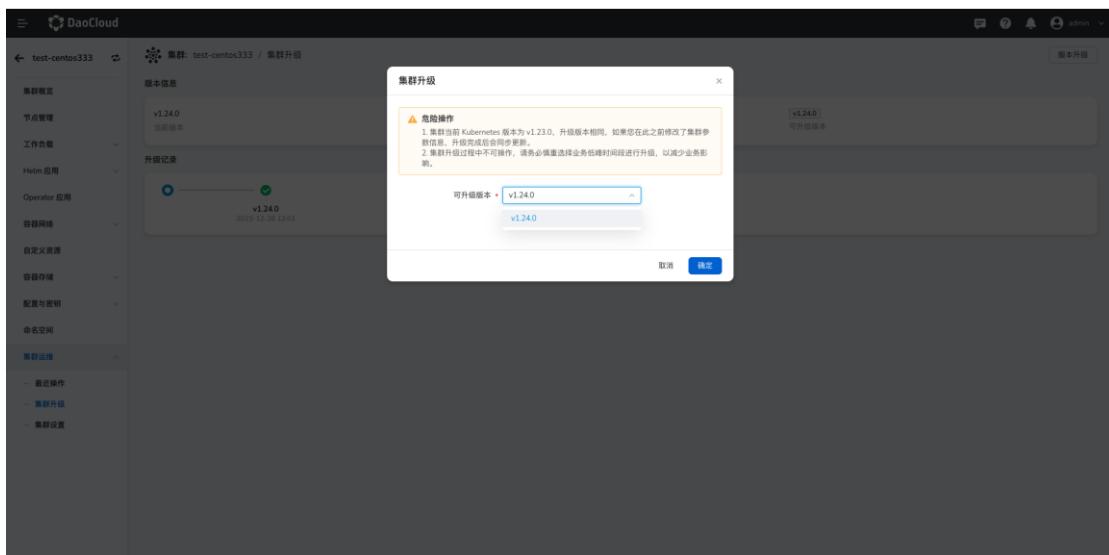
1. 选择新创建的集群，进去详情界面。

2. 然后在左侧导航栏点击 **集群运维 -> 集群升级**，在页面右上角点击 **版本升级**。



cluster03

3. 选择可用的集群进行升级。



cluster04

限制 Docker 单容器可占用的磁盘空间

Docker 在 17.07.0-ce 版本中引入 `overlay2.size`，本文介绍如何使用 `overlay2.size` 来限制 docker 单容器可占用的磁盘空间。

前提条件

在配置 docker overlay2.size 之前，需要调整操作系统中文件系统类型为 xfs 并使用 pquota 方式进行设备挂载。

格式化设备为 XFS 文件系统，可以执行以下命令：

```
mkfs.xfs -f /dev/xxx
```

!!! note

pquota 限制的是项目（project）磁盘配额。

设置单容器磁盘可占用空间

满足以上条件后，用户可以通过设置 docker overlay2.size 来限制单容器磁盘占用空间大小。

命令行示例如下：

```
sudo dockerd -s overlay2 --storage-opt overlay2.size=1G
```

场景演练

接下来以一个实际的例子来演练一下限制 docker 单容器可占用的磁盘空间整体实现流程。

目标

部署一个 Kubernetes 集群，并限制 docker 单容器可占用的磁盘空间大小为 1G，超出 1G 将无法使用。

操作流程

1. 登录目标节点，查看 fstab 文件，获取当前设备的挂载情况。

```
$ cat /etc/fstab
```

```
# /etc/fstab
```

```

# Created by anaconda on Thu Mar 19 11:32:59 2020
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
/dev/mapper/centos-root / xfs defaults 0 0
UUID=3ed01f0e-67a1-4083-943a-343b7fed1708 /boot xfs defau
lts 0 0
/dev/mapper/centos-swap swap swap defaults 0 0

```

以图示节点设备为例，可以看到 XFS 格式设备 /dev/mapper/centos-root 以默认方式

defaults 挂载到 / 根目录。

2. 配置 xfs 文件系统使用 pquota 方式挂载。

1. 修改 fstab 文件，将挂载方式从 defaults 更新为 rw,pquota；

```

# 修改 fstab 配置
$ vi /etc/fstab
- /dev/mapper/centos-root /
0 0
+ /dev/mapper/centos-root /
xfs defaults
0 0

# 验证配置是否有误
$ mount -a

```

2. 查看 pquota 是否生效

```

xfs_quota -x -c print
[root@localhost ~]# xfs_quota -x -c print
Filesystem          Pathname
/                  /dev/mapper/centos-root (pquota)
/boot              /dev/sda1
[root@localhost ~]# 

```

看看 fstab 配置

!!! note

如果 pquota 未生效，请检查操作系统是否开启 pquota 选项，如果未开启，需要在系统引导配置 /etc/grub2.cfg 中添加 `rootflags=pquota` 参数，配置完成后，需要 reboot 重启操作系统。

```

## BEGIN /etc/grub.d/30_linux_##
menuentry 'CentOS Linux (3.10.0-957.el7.x86_64) 7 (Core)' --class centos --class gnu-linux --class gnu --class unix --unrestricted $menuentry_id_option 'grublinux-3.10.0-957.el7.x86_64-advanced-99c549-6129-487c-8f99-aee22b389ebo' {
    load_video
    set gfxpayload=keep
    insmod gzio
    insmod part_msdos
    insmod ext4
    set root='hd0,msdos1'
    if [ x$feature_platform_search_hint = xy ]; then
        search --no-floppy --fs-uuid --set=root ${lsblk}+hd0,msdos1 --hint-efi=hd0,msdos1 --hint-baremetal=hd0,msdos1 --hint='hd0,msdos1' 3ed01f0e-67a1-4883-943a-343b7fed1708
    else
        search --no-floppy --fs-uuid --set=root 3ed01f0e-67a1-4883-943a-343b7fed1708
    fi
    linux /vmlinuz-3.10.0-957.el7.x86_64 root=/dev/mapper/centos-root ro crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swapp rhgb quiet LANG=en_US.UTF-8 rootflags=padata
    initrd /initramfs-3.10.0-957.el7.x86_64.img
}

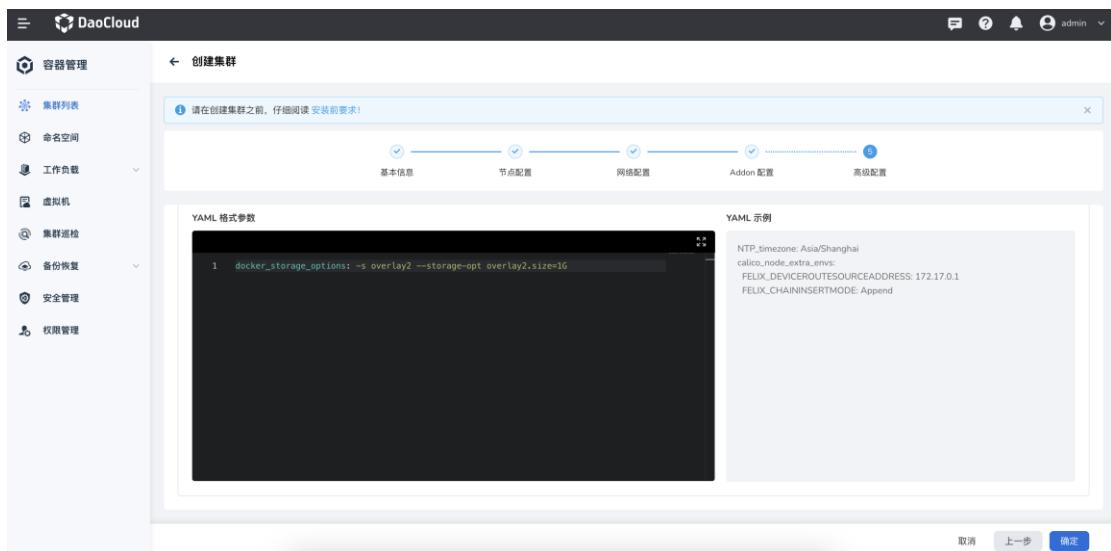
menuentry 'CentOS Linux (0-rescue-3d84eb000af14f8de9758affcc0e8375) 7 (Core)' --class centos --class gnu-linux --class gnu --class unix --unrestricted $menuentry_id_option 'grublinux-0-rescue-3d84eb000af14f8de9758affcc0e8375-advanced-99c549-6129-487c-8f99-aee22b389ebo' {
    load_video
    insmod gzio
    insmod part_msdos
    insmod ext4
    set root='hd0,msdos1'
    if [ x$feature_platform_search_hint = xy ]; then
        search --no-floppy --fs-uuid --set=root ${lsblk}+hd0,msdos1 --hint-efi=hd0,msdos1 --hint-baremetal=hd0,msdos1 --hint='hd0,msdos1' 3ed01f0e-67a1-4883-943a-343b7fed1708
    else
        search --no-floppy --fs-uuid --set=root 3ed01f0e-67a1-4883-943a-343b7fed1708
    fi
    linux /vmlinuz-0-rescue-3d84eb000af14f8de9758affcc0e8375 root=/dev/mapper/centos-root ro crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swapp rhgb quiet rootflags=padata
    initrd /initramfs-0-rescue-3d84eb000af14f8de9758affcc0e8375.img
}

```

操作系统开启 pquota 选项

3. 创建集群 -> 高级配置 -> 自定义参数 中添加 docker_storage_options 参数，设置单

容器磁盘可占用空间。



添加自定义参数

!!! note

也可以基于 kubelet manifest 操作，在 vars.conf 里添加 `docker_storage_options` 参数

。

```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
 name: sample-vars-conf
 namespace: kubelet-system
data:
 group_vars.yml: |
 unsafe_show_logs: true
 container_manager: docker

```

```

+ docker_storage_options: -s overlay2 --storage-opt overlay2.size=1G # 新增 docker
r_storage_options 参数

 kube_network_plugin: calico
 kube_network_plugin_multus: false
 kube_proxy_mode: iptables
 etcd_deployment_type: kubeadm
 override_system_hostname: true
...
```

```

4. 查看 dockerd 服务运行配置，检查磁盘限制是否设置成功。

```

[root@localhost ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/etc/systemd/system/docker.service; enabled; vendor preset: disabled)
   Active: active (running) since Wed Mar 28 2024-03-13 04:37:05 EDT; 24min ago
     Docs: https://docs.docker.com
 Main PID: 19489 (dockerd)
 Tasks: 18
 Memory: 1.7G
 CGroup: /system.slice/docker.service
         └─19489 /usr/bin/dockerd -l /var/run/docker.sock -s overlay2 --storage-opt overlay2.size=1G

Mar 13 04:43:18 node1 dockerd[19489]: time="2024-03-13T04:43:18.449036277-04:00" level=warning msg="Published ports are discarded when using host network mode"
Mar 13 04:43:48 node1 dockerd[19489]: time="2024-03-13T04:43:48.633036339-04:00" level=info msg="ignoring event" container=c1e0312086f753791 module=dockerd namespace=moby topic=/tasks/delete type="events.TaskDelete"
Mar 13 04:43:48 node1 dockerd[19489]: time="2024-03-13T04:43:48.913076095-04:00" level=info msg="ignoring event" container=f22439eb3117e0d931784274f186c27227f9cc5c71 module=dockerd namespace=moby topic=/tasks/delete type="events.TaskDelete"
Mar 13 04:43:48 node1 dockerd[19489]: time="2024-03-13T04:43:48.913076095-04:00" level=info msg="ignoring event" container=f22439eb3117e0d931784274f186c27227f9cc5c71 module=dockerd namespace=moby topic=/tasks/delete type="events.TaskDelete"
Mar 13 04:43:57 node1 dockerd[19489]: time="2024-03-13T04:43:57.984073877-04:00" level=error msg="collecting stats for container \"$/k8s_kube-optServer_node1_kube-system_ae070ec779/c507391b1b9c51619041:0\". Could not get container for 33ed84a0754fe30479e..."
Mar 13 04:44:01 node1 dockerd[19489]: time="2024-03-13T04:44:01.891098115-04:00" level=info msg="ignoring event" container=d4d91a12b9ad3b2f8330a153251827971160447/7d7051f1787c1bdc07 module=dockerd namespace=moby topic=/tasks/delete type="events.TaskDelete"
Mar 13 04:44:01 node1 dockerd[19489]: time="2024-03-13T04:44:01.891098115-04:00" level=info msg="ignoring event" container=d4d91a12b9ad3b2f8330a153251827971160447/7d7051f1787c1bdc07 module=dockerd namespace=moby topic=/tasks/delete type="events.TaskDelete"
Mar 13 04:44:18 node1 dockerd[19489]: time="2024-03-13T04:44:18.876058559-04:00" level=error msg="collecting stats for container \"$/k8s_kube-optServer_node1_kube-system_ae070ec779/c507391b1b9c51619041:0\". Could not get container for 33ed84a0754fe30479e..."
Mar 13 04:44:19 node1 dockerd[19489]: time="2024-03-13T04:44:19.890039465-04:00" level=info msg="Container failed to exit within 30s of signal 15 - using the force" container=25943bd8bc7ebce3d84932e665f0c3b78869842570dd1fc3b0c358ab5133 module=dockerd namespace=moby topic=/tasks/delete type="events.TaskDelete"
Mar 13 04:44:19 node1 dockerd[19489]: time="2024-03-13T04:44:19.9620c25092-04:00" level=info msg="ignoring event" container=25943bd8bc7ebce3d84932e665f0c3b78869842570dd1fc3b0c358ab5133 module=dockerd namespace=moby topic=/tasks/delete type="events.TaskDelete"
```

```

#### 检查容器磁盘限制

以上，完成限制 docker 单容器可占用的磁盘空间整体实现流程。

## 边缘集群部署和管理实践

对于资源受限的边缘或物联网场景，Kubernetes 无法很好的满足资源要求，为此需要一个轻量化 Kubernetes 方案，既能实现容器管理和编排能力，又能给业务应用预留更多资源空间。本文介绍边缘集群 k3s 的部署和全生命周期管理实践。

## 节点规划

### 架构

- x86\_64
- armhf
- arm64/aarch64

### 操作系统

- 可以在大多数现代 Linux 系统上工作

## CPU/内存

- 单节点 K3s 集群

	最小 CPU	推荐 CPU	最小内存	推荐内存
K3s cluster	1 core	2 cores	1.5 GB	2 GB

- 多节点 K3s 集群

	最小 CPU	推荐 CPU	最小内存	推荐内存
K3s server	1 core	2 cores	1 GB	1.5 GB
K3s agent	1 core	2 cores	512 MB	1 GB

- 节点入站规则

- 根据需要确保以下端口未被占用
- 若有特殊要求不能关闭防火墙，需确保端口为放行

协议	端口	源	目的	描述
TC P 议	2379-2380	Servers	Servers	适用于 HA 与嵌入式 etcd
TC P	6443	Agents	Servers	K3s supervisor 和 Kubernetes API Server
UD P	8472	All nodes	All nodes	仅适用于 Flannel VXLAN
TC P	10250	All nodes	All nodes	Kubelet metrics
UD P	51820	All nodes	All nodes	仅适用于带有 IPv4 的 Flannel Wireguard
UD P	51821	All nodes	All nodes	仅适用于带有 IPv6 的 Flannel Wireguard
TC P	5001	All nodes	All nodes	仅适用于嵌入分布式注册表 ( Spegel )
TC P	6443	All nodes	All nodes	仅适用于嵌入分布式注册表 ( Spegel )

- 节点角色

登录用户需具备 root 权限

server node	agent node	描述
1	0	一台 server 节点
1	2	一台 server 节点、两台 agent 节点
3	0	三台 server 节点

## 前置准备

### 1. 保存安装脚本到安装节点（任意可以访问到集群节点的节点）

```
$ cat > k3slcm <<'EOF'
#!/bin/bash
set -e

airgap_image=${K3S_AIRGAP_IMAGE:-}
k3s_bin=${K3S_BINARY:-}
install_script=${K3S_INSTALL_SCRIPT:-}

servers=${K3S_SERVERS:-}
agents=${K3S_AGENTS:-}
ssh_user=${SSH_USER:-root}
ssh_password=${SSH_PASSWORD:-}
ssh_privatekey_path=${SSH_PRIVATEKEY_PATH:-}
extra_server_args=${EXTRA_SERVER_ARGS:-}
extra_agent_args=${EXTRA_AGENT_ARGS:-}
first_server=$(cut -d, -f1 <<<"$servers,")
other_servers=$(cut -d, -f2- <<<"$servers,")

install_script_env="INSTALL_K3S_SKIP_SELINUX_RPM=true INSTALL_K3S_SELINUX_WARN=true "
[-n "$K3S_VERSION"] && install_script_env+="INSTALL_K3S_VERSION=$K3S_VERSION"

ssh_opts="-q -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -o ControlPath=/tmp/ssh_mux_%h_%p_%r -o ControlMaster=auto -o ControlPersist=10m"

if [-n "$ssh_privatekey_path"]; then
 ssh_opts+=" -i $ssh_privatekey_path"
elif [-n "$ssh_password"]; then
 askpass=$(mktemp)
 echo "echo -n $ssh_password" > $askpass
```

```

chmod 0755 $askpass
export SSH_ASKPASS=$askpass SSH_ASKPASS_REQUIRE=force
else
 echo "SSH_PASSWORD or SSH_PRIVATEKEY_PATH must be provided" && exit
 1
fi

log_info() { echo -e "\033[36m* $*\033[0m"; }
clean() { rm -f $askpass; }
trap clean EXIT

IFS=':' read -ra all_nodes <<< "$servers,$agents"
if [-n "$k3s_bin"]; then
 for node in ${all_nodes[@]}; do
 chmod +x "$k3s_bin" "$install_script"
 ssh $ssh_opts "$ssh_user@$node" "mkdir -p /usr/local/bin /var/lib/rancher/k3s/agent/images"
 log_info "Copying $airgap_image to $node"
 scp -O $ssh_opts "$airgap_image" "$ssh_user@$node:/var/lib/rancher/k3s/agent/images"
 log_info "Copying $k3s_bin to $node"
 scp -O $ssh_opts "$k3s_bin" "$ssh_user@$node:/usr/local/bin/k3s"
 log_info "Copying $install_script to $node"
 scp -O $ssh_opts "$install_script" "$ssh_user@$node:/usr/local/bin/k3s-install.sh"
 done
 install_script_env+="INSTALL_K3S_SKIP_DOWNLOAD=true "
else
 for node in ${all_nodes[@]}; do
 log_info "Downloading install script for $node"
 ssh $ssh_opts "$ssh_user@$node" "curl -sLo /usr/local/bin/k3s-install.sh https://get.k3s.io/ && chmod +x /usr/local/bin/k3s-install.sh"
 done
fi

restart_k3s() {
 local node=$1
 previous_k3s_version=$(ssh $ssh_opts "$ssh_user@$first_server" "kubectl get no -o wide | awk '$6==\"$node\" {print \$5}'")
 [-n "$previous_k3s_version" -a "$previous_k3s_version" != "$K3S_VERSION" -a -n "$k3s_bin"] && return 0 || return 1
}

token=mynodetoken
install_script_env+="${K3S_INSTALL_SCRIPT_ENV:-}"

```

```

if [-z "$other_servers"]; then
 log_info "Installing on server node [$first_server]"
 ssh $ssh_opts "$ssh_user@$first_server" "env $install_script_env /usr/local/bin/k3s-install.sh server --token $token $extra_server_args"
 ! restart_k3s "$first_server" || ssh $ssh_opts "$ssh_user@$first_server" "systemctl restart k3s.service"
else
 log_info "Installing on first server node [$first_server]"
 ssh $ssh_opts "$ssh_user@$first_server" "env $install_script_env /usr/local/bin/k3s-install.sh server --cluster-init --token $token $extra_server_args"
 ! restart_k3s "$first_server" || ssh $ssh_opts "$ssh_user@$first_server" "systemctl restart k3s.service"
IFS=':' read -ra other_server_nodes <<< "$other_servers"
for node in ${other_server_nodes[@]}; do
 log_info "Installing on other server node [$node]"
 ssh $ssh_opts "$ssh_user@$node" "env $install_script_env /usr/local/bin/k3s-install.sh server --server https://$first_server:6443 --token $token $extra_server_args"
 ! restart_k3s "$node" || ssh $ssh_opts "$ssh_user@$node" "systemctl restart k3s.service"
done
fi

if [-n "$agents"]; then
IFS=':' read -ra agent_nodes <<< "$agents"
for node in ${agent_nodes[@]}; do
 log_info "Installing on agent node [$node]"
 ssh $ssh_opts "$ssh_user@$node" "env $install_script_env K3S_TOKEN=$token K3S_URL=https://$first_server:6443 /usr/local/bin/k3s-install.sh agent --token $token $extra_agent_args"
 ! restart_k3s "$node" || ssh $ssh_opts "$ssh_user@$node" "systemctl restart k3s-agent.service"
done
fi
EOF

```

## 2. (可选) 离线环境下，在一台可联网节点下载 K3s 相关离线资源，并拷贝到安装节点

## [联网节点执行]

```

设置 K3s 版本为 v1.30.2+k3s1
$ export k3s_version=v1.30.2+k3s1

离线镜像包
arm64 链接为 https://github.com/k3s-io/k3s/releases/download/$k3s_version/k3s-airgap-images-arm64.tar.zst

```

```
$ curl -LO https://github.com/k3s-io/k3s/releases/download/$k3s_version/k3s-airgap-images-amd64.tar.zst

k3s 二进制文件
arm64 链接为 https://github.com/k3s-io/k3s/releases/download/$k3s_version/k3s-arm64
$ curl -LO https://github.com/k3s-io/k3s/releases/download/$k3s_version/k3s

安装部署脚本
$ curl -Lo k3s-install.sh https://get.k3s.io/

上述资源拷贝到安装节点文件系统上

[安装节点执行]
$ export K3S_AIRGAP_IMAGE=<资源存放目录>/k3s-airgap-images-amd64.tar.zst
$ export K3S_BINARY=<资源存放目录>/k3s
$ export K3S_INSTALL_SCRIPT=<资源存放目录>/k3s-install.sh
```

### 3. 关闭防火墙和 swap ( 若防火墙无法关闭 , 可放行上述入站端口 )

```
Ubuntu 关闭防火墙方法
$ sudo ufw disable
RHEL / CentOS / Fedora / SUSE 关闭防火墙方法
$ systemctl disable firewalld --now
$ sudo swapoff -a
$ sudo sed -i '/swap/s/^/#/' /etc/fstab
```

## 部署集群

下文测试环境信息为 Ubuntu 22.04 LTS, amd64 , 离线安装

### 1. 在安装节点根据部署规划设置节点信息 , 并导出环境变量 , 多个节点以半角逗号 , 分

隔

```
==== "1 server / 0 agent"
```shell
export K3S_SERVERS=172.30.41.5 $ export SSH_USER=root
# 若使用 public key 方式登录, 确保已将公钥添加到各节点的 ~/.ssh/authorized_keys

export SSH_PRIVATEKEY_PATH=<私钥路径>
export SSH_PASSWORD=<SSH 密码>
```
==== "1 server / 2 agent"
```shell
export K3S_SERVERS=172.30.41.5
```

```

export K3S_AGENTS=172.30.41.6,172.30.41.7
export SSH_USER=root

# 若使用 public key 方式登录，确保已将公钥添加到各节点的 ~/.ssh/authorized_keys
export SSH_PRIVATEKEY_PATH=<私钥路径>
export SSH_PASSWORD=<SSH 密码>
```
==== "3 server / 0 agent"
```shell
export K3S_SERVERS=172.30.41.5,172.30.41.6,172.30.41.7
export SSH_USER=root

# 若使用 public key 方式登录，确保已将公钥添加到各节点的 ~/.ssh/authorized_keys
export SSH_PRIVATEKEY_PATH=<私钥路径>
export SSH_PASSWORD=<SSH 密码>
```

```

## 2. 执行部署操作

以 3 server / 0 agent 模式为例，每台机器必须有一个唯一的主机名

```

若有更多 K3s 安装脚本环境变量设置需求，请设置 K3S_INSTALL_SCRIPT_ENV，其
值参考 https://docs.k3s.io/reference/env-variables
若需对 server 或 agent 节点作出额外配置，请设置 EXTRA_SERVER_ARGS 或 E
XTRA_AGENT_ARGS，其值参考 https://docs.k3s.io/cli/server https://docs.k3s.io/cli/agen
t
$ bash k3slcm
* Copying ./v1.30.2/k3s-airgap-images-amd64.tar.zst to 172.30.41.5
* Copying ./v1.30.2/k3s to 172.30.41.5
* Copying ./v1.30.2/k3s-install.sh to 172.30.41.5
* Copying ./v1.30.2/k3s-airgap-images-amd64.tar.zst to 172.30.41.6
* Copying ./v1.30.2/k3s to 172.30.41.6
* Copying ./v1.30.2/k3s-install.sh to 172.30.41.6
* Copying ./v1.30.2/k3s-airgap-images-amd64.tar.zst to 172.30.41.7
* Copying ./v1.30.2/k3s to 172.30.41.7
* Copying ./v1.30.2/k3s-install.sh to 172.30.41.7
* Installing on first server node [172.30.41.5]
[INFO] Skipping k3s download and verify
[INFO] Skipping installation of SELinux RPM
[INFO] Creating /usr/local/bin/kubectl symlink to k3s
[INFO] Creating /usr/local/bin/crictl symlink to k3s
[INFO] Creating /usr/local/bin/ctr symlink to k3s
[INFO] Creating killall script /usr/local/bin/k3s-killall.sh
[INFO] Creating uninstall script /usr/local/bin/k3s-uninstall.sh
[INFO] env: Creating environment file /etc/systemd/system/k3s.service.env

```

```
[INFO] systemd: Creating service file /etc/systemd/system/k3s.service
[INFO] systemd: Enabling k3s unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s.service → /etc/systemd/sys-
tem/k3s.service.
[INFO] systemd: Starting k3s
* Installing on other server node [172.30.41.6]
....
```

### 3. 检查集群状态

```
$ kubectl get no -owide
```

| NAME    | STATUS | ROLES                     | INTERNAL-IP | EXTERNAL-IP        | OS-IMAGE           | AGE            | VERSION                                    |
|---------|--------|---------------------------|-------------|--------------------|--------------------|----------------|--------------------------------------------|
|         |        |                           |             |                    |                    | KERNEL-VERSION | CONTAINER-RUNTIME                          |
| server1 | Ready  | control-plane,etcd,master | <none>      | Ubuntu 22.04.3 LTS | 3m51s v1.30.2+k3s1 | 172.30.41.5    | 5.15.0-78-generic containerd://1.7.17-k3s1 |
| server2 | Ready  | control-plane,etcd,master | <none>      | Ubuntu 22.04.3 LTS | 3m18s v1.30.2+k3s1 | 172.30.41.6    | 5.15.0-78-generic containerd://1.7.17-k3s1 |
| server3 | Ready  | control-plane,etcd,master | <none>      | Ubuntu 22.04.3 LTS | 3m7s v1.30.2+k3s1  | 172.30.41.7    | 5.15.0-78-generic containerd://1.7.17-k3s1 |

```
$ kubectl get pod --all-namespaces -owide
```

| NAMESPACE   | NAME                                    | TUS   | RESTARTS  | AGE     | IP     | READY      | STA            |
|-------------|-----------------------------------------|-------|-----------|---------|--------|------------|----------------|
|             |                                         |       |           |         |        | NODE       | NOMINATED NODE |
| kube-system | coredns-576bfc4dc7-z4x2s                | 8m31s | 10.42.0.3 | server1 | <none> | 1/1 <none> | Running 0      |
| kube-system | helm-install-traefik-98kh5              | 8m31s | 10.42.0.4 | server1 | <none> | 0/1 <none> | Completed 1    |
| kube-system | helm-install-traefik-crd-9xtfd          | 8m31s | 10.42.0.5 | server1 | <none> | 0/1 <none> | Completed 0    |
| kube-system | local-path-provisioner-86f46b7bf7-qt995 | 8m31s | 10.42.0.6 | server1 | <none> | 1/1 <none> | Running 0      |
| kube-system | metrics-server-557ff575fb-kptsh         | 8m31s | 10.42.0.2 | server1 | <none> | 1/1 <none> | Running 0      |
| kube-system | svclb-traefik-f95cc81c-mgcjh            | 6m28s | 10.42.1.3 | server2 | <none> | 2/2 <none> | Running 0      |
| kube-system | svclb-traefik-f95cc81c-xtb8f            | 6m28s | 10.42.2.2 | server3 | <none> | 2/2 <none> | Running 0      |
| kube-system | svclb-traefik-f95cc81c-zcsxl            | 6m28s | 10.42.0.7 | server1 | <none> | 2/2 <none> | Running 0      |
| kube-system | traefik-5fb479b77-6pbh5                 | 6m28s | 10.42.1.2 | server2 | <none> | 1/1 <none> | Running 0      |

## 升级集群

1. 如升级到 v1.30.3+k3s1 版本，按照 前置准备 步骤 2 重新下载离线资源并拷贝到安装节点，同时在安装节点导出离线资源路径环境变量。（若为联网升级，则跳过此操作）

2. 执行升级操作

```
$ export K3S_VERSION=v1.30.3+k3s1
$ bash k3slcm
* Copying ./v1.30.3/k3s-airgap-images-amd64.tar.zst to 172.30.41.5
* Copying ./v1.30.3/k3s to 172.30.41.5
* Copying ./v1.30.3/k3s-install.sh to 172.30.41.5
* Copying ./v1.30.3/k3s-airgap-images-amd64.tar.zst to 172.30.41.6
* Copying ./v1.30.3/k3s to 172.30.41.6
* Copying ./v1.30.3/k3s-install.sh to 172.30.41.6
* Copying ./v1.30.3/k3s-airgap-images-amd64.tar.zst to 172.30.41.7
* Copying ./v1.30.3/k3s to 172.30.41.7
* Copying ./v1.30.3/k3s-install.sh to 172.30.41.7
* Installing on first server node [172.30.41.5]
[INFO] Skipping k3s download and verify
[INFO] Skipping installation of SELinux RPM
[INFO] Skipping /usr/local/bin/kubectl symlink to k3s, already exists
[INFO] Skipping /usr/local/bin/crictl symlink to k3s, already exists
[INFO] Skipping /usr/local/bin/ctr symlink to k3s, already exists
[INFO] Creating killall script /usr/local/bin/k3s-killall.sh
[INFO] Creating uninstall script /usr/local/bin/k3s-uninstall.sh
[INFO] env: Creating environment file /etc/systemd/system/k3s.service.env
[INFO] systemd: Creating service file /etc/systemd/system/k3s.service
[INFO] systemd: Enabling k3s unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s.service → /etc/systemd/system/k3s.service.
[INFO] No change detected so skipping service start
* Installing on other server node [172.30.41.6]
.....
```

3. 检查集群状态

```
$ kubectl get node -owide
 NAME STATUS ROLES AGE VERSION
 INTERNAL-IP EXTERNAL-IP OS-IMAGE KERNEL-VERSION
 CONTAINER-RUNTIME
 server1 Ready control-plane,etcd,master 18m v1.30.3+k3s1 172.30.41.5 <
```

```

none> Ubuntu 22.04.3 LTS 5.15.0-78-generic containerd://1.7.17-k3s1
server2 Ready control-plane,etcd,master 17m v1.30.3+k3s1 172.30.41.6 <
none> Ubuntu 22.04.3 LTS 5.15.0-78-generic containerd://1.7.17-k3s1
server3 Ready control-plane,etcd,master 17m v1.30.3+k3s1 172.30.41.7 <
none> Ubuntu 22.04.3 LTS 5.15.0-78-generic containerd://1.7.17-k3s1

```

```

$ kubectl get po --all-namespaces -owide
NAMESPACE NAME READY STATUS RESTARTS AGE IP NODE NOMINATED-NODE READINESS GATES
TUS RESTARTS AGE IP NODE
kube-system coredns-576bfc4dc7-z4x2s 1/1 Running 0
 18m 10.42.0.3 server1 <none> <none>
kube-system helm-install-traefik-98kh5 0/1 Completed 1
 18m <none> server1 <none> <none>
kube-system helm-install-traefik-crd-9xtfd 0/1 Completed 0
 18m <none> server1 <none> <none>
kube-system local-path-provisioner-6795b5f9d8-t4rvm 1/1 Running 0
 2m49s 10.42.2.3 server3 <none> <none>
kube-system metrics-server-557ff575fb-kptsh 1/1 Running 0
 18m 10.42.0.2 server1 <none> <none>
kube-system svclb-traefik-f95cc81c-mgcjh 2/2 Running 0
 16m 10.42.1.3 server2 <none> <none>
kube-system svclb-traefik-f95cc81c-xtb8f 2/2 Running 0
 16m 10.42.2.2 server3 <none> <none>
kube-system svclb-traefik-f95cc81c-zcsxl 2/2 Running 0
 16m 10.42.0.7 server1 <none> <none>
kube-system traefik-5fb479b77-6pbh5 1/1 Running 0
 16m 10.42.1.2 server2 <none> <none>

```

## 扩容集群

1. 如添加新的 agent 节点：

```
export K3S_AGENTS=172.30.41.8
```

添加新的 server 节点如下：

```
< export K3S_SERVERS=172.30.41.5,172.30.41.6,172.30.41.7

> export K3S_SERVERS=172.30.41.5,172.30.41.6,172.30.41.7,172.30.41.8,172.30.41.9
```

2. 执行扩容操作（以添加 agent 节点为例）

```
$ bash k3slcm
* Copying ./v1.30.3/k3s-airgap-images-amd64.tar.zst to 172.30.41.5
* Copying ./v1.30.3/k3s to 172.30.41.5
```

```

* Copying ./v1.30.3/k3s-install.sh to 172.30.41.5
* Copying ./v1.30.3/k3s-airgap-images-amd64.tar.zst to 172.30.41.6
* Copying ./v1.30.3/k3s to 172.30.41.6
* Copying ./v1.30.3/k3s-install.sh to 172.30.41.6
* Copying ./v1.30.3/k3s-airgap-images-amd64.tar.zst to 172.30.41.7
* Copying ./v1.30.3/k3s to 172.30.41.7
* Copying ./v1.30.3/k3s-install.sh to 172.30.41.7
* Copying ./v1.30.3/k3s-airgap-images-amd64.tar.zst to 172.30.41.8
* Copying ./v1.30.3/k3s to 172.30.41.8
* Copying ./v1.30.3/k3s-install.sh to 172.30.41.8
* Installing on first server node [172.30.41.5]
[INFO] Skipping k3s download and verify
[INFO] Skipping installation of SELinux RPM
[INFO] Skipping /usr/local/bin/kubectl symlink to k3s, already exists
[INFO] Skipping /usr/local/bin/crictl symlink to k3s, already exists
[INFO] Skipping /usr/local/bin/ctr symlink to k3s, already exists
[INFO] Creating killall script /usr/local/bin/k3s-killall.sh
[INFO] Creating uninstall script /usr/local/bin/k3s-uninstall.sh
[INFO] env: Creating environment file /etc/systemd/system/k3s.service.env
[INFO] systemd: Creating service file /etc/systemd/system/k3s.service
[INFO] systemd: Enabling k3s unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s.service → /etc/systemd/system/k3s.service.
[INFO] No change detected so skipping service start
.....
* Installing on agent node [172.30.41.8]
[INFO] Skipping k3s download and verify
[INFO] Skipping installation of SELinux RPM
[INFO] Creating /usr/local/bin/kubectl symlink to k3s
[INFO] Creating /usr/local/bin/crictl symlink to k3s
[INFO] Creating /usr/local/bin/ctr symlink to k3s
[INFO] Creating killall script /usr/local/bin/k3s-killall.sh
[INFO] Creating uninstall script /usr/local/bin/k3s-agent-uninstall.sh
[INFO] env: Creating environment file /etc/systemd/system/k3s-agent.service.env
[INFO] systemd: Creating service file /etc/systemd/system/k3s-agent.service
[INFO] systemd: Enabling k3s-agent unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s-agent.service → /etc/systemd/system/k3s-agent.service.
[INFO] systemd: Starting k3s-agent

```

### 3. 检查集群状态

```
$ kubectl get node -owide
NAME STATUS ROLES AGE VERSION
INTERNAL-IP EXTERNAL-IP OS-IMAGE KERNEL-VERSION

```

| CONTAINER-RUNTIME |       |                           |                    |                   |                          |             |  |  |
|-------------------|-------|---------------------------|--------------------|-------------------|--------------------------|-------------|--|--|
| agent1            | Ready | <none>                    |                    | 57s               | v1.30.3+k3s1             | 172.30.41.8 |  |  |
|                   |       | <none>                    | Ubuntu 22.04.3 LTS | 5.15.0-78-generic | containerd://1.7.17-k3s1 |             |  |  |
| server1           | Ready | control-plane,etcd,master | 12m                | v1.30.3+k3s1      | 172.30.41.5              | <           |  |  |
|                   |       | none>                     | Ubuntu 22.04.3 LTS | 5.15.0-78-generic | containerd://1.7.17-k3s1 |             |  |  |
| server2           | Ready | control-plane,etcd,master | 11m                | v1.30.3+k3s1      | 172.30.41.6              | <           |  |  |
|                   |       | none>                     | Ubuntu 22.04.3 LTS | 5.15.0-78-generic | containerd://1.7.17-k3s1 |             |  |  |
| server3           | Ready | control-plane,etcd,master | 11m                | v1.30.3+k3s1      | 172.30.41.7              | <           |  |  |
|                   |       | none>                     | Ubuntu 22.04.3 LTS | 5.15.0-78-generic | containerd://1.7.17-k3s1 |             |  |  |

## 缩容集群

1. 仅在待删除节点执行 `k3s-uninstall.sh` 或 `k3s-agent-uninstall.sh`

2. 在任意 server 节点上执行：

```
kubectl delete node <节点名称>
```

## 卸载集群

1. 在所有 server 节点手动执行 `k3s-uninstall.sh`

2. 在所有 agent 节点手动执行 `k3s-agent-uninstall.sh`

## 在离线混部

企业中一般存在两种类型的工作负载：在线服务（latency-sensitive service）和离线任务（batch job）。在线服务如搜索/支付/推荐等，具有处理优先级高、时延敏感性高、错误容忍度低以及白天负载高晚上负载低等特点。而离线任务如 AI 训练/大数据处理等，具有处理优先级低、时延敏感性低、错误容忍度高以及运行时负载一直较高等特点。由于在线服务与离线任务这两类工作负载天然存在互补性，将在/离线业务混合部署是提高服务器资源利用率的有效途径。

- 可以将离线业务混部到在线业务的服务器上，让离线业务能够充分利用在线业务服务

器的空闲资源，提高在线业务服务器资源利用率，实现降本增效。

- 当业务中临时需要大量的资源，这个时候可以将在线业务弹性混部到离线业务的服务器上，优先保证在线业务的资源需求，临时需求结束后再把资源归还给离线业务。

当前使用开源项目 [Koordinator](#) 作为在离线混部的解决方案。

Koordinator 是一个基于 QoS 的 Kubernetes 混合工作负载调度系统。它旨在提高对延迟敏感的工作负载和批处理作业的运行时效率和可靠性，简化与资源相关的配置调整的复杂性，并增加 Pod 部署密度以提高资源利用率。

## Koordinator QoS

Koordinator 调度系统支持的 QoS 有五种类型：

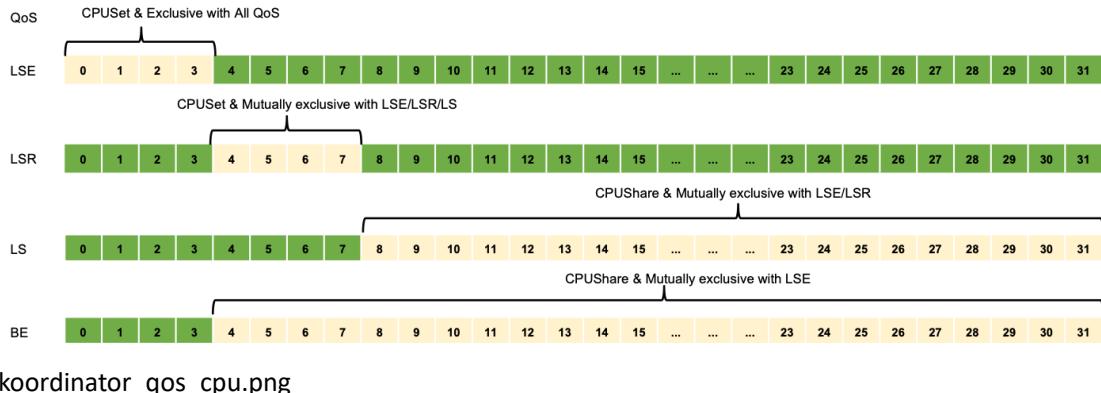
| QoS    | 特点        | 说明                                                            |
|--------|-----------|---------------------------------------------------------------|
| SYSTEM | 系统进程，资源受限 | 对于 DaemonSets 等系统服务，虽然需要保证系统的延时，但也需要限制节点上这些系统的服务容器的资源使用，以确保其不 |

| QoS                              | 特点                          | 说明                                                            |
|----------------------------------|-----------------------------|---------------------------------------------------------------|
| LSE(Latency Sensitive Exclusive) | 保留资源并组织同 QoS 的 Pod 共享<br>资源 | 占用过多的<br>资源<br>很少使用，<br>常见于中间<br>件类应用，<br>一般在独立<br>的资源池中      |
| LSR(Latency Sensitive Reserved)  | 预留资源以获得更好的确定性               | 使用<br>类似于社区<br>的<br>Guaranteed<br>, CPU 核被<br>绑定              |
| LS(Latency Sensitive)            | 共享资源，对突发流量有更好的弹性            | 微服务工作<br>负载的典型<br>QoS 级别，<br>实现更好的<br>资源弹性和<br>更灵活的资<br>源调整能力 |
| BE(Best Effort)                  | 共享不包括 LSE 的资源，资源运行质         | 批量作业的                                                         |

| QoS | 特点              | 说明                           |
|-----|-----------------|------------------------------|
|     | 量有限，甚至在极端情况下被杀死 | 典型 QoS 水平，在一定时期内稳定的计算吞吐量，低成本 |

## Koordinator QoS CPU 编排原则

- LSE/LSR Pod 的 Request 和 Limit 必须相等，CPU 值必须是 1000 的整数倍。
- LSE Pod 分配的 CPU 是完全独占的，不得共享。如果节点是超线程架构，只保证逻辑核心维度是隔离的，但是可以通过 CPUBindPolicyFullPCPUs 策略获得更好的隔离。
- LSR Pod 分配的 CPU 只能与 BE Pod 共享。
- LS Pod 绑定了与 LSE/LSR Pod 独占之外的共享 CPU 池。
- BE Pod 绑定使用节点中除 LSE Pod 独占之外的所有 CPU。
- 如果 kubelet 的 CPU 管理器策略为 static 策略，则已经运行的 K8s Guaranteed Pods 等价于 Koordinator LSR。
- 如果 kubelet 的 CPU 管理器策略为 none 策略，则已经运行的 K8s Guaranteed Pods 等价于 Koordinator LS。
- 新创建但未指定 Koordinator QoS 的 K8s Guaranteed Pod 等价于 Koordinator LS。



koordinator\_qos\_cpu.png

## 快速上手

### 前提条件

- 已经部署 DCE 5.0 容器管理平台，且平台运行正常。
- 容器管理模块已接入 Kubernetes 集群或者已创建 Kubernetes 集群，且能够访问集群的 UI 界面。
- 当前集群已安装 koordinator 并正常运行，安装步骤可参考 [koordinator 离线安装](#)。

### 操作步骤

以下示例中创建 4 个副本数为 1 的 deployment，设置 QoS 类别为 LSE, LSR, LS, BE，待 pod 创建完成后，观察各 pod 的 CPU 分配情况。

1. 创建名称为 nginx-lse 的 deployment，QoS 类别为 LSE, yaml 文件如下。

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-lse
 labels:
 app: nginx-lse
spec:
 replicas: 1
 selector:
 matchLabels:
```

```

 app: nginx-lse
 template:
 metadata:
 name: nginx-lse
 labels:
 app: nginx-lse
 koordinator.sh/qosClass: LSE # 设置 QoS 类别为 LSE
 # 调度器将在物理内核之间均匀的分配逻辑 CPU
 annotations:
 scheduling.koordinator.sh/resource-spec: '{"preferredCPUBindPolicy": "Spread
ByPCPUs"}'
 spec:
 schedulerName: koord-scheduler # 使用 koord-scheduler 调度器
 containers:
 - name: nginx
 image: release.daocloud.io/kpanda/nginx:1.25.3-alpine
 resources:
 limits:
 cpu: '2'
 requests:
 cpu: '2'
 priorityClassName: koord-prod

```

2. 创建名称为 nginx-lsr 的 deployment , QoS 类别为 LSR, yaml 文件如下。

```

apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-lsr
 labels:
 app: nginx-lsr
spec:
 replicas: 1
 selector:
 matchLabels:
 app: nginx-lsr
 template:
 metadata:
 name: nginx-lsr
 labels:
 app: nginx-lsr
 koordinator.sh/qosClass: LSR # 设置 QoS 类别为 LSR
 # 调度器将在物理内核之间均匀的分配逻辑 CPU
 annotations:
 scheduling.koordinator.sh/resource-spec: '{"preferredCPUBindPolicy": "Spread
ByPCPUs"}'

```

```

 ByPCPUs"}'

spec:
 schedulerName: koord-scheduler # 使用 koord-scheduler 调度器
 containers:
 - name: nginx
 image: release.daocloud.io/kpanda/nginx:1.25.3-alpine
 resources:
 limits:
 cpu: '2'
 requests:
 cpu: '2'
 priorityClassName: koord-prod

```

3. 创建名称为 nginx-ls 的 deployment , QoS 类别为 LS, yaml 文件如下。

```

apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-ls
 labels:
 app: nginx-ls
spec:
 replicas: 1
 selector:
 matchLabels:
 app: nginx-ls
 template:
 metadata:
 name: nginx-ls
 labels:
 app: nginx-ls
 koordinator.sh/qosClass: LS # 设置 QoS 类别为 LS
 # 调度器将在物理内核之间均匀的分配逻辑 CPU
 annotations:
 scheduling.koordinator.sh/resource-spec: '[{"preferredCPUBindPolicy": "Spread
ByPCPUs"}'

spec:
 schedulerName: koord-scheduler
 containers:
 - name: nginx
 image: release.daocloud.io/kpanda/nginx:1.25.3-alpine
 resources:
 limits:
 cpu: '2'
 requests:

```

```
cpu: '2'
priorityClassName: koord-prod
```

4. 创建名称为 nginx-be 的 deployment , QoS 类别为 BE, yaml 文件如下。

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-be
 labels:
 app: nginx-be
spec:
 replicas: 1
 selector:
 matchLabels:
 app: nginx-be
 template:
 metadata:
 name: nginx-be
 labels:
 app: nginx-be
 koordinator.sh/qosClass: BE # 设置 QoS 类别为 BE
 # 调度器将在物理内核之间均匀的分配逻辑 CPU
 annotations:
 scheduling.koordinator.sh/resource-spec: '{"preferredCPUBindPolicy": "SpreadByPCPUs"}'
 spec:
 schedulerName: koord-scheduler # 使用 koord-scheduler 调度器
 containers:
 - name: nginx
 image: release.daocloud.io/kpanda/nginx:1.25.3-alpine
 resources:
 limits:
 kubernetes.io/batch-cpu: 2k
 requests:
 kubernetes.io/batch-cpu: 2k
 priorityClassName: koord-batch
```

查看 pod 状态，当 pod 处于 running 后，查看各 pod 的 CPU 分配情况。

```
[root@controller-node-1 ~]# kubectl get pod
```

| NAME                       | READY | STATUS  | RESTARTS | AGE   |
|----------------------------|-------|---------|----------|-------|
| nginx-be-577c946b89-js2qn  | 1/1   | Running | 0        | 4h41m |
| nginx-ls-54746c8cf8-rh4b7  | 1/1   | Running | 0        | 4h51m |
| nginx-lse-56c9cd77f5-cdqbd | 1/1   | Running | 0        | 4h41m |
| nginx-lsr-c7fdb97d8-b58h8  | 1/1   | Running | 0        | 4h51m |

本示例中使用 `get_cpuset.sh` 脚本查看 Pod 的 cpuset 信息，脚本内容如下。

```
#!/bin/bash

获取 Pod 的名称和命名空间作为输入参数
POD_NAME=$1
NAMESPACE=${2-default}

确保提供了 Pod 名称和命名空间
if [-z "$POD_NAME"] || [-z "$NAMESPACE"]; then
 echo "Usage: $0 <pod_name> <namespace>"
 exit 1
fi

使用 kubectl 获取 Pod 的 UID 和 QoS 类别
POD_INFO=$(kubectl get pod "$POD_NAME" -n "$NAMESPACE" -o jsonpath="{.metadata.uid} {.status.qosClass} {.status.containerStatuses[0].containerID}")
read -r POD_UID POD_QOS CONTAINER_ID <<< "$POD_INFO"

检查 UID 和 QoS 类别是否成功获取
if [-z "$POD_UID"] || [-z "$POD_QOS"]; then
 echo "Failed to get UID or QoS Class for Pod $POD_NAME in namespace $NAMESPACE."
 exit 1
fi

POD_UID="${POD_UID//-/_}"
CONTAINER_ID="${CONTAINER_ID//containerd:\//cri-containerd-}"".scope

根据 QoS 类别构建 cgroup 路径
case "$POD_QOS" in
 Guaranteed)
 QOS_PATH="kubepods-pod.slice/$POD_UID.slice"
 ;;
 Burstable)
 QOS_PATH="kubepods-burstable.slice/kubepods-burstable-pod$POD_UID.slice"
 ;;
 BestEffort)
 QOS_PATH="kubepods-besteffort.slice/kubepods-besteffort-pod$POD_UID.slice"
 ;;
 *)
 echo "Unknown QoS Class: $POD_QOS"
 exit 1
 ;;
esac
```

```

esac

CPUGROUP_PATH="/sys/fs/cgroup/kubepods.slice/$QOS_PATH"

检查路径是否存在
if [! -d "$CPUGROUP_PATH"]; then
 echo "CPUs cgroup path for Pod $POD_NAME does not exist: $CPUGROUP_PA
TH"
 exit 1
fi

读取并打印 cpuset 值
CPUSERT=$(cat "$CPUGROUP_PATH/$CONTAINER_ID/cpuset.cpus")
echo "CPU set for Pod $POD_NAME ($POD_QOS QoS): $CPUSERT"

```

查看各 Pod 的 cpuset 分配情况。

1.QoS 类型为 LSE 的 Pod, 独占 0-1 核, 不与其他类型的 Pod 共享 CPU。

```
[root@controller-node-1 ~]# ./get_cpuset.sh nginx-lse-56c9cd77f5-cdqbd
CPU set for Pod nginx-lse-56c9cd77f5-cdqbd (Burstable QoS): 0-1
```

2.QoS 类型为 LSR 的 Pod, 绑定 CPU 2-3 核, 可与 BE 类型的 Pod 共享。

```
[root@controller-node-1 ~]# ./get_cpuset.sh nginx-lsr-c7fdb97d8-b58h8
CPU set for Pod nginx-lsr-c7fdb97d8-b58h8 (Burstable QoS): 2-3
```

3.QoS 类型为 LS 的 Pod, 使用 CPU 4-15 核, 绑定了与 LSE/LSR Pod 独占之外的共享 CPU 池。

```
[root@controller-node-1 ~]# ./get_cpuset.sh nginx-ls-54746c8cf8-rh4b7
CPU set for Pod nginx-ls-54746c8cf8-rh4b7 (Burstable QoS): 4-15
```

4.QoS 类型为 BE 的 pod, 可使用 LSE Pod 独占之外的 CPU。

```
[root@controller-node-1 ~]# ./get_cpuset.sh nginx-be-577c946b89-js2qn
CPU set for Pod nginx-be-577c946b89-js2qn (BestEffort QoS): 2,4-12
```

## Koordinator 离线安装

Koordinator 是一个基于 QoS 的 Kubernetes 混合工作负载调度系统。它旨在提高对延迟敏感的工作负载和批处理作业的运行时效率和可靠性，简化与资源相关的配置调整的复杂性，并增加 Pod 部署密度以提高资源利用率。

DCE 5.0 预置了 Koordinator v1.5.0 离线包。

本文介绍如何离线部署 Koordinator。

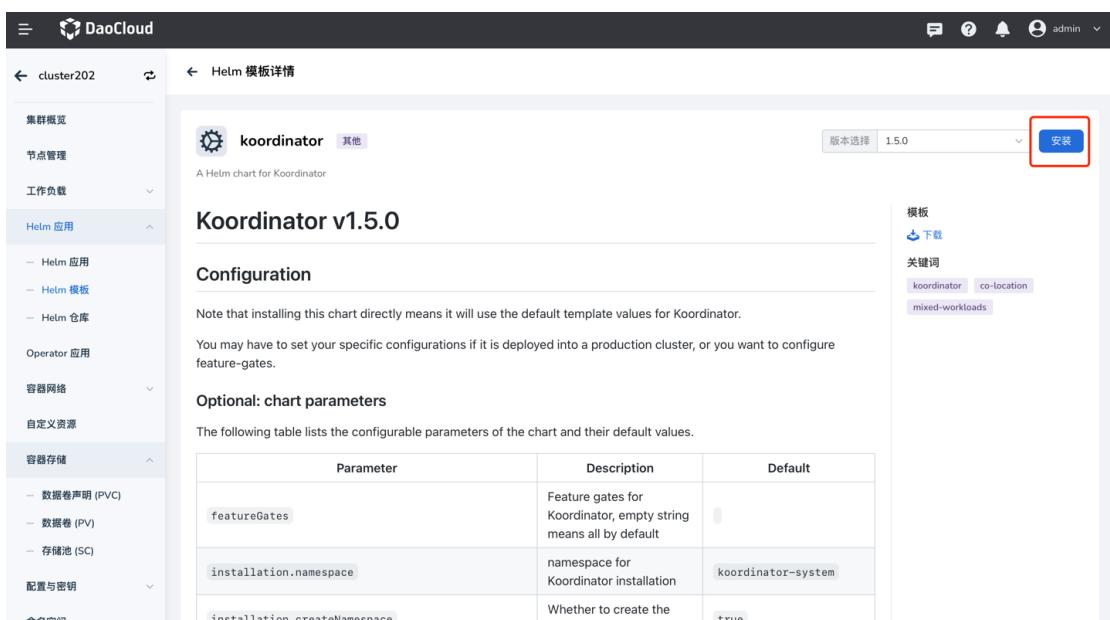
## 前提条件

1. 用户已经在平台上安装了 v0.20.0 及以上版本的 addon 离线包。
2. 待安装集群的 Kubernetes version  $\geq 1.18$ .
3. 为了最好的体验，推荐使用 linux kernel 4.19 或者更高版本。

## 操作步骤

参考如下步骤为集群安装 Koordinator 插件。

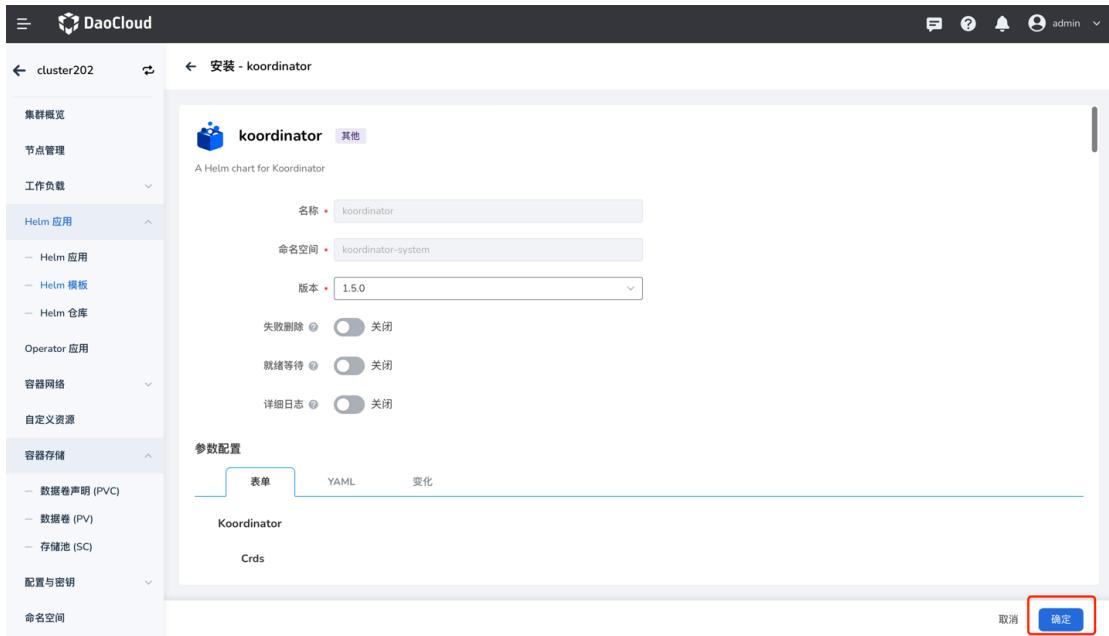
1. 登录平台，进入 容器管理 -> 待安装 Koordinator 的集群 -> 进入集群详情。
2. 在 Helm 模板 页面，选择 全部仓库，搜索 koordinator。
3. 选择 koordinator，点击 安装。



The screenshot shows the Helm chart details for Koordinator v1.5.0. On the right side, there is a large blue 'Install' button with a red rectangular border around it. The left sidebar shows navigation links like 'cluster202', 'Helm 应用', and 'koordinator'. The main content area displays the chart's configuration section, which includes notes about default values and optional parameters. A table lists parameters such as 'featureGates', 'installation.namespace', and 'installation.createNamespace' with their descriptions and defaults.

点击安装

4. 进入 koordinator 安装页面，点击 确定，使用默认配置安装 koordinator。



### 使用默认配置

## 5. 查看 koordinator-system 命名空间下的 Pod 是否正常运行

| 容器组名称                                         | 状态  | 容器 (正常/总...) | 命名空间              | 容器组 IP        | 节点               | 重启次数 | CPU 请求值/限制        | 内存请求值/限制        | GPU 配额 | 创建     |
|-----------------------------------------------|-----|--------------|-------------------|---------------|------------------|------|-------------------|-----------------|--------|--------|
| koordlet-drmvp2                               | 运行中 | 1/1          | koordinator-sy... | 10.6.202.125  | controller-no... | 0    | 不限制 / 0.2 C...    | 不限制 / 256 Mi... | -      | 202... |
| koord-descheduler-5fc9b698cc-9tck2            | 运行中 | 1/1          | koordinator-sy... | 10.233.74.112 | controller-no... | 5    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-manager-6b9f5c6bff-qr58v                | 运行中 | 1/1          | koordinator-sy... | 10.233.74.78  | controller-no... | 5    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-descheduler-5fc9b698cc-8wgfq            | 运行中 | 1/1          | koordinator-sy... | 10.233.74.116 | controller-no... | 5    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-manager-6b9f5c6bff-brdrb                | 运行中 | 1/1          | koordinator-sy... | 10.233.74.74  | controller-no... | 4    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-scheduler-7f8df5887f-6rwzb              | 运行中 | 1/1          | koordinator-sy... | 10.233.74.106 | controller-no... | 5    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-scheduler-7f8df5887f-9hor7              | 运行中 | 1/1          | koordinator-sy... | 10.233.74.69  | controller-no... | 5    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-descheduler-5fc9b698cc-dx986            | 已完成 | 0/1          | koordinator-sy... | -             | controller-no... | 0    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-descheduler-5fc9b698cc-q5k72            | 已完成 | 0/1          | koordinator-sy... | -             | controller-no... | 0    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-manager-6b9f5c6bff-dfrkr                | 已完成 | 0/1          | koordinator-sy... | -             | controller-no... | 0    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-descheduler-5fc9b698cc-vnc58            | 已完成 | 0/1          | koordinator-sy... | -             | controller-no... | 0    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-scheduler-7f8df5887f-42pf1              | 已完成 | 0/1          | koordinator-sy... | -             | controller-no... | 0    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| koord-scheduler-7f8df5887f-w64t9              | 已完成 | 0/1          | koordinator-sy... | -             | controller-no... | 0    | 0.5 Core / 1 C... | 256 Mi / 1 Gi   | -      | 202... |
| helm-operation-install-koordinator-4btc6mv... | 已完成 | 0/1          | koordinator-sy... | -             | controller-no... | 0    | 0.1 Core / 0.1... | 400 Mi / 400 Mi | -      | 202... |

### 查看 Pod

## 接入外部集群时获取永久 Token

通过 kubeconfig 可以快速将一个外部集群接入到集群管理中。为了保障接入的稳定性，

kubeconfig 内需要使用较长时效的 Token（建议使用永久 Token）。然而，不同集群服务提供商，如 AWS EKS 和 GKE，获取永久 Token 的方式不同，他们通常只提供有效期为 24 小时的 Token。

!!! note

具体的产品操作界面参考： [接入集群](./user-guide/clusters/integrate-cluster.md)

## 创建具有集群管理员权限的 ServiceAccount

为了解决上述问题，可以创建一个拥有集群管理员权限的 ServiceAccount，并使用该 ServiceAccount 的 kubeconfig 来接入集群。

!!! warning

执行以下步骤时，请确保已经配置了 AWS 或者 GCP CLI，并有权限访问该集群，否则会报错。

1. 创建 ServiceAccount 和 ClusterRoleBinding 的 YAML 配置：

```
cat >eks-admin-service-account.yaml <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
 name: eks-admin
 namespace: kube-system

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
 name: eks-admin
roleRef:
 apiGroup: rbac.authorization.k8s.io
 kind: ClusterRole
 name: cluster-admin
subjects:
- kind: ServiceAccount
 name: eks-admin
 namespace: kube-system
EOF
```

2. 使用以下命令应用配置：

```
kubectl apply -f eks-admin-service-account.yaml
```

## 为 ServiceAccount 生成 Secret

在 Kubernetes 1.24 及以上版本中，创建 ServiceAccount 默认不会创建包含 CA 证书和 user token 的 secret，需要自行关联。

1. 创建 Secret 的 YAML 配置：

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
 name: eks-admin-secret
 namespace: kube-system
 annotations:
 kubernetes.io/service-account.name: eks-admin
 type: kubernetes.io/service-account-token
EOF
```

2. 查找名为 eks-admin 的 ServiceAccount 密钥：

```
kubectl -n kube-system get secret | grep eks-admin | awk '{print $1}'
```

## 检索 ServiceAccount Token

1. 查看 eks-admin-secret 密钥的详情：

```
kubectl -n kube-system describe secret eks-admin-secret
```

现在，就可以看到 token 信息。查看是否有 exp 字段，这个字段的值就是 token 的过期时间。如果没有就是永久 token。

## 配置 kubeconfig

1. 使用获取到的 Token，设置 kubeconfig：

```
kubectl config set-credentials eks-admin --token=eyJhbGciOiJSUzI...
kubectl config set-cluster joincluster --insecure-skip-tls-verify=true --server=https://XXXXXX
XX.gr7.ap-southeast-1.eks.amazonaws.com
kubectl config set-context ekscontext --cluster=joincluster --user=eks-admin
```

2. 本地测试是否可以连接集群：

```
kubectl config use-context ekscontext
kubectl get node
```

## 导出并使用 kubeconfig

导出 kubeconfig 信息：

```
kubectl config view --minify --flatten --raw
```

复制导出的内容，添加到集群管理中，完成集群的接入。

!!! note

- 1.24 及以上的集群版本创建 ServiceAccount 默认不会创建包含 CA 证书和 user token 的 secret，需要自行关联。

参阅 K8s 官方说明：[Kubernetes ServiceAccount](<https://kubernetes.io/zh-cn/docs/tasks/configure-pod-container/configure-service-account/#manually-create-an-api-token-for-a-serviceaccount>)。

- 使用 JWT 工具解析 token 可以查看 token 的过期时间，参阅 [jwt.io](<https://jwt.io>)。